MapReduce

一、建立自己的MapReduce框架

框架设计

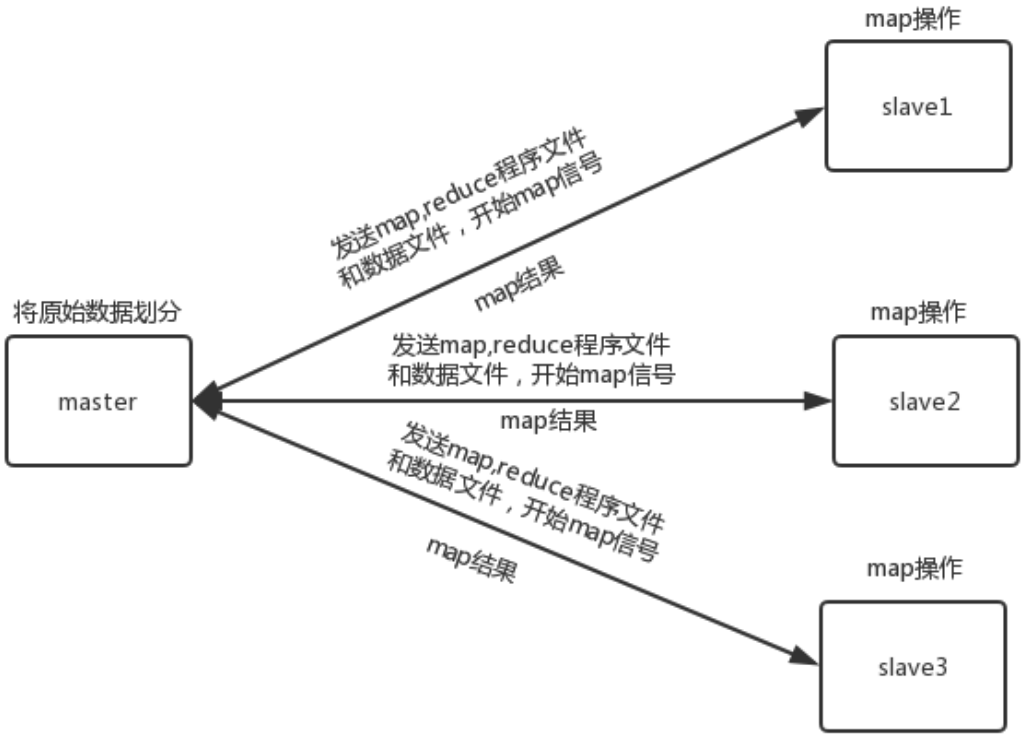
本次实验把thumm01(Master)作为控制节点，thumm02～04(Slave)三台机器作为运算节点，大概流程如下

