

# Hadoop 2.X 管理与开发

## 一、Hadoop 的起源与背景知识

### (一) 什么是大数据

**大数据（Big Data）**，指无法在一定时间范围内用常规软件工具进行捕捉、管理和处理的数据集合，是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。

大数据的 5 个特征（IBM 提出）：

- Volume （大量）
- Velocity （高速）
- Variety （多样）
- Value （价值）
- Veracity （真实性）

大数据的典型案例：

- 电商网站的商品推荐



- 基于大数据的天气预报



## (二) OLTP 与 OLAP

- OLTP: **On-Line Transaction Processing (联机事务处理过程)**。也称为面向交易的处理过程，其基本特征是前台接收的用户数据可以立即传送到计算中心进行处理，并在很短的时间内给出处理结果，是对用户操作快速响应的方式之一。OLTP 是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。

典型案例：银行转账

```
开启事务  
从转出账号中扣钱  
往转入账号中加钱  
提交事务
```

- OLAP: **On-Line Analytic Processing (联机分析处理过程)**。OLAP 是**数据仓库**系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

典型案例：商品推荐

```
抽取（读取）历史订单  
分析历史订单，找到最受欢迎的商品  
展示结果
```

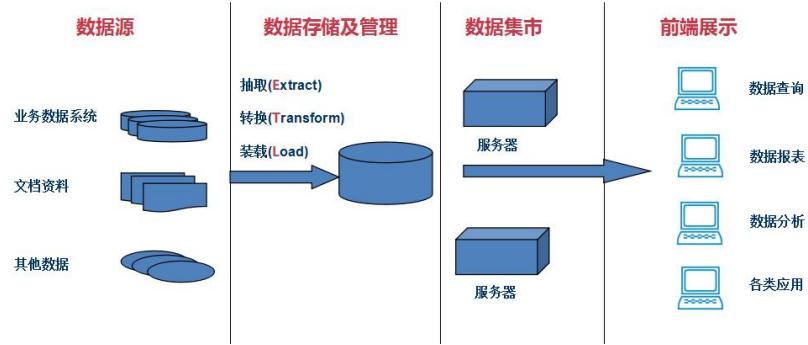
- OLTP 和 OLAP 的区别：

	OLTP	OLAP
用户	操作人员，低层管理人员	决策人员，高级管理人员
功能	日常操作处理	分析决策
DB 设计	面向应用	面向主题
数据	当前的，最新的细节的，二维的分立的	历史的，聚集的，多维的，集成的，统一的
存取	读写数十条记录	读上百万条记录
工作单位	简单的事务	复杂的查询
DB 大小	100MB-GB	100GB-TB

## (三) 数据仓库

数据仓库，英文名称为 **Data Warehouse**，可简写为 DW 或 DWH。数据仓库，是为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。它是单个数据存储，出于分析性报告和决策支持目的而创建。

### 数据仓库的结构和建立过程



## (四) Google 的基本思想

### Hadoop 的思想来源：Google

- Google 搜索引擎，Gmail，安卓，AppspotGoogle Maps，Google earth，Google 学术，Google 翻译，Google+，下一步 Google what？

A screenshot of a Google search results page. The search query "Hadoop" is entered in the search bar. The results show:

- 所有结果**: A sidebar with links to Images, Maps, Videos, News, Shopping, Books, and More.
- Welcome to Apache™ Hadoop™!**: A link to the official Apache Hadoop website ([hadoop.apache.org/](http://hadoop.apache.org/)). Description: Official site of the Apache project to provide an open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
- Hadoop 快速入门**: A link to the quickstart guide ([hadoop.apache.org/common/docs/r0.19.2/.../quickstart.html](http://hadoop.apache.org/common/docs/r0.19.2/.../quickstart.html)). Description: This document is intended to help you quickly complete a single machine's Hadoop installation and usage. It covers Hadoop's distributed file system (HDFS) and MapReduce framework.
- Hadoop 百度百科**: A link to the Baidu Encyclopedia entry ([baike.baidu.com/view/908354.htm](http://baike.baidu.com/view/908354.htm)). Description: A distributed system infrastructure developed by the Apache Foundation. Users can use it to understand the details of distributed systems under certain circumstances. It uses the power of distributed computation and storage. Hadoop implements ...
- Hadoop - 分布式文件系统-开源中国**: A link to the Open China website ([www.oschina.net](http://www.oschina.net)). Description: Hadoop is not just a distributed storage system, but a system designed for distributed processing of large data sets.

- Google 的低成本之道

- 不使用超级计算机，不使用存储（淘宝的去 i，去 e，去 o 之路）
- 大量使用普通的 pc 服务器（去掉机箱，外设，硬盘），提供有冗余的集群服务
- 全世界多个数据中心，有些附带发电厂
- 运营商向 Google 倒付费



- Google 的三篇论文（Hadoop 的思想来源）

- GFS (Google File System: Google 的文件系统)

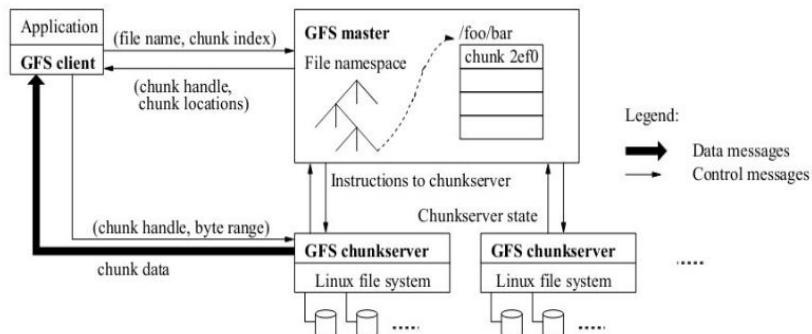
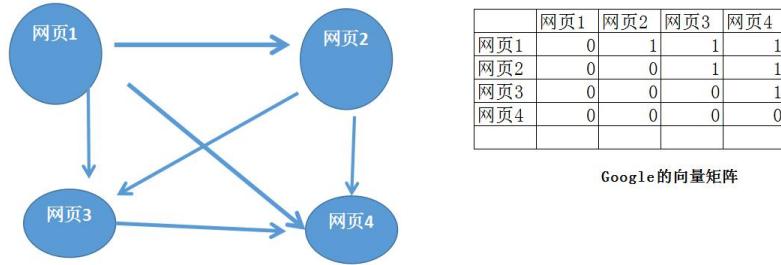


Figure 1: GFS Architecture

- 倒排索引



■ Page Rank (排名先后)



- BigTable (大表) : BigTable 是 Google 设计的分布式数据存储系统，用来处理海量的数据的一种**非关系型**的数据库。

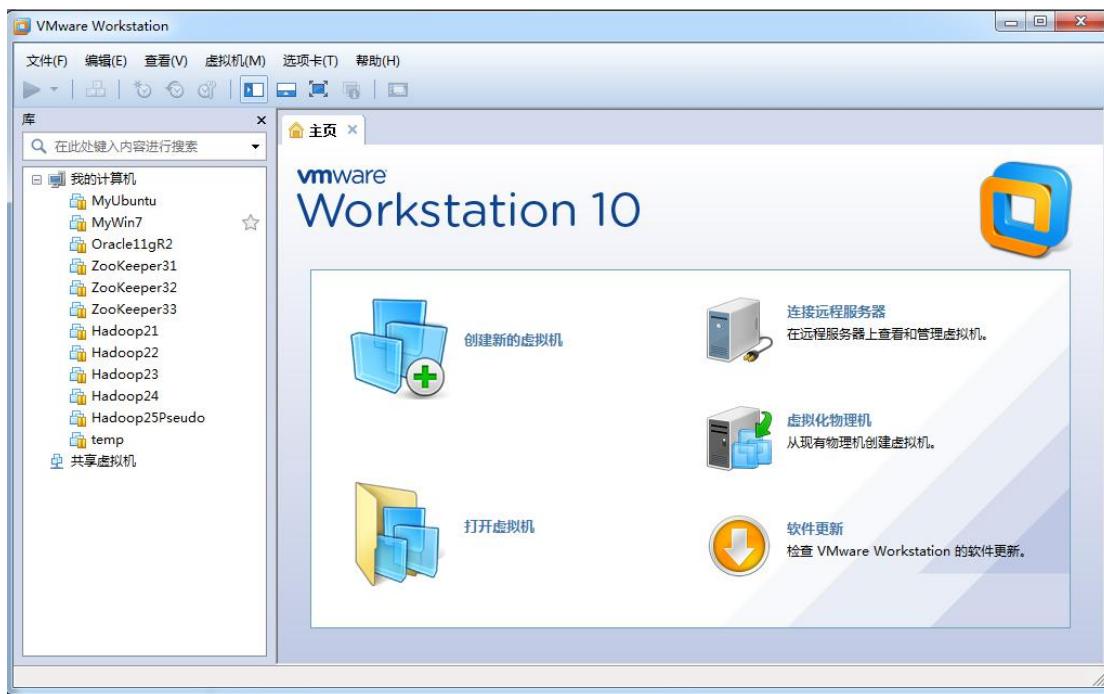
Eg: 什么是关系模型？（关系型数据库）

- ◆ 常见的 NoSQL 数据库 (key-value 值) :
  - (1) HBase: 基于 HDFS 的 NoSQL 数据库、面向列的
  - (2) Redis: 基于内存的 NoSQL 数据库、支持持久化: rdb 和 aof
  - (3) MongoDB: 面向文档的 NoSQL
  - (4) Cassandra: 面向列的 NoSQL 数据库
- Hadoop 的源起——Lucene, 从 lucene 到 nutch, 从 nutch 到 hadoop
  - 2003-2004 年, Google 公开了部分 GFS 和 Mapreduce 思想的细节, 以此为基础 Doug Cutting 等人用了 2 年业余时间实现了 DFS 和 Mapreduce 机制, 使 Nutch 性能飙升
  - Yahoo 招安 Doug Cutting 及其项目
  - Hadoop 于 2005 年秋天作为 Lucene 的子项目 Nutch 的一部分正式引入 Apache 基金会。2006 年 3 月份, Map-Reduce 和 Nutch Distributed File System (NDFS) 分别被纳入称为 Hadoop 的项目中
  - 名字来源于 Doug Cutting 儿子的玩具大象



## 二、实验环境

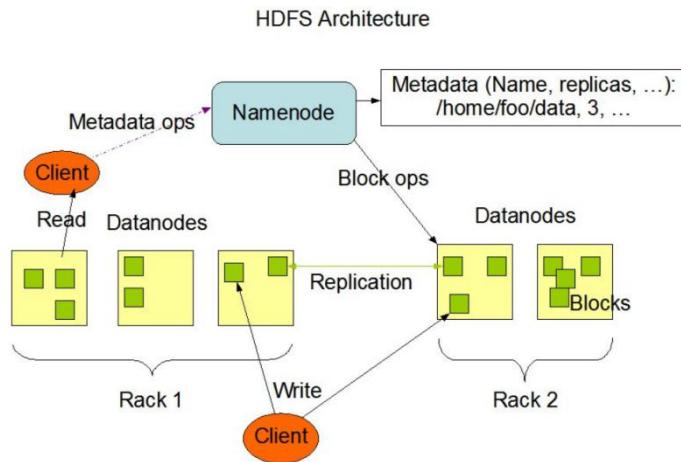
介质	说明
VMware	虚拟机管理器
MySQL	MySQL 数据库的安装介质
WinSCP-5.9.2-Portable.zip	FTP 客户端工具
Putty	登录 Linux 的命令行客户端
Enterprise-R5-U2-Server-i386-dvd.iso	Oracle Linux 安装介质
jdk-7u75-linux-i586.tar.gz	JDK 安装包
hadoop-2.4.1.tar.gz	Hadoop 安装包
hbase-0.96.2-hadoop2-bin.tar.gz	HBase NoSQL 安装包
hue-3.7.0-cdh5.4.2.tar.gz	HUE（基于 Web 的管理工具）安装包
apache-hive-0.13.0-bin.tar.gz	Hive（基于 Hadoop 的数据仓库）安装包
apache-hive-0.13.0-src.tar.gz	Hive（基于 Hadoop 的数据仓库）源码
apache-flume-1.6.0-bin.tar.gz	Flume（日志采集工具）安装包
sqoop-1.4.5-bin_hadoop-0.23.tar.gz	Sqoop（HDFS 与 RDBMS 的数据交换）安装包
pig-0.14.0.tar.gz	Pig（基于 Hadoop 的数据分析引擎）安装包
zookeeper-3.4.6.tar.gz	ZooKeeper（实现 Hadoop 的 HA）安装包



Eg: Hadoop 的 HDFS 和 MapReduce 示例 Demo。

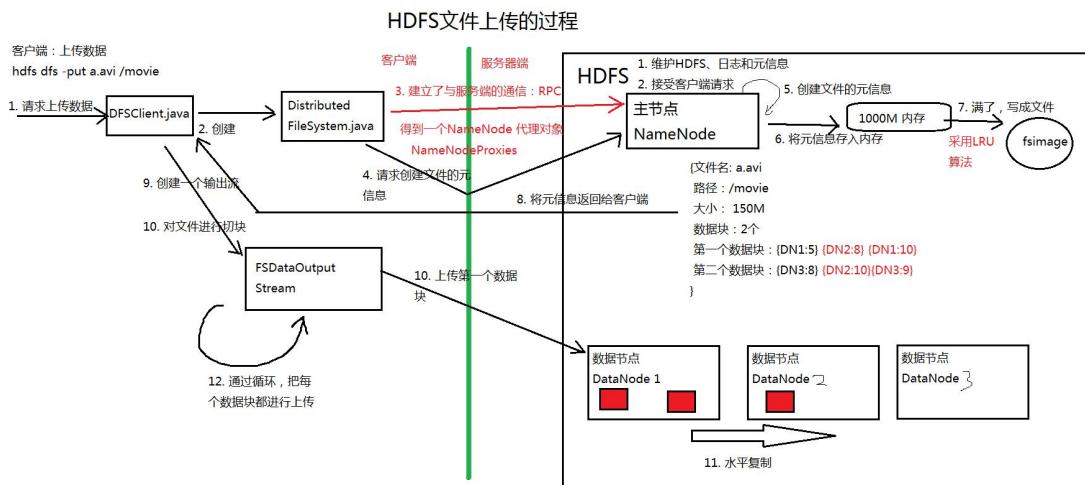
## 三、Apache Hadoop 的体系结构（重点）

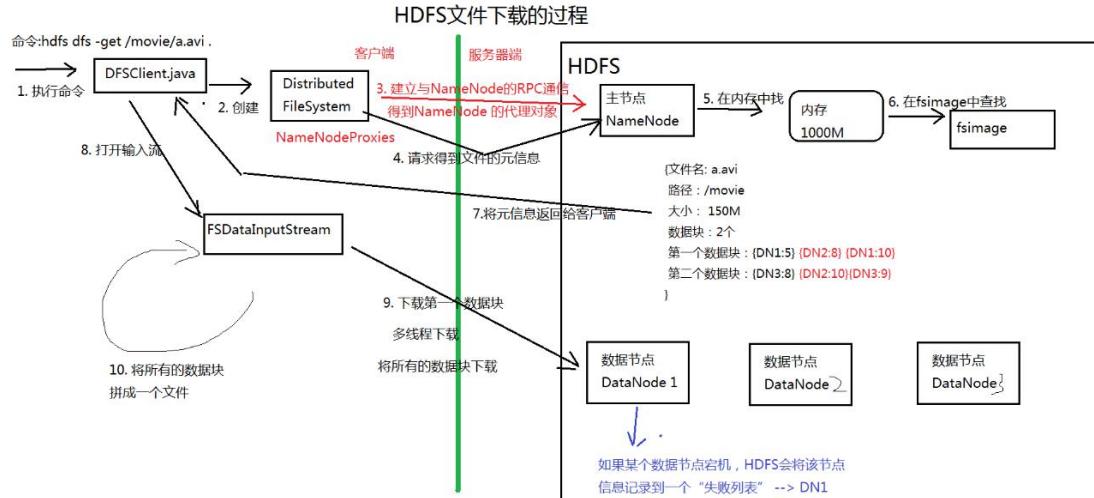
### （一）分布式存储：HDFS



#### ● NameNode（名称节点）

1. 维护 HDFS 文件系统，是 HDFS 的主节点。
2. 接受客户端的请求：上传文件、下载文件、创建目录等等





### 3. 记录客户端操作的日志（edits 文件），保存了 HDFS 最新的状态

- ① Edits 文件保存了自最后一次检查点之后所有针对 HDFS 文件系统的操作，比如：增加文件、重命名文件、删除目录等等
- ② 保存目录：\$HADOOP\_HOME/tmp/dfs/name/current

```
[root@hadoop25pesudo current]# ll -lt
total 3112
-rw-r--r--. 1 root root 1048576 May 29 21:41 edits_inprogress_0000000000000000122
-rw-r--r--. 1 root root      62 May 29 21:41 fsimage_0000000000000000121.md5
-rw-r--r--. 1 root root     3231 May 29 21:41 fsimage_0000000000000000121
-rw-r--r--. 1 root root       4 May 29 21:41 seen_txid
-rw-r--r--. 1 root root      42 May 29 21:41 edits_0000000000000000120-0000000000000000121
-rw-r--r--. 1 root root 1048576 May 29 21:33 edits_0000000000000000080-0000000000000000119
-rw-r--r--. 1 root root      62 May 29 21:31 fsimage_0000000000000000079.md5
-rw-r--r--. 1 root root     2853 May 29 21:31 fsimage_0000000000000000079
-rw-r--r--. 1 root root     7579 May 29 21:31 edits_000000000000000004-0000000000000000079
-rw-r--r--. 1 root root 1048576 May 29 21:15 edits_000000000000000003-0000000000000000003
-rw-r--r--. 1 root root      42 May 29 21:15 edits_000000000000000001-0000000000000000002
-rw-r--r--. 1 root root      205 May 29 21:14 VERSION
```

- ③ 可以使用 hdfs oev -i 命令将日志（二进制）输出为 XML 文件

```
hdfs oev -i edits_inprogress_0000000000000000122 -o ~/temp/log.xml
```

输出结果为：

```
<?xml version="1.0" encoding="UTF-8"?>
<EDITS>
  <EDITS_VERSION>56</EDITS_VERSION>
  <RECORD>
    <OPCODE>OP_START_LOG_SEGMENT</OPCODE>
    <DATA>
      <TXID>122</TXID>
    </DATA>
  </RECORD>
  <RECORD>          操作：创建目录
    <OPCODE>OP_MKDIR</OPCODE>
    <DATA>
      <TXID>123</TXID>
      <LENGTH>0</LENGTH>
      <INODEID>16429</INODEID>
      <PATH>/data</PATH> 目录的名字
      <TIMESTAMP>1464529308840</TIMESTAMP>
      <PERMISSION_STATUS>
        <USERNAME>root</USERNAME>
        <GROUPNAME>supergroup</GROUPNAME>
        <MODE>493</MODE>
      </PERMISSION_STATUS>
    </DATA>
  </RECORD>
</EDITS>
```

4. 维护文件元信息，将内存中不常用（采用 LRU 算法）的文件元信息保存在硬盘上（`fsimage` 文件）
  - ① `fsimage` 是 HDFS 文件系统存于硬盘中的元数据检查点，里面记录了自最后一次检查点之前 HDFS 文件系统中所有目录和文件的序列化信息
  - ② 保存目录：`$HADOOP_HOME/tmp/dfs/name/current`
  - ③ 可以使用 `hdfs oiv -i` 命令将日志（二进制）输出为文本（文本和 XML）

```
hdfs oiv -i fsimage_00000000000000000000121 -o ~/temp/fsimage.txt
```

```
hdfs oiv -i fsimage_00000000000000000000121 -o ~/temp/fsimage.xml -p XML
```

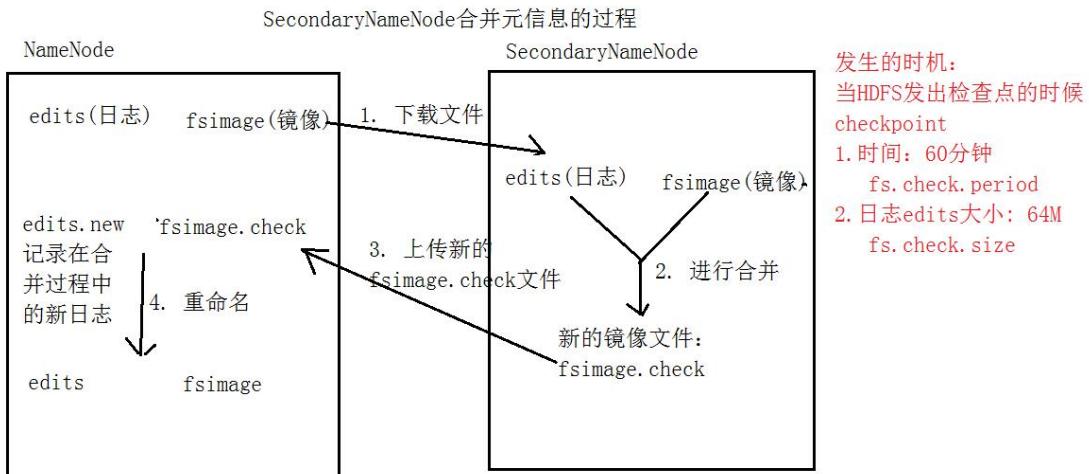
### ● DataNode (数据节点)

1. 以数据块为单位，保存数据
  - ① Hadoop 1.0 的数据块大小： 64M
  - ② Hadoop 2.0 的数据块大小： 128M
2. 在全分布模式下，至少两个 DataNode 节点
3. 数据保存的目录：由 `hadoop.tmp.dir` 参数指定  
例如：

```
$HADOOP_HOME/tmp/dfs/data/current/BP-437780861-192.168.137.25-1464527644399/current
```

### ● Secondary NameNode (第二名称节点)

1. 主要作用是进行日志合并
2. 日志合并的过程：



### ● HDFS 存在的问题

- ① NameNode 单点故障，难以应用到在线场景

解决方案：Hadoop 1.0 中，没有解决方案。

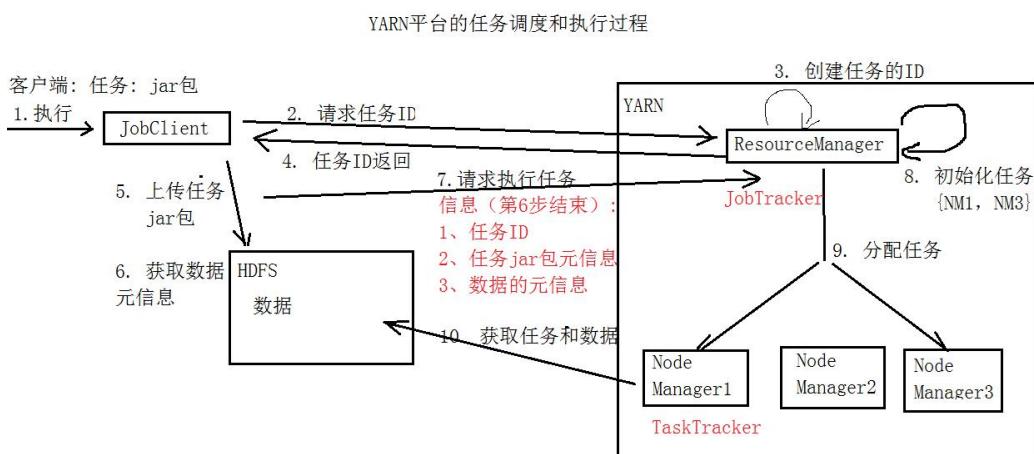
Hadoop 2.0 中，使用 ZooKeeper 实现 NameNode 的 HA 功能。

- ② NameNode 压力过大，且内存受限，影响系统扩展性

解决方案：Hadoop 1.0 中，没有解决方案。

Hadoop 2.0 中，使用 NameNode 的联盟实现其水平扩展。

## (二) YARN: 分布式计算(MapReduce)

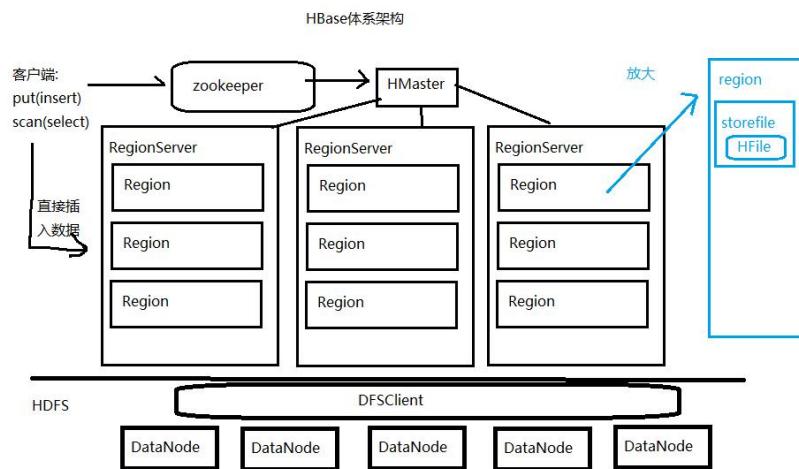


### ● ResourceManager (资源管理器)

- ① 接受客户端的请求: 执行任务  
② 分配资源  
③ 分配任务

- **NodeManager(节点管理器：运行任务 MapReduce)**  
① 从 DataNode 上获取数据，执行任务

### (三) HBase 的体系结构



## 四、Hadoop 2.X 的安装与配置

### (一) Hadoop 安装部署的预备条件

- 安装 Linux
- 安装 JDK

## (二) Hadoop 的目录结构



## (三) Hadoop 安装部署的三种模式

- 本地模式
- 伪分布模式
- 全分布模式

## 本地模式的配置

参数文件	配置参数	参考值
hadoop-env.sh	JAVA_HOME	/root/training/jdk1.8.0_144

## 伪分布模式的配置

参数文件	配置参数	参考值
hadoop-env.sh	JAVA_HOME	/root/training/jdk1.8.0_144
hdfs-site.xml	dfs.replication	1
	dfs.permissions	false
core-site.xml	fs.defaultFS	hdfs://<hostname>:9000
	hadoop.tmp.dir	/root/training/hadoop-2.7.3/tmp
mapred-site.xml	mapreduce.framework.name	yarn
yarn-site.xml	yarn.resourcemanager.hostname	<hostname>
	yarn.nodemanager.aux-services	mapreduce_shuffle

## 全分布模式的配置

参数文件	配置参数	参考值
hadoop-env.sh	JAVA_HOME	/root/training/jdk1.8.0_144
hdfs-site.xml	dfs.replication	2
	dfs.permissions	false
core-site.xml	fs.defaultFS	hdfs://<hostname>:9000
	hadoop.tmp.dir	/root/training/hadoop-2.7.3/tmp
mapred-site.xml	mapreduce.framework.name	yarn
yarn-site.xml	yarn.resourcemanager.hostname	<hostname>
	yarn.nodemanager.aux-services	mapreduce_shuffle
slaves	DataNode 的地址	从节点 1 从节点 2

### 如果出现以下警告信息：

Java HotSpot(TM) Client VM warning: You have loaded library /root/training/hadoop-2.7.3/lib/native/libhadoop.so.1.0.0 which might have disabled stack guard. The VM will try to fix the stack guard now.

