Hadoop Streaming

刘磊

2020 年 9 月

Hadoop 框架是用 Java 语言写的，也就是说，Hadoop 框架中运行的所有应用程序都要用 Java 语言来写才能正常地在 Hadoop 集群中运行。如果不会 Java 语言怎么办?

Hadoop 提供了 Hadoop Streaming 这个编程工具，它允许用户使用任何可执行文件或者脚本文件作为 Mapper 和 Reducer，因此我们可以选择自己熟悉的编程语言，编写 Mapper 和 Reducer 程序来使用 Hadoop 集群。

Streaming 的原理是用 Java 实现一个包装用户程序的 MapReduce 程序，该程序负责调用 MapReduce Java 接口获取 key/value 对输入，创建一个新的进程启动包装的用户程序，将数据通过管道传递给包装的用户程序处理，然后调用 MapReduce Java 接口将用户程序的输出切分成 key/value 对输出。

我们以 Python 语言为例，Mapper 文件内容如下

```
import sys

for line in sys.stdin:
    line = line.strip().split(' ')
    for word in line:
        if word.strip() != "":
            print("%s\t%s"%(word, 1))
```

Reducer 文件内容如下

```
import sys

current_word = None
count_pool = []
sum = 0
```

```
for line in sys.stdin:
    word, val = line.strip().split('\t')

    if current_word == None:
        current_word = word

    if current_word != word:
        for count in count_pool:
            sum += count
        print("%s\t%s" % (current_word, sum))
        current_word = word
        count_pool = []
        sum = 0

    count_pool.append(int(val))

for count in count_pool:
    sum += count
print("%s\t%s" % (current_word, str(sum)))
```

将上面的代码分别保存成 mapper.py 和 reducer.py，使用 shell 命令的管道功能测试 Mapper 和 Reducer 程序。命令如下：

```
cat testfile | python3 mapper.py | sort ' ' -k 1 | python3 reducer.py
```

将 testfile 作为数据传递给 mapper.py 处理，再将处理结果进行排序，之后再把排序结果交给 reducer.py 处理。注意，(1)这里我们直接使用 python3 来运行 python 程序，也可以在 python 程序首行指明解释程序然后赋予可执行权限。(2)sort 为排序，-t 表示指定分隔符，-t ' '表示用 tab 进行分隔，-k 表示排序时指定的键是在分隔后的哪个 field，-k 1 表示按分隔符分隔后的第一个 field，也即我们在 mapper 中打印的 key/value 中的 key。测试无误后，就可以将任务放到 Hadoop 集群上运行。命令格式如下：

```
Usage: $HADOOP_HOME/bin/hadoop jar hadoop-streaming.jar [options]
```

参数

-input <path>：指定作业输入，path 可以是文件或者目录，可以使用*通配符，-input

选项可以使用多次指定多个文件或目录作为输入。

-output <path>：指定作业输出目录，path 必须不存在，而且执行作业的用户必须有创建该目录的权限，-output 只能使用一次。

-mapper：指定 mapper 可执行程序或 Java 类，必须指定且唯一。

-reducer：指定 reducer 可执行程序或 Java 类，必须指定且唯一。

-file, -cacheFile, -cacheArchive：分别用于向计算节点分发本地文件、HDFS 文件和 HDFS 压缩文件。

-numReduceTasks：指定 reducer 的个数，如果设置-numReduceTasks 0 或者-reducer NONE 则没有 reducer 程序，mapper 的输出直接作为整个作业的输出。

-jobconf / -D NAME=VALUE：指定作业参数，NAME 是参数名，VALUE 是参数值，可以指定的参数参考 hadoop-default.xml。

可以通过以下命令查看完整的命令参数介绍

```
hadoop jar /apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.0.0.jar --help
```

