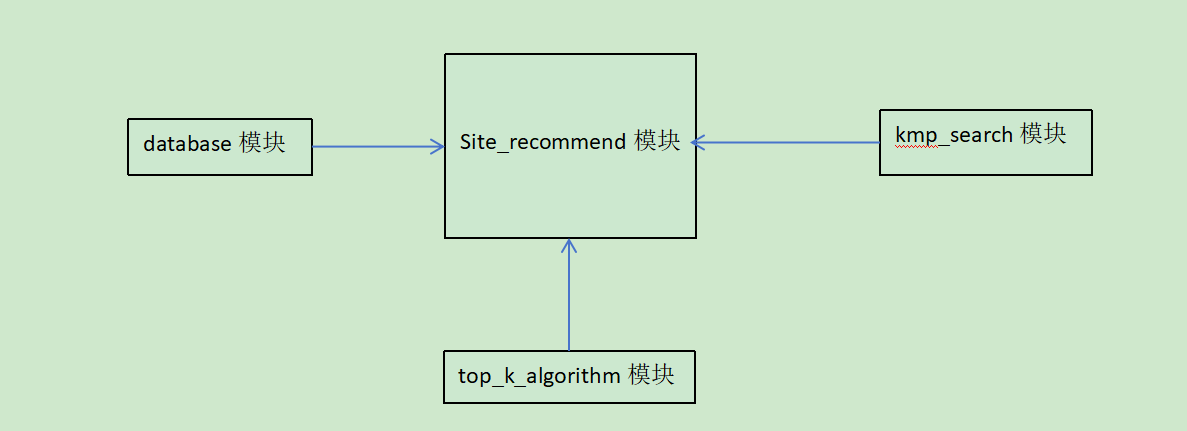
各模块设计报告

一 景点推荐模块

1.1模块总设计



各子模块功能如下：

kmp\_search：KMP匹配算法

top\_k\_algorithm：使用堆排序实现的TopK的功能

site\_recommend：实现根据分类标准进行景点排序，支持查找功能（可进行模糊匹配），可只显示部分结果，也可以显示全部的查找结果

database相关模块：包含了database\_connection, read\_data 和 update\_databse模块，负责数据库的连接和crud功能。

1.2核心子模块KMP

构建next数组

next[i]表示串pattern的最长相同前后缀的长度

// 实现部分匹配表构建

std::vector<int> KMP::buildNext(const std::string& pattern) {

int n = pattern.length();

std::vector<int> next(n, 0); // 初始化next数组，全部设为0

for (int i = 1, j = 0; i < n; ++i) // 当字符不匹配时，回退j到next[j-1]的位置

{

while (j > 0 && pattern[i] != pattern[j]) {

j = next[j - 1];

}

// 如果字符匹配，j向前移动

if (pattern[i] == pattern[j]) {

++j;

}

// 记录当前位置的next值

next[i] = j;

}

return next;

}

匹配过程：

构建模式串的next数组

初始化文本指针i=0和模式指针j=0

逐个比较字符

当字符不匹配时，利用next数组跳过一些比较

当字符匹配时，两个指针都向前移动

如果j=模式串长度，则匹配成功

// KMP匹配算法实现

bool KMP::kmpMatch(const std::string& text, const std::string& pattern) {

std::vector<int> next = buildNext(pattern); // 构建next数组

int n = text.length();

int m = pattern.length();

for (int i = 0, j = 0; i < n; ++i) // 当不匹配时，利用next数组调整j的位置

{

while (j > 0 && text[i] != pattern[j]) {

j = next[j - 1];

}

// 字符匹配时，j向前移动

if (text[i] == pattern[j]) {

++j;

}

// 完全匹配成功

if (j == m) {

return true;

}

}

return false;

}

1.3核心子模块TopKAlgorithm

topK排序实现模板函数：getTopK()

使用堆排序算法思路，自定义模板根堆类实现。传入参数为待排向量，K值和比较器（从而实现不同字段的比较，可通过lambda表达式传入）

template<typename T, typename criteria>

std::vector<T> getTopK(const std::vector<T>& items, int k, criteria cmp) {

my\_pq<T, criteria> minHeap(cmp);

// 初始化堆

for (int i = 0; i < std::min(k, (int)items.size()); ++i) {

minHeap.push(items[i]);

}

// 处理剩余元素

for (size\_t i = k; i < items.size(); ++i) {

if (cmp(items[i], minHeap.top())) {

minHeap.pop();

minHeap.push(items[i]);

}

}

// 提取结果

std::vector<T> result;

while (!minHeap.empty()) {

result.push\_back(minHeap.top());

minHeap.pop();

}

std::reverse(result.begin(), result.end());

return result;

}

// 提取结果

std::vector<T> result;

while (!minHeap.empty()) {

result.push\_back(minHeap.top());

minHeap.pop();

}

std::reverse(result.begin(), result.end());

return result;

}

自定义根堆类my\_pq的实现：

上浮下沉操作

// 自定义上浮操作（用于push）

void sift\_up(size\_t index) {

while (index > 0) {

size\_t parent = (index - 1) / 2;

if (cmp(heap[parent], heap[index])) {

std::swap(heap[parent], heap[index]);

index = parent;