It's highly recommended that you fix the library with 'execstack -c <libfile>', or link it with '-z noexecstack'.

只需要在以下两个文件中增加下面的环境变量，即可：

- hadoop-env.sh 脚本中

```
export JAVA_HOME=/root/training/jdk1.8.0_144
export HADOOP_HOME=/root/training/hadoop-2.7.3
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

- yarn-env.sh 脚本中

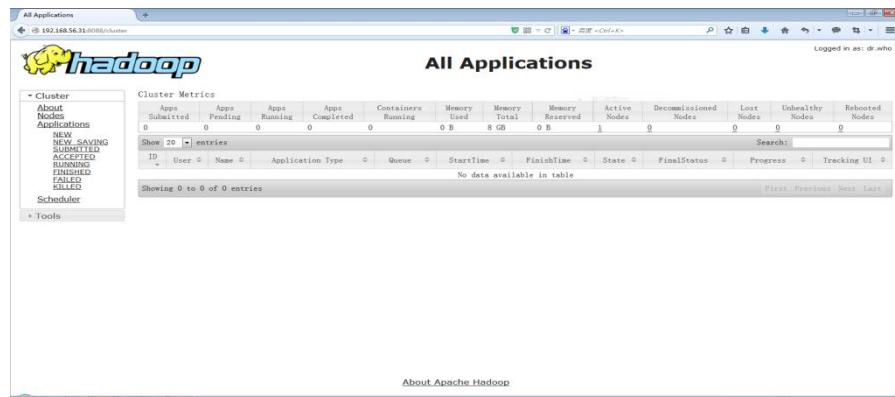
```
export JAVA_HOME=/root/training/jdk1.8.0_144
export HADOOP_HOME=/root/training/hadoop-2.7.3
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

## (四) 验证 Hadoop 环境

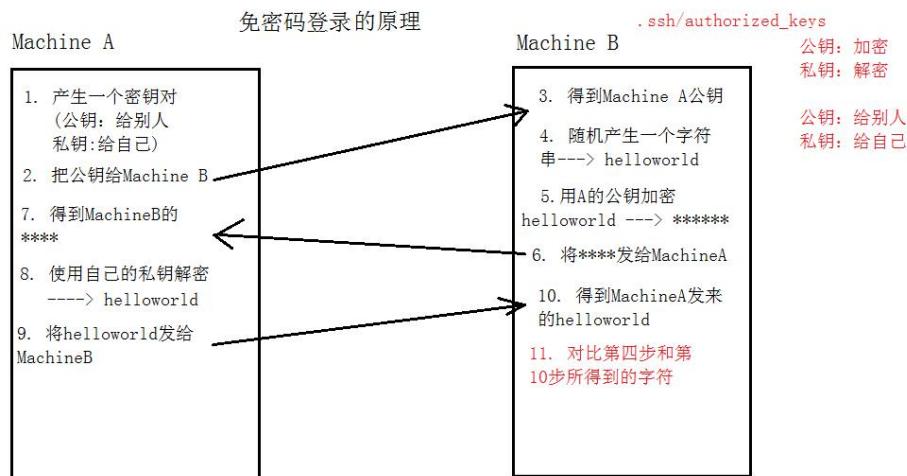
- HDFS Console: <http://192.168.157.11:50070>



- Yarn Console: <http://192.168.157.11:8088>



## (五) 配置 SSH 免密码登录

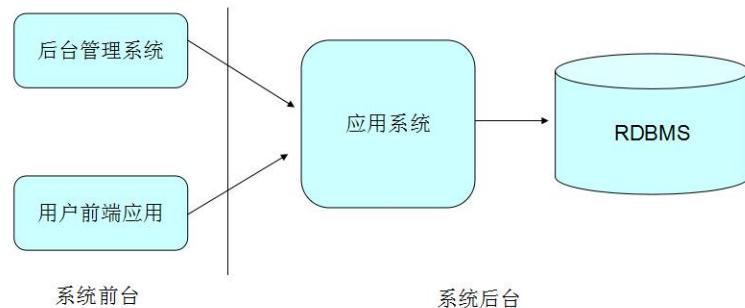


Demo: 配置 Linux SSH 的免密码登录。

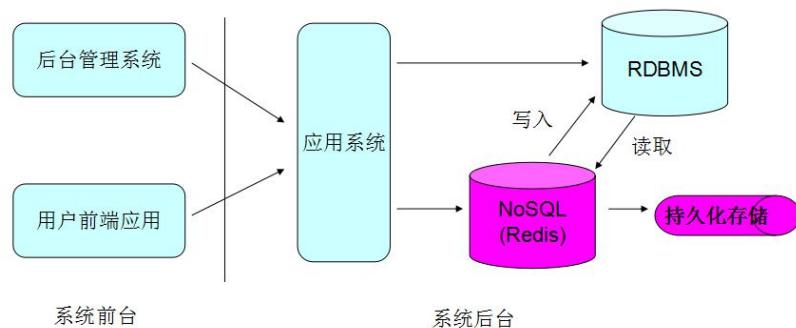
## 五、Hadoop 应用案例分析

### (一) 互联网应用的架构

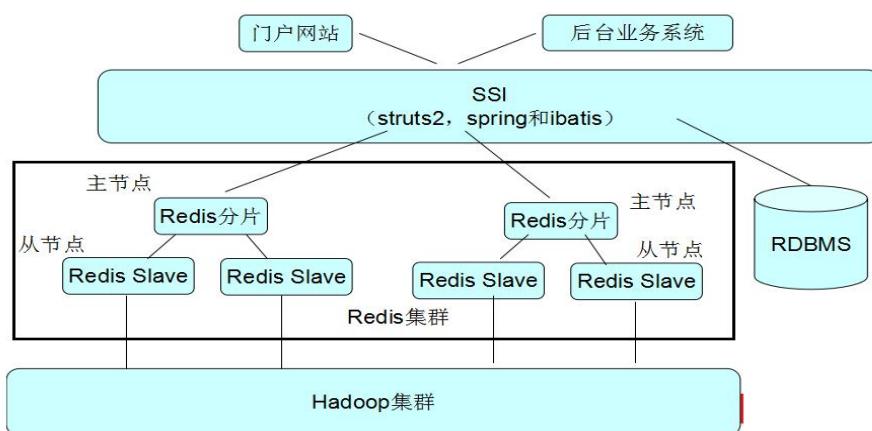
① 传统的架构



② 改良后的架构



③ 完整的架构图



## (二) 日志分析

### ① 需求说明

- 对某技术论坛的 apache server 日志进行分析，计算论坛关键指标，供运营者决策

### ② 论坛日志数据有两部分

- 历史数据约 56GB，统计到 2012-05-29
- 自 2013-05-30 起，每天生成一个数据文件，约 150MB

```

27.19.74.143 - - [30/May/2013:17:38:20 +0800] "GET /static/image/common/faq.gif HTTP/1.1" 200 1127
110.52.250.126 - - [30/May/2013:17:38:20 +0800] "GET /data/cache/style_1_widthauto.css?y7a HTTP/1.1" 200 1292
27.19.74.143 - - [30/May/2013:17:38:20 +0800] "GET /static/image/common/hot_1.gif HTTP/1.1" 200 680
27.19.74.143 - - [30/May/2013:17:38:20 +0800] "GET /static/image/common/hot_2.gif HTTP/1.1" 200 682
27.19.74.143 - - [30/May/2013:17:38:20 +0800] "GET /static/image/filetype/common.gif HTTP/1.1" 200 90
110.52.250.126 - - [30/May/2013:17:38:20 +0800] "GET /source/plugin/wsh_wx/img/wsh_zk.css HTTP/1.1" 200 1482
110.52.250.126 - - [30/May/2013:17:38:20 +0800] "GET /data/cache/style_1_forum_index.css?y7a HTTP/1.1" 200 2331
110.52.250.126 - - [30/May/2013:17:38:20 +0800] "GET /source/plugin/wsh_wx/img/wx_jqr.gif HTTP/1.1" 200 1770
27.19.74.143 - - [30/May/2013:17:38:20 +0800] "GET /static/image/common/recommend_1.gif HTTP/1.1" 200 1030

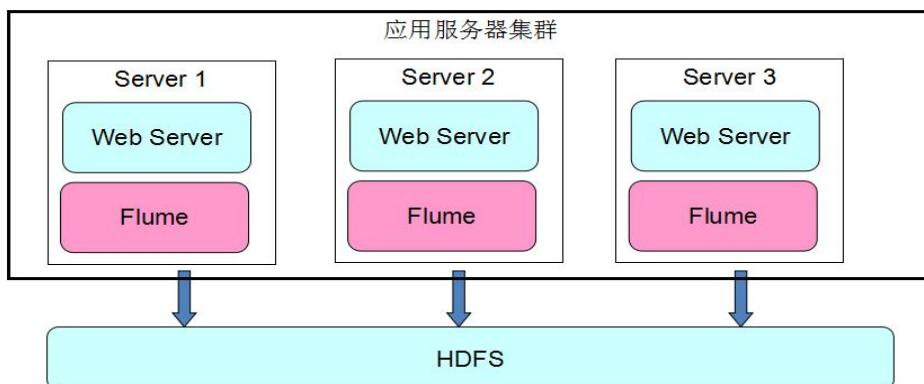
```

访问的IP地址	访问时间	访问的资源	访问 本次 状态	流量
---------	------	-------	-------------	----

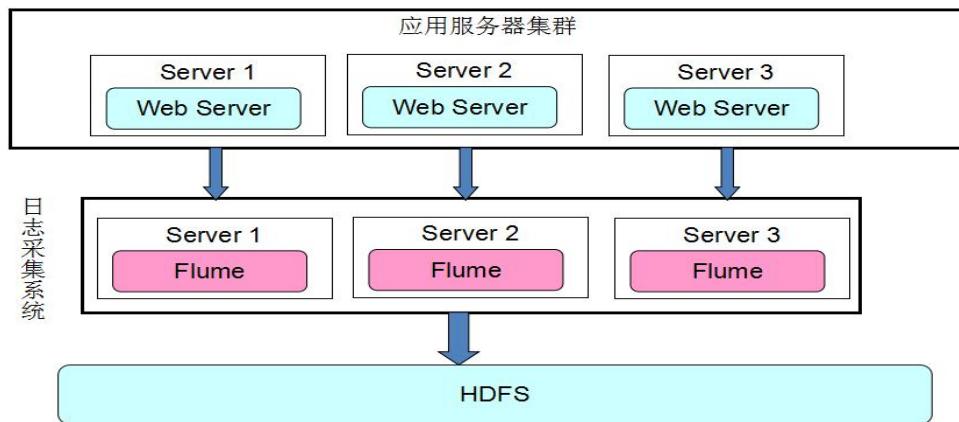
### ③ 关键指标

指标	说明	计算公式
浏览量PV	页面浏览量即为PV(Page View)，是指所有用户浏览页面的总和，一个独立用户每打开一个页面就被记录1次。	记录计数
访客数UV（包括新访客数、新访客比例）	访客数（UV）即唯一访客数，一天之内网站的独立访客数(以Cookie为依据)，一天内同一访客多次访问网站只计算1个访客。	对访问member.php?mod=register的不同ip，进行计数
IP数	一天之内，访问网站的不同独立IP个数加和。其中同一IP无论访问了几个页面，独立IP数均为1。	对不同IP进行计数
跳出率	只浏览了一个页面便离开了网站的访问次数占总的访问次数的百分比，即只浏览了一个页面的访问次数 / 全部的访问次数汇总。	统计一天内只出现一条记录的ip，称为跳出数
版块热度排行榜	版块的访问情况排行。	按访问次数、停留时间统计排序

### ④ 系统架构



⑤ 改良后的系统架构



⑥ HBase 表的结构

明细表	
行键	date:ip
明细列族	detail:xxx

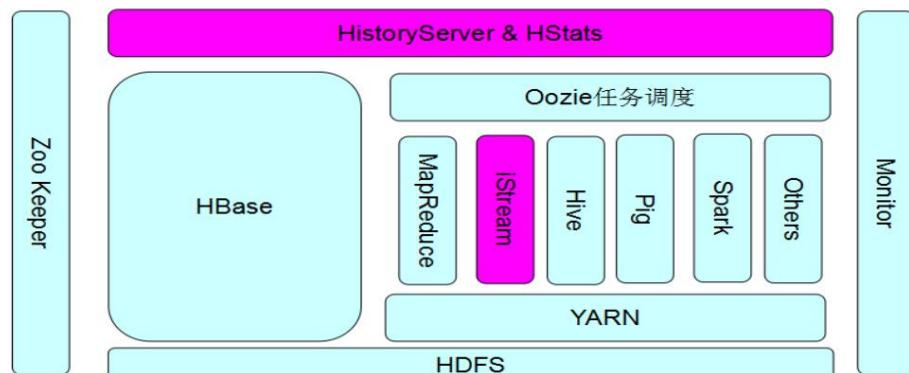
行键 (date:ip) details(明细列族)			
resource (访问资源)		status (状态)	流量
[30/May/2013:17:38:20 +0800] 27.19.74.143	"GET /static/image/common/recommend_1.gif HTTP/1.1"	200	1292

⑦ 日志分析的执行过程

- 周期性把日志数据导入到 hdfs 中
- 周期性把明细日志导入 hbase 存储
- 周期性使用 hive 进行数据的多维分析
- 周期性把 hive 分析结果导入到 mysql 中

### (三) Hadoop 在淘宝的应用

淘宝的搜索计算平台架构



## 六、HDFS

### (一) HDFS 的命令行操作

- HDFS 操作命令 (**HDFS 操作命令帮助信息: hdfs dfs**)

命令	说明	示例
-mkdir	在 HDFS 上创建目录	<ul style="list-style-type: none"> <li>在 HDFS 上创建目录/data hdfs dfs -mkdir /data</li> <li>在 HDFS 上级联创建目录/data/input hdfs dfs -mkdir -p /data/input</li> </ul>
-ls	列出 hdfs 文件系统根目录下的目录和文件	<ul style="list-style-type: none"> <li>查看 HDFS 根目录下的文件和目录 hdfs dfs -ls /</li> <li>查看 HDFS 的/data 目录下的文件和目录 hdfs dfs -ls /data</li> </ul>
-ls -R	列出 hdfs 文件系统所有的目录和文件	<ul style="list-style-type: none"> <li>查看 HDFS 根目录及其子目录下的文件和目录 hdfs dfs -ls -R /</li> </ul>
-put	上传文件或者从键盘输入字符到 HDFS	<ul style="list-style-type: none"> <li>将本地 Linux 的文件 data.txt 上传到 HDFS hdfs dfs -put data.txt /data/input</li> <li>从键盘输入字符保存到 HDFS 的文件 hdfs dfs -put - /aaa.txt</li> </ul>
-moveFromLocal	与 put 相类似，命令执行后源文件 local src 被删除，也可以从从键盘读取输入到 hdfs file 中	hdfs dfs -moveFromLocal data.txt /data/input
-copyFromLocal	与 put 相类似，也可以从从键盘读取输入到 hdfs file 中	hdfs dfs -copyFromLocal data.txt /data/input
-copyToLocal		
-get	将 HDFS 中的文件被复制到本地	hdfs dfs -get /data/inputdata.txt /root/
-rm	每次可以删除多个文件或目录	hdfs dfs -rm < hdfs file > ... 删除多个文件 hdfs dfs -rm -r < hdfs dir>... 删除多个目录
-getmerge	将 hdfs 指定目录下所有文件排序后合并到 local 指定的文件中，文件不存在时会自动创建，文件存在时会覆盖里面的内容	将 HDFS 上/data/input 目录下的所有文件，合并到本地的 a.txt 文件中 hdfs dfs -getmerge /data/input /root/a.txt
-cp	拷贝 HDFS 上的文件	
-mv	移动 HDFS 上的文件	
-count	统计 hdfs 对应路径下的目录个数，文件个数，文件总计大小 显示为目录个数，文件个数，文件总计大小，输入路径	
-du	显示 hdfs 对应路径下每个文件夹和文件的大小	hdfs dfs -du /
-text、-cat	将文本文件或某些格式的非文本文件通过文本格式输出	
balancer	如果管理员发现某些 DataNode 保存数据过多，某些 DataNode 保存数据相对较少，可以使用上述命令手动启动内部的均衡过程	

- HDFS 管理命令

**HDFS 管理命令帮助信息：hdfs dfsadmin**

命令	说明	示例
-report	显示文件系统的基本数据	hdfs dfsadmin -report
-safemode	HDFS 的安全模式命令 < enter   leave   get   wait >	hdfs dfsadmin -safemode enter leave get wait

## (二) HDFS 的 Java API

通过 HDFS 提供的 Java API，我们可以完成以下的功能：

- ① 在 HDFS 上创建目录
- ② 通过 FileSystem API 读取数据（下载文件）
- ③ 写入数据（上传文件）
- ④ 查看目录及文件信息
- ⑤ 查找某个文件在 HDFS 集群的位置
- ⑥ 删除数据
- ⑦ 获取 HDFS 集群上所有数据节点信息

- 在 HDFS 上创建目录

```
@Test
public void testMkdir() throws Exception{
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://192.168.137.25:9000");
    FileSystem fs = FileSystem.get(conf);
    //创建目录
    boolean flag = fs.mkdirs(new Path("/inputdata"));
    System.out.println(flag);
}
```

- 通过 FileSystem API 读取数据（下载文件）

```
@Test
public void testDownload() throws Exception {
    //构造一个输入流 <----HDFS
    FileSystem fs = FileSystem.get(new URI("hdfs://192.168.137.25:9000"),
    new Configuration());
    InputStream in = fs.open(new Path("/inputdata/a.war"));

    //构造一个输出流
    OutputStream out = new FileOutputStream("d:\\temp\\aaaa.war");
    IOUtils.copy(in, out);
}
```

- 写入数据（上传文件）

```
@Test
public void testUpload() throws Exception{
    //指定上传的文件(输入流)
    InputStream in = new FileInputStream("D:\\temp\\testds.war");

    //构造输出流 --> HDFS
    FileSystem fs = FileSystem.get(new URI("hdfs://192.168.137.25:9000"),
    new Configuration());
    OutputStream out = fs.create(new Path("/inputdata/a.war"));

    //工具类--》直接实现文件的上传和下载
    IOUtils.copy(in, out);
}
```

- 查看目录及文件信息

```
@Test
public void checkFileInfo() throws Exception {
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://192.168.137.25:9000");

    FileSystem fs = FileSystem.get(conf);
    FileStatus[] status = fs.listStatus(new Path("/hbase"));

    for (FileStatus f : status) {
        String dir = f.isDirectory() ? "目录" : "文件";
        String name = f.getPath().getName();
        String path = f.getPath().toString();
        System.out.println(dir + "----" + name + "  path:" + path);
        System.out.println(f.getAccessTime());
        System.out.println(f.getBlockSize());
        System.out.println(f.getGroup());
        System.out.println(f.getLength());
        System.out.println(f.getModificationTime());
        System.out.println(f.getOwner());
        System.out.println(f.getPermission());
        System.out.println(f.getReplication());
    }
}
```

- 查找某个文件在 HDFS 集群的位置

```
@Test
public void findFileBlockLocation() throws Exception{
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://192.168.137.25:9000";

    FileSystem fs = FileSystem.get(conf);

    FileStatus fStatus = fs.getFileStatus(new Path("/mydir1/temp1/mydata.txt"));

    BlockLocation[] blocks = fs.getFileBlockLocations(fStatus, 0, fStatus.getLen());
    for(BlockLocation block : blocks){
        System.out.println(Arrays.toString(block.getHosts())
                           + "\t"
                           + Arrays.toString(block.getNames()));
    }
}
```

- 删 除 数据

```
@Test
public void deleteFile() throws Exception{
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://192.168.137.25:9000");

    FileSystem fs = FileSystem.get(conf);
    //第二个参数表示是否递归
    boolean flag = fs.delete(new Path("/mydir1/temp1/mydata.txt"), false);
    System.out.println(flag ? "删除成功" : "删除失败");
}
```

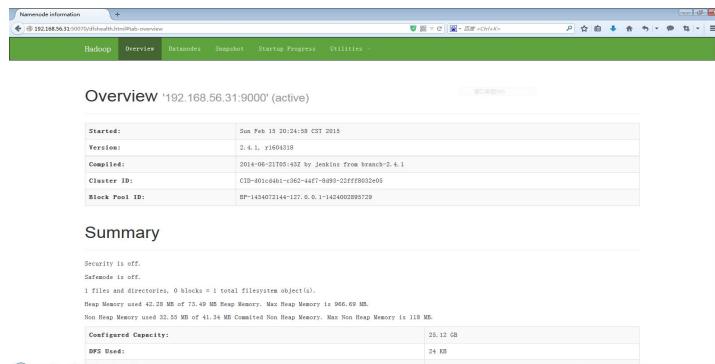
- 获取 HDFS 集群上所有数据节点信息

```
@Test
public void testDataNode() throws Exception {
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://192.168.137.25:9000");

    DistributedFileSystem fs = (DistributedFileSystem) FileSystem.get(conf);
    DatanodeInfo[] dataNodeStats = fs.getDataNodeStats();

    for (DatanodeInfo dataNode : dataNodeStats) {
        System.out.println(dataNode.getHostName() + "\t" + dataNode.getName());
    }
}
```

### (三) HDFS 的 Web Console



### (四) HDFS 的回收站

- 默认回收站是关闭的，可以通过在 `core-site.xml` 中添加 `fs.trash.interval` 来打开并配置时间阈值，例如：

```
<property>
  <name>fs.trash.interval</name>
  <value>1440</value>
</property>
```

- 删除文件时，其实是放入回收站/trash
- 回收站里的文件可以快速恢复
- 可以设置一个时间阈值，当回收站里文件的存放时间超过这个阈值，就被彻底删除，并且释放占用的数据块
- 查看回收站

```
hdfs dfs -lsr /user/root/.Trash/Current
```

- 从回收站中恢复

```
hdfs dfs -cp /user/root/.Trash/Current/input/data.txt /input
```

### (五) HDFS 的快照

- 一个 `snapshot`(快照)是一个全部文件系统、或者某个目录在某一时刻的镜像
- 快照应用在以下场景中：
  - 防止用户的错误操作
  - 备份
  - 试验/测试
  - 灾难恢复
- HDFS 的快照操作
  - 开启快照

```
hdfs dfsadmin -allowSnapshot /input
```

  - 创建快照

```
hdfs dfs -createSnapshot /input backup_input_01
```

■ 查看快照

```
hdfs lsSnapshottableDir
```

■ 对比快照

```
hdfs snapshotDiff /input backup_input_01 backup_input_02
```

■ 恢复快照

```
hdfs dfs -cp /input/.snapshot/backup_input_01/data.txt /input
```

## (六) HDFS 的用户权限管理

- 超级用户：

- 启动 namenode 服务的用户就是超级用户，该用户的组是 supergroup

```
[root@hadoop25pesudo ~]# hdfs dfs -ls /
Found 1 items
drwxr-xr-x 1 root supergroup 0 2016-05-29 21:31 /hbase
```

权限            用户名        组名

- Shell 命令变更

命令	说明
chmod [-R] mode file ...	只有文件的所有者或者超级用户才有权限改变文件模式。
chgrp [-R] group file ... chgrp [-R] group file ...	使用 chgrp 命令的用户必须属于特定的组且是文件的所有者，或者用户是超级用户
chown [-R] [owner][:[group]] file ...	文件的所有者的只能被超级用户更改。
ls(r) file ...	输出格式做了调整以显示所有者、组和模式。

- 文件系统 API 变更

- public FSDataOutputStream create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress) throws IOException;
- public boolean mkdirs(Path f, FsPermission permission) throws IOException;
- public void setPermission(Path p, FsPermission permission) throws IOException;
- public void setOwner(Path p, String username, String groupName) throws IOException;
- public FileStatus getFileStatus(Path f) throws IOException; 也会返回路径关联的所有者、组和模式属性。

## (七) HDFS 的配额管理

- 什么是配额?
  - 配额就是 HDFS 为每个目录分配的大小空间,新建立的目录是没有配额的,最大的配额是 Long.Max\_Value。配额为 1 可以强制目录保持为空。
- 配额的类型
  - 名称配额
    - ◆ 用于设置该目录中能够存放的最多文件(目录)个数。
  - 空间配额
    - ◆ 用于设置该目录中最大能够存放的文件大小。
- 配额的应用案例
  - 设置名称配额
    - ◆ 命令: `dfsadmin -setQuota <N> <directory>...<directory>`
    - `dfsadmin -clrQuota <directory>...<directory>`
    - ◆ 示例: 设置 目录的名称配额为 3
    - `hdfs dfsadmin -setQuota 3 /input`
    - 清除 目录的名称配额
    - `hdfs dfsadmin -clrQuota /input`
  - 设置空间配额
    - ◆ 命令: `dfsadmin -setSpaceQuota <quota> <dirname>...<dirname>`
    - `dfsadmin -clrSpaceQuota <dirname>...<dirname>`
    - ◆ 示例: 设置 目录的空间配额为 1M
    - `hdfs dfsadmin -setSpaceQuota 1048576 /input`
    - 清除 目录的空间配额
    - `hdfs dfsadmin -clrSpaceQuota /input`
- 注意: 如果 hdfs 文件系统中的文件个数或者大小超过了配额限制,会出现错误。

## (八) HDFS 的安全模式

- 什么时候安全模式?

