2018 年 12 月

哈爾濱工業大學

大数据计算基础

题　　目：实时网约车系统设计

专　　业：计算机科学与技术大数据方向

学　　号：1160300527

姓　　名：肖松

课程类别：必修

1. 问题描述

1.题号：41
2.题目：实时网约车系统设计
3.问题描述：

a) 应用背景：网约车是网络预约出租汽车的简称。在构建多样化服务体系方面，出租车将分为巡游出租汽车和网络预约出租汽车。通过互联网的即时通信模式，可以同时缓解空车司机无法寻找到客户、寻车用户无法找到出租车的问题。本题目目标是，实现并维护一个简易的分布式的系统，让每个寻车用户与租车司机实现实时配对。

b) 设计部分设计一个分布式的存储维护方法，分别保存并处理实时的寻车用户和租车司机的信息，提出一个有效的配对方法使双方的等待时间不至于过长、接车距离不至于过远。

c) 实验部分实现并维护一个简易的分布式的管理系统，在实时的数据流上，让每个寻车用户与租车司机实现实时配对。使用上述有效的配对方法降低等待时间和接车距离。

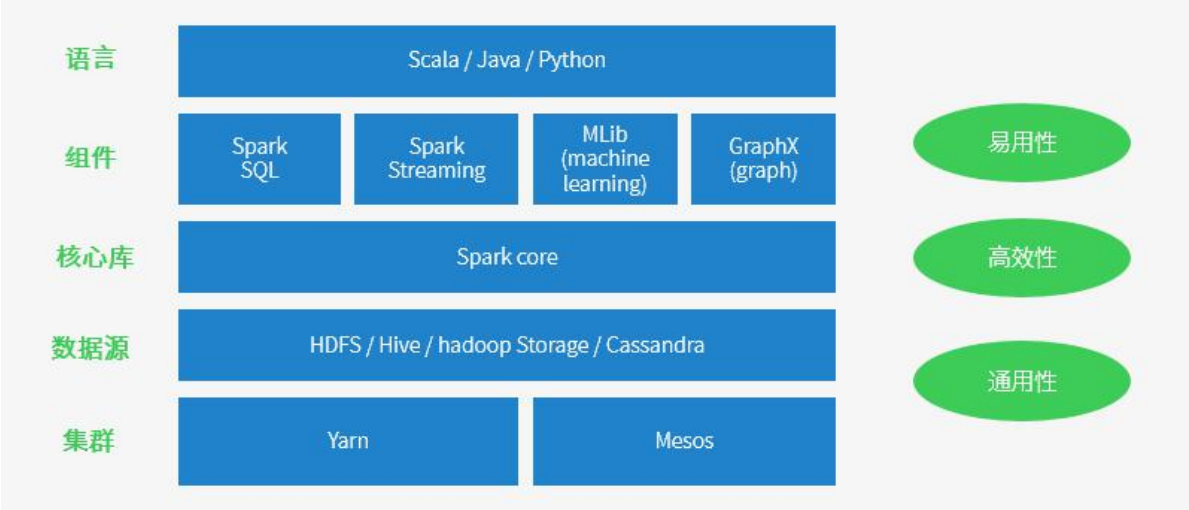
技术难点：
1. 司机与乘客信息以流的形式到达，需要从中分别获取司机以及乘客信息并进行处理。
2. 如果把所有的司机以及乘客放在一起处理匹配，数据规模较大，恐怕难以在有限时间内完成，因此可以划分为不同区域进行匹配。
3. 不能够保证每一轮匹配中所有乘客与司机都能够成功配对，对于匹配失败的司机及乘客需要进行状态的保存，当进行下一轮匹配时，之前匹配失败的乘客应该优先进行匹配。
4. 如果有多个满足要求的司机时，应该优先选择与接车地点距离最短的司机进行配对。

2. 基于的系统/算法

系统：在本次大作业中，主要使用的是 spark 的扩展 spark-streaming。Spark-streaming 将实时流切分为若干批（每批 X 秒），将每批数据视为 RDD 进行处理，从而实现对流的处理。虽然无法实现毫秒级的处理速度，但整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。由于在网约车系统中数据是以流形式源源不断到来的，需要对流数据进行处理，在处理过程中，还需要对中间状态以及结果进行保存。同时，由于网约车任务并不需要毫秒级的处理速度，因此，采用了 spark-streaming 作为本次作业的系统。

