实 验 报 告

实验 8 & 9

517030910374
郭嘉宋

目录:

注：在编程过程中我也遇到了很多困难和 Bug，还有很多思考了很久得出的结论与解释，以及一些需要注释的内容，都在下述 “2. 实验过程” 中，用加粗的 NOTE 加以了说明。

正文:

1. 实验准备

(1) 实验环境

本实验主要采用 Ubuntu 系统下的 python2.7 进行，使用 miniconda 环境，Ubuntu 系统安装在 VMware Workstation 提供的虚拟机中，同时使用 Hadoop 对创建的用户 hduser 远程访问模拟分布式开发、服务器开发等。

(2) 实验目的

实验 8:

配置 Hadoop 环境，了解 Hadoop，熟悉 HDFS 和 MapReduce，使用 MapReduce 计算圆周率π的值，体会 Hadoop 的高可靠性、高容错性和高效性。

实验 9:

继续学习 Hadoop streaming，了解 mapper 和 reducer 的运行关系，编写自己的 mapper 和 reducer, 以及试着写一个 linux 下的 bash 脚本来运行 Hadoop 文件。使用 Hadoop 完成分词统计, 以及了解 PageRank 的算法, 写一个 bash 脚本来模拟计算网页的 PageRank。

(3) 实验原理

实验 8:

Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构。用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。

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

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

Hadoop 由 Apache Software Foundation 公司于 2005 年秋天作为 Lucene 的子项目 Nutch 的一部分正式引入。它受到最先由 Google Lab 开发的 Map/Reduce 和 Google File System(GFS) 的启发。

Hadoop 是可靠的，因为它假设计算元素和存储会失败，因此它维护多个工作数据副本，确保能够针对失败的节点重新分布处理。

Hadoop 是高效的，因为它以并行的方式工作，通过并行处理加快处理

速度。

Hadoop 还是可伸缩的，能够处理 PB 级数据。

此外，Hadoop 依赖于社区服务，因此它的成本比较低，任何人都可以使用。

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。它主要有以下几个优点：

高可靠性。Hadoop 按位存储和处理数据的能力值得人们信赖。

高扩展性。Hadoop 是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。

高效性。Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。

高容错性。Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。

低成本。与一体机、商用数据仓库以及 QlikView、Yonghong Z-Suite 等数据集市相比，hadoop 是开源的，项目的软件成本因此会大大降低。

Hadoop 带有用 Java 语言编写的框架，因此运行在 Linux 生产平台上是非常理想的。

实验 9:

Hadoop Streaming 框架，最大的好处是，让任何语言编写的 map, reduce 程序能够在 hadoop 集群上运行，map/reduce 程序只要遵循从标准输入 stdin 读，写出到标准输出 stdout 即可。

Haoop 支持用其他语言来编程，需要用到名为 Streaming 的通用 API。

Streaming 主要用于编写简单，短小的 MapReduce 程序，可以通过脚本语言编程，开发更快捷，并充分利用非 Java 库。

HadoopStreaming 使用 Unix 中的流与程序交互，从 stdin 输入数据，从 stdout 输出数据。

练习 2 我使用了自己的算法，将在实验步骤中具体阐述。

2. 实验过程

(1) 实验 8

1.1 实验步骤

首先在按照教程安装 Hadoop 以及相应组件。安装完成后，进入创建的新用户 hduser，然后再打开 hadoop 服务，如下图所示：

```
hduser@ubuntu:~$ jps
28404 NodeManager
28268 ResourceManager
28711 Jps
27894 DataNode
28107 SecondaryNameNode
27763 NameNode
hduser@ubuntu:~$
```

练习 1:

输入如下命令，计算圆周率:

```
hduser@ubuntu:~$ hadoop jar /usr/local/hadoop/share/hadoop/mapred
uce/hadoop-mapreduce-examples-2.2.0.jar pi 2 10
Number of Maps  = 2
Samples per Map = 10
```

NOTE：上图中命令最后两个参数分别为运行的 map 个数和每次运算投掷的个数，因此第二个参数的大小将极大影响π的精确度。

相似的，只需要按照表格中对应输入两个参数即可求出结果。

练习 2:

为得到更加精确的π值，此时我将两个参数取为 100 和 100000000，

如下图所示:

```
hduser@ubuntu:~$ hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapr
educe-examples-2.2.0.jar pi 100 100000000
```

