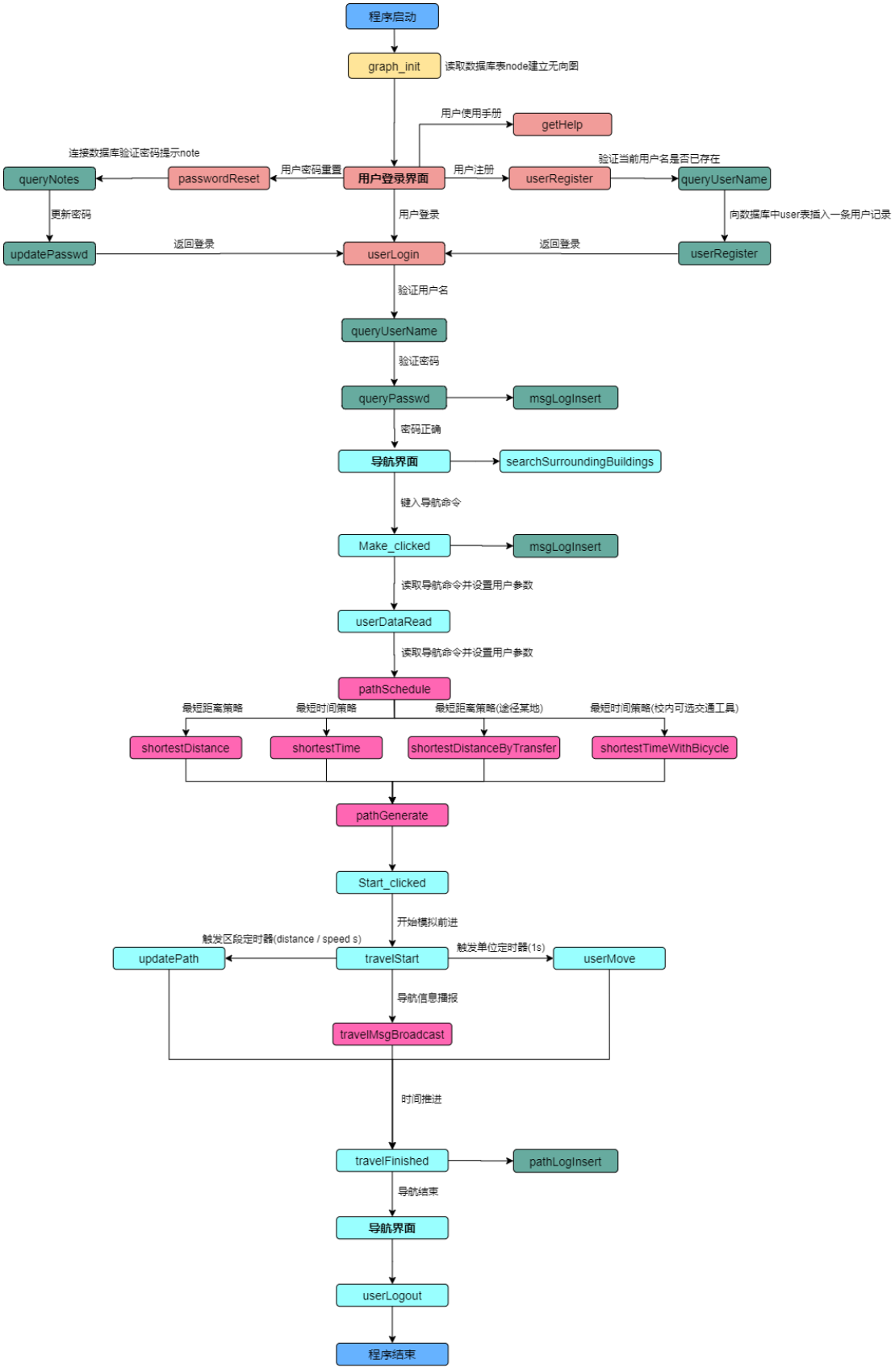
详细设计目录

一、模块划分及流程图

1.模块划分

模块名	模块功能
导航主界面 campusGuide	①命令分析功能的实现； ②模拟时间推进； ③根据制定的路线，模拟用户移动； ④图形化地图的展示； ⑤其他必要的参数设置、数据读取、状态变化函数。
功能算法实现 guide	①路线规划函数的实现； ②查询周边建筑物等扩展功能的实现。
数据库管理 database	①连接数据库读取用户名、密码； ②连接数据库写入键入命令日志 msgLog 与路径日志 pathLog。
登录注册界面 manage	①用户登录、注册、重置密码； ②系统使用帮助手册、功能说明。
公有设计 public	①声明结构、全局变量与宏； ②读取数据库中存储的静态数据建立无向图及邻接矩阵； ③提供一些常用的函数，如计算两点间距离等函数的声明与实现。

2.总体结构流程图


程序启动
graph_init
读取数据库表node建立无向图
用户使用手册
getHelp
连接数据库验证密码提示note
queryNotes
passwordReset
用户密码重置
用户登录界面
用户注册
userRegister
验证当前用户名是否已存在
queryUserName
更新密码
用户登录
向数据库中user表插入一条用户记录
updatePasswd
返回登录
userLogin
返回登录
userRegister
验证用户名
queryUserName
验证密码
queryPasswd
msgLogInsert
密码正确
导航界面
searchSurroundingBuildings
键入导航命令
Make_clicked
msgLogInsert
读取导航命令并设置用户参数
userDataRead
读取导航命令并设置用户参数
pathSchedule
最短距离策略
最短时间策略
最短距离策略(途径某地)
最短时间策略(校内可选交通工具)
shortestDistance
shortestTime
shortestDistanceByTransfer
shortestTimeWithBicycle
pathGenerate
Start_clicked
开始模拟前进
触发区段定时器(distance / speed s)
触发单位定时器(1s)
updatePath
travelStart
userMove
导航信息播报
travelMsgBroadcast
时间推进
travelFinished
pathLogInsert
导航结束
导航界面
userLogout
程序结束

