迭代一计划书

1. 阶段目标定义
    1.1. 明确需求
    1.2. 架构设计
    1.3. 接口设计
    1.4. 技术选型
2. 需求定义与分析
    2.1. 核心业务需求
    乘客
        1.发出请求 `/request/customerRequest` POST
        2.撤销请求 `/request/cancelRequest` POST
    司机
        1.开始听单 `/driver/startBusiness` POST
        2.结束听单 `/driver/endBusiness` GET
        3.确认订单 `/driver/confirmOrder` GET
3. 设计决策
    3.1. 系统架构
    3.2. 解决方案
        3.2.1. 开发技术选型
        3.2.2. 团队协作工具
        3.2.3. 测试工具
    3.3. 技术难点
        3.3.1. 司机状态维护
        3.3.2. 算法模块
        3.3.3. Request-Response,Blocking,Polling 和 WebSocket
        3.3.4. 路线规划
    3.4. 客户端界面
        司乘登录
        乘客约车
        司机听单
        司机接单
        司机订单列表
        乘客评价订单
    3.5. 类图

1. 阶段目标定义

1.1. 明确需求

需求调研，背景调研，了解客户想要的软件需要对实际场景提供哪些具体的解决方案

1.2. 架构设计

明确系统各模块功能，提供项目架构设计图、业务流程图

1.3. 接口设计

设计各功能接口，确定数据结构、参数字段、通信协议

1.4. 技术选型

确定技术栈、软件、SDK等必要的开发工具

2. 需求定义与分析

这里直接以接口文档的形式呈现。

2.1. 核心业务需求

分三个视角看

乘客

1.发出请求 `/request/customerRequest` POST

乘客的请求应该附带以下信息

```
{
  "id": number, //这里是用户的id，即user_id
  "startX": double,//39.90,乘客上车点的经纬度
  "startY": double,//116.39
  "start_name":string,
  "desX": double,//39.90,乘客下车点的经纬度
  "desY": double,//116.39
  "des_name": string,
  "request_time": time,//请求发起时间   年月日  时分秒
  "is_reservation":boolean,//是否预约,
  "appointment_time":time,//预约起始时间 年月日  时分秒
  "is_instead":boolean,//是否代叫,
  "customer_phone":string,//被代叫人手机号
  "priority": int //优先级
}
```

服务器拿到请求后:

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，一条消息应该有以下的参数

```
{
  reqId,//请求id
  is_reservation,//是否预约
  is_instead//是否代叫
}
```

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

执行lua脚本前应该先创建消息队列

```
XGROUP CREATE customer.request g1 0 MKSTREAM
```

4.向服务端返回生成请求的id(防止精度丢失返回字符串)

2.撤销请求 `/request/cancelRequest` POST

乘客撤销的请求应该带有以下参数

```
{
  id,//乘客的用户id
  requestList:[...]//指定删除的请求Id列表
}
```

服务器拿到撤销请求后:

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

a. 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

b. 如果不是，继续执行

2. 逻辑删除该请求
3. 查询该请求是否是特殊请求，如果是，级联删除特殊请求
4. 查询该请求是否有对应生成的订单

3.1 如果没有对应订单

3.1.1 逻辑删除请求

3.1.2 通知乘客请求已经撤销(响应)

3.2 如果有对应订单

3.2.1 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

3.2.2 逻辑删除对应订单，同时维护司机订单driverOrder哈希表

3.2.3 检查司机是否在线

3.2.3.1 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

3.2.3.2 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

3.2.4 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

3.2.5 ws通知订单对应的司机订单已经被取消

3.2.6 通知乘客请求已经撤销(响应)

司机

1.开始听单 `/driver/startBusiness` POST

司机听单请求应该带有以下参数

```
{
  id,//用户id
  curX,//当前位置经度
  curY//当前位置纬度
}
```

ps: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。空闲司机列表中的每一个元素应该有以下参数

```
{
  id,//用户id
  curX, //当前位置经度
  curY, //当前位置纬度
}
```

服务器拿到司机听单的请求后:

1. 伪造请求校验:是否是司机
2. 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
3. 查询该司机是否已经接单，如果已经开始接单，响应客户端
4. 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
5. 将该司机添加至距离最近的一个聚簇中

