

MapReduce&Yarn

1 MapReduce概述

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，并行运行在一个Hadoop集群上。

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。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

可以实现上千台服务器集群并发工作，提供数据处理能力。

1.2.2 缺点

1) 不擅长实时计算

MapReduce无法像MySQL一样，在毫秒或者秒级内返回结果。会产生很多IO操作。

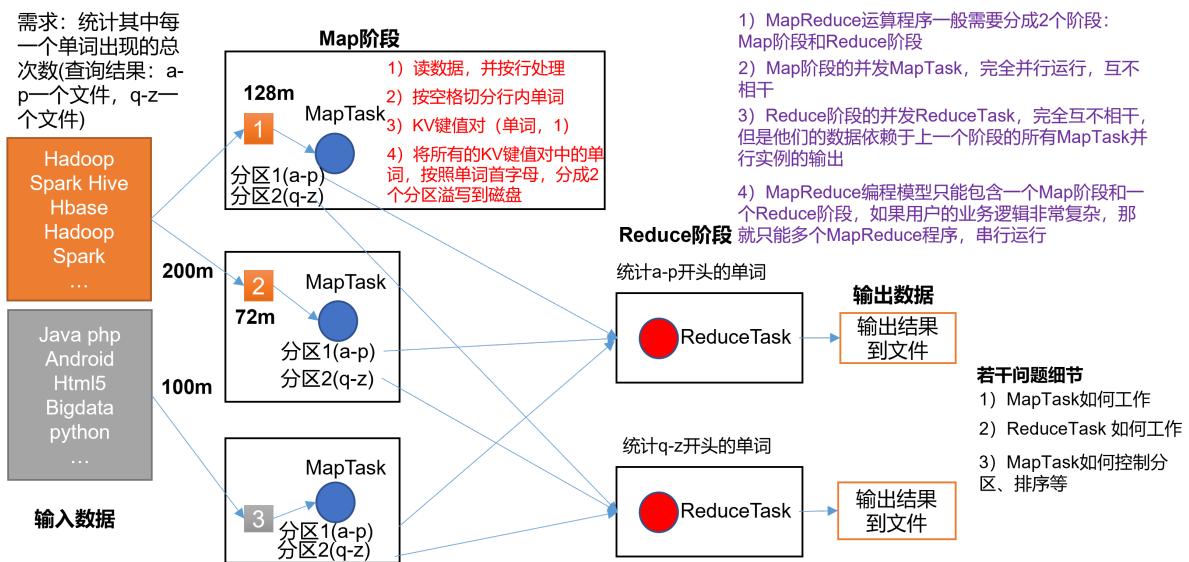
2) 不擅长流式计算

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。

3) 不擅长DAG（有向无环图）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



- (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ReduceTask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。

总结：分析WordCount数据流走向深入理解MapReduce核心思想。

1.4 MapReduce进程

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 官方WordCount源码

采用反编译工具反编译源码，发现WordCount案例有Map类、Reduce类和驱动类。且数据的类型是Hadoop自身封装的序列化类型。

1.6 常用数据序列化类型

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
Null	NullWritable

1.7 MapReduce编程规范

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（继承Hadoop提供的Mapper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（继承Hadoop提供的Reducer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

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

1.8 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) 编程思路

按照MapReduce编程规范，分别编写Mapper，Reducer，Driver。

3) 环境准备

- (1) 创建maven工程

(2) 在pom.xml文件中添加如下依赖

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>2.12.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>
<dependencies>
```

(3) 在项目的src/main/resources目录下，新建一个文件，命名为log4j2.xml，在文件中填入。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error" strict="true" name="XMLConfig">
    <Appenders>
        <!-- 类型名为Console，名称为必须属性 -->
        <Appender type="Console" name="STDOUT">
            <!-- 布局为PatternLayout的方式,
                输出样式为[INFO] [2018-01-22 17:34:01][org.test.Console]I'm here -->
            <Layout type="PatternLayout"
                    pattern="[%-p] [%d{yyyy-MM-dd HH:mm:ss}] [%c{10}]%m%n" />
        </Appender>
    </Appenders>

    <Loggers>
        <!-- 可加性为false -->
        <Logger name="test" level="info" additivity="false">
            <AppenderRef ref="STDOUT" />
        </Logger>

        <!-- root loggerConfig设置 -->
        <Root level="info">
            <AppenderRef ref="STDOUT" />
        </Root>
    </Loggers>
</Configuration>
```

4) 编写程序

(1) 编写Mapper类

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

```

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/15/20:15
 * @Description:
 * 以WordCount案例为例:
 * 自定义的Mapper类 需要继承Hadoop提供的Mapper 并且根据具体业务指定输入数据和输出数据的数据
类型
 *
 * 输入数据类型
 * KEYIN, 读取文件的偏移量 数字(LongWritable)
 * VALUEIN, 读取文件的一行数据 文本(Text)
 * 输出数据类型
 * KEYOUT, 输出数据key的类型 就是一个单词(Text)
 * VALUEOUT 输出数据value的类型 给单词的标记1数字(IntWritable)
 */
public class WordCountMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {

    private Text outKey = new Text();

    private IntWritable outValue = new IntWritable(1);

    /**
     * Map端的核心处理方法, 每输入一行数据会调用一次map方法
     * @param key 输入数据的key
     * @param value 输入数据的value
     * @param context 上下文对象
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, IntWritable>.Context context) throws IOException, InterruptedException {
        //获取当前输入的数据
        String line = value.toString();
        //切割数据
        String[] datas = line.split(" ");
        //遍历集合 封装 输出数据的key和value
        for (String data : datas) {
            outKey.set(data);
            context.write(outKey, outValue);
        }
    }
}

```

(2) 编写Reducer类

```

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

```

```

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/15/20:15
 * @Description:
 * 以wordCount案例为例:
 * 自定义的Reducer类 需要继承Hadoop提供的Reducer 并且根据具体业务指定输入数据和输出数据的数据类型
 *
 * 输入数据类型
 * KEYIN, Map端输出的key的数据类型(Text)
 * VALUEIN, Map端输出的value的数据类型(IntWritable)
 * 输出数据类型
 * KEYOUT, 输出数据key的类型 就是一个单词(Text)
 * VALUEOUT 输出数据value的类型 单词出现的总次数(IntWritable)
 */
public class WordCountReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {

    private Text outKey = new Text();
    private IntWritable outValue = new IntWritable();

    /**
     * Reduce阶段的核心业务处理方法， 一组相同的key的values会调用一次reduce方法
     * @param key
     * @param values
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Reducer<Text,
IntWritable, Text, IntWritable>.Context context) throws IOException,
InterruptedException {
        int totalCount = 0;
        // 遍历values
        for (IntWritable value : values) {
            //对value进行累加 输出结果
            totalCount += value.get();
        }
        //封装key和value
        outKey.set(key);
        outValue.set(totalCount);
        context.write(outKey, outValue);
    }
}

```

(3) 编写Driver驱动类

```

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

```

```

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/15/20:36
 * @Description:
 * MR程序的驱动类，主要用于提交MR任务
 */
public class WordCountDriver {
    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        //声明配置对象
        Configuration conf = new Configuration();
        //声明Job对象
        Job job = Job.getInstance(conf);
        //指定当前Job的驱动类
        job.setJarByClass(WordCountDriver.class);
        //指定当前Job的Mapper和Reducer
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        //指定Map端输出数据key的类型和输出数据value的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        //指定最终输出结果的key的类型和value的类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        //指定输入数据的目录 和 输出数据的目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //提交job
        job.waitForCompletion(true);
    }
}

```

5) 测试

- **本地测试**

- (1) 需要首先配置好HADOOP_HOME变量以及Windows运行依赖
- (2) 在IDEA/Eclipse上运行程序

- **集群测试**

- (1) 将程序打成jar包，然后拷贝到Hadoop集群中

步骤详情：右键->Run as->maven install。等待编译完成就会在项目的target文件夹中生成jar包。如果看不到。在项目上右键->Refresh，即可看到。修改不带依赖的jar包名称为wc.jar，并拷贝该jar包到Hadoop集群。

- (2) 启动Hadoop集群
- (3) 执行WordCount程序

```
hadoop jar MapReduce-1.0-SNAPSHOT.jar com.xulan.mr.wordCountDriver /wcinput  
/wcoutput
```

2 Hadoop序列化

2.1 序列化概述

- **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）以便于存储到磁盘（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）或者是磁盘的持久化数据，转换成内存中的对象。

- **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。

- **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）。

- **Hadoop序列化特点**

- (1) **紧凑**：高效使用存储空间。
- (2) **快速**：读写数据的额外开销小。
- (3) **可扩展**：随着通信协议的升级而可升级。
- (4) **互操作**：支持多语言的交互。

2.2 自定义bean对象实现序列化接口（Writable）

在企业开发中往往常用的基本序列化类型不能满足所有需求，比如在Hadoop框架内部传递一个bean对象，那么该对象就需要实现序列化接口。

具体实现bean对象序列化步骤如下7步。

- (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过程要求对key必须能排序。[详见后面排序案例](#)。

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

2.3 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、总下行流量、总流量

2) 需求分析

1. 输入数据格式

7	13560436666	120.196.100.99	1116	954	200
Id	手机号码	网络ip	上行流量	下行流量	网络状态码

期望输出数据格式

13560436666	1116	954	2070
手机号码	上行流量	下行流量	总流量

2. Map阶段

(1) 读取一行数据，切分字段

7	13560436666	120.196.100.99	1116	954	200
---	-------------	----------------	------	-----	-----

(2) 抽取手机号、上行流量、下行流量

13560436666	1116	954
手机号码	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);

3. Reduce阶段

(1) 累加上行流量和下行流量得到总流量。

13560436666	1116	+	954	=	2070
手机号码	上行流量		下行流量		总流量

3) 编写MapReduce程序

(1) 编写流量统计的Bean对象

```
package com.xulan.mr.writable;

import org.apache.hadoop.io.Writable;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:22
 * @Description:
 * 流量对象 (实现hadoop序列化)
 */
public class FlowBean implements Writable {

    private Integer upFlow;
    private Integer downFlow;
    private Integer sumFlow;

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

    public Integer getUpFlow() {
        return upFlow;
    }

    public Integer getDownFlow() {
        return downFlow;
    }

    public Integer getSumFlow() {
        return sumFlow;
    }

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

```

public void setSumFlow(Integer sumFlow) {
    this.sumFlow = sumFlow;
}

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

/**
 * 序列化方法
 * @param out
 * @throws IOException
 */
@Override
public void write(DataOutput out) throws IOException {
    out.writeInt(upFlow);
    out.writeInt(downFlow);
    out.writeInt(sumFlow);
}

/**
 * 反序列化方法
 * @param in
 * @throws IOException
 */
@Override
public void readFields(DataInput in) throws IOException {
    upFlow = in.readInt();
    downFlow = in.readInt();
    sumFlow = in.readInt();
}
}

```

(2) 编写Mapper类

```

package com.xulan.mr.writable;

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowMapper extends Mapper<LongWritable, Text, Text, FlowBean> {

    private Text outKey = new Text();
    private FlowBean outValue = new FlowBean();
}

```

```

    /**
     * 核心业务逻辑处理
     * @param key
     * @param value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, FlowBean>.Context context) throws IOException, InterruptedException {
        //获取当前行数据
        String line = value.toString();
        //切割数据
        String[] phoneData = line.split("\t");

        //当前数据: 7 13560436666 120.196.100.99 1116 954 200
        //获取输出数据的key(手机号)
        outKey.set(phoneData[1]);
        //获取输出数据的value
        int up = Integer.parseInt(phoneData[phoneData.length - 3]);
        int down = Integer.parseInt(phoneData[phoneData.length - 2]);
        outValuesetUpFlow(up);
        outValue.setDownFlow(down);
        outValue.setSumFlow(up+down);

        //将数据输出
        context.write(outKey,outValue);
    }
}

```

(3) 编写Reducer类

```

package com.xulan.mr.writable;

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowReduce extends Reducer<Text, FlowBean, Text, FlowBean> {

    private Text outKey = new Text();
    private FlowBean outValue = new FlowBean();

    /**
     * 核心业务逻辑处理
     * @param key
     * @param values
     * @param context
     */
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Reducer<Text, FlowBean, Text, FlowBean>.Context context) throws IOException, InterruptedException {
        outValuesetUpFlow(0);
        for (FlowBean value : values) {
            outValuesetUpFlow(outValue.setUpFlow(value.setUpFlow));
            outValue.setDownFlow(outValue.getDownFlow() + value.getDownFlow());
            outValue.setSumFlow(outValue.getSumFlow() + value.getSumFlow());
        }
        context.write(outKey,outValue);
    }
}

```

```

    * @throws IOException
    * @throws InterruptedException
    */
@Override
protected void reduce(Text key, Iterable<FlowBean> values, Reducer<Text, FlowBean, Text, FlowBean>.Context context) throws IOException, InterruptedException {
    //遍历一组相同key的values
    int totalUpFlow = 0;
    int totalDownFlow = 0;
    int totalSumFlow = 0;

    for (FlowBean value : values) {
        totalUpFlow += value.getUpFlow();
        totalDownFlow += value.getDownFlow();
        totalSumFlow += value.getSumFlow();
    }

    //封装输出数据的key
    outKey.set(key);
    //封装输出数据的value
    outValuesetUpFlow(totalUpFlow);
    outValue.setDownFlow(totalDownFlow);
    outValue.setSumFlow(totalSumFlow);

    //将数据输出
    context.write(outKey, outValue);
}
}

```

(4) 编写Driver驱动类

```

package com.xulan.mr.writable;

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowDriver {
    public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(FlowDriver.class);
    }
}

```

```

        job.setMapperClass(FlowMapper.class);
        job.setReducerClass(FlowReduce.class);

