1 程序体系架构概览

本系统采用面向对象的程序设计思想，总体结构可分为“交互层”“控制层”和“算法层”。

其中交互层负责响应用户操作，向控制层发送请求，并将控制层输出的信息通过 GUI 反馈给用户；控制层负责系统的整体运行，以及数据维护/处理，日志记录等；算法层负责根据旅客请求生成相应的旅行方案，并交付控制层执行。

如上图，程序主要的信息通信，以及数据流动仅发生在两相邻层次结构之间，即交互层—控制层，控制层—算法层。

2 控制层

控制层划分为四个子模块，分别负责启动程序、控制系统以及管理数据，结构图如下：

2.1 程序核心库 ProgramLib.h

程序核心库主要包含如下内容：

1. 程序运行所必备的头文件，涵盖输入输出流、标准模板库、进程库、图形界面库。

2. 一些通用数据类型的别名，目的是提高开发效率和源代码的可读性：

【例】将 `DangerIndex` 作为风险值变量类型 `int` 的别名：

```
typedef int DangerIndex;
```

将 `CityName` 作为城市名称类型 `std::string` 的别名：

```
typedef std::string CityName;
```

完整的数据类型别名见《数据结构说明和数据字典》。

3. 枚举类型，目前包含以下三种：城市类型 `CityKind`、交通工具类型 `VehKind` 和旅客状态类型 `Status`。

【例】旅客状态类型：

```
typedef enum {
    LOW = 1, MID = 2, HIGH = 3
} CityKind;
```

完整的枚举类型见《数据结构说明和数据字典》。

4. 常量定义，包括系统时钟频率 `CLOCK_FREQ_NORMAL`、最大旅行持续时间 `INS_TIME_MAX` 等常数。如下：

```
constexpr Time INS_TIME_MAX = 240;
constexpr DangerIndex DANGER_MAX = INT_MAX;
constexpr int CLOCK_FREQ_NORMAL = 1e4;
constexpr int CLOCK_FREQ_MEDIUM = 1e3;
constexpr int CLOCK_FREQ_PRECISE = 1e2;
```

可见系统维护着三种不同精度的时钟频率，它们分别有不同的用途，后文会详细阐述。

2.2 程序入口点模块 Main.cpp

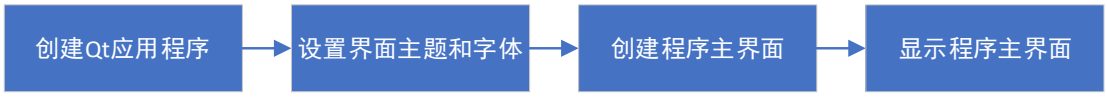
程序入口点模块负责初步初始化界面，首先，在全局变量定义一个空指针：

```
COVID_19_Travel_System* MainWindow{nullptr};
```

然后进入主函数 `main()`，创建 Qt 应用程序、设置界面风格和字体，并显示界面：

```
QApplication App(argc, argv);
QApplication::setStyle(QStyleFactory::create("Fusion"));
QFont UIFont{"Microsoft YaHei UI"};
UIFont.setPixelSize(12);
App.setFont(UIFont);
MainWindow = new COVID_19_Travel_System;
MainWindow->show();
return App.exec();
```

该模块整体工作流程如下：



此后会调用主界面的构造函数（在交互层详细说明），开始执行程序。

2.3 系统控制模块 Control.cpp

2.3.1 系统核心的初始化

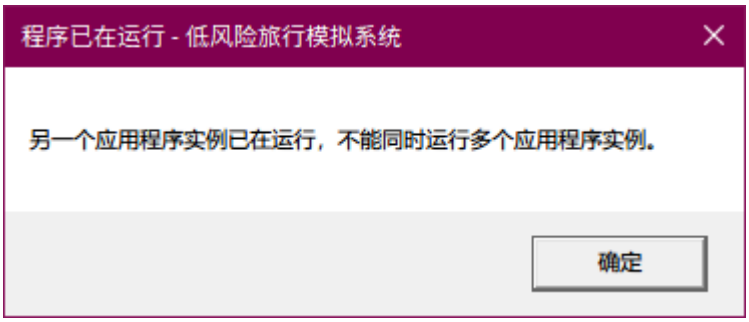
在 Control.cpp 文件中，创建了 Control 类型全局对象 System，作为系统控制类的实例：

```
Control System;
```

系统的初始化主要依赖 Control 类的构造函数，该函数执行流程如下：

1. 调用函数 `checkProcessExist()`，检查程序是否已运行在当前操作系统

检查程序是否已经在运行，如果已经运行，则阻止程序继续启动，并告知用户：



该函数的具体实现如下：

```
1.  //如果程序已经有一个在运行，则返回 true
2.  void Control::checkProcessExist() const {
3.      HANDLE hMutex = CreateMutex(NULL, FALSE, L"DevState");
4.      if (hMutex && (GetLastError() == ERROR_ALREADY_EXISTS)) {
5.          CloseHandle(hMutex);
6.          hMutex = NULL;
7.          MessageBox(nullptr,
8.              L"另一个应用程序实例已在运行，不能同时运行多个应用程序实例。",
9.              L"程序已在运行 - 低风险旅行模拟系统", MB_OK);
10.         exit(0);
11.     }
12. }
```

其中运用了 Windows 系统中的进程句柄，来判断程序状态。

2. 调用函数 `startLogSystem()`，启动日志记录系统

日志文件名称为"COVID_19_Travel_System_Log.log"，以文件输入/输出流的方式进行打开，并在程序运行期间一直保持打开状态：

```
1.  logFile = new std::ofstream;
2.  logFile->open("COVID_19_Travel_System_Log.log");
```

3. 调用 `startSysClock()` 函数以创建系统时钟线程，并启动系统时钟

系统时钟以子线程的方式运行：

```
sysClock = new std::thread(&Control::incSysTime, this);
```

并以 incSysTime()函数为线程入口函数：

```
sysClock->detach();
```

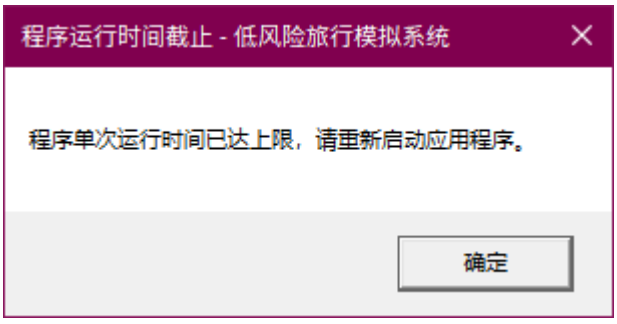
其中，`incSysTime()`函数体内以死循环的方式推进系统时间，并通过 `Sleep()`函数控制时间刷新周期，在标准时间规则下，若系统处于非暂停状态，则系统每 0.1s 更新一次高精度时间（用于地图刷新），并以“标准时间=高精度时间/100”的换算关系，使标准时间每 10s 刷新一次。

```
while (true) {
    // 以最高精度频率刷新时间
    Sleep(CLOCK_FREQ_PRECISE);
    // 系统不处于暂停状态下才刷新
    if (!sysPause) {
        // 高精度时间递增
        sysTimePrecise++;
        // 常规精度时间根据高精度时间换算
        sysTimeNormal = sysTimePrecise / (CLOCK_FREQ_NORMAL / CLOCK_FREQ_PRECISE);
    }
    ...（略）
}
```