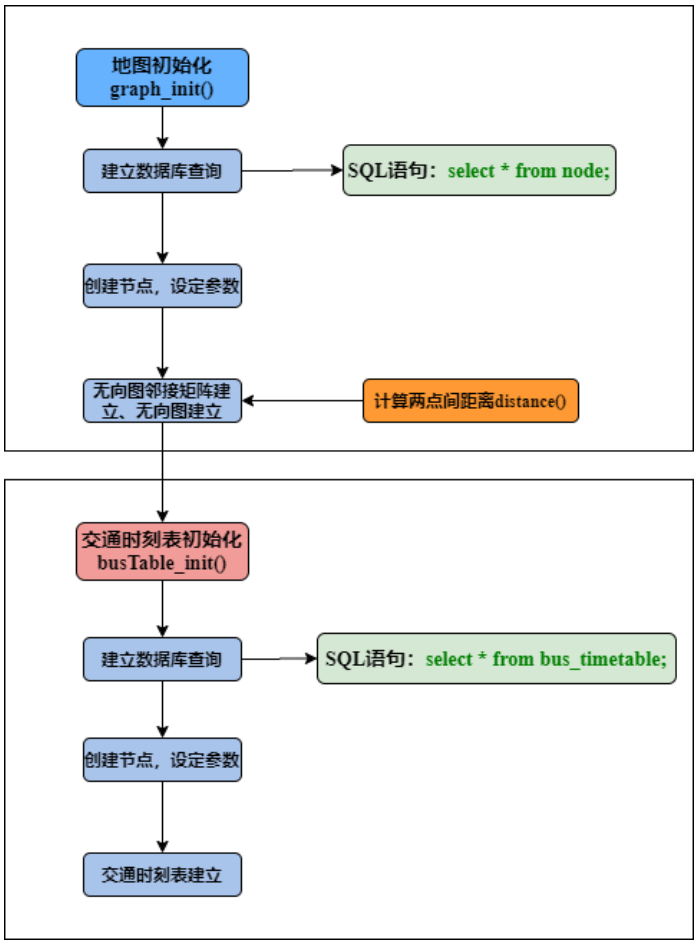
二、各模块流程及调用关系描述

1.公有设计模块 public

①系统启动，开始初始化。首先连接数据库，从数据库表 node 中读取地图节点数据建立无向图，并设置各邻接矩阵。

②同样地，连接数据库从库表 bus_timetable 中读取数据建立交通时刻表。

函数原型	函数功能
void graph_init()	初始化无向图
int distance(unsigned A, unsigned B)	返回[A, B]间的距离
int distance(QPoint &A, QPoint &B)	重载：直接计算两坐标间距离
void busTable_init()	初始化交通时刻表



2.数据库管理模块 database

该模块只定义了一个 database 类及类成员函数，没有实际的动作。因此仅给出其中定义的函数接口，具体操作等后续引用时描述。

函数原型	函数功能
static database *getDatabase()	若数据库实例已存在则返回，不存在则创建一个实例再返回。
bool userRegister(const QString &userName, const QString &passwd, const QString &notes)	输入用户名、密码、提示来尝试注册一个用户。注册成功则向数据库表 user 中插入一条记录同时返回 true，失败则返回 false 与消息框。
bool queryUserName(const QString &userName)	登录或注册时用于检测数据库表 user 中是否存在该用户，存在 true，否则 false。
bool queryPasswd(const QString &userName, const QString &passwd)	登录时验证密码是否正确，是则 true，否则 false。
bool queryNotes(const QString &userName, const QString &notes)	重置密码时需要验证该用户名对应的关键字是否正确，是则 true，否则 false。
void updatePasswd(const QString &userName, const QString &passwd, const QString &notes)	重置密码时更新数据库表 user 中的用户密码
static bool msgLogInsert(const QString logTime, const QString userName, const QString userState, const QString logMsg)	用户操作时，向数据库表 msg_log 中插入一条用户操作记录(logTime, username, userState, logMsg)，插入成功 true，失败 false。
static bool pathLogInsert(const QString userName, const QString timeStart, const QString from, const QString trans, const QString to, const QString strategy, const QString pathMsg, const QString timecost)	导航完成时，向数据库表 path_log 中插入一条路径日志记录(userName, timeStart, from, trans, to, strategy, pathMsg, timecost)，插入成功 true，失败 false。
static void getPreferredPath(const QString userName, QList<QString> &commonPath)	从数据库表 user 中获取该用户对应的常用路径，用作偏好设置。
static void updateCommonPath(const QString userName)	导航完成时，更新一次常用路径。
static void updateLastPath(const QString userName, QString src, QString trans, QString dst, QString strategy)	导航完成时，更新一次最近使用路径。

3.用户登录界面 manage

①首先注册用户，需要输入用户名、密码以及用于重置密码的提示，系统会对用户名进行查重，若当前用户名未被注册，则向数据库表 user 中插入一条用户记录。

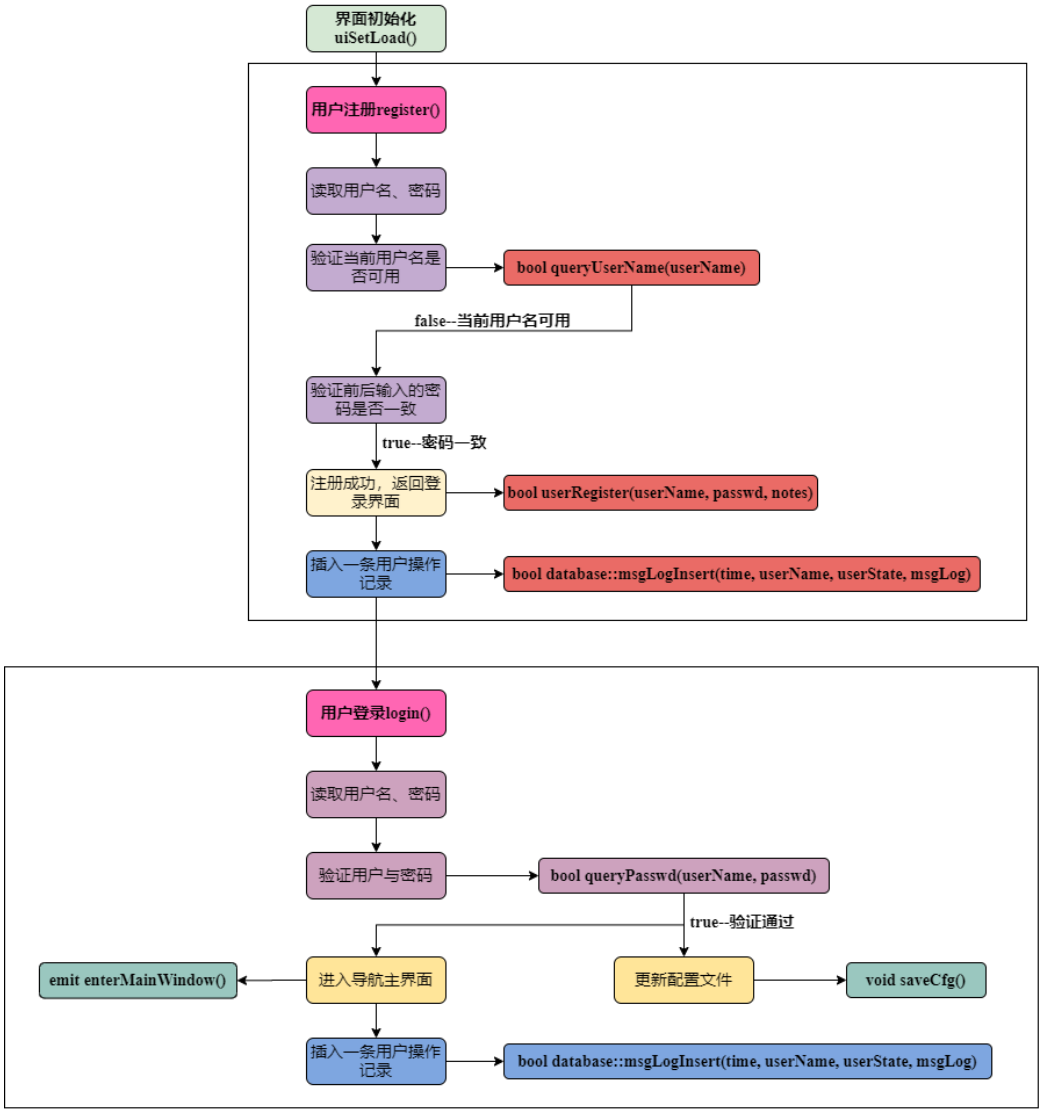
②登录时输入用户名、密码（可以选择记住密码和自动登录），系统验证用户名及密码通过后弹出导航主界面，用户名与密码不匹配时弹出错误对话框。

