MapReduce 常用三大组件

目录

1、流量统计项目案例

1.1、数据样例

数据样本：

1363157984040 13602846565 5C-0E-8B-8B-B6-00:CMCC 120.197.40.4 2052.flash2-http.qq.com 综合门户 15 12 1938 2910 200

字段释义：

时间戳	ts	long
手机号	phone	String
基站编号	Id	String
IP	Ip	String
url	url	String
url 类型	Type	String
发送数据包	Send	Int
接受数据包	Receive	Int
上行流量	upflow	Long
下行流量	downflow	Long
响应	Status	String

1.2、需求

1、 统计每一个用户（手机号）所耗费的总上行流量、总下行流量，总流量
2、 得出上题结果的基础之上再加一个需求：将统计结果按照总流量倒序排序
3、 将流量汇总统计结果按照手机归属地不同省份输出到不同文件中

2、MapReduce 中的 Combiner

2.1、什么是 Combiner

Combiner 是 MapReduce 程序中 Mapper 和 Reducer 之外的一种组件，它的作用是在 maptask 之后给 maptask 的结果进行局部汇总，以减轻 reducetask 的计算负载，减少网络传输

2.2、如何使用 Combiner

Combiner 和 Reducer 一样，编写一个类，然后继承 Reducer，reduce 方法中写具体的 Combiner 逻辑，然后在 job 中设置 Combiner 组件：**job.setCombinerClass(FlowSumCombine.class)**

```
public static class FlowSumCombine extends Reducer<Text, FlowBean, Text, FlowBean> {
    FlowBean v = new FlowBean();

    // combiner 的逻辑和 reducer 的逻辑一样
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values,
            Context context) throws InterruptedException, IOException {
        long upFlowCount = 0;
        long downFlowCount = 0;
        for (FlowBean bean : values) {
            upFlowCount += bean.getUpfFlow();
            downFlowCount += bean.getDownFlow();
        }
        v.set(key.toString(), upFlowCount, downFlowCount);
        context.write(key, v);
    }
}
```

2.3、使用 Combiner 注意事项

1、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

2、Combiner 的输出 kv 类型应该跟 Reducer 的输入 kv 类型对应起来
Combiner 的输入 kv 类型应该跟 Mapper 的输出 kv 类型对应起来

3、Combiner 的使用要非常谨慎，因为 Combiner 在 MapReduce 过程中可能调用也可能不调用，可能调一次也可能调多次，所以：
Combiner 使用的原则是：**有或没有都不能影响业务逻辑，都不能影响最终结果**

3、MapReduce 中的序列化

3.1、概述

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（参与序列化的对象的类都要实现 Writable 接口），精简，高效

Hadoop 中的序列化框架已经对基本类型和 null 提供了序列化的实现了。分别是：

byte	ByteWritable
short	ShortWritable
int	IntWritable
long	LongWritable
float	FloatWritable
double	DoubleWritable
String	Text
null	NullWritable

3.2、Java 序列化

以案例为例说明：

```
public class Student implements Serializable{

    public Student() {
        super();
    }

    public Student(int id, String name, int age) {
        super();
        this.id = id;
        this.name = name;
        this.age = age;
    }

    private int id;
    private String name;
    private int age;

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + ", age=" + age + "]";
    }
}
```

3.3、自定义对象实现 MapReduce 框架的序列化

如果需要将自定义的 bean 放在 key 中传输，则还需要实现 Comparable 接口，因为 MapReduce 框中的 shuffle 过程一定会对 key 进行排序，此时，自定义的 bean 实现的接口应该是：
public class FlowBean **implements WritableComparable<FlowBean>**

以案例为例说明
下面是进行了序列化的 FlowBean 类：

```
package com.ghgj.mr.flow;

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 String phone;
    private long upfFlow;
    private long downFlow;
    private long sumFlow;

    // 序列化框架在反序列化操作创建对象实例时会调用无参构造
```

```
public FlowBean() {
}

public void set(String phone, long upfFlow, long downFlow) {
	this.phone = phone;
	this.upfFlow = upfFlow;
	this.downFlow = downFlow;
	this.sumFlow = upfFlow + downFlow;
}

/*public void set(long upfFlow, long downFlow) {
	this.upfFlow = upfFlow;
	this.downFlow = downFlow;
	this.sumFlow = upfFlow + downFlow;
}*/

public String getPhone() {
	return phone;
}

public void setPhone(String phone) {
	this.phone = phone;
}

public long getUpfFlow() {
	return upfFlow;
}

public void setUpfFlow(long upfFlow) {
	this.upfFlow = upfFlow;
}

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;
    }

    // 序列化方法
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(phone);
        out.writeLong(upfFlow);
        out.writeLong(downFlow);
        out.writeLong(sumFlow);
    }

    // 反序列化方法
    // 注意： 字段的反序列化顺序与序列化时的顺序保持一致,並且类型也一致
    @Override
    public void readFields(DataInput in) throws IOException {
        this.phone = in.readUTF();
        this.upfFlow = in.readLong();
        this.downFlow = in.readLong();
        this.sumFlow = in.readLong();
    }

    @Override
    public String toString() {
        return phone +"\t" + upfFlow + "\t" + downFlow + "\t" + sumFlow;
    }

    @Override
    public int compareTo(FlowBean fb) {
        return (int)(fb.getSumFlow() - this.sumFlow);
    }
}
```