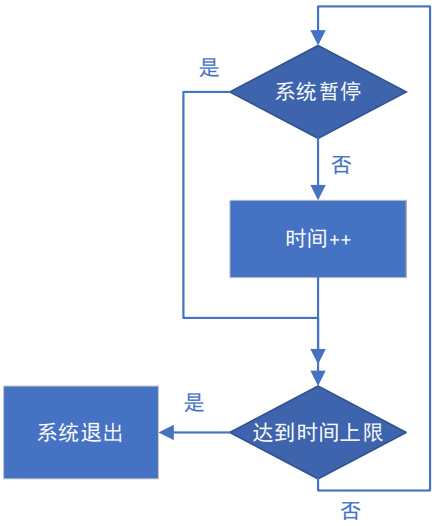
出于安全性考虑，当系统高精度时间值达到 `int` 类型的上限时，程序会强制退出，以防止溢出：

```
// 防止溢出，高精度时间达到最大值时强制关闭程序
if (sysTimePrecise == INT_MAX) {
    MessageBox(nullptr,
        L"程序单次运行时间已达上限，请重新启动应用程序。",
        L"程序运行时间截止 - 低风险旅行模拟系统", MB_OK);
    exit(0);
}
```

提示如下：



系统时钟逻辑的工作流程概括如下：



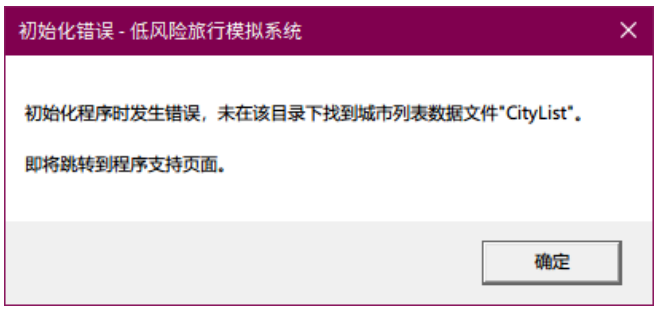
4. 依次调用 `initCityList()`、`initSchedule()`函数，初始化城市列表/时刻表

系统先初始化城市列表，再初始化时刻表，顺序不能颠倒。这两个函数的功能和实现类似，因此下文只介绍城市列表的初始化。

首先将数据文件以文件输入流的方式打开，并检查文件是否打开成功：

```
1.  std::ifstream cityListFile;
2.  cityListFile.open("CityList");
3.  // 检查文件状态，若打开失败则弹出提示，终止程序
4.  if (!cityListFile.is_open()) {
5.      Debug << "初始化城市列表失败" << std::endl;
6.      *logFile << "初始化城市列表失败" << std::endl;
7.      MessageBox(nullptr,
8.          L"初始化程序时发生错误，未在该目录下找到城市列表数据文件\"CityList\"。
    \n\n 即将跳转到程序支持页面。",
9.          L"初始化错误 - 低风险旅行模拟系统", MB_OK);
10.     // 指引用户在线下载数据文件
11.     openSupport();
12.     exit(-1);
13. }
```

如果文件打开失败，则弹出提示框，并退出程序，引导用户下载缺失的数据文件：



若文件打开成功，则逐行读取文件内容，并存储到城市列表向量，直到遇到文件尾，并关闭文件：

```
while (!cityListFile.eof()) {
    int intKind;
    CityID ID;
    CityName name;
    CityKind kind;
    int posX, posY;
    cityListFile >> ID >> name >> intKind >> posX >> posY;
    kind = static_cast<CityKind>(intKind);
    City* newCity = new City(ID, name, kind, posX, posY);
    cityList.push_back(newCity);
    cityNum++;
}
Debug << "初始化城市列表成功" << std::endl;
*logFile << "初始化城市列表成功" << std::endl;
cityListFile.close();
```

5. 调用 `startStatusRefresher()` 函数，创建旅客状态刷新器子线程

```
travelStatusRefresher = new std::thread(&Control::updateSystem, this);
travelStatusRefresher->detach();
```

该线程的入口点函数为 `updateSystem()`，功能主要是记录旅客旅行事件，输出到日志文件：

```
*logFile << "当前时间: " << System.sysTimeNormal <<
"\t 旅行 ID: " << ins->getID() << ", 当前位置: " <<
ins->getStatusLog(System.sysTimeNormal) << std::endl;
```

至此，系统核心初始化完毕。

2.3.2 系统的关闭

1. 当程序关闭时，系统必须释放资源，包括城市列表，时刻表和旅客列表：

```
for (auto city : cityList) {
    delete city;
}
for (auto veh : schedule) {
    delete veh;
}
for (auto ins : instanceList) {
    delete ins;
}
```

2. 系统必须关闭子线程，包括时钟线程和旅客状态刷新线程：

```
delete sysClock;
```

```
2.  delete travelStatusRefresher;
```

3. 系统还应关闭日志文件：

```
1.  logFile->close();
2.  delete logFile;
```

2.3.3 旅行的创建

当用户通过图形界面创建一个旅行并且输入合法时，GUI 会封装并传递一个 InputData 类型的对象，作为旅行请求信息，该信息即可递交控制层处理。

```
bool Control::addTravelInstance(InputData& input_);
```

控制层调用系统核心算法（后文详细阐述）：

```
1.  Plan* plan = Algorithm.generatePlan(input_.beginID, input_.endID,
2.  (input_.beginTime % 24), lastTime, finalDangerIndex, input_.limitTime);
```

系统会检查算法层的输出结果，若旅行方案非空，则创建旅行，在日志中做相应的记录，并返回 true 至交互层：

```
1.  if (plan) {
2.      newInstance = new TravelInstance(ID, input_.request, input_.beginID, inp
    ut_.endID,
3.          input_.beginTime, lastTime, finalDangerIndex, plan, input_.limitTime
    );
4.      instanceNum++;
5.      instanceList.push_back(newInstance);
6.      *logFile << "当前时间: " << System.sysTimeNormal << "\t 新建旅行实
    例: " << newInstance->toStringLog() << std::endl;
7.          return true;
8.  }
```

如果未找到旅行方案，则返回 false 至交互层，并在日志中做相应的记录：

```
1.  *logFile << "当前时间: " << System.sysTimeNormal << "\t" << "尝试创建旅
    行: " <<
2.      System.IDtoCity(input_.beginID)->getName() << "->" << System.IDtoCity(in
    put_.endID)->getName() <<
3.      ", 出发时间: " << input_.beginTime << " 时" << ", 限
    时: " << input_.limitTime << " 小时" <<
4.      ", 旅行限时过短，未找到合适的旅行方案" << std::endl;
5.  return false;
```

其执行流程总结如下：

2.3.4 日志记录系统

为了便于调试和分析，程序特引入日志记录/输出系统，日志文件记录以下事件：

1. 系统级事件

①系统启动成功；

②数据文件的初始化状态；

③系统退出。

```
================================================================================
系统启动
================================================================================
初始化城市列表成功
====================
初始化时刻表成功
================================================================================
```

