# 项目练习- MapReduce与Spark内存计算的性能对比[¶](#项目练习--MapReduce与Spark内存计算的性能对比)

## 实验介绍[¶](#实验介绍)

#### 1.实验内容[¶](#1.实验内容)

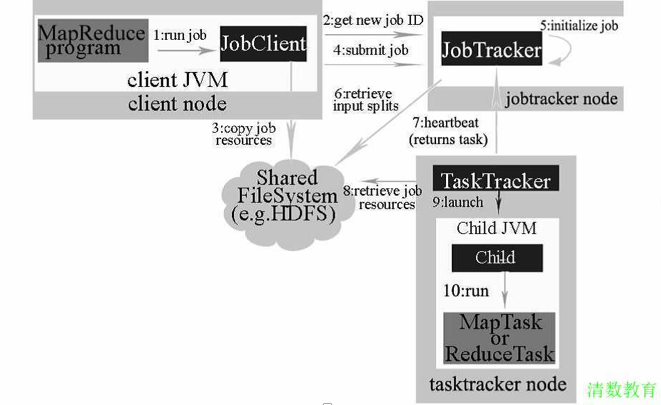
本节课程我们使用Hadoop MapReduce、Spark框架分别运行wordcount分析程序，来对MapReduce和Spark的性能进行对比。

#### 2.概述[¶](#2.概述)

（1）Hadoop MapReduce

Hadoop 的设计思路来源于 Google 的 GFS 和 MapReduce。它是一个开源软件框架，通过在集群计算机中使用简单的编程模型，可编写和运行分布式应用程序处理大规模数据。

第一代 Hadoop MapReduce 是一个在计算机集群上分布式处理海量数据集的软件框架，包括一个 JobTracker 和一定数量的 TaskTracker。运行流程图如下图所示。



在最上层有 4 个独立的实体，即客户端、JobTracker、TaskTracker 和分布式文件系统。客户端提交 MapReduce 作业；JobTracker 协调作业的运行；JobTracker 是一个 Java 应用程序，它的主类是 JobTracker；TaskTracker 运行作业划分后的任务，TaskTracker 也是一个 Java 应用程序，它的主类是 TaskTracker。Hadoop 运行 MapReduce 作业的步骤主要包括提交作业、初始化作业、分配任务、执行任务、更新进度和状态、完成作业等 6 个步骤。

（2）Spark

Spark 是一个基于内存计算的开源的集群计算系统，目的是让数据分析更加快速。Spark 提供了基于内存的计算集群，在分析数据时将数据导入内存以实现快速查询，“速度比”基于磁盘的系统，如比 Hadoop 快很多。Spark 最初是为了处理迭代算法，如机器学习、图挖掘算法等，以及交互式数据挖掘算法而开发的。在这两种场景下，Spark 的运行速度可以达到 Hadoop 的几百倍。

#### 3.数据集介绍[¶](#3.数据集介绍)

本实验的数据集是基于实验平台上传至共享目录：/mnt/nfsdata/xiniulab/lab/data/tools/project/MapReduce-Spark-word的word.txt文件。

