北京郵電大學

《学生游学系统》项目开发文档

——各模块设计报告

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

小组：第 09 小组

成员：张晨阳 2022211683

廖轩毅 2022211637

徐路 2022211644

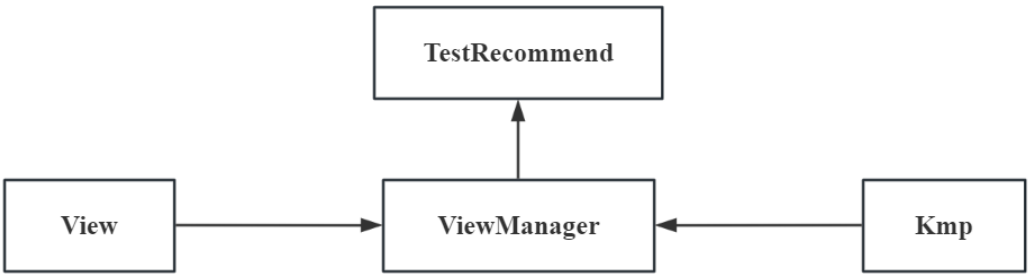
2024 年 5 月 28 号

目录

1.游学推荐模块

1.1. 模块总设计

游学推荐模块设计如图：



各子模块主要功能如下：

TestRecommend：响应前端的各种游学推荐请求；

ViewManager：实现游学推荐的主要功能：排序、搜索等；

View：存储各景区、校园的数据结构；

Kmp：实现 Kmp 字符串匹配算法；

本报告主要介绍核心算法功能的实现，即子模块 ViewManager、Kmp。

1.2. 核心子模块 ViewManager

该模块负责管理景点信息，从数据库获取景点数据信息，提供排序和搜索功能。

具体函数设计实现如下：

1.2.1. getViews 函数

本函数从数据库中获取景点信息并存储在列表中。

处理流程如下：

1) 初始化数据库连接。
2) 连接到数据库。
3) 执行查询以获取所有景点信息。
4) 遍历查询结果，创建 View 对象，并将其添加到列表中。

```
/*获取景点数组*/
void ViewManager::getViews() {
    MYSQL my_sql;
    MYSQL_RES* res;  // 查询结果集
    MYSQL_ROW row;   // 记录结构体

    // 初始化数据库
    mysql_init(&my_sql);

    // 连接数据库
    if (!mysql_real_connect(&my_sql, "localhost", "root", "abc123", "test3",
3306, NULL, 0)) {
        std::cout << "错误原因: " << mysql_error(&my_sql) << "\n";
        std::cout << "连接数据库失败" << "\n";
        exit(-1);
    }

    // 查询 nodes
    mysql_query(&my_sql, "select * from views;");

    // 获取结果集
    int count = 0;
    res = mysql_store_result(&my_sql);
    while (row = mysql_fetch_row(res)) {
        View temp;
        temp.LocationID = std::stoi(row[0]);
        temp.Name = row[1];
        temp.Type = row[2];
        temp.Popularity = std::stoi(row[3]);
        temp.Ratings = std::stod(row[4]);
        views.push_back(temp);
    }
}
```

1.2.2. Recommendation 函数

本函数根据用户的选择推荐景点。

传入参数：

1) 用户依次选择推荐对象（景点、学校或全部）
2) 选择排序方式（热度或评分）
3) 选择推荐数量（前十或全部）
4) 是否进行关键词搜索。

算法流程：

1) 获取用户输入的选择和搜索关键词。
2) 根据用户选择调用 getScore 函数计算综合评分。
3) 调用 q_sort 函数对景点列表进行排序。
4) 根据用户选择的条件过滤和输出景点信息

1.2.3. getScore 函数

本函数计算景点的综合评分。

传入参数：评分和热度的权重系数

处理流程：

1) 遍历所有景点。
2) 根据给定的权重计算每个景点的综合评分。
3) 更新景点对象的评分属性。

算法时间复杂度：$O(n)$，其中 n 是景点数量。

```
1. /*获取综合评分*/
2. void ViewManager::getScore(int a, int b) {
3.     int popularity;
4.     int ratings;
5.     for (int i = 0; i < TEST_MAXSIZE; i++) {
6.         if (views[i].Popularity > MAX_POPULARITY && b != 0)
7.             popularity = 50000;
8.         else
9.             popularity = views[i].Popularity;
10.        ratings = (int)(views[i].Ratings * 1000);
11.        views[i].Score = a * popularity + b * ratings;  // 计算公式
```

```
    }
}
```

1.2.4. q_sort 函数

本函数使用快速排序算法对景点列表进行排序。

传入参数：排序的起始和结束索引。

算法流程：

1) 选择一个中间元素作为基准。
2) 初始化左右指针进行分区操作。
3) 对基准值左边和右边的子数组递归地进行快速排序。

算法时间复杂度：$O(nlogn)$，在最坏情况下的复杂度。

```
/*排序算法*/
void ViewManager::q_sort(int left, int right) {
    int p = views[(left + right) / 2].Score;
    int i = left;
    int j = right;
    while (i <= j) {
        while (views[i].Score > p)
            i++;
        while (views[j].Score < p)
            j--;
        if (i <= j) {
            std::swap(views[i], views[j]);
            i++;
            j--;
        }
    }
    if (j > left)
        q_sort(left, j);
    if (right > i)
        q_sort(i, right);
}
```

1.2.5. selectSort 函数

本算法是一个选择排序的变体，用于对景点列表进行排序，并且只展示前十个符合条件的景点。

算法流程如下：

1) 遍历 views 列表，根据 object 类型和 search_string 搜索条件筛选符合条件的景点。
2) 使用 kmp 算法检查景点名称是否包含搜索字符串。
3) 对筛选出的景点列表 filteredViews 进行排序。
4) 重复以下步骤，直到列表排序完成或达到前十个元素：
5) 寻找当前未排序部分中得分最高的景点。
6) 将该景点与未排序部分的第一个元素交换位置。
7) 返回排序后的前十个景点列表。