我们可以将上述命令写成一个 shell 脚本方便使用。我们首先将 hadoop-streaming-3.0.0.jar 的路径，和程序输入输出路径保存成变量，方便后面的书写命令。其次，我们判断输出目录是否已经存在，如果存在就删除。最后是 hadoop streaming 命令，在命令中我们给任务起名为“WordCount”。

```
STREAMING_JAR_PATH=/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.0.0.jar

INPUT_FILE_PATH="/input/files/testfile*"
OUTPUT_PATH="/output/streaming/wordcount"

if hadoop fs -test -d $OUTPUT_PATH
then
    hadoop fs -rm -r -skipTrash $OUTPUT_PATH
```

```
fi

hadoop jar $STREAMING_JAR_PATH \
-D mapred.job.name="WordCount" \
-file ~/hadoop_streaming/wordcount/mapper.py \
-file ~/hadoop_streaming/wordcount/reducer.py \
-input $INPUT_FILE_PATH \
-output $OUTPUT_PATH \
-mapper "python3 mapper.py" \
-reducer "python3 reducer.py"
```

从上面的过程可以看出，Mapper 和 Reducer 都是可执行文件，它们从标准输入读入数据（一行一行读）， 并把计算结果发给标准输出。Streaming 工具会创建一个 Map/Reduce 作业， 并把它发送给合适的集群，同时监视这个作业的整个执行过程。

如果一个可执行文件被用于 Mapper，则在 Mapper 初始化时， 每一个 Mapper 任务会把这个可执行文件作为一个单独的进程启动。 Mapper 任务运行时，它把输入切分成行并把每一行提供给可执行文件进程的标准输入。 同时，Mapper 收集可执行文件进程标准输出的内容，并把收到的每一行内容转化成 key/value 对，作为 Mapper 的输出。 默认情况下，一行中第一个 Tab 之前的部分作为 Key，之后的（不包括 Tab）作为 Value。 如果没有 Tab，整行作为 Key 值，Value 值为 null。不过，这可以定制。

如果一个可执行文件被用于 Reducer，每个 Reducer 任务会把这个可执行文件作为一个单独的进程启动。 Reducer 任务运行时，它把输入切分成行并把每一行提供给可执行文件进程的标准输入。 同时，Reducer 收集可执行文件进程标准输出的内容，并把每一行内容转化成 Key/Value 对，作为 Reducer 的输出。 默认情况下，一行中第一个 Tab 之前的部分作为 Key，之后的（不包括 Tab）作为 Value。

计算微博的 TF-IDF

TF-IDF 是一种统计方法，用以评估一个词对语料库中一篇文档的重要程度。词的重要性随着它在文档中出现的次数成正比增加，但同时会随着它在语料库中出现的文档数降低。TF-IDF 综合考虑两方面的重要性：

- TF(Term Frequency)：词频指的是某一个给定的词语在一篇文档中出现的次数。
- IDF(Inverse Document Frequency)：逆文档频率是一个词语普遍重要性的度量。某一特定词语的 IDF，由总文档数目除以出现该词的文档数，再将得到的商取对数得到

$$IDF = \log\left(\frac{N}{DF}\right)$$

其中 N 语料总的文档数，DF 为出现某个词的文档数。

TF-ID 计算公式为

$$TF - IDF = TF \times \log\left(\frac{N}{DF}\right)$$

我们以一个微博语料为例，介绍如何使用 MapReduce 计算每条微博中出现的词的 TF-IDF。数据文件为两列，第一列是博主 ID，第二列是微博内容，以 Tab 进行分隔。内容如下表所示。

