第3章

分布式计算框架 MapReduce

本章重点：

MapReduce 编程模型

使用 MapReduce 开发常用的功能

本章目标：

了解 MapReduce 是什么

掌握 MapReduce 编程模型

掌握 MapReduce 中常见核心 API 的编程

掌握使用 MapReduce 开发常用的功能

本章任务

学习本章，需要完成以下 3 个工作任务。请记录下来学习过程中所遇到的问题，可以通过自己的努力或访问 kgc.cn 解决。

任务 1：MapReduce 编程模型

理解并掌握 MapReduce 的编程模型，进一步加深对大数据并行计算模型的理解与思考。

任务 2：MapReduce 进阶

通过学习 MapReduce 各个组件的概念和原理加深对 MapReduce 底层原理和计算模型的掌握。

任务 3：MapReduce 高级编程

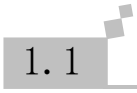
掌握 MapReduce 开发常用的应用，例如 Join、排序、二次排序、合并小文件等。

任务 1

MapReduce 编程模型

关键步骤如下：

- MapReduce 是什么，适合做什么，不适合做什么。
- MapReduce 中 map 和 reduce 方法的功能。
- 开发 MapReduce 版本的 wordcount 程序并提交到集群运行。



1.1 MapReduce 概述

1.1.1 MapReduce 是什么

MapReduce 是 Google 开源的一项重要技术，首先它是一个编程模型，用以进行大数据量的计算。对于大数据量的计算，通常采用的处理方式就是并行计算。但对许多开发者来说，自己完完全全实现一个并行计算程序难度太大，而 MapReduce 就是一种简化并行计算的编程模型，它使得那些没有多少并行计算经验的开发人员也可以开发并行应用程序。这也就是 MapReduce 的价值所在，通过简化编程模型，降低了开发并行应用程序的入门门槛。

1.1.2 MapReduce 设计目标

MapReduce 的设计目标是方便编程人员在不熟悉分布式并行编程的情况下，将自己的程序运行在分布式系统上。MapReduce 采用的是“分而治之”的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个子节点共同完成，然后整合各个子节点的中间结果，得到最终的计算结果。简而言之，MapReduce 就是“分散任务，汇总结果”。

1.1.3 MapReduce 特点

1） MapReduce 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，这个分布式程序可以分布到大量廉价的 PC 机器运行。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。 就是因为这个特点使得 MapReduce 编程变得非常流行。

2） 良好的扩展性。当你的计算资源不能得到满足的时候，你可以通过简单的增加机器来扩展它的计算能力。

3） 高容错性。MapReduce 设计的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。比如其中一台机器挂了，它可以把上面的计算任务转移到另外一个节点上面上运行，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。

4） 能对 PB 级以上海量数据的进行离线处理。适合离线处理而不适合实时处理，比如像毫秒级别的返回一个结果，MapReduce 很难做到。

1.1.4 MapReduce 不擅长的场景

易于编程 MapReduce 虽然具有很多的优势，但是它也有不擅长的地方。这里的不擅长不代表它不能做，而是在有些场景下实现的效果差，并不适合 MapReduce 来处理，主要表现在以下几个方面。

1） 实时计算：MapReduce 无法像 MySQL 一样，在毫秒或者秒级内返回结果。

2） 流式计算：流式计算的输入数据时动态的，而 MapReduce 的输入数据集是静态的，不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。

3） 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，导致性能非常的低下。

1.2 MapReduce 编程模型

1.2.1 编程模型概述

从 MapReduce 自身的命名特点可以看出，MapReduce 由两个阶段组成：Map 和 Reduce。用户只需编写 map()和 reduce()两个函数，即可完成简单的分布式程序的设计。

map()函数以 key/value 对作为输入，产生另外一系列 key/value 对作为中间输出写入本地磁盘。MapReduce 框架会自动将这些中间数据按照 key 值进行聚集，且 key 值相同（用户可设定聚集策略，默认情况下是对 key 值进行哈希取模）的数据被统一交给 reduce()函数处理。

reduce()函数以 key 及对应的 value 列表作为输入，经合并 key 相同的 value 值后，产生另外一系列 key/value 对作为最终输出写入 HDFS。

MapReduce 将作业（一个 MapReduce 应用程序）的整个运行过程分为两个阶段：Map 阶段和 Reduce 阶段。

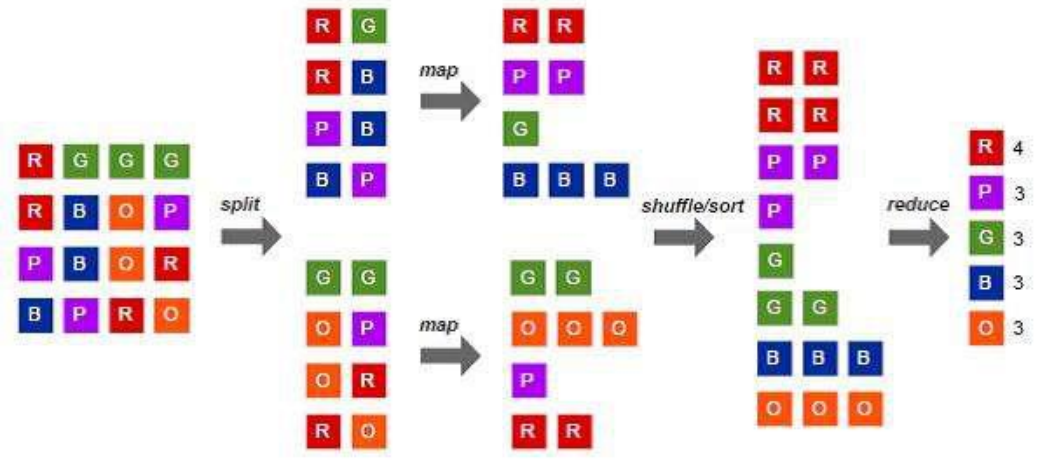


图 1.1　MapReduce 编程模型

1)　Map 阶段由一定数量的 Map Task 组成；

- 输入数据格式解析：InputFormat（把输入文件进行分片）
- 输入数据处理：Mapper
- 数据分组：Partitioner

2)　Reduce 阶段由一定数量的 Reduce Task 组成；

- 数据远程拷贝（从 Map Task 的输出拷贝部分数据）
- 数据按照 key 排序和分组（key 相同的都挨在一起，按照 key 进行分组操作，每一组交由 reducer 进行处理）
- 处理处理：Reducer

数据输出格式：OutputFormat （输出文件格式，分隔符等的设置）

1.2.2　编程模型三步曲

1）Input：一系列 k1/v1 对

2）Map 和 Reduce：　Map：(k1,v1) --> list(k2,v2) ，Reduce：(k2, list(v2)) --> list(k3,v3)

其中：k2/v2 是中间结果对

3）Output：一系列(k3,v3)对

1.3 MapReduce WordCount 编程实例

通过上一节的学习，我们已经理解了 MapReduce 的基本编程模型，为了加深对 MapReduce 的理解，本节将以一个 WordCount 程序来详细解释 MapReduce 模型。一个最简单的 MapReduce 应用程序至少包含 3 个部分：一个 Map 函数、一个 Reduce 函数和一个 main 函数。在运行一个 MapReduce 计算任务时候，任务过程被分为两个阶段：map 阶段和 reduce 阶段，每个阶段都是用键值对（key/value）作为输入（input）和输出（output），main 函数将作业控制和文件输入/输出结合起来。

