# hadoop

## hadoop快速入门

### haoop背景介绍

* **什么是HADOOP**

1. HADOOP是apache旗下的一套开源**软件平台**
2. HADOOP提供的功能：利用服务器集群，根据用户的自定义业务逻辑，**对海量数据进行分布式处理**
3. HADOOP的核心组件有
   * 1. HDFS（分布式文件系统）
     2. YARN（运算资源调度系统）
     3. MAPREDUCE（分布式运算编程框架）
4. 广义上来说，HADOOP通常是指一个更广泛的概念——HADOOP生态圈

* **HADOOP产生背景**

1. HADOOP**最早起源于Nutch**。Nutch的设计目标是构建一个大型的全网搜索引擎，包括网页抓取、索引、查询等功能，但随着抓取网页数量的增加，**遇到了严重的可扩展性问题——**如何解决数十亿网页的存储和索引问题。
2. 2003年、2004年**谷歌发表的两篇论文为该问题提供了可行的解决方案**。

——分布式文件系统（GFS），可用于处理海量网页的**存储**

——分布式计算框架MAPREDUCE，可用于处理海量网页的**索引计算**问题。

1. Nutch的开发人员完成了相应的**开源实现HDFS和MAPREDUCE**，并从Nutch中剥离成为独立项目HADOOP，到2008年1月，HADOOP成为Apache顶级项目，迎来了它的快速发展期。

* **HADOOP在大数据、云计算中的位置和关系**

1. 云计算是分布式计算、并行计算、网格计算、多核计算、网络存储、虚拟化、负载均衡等传统计算机技术和互联网技术融合发展的产物。借助IaaS(基础设施即服务)、PaaS(平台即服务)、SaaS（软件即服务）等业务模式，把强大的计算能力提供给终端用户。
2. 现阶段，云计算的**两大底层支撑技术**为“**虚拟化**”和“**大数据技术**”
3. 而HADOOP则是云计算的PaaS层的解决方案之一，并不等同于PaaS，更不等同于云计算本身。

* **国内外HADOOP应用案例介绍**

**1、HADOOP应用于数据服务基础平台建设**



**2、HADOOP用于用户画像**



**3、HADOOP用于网站点击流日志数据挖掘**

****

**金融行业： 个人征信分析**

**证券行业： 投资模型分析**

**交通行业： 车辆、路况监控分析**

**电信行业：用户上网行为分析**

**......**

**总之：hadoop并不会跟某种具体的行业或者某个具体的业务挂钩，它只是一种用来做海量数据分析处理的工具**

* **国内HADOOP的就业情况分析**

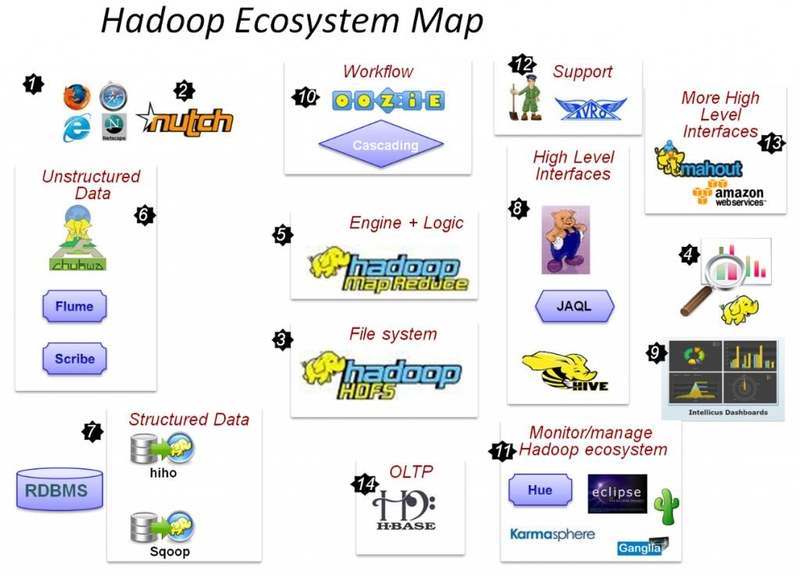
1. HADOOP就业整体情况
2. 大数据产业已纳入**国家十三五规划**
3. 各大城市都在进行**智慧城市项目**建设，而智慧城市的根基就是大数据综合平台
4. 互联网时代数据的种类，增长都呈现**爆发式增长**，各行业对数据的价值日益重视
5. 相对于传统JAVAEE技术领域来说，大数据领域的**人才相对稀缺**
6. 随着现代社会的发展，数据处理和数据挖掘的重要性只会增不会减，因此，大数据技术是一个尚在蓬勃发展且具有**长远前景的领域**
7. HADOOP就业职位要求

大数据是个复合专业，包括应用开发、软件平台、算法、数据挖掘等，因此，**大数据技术领域的就业选择是多样的**，但就HADOOP而言，通常都需要具备以下技能或知识：

1. HADOOP分布式集群的平台搭建
2. HADOOP分布式文件系统HDFS的原理理解及使用
3. HADOOP分布式运算框架MAPREDUCE的原理理解及编程
4. Hive数据仓库工具的熟练应用
5. Flume、sqoop、oozie等辅助工具的熟练使用
6. Shell/python等脚本语言的开发能力
7. HADOOP相关职位的薪资水平

大数据技术或具体到HADOOP的就业需求目前主要集中在北上广深一线城市，**薪资待遇普遍高于传统JAVAEE开发人员**.

* **HADOOP生态圈以及各组成部分的简介**



HADOOP（hdfs、MAPREDUCE、yarn） 元老级大数据处理技术框架，擅长离线数据分析

Zookeeper 分布式协调服务基础组件

Hbase 分布式海量数据库，离线分析和在线业务通吃

Hive sql 数据仓库工具，使用方便，功能丰富，基于MR延迟大

Sqoop数据导入导出工具

Flume数据采集框架

重点组件：

HDFS：分布式文件系统

MAPREDUCE：分布式运算程序开发框架

HIVE：基于大数据技术（文件系统+运算框架）的SQL数据仓库工具

HBASE：基于HADOOP的分布式海量数据库

**ZOOKEEPER：分布式协调服务基础组件**

Mahout：基于mapreduce/spark/flink等分布式运算框架的机器学习算法库

Oozie：工作流调度框架

Sqoop：数据导入导出工具

Flume：日志数据采集框架

### 分布式系统概述

*注：由于大数据技术领域的各类技术框架基本上都是分布式系统，因此，理解hadoop、storm、spark等技术框架，都需要具备基本的分布式系统概念*

#### 分布式软件系统(Distributed Software Systems)

* 该软件系统会划分成多个子系统或模块，各自运行在不同的机器上，子系统或模块之间通过网络通信进行协作，实现最终的整体功能
* 比如分布式操作系统、分布式程序设计语言及其编译(解释)系统、分布式文件系统和分布式数据库系统等。

#### 分布式软件系统举例：solrcloud

1. 一个solrcloud集群通常有多台solr服务器
2. 每一个solr服务器节点负责存储整个索引库的若干个shard（数据分片）
3. 每一个shard又有多台服务器存放若干个副本互为主备用
4. 索引的建立和查询会在整个集群的各个节点上并发执行
5. solrcloud集群作为整体对外服务，而其内部细节可对客户端透明

**总结：利用多个节点共同协作完成一项或多项具体业务功能的系统就是分布式系统。**

#### \*分布式应用系统模拟开发(自己代码写出来)

**hdfs:存储**

**MapReduce:不是移动数据,而是传输程序**

**yarn模拟实现思路,最好自己可以实现出来**

最好自己实现

**需求：**可以实现由主节点将运算任务发往从节点，并将各从节点上的任务启动；

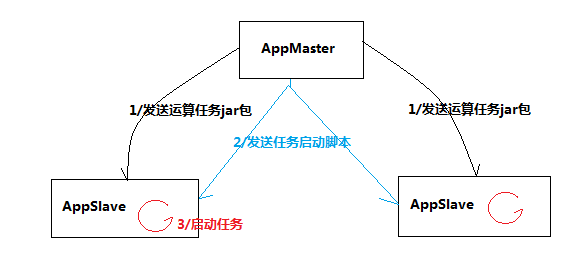
**程序清单：**

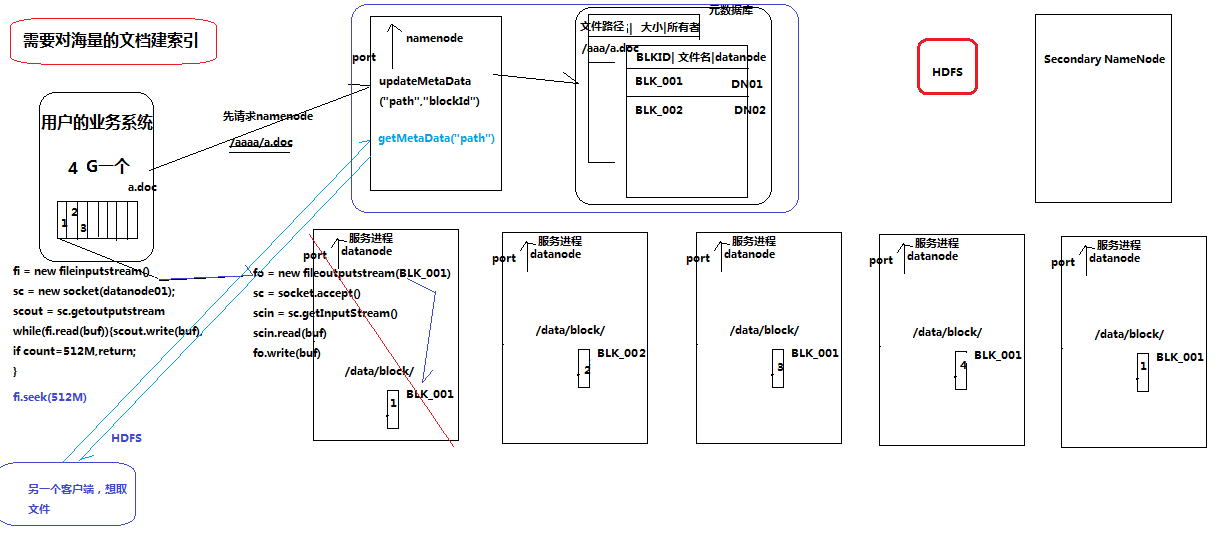
AppMaster

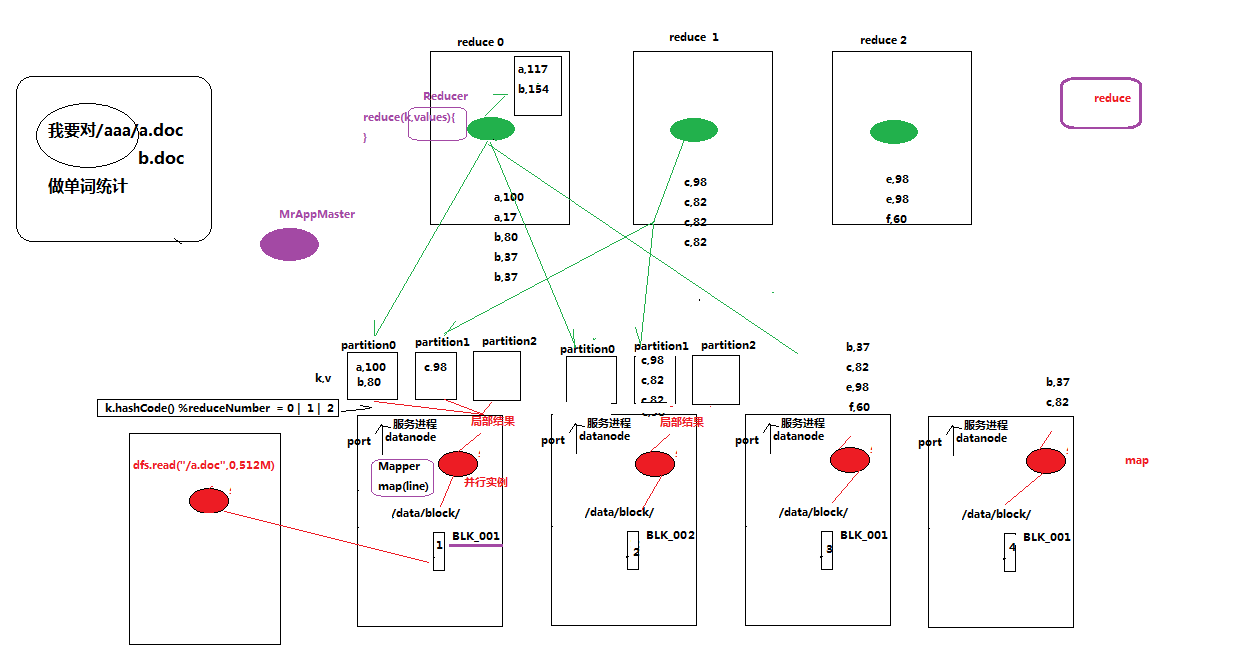
AppSlave/APPSlaveThread

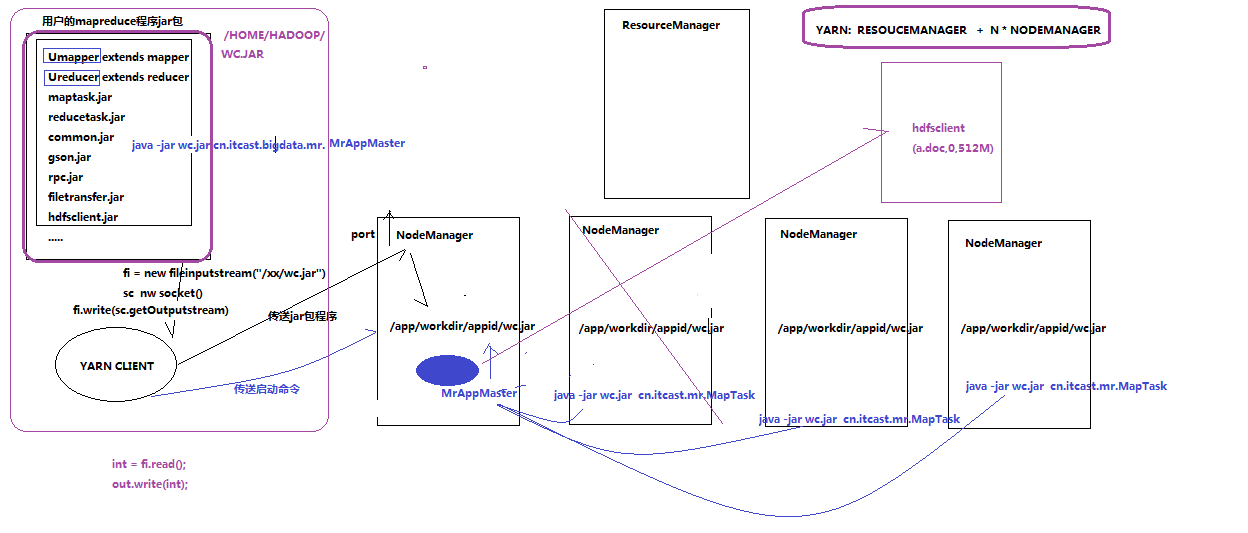
Task

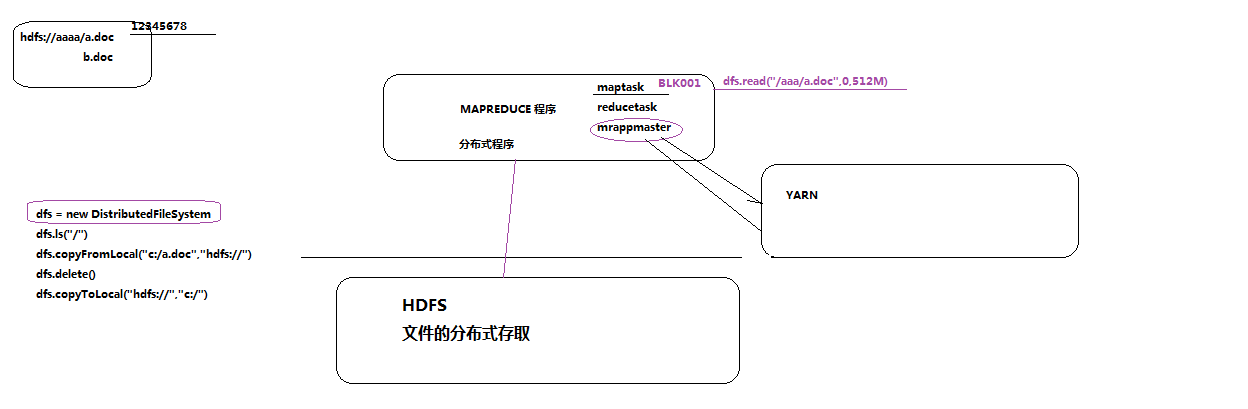
**程序运行逻辑流程：**

****







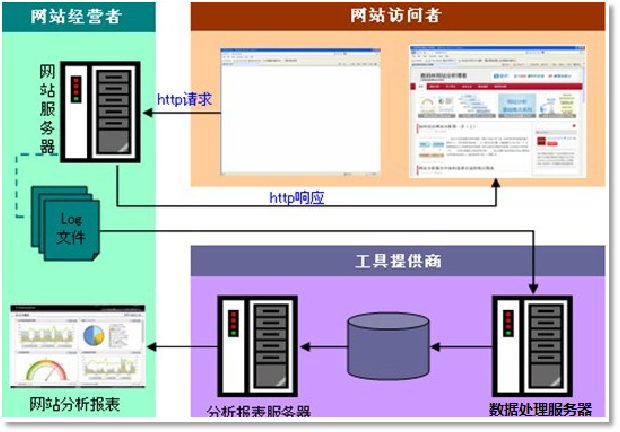


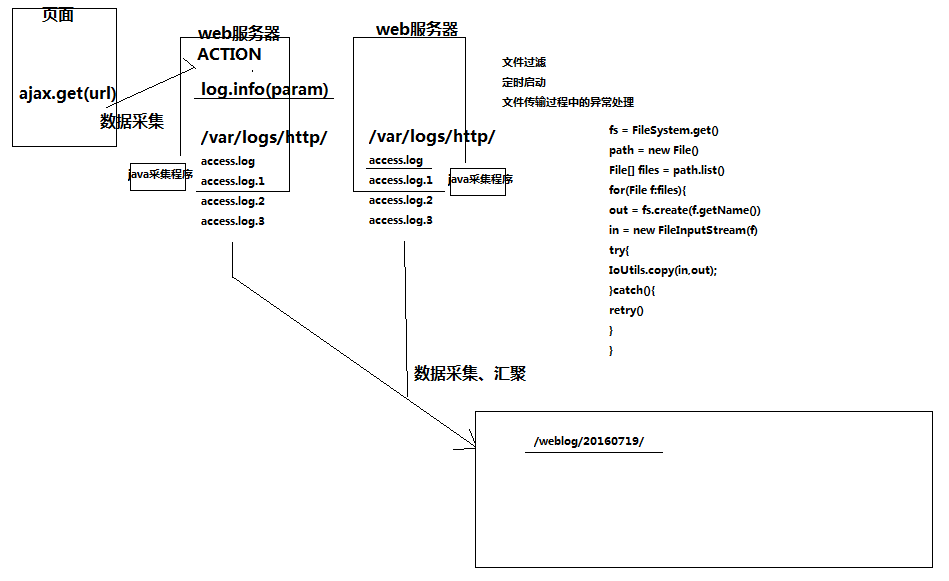
### 离线分析系统结构介绍

*注：本环节主要感受数据分析系统的宏观概念及处理流程，初步理解hadoop等框架在其中的应用环节，不用过于关注代码细节*

线数据分析流程

一个应用广泛的数据分析系统：“web日志数据挖掘”





#### 需求分析

##### 案例名称

网站或app点击流日志数据挖掘系统

一般中型的网站(10W的PV以上)，每天会产生1G以上Web日志文件。大型或超大型的网站，可能每小时就会产生10G的数据量。

具体来说，比如某电子商务网站，在线团购业务。每日PV数100w，独立IP数5w。用户通常在工作日上午10:00-12:00和下午15:00-18:00访问量最大。日间主要是通过PC端浏览器访问，休息日及夜间通过移动设备访问较多。网站搜索浏量占整个网站的80%，PC用户不足1%的用户会消费，移动用户有5%会消费。

对于日志的这种规模的数据，用HADOOP进行日志分析，是最适合不过的了。

##### 案例需求描述

“Web点击流日志”包含着网站运营很重要的信息，通过日志分析，我们可以知道网站的访问量，哪个网页访问人数最多，哪个网页最有价值，广告转化率、访客的来源信息，访客的终端信息等。

##### 数据来源

本案例的数据主要由**用户的点击行为记录**

获取方式：在页面预埋一段js程序，为页面上想要监听的标签绑定事件，只要用户点击或移动到标签，即可触发ajax请求到后台servlet程序，用log4j记录下事件信息，从而在web服务器（nginx、tomcat等）上形成不断增长的日志文件。

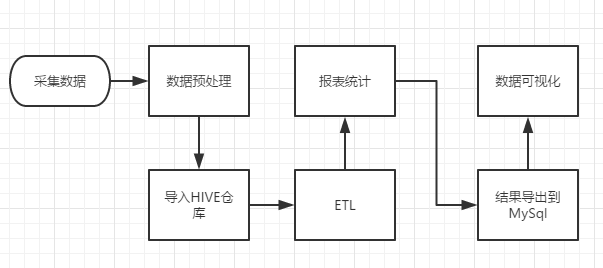
形如：

|  |
| --- |
| 58.215.204.118 - - [18/Sep/2013:06:51:35 +0000] "GET /wp-includes/js/jquery/jquery.js?ver=1.10.2 HTTP/1.1" 304 0 "http://blog.fens.me/nodejs-socketio-chat/" "Mozilla/5.0 (Windows NT 5.1; rv:23.0) Gecko/20100101 Firefox/23.0" |

#### 数据处理流程

##### 流程图解析

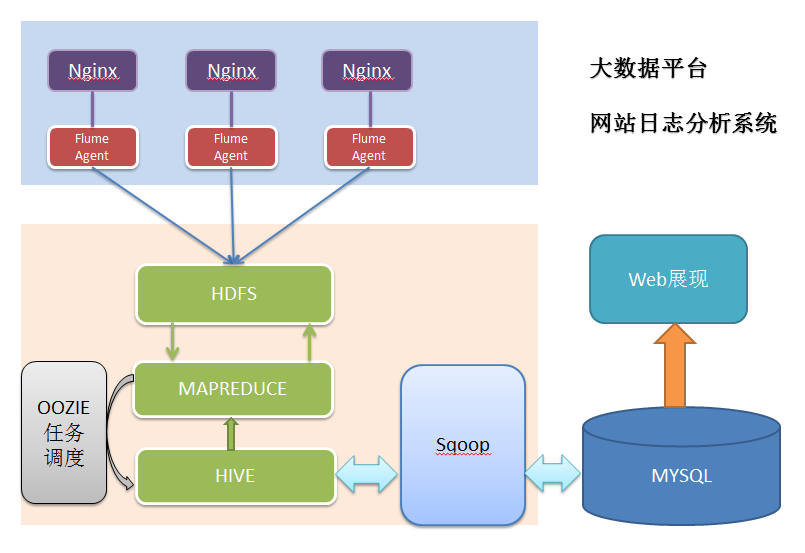
本案例跟典型的BI系统极其类似，整体流程如下：

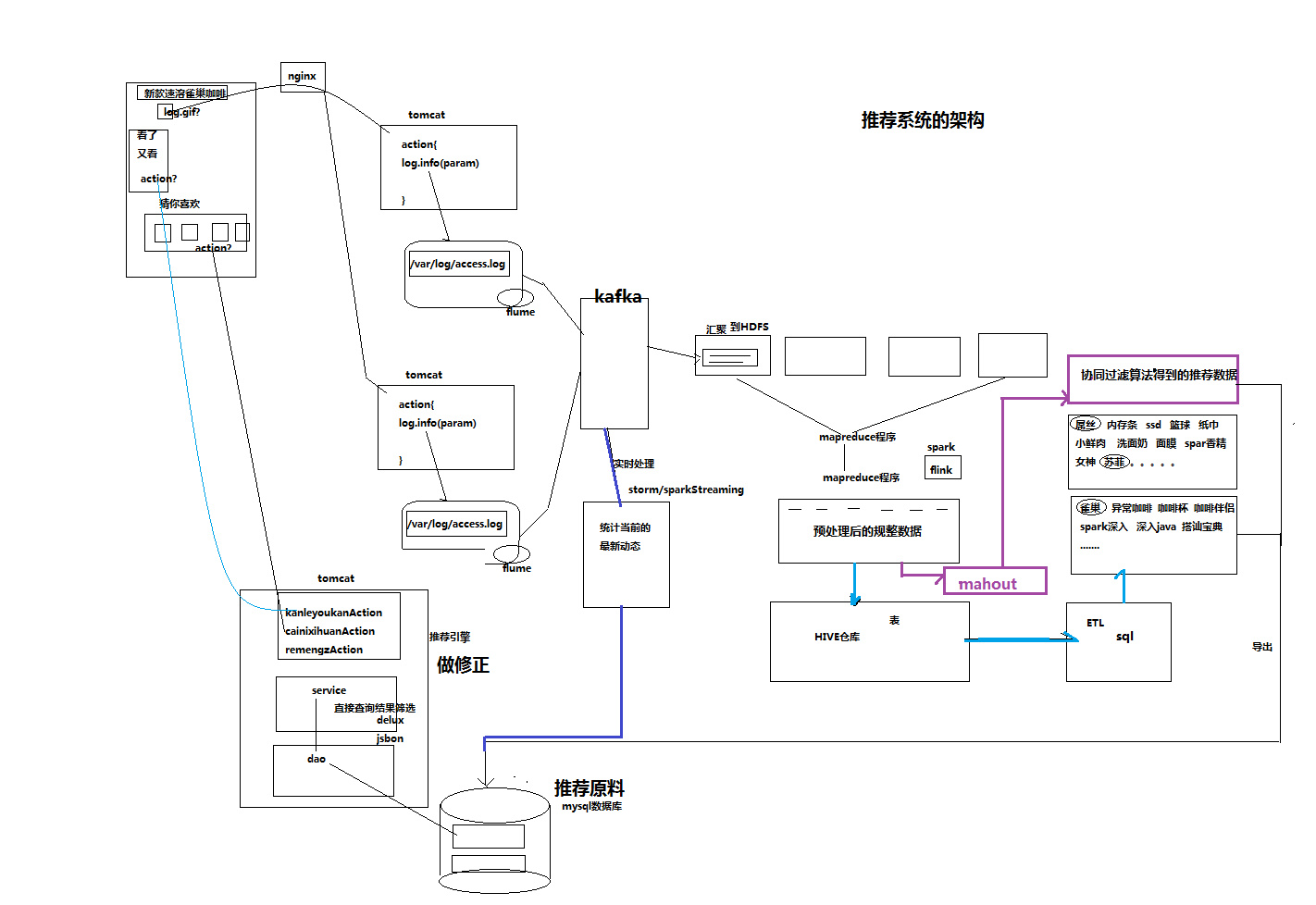
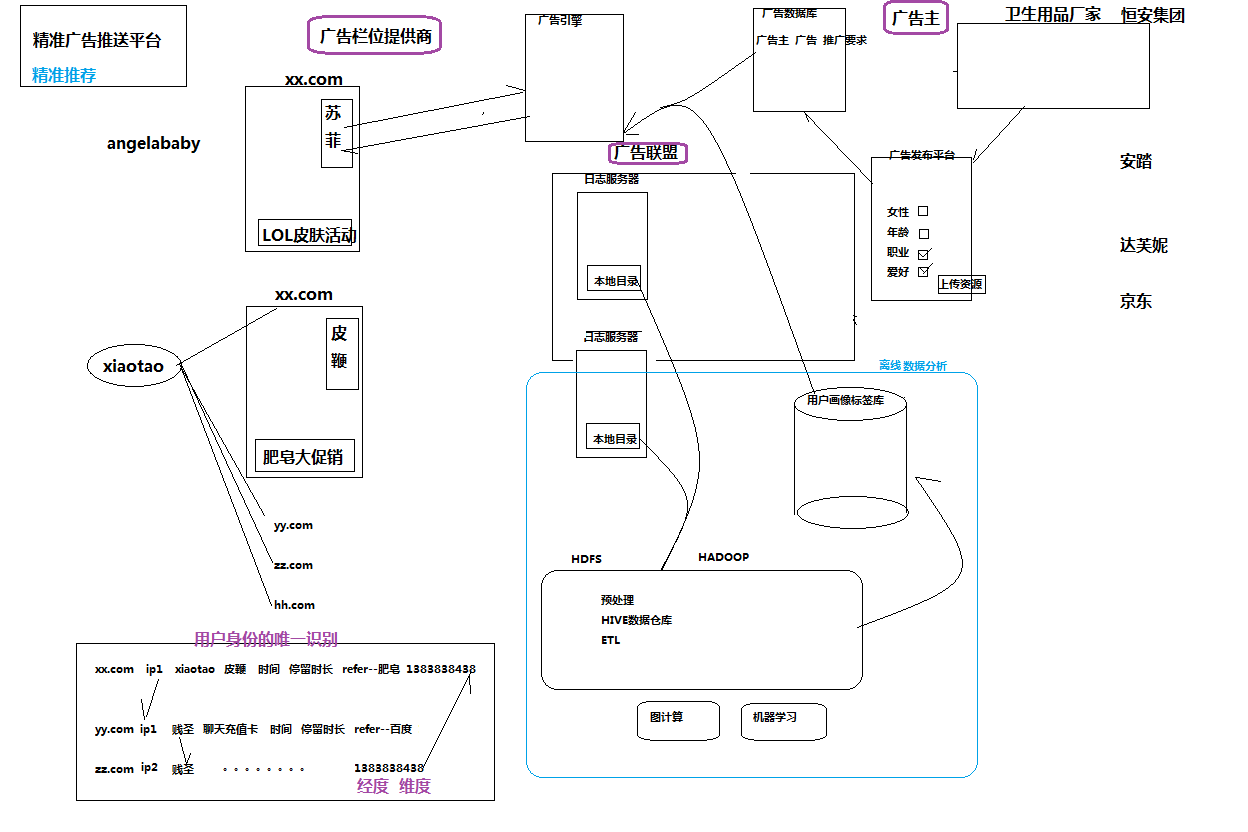


但是，由于本案例的前提是处理海量数据，因而，流程中各环节所使用的技术则跟传统BI完全不同，后续课程都会一一讲解：

1. 数据采集：定制开发采集程序，或使用开源框架FLUME
2. 数据预处理：定制开发mapreduce程序运行于hadoop集群
3. 数据仓库技术：基于hadoop之上的Hive
4. 数据导出：基于hadoop的sqoop数据导入导出工具
5. 数据可视化：定制开发web程序或使用kettle等产品
6. 整个过程的流程调度：hadoop生态圈中的oozie工具或其他类似开源产品

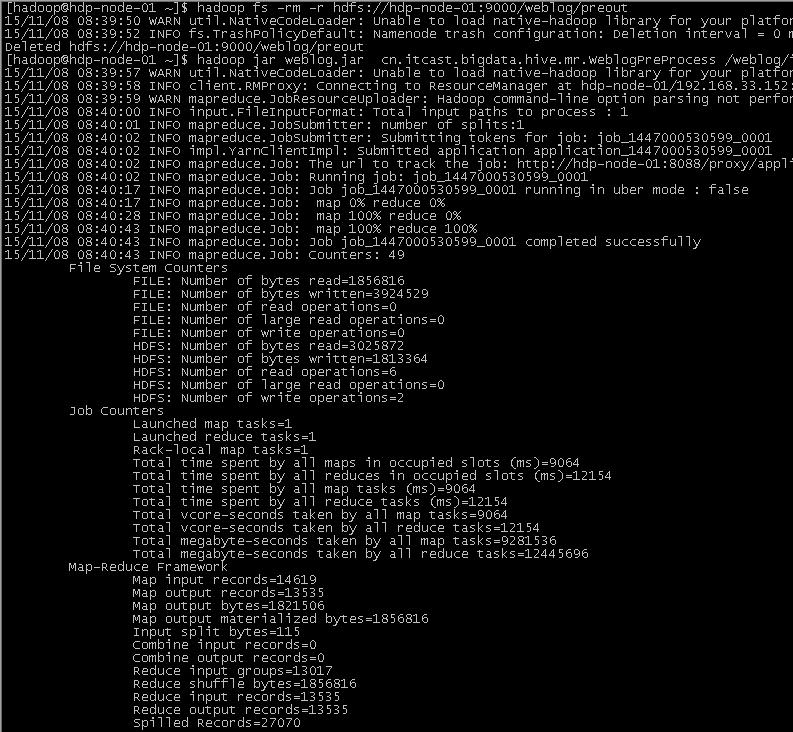
##### 项目技术架构图



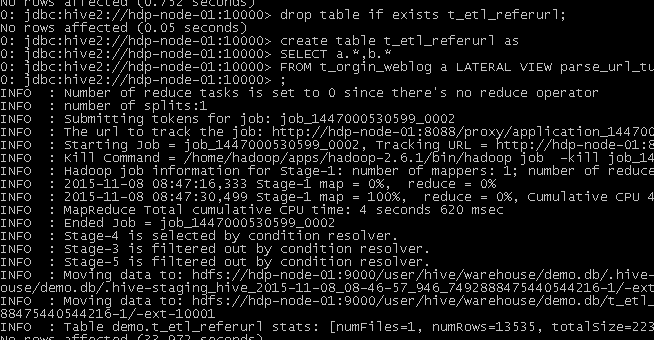
 

##### 项目相关截图（感性认识，欣赏即可）

1. Mapreudce程序运行



1. 在Hive中查询数据



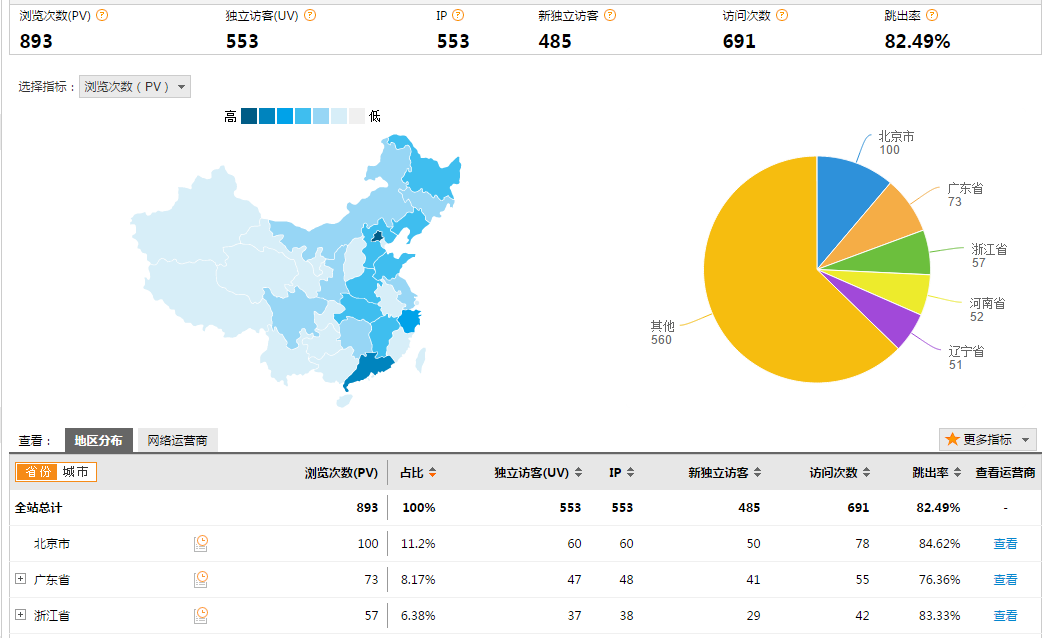
1. 将统计结果导入mysql

|  |
| --- |
| ./sqoop export --connect jdbc:mysql://localhost:3306/weblogdb --username root --password root --table t\_display\_xx --export-dir /user/hive/warehouse/uv/dt=2014-08-03 |