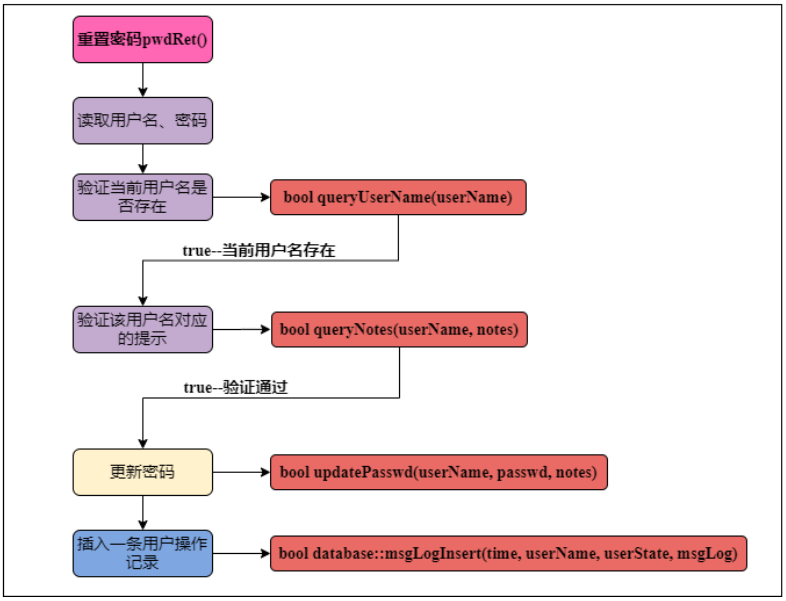
③忘记密码时可重置密码，只需要输入要重置的用户以及对应的提示就可以重置密码。

④系统提供用户使用手册帮助使用。

函数原型	函数功能
void uiSetLoad()	设定界面参数
void on_RP_Register_clicked()	【按钮槽】注册用户时，验证用户名、密码，通过则向数据库表 user 中插入一条记录同时返回登录界面，失败则清空输入并弹出失败对话框。
void enterMainWindow()	【信号】登录验证通过进入主窗口
void on_login_clicked()	【按钮槽】验证输入的用户名、密码是否存在、是否与数据库中保存的一致，是则发出 enterMainWindow()信号，否则弹出错误提示框并清空编辑栏。
void on_FP_reset_clicked()	【按钮槽】重置密码时验证用户名与提示，通过则将新密码更新为密码。
void on_FP_back_clicked();	【按钮槽】返回登录界面。
void on_help_clicked()	【按钮槽】弹出帮助手册。
void switchPage()	【按钮槽】帮助手册的翻页函数。
QWizardPage *createPage1()	帮助手册第一页
QWizardPage *createPage2()	帮助手册第二页
QWizardPage *createPage3()	帮助手册第三页
void on_autoLogin_stateChanged(int arg1)	【按钮槽】当自动登录的选框状态改变时，修改配置文件对应参数，同时设置记住密码框的选中状态；选中自动登录时默认选中记住密码，当取消记住密码时自动登录也随之取消。
void on_remPasswd_stateChanged(int arg1)	【按钮槽】当记住密码的选框状态改变时，修改配置文件对应参数。
void saveCfg()	将记住密码、自动登录的状态变化写回文件
void loadCfg()	读取配置文件用以设置记住密码、自动登录参数
void autoLogin()	【信号】自动登录参数有效，发出信号触发登录按钮

1.所有的按钮槽函数名以功能简写，如on_pushButton_login_clicked() --> login();
2.数据库操作的bool返回值如无特殊说明，都是SQL语句的执行结果，true执行成功，false反之。