2. 旅客事件

①旅客创建了一个旅行，若成功，则记录旅行创建时间、旅客编号、旅行类型以及旅行起点/终点；若失败，记录失败原因；

②旅客的状态发生变化，每小时记录一次。若旅客在候车，记录所在城市以及剩余等待时间；若旅客在乘车，记录所乘车次以及剩余乘车时间；

③旅客到达终点后，旅客状态不再更新，系统不再记录该旅客后续状态信息。

```
当前时间: 17    新建旅行实例: ID: 0, 类型: 不限时, 起点: 北京, 终点: 郑州
当前时间: 17    旅行 ID: 0, 当前位置: 北京(候车), 还需等待 3 小时
当前时间: 17    尝试创建旅行: 北京->郑州, 出发时间: 17 时, 限时: 0 小时, 旅行限时过短, 未找到合适的旅行方案
当前时间: 17    新建旅行实例: ID: 1, 类型: 不限时, 起点: 北京, 终点: 郑州
当前时间: 18    旅行 ID: 0, 当前位置: 北京(候车), 还需等待 2 小时
当前时间: 18    旅行 ID: 1, 当前位置: 北京(候车), 还需等待 2 小时
当前时间: 19    旅行 ID: 0, 当前位置: 北京(候车), 还需等待 1 小时
当前时间: 19    旅行 ID: 1, 当前位置: 北京(候车), 还需等待 1 小时
当前时间: 20    旅行 ID: 0, 当前位置: C2073 次汽车[北京->天津], 还需乘坐 2 小时
当前时间: 20    旅行 ID: 1, 当前位置: C2073 次汽车[北京->天津], 还需乘坐 2 小时
```

2.3.5 其它辅助函数

1. 城市编号转换函数 `IDtoCity()`

该函数的功能是将城市编号转换为对应的城市指针，因为城市指针在城市列表中按序存放，因此可以做到直接转换：

```
City* Control::IDtoCity(CityID id_) {
    CityList::iterator iter = System.cityList.begin();
    if (id_ < cityList.size()) {
        return *(iter + id_);
    }
    return nullptr;
}
```

其中为了安全性，系统会检查编号的边界。

2. 等待时间获取函数 `getWaitTime()`

该函数用于计算两趟临近车次间的候车时间，需要考虑候车期间是否跨天。如果不跨天，直接相减即可；如果跨天，相减后结果为负数，需要加 24 进行修正（因为候车时间不会超过 23 小时）。

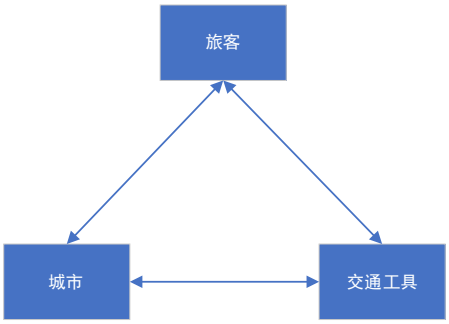
```
// 不跨天，直接相减
if (lastVehArriveTime_ <= nextVehStartTime_) {
    return nextVehStartTime_ - lastVehArriveTime_;
}
// 跨天，进行模运算
return nextVehStartTime_ - lastVehArriveTime_ + 24;
```

3. 系统暂停/继续；获取系统暂停状态

系统暂停/继续函数通过修改 `sysPause` 的值来控制系统时间。

2.4 数据管理模块 DataStructure.cpp

本系统中的三大核心对象即为城市（City）、交通工具（Vehicle）和旅客（TravelInstance）。三者之间相互依存：



2.4.1 城市

城市具有四大属性：城市编号（`ID`）、城市名称（`name`）、风险类型（`kind`）和坐标（`pos`）：

```
CityID ID;                  // 城市编号
CityName name;              // 城市名称
CityKind kind;              // 城市风险类型
QPointF pos;                // 城市坐标
```

城市唯一的类方法即通过停留时间计算风险值，函数为 `getCityDangerIndex()`：

```
switch (kind) {
    case LOW:   return stayTime_ * 2;
    case MID:   return stayTime_ * 5;
    case HIGH:  return stayTime_ * 9;
    default:    return 0;
}
```

在这里为了便于计算和表示，将城市单位风险值扩大十倍，待全部计算结束后再缩小十倍进行处理。

2.4.2 交通工具

交通工具有七个属性，分别为编号（ID）、类型（kind）、名称（kind）、指向出发/到达城市的指针（srcCity/dstCity）以及出发/运行时间（startTime/runTime）。

```
VehID ID;                                   // 交通工具编号
VehKind kind;                               // 交通工具种类
VehName name;                               // 交通工具名称
City* srcCity, * dstCity;                   // 交通工具始发站和终到站
Time startTime, runTime;                    // 交通工具开点和运行时间
```

交通工具类主要有如下几个类方法：

1. 计算交通工具到达时间 getArriveTime()

交通工具到达时间是指到达终点的当日 24 小时制时间，其计算方法如下：

```
return (startTime + runTime) % 24;
```

2. 计算本车次的风险值 getVehDangerIndex()

根据要求，交通工具的风险=该交通工具单位时间风险值*该班次起点城市的单位风险值*乘坐时间。因此计算方法如下：

```
switch (kind) {
    case AIRPLANE: return srcCity->getCityDangerIndex(1) * 9 * getRunTime();
    case TRAIN:    return srcCity->getCityDangerIndex(1) * 5 * getRunTime();
    case BUS:      return srcCity->getCityDangerIndex(1) * 2 * getRunTime();
    default:       return 0;
}
```

3. 用于旅行方案展示的描述性字符串 toUIString()

该函数输出本车次的全部主要信息，包括出发城市、出发时间、车次名称、乘坐时间、到达时间以及到达城市，输出格式类型为 std::string。

2.4.3 旅客

旅客是数据管理模块中属性最多的一个类，列举如下：

```
TravelID ID;                                // 旅客编号
PlanKind kind;                              // 旅行实例类型
City* beginCity, * endCity;                 // 旅行起点和旅行终点
Time limitTime, beginTime, lastTime;        // 限时、开始时间和持续时间
Plan plan;                                  // 旅行方案
DangerIndex dangerIndex;                    // 旅行总风险值
std::vector<Time> keyTime;                  // 旅行关键时间节点序列
```

其中有两点需要说明：

1. 旅行方案 plan

旅行方案的本质是一个车次序列，唯一反映了旅客的行程安排。

2. 关键时间节点序列 keyTime

关键时间节点序列是旅客关键时间发生的时刻的有序序列，其中关键事件如下：

①旅客离开某城市（开始乘车）

②旅客到达某城市（下车）

本序列将这些关键事件发生的事件按顺序存储到时间向量，便于旅行模拟。

旅客类的关键类方法如下：

1. 旅客构造函数 TravelInstance()

当且仅当找到合适的旅行方案，便会创建旅客，创建旅客的第一个任务便是根据旅行方案 plan 生成关键时间节点序列 keyTime。此时只需在第一次发车的基础上，交替累加乘车时间和候车时间，便可生成关键时间节点序列 keyTime。

```
// 根据生成的旅行方案获取关键时间节点序列
for (auto vehIter = plan.begin(); vehIter != plan.end(); vehIter++) {
    // 定义每班车次的到达时间
    Time originTime;
    // 如果在始发地，特定义为旅行开始时间；否则为实际到达时间
    if ((*vehIter)->getSrcCity() == beginCity) {
        originTime = beginTime % 24;
    }
    else {
        originTime = (*(vehIter - 1))->getArriveTime();
    }
    // 按时间次序存储候车时间与乘车时间
    keyTime.push_back(System.getWaitTime(originTime, (*vehIter)->getStartTime()));
    keyTime.push_back((*vehIter)->getRunTime());
}
// 累加，变成方案内每个关键时间节点的时刻
for (auto timeIter = keyTime.begin() + 1; timeIter != keyTime.end(); timeIter++) {
    *timeIter += *(timeIter - 1);
}
// 在累加的基础上再分别加上旅行起始时间，以生成全局时间序列
for (auto& time : keyTime) {
    time += beginTime;
}
```