} else {

break;

}

}

}

// 自定义下沉操作（用于pop）

void sift\_down(size\_t index) {

size\_t size = heap.size();

while (true) {

size\_t left = 2 \* index + 1;

size\_t right = 2 \* index + 2;

size\_t largest = index;

if (left < size && cmp(heap[largest], heap[left])) {

largest = left;

}

if (right < size && cmp(heap[largest], heap[right])) {

largest = right;

}

if (largest != index) {

std::swap(heap[index], heap[largest]);

index = largest;

} else {

break;

}

}

}

弹出堆和进入堆

void push(const T& item) {

heap.push\_back(item); // 先将元素放入数组末尾

sift\_up(heap.size() - 1); // 上浮操作

}

void pop() {

if (heap.empty()) return;

std::swap(heap.front(), heap.back()); // 将堆顶与末尾元素交换

heap.pop\_back(); // 移除原堆顶元素

if (!heap.empty()) {

sift\_down(0); // 从根节点开始下沉调整

}

}

1.4核心子模块site\_recommend

site\_recommend模块：加载数据库的信息，根据排序标准进行景点排序，可进行景点查询（可进行模糊匹配）。

1.4.1景点排序函数：景点排序的核心实现，根据用户的排序标准调用TopK函数进行排序，并进行页面更新

void site\_recommend::sort\_site()

{

QApplication::setOverrideCursor(Qt::WaitCursor);

int totalItems = locationlists.size();

QString str = button\_group->checkedButton()->text();

if(topkuse){

qDebug() << "此时仅排序出前十个";

totalItems = 10;

}

if(!str.compare("按学习分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.study > b.study;

});

}

else if(!str.compare("按美食分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.food > b.food;

});

}

else if(!str.compare("按旅游分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.trip > b.trip;

});

}

else if(!str.compare("按运动分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.sport > b.sport;

});

}

else if(!str.compare("按热度分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.popularity > b.popularity;

});

}

else if(!str.compare("按评分分数排序")) {

topklocations = getTopK(locationlists, totalItems, [](const location &a, const location &b) {

return a.score > b.score;

});

}

updatePagination(!topkuse);

QApplication::restoreOverrideCursor();

}

1.4.2点击查找按钮：用户输入完查找的内容后点击查找按钮，调用search\_site函数进行查找，并返回查找结果

void site\_recommend::on\_search\_site\_button\_clicked()

{

QString searchText = ui->search\_line->text();

if(searchText.isEmpty()) {

QMessageBox::information(this, "提示", "请输入搜索关键词");

return;

}

try {

auto results = search\_site(searchText.toStdString(), locations);

if(results.empty()) {

QMessageBox::information(this, "提示", "未找到匹配景点");

}

locationlists = results;

emit button\_group->buttonClicked(button\_group->checkedButton());

} catch(const std::exception &e) {

qCritical() << "搜索出错:" << e.what();

QMessageBox::critical(this, "错误", "搜索过程中发生错误");

}

}

1.4.3查找景点函数：根据用户输入的内容，调用kmp\_search函数，进行字符串匹配，实现查找功能

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

{

std::vector<location> newlocations;

const int max\_results = 100;

qDebug() << "开始搜索，关键词:" << QString::fromStdString(str);

qDebug() << "总数据量:" << locations.size();

for(const location &l : locations) {

if(KMP::kmpMatch(l.title, str)) {

qDebug() << "找到匹配:" << QString::fromStdString(l.title);

newlocations.push\_back(l);

if(newlocations.size() >= max\_results) {

break;

}

}

}

qDebug() << "搜索完成，结果数:" << newlocations.size();

return newlocations;

}

1.4.4展示景点函数：在前端界面显示景点的详细信息

void site\_recommend::show\_location(std::vector<location> locations)

{

ui->locationLists->clear();

for (const location &l : locations) {

std::ostringstream oss;

// 设置左对齐

oss << std::left;

// 设置固定宽度格式

oss << std::setw(40) << l.title // 景点名称

<< std::setw(15) << ("热度: " + QString::number(l.popularity).toStdString())

<< std::setw(15) << ("评分: " + QString::number(l.score, 'f', 1).toStdString())

<< std::setw(15) << ("运动: " + QString::number(l.sport, 'f', 1).toStdString())

<< std::setw(15) << ("美食: " + QString::number(l.food, 'f', 1).toStdString())

<< std::setw(15) << ("旅游: " + QString::number(l.trip, 'f', 1).toStdString())

<< std::setw(15) << ("学习: " + QString::number(l.study, 'f', 1).toStdString());

QListWidgetItem \*item = new QListWidgetItem(ui->locationLists);

item->setText(QString::fromStdString(oss.str()));

item->setData(Qt::UserRole, QVariant::fromValue(l));

}

}

1.4.5双击景点后显示对应的旅游日记

void site\_recommend::on\_locationLists\_itemActivated(QListWidgetItem \*item)

{

std::string title = item->data(Qt::UserRole).value<location>().title;

diarywindow \*dw = new diarywindow(u, QString::fromStdString(title));

dw->show();

}

1.4.6更新页面的核心逻辑，获取当前页码数，获取分页数据，并更新显示

void site\_recommend::updatePagination(bool keepPage)

