**数据库课程设计报告**

**网约车运营管理系统**

**班级：工业软件菁英班**

**姓名：李彦浩**

**学号：202100300063**

目录

[一、系统概述 2](#_Toc76632030)

[二、需求分析 3](#_Toc76632031)

[2.1系统功能分析 3](#_Toc76632032)

[2.2系统数据分析 5](#_Toc76632033)

[2.3系统非功能分析（可选） 7](#_Toc76632034)

[三、系统设计 8](#_Toc76632035)

[3.1应用程序设计 8](#_Toc76632036)

[3.2数据库设计 8](#_Toc76632037)

[3.2.1概念设计 19](#_Toc76632038)

[3.2.2逻辑设计 19](#_Toc76632039)

[3.2.3物理设计 20](#_Toc76632040)

[四、系统实现 22](#_Toc76632041)

[4.1关键技术实现（可选） 22](#_Toc76632042)

[4.2功能实现 37](#_Toc76632043)

[五、系统测试 68](#_Toc76632044)

[六、总结 101](#_Toc76632045)

# 一、系统概述

本项目系主要针对安卓app市场的网约车运营管理系统。项目作者经过试用当前市面上主流的闭源网约车软件，例如：滴滴、高德（他们不只做地图软件，还内嵌了网约车系统）、飞滴等，通过分析、思考，根据学院官方给出的需求概要，设计了并尝试复现了网约车应用中较为核心的业务与功能。

系统主要分为两部分：第一部分负责网约车系统中的核心业务（功能），包含以下业务：

1. 用户选择出发地、目的地、订单类型、特殊订单信息，发起约车请求；
2. 司机开始听单；
3. 系统通过聚类算法实现智能化派单，并发送订单信息给司机客户端询问司机是否接单；
4. 司机确认是否接单；
5. 司机开始导航，主要涉及路线规划接口；
6. 司机接到乘客，进行手机尾号验证；
7. 司机送客至目的地处提交订单，系统提醒对应乘客完成评价；
8. 乘客评价订单。

第二部分负责非核心业务（功能），包括以下业务：

1. 用户身份权限鉴定（鉴权模块）；
2. 用户获取验证码；
3. 用户注册；
4. 用户登陆；
5. 司机注册驾照；
6. 司机注册车辆；
7. 显示订单列表；
8. 删除(隐藏)某一订单；
9. 显示订单评价等。

系统用户包括：司机与乘客，并没有实现管理员端。主要原因是工期短，人力少。

# 二、需求分析

## 2.1系统功能分析

系统主要分为两部分：

第一部分负责网约车系统中的核心业务（功能），包含以下业务：

1.用户选择出发地、目的地、订单类型、特殊订单信息，发起约车请求；

2.司机开始听单；

1. 系统通过聚类算法实现智能化派单，并发送订单信息给司机客户端询问司机是否接单；
2. 司机确认是否接单；
3. 司机开始导航，主要涉及路线规划接口；
4. 司机接到乘客，进行手机尾号验证；
5. 司机送客至目的地处提交订单，系统提醒对应乘客完成评价；
6. 乘客评价订单。

第二部分负责非核心业务（功能），包括以下业务：

1.用户身份权限鉴定（鉴权模块）；

2.用户获取验证码；

3.用户注册；

4.用户登陆；

5.司机注册驾照；

6.司机注册车辆；

7.显示订单列表；

8.删除(隐藏)某一订单；

1. 显示订单评价等。

系统包括两类用户：乘客与司机。鉴权模块由两部分组成。一个是鉴权基模块，负责校验是否为本系统的用户，采用redis实现token进行用户鉴权。

另一部分是权限子模块，针对不同的业务接口做不同的数据校验。举个简单的例子，如果司机提交订单时的位置距离乘客要求的目的地太远（超过一个指定阈值），那么他将不具有提交订单的权限。

## 2.2系统数据分析

系统需要的数据可能包括以下内容：

1. 用户登陆后获取的身份令牌token；
2. 乘客用户发起约车请求后将约车请求部分信息临时存储到Redis消息队列中；
3. 乘客用户发起约车请求后将约车请求详细信息(包括特殊约车请求，例如预约、代叫)持久化到MySQL中(这是为了做数据同步校验的)；
4. 司机用户接受订单后将订单消息持久化到数据库；
5. 乘客对订单的评价也需要持久化到数据库中（回显业务可能使用）。

数据之间联系方面：

1. Redis消息队列中的订单概要信息应该与MySQL中的订单详细信息保持一致，也就是做到数据同步。
2. Redis消息队列中的消息确认以及异常消息处理应该与派发订单模块中的所有派发结果相一致。也就是说，不可能有消息经过确认但是订单没有被派发，也不可能有订单得到派发但对应的消息还没有得到确认。
3. Redis中的token应该与客户端本地缓存的token保持一致，否则用户无法通过鉴权模块。
4. MySQL中订单与约车请求表存在单射映射。也就是说，一个订单至多对应一个请求，不可能存在两个不同的请求（主键id不同）但对应的订单相同。且可能有的请求没有转化为订单并得到派发，所以有些请求没有对应订单，这也是为什么是单射映射而不是双射映射的关系。
5. 与4相似的是，一个司机至多拥有一辆车，至多注册一个驾照。如果一个司机没有注册过驾照，那么他不能注册车辆。如果他注册的车辆所需的驾照类型和他注册的驾照类型不符，那么他不能注册该车辆。如果一个司机没有注册车辆，那么他无法开始听单。
6. 用户总表与乘客、司机用户表存在泛化/特化关系。用高级语言的概念来说，就是用户是乘客、司机的基类。乘客、司机是用户的子类。那么用户总表与乘客、司机表的并集存在双射映射关系。
7. 与6相似的是，约车请求总表与预约请求、代叫请求表也存在泛化/特化的关系。约车请求总表与预约请求、代叫请求的并集
8. MySQL中存在联系集：司机与车辆从属关系联系集。需要保证该表与司机实体表、车辆实体表保持一致。

## 2.3系统非功能分析（可选）

设计了一个基于jmeter的压力测试，经过测试，本系统可以抗住1000qps(1000个乘客在短时间内发出1000条约车请求，1000个司机听单，实现对1000个司机的派单)，异常率控制到0.1以下。抛去日志i/o的时间，可以在10s左右完成1000个订单的分配。吞吐量可达50次/秒左右。具体测试结果复现方式见于测试文档与录屏中。

安全方面，在每个接口中做了数据合法性的校验，尤其是对于伪造请求的校验，比较严格。比如校验开始听单的请求是否真的来自一位司机。以及这位司机是否真的处于还未听单的状态。

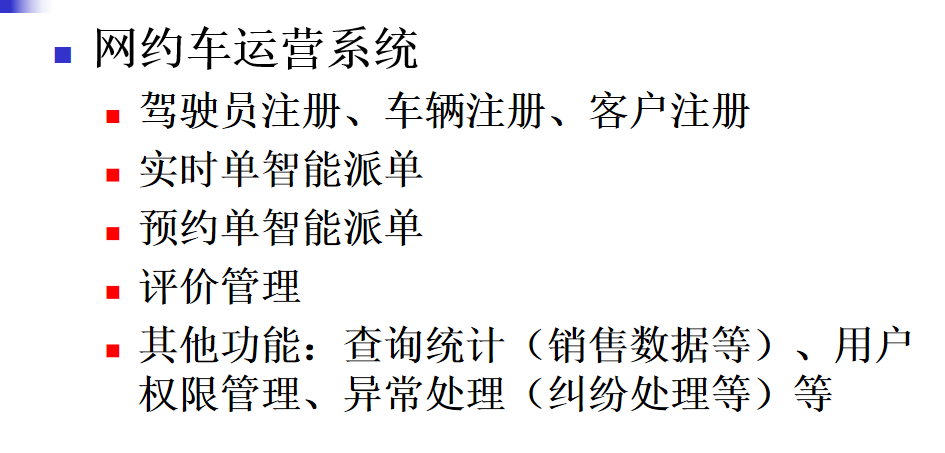
另外，还基于Redis的String数据结构实现了类似于Spring Security的用户权限校验模块(鉴权子系统)。对于没有身份令牌的用户，将拦截并打回其请求。(error code401)

可用性方面，采用uniapp前端框架，并参照uni的部分开源模板，设计实现了易用性较强的客户端界面，尽可能地方便用户的使用。

# 三、系统设计

## 3.1应用程序设计

下图为官方ppt中对于网约车运营管理系统需求的概要。



如果把这份概要作为需求文档则过于简陋，而且还有很多必要的业务流程不包含在这个概要中。因此需要重新梳理一份需求文档。

经过思考，将所有业务分为两个部分：分别为核心业务与非核心业务。

核心业务其实是一整套流程，包括以下步骤。

1. 乘客用户发起约车请求
2. 司机用户点击开始听单（开始听单指：向服务端表达自己准备好接单的意愿）
3. 服务端进行约车请求的调度、派发
4. 司机用户接到订单
   1. 司机用户拒绝订单，该约车请求重新排队，该司机不会再接到该约车请求，对于当前的约车请求而言 end
   2. 司机用户接受订单，继续
5. 司机用户向服务端汇报其接到乘客
6. 司机用户在到达目的地（附近）后提交订单
7. 乘客用户提交评价（也可以选择不评价）

α. 乘客用户取消指定约车请求：则对应约车请求与相关订单 end

β. 司机用户取消指定订单：对应约车请求重新排队，相关订单 end

γ. 司机用户结束听单(向服务端表达自己直到下一次开始听单前不再接单的意愿）

注:α、β、γ三步，可以在任意可行的时刻执行。（可行指客户端提供了该操作，例如司机开始听单后就有了γ对应的选项：结束听单，但在司机开始听单之前客户端不会提供该选项）而所有以自然数作为索引的流程，执行的顺序需要满足其索引的顺序，客户端和服务端都将对这些行为的顺序做校验与检测。

以上流程需要注意的是,α与β在第5步(司机用户向服务端汇报其接到乘客)之前的表现均为服务端维护相关请求/订单的状态，向客户端响应取消了相关订单。但一旦司机接到了乘客，那么无论是司机还是乘客都无权结束订单，直至司机抵达目的地附近并提交该订单为止。而γ在任意可行时刻都将汇报该司机不再听单的意愿。

非核心业务包括以下内容：

1. 获取验证码（没有调用付费api）
2. 司乘注册
3. 司乘登陆
4. 司机驾照注册（没有注册驾照的司机无法注册车辆）
5. 司机车辆注册（没有注册车辆的司机无法开始听单）
6. 司乘客户端回显已完成订单内容
7. 司乘删除已完成订单
8. 司机车辆形式路线规划（高德地图api）
9. 回显乘客对订单的评价（乘客评价方面已经纳入了核心业务流程生命周期中）

根据该需求文档，设计系统架构如下：

本次课设采用Client/Server前后端分离架构 。

客户端采用uniapp框架，可以一次编码，多次条件编译到各个不同平台运行，本项目主要面向安卓app产品市场，不适配其他软件平台。（h5,小程序,ios等）

服务端采用spring框架 ，采取单体架构 ，服务端架构上采取spring主流的Model-View-Service模块化解决方案。同时自主开发算法模块 ，专门负责项目中最核心的派单业务。其余工具类模块不再赘述。

数据源方面，采用多数据源。对于重要但读写较少的数据，基于MySQL磁盘数据库实现数据持久化；对于临时的但读写较多的数据，基于Redis内存数据库实现快I/O操作。在数据源操作框架方面，MySQL数据源采用Mybatis-plus框架，Redis采用基于lettuce的客户端。

第三方服务方面，采用高德地图API作为精准定位、实时位置监听、地图展示、地图标注、路线规划等功能的解决方案。

中间件方面，对于数据传输、数据转换功能，采用hu-tool作为解决方案；对于并发情况下的池化需求（线程池、数据库连接池），采用commons-pool2作为解决方案。

开发工具方面，采用lombok注解协助实体类开发；采用Mybatis-plus code generator作为模板代码生成器；采用velocity作为模板生成引擎；采用RESP作为Redis图形界面客户端；采用MySQLWorkBench作为MySQL图形界面客户端；采用HbuilderX作为客户端开发工具；采用JetBrains IDEA Ultimate作为服务端开发工具；采用Notion作为文档工具。

测试方面，采用HbuilderX作为客户端图形界面测试工具；采用Postman作为网络请求接口测试工具；采用Jmeter作为压力测试、性能测试工具。

在整个系统中，主要分为以下几个模块:

* 1. 用户权限鉴定模块
  2. 乘客用户约车请求模块（包括发起约车请求、撤销约车请求等）
  3. 司机用户操作订单模块（包括开始听单、结束听单、确认是否接受被派发到的订单、撤销已经接到的订单、接到乘客、提交订单等）
  4. 乘客用户订单评价模块（包括提交评价、回显评价等）
  5. 订单派发模块，负责将消息队列中的订单派发给司机
  6. 订单派发算法模块，负责根据地理位置维护空闲司机的数据结构，优先选取近点司机进行订单的派发。
  7. 用户完善信息模块（包括用户注册，司机注册驾照，司机注册车辆，用户隐藏已经完成的订单，乘客用户复评订单等）

在这些模块中，用户权限鉴定模块是整个系统中优先级最高的模块。在其他所有模块开始工作前，由用户权限鉴定模块检查向服务端发送的请求是否真的来自系统中一个在线的用户，如果没有通过身份验证，则拦截之。

用户约车请求模块和司机操作订单模块是订单派发模块的数据源。如果要进行订单派发，首先保证要有订单，然后保证要有司机。缺少司机则订单无人可派，缺少订单则没有派个司机的订单。因此这两个模块是订单派发模块所有行为的基础。

而订单派发算法模块（DBSCAN算法）是订单派发模块的“大脑”，他决定了将一个订单派发给哪个司机。（或可能这个订单无法派发给任何一个司机，比如在所有司机都在忙碌时）

## 3.2数据库设计

1. 为了区分不同的用户类型，需要表role

该表包含两个字段,id和role\_type，user表中role\_id字段将作为外键，引用该表的id字段

1. 为了持久化用户的基本信息，需要表user

user表包含如下字段:

id:主键，用户唯一标识

name:用户名，字符串

phone:手机号，字符串

email:邮箱，字符串

password:密码，字符串（需经过服务端加密，以密文存储）

credit:信用额度，整型

role\_id:外码，引用role表中id字段

deleted:逻辑删除字段，整型

version:版本锁字段，整型

注：所有表中都包含由deleted和version字段，之后的表设计不再赘述。

* 1. 为了持久化乘客用户的信息，需要表customer

customer表包含如下字段：

id:主键，乘客用户唯一标识

carpool\_rate：拼车率，浮点型

remain:账户余额，浮点型

user\_id:外码，引用user表中id字段

* 1. 为了持久化司机用户的信息，需要表driver

driver表中有如下字段

id:主键，司机用户唯一标识

comment\_rate:好评率,浮点型

income:收入，浮点型

user\_id:外码，引用user表中id字段

license:驾照类型，字符串

1. 为了持久化用户的约车请求的基本信息，需要表request

request表中有如下字段：

id:主键，约车请求唯一标识

customer\_id:外码，引用user表中id字段

request\_time:约车发起时间，时间戳

response\_time:约车响应时间，时间戳

priority:优先级，整型

carpool:是否拼车，bit型

accepted:是否有司机接单，bit型

des\_x:目的地经度，浮点型

des\_y：目的地纬度，浮点型

start\_x：上车点经度，浮点型

start\_y:上车点纬度，浮点型

start\_name:上车点名称，字符串

des\_name:目的地名称，字符串

* 1. 为了持久化用户预订约车的信息，需要表reservation

reservation表中包含以下字段:

id:主键，预约请求唯一标识

request\_id:外码，引用request表中id字段

reserve\_time:预约时间

* 1. 为了持久化用户代叫约车的信息，需要表instead

instead表中包含以下字段:

id:主键，代叫请求唯一标识

request\_id：外码，引用request表中id字段

customer\_phone:乘客手机号

1. 为了持久化形成的订单信息，需要表indent

indent表中包含以下字段：

id:主键，订单唯一标识

request\_id:外码，引用request表中id字段

driver\_id:外码，引用user表中id字段

start\_time:订单开始时间，时间戳

receive\_time:接到乘客时间，时间戳

end\_time:订单结束时间，时间戳

end\_x:最终实际结束位置经度，浮点型

end\_y：最终实际结束位置纬度，浮点型

distance:行车距离，浮点型

price:价格，浮点型

priority:优先级，整型

number:司机车牌号，字符串

driver\_deleted:司机是否选择隐藏，bit型

customer\_deleted:乘客是否选择隐藏，bit型

1. 为了持久化用户对订单的评价，需要表comment

comment表中包含以下字段:

