

//王泽宇\_181250148\_金融大数据处理技术作业 6  
//Git 仓库链接: <https://github.com/ZeyuuOne/MapReduceFriends>

## 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
build/libs	jar 文件所在目录
gradle/wrapper	gradle-wrapper 所在目录
src/main/java	源代码所在目录
.gitignore	.gitignore 文件
build.gradle	gradle 编译配置文件
gradlew	gradlew Linux Shell 脚本
gradlew.bat	gradlew Windows 脚本
README.pdf	作业报告
settings.gradle	gradle 配置文件

Jar 文件中包含两个可作为程序入口的类,运行时指定 BasicFriends 类即运行使用基本数据类型实现的程序,指定 AdvancedFriends 类即运行使用自定义数据类型实现的程序。

## 二、MapReduce 程序设计

程序的目的是求出用户集合上两两用户之间的共同好友,其中好友关系为单向关系,每个用户由唯一的整数 ID 标识。程序输入为形如<用户 ID, 好友 ID 列表>的键值对,程序输出<用户 ID 对, 共同好友 ID 列表>的键值对。程序实现没有参考互联网上的样例,由我独立开发完成。

程序设计如下:

1. 将输入的好友 ID 列表拆分,处理为形如<好友 ID, 用户 ID>的键值对,即求好友关系矩阵的转置;
2. 归并用户 ID,形成形如<好友 ID, 用户 ID 列表>的键值对,对每个用户 ID 列表,求其两两组合(即求其与自身笛卡尔积矩阵的严格上三角或下三角部分),处理为形如<用户 ID 对, 共同好友 ID>的键值对;
3. 归并共同好友 ID,形成形如<用户 ID 对, 共同好友 ID 列表>的键值对。

由于好友关系为单向关系,因此在求用户 ID 的两两组合之前必须对好友关系矩阵求转置。由于用户 ID 对当中两个用户是并列关系,不存在次序,同时也无需考虑用户和自身的共同好友,因此只需计算用户 ID 列表与自身笛卡尔积矩阵的严格上三角或下三角部分。

由于对相同 key 所对应 value 的归并由 MapReduce 框架的 Reduce 步骤提供,因此采用两个 Job 来实现上述程序设计。其中第 1 步由 job1 的 Mapper 完成,第 2 步由 job1 的 Reducer 完成,第 3 步由 job2 的 Reducer 完成。

### 三、使用基本数据类型实现的 MapReduce 程序

首先定义 TranspositionMapper 类，在 map() 函数中拆分输入行，以第 1 个 ID 为用户 ID，其余 ID 为好友 ID，输出<好友 ID，用户 ID>键值对。

```
1. @Override
2. public void map(Object key, Text value, Context context) throws IOException,
   InterruptedException {
3.     String line = value.toString().replace(",", "");
4.     StringTokenizer itr = new StringTokenizer(line);
5.     if (!itr.hasMoreTokens()) return;
6.     host.set(Integer.parseInt(itr.nextToken()));
7.     while (itr.hasMoreTokens()) {
8.         friend.set(Integer.parseInt(itr.nextToken()));
9.         context.write(friend, host);
10.    }
11. }
```

而后定义 PairingReducer 类，在 reduce() 函数中将 values 处理为两两组合。经过长时间的测试和查询，发现 Iterable 类型的 values 不允许两个独立的 Iterator 实例同时存在，即使将 values.iterator() 赋值给两个 Iterator 变量，它们对应的仍是同一个实例，因此无法对 values 直接执行两重遍历来获取其和自身的笛卡尔积。

又由于 values 中的用户是并列关系，不存在次序，且 value 类型为整数，因此考虑先将 values 处理为 TreeSet 类型，再处理为两两组合。又经过测试和查询，发现 com.google.common.collect.Sets.newTreeSet() 无法将此处的 Iterable 正确处理为 TreeSet，因此自己实现这一过程。

由于要使用基本数据类型表示用户 ID 对，因此选用 Text 类型，直接存储用户 ID 对的字符串形式。

```
1. @Override
2. public void reduce(IntWritable key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
3.     Set<Integer> hosts = new TreeSet<>();
4.     for (IntWritable val : values) {
5.         hosts.add(val.get());
6.     }
7.     for (int i = 0; i < hosts.size(); i++) {
8.         for (int j = i + 1; j < hosts.size(); j++) {
9.             pair.set(hosts.toArray()[i].toString() + ", " + hosts.toArray()[j].toString());
10.            context.write(pair, key);
11.        }
12.    }
13. }
```

