**各模块设计说明**

**（一）Initialize初始化模块**

该模块包含四个函数，功能分别如下：

①void InitCityList()：初始化城市，设置每个城市的标号、城市名、在地图上的坐标、风险值。本系统共设置了16个城市（6个高风险，5个中风险，5个低风险），即为图构造了16个顶点。

②int ToCityID(std::string city) ：时刻表文件里的城市是字符串类型，将其转换为该城市对应的标号并返回。

③void InitTable()：通过文件初始化时刻表，每添加一个班次的交通，就根据该班次的起始站和终点站构造一条有向边。

④void LoadImg()：加载图片，为动画界面的贴图做准备。

⑤void InitCanvas()：初始化动画界面（未启动系统界面）。

函数①③执行完后，整个系统的图就构建好了，后续乘客制定路线都根据这个图来计算。

本模块不需调用其他模块。

**（二）main主模块**

该模块最主要的函数为：

1.void TransferSystemState()：根据与用户交互的信息更新状态

系统共设计了6个状态，分别为NORMAL(常规状态)、CLICK\_DEPART(点击出发地状态)、CLICK\_DEST(点击目的地状态)、CLICK\_STRATEGY (点击策略状态)、CHANGE\_DEST (更改目的地状态)、CHANGE\_STRA (更改策略状态)。

①常规状态：

用户在动画界面上点击，系统获取鼠标信息，根据鼠标点击的坐标判断用户点击按钮或功能。

当用户点击动画界面上的“添加乘客”按钮时，计时停止，更改状态为“点击出发地状态”，并初始化相关变量，设置passStartTime和passStartTime。打印“添加乘客”事件到日志中。

当用户点击动画界面上右侧信息栏目的地框下的“更改”按钮时（只有在系统中有乘客的情况下才有效），计时停止，更改状态为“更改目的地状态”，并初始化相关变量。打印该事件到日志中。

当用户点击动画界面上右侧信息栏策略选择框下的“更改”按钮时（只有在系统中有乘客的情况下才有效），计时停止，更改状态为“更改策略状态”，并初始化相关变量。打印该事件到日志中。

当用户点击动画界面上的乘客表时，点击序号即可查询到相对应的用户的所有信息，查询得到的信息会被打印到动画界面和日志中。查询时系统不会暂停。

②点击出发地状态：

用户在动画界面上点击，系统获取鼠标信息，根据鼠标点击的坐标判断用户点击的是城市还是按钮。若点击的是城市，设置passDeparture为点击城市的标号并贴图；若点击的是“取消”，就返回常规状态，恢复计时；若点击的是确定，就进入下一状态“点击目的地状态”（只有在点击过城市之后才会进入）。调用输出模块输出用户点击的信息到日志里，更新动画界面。

③点击目的地状态：

同样先根据鼠标点击的坐标判断鼠标点击了什么，若点击的是城市，设置passDestination为点击城市的标号并贴图；若点击的是“取消”，就返回常规状态，恢复计时；若点击的是确定，就进入下一状态“点击策略状态”（只有在点击过城市之后才会进入且点击的目的地跟出发地不能是同一个城市，若是同一个城市，点击确认了也不会进入下一阶段，且会调用输出模块输出错误信息）。调用输出模块输出用户点击的信息到日志里，更新动画界面。

④点击策略状态：

根据鼠标点击的坐标判断鼠标点击了什么，默认策略为最少风险策略，若点击了策略就设置passStrategy为该策略；若点击了取消，回到常规状态；若点击确认，则调用输入模块获取用户输入的姓名和限定时间（在选择的策略是限时最少风险策略时需要），汇总所有输入的信息，调用旅客模块生成旅客，回到常规状态。调用输出模块输出该乘客的所有信息。

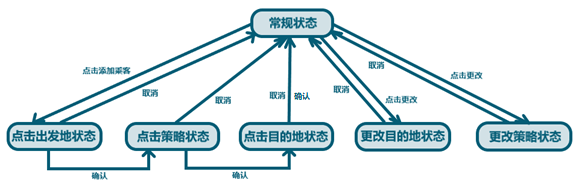
⑤更改目的地状态：

功能实现算法类似于点击目的地状态，这里只讲区别：乘客的出发地会更改为其当前所在城市，且点击的目的地可以跟出发地相同（前提是更改后的出发地与初始出发地不同，若更改后的出发地==初始出发地&&点击的目的地==初始出发地，则调用输出模块输出错误信息）。若新目的地不等于初始目的地，就调用旅客模块更新，否则就按原方案继续旅行。

