目录

[MapReduce 5](#_Toc14583)

[第1章 MapReduce概述 5](#_Toc2884)

[1.1 MapReduce定义 5](#_Toc24631)

[1.2 MapReduce优缺点 5](#_Toc4670)

[1.3 MapReduce核心思想 5](#_Toc27700)

[1.4 MapReduce进程 6](#_Toc17947)

[1.5 官方WordCount源码 6](#_Toc6462)

[1.6 常用数据序列化类型 6](#_Toc29888)

[1.7 MapReduce编程规范 7](#_Toc15659)

[1.8 WordCount案例实操 7](#_Toc1546)

[第2章 Hadoop序列化 11](#_Toc27709)

[2.1 序列化概述 11](#_Toc16803)

[2.2 自定义bean对象实现序列化接口（Writable） 12](#_Toc2591)

[2.3 序列化案例实操 12](#_Toc29841)

[第3章 MapReduce框架原理 16](#_Toc20415)

[3.1 InputFormat数据输入 16](#_Toc31767)

[3.2 MapReduce工作流程 31](#_Toc21783)

[3.3 Shuffle机制 33](#_Toc10476)

[3.4 MapTask工作机制 48](#_Toc26147)

[3.5 ReduceTask工作机制 49](#_Toc17612)

[3.6 OutputFormat数据输出 51](#_Toc25745)

[3.7 Join多种应用 55](#_Toc5618)

[3.8 计数器应用 63](#_Toc17394)

[3.9 数据清洗（ETL） 64](#_Toc22266)

[3.10 MapReduce开发总结 69](#_Toc23669)

[第4章 Hadoop数据压缩 70](#_Toc3901)

[4.1 概述 70](#_Toc28784)

[4.2 MR支持的压缩编码 70](#_Toc22181)

[4.3 压缩方式选择 71](#_Toc25416)

[4.4 压缩位置选择 72](#_Toc10635)

[4.5 压缩参数配置 72](#_Toc21617)

[4.6 压缩实操案例 73](#_Toc21535)

[第5章 Yarn资源调度器 77](#_Toc10146)

[5.1 Yarn基本架构 77](#_Toc31332)

[5.2 Yarn工作机制 78](#_Toc2926)

[5.3 作业提交全过程 79](#_Toc10446)

[5.4 资源调度器 80](#_Toc23010)

[5.5 任务的推测执行 82](#_Toc31485)

[第6章 Hadoop企业优化 83](#_Toc19160)

[6.1 MapReduce 跑的慢的原因 83](#_Toc4399)

[6.2 MapReduce优化方法 83](#_Toc28192)

[6.3 HDFS小文件优化方法 86](#_Toc4701)

[第7章 MapReduce扩展案例 86](#_Toc21420)

[7.1 倒排索引案例（多job串联） 86](#_Toc10267)

[7.2 TopN案例 90](#_Toc14947)

[7.3 找博客共同好友案例 95](#_Toc2750)

[第8章 常见错误及解决方案 100](#_Toc8776)

[Flume 102](#_Toc10058)

[第1章 概述 102](#_Toc17684)

[1.1 Flume定义 102](#_Toc30262)

[1.2 Flume的优点 102](#_Toc20605)

[1.3 Flume组成架构 102](#_Toc4373)

[1.4 Flume拓扑结构 104](#_Toc22451)

[1.5 Flume Agent内部原理 106](#_Toc18070)

[第2章 快速入门 106](#_Toc24037)

[2.1 Flume安装地址 106](#_Toc10112)

[2.2 安装部署 106](#_Toc10697)

[第3章 企业开发案例 106](#_Toc7315)

[3.1 监控端口数据官方案例 106](#_Toc21111)

[3.2 实时读取本地文件到HDFS案例 109](#_Toc21623)

[3.3 实时读取目录文件到HDFS案例 111](#_Toc14366)

[3.4 单数据源多出口案例(选择器) 114](#_Toc4643)

[3.5 单数据源多出口案例(Sink组) 118](#_Toc28504)

[3.6 多数据源汇总案例 121](#_Toc26456)

[第4章 Flume监控之Ganglia 125](#_Toc7211)

[4.1 Ganglia的安装与部署 125](#_Toc1066)

[4.2 操作Flume测试监控 127](#_Toc25076)

[第5章 自定义Source 127](#_Toc19378)

[5.1 介绍 127](#_Toc30674)

[5.2 需求 128](#_Toc8022)

[5.2 分析 128](#_Toc11598)

[5.3 编码 128](#_Toc7501)

[5.4 测试 130](#_Toc18765)

[第6章 自定义Sink 130](#_Toc13383)

[6.1 介绍 130](#_Toc15764)

[6.2 需求 131](#_Toc15722)

[6.3 编码 131](#_Toc15205)

[6.4 测试 133](#_Toc5876)

[第7章 知识扩展 133](#_Toc23031)

[7.1 常见正则表达式语法 134](#_Toc19787)

[7.2 自定义MySQLSource 134](#_Toc17892)

[7.3 练习 144](#_Toc3402)

[第8章 企业真实面试题（重点） 144](#_Toc26810)

[8.1 你是如何实现Flume数据传输的监控的 144](#_Toc12790)

[8.2 Flume的Source，Sink，Channel的作用？你们Source是什么类型？ 144](#_Toc12876)

[8.3 Flume的Channel Selectors 145](#_Toc4374)

[8.4 Flume参数调优 145](#_Toc14728)

[8.5 Flume的事务机制 145](#_Toc1335)

[8.6 Flume采集数据会丢失吗? 146](#_Toc22729)

[Sqoop 147](#_Toc26916)

[第1章 Sqoop简介 147](#_Toc18954)

[第2章 Sqoop原理 147](#_Toc6116)

[第3章 Sqoop安装 147](#_Toc10597)

[3.1 下载并解压 147](#_Toc23976)

[3.2 修改配置文件 147](#_Toc14521)

[3.3 拷贝JDBC驱动 147](#_Toc26980)

[3.4 验证Sqoop 148](#_Toc21883)

[3.5 测试Sqoop是否能够成功连接数据库 148](#_Toc3038)

[第4章 Sqoop的简单使用案例 148](#_Toc23025)

[4.1 导入数据 148](#_Toc14450)

[4.2 导出数据 150](#_Toc11630)

[4.3 脚本打包 150](#_Toc7406)

[第5章 Sqoop一些常用命令及参数 151](#_Toc9031)

[5.1 常用命令列举 151](#_Toc4383)

[5.2 命令&参数详解 152](#_Toc1690)

[Oozie 161](#_Toc25618)

[第1章 Oozie简介 161](#_Toc24861)

[第2章 Oozie的功能模块介绍 161](#_Toc31766)

[2.1模块 161](#_Toc15755)

[2.2 Workflow常用节点 161](#_Toc10234)

[第3章 Oozie的部署 161](#_Toc27252)

[3.1 部署Hadoop（CDH版本的） 161](#_Toc13092)

[3.2 部署Oozie 162](#_Toc10349)

[第4章 Oozie的使用 163](#_Toc7024)

[4.1 案例一：Oozie调度shell脚本 163](#_Toc18177)

[4.2 案例二：Oozie逻辑调度执行多个Job 165](#_Toc32504)

[4.3 案例三：Oozie调度MapReduce任务 166](#_Toc3524)

[4.4 案例四：Oozie定时任务/循环任务 167](#_Toc18744)

[第5章 常见问题总结 170](#_Toc31618)

# MapReduce

# 第1章 MapReduce概述

## 1.1 MapReduce定义

Mapreduce是一个分布式运算程序的编程框架，是用户开发“基于Hadoop的数据分析应用”的核心框架，核心功能是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在一个Hadoop集群上。

## 1.2 MapReduce优缺点

### 1.2.1 优点

1、Mapreduce易于编程

它简单的实现一些接口就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的PC机器上运行。

1. 良好的扩展性

当计算资源得不到满足的时候，可以通过简单的增加机器来扩展它的计算能力

1. 高容错性

当一台机器挂了，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，完全由Hadoop内部完成的。

1. 适合PB级以上的海量数据离线处理

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

2、不擅长流式计算

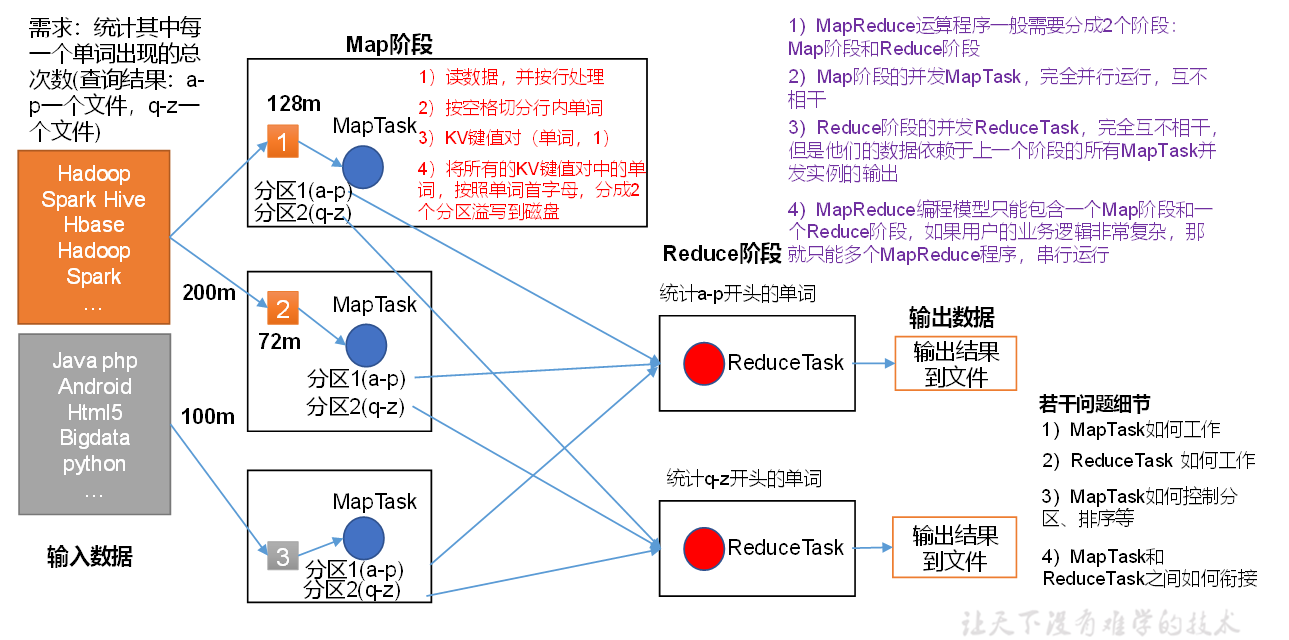
Mapreduce自身设计特点决定了数据源必须是静态的。

1. 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。每个Mapreduce作业的输出结果都会写入到磁盘，造成大量的磁盘IO，导致性能非常低下。

## 1.3 MapReduce核心思想

MapReduce核心编程思想，如图4-1所示。



1）分布式的运算程序往往需要分成至少2个阶段。

2）第一个阶段的MapTask并发实例，完全并行运行，互不相干。

3）第二个阶段的ReduceTask并发实例互不相干，但是他们的数据依赖于上一个阶段的所有MapTask并发实例的输出。

4）MapReduce编程模型只能包含一个Map阶段和一个Reduce阶段，如果用户的业务逻辑非常复杂，那就只能多个MapReduce程序，串行运行。

总结：分析WordCount数据流走向深入理解MapReduce核心思想。

## 1.4 MapReduce进程

一个完整的MapReduce程序在分布式运行时有三类实例进程

1. MrAppMaster：负责整个程序的过程调度及状态协调
2. MapTask：负责Map阶段的整个数据处理流程
3. ReduceTask：负责Reduce阶段的整个数据处理流程

## 1.5 官方WordCount源码

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

## 1.6 常用数据序列化类型

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 |

## 1.7 MapReduce编程规范

用户编写的程序分成三个部分：Mapper、Reducer和Driver。

1. Mapper阶段
2. 用户自定义的Mapper要继承自己的父类
3. Mapper的输入数据是KV对的形式（KV的类型可自定义）
4. Mapper中的业务逻辑写在map()方法中
5. Mapper的输出数据是KV对的形式（KV的类型可自定义）
6. map()方法（MapTask进程）对每一个<K,V>调用一次
7. Reduce阶段
8. 用户自定义Reduce要继承自己的父类
9. Reduce的输入数据类型对应Mapper的输出数据类型，也是KV
10. Reduce的业务逻辑写在reduce()方法中
11. 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，用于提交整个程序到Yarn集群，提交的是封装了MapReduce程序相关运行参数的job对象

## 1.8 WordCount案例实操

1．需求

在给定的文本文件中统计输出每一个单词出现的总次数

（1）输入数据

catguigu atguigu

ss ss

cls cls

jiao

banzhang

xue

hadoop

（2）期望输出数据

atguigu 2

banzhang 1

cls 2

hadoop 1

jiao 1

ss 2

xue 1

2．需求分析

按照MapReduce编程规范，分别编写Mapper，Reducer，Driver，如图4-2所示。



图4-2 需求分析

3．环境准备

（1）创建maven工程

（2）在pom.xml文件中添加如下依赖

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>RELEASE</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-common</artifactId>

<version>2.7.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>2.7.2</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.7.2</version>

</dependency>

</dependencies>

（2）在项目的src/main/resources目录下，新建一个文件，命名为“log4j.properties”，在文件中填入。

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

4．编写程序

（1）编写Mapper类

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

（2）编写Reducer类

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

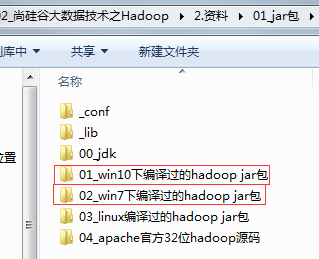
（3）编写Driver驱动类

|  |
| --- |
| package com.atguigu.mapreduce.wordcount;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class WordcountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  // 1 获取配置信息以及封装任务  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置jar加载路径  job.setJarByClass(WordcountDriver.class);  // 3 设置map和reduce类  job.setMapperClass(WordcountMapper.class);  job.setReducerClass(WordcountReducer.class);  // 4 设置map输出  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);  // 5 设置最终输出kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);    // 6 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

5．本地测试

（1）如果电脑系统是win7的就将win7的hadoop jar包解压到非中文路径，并在Windows环境上配置HADOOP\_HOME环境变量。如果是电脑win10操作系统，就解压win10的hadoop jar包，并配置HADOOP\_HOME环境变量。

注意：win8电脑和win10家庭版操作系统可能有问题，需要重新编译源码或者更改操作系统。



（2）在Eclipse/Idea上运行程序

6．集群上测试

（0）用maven打jar包，需要添加的打包插件依赖

注意：标记红颜色的部分需要替换为自己工程主类

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin </artifactId>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

<archive>

<manifest>

<mainClass>com.atguigu.mr.WordcountDriver</mainClass>

</manifest>

</archive>

</configuration>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

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

（1）将程序打成jar包，然后拷贝到Hadoop集群中

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

（2）启动Hadoop集群

（3）执行WordCount程序

[atguigu@hadoop102 software]$ hadoop jar wc.jar

com.atguigu.wordcount.WordcountDriver /user/atguigu/input /user/atguigu/output

# 第2章 Hadoop序列化

## 2.1 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

反序列化就是将受到的字节序列（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

序列化可以将对象发送到远程计算机

2.1.3为什么不用java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（校验信息，Header，继承体系等），不便在网络中高效传输，所以Hadoop有一套自己的序列化机制（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升级
4. 互操作：支持多语言的交互

## 2.2 自定义bean对象实现序列化接口（Writable）

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

具体实现bean对象序列化步骤如下7步。

（1）必须实现Writable接口

（2）反序列化时，必须有空参构造，因为需要反射调用空参构造函数

|  |
| --- |
| public FlowBean() {  super();  } |

（3）重写序列化方法

|  |
| --- |
| @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  } |

（4）重写反序列化方法

|  |
| --- |
| @Override  public void readFields(DataInput in) throws IOException {  upFlow = in.readLong();  downFlow = in.readLong();  sumFlow = in.readLong();  } |

（5）注意反序列化的顺序和序列化的顺序完全一致

（6）要想把结果显示在文件中，需要重写toString()，可用”\t”分开，方便后续用。

（7）如果需要将自定义的bean放在key中传输，则还需要实现Comparable接口，因为MapReduce框中的Shuffle过程要求对key必须能排序。详见后面排序案例。

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

## 2.3 序列化案例实操

1. 需求

统计每一个手机号耗费的总上行流量、下行流量、总流量

1. 输入数据

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



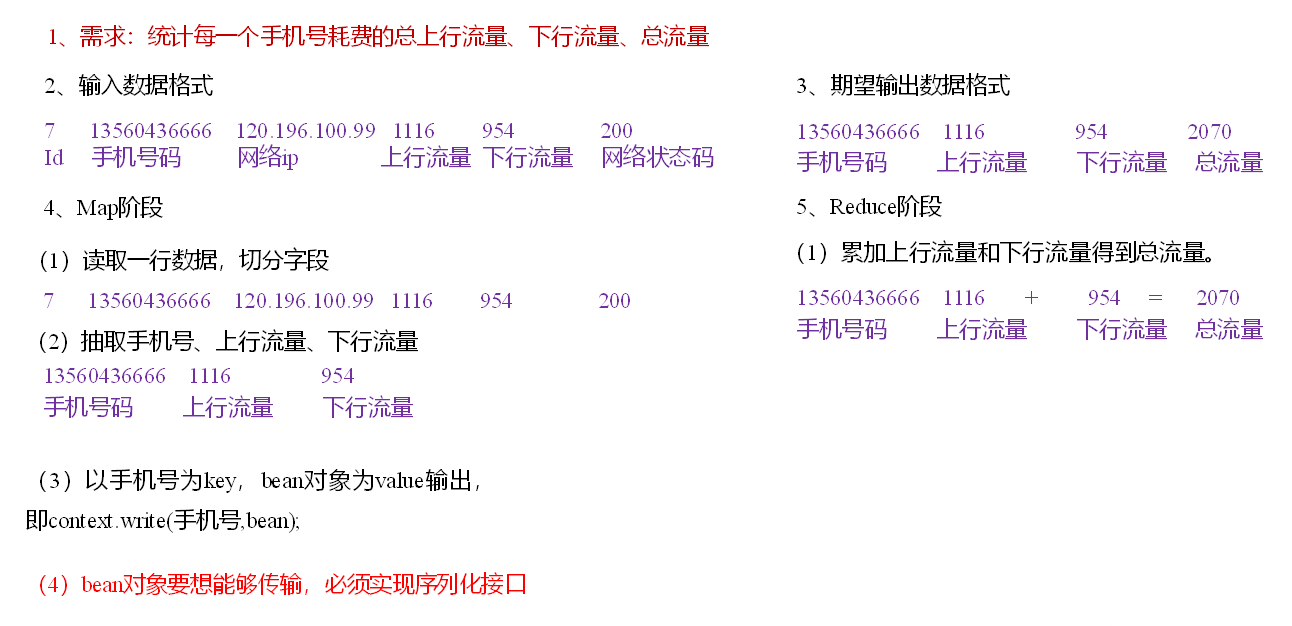
（2）输入数据格式：

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

（3）期望输出数据格式

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

2．需求分析



3．编写MapReduce程序

（1）编写流量统计的Bean对象

|  |
| --- |
| package com.atguigu.mapreduce.flowsum;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  // 1 实现writable接口  public class FlowBean implements Writable{  private long upFlow;  private long downFlow;  private long sumFlow;    //2 反序列化时，需要反射调用空参构造函数，所以必须有  public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }    //3 写序列化方法  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }    //4 反序列化方法  //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致  @Override  public void readFields(DataInput in) throws IOException {  this.upFlow = in.readLong();  this.downFlow = in.readLong();  this.sumFlow = in.readLong();  }  // 6 编写toString方法，方便后续打印到文本  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  public long getUpFlow() {  return upFlow;  }  public void setUpFlow(long upFlow) {  this.upFlow = upFlow;  }  public long getDownFlow() {  return downFlow;  }  public void setDownFlow(long downFlow) {  this.downFlow = downFlow;  }  public long getSumFlow() {  return sumFlow;  }  public void setSumFlow(long sumFlow) {  this.sumFlow = sumFlow;  }  } |

（2）编写Mapper类

|  |
| --- |
| package com.atguigu.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{    FlowBean v = new FlowBean();  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();  // 2 切割字段  String[] fields = line.split("\t");  // 3 封装对象  // 取出手机号码  String phoneNum = fields[1];  // 取出上行流量和下行流量  long upFlow = Long.parseLong(fields[fields.length - 3]);  long downFlow = Long.parseLong(fields[fields.length - 2]);  k.set(phoneNum);  v.set(downFlow, upFlow);  // 4 写出  context.write(k, v);  }  } |

（3）编写Reducer类

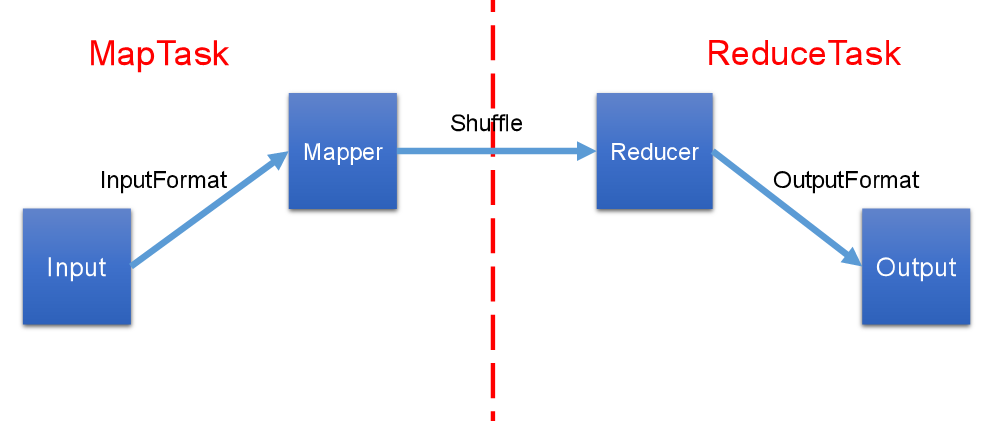
|  |
| --- |
| package com.atguigu.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {  @Override  protected void reduce(Text key, Iterable<FlowBean> values, Context context)throws IOException, InterruptedException {  long sum\_upFlow = 0;  long sum\_downFlow = 0;  // 1 遍历所用bean，将其中的上行流量，下行流量分别累加  for (FlowBean flowBean : values) {  sum\_upFlow += flowBean.getUpFlow();  sum\_downFlow += flowBean.getDownFlow();  }  // 2 封装对象  FlowBean resultBean = new FlowBean(sum\_upFlow, sum\_downFlow);  // 3 写出  context.write(key, resultBean);  }  } |

（4）编写Driver驱动类

|  |
| --- |
| package com.atguigu.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputflow", "e:/output1" };  // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

# 第3章 MapReduce框架原理

## 3.1 InputFormat数据输入



### 3.1.1 切片与MapTask并行度决定机制

1．问题引出

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

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

2．MapTask并行度决定机制

数据块：Block是HDFS物理上把数据分成一块一块。

数据切片：数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。

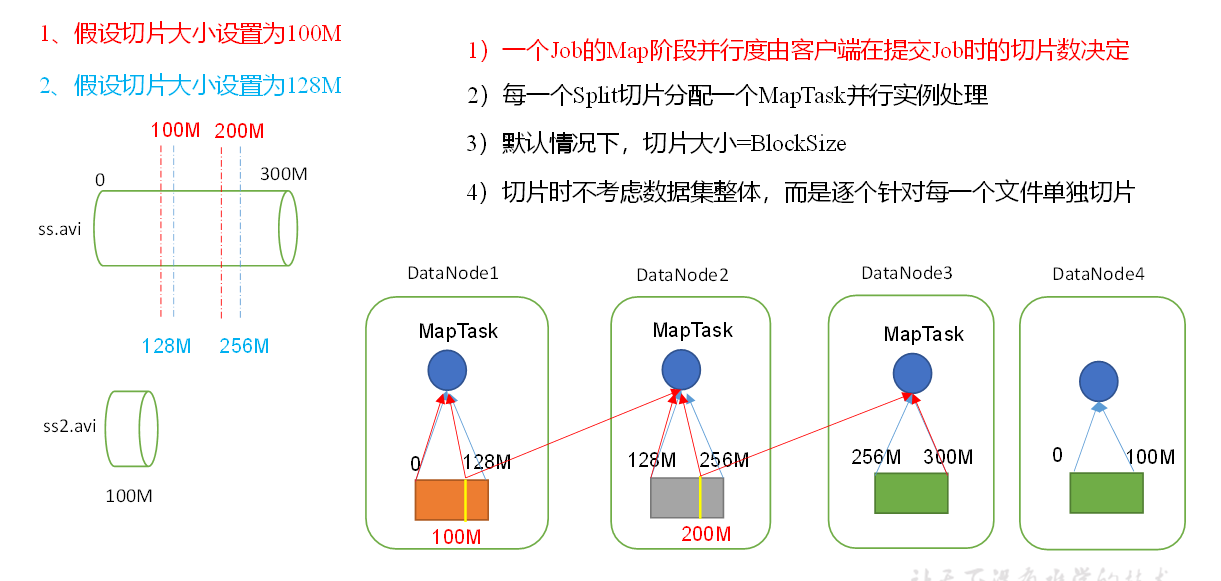


图4-11 MapTask并行度决定机制

### 3.1.2 Job提交流程源码和切片源码详解

1．Job提交流程源码详解，如图4-8所示

waitForCompletion()

submit();

// 1建立连接

connect();

// 1）创建提交Job的代理

new Cluster(getConfiguration());

// 2）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());

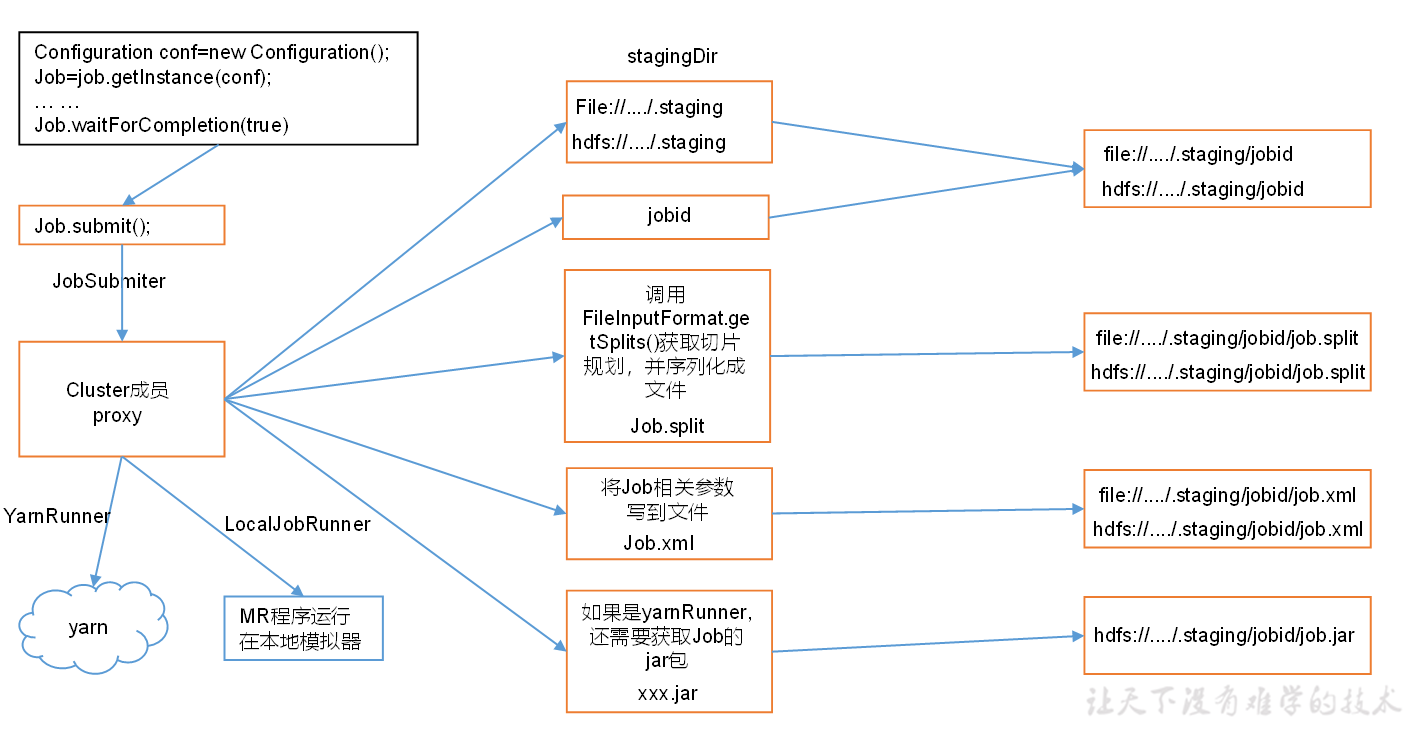


图4-8 Job提交流程源码分析

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

1. 程序先找到你数据存储的目录
2. 开始遍历处理（规划切片）目录下的每一个文件
3. 遍历第一个文件ss.txt

a）获取文件大小fs.sizeOf(SS.txt)

b）计算切片大小

computeSplitSize(Math.max(minSize, Math.min(maxSize, blocksize))) = blocksize = 128M

c）默认情况下，切片大小=blocksize

d）开始切，形成第1个切片：ss.txt——0：128M

第2个切片ss.txt——128：256M

第3个切片ss.txt——256M：300M

（每次切片时，都要判断切完剩下的部分是否大于块的1.1倍，不大于1.1倍就划分一块切片）

e）将切片信息写到一个切片规划文件中

f）整个切片的核心过程在getSplit()方法中完成

g）InputSplit只记录了切片的元数据信息，比如起始位置、长度以及所在的节点列表等。

（4）提交切片规划文件到YARN上，YARN上的ApplicationMaster就可以根据切片规划文件计算开启MapTask个数。

### 3.1.3 FileInputFormat切片机制

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



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

因此在默认情况下，切片大小=blocksize。

1. 切片大小设置

maxsize（切片最大值）：参数如果调到比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）：参数如果调的比blockSize大，则可以让切片变得比blockSize还大。

1. 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();

### 3.1.4 CombineTextInputFormat切片机制

框架默认的TextInputFormat切片机制是对任务按文件规划切片，不管文件多小都会是一个单独的切片，都会交给一个MapTask，这样如果有大量小文件，就会产生大量的MapTask，处理效率极其低下。

1、应用场景：

CombineTextInputFormat用于小文件过多的场景，它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个MapTask处理。

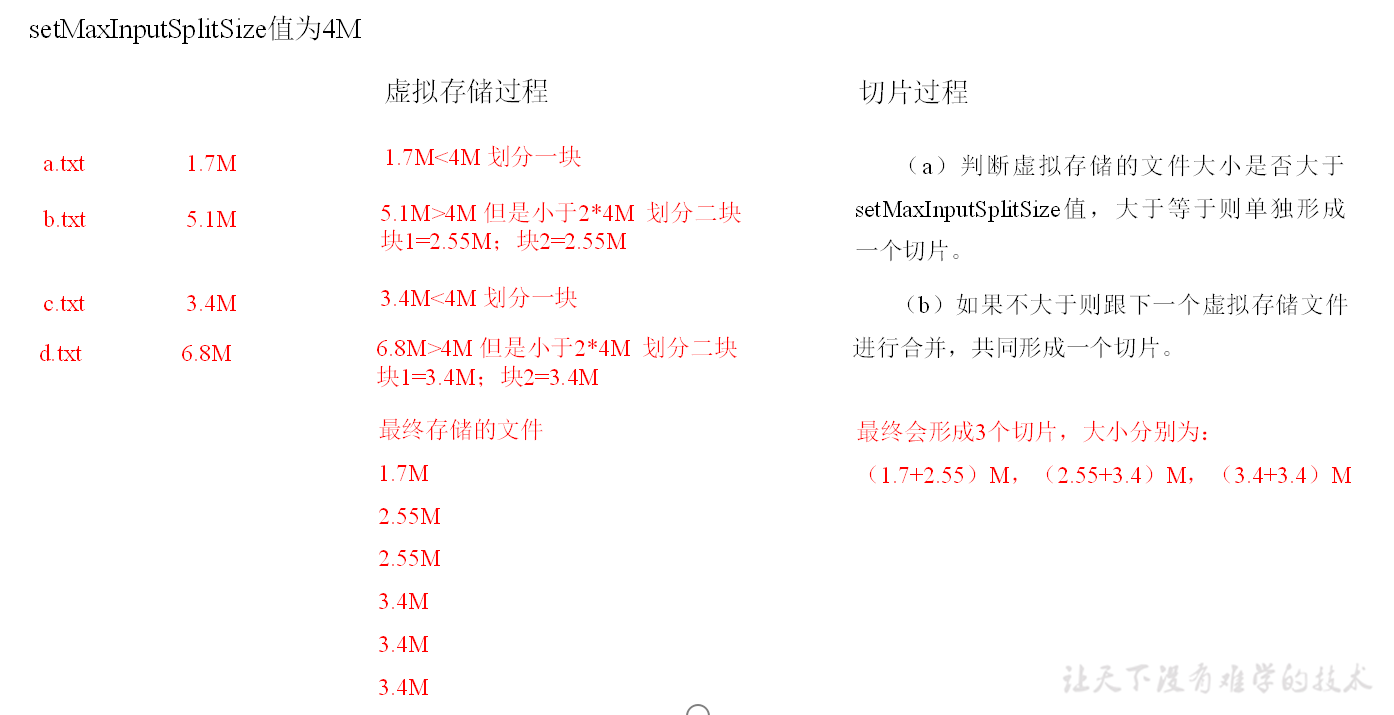
2、虚拟存储切片最大值设置

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

注意：虚拟存储切片最大值设置最好根据实际的小文件大小情况来设置具体的值。

3、切片机制

生成切片过程包括虚拟存储过程和切片过程二部分。



（1）虚拟存储：

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

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

（2）切片：

（a）判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。

（b）如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。

（c）测试举例：有4个小文件大小分别为1.7M、5.1M、3.4M以及6.8M这四个小文件，则虚拟存储之后形成6个文件块，大小分别为：

1.7M，（2.55M、2.55M），3.4M以及（3.4M、3.4M）

最终会形成3个切片，大小分别为：

（1.7+2.55）M，（2.55+3.4）M，（3.4+3.4）M

### 3.1.5 CombineTextInputFormat案例实操

1．需求

将输入的大量小文件合并成一个切片统一处理。

（1）输入数据

准备4个小文件

（2）期望

期望一个切片处理4个文件

2．实现过程

（1）不做任何处理，运行1.6节的WordCount案例程序，观察切片个数为4。

number of split：4

（2）在WordcountDriver中增加如下代码，运行程序，并观察运行的切片个数为3。

（a）驱动类中添加代码如下：

// 如果不设置InputFormat，它默认用的是TextInputFormat.class

job.setInputFormatClass(CombineTextInputFormat.class);

//虚拟存储切片最大值设置4m

CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);

（b）运行如果为3个切片。

number of split：3

**（3）在WordcountDriver中增加如下代码，运行程序，并观察运行的切片个数为1。**

（a）驱动中添加代码如下：

// 如果不设置InputFormat，它默认用的是TextInputFormat.class

job.setInputFormatClass(CombineTextInputFormat.class);

//虚拟存储切片最大值设置20m

CombineTextInputFormat.setMaxInputSplitSize(job, 20971520);

（b）运行如果为1个切片。

number of split：1

### 3.1.6 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。

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钟的其实字节偏移量，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：

1. Rich learning form）
2. Intelligent learning engine）
3. Learning more convenient）
4. From the real demand for more close to the enterprise）
5. **KeyValueInputFormat**

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(KeyValueLineRecordReaderKEY\_VALUE\_SEPERATOR, “t”) 来设定分隔符，默认分割符是tab(\t)。以下是一个示例，输入一个包含4条记录的分片。其中——>表示一个（水平方向）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。

1. **NlineInputFormat**

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，假设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。

### 3.1.7 KeyValueTextInputFormat使用案例

1．需求

统计输入文件中每一行的第一个单词相同的行数。

（1）输入数据

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

（2）期望结果数据

banzhang 2

xihuan 2

2．需求分析



3．代码实现

（1）编写Mapper类

|  |
| --- |
| package com.atguigu.mapreduce.KeyValueTextInputFormat;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class KVTextMapper extends Mapper<Text, Text, Text, LongWritable>{  // 1 设置value  LongWritable v = new LongWritable(1);    @Override  protected void map(Text key, Text value, Context context)throws IOException, InterruptedException {  // banzhang ni hao  // 2 写出  context.write(key, v);  }  } |

（2）编写Reducer类

|  |
| --- |
| package com.atguigu.mapreduce.KeyValueTextInputFormat;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class KVTextReducer extends Reducer<Text, LongWritable, Text, LongWritable>{  LongWritable v = new LongWritable();  @Override  protected void reduce(Text key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {  long sum = 0L;  // 1 汇总统计  for (LongWritable value : values) {  sum += value.get();  }  v.set(sum);  // 2 输出  context.write(key, v);  }  } |

（3）编写Driver类

|  |
| --- |
| package com.atguigu.mapreduce.keyvaleTextInputFormat;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.LongWritable;  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;  public class KVTextDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  Configuration conf = new Configuration();  // 设置切割符  conf.set(KeyValueLineRecordReader.KEY\_VALUE\_SEPERATOR, " ");  // 1 获取job对象  Job job = Job.getInstance(conf);  // 2 设置jar包位置，关联mapper和reducer  job.setJarByClass(KVTextDriver.class);  job.setMapperClass(KVTextMapper.class);  job.setReducerClass(KVTextReducer.class);  // 3 设置map输出kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(LongWritable.class);  // 4 设置最终输出kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(LongWritable.class);  // 5 设置输入输出数据路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  // 设置输入格式  job.setInputFormatClass(KeyValueTextInputFormat.class);  // 6 设置输出数据路径  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 提交job  job.waitForCompletion(true);  }  } |

### 3.1.8 NLineInputFormat使用案例

1．需求

对每个单词进行个数统计，要求根据每个输入文件的行数来规定输出多少个切片。此案例要求每三行放入一个切片中。

（1）输入数据

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

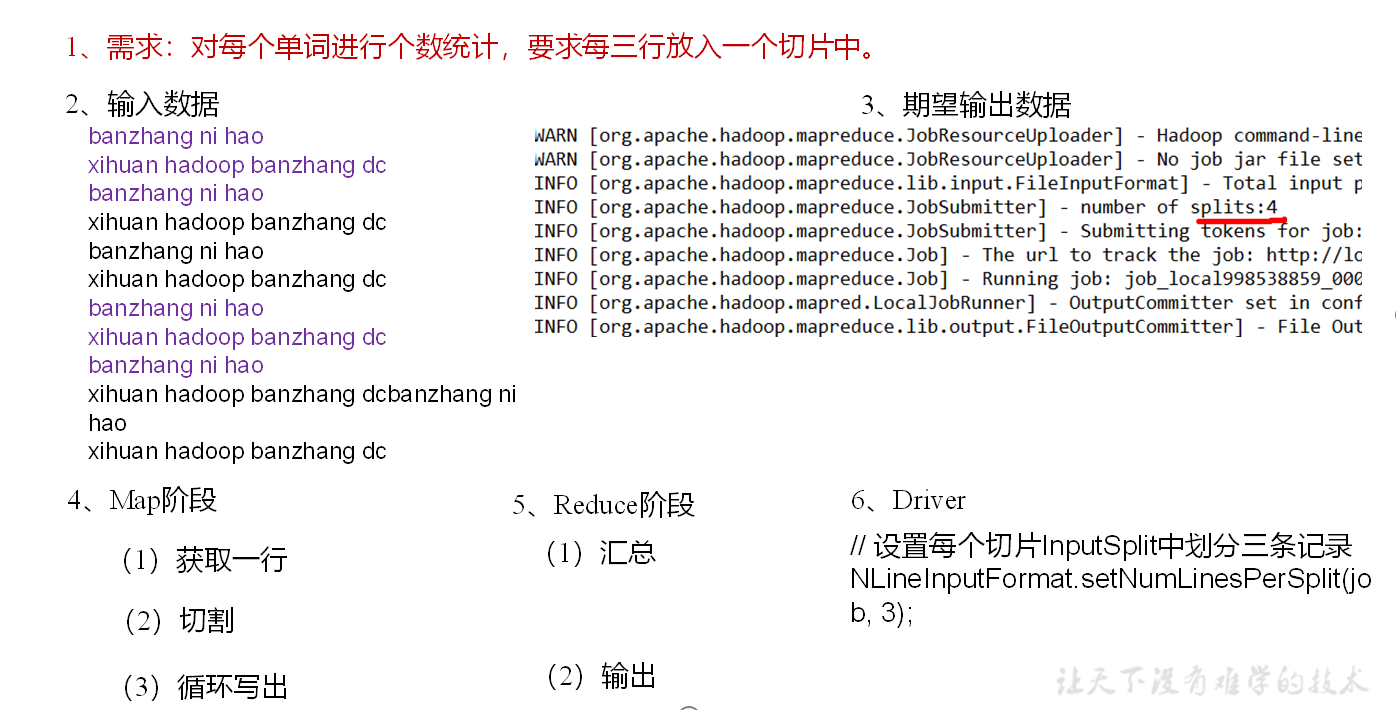
xihuan hadoop banzhang banzhang ni hao

xihuan hadoop banzhang

（2）期望输出数据

Number of splits:4

2．需求分析



3．代码实现

（1）编写Mapper类

|  |
| --- |
| package com.atguigu.mapreduce.nline;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class NLineMapper extends Mapper<LongWritable, Text, Text, LongWritable>{  private Text k = new Text();  private LongWritable v = new LongWritable(1);    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();  // 2 切割  String[] splited = line.split(" ");  // 3 循环写出  for (int i = 0; i < splited.length; i++) {  k.set(splited[i]);  context.write(k, v);  }  }  } |

（2）编写Reducer类

|  |
| --- |
| package com.atguigu.mapreduce.nline;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class NLineReducer extends Reducer<Text, LongWritable, Text, LongWritable>{  LongWritable v = new LongWritable();    @Override  protected void reduce(Text key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {  long sum = 0l;  // 1 汇总  for (LongWritable value : values) {  sum += value.get();  }  v.set(sum);  // 2 输出  context.write(key, v);  }  } |

（3）编写Driver类

|  |
| --- |
| package com.atguigu.mapreduce.nline;  import java.io.IOException;  import java.net.URISyntaxException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.LongWritable;  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.NLineInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class NLineDriver {  public static void main(String[] args) throws IOException, URISyntaxException, ClassNotFoundException, InterruptedException {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputword", "e:/output1" };  // 1 获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 7设置每个切片InputSplit中划分三条记录  NLineInputFormat.setNumLinesPerSplit(job, 3);    // 8使用NLineInputFormat处理记录数  job.setInputFormatClass(NLineInputFormat.class);    // 2设置jar包位置，关联mapper和reducer  job.setJarByClass(NLineDriver.class);  job.setMapperClass(NLineMapper.class);  job.setReducerClass(NLineReducer.class);    // 3设置map输出kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(LongWritable.class);    // 4设置最终输出kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(LongWritable.class);    // 5设置输入输出数据路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 6提交job  job.waitForCompletion(true);  }  } |

4．测试

（1）输入数据

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang

banzhang ni hao

xihuan hadoop banzhang banzhang ni hao

xihuan hadoop banzhang

（2）输出结果的切片数，如图4-10所示：

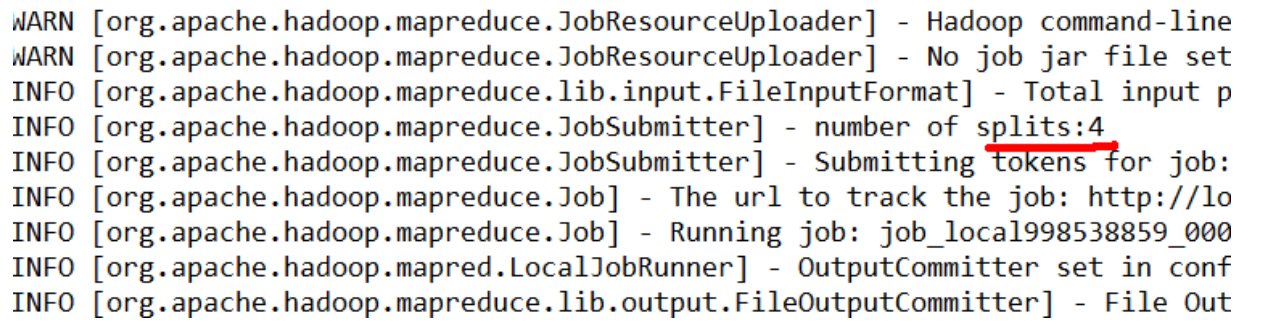


图4-10 输出结果的切片数

### 3.1.9 自定义InputFormat

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

### 3.1.10 自定义InputFormat案例实操

无论HDFS还是MapReduce，在处理小文件时效率都非常低，但又难免面临处理大量小文件的场景，此时就需要有相应解决方案。可以自定义InputFormat实现小文件的合并。

1．需求

将多个小文件合并成一个SequenceFile文件（SequenceFile文件是Hadoop用来存储二进制形式的key-value对的文件格式），SequenceFile里面存储着多个文件，存储的形式是文件路径+名称为key，文件内容为value。

（1）输入数据

1. 期望输出文件格式

part-r-00000

1. 需求分析
2. 自定义一个类继承FileInputFormat

（1）重写isSplitable()方法，返回false不可分割

（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)

3．程序实现

（1）自定义InputFromat

|  |
| --- |
| package com.atguigu.mapreduce.inputformat;  import java.io.IOException;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.mapreduce.InputSplit;  import org.apache.hadoop.mapreduce.JobContext;  import org.apache.hadoop.mapreduce.RecordReader;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  // 定义类继承FileInputFormat  public class WholeFileInputformat extends FileInputFormat<Text, BytesWritable>{  @Override  protected boolean isSplitable(JobContext context, Path filename) {  return false;  }  @Override InputSplit 输入分片  public RecordReader<Text, BytesWritable> createRecordReader(InputSplit split, TaskAttemptContext context) throws IOException, InterruptedException {  WholeRecordReader recordReader = new WholeRecordReader();  recordReader.initialize(split, context);  return recordReader;  }  } |

（2）自定义RecordReader类

|  |
| --- |
| package com.atguigu.mapreduce.inputformat;  import java.io.IOException;  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.NullWritable;  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;  public class WholeRecordReader extends RecordReader<Text, BytesWritable>{  private Configuration configuration;  private FileSplit split;  private boolean isProgress= true;  private BytesWritable value = new BytesWritable();  private Text k = new Text();  @Override  public void initialize(InputSplit split, TaskAttemptContext context) throws IOException, InterruptedException {  this.split = (FileSplit)split;  configuration = context.getConfiguration();  }  @Override  public boolean nextKeyValue() throws IOException, InterruptedException {  if (isProgress) {  // 1 定义缓存区  byte[] contents = new byte[(int)split.getLength()]; // 定义一个byte数组，大小为分片长度  FileSystem fs = null;  FSDataInputStream fis = null;  try {  // 2 获取文件系统  Path path = split.getPath(); // 获得分片路径  fs = path.getFileSystem(configuration); // 获取分片路径所处的文件系统  // 3 读取数据  fis = fs.open(path);  // 4 读取文件内容  IOUtils.readFully(fis, contents, 0, contents.length);  // 5 输出文件内容  value.set(contents, 0, contents.length);  // 6 获取文件路径及名称  String name = split.getPath().toString();  // 7 设置输出的key值  k.set(name);  } catch (Exception e) {    }finally {  IOUtils.closeStream(fis);  }    isProgress = false; // 此方法只能调用一次，其他时候都是返回false  return true; // 返回true  }  return false; // 返回false  }  @Override  public Text getCurrentKey() throws IOException, InterruptedException {  return k;  }  @Override  public BytesWritable getCurrentValue() throws IOException, InterruptedException {  return value;  }  @Override  public float getProgress() throws IOException, InterruptedException {  return 0;  }  @Override  public void close() throws IOException {  }  } |

（3）编写SequenceFileMapper类处理流程

|  |
| --- |
| package com.atguigu.mapreduce.inputformat;  import java.io.IOException;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.lib.input.FileSplit;  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);  }  } |

（4）编写SequenceFileReducer类处理流程

|  |
| --- |
| package com.atguigu.mapreduce.inputformat;  import java.io.IOException;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class SequenceFileReducer extends Reducer<Text, BytesWritable, Text, BytesWritable> {  @Override  protected void reduce(Text key, Iterable<BytesWritable> values, Context context) throws IOException, InterruptedException {  context.write(key, values.iterator().next());  }  } |

（5）编写SequenceFileDriver类处理流程

|  |
| --- |
| package com.atguigu.mapreduce.inputformat;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  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;  public class SequenceFileDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputinputformat", "e:/output1" };  // 1 获取job对象  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  // 2 设置jar包存储位置、关联自定义的mapper和reducer  job.setJarByClass(SequenceFileDriver.class);  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);    // 3 设置map输出端的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(BytesWritable.class);    // 4 设置最终输出端的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(BytesWritable.class);  // 5 设置输入输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 提交job  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

