

尚硅谷大数据技术之 Hadoop (MapReduce)

(作者:尚硅谷大数据研发部)

版本: V3.3

第1章 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,并发运行在一个 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。

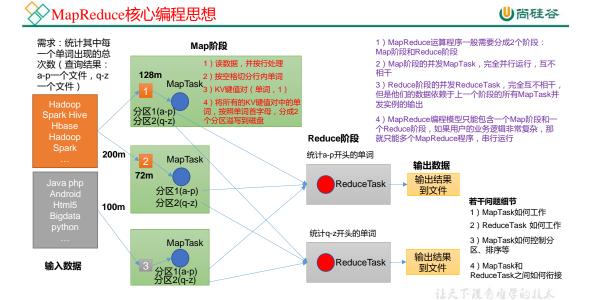
2) 不擅长流式计算

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。

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网



- (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
Мар	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。



MapReduce编程规范



- 1. Mapper阶段
 - (1)用户自定义的Mapper要继承自己的父类
 - (2) Mapper的输入数据是KV对的形式(KV的类型可自定义)
 - (3) Mapper中的业务逻辑写在map()方法中
 - (4) Mapper的输出数据是KV对的形式(KV的类型可自定义)
 - (5) map()方法(MapTask进程)对每一个<K,V>调用一次

让天下没有难学的技术



MapReduce编程规范

●尚硅谷

- 2. Reducer阶段
 - (1)用户自定义的Reducer要继承自己的父类
 - (2) Reducer的输入数据类型对应Mapper的输出数据类型,也是KV
 - (3) Reducer的业务逻辑写在reduce()方法中
 - (4) ReduceTask进程对每一组相同k的<k,v>组调用一次reduce()方法
- 3. Driver阶段

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

让天下没有难学的技术

1.8 WordCount 案例实操

1.8.1 本地测试

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

(1) 输入数据



hello.txt

(2) 期望输出数据

atguigu 2

banzhang 1

cls 2

hadoop 1

jiao 1

ss 2

xue 1

2) 需求分析

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



需求:统计一堆文件中单词出现的个数(WordCount案例)

⊎尚硅谷

```
输入数据
                                                      2、输出数据
           atguigu atguigu
                                                                atguigu 2
                                                                banzhang1
           SS SS
           cls cls
                                                                cls
           iiao
                                                                hadoop 1
           banzhang
                                                                jiao
           xue
                                                                SS
           hadoop
                                                                      1
                                                                xue
3, Mapper
                                4, Reducer
                                                      5, Driver
// 3.1 将MapTask传给我们的文本
                               // 4.1 汇总各个key的个数
                                                      // 5.1 获取配置信息,获取job对象实例
内容先转换成String
   atguigu atguigu
                                  atguigu, 1
                                                      // 5.2 指定本程序的jar包所在的本地路径
                                  atguigu, 1
                                                      // 5.3 关联Mapper/Reducer业务类
// 3.2 根据空格将这一行切分成单词
                               // 4.2 输出该key的总次数
                                                      // 5.4 指定Mapper输出数据的kv类型
   atguigu
                                                      // 5.5 指定最终输出的数据的kv类型
                                  atguigu, 2
   atguigu
// 3.3 将单词输出为<单词, 1>
                                                      // 5.6 指定job的输入原始文件所在目录
                                                      // 5.7 指定job的输出结果所在目录
   atguigu, 1
                                                      // 5.8 提交作业天下没有难得的技术
   atguigu, 1
```

3) 环境准备

- (1) 创建 maven 工程,MapReduceDemo
- (2) 在 pom.xml 文件中添加如下依赖

```
<dependencies>
   <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-client</artifactId>
      <version>3.1.3
   </dependency>
   <dependency>
      <groupId>junit
      <artifactId>junit</artifactId>
      <version>4.12
   </dependency>
   <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.30
   </dependency>
</dependencies>
```

(2) 在项目的 src/main/resources 目录下,新建一个文件,命名为"log4j.properties",在

文件中填入。

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

(3) 创建包名: com.atguigu.mapreduce.wordcount

4) 编写程序



(1) 编写 Mapper 类

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

(2) 编写 Reducer 类

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



(3) 编写 Driver 驱动类

```
package com.atguigu.mapreduce.wordcount;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCountDriver {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        // 1 获取配置信息以及获取 job 对象
       Configuration conf = new Configuration();
       Job job = Job.getInstance(conf);
        // 2 关联本 Driver 程序的 jar
        job.setJarByClass(WordCountDriver.class);
        // 3 关联 Mapper 和 Reducer 的 jar
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        // 4 设置 Mapper 输出的 kv 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        // 5 设置最终输出 kv 类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // 6 设置输入和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
       // 7 提交 job
       boolean result = job.waitForCompletion(true);
       System.exit(result ? 0 : 1);
```

5) 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

1.8.2 提交到集群测试

集群上测试

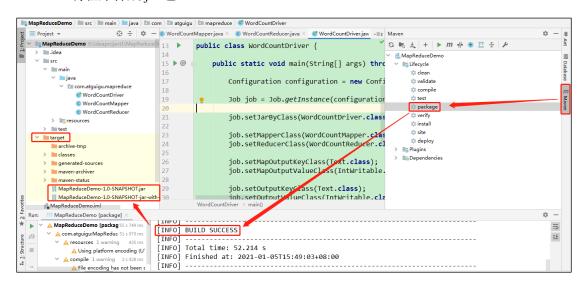
(1) 用 maven 打 jar 包,需要添加的打包插件依赖



```
<version>3.6.1
          <configuration>
             <source>1.8</source>
             <target>1.8</target>
          </configuration>
      </plugin>
      <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
             <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
             </descriptorRefs>
          </configuration>
          <executions>
             <execution>
                <id>make-assembly</id>
                <phase>package</phase>
                <goals>
                    <goal>single</goal>
                </goals>
             </execution>
          </executions>
      </plugin>
   </plugins>
</build>
```

注意:如果工程上显示红叉。在项目上右键->maven->Reimport 刷新即可。

(2) 将程序打成 jar 包



- (3)修改不带依赖的 jar 包名称为 wc.jar, 并拷贝该 jar 包到 Hadoop 集群的/opt/module/hadoop-3.1.3 路径。
 - (4) 启动 Hadoop 集群

```
[atguigu@hadoop102 hadoop-3.1.3]sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

(5) 执行 WordCount 程序

```
[atguigu@hadoop102 hadoop-3.1.3]$ hadoop jar wc.jar com.atguigu.mapreduce.wordcount.WordCountDriver /user/atguigu/input /user/atguigu/output
```



第2章 Hadoop 序列化

2.1 序列化概述

1) 什么是序列化

序列化就是把内存中的对象,转换成字节序列(或其他数据传输协议)以便于存储到磁盘(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<mark>磁盘的持久化数据,转换</mark>成内存中的对象。

2) 为什么要序列化

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

3) 为什么不用 Java 的序列化

Java 的序列化是一个重量级序列化框架(Serializable),一个对象被序列化后,会附带很多额外的信息(各种校验信息,Header,继承体系等),不便于在网络中高效传输。所以,Hadoop 自己开发了一套序列化机制(Writable)。

- 4) 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) 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

(1) 输入数据



phone_data.txt

(2) 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

(3) 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) 需求分析



🔆 序列化案例分析



```
1、需求:统计每一个手机号耗费的总上行流量、下行流量、总流量
```

2、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

4、Map阶段

(1)读取一行数据,切分字段

7 13560436666 120.196.100.99 1116 954 200

下行流量

(2)抽取手机号、上行流量、下行流量 13560436666 1116 954

(3)以手机号为key, bean对象为value输出,

即context.write(手机号,bean);

手机号码 上行流量

(4) bean对象要想能够传输,必须实现序列化接口

3、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

5、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

让天下没有难学的技术

3) 编写 MapReduce 程序

(1) 编写流量统计的 Bean 对象

```
package com.atguigu.mapreduce.writable;
import org.apache.hadoop.io.Writable;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
//1 继承 Writable 接口
public class FlowBean implements Writable {
   private long upFlow; //上行流量
   private long downFlow; //下行流量
   private long sumFlow; //总流量
   //2 提供无参构造
   public FlowBean() {
   }
   //3 提供三个参数的 getter 和 setter 方法
   public long getUpFlow() {
      return upFlow;
   public void setUpFlow(long upFlow) {
      this.upFlow = upFlow;
   public long getDownFlow() {
      return downFlow;
   public void setDownFlow(long downFlow) {
      this.downFlow = downFlow;
   public long getSumFlow() {
      return sumFlow;
```