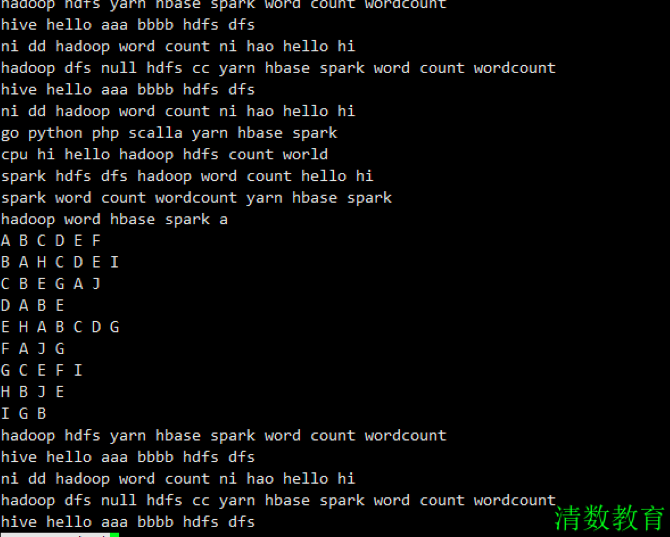
（1）数据集的大小，如下图：



（2）数据集的字符串数：



（3）字符集的内容



## 实验要求[¶](#实验要求)

#### 1. Java （需要安装1.6.x及其以上版本)[¶](#1.-Java-（需要安装1.6.x及其以上版本))

在终端输入 java -version 来查看Java 版本，这里我们使用的是1.7版本；



#### 2. Hadoop和Spark[¶](#2.-Hadoop和Spark)

在这里，我们使用了Hadoop V2.6.0版本以及Spark-2.0.1版本。

## 实验步骤[¶](#实验步骤)

#### 1.实验前准备[¶](#1.实验前准备)

$ su - hadoop

口令输入：hadoop

$ bash

$ echo $HADOOP\_HOME

/hadoop/hadoop

上述输出确认hadoop的环境变量设置有效，如果无效则激活环境变量：

$ source ~/.bash\_profile

启动ssh,口令输入：hadoop

hadoop@357987c120a9:~$ sudo service ssh start

[sudo] password for hadoop:

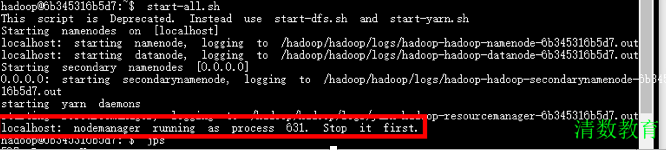
[ ok ] Starting OpenBSD Secure Shell server: sshd.

hadoop@357987c120a9:~$

#### 2.启动hadoop[¶](#2.启动hadoop)

启动命令为：

$ start-all.sh



最后一行“localhost: nodemanager running as process 631. Stop it first.”，是因为之前已经开启了这些服务，正在运行，再开启就提示先关闭。

检查是否运行成功

#执行jps命令可以查看到hadoop的几个主要进程:

$ jps

Jps

ResourceManager

NameNode

SecondaryNameNode

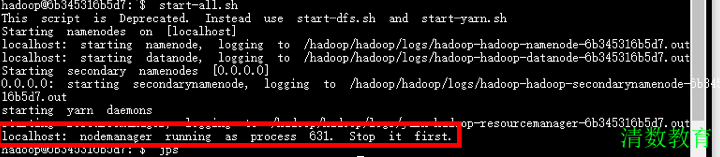
DataNode

即证明NodeManager可能未成功启动，可以将hadoop关掉重启一次：

$ stop-all.sh

$ start-all.sh

如下图，即为启动成功



$ jps

NameNode

Jps

NodeManager

ResourceManager

DataNode

SecondaryNameNode

#### 3.启动spark[¶](#3.启动spark)

①首先启动master

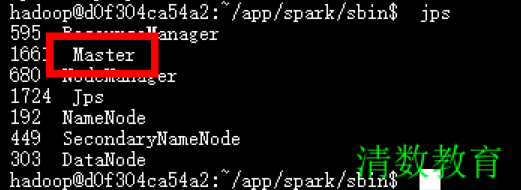
$ cd /hadoop/app/spark/sbin/

$ ./start-master.sh



查看Master进程是否启动

$ jps



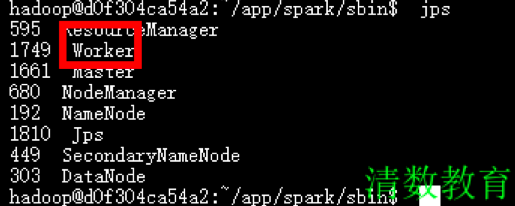
②启动slave

$ ./start-slave.sh spark://127.0.0.1:7077



查看Woker进程是否启动

$ jps



#### 4.将本次实验的数据文件上传到HDFS文件系统[¶](#4.将本次实验的数据文件上传到HDFS文件系统)

该数据集我们已经放在了共享目录/home/ds/data/tools/project/MapReduce-Spark-word下，我们可以建立一个/hadoop/data目录来看一下该文件夹下的个文件大小等信息。

$ mkdir /hadoop/data

$ cp -r /home/ds/data/tools/project/MapReduce-Spark-word/word.txt /hadoop/data

$ cd /hadoop/data

查看该目录下是否有了word.txt文件

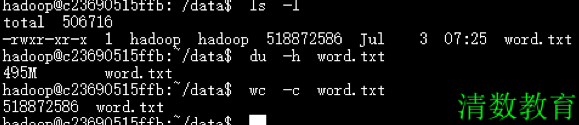
$ ls -l

查看数据集的大小：

$ du -h word.txt

查看数据集的字符串数：

$ wc -c word.txt



查看字符集的内容：

$ more word.txt

more命令说明：

①more命令会以一页一页的显示方便使用者逐页阅读，而最基本的指令就是按空白键（space）就往下一页显示，按 b 键就会往回（back）一页显示，而且还有搜寻字串的功能 。more命令从前向后读取文件，因此在启动时就加载整个文件。

