作业 2

1 导入和导出数据

1. 从本地导入数据到 HDFS

在 HDFS 本人的目录中新建一个文件夹用于保存文件：

```
2018210904@thumm01:~$ hadoop fs -mkdir /dsjxtjc/2018210904/data
```

将本地文件 file_1 复制到 HDFS 上刚才创建的文件夹中：

```
2018210904@thumm01:~$ hadoop fs -copyFromLocal ./file_1 /dsjxtjc/2018210904/data
/file_1
```

查看确认文件导入成功：

```
2018210904@thumm01:~$ hadoop fs -ls /dsjxtjc/2018210904/data
Found 1 items
-rw-r--r--   2 2018210904 dsjxtjc        273 2018-11-20 16:01 /dsjxtjc/201821090
4/data/file_1
```

2. 将 HDFS 上的数据导出到本地

将刚才上传的 file_1 导出到本地，命名为 a_file：

```
2018210904@thumm01:~$ hadoop fs -copyToLocal /dsjxtjc/2018210904/data/file_1 ./a
_file
```

在本地查看确认文件导出成功：

```
2018210904@thumm01:~$ ls -l
total 14981736
-rw-r--r-- 1 2018210904 dsjxtjc        273 Nov 20 16:32 a_file
```

2 在 Hadoop 上使用 MapReduce 进行字符统计

1. 用 java 实现字符统计

按照实验指导书上的步骤在 Hadoop 上使用 MapReduce 进行字符统计，将运行结果保存到 HDFS 上的/dsjxtjc/2018210904/wc_output 中：

```
2018210904@thumm01:~$ hadoop com.sun.tools.javac.Main WordCount.java
2018210904@thumm01:~$ jar cf wc.jar WordCount*.class
2018210904@thumm01:~$ hadoop jar wc.jar WordCount /dsjxtjc/wc_data.txt /dsjxtjc/
2018210904/wc_output
```

运行结束后，将远程结果复制到本地查看：

```
2018210904@thumm01:~$ hadoop fs -copyToLocal /dsjxtjc/2018210904/wc_output /home
/dsjxtjc/2018210904/wc_output
```

在本地查看 wc_output 中的 part-r-00000 文件的前十行：

```
2018210904@thumm01:~/wc_output$ head part-r-00000
'tis    1000
Again   1000
Alabama 3000
Alleghenies     1000
Almighty        1000
America 5000
American        4000
And     12000
But     4000
California      1000
```

2. 用 python 实现字符统计

也可以使用 Hadoop 流的 API 用 python 实现 MapReduce，首先用 python 写一个 mapper.py 和 reducer.py 实现 map 和 reduce 的处理逻辑，然后在本地进行测试，测试成功之后在 Hadoop 上运行。运行脚本为 run.sh。mapper.py 和 reducer.py 内容如下：

```
2018210904@thumm01:~$ cat mapper.py
#!/usr/bin/env python
import sys
for line in sys.stdin:
        line=line.strip()
        print("%s\t%s"% (line,1))
```

```
2018210904@thumm01:~$ cat reducer.py
#!/usr/bin/env python
import sys
lastkey=None
t=0
for line in sys.stdin:
        (key,val)=line.strip().split("\t")
        if lastkey and key!=lastkey:
                print("%s\t%s" % (lastkey,t))
                lastkey=key
                t=1
        else:
                lastkey=key
                t+=1
```

脚本 run.sh 内容如下：

```
2018210904@thumm01:~$ cat run.sh
#!/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/us
r/local/games:/snap/bin:/opt/server/jdk1.7.0_79/bin:/data/hadoop-2.7.6/bin:/data
/spark-2.1.0-bin-hadoop2.7//bin
export PATH
hadoop fs -rm -r /dsjxtjc/2018210904/wc_py_output
hadoop jar /data/hadoop-2.7.6/share/hadoop/tools/lib/hadoop-streaming-2.7.6.jar
-input /dsjxtjc/wc_data.txt -output /dsjxtjc/2018210904/wc_py_output -mapper "ma
pper.py" -reducer "reducer.py" -file /home/dsjxtjc/2018210904/mapper.py -file /h
ome/dsjxtjc/2018210904/reducer.py
%config InlineBackend.figure_format='retina'
%config InlineBackend.figure_format='retina'
```

运行脚本 run.sh，得到的结果保存在 HDFS 中的/dsjxtjc/2018210904/wc_py_output 中。把该文件夹从 HDFS 复制到本地查看结果：

```
2018210904@thumm01:~$ hadoop fs -copyToLocal /dsjxtjc/2018210904/wc_py_output /h
ome/dsjxtjc/2018210904
```

```
2018210904@thumm01:~/wc_py_output$ head part-00000
'tis    1000
Again   1000
Alabama 3000
Alleghenies     1000
Almighty        1000
America 5000
American        4000
And     12000
But     4000
California      1000
```

可以发现两个实验都得到了正确的结果。

3 设计自己的 DFS+MapReduce 框架，统计均值和方差

1. 系统的基本配置参数

DFS：thumm01 是 namenode(管理节点)，thumm02-05 是 datanode(数据节点)，每个数据块大小设置为 108MB，提供的命令行接口有 upload_file(上传文件)，download_file(下载文件)，ls(列出文件)，delete(删除文件)，exit(退出)。

