用户文档 (https://md.byr.moe/yxfPRxL4Sea-1yiot-KlCA)

旅行模拟查询系统开发文档

一、需求分析

问题描述

城市之间有三种交通工具（汽车、火车和飞机）相连，某旅客于某一时刻向系统提出旅行要求，系统根据该旅客的要求为其设计一条旅行线路并输出；系统能查询当前时刻旅客所处的地点和状态（停留城市/所在交通工具）。

功能需求

- 城市总数不少于10个
- 建立汽车、火车和飞机的时刻表（航班表）
  - 有沿途到站及票价信息
  - 不能太简单（不能总只是1班车次相连）
- 旅客的要求包括：
  - 起点
  - 终点
  - 途经某些城市和旅行策略
    旅行策略有：
    - 最少费用策略：无时间限制，费用最少即可
    - 最少时间策略：无费用限制，时间最少即可
    - 限时最少费用策略：在规定的时间内所需费用最省
- 旅行模拟查询系统以时间为轴向前推移，每10秒左右向前推进1个小时(非查询状态的请求不计时)
- 不考虑城市内换乘交通工具所需时间
- 系统时间精确到小时
- 建立日志文件，对旅客状态变化和键入等信息进行记录
- 某旅客在旅行途中可更改旅行计划，系统应做相应的操作
- 用图形绘制地图，并在地图上反映出旅客的旅行过程。

需求分析

- 设计最短路径的算法及存储其需要信息的存储：本设计中最短路径的算法利用迪杰斯特拉算法，存储方法为便于查询的时刻表存储和便于计算的邻接矩阵。
- 该程序所做的工作模拟旅游交通查询，为旅客提供种最优决策的交通查询。此程序规定：
  - 在程序中输入城市名称时，需输入一个字符串类型；
  - 输入运输工具及其编号时，需输入一个字符串类型；
  - 输入列车的费用时，需输入一个浮点数据；
  - 输入列车开始时间和到达时间时，均需分别输入年、月、日、时4个整型数据；
  - 在选择策略时，应输入与所选功能对应的一个整型数据。
- 程序的输出信息主要是：
  - 最快需要多少时间才能到达
  - 最少需要多少费用才能到达
  - 输出在途中经过的城市名称
- 程序的功能包括

    - 火车信息查询
    - 最短路径查询
    - 火车信息编辑
    - 读入修改信息
    - 查看火车信息
    - 查看城市信息
    - 提供三种最优决策
        - 最快到达
        - 最省钱到达
        - 最短路程到达
- 模拟旅游交通查询系统，能实现现实生活中对火车以及旅客的各种需求做出相应的程序；能实现选择功能。

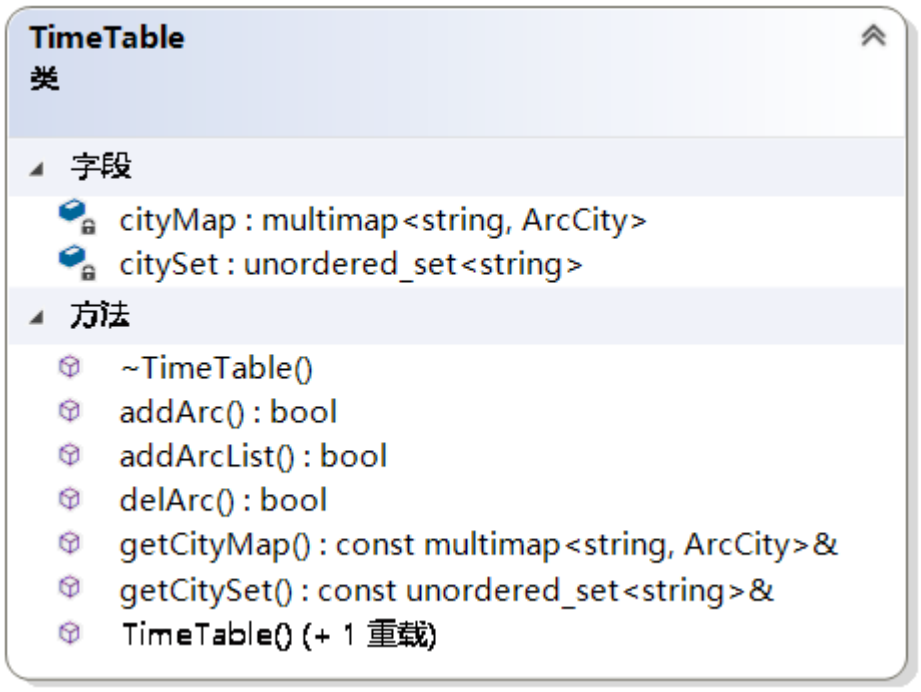
二、概要设计

软件开发环境

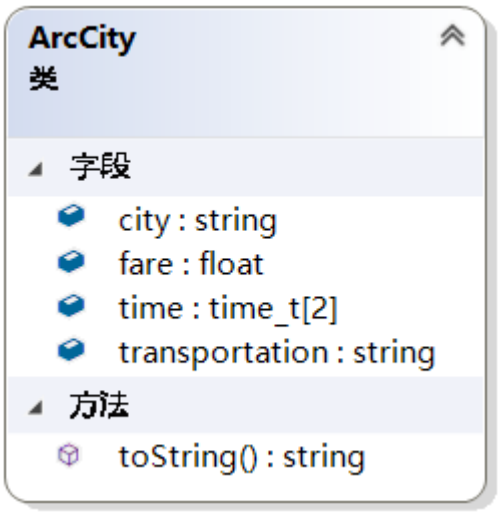
1. **WSL**
    - Windows 10 1809
    - Visual Studio Community 2019 16.1.1
    - Ubuntu 18.04
    - cmake version 3.10.2
    - g++ (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0
    - GNU Make 4.1
    - libboost 1.65.1
    - libssl 1.1
    - Chrome 74.0.3729.157
2. **Docker**
    - Ubuntu 18.04
    - cmake version 3.10.2
    - g++ (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0
    - GNU Make 4.1
    - libboost 1.65.1
    - libssl 1.1
    - Chrome 74.0.3729.157

