DS-Lab4

曹金坤 515260910022

本文档用以简要介绍我对于分布式系统Lab4-MapReduce的实现方法，部分内容借助源代码进行直接展示和说明。

Part I: Map/Reduce input and output

该部分实现了对于输入内容的最基本的map和reduce的操作，具体的实现代码集中在**Mapper.java**和**Reducer.java**两个文件中。

doMap

对于输入的文件内容，首先读取他的文本内容，因为文件的实现是基于java的，所以我使用了基于*InputStreamReader*的方法。之后，通过一个KeyValue数据类型的列表，它捕获了已经被预先定义的*mapF*对于之前读取的文件内容的map结果。

同时，对于不同的reducer任务，mapper应该分配不同的KeyValue的list给他，以此来实现之后可能的分布式运算。这种对于不同list的组织和分配，我简单利用了一个List<List>来实现。

最后，为了对于获取的KeyValue对的记录，我考虑使用了方便的结构化文件类型来进行，具体来说我选择了Json文件进行记录。这部分我调用了Java中支持的对于String类型转化为Json对象的接口，非常方便。至此，doMap也就完成了，原始的输入文件被转化成了若干个json格式的中间记录文件。

下面是完整的**doMap**函数的代码实现：

```
// core.Mapper.java
public static void doMap(String jobName, int mapTask, String inFile, int nReduce, MapFunc
mapF) {
        try{
            StringBuffer sb = new StringBuffer();
            InputStream is = new FileInputStream(inFile);
            String line;
            BufferedReader reader = new BufferedReader((new InputStreamReader(is)));
            line = reader.readLine();
            while(line != null){
                sb.append(line);
                sb.append('\n');
                line = reader.readLine();
            }
            reader.close();
            is.close();
            String fileContent = sb.toString();
            List<KeyValue>  kvList = mapF.map(inFile, fileContent);
            List<List<KeyValue>> mapResultList = new ArrayList<>();
            for(int m=0; m < nReduce; m++){
                // initialize the lists in the map results
                mapResultList.add(new ArrayList<>());
            }
            for(int i=0; i < kvList.size(); i++){
```

```
                // set the value in map result lis with hash value after mod
                int id = hashCode(kvList.get(i).key) % nReduce;
                mapResultList.get(id).add(kvList.get(i));
            }
            for(int r=0; r < nReduce; r++){
                // for each reduce task, create an immediate file
                String fileName = Utils.reduceName(jobName, mapTask, r);
                File file = new File(fileName);
                if(!file.exists()){
                    file.createNewFile();
                }
                // record the mapping and hashing results in json files
                String jsonValue = JSON.toJSONString(mapResultList.get(r));
                FileWriter fw = new FileWriter(fileName);
                BufferedWriter bw = new BufferedWriter(fw);
                bw.write(jsonValue);
                bw.close();
            }
        }
        catch(Exception e){
            System.out.println("something bad in doMap\n");
        }
    }
```

doReduce

作为Map-Reduce的第二个阶段，基于Map阶段已经得到的统一格式的中间记录文件，下面调用doReduce函数进行最后的各个文件的内容的计算。为了方便，整个过程的核心数据结构是一个Map<String, List>结构。它保存了不同的key和对应的value的列表之间的映射关系。

因为在map阶段，生成的中间结果是用json文件进行存储的，所以在reduce的开始，就需要对这些json文件的内容进行解释。从json文本的内容中，对不同的文件重建出了存储的KeyValue列表的内容。对于不同的key，会把它在当前的json文件中对应的String的内容，存储到前述的全局映射结构中。完成了这个阶段之后，所有的中间文件的内容就已经从文件转移到了函数内维护的全局Map结构之中，方便进行之后的操作。