---

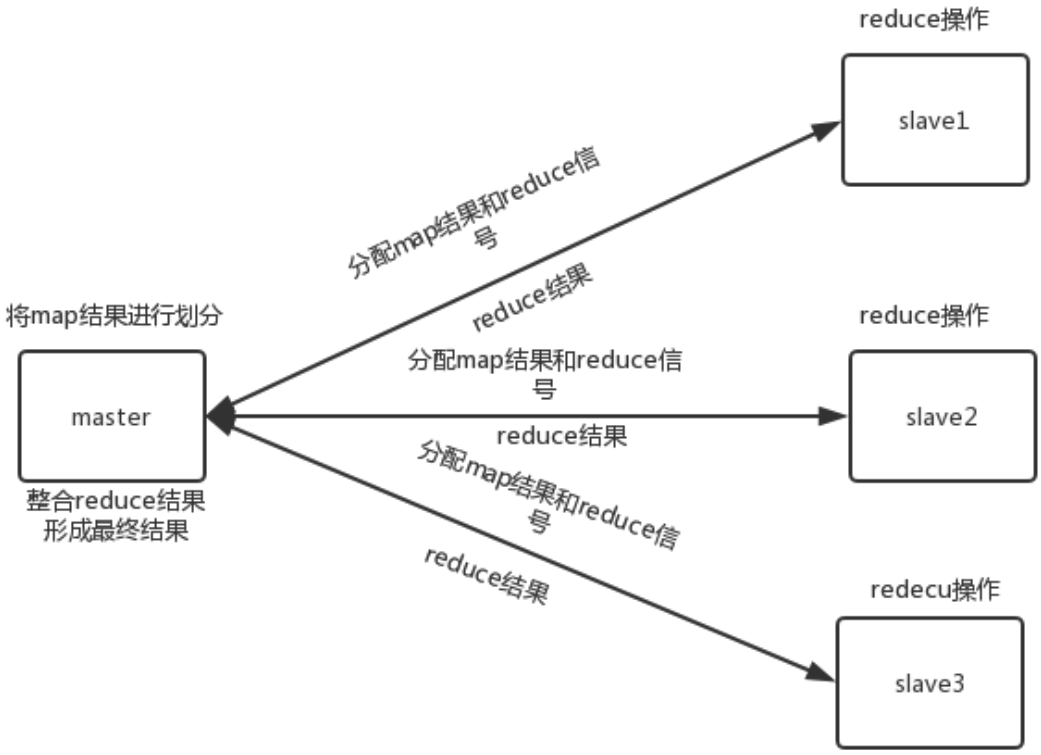
1. Master分发数据文件、map和reduce程序文件给各个Slave，然后发送map信号
2. Slave对收到的数据执行mapper.py，并将结果按照<key,value>中的key分成不同的txt （如：aa.txt里存放所有的<aa,1>记录），并将这些word.txt发送给Master
3. Master收到所有Slave的所有map结果后，将相同key的结果合并（即：把不同Slave执行map结果中的相同单词的文件合为一个文件），然后对这些map结果进行任务划分，然后向各个Slave发送划分的map结果和reduce信号
4. Slave收到数据和信号后开始执行reducer.py，执行结果存入txt中并且发送给Master
5. Master收到所有reduce结果后将其合并形成最终结果

流程图如下：

map阶段：



reduce阶段：



框架原理和实现

数据文件分块

这次的wordcount要按行来读取，上次实验是根据文件大小分块，可能会导致分块文件的第一行和最后一行数据不全，也就是被截取到另外的文件中的情况

我在网上查找了在HDFS上进行MapReduce关于处理这个问题的机制[《MapReduce中如何处理跨行的Block和InputSplit》]

显然这次的实验时间和我个人能力难以实现与HDFS一样的机制，所以重新设计文件分块函数来发送数据到各个Slave，本次实验有三台用来运算的节点，所以采取如下方法来实现文件按行的均匀分块

- 使用readlines函数将原始数据全部按行读出形成一个list，每一个元素就是一行
- 将读出的list按照list的大小均分成三分写入新的文件分块中
- 考虑到原始文件行数也许不能被三整除，所以最后一个分块从循环中拿出来单独处理

```
#数据按行均分slave_num个
def splitfile(file):
    fp = open(file,'r')
    lines = fp.readlines()
    fp.close()
    chunk = int(len(lines)/slave_num)
    index = 0
    for i in range(slave_num-1):
        fp = open("data"+str(i+1)+".txt",'w')
        fp.writelines(lines[index:index+chunk])
        index += chunk
        fp.close()
    fp = open("data" + str(slave_num) + ".txt", 'w')
    fp.writelines(lines[index:])
    fp.close()
```

reduce任务分配（shuffle）

本实验设计中所有mapper在执行map完毕后将map结果传送给Master，由Master将数据统一后再进行reduce任务划分，再将划分后的map结果传送给各个Slave作reduce操作