id:主键，评价唯一标识

order\_id:外码，引用indent表中id字段

comment:评分，整形

content:评价内容，字符串

1. 为了持久化车辆的基本信息，需要表vehicle

vehicle表中包含以下字段：

id:主键，车辆唯一标识

number:车牌号，字符串

color:车辆颜色,字符串

type:驾车所需驾照类型，字符串

brand:车辆品牌，字符串

1. 为了持久化车辆从属的联系，需要表belong

唯一的联系集

belong表中包含以下字段：

id:主键，联系唯一标识。

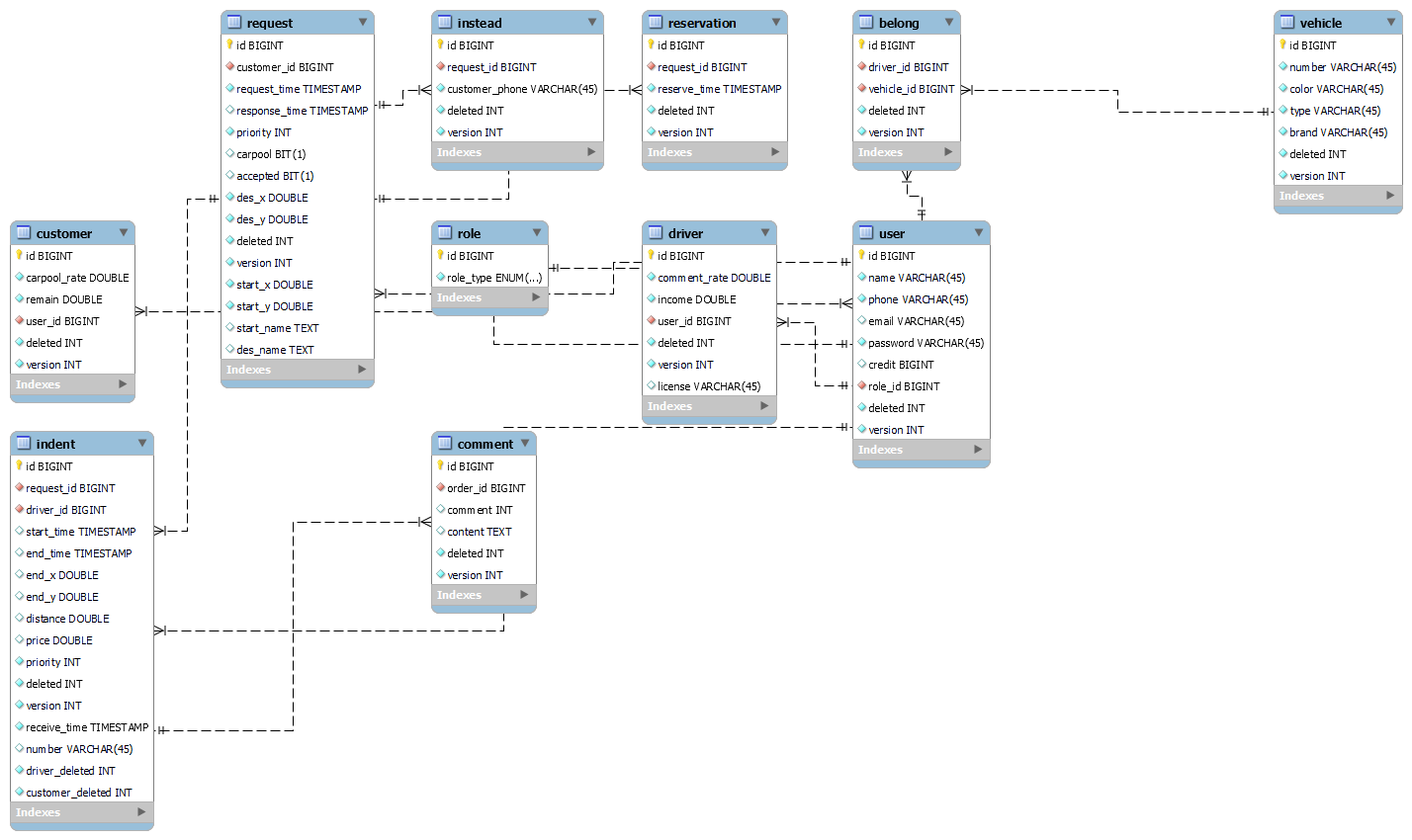
driver\_id:外码，引用user表中id字段

vehicle\_id:外码，引用vehicle表中id字段

其中2、3与其附属的a、b形成了泛化\特化的关系

### 3.2.1概念设计

该ER图由MySQLWorkBench导出。



### 3.2.2逻辑设计

由于 每一张表均有逻辑删除字段deleted和版本锁字段version。因此在关系模式中就不再写了。

* 1. role(id,role\_type)
  2. user(id,name,phone,email,password,credit,role\_id)
  3. customer(id,carpool\_rate,remain,user\_id)
  4. driver(id,comment\_rate,income,user\_id,license)
  5. request(id,customer\_id,request\_time,response\_time,priority,carpool,accepted,des\_x,des\_y,start\_x,start\_y,start\_name,des\_name)
  6. reservation(id,request\_id,reserve\_time)
  7. instead(id,request\_id,customer\_phone)
  8. indent(id,request\_id,driver\_id,start\_time,receive\_time,end\_time,end\_x,end\_y,distance,price,priority,number,driver\_deleted,customer\_deleted)
  9. comment(id,order\_id,comment,content)
  10. vehicle(id,number,color,type,brand)
  11. belong(id,driver\_id,vehicle\_id)

### 3.2.3物理设计

本项目数据的索引尽可能遵守主流项目开发所遵循的建立索引的原则。例如：选择唯一性索引、为排序、分组、联表查询的字段建立索引、为经常作为查询条件的字段建立索引。

接下来介绍表中的索引。

首先任何一张表都建立了主键索引，也即每张表的id字段上都有Primary索引。

* + 1. role表：除primary索引外没有别的索引
    2. user表：role\_id字段上有一般类型(Index)索引
    3. customer表:user\_id上有唯一(unique)索引,这来自于customer表与user表中role\_id=1的部分的双射映射。
    4. driver表: user\_id上有唯一(unique)索引，这来自于driver表与user表中role\_id=2的部分的双射映射。
    5. request表:customer\_id上有一般索引
    6. reservation表:request\_id上有一般索引
    7. instead表:request\_id上有一般索引
    8. indent表:request\_id和driver\_id上有一般索引
    9. comment表:order\_id上有一般索引
    10. vehicle表:除primary索引外没有别的索引
    11. belong表：driver\_id和vehicle\_id上均有唯一索引和一般索引

# 四、系统实现

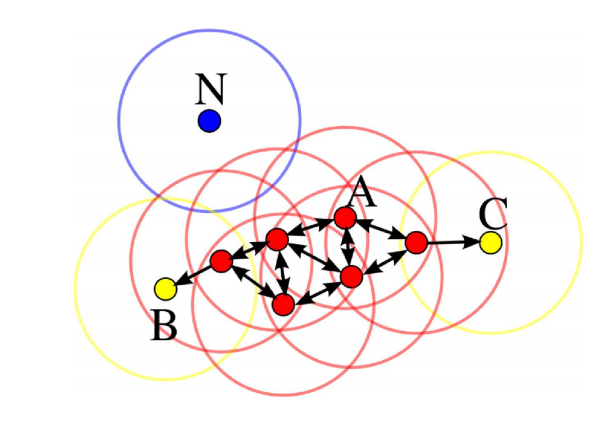
## 4.1关键技术实现（可选）

#### 1.算法模块

算法模块维护了一个数据结构，本质上是一个列表的列表，用来存放所有处于空闲状态的司机。这是为了方便我们在派单时检索最接近司机出发点的聚簇，而不是通过暴力搜索/线性搜索的方式去查找合适的司机。

该算法为聚类算法中的DBSCAN算法。

DBSCAN(Density-based spatial clustering of applications with noise)是Martin Ester, Hans-PeterKriegel等人于1996年提出的一种基于密度的聚类方法，聚类前不需要预先指定聚类的个数，生成的簇的个数不定（和数据有关）。该算法利用基于密度的聚类的概念，即要求聚类空间中的一定区域内所包含对象（点或其他空间对象）的数目不小于某一给定阈值。该方法能在具有噪声的空间数据库中发现任意形状的簇，可将密度足够大的相邻区域连接，能有效处理异常数据。



（DBSCAN算法原理图解）

以下是wiki-pedia上的DBSCAN伪代码。

DBSCAN(D, eps, MinPts) {

C = 0

for each point P in dataset D {

if P is visited

continue next point

mark P as visited

NeighborPts = regionQuery(P, eps)

if sizeof(NeighborPts) < MinPts

mark P as NOISE

else {

C = next cluster

expandCluster(P, NeighborPts, C, eps, MinPts)

}

}

}

expandCluster(P, NeighborPts, C, eps, MinPts) {

add P to cluster C

for each point P' in NeighborPts {

if P' is not visited {

mark P' as visited

NeighborPts' = regionQuery(P', eps)

if sizeof(NeighborPts') >= MinPts

NeighborPts = NeighborPts joined with NeighborPts'

}

if P' is not yet member of any cluster

add P' to cluster C

}

}

regionQuery(P, eps)

return all points within P's eps-neighborhood (including P)

我们可以建立聚簇集合存放整个空闲司机的列表，其中每个聚簇保存司机的id和实时位置。

这样我们可以将线性搜索整个空闲司机集合优化为线性搜索最接近乘客上车点的司机聚簇（后者往往数量远小于前者），然后从聚簇中选取首个愿意接受该订单的司机，将订单尝试分配给他。（询问他接受订单的意愿)

那么最关键的问题来了：司机是流动的，不是固定不动的，他们需要为了接单或其他原因不断地移动，一个固定的司机聚簇集合显然是不合理的。

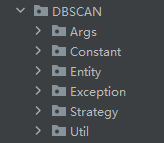
因此我们不能只是初始化一个容器并将空闲司机拉进去就完成任务，我们需要制定一个能够动态更新整个聚簇集合的解决方案。

这里采取的解决方案为，创建一个统计量：队列流水 。队列流水的初始值为0.当为一个司机分配一个订单后，队列流水增加1。当队列流水超出一个指定的阈值后，清空队列流水，然后更新聚簇。(利用当前空闲司机列表里的值更新）

以上策略依据一个没有经过统计但又符合直观感受的事实：当一个司机没有被派发订单时，他们通常都在附近小范围地移动；当一个司机被派发到一个订单，他可能需要移动很远的距离。

而队列流水在达到阈值前后的表现，根据上文，即：队列流水在阈值之下，小部分的司机产生大范围的移动，大部分的司机产生小范围的移动，总体上来说聚簇没有改变太多；队列流水达到阈值，大部分的司机产生大范围的移动，小部分的司机产生小范围的移动，聚簇发生巨大改变，需要更新。

本项目中使用的DBSCAN算法是我自主开发的，没有引入任何依赖。



接下来是该算法模块的整个包结构。

1. 上文提到过，DBSCAN算法需要依据两个参数，一个是半径eps，一个是最小传播点minPoints。因此我们需要一个Args类来为DBSCAN算法的calculator注入参数
2. 常量配置Constant，比如默认的坐标系标准参数等。
3. 算法模块实体类Entity ，聚簇实体、点实体以及计算结果存放的实体
4. 异常处理Exception，比如参数传递错误的异常类等。
5. 策略Strategy，主要针对距离策略，我们知道距离可以分很多种，比如曼哈顿距离（城乡街道距离）、欧式距离、地表距离等，我们需要为calculator注入一个计算策略。
6. 工具Util，主要是实现DBSCAN算法的部分。

下面是我实现的DBSCAN算法的核心代码(JAVA)

public void calculateClusters(Object[][] dataSet) throws DBSCAN\_POINT\_EXCEPTION {  
 setDataSet(dataSet);  
 double eps = args.getEps();  
 long minPts = args.getMinPts();  
  
 while(!unvisited.isEmpty()){  
 */\*\*  
 \* check if this point is noise or core point  
 \*/* Point point = unvisited.get(*LIST\_HEAD*);  
 List<Point> neighborPts = regionQuery(point, eps);  
 */\*\*  
 \* mark it as visited  
 \*/* unvisited.remove(*LIST\_HEAD*);  
 if(neighborPts.size()<minPts){  
 */\*\*  
 \* a noise point,push it into the noise list  
 \*/* noises.add(point);  
 }  
 else{  
 */\*\*  
 \* a core point,find its cluster  
 \*/* Cluster cluster = expandCluster(point, neighborPts, eps, minPts);  
 */\*\*  
 \* add this cluster to the cluster list  
 \*/* clusters.add(cluster);  
 }  
 }  
 }  
  
 private List<Point> regionQuery(Point p,double eps){  
 List<Point> ans = new ArrayList<>();  
 for (Point q : points) {  
 */\*\*  
 \* same coordinates,continue  
 \*/* if(p.equals(q)){  
 ans.add(q);  
 continue;  
 }  
 */\*\*  
 \* q is in the eps-neighbourhood of p  
 \*/* if(distanceStrategy.calculateDistance(p,q) <= eps){  
 ans.add(q);  
 }  
 }  
 return ans;  
 }  
  
 private Cluster expandCluster(Point core,List<Point> neighborPts,double eps,long minPts){  
 Cluster c = new Cluster(  
 new LinkedList<Point>(),  
 core  
 );  
 */\*\*  
 \* the core point must be an element of the cluster  
 \*/*// c.offer(core);  
 while(!neighborPts.isEmpty()){  
 */\*\*  
 \* update the neighbors  
 \*/* Point p = neighborPts.get(*LIST\_HEAD*);  
 if(!isVisited(p)){  
 unvisited.remove(p);  
 */\*\*  
 \* new neighborPts  
 \*/* List<Point> accretion = regionQuery(p, eps);  
 if(accretion.size()>=minPts){  
 */\*\*  
 \* propagate  
 \*/* neighborPts.addAll(accretion);  
 }  
 }  
 if(!isMemberOfAnyCluster(p)){  
 c.offer(p);  
 }  
 neighborPts.remove(*LIST\_HEAD*);  
 }  
 return c;  
 }  
  
  
 private boolean isVisited(Point p){  
 return !unvisited.contains(p);  
 }  
  
 private boolean isMemberOfAnyCluster(Point p){  
 for (Cluster cluster : clusters) {  
 if(cluster.contains(p)){  
 return true;  
 }  
 }  
 return false;  
 }

#### 2. **Request-Response,Blocking,Polling 和 WebSocket**

对于订单派发模块来说，如何通知司机客户端派发订单的结果是需要我们仔细考虑的。主要难点在于网络通信的方式上。

一个web应用，最常用的客户端与服务端通信的方式是Request-Response(请求响应）方式。客户端调用api发送请求到指定ip的指定端口的指定路径，服务端根据请求头（体）中的参数，予以校验、处理，然后响应结果。

但在本项目的系统设计中，派发订单的业务层是由一个single thread executor(单线程执行器)处理的，这个单线程执行器从redis消息队列中取出消息，然后尝试为这条消息中存放的约车请求分配一个司机。

也就是说，根本没有任何途径可以请求该单线程执行器，因为在该执行器中没有映射任何可供请求的接口！

即便我们更换系统的设计，提供一个可以被司机客户端请求的接口，该接口为司机提供一个订单。则存在两种可能的方案：

1. Blocking(阻塞)：服务端/数据库维护一个派单结果的集合，司机获取派单的请求来到后，不断地扫描该集合，直到发现一个为该司机分配的订单，然后带着这个结果响应；否则，该线程将一直等待。
2. Polling(轮询)：客户端设置定时器，不断地向服务端发送请求，每次请求来到后扫描一次且仅扫描一次派单结果集合，如果发现为该司机分配的订单，带着该订单信息响应；否则，响应为空。如果响应结果为空，则发送下一次请求；否则，停止定时器的重复发送请求的行为。

我们可以看到这两种解决方案都需要时时刻刻重复地检查派单结果，(Blocking是一次请求，每次请求多次检查；Polling是多次请求，每次请求检查1次）对服务端产生了巨大的性能影响。这对于项目非常不利，所以我们要抛弃这两种方案，寻求新的解决方式。