将输出结果保存即可。

1.2 实验结果

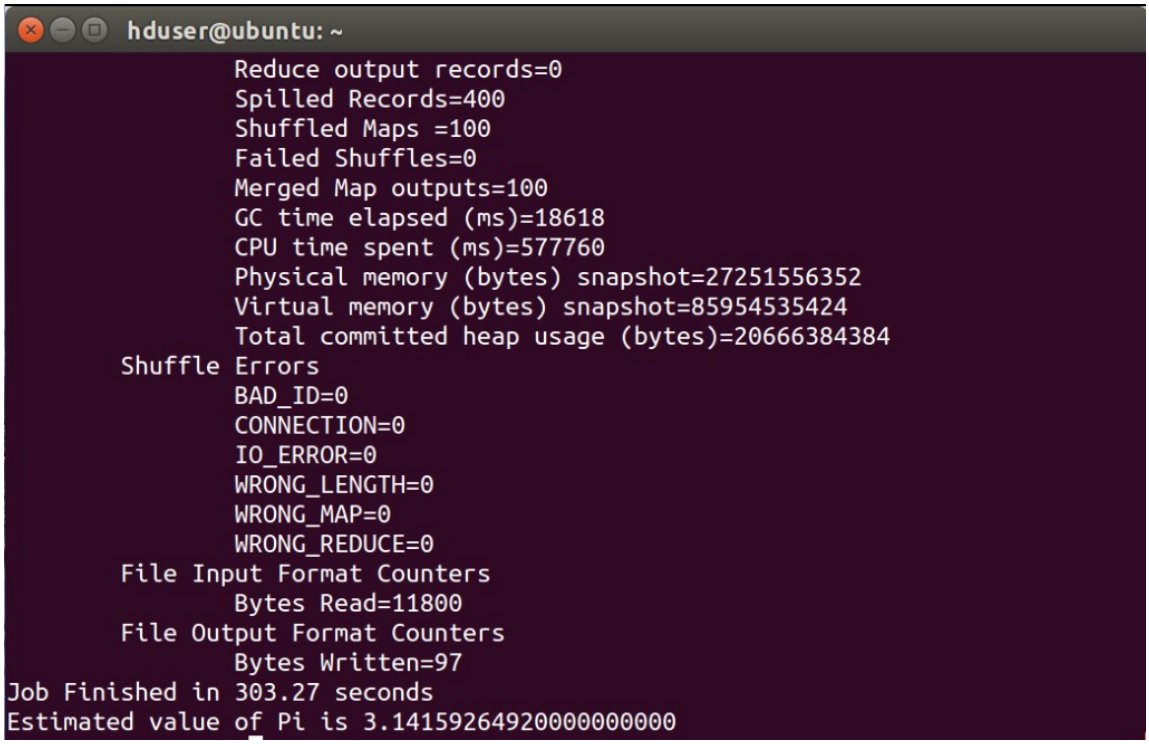
练习 1:

```
hduser@ubuntu: ~
                Reduce output records=0
                Spilled Records=8
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=243
                CPU time spent (ms)=3560
                Physical memory (bytes) snapshot=663597056
                Virtual memory (bytes) snapshot=2579591168
                Total committed heap usage (bytes)=493879296
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=236
        File Output Format Counters
                Bytes Written=97
Job Finished in 34.835 seconds
Estimated value of Pi is 3.80000000000000000000
hduser@ubuntu:~$
```

如上图所示，此时为参数为 2、10 时的测试结果，非常直观，完整的实验结果如下表格显示：

Number of Maps	Number of samples	Time(s)	π
2	10	34.835	3.8
5	10	47.344	3.28
10	10	55.851	3.2
2	100	29.203	3.12
10	100	38.916	3.148
100	100000000	303.27	3.1415926492

练习 2:



最终计算的π值精确到了小数点后 10 位，且结果正确十分准确。

(2) 实验 9

2.1 实验步骤

练习 1:

我未对 mapper.py 进行修改，依旧沿用其分词的功能，如下图所示：

```
#!/usr/bin/env python
import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer1.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

mapper.py 的功能十分简单，就是将原文分成一个个单词输出并传参给 reduce.py。

reducer.py 如下图所示，首先我先创建了一个数组统计每个字母出现的单词数。