安全模式是 hadoop 的一种保护机制,用于保证集群中的数据块的安全性。**如果 HDFS 处于安全模式,则表示 HDFS 是只读状态。**
- 当集群启动的时候,会首先进入安全模式。当系统处于安全模式时会检查数据块的完整性。假设我们设置的副本数(即参数 `dfs.replication`)是 5,那么在 `datanode` 上就应该有 5 个副本存在,假设只存在 3 个副本,那么比例就是  $3/5=0.6$ 。在配置文件 `hdfs-default.xml` 中定义了一个最小的副本的副本率 0.999,如图:

```
<property>
  <name>dfs.namenode.safemode.threshold-pct</name>
  <value>0.999f</value>
  <description>
    Specifies the percentage of blocks that should satisfy
    the minimal replication requirement defined by dfs.namenode.replication.min.
    Values less than or equal to 0 mean not to wait for any particular
    percentage of blocks before exiting safemode.
    Values greater than 1 will make safe mode permanent.
  </description>
</property>
```

我们的副本率 0.6 明显小于 0.99，因此系统会自动的复制副本到其他的 dataNode，使得副本率不小于 0.999。如果系统中有 8 个副本，超过我们设定的 5 个副本，那么系统也会删除多余的 3 个副本。

- 虽然不能进行修改文件的操作，但是可以浏览目录结构、查看文件内容的。
- 在命令行下是可以控制安全模式的进入、退出和查看的。
  - 命令 `hdfs dfsadmin -safemode get` 查看安全模式状态
  - 命令 `hdfs dfsadmin -safemode enter` 进入安全模式状态
  - 命令 `hdfs dfsadmin -safemode leave` 离开安全模式

## (九) HDFS 的底层原理

- HDFS 的底层通信原理采用的是：RPC 和动态代理对象 Proxy

- RPC

- 什么是 RPC？

Remote Procedure Call，远程过程调用。也就是说，调用过程代码并不是在调用者本地运行，而是要实现调用者与被调用者二地之间的连接与通信。

RPC 的基本通信模型是基于 Client/Server 进程间相互通信模型的一种同步通信形式；它对 Client 提供了远程服务的过程抽象，其底层消息传递操作对 Client 是透明的。

在 RPC 中，Client 即是请求服务的调用者(Caller)，而 Server 则是执行 Client 的请求而被调用的程序 (Callee)。

- RPC 示例

- 服务器端

```

package demo.hadoop.rpc.server;

import org.apache.hadoop.ipc.VersionedProtocol;

//包含了客户端需要调用的业务方法
public interface MyInterface extends VersionedProtocol {
    //指明本接口的ID
    public static long versionID = 1;
    //具体的业务方法
    public String sayHello(String name);
}

package demo.hadoop.rpc.server;

import java.io.IOException;
import org.apache.hadoop.ipc.ProtocolSignature;

public class MyInterfaceImpl implements MyInterface {
    @Override
    public ProtocolSignature getProtocolSignature(String arg0, long arg1,
        int arg2) throws IOException {
        return new ProtocolSignature(MyInterface.versionID, null);
    }

    @Override
    public long getProtocolVersion(String arg0, long arg1) throws IOException {
        return MyInterface.versionID;
    }

    @Override
    public String sayHello(String name) {
        System.out.println("***** 服务器端的代码调用 *****");
        // 业务方法
        return "Hello " + name;
    }
}

```

```

package demo.hadoop.rpc.server;
import java.io.IOException;
import org.apache.hadoop.HadoopIllegalArgumentException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
import org.apache.hadoop.ipc.RPC.Server;

public class RPCServer {
    //RPC的server
    private Server server = null;
    public RPCServer() throws Exception{
        //定义一个builder
        RPC.Builder builder = new RPC.Builder(new Configuration());
        //定义server的参数
        builder.setBindAddress("localhost");
        builder.setPort(1107);
        //将我们的接口注册到server上
        builder.setProtocol(MyInterface.class);
        builder.setInstance(new MyInterfaceImpl());
        //创建Server
        server = builder.build();
        //启动Server
        server.start();
    }
    public static void main(String[] args) throws Exception {
        new RPCServer();
    }
}

```

## ● 客户端

```

package demo.hadoop.rpc.client;
import java.io.IOException;
import java.net.InetSocketAddress;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
import demo.hadoop.rpc.server.MyInterface;
public class RPClient {
    public static void main(String[] args) throws Exception {
        System.out.println("***** 客户端的程序 *****");
        //创建一个Server的对象（句柄）----> 调用MyInterface中的业务方法
        //proxy不是服务器端MyInterface的真正对象，而是他的一个代理对象
        MyInterface proxy = RPC.getProxy(MyInterface.class,
                                         MyInterface.versionID,
                                         new InetSocketAddress("localhost", 1107),
                                         new Configuration());
        //通过代理对象来调用Server的业务程序
        String str = proxy.sayHello("Tom");
        System.out.println(str);
    }
}

```

## ● Java 动态代理对象

- 为其他对象提供一种代理以控制对这个对象的访问。
- 核心是使用 JDK 的 Proxy 类

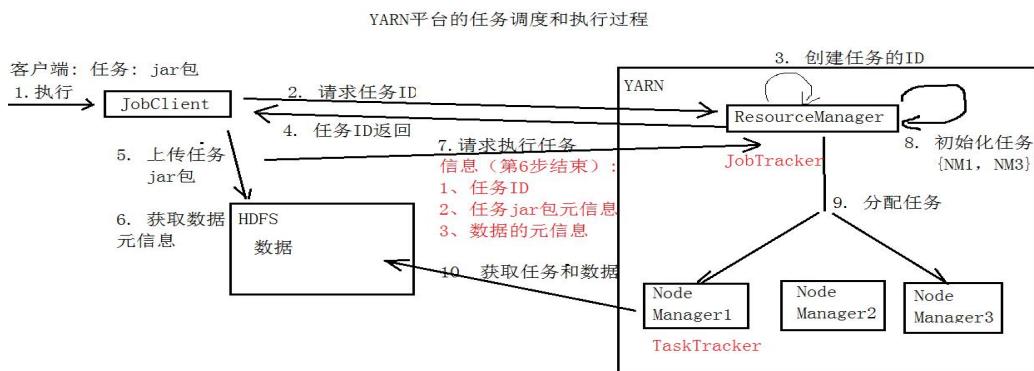
```

public class TestMain {
    public static void main(String[] args) {
        //创建一个真实的对象
        final Business b = new BusinessImpl();
        //为真实对象创建一个代理对象
        Business proxy = (Business) Proxy.newProxyInstance(TestMain.class.getClassLoader(),
                b.getClass().getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                        // 增强（重写）真实的对象功能
                        if(method.getName().equals("sayHello")){
                            //重写
                            System.out.println("***** 在代理对象中重写*****");
                            System.out.println("-----");
                            return null;
                        }else{
                            //调用的是其他方法
                            return method.invoke(b, args);
                        }
                    }
                });
        //调用代理对象
        System.out.println(proxy.sayHello("Tom"));
    }
}

```

## 七、MapReduce

### (一) MapReduce 在 Yarn 平台上 运行过程



### (二) 第一个 MapReduce 程序：WordCount

#### ● Mapper 的实现

```

//public class WordCountMapper extends Mapper<int, String, String, int> {
public class WordCountMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // 分词
        //key1      value1   key2      value2
        // 1          I love Beijing
        String str = value.toString();
        String[] words = str.split(" ");
        for(String word:words){
            //输出key2      value2
            // I           1
            // love        1
            // Beijing     1
            context.write(new Text(word), new LongWritable(1));
        }
    }
}
    
```

#### ● Reducer 的实现

```

//public class WordCountReducer extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
public class WordCountReducer extends Reducer<Text, LongWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text word, Iterable<LongWritable> value, Context context)  throws
        //key3      value3   key4      value4
        // I          {1,1,1}
        //love       {1,1,1}
        long sum = 0;
        for(LongWritable v:value){
            sum += v.get();
        }
        //输出  key4      value4
        context.write(new Text(word), new LongWritable(sum));
    }
}
    
```

## ● 主程序的实现

```
//Mapreducer的主程序
public static void main(String[] args) throws Exception {
    Job job = new Job(new Configuration());
    //指明程序的入口
    job.setJarByClass(WordCountMain.class);
    //指明任务中的mapper
    job.setMapperClass(WordCountMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);
    //设置Combiner
    job.setCombinerClass(WordCountReducer.class);
    //指明任务中的reducer
    job.setReducerClass(WordCountReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);
    //指明任务的输入路径和输出路径 ---> HDFS的路径
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    //启动任务
    job.waitForCompletion(true);
}
```

## (三) WordCount 的数据流过程



## (四) 使用 MapReduce 处理数据



emp.csv

- ① 排序：注意排序按照 **Key2** 排序

需要实现 **WritableComparable** 接口



SortMapper.java



Employee.java



SortMain.java

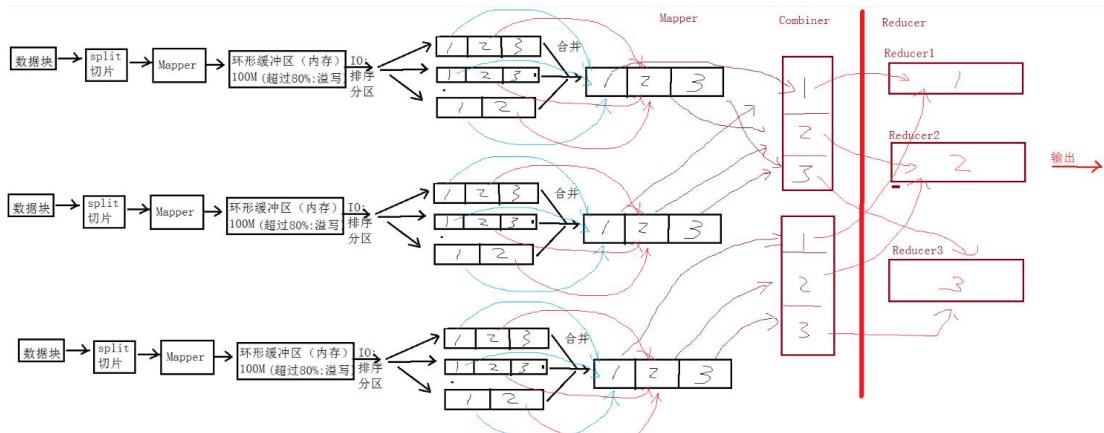
## ② 分区

```
public class EmployeePartition extends Partitioner<LongWritable, Employee> {
    @Override
    public int getPartition(LongWritable key2, Employee e, int numPartition) {
        // 分区的规则
        if(e.getDeptno() == 10){
            return 1*numPartition;
        }else if(e.getDeptno() == 20){
            return 2*numPartition;
        }else{
            return 3*numPartition;
        }
    }
}
```



## ③ 合并: *Combiner* 是一个特殊的 *Reducer*。但不是所有的情况都可以使用 *Combiner*。

## (五) Shuffle 的过程



## (六) 使用 MRUnit 进行单元测试过程

- 使用的时候需要从官网 <http://mrunit.apache.org/> 下载 jar 包
- 基本原理是 JUnit 和 EasyMock，其核心的单元测试依赖于 JUnit，并且 MRUnit 实现了一套 Mock 对象来控制 MapReduce 框架的输入和输出；语法也比较简单。
- 注意：需要把 `mockito-all-1.8.5.jar` 从 Build Path 中去掉
- 以 WordCount 为例：
  - 测试 Mapper

```
@Test
public void testMapper() throws Exception{
    System.setProperty("hadoop.home.dir", "D:\\tools\\hadoop-2.4.1\\hadoop-2.4.1");

    WordCountMapper mapper = new WordCountMapper();

    MapDriver<LongWritable, Text, Text, LongWritable> mapDriver = MapDriver.newMapDriver(mapper);

    mapDriver.withInput(new LongWritable(1), new Text("I love Beijing"));
    mapDriver.withOutput(new Text("I"), new LongWritable(1))
        .withOutput(new Text("love"), new LongWritable(1))
        .withOutput(new Text("Beijing"), new LongWritable(1));
    mapDriver.runTest();
}
```

### ■ 测试 Reducer

```
@Test
public void testReducer() throws Exception{
    System.setProperty("hadoop.home.dir", "D:\\tools\\hadoop-2.4.1\\hadoop-2.4.1");
    WordCountReducer reducer = new WordCountReducer();

    ReduceDriver<Text, LongWritable, Text, LongWritable> reducerDriver = ReduceDriver.newReduceDriver(reducer);

    List<LongWritable> values = new ArrayList<LongWritable>();
    values.add(new LongWritable(1));
    values.add(new LongWritable(1));
    values.add(new LongWritable(1));

    reducerDriver.withInput(new Text("Beijing"), values);
    reducerDriver.withOutput(new Text("Beijing"), new LongWritable(3));
    reducerDriver.runTest();
}
```

### ■ 测试 MapReducer

```
@Test
public void testMapReduce() throws Exception{
    System.setProperty("hadoop.home.dir", "D:\\tools\\hadoop-2.4.1\\hadoop-2.4.1");

    WordCountMapper mapper = new WordCountMapper();
    WordCountReducer reducer = new WordCountReducer();

    MapReduceDriver<LongWritable, Text, Text, LongWritable, Text, LongWritable> driver =
        MapReduceDriver.newMapReduceDriver(mapper, reducer);

    driver.withInput(new LongWritable(1), new Text("I love Beijing"))
        .withInput(new LongWritable(4), new Text("I love China"));
    driver.withOutput(new Text("Beijing"), new LongWritable(1))
        .withOutput(new Text("China"), new LongWritable(1))
        .withOutput(new Text("I"), new LongWritable(2))
        .withOutput(new Text("love"), new LongWritable(2));

    driver.runTest();
}
```

## (七) MapReduce 作业任务的管理

- 通过 Web Console 监控作业的运行

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1464535024572_0001	root	word count	MAPREDUCE	default	Sun, 29 May 2016 15:45:11 GMT	Sun, 29 May 2016 15:45:31 GMT	FINISHED	SUCCEEDED		<a href="#">History</a>

- 通过 yarn application 命令来进行作业管理

- ① 列出帮助信息：

```
yarn application --help
```

- ② 查看运行 Mapreduce 程序：

```
yarn application -list
```

- ③ 查看应用状态：

```
yarn application -status <application_id>
```

- ④ 强制杀死应用：

```
yarn application -kill <application_id>
```

## (八) MapReduce 案例集锦

- 通过下面六个案例，强化 MapReduce 程序的开发和理解

- 数据去重
- 数据排序
- 平均成绩
- 单表关联

```
JONES      (FORD;SCOTT;)
BLAKE      (JAMES;TURNER;ALLEN;MARTIN;WARD;)
CLARK      (MILLER;)
SCOTT      (ADAMS;)
KING       (BLAKE;JONES;CLARK;)
FORD       (SMITH;)
```

### ■ 多表关联

ACCOUNTING	MILLER;KING;CLARK;
RESEARCH	ADAMS;SCOTT;SMITH;JONES;FORD;
SALES	TURNER;ALLEN;BLAKE;MARTIN;WARD;JAMES;
OPERATIONS	

### ■ 倒排索引

数据：

```
data01.txt  
| I love Beijing and love Shanghai  
  
data02.txt  
| I love China  
  
data03.txt  
| Beijing is the capital of China
```

结果：

Beijing	(data01.txt:1) (data03.txt:1)
China	(data03.txt:1) (data02.txt:1)
I	(data02.txt:1) (data01.txt:1)
Shanghai	(data01.txt:1)
and	(data01.txt:1)
capital	(data03.txt:1)
is	(data03.txt:1)
love	(data01.txt:2) (data02.txt:1)
of	(data03.txt:1)
the	(data03.txt:1)

## ● 数据去重

“数据去重”主要是为了掌握和利用 并行化思想来对数据进行 有意义的 筛选。  
统计大数据集上的数据种类个数统计大数据集上的数据种类个数、 从网站日志中计算  
访问地等这些看似庞杂的任务都会涉及数据去重。

### 1、问题描述

对数据文件中的数据进行去重。数据文件中的每行都是一个数据。

样例输入文件如下所示：

文件 1:

2012-3-1 a  
2012-3-2 b  
**2012-3-3 c**  
**2012-3-4 d**  
2012-3-5 a  
2012-3-6 b  
2012-3-7 c  
**2012-3-3 c**

文件 2:

2012-3-1 b  
2012-3-2 a  
2012-3-3 b  
**2012-3-4 d**  
2012-3-5 a  
2012-3-6 c  
2012-3-7 d  
**2012-3-3 c**

样例输出如下所示：

结果：  
2012-3-1 a  
2012-3-1 b  
2012-3-2 a  
2012-3-2 b  
2012-3-3 b  
2012-3-3 c  
2012-3-4 d  
2012-3-5 a  
2012-3-6 b  
2012-3-6 c  
2012-3-7 c  
2012-3-7 d

## 2、设计思路

数据去重的最终目标是让 原始数据中 出现次数 超过一次的 数据在 输出文件中 只出现一次。我们自然而然会想到将同一个数据的所有记录都交给一台 reduce 机器，无论这个数据出现多少次，只要在最终结果中输出一次就可以了。具体就是 reduce 的 输入应该以 数据作为 key，而对 value-list 则 没有要求。当 reduce 接收到一个<key, value-list>时就 直接将 key 复制到输出的 key 中，并将 value 设置成 空值。

在 MapReduce 流程中， map 的输出<key, value>经过 shuffle 过程聚集成<key, value-list>后会交给 reduce。所以从设计好的 reduce 输入可以反推出 map 的输出 key 应为数据, value 任意。继续反推, map 输出数据的 key 为数据，而在这个实例中每个数据代表输入文件中的一行内容，所以 map 阶段要完成的任务就是在采用 Hadoop 默认的作业输入方式之后，将 value 设置为 key，并直接输出（输出中的 value 任意）。map 中的结果经过 shuffle 过程之后交给 reduce。reduce 阶段不会管每个 key 有多少个 value，它直接将输入的 key 复制为输出的 key，并输出就可以了（输出中的 value 被设置成空了）。

## 3、程序代码

### ● 数据排序

“ 数据排序 ” 是许多实际任务执行时要完成的第一项工作，比如 学生成绩评比、数据建立索引数据建立索引等。这个实例和数据去重类似，都是先对原始数据进行 初步处理，为 进一步的数据操作打好基础。

#### 1、问题描述

对输入文件中数据进行排序。输入文件中的每行内容均为一个数字，即一个数据。要求在输出中每行有 两个间隔的数字，其中， 第一个代表原始数据在原始数据集中的位次，第二个代表原始数据。

输入文件：

文件 1：  
2  
32  
654  
32  
15  
756  
65223

文件 2：  
5956  
22  
650  
92

文件 3：  
26  
54  
6

输出文件：

输出结果：  
1 2  
2 6  
3 15  
4 22  
5 26  
6 32  
7 32  
8 54  
9 92  
10 650  
11 654  
12 756  
13 5956  
14 65223

## 2、设计思路

该示例仅仅要求对输入数据进行排序，熟悉 MapReduce 过程的读者会很快想到在 MapReduce 过程中就有排序，是否可以利用这个默认的排序，而不需要自己再实现具体的排序呢？答案是肯定的。

但是在使用之前首先需要了解它的默认排序规则。它是按照 key 值进行排序的，如果 key 为封装 int 的 IntWritable 类型，那么 MapReduce 按照数字大小对 key 排序，如果 key 为封装为 String 的 Text 类型，那么 MapReduce 按照字典顺序对字符串排序。

了解了这个细节，我们就知道应该使用封装 int 的 IntWritable 型数据结构了。也就是在 map 中将读入的数据转化成 IntWritable 型，然后作为 key 值输出（value 任意）。reduce 拿到<key, value-list>之后，将输入的 key 作为 value 输出，并根据 value-list 中元素的个数决定输出的次数。输出的 key（即代码中的 linenum）是一个全局变量，它统计当前 key 的位次。需要注意的是这个程序中没有配置 Combiner，也就是在 MapReduce 过程中不使用 Combiner。这主要是因为使用 map 和 reduce 就已经能够完成任务了。

## ● 平均成绩

“平均成绩”主要目的还是在重温经典“WordCount”例子，可以说是在基础上的微变化微变化版，该实例主要就是实现一个计算学生平均成绩的例子。

### 1. 问题描述

对输入文件中数据进行就算学生平均成绩。输入文件中的每行内容均为一个学生的姓名和他相应的成绩，如果有多门学科，则每门学科为一个文件。要求在输出中每行有两个间隔的数据，其中，第一个代表学生的姓名，第二个代表其平均成绩。

输入文件：

数学成绩:
Tom 88
Mary 99
Mike 66
Jone 77

语文成绩:
Tom 78
Mary 89
Mike 96
Jone 67

英语成绩:
Tom 80
Mary 82
Mike 84
Jone 86

输出文件：

平均成绩:
Tom 82
Mary 90
Mike 82
Jone 76

### 2. 设计思路

计算学生平均成绩是一个仿“WordCount”例子，用来重温一下开发 MapReduce 程序的流程。程序包括两部分的内容：Map 部分和 Reduce 部分，分别实现了 map 和 reduce 的功能。

Map 处理的是一个纯文本文件，文件中存放的数据时每一行表示一个学生的姓名和他相应一科成绩。Mapper 处理的数据是由 InputFormat 分解过的数据集，其中 InputFormat 的作用是将数据集切割成小数据集 InputSplit，每一个 InputSplit 将由一个 Mapper 负责处理。此外，InputFormat 中还提供了一个 RecordReader 的实现，并将一个 InputSplit 解析成对提供了 map 函数。InputFormat 的默认值是 TextInputFormat，它针对文本文件，按行将文本切割成 InputSplit，并用 LineRecordReader 将 InputSplit 解析成对，key 是行在文本中的位置，value 是文件中的一行。

Map 的结果会通过 partition 分发到 Reducer，Reducer 做完 Reduce 操作后，将通过以格式 OutputFormat 输出。

Mapper 最终处理的结果对

## ● 单表关联

前面的实例都是在数据上进行一些简单的处理,为进一步的操作打基础。“单表关联”这个实例要求从给出的数据中寻找所关心的数据,它是对原始数据所包含信息的挖掘。下面进入这个实例。

### 1. 问题描述

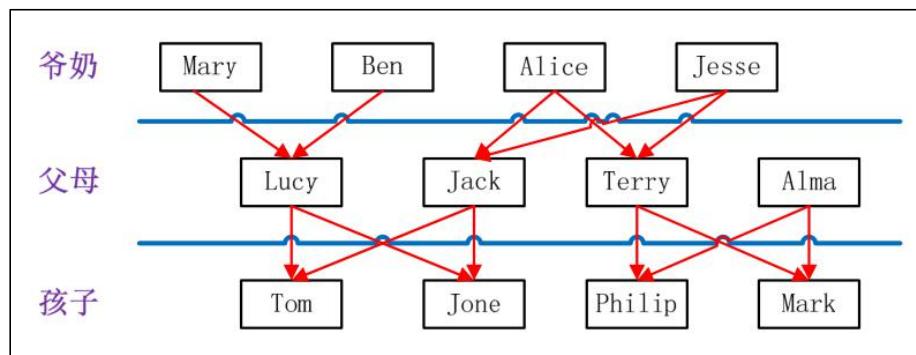
实例中给出 child-parent(孩子——父母)表,要求输出 grandchild-grandparent(孙子——爷奶)表。

