# 模块设计

## 1.1模块一（widget）

#### 1.1.1模块概述

该模块由工程创建时创建，不带有菜单项，无状态栏等，不重写paintEvent函数进行绘图，仅进行ui界面设计以及通过main函数创建该对象，调用show函数显示窗口。

#### 1.1.2功能概述

添加“添加乘客”、“发起查询请求”按钮控件，“选择乘客”下拉框和“旅行建议”textBroser显示旅客本次旅程的旅行建议信息，以及将添加的widget提升为封装的mapwidget类，以进行特定的重绘事件，绘制地图，以及乘客的旅行状态。

该模块仅需提供按钮的相应逻辑，以及从子类中获取信息以输出显示图形，以及旅行建议信息。

#### 1.1.3数据结构

#### 1.1.3.1全局数据结构

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 描述 | 功能 |
| num\_traveler | int | 旅客数量 | 记录添加的旅客数量 |
| tra | Struct travelers | struct **travelers**{  QDateTime starttime;  QDateTime totaltime;  QString plan;  int currenttraveler;  }; | Mapwidget类通过开启定时器时刻监控widget中tra内的值，通过starttime,totaltime,plan,  currenttraveler四个值重绘旅客在map上的状态。 |
| traveler\_demand | TravelDemand \* | 旅客对象 | 点击一次添加乘客，创建一个旅客对象，其属性存放在该对象中 |
| tree | ScheduelTree \* | 城市时刻表对象 | 点击城市按钮时，创建该对象，显示从该城市触发的所有车次时刻表 |
| city\_clicked | int | 用户点击的城市 | 获取用户点击的是哪一城市，信号与槽链接，显示该城市信息 |
| map | QMap<int,TravelDemand\*> | 存放旅客的容器 | 点击添加乘客时，创建的对象插入到该容器中，用一个int与其映射 |
| now\_traveler | Int | 当前操作的用户 | 点击下拉框中的其他乘客时，修改这一值为点击的乘客，从而进行该乘客的所有操作 |

#### 1.1.3.2局部数据结构

private:

QString StrFromTra;//从子类TravelDemand中获取的字符串，表示当前用户的旅行信息

//实现时间更新

int timecount;

int co;//小时进位标志

int co1;//天进位标志

int co2;//月进位标志

int id1;//定时器1的Id，用于更新查询系统的时间

int current\_hour2; //获取当前的小时,当监听到子类获取当前时间信号时传值

QDateTime current\_date\_time;

QString date\_hour;

QString current\_hour;

QString current\_y;

QString current\_m;

QString current\_d;

//参与时间进程的变量

QTimer \*mstimer;

QThread \*timethread;

#### 函数说明

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void **paint**() | //TravelDemand界面中点击确认按钮时，或者切换乘客时执行该函数，更新tra中的内容，保证mapwidget中重绘信息正确。 |
| void **CreateConnect**() | 构造函数中创建信号与槽的链接，共添加新乘客、点击下拉框中的某一乘客、查询这三个信号与槽 |
| void **CreateButton**(); | //封装城市按钮创建 |
| void **StartTimer**(); | 构造函数中执行一次，从系统获取时间到查询系统中，并显示改时间，打开id1定时器，持续进行时间的更新 |
| void **createListenSignal**(); | //给每个旅客对象链接信号与槽（添加新乘客时调用） |
| void **initTimeThread**(); | //计时功能及所需的线程 |
| void ***timerEvent***(QTimerEvent \*e); | //定时器事件,刷新时间 |

***timerEvent()*** 中需要调用的子函数

|  |  |
| --- | --- |
| int **updateHour**(int num) | //更新小时 |
| int **updateDay**(int num,int m,int y); | //更新天 |
| int **updateMonth**(int num); | //更新月 |
| int **updateYear**(int num); | //更新年 |