回顾这整个派单请求的流程，我们可以发现，我们之所以想要请求响应模式，是因为我们想要给司机的客户端响应一条分配订单的消息，也就是说，在请求响应两部分中，其实我们只想要响应这一部分，其实我们不关心请求的部分，因为请求只是为了给服务端提供一个响应的目的地而已！也就是说，如果我们知道目的地，我们就只想向客户端”响应“，压根就没请求什么事情了。

也就是我们要解决两件事情：第一，建立一个存在于服务端与客户端之间的连接，并且在服务端留存该客户端的”地址“；第二，仅在我们想要向客户端推送消息时服务端才推送消息，换言之，服务端拥有”实时“推送消息的能力。

由此引入全双工网络通信技术WebSocket ，ws使得服务端与客户端之间存在实时双向通信能力。例如直播弹幕、在线聊天室、用户群组广播、实时消息推送，都是采用ws实现的。

对于上文中提出的第一件事情，在ws连接时，留存客户端session至服务端维护的哈希表sessionMap中；对于第二件事，在需要向客户端推送消息时，调用ws的发送消息的api，向客户端发送消息。

ws的能力不止于此，上文提过，ws是“双向的”、“全双工的“。因此客户端也有能力通过ws向服务端发送消息，本项目的服务端司机实时位置更新，就是用ws实现的。

#### 3. 路线规划

路线规划的算法并非由我自己实现，而是采用调用第三方服务：高德地图路线规划api 。采用version3。

高德地图路线规划api官方文档:[路径规划-API文档-开发指南-Web服务 API | 高德地图API (amap.com)](https://lbs.amap.com/api/webservice/guide/api/direction" \l "driving)

路线规划回显到地图组件上的方式很简单：传入一个经过的地理坐标点的数组，然后对其进行连线，将线路回显到地图组件上。

而高德地图路线规划api就是做这件事情的,接口地址:<https://restapi.amap.com/v5/direction/driving?parameters>

具体传入参数:

{

key:key,//高德地图的key(web服务)

origin:{

longitude:lng,

latitude:lat//经度在前纬度在后

},

destination:,//同理origin

wayPoints:[{longitude:,latitude},...],//途径点，最多16个

extensions:all//展示完整的响应结果

}

使用官网的一个demo，可以大致观察到响应的结构:[restapi.amap.com/v3/direction/driving?origin=116.481028,39.989643&destination=116.465302,40.004717&extensions=all&output=json&key=](https://restapi.amap.com/v3/direction/driving?origin=116.481028,39.989643&destination=116.465302,40.004717&extensions=all&output=json&key=30ecd22f81bbeee46e90015c5f024fdf)${key}



可以看到，该api不仅返回了所有途径的点，甚至还包括不同策略：速度最快、距离最短、堵车最少等。更有行驶提示的文本，比如”向北行驶109米右转“等，可以说很全面了。

我们只需要把响应中的这组点设置到地图组件的polyline属性中，指定线宽和颜色，地图组件就能帮我们连点成线，绘制路线了。

## 4.2功能实现

按照系统设计，描述系统的主要功能是如何实现的，包括关键源代码分析和系统运行截图。

这里主要介绍核心业务流程。

1. 用户发起约车请求
   1. 先判断这个请求是否是伪造的(验证user\_id下属customer\_id有效)
   2. 生成请求，如果是特殊类型的请求(如代叫，预约)，则向reservation,instead表中添加特殊请求
   3. 执行lua脚本，将请求的具体信息写入redis消息队列
   4. 后续的接口想知道这条请求的具体参数，可以根据这三个字段向数据库中查询
   5. 执行lua脚本前应该先创建消息队列
   6. XGROUP CREATE customer.request g1 0 MKSTREAM
   7. 向服务端返回生成请求的id(防止精度丢失返回字符串)

代码如下：

private Request createRequest(RequestDTO requestDTO){  
 Long uid = requestDTO.getId();  
 Integer priority = requestDTO.getPriority();  
 Double startX = requestDTO.getStartX();  
 Double startY = requestDTO.getStartY();  
 String startName = requestDTO.getStart\_name();  
 Double desX = requestDTO.getDesX();  
 Double desY = requestDTO.getDesY();  
 String desName = requestDTO.getDes\_name();  
 //创建基础request  
 Request req = new Request();  
 req.setCustomerId(uid);req.setPriority(priority);req.setRequestTime(LocalDateTime.*now*());req.setStartX(startX);req.setStartY(startY);req.setStartName(startName);  
 req.setDesX(desX);req.setDesY(desY);req.setDesName(desName);  
 //存入数据库  
 save(req);  
 //获取自动生成的订单id  
 Long reqId = req.getId();  
 //如果是预约  
 Boolean isReservation = requestDTO.getIs\_reservation();  
 if (isReservation) {  
 LocalDateTime appointmentTime = requestDTO.getAppointment\_time();  
 Reservation reservation = new Reservation();  
 reservation.setRequestId(reqId);  
 reservation.setReserveTime(appointmentTime);  
 reservationDao.insert(reservation);  
 }  
 //如果是代叫  
 Boolean isInstead = requestDTO.getIs\_instead();  
 if(isInstead){  
 String customerPhone = requestDTO.getCustomer\_phone();  
 Instead instead = new Instead();  
 instead.setRequest\_id(reqId);  
 instead.setCustomer\_phone(customerPhone);  
 insteadDao.insert(instead);  
 }  
 return req;  
}

系统运行截图如下：



1. 乘客撤销约车请求
   1. 伪造请求校验，对于每一条reqeustList中的记录，查询是否为该id的乘客之前发起的请求
   2. 如果是，不能暴力return，因为要保持这个请求操作的原子性，应该通过ws通知客户端具体哪一条撤销失败
   3. 如果不是，继续执行
   4. 逻辑删除该请求
   5. 查询该请求是否是特殊请求，如果是，级联删除特殊请求
   6. 查询该请求是否有对应生成的订单
   7. 如果没有对应订单
   8. 逻辑删除请求
   9. 通知乘客请求已经撤销(响应)
   10. 如果有对应订单
   11. 检查是否已经接到了乘客，接到乘客后无论司机还是乘客都不能再取消订单
   12. 逻辑删除对应订单，同时维护司机订单driverOrder哈希表
   13. 检查司机是否在线
   14. 如果不在线，说明司机已经取消听单，不需要更新他的状态，只要直接通知司机订单已经被取消
   15. 如果在线，判断删除订单后司机应该是什么状态。如果状态较之前的状态发生变化，说明司机从忙碌变为空闲(只可能发生这种情况，因为不可能删除订单后从空闲变为忙碌)，维护司机状态DriverStatus哈希表，更新队列流水，将司机加入最近的聚簇
   16. 从忙碌司机列表中取出订单对应的司机，加入到最近的聚簇中，同时更新队列流水
   17. ws通知订单对应的司机订单已经被取消
   18. 通知乘客请求已经撤销(响应)

代码如下：

public Result cancelRequest(CancelRequestDTO cancelRequestDTO) {  
 //伪造请求校验  
 Long usr = cancelRequestDTO.getId();  
 List<Long> requestList = cancelRequestDTO.getRequestList();  
 for (Long reqId : requestList) {  
 Integer count = requestDao.queryCountByRequestIdAndCustomerId(reqId, usr);  
 if(count==null||count==0){  
 //这里不能暴力return，可能影响前面或后面的对数据库的操作  
 *sendMessage*("指定删除的请求不合法:"+reqId,*sessionMap*.get(usr));  
 continue;  
 }  
 //获取该请求对应的订单  
 Indent indent = indentDao.queryOrderByReqId(reqId);  
 if(null== indent){  
 //没有对应的订单，直接把请求删了  
 requestDao.logicDeleteRequestByReqId(false,reqId);  
 //仅通知乘客订单取消  
 *sendCancelRequestResult*(*cancelRequest*,*sessionMap*.get(usr));  
 continue;  
 }  
 //有对应订单  
 Long driverId = indent.getDriverId();  
 //检查是否已经接到乘客  
 RequestInfo receiveCheck = *driverRunning*.get(driverId);  
 if(receiveCheck!=null&&receiveCheck.getReqId().equals(reqId)){  
 *sendMessage*("司机已经接到您了,不能再取消订单:"+reqId,*sessionMap*.get(usr));  
 continue;  
 }  
 synchronized (reqId){  
 //逻辑删除该请求  
 requestDao.logicDeleteRequestByReqId(false,reqId);  
 //查询是否是特殊请求,级联删除  
 if(*is\_Reservation*(reqId)){  
 reservationDao.logicDeleteReservationByReqId(reqId);  
 }  
 if(*is\_Instead*(reqId)){  
 insteadDao.logicDeleteInsteadByReqId(reqId);  
 }  
 //逻辑删除对应订单  
 indentDao.logicDeleteOrderByReqId(reqId);  
 log.debug("司机"+driverId+"订单列表:"+ *driverOrder*.get(driverId));  
 //同时删除哈希表中订单  
 boolean has\_Order = *deleteOrder*(reqId, driverId);  
 if(!has\_Order){  
 log.debug("该订单不存在");  
 }  
 }  
 //检查司机是否在线  
 DriverInfo onlineCheck = *DriverStatus*.get(driverId);  
 if(onlineCheck==null){  
 //司机已经取消听单，不需要更新他的状态  
 *queueFlow*++;  
 //通知司机订单已经取消 要加上reqId啊，不然前端怎么知道哪个订单被取消了可恶  
 *sendCancelRequestResult*(*cancelRequest*+reqId,*sessionMap*.get(driverId));  
 }  
 //司机在线,判断司机此时应该是什么状态  
 boolean status = *judgeStatus*(driverId);  
 if(status!=onlineCheck.getIdle()){  
 //不相等只有一种情况，从忙碌到空闲，因为不可能删除订单把司机从空闲删除到忙碌的  
 //从忙碌司机列表中取出订单对应的司机  
 DriverInfo driverInfo = *changeStatus*(driverId,true);  
 *queueFlow*++;  
 //将该司机加入距离最近的聚簇中  
 *addDriverToCluster*(driverInfo);  
 }  
 //通知司机订单已经取消 要加上reqId   
 *sendCancelRequestResult*(*cancelRequest*+reqId,*sessionMap*.get(driverId));  
 }  
 return Result.*ok*();  
}

1. 司机开始听单
   1. 即使ws负责了实时位置更新，我们仍要在开始听单时附上司机位置，因为这涉及到我们应该在空闲司机列表中存放什么样的数据的问题。
   2. 伪造请求校验:是否是司机
   3. 判断该司机是否已经注册车辆，没有注册车辆无法开始听单
   4. 查询该司机是否已经接单，如果已经开始接单，响应客户端
   5. 封装空闲司机列表中需要的对象，添加至空闲司机列表，更新队列流水
   6. 将该司机添加至距离最近的一个聚簇中
   7. 响应客户端已经开始听单

代码如下：

public Result startBusiness(DriverDTO driverDTO) throws DBSCAN\_ARGS\_EXCEPTION, DBSCAN\_POINT\_EXCEPTION {  
 //伪造请求校验:是否是司机  
 Long driverId = driverDTO.getId();  
 Long role = userDao.queryRoleByUID(driverId);  
 if(role!=*Driver\_Role*){  
 return Result.*fail*("当前用户不为司机类型，不能开始听单");  
 }  
 //判断该司机是否已经注册车辆，没有注册车辆无法开始听单  
 Integer count = belongDao.queryCountByDriver(driverId);  
 if(count==null||count==0){  
 return Result.*fail*("当前用户未注册车辆，不能开始听单");  
 }  
 //先查询这个司机是否已经在听单  
 if(*DriverStatus*.get(driverId)!=null){  
 return Result.*fail*("当前司机已经处于接单状态");  
 }  
 //查询id对应司机的信用分，和当前定位一起封装为DriverInfo  
 User user = userDao.queryById(driverId);  
 DriverInfo driverInfo = new DriverInfo(driverId, driverDTO.getCurX(), driverDTO.getCurY(), user.getCredit(),*judgeStatus*(driverId));  
// log.debug(driverInfo.getIdle().toString());  
 log.debug(*driverOrder*.get(driverId)==null?"":*driverOrder*.get(driverId).toString());  
 log.debug(*driverRunning*.get(driverId)==null?"":*driverRunning*.get(driverId).toString());  
 synchronized(user.getId()) {  
 //将司机信息添加到空闲司机列表中  
 *DriverStatus*.put(driverId,driverInfo);  
 //如果是第一个加入的司机，那么updateBusyIdleFlow会初始化并更新聚簇，就不需要将当前司机加入最近的聚簇了  
 if(!*updateBusyIdleFlow*()){  
 //在队列流水到达阈值之前，寻找一个距离当前节点最近的key，将该司机加入对应的聚簇中  
 *addDriverToCluster*(driverInfo);  
 }  
 }  
 return Result.*ok*();  
 }

系统运行截图如下：



1. 司机结束听单
   1. 从单一职责的方面来考虑，结束听单只用于向服务端更新当前司机的听单意愿，而不应该牵涉任何订单分配、更新与删除的功能。如果我们在更新司机听单意愿的同时，选择将司机已经分配的所有订单取消，那么司机为了完成当前订单，他只能选择推迟对自己听单意愿的更新，也就是说，原先的设计将导致司机在本身不想听单时却向系统汇报自己仍愿意听单。这会导致，由于司机向系统汇报自己仍愿意听单，则系统会持续给司机分发订单。当这个分发结果送到司机手中时，因为司机本身不再愿意听单了（他只是为了保留自己手上的订单而选择汇报不符合实际的意愿），所以他会手动选择拒绝订单，那么这个订单对应请求的消息将回到消息队列重新排队。当这样汇报错误意愿的司机变多后，乘客将面临一种困境，为该乘客分配的请求遭到许多伪报愿意接单的司机的拒绝，这个请求不得不反复地回到消息队列重新分配，然后反复地经过系统计算分配给下一个很可能也是伪报意愿的司机，然后循环上面的过程。这对乘客来说效率极低，因此需要修改结束听单的流程，让这个接口只做一件事情：汇报司机听单与否的意愿给服务器。而司机主动移除订单的功能，另外设计接口实现。
   2. 查询是否为伪造的请求(先查询数据库确保这个id确实是个司机，然后向用户登陆状态信息的ws查询，看当前司机是否在线)
   3. 在空闲司机列表中遍历查询，移除并级联更新聚簇，如果成功，响应司机他已经结束听单
   4. 在忙碌司机列表中遍历查询，移除，如果成功，响应司机他已经结束听单
   5. 经过了伪造检验，又不在空闲且不在忙碌司机列表，后端代码写错掉了，响应前端逻辑出问题了

代码如下：

public Result endBusiness(Long id) {  
 //查询是否是伪造的请求  
 Long role\_id = userDao.queryRoleByUID(id);  
 if(role\_id!=*Driver\_Role*||*sessionMap*.get(id)==null){  
 return Result.*fail*("当前司机不存在");  
 }  
 //从在线司机列表中移除该司机  
 DriverInfo driverInfo = *DriverStatus*.remove(id);  
 //从拒绝订单哈希表中移除该司机  
 *driverReject*.remove(id);  
 //从聚簇中移除该司机  
 *removeDriverFromCluster*(id);  
 if(driverInfo!=null){  
 //响应司机他已经取消听单  
 return Result.*ok*();  
 }  
 return Result.*fail*("服务端代码逻辑异常:正常请求结束听单的司机不在在线司机列表中");  
}