我们一起来看看 WordCount 程序的需求：现在有大量的文件，每个文件又有大量的单词，要求统计每个单词出现的词频。

1.3.1 WordCount 实现设计分析

1） Map 过程：并行读取文本，对读取的单词进行 map 操作，每个词都以<key,value>形式生成。

读取第一行 Hello World Bye World，分割单词形成 Map。

<Hello,1> <World,1> <Bye,1> <World,1>

读取第二行 Hello Hadoop Bye Hadoop，分割单词形成 Map。

<Hello,1> <Hadoop,1> <Bye,1> <Hadoop,1>

读取第三行 Bye Hadoop Hello Hadoop，分割单词形成 Map。

<Bye,1> <Hadoop,1> <Hello,1> <Hadoop,1>

2） Reduce 操作是对 map 的结果进行排序，合并，最后得出词频。

reduce 将形成的 Map 根据相同的 key 组合成 value 数组。<Bye,1,1,1> <Hadoop,1,1,1,1> <Hello,1,1,1> <World,1,1>。循环执行 Reduce(K,V[])，分别统计每个单词出现的次数，<Bye,3> <Hadoop,4> <Hello,3> <World,2>

1.3.2 WordCount 代码开发

在上面我们已经详细描述了 MapReduce 的编程模型，接下来，请各位跟我一起使用 Maven 来完成相关程序的开发，至于开发工具，大家可以根据自己的爱好选择，Eclipse 或者 IDEA 均可。为了方便大家理解，重要程序处标注了相关解释！

1）新建工程，添加 Maven 依赖

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
```

```
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>${hadoop.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>${hadoop.version}</version>
</dependency>
```

2）完整的 WordCount 程序代码

```
package com.kgc.bigdata.hadoop.mapreduce.wordcount;


import com.kgc.bigdata.hadoop.mapreduce.partitioner.PartitionerApp;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
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.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import java.io.IOException;
import java.net.URI;
import java.util.StringTokenizer;


/**
```

```
 * WordCount 的 MapReduce 实现
 */
public class WordCountApp {
    public static class MyMapper
            extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

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

    public static class MyReducer
            extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        String INPUT_PATH = "hdfs://hadoop000:8020/wc";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputwc";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
```

```
            if (fileSystem.exists(new Path(OUTPUT_PATH))) {
                fileSystem.delete(new Path(OUTPUT_PATH), true);
            }

            Job job = Job.getInstance(conf, "WordCountApp");

            // run jar class
            job.setJarByClass(WordCountApp.class);

            // 设置 map
            job.setMapperClass(MyMapper.class);
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);

            // 设置 reduce
            job.setReducerClass(MyReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

            // 设置 input formart
            job.setInputFormatClass(TextInputFormat.class);
            Path inputPath = new Path(INPUT_PATH);
            FileInputFormat.addInputPath(job, inputPath);

            // 设置 output format
            job.setOutputFormatClass(TextOutputFormat.class);
            Path outputPath = new Path(OUTPUT_PATH);
            FileOutputFormat.setOutputPath(job, outputPath);

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

代码 1.1　WordCount 代码实现

1.3.3　WordCount 代码说明

1）对于 map 函数的方法。

```
public void map(Object key, Text value, Context context)
```

继承 Mapper 类，实现 map 方法，这里有三个参数，前面两个 Object key, Text value 就是输入的 key 和 value，第三个参数 Context context 记录的是整个上下文，比如我们可以通过 context 将数据写出去。

2）对于 reduce 函数的方法。

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
```

继承 Reducer 类，实现 reduce 方法，reduce 函数的输入也是一个 key/value 的形式，不过它的 value 是一个迭代器的形式 Iterable<IntWritable> values，也就是说 reduce 的输入是一个 key 对应一组的值的 value，reduce 也有 context 和 map 的 context 作用一致。

3）对于 main 函数的调用。

创建 Configuration 类：Configuration conf = new Configuration();

运行 MapReduce 程序前都要初始化 Configuration，该类主要是读取 MapReduce 系统配置信息。

创建 Job 类：

```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
```

第一行就是在构建一个 job，有两个参数，一个是 conf，另外一个是这个 job 的名称。

第二行就是设置我们自己开发的 MapReduce 类；

第三行和第四行就是设置 map 函数和 reduce 函数实现类。

设置输出的键值对的类型：

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

这个是定义输出的 key/value 的类型，也就是最终存储在 hdfs 上结果文件的 key/value 的类型。

设置 Job 的输入输出路径并提交到集群运行：

```
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

第一行就是构建输入的数据文件，第二行是构建输出的数据文件，最后一行如果 job 运行成功了，我们的程序就会正常退出。

1.3.4 WordCount 提交到集群运行

在这里我们第一次接触 MapReduce，因此这里以伪分布式集群的方式运行 WordCount 程序！感兴趣的朋友也可以在完全分布式集群中运行，具体过程笔者在此不再讨论（其实是一模一样的）。操作步骤如下所示：

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /wc
hadoop fs -put hello.txt /wc
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.wordcount.WordCountApp
```

4) 查看作业输出结果

```
hadoop fs -text /outputwc/part-*
hello     3
welcome 1
world     2
```

至此，在学习了以上相关知识后，任务 1 就可以完成了。

任务 2

MapReduce 进阶

关键步骤如下：

- MapReduce 类型。
- MapReduce 输入格式。
- MapReduce 输出格式。
- MapReduce 中的 Combiner、Partitioner、RecordReader 的使用。

2.1 MapReduce 类型

2.1.1 MapReduce 类型概述

使用 Hadoop 中的 MapReduce 编程模型处理过程非常简单，只需要定义好 map 和 reduce 函数的输入和输出是键值对的类型即可，那么我们来看看各种数据类型是如何在 MapReduce 中使用的。

MapReduce 中的 map 和 reduce 函数需要遵循如下的格式：

```
map: (K1,V1) -> list(K2,V2)
```

```
reduce: (K2, list(V2)) -> list(K3,V3)
```

从这个需要遵循的格式我们可以看出：reduce 函数的输入类型必须与 map 函数的输出类型一致。

2.1.2 MapReduce 中常用的设置

1）输入数据类型由输入格式（InputFormat）设置。比如：TextInputFormat 的 Key 的类型就是 LongWritable，Value 的类型是 Text；

2）map 的输出的 Key 的类型通过 setMapOutputKeyClass 设置，Value 的类型通过 setMapOutputValueClass 设置；

3） reduce 的输出的 Key 的类型通过 setOutputKeyClass 设置，Value 的类型通过 setOutputValueClass 设置；

2.2 MapReduce 输入格式

MapReduce 处理的数据文件，一般情况下输入文件一般是存储在 HDFS 里面。这些文件的格式可以是任意的：我们可以使用基于行的日志文件， 也可以使用二进制格式，多行输入记录或者其它一些格式。这些文件一般会很大，达到数十 GB，甚至更大。那么 MapReduce 是如何读取这些数据的呢

2.2.1 InputFormat 接口

InputFormat 接口决定了输入文件如何被 Hadoop 分块。InputFormat 能够从一个 job 中得到一个 split 集合（InputSplit[]），然后再为这个 split 集合配上一个合适的 RecordReader（getRecordReader）来读取每个 split 中的数据。 下面我们来看一下 InputFormat 接口由哪些抽象方法组成。

```
public interface InputFormat<K, V> {
    InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;