⑥更改策略状态：

功能实现算法类似于点击策略状态，这里只讲区别：只有当输入的策略与原策略有所不同时才会调用旅客模块更新旅客信息。

下图为各个状态的转换关系：



2.void Update()：更新时序并完成系统中所有旅客的更新

若系统暂停，更新本次暂停时长；若系统要从暂停状态恢复计时，更新总共暂停时长，将本次暂停时长清零。

更新全局时间，公式为（当前时间-起始时间-暂停总时长-本次暂停时长）/单位时间对应的毫秒数（本系统中为10000）。

如果时序更新了，调用旅客模块更新旅客状态等信息，并调用输出模块输出相关信息。

如果系统没有暂停，就调用旅客模块更新旅客在地图上的坐标。

若到了画面更新间隔（= 当前时间-上次更新时间），则调用输出模块更新画面并将上次更新时间改为当前时间。

本模块需要调用输入输出模块和旅客模块

**（三）InputOutput输入输出模块**

输入部分为输入乘客的姓名和限定时间。

输出部分分为两种，一种是动画界面的输出，一种是日志的输出。

**（四）Passenger旅客模块**

本模块主要用于管理和更新用户的信息。

当系统添加用户时，本模块会根据从主模块得到的信息生成新的乘客，并为该乘客制定路线，根据策略调用图模块相应的算法，将返回的路线结果保存，并调用输出模块输出到日志与动画界面。

当用户更改旅行计划时，本模块在调用图模块重新制定方案前先根据状态做出操作：若乘客是在移动状态下更新的计划，则先保存当前正在移动中的的路线，然后清空路线方案，调用图模块重新制定路线，最后把之前保存的路线加回乘客路线中，保证旅客旅行的连贯性；若是在静止（等待或到达终点了）状态更新的计划，则直接清空路线方案再重新制定即可。每次重定计划都要注意已用时间等信息的更新。

当系统时序更新时，要更新乘客的坐标，根据坐标和时间判断是否需要更新乘客状态，需要则更新。乘客的每条路线中都包含了x坐标更新间隔和y坐标更新间隔，若乘客是在移动状态，则到了这时间间隔就在动画界面上移动一个像素，移动方向根据起点和终点的相对位置判断，若到达了某个城市，则根据该城市是否是目的地来决定将状态转移为等待还是到达目的地。

若乘客是在等待状态，如果系统时间==即将行驶的路线的出发时间，则更改为移动状态并开始移动。注意一种情况，比如：某乘客在x日5：00-6：00间制定了计划，此时系统使用及显示的时间仍是5：00，若乘客得到的路线的第一条是在五点出发，要注意这里的五点是x+1日的五点，因为x日五点的这趟车/飞机在该乘客制定计划时已经出发了，乘客已经错过了这天的出行，因此制定完计划后，虽然系统时间是5：00（等于第一条路线的出发时间），乘客状态仍为等待出发状态不出发。每种出行方式都是在整点出发，而乘客制定方案都是在两个整点（记为a,a+1）之间的时间制定的，按照现实逻辑，乘客是无法搭乘当天a点出发的所有交通工具，只能等第二天再搭乘。（即虽然以小时为单位，但不能简单的把时间看成离散点，时间是连续的，因此设置了一个全局变量记录精确到小数的全局时间是为了模拟分钟和秒）

若乘客是在到达目的地状态，则其所有信息可以不再更新（除非该乘客点击了更改目的地需要继续出行）。

**（五）Graph图模块：该模块包含了系统的核心算法**

规划路径算法有两种，根据旅客的策略调用相对应的算法进行路径计算。

①当旅客策略为最少风险策略或最少时间策略时，要考虑的因素只有一个，风险值/时间。算法核心是dijkstra算法。由于本系统的图是含有多重边（城市之间可能有多个班次的出行方式）的有向图，dijkstra算法不能直接使用，需要对多重边进行筛选。每条边都有一个权值，在最少风险策略下该权值计算的是风险值，在最少时间策略下该权值计算的是时间。值得注意的是每条边的权值并不是一个固定值，它在不同路线下的值是不同的，因为选择了不同的路径，在该路径上等待的时间是不同的（列车/航班并不是时时都有，需要等待到该班次的出发时间才能出发）。有关每条边的权值的计算公式如下：