之后，对于Map中存储的所有的key进行排序，这些排序之后的内容又会转化为一个对应的Json内容进行存储。这个文件便是之后用于merge阶段的依据文件。这个部分的操作较为简单，不进行赘述了。细节可以在下方的代码中获得。

```
// core.Reducer.java
public static void doReduce(String jobName, int reduceTask, String outFile, int nMap,
ReduceFunc reduceF) {
        Map<String, List<String>> resMap = new HashMap<>();
        for(int i=0; i < nMap; i++){
            // read files produced by doMap and build the key-value map
            try {
                String fileName = Utils.reduceName(jobName,i, reduceTask);
                Path filePath = new File(fileName).toPath();
                String content = new String(Files.readAllBytes(filePath));
                // get the json content from mapped files
                JSONArray jsonContent = JSONArray.parseArray(content);
                List<KeyValue> kvList = JSONObject.parseArray(jsonContent.toJSONString(),
KeyValue.class);
```

```
                for(int keyId=0; keyId < kvList.size(); keyId++){
                    boolean inMap = resMap.containsKey(kvList.get(keyId).key);
                    if(!inMap){
                        resMap.put(kvList.get(keyId).key, new ArrayList<>());
                    }
                    String key = kvList.get(keyId).key;
                    String value = kvList.get(keyId).value;
                    resMap.get(key).add(value);
                }
            }
            catch (IOException e){
                e.printStackTrace();
            }
        }

        List<String> keyList = new ArrayList<>(resMap.keySet());
        Collections.sort(keyList);
        JSONObject jsonObj = new JSONObject();
        for(int i=0; i < keyList.size(); i++){
            String key = keyList.get(i);
            String[] valueArray = new String[resMap.get(key).size()];
            String[] targetList = resMap.get(key).toArray(valueArray);
            String value = reduceF.reduce(key, targetList);
            jsonObj.put(key, value);
        }

        String fileName = Utils.mergeName(jobName, reduceTask);
        File reduceFile = new File(fileName);
        try {
            if(!reduceFile.exists()){
                reduceFile.createNewFile();
            }

            FileWriter fw = new FileWriter(fileName);
            BufferedWriter bw = new BufferedWriter(fw);
            bw.write(jsonObj.toString());
            bw.close();
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
```

Part II: Single-worker word count

对于wordcount的具体实现，简单的来说就是通过*mapFunc*函数对文件内容进行处理，之后把不同的关键词提取出来，存储到一个KeyValue类型的列表之中。其中，在进行文本处理的时候，对于许多的特别字符需要进行过滤，仅仅保留题目所需的英文字母和数字内容。然后，如果正则表达式的匹配，构建所需要的List结果。具体的内容实现如下：

```
    public static List<KeyValue> mapFunc(String file, String value) {
        List<KeyValue> kvList = new ArrayList<>();
```

```
        String[] digitArray = // list of nonlegal charactristics;
        for(int i=0; i < digitArray.length; i++){
            value = value.replace(digitArray[i], " ");
        }
        String[] tempString = value.split("\\s+");
        for(int i=0; i < tempString.length; i++){
            String s = tempString[i];
            if(s.matches("[a-zA-Z0-9]+")){
                kvList.add(new KeyValue(s, ""));
            }
        }
        return kvList;
    }
```

在*reduceFunc*函数中，策略较为简单，返回记录的单词list长度即可，不进行额外的赘述了。

Part III: Distributing MapReduce tasks

这个部分的主要内容是把之前的Sequential实现方式，变成一种可以模拟分布式运行的、通过RPC进行过程调用的方法。这个部分涉及到了多线程的内容，所幸Java中对这部分有了较好的支持。为了实现不同的worker的工作，我首先定义了一个工作类*threadWorker*，它记录了一个任务需要的一些信息，同时拥有一个计数器，这个计数器用于*thread*的生命周期进行调节，具体的实现如下：

```
class threadWorker implements Runnable{
    // the class to hold task for a single thread
    private CountDownLatch counter;
    private int taskNum;
    private int nOthers;
    private String workerId;

    private threadWorker(CountDownLatch counter, int taskNum, int nOthers){
        this.counter = counter;
        this.taskNum = taskNum;
        this.nOthers = nOthers;
        this.workerId = "";
    }

    public void run(){
        while(true){
        DoTaskArgs args = new DoTaskArgs(jobName, mapFiles[taskNum], phase,
        taskNum, nOthers);
        try{
            this.workerId = registerChan.read();
            Call.getWorkerRpcService(this.workerId).doTask(args);
            Utils.debug(this.workerId + "finished assigned task");
            registerChan.write(this.workerId);
            break;
        }
        catch (Exception e){
            // fault tolerance
            e.printStackTrace();
            continue;
```

```
        }
    }
        this.counter.countDown();
    }
}
```