```
public void setSumFlow(long sumFlow) {
   this.sumFlow = sumFlow;
public void setSumFlow() {
   this.sumFlow = this.upFlow + this.downFlow;
//4 实现序列化和反序列化方法,注意顺序一定要保持一致
@Override
public void write(DataOutput dataOutput) throws IOException {
   dataOutput.writeLong(upFlow);
   dataOutput.writeLong(downFlow);
   dataOutput.writeLong(sumFlow);
@Override
public void readFields(DataInput dataInput) throws IOException {
   this.upFlow = dataInput.readLong();
   this.downFlow = dataInput.readLong();
   this.sumFlow = dataInput.readLong();
//5 重写 ToString
@Override
public String toString() {
   return upFlow + "\t" + downFlow + "\t" + sumFlow;
```

(2) 编写 Mapper 类

```
package com.atguigu.mapreduce.writable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
public class FlowMapper extends Mapper<LongWritable, Text, Text, FlowBean>
   private Text outK = new Text();
   private FlowBean outV = new FlowBean();
   @Override
   protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      //1 获取一行数据,转成字符串
      String line = value.toString();
      //2 切割数据
      String[] split = line.split("\t");
      //3 抓取我们需要的数据:手机号,上行流量,下行流量
      String phone = split[1];
      String up = split[split.length - 3];
      String down = split[split.length - 2];
      //4 封装 outK outV
      outK.set(phone);
```



```
outV.setUpFlow(Long.parseLong(up));
outV.setDownFlow(Long.parseLong(down));
outV.setSumFlow();

//5 写出 outK outV
context.write(outK, outV);
}
```

(3) 编写 Reducer 类

```
package com.atguigu.mapreduce.writable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
public class FlowReducer extends Reducer<Text, FlowBean, Text, FlowBean>
   private FlowBean outV = new FlowBean();
   @Override
   protected void reduce(Text key, Iterable<FlowBean> values, Context
context) throws IOException, InterruptedException {
      long totalUp = 0;
      long totalDown = 0;
      //1 遍历 values,将其中的上行流量,下行流量分别累加
      for (FlowBean flowBean : values) {
          totalUp += flowBean.getUpFlow();
          totalDown += flowBean.getDownFlow();
      //2 封装 outKV
      outV.setUpFlow(totalUp);
      outV.setDownFlow(totalDown);
      outV.setSumFlow();
      //3 写出 outK outV
      context.write(key,outV);
```

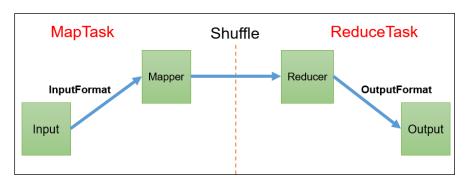
(4) 编写 Driver 驱动类

```
package com.atguigu.mapreduce.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;
public class FlowDriver {
  public static void
                          main(String[] args)
                                                 throws IOException,
ClassNotFoundException, InterruptedException {
      //1 获取 job 对象
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf);
      //2 关联本 Driver 类
```



```
job.setJarByClass(FlowDriver.class);
//3 关联 Mapper 和 Reducer
job.setMapperClass(FlowMapper.class);
job.setReducerClass(FlowReducer.class);
//4 设置 Map 端输出 KV 类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(FlowBean.class);
//5 设置程序最终输出的 KV 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);
//6 设置程序的输入输出路径
FileInputFormat.setInputPaths(job, new Path("D:\\inputflow"));
FileOutputFormat.setOutputPath(job, new Path("D:\\flowoutput"));
//7 提交 Job
boolean b = job.waitForCompletion(true);
System.exit(b ? 0:1);
```

第3章 MapReduce 框架原理



3.1 InputFormat 数据输入

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。

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问:尚硅谷官网

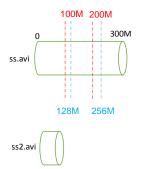




数据切片与MapTask并行度决定机制

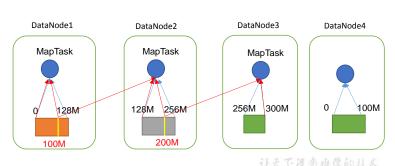
⊎尚硅谷

- 1、假设切片大小设置为100M
- 2、假设切片大小设置为128M



100M

- 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

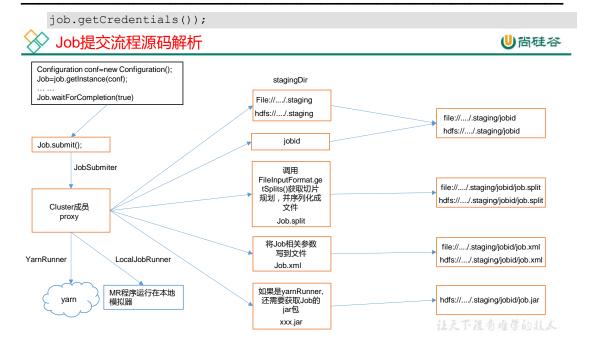


3.1.2 Job 提交流程源码和切片源码详解

1) Job 提交流程源码详解

```
waitForCompletion()
submit();
// 1 建立连接
   connect();
       // 1) 创建提交 Job 的代理
       new Cluster(getConfiguration());
           // (1) 判断是本地运行环境还是 yarn 集群运行环境
           initialize(jobTrackAddr, conf);
// 2 提交 job
   submitter.submitJobInternal(Job.this, cluster)
   // 1) 创建给集群提交数据的 Stag 路径
   Path jobStagingArea = JobSubmissionFiles.getStagingDir(cluster, conf);
   // 2) 获取 jobid , 并创建 Job 路径
   JobID jobId = submitClient.getNewJobID();
   // 3) 拷贝 jar 包到集群
   copyAndConfigureFiles(job, submitJobDir);
   rUploader.uploadFiles(job, jobSubmitDir);
   // 4) 计算切片, 生成切片规划文件
   writeSplits(job, submitJobDir);
       maps = writeNewSplits(job, jobSubmitDir);
       input.getSplits(job);
   // 5) 向 Stag 路径写 XML 配置文件
   writeConf(conf, submitJobFile);
   conf.writeXml(out);
   // 6) 提交 Job, 返回提交状态
   status =
               submitClient.submitJob(jobId, submitJobDir.toString(),
```

尚硅谷大数据技术之 Hadoop (MapReduce)



2) FileInputFormat 切片源码解析 (input.getSplits(job))



FileInputFormat切片源码解析

⊎尚硅谷

- (1)程序先找到你数据存储的目录。
- (2)开始遍历处理(规划切片)目录下的每一个文件
- (3)遍历第一个文件ss.txt
 - a) 获取文件大小fs.sizeOf(ss.txt)
 - b) 计算切片大小 computeSplitSize(Math.max(minSize,Math.min(maxSize,blocksize)))=blocksize=128M
 - c)默认情况下,切片大小=blocksize
 - d) 开始切,形成第1个切片:ss.txt—0:128M第2个切片ss.txt—128:256M第3个切片ss.txt—256M:300M(每次切片时,都要判断切完剩下的部分是否大于块的1.1倍,不大于1.1倍就划分一块切片)
 - e)将切片信息写到一个切片规划文件中
 - f)整个切片的核心过程在getSplit()方法中完成
 - g) InputSplit只记录了切片的元数据信息,比如起始位置、长度以及所在的节点列表等。
- (4)提交切片规划文件到YARN上,YARN上的MrAppMaster就可以根据切片规划文件计算开启MapTask个数。

让天下没有难学的技术



3.1.3 FileInputFormat 切片机制



FileInputFormat切片机制

●尚硅谷

1、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

2、案例分析

(1)输入数据有两个文件:

(2)经过FileInputFormat的切片机制

file1.txt 320M file2.txt 10M

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

让天下没有难学的技术



> FileInputFormat切片大小的参数配置

●尚硅谷

(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

因此,默认情况下,切片大小=blocksize。

(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();

让天下没有难学的技术

3.1.4 TextInputFormat

1) FileInputFormat 实现类

思考:在运行 MapReduce 程序时,输入的文件格式包括:基于行的日志文件、二进制格式文件、数据库表等。那么,针对不同的数据类型,MapReduce 是如何读取这些数据的呢?

FileInputFormat 常见的接口实现类包括: TextInputFormat、KeyValueTextInputFormat、NLineInputFormat、CombineTextInputFormat 和自定义InputFormat 等。

2) TextInputFormat

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)
(20,Intelligent learning engine)
(49,Learning more convenient)
(74,From the real demand for more close to the enterprise)
```