算法时间复杂度：$O(n)$，由于这里只进行最多 10 次选择排序操作，因此其时间复杂度可以近似为$O(1)$。

```
1. // 选择排序，实现非全排列展示前十个
2. std::vector<View> ViewManager::selectSort(int length, int object, std::string
search_string) {
3.     std::vector<View> filteredViews;
4.     if (object == 0) {
5.         for (int i = 0; i < length; i++)
6.             if (views[i].Type=="attraction" && kmp(search_string, views[i].Name))
7.                 filteredViews.push_back(views[i]);
8.     } else if (object == 1) {
9.         for (int i = 0; i < length; i++)
10.            if (views[i].Type == "school" && kmp(search_string, views[i].Name))
11.                filteredViews.push_back(views[i]);
12.    } else if (object == 2) {
13.        for (int i = 0; i < length; i++)
14.            if (kmp(search_string, views[i].Name))
15.                filteredViews.push_back(views[i]);
16.    }
17.    int index;
18.    for (int i = 0; i < filteredViews.size() && i < 10; i++) {
19.        index = i;
20.        for (int j = i + 1; j < filteredViews.size(); j++) {
21.            if (filteredViews[j].Score > filteredViews[index].Score)
```

```
22.                 index = j;
23.         }
24.         std::swap(filteredViews[i], filteredViews[index]);
25.     }
26.     return filteredViews;
27. }
```

1.3. 核心子模块 Kmp

该模块实现了 KMP（Knuth-Morris-Pratt）字符串匹配算法，用于在一个字符串中查找另一个字符串的出现位置。

具体函数设计实现如下：

1.3.1. get_nextval 函数

该函数用于计算模式字符串 t 的 next 数组，用于 KMP 算法。

传入参数：待处理字符串，nextval 数组。

算法流程：

1) 初始化变量 j 为 0，k 为 -1；
2) 设置 nextval[0] 为 -1。

```
1. void get_nextval(std::string t, int nextval[]) {
2.     int j = 0, k = -1;
3.     int t_len = t.length();
4.     nextval[0] = -1;
5.     while (j < t_len)
6.         if (k == -1 || t[j] == t[k]) {
7.             j++;
8.             k++;
9.             if (t[j] != t[k])
10.                 nextval[j] = k;
11.             else
12.                 nextval[j] = nextval[k];
13.         } else
14.             k = nextval[k];
15. }
```

1.3.2. kmp 函数

该函数用于在目标字符串 s 中查找模式字符串 t。

传入参数：目标字符串 s，模式字符串 t。

算法流程：

1) 如果 t 等于 "-1"，返回 true；
2) 初始化 nextval 数组，长度为 t 的长度；
3) 调用 get_nextval 函数计算模式字符串 t 的 nextval 数组；
4) 初始化变量 i 和 j 为 0；
5) 进入 while 循环；
6) 如果 j 大于或等于模式字符串长度，返回 true，表示匹配成功；否则返回 false。

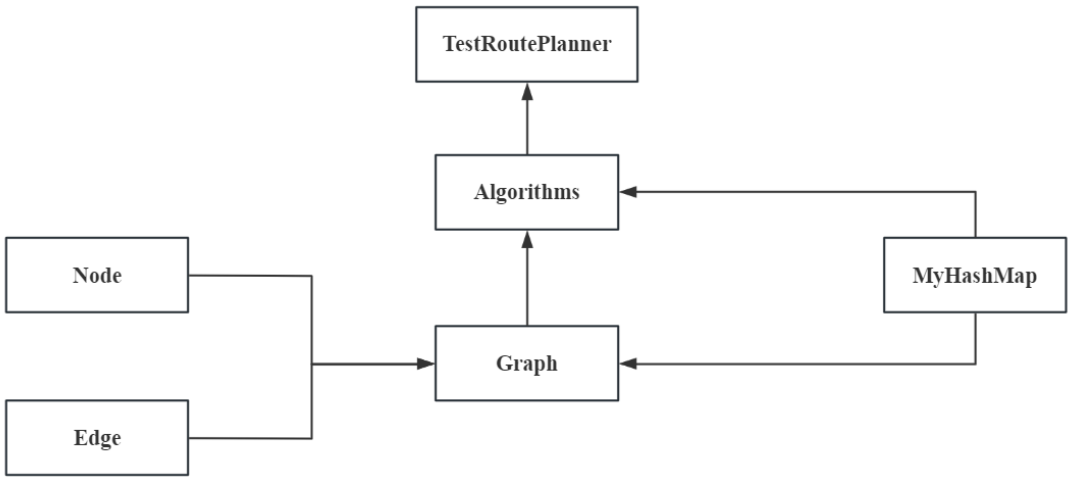
算法时间复杂度：$O(n)$

```
1. bool kmp(std::string t, std::string s) {
2.     if (t == "-1")
3.         return 1;
4.     int line_limit;
5.     line_limit = t.length();
6.     int nextval[line_limit];
7.     int i = 0, j = 0;
8.     int s_len = s.length(), t_len = t.length();
9.     get_nextval(t, nextval);
10.    while (i < s_len && j < t_len)
11.        if (j == -1 || s[i] == t[j]) {
12.            i++;
13.            j++;
14.        } else
15.            j = nextval[j];
16.    if (j >= t_len)
17.        return true;
18.    else
19.        return false;
20. }
```

2.路线规划模块

2.1. 模块总设计

路线规划模块设计如图：



各子模块主要功能如下：

TestRoutePlanner：响应前端的各种路线规划请求；

Algorithms：实现多种路线规划的核心算法；

Graph：存储当前地图的数据结构；

MyHashMap：自己实现的哈希表数据结构；

Node：用于存储不同建筑、设施的数据结构；

Edge：用于存储不同道路的数据结构。

本报告主要介绍核心算法功能的实现，即子模块 Algorithms。

2.2. 核心子模块 Algorithms

本模块定义了一些函数，这些函数提供了图论中的多种路径搜索算法，包括最短路径和最快路径的查找，以及旅行商问题（TSP）的解决方案。

具体函数设计实现如下：

2.2.1. findMinDistanceNode 函数

该函数作为辅助函数，用于在未访问的节点中找到具有最小距离的节点。

传入参数如下：

1) **distance**：存储节点距离的哈希表。
2) **visited**：存储节点访问状态的哈希表。
3) **size**：节点总数。

算法流程如下：

1) 初始化最小距离为无穷大，未找到的节点编号为 -1；
2) 遍历所有节点，检查未访问的节点，并更新最小距离和对应的节点编号；
3) 返回找到的最小距离节点。

算法时间复杂度：$O(n)$，其中 n 是节点总数。

```
1. // 辅助函数，用于找到尚未访问的具有最小距离（时间）的节点
2. int findMinDistanceNode(HashMap<int, double, HashFunc>& distance, HashMap<int,
                           bool, HashFunc>& visited, int size) {
3.     double minDistance = std::numeric_limits<double>::max();
4.     int minNode = -1;
5.     for (int i = 0; i < size; i++) {
6.         double nodeDistance = *(distance.find(i));
7.         if (!*(visited.find(i)) && nodeDistance < minDistance) {
8.             minDistance = nodeDistance;
9.             minNode = i;
10.        }
11.    }
12.    return minNode;
13. }
```

2.2.2.findShortestPath 函数

该函数使用 Dijkstra 算法寻找图中两点间的最短路径。

传入参数如下：

1) **graph**：图对象;
2) **startNodeID**：起始节点 ID;
3) **endNodeID**：目标节点 ID.

