

作业5 191840265 吴偲羿 用hadoop实现 wordCount

1.程序基本设计思路

作业讲义中有关wordCount其实已经非常清楚，大概花费50行左右的代码就可以实现最最基本的对于一个文本的词频统计。但是由于本次实验设计到1.多文本一起处理2.考虑标点符号，停词，数字与过短单词3.需要对输出进行排序，实际代码规模到达了300-400行。但是基本程序设计思路是没有变化的

主要的处理方法：

1.预处理

```
private Set<String> wordsToSkip = new HashSet<>(); //停词
private Set<String> punctuations = new HashSet<>(); // 标点符号
```

```
private void parseSkipFile(String fileName) { //parseSkip函数处理两个停词文件，分析并加入wordstoSkip与punctuations的集合
    try {
        fis = new BufferedReader(new FileReader(fileName));
        String temp = null;
        while ((temp = fis.readLine()) != null) {
            wordsToSkip.add(temp);
        }
    } catch (IOException ioe) {
        System.err.println("Error parsing the file " + StringUtils.stringifyException(ioe));
    }
}

private void parseSkipPunctuations(String fileName) {
    try {
        fis = new BufferedReader(new FileReader(fileName));
        String temp= null;
        while ((temp = fis.readLine()) != null) {
            punctuations.add(temp);
        }
    } catch (IOException ioe) {
        System.err.println("Error parsing the file" + StringUtils.stringifyException(ioe));
    }
}

//以上预处理结束
```

通过读取给定的停词，标点符号文件将需要跳过的单词写入有关列表中，以便之后词频统计时进行处理

2.正式词频统计（mapper）

我们通过TokenizerFileMapper打开输入文件夹，按照文件数量分配mapper，根据行 进行分词。由于需要分文件进行统计，设计时返回map的key用‘-’连接了单词与所在文件，以便区分，如下：

这里与示例程序思路是没有差别的，除了在统计特定单词时，需要对其进行是否需要skip的判断 如下：

```

while (itr.hasMoreTokens()) {
    String temp = itr.nextTokent(); //从迭代器中读取单词
    if(temp.length()>=3&&!Pattern.compile("^[-\\+]?[\\d]*$").matcher(temp).matches()&&!wordsToSkip.contains(temp)) {
        word.set(temp+"|"+fileName);
        context.write(word, one);
        Counter counter = context.getCounter(
            TokenizerFileMapper.CountersEnum.class.getName(),
            TokenizerFileMapper.CountersEnum.INPUT_WORDS.toString());
        counter.increment(1);
    }
}
}

```

其中

```
!Pattern.compile("^[-\\+]?[\\d]*$").matcher(temp).matches()
```

是用正则表达式对单词是否是数字 的判断。

这里reduce函数与示例函数没有差别。为了进行后面排序的处理，我们将此词频统计的结果输出至temp_part文件夹。作为下一任务 排序 的输入。

3.排序

通过上一步的mapreduce工作 我们获得了所有文件中的单词，单词所在文件以及该单词的词频。现在我们通过一个新的mapreduce任务对其进行排序。

我们了解 在hadoop中我们有job.setSortComparatorClass()设置的key比较函数类，描述如下：

```

/*
 * 进入同一个reduce的key是按照顺序排好的，该类使得：
 * 如果连续（注意，一定连续）的两条或多条记录满足同组（即compare方法返回0）的条件，
 * 即使key不相同，他们的value也会进入同一个values,执行一个reduce方法。
 * 相反，如果原来key相同，但是并不满足同组的条件，他们的value也不会进入一个valeur。
 * 最后返回的key是：满足这些条件的一组key中排在最后的那个。
 */

```

然而，如图所示，默认最后返回的key是排在最后的一个，而我们可以获得最大的词频，即降序排序，因此我们重写排序类：

```

private static class IntWritableDecreasingComparator extends IntWritable.Comparator {
    public int compare(WritableComparable a, WritableComparable b) { return -compare(a, b); }
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return -compare(b1, s1, l1, b2, s2, l2);
    }
}

```

这里 compare前的‘-’号 是因为compare根据比较大小输出-1，0和1，通过‘-’即可解决升序，降序问题。

这样，通过这个类，我们获得了之前对所有文件进行词频中，相同key(即相同文件中的相同单词)的最高value。下一步只需统计每个文件中前100的单词了。

我们定义一个hashmap<string,int>，来储存某一文件已输出的高频单词数，我们共需输出40个文件各100个单词，因此当hashmap中int的sum达到4000时，代表我们的任务就结束了。

这里我们需要先将key中的word与文件名拆分开：

```
String docId = val.toString().split( regex: "-")[1];
docId = docId.substring(0, docId.length()-4);
docId = docId.replaceAll( regex: "-", replacement: "");
String oneWord = val.toString().split( regex: "-")[0];
```