样例输入如下所示:

样例输出如下所示:

child	parent	grandchild	grandparent
Tom	Lucy	Tom	Alice
Tom	Jack	Tom	Jesse
Jone	Lucy	Jone	Alice
Jone	Jack	Jone	Jesse
Lucy	Mary	Tom	Mary
Lucy	Ben	Tom	Ben
Jack	Alice	Jone	Mary
Jack	Jesse	Jone	Ben
Terry	Alice	Philip	Alice
Terry	Jesse	Philip	Jesse
Philip	Terry	Mark	Alice
Philip	Alma	Mark	Jesse
Mark	Terry		
Mark	Alma		

家族树状关系谱:



### 2. 设计思路

分析这个实例,显然需要进行单表连接,连接的是左表的 parent 列和右表的 child 列,且左表和右表是同一个表。

连接结果中除去连接的两列就是所需要的结果——“grandchild--grandparent”表。要用 MapReduce 解决这个实例,首先应该考虑如何实现表的自连接;其

次就是 连接列的 设置；最后是 结果的 整理。

考慮到 MapReduce 的 shuffle 过程会将相同的 key 会连接在一起， 所以可以将 map 结果的 key 设置成 待连接的 列， 然后列中相同的值就自然会连接在一起了。 再与最开始的分析联系起来：

要连接的是左表的 parent 列和右表的 child 列， 且左表和右表是同一个表， 所以在 map 阶段将 读入数据 分割成 child 和 parent 之后， 会将 parent 设置成 key， child 设置成 value 进行输出，并作为 左表；再将 同一对 child 和 parent 中的 child 设置成 key， parent 设置成 value 进行输出，作为 右表。为了 区分输出中的 左右表，需要在输出的 value 中 再加上左右表左右表 的 信息，比如在 value 的 String 最开始处加上 字符 1 表示 左表， 加上 字符 2 表示 右表。这样在 map 的结果中就形成了左表和右表，然后在 shuffle 过程中完成连接。reduce 接收到连接的结果，其中每个 key 的 value-list 就包含了“grandchild-grandparent”关系。取出每个 key 的 value-list 进行解析， 将 左表中的 child 放入一个 数组，右表中的 parent 放入一个 数组，然后对 两个数组求笛卡尔积就是最后的结果了。

## ● 多表关联

多表关联和单表关联类似，它也是通过对原始数据进行一定的处理， 从其中挖掘出关心的信息。

### 1. 问题描述

输入是两个文件，一个代表 工厂表，包含 工厂名列和 地址编号列；另一个代表 地址表包 地址名列和 地址编号列。要求从 输入数据中找出 工厂名和 地址名的 对应关系，输出“工厂名——地址名”表。

输入文件：

工厂信息：		地址信息：	
factoryname	addressed	addressID	addressname
Beijing Red Star	1	1	Beijing
Shenzhen Thunder	3	2	Guangzhou
Guangzhou Honda	2	3	Shenzhen
Beijing Rising	1	4	Xian
Guangzhou Development Bank	2		
Tencent	3		
Back of Beijing	1		

输出文件：

factoryname	addressname
Back of Beijing	Beijing
Beijing Red Star	Beijing
Beijing Rising	Beijing
Guangzhou Development Bank	Guangzhou
Guangzhou Honda	Guangzhou
Shenzhen Thunder	Shenzhen
Tencent	Shenzhen

多表关联和单表关联相似，都类似于数据库中的自然连接。相比单表关联，多表关联的左右表和连接列更加清楚。所以可以采用和单表关联的相同的方式，`map`识别出输入的行属于哪个表之后，对其进行分割，将连接的列值保存在`key`中，另一列和左右表标识保存在`value`中，然后输出。`reduce`拿到连接结果之后，解析`value`内容，根据标志将左右表内容分开存放，然后求笛卡尔积，最后直接输出。

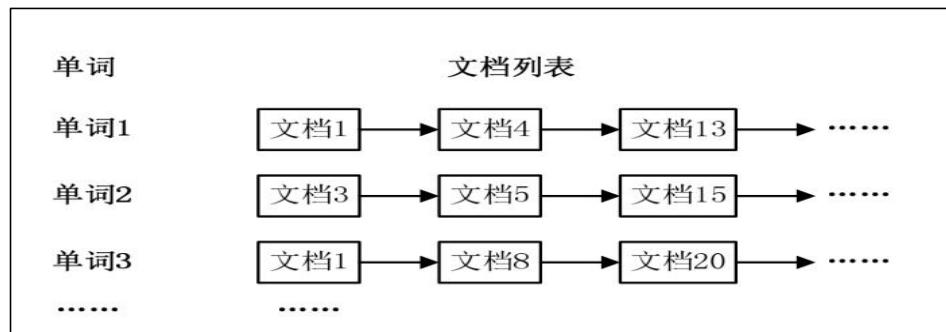
这个实例的具体分析参考单表关联实例。

## ● 倒排索引

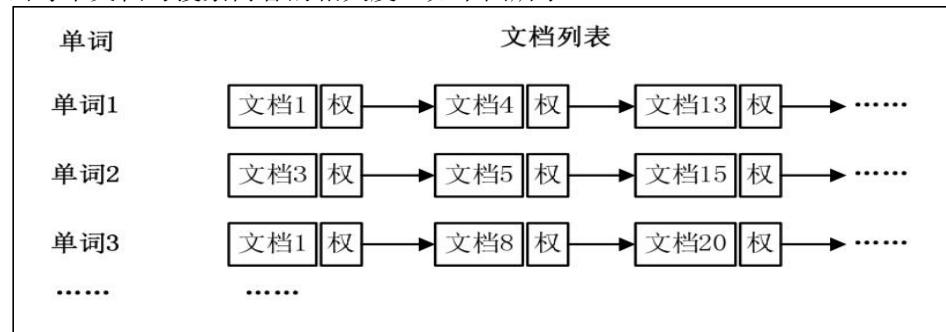
“倒排索引”是文档检索系统中最常用的 数据结构，被广泛地应用于全文搜索引擎。它主要是用来存储某个单词（或词组）在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行相反的操作，因而称为倒排索引（Inverted Index）。

### 1. 问题描述

通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档或者是标识文档的ID号，或者是指文档所在位置的URL，如下图所示。

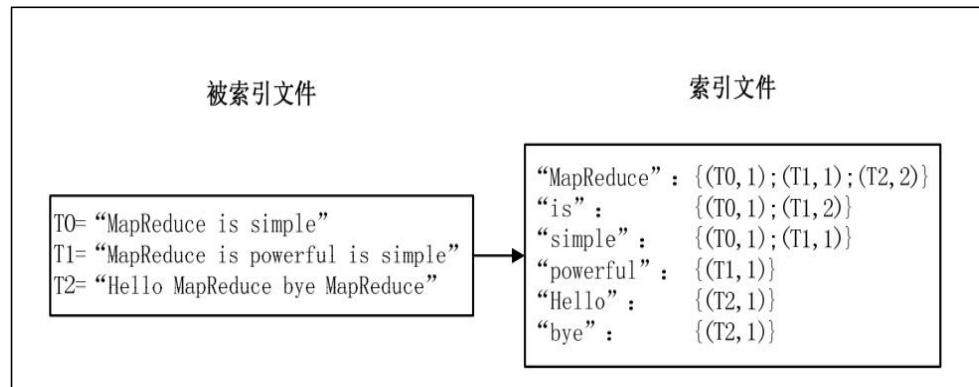


从上图可以看出，单词1出现在{文档1, 文档4, 文档13, .....}中，单词2出现在{文档3, 文档5, 文档15, .....}中，而单词3出现在{文档1, 文档8, 文档20, .....}中。在实际应用中，还需要给每个文档添加一个权值，用来指出每个文档与搜索内容的相关度，如下图所示：



最常用的是使用词频作为权重，即记录单词在文档中出现的次数。以英文为例，如下图所示，索引文件中的“MapReduce”一行表示：“MapReduce”这个单词在文本T0中出现过1次，T1中出现过1次，T2中出现过2次。当搜索条

件为“MapReduce”、“is”、“Simple”时，对应的集合为： $\{T_0, T_1, T_2\} \cap \{T_0, T_1\} \cap \{T_0, T_1\} = \{T_0, T_1\}$ ，即文档  $T_0$  和  $T_1$  包含了所要索引的单词，而且只有  $T_0$  是连续的。



更复杂的权重还可能要记录单词在多少个文档中出现过，以实现 TF-IDF (TermFrequency-Inverse Document Frequency) 算法，或者考虑单词在文档中的位置信息（单词是否出现在标题中，反映了单词在文档中的重要性）等。

输入文件如下：

文件 1:  
MapReduce is simple

文件 2:  
MapReduce is powerful is simple

文件 3:  
Hello MapReduce bye MapReduce

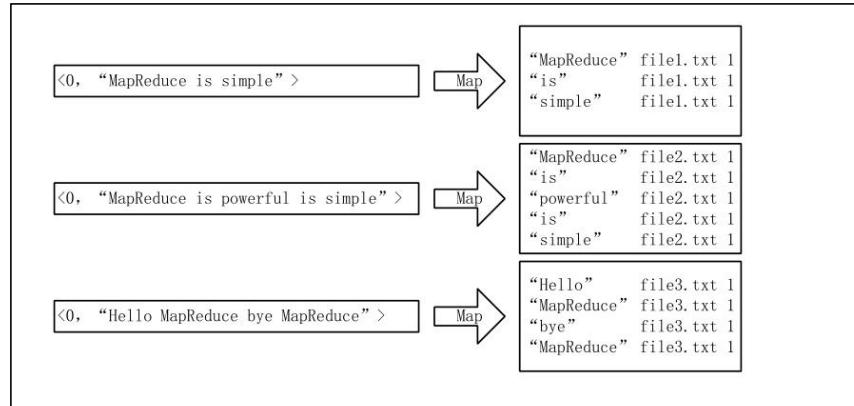
输出文件如下：

<b>MapReduce</b>	<b>file1.txt:1;file2.txt:1;file3.txt:2;</b>
<b>is</b>	<b>file1.txt:1;file2.txt:2;</b>
<b>simple</b>	<b>file1.txt:1;file2.txt:1;</b>
<b>powerful</b>	<b>file2.txt:1;</b>
<b>Hello</b>	<b>file3.txt:1;</b>
<b>bye</b>	<b>file3.txt:1;</b>

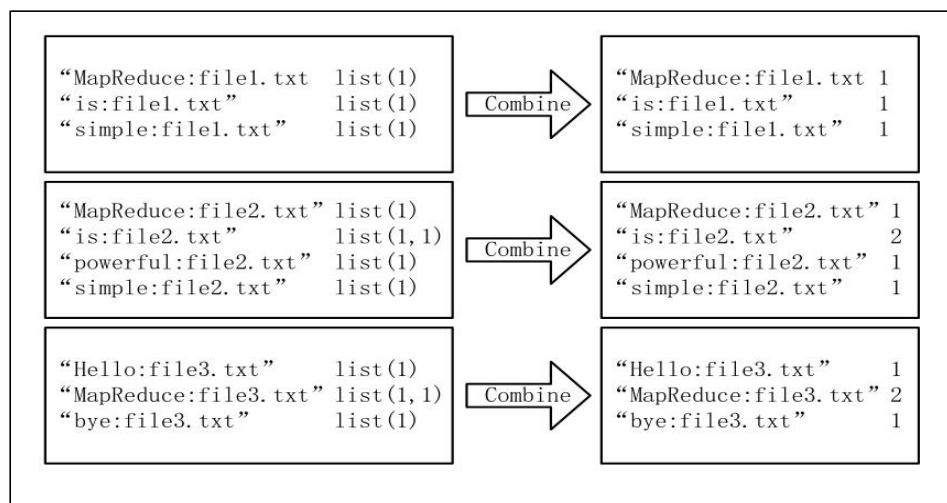
## 2. 设计思路

实现“倒排索引”只要关注的信息为：单词、文档 URL 及词频。但是在实现过程中，索引文件的格式与图 6.1-3 会略有不同，以避免重写 OutPutFormat 类。下面根据 MapReduce 的处理过程给出 倒排索引的 设计思路。

- Mapper 的过程



- Combiner 的过程

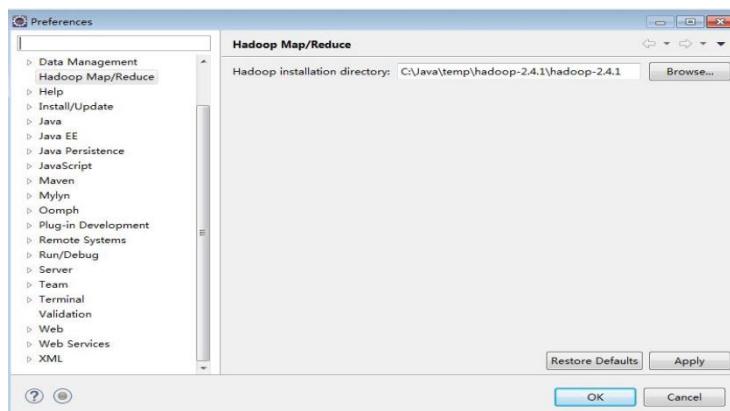
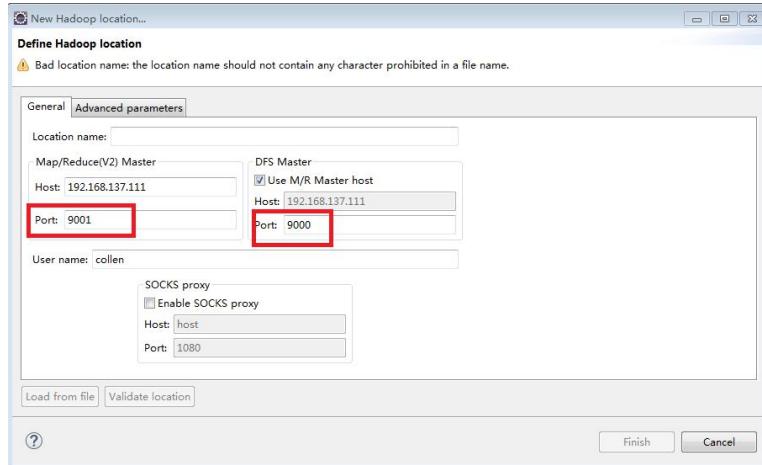


- Reducer 的过程

经过上述两个过程后，Reduce 过程只需将相同 key 值的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行处理了。

## (十) 搭建 Hadoop 的 Eclipse 开发环境





- 设置环境变量：HADOOP\_HOME，并把下面的 bin 目录加入 PATH 路径中：

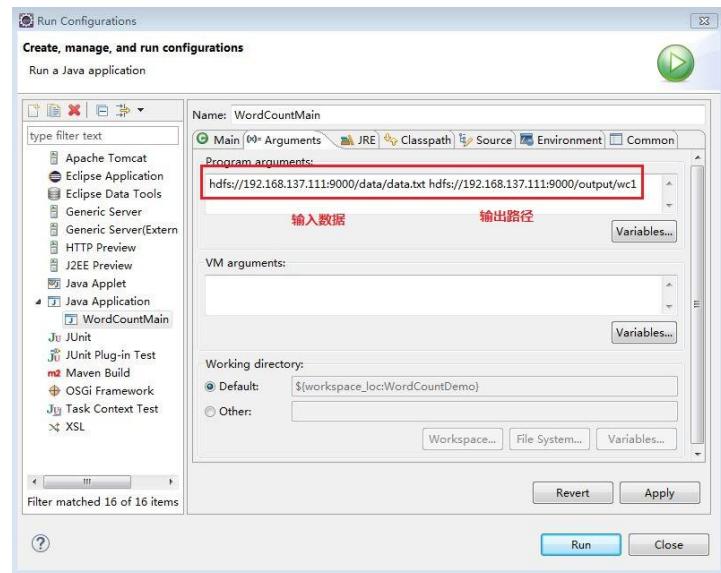
```
C:\Users\collen>echo %HADOOP_HOME%
C:\Java\temp\hadoop-2.4.1\hadoop-2.4.1
```

```
C:\Users\collen>echo %PATH%
C:\Java\temp\hadoop-2.4.1\hadoop-2.4.1\bin;C:\;
1;C:\Program Files (x86)\Sennheiser\SoftphoneS
```

- 将 hadoop.dll 文件复制到 C:\Windows\System32 目录下
- 修改 Hadoop 的源码：org.apache.hadoop.io.nativeio.NativeIO

```
public static boolean access(String path, AccessRight desiredAccess)
    throws IOException {
    //return access0(path, desiredAccess.accessRight());
    return true; 改为永远返回true
}
```

赵强老师：大数据课程课件

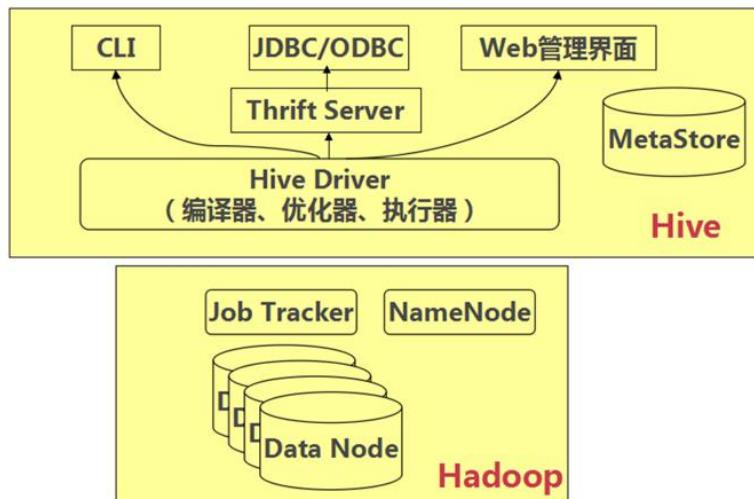


## 八、Hive

### (一) 什么是 Hive

- 构建在 Hadoop 上的**数据仓库**平台，为数据仓库管理提供了许多功能
- 起源自 facebook 由 Jeff Hammerbacher 领导的团队
- 2008 年 facebook 把 hive 项目贡献给 Apache
- 定义了一种类 SQL 语言 HiveQL。可以看成是将 SQL 到 Map-Reduce 的映射器
- 提供 Hive shell、JDBC/ODBC、Thrift 客户端等接

### (二) Hive 的体系结构



- 用户接口主要有三个：CLI，JDBC/ODBC 和 WebUI
  - CLI，即 Shell 命令行
  - JDBC/ODBC 是 Hive 的 Java，与使用传统数据库 JDBC 的方式类似
  - WebGUI 是通过浏览器访问 Hive
- Hive 将元数据存储在数据库中(metastore)，目前只支持 mysql、derby。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 (plan) 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 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成（包含 \* 的查询，比如 select \* from table 不会生成 MapReduce 任务）

## (三) Hive 的管理

- Hive 的安装

- 在虚拟机上安装 MySQL:

- ◆ yum remove mysql-libs
- ◆ rpm -ivh mysql-community-common-5.7.19-1.el7.x86\_64.rpm
- ◆ rpm -ivh mysql-community-libs-5.7.19-1.el7.x86\_64.rpm
- ◆ rpm -ivh mysql-community-client-5.7.19-1.el7.x86\_64.rpm
- ◆ rpm -ivh mysql-community-server-5.7.19-1.el7.x86\_64.rpm
- ◆ rpm -ivh mysql-community-devel-5.7.19-1.el7.x86\_64.rpm (可选)

启动 MySQL: service mysqld start

或者: systemctl start mysqld.service

查看 root 用户的密码: cat /var/log/mysqld.log | grep password

登录后修改密码: alter user 'root'@'localhost' identified by 'Welcome\_1';

- MySQL 数据库的配置:

- ◆ 创建一个新的数据库: create database hive;
- ◆ 创建一个新的用户:

create user 'hiveowner'@'%' identified by 'Welcome\_1';

- ◆ 给该用户授权

grant all on hive.\* TO 'hiveowner'@'%';

grant all on hive.\* TO 'hiveowner'@'localhost' identified by 'Welcome\_1';

- 嵌入式模式（本地模式）

- ◆ 元数据信息存储在内置的 Derby 数据库中
- ◆ 只运行建立一个连接
- ◆ 设置 Hive 的环境变量
- ◆ 设置以下参数:

参数文件	配置参数	参考值
hive-site.xml	javax.jdo.option.ConnectionURL	jdbc:derby:;databaseName=metastore_db;create=true
	javax.jdo.option.ConnectionDriverName	org.apache.derby.jdbc.EmbeddedDriver
	hive.metastore.local	true
	hive.metastore.warehouse.dir	file:///root/training/apache-hive-2.3.0-bin/warehouse

- ◆ **初始化 MetaStore:**

**schematool -dbType derby -initSchema**

- 远程模式
  - ◆ 元数据信息存储在远程的 MySQL 数据库中
  - ◆ 注意一定要使用高版本的 MySQL 驱动（5.1.43 以上的版本）



mysql-connector-java-5.1.43-bin.jar

参数文件	配置参数	参考值
hive-site.xml	javax.jdo.option.ConnectionURL	jdbc:mysql://localhost:3306/hive?useSSL=false
	javax.jdo.option.ConnectionDriverName	com.mysql.jdbc.Driver
	javax.jdo.option.ConnectionUserName	hiveowner
	javax.jdo.option.ConnectionPassword	Welcome_1

◆ 初始话 MetaStore:  
**schematool -dbType mysql -initSchema**

- Hive 的管理
  - CLI (Command Line Interface) 方式
  - HWI (Hive Web Interface) 方式 (只在 2.2.0 以前的版本才有)
    - ◆ 安装步骤:
      - tar -zxvf apache-hive-0.13.0-src.tar.gz
      - jar cvfM0 hive-hwi-0.13.0.war -C web/ .
      - 修改 hive-site.xml

参数文件	配置参数	参考值
hive-site.xml	hive.hwi.listen.host	0.0.0.0
	hive.hwi.listen.port	9999
	hive.hwi.war.file	lib/hive-hwi-0.13.0.war

- 注意：拷贝 **JDK/lib/tools.jar ----> \$HIVE\_HOME/lib**
- ◆ 启动方式: `hive --service hwi &`
- ◆ 用于通过浏览器来访问 hive: <http://<host ip>:9999/hwi/>
- 远程服务: 端口号 10000
- ◆ 启动方式: `hive --service hiveserver &`

## (四) Hive 的数据类型

- 基本数据类型
  - tinyint/smallint/int/bigint: 整数类型
  - float,double: 浮点数类型
  - boolean: 布尔类型
  - string: 字符串类型
- 复杂数据类型
  - Array: 数组类型，由一系列相同数据类型的元素组成

- Map: 集合类型，包含 key->value 键值对，可以通过 key 来访问元素。
- Struct: 结构类型，可以包含不同数据类型的元素。这些元素可以通过“点语法”的方式来得到所需要的元素

- 时间类型

- Date: 从 Hive0.12.0 开始支持
- Timestamp: 从 Hive0.8.0 开始支持

## (五) Hive 的数据模型

- Hive 的数据存储

- 基于 HDFS
- 没有专门的数据存储格式
- 存储结构主要包括：数据库、文件、表、视图
- 可以直接加载文本文件 (.txt 文件)
- 创建表时，指定 Hive 数据的列分隔符与行分隔符

- 表

- Inner Table (内部表)

- ◆ 与数据库中的 Table 在概念上是类似
    - ◆ 每一个 Table 在 Hive 中都有一个相应的目录存储数据
    - ◆ 所有的 Table 数据（不包括 External Table）都保存在这个目录中
    - ◆ 删除表时，元数据与数据都会被删除

```
create table emp
(empno int,
ename string,
job string,
mgr int,
hiredate string,
sal int,
comm int,
deptno int)
row format delimited fields terminated by ',';
```

- Partition Table (分区表)

- ◆ Partition 对应于数据库的 Partition 列的密集索引
    - ◆ 在 Hive 中，表中的一个 Partition 对应于表下的一个目录，所有的 Partition 的数据都存储在对应的目录中

```
create table emp_part
(empno int,
ename string,
job string,
mgr int,
hiredate string,
sal int,
comm int)
partitioned by (deptno int)
row format delimited fields terminated by ',';
```

- ◆ 往分区表中插入数据：

```
insert into table emp_part partition(deptno=10)
    select empno,ename,job,mgr,hiredate,sal,comm from emp where deptno=10;
insert into table emp_part partition(deptno=20)
    select empno,ename,job,mgr,hiredate,sal,comm from emp where deptno=20;
```

◆ 在 Hive 中，通过 SQL 的执行计划获知分区表提高的效率

#### ■ External Table (外部表)

- ◆ 指向已经在 HDFS 中存在的数据，可以创建 Partition
- ◆ 它和内部表在元数据的组织上是相同的，而实际数据的存储则有较大的差异
- ◆ 外部表 只有一个过程，加载数据和创建表同时完成，并不会移动到数据仓库目录中，只是与外部数据建立一个链接。当删除一个外部表时，仅删除该链接

```
create external table ex_student
(sid int, sname string,age int)
row format delimited fields terminated by ','
location '/students';
```

#### ■ Bucket Table (桶表)

- ◆ 桶表是对数据进行哈希取值，然后放到不同文件中存储。
- ◆ 需要设置环境变量：set hive.enforce.bucketing = true;

```
create table emp_bucket
(empno int,
ename string,
job string,
mgr int,
hiredate string,
sal int,
comm int,
deptno int)
clustered by (job) into 4 buckets
row format delimited fields terminated by ',';
```

#### ● 视图 (View)

- 视图是一种虚表，是一个逻辑概念；可以跨越多张表
- 视图建立在已有表的基础上，视图赖以建立的这些表称为基表
- 视图可以简化复杂的查询

```
create view myview as select sname from student1;
```

## (六) Hive 数据的导入

#### ● Hive 支持两种方式的数据导入

- 使用 load 语句导入数据
- 使用 sqoop 导入关系型数据库中的数据

#### ● 使用 load 语句导入数据

- 数据文件：

```
student.csv  
1, Tom, 23  
2, Mary, 24  
3, Mike, 22  
  
create table student(sid int, sname string, age int);
```

- 导入本地的数据文件

```
load data local inpath '/root/training/data/student.csv' into table student;
```

**注意：Hive 默认分隔符是：tab 键。所以需要在建表的时候，指定分隔符。**

```
create table student1  
(sid int, sname string, age int)  
row format delimited fields terminated by ',';
```

- 导入 HDFS 上的数据

```
create table student2  
(sid int, sname string, age int)  
row format delimited fields terminated by ',';
```

```
load data inpath '/input/student.csv' into table student2;
```

- 使用 **sqoop** 导入关系型数据库中的数据

- 将关系型数据的表结构复制到 hive 中

```
sqoop create-hive-table --connect jdbc:mysql://localhost:3306/test --table username --username root --password 123456 --hive-table test
```

其中 --table username 为 mysql 中的数据库 test 中的表 --hive-table test 为 hive 中新建的表名称

- 从关系数据库导入文件到 hive 中

```
sqoop import --connect jdbc:mysql://localhost:3306/test --username root --password mysql-password --table t1 --hive-import
```

- 将 hive 中的表数据导入到 mysql 中

```
sqoop export --connect jdbc:mysql://localhost:3306/test --username root --password admin --table uv_info --export-dir /user/hive/warehouse/uv/dt=2011-08-03
```

## (七) Hive 的查询

- 简单查询

- 过滤和排序

- Hive 的函数

- 数学函数

- ◆ round
- ◆ ceil
- ◆ floor

- 字符函数

- ◆ lower
- ◆ upper
- ◆ length
- ◆ concat
- ◆ substr
- ◆ trim
- ◆ lpad
- ◆ rpad

- 收集函数

- ◆ size

- 日期函数

- ◆ to\_date
- ◆ year
- ◆ month
- ◆ day
- ◆ weekofyear
- ◆ datediff
- ◆ date\_add
- ◆ date\_sub

- 条件函数

- ◆ if
- ◆ coalesce
- ◆ case... when...