2. 旅客状态获取函数 `getStatusUI()/getStatusLog()`

两个函数都用于旅客状态获取旅客状态，只不过一个用于 UI 展示，一个用于日志记录，在此只阐述其一。

首先系统检查当前系统时间是否落在 `keyTime` 序列范围中，若小于 `keyTime` 的最小值，则说明旅行尚未开始；若大于 `keyTime` 的最大值，则说明旅客已经到达终点：

```
// 如果当前时间小于旅行开始时间，说明该旅行自定义了开始时间，且还未开始
if (curTime_ < beginTime) {
    statusStr = "\n" + beginCity->getName() + "（尚未开始）\n";
}
// 如果当前时间大于旅行结束时间，说明已到达目的地
else if (curTime_ >= *(keyTime.end() - 1)) {
    statusStr = "\n" + endCity->getName() + "（终点）\n";
}
```

如果系统时间是上述两种情况外的其它情况，则说明旅客正在旅行。现在考虑 `keyTime` 序列的特点：若将关键时间节点值从 0 开始按发生顺序编号，则偶数-奇数编号之间的时刻，旅客一定处于乘车状态；而奇数-偶数编号之间的时刻，旅客一定处于候车状态。因此有如下遍历算法：

```
for (int i{0}; i < keyTime.size(); i++) {
    // 寻找系统当前时间落在那个区间内
    if (curTime_ < *(keyTime.begin() + i)) {
        Vehicle* curVeh = *(plan.begin() + i / 2);
        // 如果落在偶数区间，说明在候车；否则在乘车
        if (0 == i % 2) {
            （候车）
        }
        else {
            （乘车）
        }
        break;
    }
}
```

3 算法层

3.1 算法的原理和执行

3.1.1 模型的构建

本算法所依托的数据结构主要为有向图，实现方法为邻接表：

```
// 邻接表边结点
class arcNode {
public:
    int verID;                  // 终点顶点编号
    DangerIndex dangerIndex;    // 边权值，此处为风险值
    arcNode* nextArc;           // 与该边具有相同起点顶点的下一条边
    arcNode() = delete;
    arcNode(int verID_, DangerIndex dangerIndex_)
        : verID(verID_), dangerIndex(dangerIndex_), nextArc(nullptr) {
    }
};
// 邻接表顶点结点
class verNode {
public:
    int verID;                  // 顶点编号
    arcNode* firstArc;          // 以该顶点为起点的第一条边
};
```

其中，有向图的建模方式如下：

1. 设城市数目为 M，理论最大旅行持续时间为 N（单位：小时。算法默认设置为 10 天），建立有向图 $G=(V,E)$，其中结点数为 $M\times N$，即为每个城市的每个时刻定义一个结点，并设 `V[i][j]` 表示城市 i 在时刻 j 所代表的结点。其中城市数目根据文件内容动态调整：

```
int nodeNum;                    // 风险图总顶点数
int** nodeID;                   // 顶点索引数组
verNode* verList;               // 顶点列表
```

结点的初始化如下：

```
// 初始化结点编号
for (CityID cityID{0}; cityID < System.cityNum; cityID++) {
    for (Time time{0}; time < INS_TIME_MAX; time++) {
        nodeID[cityID][time] = cityID * INS_TIME_MAX + time;
    }
}
// 初始化邻接表
for (int i{0}; i < nodeNum; i++) {
    verList[i].verID = i;
    verList[i].firstArc = nullptr;
}
```

2. 对于任意城市 A，若其单位风险值为 W，对于任意的 $0\le i<N$，有一条以结点 `V[A][i]` 为起点，以结点 `V[A][i+1]` 为终点的边，且权值为 W。

```
// 在同一城市相邻时刻添加一条权值为单位风险的边
for (CityID cityID{0}; cityID < System.cityNum; cityID++) {
    for (Time time{0}; time < INS_TIME_MAX - 1; time++) {
```

```
            addArcNode(nodeID[cityID][time], nodeID[cityID][time + 1],
                System.IDtoCity(cityID)->getCityDangerIndex(1));
        }
    }
```

3. 对于任一交通工具，若其于当日 t 时刻在城市 A 出发，运行 l 小时后到达城市 B，乘坐风险为 W，则对于任意的 $0 \le j < N/24$，有一条以结点 `V[A][t+24*j]` 为起点，以结点 `V[B][t+l+24*j]` 为终点的边，且权值为 W。

```
// 遍历时刻表，添加相对应的边
for (auto veh : System.schedule) {
    Time startTime = veh->getStartTime(),
        arriveTime = veh->getStartTime() + veh->getRunTime();
    CityID srcID = veh->getSrcCity()->getID(),
        dstID = veh->getDstCity()->getID();
    // 注意车次的周期性
    while (arriveTime < INS_TIME_MAX) {
        addArcNode(nodeID[srcID][startTime], nodeID[dstID][arriveTime],
            veh->getVehDangerIndex());
        startTime += 24;
        arriveTime += 24;
    }
}
```

3.1.2 问题的求解

1. 使用堆优化的最短路径算法 Dijkstra

相比传统 Dijkstra 算法，堆优化算法首先用 `dangerIndex` 数组记录起点到每个结点的最短路径，再用一个集合 T 保存已经找到最短路径的结点，然后从 `dangerIndex` 数组中选择最小值，则该值就是源点 `nodeID[beginID_][beginTime_]` 到该值对应的结点 `topNode` 的最短路径，把该结点 `topNode` 加入到 T 中，并且用结点 `topNode` 当前的最短路径估计值对离开结点 `topNode` 所指向的结点 `curNode` 进行松弛操作，即判断是否有 `dangerIndex[curNode.verID]>dangerIndex[topNode.verID]+w`（w 是连接 `topNode` 与 `curNode` 的边的长度），若有，则更新 `dangerIndex[curNode.verID]`。这样不断从 `dangerIndex` 数组中选择最小值对应的结点来进行松弛操作，直至所有结点都在集合 T 中为止。

```
// 初始化
for (int i{0}; i < nodeNum; i++) {
    dangerIndex[i].verID = i;
    // 估算距离置 DANGER_MAX
    dangerIndex[i].dangerIndex = DANGER_MAX;
```

```
    // 每个顶点都无父亲节点
    nodePath[i] = -1;
    // 都未找到最短路
    nodeVisited[i] = false;
}
// 源点到源点最短路权值为 0
dangerIndex[nodeID[beginID_][beginTime_]].dangerIndex = 0;
// 压入优先队列中
nodeQueue.push(dangerIndex[nodeID[beginID_][beginTime_]]);
// 队列空说明完成了操作
while (!nodeQueue.empty()) {
    // 取最小估算距离顶点
    tmpNode topNode = nodeQueue.top();
    nodeQueue.pop();
    int topID = topNode.verID;
    // 避免对已访问节点的不必要操作
    if (nodeVisited[topID]) {
        continue;
    }
    nodeVisited[topID] = true;
    arcNode* curNode = verList[topID].firstArc;
    // 找所有与他相邻的顶点，进行松弛操作，更新估算距离，压入队列
    while (nullptr != curNode) {
        int curID = curNode->verID;
        if (!nodeVisited[curID] &&
            dangerIndex[curID].dangerIndex > dangerIndex[topID].dangerIndex
    + curNode->dangerIndex) {
            dangerIndex[curID].dangerIndex = dangerIndex[topID].dangerIndex
    + curNode->dangerIndex;
            nodePath[curID] = topID;
            nodeQueue.push(dangerIndex[curID]);
        }
        curNode = curNode->nextArc;
    }
}
```