6. 响应客户端已经开始听单

2.结束听单 `/driver/endBusiness` GET

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

司机结束听单应该带有以下参数

```
{
  id//用户id
}
```

服务器拿到司机结束听单的请求后:

1. 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)
2. 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
3. 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
4. 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

3.确认订单 `/driver/confirmOrder` GET

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

1. ws通知司机分配给他的订单消息，询问他是否接单

那么我们现在假设司机的客户端已经拿到了订单消息，他确认订单的请求应该带有以下参数

```
{
  id,//司机的用户id
  requestId,//为其分配的请求id
  accept,//是否接单
}
```

服务器接到请求后:

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
2. 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。

   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

   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

   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

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

3. 如果司机接受订单

4.1 如果他接受的是预约单

保持他现有的状态

4.2 如果他接受的是实时单

4.2.1 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

4.2.2 司机状态空闲变忙碌，把司机从聚簇中移除

4.3 更新队列流水，如果队列流水达到阈值，更新聚簇

4.4 更改请求信息的某些字段，比如是否接单以及响应时间

4.5 生成订单，交由driverOrder哈希表维护

4.6 ws响应乘客他的订单已经被某个司机接受

```
//可以多不能少
{
  reqId,//订单id
  licensePlateNumber,//车牌号
  carType,//车的类型
  carColor,//车颜色
  carBrand,//车品牌
  driverUserId,//司机的用户id？
  driverName,//司机用户名
  driverPhone,//司机电话
  driverCredit//我猜是信用值
  cancelInfo//司机取消当前订单
}
```

PS:这里的一个小障碍是司机客户端缓存的订单信息可能有多条，比如一条实时单和多条预约单，到时候要做JSON和object之间的转换

4.提交订单 `/driver/commitOrder` GET

主要用于司机在将乘客送至目的地后手动提交订单，生成结束时间，结束位置(可能后续要统计和请求中的位置是否(大致)一致)，距离(这个有一些库可以很容易计算），价格(这个不好弄，需要有一个计算价格的策略。

一个司机提交订单的请求应该带有以下参数

```
{
  id,//用户id
  reqId,//对应请求的id
  endX,//结束位置经度
  endY,//结束位置纬度
}
```

ps:不需要带endTime,一个订单结束的时间点应该是服务器处理的时间点,而不是司机请求的时间点。也不需要带price，订单计价应该由服务端完成。

服务器接到请求后:

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
    a. 网约车订单计价:起步价+时间费+里程费+超里程费+夜间费
    b. 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
    c. 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/公里，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
4. 更改数据库中对应订单的对应信息项
5. 将司机状态修改为空闲，清空他正在跑的单子
6. 将司机加入聚簇中
7. 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
8. 响应司机他已经提交了订单，以及将得到的支付的价格

5.删除订单 `/driver/deleteOrder` POST

司机有能力指定删除他被分配到且本人同意的订单

```
{
  id,//用户的id
  requestList:[...],//指定删除的请求
}
```

ps:JS有精度丢失，返回requestList时里面全是字符串，我后端转

服务器接到请求后:

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

6.接到乘客 `/driver/` `receivePassenger` GET

司机在接到乘客，完成手机号校验后应该向服务器确认他已经接到了乘客

```
{
  id,//司机的用户id
  reqId,//接的是哪一单的乘客
}
```

服务器接到请求后:

1. 检查司机是否在线，如果在线将司机的状态改为忙碌(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

7.抵达出发点 `/driver/arriveStart` GET

司机在抵达出发点后可以通知乘客他已经抵达了出发点

请求应该带有以下参数

```
{
  id,//司机用户id
  reqId,//对应请求id
  curX,//当前经度
  curY,//当前纬度
}
```

服务器接到请求后:

1. 伪造请求校验:是否为司机
2. 检查是否存在对应的订单
3. 检查当前是否距离请求中的出发点较近，如果太远应该响应司机距离出发点还有距离
4. ws通知乘客司机已经抵达出发点
5. 响应司机请求成功