接着定义 MergingReducer 类，在 reduce()函数中将 values 处理为字符串，选用 Text 类型表示共同好友 ID 列表。

```
1. @Override
2. public void reduce(Text key, Iterable<IntWritable> values, Context context)
   throws IOException, InterruptedException {
3.     StringBuilder friendsString = new StringBuilder();
4.     boolean first = true;
5.     for (IntWritable val : values) {
6.         if (!first) {
7.             friendsString.append(", ");
8.         } else {
9.             first = false;
10.        }
11.        friendsString.append(val.toString());
12.    }
13.    friends.set(friendsString.toString());
14.    context.write(key, friends);
15. }
```

最后定义 FriendsOutputFormat 类和 FriendsRecordWriter 类，在 write()函数中按格式输出最终结果。

```
1. @Override
2. public void write(Text key, Text value) throws IOException, InterruptedExcep
   tion {
3.     fos.write(("[" + key.toString() + "], [" + value.toString() + "])\n").g
       etBytes());
4. }
```

由于 Mapper 输出的键值对类型和 Reducer 输出的键值对类型不同，在 main()函数中单独设置 Mapper 的输出类型。以 SequenceFile 作为 job1 和 job2 之间的数据传递格式。由于 job2 无需 Map 步骤，因此使用默认的 IdentityMapper 即可。

```
1. job1.setMapOutputKeyClass(IntWritable.class);
2. job1.setMapOutputValueClass(IntWritable.class);
3. job1.setOutputFormatClass(SequenceFileOutputFormat.class);
4.
5. job2.setInputFormatClass(SequenceFileInputFormat.class);
6. job2.setMapOutputKeyClass(Text.class);
7. job2.setMapOutputValueClass(IntWritable.class);
```

使用基本数据类型实现的好友 MapReduce 程序即编写完成。

## 四、使用自定义数据类型实现的 MapReduce 程序

由于程序中的用户 ID 对和共同好友 ID 列表中的元素都是并列关系，不存在次序，只需定义一个 ID 集合类型即可满足程序需求，因此在上述基本数据类型实现的程序基础上，定义继承自 `TreeSet<Integer>`、使用 `WritableComparable` 接口的 `IntSet` 类，替换 `Text` 类型。

在 `IntSet` 类中，IDE 提示需要实现 `WritableComparable` 接口的纯虚函数 `compareTo()`、`write()`和 `readFields()`。在 `compareTo()`函数中，参考字符串比较大小的方式，首先对集合中最小的元素作比较，若相等则再比较下一位。若比较到最大元素仍相等，则 `size()`更大者视为大。若 `size()`也相等，则视为相等。

```
1. @Override
2. public int compareTo(Object o) {
3.     for (int i = 0; i < this.size(); i++) {
4.         if (i >= ((IntSet) o).size()) return 1;
5.         if (((Integer) this.toArray()[i]).compareTo((Integer) ((IntSet) o).toArray()[i]) > 0) return 1;
6.         else if (((Integer) this.toArray()[i]).compareTo((Integer) ((IntSet) o).toArray()[i]) < 0) return -1;
7.     }
8.     if (this.size() == ((IntSet) o).size()) return 0;
9.     else return -1;
10. }
```

在 `write()`和 `readFields()`函数中，先读写 `this.size()`，再依次读写每个元素。经过测试，发现在 `readFields()`函数中需要先调用 `this.clear()`将集合清空，否则会使后读 `SequenceFile` 得到的集合有先读集合的所有元素，猜测是因为程序框架使用同一个实例来初始化各个实例。

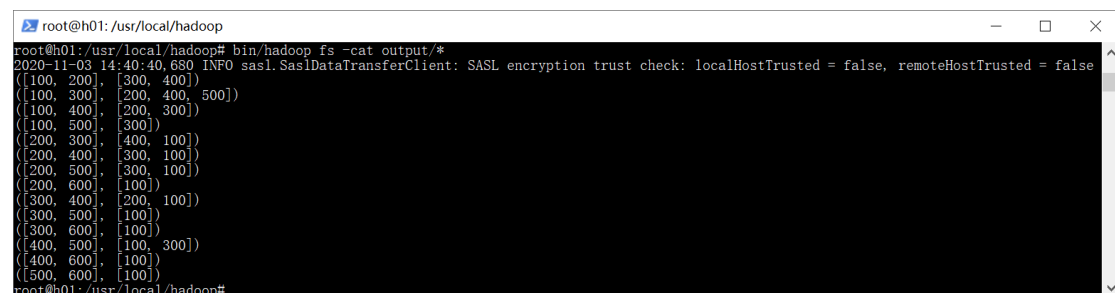
```
1. @Override
2. public void write(DataOutput out) throws IOException {
3.     out.writeInt(this.size());
4.     for (Integer i : this) {
5.         out.writeInt(i);
6.     }
7. }
8.
9. @Override
10. public void readFields(DataInput in) throws IOException {
11.     this.clear();
12.     int s = in.readInt();
13.     for (int i = 0; i < s; i++) {
14.         int t = in.readInt();
15.         this.add(t);
16.     }
17. }
```