MapReduce 框架：thumm01 向 thumm02-05 提交 map 任务，thumm02-05 分别利用本地存储的文件块执行 map 任务，并将得到的结果传给 thumm01，在 thumm01 上执行 reduce 任务并得到最终结果。因此有 4 台机器并行执行 map 任务，1 台机器执行 reduce 任务。

数据：数据文件为 number_dataset，大小为 3.5G，内容为每行一个浮点数，总任务是求出该文件中所有数的均值和方差。

2. 系统的整体框架

该系统一共包含 5 个.py 文件，其功能概述，运行位置和相互调用关系如下表所示：

文件名	功能概述	运行位置	调用文件
control_all.py	在主节点上负责对整体流程的控制	01	my_dfs.py my_reduce.py
my_dfs.py	搭建一个 DFS	01	无
map_master.py	在任务节点上负责对 map 流程的控制	02-05	my_map.py
my_map.py	通用的 map 类	02-05	无
my_reduce.py	通用的 reduce 类	01	无

以上 5 个.py 文件和数据文件初始时均位于 thumm01 上，thumm02-05 初始时没有任何文件，运行时在 thumm01 上执行'python3 control_all.py 文件名' 命令即可。control_all.py 中执行的流程大致如下：

调用 my_dfs.py 实例化一个 DFS 系统和一个 DFS 客户端 → 把数据文件写入 DFS 系统中 → 给每个节点发送需要处理的文件块列表信息和 my_map.py，map_master.py 三个文件 → 通过远程命令在 4 个节点上执

行 python map_master.py 命令，实现并行的 map 过程 → 对 4 个节点发回的结果执行 reduce 任务，得到最终结果

下面详细解释一下以上的每个步骤：

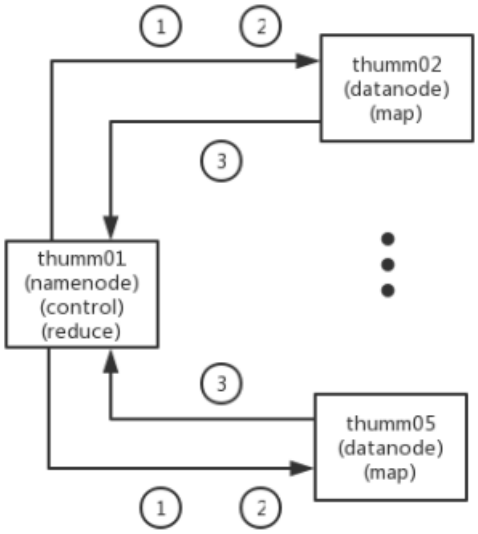
① 调用 my_dfs.py 实例化一个 DFS 系统和一个 DFS 客户端并写入数据文件：假设要处理的数据文件并没有事先写入 DFS 中，运行 control_all.py 时才实例化一个 DFS 系统和一个 DFS 客户端，通过该客户端的 write 方法把数据文件写到 DFS 中。此过程中实现文件的分块，每个数据节点上都存有数量基本相等的文件块，每个文件块复制三份。为了简化程序，三份文件存放在相邻的三个数据节点上 (05 和 02 看作相邻)，即如果映射表中某文件块映射到 thumm02，则 03 和 04 中也复制一份，复制的文件位置不写入映射表中，映射表中每个文件块只对应一个数据节点。三份文件都是从 01 分别传给三个数据节点。

② 给每个节点发送需要处理的文件块列表信息和 my_map.py，map_master.py 三个文件：因为文件冗余，所以每个数据节点并不需要对其上存放的所有文件块都进行 map 操作，因此我们要告知每个节点需要处理哪些文件块，令每个节点处理映射表中与其对应的文件块即可。因此需要从映射表中找出每个节点对应的文件块，把它们的标识符列表写入一个文件，对于第 i 个节点，该文件名为 processi。将这 4 个文件分别传送给相应的节点，然后再将 my_map.py，map_master.py 传送给所有数据节点。这样，每个数据节点上都有数据块、需要处理的数据块名称、处理的程序。

③ 通过远程命令在 4 个节点上执行 python map_master.py 命令，实现并行的 map 过程：此时准备工作已经完成，通过 subprocess.Popen 向 4 个数据节点发送 'python map_master.py' 命令执行 map 任务。由于 python 中 subprocess 开启的子进程是并行运行的，所以 4 台机器上并行执行 map 任务。每个节点执行完 map 任务后把结果传送回 thumm01，第 i 个节点传回的结果文件是 resultprocessi。

④ 对 4 个节点发回的结果执行 reduce 任务，得到最终结果：等到 4 个子进程都执行完之后，thumm01 接收到 4 个 map 后的结果文件，然后对 map 的结果进行 reduce 任务，得到最终的结果，写入 finalresult.txt 文件。

