

HDFS

DN多目录配置_案例

DataNode也可以配置成多目录，每个目录存储的数据不一样，即：数据不是副本

集体配置如下：

```
<property>
<name>dfs.datanode.data.dir</name>
<value>file://${hadoop.tmp.dir}/dfs/data1,file://${hadoop.tmp.dir}/dfs/data2</value>
</property>
```

(所有节点) 删除data, logs, 格式化namenode, 重新启动集群

向集群上传2个文件，分别查看data1和data2，发现只是将数据存在了两个不同的目录里，而不是备份

HDFS新特性

集群间数据拷贝

- SCP命令实现两个远程主机之间的文件复制

```
#推
scp -r hello.txt root@tarena103:/java/servlet/hello.txt
#拉
scp -r root@tarena103:/java/servlet/hello.txt ./hello.txt
#通过本地服务器中转另外两个远程服务器的文件复制，如果在两个远程主机之间ssh没有配置的情况下，可以使用
scp -r root@tarena103:/java/servlet/hello.txt root@tarena102:/zg/cq/hello.txt
```

- 采用distcp命令实现两个hadoop集群之间的递归数据复制

```
#注意两个集群地址之间有空格
[hwhadoop@tarena101 hadoop-2.9.2]$ bin/hadoop distcp
hdfs://tarena102:9000/zg/cq/hello
hdfs://tarena103:9000/java/servlet/hello.txt
```

小文件归档案例

1、HDFS存储小文件弊端

每个文件均按块存储，每个块的元数据存储在NameNode的内存中，因此HDFS存储小文件会非常低效。因为**大量的小文件会耗尽NameNode中的大部分内存。但注意，存储小文件所需要的磁盘容量和数据块的大小无关。**例如，一个1MB的文件设置为128MB的块存储，实际使用的是1MB的磁盘空间，而不是128MB。

2、解决存储小文件办法之一

HDFS存档文件或HAR文件，是一个更高效的文件存档工具，它将文件存入HDFS块，在减少NameNode内存使用的同时，允许对文件进行透明的访问。**具体说来，HDFS存档文件对内还是一个一个独立文件，对NameNode而言却是一个整体，减少了NameNode的内存。**



(1) 需要启动 YARN 进程

```
[atguigu@hadoop102 hadoop-2.7.2]$ start-yarn.sh
```

(2) 归档文件。

把 /user/atguigu/input 目录里面的所有文件归档成一个叫 input.har 的归档文件，并把归档后文件存储到 /user/atguigu/output 路径下。

```
[atguigu@hadoop102 hadoop-2.7.2]$ bin/hadoop archive -  
archiveName      input.har      -p          /user/atguigu/input  
/user/atguigu/output,
```

(3) 查看归档。

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -lsr  
/user/atguigu/output/input.har  
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -lsr  
har:///user/atguigu/output/input.har
```

(4) 解归档文件。

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -cp har:///  
user/atguigu/output/input.har/* /user/atguigu/
```

向集群上传3个小文件

```
hadoop fs -put p1.txt /user/tarena/input  
hadoop fs -put p2.txt /user/tarena/input  
hadoop fs -put p3.txt /user/tarena/input
```

按照上述操作将3个文件归档为har文件

查看是看不到har文件信息的，所以要使用har:///协议查看har文件。

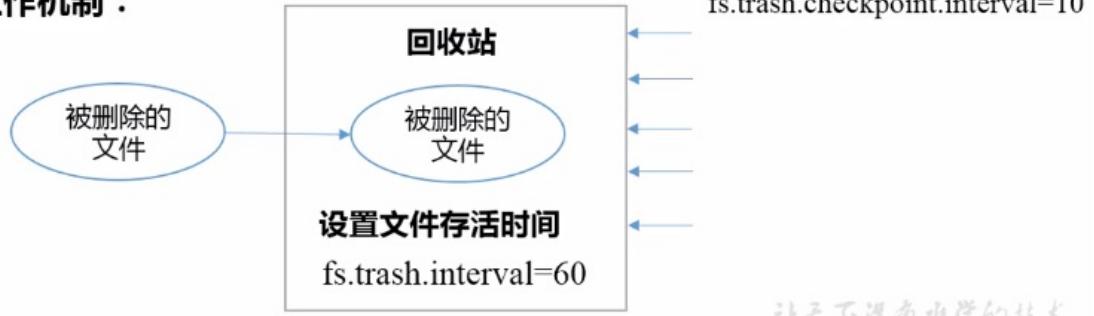
回收站案例

开启回收站功能，可以将删除的文件在不超时的情况下，恢复元数据，起到防止误删除，备份等作用

一、开启回收站功能参数说明：

- 1、默认值fs.trash.interval=0，0表示禁用回收站；**其他值表示设置文件的存活时间。**
- 2、默认值fs.trash.checkpoint.interval=0，检查回收站的间隔时间。**如果该值为0，则该值设置和fs.trash.interval的参数值相等。**
- 3、要求fs.trash.checkpoint.interval<=fs.trash.interval。

二、回收站工作机制：



2. 启用回收站

修改 core-site.xml，配置垃圾回收时间为 1 分钟。

```
<property>
    <name>fs.trash.interval</name>
    <value>1</value>
</property>
```

3. 查看回收站

回收站在集群中的路径：/user/atguigu/.Trash/.....

4. 修改访问垃圾回收站用户名

进入垃圾回收站用户名，默认是 dr.who，修改为 atguigu 用户。

[core-site.xml]

```
<property>
    <name>hadoop.http.staticuser.user</name>
    <value>atguigu</value>
</property>
```

5. 通过程序删除的文件不会经过回收站，需要调用 moveToTrash() 才进入回收站

Trash trash = New Trash(conf);

5. 恢复回收站数据

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -mv
/usr/atguigu/.Trash/Current/user/atguigu/input
/usr/atguigu/input
```

6. 清空回收站

```
[atguigu@hadoop102 hadoop-2.7.2]$ hadoop fs -expunge
```

快照管理

快照管理

快照相当于对目录做一个备份。**并不会立即复制所有文件**，而是指向同一个文件。**当写入发生时，才会产生新文件。**

- (1) `hdfs dfsadmin -allowSnapshot 路径` (功能描述：开启指定目录的快照功能)
- (2) `hdfs dfsadmin -disallowSnapshot 路径` (功能描述：禁用指定目录的快照功能，默认是禁用)
- (3) `hdfs dfs -createSnapshot 路径` (功能描述：对目录创建快照)
- (4) `hdfs dfs -createSnapshot 路径 名称` (功能描述：指定名称创建快照)
- (5) `hdfs dfs -renameSnapshot 路径 旧名称 新名称` (功能描述：重命名快照)
- (6) `hdfs lsSnapshottableDir` (功能描述：列出当前用户所有可快照目录)
- (7) `hdfs snapshotDiff 路径1 路径2` (功能描述：比较两个快照目录的不同之处)
- (8) `hdfs dfs -deleteSnapshot <path> <snapshotName>` (功能描述：删除快照)

如果创建时不指定快照名称，系统默认会用当前时间戳作为快照名称。

在禁用之前一定要删除该目录下所有的快照。

实际操作：**

1、开启/禁用指定目录的快照功能

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs dfsadmin -allowSnapshot /java/home/  
-----  
Allowing snapshot on /java/home/ succeeded  
#禁用，在禁用之前记得要将这个快照删除才能禁用  
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs dfsadmin -disallowSnapshot /java/home/
```

2、对目录创建快照

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs dfs -createSnapshot /java/home/  
-----  
Created snapshot /java/home/.snapshot/s20181215-121522.220
```

3、查看快照

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs dfs -ls -R /java/home/.snapshot/  
-----  
lsr: DEPRECATED: Please use 'ls -R' instead.  
drwxr-xr-x - hwhadoop supergroup 0 2018-12-15 12:15  
/java/home/.snapshot/s20181215-121522.220  
-rw-r--r-- 3 hwhadoop supergroup 106210 2018-12-15 10:31  
/java/home/.snapshot/s20181215-121522.220/LICENSE.txt  
-rw-r--r-- 3 hwhadoop supergroup 14 2018-12-15 11:10  
/java/home/.snapshot/s20181215-121522.220/readme.txt  
-rw-r--r-- 3 hwhadoop supergroup 93 2018-12-15 10:29  
/java/home/.snapshot/s20181215-121522.220/safemode.sh
```

4、重命名快照

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs dfs -renameSnapshot /java/home/ s20181215-121522.220  
tarena1215
```

5、列出当前用户的所有快照列表

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs lsSnapshottableDir  
-----  
drwxr-xr-x 0 hwhadoop supergroup 0 2018-12-15 12:15 1 65536 /java/home
```

6、比较两个快照目录的不同之处

```
#可以在hdfs snapshotDiff的命令帮助  
#在帮助中可以看到如此打印:users can use "." to present the current status, and ...  
[hwhadoop@tarena101 hadoop-2.9.2]$ hdfs snapshotDiff /java/home/ . .snapshot/tarena1215  
#如果没有变化  
Difference between current directory and snapshot tarena1215 under directory  
/java/home:  
-----  
#如果有  
Difference between current directory and snapshot tarena1215 under directory /java/home:  
M .  
#这里代表快照比较之前的文件夹少了一个NOTICE.txt文件  
- ./NOTICE.txt
```

7、恢复快照

就是将快照里的文件拷贝到其他地方去，它无法像Linux的快照一样可以立即恢复到快照记录的状态。

```
hdfs dfs -cp /java/home/.snapshot/s21321-34423.027 /java/
```

MapReduce

MapReduce负责计算功能，HDFS负责存储

hadoop通过他们实现了海量数据的存储和计算。

MapReduce概述

MapReduce的定义

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。

MapReduce的优缺点

主要集中在项目选型阶段。让框架做最擅长的事情。

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。

不擅长做小文件的处理。

核心思想

需求：统计其中每一个单词出现的总次数(查询结果：a-p一个文件，q-z一个文件)

Hadoop
Spark
Hive
Hbase
Hadoop
Spark
...

Java
php
Android
Html5
Bigdata
python
...

输入数据

Map阶段



Reduce阶段

统计a-p开头的单词



统计q-z开头的单词



1) MapReduce运算程序一般需要分成2个阶段：Map阶段和Reduce阶段

2) Map阶段的并发MapTask，完全并行运行，互不相干

3) Reduce阶段的并发ReduceTask，完全互不相干，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

若干问题细节

1) MapTask如何工作

2) ReduceTask 如何工作

3) MapTask如何控制分区、排序等

4) MapTask和ReduceTask之间如何衔接

进程

一个完整的MapReduce程序在分布式运行时有三类实例进程：

1) **MrAppMaster** : 负责整个程序的过程调度及状态协调。

2) **MapTask** : 负责Map阶段的整个数据处理流程。

3) **ReduceTask** : 负责Reduce阶段的整个数据处理流程。

官方案例源码解析和数据类型

wordcount案例有Map类，Reduce类和驱动类，且数据类型是Hadoop本身封装的序列化类型

• 1.5 常用数据序列化类型

表 4-1 常用的数据类型对应的 Hadoop 数据序列化类型

Java 类型	Hadoop Writable 类型
boolean	BooleanWritable
byte	ByteWritable
int	IntWritable
float	FloatWritable
long	LongWritable
double	DoubleWritable
String	Text
map	MapWritable
array	ArrayWritable

编程规范

用户编写MapReduce分成三个部分：Mapper、Reducer、Driver

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

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

WordCount案例分析

1, 需求：

在给定的文本文件中统计输出每一个单词出现的总次数

2, 需求分析

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

1、输入数据

```
atguigu atguigu  
ss ss  
cls cls  
jiao  
banzhang  
xue  
hadoop
```

2、输出数据

atguigu	2
banzhang	1
cls	2
hadoop	1
jiao	1
ss	2
xue	1

3、Mapper

```
// 3.1 将MapTask传给我们的文本  
内容先转换成String
```

```
atguigu atguigu
```

```
// 3.2 根据空格将这一行切分成单词
```

```
atguigu  
atguigu
```

```
// 3.3 将单词输出为<单词，1>
```

```
atguigu, 1  
atguigu, 1
```

4、Reducer

```
// 4.1 汇总各个key的个数
```

```
atguigu, 1  
atguigu, 1
```

```
// 4.2 输出该key的总次数
```

```
atguigu, 2
```

5、Driver

```
// 5.1 获取配置信息，获取job对象实例
```

```
// 5.2 指定本程序的jar包所在的本地路径
```

```
// 5.3 关联Mapper/Reducer业务类
```

```
// 5.4 指定Mapper输出数据的kv类型
```

```
// 5.5 指定最终输出的数据的kv类型
```

```
// 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
```

```
// 5.8 提交作业
```

WordCount案例Mapper

环境准备：

创建maven工程

导入依赖

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.8.2</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.9.2</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-client -->
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.9.2</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-hdfs -->
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>2.9.2</version>
    </dependency>
</dependencies>
```

在资源路径中创建log4j.properties

```
log4j.rootLogger=debug, stdout, R
#输出到控制台
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %p [%t] %c.%m(%L) | %m%n