{

qDebug() << "---- 分页更新开始 ----";

qDebug() << "当前数据总量:" << locationlists.size();

qDebug() << "请求保持页码:" << keepPage;

qDebug() << "当前页码(before):" << currentPage;

// 计算总页数

int totalPages = getTotalPages();

if (totalPages == 0) totalPages = 1; // 至少1页

// 修正当前页码

if (!keepPage) {

currentPage = 0;

} else {

// 使用std::clamp确保页码在有效范围内

currentPage = std::max(0, std::min(currentPage, totalPages - 1));

}

qDebug() << "修正后页码:" << currentPage;

qDebug() << "总页数:" << totalPages;

// 更新按钮状态

ui->previous\_page->setEnabled(currentPage > 0 && !topkuse);

ui->next\_page->setEnabled(currentPage < totalPages - 1 && !topkuse);

// 获取分页数据

int startIdx = currentPage \* itemsPerPage;

int endIdx = std::min(startIdx + itemsPerPage, (int)locationlists.size());

qDebug() << "分页范围:" << startIdx << "-" << endIdx;

qDebug() << "now topklocations size is:" << topklocations.size();

pagedLocations.clear();

if (startIdx < locationlists.size()) {

pagedLocations.assign(topklocations.begin() + startIdx,

topklocations.begin() + endIdx);

}

// 更新显示

show\_location(pagedLocations);

if(!topkuse){

ui->page\_label->setText(

QString("第 %1 页/共 %2 页 (共%3条)")

.arg(currentPage + 1)

.arg(totalPages)

.arg(locationlists.size())

);

} else {

ui->page\_label->clear();

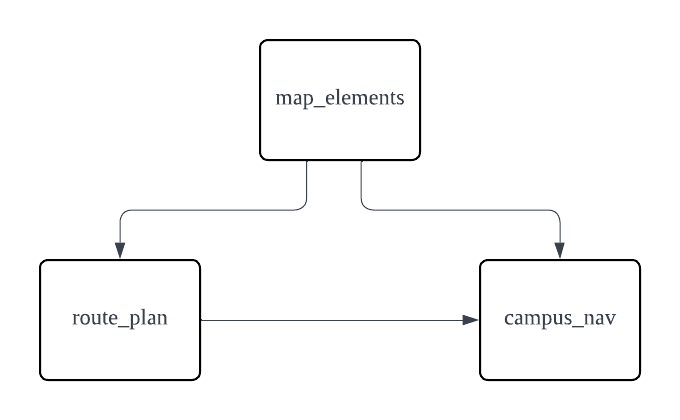
}

qDebug() << "---- 分页更新完成 ----\n";

}

1. 校园导航模块

2.1 模块总设计



各模块功能：

map\_elements: 基本的场所、道路等数据结构定义

route\_plan: 底层算法模块，包含地图数据读取、图建立、路线规划、场所查询

campus\_nav: 负责前端页面展示和用户操作交互

2.2 核心子模块route\_plan

2.2.1 load 函数

本函数加载读取json格式的地图场所和道路数据，写入places和roads向量

void route\_plan::load(QString file\_path) {

QFile file(file\_path);

if (!file.open(QIODevice::ReadOnly)) {

qWarning() << "Failed to open file";

return;

}

QByteArray data = file.readAll();

file.close();

QJsonDocument doc = QJsonDocument::fromJson(data);

if (doc.isNull() || !doc.isObject()) {

qWarning() << "Invalid JSON format";

return;

}

QJsonObject root = doc.object();

QJsonArray features = root["features"].toArray();

for (const QJsonValue &feature : features) {

QJsonObject featureObj = feature.toObject();

QJsonObject geometry = featureObj["geometry"].toObject();

QJsonObject properties = featureObj["properties"].toObject();

QString type = geometry["type"].toString();

if (type == "Point") {

QJsonArray coordinates = geometry["coordinates"].toArray();

coor loct = {coordinates[0].toDouble(), coordinates[1].toDouble()};

QString name = properties["name"].toString();

QString desc = properties["description"].toString();

places.emplace\_back(loct, name, desc);

} else if (type == "LineString") {

QJsonArray coordinates = geometry["coordinates"].toArray();

if (coordinates.size() < 2) continue;

QJsonArray start = coordinates[0].toArray();

QJsonArray end = coordinates[1].toArray();

coor startCoor = {start[0].toDouble(), start[1].toDouble()};

coor endCoor = {end[0].toDouble(), end[1].toDouble()};

road\_type rType = properties["description"].toString() == "自行车道" ? cycleway : sidewalk;

double cong = properties["congestion"].toDouble();

road r(startCoor, endCoor, rType, cong);

r.setLength( calcu\_length(r.getStart(), r.getEnd()) );

roads.push\_back(r);

}

}

}

2.2.2 create\_graph 函数

根据场所和道路数据建立图的邻接表，三张表边的权重不一样，分别对应三种导航模式

void route\_plan::create\_graph() {

unordered\_map<coor, int> coor\_to\_id;

for (int i = 0; i < places.size(); ++i) {

coor\_to\_id[places[i].getLoct()] = i;