下面是统计上行流量和下行流量之和的 MR 程序 FlowSum：

```
package com.ghgj.mr.flow;

import java.io.IOException;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
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;

public class FlowSum {

    // 在 kv 中传输我们自定义的对象是可以的，但是必须实现 hadoop 的序列化机制 implements Writable, 如果要排序，还要实现 Comparable 接口，hadoop 为我们提供了一个方便的类，叫做 WritableComparable，直接实现就好
    public static class FlowSumMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
        Text k = new Text();
        FlowBean v = new FlowBean();

        @Override
        protected void map(LongWritable key, Text value, Context context)
                throws InterruptedException, IOException {
            // 将读到的一行数据进行字段切分
            String line = value.toString();
            String[] fields = StringUtils.split(line, "\t");

            // 抽取业务所需要的各字段
            String phone = fields[1];
            long upFlow = Long.parseLong(fields[fields.length - 3]);
            long dFlow = Long.parseLong(fields[fields.length - 2]);

            k.set(phone);
            v.set(phone, upFlow, dFlow);
            context.write(k, v);
        }
    }

    public static class FlowSumReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
        FlowBean v = new FlowBean();

        // reduce 方法接收到的 key 是某一组<a 手机号，bean><a 手机号，bean><a 手机号，bean>中的第一个手机号，reduce 方法接收到的 vlaues 是这一组 kv 中的所有 bean 的一个迭代器
        @Override
        protected void reduce(Text key, Iterable<FlowBean> values,
                Context context) throws InterruptedException, IOException {
            long upFlowCount = 0;
            long downFlowCount = 0;
```

```
            for (FlowBean bean : values) {
                upFlowCount += bean.getUpfFlow();
                downFlowCount += bean.getDownFlow();
            }
            v.set(key.toString(), upFlowCount, downFlowCount);
            context.write(key, v);
        }
    }

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

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);
        // 告诉框架，我们的程序所在 jar 包的路径
        job.setJarByClass(FlowSum.class);

        // 告诉框架，我们的程序所用的 mapper 类和 reducer 类
        job.setMapperClass(FlowSumMapper.class);
        job.setReducerClass(FlowSumReducer.class);

        // 告诉框架，我们的 mapperreducer 输出的数据类型
        /*
         * job.setMapOutputKeyClass(Text.class);
         * job.setMapOutputValueClass(FlowBean.class);
         */
        // 如果 map 阶段输出的数据类型跟最终输出的数据类型一致，就只要以下两行代码来指定
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        // 框架中默认的输入输出组件就是这俩货，所以可以省略这两行代码
        /*
         * job.setInputFormatClass(TextInputFormat.class);
         * job.setOutputFormatClass(TextOutputFormat.class);
         */

        // 告诉框架，我们要处理的文件在哪个路径下
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        // 告诉框架，我们的处理结果要输出到哪里去
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        boolean res = job.waitForCompletion(true);
        System.exit(res ? 0 : 1);
```

```
    }
}
```

4、MapReduce 中的 Sort

需求：把上例求得的流量综合从大到小倒序排

基本思路：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
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，所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到 key 中，让 key 实现接口：WritableComparable，然后重写 key 的 compareTo 方法

下面是 MapReduce 程序 FlowSumSort 的实现：

```
package com.ghgj.mr.flow;

import java.io.IOException;

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

/**
 * 实现流量汇总并且按照流量大小倒序排序 前提：处理的数据是已经汇总过的结果文件
 */
public class FlowSumSort {
    public static class FlowSumSortMapper extends Mapper<LongWritable, Text, FlowBean,
Text> {
        FlowBean k = new FlowBean();
        Text v = new Text();

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

```
                String phone = fields[0];
                long upFlowSum = Long.parseLong(fields[2]);
                long dFlowSum = Long.parseLong(fields[3]);
                k.set(phone, upFlowSum, dFlowSum);
                v.set(phone);
                context.write(k, v);
            }
        }

        public static class FlowSumSortReducer extends Reducer<FlowBean, Text, Text, FlowBean> {
            @Override
            protected void reduce(FlowBean bean, Iterable<Text> phones, Context context) throws
IOException, InterruptedException {
                context.write(phones.iterator().next(), bean);
            }
        }