- 各个Slave在map完毕后对结果排序后将结果回传给Master
- 当Master收到所有的map结果后，将其合并成一个文件mapout.txt
- 对mapmapout.txt文件计算其行数，并且按行进行排序
- 根据行数除以三得到大概的每段长度
- 从分段点往前十行开始，这里的十行是我设置的一个偏移量，找寻第一个不同于前一行的行，其位置就是文件的切分点
- 切分好文件后分别发送给Slave，并且发送开始reduce信号

```
# 分配reduce任务
def shuffle():
    fp = open("mapout.txt",'r')
    lines = fp.readlines()
```

```
    lines.sort()
    length = len(lines)
    chunk = int(length/slave_num)

    index1 = chunk-10
    last = lines[index1]
    while lines[index1] == last:
        index1 += 1

    index2 = 2*chunk-10
    last = lines[index2]
    while lines[index2] == last:
        index2 += 1

    senddata(lines[:index1], 3, slave[0])
    senddata(lines[index1:index2], 3, slave[1])
    senddata(lines[index2:], 3, slave[2])

# 发送分配的reduce任务数据
def senddata(list, sign, DestIp):
    sk_slave = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sk_slave.settimeout(10)
    sk_slave.connect((DestIp, 7777))
    fhead = struct.pack('128si', "reduce_data.txt", sign)
    sk_slave.send(fhead)
    for line in list:
        sk_slave.send(line)
    print "发送分配的reduce数据给" + DestIp + "完毕"
    sk_slave.close()
```

文件传送

本次试验还是使用socket来传送各种文件，原理同上次实验

发送端使用slice对文件分片进行发送

```
# 发送文件
def sendfile(file, sign, DestIp):
    sk_slave = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sk_slave.settimeout(10)
    sk_slave.connect((DestIp, 7777))
    fhead = struct.pack('128si', file, sign)
    sk_slave.send(fhead)

    fp = open(file, 'rb')
    for slice in fp:
        sk_slave.send(slice)
    fp.close()
```

```
    print "发送" + file + "给" + DestIp + "完毕"
    sk_slave.close()
```

接收端每次接收1024字节直至文件结束

```
fp = open(filepath, 'a')
while 1:
    data = conn.recv(1024)
    if not data:
        break
    fp.write(data)
fp.close()
print "recv %s complete" % filename
```

Master与Slave通信

定义一个信息头长度，其中128s表示**文件名**，长度最多128字节，i代表**信号**

```
fileinfo_size = struct.calcsize('128si')
```

发送端先发送定长的打包后的信息

```
fhead = struct.pack('128si', file, sign)
    socket.send(fhead)
```

接收端接收后拆包来获得信息

```
from_recv = conn.recv(fileinfo_size)
    file, sign = struct.unpack('128si', from_recv)
```

信号含义:

Master ==> Slave

11 mapper.py文件传送信号

12 reducer.py文件传送信号

13 data.txt文件传送信号

2 开始map信号

3 开始reduce信号

Slave ==> Master

1 map结果文件，Master收到3个1信号代表3台机器都map完毕

2 reduce结果文件，Master收到3个2信号代表3台机器都reduce完毕

python中执行python文件

因为Master发送了mapper.py和reducer.py给各个Slave，Slave需要在接收到相关信号后执行map或reduce

这里使用system函数在python中调用命令行，再在命令行中执行mapper.py或reducer.py，并且输出重定向到文件

以执行reducer.py为例

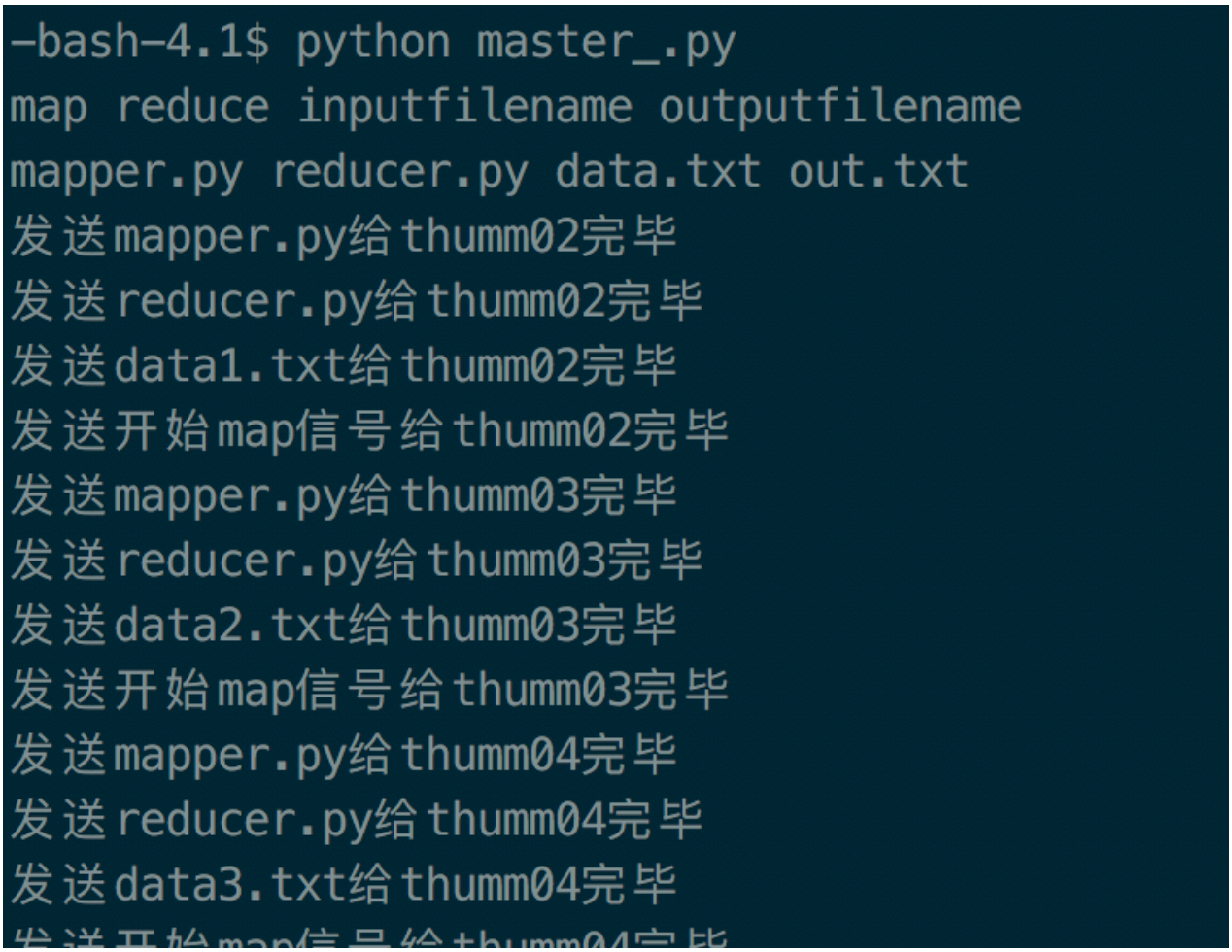
```
os.system("cat %s | ./reducer.py > reduceout.txt" % filename)
```

实验结果

分别在Master和Slave123上运行相应python程序，

原始数据是我自己弄的一个大概100M的文章，mapper和reducer则是python的wordcount程序，具体见附件

在Master上输入`mapper.py reducer.py data.txt out.txt`，然后Master输出如下：(上次实验出现的用户名显示问题暂时懒得管。。。) 用时102秒



二、在Hadoop上使用MapReduce

实验步骤

在HDFS我的学号目录下创建了一个input文件夹用来存放wordcount数据

```
$ hadoop fs -mkdir input
```

使用上一节相同的数据文件，将其传到HDFS的input文件夹下（根目录是我的学号）

```
$ hadoop fs -put data.txt input
```

接下来使用与上一节相同的mapper.py和reducer.py程序

使用python编写的wordcount在hadoop上运行，需要使用到hadoop-streaming-*.jar这个包

首先找到学校服务器上这个jar包的位置

```
$ cd $HADOOP_HOME
$ find ./ -name "*streaming*"
```

显示如下

```
-bash-4.1$ cd $HADOOP_HOME
-bash-4.1$ find ./ -name "*streaming*"
./share/hadoop/tools/sources/hadoop-streaming-2.5.2-test-sources.jar
./share/hadoop/tools/sources/hadoop-streaming-2.5.2-sources.jar
./share/hadoop/tools/lib/hadoop-streaming-2.5.2.jar
-bash-4.1$
```

如图第三个jar包即为我所要找的，我已提前将mapper.py和reducer.py传到服务器上，输入以下命令在hadoop上运行mapreduce程序

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.5.2.jar
\
$ -files ./mapper.py,./reducer.py \
$ -mapper ./mapper.py \
$ -reducer ./reducer.py \
$ -input input/*.txt \
$ -output output
```

然后等待结果，显示如下(太长了不方便截图)