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

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

        FileInputFormat.setInputPaths(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\资料\\07_测试数据\\phone_data"));
        FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\out\\data_1"));

        job.waitForCompletion(true);
    }
}

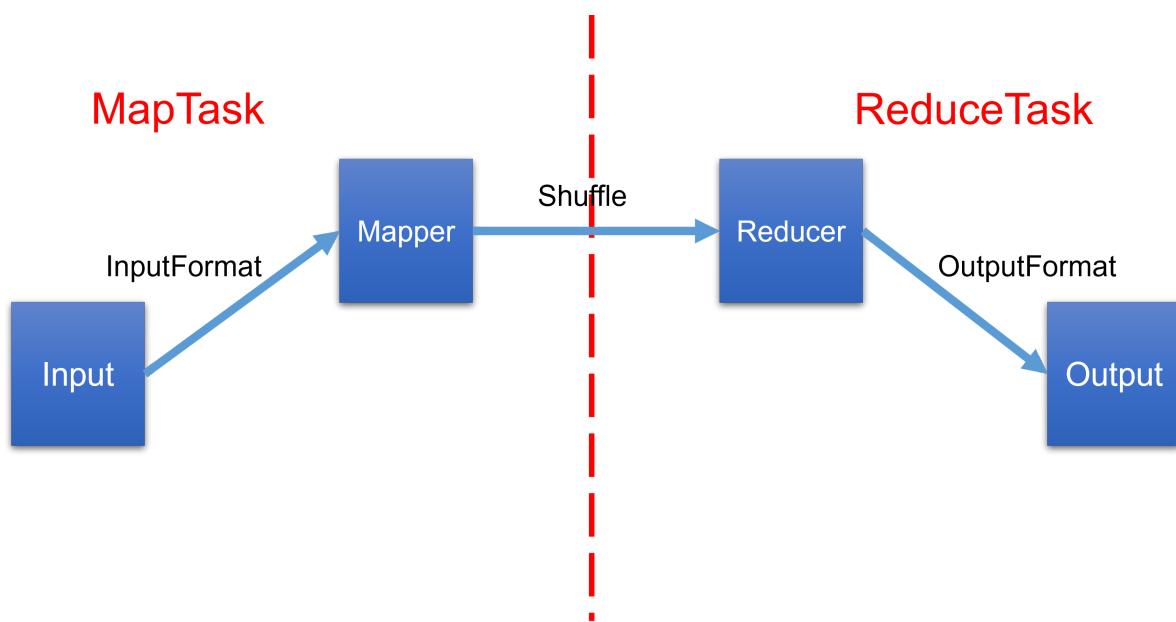
```

3 MapReduce框架原理

3.1 InputFormat数据输入

MapReduce的执行大概流程

简易版:	InputFormat	-->	Mapper	-->	Reducer	-->	OutputFormat
详细版:	InputFormat	-->	map sort	-->	copy sort reduce	-->	OutputFormat



```

//MapTask源码:
if (isMapTask()) {
    // If there are no reducers then there won't be any sort. Hence the map
    // phase will govern the entire attempt's progress.
    //如果没有reducer, 将不会进行排序
    if (conf.getNumReduceTasks() == 0) {
        mapPhase = getProgress().addPhase("map", 1.0f);
    } else {
        // If there are reducers then the entire attempt's progress will be
        // split between the map phase (67%) and the sort phase (33%).
    }
}

```

```

    // 如果有reducer, map阶段和排序阶段分配67%和33%
    mapPhase = getProgress().addPhase("map", 0.667f);
    sortPhase = getProgress().addPhase("sort", 0.333f);
}
}

```

```

//ReduceTask源码

if (isMapOrReduce()) {
    copyPhase = getProgress().addPhase("copy"); //将集群中多个MapTask任务输出数据拷贝
    sortPhase = getProgress().addPhase("sort"); //将数据进行归并排序
    reducePhase = getProgress().addPhase("reduce"); //执行reduce方法
}

```

3.1.1 切片与MapTask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。

思考：1G的数据，启动8个MapTask，可以提高集群的并发处理能力。那么1K的数据，也启动8个MapTask，会提高集群性能吗？MapTask并行任务是否越多越好呢？哪些因素影响了MapTask并行度？

2) MapTask并行度决定机制

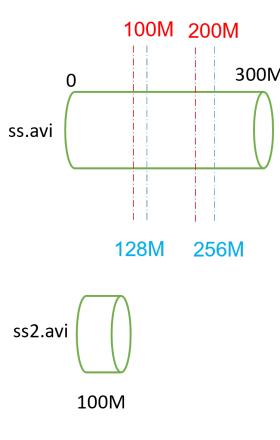
数据块：Block是HDFS物理上把数据分成一块一块。数据块是HDFS存储数据单位。

数据切片：数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。数据切片是MapReduce程序计算输入数据的单位，一个切片会对应启动一个MapTask。

- 切片的概念：从文件的逻辑上的进行大小的切分，一个切片多大，将来一个MapTask的处理的数据就多大。
- 一个切片就会产生一个MapTask
- 切片时只考虑文件本身，不考虑数据的整体集。
- 切片大小和切块大小默认是一致的，这样设计目的为了避免将来切片读取数据的时候有跨机器的情况

1、假设切片大小设置为100M

2、假设切片大小设置为128M

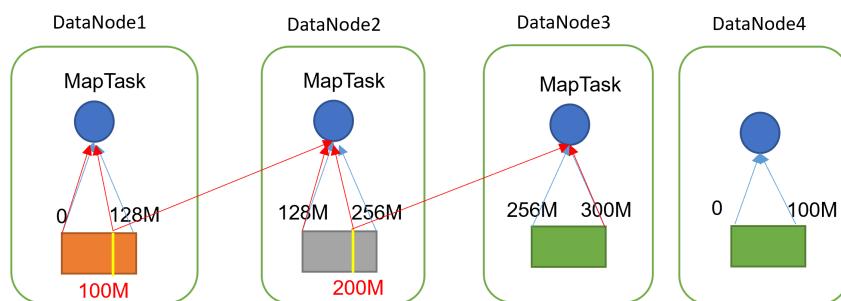


1) 一个Job的Map阶段并行度由客户端在提交Job时的切片数决定

2) 每一个Split切片分配一个MapTask并行实例处理

3) 默认情况下，切片大小=BlockSize

4) 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



3.1.2 InputFormat的体系结构

InputFormat是一个抽象类

- **FileInputFormat:** InputFormat的子实现类，实现切片逻辑。实现了getSplits() 负责切片。
(FileInputFormat也是一个抽象类)
- **TextInputFormat:** FileInputFormat的子实现类，实现读取数据的逻辑。createRecordReader()
返回一个RecordReader，在RecordReader中实现了读取数据的方式：按行读取。
- **CombineTextInputFormat:** FileInputFormat的子实现类，此类中也实现了 一套切片逻辑（处理：适用于小文件计算场景。）

3.1.3 FileInputFormat切片源码解析

FileInputFormat负责切片 (getSplits)

(1) 源码中计算切片大小的公式

```
Math.max(minSize, Math.min(maxSize, blockSize));
```

mapreduce.input.fileinputformat.split.minsize=1 默认值为1

mapreduce.input.fileinputformat.split.maxsize= Long.MAXValue 默认值Long.MAXValue

(2) 切片大小设置

maxsize (切片最大值) : 参数如果调得比blockSize小，则会让切片变小，而且就等于配置的这个参数的值。

minsize (切片最小值) : 参数调的比blockSize大，则可以让切片变得比blockSize还大。

(3) 获取切片信息API

```
// 获取切片的文件名称
String name = inputSplit.getPath().getName();
// 根据文件类型获取切片信息
FileSplit inputSplit = (FileSplit) context.getInputSplit();
```

getSplits源码

```
// 切片源码
public List<Inputsplit> getSplits(JobContext job) throws IOException {
    //java中的计时器
    Stopwatch sw = new Stopwatch().start();

    // minsize = 1(默认情况)
    // 但是我们也可以通过改变mapreduce.input.fileinputformat.split.minsize 配置项来改
    // 变minsize大小
    long minsize = Math.max(getFormatMinsplitsize(), getMinsplitsize(job));

    // maxsize = Long类型的最大值(默认情况)
    // 但是我们也可以通过改变mapreduce.input.fileinputformat.split.maxsize 配置项来改
    // 变maxsize大小
    long maxsize = getMaxSplitsize(job);

    // 管理最终切完片的对象的集合 最终返回的就是此集合
    List<InputSplit> splits = new ArrayList<InputSplit>();

    // 获取当前文件的详情
    FileStatus fileStatus = fs.getFileStatus(new Path(name));
    if (fileStatus != null) {
        long blockSize = fileStatus.getBlockSize();
        if (blockSize > 0) {
            int numSplits = (int) Math.ceil((double) fileStatus.getLength() / blockSize);
            for (int i = 0; i < numSplits; i++) {
                splits.add(new FileSplit(name, i, i + 1, blockSize));
            }
        } else {
            splits.add(new FileSplit(name, 0, 1, 1));
        }
    } else {
        splits.add(new FileSplit(name, 0, 1, 1));
    }
}
```

```

List<FileStatus> files = listStatus(job);

boolean ignoreDirs = !getInputDirRecursive(job)
    &&
job.getConfiguration().getBoolean(INPUT_DIR_NONRECURSIVE_IGNORE_SUBDIRS, false);

// 遍历获取到的文件列表，一次按照文件为单位进行切片
for (FileStatus file: files) {

    // 如果是忽略文件以及是文件夹就不进行切片
    if (ignoreDirs && file.isDirectory()) {
        continue;
    }

    // 获取文件的路径
    Path path = file.getPath();
    // 获取文件的内容大小
    long length = file.getLength();
    // 如果不是空文件 继续切片
    if (length != 0) {

        // 获取文件的具体的块信息
        BlockLocation[] blkLocations;
        if (file instanceof LocatedFileStatus) {
            blkLocations = ((LocatedFileStatus) file).getBlockLocations();
        } else {
            FileSystem fs = path.getFileSystem(job.getConfiguration());
            blkLocations = fs.getFileBlockLocations(file, 0, length);
        }

        // 核心逻辑：判断是否要进行切片（主要判断当前文件是否是压缩文件，有一些压缩文件时
        // 不能够进行切片）
        if (isSplittable(job, path)) {
            // 获取HDFS中的数据块的大小
            long blockSize = file.getBlockSize();
            // 计算切片的大小--> 128M 默认情况下永远都是块大小
            long splitSize = computeSplitsize(blockSize, minsize, maxSize);
            -- 内部方法：
                protected long computeSplitsize(long blockSize, long
minsize, long maxSize) {
                    return Math.max(minsize, Math.min(maxSize, blockSize));
                }

            long bytesRemaining = length;

            // 判断当前的文件的剩余内容是否要继续切片 SPLIT_SLOP = 1.1
            // 判断公式：bytesRemaining)/splitSize > SPLIT_SLOP
            // 用文件的剩余大小/切片大小 > 1.1 才继续切片（这样做的目的是为了让我们每一个MapTask处理的数据更加均衡）
            while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
                int blkIndex = getBlockIndex(blkLocations, length-
bytesRemaining);
                splits.add(makeSplit(path, length-bytesRemaining, splitSize,
                    blkLocations[blkIndex].getHosts(),
                    blkLocations[blkIndex].getCachedHosts()));
                bytesRemaining -= splitSize;
            }
        }
    }
}

```

```

        // 如果最后文件还有剩余且不足一个切片大小，最后再形成最后的一个切片
        if (bytesRemaining != 0) {
            int blkIndex = getBlockIndex(blkLocations, length-
bytesRemaining);
                splits.add(makeSplit(path, length-bytesRemaining,
bytesRemaining,
                                blkLocations[blkIndex].getHosts(),
                                blkLocations[blkIndex].getCachedHosts()));
            }
        } else { // not splitable 不能切分
            if (LOG.isDebugEnabled()) {
                // Log only if the file is big enough to be splitted
                if (length > Math.min(file.getBlocksize(), minsize)) {
                    LOG.debug("File is not splittable so no parallelization
"
                            + "is possible: " + file.getPath());
                }
            }
            splits.add(makeSplit(path, 0, length,
blkLocations[0].getHosts(),
                                blkLocations[0].getCachedHosts()));
        }
    } else {
        //Create empty hosts array for zero length files
        splits.add(makeSplit(path, 0, length, new String[0]));
    }
}
// Save the number of input files for metrics/loadgen
job.getConfiguration().setLong(NUM_INPUT_FILES, files.size());

//计时器停止
sw.stop();
if (LOG.isDebugEnabled()) {
    LOG.debug("Total # of splits generated by getSplits: " + splits.size()
            + ", TimeTaken: " + sw.now(TimeUnit.MILLISECONDS));
}
return splits;
}

```

3.1.4 TextInputFormat读取数据

TextInputFormat实现按行读取 (createRecordReader)

TextInputFormat 实现读取数据的逻辑 `createRecordReader()` 返回一个 `RecordReader`，在 `RecordReader` 中实现了读取数据的方式：按行读取。源码如下：

```

@Override
public RecordReader<LongWritable, Text>
createRecordReader(InputSplit split,
                  TaskAttemptContext context) {
    String delimiter = context.getConfiguration().get(
        "textinputformat.record.delimiter");
    byte[] recordDelimiterBytes = null;
    if (null != delimiter)
        recordDelimiterBytes = delimiter.getBytes(Charsets.UTF_8);
    //按行读取
    return new LineRecordReader(recordDelimiterBytes);
}

```

TextInputFormat是默认的FileInputFormat实现类。按行读取每条记录。键是存储该行在整个文件中的起始字节偏移量，LongWritable类型。值是这行的内容，不包括任何行终止符（换行符和回车符），Text类型。

以下是一个示例，比如，一个分片包含了如下4条文本记录

```

Rich learning form
Intelligent learning engine
Learning more convenient
From the real demand for more close to the enterprise

```

每条记录表示为以下键/值对：

