

# 作者简介

## 王健

毕业于哈尔滨工业大学。先后就职于浪潮集团、北京传智播客、Oracle(中国)北京教研中心，又先后在山东交通学院、齐鲁工业大学讲授 Java、Android、Hadoop 等相关课程。从事软件开发和教育事业 15 年。自由撰稿人，在 [www.open-open.com](http://www.open-open.com) 发表文章 100 余篇。技术爱好者，一直从事技术领域的一线工作，至力于新技术的研究与推广。



# 内容简介

本书是关于目前最新 Hadoop 快速上手的教程。Hadoop 自出现以来，就被广泛关注。特别是近几年数据增长量急剧膨胀，加上历史数据。如何处理这些海量数据成为炙手可热的问题。Hadoop 的出现，无疑是良药奇方。Hadoop 最早从 0.x 版本到 1.x 版本，再到 2.x 版本，发生了很大的变化。伴随 hadoop 的成长，hadoop 已经不再是一个简单的数据分布式存储平台和工具，已经成长为一个完整的 hadoop 生态圈。相信读者会在后面不断的学习过程中，体会到这一点。

本书，力从实践操作开发讲起，去除那些纯理论的知识。等基本的操作已经可以掌握以后，再来讲解理论知识。所以，本书是先实践再理论的过程。

**未经许可，不得以任何形式复制、抄袭本书内容。**

**内部教材，版权所有，侵权必究。**

同步视频资源地址：<http://pan.baidu.com/s/1kUQ6qnt> 密码：vjo3

编著：王健

封面设计：张剑

责任校对：王健

邮箱：wangjian\_me@126.com

开本：185\*260 1/16          字数：15 万

版次：2017 年 8 月 第 1 版

定价：45 元

# 目录

第 1 章 Hadoop 简介.....	1
主要内容.....	1
1.1、Hadoop2.x 新特性.....	1
1.2、虚拟机.....	2
1.3、安装 Linux 操作系统.....	3
1.4、SSH 工具与使用.....	8
1.5、小结.....	8
第 2 章 Hadoop 伪分布式.....	9
主要内容.....	9
2.1、安装 Java 环境.....	9
2.2、安装独立运行的 hadoop.....	10
2.3、Hadoop 伪分布式安装与配置.....	13
2.4、hdfs 命令简介.....	21
2.5、Java 代码操作 hdfs.....	23
2.6、小结.....	25
第 3 章 HDFS 文件系统.....	26
主要内容.....	26
3.1、HDFS 的体系结构.....	27
3.2、NameNode 的工作.....	27
3.3、Secondary NameNode.....	29
3.4、DataNode.....	30
3.5、HDFS 的命令.....	31
3.6、RPC 远程过程调用.....	32
3.7、小结.....	35
第 4 章 MapReduce.....	37
主要内容.....	37
4.1、MapReduce 的运算过程.....	37
4.2、WordCount 示例.....	39
4.3、序列化的概念 Writable 接口.....	44
4.4、Partitioner 编程.....	46
4.5、自定义排序.....	53
4.6、Combiner 编程.....	56
4.7、Shuffle.....	58
4.8、小结.....	63
第 5 章 自定义 InputFormat 类.....	65
内容简介.....	65

5.1、自定义文件输入流.....	65
5.2、在 Excel 中解析通话记录统计.....	69
5.3、小结.....	79
第 6 章 Hadoop 集群配置.....	81
内容简介.....	81
6.1、配置 hadoop 集群.....	81
6.2、小结.....	89
第 7 章 Zookeeper 分布式协调技术.....	92
内容简介.....	92
7.1、zookeeper 简介.....	92
7.2、单一节点安装 zookeeper.....	96
7.3、zookeeper 集群安装.....	98
7.4、配置 hadoop 高可靠集群.....	99
7.5、用 Java 代码操作集群.....	114
7.6、小结.....	116
第 8 章 sqoop.....	117
内容简介.....	117
8.1、安装 sqoop.....	117
8.2、sqoop 基本命令.....	119
8.3、导入导出命令.....	120
8.4、小结.....	123
第 9 章 HBase.....	124
内容简介.....	124
9.1、HBase 的特点.....	124
9.2、HBase 单节点安装.....	128
9.3、HBase Shell 的基本操作.....	130
9.4、HBase 伪分布式安装.....	138
9.5、HBase JavaAPI 接口.....	141
9.6、HBase 集群.....	149
9.7、导入数据到 Hbase.....	155
9.8、小结.....	157
第 10 章 Hive.....	158
内容简介.....	158
10.1、Hive1.x 的安装与使用.....	160
10.2、Hive 命令.....	162
10.3、使用 MySQL 数据库存储 metastore.....	167
10.4、Hive 外部表.....	170
10.5、Hive 表分区.....	171
10.6、使用 sqoop 将数据导入 hive.....	177

10.7、hive 函数.....	181
10.8、Hive 自定义函数.....	187
10.9、小结.....	191
第 11 章 Flume.....	193
内容简介.....	193
11.1、Flume 的安装与配置.....	195
11.2、部署 Flume agent.....	196
11.3、小结.....	201
附录 1: .....	202
1、 如何在 Mapper 中获取读取的文件名.....	202
2、 获取 NameNode 状态.....	202
3、 检查 ResourceManager 状态.....	202
4、 NameNode 的 ID 和 DataNode 的 id 不相同时导致 DataNode 启动不成功.....	202
5、 NodeManager 自己退出的问题.....	203
6、 DataNode 没有显示的问题.....	204

# 第 1 章 Hadoop 简介

## 主要内容

- Hadoop 简介
- 虚拟机的安装与配置
- Linux 的操作系统安装
- SSH 工具

Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构。

用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。

Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS 放宽了（relax）POSIX 的要求，可以以流的形式访问（streaming access）文件系统中的数据。

Hadoop 的框架最核心的设计就是：HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，则 MapReduce 为海量的数据提供了计算。

## 1.1、Hadoop2.x 新特性

Hadoop1.0 即第一代 Hadoop，由分布式存储系统 HDFS 和分布式计算框架 MapReduce 组成，其中 HDFS 由一个 NameNode 和多个 DataNode 组成，MapReduce 由一个 JobTracker 和多个 TaskTracker 组成。

Hadoop2.0 即第二代 hadoop 为克服 Hadoop1.0 中的不足：针对 Hadoop1.0 单 NameNode 制约 HDFS 的扩展性问题，提出 HDFS Federation，它让多个 NameNode 分管

不同的目录进而实现访问隔离和横向扩展，同时彻底解决了 NameNode 单点故障问题；针对 Hadoop1.0 中的 MapReduce 在扩展性和多框架支持等方面的不足，它将 JobTracker 中的资源管理和作业控制分开，分别由 ResourceManager（负责所有应用程序的资源分配）和 ApplicationMaster（负责管理一个应用程序）实现，即引入了资源管理框架 Yarn。同时 Yarn 作为 Hadoop2.0 中的资源管理系统，它是一个通用的资源管理模块，可为各类应用程序进行资源管理和调度，不仅限于 MapReduce 一种框架，也可以为其其他框架使用，如 Tez、Spark、Storm 等。Hadoop2.x 的体系结构如图 1.1.1 所示。

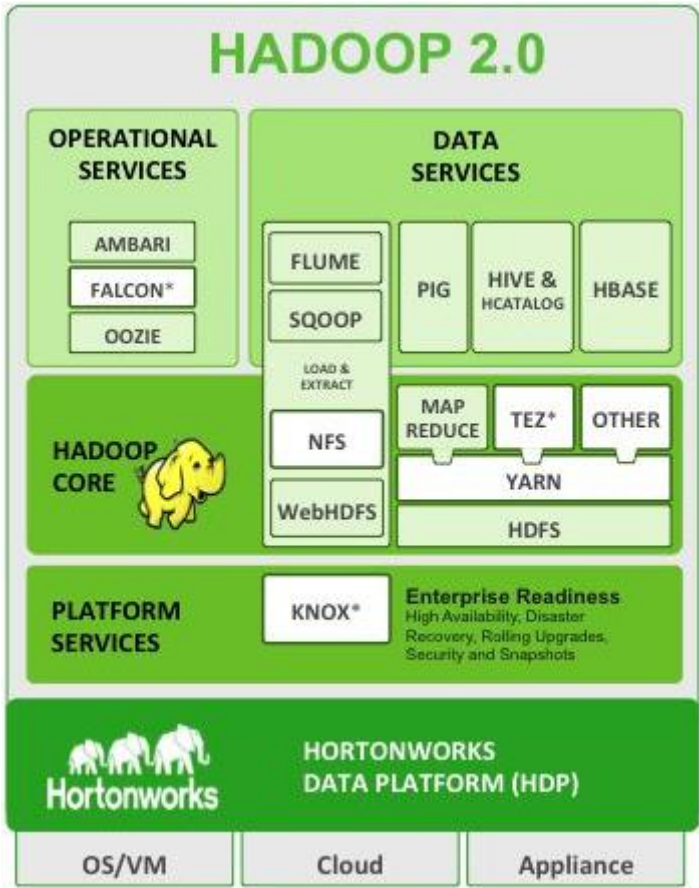


图 1.1.1

## 1.2、虚拟机

本书中，作者将选择使用 Virtual Box 做为虚拟环境安装 Linux 和 Hadoop。

VirtualBox 最早由 SUN 公司开发。由于 SUN 公司目前已经被 Oracle 收购，所以可以在 Oracle 公司的官方网站上下载到 VirtualBox 虚拟机软件的安装程序。下载地址为：<https://www.virtualbox.org>。到笔者发稿时，VirtualBox 的最新版本为 5.1。建议读者在 Window7 或 window10 x64 位的操作系统上安装 VirtualBox 会得到更高的性能。

同时，由于 VirtualBox 需要虚拟化设备的支持，如果在安装操作系统时，不支持将 x64 位的 CentOS 安装到 VirtualBox 里，可以在宿主机开机时进入 BIOS，并打开 CPU 的虚拟化设备。打开 CPU 的虚拟化设备，如图 1.2.1 所示：

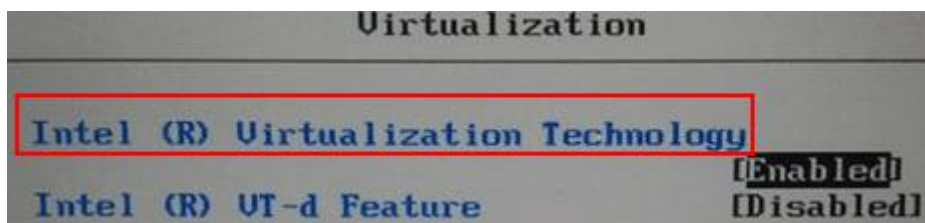


图 1.2.1

### 1.3、安装 Linux 操作系统

本书,将使用CentOS 7做为虚拟化环境来学习和安装Hadoop.首先安装VirtualBox安装程序，如图 1.3.1 所示：

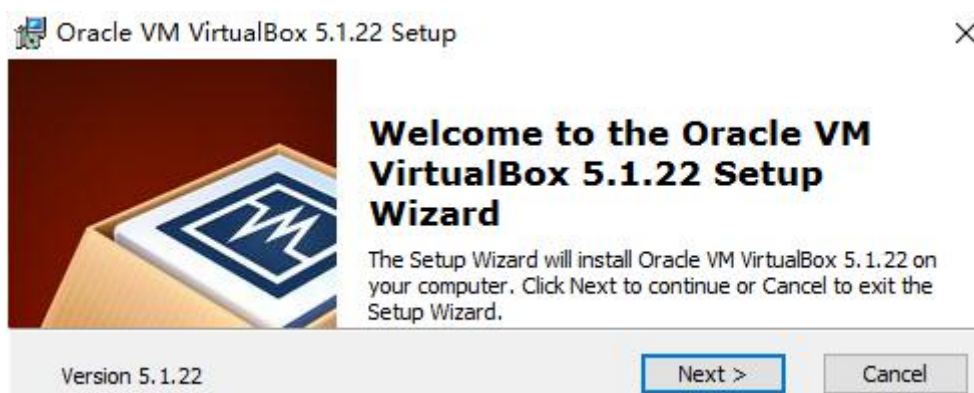


图 1.3.1

在安装完成 VirtualBox 以后，启动 VirtualBox，安装 CentOS 7 的步骤如下：



- 1) 新建，如图 1.3.2 所示



图 1.3.2

- 2) 输入操作系统的名称和选择操作系统的版本。如图 1.3.3

名称(N): CentOS7

类型(T): Linux

版本(V): Red Hat (64-bit)

图 1.3.3

- 3) 为新的系统分配内存，建议 2G 或以上，这样根据您宿主机的内存而定。图 1.3.4。

### 内存大小

选择分配给虚拟电脑的内存大小(MB)。

建议的内存大小为 1024 MB。



图 1.3.4

- 4) 为新的系统创建硬盘，设置为动态增加，建议最大设置为 30G 或以上。同时选择虚拟文件所保存的目录，默认的情况下，会将虚拟化文件保存到 C 盘上。笔者以为最好保存到非系统盘上，如 D:/OS 目录下将是不错的选择。如图 1.3.5。



图 1.3.5

- 5) 选择创建以后,点右键进入设置界面,在存储-盘片的位置选择已经下载的 CentOS7 的 iso 镜像文件。图 1.3.6。

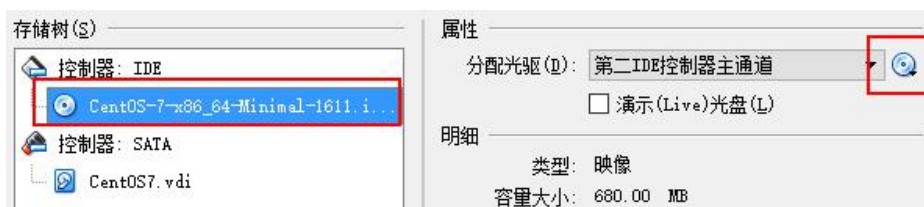


图 1.3.6

- 6) 查看网络设置,将网络 1 设置为 NAT 用于连接外网,将网络 2 设置为 Host Only 用于与宿主机进行通讯。网络连接 1 的设置见图 1.3.7。

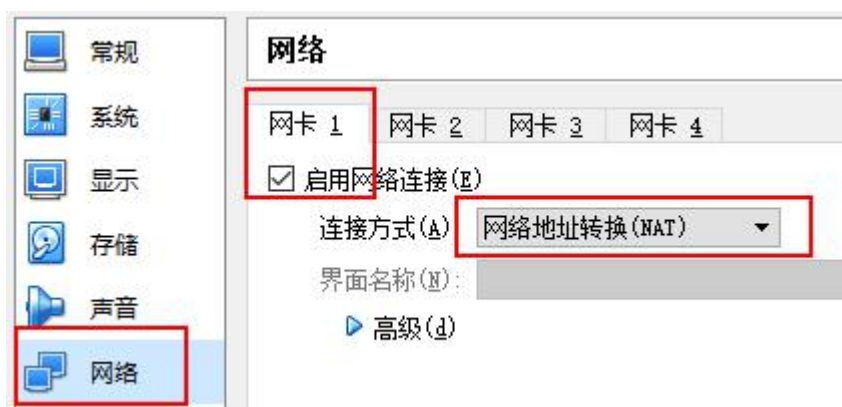


图 1.3.7

网络连接 2 的设置见图 1.3.8:

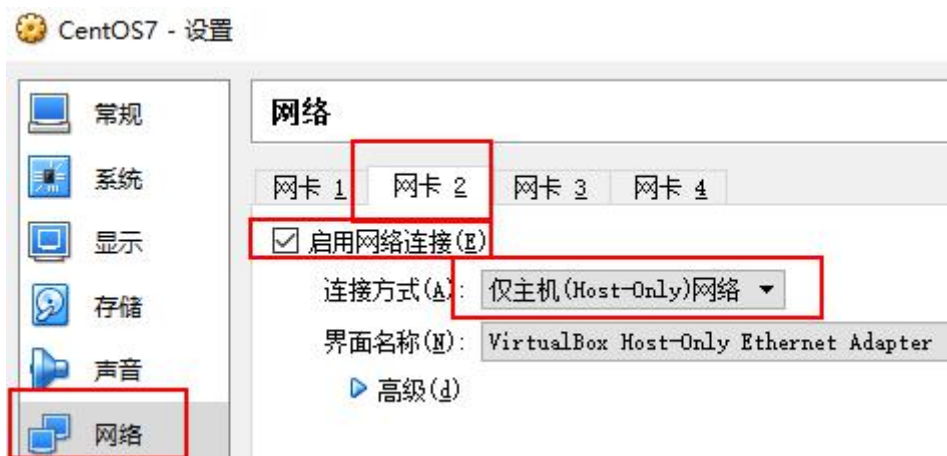


图 1.3.8

- 7) 现在启动这个虚拟机,将会进入安装 CentOS7 的界面,选择 Install CentOS Linux7, 然后就开始安装 CentOS Linux。如图 1.3.9。



图 1.3.9

- 8) 在安装过程中出现选择语言项目,可以选择【中文】。选择安装介质,如图 1.3.10 所示。进入安装位置,选择整个磁盘即可,如图 1.3.11 所示。选择最小安装即可。注意,必须要同时选择打开 CentOS 的网络,如图 1.3.12 所示。否则安装成功以后, CentOS 将没有网卡设置的选项。



图 1.3.10



图 1.3.11

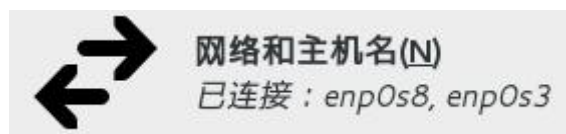


图 1.3.12



图 1.3.12

- 9) 在安装过程中，创建一个非 root 用户，并选择属于管理员组。在其后的操作中，笔者不建议使用 root 用于进行具体的操作。一般情况下，只要执行 sudo 即可以用 root 用户执行相关命令。图 1.3.13 所示。



图 1.3.13



图 1.3.13

- 10) 在安装完成以后，重新启动，并测试是否可以使用之前创建的非 root 用户的用户和名密码登录。



### 【注意】

1. 本书关不是讲 VirtualBox 虚拟机的使用，所以，只给出具体的操作步骤。
2. 在安装过程中，鼠标会在虚拟机和宿主机之间切换。如果要从虚拟机中退出鼠标请按键盘右边的 Ctrl 即可。
3. 关于 Linux 命令请读者自行参考 Linux 手册。如：vim/vi、sudo、ls、cp、mv、tar、chmod、chown、scp、ssh-keygen、ssh-copy-id、cat、mkdir 等将是后面经常使用的命令。

## 1.4、SSH 工具与使用

笔者比较喜欢使用 xShell 做为 Linux 操作的远程命令行工具。同时配合它的 xFtp 可以实现文件的上传与下载。与此软件类似的还有很多软件，如小巧而强大的 putty。与 xshell 功能类似的 CecurityCRT 等。

xshell 是收费软件，不过，读者可以在安装时，选择 free for school 选择免费使用。在安装完成以后，在命令行使用 ssh 即可以登录 linux。

```
ssh 192.168.56.101
```

## 1.5、小结

- ❖ 虚拟机及网络的配置。
- ❖ Linux 操作系统之 CentOS7 的安装过程。
- ❖ 使用 xshell 登录 CentOS7。
- ❖ Linux 的基本命令。

# 第 2 章 Hadoop 伪分布式

## 主要内容

- 安装独立运行的 hadoop。
- hadoop 伪分布式的安装与配置。
- hdfs 的命令。
- Java 操作 hdfs。

Hadoop 的运行方式可以分为三种：

- 1) 独立运行的 hadoop，可以直接在本地运行 mapreduce 程序，并将输出结果保存到本地磁盘上。
- 2) 伪分布式运行的 hadoop，运行 mapreduce 程序并将结果输出到 hdfs 上。
- 3) 集群（HA）运行的 hadoop。用于生产环境的高可靠集群。借助 zookeeper 实现宕机容灾和自动切换。

为了快速上手，我们会运行一个独立的 mapreduce。独立运行的 mapreduce 会读取本地的文件，然后将输出的数据保存到本地文件系统即 Linux 文件系统上。



### 【注意】

本书，后面的环境，都将使用 CentOS7 和 JDK1.8\_x64 做为基础环境。

## 2.1、安装 Java 环境

由于 hadoop 的运行需要 jdk 的环境，所以，在安装 hadoop 之前，必须要先安装 Java 开发环境，即 JDK。

作者使用 xshell 登录 CentOS7 服务器，并使用 xftp 将 JDK1.8 上传到 CentOS7 上。如果对 xshell 和 xftp 不太熟悉，可以通过 xshell 的官网进行学习，xshell 的官方网站为：<https://www.netsarang.com/>。xshell 为收费软件，在安装时，可以选择 free

for school 使用免费选项。

### 步 1、解压 JDK

使用 tar 命令，解压 jdk-1.8.tar.gz:

```
$tar -zxvf jdk-1.8.tar.gz
```

为了让所有用户才可以使用 jdk，可以将 jdk 安装到/usr/local 目录下。

将解压后的文件，拷贝到/usr/local 目录下, 请使用 root 用户:

```
$sudo cp -r jdk-1.8.0 /usr/local
```

### 步 2、配置环境变量

配置环境变量的方式比较多，作者个人，喜欢以创建独立文件的形式，配置环境变量。

创建环境变量文件:

```
$sudo vi /etc/profile.d/java.sh
```

在 java.sh 文件中配置以下信息:

```
#!/bin/bash

export JAVA_HOME=/usr/local/jdk-1.8.0

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

环境变量生效:

```
$source /etc/profile
```

### 步 3、测试 JDK 是否安装成功

执行 java 命令，如果可以正确的显示版本则说明 JDK 已经安装成功:

```
$java -version

1.8.0
```

## 2.2、安装独立运行的 hadoop

独立运行的 hadoop 可以帮助你快速的运行一个 mapreduce 示例，以了解 mapreduce

的运行。后面的测试和基本命令的讲解应该至少应在伪分布式下运行，最好在高可靠集群环境下运行。有些应用，如 hbase、hive 它需要真实的集群环境。

### 步 1、下载 Hadoop

可以使用 wget 下载 hadoop, wget 是 linux 的下载工具。如果没有安装，可以使用 yum 安装 wget, 命令如下：

```
$sudo yum install -y wget
```

下载 hadoop:

```
$wget  
https://mirrors.tuna.tsinghua.edu.cn/apache/hadoop/common/hadoop-2.8.0/hadoop-2.8.0.tar.gz
```

也可以将已经下载好的 hadoop-2.8.0.tar.gz 文件通过 xftp 上传到 Linux 服务器上去。

### 步 2、解压并配置 JAVA\_HOME

下载/上传完成以后解压到当前目录下(使用普通用户即可)：

```
$tar -zxvf hadoop-2.8.0.tar.gz
```

配置环境 Java 的环境变量，修改 hadoop-2.8.0/etc/hadoop/hadoop-env.sh，找到：\${JAVA\_HOME} 设置为本机 JAVA\_HOME 的地址：

```
# set to the root of your Java installation  
export JAVA_HOME=/usr/java/latest
```

输入 hadoop 命令，查看输出, hadoop 可执行文件，在 hadoop\_home/bin 目录下：

```
$ bin/hadoop version
```

```
hadoop 2.8.0
```

上面的命令，会输出 hadoop 的版本信息，则独立运行的方式安装成功。

### 步 3、独立运行的 mapreduce 示例

hadoop 可以运行在一个非分布式的环境下，如运行为一个独立的 java 进程。现在让我们运行一个 wordcount 的 mapreduce 示例。

在当前用户的目录下，创建一个子目录：



```
~$ mkdir input
```

进入 input 目录，并编程一个文件，内容如下：

```
$ vim word.txt  
$ cat word.txt  
Hello   Jack  
Hello   Jerry  
Hello   Jack  
Hello   Mary  
Hello   Alex
```

执行 wordcount 测试：

```
$bin/hadoop jar \  
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.8.0.jar \  
wordcount \  
/home/wangjian/input \  
/home/wangjian/out
```

命令说明：

Hadoop jar 用于执行一个 mapreduce 示例。在 Linux 中，如果命令有多行，可以通过输入\ (斜线) 换行。  
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.8.0.jar 用于指定执行哪一个 jar 文件。wordcount 是执行的任务，/home/wangjian/input 是输入的目录，/home/wangjian/out 是程序执行成功以后的输出目录。

程序执行成功以后，进入 out 输出目录，查看输出目录中的数据：

```
$ cat /home/wangjian/out/*  
  
Alex      1  
Hello     5  
Jack      2
```

Jerry	1
Mary	1

可见，已经对 input 目录中文件中的数据进行统计。到此，独立运行模式的 hadoop 已经运行完成。

## 2.3、Hadoop 伪分布式安装与配置

Hadoop 伪分布式，即在单机模式下运行 Hadoop。在这种环境下，将运行一个 NameNode 进程和一个 SecondaryNameNode 进程，一个 ResourceManager 进程及一个 NodeManager 和一个 DataNode 进程。共 5 个进程。其中 SecondaryNameNode 用于帮助 NameNode 进行日志和镜像文件的合并。ResourceManager 用于处理 MapReduce 任务，它运行在 yarn 之上。NodeManager 用于处理与 DataNode 的通讯和数据处理，DataNode 用于存取数据。

伪分布式可以让读者快速的学习 hdfs 的命令及开发 Mapreduce 应用。对于学习 hadoop 有很大的帮助。

在安装之前，笔者有以下建议：

- 1) 配置静态 ip 地址。虽然是单机模式，但也建议配置静态的 ip 地址，这有助于以后配置集群环境时，养成良好的习惯。
- 2) 修改主机名称。笔者的习惯是主机的名称为公司名称+本机 ip 地址的最后一段。如你的公司英文名称为 weric，本机的 ip 地址为：192.168.56.101 则可以修改本主机的名称为 weric101。考虑到读者可能没有具体的公司，也可以将主机名称命名为 hadoop101 等。总之便于记忆即可。
- 3) 由于启动 hadoop 的各个进程使用的是 ssh。所以，必须要配置本机免密码登录。本章后面的步骤会讲到如何配置 ssh 免密码登录。
- 4) 关闭防火墙。如果你的 CentOS7 没有安装防火墙，可以不用关闭了，如果已经安装了，请检查防火墙的状态，如果是运行状态请关闭防火墙并禁用防火墙开机自

动运行。

- 5) 使用非 root 用户。笔者在写本书前看了很多其他的教程，这些教程大多以 root 用户做为操作用户。但在真实的环境中，一般 root 用户只做一些特殊的操作。所以，笔者还是建议使用非 root 用户。本书的作者为王健，所以，后面的操作，都以 weangjian 做为操作的用户。读者也可以自己创建一个用户。并将此用户添加到 wheel 组，以便于执行 sudo 命令。

### 步 1、配置静态 IP 地址

使用 ssh 登录 CentOS7。然后使用 ifconfig 查看 ip 地址，如果没有 ifconfig 命令，可以使用 **sudo yum -y install net-tools** 安装 ifconfig 命令。其实在 centOS7 中，已经使用 ip addr 命令显示当前主机的 ip 地址。所以，你也可以不安装 net-tools。

```
$ ifconfig

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.56.201 netmask 255.255.255.0
```

上例显示为两块网卡，其中 enp0s3 的 ip 地址为：10.0.2.15 此网卡为 NAT 网络，用于上网。enp0s8 的 ip 地址为：192.168.56.201，此网卡为 HostOnly 网络，用于与宿主机进行通讯。我们要修改的就是 enp0s8 这个网卡，将它的 ip 地址设置为固定 ip。

ip 设置保存在文件中，这个目录为：/etc/sysconfig/network-scripts/。使用 cd 命令，切换到这个目录下。使用 ls 显示这个目录下的所有文件，你可能只会发现：ifcfg-enp0s3 这个文件，现在使用 cp 命令将 ifcfg-enp0s3 拷贝一份为 ifcfg-enp0s8。

```
$ sudo cp ifcfg-enp0s3 ifcfg-enp0s8
```

使用 vim 命令修改为静态 ip 地址：

```
$ sudo vim ifcfg-enp0s8
```

将原来的 dhcp 修改成 static 即静态的地址，并设置具体的 ip 地址。其中，每一个网卡，都应该具有唯一的 UUID，所以建议修改任意的一个值，以便于与之前的 enp0s3

的 uuid 不同。部分修改内容如下：

```
BOOTPROTO="static"

NAME="enp0s8"

UUID="d2a8bd92-cf0d-4471-8967-3c8aee78d101"

DEVICE="enp0s8"

IPADDR="192.168.56.101"
```

现在重新启动网络：

```
$ sudo systemctl restart network.service
```

由于重新启动网络，会修改 ip 地址，所以，在重新启动网络以后，必须要使用新的 ip 地址重新登录。（建议使用 xshell 远程登录）。

```
ssh 192.168.56.101
```

## 步 2、修改主机名称

使用 hostname 命令，检查当前主机的名称：

```
$ hostname

localhost
```

使用 hostnamectl 命令，修改主机的名称：

```
$ sudo hostnamectl set-hostname weric101
```

重新检查主机名称，可以发现，已经修改成了 weric101。

## 步 3、配置 hosts 文件

hosts 文件是本地 DNS 解析文件。配置此文件，可以根据主机名找到对应的 ip 地址。

使用 vi 命令，打开这个文件，并在文件中追加以下配置：

```
$ sudo vim /etc/hosts

192.168.56.101 weric101
```

## 步 3、关闭防火墙

CentOS7 默认情况下，没有安装防火墙。你你可以通过命令 `sudo firewall-cmd`

--state 检查防火墙的状态，如果显示 command not found 一般为没有安装防火墙，此步可以忽略。以下命令检查防火墙的状态：

```
$ sudo firewall-cmd --state
```

running 表示，防火墙正在运行。以下命令用于停止和禁用防火墙：

```
$sudo systemctl stop firewalld.service
```

```
$sudo systemctl disable firewalld.service
```

#### 步 4、配置当前用户免密码登录自己

使用 ssh 登录另一台主机，必须要输入密码，即使登录当前主机也必须要输入密码，检查登录当前主机是否输入密码：

```
$ ssh weric101
```

正如上面显示的，如果提示输入密码，则没有 ssh 免密码登录。可以使用 ssh-keygen 命令生成公钥和私钥文件，并将公钥文件拷贝到被 ssh 登录的主机上。以下是 ssh-keygen 命令，输入以后直接按两次回车即可以生成公钥和私钥文件：

```
$ ssh-keygen -t rsa
```

生成的公钥和私钥文件，将被放到 ~/.ssh/ 目录下。其中 id\_rsa 文件为私钥文件，rd\_rsa.pub 为公钥文件。现在我们再使用 ssh-copy-id 将公钥文件发送到目标主机。由于是登录本机，所以，直接输入本机的名称即可：

```
$ ssh-copy-id weric101
```

此命令执行以后，会在 ~/.ssh 目录下，多出一个用于认证的文件，其中保存了某个主机可以登录的公钥信息，这个文件为： ~/.ssh/authorized\_keys。如果读者感兴趣，可以使用 cat 命令查看这个文件中的内容。

现在再使用 ssh weric101 登录本机，你将发现，不用再输入密码，即可以直接登录成功。

#### 步 5、安装和配置 hadoop

使用 xftp 将 hadoop 安装文件上传到 CentOS7。在 CentOS 的根目录下。

创建一个目录/weric。用于安装以后所有 hadoop 及其他集群的软件：

```
$ sudo mkdir /weric
```

使用 sudo 创建的目录，所属用户和组都是 root。为了让当前非 root 用户使用这个目录，使用 chown 修改所属用户和组：

```
$ sudo chown wangjian:wangjian /weric
```

现在将 hadoop 的安装文件，解压到这个目录，使用 tar 命令：

```
$ tar -zxvf ~/hadoop-2.8.0.tar.gz -C /weric
```

上面的命令中，如果当前在/weric 目录下可以省去后面的-C 参数。-C 参数用于指定将文件解压到指定的目录下。在解压完成以后，要修改 hadoop 的配置文件。hadoop 的配置文件，位于/weric/hadoop-2.8.0/etc/hadoop 目录下，好 hadoop 解压目录下的 etc/hadoop 目录下。一共需要修改五个配置文件。

配置 JAVA\_HOME，在配置文件 hadoop-env.sh 文件中配置：

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_131
```

配置 core-site.xml 文件：

```
<configuration>

    <!--用于指定 hdfs namenode 的地址 -->

    <property>

        <name>fs.defaultFS</name>

        <value>hdfs://weric101:9000</value>

    </property>

    <!-- 用于指定 hadoop 运行时产生文件的保存目录，name,data 目录将默认
保存到此目录下,所配置的目录，当前用户应该有写的权限-->

    <property>

        <name>hadoop.tmp.dir</name>

        <value>/weric/hadoop_tmp_dir</value>

    </property>

</configuration>
```

配置 hdfs-site.xml 文件:

```
<configuration>

    <!-- 指定 hdfs 保存数据副本的数量-->

    <property>

        <name>dfs.replication</name>

        <value>1</value>

    </property>

</configuration>
```

配置 mapred-site.xml 文件, 由于\$HADOOP\_HOME/etc/hadoop 目录下, 并没有 mapred-site.xml 文件, 可以将 mapred-site.xml.template 文件修改成 mapred-site.xml 文件。然后再修改里面的内容:

```
$ cp mapred-site.xml.template mapred-site.xml
$ vim mapred-site.xml

<configuration>

    <property>

        <!-- 配置 mapreduce 运行的平台为 yarn -->

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

    </property>

</configuration>
```

配置 yarn-site.xml 文件:

```
<configuration>

    <!-- NodeManager 获取数据的方式为 shuffle 的方式-->

    <property>

        <name>yarn.nodemanager.aux-services</name>

        <value>mapreduce_shuffle</value>
```

```
</property>

<!-- 指定 yarn 的老大 resourcemanager 的地址-->

<property>

    <name>yarn.resourcemanager.hostname</name>

    <value>weric101</value>

</property>

</configuration>
```

## 步 6、配置 hadoop 环境变量

可以以独立的方式配置环境变量，只需要在/etc/profile.d/目录下创建一个 sh 文件即可。这样做的好处是文件相对比较独立。

```
$ sudo vim /etc/profile.d/hadoop.sh
```

并在里面添加以下内容：

```
#!/bin/sh

export HADOOP_HOME=/weric/hadoop-2.8.0

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

使用 source 命令，让环境变量生效：

```
$source /etc/profile
```

然后使用 hdfs version 查看命令环境变量是否生效，如果配置成功，则会显示 hadoop 的版本：

```
[wangjian@weric101 weric]$ hdfs version

Hadoop 2.8.0
```

## 步 7、初始化 hadoop 的文件系统

hadoop 在使用之前，必须要先初始化 hdfs 文件系统，初始化的文件系统将会生成 为 hadoop.tmp.dir 配置的目录下，即上面配置的/weric/hadoop\_tmp\_dir 目录下。

```
$hdfs namenode -format
```

在执行命令完成以后，请在输出的日志中，找到：



```
Storage directory /weric/hadoop_tmp_dir/dfs/name has been successfully formatted.
```

这句话，即为初始化成功。

## 步 8、启动 hdfs 和 yarn

启动和停止 hdfs 及 yarn 的脚本在 \$HADOOP\_HOME/sbin 目录下。其中 start-dfs.sh 为启动 hdfs 的脚本，start-yarn.sh 为启动 ResourceManager 的脚本。以下分别启动 dfs 和 yarn：

```
$ hadoop-2.8.0/sbin/start-dfs.sh  
$ hadoop-2.8.0/sbin/start-yarn.sh
```

启动完成以后，通过 jps 来查看 java 进程快照，你会发现有五个进程正在运行：