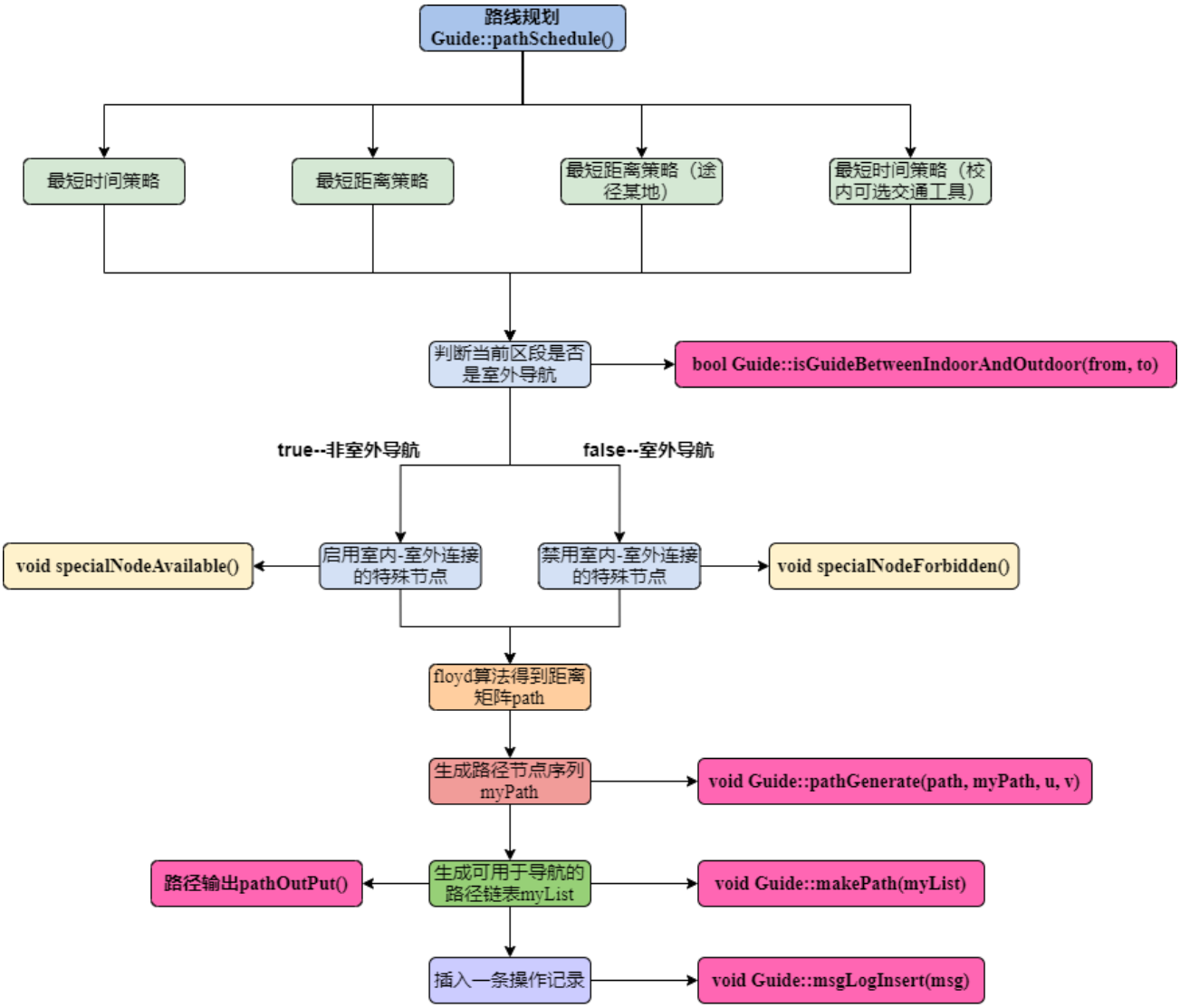




4.功能算法设计 guide

该模块中大部分函数都是用来实现路线规划、模拟导航的，因此以一个路线导航实例[from, to]来说明一次导航路线的规划是如何完成的：①首先，系统读取用户在行编辑器内设定的导航参数，选择调用不同的路径规划算法； ②其次，路径规划算法计算得到 path 矩阵，并根据选定的起点和终点来生成路径节点序列； ③最后，根据得到的路径节点序列，生成可用于系统模拟导航的路径链表。

需要特别说明的两点：(1)在室外导航时不能使用室内的节点，因此需要禁用；同理在室内导航时禁用室外节点。(2)所有策略的路线规划算法其核心都是 floyd 算法，其正确性已于概要设计中证明，文末再附证明。



该类主要用于实现一些功能供导航主界面调用，因此定义了大量的静态函数作类外调用，这里给出该类中完整的函数接口。

函数原型	函数功能说明
static void pathSchedule()	根据用户参数进行路径规划

static bool **isEnterIndoor**(int seq)	判断当前精确导航节点是否有效，有效 true，无效 false
static void **travelMsgConcat**(QString &msg)	生成导航信息实时播报并返回
static QString **timeTransfer**(unsigned time)	将毫秒时间转化为(hh:mm:ss)形式的时间
static void **shortestDistance**(QList<unsigned> &myPath, unsigned from, unsigned to)	【路线规划】最短距离策略的路径规划算法，生成[from, to]的路径节点序列并放入链表中。
static void **pathGenerate**(int path[][MAX_NUM], QList<unsigned> &myPath, unsigned u, unsigned v)	【路线生成】根据路线规划算法得到的 path 矩阵生成[u, v]的路径节点序列
static void **makePath**(QList<unsigned> &myList)	【路线生成】根据 myList 提供的节点序列制定可用于导航的路径链表
static QString **pathOutput**()	【路线输出】输出用户当前的路线
static void **shortestTime**(QList<unsigned> &myPath, unsigned from, unsigned to)	【路线规划】最短时间策略的路径规划算法，生成[from, to]的路径节点序列并放入链表中。
static void **shortestDistanceByTransfer**(QList<unsigned> &myPath, unsigned from, unsigned to)	【路线规划】途径某地的最短距离策略的路径规划算法，生成[from, to]的路径节点序列并放入链表中。
static void **shortestTimeWithBicycle**(QList<unsigned> &myPath, unsigned from, unsigned to)	【路线规划】可选交通工具的最短时间策略的路径规划算法，生成[from, to]的路径节点序列并放入链表中。
static void **searchSurroundingBuildings**(QStandardItemModel *_model, QString name, unsigned ridus)	【查询】查询周边 ridus 米内的建筑物信息，以链表_strList 返回。
static bool **isTransferArrived**()	判断当前时刻是否已经通过中转点，主要用于暂停与恢复。
static bool **isCampusMoving**()	判断当前区段是否是在校区间移动，true 是，false 否。
static bool **isGuideBetweenIndoorAndOutdoor**(unsigned from, unsigned to)	判断[from, to]是否是在室外导航，true 是，false 否。
static void **addListItem**(QStandardItemModel* _model, const QString _str, const unsigned people)	向_model 中加入一个内容为_str 的子项目，其颜色根据 people 改变。
static int **findSuitableBus**(QTime start_time, QDate date , unsigned from, unsigned to, unsigned type, int &waitTime)	在交通时刻表中查找距离当前时间最短的校车/班车，返回该车次在容器中的序号以及等待时间。
static void **getSitePara**(QString str, QString &campus, QString &siteName, unsigned &roomId)	读取偏好设置路径中的路径并将其设置为当前的导航参数
static void **msgLogInsert**(QString msg)	【日志】向数据库表 msg_log 中插入一条操作记录日志
static void **pathLogInsert**(QString msg)	【日志】向数据库表 path_log 中插入一条路径记录日志

在路径规划算法中调用了四个定义在 public 中的函数：

函数原型	函数功能说明
void **specialNodeAvaliable**()	启用室内-室外特殊连接节点
void **specialNodeForbidden**()	禁用室内-室外特殊连接节点
unsigned **getIndexOfGraph**(QString nodeName)	根据节点名查找其对应的地图节点序号并返回
unsigned **getIndexOfGraph**(QString nodeName, QString campus)	根据节点名与校区查找其对应的地图节点序号并返回

5.导航主界面 campusGuide

该模块主要用于实现分析命令、模拟用户移动，都是系统的核心部分，由于主观设计部分较多，下面以代码+注释的形式逐行解释。

(1) 命令分析

①信号关联

```
    // 当编辑栏中文本改变时自动调用命令分析槽，检索出图中包含该关键字的节点
    connect(ui->lineEdit_src, SIGNAL(textChanged(QString)), this, SLOT(commandParse()));
    connect(ui->lineEdit_dst, SIGNAL(textChanged(QString)), this, SLOT(commandParse()));
    connect(ui->lineEdit_trans, SIGNAL(textChanged(QString)), this, SLOT(commandParse()));
```

