第 3 章

CHAPTER 3

大数据基础

本书源代码下载

分布式机器学习为什么需求大数据呢？一方面随着海量用户数据的积累，单机运算已经不能满足需求。基于海量数据，机器学习训练之前需要做数据预处理、特征工程等，需要在大数据平台上进行；另一方面是机器学习训练过程的中间结果集可能会数据膨胀，依然需要大数据平台来承载，也就是说为了高性能的数据处理、分布式计算等，分布式机器学习是以大数据平台为基础的，所以下面我们来讲一下常用的大数据技术。

3.1 Hadoop 大数据平台搭建

Hadoop 是一种分析和处理大数据的软件平台，是一个用 Java 语言实现的 Apache 开源软件框架，在大量计算机组成的集群中实现了对海量数据的分布式计算。Hadoop 是大数据平台的标配，不管哪个公司的大数据部门，基本以 Hadoop 为核心。下面我们详细讲解 Hadoop 的原理和常用的一些操作命令。

3.1.1 Hadoop 原理和功能介绍

Hadoop 是一个由 Apache 基金会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，HDFS）。HDFS 有高容错性的特点，并且被设计并部署在低廉的（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 POSIX（relax）的要求，可以以流的形式访问（streaming access）文件系统中的数据。

Hadoop 最核心的框架设计有三大块：HDFS 分布式存储、MapReduce 计算引擎、Yarn 资源调度和管理。针对 Hadoop 这三大块核心，我们详细来讲一下。

1. HDFS 架构原理

HDFS 全称 Hadoop 分布式文件系统，其最主要的作用是作为 Hadoop 生态中各系统的存储服务。HDFS 为海量的数据提供了存储，可以认为它是一个分布式数据库，用来存储数据。HDFS 主要包含了 6 个服务：

1) NameNode

负责管理文件系统的 NameSpace 及客户端对文件的访问，NameNode 在 Hadoop 2 可以有多个，在 Hadoop 1 只能有一个，存在单点故障。HDFS 中的 NameNode 称为元数据节点，DataNode 称为数据节点。NameNode 维护了文件与数据块的映射表及数据块与数据节点的映射表，而真正的数据存储在 DataNode 上。NameNode 的功能如下：

(1) 维护和管理 DataNode 的主守护进程。

(2) 记录存储在集群中的所有文件的元数据，例如 Block 的位置、文件大小、权限和层次结构等，有两个文件与元数据关联。

(3) FsImage：包含自 NameNode 开始以来文件的 NameSpace 的完整状态。

(4) EditLogs：包含最近对文件系统进行的与最新 FsImage 相关的所有修改。它记录了发生在文件系统元数据上的每个更改。例如，如果一个文件在 HDFS 中被删除，那么 NameNode 就会立即在 EditLog 中记录这个操作。

(5) 定期从集群中的所有 DataNode 接收心跳信息和 Block 报告，以确保 DataNode 处于活动状态。

(6) 保留了 HDFS 中所有 Block 的记录及这些 Block 所在的节点。

(7) 负责管理所有 Block 的复制。

(8) 在 DataNode 失败的情况下，NameNode 会为副本选择新的 DataNode，平衡磁盘使用并管理到 DataNode 的通信流量。

(9) DataNode 则是 HDFS 中的从节点，与 NameNode 不同的是，DataNode 是一种商品硬件，它并不具有高质量或高可用性。DataNode 是一个将数据存储在本地文件 ext3 或 ext4 中的 Block 服务器。

2) DataNode

用于管理它所在节点上的数据存储：

(1) 这些是从属守护进行或在每台从属机器上运行的进程。

(2) 实际的数据存储在 DataNode 上。

(3) 执行文件系统客户端底层的读写请求。

(4) 定期向 NameNode 发送心跳报告及 HDFS 的整体健康状况，默认频率为 3 秒/次。

(5) 数据块(Block)：通常在任何文件系统中，都将数据存储为 Block 集合。Block 是硬盘上存储数据的最不连续的位置。在 Hadoop 集群中，每个 Block 的默认大小为 128M(此处指 Hadoop 2. x 版本，Hadoop 1. x 版本为 64M)，我们也可以通过如下配置 Block 的大小：dfs. block. size 或 dfs. blocksize=64M。

(6) 数据复制：HDFS 提供了一种将大数据作为数据块存储在分布式环境中的可靠方

法，即将这些 Block 复制以容错。默认的复制因子是 3，我们也可以通过如下配置并复制因子：

fs. replication＝3，每个 Block 被复制 3 次存储在不同的 DataNode 中。

3）FailoverController

故障切换控制器，负责监控与切换 NameNode 服务。

4）JournalNode

用于存储 EditLog；记录文件和数映射关系，操作记录，恢复操作。

5）Balancer

用于平衡集群之间各节点的磁盘利用率。

6）HttpFS

提供 HTTP 方式访问 HDFS 的功能。总地看来，NameNode 和 DataNode 是 HDFS 的核心，也是客户端操作数据需要依赖的两个服务。

2. MapReduce 计算引擎

MapReduce 计算引擎发布过两个版本，Hadoop 1 版本的时候叫 MRv1，Hadoop 2 版本的时候叫 MRv2。MapReduce 则为海量的数据提供了计算引擎，用里面的数据做运算，运算快。一声令下，多台机器团结合作进行运算，每台机器分一部分任务，同时并行运算。等所有机器分配的任务运算完，汇总报道，总任务全部完成。

1）MapReduce 1 架构原理

在 Hadoop 1. x 的时代，其核心是 JobTracker。

JobTracker：主要负责资源监控管理和作业调度。

（1）监控所有 TaskTracker 与 Job 的健康状况，一旦发现失败就将相应的任务转移到其他节点。

（2）与此同时，JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息报告给任务调度器，而任务调度器会在资源出现空闲时选择合适的任务使用这些资源。

TaskTracker：JobTracker 与 Task 之间的桥梁。

（1）从 JobTracker 接收并执行各种命令：运行任务、提交任务、Kill 任务和重新初始化任务。

（2）周期性地通过心跳机制，将节点健康情况和资源使用情况、各个任务的进度和状态等汇报给 JobTracker。

MapReduce 1 框架的主要局限：

（1）JobTracker 是 MapReduce 的集中处理点，存在单点故障，可靠性差。

（2）JobTracker 完成了太多的任务，造成了过多的资源消耗，当 MapReduce Job 非常多的时候，会造成很大的内存开销，这也增加了 JobTracker 失效的风险，这便是业界普遍总结出旧版本 Hadoop 的 MapReduce 只能支持上限为 4000 节点的主机，扩展性能差。

（3）可预测的延迟：这是用户非常关心的。小作业应该尽可能快地被调度，而当前基于 TaskTracker→JobTracker ping(heart beat)的通信方式代价和延迟过大，比较好的方式是 JobTracker → TaskTracker ping，这样 JobTracker 可以主动扫描有作业运行的

TaskTracker。

2）MapReduce 2 架构原理

Hadoop 2 版本之后有 Yarn，而 Hadoop 1 版本的时候还没有 Yarn。MapReduce 2 用 Yarn 来管理，下面我们来讲一下 Yarn 资源调度。

3. Yarn 资源调度和管理

1）ResourceManager

ResourceManager(RM)是资源调度器，包含两个主要的组件：定时调用器(Scheduler)及应用管理器(ApplicationManager，AM)。

(1) 定时调度器：根据容量、队列等限制条件，将系统中的资源分配给各个正在运行的应用。这里的调度器是一个“纯调度器”，因为它不再负责监控或者跟踪应用的执行状态等，此外，它也不再负责因应用执行失败或者硬件故障而需要重新启动的失败任务。调度器仅根据各个应用的资源需求进行调度，这是通过抽象概念“资源容器”完成的，资源容器(Resource Container)将内存、CPU、磁盘和网络等资源封装在一起，从而限定每个任务使用的资源量。总而言之，定时调度器负责向应用程序分配资源，它不做监控及应用程序的状态跟踪，并且它不保证由于应用程序本身或硬件出错而重新启动执行失败的应用程序。

(2) 应用管理器：主要负责接收作业，协助获取第一个容器用于执行 AM 和提供重启失败的 AM container 服务。

2）NodeManager

NodeManager 简称 NM，是每个节点上的框架代理，主要负责启动应用所需的容器，监控资源(内存、CPU、磁盘和网络等)的使用情况并将之汇报给定时调度器。

3）ApplicationMaster

每个应用程序的 ApplicationMaster 负责从 Scheduler 申请资源，并跟踪这些资源的使用情况及监控任务进度。

4）Container

Container 是 Yarn 中资源的抽象，它将内存、CPU、磁盘和网络等资源封装在一起。当 AM 向 RM 申请资源时，RM 为 AM 返回的资源便是用 Container 表示的。

了解了 Hadoop 的原理和核心组件，我们讲解如何安装、部署和搭建分布式集群。

3.1.2 Hadoop 安装部署

Hadoop 拥有 Apache 社区版和第三方发行版 CDH，Apache 社区版的优点是完全开源并可免费使用社区活跃文档，其资料翔实。缺点是版本管理比较混乱，各种版本层出不穷，很难选择，并且在选择生态组件时需要大量考虑兼容性问题、版本匹配问题、组件冲突问题和编译问题等。集群的部署、安装及配置复杂，需要编写大量配置文件分发到每台节点，容易出错，效率低。集群运维复杂，需要安装第三方软件辅助。CDH 版是由第三方 Cloudera 公司基于社区版本做了一些优化和改进，稳定性更强一些。CDH 版分免费版和商业版。

CDH 版的安装可以使用 Cloudera Manager(CM)通过管理界面的方式来安装,非常简单。Cloudera Manager 是 Cloudera 公司开发的一款大数据集群安装部署利器,这款利器具有集群自动化安装、中心化管理、集群监控和报警等功能,使得安装集群从几天的时间缩短为几小时以内,运维人员从数十人降低到几人之内,极大地提高了集群管理的效率。

不管是 CDH 版还是 Apache 社区版,我们都是使用 tar 包来手动部署,所有的环境需要我们一步步来操作,Hadoop 的每个配置文件也需要我们手工配置,通过这种方式安装的优势是比较灵活,集群服务器也不需要连外网,但这种方式对开发人员的要求比较高,对各种开发环境和配置文件都需要了解清楚。不过这种方式更方便我们了解 Hadoop 的各个模块和工作原理。

下面我们使用这种方式来手动地安装分布式集群,我们的例子是部署 5 台服务器,用两个 NameNode 节点做 HA,5 个 DataNode 节点,两个 NameNode 节点也同时作为 DataNode 使用。一般当服务器数量不多的时候,为了尽量地充分利用服务器的资源,NameNode 节点可以同时是 DataNode。

安装步骤如下:

1. 创建 Hadoop 用户

1) useradd hadoop

```
#设密码
passwd hadoop
#命令
usermod -g hadoop hadoop
```

2) vi/root/sudo

```
#添加一行
hadoop ALL=(ALL) NOPASSWD:ALL
chmod u+w /etc/sudoers
```

3) 编辑/etc/sudoers 文件

```
#也就是输入命令
vi /etc/sudoers
#进入编辑模式,找到这一行
root ALL=(ALL) ALL
#在它的下面添加
hadoop ALL=(ALL) NOPASSWD:ALL
#这里的 hadoop 是你的用户名,然后保存并退出
```

4) 撤销文件的写权限

```
#也就是输入命令
chmod u-w /etc/sudoers
```

2. 设置环境变量

```
#编辑/etc/profile 文件
vim /etc/profile
```

输入以下配置,如代码 3.1 所示。

【代码 3.1】 环境变量

```
export JAVA_HOME = /home/hadoop/software/jdk1.8.0_121
export SPARK_HOME = /home/hadoop/software/spark21
export SCALA_HOME = /home/hadoop/software/scala - 2.11.8
export SQOOP_HOME = /home/hadoop/software/sqoop
export HADOOP_HOME = /home/hadoop/software/hadoop2
export PATH = $ PATH: $ HADOOP_HOME/bin
export PATH = $ PATH: $ HADOOP_HOME/sbin
export HADOOP_MAPARED_HOME = $ {HADOOP_HOME}
export HADOOP_COMMON_HOME = $ {HADOOP_HOME}
export HADOOP_HDFS_HOME = $ {HADOOP_HOME}
export YARN_HOME = $ {HADOOP_HOME}
export HADOOP_CONF_DIR = $ {HADOOP_HOME}/etc/hadoop
export HIVE_HOME = /home/hadoop/software/hadoop2/hive
export PATH = $ JAVA_HOME/bin: $ HIVE_HOME/bin: $ SQOOP_HOME/bin: $ PATH
export CLASSPATH = .: $ JAVA_HOME/lib/dt.jar: $ JAVA_HOME/lib/tools.jar
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL
export FLUME_HOME = /home/hadoop/software/flume
export PATH = $ PATH: $ FLUME_HOME/bin
export HBASE_HOME = /home/hadoop/software/hbase - 0.98.8 - hadoop2
export PATH = $ PATH: $ HBASE_HOME/bin
export SOLR_HOME = /home/hadoop/software/solrcloud/solr - 6.4.2
export PATH = $ PATH: $ SOLR_HOME/bin
export M2_HOME = /home/hadoop/software/apache - maven - 3.3.9
export PATH = $ PATH: $ M2_HOME/bin
export PATH = $ PATH:/home/hadoop/software/apache - storm - 1.1.0/bin
export OOZIE_HOME = /home/hadoop/software/oozie - 4.3.0
export SQOOP_HOME = /home/hadoop/software/sqoop - 1.4.6 - cdh5.5.2
export PATH = $ PATH: $ SQOOP_HOME/bin
#按:wq 保存,保存后环境变量还没有生效,执行以下命令才会生效
source /etc/profile
#然后修改 Hadoop 的安装目录为 Hadoop 用户所有
chown - R hadoop:hadoop /data1/software/hadoop
```

3. 设置 local 无密码登录

```
su - hadoop
cd ~/.ssh #如果没有.ssh 则 mkdir ~/.ssh
ssh - keygen - t rsa
cd ~/.ssh
```

```
cat id_rsa.pub >> authorized_keys
sudo chmod 644 ~/.ssh/authorized_keys
sudo chmod 700 ~/.ssh
#然后重启 sshd 服务
sudo /etc/rc.d/init.d/sshd restart
```

有些情况下会遇到下面所示报错,可以用下面所示的方法来解决。

常见错误:

```
ssh -keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Could not create directory '/home/hadoop/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
open /home/hadoop/.ssh/id_rsa failed: Permission denied.
Saving the key failed: /home/hadoop/.ssh/id_rsa.
```

解决办法:

```
在 root 用户下操作 yum remove selinux*
```

4. 修改/etc/hosts 主机名和 IP 地址的映射文件

```
sudo vim /etc/hosts
#增加
172.172.0.11   data1
172.172.0.12   data2
172.172.0.13   data3
172.172.0.14   data4
172.172.0.15   data5
```

5. 设置远程无密码登录

使用 Hadoop 用户:

每台机器先本地无密钥部署一遍,因为我们搭建的是双 NameNode 节点,需要从这两个服务器上把 authorized_keys 文件复制到其他机器上,主要目的是使 NameNode 节点可以直接访问 DataNode 节点。

把双 NameNode HA 的 authorized_keys 复制到 slave 上。

从 NameNode1 节点上复制:

```
scp authorized_keys hadoop@data2:~/.ssh/authorized_keys_from_data1
scp authorized_keys hadoop@data3:~/.ssh/authorized_keys_from_data1
scp authorized_keys hadoop@data4:~/.ssh/authorized_keys_from_data1
scp authorized_keys hadoop@data5:~/.ssh/authorized_keys_from_data1
```

然后从 NameNode2 节点上复制:

```
scp authorized_keys hadoop@data1:~/.ssh/authorized_keys_from_data2
scp authorized_keys hadoop@data3:~/.ssh/authorized_keys_from_data2
scp authorized_keys hadoop@data4:~/.ssh/authorized_keys_from_data2
scp authorized_keys hadoop@data5:~/.ssh/authorized_keys_from_data2
```

6. 每台都关闭机器的防火墙

```
#关闭防火墙
sudo /etc/init.d/iptables stop
#关闭开机启动
sudo chkconfig iptables off
```

7. jdk 安装

因为 Hadoop 是基于 Java 开发的，所以我们需要安装 jdk 环境：

```
cd /home/hadoop/software/
#上传
rz jdk1.8.0_121.gz
tar xvzf jdk1.8.0_121.gz
```

然后修改环境变量并指定到这个 jdk 目录就算安装完成了：

```
vim /etc/profile
export JAVA_HOME = /home/hadoop/software/jdk1.8.0_121
source /etc/profile
```

8. Hadoop 安装

Hadoop 安装就是将一个 tar 包放上去并解压缩后再进行各个文件的配置。

```
#上传 hadoop-2.6.0-cdh5.tar.gz 到/home/hadoop/software/
tar xvzf hadoop-2.6.0-cdh5.tar.gz
mv hadoop-2.6.0-cdh5 hadoop2
cd /home/hadoop/software/hadoop2/etc/hadoop

vi hadoop-env.sh
#修改 JAVA_HOME 值
export JAVA_HOME = /home/hadoop/software/jdk1.8.0_121
vi yarn-env.sh
#修改 JAVA_HOME 值
export JAVA_HOME = /home/hadoop/software/jdk1.8.0_121
```

修改 Hadoop 的主从节点文件，slaves 是从节点，masters 是主节点。需要说明的是一个主节点也可以同时是从节点，也就是说这个节点可以同时是 NameNode 节点和 DataNode 节点。

```
vim slaves
```

添加这 5 台机器的节点：

```
data1
data2
data3
data4
data5
vim masters
```

添加两个 NameNode 节点：

```
data1
data2
```

下面来修改 Hadoop 的配置文件：

1）编辑 core-site.xml 文件

core-site.xml 文件用于定义系统级别的参数，如 HDFS URL、Hadoop 的临时目录等。这个文件主要是修改 fs.defaultFS 节点，改成 hdfs://ai，ai 是双 NameNode HA 的虚拟域名，hadoop.tmp.dir 节点也非常重要，如果不配置，Hadoop 重启后可能会有问题。

然后就是配置 ZooKeeper 的地址 ha.zookeeper.quorum。

```
<configuration>
<property>
<name>fs.defaultFS</name>
<value>hdfs://ai</value>
</property>
<property>
<name>ha.zookeeper.quorum</name>
<value>data1:2181,data2:2181,data3:2181,data4:2181,data5:2181</value>
</property>
<property>
<name>dfs.cluster.administrators</name>
<value>hadoop</value>
</property>
<property>
<name>io.file.buffer.size</name>
<value>131072</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/home/hadoop/software/hadoop/tmp</value>
<description>Abase for other temporary directories.</description>
</property>
<property>
<name>hadoop.proxyuser.hduser.hosts</name>
<value>*</value>
```

```
</property>
<property>
<name>hadoop.proxyuser.hduser.groups</name>
<value>*</value>
</property>
</configuration>
```

2）编辑 hdfs-site.xml 文件

hdfs-site.xml 文件用来配置名称节点和数据节点的存放位置、文件副本的个数和文件的读取权限等。

dfs.nameservices 设置双 NameNode HA 的虚拟域名。

dfs.ha.namenodes.ai 指定两个节点名称。

dfs.namenode.rpc-address.ai.nn1 指定 HDFS 访问节点 1。

dfs.namenode.rpc-address.ai.nn2 指定 HDFS 访问节点 2。

dfs.namenode.http-address.ai.nn1 指定 HDFS 的 Web 访问节点 1。

dfs.namenode.http-address.ai.nn2 指定 HDFS 的 Web 访问节点 2。

dfs.namenode.name.dir 定义 DFS 的名称节点在本地文件系统的位置。

dfs.datanode.data.dir 定义 DFS 数据节点存储数据块时存储在本地文件系统的位置。

dfs.replication 默认的块复制数量。

dfs.Webhdfs.enabled 设置是否通过 HTTP 协议读取 HDFS 文件，如果选是，则集群安全性较差。

```
vim hdfs-site.xml
<configuration>
<property>
<name>dfs.nameservices</name>
<value>ai</value>
</property>
<property>
<name>dfs.ha.namenodes.ai</name>
<value>nn1,nn2</value>
</property>
<property>
<name>dfs.namenode.rpc-address.ai.nn1</name>
<value>data1:9000</value>
</property>
<property>
<name>dfs.namenode.rpc-address.ai.nn2</name>
<value>data2:9000</value>
</property>
<property>
<name>dfs.namenode.http-address.ai.nn1</name>
<value>data1:50070</value>
```

```
</property>
<property>
<name>dfs.namenode.http-address.ai.nn2</name>
<value>data2:50070</value>
</property>
<property>
<name>dfs.namenode.shared.edits.dir</name>
<value>qjournal://data1:8485;data2:8485;data3:8485;data4:8485;data5:8485/aicluster
</value>
</property>
<property>
<name>dfs.client.failover.proxy.provider.ai</name>
<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
<property>
<name>dfs.ha.fencing.methods</name>
<value>sshfence</value>
</property>
<property>
<name>dfs.ha.fencing.ssh.private-key-files</name>
<value>/home/hadoop/.ssh/id_rsa</value>
</property>
<property>
<name>dfs.journalnode.edits.dir</name>
<value>/home/hadoop/software/hadoop/journal/data</value>
</property>
<property>
<name>dfs.ha.automatic-failover.enabled</name>
<value>true</value>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:/home/hadoop/software/hadoop/dfs/name</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/home/hadoop/software/hadoop/dfs/data</value>
</property>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
<name>dfs.Webhdfs.enabled</name>
<value>true</value>
</property>
<property>
```

```
< name > dfs. permissions </name >
< value > true </value >
</property >
< property >
< name > dfs. client. block. write. replace - datanode - on - failure. enable </name >
< value > true </value >
</property >
< property >
< name > dfs. client. block. write. replace - datanode - on - failure. policy </name >
< value > NEVER </value >
</property >
< property >
< name > dfs. datanode. max. xcievers </name >
< value > 4096 </value >
</property >
< property >
< name > dfs. datanode. balance. bandwidthPerSec </name >
< value > 104857600 </value >
</property >
< property >
< name > dfs. qjournal. write - txns. timeout. ms </name >
< value > 120000 </value >
</property >
</configuration >
```

3）编辑 mapred-site. xml 文件

主要修改 mapreduce. jobhistory. address 和 mapreduce. jobhistory. webapp. address 两个节点，配置历史服务器地址，通过历史服务器查看已经运行完的 MapReduce 作业记录，例如用了多少个 Map、用了多少个 Reduce、作业提交时间、作业启动时间和作业完成时间等信息。默认情况下，Hadoop 历史服务器是没有启动的，我们可以通过下面的命令来启动 Hadoop 历史服务器：

```
$ sbin/mr - jobhistory - daemon. sh start historyserver
```