利用 spark-streaming，监听本地特定端口，并每隔一定时间处理一次接受到的数据。在每轮处理时，将接受到的数据根据类型划分为司机与乘客，并与上一轮保存的状态信息进行合并后一起进行匹配。对于匹配成功的数据，将匹配成功信息写入记录文件中，对于匹配失败的数据，则保存到状态中等待下一轮的匹配。

Spark 生态如下：



Spark 架构采用了分布式计算中的 Master-Slave 模型。Master 是对应集群中的含有 Master 进程的节点，Slave 是集群中含有 Worker 进程的节点。

Master 作为整个集群的控制器，负责整个集群的正常运行；

Worker 相当于计算节点，接收主节点命令与进行状态汇报；

Executor 负责任务的执行；

Client 作为用户的客户端负责提交应用；

Driver 负责控制一个应用的执行。

Spark 集群部署后，需要在主节点和从节点分别启动 Master 进程和 Worker 进程，对整个集群进行控制。在一个 Spark 应用的执行过程中，Driver 和 Worker 是两个重要角色。Driver 程序是应用逻辑执行的起点，负责作业的调度，即 Task 任务的分发，而多个 Worker 用来管理计算节点和创建 Executor 并行处理任务。在执行阶段，Driver 会将 Task 和 Task 所依赖的 file 和 jar 序列化后传递给对应的 Worker 机器，同时 Executor 对相应数据分区的任务进行处理。

3. 系统设计/算法设计

系统设计：

1. 数据生成：生成测试数据，并完成发送（用于测试时使用）

具体实现：随机生成司机/乘客的经纬度（对于司机，经纬度就是其所处位置，对于乘客，经纬度就是其预约地点的位置），并给每个乘客与司机设置一个 id 号，取 System.currentTime() 作为乘客/司机发送信息的时间，并为乘客随机生成到达的时间（在司机中该字段设为-1）。然后通过 socket 向服务器发送数据

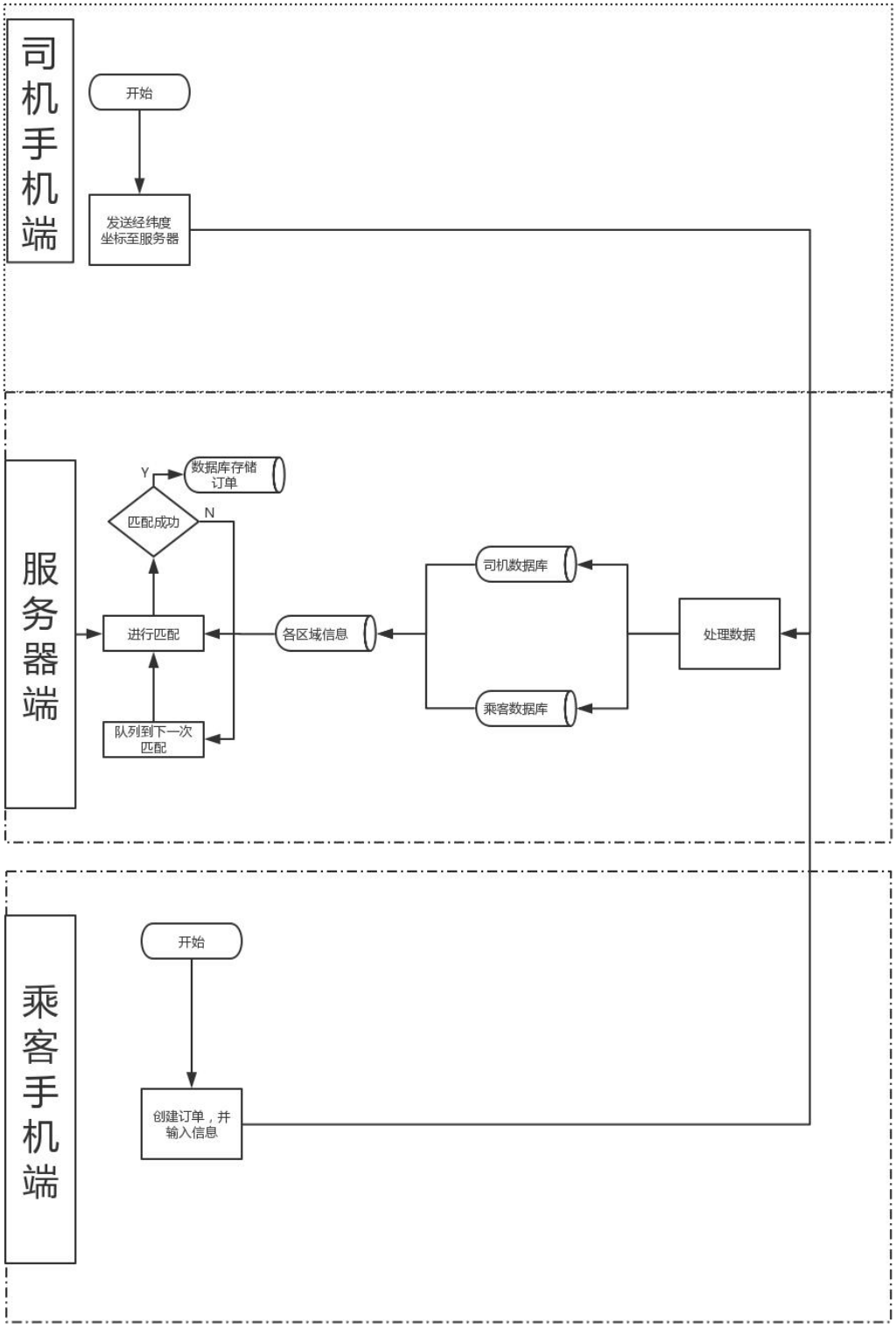
2. 接收数据：在服务器端运行，接收监听的端口传来的数据。由于使用的是 spark-streaminng，因此每间隔指定时间完成一次处理，并将接收到的数据转化为 Dstream 保存。