#输出DEBUG级别以上的日志到文件
log4j.appender.debug=org.apache.log4j.DailyRollingFileAppender
log4j.appender.debug.layout=org.apache.log4j.PatternLayout
log4j.appender.debug.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %p [%t] %c.%m(%L) | %m%n
log4j.appender.debug.File=./logs/debug.txt
log4j.appender.debug.DatePattern='yyyy-MM-dd'
log4j.appender.debug.Threshold=DEBUG
log4j.appender.debug.Append=true
log4j.appender.debug.Encoding=UTF-8

#输出DEBUG级别以上的日志到文件
log4j.appender.error=org.apache.log4j.DailyRollingFileAppender
log4j.appender.error.layout=org.apache.log4j.PatternLayout
log4j.appender.error.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %p [%t] %c.%m(%L) | %m%n
```

```

log4j.appender.error.File=./logs/error.txt
log4j.appender.error.DatePattern='yyyy-MM-dd
log4j.appender.error.Threshold=ERROR
log4j.appender.error.Append=true
log4j.appender.error.Encoding=UTF-8

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=firestorm.log

log4j.appender.R.MaxFileSize=100KB
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n

log4j.logger.com.codefutures=DEBUG

```

创建包和类:

```

package com.tarena.mr.mrdao;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
/***
 * map阶段
 * KEYIN: 输入数据的key
 * VALUEIN: 输入数据的value
 * KEYOUT: 输出数据的key
 * VALUEOUT: 输出数据的value
 */
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    //如果在循环中new, 将会创建很多对象, 因此提为成员变量
    private Text text = new Text();
    private IntWritable iw = new IntWritable(1);
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        //1,tarena tarena 是Text类型, 转换为String类型
        String s = value.toString();
        //2,切分单词
        String[] ws = s.split(" ");
        //3,循环写出
        for (int i = 0; i < ws.length; i++) {
            text.set(ws[i]);
            //tarena:1
            context.write(text, iw);
        }
    }
}

```

WordCount案例Reducer

编写包和java类

```

package com.tarena.mr.mrdao;

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

```

```

import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

/**
 * KEYIN:VALUEIN map阶段输出的key和value
 * KEYOUT:VALUEOUT 输出的key和value
 */
public class WordCountReduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable iw = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        //1,之前处理的tarena:1,tarena:1
        //values中会有两个1
        //-----
        //累加求和
        int sum = 0;
        for (IntWritable i:values) {
            sum += i.get();
        }
        //写出 tarena 2
        iw.set(sum);
        context.write(key,iw);
    }
}

```

WordCount案例Driver

```

package com.tarena.mr.mrdao;

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

import java.io.IOException;

public class wordCountDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(wordCountDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(wordCountMapper.class);
            job.setReducerClass(wordCountReduce.class);
            //4,设置Mapper阶段输出数据的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);
            //5,设置最终数据输出的key和value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
            //6,设置输入路径和输出路径
            FileInputFormat.setInputPaths(job,new Path(args[0]));
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
        }
    }
}

```

```

        //7, 提交job
        job.waitForCompletion(true);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

WordCount案例测试

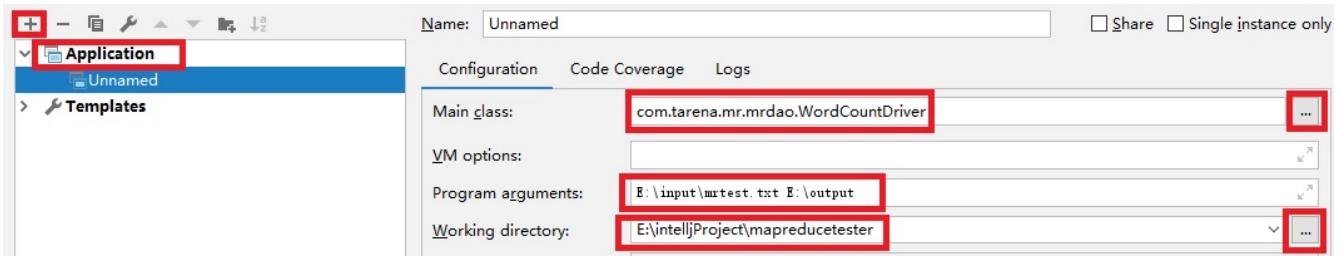
在运行前在磁盘中添加源文件， E:\input\mrtest.txt

```

abc acc
jack jackie
rose rose dick
thanks dick
lucy lily and lucy

```

在编辑工具中设置Run Configurations



在设置的output文件夹中看到输出结果

```

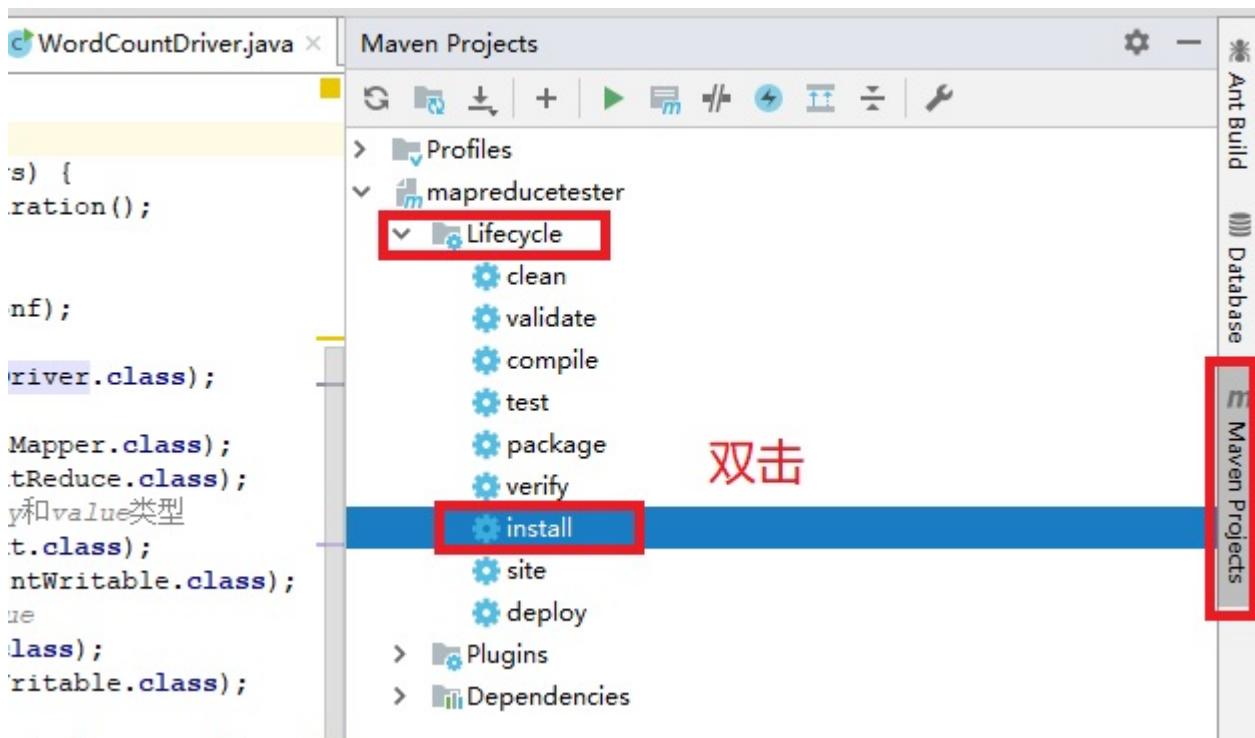
abc      1
acc      1
and      1
dick     2
jack     1
jackie   1
lily     1
lucy     2
rose     2
thanks   1

```

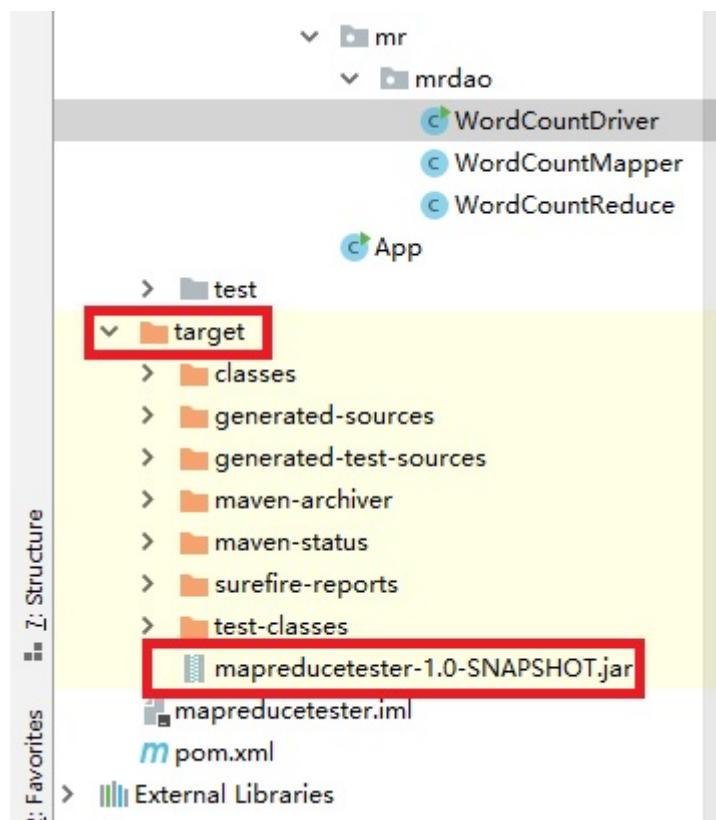
可以使用Debug调试查看执行过程， Debug时还是在Run Configurations环境下调试

WordCount案例在集群上运行

将写的wordcount项目打成jar包， 直接使用IntelliJ的插件即可



打包完成后jar包放在项目路径的target包中



使用xftp将jar包上传到集群

在集群中创建一个路径和两个文件a.txt和b.txt

```
[hwhadoop@tarena101 hadoop-2.9.2]$ hadoop fs -mkdir -p /test/mr/input/
#将两个文件上传到集群中刚才建立的input中
[hwhadoop@tarena101 hadoop-2.9.2]$ hadoop fs -put a.txt /test/mr/input/
[hwhadoop@tarena101 hadoop-2.9.2]$ hadoop fs -put b.txt /test/mr/input/
```

使用hadoop jar调用上传的jar中编写好的wordcount功能

```
#注意这里的output路径在执行之前不能存在  
[hwhadoop@tarena101 hadoop-2.9.2]$ hadoop jar mr.jar com.tarena.mr.mrdao.WordCountDriver  
/test/mr/input/ /test/mr/output/
```

完成后看见map 100% reduce 100%即告成功，去output路径下下载生成的文件查看wordcount结果

Hadoop的序列化

序列化概述

2.1.1 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**磁盘的持久化数据**，转换成内存中的对象。

2.1.2 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。**

2.1.3 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）。

Hadoop序列化特点：

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

让天下没有难学的技术

自定义bean对象实现序列化接口

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。

I

(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;  
}
```

序列化案例

需求

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。

(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

phone_data (2).txt - 记事本

1	13736230513	192.196.100.1	www.atguigu.com	2481	24681	200
2	13846544121	192.196.100.2		264	0	200
3	13956435636	192.196.100.3		132	1512	200
4	13966251146	192.168.100.1		240	0	404
5	18271575951	192.168.100.2	www.atguigu.com	1527	2106	200
6	84188413	192.168.100.3	www.atguigu.com	4116	1432	200
7	13590439668	192.168.100.4		1116	954	200
8	15910133277	192.168.100.5	www.hao123.com	3156	2936	200
9	13729199489	192.168.100.6		240	0	200
10	13630577991	192.168.100.7	www.shouhu.com	6960	690	200
11	15043685818	192.168.100.8	www.baidu.com	3659	3538	200
12	15959002129	192.168.100.9	www.atguigu.com	1938	180	500
13	13560439638	192.168.100.10		918	4938	200
14	13470253144	192.168.100.11		180	180	200
15	13682846555	192.168.100.12	www.qq.com	1938	2910	200
16	13992314666	192.168.100.13	www.gaga.com	3008	3720	200
17	13509468723	192.168.100.14	www.qinghua.com	7335	110349	404
18	18390173782	192.168.100.15	www.sogou.com	9531	2412	200
19	13975057813	192.168.100.16	www.baidu.com	11058	48243	200
20	13768778790	192.168.100.17		120	120	200
21	13568436656	192.168.100.18	www.alibaba.com	2481	24681	200
22	13568436656	192.168.100.19		1116	954	200

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
---	-------------	----------------	------	-----	-----

(2) 抽取手机号、上行流量、下行流量

13560436666	1116	954
手机号码	上行流量	下行流量

(3) 以手机号为key，bean对象为value输出，

即context.write(手机号,bean);

(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

FlowBean

```
package com.tarena.pojo;

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

public class FlowBean implements Writable {
    private long upFlow;
    private long downFlow;
    private long sumFlow;

    public FlowBean() {
    }

    public FlowBean(long upFlow, long downFlow) {
        this.upFlow = upFlow;
        this.downFlow = downFlow;
        this.sumFlow = upFlow + downFlow;
    }

    //序列化
    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeLong(upFlow);
        dataOutput.writeLong(downFlow);
        dataOutput.writeLong(sumFlow);
    }