这样就可以在相应机器的 19888 端口上打开历史服务器的 Web UI 界面，查看已经运行完成的作业情况。历史服务器可以单独在一台机器上启动，参数配置如下：

```
vim mapred - site. xml
    < configuration >
    < property >
        < name > mapreduce. framework. name </name >
        < value > yarn </value >
    </property >
    < property >
        < name > mapreduce. jobhistory. address </name >
        < value > data1:10020 </value >
```

```
</property>
<property>
    <name>mapred.child.env</name>
    <value>LD_LIBRARY_PATH=/usr/lib64</value>
</property>
<property>
    <name>mapreduce.jobhistory.Webapp.address</name>
    <value>data1:19888</value>
</property>
<property>
    <name>mapred.child.Java.opts</name>
    <value>-Xmx3072m</value>
</property>
<property>
    <name>mapreduce.task.io.sort.mb</name>
    <value>1000</value>
</property>
<property>
    <name>mapreduce.jobtracker.expire.trackers.interval</name>
    <value>1600000</value>
</property>
<property>
    <name>mapreduce.tasktracker.healthchecker.script.timeout</name>
    <value>1500000</value>
</property>
<property>
    <name>mapreduce.task.timeout</name>
    <value>88800000</value>
</property>
<property>
    <name>mapreduce.map.memory.mb</name>
    <value>8192</value>
</property>
<property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>8192</value>
</property>
<property>
<name>mapreduce.reduce.Java.opts</name>
<value>-Xmx6144m</value>
</property>
</configuration>
```

4）编辑 yarn-site.xml 文件

主要对 Yarn 资源调度的配置，核心配置参数如下：

```
yarn.resourcemanager.address
```

参数解释：ResourceManager 对客户端暴露地址。客户端通过该地址向 RM 提交应用程序和杀死应用程序等。

默认值：${yarn.resourcemanager.hostname}:8032
yarn.resourcemanager.scheduler.address

参数解释：ResourceManager 对 ApplicationMaster 暴露访问地址。ApplicationMaster 通过该地址向 RM 申请资源、释放资源等。

默认值：${yarn.resourcemanager.hostname}:8030
yarn.resourcemanager.resource-tracker.address

参数解释：ResourceManager 对 NodeManager 暴露地址。NodeManager 通过该地址向 RM 汇报心跳和领取任务等。

默认值：${yarn.resourcemanager.hostname}:8031
yarn.resourcemanager.admin.address

参数解释：ResourceManager 对管理员暴露访问地址。管理员通过该地址向 RM 发送管理命令等。

默认值：${yarn.resourcemanager.hostname}:8033
yarn.resourcemanager.Webapp.address

参数解释：ResourceManager 对外暴露 Web UI 地址。用户可通过该地址在浏览器中查看集群各类信息。

默认值：${yarn.resourcemanager.hostname}:8088
yarn.resourcemanager.scheduler.class

参数解释：启用的资源调度器主类。目前可用的有 FIFO、Capacity Scheduler 和 Fair Scheduler。

默认值：
org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler
yarn.resourcemanager.resource-tracker.client.thread-count

参数解释：处理来自 NodeManager 的 RPC 请求的 Handler 数目。

默认值：50
yarn.resourcemanager.scheduler.client.thread-count

参数解释：处理来自 ApplicationMaster 的 RPC 请求的 Handler 数目。

默认值：50
yarn.scheduler.minimum-allocation-mb/ yarn.scheduler.maximum-allocation-mb

参数解释：单个可申请的最小/最大内存资源量。例如设置为 1024 和 3072，则运行 MapReduce 作业时，每个 Task 最少可申请 1024MB 内存，最多可申请 3072MB 内存。

默认值:1024/8192
yarn.scheduler.minimum-allocation-vcores/yarn.scheduler.maximum-allocation-vcores

参数解释:单个可申请的最小/最大虚拟 CPU 个数。例如设置为 1 和 4,则运行 MapReduce 作业时,每个 Task 最少可申请 1 个虚拟 CPU,最多可申请 4 个虚拟 CPU。

默认值:1/32
yarn.resourcemanager.nodes.include-path/yarn.resourcemanager.nodes.exclude-path

参数解释:NodeManager 黑白名单。如果发现若干个 NodeManager 存在问题,例如故障率很高,任务运行失败率高,则可以将之加入黑名单中。注意,这两个配置参数可以动态生效。(调用一个 refresh 命令即可)

默认值:""
yarn.resourcemanager.nodemanagers.heartbeat-interval-ms

参数解释:NodeManager 心跳间隔。

默认值:1000(单位为毫秒)

一般需要修改的地方在下面的配置中加粗了。这个配置文件是 Yarn 资源调度器最核心的配置,下面的代码是一个实例配置。有一个需要注意的配置技巧,分配的内存和 CPU 一定要配套,需要根据你的服务器情况,计算最小分配内存来分配 CPU 等。如果这个计算不准确,可能会造成 Hadoop 进行任务资源分配的时候 CPU 资源用尽了,但内存还剩很多,但对于 Hadoop 来讲,只要 CPU 或内存有一个占满,后面的任务就不能再分配了,所以设置不好会造成 CPU 和内存资源的浪费。

另外一个需要注意的地方是将 yarn.nodemanager.webapp.address 节点复制到每台 Hadoop 服务器上后需记得把节点值的 IP 地址改成本机。如果这个地方忘了改,就可能会出现 NodeManager 启动不了的问题。

```
vim yarn-site.xml
<configuration>
<property>
<name>yarn.nodemanager.Webapp.address</name>
<value>172.172.0.11:8042</value>
</property>
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>data1:8031</value>
</property>
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>data1:8030</value>
</property>
<property>
<name>yarn.resourcemanager.scheduler.class</name>
```

```
< value > org. apache. hadoop. yarn. server. resourcemanager. scheduler. capacity. CapacityScheduler
</value >
</property >
< property >
< name > yarn. resourcemanager. address </name >
< value > data1 :8032 </value >
</property >
< property >
< name > yarn. nodemanager. local - dirs </name >
< value > $ {hadoop. tmp. dir}/nodemanager/local </value >
</property >
< property >
< name > yarn. nodemanager. address </name >
< value > 0. 0. 0. 0 :8034 </value >
</property >
< property >
< name > yarn. nodemanager. remote - app - log - dir </name >
< value > $ {hadoop. tmp. dir}/nodemanager/remote </value >
</property >
< property >
< name > yarn. nodemanager. log - dirs </name >
< value > $ {hadoop. tmp. dir}/nodemanager/logs </value >
</property >
< property >
< name > yarn. nodemanager. aux - services </name >
< value > mapreduce_shuffle </value >
</property >
< property >
< name > yarn. nodemanager. aux - services. mapreduce. shuffle. class </name >
< value > org. apache. hadoop. mapred. ShuffleHandler </value >
</property >
< property >
< name > mapred. job. queue. name </name >
< value > $ {user. name}</value >
</property >
< property >
< name > yarn. nodemanager. resource. memory - mb </name >
< value > 116888 </value >
</property >
< property >
< name > yarn. scheduler. minimum - allocation - mb </name >
< value > 5120 </value >
</property >
< property >
< name > yarn. scheduler. maximum - allocation - mb </name >
< value > 36688 </value >
</property >
```

```
<property>
<name>yarn.scheduler.maximum-allocation-vcores</name>
<value>8</value>
</property>
<property>
<name>yarn.nodemanager.resource.cpu-vcores</name>
<value>50</value>
</property>
<property>
<name>yarn.scheduler.minimum-allocation-vcores</name>
<value>2</value>
</property>
<property>
<name>yarn.nm.liveness-monitor.expiry-interval-ms</name>
<value>700000</value>
</property>
<property>
<name>yarn.nodemanager.health-checker.interval-ms</name>
<value>800000</value>
</property>
<property>
<name>yarn.nm.liveness-monitor.expiry-interval-ms</name>
<value>900000</value>
</property>
<property>
<name>yarn.resourcemanager.container.liveness-monitor.interval-ms</name>
<value>666000</value>
</property>
<property>
<name>yarn.nodemanager.localizer.cache.cleanup.interval-ms</name>
<value>688000</value>
</property>
</configuration>
```

5）编辑 capacity-scheduler.xml 文件

在前面讲的 yarn-site.xml 配置文件中，我们配置的调度器是容量调度器，就是这个节点指定的配置 yarn.resourcemanager.scheduler.class，容量调度器是 Hadoop 默认的调度器，另外还有公平调度器，下面将分别讲解，看看它们有什么区别。

（1）公平调度器

公平调度器的核心理念是随着时间的推移平均分配工作，这样每个作业都能平均地共享到资源。结果只需较少时间执行的作业能够较早访问 CPU，而那些需要较长时间执行的作业需要较长时间才能结束。这样的执行方式可以在 Hadoop 作业之间形成交互，而且可以让 Hadoop 集群对提交的多种类型作业做出更快的响应。公平调度器是由 Facebook 开发出来的。

Hadoop 的实现会创建一个作业组池，将作业放在其中供调度器选择。每个池会分配一组作业共享以平衡池中作业的资源（更多的共享意味着作业执行所需的资源更多）。默认情况下，所有池的共享资源相等，但可以进行配置，根据作业类型提供更多或更少的共享资源。如果需要的话，还可以限制同时活动的作业数，以尽量减少拥堵，让工作及时完成。

为了保证公平，每个用户被分配一个池。在这样的方式下，无论一个用户提交多少作业，他分配的集群资源都与其他用户一样多（与他提交的工作数无关）。无论分配到池的共享资源有多少，如果系统未加载，那么作业收到的共享资源不会被使用（在可用作业之间分配）。

调度器会追踪系统中每个作业的计算时间。调度器还会定期检查作业接收到的计算时间和在理想的调度器中应该收到的计算时间的差距，并会使用该结果来确定任务的亏空。调度器作业接着会保证亏空最多的任务最先执行。

在 mapred-site. xml 文件中配置公平共享。该文件会定义对公平调度器行为的管理。一个 xml 文件（即 mapred. fairscheduler. allocation. file 属性）定义了每个池的共享资源的分配。为了优化作业大小，我们可以设置 mapread. fairscheduler. sizebasedweight 将共享资源分配给作业作为其大小的函数。还有一个类似的属性可以通过调整作业的权重让更小的作业在 5 分钟之后运行得更快（mapred. fairscheduler. weightadjuster）。我们还可以用很多其他的属性来调优节点上的工作负载（例如某个 TaskTracker 能管理的 maps 和 reduces 数目）并确定是否执行抢占。

（2）容量调度器

容量调度器的原理与公平调度器有些相似，但也有一些区别。首先，容量调度器用于大型集群，它们有多个独立用户和目标应用程序。由于这个原因，容量调度器能提供更大的控制和能力，提供用户之间最小容量并保证在用户之间共享多余的容量。容量调度器是由 Yahoo！开发出来的。

在容量调度器中，创建的是队列而不是池，每个队列的 map 和 reduce 插槽数都可以配置。每个队列都会分配一个有保证的容量（集群的总容量是每个队列容量之和）。

队列处于监控之下，如果某个队列未使用分配的容量，那么这些多余的容量会被临时分配到其他队列中。由于队列可以表示一个人或大型组织，那么所有的可用容量都可以由其他用户重新使用。

与公平调度器的另一个区别是可以调整队列中作业的优先级。一般来说，具有高优先级的作业访问资源比低优先级作业更快。Hadoop 路线图包含了对抢占的支持（临时替换出低优先级作业，让高优先级作业先执行），但该功能尚未实现。

还有一个区别是对队列进行严格的访问控制（假设队列绑定到一个人或组织）。这些访问控制是按照每个队列进行定义的。对于将作业提交到队列的能力和查看修改队列中作业的能力都有严格限制。

容量调度器可在多个 Hadoop 配置文件中配置。队列在 hadoop-site. xml 中定义，在 capacity-scheduler. xml 中配置，在 mapred-queue-acls. xml 中配置 ACL。单个的队列属性

包括容量百分比(集群中所有的队列容量少于或等于100)、最大容量(队列多余容量使用的限制)及队列是否支持优先级。更重要的是可以在运行时调整队列优先级,从而可以在集群的使用过程中改变或避免中断的情况。

我们的实例用的是容量调度器,看以下配置参数:

mapred.capacity-scheduler.queue.<queue-name>.capacity:

设置容量调度器中各个queue的容量,这里指的是占用集群的slots的百分比,需要注意的是,所有queue的配置项加起来必须小于或等于100,否则会导致JobTracker启动失败。

mapred.capacity-scheduler.queue.<queue-name>.maximum-capacity:

设置容量调度器中各个queue最大可以占有的容量,默认为-1,表示最大可以占有集群100%的资源,这样和设置为100的效果是一样的。

mapred.capacity-scheduler.queue.<queue-name>.minimum-user-limit-percent:

当queue中多个用户出现slots竞争的时候,可以限制每个用户的slots资源的百分比。例如,当minimum-user-limit-percent设置为25%时,如果queue中有多余的4个用户同时提交job,那么容量调度器保证每个用户占有的slots不超过queue中slots数的25%,默认为100表示不对用户作限制。

mapred.capacity-scheduler.queue.<queue-name>.user-limit-factor:

设置queue中用户可占用queue容量的系数,默认为1,表示queue中每个用户最多只能占有queue的容量(即mapred.capacity-scheduler.queue.<queue-name>.capacity),因此需要注意的是,如果queue中只有一个用户提交job,且希望此用户在集群不繁忙的时候可扩展到mapred.capacity-scheduler.queue.<queue-name>.maximum-capacity指定的slots数,则必须相应地调大user-limit-factor系数。

mapred.capacity-scheduler.queue.<queue-name>.supports-priority:

设置容量调度器中各个queue是否支持job优先级,不用过多解释。

mapred.capacity-scheduler.maximum-system-jobs:

设置容量调度器中各个queue中全部可初始化后并发执行的job数,需要注意的是各个queue会按照自己占有集群slots资源的比例(即mapred.capacity-scheduler.queue.<queue-name>.capacity)决定每个queue最多同时并发执行的job数。例如,假设maximum-system-jobs为20个,而queue1占集群10%的资源,那么意味着queue1最多可同时并发运行2个job,如果碰巧是运行时间比较长的job,那么将直接导致其他新提交的job被Job Tracker阻塞而不能进行初始化。

mapred.capacity-scheduler.queue.<queue-name>.maximum-initialized-active-tasks:

设置 queue 中所有并发运行 job 包含的 task 数的上限值，如果超过此限制，则新提交到该 queue 中的 job 会被排队并缓存到磁盘上。

```
mapred.capacity-scheduler.queue.<queue-name>.maximum-initialized-active-tasks-per-user:
```

设置 queue 中每个特定用户并发运行 job 包含的 task 数的上限值，如果超过此限制，则该用户新提交到该 queue 中的 job 会被排队并缓存到磁盘上。

```
mapred.capacity-scheduler.queue.<queue-name>.init-accept-jobs-factor:
```

设置每个 queue 中可容纳接收的 job 总数(maximum-system-jobs×queue-capacity)的系数，举个例子，如果 maximum-system-jobs 为 20，queue-capacity 为 10%，init-accept-jobs-factor 为 10，则当 queue 中 job 总数达到 10×(20×10%)=20 时，新的 job 将被 JobTracker 拒绝提交。

下面的配置实例配置了 Hadoop 和 Spark 两个队列，Hadoop 队列分配了 92%的资源，参见 yarn.scheduler.capacity.root.hadoop.capacity 配置，Spark 队列分配了 8%的资源，参见 yarn.scheduler.capacity.root.spark.capacity 配置：

```
vim capacity-scheduler.xml
<configuration>
    <property>
        <name>yarn.scheduler.capacity.maximum-applications</name>
        <value>10000</value>
    </property>
    <property>
        <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
        <value>0.1</value>
    </property>
    <property>
        <name>yarn.scheduler.capacity.resource-calculator</name>
    <value>org.apache.hadoop.yarn.util.resource.DominantResourceCalculator</value>
    </property>
    <property>
        <name>yarn.scheduler.capacity.node-locality-delay</name>
        <value>-1</value>
    </property>
    <property>
        <name>yarn.scheduler.capacity.root.queues</name>
        <value>hadoop,spark</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.hadoop.capacity</name>
    <value>92</value>
    </property>
```

```
    <property>
    <name>yarn.scheduler.capacity.root.hadoop.user-limit-factor</name>
    <value>1</value>
    </property>
    <property><name>yarn.scheduler.capacity.root.hadoop.maximum-capacity</name>
    <value>-1</value>
    </property>
    <property><name>yarn.scheduler.capacity.root.hadoop.state</name>
    <value>RUNNING</value>
    </property>
    <property><name>yarn.scheduler.capacity.root.hadoop.acl_submit_applications</name>
    <value>hadoop</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.hadoop.acl_administer_queue</name>
    <value>hadoop hadoop</value>
    </property>
    <!--sparkquene-->
    <property>
    <name>yarn.scheduler.capacity.root.spark.capacity</name>
    <value>8</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.spark.user-limit-factor</name>
    <value>1</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.spark.maximum-capacity</name>
    <value>-1</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.spark.state</name>
    <value>RUNNING</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.spark.acl_submit_applications</name>
    <value>hadoop</value>
    </property>
    <property>
    <name>yarn.scheduler.capacity.root.spark.acl_administer_queue</name>
    <value>hadoop hadoop</value>
    </property>
    <!--end-->
</configuration>
```

以上把 Hadoop 的配置文件都配置好了，然后把这台服务器 Hadoop 的整个目录复制到其他机器上就可以了。记得有个地方需要修改，yarn-site.xml 里 yarn.nodemanager.

webapp.address 需将每台 Hadoop 服务器上的 IP 地址改成本机地址。如果这个地方忘了改，就可能出现 Node Manager 启动不了的问题。

```
scp -r /home/hadoop/software/hadoop2 hadoop@data2:/home/hadoop/software/
scp -r /home/hadoop/software/hadoop2 hadoop@data3:/home/hadoop/software/
scp -r /home/hadoop/software/hadoop2 hadoop@data4:/home/hadoop/software/
scp -r /home/hadoop/software/hadoop2 hadoop@data5:/home/hadoop/software/
```

另外还有个地方需要优化，默认情况下，如果 Hadoop 运行多个 reduce 可能会报错：

```
Failed on local exception: Java.io.IOException: Couldn't set up IO streams; Host Details : local
host
```

解决办法：集群所有节点增加如下配置：

```
#在文件中增加
    sudo vi /etc/security/limits.conf
    hadoop soft nproc 100000
    hadoop hard nproc 100000
```

重启整个集群的每个节点，重启 Hadoop 集群即可。

到现在为止环境安装一切准备就绪，下面我们就开始对 Hadoop 的 HDFS 分布式文件系统格式化，就像我们买了新计算机后磁盘需要格式化才能用一样。由于我们的实例采用 NameNode HA 双节点模式，它是依靠 ZooKeeper 来实现的，所以我们现在需要先安装好 ZooKeeper 才行。在每台服务器上启动 ZooKeeper 服务：

```
/home/hadoop/software/zookeeper-3.4.6/bin/zkServer.sh restart
```

在 NameNode1 上的 data1 服务器初始化 ZooKeeper：

```
hdfs zkfc -formatZK
```

分别在 5 台 Hadoop 集群上启动 journalnode 服务，执行命令：

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

在 NameNode1 上的 data1 服务器格式化 HDFS：

```
hdfs namenode -format
```

然后启动这台机器上的 NameNode 节点服务：

```
hadoop-daemon.sh start namenode
```

在第二个 NameNode 上执行 data2：

```
hdfs namenode -bootstrapStandby
hadoop-daemon.sh start namenode
```

最后我们启动 Hadoop 集群：

```
start-all.sh
```

启动集群过程如下：

```
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [datanode1 datanode2]
datanode2: starting namenode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-namenode-datanode2.out
datanode1: starting namenode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-namenode-datanode1.out
datanode2: Java HotSpot(TM) 64-Bit Server VM warning: UseCMSCompactAtFullCollection is deprecated and will likely be removed in a future release.
datanode2: Java HotSpot(TM) 64-Bit Server VM warning: CMSFullGCsBeforeCompaction is deprecated and will likely be removed in a future release.
datanode1: Java HotSpot(TM) 64-Bit Server VM warning: UseCMSCompactAtFullCollection is deprecated and will likely be removed in a future release.
datanode1: Java HotSpot(TM) 64-Bit Server VM warning: CMSFullGCsBeforeCompaction is deprecated and will likely be removed in a future release.
172.172.0.12: starting datanode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-datanode-datanode2.out
172.172.0.11: starting datanode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-datanode-datanode1.out
172.172.0.14: starting datanode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-datanode-datanode4.out
172.172.0.13: starting datanode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-datanode-datanode3.out
172.172.0.15: starting datanode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-datanode-datanode5.out
Starting journal nodes [172.172.0.11 172.172.0.12 172.172.0.13 172.172.0.14 172.172.0.15]
172.172.0.14: starting journalnode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-journalnode-datanode4.out
172.172.0.11: starting journalnode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-journalnode-datanode1.out
172.172.0.13: starting journalnode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-journalnode-datanode3.out
172.172.0.15: starting journalnode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-journalnode-datanode5.out
172.172.0.12: starting journalnode, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-journalnode-datanode2.out
Starting ZK Failover Controllers on NN hosts [datanode1 datanode2]
datanode1: starting zkfc, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-zkfc-datanode1.out
datanode2: starting zkfc, logging to /home/hadoop/software/hadoop2/logs/hadoop-hadoop-zkfc-datanode2.out
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-resourcemanager-datanode1.out
```

```
172.172.0.15: starting nodemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-nodemanager-datanode5.out
172.172.0.14: starting nodemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-nodemanager-datanode4.out
172.172.0.12: starting nodemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-nodemanager-datanode2.out
172.172.0.13: starting nodemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-nodemanager-datanode3.out
172.172.0.11: starting nodemanager, logging to /home/hadoop/software/hadoop2/logs/yarn-hadoop-nodemanager-datanode1.out
```