具体实现：创建 SparkConf 对象，并通过创建的 SparkConf 对象创建 JavaStreamingContext 对象并设置间隔时间，利用 JavaStreamingContext 从指定端口号与主机接收 socket 消息

3. 数据处理：由于接收的数据为字符串，因此，要根据字符串中的信息创建各个对象，并且将司机与乘客信息进行分别处理，然后将其分别放到不同区域中，并与状态信息进行合并
具体实现：通过 Map 方法，将每个字符串对应信息转换为乘客/司机，然后通过 fileter 分别筛选出乘客以及司机，然后再次通过 Map 与 Reduce 将相同区域的司机/乘客分别放置进行处理。

4. 匹配模块：将相同区域内的司机/乘客进行匹配，优先为预约时间最早的乘客匹配最近的司机，并记录匹配成功的司机/乘客信息，将匹配失败的乘客/司机信息进行状态保存
具体实现：通过 join 实现相同区域内的司机与乘客的连接，然后在每个区域内对乘客以预约时间进行排序，最后选择距离最近的司机，对匹配成功的司机进行标记（不再参与后续匹配），并在匹配结束后根据标记筛选匹配失败的进行保存，将匹配成功的信息写入文件中

5. 信息保存：系统会将每次接收到的司机/乘客信息进行保存，写入 json 文件中


司机手机端
开始
发送经纬度坐标至服务器
服务器端
Y
数据库存储订单
匹配成功
N
进行匹配
队列到下一次匹配
各区域信息
司机数据库
乘客数据库
处理数据
乘客手机端
开始
创建订单，并输入信息

