Hadoop Day 04

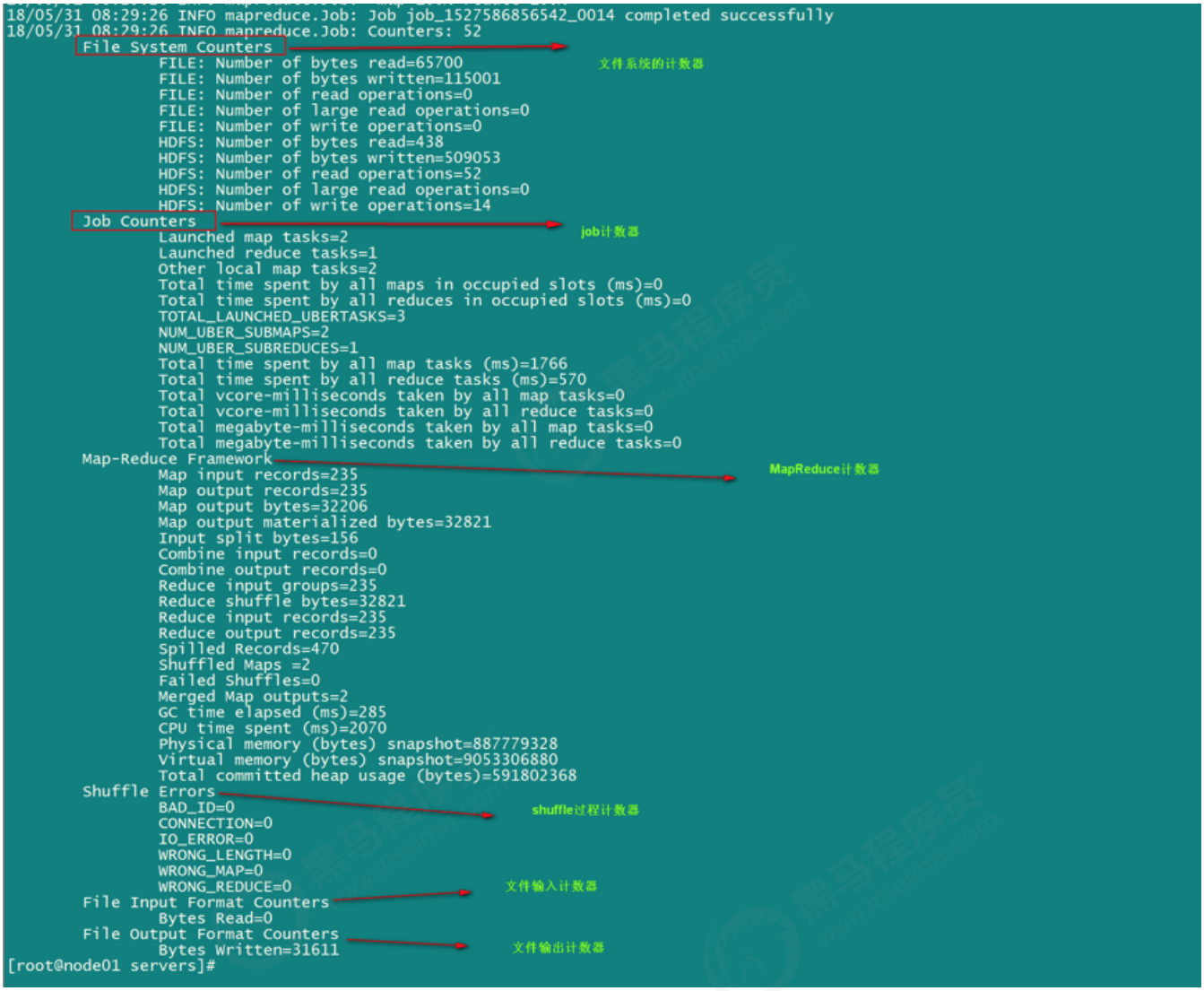
1. MapReduce 中的计数器

计数器是收集作业统计信息的有效手段之一，用于质量控制或应用级统计。计数器还可辅助诊断系统故障。如果需要将日志信息传输到 map 或 reduce 任务， 更好的方法通常是看能否用一个计数器值来记录某一特定事件的发生。对于大型分布式作业而言，使用计数器更为方便。除了因为获取计数器值比输出日志更方便，还有根据计数器值统计特定事件的发生次数要比分析一堆日志文件容易得多。

hadoop内置计数器列表

MapReduce任务计数器	**org.apache.hadoop.mapreduce.TaskCounter**
文件系统计数器	org.apache.hadoop.mapreduce.FileSystemCounter
FileInputFormat计数器	org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter
FileOutputFormat计数器	org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter
作业计数器	org.apache.hadoop.mapreduce.JobCounter