#### 项目最终效果

经过完整的数据处理流程后，会周期性输出各类统计指标的报表，在生产实践中，最终需要将这些报表数据以可视化的形式展现出来，本案例采用web程序来实现数据可视化

效果如下所示：



### \*集群搭建

#### 集群简介

HADOOP集群具体来说包含两个集群：HDFS集群和YARN集群，两者逻辑上分离，但物理上常在一起

HDFS集群：

负责海量数据的存储，集群中的角色主要有 NameNode / DataNode

YARN集群：

负责海量数据运算时的资源调度，集群中的角色主要有 ResourceManager /NodeManager

*(那mapreduce是什么呢？它其实是一个应用程序开发包)*

本集群搭建案例，以5节点为例进行搭建，角色分配如下：

|  |
| --- |
| hdp-node-01 NameNode SecondaryNameNode  hdp-node-02 ResourceManager  hdp-node-03 DataNode NodeManager  hdp-node-04 DataNode NodeManager  hdp-node-05 DataNode NodeManager |

部署图如下：



#### 服务器准备

本案例使用虚拟机服务器来搭建HADOOP集群，所用软件及版本：

* Vmware 11.0
* Centos 6.5 64bit

#### 网络环境准备

* 采用NAT方式联网
* 网关地址：192.168.33.1
* 3个服务器节点IP地址：192.168.33.101、192.168.33.102、192.168.33.103
* 子网掩码：255.255.255.0

#### 服务器系统设置

* 添加HADOOP用户
* 为HADOOP用户分配sudoer权限
* 同步时间
* 设置主机名
  + hdp-node-01
  + hdp-node-02
  + hdp-node-03
* 配置内网域名映射：
  + 192.168.33.101 hdp-node-01
  + 192.168.33.102 hdp-node-02
  + 192.168.33.103 hdp-node-03
* 配置ssh免密登陆
* 配置防火墙

#### Jdk环境安装

* 上传jdk安装包
* 规划安装目录 /home/hadoop/apps/jdk\_1.7.65
* 解压安装包
* 配置环境变量 /etc/profile

#### HADOOP安装部署

补充： HADOOP的版本选择

1、APACHE官方版本

大版本： 0.20.2

1.x

2.x 2.6.4 2.7.3

2、商业发行版（商业版提供完善的管理系统、修bug可能领先于官方）：

Cloudera公司的CDH：5.7.x

* 上传HADOOP安装包
* 规划安装目录 /root/apps/hadoop-2.6.4
* 解压安装包
* 修改配置文件 $HADOOP\_HOME/etc/hadoop/

最简化配置如下：

vi hadoop-env.sh

|  |
| --- |
| # The java implementation to use.  export JAVA\_HOME=/root/apps/jdk1.7.0\_51 |

vi core-site.xml

|  |
| --- |
| <configuration>  <property>  <name>fs.defaultFS</name>  <value>hdfs://hdp-node-01:9000</value>  </property>  <property>  <name>hadoop.tmp.dir</name>  <value>/root/apps/hadoop-2.6.4/tmp</value>  </property>  </configuration> |

vi hdfs-site.xml

|  |
| --- |
| <configuration>  <property>  <name>dfs.namenode.name.dir</name>  <value>/home/hadoop/data/name,/path2/,/path3/,nfs://</value>  </property>  <!---namenode配置多个目录和datanode配置多个目录，有什么区别？---->  <property>  <name>dfs.datanode.data.dir</name>  <value>/home/hadoop/data/data,/path2/</value>  </property>  <property>  <name>dfs.replication</name>  <value>3</value>  </property>  <property>  <name>dfs.secondary.http.address</name>  <value>hdp-node-01:50090</value>  </property>  </configuration> |

vi mapred-site.xml

|  |
| --- |
| <configuration>  <property>  <name>mapreduce.framework.name</name>  <value>yarn</value>  </property>  </configuration> |

vi yarn-site.xml

|  |
| --- |
| <configuration>  <property>  <name>yarn.resourcemanager.hostname</name>  <value>hadoop01</value>  </property>  <property>  <name>yarn.nodemanager.aux-services</name>  <value>mapreduce\_shuffle</value>  </property>  </configuration> |

vi salves

|  |
| --- |
| hdp-node-01  hdp-node-02  hdp-node-03 |

#### 启动集群

初始化HDFS

|  |
| --- |
| bin/hadoop namenode -format |

启动HDFS

|  |
| --- |
| sbin/start-dfs.sh |

启动YARN

|  |
| --- |
| sbin/start-yarn.sh |

#### 测试

**1、上传文件到HDFS**

从本地上传一个文本文件到hdfs的/wordcount/input目录下

|  |
| --- |
| [HADOOP@hdp-node-01 ~]$ HADOOP fs -mkdir -p /wordcount/input  [HADOOP@hdp-node-01 ~]$ HADOOP fs -put /home/HADOOP/somewords.txt /wordcount/input |

**2、运行一个mapreduce程序**

在HADOOP安装目录下，运行一个示例mr程序

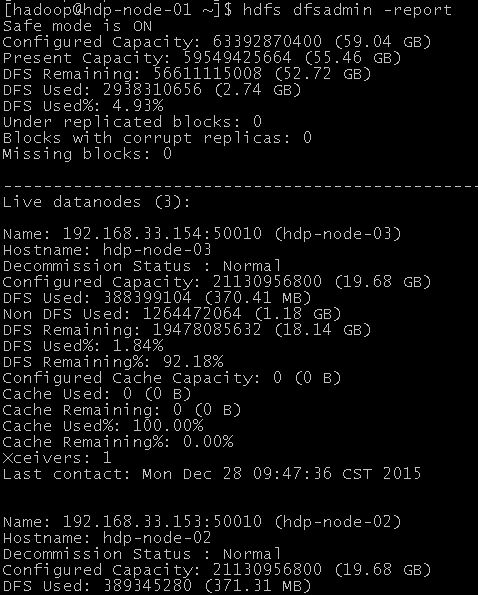
|  |
| --- |
| cd $HADOOP\_HOME/share/hadoop/mapreduce/  hadoop jar mapredcue-example-2.6.4.jar wordcount /wordcount/input /wordcount/output |

### 集群使用初步

#### HDFS使用

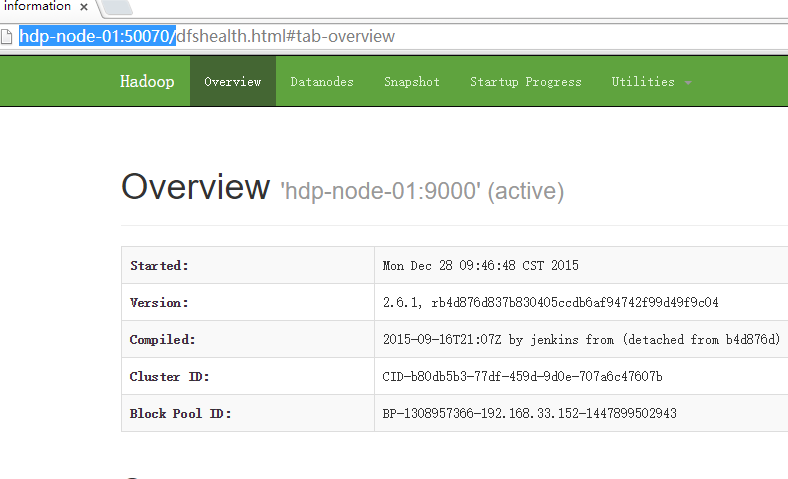
1、查看集群状态

命令： hdfs dfsadmin –report



可以看出，集群共有3个datanode可用

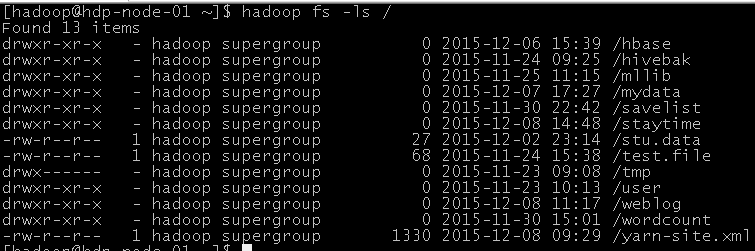
也可打开web控制台查看HDFS集群信息，在浏览器打开<http://hdp-node-01:50070/>



2、上传文件到HDFS

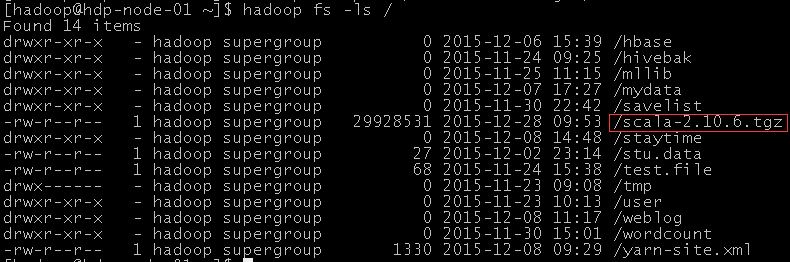
* 查看HDFS中的目录信息

命令： hadoop fs –ls /



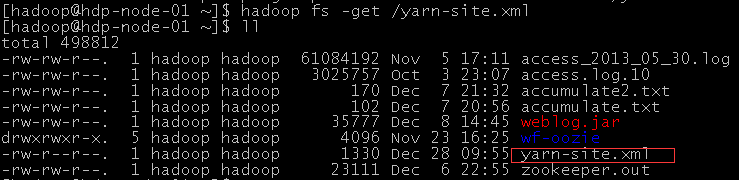
* 上传文件

命令： hadoop fs -put ./ scala-2.10.6.tgz to /



* 从HDFS下载文件

命令： hadoop fs -get /yarn-site.xml



#### MAPREDUCE使用

mapreduce是hadoop中的分布式运算编程框架，只要按照其编程规范，只需要编写少量的业务逻辑代码即可实现一个强大的海量数据并发处理程序

##### Demo开发——wordcount

1、需求

从大量（比如T级别）文本文件中，统计出每一个单词出现的总次数

2、mapreduce实现思路

**Map阶段：**

1. 从HDFS的源数据文件中逐行读取数据
2. 将每一行数据切分出单词
3. 为每一个单词构造一个键值对(单词，1)
4. 将键值对发送给reduce

**Reduce阶段：**

1. 接收map阶段输出的单词键值对
2. 将相同单词的键值对汇聚成一组
3. 对每一组，遍历组中的所有“值”，累加求和，即得到每一个单词的总次数
4. 将(单词，总次数)输出到HDFS的文件中
5. 具体编码实现

(1)定义一个mapper类

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

(2)定义一个reducer类

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

(3)定义一个主类，用来描述job并提交job

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

##### 程序打包运行

1. 将程序打包
2. 准备输入数据

vi /home/hadoop/test.txt

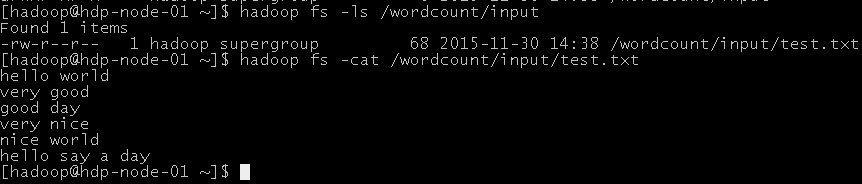
|  |
| --- |
| Hello tom  Hello jim  Hello ketty  Hello world  Ketty tom |

在hdfs上创建输入数据文件夹：

hadoop fs mkdir -p /wordcount/input

将words.txt上传到hdfs上

hadoop fs –put /home/hadoop/words.txt /wordcount/input



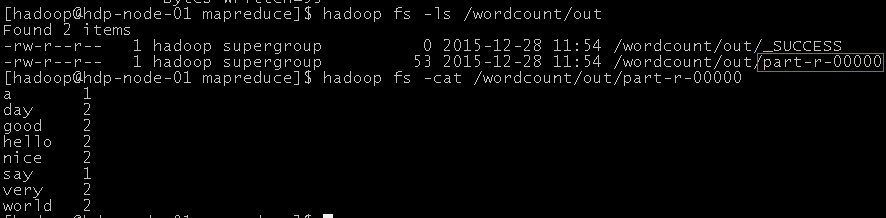
1. 将程序jar包上传到集群的任意一台服务器上
2. 使用命令启动执行wordcount程序jar包

$ hadoop jar wordcount.jar cn.itcast.bigdata.mrsimple.WordCountDriver /wordcount/input /wordcount/out



1. 查看执行结果

$ hadoop fs –cat /wordcount/out/part-r-00000



## HDFS详解

### HDFS的概念和特性

**首先，它是一个文件系统**，用于存储文件，通过统一的命名空间——目录树来定位文件

**其次，它是分布式的**，由很多服务器联合起来实现其功能，集群中的服务器有各自的角色；

**重要特性如下：**

1. HDFS中的文件在物理上是**分块存储（block）**，块的大小可以通过配置参数( dfs.blocksize)来规定，默认大小在hadoop2.x版本中是128M，老版本中是64M
2. HDFS文件系统会给客户端提供一个**统一的抽象目录树**，客户端通过路径来访问文件，形如：hdfs://namenode:port/dir-a/dir-b/dir-c/file.data
3. **目录结构及文件分块位置信息(元数据)**的管理由namenode节点承担

——namenode是HDFS集群主节点，负责维护整个hdfs文件系统的目录树，以及每一个路径（文件）所对应的block块信息（block的id，及所在的datanode服务器）

1. 文件的各个block的存储管理由datanode节点承担

---- datanode是HDFS集群从节点，每一个block都可以在多个datanode上存储多个副本（副本数量也可以通过参数设置dfs.replication，默认是3）

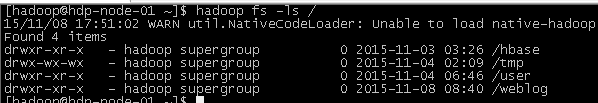
1. HDFS是设计成适应一次写入，多次读出的场景，且不支持文件的修改

*(注：适合用来做数据分析，并不适合用来做网盘应用，因为，不便修改，延迟大，网络开销大，成本太高)*

### HDFS的shell(命令行客户端)操作

#### HDFS命令行客户端使用

HDFS提供shell命令行客户端，使用方法如下：



#### 命令行客户端支持的命令参数

|  |
| --- |
| [-appendToFile <localsrc> ... <dst>]  [-cat [-ignoreCrc] <src> ...]  [-checksum <src> ...]  [-chgrp [-R] GROUP PATH...]  [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]  [-chown [-R] [OWNER][:[GROUP]] PATH...]  [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]  [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]  [-count [-q] <path> ...]  [-cp [-f] [-p] <src> ... <dst>]  [-createSnapshot <snapshotDir> [<snapshotName>]]  [-deleteSnapshot <snapshotDir> <snapshotName>]  [-df [-h] [<path> ...]]  [-du [-s] [-h] <path> ...]  [-expunge]  [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]  [-getfacl [-R] <path>]  [-getmerge [-nl] <src> <localdst>]  [-help [cmd ...]]  [-ls [-d] [-h] [-R] [<path> ...]]  [-mkdir [-p] <path> ...]  [-moveFromLocal <localsrc> ... <dst>]  [-moveToLocal <src> <localdst>]  [-mv <src> ... <dst>]  [-put [-f] [-p] <localsrc> ... <dst>]  [-renameSnapshot <snapshotDir> <oldName> <newName>]  [-rm [-f] [-r|-R] [-skipTrash] <src> ...]  [-rmdir [--ignore-fail-on-non-empty] <dir> ...]  [-setfacl [-R] [{-b|-k} {-m|-x <acl\_spec>} <path>]|[--set <acl\_spec> <path>]]  [-setrep [-R] [-w] <rep> <path> ...]  [-stat [format] <path> ...]  [-tail [-f] <file>]  [-test -[defsz] <path>]  [-text [-ignoreCrc] <src> ...]  [-touchz <path> ...]  [-usage [cmd ...]] |

#### 常用命令参数介绍

|  |
| --- |
| -help  功能：输出这个命令参数手册 |
| **-ls**  **功能：显示目录信息**  *示例： hadoop fs -ls hdfs://hadoop-server01:9000/*  *备注：这些参数中，所有的hdfs路径都可以简写*  *-->hadoop fs -ls / 等同于上一条命令的效果* |
| **-mkdir**  **功能：在hdfs上创建目录**  *示例：hadoop fs -mkdir -p /aaa/bbb/cc/dd* |
| **-moveFromLocal**  **功能：从本地剪切粘贴到hdfs**  *示例：hadoop fs - moveFromLocal /home/hadoop/a.txt /aaa/bbb/cc/dd*  **-moveToLocal**  **功能：从hdfs剪切粘贴到本地**  *示例：hadoop fs - moveToLocal /aaa/bbb/cc/dd /home/hadoop/a.txt* |
| **--appendToFile**  **功能：追加一个文件到已经存在的文件末尾**  *示例：hadoop fs -appendToFile ./hello.txt hdfs://hadoop-server01:9000/hello.txt*  *可以简写为：*  *Hadoop fs -appendToFile ./hello.txt /hello.txt* |
| **-cat**  **功能：显示文件内容**  *示例：hadoop fs -cat /hello.txt*  **-tail**  **功能：显示一个文件的末尾**  *示例：hadoop fs -tail /weblog/access\_log.1*  **-text**  **功能：以字符形式打印一个文件的内容**  *示例：hadoop fs -text /weblog/access\_log.1* |
| **-chgrp**  **-chmod**  **-chown**  **功能：linux文件系统中的用法一样，对文件所属权限**  *示例：*  *hadoop fs -chmod 666 /hello.txt*  *hadoop fs -chown someuser:somegrp /hello.txt* |
| **-copyFromLocal**  **功能：从本地文件系统中拷贝文件到hdfs路径去**  *示例：hadoop fs -copyFromLocal ./jdk.tar.gz /aaa/*  **-copyToLocal**  **功能：从hdfs拷贝到本地**  *示例：hadoop fs -copyToLocal /aaa/jdk.tar.gz* |
| **-cp**  **功能：从hdfs的一个路径拷贝hdfs的另一个路径**  *示例： hadoop fs -cp /aaa/jdk.tar.gz /bbb/jdk.tar.gz.2*  **-mv**  **功能：在hdfs目录中移动文件**  *示例： hadoop fs -mv /aaa/jdk.tar.gz /* |
| **-get**  **功能：等同于copyToLocal，就是从hdfs下载文件到本地**  示例：hadoop fs -get /aaa/jdk.tar.gz  **-**  **功能：合并下载多个文件**  *示例：比***getmerge** *如hdfs的目录 /aaa/下有多个文件:log.1, log.2,log.3,...*  hadoop fs -getmerge /aaa/log.\* ./log.sum |
| **-put**  **功能：等同于copyFromLocal**  *示例：hadoop fs -put /aaa/jdk.tar.gz /bbb/jdk.tar.gz.2* |
| **-rm**  **功能：删除文件或文件夹**  *示例：hadoop fs -rm -r /aaa/bbb/*  **-rmdir**  **功能：删除空目录**  *示例：hadoop fs -rmdir /aaa/bbb/ccc* |
| **-df**  **功能：统计文件系统的可用空间信息**  *示例：hadoop fs -df -h /*  **-du**  **功能：统计文件夹的大小信息**  *示例：*  *hadoop fs -du -s -h /aaa/\** |
| **-count**  **功能：统计一个指定目录下的文件节点数量**  *示例：hadoop fs -count /aaa/* |
| **-setrep**  **功能：设置hdfs中文件的副本数量**  *示例：hadoop fs -setrep 3 /aaa/jdk.tar.gz* |

补充：查看dfs集群工作状态的命令

hdfs dfsadmin -report

### hdfs的工作机制

*（工作机制的学习主要是为加深对分布式系统的理解，以及增强遇到各种问题时的分析解决能力，形成一定的集群运维能力）*

*注：很多不是真正理解hadoop技术体系的人会常常觉得HDFS可用于网盘类应用，但实际并非如此。要想将技术准确用在恰当的地方，必须对技术有深刻的理解*

#### 概述

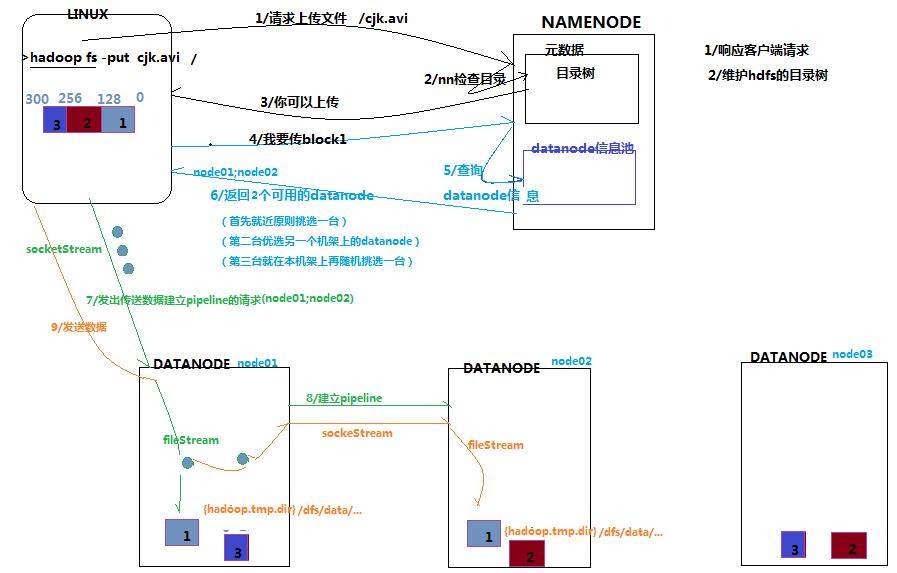
1. HDFS集群分为两大角色：NameNode、DataNode (Secondary Namenode)
2. NameNode负责管理整个文件系统的元数据
3. DataNode 负责管理用户的文件数据块
4. 文件会按照固定的大小（blocksize）切成若干块后分布式存储在若干台datanode上
5. 每一个文件块可以有多个副本，并存放在不同的datanode上
6. Datanode会定期向Namenode汇报自身所保存的文件block信息，而namenode则会负责保持文件的副本数量
7. HDFS的内部工作机制对客户端保持透明，客户端请求访问HDFS都是通过向namenode申请来进行

#### HDFS写数据流程

##### 概述

客户端要向HDFS写数据，首先要跟namenode通信以确认可以写文件并获得接收文件block的datanode，然后，客户端按顺序将文件逐个block传递给相应datanode，并由接收到block的datanode负责向其他datanode复制block的副本

##### 详细步骤图



##### 详细步骤解析

1、根namenode通信**请求上传文件**，namenode检查目标文件是否已存在，父目录是否存在

2、namenode返回是否可以上传

3、client请求第一个 block该传输到哪些datanode服务器上

4、namenode返回3个datanode服务器ABC

5、client请求3台dn中的一台A上传数据（本质上是一个RPC调用，建立pipeline），A收到请求会继续调用B，然后B调用C，将真个pipeline建立完成，逐级返回客户端

6、client开始往A上传第一个block（先从磁盘读取数据放到一个本地内存缓存），以packet为单位，A收到一个packet就会传给B，B传给C；A每传一个packet会放入一个应答队列等待应答

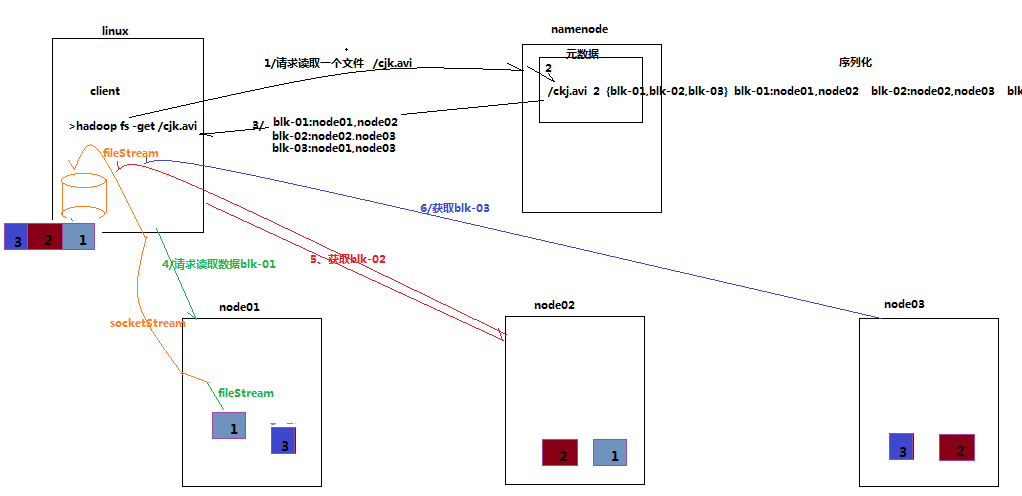
7、当一个block传输完成之后，client再次请求namenode上传第二个block的服务器。

#### HDFS读数据流程

##### 概述

客户端将要读取的文件路径发送给namenode，namenode获取文件的元信息（主要是block的存放位置信息）返回给客户端，客户端根据返回的信息找到相应datanode逐个获取文件的block并在客户端本地进行数据追加合并从而获得整个文件

##### 详细步骤图



##### 详细步骤解析

1、跟namenode通信查询元数据，找到文件块所在的datanode服务器

2、挑选一台datanode（就近原则，然后随机）服务器，请求建立socket流

3、datanode开始发送数据（从磁盘里面读取数据放入流，以packet为单位来做校验）

4、客户端以packet为单位接收，现在本地缓存，然后写入目标文件

### NAMENODE工作机制

学习目标：理解namenode的工作机制尤其是**元数据管理**机制，以增强对HDFS工作原理的理解，及培养hadoop集群运营中“性能调优”、“namenode”故障问题的分析解决能力

*问题场景：*

*1、集群启动后，可以查看目录，但是上传文件时报错，打开web页面可看到namenode正处于safemode状态，怎么处理？*

解释：

safemode是namenode的一种状态（active/standby/safemode安全模式）

namenode进入安全模式的原理：

a、namenode发现集群中的block丢失率达到一定比例时（0.01%），namenode就会进入安全模式，在安全模式下，客户端不能对任何数据进行操作，只能查看元数据信息（比如ls/mkdir）

b、如何退出安全模式？

找到问题所在，进行修复（比如修复宕机的datanode）

或者可以手动强行退出安全模式（没有真正解决问题）： hdfs namenode --safemode leave

c、在hdfs集群正常冷启动时，namenode也会在safemode状态下维持相当长的一段时间，此时你不需要去理会，等待它自动退出安全模式即可

(原理：

namenode的内存元数据中，包含文件路径、副本数、blockid，及每一个block所在datanode的信息，而fsimage中，不包含block所在的datanode信息，那么，当namenode冷启动时，此时内存中的元数据只能从fsimage中加载而来，从而就没有block所在的datanode信息——>就会导致namenode认为所有的block都已经丢失——>进入安全模式——>datanode启动后，会定期向namenode汇报自身所持有的blockid信息，——>随着datanode陆续启动，从而陆续汇报block信息，namenode就会将内存元数据中的block所在datanode信息补全更新——>找到了所有block的位置，从而自动退出安全模式)

*2、Namenode服务器的磁盘故障导致namenode宕机，如何挽救集群及数据？*

可以从SecondaryNameNode把元数据拷贝过来

*3、Namenode是否可以有多个？namenode内存要配置多大？namenode跟集群数据存储能力有关系吗？*

可以有多个,hadoop有HA机制,可以解决这个问题

内存根据需求来配,根据需要存多少数据来算一下,一个记录大概占150字节

*4、文件的blocksize究竟调大好还是调小好？--结合mapreduce*

结合MapReduce,业务数据运行起来测试,

*……*

*诸如此类问题的回答，都需要基于对namenode自身的工作原理的深刻理解*

#### NAMENODE职责

NAMENODE职责：

负责客户端请求的响应

元数据的管理（查询，修改）

#### 元数据管理

namenode对数据的管理采用了三种存储形式：

内存元数据(NameSystem)

磁盘元数据镜像文件

数据操作日志文件（可通过日志运算出元数据）

##### 元数据存储机制

A、内存中有一份完整的元数据(**内存meta data**)

B、磁盘有一个“准完整”的元数据镜像（**fsimage**）文件(在namenode的工作目录中)

C、用于衔接内存metadata和持久化元数据镜像fsimage之间的操作日志（**edits文件**）*注：当客户端对hdfs中的文件进行新增或者修改操作，操作记录首先被记入edits日志文件中，当客户端操作成功后，相应的元数据会更新到内存meta.data中*

##### 元数据手动查看

可以通过hdfs的一个工具来查看edits中的信息

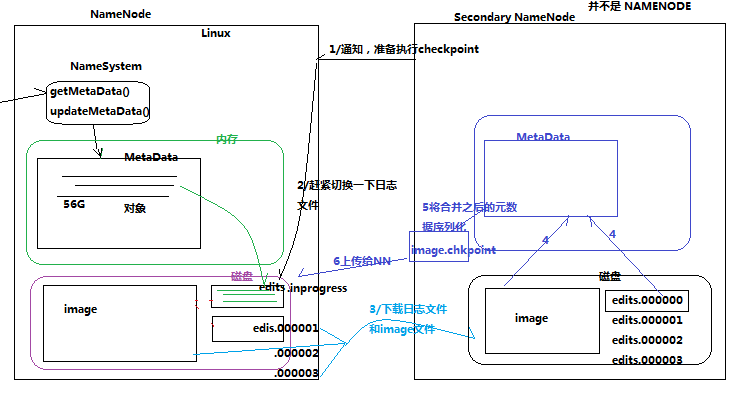
bin/hdfs oev -i edits -o edits.xml

bin/hdfs oiv -i fsimage\_0000000000000000087 -p XML -o fsimage.xml

##### 元数据的checkpoint

每隔一段时间，会由secondary namenode将namenode上积累的所有edits和一个最新的fsimage下载到本地，并加载到内存进行merge（这个过程称为checkpoint）

* checkpoint的详细过程



* checkpoint操作的触发条件配置参数

|  |
| --- |
| dfs.namenode.checkpoint.check.period=60 #检查触发条件是否满足的频率，60秒  dfs.namenode.checkpoint.dir=file://${hadoop.tmp.dir}/dfs/namesecondary  #以上两个参数做checkpoint操作时，secondary namenode的本地工作目录  dfs.namenode.checkpoint.edits.dir=${dfs.namenode.checkpoint.dir}  dfs.namenode.checkpoint.max-retries=3 #最大重试次数  dfs.namenode.checkpoint.period=3600 #两次checkpoint之间的时间间隔3600秒  dfs.namenode.checkpoint.txns=1000000 #两次checkpoint之间最大的操作记录 |

* checkpoint的附带作用

namenode和secondary namenode的工作目录存储结构完全相同，所以，当namenode故障退出需要重新恢复时，可以从secondary namenode的工作目录中将fsimage拷贝到namenode的工作目录，以恢复namenode的元数据

* 元数据目录说明

在第一次部署好Hadoop集群的时候，我们需要在NameNode（NN）节点上格式化磁盘：

|  |
| --- |
| $HADOOP\_HOME/bin/hdfs namenode -format |

格式化完成之后，将会在$dfs.namenode.name.dir/current目录下如下的文件结构

|  |
| --- |
| current/  |-- VERSION  |-- edits\_\*  |-- fsimage\_0000000000008547077  |-- fsimage\_0000000000008547077.md5  `-- seen\_txid |

其中的dfs.name.dir是在hdfs-site.xml文件中配置的，默认值如下：

|  |
| --- |
| <property>  <name>dfs.name.dir</name>  <value>file://${hadoop.tmp.dir}/dfs/name</value>  </property>  hadoop.tmp.dir是在core-site.xml中配置的，默认值如下  <property>  <name>hadoop.tmp.dir</name>  <value>/tmp/hadoop-${user.name}</value>  <description>A base for other temporary directories.</description>  </property> |

dfs.namenode.name.dir属性可以配置多个目录，

如/data1/dfs/name,/data2/dfs/name,/data3/dfs/name,....。各个目录存储的文件结构和内容都完全一样，相当于备份，这样做的好处是当其中一个目录损坏了，也不会影响到Hadoop的元数据，特别是当其中一个目录是NFS（网络文件系统Network File System，NFS）之上，即使你这台机器损坏了，元数据也得到保存。  
下面对$dfs.namenode.name.dir/current/目录下的文件进行解释。  
1、VERSION文件是Java属性文件，内容大致如下：

|  |
| --- |
| #Fri Nov 15 19:47:46 CST 2013  namespaceID=934548976  clusterID=CID-cdff7d73-93cd-4783-9399-0a22e6dce196  cTime=0  storageType=NAME\_NODE  blockpoolID=BP-893790215-192.168.24.72-1383809616115  layoutVersion=-47 |

其中  
　　（1）、namespaceID是文件系统的唯一标识符，在文件系统首次格式化之后生成的；  
　　（2）、storageType说明这个文件存储的是什么进程的数据结构信息（如果是DataNode，storageType=DATA\_NODE）；  
　　（3）、cTime表示NameNode存储时间的创建时间，由于我的NameNode没有更新过，所以这里的记录值为0，以后对NameNode升级之后，cTime将会记录更新时间戳；  
　　（4）、layoutVersion表示HDFS永久性数据结构的版本信息， 只要数据结构变更，版本号也要递减，此时的HDFS也需要升级，否则磁盘仍旧是使用旧版本的数据结构，这会导致新版本的NameNode无法使用；  
　　（5）、clusterID是系统生成或手动指定的集群ID，在-clusterid选项中可以使用它；如下说明

1. 使用如下命令格式化一个Namenode：

$HADOOP\_HOME/bin/hdfs namenode -format [-clusterId <cluster\_id>]

选择一个唯一的cluster\_id，并且这个cluster\_id不能与环境中其他集群有冲突。如果没有提供cluster\_id，则会自动生成一个唯一的ClusterID。

b、使用如下命令格式化其他Namenode：

$HADOOP\_HOME/bin/hdfs namenode -format -clusterId <cluster\_id>

c、升级集群至最新版本。在升级过程中需要提供一个ClusterID，例如：

$HADOOP\_PREFIX\_HOME/bin/hdfs start namenode --config $HADOOP\_CONF\_DIR  -upgrade -clusterId <cluster\_ID>

如果没有提供ClusterID，则会自动生成一个ClusterID。

　　（6）、blockpoolID：是针对每一个Namespace所对应的blockpool的ID，上面的这个BP-893790215-192.168.24.72-1383809616115就是在我的ns1的namespace下的存储块池的ID，这个ID包括了其对应的NameNode节点的ip地址。  
　　  
2、$dfs.namenode.name.dir/current/seen\_txid非常重要，是存放transactionId的文件，format之后是0，它代表的是namenode里面的edits\_\*文件的尾数，namenode重启的时候，会按照seen\_txid的数字，循序从头跑edits\_0000001~到seen\_txid的数字。所以当你的hdfs发生异常重启的时候，一定要比对seen\_txid内的数字是不是你edits最后的尾数，不然会发生建置namenode时metaData的资料有缺少，导致误删Datanode上多余Block的资讯。

3、$dfs.namenode.name.dir/current目录下在format的同时也会生成fsimage和edits文件，及其对应的md5校验文件。

补充：seen\_txid

文件中记录的是edits滚动的序号，每次重启namenode时，namenode就知道要将哪些edits进行加载edits

### DATANODE工作机制

*问题场景：*

*1、集群容量不够，怎么扩容？*

*2、如果有一些datanode宕机，该怎么办？*

*3、datanode明明已启动，但是集群中的可用datanode列表中就是没有，怎么办？*

*以上这类问题的解答，有赖于对datanode工作机制的深刻理解*

#### 概述

1、Datanode工作职责：

存储管理用户的文件块数据

定期向namenode汇报自身所持有的block信息（通过心跳信息上报）

（这点很重要，因为，当集群中发生某些block副本失效时，集群如何恢复block初始副本数量的问题）

|  |
| --- |
| <property>  <name>dfs.blockreport.intervalMsec</name>  <value>3600000</value>  <description>Determines block reporting interval in milliseconds.</description>  </property> |

