# 开源大数据平台

目录

[开源大数据平台 1](#_Toc503878754)

[1. 前期准备 2](#_Toc503878755)

[2. JDK、Scala环境 2](#_Toc503878756)

[3. MySQL安装 3](#_Toc503878757)

[4. SSH无密钥访问 3](#_Toc503878758)

[5. Hadoop2.7.4安装 3](#_Toc503878759)

[6. Spark2.1.0安装 6](#_Toc503878760)

[7. Spark集群测试 7](#_Toc503878761)

[8. Zookeeper3.4.10部署 7](#_Toc503878762)

[9. Apache Flume1.8.0配置 8](#_Toc503878763)

[10. Kafka0.10配置 10](#_Toc503878764)

[11. HBase1.2.6安装 11](#_Toc503878765)

[12. Hive1.2.2安装 13](#_Toc503878766)

[13. Spark SQL 操作 Hive 14](#_Toc503878767)

[14. Parquet类型 17](#_Toc503878768)

[15. IDEA Scala项目 19](#_Toc503878769)

[16. 集群启动关闭脚本 20](#_Toc503878770)

[17. Docker容器 22](#_Toc503878771)

[18. RDD操作算子 24](#_Toc503878772)

[19. Spark Scala编程 26](#_Toc503878773)

[20. Spark MLlib应用 28](#_Toc503878774)

## 前期准备

Vmware虚拟机三台：workstation 10、RedHat 6.5

节点：bigdata1（主节点）、bigdata2 和 bigdata3 （从节点）

用户名：root 密码：123456

配置：内存2G（可扩充）、存储空间40G、单处理器

各节点静态IP：10.1.29.128、10.1.29.133、10.1.29.134

组件版本：

**需要考虑组件之间的适配性、组件之间的相互依赖、最低兼容版本**

**jdk1.8.0\_151、hadoop2.7.4、spark2.1.0、scala2.10.4、kafka2.10-0.10.2.0、apache flume1.8.0、zookeeper3.4.10、apache hive1.2.2**

## JDK、Scala环境

1. 安装JDK

先卸载系统自带的 OpenJDK Runtime Environment

rpm -qa | grep jdk 查看存在的jdk包

rpm -e --nodeps java-1.6.0-openjdk-1.6.0.0 删除存在的包及其依赖

再安装自己的jdk：

tar -zxvf jdk-7u79

添加环境变量 vim /etc/profile

JAVA\_HOME=/usr/java/jdk1.7.0\_79

CLASSPATH=.:$JAVA\_HOME/lib.tools.jar

PATH=$JAVA\_HOME/bin:$PATH

export JAVA\_HOME CLASSPATH PATH

再使其生效 source /etc/profile

查看当前jdk版本是否为安装版本 java -version

或者echo $JAVA\_HOME

1. 安装scala

解压安装包 tar -zxvf scala-2.10.4.tgz

#增加SCALA\_HOME

vim /etc/profile

#增加如下内容;

#Scala Home

export SCALA\_HOME=/usr/share/scala

#刷新配置

source /etc/profile

## MySQL安装

MySql安装在bigdata3节点上

先启动mysql服务 service mysqld start

再使用mysql -u root -p root 登录

创建Hive用户：

mysql> CREATE USER 'hive' IDENTIFIED BY 'hive';

mysql> GRANT ALL PRIVILEGES ON \*.\* TO 'hive'@'bigdata3' WITH GRANT OPTION;

mysql> GRANT ALL PRIVILEGES ON \*.\* TO 'hive'@'bigdata3' IDENTIFIED BY 'hive';

GRANT ALL PRIVILEGES ON \*.\* TO 'hive'@'%' IDENTIFIED BY 'hive' WITH GRANT OPTION;

mysql>flush privileges;

创建Hive数据库:

mysql>create database hive;

**注意：**

启动mysql服务 service mysqld start

出现如下错误Can't connect to local MySQL server through socket '/var/lib/mysql/mysql.sock'

可能是多个mysql进程占用同一个socket

修改mv /var/lib/mysql/mysql.sock /var/lib/mysql/mysql.sock.template重新启动服务

使用hive用户登录mysql：

mysql -h bigdata3 -u hive -p 输入密码hive登入mysql

## SSH无密钥访问

修改主机名：重启后生效

vim /etc/sysconfig/network

在每个节点上执行如下命令

ssh-keygen -t rsa -P ''

ssh-copy-id -i ~/.ssh/id\_rsa.pub <IP> //<IP>替换成需要连接的机器IP地址 包括自身

10.1.29.128、10.1.29.133、10.1.29.134

## Hadoop2.7.4安装

解压jar包

修改相应的配置文件：

（1）/etc/profile：

增加如下内容：

#hadoop enviroment

export HADOOP\_HOME=/opt/hadoop-2.7.3/

export PATH="$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin:$PATH"

export HADOOP\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

export YARN\_CONF\_DIR=$HADOOP\_HOME/etc/hadoop

（2）$HADOOP\_HOME/etc/hadoop/hadoop-env.sh

修改JAVA\_HOME 如下：

export JAVA\_HOME=/usr/java/jdk1.8.0\_112/

（3）$HADOOP\_HOME/etc/hadoop/slaves

bigdata2

bigdata3

（4）$HADOOP\_HOME/etc/hadoop/core-site.xml

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://bigdata1:9000</value>

</property>

<property>

<name>io.file.buffer.size</name>

<value>131072</value>

</property>

<property>

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

<value>/software/hadoop-2.7.4/tmp</value>

</property>

</configuration>

（5）$HADOOP\_HOME/etc/hadoop/hdfs-site.xml

<configuration>

<property>

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

<value>bigdata1:50090</value>

</property>

<property>

<name>dfs.replication</name>

<value>2</value>

</property>

<property>

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

<value>file:/software/hadoop-2.7.4/hdfs/name</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>file:/software/hadoop-2.7.4/hdfs/data</value>

</property>

</configuration>

（6）$HADOOP\_HOME/etc/hadoop/mapred-site.xml

复制template，生成xml：

cp mapred-site.xml.template mapred-site.xml

修改内容：

<configuration>

<property>

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

<value>yarn</value>

</property>

<property>

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

<value> bigdata1:10020</value>

</property>

<property>

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

<value> bigdata1:19888</value>

</property>

</configuration>

（7）$HADOOP\_HOME/etc/hadoop/yarn-site.xml

<!-- Site specific YARN configuration properties -->

<property>

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

<value>mapreduce\_shuffle</value>

</property>

<property>

<name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>

<value>org.apache.hadoop.mapred.ShuffleHandler</value>

</property>

<property>

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

<value>bigdata1:8032</value>

</property>

<property>

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

<value>bigdata1:8030</value>

</property>

<property>

<name>yarn.resourcemanager.resource-tracker.address</name>

<value>bigdata1:8031</value>

</property>

<property>

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

<value>bigdata1:8033</value>

</property>

<property>

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

<value>bigdata1:8088</value>

</property>

<property>

<name>yarn.nodemanager.resource.memory-mb</name>

<value>768</value>

</property>

至此master节点的hadoop搭建完毕

再启动之前我们需要

格式化一下namenode

hadoop namenode -format

## Spark2.1.0安装

修改相应的配置文件：

（1）/etc/profie

#Spark enviroment

export SPARK\_HOME=/opt/spark-2.1.0-bin-hadoop2.7/

export PATH="$SPARK\_HOME/bin:$PATH"

（2）$SPARK\_HOME/conf/spark-env.sh

cp spark-env.sh.template spark-env.sh

#配置内容如下：

export SCALA\_HOME=/usr/share/scala

export JAVA\_HOME=/usr/java/jdk1.8.0\_112/

export SPARK\_MASTER\_IP=master

export SPARK\_WORKER\_MEMORY=1g

export HADOOP\_CONF\_DIR=/opt/hadoop-2.7.3/etc/hadoop

（3）$SPARK\_HOME/conf/slaves

cp slaves.template slaves

配置内容如下

bigdata2

bigdata3

（4）将配置好的spark文件复制到worker节点

scp spark-2.1.0-bin-hadoop2.7 root@bigdata2:/software

修改/etc/profile，增加spark相关的配置，如MASTER节点一样

至此 Spark配置完成

## Spark集群测试

1. 启动集群的脚本：

start-cluster.sh 内容如下：

#!/bin/bash

echo -e "\033[31m ========Start The Cluster======== \033[0m"

echo -e "\033[31m Starting Hadoop Now !!! \033[0m"

/software/hadoop-2.7.4/sbin/start-all.sh

echo -e "\033[31m Starting Spark Now !!! \033[0m"

/software/spark-2.1.0-bin-hadoop2.7 /sbin/start-all.sh

echo -e "\033[31m The Result Of The Command \"jps\" : \033[0m"

jps

echo -e "\033[31m ========END======== \033[0m"

1. 关闭集群脚本stop-cluser.sh如下：

#!/bin/bash

echo -e "\033[31m ===== Stoping The Cluster ====== \033[0m"

echo -e "\033[31m Stoping Spark Now !!! \033[0m"

/ software /spark-2.1.0-bin-hadoop2.7/sbin/stop-all.sh

echo -e "\033[31m Stopting Hadoop Now !!! \033[0m"

/ software /hadoop-2.7.4/sbin/stop-all.sh

echo -e "\033[31m The Result Of The Command \"jps\" : \033[0m"

jps

echo -e "\033[31m ======END======== \033[0m"

## 8. Zookeeper3.4.10部署

1）下载zookeeper安装文件（zookeeper-3.4.5压缩包）

2）将zookeeper-3.4.5压缩包加压到flumekafka1虚拟机的/home/bms目录下

3）配置zookeeper环境变量

l  vim /etc/profile

l  在上述文件中添加如下内容

export  ZOOKEEPER\_HOME=/home/bms/zookeeper-3.4.5

export PATH=$PATH:${ZOOKEEPER\_HOME }/bin

4）将/home/bms/zookeeper-3.4.5/conf/ 目录下的zoo\_sample.cfg 改名成 zoo.cfg操作命令是：

l  cd /home/bms/zookeeper-3.4.5/conf

l  mv zoo\_sample.cfg  zoo.cfg

5）修改zoo.cfg文件内容如下

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can passbetween

# sending a request and getting anacknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/home/bms/zookeeper-3.4.5/zookeeper-data

# the port at which the clients will connect

clientPort=2222

server.1=flumekafka1:2888:3888

server.2= flumekafk21:2888:3888

server.3= flumekafka3:2888:3888

6）分别在将flunekafka1上/home/bms/zookeeper-3.4.5整个目录拷贝到另外两台slave虚拟机上

注意：最后记得在部署zookeeper的节点上的/root/zookeeper/zookeeper-data的目录下新建一个myid文件里写上zoo.cfg文件对应的server号码，flumekafka1写1，flumekafka2写2，flumekafka3写3，这样基本就完成了zookeeper的配置了。

7）分别在三台slave机器上执行下面的操作

l  cd /home/bms/zookeeper-3.4.5/bin

l  zookeeper/bin/zkServer.sh start

8)     分别在三台slave机器上执行下面的操作

l  zookeeper/bin/zkServer.sh status(验证zookeeper的启动状态)