整体流程中的文件传输如下图所示：



①：01 作为 namenode 把文件块分发给 datanode，实现数据文件写入 DFS

②：01 作为 MapReduce 主节点，向 02-05 发送 process2-process5，my_map.py，map_master.py

③：02-05 执行完 map 任务后把结果 resultprocess2-resultprocess5 发送给 01，01 执行 reduce 任务

3. 每个部分的具体实现

DFS：DFS 的建立在 my_dfs.py 文件中实现。该文件中包含的类如下：

class Client：实例化 Client 类相当于建立了一个使用 DFS 的用户，可以通过 Client 类在 DFS 上执行上传、下载、删除等操作。

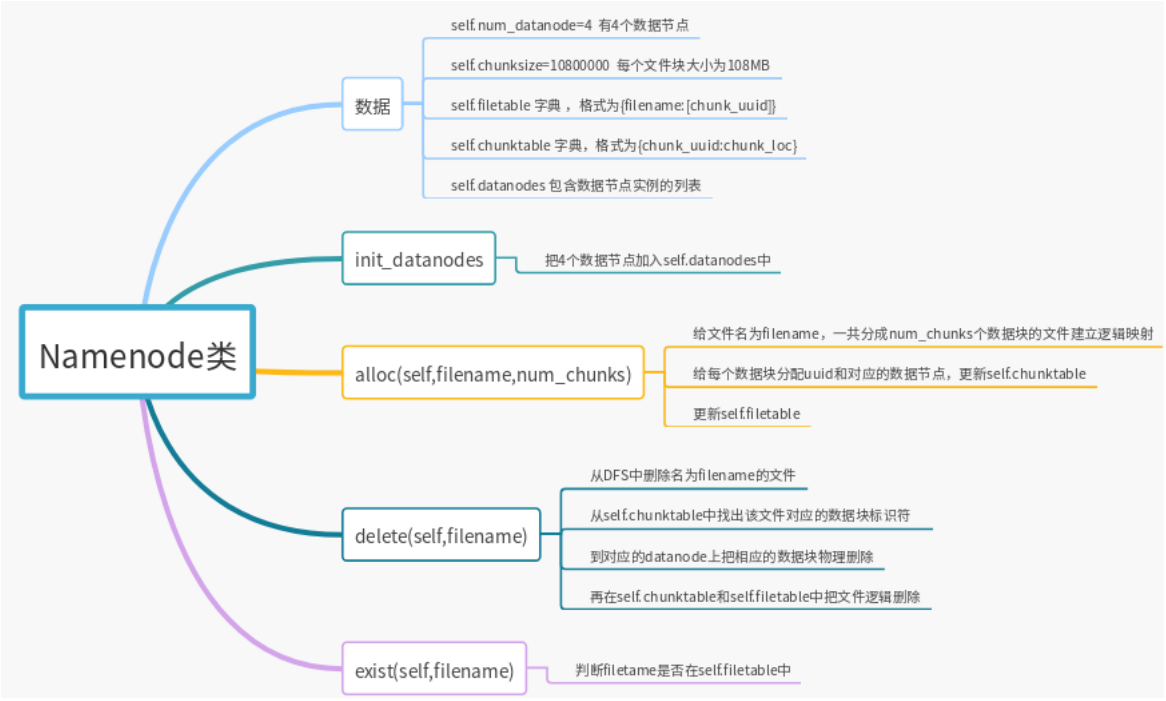
class Namenode：元节点类。实例化 Namenode 类相当于建立了一个 DFS。它主要维护 DFS 中的文件名与文件块标识符、文件块标识符与存储的数据节点位置之间的逻辑映射关系。

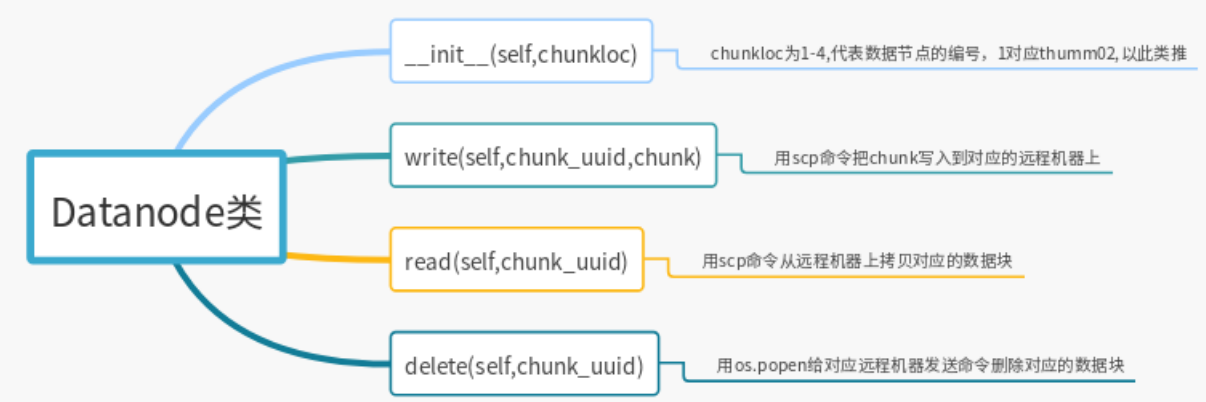
class Datanode：数据节点类。实例化 Datanode 相当于把一台远程机器作为 DFS 中的一个数据节点。它主要实现文件块的物理存储、读取、删除等。