2、Datanode掉线判断时限参数

datanode进程死亡或者网络故障造成datanode无法与namenode通信，namenode不会立即把该节点判定为死亡，要经过一段时间，这段时间暂称作超时时长。HDFS默认的超时时长为10分钟+30秒。如果定义超时时间为timeout，则超时时长的计算公式为：

timeout = 2 \* heartbeat.recheck.interval + 10 \* dfs.heartbeat.interval。

而默认的heartbeat.recheck.interval 大小为5分钟，dfs.heartbeat.interval默认为3秒。

需要注意的是hdfs-site.xml 配置文件中的heartbeat.recheck.interval的单位为毫秒，dfs.heartbeat.interval的单位为秒。所以，举个例子，如果heartbeat.recheck.interval设置为5000（毫秒），dfs.heartbeat.interval设置为3（秒，默认），则总的超时时间为40秒。

|  |
| --- |
| <property>  <name>heartbeat.recheck.interval</name>  <value>2000</value>  </property>  <property>  <name>dfs.heartbeat.interval</name>  <value>1</value>  </property> |

#### 观察NATANODE功能

上传一个文件，观察文件的block具体的物理存放情况：

在每一台datanode机器上的这个目录中能找到文件的切块：

/home/hadoop/app/hadoop-2.4.1/tmp/dfs/data/current/BP-193442119-192.168.2.120-1432457733977/current/finalized

### HDFS的java操作

*hdfs在生产应用中主要是客户端的开发，其核心步骤是从hdfs提供的api中构造一个HDFS的访问客户端对象，然后通过该客户端对象操作（增删改查）HDFS上的文件*

#### 搭建开发环境

1、引入依赖

|  |  |
| --- | --- |
| |  | | --- | | <dependency>  <groupId>org.apache.hadoop</groupId>  <artifactId>hadoop-client</artifactId>  <version>2.6.4</version>  </dependency> | |

*注：如需手动引入jar包，hdfs的jar包----hadoop的安装目录的share下*

2、window下开发的说明

建议在linux下进行hadoop应用的开发，不会存在兼容性问题。如在window上做客户端应用开发，需要设置以下环境：

1. 用老师给的windows平台下编译的hadoop安装包解压一份到windows的任意一个目录下
2. 在window系统中配置HADOOP\_HOME指向你解压的安装包目录
3. 在windows系统的path变量中加入HADOOP\_HOME的bin目录

#### 获取api中的客户端对象

在java中操作hdfs，首先要获得一个客户端实例

|  |
| --- |
| Configuration conf = new Configuration()  FileSystem fs = FileSystem.get(conf) |

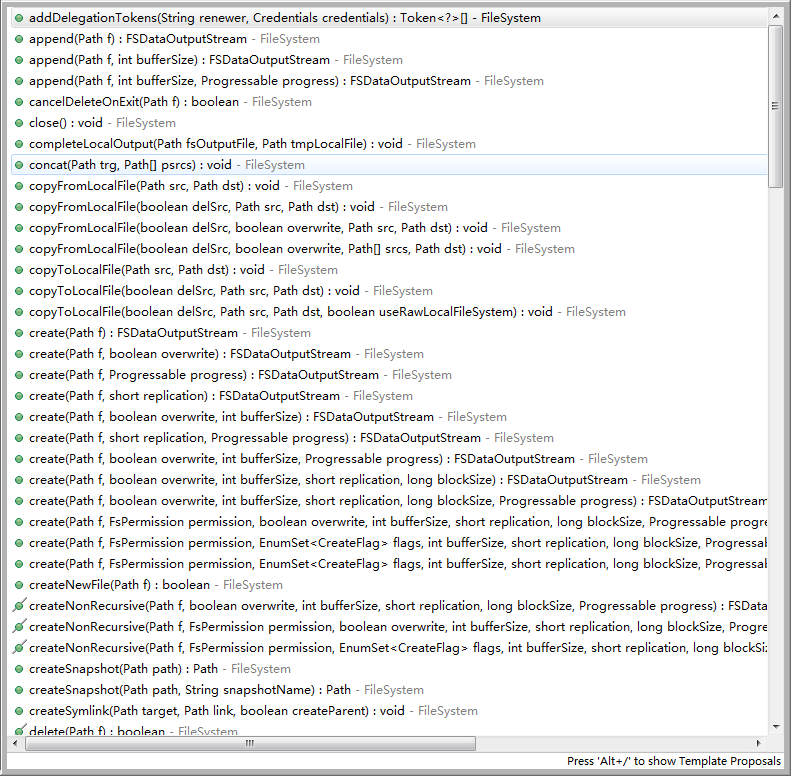
而我们的操作目标是HDFS，所以获取到的fs对象应该是DistributedFileSystem的实例；

get方法是从何处判断具体实例化那种客户端类呢？

**——从conf中的一个参数 fs.defaultFS的配置值判断；**

如果我们的代码中没有指定fs.defaultFS，并且工程classpath下也没有给定相应的配置，conf中的默认值就来自于hadoop的jar包中的core-default.xml，默认值为： file:///，则获取的将不是一个DistributedFileSystem的实例，而是一个本地文件系统的客户端对象

#### DistributedFileSystem实例对象所具备的方法



#### HDFS客户端操作数据代码示例：

##### 文件的增删改查

|  |
| --- |
| public class HdfsClient {  FileSystem fs = null;  @Before  public void init() throws Exception {  // 构造一个配置参数对象，设置一个参数：我们要访问的hdfs的URI  // 从而FileSystem.get()方法就知道应该是去构造一个访问hdfs文件系统的客户端，以及hdfs的访问地址  // new Configuration();的时候，它就会去加载jar包中的hdfs-default.xml  // 然后再加载classpath下的hdfs-site.xml  Configuration conf = new Configuration();  conf.set("fs.defaultFS", "hdfs://hdp-node01:9000");  /\*\*  \* 参数优先级： 1、客户端代码中设置的值 2、classpath下的用户自定义配置文件 3、然后是服务器的默认配置  \*/  conf.set("dfs.replication", "3");  // 获取一个hdfs的访问客户端，根据参数，这个实例应该是DistributedFileSystem的实例  // fs = FileSystem.get(conf);  // 如果这样去获取，那conf里面就可以不要配"fs.defaultFS"参数，而且，这个客户端的身份标识已经是hadoop用户  fs = FileSystem.get(new URI("hdfs://hdp-node01:9000"), conf, "hadoop");  }  /\*\*  \* 往hdfs上传文件  \*  \* @throws Exception  \*/  @Test  public void testAddFileToHdfs() throws Exception {  // 要上传的文件所在的本地路径  Path src = new Path("g:/redis-recommend.zip");  // 要上传到hdfs的目标路径  Path dst = new Path("/aaa");  fs.copyFromLocalFile(src, dst);  fs.close();  }  /\*\*  \* 从hdfs中复制文件到本地文件系统  \*  \* @throws IOException  \* @throws IllegalArgumentException  \*/  @Test  public void testDownloadFileToLocal() throws IllegalArgumentException, IOException {  fs.copyToLocalFile(new Path("/jdk-7u65-linux-i586.tar.gz"), new Path("d:/"));  fs.close();  }  @Test  public void testMkdirAndDeleteAndRename() throws IllegalArgumentException, IOException {  // 创建目录  fs.mkdirs(new Path("/a1/b1/c1"));  // 删除文件夹 ，如果是非空文件夹，参数2必须给值true  fs.delete(new Path("/aaa"), true);  // 重命名文件或文件夹  fs.rename(new Path("/a1"), new Path("/a2"));  }  /\*\*  \* 查看目录信息，只显示文件  \*  \* @throws IOException  \* @throws IllegalArgumentException  \* @throws FileNotFoundException  \*/  @Test  public void testListFiles() throws FileNotFoundException, IllegalArgumentException, IOException {  // 思考：为什么返回迭代器，而不是List之类的容器  RemoteIterator<LocatedFileStatus> listFiles = fs.listFiles(new Path("/"), true);  while (listFiles.hasNext()) {  LocatedFileStatus fileStatus = listFiles.next();  System.out.println(fileStatus.getPath().getName());  System.out.println(fileStatus.getBlockSize());  System.out.println(fileStatus.getPermission());  System.out.println(fileStatus.getLen());  BlockLocation[] blockLocations = fileStatus.getBlockLocations();  for (BlockLocation bl : blockLocations) {  System.out.println("block-length:" + bl.getLength() + "--" + "block-offset:" + bl.getOffset());  String[] hosts = bl.getHosts();  for (String host : hosts) {  System.out.println(host);  }  }  System.out.println("--------------为angelababy打印的分割线--------------");  }  }  /\*\*  \* 查看文件及文件夹信息  \*  \* @throws IOException  \* @throws IllegalArgumentException  \* @throws FileNotFoundException  \*/  @Test  public void testListAll() throws FileNotFoundException, IllegalArgumentException, IOException {  FileStatus[] listStatus = fs.listStatus(new Path("/"));  String flag = "d-- ";  for (FileStatus fstatus : listStatus) {  if (fstatus.isFile()) flag = "f-- ";  System.out.println(flag + fstatus.getPath().getName());  }  }  } |

##### 通过流的方式访问hdfs

|  |
| --- |
| */\*\**  *\* 相对那些封装好的方法而言的更底层一些的操作方式*  *\* 上层那些mapreduce spark等运算框架，去hdfs中获取数据的时候，就是调的这种底层的api*  *\* @author*  *\**  *\*/*  *public class StreamAccess {*    *FileSystem fs = null;*  *@Before*  *public void init() throws Exception {*  *Configuration conf = new Configuration();*  *fs = FileSystem.get(new URI("hdfs://hdp-node01:9000"), conf, "hadoop");*  *}*    */\*\**  *\* 通过流的方式上传文件到hdfs*  *\* @throws Exception*  *\*/*  *@Test*  *public void testUpload() throws Exception {*    *FSDataOutputStream outputStream = fs.create(new Path("/angelababy.love"), true);*  *FileInputStream inputStream = new FileInputStream("c:/angelababy.love");*    *IOUtils.copy(inputStream, outputStream);*    *}*    *@Test*  *public void testDownLoadFileToLocal() throws IllegalArgumentException, IOException{*    *//先获取一个文件的输入流----针对hdfs上的*  *FSDataInputStream in = fs.open(new Path("/jdk-7u65-linux-i586.tar.gz"));*    *//再构造一个文件的输出流----针对本地的*  *FileOutputStream out = new FileOutputStream(new File("c:/jdk.tar.gz"));*    *//再将输入流中数据传输到输出流*  *IOUtils.copyBytes(in, out, 4096);*      *}*      */\*\**  *\* hdfs支持随机定位进行文件读取，而且可以方便地读取指定长度*  *\* 用于上层分布式运算框架并发处理数据*  *\* @throws IllegalArgumentException*  *\* @throws IOException*  *\*/*  *@Test*  *public void testRandomAccess() throws IllegalArgumentException, IOException{*  *//先获取一个文件的输入流----针对hdfs上的*  *FSDataInputStream in = fs.open(new Path("/iloveyou.txt"));*      *//可以将流的起始偏移量进行自定义*  *in.seek(22);*    *//再构造一个文件的输出流----针对本地的*  *FileOutputStream out = new FileOutputStream(new File("c:/iloveyou.line.2.txt"));*    *IOUtils.copyBytes(in,out,19L,true);*    *}*        */\*\**  *\* 显示hdfs上文件的内容*  *\* @throws IOException*  *\* @throws IllegalArgumentException*  *\*/*  *@Test*  *public void testCat() throws IllegalArgumentException, IOException{*    *FSDataInputStream in = fs.open(new Path("/iloveyou.txt"));*    *IOUtils.copyBytes(in, System.out, 1024);*  *}*  *}* |

##### 场景编程

在mapreduce 、spark等运算框架中，有一个核心思想就是将运算移往数据，或者说，就是要在并发计算中尽可能让运算本地化，这就需要获取数据所在位置的信息并进行相应范围读取

以下模拟实现：获取一个文件的所有block位置信息，然后读取指定block中的内容

|  |
| --- |
| *@Test*  *public void testCat() throws IllegalArgumentException, IOException{*    *FSDataInputStream in = fs.open(new Path("/weblog/input/access.log.10"));*  *//拿到文件信息*  *FileStatus[] listStatus = fs.listStatus(new Path("/weblog/input/access.log.10"));*  *//获取这个文件的所有block的信息*  *BlockLocation[] fileBlockLocations = fs.getFileBlockLocations(listStatus[0], 0L, listStatus[0].getLen());*  *//第一个block的长度*  *long length = fileBlockLocations[0].getLength();*  *//第一个block的起始偏移量*  *long offset = fileBlockLocations[0].getOffset();*    *System.out.println(length);*  *System.out.println(offset);*    *//获取第一个block写入输出流*  *// IOUtils.copyBytes(in, System.out, (int)length);*  *byte[] b = new byte[4096];*    *FileOutputStream os = new FileOutputStream(new File("d:/block0"));*  *while(in.read(offset, b, 0, 4096)!=-1){*  *os.write(b);*  *offset += 4096;*  *if(offset>=length) return;*  *};*  *os.flush();*  *os.close();*  *in.close();*  *}* |

### 案例1：开发shell采集脚本

#### 需求说明

点击流日志每天都10T，在业务应用服务器上，需要准实时上传至（Hadoop HDFS）上

#### 需求分析

一般上传文件都是在凌晨24点操作，由于很多种类的业务数据都要在晚上进行传输，为了减轻服务器的压力**，避开高峰期**。

如果需要伪实时的上传，则采用定时上传的方式

#### 技术分析

**HDFS SHELL**: hadoop fs –put xxxx.log /data 还可以使用 Java Api

满足上传一个文件，不能满足定时、周期性传入。

**定时调度器**：

**Linux crontab**

crontab -e

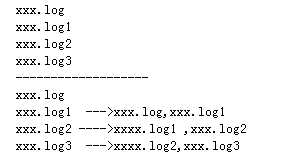
\*/5 \* \* \* \* $home/bin/command.sh //五分钟执行一次

系统会自动执行脚本，每5分钟一次，执行时判断文件是否符合上传规则，符合则上传

#### 实现流程

##### 日志产生程序

日志产生程序将日志生成后，产生一个一个的文件，使用滚动模式创建文件名。



日志生成的逻辑由业务系统决定，比如在log4j配置文件中配置生成规则，如：当xxxx.log 等于10G时，滚动生成新日志

|  |
| --- |
| *log4j.logger.msg=info,msg*  *log4j.appender.msg=cn.maoxiangyi.MyRollingFileAppender*  *log4j.appender.msg.layout=org.apache.log4j.PatternLayout*  *log4j.appender.msg.layout.ConversionPattern=%m%n*  *log4j.appender.msg.datePattern='.'yyyy-MM-dd*  *log4j.appender.msg.Threshold=info*  *log4j.appender.msg.append=true*  *log4j.appender.msg.encoding=UTF-8*  *log4j.appender.msg.MaxBackupIndex=100*  *log4j.appender.msg.MaxFileSize=10GB*  *log4j.appender.msg.File=/home/hadoop/logs/log/access.log* |

细节：

1. 如果日志文件后缀是1\2\3等数字，该文件满足需求可以上传的话。把该文件移动到准备上传的工作区间。
2. 工作区间有文件之后，可以使用hadoop put命令将文件上传。

阶段问题：

1. 待上传文件的工作区间的文件，在上传完成之后，是否需要删除掉。

##### 伪代码

使用ls命令读取指定路径下的所有文件信息，

ls | while read line

//判断line这个文件名称是否符合规则

if line=access.log.\* (

将文件移动到待上传的工作区间

)

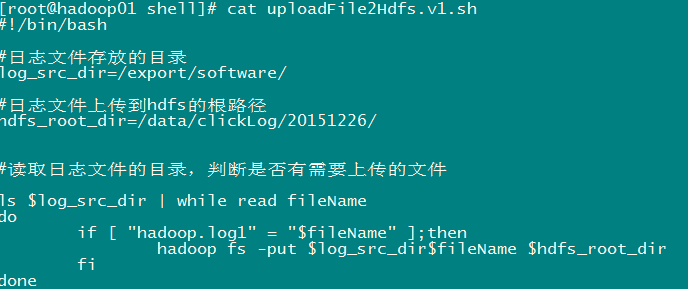
//批量上传工作区间的文件

hadoop fs –put xxx

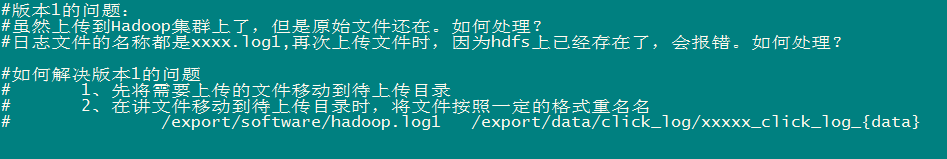
**脚本写完之后**，配置linux定时任务，每5分钟运行一次。

#### 代码实现

代码第一版本，实现基本的上传功能和定时调度功能



代码第二版本：增强版V2(基本能用，还是不够健全)





#### 效果展示及操作步骤

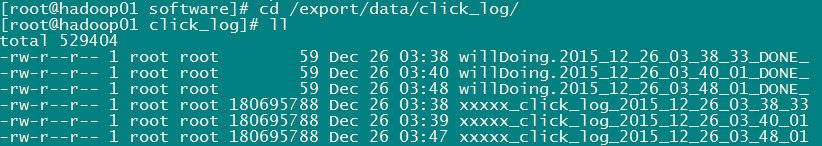
1、日志收集文件收集数据，并将数据保存起来，效果如下：



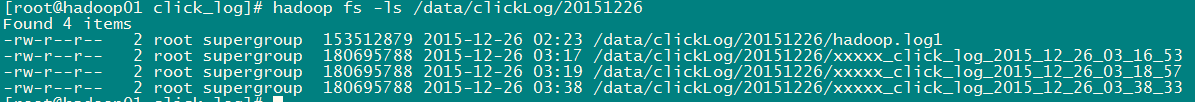
2、上传程序通过crontab定时调度



3、程序运行时产生的临时文件



4、Hadoo hdfs上的效果



## MapReduce详解

### MAPREDUCE实践篇(wordcount)（1）

#### MAPREDUCE 示例编写及编程规范

##### 编程规范

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

##### wordcount示例编写

需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(1)定义一个mapper类

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

(2)定义一个reducer类

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

(3)定义一个主类，用来描述job并提交job

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

#### MAPREDUCE程序运行模式

##### 本地运行模式

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

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

*％HADOOP\_HOME％ = d:/hadoop-2.6.1*

*%PATH% = ％HADOOP\_HOME％\bin*

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

##### 集群运行模式

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

A、将程序打成JAR包，然后在集群的任意一个节点上用hadoop命令启动

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

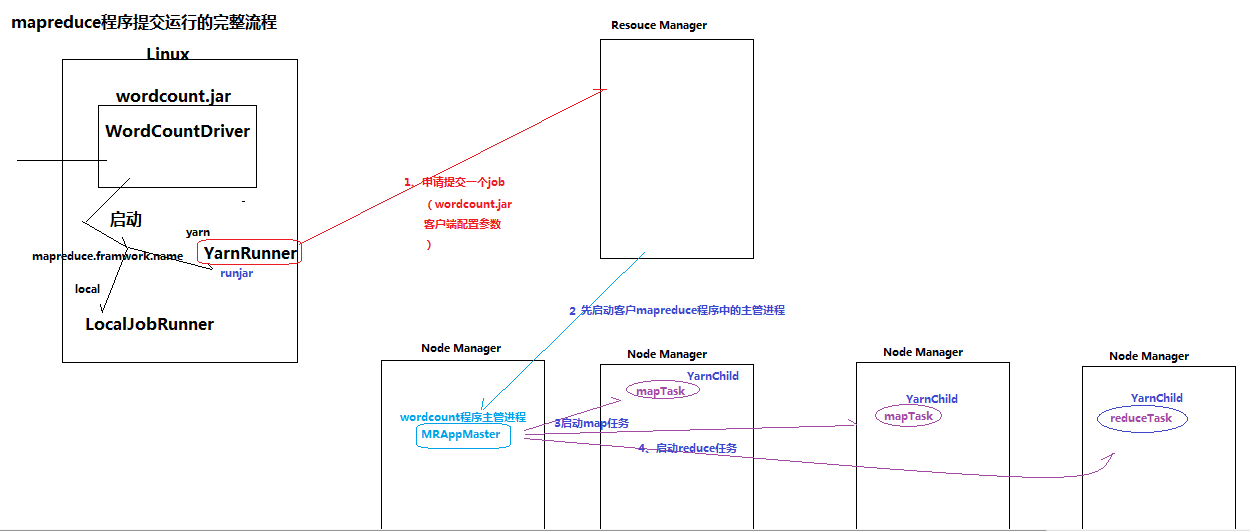
(hadoop jar jarFile mainClass args...)

B、直接在linux的eclipse中运行main方法

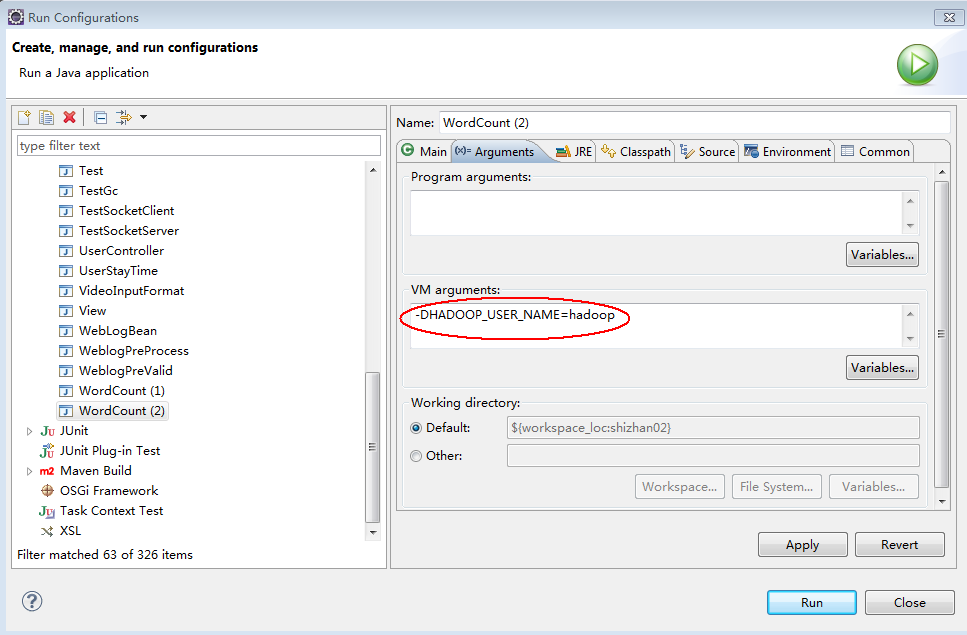
**（项目中要带参数：mapreduce.framework.name=yarn以及yarn的两个基本配置）**

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

mapreduce程序在集群中运行时的大体流程：



附：在windows平台上访问hadoop时改变自身身份标识的方法之二：



### MapReduce原理篇(1)

*Mapreduce是一个分布式运算程序的****编程框架****，是用户开发“基于hadoop的数据分析应用”的核心框架；*

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

#### 为什么要MapReduce

（1）海量数据在单机上处理因为硬件资源限制，无法胜任

（2）而一旦将单机版程序扩展到集群来分布式运行，将极大增加程序的复杂度和开发难度

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

*设想一个海量数据场景下的wordcount需求：*

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

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

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

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

#### MapReduce框架结构及核心运行机制

##### 结构

一个完整的mapreduce程序在分布式运行时有三类实例进程：

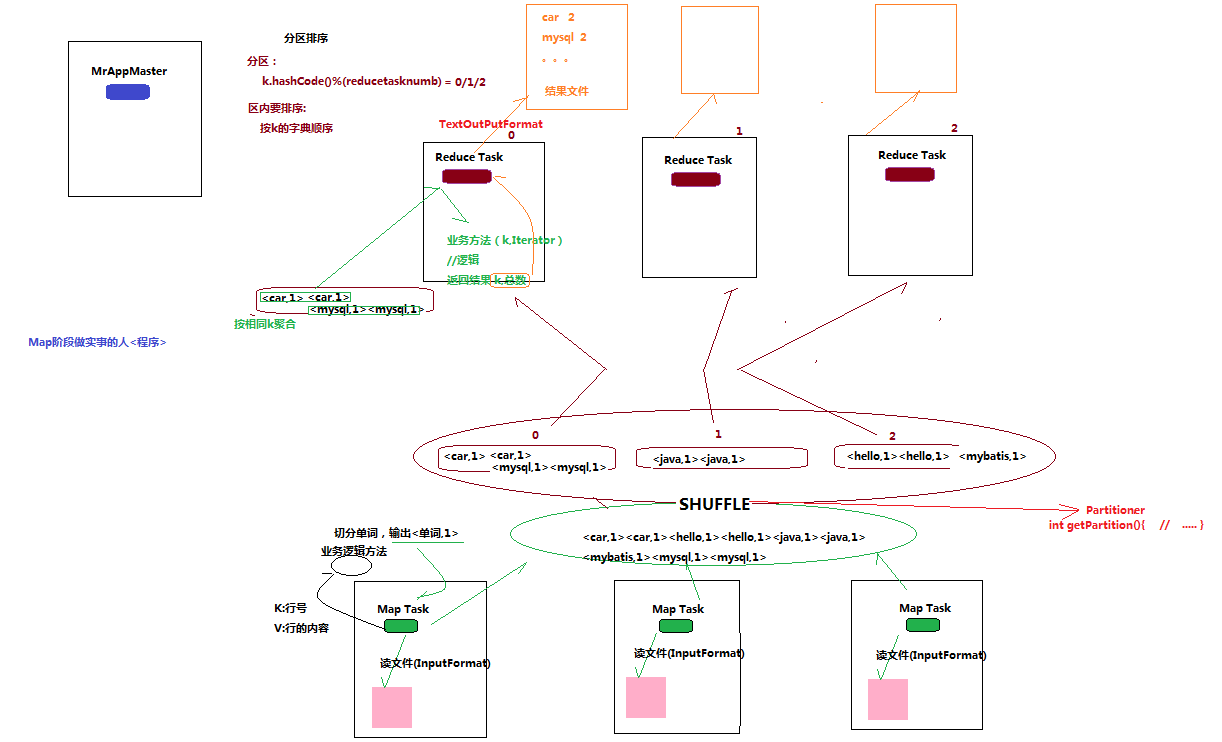
1、MRAppMaster：负责整个程序的过程调度及状态协调

2、mapTask：负责map阶段的整个数据处理流程

3、ReduceTask：负责reduce阶段的整个数据处理流程

##### MR程序运行流程

流程示意图



流程解析

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

#### MapTask并行度决定机制

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

*那么，mapTask并行实例是否越多越好呢？其并行度又是如何决定呢？*

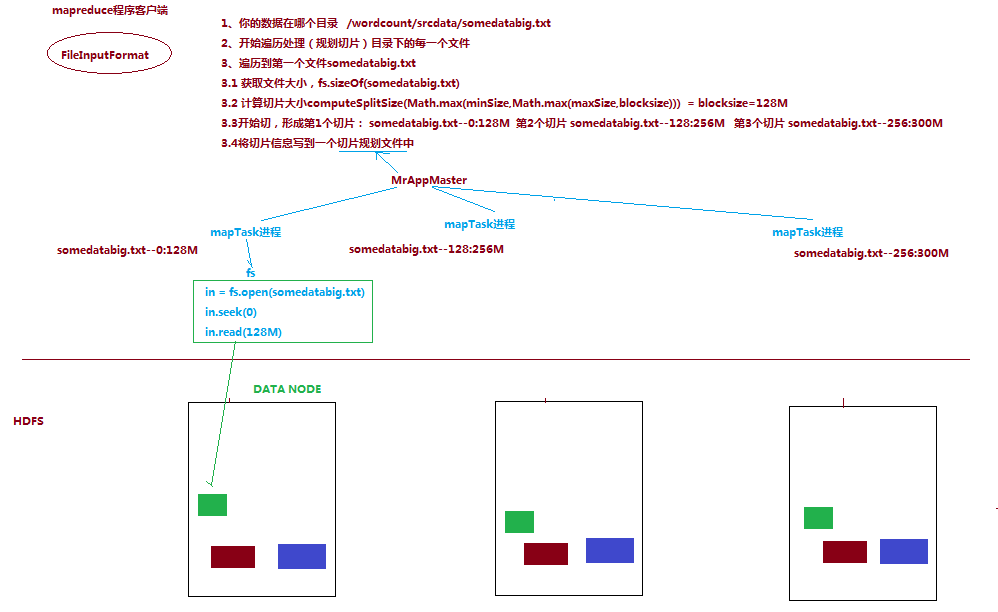
1.3.1 mapTask并行度的决定机制

一个job的map阶段并行度由客户端在提交job时决定

而客户端对map阶段并行度的规划的基本逻辑为：

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

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



#### FileInputFormat切片机制

**1、切片定义在InputFormat类中的getSplit()方法**

**2、FileInputFormat中默认的切片机制**：

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

比如待处理数据有两个文件：

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

经过FileInputFormat的切片机制运算后，形成的切片信息如下：

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

**3、FileInputFormat中切片的大小的参数配置**

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

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

因此，**默认情况下，切片大小=blocksize**

maxsize（切片最大值）：

参数如果调得比blocksize小，则会让切片变小，而且就等于配置的这个参数的值

minsize （切片最小值）：

参数调的比blockSize大，则可以让切片变得比blocksize还大

**但是，不论怎么调参数，都不能让多个小文件“划入”一个split**

选择并发数的影响因素：

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

#### map并行度的经验之谈

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

* 如果job的每个map或者 reduce task的运行时间都只有30-40秒钟，那么就减少该job的map或者reduce数，每一个task(map|reduce)的setup和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个task都非常快就跑完了，就会在task的开始和结束的时候浪费太多的时间。

配置task的JVM重用(**JVM重用技术**不是指同一Job的两个或两个以上的task可以同时运行于同一JVM上，而是排队按顺序执行。)以改善该问题：

*（mapred.job.reuse.jvm.num.tasks，默认是1，表示一个JVM上最多可以顺序执行的task*

*数目（属于同一个Job）是1。也就是说一个task启一个JVM）*

* 如果input的文件非常的大，比如1TB，可以考虑将hdfs上的每个block size设大，比如设成256MB或者512MB

#### ReduceTask并行度的决定

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

//默认值是1，手动设置为4

job.setNumReduceTasks(4);

如果数据分布不均匀，就有可能在reduce阶段产生数据倾斜

*注意： reducetask数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能有1个reducetask*

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

### MAPREDUCE原理篇（2）

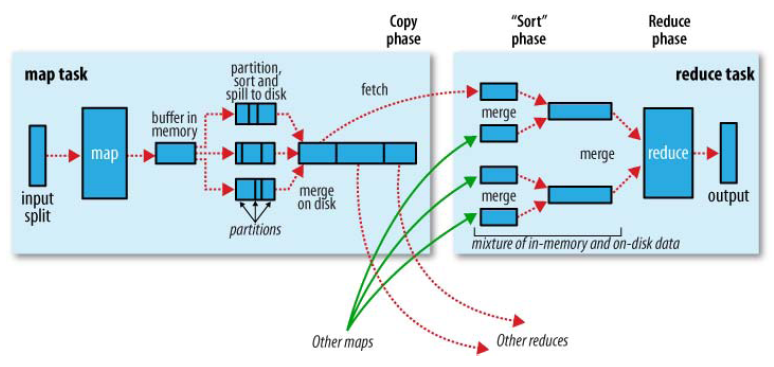
#### mapreduce的shuffle机制

##### 概述：

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

##### 主要流程：

Shuffle缓存流程：



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

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

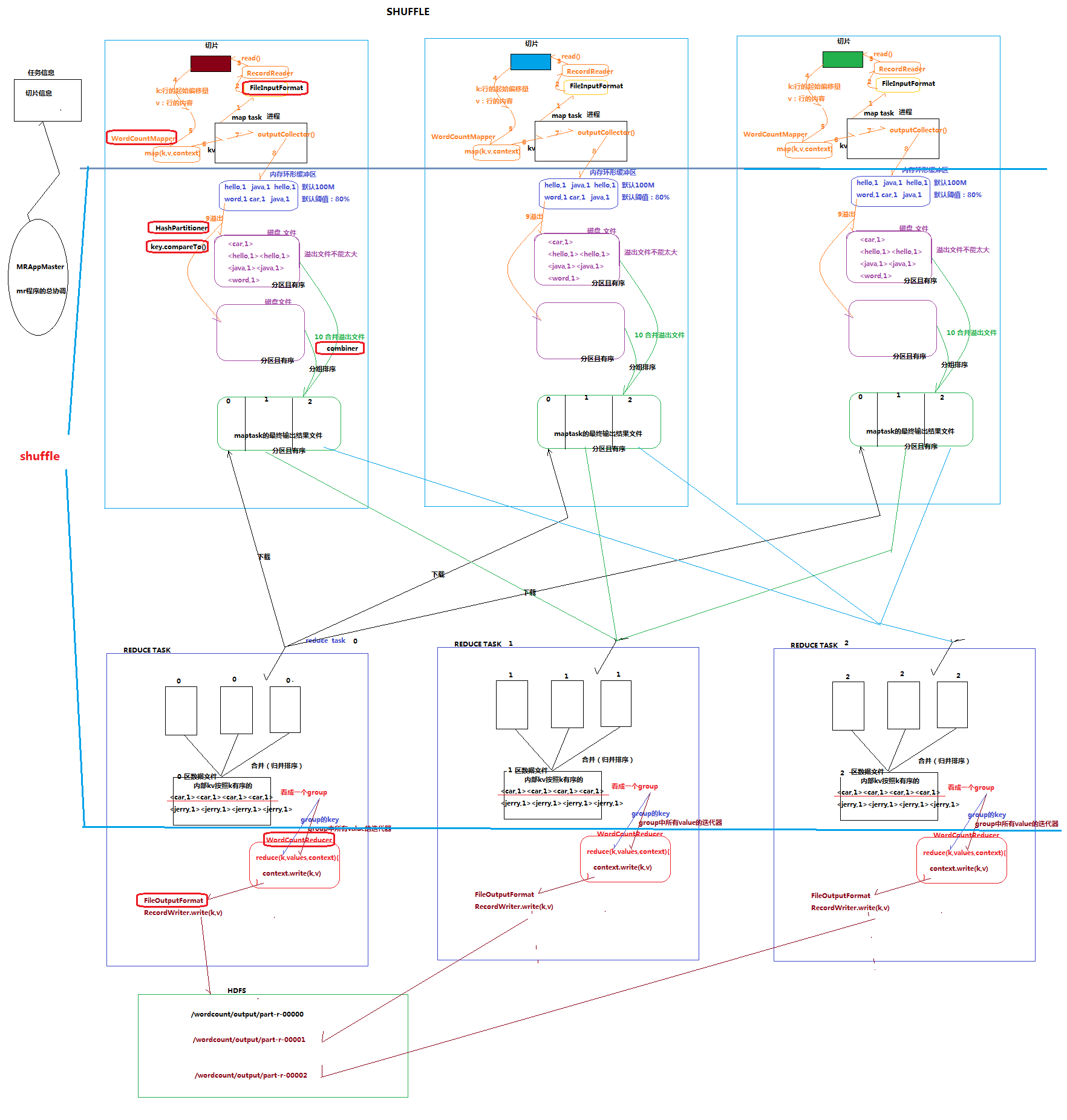
##### 详细流程

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

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

缓冲区的大小可以通过参数调整, 参数：io.sort.mb 默认100M

##### 详细流程示意图



### MAPREDUCE中的序列化

#### 概述

Java的序列化是一个重量级序列化框架（Serializable），一个对象被序列化后，会附带很多额外的信息（各种校验信息，header，继承体系。。。。），不便于在网络中高效传输；