```
2608 NameNode  
2737 DataNode  
3505 Jps  
2898 SecondaryNameNode  
3156 NodeManager  
3055 ResourceManager
```

其中：NameNode、SecondaryNameNode、DataNode 是通过 start-dfs.sh 脚本启动的。ResourceManager 和 NodeManager 由 start-yarn.sh 脚本启动的。

在启动成功以后，也可以通过 <http://192.168.56.101:50070> 查看 namenode 的信息，如图 2.3.1 所示：



图 2.3.1

可以通过 `http://192.168.56.101:8088` 查看 mapreduce 的信息,如图 2.3.2 所示:

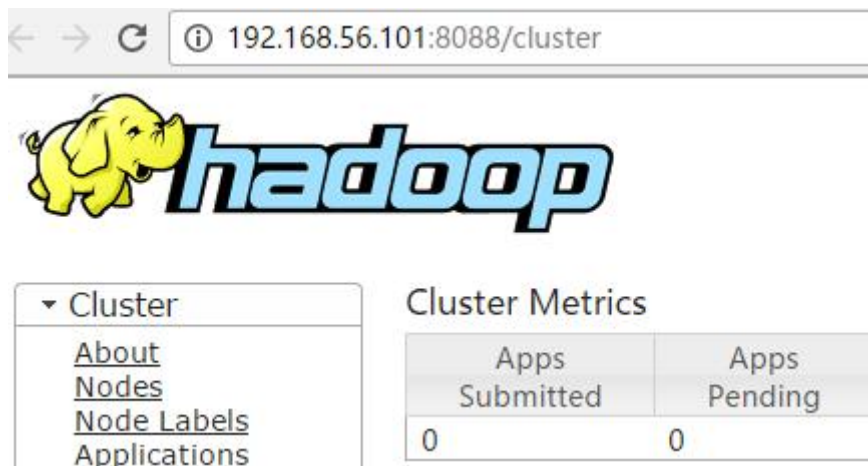


图 2.3.2

至此, hadoop 单机即伪分布式模式安装和配置成功。但是万里长征,我们这才是小小的一步。以下将通过 hadoop 的 `hdfs` 命令,操作 hadoop 的 `hdfs` 文件系统。

## 2.4、hdfs 命令简介

`hdfs` 命令,位于 `$HADOOP_HOME/bin` 目录下。由于已经配置了 `HADOOP` 的环境变量,所以此命令,可以在任意的目录下执行。可以通过直接输入 `hdfs` 命令,查看它的使用帮助:

```
$ hdfs

Usage: hdfs [--config confdir] [--loglevel loglevel] COMMAND
where COMMAND is one of:

    dfs      run a filesystem command on the file systems supported in Hadoop.

    classpath      prints the classpath

    namenode -format  format the DFS filesystem

    secondarynamenode  run the DFS secondary namenode

    namenode        run the DFS namenode

    journalnode      run the DFS journalnode
```

zkfc	run the ZK Failover Controller daemon
datanode	run a DFS datanode
debug	run a Debug Admin to execute HDFS debug commands
dfsadmin	run a DFS admin client
haadmin	run a DFS HA admin client
fsck	run a DFS filesystem checking utility
balancer	run a cluster balancing utility
jmxget	get JMX exported values from NameNode or DataNode.
mover	run a utility to move block replicas across storage types
oiv	apply the offline fsimage viewer to an fsimage
oiv_legacy	apply the offline fsimage viewer to an legacy fsimage
oev	apply the offline edits viewer to an edits file
fetchdt	fetch a delegation token from the NameNode
getconf	get config values from configuration
groups	get the groups which users belong to
snapshotDiff	diff two snapshots of a directory or diff the current directory contents with a snapshot
lsSnapshottableDir	list all snapshottable dirs owned by the current user
	Use -help to see options
portmap	run a portmap service
nfs3	run an NFS version 3 gateway
cacheadmin	configure the HDFS cache
crypto	configure HDFS encryption zones

```
storagepolicies      list/get/set block storage policies

version              print the version

Most commands print help when invoked w/o parameters.
```

上面的这些命令，在后面的课程中，基本都会涉及。现在让我们来查看几个使用比较多年命令。在上面的列表中，第一个 `dfs` 是经常被使用到的命令。可以通过 `hdfs dfs -help` 查看 `dfs` 的具体使用。由于参数过多，本书就不一一列表举了。`dfs` 命令，就是通过命令行，操作 `hdfs` 目录或是文件的命令，类似于 `linux` 文件命令一样，只不过操作的是 `hdfs` 文件系统。以下是列表的形式列示几个常用命令：

命令	功能	示例
<code>-ls</code>	用于显示 <code>hdfs</code> 文件系统上的所有目录和文件	<code>hdfs dfs -ls /</code> <code>hdfs dfs -ls hdfs://weric101:9000/</code>
<code>-mkdir</code>	在 <code>hdfs</code> 上创建一个新的目录	<code>hdfs dfs -mkdir /weric</code>
<code>-rm -r</code>	删除 <code>hdfs</code> 上的一个目录，其中 <code>-r</code> 参数为递归删除所有子目录。如果没有使用 <code>-r</code> 参数，则是删除一个文件。	<code>hdfs dfs -rm -r /weric</code>
<code>-cat</code>	显示 <code>hdfs</code> 上某个文件中的所有数据	<code>hdfs dfs -cat /weric/a.txt</code>
<code>-put</code>	将本地文件上传到 <code>hdfs</code> 上去	<code>hdfs dfs -put /hello.txt /hello.txt</code>
<code>-get</code>	从 <code>hdfs</code> 上将文件保存到本地	<code>hdfs dfs -get /src.txt ~/hello.txt</code>

其他更多命令如 `-putFromLocal`、`-copyToLocal`、`-cp`、`-mv` 等都可以参考帮助实现。

## 2.5、Java 代码操作 hdfs

通过 `Java` 代码操作 `hdfs` 可以获取 `hdfs` 命令行所有的能力。并且可以用于后面 `mapreduce` 开发过程。主要接口文件：`FileSystem` 在 `hadoop-commons.jar` 中。以下是使用到的具体类：

```
import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path
```

创建 `FileSystem` 对象，创建 `FileSystem` 对象，就是通知 Java 代码哪儿可以访问到 hdfs 的地址，这个 uri 的配置即为之前配置的 `fs.defaultFS` 的值。由于经常开发 java 代码为 windows 操作系统，所以，应该输入完整的 ip 地址。除非在 windows 上也配置了 `weric101` 与 `192.168.56.101` 的对应关系。以下是创建 `FileSystem` 对象的代码：

```
FileSystem fs =

    FileSystem.get(new URI("hdfs://192.168.56.101:9000"),

        new Configuration(), "wangjian");
```

在上面的代码中，第一个参数为 hdfs 的地址，第二个参数为配置信息，目前这个参数为一个新的对象，且没有配置任何的内容，但在高可靠 (HA) 的配置中，它将起到配置高可靠访问形式的作用。第三个参数 `wangjian` 为 hdfs 文件系统的用户。如果你的 windows 用户与 linux 的用户名不一致。传递一个名称后，hadoop 会模拟此用户名称去操作 hdfs 的文件系统。否则会抛出：`Access Denied` 访问被拒绝的异常。

上传文件，从本地向服务器 copy 文件：

```
fs.copyFromLocalFile(new Path("D:/a/a.txt"), new Path("/a.txt"));

fs.close();
```

创建一个目录：

```
fs.mkdirs(new Path("/dir1"));
```

下载服务器的文件到本地

```
InputStream in = fs.open(new Path("/a.txt"));

IOUtils.copyBytes(in, new FileOutputStream("D:/a/b.txt"), 1024 * 4);

in.close();
```

## 2.6、小结

- ❖ 配置 hadoop 单机即伪分布式的环境。
- ❖ 配置 hadoop 的五个配置文件。
- ❖ 使用 hadoop 的脚本 start-dfs.sh 和 start-yarn.sh 启动 hdfs 和 yarn。
- ❖ hadoop 启动以后的五个进程分别是哪几个。
- ❖ hadoop 启动以后，通过 50070，8088 两个端口访问 namenode/hdfs 和 resourcemanager/yarn。
- ❖ hdfs 的命令行操作。
- ❖ Java 操作 hdfs。

## 第 3 章 HDFS 文件系统

### 主要内容

- HDFS 的体系结构
- HDFS 命令
- RPC 远程调用
- Hadoop 各进程的功能

Hadoop DISTRIBUTED FILE SYSTEM, 简称 HDFS, 是一个分布式文件系统。它是谷歌的 GFS 提出之后出现的另外一种文件系统。它有一定高度的容错性, 而且提供了高吞吐量的数据访问, 非常适合大规模数据集上的应用。HDFS 提供了一个高度容错性和高吞吐量的海量数据存储解决方案。

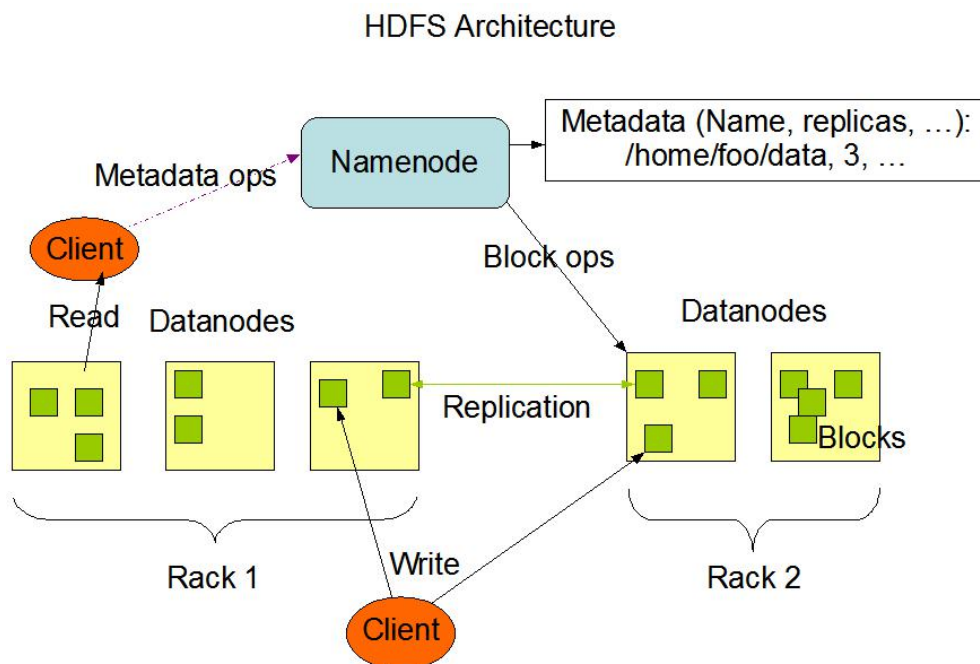


图 3.1.1

## 3.1、HDFS 的体系结构

图 3.1.1 展示了，客户端如何与 hdfs 进行交互过程：

对于图 3.1.1 的具体说明如下：

- 1) 客户端与 NameNode 进行交互, NameNode 通知 Client 将数据保存在哪一台 DataNode 上。由 Client 保存数据到 DataNode 上。NameNode 中 metadata 用于在内存保存数据保存的元信息。在 `{hadoop.tmp.dir}/dfs/name` 目录下，保存了 edits 和 fsimages 文件，它们分别是日志文件和 metadata 的镜像文件。
- 2) 在保存成功以后，由 NameNode 来保存数据的元信息。即什么数据保存到哪一台机器上。
- 3) 每一块文件的大小在 Hadoop2.0 以后为 128M。
- 4) SecondaryNameNode 用于时时管理 NameNode 中的元数据信息。执行更新和合并的工作。

## 3.2、NameNode 的工作

在伪分布式的环境中，NameNode 只有一个。但在分布式的环境中，NameNode 被抽象为 NameService。而每一个 NameService 下最多可以有两个 nameNode。这种情况下，一个 NameNode 为 Active，另一个为 Standby。NameNode 主要用于保存元数据保存并接收客户端的请求。NameNode 的具体工作为：

- 1) 保存 metadata 元数据，始终在内存中保存 metadata 元数据信息用于处理读请求。
- 2) 维护 fsimage 文件，也就是 metadata 元数据信息的的镜像。此文件保存在 hdfs 目录下的 name 目录下。
- 3) 当写请示来时，首先写 editlog 即向 edits 写日志。成功返回以后才会写内存。
- 4) SecondaryNameNode 同时维护 fsimage 和 edits 文件以更新 NameNode 的 metadata 元数据。图 3.2.1 展示了 NameNode 怎么把元数据保存到磁盘上的。





图 3.2.1

这里有两个不同的文件：

1)、fsimage 文件，它是在 NameNode 启动时对整个文件系统的快照，是 metadata 的镜像。

2)、edit logs 文件，每当写操作发生时，NameNode 会首先修改这个文件，然后再去修改 metadata。

元数据信息包含：

1)、fsimage 为元数据的镜像文件，用于保存一段时间 NameNode 中元数据的信息。

2)、edits 保存了数据的操作日志。

3)、fstime 保存最近一次 checkpoint 的时间。fstime 保存在内存中。

这些文件都保存在 Linux 系统文件中，如下：

```

$ pwd

/tmp/hadoop-root/dfs/name/current

$ls

edits_000..01-00...013 fsimage_0000..000013      seen_txid
edits_000..014-00..023  fsimage_00000..00013.md5  VERSION
edits_000..024-00..025  fsimage_000..000025
edits_inprogress_000..026      fsimage_000..025.md5
  
```

NameNode 的工作特点：

NameNode 始终在内存中保存 metadata。在处理读写数据时，会先写 edits 到

磁盘，成功返回以后修改内存中的 metadata。

NameNode 会维护一个 fsimage 文件，此文件是 metadata 保存在磁盘上的镜像文件。（Hadoop2.x 中 fsimage 与 metadata 保持时时同步。Hadoop1.x 不是时时同步）每隔一段时间 SecondaryNameNode 会合并 fsimage 和 edits 来更新内存中的 metadata。

### 3.3、Secondary NameNode

HA（高可用）的一个解决方案。在伪分布式中存在 SecondaryNameNode，在正式的集群中没有 SecondaryNameNode。SecondaryNameNode 的职责是合并 NameNode 的 edit logs 到 fsimage 文件中。正如图 3.3.1 所示：

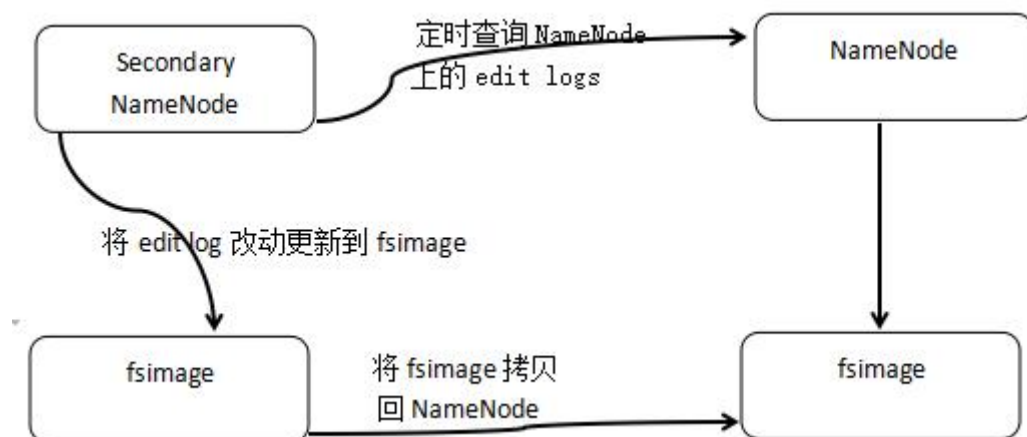


图 3.3.1

图 3.3.1 展示了 Secondary NameNode 是怎么工作的。

- 1)、它定时到 NameNode 去获取 edit logs[内存中的 editlog 即为 metadata]，并更新到 fsimage 上。
- 2)、NameNode 在下次重启时会使用这个新的 fsimage 文件，将这个 fsimages 文件恢复到内存创建新的 metadata 元数据信息。
- 3)、Secondary NameNode 的整个目的是在 HDFS 中提供一个检查点。它只是 NameNode 的一个助手。

现在，我们明白了 Secondary NameNode 所做的不过是帮助 NameNode 更好的工作。它不是要取代掉 NameNode 也不是 NameNode 的备份。

### 3.4、DataNode

DataNode 的功能是：

1)、提供真实的存储服务。

2)、在 Hadoop2.0 里面每一个文件块的大小为 128M。在 Hadoop2.0 中，文件默认块大小为 128M，如果一个文件，没有 128M 则上传的文件，将会占用一个实际大小的空间。如果文件大于 128M 则文件将会被分割成多个文件块。你可以通过上传一个大于 128M 的文件，而后，查看上传以后文件是否分成多个文件的形式保存来查看。

3)、保存的目录为以下在 core-site.xml 中配置的内容：hadoop.tmp.dir，如果没有配置则默认值为：/tmp/hadoop-\${user.name}。在上面的目录下，有一个 data 目录。里面就是保存数据的位置。

4)、在 hdfs-site.xml 中配置 dfs.replication 的值，默认值为 3。

5)、假设一个文件的大小为 384M(正好可以被 hadoop 切分成 3 份,即  $384/128=3$ )，假设 Hadoop 的集群中，DataNode 的个数为 4，dfs.replication 的值为 3，即每一个文件块默认保存 3 份，则图 3.4.1 展示了文件在 hadoop 集群中保存示例。

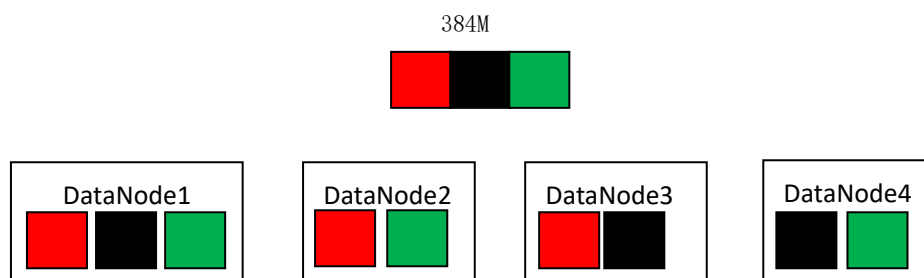


图 3.4.1

通过图 3.4.1 可以看出，一个 384M 的文件，被切分为三块分别为红、黑、绿。由于指定的副本数量为 3，DataNode 节点的数量为 4，在 DataNode1 节点上，保存了文

件块红、黑、绿,DataNode2保存了文件块红、绿,DataNode3保存了文件红、黑,DataNode4保存了文件黑、绿。而所有文件块最都有三份副本。

### 3.5、HDFS 的命令

在 hadoop1.x 版本中,使用 hadoop 命令管理 hdfs 文件系统。在 Hadoop2.x 版本中,使用 hdfs 命令管理 hdfs 文件系统。

以下是 hadoop1.0 版本的命令,现在依然可以使用:

```
$hadoop fs -ls hdfs://192.168.56.203:9000/
Found 2 items
-rw-r--r--      1 root  supergroup          1627  2017-05-10  17:44
hdfs://192.168.56.203:9000/a.txt
drwxr-xr-x      - root  supergroup           0  2017-05-11  11:58
hdfs://192.168.56.203:9000/dir1
```

也可以省去 hdfs://192.168.56.302:9000, 直接输入/ (斜线) 即可:

```
$ hadoop fs -ls /
Found 2 items
-rw-r--r--      1 root  supergroup          1627  2017-05-10  17:44 /a.txt
drwxr-xr-x      1- root  supergroup           0  2017-05-11  11:58 /dir1
```

以下是 hadoop2.0 的命令,使用 hdfs 开头,使用 hdfs 命令,并指定完整地址:

```
$ hdfs dfs -ls hdfs://192.168.56.203:9000/
-rw-r--r-- .. ... .. hdfs://192.168.56.203:9000/a.txt
drwxr-xr-x .. ... .. hdfs://192.168.56.203:9000/dir1
```

使用 hdfs 命令,省略完整地址:

```
$ hdfs dfs -ls /
Found 2 items
```

```
-rw-r--r--    .. ... ..      1627 2017-05-10 17:44 /a.txt
drwxr-xr-x    .. ... ..           0 2017-05-11 11:58 /dir1
```

以下是几个常用的命令，显示服务器文件列表：

```
hdfs dfs -ls /
hdfs dfs -ls hdfs://192.168.56.203:9000/
```

将本地文件 copy 到 hdfs 上去：

```
$hdfs dfs -copyFromLocal ~/home/wangjian/some.txt /some.txt
```

查看服务器上的文件内容：

```
$hdfs dfs -cat /some.txt
```

从服务器下载文件到本地：

```
$hdfs dfs -copyToLocal /test1.txt test1.txt
```

服务器文件和文件夹记数

```
$hdfs dfs -count /
```

向服务器上传文件

```
$hdfs dfs -put test1.txt /test2.txt
```

从服务器获取文件到本地

```
$hdfs dfs -get /test2.txt test3.txt
```

## 3.6、RPC 远程过程调用

在 hadoop 中，多个进程 (NameNode/DataNode 等等) 之间数据的传递和访问都是通过 RPC 实现的。

RPC 是不同进程之间方法调用的解决方案。RPC 调用的原理是通过网络代理实现远程方法的调用。这些功能，已经被 Hadoop 封装，直接使用 Hadoop 提供的类：RPC 即可以提供服务器。

被调用的类，由于是通过动态代理实现的，所有，必须要拥有一个接口。且接口

上，必须要拥有一个 `public static final long versionID=xxxxL`。

以下简单通过 Hadoop RPC 代码调用示例，演示 RPC 在多个进程之间调用的过程。

### 步 1：开发一个接口

在接口中，必须要拥有一个 `versionID` 唯一的标识当前接口。

```
package cn.hadoop;

/**
 * 开发一个接口，远程调用必须要使用此接口。
 */
public interface IHello {

    public static final long versionID = 1L;//接口上必须要拥有一个 id

    String say(String name);

}
```

### 步 2：开发实现类，并实现接口中的方法

实现接口，且实现里面的方法。

```
package cn.hadoop;

/**
 * 1:声明一个类，并声明一个方法，返回一个任意的字符串<br>
 * 一个类如果希望被远程 RPC 调用，这个类必须要实现一个接口<br>
 * 因为内部的原理是 JDK 动态代理
 */
public class HelloServer implements IHello {

    public String say(String name){

        System.err.println("this is in Server...");

        return "Hi "+name;

    }

}
```

### 步 3：开发服务器

服务器通过 RPC.Builder 来创建服务。

```
package cn.hadoop;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;
import org.apache.hadoop.ipc.RPC.Server;

public class Demo01_Server {

    /**
     * 开一个入口方法，使用 RPC 启动服务
     * @param args
     */

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

        Server server = new RPC.Builder(new Configuration())// 创建 Builder
            .setProtocol(IHello.class)// 设置接口
            .setInstance(new HelloServer())// 设置实现类
            .setBindAddress("192.168.1.101")// 设置地址
            .setPort(9543)// 设置端口
            .build();

        server.start();// 启动服务器 - 服务器启动以后，会一直运行，有客户端
        连接即会实现远程调用
    }
}
```

### 步 4：开发客户端

使用 RPC.getProxy 获取本地的一个代理，但是接口类必须要与服务器的接口类，保持一致：

```
package cn.hadoop;
```

```

import java.net.InetSocketAddress;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.ipc.RPC;

public class Demo01_Client {

    /**
     * 3: 声明 一个新的进程使用 rpc 实现远程调用<br>
     * 但必须要使用同一个接口类
     */

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

        // 直接返回接口的代理对象

        IHello helloProxy = RPC.getProxy(IHello.class, 1L,

            new InetSocketAddress("192.168.1.101", 9543),

            new Configuration());

        //只能调用 say 方法，不能调用除接口中以外的任何的方法

        String str = helloProxy.say("Hadoop");

        System.err.println("返回的数据是: " + str);

    }

}

```

最后，请运行上面的程序代码，如果可以正常返回数据则说明 RPC 远程调用成功。

在 Hadoop 中，很多都是通过 RPC 调用实现的。

### 3.7、小结

- ❖ hadoop 默认的配置在 core-default.xml 文件中。此文件在 hadoop-common.jar 包中。里面的部分配置如：

```
<property>
```



```

<name>hadoop.common.configuration.version</name>

<value>0.23.0</value>  hadoop 的版本，可见目前的版本是从 0.23.0 发展来的

<description>version of this configuration file</description>
</property>

<property>

  <name>hadoop.tmp.dir</name>

  <value>/tmp/hadoop-${user.name}</value>  默认的数据文件目录

  <description>A base for other temporary directories.</description>
</property>

```

建议不要修改 core-default.xml 文件中的内容，如果需要修改，可以修改 core-site.xml 中的内容。

- ❖ 在 hadoop-hdfs.jar 中包含有 hdfs-default.xml 文件，里面保存了默认的 hdfs 配置如：

```

<property>  默认的副本数量

  <name>dfs.replication</name>

  <value>3</value>

</property>

<property>  默认的文件块大小 128M

  <name>dfs.blocksize</name>

  <value>134217728</value>

</property>

```

- ❖ FileSystem 创建类为 DistributedFileSystem。内部使用 RPC 实现远程调用。

## 第 4 章 MapReduce

### 主要内容

- MapReduce 的执行原理、执行过程。
- 数据类型及数据格式。
- Writable 接口与序列化机制。
- MapReduce 的源码分析。

MapReduce 是一种可用于数据处理的编程框架。MapReduce 采用“分而治之”的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个分节点共同完成，然后通过整合各个节点的中间结果，得到最终结果。简单地说，MapReduce 就是“任务的分解与结果的汇总”。

在分布式计算中，MapReduce 框架负责处理了并行编程中分布式存储、工作调度、负载均衡、容错均衡、容错处理以及网络通信等复杂问题，把处理过程高度抽象为两个函数：map 和 reduce，map 负责把任务分解成多个任务，reduce 负责把分解后多任务处理的结果汇总起来。

### 4.1、MapReduce 的运算过程

MapReduce 为分布式计算模型，分布式计算最早由 google 提出。MapReduce 将运算的过程分为两个阶段，map 和 reduce 阶段。用户只需要实现 map 和 reduce 两个函数即可。这两个函数参数的形式都是：Key、Value 对。表示函数的输入和输出信息。

图 4.1.1 展示了 MapReduce 的运算过程。将大任务交给多个机器分布式进行计算，然后再进行汇总合并。

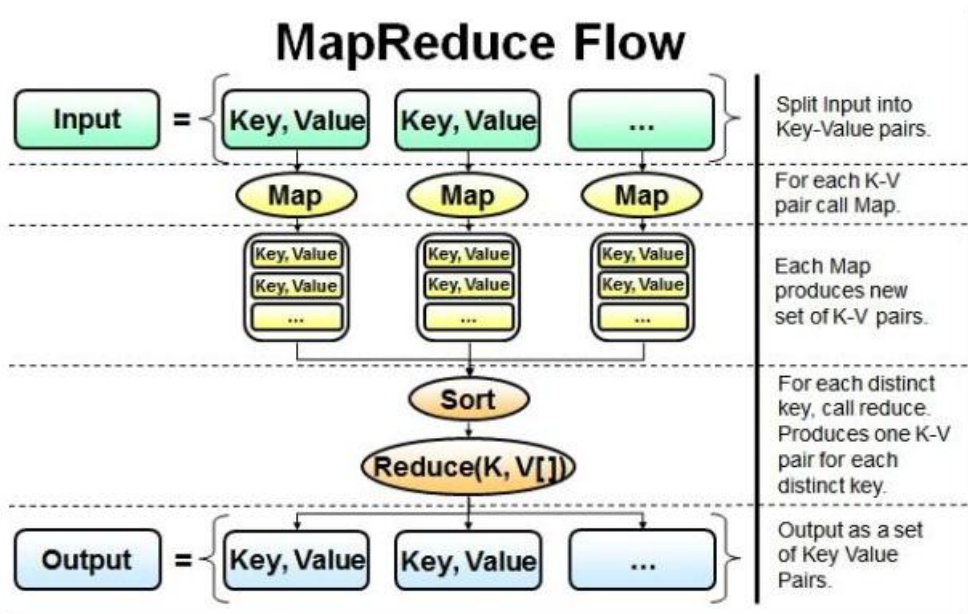


图 4.1.1

(图片来自于 Hadoop 官网)

以 WordCount 为示例，再为读者讲解 MapReduce 的过程。请先看图 4.1.2:

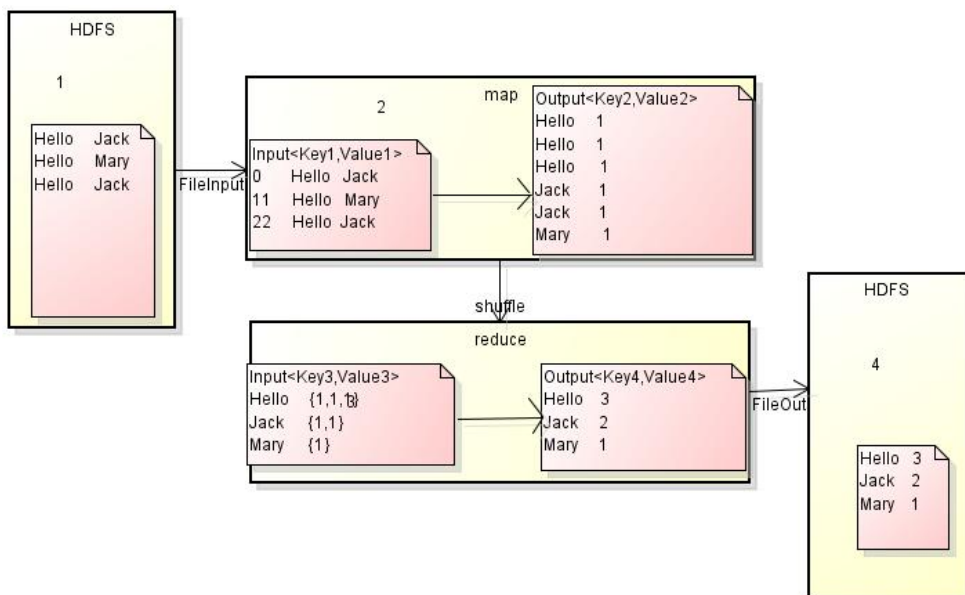


图 4.1.2

标注为 1 的部分为 hadoop 的 hdfs 文件系统,被处理的数据,应该首先保存到 hdfs

文件系统上。

标注为 2 的部分，将接收 `FileInputFormat` 的输入数据，在处理 `Wordcount` 示例时，接收到的数据 `key1` 为 `LongWritable` 类型，即为字节的偏移量。如 2 中第一行输入为 0，其中 0 为字节 0 下标的开始，第二行为 11，则 11 为文本中第二行字节的偏移量，依次类推。而 `Value1` 则为 `Text` 即文本类型，其中如第一行 `Hello Jack` 为读取的第一行的数据，依次类推。然后，此时我们将开发代码对 `Value1` 的数据进行处理，以空格或是 `\t` 做为分割，分别将 `Hello` 和 `Jack` 依次输出。此时每一次输出的算是一个字符，所以，在 `map` 中的输出格式为 `key2` 为 `Text` 类型，而 `Value2` 则为 `LongWritable` 类型。

标注为 3 的部分，接收 `map` 的输出，所以 `Key3` 和 `Value3` 的类型应该与 `Key2` 和 `value2` 的类型一致。现在我们只需要将 `value` 中的值相加，就可以得到 `Hello` 出现的次数。然后直接输出给 `Key4` 和 `value4`。所以，`key4` 的 `value4` 的类型也可以是 `Text` 和 `LongWritable`。

最终，数据将保存到 HDFS 上。这将是 `key4` 和 `value4` 的数据。

如果你已经简单了解了 `MapReduce` 的过程，就可以快速的来开发 `WordCount` 的代码了。



#### 【注意】

`LongWritable` 和 `Text` 为 Hadoop 中的序列化类型。可以简单的理解为 Java 中的 `Long` 和 `String`。

## 4.2、WordCount 示例

为了快速的让读者掌握 `MapReduce`，以下将从项目创建开始讲起，直到项目可以运行。

### 步 1、创建 Java Maven 项目并添加 maven 依赖

在开发中，可以使用 `maven` 解决依赖包的问题。这目前是大多数公司的通用选择。

如果还没有学会使用 maven，建议读者可以参考我的《maven 快速入门》教程。依赖库：

```
<dependency>

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

    <artifactId>hadoop-mapreduce-client-core</artifactId>

    <version>2. 8. 1</version>

</dependency>

<dependency>

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

    <artifactId>hadoop-hdfs</artifactId>

    <version>2. 8. 1</version>

</dependency>

<dependency>

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

    <artifactId>hadoop-common</artifactId>

    <version>2. 8. 1</version>

</dependency>
```

## 步 2、开发 Mapper

开发 map 只需要继承类 Mapper 即可。注意输入和输出的 key value 类型。

```
package cn.weric.hadoop.wordcount;

import java.io.IOException;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

/**

 * 开发 mapper 用于接收数据，输出数据

 * @author wangjian
```

```

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

    @Override

    public void map(LongWritable key, Text value,

        Mapper<LongWritable, Text, Text, LongWritable>.Context context)

        throws IOException, InterruptedException {

        //处理 Value，使用正则表达式进行字符串分割

        String vv = value.toString();//Hello \t Jack

        //根据\t 和空格进行分割

        String[] strs = vv.split("\\s"); //[Hello, jack]

        //循环输出，第一个字符为 1

        for(String s:strs){

            context.write(new Text(s),new LongWritable(1L));

        }

    }

}

```

### 步 3、开发 Reducer

开发 Reducer 也比较简单，只要继承 Hadoop 的 Reducer 即可。注意输入和输出的 key value 类型。

```

package cn.weric.hadoop.wordcount;

import java.io.IOException;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

/**Reducer 示例 */

```

```

public class WordCountReducer extends Reducer<Text, LongWritable, Text,
LongWritable> {

    public void reduce(Text text, Iterable<LongWritable> value,

        Reducer<Text, LongWritable, Text, LongWritable>.Context context)
throws IOException, InterruptedException {

        long count = 0;

        //进行计算, 迭代累计出现的次数
        for (LongWritable ll : value) {

            count += ll.get();

        }

        //输出到 hdfs 上去
        context.write(text, new LongWritable(count));

    }
}

```

#### 步 4、现在开发一个 main 方法，用于运行 mapreduce

运行一个 mapreduce 任务，需要使用到 job 对象。以下是完整代码。

```

package cn.weric.hadoop.wordcount;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

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

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

/**使用 Job 对象运行 MapReduce 任务*/

public class WordCount {

```

```

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

    // 注册任务

    Job job = Job.getInstance(new Configuration());

    // 设置执行任务的类是哪一个类 hadoop jar ..

    job.setJarByClass(WordCount.class);

    // 设置 mapper, 及输出的数据

    job.setMapperClass(WordCountMapper.class);

    job.setMapOutputKeyClass(Text.class);

    job.setMapOutputValueClass(LongWritable.class);

    // 设置 reducer, 设置整个输出的类型

    job.setReducerClass(WordCountReducer.class);

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(LongWritable.class);

    // 设置读取的文件, 接收命令行第一个参数, 表示读取的文件名

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

    // 输出到的目录, 接收命令行第二个参数为输出目录

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

    // 开始执行

    job.waitForCompletion(true);

}
}

```

## 步 5、现在我将项目打成 jar 包

将上面的代码, 打开 jar 包。可以使用 Eclipse 的 `export>jar` 并指定 Main 函数所在的类。如图 4.2.1 所示。



Select the class of the application entry point:

Main class:

图 4.2.1

### 步 6、执行 mapreduce 任务查看结果

将打好的 jar 包放到安装有 hadoop 的服务器上，并执行以下命令：

```
$hadoop jar wordcount.jar /word.txt /out01
```

Hadoop jar 用于执行一个 mapreduce 的程序。wordcount.jar 是我们之前打包的 mapreduce 程序包。/word.txt 为 hdfs 上的文件，/out01 为输出目标为 hdfs 上的 /out01 目录。必须要保证之前没有 /out01 目录。

执行完成以后，可以通过以下命令，查看输出的结果：

```
$hdfs dfs -cat /out01/*
```

到此，我们的第一个 MapReduce 已经开发和运行完成。如果你能顺利的完成所有过程。那么恭喜你。你已经开始 Hadoop 之旅。



#### 【注意】

可以将 Mapper 和 Reducer 开发成内部类，但这两个内部类，必须是 public static 修饰符。

## 4.3、序列化的概念 Writable 接口

在 Hadoop 中，LongWritable/Text 都是被序列化的类，也只有被序列化的类，才可进行传递。在实际的开发中，为了适应不同业务的要求，有时，必须要自己开发 Writable 类的子类，以实现 Hadoop 中的序列化实现数据传递。以下是 JDK 的序列与 Hadoop 序列化的比较。

JDK 的序列化接口为 java.io.Serializable，用于将对象转换成字节流输出即为

序列化，再将字节流转换成对象，即为反序列化。

Hadoop 的序列化接口为 `org.apache.hadoop.io.Writable`。它的特点是紧凑（高效的使用存储空间）、快速（读写开销小）、可扩展（可以透明的读取数据）、互操作（支持多语言）。

`Writable` 接口的两个主要方法，`write(DataOutput)` 将成员变量按顺序写出和 `readFields(DataInput)` 顺序的读取成员变量的值。

`Writable` 接口的基本实现：

就是在 `write/readFields` 中顺序的写出和读取成员变量的值。以下代码，省略了 `setters` 和 `getters` 方法，请注意 `write` 和 `readFields` 中的代码，必须要按顺序读写数据。

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

/**保存电话流量信息 */
public class PhoneDataWritable implements Writable {

    private String telNo;// 电话号码

    private Long upload;// 上行流量

    private Long download;// 下行流量

    private Long total;// 总流量

    public void write(DataOutput out) throws IOException {

        out.writeUTF(telNo);// 写出数据，就是按顺序写出成员变量

        out.writeLong(upload);

        out.writeLong(download);

        out.writeLong(total);

    }
}
```

```

public void readFields(DataInput in) throws IOException {

    telNo = in.readUTF(); //按顺序读取数据

    upload = in.readLong();

    download = in.readLong();

    total = in.readLong();

}
}