算法流程如下：

1) 初始化距离、前驱节点和访问标记的哈希表。
2) 设置起始节点的距离为 0。
3) 循环直到找到目标节点或所有节点被访问。
4) 使用 findMinDistanceNode 找到最小距离节点。
5) 更新邻居节点的距离和前驱节点。
6) 重建路径并返回结果。

算法时间复杂度：$O(n^2)$，在最坏情况下每个节点都可能成为最小距离节点。

```
1. // 寻找最短路径算法
2. Algorithms::PathResult Algorithms::findShortestPath(Graph& graph, int
                                                    startNodeID, int endNodeID) {
3.     int numNodes = graph.size;  // 图中节点总数
4.     // 初始化距离、前驱节点和访问标记的哈希表
5.     HashMap<int, double, HashFunc> distances(numNodes);
6.     HashMap<int, int, HashFunc> predecessors(numNodes);
7.     HashMap<int, bool, HashFunc> visited(numNodes);
8.     PathResult result;
9.
10.    // 初始化
11.    for (int i = 0; i < numNodes; ++i) {
12.        distances.insert(i, std::numeric_limits<double>::max());
13.        predecessors.insert(i, -1);
14.        visited.insert(i, false);
15.    }
16.    distances[startNodeID] = 0;
17.
18.    for (int i = 0; i < numNodes; ++i) {
19.        int u = findMinDistanceNode(distances, visited, numNodes);
20.        if (u == -1)
21.            break;  // 所有节点都已访问
```

```
22.         if (u == endNodeID)
23.             break;  // 找到最短路径
24.
25.         visited[u] = true;
26.
27.         for (const auto& edge : graph.getNode(u)->edges) {
28.             int v = edge->getTo()->id;
29.             double alt = distances[u] + edge->distance;
30.             if (alt < distances[v]) {
31.                 distances[v] = alt;
32.                 predecessors[v] = u;
33.             }
34.         }
35.     }
36.
37.     // 重建从 endNodeId 到 startNodeId 的路径
38.     std::stack<int> pathStack;
39.     for (int at = endNodeID; at != -1; at = predecessors[at])
40.         pathStack.push(at);
41.
42.     if (!pathStack.empty() && pathStack.top() == startNodeID) {  // 如果路径存在
43.         while (!pathStack.empty()) {
44.             result.path.push_back(pathStack.top());
45.             pathStack.pop();
46.         }
47.         result.length = distances[endNodeID];
48.     }
49.
50.     return result;
51. }
```

2.2.3.findFastestPath 函数

本函数用于寻找图中两点间的最快路径。

传入参数如下：

1) **graph**：图对象。
2) **startNodeID**：起始节点 ID。
3) **endNodeID**：目标节点 ID。
4) **mode**：交通方式。

算法流程与 findShortestPath 类似，但考虑交通方式和拥堵对时间的影响。

算法时间复杂度：$O(n^2)$，在最坏情况下每个节点都可能成为最小距离节点。

```
1. // 寻找最快路径算法
2. Algorithms::PathResult Algorithms::findFastestPath(Graph& graph, int
                                    startNodeID, int endNodeID, int mode) {
3.     int numNodes = graph.size;  // 图中节点总数
4.     // 初始化时间、前驱节点和访问标记的哈希表
5.     HashMap<int, double, HashFunc> times(numNodes);
6.     HashMap<int, int, HashFunc> predecessors(numNodes);
7.     HashMap<int, bool, HashFunc> visited(numNodes);
8.     PathResult result;
9.
10.    for (int i = 0; i < numNodes; i++) {
11.        times.insert(i, std::numeric_limits<double>::max());  // 初始化时间为无穷
大
12.        predecessors.insert(i, -1);                          // 初始化前驱节点为-1
13.        visited.insert(i, false);                            // 初始化节点为未访问
14.    }
15.    times[startNodeID] = 0;  // 起点时间设为 0
16.
17.    // 使用 Dijkstra 算法计算最快路径
18.    for (int i = 0; i < numNodes; i++) {     // 找到未访问的最小时间节点
19.        int u = findMinDistanceNode(times, visited, numNodes);
20.        if (u == -1)                              // 无可访问的节点（都已访问）
21.            break;
22.        if (u == endNodeID)  // 如果找到终点，则跳出循环
23.            break;
24.
25.        visited[u] = true;  // 标记该节点为已访问
26.
27.        // 遍历所有出边，更新时间和前驱节点
```

```
28.         int transportmode = static_cast<Edge::type>(mode);
29.         for (const auto& edge : graph.getNode(u)->edges) {
30.             if (edge->transportMode == transportmode) {
31.                 int v = edge->getTo()->id;
32.                 double alt = times[u] + edge->getLength() / (edge->getSpeed() *
                                (1 - edge->getCongestion()));
33.                 if (alt < times[v]) {
34.                     times[v] = alt;
35.                     predecessors[v] = u;
36.                 }
37.             } else {
38.                 // 如果不是当前交通方式的边，则跳过
39.                 continue;
40.             }
41.         }
42.     }
43.
44.     // 从终点开始。使用前驱节点重建路径
45.     std::stack<int> pathStack;
46.     for (int at = endNodeID; at != -1; at = predecessors[at])
47.         pathStack.push(at);  // 将节点压入栈
48.
49.     // 如果路径存在（即栈顶元素为起点），则构建 PathResult 对象
50.     if (!pathStack.empty() && pathStack.top() == startNodeID) {
51.         while (!pathStack.empty()) {
52.             result.path.push_back(pathStack.top());  // 将栈顶元素添加到路径中
53.             pathStack.pop();                         // 弹出栈顶元素
54.         }
55.         result.time = times[endNodeID];  // 设置路径的总时间
56.     }
57.
58.     return result;  // 返回路径结果对象
59. }
```