1. 总风险 = 等待风险（变化值） + 路途风险（固定值）------最少风险策略
2. 总时间 = 等待时间（变化值） + 路途时间（固定值）------最少时间策略
3. 等待时间 = 本城市本条边的出发时间 – 上一条边到达本城市的时间
4. 等待风险 = 等待时间 \* 本城市的风险值
5. 路途时间 = 本条边的到站时间 – 本条边的出发时间 + 24 \* 跨越天数
6. 路途风险 = 路途时间 \* 该交通方式风险 \* 本城市风险

注意计算等待时间时，有两种情况需要再加上24小时：一是等待时间小于0（即跨越了一天）；二是乘客制定计划的时间与该边的出发时间相同且乘客的出发地与该边的出发城市相同（即制定计划时当天的这趟车/航班已经出发了，只能等第二天的这趟车/航班）。

本算法用Vertex点类（该类的信息见数据结构说明部分）来模拟所有城市计算路径，用一个vector容器来记录所有固定过的点（marked城市）便于后序遍历。算法开始时，将出发城市的点初始化，并将其加入marked city容器中。之后进入循环，循环条件为固定城市小于总城市数且终点没有被固定。每次循环对每个marked城市都进行遍历，遍历一个marked城市时，要对该城市的所有非固定邻居进行遍历，该城市与其某个邻居之间可能有多条边，要对每条边都进行遍历：重新计算这条边的权值（计算公式见上），然后将权值最小的边的标号记录下来，这个最小权值作为当前遍历的marked城市与本遍历邻居的权值，如果该邻居以正在遍历它的marked城市做中间节点可以有更小的到出发点的权值，则更新。

都遍历结束后，寻找目前可达的最短的一条没有被固定的路径，并把该路径的末点加入marked容器里（即固定所有非固定点中到出发点权值最小的点）。

目的地点被固定后即可导出路线和路线相关数据，并返回给旅客模块。

①当旅客策略为限时最少风险策略时，要考虑的因素有两个，风险值和限定时间。算法核心是深度优先搜索+剪枝（需要①中的dijktra算法辅助）。

算法开始先用①中的算法分别计算出最少时间策略下的路径和最少风险策略下的路径。

若最少时间策略计算出的总时间都大于限定时间，则该限定时间下，乘客不可能到达目的地，此时为了让用户尽可能接近于限定时间到达，将会返回最少时间策略的路径给用户，并提示用户错误信息。

若最少时间策略计算出的总时间满足限定时间，则比较最少风险策略的总时间是否满足限定条件，若满足，则返回最少风险策略的方案给用户，此时风险是最少的；若不满足，则需用深度优先搜索和剪枝重新进行路径规划，思路如下：

用城市栈结构体（包含信息见数据结构说明部分）来模拟计算路线。计算路线之前，将最小时间策略求出的风险值作为风险值上界upperRiskBound。

剪枝条件为：

(入栈的边的总权值 + 某条边的风险值> upperRiskBound || 入栈的边的总时间 + 某条边的时间 > 限定时间)

首先将出发城市压栈。进入循环进行深度优先搜索，每次循环都对栈顶元素进行遍历，遍历栈顶元素的邻居的边，计算边的权值（计算方法见①），若加上该条边不满足剪枝条件，则把这条边的终点压栈，若新入栈的点是目的地点，则保存路径并将终点弹栈，以便寻找其他可能的路径，否则退出对原栈顶的遍历，改去遍历新栈顶；若满足剪枝条件，则不加这条边，遍历下一条边。若当前栈顶找不到可以加入的边，则将当前栈顶弹出栈。注意每次压栈和弹栈都要对点做好标记，避免之后重复遍历。所有可能的路径都被遍历过的话，栈会被逐步清空然后退出循环。

将计算得到的最后路线及相关结果返回给旅客模块。

**各模块之间关系如下图：**

