综合推荐

对于日记推荐和景点推荐，逻辑是类似的，这里只取景点进行详细介绍。

前端向后端发送的数据结构：

```
boolean views;           //是否根据浏览量排序返回结果
boolean score;           //是否根据评分排序返回结果
int length;              //返回结果的最大个数
List<String> keywords;   //关键词指用户在搜索框中输入的内容
List<String> tags;       //用户选择的一些标签
```

检索和排序逻辑：

- 先根据`tags`筛选出符合标签的所有旅游景点，如果标签为空则选中所有旅游景点进行下一步过滤。
- 对于旅游景点的`name`字段和`description`字段，根据用户提供的`keywords`字段分别进行字符串匹配，将每次匹配返回的结果存入自建数据结构`Set`中，避免重复选中。
- 根据用户要求对选中的结果进行排序，返回前`length`个结果。
- 如果`tags`为空，则不根据标签筛选旅游景点，返回所有的旅游景点进入下一步过滤。
- 如果`keywords`为空，则不根据关键词进行进一步的过滤，返回根据`tags`筛选出的所有旅游景点。
- 如果`length`的值小于等于0，则返回所有过滤后的结果，不对长度进行限制。
- `views`和`score`的取值和排序行为的关系满足下表

取值关系	排序行为
views=true && score=false	按照浏览量从高到低排序
views=false && score=true	按照评分从高到低排序
views=true && score=true	按照“评分优先，浏览量次之”的顺序从高到低排序
views=false && score=false	按照景点名称的字典顺序排序

搜索逻辑

1. **获取标签和日记**：

- 从请求中获取标签列表。如果标签列表不为空，则遍历每个标签，通过`tagService`获取与标签相关的日记，并将这些日记添加到自定义集合`diarySet`中。否则，获取所有日记并添加到集合中。

2. **过滤关键词**：

- 从请求中获取关键词列表。如果关键词列表为空，则将集合中的所有日记添加到最终列表`diaryList`中。否则，遍历集合中的每篇日记，使用 BM 算法和 AC 自动机算法检查标题和内容是否包含关键词。如果匹配成功，则将日记添加到最终列表中。

3. **排序和截取**：

- 将最终列表转换为数组，并根据请求中的排序要求（视图数或评分）获取相应的比较器。
- 如果请求的长度小于等于0，使用快速排序按降序排序整个数组。否则，获取数组中最后 N 个元素并反转顺序。

4. **返回结果**：

- 将排序后的数组转换为列表并返回，作为最终结果。

```
@Override
public List<Diary> getSortedDiary(GetSortedResultRequest req){
    // 获取请求中的标签列表
    List<String> tags = req.getTags();
    // 自定义集合类，用于存储日记
    MySet<Diary> diarySet = new MySet<>();

    // 如果标签列表不为空
    if(tags != null && !tags.isEmpty()) {
        // 遍历每个标签
        for (String tag : tags) {
            // 通过标签获取对应的日记列表
            List<Diary> tempDiaries = tagService.getDiaryByTag(tag);
            // 将每篇日记添加到自定义集合中，键为用户名和标题的组合
            for (Diary diary : tempDiaries) {
                diarySet.add(diary.getUsername() + "@" + diary.getTitle(), diary);
            }
        }
    } else {
        // 如果标签列表为空，则获取所有日记
        for(Diary diary : getAllDiaries()) {
            // 将每篇日记添加到自定义集合中
            diarySet.add(diary.getUsername() + "@" + diary.getTitle(), diary);
        }
    }

    // 获取请求中的关键词列表
    List<String> keywords = req.getKeywords();
    // 用于存储最终过滤后的日记列表
    List<Diary> diaryList = new ArrayList<>();

    // 如果关键词列表为空
    if(keywords == null || keywords.isEmpty()) {
        // 将所有日记添加到最终列表中
        diaryList.addAll(diarySet.values());
    } else {
        // 遍历自定义集合中的每篇日记
        for(Diary diary : diarySet.values()) {
            String title = diary.getTitle();
            String content = diary.getContent();
            boolean isTitleMatch = false;
            // 使用 AC 自动机算法匹配内容中的关键词
            boolean isContentMatch = MatchUtils.acAutomatonMatch(keywords, content);
            // 遍历每个关键词，使用 BM 算法匹配标题中的关键词
            for(String keyword : keywords){
                if(MatchUtils.bmMatch(keyword, title) != -1){
                    isTitleMatch = true;
                    break;
                }
```

```
            }
            // 如果标题或内容匹配上关键词，则将该日记添加到最终列表中
            if(isTitleMatch || isContentMatch){
                diaryList.add(diary);
            }
        }
    }

    // 将最终列表转换为数组
    Diary[] diaryArray = diaryList.toArray(new Diary[0]);
    // 获取比较器，根据请求中的视图数和评分进行排序
    Comparator<Diary> comparator = Diary.getComparator(req.isViews(),
req.isScore());

    // 如果请求中的长度小于等于0，使用快速排序按降序排序
    if(req.getLength() <= 0){
        SortUtils.quickSort(diaryArray, comparator.reversed());
    } else {
        // 否则，获取最后 N 个元素，并反转顺序
        diaryArray = SortUtils.getLastN(diaryArray, req.getLength(),
comparator);
        SortUtils.reverse(diaryArray);
    }

    // 将排序后的数组转换为列表并返回
    return Arrays.asList(diaryArray);
}
```

地图导航

在前端的访问接口上，我们充分考虑到了用户操作的特点：从一个固定的下拉框里选择地点，或者必须准确背下来地点的名称/id都太过麻烦和原始，不利于用户操作。

因此，我选择将导航的接口设计为只需要传递用户两次点击的坐标，然后由后端来自动判断这两个坐标距离哪一个点最近，然后将它靠过去再开始导航。用户**只需要点击开始选择地点的按钮，然后任意点击他想要的地点就可以完成选择**