服务器

1.分配订单

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

1. 在redis中创建消费者组作为消息队列
2. 开启独立线程，专门用来执行分配订单的工作
3. 获取消息队列中的消息
4. 转换为请求消息
5. 确认消息
6. 查询请求是否还存在，因为可能乘客已经取消了请求
    a. 如果请求还存在，继续执行
    b. 否则，进入下一轮消息处理
7. 为该订单分配司机
    a. 查询当前聚簇集合是否存在，不存在则更新聚簇
    b. 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
    c. 获取该聚簇的首个元素
    d. 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
    e. 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
    f. 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
    g. 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
8. 异常处理

2.2. 非核心业务需求

1.回显相关订单 `/indent/showOrder` GET（已经完成的订单）

司乘个人中心回显历史订单，返回的参数中每一项都应该带有

```
{
  indentId,//订单id(前端可以不用，但是我要带上这个信息)
  reqId,//请求id(前端可以不用，但我要带上这个信息)
  responseTime,//乘客下单时间 （  responseTime这个应该是乘客的下单时间，order的生成时间才是司机的接单时间，但是现在order里面只有开始时间和结束时间，如果是
  receiveTime,//司机接单时间
  startTime,//出发时间
  endTime，//结束时间
  appointTime,//可能为空，预约时间（乘客的预约单只显示预约时间/出发时间）
  startX,//出发经度
  startY,//出发纬度
  endX,//到达经度
  endY,//到达纬度
  startName,//出发地点名称
  endName,//到达地点名称
  price//价格
  driverName;//司机
  customerName;//乘客
  number;//车牌号
  driverPhone;
  customerPhone;
}
```

请求的参数中应该带有

```
{
  id,//司乘用户id
  role//用户类型(1乘客2司机)
}
```

服务器拿到请求后:

1. 新建列表，存放所有待返回的DTO
2. 根据是司机还是乘客分类讨论
    a. 如果是乘客，根据id查询所有被接受的请求，再根据每个请求的id查询出所有订单

       这里注意等价，被接受可以推得有订单，有订单可以推得被接受，这两个东西创建和销毁是同时的
    b. 如果是司机，根据id查询所有订单，再根据每个订单的request_id查询所有对应请求
3. 将订单与请求中的信息拼起来，新建对象置入列表
4. 返回列表

2.司机注册车辆 `/vehicle/register` POST

请求的参数中应该带有

```
{
  id,//司机用户id
  number,//车牌号
  color,//车辆颜色
  type,//车辆对应驾照类型
  brand//车辆品牌
}
```

服务器拿到请求后:

1. 伪造请求校验:是否为司机
2. 检查该账号是否已经注册过车辆
3. 检查该车辆是否已被注册过
4. 检查司机的驾照类型和车辆类型是否符合
5. 检查注册车型是否是合法网约车类型

6. 检查车牌号是否已经被注册过
7. 为车辆注册
8. 注册属主关系
9. 响应司机注册成功

3.乘客评价订单 `/comment/commentOrder` POST

司机提交订单后,ws向对应乘客发送待评价的信息,乘客客户端弹出评价表单，随后提交评价(当然也可能不提交，那么就没有这个订单对应的评价)

请求的参数中应该带有

```
{
  id,//乘客的用户id
  reqId,//评价订单对应的请求id
  comment,//x星好评
  content//评价内容
}
```

服务器拿到请求后:

1. 伪造请求校验
    a. 订单是否存在
    b. comment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

4.回显相关订单 删除某个订单 `/indent/deleteOrder` POST

前端请求参数

```
{
"indentId":indentId
"role"://1为乘客，2为司机
}
```

服务器拿到请求后:

1. 更改数据库对应indent信息
2. 响应更改成功

5.司机驾车行驶路线规划 GET

其实这个确实也是后端做的，只不过不是我们的后端，而是高德地图官方提供的接口

```
URL：https://restapi.amap.com/v5/direction/driving?parameters
```

请求应该带有以下参数

```
{
  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
}
```

6.回显评价 `/comment` / `showComment` GET

进入订单评价页后回显评价(如果没有回显为空)