}

graph\_d.resize(places.size());

graph\_t.resize(places.size());

graph\_m.resize(places.size());

for (const auto &road : roads) {

int start\_id, end\_id;

auto it1 = coor\_to\_id.find(road.getStart());

if (it1 == coor\_to\_id.end()) continue;

start\_id = it1->second;

auto it2 = coor\_to\_id.find(road.getEnd());

if (it2 == coor\_to\_id.end()) continue;

end\_id = it2->second;

graph\_d[start\_id].emplace\_back(end\_id, road.getLength(), sidewalk);

graph\_d[end\_id].emplace\_back(start\_id, road.getLength(), sidewalk);

graph\_t[start\_id].emplace\_back(end\_id, road.getLength() / (1.3 \* road.getCong()), sidewalk);

graph\_t[end\_id].emplace\_back(start\_id, road.getLength() / (1.3 \* road.getCong()), sidewalk);

road\_type type = road.getType();

if(type == cycleway) {

graph\_m[start\_id].emplace\_back(end\_id, road.getCong() \* road.getLength() / (4.0 \* road.getCong()), cycleway);

graph\_m[end\_id].emplace\_back(start\_id, road.getCong() \* road.getLength() / (4.0 \* road.getCong()), cycleway);

} else if (type == sidewalk) {

graph\_m[start\_id].emplace\_back(end\_id, road.getCong() \* road.getLength() / (1.3 \* road.getCong()), sidewalk);

graph\_m[end\_id].emplace\_back(start\_id, road.getCong() \* road.getLength() / (1.3 \* road.getCong()), sidewalk);

}

}

}

2.2.3 shortest\_path 函数

根据传入的起点编号，目的点编号数组，选中导航策略计算最短路径，返回值为最短度量（距离/时间），具体路线信息记录在record向量中。大框架为贪心算法，循环内部采用Dijkstra算法的思路，每次搜索到一个目的地后将其设为新的起点搜索下一个。

double route\_plan::shortest\_path(int start, const vector<int>& end,

vector<vector<place\_info>>& graph, vector<place\_info>& record){

const double INF = numeric\_limits<double>::max();

int cur = start;

double all\_dist = 0;

vector<bool> visited(end.size(), false);

for(int i = 0; i < end.size(); ++i) {

vector<double> dist(graph.size(), INF);

vector<place\_info> prev(graph.size());

priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;

dist[cur] = 0;

pq.push({0, cur});

int prior;

while (!pq.empty()) {

auto [d, u] = pq.top();

pq.pop();

for(int j = 0; j < end.size(); ++j) {

if (u == end[j] && !visited[j]) {

prior = cur;

cur = end[j];

visited[j] = true;

goto next;

}

}

if (d > dist[u]) continue;

for (const auto &neighbor : graph[u]) {

int v = neighbor.getId();

double weight = neighbor.getWeight();

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

prev[v] = place\_info(u, weight, neighbor.getType());

pq.push({dist[v], v});

}

}

}

next:

int next = cur;

vector<place\_info> temp;

while(next != prior) {

temp.emplace\_back(next, all\_dist + dist[next], prev[next].getType());

next = prev[next].getId();

}

for(auto it = temp.rbegin(); it != temp.rend(); ++it){

record.push\_back(\*it);

}

all\_dist += dist[cur];

}

return all\_dist;

}

2.2.4 search\_place 函数

同样使用Dijkstra算法，搜索返回当前位置start附近范围在max\_dist内的场所数组

vector<place> route\_plan::search\_place(int start, double max\_dist) {

const double INF = numeric\_limits<double>::max();

vector<place> result;

vector<double> dist(graph\_d.size(), INF);

priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

auto [d, u] = pq.top();

pq.pop();

if (d > max\_dist) break; // Stop if the distance exceeds max\_dist

if (d > dist[u]) continue;

else{

places[u].dist = d; // Store the distance in the place object

result.push\_back(places[u]);

}

for (const auto &neighbor : graph\_d[u]) {

int v = neighbor.getId();

double weight = neighbor.getWeight();

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

return result;

}

2.3 核心子模块campus\_nav

2.3.1 on\_pushButton\_cur\_clicked 函数

更新当前位置

void campus\_nav::on\_pushButton\_cur\_clicked()

{

if(place\_name\_to\_id.find(ui->lineEdit\_start->text()) != place\_name\_to\_id.end()){

cur\_loc = place\_name\_to\_id[ui->lineEdit\_start->text()];

currentDrawMode = None;

update();

QMessageBox::information(this, "", "位置更新成功");

} else{

QMessageBox::warning(this, "", "未找到该位置，请检查输入是否正确");

}

}

2.3.2 on\_pushButton\_search\_clicked 函数 + show\_nearby函数

前者调用route\_plan::search\_place函数搜索附近场所，后者展示到表格中

void campus\_nav::on\_pushButton\_search\_clicked()

{

std::vector<place> nearby;

if (cur\_loc == -1) {

qDebug() << "Please select a current location first.";

return;

}

nearby = rp.search\_place(cur\_loc, ui->doubleSpinBox->value());

std::vector<place> result;

for (const place& p : nearby) {

if(p.dist == 0) continue;

if (ui->comboBox\_placeType->currentText() == "选择场所类型" || p.getType() == ui->comboBox\_placeType->currentText()) {

result.push\_back(p);