    RecordReader<K, V> getRecordReader(InputSplit split,
        JobConf job, Reporter reporter) throws IOException;
}
```

方法作用说明：

1）getSplits(JobContext context) 方法负责将一个大数据逻辑分成许多片。比如数据库表有 100 条数据，按照主键 ID 升序存储。假设每 20 条分成一片，这个 List 的大小就是 5，然后每个 InputSplit 记录两个参数，第一个为这个分片的起始 ID，第二个为这个分片数据的大小，这里是 20.很明显 InputSplit 并没有真正存储数据。只是提供了一个如何将数据分片的方法。

2）createRecordReader(InputSplit split,TaskAttemptContext context)方法根据 InputSplit 定义的方

法，返回一个能够读取分片记录的 RecordReader。getSplit 用来获取由输入文件计算出来的 InputSplit， 后面会看到计算 InputSplit 时，会考虑输入文件是否可分割、文件存储时分块的大小和文件大小等因素；而 createRecordReader() 提供了前面说的 RecordReader 的实现， 将 Key-Value 对从 InputSplit 中正确读出来，比如 LineRecordReader，它是以偏移值为 Key，每行的数据为 Value，这使所有 createRecordReader() 返回 LineRecordReader 的 InputFormat 都是以偏移值为 Key，每行数据为 Value 的形式读取输入分片的。

2.1.2 InputFormat 接口实现类

InputFormat 接口实现类有很多，其层次结构如下图所示

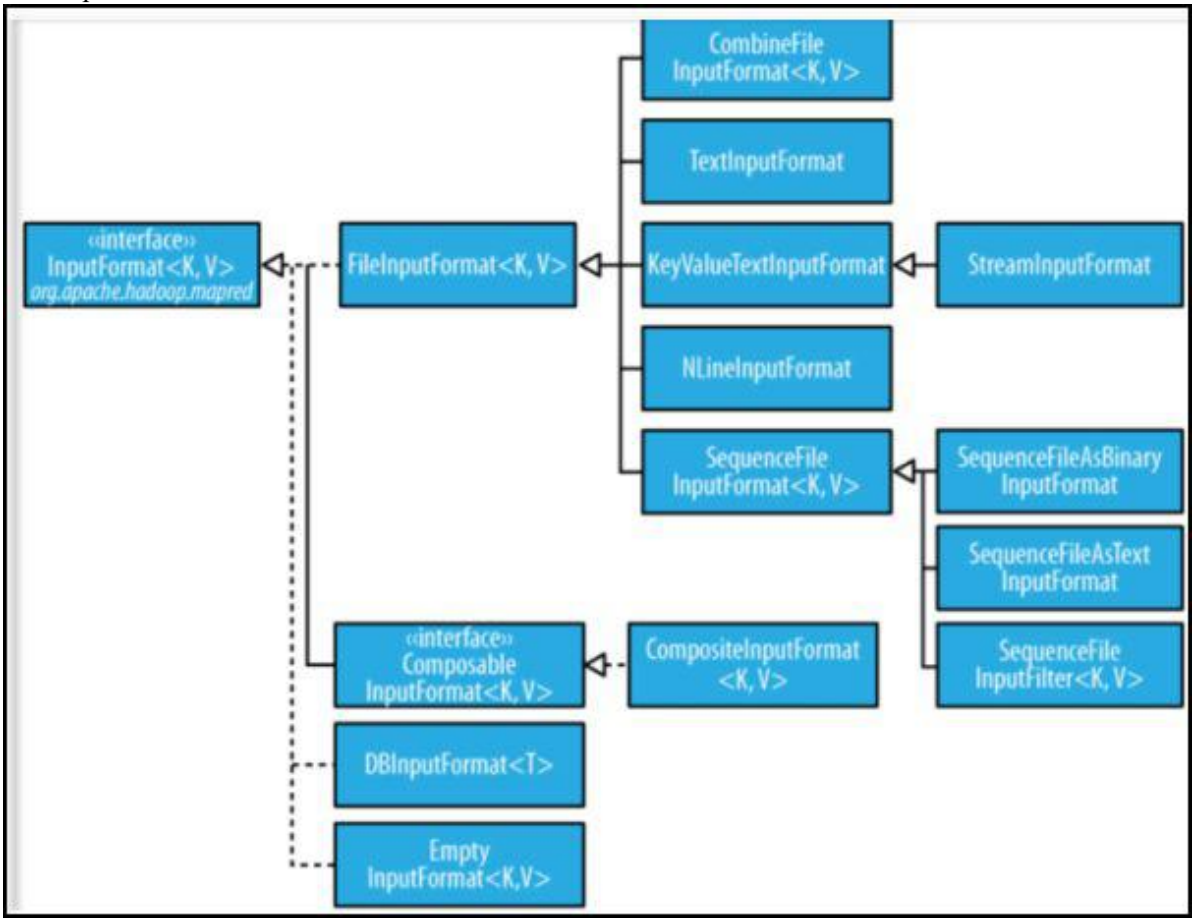


图 1.2 InputFormat 实现类

常用的 InputFormat 实现类介绍：

1） FileInputFormat

FileInputFormat 是所有使用文件作为其数据源的 InputFormat 实现的基类，它的主要作用是指出作业的输入文件位置。因为作业的输入被设定为一组路径， 这对指定作业输入提供了很强的灵活性。FileInputFormat 提供了四种静态方法来设定 Job 的输入路径：

```
public static void addInputPath(Job job,Path path);

public static void addInputPaths(Job job,String commaSeparatedPaths);

public static void setInputPaths(Job job,Path... inputPaths);

