

以直播间数目以及直播热度为依据研究直播管理方案

用hadoop+spark简易实现对直播相关数据的处理



2020-11-28

南京大学软件学院18级

组长

王崇羽

组员

何知谦

刘一也

蒋沂霄

周褀祯

# 以直播间数目以及直播热度为依据研究直播管理方案

[以直播间数目以及直播热度为依据研究直播管理方案](#header-n254)  
 [零、小组简介](#header-n257)  
 [一、业务场景](#header-n266)  
 [1.1 平台内问题](#header-n269)  
 [1.2 平台间问题](#header-n280)  
 [1.3 问题规约](#header-n287)  
 [二、数据获取](#header-n291)  
 [2.1 数据来源：](#header-n292)  
 [2.2 虎牙：](#header-n295)  
 [2.2.1 数据获取](#header-n298)  
 [2.2.2 数据格式](#header-n305)  
 [2.2.3 预处理及存储](#header-n308)  
 [2.3 斗鱼：](#header-n318)  
 [2.3.1 数据获取](#header-n323)  
 [2.3.2 数据格式](#header-n330)  
 [2.3.3 预处理及存储](#header-n333)  
 [2.4 存储文件截图：](#header-n344)  
 [三、流式计算](#header-n112)  
 [3.1 流准备和监听](#header-n113)  
 [3.1.1 流准备](#header-n114)  
 [3.1.2流监听](#header-n121)  
 [平台内监听（short*time*streaming.py）](#header-n124)  
 [平台间监听](#header-n127)  
 [3.2 流计算](#header-n135)  
 [3.2.1 平台内项目（short*time*streaming.py）](#header-n136)  
 [3.2.2 平台间项目（long*time*streaming.py）](#header-n763)  
 [四、计算结果动态展示](#header-n217)  
 [4.1 index.html](#header-n782)  
 [4.2 render.js](#header-n787)  
 [4.3 其他.js文件](#header-n794)  
 [4.4 test.sh](#header-n799)  
 [4.5 test.txt](#header-n851)  
 [五、业务需求实现情况](#header-n218)  
 [六、困难&解决](#header-n219)

## 零、小组简介

组号：14

选题：直播数据相关

组长：181830175王崇羽，主要负责数据预处理、流准备与监听、流计算、计算结果可视化

组员：181830072何知谦 ，主要负责数据预处理、流计算

181830123刘一也 ，主要负责计算结果可视化

181250059蒋沂霄 ，主要负责数据爬取、数据预处理

181250206周褀祯，主要负责数据爬取、数据预处理

## 一、业务场景

2015年是被称为直播元年的一年。自此之后，各式各样的直播如雨后春笋般冒出。无论是刚起步的直播平台，亦或是获得了成熟发展的直播平台，都需要面临对一个至关重要的问题——如何合理管理直播。

本小组通过对直播间数量以及直播人气进行处理分析，从而能给出一份可以动态调整的直播管理方案。本小组的业务需求计算一共有6个，可以分为平台内问题与平台间问题两个板块：

### 1.1 平台内问题

通过实时数据的变化的流式计算以能够实时动态管理直播平台。

1. 直播间总数目与直播间总人气的动态计算
2. 每个直播间的平均人气动态计算
3. 不同板块直播间数目与人气增减的动态计算
4. 不同板块直播间平均人气的动态计算

### 1.2 平台间问题

通过比较不同平台间一段时间内的数据计算（形式仍然通过流式计算来进行）以能够制定一个更有针对性的管理策略。

1. 不同平台间直播间总数目与直播间总人气的变化计算
2. 不­­­­­­­­­­­­­­­­­­­­­同平台间不同板块直播间数目与人气的变化计算

### 1.3 问题规约

考虑到数据爬取成本与效率，我们选取了斗鱼与虎牙两个直播平台每个时间段排名前十的板块作为研究对象。其他板块的管理成本相对较低，可以不计入本次的动态管理研究之中。在研究平台内问题时，我们将不考虑平台间的对比，只考虑单平台内的纵向时间对比。

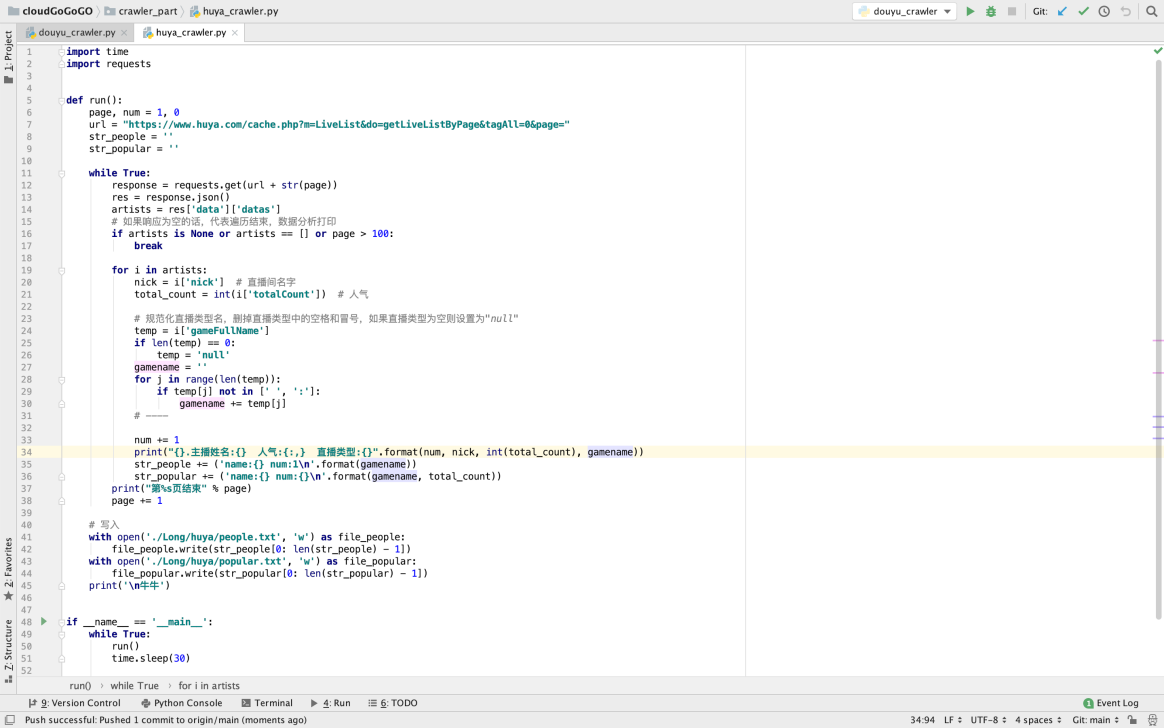
## 二、数据获取

### 2.1 数据来源：

数据来源于虎牙和斗鱼直播平台，包括两个平台当前所有在播的直播间的各项数据（包括直播间名字、介绍、分类、标语、热度等），下面将分别介绍虎牙、斗鱼直播数据的获取、存储及预处理过程，并在最后一部分放上数据在存储文件中的格式截图

### 2.2 虎牙：

代码如下图：



#### 2.2.1 数据获取

每一页数据需要单独爬取，url格式为：

**"https://www.huya.com/cache.php?m=LiveList&do=getLiveListByPage&tagAll=0&page="+page**

如第二十页数据url为：

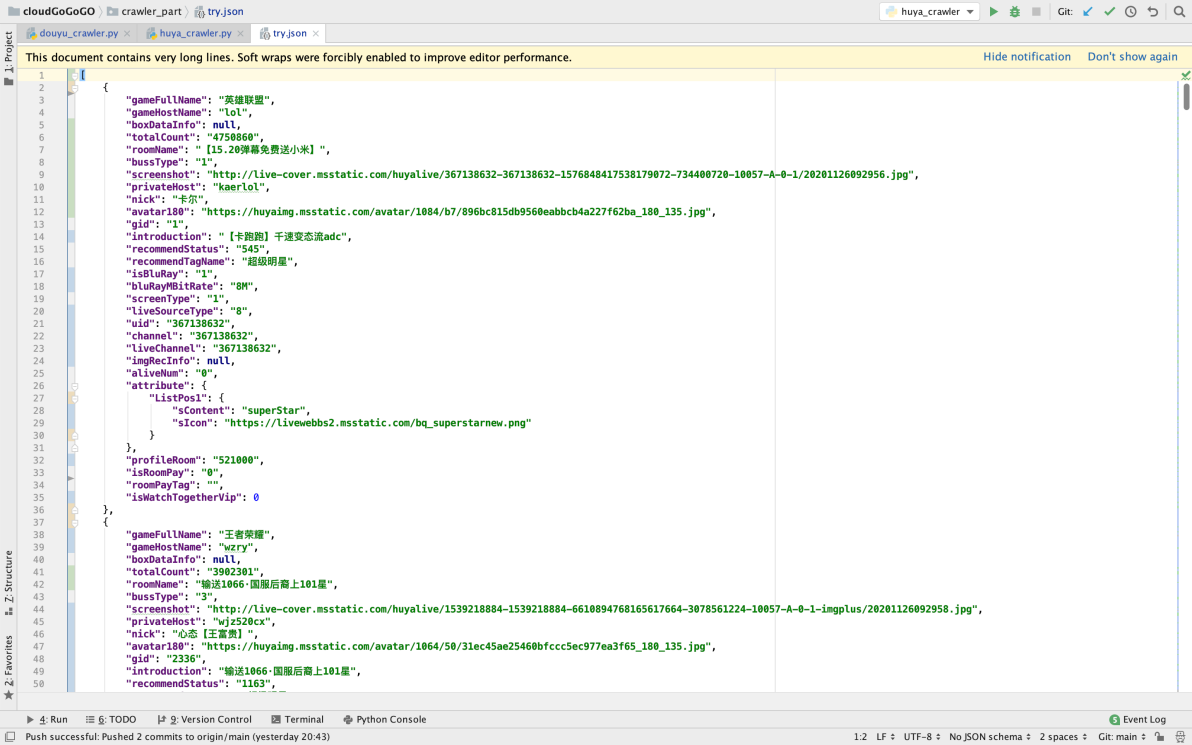
**https://www.huya.com/cache.php?m=LiveList&do=getLiveListByPage&tagAll=0&page=20**

用requests包中get函数对每一页循环爬取，获得一个包含服务器资源的Response对象response(代码11～12行)

由于爬取到的直播间数据默认按照热度降序排列，且排名靠后的直播间几乎没有人观看，对于数据分析用处不大，为爬取效率考虑，只爬取前100页数据(代码16～17行)，可更改爬取页数以调整爬取效果(代码16行)