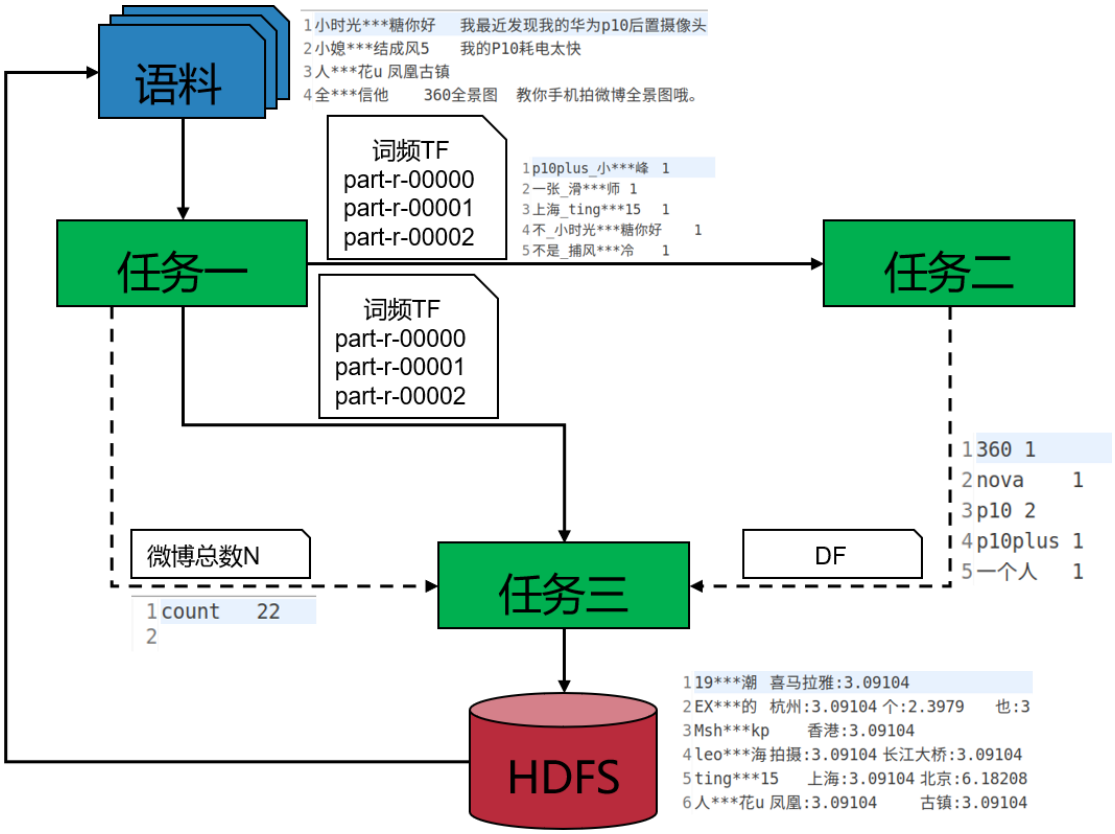
ID	内容
小媳***结成风 5	我的 P10 耗电太快
全***信他	360 全景图　教你手机拍微博全景图哦。
…	…

输出结果如下图所示，第一列是博主 ID，从第二列开始为微博中出现的每个词及其 TF-IDF 值，以冒号分隔。列与列以 Tab 分隔。

```
19***潮	喜马拉雅:3.09104
EX***的	杭州:3.09104 个:2.3979	也:3.09104	买:3.09104	以后:3.09104 路上:3.
Msh***kp	香港:3.09104
leo***海 拍摄:3.09104 长江大桥:3.09104 最想:3.09104 上:3.09104	一起:3.09104
ting***15	上海:3.09104 北京:6.18208
人***花u 凤凰:3.09104	古镇:3.09104
```

从 TF-IDF 的计算公式可知，要计算一个词在一条微博中的 TF-IDF，我们需要统计这个词

在这条微博出现的次数，也就是这个词针对这个微博的 TF。注意 TF 是一个局部量，不同微博中同一个词的 TF 可能是不同的。还需要统计这个词的 DF，也就是在语料中的多少篇微博中出现过，这是一个全局量。另外还需要总的微博条数 N。针对需要计算的量，我们将计算过程分为三个子任务，每个任务由一个 MapRecude 完成。如下图所示：



任务一：以整个语料作为输入，统计每个词的 TF 和微博总数 N。

Mapper 处理数据主要代码如下：

```
        String[] line = value.toString().split("\t");  //以 tab 键为分隔符
        if (line.length >= 2) {
            String id = line[0].trim();     //微博的 ID
            String content = line[1].trim();  //微博的内容
            StringReader sr = new StringReader(content);
            IKSegmenter iks = new IKSegmenter(sr, true);  //使用
            Lexeme lexeme = null;
            while ((lexeme = iks.next()) != null) {
```

```
                String word = lexeme.getLexemeText(); //word 就是分完的每
个词
                context.write(new Text(word + "_" + id), new
IntWritable(1));
            }
            sr.close();
            context.write(new Text("count"), new IntWritable(1));
        }
```