②命令分析槽的实现

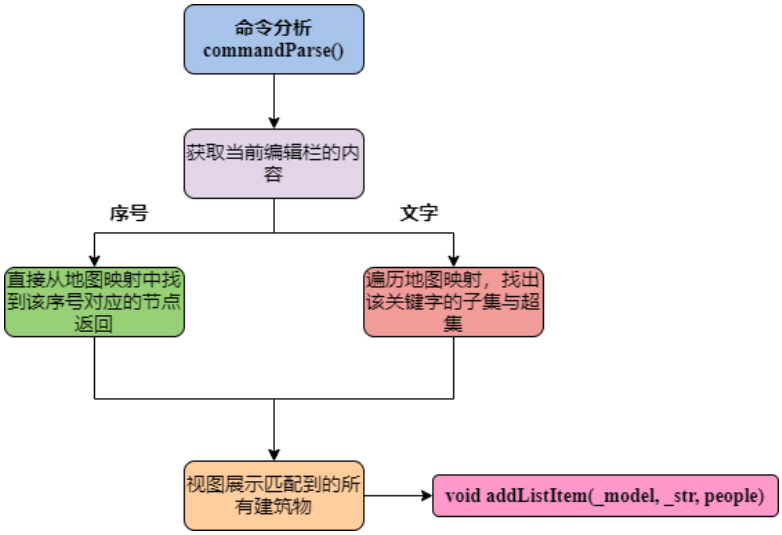
视图展示当前命令关键字的子集与超集。如输入公寓，会得到诸如“学生公寓”、“教师公寓”等结果。

```
// 命令分析，返回所有包含该关键字的建筑物
void campusGuide::commandParse()
{
    QLineEdit *lineEdit = qobject_cast<QLineEdit*>(sender());//得到按下的按钮的指针
    QString command = lineEdit->text();

    bool seqFlag;
    unsigned seq = command.toInt(&seqFlag);
    _matchModel = new QStandardItemModel(this);

    if(seqFlag)   // 当前输入为节点编号则直接在容器中查询该节点后返回
    {
        if(seq<=0 || seq>100)
        {
            lineEdit->clear();
            QMessageBox::information(nullptr, "输入错误", "超出节点编号范围，普通节点(1~100)", QMessageBox::Ok);
            return;
        }
        QString str = QString("[%1] %2").arg(graph_nodes[seq]->campus).arg(graph_nodes[seq]->name);
        addListItem(_matchModel, str, graph_nodes[seq]->people);
    }
    else
    {
        for(auto it=graph_nodes.begin(); it!=graph_nodes.end(); it++)
        {
            if(command.contains((*it)->name, Qt::CaseSensitive)     // 查询关键字command的子集与超集
                    || (*it)->name.contains(command, Qt::CaseSensitive))
            {
                QString str = QString("[%1] %2").arg((*it)->campus).arg((*it)->name);
                addListItem(_matchModel, str, (*it)->people);  // 将当前建筑物加入匹配得到的视图
            }
        }// end of for
    }// end of else

    ui->siteView->setModel(_matchModel);
    ui->textBrowser_msg->hide();
    ui->siteView->show();
```



命令分析演示 1—公寓：

命令分析演示 2—9：

(2) 模拟移动函数的实现

模拟用户移动主要通过定时器实现，每触发一次单位定时器，就根据流逝的时间乘以速度将用户作移动。而每触发一次区段定时器，就代表当前区段已经完成，切换到下一个区段继续移动，直到终点。

其中最主要的难点不在于定时器的设置，而是如何让程序在定时器计时器间等待同时保持主窗口响应，最终采用 Qt 的局部事件循环 QEventLoop 让程序在该时间内局部休眠，成功模拟了用户移动过程。

简单流程如下：

①开始移动前的参数设置：设定时间推移标志为校区内移动、用户状态、出发时间、当前位置、移动用户到起始位置、设定当前显示的地图。

```
timeFlag = TIME_FLAG_CAMPUS;    // 校区内行进，时间加速推进
uuser.userState = "行进中";
uuser.walked = 0;
uuser.campus = graph_nodes[uuser.Path.first()->from]->campus;
uuser.start_time = QDateTime::currentDateTime().toString(tr("yyyy-MM-dd hh:mm:ss"));
ui->userLabel->move(graph_nodes[uuser.Path.first()->from]->COI);
uuser.location = ui->userLabel->pos();
whichMap = uuser.campus;
ui->mapChange->click();
```

②开始模拟移动，启动单位定时器 moveTimer 和区段定时器 userTimer：前者每(1000/timeFlag)触发一次超时信号，更新一次用户当前的位置；后者以区段时间(1000/timeFlag * (*it)->sectionTime + 4*timeFlag)作单次计时，当该定时器超时，即代表该区段已经完成。
（加了 4*timeFlag 的修正时间，因为循环中其他语句有一定执行时间，会导致原定的计时时间被占用一部分，因此需要额外加时间作修正。）