#### 2.2.2 数据格式

将数据获取阶段获得的Response对象转为json对象，只取其中‘data’部分存入artists(代码13～14行)，artists内容如下图



#### 2.2.3 预处理及存储

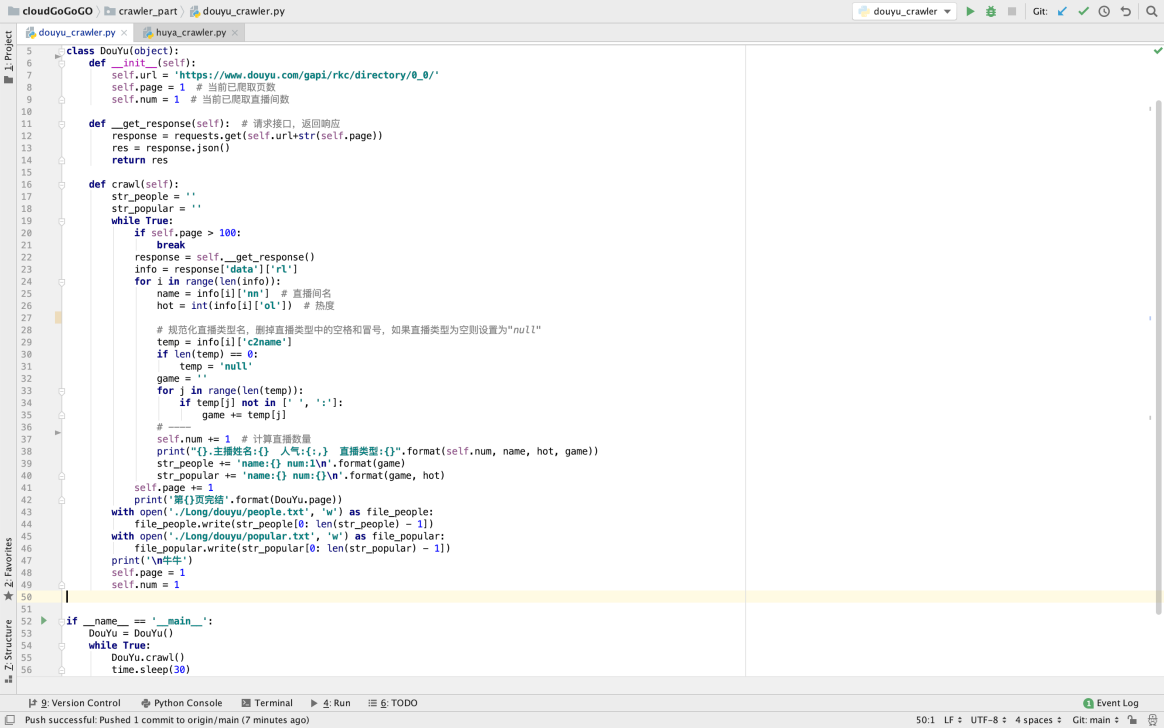
* 由于artists中大部分数据对分析无用，如直播间名字、标语等，所以只抽取直播类型和直播间热度数据存入文档(上图中gameFullName和totalCount两项)。
* 对artists中每一项循环(代码19行)，每一项都是单个直播间的数据
* 首先对直播类型数据进行处理：去除直播类型名称中的空格和冒号，如果直播类型为空的话则设置为null，例如将“CS:GO”改为“CSGO”(代码24～30行)
* 将该直播间的类型以“name:{} num:1”的格式添加到字符串str\_people末尾，其中name后接该直播间类型，如“name:英雄联盟 num:1”，每条数据都单独占一行(代码35行)
* 将该直播间的热度以“name:{} num:{}”的格式添加到字符串str\_popular末尾，其中name后接该直播间类型，num后接该直播间热度，如“name:英雄联盟 num:10037”，每条数据都单独占一行(代码36行)
* long模式下，前100页循环爬取完毕后，将str*people和str*popular字符串分别写入项目下的crawler*part/Long/huya/people.txt和crawler*part/Long/huya/popular.txt文件(代码41～44行)，如果run的时候此处报错的话请将上述路径改为完整路径，两个文件都在项目文件中/crawler\_part/Long/huya/目录下
* short模式下，前100页循环爬取完毕后，将str*people和str*popular字符串分别写入项目下的crawler*part/short/people.txt和crawler*part/short/popular.txt文件(代码41～44行，需要自行修改代码中文件路径)，如果run的时候此处报错的话请将上述路径改为完整路径，两个文件都在项目文件中/crawler\_part/short/目录下
* 本次爬取完成，打印一个“牛牛”表示做得很棒

### 2.3 斗鱼：

采用了面向对象编程，虽然并没有什么用。

爬取时先生成DouYu对象，每一次爬取调用crawl方法即可。

DouYu类代码如下图：

******

#### 2.3.1 数据获取

每一页数据依然需要单独爬取，url格式为：

"https://www.douyu.com/gapi/rkc/directory/0\_0/"+page

如第六十八页数据url为：

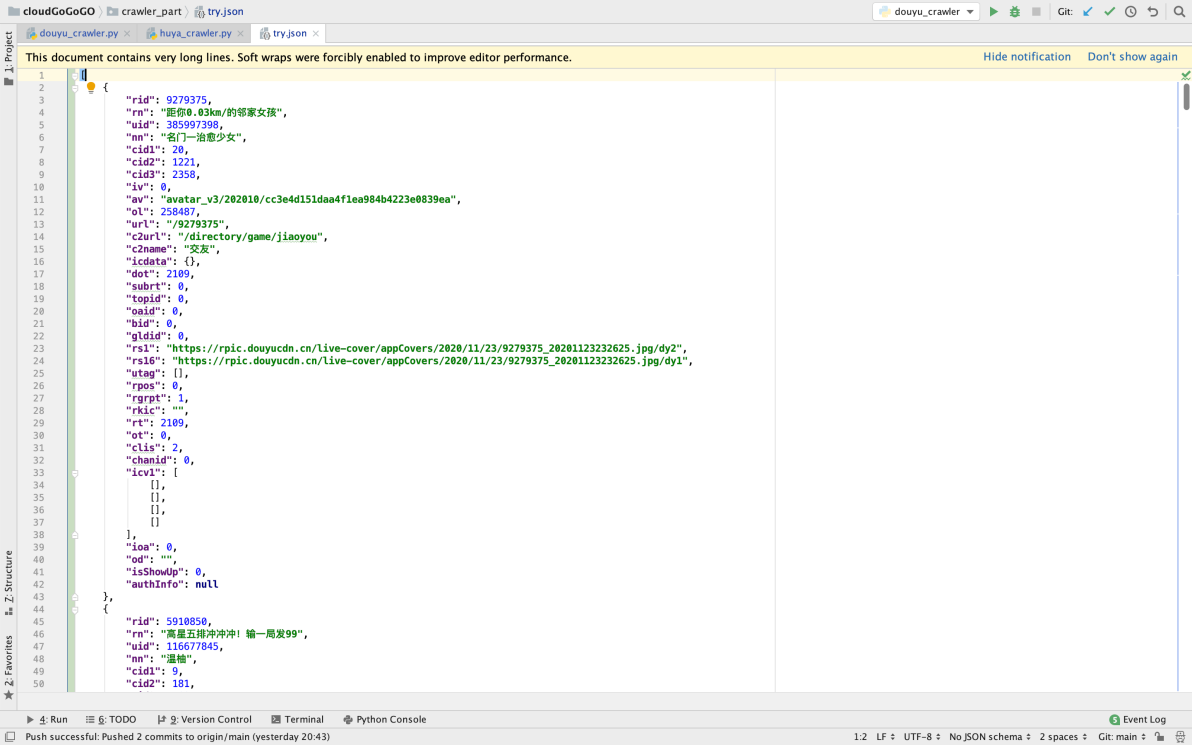
[**https://www.douyu.com/gapi/rkc/directory/0\_0/6**](https://www.huya.com/cache.php?m=LiveList&do=getLiveListByPage&tagAll=0&page=20)**8**

用requests包中get函数对每一页循环爬取，获得一个包含服务器资源的Response对象response，将其转换为json对象并返回该对象(代码11～14行)

由于爬取到的直播间数据默认按照热度降序排列，且排名靠后的直播间几乎没有人观看，对于数据分析用处不大，所以只爬取前100页数据(代码20～21行)，可更改爬取页数以调整爬取效果(代码20行)

#### 2.3.2 数据格式

将数据获取阶段获得的json对象取‘data’部分存入info(代码22～23行)，info内容如下图

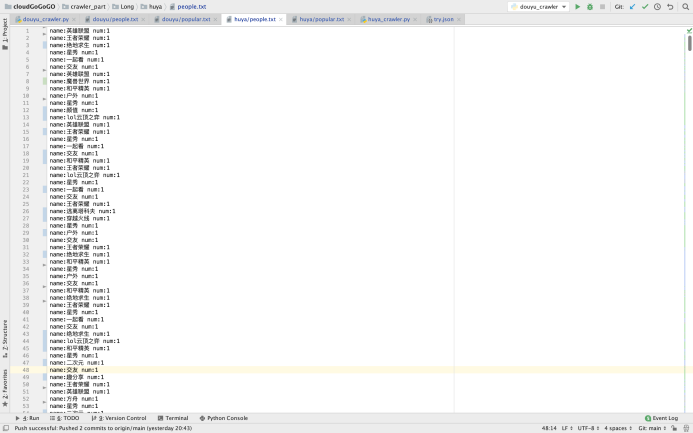


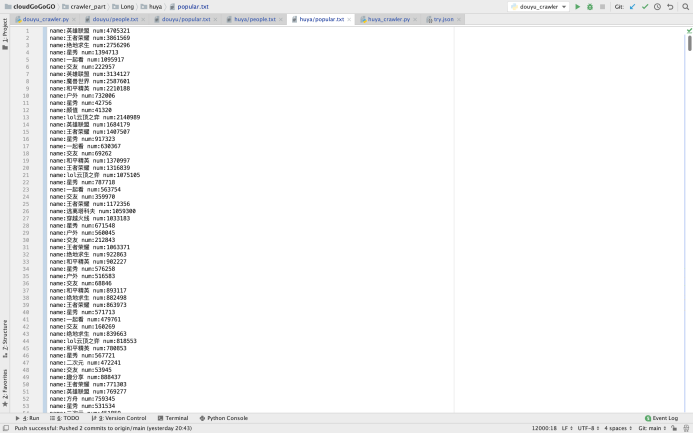
#### 2.3.3 预处理及存储

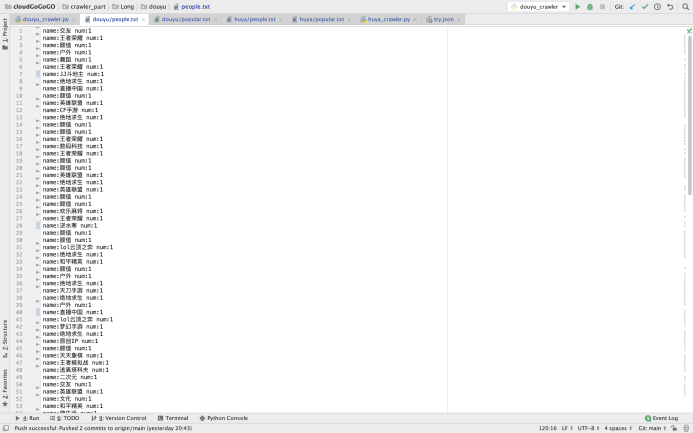
* 由于info中大部分数据对分析无用，如直播间名字、标语等，所以只抽取直播类型和直播间热度数据存入文档(上图中c2name和ol两项)。
* 对info中每一项循环(代码24行)，每一项都是单个直播间的数据
* 首先对直播类型数据进行处理：去除直播类型名称中的空格和冒号，如果直播类型为空的话则设置为null，例如将“CS:GO”改为“CSGO”(代码29～35行)
* 将该直播间的类型以“name:{} num:1”的格式添加到字符串str\_people末尾，其中name后接该直播间类型，如“name:英雄联盟 num:1”，每条数据都单独占一行(代码39行)
* 将该直播间的热度以“name:{} num:{}”的格式添加到字符串str\_popular末尾，其中name后接该直播间类型，num后接该直播间热度，如“name:英雄联盟 num:10037”，每条数据都单独占一行(代码40行)
* long模式下，前100页循环爬取完后，将str*people和str*popular字符串分别写入项目下的crawler*part/Long/douyu/people.txt和crawler*part/Long/douyu/popular.txt文件(代码43～46行)，如果run的时候此处报错的话请将上述路径改为完整路径，两个文件都在项目文件中/crawler\_part/Long/douyu/目录下
* short模式下，前100页循环爬取完后，将str*people和str*popular字符串分别写入项目下的crawler*part/short/people.txt和crawler*part/short/popular.txt文件(代码43～46行，需要自行修改代码中文件路径)，如果run的时候此处报错的话请将上述路径改为完整路径，两个文件都在项目文件中/crawler\_part/short/目录下
* 本次爬取完成，打印一个“牛牛”表示做得很棒
* DouYu对象只创建一次，但要进行多次爬取，所以每次调用crawl方法最后需要将类变量重置，即将page(页数)和num(直播间数量)设置为1(代码48～49行)