1. 司机确认订单
   1. 司机确认订单的本质是一个请求-响应模式下的行为，在司机发起确认订单的请求前，我们要让他知道自己已经被分配了一个订单，所以：ws通知司机分配给他的订单消息，询问他是否接单。
   2. 伪造请求检验:司机是否在线(正处于接单状态,如果结束听单那么服务器理应不会给司机分配订单，司机也就没有能力确认订单)，只有在线的司机才能确认订单
   3. 如果司机不接受订单，保持他现有的状态(空闲/忙碌)。因为可能要求司机确认的订单是个预约单，预约单完全可以分配给一个现在正在接别的单的司机。
   4. 获取该订单对应的请求，将这个请求利用lua脚本重新写回到redis消息队列中排队
   5. 应该记录司机拒绝订单的信息，以后不再将其分配给该司机
   6. 在后端维护一个司机拒绝听单的哈希表，记录该司机所有拒绝的订单
   7. 响应司机订单拒绝成功，他在将来的一段时间内应该不会再接到这个订单了
   8. 如果司机接受订单
   9. 如果他接受的是预约单，保持他现有的状态
   10. 如果他接受的是实时单。线程并发问题:如果一个司机没有立即返回服务器他是否愿意接单，可能在短时间内由于他是空闲状态而服务器给他分配了两单实时单，但一个司机同一时间只能接一单实时单，那么如果他接了前一单，无论后一单他是否接受，都应该被写回消息队列。因此先判断该司机是否已经处于忙碌状态，如果处于忙碌状态，将请求写回消息队列，响应客户端当前司机正在忙碌，不能再接实时单。司机状态空闲变忙碌，把司机从聚簇中移除。
   11. 更新队列流水，如果队列流水达到阈值，更新聚簇
   12. 更改请求信息的某些字段，比如是否接单以及响应时间
   13. 生成订单，交由driverOrder哈希表维护
   14. ws响应乘客他的订单已经被某个司机接受

代码如下：

public Result confirmOrder(Long id, Long requestId, Boolean accept) {  
 //伪造请求检验:司机是否在线  
 if(*DriverStatus*.get(id)==null){  
 return Result.*fail*("该司机已经离线");  
 }  
 //无论是否接受，把他从driverDispatch中放出来  
 *driverDispatch*.remove(id);  
 //获取请求消息  
 Request req = requestDao.queryRequestIdByReqId(requestId);  
 //不接受订单  
 if(!accept){  
 //将该订单计入司机拒绝的订单列表中  
 recordReject(id,requestId);  
 //通过lua脚本将其放回消息队列，重新排队  
 rewriteRequest(requestId);  
 return Result.*ok*();  
 }  
 //线程并发问题:如果一个司机没有立即返回服务器他是否愿意接单  
 //可能在短时间内由于他是空闲状态而服务器给他分配了两单实时单  
 //但一个司机同一时间只能接一单实时单，那么如果他接了前一单，无论后一单他是否接受，都应该被写回消息队列  
  
 */\*\*  
 \* 线程并发问题:司机确认接受前应该确保请求仍存在  
 \*/* Integer count = requestDao.queryRequestCountByReqId(req.getId());  
 if(count<=0){  
 return Result.*fail*("乘客已经取消了约车请求,无法接受订单");  
 }  
  
 //是否是预约单  
 Integer reserveCount = reservationDao.queryCountByReqId(requestId);  
 //实时单  
 if(reserveCount==null||reserveCount==0) {  
 DriverInfo driverInfo = *DriverStatus*.get(id);  
 if (!driverInfo.getIdle()) {  
 rewriteRequest(requestId);  
 return Result.*ok*("当前司机已经处于忙碌状态,不能再接实时单");  
 }  
 //司机状态空闲变忙碌  
 if (driverInfo.getIdle()) {  
 *changeStatus*(id, false);  
 //从聚簇中移除  
 *removeDriverFromCluster*(id);  
 }  
 }  
 //更新队列流水  
 *queueFlow*++;  
 //若队列流水达到某一阈值，更新聚簇  
 if(*queueFlow*>=*queueFlowMax*){  
 try {  
 *updateClusters*();  
 } catch (DBSCAN\_ARGS\_EXCEPTION | DBSCAN\_POINT\_EXCEPTION e) {  
 throw new RuntimeException(e);  
 }  
 }  
 Long customerId = req.getCustomerId();  
 //更改请求信息，针对是否接单以及响应时间  
 LocalDateTime now = LocalDateTime.*now*();  
 log.debug("司机接单时间:"+now.toString());  
 requestDao.updateRequestByReqId(true, now,requestId);  
 //生成订单,并交由哈希表维护  
 createOrder(id,req);  
 //创建订单具体信息  
 RequestInfo requestInfo = new RequestInfo();  
 requestInfo.setReqId(requestId);  
 requestInfo.setIs\_reservation(*is\_Reservation*(requestId));  
 requestInfo.setIs\_instead(*is\_Instead*(requestId));  
 //获取司机对应订单列表  
 List<RequestInfo> reqInfoList = *driverOrder*.get(id);  
 if(null== reqInfoList){  
 reqInfoList = new ArrayList<>();  
 }  
 //加入新的订单  
 reqInfoList.add(requestInfo);  
 //重设司机对应订单列表  
 *driverOrder*.put(id,reqInfoList);  
 log.debug("司机"+id+"订单列表:"+reqInfoList.toString());  
 log.debug("当前在线用户:"+*sessionMap*.toString());  
 //ws响应乘客司机的信息  
 //查询所有相关信息  
 Long vehicleId = belongDao.queryVehicleIdByDriverId(id);  
 Vehicle vehicle = vehicleDao.queryVehicleById(vehicleId);  
 User user = userDao.queryById(id);  
 //这个时候cancelInfo为空  
 StringWSDriverMessageForCustomer stringWSDriverMessageForCustomer = new StringWSDriverMessageForCustomer(requestId.toString(), vehicle.getNumber(), vehicle.getType(), vehicle.getColor(), vehicle.getBrand(), id.toString(), user.getName(), user.getPhone(), user.getCredit().toString(),null);  
 *sendObject*(stringWSDriverMessageForCustomer,*sessionMap*.get(customerId));  
 return Result.*ok*();  
}

系统运行截图如下：



1. 司机接到乘客
   1. 检查司机是否在线，如果在线将司机状态改为忙碌(即便他已经是忙碌状态)
   2. 更新其正在跑的单子，把他从聚簇中移除
   3. 响应司机服务器确认了这一消息
   4. 而手机号校验交给了客户端来完成，客户端通过本地缓存(localStorage)存储了乘客的手机号，乘客上车好进行尾号的校验，通过即可开始本次行程

代码如下：

public Result receivePassenger(Long id,Long reqId) {  
 */\*\*  
 \* 线程并发问题:司机接到乘客前应确保对应请求仍存在  
 \*/* Integer count = requestDao.queryRequestCountByReqId(reqId);  
 if(count<=0){  
 return Result.*fail*("乘客已经取消对应约车请求,无法接到该乘客");  
 }  
  
 //司机可能在取消听单后接到之前确认过的订单的乘客，这个时候changeStatus返回null，直接跳过了  
 //无论司机现在是什么状态，把他置为忙碌  
 DriverInfo driverInfo = *changeStatus*(id,false);  
 if(driverInfo!=null){  
 //更新其正在跑的单子  
 *driverRunning*.put(id,*getRequestInfo*(reqId));  
 //从聚簇中移除  
 *removeDriverFromCluster*(id);  
 //更新当前order的开始时间  
 indentDao.updateOrderStartTimeByReqId(LocalDateTime.*now*(),reqId);  
 Long customerId = requestDao.queryCustomerIdByReqId(reqId);  
 *sendObject*(new WSReceivePassengerForCustomer(reqId.toString(),true),*sessionMap*.get(customerId));  
 }  
 return Result.*ok*();  
}

系统运行截图如下：



1. 司机提交订单（到达目的地后）
   1. 校验是否为伪造的请求(数据库中是否有对应订单，这个订单是否逻辑存在)
   2. 校验当前结束位置是否和请求中要求的结束位置大致一致(地球距离不大于一个阈值)
   3. 如果有效，根据提供的信息计算价格
   4. 网约车订单计价:起步价+时间费+里程费+超里程费+夜间费
   5. 更改数据库中对应订单的对应信息项
   6. 将司机状态修改为空闲，清空他正在跑的单子
   7. 将司机加入聚簇中
   8. ws通知乘客他的订单已经完成，包括需要支付的价格以及对应的请求id
   9. 响应司机他已经提交了订单，以及将得到的支付的价格

代码如下：

public Result commitOrder(CommitOrderDTO commitOrderDTO) {  
 //校验是否为伪造的请求:数据库中是否有对应订单，这个订单是否逻辑存在  
 Integer count = indentDao.queryCountByRequestIdAndDriverId(commitOrderDTO.getReqId(), commitOrderDTO.getId());  
 if(count==null||count==0){  
 return Result.*fail*("数据库中没有对应的合法订单");  
 }  
 //校验当前结束位置是否和请求中要求的结束位置大致一致  
 Request request = requestDao.queryRequestIdByReqId(commitOrderDTO.getReqId());  
 //计算等待时间差  
 LocalDateTime requestTime = request.getRequestTime();  
 LocalDateTime responseTime = request.getResponseTime();  
 Duration between = Duration.*between*(requestTime, responseTime);  
 long minute = between.toMinutes();  
 Double startX = request.getStartX();  
 Double startY = request.getStartY();  
 Double desX = request.getDesX();  
 Double desY = request.getDesY();  
 Double endX = commitOrderDTO.getEndX();  
 Double endY = commitOrderDTO.getEndY();  
 Geo\_Distance geoDistance = new Geo\_Distance(*defaultEllipsoid*);  
 double tarDiff = geoDistance.calculateDistance(  
 new Point(desX, desY),  
 new Point(endX, endY)  
 );  
 if(tarDiff>*maxTargetDiff*){  
 return Result.*fail*("距离实际目的地太远,无法提交订单");  
 }  
 //更改数据库中对应订单的对应信息项  
 //需要更改 price,distance,endX,endY,endTime  
 //测算distance  
 double distance;  
 distance = geoDistance.calculateDistance(  
 new Point(startX, startY),  
 new Point(endX, endY)  
 )/1000;  
 //如果有效，根据提供的信息计算价格  
 double price = 0.0;  
 //网约车订单计价:起步价+时间费+里程费+超里程费+夜间费  
 //一般来说不同类型的车在不同区域这些价格都不一样，但我懒得做了  
 //所以统一起步价:14米，里程费3.6米/公里，时间费(等待费)1.2米/分钟，超里程阈值300公里，超里程费2米/公里，夜间域为23:00-05:00，夜间费0.4米/公里  
 double mileCost = (Math.*min*(distance,*OverMileAge*)\**MileAgeCost*);  
 double waitCost = minute\**WaitCost*;  
 double overMileCost = distance>*OverMileAge*?(distance-*OverMileAge*)\**OverMileAgeCost*:0;  
 log.debug("行车距离:"+distance+",里程费:"+mileCost+",等待费:"+waitCost+",超里程费:"+overMileCost);  
 price+=14.0+mileCost+waitCost+overMileCost;  
 //获取该订单的结束时间  
 LocalDateTime endTime = LocalDateTime.*now*();  
 //*TODO:计算夜间费* //更新订单信息  
 indentDao.updateOrderByRequestIdAndDriverId(price,distance,endX,endY,endTime,commitOrderDTO.getReqId(),commitOrderDTO.getId());  
 //司机可能在取消听单后提交之前的订单，这个时候在线司机列表里是访问不到这个司机的,为空直接跳过了  
 //将司机状态修改为空闲，清空他正在跑的单子，同时清空他接收的订单列表中的该单子  
 DriverInfo driverInfo = *changeStatus*(commitOrderDTO.getId(), true);  
 if(driverInfo!=null){  
 *driverRunning*.remove(commitOrderDTO.getId());  
 }  
 *deleteDriverOrder*(commitOrderDTO.getId(),commitOrderDTO.getReqId());  
 //把司机放到聚簇中  
 *addDriverToCluster*(driverInfo);  
 //ws通知乘客他的订单已经完成，包括需要支付的价格以及对应的请求id  
 log.debug("当前司机状态:"+*DriverStatus*.get(commitOrderDTO.getId())+",当前聚簇大小:"+*getClustersSize*()+",当前司机正在跑的单子:"+*driverRunning*.get(commitOrderDTO.getId()));  
 StringWSOrderMessageForCustomer stringWSOrderMessageForCustomer = new StringWSOrderMessageForCustomer();  
 stringWSOrderMessageForCustomer.setPrice(price+"");  
 stringWSOrderMessageForCustomer.setReqId(request.getId().toString());  
 *sendObject*(stringWSOrderMessageForCustomer,*sessionMap*.get(request.getCustomerId()));  
 //响应司机他已经提交了订单，以及将得到的支付的价格  
 return Result.*ok*(price);  
}

系统运行截图如下：



1. 司机撤销已经接到的订单
   1. 伪造请求校验，对于每一个需要删除的请求，检查数据库中是否存在对此的订单
   2. 对于每个需要删除的请求，修改其是否被接单以及响应时间字段
   3. 逻辑删除其对应的订单
   4. lua脚本将请求写回消息队列重新排队
   5. 根据该司机是否应为空闲状态修改其状态
   6. 响应司机已经删除了请求

代码如下：

public Result deleteOrder(DeleteOrderDTO deleteOrderDTO) {  
 //伪造请求校验  
 Long id = deleteOrderDTO.getId();  
 List<Long> requestList = deleteOrderDTO.getRequestList();  
 for (Long reqId : requestList) {  
 Integer count = indentDao.queryCountByRequestIdAndDriverId(reqId, id);  
 if(count==null||count==0){  
 return Result.*fail*("非法请求:订单不存在");  
 }  
 }  
 RequestInfo running = *driverRunning*.get(id);  
 //由于该司机删除了这一订单，所以他拒绝的订单中应该添加这一项  
 List<Long> reject = *driverReject*.get(id);  
 for (Long reqId : requestList) {  
 if(running!=null&&Objects.*equals*(reqId, running.getReqId())){  
 return Result.*fail*("不能删除已经接到乘客的订单");  
 }  
 //修改是否被接单以及响应时间(没被接单,没得到响应)  
 requestDao.updateRequestByReqId(false,null,reqId);  
  
 //这里应该通知乘客司机取消接单了，更换下一个司机，不然乘客的前端还有原来司机的信息啊  
 StringWSDriverMessageForCustomer stringWSDriverMessageForCustomer = new StringWSDriverMessageForCustomer(reqId.toString(), null, null, null, null, null, null,null,null,"订单已取消");  
 Request request = requestDao.queryRequestIdByReqId(reqId);  
 *sendObject*(stringWSDriverMessageForCustomer,*sessionMap*.get(request.getCustomerId()));  
 synchronized (reqId){  
 //逻辑删除对应订单  
 indentDao.logicDeleteOrderByReqId(reqId);  
 //lua脚本写回消息队列  
 rewriteRequest(reqId);  
 //将其从司机订单列表中移除  
 *deleteDriverOrder*(id,reqId);  
 //添加到司机拒绝的订单中  
 *addToRejectList*(id,reqId);  
 }  
 }  
 //更新司机状态  
 DriverInfo driverInfo = *DriverStatus*.get(id);  
 if(driverInfo!=null){  
 //司机在线，更新司机状态  
 boolean status = *judgeStatus*(id);  
 log.debug("司机取消订单后的状态应为:"+(status?"空闲":"忙碌"));  
 driverInfo.setIdle(status);  
 if(status){  
 //如果司机处于空闲状态，应该把他加入回聚簇  
 *addDriverToCluster*(driverInfo);  
 }  
 }  
 //响应司机已经删除了请求  
 return Result.*ok*();  
}

1. 服务端派发订单
   1. 分配订单不需要请求，而是服务器从redis消息队列中取出消息，然后尝试将这些消息转换成订单分发给空闲的司机
   2. 在redis中创建消费者组作为消息队列
   3. 开启独立线程，专门用来执行分配订单的工作
   4. 获取消息队列中的消息
   5. 转换为请求消息
   6. 确认消息
   7. 查询请求是否还存在，因为可能乘客已经取消了请求
   8. 如果请求还存在，继续执行
   9. 否则，进入下一轮消息处理
   10. 为该订单分配司机
   11. 查询当前聚簇集合是否存在，不存在则更新聚簇
   12. 从聚簇集合中查询距离当前请求上车点位置距离最近的聚簇
   13. 获取该聚簇的首个元素
   14. 查询该司机是否拒绝了该订单，如果拒绝，将该司机放到队尾
   15. 如果一个聚簇的所有司机都拒绝了该订单，换下一个聚簇，重复b的操作
   16. 如果所有司机都拒绝了该订单，将该订单写回消息队列，然后返回
   17. 此时要结合上述的司机确认订单。ws通知司机他被分配到的订单
   18. 异常处理：Redis提供了pending机制，如果一条消息没有得到确认，将列入pending list中，我们可以从该列表中重新取出因为异常而没有得到处理的消息，重新处理。

