# MapReduce详解01

|  |  |
| --- | --- |
| MapReduce快速入门 | 如何理解map、reduce计算模型 |
| Mapreudce程序运行演示 |
| MapReduce编程规范及示例编写 |
| MapReduce程序运行模式及debug方法 |
| MapReduce高级特性 | MapReduce程序的核心机制 |
| MapReduce的序列化框架 |
| MapReduce的排序实现 |
| MapReduce的分区机制及自定义分区 |
| MapReduce的数据压缩 |
| MapReduce与yarn的结合 |
| MapReduce编程案例 |
|  | MapReduce 参数优化 |

目标：

* 掌握MapReduce分布式运算框架的**编程思想**
* 掌握MapReduce常用算法的**编程套路**
* 掌握MapReduce分布式运算框架的**运行机制**，具备一定自定义开发的能力

# 1. MapReduce原理篇(1)

MapReduce是一个分布式运算程序运行的容器**框架**，是用户开发“基于hadoop的数据分析应用”的核心框架；

MapReduce**核心功能**是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在一个hadoop集群上；

## 1.1 为什么要MapReduce

(1)海量数据在单机上处理因为硬件资源限制，无法胜任

(2)一旦将单机版程序扩展到集群来分布式运行，将极大增加程序的复杂度和开发难度

(3)引入MapReduce框架后，开发人员可以将绝大部分工作集中在业务逻辑的开发上，而将分布式计算中的复杂性交由框架来处理

设想一个海量数据场景下的wordcount需求：

|  |
| --- |
| 单机版：内存受限，磁盘受限，运算能力受限  分布式：   1. 文件分布式存储(HDFS) 2. 运算逻辑需要至少分成2个阶段(一个阶段独立并发，一个阶段汇聚) 3. 运算程序如何分发 4. 程序如何分配运算任务(切片) 5. 两阶段的程序如何启动？如何协调？ 6. 整个程序运行过程中的监控？容错？重试？ |

可见在程序由单机版扩成分布式时，会引入大量的复杂工作。为了提高开发效率，可以将分布式程序中的公共功能封装成框架，让开发人员可以将精力集中于业务逻辑。

而MapReduce就是这样一个分布式程序的通用框架，其应对以上问题的整体结构如下：

|  |
| --- |
| 1. MRAppMaster(MapReduce application master) 2. MapTask 3. ReduceTask |