```
-bash-4.1$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-
2.5.2.jar -files ./mapper.py,./reducer.py -mapper ./mapper.py -reducer
./reducer.py -input input/*.txt -output output
packageJobJar: [/opt/data/hadoop/hadoop-unjar5339984041442313943/] []
/tmp/streamjob558792385327292602.jar tmpDir=null
17/12/01 19:10:16 INFO mapred.FileInputFormat: Total input paths to process
: 1
17/12/01 19:10:16 INFO mapreduce.JobSubmitter: number of splits:2
17/12/01 19:10:16 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1511440732705_0405
17/12/01 19:10:16 INFO impl.YarnClientImpl: Submitted application
application_1511440732705_0405
17/12/01 19:10:16 INFO mapreduce.Job: The url to track the job:
http://thumm08:8088/proxy/application_1511440732705_0405/
17/12/01 19:10:16 INFO mapreduce.Job: Running job: job_1511440732705_0405
17/12/01 19:10:20 INFO mapreduce.Job: Job job_1511440732705_0405 running in
uber mode : false
17/12/01 19:10:20 INFO mapreduce.Job:  map 0% reduce 0%
17/12/01 19:10:31 INFO mapreduce.Job:  map 22% reduce 0%
17/12/01 19:10:32 INFO mapreduce.Job:  map 35% reduce 0%
17/12/01 19:10:34 INFO mapreduce.Job:  map 47% reduce 0%
```

```
17/12/01 19:10:36 INFO mapreduce.Job:  map 55% reduce 0%
17/12/01 19:10:39 INFO mapreduce.Job:  map 63% reduce 0%
17/12/01 19:10:42 INFO mapreduce.Job:  map 67% reduce 0%
17/12/01 19:10:44 INFO mapreduce.Job:  map 83% reduce 0%
17/12/01 19:10:55 INFO mapreduce.Job:  map 100% reduce 17%
17/12/01 19:10:58 INFO mapreduce.Job:  map 100% reduce 37%
17/12/01 19:11:01 INFO mapreduce.Job:  map 100% reduce 46%
17/12/01 19:11:05 INFO mapreduce.Job:  map 100% reduce 54%
17/12/01 19:11:08 INFO mapreduce.Job:  map 100% reduce 62%
17/12/01 19:11:11 INFO mapreduce.Job:  map 100% reduce 68%
17/12/01 19:11:14 INFO mapreduce.Job:  map 100% reduce 71%
17/12/01 19:11:17 INFO mapreduce.Job:  map 100% reduce 73%
17/12/01 19:11:20 INFO mapreduce.Job:  map 100% reduce 77%
17/12/01 19:11:23 INFO mapreduce.Job:  map 100% reduce 80%
17/12/01 19:11:26 INFO mapreduce.Job:  map 100% reduce 83%
17/12/01 19:11:29 INFO mapreduce.Job:  map 100% reduce 86%
17/12/01 19:11:32 INFO mapreduce.Job:  map 100% reduce 89%
17/12/01 19:11:36 INFO mapreduce.Job:  map 100% reduce 93%
17/12/01 19:11:42 INFO mapreduce.Job:  map 100% reduce 98%
17/12/01 19:11:44 INFO mapreduce.Job:  map 100% reduce 100%
17/12/01 19:11:44 INFO mapreduce.Job: Job job_1511440732705_0405 completed
successfully
17/12/01 19:11:44 INFO mapreduce.Job: Counters: 49
    File System Counters
        FILE: Number of bytes read=174080556
        FILE: Number of bytes written=348479054
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=105175642
        HDFS: Number of bytes written=533392
        HDFS: Number of read operations=9
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=2
        Launched reduce tasks=1
        Data-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=105434
        Total time spent by all reduces in occupied slots (ms)=174624
        Total time spent by all map tasks (ms)=52717
        Total time spent by all reduce tasks (ms)=58208
        Total vcore-seconds taken by all map tasks=52717
        Total vcore-seconds taken by all reduce tasks=58208
        Total megabyte-seconds taken by all map tasks=80973312
        Total megabyte-seconds taken by all reduce tasks=178814976
    Map-Reduce Framework
        Map input records=2276150
        Map output records=18097200
```

