# 第1章 MapReduce入门

## 1.1 MapReduce定义

Mapreduce是一个分布式运算程序的编程框架，是用户开发“基于hadoop的数据分析应用”的核心框架。

Mapreduce核心功能是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，并发运行在一个hadoop集群上。

框架是一个半成品

## 1.2 MapReduce优缺点

### 1.2.1 优点

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机器上运行。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．适合PB级以上海量数据的离线处理

它适合离线处理而不适合在线处理。比如像毫秒级别的返回一个结果，MapReduce很难做到。

### 1.2.2 缺点

**MapReduce不擅长做实时计算、流式计算、DAG（有向图）计算。**

1. 实时计算

MapReduce无法像Mysql一样，在毫秒或者秒级内返回结果。

1. 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。

1. 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。

## 1.3 MapReduce核心思想

MapReduce核心编程思想，如图所示



图MapReduce核心编程思想

1）分布式的运算程序往往需要分成至少2个阶段。

2）第一个阶段的maptask并发实例，完全并行运行，互不相干。

3）第二个阶段的reduce task并发实例互不相干，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。

4）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。

## 1.4 MapReduce进程（MR）

一个完整的mapreduce程序在分布式运行时有三类实例进程：

1）MrAppMaster：负责整个程序的过程调度及状态协调。

2）MapTask：负责map阶段的整个数据处理流程。

3）ReduceTask：负责reduce阶段的整个数据处理流程。

## 1.5 MapReduce编程规范（八股文）

用户编写的程序分成三个部分：Mapper、Reducer和Driver。

1．Mapper阶段

（1）用户自定义的Mapper要继承自己的父类

（2）Mapper的输入数据是KV对的形式（KV的类型可自定义）

（3）Mapper中的业务逻辑写在map()方法中

（4）Mapper的输出数据是KV对的形式（KV的类型可自定义）

（5）map()方法（maptask进程）对每一个<K,V>调用一次

2．Reducer阶段

（1）用户自定义的Reducer要继承自己的父类

（2）Reducer的输入数据类型对应Mapper的输出数据类型，也是KV

（3）Reducer的业务逻辑写在reduce()方法中

（4）Reducetask进程对每一组相同k的<k,v>组调用一次reduce()方法

3．Driver阶段(关联Mapper和Reducer，并且提交任务到集群)

相当于yarn集群的客户端，用于提交我们整个程序到yarn集群，提交的是封装了mapreduce程序相关运行参数的job对象

## 1.6[掌握] WordCount案例实操

1．需求

在给定的文本文件中统计输出每一个单词出现的总次数

2．数据准备



3．分析

按照mapreduce编程规范，分别编写Mapper，Reducer，Driver，如图所示。



图 需求分析

4．导入相应的依赖坐标+日志添加

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>RELEASE</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-common</artifactId>

<version>2.7.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>2.7.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.7.2</version>

</dependency>

</dependencies>

在项目的src/main/resources目录下，新建一个文件，命名为“log4j.properties”，在文件中填入

|  |
| --- |
| log4j.rootLogger=debug, stdout  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n  log4j.appender.logfile=org.apache.log4j.FileAppender  log4j.appender.logfile.File=target/spring.log  log4j.appender.logfile.layout=org.apache.log4j.PatternLayout  log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n |

5．编写程序

（1）编写mapper类

|  |
| --- |
| package com.bigdata.mapreduce;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class WordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{    Text k = new Text();  IntWritable v = new IntWritable(1);    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割  String[] words = line.split(" ");    // 3 输出  for (String word : words) {    k.set(word);  context.write(k, v);  }  }  } |

（2）编写reducer类

|  |
| --- |
| package com.bigdata.mapreduce.wordcount;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class WordcountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{  int sum;  IntWritable v = new IntWritable();  @Override  protected void reduce(Text key, Iterable<IntWritable> value,  Context context) throws IOException, InterruptedException {    // 1 累加求和  sum = 0;  for (IntWritable count : value) {  sum += count.get();  }    // 2 输出  v.set(sum);  context.write(key,v);  }  } |