## 3.2 MapReduce工作流程

1．流程示意图，如图4-6，4-7所示

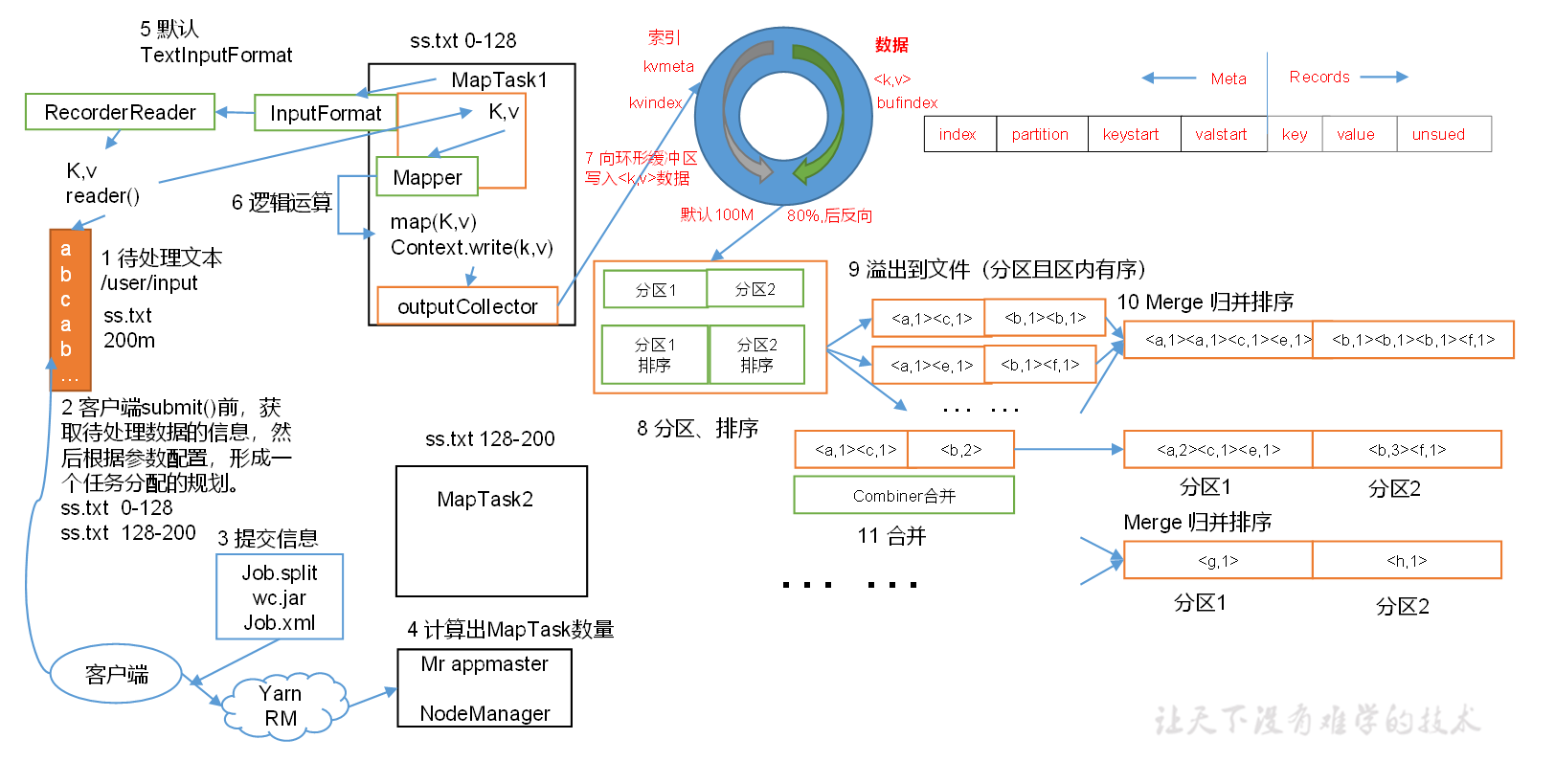


图4-6 MapReduce详细工作流程（一）

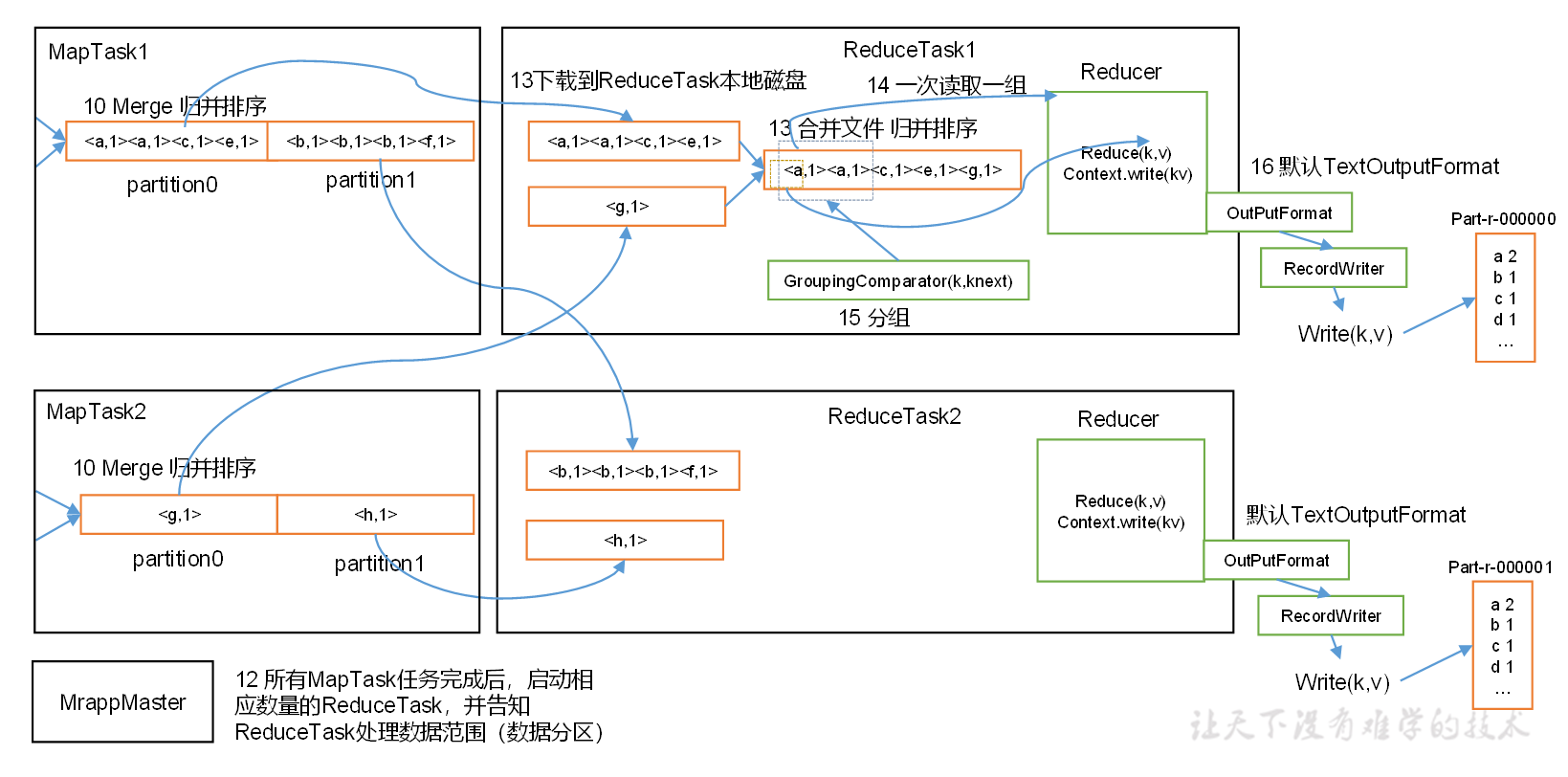


图4-7 MapReduce详细工作流程（二）

2．流程详解

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

1）MapTask收集我们的map()方法输出的kv对，放到内存缓冲区中；

2）从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件；

3）多个溢出文件会被合并成大的溢出文件；

4）在溢出过程及合并的过程中，都要调用Partitioner进行分区和针对key进行排序；

5）ReduceTask根据自己的分区号，去各个MapTask机器上取相应的结果分区数据；

6）ReduceTask会取到同一个分区的来自不同MapTask的结果文件，ReduceTask会将这些文件再进行合并（归并排序）

7）合并成大文件后，Shuffle的过程也就结束了，后面进入ReduceTask的逻辑运算过程（从文件中取出一个一个的键值对Group，调用用户自定义的reduce()方法）

3．注意

Shuffle中的缓冲区大小会影响到MapReduce程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快。缓冲区的大小可以通过参数调整，参数：io.sort.mb默认100M。

4．源码解析流程

context.write(k, NullWritable.get());

output.write(key, value);

collector.collect(key, value,partitioner.getPartition(key, value, partitions));

HashPartitioner();

collect()

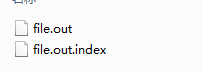
close()

collect.flush()

sortAndSpill()

sort() QuickSort

mergeParts();



collector.close();

## 3.3 Shuffle机制

### 3.3.1 Shuffle机制

Map方法之后，Reduce方法之前的数据处理过程称之为Shuffle。如图4-14所示。

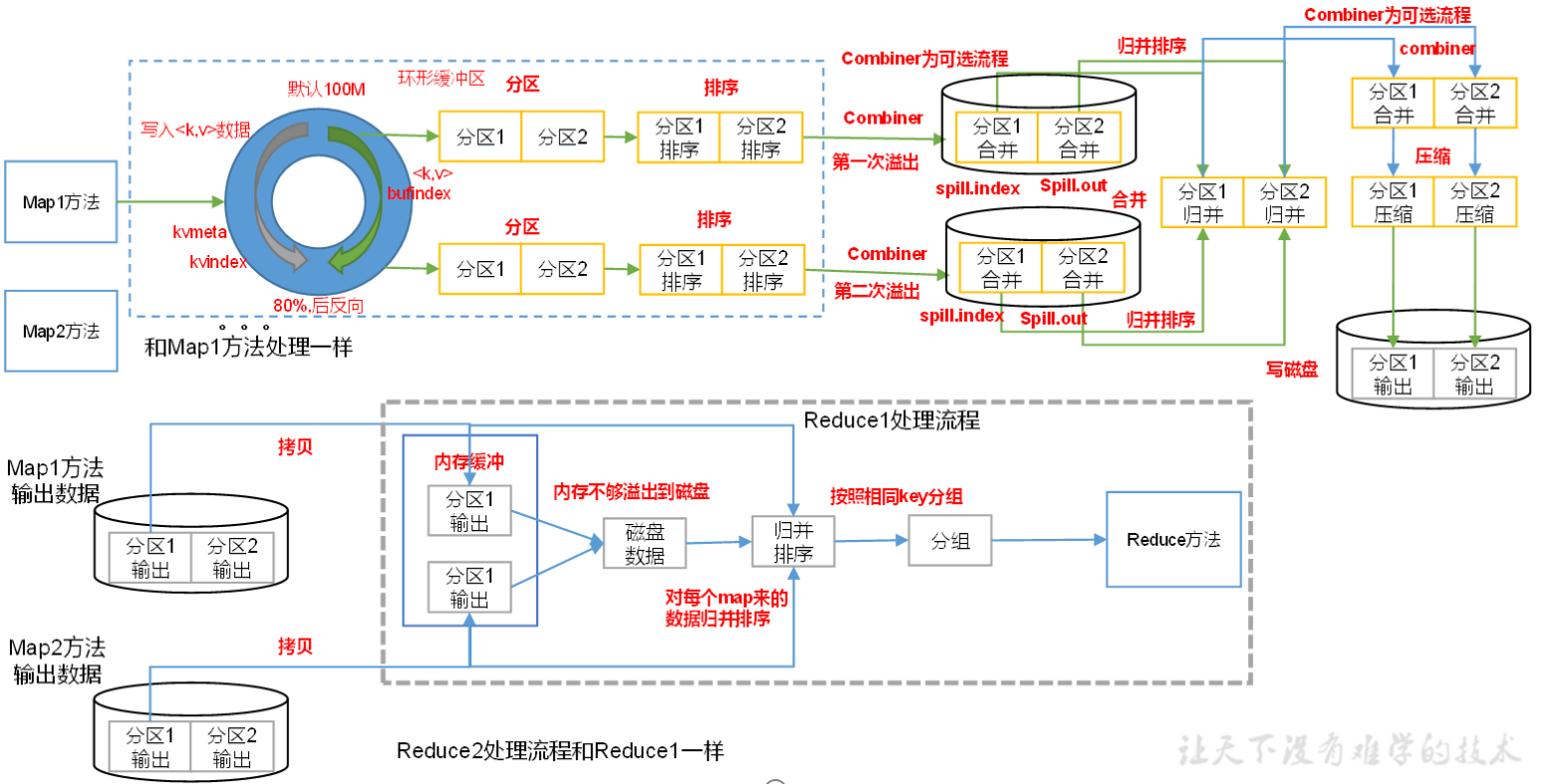


图4-14 Shuffle机制

### 3.3.2 Partition分区

1、问题引出

要求将统计结果按照条件输出到不同文件中（分区）。比如：将统计结果按照手机归属地不同省份输出到不同文件中（分区）。

2、默认Partitioner分区

public class HashPartitioner<K, V> extends Partitioner<K,V>{

Public int getPartition(K key, V value, int numReduceTasks){

Return (key.hashCode() & Integer.MAC\_VALUE) % numReduceTasks;

}

}

默认分区是根据key的hashCode对Reduce Tasks个数取模得到的，用户没法控制哪个key存储到哪个分区。

1. 自定义Partiotioner步骤
2. 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;

}

}

1. 在Job驱动中，设置自定义Partitioner

job.setPartitionerClass(CustomPartitioner.class);

1. 自定义Partition后，要根据自定义Partitioner的逻辑设计相应数量的ReduceTask

Job.setNumReduceTasks(5);

1. 分区总结
2. 如果ReduceTask的数量 > getPartition的结果数，则会多产生几个空的输出文件
3. 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；
4. 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ReduceTas，也就只会产生一个结果文件；
5. 分区号必须从0开始逐一累加
6. 案例分析

假设自定义分区数为5，则

1. Job.setNumReduceTasks(1)；会正常运行，但只产生一个输出文件
2. Job.setNumReduceTasks(2)；会报错
3. Job.setNumReduceTasks(6)；大于分区数，程序正常运行，但会产生空文件

### 3.3.3 Partition分区案例实操

1．需求

将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1. 输入数据

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

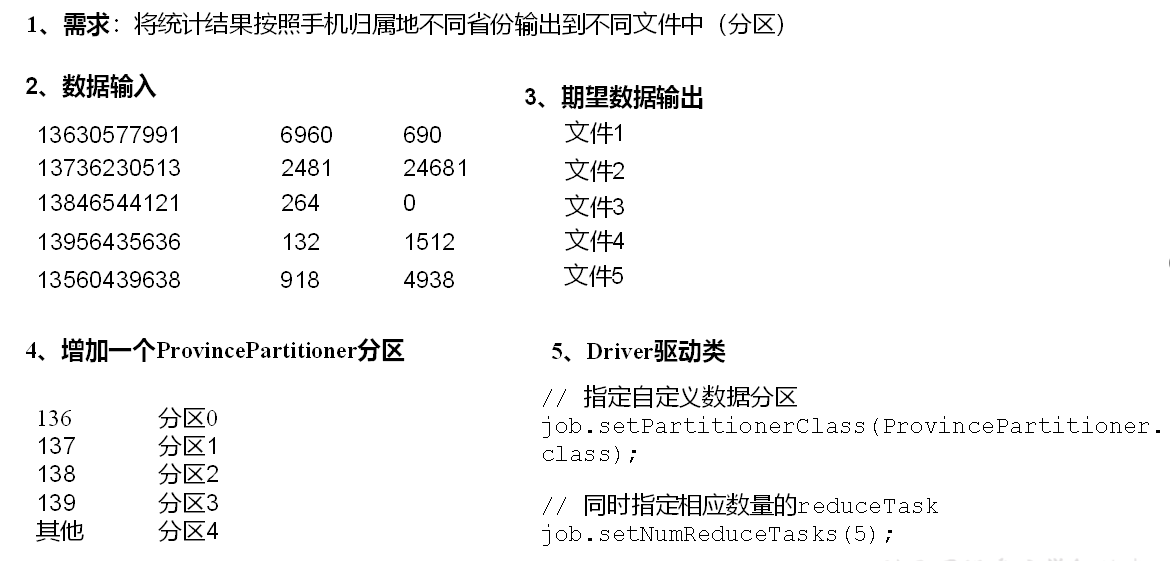
5 18271575951 192.168.100.2 www.atguigu.com 1527 2106 200



（2）期望输出数据

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

2．需求分析



3．在案例2.4的基础上，增加一个分区类

package com.atguigu.mapreduce.flowsum;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

@Override

public int getPartition(Text key, FlowBean value, int numPartitions) {

// 1 获取电话号码的前三位

String preNum = key.toString().substring(0, 3);

int partition = 4;

// 2 判断是哪个省

if ("136".equals(preNum)) {

partition = 0;

}else if ("137".equals(preNum)) {

partition = 1;

}else if ("138".equals(preNum)) {

partition = 2;

}else if ("139".equals(preNum)) {

partition = 3;

}

return partition;

}

}

4．在驱动函数中增加自定义数据分区设置和ReduceTask设置

package com.atguigu.mapreduce.flowsum;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

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

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

public class FlowsumDriver {

public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {

// 输入输出路径需要根据自己电脑上实际的输入输出路径设置

args = new String[]{"e:/output1","e:/output2"};

// 1 获取配置信息，或者job对象实例

Configuration configuration = new Configuration();

Job job = Job.getInstance(configuration);

// 2 指定本程序的jar包所在的本地路径

job.setJarByClass(FlowsumDriver.class);

// 3 指定本业务job要使用的mapper/Reducer业务类

job.setMapperClass(FlowCountMapper.class);

job.setReducerClass(FlowCountReducer.class);

// 4 指定mapper输出数据的kv类型

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(FlowBean.class);

// 5 指定最终输出的数据的kv类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(FlowBean.class);

// 8 指定自定义数据分区

job.setPartitionerClass(ProvincePartitioner.class);

// 9 同时指定相应数量的reduce task

job.setNumReduceTasks(5);

// 6 指定job的输入原始文件所在目录

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

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

// 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行

boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);

}

}

### 3.3.4 WritableComparable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默认行为，任何应用程序中的数据均会被排序，而逻辑上不管是否需要。

默认按照字典顺序排序，排序方法是快速排序。

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，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，则溢写到磁盘，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产生一个更大的文件；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额所有数据进行一次归并排序。

1. 排序的分类
2. 部分排序

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

1. 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，丧失了MapReduce提供的并行架构

1. 辅助排序（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

1. 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。

2．自定义排序WritableComparable

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

@Override

public int compareTo(FlowBean o) {

int result;

// 按照总流量大小，倒序排列

if (sumFlow > bean.getSumFlow()) {

result = -1;

}else if (sumFlow < bean.getSumFlow()) {

result = 1;

}else {

result = 0;

}

return result;

}

### 3.3.5 WritableComparable排序案例实操（全排序）

1．需求

根据案例2.3产生的结果再次对总流量进行排序。

（1）输入数据

原始数据 第一次处理后的数据

（2）期望输出数据

13509468723 7335 110349 117684

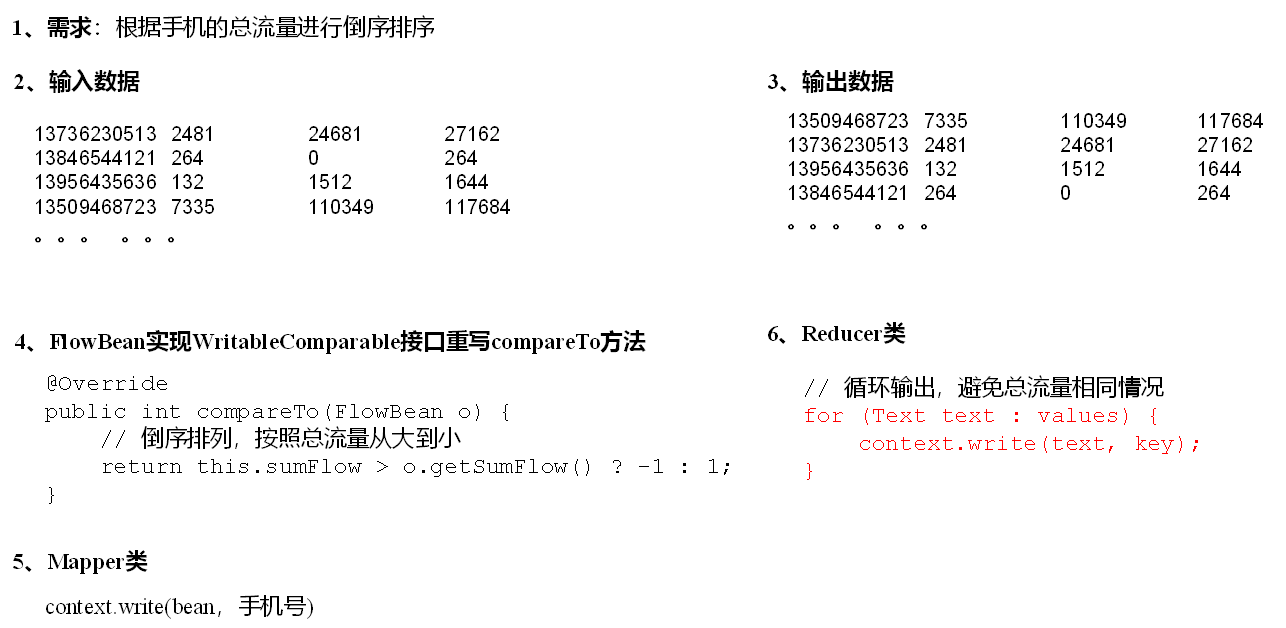
13736230513 2481 24681 27162

13956435636 132 1512 1644

13846544121 264 0 264

。。。 。。。

2．需求分析



3．代码实现

（1）FlowBean对象在在需求1基础上增加了比较功能

package com.atguigu.mapreduce.sort;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class FlowBean implements WritableComparable<FlowBean> {

private long upFlow;

private long downFlow;

private long sumFlow;

// 反序列化时，需要反射调用空参构造函数，所以必须有

public FlowBean() {

super();

}

public FlowBean(long upFlow, long downFlow) {

super();

this.upFlow = upFlow;

this.downFlow = downFlow;

this.sumFlow = upFlow + downFlow;

}

public void set(long upFlow, long downFlow) {

this.upFlow = upFlow;

this.downFlow = downFlow;

this.sumFlow = upFlow + downFlow;

}

public long getSumFlow() {

return sumFlow;

}

public void setSumFlow(long sumFlow) {

this.sumFlow = sumFlow;

}

public long getUpFlow() {

return upFlow;

}

public void setUpFlow(long upFlow) {

this.upFlow = upFlow;

}

public long getDownFlow() {

return downFlow;

}

public void setDownFlow(long downFlow) {

this.downFlow = downFlow;

}

/\*\*

\* 序列化方法

\* @param out

\* @throws IOException

\*/

@Override

public void write(DataOutput out) throws IOException {

out.writeLong(upFlow);

out.writeLong(downFlow);

out.writeLong(sumFlow);

}

/\*\*

\* 反序列化方法 注意反序列化的顺序和序列化的顺序完全一致

\* @param in

\* @throws IOException

\*/

@Override

public void readFields(DataInput in) throws IOException {

upFlow = in.readLong();

downFlow = in.readLong();

sumFlow = in.readLong();

}

@Override

public String toString() {

return upFlow + "\t" + downFlow + "\t" + sumFlow;

}

@Override

public int compareTo(FlowBean o) {

int result;

// 按照总流量大小，倒序排列

if (sumFlow > bean.getSumFlow()) {

result = -1;

}else if (sumFlow < bean.getSumFlow()) {

result = 1;

}else {

result = 0;

}

return result;

}

}

（2）编写Mapper类

package com.atguigu.mapreduce.sort;

import java.io.IOException;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean, Text>{

FlowBean bean = new FlowBean();

Text v = new Text();

@Override

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

// 1 获取一行

String line = value.toString();

// 2 截取

String[] fields = line.split("\t");

// 3 封装对象

String phoneNbr = fields[0];

long upFlow = Long.parseLong(fields[1]);

long downFlow = Long.parseLong(fields[2]);

bean.set(upFlow, downFlow);

v.set(phoneNbr);

// 4 输出

context.write(bean, v);

}

}

（3）编写Reducer类

package com.atguigu.mapreduce.sort;

import java.io.IOException;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text, FlowBean>{

@Override

protected void reduce(FlowBean key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

// 循环输出，避免总流量相同情况

for (Text text : values) {

context.write(text, key);

}

}

}

（4）编写Driver类

package com.atguigu.mapreduce.sort;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

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

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

public class FlowCountSortDriver {

public static void main(String[] args) throws ClassNotFoundException, IOException, InterruptedException {

// 输入输出路径需要根据自己电脑上实际的输入输出路径设置

args = new String[]{"e:/output1","e:/output2"};

// 1 获取配置信息，或者job对象实例

Configuration configuration = new Configuration();

Job job = Job.getInstance(configuration);

// 2 指定本程序的jar包所在的本地路径

job.setJarByClass(FlowCountSortDriver.class);

// 3 指定本业务job要使用的mapper/Reducer业务类

job.setMapperClass(FlowCountSortMapper.class);

job.setReducerClass(FlowCountSortReducer.class);

// 4 指定mapper输出数据的kv类型

job.setMapOutputKeyClass(FlowBean.class);

job.setMapOutputValueClass(Text.class);

// 5 指定最终输出的数据的kv类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(FlowBean.class);

// 6 指定job的输入原始文件所在目录

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

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

// 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行

boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);

}

}

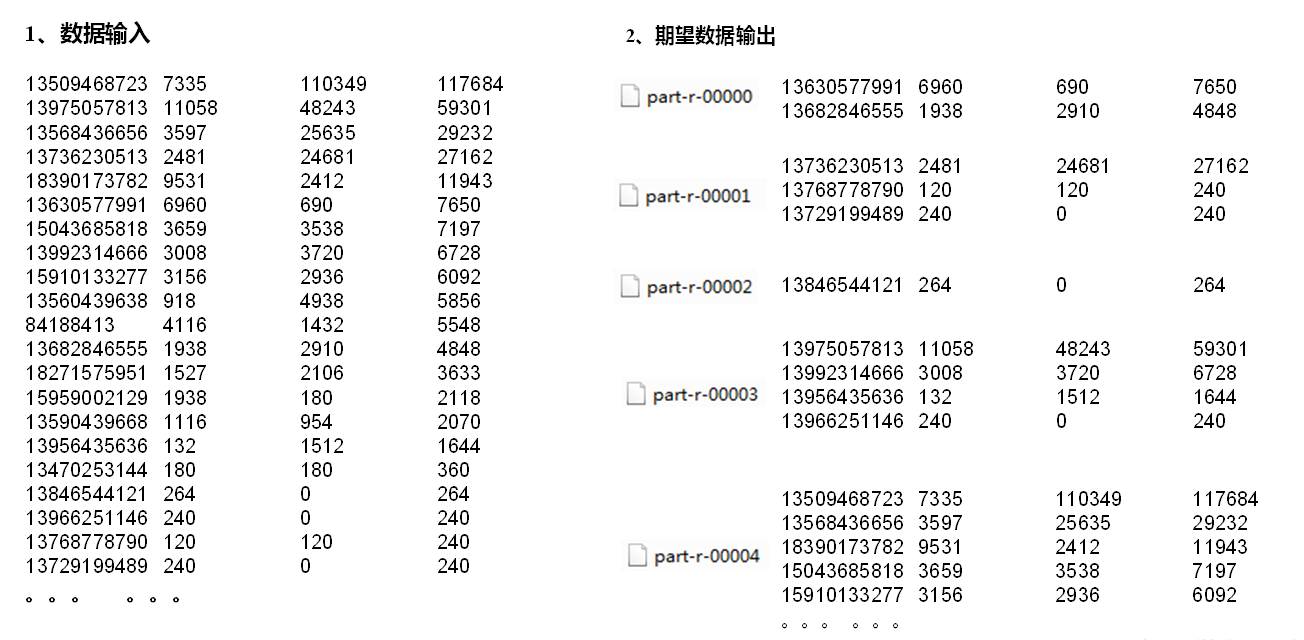
### 3.3.6 WritableComparable排序案例实操（区内排序）

1．需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2．需求分析

基于前一个需求，增加自定义分区类，分区按照省份手机号设置。



3．案例实操

（1）增加自定义分区类

package com.atguigu.mapreduce.sort;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<FlowBean, Text> {

@Override

public int getPartition(FlowBean key, Text value, int numPartitions) {

// 1 获取手机号码前三位

String preNum = value.toString().substring(0, 3);

int partition = 4;

// 2 根据手机号归属地设置分区

if ("136".equals(preNum)) {

partition = 0;

}else if ("137".equals(preNum)) {

partition = 1;

}else if ("138".equals(preNum)) {

partition = 2;

}else if ("139".equals(preNum)) {

partition = 3;

}

return partition;

}

}

（2）在驱动类中添加分区类

// 加载自定义分区类

job.setPartitionerClass(ProvincePartitioner.class);

// 设置Reducetask个数

job.setNumReduceTasks(5);

### 3.3.7 Combiner合并

（1）Combine是MR程序中Mapper和Reducer之外的一种组件

（2）Combine组件的父类就是Reducer。

（3）Combine和Reducer的区别在于运行的位置

Combiner是在每一个MapTask所在的节点运行；

Reducer是接收全局所有Mapper的输出结果；

（4）Combiner的意义就是对每一个MapTask的输出进行局部汇总，以减小网络传输量。

（5）Combiner能够应用的前提是不能影响最终的业务逻辑，而且，Combiner的输出kv应该跟Reducer的输入kv类型要对应起来。

Mapper Reducer

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

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

（6）自定义Combiner实现步骤

（a）自定义一个Combiner继承Reducer，重写Reduce方法

|  |
| --- |
| public class WordcountCombiner extends Reducer<Text, IntWritable, Text,IntWritable>{  @Override  protected void reduce(Text key, Iterable<IntWritable> values,Context context) throws IOException, InterruptedException {  // 1 汇总操作  int count = 0;  for(IntWritable v :values){  count += v.get();  }  // 2 写出  context.write(key, new IntWritable(count));  }  } |

（b）在Job驱动类中设置

|  |
| --- |
| job.setCombinerClass(WordcountCombiner.class); |

### 3.3.8 Combiner合并案例实操

1．需求

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

（1）数据输入



（2）期望输出数据

期望：Combine输入数据多，输出时经过合并，输出数据降低。

2．需求分析

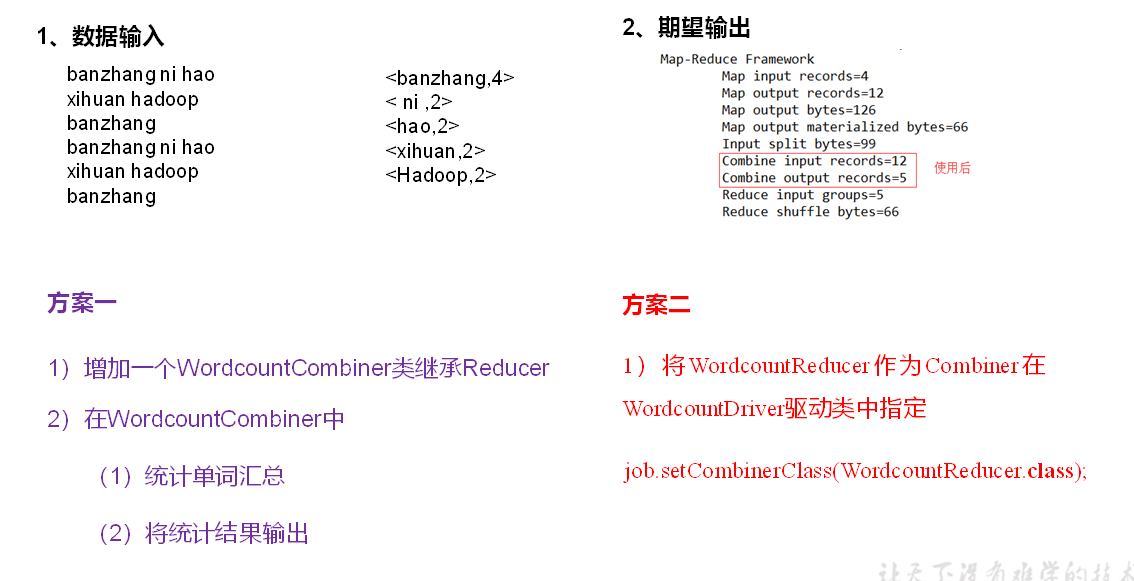


图4-15 Combiner的合并案例

3．案例实操-方案一

1）增加一个WordcountCombiner类继承Reducer

package com.atguigu.mr.combiner;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

public class WordcountCombiner extends Reducer<Text, IntWritable, Text, IntWritable>{

IntWritable v = new IntWritable();

@Override

protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {

// 1 汇总

int sum = 0;

for(IntWritable value :values){

sum += value.get();

}

v.set(sum);

// 2 写出

context.write(key, v);

}

}

2）在WordcountDriver驱动类中指定Combiner

// 指定需要使用combiner，以及用哪个类作为combiner的逻辑

job.setCombinerClass(WordcountCombiner.class);

4．案例实操-方案二

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

// 指定需要使用Combiner，以及用哪个类作为Combiner的逻辑

job.setCombinerClass(WordcountReducer.class);

运行程序，如图4-16，4-17所示

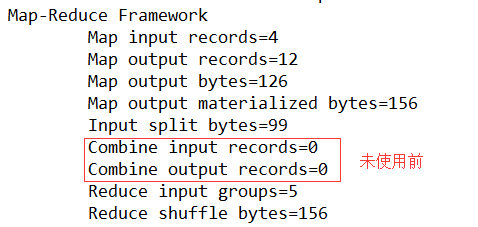


图4-16 未使用前

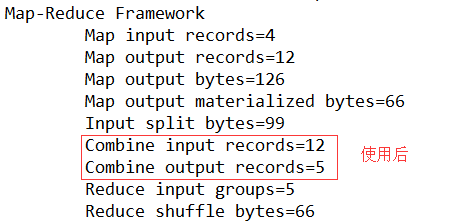


图4-17 使用后

### 3.3.9 GroupingComparator分组（辅助排序）

对Reduce阶段的数据根据某一个或几个字段进行分组。

分组排序步骤：

（1）自定义类继承WritableComparator

（2）重写compare()方法

@Override

public int compare(WritableComparable a, WritableComparable b) {

// 比较的业务逻辑

return result;

}

（3）创建一个构造将比较对象的类传给父类

protected OrderGroupingComparator() {

super(OrderBean.class, true);

}

### 3.3.10 GroupingComparator分组案例实操

1．需求

有如下订单数据

表4-2 订单数据

|  |  |  |
| --- | --- | --- |
| 订单id | 商品id | 成交金额 |
| 0000001 | Pdt\_01 | 222.8 |
| Pdt\_02 | 33.8 |
| 0000002 | Pdt\_03 | 522.8 |
| Pdt\_04 | 122.4 |
| Pdt\_05 | 722.4 |
| 0000003 | Pdt\_06 | 232.8 |
| Pdt\_02 | 33.8 |

现在需要求出每一个订单中最贵的商品。

（1）输入数据



（2）期望输出数据

1 222.8

2 722.4

3 232.8

2．需求分析

（1）利用“订单id和成交金额”作为key，可以将Map阶段读取到的所有订单数据按照id升序排序，如果id相同再按照金额降序排序，发送到Reduce。

（2）在Reduce端利用groupingComparator将订单id相同的kv聚合成组，然后取第一个即是该订单中最贵商品，如图4-18所示。

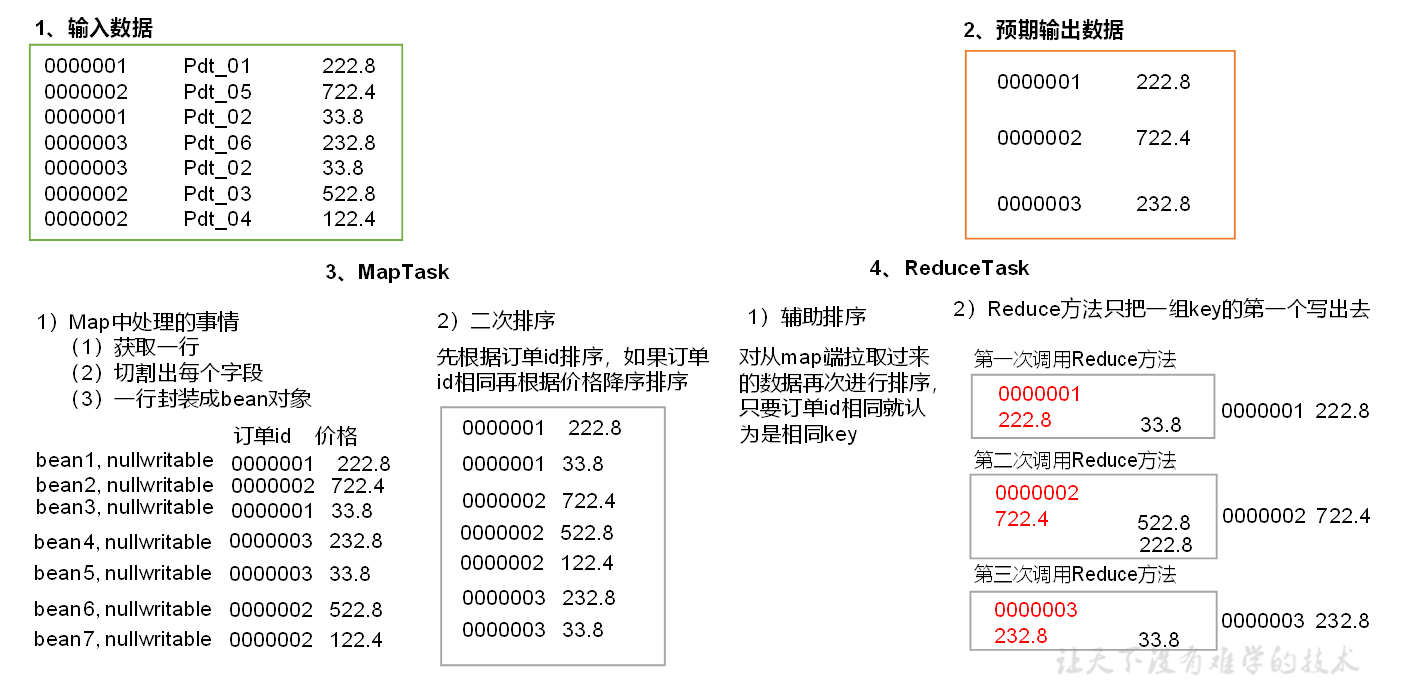


图4-18 过程分析

3．代码实现

（1）定义订单信息OrderBean类

|  |
| --- |
| package com.atguigu.mapreduce.order;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.WritableComparable;  public class OrderBean implements WritableComparable<OrderBean> {  private int order\_id; // 订单id号  private double price; // 价格  public OrderBean() {  super();  }  public OrderBean(int order\_id, double price) {  super();  this.order\_id = order\_id;  this.price = price;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeInt(order\_id);  out.writeDouble(price);  }  @Override  public void readFields(DataInput in) throws IOException {  order\_id = in.readInt();  price = in.readDouble();  }  @Override  public String toString() {  return order\_id + "\t" + price;  }  public int getOrder\_id() {  return order\_id;  }  public void setOrder\_id(int order\_id) {  this.order\_id = order\_id;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  // 二次排序  @Override  public int compareTo(OrderBean o) {  int result;  if (order\_id > o.getOrder\_id()) {  result = 1;  } else if (order\_id < o.getOrder\_id()) {  result = -1;  } else {  // 价格倒序排序  result = price > o.getPrice() ? -1 : 1;  }  return result;  }  } |

（2）编写OrderSortMapper类

|  |
| --- |
| package com.atguigu.mapreduce.order;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class OrderMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable> {  OrderBean k = new OrderBean();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 封装对象  k.setOrder\_id(Integer.parseInt(fields[0]));  k.setPrice(Double.parseDouble(fields[2]));    // 4 写出  context.write(k, NullWritable.get());  }  } |

（3）编写OrderSortGroupingComparator类

|  |
| --- |
| package com.atguigu.mapreduce.order;  import org.apache.hadoop.io.WritableComparable;  import org.apache.hadoop.io.WritableComparator;  public class OrderGroupingComparator extends WritableComparator {  protected OrderGroupingComparator() {  super(OrderBean.class, true);  }  @Override  public int compare(WritableComparable a, WritableComparable b) {  OrderBean aBean = (OrderBean) a;  OrderBean bBean = (OrderBean) b;  int result;  if (aBean.getOrder\_id() > bBean.getOrder\_id()) {  result = 1;  } else if (aBean.getOrder\_id() < bBean.getOrder\_id()) {  result = -1;  } else {  result = 0;  }  return result;  }  } |

（4）编写OrderSortReducer类

|  |
| --- |
| package com.atguigu.mapreduce.order;  import java.io.IOException;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.mapreduce.Reducer;  public class OrderReducer 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());  }  } |

（5）编写OrderSortDriver类

|  |
| --- |
| package com.atguigu.mapreduce.order;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class OrderDriver {  public static void main(String[] args) throws Exception, IOException {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[]{"e:/input/inputorder" , "e:/output1"};  // 1 获取配置信息  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  // 2 设置jar包加载路径  job.setJarByClass(OrderDriver.class);  // 3 加载map/reduce类  job.setMapperClass(OrderMapper.class);  job.setReducerClass(OrderReducer.class);  // 4 设置map输出数据key和value类型  job.setMapOutputKeyClass(OrderBean.class);  job.setMapOutputValueClass(NullWritable.class);  // 5 设置最终输出数据的key和value类型  job.setOutputKeyClass(OrderBean.class);  job.setOutputValueClass(NullWritable.class);  // 6 设置输入数据和输出数据路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 8 设置reduce端的分组  job.setGroupingComparatorClass(OrderGroupingComparator.class);  // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

## 3.4 MapTask工作机制

MapTask工作机制如图4-12所示。

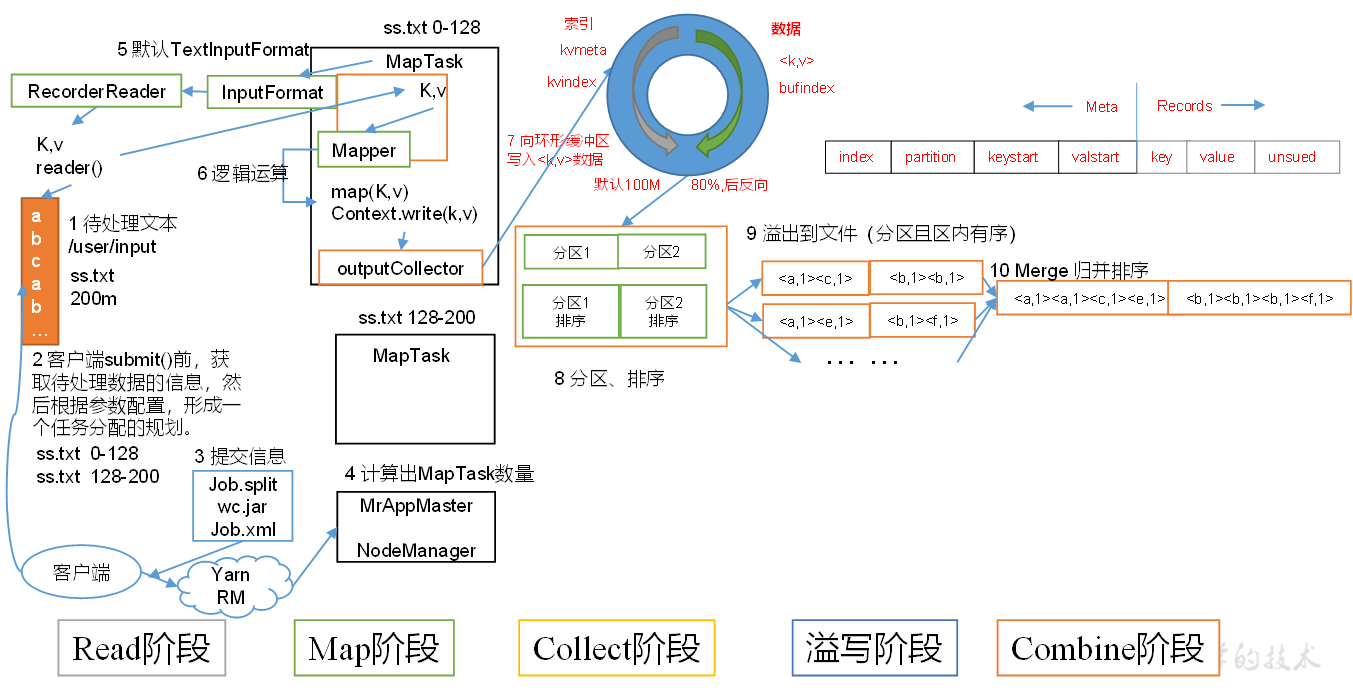


图4-12 MapTask工作机制

（1）Read阶段：MapTask通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value。

（2）Map阶段：该节点主要是将解析出的key/value交给用户编写map()函数处理，并产生一系列新的key/value。

（3）Collect收集阶段：在用户编写map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它会将生成的key/value分区（调用Partitioner），并写入一个环形内存缓冲区中。

（4）Spill阶段：即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。

溢写阶段详情：

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

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

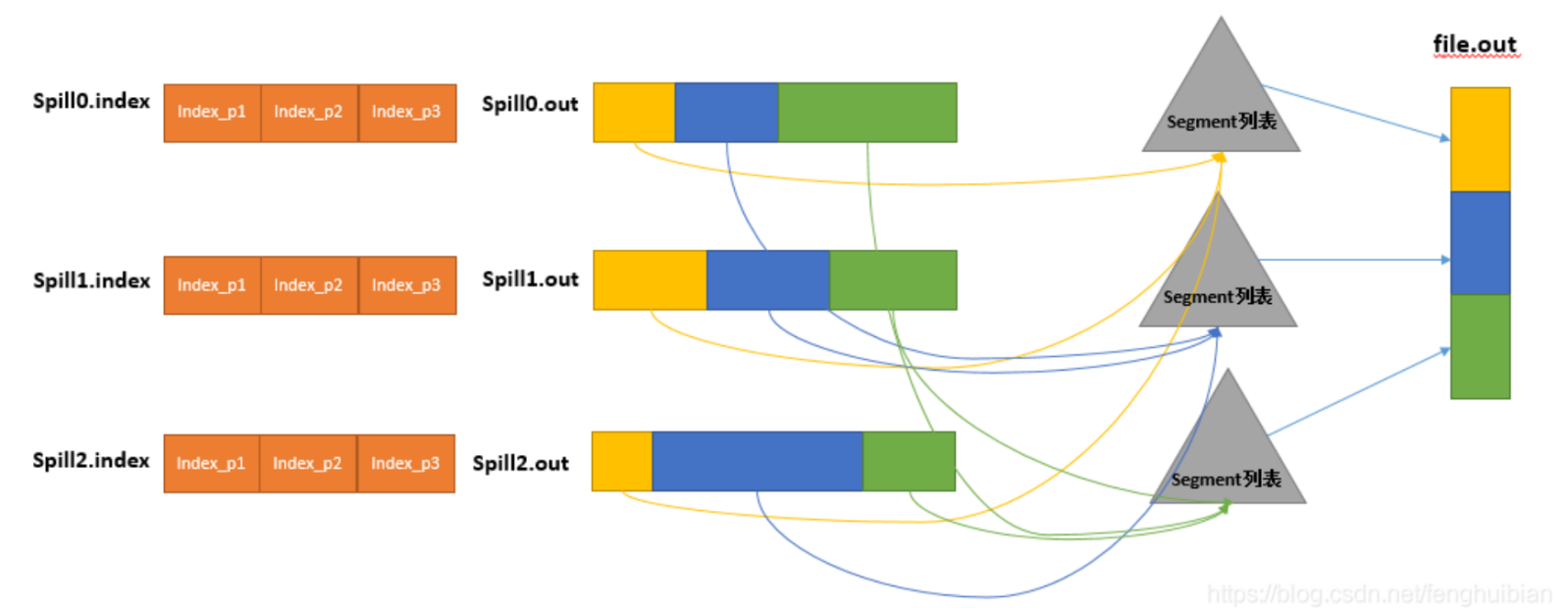
步骤3：将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小超过1MB，则将内存索引写到文件output/spillN.out.index中。

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

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

在进行文件合并过程中，MapTask以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并io.sort.factor（默认10）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