②空格键 向下滚动一屏

③q 退出more

因此，向下一页按“空格键”；退出按q

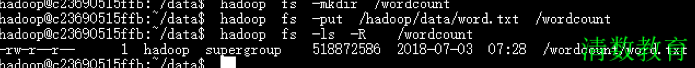


将文件上传到HDFS/wordcount

$ hadoop fs -mkdir /wordcount

$ hadoop fs -put /hadoop/data/word.txt /wordcount

$ hadoop fs -ls -R /wordcount



#### 5.MapReduce实现WordCount实例（Python）[¶](#5.MapReduce实现WordCount实例（Python）)

创建/hadoop/data/mapreduce，并进入到/hadoop/data/mapreduce目录下

$ mkdir /hadoop/data/mapreduce

$ cd /hadoop/data/mapreduce

（1）首先编写MapReduce WordCount 代码。

①首先编写map阶段的代码，创建一个Python程序，命名为“count\_mapper.py”

$ vi count\_mapper.py

写入如下内容：

#!/usr/bin/env python

import sys

for line in sys.stdin:

line = line.strip()

words = line.split()

for word in words:

print '%s\t%s' % (word, 1)

②编写Reduce阶段的代码，创建一个Python程序，命名为“count\_reducer.py”

$ vi count\_reducer.py

写入如下内容：

#!/usr/bin/env python

from operator import itemgetter

import sys

current\_word = None

current\_count = 0

word = None

for line in sys.stdin:

line = line.strip()

word, count = line.split('\t', 1)

try:

count = int(count)

except ValueError:

continue

if current\_word == word:

current\_count += count

else:

if current\_word:

print '%s\t%s' % (current\_word, current\_count)

current\_count = count

current\_word = word

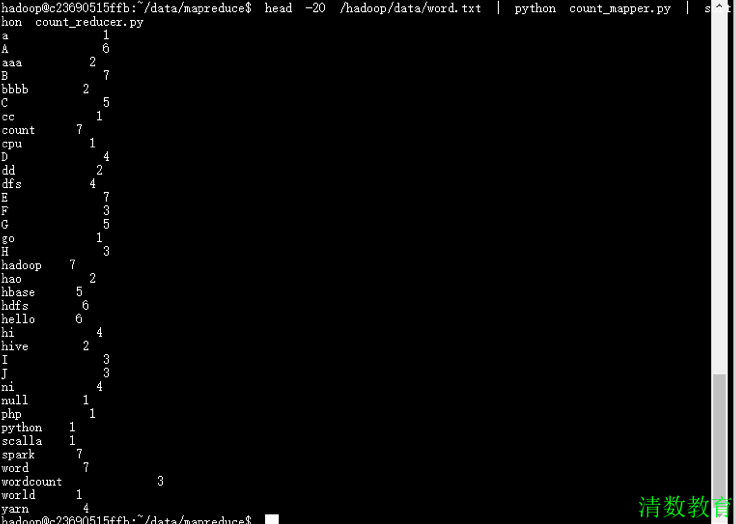
# do not forget to output the last word if needed!

if current\_word == word:

print '%s\t%s' % (current\_word, current\_count)

（2）程序编写完成后，首先在本地测试一下map和reduce，命令及图片如下：

$ head -20 /hadoop/data/word.txt | python count\_mapper.py | sort | python count\_reducer.py