## 1.2 MapReduce框架结构及核心运行机制

### 1.2.1 结构

一个完整的MapReduce程序在分布式运行时有三类实例进程：

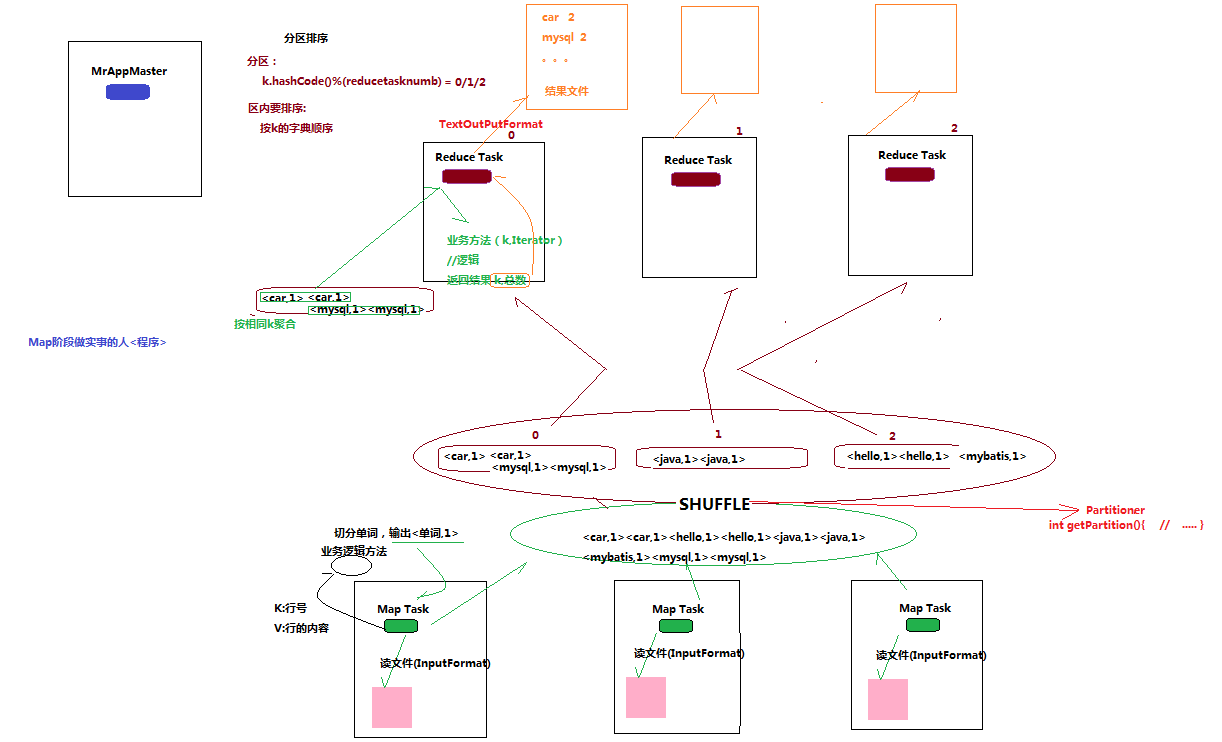
1、MRAppMaster：负责整个程序的过程调度及状态协调

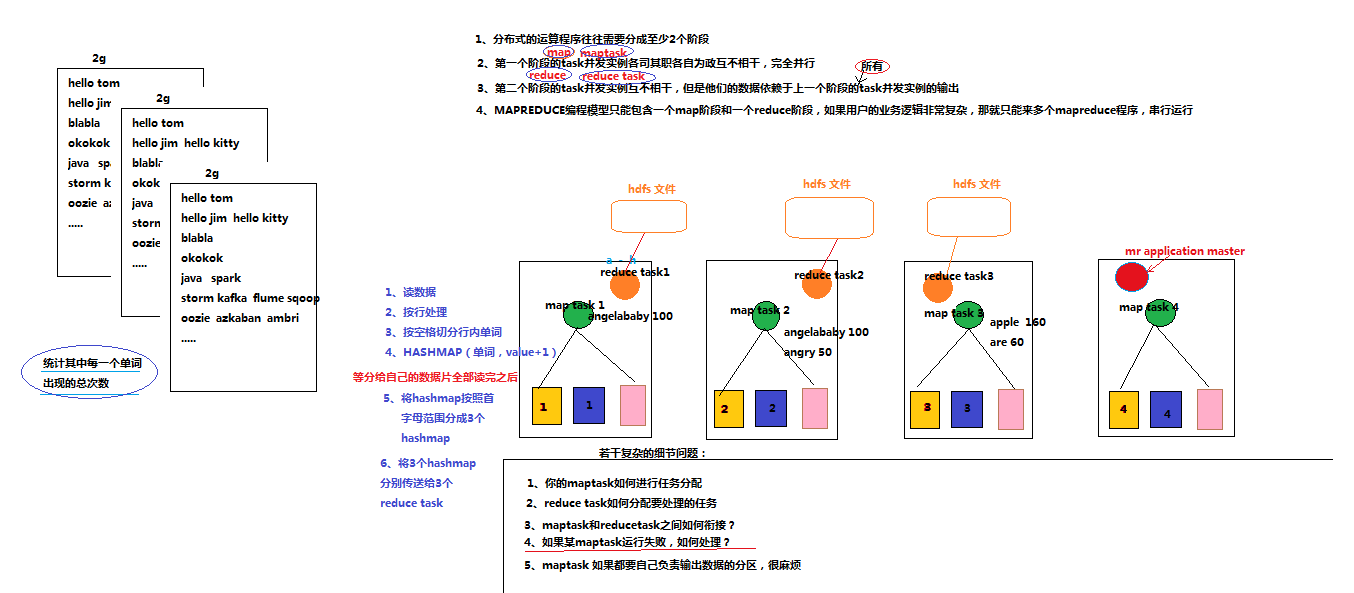
2、MapTask：负责map阶段的整个数据处理流程

3、ReduceTask：负责reduce阶段的整个数据处理流程

### 1.2.2 MR程序运行流程

#### 1.2.2.1 流程示意图





#### 1.2.2.2 流程解析

1. 一个mr程序启动的时候，最先启动的是MRAppMaster，MRAppMaster启动后根据本次job的描述信息，计算出需要的MapTask实例数量，然后向集群申请机器启动相应数量的MapTask进程
2. MapTask进程启动之后，根据给定的数据切片范围进行数据处理，主体流程为：
   1. 利用客户指定的inputformat来获取RecordReader读取数据，形成输入KV对
   2. 将输入KV对传递给客户定义的map()方法，做逻辑运算，并将map()方法输出的KV对收集到缓存
   3. 将缓存中的KV对按照K分区排序后不断溢写到磁盘文件
3. MRAppMaster监控到所有MapTask进程任务完成之后，会根据客户指定的参数启动相应数量的reducetask进程，并告知reducetask进程要处理的数据范围(数据分区)
4. Reducetask进程启动之后，根据MRAppMaster告知的待处理数据所在位置，从若干台MapTask运行所在机器上获取到若干个MapTask输出结果文件，并在本地进行重新归并排序，然后按照相同key的KV为一个组，调用客户定义的reduce()方法进行逻辑运算，并收集运算输出的结果KV，然后调用客户指定的outputformat将结果数据输出到外部存储

## 1.3 MapTask并行度决定机制

MapTask的并行度决定map阶段的任务处理并发度，进而影响到整个job的处理速度

那么，MapTask并行实例是否越多越好呢？其并行度又是如何决定呢？

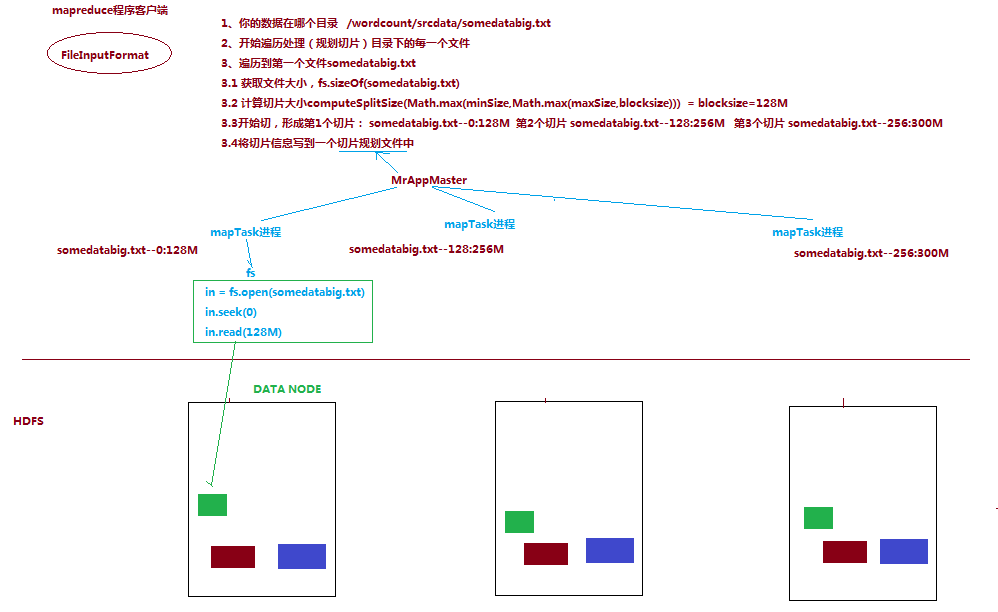
### 1.3.1 MapTask并行度的决定机制

一个job的map阶段并行度由客户端在提交job时决定

而客户端对map阶段并行度的规划的基本逻辑为：

将待处理数据执行逻辑切片(即按照一个特定切片大小，将待处理数据划分成逻辑上的多个split)，然后每一个split分配一个MapTask并行实例处理

这段逻辑及形成的切片规划描述文件，由FileInputFormat实现类的getSplits()方法完成，其过程如下图：



### 1.3.2 FileInputFormat切片机制

#### 1、切片定义在InputFormat类中的getSplit()方法

#### 2、FileInputFormat中默认的切片机制：

* 1. 简单地按照文件的内容长度进行切片
  2. 切片大小，默认等于block大小
  3. 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

比如待处理数据有两个文件：

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

经过FileInputFormat的切片机制运算后，形成的切片信息如下：

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

#### 3、FileInputFormat中切片的大小的参数配置

通过分析源码，在FileInputFormat中，计算切片大小的逻辑：Math.max(minSize, Math.min(maxSize, blockSize)); 切片主要由这几个值来运算决定

|  |
| --- |
| minsize：默认值：1  配置参数： mapreduce.input.fileinputformat.split.minsize |
| maxsize：默认值：Long.MAXValue  配置参数：mapreduce.input.fileinputformat.split.maxsize |
| blocksize |