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



## 3.5 ReduceTask工作机制

1．ReduceTask工作机制

ReduceTask工作机制，如图4-19所示。

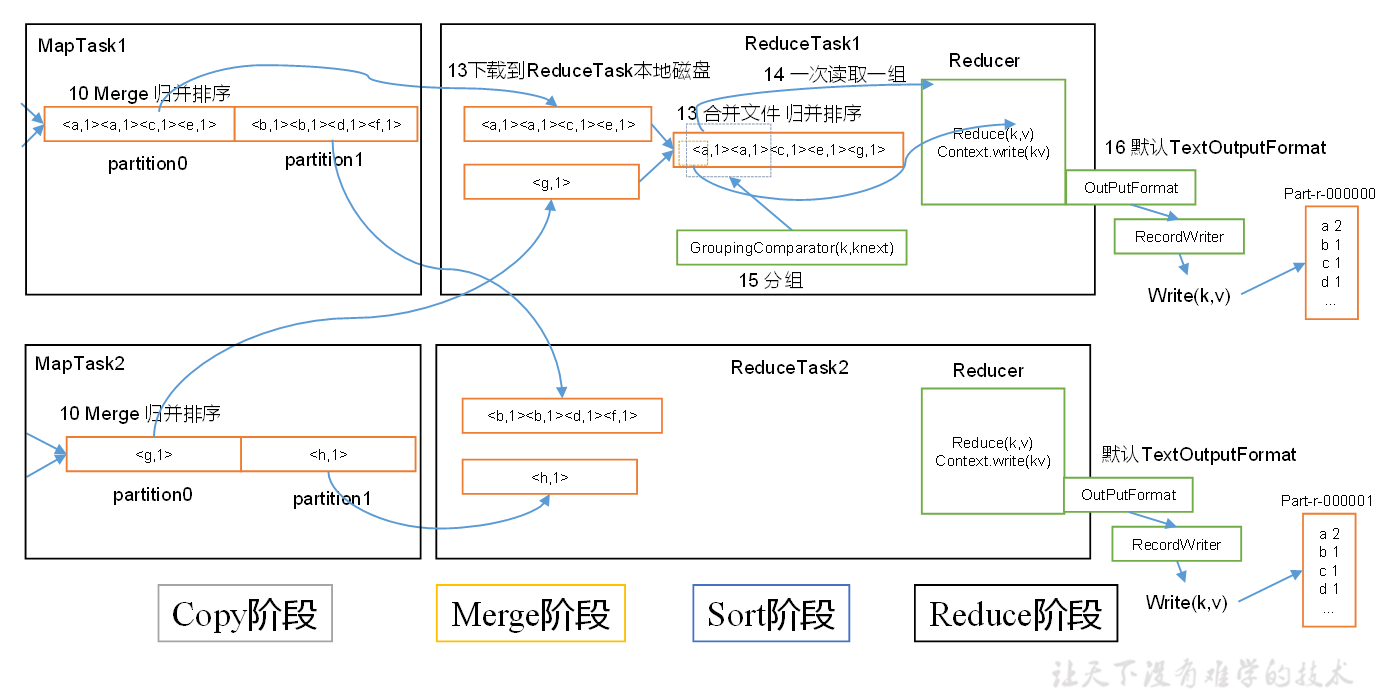


图4-19 ReduceTask工作机制

（1）Copy阶段：ReduceTask从各个MapTask上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

（2）Merge阶段：在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

（3）Sort阶段：按照MapReduce语义，用户编写reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。

（4）Reduce阶段：reduce()函数将计算结果写到HDFS上。

2．设置ReduceTask并行度（个数）

ReduceTask的并行度同样影响整个Job的执行并发度和执行效率，但与MapTask的并发数由切片数决定不同，ReduceTask数量的决定是可以直接手动设置：

// 默认值是1，手动设置为4

job.setNumReduceTasks(4);

3．实验：测试ReduceTask多少合适

（1）实验环境：1个Master节点，16个Slave节点：CPU:8GHZ，内存: 2G

（2）实验结论：

表4-3 改变ReduceTask （数据量为1GB）

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

1. 注意事项
2. ReduceTask=0，表示没有Reduce阶段，输出文件个数和Map个数一致。
3. ReduceTask默认值就是1，所以输出文件个数为1
4. 如果数据分布不均匀，就有可能在Reduce阶段产生数据倾斜
5. ReduceTask数量设置需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。
6. 具体多少个ReduceTask，需要根据集群性能而定。
7. 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，执行分区的前提时先判断ReduceNum个数是否大于1。

## 3.6 OutputFormat数据输出

### 3.6.1 OutputFormat接口实现类

OutputFormat是MapReduce输出的基类，所有实现MapReduce输出都实现了OutputFormat接口。

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。

### 3.6.2 自定义OutputFormat

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()。

### 3.6.3 自定义OutputFormat案例实操

1．需求

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

（1）输入数据



（2）期望输出数据

2．需求分析



3．案例实操

（1）编写FilterMapper类

|  |
| --- |
| package com.atguigu.mapreduce.outputformat;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FilterMapper extends Mapper<LongWritable, Text, Text, NullWritable>{    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 写出  context.write(value, NullWritable.get());  }  } |

（2）编写FilterReducer类

|  |
| --- |
| package com.atguigu.mapreduce.outputformat;  import java.io.IOException;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FilterReducer extends Reducer<Text, NullWritable, Text, NullWritable> {  Text k = new Text();  @Override  protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {  // 1 获取一行  String line = key.toString();  // 2 拼接  line = line + "\r\n";  // 3 设置key  k.set(line);  // 4 输出  context.write(k, NullWritable.get());  }  } |

（3）自定义一个OutputFormat类

|  |
| --- |
| package com.atguigu.mapreduce.outputformat;  import java.io.IOException;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.RecordWriter;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FilterOutputFormat extends FileOutputFormat<Text, NullWritable>{  @Override  public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext job) throws IOException, InterruptedException {  // 创建一个RecordWriter  return new FilterRecordWriter(job);  }  } |

（4）编写RecordWriter类

|  |
| --- |
| package com.atguigu.mapreduce.outputformat;  import java.io.IOException;  import org.apache.hadoop.fs.FSDataOutputStream;  import org.apache.hadoop.fs.FileSystem;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.RecordWriter;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  public class FilterRecordWriter extends RecordWriter<Text, NullWritable> {  FSDataOutputStream atguiguOut = null;  FSDataOutputStream otherOut = null;  public FilterRecordWriter(TaskAttemptContext job) {  // 1 获取文件系统  FileSystem fs;  try {  fs = FileSystem.get(job.getConfiguration());  // 2 创建输出文件路径  Path atguiguPath = new Path("e:/atguigu.log");  Path otherPath = new Path("e:/other.log");  // 3 创建输出流  atguiguOut = fs.create(atguiguPath);  otherOut = fs.create(otherPath);  } catch (IOException e) {  e.printStackTrace();  }  }  @Override  public void write(Text key, NullWritable value) throws IOException, InterruptedException {  // 判断是否包含“atguigu”输出到不同文件  if (key.toString().contains("atguigu")) {  atguiguOut.write(key.toString().getBytes());  } else {  otherOut.write(key.toString().getBytes());  }  }  @Override  public void close(TaskAttemptContext context) throws IOException, InterruptedException {  // 关闭资源  IOUtils.closeStream(atguiguOut);  IOUtils.closeStream(otherOut);  }  } |

（5）编写FilterDriver类

|  |
| --- |
| package com.atguigu.mapreduce.outputformat;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FilterDriver {  public static void main(String[] args) throws Exception {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputoutputformat", "e:/output2" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(FilterDriver.class);  job.setMapperClass(FilterMapper.class);  job.setReducerClass(FilterReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(NullWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 要将自定义的输出格式组件设置到job中  job.setOutputFormatClass(FilterOutputFormat.class);    FileInputFormat.setInputPaths(job, new Path(args[0]));  // 虽然我们自定义了outputformat，但是因为我们的outputformat继承自fileoutputformat  // 而fileoutputformat要输出一个\_SUCCESS文件，所以，在这还得指定一个输出目录  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

## 3.7 Join多种应用

### 3.7.1 Reduce Join

Map端的主要工作：为来自不同表或文件的key/value对，打标签以区别不同来源的记录。然后用连接字段作为key，其余部分和新加的标志作为value，最后进行输出。

Reduce端的主要工作：在Reduce端以连接字段作为key的分组已经完成，我们只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，最后进行合并就可以了。

### 3.7.2 Reduce Join案例实操

1．需求

表4-4 订单数据表t\_order

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

表4-5 商品信息表t\_product



|  |  |
| --- | --- |
| pid | pname |
| 01 | 小米 |
| 02 | 华为 |
| 03 | 格力 |

将商品信息表中数据根据商品pid合并到订单数据表中。

表4-6 最终数据形式

|  |  |  |
| --- | --- | --- |
| id | pname | amount |
| 1001 | 小米 | 1 |
| 1004 | 小米 | 4 |
| 1002 | 华为 | 2 |
| 1005 | 华为 | 5 |
| 1003 | 格力 | 3 |
| 1006 | 格力 | 6 |

2．需求分析

通过将关联条件作为Map输出的key，将两表满足Join条件的数据并携带数据所来源的文件信息，发往同一个ReduceTask，在Reduce中进行数据的串联，如图4-20所示。

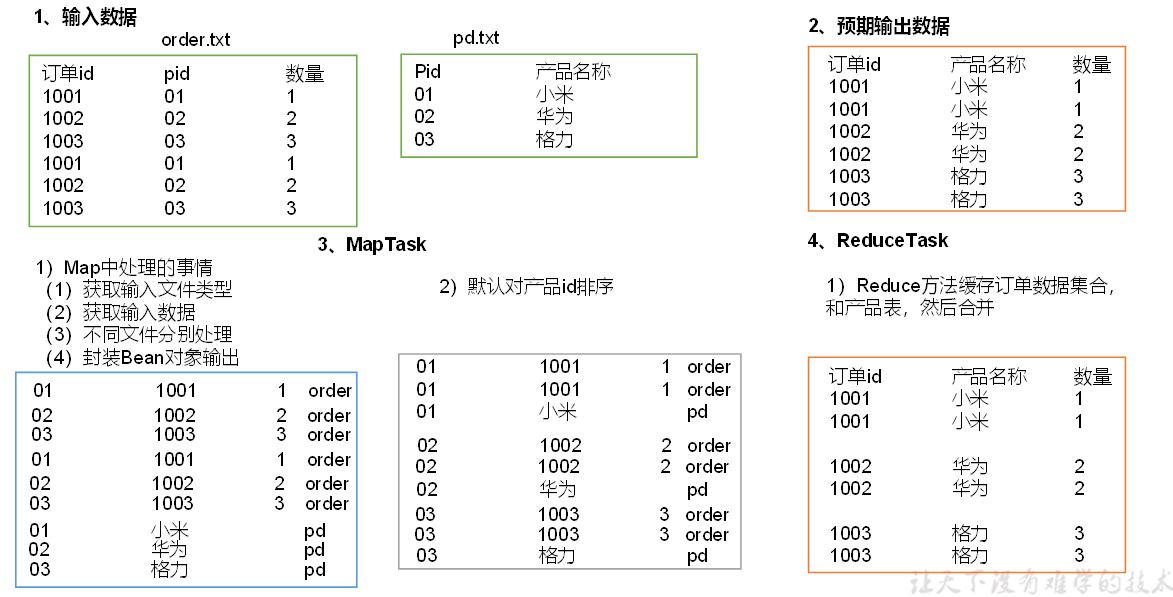


图4-20 Reduce端表合并

3．代码实现

1）创建商品和订合并后的Bean类

|  |
| --- |
| package com.atguigu.mapreduce.table;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  public class TableBean implements Writable {  private String order\_id; // 订单id  private String p\_id; // 产品id  private int amount; // 产品数量  private String pname; // 产品名称  private String flag; // 表的标记  public TableBean() {  super();  }  public TableBean(String order\_id, String p\_id, int amount, String pname, String flag) {  super();  this.order\_id = order\_id;  this.p\_id = p\_id;  this.amount = amount;  this.pname = pname;  this.flag = flag;  }  public String getFlag() {  return flag;  }  public void setFlag(String flag) {  this.flag = flag;  }  public String getOrder\_id() {  return order\_id;  }  public void setOrder\_id(String order\_id) {  this.order\_id = order\_id;  }  public String getP\_id() {  return p\_id;  }  public void setP\_id(String p\_id) {  this.p\_id = p\_id;  }  public int getAmount() {  return amount;  }  public void setAmount(int amount) {  this.amount = amount;  }  public String getPname() {  return pname;  }  public void setPname(String pname) {  this.pname = pname;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(order\_id);  out.writeUTF(p\_id);  out.writeInt(amount);  out.writeUTF(pname);  out.writeUTF(flag);  }  @Override  public void readFields(DataInput in) throws IOException {  this.order\_id = in.readUTF();  this.p\_id = in.readUTF();  this.amount = in.readInt();  this.pname = in.readUTF();  this.flag = in.readUTF();  }  @Override  public String toString() {  return order\_id + "\t" + pname + "\t" + amount + "\t" ;  }  } |

2）编写TableMapper类

|  |
| --- |
| package com.atguigu.mapreduce.table;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.lib.input.FileSplit;  public class TableMapper extends Mapper<LongWritable, Text, Text, TableBean>{  String name;  TableBean bean = new TableBean();  Text k = new Text();    @Override  protected void setup(Context context) throws IOException, InterruptedException {  // 1 获取输入文件切片  FileSplit split = (FileSplit) context.getInputSplit();  // 2 获取输入文件名称  name = split.getPath().getName();  }  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取输入数据  String line = value.toString();  // 2 不同文件分别处理  if (name.startsWith("order")) {// 订单表处理  // 2.1 切割  String[] fields = line.split("\t");  // 2.2 封装bean对象  bean.setOrder\_id(fields[0]);  bean.setP\_id(fields[1]);  bean.setAmount(Integer.parseInt(fields[2]));  bean.setPname("");  bean.setFlag("order");    k.set(fields[1]);  }else {  // 产品表处理  // 2.3 切割  String[] fields = line.split("\t");  // 2.4 封装bean对象  bean.setP\_id(fields[0]);  bean.setPname(fields[1]);  bean.setFlag("pd");  bean.setAmount(0);  bean.setOrder\_id("");    k.set(fields[0]);  }  // 3 写出  context.write(k, bean);  }  } |

3）编写TableReducer类

|  |
| --- |
| package com.atguigu.mapreduce.table;  import java.io.IOException;  import java.util.ArrayList;  import org.apache.commons.beanutils.BeanUtils;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TableReducer extends Reducer<Text, TableBean, TableBean, NullWritable> {  @Override  protected void reduce(Text key, Iterable<TableBean> values, Context context) throws IOException, InterruptedException {  // 1准备存储订单的集合  ArrayList<TableBean> orderBeans = new ArrayList<>();  // 2 准备bean对象  TableBean pdBean = new TableBean();  for (TableBean bean : values) {  if ("order".equals(bean.getFlag())) {// 订单表  // 拷贝传递过来的每条订单数据到集合中  TableBean orderBean = new TableBean();  try {  BeanUtils.copyProperties(orderBean, bean); //bean对象拷贝到orderBean中  } catch (Exception e) {  e.printStackTrace();  }  orderBeans.add(orderBean); // orderBean拷贝到orderBeans  } else {// 产品表  try {  // 拷贝传递过来的产品表到内存中  BeanUtils.copyProperties(pdBean, bean); //bean对象拷贝到pdBean中  } catch (Exception e) {  e.printStackTrace();  }  }  }  // 3 表的拼接  for(TableBean bean:orderBeans){  bean.setPname(pdBean.getPname());  // 4 数据写出去  context.write(bean, NullWritable.get());  }  }  } |

4）编写TableDriver类

|  |
| --- |
| package com.atguigu.mapreduce.table;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TableDriver {  public static void main(String[] args) throws Exception {  // 0 根据自己电脑路径重新配置  args = new String[]{"e:/input/inputtable","e:/output1"};  // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 指定本程序的jar包所在的本地路径  job.setJarByClass(TableDriver.class);  // 3 指定本业务job要使用的Mapper/Reducer业务类  job.setMapperClass(TableMapper.class);  job.setReducerClass(TableReducer.class);  // 4 指定Mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(TableBean.class);  // 5 指定最终输出的数据的kv类型  job.setOutputKeyClass(TableBean.class);  job.setOutputValueClass(NullWritable.class);  // 6 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

4．测试

运行程序查看结果

|  |
| --- |
| 1001 小米 1  1001 小米 1  1002 华为 2  1002 华为 2  1003 格力 3  1003 格力 3 |

5．总结

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

解决方式：Map端实现数据合并

### 3.7.3 Map Join

1．使用场景

Map Join适用于一张表十分小、一张表很大的场景。

2．优点

思考：在Reduce端处理过多的表，非常容易产生数据倾斜。怎么办？

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

3．具体办法：采用DistributedCache

（1）在Mapper的setup阶段，将文件读取到缓存集合中。

（2）在驱动函数中加载缓存。

// 缓存普通文件到Task运行节点。

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

### 3.7.4 Map Join案例实操

1．需求

表4-4 订单数据表t\_order



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

表4-5 商品信息表t\_product

|  |  |
| --- | --- |
| pid | pname |
| 01 | 小米 |
| 02 | 华为 |
| 03 | 格力 |

将商品信息表中数据根据商品pid合并到订单数据表中。



表4-6 最终数据形式

|  |  |  |
| --- | --- | --- |
| id | pname | amount |
| 1001 | 小米 | 1 |
| 1004 | 小米 | 4 |
| 1002 | 华为 | 2 |
| 1005 | 华为 | 5 |
| 1003 | 格力 | 3 |
| 1006 | 格力 | 6 |

2．需求分析

MapJoin适用于关联表中有小表的情形。

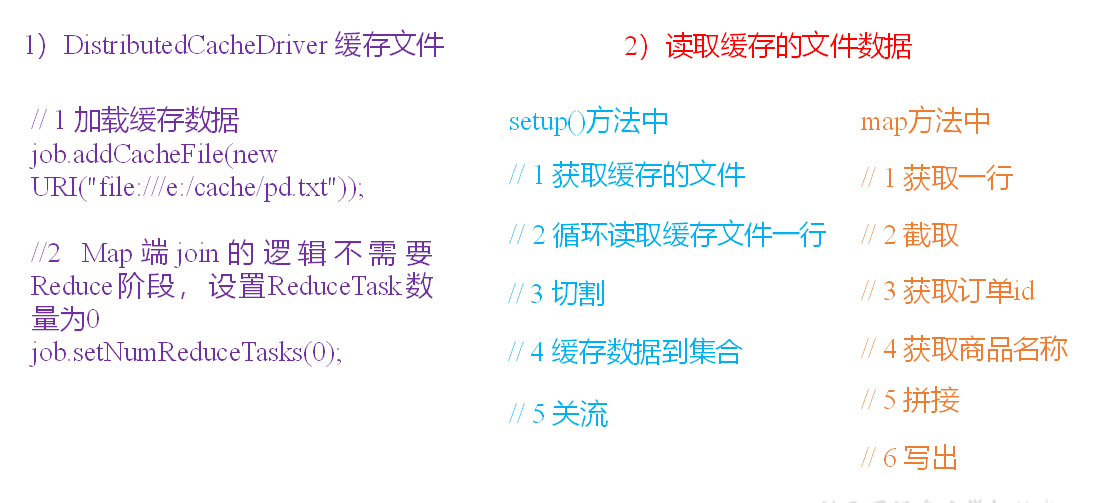


图4-21 Map端表合并

3．实现代码

（1）先在驱动模块中添加缓存文件

|  |
| --- |
| package test;  import java.net.URI;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class DistributedCacheDriver {  public static void main(String[] args) throws Exception {    // 0 根据自己电脑路径重新配置  args = new String[]{"e:/input/inputtable2", "e:/output1"};  // 1 获取job信息  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置加载jar包路径  job.setJarByClass(DistributedCacheDriver.class);  // 3 关联map  job.setMapperClass(DistributedCacheMapper.class);    // 4 设置最终输出数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 5 设置输入输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 加载缓存数据  job.addCacheFile(new URI("file:///e:/input/inputcache/pd.txt"));    // 7 Map端Join的逻辑不需要Reduce阶段，设置reduceTask数量为0  job.setNumReduceTasks(0);  // 8 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

（2）读取缓存的文件数据

|  |
| --- |
| package test;  import java.io.BufferedReader;  import java.io.FileInputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.util.HashMap;  import java.util.Map;  import org.apache.commons.lang.StringUtils;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class DistributedCacheMapper extends Mapper<LongWritable, Text, Text, NullWritable>{  Map<String, String> pdMap = new HashMap<>();    @Override  protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context) throws IOException, InterruptedException {  // 1 获取缓存的文件  URI[] cacheFiles = context.getCacheFiles();  String path = cacheFiles[0].getPath().toString();  BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(path), "UTF-8"));  String line;  while(StringUtils.isNotEmpty(line = reader.readLine())){  // 2 切割  String[] fields = line.split("\t");  // 3 缓存数据到集合  pdMap.put(fields[0], fields[1]);  }  // 4 关流  reader.close();  }  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();  // 2 截取  String[] fields = line.split("\t");  // 3 获取产品id  String pId = fields[1];  // 4 获取商品名称  String pdName = pdMap.get(pId);  // 5 拼接  k.set(line + "\t"+ pdName);  // 6 写出  context.write(k, NullWritable.get());  }  } |

## 3.8 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
2. 采用枚举类的方式统计计数

enum MyCounter{MALFORORMED,NORMAL}

对枚举定义的自定义计数器加1

context.getCounter(MyCounter.MALFORORMED).increment(1);

1. 采用计数器组、计数器名称的方式统计

context.getCounter(“counterGroup”, “counter”).increment(1);

组名和计数器名随便起，但要有意义

1. 计数结果在程序运行后的控制台上查看。

2、计数器案例实操 （详见数据清洗案例）

## 3.9 数据清洗（ETL）

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

### 3.9.1 数据清洗案例实操-简单解析版

1．需求

去除日志中字段长度小于等于11的日志。

（1）输入数据



（2）期望输出数据

每行字段长度都大于11。

2．需求分析

需要在Map阶段对输入的数据根据规则进行过滤清洗。

3．实现代码

（1）编写LogMapper类

|  |
| --- |
| package com.atguigu.mapreduce.weblog;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class LogMapper extends Mapper<LongWritable, Text, Text, NullWritable>{    Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取1行数据  String line = value.toString();  // 2 解析日志  boolean result = parseLog(line,context);  // 3 日志不合法退出  if (!result) {  return;  }  // 4 设置key  k.set(line);  // 5 写出数据  context.write(k, NullWritable.get());  }  // 2 解析日志  private boolean parseLog(String line, Context context) {  // 1 截取  String[] fields = line.split(" ");  // 2 日志长度大于11的为合法  if (fields.length > 11) {  // 系统计数器  context.getCounter("map", "true").increment(1);  return true;  }else {  context.getCounter("map", "false").increment(1);  return false;  }  }  } |

（2）编写LogDriver类

|  |
| --- |
| package com.atguigu.mapreduce.weblog;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class LogDriver {  public static void main(String[] args) throws Exception {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputlog", "e:/output1" };  // 1 获取job信息  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  // 2 加载jar包  job.setJarByClass(LogDriver.class);  // 3 关联map  job.setMapperClass(LogMapper.class);  // 4 设置最终输出类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 设置reducetask个数为0  job.setNumReduceTasks(0);  // 5 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 提交  job.waitForCompletion(true);  }  } |

### 3.9.2 数据清洗案例实操-复杂解析版

1．需求

对Web访问日志中的各字段识别切分，去除日志中不合法的记录。根据清洗规则，输出过滤后的数据。

（1）输入数据



（2）期望输出数据

都是合法的数据

2．实现代码

（1）定义一个bean，用来记录日志数据中的各数据字段

|  |
| --- |
| package com.atguigu.mapreduce.log;  public class LogBean {  private String remote\_addr;// 记录客户端的ip地址  private String remote\_user;// 记录客户端用户名称,忽略属性"-"  private String time\_local;// 记录访问时间与时区  private String request;// 记录请求的url与http协议  private String status;// 记录请求状态；成功是200  private String body\_bytes\_sent;// 记录发送给客户端文件主体内容大小  private String http\_referer;// 用来记录从那个页面链接访问过来的  private String http\_user\_agent;// 记录客户浏览器的相关信息  private boolean valid = true;// 判断数据是否合法  public String getRemote\_addr() {  return remote\_addr;  }  public void setRemote\_addr(String remote\_addr) {  this.remote\_addr = remote\_addr;  }  public String getRemote\_user() {  return remote\_user;  }  public void setRemote\_user(String remote\_user) {  this.remote\_user = remote\_user;  }  public String getTime\_local() {  return time\_local;  }  public void setTime\_local(String time\_local) {  this.time\_local = time\_local;  }  public String getRequest() {  return request;  }  public void setRequest(String request) {  this.request = request;  }  public String getStatus() {  return status;  }  public void setStatus(String status) {  this.status = status;  }  public String getBody\_bytes\_sent() {  return body\_bytes\_sent;  }  public void setBody\_bytes\_sent(String body\_bytes\_sent) {  this.body\_bytes\_sent = body\_bytes\_sent;  }  public String getHttp\_referer() {  return http\_referer;  }  public void setHttp\_referer(String http\_referer) {  this.http\_referer = http\_referer;  }  public String getHttp\_user\_agent() {  return http\_user\_agent;  }  public void setHttp\_user\_agent(String http\_user\_agent) {  this.http\_user\_agent = http\_user\_agent;  }  public boolean isValid() {  return valid;  }  public void setValid(boolean valid) {  this.valid = valid;  }  @Override  public String toString() {  StringBuilder sb = new StringBuilder();  sb.append(this.valid);  sb.append("\001").append(this.remote\_addr);  sb.append("\001").append(this.remote\_user);  sb.append("\001").append(this.time\_local);  sb.append("\001").append(this.request);  sb.append("\001").append(this.status);  sb.append("\001").append(this.body\_bytes\_sent);  sb.append("\001").append(this.http\_referer);  sb.append("\001").append(this.http\_user\_agent);    return sb.toString();  }  } |

（2）编写LogMapper类

|  |
| --- |
| package com.atguigu.mapreduce.log;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class LogMapper extends Mapper<LongWritable, Text, Text, NullWritable>{  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取1行  String line = value.toString();  // 2 解析日志是否合法  LogBean bean = parseLog(line);  if (!bean.isValid()) {  return;  }  k.set(bean.toString());  // 3 输出  context.write(k, NullWritable.get());  }  // 解析日志  private LogBean parseLog(String line) {  LogBean logBean = new LogBean();  // 1 截取  String[] fields = line.split(" ");  if (fields.length > 11) {  // 2封装数据  logBean.setRemote\_addr(fields[0]);  logBean.setRemote\_user(fields[1]);  logBean.setTime\_local(fields[3].substring(1));  logBean.setRequest(fields[6]);  logBean.setStatus(fields[8]);  logBean.setBody\_bytes\_sent(fields[9]);  logBean.setHttp\_referer(fields[10]);  if (fields.length > 12) {  logBean.setHttp\_user\_agent(fields[11] + " "+ fields[12]);  }else {  logBean.setHttp\_user\_agent(fields[11]);  }    // 大于400，HTTP错误  if (Integer.parseInt(logBean.getStatus()) >= 400) {  logBean.setValid(false);  }  }else {  logBean.setValid(false);  }  return logBean;  }  } |

（3）编写LogDriver类

|  |
| --- |
| package com.atguigu.mapreduce.log;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class LogDriver {  public static void main(String[] args) throws Exception {  // 1 获取job信息  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  // 2 加载jar包  job.setJarByClass(LogDriver.class);  // 3 关联map  job.setMapperClass(LogMapper.class);  // 4 设置最终输出类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 5 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 提交  job.waitForCompletion(true);  }  } |

## 3.10 MapReduce开发总结

在编写MapReduce程序时，需要考虑如下几个方面：

1. 输入数据接口：InputFormat
2. 默认使用的实现类是TextInputFormat
3. 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，行内容为value返回。
4. 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)。
5. NlineInputFormat按照指定的行数N来划分切片。
6. CombineTextInputFormat可以把多个小文件合并成一个切片处理，提高处理效率。
7. 用户还可以自定义InputFormat
8. 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()

1. Partition分区
2. 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）如果业务有别的要求，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，但是使用时不能影响原有的业务处理结果。

1. Reduce端分组：GroupingComparator

在Reduce端对key进行分组。应用在接收的key为bean对象时，想让一个或者几个字段相同（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。

1. 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()

1. 输出数据接口：OutputFormat
2. 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。
3. 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。

# 第4章 Hadoop数据压缩

## 4.1 概述

**1、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。不过尽管压缩和解压操作的CPU开销不高，其性能的提升和资源的节省并非没有代价。

1. **压缩策略和原则**

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，运用不当降低性能。

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

## 4.2 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 |

为了支持多种压缩/解压缩算法，Hadoop引入了编码/解码器，如下表所示。

表4-8

|  |  |
| --- | --- |
| 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.

## 4.3 压缩方式选择

### 4.3.1 Gzip压缩

优点：压缩率比较高，而且压缩/解压速度也快，Hadoop本身支持，在应用中处理Gzip格式的文件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（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。

### 4.3.2 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得情况；或者对单个很大得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。

### 4.3.3 Lzo压缩

优点：压缩/解压速度比较快，合理的压缩率；支持split，也是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要建索引，还要指定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优点越明显。

### 4.3.4 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作业的输出和另一个MapReduce作业的输入。

## 4.4 压缩位置选择

压缩可以在MapReduce作用的任意阶段启用，如图4-22所示。

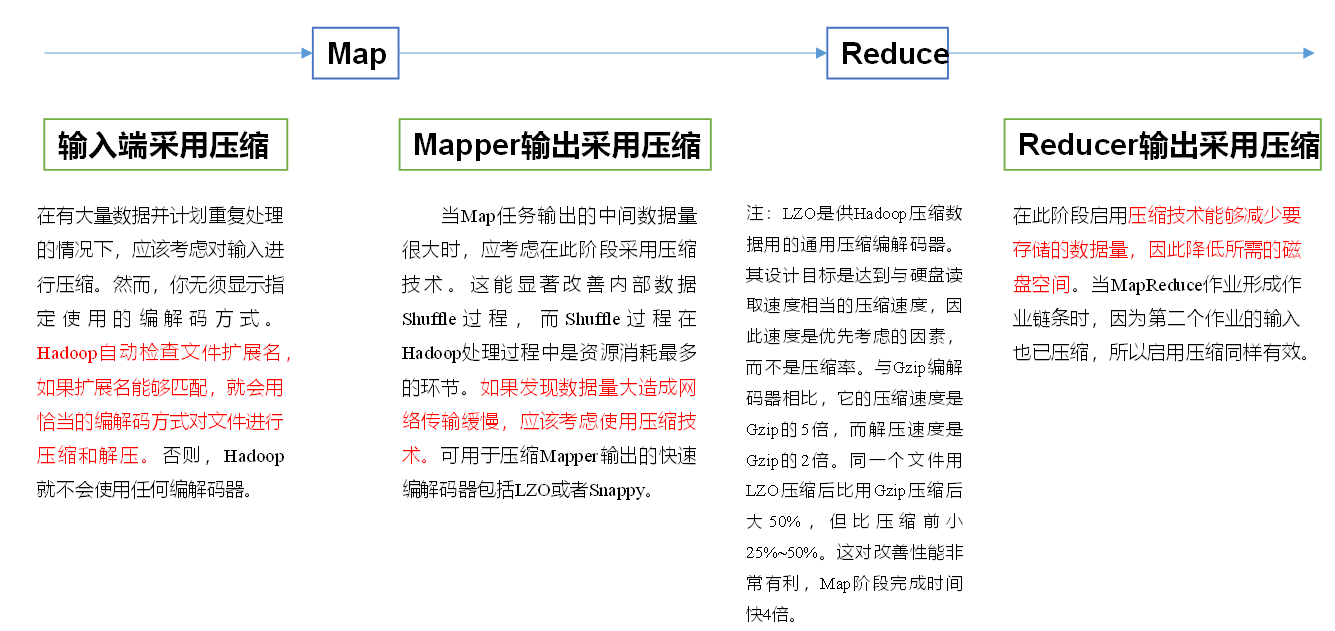


图4-22 MapReduce数据压缩

## 4.5 压缩参数配置

要在Hadoop中启用压缩，可以配置如下参数：

表4-10 配置参数

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 默认值 | 阶段 | 建议 |
| io.compression.codecs  （在core-site.xml中配置） | org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec | 输入压缩 | Hadoop使用文件扩展名判断是否支持某种编解码器 |
| mapreduce.map.output.compress（在mapred-site.xml中配置） | false | mapper输出 | 这个参数设为true启用压缩 |
| mapreduce.map.output.compress.codec（在mapred-site.xml中配置） | org.apache.hadoop.io.compress.DefaultCodec | mapper输出 | 企业多使用LZO或Snappy编解码器在此阶段压缩数据 |
| mapreduce.output.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BLOCK |

## 4.6 压缩实操案例

### 4.6.1 数据流的压缩和解压缩

CompressionCodes有两个方法可以用于轻松地压缩或解压缩数据。

想要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，想要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：

表4-11

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

package com.atguigu.mapreduce.compress;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IOUtils;

import org.apache.hadoop.io.compress.CompressionCodec;

import org.apache.hadoop.io.compress.CompressionCodecFactory;

import org.apache.hadoop.io.compress.CompressionInputStream;

import org.apache.hadoop.io.compress.CompressionOutputStream;

import org.apache.hadoop.util.ReflectionUtils;

public class TestCompress {

public static void main(String[] args) throws Exception {

compress("e:/hello.txt","org.apache.hadoop.io.compress.BZip2Codec");

// decompress("e:/hello.txt.bz2");

}

// 1、压缩

private static void compress(String filename, String method) throws Exception {

// （1）获取输入流

FileInputStream fis = new FileInputStream(new File(filename));

Class codecClass = Class.forName(method);

CompressionCodec codec = (CompressionCodec)ReflectionUtils.newInstance(codecClass,new Configuration());

// （2）获取输出流

FileOutputStream fos = new FileOutputStream(new File(filename + codec.getDefaultExtension()));

CompressionOutputStream cos = codec.createOutputStream(fos);

// （3）流的对拷

IOUtils.copyBytes(fis, cos, 1024\*1024\*5, false);

// （4）关闭资源

cos.close();

fos.close();

fis.close();

}

// 2、解压缩

private static void decompress(String filename) throws FileNotFoundException, IOException {

// （0）校验是否能解压缩

CompressionCodecFactory factory = new CompressionCodecFactory(new Configuration());

CompressionCodec codec = factory.getCodec(new Path(filename));

if (codec == null) {

System.out.println("cannot find codec for file " + filename);

return;

}

// （1）获取输入流

CompressionInputStream cis = codec.createInputStream(new FileInputStream(new File(filename)));

// （2）获取输出流

FileOutputStream fos = new FileOutputStream(new File(filename + ".decoded"));

// （3）流的对拷

IOUtils.copyBytes(cis, fos, 1024\*1024\*5, false);

// （4）关闭资源

cis.close();

fos.close();

}

}

### 4.6.2 Map输出端采用压缩

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

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

|  |
| --- |
| package com.atguigu.mapreduce.compress;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.compress.BZip2Codec;  import org.apache.hadoop.io.compress.CompressionCodec;  import org.apache.hadoop.io.compress.GzipCodec;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class WordCountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  Configuration configuration = new Configuration();  // 开启map端输出压缩  configuration.setBoolean("mapreduce.map.output.compress", true);  // 设置map端输出压缩方式  configuration.setClass("mapreduce.map.output.compress.codec", BZip2Codec.class, CompressionCodec.class);  Job job = Job.getInstance(configuration);  job.setJarByClass(WordCountDriver.class);  job.setMapperClass(WordCountMapper.class);  job.setReducerClass(WordCountReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result ? 1 : 0);  }  } |

2．Mapper保持不变

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

3．Reducer保持不变

|  |
| --- |
| package com.atguigu.mapreduce.compress;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class WordCountReducer 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;  // 1 汇总  for(IntWritable value:values){  sum += value.get();  }  v.set(sum);  // 2 输出  context.write(key, v);  }  } |

### 4.6.3 Reduce输出端采用压缩

基于WordCount案例处理。

1．修改驱动

|  |
| --- |
| package com.atguigu.mapreduce.compress;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.compress.BZip2Codec;  import org.apache.hadoop.io.compress.DefaultCodec;  import org.apache.hadoop.io.compress.GzipCodec;  import org.apache.hadoop.io.compress.Lz4Codec;  import org.apache.hadoop.io.compress.SnappyCodec;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class WordCountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  job.setJarByClass(WordCountDriver.class);    job.setMapperClass(WordCountMapper.class);  job.setReducerClass(WordCountReducer.class);    job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);    FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 设置reduce端输出压缩开启  FileOutputFormat.setCompressOutput(job, true);    // 设置压缩的方式  FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);  //FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);  //FileOutputFormat.setOutputCompressorClass(job, DefaultCodec.class);    boolean result = job.waitForCompletion(true);    System.exit(result?1:0);  }  } |

2．Mapper和Reducer保持不变（详见4.6.2）

# 第5章 Yarn资源调度器

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

## 5.1 Yarn基本架构

  YARN主要由ResourceManager、NodeManager、ApplicationMaster和Container等组件构成，如图4-23所示。

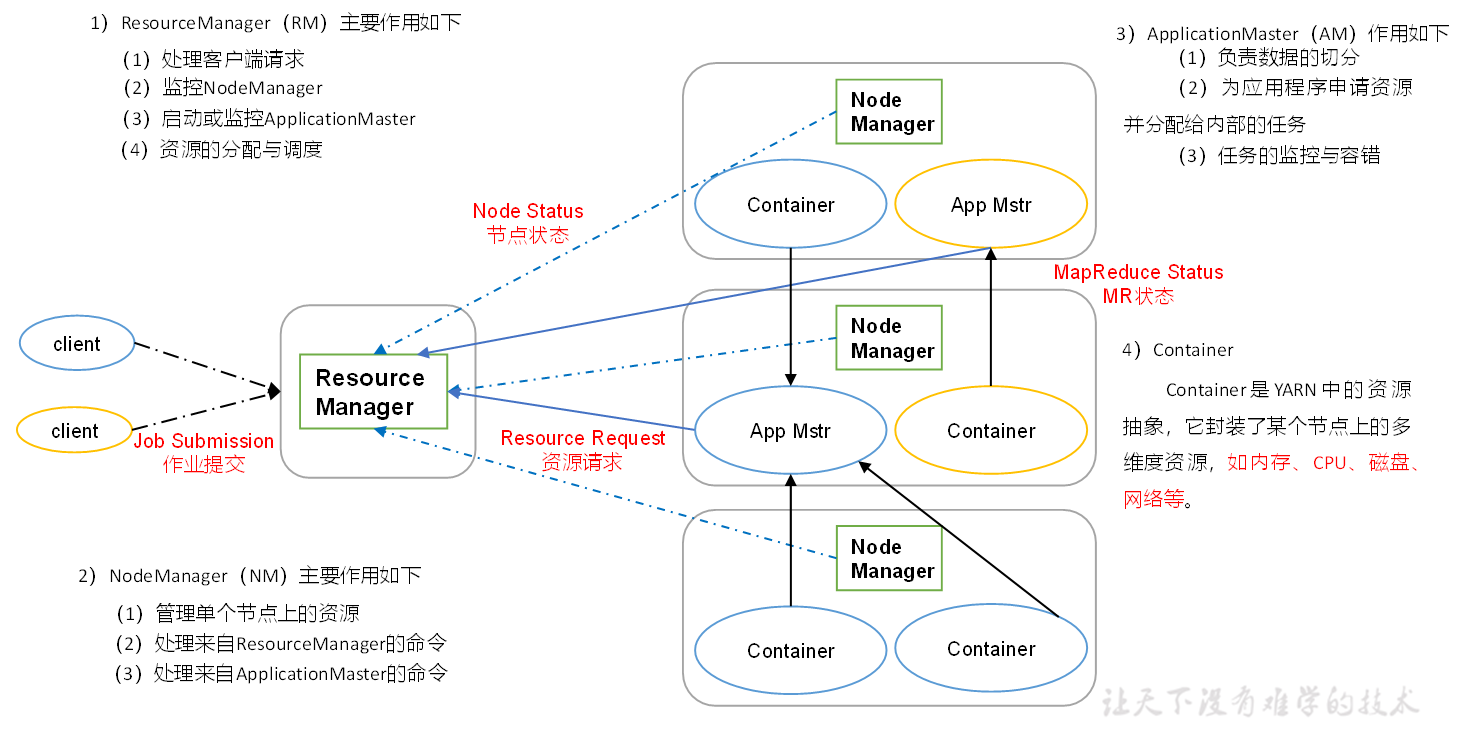


图4-23 Yarn基本架构

## 5.2 Yarn工作机制

1．Yarn运行机制，如图4-24所示。

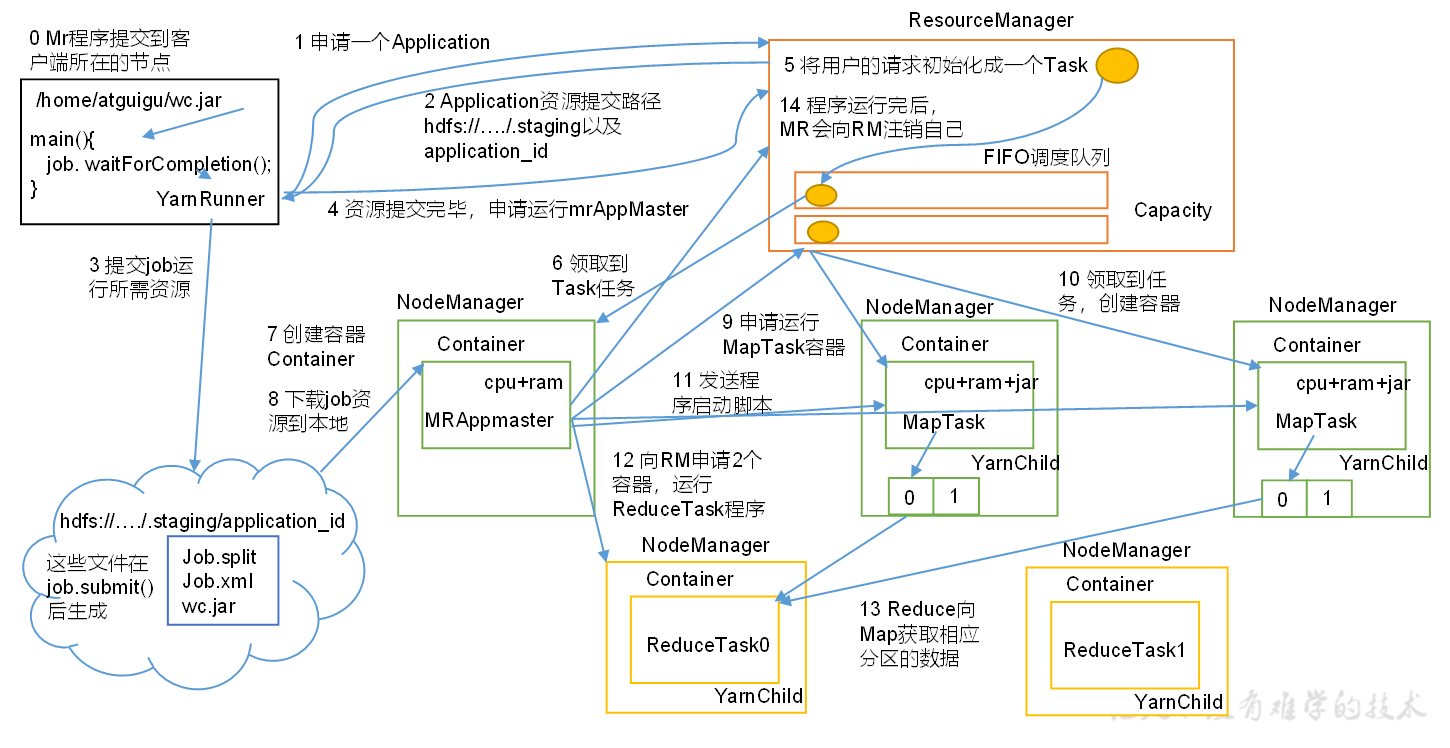


图4-24 Yarn工作机制

2．工作机制详解

（1）MR程序提交到客户端所在的节点。

（2）YarnRunner向ResourceManager申请一个Application。

（3）ResourceManager将该应用程序的资源路径返回给YarnRunner。

（4）该程序将运行所需资源提交到HDFS上。

（5）程序资源提交完毕后，申请运行mrAppMaster。

（6）ResourceManager将用户的请求初始化成一个Task。

（7）其中一个NodeManager领取到Task任务。

（8）该NodeManager创建容器Container，并产生mrAppmaster。

（9）Container从HDFS上拷贝资源到本地。

（10）mrAppmaster向ResourceManager申请运行MapTask资源。

（11）ResourceManager将MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。

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

（13）MrAppMaster等待所有MapTask运行完毕后，向ResourceManager申请容器，运行ReduceTask。

（14）ReduceTask向MapTask获取相应分区的数据。

（15）程序运行完毕后，MR会向ResourceManager申请注销自己。

## 5.3 作业提交全过程

1．作业提交过程之YARN，如图4-25所示。

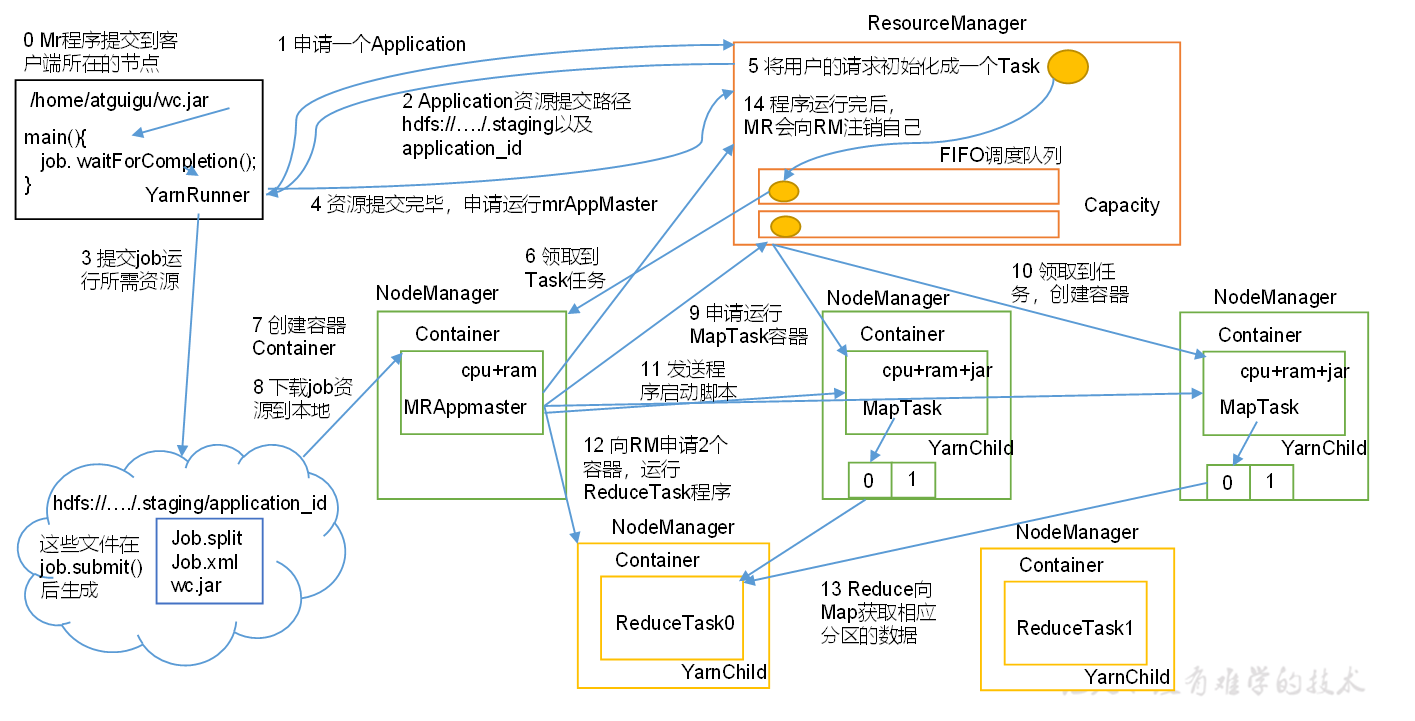


图4-25 作业提交过程之Yarn

作业提交全过程详解

（1）作业提交

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

第2步：Client向RM申请一个作业id。

第3步：RM给Client返回该job资源的提交路径和作业id。

第4步：Client提交jar包、切片信息和配置文件到指定的资源提交路径。

第5步：Client提交完资源后，向RM申请运行MrAppMaster。

（2）作业初始化

第6步：当RM收到Client的请求后，将该job添加到容量调度器中。

第7步：某一个空闲的NM领取到该Job。

第8步：该NM创建Container，并产生MRAppmaster。

第9步：下载Client提交的资源到本地。

（3）任务分配

第10步：MrAppMaster向RM申请运行多个MapTask任务资源。

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

（4）任务运行

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

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

第14步：ReduceTask向MapTask获取相应分区的数据。

第15步：程序运行完毕后，MR会向RM申请注销自己。

（5）进度和状态更新

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

（6）作业完成

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

2．作业提交过程之MapReduce，如图4-26所示

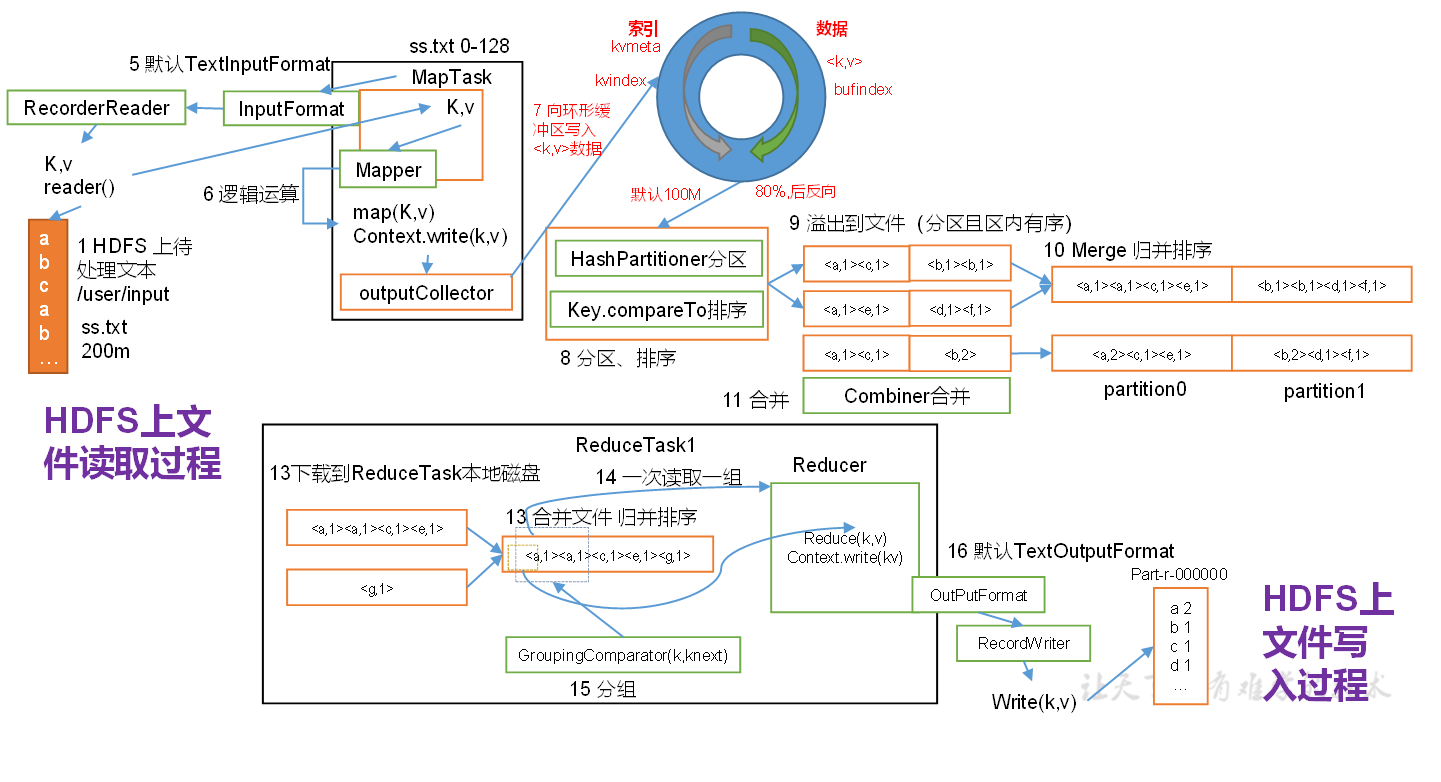


图4-26 作业提交过程之MapReduce

## 5.4 资源调度器

目前，Hadoop作业调度器主要有三种：FIFO、Capacity Scheduler和Fair Scheduler。Hadoop2.7.2默认的资源调度器是Capacity Scheduler。