如果是停止集群则用这个命令：stop-all.sh

停止集群过程如下：

```
This script is Deprecated. Instead use stop-dfs.sh and stop-yarn.sh
Stopping namenodes on [datanode1 datanode2]
datanode1: stopping namenode
datanode2: stopping namenode
172.172.0.12: stopping datanode
172.172.0.11: stopping datanode
172.172.0.15: stopping datanode
172.172.0.13: stopping datanode
172.172.0.14: stopping datanode
Stopping journal nodes [172.172.0.11 172.172.0.12 172.172.0.13 172.172.0.14 172.172.0.15]
172.172.0.11: stopping journalnode
172.172.0.13: stopping journalnode
172.172.0.12: stopping journalnode
172.172.0.15: stopping journalnode
172.172.0.14: stopping journalnode
Stopping ZK Failover Controllers on NN hosts [datanode1 datanode2]
datanode2: stopping zkfc
datanode1: stopping zkfc
stopping yarn daemons
stopping resourcemanager
172.172.0.13: stopping nodemanager
172.172.0.12: stopping nodemanager
172.172.0.15: stopping nodemanager
172.172.0.14: stopping nodemanager
172.172.0.11: stopping nodemanager
no proxyserver to stop
```

启动成功后在每个节点上会看到对应 Hadoop 进程，NameNode1 主节点上看到的进程如下：

```
5504 ResourceManager
4912 NameNode
5235 JournalNode
5028 DataNode
```

```
5415 DFSZKFailoverController
90 QuorumPeerMain
5628 NodeManager
```

ResourceManager 就是 Yarn 资源调度的进程。NameNode 是 HDFS 的 NameNode 主节点。JournalNode 是 JournalNode 节点。DataNode 是 HDFS 的 DataNode 从节点和数据节点。DFSZKFailoverController 是 Hadoop 中 HDFS NameNode HA 实现的中心组件，它负责整体的故障转移控制等。它是一个守护进程，通过 main()方法启动，继承自 ZKFailoverController。QuorumPeerMain 是 ZooKeeper 的进程。NodeManager 是 Yarn 在每台服务器上的节点管理器，是运行在单个节点上的代理，它管理 Hadoop 集群中单个计算节点，功能包括与 ResourceManager 保持通信、管理 Container 的生命周期、监控每个 Container 的资源使用(内存、CPU 等)情况、追踪节点健康状况、管理日志和不同应用程序用到的附属服务等。

NameNode2 主节点 2 上的进程如下：

```
27232 NameNode
165 QuorumPeerMain
27526 DFSZKFailoverController
27408 JournalNode
27313 DataNode
27638 NodeManager
```

这样便会少很多进程，因为做主节点的 HA 也会有一个 NameNode 进程，如果没有，说明这个节点的 NameNode 挂了，我们需要重启它，并需要查看挂掉的原因。

下面是其中一台 DataNode 上的进程，却没有 NameNode 进程了：

```
114 QuorumPeerMain
17415 JournalNode
17320 DataNode
17517 NodeManager
```

我们除了能看到集群每个节点的进程，还能根据进程判断哪个集群节点有问题，但这样不是很方便，这需要我们每台服务器逐个来看。Hadoop 提供了 Web 界面，可以非常方便地查看集群的状况。一个是 Yarn 的 Web 界面，在 ResourceManager 进程所在的那台机器上访问，也就是 Yarn 的主进程，访问地址是 http://namenodeip：8088/，端口是 8088，当然这个是默认端，可以通过配置文件来改，不过一般不与其他端口冲突的话是不需要修改的；另一个是两个 NameNode 的 Web 界面，端口是 50070，能非常方便查看 HDFS 集群状态，包括总空间、使用空间和剩余空间，这样每台服务器节点情况便一目了然，访问地址是：http://namenodeip：50070/。我们来看一下这两个界面，Yarn 的 Web 界面如图 3.1 所示。NameNode 的 Web 界面如图 3.2 所示。

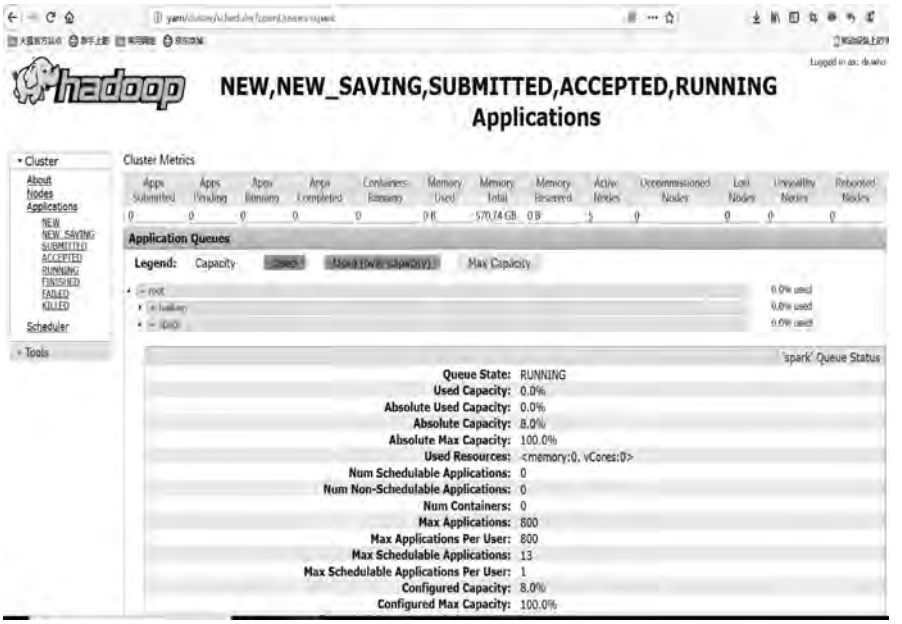

图 3.1 Yarn 的 Web 界面截图

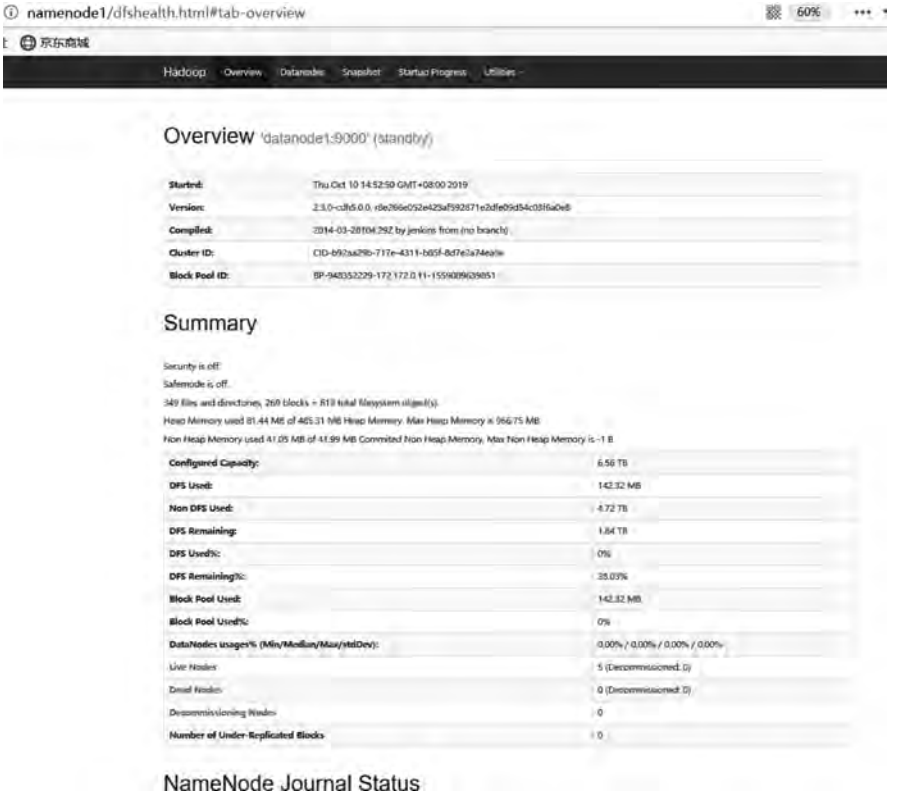

图 3.2 NameNode 的 Web 界面截图

3.1.3 Hadoop 常用操作命令

Hadoop 操作命令主要分 Hadoop 集群启动维护命令、HDFS 文件操作命令、Yarn 资源调度相关命令，我们来分别讲解一下。

1. Hadoop 集群启动维护

```
#整体启动 Hadoop 集群
start-all.sh
#整体停止 Hadoop 集群
stop-all.sh
#单独启动 NameNode 服务
hadoop-daemon.sh start namenode
#单独启动 DataNode 服务
hadoop-daemon.sh start datanode
#在某台机器上单独启动 NodeManager 服务
yarn-daemon.sh start nodemanager
#单独启动 HistoryServer
mr-jobhistory-daemon.sh start historyserver
```

2. HDFS 文件操作命令

操作使用 hadoop dfs 或者 hadoop fs 命令都可以，简化操作时间，建议使用 hadoop fs 命令。

1）列出 HDFS 下的文件

```
hadoop fs -ls /
hadoop fs -ls /ods/kc/dim/ods_kc_dim_product_tab/
```

2）查看文件的尾部的记录

```
hadoop fs -tail /ods/kc/dim/ods_kc_dim_product_tab/product.txt
```

3）上传本地文件到 Hadoop 的 HDFS 上

```
hadoop fs -put product.txt /ods/kc/dim/ods_kc_dim_product_tab/
```

4）把 Hadoop 上的文件下载到本地系统中

```
hadoop fs -get /ods/kc/dim/ods_kc_dim_product_tab/product.txt product.txt
```

5）删除文件和删除目录

```
hadoop fs -rm /ods/kc/dim/ods_kc_dim_product_tab/product.txt
hadoop fs -rmr /ods/kc/dim/ods_kc_dim_product_tab/
```

6）查看文件

```
#谨慎使用，尤其当文件内容太长时
hadoop fs -cat /ods/kc/dim/ods_kc_dim_product_tab/product.txt
```

7）建立目录

```
hadoop fs -mkdir /ods/kc/dim/ods_kc_dim_product_tab/(目录/目录名)
#只能一级一级地建目录,建完一级才能建下一级。如果-mkdir -p价格,-p参数会自动把不存
#在的文件夹都创建上
```

8）本集群内复制文件

```
hadoop fs -cp 源路径
```

9）跨集群对拷,适合做集群数据迁移使用

```
hadoop distcp hdfs://master1/ods/ hdfs://master2/ods/
```

10）通过 Hadoop 命令把多个文件的内容合并起来

```
#hadoop fs -getmerge 位于HDFS中的原文件(里面有多个文件)合并后的文件名(本地)
```

例如：

```
hadoop fs -getmerge /ods/kc/dim/ods_kc_dim_product_tab/* all.txt
```

3. Yarn 资源调度相关命令

1）application

使用语法：

```
yarn application [options]          #打印报告,申请和杀死任务
-appStates <States>                 #与-list一起使用,可根据输入的逗号分隔应用程序状态列
                                    #表来过滤应用程序。有效的应用程序状态可以是以下之一:ALL,
                                    #NEW,NEW_SAVING,SUBMITTED,ACCEPTED,#RUNNING,FINISHED,
                                    #FAILED,KILLED
-appTypes <Types>                   #与-list一起使用,可以根据输入的逗号分隔应用程序类型列
                                    #表来过滤应用程序
-list                               #列出RM中的应用程序。支持使用-appTypes来根据应用程序
                                    #类型过滤应用程序,并支持使用-appStates来根据应用程序
                                    #状态过滤应用程序
-kill <ApplicationId>               #终止应用程序
-status <ApplicationId>             #打印应用程序的状态
```

2）applicationattempt

使用语法：

```
yarn applicationattempt [options]       #打印应用程序尝试的报告
-help                                   #帮助
-list <ApplicationId>                   #获取到应用程序尝试的列表,其返回值Application-
                                        #Attempt-Id等于<Application Attempt Id>
-status <Application Attempt Id>        #打印应用程序尝试的状态
```

3）classpath

使用语法：

```
yarn classpath                          #打印需要得到 Hadoop 的 jar 和所需要的 lib 包路径
```

4）container

使用语法：

```
yarn container [options]                #打印 Container(s)的报告
-help                                   #帮助
-list <Application Attempt Id>          #应用程序尝试的 Containers 列表
-status <ContainerId>                   #打印 Container 的状态
```

5）jar

使用语法：

```
yarn jar <jar> [mainClass] args…        #运行 jar 文件,用户可以将写好的 Yarn 代码打包成
                                        #jar 文件,用这个命令去运行它
```

6）logs

使用语法：

```
yarn logs -applicationId <application ID> [options]
                                        #转存 Container 的日志
-applicationId <application ID>         #指定应用程序 ID,应用程序的 ID 可以在 yarn.
                                        #resourcemanager.webapp.address 配置的路径
                                        #查看(即:ID)
-appOwner <AppOwner>                    #应用的所有者(如果没有指定就是当前用户)应用程序
                                        #的 ID 可以在 yarn.resourcemanager.webapp.address
                                        #配置的路径查看(即:User)
-containerId <ContainerId>              #Container Id
-help                                   #帮助
-nodeAddress <NodeAddress>              #节点地址的格式:nodename:port (端口是配置文件中:
                                        #yarn.nodemanager.Webapp.address 参数指定)
```

7）node

使用语法：

```
yarn node [options]                     #打印节点报告
-all                                    #所有的节点,不管是什么状态的
-list                                   #列出所有 RUNNING 状态的节点。支持 -states 选
                                        #项过滤指定的状态,节点的状态包含 NEW,RUNNING,
                                        #UNHEALTHY,DECOMMISSIONED,LOST,REBOOTED。
                                        #支持 -all 显示所有的节点
-states <States>                        #和 -list 配合使用,用逗号分隔节点状态,只显示这
                                        #些状态的节点信息
-status <NodeId>                        #打印指定节点的状态
```

8）queue

使用语法：

```
yarn queue [options]                    #打印队列信息
-help                                   #帮助
-status                                 #<QueueName>打印队列的状态
```

9）daemonlog

使用语法：

```
yarn daemonlog -getlevel <host:httpport><classname>
yarn daemonlog -setlevel <host:httpport><classname><level>
-getlevel <host:httpport><classname>
                                        #打印运行在<host:port>的守护进程的日志级别。
                                        #这个命令内部会连接 http://<host:port>/
                                        #logLevel?log=<name>
-setlevel <host:httpport><classname><level>
                                        #设置运行在<host:port>的守护进程的日志级别。
                                        #这个命令内部会连接 http://<host:port>/
                                        #logLevel?log=<name>
```

10）nodemanager

使用语法：

```
yarn nodemanager                        #启动 NodeManager
```

11）proxyserver

使用语法：

```
yarn proxyserver                        #启动 Web proxy server
```

12）resourcemanager

使用语法：

```
yarn resourcemanager [-format -state -store]
                                        #启动 ResourceManager
-format -state -store                   #RMStateStore 的格式。如果过去的应用程序不再需要，
                                        #则清理 RMStateStore，RMStateStore 仅仅在 ResourceManager
                                        #没有运行的时候才运行 RMStateStore
```

13）rmadmin

使用语法：

```
#运行 Resourcemanager 管理客户端
yarn rmadmin [-refreshQueues]
             [-refreshNodes]
             [-refreshUserToGroupsMapping]
             [-refreshSuperUserGroupsConfiguration]
```

```
            [ - refreshAdminAcls]
            [ - refreshServiceAcl]
            [ - getGroups [username]]
            [ - transitionToActive [ -- forceactive] [ -- forcemanual] < serviceId >]
            [ - transitionToStandby [ -- forcemanual] < serviceId >]
            [ - failover [ -- forcefence] [ -- forceactive] < serviceId1 >< serviceId2 >]
            [ - getServiceState < serviceId >]
            [ - checkHealth < serviceId >]
            [ - help [cmd]]
- refreshQueues    # 重载队列的 ACL、状态和调度器特定的属性,ResourceManager 将重载
                   # mapred - queues 配置文件
- refreshNodes     # 动态刷新 dfs. hosts 和 dfs. hosts. exclude 配置,无须重启 NameNode dfs. hosts:
                   # 列出了允许连入 NameNode 的 DataNode 清单(IP 或者机器名)dfs. hosts. exclude:
                   # 列出了禁止连入 NameNode 的 DataNode 清单(IP 或者机器名)重新读取 hosts 和
                   # exclude 文件,更新允许连到 NameNode 或那些需要退出或入编的 DataNode 的集合
- refreshUserToGroupsMappings              # 刷新用户到组的映射
- refreshSuperUserGroupsConfiguration      # 刷新用户组的配置
- refreshAdminAcls                         # 刷新 ResourceManager 的 ACL 管理
- refreshServiceAcl                        # ResourceManager 重载服务级别的授权文件
- getGroups [username]                     # 获取指定用户所属的组
- transitionToActive [ - forceactive] [ - forcemanual] < serviceId >
                                           # 尝试将目标服务转为 Active 状态。如果使用了
                                           # - forceactive 选项,不需要核对非 Active 节点。
                                           # 如果采用了自动故障转移,这个命令不能使用。
                                           # 虽然你可以重写 - forcemanual 选项,但需要谨
                                           # 慎操作
- transitionToStandby [ - forcemanual] < serviceId >
                                           # 将服务转为 Standby 状态。如果采用了自动
                                           # 故障转移,这个命令不能使用。虽然你可以重
                                           # 写 - forcemanual 选项,但需要谨慎操作
- failover [ - forceactive] < serviceId1 >< serviceId2 >
                                           # 启动从 serviceId1 到 serviceId2 的故障转移。
                                           # 如果使用了 - forceactive 选项,即使服务没有
                                           # 准备,也会尝试故障转移到目标服务。如果采用
                                           # 了自动故障转移,这个命令不能使用
- getServiceState < serviceId >            # 返回服务的状态(注:ResourceManager 不是 HA 的
                                           # 时候,是不能运行该命令的)
- checkHealth < serviceId >                # 请求服务器执行健康检查,如果检查失败,RMAdmin
                                           # 将用一个非零标示退出(注:Resource Manager 不是
                                           # HA 的时候,是不能运行该命令的)
- help [cmd]                               # 显示指定命令的帮助,如果没有指定,则显示命令的
                                           # 帮助
```

14) scmadmin

使用语法:

```
yarn scmadmin [options]                    # 运行共享缓存管理客户端
```

```
-help                                    #查看帮助
-runCleanerTask                          #运行清理任务
```

15) sharedcachemanager

使用语法：

```
yarn sharedcachemanager                  #启动共享缓存管理器
```

16) timelineserver

使用语法：

```
yarn timelineserver                      #启动 timelineserver
```

到目前为止 Hadoop 平台搭建好了，里面本身是没有数据的，所以下一步的工作就是建设数据仓库，而数据仓库是以 Hive 为主流的，所以下面我们来讲解 Hive。

3.2 Hive 数据仓库实战

Hive 作为大数据平台 Hadoop 之上的主流应用，一般公司都是用它作为公司的数据仓库，分布式机器学习的训练数据和数据处理也经常用它来处理，下面介绍它的常用功能。

3.2.1 Hive 原理和功能介绍

Hive 是建立在 Hadoop 之上的数据仓库基础构架。它提供了一系列工具，可以用来进行数据提取、转化和加载(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 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，Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。

Hive 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 SQL 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。同时，这个 Hive 也允许熟悉 MapReduce 的开发者开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂分析工作，例如 UDF 函数。

简单来讲，Hive 从表面看来，你可以把它当成类似 MySQL 差不多的东西，就是一个数据库而已。按本质来讲，它也并不是数据库。其实它就是一个客户端工具而已，数据是在 Hadoop 的 HDFS 分布式文件系统上存着，只是它提供一种方便的方式让你很轻松从 HDFS 查询数据和更新数据。Hive 既然是一个客户端工具，就不需要启动什么服务，只需解压就能用。操作方式通过写类似 MySQL 的 SQL 语句对 HDFS 操作，提交 SQL 后，Hive 会把 SQL 解析成 MapReduce 程序去执行，分布式多台机器并行地执行。当数据存入 HDFS 后，大部分统计工作可以通过写 Hive SQL 的方式来完成，大大提高了工作效率。

3.2.2 Hive 安装部署

Hive 的安装部署非常简单，因为它本身是 Hadoop 的一个客户端，而不是一个集群服务，所以把安装包解压后修改配置就可以用。在哪台机器上登录 Hive 客户端就在哪台机器上部署，不用在每台服务器上都部署。安装过程如下：

```
#上传 hive.tar.gz 到/home/hadoop/software/hadoop2
¥cd /home/hadoop/software/hadoop2
tar xvzf hive.tar.gz
cd hive/conf
mv hive-env.sh.template hive-env.sh
mv hive-default.xml.template hive-site.xml
vim ../bin/hive-config.sh
#增加
export JAVA_HOME=/home/hadoop/software/jdk1.8.0_121
export HIVE_HOME=/home/hadoop/software/hadoop2/hive
export HADOOP_HOME=/home/hadoop/software/hadoop2
```

修改以下配置字节点，主要是配置 Hive 的元数据存储用 MySQL，因为默认的是 Derby 文件数据库，实际公司用的时候都是改成用 MySQL 数据库。

```
vim hive-site.xml
<property>
<name>Javax.jdo.option.ConnectionURL</name>
<value>jdbc:mysql://192.168.1.166:3306/chongdianleme_hive?createDatabaseIfNotExist=true</value>
</property>
<property>
<name>Javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>Javax.jdo.option.ConnectionUserName</name><value>root</value>
</property>
    <property>
            <name>Javax.jdo.option.ConnectionPassword</name>
            <value>123456</value>
        </property>
    <property>
    <name>hive.metastore.schema.verification</name>
    <value>false</value>
    <description>
    </description>
    </property>
```

因为 Hive 默认配置并没有把 MySQL 的驱动 jar 包集成进去，所以需要我们手动上传

mysql-connector-Java-*.*-bin.jar 到/home/hadoop/software/hadoop2/hive/lib 目录下，Hive 客户端启动的时候会自动加载这个目录下的所有 jar 包。

部署就这么简单，我们在 Linux 客户端输入 Hive 并按回车键就可以进到控制台命令窗口，后面就可以建表、查询数据和更新数据等操作了。下面我们看一下 Hive 的常用 SQL 操作。

3.2.3 Hive SQL 操作

Hive 查询数据、更新数据前需要先建表，有了表之后我们可以往表里写入数据，之后才可以用 Hive 执行查询和更新等操作。

1. 建表操作

```
#建 Hive 表脚本
create EXTERNAL table IF NOT EXISTS ods_kc_fact_clicklog_tab(userid string,kcid string,time
string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t'
stored as textfile
location '/ods/kc/fact/ods_kc_fact_clicklog/';
#EXTERNAL 关键词的意思是创建外部表,目的是当你 drop table 的时候外部表数据不会被删除,
#只会删除表结构,表结构又叫作元数据.想恢复表结构只需要把这个表再创建一次就可以,
#表里面的数据还存在,所以为了保险并防止误操作,一般 Hive 数据仓库建外部表
TERMINATED BY '\t'                                    #列之间分隔符
location '/ods/kc/fact/ods_kc_fact_clicklog/';        #数据存储路径
```