（3）编写驱动类

|  |
| --- |
| package com.bigdata.mapreduce.wordcount;  import java.io.IOException;  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;  public class WordcountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  // 1 获取配置信息以及封装任务  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置jar加载路径  job.setJarByClass(WordcountDriver.class);  // 3 设置map和reduce类  job.setMapperClass(WordcountMapper.class);  job.setReducerClass(WordcountReducer.class);  // 4 设置map输出  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);  // 5 设置Reduce输出  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);    // 6 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 提交  job.waitForCompletion(true);  }  } |

5．本地测试

（1）在windows环境上配置HADOOP\_HOME环境变量

（2）在eclipse/idea上运行程序

6．集群上测试

（1）将程序打成jar包，然后拷贝到hadoop集群中

（2）启动hadoop集群

（3）执行wordcount程序

[hadoop@hadoop101 software]$ hadoop jar wc.jar

com.bigdata.wordcount.WordcountDriver /user/hadoop/input /user/bigdata/output1

# 第2章 Hadoop序列化

## 2.1 序列化概述

### 2.1.1 什么是序列化

序列化就是把内存中的对象，转换成字节序列（或其他数据传输协议）以便于存储（持久化）和网络传输。

反序列化就是将收到字节序列（或其他数据传输协议）或者是硬盘的持久化数据，转换成内存中的对象。

### 2.1.2 为什么要序列化

        一般来说，“活的”对象只生存在内存里，关机断电就没有了。而且“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。 然而序列化可以存储“活的”对象，可以将“活的”对象发送到远程计算机。

### 2.1.3 为什么不用Java的序列化 serilazable

 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，继承体系等），不便于在网络中高效传输。所以，hadoop自己开发了一套序列化机制（Writable），特点如下：

1．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

2．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；

3．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；

4．互操作

能支持不同语言写的客户端和服务端进行交互；

## 2.2 常用数据序列化类型

表4-1 常用的数据类型对应的hadoop数据序列化类型

|  |  |
| --- | --- |
| Java类型 | Hadoop Writable类型 |
| boolean | BooleanWritable |
| byte | ByteWritable |
| int | IntWritable |
| float | FloatWritable |
| long | LongWritable |
| double | DoubleWritable |
| String | Text |
| map | MapWritable |
| array | ArrayWritable |

## 2.3 自定义bean对象实现序列化接口（Writable）

自定义bean对象要想序列化传输，必须实现序列化接口，需要注意以下7项。

（1）必须实现Writable接口

（2）反序列化时，需要反射调用空参构造函数，所以必须有空参构造

|  |
| --- |
| public FlowBean() {  super();  } |

（3）重写序列化方法

|  |
| --- |
| @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  } |

（4）重写反序列化方法

|  |
| --- |
| @Override  public void readFields(DataInput in) throws IOException {  upFlow = in.readLong();  downFlow = in.readLong();  sumFlow = in.readLong();  } |

（5）注意反序列化的顺序和序列化的顺序完全一致

（6）要想把结果显示在文件中，需要重写toString()，可用”\t”分开，方便后续用。

## 2.4 序列化案例实操

1. 需求

统计每一个手机号耗费的总上行流量、下行流量、总流量

2．数据准备



输入数据格式：

|  |
| --- |
| 1363157993055 13560436666 C4-17-FE-BA-DE-D9:CMCC 120.196.100.99 18 15 1116 954 200  手机号码 上行流量 下行流量 |

输出数据格式

|  |
| --- |
| 1356·0436666 1116 954 2070  手机号码 总上行流量 总下行流量 总流量 |

3．分析

基本思路：

Map阶段：

（1）读取一行数据，切分字段

（2）抽取手机号、上行流量、下行流量

（3）以手机号为key，bean对象为value输出，即context.write(手机号,bean);