数据结构设计

1. 汽车、火车和飞机的时刻表 **class TimeTable**

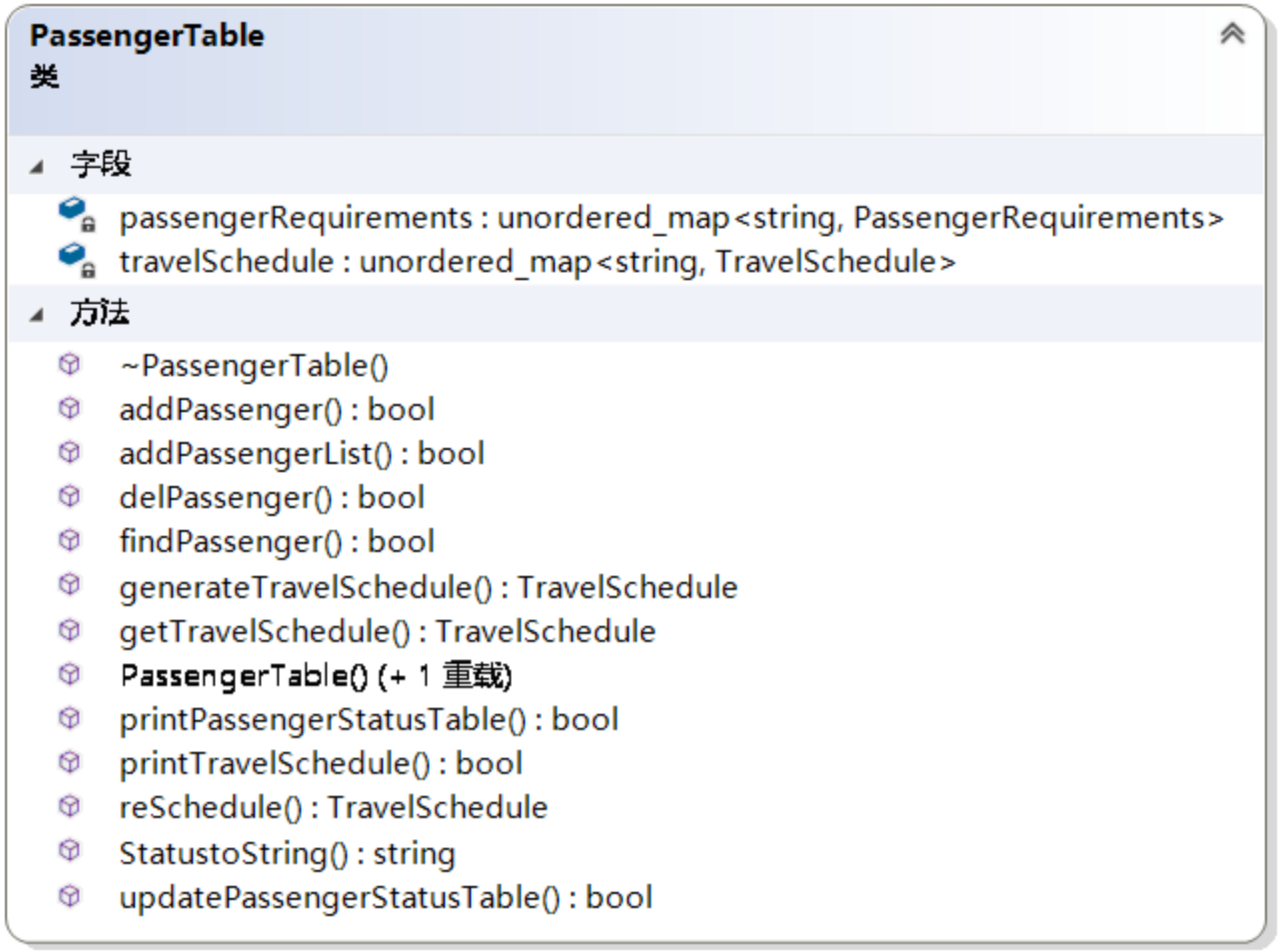


    1. 城市集合 CitySet: unordered_set<string>
    2. 城市地图 CityMap: multimap<string, ArcCity>
        1. **class ArcCity**

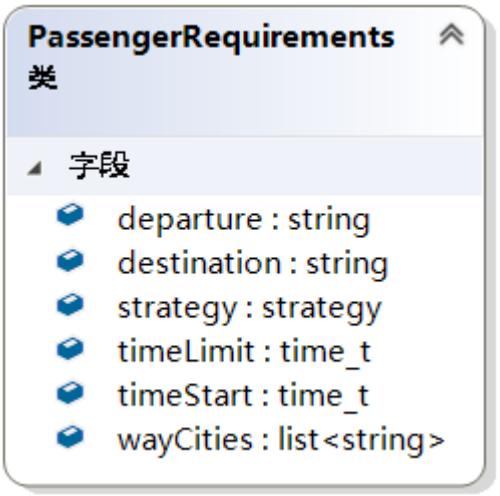


            1. 终点 city
            2. 起始时间 time[2]
            3. 票价 fare
            4. 运输方式 transportation

2. 旅客表 **class PassengerTable**

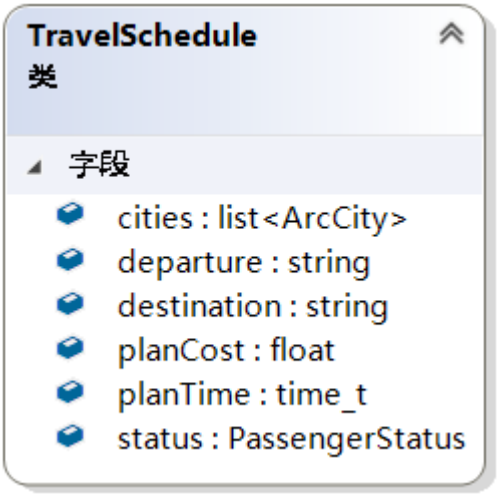


1. 旅客需求表 **class PassengerRequirements**

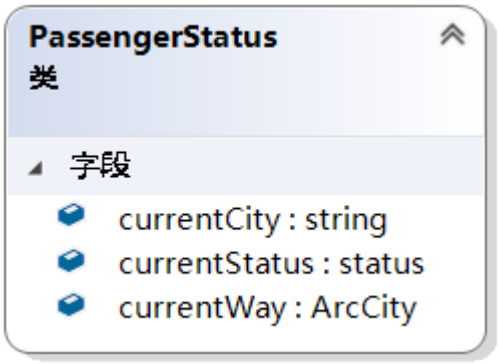


1. 起点 departure
2. 终点 destination
3. 策略 strategy
4. 时间限制 timeLimit
5. 途经城市 wayCities: list<string>

2. 旅行计划表 **class TravelSchedule**



1. 起点 departure
2. 终点 destination
3. 途经城市 cities: list<ArcCity>
4. 计划花费 planCost
5. 计划时间 planTIme
6. 当前状态 **class PassengerStatus**



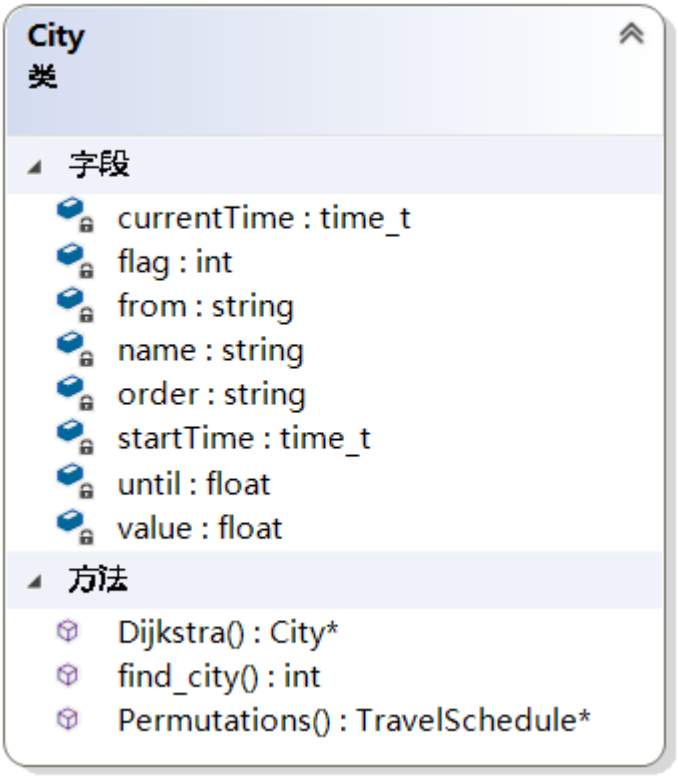
1. 当前城市 currentCity
2. 当前状态 currentStatus
3. 当前行程 currentWay