9）在master上面执行下面的命令

l  cd /homehbase/bin/start-hbase.sh

## Apache Flume1.8.0配置

配置Flume：

1. 配置kafka-s.cfg

$ cd /software/flume/conf/ #切换到kafka的配置文件目录

$ cp flume-conf.properties.template kafka-s.cfg #将flume的配置文件模板复制为kafka-s.cfg

kafka-s.cfg的详细内容如下：

#配置flume agent的source、channel、sink

a1.sources = r1

a1.channels = c1

a1.sinks=k1

#配置source

a1.sources.r1.type = exec

a1.sources.r1.command=tail -F /tmp/logs/kafka.log

# 配置channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

#配置sink

a1.sinks.k1.channel = c1

a1.sinks.k1.type =org.apache.flume.sink.kafka.KafkaSink

#设置Kafka的Topic

a1.sinks.k1.kafka.topic = topicTest

#设置Kafka的broker地址和端口号

a1.sinks.k1.kafka.bootstrap.servers = bigdata1:9092,bigdata2:9092,bigdata3:9092

#配置批量提交的数量

a1.sinks.k1.kafka.flumeBatchSize = 20

a1.sinks.k1.kafka.producer.acks = 1

a1.sinks.k1.kafka.producer.linger.ms = 1

a1.sinks.ki.kafka.producer.compression.type= snappy

# 绑定source和sink 到channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel=c1

关于上面配置文件中需要注意的3点：

a、a1.sources.r1.command=tail-F /tmp/logs/kafka.log

b、a1.sinks.k1.kafka.bootstrap.servers= s201:9092,s202:9092,s203:9092

c、a1.sinks.k1.topic= mytest

由配置文件我们可以了解到：

1）我们需要在/tmp/logs下建一个kafka.log的文件，且向文件中输出内容；

2）flume连接到kafka的地址是 s201:9092,s202:9092,s203:9092，注意不要配置错误了；

3）flume会将采集后的内容输出到Kafkatopic 为mytest上，所以我们启动zk，kafka后需要打开一个终端消费topic mytest的内容，这样就可以看到flume与kafka之间玩起来了。

2. 创建/tmp/logs/kafka.log文件目录及文件

在/tmp/logs下建立空文件kafka.log；如果/tmp目录下没有logs目录，则需要先创建logs目录。

3. 创建生成日志数据shell脚本

在hadoop用户目录下创建kafkaoutput.sh脚本，并赋予执行权限，用来向/tmp/logs/kafka.log输出内容。

kafkaoutput.sh脚本的具体内容如下：

for((i=0;i<=1000;i++));

do echo"kafka\_test-"+$i>>/tmp/logs/kafka.log;

done

4. 启动Flume

$ flume-ng agent --conf /software/flume/conf/ --conf-file flume-conf.properties --name a1 -Dflume.root.logger=INFO,console

## Kafka0.10配置

Zookeeper与Kafka的配置

1. 环境是Linux操作系统，现有三台虚拟机（IP分别是192.168.100.104、192.168.100.105、192.168.100.106）

2. 首先要做的事情是将上述三个虚拟机的的hostname分别进行修改为：flumekafka1、flumekafka2、flumekafka3

3. 下面进行相关的配置：

（1）虚拟机（192.168.100.104）进行如下操作

a）对文件： $KAFKA\_HOME/config/server.properties进行修改编辑：

broker.id=0

port=9092

host.name=flumekafka1

advertised.host.name=flumekafka1

num.partitions=2

zookeeper.contact=flumekafka1:2181,flumekafka2:2181,flumekafka3:2181

（2）虚拟机（192.168.100.105）进行如下操作

a）对文件： $KAFKA\_HOME/config/server.properties进行修改编辑：

broker.id=1

port=9092

host.name=flumekafka2

advertised.host.name=flumekafka2

num.partitions=2

zookeeper.contact=flumekafka1:2181,flumekafka2:2181,flumekafka3:2181

（3）虚拟机（192.168.100.106）进行如下操作

a）对文件： $KAFKA\_HOME/config/server.properties进行修改编辑：

broker.id=2

port=9092

host.name=flumekafka3

advertised.host.name=flumekafka3

num.partitions=2

zookeeper.contact=flumekafka1:2181,flumekafka2:2181,flumekafka3:2181

（4)启动kafka服务, 在flumekafka1,flumekafka2,flumekafka3上分别运行：

$ kafka-server-start.sh $KAFKA\_HOME/config/server.properties

（5) 新建一个TOPIC(replication-factor=num of brokers)

$ kafka-topics.sh --create --topic flume --replication-factor 1 --partitions 1 --zookeeperflumekafka1:2181

（6)在flumekafka2上，开一个终端，发送消息至kafka队列

$ kafka-console-producer.sh --broker-list flumekafka1:9092 --sync --topic flume

然后在发送消息的终端输入：Hello Kafka

（7）在flumekafka3上，开一个终端，显示消息的消费

$ kafka-console-consumer.sh --zookeeper bigdata1:2181 --topic topicTest --from-beginning

## HBase1.2.6安装

1. 在Hadoop配置的基础上，配置环境变量HBASE\_HOME、hbase-env.sh

　　编辑 vim /etc/profile 加入：

#Hbase environment

　　　　export HBASE\_HOME=/software/hbase

　　　　export PATH=$HBASE\_HOME/bin:$PATH

　　编辑vim /opt/hbase-1.2.6/conf/hbase-env.sh 加入：

export JAVA\_HOME=/opt/java\_environment/jdk1.7.0\_80（jdk安装路径）

去掉注释 # export HBASE\_MANAGES\_ZK=true，使用hbase自带zookeeper。

1. 配置hbase-site.xml文件

　　　　<configuration>

　　　　　　<property>

　　　　　　　　<name>hbase.rootdir</name> <!-- hbase存放数据目录 -->

　　　　　　　　<value>hdfs://bigdata1:9000/opt/hbase/hbase\_db</value>

　　　　　　　 <!-- 端口要和Hadoop的fs.defaultFS端口一致-->

　　　　　　</property>

　　　　　　<property>

　　　　　　　　<name>hbase.cluster.distributed</name> <!-- 是否分布式部署 -->

　　　　　　　　<value>true</value>

　　　　　　</property>

　　　　　　<property>

　　　　　　　　<name>hbase.zookeeper.quorum</name> <!-- list of zookooper -->

　　　　　　　　<value> bigdata1,bigdata2,bigdata3</value>

　　　　　　</property>

　　　　　　<property><!--zookooper配置、日志等的存储位置 -->

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

　　　　　　　　　<value>/software/hbase/zookeeper</value>

　　　　　　</property>

　　　　</configuration>

1. 配置regionservers

　　编辑 vim /opt/hbase-1.2.6/conf/regionservers 去掉默认的localhost，

　　加入slave1、slave2，写成二行保存退出

　　然后把在master上配置好的hbase，通过远程复制命令

　　scp -r /opt/hbase-1.2.6 192.168.172.72/73:/opt/hbase-1.2.6

　　复制到slave1、slave2对应的位置

4. 启动与停止Hbase

　（1）在Hadoop已经启动成功的基础上，输入start-hbase.sh，过几秒钟便启动完成，

　　　输入jps命令查看进程是否启动成功，若 master上出现HMaster、HQuormPeer，

　　　slave上出现HRegionServer、HQuorumPeer，就是启动成功了。

　 (2）输入hbase shell 命令 进入hbase命令模式

　　　 输入status命令可以看到如下内容，1个master，2 servers，3机器全部成功启动。

　 　1 active master, 0 backup masters, 2 servers, 0 dead, 2.0000 average load

　（3）接下来配置本地hosts，（前边配置过的无需再配置了）

　　　编辑 C:\Windows\System32\drivers\etc的hosts文件，加入

192.168.172.71 master

192.168.172.72 slave1

　　　 192.168.172.73 slave2

　在浏览器中输入http://master:16010就可以在界面上看到hbase的配置了

（4）当要停止hbase时输入stop-hbase.sh，过几秒后hbase就会被停止了。

## Hive1.2.2安装

Hive是基于Hadoop构建的一套数据仓库分析系统，它提供了丰富的SQL查询方式来分析存储在Hadoop 分布式文件系统中的数据。其在Hadoop的架构体系中承担了一个SQL解析的过程，它提供了对外的入口来获取用户的指令然后对指令进行分析，解析出一个MapReduce程序组成可执行计划，并按照该计划生成对应的MapReduce任务提交给Hadoop集群处理，获取最终的结果。元数据——如表模式——存储在名为metastore的数据库中。

1. 配置系统环境变量：

修改 /etc/profile 文件 vi /etc/profile 来修改（root用户操作）：

设置 Hive环境变量：

# Hive environment

export HIVE\_HOME=/software/hive

export PATH=$HIVE\_HOME/bin:$HIVE\_HOME/conf:$PATH

使环境变量生效:

source /etc/profile

1. 这里选用mySql作为元数据库，将mySql和Hive安装在服务器上

3. 修改Hive配置文件:   
cd /apache-hive-1.2.1-bin/conf   
cp hive-default.xml.template hive-site.xml   
vim hive-site.xml ：

<configuration>

<property>

<name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://bigdata3:3306/hive?createDatabaseIfNotExist=true</value>

<description>JDBC connect string for a JDBC metastore</description>

</property>

<property>

<name>javax.jdo.option.ConnectionDriverName</name>

<value>com.mysql.jdbc.Driver</value>

<description>Driver class name for a JDBC metastore</description>

</property>

<property>

<name>javax.jdo.option.ConnectionUserName</name>

<value>hive</value>

<description>username to use against metastore database</description>

</property>

<property>

<name>javax.jdo.option.ConnectionPassword</name>

<value>hive</value>

<description>password to use against metastore database</description>

</property>

</configuration>

4. 拷贝JDBC驱动包   
将mySql的JDBC驱动包复制到Hive的lib目录下   
cp mysql-connector-java.bin.jar /usr/local/hadoop/ apache-hive-1.2.1-bin /lib

5. 分发Hive分别到slave1,slave2,slave3上   
scp -r /usr/local/hadoop/apache-hive-1.2.1-bin slave1:/usr/local/hadoop/   
scp -r /usr/local/hadoop/apache-hive-1.2.1-bin slave2:/usr/local/hadoop/   
scp -r /usr/local/hadoop/apache-hive-1.2.1-bin slave3:/usr/local/hadoop/ 

6. 测试Hive

**注意：在测试Hive之前要启动hadoop 、MySQL**  
进入到Hive的安装目录，命令行：   
cd /usr/local/hadoop/apache-hive-1.2.1-bin/bin   
hive   
hive>show tables;   
正常显示，即是安装配置成功。 

## Spark SQL 操作 Hive

1. Spark配置：  
将hive的配置文件拷贝给spark  
 copy $HIVE\_HOME/conf/hive-site.xml $SPARK\_HOME/conf/  
将mysql的jdbc驱动包拷贝给spark  
 copy $HIVE\_HOME/lib/mysql-connector-java-5.1.12.jar或者软链到$SPARK\_HOME/lib/

2. 启动spark-sql的shell交互界面  
spark-sql已经集成在spark-shell中，启动spark-shell使用spakr-sql的shell交互接口  
 bin/spark-shell --master spark://bigdata1:7077 --jars jars/mysql-connector-java-5.1.34-bin.jar

或者启动spark-sql界面  
 bin/spark-sql --master spark://bigdata1:7077 --jars jars/mysql-connector-java-5.1.34- bin.jar

就可以使用hivesql了

3. 在交互界面输入sql进行查询  
注：以下所用到的库和表，都是已经在hive中存在的库和表  
 如果在spark-shell中执行sql查询，使用sqlContext对象调用sql()方法  
scala> sqlContext.sql("select remote\_addr from dw\_weblog.t\_ods\_detail group by remote\_addr").collect.foreach(println)

Scala> sqlContext.sql(“show databases;”)  
  
如果是在spark-sql中执行sql查询，则可以直接输入sql语句  
scala> show databases  
scala> use dw\_weblog  
scala> select remote\_addr from dw\_weblog.t\_ods\_detail group by remote\_addr  
  
**如果是spark-2.1.0-bin-hadoop2.7,它没有sqlContext，所以要先执行：val sqlContext = new org.apache.spark.sql.SQLContext(sc)。**

4. 在IDEA中编写代码使用hive-sql  
如下所示：  
val hiveContext = new HiveContext(sc)  
    import hiveContext.implicits.\_  
    import hiveContext.sql  
    //指定库  
sql("use dw\_weblog")  
//执行标准sql语句  
sql("create table sparksql as select remote\_addr,count(\*) from t\_ods\_detail group by remote\_addr")