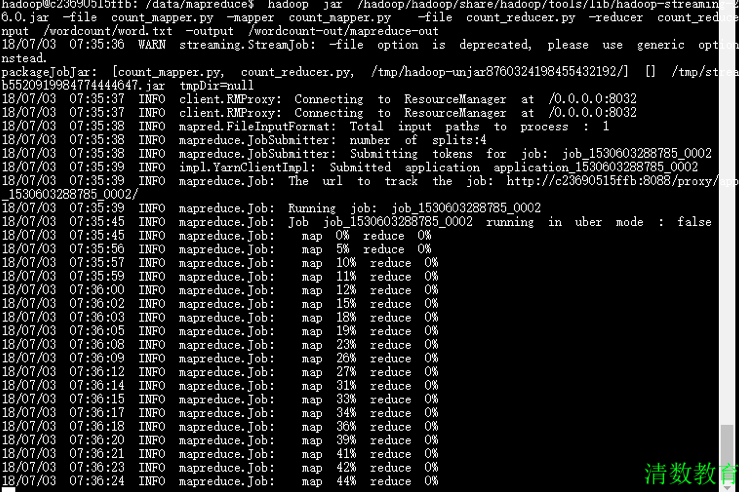


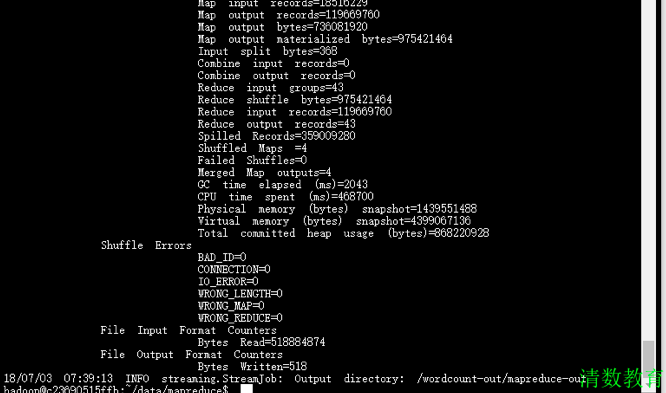
如上图，就证明map和reduce程序编写成功。

（3）运行该实例，命令如下：

$ hadoop jar /hadoop/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.6.0.jar -file count\_mapper.py -mapper count\_mapper.py -file count\_reducer.py -reducer count\_reducer.py -input /wordcount/word.txt -output /wordcount-out/mapreduce-out

（5）运行结果

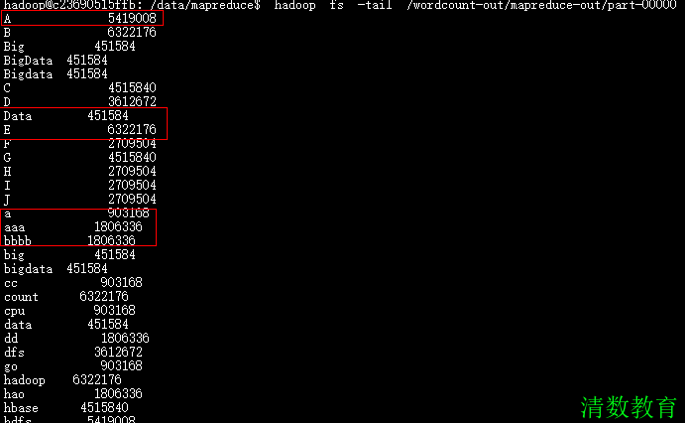




由上图可以看出，MapReduce对于495M的word.txt的文件进行WordCount分析用时大概3分钟37秒。

（6）查看结果

$ hadoop fs -tail /wordcount-out/mapreduce-out/part-00000



#### 6.Spark实现WordCount实例（python）[¶](#6.Spark实现WordCount实例（python）)

创建/hadoop/data/spark，并进入到/hadoop/data/spark目录下

$ mkdir /hadoop/data/spark

$ cd /hadoop/data/spark

（1）首先编写Spark WordCount 代码，创建一个Python程序，命名为“wordcount.py”。

$ vi wordcount.py

写入如下内容：

#!/usr/bin/env python

from pyspark import SparkContext

inputFile = 'hdfs://localhost:9000/wordcount/word.txt'

outputFile = 'hdfs://localhost:9000/wordcount-out/spark-out'

sc = SparkContext('local', 'wordcount')

text\_file = sc.textFile(inputFile)

counts = text\_file.flatMap(lambda line: line.split(' ')).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)

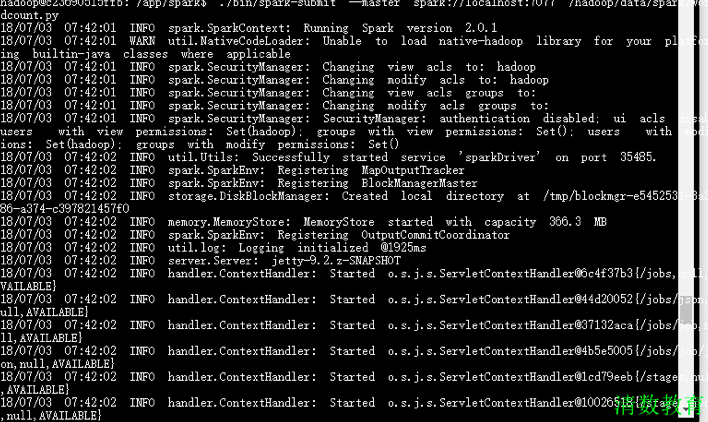
counts.saveAsTextFile(outputFile)

