day04 MapReduce

一、内容回顾

二、教学目标

2.1. 教学重点

- MapReduce的简介及核心思想
- MapReduce的编程规范
- yarn的设计思想
- 案例:wordcount
- 分片机制
- MapTask的执行流程
- ReduceTask的执行流程
- shuffle流程
- 自定义类型
- Partitioner组件

2.2. 教学难点

- 分片机制
- 自定义类型
- MapTask的执行流程
- ReduceTask的执行流程

三、教学导读

我们已经学习过了HDFS，大体量的数据存储于HDFS上。而HDFS是一个分布式的文件系统，存储于HDFS的文件将被分为不同的数据块。那么我们怎么对HDFS的数据进行计算呢？我们以Hadoop中非常经典的WordCount案例为例。所谓的WordCount，指的是统计指定的文件中的每一个单词出现的次数。

如果我们要统计的文件在一个机器上，我们可以使用Java程序快速的实现这个效果，但是现在我们要处理的数据需要分布在多台机器上。那么，这个问题处理起来就非常复杂了，因为这是一个很典型的分布式的应用程序。

我们可以将所有的数据块都移动到一个节点上，然后在同一个节点进行单词的统计。而这就是非常典型的**移动数据**，这个效率是非常低下的！我们要处理的数据量一般都比较大，那么在节点之间进行拷贝的时候就会消耗大量的时间，而且还有可能出现一台设备存储不了这样的数据的情况。况且，此时集群中的其他的节点的计算资源都是闲置的，浪费资源。因此，这种解决方案行不通！

那我们换一个思路，我们将统计一个节点的单词数量的程序分发给每一个数据节点，然后在不同的节点上进行单词的词频统计。计算程序一般体积都非常的小，我们可以快速的实现计算程序在不同的节点之间进行分发，同时也可以利用到每一个节点的计算资源，充分的利用了集群。但是，在这个过程中我们需要处理的问题是:

> 1. 如何将计算程序分发给不同的节点
> 2. 如何为每一个计算任务分配资源
> 3. 如何将每一个节点的计算结果汇总在一起
> 4. 如何进行计算过程中的监控
> 5. 如果某一个节点出故障了，如何将计算程序再分发到其他的节点继续计算
> 6. ...

而这些问题正是分布式计算所需要考虑的问题。怎么解决呢？ MapReduce就来了！

MapReduce是一个分布式计算框架，使得开发人员在不了解分布式计算的细节的情况下，对分布式文件系统的数据进行计算。开发人员不需要关注上述的分布式计算的细节，把重心放在计算逻辑上即可。大大的简化了分布式计算给开发人员带来的压力！

四、教学内容

4.1. MapReduce入门

4.1.1. MapReduce是什么【了解】

2004年，谷歌发表了一篇名为《MapReduce》的论文，主要介绍了如何在分布式的存储系统上对数据进行高效率的计算。2005年，Nutch团队使用Java语言实现了这个技术，并命名为MapReduce。时至今日，MapReduce是Apache Hadoop的核心模块之一，是运行在HDFS上的分布式运算程序的编程框架，用于大规模数据集（大于1TB）的并行运算。其中的概念，"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。

> 《阅读资料》
>
> - 在过去的很长一段时间里，CPU的性能都会遵循"摩尔定律"，在性能上每隔18个月左右就是提高一倍。那个时候，不需要对程序做任何改变，仅仅通过使用更优秀的CPU，就可以进行性能提升。但是现在，在CPU性能提升的道路上，人类已经到达了制作工艺的瓶颈，因此，我们不能再把希望寄托在性能更高的CPU身上了。
> - 现在这个时候，大规模数据存储在分布式文件系统上，人们也开始采用分布式并行编程来提高程序的性能。分布式程序运行在大规模计算机集群上，集群是大量的廉价服务器，可以并行执行大规模数据处理任务，这样就获得了海量的计算能力
> - 分布式并行编程比传统的程序有明显的区别，它运行在大量计算机构成的集群上，可以充分利用集群的并行处理能力；同时，通过向集群中增加新的计算节点，就可以很容易的实现集群计算能力的扩展。

4.1.2. 为什么要学习MapReduce【了解】

MapReduce主要解决的是分布式文件存储系统上，数据的分布式计算的问题。在上述导读部分我们介绍过一个WordCount的案例，就是一个非常典型的分布式计算的案例。如果我们将所有的需要处理的数据移动到一个节点上进行处理，那么只是在数据传输的过程中就得消耗大量的时间，而且还可能在一台节点存不下这大量的数据。就算是能够存储下，也能够接受数据移动所带来的时间消耗，集群中其他节点的计算资源也都是在闲置的，不能高效率的利用集群。

因此我们就需要进行分布式的计算，将计算程序分发给不同的节点。在每一个节点上处理自己节点的数据，最后将每一个节点的数据处理结果汇总在一起。而在分布式计算的过程中会遇到很多的分布式计算的细节问题，这些问题都是需要开发人员去考虑的。那么如何去解决这些问题呢?

MapReduce是一个开源的、分布式的计算框架，封装了分布式计算程序的实现细节，使得开发人员不需要了解分布式计算底层实现的情况下，就可以去开发一个分布式的计算程序。开发人员只需要将重心放在业务逻辑的实现即可，不需要关注分布式开发的底层细节。因此，对于开发人员来说，可以简化不少的工作量，提交程序开发的效率!

4.1.3. MapReduce的优缺点【了解】

4.1.3.1. 优点

- 易于编程

  它简单的实现一些接口，就可以完成一个分布式程序，这个程序可以分布到大量的廉价的pc机器上运行。也就是说你写一个分布式程序，跟写一个简单的串行程序是一模一样的。就是因为这个特性使的MapReduce编程变得非常流行。

- 良好的扩展性

  当你的计算资源得不到满足的时候，你可以简单的通过增加机器来扩展它的计算能力

- 高容错性

  MapReduce的设计初衷就是使程序能够部署在廉价的pc机器上，这就要求它具有很高的容错性。比如一个机器挂了，它可以把上面的计算任务转移到另一个节点上运行，不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由hadoop内部完成的。

- 适合PB级以上海量数据的离线处理

4.1.3.2. 缺点

- 不适合做实时计算

  MapReduce无法做到像Mysql那样做到毫秒或者秒级的返回结果

- 不适合做流式计算

  流式计算的输入数据是动态的，而MapReduce的输入数据集是静态的，不能流态变化。这是MR自身的设计特点决定了数据源必须是静态的。

- 不适合DAG(有向图)计算

  多个应用程序存在依赖关系，后一个应用程序的输入为前一个应用程序的输出，在这种情况下，MapReduce并不是不能做，而是使用后每个MapReduce作业的输出结果都会写入到磁盘，会造成大量的磁盘IO，导致性能非常低下。

4.1.4. MapReduce的核心思想【掌握】

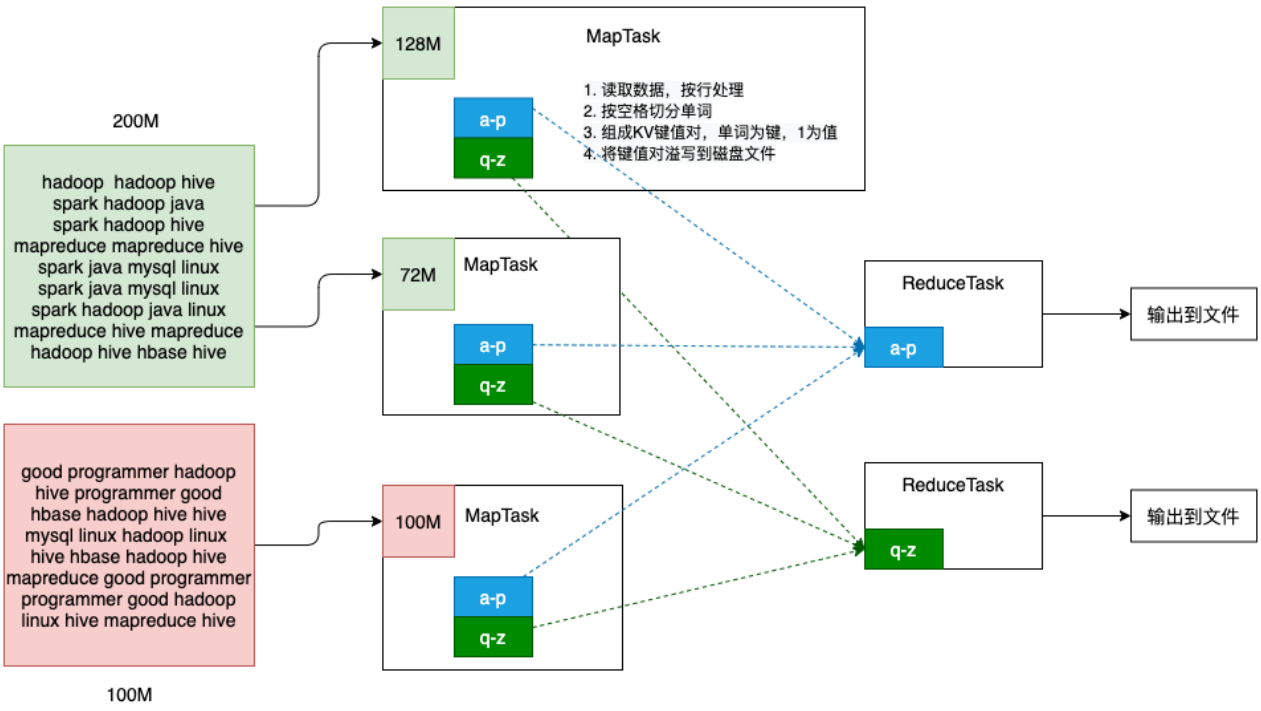
1. MapReduce设计的一个理念是"计算向数据靠拢"（移动计算），而不是"数据向计算靠拢"（移动数据）
2. 将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，移动到有数据存储的集群节点上，一是可以减少节点间的数据移动开销。二是在存储节点上可以并行计算，大大提高计算效率问题。

   > 因为移动数据需要大量的网络传输开销，尤其是在大规模数据环境下，这种开销尤为惊人，所以移动计算要比移动数据更加经济。

3. MapReduce一个完整的运算分为Map和Reduce两个部分。Map会处理本节点的原始数据，产生的数据会临时存储到本地磁盘。Reduce会跨节点fetch属于自己的数据，并进行处理，产生的数据会存储到HDFS上。

> 《阅读资料》
>
> Hadoop的MapReduce核心技术起源于谷歌在2004年发表的关于MapReduce系统的论文介绍。论文中有这么一句话：Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages。这句话提到了MapReduce思想来源，大致意思是，MapReduce的灵感来源于函数式语言（比如Lisp）中的内置函数map（映射）和reduce（规约）。
>
> 简单来说，在函数式语言里，map表示对一个列表（List）中的每个元素做计算，reduce表示对一个列表中的每个元素做迭代计算。它们具体的计算是通过传入的函数来实现的，map和reduce提供的是计算的框架。我们想一下，reduce既然能做迭代计算，那就表示列表中的元素是相关的（比如我想对列表中的所有元素做相加求和，那么列表中至少都应该是数值吧）。而map是对列表中每个元素做单独处理的，这表示列表中可以是杂乱无章的数据。这样看来，就有点联系了。在MapReduce里，Map处理的是原始数据，自然是杂乱无章的，每条数据之间互相没有关系；到了Reduce阶段，数据是以key后面跟着若干个value来组织的，这些value有相关性，至少它们都在一个key下面，于是就符合函数式语言里map和reduce的基本思想了。

需求: 统计每一个单词出现的次数，结果a-p开头的单词存储一个文件、q-z开头的单词存储一个文件。



4.1.5. MapReduce的阶段分类【掌握】

MapReduce的程序在运行的过程中，一般分为两个阶段: Map阶段和Reduce阶段

4.1.5.1. 第一阶段: Map

第一阶段，也称之为Map阶段。这个阶段会有若干个MapTask实例，完全并行运行，互不相干。每个MapTask会读取分析一个InputSplit(输入分片，简称分片)对应的原始数据。计算的结果数据会临时保存到所在节点的本地磁盘里。

该阶段的编程模型中会有一个map函数需要开发人员重写，map函数的输入是一个<key,value>对,map函数的输出也是一个<key,value>对,key和value的类型需要开发人员指定。参考下图:

4.1.5.2. 第二阶段: Reduce

第二阶段，也称为Reduce阶段。这个阶段会有若干个ReduceTask实例并发运行，互不相干。但是他们的数据依赖于上一个阶段的所有maptask并发实例的输出。一个ReudceTask会从多个MapTask运行节点上fetch自己要处理的分区数据。经过处理后，输出到HDFS上。

该阶段的编程模型中有一个reduce函数需要开发人员重写，reduce函数的输入也是一个<key，value>对，reduce函数的输出也是一个<key，value>对。这里要强调的是，reduce的输入其实就是map的输出，只不过map的输出经过shuffle技术后变成了<key,List<Value>>而已。参考下图:

注意: MapReduce编程模型只能包含一个map阶段和一个reduce阶段，如果用户的业务逻辑非常复杂，那就只能多个MapReduce程序，串行运行。

4.1.6. MapReduce的编程规范【掌握】

用户编写MapReduce程序的时候，需要设计至少三个类: Mapper、Reducer、Driver(用于提交MR的任务)

4.1.6.1. Mapper

1. 自定义类，继承 **org.apache.hadoop.mapreduce.Mapper** 类型
2. 定义K1,V1,K2,V2的泛型（K1,V1是Mapper的输入数据类型，K2,V2是Mapper的输出数据类型）
3. 重写map方法（处理逻辑）

```java
/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, IntWritable>.Context context) throws IOException, InterruptedException {
        // Map阶段，在每一个数据节点需要进行的处理逻辑
    }
}
```

KEYIN：即K1的数据类型，表示读取到的一行数据的行偏移量，只能设置为LongWritable类型。
VALUEIN：即V1的数据类型，表示读取到的一行的数据，只能设置为Text类型。
KEYOUT：即K2的数据类型，表示Map阶段写出的键值对中键的类型。
VALUEOUT：即V2的数据类型，表示Map阶段写出的键值对中值的类型。

注意: Map方法，每一个键值对都会调用一次。

4.1.6.2. Reducer

1. 自定义类，继承 **org.apache.hadoop.mapreduce.Reducer** 类型
2. 定义K2,V2,K3,V3的泛型（K2,V2是Reducer的输入数据类型，K3,V3是Reducer的输出数据类型）
3. 重写reduce方法的处理逻辑

```java
/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws IOException, InterruptedExcept
        // Reduce阶段需要进行的逻辑处理，一般进行合并的操作
    }
}
```

KEYIN: 即K2的数据类型，与Map阶段输出的K2数据类型一致。
VALUEIN: 即V2的数据类型，与Map阶段输出的V2的数据类型一致。
KEYOUT: 即K3的数据类型，最终需要输出的数据类型。
VALUEOUT: 即V3的数据类型，最终需要输出的数据类型。

注意: reduce方法，默认按key分组，每一组都调用一次。

4.1.6.3. Driver

MapReduce的程序，需要进行执行之前的属性配置与任务的提交，这些操作都需要在Driver类中来完成。

4.1.7. WordCount案例演示【重点】

4.1.7.1. 案例需求

给定一个路径，统计这个路径下所有的文件中的每一个单词出现的次数。

> 放轻松，这个需求非常的简单，相当于我们在学习Java的时候写的Hello World程序那么简单，是一个入门案例。

4.1.7.2. 测试数据

```
a.txt
hello qianfeng hello 1999 hello beijing hello
world hello hello java good

b.txt
hello xisanqi hello bingbing
hello chenchen hello
ACMilan hello china

c.txt
hello hadoop hello java hello storm hello spark hello redis hello zookeeper
hello hive hello hbase hello flume
```

4.1.7.3. pom文件

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>3.3.1</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-client -->
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.3.1</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-hdfs -->
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>3.3.1</version>
    </dependency>
</dependencies>
```

4.1.7.4. 定义Mapper类

```
package wc;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
```

```
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    /**
     * Map阶段，每当读取到一行的数据，都会触发这个方法
     * 注意：LongWritable、Text、IntWritable都是Hadoop中的序列化的数据类型，相当于Java中的long、String、int类型。这部分内容在后续的序列化部分讲解。
     *
     * @param key 读取到数据的行偏移量，表示这一行的数据的首字符在这个数据块中是第几个字符
     * @param value 读取到的一行的数据
     * @param context 操作上下文
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, IntWritable>.Context context) throws IOException, InterruptedException {
        // 1. 获取读取到的每一行的数据
        String line = value.toString();
        // 2. 将每一行的数据进行切割，得到每一个单词
        String[] words = line.split("\\s+");
        // 3. 将每一个单词与数字1拼接成键值对，写出
        for (String word : words) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

4.1.7.5. 定义Reducer类

```
package wc;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    /**
     * 经过Shuffle阶段的处理，Map阶段写出的所有键值对按照Key进行了分组，并将所有的值都存入一个集合中。形成了 <K2, Iterable<V2>>的键值对。每一个Key都会触发一次这个方法。
     *
     * @param key K2, 在这个需求中就是每一个单词
     * @param values V2, 一个单词出现的所有的次数
     * @param context 操作上下文
     */
    @Override
```

```
    protected void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable,
Text, IntWritable>.Context context) throws IOException, InterruptedException {
        // 遍历这个Key对应的所有的次数，累加到一起即可得到单词出现的总次数
        // 1. 定义一个变量，用来记录总的次数
        int times = 0;
        // 2. 遍历每一个次数
        for (IntWritable value : values) {
            // 3. 累加次数，因为value是IntWritable类型的，因此需要使用get获取到包装的整型的值
            times += value.get();
        }
        // 4. 将最终的结果写出
        context.write(key, new IntWritable(times));
    }
}
```

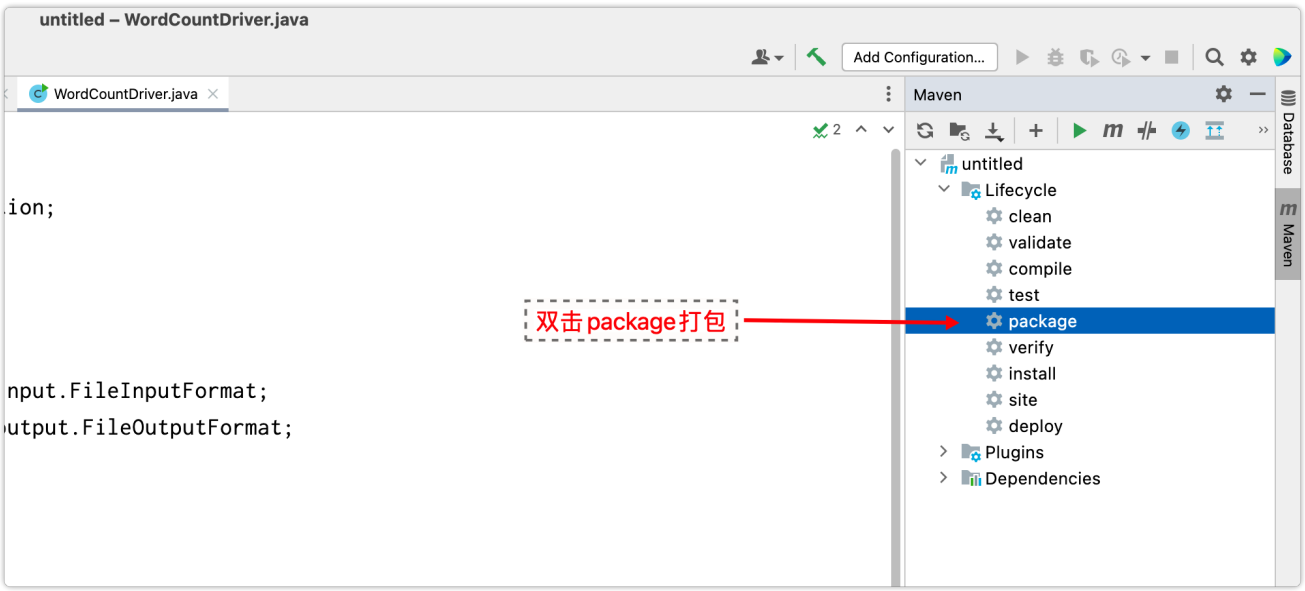
4.1.7.6. 定义Driver类

```
package wc;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class WordCountDriver {
    public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
        // 1. 获取配置信息
        //    加载默认的配置，即core-default.xml、hdfs-default.xml、mapred-default.xml和yarn-
default.xml中的配置信息
        //    然后读取项目中的core-site.xml、hdfs-site.xml、mapred-site.xml和yarn-site.xml中配置的
信息，更新某些属性的值
        Configuration conf = new Configuration();
        // 2. 如果需要继续更新某些属性的值，可以在代码中更新
        conf.set("fs.defaultFS", "hdfs://192.168.10.101:9820");
        // 3. 创建Job对象
        Job job = Job.getInstance(conf);

        // 4. 设置Mapper类型
        job.setMapperClass(WordCountMapper.class);
        // 5. 设置Reducer类型
        job.setReducerClass(WordCountReducer.class);
```