同时改写 PairingReducer 类、MergingReducer 类和 FriendsRecordWriter 类的输入输出类型和相关输入输出代码，并在 main() 函数中设置对应的类型。

使用自定义数据类型实现的好友 MapReduce 程序即编写完成。

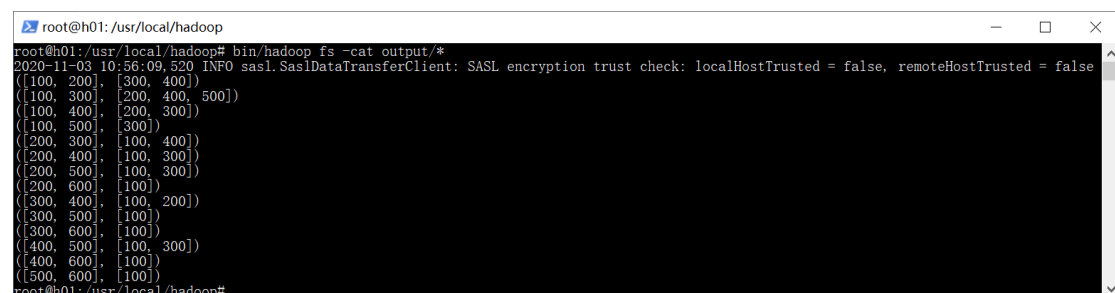
## 五、集群运行测试

在 Hadoop 集群运行 BasicFriends 类（使用基本数据类型实现），结果如下：



```
root@h01: /usr/local/hadoop
root@h01:/usr/local/hadoop# bin/hadoop fs -cat output/*
2020-11-03 14:40:40.680 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
([100, 200], [300, 400])
([100, 300], [200, 400, 500])
([100, 400], [200, 300])
([100, 500], [300])
([200, 300], [400, 100])
([200, 400], [300, 100])
([200, 500], [300, 100])
([200, 600], [100])
([300, 400], [200, 100])
([300, 500], [100])
([300, 600], [100])
([400, 500], [100, 300])
([400, 600], [100])
([500, 600], [100])
root@h01:/usr/local/hadoop#
```

在 Hadoop 集群运行 AdvancedFriends 类（使用自定义数据类型实现），结果如下：



```
root@h01: /usr/local/hadoop
root@h01:/usr/local/hadoop# bin/hadoop fs -cat output/*
2020-11-03 10:56:09.520 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
([100, 200], [300, 400])
([100, 300], [200, 400, 500])
([100, 400], [200, 300])
([100, 500], [300])
([200, 300], [100, 400])
([200, 400], [100, 300])
([200, 500], [100, 300])
([200, 600], [100])
([300, 400], [100, 200])
([300, 500], [100])
([300, 600], [100])
([400, 500], [100, 300])
([400, 600], [100])
([500, 600], [100])
root@h01:/usr/local/hadoop#
```

可见结果符合预期。且由于 AdvancedFriends 类中共同好友 ID 列表采用 Set 实现，共同好友 ID 是由小至大排序的。

## 六、思考与讨论

查看 log 信息发现，BasicFriends 类 job1 的 Map 用时 4710 毫秒，Reduce 用时 2448 毫秒，job2 的 Map 用时 1527 毫秒，Reduce 用时 1858 秒；AdvancedFriends 类 job1 的 Map 用时 3179 毫秒，Reduce 用时 1627 毫秒，job2 的 Map 用时 1419 毫秒，job2 的 Reduce 用时 1598 毫秒。可见使用自定义数据类型实现的程序效率显著高于使用基本数据类型实现的程序，猜测是因为 Text 的处理效率低于 IntSet 的处理效率。

本次作业涉及的新 API 不多，但诸如 Iterable 类型的遍历、Mapper 输出类型的设置等问题需要一定时间来找到并解决。在有足够 Java 面向对象程序编写基础的情况下，自定义数据类型的实现不算困难。