```
        Map output bytes=137886150
        Map output materialized bytes=174080562
        Input split bytes=198
        Combine input records=0
        Combine output records=0
        Reduce input groups=44735
        Reduce shuffle bytes=174080562
        Reduce input records=18097200
        Reduce output records=44735
        Spilled Records=36194400
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=322
        CPU time spent (ms)=131500
        Physical memory (bytes) snapshot=3127742464
        Virtual memory (bytes) snapshot=7301087232
        Total committed heap usage (bytes)=4064804864
    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=105175444
    File Output Format Counters
        Bytes Written=533392
17/12/01 19:11:44 INFO streaming.StreamJob: Output directory: output
```

将HDFS的output文件夹下的part-00000文件拷回本地

```
hadoop fs -get output/part-00000
```

性能对比

可以从hadoop运行信息中找到程序起始时间为19:10:16，结束时间为19:11:44，用时88秒

相比我自己的框架，还是hadoop效率较高，文件越大，hadoop的优势越大，毕竟hadoop是这么成熟的一个框架，我自己写的肯定比不上

三、实现PageRank算法

原理

将矩阵按列分割，得到M = $[M_1 M_2 M_3 \cdots M_k]$，再将概率分布矩阵 V 按行分割，得到V^T = $[V_1 V_2 V_3 \cdots V_k]$，每个 M_i、V_i对应一台实体机器。

1. Mapper： 将 M_i中每一行与 V_i相乘，得到<key,value>对。 其中 key 代表了第 i 行，即网站编号；value 为矩阵相乘结果。
2. Reducer：计算所有 value 之和。
3. 迭代： 对于每个 Reducer 的结果 O_j，计算得到新的 V_{i+1}，其中每一个元素 $v_j = \alpha O_j + (1-\alpha)\begin{bmatrix} \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{bmatrix}$,将 V_{i+1} 传输至下一轮迭代。

实现

数据输入格式设计

由于mapper和reducer是按行进行处理，所以要对矩阵进行预处理使之符合格式要求

对于原理中的M_i, V_i是放在同一个mapper中进行计算的，所以M_i和V_i要放入同一行中,又因为M_i是列向量，我把对mapper的每一行输入设计为

$$M_i^T, V_i \tag{1}$$

即每一行的前N个数为列向量M_i的的转置，第N+1个数为V_i, N为网页数

数据预处理

为了使原始数据转化为上述设计的格式，需要对原始数据进行预处理

原始数据格式为：

对矩阵M作转置后存放在MT.txt中，格式如 $\begin{bmatrix} M_1^T \\ \vdots \\ M_N^T \end{bmatrix}$，其中每一行为M的列向量；列向量V则按照原始格式存放在V.txt,格式如 $\begin{bmatrix} V_1 \\ \vdots \\ V_N \end{bmatrix}$

数据预处理即把上述两个文件进行列拼接，将 $\begin{bmatrix} M_1^T \\ \vdots \\ M_N^T \end{bmatrix}$ 和 $\begin{bmatrix} V_1 \\ \vdots \\ V_N \end{bmatrix}$ 拼接为 $\begin{bmatrix} M_1^T & V_1 \\ \vdots & \vdots \\ M_N^T & V_N \end{bmatrix}$，形成新的数据文件存放在

MTV.txt中

代码实现 PageRank_pretreatment.py

```
#encoding:utf8

MT = open("MT.txt",'r')
V = open("V.txt",'r')
MTV = open("MTV.txt",'w')

MT_line = MT.readline().strip()
V_line = V.readline().strip()
MTV_lines = []

while MT_line:
    MTV_lines.append(MT_line + " " + V_line + "\n")
    MT_line = MT.readline().strip()
    V_line = V.readline().strip()
MT.close()
V.close()

#删除最后一行的换行符
MTV_lines[-1] = MTV_lines[-1][:-1]
MTV.writelines(MTV_lines)
MTV.close()
```

mapper设计

对于每一行（共N+1个数）的输入，取前N个为M_i^T,最后一个为V_i，对于M_i^T中的每一列分别计算与V_i的乘积，输出<key, value>,其中key为网站编号，value为相乘结果

代码实现 PageRank_mapper.py