因此，**默认情况下，切片大小=blocksize**

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

选择并发数的影响因素：

1. 运算节点的硬件配置
2. 运算任务的类型：CPU密集型还是IO密集型
3. 运算任务的数据量

## 1.4 map并行度的经验之谈

如果硬件配置为2\*12core + 64G，恰当的map并行度是大约每个节点20-100个map，**最好每个map的执行时间至少一分钟。**

* 如果job的每个map或者 reduce task的运行时间都只有30-40秒钟，那么就减少该job的map或者reduce数，每一个task(map|reduce)的setup和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个task都非常快就跑完了，就会在task的开始和结束的时候浪费太多的时间。配置task的JVM重用可以改善该问题(**JVM重用技术**是排队按顺序执行)：(mapred.job.reuse.jvm.num.tasks，默认是1，表示一个JVM上最多可以顺序执行的task数目(属于同一个Job)是1。也就是说一个task启一个JVM)
* 如果input的文件非常的大，比如1TB，可以考虑将hdfs上的每个block size设大，比如设成256MB或者512MB

## 1.5 ReduceTask并行度的决定

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);

如果数据分布不均匀，就有可能在reduce阶段产生数据倾斜

注意： 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

尽量不要运行太多的reduce task。对大多数job来说，最好rduce的个数最多和集群中的reduce持平，或者比集群的 reduce slots小。这个对于小集群而言，尤其重要。

## 1.6 MapReduce程序运行演示

Hadoop的发布包中内置了一个hadoop-mapreduce-example-2.4.1.jar，这个jar包中有各种MR示例程序，可以通过以下步骤运行：

启动hdfs，yarn

然后在集群中的任意一台服务器上启动执行程序(比如运行wordcount)：

hadoop jar hadoop-mapreduce-example-2.4.1.jar wordcount /wordcount/data /wordcount/out

# 2. MapReduce实践篇(1)

## 2.1 MapReduce 示例编写及编程规范

### 2.1.1 编程规范

1. **用户编写的程序分成三个部分**：Mapper，Reducer，Driver(提交运行mr程序的客户端)
2. Mapper的输入数据是KV对的形式(KV的类型可自定义)
3. Mapper的输出数据是KV对的形式(KV的类型可自定义)
4. Mapper中的业务逻辑写在map()方法中
5. map()方法(MapTask进程)对每一个<K,V>调用一次
6. Reducer的输入数据类型对应Mapper的输出数据类型，也是KV
7. Reducer的业务逻辑写在reduce()方法中
8. Reducetask进程对每一组相同k的<k,v>组调用一次reduce()方法
9. 用户自定义的Mapper和Reducer都要继承各自的父类
10. 整个程序需要一个Drvier来进行提交，提交的是一个描述了各种必要信息的job对象

### 1.7.2 wordcount示例编写

需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(1)定义一个mapper类

|  |
| --- |
| //首先要定义四个泛型的类型  //keyin: LongWritable valuein: Text  //keyout: Text valueout:IntWritable  public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{  //map方法的生命周期： 框架每传一行数据就被调用一次  //key : 这一行的起始点在文件中的偏移量  //value: 这一行的内容  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  //拿到一行数据转换为string  String line = value.toString();  //将这一行切分出各个单词  String[] words = line.split(" ");  //遍历数组，输出<单词，1>  for(String word:words){  context.write(new Text(word), new IntWritable(1));  }  }  } |

(2)定义一个reducer类

|  |
| --- |
| //生命周期：框架每传递进来一个kv 组，reduce方法被调用一次  @Override  protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {  //定义一个计数器  int count = 0;  //遍历这一组kv的所有v，累加到count中  for(IntWritable value:values){  count += value.get();  }  context.write(key, new IntWritable(count));  }  } |

(3)定义一个主类，用来描述job并提交job

|  |
| --- |
| public class WordCountRunner {  //把业务逻辑相关的信息(哪个是mapper，哪个是reducer，要处理的数据在哪里，输出的结果放哪里……)描述成一个job对象  //把这个描述好的job提交给集群去运行  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job wcjob = Job.getInstance(conf);  //指定我这个job所在的jar包  //wcjob.setJar("/home/hadoop/wordcount.jar");  wcjob.setJarByClass(WordCountRunner.class);    wcjob.setMapperClass(WordCountMapper.class);  wcjob.setReducerClass(WordCountReducer.class);  //设置我们的业务逻辑Mapper类的输出key和value的数据类型  wcjob.setMapOutputKeyClass(Text.class);  wcjob.setMapOutputValueClass(IntWritable.class);  //设置我们的业务逻辑Reducer类的输出key和value的数据类型  wcjob.setOutputKeyClass(Text.class);  wcjob.setOutputValueClass(IntWritable.class);    //指定要处理的数据所在的位置  FileInputFormat.setInputPaths(wcjob, "hdfs://hdp-server01:9000/wordcount/data/big.txt");  //指定处理完成之后的结果所保存的位置  FileOutputFormat.setOutputPath(wcjob, new Path("hdfs://hdp-server01:9000/wordcount/output/"));  //向yarn集群提交这个job  boolean res = wcjob.waitForCompletion(true);  System.exit(res?0:1);  } |

## 2.2 MapReduce程序运行模式

### 2.2.1 本地运行模式

1. MapReduce程序是被提交给LocalJobRunner在本地以单进程的形式运行
2. 而处理的数据及输出结果可以在本地文件系统，也可以在hdfs上
3. 怎样实现本地运行？写一个程序，不要带集群的配置文件(本质是你的mr程序的conf中是否有mapreduce.framework.name=local以及yarn.resourcemanager.hostname参数)
4. 本地模式非常便于进行业务逻辑的debug，只要在eclipse中打断点即可

如果在windows下想运行本地模式来测试程序逻辑，需要在windows中配置环境变量：

％HADOOP\_HOME％ = d:/hadoop-2.6.1

%PATH% = ％HADOOP\_HOME％\bin

并且要将d:/hadoop-2.6.1的lib和bin目录替换成windows平台编译的版本

### 2.2.2 集群运行模式

1. 将MapReduce程序提交给yarn集群resourcemanager，分发到很多的节点上并发执行
2. 处理的数据和输出结果应该位于hdfs文件系统
3. 提交集群的实现步骤：