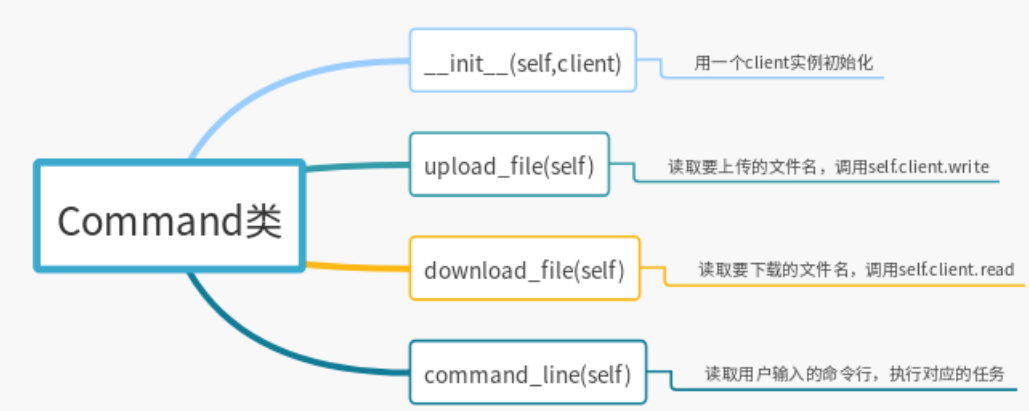
class Command：该类提供了一个让人通过 Client 类使用 DFS 的命令行接口。通过 Command 类可以根据人的命令通过调用 Client 类中的方法实现与 DFS 的交互。

每个类的具体设计如下：


Client类
__init__(self,namenode)
在以namenode为元节点的DFS上建立一个用户类
write(self,filename,data)
把data写入到DFS中，文件名为filename
更新namenode中的逻辑映射
把数据写入到对应的datanode中
read(self,filename)
把文件名为filename的文件从DFS中读取出来
从namenode映射表中找出该文件对应的数据块标识符
到对应的datanode上读取数据块内容并拼接在一起
delete(self,filename)
从DFS中删除名为filename的文件
从namenode映射表中找出该文件对应的数据块标识符
到对应的datanode上把相应的数据块物理删除，再在namenode中把文件逻辑删除


Namenode类
数据
self.num_datanode=4 有4个数据节点
self.chunksize=10800000 每个文件块大小为108MB
self.filetable 字典，格式为{filename:[chunk_uuid]}
self.chunktable 字典，格式为{chunk_uuid:chunk_loc}
self.datanodes 包含数据节点实例的列表
init_datanodes
把4个数据节点加入self.datanodes中
alloc(self,filename,num_chunks)
给文件名为filename，一共分成num_chunks个数据块的文件建立逻辑映射
给每个数据块分配uuid和对应的数据节点，更新self.chunktable
更新self.filetable
delete(self,filename)
从DFS中删除名为filename的文件
从self.chunktable中找出该文件对应的数据块标识符
到对应的datanode上把相应的数据块物理删除
再在self.chunktable和self.filetable中把文件逻辑删除
exist(self,filename)
判断filetame是否在self.filetable中





以上就是 DFS 的具体实现。

MapReduce 框架:

首先介绍对任务拆分的原理：该任务是求一个大文件中所有数据的均值和方差，假设这一组数据为 x_1,x_2,...,x_n, 则均值为

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

方差为

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \mu^2$$

则每个 map 输入文件块，输出 [数据个数 n，数据的和 s_1，数据的平方和

s_2]。reduce 根据所有 map 输出结果计算均值：

$$\mu = \frac{\sum_{map} s_1}{\sum_{map} n}$$

方差：

$$\sigma^2 = \frac{\sum_{map} s_2}{\sum_{map} n} - \mu^2$$

输出结果为 $[\mu, \sigma^2]$。因此 map 可以有很多个，reduce 只需要一个。

在本次任务中，并行处理主要体现在以下两个方面：1. 主控制程序通过 subprocess.Popen 建立 4 个并行的子进程给 4 台远程主机发送 map 指令，使 4 台远程主机实现并行处理。2. 在每台远程主机上控制 map 流程的程序中通过 multiprocess 每一时刻并行进行 5 个文件块的 map 操作。这样，每台远程主机最后生成一个名为 resultprocessi(i=2-5) 的结果文件，文件内容是一个字典，字典的每一项对应一个文件块的 map 结果，即格式为 {序号：[数据个数 n，数据的和 s_1，数据的平方和 s_2]}。一台远程主机上所有需要 map 的文件块都处理完之后，远程主机把结果写入结果文件，发送给 thumm01。thumm01 获得 4 个结果文件后，对其进行 reduce 操作，求出均值和方差，写入 finalresult.txt 文件。

对于每个文件块 map 过程的实现，在 my_map.py 中定义了 Map_worker 类，该类用于执行一个任务，调用该类的 RunMapper 方法可以获得 self.output。该类的初始化需要一个 mapper_class，即用户自己根据不同任务编写的 mapper 类，本任务中即为 map_master.py 中的 CalmeanvarMapper 类。Mapper 类是一个给用户提供的接口，用户自己编写的 mapper 类需要继承 Mapper，重载 map 方法。这里主要是为了保证程序的复用性，如果执行其他任务只需要重写 mapper 类即可。my_reduce.py 和 CalmeanvarReducer 类与此同理。在 map_master.py 中，对于一个文件块进行 map 处理需要调用 do_map 函数，运行一个 Map_worker.RunMapper，再读取 worker.output 即可。