2. 对于非限时旅行策略

设理论旅行最大持续时间为 N，若旅客在当日 t 时刻，从城市 A 出发，欲到达城市 B。则对于任意的 $t<i\leqslant N$，系统使用 Dijkstra 算法，求出源结点 `V[A][t]`到目的结点 `V[B][i]` 的最短路径集合 P，且每条路径的权值为 W_i。则 Min(W_i)即为算法所求得的全局最优解。

```
// 初始化最短距离
minDis = dangerIndex[nodeID[endID_][beginTime_]].dangerIndex;
minNode = nodeID[endID_][beginTime_];
```

```
    // 根据时间限制，在已求出的最短路径值中筛选风险最小者
    for (Time time{beginTime_}; time < INS_TIME_MAX; time++) {
        int i = nodeID[endID_][time];
        if (minDis > dangerIndex[i].dangerIndex) {
            minDis = dangerIndex[i].dangerIndex;
            minNode = i;
        }
    }
```

3. 对于限时旅行策略

若旅客在当日 t 时刻，从城市 A 出发，欲到达城市 B，旅行限时为 L。则对于任意的 $t<i\leqslant t+L$，系统使用 Dijkstra 算法，求出源结点 `V[A][t]`到目的结点 `V[B][i]`的最短路径集合 P，且每条路径的权值为 W_i。则 Min(W_i)即为算法所求得的全局最优解。

```
    // 初始化最短距离
    minDis = dangerIndex[nodeID[endID_][beginTime_]].dangerIndex;
    minNode = nodeID[endID_][beginTime_];
    // 根据时间限制，在已求出的最短路径值中筛选风险最小者
    for (Time time{beginTime_}; time < min(beginTime_ + limitTime_, INS_TIME_MAX
    ); time++) {
        int i = nodeID[endID_][time];
        if (minDis > dangerIndex[i].dangerIndex) {
            minDis = dangerIndex[i].dangerIndex;
            minNode = i;
        }
    }
```

4. 最短路径的预处理

首先根据 `nodePath` 数组生成路径：

```
    bestPath.push_back(minNode);
    int curNode = minNode;
    // 迭代搜索父节点，生成路径
    while (nodePath[curNode] != -1) {
        bestPath.push_back(nodePath[curNode]);
        curNode = nodePath[curNode];
    }
    bestPath.push_back(nodeID[beginID_][beginTime_]);
```

并将路径存入容器：

```
    for (auto bestNode : bestPath) {
        if (node2CityID(bestNode) != currentCityID) {
            cityPath.push_back(bestNode);
            currentCityID = node2CityID(bestNode);
        }
    }
```

由上述最短路径算法所求得的结果为逆序路径，还需转变为正序路径：

```
// 逆序路径变换为正序路径
reverse(cityPath.begin(), cityPath.end());
```

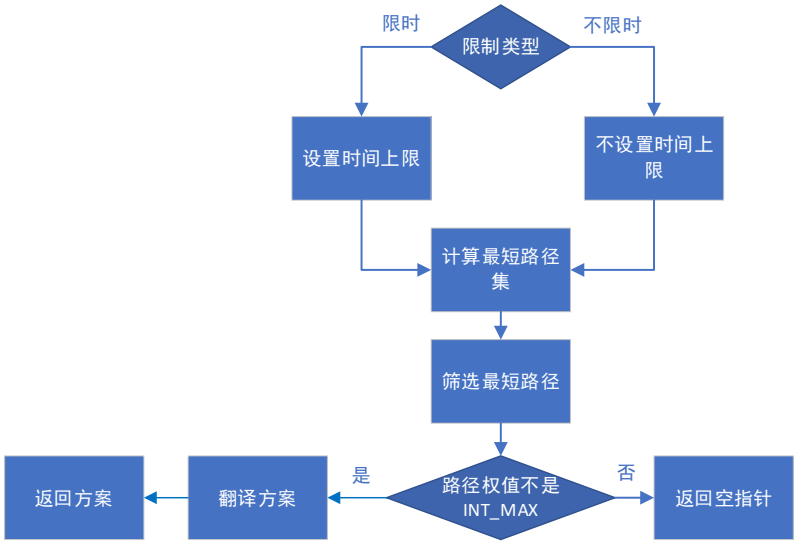
5. 最短路径的翻译

最短路径求出后，仅为一组结点序列，并且每个城市对应不同时间的多个结点，因此还需翻译为具体城市和车次：

```
// 根据路径寻找对应的车次
for (auto cityIter = cityPath.begin(); cityIter != (cityPath.end() - 1); cityIter++) {
    City* srcCity = System.IDtoCity(node2CityID(*cityIter)),
        * dstCity = System.IDtoCity(node2CityID(*(cityIter + 1)));
    Time startTime = node2Time(*cityIter) % 24;
    // 若已找到车次，则不继续遍历
    for (auto veh : System.schedule) {
        if (veh->getSrcCity() == srcCity &&
            veh->getDstCity() == dstCity &&
            veh->getStartTime() == startTime) {
            plan->push_back(veh);
            break;
        }
    }
}
```

以上工作执行完毕后，算法已经生成了旅行计划 `plan`。

算法整体执行流程如下：



3.2 算法分析

在常规情况下，限时旅行策略和非限时旅行策略可共用一套算法。以下综合讨论：

3.2.1 算法空间复杂度分析

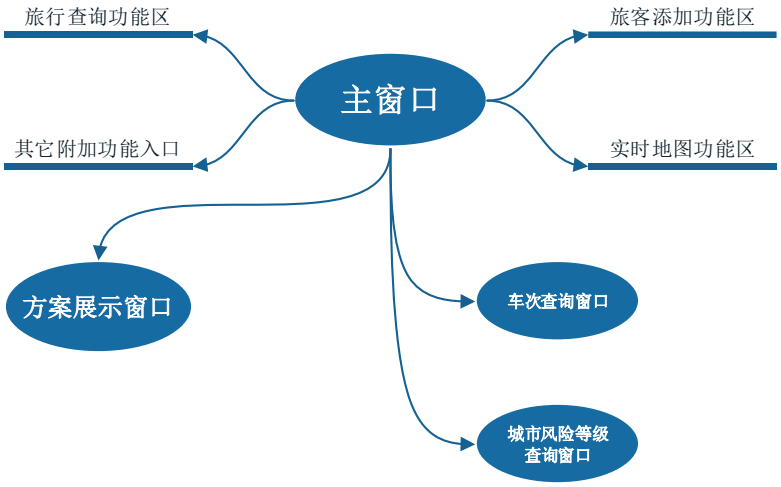
有向图采用邻接表存储，设 M 为城市数目，T 为旅行最大持续时间，N 为时刻表车次数目，考虑到结点索引和优先队列所需空间，因此算法空间复杂度为 $O((M+N)T)$。