```
answer_count = []
for i in range(27):
    answer_count.append(0)
```

再从 mapper.py 传入参数，对应的字符长度和个数加到字典和 list 中，如下图所示：

```
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    while True:
        if word[0] != chr(ord('a') + i):
            i+=1
        else:
            break;
    answer[chr(ord('a')+i)]+=len(word)
    answer_count[1+i]+=1
```

最后求出平均字符长度，输出即可，如下图所示：

```
final=[]
for i in range(27):
    final.append(0)
for i in range(26):
    if answer_count[i+1]==0:
        print chr(ord('a')+i),'\t',0
        # print chr(ord('a')+i),'\t',final[i + 1]
    else:

        final[i+1]=float(answer[chr(ord('a')+i)])/answer_count[i+1]
        print chr(ord('a')+i),'\t',final[i+1]
```

练习 2:

我模拟的测试文件，其每行的含义是，如第一行表示第 1 个 url，他最初的 PageRank 是 10，他指向的 url 是 2、4、6 。我以 100 表示满的 PageRank，因此最初给 9 个 url 每个的 PageRank 都为 10，最终的 PageRank 数值大小即为点到该网页的概率，如下图所示。

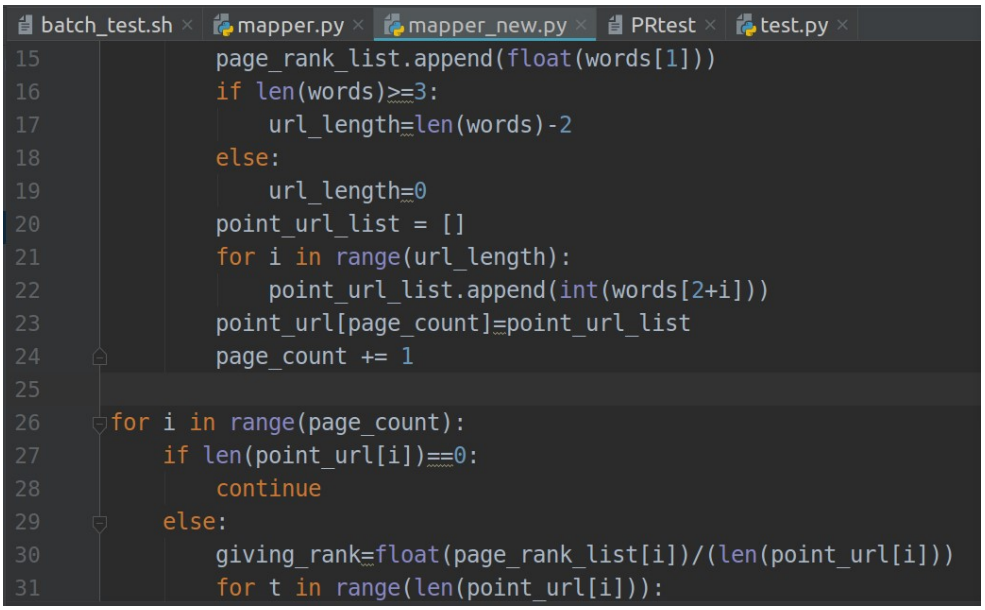
PRtest ×

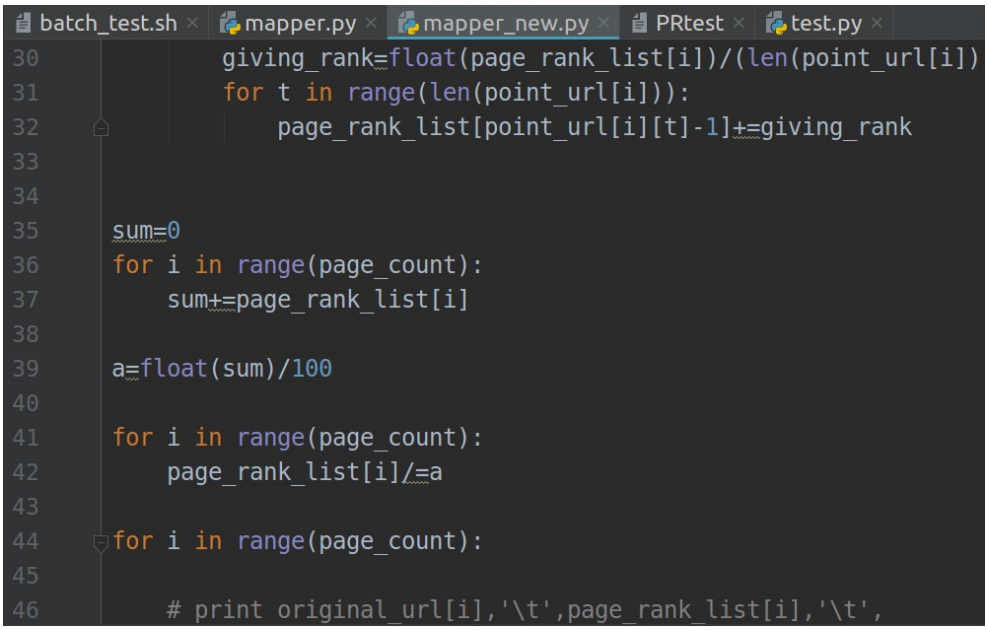
```
1 10 2 4 6
2 10 3 6 8
3 10 6
4 10 6 7
5 10 1 2
6 10 9 3
7 10 3 1 2 4
8 10 5 7 9
9 10 6 1 2 3 4 5
```

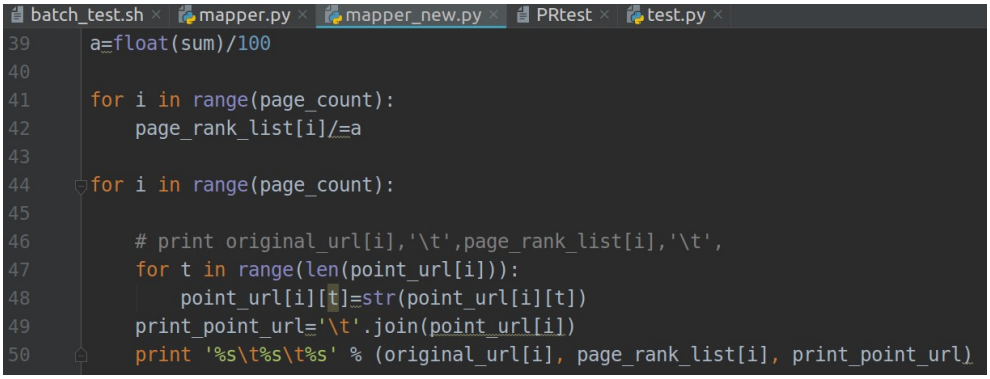
首先我尝试不改变例子中的结构，仅使用 mapper 进行试验，因此我让 mapper 功能如下图所示，对每一行存入数组和链表，最终输出结果，如下图所示：

batch_test.sh × mapper.py × mapper_new.py × PRtest × test.py