2.2.4.permutations 函数

本函数用于生成所有可能的排列，用于 TSP 问题的暴力解法。

传入参数如下：

1) **result**：当前最短路径结果。
2) **graph**：图对象。
3) **startNodeID**：起始节点 ID。
4) **arr**：节点 ID 数组。
5) **l**：左侧指针。
6) **r**：右侧指针。

算法流程如下：

1) 如果左右指针相遇，计算路径长度和路径。
2) 如果当前路径长度小于已知最短路径，则更新结果。
3) 对数组中的每个元素，交换当前元素与左侧元素，递归生成排列。

```
1. void permutations(Algorithms::PathResult& result, Graph& graph, int startNodeID,
                     std::vector<int> arr, int l, int r) {
2.     if (l == r) {
3.         // 基础条件：如果左右指针相遇
4.         double tempLength = 0;
5.         std::vector<int> tempPath;
6.         for (int i = 0; i < arr.size(); i++) {
7.             if (i == 0) {
8.                 Algorithms::PathResult temp = Algorithms::findShortestPath(graph,
                                                    startNodeID, arr[i]);
9.                 tempLength += temp.length;
10.                for (int j = 0; j < temp.path.size(); j++) {
11.                    tempPath.push_back(temp.path[j]);
12.                }
13.            } else {
14.                Algorithms::PathResult temp = Algorithms::findShortestPath(graph,
                                                    arr[i - 1], arr[i]);
15.                tempLength += temp.length;
16.                for (int j = 1; j < temp.path.size(); j++) {
17.                    tempPath.push_back(temp.path[j]);
18.                }
19.            }
20.        }
```

```
21.         Algorithms::PathResult temp = Algorithms::findShortestPath(graph,
                                          arr.back(), startNodeID);
22.         tempLength += temp.length;
23.         for (int j = 1; j < temp.path.size(); j++) {
24.             tempPath.push_back(temp.path[j]);
25.         }
26.
27.         if (tempLength < result.length) {
28.             result.length = tempLength;
29.             result.path = tempPath;
30.         }
31.     } else {
32.         // 对于数组中的每个元素，将其与左侧元素交换，然后递归打印右侧子数组的排列
33.         for (int i = l; i <= r; i++) {
34.             // 交换 arr[l] 和 arr[i]
35.             std::swap(arr[l], arr[i]);
36.             // 递归打印右侧子数组的排列
37.             permutations(result, graph, startNodeID, arr, l + 1, r);
38.             // 回溯：交换回来，恢复原样
39.             std::swap(arr[l], arr[i]);
40.         }
41.     }
42. }
```

2.2.5.findBruteForcePath 函数

该函数使用排列生成所有可能路径，寻找 TSP 问题的解。

传入参数如下：

1) **graph**：图对象。
2) **startNodeID**：起始节点 ID。
3) **targets**：目标节点集合。

算法流程如下：

1) 初始化结果，设置初始路径长度为无穷大。
2) 调用 permutations 函数生成所有排列并计算路径长度。
3) 返回最短路径结果。

算法时间复杂度：$O(n!\,n^2)$，因为它们生成所有可能的排列，且每个排列都要执行 Dijkstra 算法。

```
1. // 暴力算法，全排列
2. Algorithms::PathResult Algorithms::findBruteForcePath(Graph& graph, int
                                    startNodeID, std::vector<int>& targets) {
3.     PathResult result;
4.     int size = targets.size();
5.     result.length = INF;
6.     permutations(result, graph, startNodeID, targets, 0, size - 1);
7.
8.     return result;
9. }
```

2.2.6.simulatedAnnealing 函数

该函数基于模拟退火算法，用于寻找图中从起始节点到一系列节点的最优路径。即解决 TSP 问题。

传入参数如下：

1) **completeGraph**：一个完全图对象。
2) **startNodeID**：起始节点的 ID。
3) **nodes**：一个包含目标节点 ID 的向量。

算法流程如下：

1) 初始化当前结果和最佳结果，设置起始温度和冷却率。
2) 在温度高于结束温度时，进行循环，每次循环中：
   (1) 随机扰动当前路径，交换两个节点的位置。
   (2) 计算新路径的长度，并与当前结果进行比较，根据退火概率接受新路径。
   (3) 如果新路径更优，则更新最佳结果。
3) 每次循环后，降低温度。
4) 返回最佳结果。

```
1. // 模拟退火算法相关参数
2. const double START_TEMPERATURE = 50000.0;
3. const double END_TEMPERATURE = 1.0;
4. const double COOLING_RATE = 0.998;
5.
6. // 模拟退火算法
7. Algorithms::PathResult Algorithms::simulatedAnnealing(Graph& completeGraph, int
startNodeID, std::vector<int>& nodes) {
8.     // 随机数生成器
9.     std::random_device rd;
10.    std::mt19937 gen(rd());
11.    std::uniform_real_distribution<> dis(0.0, 1.0);
12.
13.    // 初始化路径和路径长度
14.    Algorithms::PathResult currentResult;
15.    currentResult.path = nodes;
16.    currentResult.path.insert(currentResult.path.begin(), startNodeID);
17.    currentResult.path.push_back(startNodeID);
18.
```

```
    for (int i = 0; i < currentResult.path.size() - 1; ++i)
        for (auto edg : completeGraph.getNode(currentResult.path[i])->edges)
            if (edg->getTo()->id == currentResult.path[i + 1]) {
                currentResult.length += edg->getLength();
                break;
            }

    Algorithms::PathResult bestResult = currentResult;

    double temperature = START_TEMPERATURE;

    // 退火过程
    while (temperature > END_TEMPERATURE) {
        std::vector<int> newPath = currentResult.path;

        // 扰动
        int i = 1 + rand() % (newPath.size() - 2);  // 避免改变起点和终点
        int j = 1 + rand() % (newPath.size() - 2);
        std::swap(newPath[i], newPath[j]);

        Algorithms::PathResult newResult;
        newResult.path = newPath;
        for (int i = 0; i < newResult.path.size() - 1; ++i)
            for (auto edgs : completeGraph.getNode(newResult.path[i])->edges)
                if (edgs->getTo()->id == newResult.path[i + 1]) {
                    newResult.length += edgs->getLength();
                    break;
                }

        double delta = newResult.length - currentResult.length;
        if (delta < 0 || dis(gen) < exp(-delta / temperature))
            currentResult = newResult;

        if (currentResult.length < bestResult.length)
            bestResult = currentResult;

        temperature *= COOLING_RATE;
    }

    return bestResult;
}
```