public static void setInputPaths(Job job,String commaSeparatedPaths);
```

2） KeyValueTextInputFormat

每一行均为一条记录，被分隔符（缺省是 tab）分割为 key（Text）,value（Text）。可以通过 mapreduce.input.keyvaluelinerecordreader.key.value,separator 属性（或者旧版本 API 中的 key.value.separator.in.input.line）来设定分隔符。

2.3 MapReduce 输出格式

针对前面介绍的输入格式，Hadoop 都有相应的输出格式。默认情况下只有一个 Reduce，输出只有一个文件，默认文件名为 part-r-00000，输出文件的个数与 Reduce 的个数一致。 如果有两个 Reduce，输出结果就有两个文件，第一个为 part-r-00000，第二个为 part-r-00001，依次类推。

2.3.1 OutputFormat 接口

OutputFormat 主要用于描述输出数据的格式，它能够将用户提供的 key/value 对写入特定格式的文件中。 通过 OutputFormat 接口，实现具体的输出格式，过程有些复杂也没有这个必要。Hadoop 自带了很多 OutputFormat 的实现，它们与 InputFormat 实现相对应，足够满足我们业务的需要。

2.3.2 OutputFormat 接口实现类

OutputFormat 接口实现类有很多， 其层次结构如下图所示。

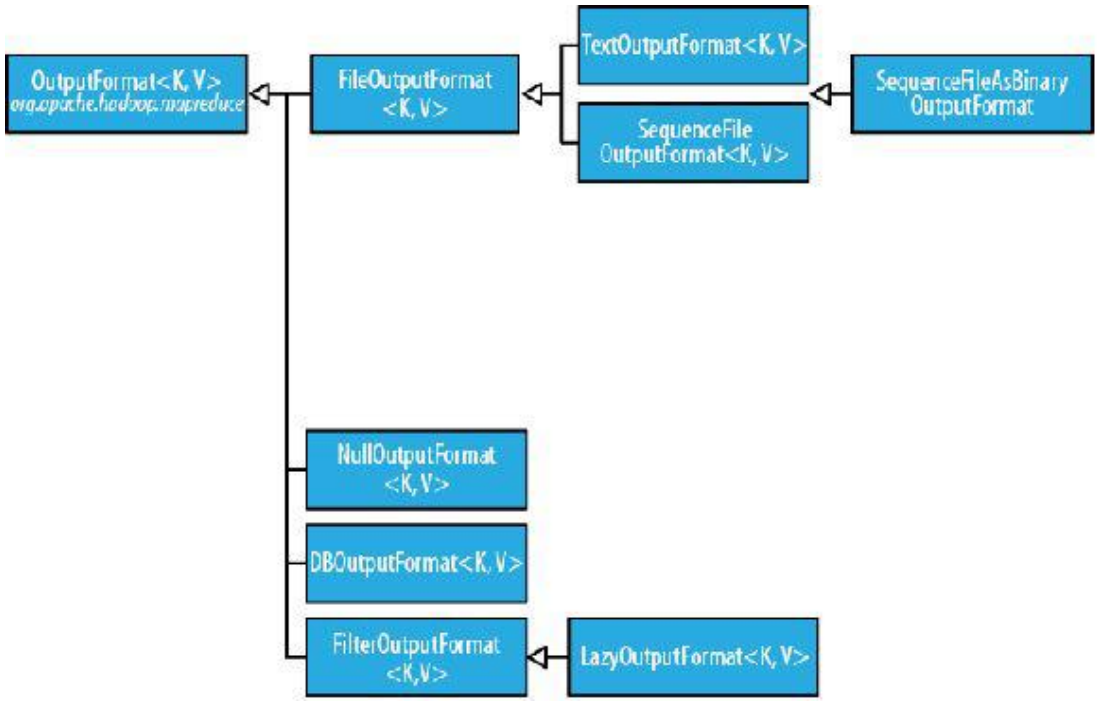


图 1.3　OutputFormat 实现类

OutputFormat 是 MapReduce 输出的基类，所有实现 MapReduce 输出都实现了 OutputFormat 接口。

常用的 OutputFormat 实现类介绍：

1）文本输出

默认的输出格式是 TextOutputFormat，它把每条记录写为文本行。它的键和值可以是实现了 Writable 的任意类型，因为 TextOutputFormat 调用 toString() 方法把它们转换为字符串。 每个键/值对由制表符进行分割，当然也可以设定 mapreduce.output.textoutputformat.separator 属性（旧版本 API 中的 mapred.textoutputformat.separator）改变默认的分隔符。 与 FileOutputFormat 对应的输入格式是 KeyValueTextInputFormat，它通过可配置的分隔符将键/值对文本分割。

可以使用 NullWritable 来省略输出的键或值（或两者都省略，相当于 NullOutputFormat 输出格式，后者什么也不输出）。 这也会导致无分隔符输出，以使输出适合用 TextInputFormat 读取。

2）二进制输出

SequenceFileOutputFormat 将它的输出写为一个顺序文件。如果输出需要作为后续 MapReduce 任务的输入，这便是一种好的输出格式， 因为它的格式紧凑，很容易被压缩。

2.4 Combiner

2.4.1 Combiner 概述

通过上面章节的学习我们可知，Hadoop 框架使用 Mapper 将数据处理成一个<key,value>键值对，然后在网络节点间对其进行整理(shuffle)，然后使用 Reducer 处理数据并进行最终输出。 试想如果存在这样一个实际场景：

如果有 10 亿个数据，Mapper 会生成 10 亿个键值对在网络间进行传输，但如果我们只是对数据求最大值，那么很明显的 Mapper 只需要输出它所知道的最大值即可。这样做不仅可以减轻网络压力，同样也可以大幅度提高程序效率。

在 MapReducer 框架中中，Combiner 就是为了避免 map 任务和 reduce 任务之间的数据传输而设置的，Hadoop 允许用户针对 map task 的输出指定一个合并函数。即为了减少传输到 Reduce 中的数据量。它主要是为了削减 Mapper 的输出数量，从而减少网络带宽和 Reducer 之上的负载。

我们可以把 Combiner 操作看成是一个在每个单独的节点上先做一次 Reducer 操作，其输出和输出的参数是和 Reduce 一样的。以 WordCount 为例，Combiner 的执行过程如下图所示：

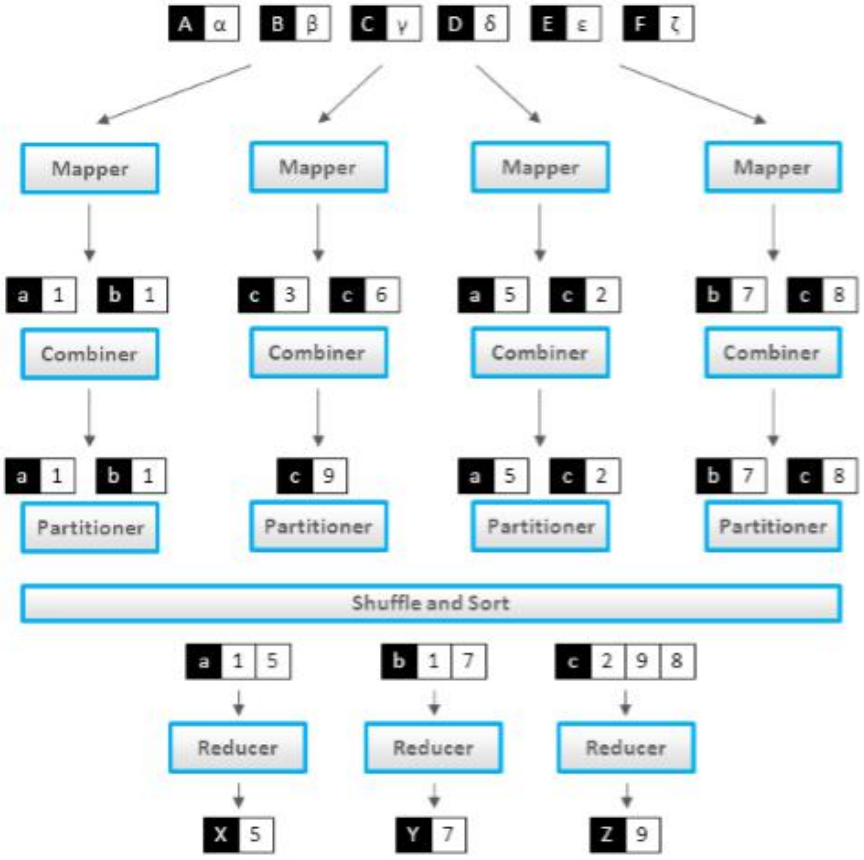


图 1.4　Combiner 执行流程

注意事项：对于求和、求最值的方式我们是可以使用 Combiner 的，但是求平均数是不能使用 Combiner 的。

2.4.2　Combiner 在 WordCount 中的使用

我们可以在 Map 输出之后添加一步 Combiner 的操作，先进行一次聚合，再由 Reduce 来处理，

进而使得传输数据减少，提高执行效率。

```
package com.kgc.bigdata.hadoop.mapreduce.combiner;

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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;
import java.util.StringTokenizer;

/**
 * WordCount 中使用 Combiner
 */
public class WordCountCombinerApp {
    public static class TokenizerMapper
            extends Mapper<Object, Text, Text, IntWritable>{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

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

    public static class IntSumReducer
            extends Reducer<Text,IntWritable,Text,IntWritable> {
```

```
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context
        ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCountCombinerApp.class);
        job.setMapperClass(TokenizerMapper.class);

        //通过 job 设置 Combiner 处理类，其实逻辑就可以直接使用 Reducer
        job.setCombinerClass(IntSumReducer.class);

        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

代码 1.2　Combiner 操作

2.5 Partitioner

2.5.1 Partitioner 概述

在进行 MapReduce 计算时，有时候需要把最终的输出数据分到不 同的文件中，比如按照省份划分的话，需要把同一省份的数据放到一个文件中；按照性别划分的话，需要把同一性别的数据放到一个文件中。我们知道最终的输出数 据是来自于 Reducer 任务。那么，如果要得到多个文件，意味着有同样数量的 Reducer 任务在运行。Reducer 任务的数据来自于 Mapper 任 务，也就说 Mapper 任务要划分数据，对于不同的数据分配给不同的 Reducer 任务运行。Mapper 任务划分数据的过程就称作 Partition。 负责实现划分数据的类称作 Partitioner。

MapReduce 默认的 partitioner 是 HashPartitioner。默认情况下，partitioner 先计算 key 的散列值（通常为 md5 值）。然后通过 reducer 个数执行取模运算：key.hashCode%(reducer 个数)。这种方式不仅能够随机地将整个 key 空间平均分发给每个 reducer,同时也能确保不同 mapper 产生的相同 key 能被分发到同一个 reducer。