请求应该带有以下参数

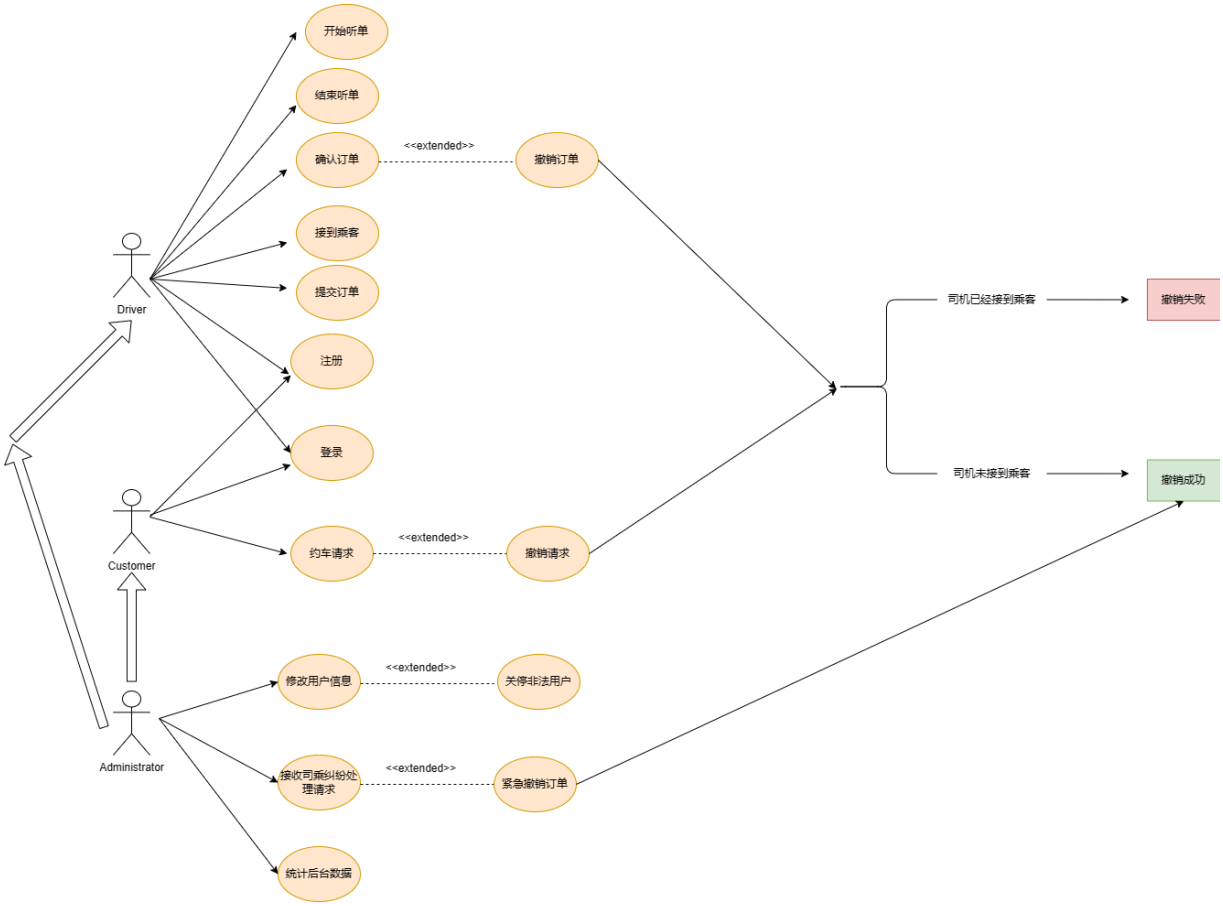
```
{
  id:id,//用户id
  reqId:reqId//订单对应的请求id
}
```

服务器拿到请求后

1. 伪造请求校验:检验该请求对应的订单是否已经完成
2. 查询对应订单的评价
3. 响应该评价(如果没有评价响应为空)

   返回的结果中应带有以下参数

用例图如下：



3. 设计决策

3.1. 系统架构

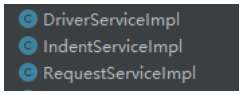
1. 服务端设计

   主流的`SpringMVC`解决方案，即Controller→Service→DAO→Entity的实现方式。

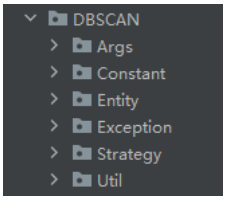
   同时，作为一个前后端分离结构的项目，也少不了作为鉴权的拦截器`Interceptor`模块。