基于这个线程程序，对于一个任务列表，可以进行一个简单的任务分配了。具体的来说，在创建一个计数器之后，把它作为全局共享的*CounterDownLatch*对象，然后对于每个task分别创建一个*threadWorker*对象并且运行，在这之后，利用*CounterDownLatch*的**await**操作保证最后所有线程的回收即可。

Part IV: Handling worker failures

该部分的内容仅仅和scheduler相关，在上一个部分中，通过时钟的配合，以及在*threadWorker*中面对连接中断等原因导致的运行异常的时候不结束运行，而是创建一个新的RpcService来继续进行任务，由此保证了一定的容错。相关的代码实现见前一部分的代码，fault tolerance部分进行了注释。

Part V: Inverted index generation

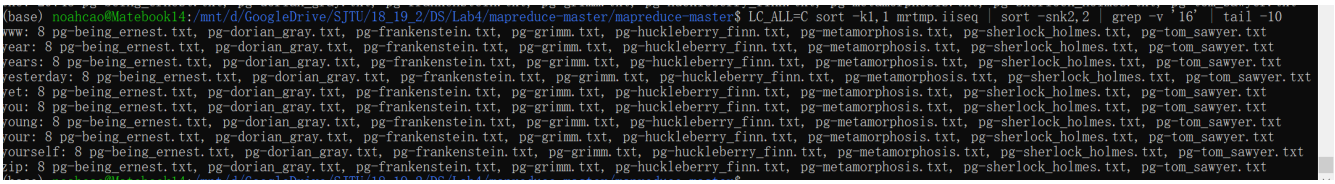
在该部分中，不同于之前对于文本中单词的计数，只需要知道在多少个文件中出现了对应的单词即可。所有的文件修改被限制在了*invertedIndex.java*中，其中*mapFunc*和*reduceFunc*两个函数和wordcount中的有所不同。其中，mapFunc中，只要在进行KeyValue的列表生成的时候，把每个单词(key)当前对应的文件名(value)保存到最后返回的列表中即可。

reduce阶段则有更多的不同。首先，因为一个单词在每个文件中出现多次和一次，都只是对应最后结果中的“1”而已，所以我们更改了相关的逻辑：

1. 对于输入的values数列，对于其中每个文件名，检查它是否已经在本次运行中出现过，如果出现过则说明对于当前的key，它之前被检测到出现过在该文件中，此时不进行额外的操作，否则将该文件名添加到一个集合中；
2. 在构建前述的集合的时候，同时生成一个包含了该单词出现的文件数目和对应的文件名的字符串，这个字符串被最后作为结果返回。

```
public static String reduceFunc(String key, String[] values) {
        int vLen = values.length;
        String resStr = "";
        Collection seenValues = new HashSet();
        for(int i=0; i < vLen; i++){
            if(seenValues.contains(values[i])){
                continue;
            }
            else {
                seenValues.add(values[i]);
                resStr = resStr + values[i] + ", ";
            }
        }
        if(vLen > 0){
            resStr = resStr.substring(0, resStr.length()-2);
        }
        String res = seenValues.size() + " " + resStr;
        return res;
    }
```

和作业文档中要求的一样，在运行了*invertedIndex.java*文件后，对生成的结果文件进行检查：



判断结果符合题目的要求，该部分完成。