    //反序列化
    @Override
    public void readFields(DataInput dataInput) throws IOException {
        //注意：序列化和反序列化的顺序必须一致
        upFlow = dataInput.readLong();
        downFlow = dataInput.readLong();
        sumFlow = dataInput.readLong();
    }

    //修改toString()方便后来拆分时用
}
```

```

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

public long getUpFlow() {
    return upFlow;
}

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

public long getDownFlow() {
    return downFlow;
}

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

public long getSumFlow() {
    return sumFlow;
}

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

public void addSum(long upFlow, long downFlow) {
    this.upFlow = upFlow;
    this.downFlow = downFlow;
    this.sumFlow = upFlow + downFlow;
}
}

```

Mapper

```

package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
    Text k = new Text();
    FlowBean v = new FlowBean();
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        //1,获取一行
        String s = value.toString();
        //2,切割
        String[] split = s.split("\t");
        //3,封装对象
        k.set(split[1]);
        v.setUpFlow(Long.parseLong(split[split.length-3]));
    }
}

```

```

        v.setDownFlow(Long.parseLong(split[split.length-2]));
        //4,写出
        context.write(k,v);
    }
}

```

Reducer

```

package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
    FlowBean v = new FlowBean();
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Context context)
        throws IOException, InterruptedException {
        long sumUp = 0;
        long sumDown = 0;
        //1,求和
        for (FlowBean bean:values) {
            sumUp += bean.getUpFlow();
            sumDown += bean.getDownFlow();
        }
        v.addSum(sumUp,sumDown);
        //2,写出
        context.write(key,v);
    }
}

```

Driver

```

package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class FlowCountDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\input","F:\\teacher\\output"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(FlowCountDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(FlowCountMapper.class);

```

```

        job.setReducerClass(FlowCountReducer.class);
        //4,设置Mapper阶段输出数据的key和value类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);
        //5,设置最终数据输出的key和value
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);
        //6,设置输入路径和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //7, 提交job
        job.waitForCompletion(true);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

}

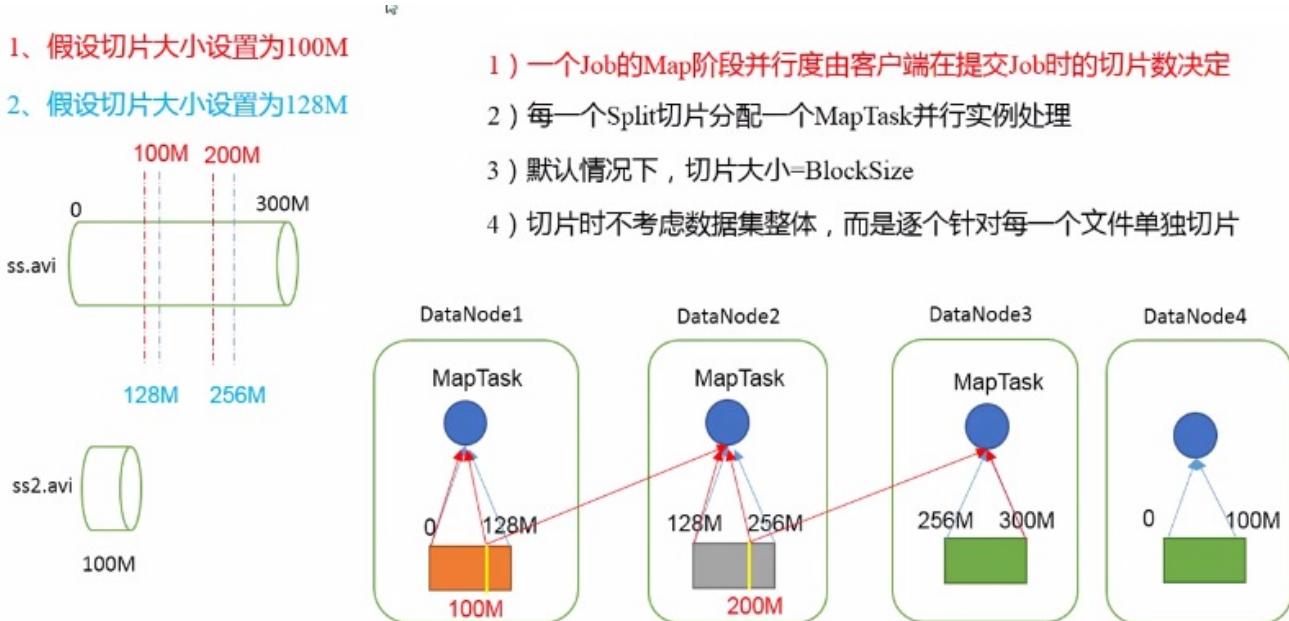
```

Debug调试

MapReduce框架原理

InputFormat数据输入

切片与MapTask并行度决定机制



Job提交流程源码解析(retry)

```

waitForCompletion()
submit();
// 1 建立连接
connect();

```

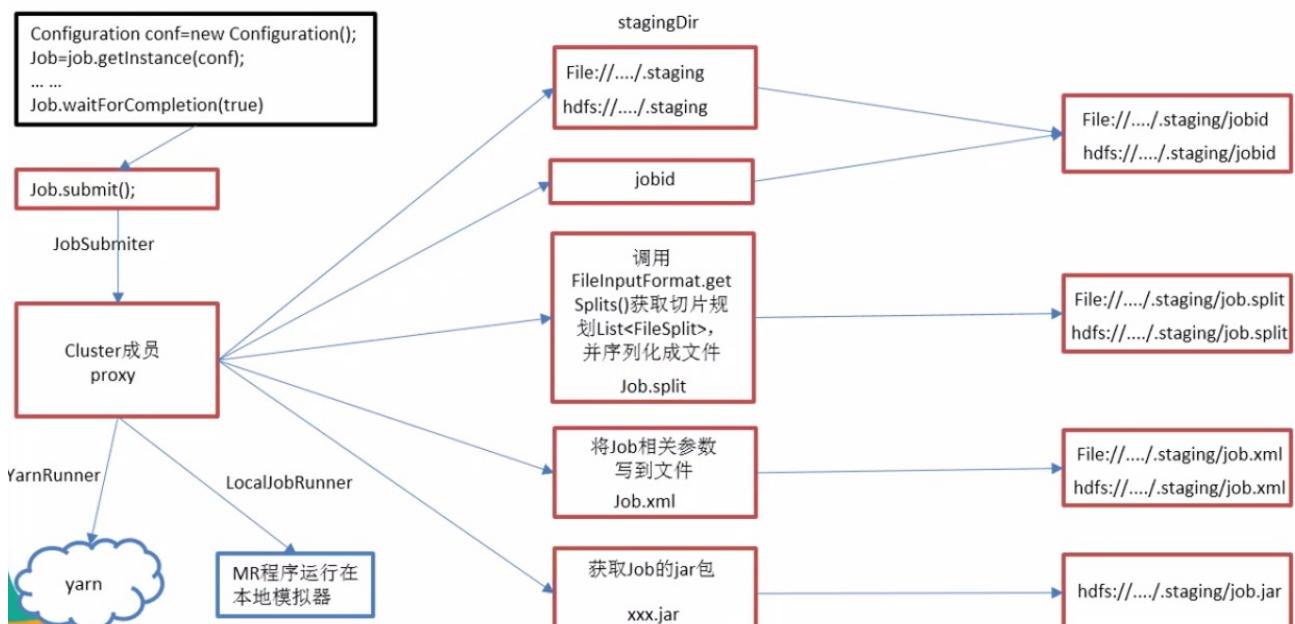
```

// 1) 创建提交 job 的代理
new Cluster(getConfiguration());
// (1) 判断是本地 yarn 还是远程
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(), job.getCredentials());

```

Job提交流程源码解析



Job切片机制源码解析

FileInputFormat切片源码解析

- (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



`computeSliteSize(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。

FileInputFormat切片机制和配置参数

FileInputFormat切片机制

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：

file1.txt 320M
file2.txt 10M

(2) 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

FileInputFormat切片大小的参数配置

(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();
```

CombineTextInputFormat理论

框架默认的TextInputFormat切片机制是对任务按文件规划切片，不管文件大小，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

- 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

- 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文件大小情况来设置。

- 切片机制

生成切片过程包括：虚拟存储过程和切片过程二部分

setMaxInputSplitSize值为4M

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

让天下没有难学的技术

CombineTextInputFormat案例

- 需求

将输入的大量小文件合并成一个切片统一处理

输入数据：

准备4个小文件

期望：

期望一个切片处理4个文件

- 实现过程

不做任何处理运行刚才上面的wordCount案例，观察切片为4个

- 在WordCountDriver中添加如下代码，运行，观察切片个数减少

```
package com.tarena.mr.mrdao;

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.CombineFileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.CombineTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import java.io.IOException;

public class WordCountDriver {
    public static void main(String[] args) {
        args = new String[]{"E:\\input","E:\\output"};
        Configuration conf = new Configuration();
        try {
```

```

//1,获取job对象
Job job = Job.getInstance(conf);
//2,设置jar存储位置
job.setJarByClass(WordCountDriver.class);
//3,关联Map和Reduce类
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReduce.class);
//4,设置Mapper阶段输出数据的key和value类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
//5,设置最终数据输出的key和value
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

//如果不设置InputFormat, 它默认使用TextInputFormat.cladd
job.setInputFormatClass(CombineTextInputFormat.class);
//虚拟存储切片最大值设置20m
CombineTextInputFormat.setMaxInputsplitsize(job,20971520);

//6,设置输入路径和输出路径
FileInputFormat.setInputPaths(job,new Path(args[0]));
FileOutputFormat.setOutputPath(job,new Path(args[1]));
//7, 提交job
job.waitForCompletion(true);
} catch (Exception e) {
e.printStackTrace();
}
}

}

//打印日志中观察到值有1个切片
//INFO - number of splits: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。

TextInputFormat实现类

1. 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)  
(19, Intelligent learning engine)  
(47, Learning more convenient)  
(72, From the real demand for more close to the enterprise)
```

⋮

可以在之前的wordcount案例的Mapper类中加入代码打印key来验证

```
public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean> {  
    Text k = new Text();  
    FlowBean v = new FlowBean();  
    @Override  
    protected void map(LongWritable key, Text value, Context context) throws IOException,  
    InterruptedException {  
        //验证key  
        System.out.println(key.toString());  
        //1,获取一行  
        String s = value.toString();  
        //2,切割  
        String[] split = s.split("\t");  
        //3,封装对象  
        k.set(split[1]);  
        vsetUpFlow(Long.parseLong(split[split.length-3]));  
        v.setDownFlow(Long.parseLong(split[split.length-2]));  
        //4,写出  
        context.write(k, v);  
    }  
}
```

KeyValueTextInputFormat案例分析

2. KeyValueTextInputFormat

每一行均为一条记录，被分隔符分割为 key , value 。可以通过在驱动类中设置 conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, "\t"); 来设定分隔符。默认分隔符是 tab (\t) 。

以下是一个示例，输入是一个包含4条记录的分片。其中——>表示一个（水平方向的）制表符。

```
line1 —>Rich learning form  
line2 —>Intelligent learning engine  
line3 —>Learning more convenient  
line4 —>From the real demand for more close to the enterprise
```

每条记录表示为以下键/值对：

```
(line1,Rich learning form)  
(line2,Intelligent learning engine)  
(line3,Learning more convenient)  
(line4,From the real demand for more close to the enterprise)
```

此时的键是每行排在制表符之前的Text序列。

江苏大学图书馆

需求

统计输入文件每一行的第一个单词相同的行数

输入数据

```
hadoop ni hao  
java good morning  
hadoop o hai yo  
java kon ni ti wa
```

期望结果数据

```
hadoop 2  
java 2
```

1、需求：统计输入文件中每一行的第一个单词相同的行数。

2、输入数据

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

4、Map阶段

banzhang ni hao
(1) 设置key和value
<banzhang,1>

(2) 写出

5、Reduce阶段

<banzhang,1>
<banzhang,1>
(1) 汇总
<banzhang,2>

3、期望输出数据

```
banzhang 2  
xihuan 2
```

6、Driver

```
// (1) 设置切割符  
conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, " ");  
  
// (2) 设置输入格式  
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

KeyValueTextInputFormat案例实现

Mapper

```
package com.tarena.kvtester;

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

import java.io.IOException;

public class KVTextMapper extends Mapper<Text,Text,Text,IntWritable> {
    IntWritable v = new IntWritable(1);
    @Override
    protected void map(Text key, Text value, Context context) throws IOException,
    InterruptedException {
        //写出
        context.write(key,v);
    }
}
```

Reducer

```
package com.tarena.kvtester;

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

import java.io.IOException;

public class KVTextReducer extends Reducer<Text, IntWritable, Text,IntWritable> {
    IntWritable v = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
    IOException, InterruptedException {
        int sum = 0;
        //累加求和
        for (IntWritable i:values) {
            sum += i.get();
        }
        //封装IntWritable对象
        v.set(sum);
        //写出
        context.write(key,v);
    }
}
```

Driver

```
package com.tarena.kvtester;