在后端一处不起眼的改动，却使得用户的操作体验大大提升，不必在近百个地点中寻找自己的目的地

（此处列举的是点到点的导航，途径多点的导航于此类似， 也是传递一系列的点击坐标即可）

```
@PostMapping({"/p2p"})
public Result<List<Point>> P2Pguide(int x1, int y1,int x2,int y2) {
    Point from = pointService.getByPos(x1, y1);
    Point to = pointService.getByPos(x2, y2);

    List<Point> guidePaths = guideService.directGuide(from.getId(),
to.getId());
    return getResult(guidePaths);
}
@PostMapping({"/by_many"})
//注意，传入参数中，第一个点是起始点的id，后面的是需要经过的点的id，不用把最终回到原点的需求也写进来
public Result<List<Point>> byManyguide(@RequestBody byManyPostBody pointdata)
{
```

```
    List<Integer> pointIds = pointdata.getPointdata().stream()
        .filter(a->a.getX()>50&&a.getY()>50)
        .map(a -> pointService.getByPos(a.getX(), a.getY()).getId())
        .collect(Collectors.toList());
    List<Point> guidePaths = guideService.byManyPointsGuide(pointIds);
    return getResult(guidePaths);
}
//.....//
 @Override
public Point getByPos(int x, int y) {
    Point p=new Point(-1,x,y);
    AtomicReference<Point> res=new AtomicReference<>();
    AtomicInteger dist= new AtomicInteger(1 << 30);
    points.forEach((k,v)->{
        int tempD=p.getDistance(v);
        if(tempD< dist.get()) {
            dist.set(tempD);
            res.set(v);
        }
    });
    assert res.get()!=null;
    return res.get();
}
```

点到点的导航

综合考虑了SPFA、dijstra、floyd这几类最短路算法。

由于地图导航的情形下不存在负权边/环，所以没有必要采用SPFA；而floyd算法一次性O(E^3)过于臃肿，所以我们使用了dijstra+缓存的方案来完成最短路导航

复杂度分析：采用堆优化，最差为O(ElgV)，并且采取了缓存路径的方式，空间换时间，每次导航都可以保存部分路径从而加速。（并且此情况到缓存完所有的点到点路径也只需要$O(E^2lgV)$的复杂度，在稀疏图上效率高于floyd）

所以期望在进行了E次导航之后，复杂度可以降至O(1)（直接从缓存取路径即可）

```
public List<Point> dijkstra(int start, int end) {
    //起点start到各个点的路径是否有缓存
    if (!cachedPaths[start][end].isEmpty()) {
        return cachedPaths[start][end];
    }

    //cachedPaths[start][end].add(points.get(start));
    int[] distance = new int[points.size()], used = new int[points.size()];
    PriorityQueue<Node> node = new PriorityQueue<>(Comparator.comparingInt(o
-> o.value));

    Arrays.fill(distance, Integer.MAX_VALUE / 2);
    node.add(new Node(start, 0));
    distance[start] = 0;
    used[start] = 1;
    while (!node.isEmpty()) {
        //要被用来开始松弛的城市N
        int city = node.poll().to;
        Point from = points.get(city);
```

```
        used[city] = 1;
        List<Point> arr = map.get(city);
        //if (arr.isEmpty()&&city!=end) return new ArrayList<>();
        //遍历这个城市的邻边
        for (Point n : arr) {
            int toCity = n.getId();

            if (used[toCity] != 0) continue;
            if (cachedPaths[city][toCity].isEmpty()) cachedPaths[city][toCity].add(n);
            //如果有哪个相邻的点，满足：从已知的起始点到达该点的方式的距离，大于从起始点到达N再从N到达这个点的距离，就替换到达方式为后者
            if (distance[toCity] > distance[city] + n.getDistance(from)) {
                distance[toCity] = distance[city] + n.getDistance(from);
                node.offer(new Node(toCity, distance[toCity]));

                ArrayList<Point> temp = new ArrayList<>();
                temp.addAll(cachedPaths[start][city]);
                temp.addAll(cachedPaths[city][toCity]);
                cachedPaths[start][toCity] = temp;
            }
        }
    }
    cachedPaths[end][start] = cachedPaths[start][end];
    f[start]=distance;
    if (distance[end] != Integer.MAX_VALUE / 2) {
        return cachedPaths[start][end];
    } else {
        return new ArrayList<>();
    }
    //若返回空表，则为不连通
}
```

```
public void floydRun(int max) {//未用到，因为发现dij效率更高，求路径也方便
    for (int k = 0; k < max; k++) {
        for (int i = 0; i < max; i++) {
            if (f[i][k] != Integer.MAX_VALUE / 2) {
                for (int j = 0; j < max; j++) {
                    if (f[k][j] != Integer.MAX_VALUE / 2) {
                        f[i][j] = max(f[i][j], f[i][k] * f[k][j]);
                        //f[j][i] = min(f[j][i], f[i][k] + f[k][j]);
                    }
                }
            }
        }
    }
}
```

途径多点的导航

我们考虑到了在需要途径的点数非常多的情况下（超过20个点，虽然这种概率非常小），一次导航需要的时间就会达到秒级，可能对用户的操作体验造成影响。

因此，我们选择在**途经点太多时**放弃求精确解，转而采用简单的搜索方法来求出一个遍历途经点的**近似解。**

在这样的设计下，20个途经点以内的导航会在一秒内给出最短遍历路径的精确解，而更多的点则会瞬间给出一个近似解

不过在测试中，这样超过20个途经点的近似解在人眼看来几乎不会有什么绕路的情况，说明并没有为了用户体验而牺牲掉正确性，而是兼顾了两者。

```
@Override
public List<Point> byManyPointsGuide(List<Integer> passedPoints) {
    if(passedPoints.isEmpty()) return new ArrayList<>();
    Point start=sr.getPoint(passedPoints.get(0));

    List<Point> res=new ArrayList<>();
    res.add(start);
    int passedSize = passedPoints.size();

    if(passedSize >18){
        //如果size大于20，就采取近似解，通过搜索剪枝来确定路径
        while (!passedPoints.isEmpty()){
            Point now=res.get(res.size()-1);
             Point temp = new Point();
            int min = 1 << 30,rmi=-1;
            for (int i = 0; i < passedPoints.size(); i++) {
                int a=passedPoints.get(i);
                if(sr.floydAsk(now.getId(), a)< min){
                    min =sr.floydAsk(now.getId(), a);
                    temp =sr.getPoint(a);
                    rmi=i;
                }
            }
            passedPoints.remove(rmi);
            res.add(temp);
        }
        res.add(start);
    }else{
        //如果size小于20，使用哈密顿回路寻找到精确解，状压dp
        int[][] dp=new int[1<<passedSize][passedSize];
        List<Point>[] atPointPaths=new List[passedSize];//不论走过了什么点，怎么走的，存储最后位于k点的中距离最短的走法
        int[] atPointDist=new int[passedSize];
        for (int i = 0; i < atPointDist.length; i++) {
            atPointPaths[i]=new ArrayList<>();
        }
        for (int i = 0; i < dp.length; i++) {
            Arrays.fill(dp[i], Integer.MAX_VALUE/2);
        }
        dp[1][0]=0;