- 聚合函数

- ◆ count
- ◆ sum
- ◆ min
- ◆ max
- ◆ avg

## ● Hive 的多表查询

- 只支持：等连接，外连接，左半连接
- 不支持非相等的 join 条件
- <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins>

## ● Hive 的子查询

- hive 只支持：from 和 where 子句中的子查询
- <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SubQueries>

```
hive> select *  
> from emp  
> where emp.deptno in (select deptno from dept where dname='SALES');
```

## (八) Hive 的客户端操作： JDBC

- 首先启动 Hive 远程服务：hiveserver2 &
- 需要 Hive lib 目录下的 jar 包
- 演示案例：查询数据



- 如果出现以下错误：

```
java.lang.RuntimeException: org.apache.hadoop.ipc.RemoteException  
(org.apache.hadoop.security.authorize.AuthorizationException):  
User root is not allowed to impersonate anonymous
```

修改 hadoop 配置文件 etc/hadoop/core-site.xml,加入如下配置项

```
<property>  
  <name>hadoop.proxyuser.root.hosts</name>  
  <value>*</value>  
</property>  
<property>  
  <name>hadoop.proxyuser.root.groups</name>  
  <value>*</value>  
</property>
```

## (十) Hive 的自定义函数

- **Hive 的自定义函数（UDF）： User Defined Function**
  - 可以直接应用于 select 语句，对查询结构做格式化处理后，再输出内容
- **Hive 自定义函数的实现细节**
  - 自定义 UDF 需要继承 org.apache.hadoop.hive.ql.exec.UDF
  - 需要实现 evaluate 函数，evaluate 函数支持重载
- **Hive 自定义函数案例**
  - 案例一：拼加两个字符串

```

1 package demo.UDF;
2
3 import org.apache.hadoop.hive.ql.exec.UDF;
4
5 public class MyConcatString extends UDF {
6
7     public String evaluate(String a,String b){
8         return a.toString()+"*****"+b.toString();
9     }
10 }
```

- 案例二：判断员工表中工资的级别

```

1 package demo.UDF;
2
3 import org.apache.hadoop.hive.ql.exec.UDF;
4
5 public class CheckSalaryGrade extends UDF {
6
7     public String evaluate(String salary){
8         //判断薪水的级别
9         int sal = Integer.parseInt(salary.trim());
10
11         if(sal <1000){
12             return "Grade A";
13         }else if(sal>=1000 && sal<3000){
14             return "Grade B";
15         }else{
16             return "Grade C";
17         }
18     }
19 }
```

- **Hive 自定义函数的部署**

- 把程序打包放到目标机器上去
- 进入 hive 客户端，添加 jar 包：  
hive> add jar /root/temp/udf.jar;
- 创建临时函数：  
hive> create temporary function myconcat as 'udf.ConcatString';  
hive> create temporary function checksalary as 'udf.CheckSalaryGrade';

- **Hive 自定义函数的调用**

- 查询 HQL 语句：  
select myconcat(ename,job) from emp;

- ```
select ename,sal,checksalary(sal) from emp;
```
- 销毁临时函数：  

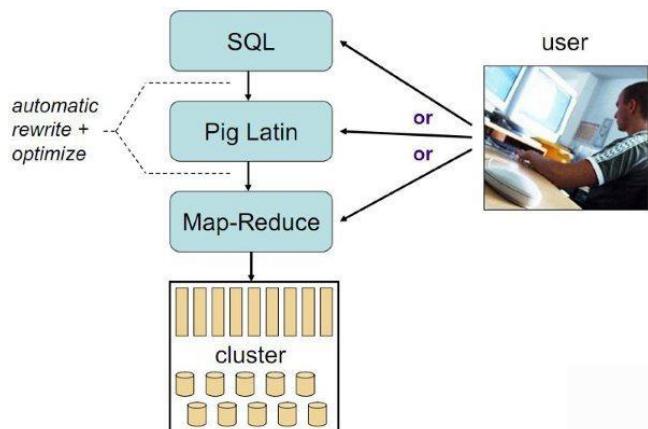
```
hive> DROP TEMPORARY FUNCTION checksalary;
```

## 九、Pig

### (一) 什么是 Pig?

- Pig 是一个用来处理大规模数据集的平台，由 Yahoo! 贡献给 Apache
- **Pig 可以简化 MapReduce 任务的开发**
- Pig 可以看做 **hadoop 的客户端软件**，可以连接到 hadoop 集群进行数据分析工作
- Pig 方便不熟悉 java 的用户，使用一种较为简便的类似二 SQL 的面向数据流的语言 pig Latin 进行数据处理
- **Pig Latin** 可以进行排序、过滤、求和、分组、关联等常用操作，还可以自定义函数，这是一种面向数据分析处理的轻量级脚本语言
- Pig 可以看做是 Pig Latin 到 MapReduce 的映射器
- Pig 可以自动对集群进行分配和回收，开箱即用地对 MapReduce 程序进行优化

### (二) Pig 的体系结构



### (三) Pig 的安装和工作模式

#### ● 安装步骤

- 下载并解压 pig 安装包（<http://pig.apache.org/>）
- 设置环境变量

## ● 工作模式

- 本地模式: pig -x local
- MapReduce 模式
  - ◆ 设置 PIG\_CLASSPATH 环境变量，指向 Hadoop 的配置目录
  - ◆ 启动: pig

## ● Pig 的常用命令

- ls、cd、cat、mkdir、pwd
- copyFromLocal、copyToLocal
- sh
- register、define

# (四) Pig 的内置函数

## ● 计算函数

|            |                                                                                   |
|------------|-----------------------------------------------------------------------------------|
| avg        | 计算包中项的平均值                                                                         |
| concat     | 把两个字节数组或者字符数组连接成一个                                                                |
| count      | 计算包中非空值的个数                                                                        |
| count_star | 计算包中项的个数，包括空值                                                                     |
| diff       | 计算两个包的差                                                                           |
| max        | 计算包中项的最大值                                                                         |
| min        | 计算包中项的最小值                                                                         |
| size       | 计算一个类型的大小，数值型的大小为 1;<br>对于字符数组，返回字符的个数；<br>对于字节数组，返回字节的个数；<br>对于元组，包，映射，返回其中项的个数。 |
| sum        | 计算一个包中项的值的总和                                                                      |
| TOKENIZE   | 对一个字符数组进行标记解析，并把结果词放入一个包                                                          |

## ● 过滤函数

|         |              |
|---------|--------------|
| isempty | 判断一个包或映射是否为空 |
|---------|--------------|

## ● 加载存储函数

|               |                                 |
|---------------|---------------------------------|
| PigStorage    | 用字段分隔文本格式加载或存储关系，这是默认的存储函数      |
| BinStorage    | 从二进制文件加载一个关系或者把关系存储到二进制文件       |
| BinaryStorage | 从二进制文件加载只是包含一个类型为 bytearray 的字段 |

|            |                                     |
|------------|-------------------------------------|
|            | 的元组到关系，或以这种格式存储一个关系                 |
| TextLoader | 从纯文本格式加载一个关系                        |
| PigDump    | 用元组的 <code>toString()</code> 形式存储关系 |

## (五) 使用 Pig Latin 语句分析数据

- **Pig 的数据模型**

Bag: 表

Tuple: 行, 记录

Field: 属性

Pig 不要求同一个 bag 里面的各个 tuple 有相同数量或相同类型的 field

- 注意: 启动 historyserver:

`mr-jobhistory-daemon.sh start historyserver`

可以通过: <http://<IP>:19888/jobhistory> 查看历史任务记录

- 常用的 Pig Latin 语句

- **LOAD:** 指出载入数据的方法

一般形式 :

```
data = LOAD '/Sample/NYSE_dividends'
```

指定映射关系 :

```
data = LOAD '/Sample/NYSE_dividends' AS (exchange, symbol, date, dividend);
```

指定映射关系，并指定分隔符形式

```
data = LOAD '/Sample/data' USING PigStorage('\t') AS (exchange, symbol, date, dividend);
```

- **FOREACH:** 逐行扫描进行某种处理

对数据表进行逐行处理是十分重要的功能，因此 PIG Latin 提供了 FOREACH 命令。通过一组表达式对数据流中的每一行进行运算，产生的结果就是用于下一个算子的数据集。

下面的语句加载整个数据，但最终结果 B 中只保留其中的 user 和 id 字段：

```
A = LOAD 'input' AS (user:chararray, id:long, address:chararray, phone:chararray, preferences:map[]);
B = FOREACH A GENERATE user, id;
```

表达式中可以使用“\*”代表全部列，还可以使用“..”表示范围内的列，这对简化命令文本很有用：

```

prices = LOAD 'NYSE_daily' AS (exchange, symbol, date, open,high, low, close, volume, adj_close);
beginning = FOREACH prices GENERATE ..open; -- exchange, symbol, date, open
middle = FOREACH prices GENERATE open..close; -- open, high, low, close
end = FOREACH prices GENERATE volume..; -- volume, adj_close

```

### ■ **FILTER:** 过滤行

使用 **FILTER** 可以对数据表中的数据进行过滤工作。该关键字经常与 **BY** 一起联用，对表中对应的特征值进行过滤，例如：

```

original = LOAD 'data' AS (name, number, ino);
flitered_data = FILTER original BY number > 1000;

```

### ■ **GROUP 分组**

按照键值相同的规则归并数据。在 SQL 中，**group** 操作必然与聚集函数组合使用，而在 pig 中，**group** 操作将产生与键值有关的 **bag**。

```

daily = LOAD 'NYSE_daily' AS (exchange, stock);
grpds = GROUP daily BY stock;
cnt = FOREACH grpds GENERATE group, COUNT(daily);

```

这里的 **group** 是默认的变量，表示使用 **GROUP** 聚合后得到的数据组，再通过 **FOREACH** 对每一组数据进行单独处理。

```

daily = LOAD 'NYSE_daily' AS (exchange, stock);
grpds = GROUP daily by stock;
STORE grpds INTO 'by_group';

```

### ■ **Distinct 剔除重复项**

数据表中若存在重复项，并且需要剔除这些数据，那么可以使用 **DISTINCT** 命令，剔除数据表中重复项，该方法与 SQL 中的用法一致。

```

daily = LOAD 'NYSE_daily' AS (exchange:chararray, symbol:chararray);
uniq = DISTINCT daily;

```

### ■ **Union 联合**

**Union** 的作用与 SQL 中的 **Union** 相似，使用该操作符，可以将不同的数据表，组合成同一的数据表。假设我们有数据表 A 和表 B，都记录了行车数据：

```

A = LOAD 'table1' USING PigStorage(',') as (carno:int,date:chararray,addno:long);
B = LOAD 'table2' USING PigStorage(',') as (carno:int,date:chararray,addno:long);
X = UNION A, B;
DUMP X;

```

### ■ **Join 链接**

连接一组key：

```
jnd = JOIN daily BY symbol, divs BY symbol;
```

连接两组key：

```
jnd = JOIN daily BY (symbol, date), divs BY (symbol, date);
```

外连接：

```
jnd = JOIN daily BY (symbol, date) left outer, divs BY (symbol, date);
```

多表连接：

```
A = LOAD 'input1' AS (x, y);
B = LOAD 'input2' AS (u, v);
C = LOAD 'input3' AS (e, f);
alpha = JOIN A BY x, B BY u, C BY e;
```

- **DUMP:** 把结果显示到屏幕
- **STORE:** 把结果保存到文件

## ● WordCount 实现

① 加载数据

```
mydata = load '/data/data.txt' as (line:chararray);
```

② 将字符串分割成单词

```
words = foreach mydata generate flatten(TOKENIZE(line)) as word;
```

③ 对单词进行分组

```
grpds = group words by word;
```

④ 统计每组中单词数量

```
cndt = foreach grpds generate group,COUNT(words);
```

⑤ 打印结果

```
dump cndt;
```

## (六) Pig 的自定义函数

### ● 概述

- 支持使用 Java、Python、Javascript 三种语言编写 UDF
- Java 自定义函数较为成熟，其它两种功能还有限
- 需要的 jar 包：
  - ◆ /root/training/pig-0.14.0/pig-0.14.0-core-h2.jar
  - ◆ /root/training/pig-0.14.0/lib
  - ◆ /root/training/pig-0.14.0/lib/h2
  - ◆ /root/training/hadoop-2.4.1/share/hadoop/common
  - ◆ /root/training/hadoop-2.4.1/share/hadoop/common/lib

## ● Pig 函数的类型

- 自定义过滤函数
- 自定义运算函数
- 自定义加载函数

## ● 部署自定义函数

- 注册 jar 包 register /root/temp/pig.jar
- 为自定义函数起别名：define myfilter demo.pig.MyFilterFunction;

## ● 自定义过滤函数

- 示例：如果员工工资大于等于 3000 块钱，则被选择出来。

```
package demo.pig;

import java.io.IOException;
import org.apache.pig.FilterFunc;
import org.apache.pig.data.Tuple;

//如果员工工资大于等于3000块钱，则被选择出来。
public class IsSalaryTooHigh extends FilterFunc{

    /*调用：
     *  * emp = load ***
     *  * emp_toohigh = filter emp by demo.pig.IsSalaryTooHigh(sal);
     */

    @Override
    public Boolean exec(Tuple tuple) throws IOException {
        int sal = (Integer) tuple.get(0);
        return sal >= 3000?true:false;
    }
}
```

- 用法：filter emp by demo.pig.IsSalaryTooHigh(sal)
- 也可以创建函数的别名

## ● 自定义运算函数

- 判断员工薪水是级别
  - ◆ 如果 sal <= 1000， 则为： Grade A
  - ◆ 如果 sal > 1000 && sal<=3000， 则为： Grade B
  - ◆ 如果 sal > 3000， 则为： Grade C
- 参考运行结果

```
(SMITH,800,Grade A)
(ALLEN,1600,Grade B)
(WARD,1250,Grade B)
(JONES,2975,Grade B)
(MARTIN,1250,Grade B)
(BLAKE,2850,Grade B)
(CLARK,2450,Grade B)
(SCOTT,3000,Grade B)
(KING,5000,Grade C)
(TURNER,1500,Grade B)
(ADAMS,1100,Grade B)
(JAMES,950,Grade A)
(FORD,3000,Grade B)
(MILLER,1300,Grade B)
```

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class CheckSalaryGrade extends EvalFunc<String> {

    @Override
    public String exec(Tuple tuple) throws IOException {
        /*
         * 如果sal <= 1000, 则为, Grade A
         * 如果sal > 1000 && sal<=3000, 则为, Grade B
         * 如果sal > 3000, 则为, Grade C
         */

        //员工的薪水
        int sal = (Integer) tuple.get(0);

        if(sal<=1000){
            return "Grade A";
        }else if(sal>1000 && sal <=3000){
            return "Grade B";
        }else{
            return "Grade C";
        }
    }
}
```

- 用法: emp1 = foreach emp generate ename,demo.pig.CheckSalaryGrade(sal);

## ● 自定义加载函数

- 默认情况下，一行数据会被解析成一个 Tuple
- 比如：员工信息

```
(7369,SMITH,CLERK,7902,1980/12/17,800,,20)
(7499,ALLEN,SALESMAN,7698,1981/2/20,1600,300,30)
(7521,WARD,SALESMAN,7698,1981/2/22,1250,500,30)
(7566,JONES,MANAGER,7839,1981/4/2,2975,,20)
(7654,MARTIN,SALESMAN,7698,1981/9/28,1250,1400,30)
(7698,BLAKE,MANAGER,7839,1981/5/1,2850,,30)
(7782,CLARK,MANAGER,7839,1981/6/9,2450,,10)
(7788,SCOTT,ANALYST,7566,1987/4/19,3000,,20)
(7839,KING,PRESIDENT,,1981/11/17,5000,,10)
(7844,TURNER,SALESMAN,7698,1981/9/8,1500,0,30)
(7876,ADAMS,CLERK,7788,1987/5/23,1100,,20)
(7900,JAMES,CLERK,7698,1981/12/3,950,,30)
(7902,FORD,ANALYST,7566,1981/12/3,3000,,20)
(7934,MILLER,CLERK,7782,1982/1/23,1300,,10)
```

- 特殊情况：单词统计的时候。这时候：希望每个单词能被解析成一个 Tuple，从而便于处理

```
I love Beijing
I love China
Beijing_is the capital of China
```

- 需要 MapReduce 的库（jar 文件）

```

public Tuple getNext() throws IOException {
    Tuple result = null;
    try{
        if(!reader.nextKeyValue()){//表示读取完了
            return null;
        }

        result = TupleFactory.getInstance().newTuple();
        Text value = (Text) reader.getCurrentValue();
        String str = value.toString();

        //生成一个bag
        DataBag bag = BagFactory.getInstance().newDefaultBag();

        //分词
        String[] words = str.split(" ");
        for(String word:words){
            Tuple t = TupleFactory.getInstance().newTuple();
            t.append(word);

            bag.add(t);
        }
        result.append(bag);
    }catch(Exception ex){
        ex.printStackTrace();
    }
    return result;
}

```

```

public class MyLoadFunction extends LoadFunc{

    private RecordReader reader;

    @Override
    public InputFormat getInputFormat() throws IOException {
        return new TextInputFormat();
    }

    @Override
    public void prepareToRead(RecordReader reader, PigSplit split)
        throws IOException {
        this.reader = reader;
    }

    @Override
    public void setLocation(String location, Job job) throws IOException {
        FileInputFormat.setInputPaths(job, location);
    }

    ...
}

```

- 运行加载函数: records = load '/input/data.txt' using demo.pig.MyLoadFunction();

```

((I), (love), (Beijing))
((I), (love), (China))
(({Beijing}, (is), (the), (capital), (of), (China)))

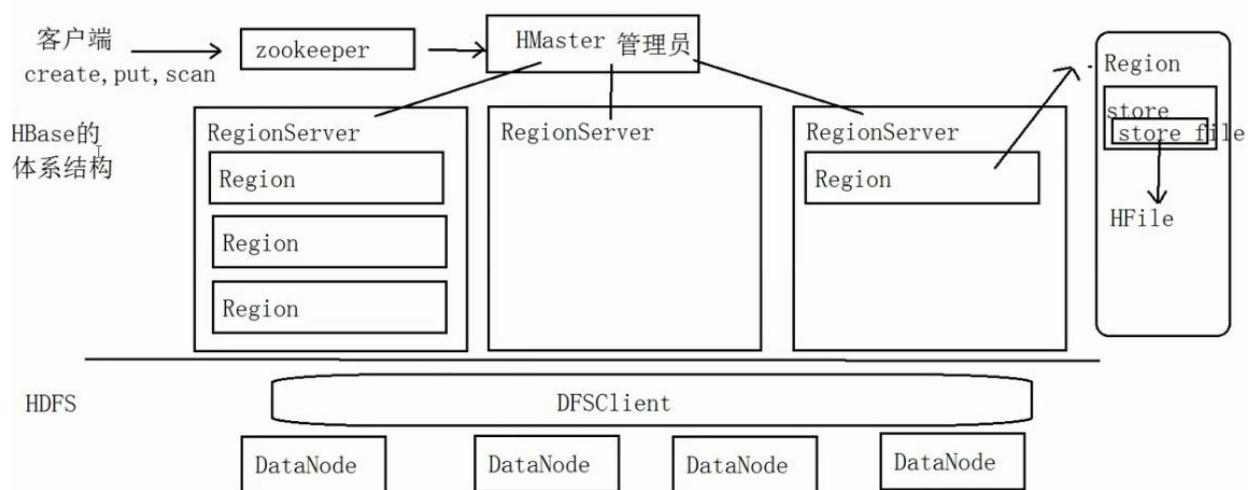
```

## 十、HBase

### (一) 什么是 HBase?

HBase 是一个分布式的、面向列的开源数据库，该技术来源于 Fay Chang 所撰写的 Google 论文“Bigtable：一个结构化数据的分布式存储系统”。就像 Bigtable 利用了 Google 文件系统（File System）所提供的分布式数据存储一样，HBase 在 Hadoop 之上提供了类似于 Bigtable 的能力。HBase 是 Apache 的 Hadoop 项目的子项目。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同的是 HBase 基于列的而不是基于行的模式。

### (二) HBase 的体系结构



### (三) HBase 的表结构

HBase的表结构  
students表  
一个列族就是一个region

| rowkey<br>(行键) | info 列族 |        |     | grade   |      |         |
|----------------|---------|--------|-----|---------|------|---------|
|                | name    | gender | age | chinese | math | english |
| stu001         | Tom     |        |     |         |      |         |
| stu001         |         | 男      |     |         |      |         |
| stu001         |         |        |     |         | 90   |         |
| stu002         |         |        | 28  |         |      |         |
|                |         |        |     |         |      |         |

1. rowkey是可以重复的  
 2. 相同rowkey的记录是同一行记录  
  
 create  
 'students','info','grade'  
 ;

### (四) HBase 的安装和部署

- 本地模式
- 伪分布模式
- 集群模式

#### 本地模式的配置

| 参数文件           | 配置参数          | 参考值                                    |
|----------------|---------------|----------------------------------------|
| .bash_profile  | HBASE_HOME    | /root/training/hbase-1.3.1             |
| hbase-env.sh   | JAVA_HOME     | /root/training/jdk1.8.0_144            |
| hbase-site.xml | hbase.rootdir | file:///root/training/hbase-1.3.1/data |

#### 伪分布模式

| 参数文件           | 配置参数                      | 参考值                               |
|----------------|---------------------------|-----------------------------------|
| .bash_profile  | HBASE_HOME                | /root/training/hbase-1.3.1        |
| hbase-env.sh   | JAVA_HOME                 | /root/training/jdk1.8.0_144       |
|                | HBASE_MANAGES_ZK          | true                              |
| hbase-site.xml | hbase.rootdir             | hdfs://192.168.157.111:9000/hbase |
|                | hbase.cluster.distributed | true                              |
|                | hbase.zookeeper.quorum    | 192.168.157.111                   |
|                | dfs.replication           | 1                                 |
| regionservers  |                           | 192.168.157.111                   |

## 集群模式

| 参数文件           | 配置参数                      | 参考值                                |
|----------------|---------------------------|------------------------------------|
| .bash_profile  | HBASE_HOME                | /root/training/hbase-1.3.1         |
| hbase-env.sh   | JAVA_HOME                 | /root/training/jdk1.8.0_144        |
|                | HBASE_MANAGES_ZK          | true                               |
| hbase-site.xml | hbase.rootdir             | hdfs://192.168.157.111:9000/hbase  |
|                | hbase.cluster.distributed | true                               |
|                | hbase.zookeeper.quorum    | 192.168.157.111                    |
|                | dfs.replication           | 2                                  |
|                | hbase.master.maxclockskew | 180000                             |
| regionservers  |                           | 192.168.157.113<br>192.168.157.114 |

注意每台机器的时间，可以使用 `date -s 11/22/2016` 设置时间

## HBase Web Console (端口: 16010)

The screenshot shows the Apache HBase Web Console interface at port 16010. The top navigation bar includes links for Home, Table Details, Procedures, Local Logs, Log Level, Debug Dump, Metrics Dump, and HBase Configuration. The main content area is divided into two sections:

- Region Servers**: Displays a table with columns: Base Stats, Memory, Requests, Storefiles, Compactions, ServerName, Start time, Version, Requests Per Second, and Num. Regions. One row is shown for the master server: bigdata11,16020,1504513076576, started on Mon Sep 04 16:17:56 CST 2017, version 1.3.1, with 0 requests per second and 3 regions.
- Backup Masters**: Displays a table with columns: ServerName, Port, and Start Time. One row is shown for the backup master: Total:0.

## (五) -ROOT-和.META.

- HBase 中有两张特殊的 Table, -ROOT-和.META.
  - ◆ -ROOT- : 记录了.META.表的 Region 信息, -ROOT-只有一个 region
  - ◆ .META. : 记录了用户创建的表的 Region 信息, .META.可以有多个 region
- Zookeeper 中记录了-ROOT-表的 location
- Client 访问用户数据之前需要首先访问 zookeeper, 然后访问-ROOT-表, 接着访问.META.表, 最后才能找到用户数据的位置去访问。

## (六) HBase Shell

- HBase 提供了一个 shell 的终端给用户交互

| 名称            | 命令表达式                                               |
|---------------|-----------------------------------------------------|
| 创建表           | create '表名称', '列族名称 1','列族名称 2','列族名称 N'            |
| 添加记录          | put '表名称', '行名称', '列名称:', '值'                       |
| 查看记录          | get '表名称', '行名称'                                    |
| 查看表中的记录数      | count '表名称'                                         |
| 删除记录          | delete '表名', '行名称', '列名称'                           |
| 删除一张表         | 先要屏蔽该表, 才能对该表进行删除, 第一步 disable '表名称' 第二步 drop '表名称' |
| 查看所有记录        | scan "表名称"                                          |
| 查看某个表某个列中所有数据 | scan "表名称", {COLUMNS=>'列族名称:列名称'}                   |
| 更新记录          | 就是重写一遍进行覆盖                                          |

## (七) HBase 的 Java 编程接口

- 创建表