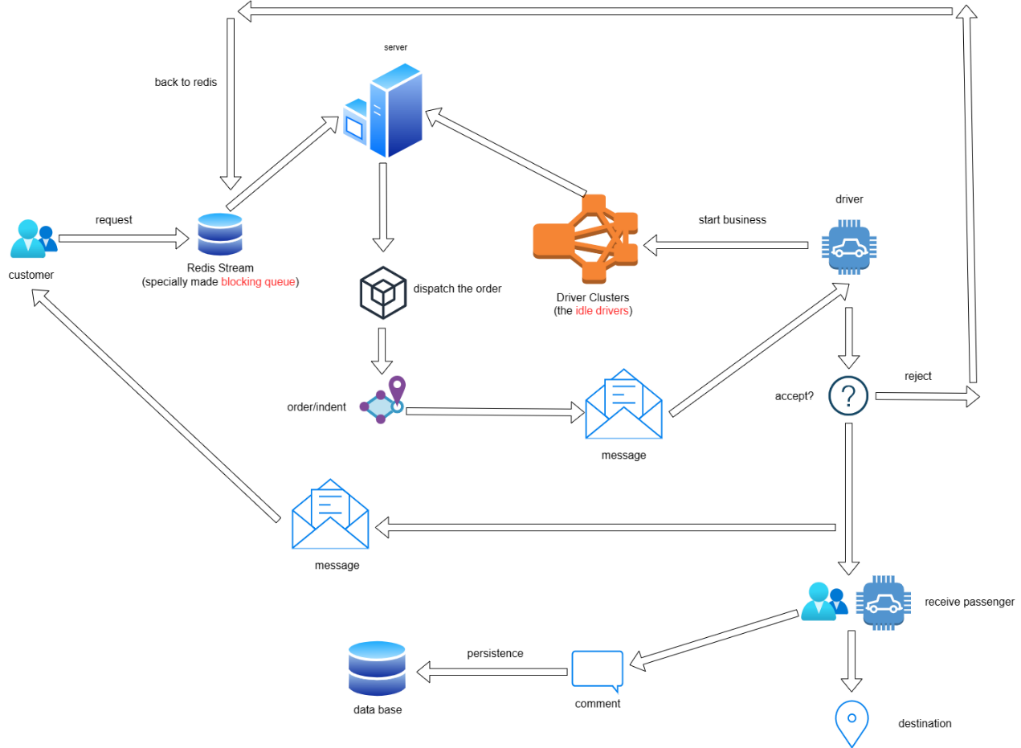
代码如下：

public void run() {  
 while(true){  
 if(*getIdleCount*()==0){  
// log.debug("当前没有空闲司机,聚簇中司机数量:"+getClustersSize());  
 //当前没有空闲司机，没法分配订单  
 continue;  
 }  
 try {  
 //获取消息队列中的订单信息  
 List<MapRecord<String, Object, Object>> req = stringRedisTemplate.opsForStream().read(  
 Consumer.*from*(*ORDER\_CONSUMER\_GROUP*, *ORDER\_CONSUMER*),  
 StreamReadOptions.*empty*().count(1).block(Duration.*ofSeconds*(2)),  
 StreamOffset.*create*(*STREAM\_ORDER*, ReadOffset.*lastConsumed*())  
 );  
 if(req == null || req.isEmpty()){  
 continue;  
 }  
 MapRecord<String, Object, Object> record = req.get(0);  
 //确认消息  
 stringRedisTemplate.opsForStream().acknowledge(*STREAM\_ORDER*,*ORDER\_CONSUMER\_GROUP*,record.getId());  
 Map<Object, Object> value = record.getValue();  
 RequestInfo requestInfo = BeanUtil.*fillBeanWithMap*(value, new RequestInfo(), true);  
 Long reqId = requestInfo.getReqId();  
// log.debug("请求id:"+reqId);  
 synchronized (reqId){  
 //查询数据库，判断请求是否还存在  
 Integer count = requestDao.queryRequestCountByReqId(reqId);  
 if(count==null||count==0){  
 continue;//订单已经取消  
 }  
 //分配订单  
 Request request = requestDao.queryRequestIdByReqId(reqId);  
 log.debug(*ORDER\_DISPATCHING\_Info*(reqId));  
 dispatchOrders(request,requestInfo.getIs\_instead(),requestInfo.getIs\_reservation());  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 try {  
 handlePendingList();  
 } catch (DBSCAN\_ARGS\_EXCEPTION | DBSCAN\_POINT\_EXCEPTION ex) {  
 throw new RuntimeException(ex);  
 }  
 }  
 }  
 }

private synchronized void dispatchOrders(Request request,Boolean is\_instead,Boolean is\_reservation) throws DBSCAN\_ARGS\_EXCEPTION, DBSCAN\_POINT\_EXCEPTION {  
 //查询聚簇是否存在，不存在更新聚簇  
 if(null == *clusters* || null == *queueFlow*){  
 *updateClusters*();  
 }  
 log.debug(*CLUSTER\_SIZE\_Info*());  
// System.out.println(CLUSTER\_SIZE\_Info());  
 //轮询所有聚簇直到发现一个没有拒绝过该订单的司机  
 boolean flag = false;//是否存在没有拒绝过的司机  
 Long driverId = null;  
 //先把当前位置转换为Point实例  
 Point point = *pointTransfer*(request);  
 int length = *clusters*.size();  
 LeastDisCalculator leastDisCalculator = new LeastDisCalculator(*defaultEllipsoid*);  
 Long reqId = request.getId();  
 outer:  
 for(int i=1;i<=length;i++){  
 Cluster kthLeastDistanceCluster = leastDisCalculator.getKthLeastDistanceCluster(point, *clusters*, i);  
 for (Point p : kthLeastDistanceCluster.getCq()) {  
 DBSCAN\_driverPoint driverPoint = (DBSCAN\_driverPoint) p;  
 long dId = driverPoint.getUId();  
 if(*driverDispatch*.get(dId)!=null){  
 //已经为他分配了订单，且该司机还没确认订单  
 continue ;  
 }  
 List<Long> rejectList = *driverReject*.get(dId);  
 //该司机拒绝的列表中不包含该请求  
 if(rejectList==null||!rejectList.contains(reqId)){  
 flag = true;  
 driverId = dId;  
 break outer;  
 }  
 }  
 }  
 *driverDispatch*.put(driverId,reqId);  
 log.debug(*DRIVER\_REJECT\_LIST\_Info*(driverId));  
 //分配不成功,没法写回消息队列了，不然只有一条消息的时候会疯狂反复处理然后服务器崩掉  
 if(!flag){  
 *sendMessage*("没有司机愿意接单",*sessionMap*.get(request.getCustomerId()));  
// rewriteRequest(reqId);  
 return;  
 }  
 log.debug(*DISPATCH\_DRIVERID\_Info*(driverId));  
 //将订单消息通知给司机客户端  
 StringWSRequestMessageForDriver message = createMessage(request, is\_instead, is\_reservation);  
 String dispatchResult = *DISPATCH\_RESULT*(reqId, driverId);  
 log.debug(dispatchResult);  
 //日志以追加的方式写入到指定文件当中  
 *recordLog*(*logTimePrefixSuffix*(dispatchResult),*logFilePath*,true);  
  
 *sendDispatchResult*(message,*sessionMap*.get(driverId));  
 }

1. DBSCAN算法实现
   1. 首先是包结构
   2. 上文提到过，DBSCAN算法需要依据两个参数，一个是半径eps，一个是最小传播点minPoints。因此我们需要一个Args类来为DBSCAN算法的calculator注入参数
   3. 常量配置Constant，比如默认的坐标系标准参数等。
   4. 算法模块实体类Entity ，聚簇实体、点实体以及计算结果存放的实体
   5. 异常处理Exception，比如参数传递错误的异常类等。
   6. 策略Strategy，主要针对距离策略，我们知道距离可以分很多种，比如曼哈顿距离（城乡街道距离）、欧式距离、地表距离等，我们需要为calculator注入一个计算策略。
   7. 工具Util，主要是实现DBSCAN算法的部分。

这里附上一张基于DBSCAN聚簇集合的流程图。



具体的流程上文已经有过介绍，这里不再赘述。这张流程图不同的意义在于集成了DBSCAN生成的聚簇集合。

代码如下：（仅核心代码）

public void calculateClusters(Object[][] dataSet) throws DBSCAN\_POINT\_EXCEPTION {  
 setDataSet(dataSet);  
 double eps = args.getEps();  
 long minPts = args.getMinPts();  
  
 while(!unvisited.isEmpty()){  
 */\*\*  
 \* check if this point is noise or core point  
 \*/* Point point = unvisited.get(*LIST\_HEAD*);  
 List<Point> neighborPts = regionQuery(point, eps);  
 */\*\*  
 \* mark it as visited  
 \*/* unvisited.remove(*LIST\_HEAD*);  
 if(neighborPts.size()<minPts){  
 */\*\*  
 \* a noise point,push it into the noise list  
 \*/* noises.add(point);  
 }  
 else{  
 */\*\*  
 \* a core point,find its cluster  
 \*/* Cluster cluster = expandCluster(point, neighborPts, eps, minPts);  
 */\*\*  
 \* add this cluster to the cluster list  
 \*/* clusters.add(cluster);  
 }  
 }  
 }  
  
 private List<Point> regionQuery(Point p,double eps){  
 List<Point> ans = new ArrayList<>();  
 for (Point q : points) {  
 */\*\*  
 \* same coordinates,continue  
 \*/* if(p.equals(q)){  
 ans.add(q);  
 continue;  
 }  
 */\*\*  
 \* q is in the eps-neighbourhood of p  
 \*/* if(distanceStrategy.calculateDistance(p,q) <= eps){  
 ans.add(q);  
 }  
 }  
 return ans;  
 }  
  
 private Cluster expandCluster(Point core,List<Point> neighborPts,double eps,long minPts){  
 Cluster c = new Cluster(  
 new LinkedList<Point>(),  
 core  
 );  
 */\*\*  
 \* the core point must be an element of the cluster  
 \*/*// c.offer(core);  
 while(!neighborPts.isEmpty()){  
 */\*\*  
 \* update the neighbors  
 \*/* Point p = neighborPts.get(*LIST\_HEAD*);  
 if(!isVisited(p)){  
 unvisited.remove(p);  
 */\*\*  
 \* new neighborPts  
 \*/* List<Point> accretion = regionQuery(p, eps);  
 if(accretion.size()>=minPts){  
 */\*\*  
 \* propagate  
 \*/* neighborPts.addAll(accretion);  
 }  
 }  
 if(!isMemberOfAnyCluster(p)){  
 c.offer(p);  
 }  
 neighborPts.remove(*LIST\_HEAD*);  
 }  
 return c;  
 }  
  
  
 private boolean isVisited(Point p){  
 return !unvisited.contains(p);  
 }  
  
 private boolean isMemberOfAnyCluster(Point p){  
 for (Cluster cluster : clusters) {  
 if(cluster.contains(p)){  
 return true;  
 }  
 }  
 return false;  
 }

1. 乘客提交评价
   1. 伪造请求校验：订单是否存在，评分是否合法
   2. 获取对应订单id，生成评价记录，存入数据库
   3. 响应乘客已经提交了评价

代码如下：

public Result commentOrder(CommentOrderDTO commentOrderDTO) {  
 //伪造请求校验  
 //comment是否合法  
 Integer comment = commentOrderDTO.getComment();  
 if(comment!=null&&(comment<0||comment>5)){  
 return Result.*fail*("评价星级不合法,应为空或在1到5之间");  
 }  
 //是否匹配请求  
 Long id = commentOrderDTO.getId();  
 Long reqId = commentOrderDTO.getReqId();  
 Request request = requestDao.queryRequestIdByReqId(reqId);  
 if(!request.getCustomerId().equals(id)){  
 return Result.*fail*("乘客与请求不匹配");  
 }  
 Indent indent = indentDao.queryOrderByReqId(reqId);  
 if(indent==null){  
 return Result.*fail*("不存在对应订单");  
 }  
 //生成评价记录  
 Comment record = new Comment();  
 record.setComment(comment);  
 record.setOrder\_id(indent.getId());  
 record.setContent(commentOrderDTO.getContent());  
 save(record);  
 return Result.*ok*();  
}

系统运行截图如下：



# 五、系统测试

其实在附录中我给出了测试文档，但为了防止没看到测试文档的情况，这里再摘录一份。

我将测试分为三个环节，分别为：

1. 客户端测试，主要测试客户端和服务端交互逻辑是否正常
2. 接口测试，主要测试服务端接口是否能合理地处理请求（包括异常值、边界值等）
3. 压力测试，主要测试服务器性能，包括高qps下的服务器行为，以及线程并发问题

## 5.1 客户端测试

#### 0. 测试目标

通过客户端界面进行司乘约车以及完成订单的核心流程(主要演示一个订单的完整生命周期）

#### 1. 环境准备

1. 可以通过uniapp框架提供的官方编辑器HbuilderX编译运行客户端
2. 或者可以通过cli的方式编译运行整个客户端项目

具体详情请见uniapp官网:[uni-app官网 (dcloud.net.cn)](https://uniapp.dcloud.net.cn/quickstart-hx.html)

#### 2. 测试流程

整个测试流程涉及两个(或至少两个)客户端，请确保您复现该测试时拥有足够的设备。

1. 事先声明

本项目仅关注安卓app端，如果在其他端(ios,微信小程序,浏览器)出现任何异常、错误或无问题，本项目概不负责。

1. 司乘登陆

这建立在已经注册的前提下，如果还没有为司机/乘客注册账号，可以通过客户端界面，或sql脚本，注册账号。

这里假设已经注册好了两个账号，在登陆时，请选择正确的用户类型，否则将无法登陆。

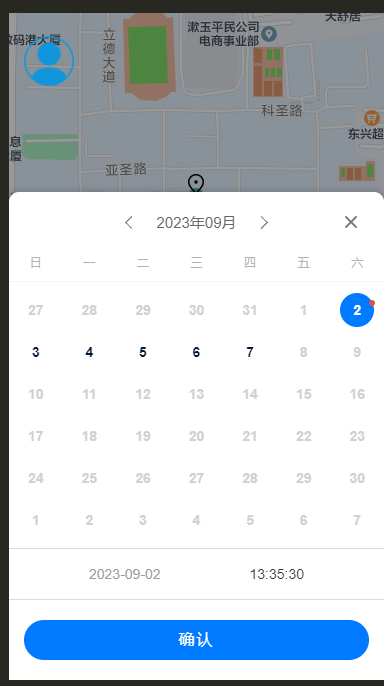
本系统规定，登陆时，只要验证码和密码有一个正确即可，当然，你同时填写两个，且均正确，也是可以登陆的。



1. 乘客发起约车请求
   1. 请选择合法的上车点与下车点。可以在搜索栏输入地点，高德地图/腾讯地图的api将为您搜索相关地点。
   2. 您也可以选择是否代叫或预约，这全按照用户的需求决定。
   3. 点击开始约车即可发送约车请求







1. 司机开始听单
   1. 司机点击开始听单，即可表达自己准备听单的意愿



1. 系统分配订单
   1. 这个对于用户来说是黑盒的，具体实现参照其他文档与压测环节。
2. 司机接到订单
   1. 司机可以选择接单或不接，如果不接，之后他将不会再收到这个订单。



1. 司机开始导航
   1. 在已经接单列表中，点击某一订单的导航图标，即可开始导航
   2. 点击后，地图上将出现导航路线与乘客上下车点





1. 司机接到乘客
   1. 司机点击接到乘客，确认手机尾号正确后，即可开始接送乘客。





1. 司机提交订单
   1. 注意！如果司机距离目的地太远，将无法提交订单！
   2. 提交订单后，乘客端将收到评价提醒





1. 乘客提交评价
   1. 填写评分级别
   2. 填写评价内容
   3. 点击提交即可



## 5.2接口测试

本接口测试采用的工具为Postman。官网地址:[Postman API Platform | Sign Up for Free](https://www.postman.com/)

