智能旅行规划助手-OceanGuide

项目概述

背景介绍

近年来，随着旅游业的发展，越来越多的人选择通过旅游来放松心情、体验不同的文化和风景。然而，传统的旅行规划应用在提供个性化路线推荐时存在一些局限性，例如推荐路线单一、操作繁琐、智能化程度不足等问题，无法很好地满足用户的多样化需求。为了解决这些问题，我们开发了一个基于地理大模型的智能旅行规划助手——OceanGuide。

目标与解决的问题

OceanGuide旨在利用先进的AI技术和强大的数据库支持，为用户提供直观、个性化、多样化的旅行路线规划服务。我们的目标是创建一个能够根据用户的偏好和需求，结合地理信息，生成满足用户个性化需求的旅行路线的应用系统。通过使用OceanBase数据库，我们不仅确保了数据的安全性和高效处理能力，还利用其MySQL兼容模式简化了数据库迁移过程，并提升了系统的可扩展性和稳定性。

解决方案的独特优势

OceanGuide采用了一系列创新性的技术手段，包括对话式交互设计、GIS深度融合以及基于大语言模型（LLM）的智能路线规划等。特别是OceanBase数据库的引入，使得我们能够在保证高性能的同时，实现对大规模数据的有效管理。OceanBase作为一款分布式关系型数据库，提供了高可用性、强一致性及水平扩展能力，特别适合处理海量用户信息和复杂的POI数据集。此外，它支持MySQL协议，这极大地降低了从现有MySQL数据库迁移至OceanBase的成本和技术门槛，同时也增强了系统的灵活性和适应性。

技术方案

技术架构

OceanGuide的技术架构主要由前端界面、后端服务、数据库存储及外部API调用四大部分组成。前端界面负责展示地图、接收用户输入及显示结果；后端服务则负责处理业务逻辑、执行算法计算并返回结果给前端；数据库用于存储用户信息、地点信息及历史对话记录；而外部API主要用于获取实时地理位置信息及其他必要的第三方服务。

使用的技术栈

- **前端**：Vue.js + Vite 构建快速响应的Web应用。
- **后端**：SpringBoot + FastAPI 实现高效的RESTful API接口。
- **数据库**：OceanBase MySQL 兼容模式，用于存储和管理所有结构化数据。
- **其他工具**：Node.js, npm, KQGIS SDK, Python 等用于构建和部署整个应用生态系统。

数据处理流程及算法模型

1. **用户注册与登录**：用户通过前端页面提交注册或登录请求，后端服务验证用户名密码并通过OceanBase查询用户是否存在。
2. **兴趣偏好收集**：新用户首次登录时需填写个人兴趣偏好，这些信息将被安全地保存到OceanBase中。
3. **旅行需求输入**：用户以自然语言形式输入旅行需求，系统使用大语言模型分析并提取关键信息。

4. **个性化路线规划**：结合用户偏好、地理知识以及大语言模型生成的prompt，系统自动生成个性化旅行路线。
5. **地图可视化展示**：通过KQGIS SDK将生成的路线动态展示在地图上，供用户查看和调整。