```
3   import sys
4   page_count=0
5   page_rank_list=[]
6   point_url={}
7   original_url=[]
8   # f=open("PRtest")
9   # for line in f:
10  for line in sys.stdin:
11      line = line.strip()
12      if len(line) > 0:
13          words = line.split()
14          original_url.append(int(words[0]))
15          page_rank_list.append(float(words[1]))
```







如上图代码所示，我模仿例子在 map 中输出刷新后的 PageRank，但是运行会出

现如下报错:

```
 mode : false
18/11/21 22:28:01 INFO mapreduce.Job:  map 0% reduce 0%
18/11/21 22:28:10 INFO mapreduce.Job:  map 100% reduce 0%
18/11/21 22:28:15 INFO mapreduce.Job: Task Id : attempt_1542807399911_0017_r_000
000_0, Status : FAILED
Error: java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess fa
iled with code 1
        at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.j
ava:320)
        at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java
:533)
        at org.apache.hadoop.streaming.PipeReducer.close(PipeReducer.java:134)
        at org.apache.hadoop.io.IOUtils.cleanup(IOUtils.java:237)
        at org.apache.hadoop.mapred.ReduceTask.runOldReducer(ReduceTask.java:477
)
        at org.apache.hadoop.mapred.ReduceTask.run(ReduceTask.java:408)
        at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:162)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:421)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInforma
tion.java:1491)
        at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:157)
```

NOTE：上述问题我解决了很久，最开始以为是输出不是标准输出，测试了很久的输出格式依旧报错，即使没有循环调用，只执行 1 次还是会报错，最后经上网查阅资料后得知，mapper 是分多线程执行程序的，因此每个 mapper 并不会得到输入的全部信息，只会存入部分信息，因此在制作 list、dictionary 的时候因为只有部分信息就会和别的 mapper 冲突,故本实验必须使用 reducer 进行整合，否则多线程 mapper 是无法完成的。

修改后的 mapper 如下图所示，仅仅将所需要的信息输出，传给 reducer 处理。