在完整的处理流程中涉及多次数据读写，中间需要进行很多次数据处理，比如从读到的字符串中提取某些有用信息，把字符串数据转成浮点数列表等等，map_master.py 中的 convert_float，pre_strlist，pre_data，control_all.py 中的 pre_dict 函数都是用来进行数据处理从而得到某些特定格式数据的函数。这里不详细介绍它们的实现，可以通过具体代码和注释理解它们的原理。

4. 实验结果及分析

这一部分对实验的中间结果及最终结果进行分析。

数据文件为 number_dataset，大小为 3.5GB，该文件每一行为一个浮点数。

```
-rw-r--r-- 1 2018210904 dsjxtjc 3.5G Nov  1 12:03 number_dataset
```

```
2018210904@thumm01:~$ head number_dataset
10.35
24.77
34.85
7.85
44.90
8.65
21.18
6.05
5.85
6.45
```

在 thumm01 上运行 control_all.py：

```
2018210904@thumm01:~$ python3 control_all.py number_dataset
```

输出 number_dataset 写入 DFS 后 namenode 上的 file_table 和 chunk_table（截图仅截取一部分）：

```
{'number_dataset': [UUID('016ff7a8-ed17-11e8-8ed0-74a4b5005e55'), UUID('016ff7a9
-ed17-11e8-8ed0-74a4b5005e55'), UUID('016ff7aa-ed17-11e8-8ed0-74a4b5005e55'), UU
ID('016ff7ab-ed17-11e8-8ed0-74a4b5005e55'), UUID('016ff7ac-ed17-11e8-8ed0-74a4b5
005e55'), UUID('016ff7ad-ed17-11e8-8ed0-74a4b5005e55'), UUID('016ff7ae-ed17-11e8
```

可以发现 file_table 是一个字典，key 是文件名，value 是一个包含所有文件块标识符的列表。

```
{UUID('016ff7aa-ed17-11e8-8ed0-74a4b5005e55'): 3, UUID('016ff7b6-ed17-11e8-8ed0-
74a4b5005e55'): 3, UUID('016ff7ba-ed17-11e8-8ed0-74a4b5005e55'): 3, UUID('129212
48-ed17-11e8-8ed0-74a4b5005e55'): 4, UUID('12921235-ed17-11e8-8ed0-74a4b5005e55'
): 1, UUID('1292123c-ed17-11e8-8ed0-74a4b5005e55'): 4, UUID('12921236-ed17-11e8-
8ed0-74a4b5005e55'): 2, UUID('12921240-ed17-11e8-8ed0-74a4b5005e55'): 4, UUID('1
```

可以发现 chunk_table 是一个字典，key 是文件块标识符，value 是数据节点代号，表示该文件块存放的位置。

下图是 thumm01 向各个数据节点发送数据块的过程 (tmp.txt 是本地的临时文件，每个数据块先写入到本地的 tmp.txt 中):

```
tmp.txt                                       100%  108MB 107.8MB/s   00:00
tmp.txt                                       100%  108MB 107.8MB/s   00:00
tmp.txt                                       100%  108MB 107.8MB/s   00:00
tmp.txt                                       100%  108MB 107.8MB/s   00:00
tmp.txt                                       100%  108MB 107.8MB/s   00:01
tmp.txt                                       100%  108MB 107.8MB/s   00:01
tmp.txt                                       100%  108MB 107.8MB/s   00:01
tmp.txt                                       100%  108MB 107.8MB/s   00:01
```

传输完数据后向每个数据节点传送 processi，my_map.py，map_master.py 三个文件:

```
process2                                      100%  736     0.7KB/s   00:00
my_map.py                                     100%  495     0.5KB/s   00:00
map_master.py                                 100% 2538     2.5KB/s   00:00
process3                                      100%  736     0.7KB/s   00:00
my_map.py                                     100%  495     0.5KB/s   00:00
map_master.py                                 100% 2538     2.5KB/s   00:00
process4                                      100%  736     0.7KB/s   00:00
my_map.py                                     100%  495     0.5KB/s   00:00
map_master.py                                 100% 2538     2.5KB/s   00:00
process5                                      100%  690     0.7KB/s   00:00
my_map.py                                     100%  495     0.5KB/s   00:00
map_master.py                                 100% 2538     2.5KB/s   00:00
```

等到程序执行完之后，发现本地多了 resultprocess2-5，这些是 map 的结果文件，最终的结果文件是 finalresult.txt:

```
-rw-r--r-- 1 2018210904 dsjxtjc        868 Nov 21 07:08 resultprocess2
-rw-r--r-- 1 2018210904 dsjxtjc        867 Nov 21 07:08 resultprocess3
-rw-r--r-- 1 2018210904 dsjxtjc        869 Nov 21 07:08 resultprocess4
-rw-r--r-- 1 2018210904 dsjxtjc        816 Nov 21 07:08 resultprocess5
```

```
-rw-r--r-- 1 2018210904 dsjxtjc         39 Nov 21 07:08 finalresult.txt
```

查看 finalresult.txt 中的最终结果:

```
2018210904@thumm01:~$ cat finalresult.txt
[14.847357033611589, 3111.205607473459]2018210904@thumm01:~$
```