3. 日志文件 **class Logger**

    1. 日志文件流 logger

4. 路径算法模块

    1. **class City**

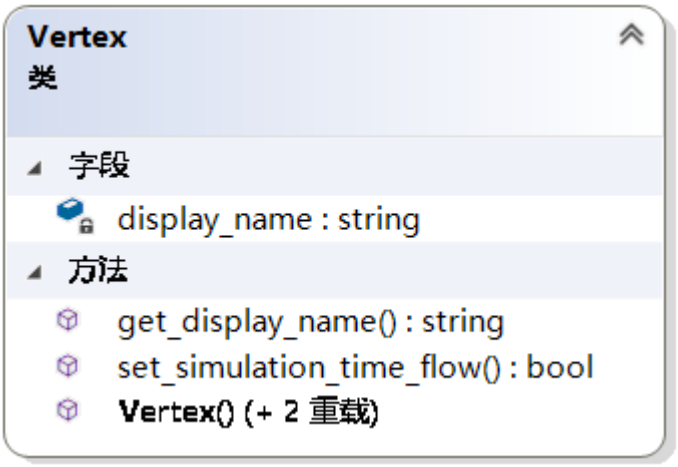


        1. 当前时间 currentTime
        2. 扫描标识 flag
        3. 来自 from
        4. 当前名称 name
        5. 航班 order

6. 从之前目的地出发时间 startTIme
7. 目前花费的钱 until
8. 此趟航班价格 value

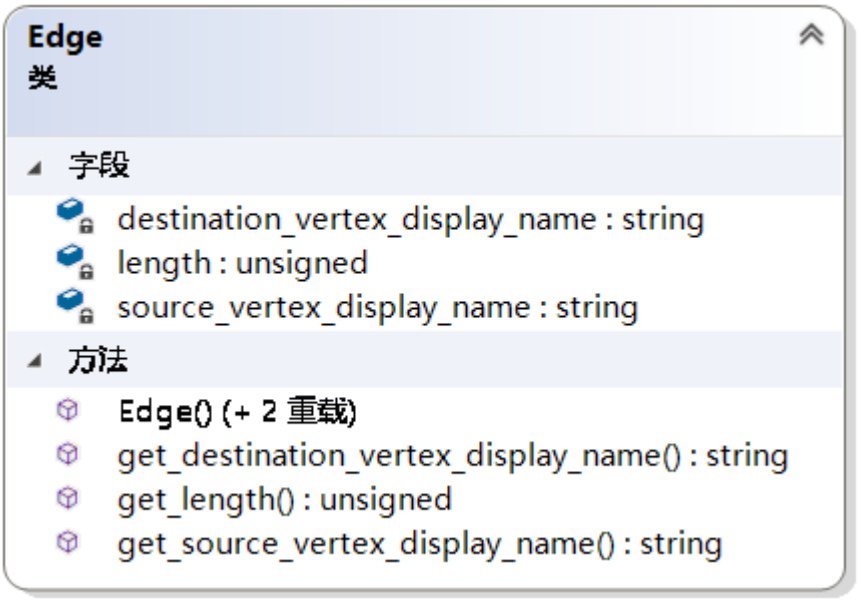
5. 通信模块 **namespace fdt**

1. 顶点 **class fdt::Vertex**



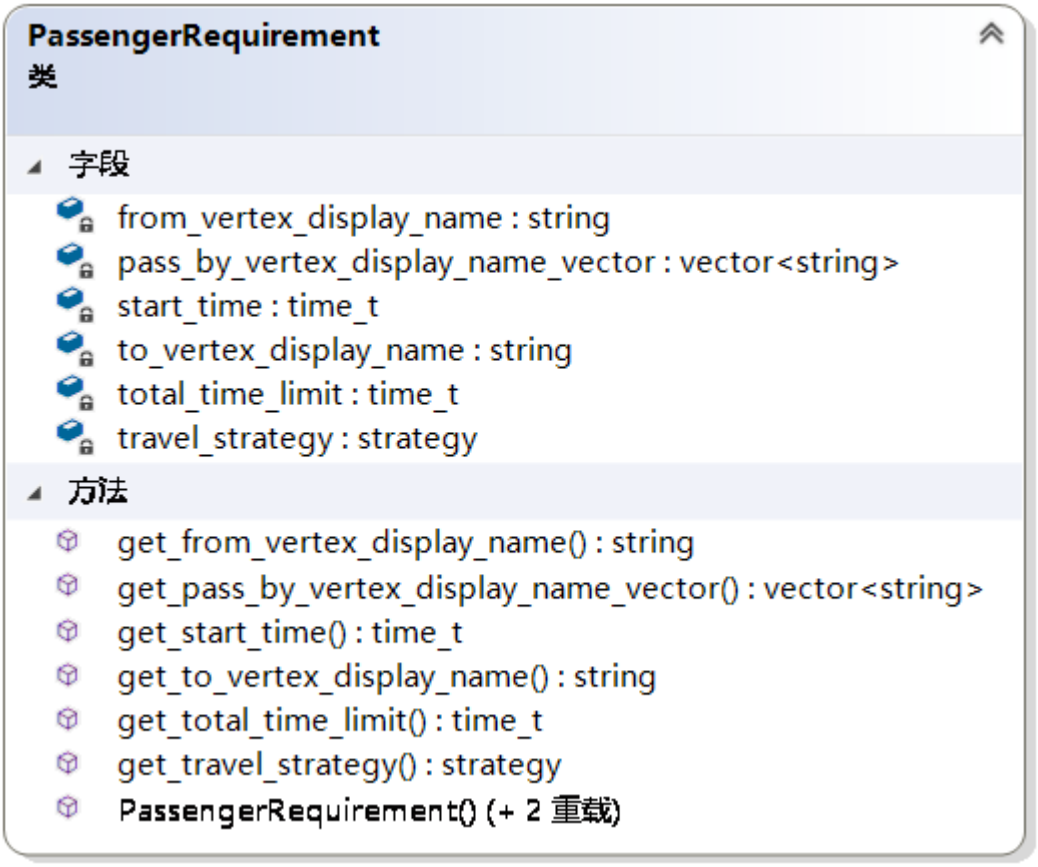
1. 功能：用于传输的顶点节点，使用display_name选择不同的顶点实例。
2. 显示名称 display_name

2. 边缘 **class fdt::Edge**



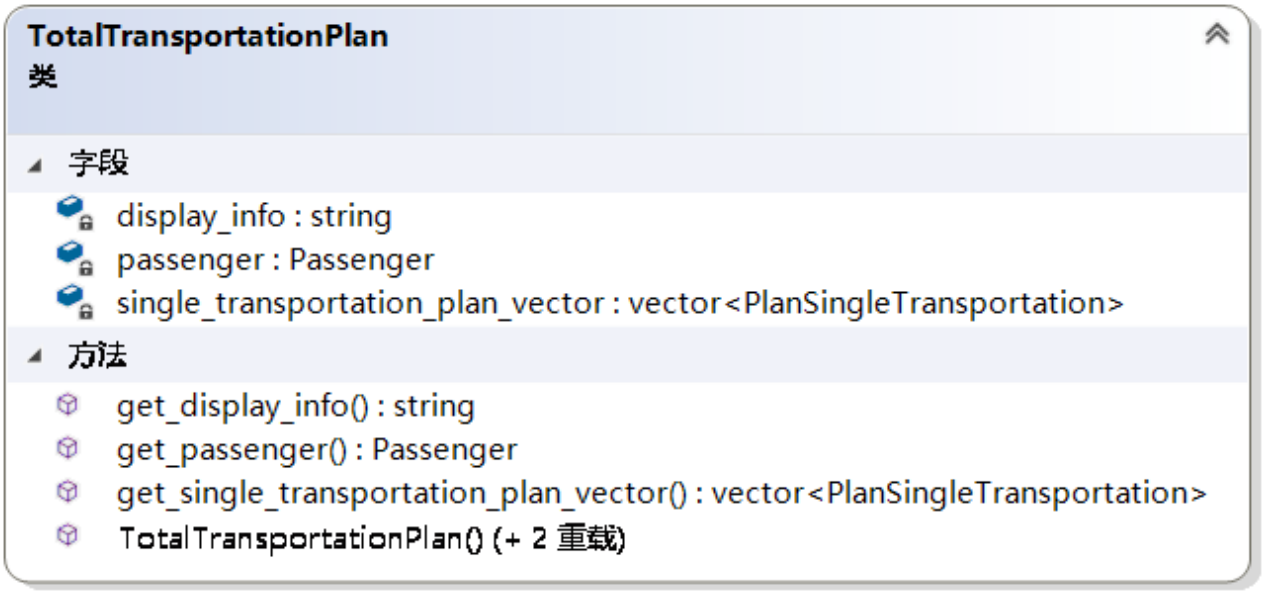
1. 功能：使用边缘在地图上绘制边缘。
2. 源顶点显示名称 source_vertex_display_name
3. 目的地顶点显示名称 destination_vertex_display_name
4. （可选的）长度 length