每次mapreduce执行完成之后，我们都会看到一些日志记录出来，其中最重要的一些日志记录如下截图



所有的这些都是MapReduce的计数器的功能，既然MapReduce当中有计数器的功能，我们如何实现自己的计数器？？？

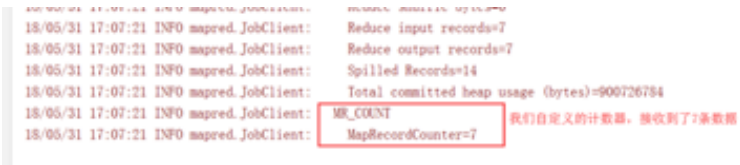
> 需求：以上面排序以及序列化为案例，统计map接收到的数据记录条数

第一种方式

第一种方式定义计数器，通过context上下文对象可以获取我们的计数器，进行记录
通过context上下文对象，在map端使用计数器进行统计

```
public class SortMapper extends
Mapper<LongWritable,Text,PairWritable,IntWritable> {

    private PairWritable mapOutKey = new PairWritable();
    private IntWritable mapOutValue = new IntWritable();

    @Override
    public  void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
//自定义我们的计数器，这里实现了统计map数据数据的条数
        Counter counter = context.getCounter("MR_COUNT",
"MapRecordCounter");
        counter.increment(1L);

        String lineValue = value.toString();
        String[] strs = lineValue.split("\t");

        //设置组合key和value ==> <(key,value),value>
        mapOutKey.set(strs[0], Integer.valueOf(strs[1]));
        mapOutValue.set(Integer.valueOf(strs[1]));
        context.write(mapOutKey, mapOutValue);
    }
}
```

运行程序之后就可以看到我们自定义的计数器在map阶段读取了七条数据



第二种方式

通过enum枚举类型来定义计数器
统计reduce端数据的输入的key有多少个，对应的value有多少个

```
public class SortReducer extends
Reducer<PairWritable,IntWritable,Text,IntWritable> {

    private Text outPutKey = new Text();
    public static enum Counter{
        REDUCE_INPUT_RECORDS, REDUCE_INPUT_VAL_NUMS,
    }
    @Override
    public void reduce(PairWritable key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
        context.getCounter(Counter.REDUCE_INPUT_RECORDS).increment(1L);
        //迭代输出
        for(IntWritable value : values) {

context.getCounter(Counter.REDUCE_INPUT_VAL_NUMS).increment(1L);
            outPutKey.set(key.getFirst());
            context.write(outPutKey, value);
        }
    }
}
```

```
18/07/19 14:36:02 INFO mapred.JobClient:     REDUCE_INPUT_RECORDS=7
18/07/19 14:36:02 INFO mapred.JobClient:     REDUCE_INPUT_VAL_NUMS=8
```

2. 规约Combiner

每一个 map 都可能会产生大量的本地输出，Combiner 的作用就是对 map 端的输出先做一次合并，以减少在 map 和 reduce 节点之间的数据传输量，以提高网络IO 性能，是 MapReduce 的一种优化手段之一

- combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件
- combiner 组件的父类就是 Reducer
- combiner 和 reducer 的区别在于运行的位置
    - Combiner 是在每一个 maptask 所在的节点运行
    - Reducer 是接收全局所有 Mapper 的输出结果
- combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量

实现步骤

1. 自定义一个 combiner 继承 Reducer，重写 reduce 方法
2. 在 job 中设置 `job.setCombinerClass(CustomCombiner.class)`

combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combiner 的输出 kv 应该跟 reducer 的输入 kv 类型要对应起来

3. 流量统计

需求一: 统计求和

统计每个手机号的上行流量总和，下行流量总和，上行总流量之和，下行总流量之和
分析：以手机号码作为key值，上行流量，下行流量，上行总流量，下行总流量四个字段作为value值，然后以这个key，和value作为map阶段的输出，reduce阶段的输入

Step 1: 自定义**map**的输出**value**对象**FlowBean**

```
public class FlowBean implements Writable {
    private Integer upFlow;
    private Integer  downFlow;
    private Integer upCountFlow;
    private Integer downCountFlow;
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeInt(upFlow);
        out.writeInt(downFlow);
        out.writeInt(upCountFlow);
        out.writeInt(downCountFlow);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        this.upFlow = in.readInt();
        this.downFlow = in.readInt();
        this.upCountFlow = in.readInt();
        this.downCountFlow = in.readInt();
    }
    public FlowBean() {
    }
    public FlowBean(Integer upFlow, Integer downFlow, Integer upCountFlow,
Integer downCountFlow) {
        this.upFlow = upFlow;
        this.downFlow = downFlow;
        this.upCountFlow = upCountFlow;
        this.downCountFlow = downCountFlow;
    }
    public Integer getUpFlow() {
        return upFlow;
    }
    public void setUpFlow(Integer upFlow) {
        this.upFlow = upFlow;
    }
    public Integer getDownFlow() {
        return downFlow;
    }
    public void setDownFlow(Integer downFlow) {
        this.downFlow = downFlow;
    }

    public Integer getUpCountFlow() {
```

```
        return upCountFlow;
    }
    public void setUpCountFlow(Integer upCountFlow) {
        this.upCountFlow = upCountFlow;
    }
    public Integer getDownCountFlow() {
        return downCountFlow;
    }
    public void setDownCountFlow(Integer downCountFlow) {
        this.downCountFlow = downCountFlow;
    }
    @Override
    public String toString() {
        return "FlowBean{" +
                "upFlow=" + upFlow +
                ", downFlow=" + downFlow +
                ", upCountFlow=" + upCountFlow +
                ", downCountFlow=" + downCountFlow +
                '}';
    }
}
```