综上所述，sparksql类似于hive，可以支持sql语法来对海量数据进行分析查询，跟hive不同的是，hive执行sql任务的底层运算引擎采用mapreduce运算框架，而sparksql执行sql任务的运算引擎是spark core，从而充分利用spark内存计算及DAG模型的优势，大幅提升海量数据的分析查询速度

**附录：Hive测试环境 http://blog.csdn.net/tototuzuoquan/article/details/75007376**

使用sqlContext.sql调用HQL

在使用之前先要启动hive，创建person表：

hive> create table person(id bigint,name string,age int) row format delimited fields terminated by " " ;

查看hdfs中person的内容：

load数据到person表中：

hive> load data inpath '/person.txt' into table person;

**如果是spark-2.1.1-bin-hadoop2.7,它没有sqlContext，所以要先执行：val sqlContext = new org.apache.spark.sql.SQLContext(sc)**

如果是spark-1.6.2-bin-hadoop2.6，不用执行：val sqlContext = new org.apache.spark.sql.SQLContext(sc)

scala> sqlContext.sql("select \* from person limit 2")

+---+--------+---+

| id| name|age|

+---+--------+---+

| 1|zhangsan| 19|

| 2| lisi| 20|

+---+--------+---+

scala>

或使用org.apache.spark.sql.hive.HiveContext （同样是在spark-sql这个shell命令下）

scala> import org.apache.spark.sql.hive.HiveContext

scala> val hiveContext = new HiveContext(sc)

scala> hiveContext.sql("select \* from person")

scala> hiveContext.sql("select \* from person").show

+---+-----------+---+

| id| name|age|

+---+-----------+---+

| 1| zhangsan| 19|

| 2| lisi| 20|

| 3| wangwu| 28|

| 4| zhaoliu| 26|

| 5| tianqi| 24|

| 6| chengnong| 55|

| 7|zhouxingchi| 58|

| 8| mayun| 50|

| 9| yangliying| 30|

| 10| lilianjie| 51|

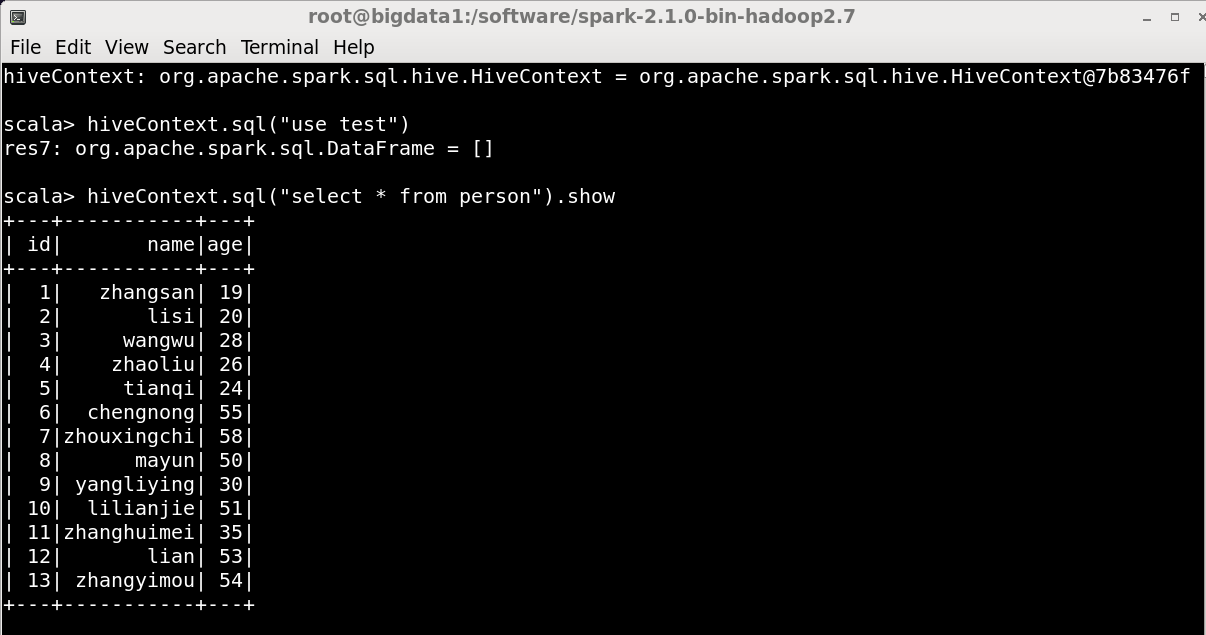
| 11|zhanghuimei| 35|

| 12| lian| 53|

| 13| zhangyimou| 54|

+---+-----------+---+

结果如下图所示：



## Parquet类型

1. 摘自官网：

Apache Parquet（读音paki） is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

Parquet是列式存储格式的一种文件类型，列式存储有以下的核心：

A.可以跳过不符合条件的数据，只读取需要的数据，降低IO数据量。

B.压缩编码可以降低磁盘存储空间。由于同一列的数据类型是一样的，可以使用更高效的压缩编码（例如Run Length Encoding和Delta Encoding）进一步节约存储空间。

C.只读取需要的列，支持向量运算，能够获取更好的扫描性能。

1. Spark SQL下的Parquet

A. 如果说HDFS是大数据时代文件系统的事实标准的话，Parquet就是大数据时代存储格式的事实标准；

B. 速度更快：从使用Spark SQL操作普通文件CSV和Parquet文件的速度对比上来看，绝大多数情况下使用Parquet会比使用CSV等普通文件速度提升10倍左右；（在一些普通文件系统无法再Spark上成功运行程序的情况下，使用Parquet很多时候都可以成功运行）；

C. Parquet的压缩技术非常稳定出色，在Spark SQL中对压缩技术的处理可能无法正常的完成工作（例如会导致Lost Task，Lost Exexutor），但是此时如果使用Parquet就可以正常的完成；

D. 极大的减少磁盘I/O，通常情况下能够减少75%的存储空间，由此可以极大地减少Spark SQL处理数据的时候的数据输入内容，尤其是在Spark 1.6.x中下推过滤器在一些情况下可以极大的进一步减少磁盘的I/O和内存的占用；

E. Spark 1.6.x+Parquet极大的提升了数据扫描的吞吐量，这极大的提高了数据的查找速度，Spark 1.6和Spark 1.5相比较而言提升了1倍的速度，在Spark 1.6.x中操作Parquet时候CPU的使用也进行了极大的优化，有效的降低了CPU的使用；

F. 采用Parquet可以极大的优化Spark的调度和执行，我们测试表面Spark如果采用Parquet可以有效的减少Stage的执行消耗，同时可以优化执行路径；

1. 读parquet文件

首先创建hive表，数据用tab分隔

create table test(name string, age int)

row format delimited

fields terminated by '\t';

加载数据

load data local inpath '/home/work/test/ddd.txt' into table test;

hive> select \* from test limit 5;

创建parquet格式表

create table test\_parquet(name string,age int) stored as parquet

查看表结构

hive> show create table test\_parquet;

可以看到数据的inputFormat是MapredParquetInputFormat

往parquet格式表中插入数据

insert into table test\_parquet select \* from test;

1. 用spark DataFrame 解析parquet文件

val conf = new SparkConf().setAppName("test").setMaster("local[2]")

val sc = new SparkContext(conf)

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val parquet : DataFrame = sqlContext.read.parquet("hdfs://192.168.1.115:9000/user/hive/warehouse/test\_parquet")

parquet.printSchema()

parquet.select(parquet("name"), parquet("age") + 1).show()

1. 用hive sql读取hive表

用hivesql直接读取hive表

在local模式下没有测试成功，打包用spark-submit测试，代码如下

val conf = new SparkConf().setAppName("test")

val sc = new SparkContext(conf)

val hiveContext = new HiveContext(sc)

val sql: DataFrame = hiveContext.sql("select \* from test\_parquet limit 10")

sql.take(10).foreach(println)

1. Spark RDD写parquet文件

//读取文件生成RDD

val file = sc.textFile("person.txt")

//定义parquet的schema，数据字段和数据类型需要和hive表中的字段和数据类型相同，否则hive表无法解析

val schema = (new StructType)

.add("id", IntegerType, false) //字段名、类型、是否允许为空

.add("name", StringType, true)

.add("age", IntegerType, true)

val rowRDD = file.map(\_.split("\t")).map(p => Row(Integer.valueOf(p(0).trim), p(1).trim, Integer.valueOf(p(2).trim)))

//将RDD转换成DataFrame

val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

//注册临时表

peopleDataFrame.registerTempTable("people")

//写parquet文件

peopleDataFrame.write.parquet("test\_parquet")

//查询表

sqlContext.sql(“select \* from people where sex = ‘m’”).show()

## IDEA Scala项目

1. IDEA创建编译打包scala maven项目

创建项目：

首先IDEA安装scala和maven插件 idea ultimate版自带maven插件

不能翻墙的可以手动下载scala插件 然后选择从本地安装

File -> New -> Project -> Scala

编译打包：

File -> Project Structure -> Artifacts -> +jars with dependency -> Main Class

Build -> Build Artifacts

在项目路径的out/artifacts/路径下生成相应的jar包

控制台打包命令：

mvn clean compile package 直接打包java项目

mvn scala:compile compile package 先编译scala代码再打包项目

1. 导入spark-core、spark-sql、saprk-hive包

Spark安装路径的lib下或者jars下提供了相应版本的这些包 可直接去拿

或者pom文件中添加需要的依赖

pom中添加依赖时需要注意顺序jar之间的冲突（很麻烦需要仔细）

## 集群启动关闭脚本

nohup <command> & 在后台执行命令 可以根据进程号查看或者结束进程

**Linux Shell远程执行命令：**

1）命令行执行登录并且在目标服务器上执行命令

[java] view plaincopy在CODE上查看代码片派生到我的代码片

ssh user@remoteNode "cd /home ; ls"

基本能完成常用的对于远程节点的管理了，几个注意的点：

双引号，必须有。如果不加双引号，第二个ls命令在本地执行

分号，两个命令之间用分号隔开

2）脚本的方式

这个正式我想要的方式，在目标服务器上执行批量的命令。

#!/bin/bash

**ssh root@192.168.0.23 << remotessh**

**killall -9 java**

**cd /data/apache-tomcat-7.0.53/webapps/**

**exit**

**remotessh**

远程执行的内容在“<< remotessh ” 至“ remotessh ”之间，在远程机器上的操作就位于其中，注意的点：<< remotessh，ssh后直到遇到remotessh这样的内容结束，remotessh可以随便修改成其他形式。在结束前，加exit退出远程节点

**1.启动Hadoop+Spark集群脚本：**

start-cluster.sh 内容如下：

#!/bin/bash

echo -e "\033[31m ========Start The Cluster======== \033[0m"

echo -e "\033[31m Starting Hadoop Now !!! \033[0m"

/software/hadoop-2.7.4/sbin/start-all.sh

echo -e "\033[31m Starting Spark Now !!! \033[0m"

/software/spark-2.1.0-bin-hadoop2.7 /sbin/start-all.sh

echo -e "\033[31m The Result Of The Command \"jps\" : \033[0m"

jps

echo -e "\033[31m ========END======== \033[0m"

2.关闭Hadoop+Spark集群脚本

stop-cluser.sh内容如下：

#!/bin/bash

echo -e "\033[31m ===== Stoping The Cluster ====== \033[0m"

echo -e "\033[31m Stoping Spark Now !!! \033[0m"

/ software /spark-2.1.0-bin-hadoop2.7/sbin/stop-all.sh

echo -e "\033[31m Stopting Hadoop Now !!! \033[0m"

/ software /hadoop-2.7.4/sbin/stop-all.sh

echo -e "\033[31m The Result Of The Command \"jps\" : \033[0m"

jps

echo -e "\033[31m ======END======== \033[0m"