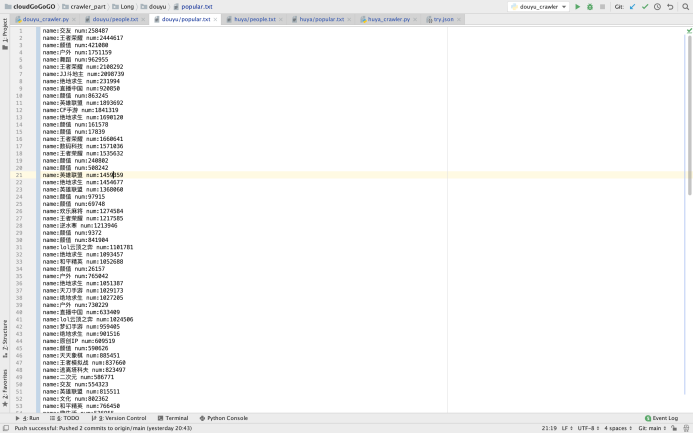
### 2.4 存储文件截图：

下面四张图按顺序依次是虎牙的people.txt文件、虎牙的popular.txt文件、斗鱼的people.txt文件、斗鱼的popular.txt文件









## 三、流式计算

### 3.1 流准备和监听

#### 3.1.1 流准备

这里展示upload.sh（平台内问题流准备脚本）进行解释说明。uploadlong.sh（平台间问题流准备脚本）实现逻辑与upload.sh类似，只是加入了平台区分上传。

#!/bin/bash  
  
# 指定hdfs dfs命令的路径  
hdfs="${HADOOP\_HOME}/bin/hdfs dfs"   
  
# 清空用于监听的hdfs文件夹  
$hdfs -rm -r /origin/people/  
$hdfs -rm -r /origin/popular/  
$hdfs -mkdir /origin/people/  
$hdfs -mkdir /origin/popular/  
  
# 创造一个死循环，反复将本地的文件打上时间戳上传到hdfs上。这里的平均处理时间是5s，所以sleep85是为了能与流监听的间隔时间90s匹配。  
while [ 1 ]   
do  
 tmp1="people`date +'%s'`.txt"  
 tmp2="popular`date +'%s'`.txt"  
 $hdfs -put -f /home/pluviophile/Documents/tmp/origin/people.txt /origin/people/$tmp1  
 $hdfs -put -f /home/pluviophile/Documents/tmp/origin/popular.txt /origin/popular/$tmp2  
 sleep 85  
done

-rm -r是删除整个文件夹。

-mkdir是新建文件夹。需要注意的是，本小组实验时已经提前使用该指令在hdfs上创建了以下文件目录：

/origin/people/ # short模式下直播人数统计  
/origin/popular/ # short模式下直播人气统计  
/longtime/douyu/people/ # long模式下斗鱼直播人数统计  
/longtime/douyu/popular/ # long模式下斗鱼直播人气统计  
/longtime/huya/people/ # long模式下虎牙直播人数统计  
/longtime/huya/popular/ # long模式下虎牙直播人气统计

-put -f是强制上传本地文件到hdfs上。第一个参数是本地文件路径，第二个参数是hdfs文件路径。需要注意的是，之前预处理板块已经提到，预处理过程实际上是对本地同一个文件的持续更新，所以第一个参数始终不变；而第二个参数需要打上时间戳是为了保证有新的文件上传到hdfs文件系统上，以便被监听到。

#### 3.1.2 流监听

注意：下列文件系统hdfs://wcy-pc:9000中wcy-pc是hdfs的namenode主机名。应用时需要修改此处配置。

使用语言为Python。

##### 平台内监听（shorttimestreaming.py）

# 设置流监听，监听源为hdfs文件系统  
 sc = SparkContext(appName="PythonStreamingShortTime")  
 ssc = StreamingContext(sc, 90)  
 people = ssc.textFileStream("hdfs://wcy-pc:9000/origin/people/")  
 popu = ssc.textFileStream("hdfs://wcy-pc:9000/origin/popular/")

这里people与popu是对传入的人数与人气的监听。

##### 平台间监听

# 设置流监听，监听源为hdfs文件系统  
 sc = SparkContext(appName="PythonStreamingLongTime")  
 ssc = StreamingContext(sc, 90)  
 douyuPeople = ssc.textFileStream("hdfs://wcy-pc:9000/longtime/douyu/people/")  
 douyuPopu = ssc.textFileStream("hdfs://wcy-pc:9000/longtime/douyu/popular/")  
 huyaPeople = ssc.textFileStream("hdfs://wcy-pc:9000/longtime/huya/people/")  
 huyaPopu = ssc.textFileStream("hdfs://wcy-pc:9000/longtime/huya/popular/")

创建流计算上下文：

* sc：SparkContext，配置spark的应用名
* ssc：StreamingContext，按照sc的配置进行进一步设置。第二个参数是批处理间隔，此处设置为90s，设置较长主要是因为数据爬取和上传文件至hdfs需要较长时间。
* 从hdfs获取已经经过了预处理的爬取数据，并实时监听该目录下文件的变化。（textFileStream自带监听变化功能，检测到变化则重新计算）

### 3.2 流计算

#### 3.2.1 平台内项目（short*time*streaming.py）

在main里面新建一个列表保存输出流，到最后一个函数时统一写入本地完成持久化

data = [] # 保存输出流

首先定义两个map函数：

**mapBynum**将平台中所有人数/人气提取出来，返回一个列表。

原始数据项为 name:名字 num:人数或 name:名字 num:人气

处理后列表中每个表项为[“num”，人数/人气]

需要注意的是，这里对数据做了降维处理，即统一将数据的key设置为“num”，value设置为对应数值，忽略了原数据的name字段。这样统一化value有助于reduceByKey统计计算总人气或总人数。

def mapBynum(x):  
 list1 = x.split(" ")[1].split(":")  
 list2 = ["num", int(list1[1])]  
 return list2

**mapByName**将平台中人数根据直播板块名字分组，返回一个列表。

原始数据项为 name:名字 people:人数或 name:名字 num:人气

处理后列表中每个表项为[名字，人数/人气]

这里将key设置为对应的板块名，有助于分别对板块进行统计。

def mapByName(x):  
 list1 = x.split(" ")  
 list2 = [(list1[0].split(":"))[1], int((list1[1].split(":"))[1])]  
 return list2

**main内定义函数**为了传入foreachRDD中实现对每个RDD进行处理。main内有以下六个函数：

def outputpeopleCounts(time, rdd):   
def outputpopuCounts(time, rdd):   
def calculateForMean(time, rdd):   
def calculateForPeopleChange(time, rdd):   
def calculateForPopuChange(time, rdd):   
def calculateForNameMean(time, rdd):

其中time是系统时间，rdd是针对DStream中的每个RDD。后文将进一步解释上述函数的作用。

**以下内容在main中完成**。部分方法在main中完成定义是为了能使用在main中初始化的输出流data。

1. 计算单个平台的直播总人数（RDD中value为总人数）

（1）.map将源Dstream中的每个元素经过lambda中函数进行处理。这里的lambda函数为mapBynum，得到键值对形式的RDD（“num”，人数）。

（2）.reduceByKey将RDD中相同key的value通过迭代的方式使用对应lambda函数处理。此处lambda函数为加法。（此处为所有的RDD的value都相加，因为所有的key都是“num”）得到总人数

\*reduceByKey方法的进一步解释：假设RDD中有('a',1),('a',2),('a',3)，三者key相同，在使用reduceByKey时将先对其中两者进行lambda计算，计算结果再与剩余第三者进行计算。

后续有相似的语句.map().reduceByKey()皆是按上述逻辑运行，故后文不赘述。

peopleCounts = people.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

（3）进行进一步的处理。

* .repartition(1)是针对DStream的方法，这里通过将partitions的数量限制为1能确保数据的集中性，便于保存在全局输出流data中。
* .foreachRDD(function)可以将传入的function应用于DStream中的每个RDD上。
* 后续有相似的语句.repartition(1).foreachRDD()皆是按上述逻辑运行，故后文不赘述。
* 传入参数time后，进行流式计算时，将会延迟到特定的系统时间点进行运算，特定时间点与设置的时间间隔有关。如果设置的是20s，则会在系统时间每分钟的第0s，第20s、第40s进行计算。这设置的90s，则仅会在系统的第0s或第30s开始运算，且每隔一分半计算一次。
* rdd.take(num)是取出rdd内的记录前num项取出返回一个列表。注意到的是，take（num）中num可以大于RDD中实际的条目数。这里只取出了第一条是因为之前的reduceByKey方法能确保最终只剩下一条记录。
* print是为了方便在控制台进行调试。
* 在全局data列表中加入一条新的字符串，格式为"allpeople "+"people " + 直播间总数 + " " + 系统时间
* 其中allpeople是一个约定的命令，方便可视化处理，具体约定参见可视化章节。

def outputpeopleCounts(time, rdd):   
 print("-------------------------------------------")   
 print("outputpeopleCounts Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["allpeople", "people", record[1]])   
 item = "allpeople " + "people " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
peopleCounts.repartition(1).foreachRDD(outputpeopleCounts)

1. 计算单个平台的直播总人气（RDD中value为总人气）

popuCounts = popu.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

进行进一步处理。逻辑与上文相似，故不赘述。

def outputpopuCounts(time, rdd):   
 print("-------------------------------------------")   
 print("outputpopuCounts Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["allpopu", "popu", record[1]])   
 item = "allpopu " + "popu " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
popuCounts.repartition(1).foreachRDD(outputpopuCounts)

3．计算平均每个直播间的人气

（1）.union 将peopleCounts和popuCounts两个DStream的所有元素联合，生成一个新的DStream。此处peopleCounts中只有一条总直播间数的记录，popuCounts中只有一条总人气的记录，peopleCounts.union(popuCounts)后将有两条记录（“num”，总直播间数）以及（“num”，总人气）

（2）.reduceByKey将键值相同的RDD的value进行lambda计算，即为将popuCounts的value（即总人气）除以peopleCounts的value（即总直播间数）得到平均每个直播间的人气

popularForEachRoom = peopleCounts.union(popuCounts).reduceByKey(lambda a, b: (0.0 + float(b)) / a)