3.1.5 CombineTextInputFormat 切片机制

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。



CombineTextInputFormat切片机制

●尚硅谷

setMaxIn	putSplitSize值是	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
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	setMaxInputSplitSize值,大于等于则单独形成 一个切片。
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
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	进行合并,共同形成一个切片。
		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	让天下没有难学的技术

(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

3.1.6 CombineTextInputFormat 案例实操

1) 需求



将输入的大量小文件合并成一个切片统一处理。

(1) 输入数据

准备 4 个小文件









(2) 期望

期望一个切片处理 4 个文件

2) 实现过程

(1) 不做任何处理,运行 1.8 节的 WordCount 案例程序,观察切片个数为 4。

number of splits:4

- (2) 在 WordcountDriver 中增加如下代码,运行程序,并观察运行的切片个数为 3。
 - (a) 驱动类中添加代码如下:

// 如果不设置 InputFormat, 它默认用的是 TextInputFormat.class job.setInputFormatClass(CombineTextInputFormat.class);

//虚拟存储切片最大值设置 4m

CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);

(b) 运行如果为3个切片。

number of splits:3

- (3) 在 WordcountDriver 中增加如下代码,运行程序,并观察运行的切片个数为 1。
 - (a) 驱动中添加代码如下:

// 如果不设置 InputFormat, 它默认用的是 TextInputFormat.class job.setInputFormatClass(CombineTextInputFormat.class);

//虚拟存储切片最大值设置 20m

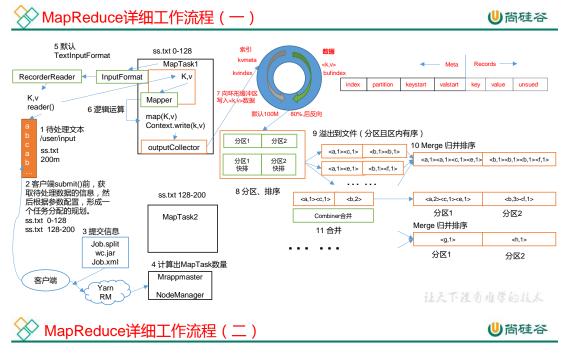
CombineTextInputFormat.setMaxInputSplitSize(job, 20971520);

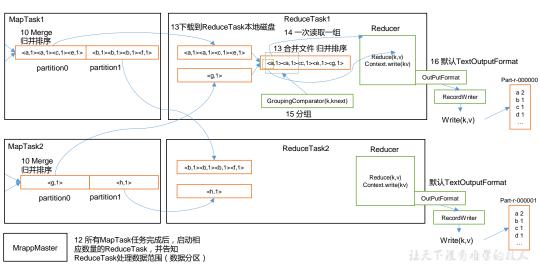
(b) 运行如果为1个切片

number of splits:1



3.2 MapReduce 工作流程





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)

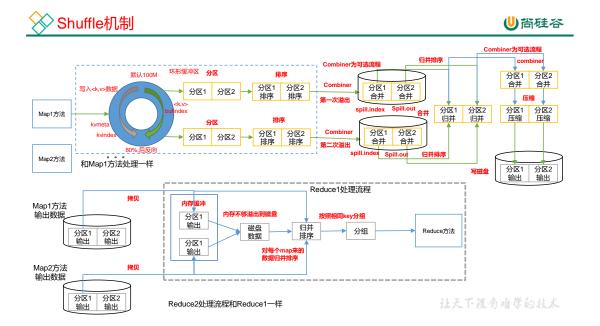
注意:

- (1) 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。
 - (2)缓冲区的大小可以通过参数调整,参数: mapreduce.task.io.sort.mb 默认 100M。

3.3 Shuffle 机制

3.3.1 Shuffle 机制

Map 方法之后, Reduce 方法之前的数据处理过程称之为 Shuffle。





3.3.2 Partition 分区





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。

让天下没有难样的技术





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);

让天下没有难学的技术





●尚硅谷

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

让天下没有难学的技术

3.3.3 Partition 分区案例实操

1)需求

将统计结果按照手机归属地不同省份输出到不同文件中(分区)

(1) 输入数据



phone_data.txt

(2) 期望输出数据

手机号 136、137、138、139 开头都分别放到一个独立的 4 个文件中,其他开头的放到一个文件中。

2) 需求分析



2、数据输入

◇ ₽

Partition分区案例分析

●尚硅谷

1、需求:将统计结果按照手机归属地不同省份输出到不同文件中(分区)

				3、别主奴饰制山		
1363057	7991	6960	690	文件1		
1373623	0513	2481	24681	文件2		
1384654	4121	264	0	文件3		
1395643	5636	132	1512	文件4		
1356043	9638	918	4938	文件5		
4、增加一个ProvincePartitioner分区		区	5、Driver驱动类			
136 137 138	分区0 分区1 分区2			// 指定自定义数据分区 job.setPartitionerClass(ProvincePartitioner class);		
139 其他	分区3 分区4			// 同时指定相应数量的 reduceTask job.setNumReduceTasks(5);		

2 批词粉促烧山

3) 在案例 2.3 的基础上,增加一个分区类

```
package com.atguigu.mapreduce.partitioner;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;
public class ProvincePartitioner extends Partitioner<Text, FlowBean> {
   public int getPartition(Text text, FlowBean flowBean, int numPartitions)
      //获取手机号前三位 prePhone
      String phone = text.toString();
      String prePhone = phone.substring(0, 3);
      //定义一个分区号变量 partition,根据 prePhone 设置分区号
      int partition;
      if("136".equals(prePhone)){
          partition = 0;
      }else if("137".equals(prePhone)){
          partition = 1;
      }else if("138".equals(prePhone)){
          partition = 2;
      }else if("139".equals(prePhone)){
         partition = 3;
      }else {
         partition = 4;
      //最后返回分区号 partition
      return partition;
```

4) 在驱动函数中增加自定义数据分区设置和 ReduceTask 设置

```
package com.atguigu.mapreduce.partitioner;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
```

尚硅谷大数据技术之 Hadoop (MapReduce)

```
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;
public class FlowDriver {
   public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
      //1 获取 job 对象
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf);
      //2 关联本 Driver 类
      job.setJarByClass(FlowDriver.class);
      //3 关联 Mapper 和 Reducer
      job.setMapperClass(FlowMapper.class);
      job.setReducerClass(FlowReducer.class);
      //4 设置 Map 端输出数据的 KV 类型
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(FlowBean.class);
      //5 设置程序最终输出的 KV 类型
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(FlowBean.class);
      //8 指定自定义分区器
      job.setPartitionerClass(ProvincePartitioner.class);
      //9 同时指定相应数量的 ReduceTask
      job.setNumReduceTasks(5);
      //6 设置输入输出路径
      FileInputFormat.setInputPaths(job, new Path("D:\\inputflow"));
      FileOutputFormat.setOutputPath(job, new Path("D\\partitionout"));
      //7 提交 Job
      boolean b = job.waitForCompletion(true);
      System.exit(b ? 0 : 1);
```



3.3.4 WritableComparable 排序



排序概述

⊎尚硅谷

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。

让天下没有难学的技术





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。

让天下没有难懂的技术



🔆 排序分类

⊎尚硅谷

(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。

让天下没有难学的技术

自定义排序 WritableComparable 原理分析

bean 对象做为 key 传输,需要实现 WritableComparable 接口重写 compareTo 方法, 就可以实现排序。

```
@Override
public int compareTo(FlowBean bean) {
    int result;

    // 按照总流量大小,倒序排列
    if (this.sumFlow > bean.getSumFlow()) {
        result = -1;
    }else if (this.sumFlow < bean.getSumFlow()) {
        result = 1;
    }else {
        result = 0;
    }

    return result;
}</pre>
```

3.3.5 WritableComparable 排序案例实操(全排序)

1)需求

根据案例 2.3 序列化案例产生的结果再次对总流量进行倒序排序。

(1) 输入数据

原始数据

第一次处理后的数据



phone_data.txt

part-r-00000

(2) 期望输出数据



```
      13509468723
      7335
      110349
      117684

      13736230513
      2481
      24681
      27162

      13956435636
      132
      1512
      1644

      13846544121
      264
      0
      264
```