```

在自定义的类实现接口 Writable 以后，就可以将这个类做为 Key 或是 value 放到 Mapper 或是 Reducer 中当做参数了。

## 4.4、Partitioner 编程

Partitioner 分区编程的主要功能是：将不同的规类输出到不同的文件中。这样在查询数据时，可以根据某个规类查询相关的数据。

Map 的结果，会通过 partition 分发到 Reducer 上，Reducer 做完 Reduce 操作后，通过 OutputFormat，进行输出。

Mapper 最终处理的键值对<key, value>，是需要送到 Reducer 去合并的，合并的时候，有相同 key 的键/值对会送到同一个 Reducer 那。哪个 key 到哪个 Reducer 的分配过程，是由 Partitioner 规定的。它只有一个方法：

```
getPartition(Text key, Text value, int numPartitions)
```

输入是 Map 的结果对<key, value>和 Reducer 的数目，输出则是分配的 Reducer（整数编号）。就是指定 Mapper 输出的键值对到哪一个 reducer 上去。系统缺省的 Partitioner 是 HashPartitioner，它以 key 的 Hash 值对 Reducer 的数目取模，得到对应的 Reducer。这样保证如果有相同的 key 值，肯定被分配到同一个 reducer 上。如果有 N 个 reducer，编号就为 0, 1, 2, 3……(N-1)。

**业务分析**，存在以下学统计信息，这些数据的格式如下：

001	张三	89	87	第一学期
001	张三	85	84	第二学期
002	李四	98	94	第一学期
002	李四	99	100	第二学期

数据说明，第一列为学号，第二列为姓名，第三列为数学成绩，第四列为物理成绩，最后一列为学期。

现在要求，根据不同的学期进行成绩总分汇总，将数据分别输出到多个不同的文件中。reducer 输出的数据格式为 part-r-00000。其中 part-r 是指由 reducer 输出的部分，00000 为文件编号。现在我们要通过 Partitioner 编程，将一例中，第一个学期的数据，都输出到 part-r-00000 上去，将第二学期的数据，输出到 part-r-00001 上去。这正是 Partitioner 编程要解决的问题。

要求输出的格式如下，part-r-00000 文件中的内容：

001	张三	176	第一学期
002	李四	192	第一学期

part-r-00001 文件中的内容为：

001	张三	169	第二学期
002	李四	199	第二学期

为了给大家展示清楚开发过程，以下将所有开发过程中的代码都放到这儿来。

准备工作：创建一个 maven 项目，并添加 hadoop 开发的依赖。这儿的依赖与之前开发 MapReducer 的依赖相同。

### 步 1：开发 Stud 类实现接口 Writable

开发 Stud 类，用于实现数据的封闭，实现接口 Writable 接口，实现序列化，序列化技术之前已经讲过。

```
public class Student implements Writable {
    private String id;
    private String name;
```

```

private Integer math;

private Integer physical;

private String term;

private Integer sum;

public void write(DataOutput out) throws IOException {

    out.writeUTF(id);

    out.writeUTF(name);

    out.writeInt(math);

    out.writeInt(physical);

    out.writeUTF(term);

    out.writeInt(sum);

}

public void readFields(DataInput in) throws IOException {

    id = in.readUTF();

    name = in.readUTF();

    math = in.readInt();

    physical = in.readInt();

    term = in.readUTF();

    sum = in.readInt();

}

//省去了 getters 和 setters

//省去了构造方法

public String toString() {

    return id + "\t" + name + "\t" + sum + "\t" + term;

}

}

```

## 步 2、开发 Mapper

Mapper 的开发，与前面类似，一行数据是一个学生的信息，所以输出的格式为 Text, Student。

```
package cn.weric.hadoop.partitioner;

import java.io.IOException;

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

public class StudMapper extends Mapper<LongWritable, Text, Text, Student> {

    protected void map(LongWritable key, Text value,

        Mapper<LongWritable, Text, Text, Student>.Context context)

        throws IOException, InterruptedException {

        // 根据空格进行字符串分割

        String[] words = value.toString().split("\\s");

        Student stud = new Student(words[0], words[1], //

            Integer.parseInt(words[2]), //

            Integer.parseInt(words[3]), //

            words[4]);

        context.write(new Text(stud.getName()), stud); // 输出

    }

}
```

## 步 3、开发 Reducer

Reducer 用于将成绩进行合并。

```
package cn.weric.hadoop.partitioner;

import java.io.IOException;

import org.apache.hadoop.io.NullWritable;
```

```

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

public class StudReducer extends Reducer<Text, Student, Student, NullWritable>{

    protected void reduce(Text text, Iterable<Student> it, Reducer<Text,
Student, Student, NullWritable>.Context context)

        throws IOException, InterruptedException {

    //实例化 Student

    Student student = new Student(null, text.toString(), 0, 0, null);

    student.setSum(0);

    it.forEach((Student stud)->{//遍历, 这是 JDK1.8 的语法

        student.setId(stud.getId());

        student.setSum(student.getSum()

            +stud.getMath()+stud.getPhysical());

        student.setTerm(stud.getTerm());

    });

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

    }

}

```

#### 步 4、开发 Partitioner

根据不的学期，返回一个 int。在后面的开发中，同时设置 Reducer 的数量为：2。

```

package cn.weric.hadoop.partitionner;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Partitioner;

public class StudentPartitioner extends Partitioner<Text, Student> {

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

        if (value.getTerm().equals("第一学期")) {

```

```

        return 0;

    } else { // 第二学期或其他

        return 1;

    }

}
}

```

### 步 5、开发 main 方法用于执行这个 Mapreduce 任务

此处要设置 Reducer 的数量。返回的数量值可根据 Partitioner 来设置，如果在 Partitioner 中返回的最大值为 1 则应该设置 Reducer 的数量为 2。

```

package cn.weric.hadoop.partitionner;

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

public class StudentMain {

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

        Job job = Job.getInstance(new Configuration());

        job.setJarByClass(StudentMain.class);

        job.setMapperClass(StudMapper.class);

        job.setMapOutputKeyClass(Text.class);

        job.setMapOutputValueClass(Student.class);

        //添加 Partitioner 类

        job.setPartitionerClass(StudentPartitioner.class);

        job.setReducerClass(StudReducer.class);
    }
}

```



```

        job.setOutputKeyClass(Student.class);

        job.setOutputValueClass(NullWritable.class);

        //设置 Reducer 的数量

        job.setNumReduceTasks(2);

        //设置输入输出目录

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

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

        job.waitForCompletion(true);

    }
}

```

## 步 6、打包运行

执行命令：

```
$ hadoop jar partitioner.jar /stud.txt /out006
```

输出的结果：

```

$ hdfs dfs -cat /out006/part-r-00000

001 Jack    177 第一学期

002 Mary    182 第一学期

$ hdfs dfs -cat /out006/part-r-00001

001 Jack    182 第二学期

002 Mary    180 第二学期

```

## 小结：

- ❖ Partitioner 分区编程，就是控制将不同的结果输出到指定的文件中。
- ❖ 开启一个 Reducer 就会拥有一个输出文件。通过设置 `job.setNumReduceTask` 可以设置开启 Reducer 的个数。此数值，必须要大于等于 Partitioner 分区返回的数量。
- ❖ 当启动的 Reducer 大于 Partitioner 返回的数量时，将会生成一些空的文件。

当 Reducer 数量小于 Partitioner 返回的数量时，将直接抛出异常 `IllegalPartition counter error`。

- ❖ 所以，在设置 Reducer 数量时，应该考虑 Partitioner 返回的分区个数。
- ❖ Mapper 在输出数据给 Reducer 时，根据不同的分区编号分发到不同的 Reducer 上去。每一个 Reducer 都会有一个编号。

## 4.5、自定义排序

在 Hadoop 开发中，通过 Mapper 输出给 Reducer 的数据已经根据 key 值进行了排序。即 `Mapper<Key1_input, Value1_input, Key2_out, Value2_out>` 其中默认根据 `key2_out` 进行排序。

所以，如果希望输出的数据可以实现排序，可以通过以下方式实现：

- 1: 开发 JavaBean 实现接口 `WritableComparable`，此类是 `Writable` 的子类。
- 2: 将 JavaBean 做为 Mapper 的 `key2` 输出给 reducer。
- 3: 实现 JavaBean 的 `compareTo` 方法，实现比较。

在开发时，可以开发多个 MapReduce 处理过程。形成一个连接使用 MapReducer 程序处理数据的过程。后面的 MapReduce 处理前面 MapReduce 输出的结果，直接达到最后所需要的数据。

现在假设第一次 MapReduce 输入的结果如下，现在需要将第一次输出的结果再次进行计算，按 sum，即第三列升序进行排序：

```
$ hdfs dfs -cat /out006/part-r-00000
001 Jack    177 第一学期
002 Mary    182 第一学期

$ hdfs dfs -cat /out006/part-r-00001
001 Jack    182 第二学期
002 Mary    180 第二学期
```

现在我们可以只开发一个 Mapper，然后直接读取数据将封装的数据输出到指定的目录下去即可。但，必须要把读取的数据封装到 JavaBean 中，其后将 JavaBean 当成 Mapper 的 key2 输出的数据，即可以实现排序。当然了，JavaBean 必须实现接口：WritableComparable。

### 步 1: 实现 WritableComparable 接口

```
public class Student implements WritableComparable<Student> {  
  
    public int compareTo(Student o) {  
  
        return this.sum - o.sum;//实现比较的方法  
  
    }  
  
    //其他代码略  
  
}
```

### 步 2: 开发一个 Mapper

将 Student 做为 key2 输出，由于数据是单一的，所以，直接使用 NullWritable 做为输出的 value2 即可。

```
package cn.weric.hadoop.partitionner;  
  
import java.io.IOException;  
  
import org.apache.hadoop.io.LongWritable;  
  
import org.apache.hadoop.io.NullWritable;  
  
import org.apache.hadoop.io.Text;  
  
import org.apache.hadoop.mapreduce.Mapper;  
  
public class StudentSortMapper extends Mapper<LongWritable, Text, Student,  
NullWritable> {  
  
    public void map(LongWritable key, Text value, Mapper<LongWritable, Text,  
Student, NullWritable>.Context context)  
  
        throws IOException, InterruptedException {  
  
        String[] str = value.toString().split("\\s+");  
  

```

```
Student student = new Student();

student.setId(str[0]);

student.setName(str[1]);

student.setSum(Integer.parseInt(str[2]));

student.setTerm(str[3]);

context.write(student, NullWritable.get());}}
```

### 步 3、开发 main 方法用于执行一个任务

由于只需要处理排序，所以，只需要一个 Mapper 即可。

```
package cn.weric.hadoop.partitioner;

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

public class StudentSortMain {

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

        Job job = Job.getInstance(new Configuration());

        job.setJarByClass(StudentSortMain.class);

        job.setMapperClass(StudentSortMapper.class);

        job.setMapOutputKeyClass(Student.class);

        job.setMapOutputValueClass(NullWritable.class);

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

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

        job.waitForCompletion(true);

    }}
```

#### 步 4、现在可以打包执行

将程序打包成 `studentsort.jar`。然后执行以下命令：

```
$ hadoop jar studentsort.jar /out006 /out002
```

现在可以查看输出的结果，可见已经实现了按总分排序了：

```
$ hdfs dfs -cat /out002/*

001      Jack      177      第一学期
002      Mary      180      第二学期
001      Jack      182      第二学期
002      Mary      182      第一学期
```

最后，在 Mapper 中，输出的数据 key2 默认会自动排序。如果希望根据自己的要求进行排序，则可以实现接口：`WritableComparable`，并实现 `compareTo` 方法。

如果 value2 没有数据，可以使用 `NullWritable` 替代。

## 4.6、Combiner 编程

Combiner 是在 Mapper 以后先对数据进行一个计算。然后再将数据发送到 Reducer。开发 Combiner 就是开发一个继承 Reducer 的类实现 Reducer 中相同的功能。Combiner 是可插拔的组件。但仅限于 Combiner 的业务与 Reducer 的业务相同。在开发中，可以通过调用 `job.setCombinerClass(..)` 添加 Combiner 类。

以下是过程，说明没有 Combiner 和有 Combiner 的两种不同的执行过程：

源数据：

```
Hello   Jack
Hello   Mary
Hello   Jack
Hello   Jack
```

没有 Combiner 的情况:

Mapper	Shuffle	Reducer
<Hello, 1>	<Hello, {1, 1, 1, 1}>	<Hello, 4>
<Jack, 1>	<Jack, {1, 1, 1}>	<Jack, 3>
<Hello, 1>	<Mary, {1} >	<Mary, 1>
<Mary, 1>		
<Hello, 1>		
<Jack, 1>		
<Hello, 1>		
<Jack, 1>		

有 Combiner 的情况:

Mapper	Combiner	Shuffle	Reducer
<Hello, 1>	<Hello, 4>	<Hello, {4}>	<Hello, 4>
<Jack, 1>	<Jack, 3>	<Jack, {3}>	<Jack, 3>
<Hello, 1>	<Mary, 1>	<Mary, {1}>	<Mary, 1>
<Mary, 1>			
<Hello, 1>			
<Jack, 1>			
<Hello, 1>			
<Jack, 1>			

通过上面的分析应该可以看到在 Combiner 中, 已经对数据进行了一次合并计算操作。这将减少在最后的 Reducer 中遍历处理的次数。

现在添加这个 Combiner, 在启动之前调用 `setCombinerClass(..)` 即可。

```
job.setCombinerClass(PhoneDataReducer.class); // 开始任务
job.waitForCompletion(true);
```

由于 Combiner 类, 就是 Reducer 类的子类, 所以, 此处就不再赘述。读者可以直

接将 Reducer 类做为 Combiner 设置到 job 中去即可。

## 4.7、Shuffle

Shuffle 是 MapReduce 的核心。Shuffle 的本义是洗牌、混洗，把一组有一定规则的数据尽量转换成一组无规则的数据，越随机越好。MapReduce 中的 Shuffle 更像是洗牌的逆过程，把一组无规则的数据尽量转换成一组具有一定规则的数据。

为什么 MapReduce 计算模型需要 Shuffle 过程？我们都知道 MapReduce 计算模型一般包括两个重要的阶段：Map 是映射，负责数据的过滤分发；Reduce 是规约，负责数据的计算归并。Reduce 的数据来源于 Map，Map 的输出即是 Reduce 的输入，Reduce 需要通过 Shuffle 来获取数据。

从 Map 输出到 Reduce 输入的整个过程可以广义地称为 Shuffle。Shuffle 横跨 Map 端和 Reduce 端，在 Map 端包括 Spill 过程，在 Reduce 端包括 copy 和 sort 过程，如图 4.7.1 所示：

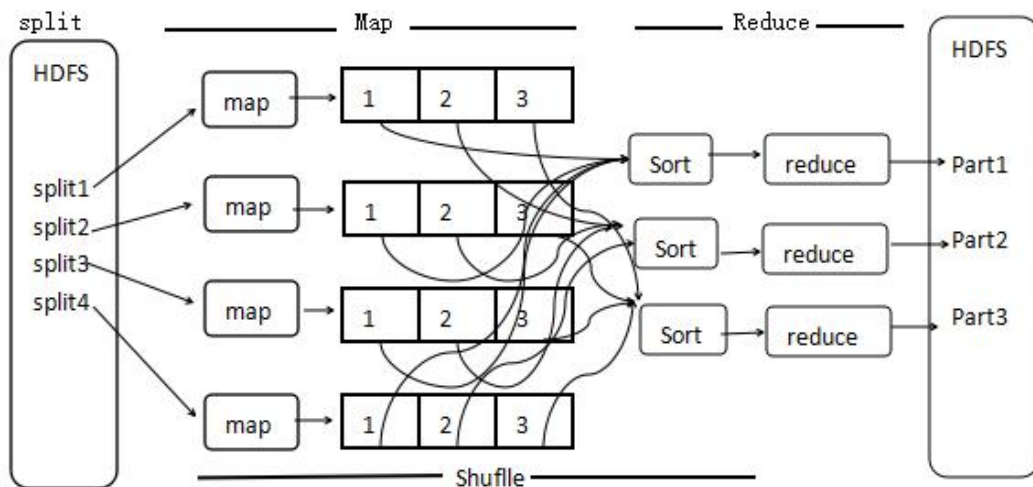


图 4.7.1

### Spill

Spill 过程包括输出、排序、溢写、合并等步骤，如图 4.7.2 所示：

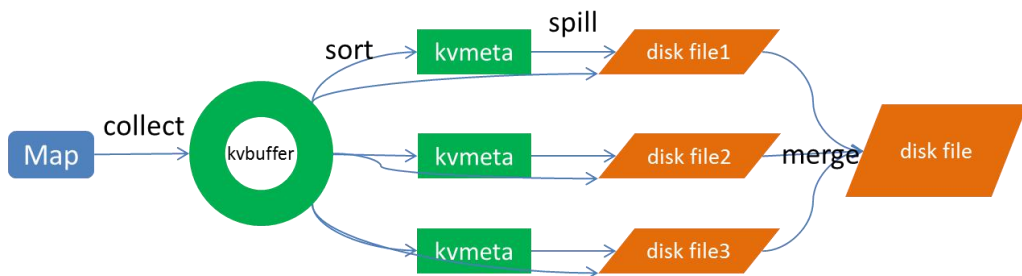


图 4.7.2

Spill 的第一个阶段 Collect，每个 Map 任务不断地以<key, value="">对的形式把数据输出到在内存中构造的一个环形数据结构中。使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。

这个数据结构其实就是个字节数组，叫 Kvbuffer，名如其义，但是这里面不光放置了<key, value="">数据，还放置了一些索引数据，给放置索引数据的区域起了一个 Kmeta 的别名，在 Kvbuffer 的一块区域上穿了一个 IntBuffer（字节序采用的是平台自身的字节序）的马甲。<key, value="">数据区域和索引数据区域在 Kvbuffer 中是相邻不重叠的两个区域，用一个分界点来划分两者，分界点不是亘古不变的，而是每次 Spill 之后都会更新一次。初始的分界点是 0，<key, value="">数据的存储方向是向上增长，索引数据的存储方向是向下增长。

Kvbuffer 的存放指针 bufindex 是一直闷着头地向上增长，比如 bufindex 初始值为 0，一个 Int 型的 key 写完之后，bufindex 增长为 4，一个 Int 型的 value 写完之后，bufindex 增长为 8。

索引是对<key, value="">在 kvbuffer 中的索引，是个四元组，包括：value 的起始位置、key 的起始位置、partition 值、value 的长度，占用四个 Int 长度，Kmeta 的存放指针 Kindex 每次都是向下跳四个“格子”，然后再向上一个格子一个格子地填充四元组的数据。比如 Kindex 初始位置是-4，当第一个<key, value="">写完之后，(Kindex+0)的位置存放 value 的起始位置、(Kindex+1)的位置存放 key 的起始位置、(Kindex+2)的位置存放 partition 的值、(Kindex+3)的位置存放 value 的长度，然



后 Kvindex 跳到-8 位置，等第二个<key, value="">和索引写完之后，Kvindex 跳到-32 位置。

Kvbuffer 的大小虽然可以通过参数设置，但是总共就那么小，<key, value="">和索引不断地增加，加着加着，Kvbuffer 总有不够用的那天，那怎么办？把数据从内存刷到磁盘上再接着往内存写数据，把 Kvbuffer 中的数据刷到磁盘上的过程就叫 Spill，多么明了的叫法，内存中的数据满了就自动地 spill 到具有更大空间的磁盘。

关于 Spill 触发的条件，也就是 Kvbuffer 用到什么程度开始 Spill，还是要讲究一下的。如果把 Kvbuffer 用得死死得，一点缝都不剩的时候再开始 Spill，那 Map 任务就需要等 Spill 完成腾出空间之后才能继续写数据；如果 Kvbuffer 只是满到一定程度，比如 80%的时候就开始 Spill，那在 Spill 的同时，Map 任务还能继续写数据，如果 Spill 够快，Map 可能都不需要为空闲空间而发愁。两利相衡取其大，一般选择后者。

Spill 这个重要的过程是由 Spill 线程承担，Spill 线程从 Map 任务接到“命令”之后就开始正式干活，干的活叫 SortAndSpill，原来不仅仅是 Spill，在 Spill 之前还有个颇具争议性的 Sort。

## Sort

Sort 先把 Kvbuffer 中的数据按照 partition 值和 key 两个关键字升序排序，移动的只是索引数据，排序结果是 Kvmeta 中数据按照 partition 为单位聚集在一起，同一 partition 内的按照 key 有序。

Spill 线程为这次 Spill 过程创建一个磁盘文件：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于“spill12.out”的文件。Spill 线程根据排过序的 Kvmeta 挨个 partition 的把<key, value="">数据吐到这个文件中，一个 partition 对应的数据吐完之后顺序地吐下个 partition，直到把所有的 partition 遍历完。一个 partition 在文件中对应的数据也叫段(segment)。

所有的 partition 对应的数据都放在这个文件里，虽然是顺序存放的，但是怎么直接知道某个 partition 在这个文件中存放的起始位置呢？强大的索引又出场了。有一个三元组记录某个 partition 对应的数据在这个文件中的索引：起始位置、原始数

据长度、压缩之后的数据长度，一个 partition 对应一个三元组。然后把这些索引信息存放在内存中，如果内存中放不下了，后续的索引信息就需要写到磁盘文件中了：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于“spill12.out.index”的文件，文件中不光存储了索引数据，还存储了 crc32 的校验数据。（spill12.out.index 不一定在磁盘上创建，如果内存（默认 1M 空间）中能放得下就放在内存中，即使在磁盘上创建了，和 spill12.out 文件也不一定在同一个目录下。）

每一次 Spill 过程就会最少生成一个 out 文件，有时还会生成 index 文件，Spill 的次数也烙印在文件名中。索引文件和数据文件的对应关系如图 4.7.3 所示：

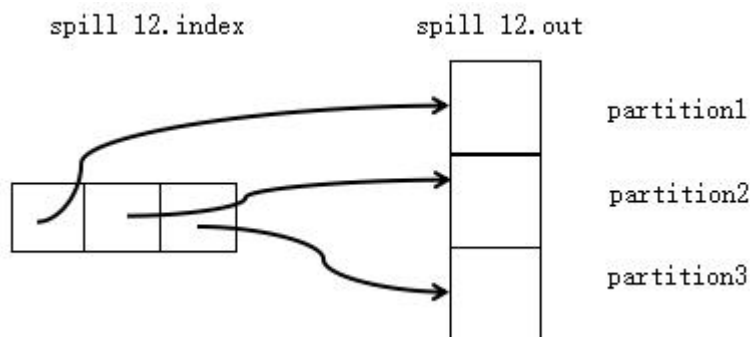


图 4.7.3

在 Spill 线程如火如荼的进行 SortAndSpill 工作的同时，Map 任务不会因此而停歇，而是一无既往地进行着数据输出。Map 还是把数据写到 kvbuffer 中，那问题就来了：<key, value="">只顾着闷头按照 bufindex 指针向上增长，kvmeta 只顾着按照 Kvindex 向下增长，是保持指针起始位置不变继续跑呢，还是另谋它路？如果保持指针起始位置不变，很快 bufindex 和 Kvindex 就碰头了，碰头之后再重新开始或者移动内存都比较麻烦，不可取。Map 取 kvbuffer 中剩余空间的中间位置，用这个位置设置为新的分界点，bufindex 指针移动到这个分界点，Kvindex 移动到这个分界点的-16 位置，然后两者就可以和谐地按照自己既定的轨迹放置数据了，当 Spill 完成，空间腾出之后，不需要做任何改动继续前进。

Map 任务总要把输出的数据写到磁盘上,即使输出数据量很小在内存中全部能装得下,在最后也会把数据刷到磁盘上。

## Merge

Map 任务如果输出数据量很大,可能会进行好几次 Spill, out 文件和 Index 文件会产生很多,分布在不同的磁盘上。最后把这些文件进行合并的 merge 过程闪亮登场。

Merge 过程怎么知道产生的 Spill 文件都在哪了呢?从所有的本地目录上扫描得到产生的 Spill 文件,然后把路径存储在一个数组里。Merge 过程又怎么知道 Spill 的索引信息呢?没错,也是从所有的本地目录上扫描得到 Index 文件,然后把索引信息存储在一个列表里。到这里,又遇到了一个值得纳闷的地方。在之前 Spill 过程中的时候为什么不直接把这些信息存储在内存中呢,何必又多了这步扫描的操作?特别是 Spill 的索引数据,之前当内存超限之后就把数据写到磁盘,现在又要从磁盘把这些数据读出来,还是需要装到更多的内存中。之所以多此一举,是因为这时 kvbuffer 这个内存大户已经不再使用可以回收,有内存空间来装这些数据了。然后为 merge 过程创建一个叫 file.out 的文件和一个叫 file.out.Index 的文件用来存储最终的输出和索引。一个 partition 一个 partition 的进行合并输出。对于某个 partition 来说,从索引列表中查询这个 partition 对应的所有索引信息,每个对应一个段插入到段列表中。也就是这个 partition 对应一个段列表,记录所有的 Spill 文件中对应的这个 partition 那段数据的文件名、起始位置、长度等等。

然后对这个 partition 对应的所有的 segment 进行合并,目标是合并成一个 segment。当这个 partition 对应很多个 segment 时,会分批地进行合并:先从 segment 列表中把第一批取出来,以 key 为关键字放置成最小堆,然后从最小堆中每次取出最小的<key, value="">输出到一个临时文件中,这样就把这一批段合并成一个临时的段,把它加回到 segment 列表中;再从 segment 列表中把第二批取出来合并输出到一个临时 segment,把其加入到列表中;这样往复执行,直到剩下的段是一批,输出到最终的文件中。

## Copy

Reduce 任务通过 HTTP 向各个 Map 任务拖取它所需要的数据。每个节点都会启动一个常驻的 HTTP server，其中一项服务就是响应 Reduce 拖取 Map 数据。当有 MapOutput 的 HTTP 请求过来的时候，HTTP server 就读取相应的 Map 输出文件中对应这个 Reduce 部分的数据通过网络流输出给 Reduce。

Reduce 任务拖取某个 Map 对应的数据，如果在内存中能放得下这次数据的话就直接把数据写到内存中。Reduce 要向每个 Map 去拖取数据，在内存中每个 Map 对应一块数据，当内存中存储的 Map 数据占用空间达到一定程度的时候，开始启动内存中 merge，把内存中的数据 merge 输出到磁盘上一个文件中。

如果在内存中不能放得下这个 Map 的数据的话，直接把 Map 数据写到磁盘上，在本地目录创建一个文件，从 HTTP 流中读取数据然后写到磁盘，使用的缓存区大小是 64K。拖一个 Map 数据过来就会创建一个文件，当文件数量达到一定阈值时，开始启动磁盘文件 merge，把这些文件合并输出到一个文件。

有些 Map 的数据较小是可以放在内存中的，有些 Map 的数据较大需要放在磁盘上，这样最后 Reduce 任务拖过来的数据有些放在内存中了有些放在磁盘上，最后会对这些来一个全局合并。

## Merge Sort

这里使用的 Merge 和 Map 端使用的 Merge 过程一样。Map 的输出数据已经是有序的，Merge 进行一次合并排序，所谓 Reduce 端的 sort 过程就是这个合并的过程。一般 Reduce 是一边 copy 一边 sort，即 copy 和 sort 两个阶段是重叠而不是完全分开的。

Reduce 端的 Shuffle 过程至此结束。

## 4.8、小结

- ❖ MapReduce 的过程被显式分为两部分 Map 和 Reduce。
- ❖ 在 MapReduce 中被传递的类必须要实现 Hadoop 的序列化接口 Writable。

- ❖ Map 中输出的 key 可以实现排序，此类必须要实现接口 WritableComparable 接口。
- ❖ 多个 MapReduce 可以分别打成不同的 Jar 包执行。这样后面的 mapReduce 处理的是前面 MapReduce 输出的数据。
- ❖ Partitioner 编程用于将不同规则的数据输出到指定编号的 Reduce 上去。
- ❖ Combiner 类似于一个前置的 Reduce，用于在将数据输出到 Reduce 之前进行一次数据的合并操作
- ❖ MapReduce 开发过程
  - 1) 研究业务需要输出的格式。
  - 2) 自定义一个类继承 Mapper 类。并指定输入输出的格式。
  - 3) 重这与 map 方法实现具体的业务逻辑。注意使用 context
  - 4) 自定义一个类继承 Reduce 类，并指定输入输出格式。
  - 5) 重写 reduce 方法。实现自己的业务代码。注意使用 context
  - 6) 开发一个 main 方法，通过 Job 对象进行组装。
  - 7) 打成 jar 包，指定主类，发到 Linux 上去通过 `hadoop jar` 来启动 mapreduce。
- ❖ MapReduce 的执行流程
  - 1) Client 通过 RPC 将请求提交给 ResourceManager。
  - 2) ResourceManager 在接收到请求以后返回一个 jobID 给 Client.
  - 3) Client 将 Jar 包上传到 HDFS(默认 HDFS 保存 10 份)，且程序执行完成以后删除 jar 文件。
  - 4) Client 通知 ResourceManager 保存数据的描述信息。
  - 5) ResourceManager 通过公平调度开启任务。将任务放到任务调度队列。
  - 6) NodeManager 通过心跳机制向 ResourceManager 领取任务。
  - 7) NodeManager 向 hdfs 领取所需要的 jar 包。开始执行任务。

## 第 5 章 自定义 InputFormat 类

### 内容简介

#### ➤ 自定义 InputFormat

在前面代码中，分析和处理的都是\*.txt 类型的文档。但在实际的工作中，可能还会解析其他文件格式，如 Excel 等、RDBMS 中的数据、PDF 等。

默认的情况下 Hadoop 解析 txt 文件返回的数据格式为：<LongWritable,Text>即前一个值为字节偏移量，后面的 Text 为每一次读取的一行数据。

现在我们先来练习开发自定义 InputFormat 类让 LongWritable 为行号而不是字节的偏移量，此时，就应该通过继承 InputFormat 实现自定义文件输入流。

### 5.1、自定义文件输入流

自定义文件输入流的关键就是继承 FileInputStream，然后重写 createRecordReader 方法。定义一个类，返回一个自定义的 RecordReader 对象，在 RecordReader 对象中，读取分片文件的数据，将每一行放到一个 List 中，List 的 size 就是行数。以下是自定义 InputFormat 类的源代码：

```
package cn.hadoop.wordcount;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.fs.FileSystem;
```

```

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

/** 自定义 InputFormat 类*/

public class LineInputFormat extends FileInputFormat<LongWritable, Text> {

    /** 必须要重写的 createRecordReader 方法*/

    public RecordReader<LongWritable, Text> createRecordReader(InputSplit
split, TaskAttemptContext context)

        throws IOException, InterruptedException {

        return new LineReader();

    }

    /** 定义 LineReader，读取文件分片数据，将一片数据放到 List 集合中，其后再
逐一返回*/

    public class LineReader extends RecordReader<LongWritable, Text>{

        private LongWritable key = new LongWritable(-1);

        private Text value = new Text();

        private InputStream in;

        private float size;//总大小

        //读取一行一行的数据放到集合中

        private List<String> lines = new ArrayList<>();

        //根据分片信息读取数据

        public void initialize(InputSplit split, TaskAttemptContext context)

```

```

throws IOException, InterruptedException {

    //将文件分片信息转成文件分片对象

    FileSplit fileSplit = (FileSplit) split;

    FileSystem fs = FileSystem.get(context.getConfiguration());

    //其后获取这一个文件片的输入流

    in = fs.open(fileSplit.getPath());

    //根据上面的输入流读取数据

    BufferedReader br = new BufferedReader(new

InputStreamReader(in));

    String line = null;

    while((line=br.readLine())!=null){

        //将读取的所有行，都放到集合中去

        lines.add(line);

    }

    //获取总大小

    size = lines.size();

    br.close();

}

/**用于判断是否还有下一行数据，如果有则返回 true 否则返回 false*/

public boolean nextKeyValue() throws IOException, InterruptedException

{

    int index = (int)key.get()+1;

    if(index<lines.size()){

        //设置成员变量的 key 和设置 value

        value.set(lines.get(index));

        key.set(index);

```



```

        return true;//如果有数据
    }

    return false;
}

//返回当前 key 值
public LongWritable getCurrentKey() throws IOException,
        InterruptedException {
    return key;
}

//返回当前 value 值
public Text getCurrentValue() throws
IOException, InterruptedException {
    return value;
}

@Override//进度
public float getProgress() throws IOException, InterruptedException {
    return (float)(key.get()/size);
}

public void close() throws IOException {
    if(in!=null){
        in.close();
    }
}
}}}

```

现在，在 main 方法中设置自定义文件输入流对象

```

job.setInputFormatClass(LineInputFormat.class); // 提交作业

job.waitForCompletion(true);// 参数是指是否显示进度

```

可选的修改 Mapper 类。此时，要根据 InputFormat 输出的数据格式来设置 Mapper 输入的数据格式即 Key1, Value1。

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text,
LongWritable> {

    private Text key =new Text();

    /**重写 map 方法*/

    public void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
LongWritable>.Context context)

        throws IOException, InterruptedException {

        long index = key.get();//现在直接按行，一行输出一个数据，显示

        if(index<10){

            this.key.set("0"+index+": "+value);

        }else{

            this.key.set(index+": "+value);

        }

        context.write(this.key,new LongWritable(1));

    }}
}
```

如果希望查看运行的结果也可以只定义一个 Mapper，然后打包运行来查看结果。

```
$ hdfs dfs -cat /wcout3/part-r-00000

00:Hello Jack 1

01:Hello Mary 1

02:Hello jack 1
```

## 5.2、在 Excel 中解析通话记录统计

现在假设数据存在于 Excel 文件中。具体的实现方案为：1、添加 POI 解析 Excel

文件。2、通过实现 `FileInputFormat` 自定义文件输入流。3、可以通过文件分片将不同的输出输出到不同的目录下。4、最后将需要的包一并打包到 `!/some.jar/lib` 目录下。最后的项目结构如图 5.2.1 所示：

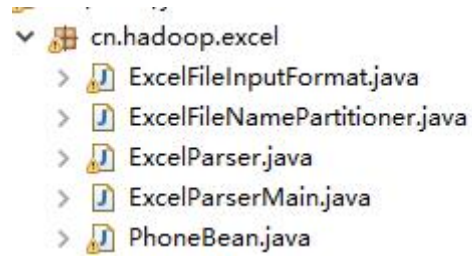


图 5.2.1

Excel 的文件格式如图 5.2.2 所示：

382	语音电话	2017-01-03 13:39:11	2分47秒	主叫	15866789399
383	语音电话	2017-01-03 12:55:23	25秒	被叫	18754170558
384	语音电话	2017-01-03 11:46:34	12秒	被叫	18678397488
385	语音电话	2017-01-03 10:54:39	20秒	被叫	15098961421
386	语音电话	2017-01-03 10:43:01	52秒	主叫	053182291779
387	语音电话	2017-01-03 10:40:23	1分43秒	主叫	13065051387

图 5.2.2

上图的通讯记录(读者可以自行去移动或是联通的网上营业厅下载通讯记录的 Excel 文件)，为联通电话的通讯记录。读者可以自己通过联通或是移动的官方网站上下载自己手机的通话记录详单进行分析即可。

**步 1：添加依赖**

```
<dependency>

    <groupId>org.apache.poi</groupId>

    <artifactId>poi</artifactId>

    <version>3.16</version>

</dependency>
```

**步 2、开发 JavaBean**

通过实现接口 `WritableComparable` 可以实现序列化和排序的功能。

```
package cn.hadoop.excel;
```

```

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

public class PhoneBean implements WritableComparable<PhoneBean> {

    private String tel;// 电话号码

    private String type;// 类型，主叫或是被叫

    private Long seconds;

    private String fileName;//根据文件名进行分组

    //其他方法略去....