2.5.2 Partitioner 案例

1）需求：分别统计每种类型手机的销售情况，每种类型手机统计数据单独存放在一个结果中。

2）代码实现：

```
package com.kgc.bigdata.hadoop.mapreduce.partitioner;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
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.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import java.io.IOException;
import java.net.URI;

/**
```

```
 * 自定义 Partitoner 在 MapReduce 中的应用
 */
public class PartitionerApp {

    private static class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        @Override
        protected void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String[] s = value.toString().split("\t");
            context.write(new Text(s[0]), new IntWritable(Integer.parseInt(s[1])));
        }

    }

    private static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        protected void reduce(Text key, Iterable<IntWritable> value, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : value) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }

    }

    public static class MyPartitioner extends Partitioner<Text, IntWritable> {

        //转发给 4 个不同的 reducer
        @Override
        public int getPartition(Text key, IntWritable value, int numPartitons) {
            if (key.toString().equals("xiaomi"))
                return 0;
            if (key.toString().equals("huawei"))
                return 1;
            if (key.toString().equals("iphone7"))
```

```
                return 2;
            return 3;
        }
    }


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

        String INPUT_PATH = "hdfs://hadoop000:8020/partitioner";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputpartitioner";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "PartitionerApp");

        // run jar class
        job.setJarByClass(PartitionerApp.class);

        // 设置 map
        job.setMapperClass(MyMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        // 设置 reduce
        job.setReducerClass(MyReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        //设置 Partitioner
        job.setPartitionerClass(MyPartitioner.class);
        //设置 4 个 reducer，每个分区一个
        job.setNumReduceTasks(4);
```

```
        // input formart
        job.setInputFormatClass(TextInputFormat.class);
        Path inputPath = new Path(INPUT_PATH);
        FileInputFormat.addInputPath(job, inputPath);

        // output format
        job.setOutputFormatClass(TextOutputFormat.class);
        Path outputPath = new Path(OUTPUT_PATH);
        FileOutputFormat.setOutputPath(job, outputPath);

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

代码 1.3　Partitioner 操作

2.5.3　提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /partitioner
hadoop fs -put part_1.txt part_2.txt /partitioner
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.partitioner.PartitionerApp
```

4) 查看作业输出结果

```
hadoop fs -ls /outputpartitioner
Found 5 items
-rw-r--r--   1 hadoop supergroup          0 2017-02-19 13:40 /outputpartitioner/_SUCCESS
-rw-r--r--   1 hadoop supergroup         51 2017-02-19 13:40 /outputpartitioner/part-r-00000
-rw-r--r--   1 hadoop supergroup         51 2017-02-19 13:40 /outputpartitioner/part-r-00001
-rw-r--r--   1 hadoop supergroup         52 2017-02-19 13:40 /outputpartitioner/part-r-00002
-rw-r--r--   1 hadoop supergroup         52 2017-02-19 13:40 /outputpartitioner/part-r-00003

hadoop fs -text /outputpartitioner/part-r-00000
xiaomi    35
```

```
hadoop fs -text /outputpartitioner/part-r-00001
huawei    11

hadoop fs -text /outputpartitioner/part-r-00002
iphone7 120

hadoop fs -text /outputpartitioner/part-r-00003
iphone7p            120
```

2.6 RecordReader

2.6.1 RecordReader 概述

RecordReader表示以怎样的方式从分片中读取一条记录，每读取一条记录都会调用RecordReader类，系统默认的 RecordReader 是 LineRecordReader，它是 TextInputFormat 对应的 RecordReader；而 SequenceFileInputFormat 对应的 RecordReader 是 SequenceFileRecordReader。LineRecordReader 是每行的偏移量作为读入 map 的 key，每行的内容作为读入 map 的 value。很多时候 Hadoop 内置的 RecordReader 并不能满足我们的需求，比如我们在读取记录的时候，希望 Map 读入的 Key 值不是偏移量而是行号或者是文件名，这时候就需要我们自定义 RecordReader。

自定义 RecordReader 的实现步骤

1） 继承抽象类 RecordReader，实现 RecordReader 的一个实例。

2） 实现自定义 InputFormat 类，重写 InputFormat 中的 CreateRecordReader()方法，返回值是自定义的 RecordReader 实例。

3） 配置 job.setInputFormatClass()为自定义的 InputFormat 实例。

2.6.2 RecordReader 案例

1）需求：统计 data 文件中奇数行和偶数行的和。

2）代码实现：

```
package com.kgc.bigdata.hadoop.mapreduce.recordreader;

import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import java.io.IOException;

/**
 * 自定义 InputFormat
 */
public class MyInputFormat extends FileInputFormat<LongWritable, Text> {

    @Override
    public RecordReader<LongWritable, Text> createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException, InterruptedException {
        //返回自定义的 RecordReader
        return new RecordReaderApp.MyRecordReader();
    }

    /**
     * 为了使得切分数据的时候行号不发生错乱，这里设置为不进行切分
     */
    protected boolean isSplitable(FileSystem fs, Path filename) {
        return false;
    }
}

package com.kgc.bigdata.hadoop.mapreduce.recordreader;

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

/**
 * 自定义 Partitioner
 */
public class MyPartitioner extends Partitioner<LongWritable, Text> {
```

```
    @Override
    public int getPartition(LongWritable key, Text value, int numPartitions) {
        //偶数放到第二个分区进行计算
        if (key.get() % 2 == 0) {
            //将输入到 reduce 中的 key 设置为 1
            key.set(1);
            return 1;
        } else {//奇数放在第一个分区进行计算
            //将输入到 reduce 中的 key 设置为 0
            key.set(0);
            return 0;
        }
    }
}

package com.kgc.bigdata.hadoop.mapreduce.recordreader;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.LineReader;

import java.io.IOException;
import java.net.URI;

/**
 * 自定义 RecordReader 在 MapReduce 中的使用
 */
public class RecordReaderApp {
```

```
public static class MyRecordReader extends RecordReader<LongWritable, Text> {

    //起始位置(相对整个分片而言)
    private long start;

    //结束位置(相对整个分片而言)
    private long end;

    //当前位置
    private long pos;

    //文件输入流
    private FSDataInputStream fin = null;
    //key、value
    private LongWritable key = null;
    private Text value = null;
    //定义行阅读器(hadoop.util 包下的类)
    private LineReader reader = null;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws IOException {

        //获取分片
        FileSplit fileSplit = (FileSplit) split;
        //获取起始位置
        start = fileSplit.getStart();
        //获取结束位置
        end = start + fileSplit.getLength();
        //创建配置
        Configuration conf = context.getConfiguration();
        //获取文件路径
        Path path = fileSplit.getPath();
        //根据路径获取文件系统
        FileSystem fileSystem = path.getFileSystem(conf);
        //打开文件输入流
        fin = fileSystem.open(path);
        //找到开始位置开始读取
        fin.seek(start);
```

```
        //创建阅读器
        reader = new LineReader(fin);
        //将当期位置置为 1
        pos = 1;

    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (key == null) {
            key = new LongWritable();
        }
        key.set(pos);
        if (value == null) {
            value = new Text();
        }
        if (reader.readLine(value) == 0) {
            return false;
        }
        pos++;

        return true;

    }

    @Override
    public LongWritable getCurrentKey() throws IOException, InterruptedException {
        return key;
    }

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

    @Override
    public float getProgress() throws IOException, InterruptedException {
```

```
            return 0;
        }

        @Override
        public void close() throws IOException {
            fin.close();

        }
    }

    public static class MyMapper extends Mapper<LongWritable, Text, LongWritable, Text> {
        @Override
        protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
LongWritable, Text>.Context context) throws IOException,
                InterruptedException {
            // 直接将读取的记录写出去
            context.write(key, value);
        }
    }

    public static class MyReducer extends Reducer<LongWritable, Text, Text, LongWritable> {

        // 创建写出去的 key 和 value
        private Text outKey = new Text();
        private LongWritable outValue = new LongWritable();

        protected void reduce(LongWritable key, Iterable<Text> values, Reducer<LongWritable,
Text, Text, LongWritable>.Context context) throws IOException,
                InterruptedException {

            System.out.println("奇数行还是偶数行：" + key);

            // 定义求和的变量
            long sum = 0;
            // 遍历 value 求和
            for (Text val : values) {
                // 累加
                sum += Long.parseLong(val.toString());
```

```
            }

            // 判断奇偶数
            if (key.get() == 0) {
                outKey.set("奇数之和为：");
            } else {
                outKey.set("偶数之和为：");

            }
            // 设置 value
            outValue.set(sum);

            // 把结果写出去
            context.write(outKey, outValue);
        }
    }

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

        String INPUT_PATH = "hdfs://hadoop000:8020/recordreader";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputrecordreader";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "RecordReaderApp");

        // run jar class
        job.setJarByClass(RecordReaderApp.class);

        // 1.1 设置输入目录和设置输入数据格式化的类
        FileInputFormat.setInputPaths(job, INPUT_PATH);
        job.setInputFormatClass(MyInputFormat.class);
```

```
            // 1.2 设置自定义 Mapper 类和设置 map 函数输出数据的 key 和 value 的类型
            job.setMapperClass(MyMapper.class);
            job.setMapOutputKeyClass(LongWritable.class);
            job.setMapOutputValueClass(Text.class);

            // 1.3 设置分区和 reduce 数量(reduce 的数量，和分区的数量对应，因为分区为一个，所以 reduce 的数量也是一个)
            job.setPartitionerClass(MyPartitioner.class);
            job.setNumReduceTasks(2);

            // 2.1 Shuffle 把数据从 Map 端拷贝到 Reduce 端。
            // 2.2 指定 Reducer 类和输出 key 和 value 的类型
            job.setReducerClass(MyReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(LongWritable.class);

            // 2.3 指定输出的路径和设置输出的格式化类
            FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));
            job.setOutputFormatClass(TextOutputFormat.class);

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

代码 1.4　RecordReader 操作

2.6.3　提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /recordreader
hadoop fs -put recordreader.txt /recordreader
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.recordreader.RecordReaderApp
```