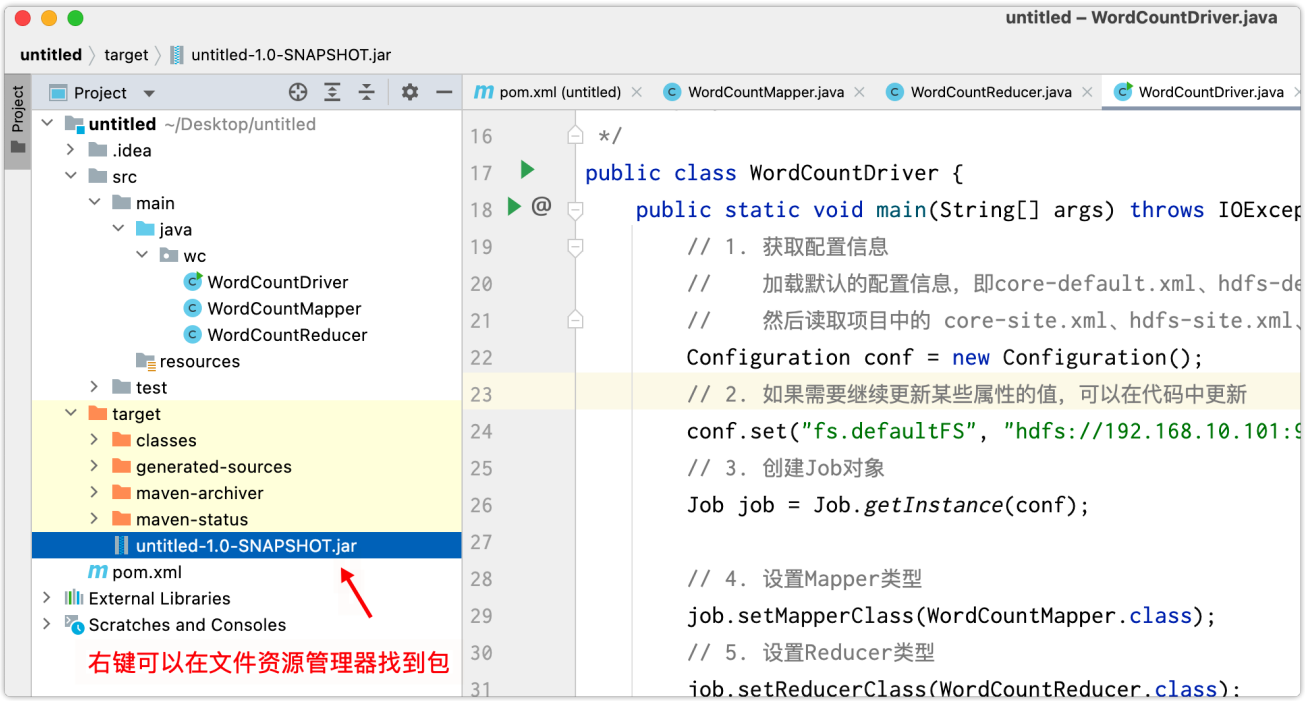
```
        // 6. 设置驱动类型
        job.setJarByClass(WordCountDriver.class);

        // 7. 设置Map阶段输出的键值对类型
        //    如果Map阶段输出的键值对类型与Reduce阶段输出的键值对类型相同，则可以省略这个设置
        //    例如：现在的Map阶段输出是<Text, IntWritable>类型的，与Reduce阶段的数据类型相同，因此可以省略不写
        // job.setMapOutputKeyClass(Text.class);
        // job.setMapOutputValueClass(IntWritable.class);

        // 8. 设置Reduce阶段输出的键值对类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 9. 设置输入输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 10. 提交Job
        System.exit(job.waitForCompletion(true) ? 0 : -1);
    }
}
```

4.1.7.7. 打jar包

打包操作:



jar包位置



4.1.7.7. 测试

1. 在本地创建好测试数据，将其上传到HDFS

```
hdfs dfs -put ~/input /wordcount
```

2. 检查YARN是否开启，如果没有开启，将其开启

```
start-yarn.sh
```

3. 执行程序

```
# hadoop jar jar包名称 驱动类全名 输入路径 输出路径
# 注意，输出路径不能存在，必须由程序自己创建。如果输出路径存在，会出现异常
hadoop jar xxxxx.jar com.qianfeng.wordcount.WordCount.Driver /wordcount/input
/wordcount/output
```

4.1.7.8. 查看结果

```
hdfs dfs -cat /wordcount/output/*
```

4.1.8. Partitioner组件【重点】

4.1.8.1. Partitioner的介绍

我们已经实现了一个WordCount的案例，统计了一个目录下所有的文件中每一个单词出现的次数，并将计算结果输出到了一个文件中存储起来。但是有时候我们的需求是需要将处理的结果存储在多个文件中，这时应该怎么做呢?

首先我们需要知道一个前提条件，在MapReduce的程序中，最终生成多少个文件是由ReduceTask的数量来决定的。在上述的案例中我们最终生成了一个文件，是因为只有一个ReduceTask。那么如果需要最终生成两个结果文件呢？我只需要将ReduceTask的数量设置为2即可。这样就带来了另外一个问题: 某一个单词的统计结果，到底应该存放在哪个文件中呢?

在MapReduce的程序中，一个MapTask处理一个分片的数据（后文会讲解），而一个ReduceTask是用来处理一个分区的数据的。因此我如果需要明确某一个单词的统计结果到底存放在哪个文件中，只需要设置好分区即可。而Partitioner就是一个分区器组件，来实现数据的分区操作的。默认的分区器是HashPartitioner，如果我们需要实现自己的分区逻辑，就需要自定义分区器了。

> HashPartitioner的逻辑:
>
> - 分区器计算每一个Key的HashCode
> - 将计算后的HashCode % ReduceTask的数量，得到一个分区的编号

4.1.8.2. 自定义Partitioner

需求: 将单词的统计结果存入三个文件: a-i开头的存一个文件、j-q开头的存一个文件、其他的存一个文件

```
package wc;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class WordCountPartitioner extends Partitioner<Text, IntWritable> {
    /**
     * 计算每一个键值对所对应的分区号，分区号从0开始
     * @param text 键
     * @param intWritable 值
     * @param i 总的ReduceTask的数量
     * @return 该键值对对应的分区
     */
    @Override
    public int getPartition(Text text, IntWritable intWritable, int i) {
        // 需求是将键值对按照首字母分为三个分区: a-i, j-q, 其他
        // 1. 获取首字母
        char firstLetter = text.toString().charAt(0);
        // 2. 判断范围
        if (firstLetter >= 'a' && firstLetter <= 'i') {
            return 0;
        } else if (firstLetter >= 'j' && firstLetter <= 'q') {
            return 1;
        }
```

```
        return 2;
    }
}
```

4.1.8.3. 应用分区器

```
package wc;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class WordCountDriver {
    public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
        // 1. 获取配置信息
        //    加载默认的配置，即core-default.xml、hdfs-default.xml、mapred-default.xml和yarn-
default.xml中的配置信息
        //    然后读取项目中的core-site.xml、hdfs-site.xml、mapred-site.xml和yarn-site.xml中配置的
信息，更新某些属性的值
        Configuration conf = new Configuration();
        // 2. 如果需要继续更新某些属性的值，可以在代码中更新
        conf.set("fs.defaultFS", "hdfs://192.168.10.101:9820");
        // 3. 创建Job对象
        Job job = Job.getInstance(conf);

        // 4. 设置Mapper类型
        job.setMapperClass(WordCountMapper.class);
        // 5. 设置Reducer类型
        job.setReducerClass(WordCountReducer.class);
        // 6. 设置驱动类型
        job.setJarByClass(WordCountDriver.class);

        // 7. 设置Map阶段输出的键值对类型
        //    如果Map阶段输出的键值对类型与Reduce阶段输出的键值对类型相同，则可以省略这个设置
        //    例如：现在的Map阶段输出是<Text, IntWritable>类型的，与Reduce阶段的数据类型相同，因此可
以省略不写
        // job.setMapOutputKeyClass(Text.class);
        // job.setMapOutputValueClass(IntWritable.class);

        // 8. 设置Reduce阶段输出的键值对类型
```

```
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 9. 设置输入输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 10. 设置分区器
        job.setPartitionerClass(WordCountPartitioner.class);
        // 11. 设置ReduceTask的数量
        //     ReduceTask的数量最好和分区的数量保持一致:
        //     如果ReduceTask的数量多于分区的数量：会出现多余的ReduceTask空占资源，不去处理任何的数据，浪费资源，且生成空的结果文件
        //     如果ReduceTask的数量少于分区的数量：会出现某个分区的数据暂时无法处理，需要等待某ReduceTask任务处理结束后再处理这个分区的数据，无法高效并发
        job.setNumReduceTasks(3);

        // 12. 提交Job
        System.exit(job.waitForCompletion(true) ? 0 : -1);
    }
}
```

4.1.9. IDE运行MapReduce的模式【掌握】

4.1.9.1. local模式测本地文件

原理:

- 将MapReduce的任务资源调度设置为local，不使用YARN进行资源调度。
- 将文件系统设置为本地文件系统，不使用HDFS

```
Configuration configuration = new Configuration();
configuration.set("mapreduce.framework.name", "local");     // 设置为本地运行模式，任务不会在YARN上运行。
configuration.set("fs.defaultFS", "file:///");              // 设置为本地文件系统，不使用HDFS。
```

4.1.9.2. local模式测集群文件

原理:

- 将MapReduce的任务资源调度设置为local，不使用YARN进行资源调度。
- 将文件系统设置为HDFS

```
Configuration configuration = new Configuration();
configuration.set("mapreduce.framework.name", "local");         // 设置为本地运行模式，任务不会在YARN上运行。
configuration.set("fs.defaultFS", "hdfs://qianfeng01:9820");    // 设置为HDFS。
```

4.1.9.3. YARN模式测集群

原理:

- 将MapReduce的任务资源调度设置为YARN
- 将文件系统设置为HDFS

```
Configuration configuration = new Configuration();
configuration.set("mapreduce.framework.name", "yarn");                  // 设置为本地运行模式，任务不会在YARN上运行。
configuration.set("fs.defaultFS", "hdfs://qianfeng01:9820");             // 设置为HDFS。
configuration.set("mapreduce.app-submission.cross-platform", "true");   // 跨平台任务提交打开。
```

注意事项:

在上述的配置都完成后，将程序打jar包，然后将jar包添加到classpath，才可以运行程序。

> 在jar包上右键 -> Add As Library

4.1.9.4. 打jar包传Linux运行

原理:

- 将写好的程序打jar包，上传到Linux
- 使用hadoop jar的命令去执行

4.2. Hadoop序列化机制【掌握】

4.2.1. 为什么要序列化

在说到为什么要序列化之前，我们需要先来了解下什么是序列化和反序列化:

- **序列化:** 序列化是指将具有结构化的内存对象转为0和1组成的字节序列，以便进行网络传输或持久存储到设备的过程。
- **反序列化:** 反序列化指的是将字节序列转为内存中具有结构化的对象的过程。

> 在基于类的编程语言中，我们说需要的数据都会被封装成对象，在内存中进行管理。可是有些时候，这样的对象，我们想直接存储到磁盘中，或者想进行网络传输，那么需要怎么做呢？需要将对象序列化成0和1组成的字节序列，字节序列就可以存储到磁盘中，或者进行网络传输了。当我们需要对象时，就可以读取磁盘上的字节序列，或者接收网络传输过来的字节序列，进行反序列化成我们需要的对象就可以了。

4.2.2. Hadoop和Java序列化对比

hadoop会涉及到大量数据的传输（网络IO），比如进程之间的通信（RPC协议），reduceTask的fetch操作。而网络带宽是极其稀缺的，因此使用序列化机制迫不及待。

> Java的序列化是一个重量级序列化框架（Serializable），一个对象被序列化后，会附带很多额外的信息（各种校验信息，header，继承体系……），这些数据不是我们需要的，也不便于在网络中高效传输.

基于Hadoop在集群之间进行通讯或者RPC调用的时候，数据的序列化要快，体积要小，占用带宽要小的需求，因此，专门为hadoop单独开发了一套精简高效的序列化机制（Writable)。此序列化机制要求具有以下特点：

- 紧凑：紧凑的格式能让我们充分利用网络带宽，而带宽是数据中心最稀缺的资源
- 快速：进程通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是基本的；
- 可扩展：协议为了满足新的需求变化，所以控制客户端和服务器过程中，需要直接引进相应的协议，这些是新协议，原序列化方式能支持新的协议报文；
- 互操作：能支持不同语言写的客户端和服务端进行交互；

需要注意的是： MapReduce的key和value,都必须是可序列化的。而针对于key而言，是数据排序的关键字，因此还需要提供比较接口：WritableComparable

4.2.3. 常用类型简介

常用的Java的数据类型与Hadoop的序列化的类型对比

Java数据类型	Hadoop序列化的数据类型	释义
byte	ByteWritable	字节类型
short	ShortWritable	短整型
int	IntWritable	整型
long	LongWritable	长整型
float	FloatWritable	单精度浮点型
double	DoubleWritable	双精度浮点型
boolean	BooleanWritable	布尔型
String	Text	字符串
array	ArrayWritable	数组
Map	MapWritable	Map
null	NullWritable	空

> NullWritable是Writable的一个特殊的类型，它的序列化长度为0。它并不从数据流中读取数据，也不写入数据。它充当占位符。例如: 在MapReduce中，如果你不需要使用键或值，就可以将键或值的类型声明为NullWritable类型。
>
> 通过调用NullWritable.get()方法可以获取到这个实例。

4.2.4. 序列化接口Writable

4.2.4.1. Writable接口

```
public interface Writable {
  /**
   * Serialize the fields of this object to <code>out</code>.
   *
   * @param out <code>DataOuput</code> to serialize this object into.
   * @throws IOException
   */
  void write(DataOutput out) throws IOException;

  /**
   * Deserialize the fields of this object from <code>in</code>.
   *
   * <p>For efficiency, implementations should attempt to re-use storage in the
   * existing object where possible.</p>
   *
   * @param in <code>DataInput</code> to deseriablize this object from.
   * @throws IOException
   */
  void readFields(DataInput in) throws IOException;
}
```

4.2.4.2. WritableComparable接口

```
public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

4.2.5. 自定义序列化类型

4.2.5.1. 为什么要自定义

Hadoop为我们提供了很多支持Writable序列化的类，例如: Text、IntWritable、LongWritable等。但是这些类并不能够满足我们所有的需求，很多时候我们是需要定义自己的类，满足自己的需求的。而如果想要把一个自定义的数据类型作为K2V2或者K3V3来使用，那么就必须要实现序列化的接口Writable。如果这个自定义的数据类型是一个Key的数据类型，则还需要在满足比较的条件，也就是再额外实现一个Comparable的接口，或者可以直接实现WritableComparable接口。

4.2.5.2. 如何自定义序列化数据类型

定义一个自定义的类型，实现序列化接口即可。

```
package com.qianfeng.mr.writable;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;

import java.io.DataInput;
```

```java
import java.io.DataOutput;
import java.io.IOException;
import java.util.Objects;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 *
 * 第一种方式：可以直接实现WritableComparable接口
 *           因为WritableComparable已经实现了Writable接口和Comparable接口
 * 第二种方式：可以实现Writable接口和Comparable接口
 *
 * 注意：如果自定义的类型，会被作为key进行传输，那么必须要实现Comparable接口，因为底层会对key进行排序。
 *      如果不作为key使用，只需要实现序列化接口Writable即可。
 */
public class TextPair implements WritableComparable {
    private Text name;
    private Text info;

    public TextPair(){
        name = new Text();
        info = new Text();
    }

    public TextPair(Text name, Text info) {
        this.name = name;
        this.info = info;
    }

    /**
     * 重载一个构造器
     * @return
     */
    public TextPair(String name,String info){
        this.name = new Text(name);
        this.info = new Text(info);
    }

    public Text getName() {
        return name;
    }

    public void setName(Text name) {
        this.name = name;
    }

    public Text getInfo() {
        return info;
    }

    public void setInfo(Text info) {
        this.info = info;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        TextPair textPair = (TextPair) o;
        return Objects.equals(name, textPair.name) &&
                Objects.equals(info, textPair.info);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, info);
    }

    public String toString(){
        return  "["+name.toString()+","+info.toString()+"]";
    }


    /**
     * 序列化方法，将属性序列化成字节序列
     * @param out
     * @throws IOException
     */
    public void write(DataOutput out) throws IOException {
        //将属性写到输出流程
        name.write(out);
        info.write(out);

        //如果不是hadoop类型，比如是java类型
//        out.writeUTF(name);
//        out.writeUTF(info);
    }


    /**
     * 反序列化方法，从流中读取字节序列进行反序列化。
     * @param in
     * @throws IOException
     */
    public void readFields(DataInput in) throws IOException {
        //要按照序列化的顺序进行反序列化
        name.readFields(in);
        info.readFields(in);
    }


    public int compareTo(Object o) {
        return 0;
    }
}
```

4.2.5.3. 测试序列化后的体积

```
package com.qianfeng.mr.writable;

import org.apache.hadoop.io.Writable;

import java.io.*;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 * @Description 测试：java类型和hadoop类型在序列化后的字节数量
 */
public class TestDemo {
    public static void main(String[] args) throws IOException {
        /**
         * 序列化java对象
         */
        StudentSer s1 = new StudentSer("小张",23);
        ObjectOutputStream  oos = new ObjectOutputStream(
                new FileOutputStream("D:/studentser.txt"));

        oos.writeObject(s1);
        oos.close();

        /**
         * 序列化hadoop对象
         */
        StudentWri s2 = new StudentWri("小张",23);
        DataOutputStream dos = new DataOutputStream(
                new FileOutputStream("D:/studentwri.txt"));
        s2.write(dos);
        dos.close();
    }
}

/**
 * java类型
 */
class StudentSer implements Serializable{
    String name;
    int age;

    public StudentSer(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
/**
 * hadoop类型
 * */
class StudentWri implements Writable{
    String name;
    int age;
    public StudentWri(String name,int age){
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(name);
        out.writeInt(age);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        name = in.readUTF();
        age = in.readInt();
    }
}
```

4.2.6. 手机流量统计案例

4.2.6.1. 手机流量的需求分析

需求: 读取文件中每一行的手机流量信息，统计每一个手机号码消费的总上行、总下行和总流量。

分析:

- 对于每一行的统计信息来说，我们需要统计手机号码、上行流量和下行流量。但是MapReduce的程序设计要求我们在Map阶段和Reduce阶段输出的都是一个键值对。因此我们需要自定义数据类型，将手机流量信息封装起来。

  > 类: PhoneFlowBean
  >
  > 属性: 手机号码phoneNumber，上行流量: upFlow，下行流量: downFlow

- Map阶段读取每一行的数据，将上行和下行流量封装到一个PhoneFlowBean对象中。

  > K2: 手机号码
  >
  > V2: PhoneFlowBean对象

- Reduce阶段将每一个手机号码对应的PhoneFlowBean对象封装的信息汇总起来

4.2.6.2. PhoneFlowBean