Reduce阶段：

（1）累加上行流量和下行流量得到总流量。

（2）实现自定义的bean来封装流量信息，并将bean作为map输出的key来传输

（3） MR程序在处理数据的过程中会对数据排序(map输出的kv对传输到reduce之前，会排序)，排序的依据是map输出的key

4．编写mapreduce程序

（（1）编写流量统计的bean对象

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  // 1 实现writable接口  public class FlowBean implements Writable{  private long upFlow ;  private long downFlow;  private long sumFlow;    //2 反序列化时，需要反射调用空参构造函数，所以必须有  public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }    //3 写序列化方法  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }    //4 反序列化方法  //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致  @Override  public void readFields(DataInput in) throws IOException {  this.upFlow = in.readLong();  this.downFlow = in.readLong();  this.sumFlow = in.readLong();  }  // 6 编写toString方法，方便后续打印到文本  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  public long getUpFlow() {  return upFlow;  }  public void setUpFlow(long upFlow) {  this.upFlow = upFlow;  }  public long getDownFlow() {  return downFlow;  }  public void setDownFlow(long downFlow) {  this.downFlow = downFlow;  }  public long getSumFlow() {  return sumFlow;  }  public void setSumFlow(long sumFlow) {  this.sumFlow = sumFlow;  }  } |

（2）编写mapper

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{    FlowBean v = new FlowBean();  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割字段  String[] fields = line.split("\t");    // 3 封装对象  // 取出手机号码  String phoneNum = fields[1];  // 取出上行流量和下行流量  long upFlow = Long.parseLong(fields[fields.length - 3]);  long downFlow = Long.parseLong(fields[fields.length - 2]);    v.setUpFlow(upFlow);  v.setDownFlow(downFlow);  k.set(phoneNum);  // 4 写出  context.write(k,v);  }  } |

（3）编写reducer

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {  @Override  protected void reduce(Text key, Iterable<FlowBean> values, Context context)  throws IOException, InterruptedException {  long sum\_upFlow = 0;  long sum\_downFlow = 0;  // 1 遍历所用bean，将其中的上行流量，下行流量分别累加  for (FlowBean flowBean : values) {  sum\_upFlow += flowBean.getSumFlow();  sum\_downFlow += flowBean.getDownFlow();  }  // 2 封装对象  FlowBean resultBean = new FlowBean(sum\_upFlow, sum\_downFlow);    // 3 写出  context.write(key, resultBean);  }  } |

（4）编写驱动

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  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;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

# [理解记忆]第3章 MapReduce框架原理

## 3.1 MapReduce工作流程

1．流程示意图，如图所示



图 MapReduce详细工作流程（一）

2．流程详解

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

1）maptask收集我们的map()方法输出的kv对，放到内存缓冲区中

2）从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件

3）多个溢出文件会被合并成大的溢出文件

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

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

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

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

3．注意

Shuffle中的缓冲区大小会影响到mapreduce程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快。

缓冲区的大小可以通过参数调整，参数：io.sort.mb 默认100M。

## 3.2 InputFormat数据输入

2．FileInputFormat操作流程

（1）找到你数据存储的目录。

（2）开始遍历处理（规划切片）目录下的每一个文件

（3）遍历第一个文件xx.txt

a）获取文件大小fs.sizeOf(xx.txt)

b）默认情况下，切片大小=blocksize

c）开始切，形成第1个切片：ss.txt—0:128M 第2个切片ss.txt—128:256M 第3个切片ss.txt—256M:300M（每次切片时，都要判断切完剩下的部分是否大于块的1.1倍，不大于1.1倍就划分一块切片） 剩下部分大于128m但是小于140m

e）将切片信息写到一个切片规划文件中

f）数据切片只是在逻辑上对输入数据进行分片，并不会再磁盘上将其切分成分片进行存储。使用**InputSplit只记录了分片的元数据信息**，比如起始位置、长度以及所在的节点列表等