}

```

### 步 3、开发文件解析对象

这个解析类，只是一个独立的类，主要用于读取 excel 文件，并从中读取数据。

```

package cn.hadoop.excel;

import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.Row;

public class ExcelParser {

    /** 接收 InputStream 返回解析的结果*/

    public static List<PhoneBean> parse(InputStream in,String fileName) {

        List<PhoneBean> list = new ArrayList<PhoneBean>();

```

```

HSSFWorkbook book = null;

try {

    book = new HSSFWorkbook(in);

    HSSFSheet sheet = book.getSheetAt(0);

    Iterator<Row> it = sheet.iterator();

    while (it.hasNext()) {

        // 如果还有下一个

        Row row = it.next();

        String tel = row.getCell(5).getStringCellValue();

        if (!tel.matches("\\d+")) {

            continue;// 如果不是数字则下一个

        }

        String times = row.getCell(3).getStringCellValue();

        Long time = parse(times);// 转成秒

        String type = row.getCell(4).getStringCellValue();

        PhoneBean bean = new PhoneBean(tel, type, time);

        bean.setFileName(fileName);

        list.add(bean);

    }

} catch (Exception e) {

    throw new RuntimeException(e);

} finally {

    try {

        book.close();

    } catch (IOException e) {

        e.printStackTrace();

    }

}

```

```

    }

    }

    return list;
}

/** 将： 2 分 40 秒转成 160 秒<br>*/
public static Long parse(String time) {

    Long times = 0L;

    if (time.contains("分")) {

        String min = time.substring(0, time.indexOf("分"));

        times += Long.parseLong(min) * 60;

        time = time.substring(time.lastIndexOf("分") + 1);

    }

    String ss = time.substring(0, time.length() - 1);

    times += Long.parseLong(ss);

    return times;
}

/**将秒转成分种 */
public static String parse(Long times){

    String str = "";

    if(times>(60*60)){

        Long _times = times/(60*60);

        str=_times+"时";

        times=times%(60*60);

    }

    if(times>60){

        Long _times=times/60;

```

```

        str=str+_times+"分";

        times=times%60;

    }

    str=str+times+"秒";

    return str;

}
}

```

#### 步 4、开发 ExcelFileInputFormat

这个类输出的数据，就是 Mapper 接收的数据类型。注意观察 FileInputFormat<PhoneBean, NullWritable>

```

/**用于从 Excel 中解析数据<br>*/
public class ExcelFileInputFormat extends FileInputFormat<PhoneBean,
NullWritable> {

    /** 读取文件*/

    public RecordReader<PhoneBean, NullWritable>
createRecordReader(InputSplit split, TaskAttemptContext context)

        throws IOException, InterruptedException {

        return new ExcelRecordReader();

    }

    class ExcelRecordReader extends RecordReader<PhoneBean, NullWritable> {

        private List<PhoneBean> list;

        private int size;

        private int index = -1;

        private PhoneBean bean = new PhoneBean();

        private InputStream in;

        /** 分片读取数据*/

```

```

        public void initialize(InputSplit split, TaskAttemptContext context)
throws IOException, InterruptedException {

            FileSplit fs = (FileSplit) split;

            FileSystem fs2 =

                FileSystem.get(context.getConfiguration());

            in = fs2.open(fs.getPath());

            String fileName =

                fs.getPath().toString();// hdfs://server:port/a.txt

            fileName =

                fileName.substring(fileName.lastIndexOf("/") + 1);

            list = ExcelParser.parse(in, fileName);// 解析 xml 文件

            size = list.size();

        }

        public boolean nextKeyValue() throws IOException,

                                InterruptedException {

            index++;

            if (index < list.size()) {

                return true;

            }

            return false;

        }

        public PhoneBean getCurrentKey() throws IOException,

                                InterruptedException {

            return list.get(index);

        }

        public NullWritable getCurrentValue() throws IOException,

```



```

InterruptedException {

    return NullWritable.get();

}

public float getProgress() throws IOException,

                                InterruptedException {

    return ((float) index / size);

}

public void close() throws IOException {

    if (in != null) {

        in.close();

    }

}

}}

```

### 步 5、开发主类及 Mapper 和 Reducer

这儿开发的 Mapper 和 Reducer 与之前的一样,只是 Mapper 中接收到的 Key 和 value 为 InputFormat 输出的 Key 和 value, 请稍加注意。

```

package cn.hadoop.excel;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

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

```

```

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

public class ExcelParserMain {

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

        Configuration config = new Configuration();

        Job job = Job.getInstance(config);

        job.setJarByClass(ExcelParserMain.class);

        job.setMapperClass(ExcelMapper.class);

        job.setMapOutputKeyClass(Text.class);

        job.setMapOutputValueClass(PhoneBean.class);

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

        job.setReducerClass(ExcelReducer.class);

        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(Text.class);

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

        job.setInputFormatClass(ExcelFileInputFormat.class);

        job.setPartitionerClass(ExcelFileNamePartitioner.class);

        job.setNumReduceTasks(3);

        job.waitForCompletion(true);

    }

    public static class ExcelMapper extends Mapper<PhoneBean,

        NullWritable, Text, PhoneBean> {

        private Text key2 = new Text();

        private LongWritable times = new LongWritable();

        public void map(PhoneBean key, NullWritable value,

            Mapper<PhoneBean, NullWritable, Text, PhoneBean>.Context context)

            throws IOException, InterruptedException {

```

```

        String key2 = key.getTel() + ":" + key.getType();

        this.key2.set(key2);

        times.set(key.getSeconds());

        context.write(this.key2, key); // 组成: xxxxxx 主叫
    }

}

public static class ExcelReducer extends Reducer<Text, PhoneBean, Text, Text> {

    private Text value4 = new Text();

    protected void reduce(Text text, Iterable<PhoneBean> it,

        Reducer<Text, PhoneBean, Text, Text>.Context context)

        throws IOException, InterruptedException {

        Long sum = new Long(0);

        for (PhoneBean bean : it) {

            sum+=bean.getSeconds();

        }

        //将秒转换

        String str = ExcelParser.parse(sum);

        value4.set(str);

        context.write(text, value4);

    }

}
}
}

```

## 步 6、打包运行

由于 poi.jar 并不是 hadoop 的包，所以，可以先将项目打成 jar 包，然后在包里面创建一个 lib 目录，并将 poi.jar 放到这个目录里面即可。

目录结构如下所示：

```
phonedata.jar
  cn
  lib
    poi.jar
  MATE-INFO
```

现在可以将 some.xls 即获取到的 excel 文件上传到 hdfs 然后执行以下命令，执行 mapreduce 任务：

```
$ hadoop jar phonedata.jar /a.xls /out001
```

查看输出的结果：

```
$ hdfs dfs -cat /out001/*

01160789841:被叫    1 分 37 秒

053112345:被叫    1 分 5 秒
```

### 5.3、小结

❖ 到目前为止，我们已经学会了运行一个单一节点的 Hadoop。以下是 hadoop 在伪分布式上的一些配置，供大家参考。

软件或配置	说明
修改主机名称	#hostnamectl set-hostname weric201
配置本地 DNS	/etc/hosts 192.168.56.201    weric201
禁止使用防火墙	#yum install firewalld  #firewall-cmd --state  #systemctl stop firewalld.service  #systemctl disable firewalld.service

SSH 免密码登录 ~/.ssh/	# ssh-keygen -t rsa -P "" #cat id_rsa.pub >> authorized_keys
core-site.xml	fs.defaultFS=hdfs://weric:9000 hadoop.tmp.dir= ...
hdfs-site.xml	dfs.replication=1 //副本数量为 1
mapred-site.xml	mapreduce.framework.name=yarn
yarn-site.xml	yarn.nodemanager.aux-services=mapreduce_shuffle yarn.resourcemanager.hostname=ip:port
格式化 hadoop 的文件系统	hdfs namenode -format 或 hadoop namenode -format
开发 MapReduce	extends Mapper<Key1, Value1, Key2, Value2> extends Reducer<Key3, Value3, Key4, Value4> 然后使用 job 来添加 Mapper 和 Reducer
开发 Partitioner - 分区管理	extends Partitioner<Key2, Value2>
开发 Combiner	Combiner 就是一个 Reducer，用于在 Map 和 Reduce 之前， 进行数据合并
自定义 InputFormat	extends FileInputFormat<Key1, Value1> 注册自定义输入类
自定义序列化类	extends Writable 只是序列化 extends WritableComparable 排序的
查看 hdfs 保存的文件信息	\$hadoop_tmp_dir/dfs/data 目录下，128M 为一个文件块 大小。

# 第 6 章 Hadoop 集群配置

## 内容简介

➤ hadoop 集群配置

在 hadoop 的集群中，有一个 NameNode, 一个 ResourceManager。（在高可靠的集群环境中，可以拥有两个 NameNode 和两个 ResourceManager，这将在后面的章节中讲解）。由于 NameNode 和 ResourceManager 是两个主要的服务，建议将它们部署到不同的服务器上。

非高可靠集群可用于快速学习 Hadoop 的集群方式，在增强了解 Hadoop 集群运行的基本原理上非常有用。

以下是将分步骤为大家详解如何搭建 hadoop 的非高可靠集群。

## 6.1、配置 hadoop 集群

### 1、配置规划

所有主机，必须要配置静态的 ip 地址、设置主机名。并通过 hosts 文件修改主机名和 ip 的对应关系，即本地 DNS 解析。

以下配置三台主机的集群，以下是配置表：

ip/主机名	安装的软件	进程
192.168.56.11/weric11	JDK	NameNode
	Hadoop	SecondaryNameNode
		DataNode
		NodeManager
192.168.56.12/weric12	JDK	ResourceManager
	Hadoop	DataNode

		NodeManager
192.168.56.13/wer1c13	JDK	DataNode
	Hadoop	NodeManager

2、安装系统

在 VirtualBox 上安装三个 CentOS7 操作系统。并修改名称名称，以便于记忆。配置好以后的效果如图 6.1.1 所示：



图 6.1.1

3、配置系统

三台服务器分别安装 JDK、配置 JDK 环境变量、配置静态 IP 地址、设置主机名、配置 hosts 文件以及配置 ssh 免密码登录。

配置主机名，在三台机器上，分别配置 weric11、weric12、weric13：

```
$sudo hostnamectl set-hostname weric11
```

设置静态 ip 地址，本机的 hostonly 网卡为 enp0s8，请根据你的具体情况进行修改，如果没有这个网卡的信息，请在/etc/sysconfig/network-scripts/目录下创建这个文本文件即可：（分别设置三台机器）

```
$ sudo vim ifcfg-enp0s8

BOOTPROTO="static"

NAME="enp0s8"

IPADDR="192.168.56.11"
```

重新启动网卡。重新启动网卡以后，如果用 ssh 远程连接，则必须要重新登录才

可以继承使用：

```
$sudo systemctl restart network.service
```

配置 hosts 文件：

```
$sudo vim /etc/hosts
```

```
192.168.56.11      weric11
```

```
192.168.56.12      weric12
```

```
192.168.56.13      weric13
```

安装 JDK，请参考前面的章节。

配置 ssh 免密码登录，配置原则是 NameNode 可以免密码登录所有的 DataNode 节点，ResourceManager 可以免密码登录所有的 DataNode 节点。每一个 DataNode 应该可以免密码登录自己：

在 weric11 上执行

```
$ ssh-keygen -t rsa
```

使用 ssh-copy-id 将公钥添加到 weric11,weric12,weric13

```
$ ssh-copy-id weric12
```

```
$ ssh-copy-id weric13
```

```
$ ssh-copy-id weric11
```

在 weric12 上执行

```
$ ssh-keygen -t rsa
```

使用 ssh-copy-id 将公钥添加到 weric11,weric12,weric13

```
$ ssh-copy-id weric12
```

```
$ ssh-copy-id weric13
```

```
$ ssh-copy-id weric11
```

在 weric13 上执行

```
$ ssh-keygen -t rsa
```

使用 ssh-copy-id 将公钥添加到 weric13，即自己



```
$ ssh-copy-id weric13
```

关闭防火墙，如果你的 CentOS 7 安装了防火墙，一定要记得关闭它。检查防火墙是否关闭，可以执行以下命令：

```
$ sudo systemctl | grep firewall
firewalld.service
loaded active running    firewalld - dynamic firewall daemon
```

上面显示，本机已经安装了防火墙。如果已经安装了防火墙，现在执行以下命令，将三台主机的防火墙全部关闭，并且禁用：

```
sudo systemctl stop firewalld.service
sudo systemctl disable firewalld.service
```

#### 4、安装和配置 hadoop

解压 Hadoop。通过 xftp 将 hadoop 的 gz 压缩包上传到 Linux，然后使用 root 用户将 hadoop 解压到/weric 目录下(目前可以自己选择，便于控制和记忆即可，尽量不要在某个用户目录下)。由于是 root 用户创建的目录，所以在创建以后，应该使用 chown 修改所属的用户和组，以便于当前用户的操作：

```
$ sudo mkdir /weric
$ sudo chown wangjian:wangjian /weric
```

现在将 hadoop 解压到这个目录：

```
$ tar -zxvf ~/hadoop-2.8.0.tar.gz -C /weric
```

配置环境变量，通过在/etc/profile.d 目录下，创建 hadoop.sh 文件，可以配置 hadoop 的环境变量。

```
$ sudo vim /etc/profile.d/hadoop.sh
```

在 hadoop.sh 文件中添加以下内容：

```
#!/bin/sh

export HADOOP_HOME=/weric/hadoop-2.8.0

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

环境变量生效

```
$ source /etc/profile
```

测试 hadoop 环境变量是否配置成功，查看 hadoop 的版本

```
$hadoop version
```

现在开始配置 hadoop。一共 6 个配置文件，包含：hadoop-env.sh、core-site.xml、hdfs-site.xml、mapred-site.xml、yarn-site.xml 和 slaves 文件。所有的这些配置文件，都在 \$HADOOP\_HOME/etc/hadoop 目录下。

配置 hadoop-env.sh 文件，在里面添加 JAVA\_HOME 的环境变量

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_131
```

配置 core-site.xml 文件，添加以下内容，其中 fs.defaultFS 用于配置 NameNode 的主机地址。hadoop.tmp.dir 用于指定 hadoop 文件系统保存的目录。

```
<configuration>

    <property>

        <name>fs.defaultFS</name>

        <value>hdfs://weric11:9000</value>

    </property>

    <property>

        <name>hadoop.tmp.dir</name>

        <value>/weric/hadoop_tmp_dir</value>

    </property>

</configuration>
```

配置 hdfs-site.xml 文件，添加以下内容。dfs.replication 用于配置 DataNode 节点的个数，即文件保存的副本数量。

```
<configuration>

    <property>

        <name>dfs.replication</name>
```

```

        <value>3</value>

    </property>

    <property>

        <name>dfs.namenode.secondary.http-address</name>

        <value>weric11:50090</value>

    </property>

</configuration>

```

配置 mapred-site.xml 文件, 由于在 \$HADOOP\_HOME/etc/hadoop 目录下, 并没有这个文件, 可以将 mapred-site.xml.template 文件, 修改得到 mapred-site.xml 文件, 并添加以下配置

```

<configuration>

    <property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

    </property>

</configuration>

```

配置 yarn-site.xml, 添加以下内容

```

<configuration>

    <property>

        <name>yarn.nodemanager.aux-services</name>

        <value>mapreduce_shuffle</value>

    </property>

    <property>

        <name>yarn.resourcemanager.address</name>

        <value>weric12:8032</value>

    </property>

```

```

    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>weric12:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>weric12:8031</value>
    </property>
</configuration>

```

配置 slaves 文件，此文件中，保存了 DataNode 节点的主机名称。一行一个。

```

weric11
weric12
weric13

```

将配置好的 hadoop 通过 scp 拷贝到其他两台主机上去，同时还要 copy 环境变量文件 hadoop.sh。在 copy 之前，可以删除 \$HADOOP\_HOME//share/docs 目录，这里面放的都是文档，网络 copy 为会有慢。copy 之前，请确保其他的主机上，已经创建了 /weric 目录，如果没有请先创建，并设置用户和组为 wangjian:wangjian。

```

$ scp -r hadoop-2.8.0/ weric12:/weric/
$ scp -r hadoop-2.8.0/ weric13:/weric/

```

以下是 copy 环境变量配置文件

```

$ scp /etc/profile.d/hadoop.sh root@weric12:/etc/profile.d/
$ scp /etc/profile.d/hadoop.sh root@weric13:/etc/profile.d/

```

## 5、格式化 hdfs

在 weric11 上执行格式化 hdfs 的命令

```
hdfs namenode -format
```

在格式化完成以后，输出以下语句，说明格式化成功，注意 successfully 关键字：

```
/weric/hadoop_tmp_dir/dfs/name has been successfully formatted
```

## 6、启动

在 weric 上执行

```
$ start-dfs.sh
```

在 weric12 机器上启动 resourcemanager，执行

```
$ start-yarn.sh
```

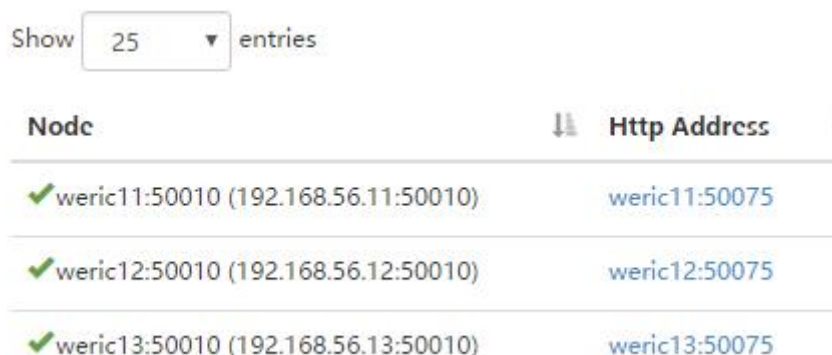
在启动完成以后，分别检查 weric11、weric12、weric13 上的服务，是否与之前设计的服务数量和名称相同。如果相同，hadoop 集群配置成功。

查看网络地址

```
http://192.168.56.11:50070
```

```
http://192.168.56.12:8088
```

通过 <http://192.168.56.11:50070> 地址，你将看到有三个 DataNode 节点，如图 6.1.2 所示，地址：<http://192.168.56.11:50070/dfshealth.html#tab-datanode>。



The screenshot shows a web interface for monitoring Hadoop DFS health. At the top, there is a 'Show' button followed by a dropdown menu set to '25' and the text 'entries'. Below this is a table with two columns: 'Node' and 'Http Address'. The table contains three rows, each representing a DataNode. Each row starts with a green checkmark icon, followed by the node's name and IP address in parentheses, and then the HTTP address for that node.

Node	Http Address
✓weric11:50010 (192.168.56.11:50010)	<a href="#">weric11:50075</a>
✓weric12:50010 (192.168.56.12:50010)	<a href="#">weric12:50075</a>
✓weric13:50010 (192.168.56.13:50010)	<a href="#">weric13:50075</a>

图 6.1.2

通过 <http://192.168.56.12:8088> 你将看到也有三个 NodeManager 节点，如图 6.1.3 所示，地址：<http://192.168.56.12:8088>。

Node State	Node Address	Node HTTP Address
RUNNING	weric12:37075	<a href="#">weric12:8042</a>
RUNNING	weric13:33815	<a href="#">weric13:8042</a>
RUNNING	weric11:36008	<a href="#">weric11:8042</a>

图 6.1.3

现在，你就可以向以前的一样使用 hdfs 命令了，也可以使用 java 代码操作 hdfs 或是发布一个 mapreduce 的任务。

### 7、现在建议测试一下 mapreduce 的 wordcount 程序

在任意的主机上测试，都是可以的。

```
$ hadoop jar
hadoop-2.8.0/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.8.0.jar
wordcount /word.txt /out001
```

## 6.2、扩展集群节点

Hadoop 集群在配置完成以后，也可以进行动态的扩展或是删除。甚至可以先临时禁用某些 DataNode 节点。本节，将为大家详解，如何在现有 hadoop 集群中添加新的 DataNode 节点，或是删除一个、禁用一个 DataNode 节点。具体步骤如下：

1、配置相同的 Linux 环境，包含已经安装好了 JDK、关闭防火墙（非常重要）及从 NameNode 节点到新环境的 SSH 免密码登录，最后，建议在所有主机的 hosts 文件中配置主机名与 ip 地址的对应关系。

2、建议在相同的目录下，已经安装了相同配置的 Hadoop 环境。可以使用 scp 将 NameNode 上的 Hadoop 拷贝到新的环境上。在 copy 之前，如果有必须请修改原配置

的副本数量，即 `dfs.replication` 的值，如原来在伪分布式模式下，`dfs.replication` 的值为 1，增加了新的 `DataNode` 节点以后，可以将 `dfs.replication` 的值修改为 2。

```
<property>
    <name>dfs.replication</name>
    <value>2</value>
</property>
```

3、拷贝 `hdfs` 目录，即 `hadoop.tmp.dir` 所配置的目录拷贝到新环境对应的目录下。由于新的环境将启动一个独立的 `DataNode`，所以，必须要独立的节点 `id`，在拷贝了 `hadoop.tmp.dir` 目录以后，应该适当修改文件 `VERSION` 文件中的内容，其中 `storageID` 任意修改一个值保持与其他 `DataNode` 的值不相同，`datanodeUuid` 的值任意修改一个值，确保与其他 `DataNode` 的 `id` 不相同。

```
vim hadoop_tmp_dir/dfs/data/current/VERSION

storageID=DS-c5404dd7-47e4-4b27-902e-193555089192

clusterID=CID-c4471577-06a4-479e-b0f7-9aea458847e3

cTime=0

datanodeUuid=e69a45e4-5567-4675-9b1b-0f5593fd2f02

storageType=DATA_NODE

layoutVersion=-57
```

4、在修改完成以后，现在就可以启动新增加节点上的 `DataNode` 和 `NodeManager` 了。启动命令：

```
$ hadoop-daemon.sh start datanode

$ yarn-daemon.sh start nodemanager
```

5、现在在 `NameNode` 节点上刷新 `DataNode` 执行以下命令

```
$ hdfs dfsadmin -refreshNodes

$ start-balancer.sh
```

在配置完成以后，通过 `ip:50070` 检查 `DataNode` 的信息，即可以显示新增加的

DataNode 节点。或是执行 `$ hdfs dfsadmin -report` 显示 hdfs 的报告信息。如果显示节点数量中已经包含了新增加的 dataNode 即为扩展成功。

### 禁用某个 DataNode 节点

也可以在现在的集群中删除或是禁用某些 DataNode 节点。禁用某些 DataNode 节点，可以在 `hdfs-site.xml` 先相应的减少 `dfs.replication` 的副本数量，建议配置的副本数量应该小于或是等于 DataNode 节点的数量。然后在 `hdfs-site.xml` 中添加以下配置：

```
<property>
  <!--配置对哪些节点排除-->
  <name>dfs.hosts.exclude</name>
  <value>/weric/hadoop-2.8.0/etc/hadoop/excludes</value>
</property>
```

现在，在 `excludes` 文件中，配置某个 DataNode 节点的 ip 地址或是名称。

```
vim excludes

weric203
```

现在，再通过 `ip:50070` 查看所有 DataNode 某个 DataNode 已经显示为 `dead`。

## 6.2、小结

- ❖ Hadoop 集群需要先安装 JDK，并配置 JAVA 环境变量。
- ❖ 共修改 hadoop 的 6 个配置文件。
- ❖ 这种集群的问题是：一旦 NameNode 宕机，整个集群即不能运行，一旦 ResourceManager 宕机，整个 Mapreduce 将不能运行。
- ❖ 扩展 Hadoop 集群，添加新的 DataNode。



# 第 7 章 Zookeeper 分布式协调技术

## 内容简介

- zookeeper 简介
- zookeeper 快速安装及基本命令
- zookeeper 分布式安装
- 使用 zookeeper 实现 hadoop 的 (HA) 高可靠集群

## 7.1、zookeeper 简介

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等。Zookeeper 是 hadoop 的一个子项目。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper 的目的就在于此。

目前，在分布式协调技术方面做得比较好的就是 Google 的 Chubby 还有 Apache 的 ZooKeeper 他们都是分布式锁的实现者。有人会问 既然有了 Chubby 为什么还要弄一个 ZooKeeper，难道 Chubby 做得不够好吗？不是这样的，主要是 Chubby 是非开源的，Google 自家用。后来雅虎模仿 Chubby 开发出了 ZooKeeper，也实现了类似的分布式锁的功能，并且将 ZooKeeper 作为一种开源的程序捐献给了 Apache，那么这样就可以使用 ZooKeeper 所提供锁服务。而且在分布式领域久经考验，它的可靠性，可用性都是经过理论和实践的验证的。所以我们 在构建一些分布式系统的时候，就可以以这类系统为起点来构建我们的系统，这将节省不少成本，而且 bug 也将更少。

# Google Chubby



## Apache ZooKeeper

图 7.1.1

Zookeeper 中的角色主要有以下三类：

角色		说明
领导者 (leader)		负责进行投票的发起和决议，更新系统状态。
学习者 (Learner)	跟随者 (Follower)	Follower 用于接收客户请求并向客户返回结果，且在选主过程中，参考投票。
	观察者 (Observer)	ObServer 可以接收客户端连接，将写请求转发给 Leader 节点。但 Observer 不参与投票过程，只同步 Leader 的状态。Observer 的目的是为了扩展系统，提高读取速度。
客户端 (Client)		请求发起方。

系统模型如图 7.1.2 所示：

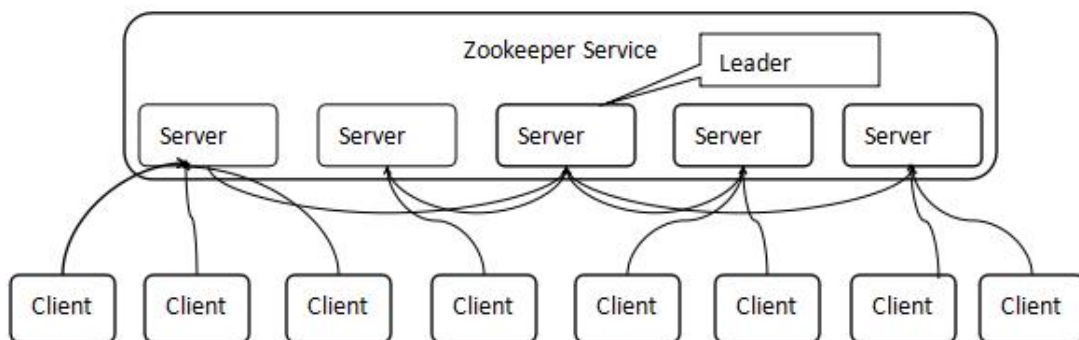


图 7.1.2

zooKeeper 中的时间。ZooKeeper 有多种记录时间的形式，其中包含以下几个主要

属性：

(1) Zxid

致使 ZooKeeper 节点状态改变的每一个操作都将使节点接收到一个 Zxid 格式的时间戳，并且这个时间戳全局有序。也就是说，也就是说，每个对 节点的改变都将产生一个唯一的 Zxid。如果 Zxid1 的值小于 Zxid2 的值，那么 Zxid1 所对应的事件发生在 Zxid2 所对应的事件之前。实际上，ZooKeeper 的每个节点维护者三个 Zxid 值，分别为：cZxid、mZxid、pZxid。

① cZxid： 是节点的创建时间所对应的 Zxid 格式时间戳。

② mZxid： 是节点的修改时间所对应的 Zxid 格式时间戳。

实现中 Zxid 是一个 64 为的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个 新的 epoch。低 32 位是个递增计数。

(2) 版本号

对节点的每一个操作都将致使这个节点版本号增加。每个节点维护着三个版本号，他们分别为：

① version： 节点数据版本号

② cversion： 子节点版本号

③ aversion： 节点所拥有的 ACL 版本号

ZooKeeper 节点属性。一个节点自身拥有表示其状态的许多重要属性，如下所示：

属性	说明
czxid	节点被创建的 zxid
mzxid	节点被修改的 zxid
ctime	节点被创建的时间
mtime	节点被修改的时间
version	节点被修改的版本号
cversion	节点所拥有的子节点被修改的版本号

aversion	节点的 ACL 被修改的版本号
ephemeralOwner	如果此节点为临时节点，那么它的值为这个节点拥有者的会话 ID，否则值为 0。
dataLength	节点数的长度
numChildren	节点拥有的子节点的长度
pzxid	最新修改的 zxid。

ZooKeeper 服务中操作。在 ZooKeeper 中有 9 个基本操作，如下所示：

操作	说明
create	创建 znode（父 znode 必须存在）
delete	删除 znode（znode 没有子节点）
exists	判断 znode 是否存在
getACL/setACL	读取或设置 ACL
getChildren	获取 znode 所有子节点的列表
getData/setData	获取/设置 znode 相关数据
sync	客户端与 zookeeper 同步

zookeeper 的配置文件。zookeeper 的默认配置文件为 \$ZOOKEEPER\_HOME/conf/zoo.cfg。其中各配置项的含义，解释如下：

1. tickTime：CS 通信心跳时间

Zookeeper 服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个 tickTime 时间就会发送一个心跳。tickTime 以毫秒为单位。如 tickTime=2000。

2. initLimit：LF 初始通信时限

集群中的 follower 服务器(F)与 leader 服务器(L)之间初始连接时能容忍的最多心跳数（tickTime 的数量）。如 initLimit=5。

3. syncLimit：LF 同步通信时限

集群中的 follower 服务器与 leader 服务器之间请求和应答之间能容忍的最多心跳数

(tickTime 的数量)。如 syncLimit=2。

#### 4. dataDir: 数据文件目录

Zookeeper 保存数据的目录, 默认情况下, Zookeeper 将写数据的日志文件也保存在这个目录里。如设置 dataDir=/home/zoo/SomeData。

#### 5. clientPort: 客户端连接端口

客户端连接 Zookeeper 服务器的端口, Zookeeper 会监听这个端口, 接受客户端的访问请求。clientPort=2181。

6. 服务器名称与地址: 集群信息 (服务器编号, 服务器地址, LF 通信端口, 选举端口)。这个配置项的书写格式比较特殊, 规则如下:

```
server.N=YYY:A:B
```

如配置以下示例

```
server.1=ip:2888:3888
```

```
server.2=ip:2888:3888
```

```
server.3=ip:2888:3888
```

## 7.2、单一节点安装 zookeeper

请确认在安装 zookeeper 之前, 已经安装了 jdk, 并正确的配置了 JAVA\_HOME 和 PATH 环境变量。单一节点安装只要解压 zookeeper 并配置 zoo.cfg 文件即可。修改 dataDir 数据目录。

### 步 1、下载 zookeeper

下载 zookeeper。可以通过 wget 下载 zookeeper。如果没有安装 wget 可以通过 yum install -y wget 安装此软件。也可以通过 xftp 上传已经下载好的 zookeeper 压缩文件。下载:

```
$wget
http://apache.fayea.com/zookeeper/zookeeper-3.4.10/zookeeper-3.4.10.tar.gz
```

## 步 2、解压

解压到指定的目录下或当前目录下：

```
tar -zxvf soft/zookeeper-3.4.10.tar.gz
```

## 步 3、修改配置文件设置 dataDir 目录

修改示例的配置文件：

```
$ cp zoo_sample.cfg zoo.cfg  
  
vim zoo.cfg  
  
dataDir=/home/${username}/data
```

将上面的\${username}修改成当前登录的用户名称，其他保持默认值即可。

## 步 4、启动 zookeeper

```
$ ./zkServer.sh start #启动服务器  
  
$ jps #查看进程  
  
2490 QuorumPeerMain #这是 zookeeper 的进程
```

## 步 5、登录客户端

```
$ ./zkCli.sh
```

```
[zk: localhost:2181(CONNECTED) 0] help #通过 help 可以查看 zk 的命令
```

或是指定连接的服务器：

```
$ ./zkCli.sh -server 192.168.56.201:2181
```

基本客户端命令：

```
[zk: 192.168.56.201:2181(CONNECTED) 2] create /weric WangJian #创建 znode  
保存  
Created /weric  
[zk: 192.168.56.201:2181(CONNECTED) 3] ls / #显示 zk 上的所有目录  
[weric, zookeeper]  
[zk: 192.168.56.201:2181(CONNECTED) 4] get /weric #某个目录中的数据
```

## 7.3、zookeeper 集群安装

zookeeper 集群，首先就在多台机器上，都安装 zookeeper，为了便于记忆，可以将 zookeeper 安装到相同的目录下，如/weric 目录下。

在每一个 zookeeper 的 dataDir 目录下，创建一个 myid 文件，里面保存的是当前 zookeeper 节点的 id。id 不一定从 1 开始。本示例中的 id 根据 ip 地址最后一段做为 id，以便于记忆。

### 步 1、配置 zoo.cfg

三台 CentOS7 主机，且关闭防止墙。然后修改 \$ZOOKEEPER\_HOME/conf/zoo.cfg，在文件最后追加以下配置：

#配置 zookeeper 数据保存目录

```
dataDir=/weric/zookeeper_data
```

#配置 zookeeper 集群

```
server.11=192.168.56.11:2888:3888
```

```
server.12=192.168.56.12:2888:3888
```

```
server.13=192.168.56.13:2888:3888
```

### 步 2、使用 scp 将文件发送到其他两台机器

```
$ scp -r zookeeper-3.4.10 weric12:/weric
```

然后修改每一个 dataDir 目录下的 myid 文件。在 weric11 主机上的 myid 中添加 11。即：

```
echo 11 > /weric/zookeeper_data/myid
```

依此类推。

### 步 3、现在分别启动三台主机

启动命令使用 zkServer.sh start：

```
$ ./zkServer.sh start
```

然后查看状态，使用 status 检查状态：

```
[zk@weric201 bin]$ ./zkServer.sh status
```

```
Mode: leader    #这是 leader 其他两台为 follower
```

现在就可以同步测试了。在一台上进行操作，查看其他两台的同步情况。

#### 步 4、测试操作

登录客户端

```
$ zkCli.sh
```

显示当前所有目录

```
[zk: localhost 2181] ls /
```

```
[zookeeper]
```

创建一个新的目录，且写入数据

```
[zk : localhost : 2181 ] create /weric WangJian
```

再次显示当前根目录下的所有数据，也可以登录其他主机，会查看到相同的结果，即表示已经同步。