   最后是协助开发、测试的`util`模块

   核心业务逻辑主要集中在三个service及其对应实现类中，分别是负责乘客及其约车请求的RequestService、负责司机及其相关订单的DrvierService、负责订单智能派发的IndentService中。

   最终，还有作为订单智能派发行为的大脑的algorithm模块中的`DBSCAN`算法。

   包结构如下：

   1. 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。

2. 客户端设计

   2.1 辅助开发UI界面的组件`component`模块

   2.2 负责UI界面开发的界面`pages`模块

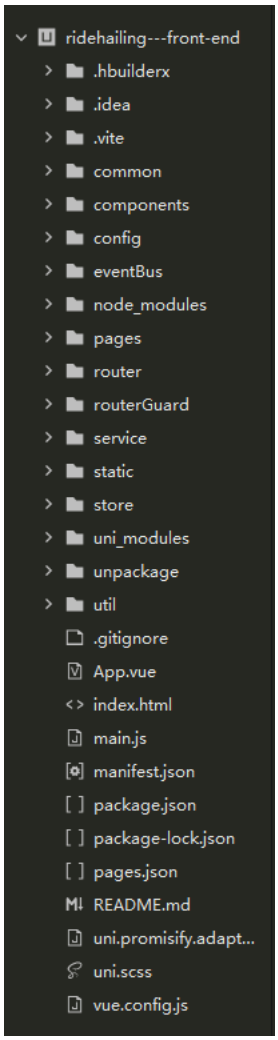
   2.3 负责页面路由的路由`pages.json`文件（模块）

   2.4 负责鉴权的路由守卫`routerGuard`模块

   2.5 负责本地数据缓存的持久化`store`模块

   2.6 负责网络请求、ws连接的服务`service`模块

   2.7 负责协助开发的工具集合`util`模块



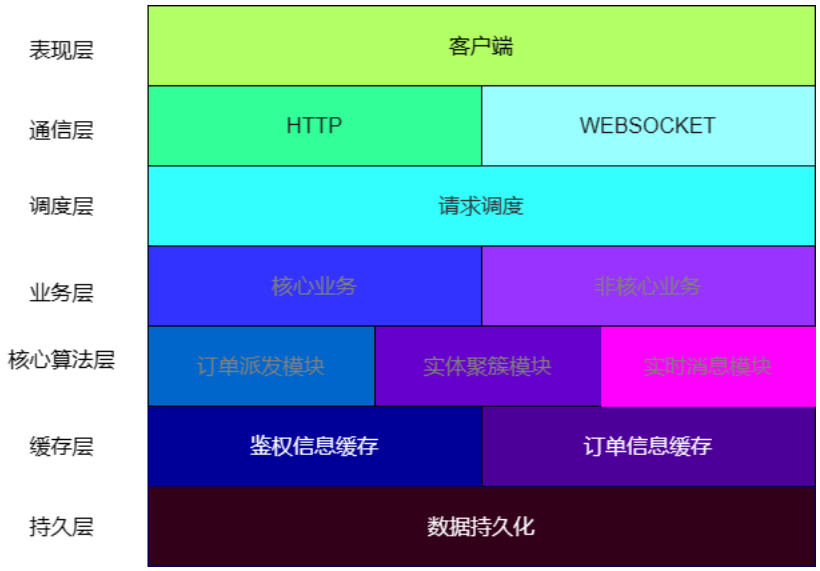
3. 数据库设计

这里主要阐述数据库概念与逻辑设计，各表中字段均为了满足必要的业务需求设计。

1. 为了区分不同的用户类型，需要表`role`
2. 为了持久化用户的基本信息，需要表`user`
    a. 为了持久化乘客用户的信息，需要表`customer`
    b. 为了持久化司机用户的信息，需要表`driver`
3. 为了持久化用户的约车请求的基本信息，需要表`request`
    a. 为了持久化用户预订约车的信息，需要表`reservation`
    b. 为了持久化用户代叫约车的信息，需要表`instead`
4. 为了持久化形成的订单信息，需要表`indent`
5. 为了持久化用户对订单的评价，需要表`comment`
6. 为了持久化车辆的基本信息，需要表`vehicle`
7. 为了持久化车辆从属的联系，需要表`belong`