3. 乘客需求 **class fdt::PassengerRequirement**



1. 功能：传递PassengerRequirement的实例，用于后端计算
2. 来自顶点显示名称 from_vertex_display_name
3. 到达顶点显示名称 to_vertex_display_name
4. 开始时间 start time
5. 途径顶点显示名称向量 pass_by_vertex_display_name_vector: vector<string>
6. 旅行策略 travel_strategy
7. 总时间限制 total_time_limit

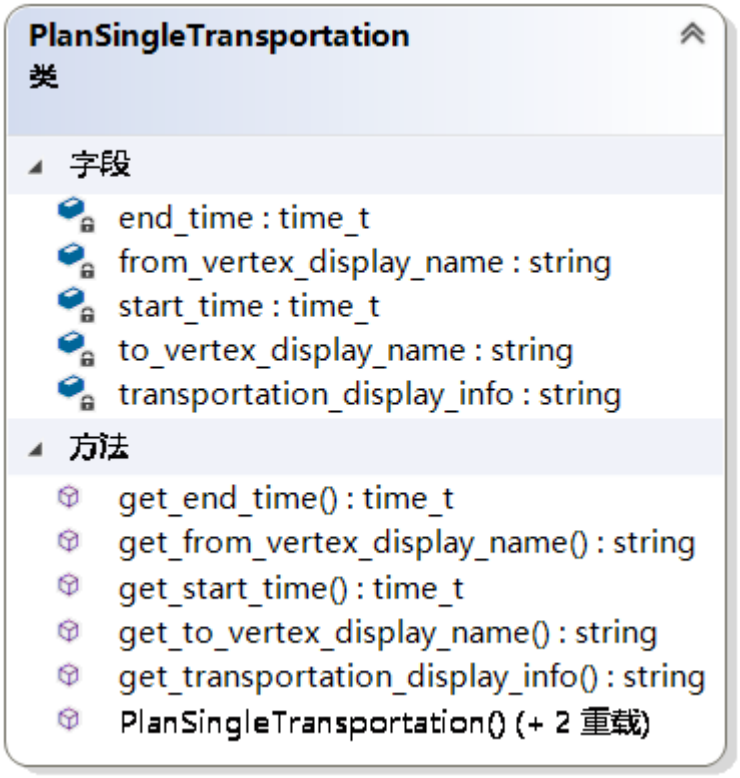
4. 运输总计划 **class fdt::TotalTransportationPlan**



1. 乘客 **class fdt::Passenger**

1. id ID

2. 单一运输计划 **class fdt::PlanSingleTransportation**



1.功能：计算结果
2.来自顶点显示名称 from_vertex_display_name
3.到达顶点显示名称 to_vertex_display_name
4.开始时间 start_time
5.结束时间 end_time
6.运输显示信息 transportation_display_info
7.注意：来自顶点和到达顶点可以是一样的

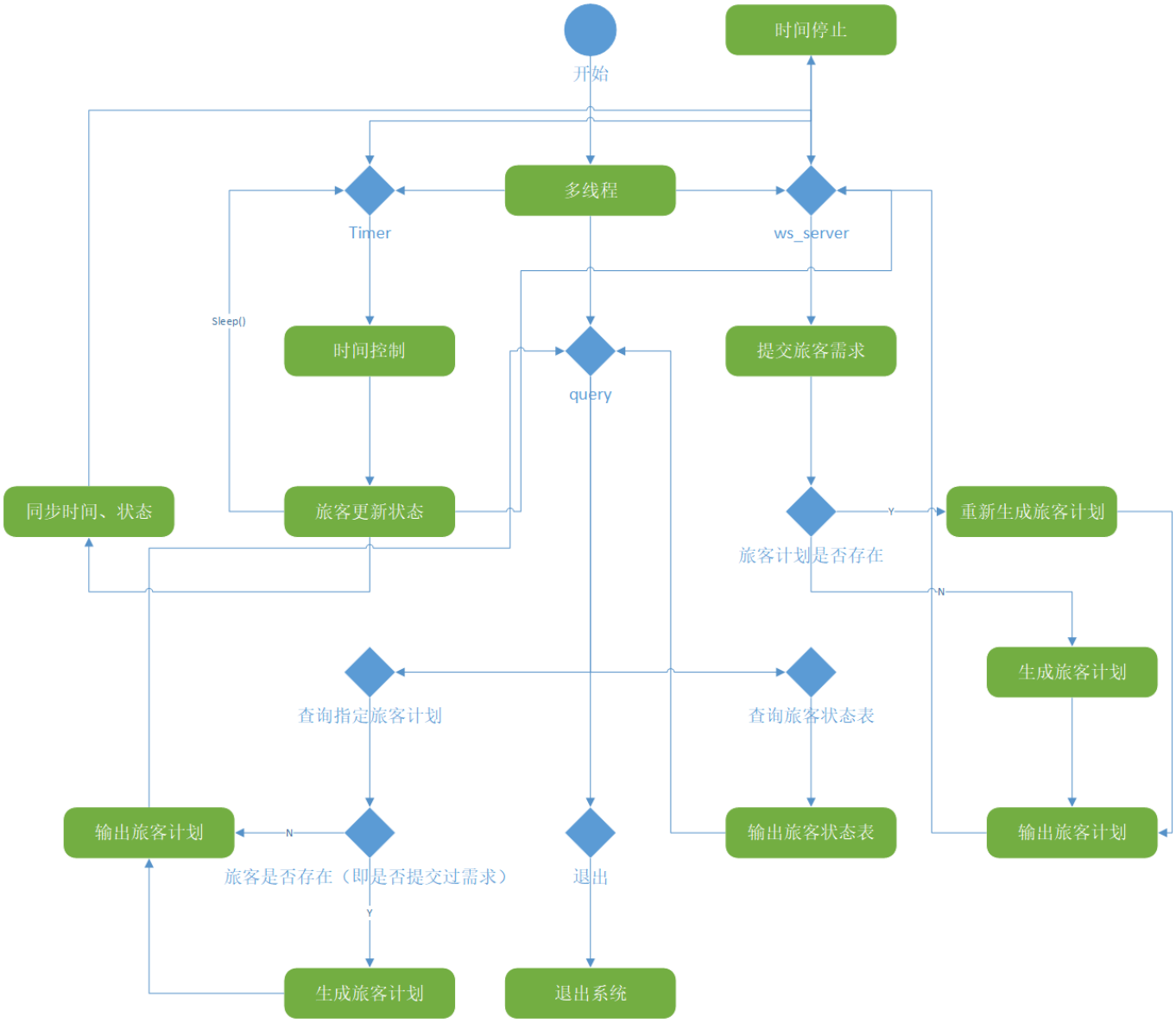
主模块和各子功能模块

1. 主模块
    1. **时间轴控制模块**
        1. 对通信模块提供信息传输接口
        2. 对其他模块进行控制
        3. 进行时间模拟
2. 子模块
    1. **时刻表模块**
        1. 对主模块提供增加、删除、修改接口
        2. 对所有模块提供查询接口
    2. **乘客模块**
        1. 对主模块提供增加、删除、修改接口
        2. 对所有模块提供查询接口