#### 1.1.5算法及算法思想

无算法

#### 1.1.6与其他模块的联系

main函数创建对象并调用show函数，其子类为TravelDemand、ScheduleTree、mapwidget、mypushbutton。

## 1.2模块二（TravelDemand）

#### 1.2.1模块概述

由Qt设计师界面类新建，设计ui界面，获取用户输入的旅行信息，添加算法获得旅行路线输出

#### 1.2.2功能概述

用户输入旅行计划信息的界面，实现初始化控件以及重新打开后的初始化，并输出当前旅客的旅行方案，由父窗口显示方案。

#### 1.2.3模块数据

#### 1.2.3.1乘客信息

int id; //乘客编号

int current\_time; //从widget获得的当前的时间

int depart; //出发地代号

int dest; //目的地代号

int num\_pass; //途径地数量

int Strategy; //采用的策略

int max\_hours; //若采用策略三，规定的最长时间

int startcity; //出发城市

int endcity; //终点城市

int max\_hour; //限定时间的最晚到达时间

int strategy; //选择的策略

int passcity[city\_num]; //途经城市

int passcity\_num; //途经城市数量

int road[road\_size]; //乘客的路径

int mintime[city\_num];//存startcity到各个city最少时间或是限时钱最少的路径

int mincost[city\_num];//存startcity到各个city最少钱

#### 1.2.3.2信号

void **Output**(); //输出信号

void **MoveTraveler**(); //

void **OutputToWidget**(); //字符串输出信号

#### 1.2.3.3算法函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| void **findway**(schedule \*\*\*sch) | //调用以三个算法寻找最佳路径 |
| void **DFS**(schedule \*\*\*sch,int nowcity,int endcost[city\_num],bool iscityvisit[city\_num],int path[city\_num],int endtime[city\_num],int DFScount); | //深度优先算法 |
| void **dijcost**(schedule \*\*\*sch) | //迪杰斯特拉算法 |
| void **dijtime**(schedule \*\*\*sch,int nowtime); | //迪杰斯特拉算法 |

#### 1.2.3.4其他函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| QString **NumToCity**(int city); | //数字转城市 |
| QString **NumToVehical**(int vehical); | //数字转交通方式 |
| QString **NumToStrategy**(int Strategy); | //数字转策略 |
| int **CityToNum**(QString); | //城市转数字 |
| void **copy**(TravelDemand \*another)； | //复制另一个TravelDemand的信息 |
| void **resetsch**(schedule \*\*\*sch); | //初始化时刻表，防止findway后时刻表变化 |
| void **outputway**(schedule \*\*\*sch); | //将旅行建议输出并写入日志中 |

#### 1.2.4算法及算法思想

（见文档结尾 附一）

#### 1.2.5与其他模块的联系

创建该对象时，创建其子类myschedule，令其读取时刻表txt文件中的时刻表信息，在该模块中的算法中使用。

同时，将算法输出的信息传递给父窗口，输出在主窗口中。

## 1.3模块三（mapwidget）

## 1.3.1模块概述

封装的mapwidget类，在widget ui界面中添加一个widget控件并提升为mapwidget类，实现所有的绘图事件。

## 1.3.2 功能概述

实现所有的绘图事件，将城市地图绘制在widget中，并打开计时器，始终检测widget中结构体tra中的各变量值，以此绘制出旅客在地图上的状态，并且可调节绘制的频率，即旅客移动时的帧率，这取决于打开的定时器间隔。

## 1.3.3 模块数据

#### 1.3.3.1

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 描述 | 功能 |
| State | int | 旅客状态 | 根据该变量的值判断旅客当前状态，用不同的图片实现绘制 |

#### 1.3.3.2函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| **MapWidget**(QWidget \*); | //构造函数，打开定时器 |
| void ***paintEvent***(QPaintEvent \*); | //重绘事件函数，绘制地图和旅客状态 |
| QPixmap **setPointGraph**(); | //设置图标函数，返回值为图片相对路径 |
| QPointF **setPointPos**(); | //设置图标位置，返回值为绘制的坐标 |
| QDateTime **getSplitTime**(QDateTime former, QDateTime later); | //获取两时间点时间间隔，返回值为QDateTime，精确到毫秒 |
| double **getTimeDifference**(QDateTime shorterDateTime, QDateTime longerDateTime); | //获取两时间点间隔时间差，返回值为int，精确到毫秒 |
| QPointF **getCityCor**(int city); | //获得城市对应坐标 |
| QPointF **getMoveDistance**(QDateTime spentTime, QDateTime start2Begin, QDateTime start2End,int from, int to); | //获得坐标增量 |
| int **CityToNum**(QString); | //城市转为数字 |
| int **VehicleToNum**(QString); | //交通工具转为数字 |