2.2.7.findOptimalPath 函数

该函数用于构造一个完全图，并在这个完全图上使用模拟退火算法来寻找最优路径。

传入参数：

1) **graph**：原始图对象。
2) **startNodeID**：起始节点的 ID。
3) **targets**：一个包含目标节点 ID 的向量。

算法流程如下：

1) 根据目标节点构造一个完全图，其中包含起点和所有目标点。
2) 在完全图中添加节点和边，边的权重是原始图中相应路径的最短长度。
3) 使用 simulatedAnnealing 函数在这个完全图上寻找最优路径。
4) 根据模拟退火的结果，构造最终的路径。这涉及到将模拟退火过程中的节点顺序转换为原始图中的实际路径。
5) 返回包含最终路径和长度的 PathResult 对象。

算法时间复杂度：近似于$O(n^4)$，由于模拟退火是一个概率性算法，其确切的时间复杂度难以精确计算，但通常会在合理的时间内收敛到一个解。

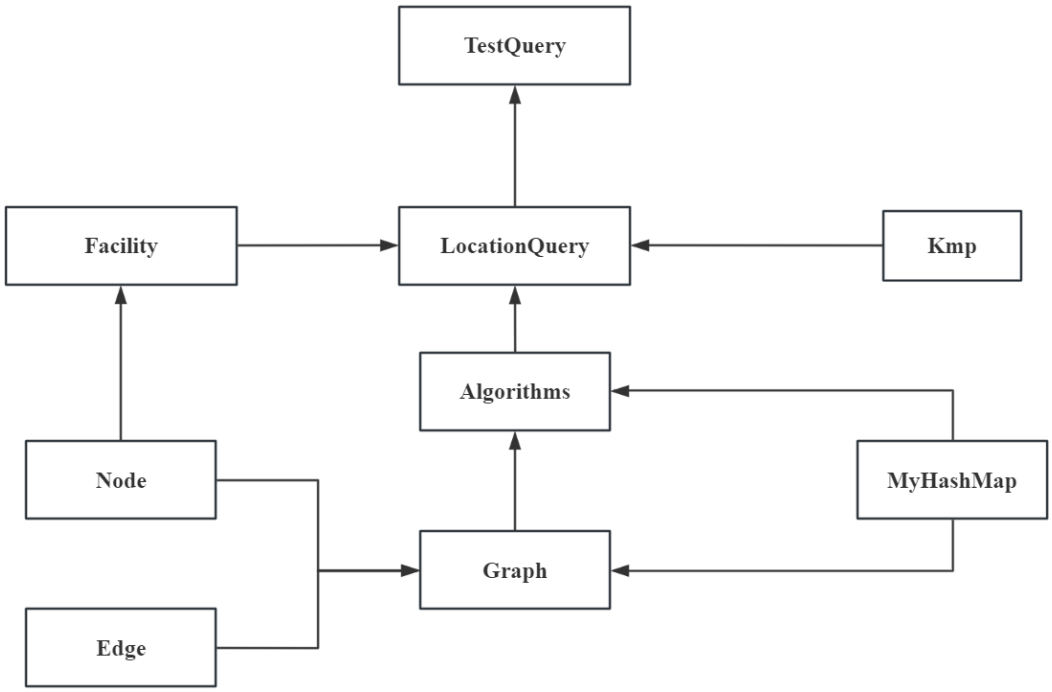
```
1. // 构造完全图并使用模拟退火算法
2. Algorithms::PathResult Algorithms::findOptimalPath(Graph& graph, int
                          startNodeID, std::vector<int>& targets) {
3.     // 构造完全图
4.     int n = targets.size();
5.     Graph completeGraph(n + 1);                          // 包含起点和所有目标点
6.     std::map<std::pair<int, int>, std::vector<int>> paths;  // 存放完全图路径
7.
8.     // 完全图节点添加
9.     Node* t = graph.getNode(startNodeID);
10.    completeGraph.addNode(startNodeID, t->getType(), t->getName(),
t->getDescription());
11.    for (int i = 0; i < n; i++) {
12.        Node* temp = graph.getNode(targets[i]);
13.        completeGraph.addNode(targets[i], temp->getType(), temp->getName(),
                                 temp->getDescription());
14.    }
15.
16.    // 完全图路径添加
```

```
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            Algorithms::PathResult result = Algorithms::findShortestPath(graph,
                                                targets[i], targets[j]);
            completeGraph.addEdge(targets[i], targets[j], result.length, 0, 0,
                                  Edge::type::WALK);
            std::pair<int, int> key1(targets[i], targets[j]);
            std::pair<int, int> key2(targets[j], targets[i]);
            paths.insert(std::make_pair(key1, result.path));
            std::reverse(result.path.begin(), result.path.end());
            paths.insert(std::make_pair(key2, result.path));
        }
        Algorithms::PathResult resultStart = Algorithms::findShortestPath(graph,
                                                startNodeID, targets[i]);
        completeGraph.addEdge(startNodeID, targets[i], resultStart.length, 0, 0,
                              Edge::type::WALK);
        std::pair<int, int> key1(startNodeID, targets[i]);
        std::pair<int, int> key2(targets[i], startNodeID);
        paths.insert(std::make_pair(key1, resultStart.path));
        std::reverse(resultStart.path.begin(), resultStart.path.end());
        paths.insert(std::make_pair(key2, resultStart.path));
    }

    // 模拟退火求解
    Algorithms::PathResult ret = simulatedAnnealing(completeGraph, startNodeID,
                                                    targets);

    // 根据途径点的顺序，添加中间的 node
    std::vector<int> finalpath;
    for (int i = 0; i < ret.path.size() - 1; i++) {
        std::pair<int, int> key(ret.path[i], ret.path[i + 1]);
        auto it = paths.find(key);
        finalpath.insert(finalpath.end(),it->second.begin(),it->second.end()-1);
    }
    finalpath.push_back(ret.path.back());
    ret.path = finalpath;

    return ret;
}
```