```
#!/usr/bin/env python

import sys
# f=open("PRtest")
# for line in f:
for line in sys.stdin:
    line = line.strip()
    if len(line) > 0:
        words = line.split()
        if len(words)>=3:
            for t in range(2,len(words)):
                words[t]=str(words[t])
            tmp=' '.join(words[2:])
        else:
            tmp=''
        print '%s\t%s\t%s' %(words[0],words[1],tmp)
```

修改后的 reducer 如下图所示:

```python
#!/usr/bin/env python

import sys
page_count=0
page_rank_list=[]
point_url={}
original_url=[]
# f=open("PRtest")
# for line in f:
for line in sys.stdin:
    line = line.strip()
    if len(line) > 0:
        words = line.split()
        original_url.append(int(words[0]))
        page_rank_list.append(float(words[1]))
        if len(words)>=3:
            url_length=len(words)-2
        else:
            url_length=0
        point_url_list = []
        for i in range(url_length):
            point_url_list.append(int(words[2+i]))
        point_url[page_count]=point_url_list
        page_count += 1

for i in range(page_count):
    if len(point_url[i])==0:
        continue
    else:
        giving_rank=float(page_rank_list[i])/(len(point_url[i]))
        for t in range(len(point_url[i])):
            page_rank_list[point_url[i][t]-1]+=giving_rank
```

```python
sum=0
for i in range(page_count):
    sum+=page_rank_list[i]

a=float(sum)/100

for i in range(page_count):
    page_rank_list[i]/=a

for i in range(page_count):

    # print original_url[i],'\t',page_rank_list[i],'\t',
    for t in range(len(point_url[i])):
        point_url[i][t]=str(point_url[i][t])
    print_point_url='\t'.join(point_url[i])
    print '%s\t%s\t%s' % (original_url[i], page_rank_list[i], print_point_url)
```

Reducer 将 mapper 杂乱的多线程结果进行整合，统计新一轮的 PageRank，如上图所示。

NOTE：在计算 PageRank 时，我先查阅了网上的简易说明，使用 L.Page 给出的公式计算，但我经过思考想尝试自己的算法，我将所有的网页给初值 10，既然 PageRank 表示的是选中此网页的概率，那么我每次就将链接的网页采用投票制，比如 1 指向了 3、6、7，那么就给 3、6、7 加上 1 的 PageRank 的 1/3，并不对 1 做任何改变，这样相比 L.Page 的阻尼系数更为简单，但是这样会使 PageRank 越来越大，最后再归一化到 100，除以相应系数即可，这样就可以依旧表示相对 100 的概率，且操作起来并没有马尔可夫链，较为容易。

2.2 实验结果

练习 1:
在 terminal 直接输入结果测试:

```
hduser@ubuntu:~$ echo "we become what we do"  | ~/experiment/src/
mapper.py  | sort  -k1,1| ~/experiment/src/reducer1.py
```

结果如下:

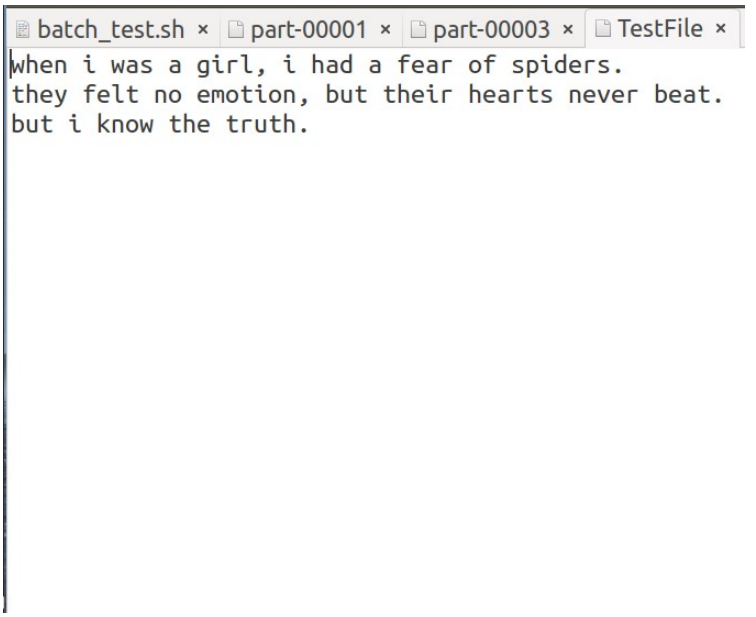
```
hduser@ubuntu:~$ echo "we become what we do"  | ~/experiment/src/
mapper.py  | sort  -k1,1| ~/experiment/src/reducer1.py
b       6.0
d       2.0
w       2.66666666667
hduser@ubuntu:~$
```

输出结果正确。

若以文件的方式进行输入，此时输入文件 TestFile，复制入 hadoop 的 tempinput，如下图所示:

```
hduser@ubuntu:~$ hadoop fs -copyFromLocal ~/input/TestFile tempin
put
```

TestFile 的内容如下图所示:



输入测试命令：

```
hduser@ubuntu:~/experiment/src$ hadoop jar /usr/local/hadoop/share/
hadoop/tools/lib/hadoop-streaming-2.2.0.jar -files mapper.py,reduce
r1.py -mapper mapper.py -reducer reducer1.py -input tempinput -outp
ut tempoutput
```

输入完毕后输入命令调出结果：

```
hduser@ubuntu:~/experiment/src$ hadoop fs -cat tempoutput/part-0000
0
18/11/21 16:33:55 WARN util.NativeCodeLoader: Unable to load native
-hadoop library for your platform... using builtin-java classes whe
re applicable
a       1.0
b       3.66666666667
e       8.0
f       4.0
g       5.0
h       4.5
i       1.0
k       4.0
n       3.5
o       2.0
s       8.0
t       4.5
w       3.5
hduser@ubuntu:~/experiment/src$
```

输出结果正确。
文件测试和直接输入测试结果皆正确，实验成功。

练习 2：

```
hduser@ubuntu:~/experiment/src/accumulator$ ./batch_test.sh 5
```

如上图所示，5 次迭代测试 PageRank，结果如下图所示：

```
1       7.22163938306   2       4       6
2       8.57238875621   3       6       8
3       23.8724433674   6
4       7.02348214496   6       7
5       5.54959698206   1       2
6       24.4456187022   9       3
7       3.84409516939   3       1       2       4
8       2.1113255655    5       7       9
9       17.3594099292   6       1       2       3       4       5
```

算法有效，直观上 6 被指向的最多，所以 PageRank 最高，而 6 最为重要，他指向的 3 PageRank 也非常高，以此类推皆可验证。

测试 10 次迭代的结果，如下图：

```
1       7.0639569182    2       4       6
2       8.31724228617   3       6       8
3       24.2847504153   6
4       6.8554432634    6       7
5       5.49972984544   1       2
6       24.9016017456   9       3
7       3.42317636302   3       1       2       4
8       1.89490026371   5       7       9
9       17.7591988991   6       1       2       3       4       5
```

结果较 5 次迭代更加准确，依旧有效，实验成功。

3. 实验总结

此次试验是我第一次接触 Hadoop，学习的时间非常长，在完成实验后也非常有成就感和自豪感。Hadoop 最开始对于萌新有些抽象，尤其是大量的 linux 命令需要从零学起，而且 Hadoop 的错误不太好找，因为 terminal 中无法显示报错信息，因此我后来都单独输入文件不通过 terminal 运行，更容易发现错误。
此次实验让我认识了 Hadoop 功能的强大，多线程使运算效率大幅增加。

注：在编程过程中我也遇到了很多困难和 Bug，还有很多思考了很久得出的结论与解释，以及一些需要注释的内容，都在下述“2. 实验过程”中，用加粗的 NOTE 加以了说明。

令我印象深刻的就是我自己设计的 PageRank 算法，虽然不是太难想到，可是省去了马尔可夫链的复杂计算，十分简便就可以计算出 PageRank 的有效值，而且数值上既反映了概率又可以体现相对大小。
最后希望在之后的课程学习中可以把 Hadoop 和搜索引擎相结合，完善最终的成果，继续努力！

o(*≧▽≦)ツ!

517030910374

郭嘉宋