接着进行进一步处理。逻辑与上文相似，故不赘述。

注意到calculateForMean最后向输出流data加入了“clear”，这是个可视化命令，具体用途参见可视化章节。

def calculateForMean(time, rdd):   
 print("-------------------------------------------")   
 print("calculateForMean Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["allmean", record[1]])   
 item = "allmean " + str(record[1]) + " " + str(time)   
 data.append(item)   
 data.append("clear")   
   
   
popularForEachRoom.repartition(1).foreachRDD(calculateForMean)

4.计算平台前十直播人数版块的直播人数变更情况。（RDD中value为总人数）

这里使用的map函数是mapByName，即返回的键值对格式是（板块名，数值），这样可以按照不同的板块进行后续计算。

peopleChange = people.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

接着进行进一步处理。

* 经过统计，发现平均来说直播间总板块数在500以下，所以这里取rdd前500条条目出来进行计算。
* 由于没有对DStream进行直接按值排序的方法，这里考虑到每次任务时间间隔较长，500条数目的冒泡排序处理绰绰有余，于是采用了最简单的冒泡排序对rdd中的条目按值从高到低排序，这里即是按板块总人数来排序。最终取出前十进行处理，记录到输出流data中。
* 简单介绍一下算法。该函数calculateForPeopleChange是为了计算该时段的板块直播间数量相较于上次计算时发生的变化。这里有一个全局的列表historyOfPeopleEachName用于记录上一时间段的板块直播间数量。在初始化时和当前时间段的信息一致。随后每完成一次计算便会清空该列表并保存当前的时间段的板块直播间数量。currentPeopleEachName是保存当前的时间段的板块直播间数量的列表。calculateChangePeopleDateList是保存计算结果。在计算时，会使用currentPeopleEachName与historyOfPeopleEachName进行比较计算。会有以下三种情况出现：
  + currentPeopleEachName与historyOfPeopleEachName都有某一板块的记录。此时以列表的形式将[该版块名,当前板块直播间人数-历史板块直播间人数]保存进入calculateChangePeopleDateList。
  + currentPeopleEachName有某一板块的记录而historyOfPeopleEachName没有。说明有新的板块的直播人数排入前十。此时直接将currentPeopleEachName的[该版块名,当前板块直播间人数]以列表的形式保存进入calculateChangePeopleDateList。
  + historyOfPeopleEachName有某一板块的记录而currentPeopleEachName没有。说明有旧的板块的直播人数被挤出前十。此时直接将historyOfPeopleEachName的[该版块名,该板块直播间人数的历史人数的相反数]以列表的形式保存进入calculateChangePeopleDateList。
* 最后在全局data列表中循环加入calculateChangePeopleDateList中的记录，格式为"peoplechange " + str(record[0]) + " " + str(record[1]) + " " + 系统时间
* 其中peoplechange 是可视化命令，具体参见可视化板块。
* record[0]与record[1]对应的是calculateChangePeopleDateList内每一条列表的板块名与板块变动数值。

def calculateForPeopleChange(time, rdd):   
 print("-------------------------------------------")   
 print("calculateForPeopleChange Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(501)   
 currentPeopleEachName = []   
 calculateChangePeopleDateList = []   
 tmpHistoryName = []   
 tmpCurrntName = []   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 for record in taken[:10]:   
 currentPeopleEachName.append(record)   
 tmpCurrntName.append(record[0])   
 if len(historyOfPeopleEachName) == 0:   
 for record in currentPeopleEachName:   
 historyOfPeopleEachName.append(record)   
 for i in historyOfPeopleEachName:   
 tmpHistoryName.append(i[0])   
 for i in range(0, len(currentPeopleEachName)):   
 if currentPeopleEachName[i][0] in tmpHistoryName:   
 for j in range(0, len(historyOfPeopleEachName)):   
 if currentPeopleEachName[i][0] == historyOfPeopleEachName[j][0]:   
 tmplist = [currentPeopleEachName[i][0],   
 int(currentPeopleEachName[i][1]) - int(historyOfPeopleEachName[j][1])]   
 calculateChangePeopleDateList.append(tmplist)   
 else:   
 tmplist = [currentPeopleEachName[i][0], int(currentPeopleEachName[i][1])]   
 calculateChangePeopleDateList.append(tmplist)   
 for i in range(0, len(historyOfPeopleEachName)):   
 if historyOfPeopleEachName[i][0] not in tmpCurrntName:   
 tmplist = [historyOfPeopleEachName[i][0], 0 - int(historyOfPeopleEachName[i][1])]   
 calculateChangePeopleDateList.append(tmplist)   
 historyOfPeopleEachName.clear()   
 for record in currentPeopleEachName:   
 historyOfPeopleEachName.append(record)   
 print(calculateChangePeopleDateList)   
 for record in calculateChangePeopleDateList:   
 item = "peoplechange " + str(record[0]) + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
peopleChange.repartition(1).foreachRDD(calculateForPeopleChange)

5.计算平台前十直播人数版块的直播人数变更情况。（RDD中value为总人气）

popuChange = popu.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

接着进行进一步处理。逻辑与上文相似，故不赘述。

def calculateForPopuChange(time, rdd):   
 print("-------------------------------------------")   
 print("calculateForPopuChange Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(501)   
 currentPopuEachName = []   
 calculateChangePopuDateList = []   
 tmpHistoryName = []   
 tmpCurrntName = []   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 for record in taken[:10]:   
 currentPopuEachName.append(record)   
 tmpCurrntName.append(record[0])   
 if len(historyOfPopuEachName) == 0:   
 for record in currentPopuEachName:   
 historyOfPopuEachName.append(record)   
 for i in historyOfPopuEachName:   
 tmpHistoryName.append(i[0])   
 for i in range(0, len(currentPopuEachName)):   
 if currentPopuEachName[i][0] in tmpHistoryName:   
 for j in range(0, len(historyOfPopuEachName)):   
 if currentPopuEachName[i][0] == historyOfPopuEachName[j][0]:   
 tmplist = [currentPopuEachName[i][0],   
 int(currentPopuEachName[i][1]) - int(historyOfPopuEachName[j][1])]   
 calculateChangePopuDateList.append(tmplist)   
 else:   
 tmplist = [currentPopuEachName[i][0], int(currentPopuEachName[i][1])]   
 calculateChangePopuDateList.append(tmplist)   
 for i in range(0, len(historyOfPopuEachName)):   
 if historyOfPopuEachName[i][0] not in tmpCurrntName:   
 tmplist = [historyOfPopuEachName[i][0], 0 - int(historyOfPopuEachName[i][1])]   
 calculateChangePopuDateList.append(tmplist)   
 historyOfPopuEachName.clear()   
 for record in currentPopuEachName:   
 historyOfPopuEachName.append(record)   
 print(calculateChangePopuDateList)   
 for record in calculateChangePopuDateList:   
 item = "popuchange " + str(record[0]) + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
 # [("PopuChange",record[0],record[1])]   
   
   
popuChange.repartition(1).foreachRDD(calculateForPopuChange)

6.计算各个板块每个直播间的平均热度。

（1）peopleWithName为按直播模块划分的人数

（2）popuWithName 为按直播模块划分的人气

（3）.join类似数据库中的join操作，将所有key相同的提取出来，value合并。此处为得到同一直播模块的总人气和总直播人数。格式为（直播模块名称，（直播总人气，直播总人数）），两个DStream中没有相同的key的字段将被抛弃。

peopleWithName = people.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)   
popuWithName = popu.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)   
meanByName = popuWithName.join(peopleWithName)

接着进行进一步处理。

* 这里同样做了一个冒泡排序取前十
* 对前十的每个直播模块使用直播总人气除以直播总人数
* 最后保存写入全局data输出流中，并将data流写入./shortOutput.txt中，最后清空全局data输出流。

def calculateForNameMean(time, rdd):   
 print("-------------------------------------------")   
 print("calculateForNameMean Time: %s" % time)   
 print("-------------------------------------------")   
 taken = rdd.take(501)   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 nameMeanList = []   
 for record in taken[:10]:   
 tmplist = [record[0], (0.0 + record[1][0]) / record[1][1]]   
 nameMeanList.append(tmplist)   
 print(nameMeanList)   
 for record in nameMeanList:   
 item = "namemean " + str(record[0]) + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
 # [("nameMean",record[0],record[1])]   
 file = open("./shortOutput.txt", "w", encoding='utf-8')   
 for i in data:   
 file.write(i + "\n")   
 file.close()   
 data.clear()   
   
meanByName.repartition(1).foreachRDD(calculateForNameMean)

最后启动ssc。

# 开始监听  
ssc.start()  
ssc.awaitTermination()

输出的本地./shortOutput.txt格式如下所示。

allpeople people 12000 2020-11-26 17:24:00   
allpopu popu 684303871 2020-11-26 17:24:00   
allmean 57025.322583333334 2020-11-26 17:24:00   
clear   
peoplechange 英雄联盟 0 2020-11-26 17:24:00   
peoplechange 颜值 0 2020-11-26 17:24:00   
peoplechange 王者荣耀 0 2020-11-26 17:24:00   
peoplechange 原神 0 2020-11-26 17:24:00   
peoplechange 绝地求生 0 2020-11-26 17:24:00   
peoplechange 一起看 0 2020-11-26 17:24:00   
peoplechange 和平精英 0 2020-11-26 17:24:00   
peoplechange 户外 0 2020-11-26 17:24:00   
peoplechange DNF 0 2020-11-26 17:24:00   
peoplechange 天刀手游 0 2020-11-26 17:24:00   
popuchange 王者荣耀 0 2020-11-26 17:24:00   
popuchange 颜值 0 2020-11-26 17:24:00   
popuchange 英雄联盟 0 2020-11-26 17:24:00   
popuchange 绝地求生 0 2020-11-26 17:24:00   
popuchange 和平精英 0 2020-11-26 17:24:00   
popuchange DNF 0 2020-11-26 17:24:00   
popuchange 天刀手游 0 2020-11-26 17:24:00   
popuchange 交友 0 2020-11-26 17:24:00   
popuchange 原神 0 2020-11-26 17:24:00   
popuchange CSGO 0 2020-11-26 17:24:00   
namemean 王者荣耀 100633.74042553191 2020-11-26 17:24:00   
namemean 颜值 82217.10138888888 2020-11-26 17:24:00   
namemean 英雄联盟 54039.15145005371 2020-11-26 17:24:00   
namemean 绝地求生 71521.49842271293 2020-11-26 17:24:00   
namemean 和平精英 91201.46825396825 2020-11-26 17:24:00   
namemean DNF 70563.14655172414 2020-11-26 17:24:00   
namemean 天刀手游 68228.07580174928 2020-11-26 17:24:00   
namemean 交友 100336.3922413793 2020-11-26 17:24:00   
namemean 原神 24967.882869692534 2020-11-26 17:24:00   
namemean CSGO 94175.91712707182 2020-11-26 17:24:00

#### 3.2.2 平台间项目（long*time*streaming.py）

在main里面新建一个列表保存输出流，到最后一个函数时统一写入本地完成持久化

data = [] # 保存输出流

首先定义两个map函数：

**mapBynum**将平台中所有人数/人气提取出来，返回一个列表。

原始数据项为 name:名字 num:人数或 name:名字 num:人气

处理后列表中每个表项为[“num”，人数/人气]