3.2.2 算法时间复杂度分析

设 M 为城市数目，T 为旅行最大持续时间，N 为时刻表车次数目，则堆搜索最短边复杂度为 $O((M+N)T\log_2((M+N)T))$，因为要搜索单点所到的所有边，所以堆中有 $MT+NT/24$ 个点代表边权，所以在计算最短路时复杂度为 $O((M+N)T\log_2((M+N)T))$，而要初始化堆所用时间为 $O(MT\log_2((M+N)T))$，则总时间复杂度为 $O((M+N)T\log_2((M+N)T))$。

4 交互层

交互层和核心即用户图形界面，是本软件提供给用户操作的接口。交互层结构图如下：

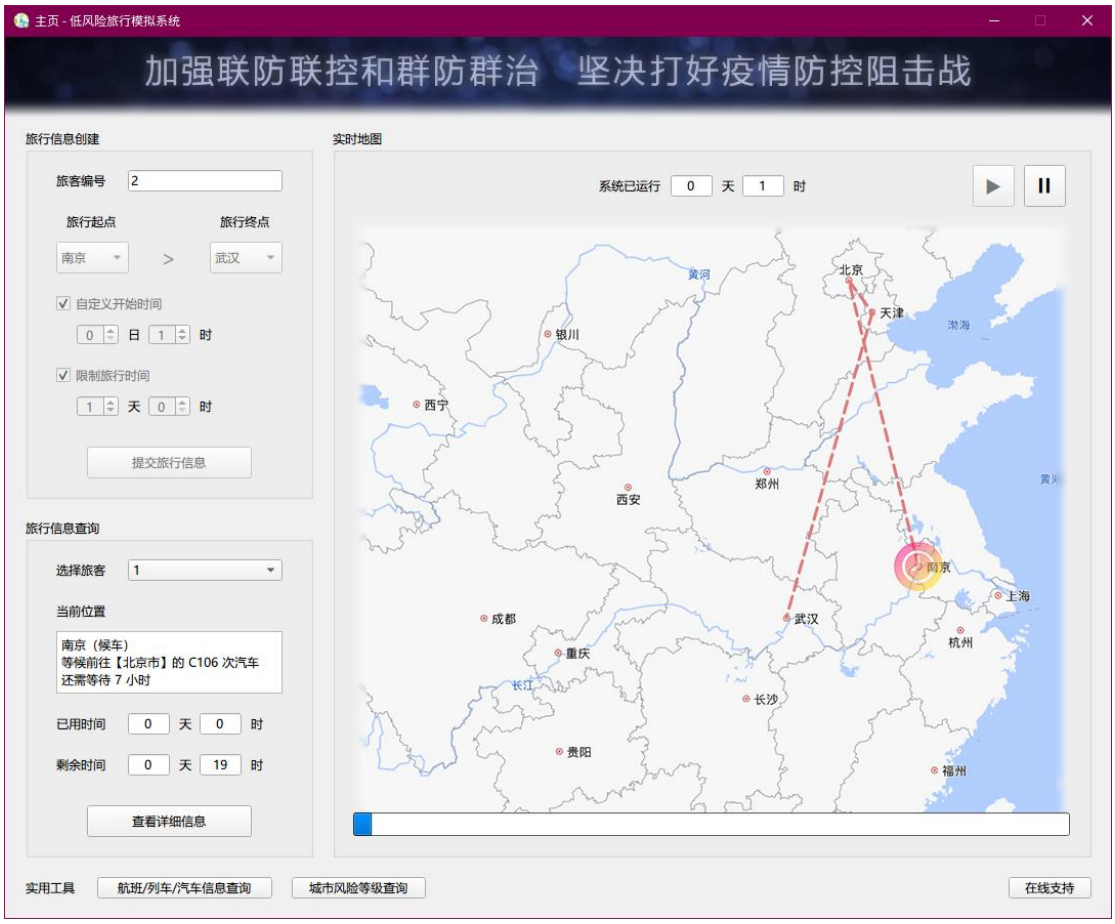


4.1 主窗口 COVID_19_Travel_System.cpp

主窗口提供如下功能：

1. **旅客添加**
2. **旅行信息查询**
3. **实时地图展示**
4. **附加功能入口**

其界面设计如下：



其中当界面启动时，系统时钟自动暂停，等待用户输入。

4.1.1 旅客添加功能区

旅客添加功能区提供旅行创建功能，应课设要求，该功能区仅当时钟暂停时才可用。该功能区界面如下：

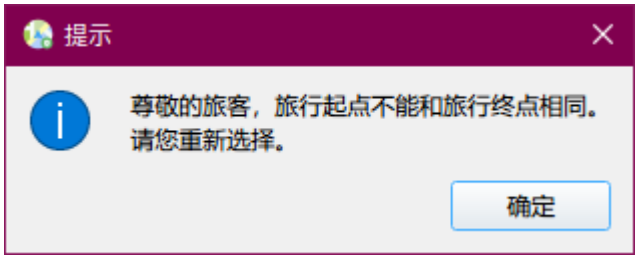
当用户欲创建旅行时，首先需确保系统时钟处于暂停状态。否则按钮不可用。用户选择城市，填写相关可选信息之后，即可点击“提交旅行信息”按钮。

当用户点击“提交旅行信息”按钮时，系统会做如下判断：

1. 若用户选择的旅行起点城市和终点城市相同时，系统会弹出提示，提示用户重新选择：

```
1.  if (ui.comboBoxBeginID->currentIndex() == ui.comboBoxEndID->currentIndex())
    {
2.      QString dlgTitle = QString::fromLocal8Bit("提示");
3.      QString strInfo = QString::fromLocal8Bit("尊敬的旅客，旅行起点不能和旅行终
    点相同。\n 请您重新选择。");
4.      QMessageBox::information(this, dlgTitle, strInfo, QString::fromLocal8Bit
    ("确定"));
5.  }
```

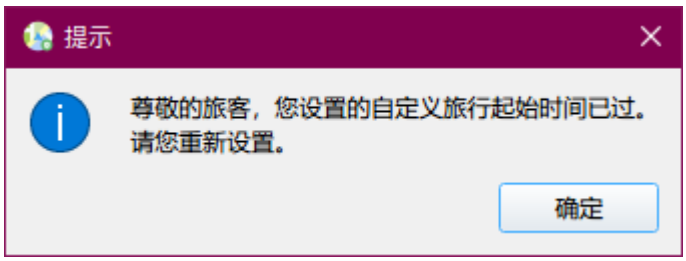
界面表现如下：



2. 若用户设置的自定义旅行时间小于系统当前时间，系统会弹出提示，提示用户重新设置：

```
1.  // 如果用户勾选了"自定义开始时间"复选框，但是输入的时间不合法，则驳回请求
2.  if (ui.checkBoxCustomizeBeginTime->isChecked() && customizeBeginTime < Syste
    m.sysTimeNormal) {
3.      QString dlgTitle = QString::fromLocal8Bit("提示");
4.      QString strInfo = QString::fromLocal8Bit("尊敬的旅客，您设置的自定义旅行起
    始时间已过。\n 请您重新设置。");
5.      QMessageBox::information(this, dlgTitle, strInfo, QString::fromLocal8Bit
    ("确定"));
6.  }
```