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.input.KeyValueLineRecordReader;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
import java.io.IOException;

public class KVTextDriver {
    public static void main(String[] args) {
        args = new String[]{"F:/teacher/kvinput","F:/teacher/kvoutput"};
        Configuration conf = new Configuration();
        //设置读取文件时以什么为间隔进行切分
        conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR," ");
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储路径
            job.setJarByClass(KVTextDriver.class);
            //3,关联mapper和reducer
            job.setMapperClass(KVTextMapper.class);
            job.setReducerClass(KVTextReducer.class);
            //4,设置mapper输出的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);
            //5,设置最终输出的key和value类型
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

            job.setInputFormatClass(KeyValueTextInputFormat.class);

            //6,设置输出路径
            FileInputFormat.setInputPaths(job,new Path(args[0]));
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
            //7,提交job
            job.waitForCompletion(true);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

结果:

```
hadoop 2 java 2
```

NLineInputFormat案例分析

如果使用 `NlineInputFormat`，代表每个 map 进程处理的 `InputSplit` 不再按 Block 块去划分，而是按 `NlineInputFormat` 指定的行数 N 来划分。即输入文件的总行数 / N = 切片数，如果不整除，切片数 = 商 + 1。

以下是一个示例，仍然以上面的4行输入为例。

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

例如，如果N是2，则每个输入分片包含两行。开启2个MapTask。

(0, Rich learning form)
(19, Intelligent learning engine)

另一个 mapper 则收到后两行：

(47, Learning more convenient)
(72, From the real demand for more close to the enterprise)

这里的键和值与TextInputFormat生成的一样。

1、需求：对每个单词进行个数统计，要求每三行放入一个切片中。

2、输入数据

4、Map阶段

(1) 获取一行

(2) 切割

(3) 循环写出

5. Reduce阶段

(1) 汇总

(2) 输出

3. 期望输出数据

```
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - Hadoop command-line  
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - No job jar file set  
INFO [org.apache.hadoop.mapreduce.lib.input.FileInputFormat] - Total input p  
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:4  
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - Submitting tokens for job:  
INFO [org.apache.hadoop.mapreduce.Job] - The url to track the job: http://lo  
INFO [org.apache.hadoop.mapreduce.Job] - Running job: job_local1998538859_000  
INFO [org.apache.hadoop.mapred.LocalJobRunner] - OutputCommitter set in conf  
INFO [org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter] - File Out
```

6. Driver

```
// 设置每个切片InputSplit中划分三条记录  
NLineInputFormat.setNumLinesPerSplit(job, 3);
```

NLineInputFormat案例实现

Mapper

```
package com.tarena.nltester;

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

import java.io.IOException;

public class NLTextMapper 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 {
```

```

        //将输入的文本转为String类型
        String s = value.toString();
        //将文本拆分
        String[] s1 = s.split(" ");
        //便利拆分结果，封装键值对
        for (String str:s1) {
            k.set(str);
            context.write(k,v);
        }
    }
}

```

Reducer

```

package com.tarena.nltester;

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

import java.io.IOException;

public class NLTextReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    IntWritable v = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        //累加求和
        int sum = 0;
        for (IntWritable iw:values) {
            sum += iw.get();
        }
        v.set(sum);
        //写出
        context.write(key,v);
    }
}

```

Driver

```

package com.tarena.nltester;

import com.tarena.kvtester.KVTextDriver;
import com.tarena.kvtester.KVTextMapper;
import com.tarena.kvtester.KVTextReducer;
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.input.KeyValueLineRecordReader;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class NLTextDriver {
    public static void main(String[] args) {

```

```
args = new String[]{"F:/teacher/nlinput","F:/teacher/nloutput"};
Configuration conf = new Configuration();
//设置读取文件时以什么为间隔进行切分
try {
    //1,获取job对象
    Job job = Job.getInstance(conf);

    //设置多少行为1个切片
    NLineInputFormat.setNumLinesPerSplit(job,3);
    //设置使用NLineInputFormat处理记录
    job.setInputFormatClass(NLineInputFormat.class);

    //2,设置jar存储路径
    job.setJarByClass(NLTextDriver.class);
    //3,关联mapper和reducer
    job.setMapperClass(NLTextMapper.class);
    job.setReducerClass(NLTextReducer.class);
    //4,设置mapper输出的key和value类型
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    //5,设置最终输出的key和value类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    //6,设置输出路径
    FileInputFormat.setInputPaths(job,new Path(args[0]));
    FileOutputFormat.setOutputPath(job,new Path(args[1]));
    //7,提交job
    job.waitForCompletion(true);
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
```

结果：

INFO - number of splits:4

自定义InputFormat步骤

在企业开发中，Hadoop框架自带的InputFormat类型不能满足所有应用场景，需要自定义InputFormat来解决实际问题。

自定义InputFormat步骤如下：

(1) 自定义一个类继承FileInputFormat。

(2) 改写RecordReader，实现一次读取一个完整文件封装为KV。

(3) 在输出时使用SequenceFileOutPutFormat输出合并文件。

案例：将小文件合并为一个大文件

1、自定义一个类继承FileInputFormat

(1) 重写isSplitable()方法，返回false不可切割

(2) 重写createRecordReader()，创建自定义的RecordReader对象，并初始化

2、改写RecordReader，实现一次读取一个完整文件封装为KV

(1) 采用IO流一次读取一个文件输出到value中，因为设置了不可切片，最终把所有文件都封装到了value中

(2) 获取文件路径信息+名称，并设置key

3、设置Driver

```
// (1) 设置输入的inputFormat  
job.setInputFormatClass(WholeFileInputformat.class);
```

```
// (2) 设置输出的outputFormat  
job.setOutputFormatClass(SequenceFileOutputFormat.class);
```

让天下没有难懂的技术

程序实现：

WholeRecordReader

```
package com.tarena.inputcustom;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.FSDataInputStream;  
import org.apache.hadoop.fs.FileSystem;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.BytesWritable;  
import org.apache.hadoop.io.IOUtils;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.InputSplit;  
import org.apache.hadoop.mapreduce.RecordReader;  
import org.apache.hadoop.mapreduce.TaskAttemptContext;  
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
```

```
import java.io.IOException;

public class WholeRecordReader extends RecordReader<Text, BytesWritable> {
    Filesplit split;
    Configuration conf;
    Text k = new Text();
    BytesWritable v = new BytesWritable();
    boolean isProgress = true;
    @Override
    public void initialize(InputSplit inputsplit, TaskAttemptContext context) throws
IOException, InterruptedException {
        //初始化
        this.split = (FileSplit) inputsplit;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if(isProgress) {
            //核心业务逻辑
            byte[] buf = new byte[(int) split.getLength()];
            //1,获取fs对象
            Path path = split.getPath();
            FileSystem fileSystem = path.getFileSystem(conf);
            //2,获取输入流
            FSDataInputStream open = fileSystem.open(path);
            //3,拷贝
            IOUtils.readFully(open, buf, 0, buf.length);
            //4,封装v
            v.set(buf, 0, buf.length);
            //5,封装k
            k.set(path.toString());
            //6,关闭资源
            IOUtils.closeStream(fileSystem);
            isProgress=false;
            return true;
        }
        return false;
    }

    @Override
    public Text getCurrentKey() throws IOException, InterruptedException {
        //获取当前的key
        return k;
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException, InterruptedException {
        //获取当前的value
        return v;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {
        return 0;
    }

    @Override
    public void close() throws IOException {
```

```
        //关闭资源
    }
}
```

WholeFileInputFormat

```
package com.tarena.inputcustom;

import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import java.io.IOException;

public class wholeFileInputFormat extends FileInputFormat<Text, BytesWritable> {

    @Override
    public RecordReader<Text, BytesWritable> createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException, InterruptedException {
        WholeRecordReader reader = new WholeRecordReader();
        reader.initialize(split,context);
        return reader;
    }
}
```

SequenceFileMapper

```
package com.tarena.inputcustom;

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

import java.io.IOException;

public class SequenceFileMapper extends Mapper<Text, BytesWritable, Text, BytesWritable> {
    @Override
    protected void map(Text key, BytesWritable value, Context context) throws IOException,
InterruptedException {
        context.write(key,value);
    }
}
```

SequenceFileReducer

```
package com.tarena.inputcustom;

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

import java.io.IOException;

public class SequenceFileReducer extends Reducer<Text, BytesWritable, Text, BytesWritable> {
    @Override
    protected void reduce(Text key, Iterable<BytesWritable> values, Context context) throws
IOException, InterruptedException {
        //1,循环写出
    }
}
```

```

        for (BytesWritable bw:values) {
            context.write(key,bw);
        }
    }
}

SequenceFileDriver

package com.tarena.inputcustom;

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

import java.io.IOException;

public class SequenceFileDriver {
    public static void main(String[] args) {
        args = new String[]{"F:/teacher/cstinput","F:/teacher/cstoutput"};
        Configuration conf = new Configuration();
        //设置读取文件时以什么为间隔进行切分
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);

            //2,设置jar存储路径
            job.setJarByClass(SequenceFileDriver.class);
            //3,关联mapper和reducer
            job.setMapperClass(SequenceFileMapper.class);
            job.setReducerClass(SequenceFileReducer.class);

            //7, 设置输入的inputFormat
            job.setInputFormatClass(WholeFileInputFormat.class);
            //8, 设置输出的outputFormat
            job.setOutputFormatClass(SequenceFileOutputFormat.class);

            //4,设置mapper输出的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(BytesWritable.class);
            //5,设置最终输出的key和value类型
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(BytesWritable.class);

            //6,设置输出路径
            FileInputFormat.setInputPaths(job,new Path(args[0]));
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
            //7,提交job
            job.waitForCompletion(true);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

}

自定义InputFormat案例Debug

3个文件，3个切片，3个mapTask

分别先后执行nextKeyValue()方法中的核心逻辑，形成key和value，并返回true

mapper接收到key和value做map阶段相关操作

循环上述步骤知道3个mapTask全部执行完成

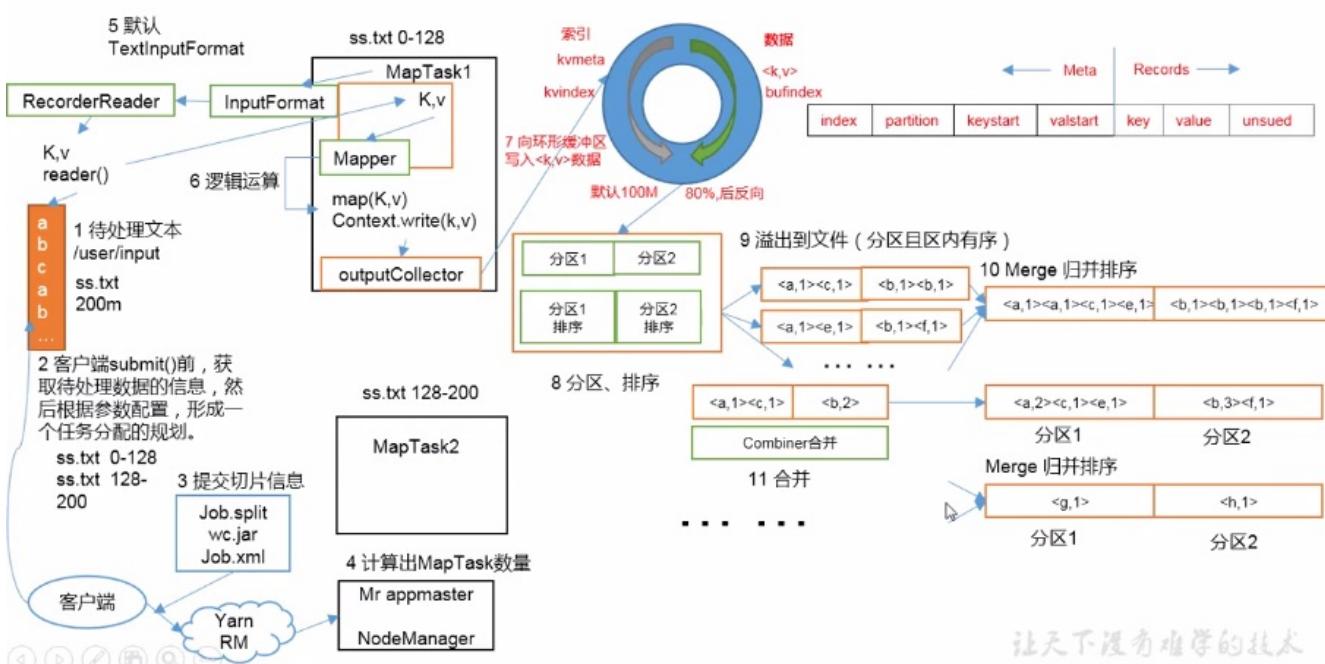
Reducer将3个mapTask执行的结果做最终汇总并输出。

InputFormat实现类总结

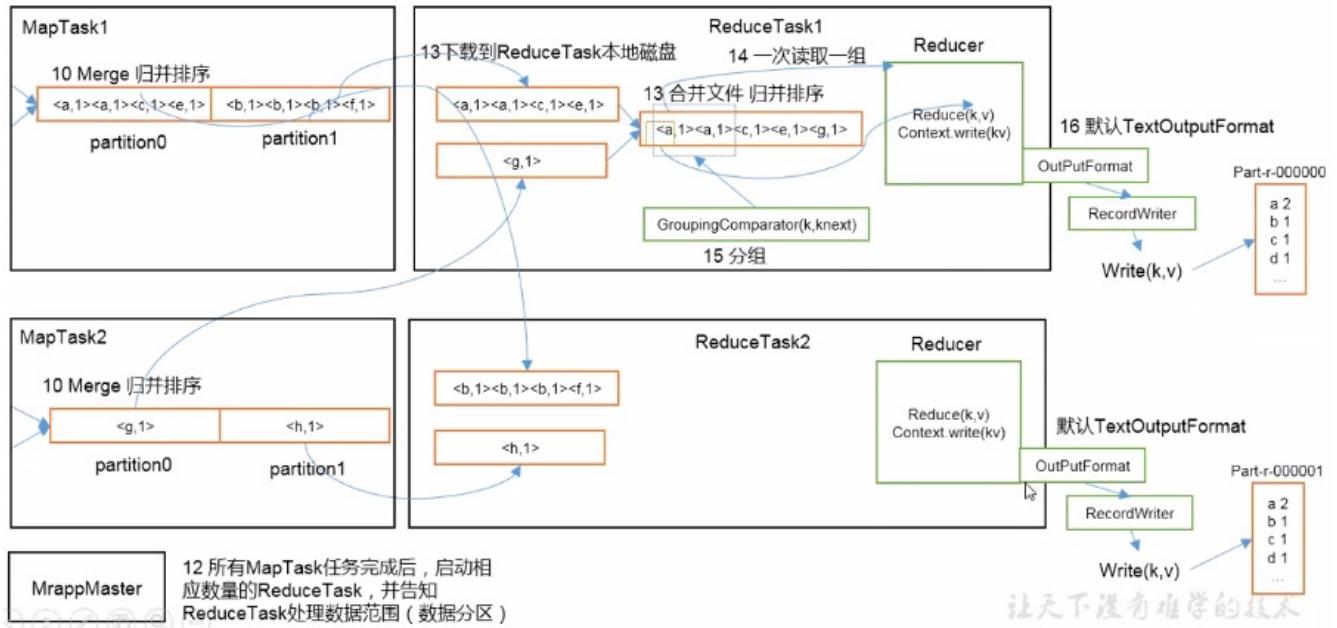
切片和K,V之间的区别

工作流程 (面试重点)

MAP阶段



Reduce阶段



Shuffle机制（面试重点）

Map方法之后，Reduce方法之前的数据处理过程称之为shuffle，后来说

分区

淘宝统计各种手机的销量。

医院统计各个科室的挂号量。

交通运管部门统计出行峰口客流量。

电力部门统计每个时段的用电总量。

HashPartition默认分区

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。

Partition分区案例

因为在Map之后Reduce之前，所以Partitioner的泛型应该是Map的输出泛型。

分区从0开始，不能间隔

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);
```