Step 2: 定义**FlowMapper**类

```
public class FlowCountMapper extends
Mapper<LongWritable,Text,Text,FlowBean> {
    @Override
    protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
        //1:拆分手机号
         String[] split = value.toString().split("\t");
         String phoneNum = split[1];
         //2:获取四个流量字段
         FlowBean flowBean = new FlowBean();
         flowBean.setUpFlow(Integer.parseInt(split[6]));
         flowBean.setDownFlow(Integer.parseInt(split[7]));
         flowBean.setUpCountFlow(Integer.parseInt(split[8]));
         flowBean.setDownCountFlow(Integer.parseInt(split[9]));

         //3:将k2和v2写入上下文中
         context.write(new Text(phoneNum), flowBean);
    }
}
```

Step 3: 定义**FlowReducer**类

```
public class FlowCountReducer extends Reducer<Text,FlowBean,Text,FlowBean>
{
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Context
context) throws IOException, InterruptedException {
       //封装新的FlowBean
        FlowBean flowBean = new FlowBean();
        Integer upFlow = 0;
        Integer  downFlow = 0;
        Integer upCountFlow = 0;
        Integer downCountFlow = 0;
        for (FlowBean value : values) {
            upFlow  += value.getUpFlow();
            downFlow += value.getDownFlow();
            upCountFlow += value.getUpCountFlow();
            downCountFlow += value.getDownCountFlow();
        }
        flowBean.setUpFlow(upFlow);
        flowBean.setDownFlow(downFlow);
        flowBean.setUpCountFlow(upCountFlow);
        flowBean.setDownCountFlow(downCountFlow);
        //将K3和V3写入上下文中
        context.write(key, flowBean);
    }
}
```

Step 4: 程序**main**函数入口**FlowMain**

```
public class JobMain extends Configured  implements Tool {
    @Override
    public int run(String[] strings) throws Exception {
        //创建一个任务对象
        Job job = Job.getInstance(super.getConf(), "mapreduce_flowcount");

        //打包放在集群运行时，需要做一个配置
        job.setJarByClass(JobMain.class);
        //第一步:设置读取文件的类: K1 和V1
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job, new
Path("hdfs://node01:8020/input/flowcount"));

        //第二步: 设置Mapper类
        job.setMapperClass(FlowCountMapper.class);
        //设置Map阶段的输出类型: k2 和V2的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        //第三,四，五，六步采用默认方式(分区，排序，规约，分组)


        //第七步 ： 设置文的Reducer类
        job.setReducerClass(FlowCountReducer.class);
        //设置Reduce阶段的输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        //设置Reduce的个数

        //第八步:设置输出类
        job.setOutputFormatClass(TextOutputFormat.class);
        //设置输出的路径
        TextOutputFormat.setOutputPath(job, new
Path("hdfs://node01:8020/out/flowcount_out"));


        boolean b = job.waitForCompletion(true);
        return b?0:1;


    }
```

```
    public static void main(String[] args) throws Exception {
        Configuration configuration = new Configuration();

        //启动一个任务
        int run = ToolRunner.run(configuration, new JobMain(), args);
        System.exit(run);
    }

}
```

需求二: 上行流量倒序排序（递减排序）

分析，以需求一的输出数据作为排序的输入数据，自定义FlowBean,以FlowBean为map输出的key，以手机号作为Map输出的value，因为MapReduce程序会对Map阶段输出的key进行排序

Step 1: 定义**FlowBean**实现**WritableComparable**实现比较排序

Java 的 compareTo 方法说明:

- compareTo 方法用于将当前对象与方法的参数进行比较。
- 如果指定的数与参数相等返回 0。
- 如果指定的数小于参数返回 -1。
- 如果指定的数大于参数返回 1。

例如：`o1.compareTo(o2);` 返回正数的话，当前对象（调用 compareTo 方法的对象 o1）要排在比较对象（compareTo 传参对象 o2）后面，返回负数的话，放在前面