对于每一个docId，rank初始设置为1，每从一个docId中获得一个高频单词，就将该高频单词计数写入result并rank++，当rank达到100时，该文件统计结束，通过MultipleOutputs写入不同文件中，我们统一写入single-output文件夹。

```
if(rank < 100){
    rank += 1;
    map.put(docId, rank);
}
result.set(oneWord.toString());
String str=rank+": "+result+", "+key;
mos.write(docId, new Text(str), NullWritable.get() );
```

这是分文件的输出。

4.对于所有文件一起统计

和前面十分类似，唯一需要修改的是，mapper输出的key不需要体现单词所在的文件了，只需输出<word,one>即可：

```
word.set(temp);
context.write(word, one);
```

我们将其写入temp-all 文件夹，进行排序

同样的,在排序类里面，我们更改如下：

```
for(Text val: values){
    if(rank<=100) {
        result.set(val.toString());
        String str = rank + ": " + result + ", " + key;
        rank++;
        context.write(new Text(str), NullWritable.get());
    }
}
```

并将结果写入output文件夹。

这样，我们通过4个任务，分文件的统计、排序与总共的统计、排序，完成了程序要求的词频统计，下面展示运行结果

首先向hdfs传入input文件

```
quinton_541@MacBook-Air-de-541 bin % ./hdfs dfs -put /Users/quinton_541/IdeaProjects/WordCount/input input
```

执行程序：

```
quinton_541@MacBook-Air-de-541 WordCount % $HADOOP_HOME/bin/hadoop jar WordCount.jar input output -skip Exceptions/stop-word-list.txt Exceptions/punctuation.txt
```

这里 我们向程序传入了4个参数 input output分别指定了输入输出文件夹，-skip后两个文件为停词的文本文件，在main函数相关处理如下：

```
String[] remainingArgs = optionParser.getRemainingArgs();
if ((remainingArgs.length != 2) && (remainingArgs.length != 5)) {
    System.err.println("Check your parameters");
    System.exit(status: 2);
}
Job job = Job.getInstance(conf, jobName: "word count");//单文件词频统计
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerFileMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
List<String> otherArgs = new ArrayList<String>();
for (int i = 0; i < remainingArgs.length; ++i) {
    if ("-skip".equals(remainingArgs[i])) {
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.getConfiguration().setBoolean(name: "wordcount.skip.patterns", value: true);
    } else {
        otherArgs.add(remainingArgs[i]);
    }
}
FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
```

remainingArgs储存参数，remainingArgs[0],remainingArgs[1]保存输入输出路径，通过判断是否存在'-skip'参数，决定是否需要停词。

```
2021-10-29 22:24:29,304 INFO mapreduce.Job: Counters: 36
  File System Counters
    FILE: Number of bytes read=687856702
    FILE: Number of bytes written=698033006
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=32114066
    HDFS: Number of bytes written=12135766
    HDFS: Number of read operations=567
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=194
    HDFS: Number of bytes read erasure-coded=0
  Map-Reduce Framework
    Map input records=23596
    Map output records=23596
    Map output bytes=286902
    Map output materialized bytes=334100
    Input split bytes=135
    Combine input records=0
    Combine output records=0
    Reduce input groups=444
    Reduce shuffle bytes=334100
    Reduce input records=23596
    Reduce output records=100
    Spilled Records=47192
    Shuffled Maps =1
    Failed Shuffles=0
    Merged Map outputs=1
    GC time elapsed (ms)=0
    Total committed heap usage (bytes)=2984247296
  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=475835
  File Output Format Counters
    Bytes Written=1482
```

输出结果如下:

Output (总词频统计)

```
1: thou, 5589
2: thy, 4004
3: shall, 3536
4: thee, 3204
5: lord, 3134
6: king, 3101
7: sir, 2976
8: good, 2837
9: come, 2492
10: let, 2317
11: love, 2285
12: enter, 2257
13: man, 1977
14: hath, 1931
15: like, 1893
16: know, 1764
17: say, 1698
18: make, 1676
19: did, 1670
20: tis, 1392
21: speak, 1189
22: time, 1181
23: tell, 1086
24: heart, 1083
25: henry, 1076
26: duke, 1075
27: think, 1061
28: doth, 1056
29: father, 1051
30: lady, 1019
31: exeunt, 985
32: men, 956
33: day, 952
34: night, 947
35: art, 934
36: queen, 922
37: look, 921
38: exit, 913
39: hear, 897
40: great, 886
41: life, 876
42: death, 872
43: hand, 865
44: god, 860
45: sweet, 852
46: act, 832
47: master, 832
48: true, 829
49: fair, 822
50: away, 816
51: mistress, 812
52: eyes, 772
53: scene, 771
54: prince, 762
55: pray, 753
56: old, 710
57: second, 706
58: honour, 703
59: world, 694
60: fear, 687
61: son, 677
62: heaven, 672
63: poor, 662
64: blood, 654
65: leave, 643
66: till, 642
67: brother, 633
```

```
1: king, 311
2: lear, 257
3: thou, 229
4: gloucester, 179
5: kent, 176
6: thy, 161
7: thee, 139
8: edgar, 136
9: edmund, 131
10: fool, 120
11: sir, 115
12: regan, 105
13: shall, 99
14: lord, 98
15: man, 93
16: come, 88
17: let, 88
18: good, 85
19: know, 84
20: goneril, 84
21: enter, 79
22: alban, 77
23: cornwall, 77
24: father, 76
25: cordelia, 64
26: gentleman, 58
27: old, 58
28: oswald, 56
29: hath, 56
30: make, 53
31: love, 53
32: tis, 53
33: speak, 51
34: like, 49
35: heart, 49
36: poor, 48
37: exit, 45
38: art, 41
39: say, 40
40: night, 39
41: eyes, 38
42: nature, 38
43: away, 37
44: life, 36
45: sister, 35
46: exeunt, 34
47: madam, 33
48: france, 32
49: tell, 31
50: act, 31
51: daughter, 30
52: fellow, 29
53: time, 29
54: duke, 29
55: daughters, 29
56: better, 29
57: son, 29
58: great, 28
59: scene, 27
60: letter, 27
61: hear, 27
```