1、需求：将统计结果按照手机号归属地不同省份输出到不同文件中（分区）

2、数据输入

13630577991	6960	690	文件1
13736230513	2481	24681	文件2
13846544121	264	0	文件3
13956435636	132	1512	文件4
13560439638	918	4938	文件5

3、期望数据输出



4、增加一个ProvincePartitioner分区

136	分区0
137	分区1
138	分区2
139	分区3
其他	分区4

5、Driver驱动类

```
// 指定自定义数据分区  
job.setPartitionerClass(ProvincePartitioner.  
class);  
  
// 同时指定相应数量的reduceTask  
job.setNumReduceTasks(5);
```

让天下没有难学的技术

代码实现：

ProvincePartitioner

```
package com.tarena.flow;  
  
import com.tarena.pojo.FlowBean;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Partitioner;  
  
public class ProvincePartitioner extends Partitioner<Text, FlowBean> {  
    @Override  
    public int getPartition(Text text, FlowBean flowBean, int i) {  
  
        int partition = 4;  
        //获取电话号码中的前三位  
        String s = text.toString();  
        String phone = s.substring(0, 3);  
        switch (phone){  
            case "138":  
                partition = 0;  
                break;  
            case "136":  
                partition = 1;  
                break;  
            case "139":  
                partition = 2;  
                break;  
            case "131":  
                partition = 3;  
                break;  
        }  
        return partition;  
    }  
}
```

FlowCountMapper

```
package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
    Text k = new Text();
    FlowBean v = new FlowBean();
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        //验证key
        System.out.println(key.toString());
        //1,获取一行
        String s = value.toString();
        //2,切割
        String[] split = s.split("\t");
        //3,封装对象
        k.set(split[1]);
        v.setUpFlow(Long.parseLong(split[split.length-3]));
        v.setDownFlow(Long.parseLong(split[split.length-2]));
        //4,写出
        context.write(k,v);
    }
}
```

FlowCountReducer

```
package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
    FlowBean v = new FlowBean();
    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Context context)
        throws IOException, InterruptedException {
        long sumUp = 0;
        long sumDown = 0;
        //1, 求和
        for (FlowBean bean:values) {
            sumUp += bean.getUpFlow();
            sumDown += bean.getDownFlow();
        }
        v.addSum(sumUp,sumDown);

        //2,写出
        context.write(key,v);
    }
}
```

FlowCountDriver

```
package com.tarena.flow;

import com.tarena.pojo.FlowBean;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class FlowCountDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\input","F:\\teacher\\output"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(FlowCountDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(FlowCountMapper.class);
            job.setReducerClass(FlowCountReducer.class);
            //4,设置Mapper阶段输出数据的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(FlowBean.class);
            //5,设置最终数据输出的key和value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(FlowBean.class);

            //设置分区
            job.setPartitionerClass(ProvincePartitioner.class);
            //设置分区数量
            job.setNumReduceTasks(5);

            //6,设置输入路径和输出路径
            FileInputFormat.setInputPaths(job,new Path(args[0]));
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
            //7, 提交job
            job.waitForCompletion(true);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

结果会在output中看到生成4个文件，分别对应4种手机号码

Partition分区案例总结

```
//如果不设置它，设置再多的分区都没有意义，永远都是一个进程在处理所有的分区  
//如果设置1，即便是有5个分区，还是只有1个结果文件输出  
//如果设置2，程序会出现混乱，不知道输出目的地应该选谁，会报IO异常  
//如果设置6，会产生6个结果文件输出，超出分区的那个结果文件中什么数据都没有。  
job.setNumReduceTasks(5);
```

4、分区总结

- (1) 如果ReduceTask的数量> getPartition的结果数，则会多产生几个空的输出文件part-r-000xx；
- (2) 如果1<ReduceTask的数量<getPartition的结果数，则有一部分分区数据无处安放，会Exception；
- (3) 如果ReduceTask的数量=1，则不管MapTask端输出多少个分区文件，最终结果都交给这一个ReduceTask，最终也就只会产生一个结果文件 part-r-00000；
- (4) 分区号必须从零开始，逐一累加。

5、案例分析

例如：假设自定义分区数为5，则

- (1) job.setNumReduceTasks(1); 会正常运行，只不过会产生一个输出文件
- (2) job.setNumReduceTasks(2); 会报错
- (3) job.setNumReduceTasks(6); 大于5，程序会正常运行，会产生空文件

让天下没有难学的技术

排序

排序概述

排序是MapReduce框架中最重要的操作之一。

MapTask和ReduceTask均会对数据按照key进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。

默认排序是按照字典顺序排序，且实现该排序的方法是快速排序。

对于MapTask，它会将处理的结果暂时放到环形缓冲区中，当环形缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次快速排序，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行归并排序。

对于ReduceTask，它从每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写磁盘上，否则存储在内存中。如果磁盘上文件数目达到一定阈值，则进行一次归并排序以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据溢写到磁盘上。当所有数据拷贝完毕后，ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序。

排序分类

(1) 部分排序

MapReduce根据输入记录的键对数据集排序。保证输出的每个文件内部有序。

(2) 全排序

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

(3) 辅助排序：(GroupingComparator分组)

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

(4) 二次排序

在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

全排序案例分析

1、需求：根据手机的总流量进行倒序排序

2、输入数据

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

3、输出数据

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

4、FlowBean实现WritableComparable接口重写compareTo方法

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

6、Reducer类

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

5、Mapper类

```
context.write(bean, 手机号)
```

注意不设置此逻辑

全排序案例FlowBean

```
package com.tarena.sort;  
  
import org.apache.hadoop.io.WritableComparable;  
  
import java.io.DataInput;  
import java.io.DataOutput;  
import java.io.IOException;  
  
public class SortBean implements WritableComparable<SortBean> {  
  
    private long upFlow;  
    private long downFlow;  
    private long sumFlow;  
  
    public SortBean() {}  
  
    public SortBean(long upFlow, long downFlow) {  
        this.upFlow = upFlow;  
        this.downFlow = downFlow;  
        this.sumFlow = upFlow + downFlow;  
    }  
  
    @Override  
    public String toString() {  
        return upFlow + "\t" + downFlow + "\t" + sumFlow;  
    }  
  
    @Override  
    public int compareTo(SortBean o) {  
        int result = 0;  
        //排序的核心逻辑  
        if(sumFlow > o.getSumFlow()){  
            result = -1;  
        }else if(sumFlow < o.getSumFlow()){  
            result = 1;  
        }  
        return result;  
    }  
}
```

```

        }else{
            result = 0;
        }
        return result;
    }

@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 {
    upFlow = dataInput.readLong();
    downFlow = dataInput.readLong();
    sumFlow = dataInput.readLong();
}

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;
}
}

```

全排序案例Mapper

```

package com.tarena.sort;

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

import java.io.IOException;

public class SortCountMapper extends Mapper<LongWritable, Text, SortBean, Text> {
    private Text v = new Text();
    private SortBean k = new SortBean();
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,

```

```

InterruptedException {
    //获取一行数据
    String line = value.toString();
    //对数据进行切割
    String[] split = line.split("\t");
    //装配数据
    v.set(split[0]);
    //虽然值需要总流量，但因为在Bean中设计了其他两个属性，如果在这里不复制则为null
    //null在序列化的时候会报异常，因此属性在map阶段都必须有值且不为null
    k.setUpFlow(Long.parseLong(split[1]));
    k.setDownFlow(Long.parseLong(split[2]));
    k.setSumFlow(Long.parseLong(split[3]));

    //写出
    context.write(k,v);
}
}

```

全排序案例实现及测试

Reduce

```

package com.tarena.sort;

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

import java.io.IOException;

public class SortCountReduce extends Reducer<SortBean, Text, Text, SortBean> {
    @Override
    protected void reduce(SortBean key, Iterable<Text> values, Context context) throws
    IOException, InterruptedException {
        for (Text text: values) {
            //这里注意，因为最终输入应该是 电话号码:流量，因此应该将电话号码作为key
            context.write(text,key);
        }
    }
}

```

Driver

```

package com.tarena.sort;

import com.tarena.flow.FlowCountDriver;
import com.tarena.flow.FlowCountMapper;
import com.tarena.flow.FlowCountReducer;
import com.tarena.flow.ProvincePartitioner;
import com.tarena.pojo.FlowBean;
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 SortCountDriver {
    public static void main(String[] args) {

```

```
Configuration conf = new Configuration();
args = new String[]{"F:\\teacher\\output","F:\\teacher\\output1"};
try {
    //1,获取job对象
    Job job = Job.getInstance(conf);
    //2,设置jar存储位置
    job.setJarByClass(SortCountDriver.class);
    //3,关联Map和Reduce类
    job.setMapperClass(SortCountMapper.class);
    job.setReducerClass(SortCountReduce.class);
    //4,设置Mapper阶段输出数据的key和value类型
    job.setMapOutputKeyClass(SortBean.class);
    job.setMapOutputValueClass(Text.class);
    //5,设置最终数据输出的key和value
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FlowBean.class);

    //6,设置输入路径和输出路径
    FileInputFormat.setInputPaths(job,new Path(args[0]));
    FileOutputFormat.setOutputPath(job,new Path(args[1]));
    //7, 提交job
    job.waitForCompletion(true);
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
```

区内排序案例分析

1、数据输入

13509468723	7335	110349	1176
13975057813	11058	48243	5930
13568436656	3597	25635	2923
13736230513	2481	24681	2716
18390173782	9531	2412	1194
13630577991	6960	690	7650
15043685818	3659	3538	7197
13992314666	3008	3720	6728
15910133277	3156	2936	6092
13560439638	918	4938	5856
84188413	4116	1432	5548
13682846555	1938	2910	4848
18271575951	1527	2106	3633
15959002129	1938	180	2118
13590439668	1116	954	2070
13956435636	132	1512	1644
13470253144	180	180	360
13846544121	264	0	264
13966251146	240	0	240
13768778790	120	120	240
13729199489	240	0	240

2、期望数据输出

<input type="checkbox"/>	part-r-00000	13630577991	6960	690	7650
		13682846555	1938	2910	4848
<input type="checkbox"/>	part-r-00001	13736230513	2481	24681	27162
		13768778790	120	120	240
		13729199489	240	0	240
<input type="checkbox"/>	part-r-00002	13846544121	264	0	264
<input type="checkbox"/>	part-r-00003	13975057813	11058	48243	59301
		13992314666	3008	3720	6728
		13956435636	132	1512	1644
		13966251146	240	0	240
<input type="checkbox"/>	part-r-00004	13509468723	7335	110349	117684
		13568436656	3597	25635	29232
		18390173782	9531	2412	11943
		15043685818	3659	3538	7197
		15910133277	3156	2936	6092

Partitioner