h）注意：block是HDFS物理上存储的数据，切片是对数据逻辑上的划分

（4）提交切片规划文件到yarn上，yarn上的MrAppMaster就可以根据切片规划文件计算开启maptask个数。

### 3.2.2 FileInputFormat切片机制

1．FileInputFormat中默认的切片机制

（1）简单地按照文件的内容长度进行切片

（2）切片大小，默认等于block大小

（3）切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

比如待处理数据有两个文件：

|  |
| --- |
| file1.txt 320M  file2.txt 10M |

经过FileInputFormat的切片机制运算后，形成的切片信息如下：

|  |
| --- |
| file1.txt.split1-- 0~128  file1.txt.split2-- 128~256  file1.txt.split3-- 256~320  file2.txt.split1-- 0~10M |

### 3.2.3 CombineTextInputFormat切片机制

关于大量小文件的优化策略

默认情况下TextInputformat对任务的切片机制是按文件规划切片，不管文件多小，都会是一个单独的切片，都会交给一个maptask，这样如果有大量小文件，就会产生大量的maptask，处理效率极其低下。

1．优化策略

（1）最好的办法，在数据处理系统的最前端（预处理/采集），将小文件先合并成大文件，再上传到HDFS做后续分析。

（2）补救措施：如果已经是大量小文件在HDFS中了，可以使用另一种InputFormat来做切片（CombineTextInputFormat），它的切片逻辑跟TextFileInputFormat不同：它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个maptask。

（3）优先满足最小切片大小，不超过最大切片大小

CombineTextInputFormat.*setMaxInputSplitSize*(job, 4194304); // 4m

CombineTextInputFormat.*setMinInputSplitSize*(job, 2097152); // 2m

举例：0.5m+1m+0.3m+5m=2m + 4.8m=2m + 4m + 0.8m

2．具体实现步骤

// 如果不设置InputFormat,它默认用的是TextInputFormat.class

job.setInputFormatClass(CombineTextInputFormat.class)

CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);// 4m

CombineTextInputFormat.setMinInputSplitSize(job, 2097152);// 2m

### 3.2.4 CombineTextInputFormat案例实操

1．需求

将输入的大量小文件合并成一个切片统一处理。

2．输入数据

准备5个小文件

3．实现过程

（1）不做任何处理，运行需求1中的wordcount程序，观察切片个数为5，如图8所示

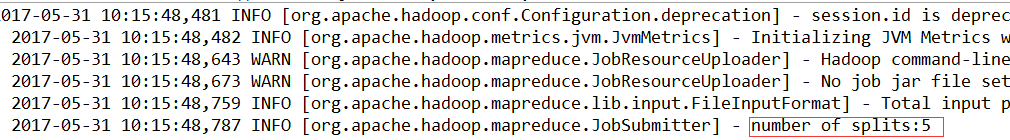


图 不使用CombineTextInputFormat

（2）在WordcountDriver中增加如下代码，运行程序，并观察运行的切片个数为1，如图所示

// 如果不设置InputFormat，它默认用的是TextInputFormat.class

job.setInputFormatClass(CombineTextInputFormat.class);

CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);// 4m

CombineTextInputFormat.setMinInputSplitSize(job, 2097152);// 2m

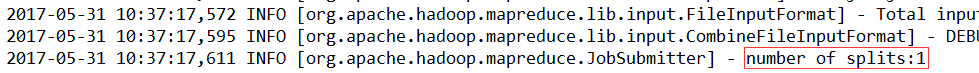


图 使用CombineTextInputFormat

## 3.3 MapTask工作机制

### 3.3.1并行度决定机制

1．问题引出

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并行任务是否越多越好呢？

2．MapTask并行度决定机制

一个job的map阶段MapTask并行度（个数），由客户端提交job时的切片个数决定，如图所示。



图 MapTask并行度决定机制

### 3.3.2 MapTask工作机制

MapTask工作机制如图所示



图 MapTask工作机制

（1）Read阶段：Map Task通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value。

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

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

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

（5）Combine阶段：当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

在进行文件合并过程中，MapTask以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并io.sort.factor（默认100）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