A、将程序打成JAR包，然后在集群的任意一个节点上用hadoop命令启动

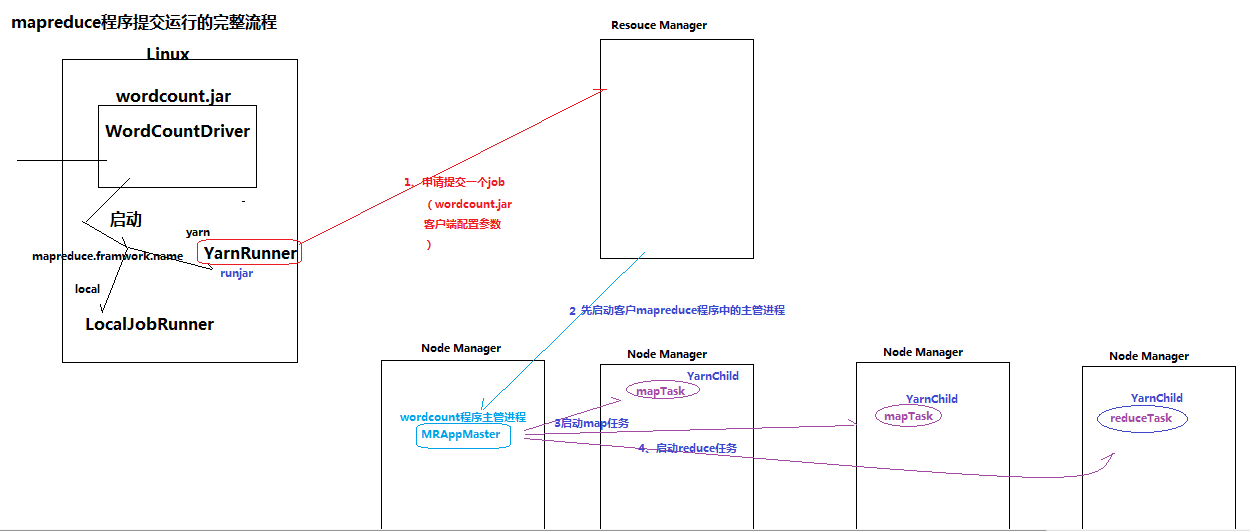
$ hadoop jar wordcount.jar cn.itcast.bigdata.mrsimple.WordCountDriver inputpath outputpath

B、直接在linux的eclipse中运行main方法

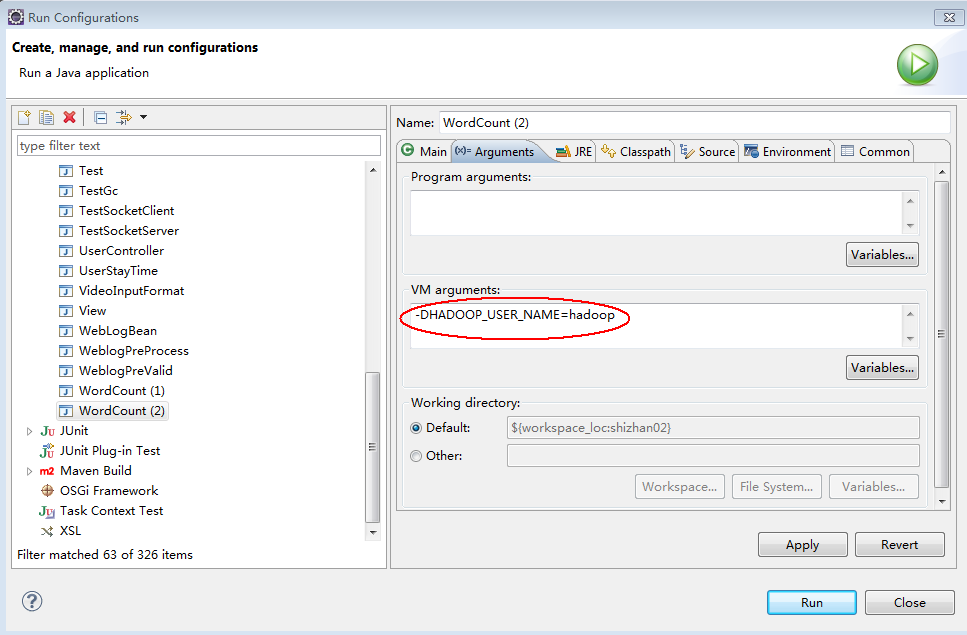
**(项目中要带参数：mapreduce.framework.name=yarn以及yarn的两个基本配置)**

C、如果要在windows的eclipse中提交job给集群，则要修改YarnRunner类

MapReduce程序在集群中运行时的大体流程：



附：在windows平台上访问hadoop时改变自身身份标识的方法之二：



## 3. MapReduce中的Combiner

(Combiner的使用要非常谨慎

因为combiner在MapReduce过程中可能调用也肯能不调用，可能调一次也可能调多次

所以：combiner使用的原则是：有或没有都不能影响业务逻辑)

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

具体实现步骤：

1. 自定义一个combiner继承Reducer，重写reduce方法
2. 在job中设置： job.setCombinerClass(CustomCombiner.class)

(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

# MapReduce原理篇(2)

## 3.1 MapReduce的shuffle机制

## 3.1.1 概述：

* MapReduce中，map阶段处理的数据如何传递给reduce阶段，是MapReduce框架中最关键的一个流程，这个流程就叫shuffle；
* shuffle: 洗牌、发牌——(核心机制：数据分区，排序，缓存)；
* 具体来说：就是将MapTask输出的处理结果数据，分发给reducetask，并在分发的过程中，对数据按key进行了分区和排序；