3.场所查询模块

3.1. 模块总设计

场所查询模块设计如图：



各子模块主要功能如下：

TestQuery：响应前端的各种场所查询请求；

Facility：用于存储设施的数据结构；

LocationQuery：实现查询设施、计算距离并排序等核心功能；

剩余子模块与前文相同。

本报告主要介绍核心算法功能的实现，即子模块 LocationQuery。

3.2. 核心子模块 LocationQuery

LocationQuery 模块提供了一套完整的基于位置的查询功能，能够高效地处理信息数据。它支持加载设施数据、查找附近设施、按类别过滤结果和按距离排序结果。

具体函数设计实现如下：

3.2.1. loadFacilities 函数

本函数用于从图中加载所有设施节点。

处理流程：遍历图中的所有节点，如果节点类型为 Node::Type::FACILITY，则将其添加到 facilities 向量中。

算法时间复杂度：$O(N)$，其中 N 是图中节点的数量。

```
1. void LocationQuery::loadFacilities() {
2.     for (int i = 0; i < graph.size; i++) {
3.         Node* node = graph.getNode(i);
4.         if (node->getType() == Node::Type::FACILITY)
5.             facilities.push_back(node);
6.     }
7. }
```

3.2.2. findNearbyFacilities 函数

本函数用于查找给定位置和半径内的设施。

传入参数：当前位置（圆心）、指定半径。

算法流程：对于 facilities 向量中的每个设施，计算与给定位置的最短路径长度，如果距离小于或等于指定半径，则将该设施添加到结果向量中。

算法时间复杂度：$O(n^3)$，遍历设施，每个设施执行一次最短路径算法。

```
1. std::vector<Node*> LocationQuery::findNearbyFacilities(Node* location, double
                                                          radius) {
2.     std::vector<Node*> nearbyFacilities;
3.
4.     for (const auto& facility : facilities) {
5.         Algorithms::PathResult pathresult = Algorithms::findShortestPath(graph,
                                               location->id, facility->id);
6.         double distance = pathresult.length;
7.         if (distance <= radius) {
8.             facility->setDistance(distance);
9.             nearbyFacilities.push_back(facility);
10.        }
11.    }
12.
13.    return nearbyFacilities;
14. }
```

3.2.3. filterResultsByCategory 函数

本函数用于按照类别过滤设施。

传入参数：findNearbyFacilities 函数查找到的节点向量、类别。

算法流程：遍历结果向量，如果设施的描述与指定类别匹配，则将其添加到过滤后的结果向量中。

算法时间复杂度：$O(N^2)$.

```
1. std::vector<Node*> LocationQuery::filterResultsByCategory(std::vector<Node*>
                                    results, std::string& category) {
2.     std::vector<Node*> filteredResults;
3.
4.     for (const auto& facility : results)
5.         if (kmp(category, facility->getDescription()))
6.             filteredResults.push_back(facility);
7.
8.     return filteredResults;
9. }
```

3.2.4. sortFacilitiesByDistance 函数

本函数用于按照距离对设施进行排序。

传入参数：filterResultsByCategory 函数过滤出的目标节点向量、两个整数类型的标记（用于快排）。

算法流程：使用快速排序算法根据设施的 getDistance 值进行排序。

算法平均时间复杂度：$O\left(n * log(n)\right)$，即快速排序的平均时间。

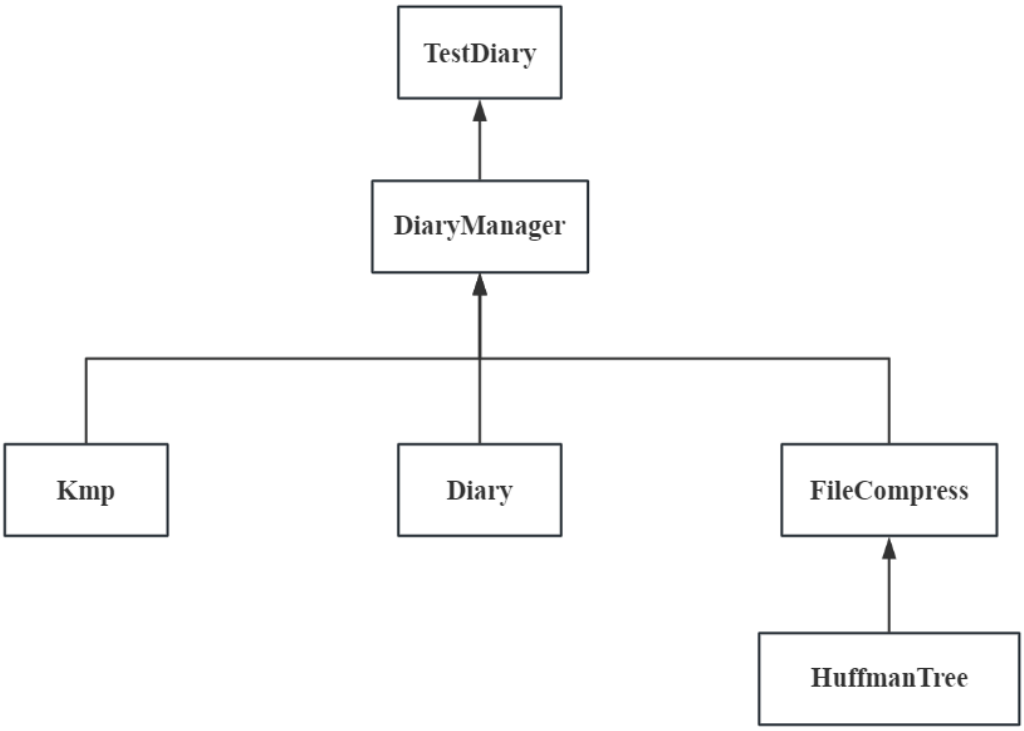
```
1. std::vector<Node*> LocationQuery::sortFacilitiesByDistance(std::vector<Node*>&
facilities, int low, int high) {
2.     if (low < high) {
3.         Node* pivot = facilities[low];  // 选择第一个元素作为基准
4.         int l = low;
5.         int r = high;
6.
7.         while (l < r) {
8.             while (l < r && facilities[r]->getDistance() >= pivot->getDistance())
9.                 r--;
10.            facilities[l] = facilities[r];
11.            while (l < r && facilities[l]->getDistance() <= pivot->getDistance())
```

```
12.                 l++;
13.             facilities[r] = facilities[l];
14.         }
15.         facilities[l] = pivot;
16.
17.         // 分别对分区前后的子数组进行排序
18.         sortFacilitiesByDistance(facilities, low, l - 1);
19.         sortFacilitiesByDistance(facilities, r + 1, high);
20.     }
21.     return facilities;
22. }
```

4.游学日记模块

4.1. 模块总设计

游学日记模块设计如图：



各子模块主要功能如下：

TestDiary：响应前端的各种游学日记请求；

DiaryManager：实现游学日记模块的主要功能、算法；

Diary：用于存储日记的数据结构；

FileCompress：实现日记压缩、解压的核心算法；

HuffmanTree：实现压缩功能所需要的哈夫曼树；

剩余子模块与前文相同。

本报告主要介绍核心算法功能的实现，即子模块 DiaryManager、FileCompress.