2.遇到的除代码以外一些问题

打包成jar包的时候报错 mkdirs failed to creat /var/...

```
at org.apache.hadoop.util.RunJar.main(RunJar.java:236)
[quinton_541@MacBook-Air-de-541 WordCount % $HADOOP_HOME/bin/hadoop jar WordCount]
.jar /input /output -skip /Exceptions/stop-word-list.txt /skip/punctuation.txt
Exception in thread "main" java.io.IOException: Mkdirs failed to create /var/fo
lders/75/z5lqntj94ql_qwb6ctdgtyhc0000gn/T/hadoop-unjar8275782537763363207/META-INF/
license
    at org.apache.hadoop.util.RunJar.ensureDirectory(RunJar.java:229)
    at org.apache.hadoop.util.RunJar.unJar(RunJar.java:202)
    at org.apache.hadoop.util.RunJar.unJar(RunJar.java:101)
    at org.apache.hadoop.util.RunJar.run(RunJar.java:310)
    at org.apache.hadoop.util.RunJar.main(RunJar.java:236)
```

解决方案：

mac os x打包hadoop的jar包报的错

```
zip -d mr.jar META-INF/LICENSE
```

输入如上指令 即可

zsh指令突然全部失效

写着写着更新了macOS Monterey 然后少有的重启了一下电脑 再次打开terminal 发现zsh六亲不认：

zsh:command not found: ls

zsh:command not found: vim


...

查了一下好像是zsh普遍的问题 关键问题还是 zsh进来时候读的配置文件的问题

由于网上的教程几乎全是在bash shell里搞定的， 所以有些环境变量不知道为什么就 抽风了。

根据教程 执行PATH=/bin:/usr/bin:/usr/local/bin:\${PATH}，指令是回来了 但是治标不治本。

查阅相关资料发现 zsh的环境变量是从~/.zshrc读取的，因此将环境变量生效的指令 source /etc/profile和export PATH指令写入.zshrc 问题解决。



```
quinton_541 — vim .zshrc — 80x24
source /etc/profile
export PATH=/bin:/usr/bin:/usr/local/bin:${PATH}
```

可以改进的地方

1.任务重复

由于本人比较懒（x 所以将统计全部词频与统计单文件词频分开来做

其实 在统计完单文件词频后 我们完全可以根据single-output中的文件，对所有词频再次进行一个mapreduce，或者在分文件进行排序时，根据之前统计词频的<word-filename,value>将word单独分开来进行排序，这样就不需要执行两次重复的词频统计力。

2.同义词合并/停词更新

也许这不是程序员需要考虑的问题 但是从程序的输出来看 确实有非常多的同义词占据了排行榜的位置：

```
1: thou, 5589
2: thy, 4004
3: shall, 3536
4: thee, 3204
5: lord, 3134
6: king, 3101
7: sir, 2976
8: good, 2837
9: come, 2492
10: let, 2317
11: love, 2285
12: enter, 2257
13: man, 1977
14: hath, 1931
15: like, 1893
16: know, 1764
17: say, 1698
18: make, 1676
19: did, 1670
20: tis, 1392
21: speak, 1189
22: time, 1181
23: tell, 1086
24: heart, 1083
25: henry, 1076
26: duke, 1075
27: think, 1061
28: doth, 1056
29: father, 1051
30: lady, 1019
```

例如 thou/thy/thee（虽然我觉得这个词完全应该放在停词里

包括一些时态是否需要合并，这些都是可以优化的地方

当然，这也可以变成程序员考虑的问题，例如动词判断时态，将其规约到动词原形(doth,does,did,doing,done->do)，但是不规则动词似乎只能枚举 手打（有点折磨人

另外 停词列表是否需要更合理的进行更新，我们发现，在统计出的高频单词中，shall,doth,hath,art这些无意义的助词占据了許多位置（当然，中世纪英语里art=are，判断art到底是啥意思好像也是一个问题，这可能要涉及到NLP的东西）