## 3.1.2 主要流程：

Shuffle缓存流程：



shuffle是MR处理流程中的一个过程，它的每一个处理步骤是分散在各个map task和reduce task节点上完成的，整体来看，分为3个操作：

1. 分区partition
2. Sort根据key排序
3. Combiner进行局部value的合并

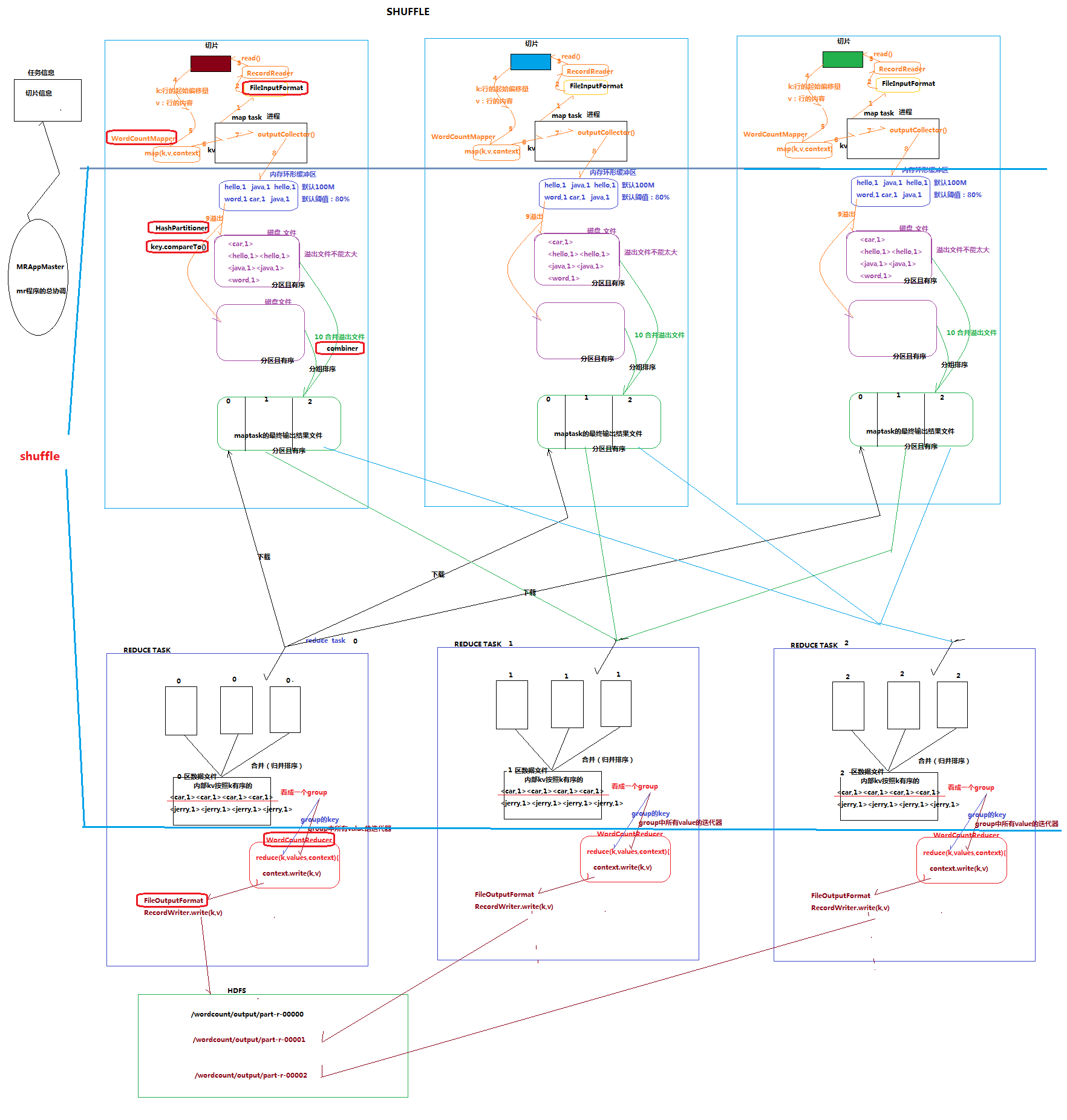
## 3.1.3 详细流程

1. MapTask收集我们的map()方法输出的kv对，放到内存缓冲区中
2. 从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件
3. 多个溢出文件会被合并成大的溢出文件
4. 在溢出过程中，及合并的过程中，都要调用partitoner进行分组和针对key进行排序
5. reducetask根据自己的分区号，去各个MapTask机器上取相应的结果分区数据
6. reducetask会取到同一个分区的来自不同MapTask的结果文件，reducetask会将这些文件再进行合并(归并排序)
7. 合并成大文件后，shuffle的过程也就结束了，后面进入reducetask的逻辑运算过程(从文件中取出一个一个的键值对group，调用用户自定义的reduce()方法)

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

缓冲区的大小可以通过参数调整, 参数：io.sort.mb 默认100M

## 3.1.4 详细流程示意图



## 3.2. MapReduce中的序列化

## 3.2.1 概述

Java的序列化是一个重量级序列化框架(Serializable)，一个对象被序列化后，会附带很多额外的信息(各种校验信息，header，继承体系。。。。)，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)，精简，高效

## 3.2.2 Jdk序列化和MR序列化之间的比较

简单代码验证两种序列化机制的差别：

|  |
| --- |
| public class TestSeri {  public static void main(String[] args) throws Exception {  //定义两个ByteArrayOutputStream，用来接收不同序列化机制的序列化结果  ByteArrayOutputStream ba = new ByteArrayOutputStream();  ByteArrayOutputStream ba2 = new ByteArrayOutputStream();  //定义两个DataOutputStream，用于将普通对象进行jdk标准序列化  DataOutputStream dout = new DataOutputStream(ba);  DataOutputStream dout2 = new DataOutputStream(ba2);  ObjectOutputStream obout = new ObjectOutputStream(dout2);  //定义两个bean，作为序列化的源对象  ItemBeanSer itemBeanSer = new ItemBeanSer(1000L, 89.9f);  ItemBean itemBean = new ItemBean(1000L, 89.9f);  //用于比较String类型和Text类型的序列化差别  Text atext = new Text("a");  // atext.write(dout);  itemBean.write(dout);  byte[] byteArray = ba.toByteArray();  //比较序列化结果  System.out.println(byteArray.length);  for (byte b : byteArray) {  System.out.print(b);  System.out.print(":");  }  System.out.println("-----------------------");  String astr = "a";  // dout2.writeUTF(astr);  obout.writeObject(itemBeanSer);  byte[] byteArray2 = ba2.toByteArray();  System.out.println(byteArray2.length);  for (byte b : byteArray2) {  System.out.print(b);  System.out.print(":");  }  }  } |

## 3.2.3 自定义对象实现MR中的序列化接口

如果需要将自定义的bean放在key中传输，则还需要实现comparable接口，因为MapReduce框中的shuffle过程一定会对key进行排序,此时，自定义的bean实现的接口应该是：

public class FlowBean implements WritableComparable<FlowBean>

需要自己实现的方法是：

|  |
| --- |
| /\*\*  \* 反序列化的方法，反序列化时，从流中读取到的各个字段的顺序应该与序列化时写出去的顺序保持一致  \*/  @Override  public void readFields(DataInput in) throws IOException {  upflow = in.readLong();  dflow = in.readLong();  sumflow = in.readLong();  }  /\*\*  \* 序列化的方法  \*/  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upflow);  out.writeLong(dflow);  //可以考虑不序列化总流量，因为总流量是可以通过上行流量和下行流量计算出来的  out.writeLong(sumflow);  }    @Override  public int compareTo(FlowBean o) {  //实现按照sumflow的大小倒序排序  return sumflow>o.getSumflow()?-1:1;  } |