建表就这么简单，但建表之前得先建数据库，数据库的创建命令如下：

```
create database chongdianleme;
```

然后选择这个数据库：

```
use chongdianleme;
```

Hive 建表的字段类型分为基础数据类型和集合数据类型。

基础数据类型：

Hive 类型	说明	Java 类型	实例
1）.tinyint	1byte 有符号的整数	byte	20
2）.smalint	2byte 有符号的整数	short	20
3）.int	4byte 有符号的整数	int	20
4）.bigint	8byte 有符号的整数	long	20
5）.boolean	布尔类型 true 或 false	boolean	true
6）.float	单精度	float	3.217
7）.double	双精度	double	3.212

8）. string	字符序列，单双即可	string	'chongdianleme'
9）. timestamp	时间戳，精确的纳秒	timestamp	'158030219188'
10）. binary	字节数组	byte[]	

集合数据类型：

Hive 类型	说明	Java 类型	实例
1）. struct	对象类型，可以通过字段名. 元素名来访问	object	struct('name','age')
2）. map	一组键值对的元组	map	map('name','zhangsan','age','23')
3）. array	数组	array	array('name','age')
4）. union	组合		

```
#输入 hive 并按回车键，执行创建表命令
#创建数据库命令
create database chongdianleme;
#使用这个数据库
use chongdianleme;
#ods 层事实表用户查看点击课程日志
create EXTERNAL table IF NOT EXISTS ods_kc_fact_clicklog_tab(userid string,kcid string,time
string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t'
stored as textfile
location '/ods/kc/fact/ods_kc_fact_clicklog_tab/';

#ods 层维表课程商品表
create EXTERNAL table IF NOT EXISTS ods_kc_dim_product_tab(kcid string,kcname string,price
float ,issale string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t'
stored as textfile
location '/ods/kc/dim/ods_kc_dim_product_tab/';
```

2. 查询数据表

1）查询课程日志表前几条记录

```
select * from ods_kc_fact_clicklog_tab limit 6;
```

2）导入一些数据到课程日志表

因为表里开始没有数据，我们需要先将数据导入进去。有多种导入方式，例如：

（1）用 Sqoop 工具从 MySQL 导入。

（2）直接把文本文件放到 Hive 对应的 HDFS 目录下。

```
cd /home/hadoop/chongdianleme
```

```
#rz 上传
#通过 Hadoop 命令上传本地文件到 Hive 表对应的 hdfs 目录
hadoop fs -put kclog.txt /ods/kc/fact/ods_kc_fact_clicklog_tab/

#查看一下此目录,可以看到在这个 Hive 表目录下有数据了
$ hadoop fs -ls /ods/kc/fact/ods_kc_fact_clicklog_tab/
Found 1 items
-rw-r--r-- 3 hadoop supergroup 590 2019-05-29 02:16 /ods/kc/fact/ods_kc_fact_clicklog_
tab/kclog.txt

#通过 Hadoop 的 tail 命令我们可以查看此目录下文件的最后几条记录
$ hadoop fs -tail /ods/kc/fact/ods_kc_fact_clicklog_tab/kclog.txt
u001  kc61800001  2019-06-02 10:01:16
u001  kc61800002  2019-06-02 10:01:17
u001  kc61800003  2019-06-02 10:01:18
u002  kc61800006  2019-06-02 10:01:19
u002  kc61800007  2019-06-02 10:01:20

#然后上传课程商品表
cd /home/hadoop/chongdianleme
#rz 上传
hadoop fs -put product.txt /ods/kc/dim/ods_kc_dim_product_tab/
#查看记录
hadoop fs -tail /ods/kc/dim/ods_kc_dim_product_tab/product.txt
```

3) 简单的查询课程日志表 SQL 语句

```
#查询前几条
select * from ods_kc_fact_clicklog_tab limit 6;
#查询总共有多少条记录
select count(1) from ods_kc_fact_clicklog_tab;
#查看有多少用户
select count(distinct userid) from ods_kc_fact_clicklog_tab;
#查看某个用户的课程日志
select * from ods_kc_fact_clicklog_tab where userid = 'u001';
#查看大于或等于某个时间的日志
select * from ods_kc_fact_clicklog_tab where time >= '2019-06-02 10:01:19';
#查看在售,并且价格大于 2000 元的日志
select * from ods_kc_dim_product where issale = '1' and price > 2000;
#查看在售或者价格大于 2000 元的日志
select * from ods_kc_dim_product where issale = '1' or price > 2000;
```

4) 以\001 分隔符建表

以\001 分割是 Hive 建表中常用的规范,之前用的\t 分隔符容易被用户输入,数据行里如果存在\t 分隔符,会和 Hive 表里的\t 分隔符混淆,这样这一行数据便会多出几列,造成列错乱。

```
#ods 层维表用户查看点击课程日志事实表
create EXTERNAL table IF NOT EXISTS ods_kc_fact_clicklog(userid string,kcid string,time
string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/fact/ods_kc_fact_clicklog/';
#ods 层维表用户查看点击课程基本信息维度表
create EXTERNAL table IF NOT EXISTS ods_kc_dim_product(kcid string,kcname string,price float ,
issale string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/dim/ods_kc_dim_product/';
```

5）基于 SQL 查询结果集合来更新数据表

把查询 SQL 语句的结果集合导出到另外一张表，用 insert overwrite table

这是更新数据表的常用方式，通过 insert overwrite table 可以把指定的查询结果集合插入这个表，插入前先把表清空。如果不加 overwrite 关键词，则不会清空，而是在原来的数据上追加。

```
#先查询 ods_kc_fact_clicklog 这个表有没有记录
select * from chongdianleme.ods_kc_fact_clicklog limit 6;
#把查询结果导入以\001 分割的表，课程日志表
insert overwrite table chongdianleme.ods_kc_fact_clicklog select userid,kcid,time from
chongdianleme.ods_kc_fact_clicklog_tab;
#再查看导入的结果
select * from chongdianleme.ods_kc_fact_clicklog limit 6;
#课程商品表
insert overwrite table chongdianleme.ods_kc_dim_product select kcid,kcname,price,issale from
chongdianleme.ods_kc_dim_product_tab;
#查看课程商品表
select * from chongdianleme.ods_kc_dim_product limit 36;
select * from ods_kc_dim_product where price > 2000;
```

6）join 关联查询——自然连接

join 关联查询可以把多个表以某个字段作为关联，同时获得多个表的字段数据，关联不上的数据将会丢弃。

```
#查询在售课程的用户访问日志
select a.userid,a.kcid,b.kcname,b.price,a.time from chongdianleme.ods_kc_fact_clicklog a
join chongdianleme.ods_kc_dim_product b on a.kcid = b.kcid where b.issale = 1;
```

7）left join 关联查询——左连接

left join 关联查询和自然连接的区别，左边的表没有关联上的数据记录不会丢弃，只是对应的右表那些记录是空值而已。

```
#查询在售课程的用户访问日志
select a.userid, a.kcid, b.kcname, b.price, a.time, b.kcid from chongdianleme.ods_kc_fact_clicklog a left join chongdianleme.ods_kc_dim_product b on a.kcid = b.kcid where b.kcid is null;
```

8）full join 关联查询——完全连接

full join 关联查询不管有没有关联上，所有的数据记录都不会丢弃，关联不上只是显示为空而已。

```
#查询在售课程的用户访问日志
select a.userid, a.kcid, b.kcname, b.price, a.time, b.kcid from chongdianleme.ods_kc_fact_clicklog a full join chongdianleme.ods_kc_dim_product b on a.kcid = b.kcid;
```

9）导入关联表 SQL 结果到新表

```
#创建要导入的表数据
create EXTERNAL table IF NOT EXISTS ods_kc_fact_etlclicklog(userid string,kcid string,time string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/fact/ods_kc_fact_etlclicklog/';
```

把查询集合的结果更新到刚才创建的表里 ods_kc_fact_etlclicklog，先清空，再导入。如果不想清空而是想追加数据则把 overwrite 关键词去掉就可以了。

```
insert overwrite table chongdianleme.ods_kc_fact_etlclicklog select a.userid,a.kcid,a.time from chongdianleme.ods_kc_fact_clicklog a join chongdianleme.ods_kc_dim_product b on a.kcid = b.kcid where b.issale = 1;
```

上面的 SQL 语句都是在 Hive 客户端操作的，执行 SQL 语句所需时间根据数据量和复杂程度不同而不同，如果不触发 MapReduce 计算只需要几毫秒，如果触发了最快也得几秒左右。一般情况下执行几分钟或几个小时很正常。对于执行时间长的 SQL 语句，客户端的计算机如果断电或网络中断，SQL 语句的执行可能也会中断，没有完全执行完整个 SQL 语句，所以在这种情况下我们可以用一个 Shell 脚本把需要执行的 SQL 语句都放在里面，以后就可以用 nohup 后台的方式去执行这个脚本。

3. 通过 Shell 脚本执行 Hive 的 SQL 语句来实现 ETL

```
#创建 demohive.sql 文件
#把下面两条 SQL 语句加进去，每个 SQL 语句后面记得加分号
insert overwrite table chongdianleme.ods_kc_fact_etlclicklog select a.userid,a.kcid,a.time from chongdianleme.ods_kc_fact_clicklog a join chongdianleme.ods_kc_dim_product b on a.kcid = b.kcid where b.issale = 1;
insert overwrite table chongdianleme.ods_kc_dim_product select kcid,kcname,price,issale from chongdianleme.ods_kc_dim_product_tab;
```

```
#创建 demoshell.sh 文件
#加入:echo "通过 Shell 脚本执行 Hive SQL 语句"
/home/hadoop/software/hadoop2/hive/bin/hive -f /home/hadoop/chongdianleme/demohive.sql;
sh demoshell.sh
#或者
sudo chmod 755 demoshell.sh
./demoshell.sh
```

以 nohup 后台进程方式执行 Shell 脚本,防止 xshell 客户端由于断网或者下班后关机或关闭客户端而导致 SQL 执行一部分便退出。

```
#创建 nohupdemoshell.sh 文件
#echo "-- nohup 后台方式执行脚本,断网、关机或客户端关闭无须担忧执行脚本中断";
nohup /home/hadoop/chongdianleme/demoshell.sh >>/home/hadoop/chongdianleme/log.txt 2>&1 &
#执行可能报错
nohup: 无法运行命令'/home/hadoop/chongdianleme/demoshell.sh':      #权限不够
#因为此脚本是不可执行文件
sudo chmod 755 demoshell.sh
sudo chmod 755 nohupdemoshell.sh
```

然后输入 tail -f log.txt 就可以看到实时执行日志。

实际上我们用 Hive 做 ETL 数据处理都可以用这种方式,通过 Shell 脚本来执行 Hive SQL,并且是定时触发,定时触发有几种方式,最简单的方式用 Linux 系统自带的 crontab 调度,但 crontab 调度不支持复杂的任务依赖。这个时候我们可以用 Azkaban、Oozie 来调度。互联网公司使用最普遍的调度方式是 Azkaban 调度。

4. crontab 调度定时执行脚本

这是 Linux 自带的本地系统调度工具,简单好用,通过 crontab 表达式定时触发一个 Shell 脚本。

```
#crontab 调度举例
crontab -e
16 1,2,23 * * * /home/hadoop/chongdianleme/nohupdemoshell.sh
最后保存,重启 cron 服务.
sudo service cron restart
```

5. Azkaban 调度

Azkaban 是一套简单的任务调度服务,整体包括三部分:webserver、dbserver 和 executorserver。Azkaban 是 Linkedin 的开源项目,开发语言为 Java。Azkaban 是由 Linkedin 开源的一个批量工作流任务调度器,用于在一个工作流内以一个特定的顺序运行一组工作和流程。Azkaban 定义了一种 KV 文件格式来建立任务之间的依赖关系,并提供一个易于使用的 Web 用户界面维护和跟踪你的工作流。

Azkaban 实际应用中经常有这些场景:每天有一个大任务,这个大任务可以分成 A,B,C 和 D 4 个小任务,A,B 任务之间没有依赖关系,C 任务依赖 A,B 任务的结果,D 任务依赖

C 任务的结果。一般的做法是，开两个终端同时执行 A，B，两个都执行完了再执行 C，最后执行 D。这样的话，整个执行过程都需要人工参加，并且得盯着各任务的进度，但是我们的很多任务都是在深更半夜执行的，可以通过写脚本设置 crontab 来执行。其实，整个过程类似于一个有向无环图(DAG)。每个子任务相当于大任务中的一个流，任务的起点可以从没有度的节点开始执行，任何没有通路的节点可以同时执行，例如上述的 A，B。总而言之，我们需要的是一个工作流的调度器，而 Azkaban 就是能解决上述问题的一个调度器。

6. Oozie 调度

Oozie 是管理 Hadoop 作业的工作流调度系统，Oozie 的工作流是一系列操作图，Oozie 协调作业是通过时间(频率)及有效数据触发当前的 Oozie 工作流程，Oozie 是针对 Hadoop 开发的开源工作流引擎，专门针对大规模复杂工作流程和数据管道设计。Oozie 围绕两个核心：工作流和协调器，前者定义任务的拓扑和执行逻辑，后者负责工作流的依赖和触发。

这节我们讲的是 Hive 常用 SQL，Hive SQL 能满足多数应用场景，但有的时候需要和自己的业务代码做混合编程来实现复杂的功能，这就需要自定义开发 Java 函数，也就是我们下面要讲解的 UDF 函数。

3.2.4 UDF 函数

Hive SQL 一般可以满足多数应用场景，但是有的时候通过 SQL 实现比较复杂，用一个函数实现会大大简化 SQL 的逻辑，再就是通过自定义函数能够和业务逻辑结合在一起实现更复杂的功能。

1. Hive 类型

Hive 中有 3 种 UDF：

1) 用户定义函数(User-Defined Function，UDF)

UDF 操作作用于单个数据行，并且产生一个数据行作为输出。大多数函数属于这一类，例如数学函数和字符串函数。简单来说，UDF 返回对应值，一对一。

2) 用户定义聚集函数(User-Defined Aggregate Function，UDAF)

UDAF 接收多个输入数据行，并产生一个输出数据行。像 COUNT 和 MAX 这样的函数就是聚集函数。简单来说，UDAF 返回聚类值，多对一。

3) 用户定义表生成函数(User-Defined Table-generating Function，UDTF)

UDTF 操作作用于单个数据行，并且产生多个数据行而生成一个表作为输出。简单来说，UDTF 返回拆分值，一对多。

在实际工作中 UDF 用得最多，下面我们重点讲解第一种 UDF 函数，也就是用户定义函数。

2. UDF 自定义函数

Hive 的 SQL 给数据挖掘工作者带来了很多便利，海量数据通过简单的 SQL 语句就可

以完成分析，但有时候 Hive 提供的函数功能满足不了业务需要，这就需要我们自己写 UDF 函数来辅助完成。UDF 函数其实就是一个简单的函数，执行过程就是在 Hive 将 UDF 函数转换成 MapReduce 程序后，执行 Java 方法，类似于在 MapReduce 执行过程中加入一个插件，方便扩展。UDF 只能实现一进一出的操作，如果需要实现多进一出，则需要实现 UDAF。Hive 可以允许用户编写自己定义的函数 UDF，并在查询中使用。我们自定义开发 UDF 函数的时候继承 org. apache. hadoop. hive. ql. exec. UDF 类即可，代码如下：

```
package com.chongdianleme.hiveudf.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
//自定义类继承 UDF
public class HiveUDFTest extends UDF {
        //字符串统一转大写字符串示例
    public String evaluate (String str){
        if(str == null || str.toString().isEmpty()){

            return new String();
        }
        return new String(str.trim().toUpperCase());
    }
}
```

下面看一下怎么部署，部署也分临时部署方式和永久生效部署方式，我们分别来讲解。

3. 临时部署测试

部署脚本代码如下：

```
#把程序打包并放到目标机器上
#进入 Hive 客户端，添加 jar 包
hive > add jar /home/hadoop/software/task/HiveUDFTest.jar;
#创建临时函数
hive > CREATE TEMPORARY FUNCTION ups AS 'hive.HiveUDFTest';
add jar /home/hadoop/software/task/udfTest.jar;
create temporary function row_toUpper as 'com.chongdianleme.hiveudf.udf.HiveUDFTest';
```

4. 永久全局方式部署

线上永久配置方式，部署脚本代码如下：

```
cd /home/hadoop/software/hadoop2/hive
#创建 auxlib 文件夹
cd auxlib
#在/home/hadoop/software/hadoop2/hive/auxlib 上传 udf 函数的 jar 包. Hive SQL 执行
#时会自动扫描/data/software/hadoop/hive/auxlib 下的 jar 包
cd /home/hadoop/software/hadoop2/hive/bin
#显示隐藏文件
ls -a
#编辑 vi .hiverc 文件加入
```

```
create temporary function row_toUpper as 'com.chongdianleme.hiveudf.udf.HiveUDFTest';
```

之后输入 Hive 命令登录客户端就可以了，客户端会自动扫描并加载所有的 UDF 函数。以上我们讲的 Hive 常用 SQL 和 UDF，以及怎么用 Shell 脚本触发执行 SQL，怎么去做定时的调度。在实际工作中，并不是盲目随意地去建表，一般都会制定一个规范，大家遵守这个规范去执行。这个规范就是我们下面要讲的数据仓库规范和模型设计。

3.2.5 Hive 数据仓库模型设计

数据仓库模型设计就是要制定一个规范，这个规范一般是做数据仓库的分层设计。我们要搭建数据仓库，把握好数据质量，对数据进行清洗、转换。要更好地区分哪个是原始数据，哪个是清洗后的数据，我们最好做一个数据分层，方便我们快速地找到想要的数据。另外，有些高频的数据不需要每次都重复计算，只需要计算一次并放在一个中间层里，供其他业务模块复用，这样节省时间，同时也减少服务器资源的消耗。数据仓库分层设计还有其他很多好处，下面举一个实例看看如何分层。

数据仓库，英文名称为 Data Warehouse，可简写为 DW 或 DWH。数据仓库是为企业所有级别的决策制定过程提供所有类型数据支持的战略集合。它是单个数据存储，出于分析性报告和决策支持目的而创建。为需要业务智能的企业提供指导业务流程改进、监视时间、成本、质量及控制。

我们再看一下什么是数据集市，数据集市(Data Mart)，也叫数据市场，数据集市就是满足特定的部门或者用户需求，按照多维的方式进行存储，包括定义维度、需要计算的指标、维度的层次等，生成面向决策分析需求的立方体数据。从范围上来说，数据是从企业范围的数据库、数据仓库，或者是更加专业的数据仓库中抽取出来的。数据中心的重点就在于它迎合了专业用户群体在分析、内容、表现及易用方面的特殊需求。数据中心的用户希望数据是由他们熟悉的术语来表现的。

上面我们说的是数据仓库和数据集市的概念，简单来说，在 Hadoop 平台上的整个 Hive 的所有表构成了数据仓库，这些表是分层设计的，我们可以分为 4 层：ods 层、mid 层、tp 临时层和数据集市层。其中数据集市可以看作数据仓库的一个子集，一个数据集市往往是针对一个项目的，例如推荐的叫推荐集市，做用户画像的项目叫用户画像集市。ods 是基础数据层，也是原始数据层，是最底层的，数据集市是偏最上游的数据层。数据集市的数据可以直接供项目使用，不用再多地去加工了。

数据仓库的分层体现在 Hive 数据表名上，Hive 存储对应的 HDFS 目录最好和表名一致，这样根据表名也能快速地找到目录，当然这不是必需的。一般大数据平台都会创建一个数据字典平台，在 Web 的界面上能够根据表名找到对应的表解释，例如表的用途、字段表结构、每个字段代表什么意思、存储目录等，而且能查询到表和表之间的血缘关系。说到血缘关系在数据仓库里经常会提起这一关系。我们在下面会单独讲一小节。下面用实例讲解推荐的数据仓库。

首先我们需要和部门所有的人制定一个建表规范，大家统一遵守这个规则。

1. 建表规范

以下建表规范仅供参考，可以根据每个公司的实际情况来制定。

1）统一创建外部表

外部表的好处是当你不小心删除了这个表，数据还会保留下来，如果是误删除，会很快地找回来，只需要把建表语句再创建一遍即可。

2）统一分 4 级，以下画线分割

分为几个级别没有明确的规定，一般分为 4 级的情况比较多。

3）列之间分隔符统一'\001'

用\001 分割的目的是为了避免因为数据也存在同样的分隔符而造成列的错乱问题。因为\001 分割符是用户不容易输入的，之前用的\t 分隔符容易被用户输入，数据行里如果存在\t 分隔符，会和 Hive 表里的\t 分隔符混淆，这样这一行数据会多出几列，造成列错乱。

4）location 指定目录统一以/结尾

指定目录统一以/结尾代表最后是一个文件夹，而不是一个文件。一个文件夹下面可以有很多文件，如果数据特别大，适合拆分成多个小文件。

5）stored 类型统一 textfile

每个公司实际情况不太一样，textfile 是文本类型文件，好处是方便查看内容，不好的地方是占用空间较大。

6）表名和 location 指定目录保持一致

表名和 location 指定目录保持一致的主要目的是为了方便见到表名就马上可以知道对应的数据存储目录在哪里，方便检索和查找。

```
#下面列举一个建表的例子给大家做一个演示
create EXTERNAL table IF NOT EXISTS ods_kc_dim_product(kcid string,kcname string,price float ,
issale string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/dim/ods_kc_dim_product/';
```

2. 数据仓库分层设计规范

上面我们建表的时候已经说了数据仓库分为 4 级，也就是说我们的数据仓库分为 4 层，即操作数据存储原始数据的 ods 层、mid 层、tp 临时层和数据集市层，下面一一讲解。

1）ods 层

操作数据存储 ODS(Operational Data Store)用来存放原始基础数据，例如维表、事实表。以下画线分为 4 级：

(1) 原始数据层；

(2) 项目名称(kc 代表视频课程类项目，Read 代表阅读类文章)；

(3) 表类型(dim 为维度表,fact 为事实表);

(4) 表名。

举几个例子:

```
#原始数据_视频课程_事实表_课程访问日志表
create EXTERNAL table IF NOT EXISTS ods_kc_fact_clicklog(userid string,kcid string,time
string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/fact/ods_kc_fact_clicklog/';
#ods 层维度表,课程基本信息表
create EXTERNAL table IF NOT EXISTS ods_kc_dim_product(kcid string,kcname string,price float ,
issale string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/dim/ods_kc_dim_product/';
```

这里涉及新的概念,什么是维度表和事实表?