1. **启动flume+kafka+sparkstreaming+hdfs集群脚本**

start-datastreaming.sh 内容如下：

#!/bin/bash

echo -e "\033[31m ========Start DataStreaming Cluster======== \033[0m"

echo -e "\033[31m Starting Hadoop Now !!! \033[0m"

/software/hadoop-2.7.4/sbin/start-all.sh

echo -e "\033[31m Starting Zookeeper Now !!! \033[0m"

$ZOOKEEPER\_HOME/bin/zkServer.sh start

ssh root@bigdata2 << remotessh

$ZOOKEEPER\_HOME/bin/zkServer.sh start

remotessh

ssh root@bigdata3 << remoteshell

$ZOOKEEPER\_HOME/bin/zkServer.sh start

remoteshell

echo -e "\033[31m Starting Kafka Now !!! \033[0m"

nohup /software/kafka/bin/kafka-server-start.sh /software/kafka/config/server.properties &

echo -e "\033[31m Create Topic kafkaTest \033[0m"

nohup /software/kafka/bin/kafka-topics.sh --create --topic topicTest --replication-factor 1 --partitions 1 --zookeeper bigdata1:2181 &

/software/kafka/bin/kafka-topics.sh --zookeeper bigdata1:2181 --list

echo -e "\033[31m Starting Flume Now !!! \033[0m"

cd /software/flume

./bin/flume-ng agent --conf conf --conf-file conf/flume-conf.properties --name a1 -Dflume.root.logger=INFO,console &

echo -e "\033[31m Starting Spark Now !!! \033[0m"

/software/spark-2.1.0-bin-hadoop2.7/sbin/start-all.sh

echo -e "\033[31m The Result Of The Command \"jps\" : \033[0m"

jps

echo -e "\033[31m ========END======== \033[0m"

1. 关闭flume+kafka+sparkstreaming+hdfs集群脚本

stop-datastreaming.sh

部分进程还需要手动关闭

## Docker容器

介绍：

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从Apache2.0协议开源。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口,更重要的是容器性能开销极低。

应用场景：

Web 应用的自动化打包和发布。

自动化测试和持续集成、发布。

在服务型环境中部署和调整数据库或其他的后台应用。

从头编译或者扩展现有的OpenShift或Cloud Foundry平台来搭建自己的PaaS环境。

优点：

1、简化程序：

Docker 让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，便可以实现虚拟化。Docker改变了虚拟化的方式，使开发者可以直接将自己的成果放入Docker中进行管理。方便快捷已经是 Docker的最大优势，过去需要用数天乃至数周的 任务，在Docker容器的处理下，只需要数秒就能完成。

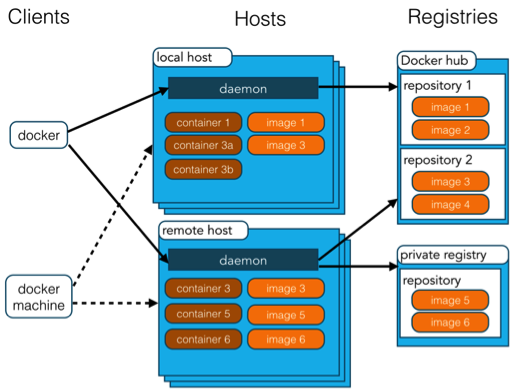
2、避免选择恐惧症：

如果你有选择恐惧症，还是资深患者。Docker 帮你打包你的纠结，比如 Docker 镜像；Docker 镜像中包含了运行环境和配置，所以 Docker 可以简化部署多种应用实例工作。比如 Web 应用、后台应用、数据库应用、大数据应用比如 Hadoop 集群、消息队列等等都可以打包成一个镜像部署。

3、节省开支：

一方面，云计算时代到来，使开发者不必为了追求效果而配置高额的硬件，Docker 改变了高性能必然高价格的思维定势。Docker 与云的结合，让云空间得到更充分的利用。不仅解决了硬件管理的问题，也改变了虚拟化的方式。

Docker容器 架构：



安装与使用：

Docker1.12在Ubuntu16.04上安装 对系统内核版本有要求 安装前先确认

1. 先决条件

添加Docker源

wxl@wxl-pc:~$ sudo apt-get update

增加CA证书

wxl@wxl-pc:~$ sudo apt-get install apt-transport-https ca-certificates

添加GPG Key(一种加密手段)

wxl@wxl-pc:~$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D

创建docker.list文件

wxl@wxl-pc:~$ sudo vim /etc/apt/sources.list.d/docker.list

#添加Ubuntu16.04LST的入口

deb https://apt.dockerproject.org/repo ubuntu-xenial main

再次更新源

wxl@wxl-pc:~$ sudo apt-get update

以防万一，清除过时的源

wxl@wxl-pc:~$ sudo apt-get purge lxc-docker

验证下APT是从正确的库源下载应用的

wxl@wxl-pc:~$ apt-cache policy docker-engine

至此，可见已经配置好了Docker的源

1. 先决条件

安装aufs驱动linux-image-extra

For Ubuntu Trusty, Wily, and Xenial, it’s recommended to install the linux-image-extra kernel package. The linux-image-extra package allows you use the aufs storage driver可以实现容器间可执行文件和运行库的共享。

更新源，会发现Hit:9 https://apt.dockerproject.org/repo ubuntu-xenial InRelease，也说明Docker在第一步1设置成功。

wxl@wxl-pc:~$ sudo apt-get update

安装 linux-image-extra

wxl@wxl-pc:~$ sudo apt-get install linux-image-extra-$(uname -r)

1. 安装Docker（如果先决条件1,2步正确完成了）

更新源

wxl@wxl-pc:~$ sudo apt-get update

通过apt命令在线安装docker

wxl@wxl-pc:~$ sudo apt-get install docker-engine

开启docker的守护进程（Docker服务开启）

wxl@wxl-pc:~$ sudo service docker start

国际惯例，用一个Hello world的来测试安装成功

wxl@wxl-pc:~$ sudo docker run hello-world

本地本来没有Hello World镜像，通过Docker源获取到，并成功现实Hello world。

## RDD操作算子

|  |  |  |
| --- | --- | --- |
| 操作类型 | 函数名 | 作用 |
| 转化操作 | map() | 参数是函数，函数应用于RDD每一个元素，返回值是新的RDD |
| flatMap() | 参数是函数，函数应用于RDD每一个元素，将元素数据进行拆分，变成迭代器，返回值是新的RDD |
| filter() | 参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD |
| distinct() | 没有参数，将RDD里的元素进行去重操作 |
| union() | 参数是RDD，生成包含两个RDD所有元素的新RDD |
| intersection() | 参数是RDD，求出两个RDD的共同元素 |
| subtract() | 参数是RDD，将原RDD里和参数RDD里相同的元素去掉 |
| cartesian() | 参数是RDD，求两个RDD的笛卡儿积 |
| 行动操作 | collect() | 返回RDD所有元素 |
| count() | RDD里元素个数 |
| countByValue() | 各元素在RDD中出现次数 |
| reduce() | 并行整合所有RDD数据，例如求和操作 |
| fold(0)(func) | 和reduce功能一样，不过fold带有初始值 |
| aggregate(0)(seqOp,combop) | 和reduce功能一样，但是返回的RDD数据类型和原RDD不一样 |
| foreach(func) | 对RDD每个元素都是使用特定函数 |

## Spark Scala编程

学着用spark官方网站： spark.apache.org

Saprk入门MapReduce实例：<http://blog.csdn.net/u013851082/article/details/70142806>

1. **SparkWordCount单词统计 standalone 模式**

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

object SparkWordCount {

def FILE\_NAME:String = "word\_count\_results\_";

def main(args:Array[String]) {

if (args.length < 1) {

println("Usage:SparkWordCount FileName");

System.exit(1);

}

//创建Spark的配置对象SparkConf，设置Spark程序的运行时的配置信息

val conf = new SparkConf().setAppName("Spark Exercise: Spark Version Word Count Program");

//SparkContext是Spark程序所有功能的唯一入口

//SparkContext核心作用：初始化Spark应用程序运行所需要的核心组件，包括//DAGScheduler，TaskScheduler，Scheduler，同时还会负责Spark程序往Master注册程序

//SparkContext是整个Spark应用程序中最为至关重要的一个对象。

val sc = n ew SparkContext(conf);

val textFile = sc.textFile(args(0));

//根据具体的数据来源（HDFS，HBase，Local FS，DB）等 通过SparkContext来创建RDD

//RDD的创建基本有三种方式，根据外部的数据来源（例如HDFS），根据Scala集合，由其//它的RDD操作产生

//数据会被RDD划分成为一系列的Partitions，分配到每个Partition的数据属于一个Task的//处理范畴

val wordCounts = textFile.flatMap(line => line.split(" ")).map(

word => (word, 1)).reduceByKey((a, b) => a + b)

wordCounts.saveAsTextFile(FILE\_NAME+System.currentTimeMillis());

println("Word Count program running results are successfully saved.");

}

}

提交代码：

-------------------------

./spark-submit \

--class com.ibm.spark.exercise.basic.SparkWordCount \

--master spark://hadoop036166:7077 \

--num-executors 3 \

--driver-memory 6g --executor-memory 2g \

--executor-cores 2 \

/home/fams/sparkexercise.jar \

hdfs://hadoop036166:9000/user/fams/\*.txt

2.**SparkSQL实例 注册临时表 查询数据**

/\*\*

\* Created by PC-WeiFang on 2018/1/9.

\* DataFrame

\* 最核心的编程抽象.可以理解为是以列的形式存储的,分布式的数据集合.

\* 和关系型数据库很类似.

\* 可以通过多种数据源来构造.如结构化的数据文件,数据库中的表、hive的表、RDD等

\*/

import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

import org.apache.spark.sql.{Row, SQLContext, SparkSession}

import org.apache.spark.{SparkConf, SparkContext, sql}

object SparkSQLDemo {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setAppName("SparkSQLHiveDemo").setMaster("local[2]")

val sc=new SparkContext(conf)

val sqlContext=new SQLContext(sc)

//设置conf,配置AppName,运行的Master(这里设置为本地模式

//创建一个sc的SQLContext对象

//创建一个sqlcontext对象(也可以是SQLContext的子类对象,如 HiveContext)

//加载数据源

//val datas=sc.textFile("person.txt")

val datas=sc.textFile("/tmp/person.txt")

//RDD转换为DataFrame有两种方式:(这里使用了第二种)

//使用反射方式推断元数据

//使用编程接口来创建DataFrame.

val rowRDD=datas.map(line=>{

val stu=line.split("\t")

Row(stu(0).toInt,stu(1),stu(2))

//创建出元素为ROW的RDD

})

//流程简介:从原始的RDD创建一个元素为row的RDD;接下来创建一个structType,来代表ROW,最后将动态定义的

//元数据应用到RDD(ROW)上

val structType=StructType(Array(

//通过编程的方式动态的构造元数据

StructField("id",IntegerType,true),

StructField("name",StringType,true),

StructField("sex",StringType,true)

))

//通过sqlContext的createDataFrame方法,创建DataFrame,

// 将row类型的RDD和数据结构structType结合到一起

val stuDF=sqlContext.createDataFrame(rowRDD,structType)

//stuDF.show()

//show方法可以把里面的数据显示出来

stuDF.createOrReplaceTempView("stu")

//注册为临时表,这样就可以使用SQL语句了.

sqlContext.sql("select name from stu where sex='m'").show()

sc.stop()