4) 查看作业输出结果

```
hadoop fs -ls /outputrecordreader
Found 3 items
-rw-r--r--   1 hadoop supergroup          0 2017-02-19 14:03 /outputrecordreader/_SUCCESS
-rw-r--r--   1 hadoop supergroup         69 2017-02-19 14:03 /outputrecordreader/part-r-00000
-rw-r--r--   1 hadoop supergroup         67 2017-02-19 14:03 /outputrecordreader/part-r-00001

hadoop fs -text /outputrecordreader/part-r-00000
奇数之和为：    25

hadoop fs -text /outputrecordreader/part-r-00001
偶数之和为：    30
```

至此，在学习了以上相关知识后，任务 2 就可以完成了。

任务 3

MapReduce 高级编程

关键步骤如下：

- 使用 MapReduce 完成 join 操作。
- 使用 MapReduce 完成排序操作。
- 使用 MapReduce 完成二次排序操作。
- 使用 MapReduce 完成小文件合并操作。

3.1 Join 的 MapReduce 实现

3.1.1 概述

熟悉 SQL 的同学都知道，使用 SQL 语法实现 join 是非常简单的，只需要一条 SQL 语句即可实现，但是在大数据场景下使用 MapReduce 编程模型实现 join 还是比较繁琐的。在实际生产中我们可以借助于 Hive、Spark SQL 等框架来实现 join，但是对于 join 的实现原理我个人觉得大家还是需要掌握的，这对于理解 join 的底层实现是很有帮助的，所以本节我们将学习如何使用 MapReduce API 来实现 join。

3.1.2 需求

实现如下 SQL 的功能：

```
select e.empno,e.ename,d.deptno,d.dname from emp e join dept d on e.deptno=d.deptno;
```

```
//测试数据 emp.txt
7369    SMITH    CLERK    7902    1980-12-17    800.00    20
7499    ALLEN    SALESMAN    7698    1981-2-20    1600.00 300.00    30
7521    WARD    SALESMAN    7698    1981-2-22    1250.00 500.00    30
7566    JONES    MANAGER 7839    1981-4-2    2975.00    20
….

//测试数据 dept.txt
10    ACCOUNTING    NEW YORK
20    RESEARCH    DALLAS
30    SALES    CHICAGO
40    OPERATIONS    BOSTON
```

3.1.3 MapReduce Map 端 join 的实现原理

1）Map 端读取所有的文件，并在输出的内容里加上标示，代表数据是从哪个文件里来的。

2）在 reduce 处理函数中，按照标识对数据进行处理。

3）然后根据 Key 去 join 来求出结果直接输出。

3.1.4 MapReduce Map 端 join 的代码实现

1) 员工类定义

```
package com.kgc.bigdata.hadoop.mapreduce.reducejoin;

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

import org.apache.hadoop.io.WritableComparable;

/**
```

```
 * 员工对象
 */
public class Emplyee implements WritableComparable {

    private String empNo = "";
    private String empName = "";
    private String deptNo = "";
    private String deptName = "";
    private int flag = 0;   //区分是员工还是部门

    public Emplyee() {
    }

    public Emplyee(String empNo, String empName, String deptNo, String deptName, int
flag) {
        this.empNo = empNo;
        this.empName = empName;
        this.deptNo = deptNo;
        this.deptName = deptName;
        this.flag = flag;
    }

    public Emplyee(Emplyee e) {
        this.empNo = e.empNo;
        this.empName = e.empName;
        this.deptNo = e.deptNo;
        this.deptName = e.deptName;
        this.flag = e.flag;
    }

    public String getEmpNo() {
        return empNo;
    }

    public void setEmpNo(String empNo) {
        this.empNo = empNo;
    }
```

```
    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public String getDeptNo() {
        return deptNo;
    }

    public void setDeptNo(String deptNo) {
        this.deptNo = deptNo;
    }

    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }

    public int getFlag() {
        return flag;
    }

    public void setFlag(int flag) {
        this.flag = flag;
    }

    @Override
    public void readFields(DataInput input) throws IOException {
        this.empNo = input.readUTF();
        this.empName = input.readUTF();
        this.deptNo = input.readUTF();
        this.deptName = input.readUTF();
```

```
            this.flag = input.readInt();
        }

        @Override
        public void write(DataOutput output) throws IOException {
            output.writeUTF(this.empNo);
            output.writeUTF(this.empName);
            output.writeUTF(this.deptNo);
            output.writeUTF(this.deptName);
            output.writeInt(this.flag);

        }

        //不做排序
        @Override
        public int compareTo(Object o) {
            return 0;
        }

        @Override
        public String toString() {
            return this.empNo + "," + this.empName + "," + this.deptNo + "," +
this.deptName;
        }
    }
```