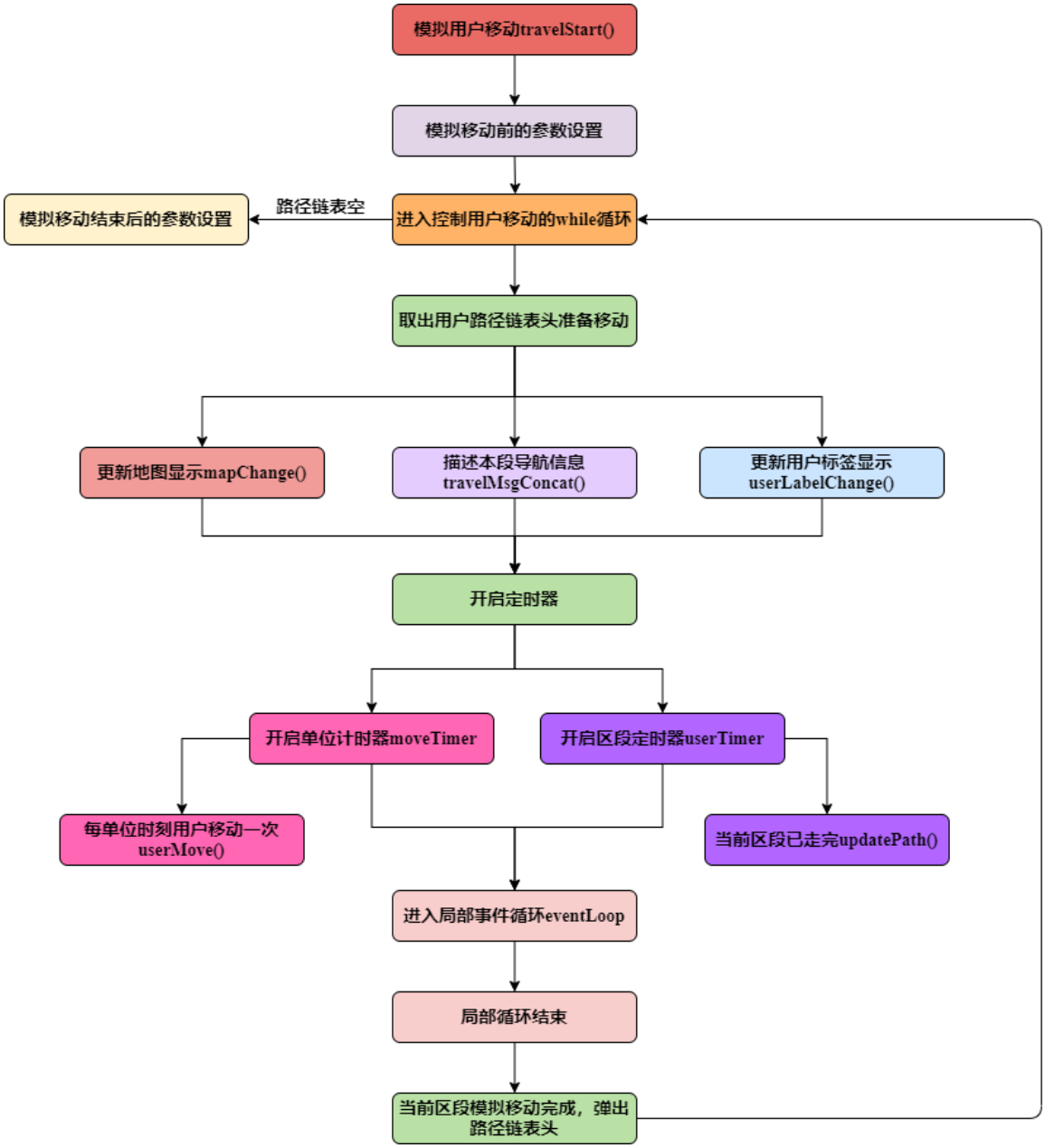
```
moveTimer->start(1000);
connect(userTimer, SIGNAL(timeout()), this, SLOT(updatePath()));
userTimer->start(1000/timeFlag * (*it)->sectionTime + 4*timeFlag); // 系统时间为10倍真实时间推进
// 额外加了4倍timeFlag的修正时间，用于校正usetTimer与sysTimer间的开启计时误差(主要针对校区间移动，校区内几乎可忽略)
```

③在循环体 while 中，先启动上述定时器，然后进入局部事件循环让主线程等待，但仍保持响应，以免循环体导致定时器无法正常运行。

```
// do nothing but wait
QEventLoop eventLoop;    // 使用局部事件循环让主线程等待，但仍保持响应。
QTimer::singleShot(1000/timeFlag * (*it)->sectionTime + 4*timeFlag, &eventLoop, SLOT(quit()));
eventLoop.exec();
```

④区段移动完成时，将该区段的剩余距离清零，同时弹出链表头为下次循环做准备。

程序执行顺序图如下：



该模块完整的接口如下：

函数原型	函数功能说明
void uiSetLoad()	加载界面的初始设置
void objcetAndSignalsInit()	实例初始化、关联控件与信号
void buttonStateChange(int arg)	对各个状态下的按钮状态做限制防止误操作
void commandParse()	【命令分析】命令分析调度
bool userDataRead()	读取用户设定的导航参数，如果输入非法则 false
void userLabelChange()	【模拟移动】根据行进方向及交通方式修改标签图片
void travelStart()	【模拟移动】开始模拟移动
void travelFinished()	【模拟移动】导航结束
void accurateIsOk(unsigned seq, unsigned lineEdit)	判断当前序号对应的建筑物是否是可以精确导航

	的“教学楼”、“公寓”。
void **updateTime**()	【时间推进】更新系统时间
void **updatePath**()	【模拟移动】模拟移动时更新区段路径
void **userMove**()	【模拟移动】模拟移动时移动用户标签
void **updatePeople**()	每 x 分钟取随机数更新一次建筑物内的人数
void **editFinished**()	编辑完成，命令分析结束
void **mapChange**()	【模拟移动】切换显示的地图
void ***closeEvent***(QCloseEvent *event)	当程序异常结束时，用户登出
void **on_logout_clicked**()	【按钮槽】用户登出
void **on_checkBox_talk_stateChanged**(int arg1)	【按钮槽】语音播报功能的单选框
void **on_start_clicked**()	【按钮槽】开始模拟移动
void **on_make_clicked**()	【按钮槽】开始制定路线
void **on_pause_clicked**()	【按钮槽】用户暂停移动
void **backToLogin**()	【信号】回到登录界面
void **on_campusGuide_destroyed**();	【按钮槽】窗口关闭时登出用户
void **on_dstChange_clicked**()	【按钮槽】行进途中改变目的地
void **on_strategyChange_clicked**()	【按钮槽】行进途中改变行进策略
void **on_search_clicked**()	【按钮槽】查询周边建筑物，以视图的形式展示
void **on_searchList_doubleClicked**(const QModelIndex &index)	【按钮槽】双击视图中某一项时
void **on_checkBox_accurate_stateChanged**(int arg1)	【按钮槽】精确导航功能的单选框
void **on_siteView_clicked**(const QModelIndex &index)	【按钮槽】命令分析对应的可选建筑物展示
void **on_preferredView_doubleClicked**(const QModelIndex &index)	【按钮槽】展示用户常用路径
void **signal_travelPause**()	【信号】暂停时发出暂停信号
void **signal_travelContinue**()	【信号】暂停恢复时发出继续信号
void **removeStagnation**()	当本次导航结束下次导航开始前，移除上一次导航过程中因暂停产生的驻点
void **slot_travelPause**();	【信号槽】使用户暂停的一系列操作
void **slot_travelContinue**()	【信号槽】使用户继续移动的一系列操作
void **preferredPathShow**()	【信号槽】每当用户完成一次导航，则更新一次偏好路径显示

三、重要模块函数的伪码算法

1.数据库管理类的构造函数

database::database(QObject *parent=nullptr)

```
1.  {
2.      声明一个数据库实例 db;
3.      验证当前实例是否已经建立默认连接，是则使用默认连接，否则建立新连接;
4.      设置数据库的连接属性：主机 IP、端口号、数据库表名、用户名及密码等;
5.      尝试连接数据库，连接失败则弹出错误提示框;
6.  }
```

2.导航结束后更新常用路径

void database::updateCommonPath(const QString userName)

```
1.  {
2.      声明一个数据库查询实例 query;
3.      填入用户名执行 SQL 查询;（按导航次数降序输出）
4.      查询失败则弹出错误提示框，成功则处理结果集;
5.      遍历结果集前三条记录，分别设置为三条常用路径;
6.      将此三条常用路径再使用 SQL 语句更新到用户表中;
7.  }
```