```

```
        for(int i=0;i<(1<<passedSize);i++) {//i代表的是一个方案的集合，其中每个位置的0/1代表没有/有经过这个点
            for(int j=0;j<passedSize;j++) {//枚举当前在哪个点
                if(((i>>j)&1)!=0) {//如果i代表的状态中有j，也就是可以表示“经过了i中bit为1的点，且当前处于j点”
                    for(int k=0;k<passedSize;k++) {//枚举所有可以走到到达j的点
                        if((i-(1<<j)>>k&1)!=0) {//在i状态中，走到j这个点之前，是否可以停在k点。如果是，才能从k转移到j
                            int dist = sr.floydAsk(sr.getPoint(passedPoints.get(k)).getId(), sr.getPoint(passedPoints.get(j)).getId());
                            if(dp[i-(1<<j)][k]+dist<dp[i][j]){//如果从k走到j比原先的更短
                                dp[i][j]=dp[i-(1<<j)][k]+ dist;
                                atPointPaths[j]=new ArrayList<>(atPointPaths[k]);//那么走到j点的路径就必然是走到k点，再到j的

 while(atPointPaths[j].remove(sr.getPoint(passedPoints.get(j))));

 atPointPaths[j].add(sr.getPoint(passedPoints.get(j)));//atPP存储了从起点到达每一个点的路径
                                atPointDist[j]=atPointDist[k]+dist;
                            }
                        }
                    }
                }
            }
        }
        //计算完成了遍历需要pass的所有点的距离，也就是得到了所有的哈密顿路径值，然后还需要走回到出发点（由于项目要求）
        int min=0;
        for (int i = 0; i < atPointPaths.length; i++) {
            if(atPointPaths[i].isEmpty()) continue;

 atPointDist[i]+=sr.floydAsk(atPointPaths[i].get(atPointPaths[i].size()-1).getId(),start.getId());//获得再回到start的距离
            atPointPaths[i].add(start);
            if(atPointDist[i]>atPointDist[min]) min=i;
        }
        res=atPointPaths[min];
    }
    res.add(0, start);

    List<Point> expandRes=new ArrayList<>();
    for (int i = 1; i < res.size(); i++) {
        List<Point> temp = directGuide(res.get(i - 1).getId(), res.get(i).getId());
        temp.remove(0);
        expandRes.addAll(temp);
    }
    expandRes.add(0, start);

    for (Integer passedPoint : passedPoints) {
        if(passedPoint!=0&&!expandRes.contains(sr.getPoint(passedPoint))) {
            expandRes=expandRes.subList(0, 1);
            break;
```

```
        }
    }

    return expandRes;
}
```

导航系统的最上层封装：计算获得的导航路径像素长度，并映射到现实距离，给用户做出提示

```
    private Result<List<Point>> getResult(List<Point> guidePaths) {
        int len = 0;
        for (int i = 0; i < guidePaths.size() - 1; i++) {
            len += guidePaths.get(i).getDistance(guidePaths.get(i + 1));
        }
        len *= PIXEL_2_METER_ARG;

        Result<List<Point>> res = guidePaths.size()<=1 ?
                Result.error("导航失败！") : Result.success("导航成功，路线长度约"
+ len + "米");
        res.data(guidePaths);
        return res;
    }
```

自建缓存

为了加速用户的查询操作，我们实现了一个综合使用多种数据结构的自建缓存系统。这些缓存主要包括 AVL 树、二叉搜索树、MultiMap、多路搜索树和红黑树等，每种数据结构都有其特定的应用场景和优势。以下是对各个缓存及其数据结构的详细介绍。

1. AVL 树

实现类：`AVLTreeConfig`

数据结构：AVL 树

用途：存储和快速查找 JSON 格式的图数据。

实现细节：

- AVL 树是一种自平衡二叉搜索树，能够保证在最坏情况下的查找、插入和删除操作的时间复杂度为 O(log n)。
- 在 `AVLTreeConfig` 中，我们从数据库中获取所有图数据，并将其插入到 AVL 树中。
- 如果数据获取失败，系统会记录错误日志并返回空的 AVL 树实例。

代码示例：

```
@Configuration
@Slf4j
public class AVLTreeConfig {
    @Autowired
    GraphMapper graphMapper;

    @Bean
    public AVLTree<JsonGraph> graphAVLTree() {
        AVLTree<JsonGraph> avlTree = new AVLTree<>();
        try {
```

```
            List<JsonGraph> jsonGraphs = graphMapper.getAllGraphs();
            if (jsonGraphs == null || jsonGraphs.isEmpty()) {
                log.error("No graphs found or failed to fetch from the database.");
                return avlTree;
            }
            for (JsonGraph jsonGraph : jsonGraphs) {
                avlTree.insert(jsonGraph.getName(), jsonGraph);
            }
        } catch (Exception e) {
            log.error("Error initializing AVLTree", e);
        }
        return avlTree;
    }
}
```

2. 二叉搜索树

实现类：`BinarySearchTreeConfig`

数据结构：二叉搜索树

用途：存储和快速查找旅游景点数据。

实现细节：

- 二叉搜索树（BST）是一种二叉树，其中每个节点都满足左子节点小于根节点，右子节点大于根节点的性质。
- 在 `BinarySearchTreeConfig` 中，我们从数据库中获取所有旅游景点数据，并将其插入到 BST 中。
- 如果数据获取失败，系统会记录错误日志并返回空的 BST 实例。