发现最终结果存放在一个列表中，均值为 14.847357033611589，方差为 3111.205607473459。

查看 thumm02 传回的 map 结果 resultprocess2 中的内容：

```
2018210904@thumm01:~$ cat resultprocess2
{0: [10799909, 160418085.54044804, 36156436856.74874], 1: [10799901, 160262788.3
4043053, 35487809437.50528], 2: [10799908, 160285105.9304471, 35615823500.49227]
, 3: [10799908, 160294944.79044342, 35741442192.60332], 4: [10799914, 160409621.
6504317, 36426152283.82185], 5: [10799916, 160404262.3504249, 36451342807.331055
], 6: [10799910, 160300637.520454, 35798962915.40765], 7: [10799912, 160310899.5
904259, 35527893185.028275], 8: [10799901, 160293903.81042948, 35691720915.05773
], 9: [10799910, 160273476.21042898, 35381508594.15448], 10: [10799903, 16027301
0.11043566, 35702844838.746315], 11: [10799904, 160382061.45047176, 36204367155.
48242], 12: [10799911, 160330176.7904274, 35918439149.17336], 13: [10799909, 160
513137.57043585, 36849377585.20305], 14: [10799911, 160364094.45043984, 36176549
410.559265], 15: [10799912, 160343810.55044916, 35974182506.615204]}2018210904@t
humm01:~$
```

发现 map 的结果是一个字典，每一项代表一个文件块的处理结果，key 是序号，value 是 [数据个数 n，数据的和 s_1，数据的平方和 s_2]。每台远程机器上处理 16 个文件块。

登录到 thumm02 上，列出/data/dsjxtjc/2018210904 文件夹下的文件：

```
[2018210904@thumm02 2018210904]$ ls -l
total 5168744
-rw-r--r-- 1 2018210904 dsjxtjc 113004253 Nov 21 06:51 016ff7a8-ed17-11e8-8ed0-7
4a4b5005e55
-rw-r--r-- 1 2018210904 dsjxtjc 113004432 Nov 21 06:52 016ff7aa-ed17-11e8-8ed0-7
4a4b5005e55
-rw-r--r-- 1 2018210904 dsjxtjc 113004509 Nov 21 06:52 016ff7ab-ed17-11e8-8ed0-7
4a4b5005e55
-rw-r--r-- 1 2018210904 dsjxtjc 113003702 Nov 21 06:52 016ff7ac-ed17-11e8-8ed0-7
4a4b5005e55
```

```
-rw-r--r-- 1 2018210904 dsjxtjc  94503923 Nov 21 07:05 1292124b-ed17-11e8-8ed0-7
4a4b5005e55
-rw-r--r-- 1 2018210904 dsjxtjc      2538 Nov 21 07:05 map_master.py
-rw-r--r-- 1 2018210904 dsjxtjc       495 Nov 21 07:05 my_map.py
-rw-r--r-- 1 2018210904 dsjxtjc      1260 Nov 21 07:05 my_map.pyc
-rw-r--r-- 1 2018210904 dsjxtjc       736 Nov 21 07:05 process2
-rw-r--r-- 1 2018210904 dsjxtjc       868 Nov 21 07:07 resultprocess2
```

说明文件传输过程是正确的。

实验证明程序按照所设计的方式正确运行，最终得到数据的均值和方差。

5. 任务错误处理机制

本程序中的任务错误处理假设错误是某数据节点上需要执行 map 任务的数据块丢失。这种情况下，在该数据节点上运行 map_master.py 时会触

发 IOError。因此可以在 map_master.py 中捕获该错误并进行处理。处理逻辑是如果读取文件块时出现 IOError，就到两个相邻的机器上拷贝一份该文件，这样只要另外两份备份文件至少存在一份，map 就能正常运行。相关代码如下：

```
#对每一个文件块执行map任务,包含容错机制。如果发现数据块丢失,到相邻的两个节点中复制过来
def do_map(chunk_uuid):
    try:
        with open('%s' % chunk_uuid,'r') as fr:
            input_data=fr.read()
    except IOError:
        os.system('scp 2018210904@thumm0%s:/data/dsjxtjc/2018210904/%s /data/dsjxtjc/2018210904' % ((int(my_id)%4+1),(chunk_uuid)))
        os.system('scp 2018210904@thumm0%s:/data/dsjxtjc/2018210904/%s /data/dsjxtjc/2018210904' % ((int(my_id)%4+2),(chunk_uuid)))
        with open('%s' % chunk_uuid,'r') as fr:
            input_data=fr.read()
    input_data=pre_data(input_data)
    worker=my_map.Map_worker(CalmeanvarMapper,input_data)
    worker.RunMapper()
    return worker.output
```

对该任务错误处理机制进行测试，在 control_all.py 中加入一段代码：

```
#以下这段代码是把存放在thumm02上的文件块删掉,用于测试容错机制
#    rm_list=pre_strlist(str([k for (k,v) in client.namenode.chunktable.items() if v==1]))
#    for i in rm_list:
#        str3='ssh 2018210904@thumm02 "cd /data/dsjxtjc/2018210904;rm %s"' % (i)
#        p3=subprocess.Popen(str3,shell=True)
#        p3.wait()
```