2) 需求分析



WritableComparable排序案例分析(全排序)

●尚硅谷

1、需求:根据手机的总流量进行倒序排序

0 0 0 0 0 0

```
2、输入数据
                                                            3、输出数据
                                                              13509468723 7335
                                                                                    110349
                                                                                               117684
 13736230513 2481
13846544121 264
                       24681
                                  27162
                                                             13736230513 2481
13956435636 132
                                                                                    24681
                                                                                               27162
                                  264
                                                                                    1512
                                                                                               1644
                       1512
                                  1644
  13956435636 132
                                                             13846544121 264
                                                                                               264
 13509468723 7335
                       110349
                                  117684
                                                              . . . . . . .
                                                            6、Reducer类
4、FlowBean实现WritableComparable接口重写compareTo方法
  @Override
                                                               // 循环输出,避免总流量相同情况
  public int compareTo(FlowBean o) {
                                                               for (Text text : values)
      // 倒序排列,按照总流量从大到小
                                                                   context.write(text, key);
       return this.sumFlow > o.getSumFlow() ? -1 : 1;
5、Mapper类
  context.write(bean, 手机号)
```

让天下没有难学的技术

3) 代码实现

(1) FlowBean 对象在在需求 1 基础上增加了比较功能

```
package com.atguigu.mapreduce.writablecompable;
import org.apache.hadoop.io.WritableComparable;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
public class FlowBean implements WritableComparable<FlowBean> {
   private long upFlow; //上行流量
   private long downFlow; //下行流量
   private long sumFlow; //总流量
   //提供无参构造
   public FlowBean() {
   //生成三个属性的 getter 和 setter 方法
   public long getUpFlow() {
      return upFlow;
   public void setUpFlow(long upFlow) {
      this.upFlow = upFlow;
```



```
public long getDownFlow() {
   return downFlow;
public void setDownFlow(long downFlow) {
   this.downFlow = downFlow;
public long getSumFlow() {
   return sumFlow;
public void setSumFlow(long sumFlow) {
   this.sumFlow = sumFlow;
public void setSumFlow() {
   this.sumFlow = this.upFlow + this.downFlow;
//实现序列化和反序列化方法,注意顺序一定要一致
@Override
public void write(DataOutput out) throws IOException {
   out.writeLong(this.upFlow);
   out.writeLong(this.downFlow);
   out.writeLong(this.sumFlow);
public void readFields(DataInput in) throws IOException {
   this.upFlow = in.readLong();
   this.downFlow = in.readLong();
   this.sumFlow = in.readLong();
//重写 ToString,最后要输出 FlowBean
@Override
public String toString() {
   return upFlow + "\t" + downFlow + "\t" + sumFlow;
@Override
public int compareTo(FlowBean o) {
   //按照总流量比较,倒序排列
   if(this.sumFlow > o.sumFlow) {
      return -1;
   }else if(this.sumFlow < o.sumFlow) {</pre>
      return 1;
   }else {
      return 0;
```

(2) 编写 Mapper 类

```
package com.atguigu.mapreduce.writablecompable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```



```
import java.io.IOException;
public class FlowMapper extends Mapper<LongWritable, Text, FlowBean, Text>
   private FlowBean outK = new FlowBean();
   private Text outV = new Text();
   @Override
   protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      //1 获取一行数据
      String line = value.toString();
      //2 按照"\t",切割数据
      String[] split = line.split("\t");
      //3 封装 outK outV
      outK.setUpFlow(Long.parseLong(split[1]));
      outK.setDownFlow(Long.parseLong(split[2]));
      outK.setSumFlow();
      outV.set(split[0]);
      //4 写出 outK outV
      context.write(outK,outV);
```

(3) 编写 Reducer 类

```
package com.atguigu.mapreduce.writablecompable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class FlowReducer extends Reducer<FlowBean, Text, Text, FlowBean>
{
    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    //遍历 values 集合,循环写出,避免总流量相同的情况
    for (Text value : values) {
        //调换 KV 位置,反向写出
        context.write(value,key);
    }
    }
}
```

(4) 编写 Driver 类

```
package com.atguigu.mapreduce.writablecompable;
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;

public class FlowDriver {
```



```
static void main(String[]
                                                   throws
                                                            IOException,
ClassNotFoundException, InterruptedException {
      //1 获取 job 对象
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf);
      //2 关联本 Driver 类
      job.setJarByClass(FlowDriver.class);
      //3 关联 Mapper 和 Reducer
      job.setMapperClass(FlowMapper.class);
      job.setReducerClass(FlowReducer.class);
      //4 设置 Map 端输出数据的 KV 类型
      job.setMapOutputKeyClass(FlowBean.class);
      job.setMapOutputValueClass(Text.class);
      //5 设置程序最终输出的 KV 类型
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(FlowBean.class);
      //6 设置输入输出路径
      FileInputFormat.setInputPaths(job, new Path("D:\\inputflow2"));
      FileOutputFormat.setOutputPath(job, new Path("D:\\comparout"));
      //7 提交 Job
      boolean b = job.waitForCompletion(true);
      System.exit(b ? 0 : 1);
```

3.3.6 WritableComparable 排序案例实操(区内排序)

1) 需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2) 需求分析

基于前一个需求,增加自定义分区类,分区按照省份手机号设置。



分区内排序案例分析



1、数据输入			2、期望数据输出	l			
13509468723 7335 13975057813 11058 13568436656 3597	110349 48243 25635	117684 59301 29232	part-r-00000	13630577991 13682846555		690 2910	7650 4848
13736230513 2481 18390173782 9531 13630577991 6960 15043685818 3659	24681 2412 690 3538	27162 11943 7650 7197	part-r-00001	13736230513 13768778790 13729199489	120	24681 120 0	27162 240 240
13992314666 3008 15910133277 3156 13560439638 918 84188413 4116	3720 2936 4938 1432	6728 6092 5856 5548	part-r-00002	13846544121	264	0	264
13682846555 1938 18271575951 1527 15959002129 1938 13590439668 1116 13956435636 132	2910 2106 180 954 1512	4848 3633 2118 2070 1644	part-r-00003	13975057813 13992314666 13956435636 13966251146	3008 132	48243 3720 1512 0	59301 6728 1644 240
13470253144 180 13846544121 264 13966251146 240 13768778790 120 13729199489 240	180 0 0 120 0	360 264 240 240 240	part-r-00004	13509468723 13568436656 18390173782 15043685818 15910133277	3659	110349 25635 2412 3538 2936	117684 29232 11943 7197 6092
						1137111 4	ate Michael de

3) 案例实操

(1) 增加自定义分区类

```
package com.atguigu.mapreduce.partitionercompable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;
public class ProvincePartitioner2 extends Partitioner<FlowBean, Text> {
   @Override
   public int getPartition(FlowBean flowBean, Text text, int numPartitions)
      //获取手机号前三位
      String phone = text.toString();
      String prePhone = phone.substring(0, 3);
      //定义一个分区号变量 partition,根据 prePhone 设置分区号
      int partition;
      if("136".equals(prePhone)){
         partition = 0;
      }else if("137".equals(prePhone)){
         partition = 1;
       }else if("138".equals(prePhone)){
         partition :
                    = 2;
       }else if("139".equals(prePhone)){
         partition = 3;
       }else {
          partition = 4;
      //最后返回分区号 partition
      return partition;
```

(2) 在驱动类中添加分区类

```
// 设置自定义分区器
job.setPartitionerClass(ProvincePartitioner2.class);
```



// 设置对应的 ReduceTask 的个数 job.setNumReduceTasks(5);

3.3.7 Combiner 合并



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
    Reducer

    357->(3+5+7)/3=5
    (3+5+7+2+6)/5=23/5
    不等于 (5+4)/2=9/2

    26->(2+6)/2=4
```

让天下没有难学的技术

(6) 自定义 Combiner 实现步骤

(a) 自定义一个 Combiner 继承 Reducer, 重写 Reduce 方法

```
public class WordCountCombiner extends Reducer<Text, IntWritable, Text,
IntWritable> {
    private IntWritable outV = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable value : values) {
        sum += value.get();
    }
    outV.set(sum);
    context.write(key,outV);
}
```

(b) 在 Job 驱动类中设置:

job.setCombinerClass(WordCountCombiner.class);