public class FlowBean implements WritableComparable {

```
private Integer upFlow;
private Integer  downFlow;
private Integer upCountFlow;
private Integer downCountFlow;
public FlowBean() {
}
public FlowBean(Integer upFlow, Integer downFlow, Integer upCountFlow,
Integer downCountFlow) {
    this.upFlow = upFlow;
    this.downFlow = downFlow;
    this.upCountFlow = upCountFlow;
    this.downCountFlow = downCountFlow;
}
@Override
public void write(DataOutput out) throws IOException {
    out.writeInt(upFlow);
    out.writeInt(downFlow);
    out.writeInt(upCountFlow);
    out.writeInt(downCountFlow);
}
@Override
public void readFields(DataInput in) throws IOException {
    upFlow = in.readInt();
    downFlow = in.readInt();
    upCountFlow = in.readInt();
    downCountFlow = in.readInt();
}
public Integer getUpFlow() {
    return upFlow;
}
public void setUpFlow(Integer upFlow) {
    this.upFlow = upFlow;
}
public Integer getDownFlow() {
    return downFlow;
}
public void setDownFlow(Integer downFlow) {
    this.downFlow = downFlow;
}
public Integer getUpCountFlow() {
    return upCountFlow;
```

```
    }
    public void setUpCountFlow(Integer upCountFlow) {
        this.upCountFlow = upCountFlow;
    }
    public Integer getDownCountFlow() {
        return downCountFlow;
    }
    public void setDownCountFlow(Integer downCountFlow) {
        this.downCountFlow = downCountFlow;
    }
    @Override
    public String toString() {
        return upFlow+"\t"+downFlow+"\t"+upCountFlow+"\t"+downCountFlow;
    }
    @Override
    public int compareTo(FlowBean o) {
        return this.upCountFlow > o.upCountFlow ?-1:1;
    }
```

}

Step 2: 定义**FlowMapper**

```
public class FlowCountSortMapper extends
Mapper<LongWritable,Text,FlowBean,Text> {
    @Override
    protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
        FlowBean flowBean = new FlowBean();
        String[] split = value.toString().split("\t");

        //获取手机号，作为V2
        String phoneNum = split[0];
        //获取其他流量字段,封装flowBean，作为K2
        flowBean.setUpFlow(Integer.parseInt(split[1]));
        flowBean.setDownFlow(Integer.parseInt(split[2]));
        flowBean.setUpCountFlow(Integer.parseInt(split[3]));
        flowBean.setDownCountFlow(Integer.parseInt(split[4]));

        //将K2和V2写入上下文中
        context.write(flowBean, new Text(phoneNum));

    }
}
```

Step 3: 定义**FlowReducer**

```
public class FlowCountSortReducer extends
Reducer<FlowBean,Text,Text,FlowBean> {
    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Context
context) throws IOException, InterruptedException {
        for (Text value : values) {
            context.write(value, key);
        }
    }
}
```

Step 4: 程序**main**函数入口

```
public class JobMain extends Configured  implements Tool {
    @Override
    public int run(String[] strings) throws Exception {
        //创建一个任务对象
        Job job = Job.getInstance(super.getConf(),
"mapreduce_flowcountsort");

        //打包放在集群运行时，需要做一个配置
        job.setJarByClass(JobMain.class);
        //第一步:设置读取文件的类: K1 和V1
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job, new
Path("hdfs://node01:8020/out/flowcount_out"));

        //第二步: 设置Mapper类
        job.setMapperClass(FlowCountSortMapper.class);
        //设置Map阶段的输出类型: k2 和V2的类型
        job.setMapOutputKeyClass(FlowBean.class);
        job.setMapOutputValueClass(Text.class);

        //第三,四，五，六步采用默认方式(分区，排序，规约，分组)


        //第七步 ： 设置文的Reducer类
        job.setReducerClass(FlowCountSortReducer.class);
        //设置Reduce阶段的输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        //设置Reduce的个数

        //第八步:设置输出类
        job.setOutputFormatClass(TextOutputFormat.class);
        //设置输出的路径
        TextOutputFormat.setOutputPath(job, new
Path("hdfs://node01:8020/out/flowcountsort_out"));


        boolean b = job.waitForCompletion(true);
        return b?0:1;
```

```
    }
    public static void main(String[] args) throws Exception {
        Configuration configuration = new Configuration();

        //启动一个任务
        int run = ToolRunner.run(configuration, new JobMain(), args);
        System.exit(run);
    }

}
```

需求三: 手机号码分区

在需求一的基础上，继续完善，将不同的手机号分到不同的数据文件的当中去，需要自定义分区来实现，这里我们自定义来模拟分区，将以下数字开头的手机号进行分开

```
135 开头数据到一个分区文件
136 开头数据到一个分区文件
137 开头数据到一个分区文件
其他分区
```

自定义分区

```
public class FlowPartition extends Partitioner<Text,FlowBean> {
    @Override
    public int getPartition(Text text, FlowBean flowBean, int i) {
        String line = text.toString();
        if (line.startsWith("135")){
            return 0;
        }else if(line.startsWith("136")){
            return 1;
        }else if(line.startsWith("137")){
            return 2;
        }else{
            return 3;
        }
    }
}
```