代码示例：

```
@Configuration
@Slf4j
public class BinarySearchTreeConfig {
    @Autowired
    GraphMapper graphMapper;

    @Bean
    public BinarySearchTree<Tourism> tourismBinarySearchTree() {
        BinarySearchTree<Tourism> binarySearchTree = new BinarySearchTree<>();
        try {
            List<Tourism> tourisms = graphMapper.getAllTourism();
            if (tourisms == null || tourisms.isEmpty()) {
                log.error("No tourism found or failed to fetch from the database.");
                return binarySearchTree;
            }
            for (Tourism tourism : tourisms) {
                binarySearchTree.insert(tourism.getName(), tourism);
            }
        } catch (Exception e) {
```

```
            log.error("Error initializing BinarySearchTree", e);
        }
        return binarySearchTree;
    }
}
```

3. MultiMap

实现类：`MultiMapConfig`

数据结构：MultiMap

用途：存储和快速查找日记标签、旅游景点标签、标签与日记、标签与旅游景点的映射关系。

实现细节：

- MultiMap 是一种允许一个键对应多个值的数据结构，适用于一对多关系的存储需求。
- 在 `MultiMapConfig` 中，我们分别创建了 `diaryTagMultiMap`、`tourismTagMultiMap`、`tagDiaryMultiMap` 和 `tagTourismMultiMap`，以存储不同的映射关系。
- 从数据库中获取相关数据并插入到 MultiMap 中。如果数据获取失败，系统会记录错误日志并返回空的 MultiMap 实例。

代码示例：

```
@Configuration
@Slf4j
public class MultiMapConfig {
    @Autowired
    TagMapper tagMapper;
    @Autowired
    DiaryMapper diaryMapper;
    @Autowired
    GraphMapper graphMapper;
    static int CAPACITY = 128;

    @Bean
    public MultiMap<String, Tag> diaryTagMultiMap() {
        MultiMap<String, Tag> multiMap = new MultiMap<>(CAPACITY);
        try {
            List<CompressedDiary> compressedDiaries =
diaryMapper.getAllCompressedDiaries();
            if (compressedDiaries == null || compressedDiaries.isEmpty()) {
                log.error("No compressed diaries found or failed to fetch
from the database.");
                return multiMap;
            }
            for (CompressedDiary compressedDiary : compressedDiaries) {
                String diaryName = compressedDiary.getUsername() + "@" +
compressedDiary.getTitle();
                for (String tagName :
tagMapper.getTagsByDiaryName(diaryName)) {
                    multiMap.put(diaryName, new Tag(tagName));
                }
            }
        } catch (Exception e) {
```

```
            log.error("Error initializing MultiMap", e);
        }
        return multiMap;
    }

    @Bean
    public MultiMap<Tourism, Tag> tourismTagMultiMap() {
        MultiMap<Tourism, Tag> multiMap = new MultiMap<>(CAPACITY);
        try {
            List<Tourism> tourisms = graphMapper.getAllTourism();
            if (tourisms == null || tourisms.isEmpty()) {
                log.error("No tourisms found or failed to fetch from the
database.");
                return multiMap;
            }
            for (Tourism tourism : tourisms) {
                for (String tagName :
tagMapper.getTagsByTourismName(tourism.getName())) {
                    multiMap.put(tourism, new Tag(tagName));
                }
            }
        } catch (Exception e) {
            log.error("Error initializing MultiMap", e);
        }
        return multiMap;
    }

    @Bean
    public MultiMap<Tag, String> tagDiaryMultiMap() {
        MultiMap<Tag, String> multiMap = new MultiMap<>(CAPACITY);
        try {
            List<String> tags = tagMapper.getAllTags();
            if (tags == null || tags.isEmpty()) {
                log.error("No tags found or failed to fetch from the
database.");
                return multiMap;
            }
            for (String tag : tags) {
                List<String> diaryNames = tagMapper.getDiaryNameByTag(tag);
                for (String diaryName : diaryNames) {
                    multiMap.put(new Tag(tag), diaryName);
                }
            }
        } catch (Exception e) {
            log.error("Error initializing MultiMap", e);
        }
        return multiMap;
    }

    @Bean
    public MultiMap<Tag, Tourism> tagTourismMultiMap() {
        MultiMap<Tag, Tourism> multiMap = new MultiMap<>(CAPACITY);
        try {
            List<String> tags = tagMapper.getAllTags();
            if (tags == null || tags.isEmpty()) {
```

```
                log.error("No tags found or failed to fetch from the database.");
                return multiMap;
            }
            for (String tag : tags) {
                List<String> tourismNames = tagMapper.getTourismNameByTag(tag);
                for (String tourismName : tourismNames) {
                    multiMap.put(new Tag(tag), graphMapper.getTourismByName(tourismName));
                }
            }
        } catch (Exception e) {
            log.error("Error initializing MultiMap", e);
        }
        return multiMap;
    }
}
```

4. 红黑树

实现类：`RedBlackTreeConfig`

数据结构：红黑树

用途：存储和快速查找拥挤图数据。

实现细节：

- 红黑树是一种自平衡二叉搜索树，具有保证最坏情况下 O(log n) 时间复杂度的优点。
- 在 `RedBlackTreeConfig` 中，我们从数据库中获取所有拥挤图数据，并将其插入到红黑树中。
- 如果数据获取失败，系统会记录错误日志并返回空的红黑树实例。

代码示例：

```
@Configuration
@Slf4j
public class RedBlackTreeConfig {
    @Autowired
    GraphMapper graphMapper;

    @Bean
    public RedBlackTree<CrowdedGraph> graphRedBlackTree() {
        RedBlackTree<CrowdedGraph> redBlackTree = new RedBlackTree<>();
        try {
            List<JsonCrowdedGraph> crowdedGraphs = graphMapper.getAllCrowdedGraphs();
            if (crowdedGraphs == null || crowdedGraphs.isEmpty()) {
                log.error("No graphs found or failed to fetch from the database.");
                return redBlackTree;
            }
            for (JsonCrowdedGraph json : crowdedGraphs) {
                redBlackTree.insert(json.getName(), json.getCrowdedGraph());
            }
```

```
        } catch (Exception e) {
            log.error("Error initializing RedBlackTree", e);
        }
        return redBlackTree;
    }
}
```

5. 字典树

实现类：`TrieTreeConfig`

数据结构：字典树（Trie）

用途：存储和快速查找日记数据。

实现细节：

- 字典树是一种专门用于快速前缀查询的数据结构，适用于高效存储和检索大量字符串。
- 在 `TrieTreeConfig` 中，我们从数据库中获取所有压缩格式的日记数据，并使用哈夫曼编码解码这些数据后，插入到字典树中。
- 如果数据获取失败，系统会记录错误日志并返回空的字典树实例。