        1. 查询旅客是否存在（即是否提交过需求）
        2. 查询旅客旅行计划
        3. 查询旅客当前状态
3. **路径算法模块**
    1. 对主模块提供查询接口
        1. 计算路径（3种策略）
4. **通信模块（cpp）**
    1. 向其他模块提供切面服务，封装通信细节
    2. 保留切面接口令其他模块实现本模块通信协议要求
    3. 监听 client 请求，并将请求包装为 cpp object 使之符合切面接口协议
5. **通信模块（ECMAScript 6）**
    1. 接收 cpp 通信模块所发数据，将数据包装为 object 进行下一步处理
    2. 包装 client 通信请求，将 object 发向 server
6. **数据可视化模块**
    1. 将 server 发来旅客及旅程安排信息绘制，以直观的方式呈现给用户
    2. 接受用户数据，抽象用户行为为数据包，充当人机交互接口

三、系统架构

1. 对模块流程进行描述



- **主模块**：程序入口位于主模块，主模块首先通过本地时间初始化timer线程，通过时间表文件初始化地图信息，并启动ws_server线程，之后执行查询模块的功能，并打印相应内容，使用日志文件处理模块存储日志信息。
- **状态动态查询显示模块**：之后主模块中执行一个不断读取用户命令行的组件，在命令行当中输入"check"以查看当前所有用户的状态，信息包含此趟行程的起止地点、时间、航班信息；在命令行中输入某一用户的用户名，即可查看该用户的所有行程计划表，内容包含每一趟行程的全部信息；在命令行中输入"#"以退出模拟旅行程序。同时，日志文件模块会对查询的信息进行存储。
- **时间轴模块**：在timer线程中，timer通过一个flag来控制虚拟时间是正常以时间为轴向前推进还是暂停，在正常以时间为轴向前推进时，timer会依据旅客计划表对旅客状态进行更新。
- **旅游线路设计和输出模块**：生成相应的旅游线路，并且提供以各种格式打印各种信息的功能。
- **通信模块**：通信模块会将主模块的地图信息传送给数据可视化模块来进行初始化，之后等待接受数据可视化模块递交的旅客需求，调用主模块的算法获取旅客计划，返回给数据可视化模

块，或者接受主模块的到旅客状态的变更信息，返回给数据可视化模块。

- **数据可视化模块**：使用从通信模块获取的地图信息进行初始化，之后负责获取用户要求的读取并递交给通信模块，并且用图形化界面模拟每个旅客的状态。
- **日志文件处理模块**:完成相应的日志文件写入和查询结果输出等功能，包括时间变更、旅客状态变更、旅行计划规划情况、查询结果等内容。

2. 数据结构的基本操作
    - 详细见概要设计/数据结构设计中的配图，在此不再赘述。
3. 主要函数调用关系（不包含与ws_server、transmission_protocol的调用关系）
    - `main()` 调用 `timer()` 、 `fdt::start_ws_server（）` 、 `PassengerTable::printPassengerStatusTable()` 、 `PassengerTable::findPassenger()` 、 `PassengerTable::printTravelSchedule()` 。
    - `timer()` 调用 `Logger::out()` 、 `PassengerTable::updatePassengerStatusTable` 。
    - `PassengerTable::printPassengerStatusTable()` 调用 `Logger::out()` 。
    - `PassengerTable::printTravelSchedule()` 调用 `TravelSchedule::getTravelSchedule()` 、 `Logger::out()` 。
    - `TravelSchedule::getTravelSchedule()` 调用 `TravelSchedule::generateTravelSchedule()` 。
    - `TravelSchedule::generateTravelSchedule()` 调用 `City::Permutations()` 。
    - `TravelSchedule::reSchedule()` 调用 `PassengerTable::delPassenger()` 、 `PassengerTable::addPassenger()` 、 `PassengerTable::getTravelSchedule` 。
    - `City::Permutations()` 调用 `City::Dijkstra()` 、 `City::findCity()` 。
    - `City::Dijkstra()` 调用 `City::findCity()` 、 `TimeTable::getCityMap` 、 `TimeTable::getCitySet` 。
4. 主函数伪码

```
int main（）{
    新建乘客表；
    新建时间表（依据读取的文件）；
    新建日志；

    新建计时器线程；
    分离计时器线程；

    start_ws_server（）；

    LOOP（读取输入）{
        如果输入是“#”
            退出程序；
        或者如果输入是“check”
            计时器暂停；
            打印所有乘客的旅行状态表；
            计时器开启；
        或者如果输入是存在的乘客名字
            计时器暂停；
            打印该乘客的旅行记录表
            计时器开启；
    }
    返回0；
}
```

四、算法设计

算法选择

程序采用基于**Dijkstra**的排列算法，主要是出于以下几点考虑：

- 程序实现的实际需求为客户对出行计划的安排，对于结果的精确性和实用性需求有着较高的要求。
- 通常情况下，用户对于途经城市数量的需求较小，于是Dijkstra算法及全排列算法在满足一定效率同时，能够将计算结果的合理性最大化。

算法函数

- 算法由两个函数构成：
  1.City* City::Dijkstra(string& start, enum strategy s, time_t start_time)，该函数为Dijksra的实现函数，传入参数列表为起始城市名称、用户选择策略以及调用时模拟进程的时刻。函数返回是City类型的指针，其中存储的是以传入城市为起点，到达其他城市的最短路径信息的列表。由于涉及航班、列车等出发时间，因此起始时间是Dijkstra计算的关键信息。
  2.TravelSchedule* City::Permutations(PassengerRequirements& require)，该函数为算法控制

函数，负责实现对途经城市的排列及调用Dijkstra进行最优路径计算，并返回给调用函数TravelSchedule类型的旅行规划。

算法计算过程

- 首先，当客户传入请求时，调用Permutation函数，初始化一个空白TravelSchedule类型的旅行计划表、存储Dijksta返回结果的City结构类型指针，随后，根据用户途经城市的旅行需求，初始化用于进行排列操作的城市索引列表，为string类型的数组，称之为viacity。至此，关键变量初始化完毕。
- 之后，以viacity的排列情况对该情况下进行最优方案求解。具体过程如下：
  1.将旅客的旅行计划逐步分解，对每一趟行程单独操作，直至旅客到达目的地。循环过程中，以每一步的起点作为调用Dijkstra的计算起点，计算到达各个其他城市的最优路径；其中需要说明的是，最优路径的定义是根据用户策略所决定的。将整张表格更新完成后，返回该城市路径表。
  2.根据每一步行程的城市路径表填充临时旅行计划表，产生在该viacity组织途径城市的方案下的最优路径，对比现有的最优方案，若新生成的方案更优，则代替旧方案。
  3.一种组织方式求解完毕后，利用C++提供的库函数prev_permutation对viacity进行重新排列，重复步骤一二，直至所有可能的城市途径方式均计算完毕。
  4.在viacity遍历完成后，此时存储的旅行计划表即为所有情况下的最优解，返回给调用函数即可。
- 对于用户的三种需求，在算法中有着不同的处理方式，具体体现在Dijkstra计算过程及Permutation函数的筛选过程中：
  1.最小花费策略：在Dijkstra计算时，当前节点的当前花销与航班价格相加，如果小于目的地城市的到达花费，则更新，以此类推；在Permutation函数控制时，若旅行计划总花销小于当前保存的旅行计划花销，则替换。
  2.最短时间策略：与最小花费策略基本相同，将考虑最小花费变为考虑最短时间。
  3.限时最省策略：在这种需求下，Dijkstra计算时可以有两种方法，即选择尽量缩短时间还是尽量减少花费，在这二者之中我选择了尽量缩短时间，这样选择的好处在于，在情况较为极端的条件之下，若以金钱为首要考虑对象，可能无法规划处所需路径，然而以时间为首要对象时，能够尽最大限度保证规划成功，在Permutation函数中，再不断缩减用户花费，从而实现限时最省的策略。

