数据结构说明报告

**数据类**

**旅游日记diary类**

class diary{

public:

int id; //数据库给予的递增的唯一标识

std::string title; //日记标题

std::string context; //日记内容

int site\_id; //日记对应的景点的id

std::string site\_name; //景点的名称

int author; //日记对应的作者的数据库id

std::string author\_name; //日记对应的作者名

int popularity; //日记的热度信息

float score; //日记的评分信息

int score\_number; //日记的打分人数

std::string image\_path; //日记配图的存储路径

};

**景点 locaiton类**

class location

{

public:

int id; //数据库给予的递增的唯一标识

std::string title; //景点名称

int type; //景区/学校标识

int popularity; //景点的热度

float score; //景点的评分

int score\_number; //景点的评分人数

std::string image\_path;

//以下均为景点的标签参数（用于排序）

float food;

float trip;

float sport;

float study;

};

**用户 user类**

class user{

public:

int id; //数据库给予的递增的唯一标识

std::string account; //用户账户

std::string password; //用户密码

};

**地图场所类 place类**

class place {

public:

place(coor loct, const QString &name, const QString &type);

coor getLoct() const { return loct; }

QString getName() const { return name; }

QString getType() const { return type; }

double dist; //可变成员变量，在场所查询时存储离当前位置的距离

private:

coor loct; //坐标

QString name; //场所名称

QString type; //场所类别

};

**地图道路类 road类**

class road {

public:

road(coor start, coor end, road\_type type, double cong);

coor getStart() const { return start; }

coor getEnd() const { return end; }

void setLength(double len) { length = len; }

double getLength() const { return length; }

road\_type getType() const { return type; }

double getCong() const { return cong; }

private:

coor start, end; //始终点坐标

double length; //道路长度

road\_type type; //道路类型（人行道/车道）

double cong; //拥挤度

};

**场所信息类 place\_info类**

作为邻接表和数据传递的基本单位

class place\_info {

public:

place\_info(int id, double weight, road\_type type);

place\_info() = default;

int getId() const { return id; }

double getWeight() const { return weight; }

road\_type getType() const { return type; }

private:

int id; //场所ID

double weight; //权重（距离或时间）

road\_type type; //道路类型（人行道/车道）

};

**工具类**

**路线规划类 route\_plan类**

class route\_plan {

public:

std::vector<place> places; // 存储所有场所信息

std::vector<road> roads; // 存储所有道路信息

std::vector< std::vector<place\_info> > graph\_d; // 存储道路图（距离）

std::vector< std::vector<place\_info> > graph\_t; // 存储道路图（步行时间）

std::vector< std::vector<place\_info> > graph\_m; // 存储道路图（骑行+步行时间）

void load(QString file\_path); // 加载地图数据

void create\_graph(); // 创建道路图

static double shortest\_path(int start, const std::vector<int>& end,

std::vector< std::vector<place\_info> >& graph,

std::vector<place\_info>& record); // 求最短路径

std::vector<place> search\_place(int start, double max\_dist); // 搜索场所

};

**哈夫曼编码类 HuffmanCoding**

// 哈夫曼编码类

class HuffmanCoding

{

private:

// 存储哈夫曼节点的优先队列

std::priority\_queue<HaffumanNode\*, std::vector<HaffumanNode\*>, CompareHnode> q;

HaffumanNode\* root; // 哈夫曼树的根节点

std::map<QChar, QString> codes; // 字符到哈夫曼编码的映射

std::map<QChar, int> data; // 字符到频率的映射

QString codestirng; // 存储编码后的字符串

diary d; // 日记对象

// 私有函数：创建哈夫曼树

void createTree();

// 私有函数：搜索字符的哈夫曼编码

QString searchCode(QChar ch, HaffumanNode\* node, QString s);

// 私有函数：生成字符频率映射

std::map<QChar, int> generateFrequencyMap(const QString& input);

// 私有函数：生成字符编码列表

std::map<QChar, QString> codeList(const std::map<QChar, int>& data);

// 私有函数：处理解压缩

QString getDecompress(QString str);

public:

// 默认构造函数

HuffmanCoding() : root(nullptr) {}

// 公共函数：压缩日记内容

void compress(const diary d);

// 公共函数：从文件中加载日记

diary load\_diary(const QString& input);

// 公共函数：保存日记到文件

int save\_diary(diary d);

};