具体设置详见：yarn-default.xml文件

<property>

<description>The class to use as the resource scheduler.</description>

<name>yarn.resourcemanager.scheduler.class</name>

<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>

</property>

1．先进先出调度器（FIFO），如图4-27所示

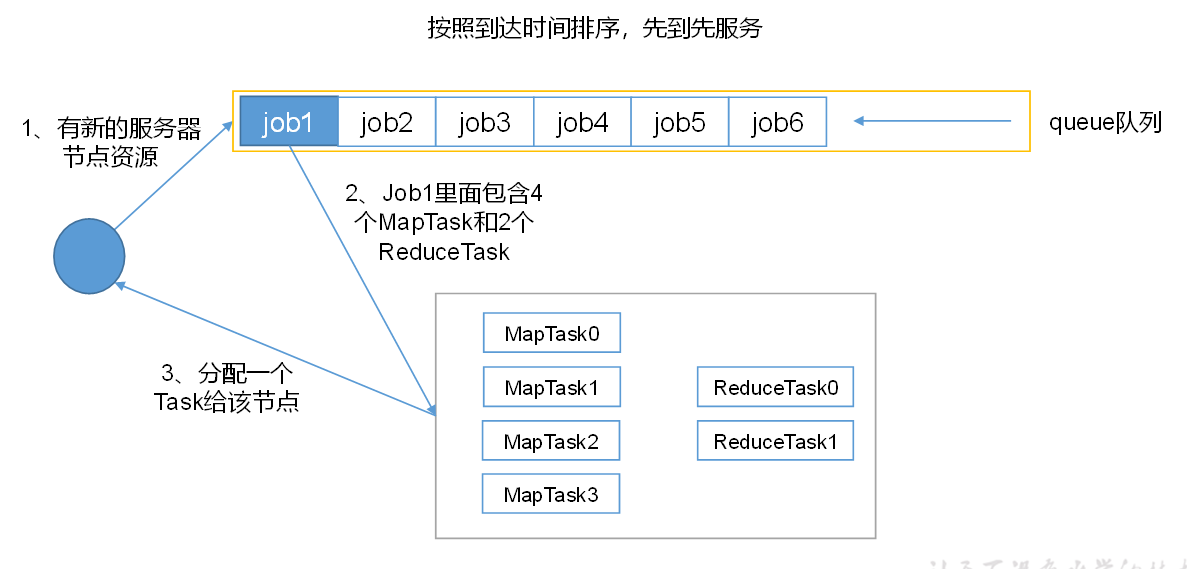
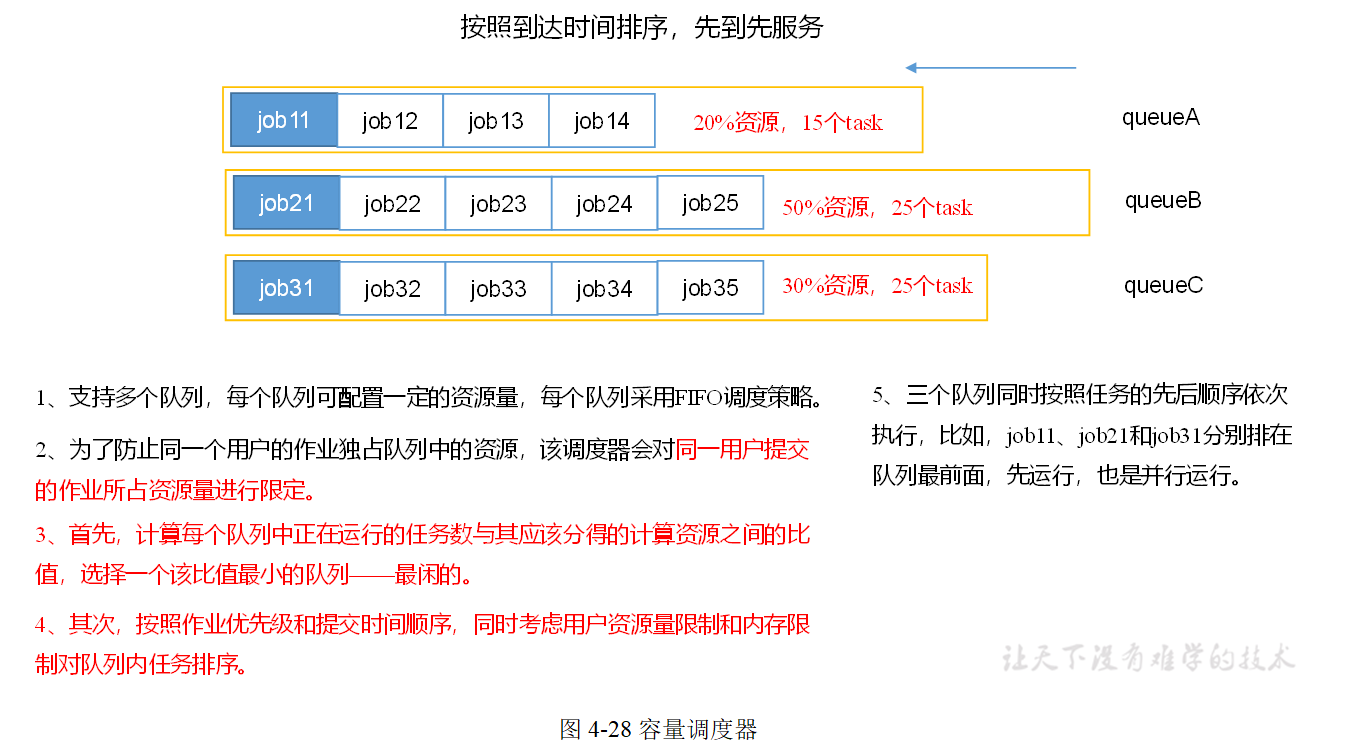


图4-27 FIFO调度器

2．容量调度器（Capacity Scheduler），如图4-28所示



3．公平调度器（Fair Scheduler），如图4-29所示

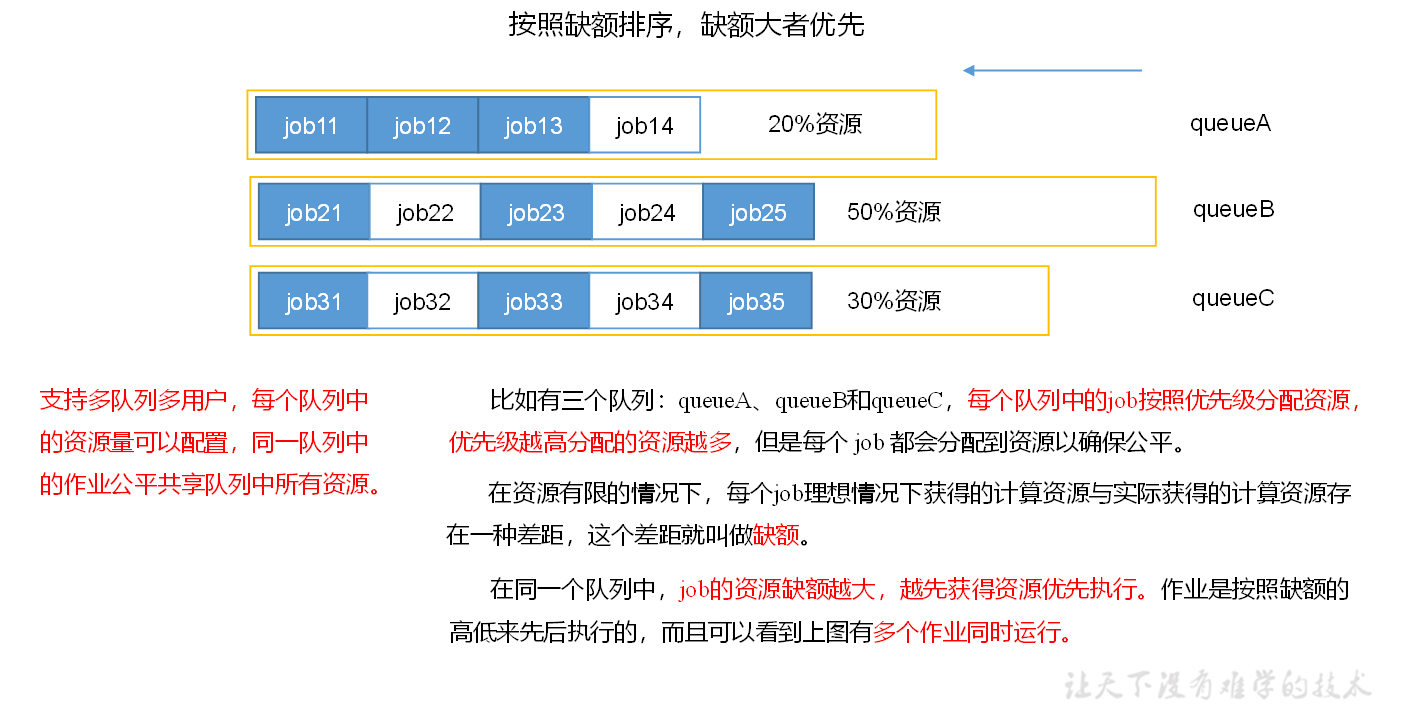


图4-29公平调度器

## 5.5 任务的推测执行

1．作业完成时间取决于最慢的任务完成时间

一个作业由若干个Map任务和Reduce任务构成。因硬件老化、软件Bug等，某些任务可能运行非常慢。

思考：系统中有99%的Map任务都完成了，只有少数几个Map老是进度很慢，完不成，怎么办？

2．推测执行机制

发现拖后腿的任务，比如某个任务运行速度远慢于任务平均速度。为拖后腿任务启动一个备份任务，同时运行。谁先运行完，则采用谁的结果。

3．执行推测任务的前提条件

（1）每个Task只能有一个备份任务

（2）当前Job已完成的Task必须不小于0.05（5%）

（3）开启推测执行参数设置。mapred-site.xml文件中默认是打开的。

<property>

<name>mapreduce.map.speculative</name>

<value>true</value>

<description>If true, then multiple instances of some map tasks may be executed in parallel.</description>

</property>

<property>

<name>mapreduce.reduce.speculative</name>

<value>true</value>

<description>If true, then multiple instances of some reduce tasks may be executed in parallel.</description>

</property>

4．不能启用推测执行机制情况

（1）任务间存在严重的负载倾斜；

（2）特殊任务，比如任务向数据库中写数据。

5．算法原理，如图4-20所示

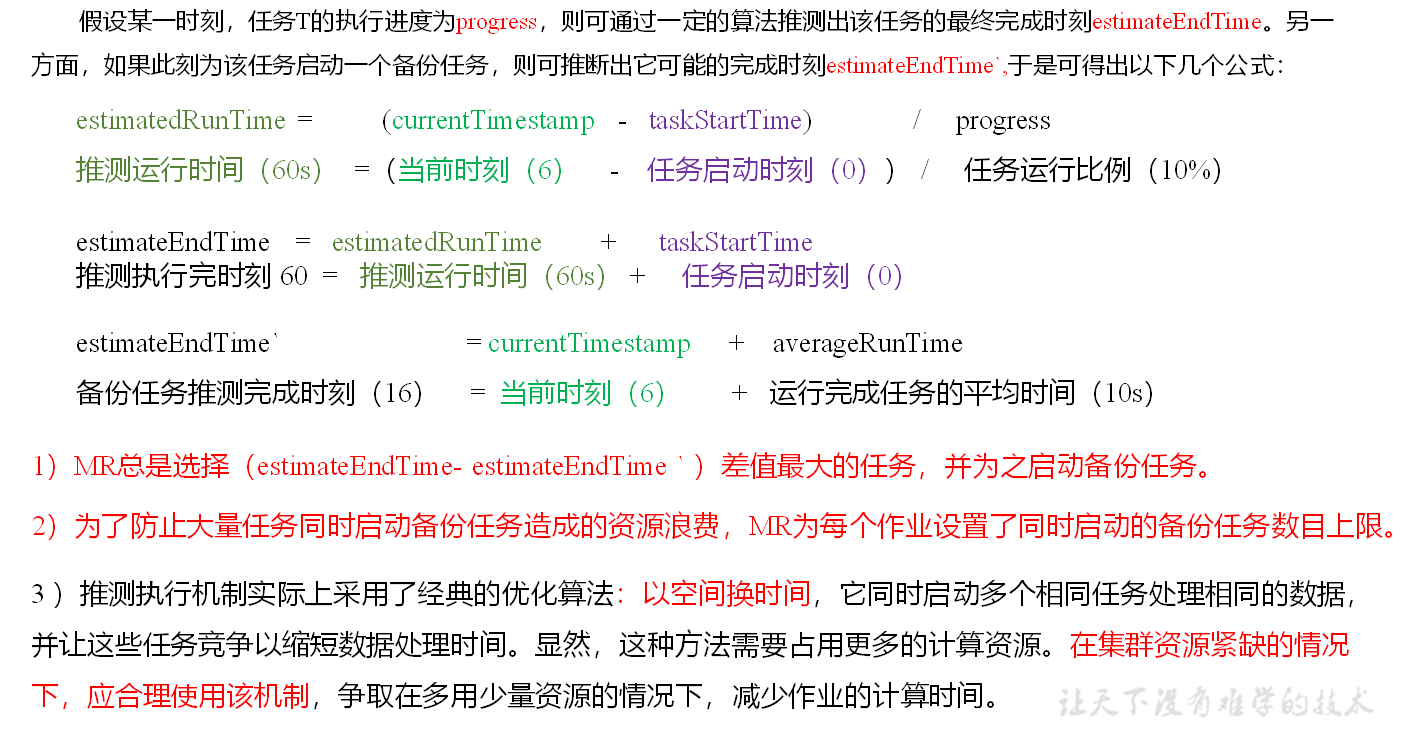


图4-30 推测执行算法原理

# 第6章 Hadoop企业优化

## 6.1 MapReduce跑的慢的原因

MapReduce程序效率的瓶颈在于两点：

1. 计算机性能

CPU、内存、磁盘、网络

1. I/O操作优化
2. 数据倾斜
3. Map和Reduce数设置不合理
4. Map运行时间太长，导致Reduce等待过久
5. 小文件过多
6. 大量的不可分块的超大文件
7. Spill次数过多
8. Merge次数过多等

## 6.2 MapReduce优化方法

MapReduce优化方法主要从六个方面考虑：数据输入、Map阶段、Reduce阶段、IO传输、数据倾斜问题和常用的调优参数。

### 6.2.1 数据输入

（1）合并小文件：在执行MR任务前将小文件进行合并，大量的小文件会产生大量的Map任务，增大Map任务装载次数，而任务的装载比较耗时，从而导致MR运行较慢。

（2）采用CombineTextInputFormat来作为输入，解决输入端大量小文件场景。

### 6.2.2 Map阶段

（1）减少溢写（Spill）次数：通过调整io.sort.mb及sort.spill。Percent参数值，增大触发spill的内存上限，减少spill次数，从而减少磁盘IO

（2）减少合并（Merge）次数：通过调整io.sort.factor参数，增大Merge的文件数目，减少Merge的次数，从而缩短MR处理时间。

（3）在Map之后，不影响业务逻辑前提下，先进行Combine处理，减少I/O。

### 6.2.3 Reduce阶段

（1）合理设置Map和Reduce数：两个都不能设置太少，也不能太多。太少会导致Task等待，延长处理时间；太多会导致Map、Reduce任务间竞争资源，造成处理超时等错误。

（2）设置Map、Reduce共存：调整slowstart.completedmaps参数，使Map运行到一定程度后，Reduce也开始运行，减少Reduce等待时间。

（3）规避使用Reduce：因为Reduce在用于连接数据集的时候将会产生大量网络消耗。

（4）合理设置Reduce端的Buffer：默认情况下，数据达到一个阈值的时候，Buffer中的数据就会写入磁盘，然后Reduce会从磁盘获得所有的数据。也就是说，Buffer和Reduce是没有直接关联的，中间多次写磁盘->读磁盘的过程，既然有这个弊端那么就可以通过参数来配置，是的Buffer中的一部分数据可以直接输送到Reduce，从而减少IO开销：mapreduce.reduce.input.buffer.percent，默认为0.0.当值大于0的时候，会保留指定比例的内存读Buffer中的数据直接拿给Reduce使用，这样一来，设置Buffer需要内存，读取数据需要内存，Reduce计算也要内存，所以要根据作业的运行情况进行调整。

### 6.2.4 I/O传输

1）采用数据压缩的方式，减少网络IO的时间，安装Snappy和Lzo压缩编码器。

2）使用SequenceFile二进制文件。

### 6.2.5 数据倾斜问题

1、数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域

数据大小倾斜——部分记录的大小远远大于平均值

2、减少数据倾斜的方法

方法1：抽样和范围分区

可以通过对原始数据进行抽样得到的结果集来预设分区边界值

方法2：自定义分区

基于输出键的背景知识进行自定义分区，例如如果Map输出键的单词来源于一本书，且其中某几个专业词汇比较多，那么就可以自定义分区将这些专业词汇发送给固定的一部分Reduce实例。而将其他的都发送给剩余的Reduce实例。

方法3：Combine

使用Combine可以大量减少数据倾斜，可能的情况下，Combine的目的就是聚合并精简数据。

方法4：采用MapJoin，避免ReduceJoin

### 6.2.6 常用的调优参数

1．资源相关参数

（1）以下参数是在用户自己的MR应用程序中配置就可以生效（mapred-default.xml）

表4-12

|  |  |
| --- | --- |
| 配置参数 | 参数说明 |
| mapreduce.map.memory.mb | 一个MapTask可使用的资源上限（单位:MB），默认为1024。如果MapTask实际使用的资源量超过该值，则会被强制杀死。 |
| mapreduce.reduce.memory.mb | 一个ReduceTask可使用的资源上限（单位:MB），默认为1024。如果ReduceTask实际使用的资源量超过该值，则会被强制杀死。 |
| mapreduce.map.cpu.vcores | 每个MapTask可使用的最多cpu core数目，默认值: 1 |
| mapreduce.reduce.cpu.vcores | 每个ReduceTask可使用的最多cpu core数目，默认值: 1 |
| mapreduce.reduce.shuffle.parallelcopies | 每个Reduce去Map中取数据的并行数。默认值是5 |
| mapreduce.reduce.shuffle.merge.percent | Buffer中的数据达到多少比例开始写入磁盘。默认值0.66 |
| mapreduce.reduce.shuffle.input.buffer.percent | Buffer大小占Reduce可用内存的比例。默认值0.7 |
| mapreduce.reduce.input.buffer.percent | 指定多少比例的内存用来存放Buffer中的数据，默认值是0.0 |

（2）应该在YARN启动之前就配置在服务器的配置文件中才能生效（yarn-default.xml）

表4-13

|  |  |
| --- | --- |
| 配置参数 | 参数说明 |
| yarn.scheduler.minimum-allocation-mb | 给应用程序Container分配的最小内存，默认值：1024 |
| yarn.scheduler.maximum-allocation-mb | 给应用程序Container分配的最大内存，默认值：8192 |
| yarn.scheduler.minimum-allocation-vcores | 每个Container申请的最小CPU核数，默认值：1 |
| yarn.scheduler.maximum-allocation-vcores | 每个Container申请的最大CPU核数，默认值：32 |
| yarn.nodemanager.resource.memory-mb | 给Containers分配的最大物理内存，默认值：8192 |

（3）Shuffle性能优化的关键参数，应在YARN启动之前就配置好（mapred-default.xml）

表4-14

|  |  |
| --- | --- |
| 配置参数 | 参数说明 |
| mapreduce.task.io.sort.mb | Shuffle的环形缓冲区大小，默认100m |
| mapreduce.map.sort.spill.percent | 环形缓冲区溢出的阈值，默认80% |

2．容错相关参数(MapReduce性能优化)

表4-15

|  |  |
| --- | --- |
| 配置参数 | 参数说明 |
| mapreduce.map.maxattempts | 每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。 |
| mapreduce.reduce.maxattempts | 每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。 |
| mapreduce.task.timeout | Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个Task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该Task处于Block状态，可能是卡住了，也许永远会卡住，为了防止因为用户程序永远Block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是600000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“AttemptID:attempt\_14267829456721\_123456\_m\_000224\_0 Timed out after 300 secsContainer killed by the ApplicationMaster.”。 |

## 6.3 HDFS小文件优化方法

### 6.3.1 HDFS小文件弊端

HDFS上每个文件都要在NameNode上建立一个索引，这个索引的大小约为150byte，这样当小文件比较多的时候，就会产生很多的索引文件，一方面会大量占用NameNode的内存空间，另一方面就是索引文件过大使得索引速度变慢。

### 6.3.2 HDFS小文件解决方案

小文件的优化无非以下几种方式：

（1）在数据采集的时候，就将小文件或小批数据合成大文件再上传HDFS。

（2）在业务处理之前，在HDFS上使用MapReduce程序对小文件进行合并。

（3）在MapReduce处理时，可采用CombineTextInputFormat提高效率。

1、HadoopArchive

是一个高效地将小文件放入HDFS块中的文件存档工具，它能将多个小文件打包成一个HAR文件，这样就减少了NameNode的内存使用。

2、SequenceFile合并小文件

SequenceFile由一系列的二进制key/value组成，如果key为文件名，value为文件内容，则可以将大批小文件合并成一个大文件。

1. CombineFileInputFormat

CombineFileInputFormat是一种新的InputFormat，用于将多个文件合并成一个单独的Split，另外它会考虑数据的存储位置。

1. 开启JVM重用

对于大量小文件Job，可以开启JVM重用会减少45%运行时间。

JVM重用原理：一个Map运行在一个JVM上，开启重用的话，该Map在JVM上运行完毕之后，JVM会继续运行其他Map。

具体设置：mapreduce.job.jvm.numtasks值在10-20之间。

# 第7章 MapReduce扩展案例

## 7.1 倒排索引案例（多job串联）

1．需求

有大量的文本（文档、网页），需要建立搜索索引，如图4-31所示。

（1）数据输入

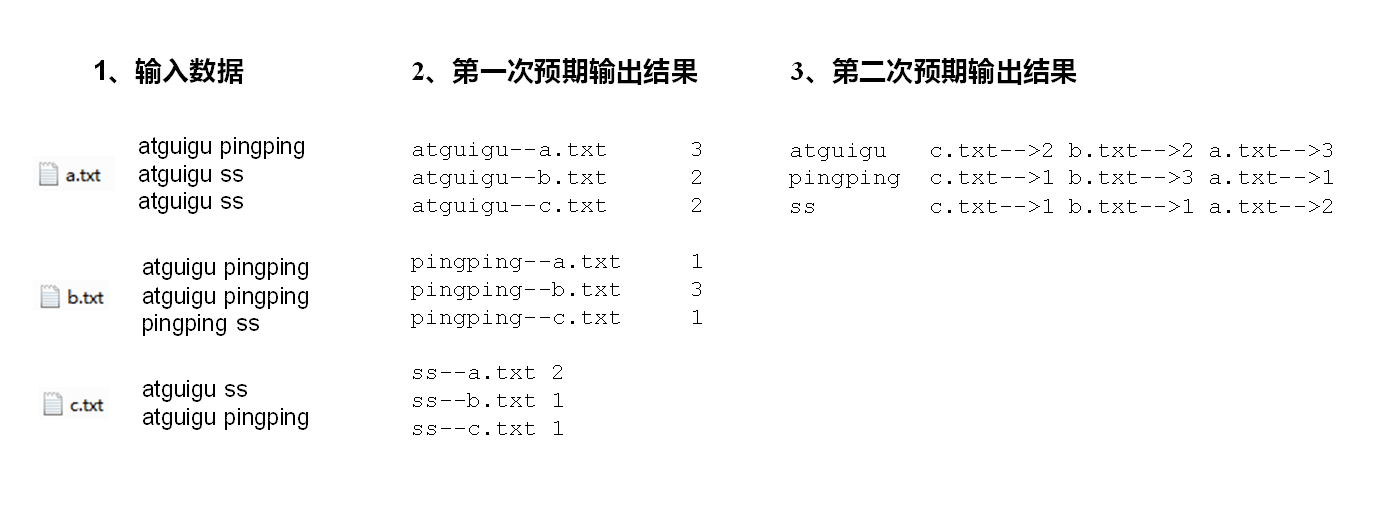
（2）期望输出数据

atguigu c.txt-->2 b.txt-->2 a.txt-->3

pingping c.txt-->1 b.txt-->3 a.txt-->1

ss c.txt-->1 b.txt-->1 a.txt-->2

2．需求分析



3．第一次处理

（1）第一次处理，编写OneIndexMapper类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.lib.input.FileSplit;  public class OneIndexMapper extends Mapper<LongWritable, Text, Text, IntWritable>{    String name;  Text k = new Text();  IntWritable v = new IntWritable();    @Override  protected void setup(Context context)throws IOException, InterruptedException {  // 获取文件名称  FileSplit split = (FileSplit) context.getInputSplit();    name = split.getPath().getName();  }    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // 1 获取1行  String line = value.toString();    // 2 切割  String[] fields = line.split(" ");  for (String word : fields) {  // 3 拼接  k.set(word+"--"+name);  v.set(1);  // 4 写出  context.write(k, v);  }  }  } |

（2）第一次处理，编写OneIndexReducer类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class OneIndexReducer 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;  // 1 累加求和  for(IntWritable value: values){  sum +=value.get();  }    v.set(sum);  // 2 写出  context.write(key, v);  }  } |

（3）第一次处理，编写OneIndexDriver类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class OneIndexDriver {  public static void main(String[] args) throws Exception {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputoneindex", "e:/output5" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(OneIndexDriver.class);  job.setMapperClass(OneIndexMapper.class);  job.setReducerClass(OneIndexReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  }  } |

（4）查看第一次输出结果

|  |
| --- |
| atguigu--a.txt 3  atguigu--b.txt 2  atguigu--c.txt 2  pingping--a.txt 1  pingping--b.txt 3  pingping--c.txt 1  ss--a.txt  2  ss--b.txt 1  ss--c.txt 1 |

4．第二次处理

（1）第二次处理，编写TwoIndexMapper类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class TwoIndexMapper extends Mapper<LongWritable, Text, Text, Text>{  Text k = new Text();  Text v = new Text();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {    // 1 获取1行数据  String line = value.toString();    // 2用“--”切割  String[] fields = line.split("--");    k.set(fields[0]);  v.set(fields[1]);    // 3 输出数据  context.write(k, v);  }  } |

（2）第二次处理，编写TwoIndexReducer类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TwoIndexReducer extends Reducer<Text, Text, Text, Text> {  Text v = new Text();  @Override  protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  // atguigu a.txt 3  // atguigu b.txt 2  // atguigu c.txt 2  // atguigu c.txt-->2 b.txt-->2 a.txt-->3  StringBuilder sb = new StringBuilder();  // 1 拼接  for (Text value : values) {  sb.append(value.toString().replace("\t", "-->") + "\t");  }  v.set(sb.toString());  // 2 写出  context.write(key, v);  }  } |

（3）第二次处理，编写TwoIndexDriver类

|  |
| --- |
| package com.atguigu.mapreduce.index;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TwoIndexDriver {  public static void main(String[] args) throws Exception {  // 输入输出路径需要根据自己电脑上实际的输入输出路径设置  args = new String[] { "e:/input/inputtwoindex", "e:/output6" };  Configuration config = new Configuration();  Job job = Job.getInstance(config);  job.setJarByClass(TwoIndexDriver.class);  job.setMapperClass(TwoIndexMapper.class);  job.setReducerClass(TwoIndexReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result?0:1);  }  } |

（4）第二次查看最终结果

atguigu c.txt-->2 b.txt-->2 a.txt-->3

pingping c.txt-->1 b.txt-->3 a.txt-->1

ss c.txt-->1 b.txt-->1 a.txt-->2

## 7.2 TopN案例

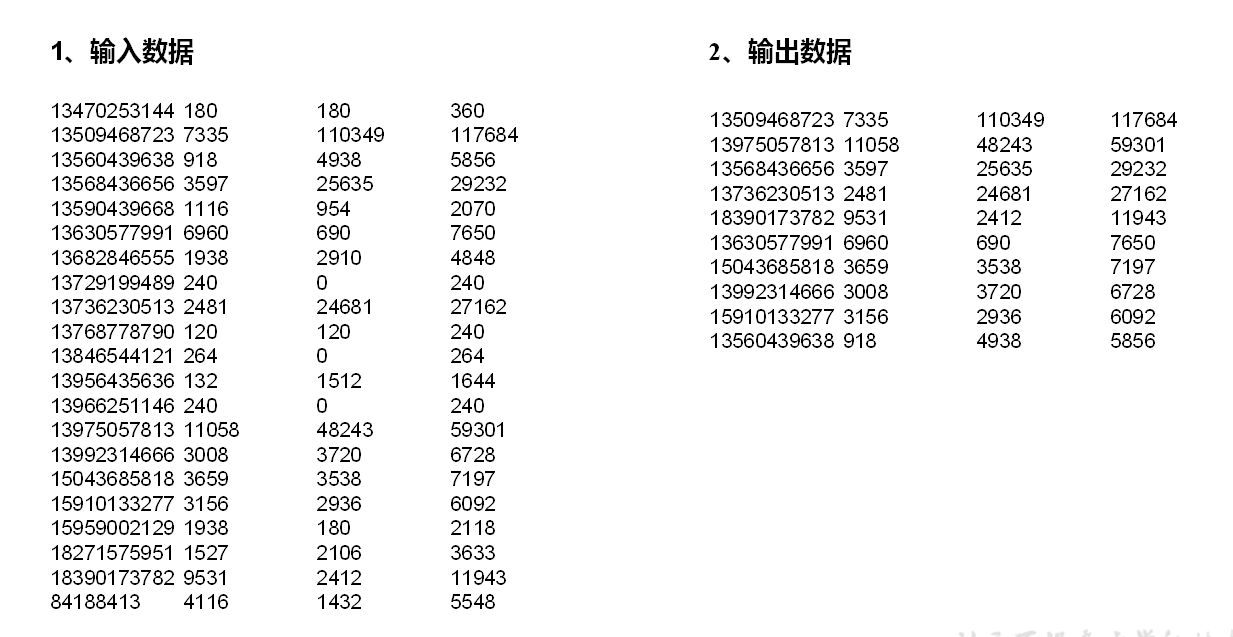
1．需求

对需求2.3输出结果进行加工，输出流量使用量在前10的用户信息

（1）输入数据 （2）输出数据

2．需求分析



3．实现代码

（1）编写FlowBean类

|  |
| --- |
| package com.atguigu.mr.top;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.WritableComparable;  public class FlowBean implements WritableComparable<FlowBean>{  private long upFlow;  private long downFlow;  private long sumFlow;      public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }  @Override  public void readFields(DataInput in) throws IOException {  upFlow = in.readLong();  downFlow = in.readLong();  sumFlow = in.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;  }  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  public void set(long downFlow2, long upFlow2) {  downFlow = downFlow2;  upFlow = upFlow2;  sumFlow = downFlow2 + upFlow2;  }  @Override  public int compareTo(FlowBean bean) {    int result;    if (this.sumFlow > bean.getSumFlow()) {  result = -1;  }else if (this.sumFlow < bean.getSumFlow()) {  result = 1;  }else {  result = 0;  }    return result;  }  } |

（2）编写TopNMapper类

|  |
| --- |
| package com.atguigu.mr.top;  import java.io.IOException;  import java.util.Iterator;  import java.util.TreeMap;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class TopNMapper extends Mapper<LongWritable, Text, FlowBean, Text>{    // 定义一个TreeMap作为存储数据的容器（天然按key排序）  private TreeMap<FlowBean, Text> flowMap = new TreeMap<FlowBean, Text>();  private FlowBean kBean;    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  kBean = new FlowBean();  Text v = new Text();    // 1 获取一行  String line = value.toString();    // 2 切割  String[] fields = line.split("\t");    // 3 封装数据  String phoneNum = fields[0];  long upFlow = Long.parseLong(fields[1]);  long downFlow = Long.parseLong(fields[2]);  long sumFlow = Long.parseLong(fields[3]);    kBean.setDownFlow(downFlow);  kBean.setUpFlow(upFlow);  kBean.setSumFlow(sumFlow);    v.set(phoneNum);    // 4 向TreeMap中添加数据  flowMap.put(kBean, v);    // 5 限制TreeMap的数据量，超过10条就删除掉流量最小的一条数据  if (flowMap.size() > 10) {  //flowMap.remove(flowMap.firstKey());  flowMap.remove(flowMap.lastKey());  }  }    @Override  protected void cleanup(Context context) throws IOException, InterruptedException {  // 6 遍历treeMap集合，输出数据  Iterator<FlowBean> bean = flowMap.keySet().iterator();  while (bean.hasNext()) {  FlowBean k = bean.next();  context.write(k, flowMap.get(k));  }  }  } |

（3）编写TopNReducer类

|  |
| --- |
| package com.atguigu.mr.top;  import java.io.IOException;  import java.util.Iterator;  import java.util.TreeMap;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TopNReducer extends Reducer<FlowBean, Text, Text, FlowBean> {  // 定义一个TreeMap作为存储数据的容器（天然按key排序）  TreeMap<FlowBean, Text> flowMap = new TreeMap<FlowBean, Text>();  @Override  protected void reduce(FlowBean key, Iterable<Text> values, Context context)throws IOException, InterruptedException {  for (Text value : values) {  FlowBean bean = new FlowBean();  bean.set(key.getDownFlow(), key.getUpFlow());  // 1 向treeMap集合中添加数据  flowMap.put(bean, new Text(value));  // 2 限制TreeMap数据量，超过10条就删除掉流量最小的一条数据  if (flowMap.size() > 10) {  // flowMap.remove(flowMap.firstKey());  flowMap.remove(flowMap.lastKey());  }  }  }  @Override  protected void cleanup(Reducer<FlowBean, Text, Text, FlowBean>.Context context) throws IOException, InterruptedException {  // 3 遍历集合，输出数据  Iterator<FlowBean> it = flowMap.keySet().iterator();  while (it.hasNext()) {  FlowBean v = it.next();  context.write(new Text(flowMap.get(v)), v);  }  }  }  （4）编写TopNDriver类  package com.atguigu.mr.top;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TopNDriver {  public static void main(String[] args) throws Exception {    args = new String[]{"e:/output1","e:/output3"};    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(TopNDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(TopNMapper.class);  job.setReducerClass(TopNReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(FlowBean.class);  job.setMapOutputValueClass(Text.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

## 7.3 找博客共同好友案例

1．需求

以下是博客的好友列表数据，冒号前是一个用户，冒号后是该用户的所有好友（数据中的好友关系是单向的）

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

（1）数据输入



2．需求分析

先求出A、B、C、….等是谁的好友

第一次输出结果

|  |
| --- |
| A I,K,C,B,G,F,H,O,D,  B A,F,J,E,  C A,E,B,H,F,G,K,  D G,C,K,A,L,F,E,H,  E G,M,L,H,A,F,B,D,  F L,M,D,C,G,A,  G M,  H O,  I O,C,  J O,  K B,  L D,E,  M E,F,  O A,H,I,J,F, |

第二次输出结果

|  |
| --- |
| A-B E C  A-C D F  A-D E F  A-E D B C  A-F O B C D E  A-G F E C D  A-H E C D O  A-I O  A-J O B  A-K D C  A-L F E D  A-M E F  B-C A  B-D A E  B-E C  B-F E A C  B-G C E A  B-H A E C  B-I A  B-K C A  B-L E  B-M E  B-O A  C-D A F  C-E D  C-F D A  C-G D F A  C-H D A  C-I A  C-K A D  C-L D F  C-M F  C-O I A  D-E L  D-F A E  D-G E A F  D-H A E  D-I A  D-K A  D-L E F  D-M F E  D-O A  E-F D M C B  E-G C D  E-H C D  E-J B  E-K C D  E-L D  F-G D C A E  F-H A D O E C  F-I O A  F-J B O  F-K D C A  F-L E D  F-M E  F-O A  G-H D C E A  G-I A  G-K D A C  G-L D F E  G-M E F  G-O A  H-I O A  H-J O  H-K A C D  H-L D E  H-M E  H-O A  I-J O  I-K A  I-O A  K-L D  K-O A  L-M E F |

3．代码实现

（1）第一次Mapper类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class OneShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{    @Override  protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)  throws IOException, InterruptedException {  // 1 获取一行 A:B,C,D,F,E,O  String line = value.toString();    // 2 切割  String[] fields = line.split(":");    // 3 获取person和好友  String person = fields[0];  String[] friends = fields[1].split(",");    // 4写出去  for(String friend: friends){  // 输出 <好友，人>  context.write(new Text(friend), new Text(person));  }  }  } |

（2）第一次Reducer类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class OneShareFriendsReducer extends Reducer<Text, Text, Text, Text>{    @Override  protected void reduce(Text key, Iterable<Text> values, Context context)throws IOException, InterruptedException {    StringBuffer sb = new StringBuffer();  //1 拼接  for(Text person: values){  sb.append(person).append(",");  }    //2 写出  context.write(key, new Text(sb.toString()));  }  } |

（3）第一次Driver类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class OneShareFriendsDriver {  public static void main(String[] args) throws Exception {    // 1 获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 2 指定jar包运行的路径  job.setJarByClass(OneShareFriendsDriver.class);  // 3 指定map/reduce使用的类  job.setMapperClass(OneShareFriendsMapper.class);  job.setReducerClass(OneShareFriendsReducer.class);    // 4 指定map输出的数据类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    // 5 指定最终输出的数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    // 6 指定job的输入原始所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 提交  boolean result = job.waitForCompletion(true);    System.exit(result?0:1);  }  } |

（4）第二次Mapper类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import java.io.IOException;  import java.util.Arrays;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class TwoShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // A I,K,C,B,G,F,H,O,D,  // 友 人，人，人  String line = value.toString();  String[] friend\_persons = line.split("\t");  String friend = friend\_persons[0];  String[] persons = friend\_persons[1].split(",");  Arrays.sort(persons);  for (int i = 0; i < persons.length - 1; i++) {    for (int j = i + 1; j < persons.length; j++) {  // 发出 <人-人，好友> ，这样，相同的“人-人”对的所有好友就会到同1个reduce中去  context.write(new Text(persons[i] + "-" + persons[j]), new Text(friend));  }  }  }  } |

（5）第二次Reducer类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TwoShareFriendsReducer extends Reducer<Text, Text, Text, Text>{    @Override  protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {    StringBuffer sb = new StringBuffer();  for (Text friend : values) {  sb.append(friend).append(" ");  }    context.write(key, new Text(sb.toString()));  }  } |

（6）第二次Driver类

|  |
| --- |
| package com.atguigu.mapreduce.friends;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TwoShareFriendsDriver {  public static void main(String[] args) throws Exception {    // 1 获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 2 指定jar包运行的路径  job.setJarByClass(TwoShareFriendsDriver.class);  // 3 指定map/reduce使用的类  job.setMapperClass(TwoShareFriendsMapper.class);  job.setReducerClass(TwoShareFriendsReducer.class);    // 4 指定map输出的数据类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    // 5 指定最终输出的数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    // 6 指定job的输入原始所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result?0:1);  }  } |

# 第8章 常见错误及解决方案

1）导包容易出错。尤其Text和CombineTextInputFormat。

2）Mapper中第一个输入的参数必须是LongWritable或者NullWritable，不可以是IntWritable. 报的错误是类型转换异常。

3）java.lang.Exception: java.io.IOException: Illegal partition for 13926435656 (4)，说明Partition和ReduceTask个数没对上，调整ReduceTask个数。

4）如果分区数不是1，但是reducetask为1，是否执行分区过程。答案是：不执行分区过程。因为在MapTask的源码中，执行分区的前提是先判断ReduceNum个数是否大于1。不大于1肯定不执行。

5）在Windows环境编译的jar包导入到Linux环境中运行，

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

报如下错误：

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

原因是Windows环境用的jdk1.7，Linux环境用的jdk1.8。

解决方案：统一jdk版本。

6）缓存pd.txt小文件案例中，报找不到pd.txt文件

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

7）报类型转换异常。

通常都是在驱动函数中设置Map输出和最终输出时编写错误。

Map输出的key如果没有排序，也会报类型转换异常。

8）集群中运行wc.jar时出现了无法获得输入文件。

原因：WordCount案例的输入文件不能放用HDFS集群的根目录。

9）出现了如下相关异常

Exception in thread "main" java.lang.UnsatisfiedLinkError: org.apache.hadoop.io.nativeio.NativeIO$Windows.access0(Ljava/lang/String;I)Z

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

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

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

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

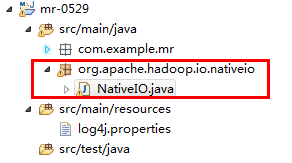
at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:356)

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

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

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

方案二：创建如下包名，并将NativeIO.java拷贝到该包名下

10）自定义Outputformat时，注意在RecordWirter中的close方法必须关闭流资源。否则输出的文件内容中数据为空。

@Override

public void close(TaskAttemptContext context) throws IOException, InterruptedException {

if (atguigufos != null) {

atguigufos.close();

}

if (otherfos != null) {

otherfos.close();

}

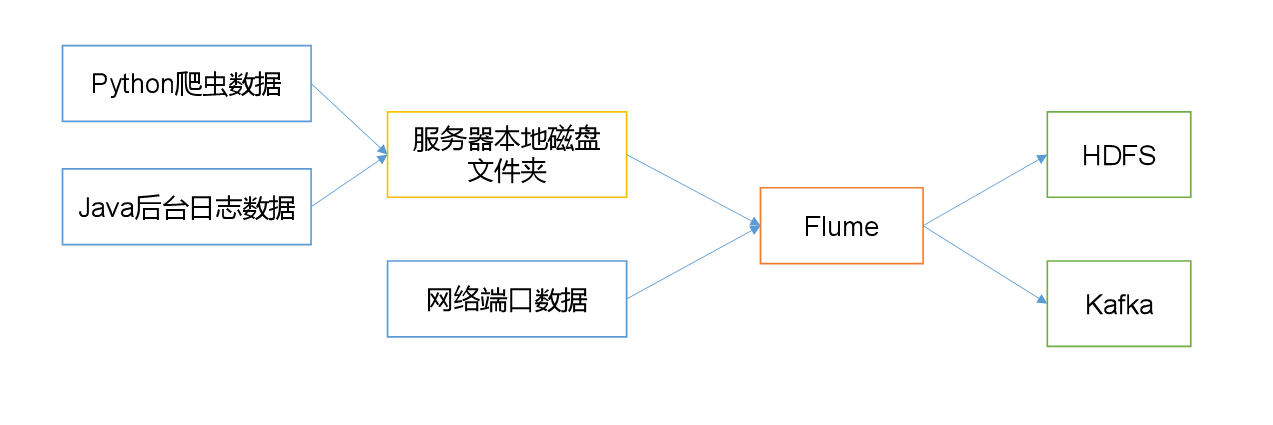
}

# Flume

第1章 概述

## 1.1 Flume定义

Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。Flume基于流式架构，灵活简单。最主要的作用就是实时读取服务器本地磁盘的数据，将数据写入HDFS。



## 1.2 Flume的优点

1. 可以和任意存储进程集成。
2. 输入的的数据速率大于写入目的存储的速率，flume会进行缓冲，减小hdfs的压力。
3. flume中的事务基于channel，使用了两个事务模型（sender + receiver），确保消息被可靠发送。

Flume使用两个独立的事务分别负责从soucrce到channel，以及从channel到sink的事件传递。一旦事务中所有的数据全部成功提交到channel，那么source才认为该数据读取完成。同理，只有成功被sink写出去的数据，才会从channel中移除。

## 1.3 Flume组成架构

Flume组成架构如图1-1，图1-2所示：

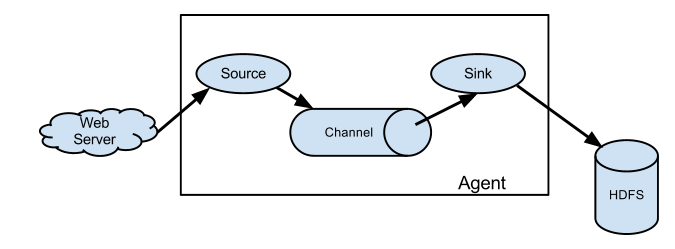


图1-1 Flume组成架构

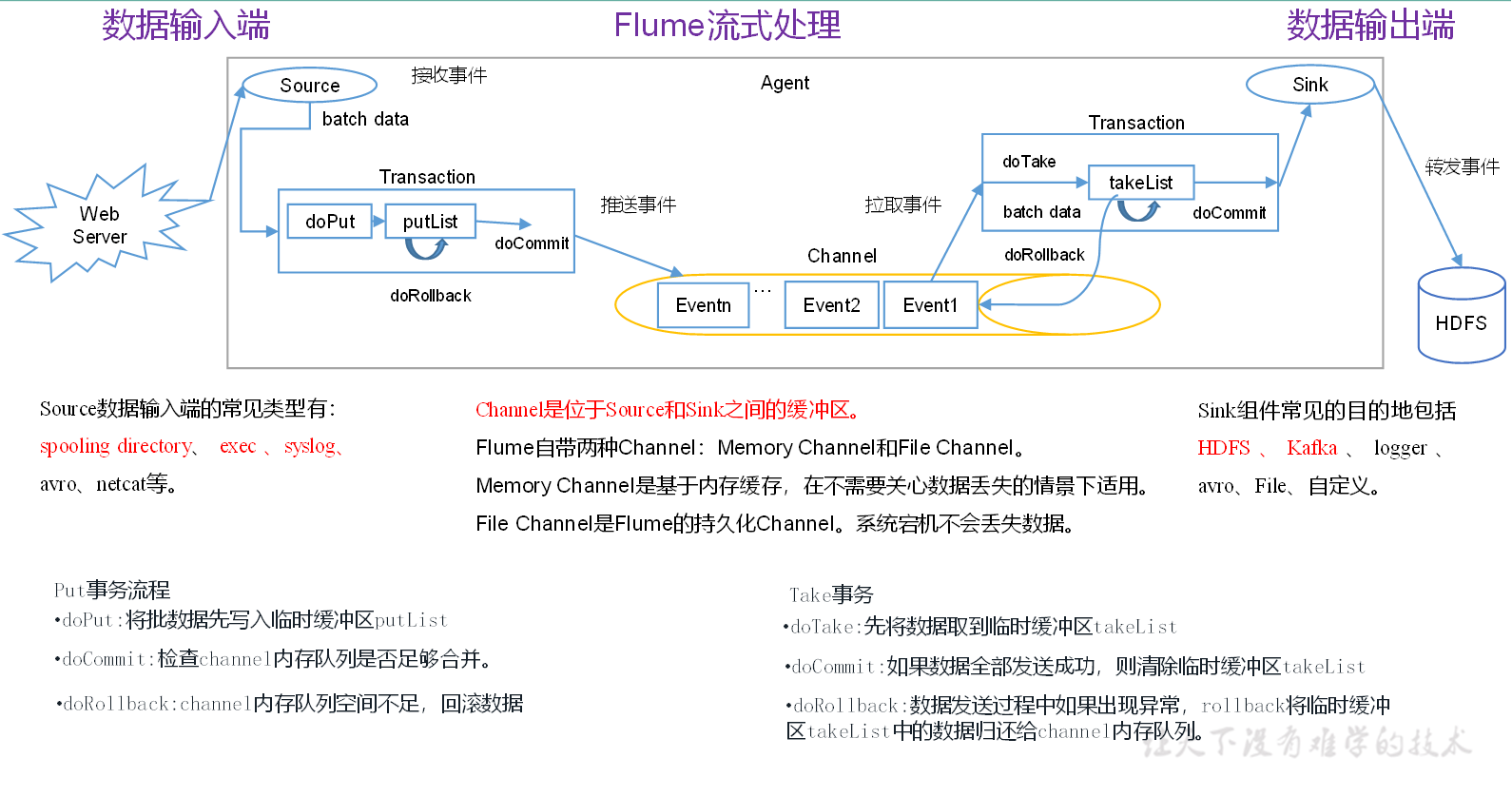


图1-2 Flume组成架构详解

下面我们来详细介绍一下Flume架构中的组件。

### 1.3.1 Agent

Agent是一个JVM进程，它以事件的形式将数据从源头送至目的。

Agent主要有3个部分组成，Source、Channel、Sink。

### 1.3.2 Source

Source是负责接收数据到Flume Agent的组件。Source组件可以处理各种类型、各种格式的日志数据，包括avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy。