#### 1.3.3.3 信号

void **update**(); //刷新画面

## 1.3.4 算法及算法思想

//设置当前图标所处位置

QPointF mapwidget::**setPointPos**()

{

Widget \*fatherPtr = (Widget \*)parentWidget();

static QPointF pointPos;

QDateTime spenttime = QDateTime::currentDateTime();

QDateTime starttime = fatherPtr->tra.starttime;

QDateTime totaltime = fatherPtr->tra.totaltime;

QString plan = fatherPtr->tra.plan;

//将字符串Plan剪短为需要的串

int sub = (int)getTimeDifference(starttime,spenttime);

int sum = 0;

for(int i = 0;;i++) {

int timebetween2city = plan.section("-",1,1).toInt()+plan.section("-",3,3).toInt();

if(timebetween2city \* 10000 < sub && timebetween2city != 0)

{

sum += plan.section("-",1,1).toInt()+plan.section("-",3,3).toInt();

int num = 0;

for(int j = 0;j < 4;j++)

num += plan.section("-",j,j).length() + 1;

plan = plan.mid(num);

sub -= timebetween2city \* 10000;

}

else {

break;

}

}

//已用时间不小于总时间，当前位置为目的地

if(spenttime >= totaltime)

{

for(int j = 0;plan.section("-",1,1) != nullptr;j++)

{

int num = plan.section("-",0,0).length() + 1;

plan = plan.mid(num);

}

pointPos = getCityCor(CityToNum(plan));

state = -2;

}

else

while(1)

{

QDateTime departuredatetime = starttime.addSecs(sum \* 10 + plan.section("-",1,1).toInt()\* 10);

QDateTime cityarrivaltime = departuredatetime.addSecs(plan.section("-",3,3).toInt() \* 10);

QDateTime start2Begin = getSplitTime(starttime, departuredatetime);

QDateTime start2End = getSplitTime(starttime, cityarrivaltime);

//已用时间不超过一段路径发车时间，状态为等待

if (getSplitTime(starttime,spenttime) <= start2Begin)

{

pointPos = getCityCor(CityToNum(plan.section("-",0,0)));

state = -1;

break;

}

//已用时间不超过一段路径的到站时间，状态为运动中

else if (getSplitTime(starttime,spenttime) <= start2End)

{

pointPos = getCityCor(CityToNum(plan.section("-",0,0)));

pointPos += getMoveDistance(starttime,spenttime, start2Begin, start2End, CityToNum(plan.section("-",0,0)), CityToNum(plan.section("-",4,4)));

state = VehicleToNum(plan.section("-",2,2));

if (spenttime == start2End)

{

qDebug() << "Arriving at " << plan.section("-",4,4);

}

break;

}

}

return pointPos;//返回的坐标为重绘后旅客的位置

}

## 1.3.5与其他模块的联系

## 1.4 模块四（ScheduleTree）

#### 1.4.1模块概述

创建时创建其子类myschedule，读取时刻表到相应数据结构中，通过读取时刻表信息，将某一城市的时刻表信息打印在TreeWidget中

#### 1.4.2功能概述

实现当点击地图上的城市按钮时，可以弹出该城市的时刻表窗口，显示该城市到与其“相邻的”城市的所有车次及票价等信息

#### 1.4.3 模块数据

//希望从这个函数得到父窗口的值

void **FromA**(int num);

//并把这个值赋给city

int city;

// void test(void);

void **VehicalTime**(void);

void **print**(int c);

1.4.3.1数据

int city; //并把这个值赋给city，显示相应城市的时刻

1.4.3.2函数

|  |  |
| --- | --- |
| 函数名 | 功能 |
| **ScheduelTree**(QWidget \*parent = nullptr); | 初始化窗口等 |
| void **FromA**(int num); | //从这个函数得到父窗口点击的城市的值 |
| void **VehicalTime**(void); | //显示各个车次的时刻表函数 |
| void **print**(int c); | //由上一函数调用，打印在TreeWidget控件上 |