**哈夫曼树节点类 HuffmanNode**

// 哈夫曼树节点结构

struct HaffumanNode {

int value; // 节点的频率值

int kind; // 节点的类型，1 表示字符节点，0 表示非叶子节点

QChar ch; // 存储的字符

HaffumanNode\* ltree; // 左子树指针

HaffumanNode\* rtree; // 右子树指针

HaffumanNode(QChar character, int val); // 字符节点构造函数

HaffumanNode(int val, HaffumanNode\* left, HaffumanNode\* right)}; // 非叶子节点构造函数

};

**TopK排序 my\_pq类**

template<typename T, typename criteria>

class my\_pq {

private:

std::vector<T> heap;

criteria cmp;

// 自定义上浮操作（用于push）

void sift\_up(size\_t index);

// 自定义下沉操作（用于pop）

void sift\_down(size\_t index);

public:

my\_pq(criteria cmp) : cmp(cmp) {}

// 插入元素到堆中

void push(const T& item);

// 弹出堆顶元素

void pop();

// 获取堆顶元素

const T& top() const;

// 堆判空

bool empty() const;

};

**KMP匹配 KMP类**

class KMP {

private:

// 构建部分匹配表（next数组）

static std::vector<int> buildNext(const std::string& pattern);

public:

// KMP核心匹配算法

static bool kmpMatch(const std::string& text, const std::string& pattern);

};

**数据库读取 read\_data类**

// 数据读取类，用于从数据库中读取各种类型的数据

class read\_data{

public:

read\_data(); // 构造函数，初始化数据库连接

~read\_data(); // 析构函数，关闭数据库连接

static read\_data& getInstance(); // 获取单例实例

std::vector<diary> read\_diary\_data() // 读取日记数据

std::vector<location> read\_location\_data();// 读取地点数据

// 根据 ID 查找日记的地点名称

QString find\_diary\_sitename(int id);

// 根据账号查找用户数据

user read\_user\_data(QString account);

// 根据 ID 查找日记的作者

QString find\_diary\_author(int id);

private:

QSqlDatabase db; // 数据库连接对象

};

**数据库操作类 update\_database**

class update\_database{

public:

update\_database(diary d); // 构造函数，接收一个日记对象

~update\_database(); // 析构函数，关闭数据库连接

void update\_data(); // 更新日记数据

void insert\_data(); // 插入新日记数据

void change\_data(); // 改变数据，如果是新日记则插入，否则更新

void insert\_user\_data(user u); // 插入用户数据

private:

QSqlDatabase db; // 数据库连接对象

diary d; // 日记对象

};

**数据库连接类 database\_connection**

// 数据库连接类，负责管理与数据库的连接

class database\_connection

{

public:

// 获取数据库对象的单例实例

static database\_connection& getInstance();

bool connect(); // 连接到数据库

void disconnect(); // 断开与数据库的连接

QSqlDatabase get\_data\_base(); // 获取 QSqlDatabase 对象

database\_connection(); // 构造函数，初始化数据库参数

~database\_connection(); // 析构函数，确保断开连接

private:

QSqlDatabase db; // 数据库连接对象

QString hostname; // 数据库主机名

QString dbname; // 数据库名称

QString username; // 数据库用户名

QString password; // 数据库密码

};

**页面类**

**主页面**

class MainWindow : public QMainWindow

{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

// 构造函数，接收用户对象和父窗口指针

MainWindow(user u, QWidget \*parent = nullptr);

~MainWindow(); // 析构函数

private slots:

// 槽函数：日记小组件被点击时的响应

void on\_diary\_widget\_clicked();

// 槽函数：校园路线小组件被点击时的响应

void on\_roadwidget\_clicked();

// 槽函数：景点小组件被点击时的响应

void on\_sitewidget\_clicked();

// 槽函数：退出按钮被点击时的响应

void on\_exitbutton\_clicked();

private:

Ui::MainWindow \*ui; // 界面元素指针

user u; // 当前用户对象

diarywindow \*diary\_window; // 日记窗口指针

campus\_nav \*cam; // 校园导航指针

site\_recommend \*site; // 景点推荐指针

};

**登录相关**

**登录窗口 login\_widget类**

// 登录窗口类，负责用户登录界面的功能和交互

class login\_widget : public QWidget{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

// 构造函数，初始化登录窗口

explicit login\_widget(QWidget \*parent = nullptr);

~login\_widget(); // 析构函数

signals:

void loginSucceeded(const user &u); // 登录成功信号，传递用户信息

private slots:

// 槽函数：处理登录按钮点击事件

void on\_loginbutton\_clicked();