}

}

show\_nearby(result);

placesToDraw = result;

currentDrawMode = DrawPlaces;

update();

}

void campus\_nav::show\_nearby(std::vector<place> result)

{

// 清空表格内容

ui->stackedWidget->setCurrentIndex(0);

ui->tableWidget\_nearby->clearContents();

if (result.empty()) {

qDebug() << "No nearby places found.";

ui->tableWidget\_nearby->setRowCount(1);

ui->tableWidget\_nearby->setSpan(0, 0, 1, 3);

ui->tableWidget\_nearby->setItem(0, 0, new QTableWidgetItem("没有找到附近的场所。"));

} else {

int row = 0;

ui->tableWidget\_nearby->setRowCount(result.size());

for (const place& p : result) {

ui->tableWidget\_nearby->setItem(row, 0, new QTableWidgetItem(p.getName()));

ui->tableWidget\_nearby->setItem(row, 1, new QTableWidgetItem(QString::number(p.dist)));

QPushButton\* addBtn = new QPushButton("添加");

ui->tableWidget\_nearby->setCellWidget(row, 2, addBtn);

// 绑定按钮点击信号到槽函数

connect(addBtn, &QPushButton::clicked, this, &campus\_nav::onAddButtonClicked);

row++;

}

}

}

2.3.3 on\_pushButton\_dest\_clicked 函数

输入场所名称添加目的地的功能

void campus\_nav::on\_pushButton\_dest\_clicked(QString place\_name)

{

if(place\_name\_to\_id.find(place\_name) == place\_name\_to\_id.end()){

QMessageBox::warning(this, "", "未找到该位置，请检查输入是否正确");

return;

}

int row = ui->tableWidget->rowCount();

ui->tableWidget->insertRow(row);

// 添加名称项

QTableWidgetItem\* item = new QTableWidgetItem(place\_name);

ui->tableWidget->setItem(row, 0, item);

// 添加删除按钮

QPushButton\* delBtn = new QPushButton("删除");

ui->tableWidget->setCellWidget(row, 1, delBtn);

// 绑定按钮点击信号到槽函数

connect(delBtn, &QPushButton::clicked, this, &campus\_nav::onDeleteButtonClicked);

}

2.3.4 on\_pushButton\_plan\_clicked 函数

点击搜索后调用route\_plan::shortest\_path函数，进行路线规划并展示

void campus\_nav::on\_pushButton\_plan\_clicked()

{

std::vector<int> dest\_id;

int rowCount = ui->tableWidget->rowCount();

for(int row = 0; row < rowCount; row++){

QString text = ui->tableWidget->item(row, 0)->text();

if(place\_name\_to\_id.find(text) != place\_name\_to\_id.end()) {

dest\_id.push\_back(place\_name\_to\_id[text]);

} else {

qDebug() << "Place not found: " << text;

}

}

ui->stackedWidget->setCurrentIndex(1);

ui->tableWidget\_route->clearContents();

std::vector<place\_info> record;

QString strategy = ui->comboBox\_strategy->currentText();

double dist;

if(strategy == "距离最短"){

dist = route\_plan::shortest\_path(cur\_loc, dest\_id, rp.graph\_d, record);

ui->lineEdit->setText("总距离:" + QString::number(dist) + "m");

}

else if(strategy == "步行时间最短"){

dist = route\_plan::shortest\_path(cur\_loc, dest\_id, rp.graph\_t, record);

ui->lineEdit->setText("总用时:" + QString::number(dist) + "s");

}

else if(strategy == "混合交通工具时间最短"){

dist = route\_plan::shortest\_path(cur\_loc, dest\_id, rp.graph\_m, record);

ui->lineEdit->setText("总用时:" + QString::number(dist) + "s");

} else {

qDebug() << "Unknown strategy: " << strategy;

return;

}

int row = 0;

ui->tableWidget\_route->setRowCount(record.size());

for (const auto &p : record){

QTableWidgetItem\* nameItem = new QTableWidgetItem(rp.places[p.getId()].getName());

ui->tableWidget\_route->setItem(row, 0, nameItem);

QTableWidgetItem\* weightItem = new QTableWidgetItem(QString::number(p.getWeight()));

ui->tableWidget\_route->setItem(row, 1, weightItem);

QTableWidgetItem\* typeItem = new QTableWidgetItem(p.getType() == cycleway ? "自行车" : "步行");

ui->tableWidget\_route->setItem(row, 2, typeItem);

row++;

}

placesToDraw.clear();

for (const auto &id : dest\_id) {

placesToDraw.push\_back(rp.places[id]);

}

routeToDraw = record;

currentDrawMode = DrawRoute;

update();

}

2.3.5 paintEvent函数 + mapCoordToPixel函数，

重写了widget类的paintEvent函数，每次操作后更新地图中的可视化点位和线路展示

void campus\_nav::paintEvent(QPaintEvent \*event){

QWidget::paintEvent(event); // 保证其他控件正常绘制

QPainter painter(this);

painter.setRenderHint(QPainter::SmoothPixmapTransform, true);

painter.setRenderHint(QPainter::Antialiasing);

QImage mapImage(":/images/data/map.png");