### 1.3.3 Channel

Channel是位于Source和Sink之间的缓冲区。因此，Channel允许Source和Sink运作在不同的速率上。Channel是线程安全的，可以同时处理几个Source的写入操作和几个Sink的读取操作。

Flume自带两种Channel：Memory Channel和File Channel。

Memory Channel是内存中的队列。Memory Channel在不需要关心数据丢失的情景下适用。如果需要关心数据丢失，那么Memory Channel就不应该使用，因为程序死亡、机器宕机或者重启都会导致数据丢失。

File Channel将所有事件写到磁盘。因此在程序关闭或机器宕机的情况下不会丢失数据。

### 1.3.4 Sink

Sink不断地轮询Channel中的事件且批量地移除它们，并将这些事件批量写入到存储或索引系统、或者被发送到另一个Flume Agent。

Sink是完全事务性的。在从Channel批量删除数据之前，每个Sink用Channel启动一个事务。批量事件一旦成功写出到存储系统或下一个Flume Agent，Sink就利用Channel提交事务。事务一旦被提交，该Channel从自己的内部缓冲区删除事件。

Sink组件目的地包括hdfs、logger、avro、thrift、ipc、file、null、HBase、solr、自定义。

### 1.3.5 Event

传输单元，Flume数据传输的基本单元，以事件的形式将数据从源头送至目的地。  Event由可选的header和载有数据的一个byte array 构成。Header是容纳了key-value字符串对的HashMap。



## 1.4 Flume拓扑结构

Flume的拓扑结构如图1-3、1-4、1-5和1-6所示：

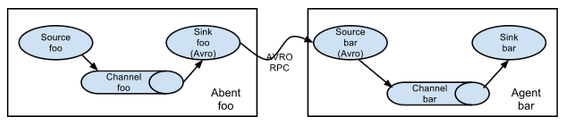


图1-3 Flume Agent连接

这种模式是将多个flume给顺序连接起来了，从最初的source开始到最终sink传送的目的存储系统。此模式不建议桥接过多的flume数量， flume数量过多不仅会影响传输速率，而且一旦传输过程中某个节点flume宕机，会影响整个传输系统。

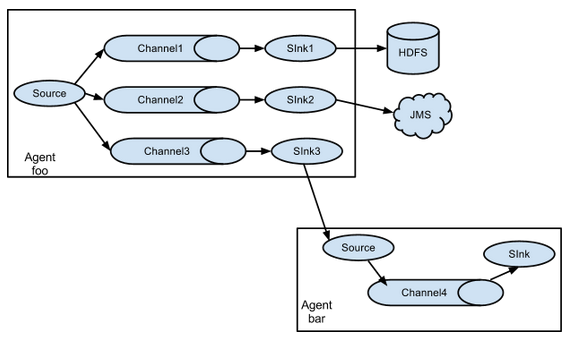


图1-4 单source，多channel、sink

Flume支持将事件流向一个或者多个目的地。这种模式将数据源复制到多个channel中，每个channel都有相同的数据，sink可以选择传送的不同的目的地。

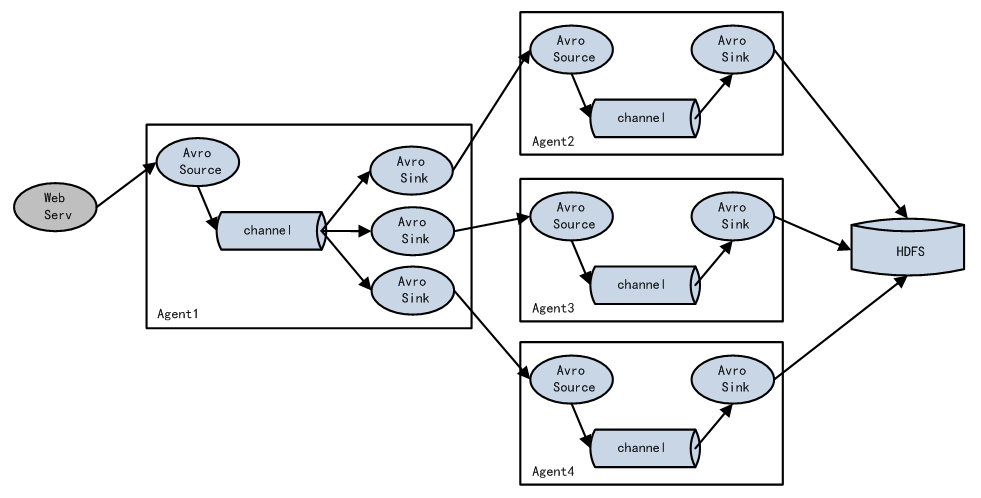


图1-5 Flume负载均衡

Flume支持使用将多个sink逻辑上分到一个sink组，flume将数据发送到不同的sink，主要解决负载均衡和故障转移问题。

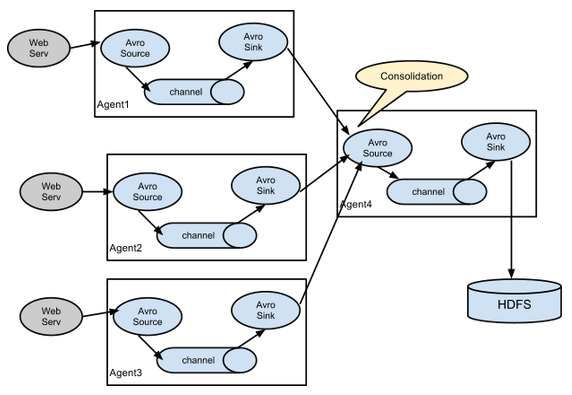
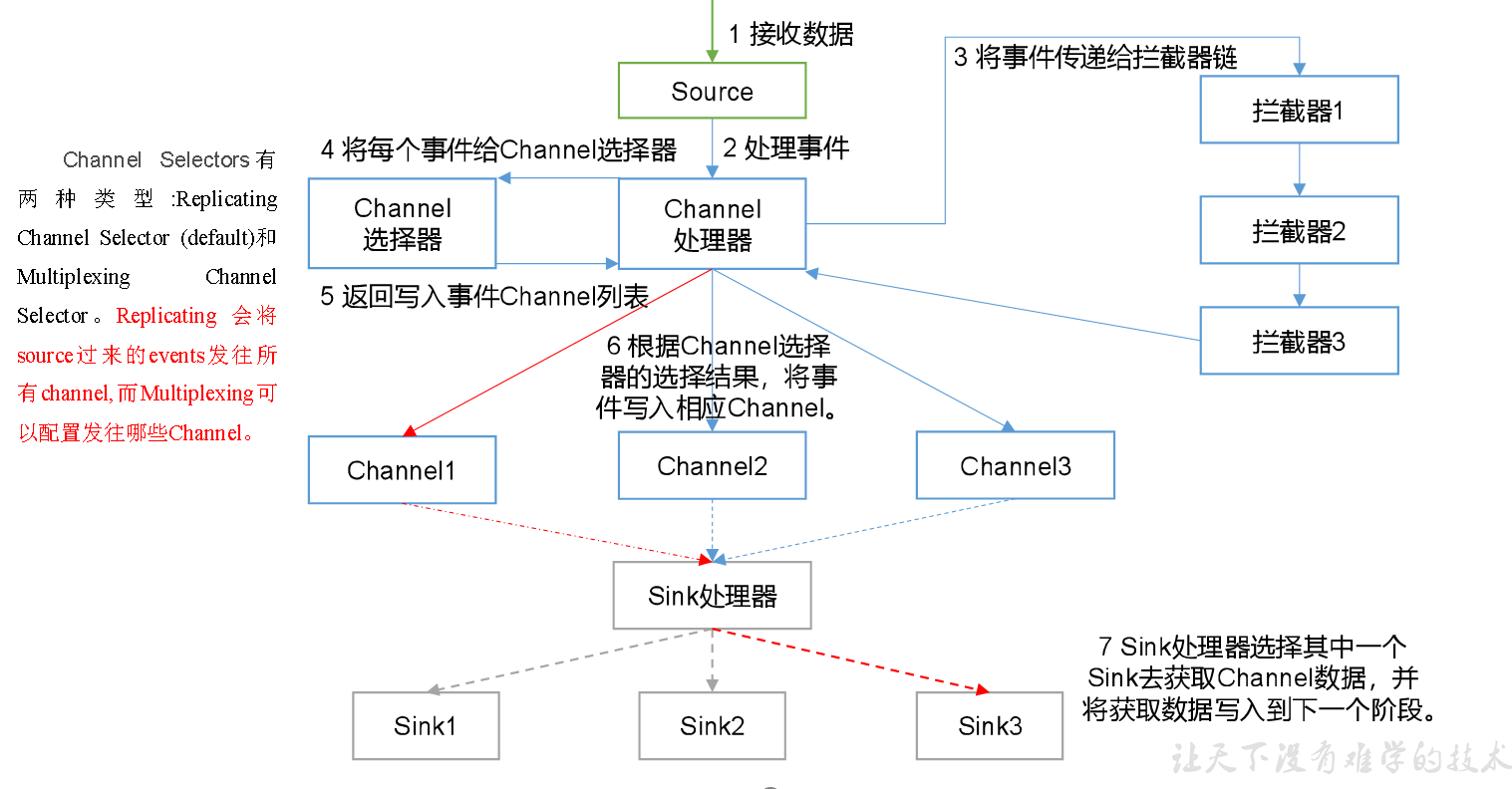


图1-6 Flume Agent聚合

这种模式是我们最常见的，也非常实用，日常web应用通常分布在上百个服务器，大者甚至上千个、上万个服务器。产生的日志，处理起来也非常麻烦。用flume的这种组合方式能很好的解决这一问题，每台服务器部署一个flume采集日志，传送到一个集中收集日志的flume，再由此flume上传到hdfs、hive、hbase、jms等，进行日志分析。

## 1.5 Flume Agent内部原理



# 第2章 快速入门

## 2.1 Flume安装地址

1） Flume官网地址

<http://flume.apache.org/>

2）文档查看地址

<http://flume.apache.org/FlumeUserGuide.html>

3）下载地址

http://archive.apache.org/dist/flume/

## 2.2 安装部署

1）将apache-flume-1.7.0-bin.tar.gz上传到linux的/opt/software目录下

2）解压apache-flume-1.7.0-bin.tar.gz到/opt/module/目录下

[atguigu@hadoop102 software]$ tar -zxf apache-flume-1.7.0-bin.tar.gz -C /opt/module/

3）修改apache-flume-1.7.0-bin的名称为flume

[atguigu@hadoop102 module]$ mv apache-flume-1.7.0-bin flume

1. 将flume/conf下的flume-env.sh.template文件修改为flume-env.sh，并配置flume-env.sh文件

[atguigu@hadoop102 conf]$ mv flume-env.sh.template flume-env.sh

[atguigu@hadoop102 conf]$ vi flume-env.sh

export JAVA\_HOME=/opt/module/jdk1.8.0\_144

# 第3章 企业开发案例

## 3.1 监控端口数据官方案例

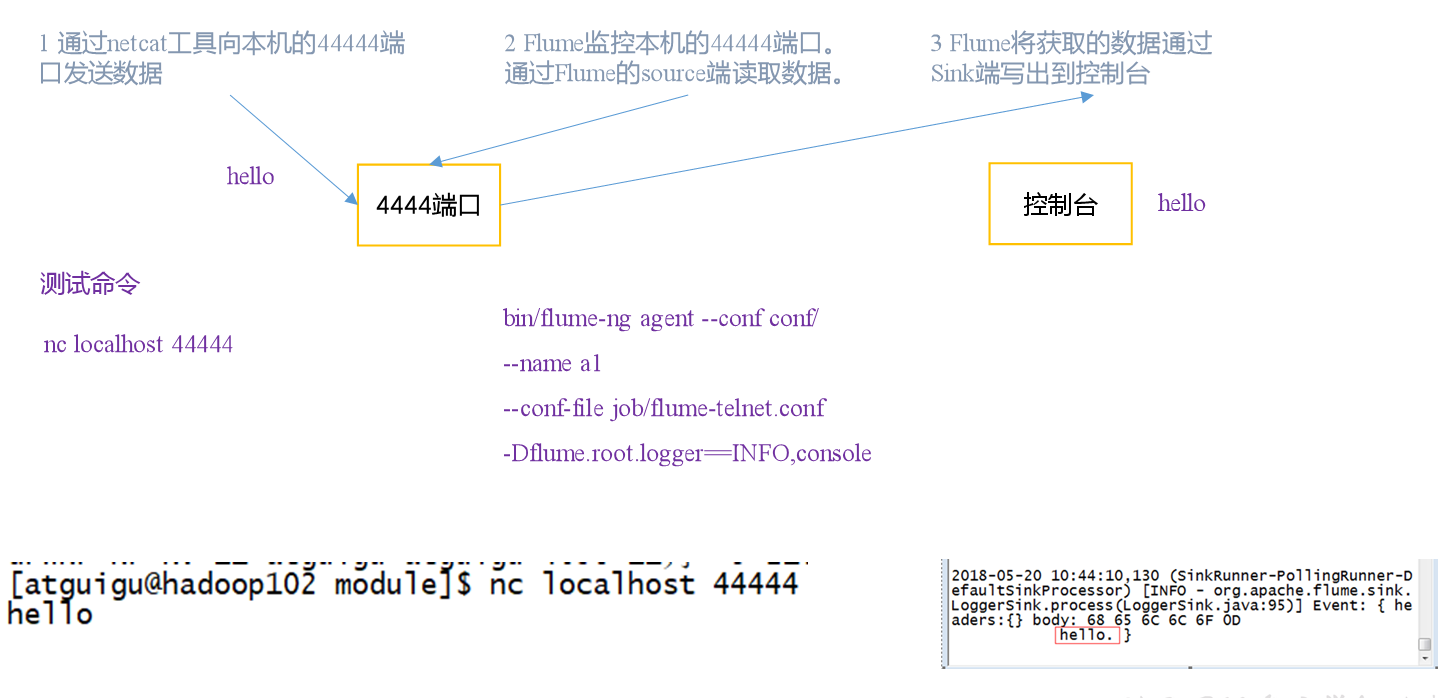
1）案例需求：

首先启动Flume任务，监控本机44444端口，服务端；

然后通过netcat工具向本机44444端口发送消息，客户端；

最后Flume将监听的数据实时显示在控制台。

2）需求分析：



3）实现步骤：

1．安装netcat工具

[atguigu@hadoop102 software]$ sudo yum install -y nc

2．判断44444端口是否被占用

[atguigu@hadoop102 flume-telnet]$ sudo netstat -tunlp | grep 44444

功能描述：netstat命令是一个监控TCP/IP网络的非常有用的工具，它可以显示路由表、实际的网络连接以及每一个网络接口设备的状态信息。

基本语法：netstat [选项]

选项参数：

-t或--tcp：显示TCP传输协议的连线状况；

-u或--udp：显示UDP传输协议的连线状况；

-n或--numeric：直接使用ip地址，而不通过域名服务器；

-l或--listening：显示监控中的服务器的Socket；

-p或--programs：显示正在使用Socket的程序识别码（PID）和程序名称；

3．创建Flume Agent配置文件flume-netcat-logger.conf

在flume目录下创建job文件夹并进入job文件夹。

[atguigu@hadoop102 flume]$ mkdir job

[atguigu@hadoop102 flume]$ cd job/

在job文件夹下创建Flume Agent配置文件flume-netcat-logger.conf。

[atguigu@hadoop102 job]$ touch flume-netcat-logger.conf

在flume-netcat-logger.conf文件中添加如下内容。

[atguigu@hadoop102 job]$ vim flume-netcat-logger.conf

添加内容如下：

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = netcat

a1.sources.r1.bind = localhost

a1.sources.r1.port = 44444

# Describe the sink

a1.sinks.k1.type = logger

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

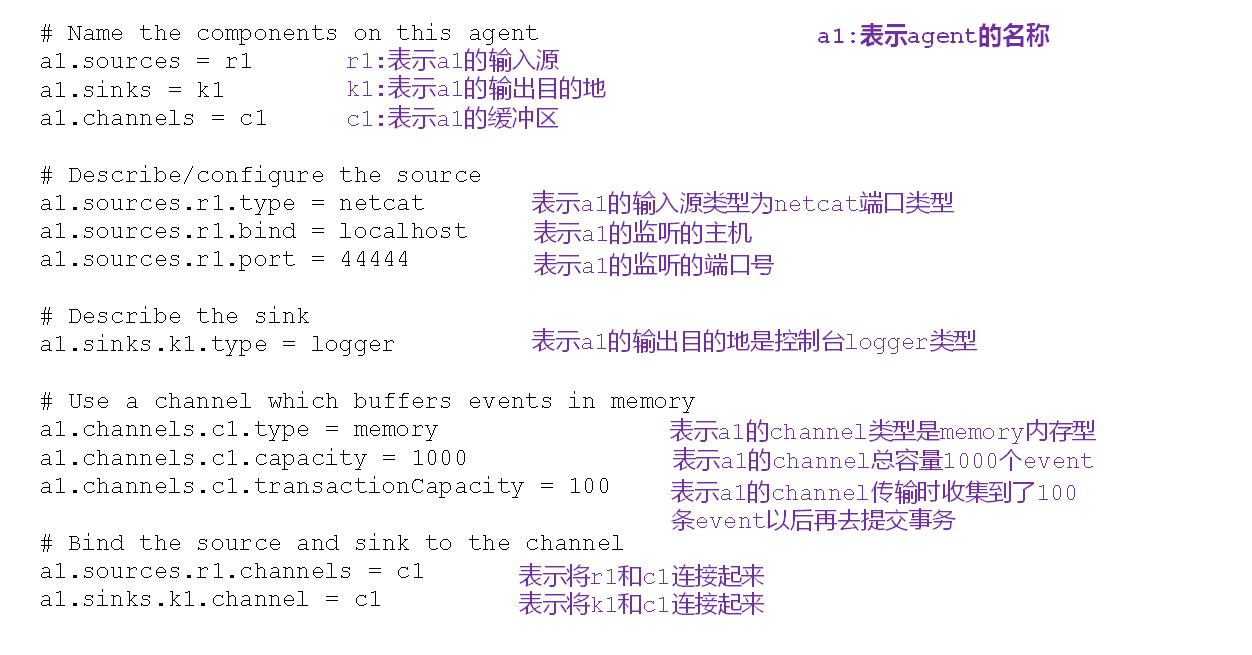
# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

注：配置文件来源于官方手册<http://flume.apache.org/FlumeUserGuide.html>

**配置文件解释：**



4. 先开启flume监听端口

第一种写法：

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/flume-netcat-logger.conf -Dflume.root.logger=INFO,console

第二种写法：

[atguigu@hadoop102 flume]$ bin/flume-ng agent -c conf/ -n a1 –f job/flume-netcat-logger.conf -Dflume.root.logger=INFO,console

参数说明：

--conf conf/ ：表示配置文件存储在conf/目录

--name a1 ：表示给agent起名为a1

--conf-file job/flume-netcat.conf ：flume本次启动读取的配置文件是job文件夹下的flume-telnet.conf

-Dflume.root.logger==INFO,console ：-D表示flume运行时动态修改flume.root.logger参数属性值，并将控制台日志打印级别设置为INFO级别。日志级别包括:log、info、warn、error。

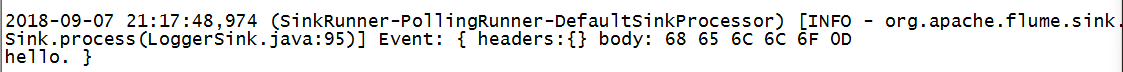
5．使用netcat工具向本机的44444端口发送内容

[atguigu@hadoop102 ~]$ nc localhost 44444

hello

atguigu

6．在Flume监听页面观察接收数据情况

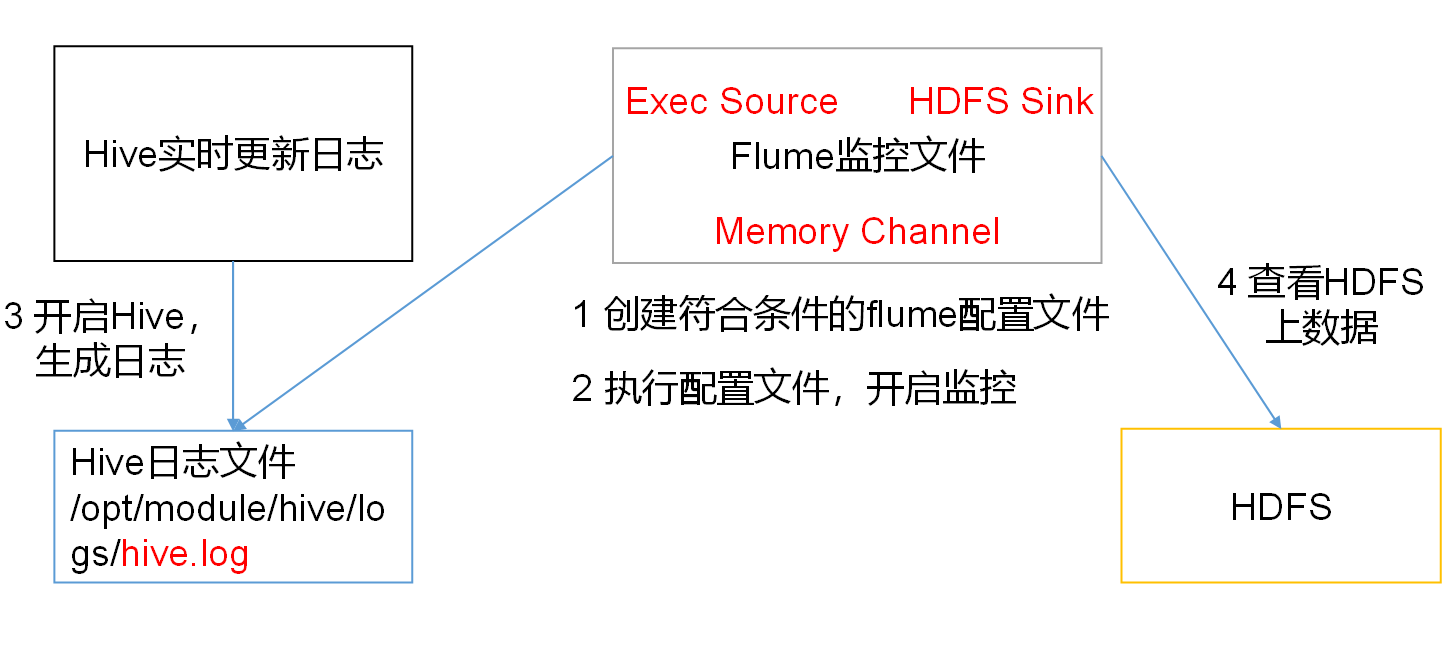


思考：nc hadoop102 44444， flume能否接收到？

## 3.2 实时读取本地文件到HDFS案例

1）案例需求：实时监控Hive日志，并上传到HDFS中

2）需求分析：



3）实现步骤：

1. Flume要想将数据输出到HDFS，必须持有Hadoop相关jar包

将commons-configuration-1.6.jar、

hadoop-auth-2.7.2.jar、

hadoop-common-2.7.2.jar、

hadoop-hdfs-2.7.2.jar、

commons-io-2.4.jar、

htrace-core-3.1.0-incubating.jar

拷贝到/opt/module/flume/lib文件夹下。

1. 创建flume-file-hdfs.conf文件

创建文件

[atguigu@hadoop102 job]$ touch flume-file-hdfs.conf

注：要想读取Linux系统中的文件，就得按照Linux命令的规则执行命令。由于Hive日志在Linux系统中所以读取文件的类型选择：exec即execute执行的意思。表示执行Linux命令来读取文件。

[atguigu@hadoop102 job]$ vim flume-file-hdfs.conf

添加如下内容

# Name the components on this agent

a2.sources = r2

a2.sinks = k2

a2.channels = c2

# Describe/configure the source

a2.sources.r2.type = exec

a2.sources.r2.command = tail -F /opt/module/hive/logs/hive.log

a2.sources.r2.shell = /bin/bash -c

# Describe the sink

a2.sinks.k2.type = hdfs

a2.sinks.k2.hdfs.path = hdfs://hadoop102:9000/flume/%Y%m%d/%H

#上传文件的前缀

a2.sinks.k2.hdfs.filePrefix = logs-

#是否按照时间滚动文件夹

a2.sinks.k2.hdfs.round = true

#多少时间单位创建一个新的文件夹

a2.sinks.k2.hdfs.roundValue = 1

#重新定义时间单位

a2.sinks.k2.hdfs.roundUnit = hour

#是否使用本地时间戳

a2.sinks.k2.hdfs.useLocalTimeStamp = true

#积攒多少个Event才flush到HDFS一次

a2.sinks.k2.hdfs.batchSize = 1000

#设置文件类型，可支持压缩

a2.sinks.k2.hdfs.fileType = DataStream

#多久生成一个新的文件

a2.sinks.k2.hdfs.rollInterval = 60

#设置每个文件的滚动大小

a2.sinks.k2.hdfs.rollSize = 134217700

#文件的滚动与Event数量无关

a2.sinks.k2.hdfs.rollCount = 0

# Use a channel which buffers events in memory

a2.channels.c2.type = memory

a2.channels.c2.capacity = 1000

a2.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel

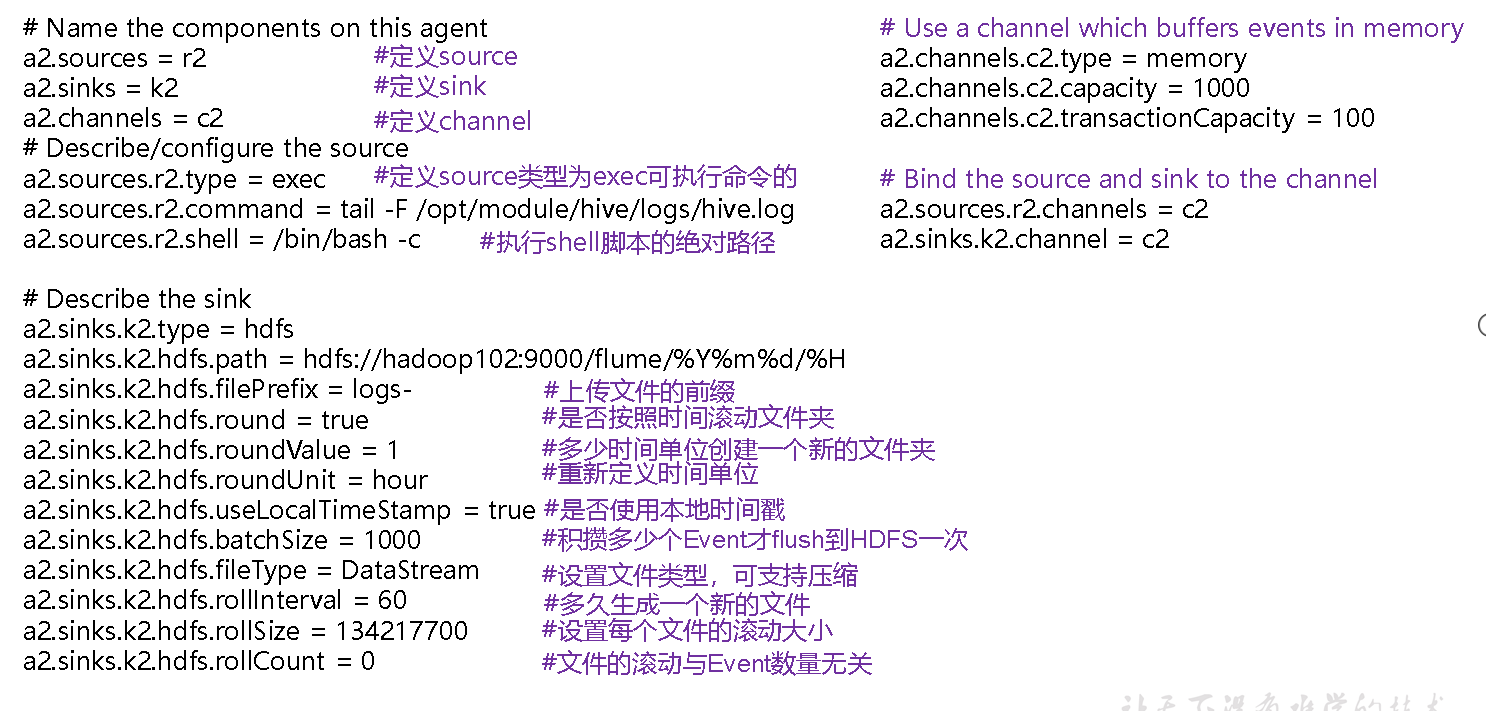
a2.sources.r2.channels = c2

a2.sinks.k2.channel = c2

**注意**：

1. 对于所有与时间相关的转义序列，Event Header中必须存在以 “timestamp”的key（除非hdfs.useLocalTimeStamp设置为true，此方法会使用TimestampInterceptor自动添加timestamp）。

a3.sinks.k3.hdfs.useLocalTimeStamp = true



1. 执行监控配置

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/flume-file-hdfs.conf

1. 开启Hadoop和Hive并操作Hive产生日志

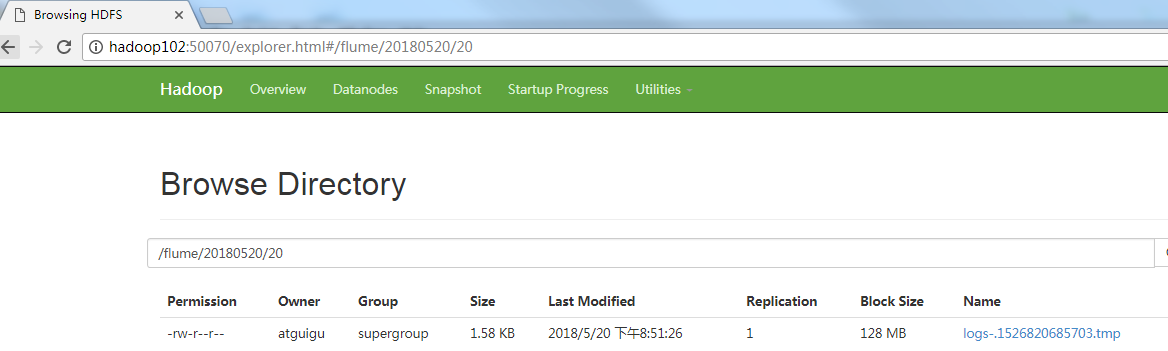
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh

[atguigu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh

[atguigu@hadoop102 hive]$ bin/hive

hive (default)>

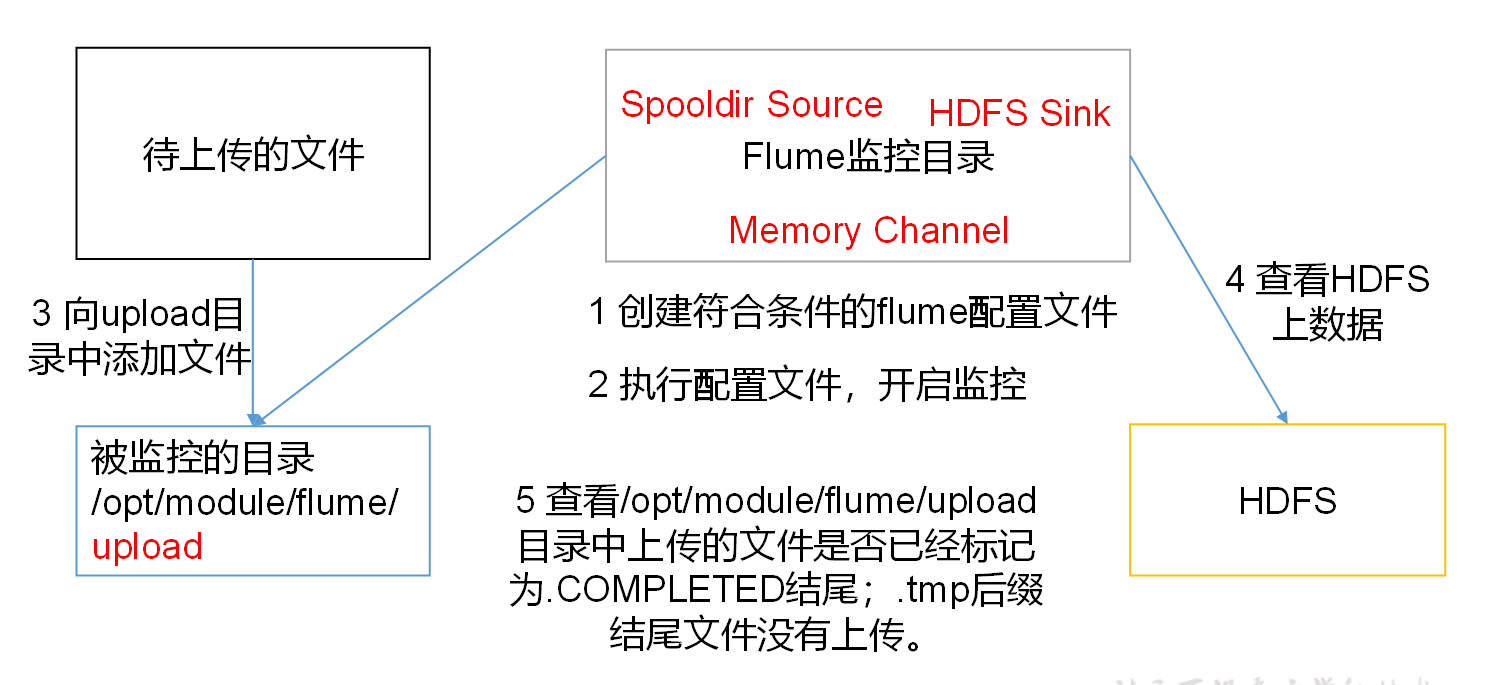
1. 在HDFS上查看文件。



## 3.3 实时读取目录文件到HDFS案例

1）案例需求：使用Flume监听整个目录的文件

2）需求分析：



3）实现步骤：

1．创建配置文件flume-dir-hdfs.conf

创建一个文件

[atguigu@hadoop102 job]$ touch flume-dir-hdfs.conf

打开文件

[atguigu@hadoop102 job]$ vim flume-dir-hdfs.conf

添加如下内容

a3.sources = r3

a3.sinks = k3

a3.channels = c3

# Describe/configure the source

a3.sources.r3.type = spooldir

a3.sources.r3.spoolDir = /opt/module/flume/upload

a3.sources.r3.fileSuffix = .COMPLETED

a3.sources.r3.fileHeader = true

#忽略所有以.tmp结尾的文件，不上传

a3.sources.r3.ignorePattern = ([^ ]\*\.tmp)

# Describe the sink

a3.sinks.k3.type = hdfs

a3.sinks.k3.hdfs.path = hdfs://hadoop102:9000/flume/upload/%Y%m%d/%H

#上传文件的前缀

a3.sinks.k3.hdfs.filePrefix = upload-

#是否按照时间滚动文件夹

a3.sinks.k3.hdfs.round = true

#多少时间单位创建一个新的文件夹

a3.sinks.k3.hdfs.roundValue = 1

#重新定义时间单位

a3.sinks.k3.hdfs.roundUnit = hour

#是否使用本地时间戳

a3.sinks.k3.hdfs.useLocalTimeStamp = true

#积攒多少个Event才flush到HDFS一次

a3.sinks.k3.hdfs.batchSize = 100

#设置文件类型，可支持压缩

a3.sinks.k3.hdfs.fileType = DataStream

#多久生成一个新的文件

a3.sinks.k3.hdfs.rollInterval = 60

#设置每个文件的滚动大小大概是128M

a3.sinks.k3.hdfs.rollSize = 134217700

#文件的滚动与Event数量无关

a3.sinks.k3.hdfs.rollCount = 0

# Use a channel which buffers events in memory

a3.channels.c3.type = memory

a3.channels.c3.capacity = 1000

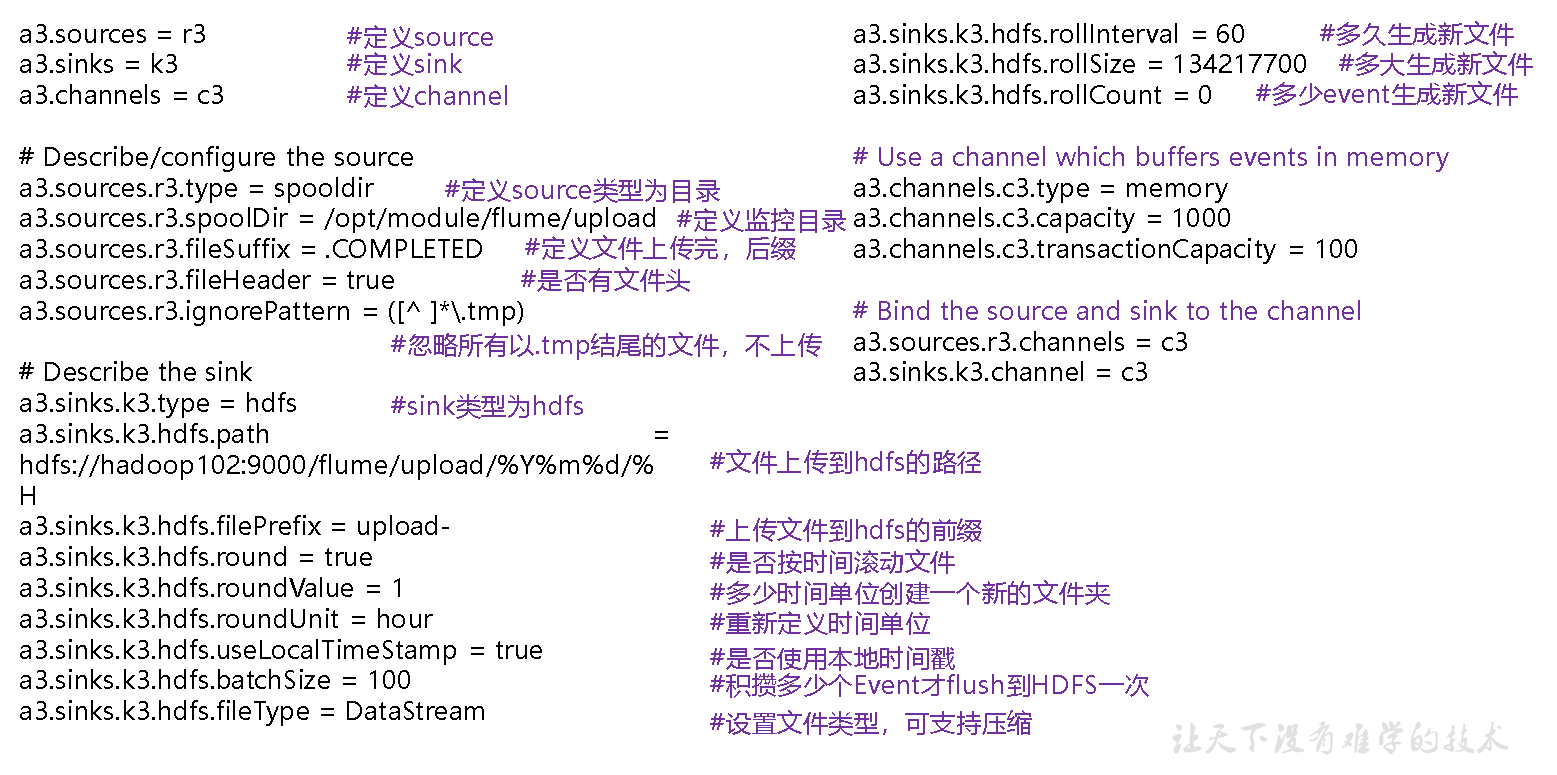
a3.channels.c3.transactionCapacity = 100

# Bind the source and sink to the channel

a3.sources.r3.channels = c3

a3.sinks.k3.channel = c3

**实时读取目录文件到HDFS：**



2. 启动监控文件夹命令

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/flume-dir-hdfs.conf

说明： 在使用Spooling Directory Source时

1. 不要在监控目录中创建并持续修改文件
2. 上传完成的文件会以.COMPLETED结尾
3. 被监控文件夹每500毫秒扫描一次文件变动

3. 向upload文件夹中添加文件

在/opt/module/flume目录下创建upload文件夹

[atguigu@hadoop102 flume]$ mkdir upload

向upload文件夹中添加文件

[atguigu@hadoop102 upload]$ touch atguigu.txt

[atguigu@hadoop102 upload]$ touch atguigu.tmp

[atguigu@hadoop102 upload]$ touch atguigu.log

4. 查看HDFS上的数据



5. 等待1s，再次查询upload文件夹

[atguigu@hadoop102 upload]$ ll

总用量 0

-rw-rw-r--. 1 atguigu atguigu 0 5月 20 22:31 atguigu.log.COMPLETED

-rw-rw-r--. 1 atguigu atguigu 0 5月 20 22:31 atguigu.tmp

-rw-rw-r--. 1 atguigu atguigu 0 5月 20 22:31 atguigu.txt.COMPLETED

## 3.4 单数据源多出口案例(选择器)

单Source多Channel、Sink如图3-1所示。

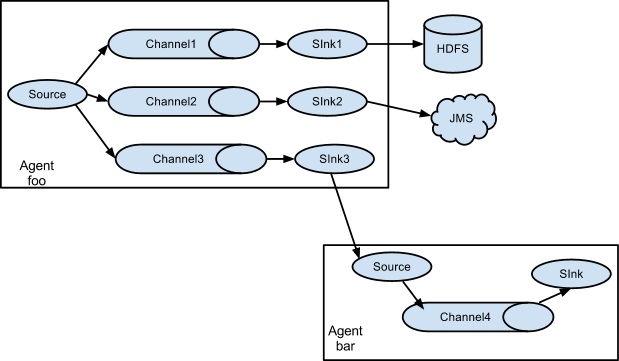
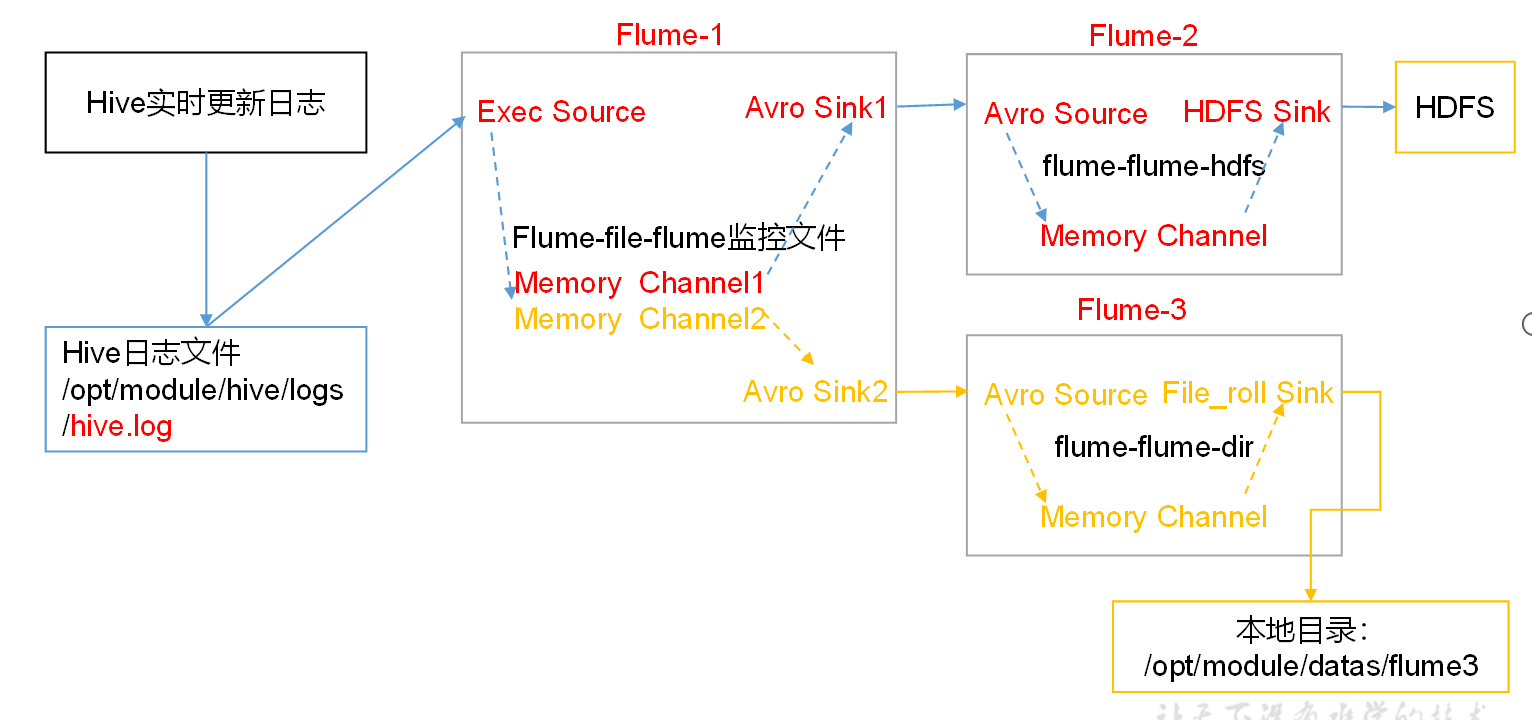


图3-1 单Source多Channel、Sink

1）案例需求：使用Flume-1监控文件变动，Flume-1将变动内容传递给Flume-2，Flume-2负责存储到HDFS。同时Flume-1将变动内容传递给Flume-3，Flume-3负责输出到Local FileSystem。

2）需求分析：

单数据源多出口案例



3）实现步骤：

0．准备工作

在/opt/module/flume/job目录下创建group1文件夹

[atguigu@hadoop102 job]$ cd group1/

在/opt/module/datas/目录下创建flume3文件夹

[atguigu@hadoop102 datas]$ mkdir flume3

1．创建flume-file-flume.conf

配置1个接收日志文件的source和两个channel、两个sink，分别输送给flume-flume-hdfs和flume-flume-dir。

创建配置文件并打开

[atguigu@hadoop102 group1]$ touch flume-file-flume.conf

[atguigu@hadoop102 group1]$ vim flume-file-flume.conf

添加如下内容

# Name the components on this agent

a1.sources = r1

a1.sinks = k1 k2

a1.channels = c1 c2

# 将数据流复制给所有channel

a1.sources.r1.selector.type = replicating

# Describe/configure the source

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F /opt/module/hive/logs/hive.log

a1.sources.r1.shell = /bin/bash -c

# Describe the sink

# sink端的avro是一个数据发送者

a1.sinks.k1.type = avro

a1.sinks.k1.hostname = hadoop102

a1.sinks.k1.port = 4141

a1.sinks.k2.type = avro

a1.sinks.k2.hostname = hadoop102

a1.sinks.k2.port = 4142

# Describe the channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

a1.channels.c2.type = memory

a1.channels.c2.capacity = 1000

a1.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1 c2

a1.sinks.k1.channel = c1

a1.sinks.k2.channel = c2

注：Avro是由Hadoop创始人Doug Cutting创建的一种语言无关的数据序列化和RPC框架。

注：RPC（Remote Procedure Call）—远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

2．创建flume-flume-hdfs.conf

配置上级Flume输出的Source，输出是到HDFS的Sink。

创建配置文件并打开

[atguigu@hadoop102 group1]$ touch flume-flume-hdfs.conf

[atguigu@hadoop102 group1]$ vim flume-flume-hdfs.conf

添加如下内容

# Name the components on this agent

a2.sources = r1

a2.sinks = k1

a2.channels = c1

# Describe/configure the source

# source端的avro是一个数据接收服务

a2.sources.r1.type = avro

a2.sources.r1.bind = hadoop102

a2.sources.r1.port = 4141

# Describe the sink

a2.sinks.k1.type = hdfs

a2.sinks.k1.hdfs.path = hdfs://hadoop102:9000/flume2/%Y%m%d/%H

#上传文件的前缀

a2.sinks.k1.hdfs.filePrefix = flume2-

#是否按照时间滚动文件夹

a2.sinks.k1.hdfs.round = true

#多少时间单位创建一个新的文件夹

a2.sinks.k1.hdfs.roundValue = 1

#重新定义时间单位

a2.sinks.k1.hdfs.roundUnit = hour

#是否使用本地时间戳

a2.sinks.k1.hdfs.useLocalTimeStamp = true

#积攒多少个Event才flush到HDFS一次

a2.sinks.k1.hdfs.batchSize = 100

#设置文件类型，可支持压缩

a2.sinks.k1.hdfs.fileType = DataStream

#多久生成一个新的文件

a2.sinks.k1.hdfs.rollInterval = 600

#设置每个文件的滚动大小大概是128M

a2.sinks.k1.hdfs.rollSize = 134217700

#文件的滚动与Event数量无关

a2.sinks.k1.hdfs.rollCount = 0

# Describe the channel

a2.channels.c1.type = memory

a2.channels.c1.capacity = 1000

a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a2.sources.r1.channels = c1

a2.sinks.k1.channel = c1

3．创建flume-flume-dir.conf

配置上级Flume输出的Source，输出是到本地目录的Sink。

创建配置文件并打开

[atguigu@hadoop102 group1]$ touch flume-flume-dir.conf

[atguigu@hadoop102 group1]$ vim flume-flume-dir.conf

添加如下内容

# Name the components on this agent

a3.sources = r1

a3.sinks = k1

a3.channels = c2

# Describe/configure the source

a3.sources.r1.type = avro

a3.sources.r1.bind = hadoop102

a3.sources.r1.port = 4142

# Describe the sink

a3.sinks.k1.type = file\_roll

a3.sinks.k1.sink.directory = /opt/module/data/flume3

# Describe the channel

a3.channels.c2.type = memory

a3.channels.c2.capacity = 1000

a3.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel

a3.sources.r1.channels = c2

a3.sinks.k1.channel = c2

提示：输出的本地目录必须是已经存在的目录，如果该目录不存在，并不会创建新的目录。

4．执行配置文件

分别开启对应配置文件：flume-flume-dir，flume-flume-hdfs，flume-file-flume。

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group1/flume-flume-dir.conf

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group1/flume-flume-hdfs.conf