```
package flow;

import org.apache.hadoop.io.Writable;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.util.Objects;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class PhoneFlowBean implements Writable {
    private String phoneNumber;
    private int upFlow;
    private int downFlow;

    public PhoneFlowBean() {
    }

    public PhoneFlowBean(String phoneNumber, int upFlow, int downFlow) {
        this.phoneNumber = phoneNumber;
        this.upFlow = upFlow;
        this.downFlow = downFlow;
    }
```

```
    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeUTF(this.phoneNumber);
        dataOutput.writeInt(this.upFlow);
        dataOutput.writeInt(this.downFlow);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        this.phoneNumber = dataInput.readUTF();
        this.upFlow = dataInput.readInt();
        this.downFlow = dataInput.readInt();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PhoneFlowBean that = (PhoneFlowBean) o;
        return upFlow == that.upFlow && downFlow == that.downFlow && Objects.equals(phoneNumber,
that.phoneNumber);
    }

    @Override
    public int hashCode() {
        return Objects.hash(phoneNumber, upFlow, downFlow);
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public int getUpFlow() {
        return upFlow;
    }

    public void setUpFlow(int upFlow) {
        this.upFlow = upFlow;
    }

    public int getDownFlow() {
        return downFlow;
    }

    public void setDownFlow(int downFlow) {
        this.downFlow = downFlow;
    }
}
```

4.2.6.3. Mapper

```
package flow;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class PhoneFlowMapper extends Mapper<LongWritable, Text, Text, PhoneFlowBean> {
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
PhoneFlowBean>.Context context) throws IOException, InterruptedException {
        // 1. 读取一行的数据
        String line = value.toString();
        // 2. 切分出每一个部分
        String[] parts = line.split("\\s+");
        // 3. 提取手机号码
        String phoneNumber = parts[1];
        // 4. 提取上行流量
        int upFlow = Integer.parseInt(parts[parts.length - 3]);
        // 5. 提取下行流量
        int downFlow = Integer.parseInt(parts[parts.length - 2]);
        // 6. 封装PhoneFlowBean对象
        PhoneFlowBean bean = new PhoneFlowBean(phoneNumber, upFlow, downFlow);
        // 7. 写出键值对K2V2
        context.write(new Text(phoneNumber), bean);
    }
}
```

4.2.6.4. Reducer

```
package flow;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
```

```
public class PhoneFlowReducer extends Reducer<Text, PhoneFlowBean, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<PhoneFlowBean> values, Reducer<Text, PhoneFlowBean,
Text, Text>.Context context) throws IOException, InterruptedException {
        // 1. 定义变量，用来统计总上行流量、总下行流量
        int sumUpFlow = 0, sumDownFlow = 0;
        // 2. 遍历所有的流量统计信息，计算总流量
        for (PhoneFlowBean bean : values) {
            sumUpFlow += bean.getUpFlow();
            sumDownFlow += bean.getDownFlow();
        }
        // 3. 计算总流量
        int sumFlow = sumUpFlow + sumDownFlow;
        // 4. 拼接结果字符串
        String flowDesc = String.format("总上行流量: %d，总下行流量: %d，总流量: %d", sumUpFlow,
sumDownFlow, sumFlow);
        // 5. 写出结果
        context.write(key, new Text(flowDesc));
    }
}
```

4.2.6.5. Driver

```
package flow;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 * @author 章鱼哥
 * @company 北京千锋互联科技有限公司
 */
public class PhoneFlowDriver {
    public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
        // 1. 读取配置信息
        Configuration configuration = new Configuration();
        //    设置本地模式调本地文件
        configuration.set("mapreduce.framework.name", "local");
        configuration.set("fs.defaultFS", "file:///");

        // 2. 创建Job
        Job job = Job.getInstance(configuration);
```

```
        // 3. 设置相关的Mapper和Reducer的类
        job.setMapperClass(PhoneFlowMapper.class);
        job.setReducerClass(PhoneFlowReducer.class);
        job.setJarByClass(PhoneFlowDriver.class);

        // 4. 设置输出的KV的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(PhoneFlowBean.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // 5. 设置输入输出的路径
        FileInputFormat.setInputPaths(job, new
Path("file/in/phoneFlow/HTTP_20130313143750.dat"));

        // 6. 检测输出路径是否存在，如果存在先删除之前的内容
        Path outPath = new Path("file/out/phoneFlow");
        FileSystem fileSystem = FileSystem.get(configuration);
        if (fileSystem.exists(outPath)) {
            fileSystem.delete(outPath, true);
        }
        FileOutputFormat.setOutputPath(job, outPath);

        // 7. 提交任务
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4.3. MapReduce基础【重点】

4.3.1. MapReduce运行流程概述

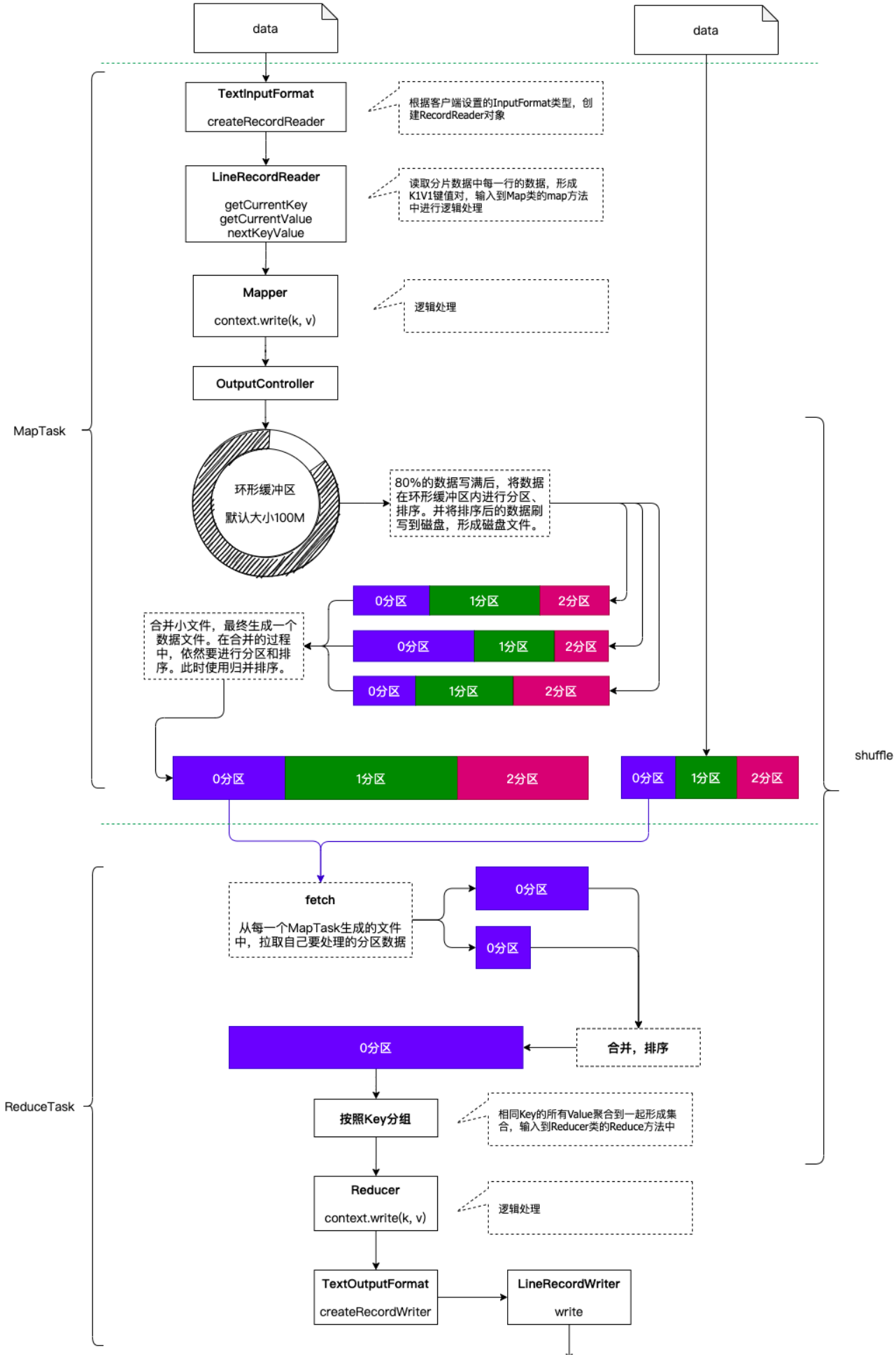
一个完整的MapReduce程序在分布式运行时有三类实例进程：

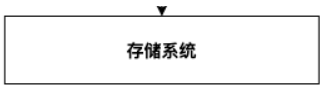
- MRAppMaster: 负责整个程序的过程调度及状态协调
- MapTask: 负责Map阶段的整个数据处理流程
- ReduceTask: 负责Reduce阶段的整个数据处理流程

当一个作业提交后(mr程序启动)，大概流程如下：

1. 一个mr程序启动的时候，会先启动一个进程Application Master,它的主类是MRAppMaster
2. ApplicationMaster启动之后会根据本次job的描述信息，计算出inputSplit的数据，也就是MapTask的数量
3. ApplicationMaster然后向ResourceManager来申请对应数量的Container来执行MapTask进程。
4. MapTask进程启动之后，根据对应的inputSplit来进行数据处理，处理流程如下
    1. 利用客户指定的inputformat来获取recordReader读取数据，形成kv键值对。
    2. 将kv传递给客户定义的Mapper类的map方法，做逻辑运算，并将map方法的输出kv收集到缓存。
    3. 将缓存中的kv数据按照k分区排序后不断的溢出到磁盘文件

5. ApplicationMaster监控mapTask进程完成之后，会根据用户指定的参数来启动相应的reduceTask进程，并告知reduceTask需要处理的数据范围
6. ReduceTask启动之后，根据ApplicationMaster告知的待处理的数据位置，从若干的已经存到磁盘的数据中拿到数据，并在本地进行一个归并排序，然后，再按照相同的key的kv为一组，调用客户自定义的reduce方法，并收集输出结果kv，然后按照用户指定的outputFormat将结果存储到外部设备。


data
data
TextInputFormat
createRecordReader
根据客户端设置的InputFormat类型，创建RecordReader对象
LineRecordReader
getCurrentKey
getCurrentValue
nextKeyValue
读取分片数据中每一行的数据，形成K1V1键值对，输入到Map类的map方法中进行逻辑处理
Mapper
context.write(k, v)
逻辑处理
OutputController
MapTask
环形缓冲区
默认大小100M
80%的数据写满后，将数据在环形缓冲区内进行分区、排序。并将排序后的数据刷写到磁盘，形成磁盘文件。
0分区
1分区
2分区
0分区
1分区
2分区
0分区
1分区
2分区
合并小文件，最终生成一个数据文件。在合并的过程中，依然要进行分区和排序。此时使用归并排序。
0分区
1分区
2分区
0分区
1分区
2分区
shuffle
fetch
从每一个MapTask生成的文件中，拉取自己要处理的分区数据
0分区
0分区
0分区
合并，排序
ReduceTask
按照Key分组
相同Key的所有Value聚合到一起形成集合，输入到Reducer类的Reduce方法中
Reducer
context.write(k, v)
逻辑处理
TextOutputFormat
createRecordWriter
LineRecordWriter
write



4.3.2. 分片机制

4.3.2.1. 分片的概念

MapReduce在进行作业提交时，会预先对将要分析的原始数据进行划分处理，形成一个个等长的逻辑数据对象，称之为输入分片（inputSplit），简称“分片”。MapReduce为每一个分片构建一个单独的MapTask，并由该任务来运行用户自定义的map方法，从而处理分片中的每一条记录。

> 分片是一个逻辑概念，分块是一个物理概念。
>
> HDFS上数据是按照块为单位进行存储的，我们是能够实实在在的看到每一个数据块的。而分片则不然，是一个逻辑概念，用来描述一个MapTask处理的数据是属于哪个文件的，从什么字节位置开始处理，处理多少个字节的数据等等信息。

4.3.2.2. 分片的大小选择

每一个MapTask处理一个分片的数据，因此分片的数量就决定了MapTask的数量。拥有多个分片，就意味着会有多个MapTask并发执行处理数据集。那么一个MapTask处理多大的数据呢？这也是由分片的大小来决定的。

如果分片设置的太小，那么管理分片的时间和构建MapTask的总时间将在整个作业的时间占比较大，影响程序的执行效率。例如: 一个分片设置为1KB的大小，计算分片、构建MapTask耗时10ms的时间，处理数据耗时10ms的时间，那这样的程序的效率是非常低下的。我们更加乐意让一个任务初始化的时间在整个任务中的时间占比尽可能低。

如果分片设置的太大，那么分片所描述的数据可能会在两个数据块中存储，那就有可能会造成网络IO的产生，需要将数据移动到一个节点上进行处理，效率更低。

因此，最佳分片大小应该和HDFS的块大小一致。

4.3.2.3. 分片源码解读

4.3.2.3.1. FileSplit

```
public class FileSplit extends InputSplit implements Writable {
    private Path file;              // 描述文件的路径信息
    private long start;             // 描述这个分片需要处理的数据起点
    private long length;            // 描述这个分片需要处理的数据长度
    private String[] hosts;         // 描述这个分片对应的数据块在哪些节点
    private SplitLocationInfo[] hostInfos;

    public FileSplit() {
    }

    public FileSplit(Path file, long start, long length, String[] hosts) {
        this.file = file;
        this.start = start;
        this.length = length;
        this.hosts = hosts;
    }
    ...
}
```

4.3.2.3.2. FileInputFormat

```
public abstract class FileInputFormat<K, V> extends InputFormat<K, V> {
    // ...
    // 定义了一个1.1倍的溢出值
    private static final double SPLIT_SLOP = 1.1D;

    // ...
    // 创建一个分片对象，设置这个分片需要处理的数据位置、起点、长度、hosts等信息
    protected FileSplit makeSplit(Path file, long start, long length, String[] hosts) {
        return new FileSplit(file, start, length, hosts);
    }

    // ...
    // 最重要的方法：获取文件所有的分片信息
    public List<InputSplit> getSplits(JobContext job) throws IOException {
        StopWatch sw = (new StopWatch()).start();
        long minSize = Math.max(this.getFormatMinSplitSize(), getMinSplitSize(job));
        long maxSize = getMaxSplitSize(job);
        List<InputSplit> splits = new ArrayList();
        List<FileStatus> files = this.listStatus(job);
        boolean ignoreDirs = !getInputDirRecursive(job) &&
job.getConfiguration().getBoolean("mapreduce.input.fileinputformat.input.dir.nonrecursive.ignore
.subdirs", false);
        Iterator var10 = files.iterator();

        while(true) {
            while(true) {
                while(true) {
                    FileStatus file;
                    do {
                        if (!var10.hasNext()) {

 job.getConfiguration().setLong("mapreduce.input.fileinputformat.numinputfiles",
(long)files.size());
                            sw.stop();
                            if (LOG.isDebugEnabled()) {
                                LOG.debug("Total # of splits generated by getSplits: " +
splits.size() + ", TimeTaken: " + sw.now(TimeUnit.MILLISECONDS));
                            }

                            return splits;
                        }

                        file = (FileStatus)var10.next();
                    } while(ignoreDirs && file.isDirectory());

                    // 重要逻辑在这里!!!
                    // 获取到文件的路径描述信息
                    Path path = file.getPath();
                    // 获取到文件的大小
```

```
                    long length = file.getLen();
                    // 如果文件的大小不等于0
                    if (length != 0L) {
                        // 获取数据块的分布信息
                        BlockLocation[] blkLocations;
                        if (file instanceof LocatedFileStatus) {
                            blkLocations = ((LocatedFileStatus)file).getBlockLocations();
                        } else {
                            FileSystem fs = path.getFileSystem(job.getConfiguration());
                            blkLocations = fs.getFileBlockLocations(file, 0L, length);
                        }

                        // 如果文件可以分片（有些文件是不可以分片的）
                        if (this.isSplitable(job, path)) {
                            // 获取一个Block的大小
                            long blockSize = file.getBlockSize();
                            // 计算分片的大小(块大小，配置文件中设置的最小分片大小，最大分片大小的中间
值)
                            long splitSize = this.computeSplitSize(blockSize, minSize, maxSize);

                            // 用来记录来剩多少字节的数据没有分片
                            long bytesRemaining;
                            int blkIndex;
                            // 循环分片开始了!
                            // 注意：循环的条件，并不是剩余数量不足分片大小！有一个1.1倍的溢出的值的!
                            for(bytesRemaining = length; (double)bytesRemaining /
(double)splitSize > 1.1D; bytesRemaining -= splitSize) {
                                blkIndex = this.getBlockIndex(blkLocations, length -
bytesRemaining);
                                // 创建一个分片！添加到分片集合中!
                                splits.add(this.makeSplit(path, length - bytesRemaining,
splitSize, blkLocations[blkIndex].getHosts(), blkLocations[blkIndex].getCachedHosts()));
                            }

                            // 循环走完后，创建一个分片来描述剩余的数据
                            if (bytesRemaining != 0L) {
                                blkIndex = this.getBlockIndex(blkLocations, length -
bytesRemaining);
                                splits.add(this.makeSplit(path, length - bytesRemaining,
bytesRemaining, blkLocations[blkIndex].getHosts(), blkLocations[blkIndex].getCachedHosts()));
                            }
                        } else {
                            if (LOG.isDebugEnabled() && length > Math.min(file.getBlockSize(),
minSize)) {
                                LOG.debug("File is not splittable so no parallelization is
possible: " + file.getPath());
                            }

                            splits.add(this.makeSplit(path, 0L, length,
blkLocations[0].getHosts(), blkLocations[0].getCachedHosts()));
                        }
                    } else {
                        splits.add(this.makeSplit(path, 0L, length, new String[0]));
```

```
                    }
                }
            }
        }
    }

    // 计算分片大小
    protected long computeSplitSize(long blockSize, long minSize, long maxSize) {
        return Math.max(minSize, Math.min(maxSize, blockSize));
    }
}
```

4.3.2.4. 分片总结

1. 分片大小参数

   通过分析源码，在FileInputFormat中，计算分片大小的逻辑：Math.max(minSize, Math.min(maxSize, blockSize)); 分片主要由这几个值来运算决定

参数	默认值	属性
minSize	1	mapreduce.input.fileinputformat.split.minsize
maxSize	Long.MAX_VALUE	mapreduce.input.fileinputformat.split.maxsize
blockSize	128M	dfs.blocksize

   通过计算的逻辑分析可以得出，分片大小的计算，是取这三个值的中间值的，因此:

   - 如果需要增大分片的大小: 调整minSize大于blockSize即可
   - 如果需要减小分片的大小: 调整maxSize小于blockSize即可

2. 分片创建过程总结

```
1. 获取文件大小及位置
2. 判断文件是否可以分片（压缩格式有的可以进行分片，有的不可以）
3. 获取分片的大小
4. 剩余文件的大小/分片大小>1.1时，循环执行封装分片信息的方法，具体如下：
   封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)
5. 剩余文件的大小/分片大小<=1.1且不等于0时，封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)
```

   注意事项: 1.1倍的冗余

   > 一个260M的文件，分几块？分几片?
   >
   > - 分块是物理概念: 128M + 128M + 4M，因此一共有3个分块。
   > - 分片是逻辑概念:
   >   - 第一个分片: 260M/128M > 1.1，因此第一个分片大小128M，剩余132M数据未分片。

> - 第二个分片: 132M/128M < 1.1，因此第二个分片大小132M
> - 因此这个文件有2个分片。

3. 多分片文件读取

   数据文件被分了多个分片，那么我们不能保证分片是正好按照行分开的，极大的可能性是一行的数据被分到了两个分片中。因此，我们在进行多个分片的数据读取的时候:

```
- 第一个分片读到末尾再多读一行
- 既不是第一个分片也不是最后一个分片第一行数据舍弃，末尾多读一行
- 最后一个分片舍弃第一行，末尾多读一行
```