```

(0,Rich learning form)
(19,Intelligent learning engine)
(47,Learning more convenient)
(72,From the real demand for more close to the enterprise)

```

3.1.5 CombineTextInputFormat小文件计算

框架默认的TextInputFormat切片机制是对任务按文件规划切片，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) 应用场景：

CombineTextInputFormat用于小文件过多的场景，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处理。

2) 虚拟存储切片最大值设置

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

注意：虚拟存储切片最大值设置最好根据实际的小文件大小情况来设置具体的值。

3) 切片机制

生成切片过程包括：虚拟存储过程和切片过程二部分。

setMaxInputSplitSize值为4M

		虚拟存储过程	切片过程
a.txt	1.7M	1.7M<4M 划分一块	(a) 判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。
b.txt	5.1M	5.1M>4M 但是小于2*4M 划分二块 块1=2.55M； 块2=2.55M	
c.txt	3.4M	3.4M<4M 划分一块	(b) 如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。
d.txt	6.8M	6.8M>4M 但是小于2*4M 划分二块 块1=3.4M； 块2=3.4M	
		最终存储的文件	最终会形成3个切片，大小分别为：
		1.7M	(1.7+2.55) M, (2.55+3.4) M, (3.4+3.4) M
		2.55M	
		2.55M	
		3.4M	
		3.4M	
		3.4M	

(1) 虚拟存储过程：

将输入目录下所有文件大小，依次和设置的setMaxInputSplitSize值比较，如果不大于设置的最大值，逻辑上划分一个块。如果输入文件大于设置的最大值且大于两倍，那么以最大值切割一块；当剩余数据大小超过设置的最大值且不大于最大值2倍，此时将文件均分成2个虚拟存储块（防止出现太小切片）。

例如setMaxInputSplitSize值为4M，输入文件大小为8.02M，则先逻辑上分成一个4M。剩余的大小为4.02M，如果按照4M逻辑划分，就会出现0.02M的小的虚拟存储文件，所以将剩余的4.02M文件切分成(2.01M和2.01M)两个文件。

(2) 切片过程：

- (a) 判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。
- (b) 如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。
- (c) 测试举例：有4个小文件大小分别为1.7M、5.1M、3.4M以及6.8M这四个小文件，则虚拟存储之后形成6个文件块，大小分别为：

1.7M, (2.55M、2.55M), 3.4M以及 (3.4M、3.4M)

最终会形成3个切片，大小分别为：

(1.7+2.55) M, (2.55+3.4) M, (3.4+3.4) M

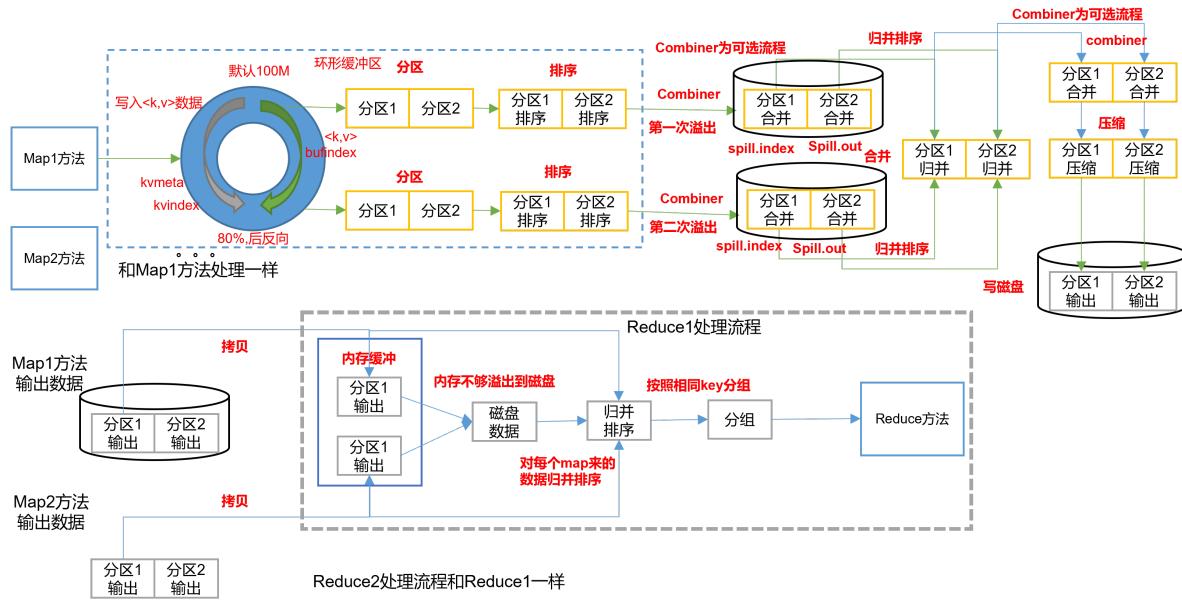
在Driver中为job指定InputFormat的实现类为CombinTextInputFormat

```
//指定CombineFileInputFormat中切片的最大值
CombineFileInputFormat.setMaxInputSplitsize(job, 4194304 * 5);
//指定InputFormat的实现，默认为FileInputFormat
job.setInputFormatClass(CombinTextInputFormat.class);

//指定输入数据的目录 和 输出数据的目录
FileInputFormat.setInputPaths(job, new Path("E:\\learning\\04_java\\02_大数据资料
\\00_hadoop\\资料\\07_测试数据\\combine"));
FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据资料
\\00_hadoop\\out\\data_5"));
```

3.2 Shuffle机制

Map方法之后，Reduce方法之前的数据处理过程称之为Shuffle。



3.2.1 Partition分区

1、Partitioner是Hadoop的分区器对象：负责给Map阶段输出数据选择分区的功能

- 默认实现HashPartitioner类。按照输出的key的 hashCode 值 和 ReduceTask的数量 进行取余操作会得到一个数字，这个数字就只当前<k,v>所属分区的编号，分区编号在Job提交的时候就已经根据指定ReduceTask的数量定义好了。

2、Hadoop默认的分区规则源码解析

- 定位MapTask的map方法中 context.write(outk, outv);
- 跟到write(outk, outv)中进入到 ChainMapContextImpl类的实现中

```
public void write(KEYOUT key, VALUEOUT value) throws IOException,  
    InterruptedException {  
    output.write(key, value);  
}
```

- 跟到 output.write(key, value) 内部 NewOutputCollector

```
public void write(K key, V value) throws IOException, InterruptedException {  
    collector.collect(key, value,  
                      partitioner.getPartition(key, value,  
                      partitions));  
}
```

- 重点理解 partitioner.getPartition(key, value, partitions);
- 跟进默认的分区规则实现 HashPartitioner类

```

public int getPartition(K key, V value,
    int numReduceTasks) {
    // 根据当前的key的hashCode值和ReduceTask的数量进行取余操作
    // 获取到的值就是当前kv所属的分区编号。
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
}

```

3、自定义分区器对象

- 自定一个分区器类，继承Hadoop提供的Partitioner类，实现getPartition() 方法，在方法中编写自己的业务逻辑，最终给当前kv返回所属的分区编号。
- 分区器使用时注意事项
 - 当ReduceTask的数量设置 > 实际用到的分区数 此时会生成空的分区文件
 - 当ReduceTask的数量设置 < 实际用到的分区数 此时会报错
 - 当ReduceTask的数量设置 = 1 结果文件会输出到一个文件中，

由以下源码可以论证：

```

// 获取当前ReduceTask的数量
partitions = jobContext.getNumReduceTasks();
// 判断ReduceTask的数量 是否大于1，找指定分区器对象
if (partitions > 1) {
    partitioner = (org.apache.hadoop.mapreduce.Partitioner<K,V>)
        ReflectionUtils.newInstance(jobContext.getPartitionerClass(), job);
} else {
    // 执行默认的分区规则，最终返回一个唯一的0号分区
    partitioner = new org.apache.hadoop.mapreduce.Partitioner<K,V>() {
        @Override
        public int getPartition(K key, V value, int numPartitions) {
            return partitions - 1;
        }
    };
}

```

- 分区编号生成的规则：根据指定的ReduceTask的数量 从0开始，依次累加。

Partition分区案例实操

1、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）

2、默认Partitioner分区

```

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。

3、自定义Partitioner步骤

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

```
public class CustomPartitioner extends Partitioner<Text, FlowBean> {  
    @Override  
    public int getPartition(Text key, FlowBean value, int numPartitions) {  
        // 控制分区代码逻辑  
        ...  
        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ReduceTask

```
job.setNumReduceTasks(5);
```

4、分区总结

- (1) 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；
- (2) 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；
- (3) 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；
- (4) 分区号必须从零开始，逐一累加。

5、案例分析

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

代码实现

```
public class PhonePartitioner extends Partitioner<Text, FlowBean> {  
  
    /**  
     * 定义当前kv所属分区规则  
     * @param text  
     * @param flowBean  
     * @param numPartitions  
     * @return  
     * 136 --> 0  
     * 137 --> 1  
     * 138 --> 2  
     * 139 --> 3  
     * 其他 --> 4  
     */  
    @Override
```

```

public int getPartition(Text text, FlowBean flowBean, int numPartitions) {
    int phonePartitions = 0;
    String phoneNum = text.toString();
    if(phoneNum.startsWith("136")){
        phonePartitions = 0;
    } else if (phoneNum.startsWith("137")) {
        phonePartitions = 1;
    } else if (phoneNum.startsWith("138")) {
        phonePartitions = 2;
    } else if (phoneNum.startsWith("139")) {
        phonePartitions = 3;
    } else {
        phonePartitions = 4;
    }
    return phonePartitions;
}
}

```

在Driver中设置reduceTask数量和自定义分区对象

```

//指定ReduceTask数量为5
job.setNumReduceTasks(5);
//指定自定义分区对象实现
job.setPartitionerClass(PhonePartitioner.class);

```

3.2.2 WritableComparable排序

1、排序概述

排序是MapReduce框架中最重要的操作之一。

MapTask和ReduceTask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**，且实现该排序的方法是**快速排序**。

- 对于MapTask，它会将处理的结果暂时放到环形缓冲区中，当环形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快速排序，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行归并排序。
- 对于ReduceTask，它从每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写磁盘上，否则存储在内存中。如果磁盘上文件数目达到一定阈值，则进行一次归并排序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溢写到磁盘上。当所有数据拷贝完毕后，ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序。

2、排序分类

(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有序。

(2) 全排序

最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个ReduceTask。但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了MapReduce所提供的并行架构。

(3) 辅助排序：(GroupingComparator分组)

在Reduce端对key进行分组。应用于：在接收的key为bean对象时，想让一个或几个字段相同（全部字段比较不相同）的key进入到同一个reduce方法时，可以采用分组排序。

(4) 二次排序

在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

3、Hadoop中实现比较排序

1. Hadoop 中的实现排序比较的方式

- 直接让参与比较的对象上实现WritableComparable 接口，并在该类中实现 compareTo方法，在compareTo中定义自己的比较规则。这种情况当运行的时候Hadoop会帮助我们生成比较器对象WritableComparator。
- 自定一个比较器对象需要继承Hadoop提供的WritableComparator类，重写该类compare()方法，在该方法中定义比较规则，注意在自定义的比较器对象中通过调用父类的super方法将自定义的比较器对象和要参与比较的对象进行关联。最后再Driver类中指定自定义的比较器对象。

2. Hadoop中获取比较器对象的规则是什么？（通过源码解析分析...）

- 定位到 MapTask 类中的init() 方法

```
// 获取比较器对象
comparator = job.getOutputKeyComparator();
```

- 定位 JobConf 类中 getOutputKeyComparator()

```
// job提交的时候 获取当前MR程序输出数据key的比较器对象
public RawComparator getOutputKeyComparator() {
    Class<? extends RawComparator> theClass = getClass(
        JobContext.KEY_COMPARATOR, null, RawComparator.class);
    if (theClass != null){
        // 如果通过配置获取到指定的比较器对象的class 直接通过反射实例化
        return ReflectionUtils.newInstance(theClass, this);
    }

    // 如果通过配置没获取到指定的比较器对象，接着判断
    // 当前参与比较的对象是否实现了WritableComparable接口
    return WritableComparator.get(getMapOutputKeyClass()
        .asSubclass(WritableComparable.class), this);
}

// get()方法就是实现获取比较器对象的逻辑
public static WritableComparator get(
    Class<? extends WritableComparable> c, Configuration conf) {

    // 根据当前传入的class文件到 comparators的Map中获取比较器对象
    // 这种情况是 当前参与比较的对象的类型是Hadoop自身的数据类型
    WritableComparator comparator = comparators.get(c);

    if (comparator == null) {
        // 考虑到一些极端情况，可能发生GC垃圾回收，导致比较器没了
        // 为了万无一失 再次让类加载一遍
        forceInit(c);
        // 重新加载后再次获取
        comparator = comparators.get(c);
        // 此时还没获取到，那就说明当前参与比较的对象的不是Hadoop自身的数据类型
        if (comparator == null) {
            // Hadoop 会给当前参与比较的对象生成比较器对象
            comparator = new WritableComparator(c, conf, true);
        }
    }
}
```

```

    }
    // Newly passed Configuration objects should be used.
    ReflectionUtils.setConf(comparator, conf);
    return comparator;
}

```

3. Hadoop自身的数据类型是如何拥有比较器对象

- 以Text为例：打开Text的源码

- 当前Text实现了WritableComparable接口
- 在该类中 定义了自己的比较器对象

```

public static class Comparator extends WritableComparator {
    public Comparator() {
        super(Text.class);
    }
}

```

3. 该类中还包含一个静态代码快

```

static {
    // register this comparator
    WritableComparator.define(Text.class, new Comparator());
}

```

4. Text类它的比较器对象被管理到一个Map中，以当前类的class文件为key当前类的比较器对象为value

```

public static void define(Class c, WritableComparator comparator) {
    comparators.put(c, comparator);
}

```

4、排序案例

1) 需求

根据序列化案例产生的结果再次对总流量进行倒序排序。

1、需求：根据手机的总流量进行倒序排序

2、输入数据

13736230513	2481	24681	27162
13846544121	264	0	264
13956435636	132	1512	1644
13509468723	7335	110349	117684
...

3、输出数据

13509468723	7335	110349	117684
13736230513	2481	24681	27162
13956435636	132	1512	1644
13846544121	264	0	264
...

4、FlowBean实现WritableComparable接口重写compareTo方法

```

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

```

6、Reducer类

```

// 循环输出，避免总流量相同情况
for (Text text : values) {
    context.write(text, key);
}

```

5、Mapper类

```
context.write(bean, 手机号)
```

2) 代码实现

- 在FlowBean中实现WritableComparable接口，重写compareTo方法

```

/**
 * 自定义排序规则
 * 根据总流量
 * @param o
 * @return
 */
@Override
public int compareTo(FlowBean o) {
    return -this.getSumFlow().compareTo(o.getSumFlow());
}

```

2、编写Mapper类