事实表:

在多维数据仓库中,保存度量值的详细值或事实的表称为“事实表”。事实数据表通常包含大量的行。事实数据表的主要特点是包含数字数据(事实),并且这些数字信息可以汇总,以提供有关单位作为历史的数据,每个事实数据表包含一个由多个部分组成的索引,该索引包含作为外键的相关性维度表的主键,而维度表包含事实记录的特性。事实数据表不应该包含描述性的信息,也不应该包含除数字度量字段及事实与维度表中对应项的相关索引字段之外的任何数据。

维度表:

维度表可以看作用户用来分析数据的窗口,维度表中包含事实数据表中事实记录的特性,有些特性提供描述性信息,有些特性指定如何汇总事实数据表数据,以便为分析者提供有用的信息,维度表包含帮助汇总数据的特性的层次结构。例如,包含产品信息的维度表通常包含将产品分为食品、饮料和非消费品等若干类的层次结构,这些产品中的每一类进一步多次细分,直到各产品达到最低级别。在维度表中,每个表都包含独立于其他维度表的事实特性,例如,客户维度表包含有关客户的数据。维度表中的列字段可以将信息分为不同层次的结构级。维度表包含了维度的每个成员的特定名称。维度成员的名称称为“属性”(Attribute)。

在我们的推荐场景中,例如这个课程访问日志表 ods_kc_fact_clicklog,数据都是用户访问课程的大量日志,针对每条记录也没有一个实际意义的主键,同一个用户有多条课程访问记录,同一个课程也会被多个用户访问,这个表就是事实表。在课程基本信息表 ods_kc_dim_product 中,每个课程都有一个唯一的课程主键,课程具有唯一性。每个课程都有基本属性。这个表就是维度表。

2）mid 层

mid 层是从 ods 层中 join 多表或某一段时间内的小表计算生成的中间表，在后续的集市层中频繁被使用。用来一次生成多次使用，避免每次关联多个表重复计算。

从 ods 层提取数据到集市层常用 SQL 方式：

```
#把某个 select 的查询结果集覆盖到某个表，相当于 truncate 和 insert 的操作
insert overwrite table chongdianleme.ods_kc_fact_etlclicklog select a.userid,a.kcid,a.time
from chongdianleme.ods_kc_fact_clicklog a join chongdianleme.ods_kc_dim_product b on a.kcid =
b.kcid where b.issale = 1;
```

3）tp 临时层

temp 临时层简称 tp，临时生成的数据统一放在这一层。系统默认有一个/tmp 目录，不要将数据放在这一目录里，这个目录很多数据是 Hive 本身存放在这一临时层的，我们不要跟它混在一起。

```
#建表举例
create EXTERNAL table IF NOT EXISTS tp_kc_fact_clicklogtotemp(userid string,kcid string,time
string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/tp/kc/fact/tp_kc_fact_clicklogtotemp/';
```

4）数据集市层

例如，用户画像集市、推荐集市和搜索集市等。数据集市层用于存放搜索项目数据，集市数据一般是由中间层和 ods 层关联表计算所得，或使用 Spark 程序处理、开发并算出来的数据。

```
#用户画像集市建表举例
create EXTERNAL table IF NOT EXISTS personas_kc_fact_userlog(userid string,kcid string,name
string,age string,sex string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/personas/kc/fact/personas_kc_fact_userlog/';
```

从开发人员的角色来划分，此工作是由专门的数据仓库工程师来负责，当然如果预算有限，也可以由大数据 ETL 工程师来负责。

Hive 非常适合离线的数据处理分析，但有些场景需要对数据做实时处理，而 HBase 数据库特别适合处理实时数据，下面我们来讲解 HBase。

3.3 HBase 实战

HBase 经常用来存储实时数据，例如 Storm/Flink/Spark Streaming 消费用户行为日志数据进行处理后存储到 HBase，我们通过 HBase 的 API 也能够毫秒级地实时查询。如果

是对 HBase 做非实时的离线数据统计，我们可以通过 Hive 建一个到 HBase 的映射表，然后写 Hive SQL 来对 HBase 的数据进行统计分析，并且这种方式可以方便地和其他的 Hive 表做关联查询，或者做更复杂的统计，所以从交互形势上 HBase 满足了实时和离线的应用场景，在互联网公司应用得也非常普遍。

3.3.1 HBase 原理和功能介绍

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

1. HBase 特性

1）HBase 构建在 HDFS 之上

HBase 是一个构建在 HDFS 上的分布式列存储系统，可以通过 Hive 的方式来查询 HBase 数据。

2）HBase 是 key/value 系统

HBase 是基于 Google Bigtable 模型开发的，是典型的 key/value 系统。

3）HBase 用于海量结构化数据存储

HBase 是 Apache Hadoop 生态系统中的重要一员，主要用于海量结构化数据存储。

4）分布式存储

HBase 将数据按照表、行和列进行存储。与 Hadoop 一样，HBase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器来增加计算和存储能力。

5）HBase 表和列都大

HBase 表的特点是大，一个表可以有数十亿行，上百万列。

6）无模式

每行都有一个可排序的主键和任意多的列，列可以根据需要动态地增加，同一张表中不同的行可以有截然不同的列，这是 MySQL 关系数据库做不到的。

7）面向列

面向列（族）的存储和权限控制，列（族）独立检索；空（null）列并不占用存储空间，表可以设计得非常稀疏。

8）数据多版本

每个单元中的数据可以有多个版本，默认 3 个版本，是单元格插入时的时间戳。

2. HBase 的架构核心组件

HBase 架构的核心组件有 Client、Hmaster、HRegionServer 和 ZooKeeper 集群协调系统

等,最核心的是 HMaster 和 HRegionServer,HMaster 是 HBase 的主节点,HRegionServer 是从节点。HBase 必须依赖于 ZooKeeper 集群。

1) Client

访问 HBase 的接口,并维护 Cache 来加快对 HBase 的访问,例如 Region 的位置信息。

2) HMaster

(1) 管理 HRegionServer,实现其负载均衡。

(2) 管理和分配 HRegion,例如在 HRegion split 时分配新的 HRegion; 在 HRegionServer 退出时迁移其内的 HRegion 到其他 HRegionServer 上。

(3) 实现 DDL 操作(Data Definition Language,namespace 和 table 的增删改,column family 的增删改等)。

(4) 管理 namespace 和 table 的元数据(实际存储在 HDFS 上)。

(5) 权限控制(ACL)。

3) HRegionServer

(1) 存放和管理本地 HRegion。

(2) 读写 HDFS,管理 Table 中的数据。

(3) Client 直接通过 HRegionServer 读写数据(从 HMaster 中获取元数据,找到 RowKey 所在的 HRegion/HRegionServer 后)。

4) ZooKeeper 集群协调系统

(1) 存放整个 HBase 集群的元数据及集群的状态信息。

(2) 实现 HMaster 主从节点的 failover。

HBase Client 通过 RPC 方式和 HMaster、HRegionServer 通信,一个 HRegionServer 可以存放 1000 个 HRegion,底层 Table 数据存储在 HDFS 中,而 HRegion 所处理的数据尽量和数据所在的 DataNode 在一起,实现数据的本地化。

3.3.2 HBase 数据结构和表详解

HBase 数据表由行键、列族组成,行键可以认为是数据库的主键,一个列族下面可以有多个列,并且列可以动态地增加,这是 HBase 的优势,本身就是一个列式存储的数据库,这点和 MySQL 关系数据库不一样,MySQL 一旦列固定了,就不能动态增加了。这点 HBase 非常灵活,可以根据业务需要动态地创建一个列。下面我们看一下表结构都由什么组成。

1. 行键 Row Key

主键用来检索记录的主键,访问 HBase Table 中的行。

2. 列族 ColumnFamily

Table 在水平方向由一个或者多个 ColumnFamily 组成,一个 ColumnFamily 可以由任意多个 Column 组成,即 ColumnFamily 支持动态扩展,无须预先定义 Column 的数量及类型,所有 Column 均以二进制格式存储,用户需要自行进行类型转换。

3. 列 column

由 HBase 中的列族 ColumnFamily + 列的名称(cell)组成列。

4. 单元格 cell

HBase 中通过 row 和 columns 确定列,一个存储单元称为 cell。

5. 版本 version

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,默认 3 个版本。

下面是一个 HBase 数据结构表实例,如表 3.1 所示。

表 3.1 HBase 表结构说明

rowkey(行键)	name(名称,单个列的列族)	kcsaleinfo(课程出售信息,多个列的列族)	
	kcname(课程名称)	price	issale
kc61800001	机器学习	6998 元	1.0 version(版本)
			2.0
			3.0

此例表中有一条数据,rowkey 主键是 kc61800001,两个列族,一个是 name,它只有一个列 kcname; 另一个是 kcsaleinfo,有两个列 price 和 issale。这是一个具体的例子,下面我们看看 HBase 如何安装部署。

3.3.3 HBase 安装部署

HBase 相对 Hadoop 来说安装比较简单,由于它依赖 ZooKeeper 集群,所以安装 HBase 之前需要事先安装好 ZooKeeper 集群。下面我们看一下 HBase 的安装步骤。

1. 先修改 Hadoop 的配置

```
#修改 etc/hadoop/hdfs-site.xml 里面的 xcievers 参数,至少为 4096
vim etc/hadoop/hdfs-site.xml
<property>
<name>dfs.datanode.max.xcievers</name>
<value>4096</value>
</property>
```

完成后,重启 Hadoop 的 HDFS 系统。

2. HBase 修改部分

```
#上传并解压 hbase 的 tar 包,修改 3 个配置文件
hbase/conf/hbase-env.sh
hbase/conf/hbase-site.xml
hbase/conf/regionservers
```

1）修改 hbase-env.sh 文件配置

```
vim hbase/conf/hbase - env.sh
#注意:HBASE_MANAGES_ZK 为 true 是 HBase 托管的 ZooKeeper.我们使用自己的 5 台 ZooKeeper,
#需要设置为 false
export JAVA_HOME = /usr/local/Java/jdk
export HBASE_MANAGES_ZK = false
export HBASE_HEAPSIZE = 8096
```

HBase 对于内存要求很高,在硬件允许的情况下配足够多的内存供它使用。HBASE_HEAPSIZE 默认 1GB,当数据量大的时候宕机频率很高。改成 8GB 基本上就很稳定了。

2）修改配置文件 hbase-site.xml

```
vim /home/hadoop/software/hbase/conf/hbase - site.xml
#另外就是 NameNode HA 模式需要把 Hadoop 的 hdfs - site.xml 复制到 hbase/conf 下,否则
#报错,ai 找不到主机名
< configuration >
< property >
< name > hbase.rootdir </name >
< value > hdfs://ai/hbase/</value >
</property >
< property >
< name > hbase.cluster.distributed </name >
< value > true </value >
</property >
< property >
< name > hbase.zookeeper.property.clientPort </name >
< value > 2181 </value >
</property >
< property >
< name > hbase.zookeeper.quorum </name >
< value > data1,data2,data3,data4,data5 </value >
</property >
< property >
< name > hbase.master.maxclockskew </name >
< value > 200000 </value >
</property >
< property >
< name > hbase.tmp.dir </name >
< value >/home/hadoop/software/hbase - 0.98.8 - hadoop2/tmp </value >
</property >
< property >
< name > zookeeper.session.timeout </name >
< value > 1200000 </value >
</property >
< property >
< name > hbase.regionserver.handler.count </name >
```

```
< value > 50 </value >
</property >
< property >
< name > hbase. client. write. buffer </name >
< value > 8388608 </value >
</property >
</configuration >
```

3）修改配置文件 regionservers

```
vim  hbase/conf/regionservers
```

加入节点的主机名：

```
data1
data2
data3
data4
data5
```

HBase 配置文件都修改好了，scp 到其他节点：

```
scp -r hbase hadoop@data2:/home/hadoop/software/
```

在任意一台启动 HBase：

```
/home/hadoop/software/hbase/bin/start - hbase. sh
```

然后看一下启动情况：
登录 hbase shell，输入 status 查看集群状态。
单独启动一个 HMaster 进程：bin/hbase-daemon. sh start master
停止：bin/hbase-daemon. sh stop master
单独启动一个 HRegionServer 进程：bin/hbase-daemon. sh start regionserver
停止：bin/hbase-daemon. sh stop regionserver
Hbase 的启动常见错误：

```
org. apache. hadoop. hbase. TableExistsException: hbase:namespace
    at org. apache. hadoop. hbase. master. handler. CreateTableHandler. prepare ( CreateTableHandler.
Java:133)
    at org. apache. hadoop. hbase. master. TableNamespaceManager. createNamespaceTable( TableNam-
espaceManager. Java:232)
    at org. apache. hadoop. hbase. master. TableNamespaceManager. start ( TableNamespaceManager.
Java:86)
    at org. apache. hadoop. hbase. master. HMaster. initNamespace( HMaster. Java:1063)
    at org. apache. hadoop. hbase. master. HMaster. finishInitialization( HMaster. Java:942)
    at org. apache. hadoop. hbase. master. HMaster. run( HMaster. Java:613)
    at Java. lang. Thread. run( Thread. Java:745)
```

错误原因：

ZooKeeper 里的/hbase 目录已经存在。

解决：登录 ZooKeeper 并删除/hbase 目录，HBase 启动的时候会自动创建这个目录。

```
/home/hadoop/software/zookeeper-3.4.6/bin/zkCli.sh -server 172.172.0.11:2181
[zk: 172.172.0.11:2181(CONNECTED) 0] ls /
[configs, zookeeper, overseer, aliases.json, live_nodes, collections, overseer_elect,
security.json,
hadoop-ha, clusterstate.json, hbase]
#删除目录:
rmr /hbase
```

3. HBase 的 Web 界面

HBase 的 Web 界面，默认是 60010 端口：http://ip:60010/

从这个 Web 界面可以比较方便地看到有几个 RegionServer 节点，以及每个节点内存消耗情况，还有其他很多信息。如果少了 RegionServer 节点，我们可以认为那个节点出问题了，需要我们手动地去启动 RegionServer 服务并查看问题的原因。HBase 的 Web 界面如图 3.3 所示。

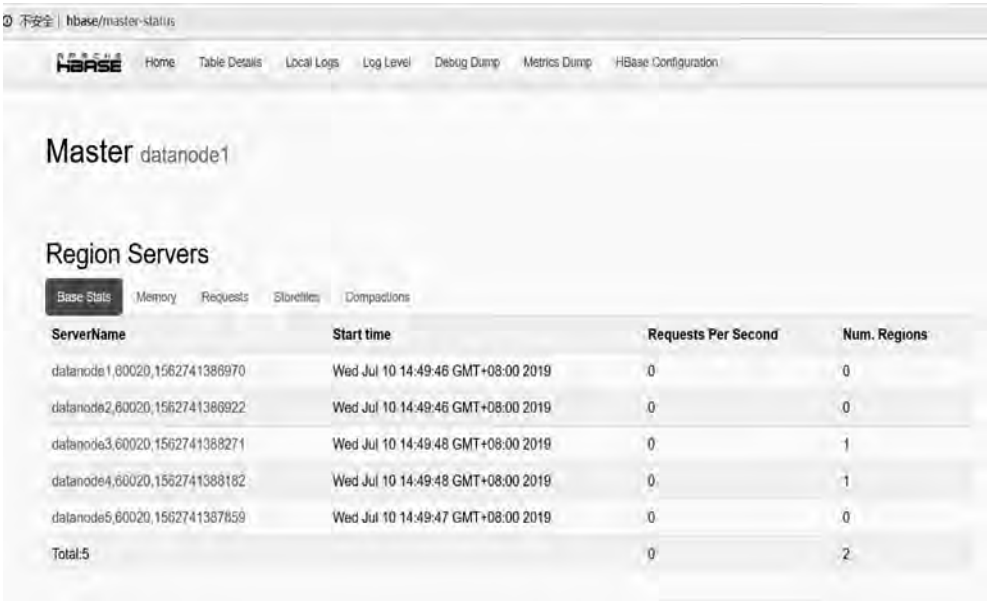

图 3.3 HBase 的 Web 界面

通过内存消耗情况 Tab 页，能方便地知道每个节点 Heap 内存消耗情况，如果使用的 Used Heap 将要超过 Max Heap，我们需要关注是否需要修改配置而把 Max Heap 调大，说明现有的配置已经不够用了。另外，需要查看程序是否可以进行优化来减小内存的消耗。HBase 内存消耗的 Web 界面如图 3.4 所示。

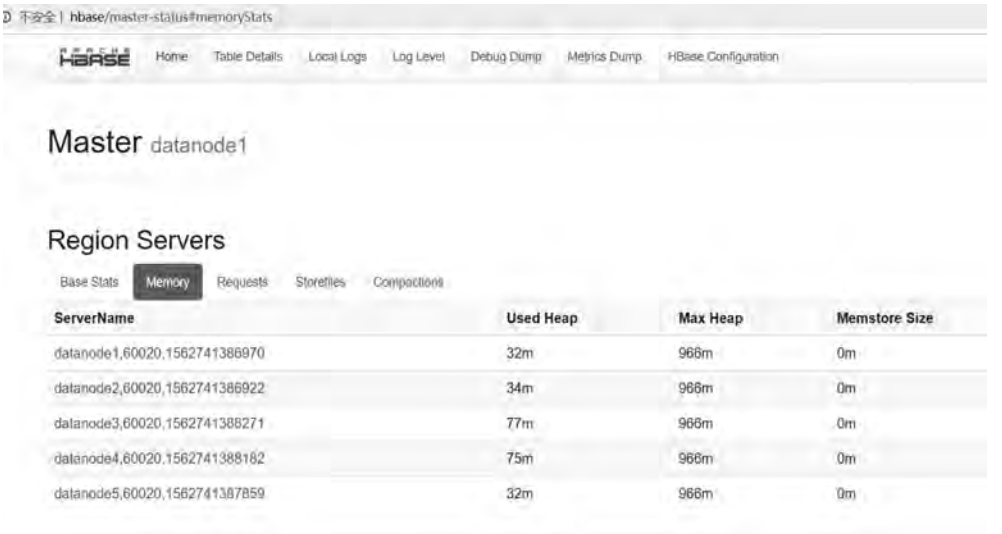

图 3.4 HBase 内存消耗的 Web 界面

3.3.4 HBase Shell 常用命令操作

HBase 数据交互有几种方式，调用 Java API、HBase Shell、Hive 集成 HBase 查询和 Phoenix 工具等都可以操作。

HBase Shell 是 HBase 自带的客户端工具，常用操作命令如下：

```
创建表:create '表名称', '列名称 1','列名称 2','列名称 N'
添加记录: put '表名称', '行名称', '列名称:', '值'
查看记录:get '表名称', '行名称'
查看表中的记录总数:count '表名称'
删除记录:delete '表名','行名称', '列名称'
删除一张表:先要屏蔽该表,才能对该表进行删除,第一步 disable '表名称';第二步 drop '表名称'
查看所有记录:scan "表名称"
查看某个表,某个列中所有数据: scan "表名称" , ['列名称:']
更新记录: 还是用 put 命令,会覆盖之前的老版本记录
```

下面我们通过举例的方式来实际看一下更多具体的命令如何使用。

1. 查看集群状态

```
hbase(main):002:0 > status
5 servers, 0 dead, 0.4000 average load
```

2. 查看 HBase 版本

```
version
```

3. 创建一个表

```
#格式: create 表名,列族 1,列族 2...列族 N
create 'chongdianleme_kc','kcname','saleinfo'
```

运行结果：

```
hbase(main):106:0 > create 'chongdianleme_kc','kcname','saleinfo'
0 row(s) in 0.3710 seconds
=> Hbase::Table - chongdianleme_kc
```

4. 查看表描述

```
describe 'chongdianleme_kc'
hbase(main):002:0 > describe 'chongdianleme_kc'
Table chongdianleme_kc is ENABLED
COLUMN FAMILIES DESCRIPTION
{NAME => 'kcname', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_
CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'F
OREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE =>
'65536', REPLICATION_SCOPE => '0'}
{NAME => 'saleinfo', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_
DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL =>
'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE =>
'65536', REPLICATION_SCOPE => '0'}
2 row(s) in 0.1610 seconds
```

5. 删除一个列族

```
#先关闭,再更新,然后再打开
disable 'chongdianleme_kc'
alter'chongdianleme_kc',NAME =>'kcname',METHOD =>'delete'
enable 'chongdianleme_kc'
hbase(main):004:0 > alter'chongdianleme_kc',NAME =>'kcname',METHOD =>'delete'
Updating all regions with the new schema…
1/1 regions updated.
Done.
0 row(s) in 1.2900 seconds
```

6. 列出所有表

```
list
hbase(main):108:0 > list
TABLE
chongdianleme_kc
1 row(s) in 0.0060 seconds
["chongdianleme_kc"]
```

7. 删除一个表

```
#先关闭,再删除
disable 'chongdianleme_kc'
drop 'chongdianleme_kc'
```

如果直接 drop 会提示并报错：

```
hbase(main):010:0> drop 'chongdianleme_kc'
ERROR: Table chongdianleme_kc is enabled. Disable it first.'
Here is some help for this command:
Drop the named table. Table must first be disabled:
  hbase> drop 't1'
  hbase> drop 'ns1:t1'
```

8. 查询表是否存在

```
exists 'chongdianleme_kc'
```

9. 判断表是否 enable

```
is_enabled 'chongdianleme_kc'
```

10. 判断表是否 disable

```
is_disabled 'chongdianleme_kc'
```

11. 插入数据

```
#在列族中插入数据,格式:put 表名,行键 id,列族名:列名,值
create 'chongdianleme_kc','kcname','saleinfo'

put 'chongdianleme_kc','kc61800001','kcname:name','大数据开发'
put 'chongdianleme_kc','kc61800001','saleinfo:price','2888'
put 'chongdianleme_kc','kc61800001','saleinfo:issale','1'

put 'chongdianleme_kc','kc61800002','kcname:name','Java 教程'
put 'chongdianleme_kc','kc61800002','saleinfo:price','199'
put 'chongdianleme_kc','kc61800002','saleinfo:issale','0'

put 'chongdianleme_kc','kc61800003','kcname:name','Python 编程教程'
put 'chongdianleme_kc','kc61800003','saleinfo:price','99'
put 'chongdianleme_kc','kc61800003','saleinfo:issale','1'

put 'chongdianleme_kc','kc61800006','kcname:name','深度学习'
put 'chongdianleme_kc','kc61800006','saleinfo:price','3999'
put 'chongdianleme_kc','kc61800006','saleinfo:issale','1'
```

```
put 'chongdianleme_kc','kc61800007','kcname:name','推荐系统'
put 'chongdianleme_kc','kc61800007','saleinfo:price','2999'
put 'chongdianleme_kc','kc61800007','saleinfo:issale','1'

put 'chongdianleme_kc','kc61800008','kcname:name','机器学习'
put 'chongdianleme_kc','kc61800008','saleinfo:price','2800'
put 'chongdianleme_kc','kc61800008','saleinfo:issale','1'

put 'chongdianleme_kc','kc61800009','kcname:name','TensorFlow 教程'
put 'chongdianleme_kc','kc61800009','saleinfo:price','888'
put 'chongdianleme_kc','kc61800009','saleinfo:issale','1'

put 'chongdianleme_kc','kc61800010','kcname:name','安卓开发教程'
put 'chongdianleme_kc','kc61800010','saleinfo:price','88'
put 'chongdianleme_kc','kc61800010','saleinfo:issale','0'

put 'chongdianleme_kc','kc20000099','kcname:name','Go 语言'
put 'chongdianleme_kc','kc20000099','saleinfo:price','99'
put 'chongdianleme_kc','kc20000099','saleinfo:issale','0'
```