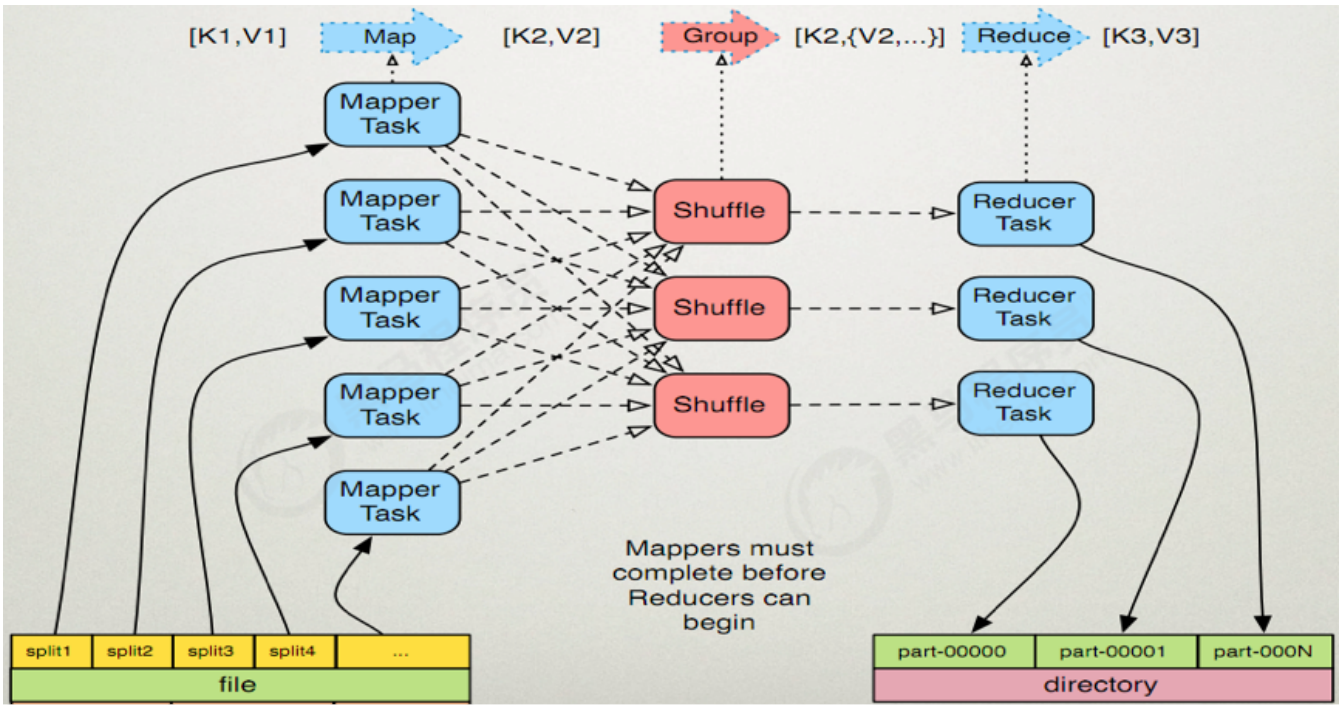
作业运行添加分区设置

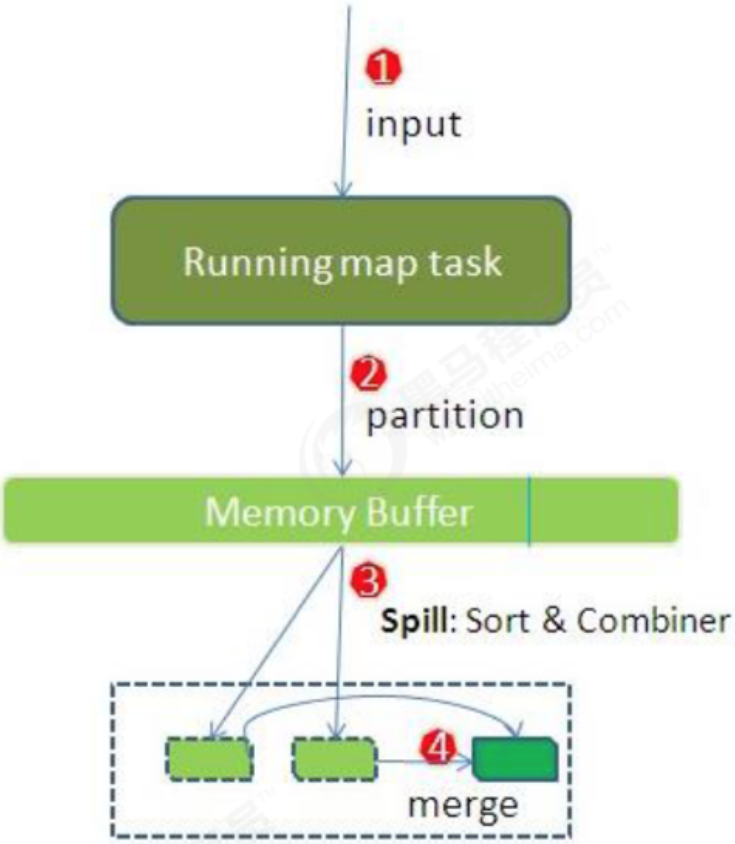
```
job.setPartitionerClass(FlowPartition.class);
```

修改输入输出路径，并放入集群运行

```
TextInputFormat.addInputPath(job,new
Path("hdfs://node01:8020/partition_flow/"));
TextOutputFormat.setOutputPath(job,new
Path("hdfs://node01:8020/partition_out"));
```