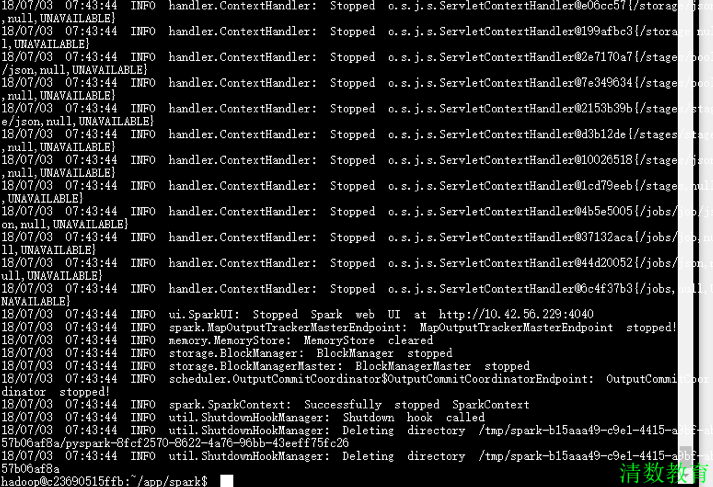
（2）运行该实例

$ cd /hadoop/app/spark/

$ ./bin/spark-submit --master spark://localhost:7077 /hadoop/data/spark/wordcount.py

（4）运行结果





（5）查看结果

$ hadoop fs -tail /wordcount-out/spark-out/part-00000



* 由上图可以看出，Spark对于495M的word.txt文件进行WordCount分析用时大概1分钟43秒。

#### 7.代码透析[¶](#7.代码透析)

#### （1）MapReduce实现WordCount实例（Python）[¶](#（1）MapReduce实现WordCount实例（Python）)

①map阶段的代码count\_mapper.py解析如下：

#!/usr/bin/env python

import sys

# 从标准输入过来的数据

for line in sys.stdin:

# 将首位的空格去掉

line = line.strip()

# 将这一行文本切分成单词（按空格）

words = line.split()

# 读一个单词写出一个<单词,1>

for word in words:

print '%s\t%s' % (word, 1)

②reduce阶段的代码count\_reducer.py解析如下：

#!/usr/bin/env python

from operator import itemgetter

import sys

current\_word = None

current\_count = 0

word = None

# 从标准输入过来的数据

for line in sys.stdin:

# 去除左右空格

line = line.strip()

# 按照tab键进行切分，得到word和次数1

word, count = line.split('\t', 1)

# 得到的1是一个字符串，需要类型转化

try:

count = int(count)

except ValueError:

# 如果不能转化成数字，输入有问题，转到下一行

continue

# 如果本次读取的单词和上一次一样，对次数加1

if current\_word == word:

current\_count += count

else:

if current\_word:

# 输出统计结果

print '%s\t%s' % (current\_word, current\_count)

current\_count = count

current\_word = word

# do not forget to output the last word if needed!

if current\_word == word:

print '%s\t%s' % (current\_word, current\_count)

#### （2）Spark实现WordCount实例（python）[¶](#（2）Spark实现WordCount实例（python）)

#!/usr/bin/env python

#导入包

from pyspark import SparkContext

# 输入输出路径，输出路径不需要自己创建，系统会自动生成

inputFile = 'hdfs://localhost:9000/wordcount/word.txt'

outputFile = 'hdfs://localhost:9000/wordcount-out/spark-out'

sc = SparkContext('local', 'wordcount')

text\_file = sc.textFile(inputFile)

##counts

counts = text\_file.flatMap(lambda line: line.split(' ')).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)

counts.saveAsTextFile(outputFile)

## 实验结果[¶](#实验结果)

由以上可知：对于495M的word.txt文件进行WordCount分析，MapReduce用时3分钟37秒，Spark用时1分钟43秒。Spark的计算速度大概是MapReduce的2倍。

当文件更大时，对于相同的案例，Spark比MapReduce的性能要更加强大。