## 3.4 Shuffle机制

### 3.4.1 Shuffle机制(洗牌)

Mapreduce确保每个reducer的输入都是按key排序的。系统执行排序的过程（即将mapper输出作为输入传给reducer）称为shuffle，如图所示。



图 shuffle机制

### 3.4.2 Partition分区

分区：把数据扎堆存放

问题引出：要求将统计结果按照条件输出到不同文件中（分区）。比如：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1. 默认partition分区 hello-->hash%reducetask数量--2 ；0，1

public class HashPartitioner<K, V> extends Partitioner<K, V> {

public int getPartition(K key, V value, int numReduceTasks) {

return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;

}

}

默认分区是根据key的hashCode对reduceTasks个数取模得到的。用户没法控制哪个key存储到哪个分区。

2．自定义Partitioner步骤

（1）自定义类继承Partitioner，重写getPartition()方法

|  |
| --- |
| public class ProvincePartitioner extends Partitioner<Text, FlowBean> {  @Override  public int getPartition(Text key, FlowBean value, int numPartitions) {  // 1 获取电话号码的前三位  String preNum = key.toString().substring(0, 3);    int partition = 4;    // 2 判断是哪个省  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

（2）在job驱动中，设置自定义partitioner：

|  |
| --- |
| job.setPartitionerClass(CustomPartitioner.class); |

（3）自定义partition后，要根据自定义partitioner的逻辑设置相应数量的reduce task

|  |
| --- |
| job.setNumReduceTasks(5); |

1. 注意

reduceTask的个数决定了有几个文件！！

如果reduceTask的数量> getPartition的结果数，则会多产生几个空的输出文件part-r-000xx；

如果1<reduceTask的数量<getPartition的结果数，则有一部分分区数据无处安放，会Exception；

如果reduceTask的数量=1，则不管mapTask端输出多少个分区文件，最终结果都交给这一个reduceTask，最终也就只会产生一个结果文件 part-r-00000；

例如：假设自定义分区数为5，则

（1）job.setNumReduceTasks(1);会正常运行，只不过会产生一个输出文件

（2）job.setNumReduceTasks(2);会报错

（3）job.setNumReduceTasks(6);大于5，程序会正常运行，会产生空文件

### 3.4.3 Partition分区案例实操

1．需求

将统计结果按照手机归属地不同省份输出到不同文件中（分区）

2．数据准备



3．分析

（1）Mapreduce中会将map输出的kv对，按照相同key分组，然后分发给不同的reducetask。默认的分发规则为：根据key的hashcode%reducetask数来分发

（2）如果要按照我们自己的需求进行分组，则需要改写数据分发（分组）组件Partitioner

自定义一个CustomPartitioner继承抽象类：Partitioner

（3）在job驱动中，设置自定义partitioner： job.setPartitionerClass(CustomPartitioner.class)

4．在案例2.4的基础上，增加一个分区类

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Partitioner;  public class ProvincePartitioner extends Partitioner<Text, FlowBean> {  @Override  public int getPartition(Text key, FlowBean value, int numPartitions) {  // 1 获取电话号码的前三位  String preNum = key.toString().substring(0, 3);    int partition = 4;    // 2 判断是哪个省  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

5．在驱动函数中增加自定义数据分区设置和reduce task设置

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  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;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  **// 8 指定自定义数据分区**  **job.setPartitionerClass(ProvincePartitioner.class);**  **// 9 同时指定相应数量的reduce task**  **job.setNumReduceTasks(5);**    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### 3.4.4 WritableComparable排序