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group1/flume-file-flume.conf

5．启动Hadoop和Hive

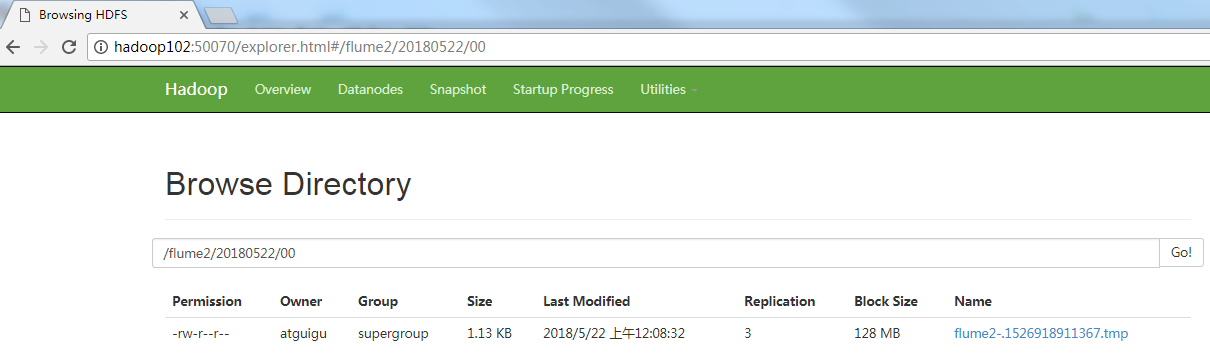
[atguigu@hadoop102 hadoop-2.7.2]$ sbin/start-dfs.sh

[atguigu@hadoop103 hadoop-2.7.2]$ sbin/start-yarn.sh

[atguigu@hadoop102 hive]$ bin/hive

hive (default)>

6．检查HDFS上数据



7检查/opt/module/datas/flume3目录中数据

[atguigu@hadoop102 flume3]$ ll

总用量 8

-rw-rw-r--. 1 atguigu atguigu 5942 5月 22 00:09 1526918887550-3

## 3.5 单数据源多出口案例(Sink组)

单Source、Channel多Sink(负载均衡)如图7-3所示。

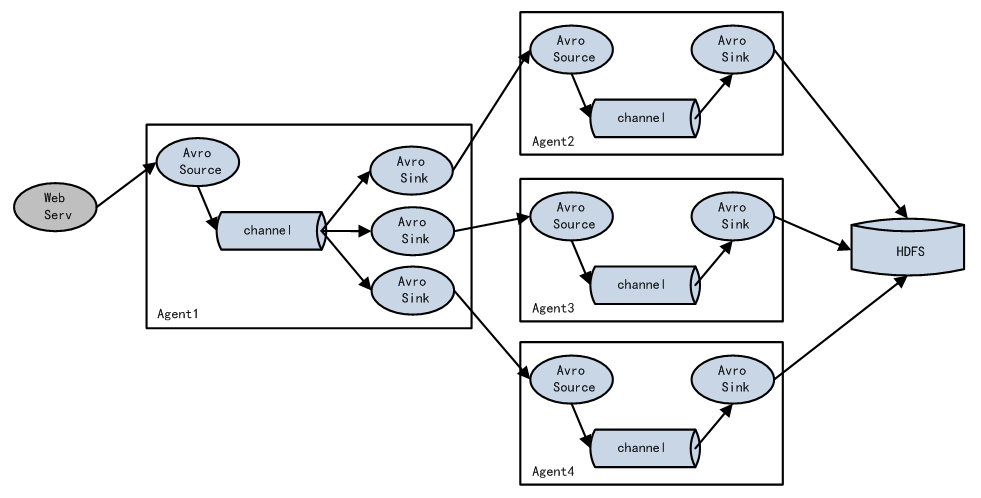
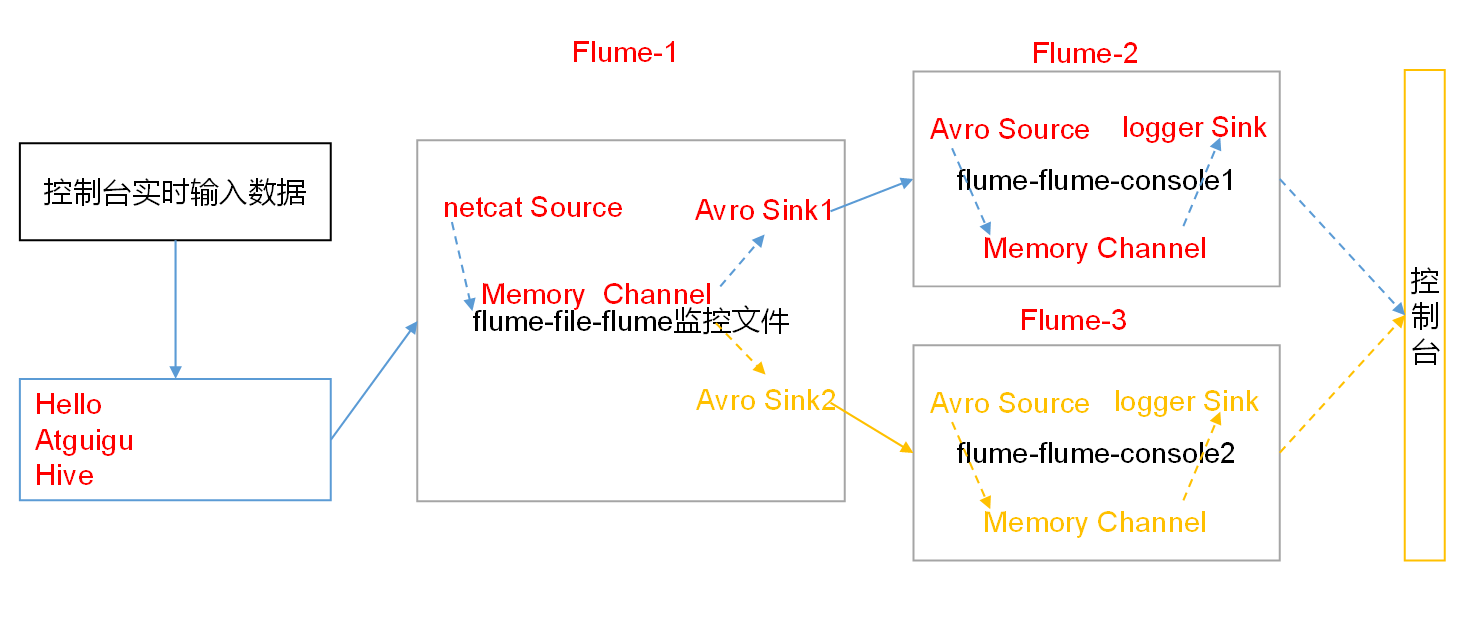


图7-3 单Source、Channel多Sink

1）案例需求：使用Flume-1监控文件变动，Flume-1将变动内容传递给Flume-2，Flume-2负责存储到HDFS。同时Flume-1将变动内容传递给Flume-3，Flume-3也负责存储到HDFS

2）需求分析：

单数据源多出口案例（sink组）



3）实现步骤：

0．准备工作

在/opt/module/flume/job目录下创建group2文件夹

[atguigu@hadoop102 job]$ cd group2/

1．创建flume-netcat-flume.conf

配置1个接收日志文件的source和1个channel、两个sink，分别输送给flume-flume-console1和flume-flume-console2。

创建配置文件并打开

[atguigu@hadoop102 group2]$ touch flume-netcat-flume.conf

[atguigu@hadoop102 group2]$ vim flume-netcat-flume.conf

添加如下内容

# Name the components on this agent

a1.sources = r1

a1.channels = c1

a1.sinkgroups = g1

a1.sinks = k1 k2

# Describe/configure the source

a1.sources.r1.type = netcat

a1.sources.r1.bind = localhost

a1.sources.r1.port = 44444

a1.sinkgroups.g1.processor.type = load\_balance

a1.sinkgroups.g1.processor.backoff = true

a1.sinkgroups.g1.processor.selector = round\_robin

a1.sinkgroups.g1.processor.selector.maxTimeOut=10000

# Describe the sink

a1.sinks.k1.type = avro

a1.sinks.k1.hostname = hadoop102

a1.sinks.k1.port = 4141

a1.sinks.k2.type = avro

a1.sinks.k2.hostname = hadoop102

a1.sinks.k2.port = 4142

# Describe the channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinkgroups.g1.sinks = k1 k2

a1.sinks.k1.channel = c1

a1.sinks.k2.channel = c1

注：Avro是由Hadoop创始人Doug Cutting创建的一种语言无关的数据序列化和RPC框架。

注：RPC（Remote Procedure Call）—远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

2．创建flume-flume-console1.conf

配置上级Flume输出的Source，输出是到本地控制台。

创建配置文件并打开

[atguigu@hadoop102 group2]$ touch flume-flume-console1.conf

[atguigu@hadoop102 group2]$ vim flume-flume-console1.conf

添加如下内容

# Name the components on this agent

a2.sources = r1

a2.sinks = k1

a2.channels = c1

# Describe/configure the source

a2.sources.r1.type = avro

a2.sources.r1.bind = hadoop102

a2.sources.r1.port = 4141

# Describe the sink

a2.sinks.k1.type = logger

# Describe the channel

a2.channels.c1.type = memory

a2.channels.c1.capacity = 1000

a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a2.sources.r1.channels = c1

a2.sinks.k1.channel = c1

3．创建flume-flume-console2.conf

配置上级Flume输出的Source，输出是到本地控制台。

创建配置文件并打开

[atguigu@hadoop102 group2]$ touch flume-flume-console2.conf

[atguigu@hadoop102 group2]$ vim flume-flume-console2.conf

添加如下内容

# Name the components on this agent

a3.sources = r1

a3.sinks = k1

a3.channels = c2

# Describe/configure the source

a3.sources.r1.type = avro

a3.sources.r1.bind = hadoop102

a3.sources.r1.port = 4142

# Describe the sink

a3.sinks.k1.type = logger

# Describe the channel

a3.channels.c2.type = memory

a3.channels.c2.capacity = 1000

a3.channels.c2.transactionCapacity = 100

# Bind the source and sink to the channel

a3.sources.r1.channels = c2

a3.sinks.k1.channel = c2

4．执行配置文件

分别开启对应配置文件：flume-flume-console2，flume-flume-console1，flume-netcat-flume。

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group2/flume-flume-console2.conf -Dflume.root.logger=INFO,console

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group2/flume-flume-console1.conf -Dflume.root.logger=INFO,console

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group2/flume-netcat-flume.conf

5. 使用netcat工具向本机的44444端口发送内容

$ nc localhost 44444

6. 查看Flume2及Flume3的控制台打印日志

## 3.6 多数据源汇总案例

多Source汇总数据到单Flume如图7-4所示。

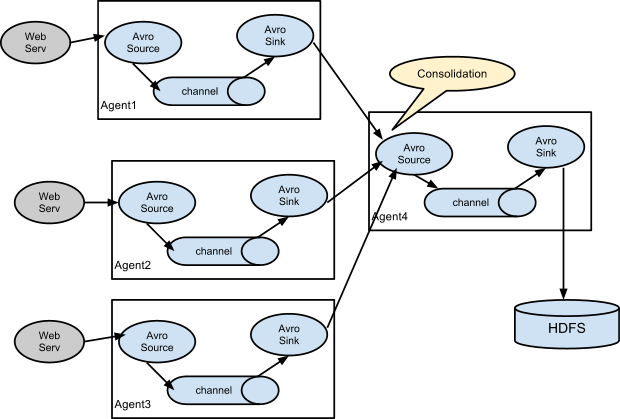


图7-4多Flume汇总数据到单Flume

1. 案例需求：

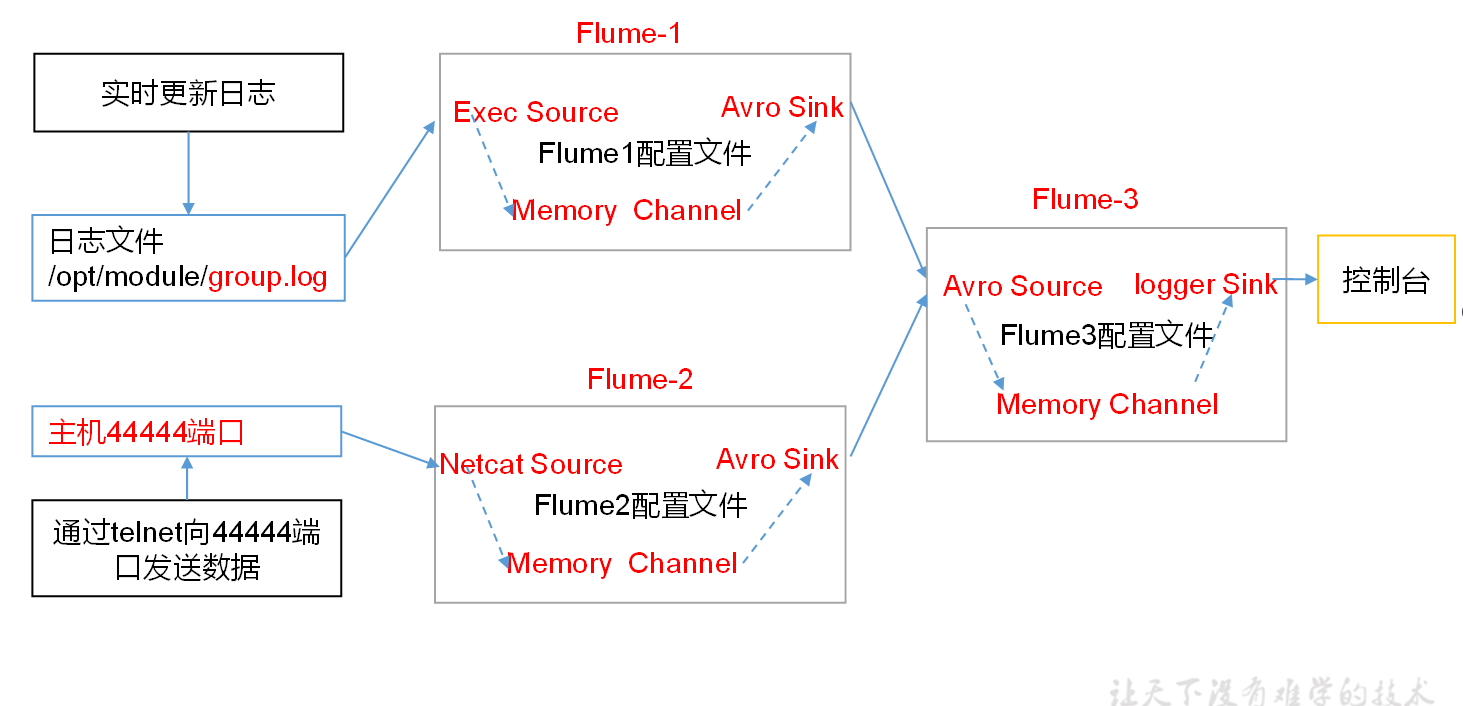
hadoop103上的Flume-1监控文件/opt/module/group.log，

hadoop102上的Flume-2监控某一个端口的数据流，

Flume-1与Flume-2将数据发送给hadoop104上的Flume-3，Flume-3将最终数据打印到控制台。

2）需求分析：

多数据源汇总案例



3）实现步骤：

0．准备工作

分发Flume

[atguigu@hadoop102 module]$ xsync flume

在hadoop102、hadoop103以及hadoop104的/opt/module/flume/job目录下创建一个group3文件夹。

[atguigu@hadoop102 job]$ mkdir group3

[atguigu@hadoop103 job]$ mkdir group3

[atguigu@hadoop104 job]$ mkdir group3

1．创建flume1-logger-flume.conf

配置Source用于监控hive.log文件，配置Sink输出数据到下一级Flume。

在hadoop103上创建配置文件并打开

[atguigu@hadoop103 group3]$ touch flume1-logger-flume.conf

[atguigu@hadoop103 group3]$ vim flume1-logger-flume.conf

添加如下内容

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F /opt/module/group.log

a1.sources.r1.shell = /bin/bash -c

# Describe the sink

a1.sinks.k1.type = avro

a1.sinks.k1.hostname = hadoop104

a1.sinks.k1.port = 4141

# Describe the channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

2．创建flume2-netcat-flume.conf

配置Source监控端口44444数据流，配置Sink数据到下一级Flume：

在hadoop102上创建配置文件并打开

[atguigu@hadoop102 group3]$ touch flume2-netcat-flume.conf

[atguigu@hadoop102 group3]$ vim flume2-netcat-flume.conf

添加如下内容

# Name the components on this agent

a2.sources = r1

a2.sinks = k1

a2.channels = c1

# Describe/configure the source

a2.sources.r1.type = netcat

a2.sources.r1.bind = hadoop102

a2.sources.r1.port = 44444

# Describe the sink

a2.sinks.k1.type = avro

a2.sinks.k1.hostname = hadoop104

a2.sinks.k1.port = 4141

# Use a channel which buffers events in memory

a2.channels.c1.type = memory

a2.channels.c1.capacity = 1000

a2.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a2.sources.r1.channels = c1

a2.sinks.k1.channel = c1

3．创建flume3-flume-logger.conf

配置source用于接收flume1与flume2发送过来的数据流，最终合并后sink到控制台。

在hadoop104上创建配置文件并打开

[atguigu@hadoop104 group3]$ touch flume3-flume-logger.conf

[atguigu@hadoop104 group3]$ vim flume3-flume-logger.conf

添加如下内容

# Name the components on this agent

a3.sources = r1

a3.sinks = k1

a3.channels = c1

# Describe/configure the source

a3.sources.r1.type = avro

a3.sources.r1.bind = hadoop104

a3.sources.r1.port = 4141

# Describe the sink

# Describe the sink

a3.sinks.k1.type = logger

# Describe the channel

a3.channels.c1.type = memory

a3.channels.c1.capacity = 1000

a3.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a3.sources.r1.channels = c1

a3.sinks.k1.channel = c1

4．执行配置文件

分别开启对应配置文件：flume3-flume-logger.conf，flume2-netcat-flume.conf，flume1-logger-flume.conf。

[atguigu@hadoop104 flume]$ bin/flume-ng agent --conf conf/ --name a3 --conf-file job/group3/flume3-flume-logger.conf -Dflume.root.logger=INFO,console

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a2 --conf-file job/group3/flume2-netcat-flume.conf

[atguigu@hadoop103 flume]$ bin/flume-ng agent --conf conf/ --name a1 --conf-file job/group3/flume1-logger-flume.conf

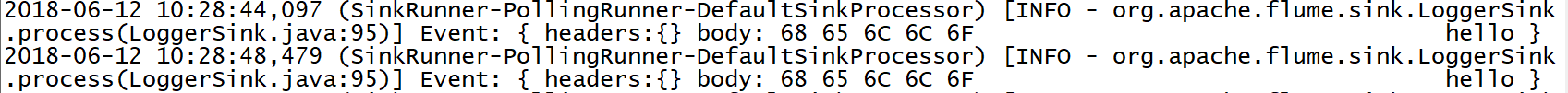
5．在hadoop103上向/opt/module目录下的group.log追加内容

[atguigu@hadoop103 module]$ echo 'hello' > group.log

6．在hadoop102上向44444端口发送数据

[atguigu@hadoop102 flume]$ telnet hadoop102 44444

7.检查hadoop104上数据



# 第4章 Flume监控之Ganglia

## 4.1 Ganglia的安装与部署

**1) 安装httpd服务与php**

[atguigu@hadoop102 flume]$ sudo yum -y install httpd php

**2) 安装其他依赖**

[atguigu@hadoop102 flume]$ sudo yum -y install rrdtool perl-rrdtool rrdtool-devel

[atguigu@hadoop102 flume]$ sudo yum -y install apr-devel

**3) 安装ganglia**

[atguigu@hadoop102 flume]$ sudo rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86\_64/epel-release-6-8.noarch.rpm

[atguigu@hadoop102 flume]$ sudo yum -y install ganglia-gmetad

[atguigu@hadoop102 flume]$ sudo yum -y install ganglia-web

[atguigu@hadoop102 flume]$ sudo yum install -y ganglia-gmond

Ganglia由gmond、gmetad和gweb三部分组成。

gmond（Ganglia Monitoring Daemon）是一种轻量级服务，安装在每台需要收集指标数据的节点主机上。使用gmond，你可以很容易收集很多系统指标数据，如CPU、内存、磁盘、网络和活跃进程的数据等。

gmetad（Ganglia Meta Daemon）整合所有信息，并将其以RRD格式存储至磁盘的服务。

gweb（Ganglia Web）Ganglia可视化工具，gweb是一种利用浏览器显示gmetad所存储数据的PHP前端。在Web界面中以图表方式展现集群的运行状态下收集的多种不同指标数据。

**4) 修改配置文件/etc/httpd/conf.d/ganglia.conf**

[atguigu@hadoop102 flume]$ sudo vim /etc/httpd/conf.d/ganglia.conf

**修改为红颜色的配置：**

# Ganglia monitoring system php web frontend

Alias /ganglia /usr/share/ganglia

<Location /ganglia>

Order deny,allow

#Deny from all

Allow from all

# Allow from 127.0.0.1

# Allow from ::1

# Allow from .example.com

</Location>

**5) 修改配置文件/etc/ganglia/gmetad.conf**

[atguigu@hadoop102 flume]$ sudo vim /etc/ganglia/gmetad.conf

**修改为：**

data\_source "hadoop102" 192.168.1.102

**6) 修改配置文件/etc/ganglia/gmond.conf**

[atguigu@hadoop102 flume]$ sudo vim /etc/ganglia/gmond.conf

**修改为：**

cluster {

name = "hadoop102"

owner = "unspecified"

latlong = "unspecified"

url = "unspecified"

}

udp\_send\_channel {

#bind\_hostname = yes # Highly recommended, soon to be default.

# This option tells gmond to use a source address

# that resolves to the machine's hostname. Without

# this, the metrics may appear to come from any

# interface and the DNS names associated with

# those IPs will be used to create the RRDs.

# mcast\_join = 239.2.11.71

host = 192.168.1.102

port = 8649

ttl = 1

}

udp\_recv\_channel {

# mcast\_join = 239.2.11.71

port = 8649

bind = 192.168.1.102

retry\_bind = true

# Size of the UDP buffer. If you are handling lots of metrics you really

# should bump it up to e.g. 10MB or even higher.

# buffer = 10485760

}

**7) 修改配置文件/etc/selinux/config**

[atguigu@hadoop102 flume]$ sudo vim /etc/selinux/config

**修改为：**

# This file controls the state of SELinux on the system.

# SELINUX= can take one of these three values:

# enforcing - SELinux security policy is enforced.

# permissive - SELinux prints warnings instead of enforcing.

# disabled - No SELinux policy is loaded.

SELINUX=disabled

# SELINUXTYPE= can take one of these two values:

# targeted - Targeted processes are protected,

# mls - Multi Level Security protection.

SELINUXTYPE=targeted

尖叫提示：selinux本次生效关闭必须重启，如果此时不想重启，可以临时生效之：

[atguigu@hadoop102 flume]$ sudo setenforce 0

**5) 启动ganglia**

[atguigu@hadoop102 flume]$ sudo service httpd start

[atguigu@hadoop102 flume]$ sudo service gmetad start

[atguigu@hadoop102 flume]$ sudo service gmond start

**6) 打开网页浏览ganglia页面**

<http://192.168.1.102/ganglia>

尖叫提示：如果完成以上操作依然出现权限不足错误，请修改/var/lib/ganglia目录的权限：

[atguigu@hadoop102 flume]$ sudo chmod -R 777 /var/lib/ganglia

## 4.2 操作Flume测试监控

**1) 修改/opt/module/flume/conf目录下的flume-env.sh配置：**

JAVA\_OPTS="-Dflume.monitoring.type=ganglia

-Dflume.monitoring.hosts=192.168.1.102:8649

-Xms100m

-Xmx200m"

**2) 启动Flume任务**

[atguigu@hadoop102 flume]$ bin/flume-ng agent \

--conf conf/ \

--name a1 \

--conf-file job/flume-netcat-logger.conf \

-Dflume.root.logger==INFO,console \

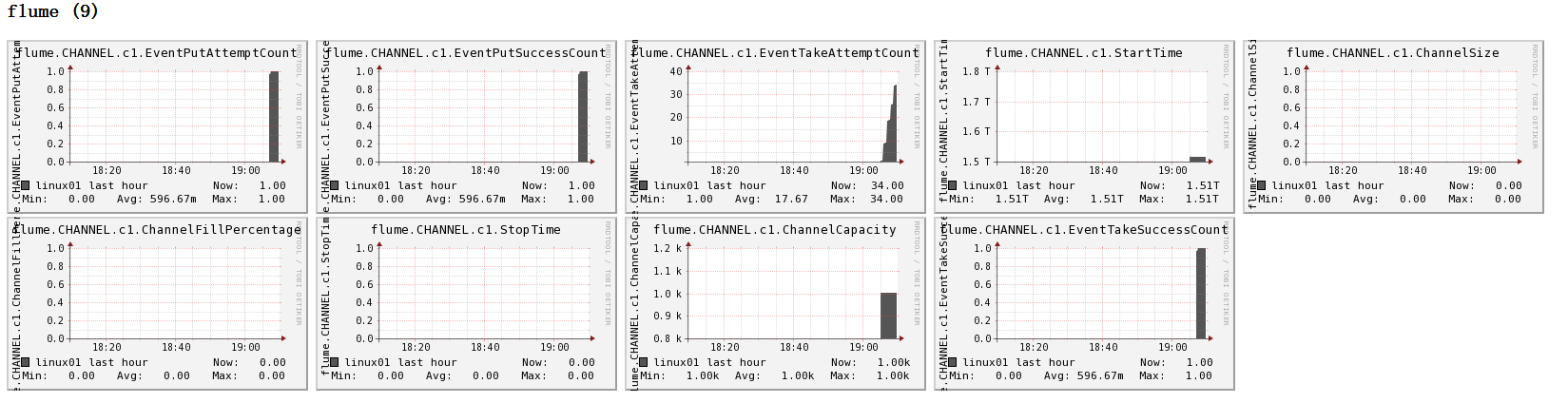
-Dflume.monitoring.type=ganglia \

-Dflume.monitoring.hosts=192.168.1.102:8649

**3) 发送数据观察ganglia监测图**

[atguigu@hadoop102 flume]$ nc localhost 44444

**样式如图：**



图例说明：

|  |  |
| --- | --- |
| 字段（图表名称） | 字段含义 |
| EventPutAttemptCount | source尝试写入channel的事件总数量 |
| EventPutSuccessCount | 成功写入channel且提交的事件总数量 |
| EventTakeAttemptCount | sink尝试从channel拉取事件的总数量。这不意味着每次事件都被返回，因为sink拉取的时候channel可能没有任何数据。 |
| EventTakeSuccessCount | sink成功读取的事件的总数量 |
| StartTime | channel启动的时间（毫秒） |
| StopTime | channel停止的时间（毫秒） |
| ChannelSize | 目前channel中事件的总数量 |
| ChannelFillPercentage | channel占用百分比 |
| ChannelCapacity | channel的容量 |

# 第5章 自定义Source

## 5.1 介绍

Source是负责接收数据到Flume Agent的组件。Source组件可以处理各种类型、各种格式的日志数据，包括avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy。官方提供的source类型已经很多，但是有时候并不能满足实际开发当中的需求，此时我们就需要根据实际需求自定义某些source。

官方也提供了自定义source的接口：

[https://flume.apache.org/FlumeDeveloperGuide.html#source](https://flume.apache.org/FlumeDeveloperGuide.html" \l "source)根据官方说明自定义MySource需要继承AbstractSource类并实现Configurable和PollableSource接口。

实现相应方法：

getBackOffSleepIncrement() //暂不用

getMaxBackOffSleepInterval() //暂不用

configure(Context context) //初始化context（读取配置文件内容）

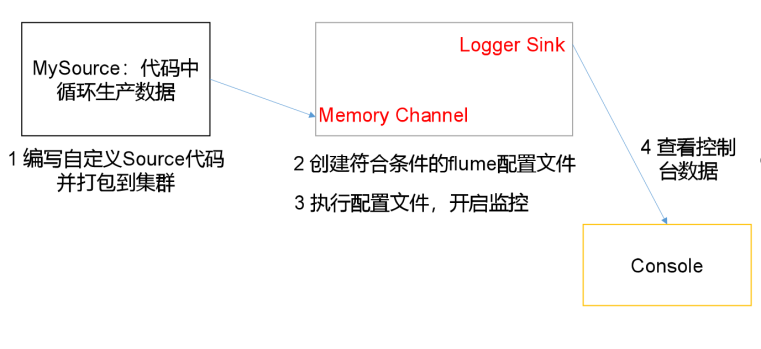
process() //获取数据封装成event并写入channel，这个方法将被循环调用。

使用场景：读取MySQL数据或者其他文件系统。

## 5.2 需求

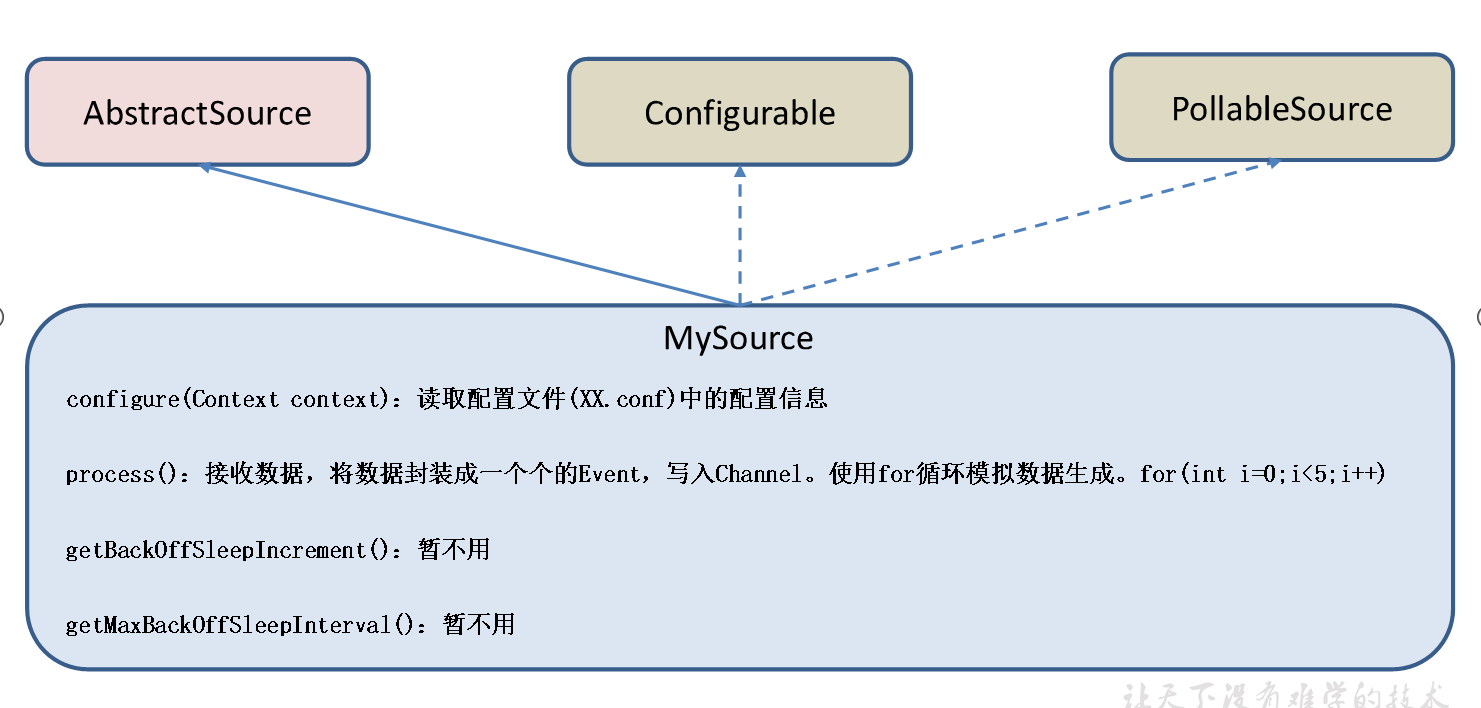
使用flume接收数据，并给每条数据添加前缀，输出到控制台。前缀可从flume配置文件中配置。

**自定义source需求：**



## 5.2 分析

自定义Source需求分析：



## 5.3 编码

**导入pom依赖**

<dependencies>

<dependency>

<groupId>org.apache.flume</groupId>

<artifactId>flume-ng-core</artifactId>

<version>1.7.0</version>

</dependency>

</dependencies>

package com.atguigu;

import org.apache.flume.Context;

import org.apache.flume.EventDeliveryException;

import org.apache.flume.PollableSource;

import org.apache.flume.conf.Configurable;

import org.apache.flume.event.SimpleEvent;

import org.apache.flume.source.AbstractSource;

import java.util.HashMap;

public class MySource extends AbstractSource implements Configurable, PollableSource {

//定义配置文件将来要读取的字段

private Long delay;

private String field;

//初始化配置信息

@Override

public void configure(Context context) {

delay = context.getLong("delay");

field = context.getString("field", "Hello!");

}

@Override

public Status process() throws EventDeliveryException {

try {

//创建事件头信息

HashMap<String, String> hearderMap = new HashMap<>();

//创建事件

SimpleEvent event = new SimpleEvent();

//循环封装事件

for (int i = 0; i < 5; i++) {

//给事件设置头信息

event.setHeaders(hearderMap);

//给事件设置内容

event.setBody((field + i).getBytes());

//将事件写入channel

getChannelProcessor().processEvent(event);

Thread.sleep(delay);

}

} catch (Exception e) {

e.printStackTrace();

return Status.BACKOFF;

}

return Status.READY;

}

@Override

public long getBackOffSleepIncrement() {

return 0;

}

@Override

public long getMaxBackOffSleepInterval() {

return 0;

}

}

## 5.4 测试

1）打包

将写好的代码打包，并放到flume的lib目录（/opt/module/flume）下。

2）配置文件

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = com.atguigu.MySource

a1.sources.r1.delay = 1000

#a1.sources.r1.field = atguigu

# Describe the sink

a1.sinks.k1.type = logger

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

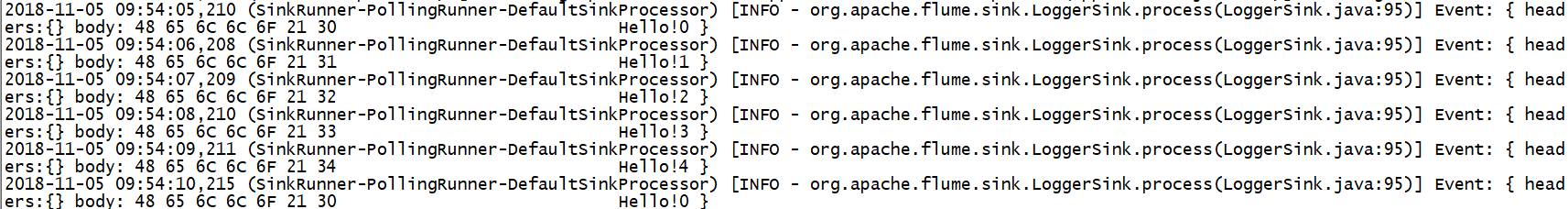
3）开启任务

[atguigu@hadoop102 flume]$ pwd

/opt/module/flume

[atguigu@hadoop102 flume]$ bin/flume-ng agent -c conf/ -f job/mysource.conf -n a1 -Dflume.root.logger=INFO,console

4）结果展示



# 第6章 自定义Sink

## 6.1 介绍

Sink不断地轮询Channel中的事件且批量地移除它们，并将这些事件批量写入到存储或索引系统、或者被发送到另一个Flume Agent。

Sink是完全事务性的。在从Channel批量删除数据之前，每个Sink用Channel启动一个事务。批量事件一旦成功写出到存储系统或下一个Flume Agent，Sink就利用Channel提交事务。事务一旦被提交，该Channel从自己的内部缓冲区删除事件。

Sink组件目的地包括hdfs、logger、avro、thrift、ipc、file、null、HBase、solr、自定义。官方提供的Sink类型已经很多，但是有时候并不能满足实际开发当中的需求，此时我们就需要根据实际需求自定义某些Sink。

官方也提供了自定义source的接口：

[https://flume.apache.org/FlumeDeveloperGuide.html#sink](https://flume.apache.org/FlumeDeveloperGuide.html" \l "sink)根据官方说明自定义MySink需要继承AbstractSink类并实现Configurable接口。

实现相应方法：

configure(Context context)//初始化context（读取配置文件内容）

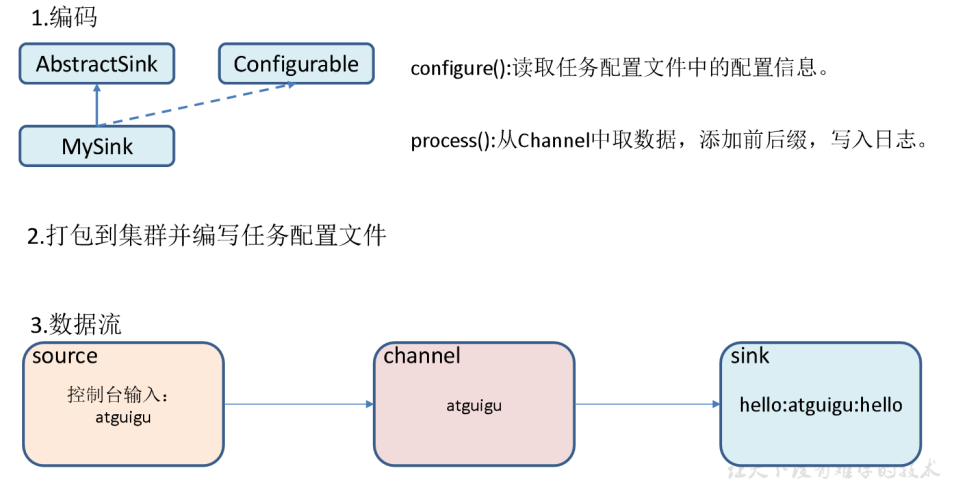
process()//从Channel读取获取数据（event），这个方法将被循环调用。

使用场景：读取Channel数据写入MySQL或者其他文件系统。

## 6.2 需求

使用flume接收数据，并在Sink端给每条数据添加前缀和后缀，输出到控制台。前后缀可在flume任务配置文件中配置。

流程分析：



## 6.3 编码

package com.atguigu;

import org.apache.flume.\*;

import org.apache.flume.conf.Configurable;

import org.apache.flume.sink.AbstractSink;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class MySink extends AbstractSink implements Configurable {

//创建Logger对象

private static final Logger LOG = LoggerFactory.getLogger(AbstractSink.class);

private String prefix;

private String suffix;

@Override

public Status process() throws EventDeliveryException {

//声明返回值状态信息

Status status;

//获取当前Sink绑定的Channel

Channel ch = getChannel();

//获取事务

Transaction txn = ch.getTransaction();

//声明事件

Event event;

//开启事务

txn.begin();

//读取Channel中的事件，直到读取到事件结束循环

while (true) {

event = ch.take();

if (event != null) {

break;

}

}

try {

//处理事件（打印）

LOG.info(prefix + new String(event.getBody()) + suffix);

//事务提交

txn.commit();

status = Status.READY;

} catch (Exception e) {

//遇到异常，事务回滚

txn.rollback();

status = Status.BACKOFF;

} finally {

//关闭事务

txn.close();

}

return status;

}

@Override

public void configure(Context context) {

//读取配置文件内容，有默认值

prefix = context.getString("prefix", "hello:");

//读取配置文件内容，无默认值

suffix = context.getString("suffix");

}

}

## 6.4 测试

1）打包

将写好的代码打包，并放到flume的lib目录（/opt/module/flume）下。

2）配置文件

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = netcat

a1.sources.r1.bind = localhost

a1.sources.r1.port = 44444

# Describe the sink

a1.sinks.k1.type = com.atguigu.MySink

#a1.sinks.k1.prefix = atguigu:

a1.sinks.k1.suffix = :atguigu

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

3）开启任务

[atguigu@hadoop102 flume]$ pwd

/opt/module/flume

[atguigu@hadoop102 flume]$ bin/flume-ng agent -c conf/ -f job/mysink.conf -n a1 -Dflume.root.logger=INFO,console

[atguigu@hadoop102 ~]$ nc localhost 44444

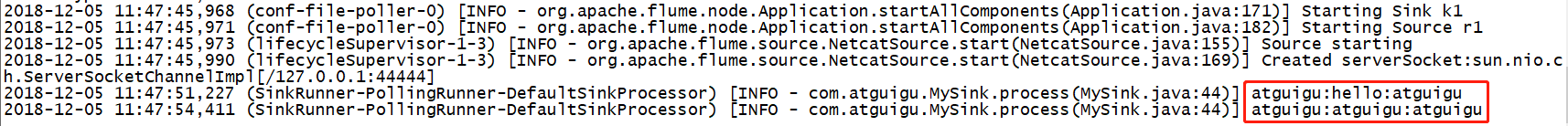
hello

OK

atguigu

OK

4）结果展示



# 第7章 知识扩展

## 7.1 常见正则表达式语法

|  |  |
| --- | --- |
| 元字符 | 描述 |
| ^ | 匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| \* | 匹配前面的子表达式任意次。例如，zo\*能匹配“z”，“zo”以及“zoo”。\*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。  注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |

## 7.2 自定义MySQLSource

### 7.2.1 自定义Source说明

实时监控MySQL，从MySQL中获取数据传输到HDFS或者其他存储框架，所以此时需要我们自己实现MySQLSource。

官方也提供了自定义source的接口：

官网说明：[https://flume.apache.org/FlumeDeveloperGuide.html#source](https://flume.apache.org/FlumeDeveloperGuide.html" \l "source)

### 7.2.2 自定义MySQLSource组成

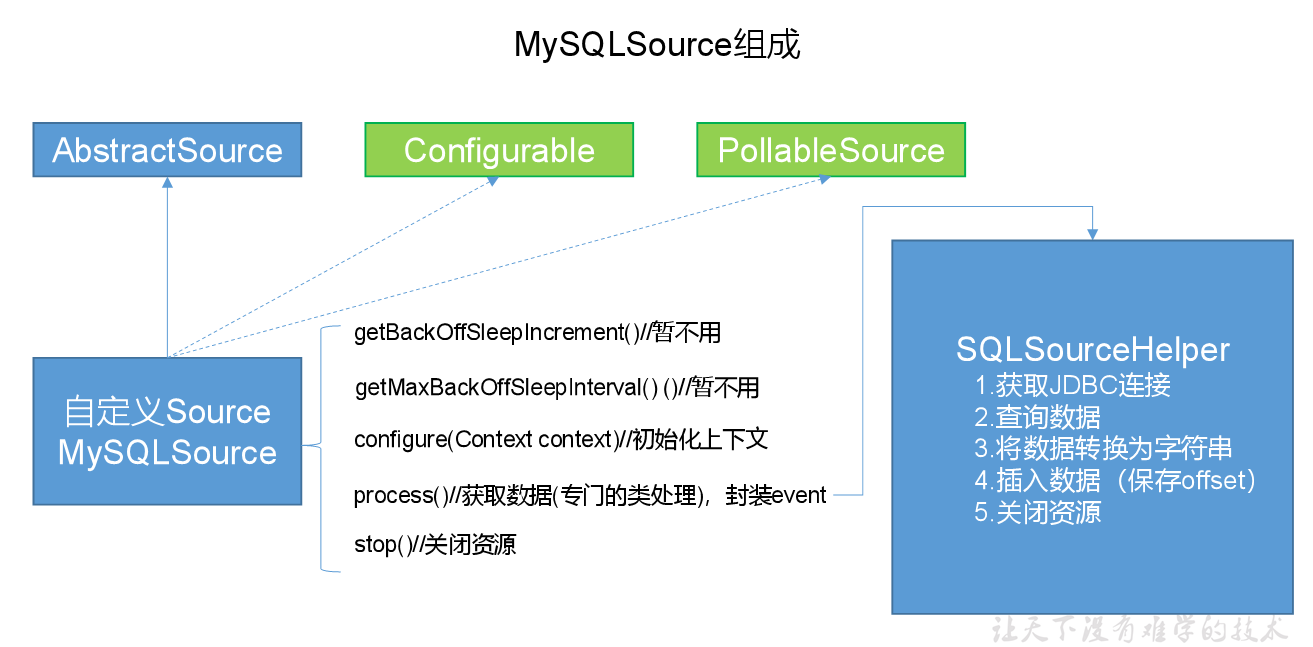


图6-1 自定义MySQLSource组成

### 7.2.3 自定义MySQLSource步骤

根据官方说明自定义mysqlsource需要继承AbstractSource类并实现Configurable和PollableSource接口。

实现相应方法：

getBackOffSleepIncrement()//暂不用

getMaxBackOffSleepInterval()//暂不用

configure(Context context)//初始化context

process()//获取数据（从mysql获取数据，业务处理比较复杂，所以我们定义一个专门的类——SQLSourceHelper来处理跟mysql的交互），封装成event并写入channel，这个方法被循环调用

stop()//关闭相关的资源

PollableSource：从source中提取数据，将其发送到channel。

Configurable：实现了Configurable的任何类都含有一个context，使用context获取配置信息。

### 7.2.4 代码实现

**1、导入pom依赖**

<dependencies>

<dependency>

<groupId>org.apache.flume</groupId>

<artifactId>flume-ng-core</artifactId>

<version>1.7.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

</dependencies>

**2、添加配置信息**

在classpath下添加jdbc.properties和log4j.properties

jdbc.properties:

dbDriver=com.mysql.jdbc.Driver

dbUrl=jdbc:mysql://hadoop102:3306/mysqlsource?useUnicode=true&characterEncoding=utf-8  
dbUser=root  
dbPassword=000000

log4j.properties:

#--------console-----------

log4j.rootLogger=info,myconsole,myfile  
log4j.appender.myconsole=org.apache.log4j.ConsoleAppender  
log4j.appender.myconsole.layout=org.apache.log4j.SimpleLayout  
#log4j.appender.myconsole.layout.ConversionPattern =%d [%t] %-5p [%c] - %m%n  
  
#log4j.rootLogger=error,myfile  
log4j.appender.myfile=org.apache.log4j.DailyRollingFileAppender  
log4j.appender.myfile.File=/tmp/flume.log  
log4j.appender.myfile.layout=org.apache.log4j.PatternLayout  
log4j.appender.myfile.layout.ConversionPattern =%d [%t] %-5p [%c] - %m%n

**3、SQLSourceHelper**

**1) 属性说明：**

|  |  |
| --- | --- |
| 属性 | 说明（括号中为默认值） |
| runQueryDelay | 查询时间间隔（10000） |
| batchSize | 缓存大小（100） |
| startFrom | 查询语句开始id（0） |
| currentIndex | 查询语句当前id，每次查询之前需要查元数据表 |
| recordSixe | 查询返回条数 |
| table | 监控的表名 |
| columnsToSelect | 查询字段（\*） |
| customQuery | 用户传入的查询语句 |
| query | 查询语句 |
| defaultCharsetResultSet | 编码格式（UTF-8） |

**2) 方法说明：**

|  |  |
| --- | --- |
| 方法 | 说明 |
| SQLSourceHelper(Context context) | 构造方法，初始化属性及获取JDBC连接 |
| InitConnection(String url, String user, String pw) | 获取JDBC连接 |
| checkMandatoryProperties() | 校验相关属性是否设置（实际开发中可增加内容） |
| buildQuery() | 根据实际情况构建sql语句，返回值String |
| executeQuery() | 执行sql语句的查询操作，返回值List<List<Object>> |
| getAllRows(List<List<Object>> queryResult) | 将查询结果转换为String，方便后续操作 |
| updateOffset2DB(int size) | 根据每次查询结果将offset写入元数据表 |
| execSql(String sql) | 具体执行sql语句方法 |
| getStatusDBIndex(int startFrom) | 获取元数据表中的offset |
| queryOne(String sql) | 获取元数据表中的offset实际sql语句执行方法 |
| close() | 关闭资源 |

1. **代码实现：**

package com.atguigu.source;

import org.apache.flume.Context;

import org.apache.flume.conf.ConfigurationException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.io.IOException;

import java.sql.\*;

import java.text.ParseException;

import java.util.ArrayList;

import java.util.List;

import java.util.Properties;

public class SQLSourceHelper {

private static final Logger LOG = LoggerFactory.getLogger(SQLSourceHelper.class);

private int runQueryDelay, //两次查询的时间间隔

startFrom, //开始id

currentIndex, //当前id

recordSixe = 0, //每次查询返回结果的条数

maxRow; //每次查询的最大条数

private String table, //要操作的表

columnsToSelect, //用户传入的查询的列

customQuery, //用户传入的查询语句

query, //构建的查询语句

defaultCharsetResultSet;//编码集

//上下文，用来获取配置文件

private Context context;