```

package com.xulan.mr.writableComparable;

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowMapper extends Mapper<LongWritable, Text, FlowBean, Text> {

    private Text outValue = new Text();
    private FlowBean outKey = new FlowBean();

    /**
     * 核心业务逻辑处理
     * @param key
     * @param value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, FlowBean, Text>.Context context) throws IOException, InterruptedException {
        //获取当前行数据
        String line = value.toString();
        //切割数据
        String[] phoneData = line.split("\t");

        //当前数据: 7 13560436666 120.196.100.99 1116 954 200
        //获取输出数据的key(手机号)
        outValue.set(phoneData[1]);
        //获取输出数据的value
        int up = Integer.parseInt(phoneData[phoneData.length - 3]);
        int down = Integer.parseInt(phoneData[phoneData.length - 2]);
        outKey.setUpFlow(up);
    }
}

```

```

        outKey.setDownFlow(down);
        outKey.setSumFlow(up+down);

        //将数据输出
        context.write(outKey,outValue);
    }
}

```

3、编写Reducer类

```

package com.xulan.mr.writableComparable;

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowReduce extends Reducer<FlowBean, Text, Text, FlowBean> {

    private Text outKey = new Text();
    private FlowBean outValue = new FlowBean();

    /**
     * 核心业务逻辑处理
     * @param key
     * @param values
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Reducer<FlowBean,
    Text, Text, FlowBean>.Context context) throws IOException, InterruptedException
    {
        for (Text value : values) {
            context.write(value,key);
        }
    }
}

```

4、设置Driver

```

package com.xulan.mr.writableComparable;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;

```

```

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/17/20:32
 * @Description:
 */
public class FlowDriver {
    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(FlowDriver.class);
        job.setMapperClass(FlowMapper.class);
        job.setReducerClass(FlowReduce.class);

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

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

        //设置自定义比较器对象
        job.setSortComparatorClass(FlowBeanComparator.class);

        //可将Partition分区应用到这里
        //设置ReduceTask的数量
        job.setNumReduceTasks(5);
        //指定自定义分区
        job.setPartitionerClass(PhonePartitioner.class);

        FileInputFormat.setInputPaths(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\资料\\07_测试数据\\phone_data"));
        FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\out\\data_4"));

        job.waitForCompletion(true);
    }
}

```

3.3.3 Combiner合并

- (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

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

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

Reducer

(3+5+7+2+6)/5=23/5 不等于 (5+4)/2=9/2

Combiner 流程的使用

- Combiner的使用场景：总的来说，为了提升MR程序的运行效率，为了减轻ReduceTask的压力，另外减少IO的开销。
- 使用Combiner
 - 自定一个Combiner类 继承Hadoop提供的Reducer
 - 在Job中指定自定义的Combiner类
- Combiner不适用的场景：Reduce端处理的数据考虑到多个MapTask的数据的整体集时 就不能提前合并了。

3.3 OutputFormat数据输出

3.3.1 OutputFormat接口实现类

OutputFormat是MapReduce输出的基类，所有实现MapReduce输出都实现了 OutputFormat接口。下面我们介绍几种常见的OutputFormat实现类。

1. 概念：OutputFormat主要负责最终数据的写出

- OutputFormat 类的体系结构
 - **FileOutputFormat** OutputFormat的子类 (实现类): 对 checkOutputSpecs() 做了具体的实现(检查提交路径)
 - **TextOutputFormat** FileOutputFormat的子类: 对 getRecordWriter() 做了具体实现
- OutputFormat的使用场景

当我们对MR最终的结果有个性化制定的需求，就可以通过自定义OutputFormat来实现
- 如何实现OutputFormat自定义：
 - ① 自定一个 OutputFormat 类，继承Hadoop提供的OutputFormat，在该类中实现 getRecordWriter() ,返回一个RecordWriter
 - ② 自定义一个 RecordWriter 并且继承Hadoop提供的RecordWriter类，在该类中重写 write() 和 close() 在这些方法中完成自定义输出。

2. 文本输出TextOutputFormat

默认的输出格式是TextOutputFormat，它把每条记录写为文本行。它的键和值可以是任意类型，因为TextOutputFormat调用toString()方法把它们转换为字符串。

3. SequenceFileOutputFormat

将SequenceFileOutputFormat输出作为后续 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。

4. 自定义OutputFormat

根据用户需求，自定义实现输出。

5. 使用场景

为了实现控制最终文件的输出路径和输出格式，可以自定义OutputFormat。

例如：要在一个MapReduce程序中根据数据的不同输出两类结果到不同目录，这类灵活的输出需求可以通过自定义OutputFormat来实现。

6. 自定义OutputFormat步骤

- (1) 自定义一个类继承FileOutputFormat。
- (2) 改写RecordWriter，具体改写输出数据的方法write()。

3.3.2 自定义OutputFormat案例实操

1) 需求分析

过滤输入的log日志，包含xu1an的网站输出到e:/xu1an.log，不包含xu1an的网站输出到e:/other.log。

2) 案例实操

- (1) 编写LogMapper类

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/16:01
 * @Description:
 */
public class LogMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

    /**
     * 核心处理方法
     * @param key
     * @param value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, NullWritable>.Context context) throws IOException, InterruptedException {
        //直接写出
        context.write(value, NullWritable.get());
    }
}
```

- (2) 编写LogReducer类

```

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/16:02
 * @Description:
 */
public class LogReducer extends Reducer<Text, NullWritable, Text, NullWritable> {
    @Override
    protected void reduce(Text key, Iterable<NullWritable> values, Reducer<Text, NullWritable, Text, NullWritable>.Context context) throws IOException, InterruptedException {
        //遍历直接写出
        for (NullWritable value : values) {
            context.write(key, NullWritable.get());
        }
    }
}

```

(3) 自定义一个OutputFormat类

```

import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/16:12
 * @Description:
 * 自定义的OutputFormat需要继承Hadoop提供的OutputFormat
 */
public class LogOutputFormat extends FileOutputFormat<Text, NullWritable>{

    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext job) throws IOException, InterruptedException {
        LogRecordWriter logRecordWriter = new LogRecordWriter(job);
        return logRecordWriter;
    }
}

```

(4) 编写LogRecordWriter类

```
package com.xulan.mr.outputFormat;
```

```
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: XulAan
 * @Date: 2022/03/22/16:16
 * @Description:
 */
public class LogRecordWriter extends RecordWriter<Text, NullWritable> {

    //定义输出路径
    private String xulanPath = "E:\\\\learning\\\\04_java\\\\02_大数据资料\\\\00_hadoop\\\\out\\\\xulan.txt";
    private String otherPath = "E:\\\\learning\\\\04_java\\\\02_大数据资料\\\\00_hadoop\\\\out\\\\other.txt";
    private FileSystem fs;
    private FSDataOutputStream xulanOutputStream;
    private FSDataOutputStream otherOutputStream;

    public LogRecordWriter(TaskAttemptContext job) throws IOException {
        //获取Hadoop的文件系统对象
        fs = FileSystem.get(job.getConfiguration());
        //获取输出流
        xulanOutputStream = fs.create(new Path(xulanPath));
        //获取输出流
        otherOutputStream= fs.create(new Path(otherPath));
    }

    /**
     * 实现数据写出的逻辑
     * @param key
     * @param value
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    public void write(Text key, NullWritable value) throws IOException,
    InterruptedException {
        //获取当前输入数据
        String logData = key.toString();
        if (logData.contains("xulan")){
            xulanOutputStream.writeBytes(logData+"\n");
        } else {
    }
```

```

        otherOutputStream.writeBytes(logData + "\n");
    }

}

/**
 * 关闭资源
 * @param context
 * @throws IOException
 * @throws InterruptedException
 */
@Override
public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
    IOUtils.closeStream(xulanOutputStream);
    IOUtils.closeStream(otherOutputStream);
}
}

```

(5) 编写LogDriver类

```

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/16:02
 * @Description:
 */
public class LogDriver {

    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(LogDriver.class);
        job.setMapperClass(LogMapper.class);
        job.setReducerClass(LogReducer.class);

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

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

        FileInputFormat.setInputPaths(job, new Path("E:\\\\learning\\\\04_java\\\\02_大
数据资料\\\\00_hadoop\\\\资料\\\\07_测试数据\\\\log"));
    }
}

```

```

FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据\\00_hadoop\\out\\data5"));

//指定OutputFormat类
job.setOutputFormatClass(LogOutputFormat.class);

job.waitForCompletion(true);

}
}

```

3.4 Join多种应用

3.4.1 Reduce Join工作原理

- Map端的主要工作：为来自不同表或文件的key/value对，打标签以区别不同来源的记录。然后用连接字段作为key，其余部分和新加的标志作为value，最后进行输出。
- Reduce端的主要工作：在Reduce端以连接字段作为key的分组已经完成，我们只需要在每一个分组当中将那些来源于不同文件的记录(在Map阶段已经打标志)分开，最后进行合并就ok了。

3.4.2 Reduce Join案例实操

1) 需求分析

通过将关联条件作为Map输出的key，将两表满足Join条件的数据并携带数据所来源的文件信息，发往同一个ReduceTask，在Reduce中进行数据的串联。

1、输入数据

order.txt		
订单id	pid	数量
1001	01	1
1002	02	2
1003	03	3
1004	01	4
1005	02	5
1006	03	6

Pid	产品名称
01	小米
02	华为
03	格力

2、预期输出数据

订单id	产品名称	数量
1001	小米	1
1004	小米	4
1002	华为	2
1005	华为	5
1003	格力	3
1006	格力	6

3、MapTask

1) Map中处理的事情

- (1) 获取输入文件类型
- (2) 获取输入数据
- (3) 不同文件分别处理
- (4) 封装Bean对象输出

01	1001	1	order
02	1002	2	order
03	1003	3	order
01	1004	4	order
02	1005	5	order
03	1006	6	order
01	小米		pd
02	华为		pd
03	格力		pd

2) 默认对产品id排序

01	1001	1	order
01	1004	4	order
01	小米		pd
02	1002	2	order
02	1005	5	order
02	华为		pd
03	1003	3	order
03	1006	6	order
03	格力		pd

4、ReduceTask

1) Reduce方法缓存订单数据集合，和产品表，然后合并

订单id	产品名称	数量
1001	小米	1
1004	小米	4
1002	华为	2
1005	华为	5
1003	格力	3
1006	格力	6

2) 代码实现

```

import org.apache.hadoop.io.Writable;

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

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan

```

```
* @Date: 2022/03/22/17:30
* @Description:
*/
public class OrderProduct implements Writable {

    //order表数据
    private String orderId;
    private String pid;
    private Integer amount;

    //product表数据
    private String pname;

    //区分数据来源
    private String title;

    public String getorderId() {
        return orderId;
    }

    public void setorderId(String orderId) {
        this.orderId = orderId;
    }

    public String getPid() {
        return pid;
    }

    public void setPid(String pid) {
        this.pid = pid;
    }

    public Integer getAmount() {
        return amount;
    }

    public void setAmount(Integer amount) {
        this.amount = amount;
    }

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
```

```

        return orderId + "\t" + pname + "\t" + amount;
    }

    //序列化
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(orderId);
        out.writeUTF(pid);
        out.writeInt(amount);
        out.writeUTF(pname);
        out.writeUTF(title);
    }

    //反序列化
    @Override
    public void readFields(DataInput in) throws IOException {
        orderId = in.readUTF();
        pid = in.readUTF();
        amount = in.readInt();
        pname = in.readUTF();
        title = in.readUTF();
    }
}

```

(2) 编写ReduceJoinMapper类

```

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/17:29
 * @Description:
 */
public class ReduceJoinMapper extends Mapper<LongWritable, Text, Text, OrderProduct> {

    private Text outKey = new Text();
    private OrderProduct outValue = new OrderProduct();
    private FileSplit inputSplit;

    @Override
    protected void setup(Mapper<LongWritable, Text, Text, OrderProduct>.Context context)
            throws IOException, InterruptedException {
        inputSplit = (FileSplit) context.getInputSplit();
    }

    /**
     * 核心业务处理方法
     * 将两个做关联的数据进行搜集
     * @param key
     */

```

```

    * @param value
    * @param context
    * @throws IOException
    * @throws InterruptedException
    */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, OrderProduct>.Context context) throws IOException, InterruptedException {
        //获取当前行数据
        String line = value.toString();
        String[] datas = line.split("\t");
        //将当前数据封装到OrderProduct
        if (inputSplit.getPath().getName().contains("order")){
            // 当前数据来源于order.txt文件 1001 01 1
            // 封装输出数据的key
            outKey.set(datas[1]);
            // 封装输出数据的value
            outValue.setOrderId(datas[0]);
            outValue.setPid(datas[1]);
            outValue.setAmount(Integer.parseInt(datas[2]));
            outValue.setPname("");
            outValue.setTitle("order");
        } else {
            // 当前数据来源于pd.txt文件 1001 01 1
            // 封装输出数据的key
            outKey.set(datas[0]);
            // 封装输出数据的value
            outValue.setOrderId("");
            outValue.setPid(datas[0]);
            outValue.setAmount(0);
            outValue.setPname(datas[1]);
            outValue.setTitle("product");
        }
        //将数据写出
        context.write(outKey,outValue);
    }
}

```

(3) 编写ReduceJoinReducer类

```

import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.List;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/17:29
 * @Description:

```

```
/*
public class ReduceJoinReducer extends Reducer<Text, OrderProduct, OrderProduct,
NullWritable> {

    private List<OrderProduct> orderList= new ArrayList<OrderProduct>();
    private OrderProduct product = new OrderProduct();
    /**
     * 核心处理方法
     *      接收Map端整合好的数据进行Join的操作
     * @param key
     * @param values
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void reduce(Text key, Iterable<OrderProduct> values, Reducer<Text,
    OrderProduct, OrderProduct, NullWritable>.Context context) throws IOException,
    InterruptedException {
        //遍历当前相同可以的一组values
        for (OrderProduct orderProduct : values) {
            //判断当前数据来源
            if (orderProduct.getTitle().equals("order")){
                try {
                    //当前数据来源order.txt文件，将当前数据管理到一个集合当中
                    OrderProduct thisOrderProduct = new OrderProduct();
                    //将当前传入orderProduct复制到thisOrderProduct
                    BeanUtils.copyProperties(thisOrderProduct,orderProduct);
                    orderList.add(thisOrderProduct);
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                }
            } else {
                //当前数据来源product.txt
                try {
                    //将当前传入orderProduct复制到product
                    BeanUtils.copyProperties(product,orderProduct);
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                }
            }
        }

        //进行join操作
        for (OrderProduct orderProductForLast : orderList) {
            orderProductForLast.setPname(product.getPname());
            context.write(orderProductForLast,NullWritable.get());
        }

        //清空orderList
        orderList.clear();
    }
}
```