这段代码把 thumm02 上要处理的文件块全部删掉。加入这段代码，运行 control_all.py，运行结束后，查看 resultprocess2：

```
2018210904@thumm01:~$ cat resultprocess2
{0: [10799912, 160310899.5904259, 35527893185.028275], 1: [10799911, 160364094.4
5043984, 36176549410.559265], 2: [10799903, 160273010.11043566, 35702844838.7463
15], 3: [10799901, 160293903.81042948, 35691720915.05773], 4: [10799904, 1603820
61.45047176, 36204367155.48242], 5: [10799911, 160330176.7904274, 35918439149.17
336], 6: [10799901, 160262788.34043053, 35487809437.50528], 7: [10799916, 160404
262.3504249, 36451342807.331055], 8: [10799914, 160409621.6504317, 36426152283.8
2185], 9: [10799910, 160300637.520454, 35798962915.40765], 10: [10799908, 160285
105.9304471, 35615823500.49227], 11: [10799910, 160273476.21042898, 35381508594.
15448], 12: [10799909, 160418085.54044804, 36156436856.74874], 13: [10799908, 16
0294944.79044342, 35741442192.60332], 14: [10799909, 160513137.57043585, 3684937
7585.20305], 15: [10799912, 160343810.55044916, 35974182506.615204]}2018210904@t
humm01:~$
```

发现仍然可以得到正确的 map 结果。查看最终结果 finalresult.txt:

```
2018210904@thumm01:~$ cat finalresult.txt
[14.847357033611589, 3111.205607473459]2018210904@thumm01:~$
```

最终结果并没有受到影响。说明任务错误处理机制能够正常工作。

4 协同过滤算法

1. 协同过滤算法的单机版

协同过滤算法包括基于用户和基于物品两种，本次实现的是基于用户的协同过滤算法，基本思路是计算用户之间的相似度，取相似度最高的 k 个邻居，根据邻居与当前用户的相似度和邻居对物品的打分来预测当前用户未涉及物品的打分，给用户推荐打分最高的前 n 个物品。

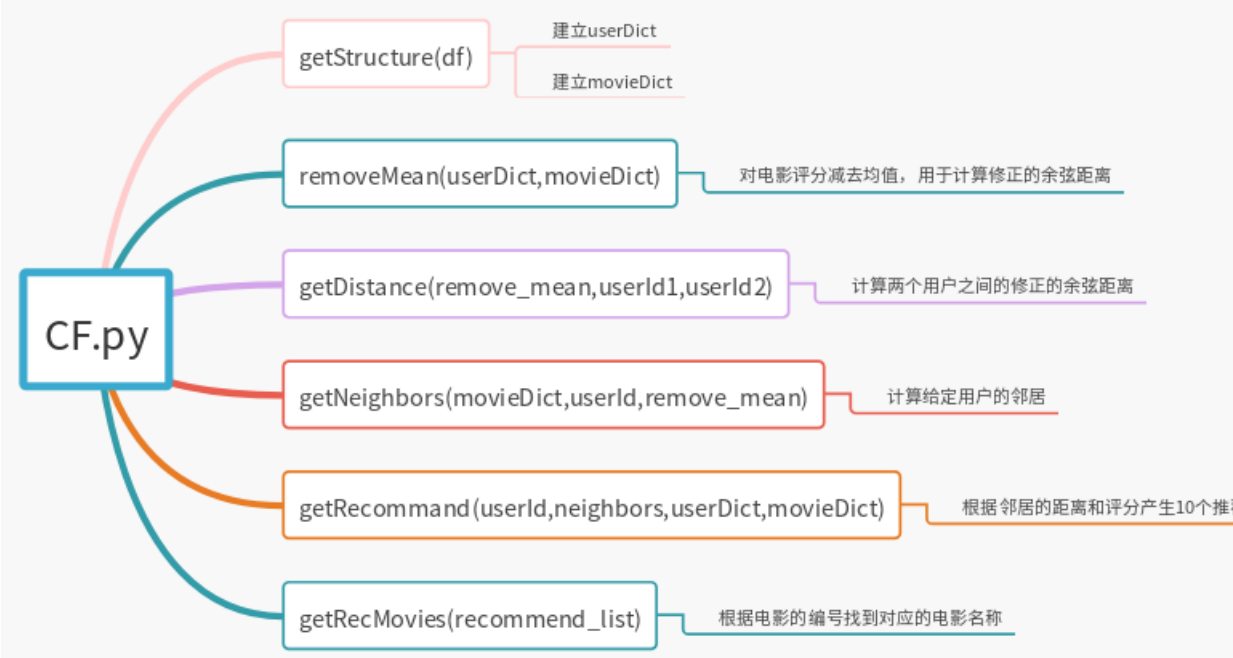
本次实验中采用的数据集是 http://grouplens.org/datasets/movielens/上下载的电影评分数据集。主要使用的两个文件是 small_ratings.csv 和 small_movies.csv。small_ratings.csv 每行包括 userId,movieId,ratings,timestamp，small_movies.csv 每行包括 movieId,title 等信息。对于单机版的基于用户的协同过滤算法描述如下：

1. 根据 small_ratings.csv 中的信息构建基于 userId 的字典 userDict 和基于 movieId 的字典 movieDict。userDict 的结构为 {userId:[[movieId1,rating1],[movieId2,rating2]...]}，movieDict 的结构为 {movieId:[[userId1,rating1],[userId2,rating2]...]}。