12. 获取一个 id 的所有数据

```
get 'chongdianleme_kc','kc61800001'
hbase(main):185:0* get 'chongdianleme_kc','kc61800001'
COLUMN CELL
kcname:name timestamp=1562812596745, value=\xE5\xA4\xA7\xE6\x95\xB0\xE6\x8D\xAE\xE5\xBC\
x80\xE5\x8F\x91
saleinfo:issale timestamp=1562812596808, value=1
saleinfo:price timestamp=1562812596787, value=2888
3 row(s) in 0.0330 seconds
```

13. 获取一个 id,一个列族的所有数据

```
get 'chongdianleme_kc','kc61800001','saleinfo'
hbase(main):186:0> get 'chongdianleme_kc','kc61800001','saleinfo'
COLUMN CELL
saleinfo:issale timestamp=1562812596808, value=1
saleinfo:price timestamp=1562812596787, value=2888
2 row(s) in 0.0100 seconds
```

14. 获取一个 id,一个列族中一个列的所有数据

```
get 'chongdianleme_kc','kc61800001','saleinfo:price'
hbase(main):187:0> get 'chongdianleme_kc','kc61800001','saleinfo:price'
COLUMN CELL
saleinfo:price timestamp=1562812596787, value=2888
```

```
1 row(s) in 0.0100 seconds
```

15. 更新一条记录

```
#给 rowId 重新 put 即可
#默认保留最近 3 个版本的数据,更新后展示最新版本的数据,但之前两个版本的数据还是能够查询
#到,只是默认不显示出来而已
put 'chongdianleme_kc','kc61800001','saleinfo:price','6000'
```

16. 通过 timestamp 来获取指定版本的数据

```
#先看一下这条数据时间戳 get 'chongdianleme_kc','kc61800001','saleinfo:price'
#然后查找指定这个时间的数据
get 'chongdianleme_kc','kc61800001',{COLUMN = >'saleinfo:price',TIMESTAMP = > 1562809654418}
get 'chongdianleme_kc','kc61800001',{COLUMN = >'saleinfo:price', VERSIONS = > 3}
```

17. 全表扫描

```
scan 'chongdianleme_kc'
hbase(main):188:0 > scan 'chongdianleme_kc'
ROW COLUMN + CELL
kc20000099 column = kcname:name, timestamp = 1562812597085, value = go\xE8\xAF\xAD\xE8\xA8\x80
kc20000099 column = saleinfo:issale, timestamp = 1562812597107, value = 0
kc20000099 column = saleinfo:price, timestamp = 1562812597097, value = 99
kc61800001 column = kcname:name, timestamp = 1562812596745, value = \xE5\xA4\xA7\xE6\x95\xB0\
xE6\x8D\xAE\xE5\xBC\x80\xE5\x8F\x91
kc61800001 column = saleinfo:issale, timestamp = 1562812596808, value = 1
kc61800001 column = saleinfo:price, timestamp = 1562812596787, value = 2888
kc61800002  column = kcname:name, timestamp = 1562812596827, value = Java\xE6\x95\x99\xE7\xA8\x8B
kc61800002 column = saleinfo:issale, timestamp = 1562812596851, value = 0
kc61800002 column = saleinfo:price, timestamp = 1562812596839, value = 199
kc61800003 column = kcname:name, timestamp = 1562812596866, value = python\xE7\xBC\x96\xE7\
xA8\x8B\xE6\x95\x99\xE7\xA8\x8B
kc61800003 column = saleinfo:issale, timestamp = 1562812596890, value = 1
kc61800003 column = saleinfo:price, timestamp = 1562812596877, value = 99
kc61800006 column = kcname:name, timestamp = 1562812596906, value = \xE6\xB7\xB1\xE5\xBA\xA6\
xE5\xAD\xA6\xE4\xB9\xA0
kc61800006 column = saleinfo:issale, timestamp = 1562812596926, value = 1
kc61800006 column = saleinfo:price, timestamp = 1562812596917, value = 3999
kc61800007 column = kcname:name, timestamp = 1562812596944, value = \xE6\x8E\xA8\xE8\x8D\x90\
xE7\xB3\xBB\xE7\xBB\x9F
kc61800007 column = saleinfo:issale, timestamp = 1562812596963, value = 1
kc61800007 column = saleinfo:price, timestamp = 1562812596954, value = 2999
kc61800008 column = kcname:name, timestamp = 1562812596978, value = \xE6\x9C\xBA\xE5\x99\xA8\
xE5\xAD\xA6\xE4\xB9\xA0
kc61800008 column = saleinfo:issale, timestamp = 1562812596998, value = 1
kc61800008 column = saleinfo:price, timestamp = 1562812596988, value = 2800
```

```
kc61800009 column=kcname:name, timestamp=1562812597012, value=TensorFlow\xE6\x95\x99\
xE7\xA8\x8B
kc61800009 column=saleinfo:issale, timestamp=1562812597031, value=1
kc61800009 column=saleinfo:price, timestamp=1562812597022, value=888
kc61800010 column=kcname:name, timestamp=1562812597047, value=\xE5\xAE\x89\xE5\x8D\x93\
xE5\xBC\x80\xE5\x8F\x91\xE6\x95\x99\xE7\xA8\x8B
kc61800010 column=saleinfo:issale, timestamp=1562812597068, value=0
kc61800010 column=saleinfo:price, timestamp=1562812597056, value=88
9 row(s) in 0.0320 seconds
```

18. 删除 id 为 kc61800001 的值的'saleinfo：price'字段

```
delete 'chongdianleme_kc','kc61800001','saleinfo:price'
hbase(main):189:0> delete 'chongdianleme_kc','kc61800001','saleinfo:price'
0 row(s) in 0.0390 seconds
```

19. 删除整行

```
deteleall 'chongdianleme_kc','kc61800001'
```

20. 查询表中有多少行

```
count 'chongdianleme_kc'
hbase(main):190:0> count 'chongdianleme_kc'
9 row(s) in 0.0250 seconds
9
```

21. 将整张表清空

```
#实际执行过程:HBase 先将表 disable,然后 drop,最后重建表来实现 truncate 的功能
truncate 'chongdianleme_kc'
```

3.3.5 HBase 客户端类 SQL 工具 Phoenix

Phoenix 是构建在 HBase 上的一个 SQL 层，能让我们用标准的 JDBC API 而不是 HBase 客户端 API 来创建表，插入数据和对 HBase 数据进行查询。Phoenix 完全使用 Java 编写，作为 HBase 内嵌的 JDBC 驱动。Phoenix 查询引擎会将 SQL 查询转换为一个或多个 HBase Scan，并编排执行以生成标准的 JDBC 结果集。简单来说有点像 Hive SQL 解析成 MapReduce。这比使用 HBase Shell 命令方便多了。

1. Phoenix 安装部署

1）解压安装 Phoenix

Phoenix 是一个压缩包，是一个客户端，首先需要解压缩出来。

2）复制依赖的 jar 包

```
#复制 phoenix 安装目录下的
phoenix-core-4.6.0-HBase-0.98.jar
phoenix-4.6.0-HBase-0.98-client.jar
phoenix-4.6.0-HBase-0.98-server.jar
#到各个 hbase 的 lib 目录下
```

3）配置文件修改

将 HBase 的配置文件 hbase-site.xml 放到 phoenix-4.6.0-bin/bin/目录下，替换 Phoenix 原来的配置文件。

4）权限修改

```
#切换到 phoenix-4.6.0-HBase-0.98/bin/ 下
cd phoenix-4.6.0-HBase-0.98/bin/
#修改 psql.py 和 sqlline.py 的权限为 777
chmod 777 psql.py
chmod 777 sqlline.py
```

5）登录 phoenix 客户端控制台进行操作

在 phoenix-4.6.0-bin/bin/下输入命令：

```
./sqlline.py localhost
```

启动客户端控制台。

2. Phoenix SQL

1）创建表

```
create table test (id varchar primary key,name varchar,age integer );
```

HBase 是区分大小写的，Phoenix 默认把 SQL 语句中的小写转换成大写，再建表，如果不希望转换，需要将表名和字段名等使用引号。HBase 默认 Phoenix 表的主键对应到 ROW，column family 名为 0，也可以在建表的时候指定 column family，创建表后使用 HBase Shell 也可以看到此表。

2）插入数据

```
upsert into test(id,name,age) values('000001','liubei',43);
```

3）查询

```
select * from chongdianleme_kc;
select count(1) from chongdianleme_kc;
select cmtid,count(1) as num from chongdianleme_kc group by issale order by num desc;
```

和 Phoenix SQL 客户端类似的还有 Presto、Impala 和 Spark SQL 等，只是 Phoenix 是专门针对 HBase 的。

3.3.6 Hive 集成 HBase 查询数据

Hive 集成 HBase 查询数据，通过 Hive 建一个到 HBase 的映射表，然后写 Hive SQL 来对 HBase 的数据进行统计分析，并且这种方式可以方便地和其他的 Hive 表做关联查询，或者做更复杂的统计。

1. 安装部署

```
#首先编辑 $HIVE_HOME/conf/hive-site.xml,添加如下
<property>
<name>hive.zookeeper.quorum</name>
<value>datanode1,datanode2,datanode3,datanode4,datanode5</value>
</property>
#然后将 $HBASE_HOME/lib 下的如下 jar 包复制到 $HIVE_HOME/auxlib 目录下
hbase-client-0.98.1-hadoop2.jar
hbase-common-0.98.1-hadoop2.jar
hbase-hadoop-compat-0.98.1-hadoop2.jar
hbase-protocol-0.98.1-hadoop2.jar
hbase-server-0.98.1-hadoop2.jar
htrace-core-2.04.jar

#环境搭建好了就可以创建 Hive 表了
#如果 HBase 表字段存储的是 long 行的字节码则 Hive 表必须使用 bigint
#登录 Hive 客户端建议设置以下参数
set hbase.client.scanner.caching=3000;
set mapred.map.tasks.speculative.execution = false;
set mapred.reduce.tasks.speculative.execution = false;
```

2. 创建课程商品 Hive 表并映射到 HBase 表

```
#需要多一个 row_key 字段,指定 Hive 字段到 HBase 字段的映射,字段名字可以不同
create external table if not exists chongdianleme_kc(
row_key string,
kcname string,
price string,
issale string
)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = "
kcname:name,
saleinfo:price,
saleinfo:issale
")
TBLPROPERTIES("hbase.table.name" = "chongdianleme_kc");
```

登录 Hive 的客户端便可以查询 chongdianleme_kc 表的数据了。使用这种方式来查询 HBase 数据比较方便。

3.3.7 HBase 升级和数据迁移

HBase 在使用过程中由于版本更新有时需要升级，升级之前 HBase 已经有数据了，这时候需要把之前的数据迁移到新版本上，下面给出一种数据迁移的方式，步骤如下：

1. 备份 HBase 表数据

```
#进入 hbase/bin 目录下，导出 HBase 数据到 Hadoop 的 HDFS
./hbase org.apache.hadoop.hbase.mapreduce.Driver export chongdianleme_kc hdfs://ai/hbase_backup/chongdianleme_kc
```

2. 备份 HBase 在 HDFS 上的目录

```
hadoop fs -mv /hbase /hbase_backup_old
```

3. 将 ZooKeeper 中的 HBase 数据删除

```
#登录/home/hadoop/software/zookeeper/bin/zkCli.sh -server localhost
ls /
rmr /hbase
```

4. 升级导入备份的 HDFS 数据

```
#注意导入前需建好 HBase 表
./hbase org.apache.hadoop.hbase.mapreduce.Driver import chongdianleme_kc hdfs://ai/hbase_backup/chongdianleme_kc
```

这种迁移方式的好处是可以保证不同版本的兼容性。

3.4 Sqoop 数据 ETL 工具实战

Sqoop 是一个数据处理的工具，用来从别的数据库导入数据到 Hadoop 平台，也可以从 Hadoop 导出到其他数据库平台，在搭建大数据平台数据仓库的时候，这是被经常使用的一个工具。

3.4.1 Sqoop 原理和功能介绍

Sqoop 是一个用来将 Hadoop 和关系型数据库中的数据相互转移的工具，可以将一个关系型数据库（例如：MySQL，Oracle，Postgres 等）中的数据导入 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导入关系型数据库中。

Sqoop 是一个数据处理工具客户端 jar 包，不需要启动单独的服务进程。在 Sqoop 迁移数据的时候会将 Sqoop 的脚本命令转换成 Hadoop 分布式计算引擎 MapReduce 程序，以分布式的方式并行导入导出数据。例如从 MySQL 导入 Hive，它可以把 MySQL 数据根据某个字段拆分成多份数据并行往 Hadoop 上写数据，性能比较高。主要特点是利用了 Hadoop 的分布式计算引擎原理。

因为它本身是一个客户端，所以不需要每台服务器都安装，在哪台服务器用就在哪台服务器上安装，安装非常简单，解压了就能用，步骤如下：

```
#上传 sqoop-1.*-cdh.**.tar.gz 到
/home/hadoop/software/
#解压后将名字改为和环境变量的目录名称一致，不配置环境变量而用绝对目录也可以
mv sqoop-1.*.-cdh* sqoop
#如果配置环境变量，直接输入 sqoop 命令
vim /etc/profile
#加入
export SQOOP_HOME=/home/hadoop/software/sqoop
#然后输入:wq 保存，让环境变量生效
source /etc/profile
#把 mysql-connector-Java-*.jar 复制到/home/hadoop/software/sqoop/lib 中
#如果导出导入用到了这个 jar 包，则会自动从这个目录扫描找到它
```

3.4.2 Sqoop 常用操作

Sqoop 最常用的操作就是从关系数据库 MySQL 导出数据到 Hadoop，再就是从 Hadoop 导出数据到 MySQL，输入 sqoop help 就可以看到它的命令参数：

```
Available commands:
  codegen                    //生成代码与数据库中的记录进行交互
  create-hive-table          //创建 hive 表
  eval                       //执行一个 SQL 语句并显示结果
  export                     //导出 rdbms 数据到 hdfs 上
  help                       //使用 sqoop 命令的帮助
  import                     //导入 rdbms 数据到 hdfs 上
  import-all-tables          //导入 rdbms 指定数据库所有的表数据到 hdfs 上
  job                        //sqoop 的作业，可创建作业、执行作业和删除作业
  list-databases             //通过 sqoop 的这个命令列出 jdbc 连接地址中所有的数据库
  list-tables                //通过 sqoop 的这个命令列出 jdbc 连接地址数据库中所有的表
  merge                      //合并增量数据
  metastore                  //运行 sqoop 的元存储
  version                    //查看 sqoop 的版本
```

我们用实例演示一下具体的操作命令：

```
#首先创建 MySQL 数据库和表结构
#在 MySQL 创建充电了么 utf8 格式数据库
```

```
CREATE DATABASE chongdianleme DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
#创建课程日志表
CREATE TABLE 'ods_kc_fact_clicklog' (
  'userid' varchar(36) NOT NULL,
  'kcid' varchar(100) NOT NULL,
  'time' varchar(100) NOT NULL
) ENGINE = InnoDB DEFAULT CHARSET = utf8;
#如果课程表存在,先删除
DROP TABLE IF EXISTS 'ods_kc_dim_product';
#创建课程表
CREATE TABLE 'ods_kc_dim_product' (
  'kcid' varchar(36) NOT NULL,
  'kcname' varchar(100) NOT NULL,
  'price' float DEFAULT '0',
  'issale' varchar(1) NOT NULL,
  PRIMARY KEY ('kcid')
) ENGINE = InnoDB DEFAULT CHARSET = utf8;
```

下面分别演示导出 Hadoop 数据到 MySQL 和从 MySQL 导入 Hadoop。

1. 导出 Hadoop 数据到 MySQL

1）在不带主键的情况下,增量导入课程日志数据到 MySQL,相当于追加数据

```
#脚本命令如下:
sqoop export --connect "jdbc:mysql://106.12.200.196:3306/chongdianleme?useUnicode=
true&characterEncoding=utf8&allowMultiQueries=true" --username root --password
chongdianleme888 -m 8 --table ods_kc_fact_clicklog --export-dir /ods/kc/fact/ods_kc_
fact_clicklog/ --input-fields-terminated-by '\001';
# --table 是要导入 MySQL 的表名
# --export-dir 是要从哪个 HDFS 目录导出
# --input-fields-terminated-by HDFS 目录数据的列分隔符
# -m 指定跑几个 map,这个不需要 reduce,只用 map 就行

#最后看一下 MySQL 是不是把数据导出成功了
select * from ods_kc_fact_clicklog;
```

2）在有主键的情况下,用 update+insert 方式导出数据

上面的追加数据因为没有主键,所以追加数据不会报错。如果有主键,当主键重复了肯定会报错,所以在这种情况下应该是已经存在这个主键就更新这条记录,不存在就插入。

```
sqoop export --connect "jdbc:mysql://106.12.200.196:3306/chongdianleme?useUnicode=
true&characterEncoding=utf8&allowMultiQueries=true" --username root --password
chongdianleme888 -m 8 --table ods_kc_dim_product --export-dir /ods/kc/dim/ods_kc_dim_
product/ --input-fields-terminated-by '\001' --update-key kcid --update-mode
allowinsert;

#解决中文乱码问题,在数据库名字后面加上
```

```
?useUnicode = true&characterEncoding = utf8&allowMultiQueries = true"

# -- table 是要导入 MySQL 的表名
# -- export - dir 是要从哪个 HDFS 目录导出
# -- input - fields - terminated - by HDFS 目录数据的列分隔符
# - m 指定跑几个 map,这个不需要 reduce,只用 map 就行
# -- update - key 指定更新 mysql 表的主键
# -- update - mode allowinsert 有新的数据是否允许插入,默认不插入,只更新

#看一下 MySQL 数据
select * from ods_kc_dim_product;
```

2. 从 MySQL 导入数据到 Hadoop

```
#首先创建 Hive 表
create EXTERNAL table IF NOT EXISTS ods_kc_dim_product_import(kcid string,kcname string,price
float ,issale string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\001'
stored as textfile
location '/ods/kc/dim/ods_kc_dim_product_import/';
```

Hive 的数据还是存储在 HDFS 上,我们可以将数据直接导入 Hive 存储的指定目录下,也可以用指定 Hive 表名的方式导入。

1）全量导入

```
#导入前,先把 HDFS 上的数据删除,然后按如下脚本导入
sqoop import -- connect " jdbc: mysql://106. 12. 200. 196: 3306/chongdianleme? useUnicode =
true&characterEncoding = utf8&allowMultiQueries = true" -- username root -- password
'chongdianleme888' -- query 'SELECT kcid,kcname,price,issale FROM ods_kc_dim_product where
price > 1000 and $ CONDITIONS' -- split - by kcid - m 8 -- target - dir /ods/kc/dim/ods_kc_dim_
product_import/ -- delete - target - dir -- fields - terminated - by '\001';

# -- query 可以是任意 SQL 语句,可关联多个表,但列和 HDFS 要对应上. $ CONDITIONS 是固定语法,
必须有
# -- split - by 跑分布式多个 map 的时候,根据 MySQL 表的哪个字段来拆分多块数据
# - m 跑几个 map
# -- target - dir 存到 HDFS 的那个目录下
# -- delete - target - dir 导入前删除 HDFS 上之前的数据
-- fields HDFS 或 Hive 表的字段分隔符
#最后看一下导入的数据
select * from ods_kc_dim_product_import;
```

2）增量导入

```
#指定 append 参数来追加数据
sqoop import -- connect " jdbc: mysql://106. 12. 200. 196: 3306/chongdianleme? useUnicode =
```

```
true&characterEncoding = utf8&allowMultiQueries = true" -- username root -- password
'chongdianleme888' -- query 'SELECT kcid,kcname,price,issale FROM ods_kc_dim_product where
price <= 1000 and $CONDITIONS' -- split-by kcid -m 8 -- target-dir /ods/kc/dim/ods_kc_dim
_product_import/ -- append -- fields-terminated-by '\001';
# -- query 可以是任意 SQL 语句,可关联多个表,但列和 HDFS 要对应上. $CONDITIONS 是固定语
#法,必须有
# -- split-by 跑分布式多个 map 的时候,根据 MySQL 表的哪个字段来拆分多块数据
# -m 跑几个 map
# -- target-dir 存到 HDFS 的那个目录下
# -- append 追加方式
# -- fields HDFS 或 Hive 表的字段分隔符
#看一下导入的 Hive 数据表
select * from ods_kc_dim_product_import;
```

以上我们列举了 MySQL 和 Hadoop 之间的导入导出常用命令,基本覆盖了常用的使用场景。对于一些复杂的数据处理任务,脚本满足不了的,一般是写程序自定义开发大数据平台做数据处理,Spark 是常用的框架,当然 Spark 不仅仅可以做数据处理,还有很多强大的功能,例如 Spark Streaming 的实时流处理应用、Spark SQL 的即时查询、MLlib 的机器学习和 GraphX 的图计算等,Spark 是一个完整的生态,下面我们讲解一下 Spark,同时也为我们后面章节讲解 Spark 分布式机器学习打基础。

3.5 Spark 基础

Spark 是用于大规模数据处理的统一分析引擎,一个可以实现快速通用的集群计算平台。它是由加州大学伯克利分校 AMP 实验室开发的通用内存并行计算框架,用来构建大型的、低延迟的数据分析应用程序。它扩展了广泛使用的 MapReduce 计算模型。高效地支撑更多计算模式,包括交互式查询和流处理。Spark 的一个主要特点是能够在内存中进行计算,及时依赖磁盘进行复杂的运算,Spark 依然比 MapReduce 更加高效。Spark 同时也是一个分布式机器学习平台。