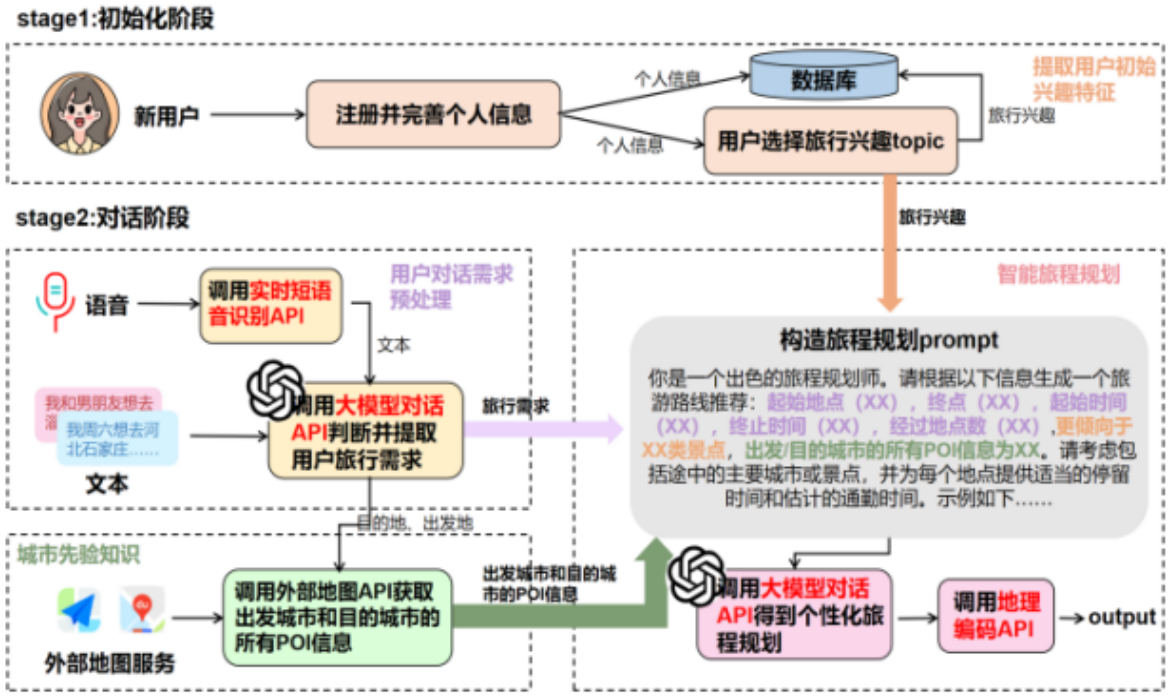
OceanBase的优点

OceanBase是一款高性能的分布式关系型数据库，具有以下优点：

- **高可用性**：支持多副本同步复制，确保数据的可靠性和持续可用性。
- **线性扩展**：可以轻松应对数据量增长的需求，无需停机维护即可增加节点数量。
- **MySQL兼容性**：无缝支持MySQL协议，降低迁移成本，提高开发效率。
- **强一致性**：即使在网络分区的情况下也能保持事务的一致性。

实施细节

系统设计图



关键代码片段

```java
@RestController
@RequestMapping("/ai")
public class SparkApi {
    @Autowired
    private ApiService apiService;
    @Autowired
    private BigModelService bigModelService;
    @Autowired
    private BigModelStreamService bigModelStreamService;

    @PostMapping("/ask")
    public Result askQuestion(SingleRecord singleRecord) {
        try {
            String question= singleRecord.getInput();
```

```
            String res = bigModelService.askQuestion(question,getMessages(singleRecord.getDialogId(), question));
            singleRecord.setOutput(res);
            apiService.insertOnce(singleRecord);
            return Result.success(singleRecord);
        } catch (Exception e) {
            return Result.error(e.getMessage());
        }
    }

    @PostMapping("/askStream")
    public SseEmitter askQuestionStream(SingleRecord singleRecord) {
        SseEmitter emitter = new SseEmitter();
        try {
            String question = singleRecord.getInput();

 bigModelStreamService.askQuestionStream(singleRecord,getMessages(singleRecord.getDialogId(),question), emitter);

        } catch (Exception e) {
            try {
                emitter.send(SseEmitter.event()
                        .name("error")
                        .data(e.getMessage()));
            } catch (IOException ex) {
                ex.printStackTrace();
            } finally {
                emitter.complete();
            }
        }
        return emitter;
    }

    public List<RoleContent> getMessages(Integer dialogId, String input){
        System.out.println(dialogId);
        List<RoleContent> historyList = new ArrayList<>();
        ArrayList<String> inputs=apiService.getInputsByDialogId(dialogId);
        ArrayList<String> outputs=apiService.getOutputsByDialogId(dialogId);
        List<Map<String, String>> messages = new ArrayList<>();

        for (int i = 0; i < inputs.size(); i++) {
            RoleContent roleContent1 = new RoleContent();
            roleContent1.setRole("user");
            roleContent1.setContent(inputs.get(i));
            historyList.add(roleContent1);

            RoleContent roleContent2 = new RoleContent();
            roleContent2.setRole("assistant");
            roleContent2.setContent(outputs.get(i));
            historyList.add(roleContent2);
        }
        RoleContent roleContent = new RoleContent();
        roleContent.setRole("user");
        roleContent.setContent(input);
        historyList.add(roleContent);
        return historyList;
    }
```

```
}