排序是MapReduce框架中最重要的操作之一。Map Task和Reduce Task均会对数据（按照key）进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。默认排序是按照字典顺序排序。

对于Map Task，它会将处理的结果暂时放到一个缓冲区中，当缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次排序，并将这些有序数据写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件。

对于Reduce Task，它从每个Map Task上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到一定阈值，则进行一次合并以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据写到磁盘上。当所有数据拷贝完毕后，Reduce Task统一对内存和磁盘上的所有数据进行一次合并。

每个阶段的默认排序

1．排序的分类

（1）部分排序：

MapReduce根据输入记录的键对数据集排序。保证输出的每个文件内部排序。

（2）全排序：

如何用Hadoop产生一个全局排序的文件？最简单的方法是使用一个分区。但该方法在处理大型文件时效率极低，因为一台机器必须处理所有输出文件，从而完全丧失了MapReduce所提供的并行架构。

替代方案：首先创建一系列排好序的文件；其次，串联这些文件；最后，生成一个全局排序的文件。主要思路是使用一个分区来描述输出的全局排序。例如：可以为上述文件创建3个分区，在第一分区中，记录的单词首字母a-g，第二分区记录单词首字母h-n, 第三分区记录单词首字母o-z。

2．自定义排序WritableComparable

（1）原理分析

bean对象实现WritableComparable接口重写compareTo方法，就可以实现排序

@Override

public int compareTo(FlowBean o) {

// 倒序排列，从大到小

return this.sumFlow > o.getSumFlow() ? -1 : 1;

if(this.sumFlow==o.getSumFlow()){

This.downFlow>o.getDownFlow() ? -1 :1

}

}

### 3.4.5 WritableComparable排序案例实操

案例一

1．需求

根据案例2.4产生的结果再次对总流量进行排序。

2．数据准备



3．分析

（1）把程序分两步走，第一步正常统计总流量，第二步再把结果进行排序

（2）context.write(bean<总流量>，手机号)

（3）FlowBean实现WritableComparable接口重写compareTo方法

|  |
| --- |
| @Override  public int compareTo(FlowBean o) {  // 倒序排列，从大到小  return this.sumFlow > o.getSumFlow() ? -1 : 1;  } |

4．代码实现

（1）FlowBean对象在在需求1基础上增加了比较功能

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.WritableComparable;  public class FlowBean implements WritableComparable<FlowBean> {  private long upFlow;  private long downFlow;  private long sumFlow;  // 反序列化时，需要反射调用空参构造函数，所以必须有  public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }  public void set(long upFlow, long downFlow) {  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }  public long getSumFlow() {  return sumFlow;  }  public void setSumFlow(long sumFlow) {  this.sumFlow = sumFlow;  }  public long getUpFlow() {  return upFlow;  }  public void setUpFlow(long upFlow) {  this.upFlow = upFlow;  }  public long getDownFlow() {  return downFlow;  }  public void setDownFlow(long downFlow) {  this.downFlow = downFlow;  }  /\*\*  \* 序列化方法  \* @param out  \* @throws IOException  \*/  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }  /\*\*  \* 反序列化方法 注意反序列化的顺序和序列化的顺序完全一致  \* @param in  \* @throws IOException  \*/  @Override  public void readFields(DataInput in) throws IOException {  upFlow = in.readLong();  downFlow = in.readLong();  sumFlow = in.readLong();  }  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  @Override  public int compareTo(FlowBean o) {  // 倒序排列，从大到小  return this.sumFlow > o.getSumFlow() ? -1 : 1;  }  } |

（2）编写mapper

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean, Text>{  FlowBean bean = new FlowBean();  Text v = new Text();  @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 封装对象  String phoneNbr = fields[0];  long upFlow = Long.parseLong(fields[1]);  long downFlow = Long.parseLong(fields[2]);    bean.set(upFlow, downFlow);  v.set(phoneNbr);    // 4 输出  context.write(bean, v);  }  } |