代码示例：

```
package com.ymj.tourstudy.config;

import com.alibaba.fastjson.JSON;
import com.ymj.tourstudy.mapper.DiaryMapper;
import com.ymj.tourstudy.pojo.CompressedDiary;
import com.ymj.tourstudy.pojo.Diary;
import com.ymj.tourstudy.utils.HuffmanResult;
import com.ymj.tourstudy.utils.HuffmanUtils;
import com.ymj.tourstudy.utils.TrieTree;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.List;

@Configuration
@Slf4j
public class TrieTreeConfig {
    @Autowired
    DiaryMapper diaryMapper;

    @Bean
    public TrieTree<Diary> diaryTrieTree() {
        TrieTree<Diary> trieTree = new TrieTree<>();
        try {
            List<CompressedDiary> compressedDiaries =
diaryMapper.getAllCompressedDiaries();
            if (compressedDiaries == null || compressedDiaries.isEmpty()) {
                log.error("No compressed diaries found or failed to fetch
from the database.");
```

```
                return trieTree;
            }
            for (CompressedDiary compressedDiary : compressedDiaries) {
                String compressedContent =
compressedDiary.getCompressedContent();
                if (compressedContent == null) {
                    log.warn("Encountered null compressed content.");
                    continue;
                }
                HuffmanResult result = JSON.parseObject(compressedContent,
HuffmanResult.class);
                if (result == null) {
                    log.warn("Failed to parse HuffmanResult from content.");
                    continue;
                }
                String content = HuffmanUtils.decode(result);
                if (content == null) {
                    log.warn("Decoded content is null.");
                    continue;
                }
                Diary diary = JSON.parseObject(content, Diary.class);
                if (diary == null) {
                    log.warn("Failed to parse Diary from decoded content.");
                    continue;
                }
                String key = diary.getUsername() + "@" + diary.getTitle();
                trieTree.insert(key, diary);
            }
        } catch (Exception e) {
            log.error("Error initializing TrieTree", e);
        }
        return trieTree;
    }
}
```

日记压缩

构建哈夫曼树

- **构建**：从最小频率的节点开始，不断将两个最小的节点合并为一个新节点，新节点的频率为两个子节点频率之和，直到只剩一个节点，即根节点。
- **编码**：从根节点开始，向左赋予'0'，向右赋予'1'，生成每个字符的编码。
- **解码**：根据编码从根节点遍历到叶节点，解析出原始数据。

```
package com.ymj.tourstudy.utils;

import java.util.Comparator;

public class HuffmanUtils {
    /**
     * 根据输入字符串构建哈夫曼树。
     * @param text 输入的字符串
     * @return 构建好的哈夫曼树的根节点
     */
```

```
    private static Node buildHuffmanTree(String text) {
        // 统计每个字符出现的频率
        MyHashMap<Character, Integer> freqMap = new MyHashMap<>(256);
        for (char c : text.toCharArray()) {
            freqMap.put(c, freqMap.get(c) == null ? 1 : freqMap.get(c) + 1);
        }

        // 使用优先队列构建哈夫曼树
        MyPriorityQueueMinHeap<Node> priorityQueue = new MyPriorityQueueMinHeap<>(Comparator.comparingInt(a -> a.frequency));
        // 将每个字符及其频率作为一个节点插入优先队列
        freqMap.forEach((character, frequency) -> {
            priorityQueue.insert(new Node(character, frequency, null, null));
        });
        // 从优先队列中取出两个频率最小的节点，合并成一个新节点，然后再插入优先队列
        while (priorityQueue.size() > 1) {
            Node left = priorityQueue.remove();
            Node right = priorityQueue.remove();
            Node parent = new Node('\0', left.frequency + right.frequency, left, right);
            priorityQueue.insert(parent);
        }

        return priorityQueue.remove();
    }

    /**
     * 根据哈夫曼树为每个字符生成编码。
     * @param root 哈夫曼树的根节点
     * @param code 当前编码
     * @param codes 存储字符及其编码的映射
     */
    private static void generateCodes(Node root, String code, MyHashMap<Character, String> codes) {
        if (root == null) return;

        if (root.left == null && root.right == null) {
            codes.put(root.character, code);
        }

        generateCodes(root.left, code + "0", codes);
        generateCodes(root.right, code + "1", codes);
    }

    /**
     * 对输入的字符串进行哈夫曼编码。
     * @param text 输入的字符串
     * @return 哈夫曼编码结果，包括编码后的字符串和哈夫曼树根节点
     */
    public static HuffmanResult encode(String text) {
        Node root = buildHuffmanTree(text);
        MyHashMap<Character, String> codes = new MyHashMap<>(256);
        generateCodes(root, "", codes);

```

```
        StringBuilder encoded = new StringBuilder();
        for (char c : text.toCharArray()) {
            encoded.append(codes.get(c));
        }

        return new HuffmanResult(root, encoded.toString());
    }

    /**
     * 根据哈夫曼树和编码后的字符串进行解码。
     * @param huffmanResult 包含哈夫曼树根节点和编码后字符串的结果对象
     * @return 解码后的原始字符串
     */
    public static String decode(HuffmanResult huffmanResult) {
        StringBuilder decoded = new StringBuilder();
        Node current = huffmanResult.getRoot();
        for (int i = 0; i < huffmanResult.getEncodedData().length(); i++) {
            if (huffmanResult.getEncodedData().charAt(i) == '0') {
                current = current.left;
            } else {
                current = current.right;
            }

            if (current.left == null && current.right == null) {
                decoded.append(current.character);
                current = huffmanResult.getRoot(); // 重置为根节点开始下一个字符的解码
            }
        }
        return decoded.toString();
    }

    public static void main(String[] args) {
        String text = "example of huffman encoding";
        HuffmanResult result = HuffmanUtils.encode(text);
        System.out.println("Encoded Huffman Data: " + result.getEncodedData());

        String decodedText = HuffmanUtils.decode(result);
        System.out.println("Decoded Text: " + decodedText);
    }
}
```

用户管理

初始化

将数据库中的所有数据加载到内存中, 具体有用户 id 的哈希表和同组用户的哈希表.