```
@Test
public void testCreateTable() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //创建客户端
    HBaseAdmin admin = new HBaseAdmin(conf);
    //创建表的描述信息
    HTableDescriptor htd = new HTableDescriptor(TableName.valueOf("students"));
    //创建列族
    HColumnDescriptor h1 = new HColumnDescriptor("info");
    HColumnDescriptor h2 = new HColumnDescriptor("grade");
    //将列加入列族
    htd.addFamily(h1);
    htd.addFamily(h2);

    //创建表
    admin.createTable(htd);
    admin.close();
}
```

- 删除表

```
@Test
public void testDropTable() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //创建客户端
    HBaseAdmin admin = new HBaseAdmin(conf);

    //disable 这张表
    admin.disableTable(TableName.valueOf("students"));
    //删除表
    admin.deleteTable(TableName.valueOf("students"));

    admin.close();
}
```

- 插入单条数据

```
@Test
public void testPut() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //指定表名
    HTable table = new HTable(conf, "students");
    //创建一条数据，行键
    Put put = new Put(Bytes.toBytes("stu000"));
    //指定数据 family 列族 qualifier 列 value 值
    put.add(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes("Tom"));

    //插入数据
    table.put(put);
    table.close();
}
```

- 插入多条数据

```
public void testPutList() throws Exception{
    //一次插入多条数据

    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //指定表名
    HTable table = new HTable(conf, "students");

    //构造集合代表要插入的数据
    List<Put> list = new ArrayList<Put>();
    for(int i=1;i<11;i++){
        Put put = new Put(Bytes.toBytes("stu00"+i));
        put.add(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes("Tom"+i));

        //将数据加入集合
        list.add(put);
    }

    table.put(list);
    table.close();
}
```

- 查询数据

```
public void testGet() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //指定要查询的表
    HTable table = new HTable(conf, "students");

    //通过get查询，指定行键
    Get get = new Get(Bytes.toBytes("stu001"));
    //执行查询
    Result result = table.get(get);

    //输出结果
    System.out.println(Bytes.toString(result.getValue(Bytes.toBytes("info"), Bytes.toBytes("name"))));
    table.close();
}
```

- 扫描表的数据

```
public void testScan() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //指定要查询的表
    HTable table = new HTable(conf, "students");
    //创建一个Scanner
    Scan scanner = new Scan();

    //扫描表
    ResultScanner result = table.getScanner(scanner);

    //打印返回的值
    for(Result r :result){
        System.out.println(Bytes.toString(r.getValue(Bytes.toBytes("info"), Bytes.toBytes("name"))));
    }

    //关闭客户端
    table.close();
}
```

## (八) HBase 上的过滤器



DataInit.java

- 列值过滤器

```
public void testSingleColumnValueFilter() throws Exception{
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");
    //创建客户端连接
    HTable table = new HTable(conf, "emp");
    //创建一个Scanner
    Scan scanner = new Scan();
    //创建一个Filter: SingleColumnValueFilter: 工资等于3000的员工
    SingleColumnValueFilter filter = new SingleColumnValueFilter(Bytes.toBytes("empinfo"), //列族
   Bytes.toBytes("sal"), //列
   CompareOp.EQUAL, //比较的规则
   Bytes.toBytes("3000")); //值

    //使用创建的过滤器
    scanner.setFilter(filter);
    //查询数据
    ResultScanner result = table.getScanner(scanner);
    for(Result r:result){
        //打印
        System.out.println(Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("ename"))));
    }
    //关闭客户端
    table.close();
}
```

- 列名前缀过滤器

```
public void testColumnPrefixFilter() throws Exception{
    //查询员工的薪水
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");
    //创建客户端查询表----> emp
    HTable table = new HTable(conf, "emp");
    //创建一个Scanner
    Scan scanner = new Scan();
    //创建一个Filter:

    //使用创建的过滤器: ColumnPrefixFilter 列名前缀过滤器
    ColumnPrefixFilter filter = new ColumnPrefixFilter(Bytes.toBytes("sal"));
    scanner.setFilter(filter);
    //查询数据
    ResultScanner result = table.getScanner(scanner);
    for(Result r:result){
        //打印
        System.out.println(Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("sal"))));
    }
    //关闭客户端
    table.close();
}
```

## ● 多个列名前缀过滤器

```
public void testMultipleColumnPrefixFilter() throws Exception{
    //查询员工的姓名和薪水
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");
    //创建客户端查询表----> emp
    HTable table = new HTable(conf, "emp");
    //创建一个Scanner
    Scan scanner = new Scan();
    //指定我们的多个列: 姓名和薪水
    byte[][] prefixes = new byte[][]{Bytes.toBytes("ename"),Bytes.toBytes("sal")};
    //创建一个Filter:
    MultipleColumnPrefixFilter filter = new MultipleColumnPrefixFilter(prefixes);
    //使用创建的过滤器: MultipleColumnPrefixFilter
    scanner.setFilter(filter);
    //查询数据
    ResultScanner result = table.getScanner(scanner);
    for(Result r:result){
        String ename = Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("ename")));
        String sal = Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("sal")));

        //打印
        System.out.println(ename+"\t"+sal);
    }
    //关闭客户端
    table.close();
}
```

## ● Rowkey 过滤器

```
public void testRowFilter() throws Exception{
    //查询员工号是7839的姓名和薪水
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");
    //创建客户端查询表----> emp
    HTable table = new HTable(conf, "emp");
    //创建一个Scanner
    Scan scanner = new Scan();
    //创建一个Filter: RowFilter 行键过滤器
    RowFilter filter = new RowFilter(CompareOp.EQUAL, new RegexStringComparator("7839"));
    //使用创建的过滤器:
    scanner.setFilter(filter);
    //查询数据
    ResultScanner result = table.getScanner(scanner);
    for(Result r:result){
        String ename = Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("ename")));
        String sal = Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("sal")));
        String deptno = Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("deptno")));

        //打印
        System.out.println(ename+"\t"+sal+"\t"+deptno);
    }
    //关闭客户端
    table.close();
}
```

## ● 在查询中使用多个过滤器

```

public void testFilter() throws Exception{
    //查询员工号是7839的姓名 第一个过滤器：rowkey过滤器 第二个过滤器：列名前缀过滤器
    //配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");
    //连接客户端端口调表----> emp
    HTable table = new HTable(conf, "emp");
    //第一个过滤器：rowkey过滤器
    RowFilter filter1 = new RowFilter(CompareOp.EQUAL, new RegexStringComparator("7839"));
    //第二个过滤器：列名前缀过滤器
    ColumnPrefixFilter filter2 = new ColumnPrefixFilter(Bytes.toBytes("ename"));
    //创建Filter的list
    FilterList list = new FilterList(Operator.MUST_PASS_ONE);
    list.addFilter(filter1);
    list.addFilter(filter2);
    //创建一个Scanner
    Scan scanner = new Scan();
    scanner.setFilter(list);
    //查询数据
    ResultScanner result = table.getScanner(scanner);
    for(Result r:result){
        //打印
        System.out.println(Bytes.toString(r.getValue(Bytes.toBytes("empinfo"), Bytes.toBytes("ename"))));
    }
    //关闭客户端
    table.close();
}

```

## (九) HBase 上的 MapReduce

- 实现 WordCount

建立测试数据：

```

create 'word','content'
put 'word','1','content:info','I love Beijing'
put 'word','2','content:info','I love China'
put 'word','3','content:info','Beijing is the capital of China'

```

```
create 'stat','content'
```

- Mapper

```

public class MyMapper extends TableMapper<Text, IntWritable> {
    //key2      value2
    @Override
    protected void map(ImmutableBytesWritable key, Result value, Context context)
        throws IOException, InterruptedException {
        //读入的数据：HBase表中的数据----> word表
        String words = Bytes.toString(value.getValue(Bytes.toBytes("content"), Bytes.toBytes("info")));
        //分词： I love Beijing
        String[] itr = words.split(" ");
        for(String w:itr){
            //直接输出
            Text w1 = new Text();
            w1.set(w);
            context.write(w1, new IntWritable(1));
        }
    }
}

```

- Reducer

```

public class MyReducer extends TableReducer<Text, IntWritable, ImmutableBytesWritable> {
    // key3      value3      输出的结果
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        //求和
        int sum = 0;
        for(IntWritable val:values){
            sum += val.get();
        }
        //输出----> HBase的表
        //构造Put，可以使用key作为行键
        Put put = new Put(Bytes.toBytes(key.toString()));
        //封装数据
        put.add(Bytes.toBytes("content"), Bytes.toBytes("info"), Bytes.toBytes(String.valueOf(sum)));
        //写入HBase
        context.write(new ImmutableBytesWritable(Bytes.toBytes(key.toString())), put);
    }
}

```

- 主程序

```
public static void main(String[] args) throws Exception {
    //指定配置信息
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.81");

    //创建job
    Job job = new Job(conf);
    //设定任务的入口
    job.setJarByClass(MyMain.class);

    //创建scan
    Scan scan = new Scan();
    //可以指定查询的某一列
    scan.addColumn(Bytes.toBytes("content"), Bytes.toBytes("info"));

    //创建查询Hbase表的Mapper
    TableMapReduceUtil.initTableMapperJob("word", scan, MyMapper.class, Text.class, IntWritable.class, job);

    //创建写入Hbase的Reducer
    TableMapReduceUtil.initTableReducerJob("stat", MyReducer.class, job);

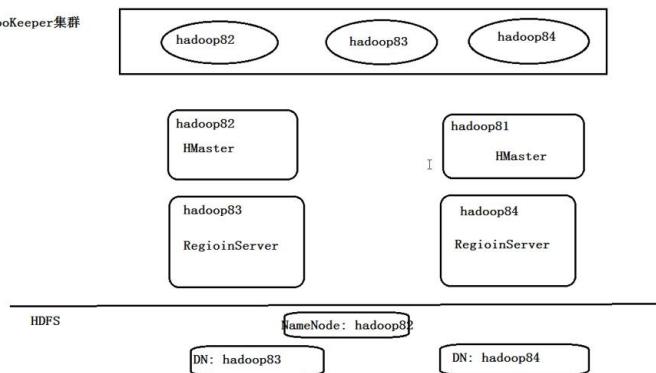
    //执行任务
    job.waitForCompletion(true);
}
```

- `export HADOOP_CLASSPATH=$HBASE_HOME/lib/*:$CLASSPATH`

- 练习：求每个部门的工资总额。

## (十) HBase 的 HA

- 架构



- 配置参数: `hbase.zookeeper.quorum`
- 单独启动 HMaster: `hbase-daemon.sh start master`

## 十一、Sqoop

### (一) 什么是 Sqoop?

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

Sqoop 项目开始于 2009 年，最早是作为 Hadoop 的一个第三方模块存在，后来为了让使用者能够快速部署，也让开发人员能够更快速的迭代开发，Sqoop 独立成为一个 Apache 项目。

### (二) Sqoop 是如何工作？

- ◆ 利用 JDBC 连接关系型数据库
- ◆ 安装包: sqoop-1.4.5-bin\_hadoop-0.23.tar.gz

### (三) 使用 Sqoop

| 命令                | 说明                                                                                      |
|-------------------|-----------------------------------------------------------------------------------------|
| codegen           | 将关系数据库表映射为一个 Java 文件、Java class 类、以及相关的 jar 包                                           |
| create-hive-table | 生成与关系数据库表的表结构对应的 HIVE 表                                                                 |
| eval              | 以快速地使用 SQL 语句对关系数据库进行操作，这可以使得在使用 import 这种工具进行数据导入的时候，可以预先了解相关的 SQL 语句是否正确，并能将结果显示在控制台。 |
| export            | 从 hdfs 中导数据到关系数据库中                                                                      |
| help              |                                                                                         |
| import            | 将数据库表的数据导入到 HDFS 中                                                                      |
| import-all-tables | 将数据库中所有的表的数据导入到 HDFS 中                                                                  |
| job               | 用来生成一个 sqoop 的任务，生成后，该任务并不执行，除非使用命令执行该任务。                                               |
| list-databases    | 打印出关系数据库所有的数据库名                                                                         |
| list-tables       | 打印出关系数据库某一数据库的所有表名                                                                      |
| merge             | 将 HDFS 中不同目录下面的数据合在一起，并存放在指定的目录中                                                        |
| metastore         | 记录 sqoop job 的元数据信息                                                                     |
| version           | 显示 sqoop 版本信息                                                                           |

### 注意：大小写

#### 实例一：

```
sqoop codegen --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --table EMP
```

也可以写成下面的形式：

```
sqoop codegen \  
--connect jdbc:oracle:thin:@192.168.137.129:1521:orcl \  
--username SCOTT --password tiger  
--table EMP
```

#### 实例二：

```
sqoop create-hive-table --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --table EMP --hive-table emphive
```

#### 实例三：

```
sqoop eval --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --query "select ename from emp where deptno=10"
```

```
sqoop eval --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT --password tiger  
--query "select ename,dname from emp,dept where emp.deptno=dept.deptno"
```

#### 实例四：

```
sqoop export --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --table STUDENTS --export-dir /students
```

#### 实例五：

```
sqoop import --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --table EMP --target-dir /emp
```

#### 实例六：

```
sqoop import-all-tables --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger -m 1
```

#### 实例七：

```
sqoop list-databases --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SYSTEM  
--password password
```

#### 实例八：

```
sqoop list-tables --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger
```

#### 实例九：

`sqoop version`

**实例十：将数据导入 HBase（需要事先将表创建）**

```
sqoop import --connect jdbc:oracle:thin:@192.168.137.129:1521:orcl --username SCOTT  
--password tiger --table EMP --columns empno,ename,sal,deptno --hbase-table emp  
--hbase-row-key empno --column-family empinfo
```

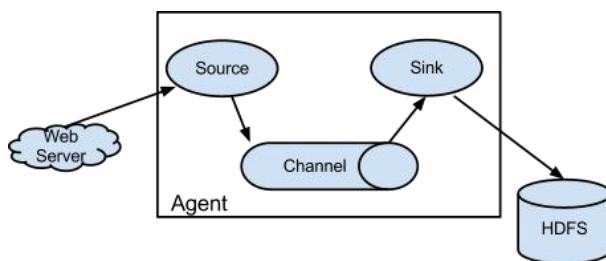
## 十二、Flume

### (一) 什么是 Flume?

Flume 是 Cloudera 提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume 支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。



### (二) Flume 的体系结构



### (三) 安装和配置 Flume

- 1、解压
- 2、修改 conf/flume-env.sh 设置 JAVA\_HOME 即可

### (四) 使用 Flume 采集日志数据

#### 案例一：

```
#bin/flume-ng agent -n a1 -f myagent/a1.conf -c conf -Dflume.root.logger=INFO,console
#定义 agent 名， source、channel、sink 的名称
a1.sources = r1
a1.channels = c1
a1.sinks = k1

#具体定义 source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 8888
```

```
#具体定义 channel  
a1.channels.c1.type = memory  
a1.channels.c1.capacity = 1000  
a1.channels.c1.transactionCapacity = 100
```

```
#具体定义 sink  
a1.sinks.k1.type = logger
```

```
#组装 source、channel、sink  
a1.sources.r1.channels = c1  
a1.sinks.k1.channel = c1
```

## 案例二：

```
#bin/flume-ng agent -n a2 -f myagent/a2.conf -c conf -Dflume.root.logger=INFO,console  
#定义 agent 名， source、channel、sink 的名称  
a2.sources = r1  
a2.channels = c1  
a2.sinks = k1
```

```
#具体定义 source  
a2.sources.r1.type = exec  
a2.sources.r1.command = tail -f /home/hadoop/a.log
```

```
#具体定义 channel  
a2.channels.c1.type = memory  
a2.channels.c1.capacity = 1000  
a2.channels.c1.transactionCapacity = 100
```

```
#具体定义 sink  
a2.sinks.k1.type = logger
```

```
#组装 source、channel、sink  
a2.sources.r1.channels = c1  
a2.sinks.k1.channel = c1
```

### 案例三：

```
#bin/flume-ng agent -n a3 -f myagent/a3.conf -c conf -Dflume.root.logger=INFO,console
#定义 agent 名， source、channel、sink 的名称
a3.sources = r1
a3.channels = c1
a3.sinks = k1

#具体定义 source
a3.sources.r1.type = spooldir
a3.sources.r1.spoolDir = /root/training/logs1

#具体定义 channel
a3.channels.c1.type = memory
a3.channels.c1.capacity = 1000
a3.channels.c1.transactionCapacity = 100

#具体定义 sink
a3.sinks.k1.type = logger

#组装 source、channel、sink
a3.sources.r1.channels = c1
a3.sinks.k1.channel = c1
```

### 案例四：

```
#bin/flume-ng agent -n a4 -f myagent/a4.conf -c conf -Dflume.root.logger=INFO,console
#定义 agent 名， source、channel、sink 的名称
a4.sources = r1
a4.channels = c1
a4.sinks = k1

#具体定义 source
a4.sources.r1.type = spooldir
a4.sources.r1.spoolDir = /root/training/logs

#具体定义 channel
a4.channels.c1.type = memory
a4.channels.c1.capacity = 10000
a4.channels.c1.transactionCapacity = 100

#定义拦截器，为消息添加时间戳
```

```
a4.sources.r1.interceptors = i1
a4.sources.r1.interceptors.i1.type
org.apache.flume.interceptor.TimestampInterceptor$Builder
=



#具体定义 sink
a4.sinks.k1.type = hdfs
a4.sinks.k1.hdfs.path = hdfs://192.168.56.111:9000/flume/%Y%m%d
a4.sinks.k1.hdfs.filePrefix = events-
a4.sinks.k1.hdfs fileType = DataStream

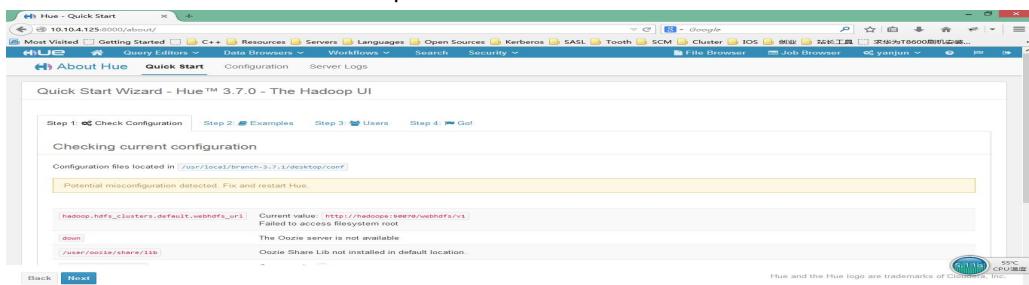
#不按照条数生成文件
a4.sinks.k1.hdfs.rollCount = 0
#HDFS 上的文件达到 128M 时生成一个文件
a4.sinks.k1.hdfs.rollSize = 134217728
#HDFS 上的文件达到 60 秒生成一个文件
a4.sinks.k1.hdfs.rollInterval = 60

#组装 source、channel、sink
a4.sources.r1.channels = c1
a4.sinks.k1.channel = c1
```

## 十三、HUE

### (一) 什么是 HUE?

Hue 是一个开源的 Apache Hadoop UI 系统，最早是由 Cloudera Desktop 演化而来，由 Cloudera 贡献给开源社区，它是基于 Python Web 框架 Django 实现的。通过使用 Hue 我们可以在浏览器端的 Web 控制台上与 Hadoop 集群进行交互来分析处理数据，例如操作 HDFS 上的数据，运行 MapReduce Job 等等。



### (二) HUE 所需要的 rpm 包

|                   |                                            |                                |
|-------------------|--------------------------------------------|--------------------------------|
| ant               | krb5-devel                                 | mysql                          |
| asciidoc          | libtidy (for unit tests only)              | mysql-devel                    |
| cyrus-sasl-devel  | libxml2-devel                              | openldap-devel                 |
| cyrus-sasl-gssapi | libxslt-devel                              | python-devel                   |
| gcc               | make                                       | sqlite-devel                   |
| gcc-c++           | mvn (from maven package or maven3 tarball) | openssl-devel (for version 7+) |
|                   |                                            | gmp-devel                      |

#### 步骤：

- 挂载光盘: mount /dev/cdrom /root/cdroom/
- 替换 yum 的源文件:



- 切换到/root/cdroom/Packages 目录，执行下面的语句:

