二 Hadoop 序列化

2.1 序列化概述

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） 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。

3） 为什么不用 Java 的序列化

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

（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

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）

1）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。

（7）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。

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

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

phone_data.txt

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	上行流量	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

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。

4）编写 mapreduce 程序

（1）编写流量统计的 bean 对象

```
package com.atguigu.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.atguigu.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.set(downFlow, upFlow);

            // 4 写出
            context.write(new Text(phoneNum), new FlowBean(upFlow, downFlow));
        }
}
```

（3）编写 reducer

```
package com.atguigu.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.atguigu.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);
    }
}
```