3.用户登录验证

void on_login_clicked()

```
1.  {
2.      读取编辑栏中的用户名与密码;
3.      if(用户名为空 或 密码为空)    退出并弹出错误对话框;
4.      连接数据库验证用户名与密码是否存在、是否一致;
5.      if(验证不通过)  退出并弹出错误对话框;
6.      登录成功，设置用户结构的用户名与状态，弹出成功对话框后进入导航主界面。
7.      更新配置文件;
8.      向数据库表中插入一条[登录]记录;
```

```
}
```

4.导航策略调度

void Guide::pathSchedule()

```
{
    if(当前导航策略为“最短距离策略”或“最短时间策略”或“最短时间策略（校内可选交通工具）”)
    {
        if(终点房间号有效)  向路径链表中插入一条从终点到终点房间号的当前导航策略对应的路径，同时删除链表尾;
        向链表中插入一条从起点到终点的当前导航策略对应的路径;
        if(起点房间号有效)  先删除链表尾，再向路径链表中插入一条由起点房间号到起点的当前导航策略对应的路径;
    }// end of 最短距离策略 … …

    if(当前导航策略为“最短距离策略（途径某地）”)
    {
        if(终点房间号有效)  向路径链表中插入一条由终点到具体房间号的最短距离路径，同时删除链表尾;
        if(中转点房间号有效)
        {
            向路径链表中插入一条由中转点到终点的最短距离路径，同时删除链表尾;
            向路径链表中插入一条由中转点房间号到中转点的最短距离路劲，同时删除表尾;
            向路径链表中插入一条由中转点到中转点房间号的最短距离路径，同时删除链表尾;
            if(起点房间号有效)
            {
                向路径链表中插入一条由起点到中转点的最短距离路径，同时删除表尾;
                向路径链表中插入一条由起点房间号到起点的最短距离路径;
            }
            else   向路径链表中插入一条由起点到中转点的最短距离路径;
        }
        else if(中转点房间号无效 但 中转点有效)
        {
            向路径链表中插入一条由中转点到终点的最短距离路径，同时删除链表尾;
            if(起点房间号有效)
            {
                向路径链表中插入一条由起点到中转点的最短距离路径，同时删除表尾;
                向路径链表中插入一条由起点房间号到起点的最短距离路径;
```

```
            }
            else  向路径链表中插入一条由起点到中转点的最短距离路径;
        }
        else // 没有中转点
        {
            if(起点房间号有效)
            {
                向路径链表中插入一条由起点到终点的最短距离路径，同时删除表尾;
                向路径链表中插入一条由起点房间号到起点的最短距离路径;
            }
            else  向路径链表中插入一条由起点到终点的最短距离路径;
        }
    }// end of 最短距离策略（途径某地）
    使地图中特殊节点可用;
    将路径链表转化为可用于导航的路径 path;
}
```

5.最短距离策略下的路径规划算法

void shortestDistance(QList<unsigned> &myPath, unsigned from, unsigned to)

```
{
    if(当前[from, to]只在室外导航)  使室内外的特殊连接节点不可用;
    else  使室内外的特殊连接节点可用;
    // part1.为该策略下的路径矩阵 graphTemp 与时间矩阵 time 赋值
    for(i=1; i<=n; i++)
      for(j=1; j<=n; j++)
      {
          graphTemp[i][j] = Graph[i][j];  // 记录从 i 到 j 的距离
          double cg = 1 - (double)Congestion[i][j]/10;  // 将拥挤度矩阵对应到速度缩放比例
          if(Graph[i][j] != INF)    time[i][j] = Graph[i][j] / cg;
          else  time[i][j] = INF;      // 记录从 i 到 j 的时间
      }

    // part2.floyd 算法
    for(k=1; k<=n; k++)      // 要经过的中间点
      for(i=1; i<=n; i++)   // 数组横坐标
        for(j=1; j<=n; j++) // 数组纵坐标
        {
            waitTime = 0;
            // 没有找到合适的公交车则置等待时间为 INF，同时返回序号-1。
```

```
            if(findSuitableBus(QTime::currentTime().addSecs(time[i][k]),
    QDate::currentDate(), i, j, k, waitTime) == -1)   continue;

            // 如果以 k 为中间点检测到的路径更短
            if(graphTemp[i][j] > graphTemp[i][k]+graphTemp[k][j])
            {
                graphTemp[i][j] = graphTemp[i][k] + graphTemp[k][j];
                time[i][j] = time[i][k] + time[k][j] + waitTime; // 更新该路径
    上的时间
                path[i][j] = k;   // 更新要经过的中间点
            }
        }
     pathGenerate(path, myPath, from, to); // 根据得到的 floyd 结果矩阵 path 输
    出节点序列
}
```

6.查找可用的公交车

int Guide::**findSuitableBus**(QTime start_time, QDate date, unsigned from, unsigned to, unsigned type, int &waitTime)

```
{
    提取当前的时间中的星期;
    if(当前节点为校车发车地 4 或 58) // 筛选交通时刻表找到离出发时间最近的一班车
    {
        for(i=0; i<busTable.size(); i++)
        {
            if(日期不对)  continue;  //直接跳过
          if(校车起点和终点与当前行进区间不一致)  continue;
          日期、校区都符合条件时，计算等待时间并与 waitTime 比较，将 waitTime 赋为非
    零较小值;
        }// end of for
        返回最小等待时间对应的车次序号;
     }// end of 校车
    else if(当前节点为班车发车地 53 或 94)  // 班车整点发车，可直接计算等待时间
          计算等待时间，并返回 INF 作为车次序号;
    其他情况置等待时间为负，并返回-1 作为车次序号;
}
```

7.路径生成算法

void Guide::pathGenerate(int path[][MAX_NUM], QList<unsigned> &myPath, unsigned u, unsigned v)

```
1.  {
2.      if(起点的序号 u < 终点序号 v)    swap(u, v)并置翻转标志为真;
3.      将起点 u 压入栈 S 中;
4.      while(path[u][v] != -1)
5.      {
6.          unsigned temp = path[u][v];
7.          if(temp 与 u 两点间不可达)    将 temp 压入栈 S 中，更新 u 为 temp;
8.          else
9.          {
10.             unsigned mid = path[u][temp];
11.             while(path[u][mid] != -1)
12.                 if(u 与 mid 两点间不可达)  更新 u 为 path[u][mid]，将 u 压入栈 S 中;
13.                 else   mid = path[u][mid];
14.         }// end of else
15.         将 mid 压入栈 S 中，更新 u 为 mid;
16.     }// end of while
17.     将终点 v 压入栈 S 中;
18.
19.     if(翻转标志为真)    将栈转化为链表加到路径链表尾;
20.     else       将栈中内容逐个弹出，放在路径链表尾，直至栈空;
21. }
```