所以，hadoop自己开发了一套序列化机制（Writable），精简，高效

#### Jdk序列化和MR序列化之间的比较

简单代码验证两种序列化机制的差别：

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

#### 自定义对象实现MR中的序列化接口

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

public class FlowBean implements WritableComparable<FlowBean>

需要自己实现的方法是：

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

### MAPREDUCE实践篇（2）

|  |  |
| --- | --- |
| MAPREDUCE  编程案例 | 流量日志统计案例 |
| 社交好友数据挖掘 |
| MR创建倒排索引 |
| 车联网数据清洗、去重、加密 |
| 小文件合并案例—自定义INPUTFORMAT |
| 数据分类输出—自定义OUTPUTFORMAT |
| TOPN算法实现—自定义Groupingcomparator |
| MAP端join算法实现—使用DistributedCache |
| MAPREDUCE  其他补充 | Mapreduce的计数器应用 |
| Mapreduce多job串联 |
| Mapreduce 参数优化 |

#### 流量统计相关需求

1. 对流量日志中的用户统计总上、下行流量

技术点： 自定义javaBean用来在mapreduce中充当value

注意： javaBean要实现Writable接口，实现两个方法

|  |
| --- |
| //序列化，将对象的字段信息写入输出流  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upflow);  out.writeLong(downflow);  out.writeLong(sumflow);  }  //反序列化，从输入流中读取各个字段信息  @Override  public void readFields(DataInput in) throws IOException {  upflow = in.readLong();  downflow = in.readLong();  sumflow = in.readLong();  } |

1. 统计流量且按照流量大小倒序排序

技术点：这种需求，用一个mapreduce -job 不好实现，需要两个mapreduce -job

第一个job负责流量统计，跟上题相同

第二个job读入第一个job的输出，然后做排序

要将flowBean作为map的key输出，这样mapreduce就会自动排序

此时，flowBean要实现接口WritableComparable

要实现其中的compareTo()方法，方法中，我们可以定义倒序比较的逻辑

1. 统计流量且按照手机号的归属地，将结果数据输出到不同的省份文件中

技术点：自定义Partitioner

|  |
| --- |
| *@Override*  *public int getPartition(Text key, FlowBean value, int numPartitions) {*    *String prefix = key.toString().substring(0,3);*  *Integer partNum = pmap.get(prefix);*    *return (partNum==null?4:partNum);*  *}* |

自定义partition后，要根据自定义partitioner的逻辑设置相应数量的reduce task

|  |
| --- |
| job.setNumReduceTasks(5); |

注意：如果reduceTask的数量>= *getPartition的结果数 ，则会多产生几个空的输出文件part-r-000xx*

*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！！！*

*如果* reduceTask的数量=1，则不管mapTask端输出多少个分区文件，最终结果都交给这一个reduceTask，最终也就只会产生一个结果文件 part-r-00000

#### Mapreduce中的排序初步

##### 需求

对日志数据中的上下行流量信息汇总，并输出按照总流量倒序排序的结果

数据如下：

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

##### 分析

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

MR程序在处理数据的过程中会对数据排序(map输出的kv对传输到reduce之前，会排序)，排序的依据是map输出的key

所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到key中，让key实现接口：WritableComparable

然后重写key的compareTo方法

##### 实现

1. 自定义的bean

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

1. mapper 和 reducer

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

#### Mapreduce中的分区Partitioner

##### 需求

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

##### 分析

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

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

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

自定义一个CustomPartitioner继承抽象类：Partitioner

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

##### 实现

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

#### mapreduce数据压缩

##### 概述

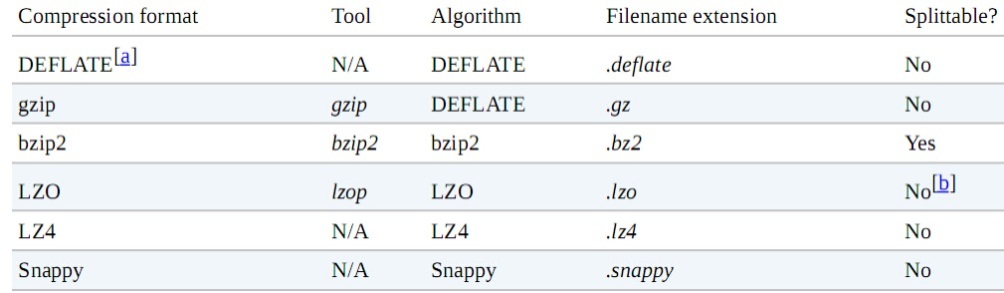
这是**mapreduce的一种优化策略：通过压缩编码对mapper或者reducer的输出进行压缩，以减少磁盘IO，**提高MR程序运行速度（但相应增加了cpu运算负担）

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

运算密集型的job，少用压缩

IO密集型的job，多用压缩

##### MR支持的压缩编码



##### Reducer输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.output.fileoutputformat.compress=false

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

mapreduce.output.fileoutputformat.compress.type=RECORD

2、在代码中设置

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

##### Mapper输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.map.output.compress=false

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

2、在代码中设置：

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

##### 压缩文件的读取

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

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

#### MAPREDUCE中的Combine

Combiner的使用要非常谨慎

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

所以：combiner使用的原则是：有或没有都不能影响业务逻辑

1. combiner是MR程序中Mapper和Reducer之外的一种组件
2. combiner组件的父类就是Reducer
3. combiner和reducer的区别在于运行的位置：

Combiner是在每一个maptask所在的节点运行

Reducer是接收全局所有Mapper的输出结果；

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

具体实现步骤：

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

(5) combiner能够应用的前提是不能影响最终的业务逻辑

而且，combiner的输出kv应该跟reducer的输入kv类型要对应起来

### 倒排索引建立

需求：有大量的文本（文档、网页），需要建立搜索索引

### 社交粉丝数据分析

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

A:B,C,D,F,E,O

B:A,C,E,K

C:F,A,D,I

D:A,E,F,L

E:B,C,D,M,L

F:A,B,C,D,E,O,M

G:A,C,D,E,F

H:A,C,D,E,O

I:A,O

J:B,O

K:A,C,D

L:D,E,F

M:E,F,G

O:A,H,I,J

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？

解题思路：

|  |
| --- |
| 第一步  map  读一行 A:B,C,D,F,E,O  输出 <B,A><C,A><D,A><F,A><E,A><O,A>  在读一行 B:A,C,E,K  输出 <A,B><C,B><E,B><K,B>  REDUCE  拿到的数据比如<C,A><C,B><C,E><C,F><C,G>......  输出：  <A-B,C>  <A-E,C>  <A-F,C>  <A-G,C>  <B-E,C>  <B-F,C>.....  第二步  map  读入一行<A-B,C>  直接输出<A-B,C>  reduce  读入数据 <A-B,C><A-B,F><A-B,G>.......  输出： A-B C,F,G,..... |

扩展：求互粉的人！！！！

### 更多MapReduce编程案例

#### reduce端join算法实现

1、需求：

订单数据表t\_order：

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

商品信息表t\_product

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

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

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

2、实现机制：

通过将关联的条件作为map输出的key，将两表满足join条件的数据并携带数据所来源的文件信息，发往同一个reduce task，在reduce中进行数据的串联

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

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

解决方案： map端join实现方式

#### map端join算法实现

1、原理阐述

适用于关联表中有小表的情形；

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

2、实现示例

--先在mapper类中预先定义好小表，进行join

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

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

#### TOPN--自定义GroupingComparator

##### 需求

有如下订单数据

|  |  |  |
| --- | --- | --- |
| 订单id | 商品id | 成交金额 |
| Order\_0000001 | Pdt\_01 | 222.8 |
| Order\_0000001 | Pdt\_05 | 25.8 |
| Order\_0000002 | Pdt\_03 | 522.8 |
| Order\_0000002 | Pdt\_04 | 122.4 |
| Order\_0000002 | Pdt\_05 | 722.4 |
| Order\_0000003 | Pdt\_01 | 222.8 |

现在需要求出每一个订单中成交金额最大的一笔交易

##### 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分区，按照金额排序，发送到reduce

2、在reduce端利用groupingcomparator将订单id相同的kv聚合成组，然后取第一个即是最大值

##### 实现

自定义groupingcomparator

|  |
| --- |
| /\*\*  \* 用于控制shuffle过程中reduce端对kv对的聚合逻辑  \* @author duanhaitao@itcast.cn  \*  \*/  public class ItemidGroupingComparator extends WritableComparator {  protected ItemidGroupingComparator() {  super(OrderBean.class, true);  }    @Override  public int compare(WritableComparable a, WritableComparable b) {  OrderBean abean = (OrderBean) a;  OrderBean bbean = (OrderBean) b;    //将item\_id相同的bean都视为相同，从而聚合为一组  return abean.getItemid().compareTo(bbean.getItemid());  }  } |

定义订单信息bean

|  |
| --- |
| /\*\*  \* 订单信息bean，实现hadoop的序列化机制  \* @author duanhaitao@itcast.cn  \*  \*/  public class OrderBean implements WritableComparable<OrderBean>{  private Text itemid;  private DoubleWritable amount;  public OrderBean() {  }  public OrderBean(Text itemid, DoubleWritable amount) {  set(itemid, amount);  }  public void set(Text itemid, DoubleWritable amount) {  this.itemid = itemid;  this.amount = amount;  }  public Text getItemid() {  return itemid;  }  public DoubleWritable getAmount() {  return amount;  }  @Override  public int compareTo(OrderBean o) {  int cmp = this.itemid.compareTo(o.getItemid());  if (cmp == 0) {  cmp = -this.amount.compareTo(o.getAmount());  }  return cmp;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(itemid.toString());  out.writeDouble(amount.get());    }  @Override  public void readFields(DataInput in) throws IOException {  String readUTF = in.readUTF();  double readDouble = in.readDouble();    this.itemid = new Text(readUTF);  this.amount= new DoubleWritable(readDouble);  }  @Override  public String toString() {  return itemid.toString() + "\t" + amount.get();  }  } |

编写mapreduce处理流程

|  |
| --- |
| /\*\*  \* 利用secondarysort机制输出每种item订单金额最大的记录  \* @author duanhaitao@itcast.cn  \*  \*/  public class SecondarySort {    static class SecondarySortMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable>{    OrderBean bean = new OrderBean();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = StringUtils.split(line, "\t");    bean.set(new Text(fields[0]), new DoubleWritable(Double.parseDouble(fields[1])));    context.write(bean, NullWritable.get());    }    }    static class SecondarySortReducer extends Reducer<OrderBean, NullWritable, OrderBean, NullWritable>{      //在设置了groupingcomparator以后，这里收到的kv数据 就是： <1001 87.6>,null <1001 76.5>,null ....  //此时，reduce方法中的参数key就是上述kv组中的第一个kv的key：<1001 87.6>  //要输出同一个item的所有订单中最大金额的那一个，就只要输出这个key  @Override  protected void reduce(OrderBean key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {  context.write(key, NullWritable.get());  }  }      public static void main(String[] args) throws Exception {    Configuration conf = new Configuration();  Job job = Job.getInstance(conf);    job.setJarByClass(SecondarySort.class);    job.setMapperClass(SecondarySortMapper.class);  job.setReducerClass(SecondarySortReducer.class);      job.setOutputKeyClass(OrderBean.class);  job.setOutputValueClass(NullWritable.class);    FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  //指定shuffle所使用的GroupingComparator类  job.setGroupingComparatorClass(ItemidGroupingComparator.class);  //指定shuffle所使用的partitioner类  job.setPartitionerClass(ItemIdPartitioner.class);    job.setNumReduceTasks(3);    job.waitForCompletion(true);    }  } |

#### 自定义outputFormat

##### 需求

现有一些原始日志需要做增强解析处理，流程：

1. 从原始日志文件中读取数据
2. 根据日志中的一个URL字段到外部知识库中获取信息增强到原始日志
3. 如果成功增强，则输出到增强结果目录；如果增强失败，则抽取原始数据中URL字段输出到待爬清单目录

##### 分析

程序的关键点是要在一个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

##### 实现

实现要点：

1. 在mapreduce中访问外部资源
2. 自定义outputformat，改写其中的recordwriter，改写具体输出数据的方法write()

代码实现如下：

数据库获取数据的工具

|  |
| --- |
| public class DBLoader {  public static void dbLoader(HashMap<String, String> ruleMap) {  Connection conn = null;  Statement st = null;  ResultSet res = null;    try {  Class.forName("com.mysql.jdbc.Driver");  conn = DriverManager.getConnection("jdbc:mysql://hdp-node01:3306/urlknowledge", "root", "root");  st = conn.createStatement();  res = st.executeQuery("select url,content from urlcontent");  while (res.next()) {  ruleMap.put(res.getString(1), res.getString(2));  }  } catch (Exception e) {  e.printStackTrace();    } finally {  try{  if(res!=null){  res.close();  }  if(st!=null){  st.close();  }  if(conn!=null){  conn.close();  }  }catch(Exception e){  e.printStackTrace();  }  }  }      public static void main(String[] args) {  DBLoader db = new DBLoader();  HashMap<String, String> map = new HashMap<String,String>();  db.dbLoader(map);  System.out.println(map.size());  }  } |

自定义一个outputformat

|  |
| --- |
| public class LogEnhancerOutputFormat extends FileOutputFormat<Text, NullWritable>{    @Override  public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext context) throws IOException, InterruptedException {  FileSystem fs = FileSystem.get(context.getConfiguration());  Path enhancePath = new Path("hdfs://hdp-node01:9000/flow/enhancelog/enhanced.log");  Path toCrawlPath = new Path("hdfs://hdp-node01:9000/flow/tocrawl/tocrawl.log");    FSDataOutputStream enhanceOut = fs.create(enhancePath);  FSDataOutputStream toCrawlOut = fs.create(toCrawlPath);      return new MyRecordWriter(enhanceOut,toCrawlOut);  }        static class MyRecordWriter extends RecordWriter<Text, NullWritable>{    FSDataOutputStream enhanceOut = null;  FSDataOutputStream toCrawlOut = null;    public MyRecordWriter(FSDataOutputStream enhanceOut, FSDataOutputStream toCrawlOut) {  this.enhanceOut = enhanceOut;  this.toCrawlOut = toCrawlOut;  }  @Override  public void write(Text key, NullWritable value) throws IOException, InterruptedException {    //有了数据，你来负责写到目的地 —— hdfs  //判断，进来内容如果是带tocrawl的，就往待爬清单输出流中写 toCrawlOut  if(key.toString().contains("tocrawl")){  toCrawlOut.write(key.toString().getBytes());  }else{  enhanceOut.write(key.toString().getBytes());  }    }  @Override  public void close(TaskAttemptContext context) throws IOException, InterruptedException {    if(toCrawlOut!=null){  toCrawlOut.close();  }  if(enhanceOut!=null){  enhanceOut.close();  }    }      }  } |

开发mapreduce处理流程

|  |
| --- |
| /\*\*  \* 这个程序是对每个小时不断产生的用户上网记录日志进行增强(将日志中的url所指向的网页内容分析结果信息追加到每一行原始日志后面)  \*  \* @author  \*  \*/  public class LogEnhancer {  static class LogEnhancerMapper extends Mapper<LongWritable, Text, Text, NullWritable> {  HashMap<String, String> knowledgeMap = new HashMap<String, String>();  /\*\*  \* maptask在初始化时会先调用setup方法一次 利用这个机制，将外部的知识库加载到maptask执行的机器内存中  \*/  @Override  protected void setup(org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException {  DBLoader.dbLoader(knowledgeMap);  }  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = StringUtils.split(line, "\t");  try {  String url = fields[26];  // 对这一行日志中的url去知识库中查找内容分析信息  String content = knowledgeMap.get(url);  // 根据内容信息匹配的结果，来构造两种输出结果  String result = "";  if (null == content) {  // 输往待爬清单的内容  result = url + "\t" + "tocrawl\n";  } else {  // 输往增强日志的内容  result = line + "\t" + content + "\n";  }  context.write(new Text(result), NullWritable.get());  } catch (Exception e) {  }  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(LogEnhancer.class);  job.setMapperClass(LogEnhancerMapper.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(LogEnhancerOutputFormat.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]));  job.waitForCompletion(true);  System.exit(0);  }  } |

### MapReduce与YARN

#### YARN概述

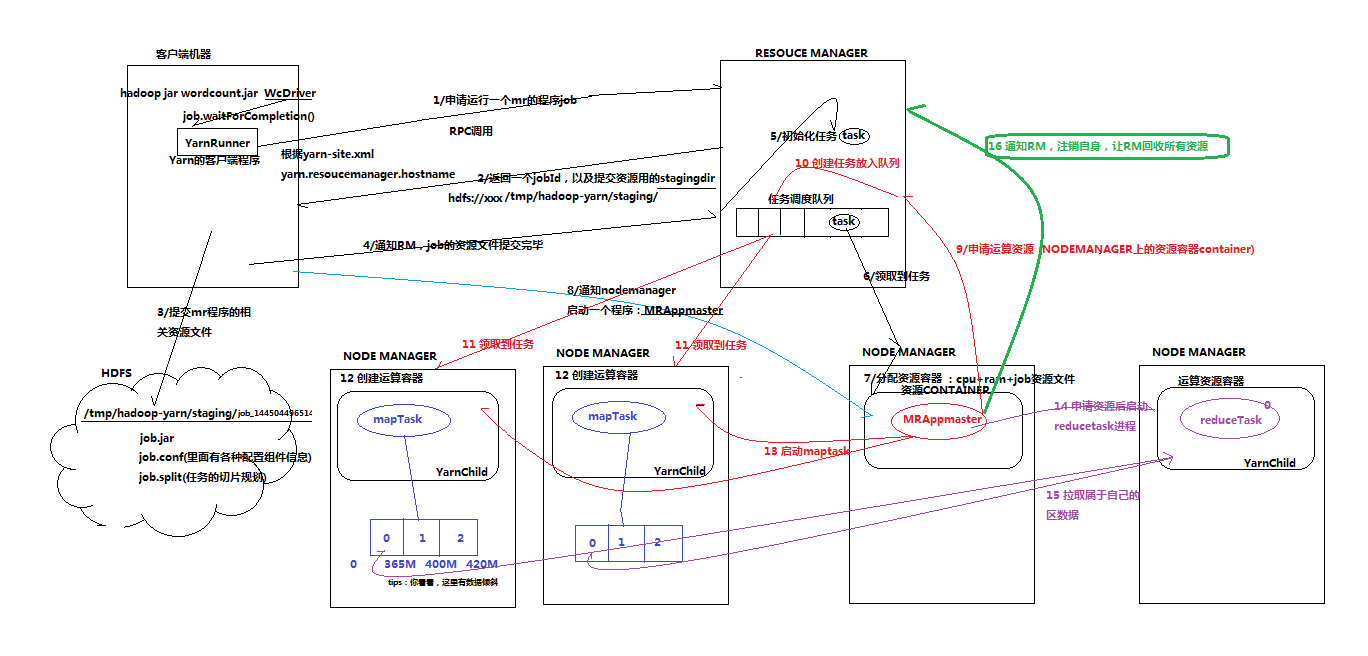
Yarn是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而mapreduce等运算程序则相当于运行于操作系统之上的应用程序

#### YARN的重要概念

1. yarn并不清楚用户提交的程序的运行机制
2. yarn只提供运算资源的调度（用户程序向yarn申请资源，yarn就负责分配资源）
3. yarn中的主管角色叫ResourceManager
4. yarn中具体提供运算资源的角色叫NodeManager
5. 这样一来，yarn其实就与运行的用户程序完全解耦，就意味着yarn上可以运行各种类型的分布式运算程序（mapreduce只是其中的一种），比如mapreduce、storm程序，spark程序，tez ……
6. 所以，spark、storm等运算框架都可以整合在yarn上运行，只要他们各自的框架中有符合yarn规范的资源请求机制即可
7. Yarn就成为一个通用的资源调度平台，从此，企业中以前存在的各种运算集群都可以整合在一个物理集群上，提高资源利用率，方便数据共享

#### Yarn中运行运算程序的示例

mapreduce程序的调度过程，如下图



### mapreduce参数优化

MapReduce重要配置参数

#### 资源相关参数

//以下参数是在用户自己的mr应用程序中配置就可以生效

(1) mapreduce.map.memory.mb: 一个Map Task可使用的资源上限（单位:MB），默认为1024。如果Map Task实际使用的资源量超过该值，则会被强制杀死。

(2) mapreduce.reduce.memory.mb: 一个Reduce Task可使用的资源上限（单位:MB），默认为1024。如果Reduce Task实际使用的资源量超过该值，则会被强制杀死。

(3) mapreduce.map.cpu.vcores: 每个Map task可使用的最多cpu core数目, 默认值: 1

(4) mapreduce.reduce.cpu.vcores: 每个Reduce task可使用的最多cpu core数目, 默认值: 1

//shuffle性能优化的关键参数，应在yarn启动之前就配置好

1. mapreduce.task.io.sort.mb 100 //shuffle的环形缓冲区大小，默认100m
2. mapreduce.map.sort.spill.percent 0.8 //环形缓冲区溢出的阈值，默认80%

(5) mapreduce.map.java.opts: Map Task的JVM参数，你可以在此配置默认的java heap size等参数, e.g.

“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc” （@taskid@会被Hadoop框架自动换为相应的taskid）, 默认值: “”

(6) mapreduce.reduce.java.opts: Reduce Task的JVM参数，你可以在此配置默认的java heap size等参数, e.g.

“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”, 默认值: “”

//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

(7) yarn.scheduler.minimum-allocation-mb 1024 给应用程序container分配的最小内存

(8) yarn.scheduler.maximum-allocation-mb 8192 给应用程序container分配的最大内存

(9) yarn.scheduler.minimum-allocation-vcores 1

(10)yarn.scheduler.maximum-allocation-vcores 32

(11)yarn.nodemanager.resource.memory-mb 8192 每台NodeManager最大可用内存

yarn.nodemanager.resource.cpu-vcores 8 每台NodeManager最大可用cpu核数

#### 容错相关参数

(1) 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。

(2) 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。

(3) mapreduce.map.failures.maxpercent: 当失败的Map Task失败比例超过该值为，整个作业则失败，默认值为0. 如果你的应用程序允许丢弃部分输入数据，则该该值设为一个大于0的值，比如5，表示如果有低于5%的Map Task失败（如果一个Map Task重试次数超过mapreduce.map.maxattempts，则认为这个Map Task失败，其对应的输入数据将不会产生任何结果），整个作业扔认为成功。

(4) mapreduce.reduce.failures.maxpercent: 当失败的Reduce Task失败比例超过该值为，整个作业则失败，默认值为0.

(5) 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，也许永远会卡主，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300000。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.”。

#### 本地运行mapreduce 作业

设置以下几个参数:

mapreduce.framework.name=local

mapreduce.jobtracker.address=local

fs.defaultFS=local

#### 效率和稳定性相关参数

(1) mapreduce.map.speculative: 是否为Map Task打开推测执行机制，默认为false

(2) mapreduce.reduce.speculative: 是否为Reduce Task打开推测执行机制，默认为false

(3) mapreduce.job.user.classpath.first & mapreduce.task.classpath.user.precedence：当同一个class同时出现在用户jar包和hadoop jar中时，优先使用哪个jar包中的class，默认为false，表示优先使用hadoop jar中的class。

(4) mapreduce.input.fileinputformat.split.minsize: FileInputFormat做切片时的最小切片大小，(5)mapreduce.input.fileinputformat.split.maxsize: FileInputFormat做切片时的最大切片大小

(切片的默认大小就等于blocksize，即 134217728)

### Mapreduce的其他补充

#### 计数器应用

在实际生产代码中，常常需要将数据处理过程中遇到的不合规数据行进行全局计数，类似这种需求可以借助mapreduce框架中提供的全局计数器来实现

示例代码如下：

|  |
| --- |
| public class MultiOutputs {  //通过枚举形式定义自定义计数器  enum MyCounter{MALFORORMED,NORMAL}  static class CommaMapper extends Mapper<LongWritable, Text, Text, LongWritable> {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String[] words = value.toString().split(",");  for (String word : words) {  context.write(new Text(word), new LongWritable(1));  }  //对枚举定义的自定义计数器加1  context.getCounter(MyCounter.MALFORORMED).increment(1);  //通过动态设置自定义计数器加1  context.getCounter("counterGroupa", "countera").increment(1);  }  } |

#### 多job串联

一个稍复杂点的处理逻辑往往需要多个mapreduce程序串联处理，多job的串联可以借助mapreduce框架的JobControl实现

示例代码：

|  |
| --- |
| ControlledJob cJob1 = new ControlledJob(job1.getConfiguration());  ControlledJob cJob2 = new ControlledJob(job2.getConfiguration());  ControlledJob cJob3 = new ControlledJob(job3.getConfiguration());    cJob1.setJob(job1);  cJob2.setJob(job2);  cJob3.setJob(job3);  // 设置作业依赖关系  cJob2.addDependingJob(cJob1);  cJob3.addDependingJob(cJob2);    JobControl jobControl = new JobControl("RecommendationJob");  jobControl.addJob(cJob1);  jobControl.addJob(cJob2);  jobControl.addJob(cJob3);      // 新建一个线程来运行已加入JobControl中的作业，开始进程并等待结束  Thread jobControlThread = new Thread(jobControl);  jobControlThread.start();  while (!jobControl.allFinished()) {  Thread.sleep(500);  }  jobControl.stop();    return 0; |
|  |

### mapreduce数据压缩

#### 概述

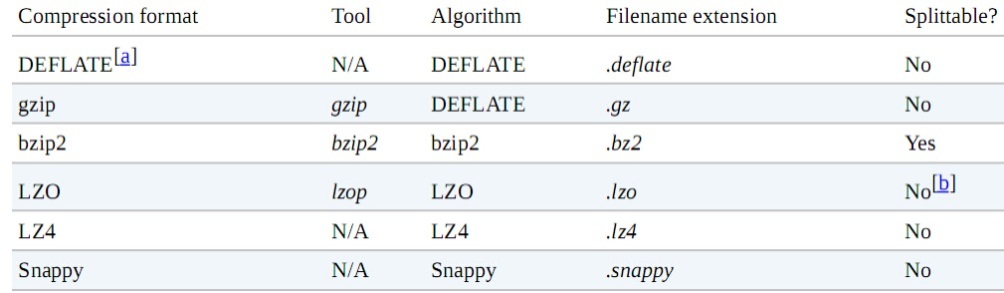
这是**mapreduce的一种优化策略：通过压缩编码对mapper或者reducer的输出进行压缩，以减少磁盘IO，**提高MR程序运行速度（但相应增加了cpu运算负担）

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

运算密集型的job，少用压缩

IO密集型的job，多用压缩

#### MR支持的压缩编码



#### Reducer输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.output.fileoutputformat.compress=true

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

或者用gzip org.apache.hadoop.io.compress.GzipCodec

mapreduce.output.fileoutputformat.compress.type=RECORD

2、在代码中设置

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

#### Mapper输出压缩

在配置参数或在代码中都可以设置reduce的输出压缩

1、在配置参数中设置

mapreduce.map.output.compress=true

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

2、在代码中设置：

|  |
| --- |
| Conf.set(“mapreduce.map.output.compress”,” true”)  conf.setBoolean(Job.MAP\_OUTPUT\_COMPRESS, true);  conf.setClass(Job.MAP\_OUTPUT\_COMPRESS\_CODEC, GzipCodec.class, CompressionCodec.class); |

#### 压缩文件的读取(源码截取)

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

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

## MapReduce加强

|  |  |
| --- | --- |
| MAPREDUCE  编程案例 | 流量日志统计案例 |
| 社交好友数据挖掘 |
| MR创建倒排索引 |
| 车联网数据清洗、去重、加密 |
| 小文件合并案例—自定义INPUTFORMAT |
| 数据分类输出—自定义OUTPUTFORMAT |
| TOPN算法实现—自定义Groupingcomparator |
| MAP端join算法实现—使用DistributedCache |
| MAPREDUCE  其他补充 | Mapreduce的计数器应用 |
| Mapreduce多job串联 |
| Mapreduce 参数优化 |

## Hadoop的HA机制

前言：正式引入HA机制是从hadoop2.0开始，之前的版本中没有HA机制

### HA的运作机制

（1）hadoop-HA集群运作机制介绍

所谓HA，即高可用（7\*24小时不中断服务）

实现高可用最关键的是消除单点故障(与高可靠区别)

hadoop-ha严格来说应该分成各个组件的HA机制——**HDFS的HA**、**YARN的HA**

（2）HDFS的HA机制详解

通过双namenode消除单点故障

双namenode协调工作的要点：

A、元数据管理方式需要改变：

内存中各自保存一份元数据

Edits日志只能有一份，只有Active状态的namenode节点可以做写操作,并且对外 提供服务

两个namenode都可以读取edits,standby只负责同步namenode状态

共享的edits放在一个共享存储中管理（qjournal和NFS两个主流实现）

B、需要一个状态管理功能模块

实现了一个zkfailover，常驻在每一个namenode所在的节点

每一个zkfailover负责监控自己所在namenode节点，利用zk进行状态标识

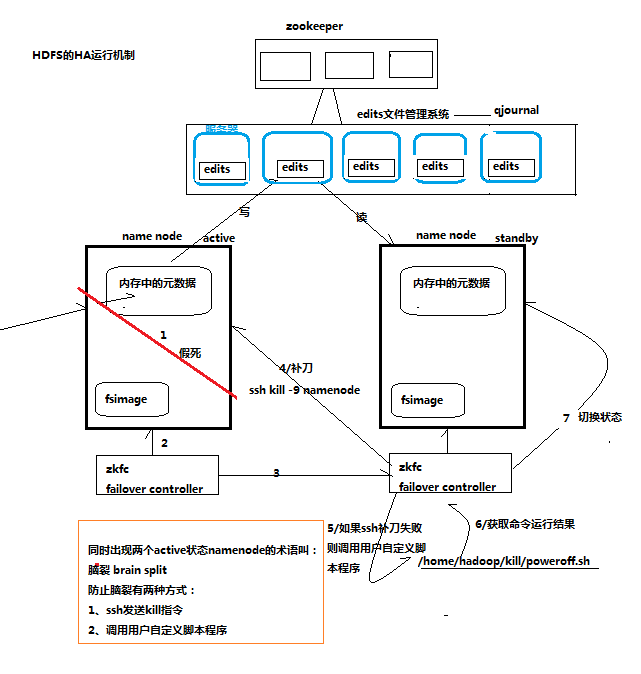
当需要进行状态切换时，由zkfailover来负责切换

切换时需要防止brain split现象的发生

（2）YARN的HA机制详解

机制直接写在ResourceManager里面,也与zookeeper连接,死掉就直接切换,比HDFS简单

### HDFS-HA图解：

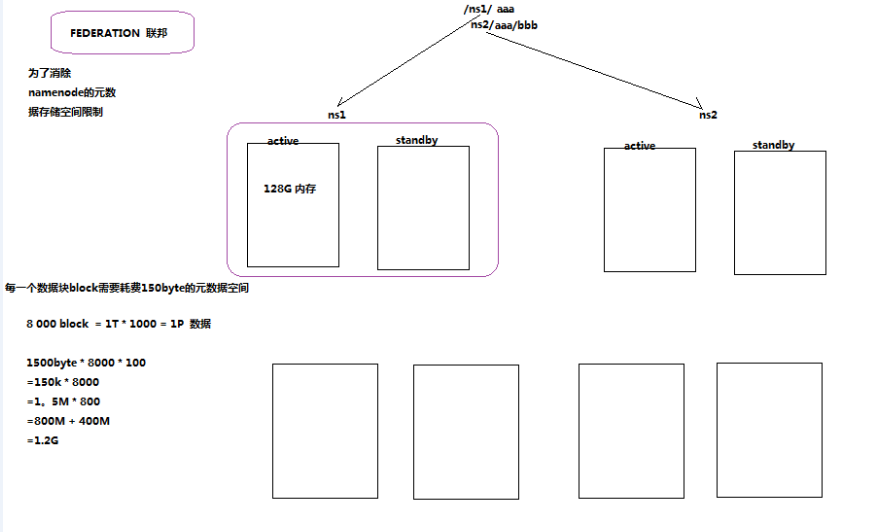


1.状态监控切换问题:写一个failover controller,将状态数据存到zk里面,用zk解决(zk集群用于故障转移)

2.元数据同步管理的问题:有一个文件服务器,namenode即写本机也写在edits文件管理系统中;并且文件系统也写成分布式的,叫做qjournal,standby定期会从文件服务器拉取数据,然后和本地的数据合并(checkpoint).

