//王泽宇_181250148_金融大数据处理技术作业 5
//Git 仓库链接：https://github.com/ZeyuuOne/MapReduceWordCount

一、文件组织结构说明

Git 仓库的文件组织结构如下：

目录名或文件名	说明
.gradle	Gradle 相关文件目录
build/libs	打包好的 jar 文件所在目录
gradle/wrapper	Gradle Wrapper 所在目录
output	输出文件目录
src/main/java	程序源文件所在目录
.gitignore	.gitignore 文件
build.gradle	Gradle 编译配置文件
gradlew	Gradle Linux Shell 可执行脚本
gradlew	Gradle Windows 可执行 bat 脚本
README.pdf	实验报告
settings.gradle	Gradle 配置文件

二、开发环境配置

本次作业，我选择 IntelliJ IDEA 作为集成开发环境。通过教育邮箱注册账号可以获得 IntelliJ IDEA (Ultimate Edition)的免费授权。由于先前在项目中使用过 Android Studio 配合 Gradle 来开发安卓应用程序，因此本次我选用 Gradle 作为 Java 项目构建工具。

首先在 IDEA 中新建项目，选择 Gradle 和 Java。在 https://search.maven.org 查询后发现，Hadoop 相关依赖项的最新版本是 3.3.0，因此在 build.gradle 中导入所需依赖项，版本设为 3.3.0，代码如下：

```
dependencies {
    compile group: 'junit', name: 'junit', version: '4.12'
    compile group: 'org.apache.hadoop', name: 'hadoop-common', version: '3.3.0'
    compile group: 'org.apache.hadoop', name: 'hadoop-client', version: '3.3.0'
    compile group: 'org.apache.hadoop', name: 'hadoop-hdfs', version: '3.3.0'
    compile group: 'org.apache.hadoop', name: 'hadoop-mapreduce', version: '3.3.0'
    compile group: 'org.apache.hadoop', name: 'hadoop-mapreduce-client', version: '3.3.0'
}
```

而后在 src/main/java 目录下新建 Java Class 文件，即可开始编写 Hadoop MapReduce 程

序。程序编写完成后通过 Gradle/Tasks/build/jar 即可将项目打包为 jar 文件，交由 Hadoop 运行。

经过测试，发现项目默认采用的 openjdk-15 版本过高，程序无法在 Hadoop 中正常运行。在 Project Structure 中将 Project SDK 降级为 adopt-openjdk-1.8，再重新对程序编译打包，即可在 Hadoop 中正常运行。

三、MapReduce 程序编写

编写 Hadoop MapReduce 程序，需要实现一个类，其中包含一个继承自 Mapper 类的内部类、一个继承自 Reducer 类的内部类，以及 main()函数。

以 WordCount 2.0 程序为基础，首先加入对 token 长度和是否为数字的判断。观察代码可发现，TokenizerMapper 类中 map()函数内的 itr.nextToken()是拆分好的 token，对其执行判断，若其长度小于 3 或是数字串，则通过 continue 进入下一次循环。

```
1.  String token = itr.nextToken();
2.  if (token.length() < 3 || isNumber(token)) continue;
```

接着加入对停词的判断。观察代码可发现，在 main()函数中可通过 job.addCacheFile()向程序传递输入文件，因此将“-stop”的后一个参数，即停词表文件的路径通过 addCacheFile()传入。

```
1.  else if ("-stop".equals(remainingArgs[i])) {
2.      job.addCacheFile(new Path(remainingArgs[++i]).toUri());
3.      job.getConfiguration().setInt("wordcount.stop.words", ++cacheIndex);
4.  }
```

为了在 Mapper 内找到 CacheFile 中的停词表文件，此处将 wordcount.stop.words 设为 int 类型，表示停词表文件在 CacheFile 中的 index。在 Mapper 类的 setup()函数即可通过 index 提取传入的停词表文件路径。

```
1.  if (-1 < conf.getInt("wordcount.stop.words", -1)) {
2.      Path stopWordsPath = new Path(URIs[conf.getInt("wordcount.stop.words", -
    1)].getPath());
3.      String stopWordsFileName = stopWordsPath.getName().toString();
4.      parseStopWordsFile(stopWordsFileName);
5.  }
```

对标点符号文件也做相同的操作。

再仿照 parseSkipFile()定义 parseStopWordsFile()函数，将文件输入处理为 Set 类型的停词表。最后在 map()函数中加入对 token 是否为停词的判断。

```
1.  boolean stop = false;
2.  for (String stopWord : stopWords) {
3.      if (stopWord.equals(token)) {
4.          stop = true;
```

```
            break;
        }
    }
    if (stop) continue;
```