// 槽函数：处理注册按钮点击事件

void on\_registerbutton\_clicked();

private:

Ui::login\_widget \*ui; // 界面元素指针

};

**注册窗口 register\_widget类**

// 注册窗口类，负责用户注册的界面和功能

class register\_widget : public QDialog

{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

// 构造函数，初始化注册窗口

explicit register\_widget(QWidget \*parent = nullptr);

~register\_widget(); // 析构函数

signals:

void closewidget(); // 关闭窗口信号

private slots:

void on\_buttonBox\_rejected(); // 槽函数：处理取消按钮点击事件

void on\_buttonBox\_accepted(); // 槽函数：处理确认按钮点击事件

private:

Ui::register\_widget \*ui; // 界面元素指针

user u; // 用户对象

};

**旅游日记相关**

**日记展示页面 diarywindow类**

// 负责日记的显示、管理和用户交互

class diarywindow : public QWidget

{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

std::vector<diary> diarys; // 存储所有日记的 vector

std::vector<diary> diarylist; // 可搜索的日记列表

std::vector<location> locations; // 地点列表

// 构造函数：初始化日记窗口，接收用户对象

explicit diarywindow(user u, QWidget \*parent = nullptr);

// 构造函数：初始化日记窗口，接收用户对象和搜索内容

explicit diarywindow(user u, QString search\_content, QWidget \*parent = nullptr);

~diarywindow(); // 析构函数

// 显示日记列表

void show\_diary(std::vector<diary> diarys);

// 根据标题搜索日记

std::vector<diary> search\_title(const std::string str, std::vector<diary> diarys);

// 根据地点搜索日记

std::vector<diary> search\_site(const std::string str, std::vector<diary> diarys, std::vector<location> locations);

// 处理窗口关闭事件

void closeEvent(QCloseEvent \*event) override;

// 更新分页显示

void updatePagination(bool keepPage = false);

signals:

void windowclose(); // 窗口关闭信号

void open\_local\_file(); // 打开本地文件信号

private slots:

void on\_writediary\_clicked(); // 槽函数：处理写日记按钮点击事件

void open\_write\_widget(const QString l, const int id); // 打开写日记窗口

void choose\_sort\_model(); // 选择排序模型

void on\_sitesearch\_clicked(); // 槽函数：处理地点搜索按钮点击事件

void on\_titlesearch\_clicked(); // 槽函数：处理标题搜索按钮点击事件

void on\_diaryslist\_itemActivated(QListWidgetItem \*item);//处理日记列表项激活事件

void on\_refresh\_clicked(); // 槽函数：处理刷新按钮点击事件

void on\_load\_local\_diary\_clicked(); // 槽函数：处理加载本地日记按钮点击事件

void on\_next\_page\_clicked(); // 槽函数：处理下一页按钮点击事件

void on\_previous\_page\_clicked(); // 槽函数：处理上一页按钮点击事件

private:

Ui::diarywindow \*ui; // 界面元素指针

user u; // 当前用户对象

QButtonGroup \*button\_grooup; // 按钮组

writewidget \*write\_widget; // 写日记窗口指针

location\_choose \*location\_choose\_widget; // 地点选择窗口指针

bool keepPage; // 是否保持当前分页状态

int currentPage = 0; // 当前页码

int itemsPerPage = 10; // 每页显示的日记项数

std::vector<diary> pagedDiarys; // 当前页数据

// 计算总页数

int getTotalPages() const;

};

**日记查看页面 diaryread类**

//负责显示单条日记的内容以及用户交互

class diaryread : public QWidget

{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

// 构造函数：初始化日记阅读窗口，接收日记信息和父窗口指针

explicit diaryread(diary info, QWidget \*parent = nullptr);

// 更新数据库中的日记信息

void diary\_data\_change();

// 更新日记评分

void diary\_score\_change(int number);

// 处理窗口关闭事件

void closeEvent(QCloseEvent \*event) override;

~diaryread(); // 析构函数

signals:

void closewidget(); // 关闭窗口信号

public slots:

// 从本地读取日记

void local\_diary\_read();

private slots:

// 槽函数：处理搜索按钮点击事件

void on\_searchbutton\_clicked();

// 槽函数：处理压缩下载按钮点击事件

void on\_compress\_donwload\_clicked();

private:

Ui::diaryread \*ui; // 界面元素指针

QButtonGroup \*button\_group; // 按钮组，用于评分按钮

diary info; // 存储当前日记信息

};