其中2、3与其附属的a、b形成了泛化\特化的关系

架构图如下：



3.2. 解决方案

3.2.1. 开发技术选型

1. 服务端技术栈，包括：
    a. **实现语言**：Java\Lua\Kotlin
    b. **集成开发工具**：Jetbrains IDEA
    c. **技术框架**：Spring + SpringBoot
    d. **常用开发组件**：
        i. **模板引擎**：
            1. lombok实体注解
            2. mybatis-plus code generator代码生成
            3. velocity模板引擎
        ii. **启动依赖**：
            1. springframe work boot
            2. mybatis plus start boot
        iii. **常用工具**
            1. commons-pool2池化工具
            2. hu-tool数据处理工具
        iv. **数据缓存与持久化依赖**
            1. redis内存数据库
            2. MySQL磁盘数据库
            3. lettuce redis客户端
            4. mybatis-plus数据库框架
            5. druid数据库连接
            6. redission redis分布式锁
    e. **是否自主研发组件/算法**（如果需要，确定组件/算法的功能、上下游，提供接口文档）：是

i. 基于ThreadLocal的**鉴权模块**，模板Spring Security

ii. 基于DBSCAN的**聚类算法模块**

iii. 基于Redis的**消息队列组件**

f. **编译打包工具**：IDEA

2. `客户端技术栈`，包括：

a. **实现语言**：H5\C3\JS\TS

b. **集成开发工具**：Visual Studio Code\Hbuilder\WebStorm

c. **技术框架**：uniapp

d. **常用开发组件**：uniapp SDK + AMap SDK

e. **是否自主研发组件/算法**（如果需要，确定组件/算法的功能、上下游，提供接口文档）：是

i. **WebSocket通信协议模块**

f. **编译打包工具**：Hbulider

3. `数据库技术栈`，包括：

a. **磁盘数据库**：MYSQL

b. **内存数据库**：Redis

c. **ORM框架**：MyBatis-Plus\Lettuce

d. **可视化管理工具**：MySQL Workbench\RESP

3.2.2. 团队协作工具

1. `代码托管工具`：git + github private，现已开源

```
# 服务端
https://github.com/Liyanhao1209/DataBaseCurriculumBackEnd.git

# 客户端
https://github.com/Liyanhao1209/DataBaseCurricumlumFrontEnd.git
```

2. `在线文档管理工具`：Notion，这个是不会开源的

3.2.3. 测试工具

1. `单元测试工具`：JUnit
2. `接口测试工具`：Postman
3. `压力性能测试工具`：Jmeter

3.3. 技术难点

3.3.1. 司机状态维护

总共需要实时地维护司机的四个状态

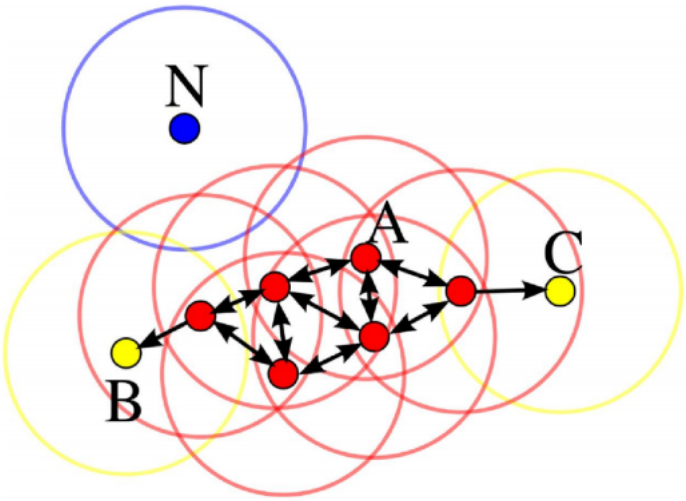
1. 需要获取司机实时位置与是否空闲（由是否接受了实时单和是否接到乘客共同决定），以便维护空闲司机的数据结构：`DriverStatus`
2. 需要获取司机拒绝接受的订单，以免之后重复分配到该司机：`driverReject`
3. 需要获取司机正在跑的订单，以免司机或乘客删除了已经接到乘客的订单：`driverRunning`
4. 需要获取司机确认接受的订单，用来判断一个司机在任意时刻的空闲/忙碌状态：`driverOrder`