```
#encoding:utf8
import sys

#每一行格式为前面是Mi的转置，最后是vi
for line in sys.stdin:
    data = line.strip().split()
    Mi = data[:-1]
    Vi = float(data[-1])
    for i in range(len(Mi)):
        value = float(Mi[i])*Vi
        print "%d %f" % (i+1,value)
```

reducer设计

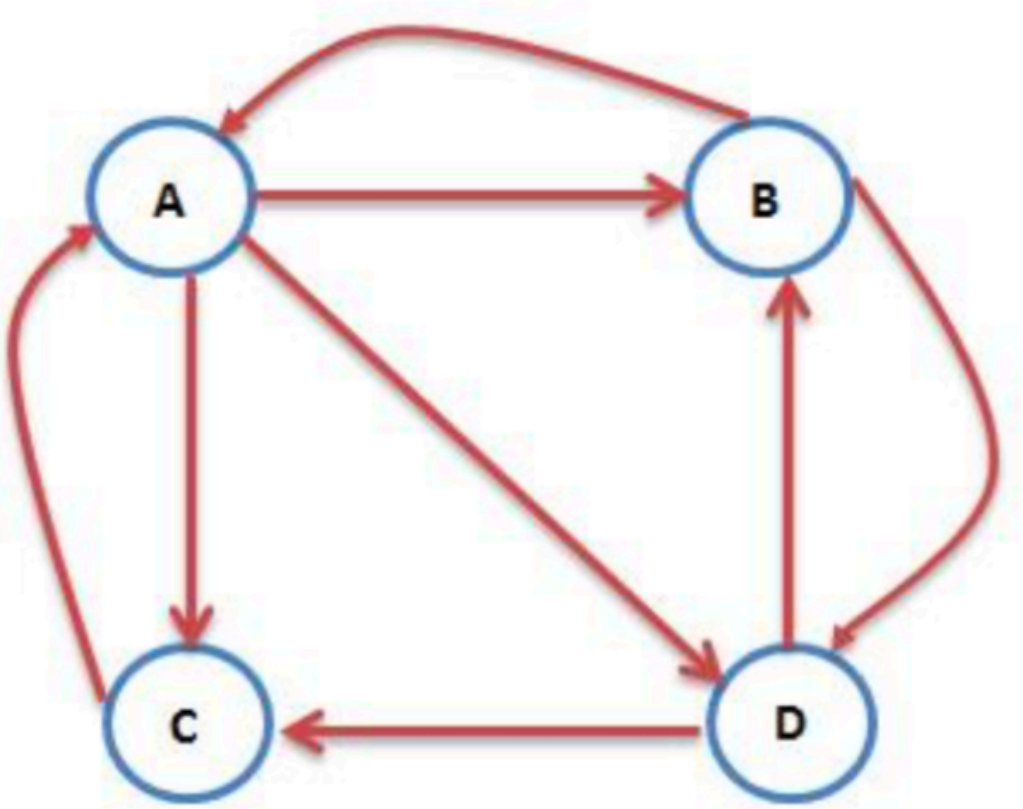
对于mapper输出的结果排序后输入reducer，将相同key的value进行累加，累加结果再进行处理（如原理中描述,这里的α取0.8），防止迭代不收敛或者收敛到一个网页

代码实现 PageRank_reducer.py