（3）编写reducer

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text, FlowBean>{  @Override  protected void reduce(FlowBean key, Iterable<Text> values, Context context)  throws IOException, InterruptedException {    for (Text text : values) {  context.write(text, key);  }  }  } |

（4）编写driver

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.IOException;  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;  public class FlowCountSortDriver {  public static void main(String[] args) throws ClassNotFoundException, IOException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowCountSortDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountSortMapper.class);  job.setReducerClass(FlowCountSortReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(FlowBean.class);  job.setMapOutputValueClass(Text.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

案例二

1．需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2．分析

基于前一个需求，增加自定义分区类即可。

3．案例实操

（1）增加自定义分区类

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Partitioner;  public class ProvincePartitioner extends Partitioner<FlowBean, Text> {  @Override  public int getPartition(FlowBean key, Text value, int numPartitions) {    // 1 获取手机号码前三位  String preNum = value.toString().substring(0, 3);    int partition = 4;    // 2 根据手机号归属地设置分区  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

（2）在驱动类中添加分区类

|  |
| --- |
| // 加载自定义分区类  job.setPartitionerClass(FlowSortPartitioner.class);  // 设置Reducetask个数  job.setNumReduceTasks(5); |

### 3.4.6 Combiner合并 <b,1> <b,1>==<b,2>

1）combiner是MR程序中Mapper和Reducer之外的一种组件。

2）combiner组件的父类就是Reducer。

3）combiner和reducer的区别在于运行的位置：

Combiner是在每一个maptask所在的节点运行;

Reducer是接收全局所有Mapper的输出结果；

4）combiner的意义就是对每一个maptask的输出进行局部汇总，以减小网络传输量。

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

Mapper

Maptask1:

3 5 7 ->(3+5+7)/3=5

Maptask2:

2 6 ->(2+6)/2=4

Reducer

(3+5+7+2+6)/5=23/5 不等于 (5+4)/2=9/2

6）自定义Combiner实现步骤：

（1）自定义一个combiner继承Reducer，重写reduce方法

|  |
| --- |
| public class WordcountCombiner extends Reducer<Text, IntWritable, Text,  IntWritable>{  @Override  protected void reduce(Text key, Iterable<IntWritable> values,  Context context) throws IOException, InterruptedException {  // 1 汇总操作  int count = 0;  for(IntWritable v :values){  count += v.get();  }  // 2 写出  context.write(key, new IntWritable(count));  }  } |

（2）在job驱动类中设置：

|  |
| --- |
| job.setCombinerClass(WordcountCombiner.class); |

### 3.4.7 Combiner合并案例实操

1．需求

统计过程中对每一个maptask的输出进行局部汇总，以减小网络传输量即采用Combiner功能，如图所示。



图 Combiner的合并案例

2．数据准备



**方案一**

1）增加一个WordcountCombiner类继承Reducer

|  |
| --- |
| package com.bigdata.mr.combiner;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class WordcountCombiner extends Reducer<Text, IntWritable, Text, IntWritable>{  @Override  protected void reduce(Text key, Iterable<IntWritable> values,  Context context) throws IOException, InterruptedException {  // 1 汇总  int count = 0;  for(IntWritable v :values){  count += v.get();  }  // 2 写出  context.write(key, new IntWritable(count));  }  } |

2）在WordcountDriver驱动类中指定combiner

|  |
| --- |
| // 指定需要使用combiner，以及用哪个类作为combiner的逻辑  job.setCombinerClass(WordcountCombiner.class); |

**方案二**

1）将WordcountReducer作为combiner在WordcountDriver驱动类中指定

// 指定需要使用combiner，以及用哪个类作为combiner的逻辑

job.setCombinerClass(WordcountReducer.class);