3.3.2. 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(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。

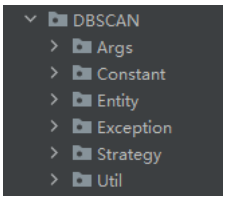
因此我们不能只是初始化一个容器并将空闲司机拉进去就完成任务，我们需要制定一个能够动态更新整个聚簇集合的解决方案。

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。

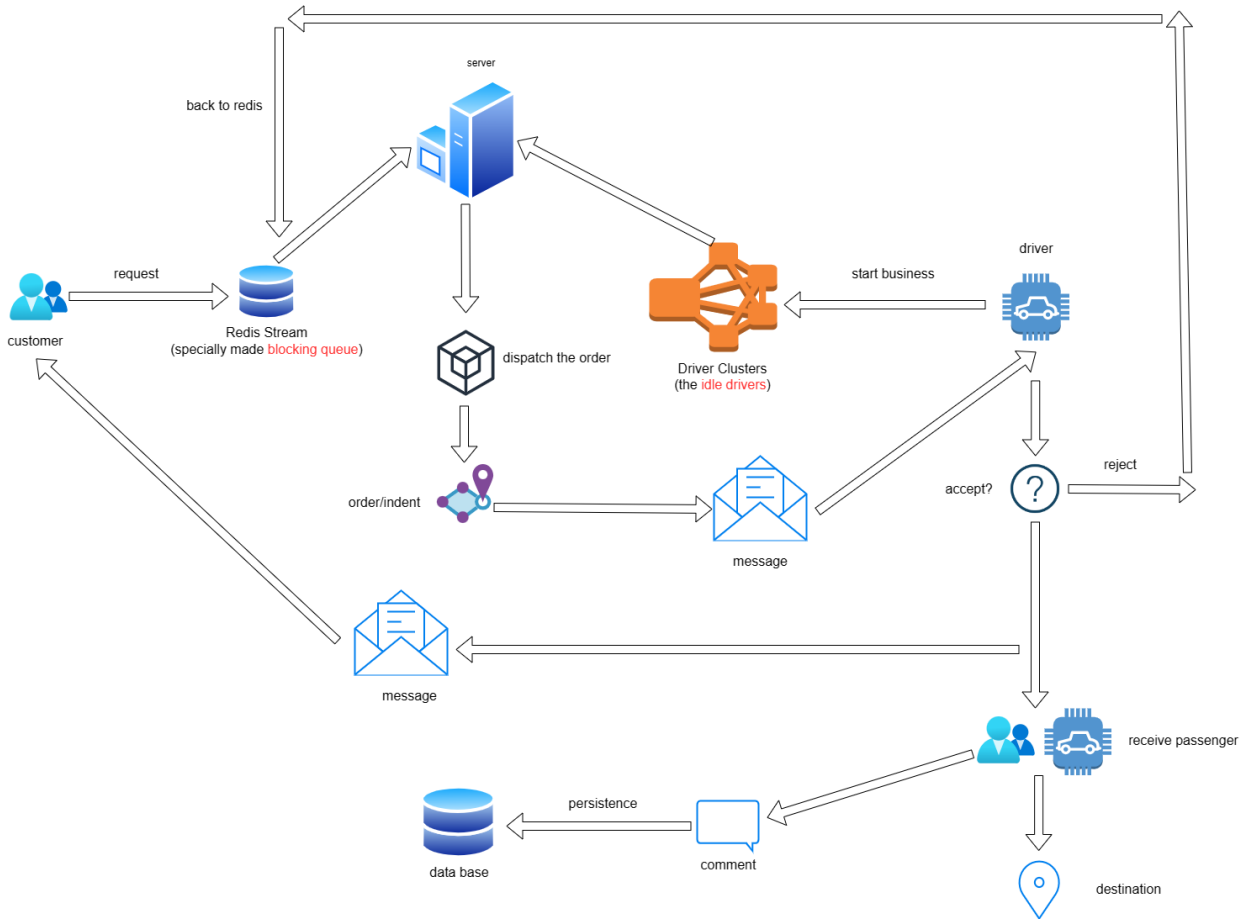
本项目中使用的DBSCAN算法是我自主开发的，没有引入任何依赖。



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。

业务流程图如下：



3.3.3. Request-Response,Blocking,Polling 和 WebSocket

上文（图）中对于订单分配部分的流程仅仅只是个设想，想要落地还是有一定困难的。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。