Postman最新版本提供了webSocket连接的接口，这也是我选择该应用作为接口测试工具的原因之一。

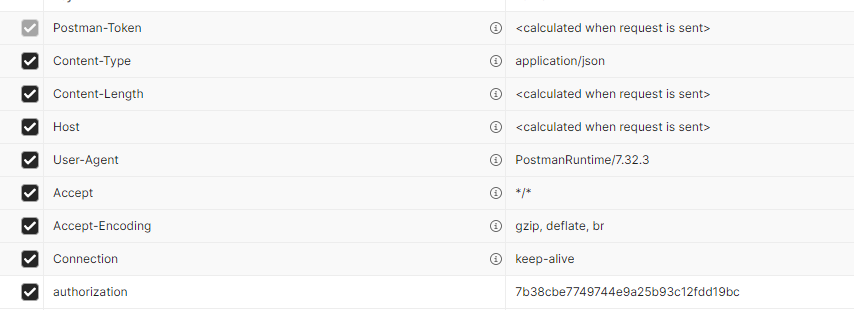
## 0. 测试目标

一个司机和一个乘客登陆，完成核心业务流程中除了乘客提交评价的任何接口的测试。

核心业务流程具体参照系统演示文档。

## 1.环境准备

1. 下载安装Postman软件，注册账户并登陆，创建所需的connection文件
2. 可以通过sql脚本，也可以通过客户端，或者可以通过Postman本身向负责注册业务的接口发送请求。总之，注册一个司机和一个乘客。
3. 通过客户端或Postman向负责登陆业务的接口发送请求。总之，获取这两个账号登陆的token。
4. 在Postman中所有需要鉴权的接口的Header中添加请求头字段authorization:${token}