代码 1.5　ReduceJoin 实现

2)　自定义 Mapper 类开发

```
    package com.kgc.bigdata.hadoop.mapreduce.reducejoin;

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

    import java.io.IOException;

    public class MyMapper extends Mapper<LongWritable, Text, LongWritable, Emplyee> {
```

```
    @Override
    protected void map(LongWritable key, Text value,
                       Context context)
            throws IOException, InterruptedException {
        String val = value.toString();
        String[] arr = val.split("\t");

        System.out.println("arr.length=" + arr.length + "      arr[0]=" + arr[0]);

        if (arr.length <= 3) {//dept
            Emplyee e = new Emplyee();
            e.setDeptNo(arr[0]);
            e.setDeptName(arr[1]);
            e.setFlag(1);

            context.write(new LongWritable(Long.valueOf(e.getDeptNo())), e);

        } else {//emp
            Emplyee e = new Emplyee();
            e.setEmpNo(arr[0]);
            e.setEmpName(arr[1]);
            e.setDeptNo(arr[7]);
            e.setFlag(0);

            context.write(new LongWritable(Long.valueOf(e.getDeptNo())), e);
        }
    }
}
```

3) 自定义 Reducer 类开发

```
package com.kgc.bigdata.hadoop.mapreduce.reducejoin;

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

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

public class MyReducer extends
        Reducer<LongWritable, Emplyee, NullWritable, Text> {

    @Override
    protected void reduce(LongWritable key, Iterable<Emplyee> iter,
                          Context context)
            throws IOException, InterruptedException {

        Emplyee dept = null;
        List<Emplyee> list = new ArrayList<Emplyee>();

        for (Emplyee tmp : iter) {
            if (tmp.getFlag() == 0) {//emp
                Emplyee emplyee = new Emplyee(tmp);
                list.add(emplyee);
            } else {
                dept = new Emplyee(tmp);
            }
        }

        if (dept != null) {
            for (Emplyee emp : list) {
                emp.setDeptName(dept.getDeptName());
                context.write(NullWritable.get(), new Text(emp.toString()));
            }
        }
    }
}
```

4) 驱动类开发

```
package com.kgc.bigdata.hadoop.mapreduce.reducejoin;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
```

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.net.URI;

/**
 * 使用 MapReduce API 完成 Reduce Join 的功能
 */
public class EmpJoinApp {


    public static void main(String[] args) throws Exception {
        String INPUT_PATH = "hdfs://hadoop000:8020/inputjoin";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputmapjoin";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "Reduce Join");

        // 设置主类
        job.setJarByClass(EmpJoinApp.class);

        //设置 Map 和 Reduce 处理类
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        //设置 Map 输出类型
        job.setMapOutputKeyClass(LongWritable.class);
        job.setMapOutputValueClass(Emplyee.class);
```

```
            //设置 Reduce 输出类型
            job.setOutputKeyClass(NullWritable.class);
            job.setOutputValueClass(Emplyee.class);

            //设置输入和输出目录
            FileInputFormat.addInputPath(job, new Path(INPUT_PATH));
            FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));

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

3.1.5 提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /inputjoin
hadoop fs -put emp.txt dept.txt /inputjoin
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.reducejoin.EmpJoinApp
```

4) 查看作业输出结果

```
hadoop fs -text /outputmapjoin/part*


7934,MILLER,10,ACCOUNTING
7839,KING,10,ACCOUNTING
7782,CLARK,10,ACCOUNTING
7876,ADAMS,20,RESEARCH
7788,SCOTT,20,RESEARCH
7369,SMITH,20,RESEARCH
7566,JONES,20,RESEARCH
7902,FORD,20,RESEARCH
7844,TURNER,30,SALES
7499,ALLEN,30,SALES
```

```
7698,BLAKE,30,SALES
7654,MARTIN,30,SALES
7521,WARD,30,SALES
7900,JAMES,30,SALES
```

3.2 排序的 MapReduce 实现

3.2.1 需求

对输入文件中数据进行排序。输入文件中的每行内容均为一个数字，即一个数据。要求在输出中每行有两个间隔的数字，其中，第一个代表原始数据在原始数据集中的位次，第二个代表原始数据。

3.2.2 MapReduce 排序的实现原理

在 MapReduce 中默认就可以进行排序的，如果 key 为封装 int 的 IntWritable 类型，那么 MapReduce 按照数字大小对 key 排序，如果 key 为封装为 String 的 Text 类型，那么 MapReduce 按照字典顺序对字符串排序。我们能否使用内置的排序来完成这个功能呢？答案是肯定的。

在使用之前首先需要了解它的默认排序规则。它是按照 key 值进行排序的。我们就知道应该使用封装 int 的 IntWritable 型数据结构了。也就是在 map 中将读入的数据转化成 IntWritable 型，然后作为 key 值输出（value 任意）。reduce 拿到<key，value-list>之后，将输入的 key 作为 value 输出，并根据 value-list 中元素的个数决定输出的次数。输出的 key 是一个全局变量，它统计当前 key 的位次。

3.2.3 MapReduce 排序的代码实现

```
package com.kgc.bigdata.hadoop.mapreduce.sort;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
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 java.io.IOException;
import java.net.URI;

/**
 * 使用 MapReduce API 实现排序
 */
public class SortApp {
    public static class MyMapper extends
            Mapper<LongWritable, Text, IntWritable, IntWritable> {
        private static IntWritable data = new IntWritable();

        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            data.set(Integer.parseInt(line));
            context.write(data, new IntWritable(1));
        }

    }

    public static class MyReducer extends
            Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
        private static IntWritable data = new IntWritable(1);

        public void reduce(IntWritable key, Iterable<IntWritable> values,
                           Context context)
                throws IOException, InterruptedException {
            for (IntWritable val : values) {
                context.write(data, key);
                data = new IntWritable(data.get() + 1);
            }
        }
    }

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

```
        String INPUT_PATH = "hdfs://hadoop000:8020/sort";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputsort";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "SortApp");

        // 设置主类
        job.setJarByClass(SortApp.class);

        //设置 Map 和 Reduce 处理类
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        //设置输出类型
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);

        //设置输入和输出目录
        FileInputFormat.addInputPath(job, new Path(INPUT_PATH));
        FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));

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

代码 1.6　排序实现

3.2.4　提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；
2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /sort
```

```
hadoop fs -put sort.txt /sort
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.sort.SortApp
```

4) 查看作业输出结果

```
hadoop fs -text /outputsort/part*
1	1
2	2
3	3
4	4
5	5
6	9
```

3.3 二次排序的 MapReduce 实现

3.3.1 概述

默认情况下，Map 输出的结果会对 Key 进行默认的排序，但是有时候需要对 Key 排序的同时还需要对 Value 进行排序，这就是所谓的二次排序。

3.3.2 需求

对输入文件中的数据（每行两列，列于列之间的分隔符是制表符），输出结果先按照第一个字段的升序排列，如果第一列的值相等，就按照第二个字段的升序排列。形如：

```
30	10
30	20
30	30
30	40

40	5
40	10
40	20
40	30

50	10
50	20
```

```
50      50
50      60
```

3.3.3 MapReduce 二次排序的实现原理

1）Mapper 任务会接收输入分片，然后不断的调用 map 函数，对记录进行处理。处理完毕后，转换为新的<key,value>输出。

2）对 map 函数输出的<key, value>调用分区函数，对数据进行分区。不同分区的数据会被送到不同的 Reducer 任务中。

3）对于不同分区的数据，会按照 key 进行排序，这里的 key 必须实现 WritableComparable 接口。该接口实现了 Comparable 接口，因此可以进行比较排序。

4）对于排序后的<key,value>，会按照 key 进行分组。如果 key 相同，那么相同 key 的<key,value>就被分到一个组中。最终，每个分组会调用一次 reduce 函数。