QRect targetRect(29, 151, 400, 586);

painter.drawImage(targetRect, mapImage);

if (currentDrawMode == DrawRoute) {

painter.setPen(QPen(Qt::blue, 3));

for (size\_t i = 0; i < routeToDraw.size(); ++i) {

if(routeToDraw[i].getType() == cycleway){

painter.setPen(QPen(Qt::green, 3)); // 自行车道用蓝色

} else {

painter.setPen(QPen(Qt::blue, 3)); // 人行道用绿色

}

coor tar1 = i == 0 ? rp.places[cur\_loc].getLoct() : rp.places[routeToDraw[i - 1].getId()].getLoct();

coor tar2 = rp.places[routeToDraw[i].getId()].getLoct();

QPointF p1 = mapCoordToPixel(tar1, targetRect);

QPointF p2 = mapCoordToPixel(tar2, targetRect);

painter.drawLine(p1, p2);

}

}

if (currentDrawMode == DrawPlaces || currentDrawMode == DrawRoute) {

painter.setBrush(Qt::darkRed);

painter.setPen(Qt::black);

for (const auto& p : placesToDraw) {

QPointF pos = mapCoordToPixel(p.getLoct(), targetRect);

painter.drawEllipse(pos, 3, 3); // 半径为3的圆点

}

}

QPixmap icon(":/images/data/icon.png"); // 替换成你的图标路径

QPointF pos = mapCoordToPixel(rp.places[cur\_loc].getLoct(), targetRect);

QSize iconSize(24, 24); // 自定义图标的绘制尺寸

QPointF topLeft(pos.x() - iconSize.width() / 2.0, pos.y() - iconSize.height() / 2.0);

QRectF iconRect(topLeft, iconSize);

painter.drawPixmap(iconRect.toRect(), icon); // 绘制

}

QPointF campus\_nav::mapCoordToPixel(coor tar, QRect r)

{

const double north = 39.96499;

const double south = 39.95806;

const double west = 116.35501;

const double east = 116.36123;

double normX = (tar.first - west) / (east - west);

double normY = (tar.second - north) / (south - north);

double pixelX = r.left() + normX \* r.width();

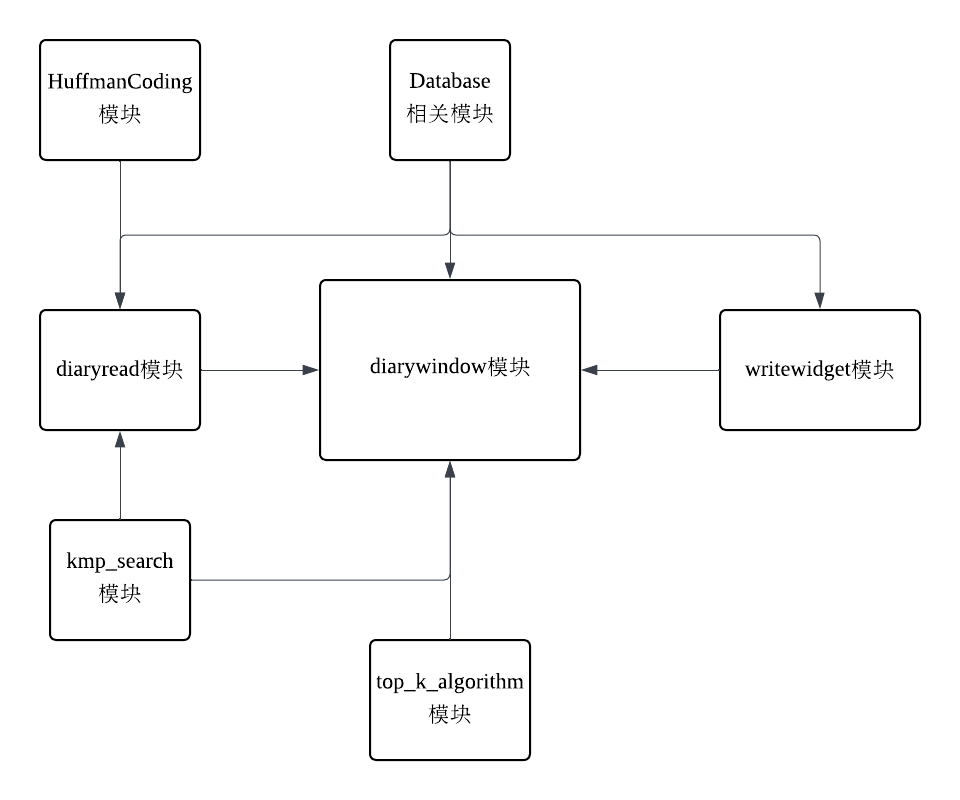
double pixelY = r.top() + normY \* r.height();

return QPointF(pixelX, pixelY);

}

三、旅游日记模块

3.1 模块总设计



各模块功能：

diarywindow模块：负责旅游日记浏览页面的呈现的功能

diaryread模块：负责旅游日记内容页面的呈现和功能

writewidget模块：负责旅游日记写作页面的呈现和功能

HuffmanCoding模块：包含哈夫曼编码/解码功能，用于提供给旅游日记的下载和本地下载文件的展示

kmp\_search模块：kmp算法实现模块，用于实现文章的全文搜索和文章的标题/景点搜索的功能

top\_k\_algorithm模块：堆排序算法的实现，用于实现旅游日记的不同种类的排序功能

database相关模块：包含了database\_connection, read\_data 和 update\_databse模块，负责数据库的连接和crud功能。