3.3.8 Combiner 合并案例实操

1)需求

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



(1) 数据输入



hello.txt

(2) 期望输出数据

期望: Combine 输入数据多,输出时经过合并,输出数据降低。

2) 需求分析



需求:对每一个MapTask的输出局部汇总(Combiner)

⊎尚硅谷

1、数据输入

banzhang ni hao xihuan hadoop banzhang banzhang ni hao xihuan hadoop banzhang

2、期望输出

Map-Reduce Framework
Map input records=4
Map output records=12
Map output bytes=126
Map output bytes=126
Map output materialized bytes=66
Input split bytes=99
Combine input records=12
Combine output records=5
Reduce input groups=5
Reduce shuffle bytes=66

方案一

- 1)增加一个WordcountCombiner类继承Reducer
- 2) 在WordcountCombiner中
 - (1) 统计单词汇总
 - (2)将统计结果输出

方案二

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

job.setCombinerClass(WordcountReducer.class);

让天下没有难学的技术

3) 案例实操-方案一

(1) 增加一个 WordCountCombiner 类继承 Reducer

```
package com.atguigu.mapreduce.combiner;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
public class WordCountCombiner extends Reducer<Text, IntWritable, Text,
IntWritable> {
   private IntWritable outV = new IntWritable();
   @Override
   protected void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values) {
          sum += value.get();
      //封装 outKV
      outV.set(sum);
```



```
//写出 outKV context.write(key,outV);
}
```

(2) 在 WordcountDriver 驱动类中指定 Combiner

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

4) 案例实操-方案二

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

// 指定需要使用 Combiner, 以及用哪个类作为 Combiner 的逻辑 job.setCombinerClass(WordCountReducer.class);

运行程序,如下图所示

```
Map-Reduce Framework
Map input records=4
Map output records=12
Map output bytes=126
Map output materialized bytes=156
Input split bytes=99
Combine input records=0
Combine output records=0
Reduce input groups=5
Reduce shuffle bytes=156
```

* (OutputFormat (org.apache.hadoop.mapreduce)

NullOutputFormat (org.apache.hadoop.mapreduce.lib.output)
 FileOutputFormat (org.apache.hadoop.mapreduce.lib.output)
 MapFileOutputFormat (org.apache.hadoop.mapreduce.lib.output)
 SequenceFileOutputFormat (org.apache.hadoop.mapreduce.lib.output)

Q DBOutputFormat (org.apache.hadoop.mapreduce.lib.db)

• TextOutputFormat (org.apache.hadoop.mapreduce.lib.output)
• FilterOutputFormat (org.apache.hadoop.mapreduce.lib.output)
• LazyOutputFormat (org.apache.hadoop.mapreduce.lib.output)

3.4 OutputFormat 数据输出

3.4.1 OutputFormat 接口实现类



🪫 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. OutputFormat实现类
- 2. 默认输出格式TextOutputFormat
- 3. 自定义OutputFormat
 - 3.1 应用场景:

例如:输出数据到MySQL/HBase/Elasticsearch等存储框架中。

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

让天下没有难学的技术

3.4.2 自定义 OutputFormat 案例实操

1) 需求

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

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网



输出到 e:/other.log。

(1) 输入数据



log.txt

(2) 期望输出数据



2) 需求分析



自定义OutputFormat案例分析



1、需求:过滤输入的log日志,包含atguigu的网站输出到e:/atguigu.log,不包含atguigu的网站输出到e:/other.log

2、输入数据

http://www.baidu.com http://www.google.com http://cn.bing.com http://www.atguigu.com http://www.sohu.com http://www.sina.com http://www.sin2a.com http://www.sin2desa.com http://www.sindsafa.com

4、自定义一个OutputFormat类

- (1) 创建一个类LogRecordWriter继承RecordWriter
 - (a) 创建两个文件的输出流:atguiguOut、otherOut
 - (b)如果输入数据包含atguigu,输出到atguiguOut流 如果不包含atguigu,輸出到otherOut流

3、输出数据



http://www.atguigu.com



http://cn.bing.com http://www.baidu.com other.loa

http://www.google.com http://www.sin2a.com http://www.sin2desa.com http://www.sina.com http://www.sindsafa.com http://www.sohu.com

5、驱动类Driver

// **要将自定义的输出格式组件设置到**job中 job.setOutputFormatClass(LogOutputFormat.class);

3) 案例实操

(1) 编写 LogMapper 类

```
package com.atguigu.mapreduce.outputformat;
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;
public
                                       Mapper<LongWritable,
        class
                 LogMapper
                             extends
                                                              Text, Text,
NullWritable> {
   @Override
   protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      //不做任何处理,直接写出一行 log 数据
      context.write(value, NullWritable.get());
   }
```



(2) 编写 LogReducer 类

```
package com.atguigu.mapreduce.outputformat;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class LogReducer extends Reducer<Text, NullWritable, Text,
NullWritable> {
    @Override
    protected void reduce(Text key, Iterable<NullWritable> values, Context
context) throws IOException, InterruptedException {
        // 防止有相同的数据,迭代写出
        for (NullWritable value : values) {
            context.write(key,NullWritable.get());
        }
    }
}
```

(3) 自定义一个 LogOutputFormat 类

```
package com.atguigu.mapreduce.outputformat;
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;
public class LogOutputFormat extends FileOutputFormat<Text, NullWritable>
   @Override
   public
                        RecordWriter<Text,
                                                          NullWritable>
getRecordWriter(TaskAttemptContext job)
                                               throws
                                                           IOException,
InterruptedException {
      //创建一个自定义的 RecordWriter 返回
      LogRecordWriter logRecordWriter = new LogRecordWriter(job);
      return logRecordWriter;
```

(4) 编写 LogRecordWriter 类

```
package com.atguigu.mapreduce.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;
public class LogRecordWriter extends RecordWriter<Text, NullWritable> {
    private FSDataOutputStream atguiguOut;
```



```
private FSDataOutputStream otherOut;
   public LogRecordWriter(TaskAttemptContext job) {
      try {
         //获取文件系统对象
         FileSystem fs = FileSystem.get(job.getConfiguration());
         //用文件系统对象创建两个输出流对应不同的目录
         atquiquOut = fs.create(new Path("d:/hadoop/atquiqu.log"));
         otherOut = fs.create(new Path("d:/hadoop/other.log"));
      } catch (IOException e) {
         e.printStackTrace();
   }
   @Override
   public void write(Text key, NullWritable value) throws IOException,
InterruptedException {
     String log = key.toString();
      //根据一行的 log 数据是否包含 atguigu,判断两条输出流输出的内容
      if (log.contains("atguigu")) {
         atguiguOut.writeBytes(log + "\n");
      } else {
         otherOut.writeBytes(log + "\n");
      }
   }
   @Override
   public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
      IOUtils.closeStream(atguiguOut);
      IOUtils.closeStream(otherOut);
```