需要注意的是，这里对数据做了降维处理，即统一将数据的key设置为“num”，value设置为对应数值，忽略了原数据的name字段。这样统一化value有助于reduceByKey统计计算总人气或总人数。

def mapBynum(x):  
 list1 = x.split(" ")[1].split(":")  
 list2 = ["num", int(list1[1])]  
 return list2

**mapByName**将平台中人数根据直播板块名字分组，返回一个列表。

原始数据项为 name:名字 people:人数或 name:名字 num:人气

处理后列表中每个表项为[名字，人数/人气]

这里将key设置为对应的板块名，有助于分别对板块进行统计。

def mapByName(x):   
 list1 = x.split(" ")   
 list2 = [(list1[0].split(":"))[1], int((list1[1].split(":"))[1])]   
 return list2

**main内定义函数**为了传入foreachRDD中实现对每个RDD进行处理。main内有以下八个函数：

def outputDouyuPeopleCounts(time, rdd):   
def outputHuyaPeopleCounts(time, rdd):   
def outputDouyuPopuCounts(time, rdd):   
def outputHuyaPopuCounts(time, rdd):   
def outputDouyuPeopleCountsForeach(time, rdd):   
def outputHuyaPeopleCountsForeach(time, rdd):   
def outputDouyuPopuCountsForeach(time, rdd):   
def outputHuyaPopuCountsForeach(time, rdd):

其中time是系统时间，rdd是针对DStream中的每个RDD。后文将进一步解释上述函数的作用。

**以下内容在main中完成。**部分方法在main中完成定义是为了能使用在main中初始化的输出流data。

1.首先分别计算斗鱼和虎牙两个平台的直播总人数（RDD中value为总人数）

（1）.map将源Dstream中的每个元素经过lambda中函数进行处理。这里的lambda函数为mapBynum，得到键值对形式的RDD（“num”，人数）。

（2）.reduceByKey将RDD中相同key的value通过迭代的方式使用对应lambda函数处理。此处lambda函数为加法。（此处为所有的RDD的value都相加，因为所有的key都是“num”）得到总人数

\*reduceByKey方法的进一步解释：假设RDD中有('a',1),('a',2),('a',3)，三者key相同，在使用reduceByKey时将先对其中两者进行lambda计算，计算结果再与剩余第三者进行计算。

后续有相似的语句.map().reduceByKey()皆是按上述逻辑运行，故后文不赘述。

douyuPeopleCounts = douyuPeople.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)   
 huyaPeopleCounts = huyaPeople.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

（3）进行进一步的处理。

* .repartition(1)是针对DStream的方法，这里通过将partitions的数量限制为1能确保数据的集中性，便于保存在全局输出流data中。
* .foreachRDD(function)可以将传入的function应用于DStream中的每个RDD上。
* 后续有相似的语句.repartition(1).foreachRDD()皆是按上述逻辑运行，故后文不赘述。
* 注意到outputDouyuPeopleCounts与outputHuyaPeopleCounts是在main内定义的方法。以outputDouyuPeopleCounts为例阐释。outputHuyaPeopleCounts实现逻辑与outputDouyuPeopleCounts相似。
* 传入参数time后，进行流式计算时，将会延迟到特定的系统时间点进行运算，特定时间点与设置的时间间隔有关。如果设置的是20s，则会在系统时间每分钟的第0s，第20s、第40s进行计算。这设置的90s，则仅会在系统的第0s或第30s开始运算，且每隔一分半计算一次。
* rdd.take(num)是取出rdd内的记录前num项取出返回一个列表。注意到的是，take（num）中num可以大于RDD中实际的条目数。这里只取出了第一条是因为之前的reduceByKey方法能确保最终只剩下一条记录。
* print是为了方便在控制台进行调试。
* 在全局data列表中加入一条新的字符串，格式为"allpeople "+"douyu " + 斗鱼直播间总数 + " " + 系统时间
* 其中allpeople是一个约定的命令，方便可视化处理，具体约定参见可视化章节。
* douyu是平台名。

def outputDouyuPeopleCounts(time, rdd):   
 s = "Time: %s" % time   
 print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["斗鱼", "people", record[1]])   
 item = "allpeople "+"douyu " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
 douyuPeopleCounts.repartition(1).foreachRDD(outputDouyuPeopleCounts)   
   
   
 def outputHuyaPeopleCounts(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["虎牙", "people", record[1]])   
 item = "allpeople "+"huya " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
 huyaPeopleCounts.repartition(1).foreachRDD(outputHuyaPeopleCounts)

2.分别计算斗鱼和虎牙两个平台的直播人气总和（RDD中value为总人气）

douyuPopuCounts = douyuPopu.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)   
huyaPopuCounts = huyaPopu.map(lambda x: (mapBynum(x)[0], mapBynum(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

接着进行进一步处理。实现逻辑与上文处理直播人数总和的逻辑类似，故不赘述。

注意到outputHuyaPopuCounts最后向输出流data加入了“clear”，这是个可视化命令，具体用途参见可视化章节。

def outputDouyuPopuCounts(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["斗鱼", "popu", record[1]])   
 item = "allpopu "+"douyu " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
 douyuPopuCounts.repartition(1).foreachRDD(outputDouyuPopuCounts)   
   
   
 def outputHuyaPopuCounts(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(1)   
 if len(taken) > 0:   
 record = taken[0]   
 print(["虎牙", "popu", record[1]])   
 item = "allpopu "+"huya " + str(record[1]) + " " + str(time)   
 data.append(item)   
 data.append("clear")   
   
   
 huyaPopuCounts.repartition(1).foreachRDD(outputHuyaPopuCounts)

3.分别计算斗鱼和虎牙两个平台各个直播频道（板块）的直播人数总和（RDD中value为总人数）

这里使用的map函数是mapByName，即返回的键值对格式是（板块名，数值），这样可以按照不同的板块进行后续计算。

douyuPeopleCountsForeach = douyuPeople.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)   
huyaPeopleCountsForeach = huyaPeople.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \   
 .reduceByKey(lambda a, b: a + b)

接着进行进一步处理。这里实现逻辑部分与上文类似，这里着重说明不一样的地方。

* 经过统计，发现平均来说直播间总板块数在500以下，所以这里取rdd前500条条目出来进行计算。
* 由于没有对DStream进行直接按值排序的方法，这里考虑到每次任务时间间隔较长，500条数目的冒泡排序处理绰绰有余，于是采用了最简单的冒泡排序对rdd中的条目按值从高到低排序，这里即是按板块总人数来排序。最终取出前十，记录到输出流data中。
* 在全局data列表中加入一条新的字符串，格式为"eachpeople"+"douyu " + 板块名 + " " +板块数值+ " " + 系统时间
* 其中eachpeople是一个约定的命令，方便可视化处理，具体约定参见可视化章节。
* douyu是平台名。

def outputDouyuPeopleCountsForeach(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(501)   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 if len(taken) > 0:   
 for record in taken[:10]:   
 print(["斗鱼", "people", record[0], record[1]])   
 item = "eachpeople "+"douyu " + record[0] + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
 douyuPeopleCountsForeach.repartition(1).foreachRDD(outputDouyuPeopleCountsForeach)   
   
   
 def outputHuyaPeopleCountsForeach(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(501)   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 if len(taken) > 0:   
 for record in taken[:10]:   
 print(["虎牙", "people", record[0], record[1]])   
 item = "eachpeople "+"huya " + record[0] + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
 huyaPeopleCountsForeach.repartition(1).foreachRDD(outputHuyaPeopleCountsForeach)

4.分别计算斗鱼和虎牙两个平台各个直播频道（板块）的人气总和（RDD中value为总人气）

douyuPopuCountsForeach = douyuPopu.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \  
 .reduceByKey(lambda a, b: a + b)  
huyaPopuCountsForeach = huyaPopu.map(lambda x: (mapByName(x)[0], mapByName(x)[1])) \  
 .reduceByKey(lambda a, b: a + b)

接着进行进一步处理。逻辑与上文类似，故不赘述。

值得注意的是，在最后一个函数中加入了将data流本地化的方法。

def outputDouyuPopuCountsForeach(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(501)   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 if len(taken) > 0:   
 for record in taken[:10]:   
 print(["斗鱼", "popu", record[0], record[1]])   
 item = "eachpopu "+"douyu " + record[0] + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
   
   
douyuPopuCountsForeach.repartition(1).foreachRDD(outputDouyuPopuCountsForeach)   
   
   
def outputHuyaPopuCountsForeach(time, rdd):   
 s = "Time: %s" % time   
 # print("-------------------------------------------")   
 # print("outputpopuCounts Time: %s" % time)   
 # print("-------------------------------------------")   
 taken = rdd.take(501)   
 for i in range(0,len(taken)-1):   
 for j in range(i+1,len(taken)):   
 if taken[i][1] < taken[j][1]:   
 tmptaken = taken[i]   
 taken[i] = taken[j]   
 taken[j] = tmptaken   
 if len(taken) > 0:   
   
 for record in taken[:10]:   
 print(["虎牙", "popu", record[0], record[1]])   
 item = "eachpopu "+"huya " + record[0] + " " + str(record[1]) + " " + str(time)   
 data.append(item)   
 file = open("./longOutput.txt", "w", encoding='utf-8')   
 for i in data:   
 file.write(i + "\n")   
 file.close()   
 data.clear()   
   
huyaPopuCountsForeach.repartition(1).foreachRDD(outputHuyaPopuCountsForeach)