8.根据输出的节点序列制定可用于导航的路线

void Guide::makePath(QList<unsigned> &myList)

```
1.  {
2.      // 逆序遍历列表以获得正确的行进节点顺序
3.      for(i=myList.size()-1; i>0; i--)
4.      {
5.          新建一个区段结构分别为其成员变量 from,to,bicycle,sectionCongestion 赋值;
6.          当前区段的行进速度 curSpeed 由交通方式决定;
7.          将该区段的剩余距离 remaingingDistance 设为两节点间距离 Graph[from][to];
8.          区段时间预计行进时间为距离与速度的比值;(向下取整)
9.
10.         将该区段加入用户的路径链表 uuser.Path 中;
```

```
    }// end of for
}
```

9.查询周边建筑物

void Guide::searchSurroundingBuildings(QStandardItemModel *_model, QString name, unsigned ridus)

```
{
    if(当前时刻正在校区间乘车移动)    退出并弹出错误提示框;

    for(i=1; i<graph_nodes.size(); i++)
    {
        if(当前节点的名字包含“路口”或“节点”)    continue;
        if(当前节点的校区与用户当前所在的校区不一致) continue;
        if(当前节点与用户当前位置的距离 <= 查找半径 ridus)
            if(待查找的建筑名为“范围内所有” || 待查建筑名==当前节点名)  将该项加入_model;
    }// end of for
}
```

10.命令分析，返回所有包含该关键字的建筑物

void campusGuide::commandParse()

```
{
    取得触发该分析槽的编辑栏指针与编辑栏内容;
    尝试将内容转化为无符号数;
    if(编辑栏内容为无符号数)  // 直接在容器中查询该节点后返回
    {
        if(该无符号数不在(1,100)区间内)    退出并弹出错误提示框;
        向_matchModel 中插入一项内容为该节点信息的项目;
     }
     else  // 需要寻找该关键字的子集与超集
     {
        for(it=graph_nodes.begin(); it!=graph_nodes.end(); it++)
        {
            if(当前节点不是室外节点) continue;  // 室内节点对室外不可见
```

```
            if(关键字包含节点名 或 节点名包含该关键字)
              向_matchModel 中插入一项内容为该节点信息的项目;
        }
    }
    设置_matchModel 为命令分析栏的标准模型将其内容展示出来;
}
```

11.控制各个状态下的按钮使能

void buttonStateChange(int arg)

```
{
    switch(arg){
      case 0:  // 程序待机状态
          make 按钮有效，start 按钮无效，暂停按钮无效;
          dstChange 按钮无效，strategyChange 按钮无效;
      break;

      case 1:  // 制定导航路线后
          start 按钮有效;
      break;

      case 2:  // 开始移动后
          make 按钮无效，start 按钮无效，暂停按钮有效;
          search 按钮有效，dstChange 按钮有效，strategyChange 按钮有效;
      break;
      }
}
```

12.模拟用户移动

void travelStart()

```
{
    // part1.模拟移动前的准备
    设置时间推进标志为校区内移动，设置用户状态为“移动中”，开始时间为当前时间;
    切换显示的地图为起始点的校区地图，将用户标签移到起点，初始化用户位置;

    // part2.根据制定的路线模拟行进
```

```
7.      while(用户路径链表 uuser.Path 非空)
8.      {
9.          生成当前区段的导航信息;
10.         语音播报标志有效时进行语音播报;
11.         根据前进的方向、交通方式、导航策略改变用户标签的显示图片;
12.         如果当前导航的校区与显示的地图不一致,切换校区;
13.         if(当前区段是在校区间乘车移动)
14.         {
15.             查找等待时间最短的可用校车或班车;
16.             插入一段新的导航信息;
17.             设置时间推进标志为等车状态;
18.             关闭用户单位移动定时器，刷新系统时间定时器;
19.             开启一个略长于等待时间的局部事件循环 1;
20.
21.             // 等待结束，开始乘车移动
22.             加载等待对话框;
23.             设置时间推进标志为校区间移动;
24.             更新当前已经消耗的时间、走过的距离、用户状态等;
25.           }//end of if
26.           else 开启用户单位移动定时器;
27.           开启用户区段定时器;
28.           开启一个略长于区段时间的局部事件循环 2;
29.
30.           // 局部循环 2 结束，用户已走完当前区段路程
31.           重置时间推进标志，隐藏等待对话框;
32.           修正用户标签的位置;
33.           弹出已经导航完的路径链表头;
34.
35.           更新用户参数，如当前走过的距离、消耗的时间、当前位置、校区等;
36.     }// end of while
37.     再次修正用户标签的位置;
38. }
```

13.暂停按钮槽

void on_pause_clicked()

```
1.  {
2.      if(当前时刻正在校区间移动)  退出并弹出错误对话框;
3.      将暂停标志位取反;
4.      if(暂停标志有效)
5.      {
```

```
            设置“暂停”按钮的文本显示为“继续”;
            插入一条用户操作记录[暂停前进];
            设置用户当前的状态为暂停;
            设置该状态下的按钮使能;
            设置当前的时间推进标志为真实时间;

            发出导航暂停信号;
        }
        else
        {
            设置“继续”按钮的文本显示为“暂停”;
            插入一条用户操作记录[继续前进];
            置停顿标志为真;

            发出导航继续信号;
        }
}
```

14.用户移动暂停

void slot_travelPause()

```
{
    清空用户路径;
    关闭用户移动相关定时器;

    新建一个地图节点“驻点”，根据 from，to 设置其参数;
    将该“驻点”插入各矩阵、容器中;

    修改当前位置，校区;
    if(暂停之前的旅程经过了中转点)    清空中转点内容;
}
```

15.用户移动恢复

void slot_travelContinue()

```
{
    重新制定导航路线 make;
```

```
3.      休眠 2s;
4.      以新的路线继续移动 start;
5.  }
```

16.行进途中变更目的地

void on_dstChange_clicked()

```
1.  {
2.      设置新的目的地;
3.      插入一条用户操作记录[行进途中变更目的地];
4.      点击“继续”按钮;
5.  }
```

17.行进途中变更行进策略

void on_strategyChange_clicked()

```
1.  {
2.      设置新的导航策略;
3.      插入一条用户操作记录[行进途中变更导航策略];
4.      点击“继续”按钮;
5.  }
```

18.查询时选择了新的目的地

void on_searchList_doubleClicked(const QModelIndex &index)

```
1.  {
2.      从 index 中分离出新目的地的地名和校区;
3.      点击“切换目的地”按钮;
4.      设置新的目的地，同时修改导航策略为最短时间策略;
5.      点击“继续”按钮，以新目的地为目标继续前进;
6.      插入一条用户操作记录[查询时选择了新目的地];
7.      插入新的路径导航信息;
8.  }
```