五、模拟系统

火车和飞机的时刻表模拟

- 在**火车和飞机的时刻表模拟**上，我们通过从网络上搜索两个城市之间交通工具的价格来获得10个特定城市之间交通工具的费用与时长的样例，并且通过简单的数据处理来生成格式合适且每日重复接近一周共接近600次火车车次、飞机航班用于测试。

时间轴模拟

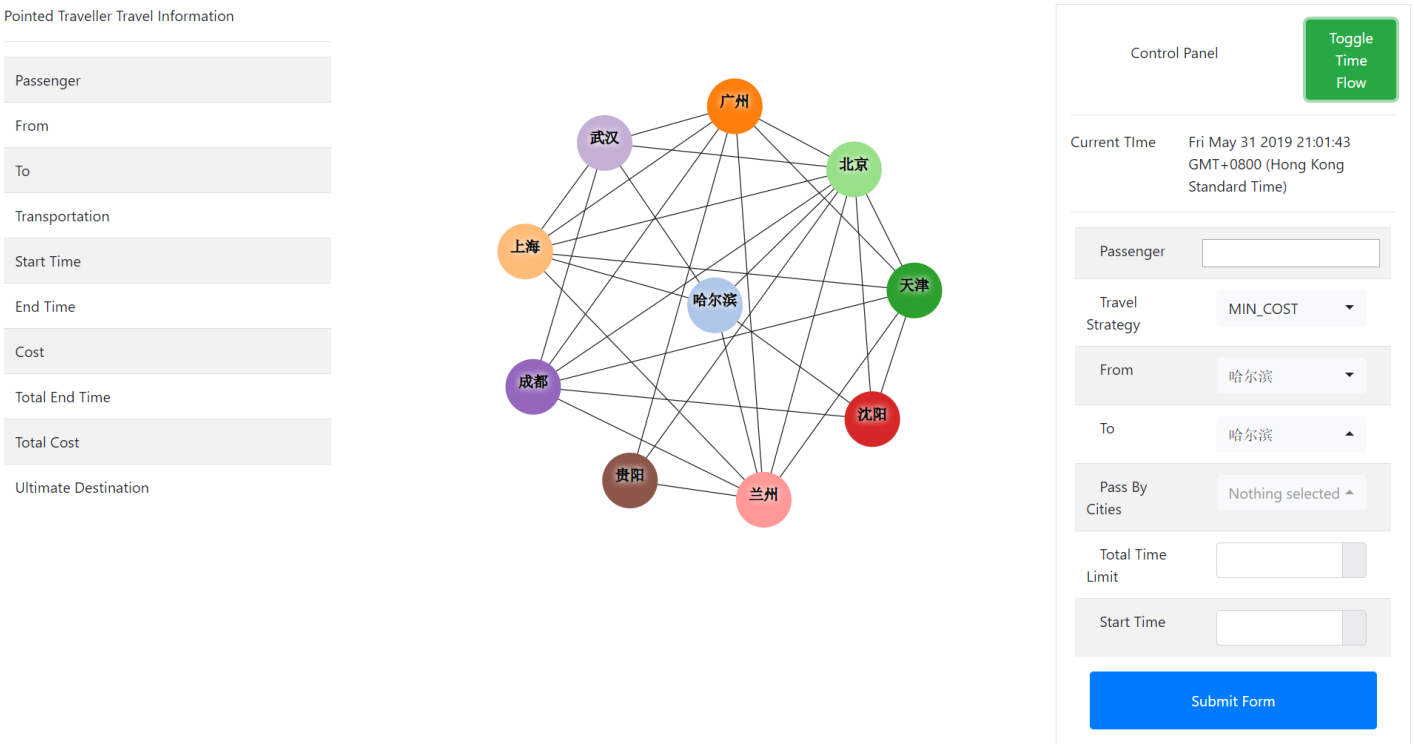
- 在**时间轴模拟**上，我们采用的了Sleep()来模拟时间刷新的间隔，忽略控制流程本身所花费的时间，这种策略可以较好的适配每5-10秒刷新1小时的时间轴控制要求。

旅客旅行模拟

- 在**旅客旅行模拟**上，我们可以在命令行界面查询旅客的计划和状态，并且旅客出发前会提醒旅客出发，出发或到达都会打印信息；此外，还可以打开配套的网页，在图形化界面查看多个旅客当前的状态和行程，自动生成的不同颜色的旅客，旅客的名字在小点上，旅客会匀速在代表2个城市的两个点间移动，或者在某个城市等待出发；在城市的模拟上，我们会自动生成代表不同城市的不同颜色的点，城市的名字在点上，城市之间通过初始时刻表弹性连接，没有固定的地理位置，只有可以拖拽的相对位置，城市点会倾向于分开，不会因随意拖拽而重叠或密集，详细请见界面设计。

六、界面设计

界面展示设计



- 界面主要分为三部分，左侧为**信息显示模块**，右侧为**用户操作模块**；中间部分为**模拟地图展示模块**。
    - 旅行模拟显示模块在中央，会根据用户需求及计算返回结果模拟旅行计划。城市节点以大圆圈表示，标有城市名称，城市之间的连线表示城市之间存在通路；用户以小圆圈表示，