```
[zk: localhost : 2181 ] ls /
```

```
[weric , zookeeper]
```

zookeeper 的命令很多，可以使用 help 查看所有可使用的命令。

```
[zk: localhost : 2181] help
```

通过上面的配置可以看出，zookeeper 的集群配置相对比较简单。只是配置 zoo.cfg 并指定所有服务节点即可。然后在每一个节点的数据目录下，将当前 id 写入到 myid 文件中即可。

## 7.4、配置 hadoop 高可靠集群

hadoop2.0 已经发布了稳定版本了，增加了很多特性，比如 HDFS HA、YARN 等。最新的 hadoop-2.4.1 又增加了 YARN HA。hadoop 的高可靠(HA)集群，可以使用 zookeeper 实现容灾的自动切换。hadoop 高可靠集群，将会运行两个 NameNode，一个 NameNode



为 Active 状态,另一个则为 Standby 状态。这两个 NameNode 属于同一个 NameService, 而一个 NameService 最多只能有两个 NameNode。NameNode 进程将会与一个叫 zkfc 的进程在同一台主机上。zkfc 即 DFSZKFailoverController 用于监控 NameNode 的状态, 并将状态汇报给 zookeeper, 一旦 NameNode 变的不可用, 就会实现自动的切块。同时, 也可以拥有两个 ResourceManager, 同样使用 zookeeper 实现自动的切换。同样的一个 ResourceManager 为 Active 状态, 另一个则为 Standby 状态。为了同步 NameNode 的 edits 等文件, 还要开启多个 JournalNode 用于时时同步日志信息。JournalNode 节点必须是单数, 最少三台。如 3, 5, 7 等等都是可以的。

以下是一个 7 台服务器集群的配置列表。一般情况下, NameNode 和 ResourceManager 要分开, 因为一个是 hdfs 的主进程, 一个是 yarn 的主进程。

概述	主机名/ip 地址	软件	进程
NameNode 两台	weric201 192.168.56.201	jdk hadoop	NameNode DFSZKFailoverController(zkfc)
	weric202 192.168.56.202	jdk hadoop	NameNode DFSZKFailoverController(zkfc)
ResouceManager 两台	weric203 192.168.56.203	jdk hadoop	ResourceManager
	weric204 192.168.56.204	jdk hadoop	ResourceManager
DataNode 三台 zookeeper 三台 JournalNode 三台	weric205 192.168.56.205	jdk hadoop zookeeper	DataNode NodeManager JournalNode QuorumPeerMain

	weric206  192.168.56.206	jdk  hadoop  zookeeper	DataNode  NodeManager  JournalNode  QuorumPeerMain
	weric207  192.168.56.207	jdk  hadoop  zookeeper	DataNode  NodeManager  JournalNode  QuorumPeerMain

这些进程的主要功能是：

**DFSZKFailoverController**：用于监控 NameNode 并将信息汇报给 Zookeeper。

**JournalNode**：用于保存共享的 edits 信息。即 NameNode 的元数据信息。

**NameService**：hadoop2.0 以后可以有两个或更多 NameNode, 使用 NameService 来管理 NameNode。每一个 NameService 可以最多拥有两个 NameNode。NameService 的水平扩展说明可以拥有更多的 NameNode。

**SecondaryNameNode**：在 HA 集群中，standby 状态的 NameNode 可以完成 checkpoint 操作,因此没必要配置 Secondary NameNode、CheckpointNode、BackupNode。如果真的配置了，还会报错。

**ResourceManager**：在 Hadoop2.4 以后，又可以有多个 ResouceManager。使用 zookeeper 进行分布式管理。以提高 Resoucemanager/yarn 的高可用。

**NameNode**：集群启动时，建议可以同时启动 2 个 NameNode。这些 NameNode 只有一个 active 的，另一个属于 standby 状态。active 状态意味着提供服务，standby 状态意味着处于休眠状态，只进行数据同步，时刻准备着提供服务，如图 7.4.1 所示为 HDFS 的结构。

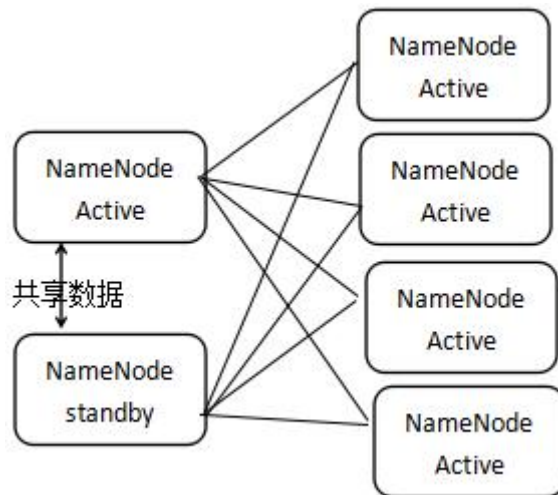


图 7.4.1

JournalNode 服务器：运行的 JournalNode 进程非常轻量，可以部署在其他的服务器上。注意：必须允许至少 3 个节点。当然可以运行更多，但是必须是奇数个，如 3、5、7、9 个等等。当运行 N 个节点时，系统可以容忍至少  $(N-1)/2$  个节点失败而不影响正常运行。

两个 NameNode 为了数据同步，会通过一组称作 JournalNodes 的独立进程进行相互通信。当 active 状态的 NameNode 的命名空间有任何修改时，会告知大部分的 JournalNodes 进程。standby 状态的 NameNode 有能力读取 JNs 中的变更信息，并且一直监控 edit log 的变化，把变化应用于自己的命名空间。standby 可以确保在集群出错时，命名空间状态已经完全同步了，如图 7.4.2 所示：

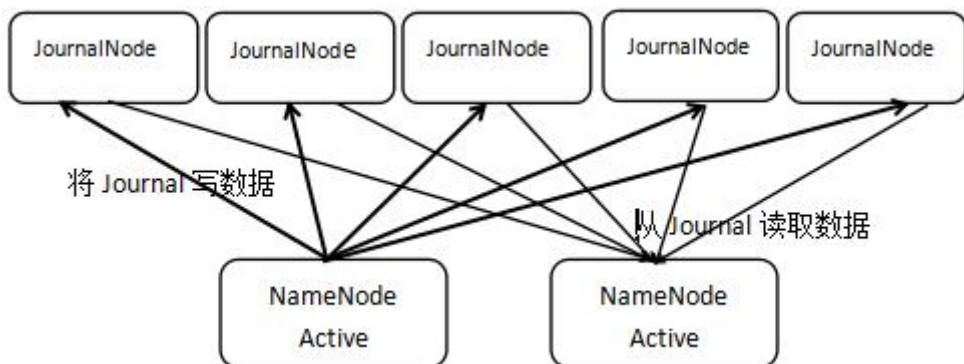


图 7.4.2

为了确保快速切换，standby 状态的 NameNode 有必要知道集群中所有数据块的位置。为了做到这点，所有的 datanodes 必须配置两个 NameNode 的地址，发送数据块位置信息和心跳给他们两个。

对于 HA 集群而言，确保同一时刻只有一个 NameNode 处于 active 状态是至关重要的。否则，两个 NameNode 的数据状态就会产生分歧，可能丢失数据，或者产生错误的结果。为了保证这点，JNs 必须确保同一时刻只有一个 NameNode 可以向自己写数据。



#### 【注意】

IPC 通讯端口为：8020。

现在我们配置一个 3 台服务器集群的示例，如果读者感兴趣，可以参考这 3 台服务器集群的过程，来实现上面 7 台或是更多集群的配置。

### 步 1、前期准备

1. 修改 Linux 主机名
2. 修改 IP
3. 修改主机名和 IP 的映射关系

/etc/hosts 里面要配置的是内网 IP 地址和主机名的映射关系

4. 关闭防火墙
5. ssh 免登陆

NameNode 必须要可以 ssh 免密码登录所有机器

ResourceManager 必须要可以免密码登录所有机器

每一个主机必须要可以免密码登录自己

6. 安装 JDK 和配置 JAVA\_HOME, PATH 环境变量。
7. 三台都安装好 zookeeper，并测试集群成功。

## 8. 配置表

主机名/IP 地址	软件	进程
weric11 192.168.56.11	JDK zookeeper Hadoop	NameNode zkfc ResourceManager NodeManager DataNode JournalNode QuorumPeerMain
weric12 192.168.56.12	JDK zookeeper Hadoop	NameNode zkfc ResourceManager NodeManager DataNode JournalNode QuorumPeerMain
weric13 192.168.56.13	JDK zookeeper Hadoop	NodeManager DataNode JournalNode QuorumPeerMain

### 步 2、在主机 weric11 上配置 hadoop

现在在 weric11 的主机上的/weric 目录下，配置 hadoop。

#### 1)、解压 hadoop

```
$ tar -zxvf ~/hadoop-2.8.0.tar.gz -C /weric
```

配置 hadoop-env.sh 文件，配置 JAVA\_HOME 环境变量

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_131
```

## 2)、配置 core-site.xml 文件

```
<configuration>

    <!-- 指定 hdfs 的 nameservice 为 ns1 -->

    <property>

        <name>fs.defaultFS</name>

        <value>hdfs://ns1</value>

    </property>

    <!-- 指定 hadoop hdfs 目录 -->

    <property>

        <name>hadoop.tmp.dir</name>

        <value>/weric/hadoop_tmp_dir</value>

    </property>

    <!-- 指定 zookeeper 地址 -->

    <property>

        <name>ha.zookeeper.quorum</name>

        <value>weric11:2181,weric12:2181,weric13:2181</value>

    </property>

</configuration>
```

## 3)、配置 hdfs-site.xml 文件，这里的配置信息比较多，请注意观察

```
<configuration>

<!--指定 hdfs 的 nameservice 为 ns1，需要和 core-site.xml 中的保持一致 -->

    <property>

        <name>dfs.nameservices</name>

        <value>ns1</value>

    </property>
```

```
<!-- ns1 下面有两个 NameNode，分别是 nn1，nn2 -->

<property>

    <name>dfs.ha.namenodes.ns1</name>

    <value>nn1,nn2</value>

</property>

<!-- nn1 的 RPC 通信地址 -->

<property>

    <name>dfs.namenode.rpc-address.ns1.nn1</name>

    <value>weric11:8020</value>

</property>

<!-- nn1 的 http 通信地址 -->

<property>

    <name>dfs.namenode.http-address.ns1.nn1</name>

    <value>weric11:50070</value>

</property>

<!-- nn2 的 RPC 通信地址 -->

<property>

    <name>dfs.namenode.rpc-address.ns1.nn2</name>

    <value>weric12:8020</value>

</property>

<!-- nn2 的 http 通信地址 -->

<property>

    <name>dfs.namenode.http-address.ns1.nn2</name>

    <value>weric12:50070</value>

</property>

<!-- 指定 NameNode 的元数据在 JournalNode 上的存放位置 -->
```

```

<property>

    <name>dfs.namenode.shared.edits.dir</name>

    <value>qjournal://weric11:8485;weric12:8485;weric13:8485/ns1</value>

</property>

<!-- 指定 JournalNode 在本地磁盘存放数据的位置 -->

<property>

    <name>dfs.journalnode.edits.dir</name>

    <value>/weric/journalnode_edits_dir</value>

</property>

<!-- 开启 NameNode 失败自动切换 -->

<property>

    <name>dfs.ha.automatic-failover.enabled</name>

    <value>true</value>

</property>

<!-- 自动切换实现方式 -->

<property>

    <name>dfs.client.failover.proxy.provider.ns1</name>

    <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverPro
xyProvider</value>

</property>

<!-- 配置隔离机制方法，多个机制用换行分割，即每个机制暂用一行-->

<property>

    <name>dfs.ha.fencing.methods</name>

    <value>

        sshfence

        shell(/bin/true)

```



```

        </value>

    </property>

    <!-- 使用 sshfence 隔离机制时需要 ssh 免登陆, 以下是$username} 请替换成
读者自己的用户名 -->

    <property>

        <name>dfs.ha.fencing.ssh.private-key-files</name>

        <value>/home/${username}/.ssh/id_rsa</value>

    </property>

    <!-- 配置 sshfence 隔离机制超时时间 -->

    <property>

        <name>dfs.ha.fencing.ssh.connect-timeout</name>

        <value>30000</value>

    </property>

</configuration>

```

#### 4)、配置 mapred-site.xml

```

<configuration>

    <!-- 指定 mr 框架为 yarn 方式 -->

    <property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

    </property>

</configuration>

```

#### 5)、配置 yarn-site.xml

```

<configuration>

    <property><!-- 开启 RM 高可靠 -->

        <name>yarn.resourcemanager.ha.enabled</name>

```

```

        <value>true</value>
    </property>
    <!-- 指定 RM 的 cluster id , 名称可以任意 -->
    <property>
        <name>yarn.resourcemanager.cluster-id</name>
        <value>weric</value>
    </property>
    <property><!-- 指定 RM 的名字 -->
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>
    <!-- 分别指定 rm1 和 rm2 的地址 -->
    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>weric11</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>weric12</value>
    </property>
    <property><!-- 指定 zk 集群地址 -->
        <name>yarn.resourcemanager.zk-address</name>
        <value>weric11:2181,weric12:2181,weric13:2181</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services</name>

```

```
<value>mapreduce_shuffle</value>

</property>

</configuration>
```

## 6)、配置 slaves 文件

slaves 是指定 DataNode 节点的位置。在里面添加主机的名称或是 ip 地址即可，一行一个

```
weric11

weric12

weric13
```

## 7)、现在配置 hadoop 的环境变量

```
$ sudo vim /etc/profile.d/hadoop.sh
```

在文件中添加以下内容

```
export HADOOP_HOME=/weric/hadoop-2.8.0

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

让环境变量生效，执行以下命令

```
$source /etc/profile
```

执行 hadoop version 命令，如果可以显示版本，则说明环境变量配置成功

```
$ hadoop version
```

## 8)、拷贝文件

将配置好的 hadoop 目录，copy 到其他主机相同的目录下使用 scp 命令。由于 share 目录下的 doc 里面都是文档，可以删除这个目录，以加快 copy 速度。

```
$ scp -r hadoop-2.8.0/ weric13:/weric/

$ scp -r hadoop-2.8.0/ weric12:/weric/
```

## 9)、将 hadoop.sh 环境变量也 copy 到其他的主机上

```
$ sudo scp /etc/profile.d/hadoop.sh root@weric12:/etc/profile.d/

$ sudo scp /etc/profile.d/hadoop.sh root@weric13:/etc/profile.d/
```

## 10)、启动 zookeeper 集群

分别在 weric11、weric12、weric13 上启动 zk。

```
$ ./zkServer.sh start
```

#查看状态：一个 leader，两个 follower

```
./zkServer.sh status
```

## 11)、启动 journalnode

分别在在 weric11、weric12、weric13 上执行。

```
$ ./hadoop-daemon.sh start journalnode
```

## 12)、格式化 HDFS

#在 weric11 上执行命令：

```
hdfs namenode -format
```

格式化后会在根据 core-site.xml 中的 hadoop.tmp.dir 配置生成个文件，然后将这个文件使用 scp 拷贝到 weric12 的相同目录下。因为，都是 NameNode 节点，必须要拥有相同的数据文件。格式化成功的标志是在输出的日志中查看是否存在以下语句：

```
Storage directory /weric/hadoop_tmp_dir/dfs/name has been successfully formatted
```

现在将格式化后的 hdfs 目录，拷贝到 weric12 主机上的相同目录下：

```
$ scp -r hadoop_tmp_dir/ weric12:/weric/
```

## 13)、格式化 zkfc

在 weric11 上执行

```
hdfs zkfc -formatZK
```

在格式化完成以后，通过 zkCli.sh 登录 zookeeper 并查看目录列表，将显示一个 hadoop-ha 的目录，表示初始化成功

```
[zk: localhost:2181(CONNECTED) 0] ls /
```

```
[zookeeper, hadoop-ha]
```

#### 14)、启动 HDFS(在 weric11 上执行)

在 weric11 上启动 hdfs 即 NameNode 同时也会将 weric12 的 nameNode 一并启动。

```
$ ./start-dfs.sh
```

#### 15)、启动 YARN

分别在 weric11 和 weric12 上启动 yarn

```
$ ./start-yarn.sh
```

在启动完成以后，根据之前的配置列表，分别检查每一个主机上的服务是否都已经启动。如果没有请查看日志错误。

#### 16)、验证高可靠

通过浏览器访问以下是地址可以查看 hdfs 的信息，如图 7.4.3 所示：

<http://192.168.1.11:50070>



图 7.4.3

通过图 7.4.3 可以看出当前 NameNode 为 active。而通过图 7.4.4 所示 weric12 上的 NameNode 为 Standby。



图 7.4.4

也可以通过以下命令，检查 NameNode 和 ResourceManager 的状态

```
$ hdfs haadmin -getServiceState nn1  
active  
  
$ hdfs haadmin -getServiceState nn2  
standby  
  
$ yarn rmadmin -getServiceState rm1  
active  
  
$ yarn rmadmin -getServiceState rm2  
standby
```

现在让我们 kill 掉 active 的 NameNode，即 kill 掉 nn1

```
$kill -9 <pid of NN>
```

然后再检查状态，这个时候 weric12 上的 NameNode 变成了 active

```
$ hdfs haadmin -getServiceState nn2  
active
```

手动启动那个挂掉的 NameNode，即 nn1，然后再检查状态，它已经成为 standby 的了

```
$. /hadoop-daemon.sh start namenode  
  
$ hdfs haadmin -getServiceState nn1  
standby
```

使用同样的方式，可以验证 ResourceManager 是否可以自动实现容灾切换。



#### 【注意】

- 1: 在集群完成以后，建议执行一个 mapreduce 测试，如 wordcount。
- 2: Hadoop 的高可靠集群每一次启动相对比较麻烦。但配置成功以后，下次启动就相对比较简单了。对于上面的示例而言，再次启动只要在 weric11 主机上执行 ./start-dfs.sh 和分别在 weric11、weric12 主机上执行 ./start-yarn.sh 即可。



### 【注意】

关于免密码登录的说明

要求能通过免登录包括使用 IP 和主机名都能免密码登录：

- 1) NameNode 能免密码登录所有的 DataNode
- 2) 各 NameNode 能免密码登录自己
- 3) 各 NameNode 间能免密码互登录
- 4) DataNode 能免密码登录自己
- 5) DataNode 不需要配置免密码登录 NameNode 和其它 DataNode。
- 6) ResourceManager 必须要免密码登录所有 DataNode 以便于启动 NodeManager。

## 7.5、用 Java 代码操作集群

用 Java 客户端面操作集群开发 hdfs，必须要指定 nameService 的配置信息。以下是代码示例，以下代码显示 hdfs 上的文件和目录：

```
package cn.hadoop;

import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.junit.Test;

public class Demo02 {

    @Test

    public void test1() throws Exception {
```

```

Configuration config = new Configuration();

// 指定 NameServicer 的名称
config.set("dfs.nameservices", "ns1");

// 指定某个 nameService 下的两个 NameNode
config.set("dfs.ha.namenodes.ns1", "nn1,nn2");

// 指定每一个 nameNode 的地址 是 8020 或是 9000, 默认是 8020
config.set("dfs.namenode.rpc-address.ns1.nn1", "192.168.56.11:9000");
config.set("dfs.namenode.rpc-address.ns1.nn2", "192.168.56.12:9000");

// 指定灾难处理类
config.set("dfs.client.failover.proxy.provider.ns1",
"org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider");

// 使用在 core-site.xml 中配置的 fs.defaultFS 的值
FileSystem fs = FileSystem.get(new URI("hdfs://ns1"), config);

// 现在就可以操作集群了

InputStream in = fs.open(new Path("/w.txt"));

OutputStream out = new FileOutputStream("d:/a/ab.txt");

byte[] bs = new byte[1024 * 4];

int len = 0;

while ((len = in.read(bs)) != -1) {
    out.write(bs, 0, len);
}

out.close();

}}

```



## 7.6、小结

- ❖ Zookeeper 是简单的 - 配置简单，命令简单
- ❖ Zookeeper 是富有表现力的 - 提供丰富的编程接口
- ❖ Zookeeper 具有高可用性 - HA
- ❖ Zookeeper 采用松耦合交互方式
- ❖ Zookeeper 是一个资源库
- ❖ hadoop 借助 zookeeper 实现 ha 高可靠。
- ❖ zkfc 用于时时向 zookeeper 汇报 NameNode 的状态，一旦 NameNode 变的不可用，就会自动的进行切换。
- ❖ JournalNode 用于时间同步两个 NameNode 之间的日志数据。
- ❖ 可以使用 hdfs haadmin 查看 NameNode 的状态。可以使用 yarn rmadmin 查看 ResourceManager 的状态。

# 第 8 章 sqoop

## 内容简介

- sqoop 的安装
- sqoop 的基本命令
- 数据导入导出示例

sqoop 是一个数据迁移工具。sqoop 非常简单，其整合了 Hive 、 Hbase 和 Oozie ，通过 map-reduce 任务来传输数据，从而提供并发特性和容错。

Sqoop 由于是将数据导入到 hdfs 中，所以需要依赖于 hadoop。即前提上 hadoop 已经安装且正确配置。

sqoop 主要通过 JDBC 和关系数据库进行交互。理论上支持 JDBC 的 database 都可以使用 sqoop 和 hdfs 进行数据交互 。如将 database 中的数据导入到 hdfs 或是将 hdfs 中的数据导入到 database 中。本部分将介绍 sqoop1.x 的使用。

## 8.1、安装 sqoop

安装 sqoop 非常简单，只要下载 sqoop 并解压到任意一个台已经安装好 hadoop 的机器上即可。

下载时，请下载以下完整的文件， 如图 8.1.1 所示：



<u>Name</u>	<u>Last modified</u>	<u>Size</u>
<a href="#">Parent Directory</a>		-
<a href="#">sqoop-1.4.6.bin_hadoop-0.23.tar.gz</a>	2015-05-08 16:28	16M
<a href="#">sqoop-1.4.6.bin_hadoop-1.0.0.tar.gz</a>	2015-05-08 16:28	16M
<a href="#">sqoop-1.4.6.bin_hadoop-2.0.4-alpha.tar.gz</a>	2015-05-08 16:28	16M
<a href="#">sqoop-1.4.6.tar.gz</a>	2015-05-08 16:28	2.1M

图 8.1.1

或在 linux 的命令行模式下，直接使用 wget 下载：

```
$wget
```

```
https://mirrors.tuna.tsinghua.edu.cn/apache/sqoop/1.4.6/sqoop-1.4.6.bin_hadoop-2.0.4-alpha.tar.gz
```

解压到 Linux 的任意目录下：

```
$ tar zxvf sqoop-1.4.6.bin_hadoop-2.0.4-alpha.tar.gz
```

解压以后，由于目录比较长，可以使用 mv 命令，修改名称：

```
$ mv sqoop-1.4.6.bin_hadoop-2.0.4-alpha sqoop-1.4.6
```

进入 sqoop 的 bin 目录，并执行 ./sqoop help

```
usage: sqoop COMMAND [ARGS]
```

Available commands:

codegen	Generate code to interact with database records
create-hive-table	Import a table definition into Hive
eval	Evaluate a SQL statement and display the results
export	Export an HDFS directory to a database table 从 hdfs 导出数据到 db
help	List available commands
import	Import a table from a database to HDFS 将数据导入到 hdfs
import-all-tables	Import tables from a database to HDFS
import-mainframe	Import datasets from a mainframe server to HDFS
job	Work with saved jobs
list-databases	List available databases on a server 显示某个数据连接上所有数据库
list-tables	List available tables in a database 显示所有数据表
merge	Merge results of incremental imports
metastore	Run a standalone Sqoop metastore

version	Display version information
---------	-----------------------------

在上面的命令中，使用的比较多的是 export 和 import。

## 8.2、sqoop 基本命令

使用 sqoop 可以将数据库中的数据导入到 hdfs 中。在使用之前，应该将对应的数据库驱动拷贝到 sqoop 下的 lib 目录下。以下示例，都将使用 mysql 数据库，所以，必须要将 mysql 数据库的驱动包放到 sqoop 下的 lib 目录下。

一般情况下，sqoop 的命令，都会比较长，所以以下多数命令使用 shell 脚本的方式实现，在脚本中直接输入\（斜线）表示命令并没有结束，此处为一个换行。关于脚本的编写读者可以自行研读 Linux 的相关教程。同时，以下的示例中读者要根据自己的具体情况修改数据连接的 ip 地址及用户名和密码。

同时为了便于操作，建议将 sqoop 配置到环境变量中，配置如下：

```
$sudo vim /etc/profile.d/sqoop.sh
```

添加以下内容

```
#!/bin/sh

export ZOOKEEPER_HOME=/cluster/zookeeper-3.4.10

export SQOOP_HOME=/cluster/sqoop-1.4.6

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

### 1)、list-databases

list-databases 用于显示某个连接上所有数据库

```
sqoop list-databases \
--connect jdbc:mysql://192.168.56.1:3306/ \
--username root \
--password 1234
```

也可以直接使用 sqoop-list-databases 命令：

```
#!/bin/bash
#使用 sqoop 命令
./sqoop-list-databases \
--connect jdbc:mysql://192.168.56.1:3306/ \
--username root \
--password 1234
```

## 2)、list-tables

list-tables 用于显示某个数据库中的所有表:

```
#!/bin/bash
./sqoop list-tables \
--connect jdbc:mysql://192.168.56.1:3306/weric \
--username root \
--password 1234
```

## 3)、eval

eval 用于执行一个 sql 语句, 并将结果输出到控制台, 请自行修改查询的表名。

```
#!/bin/bash
sqoop eval \
--connect jdbc:mysql://192.168.56.1:3306/weric \
--username root \
--password 1234 \
--query "select * from users"
```

# 8.3、导入导出命令

## 1)、import

import 命令, 用于将数据库中的数据导入到 hdfs。其中--table 参数用于将一个

表中的数据全部的导入到 hdfs 中去。

1.1)、--table 指定导出的表:

```
#!/bin/bash
./sqoop import \
--connect \
jdbc:mysql://192.168.56.1:3306/weric?characterEncoding=UTF-8 \
--username root \
--password 1234 \
--table studs \ #指定表名
-m 2 \          #指定 mapper 的个数，不能超过集群节点的数量，默认为 4
--split-by "id" \          #只要-m 不是 1 必须要指定分组的字段名称
--where "age>100 and sex='1'" \ #指定 where 条件，可以使用 “” 双引号
--target-dir /out001      #指定导入到 hdfs 以后目录
```

默认导出到 hdfs 的数据以，（逗号）分开如下所示:

```
$ hdfs dfs -cat /out001/*
2a56b3536b544f289ba79b2b5c1196c4, Jerry, 89e4a3e..3ee3e03946d85d
cc645dc7811740fc9856b1c7c8e19e89, Alex, c924e3..5a0487e23207986189d
U001, Jack, 1234
U002, Mike, 1234
```

可以使用 --fields-terminated-by 参数，指定分割符号，如 --fields-terminated-by “\t” 将侵害符号设置为制表符。

1.2)、--query 指定查询语句

如果在 import 中已经使用了 --query 语句则 --where 和 --table 将被忽略。在 --query 所指定的语句中，必须要将 \$CONDITIONS 做为条件添加到 where 子句中。如果 --query 使用后面使用 ” ” 又引号则应该使用 \ \$CONDITIONS。注意前面的 \（斜线）。

```
#!/bin/bash
```

```

sqoop import \
--connect jdbc:mysql://192.168.56.1:3306/qlu?characterEncoding=UTF-8 \
--username root \
--password 1234 \
#注意以下使用的 SQL 语句，如果使用” ” 双引号则必须要添加\在$CONDITIONS 前面
--query "select name,sex,age,addr from studs where sex='0' and addr like '
山东%' and \$CONDITIONS" \
--split-by "name" \
--fields-terminated-by "\t" \ #使用制表符号进行数据分隔
--target-dir /out002 \
-m 2

```

--query 参数的 SQL 可以写的很复杂，如下面的示例，将是一个关联的查询语句：

```

#!/bin/bash
sqoop import \
--connect jdbc:mysql://192.168.56.1:3306/studs?characterEncoding=UTF-8 \
--username root \
--password 1234 \
--query \
"select s.stud_id,s.stud_name as sname,c.course_name as cname \
from studs s inner join sc on s.stud_id=sc.sid \
inner join courses c on c.course_id=sc.cid where \$CONDITIONS" \
--target-dir /out004 \
--split-by "s.stud_id" \ #根据某个列进行分组
-m 2

```

## 2)、export 导出到关系型数据库中去

使用 sqoop export 命令可以将 hdfs 数据导出到关系型数据库中去。

```
#!/bin/bash

sqoop export \

- -connect \

  jdbc:mysql://192.168.56.1:3306/weric?characterEncoding=UTF-8 \

--username root \

--password 1234 \

--export-dir /out001 \ #指定导出的目录

--table "studs" \ #指定 hdfs 中数据与数据库中表列的对应关系

--columns "stud_id,stud_age,stud_name" \#指定 hdfs 中数据进行进行分隔

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

-m 2
```

## 8.4、小结

- ❖ SQOOP 是一个数据迁移工具。
- ❖ SQOOP 使用简单，主要使用的命令为 import 和 export。



# 第 9 章 HBase

## 内容简介

- HBase 的特点
- HBase 的存储结构
- HBase 操作命令
- HBase 伪分布式
- HBase 分布式

Hbase 是 Hadoop DataBase 的含义。HBase 是一种构建在 HDFS 之上的分布式、面向列的存储系统。在需要实时读写、随机访问超大规模数据集时,可以使用 HBase。

HBase 是 Google Bigtable 的开源实现,与 Google Bigtable 利用 GFS 作为其文件存储系统类似, HBase 利用 Hadoop HDFS 作为其文件存储系统,也是利用 hdfs 实现分布式存储的。Google 运行 MapReduce 来处理 Bigtable 中的海量数据, HBase 同样利用 Hadoop MapReduce 来处理 HBase 中的海量数据; Google Bigtable 利用 Chubby 作为协同服务, HBase 利用 Zookeeper 作为对应。

## 9.1、HBase 的特点

- 1、大: 一个表可以有上亿行,上百万列。
- 2、面向列: 面向列表(簇)的存储和权限控制,列(簇)独立检索。
- 3、稀疏: 对于为空(NULL)的列,并不占用存储空间,因此,表可以设计的非常稀疏。
- 4、无模式: 每一行都有一个可以排序的主键和任意多的列,列可以根据需要动态增加,同一张表中不同的行可以有截然不同的列。
- 5、数据多版本: 每个单元中的数据可以有多个版本,默认情况下,版本号自动分

配，版本号就是单元格插入时的时间戳。

6、数据类型单一：HBase 中的数据都是字符串，没有类型。

1)、HBase 的高并发和实时处理数据

Hadoop 是一个高容错、高延时的分布式文件系统和高并发的批处理系统，不适用于提供实时计算；HBase 是可以提供实时计算的分布式数据库，数据被保存在 HDFS 分布式文件系统上，由 HDFS 保证期高容错性，但是在生产环境中，HBase 是如何基于 hadoop 提供实时性呢？ HBase 上的数据是以 StoreFile(HFile) 二进制流的形式存储在 HDFS 上 block 块儿中；但是 HDFS 并不知道 hbase 存的是什么，它只把存储文件是为二进制文件，也就是说，hbase 的存储数据对于 HDFS 文件系统是透明的。

图 9.1.1 是 HBase 文件在 HDFS 上的存储示意图。

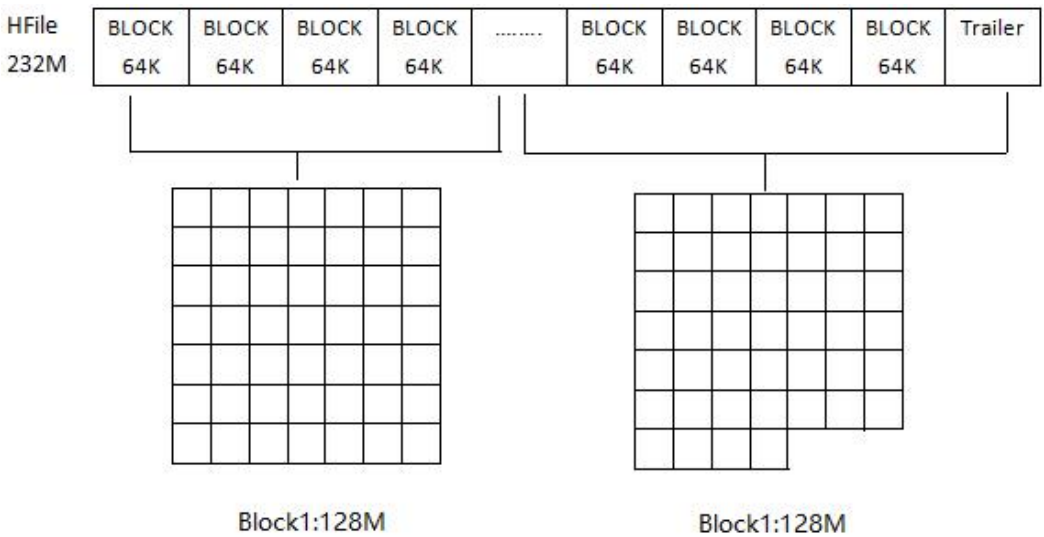


图 9.1.1

HBase HRegion servers 集群中的所有的 region 的数据在服务器启动时都是被打开的，并且在内存初始化一些 memstore，相应的这就在一定程度上加快系统响应。而 Hadoop 中的 block 中的数据文件默认是关闭的，只有在需要的时候才打开，处理完数据后就关闭，这在一定程度上就增加了响应时间。

2)、HBase 数据模型

HBase 以表的形式存储数据。表由行和列组成。列划分为若干个列族(row family)，

如图 9.1.2 所示。

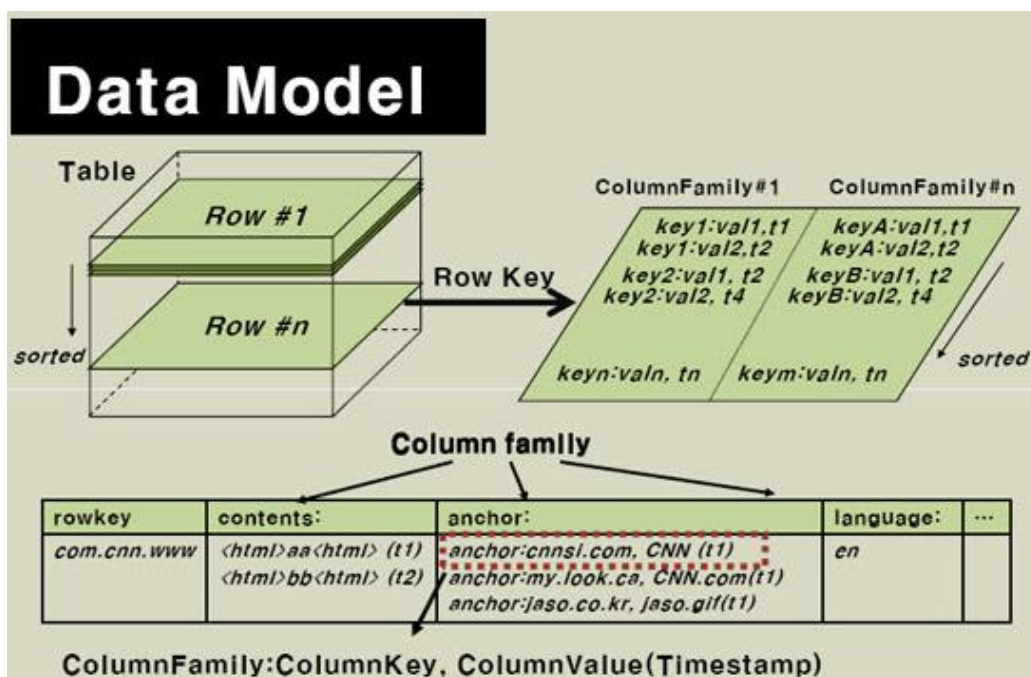


图 9.1.2

HBase 的逻辑数据模型：

Row Key	Time Stamp	Column:" info"	Column:" other"		column:"..."
"weric"	t9		"other:name"	"weric"	
	t8		"other:age"	"100"	
	t6	"<HTML>..."			"Data..."
	t5	"Text ..."			
	t3	"Other..."			