(4) 编写ReduceJoinDriver类

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/22/17:29
 * @Description:
 */
public class ReduceJoinDriver {
    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(ReduceJoinDriver.class);
        job.setMapperClass(ReduceJoinMapper.class);
        job.setReducerClass(ReduceJoinReducer.class);

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

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

        FileInputFormat.setInputPaths(job, new Path("E:\\\\learning\\\\04_java\\\\02_大数据资料\\\\00_hadoop\\\\资料\\\\07_测试数据\\\\reducejoin"));
        FileOutputFormat.setOutputPath(job, new Path("E:\\\\learning\\\\04_java\\\\02_大数据资料\\\\00_hadoop\\\\out\\\\data4"));

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

3.3.3 Map Join

- MapJoin概念

考虑MR整体的执行效率，且业务场景是一个大文件和一个小文件进行关联操作，可以使用MapJoin来实现。另外MapJoin也是解决ReduceJoin数据倾斜问题很有效的办法。

- MapJoin的思想：

- ①：分析文件之间的关系，然后定位关联字段
- ②：将小文件的数据映射到内存中的一个容器维护起来。
- ③：当MapTask处理大文件的数据时，每读取一行数据，就根据当前行中的关联字段到内存的容器里获取对象的信息。
- ④：封装结果将其输出

1) 使用场景

Map Join适用于一张表十分小、一张表很大的场景。

2) 优点

思考：在Reduce端处理过多的表，非常容易产生数据倾斜。怎么办？

在Map端缓存多张表，提前处理业务逻辑，这样增加Map端业务，减少Reduce端数据的压力，尽可能的减少数据倾斜。

3) 具体办法：采用DistributedCache

- (1) 在Mapper的setup阶段，将文件读取到缓存集合中。
- (2) 在Driver驱动类中加载缓存。

```
//缓存普通文件到Task运行节点。  
job.addCacheFile(new URI("file:///e:/cache/pd.txt"));  
//如果是集群运行，需要设置HDFS路径  
job.addCacheFile(new URI("hdfs://hadoop102:9820/cache/pd.txt"));
```

3.3.4 Map Join案例实操

1) 需求分析

MapJoin适用于关联表中有小表的情形。

1) DistributedCacheDriver 缓存文件

```
// 1 加载缓存数据  
job.addCacheFile(new  
URI("file:///e:/cache/pd.txt"));  
  
//2 Map 端 join 的逻辑不需要  
Reduce阶段，设置ReduceTask数  
量为0  
job.setNumReduceTasks(0);
```

2) 读取缓存的文件数据

// setup()方法中	map方法中
// 1 获取缓存的文件	// 1 获取一行
// 2 循环读取缓存文件一行	// 2 截取
// 3 切割	// 3 获取订单id
// 4 缓存数据到集合	// 4 获取商品名称
// 5 关流	// 5 拼接
	// 6 写出

2) 代码实现

(1) 先在MapJoinDriver驱动类中添加缓存文件

```
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.NullWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
  
import java.io.IOException;  
import java.net.URI;  
  
/**
```

```

* Created with IntelliJ IDEA.
*
* @Author: Xu1Aan
* @Date: 2022/03/24/10:26
* @Description:
*/
public class MapJoinDriver {
    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setMapperClass(MapJoinMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

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

        //设置缓存文件路径
        job.addCacheFile(URI.create("file:///E:/learning/04_java/02_大数据资料/00_hadoop/资料/07_测试数据/cachefile/pd.txt"));

        job.setNumReduceTasks(0);

        FileInputFormat.setInputPaths(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\资料\\07_测试数据\\mapjoin"));
        FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\out\\data7"));

        job.waitForCompletion(true);

    }
}

```

(2) 在MapJoinMapper类中的setup方法中读取缓存文件

```

import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.HashMap;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/24/10:26
 * @Description:
 */

```

```

* 1.处理缓存文件: 将job中设置的缓存路径获取到
* 2.根据缓存路径再结合输入流把pd.txt内容写入内存的容器中
*/
public class MapJoinMapper extends Mapper<LongWritable, Text, Text,
NullWritable>{

    private HashMap<String, String> productMap = new HashMap<String, String>();

    private Text outKey = new Text();

    /**
     * 处理缓存文件
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context
context) throws IOException, InterruptedException {
        //将job中设置的缓存路径获取到
        URI[] cacheFiles = context.getCacheFiles();
        URI cacheFile = cacheFiles[0];
        //准备输入流对象
        FileSystem fileSystem = FileSystem.get(context.getConfiguration());
        FSDataInputStream productIn = fileSystem.open(new Path(cacheFile));
        //通过流对象将数据读入, 保存到内存的Map中
        BufferedReader reader = new BufferedReader(new
InputStreamReader(productIn, "UTF-8"));
        //按行读取
        String line;
        while((line= reader.readLine())!=null){
            //保存在Map中
            String[] datas = line.split("\t");
            productMap.put(datas[0],datas[1]);
        }

        //关闭资源
        IOUtils.closeStream(reader);
    }

    /**
     * 处理mapJoin逻辑
     * @param key
     * @param value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, NullWritable>.Context context) throws IOException, InterruptedException {
        //获取当前行数据
        String lineData = value.toString();
        //切割
        String[] orderData = lineData.split("\t");
        //进行数据关联获取productName
        String productName = productMap.get(orderData[1]);
        //封装结果
    }
}

```

```
        String result = orderData[0] + "\t" + productName +"\t" + orderData[2];
        outKey.set(result);
        //将结果写出
        context.write(outKey,NullWritable.get());
    }
}
```

3.5 计数器与数据清洗(ETL)

3.5.1 计数器的应用

Hadoop为每个作业维护若干内置计数器，以描述多项指标。例如，某些计数器记录已处理的字节数和记录数，使用户可监控已处理的输入数据量和已产生的输出数据量。

1. 计数器API

(1) 采用枚举的方式统计计数

```
enum MyCounter{MALFORORMED,NORMAL}
//对枚举定义的自定义计数器加1
context.getCounter(MyCounter.MALFORORMED).increment(1);
```

(2) 采用计数器组、计数器名称的方式统计

```
context.getCounter("counterGroup", "counter").increment(1);
```

组名和计数器名称随便起，但最好有意义。

(3) 计数结果在程序运行后的控制台上查看。

2. 计数器案例实操

详见数据清洗案例。

3.5.2 数据清洗(ETL)

在运行核心业务MapReduce程序之前，往往要先对数据进行清洗，清理掉不符合用户要求的数据。清理的过程往往只需要运行Mapper程序，不需要运行Reduce程序。

1) 需求分析

需要在Map阶段对输入的数据根据规则进行过滤清洗。去除日志中字段个数小于等于11的日志行内容。

2) 代码实现

(1) 编写Driver驱动类

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 */
```

```

/*
 * @Author: Xu1Aan
 * @Date: 2022/03/24/15:23
 * @Description:
 */
public class EtlDriver {
    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(EtlDriver.class);
        job.setMapperClass(EtlMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(NullWritable.class);

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

        //设置reduce的数量为0
        job.setNumReduceTasks(0);

        FileInputFormat.setInputPaths(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\资料\\07_测试数据\\ETL"));
        FileOutputFormat.setOutputPath(job, new Path("E:\\learning\\04_java\\02_大数据资料\\00_hadoop\\out\\data1"));

        job.waitForCompletion(true);
    }
}

```

(2) 编写Mapper

```

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

import java.io.IOException;

/**
 * Created with IntelliJ IDEA.
 *
 * @Author: Xu1Aan
 * @Date: 2022/03/24/15:23
 * @Description:
 */
public class EtlMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

    private Text outkey = new Text();

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, NullWritable>.Context context) throws IOException, InterruptedException {
        //获取当前行数据
        String line = value.toString();
        //切割
        String[] datas = line.split(" ");

```

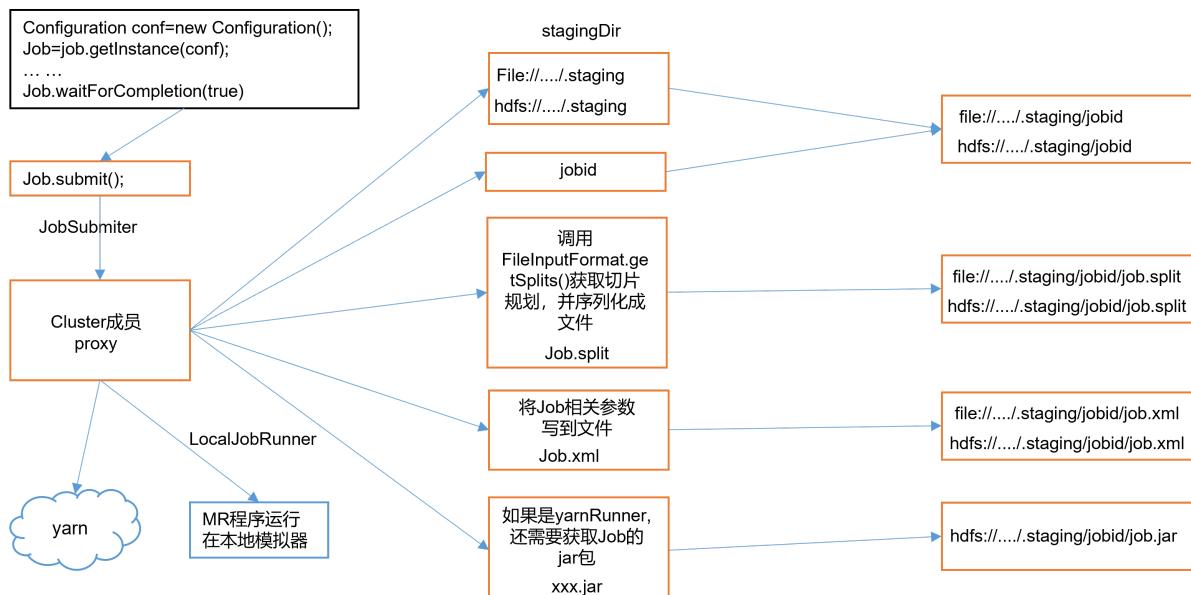
```

//遍历集合 将字段长度 小于等于11的过滤
for (String data : datas) {
    if(data.length()>11){
        outkey.set(data);
        context.write(outkey,NullWritable.get());
    }else
        return;
}

}

```

3.6 Job提交流程源码



1. 定位

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

2. 跟到 waitForCompletion() 中

```

public boolean waitForCompletion(boolean verbose
) throws IOException, InterruptedException,
// 判断当前Job的状态是否为定义阶段
ClassNotFoundException {
    if (state == JobState.DEFINE) {
        // 提交方法
        submit();
    }
    if (verbose) {
        monitorAndPrintJob();
    } else {
        // get the completion poll interval from the client.
        int completionPollIntervalMillis =
        Job.getCompletionPollInterval(cluster.getConf());
        while (!isComplete()) {
            try {
                Thread.sleep(completionPollIntervalMillis);
            } catch (InterruptedException ie) {
            }
        }
    }
}

```

```
        }
    }
    return isSuccessful();
}
```

3. 进入submit() 方法

```
public void submit() throws IOException, InterruptedException,
ClassNotFoundException {
    // 确认当前Job的状态
    ensureState(JobState.DEFINE);
    // 新老API的兼容
    setUseNewAPI();
    // 连接集群（如果是本地模式结果就是LocalRunner，如果Yarn集群结果就是YARNRunner）
    connect();
    // 开始提交Job
    final JobSubmitter submitter =
        getJobSubmitter(cluster.getFileSystem(), cluster.getClient());
    status = ugi.doAs(new PrivilegedExceptionAction<JobStatus>() {
        public JobStatus run() throws IOException, InterruptedException,
        ClassNotFoundException {
            // 提交Job
            return submitter.submitJobInternal(job.this, cluster);
        }
    });
    state = JobState.RUNNING;
    LOG.info("The url to track the job: " + getTrackingURL());
}
```

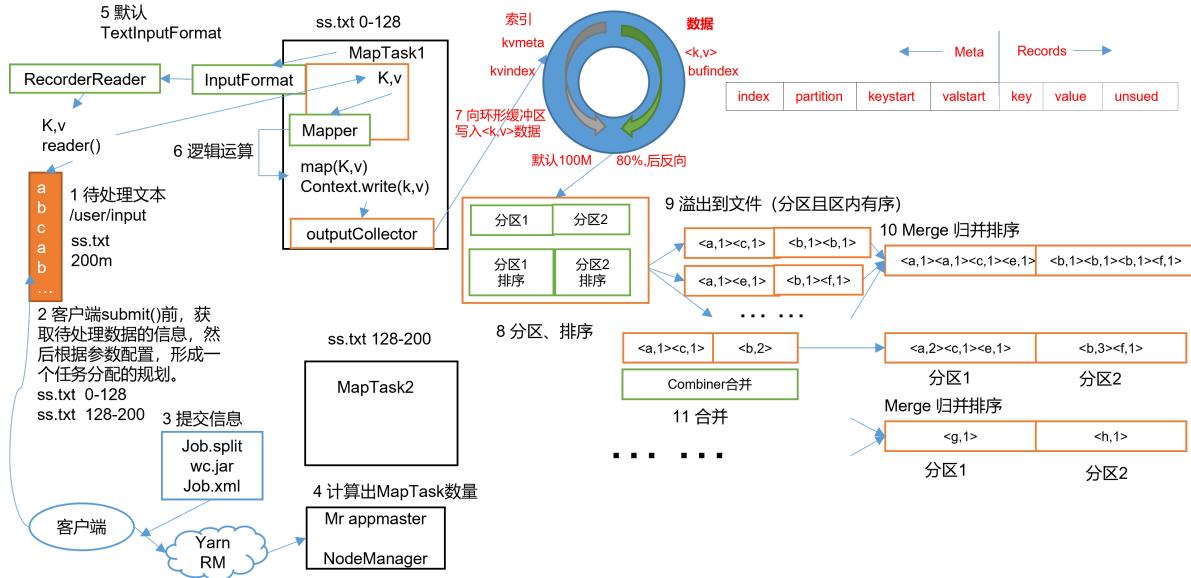
4. 进入submitJobInternal()