1. 确保服务端处于运行状态

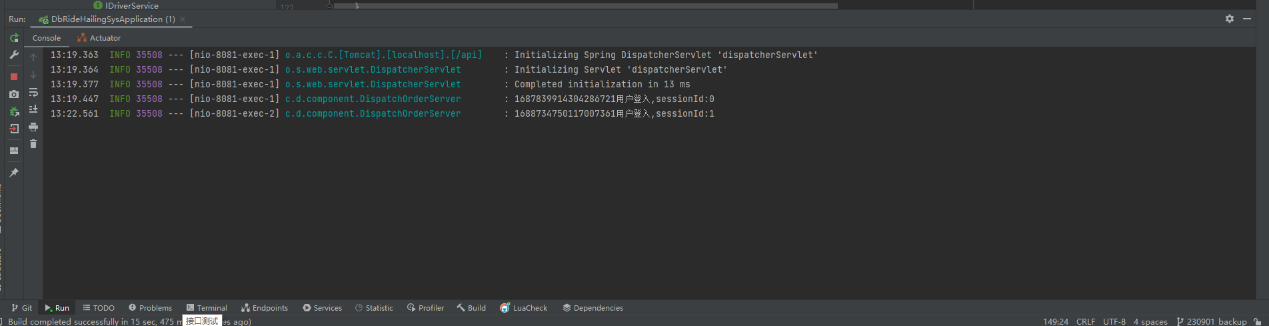
## 2. 测试流程

### 2.1 建立WebSocket连接

向服务端的指定路径分别建立与司机、乘客客户端（Postman模拟)的ws连接。

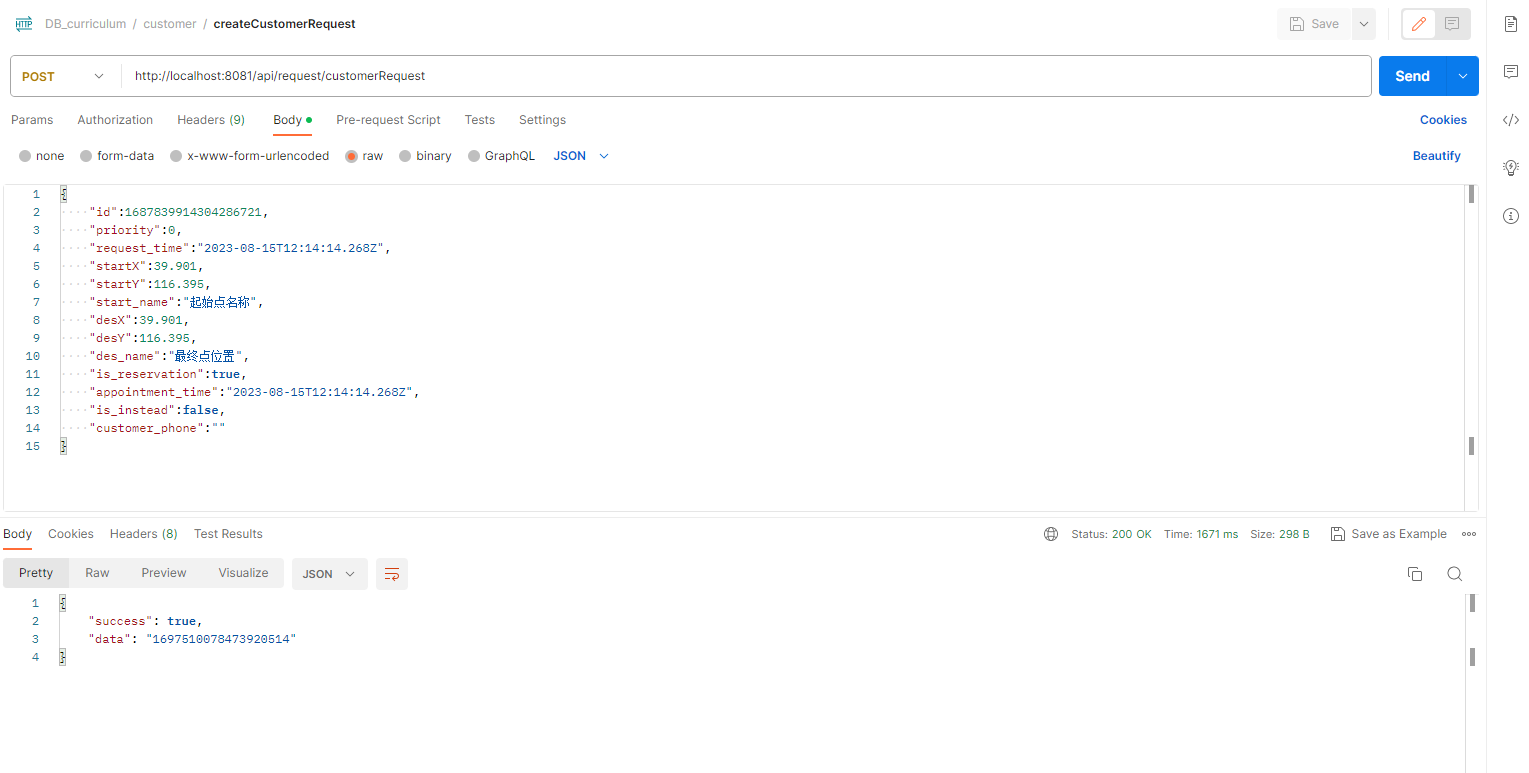


服务端我配置了控制台日志，可以看到用户已经连接上了。



### 2.2 乘客用户发起约车请求

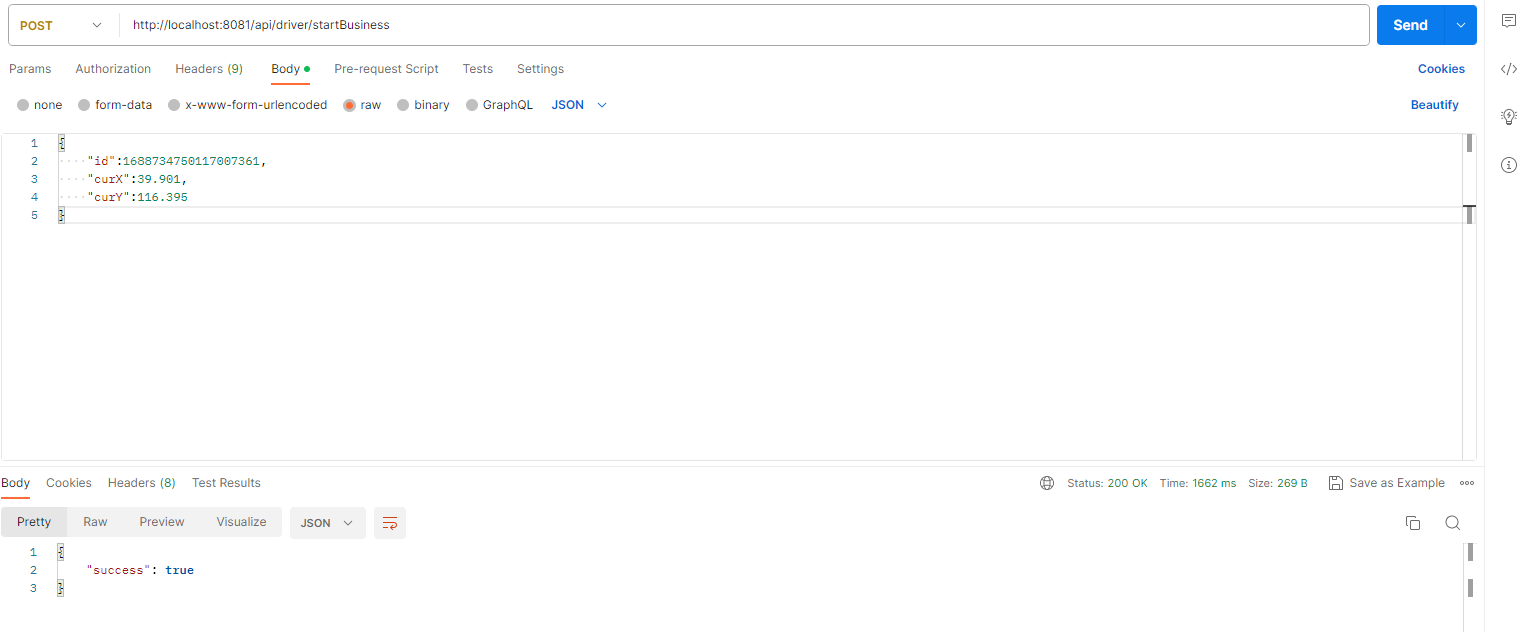
你可以自己创建这个文件，或者导入我附录中的文件。请求的参数可以自己修改。



发出请求后，获得带有请求id的响应。请注意保留这个请求id，这是为了测试后续的接口。

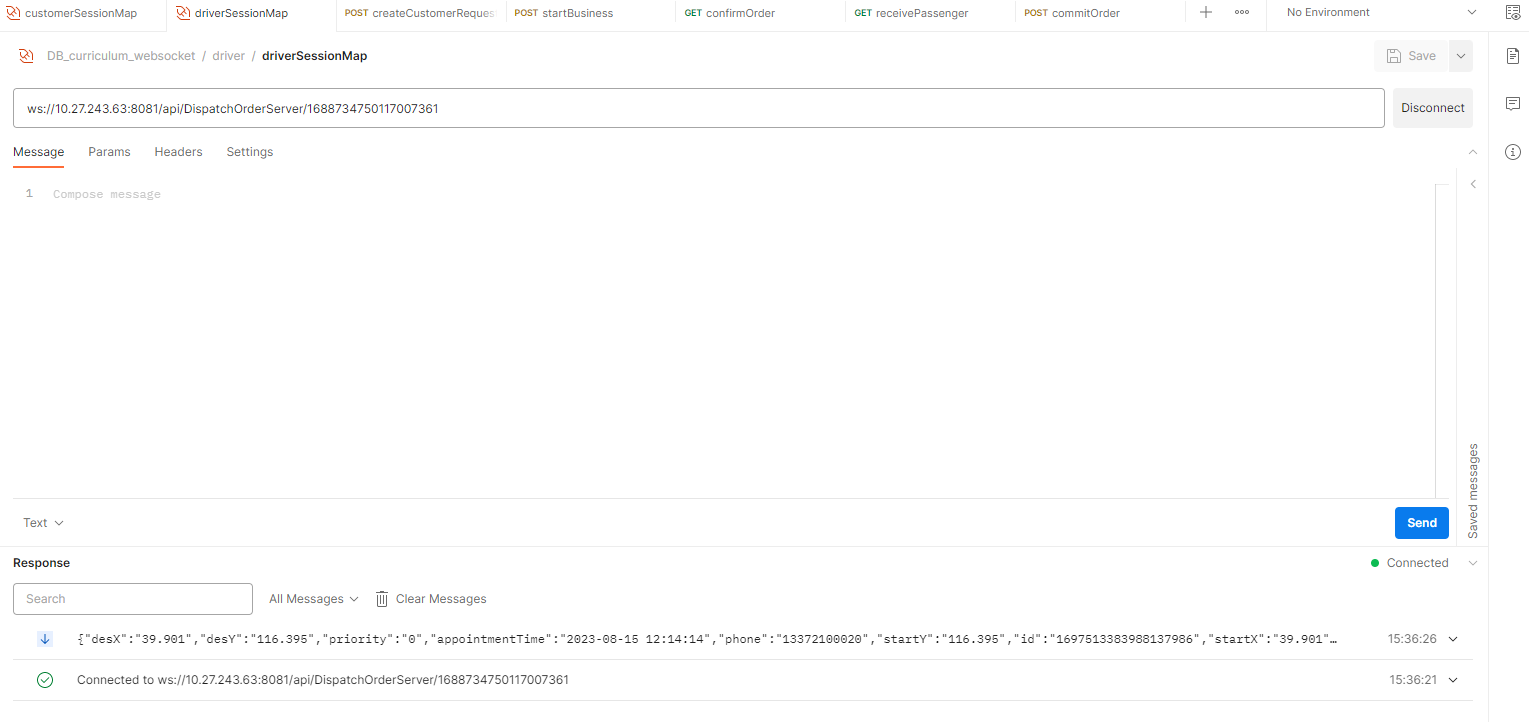
### 2.3 司机开始听单

参数可以自己修改

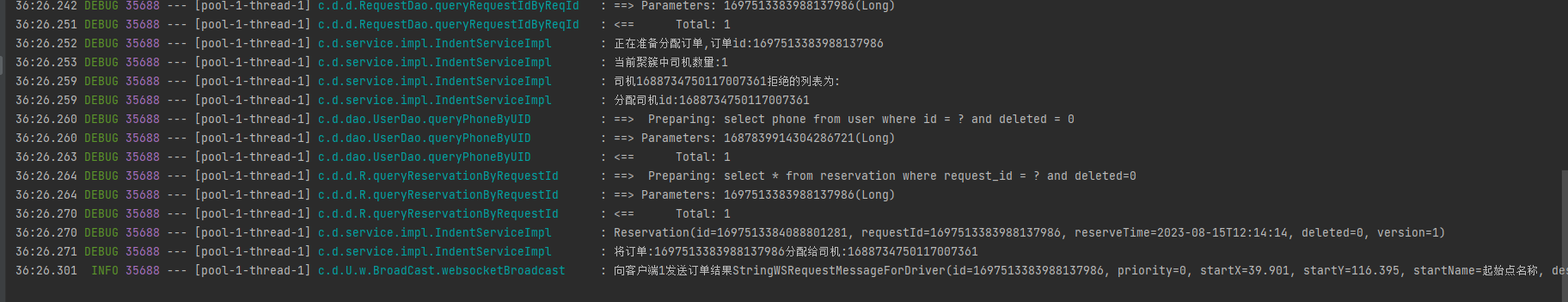


由于当前系统中只有这一个刚开始听单的空闲司机，于是刚刚发出的约车请求一定会派发给该司机。

我们回到ws的连接中查看，果然，分配给了该司机。

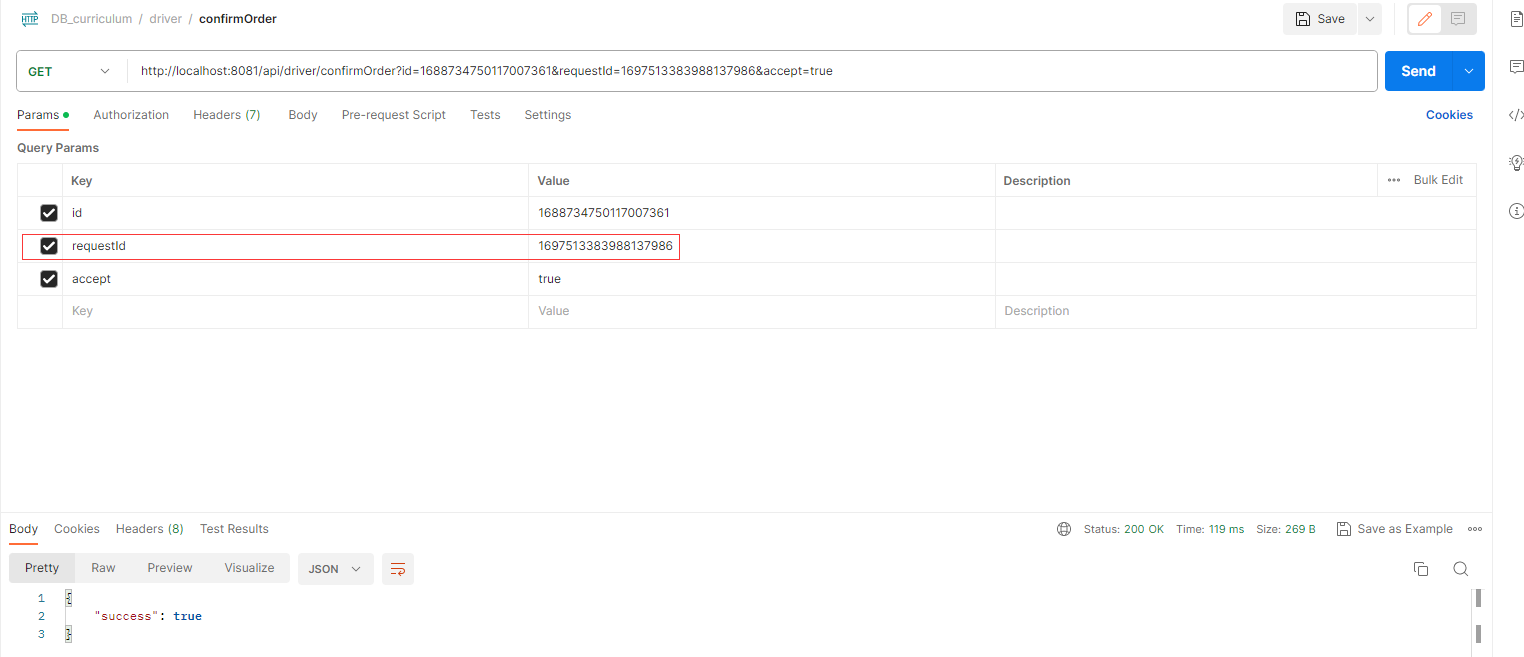


随后，控制台的日志中也打印了将订单分配给该司机的过程。



### 2.4 司机接受订单

把ws中的请求id复制到这个测试文件中的requestId参数中。

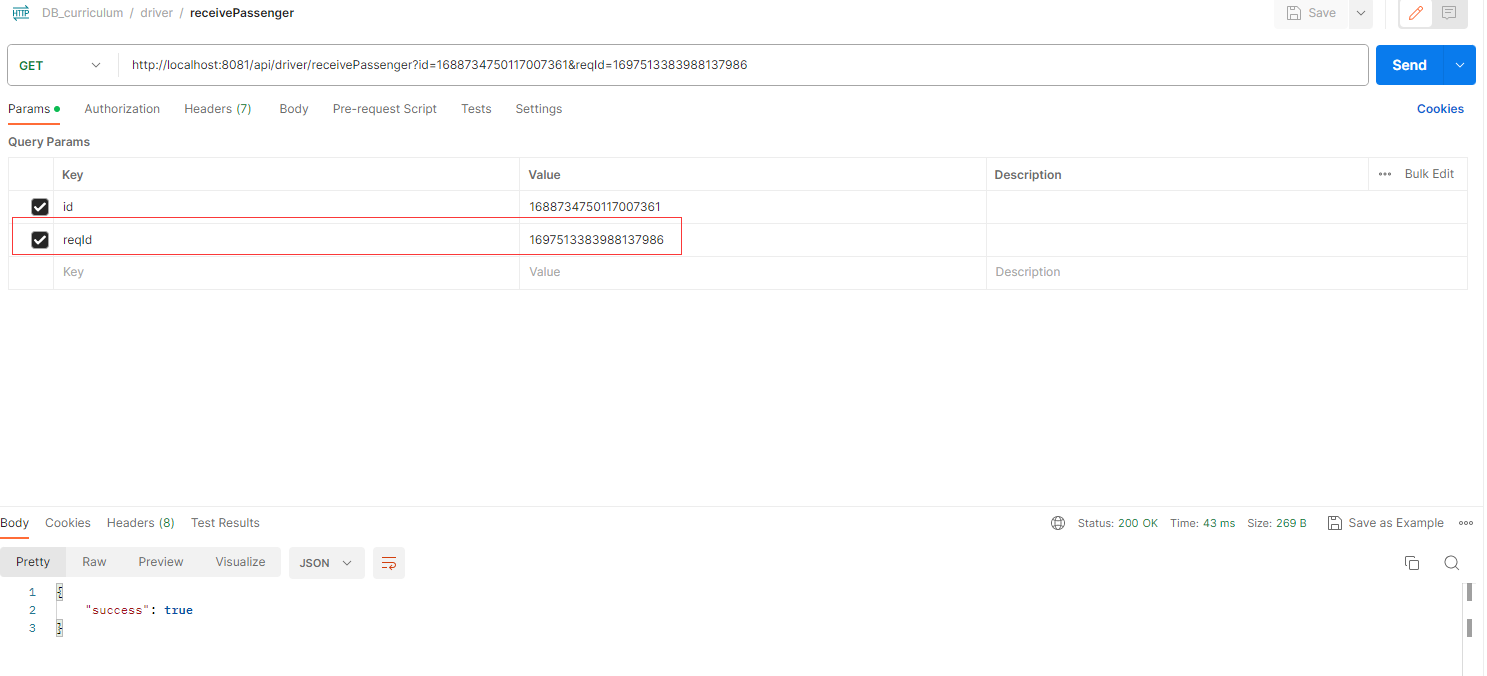


回到乘客的ws连接中，发现已经将司机和车辆信息发送给了乘客客户端



### 2.5 司机接到乘客

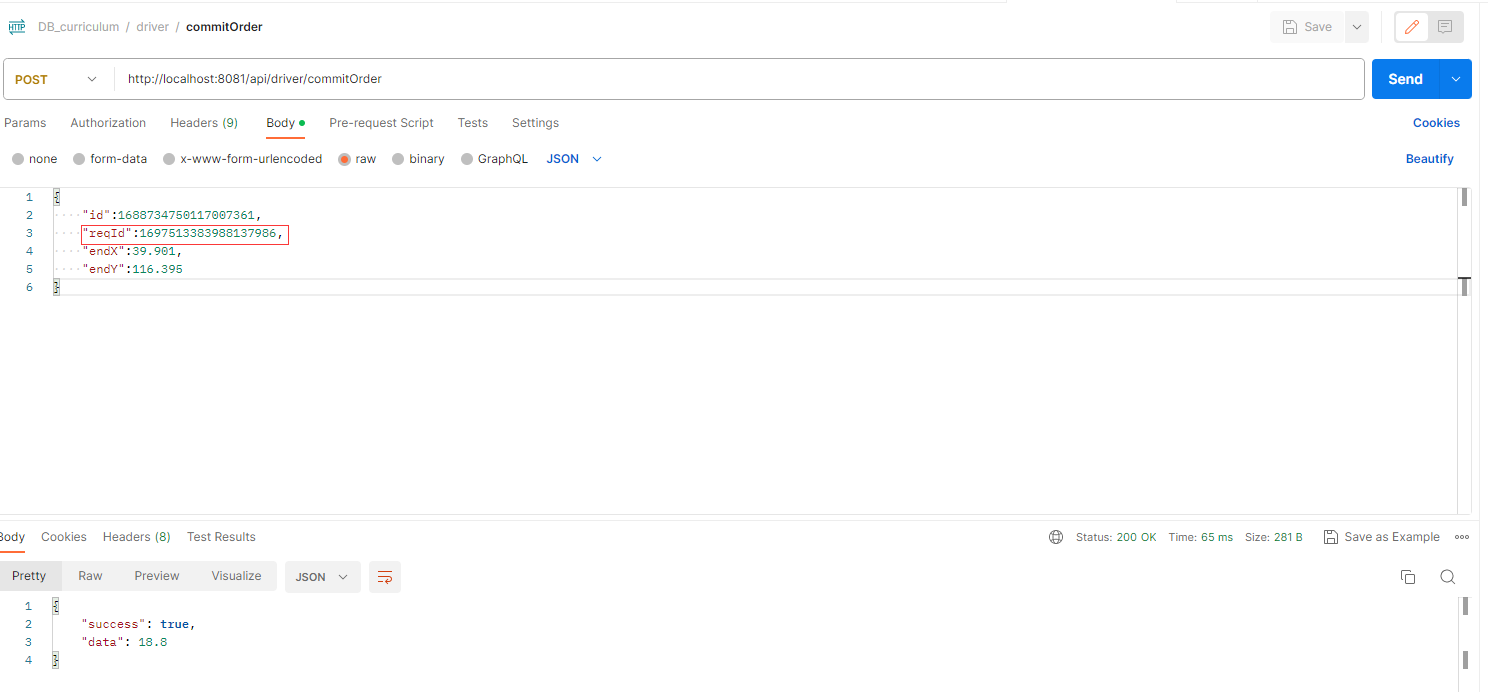
把请求id复制到reqId字段中。



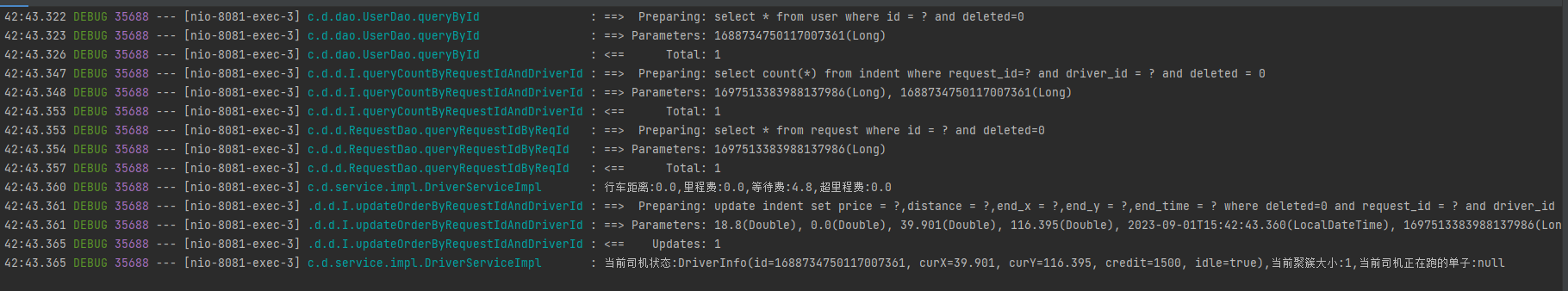
这个环节的业务比较简单，我就没有在控制台打日志。

### 2.6 司机到达目的地后提交订单

把请求id复制到reqId字段中。



在服务端控制台我打印了日志，至此，该订单结束。



注：如果你想测试别的接口，请自己编写测试文件，系统庞大、接口繁杂，我在有限的时间内仅凭一个人很难完成所有测试工作。

## 5.3 压力测试

## 0.测试目标

数据量：1000乘客发起约车请求、1000司机开始听单

具体服务：派单业务，服务端为这1000个约车请求以及司机分配订单

## 1.环境准备

环境准备分为三个部分：MySQL环境、Redis环境以及Jmeter环境

### 1.MySQL环境准备

派单业务需要同时有两部分数据才能进行：首先要有乘客约车请求，然后要有向服务端表达过听单意愿的司机，因此我们要先在数据库中准备1000个乘客与1000个司机。注意相关的表也要对应的插入数据。

执行附录中SQL脚本:createBatchUser.sql。该SQL脚本中创建了4个Procedure，分别用来循环向相关的表中插入：乘客实体，司机实体，车辆实体，司机所属车辆的联系。后二者是由于后端有校验，没有注册过车辆的司机不能接单。

至此MySQL环境准备完毕。

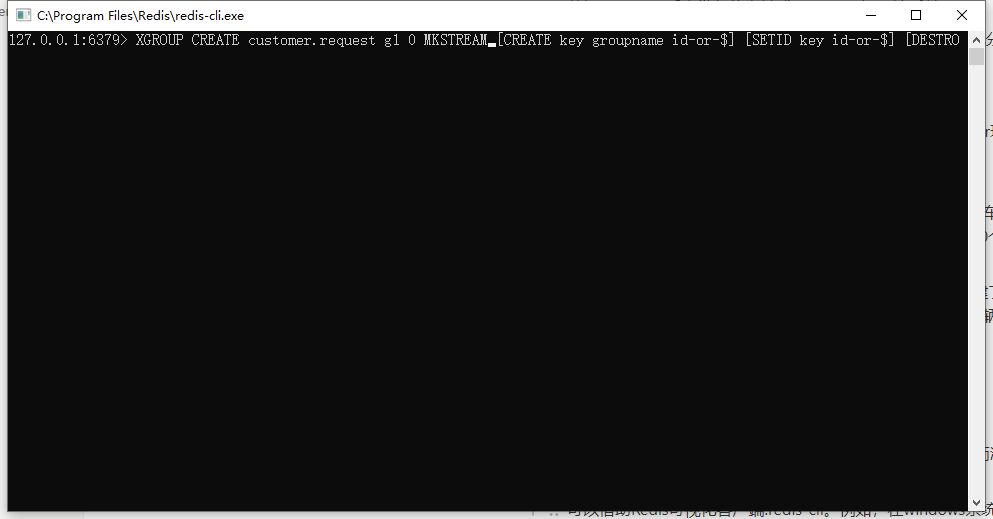
### 2.Redis环境准备

由于派单业务的实现需要借助Redis中的Stream消息队列，而消息队列的创建是在服务端部署上线前就在服务器上完成的。所以需要手动创建。

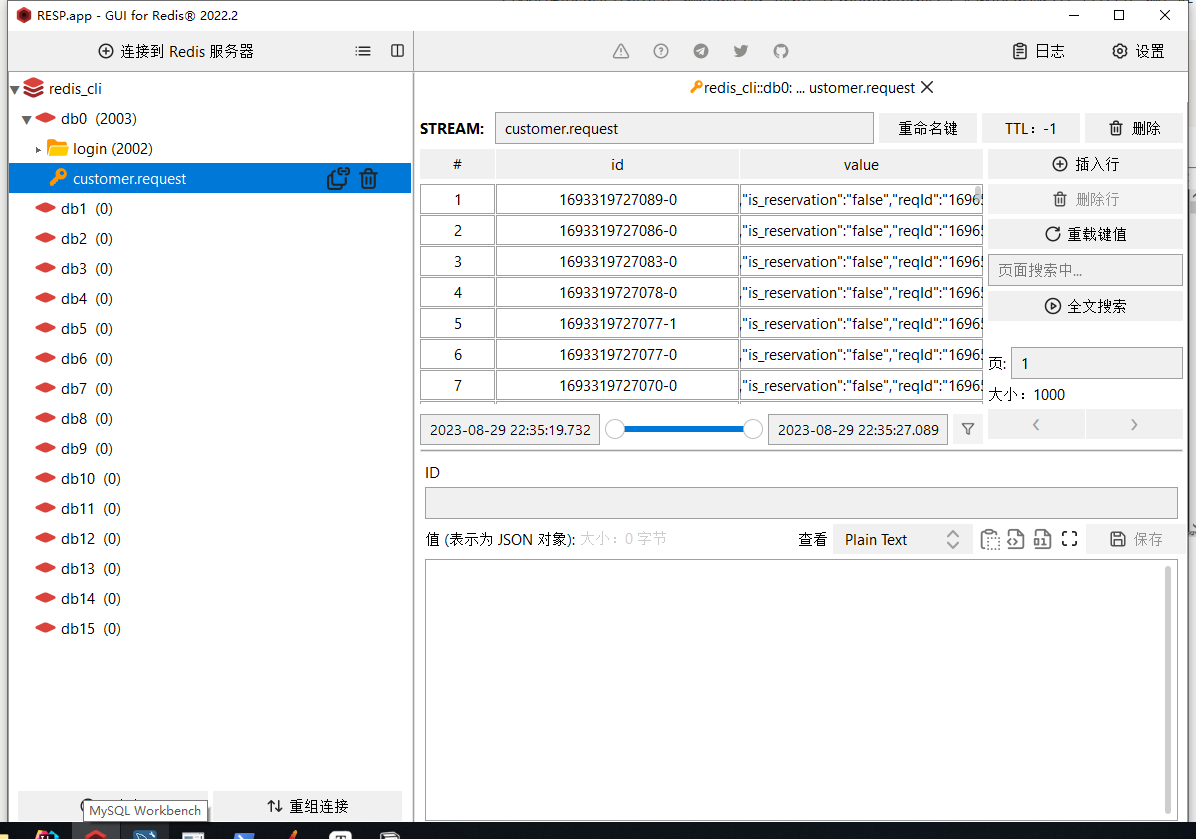
可以借助Redis可视化客户端:redis-cli。例如，在windows系统下，启动Redis服务，打开客户端，键入以下命令：