4.3.4. 运行流程之MapTask

```
1. maptask调用FileInputFormat的getRecordReader读取分片数据
2. 每行数据读取一次，返回一个(K,V)对，K是offset,V是一行数据
3. 将k-v对交给MapTask处理
4. 每对k-v调用一次map(K,V, context)方法，然后context.write(k,v)
5. 写出的数据交给收集器OutputCollector.collector()处理
6. 将数据写入环形缓冲区，并记录写入的起始偏移量，终止偏移量，环形缓冲区默认大小100M
7. 默认写到80%的时候要溢写到磁盘，溢写磁盘的过程中数据继续写入剩余20%
8. 溢写磁盘之前要先进行分区然后分区内进行排序
9. 默认的分区规则是hashpatitioner，即key的hash%reduceNum
10. 默认的排序规则是key的字典顺序，使用的是快速排序
11. 溢写会形成多个文件，在maptask读取完一个分片数据后，先将环形缓冲区数据刷写到磁盘
12. 将数据多个溢写文件进行合并，分区内排序（外部排序===》归并排序）
```

MapTask的并行度决定map阶段的任务处理并发度，进而影响到整个job的处理速度.那么，MapTask并行实例是否越多越好呢？其并行度又是如何决定呢？

```
1. 如果硬件配置为2*12core + 64G，恰当的map并行度是大约每个节点20-100个map，最好每个map的执行时间至少一分钟。

2. 如果job的每个map或者 reduce task的运行时间都只有30-40秒钟，那么就减少该job的map或者reduce数，每一个task(map|reduce)的setup和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个task都非常快就跑完了，就会在task的开始和结束的时候浪费太多的时间。

3. 配置task的JVM重用可以改善该问题：
   (mapred.job.reuse.jvm.num.tasks，默认是1，表示一个JVM上最多可以顺序执行的task数目（属于同一个Job）是1。也就是说一个task启一个JVM)

4. 如果input的文件非常的大，比如1TB，可以考虑将hdfs上的每个block size设大，比如设成256MB或者512MB
```

4.3.5. 运行流程之ReduceTask

```
1. 数据按照分区规则发送到reducetask
2. reducetask将来自多个maptask的数据进行合并，排序（外部排序===》归并排序）
3. 按照key相同分组（）
4. 一组数据调用一次reduce(k,iterable<v>values,context)
5. 处理后的数据交由reducetask
6. reducetask调用FileOutputFormat组件
7. FileOutputFormat组件中的write方法将数据写出
```

Reduce Task的并行度同样影响整个job的执行并发度和执行效率，但与Map Task的并发数由切片数决定不同，Reduc Task数量的决定是可以直接手动设置：默认值是1，手动设置为4

```
设置方法:job.setNumReduceTasks(4);
```

如果数据分布不均匀，就有可能在reduce阶段产生数据倾斜

注意： Reduce Task数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能有1个Reduce Task。尽量不要运行太多的Reduce Task。对大多数job来说，最好reduce的个数最多和集群中的reduce持平，或者比集群的 reduce slots小。这个对于小集群而言，尤其重要。

4.4. MapReduce高级【重点】

4.4.1 shuffle阶段

4.4.1.1. 概述

MapReduce会确保每个reducer的输入都是按键排序的。从map方法输出数据开始、到作为输入数据传给reduce方法的过程称为shuffle。在此，我们将学习shuffle是如何工作的，因为它有助于我们理解工作机制（如果需要优化MapReduce程序）。shuffle属于不断被优化和改进的代码库的一部分，因此会随着版本的不同，细节上可能会发生变量。不管怎样，从许多方面来看，shuffle是MapReduce的“心脏“，是奇迹发生的地方。

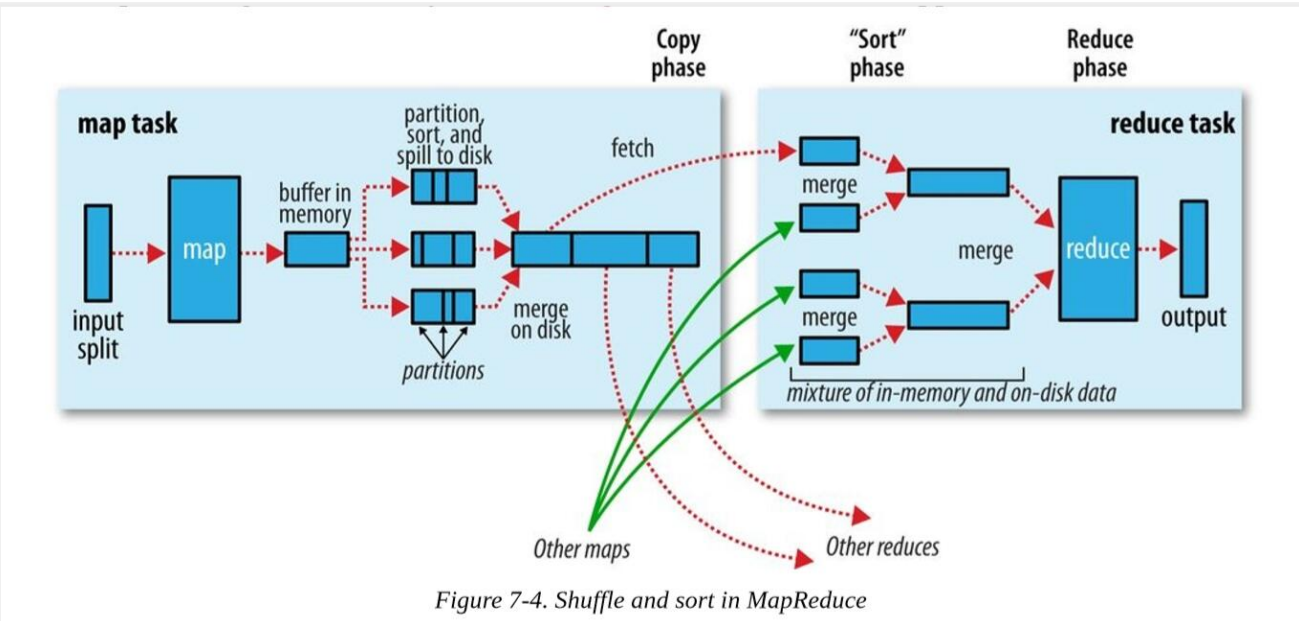


Figure 7-4. Shuffle and sort in MapReduce

4.4.1.2. map端

map方法开始产生输出数据时，并不是简单地将它写到磁盘。这个过程非常复杂，它利用缓冲的方式写到内存并出于效率的考虑进行预排序（可以参考上图）

每个map任务都会有一个环形内存缓冲区用于存储map的输出数据。在默认情况下，缓冲区的大小为100MB,这个值可以通过mapreduce.task.io.sort.mb属性来调整。一旦缓冲区的内容达到阈值（默认是0.8，或者是80%，属性是mapreduce.map.sort.spill.percent），一个后台线程便开始把内容溢写(spill)到磁盘里，这个位置由属性mapreduce.cluster.local.dir来指定的。在将数据溢写到磁盘过程中，map的输出数据继续写到缓冲区，但如果在此期间缓冲区被填满，map会被阻塞直到写磁盘过程完成。

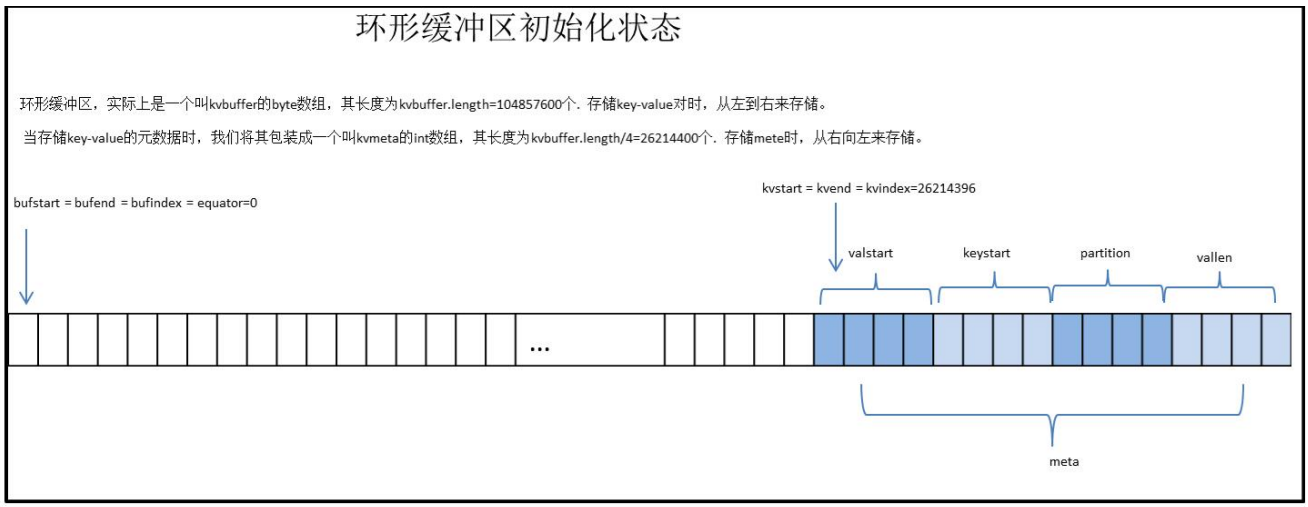
在写磁盘之前，线程会根据分区器的逻辑把数据划分为不同的分区(partition)。然后，在每个分区中，后台线程会按键进行内存中排序（QuickSort，默认是字典顺序）。如果指定了一个combiner函数，它就在排序后的输出上运行。运行combiner函数使得map输出结果更紧凑，因此减少写到磁盘的数据和传递给reducer的数据。

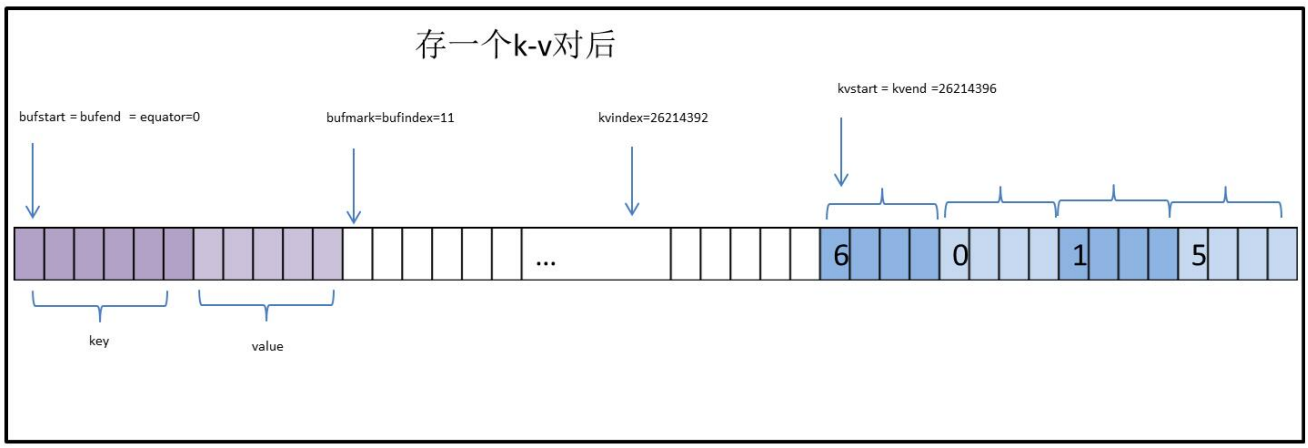
每次内存缓冲区达到溢出阈值，就会新建一个溢出文件(spill file),因此在map任务写完其最后一个输出记录之后，可能会有几个溢出文件。在MapTask任务完成之前，多个溢出文件被合并成一个已分区且已排序的输出文件。配置属性mapreduce.task.io.sort.factor控制着一次最多能合并多少个文件，默认值是10。

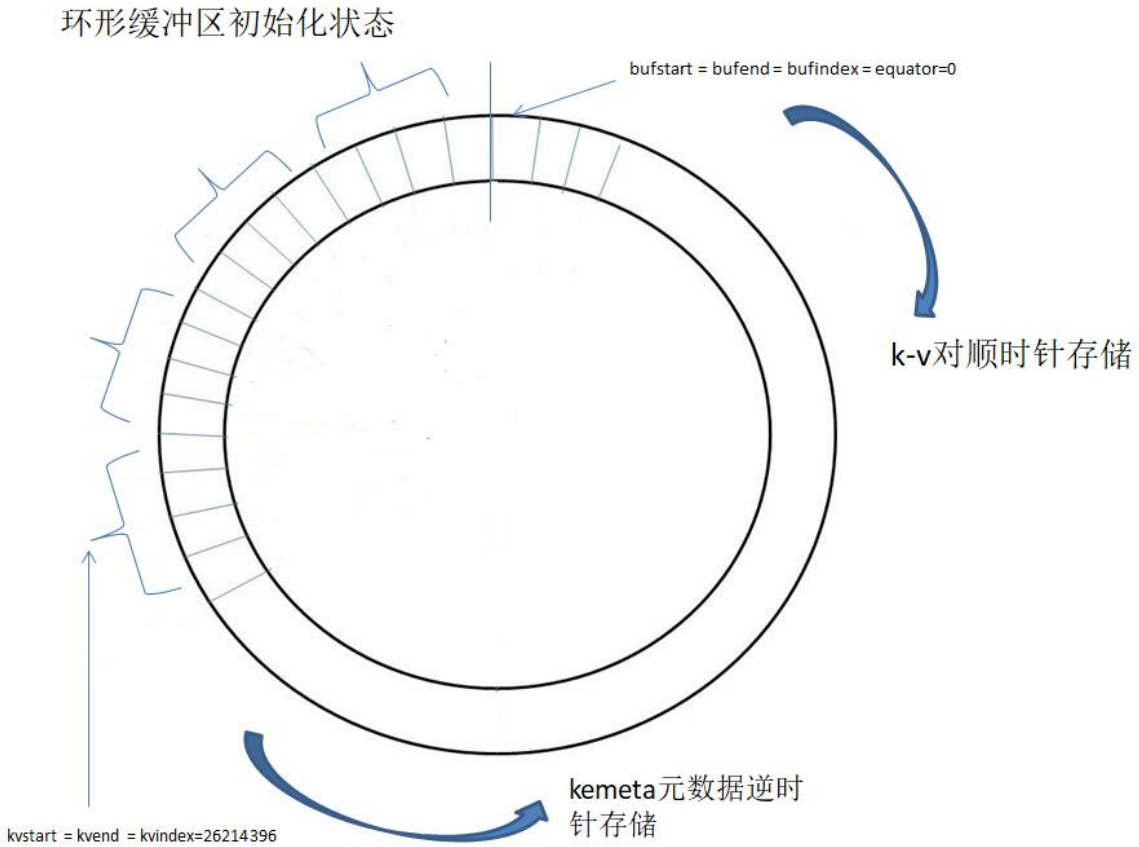
如果至少存在3个溢出文件(通过mapreduce.map.combine.minspills属性设置)时，则combiner就会在输出文件写到磁盘之前再次运行。combiner可以在输入上反复运行，但并不影响最终结果。如果只有1或2个溢出文件，那么由于map输出规模减少，因而不值得调用combiner产生开销，因此不会为该map输出再次运行combiner。

为了使写磁盘的速度更快，节约磁盘空间，并且减少传给reducer的数据量，在溢写到磁盘的过程中对数据进行压缩往往是个很好的主意。在默认情况下，输出是不压缩的，但只要将mapreduce.map.output, compress设置为true,就可以轻松启用此功能。使用的压缩库由mapreduce.map.output.compress.codec指定。

扩展 环形缓冲区的详解


环形缓冲区初始化状态
环形缓冲区，实际上是一个叫kvbuffer的byte数组，其长度为kvbuffer.length=104857600个. 存储key-value对时，从左到右来存储。
当存储key-value的元数据时，我们将其包装成一个叫kvmeta的int数组，其长度为kvbuffer.length/4=26214400个. 存储mete时，从右向左来存储。
bufstart = bufend = bufindex = equator=0
kvstart = kvend = kvindex=26214396
valstart
keystart
partition
vallen
...
meta


存一个k-v对后
bufstart = bufend = equator=0
bufmark=bufindex=11
kvindex=26214392
kvstart = kvend =26214396
...
6
0
1
5
key
value


环形缓冲区初始化状态
bufstart = bufend = bufindex = equator=0
k-v对顺时针存储
kemeta元数据逆时针存储
kvstart = kvend = kvindex=26214396

4.4.1.3. Reduce端

reducer通过HTTP得到输出文件的分区。用于文件分区的工作线程的数量由任务的mapreduce. shuffle.max. threads属性控制，此设置针对的是每一个节点管理器，而不是针对每个map任务。

现在转到处理过程的reduce部分。map输出文件位于运行MapTask的本地磁盘（注意，尽管map输出经常写到MapTask本地磁盘，但reduce输出并不这样）。现在，tasktracker需要为分区文件运行reduce任务。并且，reduce任务需要集群上若干个map任务的map输出作为其特殊的分区文件。每个map任务的完成时间可能不同，因此在每个任务完成时，reduce任务就开始复制其输出。这就是reduce任务的复制阶段。reduce任务有少量复制线程，因此能够并行取得map输出。默认值是5个线程，但这个默认值可以修改设置mapreduce.reduce.shuffle. parallelcopies 属性即可。

```
reducer如何知道要从哪台机器取得map输出呢?

map任务成功完成后，它们会使用心跳机制通知它们的application master。因此，对于指定作业，application master知道map输出和主机位置之间的映射关系。reducer中的一个线程定期询问master以便获取map输出主机的位置，直到获得所有输出位置。

由于第一个reducer可能失败，因此主机并没有在第一个reducer检索到map输出时就立即从磁盘上删除它们。相反，主机会等待，直到application master告知它删除map输出，这是作业完成后执行的。
```

如果map输出相当小，会被复制到reduce任务JVM的内存（缓冲区大小由mapreduce.reduce.shuffle.input. buffer.percent 属性控制，指定用于此用途的堆空间的百分比），否则，map输出被复制到磁盘。一旦内存缓冲区达到阈值大小（由 mapreduce.reduce.shuffle.merge.percent 决定）或达到 map 输出阈值（由 mapreduce. reduce. merge. inmem .threshold 控制），则合并后溢出写到磁盘中。如果指定combiner,则在合并期间运行它以降低写入硬盘的数据量。

随着磁盘上的溢写文件数量增多，后台线程会将它们合并为更大的、排好序的文件。这会为后面的合并节省一些时间。注意，为了合并，压缩的map输出（通过map任务）都必须在内存中被解压缩。

复制完所有map输出后，reduce任务进入排序阶段（更恰当的说法是合并阶段，因为排序是在map端进行的），这个阶段将合并map输出，维持其顺序排序。这是循环进行的。比如，如果有50个map输出，而合并因子是10（10为默认设置，由mapreduce.task. io.sort.factor,与 的合并类似），合并将进行 5 趟 ，每趟将10个文件合并成一个文件，因此最后有5个中间文件。

在最后阶段，即reduce阶段，直接把数据输入reduce函数，从而省略了一次磁盘往返行程，并没有将这5个文件合并成一个已排序的文件作为最后一趟。最后的合并可以来自内存和磁盘片段。