检验操作合法性

```java
/**
 * 验证用户身份是否可以添加/修改此日程
 * 集体类只能由管理员添加，个人类只能由学生添加
 *
 * @param u 用户
 * @param e 日程
 * @return 符合条件返回true，否则返回false
 */
@Override
public boolean identifyUser(User u, Event e) {
    if (u.isAdmin()) {
        return e.getIsGroup();
    }
    return !e.getIsGroup();
}
```

用户注册

```java
/**
 * 用户注册 (只有学生才会注册)
 *
 * @param name     用户名
 * @param mail     用户邮箱
 * @param password 密码
 * @param groupId  组id
 */
@Override
public void register(String name, String mail, String password, Integer groupId) {
    User u = new User(name, password, mail, UserType.USER_STUDENT.getValue(), groupId);
    userMapper.add(u);
    u = userMapper.getByMail(mail);
    // 更新内存中的数据
    List<User> groupUsers = userGroupIdMap.getOrDefault(groupId, new ArrayList<>());
    groupUsers.add(u);
    userGroupIdMap.put(groupId, groupUsers);
    userIdMap.put(u.getUserId(), u);
}
```

相关查询

直接操作哈希表即可

```java
/**
 * 选取同一组的用户
 *
 * @param user 管理员
 * @return 同组用户
 */
```

```java
@Override
public List<User> selectSameGroupUsers(User user) {
    return userGroupIdMap.get(user.getGroupId());
}

/**
 * 根据用户id加载用户
 *
 * @param userId 用户id
 * @return 用户
 */
@Override
public User load(Integer userId) {
    return userIdMap.get(userId);
}

/**
 * 根据用户邮箱加载用户
 *
 * @param mail 用户邮箱
 * @return 用户
 */
@Override
public User loadByMail(String mail) {
    return userMapper.getByMail(mail);
}

/**
 * 判断用户邮箱是否已存在
 *
 * @param mail 用户邮箱
 * @return 用户
 */
@Override
public boolean contains(String mail) {
    return userMapper.getByMail(mail) != null;
}

/**
 * 获取所有的组id
 *
 * @return 组id列表
 */
@Override
public List<Integer> getGroups() {
    return userMapper.getGroups();
}
```

模式匹配

BM算法

- **高效**：在许多实用情况下比传统的KMP算法或简单的暴力匹配方法更高效。
- **跳跃搜索**：能够跳过一些不必要的字符匹配，特别是当模式串不匹配时，能够根据坏字符规则和好后缀规则跳过多个字符。
- **适用性强**：尤其在模式串较长时表现更优。

```
package com.ymj.tourstudy.utils;

import java.util.*;

/**
 * 字符串匹配工具类,实现 Boyer-Moore 算法
 */
public class MatchUtils {
    /**
     * 使用 Boyer-Moore 算法在目标字符串中查找模式串
     *
     * @param pattern 模式串
     * @param target  目标字符串
     * @return 模式串在目标字符串中的起始位置索引,如果没有匹配则返回 -1
     */
    public static int bmMatch(String pattern, String target) {
        int targetLen = target.length(); // 目标串长度
        int patternLen = pattern.length(); // 模式串长度

        // 如果模式串比目标串长,没有可比性,直接返回 -1
        if (patternLen > targetLen) {
            return -1;
        }

        // 如果模式串为空,直接返回 0
        if (patternLen == 0) {
            return 0;
        }

        int[] badTable = buildBadTable(pattern); // 获得坏字符数值的数组
        int[] goodTable = buildGoodTable(pattern); // 获得好后缀数值的数组

        for (int i = patternLen - 1, j; i < targetLen; ) {
            for (j = patternLen - 1; target.charAt(i) == pattern.charAt(j);
i--, j--) {
                if (j == 0) { // 指向模式串的首字符,说明匹配成功,直接返回就可以了
                    return i;
                }
            }
            // 如果出现坏字符,那么这个时候比较坏字符以及好后缀的数组,哪个大用哪个
            i += Math.max(goodTable[patternLen - j - 1],
badTable[target.charAt(i)]);
        }
        return -1;
    }
```

```java

    /**
     * 构建坏字符表
     *
     * @param pattern 模式串
     * @return 坏字符表数组
     */
    public static int[] buildBadTable(String pattern) {
        final int tableSize = 256;
        int[] badTable = new int[tableSize];
        int patternLen = pattern.length();

        for (int i = 0; i < badTable.length; i++) {
            badTable[i] = patternLen;
        }
        for (int i = 0; i < patternLen - 1; i++) {
            int k = pattern.charAt(i);
            badTable[k] = patternLen - 1 - i;
        }
        return badTable;
    }

    /**
     * 构建好后缀表
     *
     * @param pattern 模式串
     * @return 好后缀表数组
     */
    public static int[] buildGoodTable(String pattern) {
        int patternLen = pattern.length();
        int[] goodTable = new int[patternLen];
        int lastPrefixPosition = patternLen;

        for (int i = patternLen - 1; i >= 0; --i) {
            if (isPrefix(pattern, i + 1)) {
                lastPrefixPosition = i + 1;
            }
            goodTable[patternLen - 1 - i] = lastPrefixPosition - i +
patternLen - 1;
        }

        for (int i = 0; i < patternLen - 1; ++i) {
            int suffixLen = suffixLength(pattern, i);
            goodTable[suffixLen] = patternLen - 1 - i + suffixLen;
        }
        return goodTable;
    }

    /**
     * 判断字符串的子串是否为前缀子串
     *
     * @param pattern 模式串
     * @param p       起始位置
     * @return 如果是前缀子串则返回 true,否则返回 false
     */
    private static boolean isPrefix(String pattern, int p) {
```

```
        int patternLength = pattern.length();
        for (int i = p, j = 0; i < patternLength; ++i, ++j) {
            if (pattern.charAt(i) != pattern.charAt(j)) {
                return false;
            }
        }
        return true;
    }

    /**
     * 计算后缀长度
     *
     * @param pattern 模式串
     * @param p       起始位置
     * @return 后缀长度
     */
    private static int suffixLength(String pattern, int p) {
        int patternLen = pattern.length();
        int len = 0;
        for (int i = p, j = patternLen - 1; i >= 0 && pattern.charAt(i) == pattern.charAt(j); i--, j--) {
            len += 1;
        }
        return len;
    }


}
```