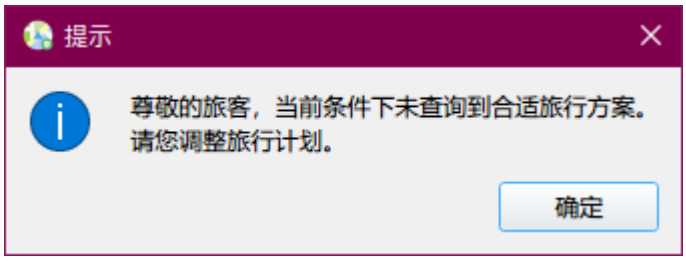
界面表现如下：



3. 当算法没有找到合适的旅行方案，系统会弹出提示：

```
// 如果旅行创建成功，则处理 UI 操作；否则弹出提示框，要求旅客修改信息
if (System.addTravelInstance(*uiInput)) {
      ...（略）
}
else {
    QString dlgTitle = QString::fromLocal8Bit("提示");
    QString strInfo = QString::fromLocal8Bit("尊敬的旅客，当前条件下未查询到合适旅行方案。\n 请您调整旅行计划。");
    QMessageBox::information(this, dlgTitle, strInfo, QString::fromLocal8Bit("确定"));
}
```

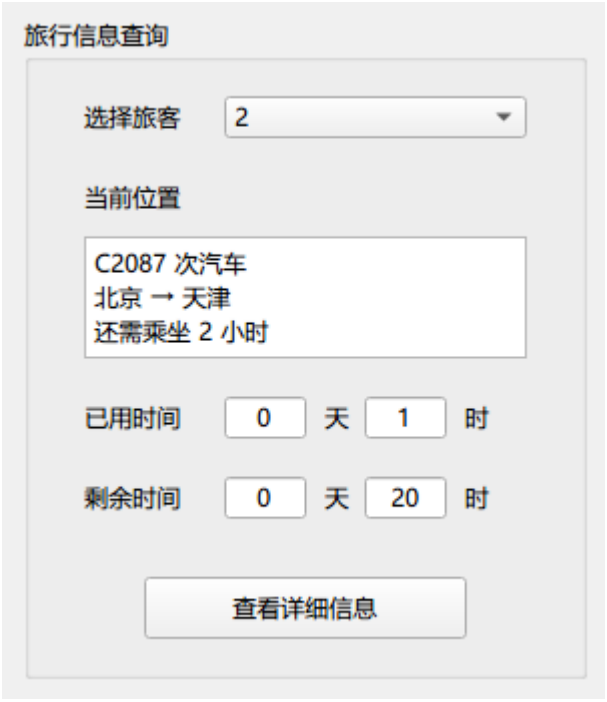
界面表现如下：



若找到旅行方案，则弹出旅行方案展示窗口（见 4.2 节）。

4.1.2 旅行查询功能区

旅行查询窗口用于展示旅客旅行信息，其中用户可以通过旅客选择下拉框选择需要展示的旅客：



旅客信息每 10s 刷新一次，通过发射界面刷新信号：

```
// 若系统不处于暂停状态，以中精度时钟频率发射刷新主界面的信号
void COVID_19_Travel_System::startUIRefresher() {
    while (true) {
        Sleep(CLOCK_FREQ_MEDIUM);
        if (!System.isPaused()) {
            sendRefreshSignal();
        }
    }
}
```

其中 `sendRefreshSignal()` 函数与界面刷新函数 `refreshUI()` 连接：

```
connect(this, &COVID_19_Travel_System::sendRefreshSignal, this, &COVID_19_Travel_System::refreshUI);
```

以刷新界面时间以及旅客信息：

```
// 刷新主界面，调用刷新时钟显示和刷新当前旅客信息的方法
void COVID_19_Travel_System::refreshUI() {
    refreshUITime();
    refreshInfo();
}
```

具体刷新内容为：旅客位置，旅行已用时间和旅行剩余时间：

```
ui.textBrowserStatus->setText(QString::fromLocal8Bit(ins->getStatusUI(System.sysTimeNormal).data()));
// 旅行开始之后才刷新；否则显示 "-"
if (System.sysTimeNormal >= ins->getBeginTime()) {
    ui.lineEditUsedDay->setText(QString::number(min((System.sysTimeNormal - ins->getBeginTime()), ins->getLastTime()) / 24));
    ui.lineEditUsedHour->setText(QString::number(min((System.sysTimeNormal - ins->getBeginTime()), ins->getLastTime()) % 24));
    ui.lineEditRestDay->setText(QString::number(max((ins->getLastTime() + ins->getBeginTime() - System.sysTimeNormal), 0) / 24));
    ui.lineEditRestHour->setText(QString::number(max((ins->getLastTime() + ins->getBeginTime() - System.sysTimeNormal), 0) % 24));
}
```

而在旅行查询功能区，用户仍可以通过点击“查看详细信息”按钮回顾旅行方案。

4.1.3 实时地图功能区

实时地图功能区用动画实时展示旅客的旅行状态，该模块以最高的刷新频率获取旅客状态，并以 60fps 的频率刷新：

```
rePaintSignal = new QTimer;
rePaintSignal->start(1000 / 60);
```

```
3. connect(rePaintSignal, SIGNAL(timeout()), this, SLOT(update()));
```

刷新时，模块先获取当前旅客序号，以显示相应旅客信息：

```
1. // 旅行列表非空时，则绘制当前选择的旅行
2. if (!System.instanceList.empty()) {
3.     TravelInstance* curIns = *(System.instanceList.begin() + MainWindow->get
   CurInsIndex());
4. ...（略）
5. }
```

然后将旅客途径城市连线：

```
1. // 根据途径城市绘制城市之间的连线
2. for (auto veh : *(curIns->getPlan())) {
3.     painter.setPen(pen);
4.     painter.drawLine(veh->getSrcCity()->getLinePos(), veh->getDstCity()->get
   LinePos());
5. }
```

随后根据旅客位置和旅客状态绘制旅客图标：

```
1. // 根据旅客状态和位置绘制图标
2. painter.drawPixmap(getIconPos(), getIconKind());
```

旅客位置的刷新原理如下：其大体原理与旅客信息查询文字输出方式类似，但其使用的系统时钟为“高精度时钟”，每 100ms 刷新一次，高精度时间通过标准时间扩大 100 倍得到。

若旅客静止，则无需刷新旅客位置，否则需要刷新旅客位置：

```
1. // 时间节点序号为奇数，表示正在乘车
2. else {
3.     curIconPos = curVeh->getSrcCity()->getIconPos();
4.     // 运动中则需要计算坐标增量
5.     curIconPos += getIconDeltaPos(*(curKeyTime->begin() + i - 1), curVeh->ge
   tRunTime(),
6.         curVeh->getSrcCity(), curVeh->getDstCity());
7.     // 根据交通工具类型选择对应图标
8.     curInsStatus = static_cast<Status>(curVeh->getKind());
9. }
```