运行程序，如图所示

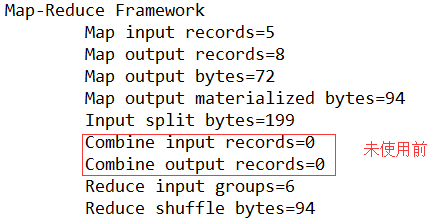


图 未使用前

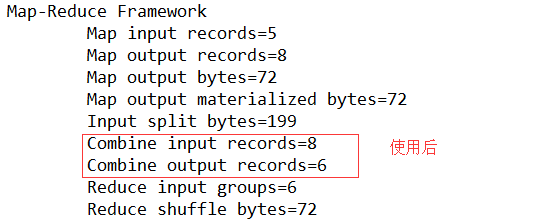
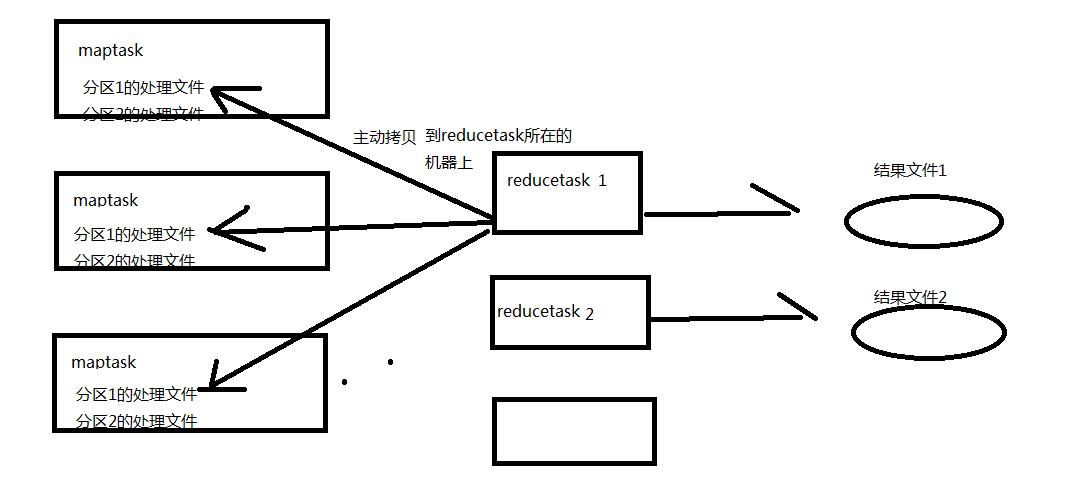


图 使用后

## 3.5 ReduceTask工作机制



1．设置ReduceTask并行度（个数）

reducetask的并行度同样影响整个job的执行并发度和执行效率，但与maptask的并发数由切片数决定不同，Reducetask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

job.setNumReduceTasks(4);

2．注意

（1）reducetask=0 ，表示没有reduce阶段，输出文件个数和map个数一致。

（2）reducetask默认值就是1，所以输出文件个数为一个。

（3）如果数据分布不均匀，就有可能在reduce阶段产生数据倾斜

（4）reducetask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reducetask。

（5）具体多少个reducetask，需要根据集群性能而定。

（6）如果分区数不是1，但是reducetask为1，是否执行分区过程。答案是：不执行分区过程。因为在maptask的源码中，执行分区的前提是先判断reduceNum个数是否大于1。不大于1肯定不执行。

3．实验：测试reducetask多少合适。

（1）实验环境：1个master节点，16个slave节点：CPU:8GHZ，内存: 2G

（2）实验结论：

表4-3 改变reduce task （数据量为1GB）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Map task =16 | | | | | | | | | | |
| Reduce task | 1 | 5 | 10 | 15 | 16 | 20 | 25 | 30 | 45 | 60 |
| 总时间 | 892 | 146 | 110 | 92 | 88 | 100 | 128 | 101 | 145 | 104 |

4．ReduceTask工作机制

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

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

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

（4）Reduce阶段：reduce()函数将计算结果写到HDFS上。