接下来将程序输出结果按照 value 降序排列。首先将 job 的输出路径设为“temp”，在 main()函数中 job 运行成功后，再定义一个 sortJob，输入路径设置为“temp”，即让 sortJob 处理 job 的输出结果。

```
FileOutputFormat.setOutputPath(job, tempPath);
Job sortJob = Job.getInstance(conf, "value sort");
FileInputFormat.addInputPath(sortJob, tempPath);
FileOutputFormat.setOutputPath(sortJob, new Path(otherArgs.get(1)));
```

由于 Hadoop MapReduce 框架会自动将结果按照 key 升序排序，此处利用这一机制，通过将 key 与 value 对换，来实现按 value 排序。只需调用 Hadoop MapReduce 框架中预定义的 InverseMapper 类作为 Mapper 即可实现 key 与 value 的对换。

```
sortJob.setMapperClass(InverseMapper.class);
```

但默认情况下，排序方式为升序，因此需要自定义 Comparator，由于此时 key 的类型是 IntWritable，自定义 Comparator 应当继承自 IntWritable.Comparator。直接调用父类的 compare 函数并加个负号来实现降序比较。

```
public static class IntWritableDecreaseComparator extends IntWritable.Comparator {
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        return -super.compare(a, b);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return -super.compare(b1, s1, l1, b2, s2, l2);
    }
}
```

将 sortJob 的 SortComparatorClass 设为自定义的 Comparator，即实现了按照 job 输出结果的 value 降序排列。

```
sortJob.setSortComparatorClass(IntWritableDecreaseComparator.class);
```

接下来将降序排列好的结果规格化输出。采用自定义 FileOutputFormat 的方式实现。首先定义 OrderOutputFormat 类，继承自 FileOutputFormat 类，IDE 提示需要实现其中的纯虚

函数 getRecordWriter()，即需要自定义 RecordWriter 类并返回。

```
1. public static class OrderOutputFormat extends FileOutputFormat<IntWritable,
   Text> {
2.     @Override
3.     public RecordWriter<IntWritable, Text> getRecordWriter(TaskAttemptContex
   t job) throws IOException, InterruptedException {
4.         return new OrderRecordWriter(job);
5.     }
6. }
```

因此定义 OrderRecordWriter 类，继承自 RecordWriter 类，IDE 提示需要实现其中的纯虚函数 write()和 close()。首先定义构造函数，在构造函数中创建文件输出流。

```
1. fs = FileSystem.get(job.getConfiguration());
2. Path outputPath = new Path("output/out.txt");
3. fos = fs.create(outputPath);
```

为了输出序号，在类中定义数据成员 order，初始化为 0，write()函数每次被调用时对其自增。同时在 write()函数中实现自定义的输出格式，如果 order 大于 99 则不再输出。

```
1. @Override
2. public void write(IntWritable key, Text value) throws IOException, Interrupt
   edException {
3.     if (order > 99) return;
4.     fos.write(((++order).toString()+": "+value.toString()+", "+key.toString(
   )+"\n").getBytes());
5. }
```

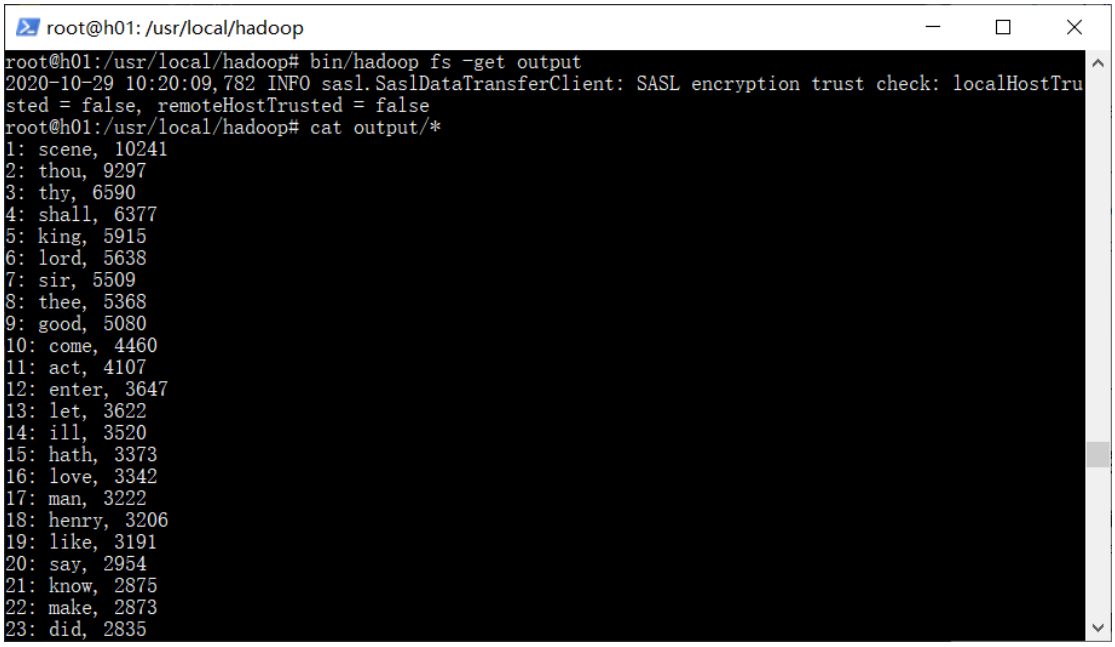
在 close()函数中只需关闭文件输出流即可。

```
1. @Override
2. public void close(TaskAttemptContext context) throws IOException, Interrupte
   dException {
3.     IOUtils.closeStream(fos);
4. }
```