设计的算法：

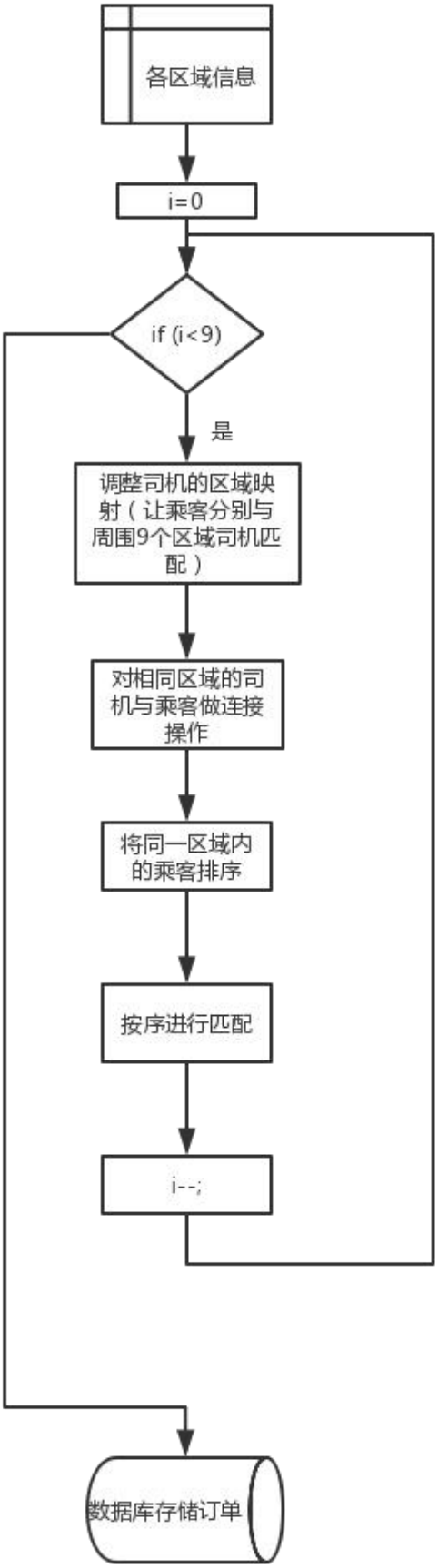
（1）数据生成：给定大小 n 与生成数据类型，随机生成一定范围内经纬度坐标的乘客或司机信息，并将生成的数据通过网络发送给服务器进行处理。在网约车应用中，数据是以网络消息的形式到达服务器的，同时，乘客与司机的坐标位置在一定范围内应该是随机的。因此，我们可以通过生成随机坐标模拟乘客与司机对应用进行测试

（2）匹配算法：在进行匹配时，我们需要考虑乘客与司机的距离限制，使接车距离不会太远。因此我们可以将乘客与司机按照经纬度划分为不同区域，对于每个区域内的乘客，只在他的在邻近各区域内寻找距离最近的乘客。如下图所示：对于区域(x,x)内的乘客，我们只需要在(x-1,x-1),(x-1,x),(x-1,x+1),(x,x-1),(x,x),(x,x+1),(x+1,x-1),(x+1,x+1),(x+1,x+1)这九块区域内考虑即可，在这九块区域外的司机，由于距离过远不予考虑。

(x-1,x-1)	(x-1,x)	(x-1,x+1)
(x,x-1)	(x,x)	(x,x+1)
(x+1,x-1)	(x+1,x)	(x+1,x+1)

图 2.1 示意图

在实际进行匹配时，将乘客按照预约的时间进行排序（时间早的优先选择），然后在司机中依次查找距离最近的司机完成匹配


各区域信息
i=0
if (i<9)
是
调整司机的区域映射（让乘客分别与周围9个区域司机匹配）
对相同区域的司机与乘客做连接操作
将同一区域内的乘客排序
按序进行匹配
i--;
数据库存储订单

3. 不同区域的映射：当我们要将(x,y)的司机信息分别映射到(x-1,y-1),(x-1,y),(x-1,y+1),(x,y-1),(x,y),(x,y+1),(x+1,y-1),(x+1,y+1),(x+1,y+1)等 9 个区域时，若我们按照 1 经度, 1 维度区域大小进行划分, 则我们可以为所有区域设置一个唯一的一个 ID，即 id=360*x+y，这样，通过对 id 进行换算就可以映射到其他区域：与 id 的差值分别为-366,-365,-364,-1,0,1,364,365,366。

4. 实验流程

实验流程：

（1）系统搭建

1. 安装 ubuntu 虚拟机
2. 在 ubuntu 环境下安装 java 环境
3. 安装 hadoop,spark
4. 安装 maven
5. 安装 IDEA，并完成 maven 相关配置
6. 创建 maven 模板，编辑 pom.xml，声明各依赖
7. 编写代码
8. 用 mvn package 指令对项目进行打包
9. 在 spark 的 bin 目录下执行：./spark-submit --class=主类名 jar 包路径 完成提交并运行