## 3.3. MapReduce与YARN

## 3.3.1 YARN概述

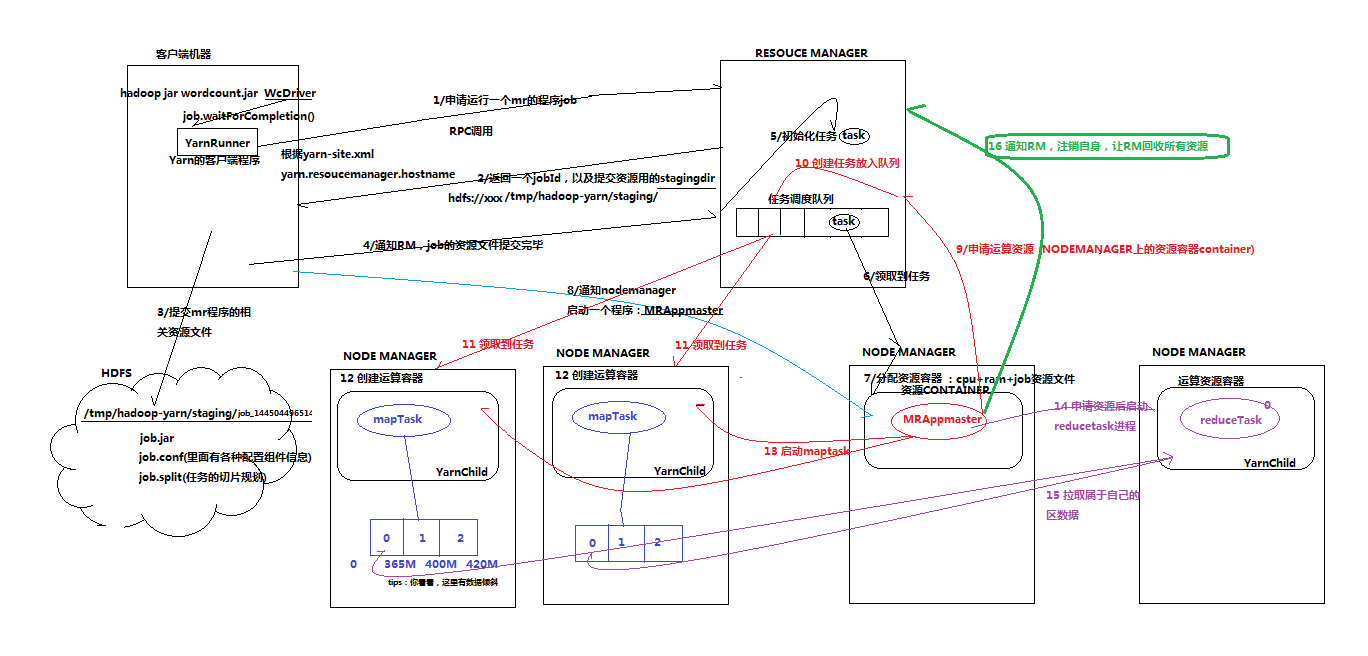
Yarn是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce等运算程序则相当于运行于操作系统之上的应用程序

## 3.3.2 YARN的重要概念

1. Yarn并不清楚用户提交的程序的运行机制
2. yarn只提供运算资源的调度(用户程序向Yarn申请资源，Yarn就负责分配资源)
3. Yarn中的主管角色叫ResourceManager
4. Yarn中具体提供运算资源的角色叫NodeManager
5. Yarn其实就与运行的用户程序完全解耦，就意味着yarn上可以运行各种类型的分布式运算程序(MapReduce只是其中的一种)，比如MapReduce、storm程序，spark程序，tez ……
6. spark、storm等运算框架都可以整合在yarn上运行，只要他们各自的框架中有符合yarn规范的资源请求机制即可
7. Yarn就成为一个通用的资源调度平台，从此，企业中以前存在的各种运算集群都可以整合在一个物理集群上，提高资源利用率，方便数据共享

## 3.3.3 Yarn中运行运算程序的示例

MapReduce程序的调度过程，如下图



# 4. MapReduce实践篇(2)

## 4.1. MapReduce中的排序初步

### 4.1.1 需求

对日志数据中的上下行流量信息汇总，并输出按照总流量倒序排序的结果

数据如下：

|  |
| --- |
| 1363157985066 13726230503 00-FD-07-A4-72-B8:CMCC 120.196.100.82 24 27 2481 24681 200  1363157995052 13826544101 5C-0E-8B-C7-F1-E0:CMCC 120.197.40.4 4 0 264 0 200  1363157991076 13926435656 20-10-7A-28-CC-0A:CMCC 120.196.100.99 2 4 132 1512 200  1363154400022 13926251106 5C-0E-8B-8B-B1-50:CMCC 120.197.40.4 4 0 240 0 200 |

### 4.1.2 分析

基本思路：实现自定义的bean来封装流量信息，并将bean作为map输出的key来传输

MR程序在处理数据的过程中会对数据排序(map输出的kv对传输到reduce之前，会排序)，排序的依据是map输出的key；所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到key中，让key实现接口：WritableComparable；然后重写key的compareTo方法

### 4.1.3 实现

1. 自定义的bean

|  |
| --- |
| public class FlowBean implements WritableComparable<FlowBean>{  long upflow;  long downflow;  long sumflow;    //如果空参构造函数被覆盖，一定要显示定义一下，否则在反序列时会抛异常  public FlowBean(){}    public FlowBean(long upflow, long downflow) {  super();  this.upflow = upflow;  this.downflow = downflow;  this.sumflow = upflow + downflow;  }    public long getSumflow() {  return sumflow;  }  public void setSumflow(long sumflow) {  this.sumflow = sumflow;  }  public long getUpflow() {  return upflow;  }  public void setUpflow(long upflow) {  this.upflow = upflow;  }  public long getDownflow() {  return downflow;  }  public void setDownflow(long downflow) {  this.downflow = downflow;  }  //序列化，将对象的字段信息写入输出流  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upflow);  out.writeLong(downflow);  out.writeLong(sumflow);  }  //反序列化，从输入流中读取各个字段信息  @Override  public void readFields(DataInput in) throws IOException {  upflow = in.readLong();  downflow = in.readLong();  sumflow = in.readLong();    }    @Override  public String toString() {  return upflow + "\t" + downflow + "\t" + sumflow;  }  @Override  public int compareTo(FlowBean o) {  //自定义倒序比较规则  return sumflow > o.getSumflow() ? -1:1;  }  } |

1. mapper 和 reducer

|  |
| --- |
| public class FlowCount {  static class FlowCountMapper extends Mapper<LongWritable, Text, FlowBean,Text > {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = line.split("\t");  try {  String phonenbr = fields[0];  long upflow = Long.parseLong(fields[1]);  long dflow = Long.parseLong(fields[2]);  FlowBean flowBean = new FlowBean(upflow, dflow);  context.write(flowBean,new Text(phonenbr));  } catch (Exception e) {  e.printStackTrace();  }  }  }  static class FlowCountReducer extends Reducer<FlowBean,Text,Text, FlowBean> {  @Override  protected void reduce(FlowBean bean, Iterable<Text> phonenbr, Context context) throws IOException, InterruptedException {  Text phoneNbr = phonenbr.iterator().next();  context.write(phoneNbr, bean);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(FlowCount.class);  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  job.setMapOutputKeyClass(FlowBean.class);  job.setMapOutputValueClass(Text.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // job.setInputFormatClass(TextInputFormat.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  }  } |