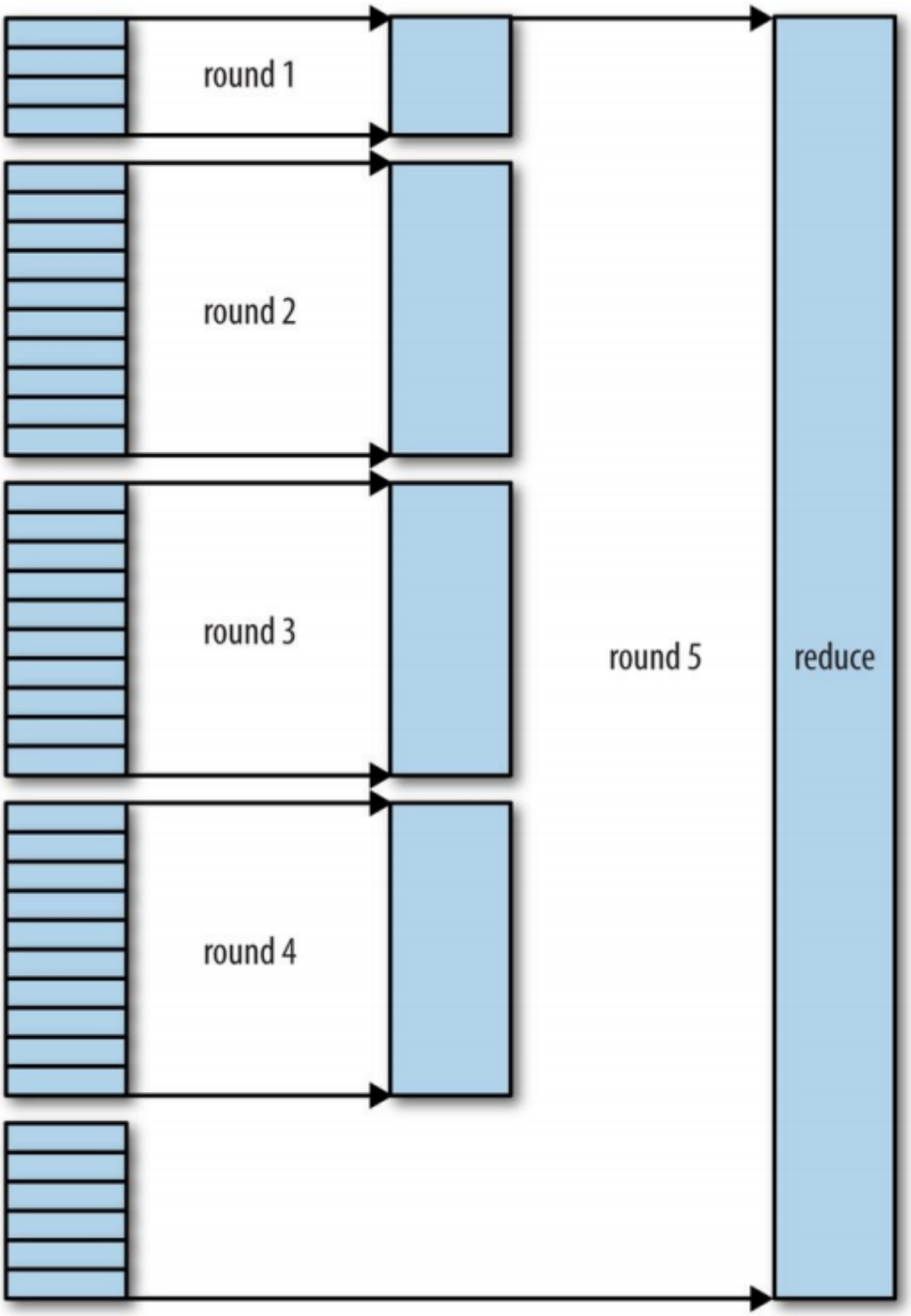


Figure 7-5. Efficiently merging 40 file segments with a merge factor of 10

4.4.1.4. shuffle流程总结

1. 从map函数输出到reduce函数接受输入数据，这个过程称之为shuffle.

2. map函数的输出，存储环形缓冲区（默认大小100M,阈值80M）

   环形缓冲区：其实是一个字节数组kvbuffer. 有一个sequator标记，kv原始数据从左向右填充(顺时针)，kvmeta是对kvbuffer的一个封装，封装成了int数组，用于存储kv原始数据的对应的元数据valstart，keystart，partition，vallen信息，从右向左(逆时针)。参考(环形缓冲区的详解一张)

3．当达到阈值时，准备溢写到本地磁盘(因为是中间数据，因此没有必要存储在HDFS上)。在溢写前要进行对元数据分区(partition)整理，然后进行排序(quick sort,通过元数据找到出key，同一分区的所有key进行排序，排序完，元数据就已经有序了，在溢写时，按照元数据的顺序寻找原始数据进行溢写)

4．如果有必要，可以在排序后，溢写前调用combiner函数进行运算，来达到减少数据的目的

5．溢写文件有可能产生多个，然后对这多个溢写文件进行再次合并(也要进行分区和排序)。当溢写个数>=3时，可以再次调用combiner函数来减少数据。如果溢写个数<3时，默认不会调用combiner函数。

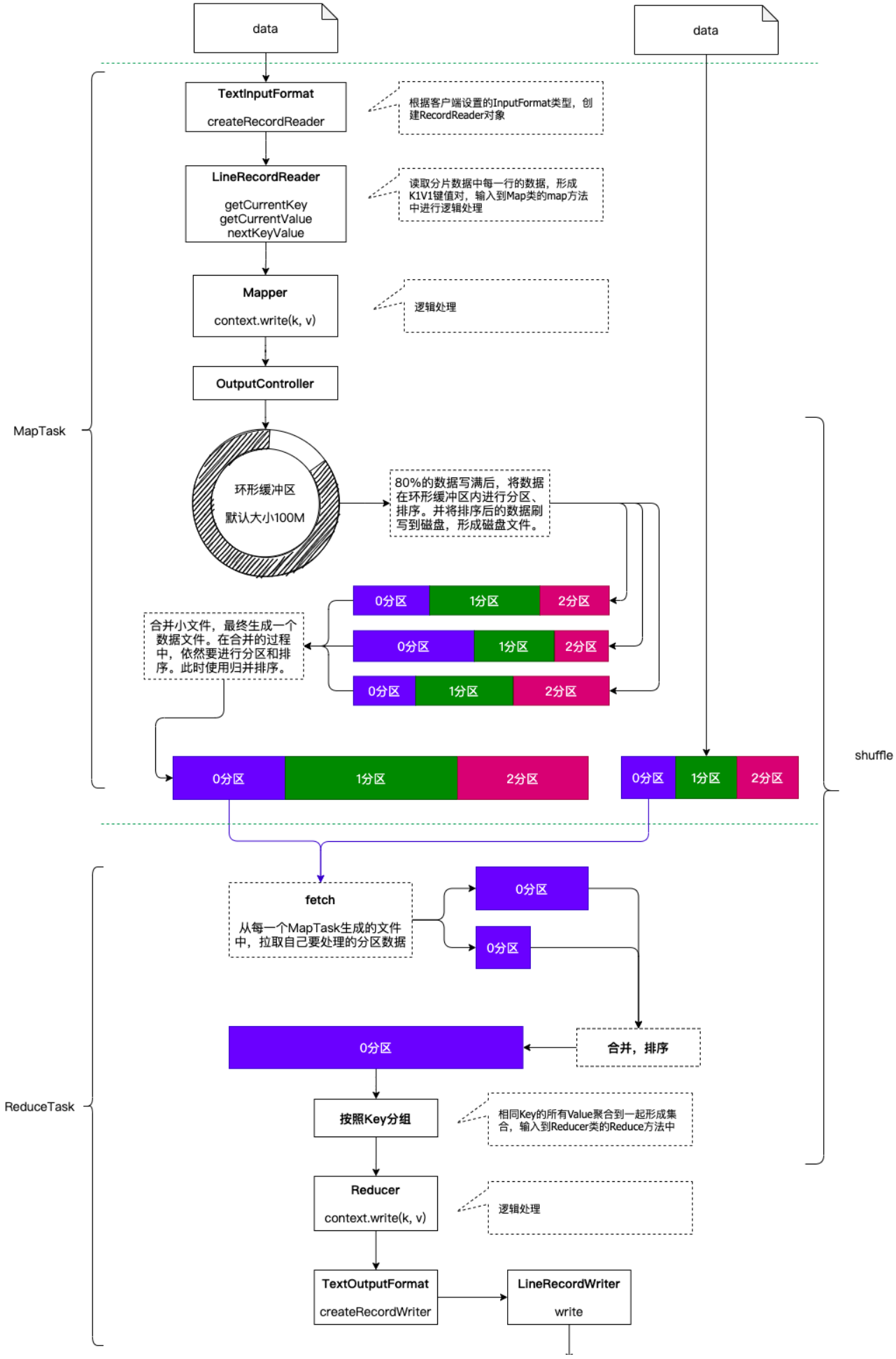
6．合并的最终溢写文件可以使用压缩技术来达到节省磁盘空间和减少向reduce阶段传输数据的目的。（存储在本地磁盘中)

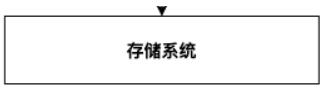
7．Reduce阶段通过HTTP写抓取属于自己的分区的所有map的输出数据(默认线程数是5，因此可以并发抓取)。

8．抓取到的数据存在内存中，如果数据量大，当达到本地内存的阈值时会进行溢写操作，在溢写前会进行合并和排序(排序阶段)，然后写到磁盘中，

9．溢写文件可能会产生多个，因此在进入reduce之前会再次合并(合并因子是10),最后一次合并要满足10这个因子，同时输入给reduce函数，而不是产生合并文件。reduce函数输出数据会直接存储在HDFS上。

4.4.1.5. shuffle整体流程图


data
data
TextInputFormat
createRecordReader
根据客户端设置的InputFormat类型，创建RecordReader对象
LineRecordReader
getCurrentKey
getCurrentValue
nextKeyValue
读取分片数据中每一行的数据，形成K1V1键值对，输入到Map类的map方法中进行逻辑处理
Mapper
context.write(k, v)
逻辑处理
OutputController
MapTask
环形缓冲区
默认大小100M
80%的数据写满后，将数据在环形缓冲区内进行分区、排序。并将排序后的数据刷写到磁盘，形成磁盘文件。
0分区
1分区
2分区
0分区
1分区
2分区
0分区
1分区
2分区
合并小文件，最终生成一个数据文件。在合并的过程中，依然要进行分区和排序。此时使用归并排序。
0分区
1分区
2分区
0分区
1分区
2分区
shuffle
fetch
从每一个MapTask生成的文件中，拉取自己要处理的分区数据
0分区
0分区
0分区
合并，排序
ReduceTask
按照Key分组
相同Key的所有Value聚合到一起形成集合，输入到Reducer类的Reduce方法中
Reducer
context.write(k, v)
逻辑处理
TextOutputFormat
createRecordWriter
LineRecordWriter
write



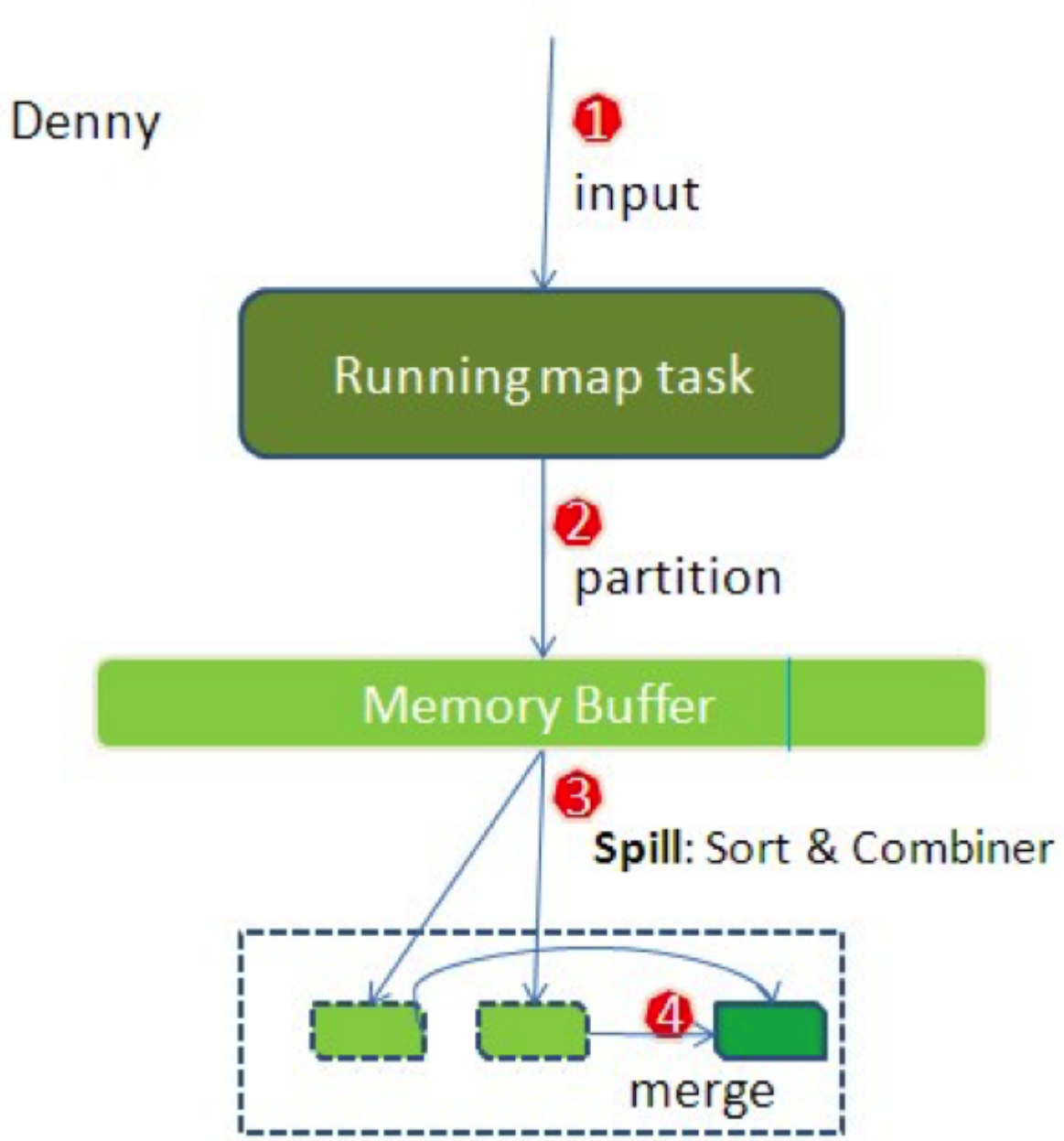
细节分析

> 在Hadoop这样的集群环境中，大部分map task与reduce task的执行是在不同的节点上。当然很多情况下Reduce执行时需要跨节点去拉取其它节点上的map task结果。如果集群正在运行的job有很多，那么task的正常执行对集群内部的网络资源消耗会很严重。这种网络消耗是正常的，我们不能限制，能做的就是最大化地减少不必要的消耗。还有在节点内，相比于内存，磁盘IO对job完成时间的影响也是可观的。从最基本的要求来说，我们对shuffle过程的期望可以有：
>
> a）完整地从map task端拉取数据到reduce 端。
>
> b）在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗。
>
> c）减少磁盘IO对task执行的影响。

OK，看到这里时，大家可以先停下来想想，如果是自己来设计这段shuffle过程，那么你的设计目标是什么。能优化的地方主要在于减少拉取数据的量及尽量使用内存而不是磁盘。shuffle过程横跨map与reduce两端,下面我会先说明总体流程,在详细说明map task和reduce task阶段.

4.4.1.6. MapTask的具体执行流程(选看)

这里的分析是基于Hadoop0.21.0的源码，以WordCount为例，并假设它有8个map task和3个reduce task。 如下图：



整个流程我分了四步。简单些可以这样说，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。

这里的每一步都可能包含着多个步骤与细节，下面我对细节来一一说明：

1）在map task执行时，它的输入数据来源于HDFS的block，当然在MapReduce概念中，map task只读取split。Split与block的对应关系可能是多对一，默认是一对一。在WordCount例子里，假设map的输入数据都是像“aaa”这样的字符串。

2）在经过mapper的运行后，我们得知mapper的输出是这样一个key/value对： key是“aaa”， value是数值1。因为当前map端只做加1的操作，在reduce task里才去合并结果集。前面我们知道这个job有3个reduce task，到底当前的“aaa”应该交由哪个reduce去做呢，是需要现在决定的。

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。

在我们的例子中，“aaa”经过Partitioner后返回0，也就是这对值应当交由第一个reducer来处理。接下来，需要将数据写入内存缓冲区中，缓冲区的作用是批量收集map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。

整个内存缓冲区就是一个字节数组，它的字节索引及key/value存储结构我没有研究过。如果有朋友对它有研究，那么请大致描述下它的细节吧。

3）这个内存缓冲区是有大小限制的，默认是100MB。当map task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，字面意思很直观。这个溢写是由单独线程来完成，不影响往缓冲区写map结果的线程。溢写线程启动时不应该阻止map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。Map task的输出结果还可以往剩下的20MB内存中写，互不影响。

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。

在这里我们可以想想，因为map task的输出是需要发送到不同的reduce端去，而内存缓冲区没有对将发送到相同reduce端的数据做合并，那么这种合并应该是体现是磁盘文件中的。从官方图上也可以看到写到磁盘中的溢写文件是对不同的reduce端的数值做过合并。所以溢写过程一个很重要的细节在于，如果有很多个key/value对需要发送到某个reduce端去，那么需要将这些key/value值拼接到一块，减少与partition相关的索引记录.

在针对每个reduce端而合并数据时，有些数据可能像这样：“aaa”/1，“aaa”/1。对于WordCount例子，就是简单地统计单词出现的次数，如果在同一个map task的结果中有很多个像“aaa”一样出现多次的key，我们就应该把它们的值合并到一块，这个过程叫reduce也叫combine。但MapReduce的术语中，reduce只指reduce端执行从多个map task取数据做计算的过程。除reduce外，非正式地合并数据只能算做combine了。其实大家知道的，MapReduce中将Combiner等同于Reducer.