标有用户名，模拟时在城市之间穿梭。城市节点之间互相存在引力和斥力，从而形成一个稳定系统，每个节点可以随意拖动，观感与使用感新颖而别致。

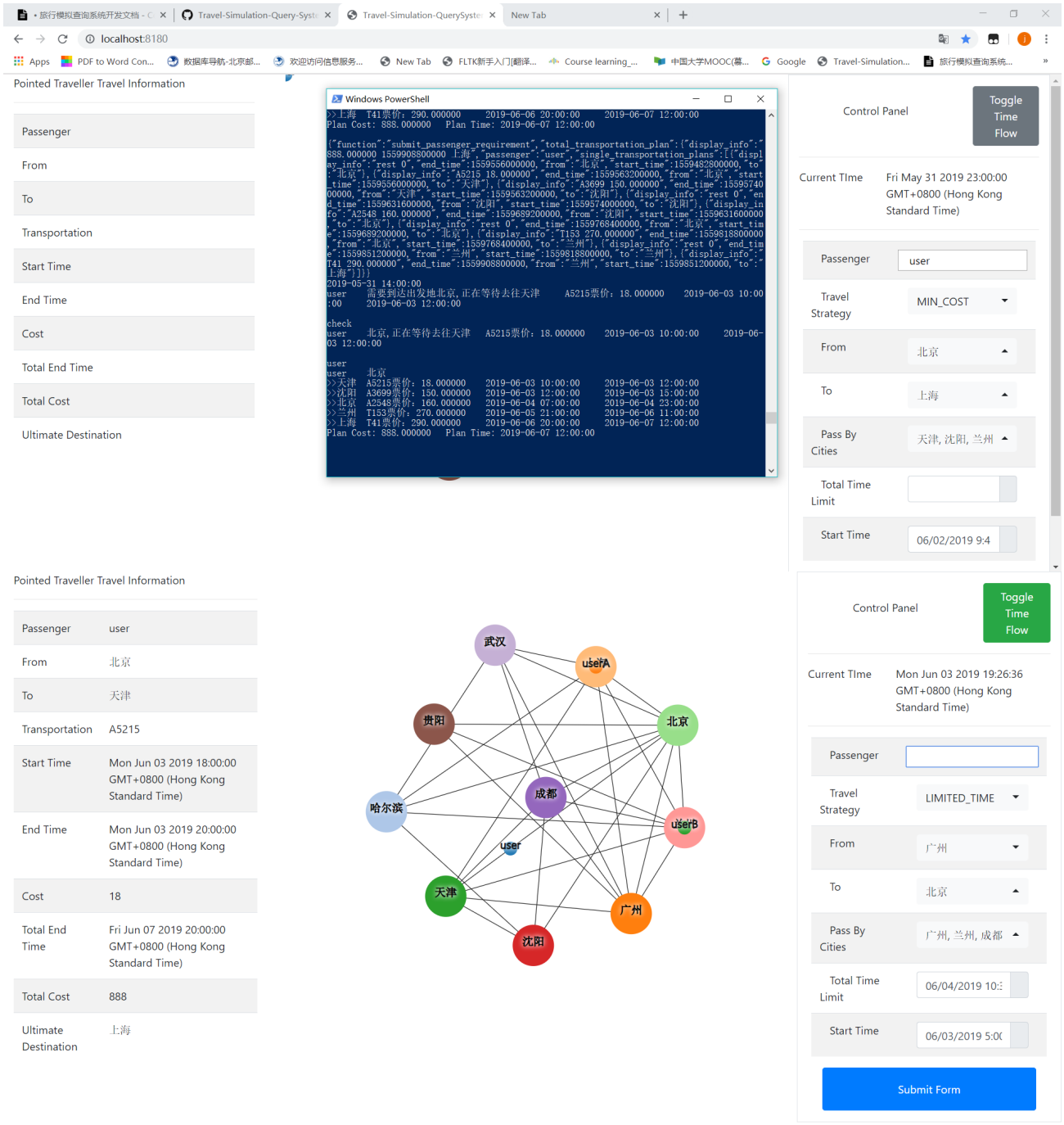
- 左侧菜单显示当前鼠标指向的用户的当前信息：当前用户状态、航班信息、旅行信息等。
- 右侧用户操作模块主要进行用户名、用户需求的输入及暂停模拟按钮控制。

界面实现设计

- **d3.js** https://d3js.org/ (https://d3js.org/)
  一款数据驱动的可视化引擎，使用本框架实现抽象旅程图模型的可视化。
  将城市抽象为无向图中的节点，在图中以大圆点表示；可能的旅行路线抽象为边，在图中以实线表示。
  采用库伦斥力以及牛顿万有引力模型进行点与点的位置模拟。
  旅客在旅行时其当前位置抽象为小圆点，通过算法计算其在整体计划中的位置，根据结果进行可视化。
  采用库中带有的分类着色标记函数对大点、小点分别按两组进行着色以区分。
- **bootstrap4** https://getbootstrap.com/ (https://getbootstrap.com/)
  流行的前端样式库。
  本例中采用其完好包装的表格样式用于左右侧栏数据展示项的显示。
  采用其按钮样式作为鲜明的控制按钮方案。
  采用其易用的 Grid 布局方式进行整体布局划分设计。
- **boostrap-timepicker** https://jdewit.github.io/bootstrap-timepicker/
  (https://jdewit.github.io/bootstrap-timepicker/)
  优秀的时间选择件，本例中将其应用于旅行起始时间、旅行最终时间的数据输入部分。

七、范例执行结果及测试情况说明

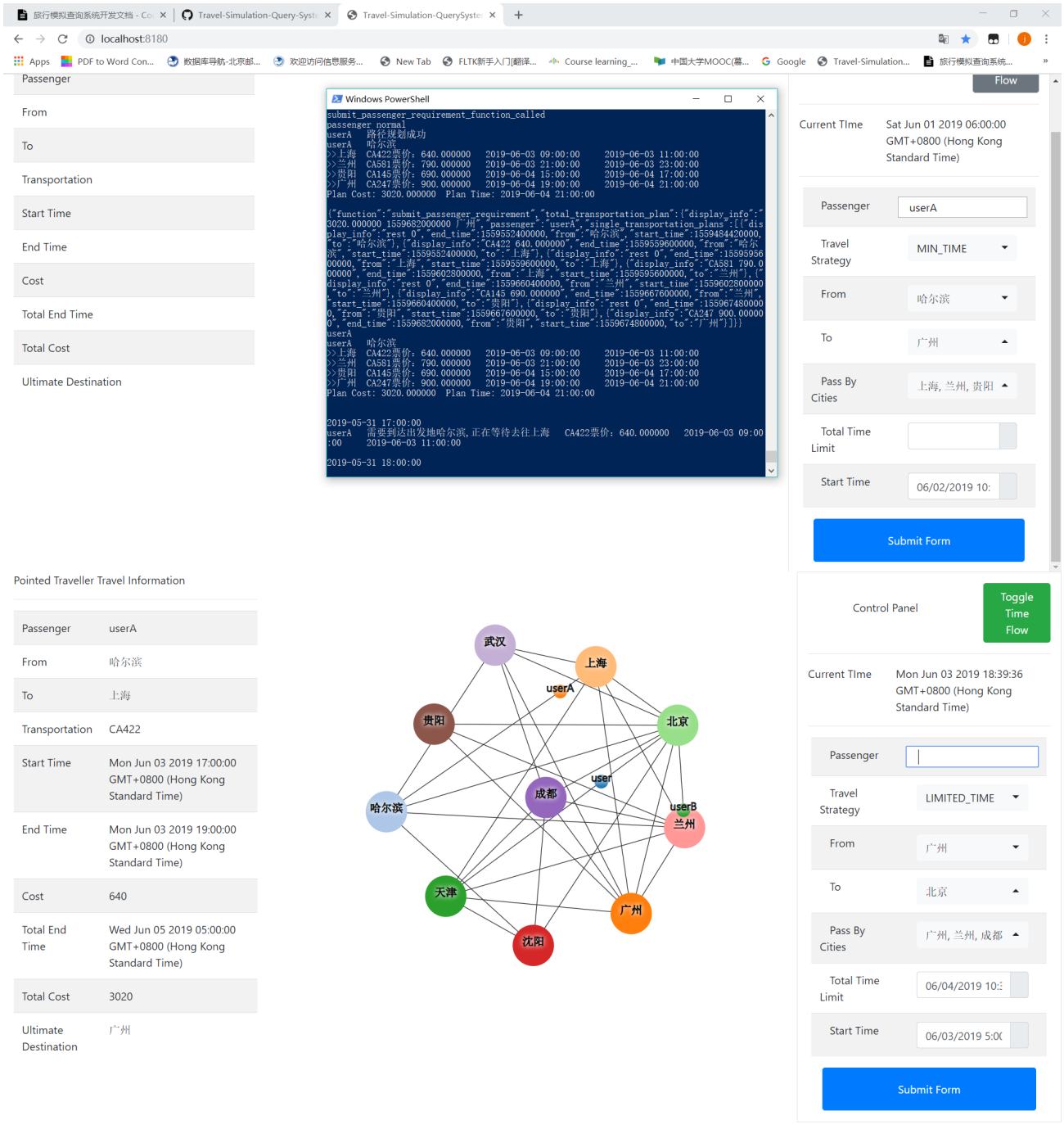
- 在测试过程中，我们分别对三种用户旅行策略进行了多组数据的检验，下面举出三组具有代表性的例子说明：
  - **最少花费测试：**
    我们取北京经由天津、沈阳、兰州最终抵达上海的最省旅行策略进行测试，测试结果如下：