在完成最后一个函数后，将对data输出流清空（data.clear()),以应对下一次的任务。

最后启动ssc。

# 开始监听   
ssc.start()   
ssc.awaitTermination()

输出的本地./longOutput.txt格式如下所示。

allpeople douyu 12000 2020-11-26 17:46:30   
allpeople huya 12000 2020-11-26 17:46:30   
allpopu douyu 682211321 2020-11-26 17:46:30   
allpopu huya 1115057960 2020-11-26 17:46:30   
clear   
eachpeople douyu 英雄联盟 1017 2020-11-26 17:46:30   
eachpeople douyu 颜值 719 2020-11-26 17:46:30   
eachpeople douyu 原神 695 2020-11-26 17:46:30   
eachpeople douyu 王者荣耀 646 2020-11-26 17:46:30   
eachpeople douyu 绝地求生 612 2020-11-26 17:46:30   
eachpeople douyu 一起看 433 2020-11-26 17:46:30   
eachpeople douyu DNF 364 2020-11-26 17:46:30   
eachpeople douyu 天刀手游 360 2020-11-26 17:46:30   
eachpeople douyu 户外 345 2020-11-26 17:46:30   
eachpeople douyu 和平精英 340 2020-11-26 17:46:30   
eachpeople huya 王者荣耀 1041 2020-11-26 17:46:30   
eachpeople huya 星秀 989 2020-11-26 17:46:30   
eachpeople huya 交友 895 2020-11-26 17:46:30   
eachpeople huya 英雄联盟 750 2020-11-26 17:46:30   
eachpeople huya 和平精英 443 2020-11-26 17:46:30   
eachpeople huya 天涯明月刀手游 320 2020-11-26 17:46:30   
eachpeople huya 颜值 271 2020-11-26 17:46:30   
eachpeople huya 穿越火线 251 2020-11-26 17:46:30   
eachpeople huya 魔兽世界 249 2020-11-26 17:46:30   
eachpeople huya 绝地求生 246 2020-11-26 17:46:30   
eachpopu douyu 王者荣耀 72695696 2020-11-26 17:46:30   
eachpopu douyu 英雄联盟 58228927 2020-11-26 17:46:30   
eachpopu douyu 颜值 54439487 2020-11-26 17:46:30   
eachpopu douyu 绝地求生 43790501 2020-11-26 17:46:30   
eachpopu douyu 和平精英 30004994 2020-11-26 17:46:30   
eachpopu douyu DNF 24427991 2020-11-26 17:46:30   
eachpopu douyu 天刀手游 23307220 2020-11-26 17:46:30   
eachpopu douyu 交友 23040995 2020-11-26 17:46:30   
eachpopu douyu 原神 17436098 2020-11-26 17:46:30   
eachpopu douyu CSGO 15951645 2020-11-26 17:46:30   
eachpopu huya 星秀 153367945 2020-11-26 17:46:30   
eachpopu huya 交友 129076682 2020-11-26 17:46:30   
eachpopu huya 王者荣耀 128669526 2020-11-26 17:46:30   
eachpopu huya 英雄联盟 72237791 2020-11-26 17:46:30   
eachpopu huya 和平精英 48947587 2020-11-26 17:46:30   
eachpopu huya 颜值 33110204 2020-11-26 17:46:30   
eachpopu huya 绝地求生 28234531 2020-11-26 17:46:30   
eachpopu huya 天涯明月刀手游 24328858 2020-11-26 17:46:30   
eachpopu huya 户外 23220750 2020-11-26 17:46:30   
eachpopu huya 一起看 20437713 2020-11-26 17:46:30

## 四、计算结果动态展示

可视化项目目录结构：

graph\_part   
├── long   
│ ├── bar1.js   
│ ├── bar.js   
│ ├── index.html   
│ ├── line1.js   
│ ├── line2.js   
│ ├── line.js   
│ ├── pie1.js   
│ ├── pie.js   
│ ├── render.js   
│ ├── test.sh   
│ ├── test.txt   
│ └── zingchart.min.js   
└── short   
 ├── index.html   
 ├── line1.js   
 ├── line2.js   
 ├── line.js   
 ├── render.js   
 ├── stack1.js   
 ├── stack2.js   
 ├── stack.js   
 ├── test.sh   
 ├── test.txt   
 └── zingchart.min.js

long和short分别为两种运行模式。

其中：

* index.html：主页HTML文件。
* render.js：实现渲染逻辑的JavaScript文件。
* zingchart.min.js：ZingChart提供的可视化组件库JavaScript文件。
* 其他.js文件：一个.js文件为一个图像的描述配置。
* test.sh：本地测试websocketd时使用的示例标准输出脚本文件。
* test.txt：本地测试websocketd时使用的需要传输的示例文件。

### 4.1 index.html

以运行模式long为例（long、short两运行模式的index.html结构组织基本一致）：

<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Long</title>  
 <script src="/zingchart.min.js"></script>  
 <style>  
 ...  
 </style>  
</head>  
  
<body>  
 <div>  
 <div id="myChartBar" class="chart--container--bar"></div>  
 <div id="myChartBar1" class="chart--container--bar"></div>  
 </div>  
 <div id="myChartLine" class="chart--container"></div>  
 <div>  
 <div id="myChartLine1" class="chart--container--line"></div>  
 <div id="myChartLine2" class="chart--container--line"></div>  
 </div>  
 <div id="myChartPie" class="chart--container"></div>  
 <div id="myChartPie1" class="chart--container"></div>  
  
 <script src="/bar.js"></script>  
 <script src="/bar1.js"></script>  
 <script src="/pie.js"></script>  
 <script src="/pie1.js"></script>  
 <script src="/line.js"></script>  
 <script src="/line1.js"></script>  
 <script src="/line2.js"></script>  
 <script src="/render.js"></script>  
</body>  
</html>

为了使得代码展示变得简洁，隐藏了部分，可以在项目代码中自行查看完整代码。

不难发现，<head>标签内指定了编码方式、标题、使用的ZingChart可视化组件库文件的路径、样式。<body>标签内描述了各个图像的排列方式，并包含了各个图像的描述配置.js文件以及包含了渲染逻辑的render.js文件。

### 4.2 render.js

以long模式的render.js为例：

var ws = new WebSocket("ws://192.168.0.121/ws/");  
  
ws.onopen = function(event) {  
 zingchart.render({  
 id: 'myChartBar',  
 data: chartConfigBar,  
 height: '100%',  
 width: '100%',  
 defaults: {  
 fontFamily: 'sans-serif'  
 }  
 });  
 zingchar.render...  
 ...  
};  
  
ws.onmessage = function(event) {  
 command = event.data;  
 arr = command.split(' ');  
  
 if (arr[0] == "allpeople") {  
 ...  
 } else if (arr[0] == "allpopu") {  
 ...  
 } else if (arr[0] == "clear") {  
 ...  
 } else if (arr[0] == "eachpeople") {  
 ...  
 } else if (arr[0] == "eachpopu") {  
 ...  
 }  
};

为了使得代码展示变得简洁，隐藏了部分具体实现，可以在项目代码中自行查看完整代码。

var ws = new WebSocket('...');用于创建一个WebSocket连接，文中指定的地址仅作示例，需要工具情况指定为**主机A**的地址。

ws.onopen函数在ws成功创建WebSocket连接时进行调用，这里初始化渲染了所有的图表。

ws.onmessage函数，在ws**每一次**接收到来自服务端的信息时被调用一次。这里event.data代表了传入的数据，将其按照空格分块成一个数组，再按照规约（详见后续对test.txt的讲解）对传入的信息进行处理，修改相关的图像描述配置信息，并重新进行渲染。

### 4.3 其他.js文件

以long中的line.js为例：

let chartConfigLine = {  
 type: 'line',  
 globals: {  
 fontSize: '14px'  
 },  
 title: {  
 text: '直播间总数变化对比图',  
 color: '#5D7D9A',  
  
 \_padding: '30 0 0 35',  
 fontSize: '30px'  
 },  
 subtitle: {  
 ...  
 },  
 \_legend: {  
 ...  
 },  
 legend: {  
 ...  
 },  
 plot: {  
 ...  
 },  
 plotarea: {  
 margin: '85px'  
 },  
 scaleX: {  
 ...  
 },  
 scaleY: {  
 ...  
 },  
 crosshairX: {  
 ...  
 },  
 series: [  
 {  
 text: '斗鱼直播间数',  
 // plot values  
 values: [0,0,0,0,0,0,0,0,0,0,0],  
 lineColor: '#3290be',  
 marker: {  
 backgroundColor: '#3290be'  
 }  
 },  
 ...  
 ]  
};

为了使得代码展示变得简洁，隐藏了部分具体实现，可以在项目代码中自行查看完整代码。

在**其他.js文件**中，会像上面的代码一样，指定了一个描述配置，作为渲染图像的依据。

### 4.4 test.sh

此处的test.sh是为了单独进行可视化模块的验证而写的。真正采用的脚本在../../compute\_part/echo.sh，实现逻辑类似。

#!/bin/bash  
while :  
do  
 cat ./test.txt  
 sleep 90  
done

每90秒打印一次./test.txt的内容。

echo.sh：

#!/bin/bash   
while :   
do   
 cat longOutput.txt   
 sleep 30   
 echo   
 sleep 30   
 echo   
 sleep 30   
done