本任务的 Mapper 与 WordCount 的 Mapper 函数实现逻辑非常接近。没对微博内容进行分析，分词后每个词生成一个键值对，由于 TF 是一个局部量，为了将 TF 与微博对应起来，输出的时候将词语与博主 ID 合并作为键（这里假设每个 ID 对应一条微博，如果每个 ID 对应多条微博，我们可以对微博进行编号，将编号与词语合并作为键）。另外我们构造一个词“count”，来统计总的微博数。Reducer 将 Mapper 结果进行合并，得到每条微博中每个词的 TF，以及总的微博数。值得一提的是，为了方便后面使用，我们将“count”的值单独输出到一个文件进行保存。因此，我们自定义了分区函数

```
public int getPartition(Text key, IntWritable value, int
numReduceTasks) {
        //如果 key 值为 count，就返回 3，其他的 key 值就平均分配到三个分区，
        if (key.equals(new Text("count"))) {
            return 3;
        } else {
         //numReduceTasks - 1 的意思是有 4 个 reduce，其中一个已经被 key 值为
count 的占用了，所以数据只能分配到剩下的三个分区中了
         //使用 super，可以调用父类的 HashPartitioner
            return super.getPartition(key, value, numReduceTasks - 1);
        }
    }
```

将“count”的值保存到一个分区，每个词的 TF 值保存到其它三个分区。

任务二：由任务一的结果统计每个词的 DF。

任务二以任务一的结果作为输入。Mapper 的主要代码如下

```
        FileSplit fs = (FileSplit) context.getInputSplit();
        //map 时拿到 split 片段所在文件的文件名
        if (!fs.getPath().getName().contains("part-r-00003")) {
```

```
        //拿到 TF 的统计结果
            String[] line = value.toString().trim().split("\t");
            if (line.length >= 2) {
                String[] ss = line[0].split("_");
                if (ss.length >= 2) {
                    String w = ss[0];
                    //统计 DF，该词在所有微博中出现的条数，一条微博即使出现两次该
词，也算一条
                    context.write(new Text(w), new IntWritable(1));
                }
            } else {
                System.out.println("error:" + value.toString() + "-----
--------");
            }
        }
```

由于计算 DF 时不需要“count”的值，处理时将文件名含 part-r-00003 的文件排除。另外，因为 DF 是一个全局量，输出的时候将 ID 从键中去掉。Reducer 做合并操作。

任务三：由前两个任务的结果计算每条微博中每个词的 TF-IDF。

将任务一计算的微博条数的输出文件和任务二得到的 DF 值的输出文件缓存到任务三的运行节点，在 setup()函数中构建两个 HashMap<String, Integer>对象来保存两个文件的数据：cmap 保存微博总条数，df 保存每个词的 DF 值。任务一的输出中，除了名为 part-r-00003 的文件，其它的文件作为任务三的输入。Map 函数的主要部分为

```
if (!fs.getPath().getName().contains("part-r-00003")) {
            String[] v = value.toString().trim().split("\t");
            if (v.length >= 2) {
                int tf = Integer.parseInt(v[1].trim());
                String[] ss = v[0].split("_");
                if (ss.length >= 2) {
                    String w = ss[0];
                    String id = ss[1];
                    //执行 W = TF * Log(N/DF)计算
                    double s = tf * Math.log(cmap.get("count") /
df.get(w));
                    //格式化，保留小数点后五位
                    NumberFormat nf = NumberFormat.getInstance();
                    nf.setMaximumFractionDigits(5);
                    //以 微博 id+词：权重 输出
```

```
                    context.write(new Text(id), new Text(w + ":" +
nf.format(s)));
                }
            } else {
                System.out.println(value.toString() + "-------------");
            }
        }
```

在计算 TF-IDF 时，TF 值从输入中获得，微博数和 DF 值从 HashMap 对象中读取。输出以博主 ID 作为键，词加 TF-IDF 值作为值。Reducer 将每条微博的结果进行合并输出。其它的具体细节，参考代码中的注释。