//为定义的变量赋值（默认值），可在flume任务的配置文件中修改

private static final int DEFAULT\_QUERY\_DELAY = 10000;

private static final int DEFAULT\_START\_VALUE = 0;

private static final int DEFAULT\_MAX\_ROWS = 2000;

private static final String DEFAULT\_COLUMNS\_SELECT = "\*";

private static final String DEFAULT\_CHARSET\_RESULTSET = "UTF-8";

private static Connection conn = null;

private static PreparedStatement ps = null;

private static String connectionURL, connectionUserName, connectionPassword;

//加载静态资源

static {

Properties p = new Properties();

try {

p.load(SQLSourceHelper.class.getClassLoader().getResourceAsStream("jdbc.properties"));

connectionURL = p.getProperty("dbUrl");

connectionUserName = p.getProperty("dbUser");

connectionPassword = p.getProperty("dbPassword");

Class.forName(p.getProperty("dbDriver"));

} catch (IOException | ClassNotFoundException e) {

LOG.error(e.toString());

}

}

//获取JDBC连接

private static Connection InitConnection(String url, String user, String pw) {

try {

Connection conn = DriverManager.getConnection(url, user, pw);

if (conn == null)

throw new SQLException();

return conn;

} catch (SQLException e) {

e.printStackTrace();

}

return null;

}

//构造方法

SQLSourceHelper(Context context) throws ParseException {

//初始化上下文

this.context = context;

//有默认值参数：获取flume任务配置文件中的参数，读不到的采用默认值

this.columnsToSelect = context.getString("columns.to.select", DEFAULT\_COLUMNS\_SELECT);

this.runQueryDelay = context.getInteger("run.query.delay", DEFAULT\_QUERY\_DELAY);

this.startFrom = context.getInteger("start.from", DEFAULT\_START\_VALUE);

this.defaultCharsetResultSet = context.getString("default.charset.resultset", DEFAULT\_CHARSET\_RESULTSET);

//无默认值参数：获取flume任务配置文件中的参数

this.table = context.getString("table");

this.customQuery = context.getString("custom.query");

connectionURL = context.getString("connection.url");

connectionUserName = context.getString("connection.user");

connectionPassword = context.getString("connection.password");

conn = InitConnection(connectionURL, connectionUserName, connectionPassword);

//校验相应的配置信息，如果没有默认值的参数也没赋值，抛出异常

checkMandatoryProperties();

//获取当前的id

currentIndex = getStatusDBIndex(startFrom);

//构建查询语句

query = buildQuery();

}

//校验相应的配置信息（表，查询语句以及数据库连接的参数）

private void checkMandatoryProperties() {

if (table == null) {

throw new ConfigurationException("property table not set");

}

if (connectionURL == null) {

throw new ConfigurationException("connection.url property not set");

}

if (connectionUserName == null) {

throw new ConfigurationException("connection.user property not set");

}

if (connectionPassword == null) {

throw new ConfigurationException("connection.password property not set");

}

}

//构建sql语句

private String buildQuery() {

String sql = "";

//获取当前id

currentIndex = getStatusDBIndex(startFrom);

LOG.info(currentIndex + "");

if (customQuery == null) {

sql = "SELECT " + columnsToSelect + " FROM " + table;

} else {

sql = customQuery;

}

StringBuilder execSql = new StringBuilder(sql);

//以id作为offset

if (!sql.contains("where")) {

execSql.append(" where ");

execSql.append("id").append(">").append(currentIndex);

return execSql.toString();

} else {

int length = execSql.toString().length();

return execSql.toString().substring(0, length - String.valueOf(currentIndex).length()) + currentIndex;

}

}

//执行查询

List<List<Object>> executeQuery() {

try {

//每次执行查询时都要重新生成sql，因为id不同

customQuery = buildQuery();

//存放结果的集合

List<List<Object>> results = new ArrayList<>();

if (ps == null) {

//

ps = conn.prepareStatement(customQuery);

}

ResultSet result = ps.executeQuery(customQuery);

while (result.next()) {

//存放一条数据的集合（多个列）

List<Object> row = new ArrayList<>();

//将返回结果放入集合

for (int i = 1; i <= result.getMetaData().getColumnCount(); i++) {

row.add(result.getObject(i));

}

results.add(row);

}

LOG.info("execSql:" + customQuery + "\nresultSize:" + results.size());

return results;

} catch (SQLException e) {

LOG.error(e.toString());

// 重新连接

conn = InitConnection(connectionURL, connectionUserName, connectionPassword);

}

return null;

}

//将结果集转化为字符串，每一条数据是一个list集合，将每一个小的list集合转化为字符串

List<String> getAllRows(List<List<Object>> queryResult) {

List<String> allRows = new ArrayList<>();

if (queryResult == null || queryResult.isEmpty())

return allRows;

StringBuilder row = new StringBuilder();

for (List<Object> rawRow : queryResult) {

Object value = null;

for (Object aRawRow : rawRow) {

value = aRawRow;

if (value == null) {

row.append(",");

} else {

row.append(aRawRow.toString()).append(",");

}

}

allRows.add(row.toString());

row = new StringBuilder();

}

return allRows;

}

//更新offset元数据状态，每次返回结果集后调用。必须记录每次查询的offset值，为程序中断续跑数据时使用，以id为offset

void updateOffset2DB(int size) {

//以source\_tab做为KEY，如果不存在则插入，存在则更新（每个源表对应一条记录）

String sql = "insert into flume\_meta(source\_tab,currentIndex) VALUES('"

+ this.table

+ "','" + (recordSixe += size)

+ "') on DUPLICATE key update source\_tab=values(source\_tab),currentIndex=values(currentIndex)";

LOG.info("updateStatus Sql:" + sql);

execSql(sql);

}

//执行sql语句

private void execSql(String sql) {

try {

ps = conn.prepareStatement(sql);

LOG.info("exec::" + sql);

ps.execute();

} catch (SQLException e) {

e.printStackTrace();

}

}

//获取当前id的offset

private Integer getStatusDBIndex(int startFrom) {

//从flume\_meta表中查询出当前的id是多少

String dbIndex = queryOne("select currentIndex from flume\_meta where source\_tab='" + table + "'");

if (dbIndex != null) {

return Integer.parseInt(dbIndex);

}

//如果没有数据，则说明是第一次查询或者数据表中还没有存入数据，返回最初传入的值

return startFrom;

}

//查询一条数据的执行语句(当前id)

private String queryOne(String sql) {

ResultSet result = null;

try {

ps = conn.prepareStatement(sql);

result = ps.executeQuery();

while (result.next()) {

return result.getString(1);

}

} catch (SQLException e) {

e.printStackTrace();

}

return null;

}

//关闭相关资源

void close() {

try {

ps.close();

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

int getCurrentIndex() {

return currentIndex;

}

void setCurrentIndex(int newValue) {

currentIndex = newValue;

}

int getRunQueryDelay() {

return runQueryDelay;

}

String getQuery() {

return query;

}

String getConnectionURL() {

return connectionURL;

}

private boolean isCustomQuerySet() {

return (customQuery != null);

}

Context getContext() {

return context;

}

public String getConnectionUserName() {

return connectionUserName;

}

public String getConnectionPassword() {

return connectionPassword;

}

String getDefaultCharsetResultSet() {

return defaultCharsetResultSet;

}

}

**4、MySQLSource**

代码实现：

package com.atguigu.source;

import org.apache.flume.Context;

import org.apache.flume.Event;

import org.apache.flume.EventDeliveryException;

import org.apache.flume.PollableSource;

import org.apache.flume.conf.Configurable;

import org.apache.flume.event.SimpleEvent;

import org.apache.flume.source.AbstractSource;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.text.ParseException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

public class SQLSource extends AbstractSource implements Configurable, PollableSource {

//打印日志

private static final Logger LOG = LoggerFactory.getLogger(SQLSource.class);

//定义sqlHelper

private SQLSourceHelper sqlSourceHelper;

@Override

public long getBackOffSleepIncrement() {

return 0;

}

@Override

public long getMaxBackOffSleepInterval() {

return 0;

}

@Override

public void configure(Context context) {

try {

//初始化

sqlSourceHelper = new SQLSourceHelper(context);

} catch (ParseException e) {

e.printStackTrace();

}

}

@Override

public Status process() throws EventDeliveryException {

try {

//查询数据表

List<List<Object>> result = sqlSourceHelper.executeQuery();

//存放event的集合

List<Event> events = new ArrayList<>();

//存放event头集合

HashMap<String, String> header = new HashMap<>();

//如果有返回数据，则将数据封装为event

if (!result.isEmpty()) {

List<String> allRows = sqlSourceHelper.getAllRows(result);

Event event = null;

for (String row : allRows) {

event = new SimpleEvent();

event.setBody(row.getBytes());

event.setHeaders(header);

events.add(event);

}

//将event写入channel

this.getChannelProcessor().processEventBatch(events);

//更新数据表中的offset信息

sqlSourceHelper.updateOffset2DB(result.size());

}

//等待时长

Thread.sleep(sqlSourceHelper.getRunQueryDelay());

return Status.READY;

} catch (InterruptedException e) {

LOG.error("Error procesing row", e);

return Status.BACKOFF;

}

}

@Override

public synchronized void stop() {

LOG.info("Stopping sql source {} ...", getName());

try {

//关闭资源

sqlSourceHelper.close();

} finally {

super.stop();

}

}

}

### 7.2.5 测试

**1、jar包准备**

1) 将mysql驱动包放入flume的lib目录下

[atguigu@hadoop102 flume]$ cp \

/opt/sorfware/mysql-libs/mysql-connector-java-5.1.27/mysql-connector-java-5.1.27-bin.jar \

/opt/module/flume/lib/

2) 打包项目并将jar包放入flume的lib目录下

**2、配置文件准备**

1）创建配置文件并打开

[atguigu@hadoop102 job]$ touch mysql.conf

[atguigu@hadoop102 job]$ vim mysql.conf

2）添加如下内容

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = com.atguigu.source.SQLSource

a1.sources.r1.connection.url = jdbc:mysql://192.168.1.102:3306/mysqlsource

a1.sources.r1.connection.user = root

a1.sources.r1.connection.password = 000000

a1.sources.r1.table = student

a1.sources.r1.columns.to.select = \*

#a1.sources.r1.incremental.column.name = id

#a1.sources.r1.incremental.value = 0

a1.sources.r1.run.query.delay=5000

# Describe the sink

a1.sinks.k1.type = logger

# Describe the channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

**3、mysql表准备**

1) 创建mysqlsource数据库

CREATE DATABASE mysqlsource；

2) 在mysqlsource数据库下创建数据表student和元数据表flume\_meta

CREATE TABLE `student` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

PRIMARY KEY (`id`)

);

CREATE TABLE `flume\_meta` (

`source\_tab` varchar(255) NOT NULL,

`currentIndex` varchar(255) NOT NULL,

PRIMARY KEY (`source\_tab`)

);

1. 向数据表中添加数据

1 zhangsan

2 lisi

3 wangwu

4 zhaoliu

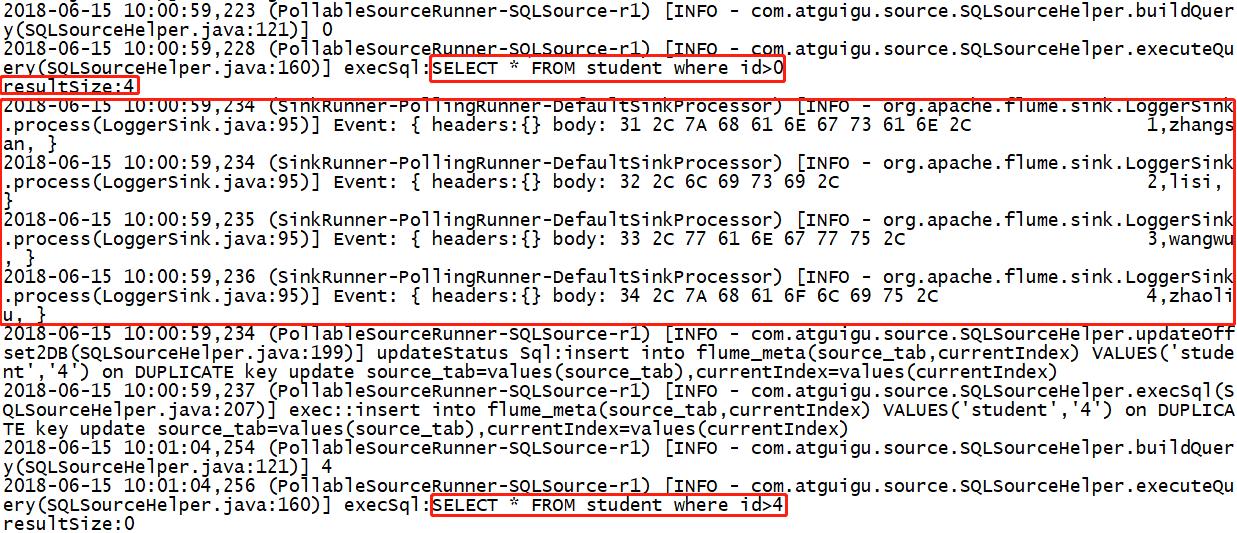
**4、测试并查看结果**

1. 任务执行

[atguigu@hadoop102 flume]$ bin/flume-ng agent --conf conf/ --name a1 \

--conf-file job/mysql.conf -Dflume.root.logger=INFO,console

1. 结果展示，如图所示：



## 7.3 练习

案例需求：

1）flume-1监控hive.log日志，flume-1的数据传送给flume-2，flume-2将数据追加到本地文件，同时将数据传输到flume-3。

2）flume-4监控本地另一个自己创建的文件any.txt，并将数据传送给flume-3。

3）flume-3将汇总数据写入到HDFS。

请先画出结构图，再开始编写任务脚本。

# 第8章 企业真实面试题（重点）

## 8.1 你是如何实现Flume数据传输的监控的

使用第三方框架Ganglia实时监控Flume。

## 8.2 Flume的Source，Sink，Channel的作用？你们Source是什么类型？

1、作用

（1）Source组件是专门用来收集数据的，可以处理各种类型、各种格式的日志数据，包括avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy

（2）Channel组件对采集到的数据进行缓存，可以存放在Memory或File中。

（3）Sink组件是用于把数据发送到目的地的组件，目的地包括Hdfs、Logger、avro、thrift、ipc、file、Hbase、solr、自定义。

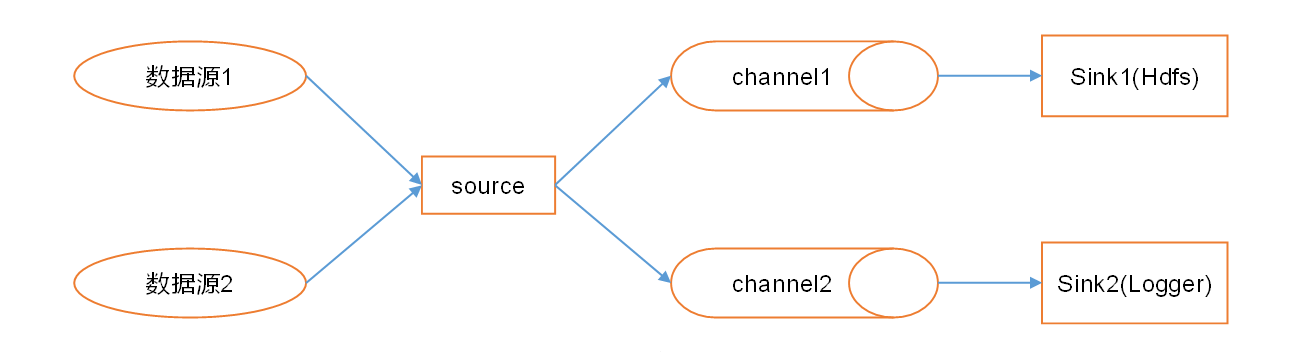
2、我公司采用的Source类型为：

（1）监控后台日志：exec

（2）监控后台产生日志的端口：netcat

Exec spooldir

## 8.3 Flume的Channel Selectors



Channel Selectors，可以让不同的项目日志通过不同的Channel到不同的Sink中去。官方文档上Channel Selectors有两种类型：Replicating Channel Selector（default）和Multiplexing Channel Selector

这两种Selector的区别是Replicating会将source过来的events发往所有channel，而Multiplexing可以选择该发往哪些Channel。

## 8.4 Flume参数调优

1. Source

增加Source个数（使用Tair Dir Source时可增加FileGroups个数）可以增大Source的读取数据的能力。例如：当某一个目录产生的文件过多时需要将这个文件目录拆分成多个文件目录，同时配置好多个Source 以保证Source有足够的能力获取到新产生的数据。

batchSize参数决定Source一次批量运输到Channel的event条数，适当调大这个参数可以提高Source搬运Event到Channel时的性能。

2. Channel

type 选择memory时Channel的性能最好，但是如果Flume进程意外挂掉可能会丢失数据。type选择file时Channel的容错性更好，但是性能上会比memory channel差。

使用file Channel时dataDirs配置多个不同盘下的目录可以提高性能。

Capacity 参数决定Channel可容纳最大的event条数。transactionCapacity 参数决定每次Source往channel里面写的最大event条数和每次Sink从channel里面读的最大event条数。transactionCapacity需要大于Source和Sink的batchSize参数。

3. Sink

增加Sink的个数可以增加Sink消费event的能力。Sink也不是越多越好够用就行，过多的Sink会占用系统资源，造成系统资源不必要的浪费。

batchSize参数决定Sink一次批量从Channel读取的event条数，适当调大这个参数可以提高Sink从Channel搬出event的性能。

## 8.5 Flume的事务机制

Flume的事务机制（类似数据库的事务机制）：Flume使用两个独立的事务分别负责从Soucrce到Channel，以及从Channel到Sink的事件传递。比如spooling directory source 为文件的每一行创建一个事件，一旦事务中所有的事件全部传递到Channel且提交成功，那么Source就将该文件标记为完成。同理，事务以类似的方式处理从Channel到Sink的传递过程，如果因为某种原因使得事件无法记录，那么事务将会回滚。且所有的事件都会保持到Channel中，等待重新传递。

## 8.6 Flume采集数据会丢失吗?

不会，Channel存储可以存储在File中，数据传输自身有事务。

# Sqoop

# 第1章 Sqoop简介

Sqoop是一款开源的工具，主要用于在Hadoop(Hive)与传统的数据库(mysql、postgresql...)间进行数据的传递，可以将一个关系型数据库（例如 ： MySQL ,Oracle ,Postgres等）中的数据导进到Hadoop的HDFS中，也可以将HDFS的数据导进到关系型数据库中。

Sqoop项目开始于2009年，最早是作为Hadoop的一个第三方模块存在，后来为了让使用者能够快速部署，也为了让开发人员能够更快速的迭代开发，Sqoop独立成为一个[Apache](https://baike.baidu.com/item/Apache/6265" \t "https://baike.baidu.com/item/sqoop/_blank)项目。

Sqoop2的最新版本是1.99.7。请注意，2与1不兼容，且特征不完整，它并不打算用于生产部署。

# 第2章 Sqoop原理

将导入或导出命令翻译成mapreduce程序来实现。

在翻译出的mapreduce中主要是对inputformat和outputformat进行定制。

# 第3章 Sqoop安装

安装Sqoop的前提是已经具备Java和Hadoop的环境。

## 3.1 下载并解压

1) 下载地址：<http://mirrors.hust.edu.cn/apache/sqoop/1.4.6/>

2) 上传安装包sqoop-1.4.6.bin\_\_hadoop-2.0.4-alpha.tar.gz到虚拟机中

3) 解压sqoop安装包到指定目录，如：

$ tar -zxf sqoop-1.4.6.bin\_\_hadoop-2.0.4-alpha.tar.gz -C /opt/module/

## 3.2 修改配置文件

Sqoop的配置文件与大多数大数据框架类似，在sqoop根目录下的conf目录中。

**1) 重命名配置文件**

$ mv sqoop-env-template.sh sqoop-env.sh

**2) 修改配置文件**

sqoop-env.sh

export HADOOP\_COMMON\_HOME=/opt/module/hadoop-2.7.2

export HADOOP\_MAPRED\_HOME=/opt/module/hadoop-2.7.2

export HIVE\_HOME=/opt/module/hive

export ZOOKEEPER\_HOME=/opt/module/zookeeper-3.4.10

export ZOOCFGDIR=/opt/module/zookeeper-3.4.10

export HBASE\_HOME=/opt/module/hbase

## 3.3 拷贝JDBC驱动

拷贝jdbc驱动到sqoop的lib目录下，如：

$ cp mysql-connector-java-5.1.27-bin.jar /opt/module/sqoop-1.4.6.bin\_\_hadoop-2.0.4-alpha/lib/

## 3.4 验证Sqoop

我们可以通过某一个command来验证sqoop配置是否正确：

$ bin/sqoop help

出现一些Warning警告（警告信息已省略），并伴随着帮助命令的输出：

Available commands:

codegen Generate code to interact with database records

create-hive-table Import a table definition into Hive

eval Evaluate a SQL statement and display the results

export Export an HDFS directory to a database table

help List available commands

import Import a table from a database to HDFS

import-all-tables Import tables from a database to HDFS

import-mainframe Import datasets from a mainframe server to HDFS

job Work with saved jobs

list-databases List available databases on a server

list-tables List available tables in a database

merge Merge results of incremental imports

metastore Run a standalone Sqoop metastore

version Display version information

## 3.5 测试Sqoop是否能够成功连接数据库

$ bin/sqoop list-databases --connect jdbc:mysql://hadoop102:3306/ --username root --password 000000

出现如下输出：

information\_schema

metastore

mysql

oozie

performance\_schema

# 第4章 Sqoop的简单使用案例

## 4.1 导入数据

在Sqoop中，“导入”概念指：从非大数据集群（RDBMS）向大数据集群（HDFS，HIVE，HBASE）中传输数据，叫做：导入，即使用import关键字。

### 4.1.1 RDBMS到HDFS

1) 确定Mysql服务开启正常

2) 在Mysql中新建一张表并插入一些数据

$ mysql -uroot -p000000

mysql> create database company;

mysql> create table company.staff(id int(4) primary key not null auto\_increment, name varchar(255), sex varchar(255));

mysql> insert into company.staff(name, sex) values('Thomas', 'Male');

mysql> insert into company.staff(name, sex) values('Catalina', 'FeMale');

3) 导入数据

**（1）全部导入**

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--table staff \

--target-dir /user/company \

--delete-target-dir \

--num-mappers 1 \

--fields-terminated-by "\t"

**（2）查询导入**

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--target-dir /user/company \

--delete-target-dir \

--num-mappers 1 \

--fields-terminated-by "\t" \

--query 'select name,sex from staff where id <=1 and $CONDITIONS;'

提示：must contain '$CONDITIONS' in WHERE clause.

如果query后使用的是双引号，则$CONDITIONS前必须加转移符，防止shell识别为自己的变量。

**（3）导入指定列**

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--target-dir /user/company \

--delete-target-dir \

--num-mappers 1 \

--fields-terminated-by "\t" \

--columns id,sex \

--table staff

提示：columns中如果涉及到多列，用逗号分隔，分隔时不要添加空格

**（4）使用sqoop关键字筛选查询导入数据**

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--target-dir /user/company \

--delete-target-dir \

--num-mappers 1 \

--fields-terminated-by "\t" \

--table staff \

--where "id=1"

### 4.1.2 RDBMS到Hive

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--table staff \

--num-mappers 1 \

--hive-import \

--fields-terminated-by "\t" \

--hive-overwrite \

--hive-table staff\_hive

提示：该过程分为两步，第一步将数据导入到HDFS，第二步将导入到HDFS的数据迁移到Hive仓库，第一步默认的临时目录是/user/atguigu/表名

### 4.1.3 RDBMS到Hbase

$ bin/sqoop import \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--table staff \

--columns "id,name,sex" \

--column-family "info" \

--hbase-create-table \

--hbase-row-key "id" \

--hbase-table "hbase\_company" \

--num-mappers 1 \

--split-by id

提示：sqoop1.4.6只支持HBase1.0.1之前的版本的自动创建HBase表的功能

解决方案：手动创建HBase表

hbase> create 'hbase\_company','info'

**(5) 在HBase中scan这张表得到如下内容**

hbase> scan 'hbase\_company'

## 4.2 导出数据

在Sqoop中，从大数据集群（HDFS，HIVE，HBASE）向非大数据集群（RDBMS）中传输数据，叫做“导出”，即使用export关键字。

### 4.2.1 HIVE/HDFS到RDBMS

$ bin/sqoop export \

--connect jdbc:mysql://hadoop102:3306/company \

--username root \

--password 000000 \

--table staff \

--num-mappers 1 \

--export-dir /user/hive/warehouse/staff\_hive \

--input-fields-terminated-by "\t"

提示：Mysql中如果表不存在，不会自动创建

## 4.3 脚本打包

使用opt格式的文件打包sqoop命令，然后执行

**1) 创建一个.opt文件**

$ mkdir opt

$ touch opt/job\_HDFS2RDBMS.opt

**2) 编写sqoop脚本**

$ vi opt/job\_HDFS2RDBMS.opt

export

--connect jdbc:mysql://hadoop102:3306/company

--username root

--password 000000

--table staff

--num-mappers 1

--export-dir /user/hive/warehouse/staff\_hive

--input-fields-terminated-by "\t"

**3) 执行该脚本**

$ bin/sqoop --options-file opt/job\_HDFS2RDBMS.opt

# 第5章 Sqoop一些常用命令及参数

## 5.1 常用命令列举

这里给大家列出来了一部分Sqoop操作时的常用参数，以供参考，需要深入学习的可以参看对应类的源代码。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **命令** | **类** | **说明** |
| 1 | import | ImportTool | 将数据导入到集群 |
| 2 | export | ExportTool | 将集群数据导出 |
| 3 | codegen | CodeGenTool | 获取数据库中某张表数据生成Java并打包Jar |
| 4 | create-hive-table | CreateHiveTableTool | 创建Hive表 |
| 5 | eval | EvalSqlTool | 查看SQL执行结果 |
| 6 | import-all-tables | ImportAllTablesTool | 导入某个数据库下所有表到HDFS中 |
| 7 | job | JobTool | 用来生成一个sqoop的任务，生成后，该任务并不执行，除非使用命令执行该任务。 |
| 8 | list-databases | ListDatabasesTool | 列出所有数据库名 |
| 9 | list-tables | ListTablesTool | 列出某个数据库下所有表 |
| 10 | merge | MergeTool | 将HDFS中不同目录下面的数据合在一起，并存放在指定的目录中 |
| 11 | metastore | MetastoreTool | 记录sqoop job的元数据信息，如果不启动metastore实例，则默认的元数据存储目录为：~/.sqoop，如果要更改存储目录，可以在配置文件sqoop-site.xml中进行更改。 |
| 12 | help | HelpTool | 打印sqoop帮助信息 |
| 13 | version | VersionTool | 打印sqoop版本信息 |

## 5.2 命令&参数详解

刚才列举了一些Sqoop的常用命令，对于不同的命令，有不同的参数，让我们来一一列举说明。

首先来我们来介绍一下公用的参数，所谓公用参数，就是大多数命令都支持的参数。

### 5.2.1 公用参数：数据库连接

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --connect | 连接关系型数据库的URL |
| 2 | --connection-manager | 指定要使用的连接管理类 |
| 3 | --driver | Hadoop根目录 |
| 4 | --help | 打印帮助信息 |
| 5 | --password | 连接数据库的密码 |
| 6 | --username | 连接数据库的用户名 |
| 7 | --verbose | 在控制台打印出详细信息 |

### 5.2.2 公用参数：import

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --enclosed-by <char> | 给字段值前加上指定的字符 |
| 2 | --escaped-by <char> | 对字段中的双引号加转义符 |
| 3 | --fields-terminated-by <char> | 设定每个字段是以什么符号作为结束，默认为逗号 |
| 4 | --lines-terminated-by <char> | 设定每行记录之间的分隔符，默认是\n |
| 5 | --mysql-delimiters | Mysql默认的分隔符设置，字段之间以逗号分隔，行之间以\n分隔，默认转义符是\，字段值以单引号包裹。 |
| 6 | --optionally-enclosed-by <char> | 给带有双引号或单引号的字段值前后加上指定字符。 |

### 5.2.3 公用参数：export

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --input-enclosed-by <char> | 对字段值前后加上指定字符 |
| 2 | --input-escaped-by <char> | 对含有转移符的字段做转义处理 |
| 3 | --input-fields-terminated-by <char> | 字段之间的分隔符 |
| 4 | --input-lines-terminated-by <char> | 行之间的分隔符 |
| 5 | --input-optionally-enclosed-by <char> | 给带有双引号或单引号的字段前后加上指定字符 |

### 5.2.4 公用参数：hive

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --hive-delims-replacement <arg> | 用自定义的字符串替换掉数据中的\r\n和\013 \010等字符 |
| 2 | --hive-drop-import-delims | 在导入数据到hive时，去掉数据中的\r\n\013\010这样的字符 |
| 3 | --map-column-hive <arg> | 生成hive表时，可以更改生成字段的数据类型 |
| 4 | --hive-partition-key | 创建分区，后面直接跟分区名，分区字段的默认类型为string |
| 5 | --hive-partition-value <v> | 导入数据时，指定某个分区的值 |
| 6 | --hive-home <dir> | hive的安装目录，可以通过该参数覆盖之前默认配置的目录 |
| 7 | --hive-import | 将数据从关系数据库中导入到hive表中 |
| 8 | --hive-overwrite | 覆盖掉在hive表中已经存在的数据 |
| 9 | --create-hive-table | 默认是false，即，如果目标表已经存在了，那么创建任务失败。 |
| 10 | --hive-table | 后面接要创建的hive表,默认使用MySQL的表名 |
| 11 | --table | 指定关系数据库的表名 |

公用参数介绍完之后，我们来按照命令介绍命令对应的特有参数。

### 5.2.5 命令&参数：import

将关系型数据库中的数据导入到HDFS（包括Hive，HBase）中，如果导入的是Hive，那么当Hive中没有对应表时，则自动创建。

**1) 命令：**

如：导入数据到hive中

|  |
| --- |
| $ bin/sqoop import \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --hive-import |

如：增量导入数据到hive中，mode=append

|  |
| --- |
| append导入：  $ bin/sqoop import \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --num-mappers 1 \  --fields-terminated-by "\t" \  --target-dir /user/hive/warehouse/staff\_hive \  --check-column id \  --incremental append \  --last-value 3 |

尖叫提示：append不能与--hive-等参数同时使用（Append mode for hive imports is not yet supported. Please remove the parameter --append-mode）

如：增量导入数据到hdfs中，mode=lastmodified

|  |
| --- |
| 先在mysql中建表并插入几条数据：  mysql> create table company.staff\_timestamp(id int(4), name varchar(255), sex varchar(255), last\_modified timestamp DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP);  mysql> insert into company.staff\_timestamp (id, name, sex) values(1, 'AAA', 'female');  mysql> insert into company.staff\_timestamp (id, name, sex) values(2, 'BBB', 'female');  先导入一部分数据：  $ bin/sqoop import \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff\_timestamp \  --delete-target-dir \  --m 1  再增量导入一部分数据：  mysql> insert into company.staff\_timestamp (id, name, sex) values(3, 'CCC', 'female');  $ bin/sqoop import \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff\_timestamp \  --check-column last\_modified \  --incremental lastmodified \  --last-value "2017-09-28 22:20:38" \  --m 1 \  --append |

尖叫提示：使用lastmodified方式导入数据要指定增量数据是要--append（追加）还是要--merge-key（合并）

尖叫提示：last-value指定的值是会包含于增量导入的数据中

**2) 参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --append | 将数据追加到HDFS中已经存在的DataSet中，如果使用该参数，sqoop会把数据先导入到临时文件目录，再合并。 |
| 2 | --as-avrodatafile | 将数据导入到一个Avro数据文件中 |
| 3 | --as-sequencefile | 将数据导入到一个sequence文件中 |
| 4 | --as-textfile | 将数据导入到一个普通文本文件中 |
| 5 | --boundary-query <statement> | 边界查询，导入的数据为该参数的值（一条sql语句）所执行的结果区间内的数据。 |
| 6 | --columns <col1, col2, col3> | 指定要导入的字段 |
| 7 | --direct | 直接导入模式，使用的是关系数据库自带的导入导出工具，以便加快导入导出过程。 |
| 8 | --direct-split-size | 在使用上面direct直接导入的基础上，对导入的流按字节分块，即达到该阈值就产生一个新的文件 |
| 9 | --inline-lob-limit | 设定大对象数据类型的最大值 |
| 10 | --m或–num-mappers | 启动N个map来并行导入数据，默认4个。 |
| 11 | --query或--e <statement> | 将查询结果的数据导入，使用时必须伴随参--target-dir，--hive-table，如果查询中有where条件，则条件后必须加上$CONDITIONS关键字 |
| 12 | --split-by <column-name> | 按照某一列来切分表的工作单元，不能与--autoreset-to-one-mapper连用（请参考官方文档） |
| 13 | --table <table-name> | 关系数据库的表名 |
| 14 | --target-dir <dir> | 指定HDFS路径 |
| 15 | --warehouse-dir <dir> | 与14参数不能同时使用，导入数据到HDFS时指定的目录 |
| 16 | --where | 从关系数据库导入数据时的查询条件 |
| 17 | --z或--compress | 允许压缩 |
| 18 | --compression-codec | 指定hadoop压缩编码类，默认为gzip(Use Hadoop codec default gzip) |
| 19 | --null-string <null-string> | string类型的列如果null，替换为指定字符串 |
| 20 | --null-non-string <null-string> | 非string类型的列如果null，替换为指定字符串 |
| 21 | --check-column <col> | 作为增量导入判断的列名 |
| 22 | --incremental <mode> | mode：append或lastmodified |
| 23 | --last-value <value> | 指定某一个值，用于标记增量导入的位置 |

### 5.2.6 命令&参数：export

从HDFS（包括Hive和HBase）中奖数据导出到关系型数据库中。

**1) 命令：**

**如：**

|  |
| --- |
| $ bin/sqoop export \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --export-dir /user/company \  --input-fields-terminated-by "\t" \  --num-mappers 1 |

**2) 参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --direct | 利用数据库自带的导入导出工具，以便于提高效率 |
| 2 | --export-dir <dir> | 存放数据的HDFS的源目录 |
| 3 | -m或--num-mappers <n> | 启动N个map来并行导入数据，默认4个 |
| 4 | --table <table-name> | 指定导出到哪个RDBMS中的表 |
| 5 | --update-key <col-name> | 对某一列的字段进行更新操作 |
| 6 | --update-mode <mode> | updateonly  allowinsert(默认) |
| 7 | --input-null-string <null-string> | 请参考import该类似参数说明 |
| 8 | --input-null-non-string <null-string> | 请参考import该类似参数说明 |
| 9 | --staging-table <staging-table-name> | 创建一张临时表，用于存放所有事务的结果，然后将所有事务结果一次性导入到目标表中，防止错误。 |
| 10 | --clear-staging-table | 如果第9个参数非空，则可以在导出操作执行前，清空临时事务结果表 |

### 5.2.7 命令&参数：codegen

将关系型数据库中的表映射为一个Java类，在该类中有各列对应的各个字段。

如：

|  |
| --- |
| $ bin/sqoop codegen \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --bindir /home/admin/Desktop/staff \  --class-name Staff \  --fields-terminated-by "\t" |

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --bindir <dir> | 指定生成的Java文件、编译成的class文件及将生成文件打包为jar的文件输出路径 |
| 2 | --class-name <name> | 设定生成的Java文件指定的名称 |
| 3 | --outdir <dir> | 生成Java文件存放的路径 |
| 4 | --package-name <name> | 包名，如com.z，就会生成com和z两级目录 |
| 5 | --input-null-non-string <null-str> | 在生成的Java文件中，可以将null字符串或者不存在的字符串设置为想要设定的值（例如空字符串） |
| 6 | --input-null-string <null-str> | 将null字符串替换成想要替换的值（一般与5同时使用） |
| 7 | --map-column-java <arg> | 数据库字段在生成的Java文件中会映射成各种属性，且默认的数据类型与数据库类型保持对应关系。该参数可以改变默认类型，例如：--map-column-java id=long, name=String |
| 8 | --null-non-string <null-str> | 在生成Java文件时，可以将不存在或者null的字符串设置为其他值 |
| 9 | --null-string <null-str> | 在生成Java文件时，将null字符串设置为其他值（一般与8同时使用） |
| 10 | --table <table-name> | 对应关系数据库中的表名，生成的Java文件中的各个属性与该表的各个字段一一对应 |

### 5.2.8 命令&参数：create-hive-table

生成与关系数据库表结构对应的hive表结构。

**命令：**

如：

|  |
| --- |
| $ bin/sqoop create-hive-table \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --hive-table hive\_staff |

**参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --hive-home <dir> | Hive的安装目录，可以通过该参数覆盖掉默认的Hive目录 |
| 2 | --hive-overwrite | 覆盖掉在Hive表中已经存在的数据 |
| 3 | --create-hive-table | 默认是false，如果目标表已经存在了，那么创建任务会失败 |
| 4 | --hive-table | 后面接要创建的hive表 |
| 5 | --table | 指定关系数据库的表名 |

### 5.2.9 命令&参数：eval

可以快速的使用SQL语句对关系型数据库进行操作，经常用于在import数据之前，了解一下SQL语句是否正确，数据是否正常，并可以将结果显示在控制台。

**命令：**

如：

|  |
| --- |
| $ bin/sqoop eval \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --query "SELECT \* FROM staff" |

**参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --query或--e | 后跟查询的SQL语句 |

### 5.2.10 命令&参数：import-all-tables

可以将RDBMS中的所有表导入到HDFS中，每一个表都对应一个HDFS目录

**命令：**

如：

|  |
| --- |
| $ bin/sqoop import-all-tables \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --warehouse-dir /all\_tables |

**参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --as-avrodatafile | 这些参数的含义均和import对应的含义一致 |
| 2 | --as-sequencefile |
| 3 | --as-textfile |
| 4 | --direct |
| 5 | --direct-split-size <n> |
| 6 | --inline-lob-limit <n> |
| 7 | --m或—num-mappers <n> |
| 8 | --warehouse-dir <dir> |
| 9 | -z或--compress |
| 10 | --compression-codec |

### 5.2.11 命令&参数：job

用来生成一个sqoop任务，生成后不会立即执行，需要手动执行。

**命令：**

如：

|  |
| --- |
| $ bin/sqoop job \  --create myjob -- import-all-tables \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000  $ bin/sqoop job \  --list  $ bin/sqoop job \  --exec myjob |

尖叫提示：注意import-all-tables和它左边的--之间有一个空格

尖叫提示：如果需要连接metastore，则--meta-connect jdbc:hsqldb:hsql://linux01:16000/sqoop

参数：

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --create <job-id> | 创建job参数 |
| 2 | --delete <job-id> | 删除一个job |
| 3 | --exec <job-id> | 执行一个job |
| 4 | --help | 显示job帮助 |
| 5 | --list | 显示job列表 |
| 6 | --meta-connect <jdbc-uri> | 用来连接metastore服务 |
| 7 | --show <job-id> | 显示一个job的信息 |
| 8 | --verbose | 打印命令运行时的详细信息 |

尖叫提示：在执行一个job时，如果需要手动输入数据库密码，可以做如下优化

|  |
| --- |
| <property>  <name>sqoop.metastore.client.record.password</name>  <value>true</value>  <description>If true, allow saved passwords in the metastore.</description>  </property> |

### 5.2.12 命令&参数：list-databases

**命令：**

如：

|  |
| --- |
| $ bin/sqoop list-databases \  --connect jdbc:mysql://hadoop102:3306/ \  --username root \  --password 000000 |

**参数：**与公用参数一样

### 5.2.13 命令&参数：list-tables

**命令：**

如：

|  |
| --- |
| $ bin/sqoop list-tables \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 |

**参数：**与公用参数一样

### 5.2.14 命令&参数：merge

将HDFS中不同目录下面的数据合并在一起并放入指定目录中

数据环境：

|  |
| --- |
| new\_staff  1 AAA male  2 BBB male  3 CCC male  4 DDD male  old\_staff  1 AAA female  2 CCC female  3 BBB female  6 DDD female |

尖叫提示：上边数据的列之间的分隔符应该为\t，行与行之间的分割符为\n，如果直接复制，请检查之。

**命令：**

如：

|  |
| --- |
| 创建JavaBean：  $ bin/sqoop codegen \  --connect jdbc:mysql://hadoop102:3306/company \  --username root \  --password 000000 \  --table staff \  --bindir /home/admin/Desktop/staff \  --class-name Staff \  --fields-terminated-by "\t"  开始合并：  $ bin/sqoop merge \  --new-data /test/new/ \  --onto /test/old/ \  --target-dir /test/merged \  --jar-file /home/admin/Desktop/staff/Staff.jar \  --class-name Staff \  --merge-key id  结果：  1 AAA MALE  2 BBB MALE  3 CCC MALE  4 DDD MALE  6 DDD FEMALE |

参数：

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --new-data <path> | HDFS 待合并的数据目录，合并后在新的数据集中保留 |
| 2 | --onto <path> | HDFS合并后，重复的部分在新的数据集中被覆盖 |
| 3 | --merge-key <col> | 合并键，一般是主键ID |
| 4 | --jar-file <file> | 合并时引入的jar包，该jar包是通过Codegen工具生成的jar包 |
| 5 | --class-name <class> | 对应的表名或对象名，该class类是包含在jar包中的 |
| 6 | --target-dir <path> | 合并后的数据在HDFS里存放的目录 |

### 5.2.15 命令&参数：metastore

记录了Sqoop job的元数据信息，如果不启动该服务，那么默认job元数据的存储目录为~/.sqoop，可在sqoop-site.xml中修改。

**命令：**

如：启动sqoop的metastore服务

|  |
| --- |
| $ bin/sqoop metastore |

**参数：**

|  |  |  |
| --- | --- | --- |
| **序号** | **参数** | **说明** |
| 1 | --shutdown | 关闭metastore |

# Oozie

# 第1章 Oozie简介

Oozie英文翻译为：驯象人。一个基于工作流引擎的开源框架，由Cloudera公司贡献给Apache，提供对Hadoop MapReduce、Pig Jobs的任务调度与协调。Oozie需要部署到Java Servlet容器中运行。主要用于定时调度任务，多任务可以按照执行的逻辑顺序调度。

# 第2章 Oozie的功能模块介绍

## 2.1模块

**1) Workflow**

顺序执行流程节点，支持fork（分支多个节点），join（合并多个节点为一个）

**2) Coordinator**

定时触发workflow

**3) Bundle**

绑定多个Coordinator

## 2.2 Workflow常用节点

**1) 控制流节点（Control Flow Nodes）**

控制流节点一般都是定义在工作流开始或者结束的位置，比如start,end,kill等。以及提供工作流的执行路径机制，如decision，fork，join等。

**2) 动作节点（Action Nodes）**

负责执行具体动作的节点，比如：拷贝文件，执行某个Shell脚本等等。

# 第3章 Oozie的部署

## 3.1 部署Hadoop（CDH版本的）

### 3.1.2 修改Hadoop配置

**core-site.xml**

<!-- Oozie Server的Hostname -->

<property>

<name>hadoop.proxyuser.atguigu.hosts</name>

<value>\*</value>

</property>

<!-- 允许被Oozie代理的用户组 -->

<property>

<name>hadoop.proxyuser.atguigu.groups</name>

<value>\*</value>

</property>

**mapred-site.xml**

<!-- 配置 MapReduce JobHistory Server 地址 ，默认端口10020 -->

<property>

<name>mapreduce.jobhistory.address</name>

<value>hadoop102:10020</value>

</property>

<!-- 配置 MapReduce JobHistory Server web ui 地址， 默认端口19888 -->

<property>

<name>mapreduce.jobhistory.webapp.address</name>

<value>hadoop102:19888</value>

</property>

**yarn-site.xml**

<!-- 任务历史服务 -->

<property>

<name>yarn.log.server.url</name>

<value>http://hadoop102:19888/jobhistory/logs/</value>

</property>

完成后：记得scp同步到其他机器节点

### 3.1.3 启动Hadoop集群

[atguigu@hadoop102 hadoop-2.5.0-cdh5.3.6]$ sbin/start-dfs.sh

[atguigu@hadoop103 hadoop-2.5.0-cdh5.3.6]$ sbin/start-yarn.sh

[atguigu@hadoop102 hadoop-2.5.0-cdh5.3.6]$ sbin/mr-jobhistory-daemon.sh start historyserver

注意：需要开启JobHistoryServer, 最好执行一个MR任务进行测试。

## 3.2 部署Oozie

### 3.2.1 解压Oozie

[atguigu@hadoop102 software]$ tar -zxvf /opt/software/cdh/oozie-4.0.0-cdh5.3.6.tar.gz -C /opt/module

### 3.2.2 在oozie根目录下解压oozie-hadooplibs-4.0.0-cdh5.3.6.tar.gz

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ tar -zxvf oozie-hadooplibs-4.0.0-cdh5.3.6.tar.gz -C ../

完成后Oozie目录下会出现hadooplibs目录。

### 3.2.3 在Oozie目录下创建libext目录

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ mkdir libext/

### 3.2.4 拷贝依赖的Jar包

1）将hadooplibs里面的jar包，拷贝到libext目录下：

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ cp -ra hadooplibs/hadooplib-2.5.0-cdh5.3.6.oozie-4.0.0-cdh5.3.6/\* libext/

2）拷贝Mysql驱动包到libext目录下：

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ cp -a /opt/software/mysql-connector-java-5.1.27/mysql-connector-java-5.1.27-bin.jar ./libext/

### 3.2.5 将ext-2.2.zip拷贝到libext/目录下

ext是一个js框架，用于展示oozie前端页面：

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ cp -a /opt/software/cdh/ext-2.2.zip libext/

### 3.2.6 修改Oozie配置文件

**oozie-site.xml**

属性：oozie.service.JPAService.jdbc.driver

属性值：com.mysql.jdbc.Driver

解释：JDBC的驱动

属性：oozie.service.JPAService.jdbc.url

属性值：jdbc:mysql://hadoop102:3306/oozie

解释：oozie所需的数据库地址

属性：oozie.service.JPAService.jdbc.username

属性值：root

解释：数据库用户名

属性：oozie.service.JPAService.jdbc.password

属性值：000000

解释：数据库密码

属性：oozie.service.HadoopAccessorService.hadoop.configurations

属性值：\*=/opt/module/CDH/hadoop-2.5.0-cdh5.3.6/etc/hadoop

解释：让Oozie引用Hadoop的配置文件

### 3.2.7 在Mysql中创建Oozie的数据库

进入Mysql并创建oozie数据库：

$ mysql -uroot -p000000

mysql> create database oozie;

### 3.2.8 初始化Oozie

**1) 上传Oozie目录下的yarn.tar.gz文件到HDFS：**

提示：yarn.tar.gz文件会自行解压

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie-setup.sh sharelib create -fs hdfs://hadoop102:8020 -locallib oozie-sharelib-4.0.0-cdh5.3.6-yarn.tar.gz

执行成功之后，去50070检查对应目录有没有文件生成。

**2) 创建oozie.sql文件**

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/ooziedb.sh create -sqlfile oozie.sql -run

**3) 打包项目，生成war包**

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie-setup.sh prepare-war

### 3.2.9 Oozie的启动与关闭

启动命令如下：

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozied.sh start

关闭命令如下：

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozied.sh stop

### 3.2.10 访问Oozie的Web页面

<http://hadoop102:11000/oozie>

# 第4章 Oozie的使用

## 4.1 案例一：Oozie调度shell脚本

目标：使用Oozie调度Shell脚本

分步实现：

1）创建工作目录

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ mkdir -p oozie-apps/shell

2）在oozie-apps/shell目录下创建两个文件——job.properties和workflow.xml文件

[atguigu@hadoop102 shell]$ touch workflow.xml

[atguigu@hadoop102 shell]$ touch job.properties

3）编辑job.properties和workflow.xml文件

**job.properties**

#HDFS地址

nameNode=hdfs://hadoop102:8020

#ResourceManager地址

jobTracker=hadoop103:8032

#队列名称

queueName=default

examplesRoot=oozie-apps

oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/shell

**workflow.xml**

<workflow-app xmlns="uri:oozie:workflow:0.4" name="shell-wf">

<!--开始节点-->

<start to="shell-node"/>

<!--动作节点-->

<action name="shell-node">

<!--shell动作-->

<shell xmlns="uri:oozie:shell-action:0.2">

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<configuration>

<property>

<name>mapred.job.queue.name</name>

<value>${queueName}</value>

</property>

</configuration>

<!--要执行的脚本-->

<exec>mkdir</exec>

<argument>/opt/module/d</argument>

<capture-output/>

</shell>

<ok to="end"/>

<error to="fail"/>

</action>

<!--kill节点-->

<kill name="fail">

<message>Shell action failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>

</kill>

<!--结束节点-->

<end name="end"/>

</workflow-app>

4）上传任务配置

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ /opt/module/cdh/hadoop-2.5.0-cdh5.3.6/bin/hadoop fs -put oozie-apps/ /user/atguigu

5）执行任务

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie job -oozie http://hadoop102:11000/oozie -config oozie-apps/shell/job.properties -run

6）杀死某个任务

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie job -oozie http://hadoop102:11000/oozie -kill 0000004-170425105153692-oozie-z-W