KMP算法

- 效率高：避免了不必要的回溯，比传统的暴力匹配方法更高效。
- 预处理加速：通过LPS数组的预处理，实现了模式串的快速移动，显著减少了比较次数。
- 固定空间需求：空间复杂度仅依赖于模式串长度，与目标字符串长度无关。

```
/**
 * AC自动机实现
 */
private static class ACAutomaton {
    private class Node {
        Map<Character, Node> children = new HashMap<>();
        Node fail;
        Set<String> output = new HashSet<>();
    }

    private Node root;

    public ACAutomaton(List<String> patterns) {
        root = new Node();
        buildTrie(patterns);
        buildFailureLinks();
    }

    private void buildTrie(List<String> patterns) {
```

```
        for (String pattern : patterns) {
            Node node = root;
            for (char c : pattern.toCharArray()) {
                node = node.children.computeIfAbsent(c, k -> new Node());
            }
            node.output.add(pattern);
        }
    }

    private void buildFailureLinks() {
        Queue<Node> queue = new LinkedList<>();
        for (Node node : root.children.values()) {
            node.fail = root;
            queue.add(node);
        }

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            for (Map.Entry<Character, Node> entry :
current.children.entrySet()) {
                char c = entry.getKey();
                Node child = entry.getValue();
                Node fail = current.fail;
                while (fail != null && !fail.children.containsKey(c)) {
                    fail = fail.fail;
                }
                if (fail == null) {
                    child.fail = root;
                } else {
                    child.fail = fail.children.get(c);
                    child.output.addAll(child.fail.output);
                }
                queue.add(child);
            }
        }
    }

    public boolean search(String target) {
        Node node = root;
        for (char c : target.toCharArray()) {
            while (node != null && !node.children.containsKey(c)) {
                node = node.fail;
            }
            if (node == null) {
                node = root;
            } else {
                node = node.children.get(c);
                if (!node.output.isEmpty()) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

```
static public boolean acAutomatonMatch(List<String> patterns, String target)
{
    ACAutomaton acAutomaton = new ACAutomaton(patterns);
    return acAutomaton.search(target);
}
```

AC自动机

- **高效**：能够快速匹配多个模式。
- **实时匹配**：一旦构建完成，可以实时检测输入流中的模式匹配。

```
    public static int kmpMatch(String pattern, String target) {
        if(pattern.isEmpty()) {
            return 0;
        }
        int[] lps = buildLPSArray(pattern);
        int i = 0, j = 0;
        int targetLen = target.length();
        int patternLen = pattern.length();

        while (i < targetLen) {
            if (pattern.charAt(j) == target.charAt(i)) {
                i++;
                j++;
            }
            if (j == patternLen) {
                return i - j; // 找到匹配
            } else if (i < targetLen && pattern.charAt(j) !=
target.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
        return -1; // 未找到匹配
    }

    /**
     * 构建部分匹配表（LPS数组）
     *
     * @param pattern 模式串
     * @return 部分匹配表（LPS数组）
     */
    private static int[] buildLPSArray(String pattern) {
        int patternLen = pattern.length();
        int[] lps = new int[patternLen];
        int length = 0;
        int i = 1;
        lps[0] = 0;

        while (i < patternLen) {
            if (pattern.charAt(i) == pattern.charAt(length)) {
```

```
                length++;
                lps[i] = length;
                i++;
            } else {
                if (length != 0) {
                    length = lps[length - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }
```

地图标注工具

在本次课程设计中，除了完成原定的Web后端开发任务，我开发了一个地图数据标注工具。该工具采用 Python 语言编写，使用 Tkinter 库进行图形用户界面 (GUI) 的构建，免费提供给同学们使用。该工具的主要功能包括加载地图图片、标记地图上的点和边、保存和加载标注数据、计算最短路径等。下面详细介绍该工具的各个功能模块和实现细节。

工具功能概述

1. **加载地图图片**：用户可以加载本地图片作为地图背景。
2. **标记点**：用户可以在地图上标记点，并为每个点命名。
3. **选择点并标记边**：用户可以选择两个点，并在它们之间画一条边，边的权重为两点间的欧几里得距离。
4. **删除边**：用户可以删除已标记的边。
5. **保存和加载标注数据**：用户可以将标注的数据保存为 JSON 文件，并可以从 JSON 文件加载标注数据。
6. **计算最短路径**：工具支持计算并展示各个点之间的最短路径距离矩阵。

主要模块及实现细节

1. 初始化和基本设置

代码示例：

```
class GraphApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Graph Marker")
        self.canvas = Canvas(root)
        self.canvas.pack(expand=tk.YES, fill=tk.BOTH)
        self.image = None
        self.points = []
        self.edges = []
        self.adjList = {}
        self.selected_points = []
        self.graph_file_path = None

```

```
        self.load_image_button = tk.Button(root, text="Load Image",
command=self.load_image)
        self.load_image_button.pack()
        self.load_graph_button = tk.Button(root, text="Load Graph",
command=self.load_graph)
        self.load_graph_button.pack()
        self.save_button = tk.Button(root, text="Save JSON",
command=self.save_json)
        self.save_button.pack()

        self.canvas.bind("<Button-1>", self.mark_point)
        self.canvas.bind("<Button-3>", self.select_point)
        self.canvas.bind("<Button-2>", self.delete_edge)

        atexit.register(self.on_exit)
```

说明:

- 初始化 Tkinter 主窗口和画布 (Canvas)。
- 定义用于存储点、边及邻接表的数据结构。
- 创建并配置按钮，用于加载图片、加载图数据、保存标注数据。
- 绑定鼠标事件，用于标记点、选择点和删除边。
- 注册程序退出时的回调函数。

2. 加载地图图片

代码示例:

```
def load_image(self):
    file_path = filedialog.askopenfilename()
    if file_path:
        self.image = Image.open(file_path)
        self.image_width, self.image_height = self.image.size
        self.image_tk = ImageTk.PhotoImage(self.image)
        self.canvas.config(width=self.image_width, height=self.image_height)
        self.canvas.create_image(0, 0, anchor=tk.NW, image=self.image_tk)
```

说明:

- 通过文件对话框选择图片文件。
- 使用 PIL 库加载图片，并将其转换为 Tkinter 可用的格式。
- 在画布上显示加载的图片。

3. 标记点

代码示例:

```
def mark_point(self, event):
    x, y = event.x, event.y
    point_name = simpledialog.askstring("Input", f"Enter name for the point at ({x}, {y}):")
    if not point_name:
        return

    self.canvas.create_oval(x - 3, y - 3, x + 3, y + 3, fill='red')
    self.canvas.create_text(x, y - 10, text=point_name, fill='red')
    point = {"name": point_name, "index": len(self.points), "x": int(x), "y": int(y)}
    self.points.append(point)
    self.adjList[self.format_point(point)] = []
```

说明:

- 捕捉鼠标点击事件的坐标 (x, y)。
- 弹出对话框让用户为该点命名。
- 在画布上绘制该点和其名称。
- 将点的信息存储在 `points` 列表和 `adjList` 邻接表中。

4. 选择点并标记边

代码示例:

```
def select_point(self, event):
    x, y = event.x, event.y
    closest_point = min(self.points, key=lambda p: (p['x'] - x) ** 2 + (p['y'] - y) ** 2)
    self.selected_points.append(closest_point)
    if len(self.selected_points) == 2:
        self.mark_edge()
        self.selected_points = []
```

说明:

- 捕捉鼠标右键点击事件的坐标 (x, y)。
- 找到距离点击位置最近的点，并将其添加到 `selected_points` 列表中。
- 如果已经选择了两个点，则调用 `mark_edge` 方法标记边，并清空 `selected_points` 列表。

代码示例:

```
def mark_edge(self):
    if len(self.selected_points) < 2:
        return

    p1, p2 = self.selected_points
    line = self.canvas.create_line(p1['x'], p1['y'], p2['x'], p2['y'], fill='blue')
    self.edges.append((p1, p2, line))

    weight = int(math.sqrt((p1['x'] - p2['x']) ** 2 + (p1['y'] - p2['y']) ** 2))
```

```
    self.adjList[self.format_point(p1)].append({
        "destination": {
            "index": p2['index'],
            "x": p2['x'],
            "y": p2['y'],
            "name": p2['name']
        },
        "weight": weight
    })
    self.adjList[self.format_point(p2)].append({
        "destination": {
            "index": p1['index'],
            "x": p1['x'],
            "y": p1['y'],
            "name": p1['name']
        },
        "weight": weight
    })
```

说明：

- 在两个点之间画一条蓝色的边，并将边的信息添加到 `edges` 列表中。
- 计算两点之间的欧几里得距离，作为边的权重。
- 更新邻接表 `adjList`，将边的信息添加到两个点的邻接列表中。

5. 删除边

代码示例：

```
def delete_edge(self, event):
    x, y = event.x, event.y
    for p1, p2, line in self.edges:
        x1, y1 = p1['x'], p1['y']
        x2, y2 = p2['x'], p2['y']
        if self.is_near_line(x, y, x1, y1, x2, y2):
            self.canvas.delete(line)
            self.edges.remove((p1, p2, line))
            self.adjList[self.format_point(p1)] = [e for e in
self.adjList[self.format_point(p1)] if
                                                   e['destination']['index']
!= p2['index']]
            self.adjList[self.format_point(p2)] = [e for e in
self.adjList[self.format_point(p2)] if
                                                   e['destination']['index']
!= p1['index']]
            break
```

说明：

- 捕捉鼠标中键点击事件的坐标 (x, y)。
- 检查点击位置是否在某条边的附近，如果是则删除该边。
- 更新画布和邻接表，移除该边的信息。

6. 保存和加载标注数据

代码示例:

```
def save_json(self):
    graph = {
        "name": "graph",
        "adjList": self.adjList,
        "shortestDistances": self.calculate_shortest_distances()
    }
    file_path = self.graph_file_path if self.graph_file_path else "graph.json"
    with open(file_path, "w") as f:
        json.dump(graph, f, indent=4)
```

说明:

- 将图的邻接表和最短路径距离矩阵保存为 JSON 文件。

代码示例:

```
def load_graph(self):
    file_path = filedialog.askopenfilename(filetypes=[("JSON files",
"*.json")])
    if file_path:
        self.graph_file_path = file_path
        with open(file_path, "r") as f:
            graph = json.load(f)
        self.adjList = graph["adjList"]
        self.points = []
        self.edges = []
        for point_str in self.adjList.keys():
            point = self.parse_point(point_str)
            self.points.append(point)
            self.canvas.create_oval(point['x'] - 3, point['y'] - 3,
point['x'] +

 3, point['y'] + 3, fill='red')
            self.canvas.create_text(point['x'], point['y'] - 10,
text=point['name'], fill='red')
        for point_str, edges in self.adjList.items():
            src_point = self.parse_point(point_str)
            for edge in edges:
                dest_point = self.points[edge["destination"]["index"]]
                line = self.canvas.create_line(src_point['x'],
src_point['y'], dest_point['x'], dest_point['y'],
                                               fill='blue')
                self.edges.append((src_point, dest_point, line))
```

说明:

- 从 JSON 文件加载图的邻接表，并在画布上重现标注的点和边。

7. 计算最短路径

代码示例:

```
def calculate_shortest_distances(self):
    n = len(self.points)
    distances = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        distances[i][i] = 0

    for src, edges in self.adjList.items():
        src_point = self.parse_point(src)
        src_index = src_point['index']
        for edge in edges:
            dest_index = edge['destination']['index']
            weight = edge['weight']
            distances[src_index][dest_index] = weight

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if distances[i][j] > distances[i][k] + distances[k][j]:
                    distances[i][j] = int(distances[i][k] + distances[k][j])

    return distances
```

说明:

- 使用 Floyd-Warshall 算法计算并返回图中各点之间的最短路径距离矩阵。

8. 程序退出时的回调函数

代码示例:

```
def on_exit(self):
    print(f"Total vertices: {len(self.points)}")
    print(f"Total edges: {len(self.edges)}")
```

说明:

- 程序退出时打印总顶点数和总边数。

总结

本地图数据标注工具提供了一套完整的功能，方便用户在地图上标记和管理点与边。通过使用 Tkinter 库，我们实现了友好的图形用户界面，使用户可以直观地进行操作。工具不仅支持基本的点和边的标注，还支持标注数据的保存和加载，并提供最短路径计算功能，极大地方便了地图数据的处理和分析。希望该工具能为同学们的课程设计和项目开发提供有力支持。