#### 1.4.4算法及算法思想

无算法

#### 1.4.5与其他模块的联系

## 1.5 模块五（myschedule）

#### 1.5.1模块概述

按条读取schedule.txt文件中的时刻表记录，并一条条存储在multimap容器中，最后将multimap容器中的内容写入struct schedule结构体中，作为与算法的接口，整合两人工作内容。

#### 1.5.2 功能概述

作为模块TravelDemand和模块ScheduleTree的子类，读取并储存时刻表，实现父类的功能（如算法、打印时刻表等）

#### 1.5.3 数据

#### 1.5.3.1

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 描述 | 功能 |
| \*\*\*head | struct **schedule** | struct **schedule**{  QString trainnum;  int starttime;  int endtime;  int cost;  int vihecle;  }; | 储存时刻表信息的数据结构 |
| database | static std::multimap<int, Attribute> | 储存时刻表 | 储存时刻表 |

#### 1.5.4 算法及算法思想

无算法

#### 1.5.5与其他模块的联系

## 1.6模块六（mypushbutton）

#### 1.6.1模块概述

封装的按钮，构造函数参数为传入的图片路径，使点击地图上的图片可以实现按钮功能

#### 1.6.2 功能概述

实现点击按钮时，出现图片向下跳动并返回原位的动画效果，以zoom1,zoom2函数实现，点击按钮后显示该城市的时刻表

#### 1.6.3 与其他模块的联系

##### 附一：

**1.2.4 算法及算法思想**

void **dijcost**(schedule \*\*\*sch)

{

const int max=200000;

//初始化

bool isfindbestway[city\_num]={0};

for(int i=0;i<city\_num;i++)

{

mincost[i]=max;

}

int path[city\_num];

int cost[city\_num][city\_num];

for(int i=0;i<city\_num;i++)

{

for(int j=0;j<city\_num;j++)

{

cost[i][j]=max;

}

}

for(int i=0;i<city\_num;i++)

{

for(int j=0;j<city\_num;j++)

{

if(sch[i][j]!=nullptr)

{

for(int k=0;k<6;k++)

{

if(sch[i][j][k].vihecle!=0)

{

if(sch[i][j][k].cost<cost[i][j])

{

cost[i][j]=sch[i][j][k].cost;

}

}

}

}

}

}

for(int i=0;i<city\_num;i++)

{

mincost[i]=cost[startcity][i];

if(mincost[i]<max)

{

path[i]=startcity;

}

else

{

path[i]=-1;

}

}

isfindbestway[startcity]=true;

mincost[startcity]=0;

//初始化结束

//开始寻找最佳路径

for(int i=1;i<city\_num;i++)

{

int min=max;

int v;

for(int j=0;j<city\_num;j++)

{

if(!isfindbestway[j]&&mincost[j]<min)

{

v=j;

min=mincost[j];

}

}

isfindbestway[v]=true;

for(int j=0;j<city\_num;j++)

{

if(!isfindbestway[j]&&(mincost[v]+cost[v][j]<mincost[j]))

{

mincost[j]=mincost[v]+cost[v][j];

path[j]=v;

}

}

}

//复原路线

int i=endcity;

int j=city\_num-1;

int r[city\_num];

while(1)

{

if(i==startcity)

{

r[j]=i;

break;

}

r[j]=i;

j--;

i=path[i];

}

i=0;

for(;j<city\_num;j++)

{

road[i]=r[j];

i++;

}

}

void **dijtime**(schedule \*\*\*sch,int nowtime)

{

const int max=200000;

//初始化

bool isfindbestway[city\_num]={0};

for(int i=0;i<city\_num;i++)

{

mintime[i]=max;

}

int path[city\_num];

for(int i=0;i<city\_num;i++)

{

if(sch[startcity][i]!=nullptr)

{

for(int j=0;j<6;j++)

{

if(sch[startcity][i][j].vihecle!=0)

{

while(sch[startcity][i][j].starttime<nowtime)