```
package com.tarena.sort;  
  
import org.apache.hadoop.io.Text;
```

```

import org.apache.hadoop.mapreduce.Partitioner;

public class ProvinceSortPartitioner extends Partitioner<SortBean, Text> {
    @Override
    public int getPartition(SortBean sortBean, Text text, int i) {
        String s = text.toString();
        String phone = s.substring(0,3);
        int p = 4;
        switch (phone){
            case "139":
                p = 0;
                break;
            case "138":
                p = 1;
                break;
            case "136":
                p = 2;
                break;
            case "137":
                p = 3;
                break;
        }
        return p;
    }
}

```

Driver

```

package com.tarena.sort;

import com.tarena.flow.FlowCountDriver;
import com.tarena.flow.FlowCountMapper;
import com.tarena.flow.FlowCountReducer;
import com.tarena.flow.ProvincePartitioner;
import com.tarena.pojo.FlowBean;
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 SortCountDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\output","F:\\teacher\\output1"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(SortCountDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(SortCountMapper.class);
            job.setReducerClass(SortCountReduce.class);
            //4,设置Mapper阶段输出数据的key和value类型
            job.setMapOutputKeyClass(SortBean.class);
            job.setMapOutputValueClass(Text.class);
        }
    }
}

```

```

//5,设置最终数据输出的key和value
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);

job.setPartitionerClass(ProvinceSortPartitioner.class);
job.setNumReduceTasks(5);

//6,设置输入路径和输出路径
FileInputFormat.setInputPaths(job,new Path(args[0]));
FileOutputFormat.setOutputPath(job,new Path(args[1]));
//7, 提交job
job.waitForCompletion(true);
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
}
}

```

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

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

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

Reducer

$(3+5+7+2+6)/5=23/5$ 不等于 $(5+4)/2=9/2$

Combiner合并案例

combiner

```

package com.tarena.mr.mrdao;

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> {
    IntWritable iw = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
        int sum = 0;
        for (IntWritable i:values) {
            sum += i.get();
        }
        iw.set(sum);
        context.write(key,iw);
    }
}

```

Driver:

```

package com.tarena.mr.mrdao;

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

import java.io.IOException;

public class WordCountDriver {
    public static void main(String[] args) {
        args = new String[]{"E:\\\\input","E:\\\\output"};
        Configuration conf = new Configuration();
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(WordCountDriver.class);
            //3,关联Mapper和Reduce类
            job.setMapperClass(WordCountMapper.class);
            job.setReducerClass(WordCountReduce.class);
            //4,设置Mapper阶段输出数据的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);
            //5,设置最终数据输出的key和value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
                //设置combiner类
            job.setCombinerClass(WordCountCombiner.class);

            FileInputFormat.setInputPaths(job,new Path(args[0]));
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
            //7, 提交job
            job.waitForCompletion(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

分组

分组排序案例分析



需求：求每个订单中最贵的商品（GroupingComparator）



1、输入数据

0000001	Pdt_01	222.8
0000002	Pdt_05	722.4
0000001	Pdt_02	33.8
0000003	Pdt_06	232.8
0000003	Pdt_02	33.8
0000002	Pdt_03	522.8
0000002	Pdt_04	122.4

2、预期输出数据

0000001	222.8
0000002	722.4
0000003	232.8

3、MapTask

1) Map中处理的事情

- (1) 获取一行
- (2) 切割出每个字段
- (3) 一行封装成bean对象

订单id 价格
bean1, nullwritable 0000001 222.8
bean2, nullwritable 0000002 722.4
bean3, nullwritable 0000001 33.8
bean4, nullwritable 0000003 232.8
bean5, nullwritable 0000003 33.8
bean6, nullwritable 0000002 522.8
bean7, nullwritable 0000002 122.4

2) 二次排序

先根据订单id排序，如果订单id相同再根据价格降序排序

0000001 222.8
0000001 33.8
0000002 722.4
0000002 522.8
0000002 122.4
0000003 232.8
0000003 33.8

4、ReduceTask

1) 辅助排序

对从map端拉取过来的数据再次进行排序，只要订单id相同就认为是相同key

2) Reduce方法只把一组key的第一个写出去

第一次调用Reduce方法
0000001
222.8
33.8

0000001 222.8

让天下没有难学的技术

分组排序案例OrderBean

```
package com.tarena.order;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;

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

public class OrderBean implements WritableComparable<OrderBean> {

    private int id;
    private double price;

    public OrderBean() {
    }

    public OrderBean(int id, double price) {
        this.id = id;
        this.price = price;
    }

    @Override
    public String toString() {
        return id + "\t" + price;
    }

    @Override
    public int compareTo(OrderBean o) {
```

```

        int result;
        if(id > o.getId()){
            result = 1;
        }else if(id < o.getId()){
            result = -1;
        }else{
            if(price > o.getPrice()){
                result = -1;
            }else if(price < o.getPrice()){
                result = 1;
            }else{
                result = 0;
            }
        }
        return result;
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeInt(id);
        dataOutput.writeDouble(price);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        id = dataInput.readInt();
        price = dataInput.readDouble();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}

```

分组排序案例Mapper

```

package com.tarena.order;

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 class OrderSortMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable> {
    OrderBean ob = new OrderBean();

```

```

@Override
protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
    //1,获取一条数据
    String s = value.toString();
    //2,拆分数据
    String[] s1 = s.split(" ");
    //3,装载数据
    ob.setId(Integer.parseInt(s1[0]));
    ob.setPrice(Double.parseDouble(s1[2]));

    //4,写出
    context.write(ob, NullWritable.get());
}
}

```

分组排序案例Reducer

```

package com.tarena.order;

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

import java.io.IOException;

public class OrderSortReduce extends Reducer<OrderBean, NullWritable, OrderBean, NullWritable>
{
    @Override
    protected void reduce(OrderBean key, Iterable<NullWritable> values, Context context)
throws IOException, InterruptedException {
        context.write(key, NullWritable.get());
    }
}

```

分组排序案例Driver

```

package com.tarena.order;

import com.tarena.pojo.FlowBean;
import com.tarena.sort.*;
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 OrderSortDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\odininput", "F:\\teacher\\odoutput"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置

```

```

        job.setJarByClass(OrderSortDriver.class);
        //3,关联Map和Reduce类
        job.setMapperClass(OrderSortMapper.class);
        job.setReducerClass(OrderSortReduce.class);
        //4,设置Mapper阶段输出数据的key和value类型
        job.setMapOutputKeyClass(OrderBean.class);
        job.setMapOutputValueClass(NullWritable.class);
        //5,设置最终数据输出的key和value
        job.setOutputKeyClass(OrderBean.class);
        job.setOutputValueClass(NullWritable.class);
        //设置分组排序类
        job.setGroupingComparatorClass(GroupingOrderSortComparator.class);

        //6,设置输入路径和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //7, 提交job
        job.waitForCompletion(true);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

分组排序案例排序类

```

package com.tarena.order;

import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class GroupingOrderSortComparator extends WritableComparator {
    protected GroupingOrderSortComparator(){
        super(OrderBean.class,true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        int result;
        OrderBean aob = (OrderBean) a;
        OrderBean bob = (OrderBean) b;
        if(aob.getId() > bob.getId()){
            result = 1;
        }else if(aob.getId() < bob.getId()){
            result = -1;
        }else{
            result = 0;
        }
        return result;
    }
}

```

分组排序案例调试

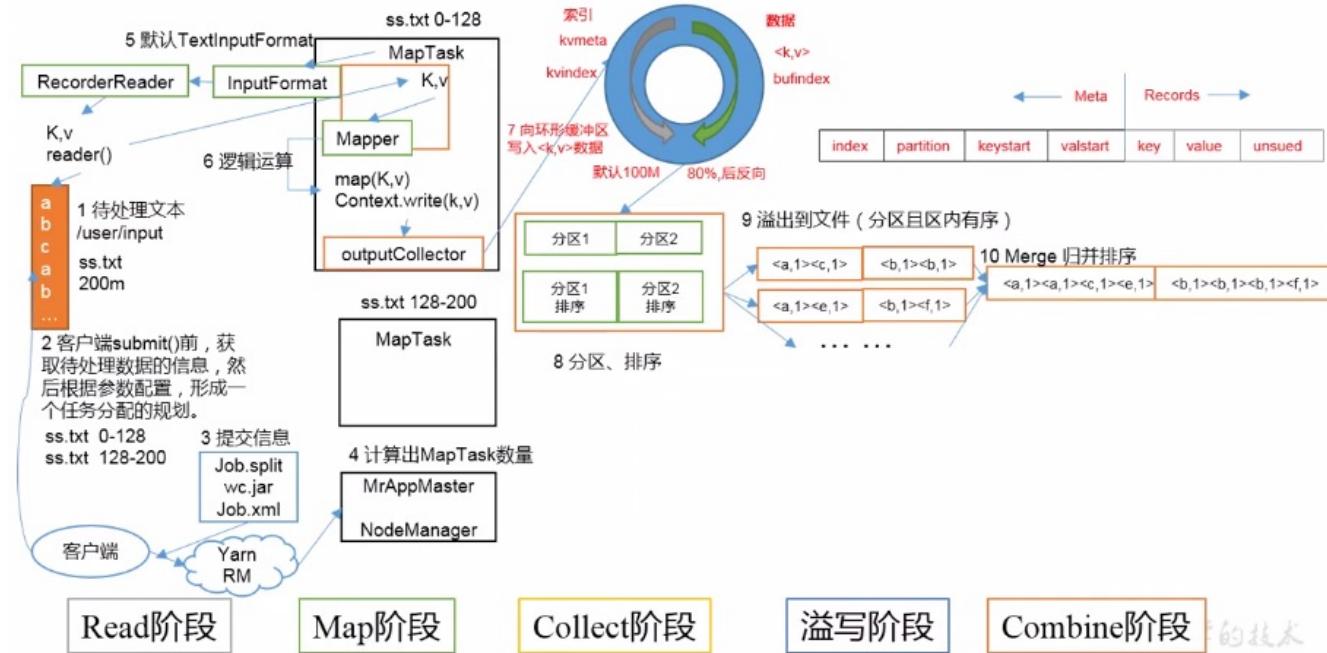
如果在GroupingOrderSortComparator类中没有构造方法，则在运行时会报空指针异常。因为没有传true，源码中会将所有属性置空。

分组排序案例扩展

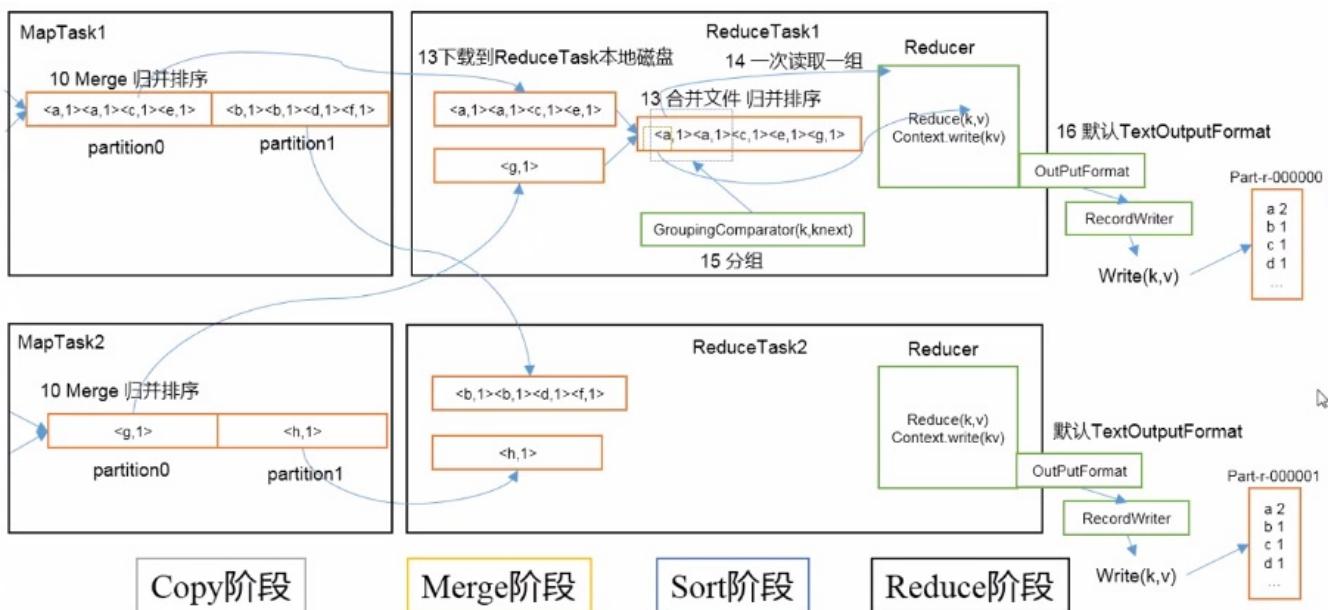
输出TOP3，只需修改Reduce类的reducer方法