**日记编写页面 writewidget类**

// 负责提供用户编写日记的界面和功能

class writewidget : public QWidget{

Q\_OBJECT // 使得该类可以使用 Qt 的信号与槽机制

public:

// 构造函数：初始化日记写作窗口，接收用户对象、父窗口指针、地点、地点 ID

explicit writewidget(user u, QWidget \*parent = nullptr, QString site = nullptr, int site\_id = 0);

~writewidget(); // 析构函数

signals:

void closewidget(); // 关闭窗口信号

private slots:

// 槽函数：处理关闭按钮点击事件

void on\_close\_clicked();

// 槽函数：处理完成文章按钮点击事件

void on\_finisharticle\_clicked();

// 槽函数：处理上传图片按钮点击事件

void on\_uploadimage\_clicked();

private:

Ui::writewidget \*ui; // 界面元素指针

QString site; // 当前地点

user u; // 当前用户对象

diary d; // 日记对象

};

**景点推荐类 site\_recommend类**

class site\_recommend : public QWidget

{

Q\_OBJECT

public:

std::vector<location> locations; // 存储从数据库读取的全部景点数据

std::vector<location> locationlists; // 当前显示的景点列表

std::vector<location> topklocations; // 存储排序后的TopK景点结果

std::vector<location> search\_site(const std::string str, std::vector<location> locations);

explicit site\_recommend(user u, QWidget \*parent = nullptr);

~site\_recommend();

void closeEvent(QCloseEvent \*event) override; //发出windowclose()信号

void changeBool();

signals:

void windowclose(); //窗口关闭时发出的信号

void return\_to\_main\_window(); //返回主窗口时发出的信号

private slots:

void sort\_site(); //执行景点排序逻辑

void show\_location(std::vector<location> locations); //locations 要显示的景点数据集合

void on\_search\_site\_button\_clicked(); //搜索按钮点击事件处理

void on\_refresh\_button\_clicked(); //刷新按钮点击事件处理

void on\_return\_button\_clicked(); //返回按钮点击事件处理

void on\_next\_page\_clicked(); //下一页按钮点击事件处理

void on\_previous\_page\_clicked(); //上一页按钮点击事件处理

void updatePagination(bool keepPage = false); // 更新分页数据和UI状态

void on\_pushButton\_clicked();

void on\_locationLists\_itemActivated(QListWidgetItem \*item); //打开对应景点的日记浏览窗口

private:

Ui::site\_recommend \*ui;

user u;

QButtonGroup \*button\_group; //管理排序按钮组的Qt按钮容器

std::vector<location> pagedLocations; // 当前页数据

int currentPage=0; // 当前页码（从0开始）

int itemsPerPage=10; // 每页显示项目数

bool topkuse = true; // 是否启用TopK限制模式

int getTotalPages() const;

};

**校园导航类 campus\_nav类**

class campus\_nav : public QWidget{

Q\_OBJECT

public:

explicit campus\_nav(QWidget \*parent = nullptr);

~campus\_nav();

void show\_nearby(std::vector<place> result); // 显示附近场所

void closeEvent(QCloseEvent \*event) override; // 关闭事件处理

signals:

void windowclose(); // 窗口关闭信号

private slots:

void on\_pushButton\_cur\_clicked(); // 更新当前位置

void on\_pushButton\_search\_clicked(); // 搜索附近场所

void on\_pushButton\_dest\_clicked(QString place\_name); // 添加目的地

void onDeleteButtonClicked(); // 删除目的地按钮点击事件

void on\_pushButton\_plan\_clicked(); // 规划路线

void on\_pushButton\_nearby\_clicked(); // 显示附近场所

void on\_pushButton\_route\_clicked(); // 显示规划路线

void onAddButtonClicked(); // 添加目的地按钮点击事件

private:

Ui::campus\_nav \*ui;

route\_plan rp; // 初始化以调用路线规划类功能

std::unordered\_map<QString, int> place\_name\_to\_id; // 存储场所名称到ID的 映射，降低复杂度

int cur\_loc = 0; // 当前用户位置

std::vector<place> placesToDraw; // 存储需要绘制的场所

std::vector<place\_info> routeToDraw; // 存储需要绘制的路线

DrawMode currentDrawMode = None; // 当前绘制模式

QPointF mapCoordToPixel(coor tar, QRect r); // 将地图坐标转换为像素坐标

protected:

void paintEvent(QPaintEvent \*event) override; // 绘制事件处理

};