**Row Key（行键）：**

与 NoSQL 数据库一样，Row Key 是用来检索记录的主键。访问 HBase table 中的行，只有三种方式：

- 1）、通过单个 Row Key 访问。
- 2）、通过 Row Key 的 range 全表扫描。

3)、Row Key 可以使任意字符串（最大长度是 64KB，实际应用中长度一般为  $10 \sim 100$ bytes），在 HBase 内部，Row Key 保存为字节数组。

在存储时，数据按照\* Row Key 的字典序（byte order）排序存储\*。设计 Key 时，要充分排序存储这个特性，将经常一起读取的行存储到一起（位置相关性）。注意字典序对 int 排序的结果是 1, 10, 100, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, ..., 9, 91, 92, 93, 94, 95, 96, 97, 98, 99。要保存整形的自然序，Row Key 必须用 0 进行左填充。

行的一次读写是原子操作（不论一次读写多少列）。这个设计决策能够使用户很容易理解程序在对同一个行进行并发更新操作时的行为。

### **列族 Column Family:**

HBase 表中的每个列都归属于某个列族。列族是表的 Schema 的一部分（而列不是），必须在使用表之前定义。列名都以列族作为前缀，例如 courses:history、courses:math 都属于 courses 这个列族。

### **时间戳:**

HBase 中通过 Row 和 Columns 确定的一个存储单元称为 Cell。每个 Cell 都保存着同一份数据的多个版本。版本通过时间戳来索引，时间戳的类型是 64 位整型。时间戳可以由 HBase（在数据写入时自动）赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显示赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 Cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的管理（包括存储和索引）负担，HBase 提供了两种数据版本回收方式。一是保存数据的最后 n 个版本，二是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

### **Cell:**

Cell 是由 {row key, column(=< family> + < label>), version} 唯一确定的单元。Cell 中的数据是没有类型的，全部是字节码形式存储。

## 9.2、HBase 单节点安装

HBase 可以运行在单节点的模式下。通过单节点的安装可以快速的学习 HBase。单节点安装，不需要 Hadoop hdfs。可以通过配置将 hbase 的数据文件保存到指定的目录上。但 HBase 依赖于 JDK，所以，JDK 还是必须要安装且配置好环境变量的。

### 步 1、下载 hbase

可以使用 wget 通过以下地址下载 hbase：

<http://apache.fayea.com/hbase/1.3.1/hbase-1.3.1-bin.tar.gz>。也可以将下载好的 hbase-1.3.1-bin.tar.gz 通过 xftp 上传到 linux。

### 步 2、解压

```
$ tar zxvf hbase-1.3.1-bin.tar.gz
```

由于是非集群方式的安装，即单一节点，所以，我将 hbase 解压到了 `~` 目录下，即，如果当前用户是 wangjian 则解压到了 `/home/wangjian` 目录下。仅用于快速的学习 hbase。

### 步 3、配置环境变量

修改当前用户的环境变量，可以通过修改 `~/.bashrc` 文件即可。这样，这个环境变量，只能当前用户有效。

```
vim ~/.bashrc
```

添加以下配置信息：

```
export HBASE_HOME=/home/${username}/hbase-1.3.1
export PATH=$PATH:$JAVA_HOME/bin:$HBASE_HOME/bin
```

请根据实际情况，修改 `~` 为你的用户名称。

### 步 4、配置 HBase

配置 `HBASE_HOME/conf/hbase-env.sh` 文件。并添加 `JAVA_HOME` 环境变量。

```
export JAVA_HOME=/home/${username}/jdk1.8.0_131
```

配置 `HBASE_HOME/conf/hbase-site.sh` 文件，添加 hbase 数据保存的目录：

```
<configuration>

    <property>

        <name>hbase.rootdir</name>

        <value>/home/${username}/hbase_rootdir</value>

    </property>

    <property>

        <name>hbase.tmp.dir</name>

        <value>/home/${username}/hbase_tmp_dir</value>

    </property>

    <property>

        <name>hbase.zookeeper.property.dataDir</name>

        <value>/home/${username}/hbase_zookeeper_datadir</value>

    </property>

</configuration>
```

hbase.rootdir 用于配置 hfs, 即 hbase 的文件系统所保存的目录。hbase.tmp.dir 保存 hbase 临时文件保存的目录, 如果不设置将保存到/tmp 目录下。hbase 自带一个 zookeeper, hbase.zookeeper.property.dataDir 用于配置 hbase 自带的 zookeeper 数据文件保存的目录。

### 步 5、启动 HBase

执行\$HBASE\_HOME/bin 目录下的 start-hbase.sh 即可以启动 hbase。

```
$/ start-hbase.sh
```

使用 start-hbase.sh 启动 hbase 以后, 会存在一个 HMaster 进程。



### 【注意】

- 启动 Hbase 之前，必须要确定已经在/etc/hosts 文件中配置了本地 DNS。
- 使用 http://ip:16010 访问之前，必须要确定防火墙已经关闭。关闭防火墙使用：  
`$sudo systemctl stop firewalld.service`。禁用防火墙使用：`$sudo systemctl disable firewalld.service`。

现在就可以通过 `http://localhost:16010` 来访问 hbase 的管理页面了。访问成功的页面如图 9.2.1 所示：

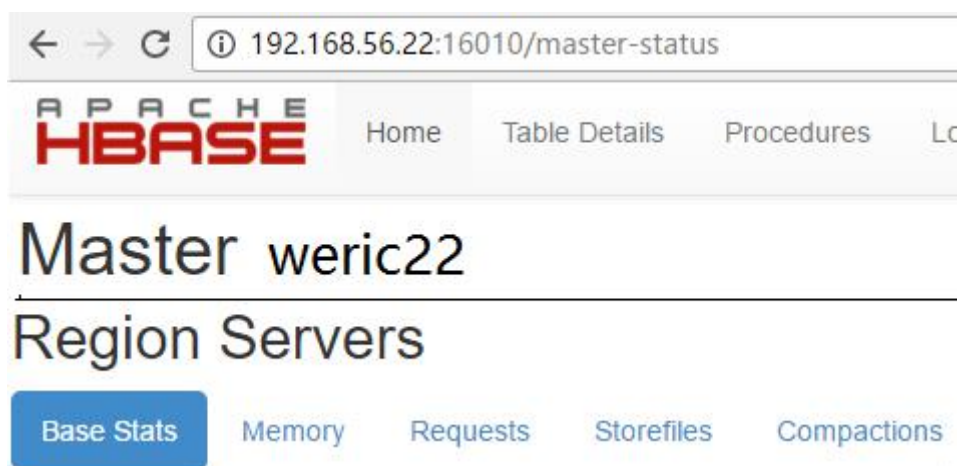


图 9.2.1

## 9.3、HBase Shell 的基本操作

HBase Shell 提供了大多数的 HBase 命令，通过 HBase Shell 用户可以方便地创建、删除及修改表，还可以向表中添加数据、列出表中的相关信息等。在启动 HBase 之后，用户可以通过下面的命令进入 HBase Shell 命令行模式，命令如下所示。HBase 脚本文件在 `$HBASE_HOME/bin/` 目录下。

```
$. /hbase shell
```

在登录成功 `hbase shell` 以后，可以使用 `help` 显示所有命令列表：

```
hbase(main):002:0> help
```

输入 help 可以看到命令分组。（注意命令都是小写，有大小写的区分）

Group Name	Commands
general	status, version
ddl	alter, create, describe, disable, drop, enable, exists, is_disable, is_enable, list
dml	count, delete, deleteall, get_counter, incr, put, scan, truncate
tools	assign, blance_switch, blancer, close_region, compact, flush, major_compact, move, split, unassign, ...
replication	add_peer, append_peer_tableCFs, disable_peer, disable_table_replication, enable_peer, ....
...	...

由于命令比较多，请读者自执行执行 help 查看所有命令。以下将演示一些简单命令的操作。如一些基本操作。

#### 查看版本 version:

```
hbase(main):008:0> version
1.3.1, r930b9 .. 35e77677a, Thu Apr 6 19:36:54 PDT 2017
```

#### 查看状态 status:

```
hbase(main):009:0> status
1 active master, 0 backup masters, 1 servers, 1 dead, 2.0000 average load
```

还有一些 DDL 操作如 create。

#### 创建一个表 create:

语法: create “表名”, “列族 1”, “列族 2”, “列族 N”

或是: create “表名”, {NAME=>“列族 1”, VERSIONS=>保存版本数量}, {...}

保存版本数量, 用于记录一个列族最多可以保存的历史记录。

```
hbase(main):005:0> create "stud", "info"
```



上面的代码中，stud 为表名，可以使用””双引号，也可以使用’ ’单引号。info 为列族。或是指定更多的信息，如使用 VERSIONS 指定保存的版本信息：

```
hbase(main):008:0> create "person", {NAME=>"info", VERSIONS=>3}
```

person 为表名，在 person 后面通过 {} 大括号声明列族为 info，版本信息为 3。在创建表以后，通过 list 可以显示所有的数据表。

### 修改表结构，添加一个新的列，使用 alter:

语法：alter “表名”，{NAME=>”列族名称”，VERSIONS=>3}

如果修改的列族不存在，则为添加一个新的列族，否则为修改已有列族信息。在修改之前，可以先通过 desc 查看表的信息，在修改之后，再通过 desc 查看表的信息。

查看表的信息，会列出这个表的所有列族信息

```
hbase(main):003:0> desc "stud"

Table stud is ENABLED

stud

COLUMN FAMILIES DESCRIPTION

{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '1', ...}
```

通过上面的列族信息可以看出，info 列族，的 versions 为 1，现在可以通过 alter 修改 info 列族的 versions 为 3：

```
hbase(main):004:0> alter "stud", {NAME=>"info", VERSIONS=>3}
```

然后，再通过 desc 查看 stud 表的信息：

```
hbase(main):005:0> desc "stud"

Table stud is ENABLED

stud

COLUMN FAMILIES DESCRIPTION

{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '3', ....}
```

此时，可以看到，versions 已经修改为 3 了。也可以通过 alter 添加一个新的列族，如果这个列族不存在则为创建一个新的列族：

```
hbase(main):006:0> alter "stud", {NAME=>"desc", VERSIONS=>3}
```

然后再通过 desc 查看 stud 表的结构:

```
hbase(main):007:0> desc "stud"
```

Table stud is ENABLED

stud

COLUMN FAMILIES DESCRIPTION

{NAME => 'desc', BLOOMFILTER => 'ROW', VERSIONS => '3', ...

{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '3', ....

可见, 已经添加了一个新的列族 desc。



### 【注意】

Hbase 所列族信息, 是按字典顺序来排序的, 所以 desc 的列族信息在 info 列族信息的前面。

## 删除一个列族

语法: alter “表名”, {NAME=>”列族名称”, METHOD=>”delete” }

删除一个列族, 同样使用 alter 关键字, 只是必须要传递 method=>”delete” 来指定的删除。删除一个列族:

```
hbase(main):009:0> alter "stud", {NAME=>"desc", METHOD=>"delete"}
```

再次查看这个表的信息:

```
hbase(main):010:0> desc "stud"
```

{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '3', .....

可见, 只剩下一个列族 info 了。即 desc 列族已经被删除。

## DML 操作

DML 操作, 用于向表中写入数据, 查询数据及删除数据。功能类似于 SQL 语句的 DML, 但命令与 SQL 不同。

## 插入数据

语法: put “表名”, “行键”, “列族:列名”, “具体值”

Hbase 写入的数据, 没有数据类型, 都是二进制数据。相同的列族属于同一行数据。

向表中写入一行数据:

```
hbase(main):012:0> put "stud","U001","info:name","Jack"
```

上例中, stud 为表名, U001 为行键, 即 rowkey 的值, info 后面的 name 为列的名称。Jack 为列值。在插入数据以后, 就可以通过表描述显示表中的所有数据了。

### 扫描表中的数据

语法: scan “表名”

scan 用于显示表中的所有数据, 类似于 SQL 语法中的 select \*from xxx。

```
hbase(main):013:0> scan "stud"
```

上例中最后一行 1 row(s).. 说明目前只有一行数据。现在就可以写入多行数据, 然后再通过 scan 查看里面的数据了。

再写入一行数据, 行键即 rowkey 与前面的行键相同:

```
hbase(main):014:0> put "stud","U001","info:age",23
```

然后再扫描表中的记录, 依然是一行数据, 因为一个行键, 代表的为一行数据。

```
hbase(main):015:0> scan "stud"
```

现在写入一个不同的行键, 然后再进行查询测试。

```
hbase(main):016:0> put "stud","U002","info:name","Mary"
```

然后再查看表中的记录, 已经发现, 为两行记录了, 因为行键即 rowkey 不同。

```
hbase(main):017:0> scan "stud"
```

可以通过 put 多次修改列的值, 首先查看 U001 和 info:name 的值:

```
hbase(main):018:0> scan "stud"
```

上面 info:name 的值为 Jack, 以下分别修改 N 次:

```
hbase(main):019:0> put "stud","U001","info:name","FirstName"
```

```
hbase(main):020:0> put "stud","U001","info:name","SecondName"
```

```
hbase(main):021:0> put "stud","U001","info:name","ThirdName"
```

然后再扫表，info:name 值为最后一次修改的记录：

```
hbase(main):022:0> scan "stud"

ROW      COLUMN+CELL
U001     column=info:name, timestamp=1500783878309, value=ThirdName
...
```

Hbase 的 versions 主要控制保存了版本记录，上面对列 info:name 的数据修改了 3 次，可以通过版本扫描，显示所有修改过的记录。

### 扫描时显示各版本的记录

可以通过指定 VERSIONS=>3 显示最近三次修改记录，使用 RAW 显示操作信息。在修改数据时，每一次都会记录一个 timestamp，即当前的时间。

```
hbase(main):034:0> scan "stud", {RAW=>true, VERSIONS=>3, COLUMNS=>"info"}

ROW      COLUMN+CELL
U001     column=info:age, timestamp=1500783145216, value=23
U001     column=info:name, timestamp=1500783878309, value=ThirdName
U001     column=info:name, timestamp=1500783873566, value=SecondName
U001     column=info:name, timestamp=1500783868549, value=FirstName
.....
2 row(s) in 0.0480 seconds
```

通过上面的查询，可以看到，VERSIONS=>3 显示了最近三次操作的记录。timestamp 为操作时间，以倒序显示。

### 扫描过虑

也可以在扫描时使用过虑功能。如指定从哪一个 rowkey 开始，则可以使用 startrow 过虑：

```
hbase(main):036:0> scan "stud", {COLUMNS=>"info", STARTROW=>"U002"}

ROW      COLUMN+CELL
U002     column=info:name, timestamp=1500783317379, value=Mary
```

也可以使用 ENDROW 指定结束的 rowkey，但请注意，ENDROW 的值是不包含的，即如果要到行键的值为 U003 且包含 U003，则应该指定 ENDROW=>"U004"：

```
hbase(main):043:0>scan
stud", {COLUMNS=>"info", STARTROW=>"U002", ENDROW=>"U004"}
ROW  COLUMN+CELL
U002      column=info:name, timestamp=1500783317379, value=Mary
U003      column=info:name, timestamp=1500784918046, value=Alex
```

上面的显示的范围为 >=U002 且 < U004。

### 值过虑

可以使用 ValueFilter 实现值过虑功能。，因为 HBase 中数据是以二进制开始保存的，所以比较方式为 binary 即二进制。如下查询值等于 Mary 的。

```
hbase(main):003:0> scan "stud", FILTER=>"ValueFilter(=, 'binary:Mary')"
```

上面显示值为 Mary 的，值得说明的是，如果多个列的值为 Mary 则会查询出多列的数据。值比较与列名无关。

### 包含

包含，类似于 contains，如果值中包含某个字符串，则可以查询出相关的记录。

以下是记录查询在值里面包含 Mary 的记录：

```
hbase(main):005:0>scan "stud", FILTER=>"ValueFilter(=, 'substring:Mary')"
```

### 列名过虑

可也可以使用 ColumnPrefixFilter 只查询某些指定的列，如以下显示所有 name 的列。

```
hbase(main):017:0> scan "stud", FILTER=>"ColumnPrefixFilter('name')"
```

使用 ColumnPrefixFilter 时，过虑的是列族后面的列名。而不是列族的名称。

### 多个条件进行组合

使用 AND 或 OR 关键字，可以串联多个过虑条件。如以下查询列名为 name 的，且值中包含 Mary 的记录。

```
hbase>scan stud",FILTER=>"ColumnPrefixFilter('name') \
    AND ValueFilter(=,'substring:Mary')"
```

### 行键过虑

可以使用 PrefixFilter 实现 rowkey 的过虑。如下只查询 rowkey 以 U001 开始的记录。

```
hbase(main):020:0> scan "stud",FILTER=>"PrefixFilter('U001')"
```

### 数据查询

语法: get “表名”, “行键” [, “列族:[列名]”]

行键即 rowkey 是必须要存在的, 列族和列名可省略。以下查询行键的值为 U001 的记录。

```
hbase(main):022:0> get "stud","U001"
```

以下只查询行键的值为 U001 的, 且列族名称为 info 的

```
hbase(main):023:0> get "stud","U001","info"
```

以下查询行键的值为 U001 且列的名称为 info:name 的

```
hbase(main):024:0> get "stud","U001","info:name"
```

### 修改数据

通过 put 存在相同的数据, 则为修改

```
hbase(main):020:0> put "stud","rk001","info:name","Jerry"
```

### 删除数据

语法: deleteall “表名”, “行键”, “列族:列名”

```
hbase(main):083:0> delete "stud","rk001","info:age"
```

### 删除整个行键中的所有数据

语法: deleteall “表名”, “行键”

```
hbase(main):087:0> deleteall "stud","rk001"
```

```
0 row(s) in 0.0190 seconds
```

### 删除整个表中的所有数据

语法: truncate “表名”

```
hbase(main):090:0> truncate "stud"
```

HBase 还有更多的操作命令，在此就不在一一赘述。读者可以有了上面的知识，完全可以通过查看 HBase 的 API 文档获取所有命令的使用方式。

## 9.4、HBase 伪分布式安装

HBase 伪分布式即单一节点的集群，是指 HBase 的数据保存到 Hadoop 伪分布式的 hdfs 上。这种情况下 HBase 会开启一个 HMaster 进程和一个 HRegion Server 进程。HBase 依然会使用自己内置的 zookeeper。

安装 HBase 伪分布式之前，请先安装一个单一节点的 Hadoop。假设 Hadoop 安装的目录为 /weric/hadoop-2.8.0，建议将 HBase 也安装到 /weric 目录下，即 /weric/hbase-1.3.1。

### 步 1：安装 Hadoop 伪分布式环境

关于 hadoop 伪分布式的安装和配置，请参考之前的章节。在安装完成以后启动 Hadoop。



#### 【注意】

- 本机到本机配置 ssh 免密码登录。
- 在启动 hadoop 之前，要通过 hdfs namenode -format 格式化 hdfs 文件系统。
- 关闭防火墙。
- 修改主机名称，如果之前已经修改可忽略。
- 测试伪分布式的 hadoop 可用，如上传 hdfs 一个文件。执行一个 wordcount 示例。

### 步 2：安装和配置 hbase

如果之前在本机安装过 hbase 请删除它，及之前的所有配置，删除之前配置的 hbase 的环境变量。现在我们将 hbase 与 hadoop 安装到相同的目录下，即 /weric 目录下。

解压 hbase 到/weric 目录下

```
$ tar -zxvf ~/hbase-1.3.1-bin.tar.gz -C /weric
```

配置 hbase 的环境变量

```
$sudo vim /etc/profile.d/hbase.sh
```

配置好 HBase 的环境变量以后, 可以使用 hbase version 查看 hbase 的版本信息

```
$ hbase version
```

```
HBase 1.3.1
```

配置 hbase-env.sh 文件, 添加 JAVA\_HOME 环境变量:

```
export JAVA_HOME=<你的 JDK 安装目录>
```

配置 hbase-site.xml, 此文件在\$HBASE\_HOME/conf 目录下。

```
<property>

  <name>hbase.cluster.distributed</name>

  <value>true</value>

</property>

<property>

  <name>hbase.rootdir</name>

  <value>hdfs://weric22:8020/hbase</value>

</property>
```

其中 hbase.cluster.distributed 用于配置 hbase 为分布式模式。hbase.rootdir 用于配置 hbase 数据保存保存到 hadoop 的 hdfs 上的目录。它的值为 hdfs://weric22:8020/hbase。其中 weric22 是主机的名称, 8020 为 hadoop2.8 默认现在就可以启动 hbase 了。

```
$ hbase-1.3.1/bin/start-hbase.sh

starting zookeeper, logging..

starting master, logging...

starting regionserver, logging...
```



通过上面的启动过程，可以发现，分布式的 Hbase 启动了三个进程，这三个进程分别是 HMaster 即 HBase 的主进程，HQuorumPeer 为 HBase 内置 zookeeper 的进程，HRegionServer 为 HBase 分布式环境下保存数据的进程。在真实分布式的环境下，HRegionServer 会有多个。HMaster 在外部 zookeeper 的协助下，也可以有多个，且一个为 Active 一个为 Backup。你还可以通过 jps 显示这些进程。

```
$ jps

5170 SecondaryNameNode
5429 NodeManager
7112 HRegionServer
6891 HQuorumPeer
4860 NameNode
5324 ResourceManager
6988 HMaster
7597 Jps
4975 DataNode
```

请注意区分上面哪些是 HBase 的进程，哪些是 Hadoop 的进程。

### 步 3、HBase 命令示例

现在，你就可向前面章节讲的一样，通过 hbase shell 操作 hbase 了。如以下操作示例：

```
hbase(main):002:0> create "stud","info"

=> Hbase::Table - stud

hbase(main):003:0> list

TABLE

stud

=> ["stud"]

hbase(main):004:0> put "stud","U001","info:name","Jack"
```

```

hbase(main):005:0> scan "stud"

ROW    COLUMN+CELL

U001   column=info:name, timestamp=1500816512881, value=Jack

hbase(main):006:0> put "stud","U001","info:name","Mary"

hbase(main):007:0> get "stud","U001"

COLUMN    CELL

info:name timestamp=1500816534554, value=Mary

hbase(main):008:0> get "stud","U001","info:name"

COLUMN    CELL

info:name timestamp=1500816534554, value=Mary

```

## 9.5、HBase JavaAPI 接口

同样的，通过 Java 代码，也可以操作 HBase。以下将展示，如何通过 Java 代码操作 HBase。



**【注意】**一般 HBase 和 Hadoop 安装到 Linux 服务器上，而开发 Java 代码一般使用 Eclipse 则是在 windows 上，此时为了可以正确的连接到 HBase 服务器，必须要配置 windows 上的 HOSTS 文件，即本地 DNS 文件，添加以下内容：192.168.56.22 weric22 即配置 weric22 对应的 ip 地址。

使用 maven 创建一个 java 项目，且添加以下依赖

(关于 Maven 的使用,请读者自行学习，也可以向本人索取学习资料。)

```

<dependency>

    <groupId>org.apache.hbase</groupId>

    <artifactId>hbase-client</artifactId>

    <version>1.3.1</version>

</dependency>

```

以下列示所有 HBase 中的所有表名：

```
@Test

public void listTables() throws Exception{

    //创建配置类，通过 HBaseConfiguration

    Configuration conf =

        HBaseConfiguration.create();

    //设置属性

    conf.set("hbase.rootdir","hdfs://weric22:8020/hbase");

    conf.set("hbase.zookeeper.quorum", "weric22:2181");

    //获取连接对象

    Connection con =

        ConnectionFactory.createConnection(conf);

    //获取数据操作对象

    Admin admin = con.getAdmin();

    if(admin instanceof HBaseAdmin){

        HBaseAdmin hadmin = (HBaseAdmin) admin;

        TableName[] tns = hadmin.listTableNames();

        for(TableName tn:tns){

            System.err.println("表名: "+tn.getNameAsString());

        }

    }

    admin.close();

}
```

在上面的代码中，通过 Configuration 设置连接 HBase 的信息，在获取到 HBaseAdmin 以后，通过 HBaseAdmin 操作 HBase 数据库。由于这些代码对于后面的操作都是相同的，所以，后面的开发过程中，这些代码将略去。

## 创建表

使用 HBaseAdmin 对象的 createTable(HTableDescriptor) 可以创建表。

```
HBaseAdmin hadmin = (HBaseAdmin) admin;

//创建表描述对象

HTableDescriptor htd = new
HTableDescriptor(TableName.valueOf("person"));

//创建列族对象

HColumnDescriptor columnDescriptor = new HColumnDescriptor("info");
columnDescriptor.setVersions(1, 3);

//给表添加一个列族

htd.addFamily(columnDescriptor);

//创建表

hadmin.createTable(htd);
```

## 查询表中的所有数据

类似于在命令行中，使用 scan 遍历表中的所有数据

```
// 通过 Connection 获取一个 Table 对象

Table table = con.getTable(TableName.valueOf("stud"));

// 声明 Scann 对象，用于配置一些查询的信息

Scan scan = new Scan();

// 查询

ResultScanner resultScanner = table.getScanner(scan);

// 遍历数据

for (Result result : resultScanner) {

    for (Cell cell : result.listCells()) {

        System.err.print("RowKey:" +
Bytes.toString(CellUtil.cloneRow(cell)));
```

```

        System.err.print("\tCellFamily:" +
            Bytes.toString(CellUtil.cloneFamily(cell)));

        System.err.print("-Qualifier:" +
            Bytes.toString(CellUtil.cloneQualifier(cell)));

        System.err.print("\tTimeStamp:" + cell.getTimestamp());

        System.err.print("\tValue:" +
            Bytes.toString(CellUtil.cloneValue(cell)));

        System.err.println();
    }
}

```

运行输出的效果如下：

```

RowKey:U001 CellFamily:info-Qualifier:age   TimeStamp:...   Value:23
RowKey:U001 CellFamily:info-Qualifier:name TimeStamp:...   Value:Mary
RowKey:U002 CellFamily:info-Qualifier:name TimeStamp:...
Value:Jerry

```

Scan 对象可以传递一些参数，如 startRow

```
Scan scan = new Scan("U002".getBytes());
```

也可以设置查询列的信息

```
scan.addColumn("info".getBytes(), "name".getBytes());
```

### 通过 RowKey 查询

只要在 Scan 中传递 Get 对象即可是 RowKey 查询

```
Scan scan = new Scan(new Get("U001".getBytes()));
```

### 根据值来进行查询

根据值来查询，使用 ValueFilter 对象，BinaryComparator 用于比较二进制数据。

```
Scan scan = new Scan();
```

```
//设置查询列的信息
```

```
BinaryComparator bc = new BinaryComparator("Mary".getBytes());
ValueFilter vf = new ValueFilter(CompareOp.EQUAL, bc);
scan.setFilter(vf);
```

### 使用正则表达式的查询

RegexStringComparator 用于执行正则表达式的查询。

```
RegexStringComparator bc = new RegexStringComparator(".*r.*");
ValueFilter vf = new ValueFilter(CompareOp.EQUAL, bc);
scan.setFilter(vf);
```

或使用 SingleColumnValueFilter 指定查询的列名：

```
RegexStringComparator bc = new RegexStringComparator(".*r.*");
SingleColumnValueFilter vf =
    new SingleColumnValueFilter(
        Bytes.toBytes("info"),
        Bytes.toBytes("name"),
        CompareOp.EQUAL, bc);
scan.setFilter(vf);
```

### 字符串包含查询

可以使用 SubStringComparator 查询包含的字符串。

```
ByteArrayComparable bc = new SubstringComparator("Mary");
```

### 前缀二进制比较器

BinaryPrefixComparator 是前缀二进制比较器。与二进制比较器不同的是，只比较前缀是否相同。以下查询 info:name 列以 Ma 为前缀的数据。

```
Scan scan = new Scan();
BinaryPrefixComparator comp
    = new BinaryPrefixComparator(Bytes.toBytes("Ma"));
SingleColumnValueFilter filter
```

```

        = new SingleColumnValueFilter(Bytes.toBytes("info"),
            Bytes.toBytes("name"), CompareOp.EQUAL, comp);
scan.setFilter(filter);

```

### 列值过滤器

SingleColumnValueFilter 用于测试值的情况为相等、不等、范围等等

下面一个检测列族 family 下的列 qualifier 的列值和字符串 "some-value" 相等的部分

```

Scan scan = new Scan();

SingleColumnValueFilter filter =
    new SingleColumnValueFilter(Bytes.toBytes("family"),
        Bytes.toBytes("qualifier"),
        CompareOp.EQUAL, Bytes.toBytes("some-value"));

scan.setFilter(filter);

```

### 排除过滤

SingleColumnValueExcludeFilter 跟 SingleColumnValueFilter 功能一样,只是不查询出该列的值。下面的代码就不会查询出 family 列族下 qualifier 列的值。

```

Scan scan = new Scan();

SingleColumnValueExcludeFilter filter
    = new SingleColumnValueExcludeFilter(Bytes.toBytes("family"),
        Bytes.toBytes("qualifier"),
        CompareOp.EQUAL, Bytes.toBytes("some-value"));

scan.setFilter(filter);

```

### 列族过滤器

FamilyFilter 用于过滤列族(通常在 Scan 过程中通过设定某些列族来实现该功能,而不是直接使用该过滤器)。

```

Scan scan = new Scan();

```

```
FamilyFilter filter
= new FamilyFilter(CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes("some-family")));
scan.setFilter(filter);
```

### 列名过滤器

QualifierFilter 用于列名 (Qualifier) 过滤。

```
QualifierFilter qff =
    new QualifierFilter(CompareOp.EQUAL,
    new BinaryComparator("name".getBytes()));
scan.setFilter(qff);
```

### 列名前缀过滤器

ColumnPrefixFilter 用于列名 (Qualifier) 前缀过滤，即包含某个前缀的所有列名。

```
Scan scan = new Scan();
ColumnPrefixFilter filter =
    new ColumnPrefixFilter(Bytes.toBytes("somePrefix"));
scan.setFilter(filter);
```

### 多个列名前缀过滤器

MultipleColumnPrefixFilter 与 ColumnPrefixFilter 的行为类似，但可以指定多个列名 (Qualifier) 前缀。

```
Scan scan = new Scan();
byte[][] prefixes = new byte[][]{
    Bytes.toBytes("prefix1"),
    Bytes.toBytes("prefix2")};
MultipleColumnPrefixFilter filter =
    new MultipleColumnPrefixFilter(prefixes); scan.setFilter(filter);
```



## 列范围过滤器

ColumnRangeFilter 该过滤器可以进行高效的列名内部扫描。

```
Scan scan = new Scan();

boolean minColumnInclusive = true;

boolean maxColumnInclusive = true;

ColumnRangeFilter filter =

new ColumnRangeFilter(

    Bytes.toBytes("minColumnName"), minColumnInclusive,

    Bytes.toBytes("maxColumnName"), maxColumnInclusive);

scan.setFilter(filter);
```

## 行键过滤器

RowFilter 行键过滤器，一般来讲，执行 Scan 使用 startRow/stopRow 方式比较好，而 RowFilter 过滤器也可以完成对某一行的过滤。

```
Scan scan = new Scan();

RowFilter filter =

    new RowFilter(CompareOp.EQUAL,

        new BinaryComparator(Bytes.toBytes("someRowKey1")));

scan.setFilter(filter);
```

## 分页过滤器

PageFilter 用于按行分页。必须要设置每次显示几行，及在 Scan 中设置开始的行值。

```
Scan scan = new Scan();

PageFilter pf = new PageFilter(5);

scan.setFilter(pf);

byte[] startRow

    = Bytes.add("U005".getBytes(), Bytes.toBytes("postfix"));
```

```
scan.setStartRow(startRow);  
  
ResultScanner resultScanner = table.getScanner(scan);
```

### 串联多个过滤器

可以使用 `FilterList` 将多个过滤器串联起来，组成 And 或 Or 的过滤器。

```
FilterList fl = new FilterList(Operator.MUST_PASS_ALL);  
  
fl.addFilter(new ValueFilter(CompareOp.EQUAL,  
    new BinaryComparator("Smith".getBytes())));  
  
fl.addFilter(new RowFilter(CompareOp.EQUAL,  
    new BinaryComparator("U005".getBytes())));  
  
scan.setFilter(fl);
```

由于过滤器比较多，更多的过滤器用法，将不再一一展示。

## 9.6、HBase 集群

在 HBase 群的情况下，也可以拥有两个 HMaster，一个为 Active 一个为 Backup。使用外部的 zookeeper 实现自动的切换。

配置 HBase 集群的前置条件：

启动 zookeeper 集群。

启动 hadoop/hdfs 集群。hbase 的数据是保存到 hadoop 的 hdfs 上。

之前，我们前面的章节(7.4)中，曾经集群过三台服务器的 hadoop。现在我们依然使用上面的这个环境。请先启动 zookeeper 和 hadoop 集群环境。

以下在 `weric11/192.168.56.11` 主机上配置，然后再 scp 到其他主机的上即可。

### 步 1、解压 hbase 到指定的目录下

如以下将 hBase 解压到 `/weric` 目录下。

```
$ tar -zxvf ~/hbase-1.3.1-bin.tar.gz -C /weric
```

### 步 2、修改 `hbase-env.sh`

配置 JAVA\_HOME 环境变量，修改 HBase\_zk 为 false。

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_131 #配置 JAVA_HOME
export HBASE_MANAGES_ZK=false #配置使得外部的 zookeeper
```

### 步 3、修改 hbase-site.xml 文件

在 hbase-site.xml 文件中配置 hbase 保存的数据在 hdfs 上。由于 hadoop 是一个高可靠的集群，所以指定的为一个 nameservice 的地址。

```
<configuration>

  <property>配置为 hdfs 的目录来保存 hbase 的数据

    <name>hbase.rootdir</name>

    <value>hdfs://ns1/hbase</value>

  </property>

  <property>配置为分布式

    <name>hbase.cluster.distributed</name>

    <value>true</value>

  </property>

  <property>配置 zookeeper 的地址

    <name>hbase.zookeeper.quorum</name>

    <value>weric11:2181,weric12:2181,weric13:2181</value>

  </property>

</configuration>
```

### 步 4、配置 Hbase 的 regionservers 文件

在 \$HBASE\_HOME/conf/regionservers 文件中配置 RegionServers。

```
weric11
weric12
weric13
```

### 步 5、添加 hdfs-site.xml 文件和 core-site.xml 文件

由于 hbase 有运行要读取 hadoop 的配置信息，所以应该将 hdfs-site.xml 文件和 core-site.xml 文件放到 \$HBASE\_HOME/conf/ 目录下，以便于让 hbase 获知 hadoop 的配置信息。

```
$ cp hadoop-2.8.0/etc/hadoop/hdfs-site.xml hbase-1.3.1/conf/
$ cp hadoop-2.8.0/etc/hadoop/core-site.xml hbase-1.3.1/conf/
```

也可以将上面两个文件中的配置在 hbase-site.xml 中再配置一次。

## 步 6、将 hbase scp 到其他的机器上

建议在拷贝之前，删除 \$HBASE\_HOME/docs 目录，这里面放的都是帮助文件，实在没有必要因此而浪费拷贝的时间。

```
$scp -r hbase-1.3.1/ weric12:/weric/
$scp -r hbase-1.3.1/ weric13:/weric/
```

## 步 7、配置环境变量

配置 HBase 的环境变量，且将环境变量的配置文件，也拷贝到其他服务器上去，然后再执行 source /etc/profile 让环境变量生效。

```
$ sudo vim /etc/profile.d/hbase.sh
```

在文件中添加以下内容：

```
#!/bin/sh

export HBASE_HOME=/weric/hbase-1.3.1

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

让环境变量生效：

```
$source /etc/profile
```

将环境变量的配置配置文件，拷贝到其他服务器上去，注意，由于 /etc/profile.d 目录为 root 所有，所以使用 root@ip 地址实现文件拷贝。

```
scp /etc/profile.d/hbase.sh root@weric202:/etc/profile.d/
scp /etc/profile.d/hbase.sh root@weric203:/etc/profile.d/
```

并让所有机器的环境变量生效。

## 步 8、启动 hbase

现在就可以在 weric11 主机上启动 hbase 了，执行 start-hbase.sh 脚本。

```
$ hbase-1.3.1/bin/start-hbase.sh
```

在启动以后，会在 weric11 服务器上，发现 HMaster 和 HRegionServer 两个服务，在其他的机上，发现 HRegionServer 服务。即表示已经启动成功。

也可以通过 <http://weric11:16010> 查看，如图 9.6.1 所示：



图 9.6.1

图 9.6.2 显示的是所有 HRegionServer 的集群信息。

Region Servers	
Base Stats	Memory Requests Storefiles
ServerName	
weric11,16020,1500993354151	
weric12,16020,1500993347553	
weric13,16020,1500993347634	
Total:3	

图 9.6.2

图 9.6.3 显示没有备份服务器。

# Backup Masters

ServerName
------------

Total:0
---------

图 9.6.3

## 步 9、启用高可靠

HBase 也可以有两个 HMaster，一个为 Active 一个为 Backup。现在在 weric12 上再启动一个 hmaster。

```
hbase-daemon.sh start master
```

再去查看，可以看到这个服务器为备份服务器，如图 9.6.4 所示。

# Backup Masters

ServerName	Port
------------	------

weric12	16000
---------	-------

Total:1
---------

图 9.6.4

## 步 10、登录 hbase 进行一个基本的操作

现在在任意的主机，登录 hbase shell 进行 CRUD 示例。

```
[wangjian@weric201 cluster]$ hbase shell
```

这些操作，和之前 HBase shell 操作相同。此处不在赘述。

## 步 11、用 Java 代码连接 hbase

如果在 windows 上的 Eclipse 上连接，请修改 HOSTS 文件添加主机影射：

```
192.168.56.11 weric11
```

```
192.168.56.12 weric12
```

192.168.56.13    weric13

此时，我们只需要设置 zookeeper 的集群地址，即可以连接 hbase,代码如下。

```
@Test

public void listTables() throws Exception {

    // 创建配置类，通过 HBaseConfiguration
    Configuration conf = HBaseConfiguration.create();

    // 设置属性
    conf.set("hbase.rootdir", "hdfs://ns1/hbase");
    conf.set("hbase.zookeeper.quorum",
"weric11:2181,weric12:2181,weric13:2181");

    // 获取连接对象
    Connection con = ConnectionFactory.createConnection(conf);

    // 获取数据操作对象
    Admin admin = con.getAdmin();

    if (admin instanceof HBaseAdmin) {

        HBaseAdmin hadmin = (HBaseAdmin) admin;

        TableName[] tns = hadmin.listTableNames();

        for (TableName tn : tns) {

            System.err.println("表名: " + tn.getNameAsString());

        }

    }

    admin.close();

    con.close();

}
```

如果显示了所有的表信息，即连接成功。其他操作，如同前面的章节一样。不再赘述。

## 9.7、导入数据到 Hbase

可以使用 sqoop 将数据库中的数据导入到 hbase 中去。使用 `--hbase-table` 等参数，用于指定 hbase 数据库的一系列参数。

### 1、现在存在以下的 mysql 表

```
mysql> create table users(  
    > id varchar(32) primary key,  
    > name varchar(30),  
    > pwd varchar(32)  
    > );  
  
mysql > insert into users values( 'U001' , ' Jack' , ' 1234' );  
  
mysql> insert into users values( 'U002' , ' Mike' , ' 1234' );
```

### 2、创建 hbase 表

```
hbase(main):028:0> create "users","info"
```

表名: users, 列族: info。

### 3、创建 shell 脚本将 mysql 中的 users 表中的数据，导入到 hbase

```
#!/bin/bash  
  
sqoop import \  
  
--connect jdbc:mysql://192.168.56.1:3306/weric3?characterEncoding=UTF-8 \  
  
--username root \  
  
-P \  
  
--table users \  
  
--hbase-table users \  
  
--column-family info \  
  
--hbase-row-key id
```

-P 用于指定在命令行运行时输入密码。



--table 用于指定 mysql 的表

--hbase-table 用于指定导入到 hbase 中的哪一个表中

--column-family 必须要指定 hbase 表的列族名称

--hbase-row-key 用于指定使用 mysql 表中的哪一个列做为 rowkey 的值。

建议添加-m(小写) 指定 Mapper 的个数。如 -m 1。

#### 4、执行脚本

```
$ chmod +x sqoop_to_hbase.sh
$ ./sqoop_to_hbase.sh
```

脚本执行以后，即会生成 MapReduce 任务。

#### 5、现在查询里面的数据

```
hbase(main):036:0> scan "users"
ROW          COLUMN+CELL
U001         column=info:name, timestamp=1501734942533, value=Jack
U001         column=info:pwd, timestamp=1501734942533, value=1234
U002         column=info:name, timestamp=1501734942533, value=Mike
U002         column=info:pwd, timestamp=1501734942533, value=1234
```