(5) 编写 LogDriver 类

```
package com.atguigu.mapreduce.outputformat;
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;
public class LogDriver {
  public static void
                          main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
      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);

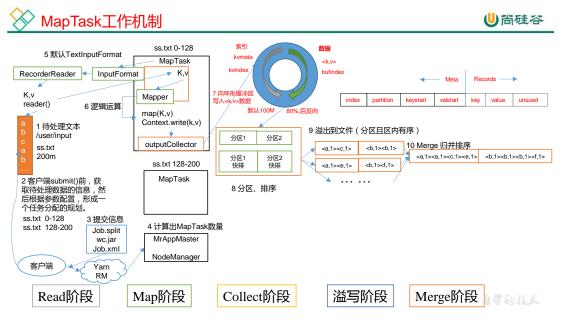
//设置自定义的 outputformat
job.setOutputFormatClass(LogOutputFormat.class);

FileInputFormat.setInputPaths(job, new Path("D:\\input"));
//虽然我们自定义了 outputformat, 但是因为我们的 outputformat 继承自
fileoutputformat
//而 fileoutputformat 要输出一个_SUCCESS 文件, 所以在这还得指定一个输出目录
FileOutputFormat.setOutputPath(job, new Path("D:\\logoutput"));

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

3.5 MapReduce 内核源码解析

3.5.1 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,且同一分区内所有数据按照 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。

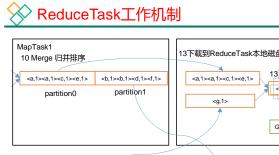
当所有数据处理完后,MapTask 会将所有临时文件合并成一个大文件,并保存到文件 output/file.out 中,同时生成相应的索引文件 output/file.out.index。

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。

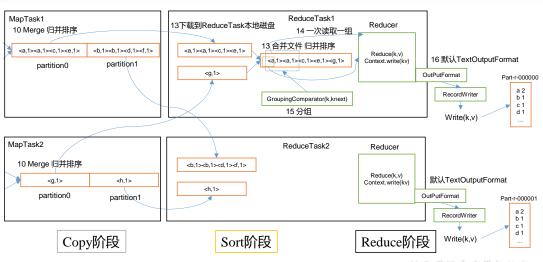
让每个 MapTask 最终只生成一个数据文件,可避免同时打开大量文件和同时读取大量 小文件产生的随机读取带来的开销。



3.5.2 ReduceTask 工作机制



●尚硅谷



- (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) Sort 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。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。
 - (3) Reduce 阶段: reduce()函数将计算结果写到 HDFS 上。

3.5.3 ReduceTask 并行度决定机制

回顾: MapTask 并行度由切片个数决定,切片个数由输入文件和切片规则决定。

思考: ReduceTask 并行度由谁决定?

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:

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网



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。

让天下没有难管的技术

3.5.4 MapTask & ReduceTask 源码解析

1) MapTask 源码解析流程

====== MapTask ======
context.write(k, NullWritable.get()); //自定义的 map 方法的写出,进入
output.write(key, value);
//MapTask727 行,收集方法,进入两次
collector.collect(key, value,partitioner.getPartition(key, value, partitions));
HashPartitioner(); //默认分区器
collect() //MapTask1082 行 map 端所有的 kv 全部写出后会走下面的 close 方法
close() //MapTask732 行
collector.flush() // 溢出刷写方法,MapTask735 行, <mark>提前打个断点</mark> ,进入
sortAndSpill() //溢写排序,MapTask1505 行,进入
sorter.sort() QuickSort //溢写排序方法,MapTask1625 行,进入
mergeParts(); //合并文件,MapTask1527 行,进入
ifile.out
ifile.out.index
collector.close(); //MapTask739 行,收集器关闭,即将进入 ReduceTask
ReduceTask 源码解析流程

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网

====== ReduceTask ==

if (isMapOrReduce()) //reduceTask324 行, 提前打断点



// reduceTask333 行,进入 init(shuffleContext); // reduceTask375 行,走到这需要先给下面的打断点 totalMaps = job.getNumMapTasks(); // ShuffleSchedulerImpl 第 120 行,提前打断点 merger = createMergeManager(context); //合并方法, Shuffle 第 80 行 // MergeManagerImpl 第 232 235 行,提前打断点 this.inMemoryMerger = createInMemoryMerger(); //内存合并 this.onDiskMerger = new OnDiskMerger(this); //磁盘合并 rIter = shuffleConsumerPlugin.run(); eventFetcher.start(); //开始抓取数据, Shuffle 第 107 行,提前打断点 eventFetcher.shutDown(); //抓取结束, Shuffle 第 141 行,提前打断点 copyPhase.complete(); //copy 阶段完成, Shuffle 第 151 行 taskStatus.setPhase(TaskStatus.Phase.SORT); //开始排序阶段, Shuffle 第 152 行 //排序阶段完成,即将进入 reduce 阶段 reduce Task 382 行 sortPhase.complete(); reduce(); //reduce 阶段调用的就是我们自定义的 reduce 方法,会被调用多次 cleanup(context); //reduce 完成之前, 会最后调用一次 Reducer 里面的 cleanup 方法

3.6 Join 应用

3.6.1 Reduce Join

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。

3.6.2 Reduce Join 案例实操

1) 需求



表 4-4 订单数据表 t order

id	pid	amount
1001	01	1
1002	02	2
1003	03	3
1004	01	4
1005	02	5
1006	03	6

表 4-5 商品信息表 t_product



pid	pname
01	小米



02	华为
03	格力

将商品信息表中数据根据商品 pid 合并到订单数据表中。

表 4-6 最终数据形式

-		
id	pname	amount
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) 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 Reduce Task, 在 Reduce 中进行数据的串联。



🪫 Reduce端表合并 (数据倾斜)



1	输	λ	数:	ĺ

	oluel.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

产品名称 pid 小米华为 02

2)默认对产品id排序

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

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

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	构力	6

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

3) 代码实现

(1) 创建商品和订单合并后的 TableBean 类

```
package com.atguigu.mapreduce.reducejoin;
import org.apache.hadoop.io.Writable;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
public class TableBean implements Writable {
   private String id; //订单id
   private String pid; //产品id
   private int amount; //产品数量
```



```
private String pname; //产品名称
private String flag; //判断是 order 表还是 pd 表的标志字段
public TableBean() {
public String getId() {
  return id;
public void setId(String id) {
  this.id = id;
public String getPid() {
  return pid;
public void setPid(String pid) {
   this.pid = pid;
public int getAmount() {
   return amount;
public void setAmount(int amount) {
  this.amount = amount;
public String getPname() {
  return pname;
public void setPname(String pname) {
  this.pname = pname;
public String getFlag() {
  return flag;
public void setFlag(String flag) {
   this.flag = flag;
@Override
public String toString() {
   return id + "\t" + pname + "\t" + amount;
@Override
public void write(DataOutput out) throws IOException {
  out.writeUTF(id);
  out.writeUTF(pid);
  out.writeInt(amount);
  out.writeUTF(pname);
   out.writeUTF(flag);
@Override
public void readFields(DataInput in) throws IOException {
```



```
this.id = in.readUTF();
this.pid = in.readUTF();
this.amount = in.readInt();
this.pname = in.readUTF();
this.flag = in.readUTF();
}
```

(2) 编写 TableMapper 类

```
package com.atguigu.mapreduce.reducejoin;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import java.io.IOException;
public class TableMapper extends Mapper<LongWritable, Text, Text, TableBean>
   private String filename;
   private Text outK = new Text();
   private TableBean outV = new TableBean();
   @Override
   protected
                      setup(Context context) throws IOException,
              void
InterruptedException {
      //获取对应文件名称
      InputSplit split = context.getInputSplit();
      FileSplit fileSplit = (FileSplit) split;
      filename = fileSplit.getPath().getName();
   protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      //获取一行
      String line = value.toString();
      //判断是哪个文件,然后针对文件进行不同的操作
      if(filename.contains("order")){ //订单表的处理
         String[] split = line.split("\t");
         //封装 outK
         outK.set(split[1]);
         //封装 outV
         outV.setId(split[0]);
         outV.setPid(split[1]);
         outV.setAmount(Integer.parseInt(split[2]));
         outV.setPname("");
         outV.setFlag("order");
      }else {
                                     //商品表的处理
         String[] split = line.split("\t");
         //封装 outK
         outK.set(split[0]);
         //封装 outV
         outV.setId("");
         outV.setPid(split[0]);
         outV.setAmount(0);
         outV.setPname(split[1]);
```



```
outV.setFlag("pd");
}

//写出 KV
context.write(outK,outV);
}
```

(3) 编写 TableReducer 类

```
package com.atguigu.mapreduce.reducejoin;
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;
public class TableReducer extends Reducer<Text, TableBean, TableBean,
NullWritable> {
   @Override
   protected void reduce(Text key, Iterable<TableBean> values, Context
context) throws IOException, InterruptedException {
      ArrayList<TableBean> orderBeans = new ArrayList<>();
      TableBean pdBean = new TableBean();
      for (TableBean value : values) {
          //判断数据来自哪个表
          if("order".equals(value.getFlag())){ //订单表
             //创建一个临时 TableBean 对象接收 value
             TableBean tmpOrderBean = new TableBean();
             try {
                BeanUtils.copyProperties(tmpOrderBean, value);
             } catch (IllegalAccessException e) {
                e.printStackTrace();
             } catch (InvocationTargetException e) {
                e.printStackTrace();
             //将临时 TableBean 对象添加到集合 orderBeans
             orderBeans.add(tmpOrderBean);
          }else {
                                              //商品表
             try {
                BeanUtils.copyProperties(pdBean, value);
             } catch (IllegalAccessException e) {
                e.printStackTrace();
             } catch (InvocationTargetException e) {
                e.printStackTrace();
      //遍历集合 orderBeans,替换掉每个 orderBean 的 pid 为 pname,然后写出
      for (TableBean orderBean : orderBeans) {
```



```
orderBean.setPname(pdBean.getPname());