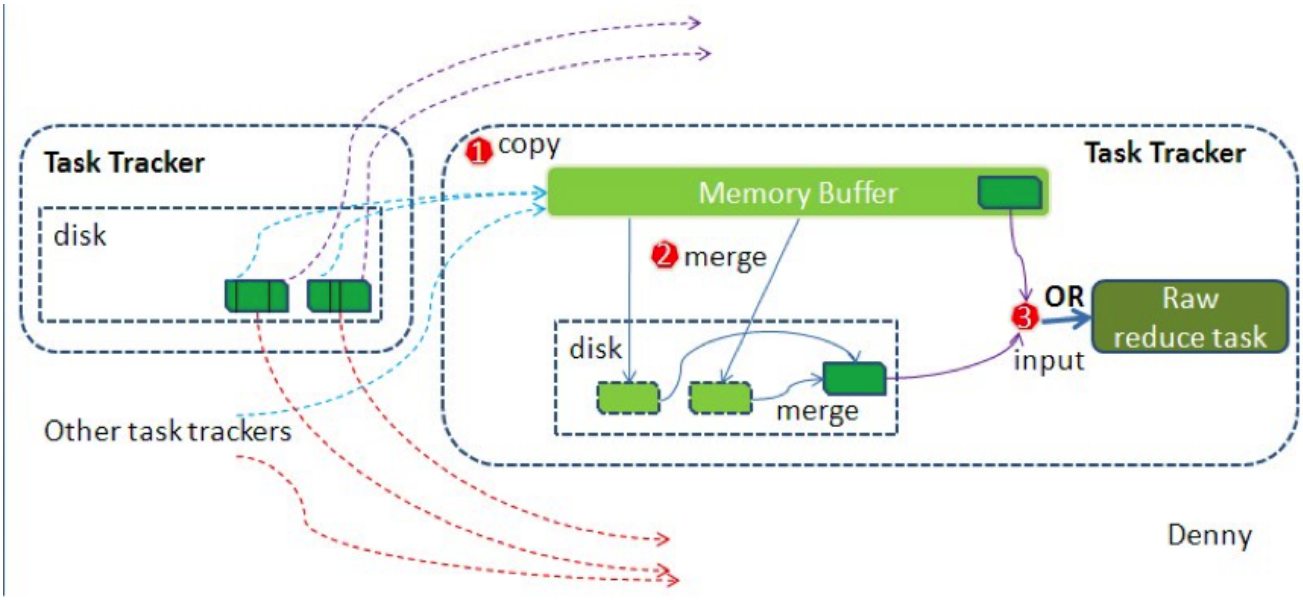
如果client设置过Combiner，那么现在就是使用Combiner的时候了。将有相同key的key/value对的value加起来，减少溢写到磁盘的数据量。Combiner会优化MapReduce的中间结果，所以它在整个模型中会多次使用。那哪些场景才能使用Combiner呢？从这里分析，Combiner的输出是Reducer的输入，Combiner绝不能改变最终的计算结果。所以从我的想法来看，Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner的使用一定得慎重，如果用好，它对job执行效率有帮助，反之会影响reduce的最终结果。

4）每次溢写会在磁盘上生成一个溢写文件，如果map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溢写文件存在。当map task真正完成时，内存缓冲区中的数据也全部溢写到磁盘中形成一个溢写文件。最终磁盘中会至少有一个这样的溢写文件存在(如果map的输出结果很少，当map执行完成时，只会产生一个溢写文件)，因为最终的文件只有一个，所以需要将这些溢写文件归并到一起，这个过程就叫做Merge。Merge是怎样的？如前面的例子，“aaa”从某个map task读取过来时值是5，从另外一个map 读取时值是8，因为它们有相同的key，所以得merge成group。什么是group。对于“aaa”就是像这样的：{“aaa”，[5，8，2，…]}，数组中的值就是从不同溢写文件中读取出来的，然后再把这些值加起来。请注意，因为merge是将多个溢写文件合并到一个文件，所以可能也有相同的key存在，在这个过程中如果client设置过Combiner，也会使用Combiner来合并相同的key。

至此，map端的所有工作都已结束，最终生成的这个文件也存放在TaskTracker够得着的某个本地目录内。每个reduce task不断地通过RPC从JobTracker那里获取map task是否完成的信息，如果reduce task得到通知，获知某台TaskTracker上的map task执行完成，shuffle的后半段过程开始启动。

4.4.1.7. ReduceTask具体执行流程(选看)

简单地说，reduce task在执行之前的工作就是不断地拉取当前job里每个map task的最终结果，然后对从不同地方拉取过来的数据不断地做merge，也最终形成一个文件作为reduce task的输入文件。见下图：



如map 端的细节图，shuffle在reduce端的过程也能用图上标明的三点来概括。reducer真正运行之前，所有的时间都是在拉取数据，做merge，且不断重复地在做。如前面的方式一样，下面我也分段地描述reduce 端的shuffle细节：

1）Copy过程，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map task所在的TaskTracker获取map task的输出文件。因为map task早已结束，这些文件就归TaskTracker管理在本地磁盘中。
每个reduce task都会有一个后台进程GetMapCompletionEvents，它获取heartbeat中（从JobTracker）传过来的已经完成的task列表，并将与该reduce task对应的数据位置信息保存到mapLocations中，mapLocations中的数据位置信息经过滤和去重（相同的位置信息因为某种原因，可能发过来多次）等处理后保存到集合scheduledCopies中，然后由几个拷贝线程（默认为5个）通过HTTP并行的拷贝数据，同时线程InMemFSMergeThread和LocalFSMerger会对拷贝过来的数据进行归并排序。

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，它基于JVM的heap size设置，因为shuffle阶段reducer不运行，所以应该把绝大部分的内存都给shuffle用。这里需要强调的是，merge有三种形式：1)内存到内存　2)内存到磁盘　3)磁盘到磁盘。默认情况下第一种形式不启用，让人比较困惑，是吧。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方式生成最终的那个文件。

3）Reducer的输入文件

不断地merge后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望它存放于内存中，直接作为reducer的输入，但默认情况下，这个文件是存放于磁盘中的。至于怎样才能让这个文件出现在内存中，之后的性能优化篇我再说。当reducer的输入文件已定，整个shuffle才最终结束。然后就是reducer执行，把结果放到HDFS上。

Hadoop处理流程中的两个子阶段严重降低了其性能。第一个是map阶段产生的中间结果要写到磁盘上，这样做的主要目的是提高系统的可靠性，但代价是降低了系统的性能，实际上，Hadoop的改进版-MapReduce Online去除了这个阶段，而采用其他更高效的方式提高系统可靠性；另一个是shuffle阶段采用HTTP协议从各个map task上远程拷贝结果，这种设计思

路（远程拷贝，协议采用http）同样降低了系统性能。实际上，Baidu公司正试图将该部分代码替换成C++代码来提高性能。

主要有两个方面影响shuffle阶段的性能：（1）数据完全是远程拷贝（2）采用HTTP协议进行数据传输。对于第一个方面，如果采用某种策略（修改框架），让你reduce task也能有locality就好了；对于第二个方面，用新的更快的数据传输协议替换HTTP，也许能更快些,如UDT协议

宏观上，Hadoop每个作业要经历两个阶段：Map phase和reduce phase。对于Map phase，又主要包含四个子阶段：从磁盘上读数据-》执行map函数-》combine结果-》将结果写到本地磁盘上；对于reduce phase，同样包含四个子阶段：从各个map task上读相应的数据（shuffle）-》sort-》执行reduce函数-》将结果写到HDFS中。

4.4.2. combiner函数

集群的可用带宽本来就很稀缺，因此在不影响结果数据的前提下，尽可能的减少磁盘IO和网络传输，是非常合适的。Hadoop允许用户针对map任务的输出指定一个combiner函数（其实是一个运行在map端的reduce函数），用于优化MR的执行效率。

特点总结：

1. Combiner是MR程序中Mapper和Reduce之外的一种组件

2. Combiner组件的父类就是Reducer

3. Combiner和Reducer之间的区别在于运行的位置

4. Reduce阶段的Reducer是每一个接收全局的Map Task 所输出的结果

5. Combiner是在合并排序后运行的。因此map端和reduce端都可以调用此函数。

6. Combiner的存在就是提高当前网络IO传输的性能，是MapReduce的一种优化手段。

7. Combiner在驱动类中的设置：

   ```
   job.setCombinerClass(MyCombiner.class);
   ```

注意：combiner不適合做求平均值这类需求，很可能就影响了结果。

4.4.3. MapReduce参数优化

4.4.3.1. 资源相关参数

以下参数是在用户自己的mr应用程序中配置在mapred-site.xml就可以生效

```
1. mapreduce.map.memory.mb: 一个Map Task可使用的资源上限（单位:MB），默认为1024。如果Map Task实际使用的资源量超过该值，则会被强制杀死。
2. mapreduce.reduce.memory.mb: 一个Reduce Task可使用的资源上限（单位:MB），默认为1024。如果Reduce Task实际使用的资源量超过该值，则会被强制杀死。
3. mapreduce.map.cpu.vcores: 每个Map task可使用的最多cpu core数目，默认值: 1
4. mapreduce.reduce.cpu.vcores: 每个Reduce task可使用的最多cpu core数目，默认值: 1
5. mapreduce.map.java.opts: Map Task的JVM参数，你可以在此配置默认的java heap size等参数.
   比如:
   -Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc” （@taskid@会被Hadoop框架自动换为相应的taskid），
     默认值: ""
6. mapreduce.reduce.java.opts: Reduce Task的JVM参数,可以在此配置默认的java heap size等参数.
   比如:
   “-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”，默认值: “”
```

下面的配置，应该在yarn启动之前就配置在服务器的yarn-site.xml配置文件中才能生效

```
7. yarn.scheduler.minimum-allocation-mb   1024   给应用程序container分配的最小内存
8. yarn.scheduler.maximum-allocation-mb   8192  给应用程序container分配的最大内存
9. yarn.scheduler.minimum-allocation-vcores     1
10. yarn.scheduler.maximum-allocation-vcores    32
11. yarn.nodemanager.resource.memory-mb   8192   每台NodeManager最大可用内存
12. yarn.nodemanager.resource.cpu-vcores    8    每台NodeManager最大可用cpu核数
```

shuffle性能优化的关键参数，应在yarn启动之前就配置好

```
13. mapreduce.task.io.sort.mb   100         //shuffle的环形缓冲区大小，默认100m
14. mapreduce.map.sort.spill.percent   0.8    //环形缓冲区溢出的阈值，默认80%
```

4.4.3.2. 容错相关参数

```
1. mapreduce.map.maxattempts: 每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
2. mapreduce.reduce.maxattempts: 每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。
3. mapreduce.map.failures.maxpercent: 当失败的Map Task失败比例超过该值时，整个作业则失败，默认值为0. 如果你的应用程序允许丢弃部分输入数据，则该该值设为一个大于0的值，比如5，表示如果有低于5%的Map Task失败（如果一个MapTask重试次数超过mapreduce.map.maxattempts，则认为这个Map Task失败，其对应的输入数据将不会产生任何结果），整个作业仍认为成功。

4. mapreduce.reduce.failures.maxpercent: 当失败的Reduce Task失败比例超过该值时，整个作业则失败，默认值为0.
5. mapreduce.task.timeout: Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该task处于block状态，可能是卡住了，也许永远会卡主，为了防止因为用户程序永远block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是300000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示
是“AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 sec
                sContainer killed by the ApplicationMaster.”。
```

4.4.3.3. 本地运行MapReduce作业

设置以下几个参数:

```
MapReduce.framework.name=local

fs.defaultFS=local(file:///)
```

4.4.3.4. 效率和稳定性相关参数

```
1) MapReduce.map.speculative: 是否为Map Task打开推测执行机制，默认为false

2) MapReduce.reduce.speculative: 是否为Reduce Task打开推测执行机制，默认为false

3) MapReduce.job.user.classpath.first & MapReduce.task.classpath.user.precedence：当同一个class同时出现在用户jar包和hadoop jar中时，优先使用哪个jar包中的class，默认为false，表示优先使用hadoop jar中的class

4) MapReduce.input.fileinputformat.split.minsize: FileInputFormat做切片时的最小切片大小

5) MapReduce.input.fileinputformat.split.maxsize:  FileInputFormat做切片时的最大切片大小(切片的默认大小就等于blocksize，即 134217728)
```

4.5. Hadoop 数据压缩【了解】

4.5.1. 概述

这是MapReduce的一种优化策略：通过压缩编码对mapper或者reducer的输出进行压缩，以减少磁盘IO，提高MR程序运行速度（但相应增加了cpu运算负担）

```
1）MapReduce支持将map输出的结果或者reduce输出的结果进行压缩，以减少网络IO或最终输出数据的体积
2）压缩特性运用得当能提高性能，但运用不当也可能降低性能
3）基本原则：
        -运算密集型的job，少用压缩
        -IO密集型的job，多用压缩
```

4.5.2. MR支持的压缩编码

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

考虑Hadoop应用处理的数据集比较大，因此需要借助压缩。下面是按照效率从高到低排列的

```
（1）使用容器格式文件，例如：顺序文件、RCFile、Avro数据格式支持压缩和切分文件。另外在配合使用一些快速压缩工具，例如：LZO、LZ4或者Snappy.
（2）使用支持切分压缩格式，例如bzip2
（3）在应用中将文件切分成块，对每块进行任意格式压缩。这种情况确保压缩后的数据接近HDFS块大小。
（4）存储未压缩文件，以原始文件存储。
```

4.5.3. Reducer输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1) 在配置参数中设置

```
压缩属性：MapReduce.output.fileoutputformat.compress     设置为true
压缩格式属性：MapReduce.output.fileoutputformat.compress.codec
        类库有：org.apache.hadoop.io.compress.DefaultCodec          .deflate
                org.apache.hadoop.io.compress.GzipCodec                          .gz
压缩类型属性：MapReduce.output.fileoutputformat.compress.type=RECORD
```

2) 也可以在代码中用

```
conf.set(“MapReduce.output.fileoutputformat.compress”,”true”)
```

3) 在代码中设置

```
Job job = Job.getInstance(conf);
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, (Class<? extends CompressionCodec>)
Class.forName(""));
```

4.5.4. Mapper输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1) 在配置参数中设置

```
MapReduce.map.output.compress=true

MapReduce.map.output.compress.codec=org.apache.hadoop.io.compress.DefaultCodec
```

2) 在代码中设置

```
conf.set(“MapReduce.map.output.compress”,” true”)
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class, CompressionCodec.class);
```

4.5.5. 压缩文件的读取

Hadoop自带的InputFormat类内置支持压缩文件的读取，比如TextInputformat类，在其initialize方法中：

```
public void initialize(InputSplit genericSplit,
                       TaskAttemptContext context) throws IOException {
    FileSplit split = (FileSplit) genericSplit;
    Configuration job = context.getConfiguration();
    this.maxLineLength = job.getInt(MAX_LINE_LENGTH, Integer.MAX_VALUE);
    start = split.getStart();
    end = start + split.getLength();
    final Path file = split.getPath();

    // open the file and seek to the start of the split
    final FileSystem fs = file.getFileSystem(job);
    fileIn = fs.open(file);
    //根据文件后缀名创建相应压缩编码的codec
    CompressionCodec codec = new CompressionCodecFactory(job).getCodec(file);
    if (null!=codec) {
        isCompressedInput = true;
        decompressor = CodecPool.getDecompressor(codec);
        //判断是否属于可切片压缩编码类型
        if (codec instanceof SplittableCompressionCodec) {
            final SplitCompressionInputStream cIn =
                ((SplittableCompressionCodec)codec).createInputStream(
                fileIn, decompressor, start, end,
                SplittableCompressionCodec.READ_MODE.BYBLOCK);
            //如果是可切片压缩编码，则创建一个CompressedSplitLineReader读取压缩数据
            in = new CompressedSplitLineReader(cIn, job,
                                               this.recordDelimiterBytes);
            start = cIn.getAdjustedStart();
            end = cIn.getAdjustedEnd();
            filePosition = cIn;
        } else {
            //如果是不可切片压缩编码，则创建一个SplitLineReader读取压缩数据，并将文件输入流转换成解压数
据流传递给普通SplitLineReader读取
            in = new SplitLineReader(codec.createInputStream(fileIn,
                                                             decompressor), job,
this.recordDelimiterBytes);
            filePosition = fileIn;
        }
    } else {
        fileIn.seek(start);
        //如果不是压缩文件，则创建普通SplitLineReader读取数据
        in = new SplitLineReader(fileIn, job, this.recordDelimiterBytes);
        filePosition = fileIn;
    }
}
```

五、案例实战

案例一：MR实战之小文件合并(自定义inputFormat)

1.1. 项目准备

1) 需求

无论hdfs还是MapReduce，对于小文件都有损效率，实践中，又难免面临处理大量小文件的场景，此时，就需要有相应解决方案

2) 测试数据

3) 分析

小文件的优化无非以下几种方式：

a）在数据采集的时候，就将小文件或小批数据合成大文件再上传HDFS

b）在业务处理之前，在HDFS上使用MapReduce程序对小文件进行合并

c）在MapReduce处理时，可采用combineInputFormat提高效率

1.2 项目实现

注意:本节实现的是上述第二种方式

1) 程序的核心机制：

自定义一个InputFormat

改写RecordReader，实现一次读取一个完整文件封装为KV

在输出时使用SequenceFileOutPutFormat输出合并文件

2) 代码如下

a) 自定义InputFromat

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class WholeFileInputFormat extends
    FileInputFormat<NullWritable, BytesWritable> {
    //设置每个小文件不可分片,保证一个小文件生成一个key-value键值对
    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
        InputSplit split, TaskAttemptContext context) throws IOException,
    InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
        return reader;
    }
}
```

b) 自定义RecordReader

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 自定义RecordReader
 */
public class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {
    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
        throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

    @Override
    public NullWritable getCurrentKey() throws IOException,InterruptedException {
        return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException {
        return processed ? 1.0f : 0.0f;
```

```
    }

    @Override
    public void close() throws IOException {
        // do nothing
    }
}
```

c) 定义MapReduce处理流程

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 定义MapReduce处理流程
 */
public class SmallFilesToSequenceFileConverter extends Configured implements
    Tool {
    static class SequenceFileMapper extends
        Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
        private Text filenameKey;

        @Override
        protected void setup(Context context) throws IOException,InterruptedException {
            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }

        @Override
        protected void map(NullWritable key, BytesWritable value,Context context) throws
IOException, InterruptedException        {
            context.write(filenameKey, value);
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        System.setProperty("HADOOP_USER_NAME", "hdfs");
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: combinefiles <in> <out>");
            System.exit(2);
        }

        Job job = Job.getInstance(conf,"combine small files to sequencefile");
        //      job.setInputFormatClass(WholeFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BytesWritable.class);
        job.setMapperClass(SequenceFileMapper.class);
        return job.waitForCompletion(true) ? 0 : 1;
```

```
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SmallFilesToSequenceFileConverter(),args);
        System.exit(exitCode);
    }
}
```

案例二: MR实战之数据分类输出(自定义outputFormat)

2.1 项目准备

1) 需求

现有一些原始日志需要做增强解析处理，流程：

a）从原始日志文件中读取数据

b）根据日志中的一个URL字段到外部知识库中获取信息增强到原始日志

c）如果成功增强，则输出到增强结果目录；如果增强失败，则抽取原始数据中URL字段输出到待爬清单目录

2) 测试数据

3) 分析

程序的关键点是要在一个MapReduce程序中根据数据的不同输出两类结果到不同目录，这类灵活的输出需求可以通过自定义outputformat来实现

2.2 项目实现

1. 实现要点

   a) 在MapReduce中访问外部资源

   b) 自定义outputformat，改写其中的recordwriter，改写具体输出数据的方法write()

2. 代码实现如下

   a) 数据库获取数据的工具

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class DBLoader {

    public static void dbLoader(HashMap<String, String> ruleMap) {
        Connection conn = null;
        Statement st = null;
        ResultSet res = null;

        try {
```

```
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://hdp-node01:3306/urlknowledge",
"root", "root");
            st = conn.createStatement();
            res = st.executeQuery("select url,content from urlcontent");
            while (res.next()) {
                ruleMap.put(res.getString(1), res.getString(2));
            }
        } catch (Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(res!=null){
                    res.close();
                }
                if(st!=null){
                    st.close();
                }
                if(conn!=null){
                    conn.close();
                }

            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }


    public static void main(String[] args) {
        DBLoader db = new DBLoader();
        HashMap<String, String> map = new HashMap<String,String>();
        db.dbLoader(map);
        System.out.println(map.size());
    }
}
```

b）自定义一个outputformat

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 自定义一个outputformat
 */
public class LogEnhancerOutputFormat extends FileOutputFormat<Text, NullWritable>{


    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext context) throws
IOException, InterruptedException {
```

```
        FileSystem fs = FileSystem.get(context.getConfiguration());
        Path enhancePath = new Path("hdfs://hdp-node01:9000/flow/enhancelog/enhanced.log");
        Path toCrawlPath = new Path("hdfs://hdp-node01:9000/flow/tocrawl/tocrawl.log");

        FSDataOutputStream enhanceOut = fs.create(enhancePath);
        FSDataOutputStream toCrawlOut = fs.create(toCrawlPath);


        return new MyRecordWriter(enhanceOut,toCrawlOut);
    }



    static class MyRecordWriter extends RecordWriter<Text, NullWritable>{

        FSDataOutputStream enhanceOut = null;
        FSDataOutputStream toCrawlOut = null;

        public MyRecordWriter(FSDataOutputStream enhanceOut, FSDataOutputStream toCrawlOut) {
            this.enhanceOut = enhanceOut;
            this.toCrawlOut = toCrawlOut;
        }

        @Override
        public void write(Text key, NullWritable value) throws IOException, InterruptedException
{

            //有了数据，你来负责写到目的地  —— hdfs
            //判断，进来内容如果是带tocrawl的，就往待爬清单输出流中写 toCrawlOut
            if(key.toString().contains("tocrawl")){
                toCrawlOut.write(key.toString().getBytes());
            }else{
                enhanceOut.write(key.toString().getBytes());
            }

        }

        @Override
        public void close(TaskAttemptContext context) throws IOException, InterruptedException {

            if(toCrawlOut!=null){
                toCrawlOut.close();
            }
            if(enhanceOut!=null){
                enhanceOut.close();
            }