```
/**
 * Internal method for submitting jobs to the system.
 *
 * The job submission process involves:
 1、检测输入输出路径的合法性
 *     Checking the input and output specifications of the job.
 2、给当前Job计算切片信息
 *     Computing the {@link InputSplit}s for the job.
 3、添加分布式缓存文件
 *     Setup the requisite accounting information for the
 *     {@link DistributedCache} of the job, if necessary.
 4、将必要的内容都拷贝到 job执行的临时目录（jar包、切片信息、配置文件）
 *     Copying the job's jar and configuration to the map-reduce system
 *     directory on the distributed file-system.
 5、提交Job
 *     Submitting the job to the <code>JobTracker</code> and optionally
 *     monitoring it's status.
 */
```

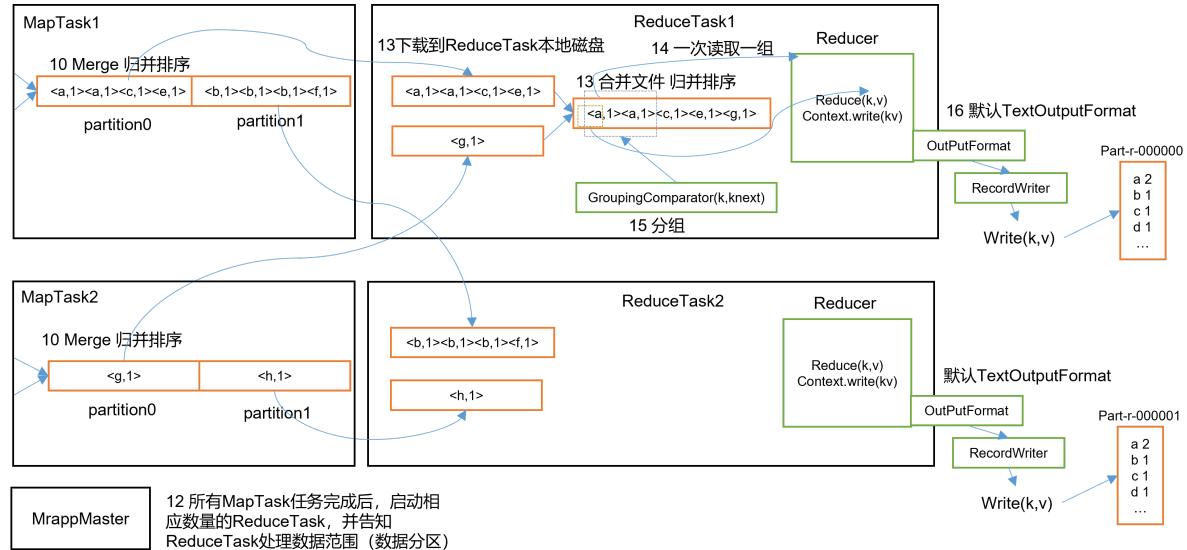
3.7 MapReduce总结

3.7.1 MapReduce工作机制

MapReduce工作机制一



MapReduce工作机制二

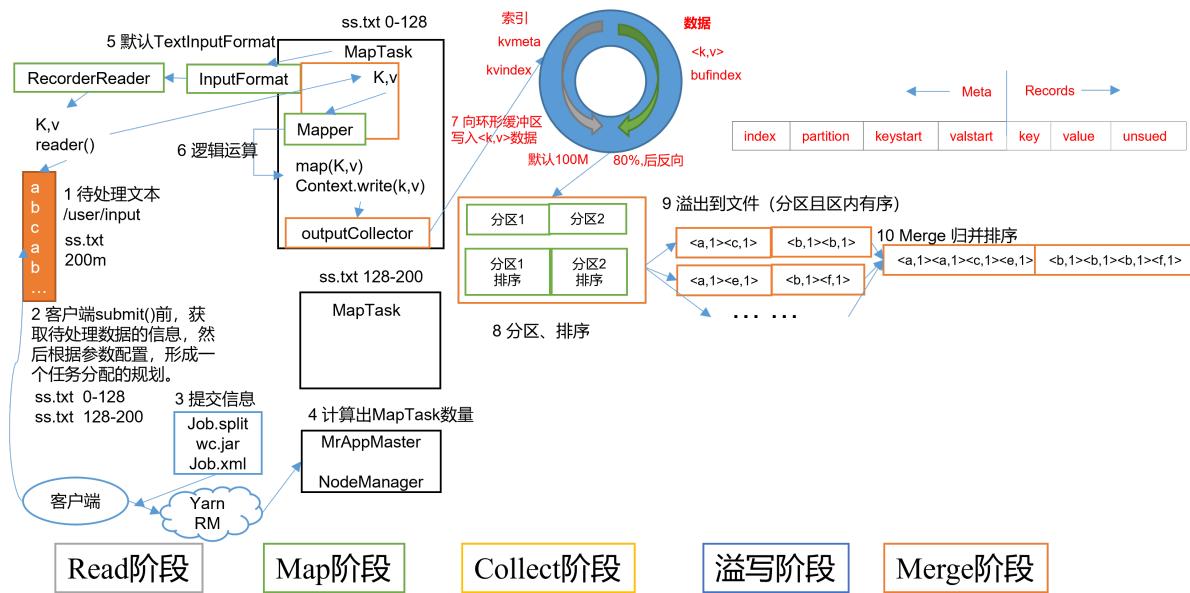


上面的流程是整个 MapReduce 最全工作流程，但是 Shuffle 过程只是从第 7 步开始到第 16 步结束，具体 Shuffle 过程详解，如下：

- (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 会抓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）

3.7.2 MapTask工作机制



(1) Read阶段：MapTask通过InputFormat获得的RecordReader，从输入InputSplit中解析出一个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、压缩等操作。

溢写阶段详情：

步骤1：利用快速排序算法对缓存区内的数据进行排序，排序方式是，先按照分区编号Partition进行排序，然后按照key进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区所有数据按照key有序。

步骤2：按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件output/spillN.out (N表示当前溢写次数) 中。如果用户设置了Combiner，则写入文件之前，对每个分区中的数据进行一次聚集操作。

步骤3：将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小超过1MB，则将内存索引写到文件output/spillN.out.index中。

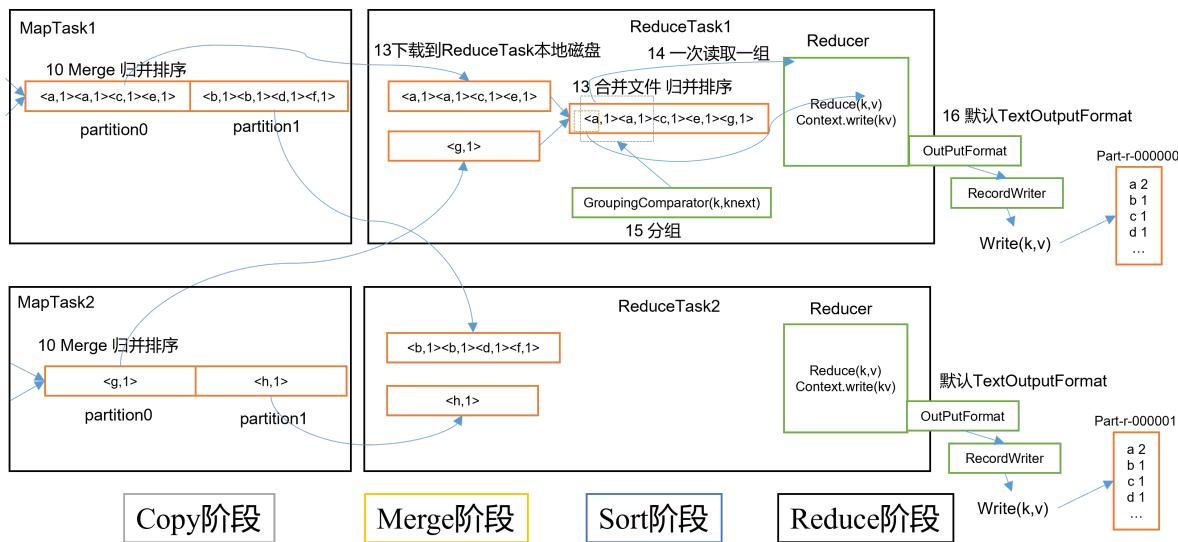
(5) Merge阶段：当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

当所有数据处理完后，MapTask会将所有临时文件合并成一个大文件，并保存到文件output/file.out中，同时生成相应的索引文件output/file.out.index。

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mapreduce.task.io.sort.factor（默认10）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。

让每个MapTask最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

3.7.3 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。

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) 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 改变ReduceTask (数据量为1GB)

MapTask =16										
ReduceTask	1	5	10	15	16	20	25	30	45	60
总时间	892	146	110	92	88	100	128	101	145	104

3) 注意事项

(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。

3.7.4 MapReduce开发总结

1. 输入数据接口：InputFormat

- (1) 默认使用的实现类是：TextInputFormat
- (2) TextInputFormat的功能逻辑是：一次读一行文本，然后将该行的起始偏移量作为key，行内容作为value返回。
- (3) CombineTextInputFormat可以把多个小文件合并成一个切片处理，提高处理效率。

2. 逻辑处理接口：Mapper

用户根据业务需求实现其中三个方法：map() setup() cleanup()

3. Partitioner分区

- (1) 有默认实现 HashPartitioner，逻辑是根据key的哈希值和numReduces来返回一个分区号；
`key.hashCode() & Integer.MAXVALUE % numReduces`
- (2) 如果业务上有特别的需求，可以自定义分区。

4. Comparable排序

- (1) 当我们用自定义的对象作为key来输出时，就必须要实现WritableComparable接口，重写其中的compareTo()方法。
- (2) 部分排序：对最终输出的每一个文件进行内部排序。
- (3) 全排序：对所有数据进行排序，通常只有一个Reduce。
- (4) 二次排序：排序的条件有两个。

5. Combiner合并

Combiner合并可以提高程序执行效率，减少IO传输。但是使用时必须不能影响原有的业务处理结果。

6. 逻辑处理接口：Reducer

用户根据业务需求实现其中三个方法：reduce() setup() cleanup()

7. 输出数据接口：OutputFormat

- (1) 默认实现类是TextOutputFormat，功能逻辑是：将每一个KV对，向目标文本文件输出一行。
- (2) 将SequenceFileOutputFormat输出作为后续 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。
- (3) 用户还可以自定义OutputFormat。

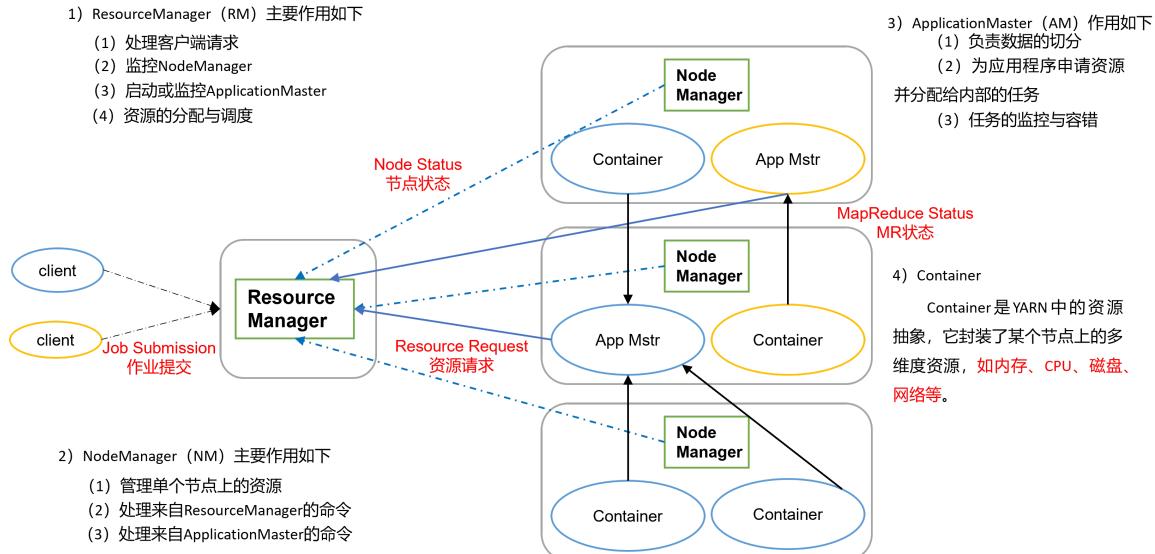
4 Yarn资源调度器

Yarn是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce等运算程序则相当于运行于操作系统之上的应用程序。

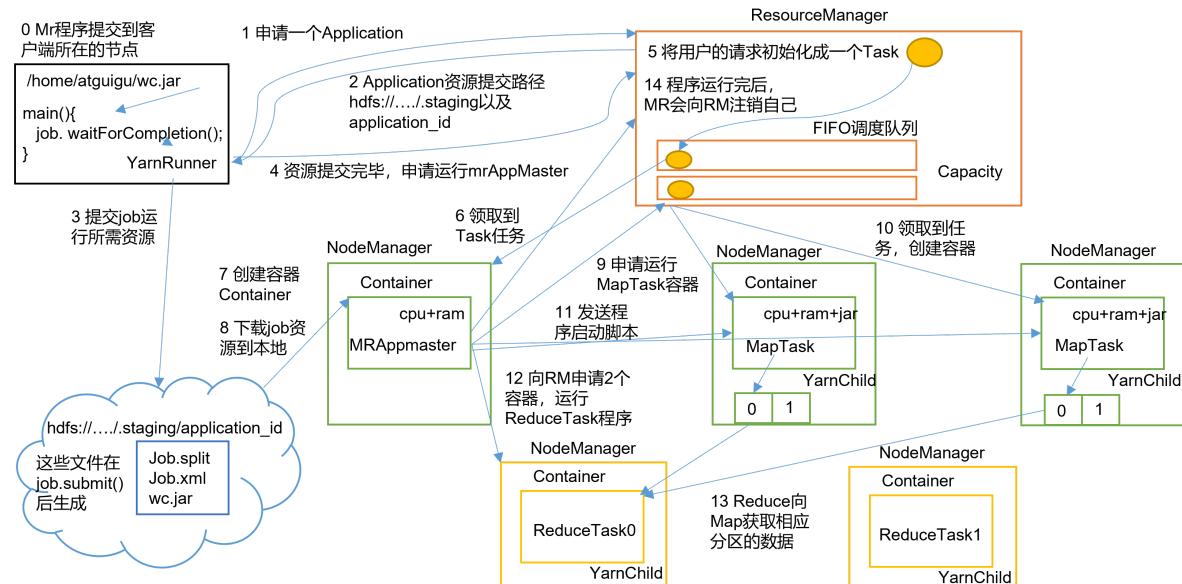
4.1 Yarn工作流程

4.1.1 Yarn基本架构

YARN主要由ResourceManager、NodeManager、ApplicationMaster和Container等组件构成。



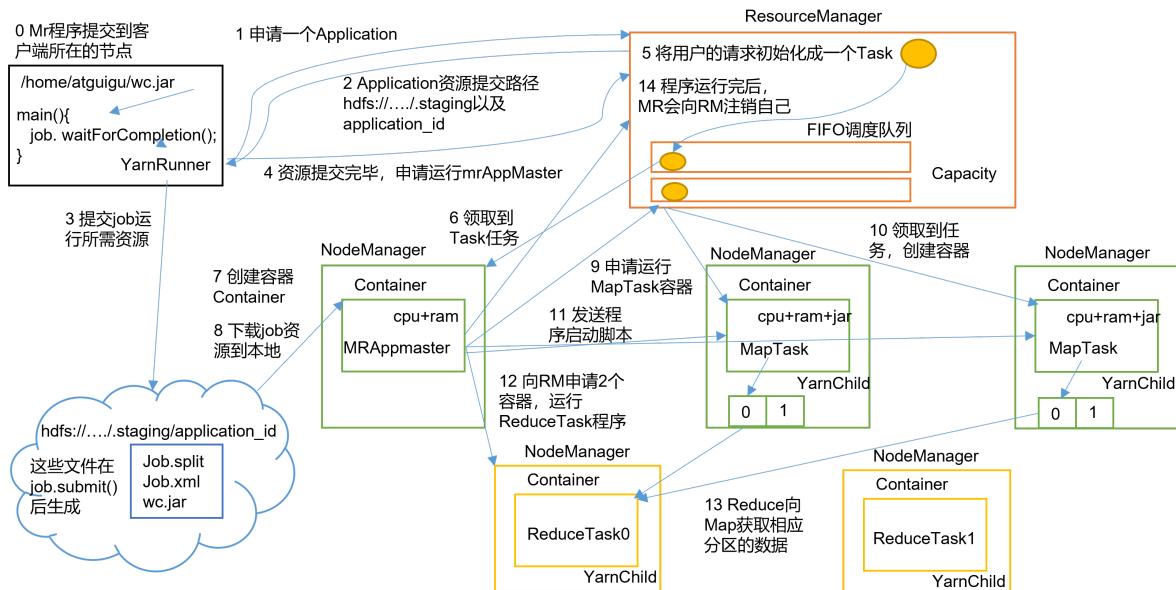
4.1.2 Yarn工作机制



- (1) MR程序提交到客户端所在的节点。
- (2) YarnRunner向ResourceManager申请一个Application。
- (3) RM将该应用程序的资源路径返回给YarnRunner。
- (4) 该程序将运行所需资源提交到HDFS上。
- (5) 程序资源提交完毕后，申请运行mrAppMaster。
- (6) RM将用户的请求初始化成一个Task。
- (7) 其中一个NodeManager领取到Task任务。
- (8) 该NodeManager创建容器Container，并产生MRAppmaster。

- (9) Container从HDFS上拷贝资源到本地。
- (10) MRAppmaster向RM申请运行MapTask资源。
- (11) RM将运行MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。
- (12) MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动MapTask，MapTask对数据分区排序。
- (13) MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。
- (14) ReduceTask向MapTask获取相应分区的数据。
- (15) 程序运行完毕后，MR会向RM申请注销自己。

4.1.3 作业提交全过程



作业提交全过程详解

(1) 作业提交

第1步：Client调用job.waitForCompletion方法，向整个集群提交MapReduce作业。

第2步：Client向RM申请一个作业id。

第3步：RM给Client返回该job 资源的提交路径和作业id。

第4步：Client提交jar包、切片信息和配置文件到指定的资源提交路径。

第5步：Client提交完资源后，向RM申请运行MrAppMaster。

(2) 作业初始化

第6步：当RM收到Client的请求后，将该job添加到容量调度器中。

第7步：某一个空闲的NM领取到该Job。

第8步：该NM创建Container，并产生MRAppmaster。

第9步：下载Client提交的资源到本地。

(3) 任务分配

第10步：MrAppMaster向RM申请运行多个MapTask任务资源。

第11步：RM将运行MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。

(4) 任务运行

第12步：MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动MapTask，MapTask对数据分区排序。

第13步：MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。

第14步：ReduceTask向MapTask获取相应分区的数据。

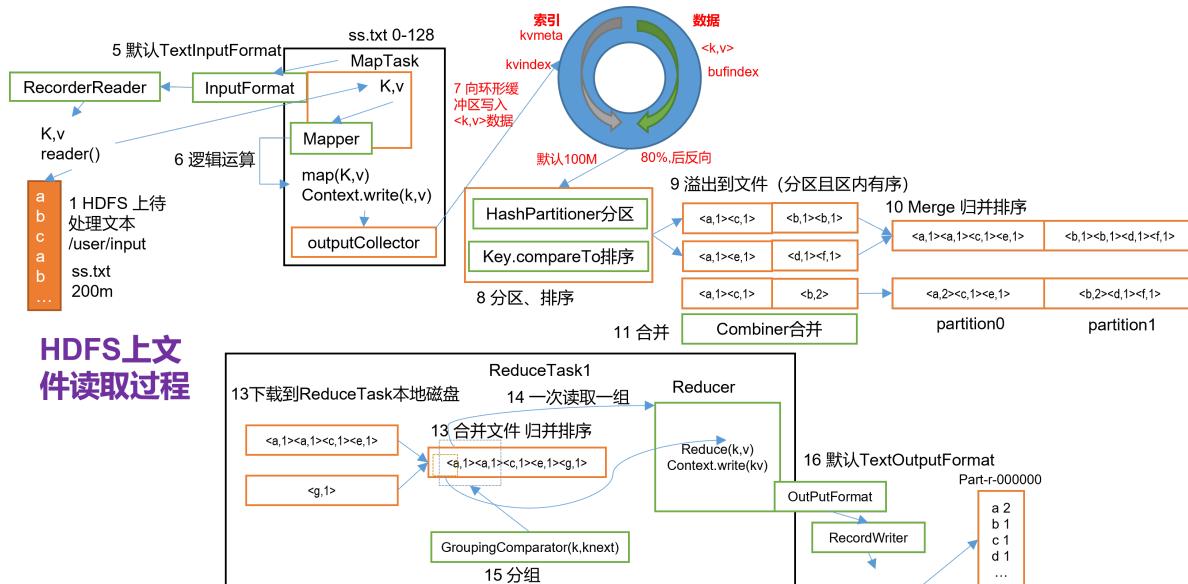
第15步：程序运行完毕后，MR会向RM申请注销自己。

(5) 进度和状态更新

YARN中的任务将其进度和状态(包括counter)返回给应用管理器，客户端每秒(通过mapreduce.client.progressmonitor.pollinterval设置)向应用管理器请求进度更新，展示给用户。

(6) 作业完成

除了向应用管理器请求作业进度外，客户端每5秒都会通过调用waitForCompletion()来检查作业是否完成。时间间隔可以通过mapreduce.client.completion.pollinterval来设置。作业完成之后，应用管理器和Container会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。



4.2 资源调度器

目前，Hadoop作业调度器主要有三种：FIFO、Capacity Scheduler和Fair Scheduler。Hadoop3.1.3默认的资源调度器是Capacity Scheduler。

1. Hadoop1.x 版本 (没有Yarn的存在) --> MR自己负责资源调度-- 出于MR执行效率不高，很大程度就是资源分配环节拖慢的考虑市面上出现 -- mesos 架构 同时为了推广 mesos 发明了Spark，在此影响下，Hadoop团队发明 Yarn 。
2. Hadoop2.x 版本 推出了Yarn结构 (把MR程序运行的时候资源调度工作分离出来)

具体设置详见：yarn-default.xml文件

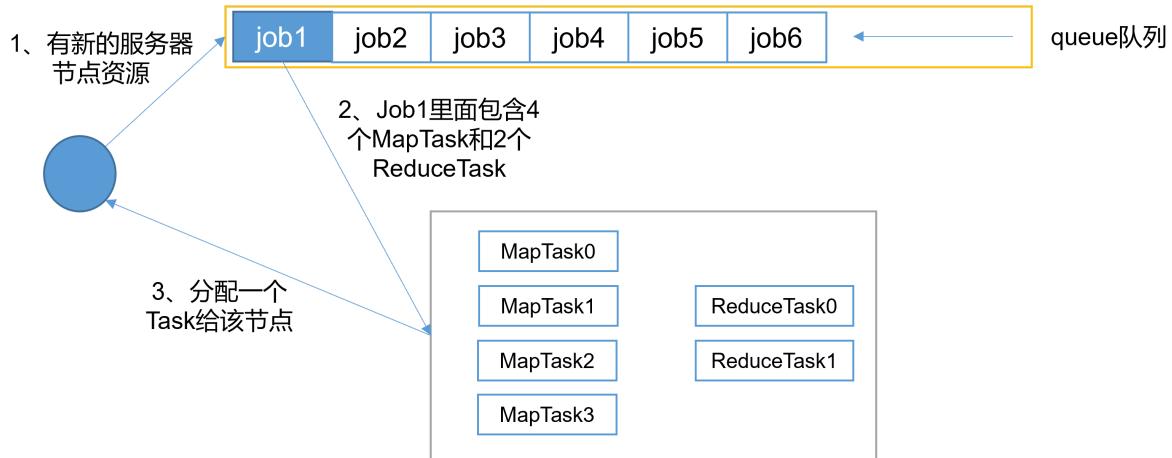
```