4. MapTask 运行机制





Picture2.png

整个Map阶段流程大体如上图所示。

简单概述：inputFile通过split被逻辑切分为多个split文件，通过Record按行读取内容给map（用户自己实现的）进行处理，数据被map处理结束之后交给OutputCollector收集器，对其结果key进行分区（默认使用hash分区），然后写入buffer，每个map task都有一个内存缓冲区，存储着map的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个map task结束后再对磁盘中这个map task产生的所有临时文件做合并，生成最终的正式输出文件，然后等待reduce task来拉数据

详细步骤

1. 读取数据组件 **InputFormat** (默认 TextInputFormat) 会通过 `getSplits` 方法对输入目录中文件进行逻辑切片规划得到 splits, 有多少个 `split` 就对应启动多少个 `MapTask`. `split` 与 `block` 的对应关系默认是一对一
2. 将输入文件切分为 `splits` 之后, 由 `RecordReader` 对象 (默认是LineRecordReader) 进行读取, 以 `\n` 作为分隔符, 读取一行数据, 返回 `<key，value>`. Key 表示每行首字符偏移值, Value 表示这一行文本内容

3. 读取 `split` 返回 `<key,value>`, 进入用户自己继承的 **Mapper** 类中，执行用户重写的 map 函数, RecordReader 读取一行这里调用一次
4. Mapper 逻辑结束之后, 将 Mapper 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 `Reducer` 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 Reducer 数量取模. 默认的取模方式只是为了平均 Reducer 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
5. 接下来, 会将数据写入内存, 内存中这片区域叫做环形缓冲区, 缓冲区的作用是批量收集 Mapper 结果, 减少磁盘 IO 的影响. 我们的 **Key/Value** 对以及 **Partition** 的结果都会被写入缓冲区. 当然, 写入之前，Key 与 Value 值都会被序列化成字节数组
    - > 环形缓冲区其实是一个数组, 数组中存放着 Key, Value 的序列化数据和 Key, Value 的元数据信息, 包括 Partition, Key 的起始位置, Value 的起始位置以及 Value 的长度. 环形结构是一个抽象概念
    - > 缓冲区是有大小限制, 默认是 100MB. 当 Mapper 的输出结果很多时, 就可能会撑爆内存, 所以需要在一定条件下将缓冲区中的数据临时写入磁盘, 然后重新利用这块缓冲区. 这个从内存往磁盘写数据的过程被称为 Spill, 中文可译为溢写. 这个溢写是由单独线程来完成, 不影响往缓冲区写 Mapper 结果的线程. 溢写线程启动时不应该阻止 Mapper 的结果输出, 所以整个缓冲区有个溢写的比例 `spill.percent`. 这个比例默认是 0.8, 也就是当缓冲区的数据已经达到阈值 `buffer size * spill percent = 100MB * 0.8 = 80MB`, 溢写线程启动, 锁定这 80MB 的内存, 执行溢写过程. Mapper 的输出结果还可以往剩下的 20MB 内存中写, 互不影响
6. 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
    - > 那哪些场景才能使用 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. Combiner 的使用一定得慎重, 如果用

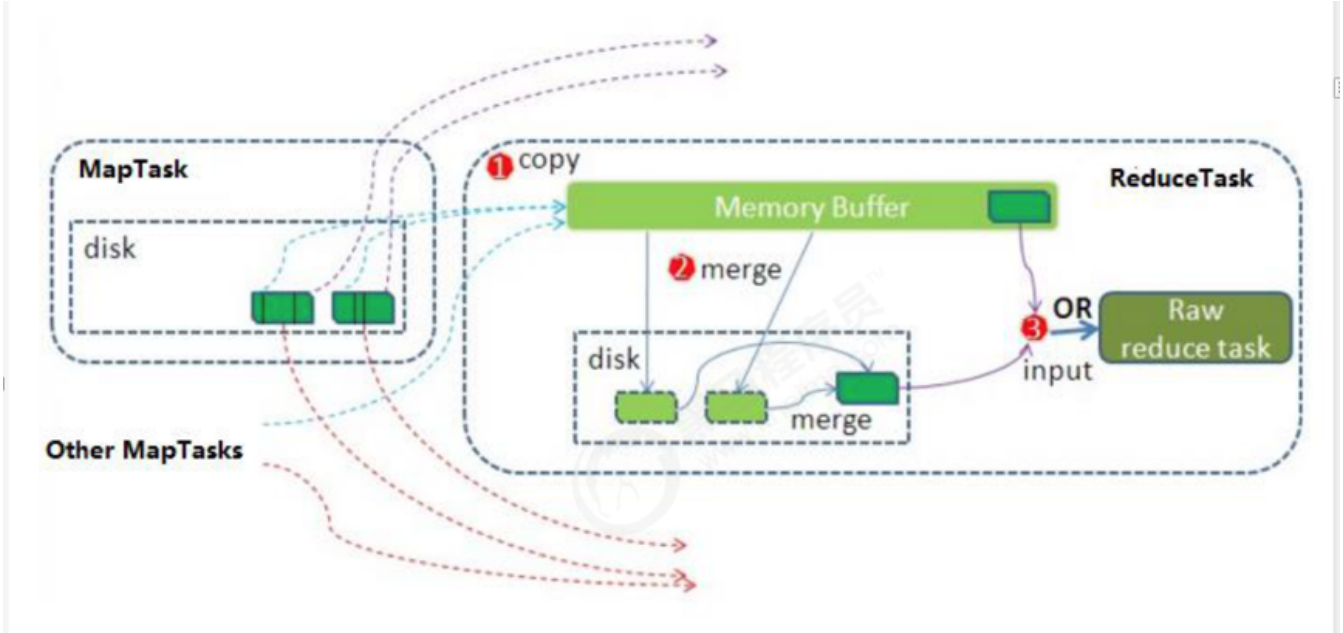
> 好, 它对 Job 执行效率有帮助, 反之会影响 Reducer 的最终结果