}

}

Spark集群提交：

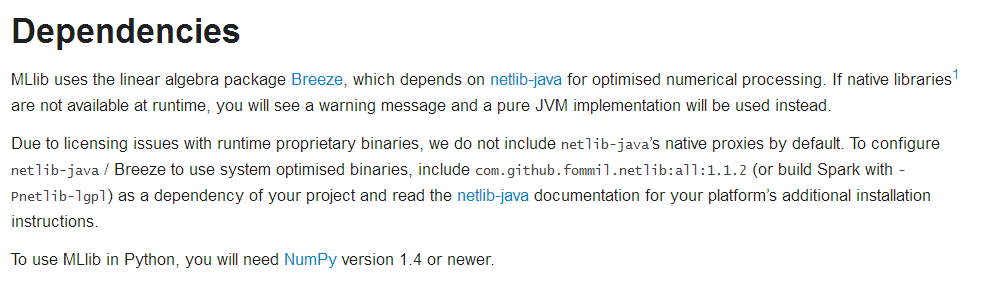
----------------------------------------

./spark-submit --class SparkScalaLocal.SparkSQLDemo /software/package/SparkScalaLocal.jar

## Spark MLlib应用

相关博客： <http://www.cnblogs.com/shishanyuan/p/4699644.html>

需要注意：spark-mllib依netlib-java 如下是官网说明 需要先添加依赖包



MLlib 是spark的可以扩展的机器学习库，由以下部分组成：通用的学习算法和工具类，包括分类，回归，聚类，协同过滤，降维等。

Spark之所以在机器学习方面具有得天独厚的优势，有以下几点原因：

（1）机器学习的计算需要在多次迭代后获得足够小的误差或者足够收敛才会停止，迭代时如果使用Hadoop的MapReduce计算框架，每次计算都要读/写磁盘以及任务的启动等工作，这回导致非常大的I/O和CPU消耗。而Spark基于内存的计算模型天生就擅长迭代计算，多个步骤计算直接在内存中完成，只有在必要时才会操作磁盘和网络。

（2）从通信的角度讲，如果使用Hadoop的MapReduce计算框架，JobTracker和TaskTracker之间由于是通过heartbeat的方式来进行的通信和传递数据，会导致非常慢的执行速度，而Spark具有出色而高效的Akka和Netty通信系统，通信效率极高。

**MLlib回归算法实例：**

1. 算法说明

线性回归是利用称为线性回归方程的函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析方法，只有一个自变量的情况称为简单回归，大于一个自变量情况的叫做多元回归，在实际情况中大多数都是多元回归。

线性回归（Linear Regression）问题属于监督学习（Supervised Learning）范畴，又称分类（Classification）或归纳学习（Inductive Learning）。这类分析中训练数据集中给出的数据类型是确定的。机器学习的目标是，对于给定的一个训练数据集，通过不断的分析和学习产生一个联系属性集合和类标集合的分类函数（Classification Function）或预测函数）Prediction Function），这个函数称为分类模型（Classification Model——或预测模型（Prediction Model）。通过学习得到的模型可以是一个决策树、规格集、贝叶斯模型或一个超平面。通过这个模型可以对输入对象的特征向量预测或对对象的类标进行分类。

回归问题中通常使用最小二乘（Least Squares）法来迭代最优的特征中每个属性的比重，通过损失函数（Loss Function）或错误函数（Error Function)定义来设置收敛状态，即作为梯度下降算法的逼近参数因子。

1. 实例介绍

该例子给出了如何导入训练集数据，将其解析为带标签点的RDD，然后使用了LinearRegressionWithSGD 算法来建立一个简单的线性模型来预测标签的值，最后计算了均方差来评估预测值与实际值的吻合度。

线性回归分析的整个过程可以简单描述为如下三个步骤：

（1）寻找合适的预测函数，即上文中的 h(x) ，用来预测输入数据的判断结果。这个过程是非常关键的，需要对数据有一定的了解或分析，知道或者猜测预测函数的“大概”形式，比如是线性函数还是非线性函数，若是非线性的则无法用线性回归来得出高质量的结果。

（2）构造一个Loss函数（损失函数），该函数表示预测的输出（h）与训练数据标签之间的偏差，可以是二者之间的差（h-y）或者是其他的形式（如平方差开方）。综合考虑所有训练数据的“损失”，将Loss求和或者求平均，记为 J(θ) 函数，表示所有训练数据预测值与实际类别的偏差。

（3）显然， J(θ) 函数的值越小表示预测函数越准确（即h函数越准确），所以这一步需要做的是找到 J(θ) 函数的最小值。找函数的最小值有不同的方法，Spark中采用的是梯度下降法（stochastic gradient descent，SGD)。

1. 程序代码

import org.apache.log4j.{Level, Logger}

import org.apache.spark.{SparkContext, SparkConf}

import org.apache.spark.mllib.regression.LinearRegressionWithSGD

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.linalg.Vectors

object LinearRegression {

def main(args:Array[String]): Unit ={

// 屏蔽不必要的日志显示终端上

Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

// 设置运行环境

val conf = new SparkConf().setAppName("Kmeans").setMaster("local[4]")

val sc = new SparkContext(conf)

// Load and parse the data

val data = sc.textFile("/home/hadoop/upload/class8/lpsa.data")

val parsedData = data.map { line =>

val parts = line.split(',')

LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(\_.toDouble)))

}

// Building the model

val numIterations = 100

val model = LinearRegressionWithSGD.train(parsedData, numIterations)

// Evaluate model on training examples and compute training error

val valuesAndPreds = parsedData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2)}.reduce (\_ + \_) / valuesAndPreds.count

println("training Mean Squared Error = " + MSE)

sc.stop()

}

}

**MLlib聚类算法实例：**

1. 算法说明

聚类（Cluster analysis）有时也被翻译为簇类，其核心任务是：将一组目标object划分为若干个簇，每个簇之间的object尽可能相似，簇与簇之间的object尽可能相异。聚类算法是机器学习（或者说是数据挖掘更合适）中重要的一部分，除了最为简单的K-Means聚类算法外，比较常见的还有层次法（CURE、CHAMELEON等）、网格算法（STING、WaveCluster等），等等。

较权威的聚类问题定义：所谓聚类问题，就是给定一个元素集合D，其中每个元素具有n个可观察属性，使用某种算法将D划分成k个子集，要求每个子集内部的元素之间相异度尽可能低，而不同子集的元素相异度尽可能高。其中每个子集叫做一个簇。

K-means聚类属于无监督学习，以往的回归、朴素贝叶斯、SVM等都是有类别标签y的，也就是说样例中已经给出了样例的分类。而聚类的样本中却没有给定y，只有特征x，比如假设宇宙中的星星可以表示成三维空间中的点集clip\_image002。聚类的目的是找到每个样本x潜在的类别y，并将同类别y的样本x放在一起。比如上面的星星，聚类后结果是一个个星团，星团里面的点相互距离比较近，星团间的星星距离就比较远了。

与分类不同，分类是示例式学习，要求分类前明确各个类别，并断言每个元素映射到一个类别。而聚类是观察式学习，在聚类前可以不知道类别甚至不给定类别数量，是无监督学习的一种。目前聚类广泛应用于统计学、生物学、数据库技术和市场营销等领域，相应的算法也非常多。

1. 实例介绍

在该实例中将介绍K-Means算法，K-Means属于基于平方误差的迭代重分配聚类算法，其核心思想十分简单：

随机选择K个中心点；

计算所有点到这K个中心点的距离，选择距离最近的中心点为其所在的簇；

简单地采用算术平均数（mean）来重新计算K个簇的中心；

重复步骤2和3，直至簇类不再发生变化或者达到最大迭代值；

输出结果。

K-Means算法的结果好坏依赖于对初始聚类中心的选择，容易陷入局部最优解，对K值的选择没有准则可依循，对异常数据较为敏感，只能处理数值属性的数据，聚类结构可能不平衡。

本实例中进行如下步骤：

1) 装载数据，数据以文本文件方式进行存放；

2) 将数据集聚类，设置2个类和20次迭代，进行模型训练形成数据模型；

3) 打印数据模型的中心点；

4) 使用误差平方之和来评估数据模型；

5) 使用模型测试单点数据；

6) 交叉评估1，返回结果；交叉评估2，返回数据集和结果。

1. 测试数据说明

在该文件中提供了6个点的空间位置坐标，使用K-means聚类对这些点进行分类。

使用的kmeans\_data.txt的数据如下所示：

0.0 0.0 0.0

0.1 0.1 0.1

0.2 0.2 0.2

9.0 9.0 9.0

9.1 9.1 9.1

9.2 9.2 9.2

1. 程序代码

import org.apache.log4j.{Level, Logger}

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.mllib.clustering.KMeans

import org.apache.spark.mllib.linalg.Vectors

object Kmeans {

def main(args: Array[String]) {

// 屏蔽不必要的日志显示在终端上

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

// 设置运行环境

val conf = new SparkConf().setAppName("Kmeans").setMaster("local[4]")

val sc = new SparkContext(conf)

// 装载数据集

val data = sc.textFile("/home/hadoop/upload/class8/kmeans\_data.txt", 1)

val parsedData = data.map(s => Vectors.dense(s.split(' ').map(\_.toDouble)))

// 将数据集聚类，2个类，20次迭代，进行模型训练形成数据模型

val numClusters = 2

val numIterations = 20

val model = KMeans.train(parsedData, numClusters, numIterations)

// 打印数据模型的中心点

println("Cluster centers:")

for (c <- model.clusterCenters) {

println(" " + c.toString)

}

// 使用误差平方之和来评估数据模型

val cost = model.computeCost(parsedData)

println("Within Set Sum of Squared Errors = " + cost)

// 使用模型测试单点数据

println("Vectors 0.2 0.2 0.2 is belongs to clusters:" + model.predict(Vectors.dense("0.2 0.2 0.2".split(' ').map(\_.toDouble))))

println("Vectors 0.25 0.25 0.25 is belongs to clusters:" + model.predict(Vectors.dense("0.25 0.25 0.25".split(' ').map(\_.toDouble))))

println("Vectors 8 8 8 is belongs to clusters:" + model.predict(Vectors.dense("8 8 8".split(' ').map(\_.toDouble))))

// 交叉评估1，只返回结果

val testdata = data.map(s => Vectors.dense(s.split(' ').map(\_.toDouble)))

val result1 = model.predict(testdata)

result1.saveAsTextFile("/home/hadoop/upload/class8/result\_kmeans1")

// 交叉评估2，返回数据集和结果

val result2 = data.map {

line =>

val linevectore = Vectors.dense(line.split(' ').map(\_.toDouble))

val prediction = model.predict(linevectore)

line + " " + prediction

}.saveAsTextFile("/home/hadoop/upload/class8/result\_kmeans2")