也可以将一个查询语句，导入到 hbase，如：

脚本如下：

```
#!/bin/bash
sqoop import \
--driver com.mysql.jdbc.Driver \
--connect jdbc:mysql://192.168.56.1:3306/weric3?characterEncoding=UTF-8 \
--username root \
--password 1234 \
--query "select id,name,age,addr from studs where \${CONDITIONS}" \
--split-by "id" \
```

```
--hbase-table users \  
--column-family info \  
--hbase-row-key id \  
-m 1
```

如果使用了 `--query` 参数，则必须要在 `where` 中添加 `$CONDITIONS` 参数。如果启动多个 mapper，即 `-m` 不是 1，则必须要指定 `--split-by` 参数，用于指定分页的字段。

## 9.8、小结

- ❖ HBase 是指 Hadoop DataBase，是面向列的数据库。
- ❖ 大，一个表可以保存上亿级别的数据。
- ❖ 利用 hdfs 实现分布式的存储。
- ❖ 保存的数据，都是二进制形式，没有数据类型。
- ❖ HBase 的主要进程是 HMaster 和 HRegionServer。
- ❖ HBase 伪分布配置时可以使用 HBase 内置的 zookeeper，也可以使用外部独立的 zookeeper，只要在 `hbase-env.sh` 中修改 `HBASE_MANAGER_ZK=false` 且在 `hbase-site.xml` 中添加配置 `hbase.zookeeper.quorum=ip:2181` 即可。即使是单一节点的 zookeeper 也可以使用。
- ❖ Java 代码连接 HBase，无论是伪分布式或是真分布式，只要配置连接到 zookeeper，即：

`hbaseConfiguration.set(“hbase.zookeeper.quorum”, “ip:2181”);` 即可以连接成功 hbase。但在连接之前一定要配置 HOSTS 文件，指定连接的 ip 地址中的主机名与 ip 地址的对应关系。

# 第 10 章 Hive

## 内容简介

- Hive 的体系结构和特点
- Hive 的命令
- Hive 表分区
- Hive 之 UDF 编程

hive 是基于 Hadoop 的一个数据仓库工具,可以将结构化的数据文件映射为一张数据库表,并提供简单的 sql 查询功能,可以将 sql 语句转换为 MapReduce 任务进行运行。其优点是学习成本低,可以通过类似于 SQL 的语句快速实现简单的 MapReduce 统计,不必开发专门的 MapReduce 应用,十分适合数据仓库的统计分析。

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具,可以用来进行数据提取转化加载(ETL),这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类似于 SQL 的查询语言,称为 HQL,它允许熟悉 SQL 的用户查询数据。同时,这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。Hive 的特点:

- ❖ 对仓库中的数据进行分析和计算。
- ❖ 建立在 Hadoop 之上。
- ❖ 一次写入多次读取。
- ❖ 可以将 HQL 语句转换成 MapReduce。
- ❖ Hive 是 SQL 语句分析引擎,将 Sql 语句转换成 MapReduce 并在 Hadoop 上的执行。
- ❖ Hive 表对应的 hdfs 的文件夹。
- ❖ Hive 的数据对应的是 Hdfs 的文件。

Hive 体系结构如图 10.1 所示：

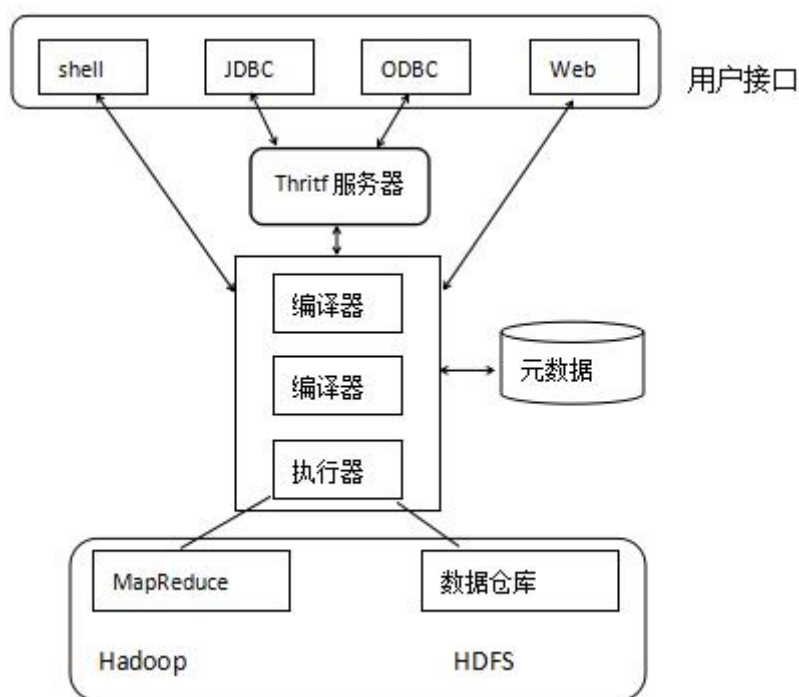


图 10.1

主要分为以下几个部分：

### 用户接口

用户接口主要有三个：CLI Client 和 WUI。其中最常用的是 CLI，Cli 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在节点，并且在该节点启动 Hive Server。WUI 是通过浏览器访问 Hive。

### 元数据存储

Hive 将元数据存储于数据库中，如 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。

目前已经支持以上的数据库包括：

支持的数据库	最小支持版本
MySQL	5.6.17
Postgres	9.1.13
Oracle	11g
MS SQL Server	2008 R2

### 解释器、编译器、优化器、执行器

解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行。

### Hadoop

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 someTable` 不会生成 MapReduce 任务）。

## 10.1、Hive1.x 的安装与使用

到本书发布时，hive 已经更新到 2.0 版本，本书的后面部分，会讲解 2.x 的应用。由于 1.x 可以快速上手，所以，此处先从 1.x 开始讲起。

### 使用 Hive 的前置条件

启动 zk、启动 hdfs 集群、启动 yarn。

### 步 1、下载 hive1.x

使用一台已经安装了 hadoop 的压力较小的服务器来安装 hive 即可。如果仅有一台服务器，都已经配置好了 Hadoop 伪分布式及 zookeeper 伪分布式也是可以进行练习的。下载地址：

```
http://mirror.bit.edu.cn/apache/hive/hive-1.2.2/apache-hive-1.2.2-bin.tar.gz
```

## 步 2、解压

可以将 hive 解压到任意的目录下，此处为了方便操作，将 hive 解压到安装集群 hadoop、hbase 的目录下。

```
$ tar -zxvf ~/apache-hive-1.2.2-bin.tar.gz -C /weric
```

## 步 3：运行 hive 命令

在解压完成以后，不用修改任何的配置文件。直接进入 bin 目录运行 hive 命令。

```
$ pwd  
  
/weric/apache-hive-1.2.2-bin/bin  
  
$ ./hive
```

更建立将 hive 的 bin 目录配置到环境变量，这样，就可以直接使用 hive 命令了。特别是后面的数据迁移工具，在执行向 hive 导入数据时，会使用到 hive 命令，所以，配置 hive 到 PATH 环境变量，还是有些必须要的。

配置环境变量，可以在/etc/profile.d 目录下，创建一个新的文件如 hive.sh 文件：

```
$sudo vim /etc/profile.d/hive.sh
```

在 hive.sh 文件中添加 hive 的环境变量如下：

```
#!/bin/sh  
  
export HIVE_HOME=<Hive install Driectory>  
  
export PATH=$PATH:$HIVE_HOME/bin
```

现在，就可以不用进入 Hive 的这安装目录去登录 Hive 的客户端了。可以在任意目前下，执行 hive 即可以登录 hive 的命令行模式：

```
$ hive
```

在登录 hive 命令行以后，执行类似于 mysql 的命令 show databases;即可以显示

当前所有的数据库。

```
hive> show databases;  
  
OK  
  
default  
  
Time taken: 1.391 seconds, Fetched: 1 row(s)
```

上面的命令,输出的结果显示为,存在一个默认的数据库。还可以执行 `show tables` 显示默认数据库下的所有表。

```
hive> show tables;
```

在启动 hive 以后,会在 hadoop hdfs 上出: `/user/hive/warehouse` 的目录,这就是用于保存 hive 数据的目录。到此为止,你的 hive 已经可以运行了。

## 10.2、Hive 命令

hive 的很多命令,就是 SQL 命令。但有些命令与 SQL 存在一些差异。以下是部分命令。

### 创建一个数据库

Hive 已经有了一个默认的数据库叫 `default`,现在你可以创建一个自己的数据库。

```
hive> create database weric;
```

在创建完成这个数据库以后,就可以使用 `show databases` 显示 hive 下的所有数据库。

```
hive> show databases;  
  
default  
  
weric
```

### 创建一个表

现在可以使用 `use weric;` 语法进入 `weric` 数据库,并创建一个表。

```
hive> create table stud(id int,name string);
```

在创建表以后，查看 hdfs 目录，会发现目录/user/hive/warehouse/weric.db/stud。这就是刚才创建的表。

### 显示表结构

你还可以使用 show create table 表名来显示表的结构。

```
① hive> show create table stud;
② OK
③ CREATE TABLE `stud` (
④   `id` int,
⑤   `name` varchar(30))
⑥ ROW FORMAT SERDE
⑦   'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
⑧ STORED AS INPUTFORMAT
⑨   'org.apache.hadoop.mapred.TextInputFormat'
⑩ OUTPUTFORMAT
⑪   'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
⑫ LOCATION
⑬   'hdfs://weric22/user/hive/warehouse/weric.db/stud'
⑭ TBLPROPERTIES (
⑮   'transient_lastDdlTime'='1501079369')
⑯ Time taken: 0.886 seconds, Fetched: 13 row(s)
```

上面显示的表结构中，行 9 用于声明这个文件的读取类。行 10, 11 声明输出数据类型。行 12, 13 操作的目录。行 14, 15 为这个表最后修改的时间，当前显示为创建的时间。

甚至可以向这个表中保存数据，由于 hive 中的表，对应的是文件，所以向 Hive 表中保存数据，就是向 hdfs 文件中保存数据。

### 向表中写入数据



```
hive> insert into stud values(2,'Mary');
```

当执行 insert 向表中写入数据时，默认会执行 mapper 任务向 hdfs 文件中写入数据。所以，上面的语句，你会发现执行了一个 mapper 任务。一般情况下，我们不会执行 insert 写入数据，而是从外部(非 hdfs 文件)或是内部(hdfs 文件)中导入数据。

现在可以执行 select 查询表中的数据

```
hive> select * from stud;
```

```
1    Jack
```

可以发现，里面已经存在了一行数据。再查询 hdfs 文件系统上的数据

```
hive> dfs -cat /user/hive/warehouse/werid.db/stud/*;
```

```
1Jack
```

在 Hive 客户端命令行上，可以直接执行 dfs 命令，类似于执行 hdfs dfs 命令，只是省去了 hdfs。通过上面的可以看出，里面已经写入了 1Jack 一行数据。且并没有分割符号。现在我们将里面的文件 000000\_0 下载到本地。

```
hive> dfs -get /user/hive/warehouse/werid.db/stud/* /home/wangjian/;
```

现在可以通过 vim 查看下载的文件：

```
$vim 000000_0
```

```
1^AJack
```

可见，1 与 Jack 之间是通过 ^A 做为分割符号的。现在我们可以创建一个表时，指定数据之间的分割符号。

### 创建一个表，并指定分割符号

```
hive> create table person(  
  >   id int,  
  >   name varchar(30)  
  > )  
  
  > row format delimited fields terminated by '\t';
```

注意上面的语句中最后的;(分号)。如果没有;(分号)就像 sql 语句没有结束一样，

所以在 Hive 中;(分号)也是语句结束的标记。上例中, 设置\t(制表)符号为字段数据之间分割的标记。

现在可以再写入一行数据测试一下数据在 hdfs 文件中的分割符号

```
hive> insert into person(id,name) values(100,'Mary');
```

查看数据的分割符号

```
hive> dfs -cat /user/hive/warehouse/weric.db/person/*;

100  Mary
```

通过上面的结果, 可以看出, 字段之间已经通过\t 进行了分割。

### load data 上传本地文件

load data 命令用于上传一个本地文件到 hive 的一个表中。其中 local 参数, 用于加载本地磁盘上的一个文件。如果没有 local 参数, 则为加载 hdfs 文件上的文件到 hive 的表中。

如要将一个文件中的数据导入到上述的 person 表中, 由于 person 表中字段之间的数据是用\t 分割的。所以, 可以先通过 vim 创建一个文件, 并用\t 分割里面的数据。

```
$vim person.txt

101      Jack

102      Mary

103      Mark

104      Alex
```

现在使用 load data 命令, 将数据导入到 person 表中去。

```
hive> load data local inpath '/home/wangjian/person.txt' into table person;
```

在执行上面的导入语句以后, 会在 hdfs 上发现 person.txt 这个文件, 这个文件所在的目录为/user/hive/warehouse/weric.db/person/person.txt。

现在查询里面的数据

```
hive> select * from person;

100      Mary
```

101	Jack
102	Mary
103	Mark
104	Alex

也可以上传一个 hdfs 文件到 person 表中, 使用 load data 命令, 不添加 local 即是从 hdfs 上加载数据。如先将某个文件上传到 hdfs 上。

```
hive> dfs -put /home/wangjian/person.txt /person.txt;
```

再使用 load data 将 hdfs 上的文件载入到 hive 的表中。需要注意的上, 上传完成以后, hdfs 上的文件会被移动到 hive 中, 即会删除原目录下的文件。

```
hive> load data inpath '/person.txt' into table person;
```

### 执行 mapreduce 任务

对数据进行统计 count, 会执行一个 mapreduce 任务。如以下代码。

```
hive> select count(*) from person;
OK
15
```

执行上面的查询语句, 你会发现一个完整的 mapreduce 过程。并最终直接将结果输出到控制台。

也可以将计算的结果输出到本地文件中。注意下例上输出的 count 是一个目录, 里面的文件才是输出的结果数据。

```
hive> insert overwrite local directory "/home/wangjian/count"
> select count(*) from person;
```

更可以将计算的结果, 输出到 hdfs 上去, 去掉 local 参数即可。同样的/count 是一个目录, 里面文件中的数据, 才是我们需要的结果。

```
hive> insert overwrite directory '/count' select count(1) from person;
```

### 执行一个过滤排序的查询, 会执行 mapreduce 任务

```
hive> select * from person where id>102 order by id;
```

```
Total MapReduce CPU Time Spent: 3 seconds 440 msec
```

```
104      Alex
```

```
104      Alex
```

```
104      Alex
```

```
105      Mark
```

```
105      Mark
```

```
105      Mark
```

### 字符统计示例

现在让我们再执行一个 Word Count 示例，统计上表，即 hdfs 文件中 name 的个数，直接执行以下命令即可。

```
hive> select name,count(name) from person group by name;
```

```
OK
```

```
Alexp    2
```

```
Jack     2
```

```
Mark     2
```

```
Mary     3
```

```
Time taken: 50.633 seconds, Fetched: 4 row(s)
```

到此为止，你已经执行了 Hive 的一些命令。除 `select *` 不会生成 MapReduce 之外，其他的命令，都会生成 mapreduce 任务。可见，Hive 大大的简化了 MapReduce 的开发。

## 10.3、使用 MySQL 数据库存储 metastore

metastore 是 hive 保存元数据的地方。默认的情况下，hive1 使用 derby（一个 Java 的内嵌的数据库）做为 metastore。在这种情况下，对于同一个目录，只能有一个用户登录客户端。可以替换成 mysql 数据库或其他数据库都是可以的。mysql 数据库，

可以安装到 linux 上，也可以安装到 windows 上，只要可以实现远程登录，就可以使用 mysql 数据库做为 metastore。当然，如果要使用 mysql 数据库，必须要把 mysql 数据库的驱动文件放到 \$HIVE\_HOME/lib 目录下。以下是具体的配置过程。

### 步 1、添加 mysql 的驱动器到 \$HIVE\_HOME/lib 目录下

使用 xftp 将 mysql 驱动器拷贝到 Linux 上。主要是放到 \$HIVE\_HOME/lib 目录下。

```
$ cp ~/mysql-connector-java-5.1.32-bin.jar \  
/weric/apache-hive-1.2.2-bin/lib/
```

### 步 2、修改 hive-site.xml 文件

在 \$HIVE\_HOME/conf 目录下，将 hive-default.xml.template 文件，拷贝成 hive-site.xml 文件，删除里面的所有内容，只保留 <configuration>，然后添加以下配置信息。拷贝文件：

```
$ cp hive-default.xml.template hive-site.xml
```

配置内容

```
① <configuration>  
②     <property>  
③         <name>javax.jdo.option.ConnectionURL</name>  
④  
<value>jdbc:mysql://192.168.56.1:3306/hive?characterEncoding=UTF-8&createDatabaseIfNotExist=true</value>  
⑤     </property>  
⑥     <property>  
⑦         <name>javax.jdo.option.ConnectionDriverName</name>  
⑧         <value>com.mysql.jdbc.Driver</value>  
⑨     </property>  
⑩     <property>  
⑪         <name>javax.jdo.option.ConnectionUserName</name>
```

```

⑫          <value>root</value>
⑬      </property>
⑭      <property>
⑮          <name>javax.jdo.option.ConnectionPassword</name>
⑯          <value>1234</value>
⑰      </property>
⑱ </configuration>

```

配置说明，上面第 3，4 行为配置连接 mysql 的字符串，其中 createDatabaseIfNotExists=true 是指如果数据库不存在，将会自动的创建数据库。行 7，8 为配置驱动器，行 11，12 为配置用户名，行 15，16 为配置登录密码。

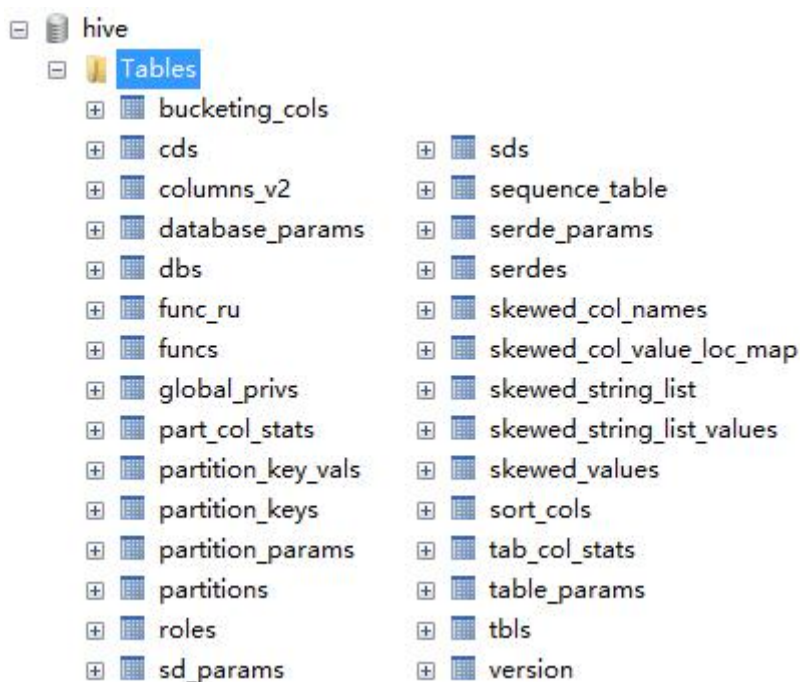


图 10.3.1

### 步 3、进入 hive 命令行

进入 hive 命令以后，数据库会自动被创建。现在可以使用 SqlYog 连接 Mysql 并查看表中的数据。如图 10.3.1 所示。

sds 表，保存了所有数据库的信息。

tbls 表保存了所有创建的表信息。

现在重复上面的执行的命令，查看表中数据的变化和 hdfs 上数据的变化。

## 10.4、Hive 外部表

如果数据已经存在于 hdfs 上，则可以通过创建外部表的方式与 hdfs 上的数据建立关系。默认的通过 create table 创建表为内部表(managed\_table)。在元数据 tbls 表中有一列 TBL\_TYPE, 如果值为 MANAGED\_TABLE 则为内部表。

通过 create external table 可以创建一个外部表。在创建时，需要指定 hdfs 上的一个目录。

现在先编辑一个 person.txt 文件, 且用\t 将数据进行分开。

```
$vim person.txt

101      Jack
102      Mary
103      Mark
104      Alex
```

在 hdfs 上创建一个目录/weric, put 这个文件到 hdfs 上的/weric 目录下。注意必须要拥有目录，不能是 hdfs 的根目录/。

```
$ hdfs dfs -mkdir /weric

$ hdfs dfs -put ~/person.txt /weric
```

然后创建一个外部表，并与/weric 目录建立关系

```
hive> create external table ext_person(id bigint,name string)

> row format

> delimited fields terminated by '\t'

> location '/weric';
```

在上面的代码中，name 可以使用 varchar(N) 也要以使用 string。注意最后 location ‘/weric’ 用于指定 hdfs 的外部目录。现在查看 tbls 表的 tbl\_type 列，值为 EXTERNAL\_TABLE，即为外部表。

查询外部表就像是查询内部表一样，可以获取数据结果

```
hive> select * from ext_person;
```

OK

101 Jack

102 Mary

103 Mark

104 Alex

## 10.5、Hive 表分区

### 1、背景

1)、在 Hive Select 查询中一般会扫描整个表内容，会消耗很多时间做没必要的工作。有时候只需要扫描表中关心的一部分数据，因此建表时引入了 partition 概念。

2)、分区表指的是在创建表时指定的 partition 的分区空间。

3)、如果需要创建有分区的表，需要在 create 表的时候传入可选参数 partitioned by 关键字。

### 2、技术细节

1)、一个表可以拥有一个或者多个分区，每个分区以文件夹的形式单独存在 hdfs 表文件夹的目录下。

2)、表和列名不区分大小写。

3)、分区是以字段的形式在表结构中存在，通过 describe table 命令可以查看到字段存在，但是该字段不存放实际的数据内容，仅仅是分区的表示。

4)、建表的语法（建分区可参见 PARTITIONED BY 参数）为：



```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name [(col_name data_type
[COMMENT col_comment], ...)] [COMMENT table_comment] [PARTITIONED BY
(col_name data_type [COMMENT col_comment], ...)] [CLUSTERED BY (col_name,
col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
[ROW FORMAT row_format] [STORED AS file_format] [LOCATION hdfs_path]
```

5)、分区建表分为2种，一种是单分区，也就是说在表文件夹目录下只有一级文件夹目录。另外一种是多分区，表文件夹下出现多文件夹嵌套模式。

a、单分区建表语句如：

```
hive> create table student(id int,name string)
> partitioned by (major string)
> row format delimited fields terminated by '\t';
```

上表创建三个列，id,name 和 major(专业)三个列。其中 major 为分区数据，即根据专业对学生信息进行分区。

b、双分区建表语句如：

```
hive> create table students(id int,name string)
> partitioned by (major string,grade string)
> row format delimited fields terminated by '\t';
```

上表创建两个分区一个为 major 即按专业分区，grade 再按年级进行分区。

双分区表，在表结构中新增加了 major 和 grade 两列。此时，保存到 hdfs 上的文件系统显示如下所示，注意，只要保存了数据以后，才会存在以下的结构，且会根据保存数据的不同，分区数据会显示为不同的目录。

```
/user/hive/warehouse/weric.db/students/major=Java/grade=2017
```

6)、添加分区数据

添加分区数据，是指在创建表时，已经创建了分区的情况下，添加分区数据。添加分区数据的语法是：

```
ALTER TABLE table_name ADD partition_spec [ LOCATION 'location1' ]
```

```
partition_spec [ LOCATION 'location2' ] ... partition_spec: : PARTITION
(partition_col      =      partition_col_value,      partition_col      =
partiton_col_value, ...)
```

用户可以用 ALTER TABLE ADD PARTITION 来向一个表中增加分区。当分区名是字符串时加引号。如：

```
hive> alter table student add partition(major='Java');
```

#### 7)、删除分区数据

```
ALTER TABLE table_name DROP partition_spec, partition_spec,...
```

用户可以用 ALTER TABLE DROP PARTITION 来删除分区及里面的数据。分区的元数据和数据将被一并被删除。如：

```
hive> alter table student drop partition(major="Java");
```

#### 8)、数据加载进分区表中语法

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]。如：
```

```
hive> load data local inpath '/home/wangjian/person.txt' into table
students

> partition(major='Java',grade='2017');
```

当数据被加载至表中时，不会对数据进行任何转换。Load 操作只是将数据复制至 Hive 表对应的位置。数据加载时在表下自动创建一个目录，文件存放在该分区下。

即当查询保存到分区中的数据时，数据的内部与加载之前的数据完全一样，如：

```
hive> dfs -cat \
/user/hive/warehouse/weric.db/students/major=Java/grade=2017/*;

101      Jack
102      Mary
103      Mark
104      Alex
```

但当执行查询时，会将分区的数据当做表的一部分查询出来，如：

```
hive> select * from students;

OK

101 Jack      Java      2017
102 Mary      Java      2017
103 Mark      Java      2017
104 Alex      Java      2017

Time taken: 0.167 seconds, Fetched: 4 row(s)
```

### 9)、基于分区的查询的语句

现在你可以使用 load data 导入更多的数据，其后在查询语句中使用分区的字段做为查询条件。

```
hive> select * from students where major='Oracle';

OK

301 Jack      Oracle    2017
302 Mary      Oracle    2017
303 Alex      Oracle    2017
304 Smith     Oracle    2017
```

### 10)、查看分区语句

可以通过 show partitions 查看一个表的分区信息。

```
hive> show partitions students;

OK

major=Java/grade=2017
major=Oracle/grade=2017

Time taken: 0.235 seconds, Fetched: 2 row(s)
```

## 3、分区示例

1、在 Hive 中，表中的一个 Partition 对应于表下的一个目录，所有的

Partition 的数据都存储在最子集的目录中。

2、总的说来 partition 就是辅助查询，缩小查询范围，加快数据的检索速度和对数据按照一定的规格和条件进行管理。

以下是创建一个班级表，拥有三个字段，id,name 和 classname 班级名称，以 classname 班级名称做为分区：

```
hive> create table students(id bigint,name string)
> partitioned by (classname string)
> row format delimited fields terminated by '\t';
```

在创建成功以后，数据保存到 hdfs 的 /user/hive/warehouse/werid.db/students 目录下。

现在保存数据到 students 表中去，先使用 vi 编辑一个文本文件，里面的内容如下：

```
vi studs.txt
101      Jack
102      Mary
...
```

然后使用 load data 将数据保存到 students 表中去，由于 students 表是有分区的，所在，在 load data 时，必须要指定分区信息，注意最后 partition(classname="two")。

```
hive> load data local inpath '/home/wangjian/studs.txt'
> into table students partition(classname="Java");
```

可以多将向里面 load 数据，并指定不同的分区信息。

```
hive> load data local inpath '/home/wangjian/studs.txt'
>into table students partition(classname="Oracle");
```

在 load 数据时，可以通过 overwrite 关键字重新设置某个分区下的所有数据：

```
hive> load data local inpath '/home/wangjian/studs2.txt'
```

```
> overwrite into table students partition(classname="Oracle");
```

现在查看 hdfs 上的目录结构如下:

```
hive> dfs -ls /user/hive/warehouse/weric.db/students;
```

Found 2 items

```
drwxr-xr-x      - wangjian  supergroup          0  2017-06-24  16:32
```

```
/user/hive/warehouse/weric.db/students/classname=Java
```

```
drwxr-xr-x      - wangjian  supergroup          0  2017-06-24  16:36
```

```
/user/hive/warehouse/weric.db/students/classname=Oracle
```

可见, 在 students 目录里面出现了两个子目录, 分别为 classname=Java 和 classname=Oracle。即为两个分区目录。

现在查询所有数据:

```
hive> select * from students;
```

```
101 Jack      Java
```

```
102 Mary      Java
```

```
301 Jack      Oracle
```

```
302 Mary      Oracle
```

```
303 Alex      Oracle
```

显示 partition 的信息:

```
hive> show partitions students;
```

```
classname=Java
```

```
classname=Oracle
```

创建具有多个分区的表, 如某个班级是哪一年级的哪一个专业, 以下即指定两个分区信息:

```
hive> create table classes(id bigint,name string)
```

```
> partitioned by(grade string,major string)
```

```
> row format delimited fields terminated by '\t';
```

导入数据, 请首先使用 vim 编辑一个文件如 cls1.txt, 然后再做导入。

```
hive> load data local inpath '/home/wangjian/cls1.txt'
```

```
>overwrite into table classes partition(grade='2016',major='computer');
```

导入数据以后, 会出现两层目录结构:

```
/user/hive/warehouse/werid.db/classes/grade=2016/major=computer
```

导入更多数据, 并查询:

```
hive> load data local inpath '/home/wangjian/cls2.txt'
```

```
>overwrite into table classes partition(grade='2017',major='ui');
```

```
hive> select * from classes;
```

1	Java1	2016	computer
2	Java2	2016	computer
9	UI1	2017	ui
10	UI2	2017	ui

添加一个新的分区, 如添加一个学院信息, 即哪一个班级, 属于哪一个学院:

```
hive> alter table classes add partition \
```

```
(grade='2015',major='computer');
```

添加一个新的分区, 是指在当前表的目录下, 创建分区目录出来的过程。本质上, 不是修改表结构, 而是添加数据目录的过程。在执行完成以后, 新的 hdfs 目录结构为:

```
/user/hive/warehouse/werid.db/classes/grade=2015/major=computer
```

删除一个分区:

```
hive> alter table classes drop \
```

```
partition(grade='2015',major='computer');
```

## 10.6、使用 sqoop 将数据导入 hive

将 mysql 数据库中的数据通过 sqoop 可以直接导入到 hive 中去, 前置条件是安装

sqoop\_1.x、配置 sqoop 环境变量 SQOOP\_HOME 和 PATH、安装 Hive1.x 及配置 HIVE 的 HIVE\_HOME 和 PATH 环境变量。关于 sqoop 的安装，请参考本书前面的章节。

sqoop 的 import 命令中，三个参数用于指定将数据导入到 hive 中。其中 hive-import 用于指定导入到 hive 仓库中，hive-overwrite 用于设置是否覆盖之前的导入，hive-table 用于指定导入到 hive 以后设置的表名，hive-database 用于指定 hive 的数据库名称。在使用 shell 脚本导入的情况下，建议接收一个参数，一是用于设置 table 参数，即导出的数据表。二是用于指定 hive-table 即导入到 hive 以后的表名移。由于是同一个参数，所以，导出的表名和导入的表名完全相同。

以下语句用于将 mysql 数据表中的数据，导入到 hive 仓库中，以下编写脚本：

```
$vim 01_import_to_hive.sql
```

脚本内容如下：

```
#!/bin/bash
sqoop import \
--connect \
jdbc:mysql://192.168.56.1:3306/qlu?characterEncoding=UTF-8 \
--username root \
--password 1234 \
--table $1 \
--hive-import \
--hive-overwrite \
--hive-table $1 \
--hive-database weric \
--fields-terminated-by '\t' \
-m 1
```

上例中，通过 hive-database 指定导入到 hive 的 weric 的数据库中。通过 hive-table 接收用户输入的参数来指定需要导入导出的表名称。现在设置这个文件为

可执行文件：

```
$ chmod +x 01_import_to_hive.sh
```

现在就可以执行这个文件，执行导入工作了：

```
$ ./01_import_to_hive.sh studs
```

后面的 studs 参数，将会传递给脚本的\$1 参数。即指定导入导出的表名。

还可以导入多个表，然后在 hive 中执行 sql 语句(本质是执行 mapreduce)进行分析。以下是另一个完整的过程。

### 步 1、分析数据

在 mysql 数据库中，存在图 10.6.1 所示数据结构。

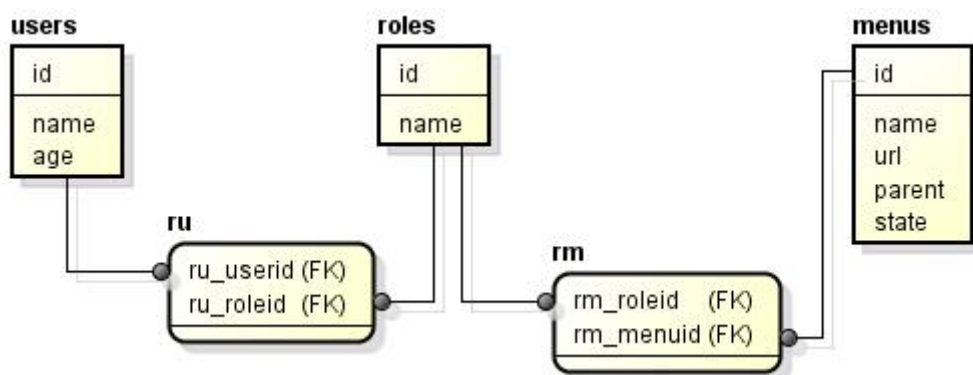


图 10.6.1

图 10.6.1 中，是一个典型的用户角色菜单权限的 E-R 图。查询某用户拥有某菜单的 SQL 语句为：

```
SELECT u. name, m. name
FROM users u INNER JOIN ru ON u.id=ru.userid
        INNER JOIN roles r ON r.id=ru.roleid
        INNER JOIN rm ON rm.roleid=r.id
        INNER JOIN menus m ON m.id=rm.menuid;
```

现在我们做的，就是将上表中的数据全部导入到 hive 中，然后在 hive 中执行上面的查询语句，即用 mapreduce 来执行查询的结果。



## 步 2、现在使用 sqoop 将 mysql 数据导入到 hive 中

为了方便操作，开发一个 shell 脚本，并通过命令行接收参数：

```
$ vim intohive.sh

#!/bin/bash

sqoop import \

--connect jdbc:mysql://192.168.56.1:3306/werice?characterEncoding=UTF-8 \

--username root \

--password 1234 \

--table $1 \

--hive-import \

--hive-overwrite \

--hive-table $1 \

--fields-terminated-by '\t' \

-m 1
```

现在执行这个脚本，传递需要导入的表，且 5 个表，顺序导入完成：

```
$ ./intohive.sh roles
```

## 步 3、登录 hive 查看已经导入的表

```
hive> show tables;

menus

rm

roles

ru

users
```

执行查询：

```
hive> SELECT u.name, m.name

> FROM users u INNER JOIN ru ON u.id=ru.userid
```

```
> inner join roles r on r.id=ru.roleid
> inner join rm on rm.roleid=r.id
> inner join menus m on m.id=rm.menuid;
```

Jack     系统管理

Jack     初始设置

Jack     班级设置

Jack     课程设置

查询完成即会显示某个用户可以拥有的菜单列表。

## 10.7、hive 函数

Hive 内部定义了很多的函数，这些函数都是通过 Hive 的 FunctionRegistry 类注册的。查看这个类的源代码，你将会发现大量类似于以下的代码：

```
system.registerGenericUDF("concat", GenericUDFConcat.class);
system.registerUDF("substr", UDFSubstr.class, false);
system.registerUDF("substring", UDFSubstr.class, false);
```

上面的代码，就是向 Hive 系统注册函数。在执行 HQL 语句的过程中，可以使用 Hive 已经定义的函数。可以使用 show functions; 查看 hive 中所有的内部函数：

```
show functions;
```

也可以使用 describe function 函数名称; 来查看具体某一个函数的用法。如：

```
describe function substr;
```

以下是 hive 中的函数及示例。

### 1)、关系运算符

在 FunctionRegistry 类中，可以找到注册关系运算符的源代码：

```
HIVE_OPERATORS.addAll(Arrays.asList(
    "+", "-", "*", "/", "%", "div", "&", "|", "^", "~",
```

```
"and", "or", "not", "!",  
"=", "==", "<=>", "!=", "<>", "<", "<=", ">", ">="));
```

通过 HIVE\_OPERATORS 关键字可知, 更准确的说法, 应该是操作符号。以下演示几个运算符的用法, 其中 "+", "-", "\*", "/", "%", "div", "&", "|", "^", "~" 符号大多用于 select 子句中。

除/运算符运算的结果为 double 类型

```
hive> select 10/3;  
  
3.3333333333333335
```

div 操作符号也是除操作, 运算的结果为整数类型

```
hive> select 10 div 3;  
  
3
```

%为取模操作, 即计算余数

```
hive> select 10 % 3;  
  
1
```

与&操作, 二进制运算, 两个值必须都为 1 都结果才是 1, 否则为 0。下例中 2 的二进制为 10, 1 二进制为 01 则 10 & 01 结果为 00, 即为 0。

```
hive> select 2 & 1;  
  
0
```

或 | 操作, 二进制运算, 两个值只要有一个为 1, 即为 1。2 的二进制为 10, 1 的二进制为 01, 则 10 | 01 的结果为 11, 即结果为 3。

```
hive> select 2 | 1;  
  
3
```

异或 ^ 运算, 二进制运算, 只要两个值不一样, 则为 1, 一样时为 0。下例中 10^01 的结果为 11 即结果为 3。

```
hive> select 2^1;  
  
3
```

按位取反~操作符号，二进制运算，~1 为 0，~0 为 1。2 的二进制为 10。所以~10 的值为 1111111111111111111111111111111101，即结果为-3。

```
hive> select ~2;

-3
```

操作符号 "=", "==", "<=>", "!=", "<>", "<", "<=", ">", ">="用在 where 子句中，用于比较。

相等比较 "=", "==", "<=>", 具有相同的功能。

```
hive> select * from students where id==301;

hive> select * from students where id=="301";

hive> select * from students where id<=>"301";
```

操作符号 "and", "or", "not", "!"。用于 where 子句中，进行与、或、非运算：

```
hive> select * from students where id=301 and major='Oracle';

hive> select * from students where id=301 or major='Java';
```

## 2)、更多函数

Hive 拥有丰富的内置函数。由于函数太多，以下仅为读者展示一部分。

### array

根据给定的元素，创建一个数组对象。

语法：array(n0, n1...)

以下创建一个字符串数组对象

```
hive> select array('Jack','Mary');

[ "Jack", " Mary" ]
```

以下创建一个整数的数组对象

```
hive> select array(1,2,3);

[1,2,3]
```

以下是由于包含一个字符串，所以创建一个字符串数组对象

```
hive> select array(1,'Jack');
```

```
[ "1", "Jack" ]
```

### **array\_contains**

判断给定的元素，是否在数组中存在

语法: `array_contains(array, value)` - Returns TRUE if the array contains value.