{

sch[startcity][i][j].starttime+=24;

sch[startcity][i][j].endtime+=24;

}

if(sch[startcity][i][j].endtime<mintime[i])

{

mintime[i]=sch[startcity][i][j].endtime;

}

path[i]=startcity;

}

}

}

else

{

path[i]=-1;

}

}

//初始化结束

isfindbestway[startcity]=true;

mintime[startcity]=nowtime;

/\*初始化结束，开始循环，每次求startcity到i的最短路径\*/

for(int i=1;i<city\_num;i++)

{

int v;

int min=max;

for(int j=0;j<city\_num;j++)

{

if(!isfindbestway[j]&&mintime[j]<min)

{

v=j;

min=mintime[j];

}

}

isfindbestway[v]=true;

for(int j=0;j<city\_num;j++)

{

if(!isfindbestway[j])

{

if(sch[v][j]!=nullptr)

{

for(int k=0;k<6;k++)

{

if(sch[v][j][k].vihecle==0)

{

break;

}

while(sch[v][j][k].starttime<mintime[v])

{

sch[v][j][k].starttime+=24;

sch[v][j][k].endtime+=24;

}

if(sch[v][j][k].endtime<mintime[j])

{

mintime[j]=sch[v][j][k].endtime;

path[j]=v;

}

}

}

}

}

}

//复原路线

int i=endcity;

int j=city\_num-1;

int r[city\_num];

while(1)

{

if(i==startcity)

{

r[j]=i;

break;

}

r[j]=i;

j--;

i=path[i];

}

i=0;

for(j;j<city\_num;j++)

{

road[i]=r[j];

i++;

}

}

void **DFS**(schedule \*\*\*sch,int nowcity,int endcost[city\_num],bool iscityvisit[city\_num],int path[city\_num],int endtime[city\_num],int DFScount)

{

if(endtime[DFScount]>max\_hour) //若超时则结束这条路径

{

return ;

}

int roadlen=0;

while(road[roadlen+1]!=-1)

{

roadlen++;

}

if(nowcity==endcity) //若到达终点城市

{

bool isallpassed=true;

for(int i=0;i<this->passcity\_num;i++)//检查是否经过所有途经城市

{

if(iscityvisit[this->passcity[i]]==false)

{

isallpassed=false;

break;

}

}

//若经过所有途经城市且方案为目前最优，则保存

if(isallpassed&&(mincost[roadlen]>endcost[DFScount]||roadlen==0))

{

for(int i=0;i<city\_num;i++)

{

if(i<=DFScount)

{

mincost[i]=endcost[i];

mintime[i]=endtime[i];

this->road[i]=path[i];

}

else

{

mincost[i]=0;

mintime[i]=0;

this->road[i]=-1;

}

}

}

//比较是否为目前最优，若最优则更新

return;

}

if(endcost[DFScount]>mincost[roadlen]&&roadlen!=0)

//剪枝，如果到现在这条路费用比最优的高且时间也更多则直接弃用

{

return;

}

DFScount++;

for(int i=0;i<city\_num;i++)

{

if(sch[nowcity][i]!=nullptr&&iscityvisit[i]==false)

{

iscityvisit[i]=true;

path[DFScount]=i;

for(int j=0;j<6;j++)

{

if(sch[nowcity][i][j].vihecle==0)

{

break;

}

endcost[DFScount]=endcost[DFScount-1]+sch[nowcity][i][j].cost;

endtime[DFScount]=sch[nowcity][i][j].endtime;

if(sch[nowcity][i][j].starttime<endtime[DFScount-1])

{

endtime[DFScount]+=24\*(((endtime[DFScount-1]-sch[nowcity][i][j].starttime)/24)+1);

}

//进入下一层递归

DFS(sch,i,endcost,iscityvisit,path,endtime,DFScount);

}

endcost[DFScount]=0;

endtime[DFScount]=0;

path[DFScount]=-1;

iscityvisit[i]=false;

}

}

}

void **findway**(schedule \*\*\*sch)

{

if(this->strategy==0)//最少钱

{

int closestcity;

if(passcity\_num==0) //若没有途经城市则直接使用dijkstra算法

{

this->dijcost(sch);