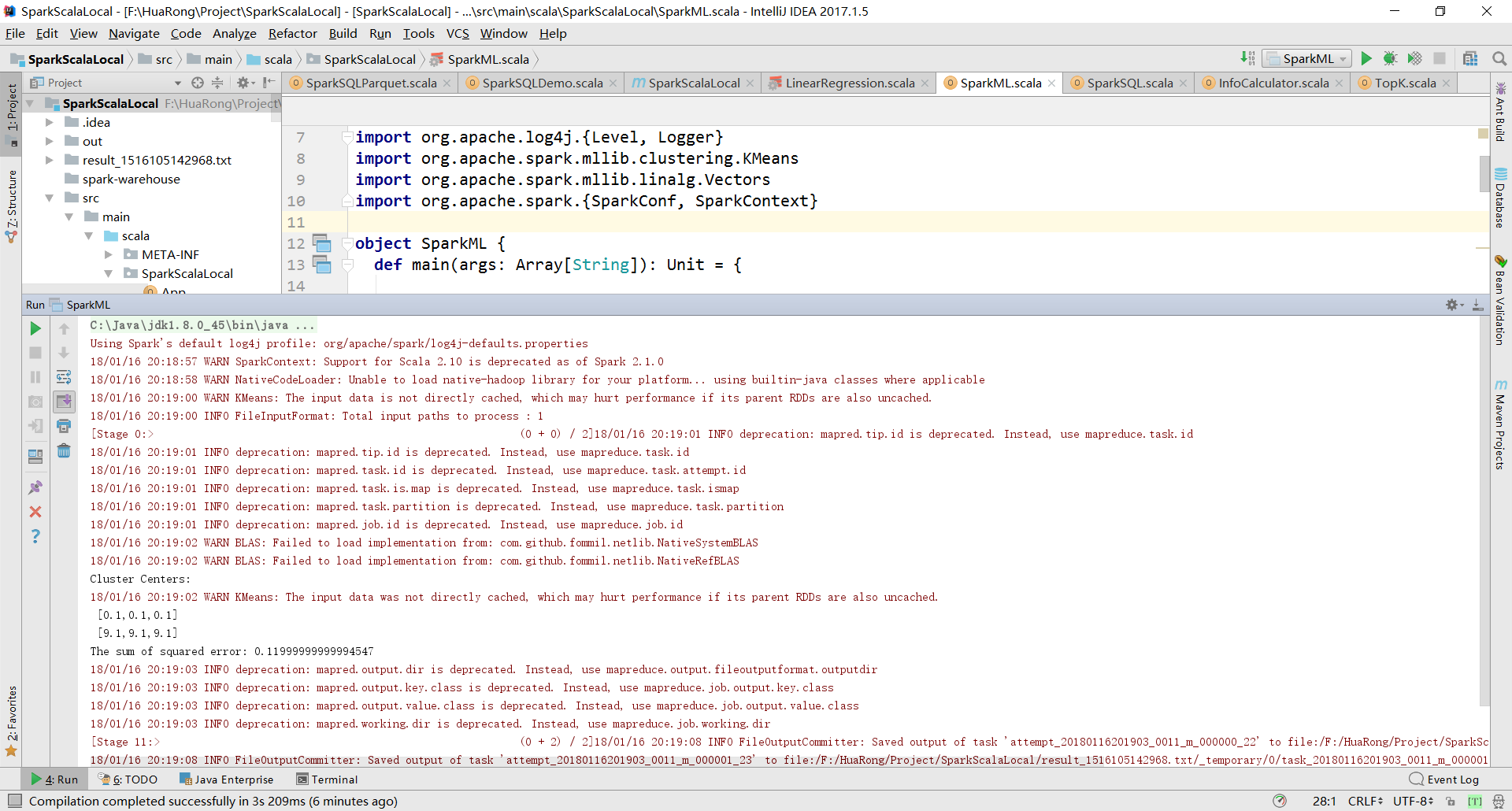
sc.stop()

}

}

在运行日志窗口中可以看到，通过计算计算出模型并找出两个簇中心点：(9.1，9.1，9.1)和(0.1，0.1，0.1)，使用模型对测试点进行分类求出分属于族簇。

结果如下图所示：



方 伟

2018/1/15

# 附录

## Flume经典案例配置

Flume相关配置博客：<http://blog.csdn.net/kumquatlemon/article/details/78528673>

Source：

avro端口：监听端口从外部客户端接收流数据

exec源：运行给定命令，期望改进程产生连续数据 cat 或者 tail

spooling目录：提取磁盘目录，监听指定目录

Channel：file、memory

Sink：HDFS、Hive、Hbase、Avro、Kafka

1. **source是spooldir，channel是memory，sink是hbase**

a1.sources = r1

a1.sinks = hbaseSink

a1.channels = memChannel

a1.sources.r1.type = spooldir

a1.sources.r1.spoolDir = /home/bms/flumeSpool

a1.sources.r1.fileHeader = true

a1.sources.r1.deletePolicy = immediate

##########the type of channel is memory#########

a1.channels.memChannel.type = memory

a1.channels.memChannel.capacity = 10000

a1.channels.memChannel.transactionCapacity = 1000

a1.channels.memChannel.byteCapacityBufferPercentage = 20

a1.channles.memChannel.byteCapacity = 8000

##########the type of sink is hbase#########

a1.sinks.hbaseSink.type = hbase

a1.sinks.hbaseSink.table=mytable

a1.sinks.hbaseSink.columnFamily=cf a1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.RegexHbaseEventSerializer

a1.sinks.hbaseSink.zookeeperQuorum=flumekafka1，flumekafka2，flumekafka3

##########the type of sink is hbase#########

a1.sources.r1.channels = memChannel

a1.sinks.k1.channel = memChannel

1. **source是spooldir，channel是memory，sink是hdfs**

a1.sources = r1

a1.sinks = hdfsSink

a1.channels = memChannel

a1.sources.r1.type = spooldir

a1.sources.r1.spoolDir = /home/bms/flumeSpool

a1.sources.r1.fileHeader = true

a1.sources.r1.deletePolicy = immediate

##########the type of channel is memory#########

a1.channels.memChannel.type = memory

a1.channels.memChannel.capacity = 10000

a1.channels.memChannel.transactionCapacity = 1000

a1.channels.memChannel.byteCapacityBufferPercentage = 20

a1.channles.memChannel.byteCapacity = 80000

##########the type of sink is hdfs#########

a1.sinks.hdfsSink.type=hdfs

a1.sinks.hdfsSink.hdfs.path=hdfs://master:9000/flume1

a1.sinks.hdfsSink.hdfs.filePrefix=lwq

a1.sinks.sink1.hdfs.round = true

a1.sinks.hdfsSink.hdfs.rollInterval = 36

a1.sinks.hdfsSink.hdfs.rollSize = 128000000

a1.sinks.hdfsSink.hdfs.rollCount = 0

a1.sinks.hdfsSink.hdfs.batchSize = 1000

a1.sinks.hdfsSink.hdfs.roundValue = 1

a1.sinks.hdfsSink.hdfs.roundUnit = minute

a1.sinks.hdfsSink.hdfs.useLocalTimeStamp = true

a1.sinks.hdfsSink.hdfs.fileType = DataStream

a1.sinks.hdfsSink.hdfs.writeFormat = Text

##########the type of sink is hdfs#########

a1.sources.r1.channels = memChannel

a1.sinks.k1.channel = memChannel

1. **source是spooldir，channel是memory，sink是kafka**

a1.sources = r1

a1.sinks = kafkaSink

a1.channels = memChannel

a1.sources.r1.type = spooldir

a1.sources.r1.spoolDir = /home/bms/flumeSpool

a1.sources.r1.fileHeader = true

a1.sources.r1.deletePolicy = immediate

##########the type of channel is memory#########

a1.channels.memChannel.type = memory

a1.channels.memChannel.capacity = 10000

a1.channels.memChannel.transactionCapacity = 1000

a1.channels.memChannel.byteCapacityBufferPercentage = 20

a1.channles.memChannel.byteCapacity = 80000

##########the type of sink is kafka#########

a1.sinks.kafkaSink.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.kafkaSink.topic=flume1 a1.sinks.kafkaSink.brokerList=flumekafka1:9092,flumekafka2:9092,flumekafka3:9092

a1.sinks.kafkaSink.requiredAcks=1

a1.sinks.kafkaSink.batchSize = 20

##########the type of sink is kafka#########

a1.sources.r1.channels = memChannel

a1.sinks.kafkaSink.channel = memChannel

1. **source是spooldir、channel是memory，sink是kafka和hdfs**

a1.sources = r1

a1.sinks = kafkaSink hdfsSink

a1.channels = kafkaChannel hdfsChannel

a1.sources.r1.type = spooldir

a1.sources.r1.spoolDir = /home/bms/flumeSpool

a1.sources.r1.fileHeader = true

a1.sources.r1.deletePolicy = immediate

##########the type of channel is kafka#########

a1.channels.kafkaChannel .type = memory

a1.channels.kafkaChannel .capacity = 10000

a1.channels.kafkaChannel .transactionCapacity = 1000

a1.channels.kafkaChannel .byteCapacityBufferPercentage = 20

a1.channles.kafkaChannel .byteCapacity = 80000

##########the type of channel is kafka#########

##########the type of channel is hdfs#########

a1.channels.hdfsChannel.type = memory

a1.channels.hdfsChannel.capacity = 10000

a1.channels.hdfsChannel.transactionCapacity = 1000

a1.channels.hdfsChannel.byteCapacityBufferPercentage = 20

a1.channles.hdfsChannel.byteCapacity = 80000

##########the type of channel is hdfs#########

##########the type of sink is kafka#########

a1.sinks.kafkaSink .type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.kafkaSink .topic=flume1

a1.sinks.kafkaSink .brokerList=flumekafka1:9092,flumekafka2:9092,flumekafka3:9092

a1.sinks.kafkaSink .requiredAcks=1

a1.sinks.kafkaSink .batchSize = 20

##########the type of sink is kafka#########

a1.sinks.hdfsSink.type=hdfs

a1.sinks.hdfsSink.hdfs.path=hdfs://master:9000/flume1

a1.sinks.hdfsSink.hdfs.filePrefix=lwq

a1.sinks.sink1.hdfs.round = true

a1.sinks.hdfsSink.hdfs.rollInterval = 36

a1.sinks.hdfsSink.hdfs.rollSize = 128000000

a1.sinks.hdfsSink.hdfs.rollCount = 0

a1.sinks.hdfsSink.hdfs.batchSize = 1000

a1.sinks.hdfsSink.hdfs.roundValue = 1

a1.sinks.hdfsSink.hdfs.roundUnit = minute

a1.sinks.hdfsSink.hdfs.useLocalTimeStamp = true

a1.sinks.hdfsSink.hdfs.fileType = DataStream

a1.sinks.hdfsSink.hdfs.writeFormat = Text

a1.sources.r1.channels = kafkaChannel hdfsChannel

a1.sinks.kafkaSink .channel = kafkaChannel

a1.sinks.hdfsSink.channel = hdfsChannel

a1.sources.r1.selector.type = replicating

## Kafka配置详解

**Broker相关配置：**

##broker标示，id为正数，且全局不得重复

broker.id=1

##日志文件存储位置

log.dir=/kafka/logs

##broker需要用zookeeper保存meta信息

zookeeper.connect=192.168.100.104:9092, 192.168.100.105:9092,192.168.100.106:9092

##用来监听连接的端口，生产者和消费者将在这个端口上建立连接

port=9092

##指定broker实例绑定的网络接口地址

host.name=192.168.100.104

##每个partition的备份个数，默认为1，应根据实际环境配置

##此值大意味着向各个broker上同步消息时，延时较大

num.partitions=2

##日志文件中每个segment的大小，默认是1G

log.segment.bytes=1024\*1024\*1024

##滚动生成新的segment文件的时间

log.roll.hours=7\*24

##segment文件将被保留的最大时间，超时间将被删除

log.retention.hours=7\*24

##partition的buffer中消息的条数，达到阈值，将被flush到磁盘

log.flush.interval.messages=10000

##消息buffer的时间，达到阈值，将被flush到磁盘

log.flush.interval.ms=3000

##partition的leader等待follower同步消息的时间，如果超时，leader将follower从同步列表中删除

replica.lag.tine.max.ms=10000

##允许follower落后消息的最大条数，如果达到阈值，leader将follower从同步列表

replica.lag.tine.max。message=4000

##消息备份的个数，默认是1

num.replica.fetchers=1

**Producer的相关配置：**

##对于开发人员来说，需要通过broker.list来指定producer需要关注的broker列表，producer通过与每个broker建立连接来获取并获取partitions，如果某个broker连接建立失败，将导致以上的partitions无法继续发布消息，格式是：host1:port,host2:port

metadata.broker.list=192.168.100.104:9092,192.168.100.105:9092,192.168.100.106:9092

##producer接收消息ack的机制，默认为0

##0：producer不会等待broker的ack

##1：当leader收到消息后发送ack

##2：所有的follower都接到消息后，发送ack

request.required.ack=1

##producer发送消息的模式，同步或者是异步

##异步意味着消息在本地buffer，并适时的批量发送

##默认是async，建议用async

producer.type=sync

##消息序列化类，将消息转化成byte[]