其中坐标变换计算方法如下：

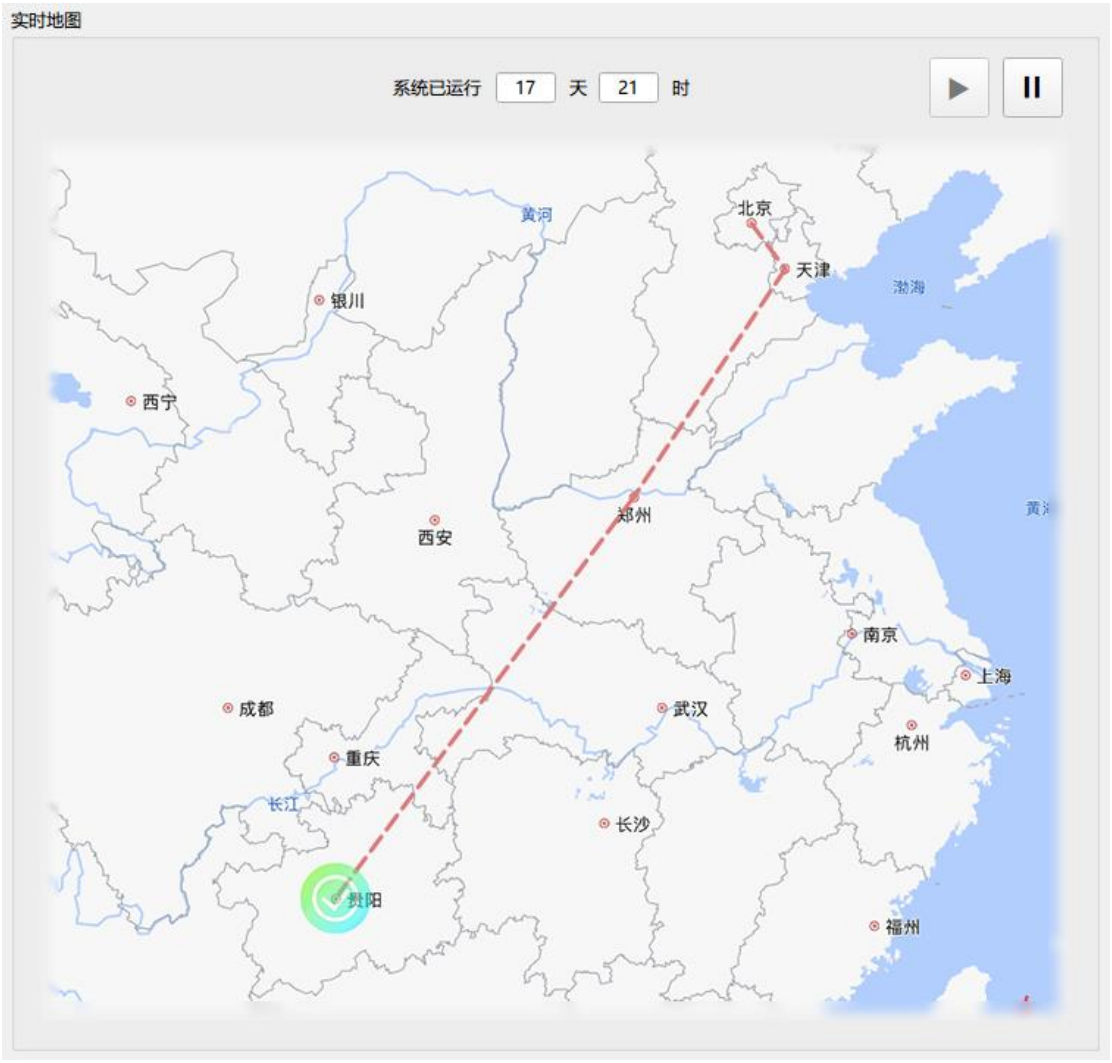
单位时间坐标变化率＝（当前系统时间－当前车次出发时间）／车次运行时间

坐标变化量＝（当前车次到达城市坐标－当前车次出发城市坐标）×单位时间坐标变化率

```
1. // 根据当前车次信息计算下一高精度时刻旅客位置的变化量
2. QPointF Map::getIconDeltaPos(Time startTime_, Time runTime_, City* srcCity_,
    City* dstCity_) {
3.     // 坐标变换比率，为"该车次已用时间 ÷ 车程"
4.     double deltaRatio = (static_cast<double>(System.sysTimePrecise) - static
   _cast<double>(toPreciseTime(startTime_)))
```

```
        / static_cast<double>(toPreciseTime(runTime_));
    // 坐标变化量，为"两地直线距离 × 坐标变换比率"
    double deltaX = (dstCity_->getIconPos() - srcCity_->getIconPos()).x() *
deltaRatio;
    double deltaY = (dstCity_->getIconPos() - srcCity_->getIconPos()).y() *
deltaRatio;
    return QPointF(deltaX, deltaY);
}
```

实时地图界面展示如下：



4.2 方案展示窗口 PlanViewer.cpp

一旦旅行创建成功，便会自动弹出旅行方案展示窗口，并自动暂停系统时间：

```
// 准备弹出旅行结果查询窗口
PlanViewer* planViewer = new PlanViewer(*(*(System.instanceList.end() - 1)))
;
planViewer->setWindowFlags(planViewer->windowFlags() & ~Qt::WindowMinMaxButt
onsHint);
```

```
4.  planViewer->setWindowModality(Qt::ApplicationModal);
5.  planViewer->show();
```

窗口展示如下：

其中窗口中展示了旅行基本信息和具体的旅行方案，用户点击“确定”按钮后，系统时钟继续运行，开始旅行模拟。

当用户点击“旅行查询功能区”中的“查看详细信息”按钮时，也可以弹出本窗口，反映当前旅客的旅行方案。此时系统时钟自动暂停，关闭后时钟继续。

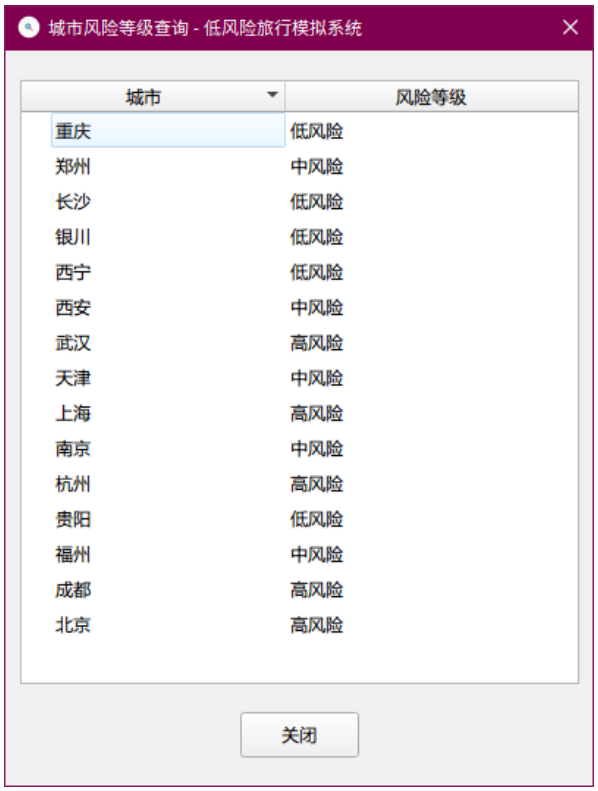
4.3 附加功能

4.3.1 车次查询窗口 VehicleQuery.cpp

便于验收，本程序特提供单独的车次查询功能，用户选择始发站和终到站即可查询相应车次信息：

4.3.2 城市风险等级查询窗口 CityDangerQuery.cpp

本软件也提供城市风险等级查询功能：



4.3.3 在线支持

当用户使用本软件出现问题，即可点击“在线支持”按钮，前往在线支持官网：