//写出修改后的 orderBean 对象
context.write(orderBean,NullWritable.get());
}

}
```

(4) 编写 TableDriver 类

```
package com.atguigu.mapreduce.reducejoin;
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;
public class TableDriver {
   public static void
                           main(String[] args)
                                                  throws
                                                             IOException,
ClassNotFoundException, InterruptedException {
      Job job = Job.getInstance(new Configuration());
      job.setJarByClass(TableDriver.class);
       job.setMapperClass(TableMapper.class);
       job.setReducerClass(TableReducer.class);
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(TableBean.class);
       job.setOutputKeyClass(TableBean.class);
       job.setOutputValueClass(NullWritable.class);
      FileInputFormat.setInputPaths(job, new Path("D:\\input"));
      FileOutputFormat.setOutputPath(job, new Path("D:\\output"));
      boolean b = job.waitForCompletion(true);
      System.exit(b ? 0 : 1);
```

4) 测试

运行程序查看结果

```
    1004
    小米
    4

    1001
    小米
    1

    1005
    华为
    5

    1002
    华为
    2

    1006
    格力
    6

    1003
    格力
    3
```

5) 总结

缺点:这种方式中,合并的操作是在 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 端实现数据合并。

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网



3.6.3 Map Join

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://hadoop102:8020/cache/pd.txt"));
```

3.6.4 Map Join 案例实操

1) 需求

表 订单数据表 t_order



id	pid	amount
1001	01	1
1002	02	2
1003	03	3
1004	01	4
1005	02	5
1006	03	6



pid	pname
01	小米
02	华为
03	格力

将商品信息表中数据根据商品 pid 合并到订单数据表中。

表 最终数据形式

表 商品信息表 t product

id	pname	amount
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) 需求分析

MapJoin 适用于关联表中有小表的情形。



Map端表合并案例分析 (Distributed cache)

⊎尚硅谷

1) DistributedCacheDriver 缓存文件

2)读取缓存的文件数据

// 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);

 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pid

01.小米 02.华为 // 5 拼接 03.格力

// 6 写出

让天下没有难停的技术

3) 实现代码

(1) 先在 MapJoinDriver 驱动类中添加缓存文件

```
package com.atguigu.mapreduce.mapjoin;
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;
import java.net.URISyntaxException;
public class MapJoinDriver {
   public static void main(String[] args) throws IOException,
URISyntaxException, ClassNotFoundException, InterruptedException {
      // 1 获取 job 信息
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf);
      // 2 设置加载 jar 包路径
      job.setJarByClass(MapJoinDriver.class);
      // 3 美联 mapper
      job.setMapperClass(MapJoinMapper.class);
      // 4 设置 Map 输出 KV 类型
```



```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(NullWritable.class);
// 5 设置最终输出 KV 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NullWritable.class);

// 加载缓存数据
job.addCacheFile(new URI("file:///D:/input/tablecache/pd.txt"));
// Map 端 Join 的逻辑不需要 Reduce 阶段,设置 reduceTask 数量为 0
job.setNumReduceTasks(0);

// 6 设置输入输出路径
FileInputFormat.setInputPaths(job, new Path("D:\\input"));
FileOutputFormat.setOutputPath(job, new Path("D:\\input"));
// 7 提交
boolean b = job.waitForCompletion(true);
System.exit(b ? 0 : 1);
}
```

(2) 在 MapJoinMapper 类中的 setup 方法中读取缓存文件

```
package com.atguigu.mapreduce.mapjoin;
import org.apache.commons.lang.StringUtils;
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;
import java.util.Map;
public class MapJoinMapper extends Mapper<LongWritable, Text, Text,
NullWritable> {
   private Map<String, String> pdMap = new HashMap<>();
   private Text text = new Text();
   //任务开始前将 pd 数据缓存进 pdMap
   @Override
   protected
             void
                     setup(Context context)
                                               throws IOException,
InterruptedException {
      //通过缓存文件得到小表数据 pd.txt
      URI[] cacheFiles = context.getCacheFiles();
      Path path = new Path(cacheFiles[0]);
      //获取文件系统对象,并开流
      FileSystem fs = FileSystem.get(context.getConfiguration());
      FSDataInputStream fis = fs.open(path);
      //通过包装流转换为 reader,方便按行读取
      BufferedReader
                        reader
                                            new
                                                    BufferedReader (new
InputStreamReader(fis, "UTF-8"));
```



```
//逐行读取,按行处理
      String line;
      while (StringUtils.isNotEmpty(line = reader.readLine())) {
         //切割一行
         //01 小米
         String[] split = line.split("\t");
         pdMap.put(split[0], split[1]);
      //关流
      IOUtils.closeStream(reader);
   @Override
  protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      //读取大表数据
      //1001 01 1
      String[] fields = value.toString().split("\t");
      //通过大表每行数据的 pid, 去 pdMap 里面取出 pname
      String pname = pdMap.get(fields[1]);
      //将大表每行数据的 pid 替换为 pname
      text.set(fields[0] + "\t" + pname + "\t" + fields[2]);
      //写出
      context.write(text, NullWritable.get());
```

3.7 数据清洗 (ETL)

"ETL,是英文 Extract-Transform-Load 的缩写,用来描述将数据从来源端经过抽取(Extract)、转换(Transform)、加载(Load)至目的端的过程。ETL 一词较常用在数据仓库,但其对象并不限于数据仓库

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

去除日志中字段个数小于等于11的日志。

(1) 输入数据



web.log

(2) 期望输出数据

每行字段长度都大于11。



2) 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。

3) 实现代码

(1) 编写 WebLogMapper 类

```
package com.atguigu.mapreduce.weblog;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class WebLogMapper extends Mapper<LongWritable, Text, Text,
NullWritable>{
   @Override
   protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
       // 1 获取 1 行数据
       String line = value.toString();
       // 2 解析日志
       boolean result = parseLog(line,context);
       // 3 日志不合法退出
       if (!result) {
           return:
       // 4 日志合法就直接写出
       context.write(value, NullWritable.get());
    }
   // 2 封装解析日志的方法
   private boolean parseLog(String line, Context context) {
       // 1 截取
       String[] fields = line.split(" ");
       // 2 日志长度大于 11 的为合法
       if (fields.length > 11) {
           return true;
       }else {
           return false;
   }
```

(2) 编写 WebLogDriver 类

```
package com.atguigu.mapreduce.weblog;
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;
```



```
public class WebLogDriver {
   public static void main(String[] args) throws Exception {
      // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
      args = new String[] { "D:/input/inputlog", "D:/output1" };
       // 1 获取 job 信息
       Configuration conf = new Configuration();
       Job job = Job.getInstance(conf);
       // 2 加载 jar 包
       job.setJarByClass(LogDriver.class);
       // 3 关联 map
       job.setMapperClass(WebLogMapper.class);
       // 4 设置最终输出类型
       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(NullWritable.class);
       // 设置 reducetask 个数为 0
       job.setNumReduceTasks(0);
       // 5 设置输入和输出路径
       FileInputFormat.setInputPaths(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));
       // 6 提交
       boolean b = job.waitForCompletion(true);
       System.exit(b ? 0 : 1);
```

3.8 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) 用户还可以自定义 OutputFormat。

第4章 Hadoop 数据压缩

4.1 概述

1) 压缩的好处和坏处

压缩的优点:以减少磁盘 IO、减少磁盘存储空间。

压缩的缺点:增加 CPU 开销。

2) 压缩原则

- (1) 运算密集型的 Job, 少用压缩
- (2) IO 密集型的 Job, 多用压缩

4.2 MR 支持的压缩编码

1) 压缩算法对比介绍

压缩格式	Hadoop 自带?	算法	文件扩展 名	是否可 切片	换成压缩格式后,原来的 程序是否需要修改
DEFLATE	是,直接使用	DEFLATE	.deflate	否	和文本处理一样,不需要 修改



Gzip	是,直接使用	DEFLATE	. gz	否	和文本处理一样,不需要 修改
bzip2	是,直接使用	bzip2	. bz2	是	和文本处理一样,不需要 修改
LZ0	否,需要安装	LZ0	. Izo	是	需要建索引,还需要指定 输入格式
Snappy	是,直接使用	Snappy	. snappy	否	和文本处理一样,不需要 修改

2) 压缩性能的比较

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8. 3GB	1. 8GB	17.5MB/s	58MB/s
bzip2	8. 3GB	1. 1GB	2.4MB/s	9.5MB/s
LZ0	8. 3GB	2. 9GB	49.3MB/s	74.6MB/s

http://google.github.io/snappy/

Snappy is a compression/decompression library. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