3.2 核心子模块：diarywindow

该模块负责旅游日记浏览页面的呈现，实现的功能包括旅游日记的排序，搜索

核心函数如下：

3.2.1 show\_diary 函数

本函数通过传递进来的日记类向量数组，生成对应的视图对象

void diarywindow::show\_diary(std::vector<diary> diarys) //日记列表初始化

{

ui->diaryslist->clear();

if(diarys.size() == 0){

QListWidgetItem \*item = new QListWidgetItem(ui->diaryslist);

item->setText("无结果，请重新搜索");

return;

}

// QVBoxLayout \*layout = new QVBoxLayout(ui->diaryslist);

for(const diary& d : diarys){

std::ostringstream oss;

// 设置输出格式

oss << std::left; // 左对齐

// 控制字段宽度，确保在每列有固定的宽度

oss << std::setw(30) << d.title // 标题

<< std::setw(50) << ("景点: " + d.site\_name) // 景点名称

<< std::setw(20) << ("作者: " + d.author\_name) // 作者名称

<< std::setw(20) << ("热度: " + QString::number(d.popularity).toStdString())

<< std::setw(20) << ("评分: " + QString::number(d.score, 'f', 1).toStdString());

item->setText(QString::fromStdString(oss.str()));

item->setData(Qt::UserRole, QVariant::fromValue(d)); // 保存额外的信息

}

}

3.2.2 choose\_sort\_model 函数

本函数接收不同排序选项点击的信号触发，会读取选项内容进行不同的排序，最后调用updatePagination函数进行页面内容管理和内容呈现

排序算法与景点推荐模块的top\_k\_algorithm相同，故不多赘述

void diarywindow::choose\_sort\_model(){ //排序方法选择

QString str =button\_grooup->checkedButton()->text();

int totalsize = diarylist.size();

if(!str.compare("按热度排序")){

diarylist = getTopK(this->diarylist, totalsize, [](const diary &a, const diary &b) {

return a.popularity > b.popularity; // 按热度排序

});

}

else if(!str.compare("按评分排序"))

{

diarylist = getTopK(this->diarylist, totalsize, [](const diary &a, const diary &b) {

return a.score > b.score; // 按评分排序

});

}

else {

diarylist = getTopK(this->diarylist, totalsize, [](const diary &a, const diary &b) {

return a.score\*1000 + a.popularity > b.score\*1000 + b.popularity;

});

}

updatePagination();

}

3.2.3 on\_sitesearch\_clicked + search\_site 函数

两个函数负责实现根据景点名称对旅游日记的搜索过滤，前者接收搜索信号，调用后者并对排序子模块调用获取新的排序结果。

采用了kmp算法进行名称匹配，故可以做到模糊搜索（部分匹配）

void diarywindow::on\_sitesearch\_clicked()

{

this->diarylist = search\_site(ui->searchbar->text().toStdString(), diarys, locations);

emit button\_grooup->buttonClicked(button\_grooup->checkedButton());

}

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

std::vector<diary> newdiarys;

int id;

for(const location &l : locations)

if(KMP::kmpMatch(l.title,str)){

id = l.id;

qDebug() << "搜到的景点id " << id;

break;

}

for(const diary &d : diarys){

if(d.site\_id == id)

newdiarys.push\_back(d);

}

return newdiarys;

}

3.2.4 on\_titlesearch\_clicked + search\_title 函数

原理与上面的景点搜索相同，故不多赘述

void diarywindow::on\_titlesearch\_clicked()

{

this->diarylist = search\_title(ui->searchbar->text().toStdString(), diarys);

emit button\_grooup->buttonClicked(button\_grooup->checkedButton());

}

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

std::vector<diary> newdiarys;

for(const diary &d : diarys)

if(KMP::kmpMatch(d.title,str)){

newdiarys.push\_back(d);

qDebug() << "搜到的日记名称：" << QString::fromStdString(d.title);

}

return newdiarys;

}

3.3 核心子模块：diaryread

该模块负责旅游日记内容页面的呈现，实现了文章的打分，下载，内容搜索的功能。

3.3.1 on\_searchbutton\_clicked 函数

在点击搜索键后调用该函数对文章中与搜索内容相匹配的字符进行标黄突出显示，无匹配内容时也会有相应显示。

字符串匹配算法与浏览页面的相同。

void diaryread::on\_searchbutton\_clicked()

{

QString str = ui->context\_search->text();

if(!KMP::kmpMatch(info.context, str.toStdString())){

QMessageBox::warning(this, "失败", "没有搜索到对应内容");

ui->context->setText(QString::fromStdString(info.context));

}

else {

QString highlightedText = QString::fromStdString(info.context);

QString replacement = "<span style='background-color: yellow;'>%1</span>";