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

            Configuration conf = new Configuration();

            Job job = Job.getInstance(conf);
            job.setJarByClass(FlowSumSort.class);

            // 告诉框架，我们的程序所用的 mapper 类和 reducer 类
            job.setMapperClass(FlowSumSortMapper.class);
            job.setReducerClass(FlowSumSortReducer.class);

            job.setMapOutputKeyClass(FlowBean.class);
            job.setMapOutputValueClass(Text.class);

            // 告诉框架，我们的 mapperreducer 输出的数据类型
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(FlowBean.class);

            // 告诉框架，我们要处理的文件在哪个路径下
            FileInputFormat.setInputPaths(job, new Path(args[0]));
            // 告诉框架，我们的处理结果要输出到哪里去
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            boolean res = job.waitForCompletion(true);
            System.exit(res ? 0 : 1);
        }
}
```

5、MapReduce 中的 Partitioner

需求：根据归属地输出流量统计数据结果到不同文件，以便于在查询统计结果时可以定位到省级范围进行

思路：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，所以：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
然后在 job 对象中，设置自定义 partitioner： **job.setPartitionerClass(ProvincePartitioner.class)**

下面是 MapReduce 程序实现
首先看 Partitioner：

```
package com.ghgj.mr.flow;

import java.util.HashMap;

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

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {
	private static HashMap<String, Integer> provincMap = new HashMap<String, Integer>();
	static {
		provincMap.put("138", 0);
		provincMap.put("139", 1);
		provincMap.put("136", 2);
		provincMap.put("137", 3);
		provincMap.put("135", 4);
	}

	@Override
	public int getPartition(Text key, FlowBean value, int numPartitions) {

		Integer code = provincMap.get(key.toString().substring(0, 3));
		if (code != null) {
			return code;
		}
		return 5;
	}
}
```

再看 MapReduce 程序 FlowSumProvince：

```
package com.ghgj.mr.flow;
```

```
import java.io.IOException;

import org.apache.commons.lang.StringUtils;
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.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;

public class FlowSumProvince {
    public static class FlowSumProvinceMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
        Text k = new Text();
        FlowBean v = new FlowBean();

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            // 将读到的一行数据进行字段切分
            String line = value.toString();
            String[] fields = StringUtils.split(line, "\t");

            // 抽取业务所需要的各字段
            String phone = fields[1];
            long upFlow = Long.parseLong(fields[fields.length - 3]);
            long downFlow = Long.parseLong(fields[fields.length - 2]);

            k.set(phone);
            v.set(phone, upFlow, downFlow);
            context.write(k, v);
        }
    }

    public static class FlowSumProvinceReducer extends Reducer<Text, FlowBean, Text, FlowBean> {

        @Override
        protected void reduce(Text key, Iterable<FlowBean> values, Context context) throws
```

```
IOException, InterruptedException {
                int upCount = 0;
                int downCount = 0;
                for (FlowBean bean : values) {
                    upCount += bean.getUpfFlow();
                    downCount += bean.getDownFlow();
                }
                FlowBean sumBean = new FlowBean();
                sumBean.set(key.toString(), upCount, downCount);
                context.write(key, sumBean);
            }
        }

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

            Configuration conf = new Configuration();

            Job job = Job.getInstance(conf);
            job.setJarByClass(FlowSumProvince.class);

            // 告诉框架，我们的程序所用的 mapper 类和 reducer 类
            job.setMapperClass(FlowSumProvinceMapper.class);
            job.setReducerClass(FlowSumProvinceReducer.class);

            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(FlowBean.class);

            // 告诉框架，我们的 mapperreducer 输出的数据类型
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(FlowBean.class);

            // 设置 shuffle 的分区组件使用我们自定义的分区组件
            job.setPartitionerClass(ProvincePartitioner.class);

            // 设置 reduce task 的数量
            job.setNumReduceTasks(6);

            // 告诉框架，我们要处理的文件在哪个路径下
            FileInputFormat.setInputPaths(job, new Path(args[0]));

            // 告诉框架，我们的处理结果要输出到哪里去
            Path out = new Path(args[1]);
            FileSystem fs = FileSystem.get(conf);
            if(fs.exists(out)){
```

```
                fs.delete(out, true);
            }

            FileOutputFormat.setOutputPath(job,out);
            boolean res = job.waitForCompletion(true);
            System.exit(res ? 0 : 1);
        }
}
```