4.2. 核心子模块 DiaryManager

该模块实现了日记管理的各项功能，包括日记的添加、打印、评分、搜索、排序、下载、解压和热度更新。

由于部分函数的实现，如快速排序、getScore 与上述模块中实现相同，故不在此赘述。

其余核心函数设计实现如下：

4.2.1. diarySearch 函数

本函数根据标题、作者、目的地和内容搜索日记，并按指定模式排序。

传入参数：搜索的标题、作者、目的地和内容，以及排序方式。

算法流程：

1) 根据 search_mode 计算评分并排序日记。
2) 清空缓存文件。
3) 遍历日记集合，使用 KMP 算法匹配搜索条件。
4) 匹配成功的日记将被打印和写入缓存文件。

算法时间复杂度：$O(n*(m+k))$，其中 m 是搜索字符串的长度，k 是日记内容的长度。

```
1. void DiaryManager::diarySearch(std::string search_title, std::string
search_author, std::string search_destination, std::string search_content, int
search_mode) {
2.     if (search_title == "-1")
3.         search_title = "";
4.     if (search_author == "-1")
5.         search_author = "";
6.     if (search_destination == "-1")
7.         search_destination = "";
8.     if (search_content == "-1")
9.         search_content = "";
10.    int a;
11.    int b;
12.    if (search_mode == 0)  // 按 popularity
13.    {
14.        a = 1;
```

```
        b = 0;
    } else if (search_mode == 1)  // 按 rating
    {
        a = 0;
        b = 1;
    }
    getScore(a, b);
    q_sort(0, diaries.size() - 1);

    // 清空下载缓存文件
    std::string filepath = "D:\\Diarytemp.txt";
    std::ofstream file(filepath.c_str());
    if (!file.is_open()) {
        std::cerr << "无法打开缓存文件！" << std::endl;
    }
    file.close();

    // 匹配操作
    for (Diary diary : diaries) {
        if (kmp(search_title, diary.title) && kmp(search_author, diary.author)
&& kmp(search_destination, diary.destination) && kmp(search_content,
diary.content)) {
            diary.DiaryPrint();
            diary.DiaryWriteintoFile();
        }
    }
}
```

4.2.2. up_popularity 函数

本函数用于根据日记内容更新日记的热度。

算法流程：遍历日记集合，使用 KMP 算法匹配内容。匹配成功的日记热度增加。

算法时间复杂度：$O(n*k)$，其中 n 是日记的数量，k 是内容字符串的长度。

```
// 热度自增
int DiaryManager::up_popularity(std::string content) {
    for (Diary diary : diaries) {
        if (kmp(content, diary.content)) {
            diary.popularity += 10;
            return diary.popularity;
        }
    }
    return -1;
}
```

4.2.3. update_rate 函数

本函数用于更新指定日记的评分。

算法流程：遍历日记集合，使用 KMP 算法匹配内容。匹配成功的日记评分更新。

算法时间复杂度：$O(n*k)$，其中 n 是日记的数量，k 是内容字符串的长度。

```
// 日记评分
int DiaryManager::update_rate(std::string content, int new_rating) {
    for (Diary diary : diaries) {
        if (kmp(content, diary.content)) {
            diary.rating = (diary.rating + new_rating) / 2;
            return diary.rating;
        }
    }
    return -1;
}
```

4.3. 核心子模块 FileCompress

本模块提供文件压缩和解压缩的功能。它使用哈夫曼编码算法对文件进行压缩，将文件内容压缩成一个 .zlx 文件，并提供解压缩功能，将 .zlx 文件恢复成原始文件。

由于本模块涉及保存文件名、文件后缀等与算法、数据结构关系不大的函数，故本文只介绍实现压缩、解压的核心算法函数。

核心函数设计如下：

4.3.1. Compress 函数

该函数用于压缩指定路径的文件。

传入参数：待处理文件路径。

算法流程：

1) 打开输入文件；
2) 调用 FillInfo 函数填充字符出现次数信息；
3) 获取压缩后的文件名，并打开输出文件；
4) 调用 CompressCore 函数执行压缩核心逻辑；
5) 关闭文件。

算法时间复杂度：$O(n)$

```
1. void FileCompress::Compress(const std::string& FilePath) {
2.     FILE* input = fopen(FilePath.c_str(), "rb");
3.     if (NULL == input) {
4.         std::cout << FilePath << " Not Found !" << std::endl;
5.         exit(1);
6.     }
7.
8.     FillInfo(input);
9.
10.    std::string CompressFileName;
11.    GetFileName(FilePath, CompressFileName);
12.    CompressFileName += ".zlx";
13.
14.    FILE* output = fopen(CompressFileName.c_str(), "wb");
15.    if (NULL == output) {
```

```
        std::cout << CompressFileName << " Can Not Be Create !" << std::endl;
        exit(2);
    }

    CompressCore(input, output, FilePath);

    fclose(input);
    fclose(output);
}
```

4.3.2. UnCompress 函数

该函数用于解压缩指定路径的文件

传入参数：文件路径。

算法流程：

1) 打开输入文件；
2) 调用“GetHead”函数读取压缩文件头部信息；
3) 创建输出文件；
4) 初始化哈夫曼树；
5) 调用“UnCompressCore”函数执行解压缩核心逻辑；
6) 关闭文件。