Serializer.class=kafka.serializer.DefaultEncoder

##消息路由类，消息在发送时，将根据此实例的方法获取partition的索引号

partitioner.class=kafka.prducer.DefaultPartitioner

##消息压缩方法none、gzip、snappy

compression.codec=none

##消息在producer端buffer的条数，仅在producer.type=async时有效

batch.num.messages=200

**Consummer的相关配置：**

##当前消费者的group名称，需要指定

Group.id=test-consummer-group

##consummer作为zookeeperclient，需要通过zk保存一些meta信息，此处为zk connectString

zookeeper.connect=hostname1:port, hostname2:port,hostname3:port

##当前consummer的标识，可以设定，也可以由系统生成

consumer.id=12

##获取消息的最大尺寸，broker不会向consummer输出大于此值的消息chuck

##每次fetch将得到多条消息，此值为总大小

fetch.messages.max.bytes=1024\*1024

##当consummer消费一定量消息之后，将会自动向zookeeper提交offset信息

##注意offset消息并不是每消费依稀消息，就像zk提交一次，而是先在本地保存，并定期提交

auto.commit.enable=true

##自动提交时间间隔，默认是1分钟

auto.commit.enable=60\*1000

## SparkStreaming性能调优

**优化运行时间：**

* 增加并行度。确保使用整个集群的资源，而不是把任务集中在几个特定的节点上。对于包含shuffle的操作，增加其并行度以确保更为充分地使用集群资源。
* 减少数据序列化、反序列化的负担。Spark Streaming默认将接收到的数据序列化后存储以减少内存的使用。但序列化和反序列化需要更多的CPU时间，因此更加高效的序列化方式（Kryo）和自定义的序列化接口可以更高效地使用CPU。
* 设置合理的batch窗口。在Spark Streaming中，Job之间有可能存在着依赖关系，后面的Job必须确保前面的Job执行结束后才能提交。若前面的Job执行时间超出了设置的batch窗口，那么后面的Job就无法按时提交，这样就会进一步拖延接下来的Job，造成后续Job的阻塞。因此，设置一个合理的batch窗口确保Job能够在这个batch窗口中结束是必须的。
* 减少任务提交和分发所带来的负担。通常情况下Akka框架能够高效地确保任务及时分发，但当batch窗口非常小（500ms）时，提交和分发任务的延迟就变得不可接受了。使用Standalone模式和Coarse-grained Mesos模式通常会比使用Fine-Grained Mesos模式有更小的延迟。

**优化内存使用：**

* 控制batch size。Spark Streaming会把batch窗口内接收到的所有数据存放在Spark内部的可用内存区域中，因此必须确保当前节点Spark的可用内存至少能够容纳这个batch窗口内所有的数据，否则必须增加新的资源以提高集群的处理能力。
* 及时清理不再使用的数据。上面说到Spark Streaming会将接收到的数据全部存储于内部的可用内存区域中，因此对于处理过的不再需要的数据应及时清理以确保Spark Streaming有富余的可用内存空间。通过设置合理的spark.cleaner.ttl时长来及时清理超时的无用数据。
* 观察及适当调整GC策略。GC会影响Job的正常运行，延长Job的执行时间，引起一系列不可预料的问题。观察GC的运行情况，采取不同的GC策略以进一步减小内存回收对Job运行的影响。

## Redis安装部署

**Redis简介**：

Redis是当前比较热门的NOSQL系统之一，它是一个key-value存储系统。和Memcache类似，但很大程度补偿了Memcache的不足，它支持存储的value类型相对更多，包括string、list、set、zset和hash。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作。在此基础上，Redis支持各种不同方式的排序。

和Memcache一样，Redis数据都是缓存在计算机内存中，不同的是，Memcache只能将数据缓存到内存中，无法自动定期写入硬盘，这就表示，一断电或重启，内存清空，数据丢失。所以Memcache的应用场景适用于缓存无需持久化的数据。而Redis不同的是它会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，实现数据的持久化。

**Redis安装**：

首先上官网下载Redis 压缩包 地址：http://redis.io/download下载稳定版4.0.6

通过远程管理工具，将压缩包拷贝到Linux服务器中，执行解压操作：

tar -zxvf redis-4.0.6.tar.gz

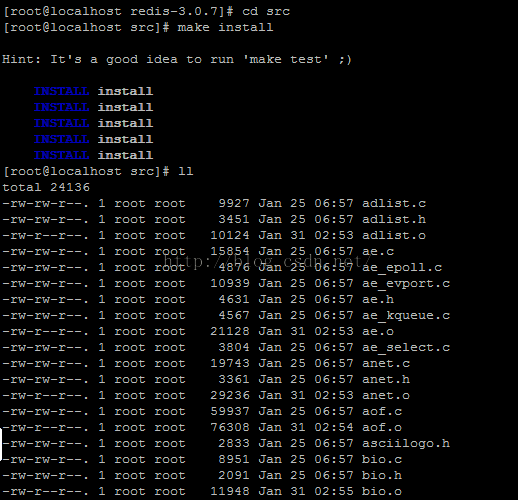
执行make 对Redis解压后文件进行编译：

cd redis-4.0.6

make

编译成功后，进入src文件夹，执行make install进行Redis安装

安装完成结果如下图所示：

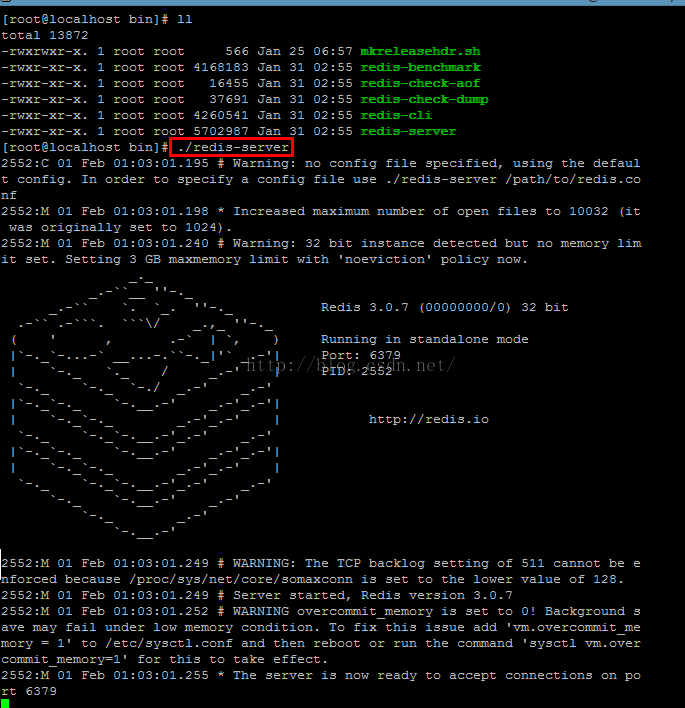


**Redis部署**：

执行Redis-server 命令 启动Redis 服务:

cd src/

./redis-server



注意：这里直接执行Redis-server 启动的Redis服务，是在前台直接运行的(效果如上图)，也就是说，执行完该命令后，如果Lunix关闭当前会话，则Redis服务也随即关闭。正常情况下，启动Redis服务需要从后台启动，并且指定启动配置文件。

后台启动redis服务：

a)首先编辑conf文件，将daemonize属性改为yes（表明需要在后台运行）

vim redis.conf

b)再次启动redis服务，并指定启动服务配置文件

src/redis-server redis.conf

c)执行redis-cli启动Redis 客户端

d) redis-cli shutdown 关闭redis客户端

e) Netstat -tunpl|grep 6379 查看redis 默认端口号6379占用情况

修改profile文件：

# vi /etc/profile

在最后行添加:

export PATH="$PATH: /software/redis-4.0.6/src"

然后马上应用这个文件：

# source /etc/profile

配置下面的内核参数，否则Redis脚本在重启或停止redis时，将会报错，并且不能自动在停止服务前同步数据到磁盘上/etc/sysctl.conf加上

#vim /etc/sysctl.conf

vm.overcommit\_memory = 1

#sysctl -p

## RocketMQ介绍

RcoketMQ 是一款低延迟、高可靠、可伸缩、易于使用的消息中间件。

具有以下特性：

1）支持发布/订阅（Pub/Sub）和点对点（P2P）消息模型；

2）在一个队列中可靠的先进先出（FIFO）和严格的顺序传递

3）支持拉（pull）和推（push）两种消息模式

4）单一队列百万消息的堆积能力

5）支持多种消息协议，如 JMS、MQTT 等

6）分布式高可用的部署架构,满足至少一次消息传递语义

7）提供 docker 镜像用于隔离测试和云集群部署

8）提供配置、指标和监控等功能丰富的 Dashboard

**专业术语：**

Producer：

消息生产者，生产者的作用就是将消息发送到 MQ，生产者本身既可以产生消息，如读取文本信息等。也可以对外提供接口，由外部应用来调用接口，再由生产者将收到的消息发送到 MQ。

Producer Group：

生产者组，简单来说就是多个发送同一类消息的生产者称之为一个生产者组。在这里可以不用关心，只要知道有这么一个概念即可。

Consumer：

消息消费者，简单来说，消费 MQ 上的消息的应用程序就是消费者，至于消息是否进行逻辑处理，还是直接存储到数据库等取决于业务需要。

Consumer Group：

消费者组，和生产者类似，消费同一类消息的多个 consumer 实例组成一个消费者组。

Topic：

Topic 是一种消息的逻辑分类，比如说你有订单类的消息，也有库存类的消息，那么就需要进行分类，一个是订单 Topic 存放订单相关的消息，一个是库存 Topic 存储库存相关的消息。

Message：

Message 是消息的载体。一个 Message 必须指定 topic，相当于寄信的地址。Message 还有一个可选的 tag 设置，以便消费端可以基于 tag 进行过滤消息。也可以添加额外的键值对，例如你需要一个业务 key 来查找 broker 上的消息，方便在开发过程中诊断问题。

Tag：

标签可以被认为是对 Topic 进一步细化。一般在相同业务模块中通过引入标签来标记不同用途的消息。

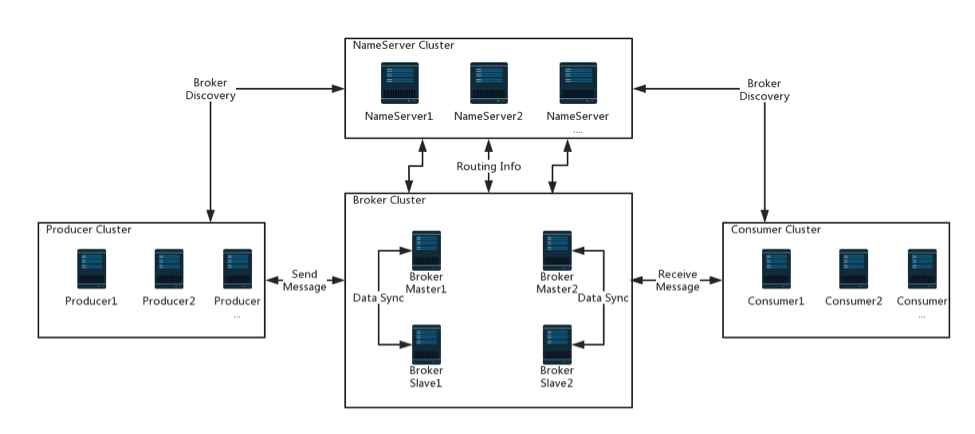
Broker：

Broker 是 RocketMQ 系统的主要角色，其实就是前面一直说的 MQ。Broker 接收来自生产者的消息，储存以及为消费者拉取消息的请求做好准备。

Name Server：

Name Server 为 producer 和 consumer 提供路由信息。

**RocketMQ 架构：**

****

由这张图可以看到有四个集群，分别是 NameServer 集群、Broker 集群、Producer 集群和 Consumer 集群：

NameServer: 提供轻量级的服务发现和路由。 每个 NameServer 记录完整的路由信息，提供等效的读写服务，并支持快速存储扩展。

Broker: 通过提供轻量级的 Topic 和 Queue 机制来处理消息存储,同时支持推（push）和拉（pull）模式以及主从结构的容错机制。

Producer：生产者，产生消息的实例，拥有相同 Producer Group 的 Producer 组成一个集群。

Consumer：消费者，接收消息进行消费的实例，拥有相同 Consumer Group 的Consumer 组成一个集群。

## ElasticSearch集群搭建

ElasticSearch是一个开源的高扩展的分布式全文检索引擎，它可以近乎实时的存储、检索数据；本身扩展性很好，可以扩展到上百台服务器，处理PB级别的数据。

Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。

几乎每个系统都会有一个搜索的功能，当搜索做到一定程度时，维护和扩展起来难度就会慢慢变大，所以很多公司都会把搜索单独独立出一个模块，用ElasticSearch等来实现。