3.防止脑裂的问题:在jvm做fullgc的时候,叫做stop the world,standby感知到active死掉,就自己切换为active,但是gc完后,又感知到另外一个active的namenode,此时就懵逼了,解决办法:1.ssh发送kill命令,可能会失败;2.最终解决办法,调用用户自定义脚本程序(开发人员自己写),返回结果成功,就切换,否则不切换.(一般不会用到此方法)

### hdfs的联邦机制



FORMAT -CLUSTERID DATATEST,格式化的时候要一样

上层抽象了一个viewfs://aaa

此方式很少公司用

### HA集群的安装部署

#### 集群节点规划

集群部署节点角色的规划（10节点）：

|  |
| --- |
| server01 namenode zkfc > start-dfs.sh  server02 namenode zkfc  server03 resourcemanager > start-yarn.sh  server04 resourcemanager  server05 datanode nodemanager  server06 datanode nodemanager  server07 datanode nodemanager  server08 journal node zookeeper datanode nodemanager  server09 journal node zookeeper datanode nodemanager  server10 journal node zookeeper datanode nodemanager |

集群部署节点角色的规划（8节点）：

|  |
| --- |
| server01 namenode zkfc > start-dfs.sh  server02 namenode zkfc  server03 resourcemanager > start-yarn.sh  server04 resourcemanager  server05 datanode nodemanager  server06 datanode nodemanager zookeeper journal node  server07 datanode nodemanager zookeeper journal node  server08 datanode nodemanager zookeeper journal node |

集群部署节点角色的规划（3节点）

|  |
| --- |
| server01 namenode resourcemanager zkfc nodemanager datanode zookeeper journal node  server02 namenode resourcemanager zkfc nodemanager datanode zookeeper journal node  server03 datanode nodemanager zookeeper journal node |

#### 环境准备

>> linux系统准备

ip地址配置

hostname配置

hosts映射配置(拷贝)

防火墙关闭

init启动级别修改3

sudoers加入hadoop用户

ssh免密登陆配置

>> java环境的配置

上传jdk，解压，修改/etc/profile

>> zookeeper集群的部署

#### 配置文件

core-site.xml

|  |
| --- |
| <configuration>  <!-- 指定hdfs的nameservice为ns1 -->  <property>  <name>fs.defaultFS</name>  <value>hdfs://ns1/</value>  </property>  <!-- 指定hadoop临时目录 -->  <property>  <name>hadoop.tmp.dir</name>  <value>/home/hadoop/app/hadoop-2.4.1/tmp</value>  </property>  <!-- 指定zookeeper地址 -->  <property>  <name>ha.zookeeper.quorum</name>  <value>mini05:2181,mini06:2181,mini07:2181</value>  </property>  </configuration> |

hdfs-site.xml

|  |
| --- |
| <configuration>  <!--指定hdfs的nameservice为ns1，需要和core-site.xml中的保持一致 -->  <property>  <name>dfs.nameservices</name>  <value>ns1</value>  </property>  <!-- ns1下面有两个NameNode，分别是nn1，nn2(不是主机名,逻辑代号) -->  <property>  <name>dfs.ha.namenodes.ns1</name>  <value>nn1,nn2</value>  </property>  <!-- nn1的RPC通信地址 -->  <property>  <name>dfs.namenode.rpc-address.ns1.nn1</name>  <value>mini01:9000</value>  </property>  <!-- nn1的http通信地址 -->  <property>  <name>dfs.namenode.http-address.ns1.nn1</name>  <value>mini01:50070</value>  </property>  <!-- nn2的RPC通信地址 -->  <property>  <name>dfs.namenode.rpc-address.ns1.nn2</name>  <value>mini02:9000</value>  </property>  <!-- nn2的http通信地址 -->  <property>  <name>dfs.namenode.http-address.ns1.nn2</name>  <value>mini02:50070</value>  </property>  <!-- 指定NameNode的edits元数据在JournalNode上的存放位置 -->  <property>  <name>dfs.namenode.shared.edits.dir</name>  <value>qjournal://mini05:8485;mini06:8485;mini07:8485/ns1</value>  </property>  <!-- 指定JournalNode在本地磁盘存放数据的位置 -->  <property>  <name>dfs.journalnode.edits.dir</name>  <value>/home/hadoop/app/hadoop-2.6.4/journaldata</value>  </property>  <!-- 开启NameNode失败自动切换 -->  <property>  <name>dfs.ha.automatic-failover.enabled</name>  <value>true</value>  </property>  <!-- 配置失败自动切换实现方式,不同名称服务的有不同自动切换的服务 -->  <property>  <name>dfs.client.failover.proxy.provider.ns1</name>  <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>  </property>  <!-- 配置隔离机制(脑裂)方法，多个机制用换行分割，即每个机制暂用一行-->  <property>  <name>dfs.ha.fencing.methods</name>  <value>  sshfence  shell(/bin/true)  </value>  </property>  <!-- 使用sshfence隔离机制时需要ssh免登陆 -->  <property>  <name>dfs.ha.fencing.ssh.private-key-files</name>  <value>/home/hadoop/.ssh/id\_rsa</value>  </property>  <!-- 配置sshfence隔离机制超时时间 (ms)-->  <property>  <name>dfs.ha.fencing.ssh.connect-timeout</name>  <value>30000</value>  </property>  /configuration> |

mapred-site.xml

|  |
| --- |
| <configuration>  <!-- 指定mr框架为yarn方式 -->  <property>  <name>mapreduce.framework.name</name>  <value>yarn</value>  </property>  </configuration> |

yarn-site.xml

|  |
| --- |
| <configuration>  <!-- 开启RM高可用 -->  <property>  <name>yarn.resourcemanager.ha.enabled</name>  <value>true</value>  </property>  <!-- 指定RM的cluster id -->  <property>  <name>yarn.resourcemanager.cluster-id</name>  <value>yrc</value>  </property>  <!-- 指定RM的名字 -->  <property>  <name>yarn.resourcemanager.ha.rm-ids</name>  <value>rm1,rm2</value>  </property>  <!-- 分别指定RM的地址 -->  <property>  <name>yarn.resourcemanager.hostname.rm1</name>  <value>mini03</value>  </property>  <property>  <name>yarn.resourcemanager.hostname.rm2</name>  <value>mini04</value>  </property>  <!-- 指定zk集群地址 -->  <property>  <name>yarn.resourcemanager.zk-address</name>  <value>mini05:2181,mini06:2181,mini07:2181</value>  </property>  <property>  <name>yarn.nodemanager.aux-services</name>  <value>mapreduce\_shuffle</value>  </property>  </configuration> |

#### slaves配置

slaves是指定子节点的位置，因为要在mini1上启动HDFS、在mini3启动yarn，所以mini1上的slaves文件指定的是datanode的位置，mini3上的slaves文件指定的是nodemanager的位置

mini5

mini6

mini7

#### 配置免密登录

1.首先要配置mini1到mini2、mini3、mini4、mini5、mini6、mini7的免密码登陆

2.配置mini3到mini4、mini5、mini6、mini7的免密码登陆

!!注意：两个namenode之间要配置ssh免密码登陆，别忘了配置mini1到mini2的免登陆

在mini2上生产一对钥匙

ssh-keygen -t rsa

ssh-coyp-id -i mini1

#### 拷贝到其他机器

scp -r /hadoop/ mini2:$PWD

其他机器如这样

#### 启动顺序

1.启动zookeeper集群（分别在hadoop05、hadoop06、tcast07上启动zk）

cd /hadoop/zookeeper-3.4.5/bin/

./zkServer.sh start

#查看状态：一个leader，两个follower

./zkServer.sh status

2.手动启动journalnode（分别在在hadoop05、hadoop06、hadoop07上执行）

cd /hadoop/hadoop-2.6.4

sbin/hadoop-daemon.sh start journalnode

#运行jps命令检验，hadoop05、hadoop06、hadoop07上多了JournalNode进程

3.格式化namenode

#在hadoop00上执行命令:

hdfs namenode -format

#格式化后会在根据core-site.xml中的hadoop.tmp.dir配置生成个文件，这里我配置的是/hadoop/hadoop-2.6.4/tmp，然后

将/hadoop/hadoop-2.6.4/tmp拷贝到hadoop02的/hadoop/hadoop-2.6.4/下。

scp -r tmp/ hadoop02:/home/hadoop/app/hadoop-2.6.4/

##也可以这样，建议hdfs namenode -bootstrapStandby

4.格式化ZKFC(在mini1上执行即可)

hdfs zkfc -formatZK

5.启动HDFS(在mini1上执行)

sbin/start-dfs.sh

6.启动YARN(#####注意#####：是在mini3上执行start-yarn.sh，把namenode和resourcemanager分开是因为性能问题，因为他们都要占用大量资源，所以把他们分开了，他们分开了就要分别在不同的机器上启动)

sbin/start-yarn.sh yarn-daemon.sh start resourcemanager

start-yarn.sh不会ssh到4上启动,不想start-dfs.sh一样,所以在mini4上用yarn-daemon.sh启动resourcemanager.

#### 集群运维测试

1.HA集群中两个namenode状态的管理命令

|  |
| --- |
| [root@mini2 hadoop-2.6.4]# bin/hdfs haadmin  Usage: DFSHAAdmin [-ns <nameserviceId>]  [-transitionToActive <serviceId> [--forceactive]]  [-transitionToStandby <serviceId>]  [-failover [--forcefence] [--forceactive] <serviceId> <serviceId>]  [-getServiceState <serviceId>]  [-checkHealth <serviceId>]  [-help <command>] |

示例： 切换nn2为active

bin/hdfs haadmin -transitionToActive nn2 --forcemanual

2.验证HDFS HA

首先向hdfs上传一个文件

hadoop fs -put /etc/profile /profile

hadoop fs -ls /

然后再kill掉active的NameNode

kill -9 <pid of NN>

通过浏览器访问：http://192.168.1.202:50070

NameNode 'hadoop02:9000' (active)

这个时候hadoop02上的NameNode变成了active

在执行命令：

hadoop fs -ls /

-rw-r--r-- 3 root supergroup 1926 2014-02-06 15:36 /profile

刚才上传的文件依然存在！！！

手动启动那个挂掉的NameNode

sbin/hadoop-daemon.sh start namenode

通过浏览器访问：http://192.168.1.201:50070

NameNode 'hadoop01:9000' (standby)

3.验证YARN：

运行一下hadoop提供的demo中的WordCount程序：

hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.1.jar wordcount /profile /out

resourcemanager不会切换

4.测试集群工作状态的一些指令 ：

bin/hdfs dfsadmin -report 查看hdfs的各节点状态信息

bin/hdfs haadmin -getServiceState nn1 获取一个namenode节点的HA状态

sbin/hadoop-daemon.sh start namenode 单独启动一个namenode进程

./hadoop-daemon.sh start zkfc 单独启动一个zkfc进程

hdfs haadmin -transactive

#### Datanode动态上下线

Datanode动态上下线很简单，步骤如下：

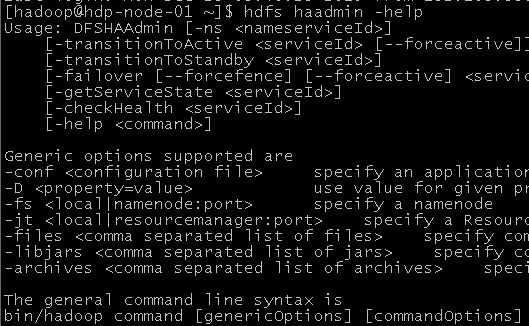
1. 准备一台服务器，设置好环境
2. 部署hadoop的安装包，并同步集群配置
3. 联网上线，新datanode会自动加入集群
4. 如果是一次增加大批datanode，还应该做集群负载重均衡

(start-balancer.sh -threshold 8 ##指定磁盘利用率，详情见下节 3)

#### Namenode状态切换管理

使用的命令上hdfs haadmin

可用 hdfs haadmin –help查看所有帮助信息



可以看到，状态操作的命令示例：

查看namenode工作状态

|  |
| --- |
| hdfs haadmin -getServiceState nn1 |

将standby状态namenode切换到active

|  |
| --- |
| hdfs haadmin –transitionToActive nn1 |

将active状态namenode切换到standby

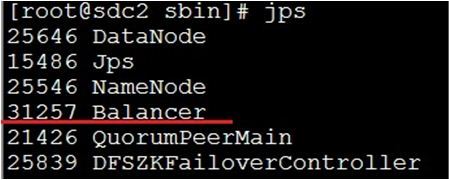
|  |
| --- |
| hdfs haadmin –transitionToStandby nn2 |

#### 数据块的balance

启动balancer的命令：

start-balancer.sh -threshold 8

运行之后，会有Balancer进程出现：



上述命令设置了Threshold为8%，那么执行balancer命令的时候，首先统计所有DataNode的磁盘利用率的均值，然后判断如果某一个DataNode的磁盘利用率超过这个均值Threshold，那么将会把这个DataNode的block转移到磁盘利用率低的DataNode，这对于新节点的加入来说十分有用。Threshold的值为1到100之间，不显示的进行参数设置的话，默认是10。

### HA下hdfs-api变化

客户端需要nameservice的配置信息，其他不变

|  |
| --- |
| /\*\*  \* 如果访问的是一个ha机制的集群  \* 则一定要把core-site.xml和hdfs-site.xml配置文件放在客户端程序的classpath下  \* 以让客户端能够理解hdfs://ns1/中 “ns1”是一个ha机制中的namenode对——nameservice  \* 以及知道ns1下具体的namenode通信地址  \* @author  \*  \*/  public class UploadFile {    public static void main(String[] args) throws Exception {    Configuration conf = new Configuration();  conf.set("fs.defaultFS", "hdfs://ns1/");    FileSystem fs = FileSystem.get(new URI("hdfs://ns1/"),conf,"hadoop");    fs.copyFromLocalFile(new Path("g:/eclipse-jee-luna-SR1-linux-gtk.tar.gz"), new Path("hdfs://ns1/"));    fs.close();    }    } |

# Hive

## Hive基本概念

### Hive简介

#### 什么是Hive

Hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供类SQL查询功能。

#### 为什么使用Hive

* 直接使用hadoop所面临的问题

人员学习成本太高

项目周期要求太短

MapReduce实现复杂查询逻辑开发难度太大

* 为什么要使用Hive

操作接口采用类SQL语法，提供快速开发的能力。

避免了去写MapReduce，减少开发人员的学习成本。

功能扩展很方便。

#### Hive的特点

* 可扩展

Hive可以自由的扩展集群的规模，一般情况下不需要重启服务。

* 延展性

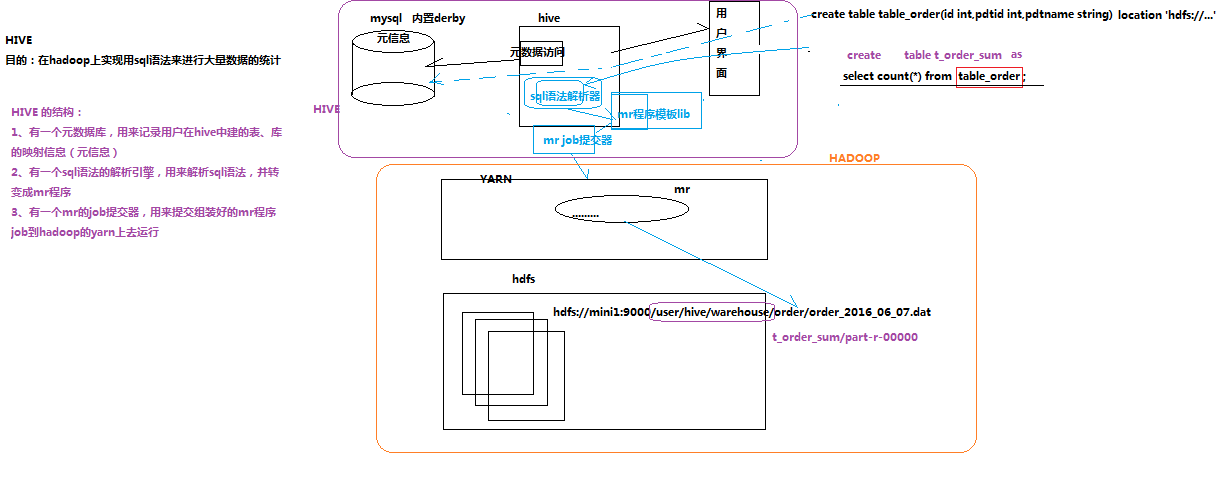
Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

* 容错

良好的容错性，节点出现问题SQL仍可完成执行。

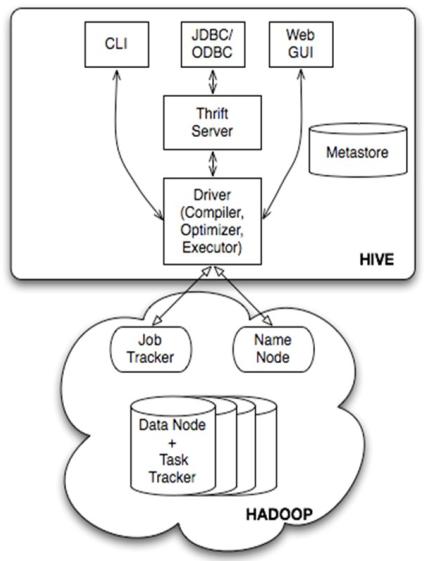
### Hive架构

#### 架构图



表对应一个文件,这种关系记录在关系型数据库里,

sql语法解析引擎:解析sql,然后根据模板库转为MapReduce程序,



Jobtracker是hadoop1.x中的组件，它的功能相当于： Resourcemanager+AppMaster

TaskTracker 相当于： Nodemanager + yarnchild

#### 基本组成

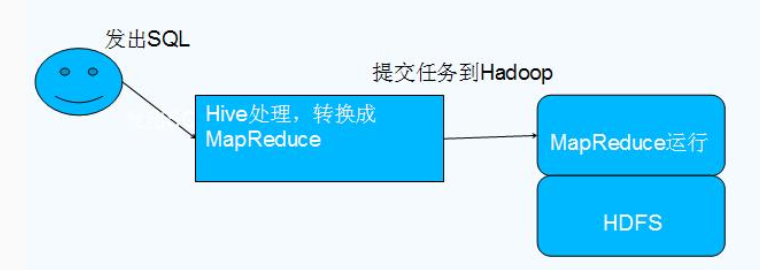
* 用户接口：包括 CLI、JDBC/ODBC、WebGUI。
* 元数据存储：通常是存储在关系数据库如 mysql , derby中。
* 解释器、编译器、优化器、执行器。

#### 各组件的基本功能

* 用户接口主要由三个：CLI、JDBC/ODBC和WebGUI。其中，CLI为shell命令行；JDBC/ODBC是Hive的JAVA实现，与传统数据库JDBC类似；WebGUI是通过浏览器访问Hive。
* 元数据存储：Hive 将元数据存储在数据库中。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
* 解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后有 MapReduce 调用执行。

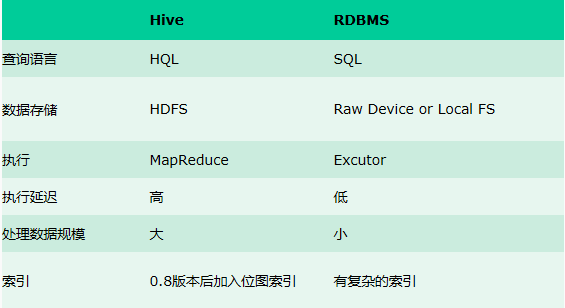
### Hive与Hadoop的关系

Hive利用HDFS存储数据，利用MapReduce查询分析数据



### Hive与传统数据库对比

1. 查询语言。由于 SQL 被广泛的应用在数据仓库中，因此，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。
2. 数据存储位置。Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。
3. 数据格式。Hive 中没有定义专门的数据格式，数据格式可以由用户指定，用户定义数据格式需要指定三个属性：列分隔符（通常为空格、”\t”、”\x001″）、行分隔符（”\n”）以及读取文件数据的方法（Hive 中默认有三个文件格式 TextFile，SequenceFile 以及 RCFile）。由于在加载数据的过程中，不需要从用户数据格式到 Hive 定义的数据格式的转换，因此，Hive 在加载的过程中不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的 HDFS 目录中。而在数据库中，不同的数据库有不同的存储引擎，定义了自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。
4. 数据更新。由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不支持对数据的改写和添加，所有的数据都是在加载的时候中确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO ...  VALUES 添加数据，使用 UPDATE ... SET 修改数据。
5. 索引。之前已经说过，Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入， Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。
6. 执行。Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的，而数据库通常有自己的执行引擎。
7. 执行延迟。之前提到，Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。
8. 可扩展性。由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。
9. 数据规模。由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。



hive现在版本也可以见索引,但是作用不是太大.

*总结：hive具有sql数据库的外表，但应用场景完全不同，hive只适合用来做批量数据统计分析*

### Hive的数据存储

1. Hive中所有的数据都存储在 HDFS 中，没有专门的数据存储格式（可支持Text，SequenceFile，ParquetFile，RCFILE等）

SequenceFile是hadoop中的一种文件格式：

文件内容是以序列化的kv对象来组织的

2、只需要在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。

3、Hive 中包含以下数据模型：DB、Table，External Table，Partition，Bucket。

* db：在hdfs中表现为${hive.metastore.warehouse.dir}目录下一个文件夹
* table：在hdfs中表现所属db目录下一个文件夹
* external table：与table类似，不过其数据存放位置可以在任意指定路径
* partition：在hdfs中表现为table目录下的子目录
* bucket：在hdfs中表现为同一个表目录下根据hash散列之后的多个文件

## HIVE的安装部署

### 安装

* derby版：

1、解压hive,直接执行bin/hive就可以使用了.默认使用derby的数据库

问题:版本不兼容问题：替换 /root/apps/hadoop/share/hadoop/yarn/lib中的老版本jline 为hive的lib中的jline-2.12.jar

缺点：多个地方安装hive后，数据库是根据hive命令执行的地方生成的,每一个hive是拥有一套自己的元数据，大家的库、表就不统一；

* 元数据库mysql版：

1、解压hive二进制安装包

2、修改配置文件hive-site.xml

3、将mysql的驱动包拷贝到{HIVE\_HOME}/lib目录下

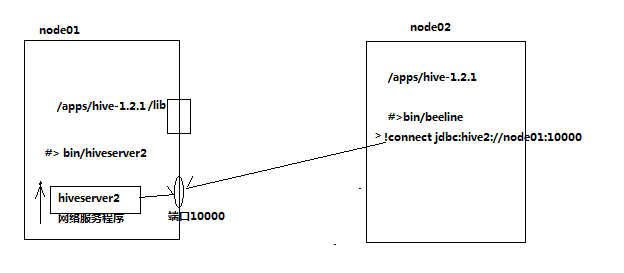
4、启动hive,直接执行bin/hive

### 使用方式

#### Hive交互shell

bin/hive

#### Hive JDBC服务

hive也可以启动为一个服务器，来对外提供

启动方式，（假如是在hadoop01上）：

启动为前台：bin/hiveserver2

启动为后台：nohup bin/hiveserver2 1>/var/log/hiveserver.log 2>/var/log/hiveserver.err &

启动成功后，可以在别的节点上用beeline去连接

* 方式（1）

hive/bin/beeline 回车，进入beeline的命令界面

输入命令连接hiveserver2

beeline> !connect jdbc:hive2//mini1:10000

（hadoop01是hiveserver2所启动的那台主机名，端口默认是10000）

* 方式（2）

或者启动就连接：

bin/beeline -u jdbc:hive2://mini1:10000 -n hadoop

接下来就可以做正常sql查询了

#### Hive命令

[hadoop@hdp-node-02 ~]$ hive -e ‘sql command’

示例：

bin/hive -e 'select \* from tingting.t\_zuoxi'

这样可以执行多条语句,将多条sql写在一个test.sql脚本里即可.

## Hive基本操作

### DDL操作

#### 创建表

* 建表语法

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table\_name

[(col\_name data\_type [COMMENT col\_comment], ...)]

[COMMENT table\_comment]

[PARTITIONED BY (col\_name data\_type [COMMENT col\_comment], ...)]

[CLUSTERED BY (col\_name, col\_name, ...)

[SORTED BY (col\_name [ASC|DESC], ...)] INTO num\_buckets BUCKETS]

[ROW FORMAT row\_format]

[STORED AS file\_format]

[LOCATION hdfs\_path]

表结构与数据格式之间的关系:建完表后,可以放入数据,也会加载到表中的数据中

load data local inpath '/root/hivedata/ip1.dat' into table t\_access;

表的分区:在create table的时候就指定分区的标识

load data local inpath '/root/hivedata/ip1.dat' into table t\_access partition(country='china');

分区就是表的目录下建立了一个子目录

查询分区的时候,加一个where条件即可

分桶:后面的字段必须是已经有的了,分区是必须是新的字段;

说明：

1. CREATE TABLE 创建一个指定名字的表。如果相同名字的表已经存在，则抛出异常；用户可以用 IF NOT EXISTS 选项来忽略这个异常。
2. EXTERNAL关键字可以让用户创建一个外部表，在建表的同时指定一个指向实际数据的路径（LOCATION），Hive 创建内部表时，会将数据移动到数据仓库指向的路径；若创建外部表，仅记录数据所在的路径，不对数据的位置做任何改变。在删除表的时候，内部表的元数据和数据会被一起删除，而外部表只删除元数据，不删除数据。

外部表可以放入任何指定目录,建表语句加入 external 后面接入location path参数.

直接对接数据来源的一般就用外部表

1. LIKE 允许用户复制现有的表结构，但是不复制数据。
2. ROW FORMAT

DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char]

[MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

| SERDE serde\_name [WITH SERDEPROPERTIES (property\_name=property\_value, property\_name=property\_value, ...)]

用户在建表的时候可以自定义 SerDe 或者使用自带的 SerDe。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用自带的 SerDe。在建表的时候，用户还需要为表指定列，用户在指定表的列的同时也会指定自定义的 SerDe，Hive通过 SerDe 确定表的具体的列的数据。

1. STORED AS

SEQUENCEFILE|TEXTFILE|RCFILE|PACKFILE

如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCEFILE。

6、CLUSTERED BY

对于每一个表（table）或者分区， Hive可以进一步组织成桶，也就是说桶是更为细粒度的数据范围划分。Hive也是 针对某一列进行桶的组织。Hive采用对列值哈希，然后除以桶的个数求余的方式决定该条记录存放在哪个桶当中。

把表（或者分区）组织成桶（Bucket）有两个理由：

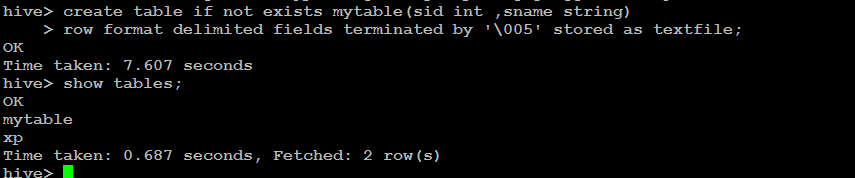
（1）获得更高的查询处理效率。桶为表加上了额外的结构，Hive 在处理有些查询时能利用这个结构。具体而言，连接两个在（包含连接列的）相同列上划分了桶的表，可以使用 Map 端连接 （Map-side join）高效的实现。比如JOIN操作。对于JOIN操作两个表有一个相同的列，如果对这两个表都进行了桶操作。那么将保存相同列值的桶进行JOIN操作就可以，可以大大较少JOIN的数据量。

（2）使取样（sampling）更高效。在处理大规模数据集时，在开发和修改查询的阶段，如果能在数据集的一小部分数据上试运行查询，会带来很多方便。

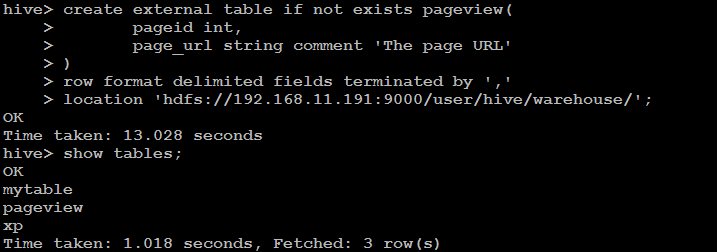
distributed + sorted = cluster by

* 具体实例

1. 创建内部表mytable。

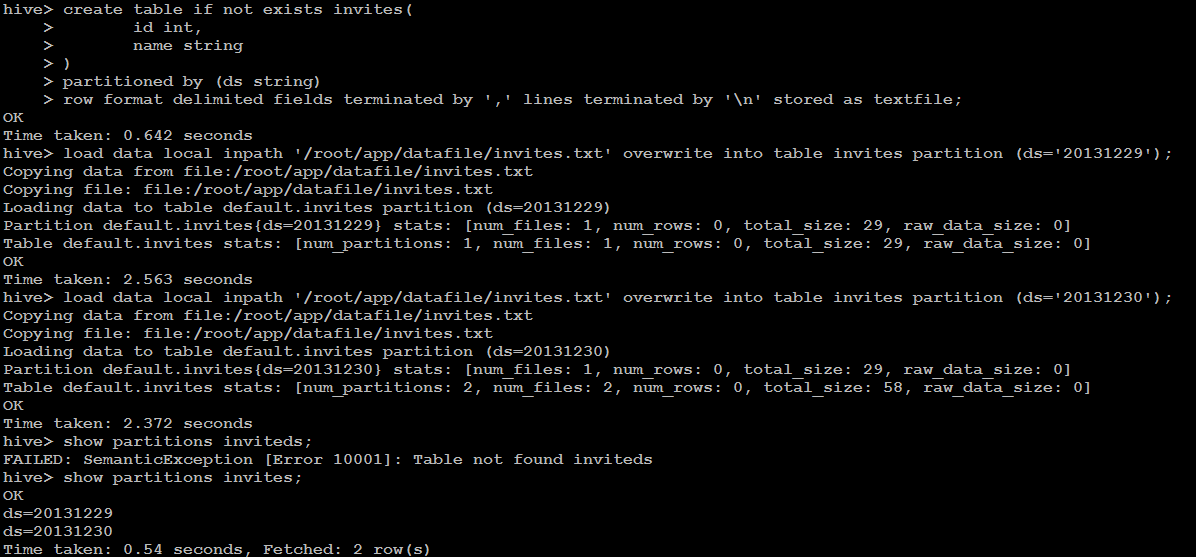


1. 创建外部表pageview。

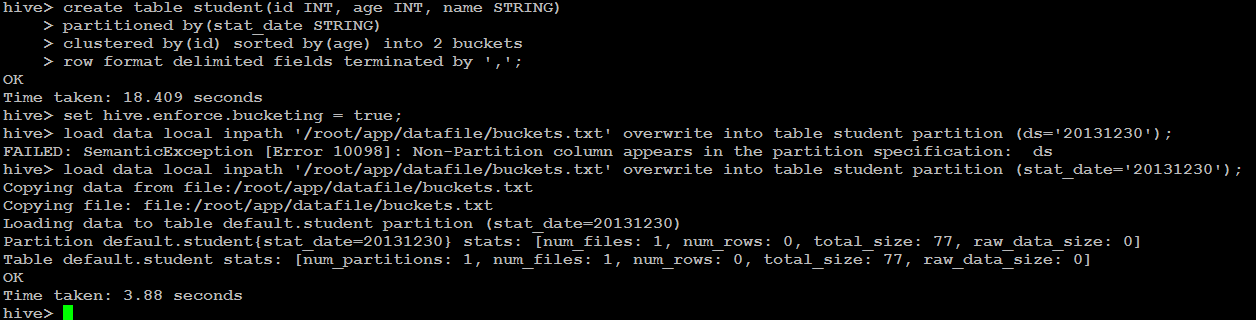


1. 创建分区表invites。

|  |
| --- |
| create table student\_p(Sno int,Sname string,Sex string,Sage int,Sdept string) partitioned by(part string) row format delimited fields terminated by ','stored as textfile; |



1. 创建带桶的表student。



#### 修改表

1.增加/删除分区

* 语法结构

ALTER TABLE table\_name ADD [IF NOT EXISTS] partition\_spec [ LOCATION 'location1' ] partition\_spec [ LOCATION 'location2' ] ...

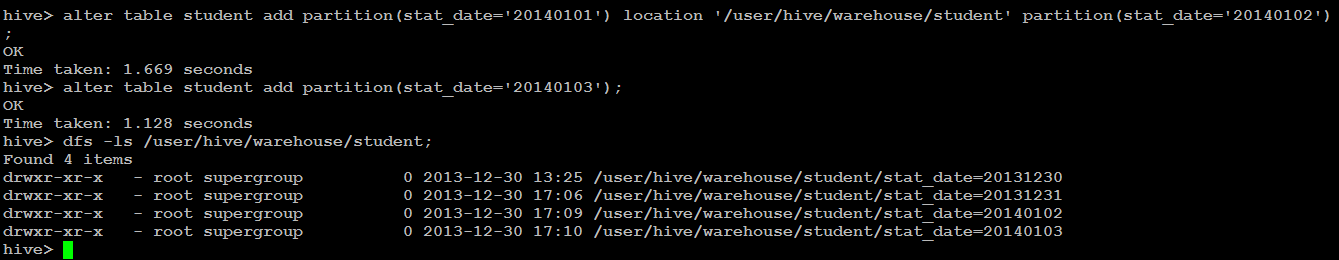
partition\_spec:

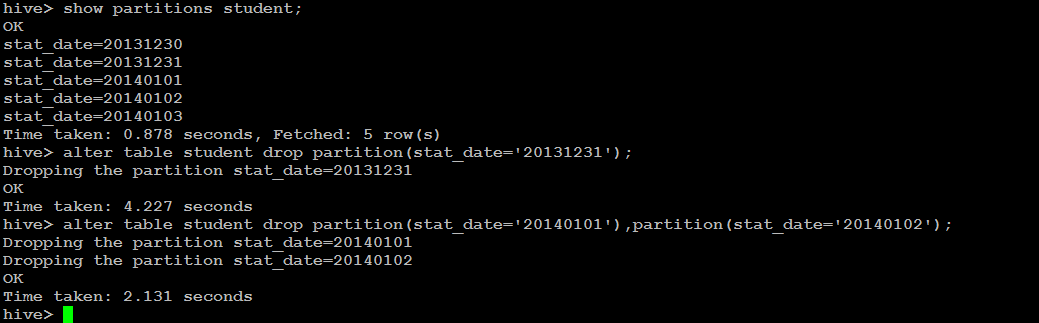
: PARTITION (partition\_col = partition\_col\_value, partition\_col = partiton\_col\_value, ...)

ALTER TABLE table\_name DROP partition\_spec, partition\_spec,...

* 具体实例

|  |
| --- |
| alter table student\_p add partition(part='a') partition(part='b'); |



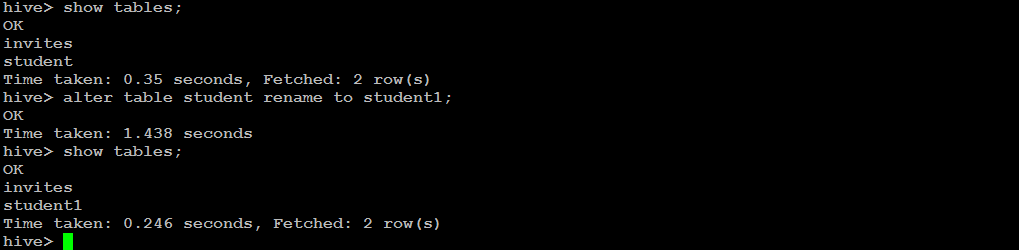


2.重命名表

* 语法结构

ALTER TABLE table\_name RENAME TO new\_table\_name

* 具体实例



3.增加/更新列

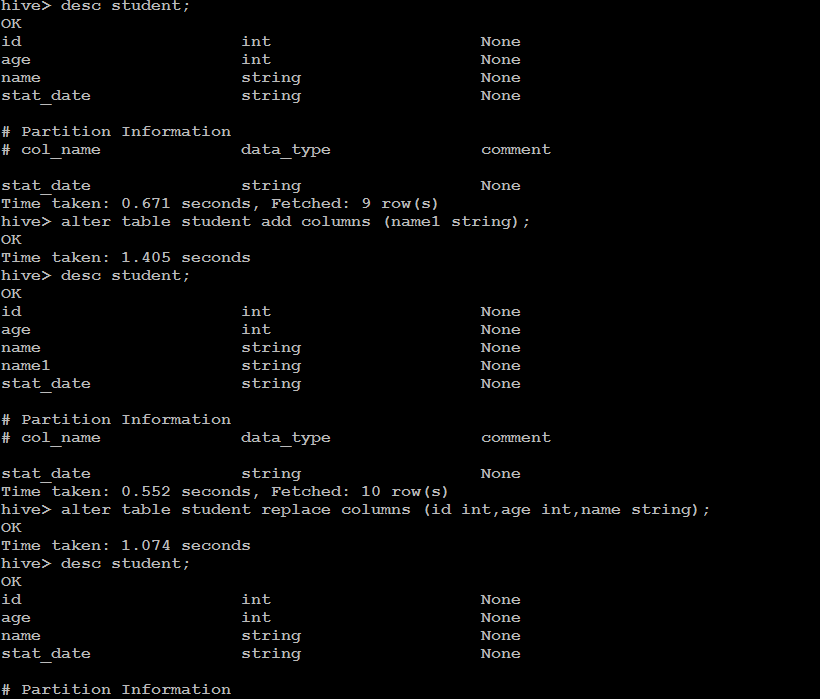
* 语法结构

ALTER TABLE table\_name ADD|REPLACE COLUMNS (col\_name data\_type [COMMENT col\_comment], ...)