```
for (NullWritable nw : values) {  
    context.write(key, NullWritable.get());  
}
```

MapTask工作机制（面试重点）



ReduceTask工作机制（面试重点）



ReduceTask个数设置

Reduce Task的并行度同样影响整个job的执行并发度和并发效率，但与MapTask的并发度由切片数决定不同，ReduceTask数量的决定是可以直接手动设置的。

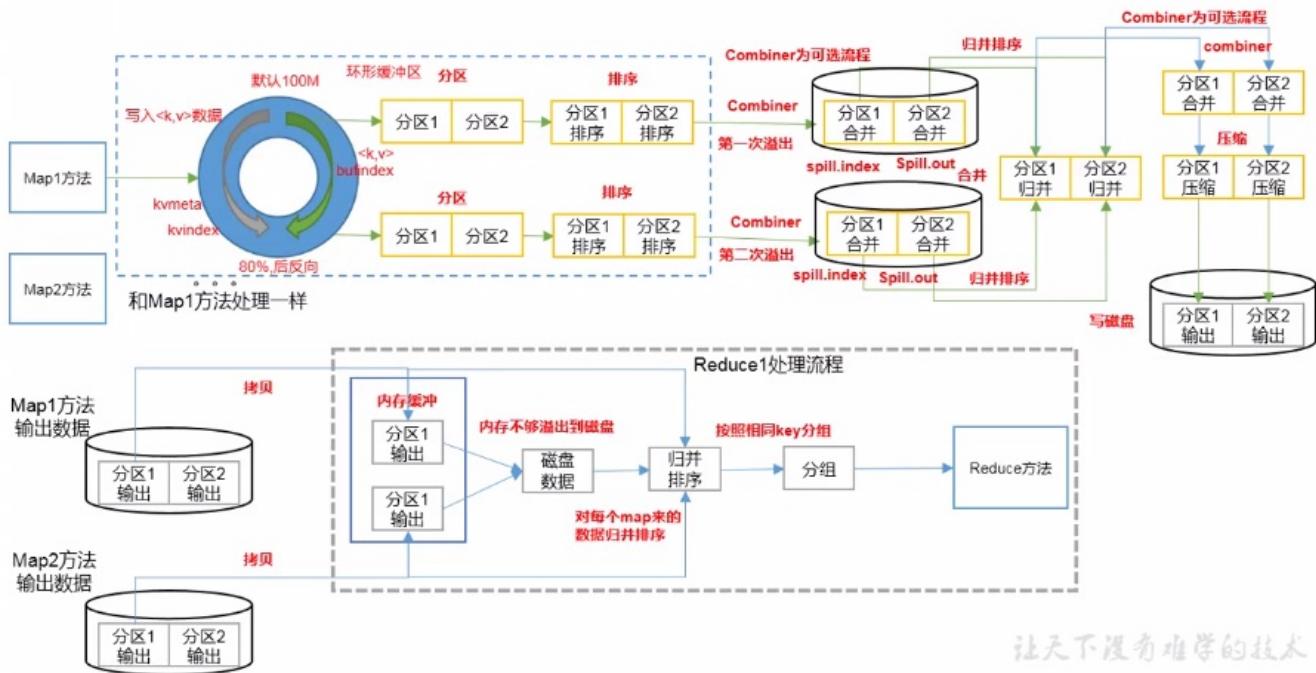
```
job.setNumReduceTasks(4);
```

测试ReduceTask设置为多少比较合适

- (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。

Shuffle机制（面试重点）

Map之后，Reduce之前的数据处理过程称之为Shuffle



OutputFormat数据输出

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. 文本输出TextOutputFormat

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

2. SequenceFileOutputFormat

将SequenceFileOutputFormat输出作为后续 MapReduce任务的输入，这便是一种好的输出格式，因为它的**格式紧凑，很容易被压缩**。

+

3. 自定义OutputFormat

根据用户需求，自定义实现输出。

让天下没有难懂的技术

自定义OutputFormat案例分析

1. 使用场景

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

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

2. 自定义OutputFormat步骤

(1) 自定义一个类继承FileOutputFormat。

(2) 改写RecordWriter，具体改写输出数据的方法write()。

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FilterRecordWriter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，输出到otherOut流

3、输出数据

 atguigu.log	http://www.atguigu.com
 other.log	http://cn.bing.com http://www.baidu.com http://www.google.com http://www.sin2a.com http://www.sin2desa.co m 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(FilterOutputFormat.class);
```

让天下没有难懂的技术

自定义OutputFormat案例实现

Mapper

```
package com.tarena.output;

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 class OuputMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        context.write(value, NullWritable.get());
    }
}
```

Reduce

```
package com.tarena.output;

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

import java.io.IOException;

public class OutputReduce extends Reducer<Text, NullWritable, Text, NullWritable> {
    Text k = new Text();
    @Override
    protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws
    IOException, InterruptedException {
        String s = key.toString();
        s += "\r\n";
    }
}
```

```
        k.set(s);
        context.write(k,NullWritable.get());
    }
}
```

FilterOutputFormat

```
package com.tarena.output;

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.lib.output.FileOutputFormat;

import java.io.IOException;

public class FilterOutputFormat extends FileOutputFormat<Text, NullWritable> {
    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext taskAttemptContext) throws IOException, InterruptedException {
        return new FilterRecordWriter(taskAttemptContext);
    }
}
```

FilterRecordWriter

```
package com.tarena.output;

import org.apache.hadoop.conf.Configuration;
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.InputSplit;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.lib.output.FileOutputFormat;

import java.io.IOException;

public class FilterRecordWriter extends RecordWriter<Text, NullWritable> {

    FSDataOutputStream fsDataOutputStream;
    FSDataOutputStream fsDataOutputStream1;

    public FilterRecordWriter(TaskAttemptContext job){
        try {
            //创建Filesystem
            FileSystem fs = FileSystem.get(job.getConfiguration());
            //创建输出tedu的流
            fsDataOutputStream = fs.create(new Path("f:/teacher/opoutput/tedu.txt"));
            //创建输出其他的流
            fsDataOutputStream1 = fs.create(new Path("f:/teacher/opoutput/other.txt"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void write(Text key, NullWritable value) throws IOException {
        fsDataOutputStream.write(key.toString().getBytes());
        fsDataOutputStream1.write(key.toString().getBytes());
    }

    public void close() throws IOException {
        fsDataOutputStream.close();
        fsDataOutputStream1.close();
    }
}
```

```

@Override
public void write(Text text, Nullwritable nullwritable) throws IOException,
InterruptedException {
    if(text.toString().contains("tedu")){
        fsDataOutputStream.write(text.toString().getBytes());
    }else{
        fsDataOutputStream1.write(text.toString().getBytes());
    }
}

@Override
public void close(TaskAttemptContext taskAttemptContext) throws IOException,
InterruptedException {
    IOutils.closeStream(fsDataOutputStream);
    IOutils.closeStream(fsDataOutputStream1);
}
}

```

Driver

```

package com.tarena.output;

import com.tarena.order.*;
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 OutputDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\opinput","F:\\teacher\\opoutput"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(OutputDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(OututMapper.class);
            job.setReducerClass(OutputReduce.class);
            //4,设置Mapper阶段输出数据的key和value类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(Nullwritable.class);
            //5,设置最终数据输出的key和value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Nullwritable.class);

            //要将自定义的输出格式组建设置到job中
            job.setOutputFormatClass(FilterOutputFormat.class);

            //6,设置输入路径和输出路径
            FileInputFormat.setInputPaths(job,new Path(args[0]));
            //虽然这里已经设置了job.setOutputFormatClass(FilterOutputFormat.class);
            //但成功时fileoutputFormat需要输出一个SUCCESS文件, 所以必须制定其输出路径。
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
        }
    }
}

```

```

        //7, 提交job
        job.waitForCompletion(true);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

JOIN的应用

ReduceJoin理论

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。

ReduceJoin案例分析

1、输入数据

order.txt

订单id	pid	数量
1001	01	1
1002	02	2
1003	03	3
1001	01	1
1002	02	2
1003	03	3

pd.txt

Pid	产品名称
01	小米
02	华为
03	格力

2、预期输出数据

订单id	产品名称	数量
1001	小米	1
1001	小米	1
1002	华为	2
1002	华为	2
1003	格力	3
1003	格力	3

3、Map Task

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	1001	1	order
02	1002	2	order
03	1003	3	order
01	小米		pd
02	华为		pd
03	格力		pd

2) 默认对产品id排序

01	1001	1	order
01	1001	1	order
01	小米		pd
02	1002	2	order
02	1002	2	order
02	华为		pd
03	1003	3	order
03	1003	3	order
03	格力		pd

4、ReduceTask

1) Reduce方法缓存订单数据集合，和产品表，然后合并

订单id	产品名称	数量
1001	小米	1
1001	小米	1
1002	华为	2
1002	华为	2
1003	格力	3
1003	格力	3

天下没有难学的技术

ReduceJoin案例TableBean

```

package com.tarena.join;

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;
    private String pid;
    private int amount;
    private String pname;
    private String flag;

    public TableBean() {
    }

    public TableBean(String id, String pid, int amount, String pname, String flag) {
        this.id = id;
        this.pid = pid;
        this.amount = amount;
        this.pname = pname;
        this.flag = flag;
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        dataOutput.writeUTF(id);
        dataOutput.writeUTF(pid);
        dataOutput.writeInt(amount);
        dataOutput.writeUTF(pname);
        dataOutput.writeUTF(flag);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        id = dataInput.readUTF();
        pid = dataInput.readUTF();
        amount = dataInput.readInt();
        pname = dataInput.readUTF();
        flag = dataInput.readUTF();
    }

    @Override
    public String toString() {
        return id + '\t' + amount + "\t" + pname;
    }
}

```

ReduceJoin案例Mapper

```

package com.tarena.join;

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 JoinMapper extends Mapper<LongWritable, Text, Text, TableBean> {
    private String name;
    private TableBean bean = new TableBean();
    private Text k = new Text();
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        FileSplit inputSplit = (FileSplit) context.getInputSplit();
        //利用切片反推出文件名
        name = inputSplit.getPath().getName();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        if(name.startsWith("order")){
            String[] split = value.toString().split("\t");
            bean.setId(split[0]);
            bean.setPid(split[1]);
            bean.setAmount(Integer.parseInt(split[2]));
            //如果没有数据不能不设置，会报空之指针，可以设置为""
            bean.setPname("");
            //来自不同的文件设置不同的标记
            bean.setFlag("order");
            //key设置为pid
            k.set(split[1]);
        }else{
            String[] split = value.toString().split("/t");
            bean.setId("");
            bean.setPid(split[0]);
            bean.setAmount(0);
            bean.setPname(split[1]);
            bean.setFlag("product");
            k.set(split[0]);
        }
        context.write(k, bean);
    }
}

```

ReduceJoin案例Reduce

```

package com.tarena.join;

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

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

public class JoinReduce extends Reducer<Text, TableBean, TableBean, NullWritable> {
    @Override
    protected void reduce(Text key, Iterable<TableBean> values, Context context) throws
    IOException, InterruptedException {

```

```

List<TableBean> list = new ArrayList<>();
TableBean ptb = new TableBean();

for (TableBean bean:values) {

    if("order".equals(bean.getFlag())){
        TableBean tb = new TableBean();
        try {

            BeanUtils.copyProperties(tb,bean);
            list.add(tb);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }else{
        try {
            BeanUtils.copyProperties(ptb,bean);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}

for (TableBean o:list) {
    o.setPname(ptb.getPname());
    context.write(o,Nullwritable.get());
}
}
}

```

ReduceJoin案例Driver

```

package com.tarena.join;

import com.tarena.flow.FlowCountDriver;
import com.tarena.flow.FlowCountMapper;
import com.tarena.flow.FlowCountReducer;
import com.tarena.pojo.FlowBean;
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 JoinDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\joininput","F:\\teacher\\joutput"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(JoinDriver.class);

```

```
//3,关联Map和Reduce类
job.setMapperClass(JoinMapper.class);
job.setReducerClass(JoinReduce.class);
//4,设置Mapper阶段输出数据的key和value类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(TableBean.class);
//5,设置最终数据输出的key和value
job.setOutputKeyClass(TableBean.class);
job.setOutputValueClass(NullWritable.class);

FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
//7, 提交job
job.waitForCompletion(true);
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
```

ReduceJoin案例Debug和总结

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

MapJoin的应用

适用于有一张小表和一张大表的场景。

在Reduce端处理过多的表，非常容易产生数据倾斜

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端数据压力，减少数据倾斜斜

采用DistributedCache

在Mapper的setup阶段，将文件读取到缓存集合中

在驱动函数中加载缓存，

```
job.addCacheFile(new URI("file://e:/cache/pd.txt"));
```

MapJoin案例分析

1) DistributedCacheDriver 缓存文件

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

2) 读取缓存的文件数据

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

MapJoin案例缓存文件处理