}

else//有途径城市

{

const int max=200000;

TravelDemand another;

another.copy(this);

bool ispasscityvisit[city\_num]={0};

int visit\_num=0;

while(1)

{

if(visit\_num==this->passcity\_num+1) //若所有途经城市都已访问

{

another.dijcost(sch);

int i=0;

int j=0;

while(another.road[i+1]!=-1)

{

if(this->road[j]!=-1)

{

j++;

continue;

}

else

{

this->road[j]=another.road[i];

i++;

j++;

}

}

this->road[j]=this->endcity;

break;

}

another.dijcost(sch);

int min=max;

for(int i=0;i<another.passcity\_num;i++)//找出距离出发地最近的途经城市

{

if(ispasscityvisit[another.passcity[i]]==true)

{

continue;

}

if(another.mincost[another.passcity[i]]<min)

{

closestcity=another.passcity[i];

min=another.mincost[another.passcity[i]];

}

}

ispasscityvisit[closestcity]=true;

visit\_num++;

another.endcity=closestcity;

for(int i=0;i<road\_size;i++)

{

another.road[i]=-1;

}

another.dijcost(sch);

int i=0;

int j=0;

while(another.road[i+1]!=-1) //将路径记录

{

if(this->road[j]!=-1)

{

j++;

continue;

}

else

{

this->road[j]=another.road[i];

i++;

j++;

}

}

//出发地变为最近的城市

another.startcity=another.endcity;

another.endcity=this->endcity;

}

}

}

else if(strategy==1)//时间最短

{

int nowtime=current\_time;

if(passcity\_num==0) //没有途经城市直接使用dijkstra

{

this->dijtime(sch,nowtime);

}

else//有途经城市

{

int closestcity;

const int max=200000;

TravelDemand another;

another.copy(this);

bool ispasscityvisit[city\_num]={0};

int visit\_num=0;

while(1)

{

if(visit\_num==this->passcity\_num+1) //若所有途经城市都已访问

{

another.dijtime(sch,nowtime);

int i=0;

int j=0;

while(another.road[i+1]!=-1)

{

if(this->road[j]!=-1)

{

j++;

continue;

}

else

{

this->road[j]=another.road[i];

i++;

j++;

}

}

this->road[j]=this->endcity;

break;

}

another.dijtime(sch,nowtime);

int min=max;

for(int i=0;i<another.passcity\_num;i++)//找出距出发地最近的城市

{

if(ispasscityvisit[another.passcity[i]]==true)

{

continue;

}

if(another.mintime[another.passcity[i]]<min)

{

closestcity=another.passcity[i];

min=another.mintime[another.passcity[i]];

}

}

ispasscityvisit[closestcity]=true;

visit\_num++;

another.endcity=closestcity;

for(int i=0;i<road\_size;i++)

{

another.road[i]=-1;

}

another.dijtime(sch,nowtime);

int i=0;

int j=0;

while(another.road[i+1]!=-1) //将路径记录

{

if(this->road[j]!=-1)

{

j++;

continue;

}

else

{

this->road[j]=another.road[i];

i++;

j++;

}

}

another.startcity=another.endcity; //出发地改变为最近城市

another.endcity=this->endcity;

}

}

}

else if(this->strategy==2)//限定时间花钱最少

{

//初始化

int endcost[city\_num],endtime[city\_num],path[city\_num];

bool iscityvisit[city\_num];

for(int i=0;i<city\_num;i++)

{

endcost[i]=0;

endtime[i]=0;

path[i]=-1;

iscityvisit[i]=false;

}

int DFScount=0;

endtime[0]=current\_time;//现在的时间

DFS(sch,startcity,endcost,iscityvisit,path,endtime,DFScount);

if(road[1]>=0&&road[1]<=14)

{

road[0]=startcity;

int roadlen=0;

while(road[roadlen]!=-1)

{

roadlen++;

}

qDebug()<<endl;

for(int i=0;i<roadlen;i++)

{

qDebug()<<mintime[i]<<" ";

}

qDebug()<<endl;

}

else

{

qDebug()<<"no way!"<<endl;

}

}

resetsch(sch); //重置时刻表

}