```
#encoding:utf8
import sys
last = None
values = 0.0
alpha = 0.8
N = 4 #网页的数量
for line in sys.stdin:
    data = line.strip().split()
    key,value = data[0],float(data[1])
    if key != last:
        if last:
            values = alpha * values + (1 - alpha) / N
            print values
        last = key
        values = value
    else:
        values += value
if last:
    values = alpha * values + (1 - alpha) / N
    print values
```

性能评测

为了方便观察收敛还是采用实例中的矩阵，先对其进行数据预处理，MT.txt存放矩阵转置，V.txt存放的是初始均等的rank，每迭代一次会刷新一次V.txt

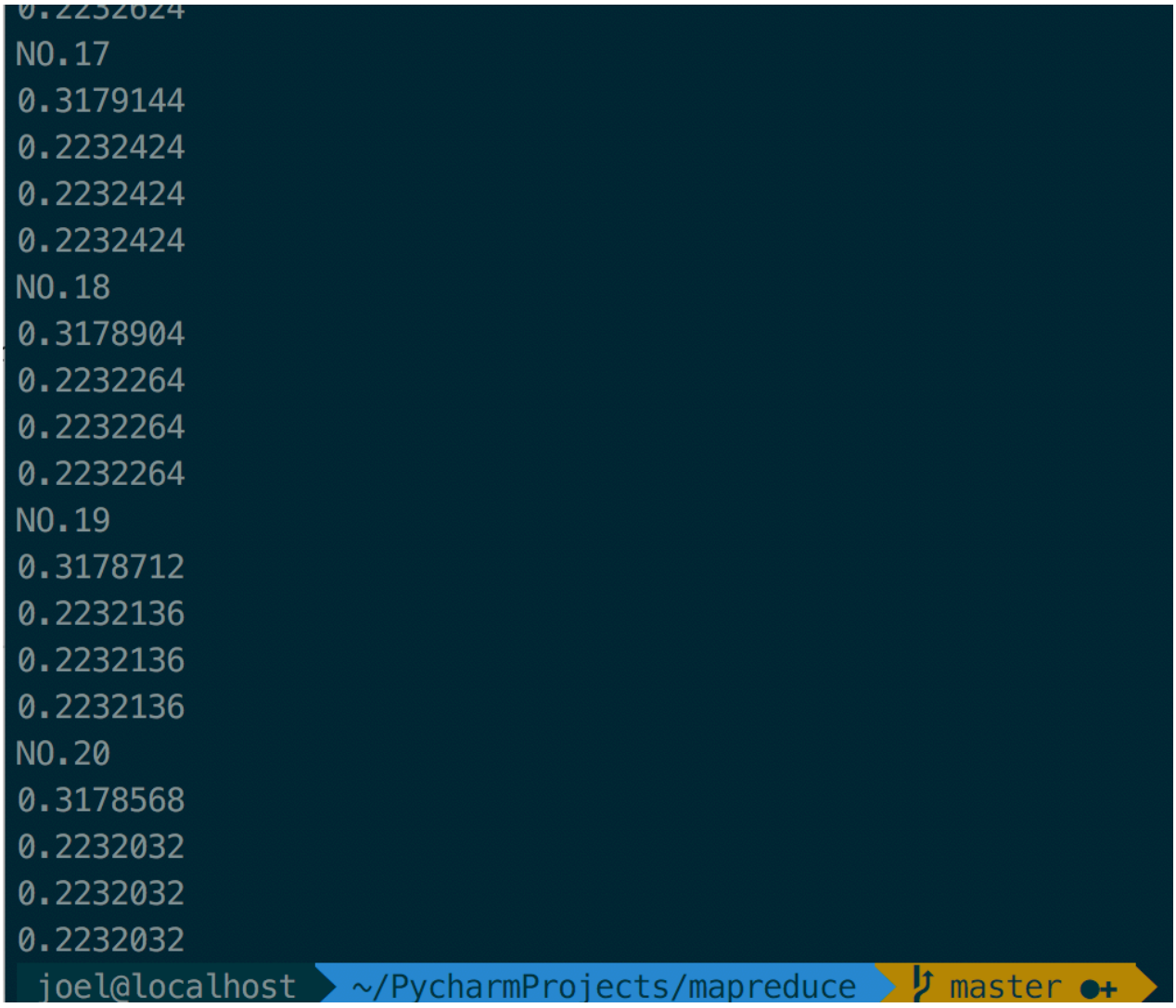


在自己电脑上利用bash脚本对程序进行迭代20次

Test.sh

```
#!/bin/bash
count=20
for i in `seq 1 $count`
do
        echo "NO.$i"
        cat V.txt
        python PageRank_pretreatment.py
        cat MTV.txt | python PageRank_mapper.py | sort -k1,1 |python
PageRank_reducer.py >V.txt
done
```

观察输出



从输出可知到20次时结果已经收敛，因为矩阵很小所以时间忽略，由于找不到大的pagerank矩阵数据，所以在这里只验证程序的正确性，因为是严格按照助教给的思路来实现，所以性能应该不差。