*注：ADD是代表新增一字段，字段位置在所有列后面(partition列前)，REPLACE则是表示替换表中****所有字段****。*

ALTER TABLE table\_name CHANGE [COLUMN] col\_old\_name col\_new\_name column\_type [COMMENT col\_comment] [FIRST|AFTER column\_name]

* 具体实例



#### 显示命令

show tables;

show databases;

show partitions table\_name;

show functions; #查看函数信息

desc extended t\_name;

desc formatted table\_name;

### DML操作

#### Load

* 语法结构

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO

TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

说明：

1. Load 操作只是单纯的复制(本地)/移动(HDFS上)操作，将数据文件移动到 Hive 表对应的位置。
2. filepath：

相对路径，例如：project/data1

绝对路径，例如：/user/hive/project/data1

包含模式的完整 URI，列如：

hdfs://namenode:9000/user/hive/project/data1

1. LOCAL关键字

如果指定了 LOCAL， load 命令会去查找本地文件系统中的 filepath。

如果没有指定 LOCAL 关键字，则根据inpath中的uri找文件

{uri解释:如果指定了 LOCAL，那么：

load 命令会去查找本地文件系统中的 filepath。如果发现是相对路径，则路径会被解释为相对于当前用户的当前路径。

load 命令会将 filepath中的文件复制到目标文件系统中。目标文件系统由表的位置属性决定。被复制的数据文件移动到表的数据对应的位置。

如果没有指定 LOCAL 关键字，如果 filepath 指向的是一个完整的 URI，hive 会直接使用这个 URI。 否则：如果没有指定 schema 或者 authority，Hive 会使用在 hadoop 配置文件中定义的 schema 和 authority，fs.default.name 指定了 Namenode 的 URI。

如果路径不是绝对的，Hive 相对于/user/进行解释。

Hive 会将 filepath 中指定的文件内容移动到 table （或者 partition）所指定的路径中。}

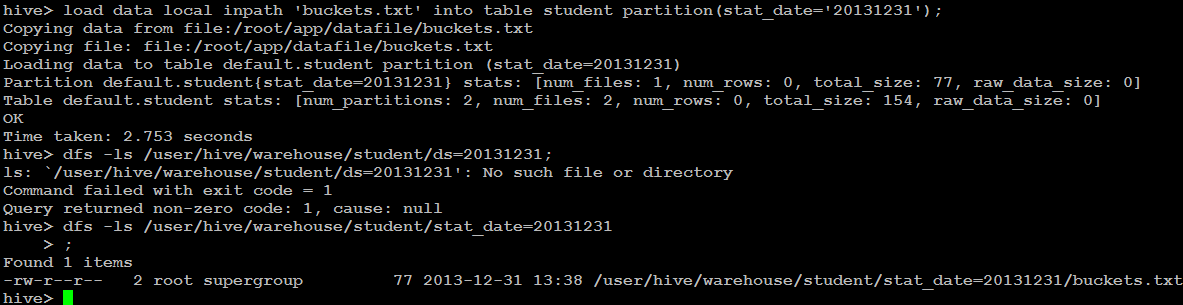
1. OVERWRITE 关键字

如果使用了 OVERWRITE 关键字，则目标表（或者分区）中的内容会被删除，然后再将 filepath 指向的文件/目录中的内容添加到表/分区中。

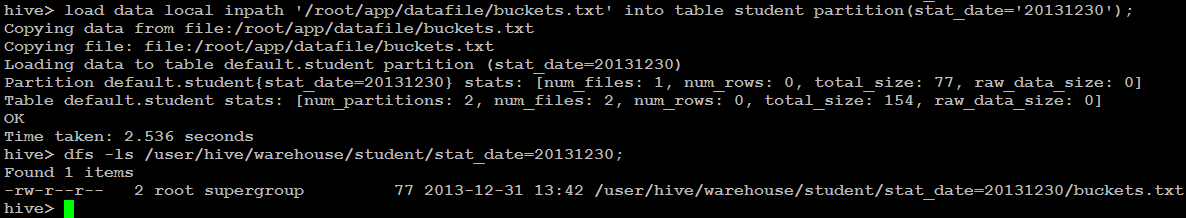
如果目标表（分区）已经有一个文件，并且文件名和 filepath 中的文件名冲突，那么现有的文件会被新文件所替代。

* 具体实例

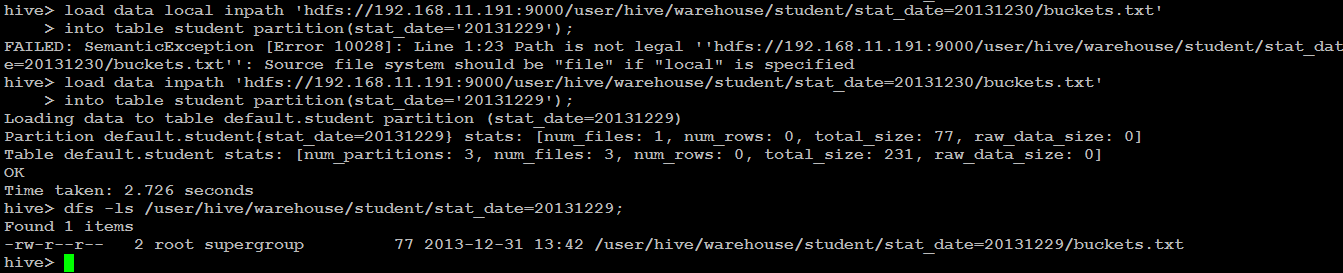
1. 加载相对路径数据。



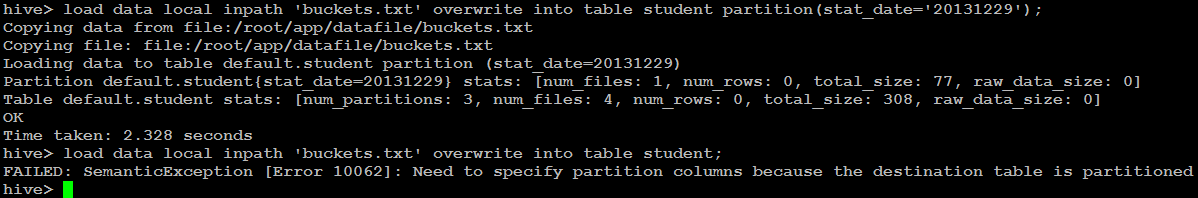
1. 加载绝对路径数据。



1. 加载包含模式数据。



1. OVERWRITE关键字使用。



#### Insert

* 将查询结果插入Hive表
* 语法结构

利用查询语句，将查询结果插入新的表

INSERT OVERWRITE [INTO] TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1 FROM from\_statement

插入一条数据

INSERT INTO TABLE VALUES(XX,YY,ZZ);

Multi Inserts多重插入:

FROM from\_statement

INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1

[INSERT OVERWRITE TABLE tablename2 [PARTITION ...]

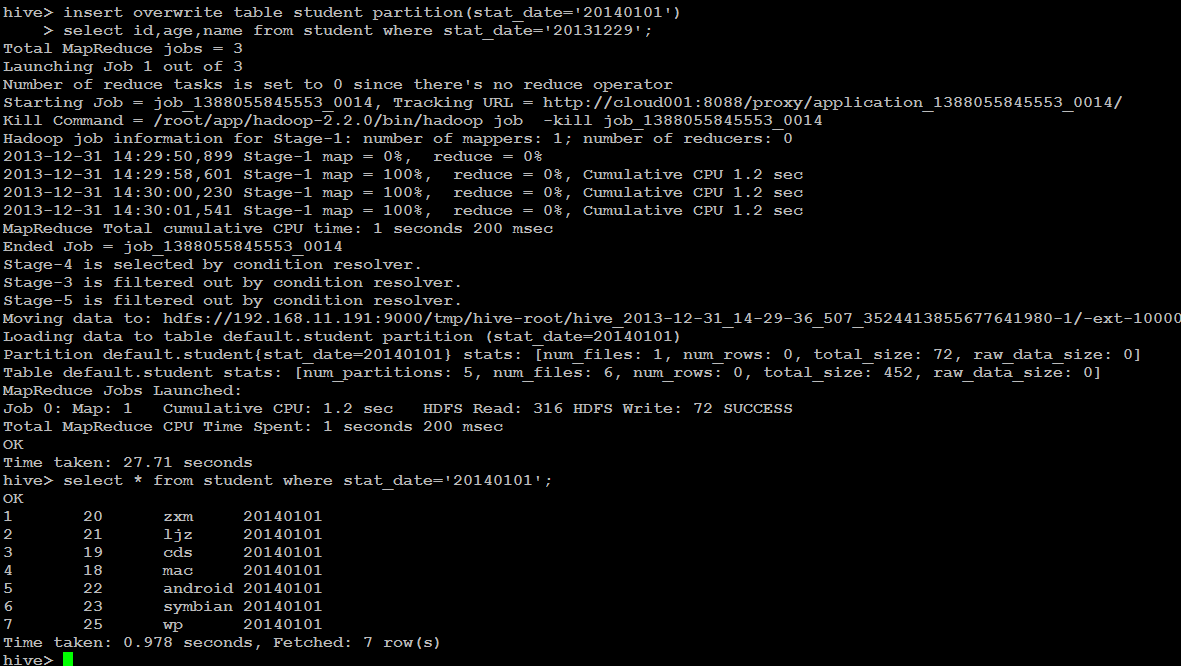
select\_statement2] ...

Dynamic partition inserts动态分区插入:

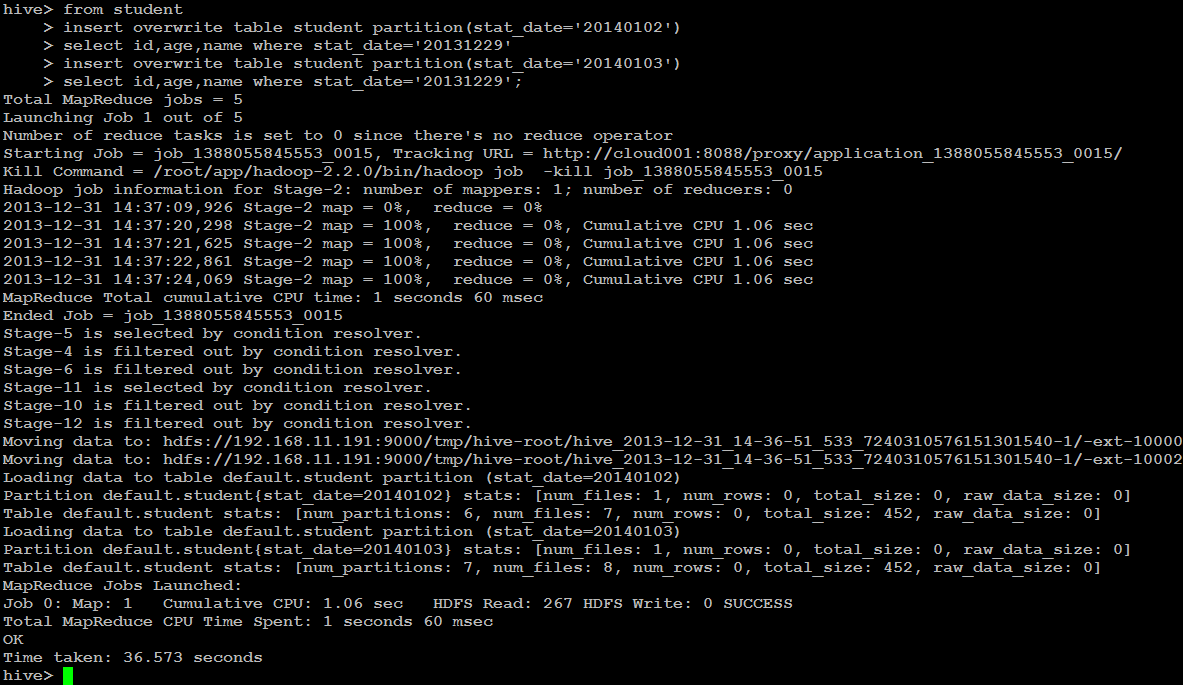
INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select\_statement FROM from\_statement

* 具体实例

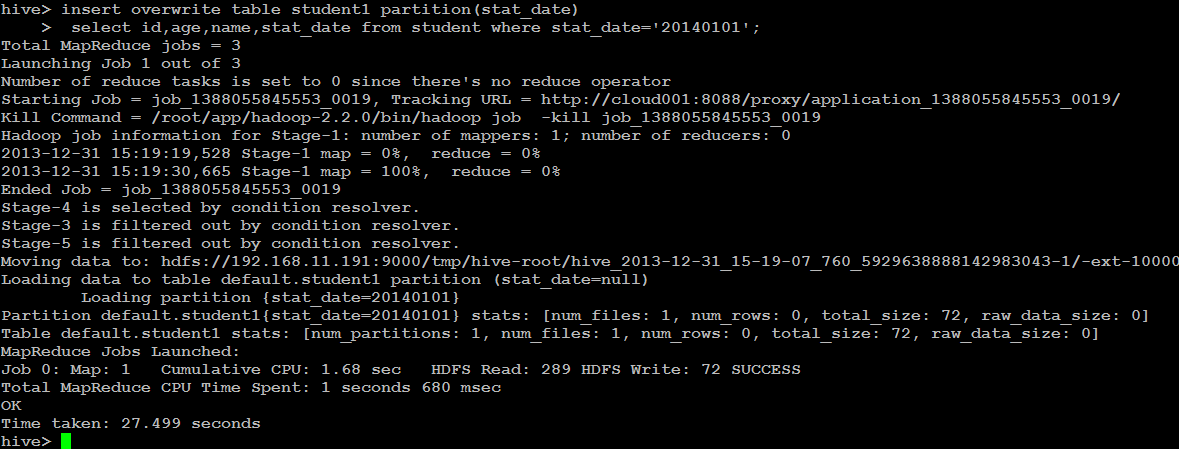
1、基本模式插入。



2、多插入模式。



3、自动分区模式。



* 导出表数据
* 语法结构

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 SELECT ... FROM ...

multiple inserts:

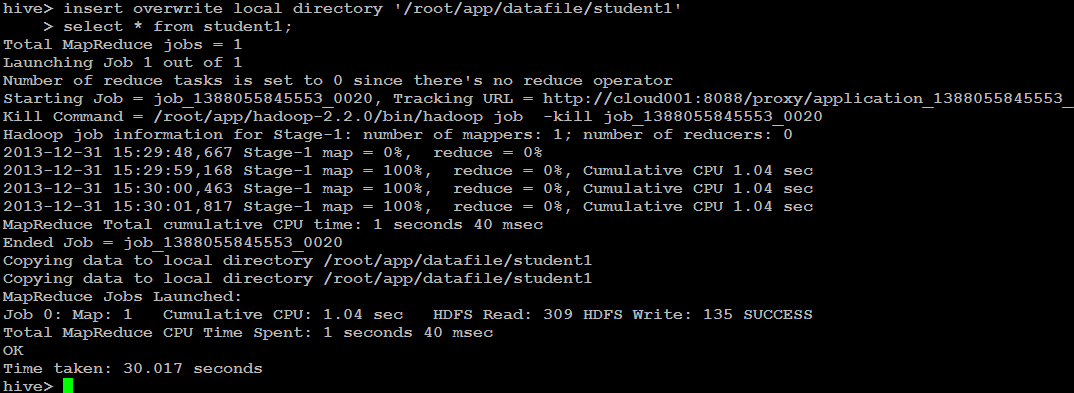
FROM from\_statement

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select\_statement1

[INSERT OVERWRITE [LOCAL] DIRECTORY directory2 select\_statement2] ...

* 具体实例

1、导出文件到本地。



说明：

*数据写入到文件系统时进行文本序列化，且每列用^A来区分，\n为换行符。用more命令查看时不容易看出分割符，可以使用: sed -e 's/\x01/|/g' filename查看。*

2、导出数据到HDFS。



#### SELECT

* 基本的Select操作
* 语法结构

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

JOIN table\_other ON expr

[WHERE where\_condition]

[GROUP BY col\_list [HAVING condition]]

[CLUSTER BY col\_list

| [DISTRIBUTE BY col\_list] [SORT BY| ORDER BY col\_list]

]

[LIMIT number]

*注：1、order by 会对输入做全局排序，因此只有一个reducer，会导致当输入规模较大时，需要较长的计算时间。*

*2、sort by不是全局排序，其在数据进入reducer前完成排序。因此，如果用sort by进行排序，并且设置mapred.reduce.tasks>1，则sort by只保证每个reducer的输出有序，不保证全局有序。*

*3、distribute by(字段)根据指定的字段将数据分到不同的reducer，且分发算法是hash散列。*

*4、Cluster by(字段) 除了具有Distribute by的功能外，还会对该字段进行排序。*

*因此，如果分桶和sort字段是同一个时，此时，cluster by = distribute by + sort by*

分桶表的作用：最大的作用是用来提高join操作的效率；

（思考这个问题：

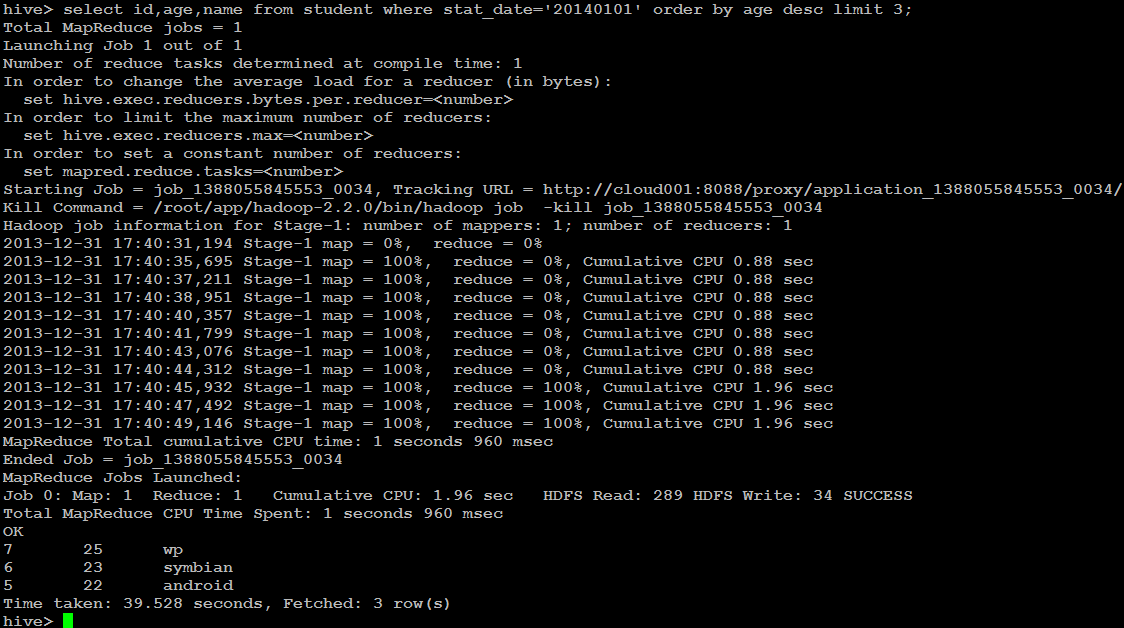
select a.id,a.name,b.addr from a join b on a.id = b.id;

如果a表和b表已经是分桶表，而且分桶的字段是id字段

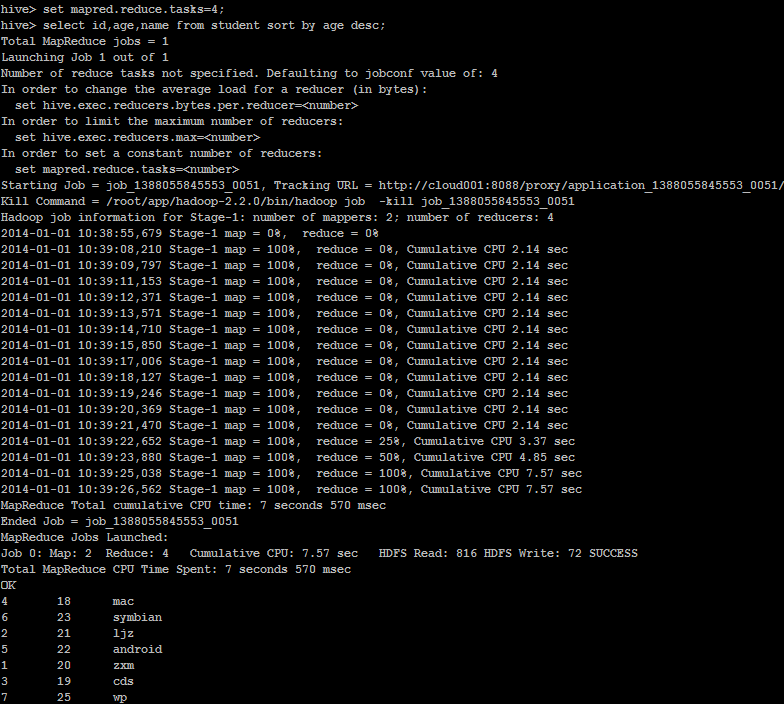
做这个join操作时，还需要全表做笛卡尔积吗？）

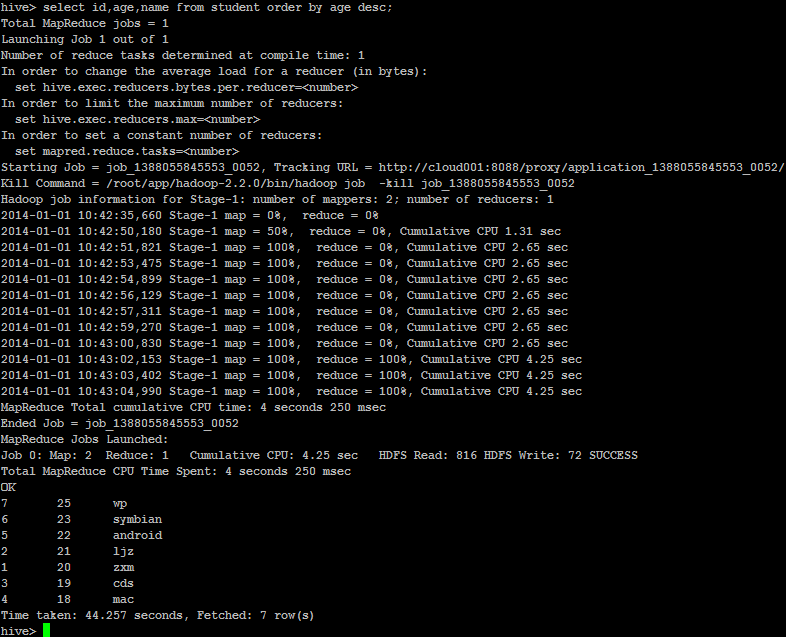
* 具体实例

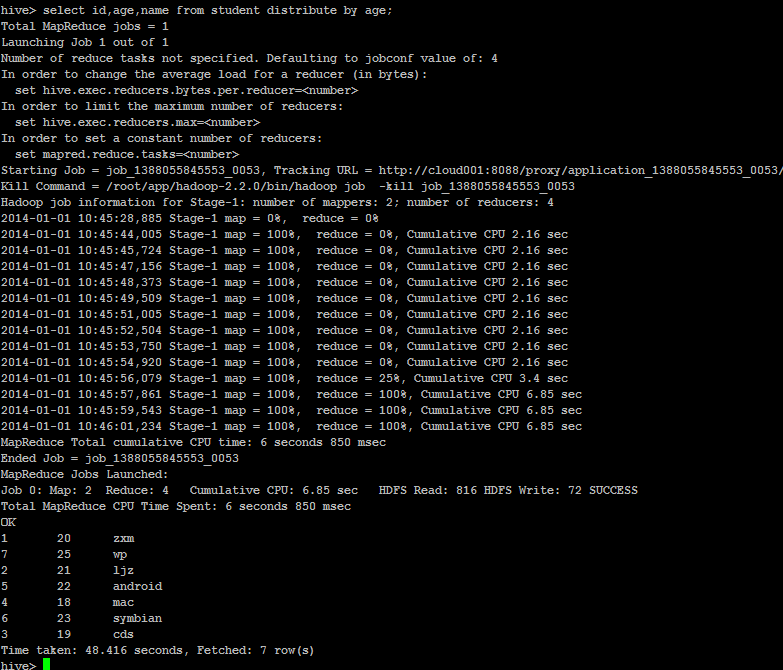
1、获取年龄大的3个学生。



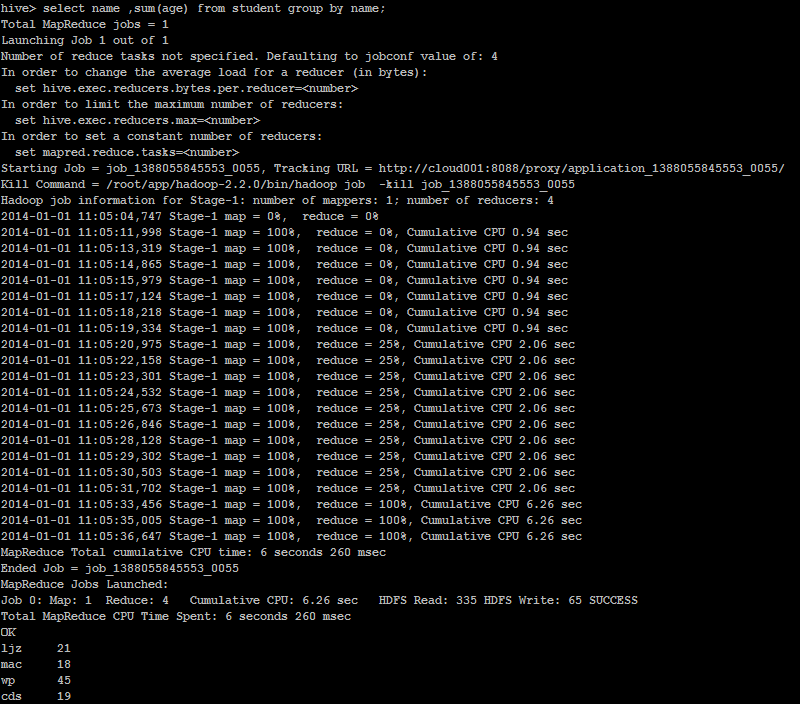
2、查询学生信息按年龄，降序排序。







3、按学生名称汇总学生年龄。



### Hive Join

* 语法结构

join\_table:

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference join\_condition

| table\_reference LEFT SEMI JOIN table\_reference join\_condition

Hive 支持等值连接（equality joins）、外连接（outer joins）和（left/right joins）。Hive **不支持非等值的连接**，因为非等值连接非常难转化到 map/reduce 任务。

另外，Hive 支持多于 2 个表的连接。

写 join 查询时，需要注意几个关键点：

**1. 只支持等值join**

例如：

SELECT a.\* FROM a JOIN b ON (a.id = b.id)

SELECT a.\* FROM a JOIN b

ON (a.id = b.id AND a.department = b.department)

是正确的，然而:

SELECT a.\* FROM a JOIN b ON (a.id>b.id)

是错误的。

**2. 可以 join 多于 2 个表。**

例如

SELECT a.val, b.val, c.val FROM a JOIN b

ON (a.key = b.key1) JOIN c ON (c.key = b.key2)

如果join中多个表的 join key 是同一个，则 join 会被转化为单个 map/reduce 任务，例如：

SELECT a.val, b.val, c.val FROM a JOIN b

ON (a.key = b.key1) JOIN c

ON (c.key = b.key1)

被转化为单个 map/reduce 任务，因为 join 中只使用了 b.key1 作为 join key。

SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1)

JOIN c ON (c.key = b.key2)

而这一 join 被转化为 2 个 map/reduce 任务。因为 b.key1 用于第一次 join 条件，而 b.key2 用于第二次 join。

**3．join 时，每次 map/reduce 任务的逻辑：**

reducer 会缓存 join 序列中除了最后一个表的所有表的记录，再通过最后一个表将结果序列化到文件系统。这一实现有助于在 reduce 端减少内存的使用量。实践中，应该把最大的那个表写在最后（否则会因为缓存浪费大量内存）。例如：

SELECT a.val, b.val, c.val FROM a

JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)

所有表都使用同一个 join key（使用 1 次 map/reduce 任务计算）。Reduce 端会缓存 a 表和 b 表的记录，然后每次取得一个 c 表的记录就计算一次 join 结果，类似的还有：

SELECT a.val, b.val, c.val FROM a

JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)

这里用了 2 次 map/reduce 任务。第一次缓存 a 表，用 b 表序列化；第二次缓存第一次 map/reduce 任务的结果，然后用 c 表序列化。

**4．LEFT，RIGHT 和 FULL OUTER 关键字用于处理 join 中空记录的情况**

例如：

SELECT a.val, b.val FROM

a LEFT OUTER JOIN b ON (a.key=b.key)

对应所有 a 表中的记录都有一条记录输出。输出的结果应该是 a.val, b.val，当 a.key=b.key 时，而当 b.key 中找不到等值的 a.key 记录时也会输出:

a.val, NULL

所以 a 表中的所有记录都被保留了；

“a RIGHT OUTER JOIN b”会保留所有 b 表的记录。

**Join 发生在 WHERE 子句之前**。如果你想限制 join 的输出，应该在 WHERE 子句中写过滤条件——或是在 join 子句中写。这里面一个容易混淆的问题是表分区的情况：

SELECT a.val, b.val FROM a

LEFT OUTER JOIN b ON (a.key=b.key)

WHERE a.ds='2009-07-07' AND b.ds='2009-07-07'

会 join a 表到 b 表（OUTER JOIN），列出 a.val 和 b.val 的记录。WHERE 从句中可以使用其他列作为过滤条件。但是，如前所述，如果 b 表中找不到对应 a 表的记录，b 表的所有列都会列出 NULL，**包括 ds 列**。也就是说，join 会过滤 b 表中不能找到匹配 a 表 join key 的所有记录。这样的话，LEFT OUTER 就使得查询结果与 WHERE 子句无关了。解决的办法是在 OUTER JOIN 时使用以下语法：

SELECT a.val, b.val FROM a LEFT OUTER JOIN b

ON (a.key=b.key AND

b.ds='2009-07-07' AND

a.ds='2009-07-07')

这一查询的结果是预先在 join 阶段过滤过的，所以不会存在上述问题。这一逻辑也可以应用于 RIGHT 和 FULL 类型的 join 中。

**Join 是不能交换位置的。**无论是 LEFT 还是 RIGHT join，都是左连接的。

SELECT a.val1, a.val2, b.val, c.val

FROM a

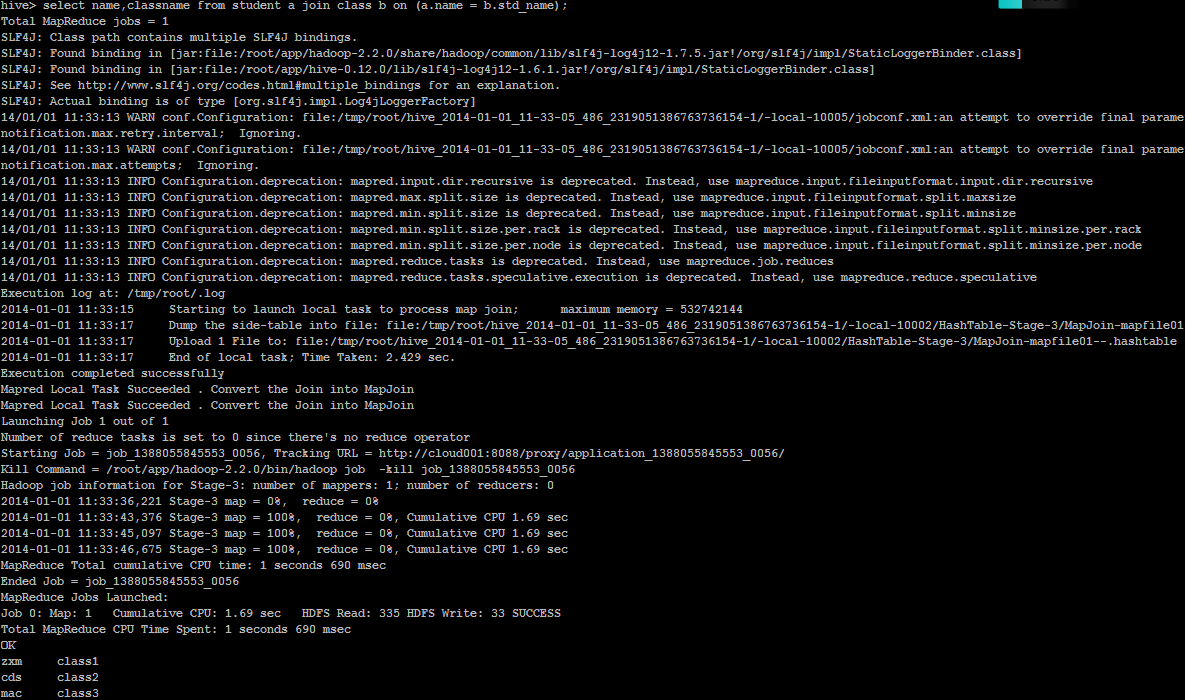
JOIN b ON (a.key = b.key)

LEFT OUTER JOIN c ON (a.key = c.key)

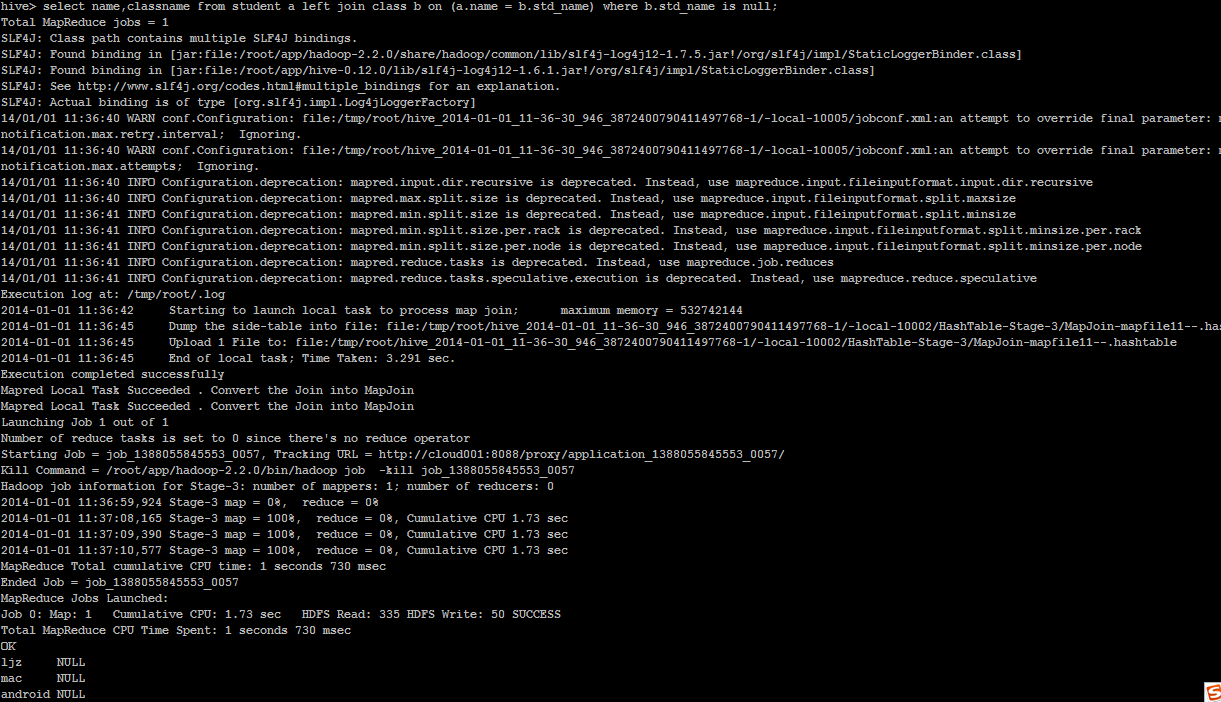
先 join a 表到 b 表，丢弃掉所有 join key 中不匹配的记录，然后用这一中间结果和 c 表做 join。这一表述有一个不太明显的问题，就是当一个 key 在 a 表和 c 表都存在，但是 b 表中不存在的时候：整个记录在第一次 join，即 a JOIN b 的时候都被丢掉了（包括a.val1，a.val2和a.key），然后我们再和 c 表 join 的时候，如果 c.key 与 a.key 或 b.key 相等，就会得到这样的结果：NULL, NULL, NULL, c.val

* 具体实例

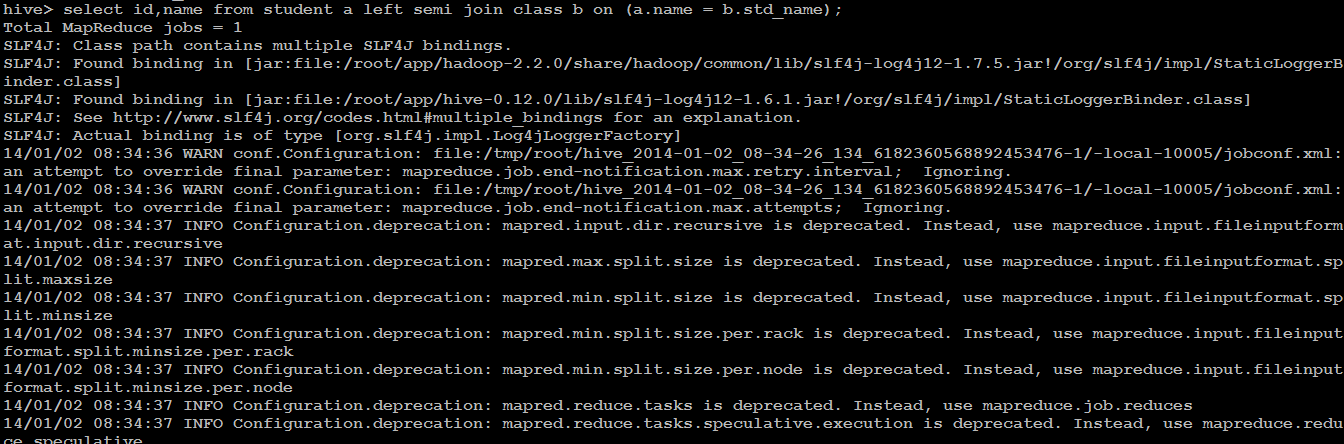
1. 获取已经分配班级的学生姓名。



1. 获取尚未分配班级的学生姓名。



1. LEFT SEMI JOIN是IN/EXISTS的高效实现。



## Hive Shell参数

### Hive命令行

补充：如何才能在yarn的管理控制台上查看已经运行完成的job的信息

需要在hadoop集群上启动一个jobhistory服务器

linux-shell>mr-jobhistory-daemon.sh start historyserver

* 语法结构

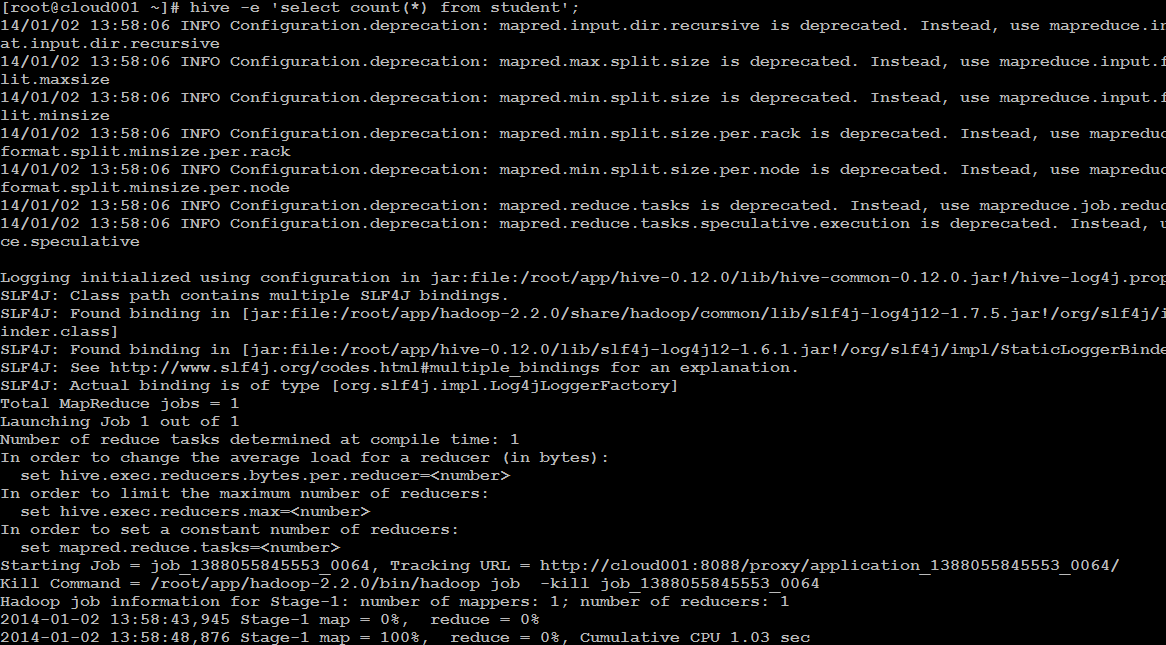
hive [-hiveconf x=y]\* [<-i filename>]\* [<-f filename>|<-e query-string>] [-S]

说明：

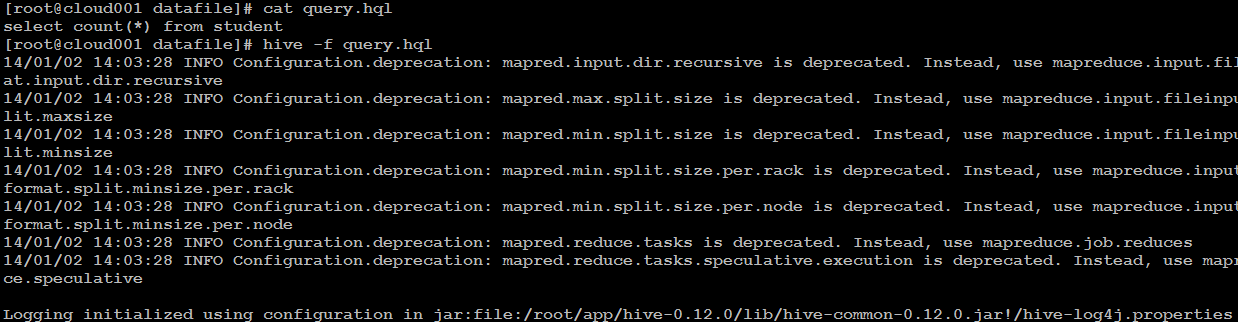
1. -i 从文件初始化HQL。
2. -e从命令行执行指定的HQL
3. -f 执行HQL脚本
4. -v 输出执行的HQL语句到控制台
5. -p <port> connect to Hive Server on port number
6. -hiveconf x=y Use this to set hive/hadoop configuration variables.