（2）查看 spark 官方文档了解 API 使用

（3）分析难点：匹配阶段。因此，为了优化匹配阶段，通过划分区域减少每次匹配的数据量

输入数据：

乘客信息：

arriveTime	id	latitude	longitude	matched	time	type
1546899150781	1	33.09187499275554	100.9510183057309	false	1546850302660	passenger
1546855491913	2	34.88165450185928	102.66543862828436	false	1546850302662	passenger
1546905804074	3	33.582831440055216	101.1943031346864	false	1546850302662	passenger
1546882845945	4	30.920659272077774	102.75873758804828	false	1546850302662	passenger
1546862692348	5	39.65176638003541	105.00784780637949	false	1546850302662	passenger
1546853177005	6	31.334189042411836	102.91380198334858	false	1546850302663	passenger
1546885738237	7	37.005107021624035	104.85102727244025	false	1546850302663	passenger
1546853402443	8	35.13354668249059	103.02173513863804	false	1546850302674	passenger
1546888005476	9	37.36525544717565	106.36473820543243	false	1546850302674	passenger
1546882044835	10	33.8697446334198	109.25351128003928	false	1546850302674	passenger
1546898535511	11	36.2785073669684	102.78579023709455	false	1546850302674	passenger
1546895547135	12	36.97530770043057	108.86280566176698	false	1546850302675	passenger

arriveTime:预约的时间

Id:乘客的标号，唯一

Latitude:预约地点的纬度

Longitude:预约地点的精度

Matched:是否已经匹配成功

Time:乘客发出请求的时间
Type:类型（分为乘客/司机）

司机信息：

arriveTime	id	latitude	longitude	matched	time	type
-1	1	37.54973776913579	100.55377921108695	false	1546850194093	driver
-1	2	33.27150562681767	100.91409611309375	false	1546850194095	driver
-1	3	36.18677197881474	101.68806910916062	false	1546850194095	driver
-1	4	36.08168829718687	104.85629301487154	false	1546850194095	driver
-1	5	37.29239422469381	102.52669074280823	false	1546850194095	driver
-1	6	37.18299122509359	109.84292775171875	false	1546850194096	driver
-1	7	32.18476036484022	102.43752146249555	false	1546850194096	driver
-1	8	31.320886006871174	108.0375065999486	false	1546850194106	driver
-1	9	35.845513462819625	105.37973031729884	false	1546850194106	driver
-1	10	36.671966125873055	101.26380722740937	false	1546850194106	driver
-1	11	31.709018380530768	104.15440292871723	false	1546850194106	driver
-1	12	32.52915046149501	106.18817128156874	false	1546850194107	driver
-1	13	32.327840846669105	103.34780162402267	false	1546850194107	driver
-1	14	34.67358863679096	109.69605039760197	false	1546850194107	driver

arriveTime:由于司机没有预约时间，因此设为-1
Id:司机的编号
Latitude:司机最后更新地址的维度
Longitude:司机最后更新地址的经度
Matched:是否成功匹配
Time:司机最后更新地址的时间
Type:类型（司机/乘客）

输出数据

```
(Area:125980,[[Passenger:2006, driver:1979, the time of arrive:1547293375373, cost time:12]
])
(Area:133516,[[Passenger:1731, driver:1015, the time of arrive:1547314447035, cost time:36]
])
(Area:127768,[[Passenger:572, driver:1498, the time of arrive:1547327326478, cost time:47]
])
(Area:141840,[[Passenger:281, driver:2992, the time of arrive:1547306700642, cost time:47]
, [Passenger:1067, driver:6, the time of arrive:1547319632678, cost time:36]
])
(Area:126704,[[Passenger:520, driver:886, the time of arrive:1547295897646, cost time:47]
])
(Area:137476,[[Passenger:1619, driver:2535, the time of arrive:1547317284978, cost time:36]
])
```