4.3 压缩方式选择

压缩方式选择时重点考虑: 压缩/解压缩速度、压缩率(压缩后存储大小)、压缩后是否可以支持切片。

4.3.1 Gzip 压缩

优点:压缩率比较高;

缺点:不支持 Split; 压缩/解压速度一般;

4.3.2 Bzip2 压缩

优点:压缩率高;支持 Split;

缺点:压缩/解压速度慢。

4.3.3 Lzo 压缩

优点:压缩/解压速度比较快;支持 Split;

缺点:压缩率一般;想支持切片需要额外创建索引。

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网

Reduce



4.3.4 Snappy 压缩

优点: 压缩和解压缩速度快;

缺点:不支持 Split; 压缩率一般;

4.3.5 压缩位置选择

压缩可以在 MapReduce 作用的任意阶段启用。

Map



MapReduce数据压缩



输入端采用压缩

无须显示指定使用的编解码方 式。Hadoop自动检查文件扩 展名,如果扩展名能够匹配, 就会用恰当的编解码方式对文 件进行压缩和解压。

企业开发:考虑因素

- 1)数据量小于块大小,重点 考虑压缩和解压缩速度比较快 的LZO/Snappy
- 2)数据量非常大,重点考虑 支持切片的Bzip2和LZQ

Mapper输出采用压缩

企业开发中如何选择:为了减 少MapTask和ReduceTask之间的网络 IO。重点考虑压缩和解压缩快的 LZO, Snappy.

Reducer输出采用压缩

看需求:

如果数据永久保存,考虑压缩 率比较高的Bzip2和Gzip。

如果作为下一个MapReduce输 入,需要考虑数据量和是否支持切 片。

4.4 压缩参数配置

1) 为了支持多种压缩/解压缩算法, Hadoop 引入了编码/解码器

压缩格式	对应的编码/解码器
DEFLATE	org. apache. hadoop. io. compress. DefaultCodec
gzip	org. apache. hadoop. io. compress. GzipCodec
bzip2	org. apache. hadoop. io. compress. BZip2Codec
LZ0	com. hadoop. compression. Izo. LzopCodec
Snappy	org. apache. hadoop. io. compress. SnappyCodec

2) 要在 Hadoop 中启用压缩,可以配置如下参数

参数	默认值	阶段	建议
io.compression.codecs	无,这个需要在命令行输入	输入压缩	Hadoop 使用文件扩展
(在 core-site. xml 中配置)	hadoop checknative 查看		名判断是否支持某种 编解码器



mapreduce. map. output. compress (在 mapred-site. xml 中配置)	false	mapper 输出	这个参数设为 true 启 用压缩
mapreduce. map. output. compress. codec(在 mapred-site. xml 中配置)	org. apache. hadoop. io. com press. DefaultCodec	mapper 输出	企业多使用 LZO 或 Snappy 编解码器在此 阶段压缩数据
mapreduce.output.fileoutpu tformat.compress(在 mapred-site.xml 中配置)	false	reducer 输出	这个参数设为 true 启 用压缩
mapreduce. output. fileoutpu tformat. compress. codec(在 mapred-site. xml 中配置)	org.apache.hadoop.io.com press.DefaultCodec	reducer 输出	使用标准工具或者编解码器,如 gzip 和bzip2

4.5 压缩实操案例

4.5.1 Map 输出端采用压缩

即使你的 MapReduce 的输入输出文件都是未压缩的文件,你仍然可以对 Map 任务的中间结果输出做压缩,因为它要写在硬盘并且通过网络传输到 Reduce 节点,对其压缩可以提高很多性能,这些工作只要设置两个属性即可,我们来看下代码怎么设置。

1) 给大家提供的 Hadoop 源码支持的压缩格式有: BZip2Codec、DefaultCodec

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCountDriver {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
       Configuration conf = new Configuration();
       // 开启 map 端输出压缩
       conf.setBoolean("mapreduce.map.output.compress", true);
       // 设置 map 端输出压缩方式
       conf.setClass("mapreduce.map.output.compress.codec",
BZip2Codec.class,CompressionCodec.class);
```



```
Job job = Job.getInstance(conf);

job.setJarByClass(WordCountDriver.class);

job.setMapperClass(WordCountMapper.class);

job.setReducerClass(WordCountReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputKeyClass(IntWritable.class);

FileInputFormat.setInputPaths(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);
}
```

2) Mapper 保持不变

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

3) Reducer 保持不变

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
```



4.5.2 Reduce 输出端采用压缩

基于 WordCount 案例处理。

1) 修改驱动

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.DefaultCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.io.compress.Lz4Codec;
import org.apache.hadoop.io.compress.SnappyCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCountDriver {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(WordCountDriver.class);
        job.setMapperClass(WordCountMapper.class);
```



```
job.setReducerClass(WordCountReducer.class);
       job.setMapOutputKeyClass(Text.class);
       job.setMapOutputValueClass(IntWritable.class);
       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(IntWritable.class);
       FileInputFormat.setInputPaths(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));
       // 设置 reduce 端输出压缩开启
       FileOutputFormat.setCompressOutput(job, true);
       // 设置压缩的方式
       FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);
       FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
       FileOutputFormat.setOutputCompressorClass(job,
DefaultCodec.class);
       boolean result = job.waitForCompletion(true);
       System.exit(result?0:1);
```

2) Mapper 和 Reducer 保持不变(<mark>详见 4.5.</mark>1)

第5章 常见错误及解决方案

- 1) 导包容易出错。尤其 Text 和 CombineTextInputFormat。
- 2)Mapper 中第一个输入的参数必须是 LongWritable 或者 NullWritable, 不可以是 IntWritable. 报的错误是类型转换异常。
- 3) java.lang.Exception: java.io.IOException: Illegal partition for 13926435656 (4), 说明 Partition 和 ReduceTask 个数没对上,调整 ReduceTask 个数。
- 4) 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。
- 5) 在 Windows 环境编译的 jar 包导入到 Linux 环境中运行,

hadoop jar wc.jar com.atguigu.mapreduce.wordcount.WordCountDriver /user/atguigu/ /user/atguigu/output

报如下错误:

Exception in thread "main" java.lang.UnsupportedClassVersionError: com/atguigu/mapreduce/wordcount/WordCountDriver: Unsupported major.minor version 52.0



原因是 Windows 环境用的 jdk1.7, Linux 环境用的 jdk1.8。

解决方案:统一jdk版本。

6)缓存 pd.txt 小文件案例中,报找不到 pd.txt 文件

原因:大部分为路径书写错误。还有就是要检查 pd.txt.txt 的问题。还有个别电脑写相对路径 找不到 pd.txt,可以修改为绝对路径。

7) 报类型转换异常。

通常都是在驱动函数中设置 Map 输出和最终输出时编写错误。

Map 输出的 key 如果没有排序,也会报类型转换异常。

8) 集群中运行 wc.jar 时出现了无法获得输入文件。

原因: WordCount 案例的输入文件不能放用 HDFS 集群的根目录。

9) 出现了如下相关异常

Exception in thread "main" java.lang.UnsatisfiedLinkError:

org.apache.hadoop.io.nativeio.NativeIO\$Windows.access0(Ljava/lang/String;I)Z

at org.apache.hadoop.io.nativeio.NativeIO\$Windows.access0(Native Method)

at org.apache.hadoop.io.nativeio.NativeIO\$Windows.access(NativeIO.java:609)

at org.apache.hadoop.fs.FileUtil.canRead(FileUtil.java:977)

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

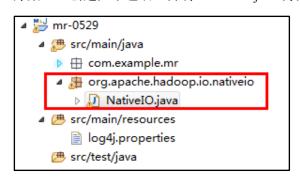
at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:356)

at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:371)

at org.apache.hadoop.util.Shell.<clinit>(Shell.java:364)

解决方案: 拷贝 hadoop.dll 文件到 Windows 目录 C:\Windows\System32。个别同学电脑还需要修改 Hadoop 源码。

方案二: 创建如下包名,并将 NativeIO.java 拷贝到该包名下





10) 自定义 Outputformat 时,注意在 RecordWirter 中的 close 方法必须关闭流资源。否则输



出的文件内容中数据为空。

```
@Override
public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
    if (atguigufos != null) {
        atguigufos.close();
    }
    if (otherfos != null) {
        otherfos.close();
    }
}
```