```
hive> select array_contains(array('Jack','Mary'),'Mary');
```

```
true
```

### **bin**

将一个 bigint 类型的数转成二进制

语法: `bin(n)` - returns n in binary

```
hive> select bin(2);
```

```
10
```

### **ceil**

返回大于当前数的最小整数

语法: `ceil(x)`

```
hive> select ceil(2.3);
```

```
3
```

### **current\_date**

返回当前时间

语法: `current_date()`

```
hive> select current_date();
```

```
2017-07-30
```

### **current\_timestamp**

返回当前的时间戳

语法: `current_timestamp()`

```
hive> select current_timestamp();
```

```
2017-07-30 21:37:44.271
```

### **explode**

将数组元素转换成多行显示

语法: explode(a)

```
hive> select explode(array("Jack", "Mary"));
```

```
Jack
```

```
Mary
```

也可以将一个 map 转换成多行显示

```
hive> select explode(map("name", "Jack", "age", 34));
```

```
name      Jack
```

```
age       34
```

### **get\_json\_object**

根据指定的路径，解析出 json 字符串中的对象，

json 中必须是 " " (双引号，即标准的 json 串)，path 部分必须是 \$. 开始。

如果只输入 \$ 表示当前整个 json 对象。

语法: get\_json\_object(json\_txt, path)

```
hive> select get_json_object("{\"name\":\"Jack\"}", '$.name');
```

```
Jack
```

```
hive> select get_json_object("[\"Jack\", \"Mary\"]", '$.[1]');
```

```
Mary
```

### **map**

创建一个 map 对象

语法: map(key0, value0, key1, value1...)

```
hive> select map("name", "Jack", "age", 34);
```

```
{"name": "Jack", "age": "34"}
```

## split

根据指定的正则表达式将字符串转换成数组，正则表达式中要使用\\(两个斜线)

语法: split(str, regex)

```
hive> select split('Jack Mary Rose','\\s+');  
["Jack","Mary","Rose"]
```

## 3)、使用 Hive 函数实现 WordCount

首先让使用 vim 创建一个文本文件。

```
$vim notes.txt
```

里面的内容如下:

```
Hello this is a text for something  
to tell you about how to process  
wordcount in hive.  
  
And you must be import into this file  
into hive.
```

现在创建一个 hive 表，只包含一个列，且分割符号为 ‘\r\n’ 即回车换行。

```
hive>create table notes(text string)  
  
>row format  
  
>delimited fields terminated by ‘\r\n’ ;
```

将 notes.txt 文件导入到 notes 表中。

```
hive> load data local inpath '/home/wangjian/notes.txt' into table notes;
```

测试查询是否是 5 行数据。

```
hive> select * from notes;  
  
OK  
  
Hello this is a text for something  
to tell you about how to process  
wordcount in hive.
```

```
And you must be import into this file  
into hive.  
Time taken: 0.159 seconds, Fetched: 5 row(s)
```

再创建一个表，用于保存每一个单词

```
hive> create table word(w string)  
  
    > row format  
  
    > delimited fields terminated by '\r\n';
```

现在我们需要将 notes 表中的每一行数据，按空格进行 split，然后再转换成行，保存到 word 表中去。以下是使用 insert overwrite 语句会先将 word 表中的数据删除然后再写入新的数据，如果使用 insert into 将会是追加数据。

```
hive> insert overwrite table word select explode(split(text, '\\s+')) from  
notes;
```

现在对 word 表进行 count 查询

```
hive> select count(w),w from word group by w;
```

这个查询，将会启动 MapReduce，并最终输出以下结果，部分内容略去。

```
1    And  
  
1    Hello  
  
...  
  
2    you
```

## 10.8、Hive 自定义函数

UDF(User Defined Function)是 Hive 中的自定义函数。当 Hive 中自有函数，不能达到我们的业务要求时，可以通过自定义 UDF 实现公司的业务逻辑。

hive 中已经存在很多的函数，这些函数，都是 hive 定义的，可以在 hive CLI 命令行下，查看已经存的 hive 函数



```
SHOW FUNCTIONS; 显示所有函数
```

```
DESCRIBE FUNCTION <function_name>; 显示某个函数的说明
```

```
DESCRIBE FUNCTION EXTENDED <function_name>; 显示某个函数的说明，及示例程序
```

关于更多的 hive 函数，可以通过以下官方网站查看：

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

Hive 中的函数，可以分为 UDF，UDAF（User- Defined Aggregation Funcation），UDTF（User-Defined Table-Generating Functions）。其中 UDAF 是用户自定义的聚合函数，UDTF 用于解决输入一行输出多行的问题。创建 UDF 函数流程如下：

- 1、自定义一个 Java 类
- 2、继承 UDF 类
- 3、重写 evaluate 方法
- 4、打成 jar 包
- 6、在 hive 执行 add jar 方法
- 7、在 hive 执行创建模板函数
- 8、在 hql 中使用

以下让我们先从开始一个简单的 UDF 开始学起。它实现两个数的和，返回计算的结果。

### 步 1、创建 Java 项目添加依赖

只需要添加 hive-exec 这一个依赖包即可。

```
<dependency>

    <groupId>org.apache.hive</groupId>

    <artifactId>hive-exec</artifactId>

    <version>1.2.2</version>

</dependency>
```

## 步 2、开发一个 Java 类

此类必须要继承 UDF 类，且包含 evaluate 方法。evaluate 方法可以根据业务的要求，定义接收的参数和返回的数据类型。evaluate 方法也可以定义很多的重载以实现不同的业务。

```
package cn.weric.hive;

import org.apache.hadoop.hive.ql.exec.UDF;

public class Calculate extends UDF {

    public int evaluate(int a, int b) {

        System.err.println("Hello:" + a + ", " + b);

        return a + b;

    }
}
```

## 步 3、将程序打成 jar 包

将项目打成 jar 包，如 calculate.jar，并通过 xftp 将 calculate.jar 上传到服务器上去。

## 步 4、执行 hive add

使用 add jar，类似于给 hive 设置 classpath 的目录。以便于让 hive 可以找到 UDF 所在的 jar 包。但这种方式，在下次重新启动 hive 时将失效。

```
hive> add jar /home/wangjian/calculate.jar;
```

如果希望添加的 jar 长期有效，可以将 jar 放到 HIVE\_HOME/auxlib 目录下。

## 步 5、添加函数

```
hive> create temporary function add as "cn.weric.hive.Calculate";
```

通过 create temporary 可以创建一个临时函数，函数名为 add，此函数将使用 cn.weric.hive.Calculate 类中的 evaluate 函数。通过 create temporary 创建的函数。当退出 hive 时，函数将失效。如果希望自定义的函数，长久有效，则可以通过修改 hive 的 FunctionRegistry 类实现。



### 【注意】

Hive 中的函数可以直接添加到 select 的语句中,类似于 mysql 可以直接使用,如:  
`select add(34.5);`

## 步 6、执行测试

执行 HQL 语句,使用 add 函数:

```
hive> select add(34,56);  
  
90  
  
Time taken: 0.397 seconds, Fetched: 1 row(s)
```

Hive 中的函数可重载,如以下用于计算的函数,即可以接收 int 类型也可以接收 String 类型:

```
package cn.weric.hive;  
  
import org.apache.hadoop.hive.ql.exec.UDF;  
  
public class Calculate extends UDF {  
  
    public int evaluate(int a, int b) {  
  
        System.err.println("Hello:" + a + ", " + b);  
  
        return a + b;  
  
    }  
  
    /** 通过重载实现另一个业务逻辑,如取邮件的地址*/  
  
    public String evaluate(String mail) {  
  
        if (mail.contains("@")) {  
  
            String str = mail.substring(mail.lastIndexOf("@") + 1);  
  
            return str;  
  
        }  
  
        return mail;  
  
    }  
  
}
```



### 【注意】

由于大多数 hive 的查询，会将生成的结果保存到 hdfs 上。如以下命令，会将执行的结果，保存到 hdfs 上。

```
hive> insert overwrite directory '/hiveout1' select add('Jack@125.com');
```

所以，在开发 UDF 时，可以接收 LongWritable/Text 等 hadoop 可以处理的数据类型。

如：

```
public Text evaluate(Text mail)..
```

但在开发本书时，在 hive1.2 版本时，即使返回 String 也可以正确的保存到 hdfs 上。

即使如此，笔者也建议使用 Hadoop 可以处理的类型。

再让我们来重审一下开发自定义 UDF 的过程：

- 1) 创建 Java 项目，并添加 Hive Query Language 的依赖。
- 2) 开发 Java 类继承 UDF 类。
- 3) 添加 evaluate 方法，并返回数据。
- 4) 使用 hive>add jar 命令，添加 jar 文件。
- 5) 使用 create temporary function 命令，添加一个函数。
- 6) 最后就是在 HQL 中使用函数。

## 10.9、小结

- ❖ Hive 是保存在 HDFS 上的数据库。
- ❖ 可以将 hdfs 数据影射成一张表，对这个表的操作，就是对 hdfs 数据文件的分析。
- ❖ hive 的数据对应的是 hdfs 的文件，hive 的数据库，对应的是 hdfs 的目录。
- ❖ hive 执行的语句被叫收 HQL 即 Hive Query Language。
- ❖ load data local inpath 是导入 Linux 文件系统中的文件到 hive 表中去。
- ❖ load data inpath 是将 hdfs 上的文件移动到 hive 表所在的目录下。

- ❖ hive 默认使用 derby 数据库做为 metedata。可以通过配置将 metedata 修改成 mysql。这样就支持多用户同时使用同一个 metedata 了。
- ❖ Hive 拥有丰富的内置函数。可以使用 show functions 查看所有函数，使用 desc function 函数名;查看具体某个函数的使用。
- ❖ 可以自定义 Hive 函数，即 UDF 开发。

# 第 11 章 Flume

## 内容简介

- Flume 简介
- Flume 的安装与配置
- Flume 部署

Flume 是 Cloudera 提供的一个高可用、高可靠、分布式的海量日志采集、聚合和传输的系统。Flume 支持定制各类数据源如 Avro、Thrift、Spooling 等。同时 Flume 提供对数据的简单处理,并将数据处理结果写入各种数据接收方,如将数据写入到 HDFS 文件系统中。

Flume 作为 Cloudera 开发的实时日志收集系统,受到了业界的认可与广泛应用。2010 年 11 月 Cloudera 开源了 Flume 的第一个可用版本 0.9.2,这个系列版本被统称为 Flume-OG (Original Generation)。随着 Flume 功能的扩展,Flume-OG 代码开始臃肿、核心组件设计不合理、核心配置不标准等缺点暴露出来,尤其是在 Flume-OG 的最后一个发行版本 0.94.0 中,日志传输不稳定的现象尤为严重。为了解决这些问题,2011 年 10 月 Cloudera 重构了核心组件、核心配置和代码架构,重构后的版本统称为 Flume-NG (Next Generation)。改动的另一原因是将 Flume 纳入 Apache 旗下,Cloudera Flume 改名为 Apache Flume。

Flume 的数据流由事件(Event)贯穿始终。事件是 Flume 的基本数据单位,它携带日志数据(字节数组形式)并且携带有头信息,这些 Event 由 Agent 外部的 Source 生成,当 Source 捕获事件后会进行特定的格式化,然后 Source 会把事件推入(单个或多个)Channel 中。你可以把 Channel 看作是一个缓冲区,它将保存事件直到 Sink 处理完该事件。Sink 负责持久化日志或者把事件推向另一个 Source。

Flume 以 agent 为最小的独立运行单位。一个 agent 就是一个 JVM。单 agent 由 Source、Sink 和 Channel 三大组件构成,如图 11.1 所示。

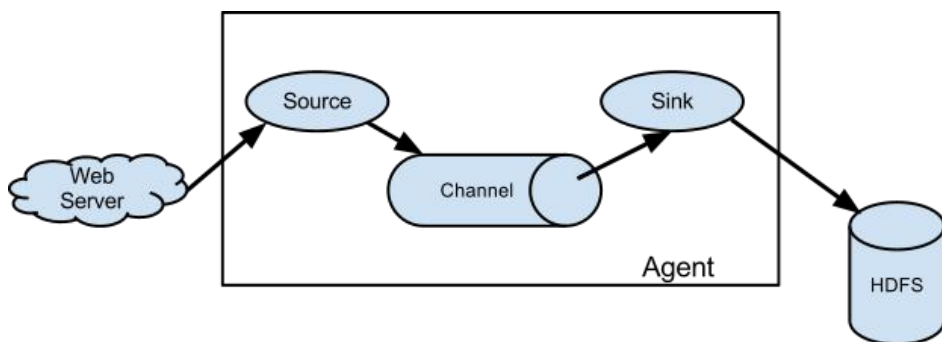


图 11.1

值得注意的是，Flume 提供了大量内置的 Source、Channel 和 Sink。不同类型的 Source、Channel 和 Sink 可以自由组合。组合方式基于用户的配置文件，非常灵活。比如：Channel 可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink 可以把日志写入 HDFS，HBase，甚至是另外一个 Source 等。Flume 支持用户建立多级流，也就是说，多个 agent 可以协同工作，并且支持 Fan-in(扇入)、Fan-out(扇出)、Contextual Routing、Backup Routes。如图 11.2 所示。

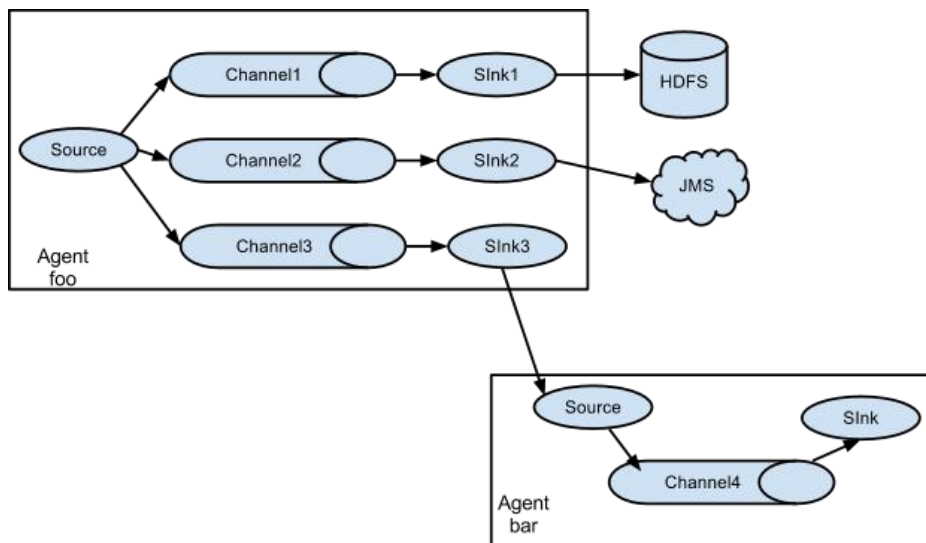


图 11.2

### Flume 的一些核心概念：

Agent 使用 JVM 运行 Flume。每台机器运行一个 agent，但是可以在一个 agent 中包含多个 Sources 和 Sinks。

Source 从 Client 收集数据，传递给 Channel。

Channel 连接 sources 和 sinks，Channel 缓存从 Source 收集来的数据。

Sink 从 Channel 收集数据，并将数据写到目标文件系统中。

## 11.1、Flume 的安装与配置

在安装 Flume 之前，请确认已经安装了 JDK 并正确配置了环境变量。

### 步 1：下载并解压 Flume

下载地址：

<http://www.apache.org/dyn/closer.lua/flume/1.7.0/apache-flume-1.7.0-bin.tar.gz>。将 flume 解压到 /weric 目录下。/weric 是之前安装 hadoop 等等其他软件的目录。

```
$ tar -zxvf ~/apache-flume-1.7.0-bin.tar.gz -C /weric
```

### 步 2：配置 flume 的环境变量

我个人比较喜欢创建一个独立的环境变量配置文件，所以使用 sudo 创建

```
$ sudo vim /etc/profile.d/flume.sh
```

在配置文件中，添加以下内容

```
#!/bin/sh

export FLUME_HOME=/weric/apache-flume-1.7.0-bin

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

环境变量生效

```
$ source /etc/profile
```

现在可以使用 version 测试 flume 的版本

```
$ flume-ng version

Flume 1.7.0
```

### 步 3：配置 flume-env.sh 文件

在 flume-env.sh 文件中配置 JAVA\_HOME 环境变量。



```
$ cp flume-env.sh.template flume-env.sh  
$ vim flume-env.sh  
export JAVA_HOME=/usr/local/java/jdk1.8.0_131
```

至此为止，flume 安装与配置已经完成。是不是非常的简单。以下是将部署两个基本的 flume agent 测试 flume。

## 11.2、部署 Flume agent

部署 flume 就是配置和启动一个 agent。flume 允许配置多个 agent，它们之间可以没有关系，也可以组成一个数据链。

### 1)、Avro

Flume 可以通过 Avro 监听某个端口并捕获传输的数据。现在我们配置一个 Source 用于监听某个端口的数据，并将它写出到 flume 的 log 中。

创建一个配置文件, 这个配置文件可以是任意的名称:

```
$vim netcat.conf
```

添加以下内容:

```
#配置三个组件  
a1.sources=r1  
a1.sinks=k1  
a1.channels=c1  
  
#配置 source  
a1.sources.r1.type=netcat  
a1.sources.r1.bind=weric22  
a1.sources.r1.port=4444  
  
#配置 sink  
a1.sinks.k1.type=logger
```

```
#配置 channel

a1.channels.c1.type=memory

a1.channels.c1.capacity=1000

a1.channels.c1.transactionCapacity=100

#绑定 source, sink 和 channel

a1.sources.r1.channels=c1

a1.sinks.k1.channel=c1
```

现在启动 Flume Agent

```
$ flume-ng agent -c conf -f netcat.conf -n a1
-Dflume.root.logger=INFO,console
```

-c conf 是指使用指定的配置文件。-f netcat.conf 是使用的配置文件。-n a1 为 agent 的名称。-Dflume.root.logger=INFO,console 即为输出的日志级别和目标。启动以后显示为:

```
17/07/31 23:28:03 INFO instrumentation.MonitoredCounterGroup: Component
type: CHANNEL, name: c1 started

17/07/31 23:28:03 INFO node.Application: Starting Sink k1

17/07/31 23:28:03 INFO node.Application: Starting Source r1

17/07/31 23:28:03 INFO source.NetcatSource: Source starting

17/07/31 23:28:03 INFO source.NetcatSource: Created

serverSocket:sun.nio.ch.ServerSocketChannelImpl[/192.168.56.22:4444]
```

可见, 已经开始监听 4444 端口。

现在打开另一个终端, 使用 telnet 登录 4444 端口, 并发送数据:

```
$ telnet weric22 4444

Trying 192.168.56.22...

Connected to weric22.

Escape character is '^['.
```

```
Hello
```

```
OK
```

```
Weric
```

```
OK
```

然后检查 flume 端接收到的信息

```
17/07/31 23:29:45 INFO sink.LoggerSink: Event: { headers:{} body: 48 65
6C 6C 6F 0D                                Hello. }
```

```
17/07/31 23:29:45 INFO sink.LoggerSink: Event: { headers:{} body: 57 65
72 69 63 0D                                Weric. }
```

通过上面的配置可以发现在 telnet 端输出的数据，都被 flume 接收并输出到控制台。即实现了第一个 flume 的部署。

**以下配置是将从端口接收到的数据，保存到 hdfs。**

```
#定义三个组件
```

```
al.sources=r1
```

```
al.channels=c1
```

```
al.sinks=k1
```

```
#定义 r1，即定义数据来源
```

```
al.sources.r1.type=netcat
```

```
al.sources.r1.bind=weric22
```

```
al.sources.r1.port=4444
```

```
#配置 sink，即输出的目标
```

```
al.sinks.k1.type=hdfs
```

```
al.sinks.k1.hdfs.path=hdfs://weric22:8020/flume/avro_to_hdfs
```

```
al.sinks.k1.hdfs.writeFormat=Text
```

```
al.sinks.k1.hdfs.fileType=DataStream
```

```
#配置中间的缓存
```

```
al.channels.cl.type=memory
al.channels.cl.capacity=1000
#将这三个组件组成一块
al.sources.r1.channels=c1
al.sinks.k1.channel=c1
```

现在启动 flume agent:

```
$ flume-ng agent -n a1 -c config -f 03_avro_to_hdfs.conf
-Dflume.root.logger=DEBUG,Console
```

使用 telnet 登录 weric22 主机的 4444 端口, 然后输入若干数据, 如下:

```
$ telnet weric22 4444
Jack <ENTER>
OK
....
```

现在可以查看 hdfs 上的数据:

```
$ hdfs dfs -ls /flume/avro_to_hdfs
Found 4 items
-rw-r--r--  /flume/avro_to_hdfs/FlumeData.1501858294240
-rw-r--r--  /flume/avro_to_hdfs/FlumeData.1501858327295
-rw-r--r--  /flume/avro_to_hdfs/FlumeData.1501858480559
-rw-r--r--  /flume/avro_to_hdfs/FlumeData.1501859145948
```

更可以查看里面保存的数据:

```
$ hdfs dfs -cat /flume/avro_to_hdfs/*
Mary
Alex
Jerry b
```

## 2)、Spool

Spool 用于监测配置的目录下新增的文件，并将文件中的数据读取出来。需要注意两点：拷贝到 spool 目录下的文件不可以再打开编辑、spool 目录下不可包含相应的子目录。现在创建 flume 的一个配置文件 spool.conf：

```
$ vim spool.conf
```

并添加以下配置信息：

```
#声明三个组件的名称

a2.sources=r1

a2.channels=c1

a2.sinks=k1

#声明 r1 即 source

a2.sources.r1.type=spooldir

a2.sources.r1.spoolDir=/weric/log

#声明 channel 即 c1

a2.channels.c1.type=memory

a2.channels.c1.capacity=1000

#声明 sinks 之 k1

a2.sinks.k1.type=hdfs

a2.sinks.k1.hdfs.path=hdfs://weric22:8020/flume/%Y%m%d%H

a2.sinks.k1.hdfs.writeFormat=Text

a2.sinks.k1.hdfs.fileType=DataStream

#由于上面使用了时间对象所以必须要设置为 true

a2.sinks.k1.hdfs.useLocalTimeStamp=true

#设置文件的前缀，默认值为 FlumeData.

a2.sinks.k1.hdfs.filePrefix=weric

#组合起来
```

```
a2.sources.r1.channels=c1  
a2.sinks.k1.channel=c1
```

上例中%Y%m%d%H是指输出年月日小时的格式的目录。fileType 的选项有三个，分别是：DataStream（原始数据流）、SequenceFile、CompressedStream。默认值为SequenceFile。hdfs.filePrefix 默认值为FlumeData，可以修改成任意的值。由于在配置中使用了%Y等时间表达式，所以必须要设置useLocalTimeStamp=true，否则会出现异常。现在启动这个 agent：

```
$ flume-ng agent -c apache-flume-1.7.0-bin/conf/ -f  
flume_config/spool.conf -n a2 -Dflume.root.logger=DEBUG,console
```

现在可以将文件直接拷贝到/weric/log 目录下了：

```
$ cp ~/note.txt /weric/log/notes.txt
```

flume 在完成上传以后，会修改文件名，默认的添加扩展为.COMPLETED：

```
$ ls  
notes.txt.COMPLETED
```

检查 hdfs 上的文件：

```
$ hdfs dfs -ls /flume/2017080122  
-rw-r--r-- ... /flume/2017080122/weric.1501579397641
```

上面略去了部分内容，其中 weric.1501579397641 即是采集完成放到 hdfs 上的文件。

## 11.3、小结

- ❖ Flume 是数据采集工具，一般主要用于采集日志信息。
- ❖ Agent 是 Flume 的核心，每一个 Agent 都包含 Source, Channel, Sink。
- ❖ 多个 Agent 可以组成一个链。
- ❖ Flume 定义了丰富的组件，只需要在配置文件中配置并组合它们即可。

# 附录 1:

## 1、如何在 Mapper 中获取读取的文件名

通过 FileSplit 对象:

```
InputSplit inputSplit = context.getInputSplit();  
FileSplit fileSplit = (FileSplit) inputSplit;  
String path = fileSplit.getPath().toString();//hdfs://server:ip/someFile
```

## 2、获取 NameNode 状态

在 HADOOP 集群的环境中。可以有多个 nameNode 以实现 HA(高可用)。可以使用以下命令，获取某个 NameNode 的状态:

```
$ hdfs haadmin -getServiceState nameNodeName
```

返回 active 为活跃状态，返回 standby 为装备状态。

## 3、检查 ResourceManager 状态

在 hadoop 集群中，可以有多个 ResourceManager，只有一个为 Active，可以使用以下命令，获取 ResourceManager 的状态:

```
$ yarn rmadmin -getServiceState resourceName  
active / standby
```

## 4、NameNode 的 ID 和 DataNode 的 id 不相同时导致 DataNode 启动不成功

以下是错误:

```
hadoop-doop-datanode-weric213.log  
  
java.io.IOException: Incompatible clusterIDs in
```

```
/hadoop/hadoop_tmp_dir/dfs/data: namenode clusterID =
CID-0c86a48e-7ba5-4f96-ab83-3bf0db3d5681; datanode clusterID =
CID-c1197f80-971b-4931-b958-5c120bcb96e5    #此地提示 namenode 和 id 和
dataNode 的 id 不相同。

    at
org.apache.hadoop.hdfs.server.datanode.DataStorage.doTransition(DataStorag
e.java:760)
```

修改: dataNode 节点所在服务器的 VERSION 文件, 修改里面的 clusterID=nameNode 的 id 即可。修改完成以后重新启动 DataNode。

5、NodeManager 自己退出的问题

查看日志:

```
2017-06-05 23:37:24,824 WARN org.apache.hadoop.ipc.Client: Failed to connect
to server: 0.0.0.0/0.0.0.0:8031: retries get failed due to exceeded maximum
allowed retries number: 10

java.net.ConnectException: 拒绝连接
```

```
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)

    at
sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:717)
```

查看官方文档:

yarn.resourcemanager.resource-tracker.addr ess	\${yarn.resourcemanager.hostname}:80 31
---	--

配置: yarn-site.xml

yarn.resourcemanager.resource-tracker.address = resourcemanager 的地址:8031。  
即配置 tracker.address 的地址为 8031。重新启动问题解决。



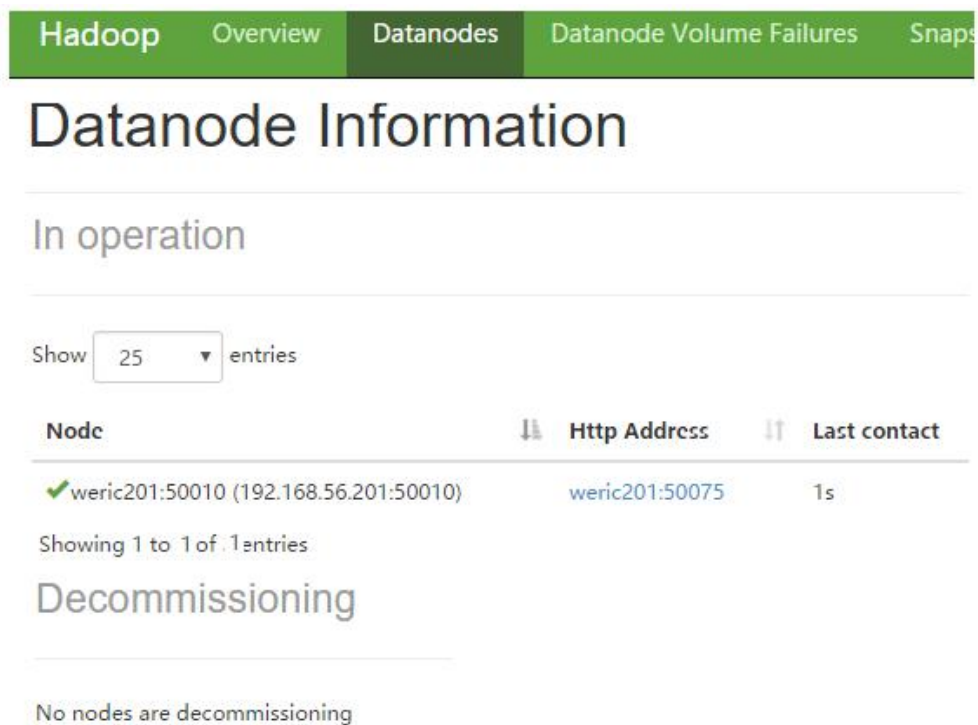
## 6、DataNode 没有显示的问题

在 Hadoop 集群启动成功以后，检查多台 DataNode 主机，发现多个 DataNode 都已经启动成功：

```
[wangjian@weric203 cluster]$ jps
3538 Jps
3460 DataNode
2598 NodeManager
2167 QuorumPeerMain
3199 JournalNode
```

但是通过 <http://192.168.56.201:50070> 查看 dataNode 的节点却只有一个，如图 12.6.1 所示：

<http://192.168.56.201:50070/dfshealth.html#tab-datanode>



The screenshot shows the Hadoop web interface for DataNode information. The top navigation bar includes 'Hadoop', 'Overview', 'Datanodes' (selected), 'Datanode Volume Failures', and 'Snapshots'. The main heading is 'Datanode Information'. Below it, the status 'In operation' is displayed. A dropdown menu shows '25' entries. A table lists the nodes:

Node	Http Address	Last contact
✓ weric201:50010 (192.168.56.201:50010)	weric201:50075	1s

Below the table, it says 'Showing 1 to 1 of 1 entries'. The 'Decommissioning' section shows 'No nodes are decommissioning'.

图 12.6.1

甚至有时会自动的切换服务器地址，如上：一会儿显示为 weric201:50075 一会又显示为 weric202:50075。如图 12.6.2 所示：

**Hadoop** Overview **Datanodes** Datanode Volume Failures Snaps

## Datanode Information

In operation

Show  entries

Node	Http Address	Last contact
✓ weric202:50010 (192.168.56.202:50010)	weric202:50075	1s

Showing 1 to 1 of 1 entries

## Decommissioning

No nodes are decommissioning

图 12.6.2

通过 `hdfs dfsadmin -report` 命令来查看，也只有一个 DataNode：

```
[wangjian@weric202 cluster]$ hdfs dfsadmin -report
Configured Capacity: 86905466880 (80.94 GB)
Present Capacity: 78347591680 (72.97 GB)
DFS Remaining: 78347563008 (72.97 GB)
DFS Used: 28672 (28 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
```

Pending deletion blocks: 0

---

**Live datanodes (1):**

Name: 192.168.56.201:50010 (weric201)

Hostname: weric201

Decommission Status : Normal

Configured Capacity: 28968488960 (26.98 GB)

DFS Used: 8192 (8 KB)

Non DFS Used: 3670679552 (3.42 GB)

DFS Remaining: 25297801216 (23.56 GB)

DFS Used%: 0.00%

DFS Remaining%: 87.33%

Configured Cache Capacity: 0 (0 B)

Cache Used: 0 (0 B)

Cache Remaining: 0 (0 B)

Cache Used%: 100.00%

Cache Remaining%: 0.00%

Xceivers: 1

Last contact: Sat Jun 17 22:16:03 CST 2017

这种情况发生的原因，有可能是：

`${hadoop.tmp.dir}/dfs/name/current/VERSION` 文件中的 `clusterID`=集群 id 的值，与 `${hadoop.tmp.dir}/dfs/data/current/VERSION` 文件中的 `cluseterID`=集群的 id 不一致的原因。

所以，只要将 `DataNode` 的文件：`VERSION` 中的 `clusterID` 设置的值与 `NameNode` 的 `VERSION` 文件的 `clusterID` 的值相同即可。

以下是 `DataNode` 的 `VERSION` 文件：

```
storageID=DS-8e1f79d3-6168-4471-9953-3b212689704d
```

#将这个值，修改成与 NameNode 的 VERSION 文件中的 clusterID 值相同即可

```
clusterID=CID-cbf8c179-4ee1-4609-888a-65cb1a00feb0
```

#同时，每一个 DataNode 的 id 应该不相同，否则只会显示一个 DataNode

#可以将后几位，修改成主机 ip 的地址

```
datanodeUuid=0a07635d-e490-43d3-a90a-6f98a991e203
```

```
storageType=DATA_NODE
```

```
layoutVersion=-57
```

现场还原：

先查看一下 namenode 中 /\$HADOOP\_HOME/tmp/dfs/name/current/下的 VERSION 文件中的 clusterID：

```
NameNode1: clusterID=CID-fee4dcb4-9615-42c0-bd46-d3b4acf02e61
```

```
NameNode2: clusterID=CID-d423e6c1-f45e-4e32-9a47-4272d69bcabc
```

同样，查看 datanode 下的 VERSION 文件：

/\$HADOOP\_HOME/tmp/dfs/data/current/目录下

```
DataNode1: clusterID=CID-d423e6c1-f45e-4e32-9a47-4272d69bcabc
```

```
DataNode2: clusterID=CID-fee4dcb4-9615-42c0-bd46-d3b4acf02e61
```

```
DataNode3: clusterID=CID-fee4dcb4-9615-42c0-bd46-d3b4acf02e61
```

现在只要将上面值，都修改成统一的 clusterID 值即可。同时需要说明的是，多个 NameNode，同一个集群必须要拥有相同的 clusterID 的值。

## 附录 2:

### 1: 使用 Shell 脚本一次启动所有 zookeeper 服务

在启动 zookeeper 集群时，一台一台的启动又麻烦又费事，为了简化启动和停止，可以使用：

ssh user@host 'command' 的方式启动指定的主机上的服务。现在可以开发一个脚本，一次启动多台服务器的上的 zookeeper。具体的脚本内容：

```
1 #!/bin/bash
2 servers="weric201 weric202 weric203"
3 for server in $servers
4 do
5     echo "当前正在启动$server....."
6     ssh wangjian@$server 'source /etc/profile;/cluster/zookeeper-3.4.10/bin/zkServer.sh
start'
7 done
```

解释：

行 2 定义一个数组，里面为三个值分别为 weric201,weric202,weric203。注意=符号两边不能有空格。数组元素使用空格分开。

行 6 为启动指定主机上的服务。之前请配置 ssh 免密码登录。

同样的，也可以开发一个脚本，一次停止所有的 zookeeper 服务。脚本内容如下：

```
#!/bin/bash
servers="weric201 weric202 weric203"
for server in $servers
do
```

```
echo "当前正在停止$server....."  
  
ssh wangjian@$server 'source /etc/profile;cluster/zookeeper-3.4.10/bin/zkServer.sh  
stop'  
done
```

注意，无论是执行哪台服务器上的脚本，只要用到环境变量，必须要在脚本里面加上 `source /etc/profile;`并用分号与后面的脚本分开。因为远程登录不会读取环境变量信息。

## 2：开发脚本启动 hadoop

根据符录 2.1 的提示，可以开发一个脚本一次完全启动 **hadoop** 集群。