7. 合并溢写文件, 每次溢写会在磁盘上生成一个临时文件 (写之前判断是否有 Combiner), 如果 Mapper 的输出结果真的很大, 有多次这样的溢写发生, 磁盘上相应的就会有多个临时文件存在. 当整个数据处理结束之后开始对磁盘中的临时文件进行 Merge 合并, 因为最终的文件只有一个, 写入磁盘, 并且为这个文件提供了一个索引文件, 以记录每个reduce对应数据的偏移量

配置

配置	默认值	解释
`mapreduce.task.io.sort.mb`	100	设置环型缓冲区的内存值大小
`mapreduce.map.sort.spill.percent`	0.8	设置溢写的比例
`mapreduce.cluster.local.dir`	`${hadoop.tmp.dir}/mapred/local`	溢写数据目录
`mapreduce.task.io.sort.factor`	10	设置一次合并多少个溢写文件

5. ReduceTask 工作机制和 ReduceTask 并行度



Picture3.png

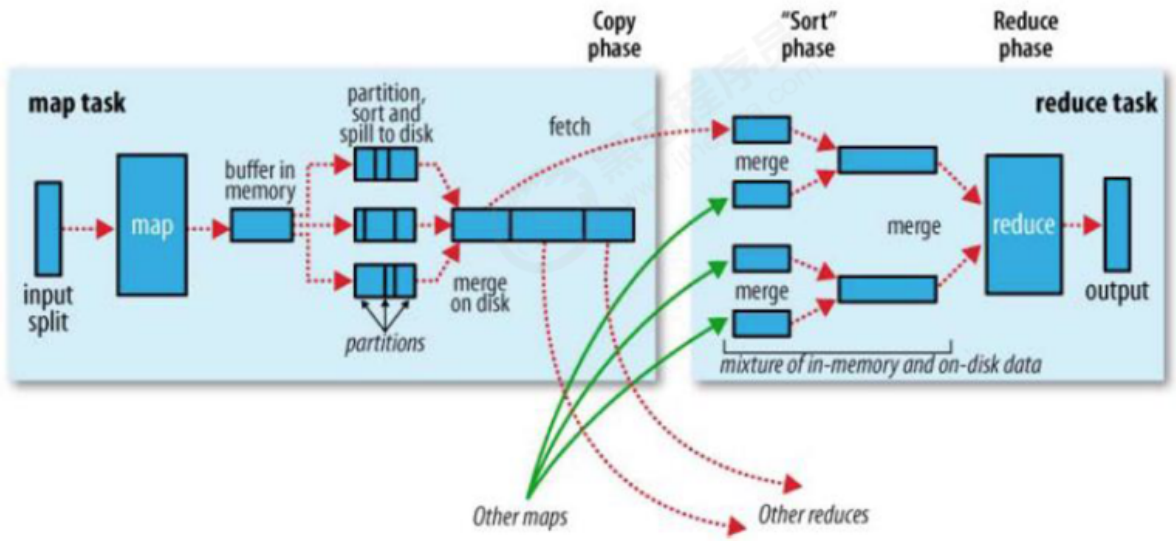
Reduce 大致分为 copy、sort、reduce 三个阶段，重点在前两个阶段。copy 阶段包含一个 eventFetcher 来获取已完成的 map 列表，由 Fetcher 线程去 copy 数据，在此过程中会启动两个 merge 线程，分别为 inMemoryMerger 和 onDiskMerger，分别将内存中的数据 merge 到磁盘和将磁盘中的数据进行 merge。待数据 copy 完成之后，copy 阶段就完成了，开始进行 sort 阶段，sort 阶段主要是执行 finalMerge 操作，纯粹的 sort 阶段，完成之后就是 reduce 阶段，调用用户定义的 reduce 函数进行处理

详细步骤

1. `Copy阶段`，简单地拉取数据。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`。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. `合并排序`。把分散的数据合并成一个大的数据后，还会再对合并后的数据排序。
4. `对排序后的键值对调用reduce方法`，键相等的键值对调用一次reduce方法，每次调用会产生零个或者多个键值对，最后把这些输出的键值对写入到HDFS文件中。