这里采用分三次sleep是为了持续激活websocket防止断开连接。

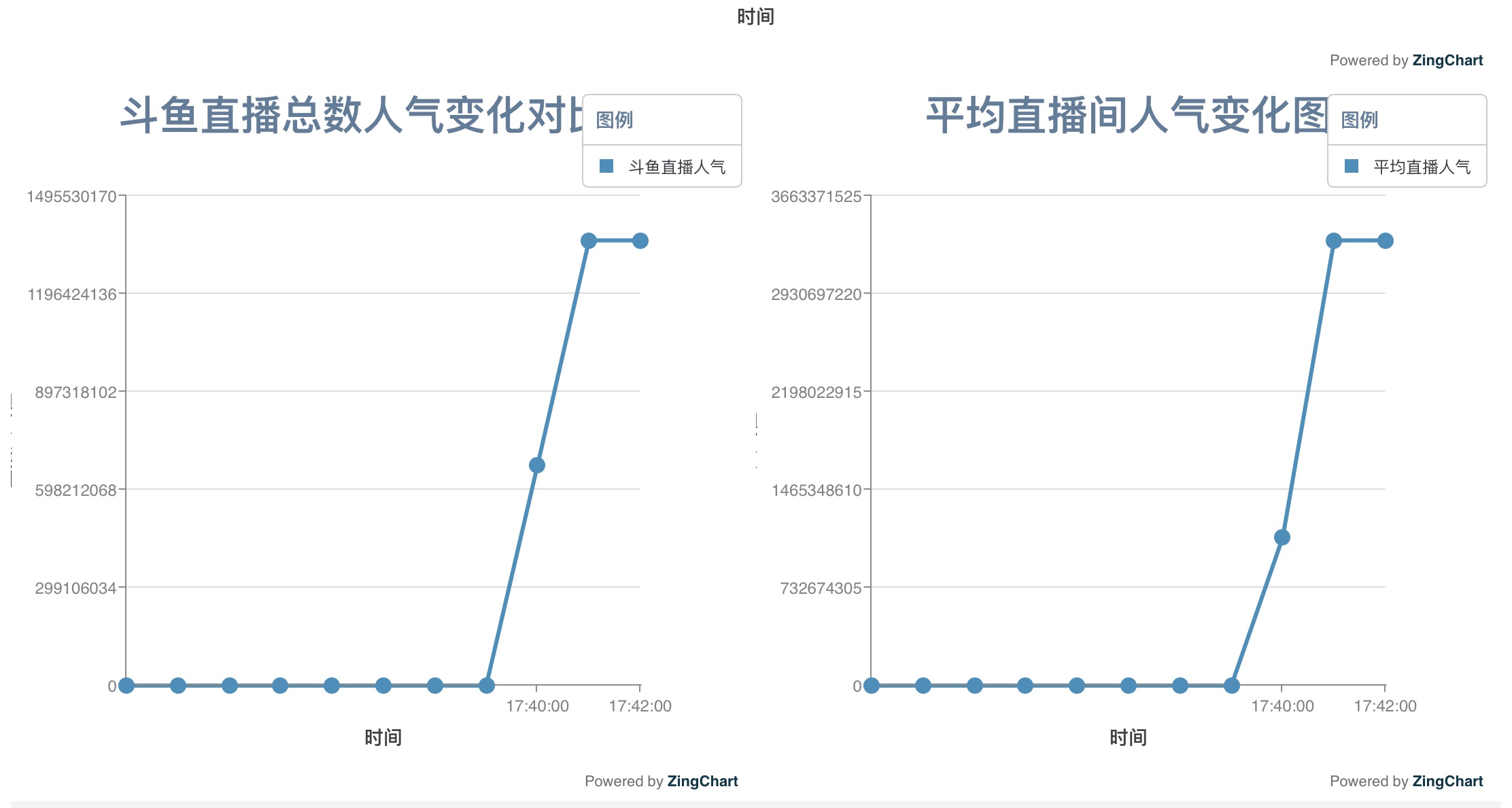
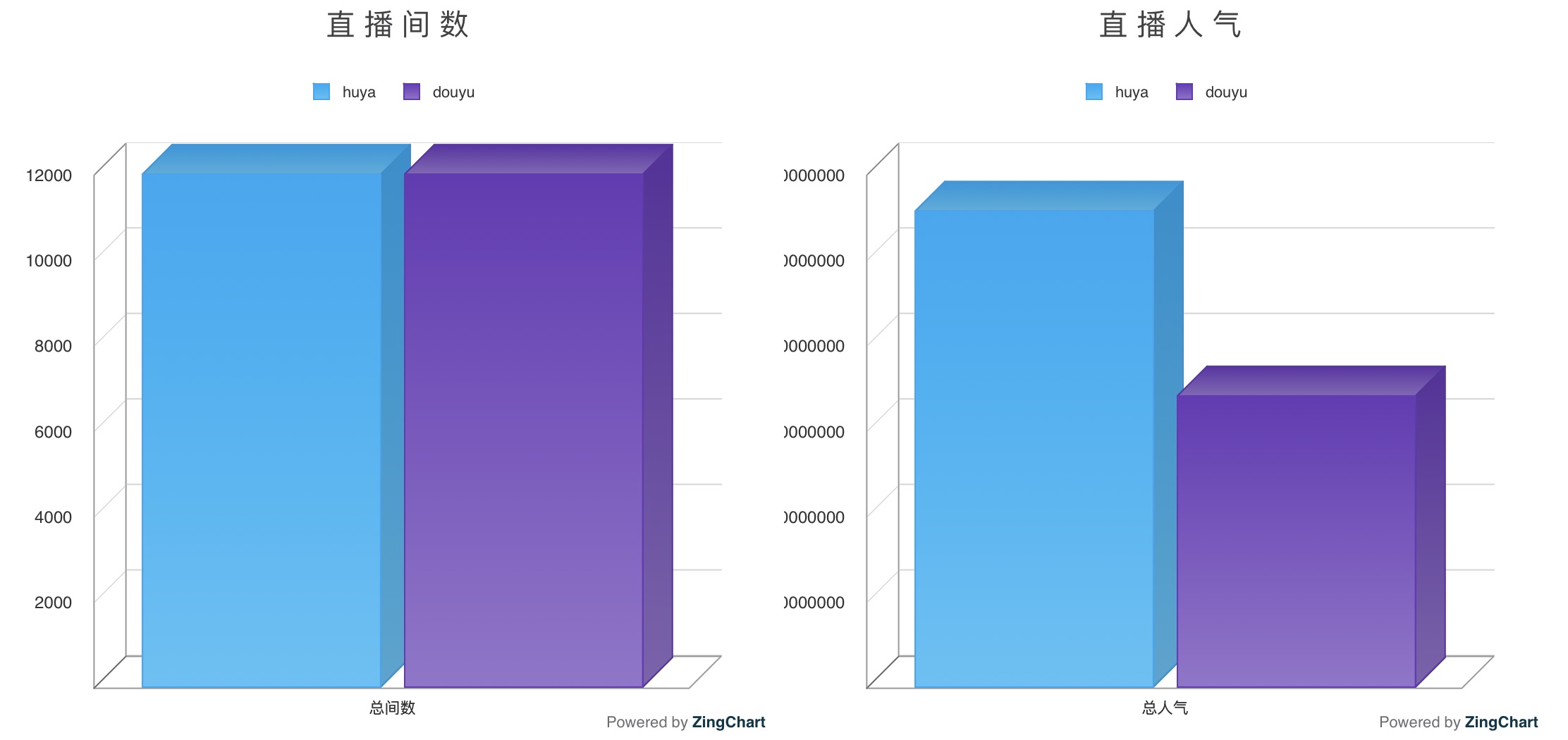
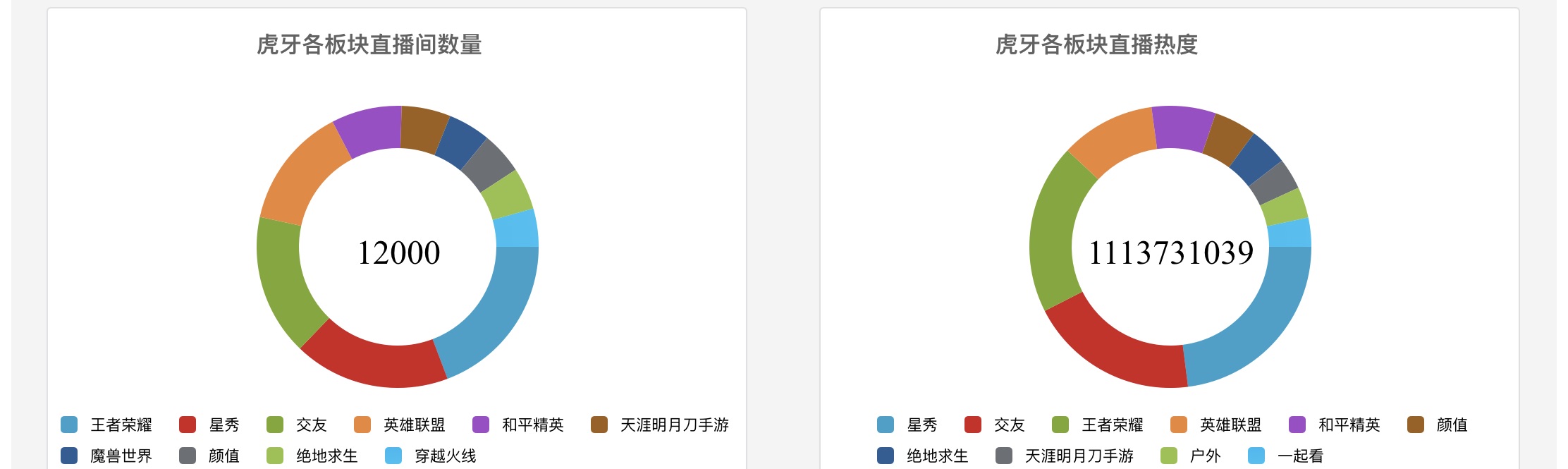
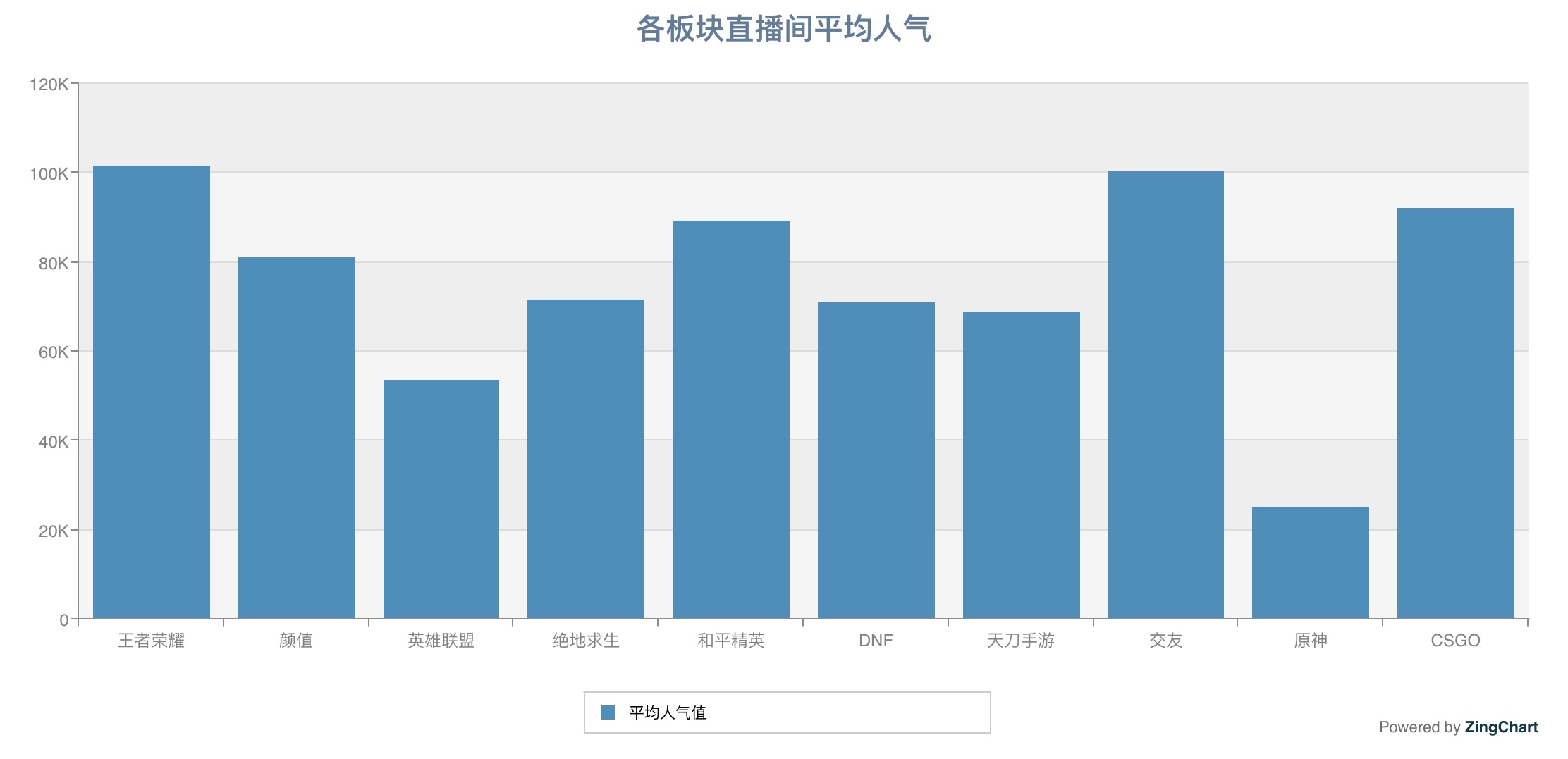
### 4.5 test.txt

此处的test.txt是为了单独进行可视化模块的验证而写的。真实采用的数据在../../compute*part/longOutput.txt以及../../compute*part/shortOutput.txt，格式与test.txt完全一致。

* long模式中的test.txt：
* allpeople douyu 21 3633 17:00:00  
  ...  
  allpopu huya 244 254 17:00:00  
    
  clear  
    
  eachpeople douyu lol 124  
  ...  
  eachpeople huya jdqs1 33  
    
  eachpopu douyu lol2 11  
  ...  
  eachpopu huya jdqs3 33
* 其中：
  + allpeople：指定总直播间数。
  + allpopu：指定总人气值。
  + clear：清除页面中折线图和饼图的数据。
  + eachpeople：指定每个平台中每个板块的直播间数。
  + eachpopu：指定每个平台中每个板块的人气值。
* short模式中的test.txt：
* allpeople people 333 1212 17:00:00  
  ...  
  allpeople people 567 1212 17:03:00  
  allpopu popu 999999 213321 17:00:00  
  allmean 3003.1 123123 17:00:00  
    
  clear  
    
  peoplechange lol15 9 121  
  ...  
  peoplechange lol11 9 121  
    
  popuchange lol15 9 121  
  ...  
  popuchange lol11 9 121  
    
  namemean lol1 8 123  
  ...  
  namemean wzry2 9 312
* 其中：
  + allpeople：指定总直播间数。
  + allpopu：指定总人气值。
  + allmean：指定平均每直播间人气值。
  + clear：清除页面中直播间数变化、直播人气变化的图像的数据。
  + peoplechange：指定某个板块相较于上一次爬取数据的直播间数量变化。
  + popuchange：指定某个板块相较于上一次爬取数据的直播人气值数量变化。
  + namemean：指定每个板块每直播间的平均人气值。