5）排序、分组后的数据会被送到 Reducer 节点。

3.3.4 MapReduce 二次排序的代码实现

```
package com.kgc.bigdata.hadoop.mapreduce.secondsort;

import org.apache.hadoop.io.WritableComparable;

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

public class IntPair implements WritableComparable<IntPair> {

    private int first = 0;
    private int second = 0;

    public void set(int left, int right) {
        first = left;
        second = right;
    }
    public int getFirst() {
        return first;
    }
    public int getSecond() {
```

```
            return second;
        }

        @Override
        public void readFields(DataInput in) throws IOException {
            first = in.readInt();
            second = in.readInt();
        }
        @Override
        public void write(DataOutput out) throws IOException {
            out.writeInt(first);
            out.writeInt(second);
        }
        @Override
        public int hashCode() {
            return first+"".hashCode() + second+"".hashCode();
        }
        @Override
        public boolean equals(Object right) {
            if (right instanceof IntPair) {
                IntPair r = (IntPair) right;
                return r.first == first && r.second == second;
            } else {
                return false;
            }
        }
        //这里的代码是关键，因为对 key 排序时，调用的就是这个 compareTo 方法
        @Override
        public int compareTo(IntPair o) {
            if (first != o.first) {
                return first - o.first;
            } else if (second != o.second) {
                return second - o.second;
            } else {
                return 0;
            }
        }
    }
```

```
package com.kgc.bigdata.hadoop.mapreduce.secondsort;

import java.io.IOException;
import java.net.URI;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class SecondarySortApp {

    public static class MyMapper extends Mapper<LongWritable, Text, IntPair,
IntWritable> {

        private final IntPair key = new IntPair();
        private final IntWritable value = new IntWritable();

        @Override
        public void map(LongWritable inKey, Text inValue,
                        Context context) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(inValue.toString());
            int left = 0;
            int right = 0;
            if (itr.hasMoreTokens()) {
                left = Integer.parseInt(itr.nextToken());
                if (itr.hasMoreTokens()) {
```

```
                    right = Integer.parseInt(itr.nextToken());
                }
                key.set(left, right);
                value.set(right);
                context.write(key, value);
            }
        }
    }

    /**
     * 在分组比较的时候，只比较原来的 key，而不是组合 key。
     */
    public static class GroupingComparator implements RawComparator<IntPair> {
        @Override
        public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
            return WritableComparator.compareBytes(b1, s1, Integer.SIZE/8, b2, s2,
Integer.SIZE/8);
        }

        @Override
        public int compare(IntPair o1, IntPair o2) {
            int first1 = o1.getFirst();
            int first2 = o2.getFirst();
            return first1 - first2;
        }
    }

    public static class MyReducer extends Reducer<IntPair, IntWritable, Text, IntWritable>
{
        private static final Text SEPARATOR = new Text("------------");
        private final Text first = new Text();

        @Override
        public void reduce(IntPair key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
            context.write(SEPARATOR, null);
            first.set(Integer.toString(key.getFirst()));
            for(IntWritable value: values) {
```

```
                context.write(first, value);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        String INPUT_PATH = "hdfs://hadoop000:8020/secondsort";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputsecondsort";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "SecondarySortApp");

        // 设置主类
        job.setJarByClass(SecondarySortApp.class);

        // 输入路径
        FileInputFormat.setInputPaths(job, new Path(INPUT_PATH));
        // 输出路径
        FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));

        //设置 Map 和 Reduce 处理类
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        // 分组函数
        job.setGroupingComparatorClass(GroupingComparator.class);

        job.setMapOutputKeyClass(IntPair.class);
        job.setMapOutputValueClass(IntWritable.class);

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

```
            // 输入输出格式
            job.setInputFormatClass(TextInputFormat.class);
            job.setOutputFormatClass(TextOutputFormat.class);

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

代码 1.7　二次排序实现

3.3.5　提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /secondsort
hadoop fs -put secondsort.txt /secondsort
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.secondsort.SecondarySortApp
```

4) 查看作业输出结果

```
hadoop fs -text /outputsecondsort/part*
-----------------------------------------------
30      10
30      20
30      30
30      40
-----------------------------------------------
40      5
40      10
40      20
40      30
-----------------------------------------------
50      10
50      20
50      50
50      60
```

3.4 合并小文件的 MapReduce 实现

3.4.1 概述

Hadoop 对处理单个大文件比处理多个小文件更有效率，另外单个文件也非常占用 HDFS 的存储空间。所以往往要将其合并起来。

3.4.2 需求

通过 MapReduce API 对小文件进行合并，输出成 SequenceFile。

3.4.3 合并小文件的代码实现

```
package com.kgc.bigdata.hadoop.mapreduce.merge;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import java.io.IOException;

/**
 * 实现将整个文件作为一条记录处理的 InputFormat
 */
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;
        }
}


package com.kgc.bigdata.hadoop.mapreduce.merge;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

/**
 * 实现一个定制的 RecordReader，这六个方法均为继承的 RecordReader
 */
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
        }
}

package com.kgc.bigdata.hadoop.mapreduce.merge;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;

import java.io.IOException;
import java.net.URI;

/**
 * 使用 MapReduce API 完成文件合并的功能
 */
public class MergeApp {

    /**
     * 将小文件打包成 SequenceFile
     */
    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);
        }
    }

    public static void main(String[] args) throws Exception {
        String INPUT_PATH = "hdfs://hadoop000:8020/inputmerge";
        String OUTPUT_PATH = "hdfs://hadoop000:8020/outputmerge";

        Configuration conf = new Configuration();
        final FileSystem fileSystem = FileSystem.get(new URI(INPUT_PATH), conf);
        if (fileSystem.exists(new Path(OUTPUT_PATH))) {
            fileSystem.delete(new Path(OUTPUT_PATH), true);
        }

        Job job = Job.getInstance(conf, "MergeApp");

        // 设置主类
        job.setJarByClass(MergeApp.class);

        job.setInputFormatClass(WholeFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BytesWritable.class);
        job.setMapperClass(SequenceFileMapper.class);
```

```
            //设置输入和输出目录
            FileInputFormat.addInputPath(job, new Path(INPUT_PATH));
            FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));

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

代码 1.8　合并小文件实现

3.4.4　提交作业到集群运行

1) 使用 mvn clean package -DskipTests 打成 hadoop-1.0-SNAPSHOT.jar，然后上传到 /home/hadoop/lib 目录下面；

2) 将测试数据上传到 HDFS 目录中

```
hadoop fs -mkdir /inputmerge
hadoop fs -put secondsort.txt /inputmerge
```

3) 提交 MapReduce 作业到集群运行

```
hadoop jar /home/hadoop/lib/hadoop-1.0-SNAPSHOT.jar
com.kgc.bigdata.hadoop.mapreduce.merge.MergeApp
```

4) 查看作业输出结果

```
hadoop fs -ls /outputmerge
Found 2 items
-rw-r--r--   1 hadoop supergroup          0    2017-02-18 19:23 /outputmerge/_SUCCESS
-rw-r--r--   1 hadoop supergroup    7630548 2017-02-18 19:23 /outputmerge/part-r-00000
```

至此，在学习了以上相关知识后，任务 3 就可以完成了。

本章总结

本章学习了以下知识点：

- 掌握 MapReduce 的编程模型。
- 掌握 MapReduce 中 Combiner、Partitioner 的使用。
- 掌握使用 MapReduce API 完成常用的功能。

本章作业

1. 使用 MapReduce API 完成如下功能。

需求：统计每个手机号码的流量（上行、下行）、数据包（上行、下行）

输出结果字段：手机号码、上行数据包数、下行数据包数、上行数据量、下行数据量。

输入文件字段说明：

序号	字段	字段类型	描述
0	reportTme	long	记录报告时间戳
1	msisdn	String	手机号码
2	apmac	String	AP mac
3	acmac	String	AC mac
4	host	String	访问的网址
5	siteType	String	网址种类
6	upPackNum	long	上行数据包数，单位：个
7	downPackNum	long	下行数据包数，单位：个
8	upPayLoad	long	上行总流量。要注意单位的转换，单位：byte
9	downPayLoad	long	下行总流量。要注意单位的转换，单位：byte
10	httpStatus	String	HTTP Response 的状态