3.5.1 Spark 原理和介绍

Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎。Spark 拥有 Hadoop MapReduce 所具有的优点,但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中,从而不再需要读写 HDFS,因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 算法。

Spark 是一种与 Hadoop 相似的开源集群计算环境,但是两者之间还存在一些不同之处,这些不同之处使 Spark 在某些工作负载方面表现得更加优越,换句话说,Spark 启用了内存分布数据集,除了能够提供交互式查询外,它还可以优化迭代工作负载。Spark 是用 Scala 语言实现的,它将 Scala 用作其应用程序框架。与 Hadoop 不同,Spark 和 Scala 能够

紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。尽管创建 Spark 是为了支持分布式数据集上的迭代作业，但实际上它是对 Hadoop 的补充，可以在 Hadoop 文件系统中并行运行，通过名为 Mesos 的第三方集群框架可以支持此行为。Spark 可用来构建大型的、低延迟的数据分析应用程序。

可以简单总结这么几点，Spark 是一个分布式内存计算框架；Spark 是一个计算引擎但没有存储功能；Spark 可以单机和分布式运行，有三种方式：Standalone 单独集群部署、Spark on Yarn 部署和 Local 本地模式。

Spark 平台是继 Hadoop 平台之后推出的分布式计算引擎，它刚出现的时候更多地是为了解决 Hadoop 的 MapReduce 计算问题，因为 Hadoop MapReduce 计算引擎是基于磁盘，而 Spark 基于内存，所以计算效率得到大大提升，下面我们从几个方面来对比 Spark 和 Hadoop。

1. Spark 和 Hadoop 框架比较

Spark 是分布式内存计算平台，它是用 Scala 语言编写，基于内存的快速、通用、可扩展的大数据分析引擎。Hadoop 是分布式管理、存储、计算的生态系统，包括 HDFS(存储)、MapReduce(计算)和 Yarn(资源调度)。

2. Spark 和 Hadoop 原理方面的比较

1) 编程模型比较

Hadoop 和 Spark 都是并行计算，两者都可以用 MR 模型进行计算，但 Spark 不仅有 MR，还有更多算子，并且 API 更丰富。

2) 作业

Hadoop 的一个作业称为一个 Job，每个 Job 里面分为 Map Task 和 Reduce Task 阶段，每个 Task 都在自己的进程中运行，当 Task 结束时，进程也会随之结束，当然 Hadoop 也可以只有 Map，而没有 Reduce。Spark 有对应的 Map 和 Reduce，但 Spark 的 ReduceByKey 和 Hadoop 的 Reduce 含义不一样，与 Hadoop 的 Reduce 比较相似的 Spark 函数是 GroupByKey。

3) 任务提交

Spark 用户提交的任务称为 Application，一个 Application 对应一个 SparkContext，Application 中存在多个 Job，每触发一次 Action 操作就会产生一个 Job。这些 Job 可以并行或串行执行，每个 Job 中有多个 Stage，Stage 是 Shuffle 过程中 DAGScheduler 通过 RDD 之间的依赖关系划分 Job 而来的，每个 Stage 里面有多个 Task，组成 TaskSet，由 TaskScheduler 分发到各个 Executor 中执行，Executor 的生命周期是和 Application 一样的，即使没有 Job 运行也是存在的，所以 Task 可以快速启动并读取内存以便进行计算。

3. Spark 和 Hadoop 详细比较

1) 执行效率

Spark 对标于 Hadoop 中的计算模块 MR，但是速度和效率比 MR 要快得多。Spark 是由于 Hadoop 中 MR 效率低下而产生的高效率快速计算引擎，批处理速度比 MR 快近 10

倍，内存中的数据分析速度比 Hadoop 快近 100 倍（源自官网描述）；实际应用中快不了这么多，一般快两三倍的样子，而官网描述的 100 倍是特殊场景。

2）文件管理系统

Spark 没有提供文件管理系统，所以它必须和其他的分布式文件系统进行集成才能运作。Spark 只是一个计算分析框架，专门用来对分布式存储的数据进行计算处理，它本身并不能存储数据。

3）Spark 操作用 Hadoop 的 HDFS

Spark 可以使用 Hadoop 的 HDFS 或者其他云数据平台进行数据存储，但是一般使用 HDFS。

4）数据操作

Spark 可以使用基于 HDFS 的 HBase 数据库，也可以使用 HDFS 的数据文件，还可以通过 jdbc 连接使用 MySQL 数据库数据。Spark 可以对数据库数据进行修改和删除，而 HDFS 只能对数据进行追加和全表删除。

5）设计模式

Spark 处理数据的设计模式与 MR 不一样，Hadoop 是从 HDFS 读取数据，通过 MR 将中间结果写入 HDFS，然后再重新从 HDFS 读取数据进行 MR，再刷写到 HDFS，这个过程涉及多次落盘操作，多次磁盘 IO 操作，效率并不高，而 Spark 的设计模式是读取集群中的数据后，在内存中存储和运算，直到全部数据运算完毕后，再存储到集群中。

6）磁盘和分布式内存

Spark 中 RDD 一般存放在内存中，如果内存不够存放数据，会同时使用磁盘存储数据。通过 RDD 之间的血缘连接、数据存入内存后切断血缘关系等机制，Spark 可以实现灾难恢复，当数据丢失时可以恢复数据，这一点与 Hadoop 类似，Hadoop 基于磁盘读写，天生数据具备可恢复性。

4. Spark 的优势

1）RDD 分布式弹性数据集

Spark 基于 RDD，数据并不存放在 RDD 中，只是通过 RDD 进行转换，通过装饰者设计模式，数据之间形成血缘关系和类型转换。

2）编程语言优势

Spark 用 Scala 语言编写，相比用 Java 语言编写的 Hadoop 程序更加简洁。

3）提供的算子更丰富

相比 Hadoop 中对于数据计算只提供了 Map 和 Reduce 两个操作，Spark 提供了丰富的算子，它可以通过 RDD 转换算子和 RDD 行动算子，实现很多复杂算法操作，这些复杂的算法在 Hadoop 中需要自己编写，而在 Spark 中通过 Scala 语言封装好后，直接用就可以了。

4）RDD 的多个算子转换，快速迭代式内存计算优势

Hadoop 中对于数据的计算，一个 Job 只有一个 Map 和 Reduce 阶段，对于复杂的计算，需要使用多次 MR，这样带来大量的磁盘 I/O 开销，效率不高，而在 Spark 中，一个 Job 可以

包含多个 RDD 的转换算子，在调度时可以生成多个 Stage，实现更复杂的功能。

5）中间结果集在内存，计算更快

Hadoop 的中间结果存放在 HDFS 中，每次 MR 都需要刷写和调用，而 Spark 中间结果优先存放在内存中，当内存不够用再存放在磁盘中，不存入 HDFS，避免了大量的 IO 和刷写及读取操作。

6）对于迭代式流式数据的处理能力比较强

Hadoop 适合处理静态数据，而对于迭代式流式数据的处理能力差，Spark 通过在内存中缓存处理数据的方式提高了处理流式数据和迭代式数据的能力，于是就有了 Spark Streaming 流式计算，类似于 Storm 和 Fink。

5. Spark 基本概念

1）RDD

RDD 是弹性分布式数据集（Resilient Distributed Dataset）的简称，它是分布式内存的一个抽象概念并提供了一种高度受限的共享内存模型。

2）DAG

DAG 是有向无环图（Directed Acyclic Graph）的简称，反映与 RDD 之间的依赖关系。

3）Driver Program

Driver Program 是控制程序，负责为 Application 构建 DAG 图。

4）Cluster Manager

Cluster Manager 是集群资源管理中心，负责分配计算资源。

5）Worker Node

Worker Node 是工作节点，负责完成具体计算。

6）Executor

Executor 是运行在工作节点上的一个进程，负责运行 Task，并为应用程序存储数据。

7）Application

Application 是用户编写的 Spark 应用程序，一个 Application 包含多个 Job。

8）Job

作业，一个 Job 包含多个 RDD 及作用于相应 RDD 上的各种操作。

9）Stage

阶段，是作业的基本调度单位，一个作业会分为多组任务，每组任务被称为“阶段”。

10）Task

任务，运行在 Executor 上的工作单元，是 Executor 中的一个线程。

总结：Application 由多个 Job 组成，Job 由多个 Stage 组成，Stage 由多个 Task 组成。Stage 是作业调度的基本单位。

6. Spark 运行流程

1）Application 首先被 Driver 构建 DAG 图并分解成 Stage；

2）Driver 向 Cluster Manager 申请资源；

3）Cluster Manager 向某些 Work Node 发送征召信号；

4）被征召的 Work Node 启动 Executor 进程响应征召，并向 Driver 申请任务；

5）Driver 分配 Task 给 Work Node；

6）Executor 以 Stage 为单位执行 Task，期间 Driver 进行监控；

7）Driver 收到 Executor 任务完成的信号后向 Cluster Manager 发送注销信号；

8）Cluster Manager 向 Work Node 发送释放资源信号；

9）Work Node 对应 Executor 停止运行。

7. RDD 数据结构

RDD 是记录只读分区的集合，是 Spark 的基本数据结构。RDD 代表一个不可变、可分区和里面的元素可并行计算的集合。一般有两种方式可以创建 RDD，第一种是读取文件中的数据生成 RDD，第二种则是通过将内存中的对象并行化得到 RDD，如代码 3.2 所示。

【代码 3.2】 Spark 创建 RDD

```
//通过读取文件生成 RDD,可以是文件也可以是目录,如果是目录则会自动加载目录下所有文件
val rdd = sc.textFile("hdfs://chongdianleme/ods/dim/data")
//通过将内存中的对象并行化得到 RDD
val numArray = Array(1,2,3,4,5)
val rdd = sc.parallelize(numArray)
//或者 val rdd = sc.makeRDD(numArray)
```

创建 RDD 之后，可以使用各种操作对 RDD 进行编程。对 RDD 的操作有两种类型，即 Transformation 操作和 Action 操作。转换操作是从已经存在的 RDD 创建一个新的 RDD，而行动操作是在 RDD 上进行计算后返回结果到 Driver。Transformation 操作都具有 Lazy 特性，即 Spark 不会立刻进行实际的计算，只会记录执行的轨迹，只有在触发 Action 操作的时候它才会根据 DAG 图真正执行。操作确定了 RDD 之间的依赖关系。RDD 之间的依赖关系有两种类型，即窄依赖和宽依赖。窄依赖时，父 RDD 的分区和子 RDD 的分区关系是一对一或者多对一的关系；宽依赖时，父 RDD 的分区和子 RDD 的分区关系是一对多或者多对多的关系。与宽依赖关系相关的操作一般具有 Shuffle 过程，即通过一个 Partitioner 函数将父 RDD 中每个分区上 Key 的不同记录分发到不同的子 RDD 分区。依赖关系确定了 DAG 切分成 Stage 的方式。切割规则为从后往前，遇到宽依赖就切割 Stage。RDD 之间的依赖关系形成一个 DAG 有向无环图，DAG 会提交给 DAGScheduler，DAGScheduler 会把 DAG 划分成相互依赖的多个 Stage，划分 Stage 的依据就是 RDD 之间的宽窄依赖。遇到宽依赖就划分 Stage，每个 Stage 包含一个或多个 Task 任务，然后将这些 Task 以 TaskSet 的形式提交给 TaskScheduler 运行。

Spark 生态系统以 SparkCore 为核心，能够读取传统文件（如文本文件）、HDFS、AmazonS3、Alluxio 和 NoSQL 等数据源，利用 Standalone、Yarn 和 Mesos 等资源调度管理，完成应用程序分析与处理。这些应用程序来自 Spark 的不同组件，如 Spark Shell 或

Spark Submit 交互式批处理方式、Spark Streaming 实时流处理应用、Spark SQL 即时查询、采样近似查询引擎 BlinkDB 的权衡查询、MLbase/MLlib 机器学习、GraphX 图处理。

Spark 机器学习实现的算法非常多，接下来我们介绍 Spark 机器学习 MLlib，后面的章节会再详细地讲解。

3.5.2 Spark MLlib 机器学习介绍

Spark 机器学习是基于 SparkCore 框架之上的，所以多是分布式运行，在分布式机器学习领域 Spark 是一个主流的框架，应用非常普遍。并且实现的算法非常全面，从分类、聚类、回归、降维、最优化和神经网络等都有，而且 API 代码调用非常简单易用，对于加载训练数据集的格式也非常统一，例如分类的一份训练数据可以同时用在多个分类算法上，不用做额外的处理，这样大大节省了开发者的时间，方便开发者快速对比各个算法之间的效果。下面我们列举一下 Spark 实现了哪些算法，随着版本的更新，还在不断地加入新的算法。

1. 分类

SVM(支持向量机)

Naive Bayes(贝叶斯)

Decision tree(决策树)

Random Forest(随机森林)

Gradient-Boosted Decision Tree(GBDT)(梯度提升树)

2. 回归

Logistic regression(逻辑回归，也可以分类)

Linear regression(线性回归)

Isotonic regression(保序回归，和时间序列算法类似，可以做销量预测)

3. 推荐

Collaborative filtering(协同过滤)

Alternating Least Squares (ALS)(交替最小二乘法)

Frequent pattern mining(频繁项集挖掘)

FP-growth(频繁模式树)

Apriori(算法)

4. Clustering(聚类算法)

K-means(K 均值)

Gaussian mixture(高斯混合模型)

Power Iteration Clustering (PIC)(快速迭代聚类)

Latent Dirichlet Allocation (LDA)(潜在狄利克雷分配模型)

Streaming K-means(流 K 均值)

5. Dimensionality reduction(降维算法)

Singular Value Decomposition (SVD)(奇异值分解)

Principal Component Analysis (PCA)(主成分分析)

6. Feature extraction and transformation(特征提取转换)

TF-IDF(词频/反文档频率)

Word2Vec(词向量)

StandardScaler(标准归一化)

Normalizer(正规化)

Feature selection(特征选取)

ElementwiseProduct(元素智能乘积)

PCA(主成分分析)

7. Optimization (developer)(最优化算法)

Stochastic gradient descent(随机梯度下降)

Limited-memory BFGS (L-BFGS)(拟牛顿法)

8. 神经网络

MLP 智能感知机——前馈神经网络

3.5.3 Spark GraphX 图计算介绍

GraphX 是 Spark 的一个重要子项目,它利用 Spark 作为计算引擎,实现了大规模图计算功能,并提供了类似 Pregel 的编程接口。GraphX 的出现将 Spark 生态系统变得更加完善和丰富,同时以其与 Spark 生态系统其他组件很好的融合,以及强大的图数据处理能力,在工业界得到了广泛的应用。

GraphX 是常用图算法在 Spark 上的并行化实现,同时提供了丰富的 API 接口。图算法是很多复杂机器学习算法的基础,在单机环境下有很多应用案例。在大数据环境下,当图的规模大到一定程度后,单机就很难解决大规模的图计算,需要将算法并行化,在分布式集群上进行大规模图处理。目前,比较成熟的方案有 GraphX 和 GraphLab 等大规模图计算框架。现在可以和 GraphX 组合使用的分布式图数据库是 Neo4J。Neo4J 是一个高性能的、非关系的、具有完全事务特性的和鲁棒的图数据库。另一个数据库是 Titan,Titan 是一个分布式的图形数据库,特别为存储和处理大规模图形而优化。二者均可作为 GraphX 的持久化层,存储大规模图数据。

Graphx 的主要接口:

基本信息接口(numEdges,num Vertices,degrees(in/out))

聚合操作 (mapVertices,mapEdges,mapTriplets)

转换接口 (mapReduceTriplets,collectNeighbors)

结构操作（reverse，subgraph，mask，groupEdges）

缓存操作（cache，unpersistVertices）

GraphX 每个图由 3 个 RDD 组成，如表 3.2 所示。

表 3.2 GraphX 图

名称	对应 RDD	包含的属性
Vertices	VertexRDD	ID、点属性
Edges	EdgeRDD	源顶点的 ID，目标顶点的 ID，边属性
Triplets	EdgeTriplet	源顶点 ID，源顶点属性，边属性，目标顶点 ID，目标顶点属性

Triplets 其实是对 Vertices 和 Edges 做了 join 操作点分割、边分割。

GraphX 图计算算法经典应用有基于最大连通图的社区发现、基于三角形计数的关系衡量和基于随机游走的用户属性传播。

3.5.4 Spark Streaming 流式计算介绍

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafka、Flume、Twitter、ZeroMQ、Kinesis 及 TCPsockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统、数据库和现场仪表盘。在“OneStackrulethemall”的基础上，还可以使用 Spark 的其他子框架，如集群学习、图计算等对流数据进行处理。

Spark 的各个子框架都是基于核心 Spark 的，Spark Streaming 在内部的处理机制是接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过 SparkEngine 处理这些批数据，最终得到处理后的一批批结果数据。

对应的批数据，在 Spark 内核里对应一个 RDD 实例，因此，对应流数据的 DStream 可以看作一组 RDD，即 RDD 的一个序列。通俗点理解的话，在流数据被分成一批一批后，通过一个先进先出的队列，SparkEngine 从该队列中依次取出一个个批数据，把批数据封装成一个 RDD，然后进行处理，这是一个典型的生产者消费者模型，对应的是生产者消费者模型的问题，即如何协调生产速率和消费速率之间的关系。

3.5.5 Scala 编程入门和 Spark 编程[1]

Scala 是一门多范式的编程语言，一种类似 Java 的编程语言，设计初衷是为了实现可伸缩的语言，并集成面向对象编程和函数式编程的各种特性。Scala 编程语言抓住了很多开发者的眼球。如果你粗略浏览 Scala 的网站，会觉得 Scala 是一种纯粹的面向对象编程语言，而又无缝地结合了命令式编程和函数式编程风格。Scala 有几项关键特性表明了它的面向对象的本质。例如，Scala 中的每个值都是一个对象，包括基本数据类型（即布尔值、数字等）

在内，连函数也是对象。另外，类可以被子类化，而且 Scala 还提供了基于 mixin 的组合(mixin-based composition)。与仅支持单继承的语言相比，Scala 具有更广泛意义上的类重用。Scala 允许在定义新类的时候重用“一个类中新增的成员定义(即相较于其父类的差异之处)”。Scala 称之为 mixin 类组合。Scala 还包含了若干函数式语言的关键概念，包括高阶函数(Higher-Order Function)、局部套用(Currying)、嵌套函数(Nested Function)和序列解读(Sequence Comprehensions)等。

Scala 是静态类型的，这就允许它提供泛型类、内部类甚至多态方法(Polymorphic Method)。另外值得一提的是，Scala 被特意设计成能够与 Java 和. NET 进行互操作。Scala 当前版本还不能在. NET 上运行，但按照计划将来可以在. NET 上运行。Scala 可以与 Java 互操作。它用 scalac 这个编译器把源文件编译成 Java 的 class 文件(即在 JVM 上运行的字节码)。你可以从 Scala 中调用所有的 Java 类库，也同样可以从 Java 应用程序中调用 Scala 的代码。用 David Rupp 的话来说，它也可以访问现存的数之不尽的 Java 类库，这让(潜在地)Java 类库迁移到 Scala 更加容易，从而 Scala 得以使用为 Java1. 4、5. 0 或者 6. 0 编写的巨量的 Java 类库和框架，Scala 会经常性地针对这几个版本的 Java 类库进行测试。Scala 可能也可以在更早版本的 Java 上运行，但没有经过正式的测试。Scala 以 BSD 许可发布，并且数年前就已经被认为相当稳定了。

说了这么多，我们还没有回答一个问题，“为什么我要使用 Scala?”Scala 的设计始终贯穿着一个理念：

创造一种更好地支持组件的语言(*The Scala Programming Language*，Donna Malayeri)，也就是说软件应该由可重用的部件构造而成。Scala 旨在提供一种编程语言，它能够统一和一般化分别来自面向对象和函数式两种不同风格的关键概念。借着这个目标与设计，Scala 得以提供一些出众的特性，包括：

- 面向对象风格
- 函数式风格
- 更高层的并发模型

Scala 把 Erlang 风格的基于 actor 的并发带进了 JVM。开发者可以利用 Scala 的 actor 模型在 JVM 上设计具伸缩性的并发应用程序，它会自动获得多核心处理器带来的优势，而不必依照复杂的 Java 线程模型来编写程序。

- 轻量级的函数语法
  高阶
  嵌套
  局部套用(Currying)
  匿名
- 与 XML 集成
  可在 Scala 程序中直接书写 XML
  可将 XML 转换成 Scala 类

• 与 Java 无缝地互操作

Scala 的风格和特性已经吸引了大量的开发者，例如 Debasish Ghosh 就觉得：我已经把玩了 Scala 好一阵子，可以说我绝对享受这个语言的创新之处。总而言之，Scala 是一种函数式面向对象语言，它融汇了许多前所未有的特性，而同时又运行于 JVM 之上。随着开发者对 Scala 的兴趣日增，以及越来越多的工具支持，Scala 语言无疑将成为你手上一件必不可少的工具。

目前很多优秀的开源框架，如 Spark、Flink 都是基于 Scala 语言开发的，在大数据领域 Scala 语言越来越被普遍地使用。由于本书涵盖知识点较多，我们只对 Scala 的一些简单常用语法做介绍，更高级的功能可以参见专门的 Scala 编程书籍。

1. Scala 基础编程

1）Hello world 入门例子

Hello world 是每个编程语言的经典入门例子，Scala 也是一样。首先在文件里声明一个 object 类型的类，这里不能用 class，object 是能直接找到 main 函数入口的，而 class 则不行。和 Java 一样，Scala 是以 main 函数作为主入口。函数前面用 def 声明。main 函数里面的参数先写参数名，后面跟着一个冒号，冒号后面是参数类型。函数的返回值不是必须指定的，它会自己推断。在函数体里面直接打印出来 Hello，world!，执行语句后面不用加分号，这点与 Java 不同。在 Java 中要是不加分号就会报错，如代码 3.3 所示。

【代码 3.3】 Hello World

```
object HelloWorld {
 def main(args: Array[String])
{
    println("Hello, world!")
 }
}
```

2）定义变量 val 和 var，val 是不可变变量，而 var 是可变变量

声明 val 是不可变的变量，如果后面强行赋值就会报错。var 是可变的变量，后面可以重新赋一个新值，如代码 3.4 所示。

【代码 3.4】 定义变量

```
scala > val msg = "Hello,World"
msg: String = Hello,World

scala > val msg2:String = "Hello again,world"
msg2: String = Hello again,world
#定义 var
var i = 0
#可以
i = i + 1
i += 1
```

```
#但是不能 i++
```