2. 寻找给定 userId 的 k 个邻居，本实验中 k 取 20。对于给定的 userId，在 userDict 中找到该 userId 评分过的所有电影，对于每个电影到 movieDict 中找出对该电影评分过的用户，这些用户都作为潜在的邻居放入一个列表中，然后计算这些潜在的邻居与当前 userId 的相似度，相似度采用修正的余弦相似度，然后进行排序，选相似度前 20 的用户作为邻居。

3. 根据邻居的电影评分产生推荐。对于 20 个邻居，遍历他们评分过的所有电影，每个电影的预测评分为 $\sum_i sim(neighbor_i) * rate_i$，即所有评分过该电影的邻居的评分与相似度乘积之和。然后进行排序，取前十个电影作为推荐。

该算法的具体实现在 CF.py 中，其包含的函数及功能如下图所示：



运行时，需要输入 python CF.py userId，给 userId 产生 10 个推荐的电影：

```
(venv) 2018210904@thumm01:~$ python CF.py 20
['Legends of the Fall (1994)']
['While You Were Sleeping (1995)']
['Matrix, The (1999)']
['Star Wars: Episode I - The Phantom Menace (1999)']
['Apollo 13 (1995)']
['Batman Forever (1995)']
["Mr. Holland's Opus (1995)"]
['Bad Boys (1995)']
['Dead Man Walking (1995)']
['Airheads (1994)']
```

2. 多机版协同过滤算法

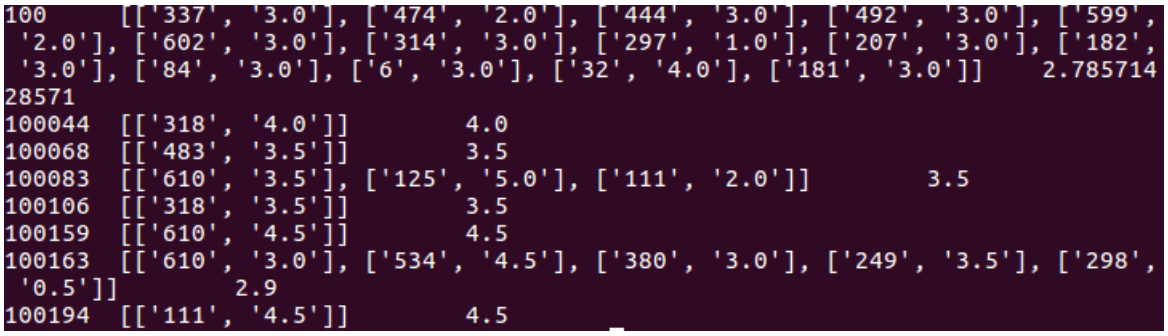
协同过滤算法中很多步骤都可以拆分成多机协同处理，所以多机版的协同过滤算法需要经过多个 map 和多个 reduce 过程串行处理后得到结果。有很多种拆分方式，根据单机版协同过滤算法，可以通过多机协同计算 userDict，movieDict，movieDict 每一项中的均值，userDict 中每个电影评分减去均值。由于时间仓促，这里只实现了用 MapReduce 求 movieDict 和每一项的评分均值，输入为 small_ratings.csv，输出的每一行

为 movieId,[[userId1,rating1],[userId2,rating2]...], 该 movieId 的平均评分。mapper 和 reducer 分别为 mapper_1.py 和 reducer_1.py，其内容如下图：

```
(venv) 2018210904@thumm01:~$ cat mapper_1.py
#!/usr/bin/env python
import sys
for line in sys.stdin:
        data=line.strip().split(",")
        key=data[1]
        if key=='movieId':
                continue
        else:
                print('%s\t%s'%(key,[data[0],data[2]]))
```

```
#!/usr/bin/env python
import sys
lastkey=None
value=[]
summation=0
count=0
for line in sys.stdin:
        (key,val)=line.strip().split("\t")
        if lastkey and key!=lastkey:
                print("%s\t%s\t%s" % (lastkey,value,summation/count))
                lastkey=key
                value=[val.replace('[','').replace(']','').replace('\'','').repl
ace(' ','').split(',')]
                summation=float(val.replace('[','').replace(']','').replace('\''
,'').replace(' ','').split(',')[1])
                count=1
        else:
                lastkey=key
                value.append(val.replace('[','').replace(']','').replace('\'',''
).replace(' ','').split(','))
                count+=1
                summation+=float(val.replace('[','').replace(']','').replace('\'
','').replace(' ','').split(',')[1])
```

在 hadoop 上进行 MapReduce 之后，得到的结果如下 (截取部分)：



可以发现得到了指定格式的输出。

实际上还可以用其他思路进行拆分，比如通过 MapReduce 构造出所有用户之间的相似度矩阵，用 map 计算某一用户的所有邻居，reduce 计算该用户未涉及项目的评分等。由于时间关系，本次实验没有涉及以上内容。协同过滤算法需要经过多个 MapReduce 作业才能完成，需要多次读写磁盘，涉及很多冗余操作，作业成本较高。针对这些问题，可以利用 spark 在迭代计算和内存计算上的优势解决。对于 spark 的应用将在下次关于 spark 的作业中详细探讨。

注：DFS+MapReduce 任务中的 5 个.py 文件均写有详细注释。协同过滤算法任务中的 CF.py,mapper_1.py,reducer_1.py 由于逻辑比较简单，并没有写注释。所有 py 文件都在文件夹中。