* 具体实例

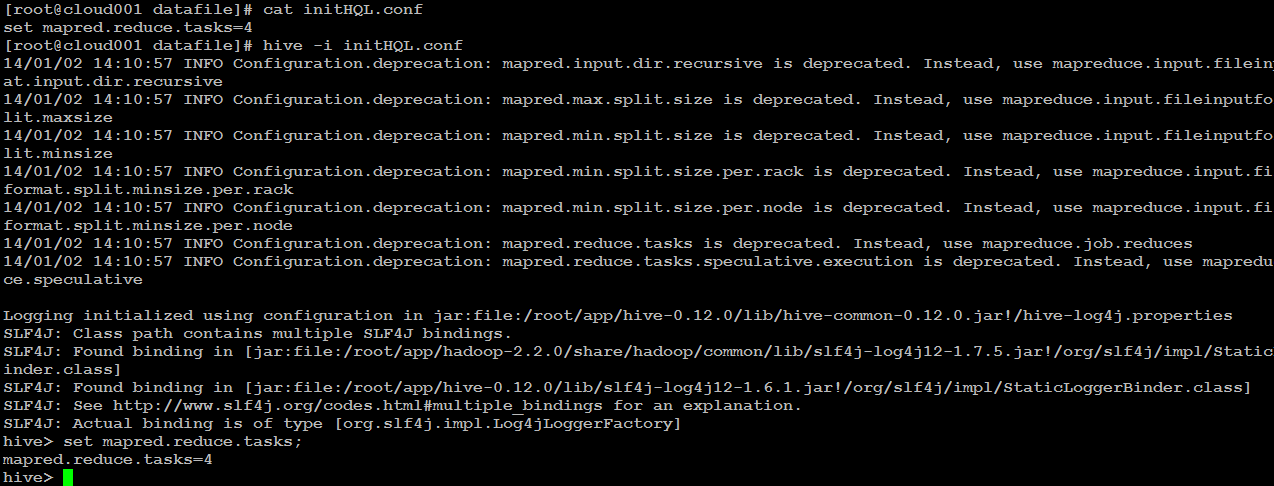
1、运行一个查询。



2、运行一个文件。



3、运行参数文件。



### Hive参数配置方式

*Hive参数大全：*

*https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties*

开发Hive应用时，不可避免地需要设定Hive的参数。设定Hive的参数可以调优HQL代码的执行效率，或帮助定位问题。然而实践中经常遇到的一个问题是，为什么设定的参数没有起作用？这通常是错误的设定方式导致的。

**对于一般参数，有以下三种设定方式：**

* 配置文件
* 命令行参数
* 参数声明

**配置文件**：Hive的配置文件包括

* 用户自定义配置文件：$HIVE\_CONF\_DIR/hive-site.xml
* 默认配置文件：$HIVE\_CONF\_DIR/hive-default.xml

用户自定义配置会覆盖默认配置。

另外，Hive也会读入Hadoop的配置，因为Hive是作为Hadoop的客户端启动的，Hive的配置会覆盖Hadoop的配置。

配置文件的设定对本机启动的所有Hive进程都有效。

**命令行参数**：启动Hive（客户端或Server方式）时，可以在命令行添加-hiveconf param=value来设定参数，例如：

bin/hive -hiveconf hive.root.logger=INFO,console

这一设定对本次启动的Session（对于Server方式启动，则是所有请求的Sessions）有效。

**参数声明**：可以在HQL中使用SET关键字设定参数，例如：

set mapred.reduce.tasks=100;

这一设定的作用域也是session级的。

set hive.exec.reducers.bytes.per.reducer=<number> 每个reduce task的平均负载数据量

hive会估算我们的总数据量，然后用总数据量除以上述参数值，就能得出需要运行的reduce task数

set hive.exec.reducers.max=<number> 设置reduce task数量的上限

set mapreduce.job.reduces=<number> 指定固定的reduce task数量

但是，这个参数在必要时<业务逻辑决定只能用一个reduce task> hive会忽略

上述三种设定方式的优先级依次递增。即参数声明覆盖命令行参数，命令行参数覆盖配置文件设定。注意某些系统级的参数，例如log4j相关的设定，必须用前两种方式设定，因为那些参数的读取在Session建立以前已经完成了。

## Hive函数

### 内置运算符

*内容较多，见《Hive官方文档》*

### 内置函数

*内容较多，见《Hive官方文档》*

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF>

测试各种内置函数的快捷方法：

1. 创建一个dual表

create table dual(id string);

1. load一个文件（一行，一个空格）到dual表
2. select substr('angelababy',2,3) from dual;

### Hive自定义函数和Transform

当Hive提供的内置函数无法满足你的业务处理需要时，此时就可以考虑使用用户自定义函数（UDF：user-defined function）。

#### 自定义函数类别

UDF 作用于单个数据行，产生一个数据行作为输出。（数学函数，字符串函数）

UDAF（用户定义聚集函数）：接收多个输入数据行，并产生一个输出数据行。（count，max）

#### UDF开发实例

* 简单UDF示例

1、先开发一个java类，继承UDF，并重载evaluate方法

|  |
| --- |
| package cn.itcast.bigdata.udf  import org.apache.hadoop.hive.ql.exec.UDF;  import org.apache.hadoop.io.Text;  public final class Lower extends UDF{  public Text evaluate(final Text s){  if(s==null){return null;}  return new Text(s.toString().toLowerCase());  }  } |

2、打成jar包上传到服务器

3、将jar包添加到hive的classpath

hive>add JAR /home/hadoop/udf.jar;

1. 创建临时函数与开发好的java class关联

|  |
| --- |
| Hive>create temporary function tolowercase as 'cn.itcast.bigdata.udf.ToProvince'; |

1. 即可在hql中使用自定义的函数tolowercase ip

Select tolowercase(name),age from t\_test;

* Json数据解析UDF开发

作业：

有原始json数据如下：

|  |
| --- |
| {"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}  {"movie":"661","rate":"3","timeStamp":"978302109","uid":"1"}  {"movie":"914","rate":"3","timeStamp":"978301968","uid":"1"}  {"movie":"3408","rate":"4","timeStamp":"978300275","uid":"1"}  {"movie":"2355","rate":"5","timeStamp":"978824291","uid":"1"}  {"movie":"1197","rate":"3","timeStamp":"978302268","uid":"1"}  {"movie":"1287","rate":"5","timeStamp":"978302039","uid":"1"} |

需要将数据导入到hive数据仓库中

我不管你中间用几个表，最终我要得到一个结果表：

|  |  |  |  |
| --- | --- | --- | --- |
| movie | rate | timestamp | uid |
| 1197 | 3 | 978302268 | 1 |

注：全在hive中完成，可以用自定义函数

#### Transform实现

Hive的 TRANSFORM 关键字***提供了在SQL中调用自写脚本的功能***

适合实现Hive中没有的功能又不想写UDF的情况

使用示例1：下面这句sql就是借用了weekday\_mapper.py对数据进行了处理.

|  |
| --- |
| CREATE TABLE u\_data\_new (  movieid INT,  rating INT,  weekday INT,  userid INT)  ROW FORMAT DELIMITED  FIELDS TERMINATED BY '\t';  add FILE weekday\_mapper.py;  INSERT OVERWRITE TABLE u\_data\_new  SELECT  TRANSFORM (movieid , rate, timestring,uid)  USING 'python weekday\_mapper.py'  AS (movieid, rating, weekday,userid)  FROM t\_rating; |

其中weekday\_mapper.py内容如下

|  |
| --- |
| #!/bin/python  import sys  import datetime  for line in sys.stdin:  line = line.strip()  movieid, rating, unixtime,userid = line.split('\t')  weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()  print '\t'.join([movieid, rating, str(weekday),userid]) |

### HIVE特殊分隔符处理

补充：hive读取数据的机制：

1. 首先用InputFormat<默认是：org.apache.hadoop.mapred.TextInputFormat >的一个具体实现类读入文件数据，返回一条一条的记录（可以是行，或者是你逻辑中的“行”）
2. 然后利用SerDe<默认：org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe>的一个具体实现类，对上面返回的一条一条的记录进行字段切割

Hive对文件中字段的分隔符默认情况下只支持单字节分隔符，如果数据文件中的分隔符是多字符的，如下所示：

01||zhangsan

02||lisi

可用以下方式处理：

#### 使用RegexSerDe通过正则表达式来抽取字段

|  |
| --- |
| drop table t\_bi\_reg;  create table t\_bi\_reg(id string,name string)  row format serde 'org.apache.hadoop.hive.serde2.RegexSerDe'  with serdeproperties(  'input.regex'='(.\*)\\|\\|(.\*)',  'output.format.string'='%1$s %2$s'  )  stored as textfile;  hive>load data local inpath '/root/hivedata/bi.dat' into table t\_bi\_reg;  hive>select \* from t\_bi\_reg; |

#### 通过自定义InputFormat解决特殊分隔符问题

其原理是在inputformat读取行的时候将数据中的“多字节分隔符”替换为hive默认的分隔符（ctrl+A 亦即 \001）或用于替代的单字符分隔符，以便hive在serde操作时按照默认的单字节分隔符进行字段抽取

|  |
| --- |
| package cn.itcast.bigdata.hive.inputformat;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapred.FileSplit;  import org.apache.hadoop.mapred.InputSplit;  import org.apache.hadoop.mapred.JobConf;  import org.apache.hadoop.mapred.LineRecordReader;  import org.apache.hadoop.mapred.RecordReader;  import org.apache.hadoop.mapred.Reporter;  import org.apache.hadoop.mapred.TextInputFormat;  public class BiDelimiterInputFormat extends TextInputFormat {  @Override  public RecordReader<LongWritable, Text> getRecordReader(  InputSplit genericSplit, JobConf job, Reporter reporter)  throws IOException {  reporter.setStatus(genericSplit.toString());  MyDemoRecordReader reader = new MyDemoRecordReader(  new LineRecordReader(job, (FileSplit) genericSplit));  return reader;  }  public static class MyDemoRecordReader implements  RecordReader<LongWritable, Text> {  LineRecordReader reader;  Text text;  public MyDemoRecordReader(LineRecordReader reader) {  this.reader = reader;  text = reader.createValue();  }  @Override  public void close() throws IOException {  reader.close();  }  @Override  public LongWritable createKey() {  return reader.createKey();  }  @Override  public Text createValue() {  return new Text();  }  @Override  public long getPos() throws IOException {  return reader.getPos();  }  @Override  public float getProgress() throws IOException {  return reader.getProgress();  }  @Override  public boolean next(LongWritable key, Text value) throws IOException {  while (reader.next(key, text)) {  String strReplace = text.toString().toLowerCase().replaceAll("\\|\\|", "|");  Text txtReplace = new Text();  txtReplace.set(strReplace);  value.set(txtReplace.getBytes(), 0, txtReplace.getLength());  return true;  }  return false;  }  }  } |

注意：上述代码中的api全部使用hadoop的老api接口 org.apache.hadoop.mapred….

然后将工程打包，并拷贝至hive安装目录的lib文件夹中，并重启hive，使用以下语句建表即可：

|  |
| --- |
| hive> create table t\_bi(id string,name string)  > row format delimited  > fields terminated by '|'  > stored as inputformat 'cn.itcast.bigdata.hive.inputformat.BiDelimiterInputFormat' outputformat 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';  hive> load data local inpath '/root/hivedata/bi.dat' into table t\_bi;  hive> select \* from t\_bi;  OK  01 zhangsan  02 lisi |

注：还需要在hive中使用add jar，才能在执行hql查询该表时把自定义jar包传递给maptask

hive>add jar /root/apps/hive/lib/myinput.jar

ss

## Hive实战

### Hive实战案例1——数据ETL

* 需求：
* 对web点击流日志基础数据表进行etl（按照仓库模型设计）
* 按各时间维度统计来源域名top10

已有数据表 “t\_orgin\_weblog” ：

|  |
| --- |
| +------------------+------------+----------+--+  | col\_name | data\_type | comment |  +------------------+------------+----------+--+  | valid | string | |  | remote\_addr | string | |  | remote\_user | string | |  | time\_local | string | |  | request | string | |  | status | string | |  | body\_bytes\_sent | string | |  | http\_referer | string | |  | http\_user\_agent | string | |  +------------------+------------+----------+--+ |

* 数据示例：

|  |
| --- |
| | true|1.162.203.134| - | 18/Sep/2013:13:47:35| /images/my.jpg | 200| 19939 | "http://www.angularjs.cn/A0d9" | "Mozilla/5.0 (Windows |  | true|1.202.186.37 | - | 18/Sep/2013:15:39:11| /wp-content/uploads/2013/08/windjs.png| 200| 34613 | "http://cnodejs.org/topic/521a30d4bee8d3cb1272ac0f" | "Mozilla/5.0 (Macintosh;| |

* 实现步骤：

1、对原始数据进行抽取转换

--将来访url分离出host path query query id

|  |
| --- |
| drop table if exists t\_etl\_referurl;  create table t\_etl\_referurl as  SELECT a.\*,b.\*  FROM t\_orgin\_weblog a LATERAL VIEW parse\_url\_tuple(regexp\_replace(http\_referer, "\"", ""), 'HOST', 'PATH','QUERY', 'QUERY:id') b as host, path, query, query\_id |

3、从前述步骤进一步分离出日期时间形成ETL明细表“t\_etl\_detail” day tm

|  |
| --- |
| drop table if exists t\_etl\_detail;  create table t\_etl\_detail as  select b.\*,substring(time\_local,0,11) as daystr,  substring(time\_local,13) as tmstr,  substring(time\_local,4,3) as month,  substring(time\_local,0,2) as day,  substring(time\_local,13,2) as hour  from t\_etl\_referurl b; |

3、对etl数据进行分区(包含所有数据的结构化信息)

|  |
| --- |
| drop table t\_etl\_detail\_prt;  create table t\_etl\_detail\_prt(  valid string,  remote\_addr string,  remote\_user string,  time\_local string,  request string,  status string,  body\_bytes\_sent string,  http\_referer string,  http\_user\_agent string,  host string,  path string,  query string,  query\_id string,  daystr string,  tmstr string,  month string,  day string,  hour string)  partitioned by (mm string,dd string); |

导入数据

|  |
| --- |
| insert into table t\_etl\_detail\_prt partition(mm='Sep',dd='18')  select \* from t\_etl\_detail where daystr='18/Sep/2013';  insert into table t\_etl\_detail\_prt partition(mm='Sep',dd='19')  select \* from t\_etl\_detail where daystr='19/Sep/2013'; |

分个时间维度统计各referer\_host的访问次数并排序

|  |
| --- |
| create table t\_refer\_host\_visit\_top\_tmp as  select referer\_host,count(\*) as counts,mm,dd,hh from t\_display\_referer\_counts group by hh,dd,mm,referer\_host order by hh asc,dd asc,mm asc,counts desc; |

4、来源访问次数topn各时间维度URL

取各时间维度的referer\_host访问次数topn

|  |
| --- |
| select \* from (select referer\_host,counts,concat(hh,dd),row\_number() over (partition by concat(hh,dd) order by concat(hh,dd) asc) as od from t\_refer\_host\_visit\_top\_tmp) t where od<=3; |

### Hive实战案例2——访问时长统计

* 需求：

从web日志中统计每日访客平均停留时间

* 实现步骤：

1. 由于要从大量请求中分辨出用户的各次访问，逻辑相对复杂，通过hive直接实现有困难，因此编写一个mr程序来求出访客访问信息（详见代码）

启动mr程序获取结果：

|  |
| --- |
| [hadoop@hdp-node-01 ~]$ hadoop jar weblog.jar cn.itcast.bigdata.hive.mr.UserStayTime /weblog/input /weblog/stayout |

1. 将mr的处理结果导入hive表

|  |
| --- |
| drop table t\_display\_access\_info\_tmp;  create table t\_display\_access\_info\_tmp(remote\_addr string,firt\_req\_time string,last\_req\_time string,stay\_long bigint)  row format delimited fields terminated by '\t';  load data inpath '/weblog/stayout4' into table t\_display\_access\_info\_tmp; |

3、得出访客访问信息表 "t\_display\_access\_info"

由于有一些访问记录是单条记录，mr程序处理处的结果给的时长是0，所以考虑给单次请求的停留时间一个默认市场30秒

|  |
| --- |
| drop table t\_display\_access\_info;  create table t\_display\_access\_info as  select remote\_addr,firt\_req\_time,last\_req\_time,  case stay\_long  when 0 then 30000  else stay\_long  end as stay\_long  from t\_display\_access\_info\_tmp; |

4、统计所有用户停留时间平均值

select avg(stay\_long) from t\_display\_access\_info;

### Hive实战案例3——级联求和accumulate

* 需求：

有如下访客访问次数统计表 t\_access\_times

|  |  |  |
| --- | --- | --- |
| 访客 | 月份 | 访问次数 |
| A | 2015-01 | 5 |
| A | 2015-01 | 15 |
| B | 2015-01 | 5 |
| A | 2015-01 | 8 |
| B | 2015-01 | 25 |
| A | 2015-01 | 5 |
| A | 2015-02 | 4 |
| A | 2015-02 | 6 |
| B | 2015-02 | 10 |
| B | 2015-02 | 5 |
| …… | …… | …… |

需要输出报表：t\_access\_times\_accumulate

|  |  |  |  |
| --- | --- | --- | --- |
| 访客 | 月份 | 月访问总计 | 累计访问总计 |
| A | 2015-01 | 33 | 33 |
| A | 2015-02 | 10 | 43 |
| ……. | ……. | ……. | ……. |
| B | 2015-01 | 30 | 30 |
| B | 2015-02 | 15 | 45 |
| ……. | ……. | ……. | ……. |

* 实现步骤

可以用一个hql语句即可实现：

|  |
| --- |
| select A.username,A.month,max(A.salary) as salary,sum(B.salary) as accumulate  from  (select username,month,sum(salary) as salary from t\_access\_times group by username,month) A  inner join  (select username,month,sum(salary) as salary from t\_access\_times group by username,month) B  on  A.username=B.username  where B.month <= A.month  group by A.username,A.month  order by A.username,A.month; |

# Flume

## Flume介绍

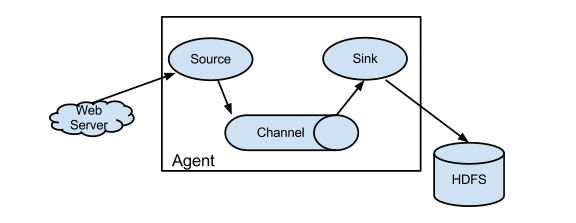
### 概述

* Flume是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。
* Flume可以采集文件，socket数据包等各种形式源数据，又可以将采集到的数据输出(下沉sink)到HDFS、hbase、hive、kafka等众多外部存储系统中
* 一般的采集需求，通过对flume的简单配置即可实现
* Flume针对特殊场景也具备良好的自定义扩展能力，

因此，flume可以适用于大部分的日常数据采集场景

### 运行机制

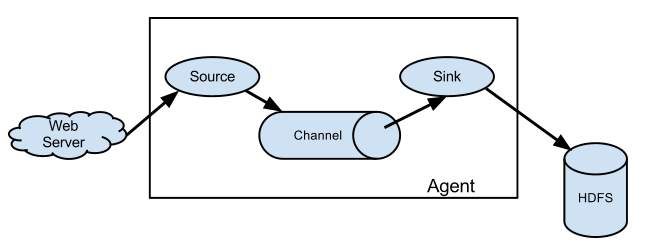
1. Flume分布式系统中最**核心的角色是agent**，flume采集系统就是由一个个agent所连接起来形成
2. **每一个agent相当于一个数据传递员(**Source 到 Channel 到 Sink之间传递数据的形式是Event事件；Event事件是一个数据流单元**)，内部有三个组件：**
   * 1. Source：采集源，用于跟数据源对接，以获取数据
     2. Sink：下沉地，采集数据的传送目的，用于往下一级agent传递数据或者往最终存储系统传递数据
     3. Channel：angent内部的数据传输通道，用于从source将数据传递到sink



### Flume采集系统结构图

#### 简单结构

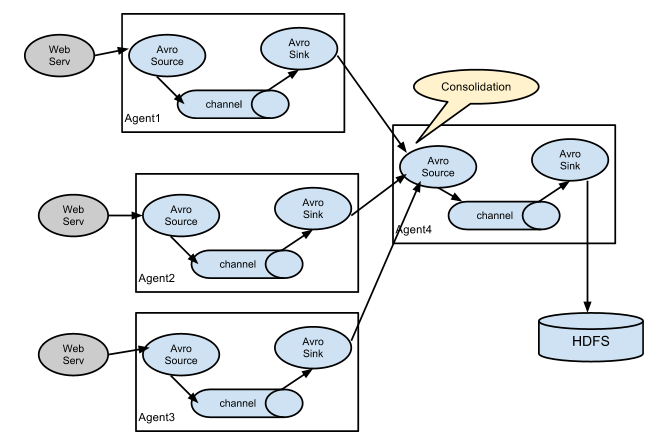
单个agent采集数据,直接就有一个agent在运行,agent运行在哪里,取决于source的实现.



#### 复杂结构

多级agent之间串联,多个agent,agent4可以先处理(转换)下数据然后下沉到hdfs中

agent4可能挂掉,所以可以给它配一个备机,准备切换.



## Flume实战案例

### Flume的安装部署

1. Flume的安装非常简单，只需要解压即可，当然，前提是已有hadoop环境

上传安装包到数据源所在节点上

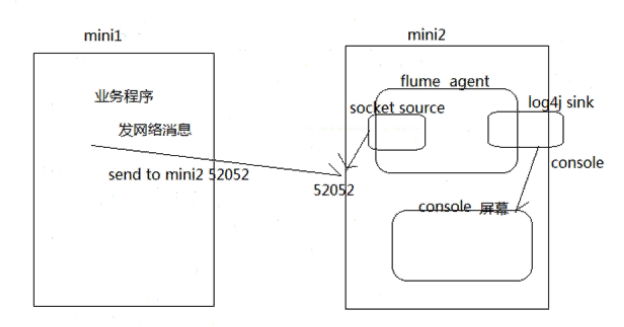
然后解压 tar -zxvf apache-flume-1.6.0-bin.tar.gz

然后进入flume的目录，修改conf下的flume-env.sh，在里面配置JAVA\_HOME

2、根据数据采集的需求**配置采集方案**，描述在配置文件中(文件名可任意自定义)

3、**指定采集方案配置文件**，在相应的节点上启动flume agent

先用一个最简单的例子来测试一下程序环境是否正常



1. 先在flume的conf目录(任意目录都行)下新建一个文件(采集方案)

vi netcat-logger.conf

|  |
| --- |
| # a1是该agent的名字  # 定义这个agent中各组件的名字  a1.sources = r1  a1.sinks = k1  a1.channels = c1  # 描述和配置source组件：r1  a1.sources.r1.type = netcat  a1.sources.r1.bind = localhost  a1.sources.r1.port = 44444  # 描述和配置sink组件：k1  a1.sinks.k1.type = logger  # 描述和配置channel组件，此处使用是内存缓存的方式  a1.channels.c1.type = memory  a1.channels.c1.capacity = 1000  a1.channels.c1.transactionCapacity = 100  # 描述和配置source channel sink之间的连接关系  a1.sources.r1.channels = c1  a1.sinks.k1.channel = c1 |

1. 启动agent去采集数据

|  |
| --- |
| bin/flume-ng agent -c conf -f conf/netcat-logger.conf -n a1 -Dflume.root.logger=INFO,console |

-c conf 指定flume自身的配置文件所在目录

-f conf/netcat-logger.con 指定我们所描述的采集方案

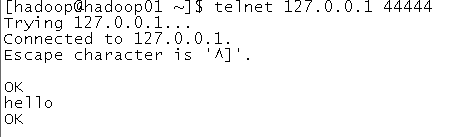
-n a1 指定我们这个agent的名字

1. 测试

先要往agent采集监听的端口上发送数据，让agent有数据可采

随便在一个能跟agent节点联网的机器上

telnet anget-hostname port （telnet mini2 44444）



### 采集案例

#### 采集文件到HDFS

* **从文件实时的读到另外一个目录**

采集需求：比如业务系统使用log4j生成的日志，日志内容不断增加，需要把追加到日志文件中的数据实时采集到hdfs

根据需求，首先定义以下3大要素

* 采集源，即source——监控文件内容更新 : exec ‘tail -F file’
* 下沉目标，即sink——HDFS文件系统 : hdfs sink
* Source和sink之间的传递通道——channel，可用file channel 也可以用 内存channel

配置文件编写：

|  |
| --- |
| agent1.sources = source1  agent1.sinks = sink1  agent1.channels = channel1  # Describe/configure tail -F source1  agent1.sources.source1.type = exec  agent1.sources.source1.command = tail -F /root/apps/logs/access\_log  agent1.sources.source1.channels = channel1  #configure host for source  agent1.sources.source1.interceptors = i1  agent1.sources.source1.interceptors.i1.type = host  agent1.sources.source1.interceptors.i1.hostHeader = hostname  # Describe sink1  agent1.sinks.sink1.type = hdfs  #a1.sinks.k1.channel = c1  agent1.sinks.sink1.hdfs.path =hdfs://sun01:9000/weblog/flume-collection/%y-%m-%d/%H-%M  agent1.sinks.sink1.hdfs.filePrefix = access\_log  agent1.sinks.sink1.hdfs.maxOpenFiles = 5000  agent1.sinks.sink1.hdfs.batchSize= 100  agent1.sinks.sink1.hdfs.fileType = DataStream  agent1.sinks.sink1.hdfs.writeFormat =Text  agent1.sinks.sink1.hdfs.rollSize = 102400  agent1.sinks.sink1.hdfs.rollCount = 1000000  agent1.sinks.sink1.hdfs.rollInterval = 60  agent1.sinks.sink1.hdfs.round = true  agent1.sinks.sink1.hdfs.roundValue = 10  agent1.sinks.sink1.hdfs.roundUnit = minute  agent1.sinks.sink1.hdfs.useLocalTimeStamp = true  # Use a channel which buffers events in memory  agent1.channels.channel1.type = memory  agent1.channels.channel1.keep-alive = 120  agent1.channels.channel1.capacity = 500000  agent1.channels.channel1.transactionCapacity = 600  # Bind the source and sink to the channel  agent1.sources.source1.channels = channel1  agent1.sinks.sink1.channel = channel1 |

* **从目录采集**

采集需求：某服务器的某特定目录下，会不断产生新的文件，每当有新文件出现，就需要把文件采集到HDFS中去

根据需求，首先定义以下3大要素

* 采集源，即source——监控文件目录 : spooldir
* 下沉目标，即sink——HDFS文件系统 : hdfs sink
* source和sink之间的传递通道——channel，可用file channel 也可以用内存channel

配置文件编写：

|  |
| --- |
| #定义三大组件的名称  agent1.sources = source1  agent1.sinks = sink1  agent1.channels = channel1  # 配置source组件  agent1.sources.source1.type = spooldir  agent1.sources.source1.spoolDir = /root/logs/  agent1.sources.source1.fileHeader = false  #配置拦截器  agent1.sources.source1.interceptors = i1  agent1.sources.source1.interceptors.i1.type = host  agent1.sources.source1.interceptors.i1.hostHeader = hostname  # 配置sink组件  agent1.sinks.sink1.type = hdfs  agent1.sinks.sink1.hdfs.path =hdfs://sun01:9000/weblog/flume-collection/%y-%m-%d/%H-%M  agent1.sinks.sink1.hdfs.filePrefix = access\_log  agent1.sinks.sink1.hdfs.maxOpenFiles = 5000  agent1.sinks.sink1.hdfs.batchSize= 100  agent1.sinks.sink1.hdfs.fileType = DataStream  agent1.sinks.sink1.hdfs.writeFormat =Text  agent1.sinks.sink1.hdfs.rollSize = 102400  agent1.sinks.sink1.hdfs.rollCount = 1000000  agent1.sinks.sink1.hdfs.rollInterval = 60  #agent1.sinks.sink1.hdfs.round = true  #agent1.sinks.sink1.hdfs.roundValue = 10  #agent1.sinks.sink1.hdfs.roundUnit = minute  agent1.sinks.sink1.hdfs.useLocalTimeStamp = true  # Use a channel which buffers events in memory  agent1.channels.channel1.type = memory  agent1.channels.channel1.keep-alive = 120  agent1.channels.channel1.capacity = 500000  agent1.channels.channel1.transactionCapacity = 600  # Bind the source and sink to the channel  agent1.sources.source1.channels = channel1  agent1.sinks.sink1.channel = channel1 |

Channel参数解释：

capacity：默认该通道中最大的可以存储的event数量

trasactionCapacity：每次最大可以从source中拿到或者送到sink中的event数量

keep-alive：event添加到通道中或者移出的允许时间

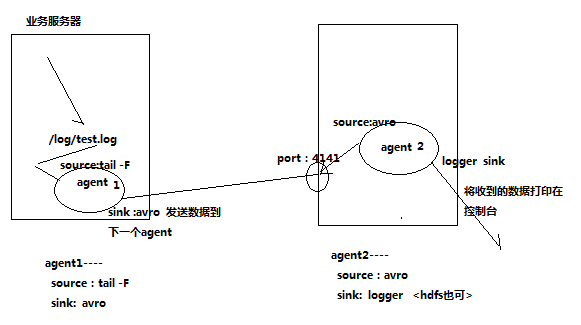
### 更多source和sink组件

Flume支持众多的source和sink类型，详细手册可参考官方文档

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

### 多个agent之间连接

使用avro source和avro sink就可以



### 高可用Flum-NG配置案例

在完成单点的Flume NG搭建后，下面我们搭建一个高可用的Flume NG集群，架构图如下所示：



　　图中，我们可以看出，Flume的存储可以支持多种，这里只列举了HDFS和Kafka（如：存储最新的一周日志，并给Storm系统提供实时日志流）。

#### 角色分配

Flume的Agent和Collector分布如下表所示：

|  |  |  |
| --- | --- | --- |
| 名称 | HOST | 角色 |
| Agent1 | mini1 | Web Server |
| Agent2 | mini2 | Web Server |
| Agent3 | mini3 | Web Server |
| Collector1 | mini4 | AgentMstr1 |
| Collector2 | mini5 | AgentMstr2 |

　　图中所示，Agent1，Agent2，Agent3数据分别流入到Collector1和Collector2，Flume NG本身提供了Failover机制，可以自动切换和恢复。在上图中，有3个产生日志服务器分布在不同的机房，要把所有的日志都收集到一个集群中存储。下 面我们开发配置Flume NG集群

#### 配置

在下面单点Flume中，基本配置都完成了，我们只需要新添加两个配置文件，它们是agent.properties和collector.properties，其配置内容如下所示：