        }


    }
}
```

c) 开发MapReduce处理流程

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 这个程序是对每个小时不断产生的用户上网记录日志进行增强(将日志中的url所指向的网页内容分析结果信息追加到每一行原始日志后面)
 */
public class LogEnhancer {

    static class LogEnhancerMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

        HashMap<String, String> knowledgeMap = new HashMap<String, String>();

        /**
         * maptask在初始化时会先调用setup方法一次 利用这个机制，将外部的知识库加载到maptask执行的机器内存中
         */
        @Override
        protected void setup(org.apache.hadoop.MapReduce.Mapper.Context context) throws IOException, InterruptedException {

            DBLoader.dbLoader(knowledgeMap);

        }

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

            String line = value.toString();

            String[] fields = StringUtils.split(line, "\t");

            try {
                String url = fields[26];

                // 对这一行日志中的url去知识库中查找内容分析信息
                String content = knowledgeMap.get(url);

                // 根据内容信息匹配的结果，来构造两种输出结果
                String result = "";
                if (null == content) {
                    // 输往待爬清单的内容
                    result = url + "\t" + "tocrawl\n";
                } else {
                    // 输往增强日志的内容
                    result = line + "\t" + content + "\n";
                }

                context.write(new Text(result), NullWritable.get());
            } catch (Exception e) {
```

```
            }
        }

    }

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

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);

        job.setJarByClass(LogEnhancer.class);

        job.setMapperClass(LogEnhancerMapper.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        // 要将自定义的输出格式组件设置到job中
        job.setOutputFormatClass(LogEnhancerOutputFormat.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));

        // 虽然我们自定义了outputformat，但是因为我们的outputformat继承自fileoutputformat
        // 而fileoutputformat要输出一个_SUCCESS文件，所以，在这还得指定一个输出目录
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
        System.exit(0);
    }
}
```

案例三: MR实战之TOPN(自定义GroupingComparator)

3.1 项目准备

1) 需求+测试数据

有如下订单数据

订单id	商品id	成交金额
Order_0000001	Pdt_01	222.8
Order_0000001	Pdt_05	25.8
Order_0000002	Pdt_03	522.8
Order_0000002	Pdt_04	122.4
Order_0000002	Pdt_05	722.4
Order_0000003	Pdt_01	222.8

现在需要求出每一个订单中成交金额最大的一笔交易

2. 分析

a) 利用“订单id和成交金额”作为key，可以将map阶段读取到的所有订单数据按照id分区，按照金额排序，发送到reduce

b) 在reduce端利用groupingcomparator将订单id相同的kv聚合成组，然后取第一个即是最大值

3.2 项目实现

a)自定义groupingcomparator

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 用于控制shuffle过程中reduce端对kv对的聚合逻辑
 */
public class ItemidGroupingComparator extends WritableComparator {

    protected ItemidGroupingComparator() {

        super(OrderBean.class, true);
    }


    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        OrderBean abean = (OrderBean) a;
        OrderBean bbean = (OrderBean) b;

        //将item_id相同的bean都视为相同，从而聚合为一组
        return abean.getItemid().compareTo(bbean.getItemid());
    }
}
```

b)定义订单信息bean

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 订单信息bean，实现hadoop的序列化机制
 */
public class OrderBean implements WritableComparable<OrderBean>{
    private Text itemid;
    private DoubleWritable amount;

    public OrderBean() {
    }
    public OrderBean(Text itemid, DoubleWritable amount) {
        set(itemid, amount);
    }

    public void set(Text itemid, DoubleWritable amount) {

        this.itemid = itemid;
        this.amount = amount;

    }

    public Text getItemid() {
        return itemid;
    }

    public DoubleWritable getAmount() {
        return amount;
    }

    @Override
    public int compareTo(OrderBean o) {
        int cmp = this.itemid.compareTo(o.getItemid());
        if (cmp == 0) {

            cmp = -this.amount.compareTo(o.getAmount());
        }
        return cmp;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(itemid.toString());
        out.writeDouble(amount.get());

    }

    @Override
    public void readFields(DataInput in) throws IOException {
        String readUTF = in.readUTF();
        double readDouble = in.readDouble();

        this.itemid = new Text(readUTF);
```

```
        this.amount= new DoubleWritable(readDouble);
    }


    @Override
    public String toString() {
        return itemid.toString() + "\t" + amount.get();
    }
}
```

c) 编写MapReduce处理流程

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 利用secondarysort机制输出每种item订单金额最大的记录
 */

public class SecondarySort {

    static class SecondarySortMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable>
{

        OrderBean bean = new OrderBean();

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {

            String line = value.toString();
            String[] fields = StringUtils.split(line, "\t");

            bean.set(new Text(fields[0]), new DoubleWritable(Double.parseDouble(fields[1])));

            context.write(bean, NullWritable.get());

        }

    }

    static class SecondarySortReducer extends Reducer<OrderBean, NullWritable, OrderBean,
NullWritable>{


        //在设置了groupingcomparator以后，这里收到的kv数据 就是:  <1001 87.6>,null  <1001
76.5>,null  ....
        //此时，reduce方法中的参数key就是上述kv组中的第一个kv的key：<1001 87.6>
        //要输出同一个item的所有订单中最大金额的那一个，就只要输出这个key
        @Override
        protected void reduce(OrderBean key, Iterable<NullWritable> values, Context context)
throws IOException, InterruptedException {
            context.write(key, NullWritable.get());
```

```
        }
    }


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

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(SecondarySort.class);

        job.setMapperClass(SecondarySortMapper.class);
        job.setReducerClass(SecondarySortReducer.class);


        job.setOutputKeyClass(OrderBean.class);
        job.setOutputValueClass(NullWritable.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //指定shuffle所使用的GroupingComparator类
        job.setGroupingComparatorClass(ItemidGroupingComparator.class);
        //指定shuffle所使用的partitioner类
        job.setPartitionerClass(ItemIdPartitioner.class);

        job.setNumReduceTasks(3);

        job.waitForCompletion(true);

    }

}
```

案例四: MR实战之计数器应用

4.1 项目准备

在实际生产代码中，常常需要将数据处理过程中遇到的不合规数据行进行全局计数，类似这种需求可以借助MapReduce框架中提供的全局计数器来实现

4.2 项目实现

示例代码如下：

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 计数器应用
 */
public class MultiOutputs {
    //通过枚举形式定义自定义计数器
    enum MyCounter{MALFORORMED,NORMAL}
```

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

            String[] words = value.toString().split(",");

            for (String word : words) {
                context.write(new Text(word), new LongWritable(1));
            }
            //对枚举定义的自定义计数器加1
            context.getCounter(MyCounter.MALFORORMED).increment(1);
            //通过动态设置自定义计数器加1
            context.getCounter("counterGroupa", "countera").increment(1);
    }
}
```

案例五: MR实战之多job串联

5.1 项目准备

一个稍复杂点的处理逻辑往往需要多个MapReduce程序串联处理，多job的串联可以借助MapReduce框架的JobControl实现

5.2 项目实现

示例代码：

```
ControlledJob cJob1 = new ControlledJob(job1.getConfiguration());
ControlledJob cJob2 = new ControlledJob(job2.getConfiguration());
ControlledJob cJob3 = new ControlledJob(job3.getConfiguration());

cJob1.setJob(job1);
cJob2.setJob(job2);
cJob3.setJob(job3);

// 设置作业依赖关系
cJob2.addDependingJob(cJob1);
cJob3.addDependingJob(cJob2);

JobControl jobControl = new JobControl("RecommendationJob");
jobControl.addJob(cJob1);
jobControl.addJob(cJob2);
jobControl.addJob(cJob3);


// 新建一个线程来运行已加入JobControl中的作业，开始进程并等待结束
Thread jobControlThread = new Thread(jobControl);
jobControlThread.start();
while (!jobControl.allFinished()) {
    Thread.sleep(500);
```

```
}
jobControl.stop();

return 0;
```

案例六: MR实战之reduce端join算法实现

6.1 项目准备

1) 需求:

订单数据表t_order:

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

商品信息表t_product

id	pname	category_id	price
P0001	小米5	1000	2
P0002	锤子T1	1000	3

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程序来实现一下SQL查询运算

6.2 项目实现

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

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description reduce端join算法实现
 */
public class OrderJoin {
    static class OrderJoinMapper extends Mapper<LongWritable, Text, Text, OrderJoinBean> {
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            //拿到一行数据，并且要分辨出这行数据所属的文件
            String line = value.toString();
```

```
            String[] fields = line.split("\t");

            //拿到itemid
            String itemid = fields[0];

            //获取到这一行所在的文件名（通过inpusplit）
            String name = "你拿到的文件名";

            //根据文件名，切分出各字段（如果是a，切分出两个字段，如果是b，切分出3个字段）

            OrderJoinBean bean = new OrderJoinBean();
            bean.set(null, null, null, null, null);
            context.write(new Text(itemid), bean);
        }
    }

    static class OrderJoinReducer extends Reducer<Text, OrderJoinBean, OrderJoinBean,
NullWritable> {

        @Override
        protected void reduce(Text key, Iterable<OrderJoinBean> beans, Context context) throws
IOException, InterruptedException {
            //拿到的key是某一个itemid,比如1000
            //拿到的beans是来自于两类文件的bean
            //  {1000,amount} {1000,amount} {1000,amount}   ---   {1000,price,name}

            //将来自于b文件的bean里面的字段，跟来自于a的所有bean进行字段拼接并输出
        }
    }
}
```

缺点：这种方式中，join的操作是在reduce阶段完成，reduce端的处理压力太大，map节点的运算负载则很低，资源利用率不高，且在reduce阶段极易产生数据倾斜

解决方案： map端join实现方式

案例七: MR实战之map端join算法实现

7.1 项目准备

适用于关联表中有小表的情形；

可以将小表分发到所有的map节点，这样，map节点就可以在本地对自己所读到的大表数据进行join并输出最终结果，可以大大提高join操作的并发度，加快处理速度

注意点

--先在mapper类中预先定义好小表，进行join

--引入实际场景中的解决方案：一次加载数据库或者用distributedcache

适用场景：一个大表join一个小表

相关表见 案例六 MapReduce实战之reduce端join算法实现

7.2. 项目实现

1) 将小表先准备在一个hdfs的目录中

2) 在代码的main方法中用job.addCacheFile()将其分发到maptask的工作目录下；还需要将reduce task的数量设置为0

3) 在代码的mapper的setup方法中用本地文件api读取小表文件到内存中

4) 在map方法中根据输入数据匹配内存小表进行拼接即可

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description map端join算法实现
 */
public class TestDistributedCache {
    static class TestDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text>{
        FileReader in = null;
        BufferedReader reader = null;
        HashMap<String,String> b_tab = new HashMap<String, String>();
        String localpath =null;
        String uirpath = null;

        //是在map任务初始化的时候调用一次
        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            //通过这几句代码可以获取到cache file的本地绝对路径，测试验证用
            Path[] files = context.getLocalCacheFiles();
            localpath = files[0].toString();
            URI[] cacheFiles = context.getCacheFiles();


            //缓存文件的用法—直接用本地IO来读取
            //这里读的数据是map task所在机器本地工作目录中的一个小文件
            in = new FileReader("b.txt");
            reader =new BufferedReader(in);
            String line =null;
```

```
            while(null!=(line=reader.readLine())){
                String[] fields = line.split(",");
                b_tab.put(fields[0],fields[1]);
            }
            IOUtils.closeStream(reader);
            IOUtils.closeStream(in);
        }

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            //这里读的是这个map task所负责的那一个切片数据（在hdfs上）
            String[] fields = value.toString().split("\t");

            String a_itemid = fields[0];
            String a_amount = fields[1];
            String b_name = b_tab.get(a_itemid);

            // 输出结果  1001   98.9    banan
            context.write(new Text(a_itemid), new Text(a_amount + "\t" + ":" + localpath + "\t"
+b_name ))
        }
    }

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

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(TestDistributedCache.class);

        job.setMapperClass(TestDistributedCacheMapper.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);

        //这里是我们正常的需要处理的数据所在路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //不需要reducer
        job.setNumReduceTasks(0);
    }
```

案例八: MR实战之web日志预处理

8.1 项目准备

1) 需求

对web访问日志中的各字段识别切分

去除日志中不合法的记录

根据KPI统计需求，生成各类访问请求过滤数据

8.2 项目实现

1) 定义一个bean，用来记录日志数据中的各数据字段

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 定义一个bean，用来记录日志数据中的各数据字段
 */
public class WebLogBean {
    private String remote_addr;// 记录客户端的ip地址
    private String remote_user;// 记录客户端用户名称,忽略属性"-"
    private String time_local;// 记录访问时间与时区
    private String request;// 记录请求的url与http协议
    private String status;// 记录请求状态；成功是200
    private String body_bytes_sent;// 记录发送给客户端文件主体内容大小
    private String http_referer;// 用来记录从那个页面链接访问过来的
    private String http_user_agent;// 记录客户浏览器的相关信息
    private boolean valid = true;// 判断数据是否合法

    public String getRemote_addr() {
        return remote_addr;
    }

    public void setRemote_addr(String remote_addr) {
        this.remote_addr = remote_addr;
    }

    public String getRemote_user() {
        return remote_user;
    }

    public void setRemote_user(String remote_user) {
        this.remote_user = remote_user;
    }

    public String getTime_local() {
        return time_local;
    }

    public void setTime_local(String time_local) {
        this.time_local = time_local;
    }

    public String getRequest() {
        return request;
    }

    public void setRequest(String request) {
        this.request = request;
    }
```

```
    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public String getBody_bytes_sent() {
        return body_bytes_sent;
    }

    public void setBody_bytes_sent(String body_bytes_sent) {
        this.body_bytes_sent = body_bytes_sent;
    }

    public String getHttp_referer() {
        return http_referer;
    }
    public void setHttp_referer(String http_referer) {
        this.http_referer = http_referer;
    }

    public String getHttp_user_agent() {
        return http_user_agent;
    }

    public void setHttp_user_agent(String http_user_agent) {
        this.http_user_agent = http_user_agent;
    }

    public boolean isValid() {
        return valid;
    }

    public void setValid(boolean valid) {
        this.valid = valid;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(this.valid);
        sb.append("\001").append(this.remote_addr);
        sb.append("\001").append(this.remote_user);
        sb.append("\001").append(this.time_local);
        sb.append("\001").append(this.request);
        sb.append("\001").append(this.status);
        sb.append("\001").append(this.body_bytes_sent);
        sb.append("\001").append(this.http_referer);
        sb.append("\001").append(this.http_user_agent);
        return sb.toString();
```

```
    }
}
```

2) 定义一个parser用来解析过滤web访问日志原始记录

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 定义一个parser用来解析过滤web访问日志原始记录
 */
public class WebLogParser {
    public static WebLogBean parser(String line) {
        WebLogBean webLogBean = new WebLogBean();
        String[] arr = line.split(" ");
        if (arr.length > 11) {
            webLogBean.setRemote_addr(arr[0]);
            webLogBean.setRemote_user(arr[1]);
            webLogBean.setTime_local(arr[3].substring(1));
            webLogBean.setRequest(arr[6]);
            webLogBean.setStatus(arr[8]);
            webLogBean.setBody_bytes_sent(arr[9]);
            webLogBean.setHttp_referer(arr[10]);

            if (arr.length > 12) {
                webLogBean.setHttp_user_agent(arr[11] + " " + arr[12]);
            } else {
                webLogBean.setHttp_user_agent(arr[11]);
            }
            if (Integer.parseInt(webLogBean.getStatus()) >= 400) {// 大于400, HTTP错误
                webLogBean.setValid(false);
            }
        } else {
            webLogBean.setValid(false);
        }
        return webLogBean;
    }

    public static String parserTime(String time) {
        time.replace("/", "-");
        return time;
    }
}
```

3) MapReduce程序

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class WeblogPreProcess {
```

```
    static class WeblogPreProcessMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
        Text k = new Text();
        NullWritable v = NullWritable.get();

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {

            String line = value.toString();
            WebLogBean webLogBean = WebLogParser.parser(line);
            if (!webLogBean.isValid())
                return;
            k.set(webLogBean.toString());
            context.write(k, v);
        }
    }

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

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(WeblogPreProcess.class);

        job.setMapperClass(WeblogPreProcessMapper.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}
```

六、内容总结

MapReduce是Hadoop中的编程模型，通过这个编程模型，我们可以在不用掌握分布式计算的情况下，编写出分布式计算程序。在MapReduce中，我们需要重点掌握以下内容：

1. MapReduce入门
    1.1. MapReduce的阶段分类
    1.2. MapReduce的编程规范
    1.3. MapReduce的Partitioner
    1.4. IDE运行MapReduce的模式
2. Hadoop的序列化机制
    2.1. 什么是序列化，为什么要序列化
    2.2. Hadoop的序列化机制
    2.3. Writable & WritableComparable
3. MapReduce核心理论

七、课后作业

1. wordcount

统计单词出现的次数，将a-n开头的单词放入一个文件，o-z开头的单词放入一个文件，其他的放入一个文件。

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
 * Map阶段的逻辑：
 *
 * 1. 需要自定义一个类，继承自Mapper类
 *    org.apache.hadoop.mapreduce.Mapper类，Hadoop2.x版本就开始使用这个类
 *    org.apache.hadoop.mapred.Mapper，是一个Hadoop1.x版本使用的接口
 *
 * 2. 设置Mapper的泛型，分别是K1，V1，K2，V2
 *    K1：表示的是逐行读取数据的行偏移量(其实是第多少个字符)，是LongWritable类型的
 *    V1：表示的是读取到的一行的数据，是Text类型的
 *
 * 3. 对于Wordcount程序来说：
 *    K2：表示需要统计的单词，是一个字符串，在这里，使用Text来表示
 *    V2：表示的是与单词组合到一起的1，是一个整型数字，在这里使用IntWritable
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class WordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    /**
     * 在这里，处理MapTask的逻辑
     * 在读取数据的时候，不用关心分布式的程序是如何运行的，需要知道的是数据是逐行读取的
     * 每当读取到一行的数据，形成了K1V1键值对，就会调用这个方法
     * @param key Map的入口数据，就是K1，在这里表示行偏移量(第多少个字符)
     * @param value Map的入口数据，就是V1，在这里表示读取的一行数据
     * @param context 操作上下文，使用它将数据写出(写K2V2)
     * @throws IOException IO异常
     * @throws InterruptedException 中断处理异常
     */
    @Override
```

```
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        /*
         * 需要实现的业务逻辑:
         * 将读取到的每一行的数据按照空格切分，得出每一个单词
         * 再将单词与1组成键值对，写出
         */

        /*
         * 1. 将Text转成String类型，进行切割
         */
        String line = value.toString();
        /*
         * 2. 切割出每一个单词
         */
        String[] words = line.split("\\s+");
        /*
         * 3. 遍历每一个单词
         */
        for (String word : words) {
            /*
             * 4. 将单词封装成K2的类型，也就是Text类型
             */
            Text k2 = new Text(word);
            /*
             * 5. 将1封装成V2的类型，也就是IntWritable类型
             */
            IntWritable v2 = new IntWritable(1);
            /*
             * 6. 将K2V2写出(到Shuffle缓冲区)
             */
            context.write(k2, v2);
        }
    }
}
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * Reduce阶段的逻辑:
 *
 * 1. 需要自定义一个类，继承自Reducer类
 *    org.apache.hadoop.mapreduce.Reducer类，Hadoop2.x版本就开始使用这个类
 *    org.apache.hadoop.mapred.Reducer，是一个Hadoop1.x版本使用的接口
 *
 * 2. 这个类中，有四个泛型，分别表示K2，V2，K3，V3
 *    K2: 就是上游Map的输出结果类型，在这里就是Text类型
 *    V2: 就是上游Map的输出结果类型，在这里就是IntWritable类型
```