```
package com.tarena.mapjoin;  
  
import com.tarena.join.JoinDriver;  
import com.tarena.join.JoinMapper;  
import com.tarena.join.JoinReduce;  
import com.tarena.join.TableBean;  
import org.apache.commons.lang.ObjectUtils;  
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) {  
        Configuration conf = new Configuration();  
        args = new String[]{"F:\\teacher\\joininput\\order.txt", "F:\\teacher\\joutput"};  
        try {  
            //1,获取job对象  
            Job job = Job.getInstance(conf);  
            //2,设置jar存储位置  
            job.setJarByClass(MapJoinDriver.class);  
            //3,关联Map和Reduce类  
            job.setMapperClass(MapJoinMapper.class);  
  
            //5,设置最终数据输出的key和value  
            job.setOutputKeyClass(TableBean.class);  
            job.setOutputValueClass(NullWritable.class);  
  
            //加载到缓存  
            job.addCacheFile(new URI("file:///F:/teacher/joinput/product.txt"));  
            //取消reduce阶段
```

```

        job.setNumReduceTasks(0);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //7, 提交job
        job.waitForCompletion(true);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
}
}
}

```

MapJoin案例测试

```

package com.tarena.mapjoin;

import org.apache.commons.lang.StringUtils;
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.FileInputStream;
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> map = new HashMap<>();
    private Text tv = new Text();
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        URI[] cacheFiles = context.getCacheFiles();
        String s = cacheFiles[0].getPath().toString();

        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new
FileInputStream(s), "UTF-8"));

        String line;
        while(StringUtils.isNotEmpty(line = bufferedReader.readLine())){
            String[] split = line.split("\t");
            map.put(split[0],split[1]);
        }

        IOUtils.closeStream(bufferedReader);
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {

```

```

        String s = value.toString();
        String[] split = s.split("\t");
        String pid = split[1];
        String s1 = map.get(pid);
        String keystr = s + "\t" + s1;
        tv.set(keystr);
        context.write(tv, NullWritable.get());
    }
}

```

计数器应用

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

1. 计数器API

(1) 采用枚举的方式统计计数

```

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

```

(2) 采用计数器组、计数器名称的方式统计

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

组名和计数器名称随便起，但最好有意义。

(3) 计数结果在程序运行后的控制台上查看。

2. 计数器案例实操

让天下没有难学的机器

数据清洗案例

在运行核心业务MapReduce程序之前，往往要先对数据进行清洗，清理掉不符合用户需求的数据，清理程序往往只需要运行Mapper程序，不需要运行reduce程序。

需求：去除日志中字段长度小于等于11的日志

Mapper

```

package com.tarena.counter;

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 class WashMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        //1.获取一行
        String s = value.toString();

```

```

//2,解析数据
boolean result = parseLog(s,context);
if(!result){
    return;
}
context.write(value,Nullwritable.get());
}

private boolean parseLog(String s, Context context) {
    boolean result = false;
    if(s.length()<11){
        result = false;
        context.getCounter("map","faul").increment(1);
    }else{
        result = true;
        context.getCounter("map","success").increment(1);
    }
    return result;
}
}

```

Driver

```

package com.tarena.counter;

import com.tarena.join.TableBean;
import com.tarena.mapjoin.MapJoinDriver;
import com.tarena.mapjoin.MapJoinMapper;
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 WashDriver {
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        args = new String[]{"F:\\teacher\\washinput","F:\\teacher\\washoutput"};
        try {
            //1,获取job对象
            Job job = Job.getInstance(conf);
            //2,设置jar存储位置
            job.setJarByClass(WashDriver.class);
            //3,关联Map和Reduce类
            job.setMapperClass(WashMapper.class);

            //5,设置最终数据输出的key和value
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(NullWritable.class);

            //取消reduce阶段
            job.setNumReduceTasks(0);

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

```

```

        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //7. 提交job
        job.waitForCompletion(true);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

MapReduce开发总结

1. 输入数据接口：InputFormat

(1) 默认使用的实现类是：TextInputFormat

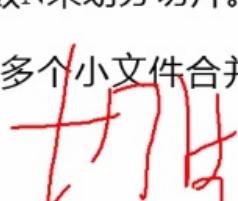
(2) TextInputFormat的功能逻辑是：一次读一行文本，然后将该行的起始偏移量作为key，行内容作为value返回。

(3) KeyValueTextInputFormat每一行均为一条记录，被分隔符分割为key，value。默认分隔符是tab (\t)。

(4) NlineInputFormat按照指定的行数N来划分切片。

(5) CombineTextInputFormat可以把多个小文件合并成一个切片处理，提高处理效率。

(6) 用户还可以自定义InputFormat。

让天下没有难学的技术

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 . Reduce端分组 : GroupingComparator

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。

7 . 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()

8. 输出数据接口 : OutputFormat

(1) 默认实现类是TextOutputFormat , 功能逻辑是 : 将每一个KV对 , 向目标文本文件输出一行。

(2) 将SequenceFileOutputFormat输出作为后续 MapReduce任务的输入 , 这便是一种好的输出格式 , 因为它的格式紧凑 , 很容易被压缩。

(3) 用户还可以自定义OutputFormat。

Hadoop数据压缩

概述

压缩技术能够有效减少底层存储系统（HDFS）读写字节数。压缩提高了网络带宽和磁盘空间的效率。在运行MR程序时，I/O操作、网络数据传输、Shuffle和Merge要花大量的时间，尤其是**数据规模很大和工作负载密集的情况下**，因此，**使用数据压缩显得非常重要。**

鉴于磁盘I/O和网络带宽是Hadoop的宝贵资源，**数据压缩对于节省资源、最小化磁盘I/O和网络传输非常有帮助。可以在任意MapReduce阶段启用压缩。**不过，尽管压缩与解压操作的CPU开销不高，其性能的提升和资源的节省并非没有代价。

压缩是提高Hadoop运行效率的一种**优化策略**。

通过对Mapper、Reducer运行过程的数据进行压缩，以减少磁盘IO，提高MR程序运行速度。

注意：采用压缩技术减少了磁盘IO，但同时增加了CPU运算负担。所以，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。

压缩基本原则：

(1) 运算密集型的job，少用压缩



(2) IO密集型的job，多用压缩

MR支持的压缩编码

表 4-7

压缩格式	hadoop 自带？	算法	文件扩展名	是否可切分	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。
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式。
Snappy	否，需要安装	Snappy	.snappy	否	和文本处理一样，不需要修改。

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
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较

表 4-9

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
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

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

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.

压缩方式选择

优点：压缩率比较高，而且压缩/解压速度也比较快；Hadoop本身支持，在应用中处理Gzip格式的文件就和直接处理文本一样；大部分Linux系统都自带Gzip命令，使用方便。

缺点：不支持Split。

应用场景：当每个文件压缩之后在130M以内的（1个块大小内），都可以考虑用Gzip压缩格式。例如说一天或者一个小时的日志压缩成一个Gzip文件。

Bzip2压缩

优点：支持Split；具有很高的压缩率，比Gzip压缩率都高；Hadoop本身自带，使用方便。

缺点：压缩/解压速度慢。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持Split，而且兼容之前的应用程序的情况。

优点：压缩/解压速度也比较快，合理的压缩率；支持Split，是Hadoop中最流行的压缩格式；可以在Linux系统下安装lzop命令，使用方便。

缺点：压缩率比Gzip要低一些；Hadoop本身不支持，需要安装；在应用中对Lzo格式的文件需要做一些特殊处理（为了支持Split需要建索引，还需要指定InputFormat为Lzo格式）。

应用场景：**一个很大的文本文件，压缩之后还大于200M以上的可以考虑，而且单个文件越大，Lzo优点越越明显。**

Snappy压缩

优点：高速压缩速度和合理的压缩率。

缺点：不支持Split；压缩率比Gzip要低；Hadoop本身不支持，需要安装。

应用场景：**当MapReduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个MapReduce作业的输出和另外一个MapReduce作业的输入。**

位置选择

**输入端采用压缩**

在有大量数据并计划重复处理的情况下，应该考虑对输入进行压缩。然而，你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。否则，Hadoop就不会使用任何编解码器。

Mapper输出采用压缩

当Map任务输出的中间数据量很大时，应考虑在此阶段采用压缩技术。这能显著改善内部数据Shuffle过程，而Shuffle过程在Hadoop处理过程中是资源消耗最多的环节。如果发现数据量大造成网络传输缓慢，应该考虑使用压缩技术。可用于压缩Mapper输出的快速编解码器包括LZO或者Snappy。

Reducer输出采用压缩

在此阶段启用压缩技术能够减少要存储的数据量，因此降低所需的磁盘空间。当MapReduce作业形成作业链条时，因为第二个作业的输入也已压缩，所以启用压缩同样有效。
↳

参数设置

压缩案例

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 (在 core-site.xml 中配置)	org.apache.hadoop.io.compress.DefultCodec, org.apache.hadoop.io.compress.GzipCodec,	输入压缩	Hadoop 使用文件扩展名判断

	org.apache.hadoop.io.compress.BZip2Codec		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.compress.DefultCodec	mapper 输出	使用 LZO 或 snappy 编解码器在此阶段压缩数据

mapreduce.output.fileoutputformat.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.fileoutputformat.compress.codec(在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	reducer 输出	使用标准工具或者编解码器，如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type (在 mapred-site.xml 中配置)	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型：NONE 和 BLOC K

CompressionCodec有两个方法可以用于轻松地压缩或解压缩数据。

要想对正在被写入一个输出流的数据进行**压缩**，我们可以使用 `createOutputStream(OutputStreamout)`方法创建一个 `CompressionOutputStream`，将其以压缩格式写入底层的流。

相反，要想对从输入流读取而来的数据进行**解压缩**，则调用 `createInputStream(InputStreamin)`函数，从而获得一个 `CompressionInputStream`，从而从底层的流读取未压缩的数据。

测试一下如下压缩方式：

DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec

```
package com.tarena.zip;
import org.apache.hadoop.conf.Configuration;
```

```

import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionOutputStream;
import org.apache.hadoop.util.ReflectionUtils;

import java.io.*;

public class zipTester {
    public static void main(String[] args) {
        //
compress("F:\\teacher\\nlinput\\kvin.txt","org.apache.hadoop.io.compress.GzipCodec");
        //
compress("F:\\teacher\\nlinput\\kvin.txt","org.apache.hadoop.io.compress.BZip2Codec");

compress("F:\\teacher\\nlinput\\kvin.txt","org.apache.hadoop.io.compress.DefaultCodec");
    }
    public static void compress(String file,String method){
        try {
            //获取输入流
            FileInputStream in = new FileInputStream(new File(file));
            //获取方法对应的类对象
            Class aclass = Class.forName(method);
            //获取压缩对象
            CompressionCodec o = (CompressionCodec) Reflectionutils.newInstance(aclass,new Configuration());
            //获取输出流
            FileOutputStream out = new FileOutputStream(new File(file +
o.getDefaultExtension()));
            //转换为具有压缩功能的输出流
            CompressionOutputStream cos = o.createOutputStream(out);
            //流的对拷
            IOUtils.copyBytes(in,cos,1024*1024,false);
            //关闭流
            IOUtils.closeStream(cos);
            IOUtils.closeStream(out);
            IOUtils.closeStream(in);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

解压缩案例

```

public class zipTester {
    public static void main(String[] args) {
        decompress("F:\\teacher\\nlinput\\kvin.txt.bzz");
    }

    private static void decompress(String s) {
        CompressionCodecFactory ccf = new CompressionCodecFactory(new Configuration());
        //获取当前压缩的方式
        CompressionCodec codec = ccf.getCodec(new Path(s));
        //检查是否支持此方式
        if(codec == null){

```

```

        return;
    }
    try {
        //获取输入流
        FileInputStream in = new FileInputStream(new File(s));
        CompressionInputStream cis = codec.createInputStream(in);
        //获取输出流
        FileOutputStream out = new FileOutputStream(new File(s + ".decode"));
        //流的对拷
        IOUtils.copyBytes(cis,out,1024*1024,false);
        //关闭流
        IOUtils.closeStream(cis);
        IOUtils.closeStream(in);
        IOUtils.closeStream(out);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

Map和Reduce启用压缩案例

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任务中间结果输出做压缩，因为他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，只需要设置两个属性即可：

以wordCount案例为基础，修改驱动类

```

package com.tarena.mr.mrdao;

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.DefaultCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.CombineTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class WordCountDriver {
    public static void main(String[] args) {
        args = new String[]{"E:\\\\input","E:\\\\output"};
        Configuration conf = new Configuration();
        try {
            //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);

            //1,获取job对象
            Job job = Job.getInstance(conf);

```

```
//2,设置jar存储位置
job.setJarByClass(wordCountDriver.class);
//3,关联Map和Reduce类
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReduce.class);
//4,设置Mapper阶段输出数据的key和value类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
//5,设置最终数据输出的key和value
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阶段输出压缩开启
FileOutputFormat.setCompressOutput(job, true);
//设置压缩方式
//FileOutputFormat.setOutputCompressorClass(job, Bzip2Codec.class);
FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
//FileOutputFormat.setOutputCompressorClass(job, DefaultCodec.class);

//7, 提交job
job.waitForCompletion(true);
} catch (Exception e) {
e.printStackTrace();
}
}
```