可以从命令行返回的结果看出，算法明显采用了牺牲时间而缩减成本的策略，规划路径采用的均为廉价的出行方式，规划结果也较为合理，无绕路等意外情况发生。

- **最短时间测试：**

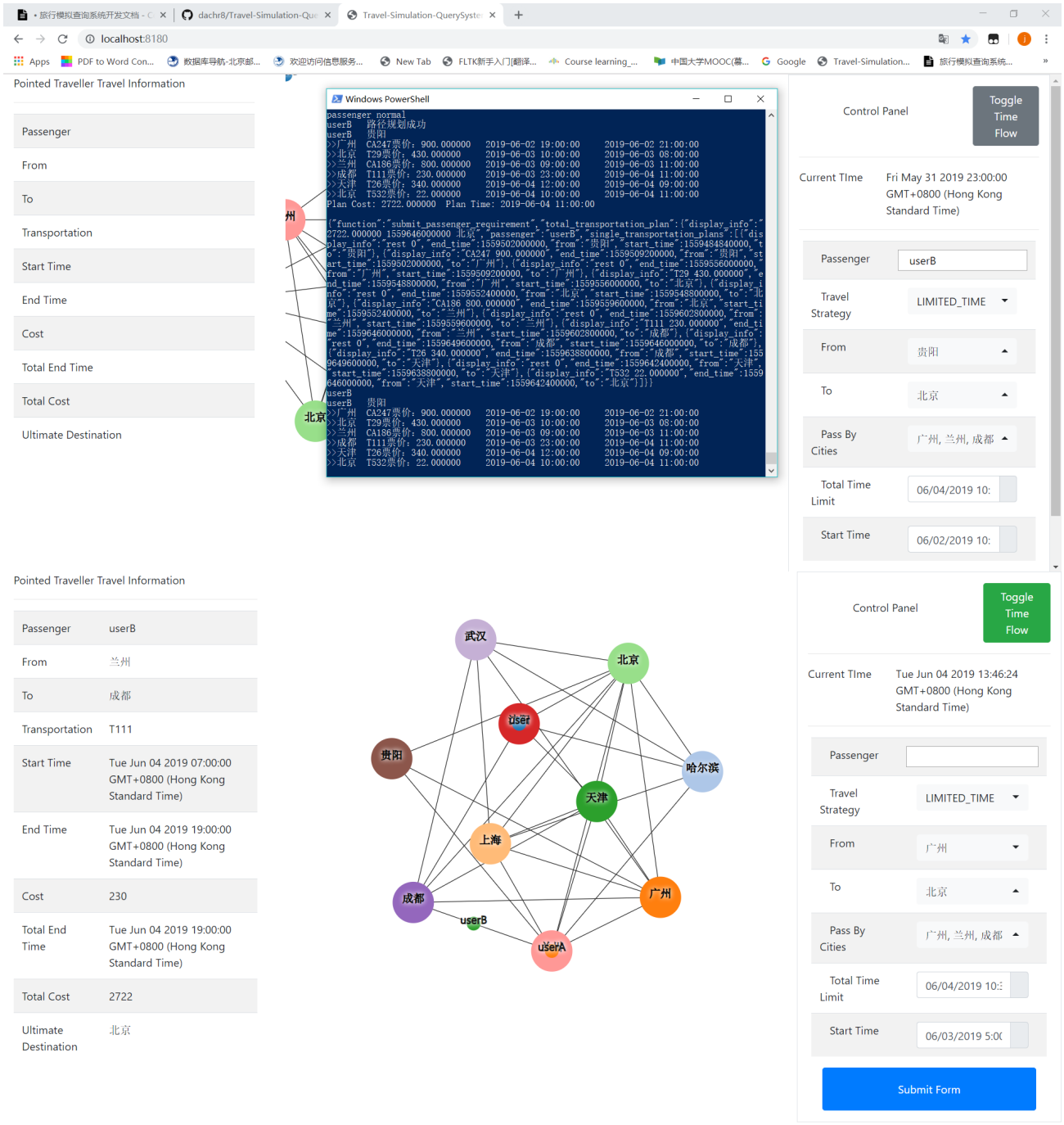
  该组测试中，我们选择由哈尔滨出发，经由上海、兰州、贵阳最终抵达广州的规划，这组测试中的城市横跨中国南北东西，距离较远，城市较多，进而分析旅行效率。



观察旅行结果，可以观察出规划的路程只花了两天时间，便完成了五个城市的旅行，花费时间少，花费较高，针对用户需求模拟效果好。

- **限时最省测试：**

  该组测试中，我们选择由贵阳出发，经由广州、兰州、成都最终抵达北京的规划，从6月2日出发，要求6月4日之前完成模拟。



在模拟结果中，算法模拟出了一条符合时限条件的路径。在时间的控制上，我们特意选取了较为严苛的时间限制，测试结果符合算法预期。算法侧重对象是时间的合理性，在优先满足时间的情况下尽量缩减花销。结果显示在时限紧张时，算法效果理想。但是在时间十分充裕的情况下，算法效果略差于以花销为侧重点的方案，因此时间十分充裕的情况推荐用户选择花销最省策略。

八、评价和改进意见

- 算法快速且准确。我们使用了基于Dijkstra的排列算法，尽管算法复杂度较高，但在中小规模问题的解决上具有良好的稳定性，几乎可以100%求出相应策略的最优解。而这一点，也保证了为

实现美观且且实用的图形化网页界面预留了充足的资源，充分地保证了用户的资源。如果我们选择了诸如模拟退火、遗传等NP算法，虽然算法复杂度会相对好看，但实际上对于此次实验的需求，可能会有浪费开发时间和运行效率等等已知的缺点，而它的优点并不足以覆盖过缺点，这也是我们在有相关NP算法知识后得出的结论。

- **界面设计**美观且实用。我们使用了WebGL和Bootstrap实现了我们的图形化界面。依赖一些优秀的图形库和资源，设计了最终的界面。
- **控制流程**仍有优化空间，目前采用的是Sleep()来模拟时间刷新的间隔，忽略控制流程本身所花费的时间，这种策略可以较好的适配每5-10秒刷新1小时的时间轴控制要求，但对于与真实时间同步的需求来说，显然是不匹配的，此时应采用快速获取系统时间的方法来提高刷新率，并且开额外的多个线程进行路径规划的计算，这样才能保证系统时间的准确性。
- **数据结构分层**控制模块、算法模块、网络模块使用了不同的存储结构来存储地图（城市及航班）、旅客（要求及计划），分别满足了容易查询、容易计算、容易传输的多个要求，而且还实现不同接口进行数据转换，合理的利用了数据。
- 原定的**用户输入**有标准输入、文件输入和图形界面输入3种方式，但经过是思考后，首先放弃了标准输入，毕竟在不暂停时间轴时进行输入会使终端变得较乱，不便于使用；再其次是文件输入，本程序实现了文件输入接口，但鉴于验收要求是自行构图且不需要提供初始乘客信息，所以我们索性也取消了控制模块的文件输入接口，使终端只能进行查询功能（乘客状态表和乘客行程计划），使用起来更加简洁；图形界面输入，可以进行每次单个用户的数据输入，在前后端都对非法操作进行了限制，实用性也很良好，但暂不支持文件传送，这主要因为对此功能没有明确需求，而且开发周期紧俏，受到时间限制，但因为程序设计预留的接口完整且文档充足，后续开发者可以很轻易调用对应方法来实现诸如此类的额外功能；此外，我们提供了“时间轴暂停”功能来减少输入时花费的时间对模拟时间的影响，便于使用和调试，而且，在图形化界面中途退出再进入时，也会默认“暂停时间”，便于调试，可以优化的点是把“时间暂停”按钮改成“时间速率”滚轴，可以有效地提升模拟系统的广泛性。