```
 *    K3: 就是最终输出到文件的结果类型，在这里就是Text类型
 *    V3: 就是最终输出到文件的结果类型，在这里就是IntWriable类型
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class WordcountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    /**
     * reduce方法的逻辑实现
     * 在reduce阶段，reduceTask会跨节点Fetch自己需要的数据到自己的节点，进行业务的处理
     * 在reduce方法执行之前，会经过一个Shuffle阶段
     * 会将Map阶段输出的结果中，所有的相同的Key的value聚合到一起
     * 在本案例中，将所有的单词出现的1聚合到了一起
     * @param key 上游Task输出的K2，这里就表示单词
     * @param values 由若干个V2组成的一个可迭代序列
     * @param context 操作上下文
     * @throws IOException IO异常
     * @throws InterruptedException 中断处理异常
     */
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        /*
         * 在本案例中，进入到reduce的数据，其实是每一个单词与这个单词所有的出现次数的集合，组成的键值对
         * 因此，在这里，就只需要统计这个单词出现的总次数即可
         */
        /*
         * 1. 设置一个变量，用于累加单词出现的次数
         */
        int sum = 0;
        /*
         * 2. 遍历这个单词出现的所有的次数集合
         */
        for (IntWritable value : values) {
            /*
             * 3. 将每一个出现的次数累加到sum中
             */
            sum += value.get();
        }
        /*
         * 4. 将单词与其出现的次数组合成键值对
         */
        context.write(key, new IntWritable(sum));
    }
}
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

/**
 * 自定义的分区器：逻辑，将a-n开头的单词放入一个分区，o-z开头的单词放入另外一个分区
```

```
 *
 * 分区器:
 *     是将Map的输出结果分为不同的逻辑分区，每一个分区的数据都有一个ReduceTask进行处理
 *     默认的分区器，是HashPartitioner
 *
 * 自定义分区器:
 *     设计一个类，继承自org.apache.hadoop.mapreduce.Partitioner，设置泛型为K2V2
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class WordcountPartitioner extends Partitioner<Text, IntWritable> {

    /**
     * 根据数据，返回不同的分区编号
     * 每一个分区，都有一个编号，是从0开始的一个递增的数字
     * @param text K2
     * @param intWritable V2
     * @param i 分区的数量
     * @return 分区的编号
     */
    @Override
    public int getPartition(Text text, IntWritable intWritable, int i) {
        /*
         * 1. 获取单词的首字母
         */
        char first = text.toString().charAt(0);
        /*
         * 2. 设置不同的分区
         */
        if (first >= 'a' && first <= 'n') {
            return 0;
        }
        return 1;
    }
}
```

```
/**
 * 这个类，其实就是MapReduce程序的客户端，负责配置作业以及提交作业
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class WordcountDriver {
    public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
        /*
         * 1. 获取集群的信息，通过配置文件获取
         *    将四大默认配置文件拷贝到项目的resources文件夹中，Configuration对象在实例化的时候会自动的
加载
         */
        Configuration configuration = new Configuration();
```

```
/*
 * 如果需要将mapreduce程序跑在yarn上，需要将mapreduce.framework.name设置为yarn
 * 由于我们已经将mapred-site.xml文件放入到了资源文件夹中，并且设置的值就是yarn
 * 所以，代码不用设置，直接去加载这个文件即可
 */
/*
 * 2. 配置操作Hadoop的用户
 */
System.setProperty("HADOOP_USER_NAME", "root");
/*
 * 3. 获取job对象
 */
Job job = Job.getInstance(configuration);
/*
 * 4. 设置Job的驱动类型
 */
job.setJarByClass(WordcountDriver.class);
/*
 * 5. 设置Mapper类
 */
job.setMapperClass(WordcountMapper.class);
/*
 * 6. 设置Reducer类
 */
job.setReducerClass(WordcountReducer.class);

/*
 * 7. 设置Map的输出类型K2V2
 *    如果Map的输出类型与Reduce的输入类型相同，则这里可以省略不写
 */
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);

/*
 * 8. 设置Reduce的输出类型K3V3
 */
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

/*
 * 9. 设置数据的输入路径
 */
FileInputFormat.setInputPaths(job, new Path(args[0]));

/*
 * 10. 设置数据的输出路径(这个输出路径需要不存在)
 */
FileSystem fileSystem = FileSystem.get(configuration);
if (fileSystem.exists(new Path(args[1]))) {
    fileSystem.delete(new Path(args[1]), true);
}
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

```
        /*
         * 设置ReduceTask的个数
         * 这里的个数设置需要和分区器中的逻辑保持一致，分区器中能处理多少个分区的数据，这里就设置多少个
reduceTask
         */
        job.setNumReduceTasks(2);

        /*
         * 11. 设置自定义的分区器
         */
        job.setPartitionerClass(WordcountPartitioner.class);

        /*
         * 12. 提交job
         */
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

2. 手机流量统计

需求:

1. 统计每一个手机号码或者固化号码的总上行流量，总下行流量以及总流量
2. 最后的结果分为两个文件进行存储，一个文件存储所有的手机号码，一个文件存储所有的固话号码

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class PhoneFlowMapper extends Mapper<LongWritable, Text, Text, PhoneFlowWritable> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        // 1. 将每一行的数据，按照TAB进行切割，得出每一个部分
        String[] split = value.toString().split("\t");
        // 2. 得到手机号码
        String phoneNumber = split[1];
        // 3. 得到上行和下行流量
        int upflow = Integer.parseInt(split[split.length - 3]);
        int downflow = Integer.parseInt(split[split.length - 2]);
        // 4. 实例化一个用来描述流量的类对象
        PhoneFlowWritable phoneFlowWritable = new PhoneFlowWritable(upflow, downflow);
```

```
        // 5. 将K2V2输出
        context.write(new Text(phoneNumber), phoneFlowWritable);
    }
}
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class PhoneFlowReducer extends Reducer<Text, PhoneFlowWritable, Text, PhoneFlowWritable>
{
    @Override
    protected void reduce(Text key, Iterable<PhoneFlowWritable> values, Context context) throws
IOException, InterruptedException {
        // 1. 设置两个变量，分别累加所有的这个手机号码的上行流量和下行流量
        int sumUp = 0, sumDown = 0;
        // 2. 遍历所有的PhoneFlowWritable
        for (PhoneFlowWritable value : values) {
            sumUp += value.upFlow;
            sumDown += value.downFlow;
        }
        // 3. 将结果输出
        context.write(key, new PhoneFlowWritable(sumUp, sumDown));
    }
}
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

/**
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class PhoneFlowPartitioner extends Partitioner<Text, PhoneFlowWritable> {
    @Override
    public int getPartition(Text text, PhoneFlowWritable phoneFlowWritable, int i) {
         return text.toString().length() == 11 ? 0 : 1;
    }
}
```

```
import com.sun.corba.se.spi.ior.Writeable;
import org.apache.hadoop.io.Writable;
import org.omg.CORBA_2_3.portable.OutputStream;
```

```java
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

/**
 * 用来描述手机流量，包含上行流量和下行流量
 * 这个类型，被用来当作V2和V3使用的，因此必须要实现序列化机制
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class PhoneFlowWritable implements Writable {

    /**
     * 上行流量
     */
    int upFlow;
    /*
     * 下行流量
     */
    int downFlow;

    public PhoneFlowWritable() {
    }

    public PhoneFlowWritable(int upFlow, int downFlow) {
        this.upFlow = upFlow;
        this.downFlow = downFlow;
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeInt(upFlow);
        dataOutput.writeInt(downFlow);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        this.upFlow = dataInput.readInt();
        this.downFlow = dataInput.readInt();
    }

    @Override
    public String toString() {
        return "upFlow=" + upFlow +
                ", downFlow=" + downFlow +
                ", totalFlow =" + (upFlow + downFlow);
    }
}
```

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 * 需求:
 *     1. 统计每一个手机号码或者固化号码的总上行流量，总下行流量以及总流量
 *     2. 最后的结果分为两个文件进行存储，一个文件存储所有的手机号码，一个文件存储所有的固话号码
 *
 * 需求分析:
 *     1. K1: 行偏移量，V1: 行记录
 *     2. K2: 手机号码，V2: 自定义的用来描述流量的类
 *     3. K3: 手机号码，V3: 自定义的用来描述流量的类
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class PhoneflowDriver {
    public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);
        job.setJarByClass(PhoneflowDriver.class);
        job.setMapperClass(PhoneFlowMapper.class);
        job.setReducerClass(PhoneFlowReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(PhoneFlowWritable.class);

        FileInputFormat.setInputPaths(job, new Path("C:\\Users\\shawn\\Desktop\\The required
Data\\HTTP_20130313143750.dat"));
        FileOutputFormat.setOutputPath(job, new Path("C:\\Users\\shawn\\Desktop\\output"));

        job.setNumReduceTasks(2);
        job.setPartitionerClass(PhoneFlowPartitioner.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

3. 最高温度统计

从给定的数据中，计算每一年的最高温度是多少
温度数据的说明：
1. 每行数据的 [15,18] 位是年份
2. 每行的第87个字符，代表温度的符号（正负）
3. 每行的第 [88,91] 位代表温度的值，如果温度是9999代表无效温度
4. 每行的第92位是一个校验位，如果是0，1，4，5，9代表有效温度

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
 * 0029029070999991901010106004+64333+023450FM-12+000599999V0202701N015919999999N0000001N9-
00781+99999102001ADDGF108991999999999999999999
 *
 * 从给定的数据中，计算每一年的最高温度是多少
 * 温度数据的说明：
 * 1. 每行数据的 [15,18] 位是年份
 * 2. 每行的第87个字符，代表温度的符号（正负）
 * 3. 每行的第 [88,91] 位代表温度的值，如果温度是9999代表无效温度
 * 4. 每行的第92位是一个校验位，如果是0，1，4，5，9代表有效温度
 *  *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class TempratureMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private Text k2;
    private IntWritable v2;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        k2 = new Text();
        v2 = new IntWritable();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        // 需要处理的逻辑：
        // 1. 从字符串中解析出来每一个需要的数据：年份、温度、校验
        // 2. 将年份作为K2，将温度作为V2，写出到环形缓冲区中

        String line = value.toString();
        // 1. 获取年份
        String year = line.substring(15, 19);
        // 2. 获取温度
        int temprature = Integer.parseInt(line.substring(87, 92));
        // 3. 获取校验位
        String check = line.substring(92, 93);
```

```
        // 4. 验证温度是否是一个合法的温度
        if (Math.abs(temprature) == 9999 || check.matches("[^01459]")) {
            // 表示不合法的温度
            return;
        }

        // 5. 所有的校验都没问题了，将年份和温度写出
        k2.set(year);
        v2.set(temprature);
        context.write(k2, v2);
    }
}
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class TempratureReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable v3;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        v3 = new IntWritable();
    }

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        // 计算每一个年份的最高温度
        int max = Integer.MIN_VALUE;
        // 遍历所有的温度
        for (IntWritable value : values) {
            max = Math.max(max, value.get());
        }
        // 将年份和最高温度输出
        v3.set(max);
        context.write(key, v3);
    }
}
```

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 *
 * K1: 行偏移量
 * V1: 行记录
 *
 * K2: 年份, Text、IntWritable
 * V2: 温度, IntWritable
 *
 * K3: 年份, Text、IntWritable
 * V3: 温度, IntWritable
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class TempratureDriver {
    public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
        System.setProperty("HADOOP_USER_NAME", "root");
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        job.setJarByClass(TempratureDriver.class);
        job.setMapperClass(TempratureMapper.class);
        job.setReducerClass(TempratureReducer.class);
        job.setCombinerClass(TempratureReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.setInputPaths(job, new Path("C:\\Users\\zhangxiangchao\\Desktop\\The
required Data\\温度数据\\温度数据"));
        FileOutputFormat.setOutputPath(job, new
Path("C:\\Users\\zhangxiangchao\\Desktop\\output"));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4. topN

通过每一个用于对每一个电影的评分文件中，解析出每一个用户所评的电影中的Top5

```
import org.apache.hadoop.io.Writable;
```

```java
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

/**
 * 一个用户对一个电影的评分
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class RatingModel implements Writable {
    private String movie;
    private int rate;
    private String datetime;
    private String uid;

    public String getMovie() {
        return movie;
    }

    public void setMovie(String movie) {
        this.movie = movie;
    }

    public int getRate() {
        return rate;
    }

    public void setRate(String rate) {
        this.rate = Integer.parseInt(rate);
    }

    public String getDatetime() {
        return datetime;
    }

    public void setDatetime(String datetime) {
        this.datetime = datetime;
    }

    public String getUid() {
        return uid;
    }

    public void setUid(String uid) {
        this.uid = uid;
    }

    public RatingModel deepcopy() {
        RatingModel model = new RatingModel();
        model.movie = this.movie;
        model.rate = this.rate;
```

```
        model.datetime = this.datetime;
        model.uid = this.uid;

        return model;
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeUTF(movie);
        dataOutput.writeInt(rate);
        dataOutput.writeUTF(datetime);
        dataOutput.writeUTF(uid);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        this.movie = dataInput.readUTF();
        this.rate = dataInput.readInt();
        this.datetime = dataInput.readUTF();
        this.uid = dataInput.readUTF();
    }

    @Override
    public String toString() {
        return "RatingModel{" +
                "movie='" + movie + '\'' +
                ", rate=" + rate +
                ", datetime='" + datetime + '\'' +
                ", uid='" + uid + '\'' +
                '}';
    }
}
```

```
import com.qf.mapreduce.temprature.TempratureDriver;
import com.qf.mapreduce.temprature.TempratureMapper;
import com.qf.mapreduce.temprature.TempratureReducer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.codehaus.jackson.map.ObjectMapper;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
```

```java
/**
 * 需求：通过每一个用于对每一个电影的评分文件中，解析出每一个用户所评的电影中的Top5
 *      C:\Users\zhangxiangchao\Desktop\The required Data\rating.json
 *
 * 分析：
 *      K1：行偏移量，V1：行记录（一个用户对一个电影的评分）
 *      K2：用户id，Text，V2：将这个用户对一个电影的评分，制作一个类（电影id和电影的评分）
 *      K3：用户id，Text，V3：这个用户评的电影评分的前5名，可以是一个自定义的类
 *
 * @author shawn
 * @company 北京千锋互联科技有限公司-大数据教学部
 */
public class TopN {

    public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
        System.setProperty("HADOOP_USER_NAME", "root");
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        job.setJarByClass(TopN.class);
        job.setMapperClass(RatingMapper.class);
        job.setReducerClass(RatingReducer.class);
        job.setCombinerClass(RatingReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(RatingModel.class);

        FileInputFormat.setInputPaths(job, new Path("C:\\Users\\zhangxiangchao\\Desktop\\The
required Data\\rating.json"));
        FileOutputFormat.setOutputPath(job, new
Path("C:\\Users\\zhangxiangchao\\Desktop\\output"));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }

    private static class RatingMapper extends Mapper<LongWritable, Text, Text, RatingModel> {
        private Text k2;
        // JSON的解析使用到这个类，可以将一个json字符串解析为一个对象
        private ObjectMapper objectMapper;

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            k2 = new Text();
            objectMapper = new ObjectMapper();
        }

        // 逻辑：解析一行的数据，将uid作为键，将用户的评分信息封装成RatingModel作为值
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            // 读取一行数据，解析为指定类的对象
```

```
            RatingModel ratingModel = objectMapper.readValue(value.toString(),
RatingModel.class);
            k2.set(ratingModel.getUid());

            context.write(k2, ratingModel);
        }
    }

    private static class RatingReducer extends Reducer<Text, RatingModel, Text, RatingModel> {
        @Override
        protected void reduce(Text key, Iterable<RatingModel> values, Context context) throws
IOException, InterruptedException {
            // 1. 实例化一个集合
            ArrayList<RatingModel> list = new ArrayList<>();
            for (RatingModel value : values) {
                list.add(value.deepcopy());
            }
            // 2. 排序
            list.sort((m1, m2) -> m2.getRate() - m1.getRate());
            // 3. 取top5
            int count = Math.min(5, list.size());
            for (int i = 0; i < count; i++) {
                context.write(key, list.get(i));
            }
        }
    }
}
```

八、解决方案

1. mapreduce二次排序

> 待排序的数据具有多个字段，首先对第一个字段进行排序，第一个字段相同的情况下，再按照第二个字段进行排序，第二次排序不会破坏第一次排序的结果。这个过程称之为二次排序

原始数据如下：

```
hadoop@apache	200
hive@apache	550
yarn@apache	580
hive@apache	159
hadoop@apache	300
hive@apache	258
hadoop@apache	300
yarn@apache	100
hadoop@apache	150
yarn@apache	560
yarn@apache	260
```

输出结果数据如下：

```
hadoop@apache	150
hadoop@apache	200
hadoop@apache	300
hadoop@apache	300
hive@apache	159
hive@apache	258
hive@apache	550
yarn@apache	100
yarn@apache	260
yarn@apache	560
yarn@apache	580
```

第一种方式：简答粗暴

> 第一个字段在规约到reduce端的reduce函数之前排好序。而我们只需要在进入reduce函数后，对第二个字段进行再次排序即可。如下代码：

```
@Override
public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {

     List<Integer> valuesList = new ArrayList<Integer>();

     // 取出value
     for(IntWritable value : values) {
         valuesList.add(value.get());
     }
```

```
        // 进行排序
        Collections.sort(valuesList);

        for(Integer value : valuesList) {
           context.write(key, new IntWritable(value));
        }

    }
```

> 但是，如果把排序工作都放到reduce端完成，当values序列长度非常大时，回对cpu和内存造成极大的负载。

第二种方式：

> 将map端输出的<key,value>中的key和value组合成一个新的key（称为newKey），value值不变。这里就变成<(key,value),value>，在针对newKey排序的时候，如果key相同，就再对value进行排序。 需要自定义的地方
>
> 1. 自定义数据类型实现组合key
>   实现方式：继承WritableComparable
> 2. 自定义partioner，形成newKey后保持分区规则仍然按照key进行。保证不打乱原来的分区。
>   实现方式：继承partitioner

1. 自定义分组，保持分组规则仍然按照key进行。不打乱原来的分组
   实现方式：继承RawComparator

2. 什么是MapReduce

```
1. MapReduce是Hadoop的一个核心技术、是一个基于分布式的对大数据集进行并行处理的一个计算框架。
2. 核心思想是移动计算而非数据。
3. 整个计算流程分为两个阶段，一个是map阶段，一个是reduce阶段
    - map阶段
        一个mapreduce作业在map阶段会先对数据进行逻辑分片处理，一个逻辑分片对应一个MapTask。每一个MapTask在处理数据的时候，都会将每一行数据解析成键值对<k1，v1>的形式，作为输入。输入给map函数。map函数处理后，也会以键值对<k2，v2>的形式作为输出。之后会对数据进行分区，排序等处理
    - reduce阶段
            在进入reduce阶段之前。reduceTask会将自己要处理的分区数据fetch到reduceTask所在的机器节点上，然后以键值对<k2,list<v2>>作为reduce函数的输入，经过函数处理后，再以键值对<k3,v3>的形式作为输出。最终结果可以保存到hdfs系统上。
```

3. getSplit怎么分片的，分片的大小

1. split是逻辑分片，再mapTask任务开始前，将文件按照指定的大小进行逻辑切分。每一个部分称之为一个split。默认情况下，split的大小与block的大小相等。均为128M.
2. 可以参考FileInputForamt类的getSplits()源码
    1. 会先获取三个参数的值，minSize,maxSize,blockSize
    2. 然后创建一个分片集合用于存储分片数据

3. 获取文件的所有块信息，进行遍历
4. 得到一个块的状态信息，然后判断是否可以切分
5. 然后根据三个参数获取分片大小
6. 循环判断文件剩余部分是否大于切片大小的1.1倍，
    1. 大于的话就调用makeSplit方法创建当前块的逻辑分片
    2. 不大于的话，就将文件剩余的部分创建一个唯一的最后一个分片。
7. 将每一个逻辑分片添加到分片集合中，等待被使用

3. 分片的大小由minSize,maxSize,blockSize三个参数决定

   算法如下：

```
Math.max(minSize,Math.min(maxSize, blockSize))，其中maxSize是取得longValueMax的值

1.如果blockSize小于maxSize && blockSize 大于 minSize之间，那么split就是blockSize；

2.如果blockSize小于maxSize && blockSize 小于 minSize之间，那么split就是minSize；

3.如果blockSize大于maxSize && maxSize   大于 minSize之间，那么split就是maxSize；

4.如果blockSize大于maxSize && maxSize   小于 minSize之间，那么split就是maxSize（不存在这种关系）。
```