## 五、业务需求实现情况

下面放几张可视化的图表展示。受限于篇幅，不进行全部展示。



通过研究一些实时图表，我们发现了一些有趣的结论。

从斗鱼、虎牙各版块直播热度可以看出：

* 1. 斗鱼中英雄联盟板块直播间最多，但是平均人气却较低，所以英雄联盟中主播的热度可能更加参差不齐，呈两极分化。也可能是因为英雄联盟板块本身的打赏吸引力不够。
  2. 虎牙、斗鱼两平台中对应的颜值或星秀板块都具有较多的直播间数，也具有较高的平均值，颜值、星秀板块的分布较为平均。（星秀板块可能更能吸引打赏，小众主播也能吸引观众，分布较为平均）
  3. 斗鱼热度明显低于虎牙（可能拥有不同的热度计算方式）。
  4. 两平台中，都是以游戏直播、星秀、颜值等为主。其中虎牙对游戏板块的依赖度明显更小，而斗鱼热度几乎集中在游戏板块。

当然，我们主要的目标是为了解决业务需求。以下将对业务需求的六个问题进行进一步阐释。

**直播间总数目与直播间总人气的动态计算**

由于我们限定了爬取的直播间数量，所以呈现出斗鱼与虎牙的直播间总数目保持在12000不变。实际场景下斗鱼与虎牙的直播间总数在21k~26K之间。而直播间总人气这一块二者变化幅度不大，较为稳定。如果出现了较为明显的上升则说明有某大主播开播；相反如果出现了较为明显的下降则说明有某大主播停止直播。这种情况下平台在管理时应当注重对大主播开播时的资源分配（例如分配更多管理人员），以及在大主播停止直播后注重观众的分流，而不是让他们都离开（例如自动跳转直播间、推荐其他主播等）。

在可视化图表中有平台的实时直播间数目、直播间总人气的实时信息柱状图以及随时间变化的实时折线图。

**每个直播间的平均人气动态计算**

计算直播间的平均人气是从平均的角度上衡量目前平台的直播收看情况。可以根据该数值估测平台的负载情况从而制定一些负载等级，每个等级可以合理分配人力资源进行管理等等。

在可视化图表中有实时直播间平均人气柱状图以及随时间变化的实时折线图。

**不同板块直播间数目与人气增减的动态计算**

针对不同板块的直播间数目与人气相较之前的增减情况可以一目了然地看到各个板块的实时变化情况，例如英雄联盟板块的直播间相较一分半前少了3个，，人气少了251w，但王者荣耀板块的直播间相较一分半前多了5个，人气多了1231w等等。同时结合直播间数目与人气增减情况，我们可以得到以下四种模型：

1).同一板块的直播间数相较上一时段下降，同时直播人气相较上一时间段下降。这种情况说明可能有某些大主播停止了直播，或者小主播停止了直播。

2). 同一板块的直播间数相较上一时段下降，同时直播人气相较上一时间段上升。这种情况说明可能有某些小主播停止了直播，但大主播还在持续制造人气，弥补了小主播离开的人气。

3). 同一板块的直播间数相较上一时段上升，同时直播人气相较上一时间段下降。这种情况说明可能有某些小主播开始了直播，但大主播可能停止了直播，小主播开播制造的人气弥补不了大主播离开的人气。

4). 同一板块的直播间数相较上一时段上升，同时直播人气相较上一时间段上升。这种情况说明可能有大主播开启了直播，或者小主播开启了直播。

根据上述四种模型，平台在管理时可以更合理更灵活地分配平台资源。

在可视化图表中有不同板块的直播间数目与人气实时增减的柱状图。

**不同板块直播间平均人气的动态计算**

计算不同板块直播间的平均人气是从平均的角度上衡量目前平台各板块的直播收看情况。可以根据该数值更灵活地估测平台的负载情况从而制定一些负载等级，每个等级可以合理分配人力资源进行管理等等。需要注意的是，拥有该项数值的板块说明该板块在直播人数与直播人气两方面都排入了平台前十，因此值得投放更多精力在上面。

在可视化图表中有实时直播间平均人气柱状图。

**不同平台间直播间总数目与直播间总人气的变化计算**

由于我们限定了爬取的直播间数量，所以呈现出斗鱼与虎牙的直播间总数目保持在12000不变。实际场景下斗鱼与虎牙的直播间总数在21k~26K之间。而直播间总人气这一块二者变化幅度不大，较为稳定，虎牙的人气整体上大于斗鱼的人气，推测是人气使用算法不同。如果不限制爬取的直播间数量，可以看出两平台的直播人数的动态变化，从而可以做出一些更进一步的分析。

在可视化图表中有斗鱼虎牙的实时直播间数目、直播间总人气的实时信息柱状图以及随时间变化的实时折线图。

**不同平台间不同板块直播间数目与人气的变化计算**

通过比较斗鱼与虎牙前十板块的直播间数目与人气的组成情况，我们可以进一步分析出两个平台不同的倾向与特点。例如之前分析提到的“两平台中，都是以游戏直播、星秀、颜值等为主。其中虎牙对游戏板块的依赖度明显更小，而斗鱼热度几乎集中在游戏板块”等特点。

在可视化图表中有斗鱼虎牙的前十板块直播间数目、直播间总人气的实时信息饼图。

## 六、困难&解决

**搭建环节**

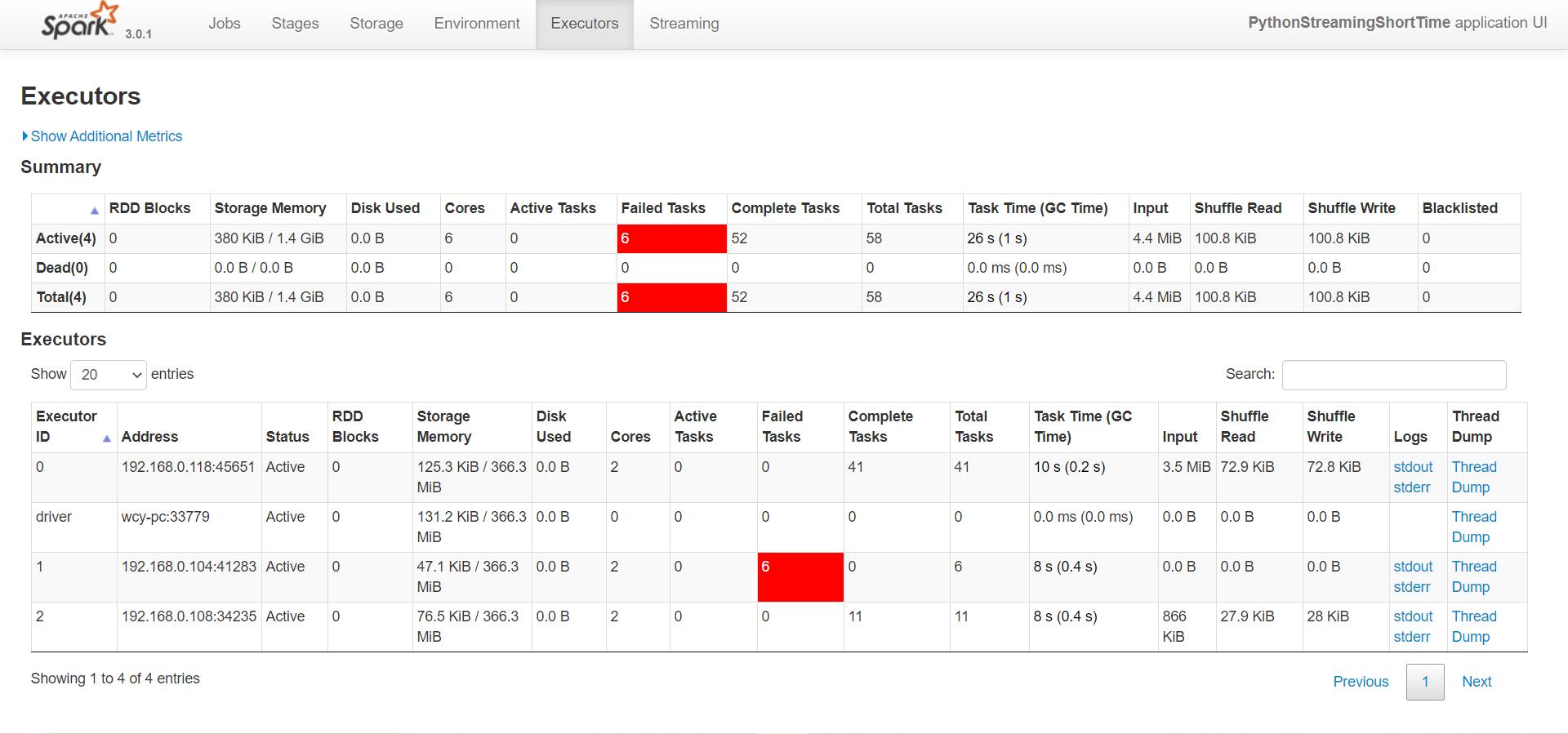
1. 开始使用windows+mac系统搭建集群，hadoop集群搭建成功但spark集群搭建失败。

解决方案：spark集群全部更换为linux系统。Hadoop集群中保留了mac系统，推测windows系统也可以得到保留，但建议将namenode部署在linux系统中。

1. Linux上免密登陆ssh配置过于麻烦，可不可以不配置？

解决方案：可以。实际上，免密登陆ssh是为了方便在hadoop与spark上使用./start-all.sh以及./stop-all.sh两个脚本。如果不配置免密登陆ssh，则分别自行启动namenode、datanode、master、worker即可。

1. 提交py任务到集群后控制台报错提示python版本不统一。



解决方案：在对应的机器上将python版本统一更新至3.8，且选择自动适配高版本。前文有提到如何自动适配高版本，此处不赘述。

**爬取与数据预处理阶段**

1. 直播爬取效率过低

解决方案：查看爬取过程后发现，爬取时尝试爬取了大量没有人气的“僵尸直播间”。于是将爬取页数限定在前100页，保证了前100页的直播间都是高质量直播间，在提高了爬取质量的同时将爬取效率提高至两倍以上。

**流准备阶段**

1. 一开始没有想到很好的推流办法，只能一直手动在控制台不停敲指令持续上传文件。

解决方案：通过使用shell脚本打上时间戳自动持续上传，提高了效率与精准度。

1. Shell脚本推流花费的时间与预想花费的时间有出入。

解决方案：仔细观察后发现，上传数据到hdfs上需要花费一定时间，统计该时间后对shell脚本进行进一步调整即可。

**流计算阶段**

1. 开始使用Scala+maven进行开发，但环境部署麻烦、新学习scala语法也带来困难。

解决方案：更换使用python开发，python有自带的pyspark库，支持大部分spark的api。对应的，不需要将程序打成jar包，直接提交py文件到集群进行运行即可。