6. Shuffle 过程

map 阶段处理的数据如何传递给 reduce 阶段，是 MapReduce 框架中最关键的一个流程，这个流程就叫 shuffle
shuffle: 洗牌、发牌 ——（核心机制：数据分区，排序，分组，规约，合并等过程）



Picture4.png

shuffle 是 Mapreduce 的核心，它分布在 Mapreduce 的 map 阶段和 reduce 阶段。一般把从 Map 产生输出开始到 Reduce 取得数据作为输入之前的过程称作 shuffle。

1. `Collect`阶段：将 MapTask 的结果输出到默认大小为 100M 的环形缓冲区，保存的是 key/value，Partition 分区信息等。
2. `Spill`阶段：当内存中的数据量达到一定的阀值的时候，就会将数据写入本地磁盘，在将数据写入磁盘之前需要对数据进行一次排序的操作，如果配置了 combiner，还会将有相同分区号和 key 的数据进行排序。
3. `Merge`阶段：把所有溢出的临时文件进行一次合并操作，以确保一个 MapTask 最终只产生一个中间数据文件。
4. `Copy`阶段：ReduceTask 启动 Fetcher 线程到已经完成 MapTask 的节点上复制一份属于自己的数据，这些数据默认会保存在内存的缓冲区中，当内存的缓冲区达到一定的阀值的时候，就会将数据写到磁盘之上。
5. `Merge`阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线程对内存到本地的数据文件进行合并操作。
6. `Sort`阶段：在对数据进行合并的同时，会进行排序操作，由于 MapTask 阶段已经对数据进行了局部的排序，ReduceTask 只需保证 Copy 的数据的最终整体有效性即可。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：mapreduce.task.io.sort.mb 默认100M

7. [案例] Reduce 端实现 JOIN

7.1. 需求

> 假如数据量巨大，两表的数据是以文件的形式存储在 HDFS 中, 需要用 MapReduce 程序来实现以下 SQL 查询运算
>
> ```
> select  a.id,a.date,b.name,b.category_id,b.price from t_order a left join t_product b on a.pid = b.id
> ```

订单数据表

id	date	pid	amount
1001	20150710	P0001	2
1002	20150710	P0001	3
1002	20150710	P0002	3

商品信息表

id	pname	category_id	price
P0001	小米5	1000	2000
P0002	锤子T1	1000	3000

实现机制

通过将关联的条件作为map输出的key，将两表满足join条件的数据并携带数据所来源的文件信息，发往同一个reduce task，在reduce中进行数据的串联

7.2. Step 1: 定义 Mapper

```
public class ReduceJoinMapper extends Mapper<LongWritable,Text,Text,Text> {
    @Override
    protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
        //首先判断数据来自哪个文件
         FileSplit fileSplit = (FileSplit) context.getInputSplit();
         String fileName = fileSplit.getPath().getName();

         if(fileName.equals("orders.txt")){
         //获取pid
             String[] split = value.toString().split(",");
             context.write(new Text(split[2]), value);
         }else{
             //获取pid
             String[] split = value.toString().split(",");
             context.write(new Text(split[0]), value);
         }
    }
}
```

7.3. Step 2: 定义 Reducer

```
public class ReduceJoinReducer extends Reducer<Text,Text,Text,Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {

        String first = "";
        String second = "";
        for (Text value : values) {
            if(value.toString().startsWith("p")){
                first = value.toString();
            }else{
                second  = value.toString();
            }
        }

        if(first.equals("")){
            context.write(key, new Text("NULL"+"\t"+second));
        }else{
            context.write(key, new Text(first+"\t"+second));
        }

    }
}
```

7.5. Step 4: Main 方法

```
public class JobMain extends Configured implements Tool {
    @Override
    public int run(String[] strings) throws Exception {
        //创建一个任务对象
        Job job = Job.getInstance(super.getConf(),
"mapreduce_reduce_join");

        //打包放在集群运行时，需要做一个配置
        job.setJarByClass(JobMain.class);
        //第一步:设置读取文件的类: K1 和V1
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job, new
Path("hdfs://node01:8020/input/reduce_join"));

        //第二步：设置Mapper类
        job.setMapperClass(ReduceJoinMapper.class);
        //设置Map阶段的输出类型: k2 和V2的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        //第三,四，五，六步采用默认方式(分区，排序，规约，分组)


        //第七步 ：设置文的Reducer类
        job.setReducerClass(ReduceJoinReducer.class);
        //设置Reduce阶段的输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        //第八步:设置输出类
        job.setOutputFormatClass(TextOutputFormat.class);
        //设置输出的路径
        TextOutputFormat.setOutputPath(job, new
Path("hdfs://node01:8020/out/reduce_join_out"));


        boolean b = job.waitForCompletion(true);
        return b?0:1;

    }

    public static void main(String[] args) throws Exception {
```

```
        Configuration configuration = new Configuration();
        //启动一个任务
        int run = ToolRunner.run(configuration, new JobMain(), args);
        System.exit(run);
    }

}
```