## 4.2. MapReduce中的分区Partitioner

### 4.2.1 需求

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

### 4.2.2 分析

MapReduce中会将map输出的kv对，按照相同key分组，然后分发给不同的reducetask

默认的分发规则为：根据key的hashcode%reducetask数来分发

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

自定义一个CustomPartitioner继承抽象类：Partitioner

然后在job对象中，设置自定义partitioner： job.setPartitionerClass(CustomPartitioner.class)

### 4.2.3 实现

|  |
| --- |
| /\*\*  \* 定义自己的从map到reduce之间的数据(分组)分发规则 按照手机号所属的省份来分发(分组)ProvincePartitioner  \* 默认的分组组件是HashPartitioner  \*  \* @author  \*  \*/  public class ProvincePartitioner extends Partitioner<Text, FlowBean> {  static HashMap<String, Integer> provinceMap = new HashMap<String, Integer>();  static {  provinceMap.put("135", 0);  provinceMap.put("136", 1);  provinceMap.put("137", 2);  provinceMap.put("138", 3);  provinceMap.put("139", 4);  }  @Override  public int getPartition(Text key, FlowBean value, int numPartitions) {  Integer code = provinceMap.get(key.toString().substring(0, 3));  return code == null ? 5 : code;  }  } |

## 4.3. MapReduce数据压缩

### 4.3.1 概述

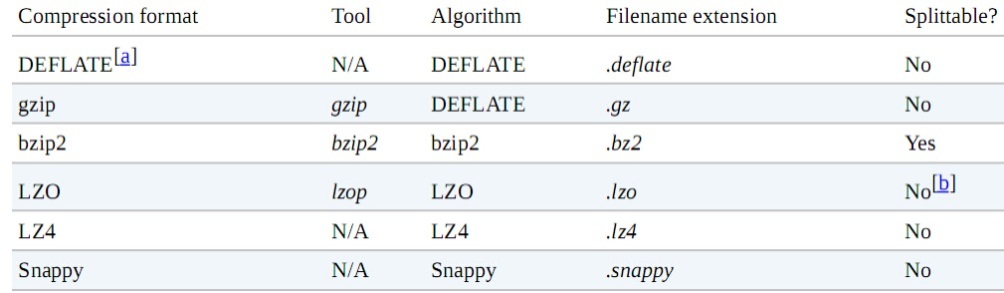
这是**MapReduce的一种优化策略：通过压缩编码对mapper或者reducer的输出进行压缩，以减少磁盘IO，**提高MR程序运行速度(但相应增加了cpu运算负担)

1. MapReduce支持将map输出的结果或者reduce输出的结果进行压缩，以减少网络IO或最终输出数据的体积
2. 压缩特性运用得当能提高性能，但运用不当也可能降低性能
3. 基本原则：

运算密集型的job，少用压缩

IO密集型的job，多用压缩

### 4.3.2 MR支持的压缩编码



### 4.3.3 Reducer输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.output.fileoutputformat.compress=false

mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.DefaultCodec

mapreduce.output.fileoutputformat.compress.type=RECORD

2、在代码中设置

|  |
| --- |
| Job job = Job.getInstance(conf);  FileOutputFormat.setCompressOutput(job, true);  FileOutputFormat.setOutputCompressorClass(job, (Class<? extends CompressionCodec>) Class.forName("")); |

### 4.3.4 Mapper输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.map.output.compress=false

mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.DefaultCodec

2、在代码中设置：

|  |
| --- |
| conf.setBoolean(Job.MAP\_OUTPUT\_COMPRESS, true);  conf.setClass(Job.MAP\_OUTPUT\_COMPRESS\_CODEC, GzipCodec.class, CompressionCodec.class); |

### 4.3.5 压缩文件的读取

Hadoop自带的InputFormat类内置支持压缩文件的读取，比如TextInputformat类，在其initialize方法中：

|  |
| --- |
| public void initialize(InputSplit genericSplit, TaskAttemptContext context) throws IOException {  FileSplit split = (FileSplit) genericSplit;  Configuration job = context.getConfiguration();  this.maxLineLength = job.getInt(MAX\_LINE\_LENGTH, Integer.MAX\_VALUE);  start = split.getStart();  end = start + split.getLength();  final Path file = split.getPath();  // open the file and seek to the start of the split  final FileSystem fs = file.getFileSystem(job);  fileIn = fs.open(file);  //根据文件后缀名创建相应压缩编码的codec  CompressionCodec codec = new CompressionCodecFactory(job).getCodec(file);  if (null!=codec) {  isCompressedInput = true;  decompressor = CodecPool.getDecompressor(codec);  //判断是否属于可切片压缩编码类型  if (codec instanceof SplittableCompressionCodec) {  final SplitCompressionInputStream cIn =  ((SplittableCompressionCodec)codec).createInputStream(  fileIn, decompressor, start, end,  SplittableCompressionCodec.READ\_MODE.BYBLOCK);  //如果是可切片压缩编码，则创建一个CompressedSplitLineReader读取压缩数据  in = new CompressedSplitLineReader(cIn, job,  this.recordDelimiterBytes);  start = cIn.getAdjustedStart();  end = cIn.getAdjustedEnd();  filePosition = cIn;  } else {  //如果是不可切片压缩编码，则创建一个SplitLineReader读取压缩数据，并将文件输入流转换成解压数据流传递给普通SplitLineReader读取  in = new SplitLineReader(codec.createInputStream(fileIn,  decompressor), job, this.recordDelimiterBytes);  filePosition = fileIn;  }  } else {  fileIn.seek(start);  //如果不是压缩文件，则创建普通SplitLineReader读取数据  in = new SplitLineReader(fileIn, job, this.recordDelimiterBytes);  filePosition = fileIn;  } |

## 4.4. 更多MapReduce编程案例

### 4.4.1 reduce端join算法实现

1、需求：

订单数据表t\_order：

|  |  |  |  |
| --- | --- | --- | --- |
| id | date | pid | amount |
| 1001 | 20150710 | P0001 | 2 |
| 1002 | 20150710 | P0001 | 3 |
| 1002 | 20150710 | P0002 | 3 |

商品信息表t\_product

|  |  |  |  |
| --- | --- | --- | --- |
| id | pname | category\_id | price |
| P0001 | 小米5 | 1000 | 2 |
| P0002 | 锤子T1 | 1000 | 3 |

假如数据量巨大，两表的数据是以文件的形式存储在HDFS中，需要用MapReduce程序来实现一下SQL查询运算：

|  |
| --- |
| select a.id,a.date,b.name,b.category\_id,b.price from t\_order a join t\_product b on a.pid = b.id |