1、agent配置

|  |
| --- |
| [root@mini1 apache-flume-1.6.0-bin]# vi conf/agent.properties  #agent1 name  agent1.channels = c1  agent1.sources = r1  agent1.sinks = k1 k2  #set gruop  agent1.sinkgroups = g1  #set channel  agent1.channels.c1.type = memory  agent1.channels.c1.capacity = 1000  agent1.channels.c1.transactionCapacity = 100  agent1.sources.r1.channels = c1  agent1.sources.r1.type = exec  agent1.sources.r1.command = tail -F /root/log/test.log  agent1.sources.r1.interceptors = i1 i2  agent1.sources.r1.interceptors.i1.type = static  agent1.sources.r1.interceptors.i1.key = Type  agent1.sources.r1.interceptors.i1.value = LOGIN  agent1.sources.r1.interceptors.i2.type = timestamp  # set sink1  agent1.sinks.k1.channel = c1  agent1.sinks.k1.type = avro  agent1.sinks.k1.hostname = mini2  agent1.sinks.k1.port = 52020  # set sink2  agent1.sinks.k2.channel = c1  agent1.sinks.k2.type = avro  agent1.sinks.k2.hostname = mini3  agent1.sinks.k2.port = 52020  #set sink group  agent1.sinkgroups.g1.sinks = k1 k2  #set failover  agent1.sinkgroups.g1.processor.type = failover  agent1.sinkgroups.g1.processor.priority.k1 = 10  agent1.sinkgroups.g1.processor.priority.k2 = 1  agent1.sinkgroups.g1.processor.maxpenalty = 10000 |

启动命令：

|  |
| --- |
| bin/flume-ng agent -n agent1 -c conf -f conf/agent.properties -Dflume.root.logger=DEBUG,console |

2、collector配置

|  |
| --- |
| [root@mini2 conf]# vi collector.properties  #set Agent name  a1.sources = r1  a1.channels = c1  a1.sinks = k1  #set channel  a1.channels.c1.type = memory  a1.channels.c1.capacity = 1000  a1.channels.c1.transactionCapacity = 100  # other node,nna to nns  a1.sources.r1.type = avro  a1.sources.r1.bind = mini2  a1.sources.r1.port = 52020  a1.sources.r1.interceptors = i1  a1.sources.r1.interceptors.i1.type = static  a1.sources.r1.interceptors.i1.key = Collector  a1.sources.r1.interceptors.i1.value = mini2  a1.sources.r1.channels = c1  #set sink to hdfs  a1.sinks.k1.type=hdfs  a1.sinks.k1.hdfs.path=/home/hdfs/flume/logdfs  a1.sinks.k1.hdfs.fileType=DataStream  a1.sinks.k1.hdfs.writeFormat=TEXT  a1.sinks.k1.hdfs.rollInterval=10  a1.sinks.k1.channel=c1  a1.sinks.k1.hdfs.filePrefix=%Y-%m-%d |

在mini3上，需要修改上述配置中的红色字体主机名为mini3

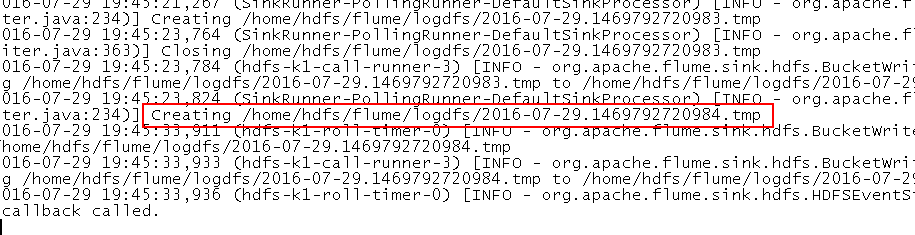
启动命令：

|  |
| --- |
| bin/flume-ng agent -n a1 -c conf -f conf/collector.properties -Dflume.root.logger=DEBUG,console |

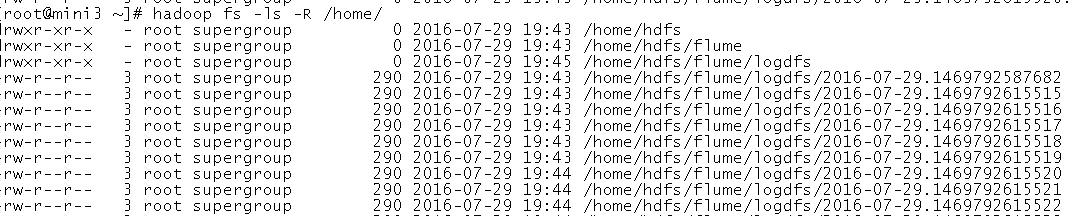
#### FAILOVER测试

下面我们来测试下Flume NG集群的高可用（故障转移）。场景如下：我们在Agent1节点上传文件，由于我们配置Collector1的权重比Collector2大，所以 Collector1优先采集并上传到存储系统。然后我们kill掉Collector1，此时有Collector2负责日志的采集上传工作，之后，我 们手动恢复Collector1节点的Flume服务，再次在Agent1上次文件，发现Collector1恢复优先级别的采集工作。具体截图如下所 示：

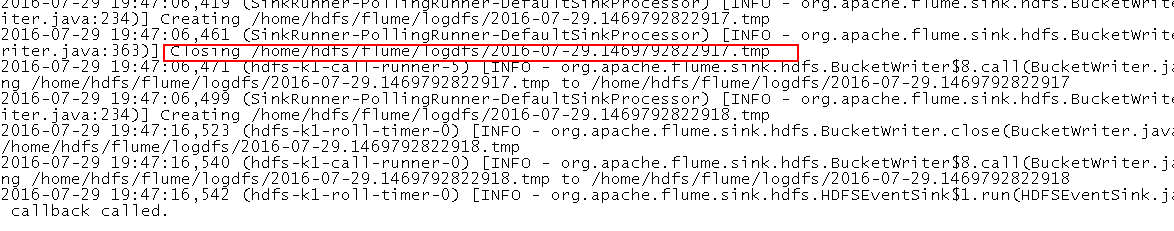
Collector1优先上传



HDFS集群中上传的log内容预览



Collector1宕机，Collector2获取优先上传权限



重启Collector1服务，Collector1重新获得优先上传的权限

# azkaban

## 概述

### 为什么需要工作流调度系统

* 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

* 各任务单元之间存在时间先后及前后依赖关系
* 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先将原始数据同步到HDFS上；
2. 借助MapReduce计算框架对原始数据进行转换，生成的数据以分区表的形式存储到多张Hive表中；
3. 需要对Hive中多个表的数据进行JOIN处理，得到一个明细数据Hive大表；
4. 将明细数据进行复杂的统计分析，得到结果报表信息；
5. 需要将统计分析得到的结果数据同步到业务系统中，供业务调用使用。

### 工作流调度实现方式

简单的任务调度：直接使用linux的crontab来定义；

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

### 常见工作流调度系统

市面上目前有许多工作流调度器

在hadoop领域，常见的工作流调度器有Oozie, Azkaban,Cascading,Hamake等

### 各种调度工具特性对比

下面的表格对上述四种hadoop工作流调度器的关键特性进行了比较，尽管这些工作流调度器能够解决的需求场景基本一致，但在设计理念，目标用户，应用场景等方面还是存在显著的区别，在做技术选型的时候，可以提供参考

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | Hamake | Oozie | Azkaban | Cascading |
| 工作流描述语言 | XML | XML (xPDL based) | text file with key/value pairs | Java API |
| 依赖机制 | data-driven | explicit | explicit | explicit |
| 是否要web容器 | No | Yes | Yes | No |
| 进度跟踪 | console/log messages | web page | web page | Java API |
| Hadoop job调度支持 | no | yes | yes | yes |
| 运行模式 | command line utility | daemon | daemon | API |
| Pig支持 | yes | yes | yes | yes |
| 事件通知 | no | no | no | yes |
| 需要安装 | no | yes | yes | no |
| 支持的hadoop版本 | 0.18+ | 0.20+ | currently unknown | 0.18+ |
| 重试支持 | no | workflownode evel | yes | yes |
| 运行任意命令 | yes | yes | yes | yes |
| Amazon EMR支持 | yes | no | currently unknown | yes |

### Azkaban与Oozie对比

对市面上最流行的两种调度器，给出以下详细对比，以供技术选型参考。总体来说，ooize相比azkaban是一个重量级的任务调度系统，功能全面，但配置使用也更复杂。如果可以不在意某些功能的缺失，轻量级调度器azkaban是很不错的候选对象。

详情如下：

* 功能

两者均可以调度mapreduce,pig,java,脚本工作流任务

两者均可以定时执行工作流任务

* 工作流定义

Azkaban使用Properties文件定义工作流

Oozie使用XML文件定义工作流

* 工作流传参

Azkaban支持直接传参，例如${input}

Oozie支持参数和EL表达式，例如${fs:dirSize(myInputDir)}

* 定时执行

Azkaban的定时执行任务是基于时间的

Oozie的定时执行任务基于时间和输入数据

* 资源管理

Azkaban有较严格的权限控制，如用户对工作流进行读/写/执行等操作

Oozie暂无严格的权限控制

* 工作流执行

Azkaban有两种运行模式，分别是solo server mode(executor server和web server部署在同一台节点)和multi server mode(executor server和web server可以部署在不同节点)

Oozie作为工作流服务器运行，支持多用户和多工作流

* 工作流管理

Azkaban支持浏览器以及ajax方式操作工作流

Oozie支持命令行、HTTP REST、Java API、浏览器操作工作流

## Azkaban介绍

Azkaban是由Linkedin开源的一个批量工作流任务调度器。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。

Azkaban定义了一种KV文件(properties)格式来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。

它有如下功能特点：

* Web用户界面
* 方便上传工作流
* 方便设置任务之间的关系
* 调度工作流
* 认证/授权(权限的工作)
* 能够杀死并重新启动工作流
* 模块化和可插拔的插件机制
* 项目工作区
* 工作流和任务的日志记录和审计

## Azkaban安装部署

### 准备工作

Azkaban Web服务器

azkaban-web-server-2.5.0.tar.gz

Azkaban执行服务器

azkaban-executor-server-2.5.0.tar.gz

MySQL

目前azkaban只支持 mysql,需安装mysql服务器,本文档中默认已安装好mysql服务器,并建立了 root用户,密码 root.

下载地址:http://azkaban.github.io/downloads.html

### 安装

将安装文件上传到集群,最好上传到安装 hive、sqoop的机器上,方便命令的执行

在当前用户目录下新建 azkabantools目录,用于存放源安装文件.新建azkaban目录,用于存放azkaban运行程序

### azkaban web服务器安装

解压azkaban-web-server-2.5.0.tar.gz

命令: tar –zxvf azkaban-web-server-2.5.0.tar.gz

将解压后的azkaban-web-server-2.5.0 移动到 azkaban目录中,并重新命名 webserver

命令: mv azkaban-web-server-2.5.0 ../azkaban

        cd ../azkaban

        mv azkaban-web-server-2.5.0 webserver

### azkaban 执行服器安装

解压azkaban-executor-server-2.5.0.tar.gz

命令:tar –zxvf azkaban-executor-server-2.5.0.tar.gz

将解压后的azkaban-executor-server-2.5.0 移动到 azkaban目录中,并重新命名 executor

命令:mv azkaban-executor-server-2.5.0  ../azkaban

cd ../azkaban

mv azkaban-executor-server-2.5.0  executor

azkaban脚本导入

解压: azkaban-sql-script-2.5.0.tar.gz

命令:tar –zxvf azkaban-sql-script-2.5.0.tar.gz

将解压后的mysql 脚本,导入到mysql中:

进入mysql

mysql> create database azkaban;

mysql> use azkaban;

Database changed

mysql> source /home/hadoop/azkaban-2.5.0/create-all-sql-2.5.0.sql;

### 创建SSL配置

参考地址: http://docs.codehaus.org/display/JETTY/How+to+configure+SSL

命令: keytool -keystore keystore -alias jetty -genkey -keyalg RSA

运行此命令后,会提示输入当前生成 keystor的密码及相应信息,输入的密码请劳记,信息如下:

输入keystore密码：

再次输入新密码:

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]：

您所在的州或省份名称是什么？

  [Unknown]：

该单位的两字母国家代码是什么

  [Unknown]：  CN

CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=CN 正确吗？

  [否]：  y

输入<jetty>的主密码

        （如果和 keystore 密码相同，按回车）：

再次输入新密码:

完成上述工作后,将在当前目录生成 keystore 证书文件,将keystore 拷贝到 azkaban web服务器根目录中.如:cp keystore azkaban/webserver

### 配置文件

注：先配置好服务器节点上的时区

1. 先生成时区配置文件Asia/Shanghai，用交互式命令 tzselect 即可
2. 拷贝该时区文件，覆盖系统本地时区配置

cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

azkaban web服务器配置

进入azkaban web服务器安装目录 conf目录

* 修改azkaban.properties文件

命令vi azkaban.properties

内容说明如下:

|  |
| --- |
| #Azkaban Personalization Settings  azkaban.name=Test                           #服务器UI名称,用于服务器上方显示的名字  azkaban.label=My Local Azkaban                               #描述  azkaban.color=#FF3601                                                 #UI颜色  azkaban.default.servlet.path=/index                         #  web.resource.dir=web/                                                 #默认根web目录  default.timezone.id=Asia/Shanghai                           #默认时区,已改为亚洲/上海 默认为美国    #Azkaban UserManager class  user.manager.class=azkaban.user.XmlUserManager   #用户权限管理默认类  user.manager.xml.file=conf/azkaban-users.xml              #用户配置,具体配置参加下文    #Loader for projects  executor.global.properties=conf/global.properties    # global配置文件所在位置  azkaban.project.dir=projects                                                #    database.type=mysql                                                              #数据库类型  mysql.port=3306                                                                     #端口号  mysql.host=mini3                                                   #数据库连接IP  mysql.database=azkaban                                                       #数据库实例名  mysql.user=root                                                                 #数据库用户名  mysql.password=root                                                          #数据库密码  mysql.numconnections=100                                                  #最大连接数    # Velocity dev mode  velocity.dev.mode=false  # Jetty服务器属性.  jetty.maxThreads=25                                                               #最大线程数  jetty.ssl.port=8443                                                                   #Jetty SSL端口  jetty.port=8081                                                                         #Jetty端口  jetty.keystore=keystore                                                          #SSL文件名  jetty.password=123456                                                             #SSL文件密码  jetty.keypassword=123456                                                      #Jetty主密码 与 keystore文件相同  jetty.truststore=keystore                                                                #SSL文件名  jetty.trustpassword=123456                                                   # SSL文件密码    # 执行服务器属性  executor.port=12321                                                               #执行服务器端口    # 邮件设置  mail.sender=xxxxxxxx@163.com                                       #发送邮箱  mail.host=smtp.163.com                                                       #发送邮箱smtp地址  mail.user=xxxxxxxx                                       #发送邮件时显示的名称  mail.password=\*\*\*\*\*\*\*\*\*\*                                                 #邮箱密码  job.failure.email=xxxxxxxx@163.com                              #任务失败时发送邮件的地址  job.success.email=xxxxxxxx@163.com                            #任务成功时发送邮件的地址  lockdown.create.projects=false                                           #  cache.directory=cache                                                            #缓存目录 |

* 用户配置

进入azkaban web服务器conf目录,修改azkaban-users.xml

vi azkaban-users.xml 增加 管理员用户

|  |
| --- |
| <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> |

* azkaban 执行服务器配置

进入执行服务器安装目录conf,修改azkaban.properties

vi azkaban.properties

|  |
| --- |
| #Azkaban  default.timezone.id=Asia/Shanghai                                              #时区    # Azkaban JobTypes 插件配置  azkaban.jobtype.plugin.dir=plugins/jobtypes                   #jobtype 插件所在位置    #Loader for projects  executor.global.properties=conf/global.properties  azkaban.project.dir=projects    #数据库设置  database.type=mysql                                                                       #数据库类型(目前只支持mysql)  mysql.port=3306                                                                                #数据库端口号  mysql.host=mini3                                                           #数据库IP地址  mysql.database=azkaban                                                                #数据库实例名  mysql.user=root                                                                         #数据库用户名  mysql.password=root                                                                   #数据库密码  mysql.numconnections=100                                                           #最大连接数    # 执行服务器配置  executor.maxThreads=50                                                                #最大线程数  executor.port=12321                                                               #端口号(如修改,请与web服务中一致)  executor.flow.threads=30                                                                #线程数 |

## 启动

**1**、**web服务器**

在azkaban web服务器目录下执行启动命令

bin/azkaban-web-start.sh

注:在web服务器根目录运行

**2**、**执行服务器**

在执行服务器目录下执行启动命令

bin/azkaban-executor-start.sh ./

注:只能要执行服务器根目录运行

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.在登录中输入刚才新的户用名及密码,点击 login.

## Azkaban实战

Azkaba内置的任务类型支持command、java

### Command类型单一job示例

1. 创建job描述文件

vi command.job

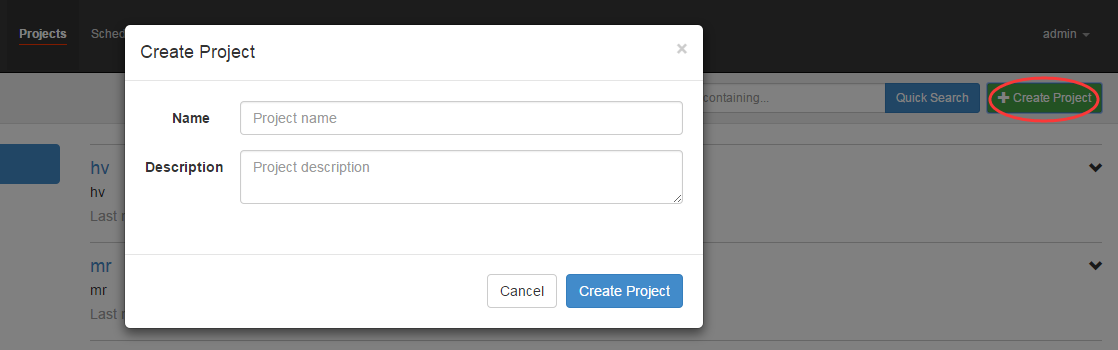
|  |
| --- |
| #command.job  type=command  command=echo 'hello' |

1. 将job资源文件打包成zip文件

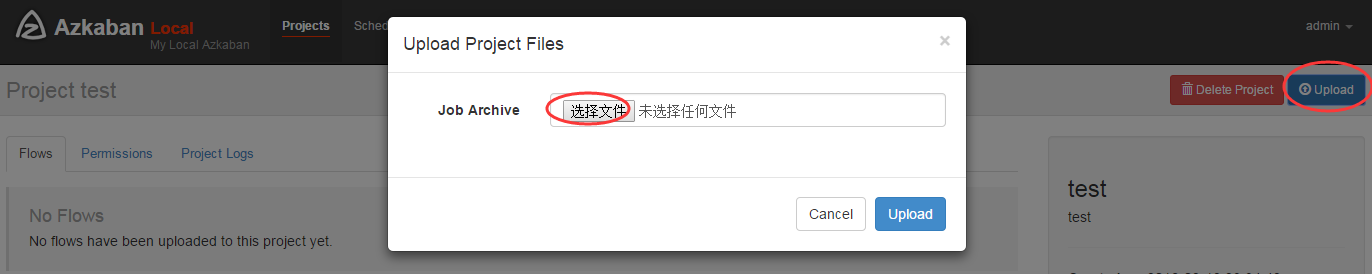
zip command.job

1. 通过azkaban的web管理平台创建project并上传job压缩包

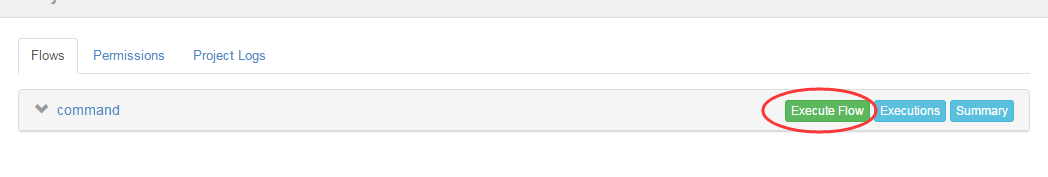
首先创建project



上传zip包



4、启动执行该job



### Command类型多job工作流flow

1. 创建有依赖关系的多个job描述

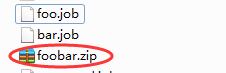
第一个job：foo.job

|  |
| --- |
| # foo.job  type=command  command=echo foo |

第二个job：bar.job依赖foo.job

|  |
| --- |
| # bar.job  type=command  dependencies=foo  command=echo bar |

1. 将所有job资源文件打到一个zip包中



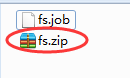
1. 在azkaban的web管理界面创建工程并上传zip包
2. 启动工作流flow

### HDFS操作任务

1. 创建job描述文件

|  |
| --- |
| # fs.job  type=command  command=/home/hadoop/apps/hadoop-2.6.1/bin/hadoop fs -mkdir /azaz |

1. 将job资源文件打包成zip文件



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

### MAPREDUCE任务

Mr任务依然可以使用command的job类型来执行

1. 创建job描述文件，及mr程序jar包（示例中直接使用hadoop自带的example jar）

|  |
| --- |
| # mrwc.job  type=command  command=/home/hadoop/apps/hadoop-2.6.1/bin/hadoop jar hadoop-mapreduce-examples-2.6.1.jar wordcount /wordcount/input /wordcount/azout |

1. 将所有job资源文件打到一个zip包中



3、在azkaban的web管理界面创建工程并上传zip包

4、启动job

### HIVE脚本任务

* 创建job描述文件和hive脚本

Hive脚本： test.sql

|  |
| --- |
| use default;  drop table aztest;  create table aztest(id int,name string) row format delimited fields terminated by ',';  load data inpath '/aztest/hiveinput' into table aztest;  create table azres as select \* from aztest;  insert overwrite directory '/aztest/hiveoutput' select count(1) from aztest; |

Job描述文件：hivef.job

|  |
| --- |
| # hivef.job  type=command  command=/home/hadoop/apps/hive/bin/hive -f 'test.sql' |

2、将所有job资源文件打到一个zip包中

3、在azkaban的web管理界面创建工程并上传zip包

4、启动job

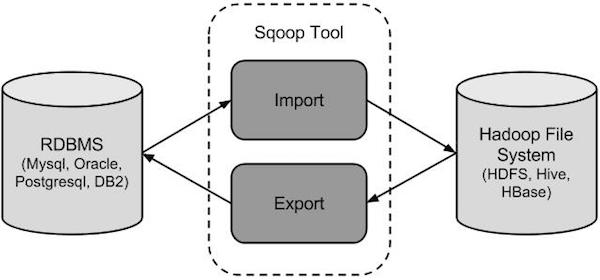
# sqoop

## 概述

sqoop是apache旗下一款**“Hadoop和关系数据库服务器之间传送数据”**的工具。

**导入数据**：MySQL，Oracle导入数据到Hadoop的HDFS、HIVE、HBASE等数据存储系统；

**导出数据：**从Hadoop的文件系统中导出数据到关系数据库



## 工作机制 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

Sqoop的原理其实就是将导入导出命令转化为mapreduce程序来执行，sqoop在接收到命令后，都要生成mapreduce程序

使用sqoop的代码生成工具可以方便查看到sqoop所生成的java代码，并可在此基础之上进行深入定制开发

## sqoop安装

安装sqoop的前提是已经具备java和hadoop的环境

1、下载并解压

最新版下载地址http://ftp.wayne.edu/apache/sqoop/1.4.6/

2、修改配置文件

$ cd $SQOOP\_HOME/conf

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

打开sqoop-env.sh并编辑下面几行：

export HADOOP\_COMMON\_HOME=/home/hadoop/apps/hadoop-2.6.1/

export HADOOP\_MAPRED\_HOME=/home/hadoop/apps/hadoop-2.6.1/

export HIVE\_HOME=/home/hadoop/apps/hive-1.2.1

3、加入mysql的jdbc驱动包

cp ~/app/hive/lib/mysql-connector-java-5.1.28.jar $SQOOP\_HOME/lib/

4、验证启动

$ cd $SQOOP\_HOME/bin

$ sqoop-version

预期的输出：

15/12/17 14:52:32 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6

Sqoop 1.4.6 git commit id 5b34accaca7de251fc91161733f906af2eddbe83

Compiled by abe on Fri Aug 1 11:19:26 PDT 2015

到这里，整个Sqoop安装工作完成。

## sqoop实战

### Sqoop的数据导入

“导入工具”导入单个表从RDBMS到HDFS。表中的每一行被视为HDFS的记录。所有记录都存储为文本文件的文本数据（或者Avro、sequence文件等二进制数据）

* 语法

下面的语法用于将数据导入HDFS。

|  |
| --- |
| $ sqoop import (generic-args) (import-args) |

* 示例

1、表数据

在mysql中有一个库userdb中三个表：emp, emp\_add和emp\_contact

表emp:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **name** | **deg** | **salary** | **dept** |
| 1201 | gopal | manager | 50,000 | TP |
| 1202 | manisha | Proof reader | 50,000 | TP |
| 1203 | khalil | php dev | 30,000 | AC |
| 1204 | prasanth | php dev | 30,000 | AC |
| 1205 | kranthi | admin | 20,000 | TP |

表emp\_add:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **hno** | **street** | **city** |
| 1201 | 288A | vgiri | jublee |
| 1202 | 108I | aoc | sec-bad |
| 1203 | 144Z | pgutta | hyd |
| 1204 | 78B | old city | sec-bad |
| 1205 | 720X | hitec | sec-bad |

表emp\_conn:

|  |  |  |
| --- | --- | --- |
| **id** | **phno** | **email** |
| 1201 | 2356742 | gopal@tp.com |
| 1202 | 1661663 | manisha@tp.com |
| 1203 | 8887776 | khalil@ac.com |
| 1204 | 9988774 | prasanth@ac.com |
| 1205 | 1231231 | kranthi@tp.com |

2、导入表表数据到HDFS

下面的命令用于从MySQL数据库服务器中的emp表导入HDFS。

|  |
| --- |
| $bin/sqoop import \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --table emp --m 1 |

如果成功执行，那么会得到下面的输出。

|  |
| --- |
| 14/12/22 15:24:54 INFO sqoop.Sqoop: Running Sqoop version: 1.4.5  14/12/22 15:24:56 INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.  INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-hadoop/compile/cebe706d23ebb1fd99c1f063ad51ebd7/emp.jar  -----------------------------------------------------  O mapreduce.Job: map 0% reduce 0%  14/12/22 15:28:08 INFO mapreduce.Job: map 100% reduce 0%  14/12/22 15:28:16 INFO mapreduce.Job: Job job\_1419242001831\_0001 completed successfully  -----------------------------------------------------  -----------------------------------------------------  14/12/22 15:28:17 INFO mapreduce.ImportJobBase: Transferred 145 bytes in 177.5849 seconds (0.8165 bytes/sec)  14/12/22 15:28:17 INFO mapreduce.ImportJobBase: Retrieved 5 records. |

为了验证在HDFS导入的数据，请使用以下命令查看导入的数据

|  |
| --- |
| $ $HADOOP\_HOME/bin/hadoop fs -cat /user/hadoop/emp/part-m-00000 |

emp表的数据和字段之间用逗号(,)表示。

|  |
| --- |
| 1201, gopal, manager, 50000, TP  1202, manisha, preader, 50000, TP  1203, kalil, php dev, 30000, AC  1204, prasanth, php dev, 30000, AC  1205, kranthi, admin, 20000, TP |

3、导入关系表到HIVE

|  |
| --- |
| bin/sqoop import --connect jdbc:mysql://hdp-node-01:3306/test --username root --password root --table emp --hive-import --m 1 |

4、导入到HDFS指定目录

在导入表数据到HDFS使用Sqoop导入工具，我们可以指定目标目录。

以下是指定目标目录选项的Sqoop导入命令的语法。

|  |
| --- |
| --target-dir <new or exist directory in HDFS> |

下面的命令是用来导入emp\_add表数据到'/queryresult'目录。

|  |
| --- |
| bin/sqoop import \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --target-dir /queryresult \  --table emp --m 1 |

下面的命令是用来验证 /queryresult 目录中 emp\_add表导入的数据形式。

|  |
| --- |
| $HADOOP\_HOME/bin/hadoop fs -cat /queryresult/part-m-\* |

它会用逗号（，）分隔emp\_add表的数据和字段。

|  |
| --- |
| 1201, 288A, vgiri, jublee  1202, 108I, aoc, sec-bad  1203, 144Z, pgutta, hyd  1204, 78B, oldcity, sec-bad  1205, 720C, hitech, sec-bad |

5、导入表数据子集

我们可以导入表的使用Sqoop导入工具，"where"子句的一个子集。它执行在各自的数据库服务器相应的SQL查询，并将结果存储在HDFS的目标目录。

where子句的语法如下。

|  |
| --- |
| --where <condition> |

下面的命令用来导入emp\_add表数据的子集。子集查询检索员工ID和地址，居住城市为：Secunderabad

|  |
| --- |
| bin/sqoop import \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --where "city ='sec-bad'" \  --target-dir /wherequery \  --table emp\_add --m 1 |

按需导入

|  |
| --- |
| bin/sqoop import \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --target-dir /wherequery2 \  --query 'select id,name,deg from emp WHERE id>1207 and $CONDITIONS' \  --split-by id \  --fields-terminated-by '\t' \  --m 1 |

下面的命令用来验证数据从emp\_add表导入/wherequery目录

|  |
| --- |
| $HADOOP\_HOME/bin/hadoop fs -cat /wherequery/part-m-\* |

它用逗号（，）分隔 emp\_add表数据和字段。

|  |
| --- |
| 1202, 108I, aoc, sec-bad  1204, 78B, oldcity, sec-bad  1205, 720C, hitech, sec-bad |

6、增量导入

增量导入是仅导入新添加的表中的行的技术。

它需要添加‘incremental’, ‘check-column’, 和 ‘last-value’选项来执行增量导入。

下面的语法用于Sqoop导入命令增量选项。

|  |
| --- |
| --incremental <mode>  --check-column <column name>  --last value <last check column value> |

假设新添加的数据转换成emp表如下：

1206, satish p, grp des, 20000, GR

下面的命令用于在EMP表执行增量导入。

|  |
| --- |
| bin/sqoop import \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --table emp --m 1 \  --incremental append \  --check-column id \  --last-value 1205 |

以下命令用于从emp表导入HDFS emp/ 目录的数据验证。

|  |
| --- |
| $ $HADOOP\_HOME/bin/hadoop fs -cat /user/hadoop/emp/part-m-\*  它用逗号（，）分隔 emp\_add表数据和字段。  1201, gopal, manager, 50000, TP  1202, manisha, preader, 50000, TP  1203, kalil, php dev, 30000, AC  1204, prasanth, php dev, 30000, AC  1205, kranthi, admin, 20000, TP  1206, satish p, grp des, 20000, GR |

下面的命令是从表emp 用来查看修改或新添加的行

|  |
| --- |
| $ $HADOOP\_HOME/bin/hadoop fs -cat /emp/part-m-\*1  这表示新添加的行用逗号（，）分隔emp表的字段。  1206, satish p, grp des, 20000, GR |

### Sqoop的数据导出

将数据从HDFS导出到RDBMS数据库

导出前，目标表必须存在于目标数据库中。

* 默认操作是从将文件中的数据使用INSERT语句插入到表中
* 更新模式下，是生成UPDATE语句更新表数据
* 语法

以下是export命令语法。

|  |
| --- |
| $ sqoop export (generic-args) (export-args) |

* 示例

数据是在HDFS 中“EMP/”目录的emp\_data文件中。所述emp\_data如下：

|  |
| --- |
| 1201, gopal, manager, 50000, TP  1202, manisha, preader, 50000, TP  1203, kalil, php dev, 30000, AC  1204, prasanth, php dev, 30000, AC  1205, kranthi, admin, 20000, TP  1206, satish p, grp des, 20000, GR |

1、首先需要手动创建mysql中的目标表

|  |
| --- |
| $ mysql  mysql> USE db;  mysql> CREATE TABLE employee (  id INT NOT NULL PRIMARY KEY,  name VARCHAR(20),  deg VARCHAR(20),  salary INT,  dept VARCHAR(10)); |

2、然后执行导出命令

|  |
| --- |
| bin/sqoop export \  --connect jdbc:mysql://hdp-node-01:3306/test \  --username root \  --password root \  --table emp2 \  --export-dir /user/hadoop/emp/ |

3、验证表mysql命令行。

|  |
| --- |
| mysql>select \* from employee;  如果给定的数据存储成功，那么可以找到数据在如下的employee表。  +------+--------------+-------------+-------------------+--------+  | Id | Name | Designation | Salary | Dept |  +------+--------------+-------------+-------------------+--------+  | 1201 | gopal | manager | 50000 | TP |  | 1202 | manisha | preader | 50000 | TP |  | 1203 | kalil | php dev | 30000 | AC |  | 1204 | prasanth | php dev | 30000 | AC |  | 1205 | kranthi | admin | 20000 | TP |  | 1206 | satish p | grp des | 20000 | GR |  +------+--------------+-------------+-------------------+--------+ |

## Sqoop作业

*注：Sqoop作业——将事先定义好的数据导入导出任务按照指定流程运行*

*很少用*

### 语法

以下是创建Sqoop作业的语法。

|  |
| --- |
| $ sqoop job (generic-args) (job-args)  [-- [subtool-name] (subtool-args)]  $ sqoop-job (generic-args) (job-args)  [-- [subtool-name] (subtool-args)] |

### 创建作业(--create)

在这里，我们创建一个名为myjob，这可以从RDBMS表的数据导入到HDFS作业。

|  |
| --- |
| bin/sqoop job --create myimportjob -- import --connect jdbc:mysql://hdp-node-01:3306/test --username root --password root --table emp --m 1 |

该命令创建了一个从db库的employee表导入到HDFS文件的作业。

### 验证作业 (--list)

**‘--list’**参数是用来验证保存的作业。下面的命令用来验证保存Sqoop作业的列表。

$ sqoop job --list

它显示了保存作业列表。

Available jobs:

myjob

检查作业(--show)

**‘--show’**参数用于检查或验证特定的工作，及其详细信息。以下命令和样本输出用来验证一个名为myjob的作业。

$ sqoop job --show myjob

它显示了工具和它们的选择，这是使用在myjob中作业情况。

|  |
| --- |
| Job: myjob  Tool: import Options:  ----------------------------  direct.import = true  codegen.input.delimiters.record = 0  hdfs.append.dir = false  db.table = employee  ...  incremental.last.value = 1206  ... |

### 执行作业 (--exec)

**‘--exec’**选项用于执行保存的作业。下面的命令用于执行保存的作业称为myjob。

|  |
| --- |
| $ sqoop job --exec myjob  它会显示下面的输出。  10/08/19 13:08:45 INFO tool.CodeGenTool: Beginning code generation  ... |

## 代码定制

以下是Sqoop代码生成命令的语法：

|  |
| --- |
| $ sqoop-codegen (generic-args) (codegen-args)  $ sqoop-codegen (generic-args) (codegen-args) |

示例：以USERDB数据库中的表emp来生成Java代码为例。

下面的命令用来生成导入

|  |
| --- |
| $ sqoop-codegen \  --import  --connect jdbc:mysql://localhost/userdb \  --username root \  --table emp |

如果命令成功执行，那么它就会产生如下的输出。

|  |
| --- |
| 14/12/23 02:34:40 INFO sqoop.Sqoop: Running Sqoop version: 1.4.5  14/12/23 02:34:41 INFO tool.CodeGenTool: Beginning code generation  ……………….  14/12/23 02:34:42 INFO orm.CompilationManager: HADOOP\_MAPRED\_HOME is /usr/local/hadoop  Note: /tmp/sqoop-hadoop/compile/9a300a1f94899df4a9b10f9935ed9f91/emp.java uses or overrides a deprecated API.  Note: Recompile with -Xlint:deprecation for details.  14/12/23 02:34:47 INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-hadoop/compile/9a300a1f94899df4a9b10f9935ed9f91/emp.jar |

验证: 查看输出目录下的文件

|  |
| --- |
| $ cd /tmp/sqoop-hadoop/compile/9a300a1f94899df4a9b10f9935ed9f91/  $ ls  emp.class  emp.jar  emp.java |

如果想做深入定制导出，则可修改上述代码文件