3.3.4. 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)

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=${key}

```
{
    "status": "1",
    "info": "OK",
    "infocode": "10000",
    "count": "1",
    "route": {
        "origin": "116.481028,39.989643",
        "destination": "116.465302,40.004717",
        "taxi_cost": "13",
        "paths": [
            {
                "distance": "3262",
                "duration": "673",
                "strategy": "速度最快",
                "tolls": "0",
                "toll_distance": "0",
                "steps": [
                    {
                        "instruction": "向北行驶109米右转",
                        "orientation": "北",
                        "distance": "109",
                        "tolls": "0",
                        "toll_distance": "0",
                        "toll_road": [],
                        "duration": "44",
                        "polyline": "116.480891,39.98937;116.480891,39.98937;116.480553,39.989606;116.480548,39.98966;116.480596,39.989708;116.481095,39.990051",
                        "action": "右转",
                        "assistant_action": [],
                        "tmcs": [
                            {
                                "lcode": [],
                                "distance": "1",
                                "status": "未知",
                                "polyline": "116.480891,39.98937;116.480891,39.98937"
                            },
                            {
                                "lcode": [],
                                "distance": "51",
                                "status": "未知",
                                "polyline": "116.480891,39.98937;116.480553,39.989606;116.480548,39.98966;116.480596,39.989708"
                            },
                            {
                                "lcode": [],
                                "distance": "57",
                                "status": "未知",
                                "polyline": "116.480596,39.989708;116.481095,39.990051"
                            }
                        ],
                        "cities": [
                            {
                                "name": "北京城区",
                                "citycode": "010",
                                "adcode": "110100",
                                "districts": [
                                    {
                                        "name": "朝阳区",
                                        "adcode": "110105"
                                    }
                                ]
                            }
                        ]
                    },
                    {
                        "instruction": "向东南行驶36米右转",
                        "orientation": "东南",
                        "distance": "36",
                        "tolls": "0",
                        "toll_distance": "0",
                        "toll_road": [],
                        "duration": "20",
                        "polyline": "116.481095,39.990051;116.481111,39.990003;116.481347,39.989794",
                        "action": "右转",
                        "assistant_action": []
```


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。

3.4. 客户端界面

司乘登录

乘客约车


中华人民共和国
最高人民法院
正义路
东交民巷
红都商务会馆
东交民巷
北京首都宾馆
23号院
北京华风宾馆
入口
8号线
良苑烤鸭店
悠惠万
北京市前门
外国语学校
前门小学
西打磨厂街
45号四合院
肖公庙
利群烤鸭店
湖北黄安会馆
京维斯旅馆
为水
西兴隆街
汉庭酒店
湖北黄冈会馆
腾讯地图
上车点
选择上车点
下车点
你要去哪儿
预约
代叫
开始约车
当前约车列表

司机听单


智慧园
大山坡小广场
城建大厦
山东大学软件园校区
舜华路
科创金融公园
博晶大厦
舜泰北路
舜泰路
舜泰广场
地质科学研究院东院区
山东省打击盗窃电力网通电信设施犯罪联合办公室
万科金域国际天泰家园
经十路辅路
华冠路
金域中心
经十路
工友之家
济南市玉顶山公墓
开始听单
已经接单

司机接单


13:38
智慧园
大山坡小广场
城建大厦
山东大学软件园校区
已为您接到订单
距您7.2591公里
起点：山东大学(中心校区)
终点：山东大学(软件园校区)
本次行程预计7.2137公里
预约时间：2023-09-02 13:35:30
确认接单
取消接单
工友之家
济南市玉顶山公墓
取消听单
已经接单

司机订单列表


13:40
订单0
预约单
起点: 山东大学(中心校区)
终点: 山东大学(软件园校区)
本次里程长度: 7.2137公里
预约时间: 2023-09-02 13:35:30
接到乘客
取消订单
联系乘客
取消所有订单
关闭
开始为您导航

乘客评价订单


评分
评论
司机:driver1
非常好!
提交评价
提交成功

3.5. 类图

数据库导出的ER图如下：