Elasticsearch还可用来做日志分析系统：ELK。是ElasticSearch + Logstash +Kibana 的简称。L:是一个完全开源的工具，它可以对你的日志进行收集、分析，并将其存储供以后使用（如，搜索）。 kibana 也是一个开源和免费的工具，他 Kibana 可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以帮助您汇总、分析和搜索重要数据日志。

**集群安装**：

1. 软件版本

操作系统：CentOS-6.5-x86\_64

ES版本：6.1.2

主机：10.1.29.128

主机: 10.1.29.133

主机：10.1.29.134

1. 部署环境

jdk版本：open-jdk.1.8

# java -version

openjdk version "1.8.0\_101"

OpenJDK Runtime Environment (build 1.8.0\_101-b13)

OpenJDK 64-Bit Server VM (build 25.101-b13, mixed mode)

下载解压安装即可：

# tar zxvf elasticsearch-6.0.4.tar.gz

修改配置：

# cd config/

# vim elasticsearch.yml

配置解析：

集群：

cluster.name: elasticsearch

节点：

node.name: bigdata1 ##节点保持唯一性

数据和日志存放目录：

#path.data: /path/to/data

#path.logs: /path/to/logs ##默认当前目录下，可以修改

是否使用swap:

bootstrap.memory\_lock:true

bootstrap.system\_call\_filter: false

广播配置：

discovery.zen.ping.unicast.hosts: ["bigdata1", " bigdata2",” bigdata3”]

discovery.zen.minimum\_master\_nodes: 1

**其它启动elasticsearch可能遇到的问题**：

因为安全问题elasticsearch 不让用root用户直接运行，所以要创建新用户

建议创建一个单独的用户用来运行ElasticSearch

创建elsearch用户组及elsearch用户

groupadd elsearch

useradd elsearch -g elsearch -p elsearch

然后切换elsearch用户启动bin/elasticsearch

Centos6不支持SecComp，而ES5.2.0默认bootstrap.system\_call\_filter为true

禁用：在elasticsearch.yml中配置bootstrap.system\_call\_filter为false，注意要在Memory下面:

bootstrap.memory\_lock: false

bootstrap.system\_call\_filter: false

切换到root用户：

vi /etc/security/limits.conf

添加如下内容:

\* soft nofile 65536

\* hard nofile 131072

\* soft nproc 40960

\* hard nproc 40960

切换到root用户：

vim /etc/security/limits.d/90-nproc.conf

修改如下内容：

\* soft nproc 1024

#修改为

\* soft nproc 4096

vim /etc/sysctl.conf

添加下面配置：

vm.max\_map\_count=655360

并执行命令：

sysctl -p

然后，重新启动elasticsearch，即可启动成功。

启动之后：测试有如下显示表示已经安装成功

# curl -XGET '10.1.29.128:9200'

{

"name" : "log-1",

"cluster\_name" : "es-log",

"cluster\_uuid" : "\_na\_",

"version" : {

"number" : "5.0.0",

"build\_hash" : "253032b",

"build\_date" : "2016-10-26T04:37:51.531Z",

"build\_snapshot" : false,

"lucene\_version" : "6.2.0"

},

"tagline" : "You Know, for Search"

}

## MongoDB分布式集群

Mongo DB是目前在IT行业非常流行的一种非关系型数据库(NoSql),其灵活的数据存储方式,备受当前IT从业人员的青睐。Mongo DB很好的实现了面向对象的思想(OO思想),在Mongo DB中 每一条记录都是一个Document对象。Mongo DB最大的优势在于所有的数据持久操作都无需开发人员手动编写SQL语句,直接调用方法就可以轻松的实现CRUD操作。

单机安装配置：

1. 下载mongodb

前往mongodb官网下载页面：[https://www.mongodb.org/downloads](http://zhangge.net/go/?url=https://www.mongodb.org/downloads)下载相应的版本，比如目前的Linux x64位最新版：mongodb-linux-x86\_64-2.6.4.tgz

1. 解压安装

# tar -zxvf mongodb-linux-x86\_64-2.6.4.gz

指定mongodb的data目录 默认/data/db

#mkdir -r /data/db

1. 启动mongodb 默认端口27017

# /software/mongodb/bin/mongod

启动客户端 进入shell访问数据库

# /software/mongodb/bin/mongo

1. mongodb数据库基本操作

show dbs:显示数据库列表

show users：显示用户

db.help()：显示数据库操作命令，里面有很多的命令

use db：进入db数据库，如果没有自动创建

db.dropDatabase()：删除当前使用数据库

db.userInfo.find()：查询userInfo中的所有数据

db.userInfo.find().limit(5)：查询前5条数据

db.userInfo.find({"age": 22})：查询age = 22的记录

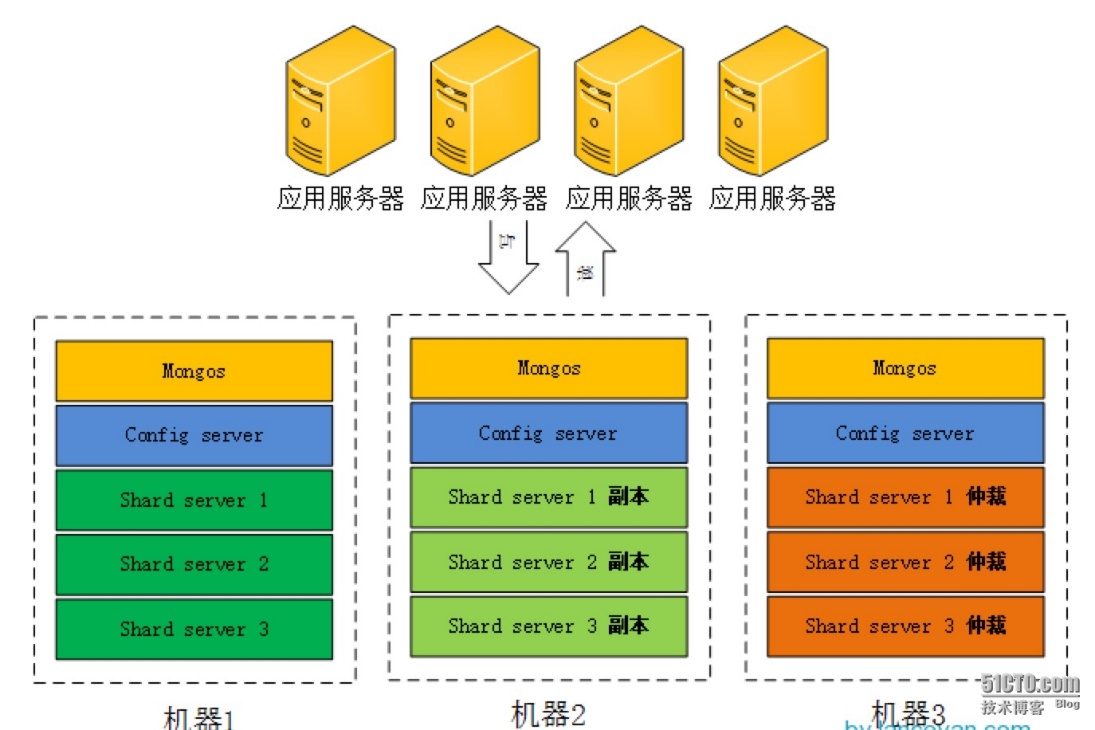
Mongodb集群：

mongodb集群有三种方式

1主从模式，类似mysql master slave方式。

2副本集模式，其实是一主多从，如果主节点挂掉，会重新在从节点选取一台为主节点。

3分片模式，针对大数据量，高负载情况。



**四个组件：mongos、config server、shard、replica set：**

mongos：数据库集群请求的入口，所有的请求都通过mongos进行协调，不需要在应用程序添加一个路由选择器，mongos自己就是一个请求分发中心，它负责把对应的数据请求请求转发到对应的shard服务器上。在生产环境通常有多mongos作为请求的入口，防止其中一个挂掉所有的mongodb请求都没有办法操作。

config server：顾名思义为配置服务器，存储所有数据库元信息（路由、分片）的配置。mongos本身没有物理存储分片服务器和数据路由信息，只是缓存在内存里，配置服务器则实际存储这些数据。mongos第一次启动或者关掉重启就会从 config server 加载配置信息，以后如果配置服务器信息变化会通知到所有的 mongos 更新自己的状态，这样 mongos 就能继续准确路由。在生产环境通常有多个 config server 配置服务器，因为它存储了分片路由的元数据，这个可不能丢失！就算挂掉其中一台，只要还有存货， mongodb集群就不会挂掉。

shard：这就是传说中的分片了。不要光想到存储空间，实际运行的数据库还有硬盘的读写、网络的IO、CPU和内存的瓶颈。在mongodb集群只要设置好了分片规则，通过mongos操作数据库就能自动把对应的数据操作请求转发到对应的分片机器上。

replica set：如果没有 replica set 是个不完整架构，假设其中的一个分片挂掉那四分之一的数据就丢失了，所以在高可用性的分片架构还需要对于每一个分片构建 replica set 副本集保证分片的可靠性。生产环境通常是 2个副本 + 1个仲裁。

## Kettle定时数据抽取

Crontab：

crontab命令常见于Unix和类Unix的操作系统之中，用于设置周期性被执行的指令。该命令从标准输入设备读取指令，并将其存放于“crontab”文件中，以供之后读取和执行。crontab储存的指令被守护进程激活，crond常常在后台运行，每一分钟检查是否有预定的作业需要执行。

**Linux下使用crond定时执行kettle的job**：

1. 服务器部署kettle

将kettle编译包上传到服务器上 使用5.1.0稳定版本

赋予/kettle/data-integration目录下.sh文件的执行权力

chmod +x \*.sh

进入目录 键入“./kitchen.sh -version”回车，如果有版本信息则表示kettle环境部署成功

1. 编写执行kettle任务的shell脚本

export JAVA\_HOME=/usr/lib/jdk1.7.0\_79

export PATH=$JAVA\_HOME/bin:$PATH

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

/app/kettle/data-integration/kitchen.sh -file=/app/kettle/jobs/test.kjb > /app/kettle/jobs/result.txt

注意：前3行设置java环境变量的语句必不可少

因为crond是个守护进程，它不归属于任何用户，虽然之前以root身份配置了java的环境变量，但是crond一样找不到java命令。所以，当crond执行kettle任务时，需要动态设置java环境变量，crond才能找到java命令。

1. 使用crontab命令添加定时任务

在终端上，键入“crontab -e”，进入定时任务文件

键入“a”，对文件进行编辑

输入如下内容：

01 00 10 02 \* /app/kettle/data-integration/test.sh

其中 01为分钟，00为小时，10为日，02为月，\*为星期几

键盘敲击exc，终端键入“:wq”，保存文件

重启crond服务

键入“cd /etc/init.d”，进入该目录

键入“./crond restart”，重启crond服务

等待执行结果，正确启动crond。

1. Kettle生成抽取表数据生成kjb文件

运行Spoon.bat 连接服务器Orcale数据库：