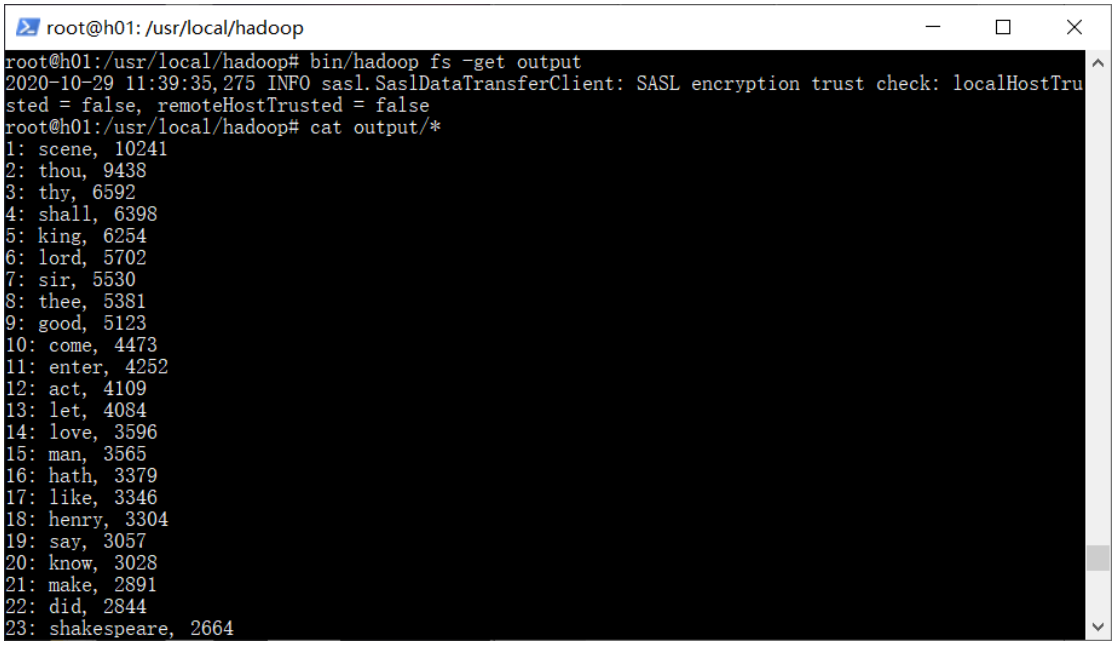
本次作业所需的 MapReduce 程序即编写完毕。

四、集群运行测试

将 jar 包在 Hadoop 集群中运行，得到结果如下。



结果符合预期。但通过观察发现，输出结果中的部分“单词”是由标点符号两边的字串连接而成，如“king's”会被认为是“kings”，这样连接得到的结果并不是真正的单词，或者是错误的单词。因此考虑更改程序逻辑，将输入文件中的标点符号替换为分隔符。再次运行得到结果如下。



我认为这一结果更具实际意义。

四、思考与讨论

查看 log 信息发现，WordCount 作业的 Map 用时 8190 毫秒，Reduce 用时 2199 毫秒，排序作业的 Map 用时 1524 毫秒，Reduce 用时 1455 毫秒，程序总用时在 15 秒左右。

由于标点符号均为单个字符，且数量不大，我认为遍历标点符号将其替换的复杂度不高，不会太影响程序效率。而停词为字符串，且数量比较大，我认为若使用字典树等数据结构对

停词做匹配能有效降低时间复杂度，提高程序效率。

由于我有一定的 Java 开发经验，本次作业的环境配置对我而言不算困难，大部分时间花在了 Hadoop MapReduce 程序的编写上。即便本次作业的程序逻辑不算复杂，参照网上的教程等可以很快做出结果，但由于涉及了较多的 Hadoop MapReduce API，真正理解并熟练运用这些 API 仍需要大量时间和精力。