算法时间复杂度：$O(n)$

```
void FileCompress::UnCompress(const std::string& FilePath) {
    FILE* input = fopen(FilePath.c_str(), "rb");
    if (NULL == input) {
        std::cout << FilePath << " Not Found !" << std::endl;
        exit(3);
    }

    // 处理头部信息
    std::string Postfix;
    GetHead(input, Postfix);

    // 创建输出文件
    size_t begin = FilePath.find_first_of("\\");
    if (begin == std::string::npos)
        begin = -1;
    size_t end = FilePath.find_last_of(".");
```

```
17.     if (end == std::string::npos)
18.         end = FilePath.length();
19.     std::string FileName = FilePath.substr(begin + 1, end - begin - 1);
20.     FileName += Postfix;
21.     FILE* output = fopen(FileName.c_str(), "wb");
22.     if (NULL == output) {
23.         std::cout << FileName << " Can Not Open !" << std::endl;
24.         exit(4);
25.     }
26.
27.     int i = 0;
28.     // 填充字符
29.     for (; i < 256; ++i) {
30.         info[i].ch = i;
31.     }
32.
33.     CodeInfo invalid;
34.     invalid.cnt = 0;
35.     HuffmanTree<CodeInfo> hfm(info, 256, invalid);
36.
37.     UnCompressCore(input, output, hfm.GetRoot());
38.
39.     fclose(input);
40.     fclose(output);
41. }
```

4.3.3. UnCompressCore 函数

本函数实现解压缩核心逻辑；

传入参数：输入数据、输出数据、哈夫曼树结构。

算法流程：

1) 初始化读取和写入缓冲区；
2) 从输入文件中读取压缩数据；
3) 根据哈夫曼解码数据并写入输出文件。

算法时间复杂度：$O(m)$（m 为文件大小）

```
1. void FileCompress::UnCompressCore(FILE* input, FILE* output,
                                     HuffmanTreeNode<CodeInfo>* pRoot) {
2.     assert(NULL != input);
3.     assert(NULL != output);
4.
5.     unsigned char ReadBuf[_SIZE_];
6.     unsigned char WriteBuf[_SIZE_];
7.     std::memset(WriteBuf, '\0', _SIZE_);
8.
9.     size_t n;
10.    size_t w_idx = 0;
11.    size_t pos = 0;
12.    HuffmanTreeNode<CodeInfo>* pCur = pRoot;
13.    long long file_len = pRoot->_weight.cnt;
14.    do {
15.        memset(ReadBuf, '\0', _SIZE_);
16.        n = fread(ReadBuf, 1, _SIZE_, input);
17.
18.        // 转换 ReadBuf 至 WriteBuf
19.        size_t r_idx = 0;
20.        for (; r_idx < n; r_idx++) {
21.            // 转换单个字节
22.            unsigned char ch = ReadBuf[r_idx];
23.            for (; pos < 8; pos++, ch <<= 1) {
24.                if ((ch & 0x80) == 0x80) {
25.                    pCur = pCur->pRight;
26.                } else {
27.                    pCur = pCur->pLeft;
28.                }
29.
30.                if (NULL == pCur->pLeft && NULL == pCur->pRight) {
```

```
                    WriteBuf[w_idx++] = pCur->_weight.ch;
                    pCur = pRoot;
                    if (w_idx == _SIZE_) {
                        fwrite(WriteBuf, 1, w_idx, output);
                        memset(WriteBuf, '\0', _SIZE_);
                        w_idx = 0;
                    }
                    file_len--;
                }  // if
                if (file_len == 0)
                    break;
            }  // for
            if (pos == 8)
                pos = 0;

        }  // for

    } while (n > 0);

    if (w_idx < _SIZE_ && w_idx > 0)
        fwrite(WriteBuf, 1, w_idx, output);
}
```

4.3.4. FillInfo 函数

本函数用于填充字符出现次数信息并生成哈夫曼编码。

算法流程：

1) 初始化 info 数组；
2) 统计每个字符在文件中出现的次数；
3) 构建哈夫曼树；
4) 生成每个字符的哈夫曼编码。

时间复杂度：$O(n\log n)$

```
1. void FileCompress::FillInfo(FILE* src) {
2.     assert(src);
3.
4.     int i = 0;
5.     // 填充字符
6.     for (; i < 256; ++i) {
7.         info[i].ch = i;
8.     }
9.
10.    // 填充出现次数
11.    unsigned char buf[_SIZE_];
12.    size_t n;
13.    do {
14.        n = fread(buf, 1, _SIZE_, src);
15.        size_t idx = 0;
16.        while (idx < n) {
17.            info[buf[idx++]].cnt++;
18.        }
19.    } while (n > 0);
20.
21.    // 填充编码
22.    CodeInfo invalid;
23.    invalid.cnt = 0;
24.    HuffmanTree<CodeInfo> hfm(info, 256, invalid);
25.
26.    FillCode(hfm.GetRoot());
27. }
```

4.3.5. CompressCore 函数

本函数用于实现压缩核心功能。

传入参数：待处理文件、文件路径。

算法流程：

1) 将输入文件的指针重置到文件开头；
2) 保存压缩文件的编码头信息；
3) 读取输入文件并进行编码转换，逐字节写入输出文件。

```
1. void FileCompress::CompressCore(FILE* src, FILE* dst, const std::string&
                                   FilePath) {
2.     assert(NULL != src);
3.     assert(NULL != dst);
4.
5.     fseek(src, 0, SEEK_SET);
6.
7.     unsigned char buf[_SIZE_];
8.     unsigned char out[_SIZE_];
9.     int out_idx = 0;
10.    size_t n;
11.    int pos = 0;
12.    unsigned char ch = 0;
13.
14.    SaveCode(dst, FilePath);
15.
16.    // 读数据
17.    do {
18.        // 依次取每个字节转换
19.        memset(buf, '\0', _SIZE_);
20.        n = fread(buf, 1, _SIZE_, src);
21.        size_t idx = 0;
22.        while (idx < n) {
23.            // 转换单个字节
24.            const std::string& CurCode = info[buf[idx++]].code;
25.            size_t len = CurCode.length();
26.            size_t i_len = 0;
27.            while (i_len < len) {
28.                for (; pos < 8 && i_len < len; pos++) {
29.                    ch <<= 1;
30.                    if (CurCode[i_len++] == '1') {
31.                        ch |= 1;
```

```
                    }
                }

                // 先缓存到 out
                if (8 == pos) {
                    out[out_idx++] = ch;
                    pos = 0;
                    ch = 0;

                    // 输出到文件
                    if (_SIZE_ == out_idx) {
                        fwrite(out, 1, out_idx, dst);
                        out_idx = 0;
                    }
                }
            }  // while
        }  // while
    } while (n > 0);

    // 处理剩余的位
    if (8 > pos && 0 < pos) {
        int j = 0;
        while (j++ < 8 - pos)
            ch <<= 1;
        out[out_idx++] = ch;
    }

    // 处理剩余的字节
    if (out_idx > 0)
        fwrite(out, 1, out_idx, dst);
}
```