<property>
  <description>The class to use as the resource scheduler.</description>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.Capacity
Scheduler</value>
</property>

```

1) 先进先出调度器 (FIFO)

按照到达时间排序，先到先服务



Hadoop最初设计目的是支持大数据批处理作业，如日志挖掘、Web索引等作业，为此，Hadoop仅提供了一个非常简单的调度机制：FIFO，即先来先服务，在该调度机制下，所有作业被统一提交到一个队列中，Hadoop按照提交顺序依次运行这些作业。

但随着Hadoop的普及，单个Hadoop集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求，典型的应用有以下几种：

批处理作业：这种作业往往耗时较长，对时间完成一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。

交互式作业：这种作业期望能及时返回结果，如SQL查询（Hive）等。

生产性作业：这种作业要求有一定量的资源保证，如统计值计算、垃圾数据分析等。

此外，这些应用程序对硬件资源需求量也是不同的，如过滤、统计类作业一般为CPU密集型作业，而数据挖掘、机器学习作业一般为I/O密集型作业。因此，简单的FIFO调度策略不仅不能满足多样化需求，也不能充分利用硬件资源。

2) 容量调度器 (Capacity Scheduler)

按照到达时间排序，先到先服务



- 1、支持多个队列，每个队列可配置一定的资源量，每个队列采用FIFO调度策略。
- 2、为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。
- 3、首先，计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列——最闲的。
- 4、其次，按照作业优先级和提交时间顺序，同时考虑用户资源量限制和内存限制对队列内任务排序。
- 5、三个队列同时按照任务的先后顺序依次执行，比如，job11、job21和job31分别排在队列最前面，先运行，也是并行运行。

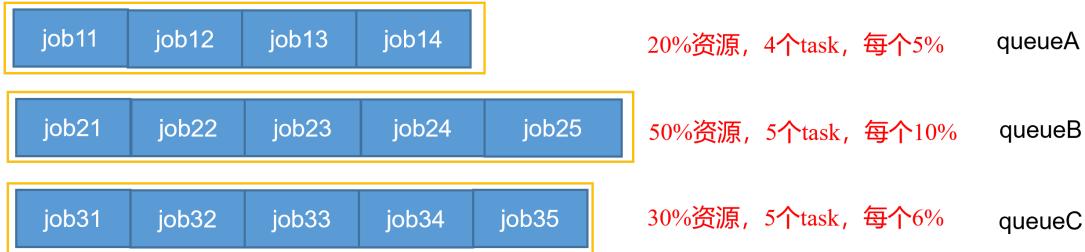
Capacity Scheduler Capacity Scheduler 是Yahoo开发的多用户调度器，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用。而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

总之，Capacity Scheduler 主要有以下几个特点：

- ①容量保证。管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的应用程序共享这些资源。
- ②灵活性，如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列借调的资源会归还给该队列。这种资源灵活分配的方式可明显提高资源利用率。
- ③多重租赁。支持多用户共享集群和多应用程序同时运行。为防止单个应用程序、用户或者队列独占集群中的资源，管理员可为之增加多重约束（比如单个应用程序同时运行的任务数等）。
- ④安全保证。每个队列有严格的ACL列表规定它的访问用户，每个用户可指定哪些用户允许查看自己应用程序的运行状态或者控制应用程序（比如杀死应用程序）。此外，管理员可指定队列管理员和集群系统管理员。
- ⑤动态更新配置文件。管理员可根据需要动态修改各种配置参数，以实现在线集群管理。

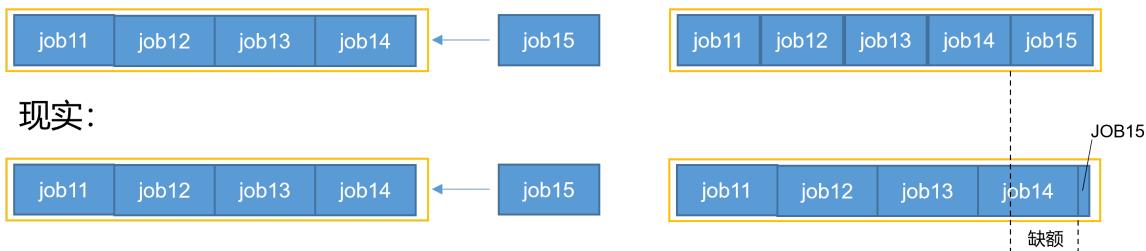
3) 公平调度器 (Fair Scheduler) (了解)

同队列所有任务共享资源，在时间尺度上获得公平的资源



- 支持多队列多作业，每个队列可以单独配置
- 同一队列的作业按照其优先级分享整个队列的资源，并发执行
- 每个作业可以设置最小资源值，调度器会保证作业获得其以上的资源

理想：



- 公平调度器设计目标是：在时间尺度上，所有作业获得公平的资源。某一时刻一个作业应获资源和实际获取资源的差距叫“缺额”
- 调度器会优先为缺额大的作业分配资源

Fair Scheduler Fair Scheduler 是 Facebook 开发的多用户调度器。

公平调度器的目的是让所有的作业随着时间的推移，都能平均地获取等同的共享资源。当有作业提交上来，系统会将空闲的资源分配给新的作业，每个任务大致上会获取平等数量的资源。和传统的调度策略不同的是它会让小的任务在合理的时间完成，同时不会让需要长时间运行的耗费大量资源的任务挨饿！

同 Capacity Scheduler 类似，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用；当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

当然，Fair Scheduler 也存在很多与 Capacity Scheduler 不同之处，这主要体现在以下几个方面：

① 资源公平共享。在每个队列中，Fair Scheduler 可选择按照 FIFO、Fair 或 DRF 策略为应用程序分配资源。其中，

FIFO策略

公平调度器每个队列资源分配策略如果选择 FIFO 的话，就是禁用掉每个队列中的 Task 共享队列资源，此时公平调度器相当于上面讲过的容量调度器。

Fair策略

Fair 策略(默认)是一种基于最大最小公平算法实现的资源多路复用方式，默认情况下，每个队列内部采用该方式分配资源。这意味着，如果一个队列中有两个应用程序同时运行，则每个应用程序可得到1/2的资源；如果三个应用程序同时运行，则每个应用程序可得到1/3的资源。

DRF策略

DRF(Dominant Resource Fairness)，我们之前说的资源，都是单一标准，例如只考虑内存(也是yarn默认的情况)。但是很多时候我们资源有很多种，例如内存，CPU，网络带宽等，这样我们很难衡量两个应用应该分配的资源比例。

那么在YARN中，我们用DRF来决定如何调度：假设集群一共有100 CPU和10T 内存，而应用A需要(2 CPU, 300GB)，应用B需要(6 CPU, 100GB)。则两个应用分别需要A(2%CPU, 3%内存)和B(6%CPU, 1%内存)的资源，这就意味着A是内存主导的，B是CPU主导的，针对这种情况，我们可以选择DRF策略对不同应用进行不同资源 (CPU和内存) 的一个不同比例的限制。

②支持资源抢占。当某个队列中有剩余资源时，调度器会将这些资源共享给其他队列，而当该队列中有新的应用程序提交时，调度器要为它回收资源。为了尽可能降低不必要的计算浪费，调度器采用了先等待再强制回收的策略，即如果等待一段时间后尚有未归还的资源，则会进行资源抢占：从那些超额使用资源的队列中杀死一部分任务，进而释放资源。

yarn.scheduler.fair.preemption=true 通过该配置开启资源抢占。

③提高小应用程序响应时间。由于采用了最大最小公平算法，小作业可以快速获取资源并运行完成

4.3 容量调度器多队列提交案例

1)需求

Yarn默认的容量调度器是一条单队列的调度器，在实际使用中会出现单个任务阻塞整个队列的情况。同时，随着业务的增长，公司需要分业务限制集群使用率。这就需要我们按照业务种类配置多条任务队列。

2)操作

将 /opt/module/hadoop-3.1.3/etc/hadoop 的 capacity-scheduler.xml 下载到本地