## 4.2 案例二：Oozie逻辑调度执行多个Job

目标：使用Oozie执行多个Job调度

分步执行：

1）编辑job.properties和workflow.xml文件

**job.properties**

nameNode=hdfs://hadoop102:8020

jobTracker=hadoop103:8032

queueName=default

examplesRoot=oozie-apps

oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/shells

**workflow.xml**

<workflow-app xmlns="uri:oozie:workflow:0.4" name="shell-wf">

<start to="p1-shell-node"/>

<action name="p1-shell-node">

<shell xmlns="uri:oozie:shell-action:0.2">

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<configuration>

<property>

<name>mapred.job.queue.name</name>

<value>${queueName}</value>

</property>

</configuration>

<exec>mkdir</exec>

<argument>/opt/module/d1</argument>

<capture-output/>

</shell>

<ok to="p2-shell-node"/>

<error to="fail"/>

</action>

<action name="p2-shell-node">

<shell xmlns="uri:oozie:shell-action:0.2">

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<configuration>

<property>

<name>mapred.job.queue.name</name>

<value>${queueName}</value>

</property>

</configuration>

<exec>mkdir</exec>

<argument>/opt/module/d2</argument>

<capture-output/>

</shell>

<ok to="end"/>

<error to="fail"/>

</action>

<kill name="fail">

<message>Shell action failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>

</kill>

<end name="end"/>

</workflow-app>

2）上传任务配置

$ bin/hadoop fs -rmr /user/atguigu/oozie-apps/

$ bin/hadoop fs -put oozie-apps/shells /user/atguigu/oozie-apps

1. 执行任务

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie job -oozie http://hadoop102:11000/oozie -config oozie-apps/shells/job.properties -run

## 4.3 案例三：Oozie调度MapReduce任务

目标：使用Oozie调度MapReduce任务

分步执行：

1）找到一个可以运行的mapreduce任务的jar包（可以用官方的，也可以是自己写的）

2）拷贝官方模板到oozie-apps

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ cp -r /opt/module/cdh/ oozie-4.0.0-cdh5.3.6/examples/apps/map-reduce/ oozie-apps/

1. **测试一下wordcount在yarn中的运行**

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ /opt/module/cdh/hadoop-2.5.0-cdh5.3.6/bin/yarn jar /opt/module/cdh/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.5.0-cdh5.3.6.jar wordcount /input/ /output/

**4) 配置map-reduce任务的job.properties以及workflow.xml**

**job.properties**

nameNode=hdfs://hadoop102:8020

jobTracker=hadoop103:8032

queueName=default

examplesRoot=oozie-apps

#hdfs://hadoop102:8020/user/admin/oozie-apps/map-reduce/workflow.xml

oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/map-reduce/workflow.xml

outputDir=map-reduce

**workflow.xml**

<workflow-app xmlns="uri:oozie:workflow:0.2" name="map-reduce-wf">

<start to="mr-node"/>

<action name="mr-node">

<map-reduce>

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<prepare>

<delete path="${nameNode}/output/"/>

</prepare>

<configuration>

<property>

<name>mapred.job.queue.name</name>

<value>${queueName}</value>

</property>

<!-- 配置调度MR任务时，使用新的API -->

<property>

<name>mapred.mapper.new-api</name>

<value>true</value>

</property>

<property>

<name>mapred.reducer.new-api</name>

<value>true</value>

</property>

<!-- 指定Job Key输出类型 -->

<property>

<name>mapreduce.job.output.key.class</name>

<value>org.apache.hadoop.io.Text</value>

</property>

<!-- 指定Job Value输出类型 -->

<property>

<name>mapreduce.job.output.value.class</name>

<value>org.apache.hadoop.io.IntWritable</value>

</property>

<!-- 指定输入路径 -->

<property>

<name>mapred.input.dir</name>

<value>/input/</value>

</property>

<!-- 指定输出路径 -->

<property>

<name>mapred.output.dir</name>

<value>/output/</value>

</property>

<!-- 指定Map类 -->

<property>

<name>mapreduce.job.map.class</name>

<value>org.apache.hadoop.examples.WordCount$TokenizerMapper</value>

</property>

<!-- 指定Reduce类 -->

<property>

<name>mapreduce.job.reduce.class</name>

<value>org.apache.hadoop.examples.WordCount$IntSumReducer</value>

</property>

<property>

<name>mapred.map.tasks</name>

<value>1</value>

</property>

</configuration>

</map-reduce>

<ok to="end"/>

<error to="fail"/>

</action>

<kill name="fail">

<message>Map/Reduce failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>

</kill>

<end name="end"/>

</workflow-app>

5）拷贝待执行的jar包到map-reduce的lib目录下

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ cp -a /opt /module/cdh/hadoop-2.5.0-cdh5.3.6/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.5.0-cdh5.3.6.jar oozie-apps/map-reduce/lib

6）上传配置好的app文件夹到HDFS

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ /opt/module/cdh/hadoop-2.5.0-cdh5.3.6/bin/hdfs dfs -put oozie-apps/map-reduce/ /user/admin/oozie-apps

7）执行任务

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie job -oozie http://hadoop102:11000/oozie -config oozie-apps/map-reduce/job.properties -run

## 4.4 案例四：Oozie定时任务/循环任务

目标：Coordinator周期性调度任务

分步实现：

1. 配置Linux时区以及时间服务器
2. 检查系统当前时区：

# date -R

注意：如果显示的时区不是+0800，删除localtime文件夹后，再关联一个正确时区的链接过去，命令如下：

# rm -rf /etc/localtime

# ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

同步时间：

# ntpdate pool.ntp.org

修改NTP配置文件：

# vi /etc/ntp.conf

去掉下面这行前面的# ,并把网段修改成自己的网段：

restrict 192.168.122.0 mask 255.255.255.0 nomodify notrap

注释掉以下几行：

#server 0.centos.pool.ntp.org

#server 1.centos.pool.ntp.org

#server 2.centos.pool.ntp.org

把下面两行前面的#号去掉,如果没有这两行内容,需要手动添加

server 127.127.1.0 # local clock

fudge 127.127.1.0 stratum 10

重启NTP服务：

# systemctl start ntpd.service，

注意，如果是centOS7以下的版本，使用命令：service ntpd start

# systemctl enable ntpd.service，

注意，如果是centOS7以下的版本，使用命令：chkconfig ntpd on

集群其他节点去同步这台时间服务器时间：

首先需要关闭这两台计算机的ntp服务

# systemctl stop ntpd.service，

centOS7以下，则：service ntpd stop

# systemctl disable ntpd.service，

centOS7以下，则：chkconfig ntpd off

# systemctl status ntpd，查看ntp服务状态

# pgrep ntpd，查看ntp服务进程id

同步第一台服务器linux01的时间：

# ntpdate hadoop102

使用root用户制定计划任务,周期性同步时间：

# crontab -e

\*/10 \* \* \* \* /usr/sbin/ntpdate hadoop102

重启定时任务：

# systemctl restart crond.service，

centOS7以下使用：service crond restart，

其他台机器的配置同理。

1. 配置oozie-site.xml文件

属性：oozie.processing.timezone

属性值：GMT+0800

解释：修改时区为东八区区时

注：该属性去oozie-default.xml中找到即可

1. 修改js框架中的关于时间设置的代码

$ vi /opt/module/oozie-4.0.0-cdh5.3.6/oozie-server/webapps/oozie/oozie-console.js

修改如下：

function getTimeZone() {

Ext.state.Manager.setProvider(new Ext.state.CookieProvider());

return Ext.state.Manager.get("TimezoneId","GMT);

}

5）重启oozie服务，并重启浏览器（一定要注意清除缓存）

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozied.sh stop

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozied.sh start

6）拷贝官方模板配置定时任务\

$ cp -r examples/apps/cron/ oozie-apps/

7）修改模板job.properties和coordinator.xml以及workflow.xml

**job.properties**

nameNode=hdfs://hadoop102:8020

jobTracker=hadoop103:8032

queueName=default

examplesRoot=oozie-apps

oozie.coord.application.path=${nameNode}/user/${user.name}/${examplesRoot}/cron

#start：必须设置为未来时间，否则任务失败

start=2017-07-29T17:00+0800

end=2017-07-30T17:00+0800

workflowAppUri=${nameNode}/user/${user.name}/${examplesRoot}/cron

EXEC=p1.sh

**coordinator.xml**

<coordinator-app name="cron-coord" frequency="${coord:minutes(5)}" start="${start}" end="${end}" timezone="GMT+0800" xmlns="uri:oozie:coordinator:0.2">

<action>

<workflow>

<app-path>${workflowAppUri}</app-path>

<configuration>

<property>

<name>jobTracker</name>

<value>${jobTracker}</value>

</property>

<property>

<name>nameNode</name>

<value>${nameNode}</value>

</property>

<property>

<name>queueName</name>

<value>${queueName}</value>

</property>

</configuration>

</workflow>

</action>

</coordinator-app>

**workflow.xml**

<workflow-app xmlns="uri:oozie:workflow:0.5" name="one-op-wf">

<start to="shell-node"/>

<action name="shell-node">

<shell xmlns="uri:oozie:shell-action:0.2">

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<configuration>

<property>

<name>mapred.job.queue.name</name>

<value>${queueName}</value>

</property>

</configuration>

<exec>${EXEC}</exec>

<file>/user/atguigu/oozie-apps/cron/${EXEC}#${EXEC}</file>

<capture-output/>

</shell>

<ok to="end"/>

<error to="fail"/>

</action>

<kill name="fail">

<message>Shell action failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>

</kill>

<end name="end"/>

</workflow-app>

8）上传配置

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ /opt/module/cdh/hadoop-2.5.0-cdh5.3.6/bin/hdfs dfs -put oozie-apps/cron/ /user/atguigu/oozie-apps

9）启动任务

[atguigu@hadoop102 oozie-4.0.0-cdh5.3.6]$ bin/oozie job -oozie http://hadoop102:11000/oozie -config oozie-apps/cron/job.properties -run

注意：oozie允许的最小执行任务的频率是5分钟

# 第5章 常见问题总结

1）Mysql权限配置

授权所有主机可以使用root用户操作所有数据库和数据表

mysql> grant all on \*.\* to root@'%' identified by '000000';

mysql> flush privileges;

mysql> exit;

2）workflow.xml配置的时候不要忽略file属性

3）jps查看进程时，注意有没有bootstrap

4）关闭oozie

如果bin/oozied.sh stop无法关闭，则可以使用kill -9 [pid]，之后oozie-server/temp/xxx.pid文件一定要删除。

5）Oozie重新打包时，一定要注意先关闭进程，删除对应文件夹下面的pid文件。（可以参考第4条目）

6）配置文件一定要生效

起始标签和结束标签无对应则不生效，配置文件的属性写错了，那么则执行默认的属性。

7）libext下边的jar存放于某个文件夹中，导致share/lib创建不成功。

8）调度任务时，找不到指定的脚本，可能是oozie-site.xml里面的Hadoop配置文件没有关联上。

9）修改Hadoop配置文件，需要重启集群。一定要记得scp到其他节点。

10）JobHistoryServer必须开启，集群要重启的。

11）Mysql配置如果没有生效的话，默认使用derby数据库。

12）在本地修改完成的job配置，必须重新上传到HDFS。

13）将HDFS中上传的oozie配置文件下载下来查看是否有错误。

14）Linux用户名和Hadoop的用户名不一致。

# Azkaban

# 第1章 概述

## 1.1 为什么需要工作流调度系统

1）一个完整的数据分析系统通常都是由大量任务单元组成：

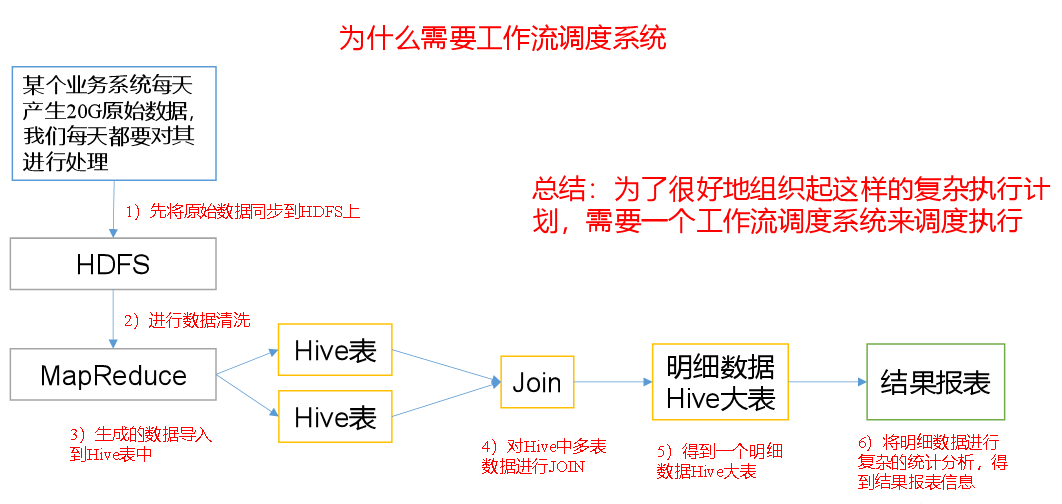
shell脚本程序，java程序，mapreduce程序、hive脚本等

2）各任务单元之间存在时间先后及前后依赖关系

3）为了很好地组织起这样的复杂执行计划，需要一个工作流调度系统来调度执行；

例如，我们可能有这样一个需求，某个业务系统每天产生20G原始数据，我们每天都要对其进行处理，处理步骤如下所示：

* 1. 通过Hadoop先将原始数据上传到HDFS上（HDFS的操作）；
  2. 使用MapReduce对原始数据进行清洗（MapReduce的操作）；
  3. 将清洗后的数据导入到hive表中（hive的导入操作）；
  4. 对Hive中多个表的数据进行JOIN处理，得到一张hive的明细表（创建中间表）；
  5. 通过对明细表的统计和分析，得到结果报表信息（hive的查询操作）；



## 1.2 Azkaban的适用场景

根据以上业务场景： （2）任务依赖（1）任务的结果，（3）任务依赖（2）任务的结果，（4）任务依赖（3）任务的结果，（5）任务依赖（4）任务的结果。一般的做法是，先执行完（1）再执行（2），再一次执行（3）（4）（5）。

这样的话，整个的执行过程都需要人工参加，并且得盯着各任务的进度。但是我们的很多任务都是在深更半夜执行的，通过写脚本设置crontab执行。其实，整个过程类似于一个有向无环图（DAG）。每个子任务相当于大任务中的一个节点，也就是，我们需要的就是一个工作流的调度器，而Azkaban就是能解决上述问题的一个调度器。

## 1.3 什么是azkaban

Azkaban是由Linkedin公司推出的一个批量工作流任务调度器，主要用于在一个工作流内以一个特定的顺序运行一组工作和流程，它的配置是通过简单的key:value对的方式，通过配置中的dependencies 来设置依赖关系。Azkaban使用job配置文件建立任务之间的依赖关系，并提供一个易于使用的web用户界面维护和跟踪你的工作流。

## 1.4 Azkaban特点

1. 兼容任何版本的hadoop
2. 易于使用的Web用户界面
3. 简单的工作流的上传
4. 方便设置任务之间的关系
5. 调度工作流
6. 模块化和可插拔的插件机制
7. 认证/授权(权限的工作)
8. 能够杀死并重新启动工作流
9. 有关失败和成功的电子邮件提醒

## 1.5 常见工作流调度系统

1）简单的任务调度：直接使用crontab实现；

2）复杂的任务调度：开发调度平台或使用现成的开源调度系统，比如ooize、azkaban等

## 1.6 Azkaban的架构

Azkaban由三个关键组件构成：



1. AzkabanWebServer：AzkabanWebServer是整个Azkaban工作流系统的主要管理者，它用户登录认证、负责project管理、定时执行工作流、跟踪工作流执行进度等一系列任务。
2. AzkabanExecutorServer：负责具体的工作流的提交、执行，它们通过mysql数据库来协调任务的执行。
3. 关系型数据库（MySQL）：存储大部分执行流状态，AzkabanWebServer和AzkabanExecutorServer都需要访问数据库。

# 第2章 Azkaban安装部署

## 2.1 安装前准备

1. 将Azkaban Web服务器、Azkaban执行服务器、Azkaban的sql执行脚本及MySQL安装包拷贝到hadoop102虚拟机/opt/software目录下
   1. azkaban-web-server-2.5.0.tar.gz
   2. azkaban-executor-server-2.5.0.tar.gz
   3. azkaban-sql-script-2.5.0.tar.gz
   4. mysql-libs.zip
2. 选择Mysql作为Azkaban数据库，因为Azkaban建立了一些Mysql连接增强功能，以方便Azkaban设置。并增强服务可靠性。（参见hive文档2.4）

## 2.2 安装Azkaban

1. 在/opt/module/目录下创建azkaban目录

[atguigu@hadoop102 module]$ mkdir azkaban

1. 解压azkaban-web-server-2.5.0.tar.gz、azkaban-executor-server-2.5.0.tar.gz、azkaban-sql-script-2.5.0.tar.gz到/opt/module/azkaban目录下

[atguigu@hadoop102 software]$ tar -zxvf azkaban-web-server-2.5.0.tar.gz -C /opt/module/azkaban/

[atguigu@hadoop102 software]$ tar -zxvf azkaban-executor-server-2.5.0.tar.gz -C /opt/module/azkaban/

[atguigu@hadoop102 software]$ tar -zxvf azkaban-sql-script-2.5.0.tar.gz -C /opt/module/azkaban/

1. 对解压后的文件重新命名

[atguigu@hadoop102 azkaban]$ mv azkaban-web-2.5.0/ server

[atguigu@hadoop102 azkaban]$ mv azkaban-executor-2.5.0/ executor

1. azkaban脚本导入，进入mysql，创建azkaban数据库，并将解压的脚本导入到azkaban数据库。

[atguigu@hadoop102 azkaban]$ mysql -uroot -p000000

mysql> create database azkaban;

mysql> use azkaban;

mysql> source /opt/module/azkaban/azkaban-2.5.0/create-all-sql-2.5.0.sql

注：source后跟.sql文件，用于批量处理.sql文件中的sql语句。

## 2.3 生成密钥对和证书

Keytool是java数据证书的管理工具，使用户能够管理自己的公/私钥对及相关证书。

-keystore    指定密钥库的名称及位置(产生的各类信息将存在.keystore文件中)

-genkey(或者-genkeypair)      生成密钥对

-alias 为生成的密钥对指定别名，如果没有默认是mykey

-keyalg  指定密钥的算法 RSA/DSA 默认是DSA

1）生成 keystore的密码及相应信息的密钥库

[atguigu@hadoop102 azkaban]$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA

输入密钥库口令:

再次输入新口令:

您的名字与姓氏是什么?

[Unknown]:

您的组织单位名称是什么?

[Unknown]:

您的组织名称是什么?

[Unknown]:

您所在的城市或区域名称是什么?

[Unknown]:

您所在的省/市/自治区名称是什么?

[Unknown]:

该单位的双字母国家/地区代码是什么?

[Unknown]:

CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown是否正确?

[否]: y

输入 <jetty> 的密钥口令

(如果和密钥库口令相同, 按回车):

再次输入新口令:

注意：

密钥库的密码至少必须6个字符，可以是纯数字或者字母或者数字和字母的组合等等

密钥库的密码最好和<jetty> 的密钥相同，方便记忆

2）将keystore 拷贝到 azkaban web服务器根目录中

[atguigu@hadoop102 azkaban]$ mv keystore /opt/module/azkaban/server/

## 2.4 时间同步配置

先配置好服务器节点上的时区

1. 如果在/usr/share/zoneinfo/这个目录下不存在时区配置文件Asia/Shanghai，就要用 tzselect 生成。

[atguigu@hadoop102 azkaban]$ tzselect

Please identify a location so that time zone rules can be set correctly.

Please select a continent or ocean.

1) Africa

2) Americas

3) Antarctica

4) Arctic Ocean

5) Asia

6) Atlantic Ocean

7) Australia

8) Europe

9) Indian Ocean

10) Pacific Ocean

11) none - I want to specify the time zone using the Posix TZ format.

#? 5

Please select a country.

1) Afghanistan 18) Israel 35) Palestine

2) Armenia 19) Japan 36) Philippines

3) Azerbaijan 20) Jordan 37) Qatar

4) Bahrain 21) Kazakhstan 38) Russia

5) Bangladesh 22) Korea (North) 39) Saudi Arabia

6) Bhutan 23) Korea (South) 40) Singapore

7) Brunei 24) Kuwait 41) Sri Lanka

8) Cambodia 25) Kyrgyzstan 42) Syria

9) China 26) Laos 43) Taiwan

10) Cyprus 27) Lebanon 44) Tajikistan

11) East Timor 28) Macau 45) Thailand

12) Georgia 29) Malaysia 46) Turkmenistan

13) Hong Kong 30) Mongolia 47) United Arab Emirates

14) India 31) Myanmar (Burma) 48) Uzbekistan

15) Indonesia 32) Nepal 49) Vietnam

16) Iran 33) Oman 50) Yemen

17) Iraq 34) Pakistan

#? 9

Please select one of the following time zone regions.

1) Beijing Time

2) Xinjiang Time

#? 1

The following information has been given:

China

Beijing Time

Therefore TZ='Asia/Shanghai' will be used.

Local time is now: Thu Oct 18 16:24:23 CST 2018.

Universal Time is now: Thu Oct 18 08:24:23 UTC 2018.

Is the above information OK?

1) Yes

2) No

#? 1

You can make this change permanent for yourself by appending the line

TZ='Asia/Shanghai'; export TZ

to the file '.profile' in your home directory; then log out and log in again.

Here is that TZ value again, this time on standard output so that you

can use the /usr/bin/tzselect command in shell scripts:

Asia/Shanghai

2）拷贝该时区文件，覆盖系统本地时区配置

[atguigu@hadoop102 azkaban]$ cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

3）集群时间同步（同时发给三个窗口）

[atguigu@hadoop102 azkaban]$ sudo date -s '2018-10-18 16:39:30'

## 2.5 配置文件

### 2.5.1 Web服务器配置

1）进入azkaban web服务器安装目录 conf目录，打开azkaban.properties文件

[atguigu@hadoop102 conf]$ pwd

/opt/module/azkaban/server/conf

[atguigu@hadoop102 conf]$ vim azkaban.properties

2）按照如下配置修改azkaban.properties文件。

#Azkaban Personalization Settings

#服务器UI名称,用于服务器上方显示的名字

azkaban.name=Test

#描述

azkaban.label=My Local Azkaban

#UI颜色

azkaban.color=#FF3601

azkaban.default.servlet.path=/index

#默认web server存放web文件的目录

web.resource.dir=/opt/module/azkaban/server/web/

#默认时区,已改为亚洲/上海 默认为美国

default.timezone.id=Asia/Shanghai

#Azkaban UserManager class

user.manager.class=azkaban.user.XmlUserManager

#用户权限管理默认类（绝对路径）

user.manager.xml.file=/opt/module/azkaban/server/conf/azkaban-users.xml

#Loader for projects

#global配置文件所在位置（绝对路径）

executor.global.properties=/opt/module/azkaban/executor/conf/global.properties

azkaban.project.dir=projects

#数据库类型

database.type=mysql

#端口号

mysql.port=3306

#数据库连接IP

mysql.host=hadoop102

#数据库实例名

mysql.database=azkaban

#数据库用户名

mysql.user=root

#数据库密码

mysql.password=000000

#最大连接数

mysql.numconnections=100

# Velocity dev mode

velocity.dev.mode=false

# Azkaban Jetty server properties.

# Jetty服务器属性.

#最大线程数

jetty.maxThreads=25

#Jetty SSL端口

jetty.ssl.port=8443

#Jetty端口

jetty.port=8081

#SSL文件名（绝对路径）

jetty.keystore=/opt/module/azkaban/server/keystore

#SSL文件密码

jetty.password=000000

#Jetty主密码与keystore文件相同

jetty.keypassword=000000

#SSL文件名（绝对路径）

jetty.truststore=/opt/module/azkaban/server/keystore

#SSL文件密码

jetty.trustpassword=000000

# Azkaban Executor settings

executor.port=12321

# mail settings

mail.sender=

mail.host=

job.failure.email=

job.success.email=

lockdown.create.projects=false

cache.directory=cache

 3）web服务器用户配置

在azkaban web服务器安装目录 conf目录，按照如下配置修改azkaban-users.xml 文件，增加管理员用户。

[atguigu@hadoop102 conf]$ vim azkaban-users.xml

<azkaban-users>

<user username="azkaban" password="azkaban" roles="admin" groups="azkaban" />

<user username="metrics" password="metrics" roles="metrics"/>

<user username="admin" password="admin" roles="admin,metrics"/>

<role name="admin" permissions="ADMIN" />

<role name="metrics" permissions="METRICS"/>

</azkaban-users>

### 2.5.2 执行服务器配置

1）进入执行服务器安装目录conf，打开azkaban.properties

[atguigu@hadoop102 conf]$ pwd

/opt/module/azkaban/executor/conf

[atguigu@hadoop102 conf]$ vim azkaban.properties

1. 按照如下配置修改azkaban.properties文件。

#Azkaban

#时区

default.timezone.id=Asia/Shanghai

# Azkaban JobTypes Plugins

#jobtype 插件所在位置

azkaban.jobtype.plugin.dir=plugins/jobtypes

#Loader for projects

executor.global.properties=/opt/module/azkaban/executor/conf/global.properties

azkaban.project.dir=projects

database.type=mysql

mysql.port=3306

mysql.host=hadoop102

mysql.database=azkaban

mysql.user=root

mysql.password=000000

mysql.numconnections=100

# Azkaban Executor settings

#最大线程数

executor.maxThreads=50

#端口号(如修改,请与web服务中一致)

executor.port=12321

#线程数

executor.flow.threads=30

## 2.6 启动executor服务器

在executor服务器目录下执行启动命令

[atguigu@hadoop102 executor]$ pwd

/opt/module/azkaban/executor

[atguigu@hadoop102 executor]$ bin/azkaban-executor-start.sh

## 2.7 启动web服务器

在azkaban web服务器目录下执行启动命令

[atguigu@hadoop102 server]$ pwd

/opt/module/azkaban/server

[atguigu@hadoop102 server]$ bin/azkaban-web-start.sh

注意：

先执行executor，再执行web，避免Web Server会因为找不到执行器启动失败。

jps查看进程

[atguigu@hadoop102 server]$ jps

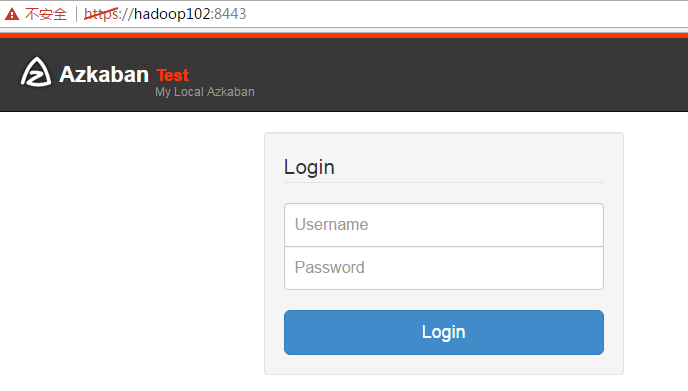
3601 AzkabanExecutorServer

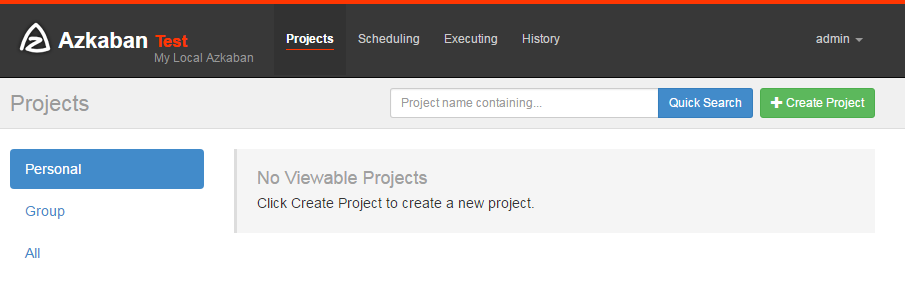
5880 Jps

3661 AzkabanWebServer

启动完成后，在浏览器(建议使用谷歌浏览器)中输入https://服务器IP地址:8443，即可访问azkaban服务了。

在登录中输入刚才在azkaban-users.xml文件中新添加的户用名及密码，点击 login。





# 第3章 Azkaban实战

Azkaba内置的任务类型支持command、java

## 3.1单一job案例

1）创建job描述文件

[atguigu@hadoop102 jobs]$ vim first.job

#first.job

type=command

command=echo 'this is my first job'

2) 将job资源文件打包成zip文件

[atguigu@hadoop102 jobs]$ zip first.zip first.job

adding: first.job (deflated 15%)

[atguigu@hadoop102 jobs]$ ll

总用量 8

-rw-rw-r--. 1 atguigu atguigu 60 10月 18 17:42 first.job

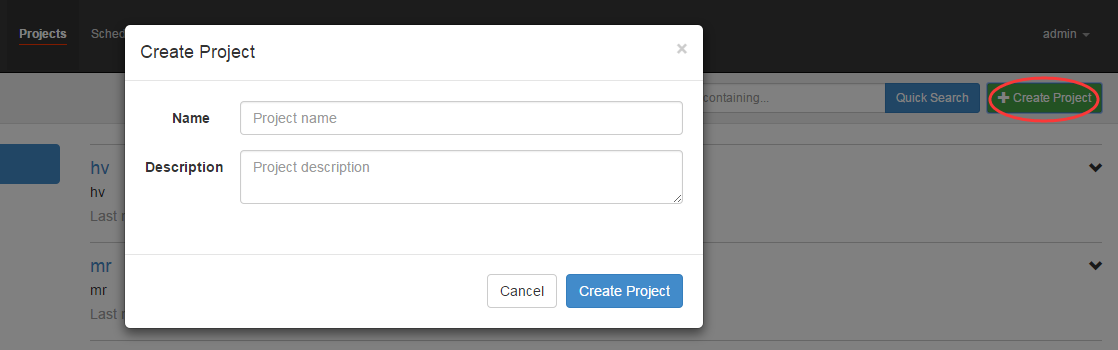
-rw-rw-r--. 1 atguigu atguigu 219 10月 18 17:43 first.zip

注意：

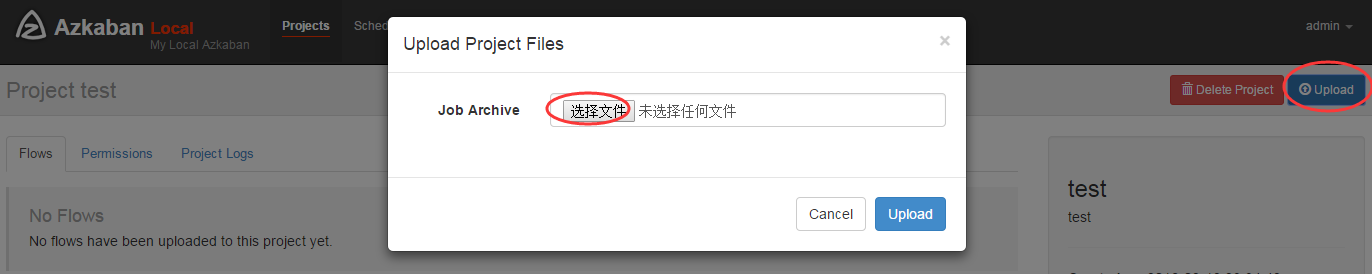
目前，Azkaban上传的工作流文件只支持xxx.zip文件。zip应包含xxx.job运行作业所需的文件和任何文件（文件名后缀必须以.job结尾，否则无法识别）。作业名称在项目中必须是唯一的。

3）通过azkaban的web管理平台创建project并上传job的zip包

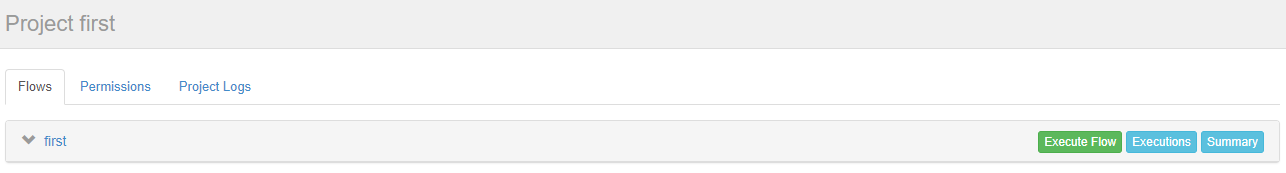
首先创建project



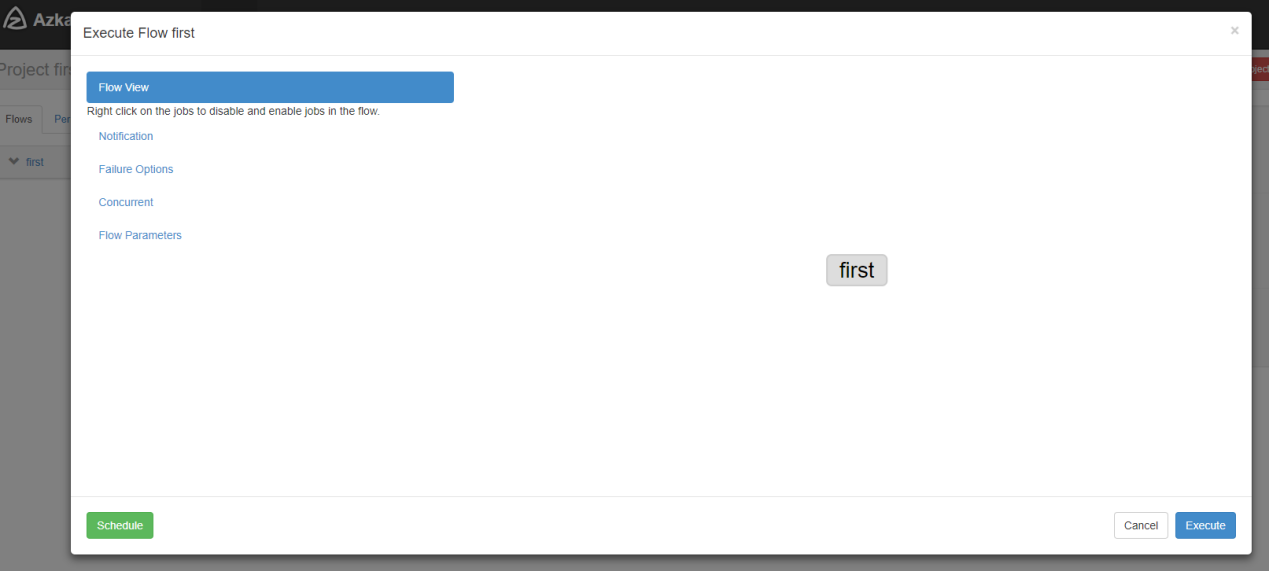
上传zip包



4）启动执行该job



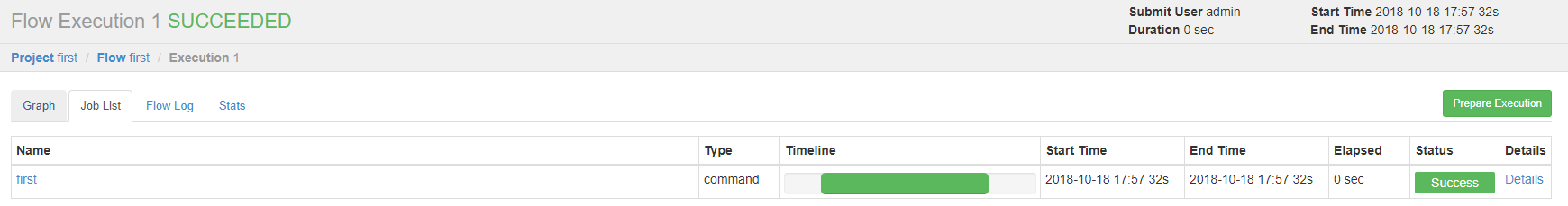
点击执行工作流



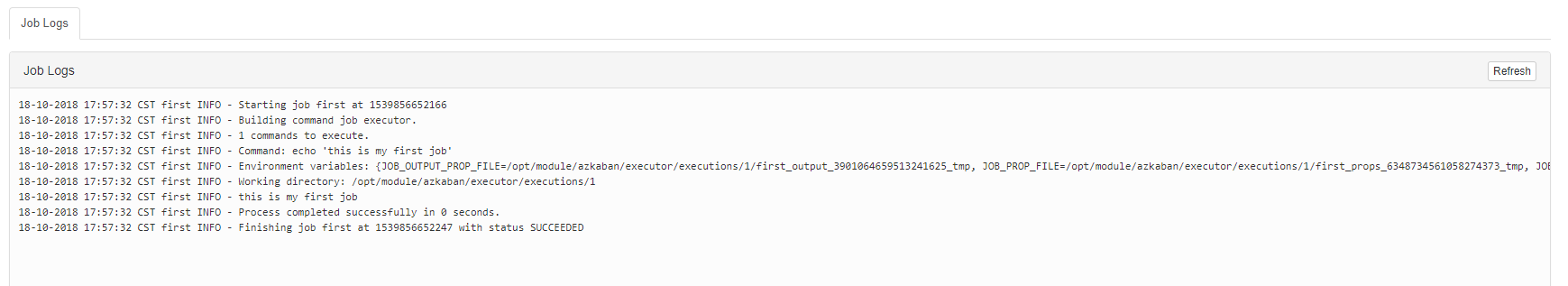
点击继续



5）Job执行成功



6）点击查看job日志



## 3.2多job工作流案例

1）创建有依赖关系的多个job描述

第一个job：start.job

[atguigu@hadoop102 jobs]$ vim start.job

#start.job

type=command

command=touch /opt/module/kangkang.txt

第二个job：step1.job依赖start.job

[atguigu@hadoop102 jobs]$ vim step1.job

#step1.job

type=command

dependencies=start

command=echo "this is step1 job"

第三个job：step2.job依赖start.job

[atguigu@hadoop102 jobs]$ vim step2.job

#step2.job

type=command

dependencies=start

command=echo "this is step2 job"

第四个job：finish.job依赖step1.job和step2.job

[atguigu@hadoop102 jobs]$ vim finish.job

#finish.job

type=command

dependencies=step1,step2

command=echo "this is finish job"

2）将所有job资源文件打到一个zip包中

[atguigu@hadoop102 jobs]$ zip jobs.zip start.job step1.job step2.job finish.job

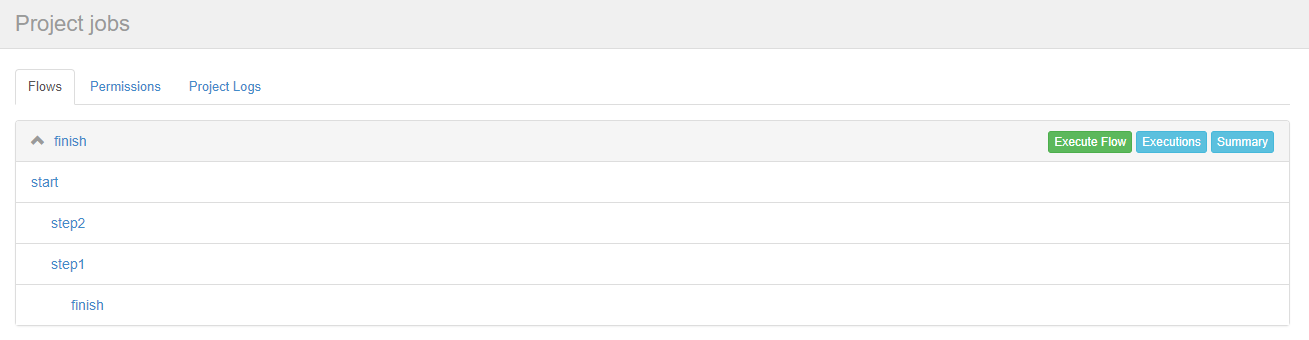
updating: start.job (deflated 16%)

adding: step1.job (deflated 12%)

adding: step2.job (deflated 12%)

adding: finish.job (deflated 14%)

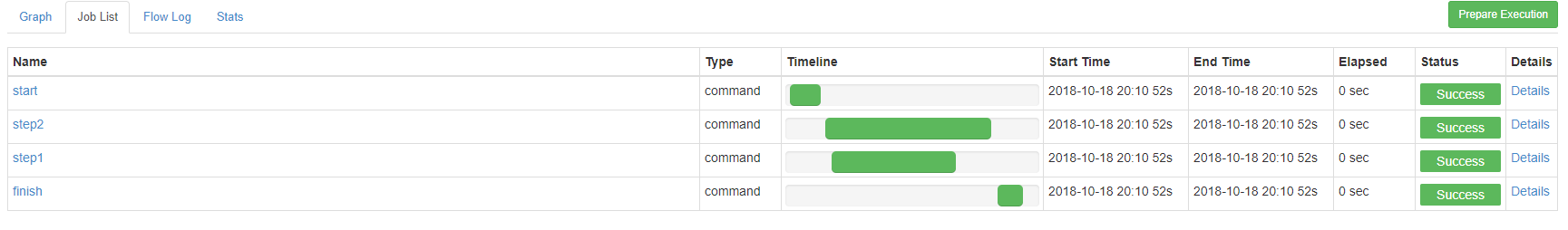
3）在azkaban的web管理界面创建工程并上传zip包



4）启动工作流flow



5）查看结果



思考：

将student.txt文件上传到hdfs，根据所传文件创建外部表，再将表中查询到的结果写入到本地文件

## 3.3 java操作任务

使用Azkaban调度java程序

1）编写java程序

import java.io.IOException;

public class AzkabanTest {

public void run() throws IOException {

// 根据需求编写具体代码

FileOutputStream fos = new FileOutputStream("/opt/module/azkaban/output.txt");

fos.write("this is a java progress".getBytes());

fos.close();

}

public static void main(String[] args) throws IOException {

AzkabanTest azkabanTest = new AzkabanTest();

azkabanTest.run();

}

}

2）将java程序打成jar包，创建lib目录，将jar放入lib内

[atguigu@hadoop102 azkaban]$ mkdir lib

[atguigu@hadoop102 azkaban]$ cd lib/

[atguigu@hadoop102 lib]$ ll

总用量 4

-rw-rw-r--. 1 atguigu atguigu 3355 10月 18 20:55 azkaban-0.0.1-SNAPSHOT.jar

3）编写job文件

[atguigu@hadoop102 jobs]$ vim azkabanJava.job

#azkabanJava.job

type=javaprocess

java.class=com.atguigu.azkaban.AzkabanTest

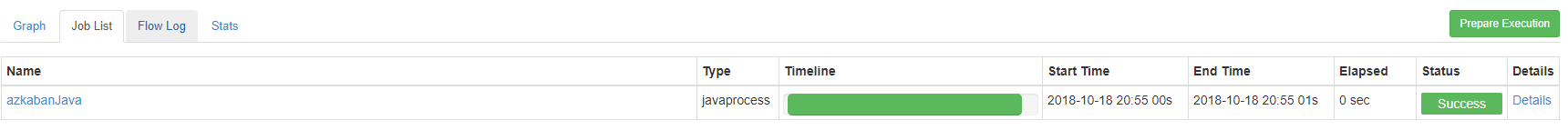
classpath=/opt/module/azkaban/lib/\*

4）将job文件打成zip包

[atguigu@hadoop102 jobs]$ zip azkabanJava.zip azkabanJava.job

adding: azkabanJava.job (deflated 19%)

5）通过azkaban的web管理平台创建project并上传job压缩包，启动执行该job



[atguigu@hadoop102 azkaban]$ pwd

/opt/module/azkaban

[atguigu@hadoop102 azkaban]$ ll

总用量 24

drwxrwxr-x. 2 atguigu atguigu 4096 10月 17 17:14 azkaban-2.5.0

drwxrwxr-x. 10 atguigu atguigu 4096 10月 18 17:17 executor

drwxrwxr-x. 2 atguigu atguigu 4096 10月 18 20:35 jobs

drwxrwxr-x. 2 atguigu atguigu 4096 10月 18 20:54 lib

-rw-rw-r--. 1 atguigu atguigu 23 10月 18 20:55 output

drwxrwxr-x. 9 atguigu atguigu 4096 10月 18 17:17 server

[atguigu@hadoop102 azkaban]$ cat output

this is a java progress

## 3.3 HDFS操作任务

1）创建job描述文件

[atguigu@hadoop102 jobs]$ vim fs.job

#hdfs job

type=command

command=/opt/module/hadoop-2.7.2/bin/hadoop fs -mkdir /azkaban

2）将job资源文件打包成zip文件

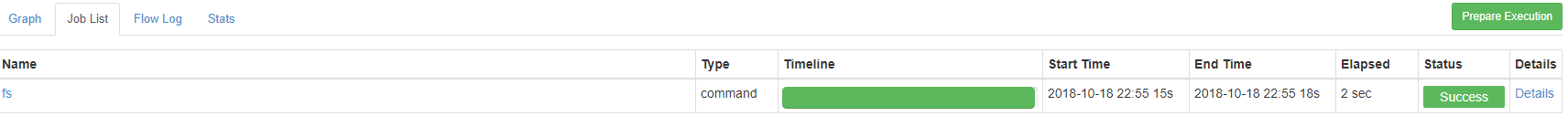
[atguigu@hadoop102 jobs]$ zip fs.zip fs.job

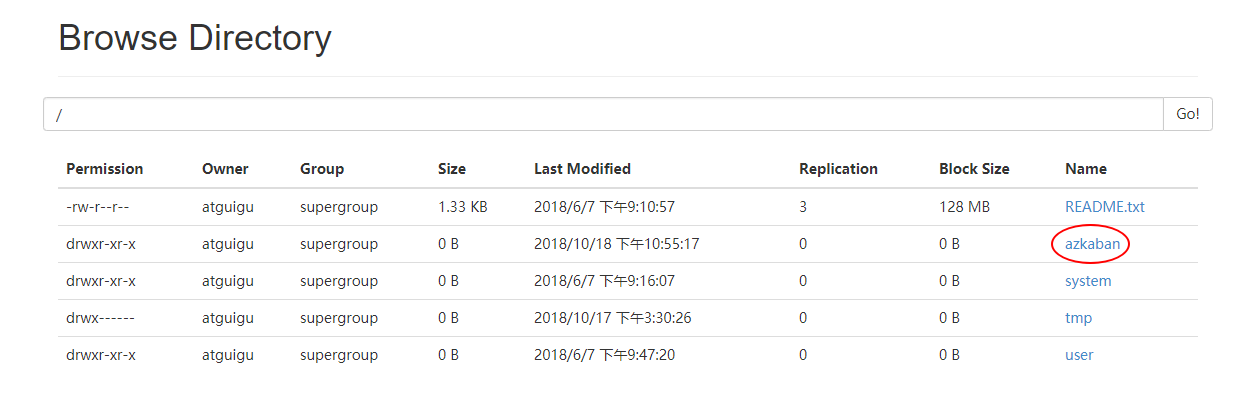
adding: fs.job (deflated 12%)

3）通过azkaban的web管理平台创建project并上传job压缩包

4）启动执行该job

5）查看结果





## 3.4 mapreduce任务

mapreduce任务依然可以使用azkaban进行调度

1. 创建job描述文件，及mr程序jar包

[atguigu@hadoop102 jobs]$ vim mapreduce.job

#mapreduce job

type=command

command=/opt/module/hadoop-2.7.2/bin/hadoop jar /opt/module/hadoop-2.7.2/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /wordcount/input /wordcount/output

1. 将所有job资源文件打到一个zip包中

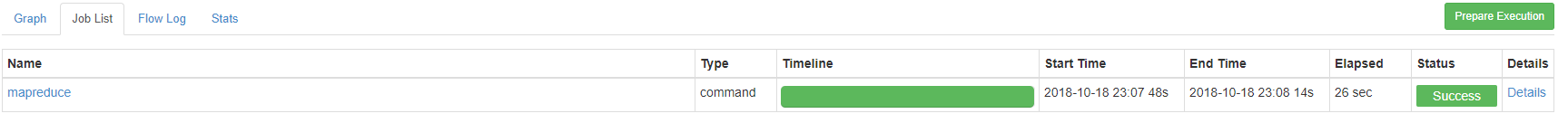
[atguigu@hadoop102 jobs]$ zip mapreduce.zip mapreduce.job

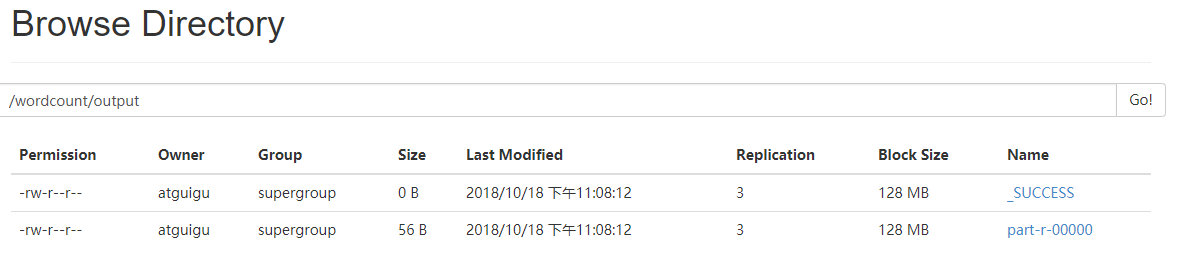
adding: mapreduce.job (deflated 43%)

3）在azkaban的web管理界面创建工程并上传zip包

4）启动job

5）查看结果





## 3.5 Hive脚本任务

1）创建job描述文件和hive脚本

（1）Hive脚本：student.sql

[atguigu@hadoop102 jobs]$ vim student.sql

use default;

drop table student;

create table student(id int, name string)

row format delimited fields terminated by '\t';

load data local inpath '/opt/module/datas/student.txt' into table student;

insert overwrite local directory '/opt/module/datas/student'

row format delimited fields terminated by '\t'

select \* from student;

（2）Job描述文件：hive.job

[atguigu@hadoop102 jobs]$ vim hive.job

#hive job

type=command

command=/opt/module/hive/bin/hive -f /opt/module/azkaban/jobs/student.sql

1. 将所有job资源文件打到一个zip包中

[atguigu@hadoop102 jobs]$ zip hive.zip hive.job

adding: hive.job (deflated 21%)

3）在azkaban的web管理界面创建工程并上传zip包

4）启动job

5）查看结果

[atguigu@hadoop102 student]$ cat /opt/module/datas/student/000000\_0

1001 yangyang

1002 huihui

1003 banzhang

1004 pengpeng