2、实现机制：

通过将关联的条件作为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reduce task，在reduce中进行数据的串联

|  |
| --- |
| public class OrderJoin {  static class OrderJoinMapper extends Mapper<LongWritable, Text, Text, OrderJoinBean> {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 拿到一行数据，并且要分辨出这行数据所属的文件  String line = value.toString();  String[] fields = line.split("\t");  // 拿到itemid  String itemid = fields[0];  // 获取到这一行所在的文件名(通过inpusplit)  String name = "你拿到的文件名";  // 根据文件名，切分出各字段(如果是a，切分出两个字段，如果是b，切分出3个字段)  OrderJoinBean bean = new OrderJoinBean();  bean.set(null, null, null, null, null);  context.write(new Text(itemid), bean);  }  }  static class OrderJoinReducer extends Reducer<Text, OrderJoinBean, OrderJoinBean, NullWritable> {  @Override  protected void reduce(Text key, Iterable<OrderJoinBean> beans, Context context) throws IOException, InterruptedException {  //拿到的key是某一个itemid,比如1000  //拿到的beans是来自于两类文件的bean  // {1000,amount} {1000,amount} {1000,amount} --- {1000,price,name}    //将来自于b文件的bean里面的字段，跟来自于a的所有bean进行字段拼接并输出  }  }  } |

缺点：这种方式中，join的操作是在reduce阶段完成，reduce端的处理压力太大，map节点的运算负载则很低，资源利用率不高，且在reduce阶段极易产生数据倾斜

解决方案： map端join实现方式

### 4.4.2 map端join算法实现

1、原理阐述

适用于关联表中有小表的情形；

可以将小表分发到所有的map节点，这样，map节点就可以在本地对自己所读到的大表数据进行join并输出最终结果，可以大大提高join操作的并发度，加快处理速度

2、实现示例

--先在mapper类中预先定义好小表，进行join

--引入实际场景中的解决方案：一次加载数据库或者用distributedcache

|  |
| --- |
| public class TestDistributedCache {  static class TestDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text>{  FileReader in = null;  BufferedReader reader = null;  HashMap<String,String> b\_tab = new HashMap<String, String>();  String localpath =null;  String uirpath = null;    //是在map任务初始化的时候调用一次  @Override  protected void setup(Context context) throws IOException, InterruptedException {  //通过这几句代码可以获取到cache file的本地绝对路径，测试验证用  Path[] files = context.getLocalCacheFiles();  localpath = files[0].toString();  URI[] cacheFiles = context.getCacheFiles();      //缓存文件的用法——直接用本地IO来读取  //这里读的数据是map task所在机器本地工作目录中的一个小文件  in = new FileReader("b.txt");  reader =new BufferedReader(in);  String line =null;  while(null!=(line=reader.readLine())){    String[] fields = line.split(",");  b\_tab.put(fields[0],fields[1]);    }  IOUtils.closeStream(reader);  IOUtils.closeStream(in);    }    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  //这里读的是这个map task所负责的那一个切片数据(在hdfs上)  String[] fields = value.toString().split("\t");    String a\_itemid = fields[0];  String a\_amount = fields[1];    String b\_name = b\_tab.get(a\_itemid);    // 输出结果 1001 98.9 banan  context.write(new Text(a\_itemid), new Text(a\_amount + "\t" + ":" + localpath + "\t" +b\_name ));    }      }      public static void main(String[] args) throws Exception {    Configuration conf = new Configuration();  Job job = Job.getInstance(conf);    job.setJarByClass(TestDistributedCache.class);    job.setMapperClass(TestDistributedCacheMapper.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(LongWritable.class);    //这里是我们正常的需要处理的数据所在路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    //不需要reducer  job.setNumReduceTasks(0);  //分发一个文件到task进程的工作目录  job.addCacheFile(new URI("hdfs://hadoop-server01:9000/cachefile/b.txt"));    //分发一个归档文件到task进程的工作目录  //job.addArchiveToClassPath(archive);  //分发jar包到task节点的classpath下  //job.addFileToClassPath(jarfile);    job.waitForCompletion(true);  }  } |

### 4.4.3 web日志预处理

1、需求：

对web访问日志中的各字段识别切分

去除日志中不合法的记录

根据KPI统计需求，生成各类访问请求过滤数据

2、实现代码：

a) 定义一个bean，用来记录日志数据中的各数据字段

|  |
| --- |
| public class WebLogBean {  private String remote\_addr;// 记录客户端的ip地址  private String remote\_user;// 记录客户端用户名称,忽略属性"-"  private String time\_local;// 记录访问时间与时区  private String request;// 记录请求的url与http协议  private String status;// 记录请求状态；成功是200  private String body\_bytes\_sent;// 记录发送给客户端文件主体内容大小  private String http\_referer;// 用来记录从那个页面链接访问过来的  private String http\_user\_agent;// 记录客户浏览器的相关信息  private boolean valid = true;// 判断数据是否合法  // Getter& Setter  } |

b)定义一个parser用来解析过滤web访问日志原始记录

|  |
| --- |
| public class WebLogParser {  public static WebLogBean parser(String line) {  WebLogBean webLogBean = new WebLogBean();  String[] arr = line.split(" ");  if (arr.length > 11) {  webLogBean.setRemote\_addr(arr[0]);  webLogBean.setRemote\_user(arr[1]);  webLogBean.setTime\_local(arr[3].substring(1));  webLogBean.setRequest(arr[6]);  webLogBean.setStatus(arr[8]);  webLogBean.setBody\_bytes\_sent(arr[9]);  webLogBean.setHttp\_referer(arr[10]);    if (arr.length > 12) {  webLogBean.setHttp\_user\_agent(arr[11] + " " + arr[12]);  } else {  webLogBean.setHttp\_user\_agent(arr[11]);  }  if (Integer.parseInt(webLogBean.getStatus()) >= 400) {// 大于400，HTTP错误  webLogBean.setValid(false);  }  } else {  webLogBean.setValid(false);  }  return webLogBean;  }    public static String parserTime(String time) {  time.replace("/", "-");  return time;  }  } |

c) MapReduce程序

|  |
| --- |
| public class WeblogPreProcess {  static class WeblogPreProcessMapper extends Mapper<LongWritable, Text, Text, NullWritable> {  Text k = new Text();  NullWritable v = NullWritable.get();  @Override  protected void map(LongWritable key, Text value, Context context) throws Exception {  String line = value.toString();  WebLogBean webLogBean = WebLogParser.parser(line);  if (!webLogBean.isValid())  return;  k.set(webLogBean.toString());  context.write(k, v);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(WeblogPreProcess.class);  job.setMapperClass(WeblogPreProcessMapper.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    job.waitForCompletion(true);    }  } |