```
sz capacity-scheduler.xml
```

配置多队列的容量调度器

默认Yarn的配置下，容量调度器只有一条Default队列。在 capacity-scheduler.xml 中可以配置多条队列，并降低default队列资源占比：

```
<!--
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License. See accompanying LICENSE file.

-->
<configuration>
```

```
<!-- 指定当前yarn集群的队列中可以最大能容纳job的数量-->
<property>
  <name>yarn.scheduler.capacity.maximum-applications</name>
  <value>10000</value>
  <description>
    Maximum number of applications that can be pending and running.
  </description>
</property>

<!-- 指定一个job能够最大占用集群的资源的百分比-->
<property>
  <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
  <value>0.1</value>
  <description>
    Maximum percent of resources in the cluster which can be used to run
    application masters i.e. controls number of concurrent running
    applications.
  </description>
</property>

<!-- 一个默认资源计算器只使用内存-->
<property>
  <name>yarn.scheduler.capacity.resource-calculator</name>

  <value>org.apache.hadoop.yarn.util.resource.DefaultResourceCalculator</value>
  <description>
    The ResourceCalculator implementation to be used to compare
    Resources in the scheduler.
    The default i.e. DefaultResourceCalculator only uses Memory while
    DominantResourceCalculator uses dominant-resource to compare
    multi-dimensional resources such as Memory, CPU etc.
  </description>
</property>

<!-- 配置自定义队列-->
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>default,hello</value>
  <description>
    The queues at the this level (root is the root queue).
  </description>
</property>

<!-- 指定集群中各个队列的额定容量-->
<property>
  <name>yarn.scheduler.capacity.root.default.capacity</name>
  <value>40</value>
  <description>Default queue target capacity.</description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.hello.capacity</name>
  <value>60</value>
  <description>Default queue target capacity.</description>
</property>

<!-- 指定不同用户提交job占用集群资源的百分比-->
<property>
  <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>
```

```
<value>1</value>
<description>
  Default queue user limit a percentage from 0.0 to 1.0.
</description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.hello.user-limit-factor</name>
  <value>1</value>
  <description>
    Default queue user limit a percentage from 0.0 to 1.0.
  </description>
</property>

<!-- 指定集群中每个队列的最大容量-->
<property>
  <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
  <value>60</value>
  <description>
    The maximum capacity of the default queue.
  </description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.hello.maximum-capacity</name>
  <value>80</value>
  <description>
    The maximum capacity of the default queue.
  </description>
</property>

<!-- 指定队列工作状态-->
<property>
  <name>yarn.scheduler.capacity.root.default.state</name>
  <value>RUNNING</value>
  <description>
    The state of the default queue. State can be one of RUNNING or STOPPED.
  </description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.hello.state</name>
  <value>RUNNING</value>
  <description>
    The state of the default queue. State can be one of RUNNING or STOPPED.
  </description>
</property>

<!-- 指定当前队列可以允许哪些用户可以提交job-->
<property>
  <name>yarn.scheduler.capacity.root.default.acl_submit_applications</name>
  <value>*</value>
  <description>
    The ACL of who can submit jobs to the default queue.
  </description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.hello.acl_submit_applications</name>
  <value>*</value>
  <description>
    The ACL of who can submit jobs to the default queue.
  </description>
</property>
```

```

        </description>
    </property>

    <!-- 指定当前队列的拥有权限的用户-->
    <property>
        <name>yarn.scheduler.capacity.root.default.acl_administer_queue</name>
        <value>*</value>
        <description>
            The ACL of who can administer jobs on the default queue.
        </description>
    </property>
    <property>
        <name>yarn.scheduler.capacity.root.hello.acl_administer_queue</name>
        <value>*</value>
        <description>
            The ACL of who can administer jobs on the default queue.
        </description>
    </property>

    <!-- 指定当前队列中可提交优先级job的所属用户-->
    <property>

        <name>yarn.scheduler.capacity.root.default.acl_application_max_priority</name>
        <value>*</value>
        <description>
            The ACL of who can submit applications with configured priority.
            For e.g, [user={name} group={name} max_priority={priority}
default_priority={priority}]
        </description>
    </property>
    <property>
        <name>yarn.scheduler.capacity.root.hello.acl_application_max_priority</name>
        <value>*</value>
        <description>
            The ACL of who can submit applications with configured priority.
            For e.g, [user={name} group={name} max_priority={priority}
default_priority={priority}]
        </description>
    </property>

    <!-- 指定当前队列中job最大存活时间-->
    <property>
        <name>yarn.scheduler.capacity.root.default.maximum-application-lifetime
        </name>
        <value>-1</value>
        <description>
            Maximum lifetime of an application which is submitted to a queue
            in seconds. Any value less than or equal to zero will be considered as
            disabled.
            This will be a hard time limit for all applications in this
            queue. If positive value is configured then any application submitted
            to this queue will be killed after exceeds the configured lifetime.
            User can also specify lifetime per application basis in
            application submission context. But user lifetime will be
            overridden if it exceeds queue maximum lifetime. It is point-in-time
            configuration.
            Note : Configuring too low value will result in killing application
            sooner. This feature is applicable only for leaf queue.
        </description>
    </property>

```

```

        </description>
    </property>
<property>
    <name>yarn.scheduler.capacity.root.hello.maximum-application-lifetime
    </name>
    <value>-1</value>
    <description>
        Maximum lifetime of an application which is submitted to a queue
        in seconds. Any value less than or equal to zero will be considered as
        disabled.
        This will be a hard time limit for all applications in this
        queue. If positive value is configured then any application submitted
        to this queue will be killed after exceeds the configured lifetime.
        User can also specify lifetime per application basis in
        application submission context. But user lifetime will be
        overridden if it exceeds queue maximum lifetime. It is point-in-time
        configuration.
        Note : Configuring too low value will result in killing application
        sooner. This feature is applicable only for leaf queue.
    </description>
</property>

<!-- 指定当前队列中job默认存活时间-->
<property>
    <name>yarn.scheduler.capacity.root.default.default-application-lifetime
    </name>
    <value>-1</value>
    <description>
        Default lifetime of an application which is submitted to a queue
        in seconds. Any value less than or equal to zero will be considered as
        disabled.
        If the user has not submitted application with lifetime value then this
        value will be taken. It is point-in-time configuration.
        Note : Default lifetime can't exceed maximum lifetime. This feature is
        applicable only for leaf queue.
    </description>
</property>
<property>
    <name>yarn.scheduler.capacity.root.hello.default-application-lifetime
    </name>
    <value>-1</value>
    <description>
        Default lifetime of an application which is submitted to a queue
        in seconds. Any value less than or equal to zero will be considered as
        disabled.
        If the user has not submitted application with lifetime value then this
        value will be taken. It is point-in-time configuration.
        Note : Default lifetime can't exceed maximum lifetime. This feature is
        applicable only for leaf queue.
    </description>
</property>

<property>
    <name>yarn.scheduler.capacity.node-locality-delay</name>
    <value>40</value>
    <description>
        Number of missed scheduling opportunities after which the CapacityScheduler
        attempts to schedule rack-local containers.
    </description>

```

When setting this parameter, the size of the cluster should be taken into account.

We use 40 as the default value, which is approximately the number of nodes in one rack.

Note, if this value is -1, the locality constraint in the container request

will be ignored, which disables the delay scheduling.

```
</description>
```

```
</property>
```

```
<property>
```

```
  <name>yarn.scheduler.capacity.rack-locality-additional-delay</name>
```

```
  <value>-1</value>
```

```
  <description>
```

Number of additional missed scheduling opportunities over the node-locality-delay

ones, after which the CapacityScheduler attempts to schedule off-switch containers,

instead of rack-local ones.

Example: with node-locality-delay=40 and rack-locality-delay=20, the scheduler will

attempt rack-local assignments after 40 missed opportunities, and off-switch assignments

after 40+20=60 missed opportunities.

When setting this parameter, the size of the cluster should be taken into account.

We use -1 as the default value, which disables this feature. In this case, the number

of missed opportunities for assigning off-switch containers is calculated based on

the number of containers and unique locations specified in the resource request,

as well as the size of the cluster.

```
</description>
```

```
</property>
```

```
<property>
```

```
  <name>yarn.scheduler.capacity.queue-mappings</name>
```

```
  <value></value>
```

```
  <description>
```

A list of mappings that will be used to assign jobs to queues

The syntax for this list is [u|g]:[name]:[queue_name][,next mapping]*

Typically this list will be used to map users to queues,

for example, u:%user:%user maps all users to queues with the same name as the user.

```
</description>
```

```
</property>
```

```
<property>
```

```
  <name>yarn.scheduler.capacity.queue-mappings-override.enable</name>
```

```
  <value>false</value>
```

```
  <description>
```

If a queue mapping is present, will it override the value specified by the user? This can be used by administrators to place jobs in queues that are different than the one specified by the user.

The default is false.

```
</description>
```

```
</property>
```

```
<property>
  <name>yarn.scheduler.capacity.per-node-heartbeat.maximum-offswitch-
assignments</name>
  <value>1</value>
  <description>
    Controls the number of OFF_SWITCH assignments allowed
    during a node's heartbeat. Increasing this value can improve
    scheduling rate for OFF_SWITCH containers. Lower values reduce
    "clumping" of applications on particular nodes. The default is 1.
    Legal values are 1-MAX_INT. This config is refreshable.
  </description>
</property>

<property>
  <name>yarn.scheduler.capacity.application.fail-fast</name>
  <value>false</value>
  <description>
    Whether RM should fail during recovery if previous applications'
    queue is no longer valid.
  </description>
</property>

</configuration>
```

以wordcount为例，将wordDriver中添加

```
//指定当前job提交队列名称
conf.set("mapreduce.job.queuename", "hello");
```