3）定义函数

函数前面用 def 声明，函数里面的参数先写参数名，后面跟着一个冒号，冒号后面是参数类型。函数的返回值不是必须指定的，它会自己推断，但也可以自己指定，例如代码 3.5 中指定返回值为 Int 整数类型，函数后面加个冒号，后面跟着返回值类型接口。有一点需要说明，如果指定了函数返回值就必须有返回值。如果不指定就比较灵活，可以有返回值，也可以没有，程序自己推断，如代码 3.5 所示。

【代码 3.5】 定义函数

```
def max(x:Int,y:Int) : Int =
{
    if (x > y) x
    else y
}
```

4）定义类

类是面向对象的，和 Java 的类差不多。类里面可以声明属性和函数，如代码 3.6 所示。

【代码 3.6】 定义类

```
class ChecksumAccumulator{
    private var sum = 0
    def add(b:Byte) :Unit = sum += b
    def checksum() : Int = ~ (sum & 0xFF) + 1
}
```

可以看到 Scala 类定义和 Java 类定义非常类似，也是以 class 开始，和 Java 不同的是 Scala 的默认修饰符为 public，也就是如果不带有访问范围的修饰符 public、protected 和 private，Scala 默认定义为 public。Scala 代码无须使用“；”结尾，也不需要使用 return 返回值，函数的最后一行的值就作为函数的返回值。

5）基本类型

Scala 与 Java 有着相同的数据类型，和 Java 的数据类型的内存布局完全一致，精度也完全一致。其中比较特殊的类型有 Unit，表示没有返回值；Nothing 表示没有值，是所有类型的子类型，创建一个类就一定有一个子类是 Nothing；Any 是所有类型的超类；AnyRef 是所有引用类型的超类；注意最大的类是 Object。

上面列出的数据类型都是对象，也就是说 Scala 没有 Java 中的原生类型。Scala 是可以对数字等基础类型调用方法的。例如数字 1 可以调方法，使用 1.方法名。

如上所示，可见到所有类型的基类与 Any。Any 之后分为两个类型 AnyVal 与 AnyRef。其中 AnyVal 是所有数值类型的父类型，AnyRef 是所有引用类型的父类型。

与其他语言稍微有点不同的是，Scala 还定义了底类型。其中 Null 类型是所有引用类型的底类型，及所有 AnyRef 的类型的空值都是 Null，而 Nothing 是所有类型的底类型，对

应 Any 类型。Null 与 Nothing 都表示空。

在基础类型中只有 String 是继承自 AnyRef 的，与 Java 相同，Scala 中的 String 也是内存不可变对象，这就意味着，所有的字符串操作都会产生新的字符串。其他的基础类型如 Int 等都是 Scala 包装的类型，例如 Int 类型对应的是 Scala.Int，它只是 Scala 包会被每个源文件自动引用。

标准类库中的 Option 类型用样例类来表示可能存在、也可能不存在的值。样例子类 Some 包装了某个值，例如 Some("Fred")；而样例对象 None 表示没有值，这比使用空字符串的意图更加清晰，比使用 Null 来表示缺少某值的做法更加安全(避免了空指针异常)。

下面列出了 Scala 支持的数据类型：

Byte：8 位有符号补码整数，数值区间为－128～127

Short：16 位有符号补码整数，数值区间为－32768～32767

Int：32 位有符号补码整数，数值区间为－2147483648～2147483647

Long：64 位有符号补码整数，数值区间为－9223372036854775808～9223372036854775807

Float：32 位，IEEE 754 标准的单精度浮点数

Double：64 位 IEEE 754 标准的双精度浮点数

Char：16 位无符号 Unicode 字符，区间值为 U＋0000～U＋FFFF

String：字符序列

Boolean：true 或 false

Unit：表示无值，和其他语言中 void 等同。用作不返回任何结果的方法的结果类型。Unit 只有一个实例值，写成()

Null：null 或空引用

Nothing：在 Scala 的类层级的最底端，它是任何其他类型的子类型

Any：所有其他类的超类

AnyRef：Scala 里所有引用类(referenceclass)的基类

上面列出的数据类型都是对象，也就是说 Scala 没有 Java 中的原生类型。Scala 是可以对数字等基础类型调用方法的。

6) If 表达式

If 是如果，else 是否则，两个分支，比较好理解，如代码 3.7 所示。

【代码 3.7】 If 表达式

```
var filename = "default.txt"
if(!args.isEmpty)
  filename = args(0)
else "default.txt"
```

Scala 语言的 if 的基本功能和其他语言没有什么不同，它根据条件执行两个不同的分支。

7）While 循环

While 指定一个条件来循环，当括号内的条件为真的时候退出循环。为 true 时叫作真，为 false 时叫作假，如代码 3.8 所示。

【代码 3.8】 While 循环

```
def gcdLoop (x: Long, y:Long) : Long = {
   var a = x
   var b = y
   while( a!= 0) {
        var temp = a
        a = b % a
        b = temp
   }
   b
}
```

Scala 的 while 循环和其他语言如 Java 功能一样，它含有一个条件和一个循环体，但是没有 break 和 continue。

8）For 循环

For 循环有如下 3 种方式，如代码 3.9 所示。

【代码 3.9】 For 循环

```
//第一种方式
for (arg <- args)
println(arg)
//第二种方式
args.foreach(println)
//第三种方式
for (i <- 0 to 2)
print(greetStrings(i))
```

9）Try catch finally 异常处理

异常处理机制，如代码 3.10 所示。

【代码 3.10】 Try catch finally 异常处理

```
import Java.io.FileReader
import Java.io.FileNotFoundException
import Java.io.IOException
try {
 val f = new FileReader("input.txt")
 // Use and close file
} catch {
 case ex: FileNotFoundException =>          // Handle missing file
 case ex: IOException =>                    // Handle other I/O error
}
```

```
finally
{
   f.close()
}
```

执行步骤是先执行 try 方法体里的代码，如果有异常就会执行 catch 里面的代码，最后才会执行 finally 里的代码。

10）Match 表达式，类似 Java 的 switch…case 语句

if else 分值只有如果、否则两个分值，如果有多个分支的时候使用 match case 表达式，很好理解，如代码 3.11 所示。

【代码 3.11】 Match 表达式

```
var myVar = "theValue";
var myResult =
    myVar match {
       case "someValue" => myVar + " A";
       case "thisValue" => myVar + " B";
       case "theValue" => myVar + " C";
       case "doubleValue" => myVar + " D";
    }
println(myResult);
```

上面对 Scala 编程做了简单的介绍，Scala 是一个基础的语言，Spark 编程在 Scala 语言基础上封装了很多现成的函数，供开发者使用，减小开发的工作量，并且使代码更加简洁。Spark 的优势之一就在于提供了大量常用的 API 函数，满足了很多应用场景，从而大大提高了开发效率。下面我们介绍 Spark 编程常用的 API 函数。

2. Spark 广播变量和累加器

通常情况下，当向 Spark 操作（如 map，reduce）传递一个函数时，它会在一个远程集群节点上执行，并会使用函数中所有变量的副本。这些变量被复制到所有的机器上，远程机器上没有被更新的变量会向驱动程序回传。在任务之间使用通用的、支持读写的共享变量是低效的。尽管如此，Spark 提供了两种有限类型的共享变量：广播变量和累加器。

1）广播变量

广播变量允许程序员将一个只读的变量缓存到每台机器上，而不用在任务之间传递变量。广播变量可被用于有效地给每个节点一个大输入数据集的副本。Spark 还尝试使用高效的广播算法来分发变量，进而减少通信的开销。Spark 的动作通过一系列的步骤执行，这些步骤由分布式的 shuffle 操作分开。Spark 自动地广播每个步骤的每个任务所需要的通用数据。这些广播数据被序列化地缓存，并在运行任务之前被反序列化出来。这意味着当我们需要在多个阶段的任务之间使用相同的数据时，或者在以反序列化形式缓存数据是十分重要的时候，显式地创建广播变量才有用。

它在所有节点的内存里缓存一个值，和 Hadoop 里面的分布式缓存 DistributeCache 类

似，如代码 3.12 所示。

【代码 3.12】 广播变量

```
val arr1 = (0 until 1000000).toArray
for (i <- 0 until 3) {
  val startTime = System.nanoTime
  val barr1 = sc.broadcast(arr1)
  val observedSizes = sc.parallelize(1 to 10, slices).map(_ => barr1.value.size)
  observedSizes.collect().foreach(i => println(i))
}
#都打印
1000000
```

2）累加器

累加器是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它可以被用来实现计数器和求总和的功能。Spark 原生地支持数字类型的累加器，编程者可以添加新支持的类型。如果在创建累加器时指定了名字，就可以在 Spark 的 UI 界面看到它。这有利于理解每个执行阶段的进程（对于 Python 还不支持）。

累加器通过对一个初始化了的变量 v 调用 SparkContext.accumulator(v)来创建。在集群上运行的任务可以通过 add 或者“+=”方法在累加器上进行累加操作。但是，它们不能读取它的值。只有驱动程序能够通过累加器的 value 方法读取它的值。

它们只能被“加”起来，就像计数器或者是“求和”。和 Hadoop 的 getCounter 类 hadoop: context.getCounter(Counters.USERS).increment(1)；功能相似，如代码 3.13 所示。

【代码 3.13】 累加器

```
Spark: val accum = sc.accumulator[Int](0,"accumJobCountInvalid")
accum += 1
```

3. Spark 转换操作

transformation 的意思是得到一个新的 RDD，方式很多，例如从数据源生成一个新的 RDD，从 RDD 生成一个新的 RDD。

1）map(func)

对调用 map 的 RDD 数据集中的每个 element 都使用 func，然后返回一个新的 RDD，这个返回的数据集是分布式的数据集。

2）filter(func)

对调用 filter 的 RDD 数据集中的每个元素都使用 func，然后返回一个包含使 func 为 true 的元素构成的 RDD。

3）flatMap(func)

和 map 差不多，但是 flatMap 生成的是扁平化结果。

4) mapPartitions(func)

和 map 很像,但是 map 是针对每个 element,而 mapPartitions 是针对每个 partition。

5) mapPartitionsWithSplit(func)

和 mapPartitions 很像,但是 func 作用在其中一个 split 上,所以 func 中应该有 index。

6) sample(withReplacement,faction,seed)

对数据进行抽样。

7) union(otherDataset)

返回一个新的 dataset,包含源 dataset 和给定 dataset 的元素的集合。

8) distinct([numTasks])

返回一个新的 dataset,这个 dataset 含有的是源 dataset 中的 distinct 的 element。

9) groupByKey(numTasks)

返回(K,Seq[V]),也就是 Hadoop 中 reduce 函数接收的 key-valuelist。

10) reduceByKey(func,[numTasks])

就是用一个给定的 reducefunc 再作用在 groupByKey 产生的(K,Seq[V]),例如求和、求平均数。

11) sortByKey([ascending],[numTasks])

按照 key 来进行排序,是升序还是降序,ascending 是 boolean 类型。

12) join(otherDataset,[numTasks])

当有两个 KV 的 dataset(K,V)和(K,W),返回的是(K,(V,W))的 dataset,numTasks 为并发的任务数。

13) cogroup(otherDataset,[numTasks])

当有两个 KV 的 dataset(K,V)和(K,W)时,返回的是(K,Seq[V],Seq[W])的 dataset,numTasks 为并发的任务数。

14) cartesian(otherDataset)

笛卡儿积就是 m×n,自然连接。

4. Spark Action 操作

action 意思是得到一个值,或者一个结果(直接将 RDDcache 保存到内存中),所有的 transformation 都采用懒策略,就是说如果只是将 transformation 提交是不会执行计算的,计算只有在 action 被提交的时候才被触发。

1) reduce(func)

聚集,但是传入的函数是两个输入参数并返回一个值,这个函数必须是满足交换律和结合律的。

2) collect()

一般在使用 filter 或者结果足够小的时候用 collect 封装并返回一个数组。

3) count()

返回的是 dataset 中 element 的个数。

4) first()

返回的是 dataset 中的第一个元素。

5) take(n)

返回前 n 个 elements,这个是 driverprogram 返回的。

6) takeSample(withReplacement,num,seed)

抽样返回一个 dataset 中的 num 个元素,随机种子 seed。

7) saveAsTextFile(path)

把 dataset 写到一个 textfile 中,或者 HDFS 中,或者 HDFS 支持的文件系统中,Spark 把每条记录都转换为一行记录,然后写到 file 中。

8) saveAsSequenceFile(path)

只能用在 key-value 对上,然后生成 SequenceFile 写到本地或者 Hadoop 文件系统中。

9) countByKey()

返回的是 key 对应的个数的一个 map,作用于一个 RDD。

10) foreach(func)

对 dataset 中的每个元素都使用 func。

5. Spark 经典 WordCount 例子

对于 Spark 来讲,单词计数是非常简单的例子,虽然看起来简单,但真正理解起来并不容易,如代码 3.14 所示。

【代码 3.14】 WordCount 例子

```
sc.textFile("/input").flatMap(_.split(" ").map((_, 1)).reduceByKey(_ + _).saveAsTextFile
("/output")
```

计算过程的逻辑是这样的,首先通过 textFile 方法从指定的文章目录 input 加载数据,不管 input 下有多少个文件,它都会一行一行地读进来,并且这个 input 目录可以有多个文件。加载进来后,数据就分布到内存 RDD 里面了,之后 flatMap 会遍历 RDD 里面的每一行记录并进行处理,因为单词是以空格分割的,我们用 split 函数拆分成多个单词使其成为一个单词数组,_.split 前面的下画线指的是 RDD 里面的每一个元素,因为这里的 RDD 是一行行的记录,所以一个元素就是一行 String 的记录字符串,然后通过 flatMap 会打平这个数据,通过后面的.map 变量的就是一个个的单词,不再是一行记录。map 方法通过一个二元组(_, 1)返回单词和计数 1,返回形成一个新的 RDD,之后通过 reduceByKey 把每个单词的计数相加求和,就得到了每个单词的计数了,最后通过 saveAsTextFile 输出到一个文件。

我们讲了 Scala 和 Spark 的编程基础后,下面通过一个项目案例来整体地看一下从编程到分布式部署的完整过程,因为 Spark 分布式机器学习打包和部署与这个过程是一样的,这也是为我们后面讲 Spark 分布式机器学习打基础。

3.5.6 Spark 项目案例实战和分布式部署

前面讲到 HBase 可以通过 JavaAPI 的方式操作 HBase 数据库，由于 Java 和 Scala 可以互相调用，本节使用 Scala 语言通过 Spark 平台来实现分布式操作 HBase 数据库，打包并部署到 Spark 集群上面。这样我们对 Spark＋Scala 项目开发就会有一个完整的认识和实际工作场景的体会。

我们首先创建一个 Spark 工程，然后创建一个 HbaseJob 的 object 类文件，项目的功能是从 HBase 批量读取课程商品表数据，然后存储到 Hadoop 的 HDFS 上，如代码 3.15 所示。

【代码 3.15】 HbaseJob.scala

```
package com.chongdianleme.mail
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.{Result, Get, HConnectionManager}
import org.apache.hadoop.hbase.util.{ArrayUtils, Bytes}
import org.apache.spark._
import scopt.OptionParser
import scala.collection.mutable.ListBuffer
/**
 * Created by 充电了么 App - 陈敬雷
   * Spark 分布式操作 HBase 实战
   * 网站:www.chongdianleme.com
   * 充电了么 App——专业上班族职业技能提升的在线教育平台
 */
object HbaseJob {
case class Params(
//输入目录的数据就是课程 ID,每行记录只有一个课程 ID,后面根据课程 ID 作为 rowKey 从
//Hbase 里查询数据
inputPath: String = "file:///D:\\chongdianleme\\Hbase 项目\\input",
                    outputPath: String = "file:///D:\\chongdianleme\\Hbase 项目\\output",
                    table: String = "chongdianleme_kc",
                    minPartitions: Int = 1,
                    mode: String = "local"
)

def main(args: Array[String]) {
val defaultParams = Params()
val parser = new OptionParser[Params]("HbaseJob") {
      head("HbaseJob: 解析参数.")
      opt[String]("inputPath")
        .text(s"inputPath 输入目录, default: ${defaultParams.inputPath}}")
        .action((x, c) => c.copy(inputPath = x))
      opt[String]("outputPath")
        .text(s"outputPath 输出目录, default: ${defaultParams.outputPath}")
```

```
        .action((x, c) => c.copy(outputPath = x))
      opt[Int]("minPartitions")
        .text(s"minPartitions , default: ${defaultParams.minPartitions}")
        .action((x, c) => c.copy(minPartitions = x))
      opt[String]("table")
        .text(s"table table, default: ${defaultParams.table}")
        .action((x, c) => c.copy(table = x))
      opt[String]("mode")
        .text(s"mode 运行模式, default: ${defaultParams.mode}")
        .action((x, c) => c.copy(mode = x))
      note("""|For example, the following command runs this app on a HbaseJob dataset:
       """.stripMargin)
  }
   parser.parse(args, defaultParams).map { params => {
println("参数值:" + params)
readFilePath(params.inputPath,params.outputPath,params.table, params.minPartitions, params.
mode)
  }
  }getOrElse {
    System.exit(1)
  }
println("充电了么App——Spark分布式批量操作HBase实战 -- 计算完成!")
}
def readFilePath(inputPath: String, outputPath:String, table:String, minPartitions:Int, mode:
String) = {
val sparkConf = new SparkConf().setAppName("HbaseJob")
   sparkConf.setMaster(mode)
val sc = new SparkContext(sparkConf)
//加载数据文件
val data = sc.textFile(inputPath,minPartitions)

   data.mapPartitions(batch(_,table)).saveAsTextFile(outputPath)
   sc.stop()
 }
def batch(keys: Iterator[String],hbaseTable:String) = {
val lineList = ListBuffer[String]()
import scala.collection.JavaConversions._
val conf = HBaseConfiguration.create()
//每批数据创建一个HBase连接,多条数据操作共享这个连接
val connection = HConnectionManager.createConnection(conf)
//获取表
val table = connection.getTable(hbaseTable)
   keys.foreach(rowKey =>{
try {
//根据rowKey主键也就是课程ID查询数据
val get = new Get(rowKey.getBytes())
//指定需要获取的列族和列
```

```
get.addColumn("kcname".getBytes(), "name".getBytes())
        get.addColumn("saleinfo".getBytes(), "price".getBytes())
        get.addColumn("saleinfo".getBytes(), "issale".getBytes())
val result = table.get(get)
var nameRS = result.getValue("kcname".getBytes(),"name".getBytes())
var kcName = "";
if(nameRS != null&&nameRS.length > 0){
          kcName = new String(nameRS);
        }
val priceRS = result.getValue("saleinfo".getBytes, "price".getBytes)
var price = ""
if (priceRS != null && priceRS.length > 0)
          price = new String(priceRS)
val issaleRS = result.getValue("saleinfo".getBytes, "issale".getBytes)
var issale = ""
if (issaleRS != null && issaleRS.length > 0)
          issale = new String(issaleRS)
        lineList += rowKey + "\001" + kcName + "\001" + price + "\001" + issale
      } catch {
case e: Exception => e.printStackTrace()
      }
    })
//每批数据操作完毕,别忘了关闭表和数据库连接
table.close()
    connection.close()
    lineList.toIterator
  }
}
```

代码开发完成后,我们看看怎样部署到 Spark 集群上去运行,运行的方式和我们的 Spark 集群怎样部署有关,Spark 集群部署有 3 种方式:Standalone 单独集群部署、Spark on Yarn 部署和 Local 本地模式,前两种都是分布式部署,后面的一种是单机方式。一般大数据部门都有 Hadoop 集群,所以推荐 Spark on Yarn 部署,这样更方便服务器资源的统一管理和分配。

Spark on Yarn 部署非常简单,主要是把 Spark 包解压就可以用了,在每台服务器上存放一份,并且放在相同的目录下。步骤如下:

1) 配置 Scala 环境变量

```
#解压 Scala 包,然后存放到 vim /etc/profile 目录
export SCALA_HOME = /home/hadoop/software/scala - 2.11.8
```

2) 解压 tar xvzf spark-*-bin-hadoop*.tgz,在每台 hadoop 服务器上存放在同一个目录下

不用任何配置值,用 spark-submit 提交就行。

Spark 环境部署好之后，把我们操作 HBase 的项目编译并打包，一个是项目本身的 jar，另一个是项目依赖的 jar 集合，分别上传到任意一台服务器就可以，不要每台服务器都传，在哪台服务器运行就在哪台服务器上上传，依赖的 jar 包放在目录/home/hadoop/chongdianleme/chongdianleme-spark-task-1.0.0/lib/下，项目本身的 jar 包存放在目录/home/hadoop/chongdianleme/下，然后通过 spark-submit 提交如下脚本即可。

```
hadoop fs -rmr /ods/kc/dim/ods_kc_dim_hbase/;
/home/hadoop/software/spark21/bin/spark-submit --jars $(echo /home/hadoop/chongdianleme/chongdianleme-spark-task-1.0.0/lib/*.jar | tr ' ',') --master yarn --queue hadoop --num-executors 1 --driver-memory 1g --executor-memory 1g --executor-cores 1 --class com.chongdianleme.mail.HbaseJob /home/hadoop/chongdianleme/hbase-task.jar --inputPath /mid/kc/dim/mid_kc_dim_kcidlist/ --outputPath /ods/kc/dim/ods_kc_dim_hbase/ --table chongdianleme_kc --minPartitions 6 --mode yarn
```

其中 hadoop fs -rmr /ods/kc/dim/ods_kc_dim_hbase/；是为了在下次执行这个任务时避免输出目录已经存在，我们先把输出目录删除，执行完之后输出目录会重新生成。

脚本参数说明：

--jars：你的程序所依赖的所有 jar 存放的目录。

--master：指定在哪里运算，如果在 Hadoop 的 Yarn 上运算则写 Yarn，如果以本地方式运算则写 Local。

--queue：如果是 Yarn 方式，就指定分配到哪个队列的资源上。

--num-executors：指定运行几个 Task。

--driver.maxResultSize：driver 的最大内存设置，默认为 1GB，比较小。超过了会内存溢出(Out of Memory，OOM)，可以根据情况设置大一些。

--executor-memory：为每个 Task 分配内存。

--executor-cores：每个 Task 分配几个虚拟 CPU。

--class：你的程序的入口类，后面跟 jar 包，再后面是 Java 或 Scala 的 main 函数的业务参数。

这就是我们从编程、编译、打包和如何部署到服务器进行分布式运行的完整过程，后面章节讲解的 Spark 分布式机器学习也是通过这种方式打包和部署的。