XGROUP CREATE customer.request g1 0 MKSTREAM

这里的消息队列名称，包括服务端的消费者组名称，都是写死的常量。



之后可以在Redis的图形化客户端RESP中看到这个消息队列



至此Redis的环境准备就绪。

### 3.Jmeter的环境准备

这里假设从零运行这个项目，以此为基础展示如何配置Jmeter数据源。

首先打开Jmeter，导入附录中的脚本：RideHailing-DispatchOrder-test.jmx。随后可以看到已经创建了4个线程组，这4个线程组均向服务器指定端口的指定路径发送Http请求，分别是:乘客登陆，司机登陆，乘客发起约车请求，司机发起开始听单。

由于后两个线程组发送的Http请求附带有用户个人的信息，也就是需要鉴权的接口，因此要在请求头中携带token，所以Jmeter环境的准备第一步就是获取Token。但获取方式的Token的方式仅有一种：即请求对应的登陆业务接口，然后才能拿到服务端生成的token。因此在此之前我们要配置首个数据源：用户登陆的数据源。

用户登陆时需要以下参数:

{

"phone":phone,//用户手机号

"password":password,//用户密码

"code":code,//用户验证码

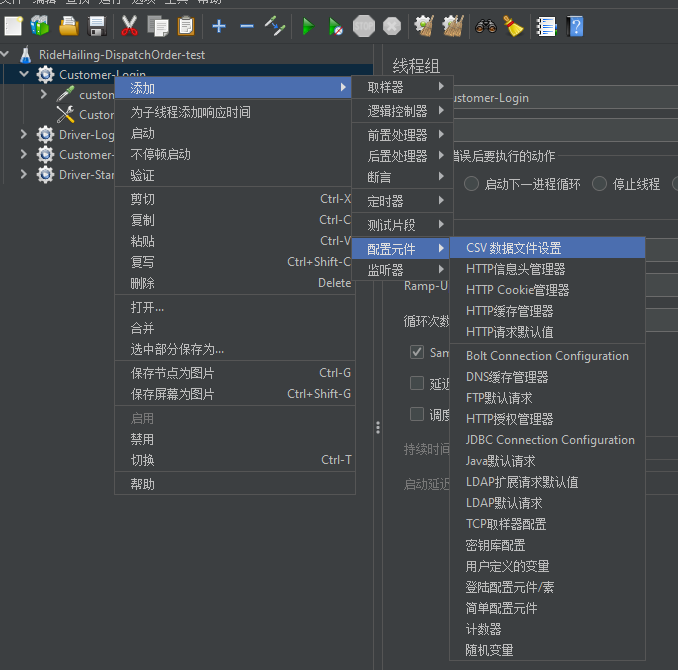
"role\_id":role\_id//用户类型,1为司机,2为乘客

}

由于服务端的逻辑是，登陆时只要提供密码或验证码中的至少一个即可，因此我们配置的数据源中仅选择提供密码，置空验证码。

因此最终的请求中，需要动态的获取三个来自数据源的数据：手机号，密码，用户类型。

对对应线程组添加配置元件：CSV数据文件设置。这个数据文件不一定要是.csv格式的，这里我选择了.txt的文本，也是可以的。



随后在变量名称中配置变量名，以逗号分隔，注意不要使用中文，没有对应编码，识别不出来的。

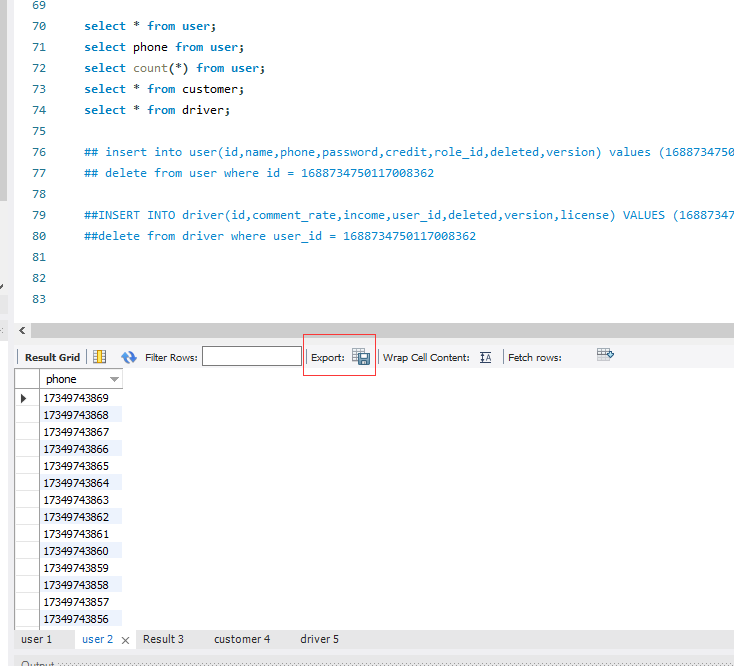
最后在文件路径处选择对应数据源文件。



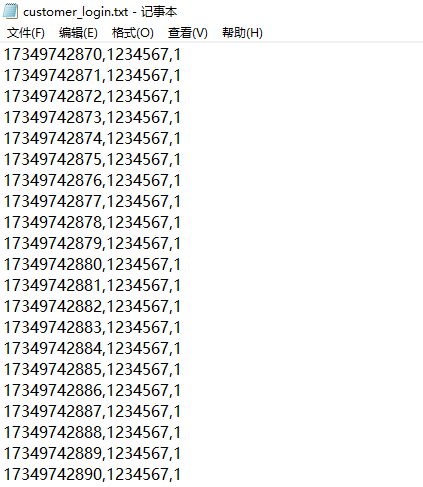
这个路径不一定适配所有电脑，最好在你本机创建一个专门存放jmeter数据源的文件夹，把需要的文件放进去。

那么这个文件从何来？我使用了MySQLWorkBench中的Export导出功能，将选择的数据导出为.txt文件。

例如，我们之前执行的SQL脚本：createBatchUser.sql 最终查询了用户的手机号，我们可以直接在这个查询结果的界面将数据导出。格式可以自己选择。

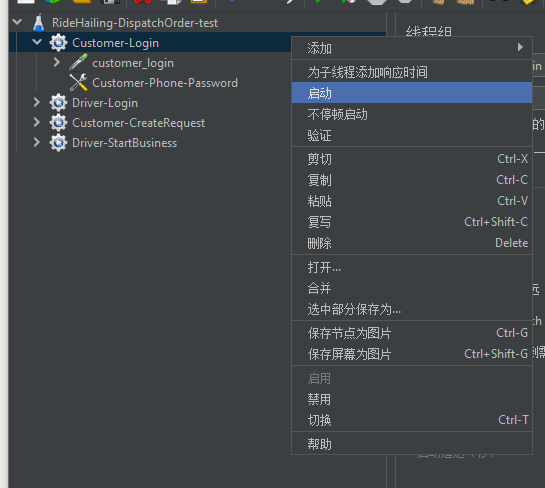


然后我们将导出文件的所有行适配成jmeter软件要求的格式:不同变量用逗号分隔，一组变量结束换行开始下一行变量，这里以customer\_login.txt 为例：

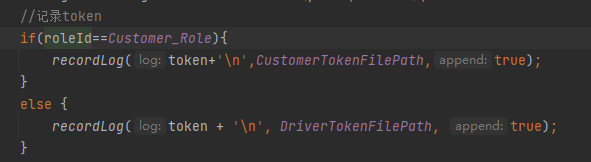


由于我为所有用户注册的密码的对应明文都为“1234567” ，因此密码可以写死，如果是乘客，那么用户类型字段role\_id应该为1。

如此配置两份登陆信息数据源（司机与乘客），然后准备使用jmeter批量请求登陆。



分别启动Customer-Login和Driver-Login 线程组，这将分别向服务器发送1000条登陆请求。随后token将存入Redis。服务端配置了将日志输出到指定文件的功能，因此我选择将token以日志的方式输出到某个文件中。



注意我输出的路径可能不适配你的计算机，因此最好将服务端输出路径的常量配置修改为你的对应路径。



（修改上图配置）

这样将获取登陆用户的token，作为请求头中authorization字段的数据源。

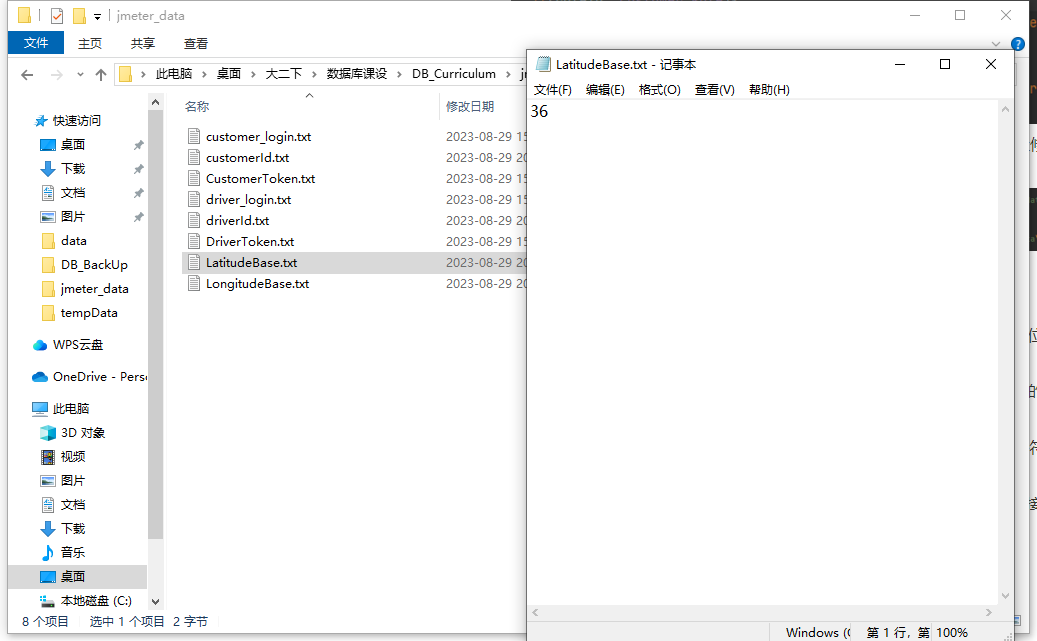
随后是发起约车请求和开始听单的请求。这里需要动态配置的字段为用户id，地理位置坐标（约车开始位置，目标位置，司机当前位置等），以及请求头中的token。

用户id方面，运行附录脚本：genDriverIdAndCustomerId.sql ，然后通过上文描述的MySQLWorkBench的Export功能导出对应数据源文件。

地理位置坐标方便，可以新建txt文件设置基准位置，然后在后方拼接jmeter随机字符串函数RandomString返回的随机字符串来实现随机均匀地理坐标。

例如我身处济南，就将济南的大致地理坐标：36，117设置为基准，在此基础上拼接随机字符串。

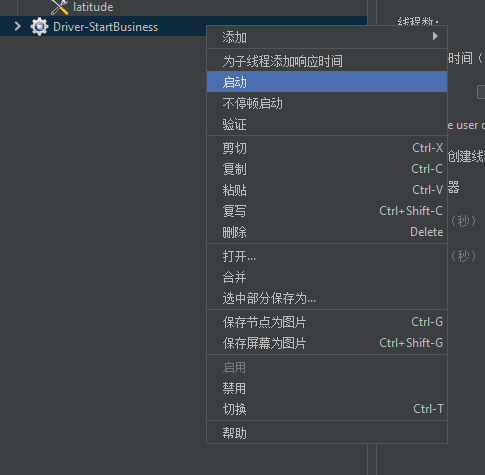




至此Jmeter的环境准备就绪。

## 2.压力测试流程以及观测压测结果

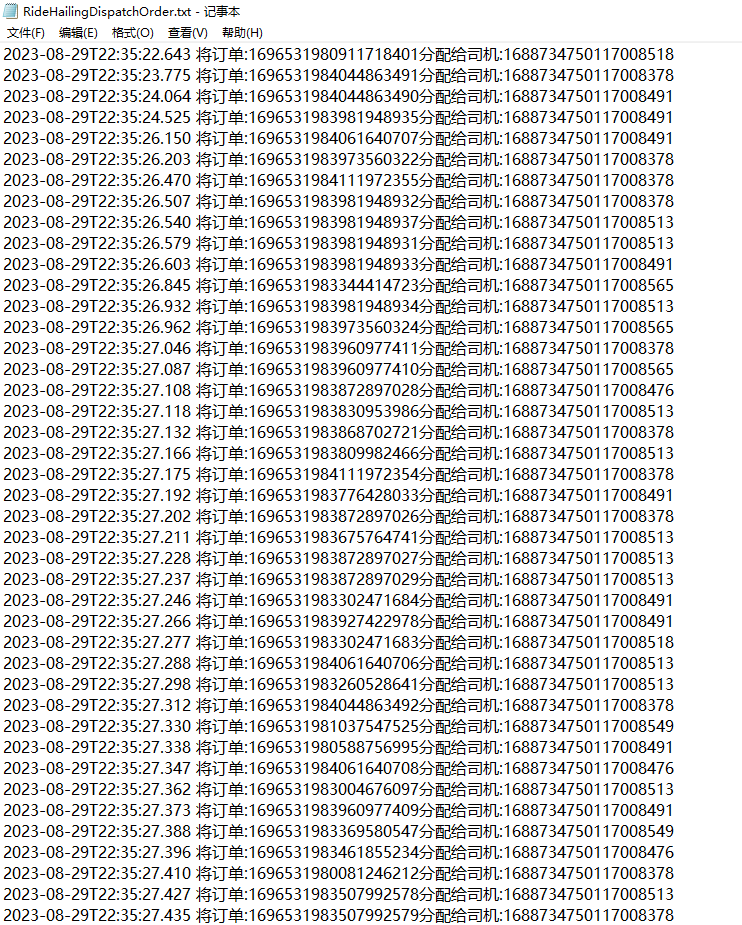
由于在环境准备中我们已经运行了两类用户登陆的线程组，因此这里不再需要了。



运行司机开始听单线程组，将所有司机列入服务端维护的空闲司机的数据结构中。

但只有空闲司机还不够，再运行乘客发起约车请求的线程组，请求约车。

然后观察C:\\\\Users\\\\Administrator\\\\RideHailingDispatchOrder.txt 文件，可以看到1000条派发订单的日志。



也可以观察Jmeter的汇总报告进行性能评估。

至此完成整个派发订单业务的压测。

# 六、总结

在大二下学习了《数据库系统》这门课程后，我掌握了一些设计、使用数据库的知识。并且认识到数据库在项目中的重要作用。因此暑假便尝试开发一个基于多数据源的“网约车运营管理系统”。其中使用到了MySQL和Redis，一个驻留在磁盘，另一个驻留在内存；一个是典型的关系型数据库，而另一个则是NoSQL。

开发过程中我也遇到过不少问题。例如一开始将订单表命名为order。随后在编写sql语句对该表进行查询时(select \* from order ...)发现order是MySQL对查询结果进行排序的保留关键字。这是由于开发经验不足导致的问题。

又如，在使用redis消息队列来进行约车请求派发时，需要进行消息的确认。而这个确认时间至关重要。我一开始将负责消息确认逻辑的代码放置在分配订单之后，这就导致了一个线程并发的问题：即两个线程同时获取消息队列中的数据，第一个线程获取完消息后，没有立即确认，而是去分配订单，这样第二个线程尝试获取消息时，获取的就是第一个线程还未确认的消息，这样可能就将一个约车请求发给两个不同司机。通过debug我发现这个问题，然后将消息确认机制提前到派发约车请求前，解决了这一问题。

开发时，我遇到的不仅仅是数据库的问题，还有很多业务上、配置上、框架特性上的问题。每次遇到一些疑难杂症都会花费我巨量的时间去调试。但我认为这是值得的，因为在本次课设后，我对于学习新知识、使用新知识以及调试bug变得越来越有信心。就如我本次采用的客户端框架uniapp和内存数据库redis都是我曾经从来没接触过的技术，而webSocket通信也是在我做业务逻辑时才找到的技术。这无疑带来了很大的学习成本，但我认为这就是一名合格的程序员应该具备的能力：自主学习并将技术用于解决现实生活中的问题上。