Area 为匹配成功的乘客所属区域编号，Passenger 表示配对的乘客编号，driver 表示配对的司机编号。The time of arrive 表示预约的到达时间。Cost time 表示从乘客发起请求到匹配成功所花的时间（单位为秒）

5. 实验

测试你设计实现的系统/算法的效果

b. 数据集：数据集的来源（是自己生成的还是下载的，生成程序是如何编写的，下载的数据是选取了一部分还是全部使用了）、数据文件或数据库占用磁盘空间大小、包含的记录个数（行数）、包含的属性及其意义
c. 测试算法/系统的效果，如误差大小，准确率等随参数的变化情况，用图表展示并作详细描述
d. 测试算法/系统的性能，如运行时间，中间结果大小，数据在不同进程间传输代价等随输入规模变化的情况，用图表展示并作详细描述

1.测试环境：
系统：Ubuntu 18.04.1 LTS
CPU:Intel® Core™ i7-6500U CPU @ 2.50GHz × 2
内存：3.9 GiB
磁盘：52.8GB
网络：本地环回测试

2. 数据集：由程序实时生成数据并发射

3. 测试结果：

```
(Area:140768,[[Passenger:989, driver:1887, the time of arrive:1547328033292, cost time:33]
])
(Area:135352,[[Passenger:2341, driver:348, the time of arrive:1547309477866, cost time:25]
])
(Area:128124,[[Passenger:4781, driver:1737, the time of arrive:1547306158299, cost time:21]
])
(Area:142176,[[Passenger:4678, driver:2611, the time of arrive:1547300172094, cost time:21]
])
(Area:129956,[[Passenger:2185, driver:3658, the time of arrive:1547339285947, cost time:25]
])
(Area:123440,[[Passenger:1572, driver:1365, the time of arrive:1547290965268, cost time:25]
])
(Area:111608,[[Passenger:3134, driver:4393, the time of arrive:1547330680963, cost time:21]
])
```

每轮数据数 / 分区数	不分区	1 纬度*1 经度	0.1 纬度*0.1 经度
100	16s	12s	10s
1000	25s	19s	13s
10000	35s	20s	16s

6. 结论

通过地理位置将数据划分在不同区域内，将相邻区域的乘客与司机进行匹配是可行的。但是分区也不应该太小，如果太小将导致匹配成功的几率变得很小，使得所需时间上升

7.实验中的踩坑

1.maven 配置中 spark-streaming,spark-core 版本需要一样,否则不兼容

2.JavaContext 不能够序列化，若要在 foreachRDD()中使用 JavaContext 时，可以用 RDD.context()方法获取 context。

3.在 rdd 的 map 之类的操作时引用了的变量必须实现 Serializable，否则会抛出 task not serializable 异常

4.reduceByKey()方法调用后，返回的 Dstream 的类型不能够发生改变：比如对 reg 的类型为 JavaPairDStream<String, Integer>，则 reg 调用 reduceByKey()方法后，也只能够返回 JavaPairDStream<String, Integer>类型变量，而不能够返回 JavaPairDStream<String, List<Integer>>，也就是 reduce 只能够对 value 进行运算等到与 value 类型相同的结果。

5.自定义的类的 get 方法一定要规范命名，否则在使用 sql 时会无法识别

6.在使用 mapWithState 时，必须先判断状态是否存在，若不存在，需要初始化状态

7.在调试时，可以通过修改配置文件，从而使得忽略 INFO 提示，防止在调试时造成困扰

8.修改环境变量时，应该直接修改/etc/profile 文件，而不是使用 export 命令添加，使用 export 命令只是临时有效，关闭命令行后失效。如果想要修改的文件立即生效，可以执行：source /etc/profile

8.项目运行过程

1.运行 mvn package 对项目进行打包（在 IDEA 中可以直接运行 maven 生成，我已经打包好了 jar 包，位于项目的 target 目录下，可直接进行以下步骤）

2.运行项目中的_generate_info，用来发送数据

3.在 spark 的 bin 目录下执行：

./spark-submit --class=_receive_info jar 包路径，可以将 jar 包提交到 spark 进行运行

4. 具体运行结果：匹配结果与司机乘客信息可以在 bin 目录下查看，文件夹名分别为：

   match

   driver-info

   passenger-info