```
yum install ant-1.7.1-13.el6.i686.rpm cyrus-sasl-devel-2.1.23-15.el6.i686.rpm  
gcc-4.4.7-11.el6.i686.rpm gcc-c++-4.4.7-11.el6.i686.rpm krb5-devel-1.10.3-33.el6.i686.rpm  
libtidy-0.99.0-19.20070615.1.el6.i686.rpm libxml2-devel-2.7.6-14.0.1.el6_5.2.i686.rpm  
libxslt-devel-1.1.26-2.0.2.el6_3.1.i686.rpm mysql-5.1.73-3.el6_5.i686.rpm  
mysql-devel-5.1.73-3.el6_5.i686.rpm openldap-devel-2.4.39-8.el6.i686.rpm  
python-devel-2.6.6-52.el6.i686.rpm sqlite-devel-3.6.20-1.el6.i686.rpm  
gmp-devel-4.3.1-7.el6_2.2.i686.rpm
```

### (三) HUE 与 Hadoop 集成

- 编辑 Hadoop 的配置文件

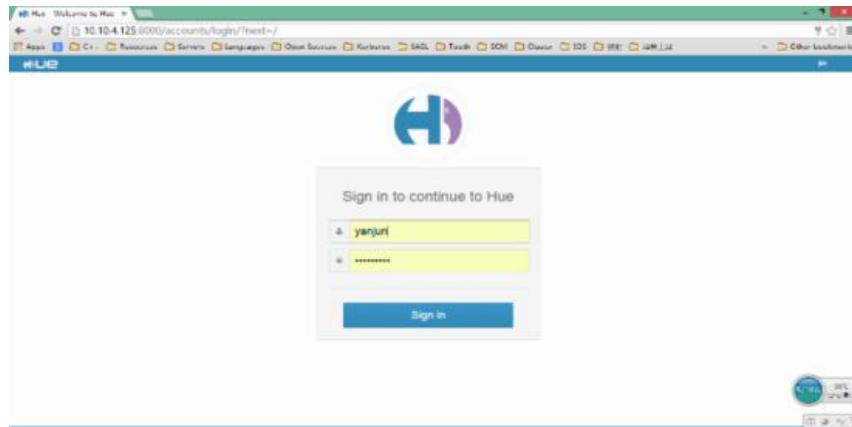
| 参数文件          | 参数                           | 参考值  | 说明                 |
|---------------|------------------------------|------|--------------------|
| hdfs-site.xml | dfs.webhdfs.enabled          | true | 开启 webhdfs 功能      |
| core-site.xml | hadoop.proxyuser.root.hosts  | *    | 设置 Hadoop 集群的代理用户  |
|               | hadoop.proxyuser.root.groups | *    | 设置 Hadoop 集群的代理用户组 |

- 安装 HUE
  - 解压：
    - ◆ tar -zxvf hue-3.7.0-cdh5.4.2.tar.gz
  - 编译安装：注意系统时间
    - ◆ PREFIX=/root/training/ make install
  - 添加用户 hue
    - ◆ adduser hue
    - ◆ chown -R hue.hue /root/training/hue/
- 修改 hue.ini (\$HUE\_HOME/desktop/conf) 参数文件 (hue.ini)

| 参数                      | 参考值                                     |
|-------------------------|-----------------------------------------|
| http_host               | 192.168.137.111                         |
| http_port               | 8888                                    |
| server_user             | root                                    |
| server_group            | root                                    |
| default_user            | root                                    |
| default_hdfs_superuser  | root                                    |
| fs_defaultfs            | hdfs://192.168.137.111:9000             |
| webhdfs_url             | http://192.168.137.111:50070/webhdfs/v1 |
| hadoop_conf_dir         | /root/training/hadoop-2.4.1/etc/hadoop  |
| resourcemanager_host    | 192.168.137.111                         |
| resourcemanager_api_url | http://192.168.137.111:8088             |
| proxy_api_url           | http://192.168.137.111:8088             |
| history_server_api_url  | http://192.168.137.111:19888            |

- 启动 HUE
  - 启动 Hadoop 相关组件
  - 启动 Hue: bin/supervisor

访问首页：<http://192.168.137.111:8888/>



## (四) HUE 与 HBase 集成

- Hue 需要读取 Hbase 的数据是使用 thrift 的方式，默认 HBase 的 thrift 服务没有开启，所有需要手动额外开启 thrift 服务。  
thrift service 默认使用的是 9090 端口，使用如下命令查看端口是否被占用  
`netstat -nl | grep 9090`
- 启动 HBase thrift service  
`hbase-daemon.sh start thrift`
- 配置 HUE  
`hbase_clusters=(Cluster|192.168.137.111:9090)`  
`hbase_conf_dir=/root/training/hbase-0.96.2-hadoop2/conf`

## (五) HUE 与 Hive 集成

- 配置 HUE  
`hive_server_host=192.168.137.111`  
`hive_server_port=10000`  
`hive_conf_dir=/root/training/apache-hive-0.13.0-bin/conf`
- 启动 Hive  
`hive --service metastore`  
`hive --service hiveserver2`

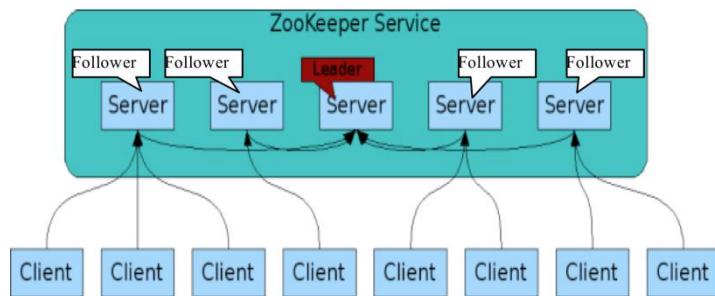


## 十四、ZooKeeper

### (一) 什么是 ZooKeeper?

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

### (二) ZooKeeper 的体系结构



### (三) Zookeeper 能帮我们做什么？

- Hadoop2.0, 使用 Zookeeper 的事件处理确保整个集群只有一个活跃的 NameNode, 存储配置信息等.
- HBase, 使用 Zookeeper 的事件处理确保整个集群只有一个 HMaster, 察觉 HRegionServer 联机和宕机, 存储访问控制列表等.

### (四) 安装和配置 Zookeeper

#### 1、在主节点（hadoop112）上配置 ZooKeeper

(\* ) 配置 /root/training/zookeeper-3.4.6/conf/zoo.cfg 文件  
dataDir=/root/training/zookeeper-3.4.6/tmp

```
server.1=hadoop112:2888:3888  
server.2=hadoop113:2888:3888  
server.3=hadoop114:2888:3888
```

(\*) 在/root/training/zookeeper-3.4.6/tmp 目录下创建一个 myid 的空文件  
echo 1 > /root/training/zookeeper-3.4.6/tmp/myid

(\*) 将配置好的 zookeeper 拷贝到其他节点，同时修改各自的 myid 文件  
scp -r /root/training/zookeeper-3.4.6/ hadoop113:/root/training  
scp -r /root/training/zookeeper-3.4.6/ hadoop114:/root/training

## 2、启动 ZooKeeper 和查看 ZooKeeper 的状态

```
zkServer.sh start
```

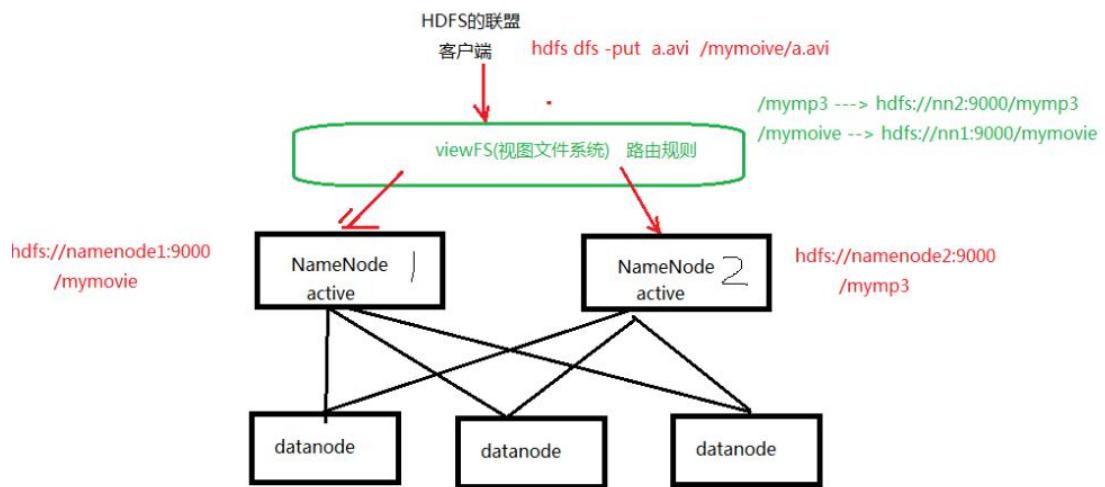
```
zkServer.sh status
```

## （五）操作 Zookeeper

- 通过 ZooKeeper 的命令行
  - zkCli.sh
  - 创建一个节点（数据）：create /mydata helloworld
- Demo：ZooKeeper 集群间信息的同步。

## 十五、Hadoop 的集群和 HA

### (一) HDFS 的联盟



#### ● 配置 HDFS 的联盟

HDFS 的 Federation（联盟）：即 集群中存在多个 namenode，每个数据节点需要向不同的 Namenode 注册

| 参数文件            | 配置参数                                   | 参考值                                |
|-----------------|----------------------------------------|------------------------------------|
| hadoop-env.sh   | JAVA_HOME                              | /root/training/jdk1.8.0_144        |
| core-site.xml   | hadoop.tmp.dir                         | /root/training/hadoop-2.7.3/tmp    |
| hdfs-site.xml   | dfs.nameservices                       | ns1,ns2                            |
|                 | dfs.namenode.rpc-address.ns1           | 192.168.157.112:9000               |
|                 | dfs.namenode.http-address.ns1          | 192.168.157.112:50070              |
|                 | dfs.namenode.secondaryhttp-address.ns1 | 192.168.157.112:50090              |
|                 | dfs.namenode.rpc-address.ns2           | 192.168.157.113:9000               |
|                 | dfs.namenode.http-address.ns2          | 192.168.157.113:50070              |
|                 | dfs.namenode.secondaryhttp-address.ns2 | 192.168.157.113:50090              |
|                 | dfs.replication                        | 2                                  |
|                 | dfs.permissions                        | false                              |
| mapred-site.xml | mapreduce.framework.name               | yarn                               |
| yarn-site.xml   | yarn.resourcemanager.hostname          | 192.168.157.112                    |
|                 | yarn.nodemanager.aux-services          | mapreduce_shuffle                  |
| slaves          | DataNode 的地址                           | 192.168.157.115<br>192.168.157.114 |

在每个 NameNode 上格式化： `hdfs namenode -format -clusterId xdl1`

## ● 配置 viewFS

修改 `core-site.xml` 文件，直接加入以下内容：

```
<property>
    <name>fs.viewfs.mounttable.xdl1.homedir</name>
    <value>/home</value>
</property>

<property>
    <name>fs.viewfs.mounttable.xdl1.link./movie</name>
    <value>hdfs://192.168.157.112:9000/movie</value>
</property>

<property>
    <name>fs.viewfs.mounttable.xdl1.link./mp3</name>
    <value>hdfs://192.168.157.113:9000/mp3</value>
</property>

<property>
    <name>fs.default.name</name>
    <value>viewfs://xdl1</value>
</property>
```

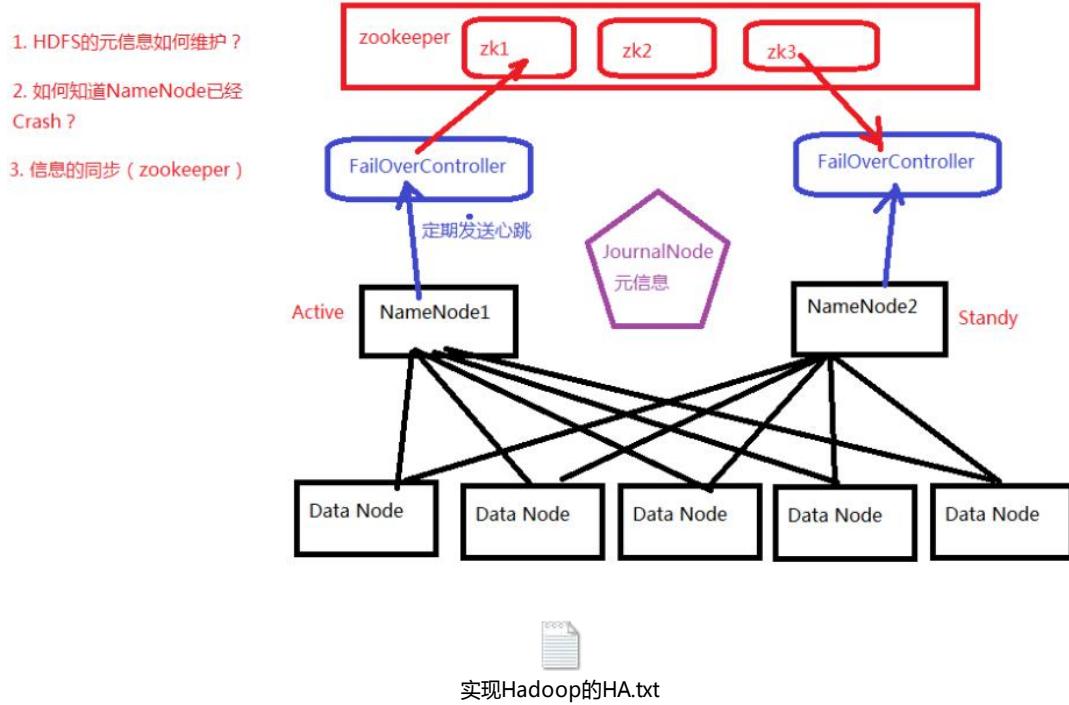
三、创建对应的目录：

```
hadoop fs -mkdir hdfs://192.168.157.112:9000/movie
hadoop fs -mkdir hdfs://192.168.157.113:9000/mp3
```

四、就可以使用了

```
hdfs dfs -put a.txt /movie/a.txt
hdfs dfs -cat /movie/a.txt
```

## (二) 利用 ZooKeeper 实现 Hadoop 的 HA



注意：

hadoop113 上的 ResourceManager 需要单独启动

命令： yarn-daemon.sh start resourcemanager

## 十六、Storm

### (一) 什么是 Storm?

Storm 为分布式实时计算提供了一组通用原语，可被用于“流处理”之中，实时处理消息并更新数据库。这是管理队列及工作者集群的另一种方式。Storm 也可被用于“连续计算”（continuous computation），对数据流做连续查询，在计算时就将结果以流的形式输出给用户。它还可被用于“分布式 RPC”，以并行的方式运行昂贵的运算。

Storm 可以方便地在一个计算机集群中编写与扩展复杂的实时计算，Storm 用于实时处理，就好比 Hadoop 用于批处理。Storm 保证每个消息都会得到处理，而且它很快——在一个小集群中，每秒可以处理数以百万计的消息。更棒的是你可以使用任意编程语言来做开发。

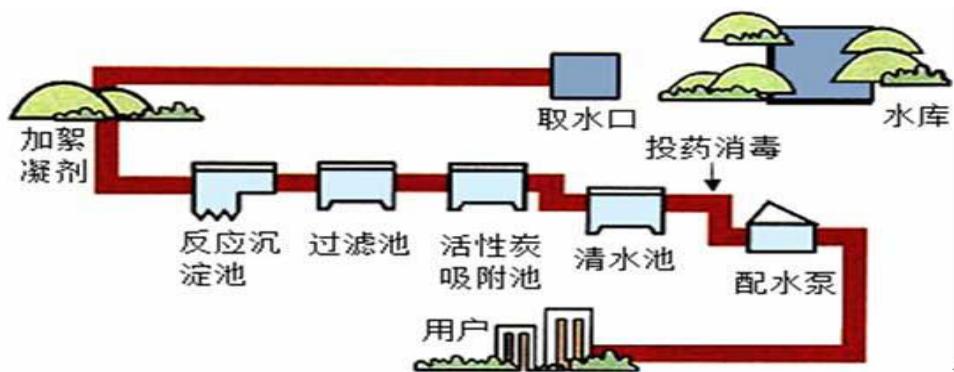
### (二) 离线计算和流式计算

#### ① 离线计算

- 离线计算：批量获取数据、批量传输数据、周期性批量计算数据、数据展示
- 代表技术：Sqoop 批量导入数据、HDFS 批量存储数据、MapReduce 批量计算、Hive

#### ② 流式计算

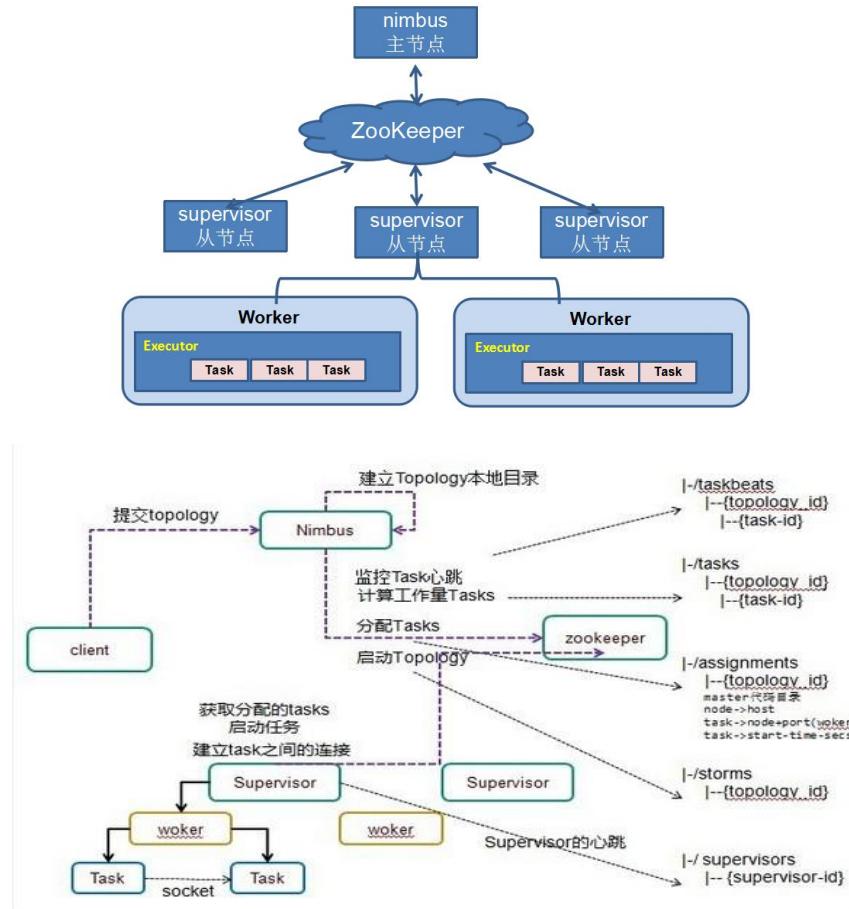
- 流式计算：数据实时产生、数据实时传输、数据实时计算、实时展示
- 代表技术：Flume 实时获取数据、Kafka/metaq 实时数据存储、Storm/JStorm 实时数据计算、Redis 实时结果缓存、持久化存储(mysql)。
- 一句话总结：将源源不断产生的数据实时收集并实时计算，尽可能快的得到计算结果



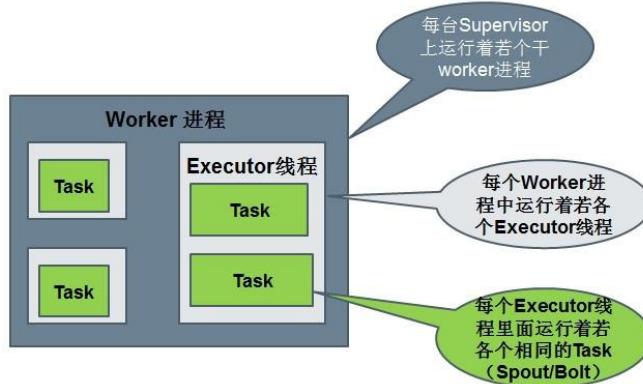
### ③ Storm 与 Hadoop 的区别

Storm 用于实时计算	Hadoop 用于离线计算
Storm 处理的数据保存在内存中，源源不断	Hadoop 处理的数据保存在文件系统中，一批一批
Storm 的数据通过网络传输进来	Hadoop 的数据保存在磁盘中
Storm 与 Hadoop 的编程模型相似	

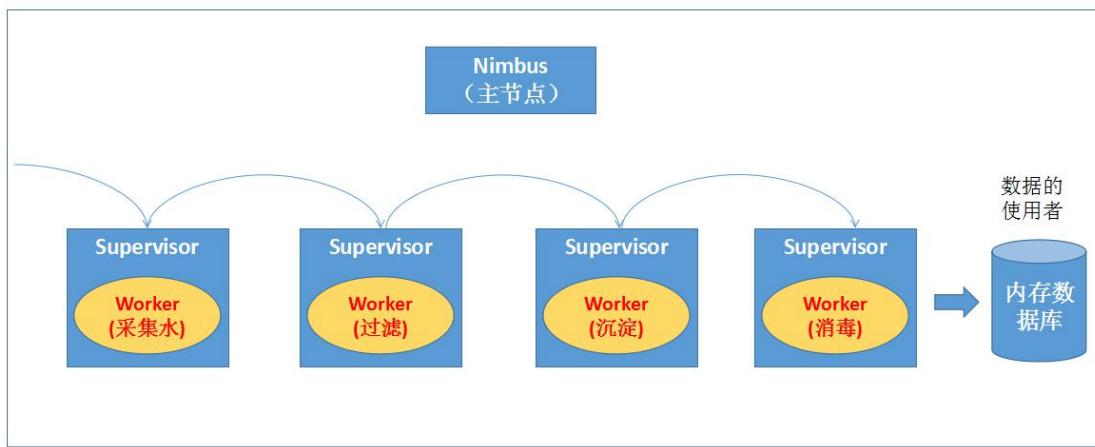
## (三) Storm 的体系结构



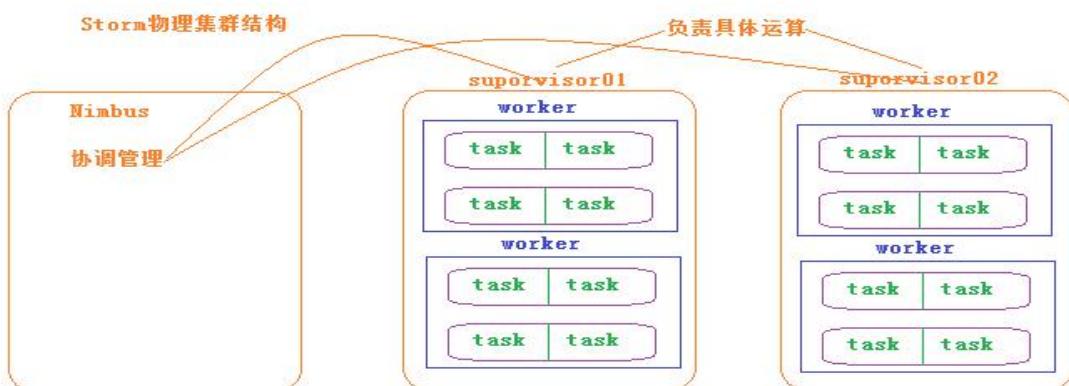
- **Nimbus**: 负责资源分配和任务调度。
- **Supervisor**: 负责接受 nimbus 分配的任务，启动和停止属于自己管理的 worker 进程。通过配置文件设置当前 supervisor 上启动多少个 worker。
- **Worker**: 运行具体处理组件逻辑的进程。Worker 运行的任务类型只有两种，一种是 Spout 任务，一种是 Bolt 任务。
- **Executor**: Storm 0.8 之后，Executor 为 Worker 进程中的具体的物理线程，同一个 Spout/Bolt 的 Task 可能会共享一个物理线程，一个 Executor 中只能运行隶属于同一个 Spout/Bolt 的 Task。
- **Task**: worker 中每一个 spout/bolt 的线程称为一个 task. 在 storm0.8 之后，task 不再与物理线程对应，不同 spout/bolt 的 task 可能会共享一个物理线程，该线程称为 executor。



## (四) Storm 的运行机制



- 整个处理流程的组织协调不用用户去关心，用户只需要去定义每一个步骤中的具体业务处理逻辑
- 具体执行任务的角色是 Worker，Worker 执行任务时具体的行为则有我们定义的业务逻辑决定



## (五) Storm 的安装配置

- 解压: tar -zxvf apache-storm-1.0.3.tar.gz -C ~/training/
- 设置环境变量

```
STORM_HOME=/root/training/apache-storm-1.0.3
export STORM_HOME

PATH=$STORM_HOME/bin:$PATH
export PATH
```

- 编辑配置文件: \$STORM\_HOME/conf/storm.yaml

```
#####
# These MUST be filled in for a storm configuration
#####
storm.zookeeper.servers:
  - "192.168.137.81"
  - "192.168.137.82"
  - "192.168.137.83"

nimbus.seeds: ["192.168.137.81"]

storm.local.dir: "/root/training/apache-storm-1.0.3/tmp"

supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703
```

- 注意: 如果要搭建 Storm 的 HA, 只需要在 nimbus.seeds 中设置多个 nimbus 即可。
- 把安装包复制到其他节点上。

## (六) 启动和查看 Storm

- 在 nimbus.host 所属的机器上启动 nimbus 服务和 logviewer 服务
  - storm nimbus &
  - storm logviewer &
- 在 nimbus.host 所属的机器上启动 ui 服务
  - storm ui &
- 在其它个点击上启动 supervisor 服务和 logviewer 服务
  - storm supervisor &
  - storm logviewer &
- 查看 storm 集群: 访问 nimbus.host:/8080, 即可看到 storm 的 ui 界面

### Storm UI

**Cluster Summary**

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors
0.9.5	5d 2h 12m 59s	4	3	13	16	18

**Topology summary**

Name	Id	Status	Uptime	Num workers	Num executors
ExclamationTopology	ExclamationTopology-1-1445615253	ACTIVE	5d 1h 55m 4s	3	18

**Supervisor summary**

Id	Host	Uptime	Slots	Used slot
6d1a8131-4eee-4515-8cbc-2ada3e70da9a	storm03	5d 2h 11m 8s	4	0
55c3d1b1-151a-4d26-8cb9-ba5ead463952	storm04	5d 2h 11m 3s	4	1
414ecb3e-3e36-4d4f-8454-49f3d1d7db17	storm02	5d 2h 11m 48s	4	1
9607a45c-43d4-4a8a-b2f3-c9de81698962	storm05	5d 2h 11m 0s	4	1

**Nimbus Configuration**

## (七) Storm 的常用命令

有许多简单且有用的命令可以用来管理拓扑，它们可以提交、杀死、禁用、再平衡拓扑。

- ① 提交任务命令格式: **storm jar** 【jar 路径】 【拓扑包名.拓扑类名】 【拓扑名称】