QRegularExpression regex(QRegularExpression::escape(str));

highlightedText.replace(regex, replacement.arg(str));

ui->context->setHtml(highlightedText); // 更新 QTextBrowser 的内容

}

}

3.3.2 on\_compress\_donwload\_clicked 函数

在下载按钮触发后调用哈夫曼编码模块的函数，实现文章的压缩本地存储（存储的文件可以在浏览页面重新以本地浏览的形式打开）

void diaryread::on\_compress\_donwload\_clicked()

{

HuffmanCoding h;

int k = h.save\_diary(info);

if(k == 0){

QMessageBox::warning(this, "失败", "保存失败");

}

else{

QMessageBox::information(this, "成功", "保存成功");

}

}

3.3.3 void diaryread::diary\_data\_change 函数

在页面关闭时触发，会对用户和的评分和浏览行为做出响应，联动数据库相关模块进行数据修改。

void diaryread::diary\_data\_change()

{

update\_database ud(info);

ud.change\_data();

}

3.4 核心子模块：writewidget

该模块负责旅游日记写作页面的呈现，实现了文章编写，日记图片上传功能的功能。

核心函数如下：

3.4.1 on\_uploadimage\_clicked 函数

响应图片上传按钮的触发，通过提示框指引转存用户选择上传的图片数据，并在结构体中存入对应的存储路径，以供显示时调用

void writewidget::on\_uploadimage\_clicked()

{

// 打开文件对话框，让用户选择文件

QString filePath = QFileDialog::getOpenFileName(this, "选择一张图片", "", "Images (\*.png \*.jpg \*.jpeg \*.bmp)");

if (!filePath.isEmpty()) {

// 定义存储图片的目标文件夹（相对路径）

QString targetDir = "data/uploaded\_images";

QDir dir(targetDir);

// 如果目录不存在则创建它

if (!dir.exists()) {

dir.mkpath("."); // 创建文件夹

}

// 获取文件名称并构建目标路径

QFileInfo fileInfo(filePath);

QString fileName = fileInfo.fileName();

QString targetPath = dir.filePath(fileName);

// 保存文件

if (QFile::copy(filePath, targetPath)) {

QString storedPath = targetPath; // 保存路径

QMessageBox::information(this, "成功", "图片上传成功！\n存储路径: " + storedPath);

qDebug() << "图片已保存到:" << storedPath;

d.image\_path = storedPath.toStdString();

} else {

QMessageBox::warning(this, "失败", "图片上传失败，请重试。");

}

}

}

3.4.2 on\_finisharticle\_clicked 函数

调用于结束编写，联动数据库相关模块，对写入的日记数据进行上传存储

void writewidget::on\_finisharticle\_clicked()

{

d.id = -1;

d.author = u.id;

d.context = ui->textEdit->toPlainText().toStdString();

d.title = ui->title->toPlainText().toStdString();

update\_database ud(d);

ud.change\_data();

writewidget::~writewidget();

}

1. 数据库模块

4.1 核心子模块：database\_connection

该模块用于与远程数据库建立连接，采用单例类，仅具有数据库的连接和断开功能

4.1.1 getInstance 函数

获取数据库单例

database\_connection& database\_connection::getInstance(){

static database\_connection instance;

return instance;

}

4.1.2 连接/断开函数

断开/连接数据库

bool database\_connection::connect(){

if(!db.open()){

qDebug() << "数据库连接失败";

return false;

}

return true;

}

void database\_connection::disconnect(){

if(db.isOpen()){

db.close();

}

}

qDebug() << "数据库连接失败";

return false;

}

return true;

}

void database\_connection::disconnect(){

if(db.isOpen()){

db.close();

}

}

4.2 核心子模块：update\_database

该模块负责对数据库数据的crud

4.2.1 日记数据修改函数

//条目更新

void update\_database::update\_data()

{

qDebug() << "正在修改信息";

QSqlTableModel model;

model.setTable("Diary");

model.setFilter("id = " + QString::number(d.id));

model.select();

QSqlRecord record = model.record(0);

qDebug() << QString::fromStdString(d.title);

record.setValue("score\_number", d.score\_number);

record.setValue("score", d.score);

record.setValue("popularity", d.popularity);

model.setRecord(0, record); // 确保模型更新记录

if (!model.submitAll()) {

qDebug() << "提交失败：" << model.lastError().text();

} else {

qDebug() << "信息更新成功";

}

}

//新条目插入

void update\_database::insert\_data()

{

QSqlTableModel model;

model.setTable("Diary");

model.select();

QSqlRecord record = model.record();

record.setValue("author", d.author);

record.setValue("site\_id", d.site\_id);

record.setValue("context", QString::fromStdString(d.context));

record.setValue("score", 0);

record.setValue("score\_number", 0);

record.setValue("popularity", 0);

record.setValue("image\_path", QString::fromStdString(d.image\_path));

record.setValue("title", QString::fromStdString(d.title));

model.insertRecord(-1, record);

model.submitAll();

}

4.2.2 用户信息新增函数

void update\_database::insert\_user\_data(user u){

QSqlTableModel model;

model.setTable("User");

model.select();

QSqlRecord record = model.record();

record.setValue("account", QString::fromStdString(u.account));

record.setValue("password", QString::fromStdString(u.password));

model.insertRecord(-1, record);

model.submitAll();

}