```
storm jar storm-starter-topologies-1.0.3.jar org.apache.storm.starter.WordCountTopology MyWordCount1
```

- ② 杀死任务命令格式: **storm kill** 【拓扑名称】 -w 10

(执行 kill 命令时可以通过-w [等待秒数]指定拓扑停用以后的等待时间)

```
storm kill topology-name -w 10
```

- ③ 停用任务命令格式: **storm deactivate** 【拓扑名称】

```
storm deactivate topology-name
```

- ④ 启用任务命令格式: **storm activate** 【拓扑名称】

```
storm activate topology-name
```

- ⑤ 重新部署任务命令格式: **storm rebalance** 【拓扑名称】

```
storm rebalance topology-name
```

再平衡使你重分配集群任务。这是个很强大的命令。比如，你向一个运行中的集群增加了节点。再平衡命令将会停用拓扑，然后在相应超时时间之后重分配工人，并重启拓扑。

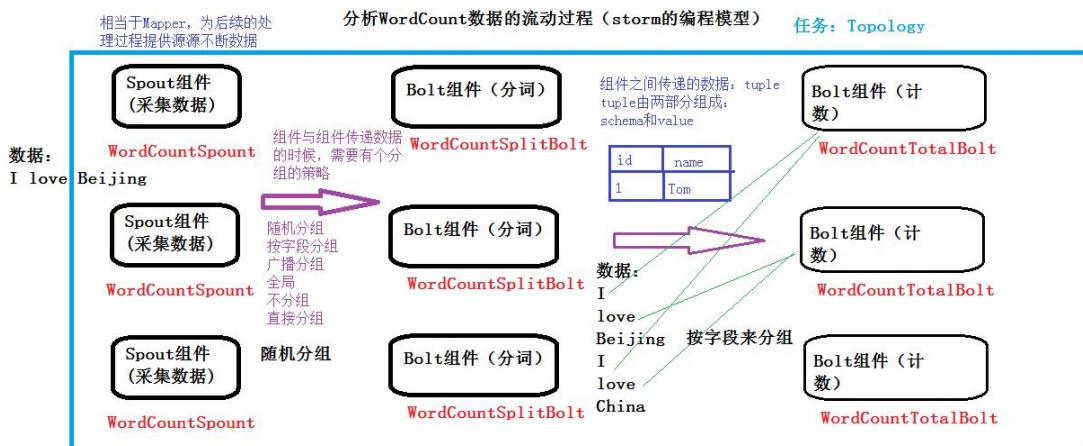
## (八) Demo 演示：WordCount 及流程分析

通过查看 Storm UI 上每个组件的 events 链接，可以查看 Storm 的每个组件（spout、bolt）发送的消息。但 Storm 的 event logger 的功能默认是禁用的，需要在配置文件中设置：topology.eventlogger.executors: 1，具体说明如下：

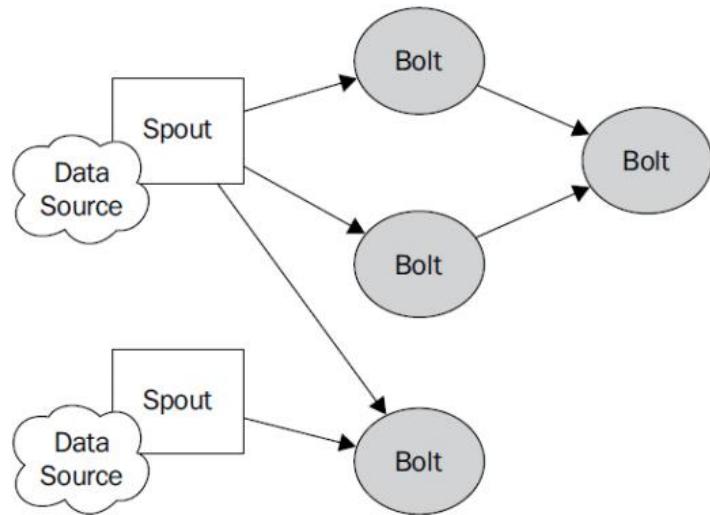
- "topology.eventlogger.executors": 0 默认，禁用
- "topology.eventlogger.executors": 1 一个 topology 分配一个 Event Logger.
- "topology.eventlogger.executors": nil 每个 worker.分配一个 Event Logger

The screenshot shows the Storm UI interface. At the top, there is a toolbar with buttons: Activate, Deactivate, Rebalance, Kill, Debug (which is highlighted with a red box), Stop Debug, and Change Log Level. Below the toolbar is a section titled 'Topology actions'. Underneath this, there is a table titled 'Component summary' with columns: Id, Topology, Executors, Tasks, and Debug. A row for 'spout' is shown with values: MyWordCount1, 5, 5, and 'events' (which is also highlighted with a red box). The entire screenshot area is enclosed in a light gray border.

### WordCount 的数据流程分析



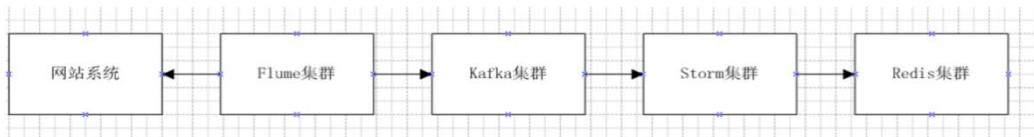
## (九) Storm 的编程模型



- **Topology:** Storm 中运行的一个实时应用程序的名称。（拓扑）
- **Spout:** 在一个 topology 中获取源数据流的组件。
  - 通常情况下 spout 会从外部数据源中读取数据，然后转换为 topology 内部的源数据。
- **Bolt:** 接受数据然后执行处理的组件，用户可以在其中执行自己想要的操作。
- **Tuple:** 一次消息传递的基本单元，理解为一组消息就是一个 Tuple。
- **Stream:** 表示数据的流向。
- **StreamGroup:** 数据分组策略
  - Shuffle Grouping : 随机分组，尽量均匀分布到下游 Bolt 中
  - Fields Grouping : 按字段分组，按数据中 field 值进行分组；相同 field 值的 Tuple 被发送到相同的 Task
  - All grouping : 广播
  - Global grouping : 全局分组，Tuple 被分配到一个 Bolt 中的一个 Task，实现事务性的 Topology。
  - None grouping : 不分组
  - Direct grouping : 直接分组 指定分组

## (十) Storm 编程案例：WordCount

流式计算一般架构图：



- Flume 用来获取数据。
- Kafka 用来临时保存数据。
- Strom 用来计算数据。
- Redis 是个内存数据库，用来保存数据。

## ■ 创建 Spout (**WordCountSpout**) 组件采集数据，作为整个 Topology 的数据源

```
public class WordCountSpout extends BaseRichSpout{
    //模拟数据
    private String[] data = {"I love Beijing", "I love Shanghai", "Beijing is the capital of China"};
    //用于往下一个组件发送消息
    private SpoutOutputCollector collector;

    @Override
    public void nextTuple() {
        Utils.sleep(3000);
        //有Storm框架调用，用于接收外部数据源的数据
        int random = (new Random()).nextInt(3);
        String sentence = data[random];

        //发送数据
        System.out.println("发送数据：" + sentence);
        this.collector.emit(new Values(sentence));
    }

    @Override
    public void open(Map arg0, TopologyContext arg1, SpoutOutputCollector collector) {
        //Spout初始化方法
        this.collector = collector;
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
}
```

## ■ 创建 Bolt (**WordCountSplitBolt**) 组件进行分词操作

```
public class WordCountSplitBolt extends BaseRichBolt{
    private OutputCollector collector;

    @Override
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        //分词
        String[] words = sentence.split(" ");
        for(String word:words){
            this.collector.emit(new Values(word,1));
        }
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

## ■ 创建 Bolt (**WordCountBoltCount**) 组件进行单词计数

```
public class WordCountBoltCount extends BaseRichBolt{
    private Map<String, Integer> result = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        int count = tuple.getIntegerByField("count");

        if(result.containsKey(word)){
            int total = result.get(word);
            result.put(word, total+count);
        }else{
            result.put(word, 1);
        }
        //直接输出到屏幕
        System.out.println("输出的结果是：" + result);
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector collector) {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

## ■ 创建主程序 Topology (WordCountTopology)，并提交到本地运行

```
public static void main(String[] args) {
    TopologyBuilder builder = new TopologyBuilder();

    //设置任务的spout组件
    builder.setSpout("wordcount_spout", new WordCountSpout());

    //设置任务的第一个Bolt组件
    builder.setBolt("wordcount_splitbolt", new WordCountSplitBolt())
        .shuffleGrouping("wordcount_spout");

    //设置任务的第二个Bolt组件
    builder.setBolt("wordcount_count", new WordCountBoltCount())
        .fieldsGrouping("wordcount_splitbolt", new Fields("word"));
    //创建Topology任务
    StormTopology wc = builder.createTopology();

    Config config = new Config();

    //提交任务到本地运行
    LocalCluster localCluster = new LocalCluster();
    localCluster.submitTopology("mywordcount", config, wc);
}
```

## ■ 也可以将主程序 Topology (WordCountTopology) 提交到 Storm 集群运行

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();

    //设置任务的spout组件
    builder.setSpout("wordcount_spout", new WordCountSpout());

    //设置任务的第一个Bolt组件
    builder.setBolt("wordcount_splitbolt", new WordCountSplitBolt())
        .shuffleGrouping("wordcount_spout");

    //设置任务的第二个Bolt组件
    builder.setBolt("wordcount_count", new WordCountBoltCount())
        .fieldsGrouping("wordcount_splitbolt", new Fields("word"));
    //创建Topology任务
    StormTopology wc = builder.createTopology();

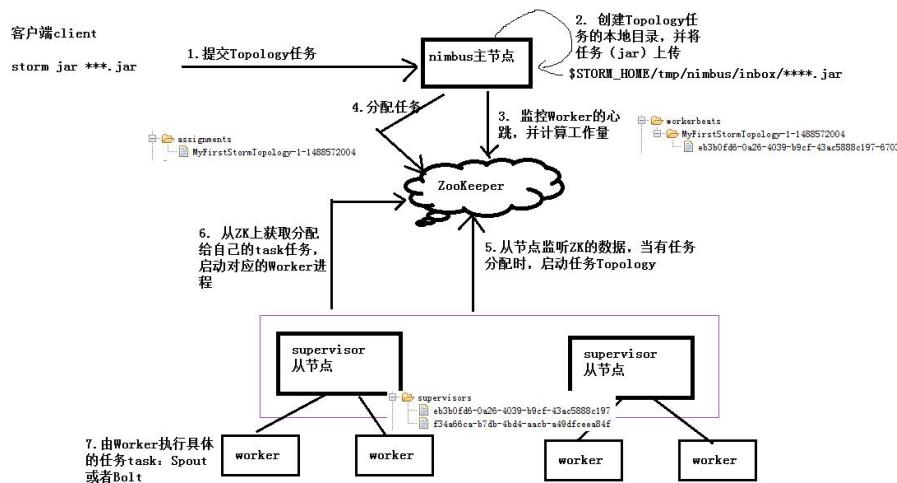
    Config config = new Config();

    //提交任务到Storm集群运行
    StormSubmitter.submitTopology(args[0], config, wc);
}
```

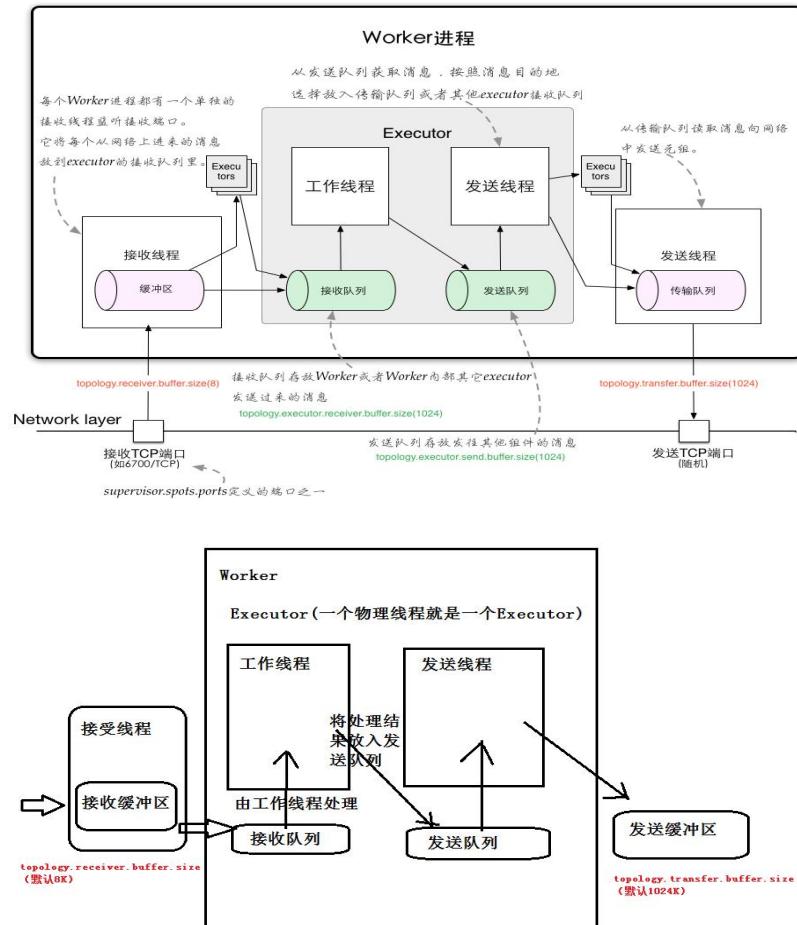
## (十一) Storm 集群在 ZK 上保存的数据结构



## (十二) Storm 集群任务提交流程



## (十三) Storm 内部通信机制



## (十四) 集成 Storm

### (1) 与 JDBC 集成

- 将 Storm Bolt 处理的结果插入 MySQL 数据库中
- 需要依赖的 jar 包
  - \$STORM\_HOME\external\sql\storm-sql-core\\*.jar
  - \$STORM\_HOME\external\storm-jdbc\storm-jdbc-1.0.3.jar
  - mysql 的驱动
  - commons-lang3-3.1.jar



- 与 JDBC 集成的代码实现

- 修改主程序 WordCountTopology，增加如下代码：

```
//创建一个JDBC Bolt将结果插入数据库中
builder.setBolt("wordcount_jdbcBolt", createJDBC Bolt()).
    shuffleGrouping("wordcount_count");
```

增加一个新方法创建 JDBC Bolt 组件

```
//创建JDBC Insert Bolt组件
//需要事先在MySQL数据库中创建对应的表: result
private static IRichBolt createJDBC Bolt() {
    ConnectionProvider connectionProvider = new MyConnectionProvider();
    JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper("aaa", connectionProvider);

    return new JdbcInsertBolt(connectionProvider, simpleJdbcMapper)
        .withTableName("result").withQueryTimeoutSecs(30);
}
```

- 实现 ConnectionProvider 接口

```
class MyConnectionProvider implements ConnectionProvider{
    private static String driver = "com.mysql.jdbc.Driver";
    private static String url = "jdbc:mysql://192.168.137.81:3306/demo";
    private static String user = "collen";
    private static String password = "password";

    //静态块
    static{//注册驱动
        try {
            Class.forName(driver);
        } catch (ClassNotFoundException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    @Override
    public Connection getConnection() {
        try {
            return DriverManager.getConnection(url, user, password);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    public void cleanup() {}
    public void prepare() {}
}
```

- 修改 WordCountBoltCount 组件，将统计后的结果发送给下一个组件写入 MySQL

```
public class WordCountBoltCount extends BaseRichBolt{
    private Map<String, Integer> result = new HashMap<String, Integer>();
    private OutputCollector collector;
    @Override
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        int count = tuple.getIntegerByField("count");

        if(result.containsKey(word)){
            int total = result.get(word);
            result.put(word, total+count);
        }else{
            result.put(word, 1);
        }
        //直接输出到屏幕
        //System.out.println("输出的结果是：" + result);
        //将统计结果发送给下一个Bolt，插入数据库
        this.collector.emit(new Values(word, result.get(word)));
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "total"));
    }
}
```

## (2) 与 Redis 集成

Redis 是一个 key-value 存储系统。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash (哈希类型)。与 Memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。

Redis 是一个高性能的 key-value 数据库。Redis 的出现，很大程度补偿了 memcached 这类 key/value 存储的不足，在部分场合可以对关系数据库起到很好的补充作用。它提供了 Java, C/C++, C#, PHP, JavaScript, Perl, Object-C, Python, Ruby, Erlang 等客户端，使用很方便。[\[1\]](#)

Redis 支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。

- 修改代码：**WordCountTopology.java**

```
builder.setBolt("wordcount_redisBolt", createRedisBolt()
    .shuffleGrouping("wordcount_count"));

//创建Redis Bolt组件
private static IRichBolt createRedisBolt() {
    JedisPoolConfig.Builder builder = new JedisPoolConfig.Builder();
    builder.setHost("192.168.137.81");
    builder.setPort(6379);
    JedisPoolConfig poolConfig = builder.build();

    //RedisStoreMapper用于指定存入Redis中的数据格式
    return new RedisStoreBolt(poolConfig, new RedisStoreMapper() {
        @Override
        public RedisDataTypeDescription getDescription() {
            return new RedisDataTypeDescription(RedisDataTypeDescription.RedisDataType.HASH, "wordCount");
        }

        @Override
        public String getValueFromTuple(ITuple tuple) {
            return String.valueOf(tuple.getIntegerByField("total"));
        }
        @Override
        public String getKeyFromTuple(ITuple tuple) {
            return tuple.getStringByField("word");
        }
    });
}
```

### (3) 与 HDFS 集成

- ◆ 需要的 jar 包：
  - \$STORM\_HOME\external\storm-hdfs\storm-hdfs-1.0.3.jar
  - HDFS 相关的 jar 包

- ◆ 开发新的 bolt 组件

```
//创建一个新的HDFS Bolt组件，把前一个bolt组件处理的结果存入HDFS
private static IRichBolt createHDFSBolt() {
    HdfsBolt bolt = new HdfsBolt();
    //HDFS的位置
    bolt.withFsUrl("hdfs://192.168.137.111:9000");

    //数据保存在HDFS上的目录
    bolt.withFileNameFormat(new DefaultFileNameFormat().withPath("/stormdata"));

    //写入HDFS的数据的分隔符 | 结果: Beijing|10
    bolt.withRecordFormat(new DelimitedRecordFormat().withFieldDelimiter("|"));

    //每5M的数据生成一个文件
    bolt.withRotationPolicy(new FileSizeRotationPolicy(5.0f, Units.MB));

    //Bolt输出tuple，当tuple达到一定的大小（每1K），与HDFS进行同步
    bolt.withSyncPolicy(new CountSyncPolicy(1000));

    return bolt;
}
```

### (4) 与 HBase 集成

- ◆ 需要的 jar 包：HBase 的相关包
- ◆ 开发新的 bolt 组件（WordCountBoltHBase.java）

```
/*
 * 在HBase中创建表，保存数据
 * create 'result','info'
 */
public class WordCountBoltHBase extends BaseRichBolt {

    public void execute(Tuple tuple) {
        // 如何处理？将上一个bolt组件发送过来的结果，存入HBase
        //取出上一个组件发送过来的数据
        String word = tuple.getStringByField("word");
        int total = tuple.getIntegerByField("total");

        //构造一个Put对象
        Put put = new Put(Bytes.toBytes(word));
        put.add(Bytes.toBytes("info"), Bytes.toBytes("word"), Bytes.toBytes(word));
        put.add(Bytes.toBytes("info"), Bytes.toBytes("total"), Bytes.toBytes(String.valueOf(total)));

        //把数据插入HBase
        try {
            table.put(put);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
    // 初始化
    //指定ZK的地址
    Configuration conf = new Configuration();
    conf.set("hbase.zookeeper.quorum", "192.168.137.111");

    //创建table的客户端
    try{
        table = new HTable(conf,"result");
    }catch(Exception ex){
        ex.printStackTrace();
    }
}

```

## (5) 与 Apache Kafka 集成

- 注意：需要把 **slf4j-log4j12-1.6.4.jar** 包去掉，有冲突（有两个）

```

private static IRichSpout createKafkaSpout() {
    //定义ZK的地址
    BrokerHosts hosts = new ZkHosts("hadoop81:2181,hadoop82:2181,hadoop83:2181");

    //指定Topic的信息
    SpoutConfig spoutConf = new SpoutConfig(hosts, "mydemo2", "/mydemo2", UUID.randomUUID().toString());

    //定义收到消息的Schema格式
    spoutConf.scheme = new SchemeAsMultiScheme(new Scheme() {
        @Override
        public Fields getOutputFields() {
            return new Fields("sentence");
        }
        @Override
        public List<Object> deserialize(ByteBuffer buffer) {
            try {
                String msg = (Charset.forName("UTF-8").newDecoder()
                    .decode(buffer)
                    .asReadOnlyBuffer()
                    .toString());
                System.out.println("*****收到的数据是msg: "+ msg);
                return new Values(msg);
            } catch (Exception e) {
                e.printStackTrace();
            }
            return null;
        }
    });
    return new KafkaSpout(spoutConf);
}

```

## (6) 与 Hive 集成

- ◆ 由于集成 Storm 和 Hive 依赖的 jar 较多，并且冲突的 jar 包很多，强烈建议使用 Maven 来搭建新的工程。

```
<dependencies>
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-core</artifactId>
        <version>1.0.3</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-hive</artifactId>
        <version>1.0.3</version>
        <type>jar</type>
    </dependency>
</dependencies>
```

- ◆ 需要对 Hive 做一定的配置（在 `hive-site.xml` 文件中）：

```
<property>
    <name>hive.in.test</name>
    <value>true</value>
</property>
```

- ◆ 需要使用下面的语句在 hive 中创建表：

```
create table wordcount
(word string, total int)
clustered by (word) into 10 buckets
stored as orc TBLPROPERTIES('transactional'='true');
```

- ◆ 启动 metastore 服务： `hive --service metastore`

- ◆ 开发新的 bolt 组件，用于将前一个 bolt 处理的结果写入 Hive

```
private static IRichBolt createHiveBolt() {
    //设置环境变量，能找到winutils.exe
    System.setProperty("hadoop.home.dir", "D:\\tools\\hadoop-2.4.1\\hadoop-2.4.1");

    //作用：将bolt组件处理后的结果tuple，如何存入hive的表
    DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
        .withColumnFields(new Fields("word", "total"));

    //配置Hive的参数信息
    HiveOptions options = new HiveOptions("thrift://hadoop111:9083", //hive的metastore
        "default", //hive数据库的名字
        "wordcount", //保存数据的表
        mapper)
        .withTxnsPerBatch(10)
        .withBatchSize(1000)
        .withIdleTimeout(10);

    // 创建一个Hive的bolt组件，将单词计数后的结果存入hive
    HiveBolt bolt = new HiveBolt(options);
    return bolt;
}
```

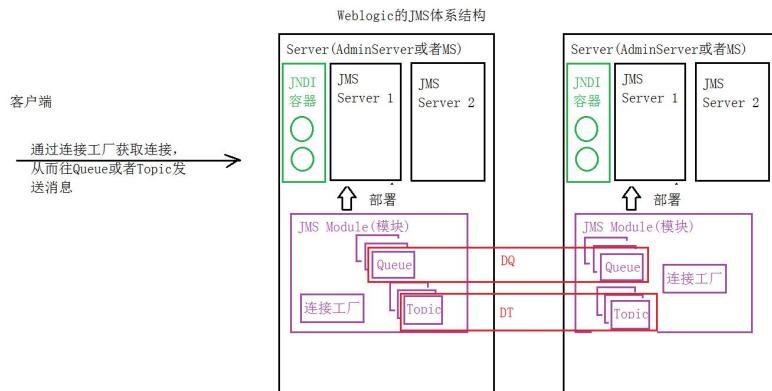
- ◆ 为了测试的方便，我们依然采用之前随机产生字符串的 Spout 组件产生数据

## (7) 与 JMS 集成

JMS 即 Java 消息服务（Java Message Service）应用程序接口，是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java 消息服务是一个与具体平台无关的 API，绝大多数 MOM 提供商都对 JMS 提供支持。

JMS 的两种消息类型：**Queue** 和 **Topic**

基于 Weblogic 的 JMS 架构：



```

private static IRichBolt createJMSBolt() {
    //创建一个JMS Bolt, 将前一个bolt发送过来的数据 写入JMS
    JmsBolt bolt = new JmsBolt();

    //指定JMSBolt的Provider
    bolt.setJmsProvider(new MyJMSPublisher());

    //指定bolt如何解析消息
    bolt.setJmsMessageProducer(new JmsMessageProducer() {

        @Override
        public Message toMessage(Session session, ITuple tuple) throws JMSEException {
            //取出上一个组件发送过来的数据
            String word = tuple.getStringByField("word");
            int total = tuple.getIntegerByField("total");
            return session.createTextMessage(word+" "+total);
        }
    });

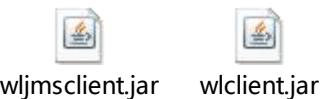
    return bolt;
}

```



WordCountTopology.java

- 需要的 weblogic 的 jar 包



- permission javax.management.MBeanTrustPermission "register";