```

核心算法说明

大语言模型引导的智能路线规划算法

该算法首先对用户输入的需求进行预处理，提取出发地、目的地等关键信息。然后调用外部地图API获取相关POI信息，将其作为额外输入传递给大语言模型。最后，基于用户偏好和地理知识生成个性化的旅行路线。

```python
def getFinalPOI(startPOI, endPOI, startTime, endTime, numPOI, userId):
    """
    :param startPOI: 用户需求五元组-起点
    :param endPOI: 用户需求五元组-终点
    :param startTime: 用户需求五元组-起始时间
    :param endTime: 用户需求五元组-终止时间
    :param numPOI: 用户需求五元组-途径点个数
    :param userId: 用户ID
    :return: 返回给用户推荐的 numPOI 个 POI 信息
    """
    # 调用用户偏好获取接口获取用户的旅行偏好
    url = f"http://backend:7778/preferences?id={userId}"
    payload = {}
    headers = {"User-Agent": "Apifox/1.0.0 (https://apifox.com)"}
    response_pre = requests.request("GET", url, headers=headers,
data=payload)
    # 检查请求是否成功
    if response_pre.status_code == 200:
        response_pre_json = response_pre.json()
        # 获取用户的旅行偏好
        interested_places = response_pre_json.get("data",
{}).get("interestedPlaces")
        interested_ways = response_pre_json.get("data",
{}).get("interestedWays")
        travel_companion = response_pre_json.get("data",
{}).get("travelCompanion")
    else:
        print(f"Request failed with status code:
{response_pre.status_code}")
        # 如果请求失败，可以设置默认值或终止程序
        interested_places = None
        interested_ways = None
        travel_companion = None

    # 如果用户偏好为空，可以设置默认值
    if not interested_places:
        interested_places = "自然风光，美食"
    if not interested_ways:
        interested_ways = "常规路线"
    if not travel_companion:
        travel_companion = "恋人"

    # 调用 POI 初步筛选接口获得初步 POI 候选集
    start_name = quote(startPOI)
    end_name = quote(endPOI)
```

```
    poi_filter_url = f"http://backend:7778/position/filter?begin={start_name}&end={end_name}&num=23"
    payload = {}
    headers.update(
        {"Accept": "*/*", "Host": "backend:7778", "Connection": "keep-alive"}
    )
    response_poi = requests.request(
        "GET", poi_filter_url, headers=headers, data=payload
    )
    selected_poi = response_poi.json().get("data", {})
    # print(selected_poi)

    # 构建 prompt，根据用户需求五元组、用户偏好以及候选 POI 得到最终推荐给用户的 POI
    preference = f"{{感兴趣的旅行地点：{interested_places}；感兴趣的旅行方式：{interested_ways}；旅行伙伴：{travel_companion}}};"
    input_info = (
        f"1.出发地点：{startPOI}；2.出发时间：{startTime}；3.终止地点：{endPOI}；"
        f"4.终止时间：{endTime}；5.途径点个数：{numPOI}个；6.旅行偏好：{preference}；"
        f"7.可参考的地点信息为：{selected_poi}"
    )
    prompt = f"""
    TASK：作为旅游行程规划师，假设你充分了解北京市的所有地点以及交通信息，且十分擅长于根据不同人群的需求安排合理的游玩地点和时间。请根据给定信息推荐{numPOI}个POI地点，构成一条顺畅且完整的旅行路线。
    REQUEST：规划一个旅游路线，包括POI地点序列。请根据以下信息规划路线：{input_info}
    ACTION：
    1. 确定本次行程的大概活动范围，根据起终点的位置规划路线。
    2. 根据游客的偏好找到所有可能感兴趣的景点、美食、购物以及休闲娱乐场所等。
    3. 综合考虑行程起终时间、交通时间、地点之间是否顺路等因素，规划一条合理的路线。
    4. 安排就餐地点时尽量选择距离参观景点一公里以内的地点。
    5. 你只需要返回 {numPOI} 个你认为最合适的地点即可，不要多于 {numPOI} 个也不要少于 {numPOI} 个。
    EXAMPLE：
    以下是一个推荐POI地点的示例，最终输出应为 JSON 格式文本：
    {{
        "pois":[
            {{
                "name": "[地点名称]",
                "longitude": [地点经度],
                "latitude": [地点纬度]
            }},
            ...
        ]
    }}
    """
    encoded_prompt = quote(prompt)

    # 创建对话的 API
    create_dialog_url = "http://backend:7778/dialog"
    create_payload = {}
    create_headers = headers.copy()
    create_headers.update(
        {"Accept": "*/*", "Host": "backend:7778", "Connection": "keep-alive"}
    )
```

```
90
91      # 设置参数，可以修改 name 和 userId 的值
92      params = {"userId": userId, "name": "流式输出尝试"}  # 确保 userId 一致
93
94      # 发送创建对话的请求
95      response = requests.request(
96          "POST",
97          create_dialog_url,
98          headers=create_headers,
99          params=params,
100         data=create_payload,
101     )
102     create_result = response.json()
103
104     # 获取返回的 dialogId
105     dialog_id = create_result["data"]["id"]
106     print(f"创建的对话 ID 为：{dialog_id}")
107
108     # 进行对话的 API
109     ask_url = f"http://backend:7778/spark/ask?input=
{encoded_prompt}&dialogId={dialog_id}&userId={userId}"
110
111     ask_payload = {}
112     ask_headers = headers.copy()
113
114     # 发送对话请求
115     response = requests.request("POST", ask_url, headers=ask_headers,
data=ask_payload)
116
117     # 打印响应内容
118     print("对话响应：")
119     print(response.text)
120
121     # 解析响应内容，获取 POI 信息
122     ask_result = response.json()
123     poi_content = ask_result["data"]["output"]
124
125     print(poi_content)
126     print(poi_content.strip())
127     # 删除```json
128     poi_content=poi_content.replace("```json", "")
129     poi_content=poi_content.replace("```", "")
130     print(poi_content)
131     # 将字符串转换为字典
132     try:
133         final_poi = json.loads(poi_content)
134     except json.JSONDecodeError as e:
135         print("解析 POI 内容时出错：", e)
136         final_poi = None
137
138
139     """
140     返回如下的 POI 集合(final_poi 是一个字典)：
141     {
142         "pois": [
143             {
144                 "name": "北京故宫博物院",
145                 "longitude": 116.403414,
```

```
                "latitude": 39.924091
            },
            ...
        ]
    }
    """
    return final_poi
```

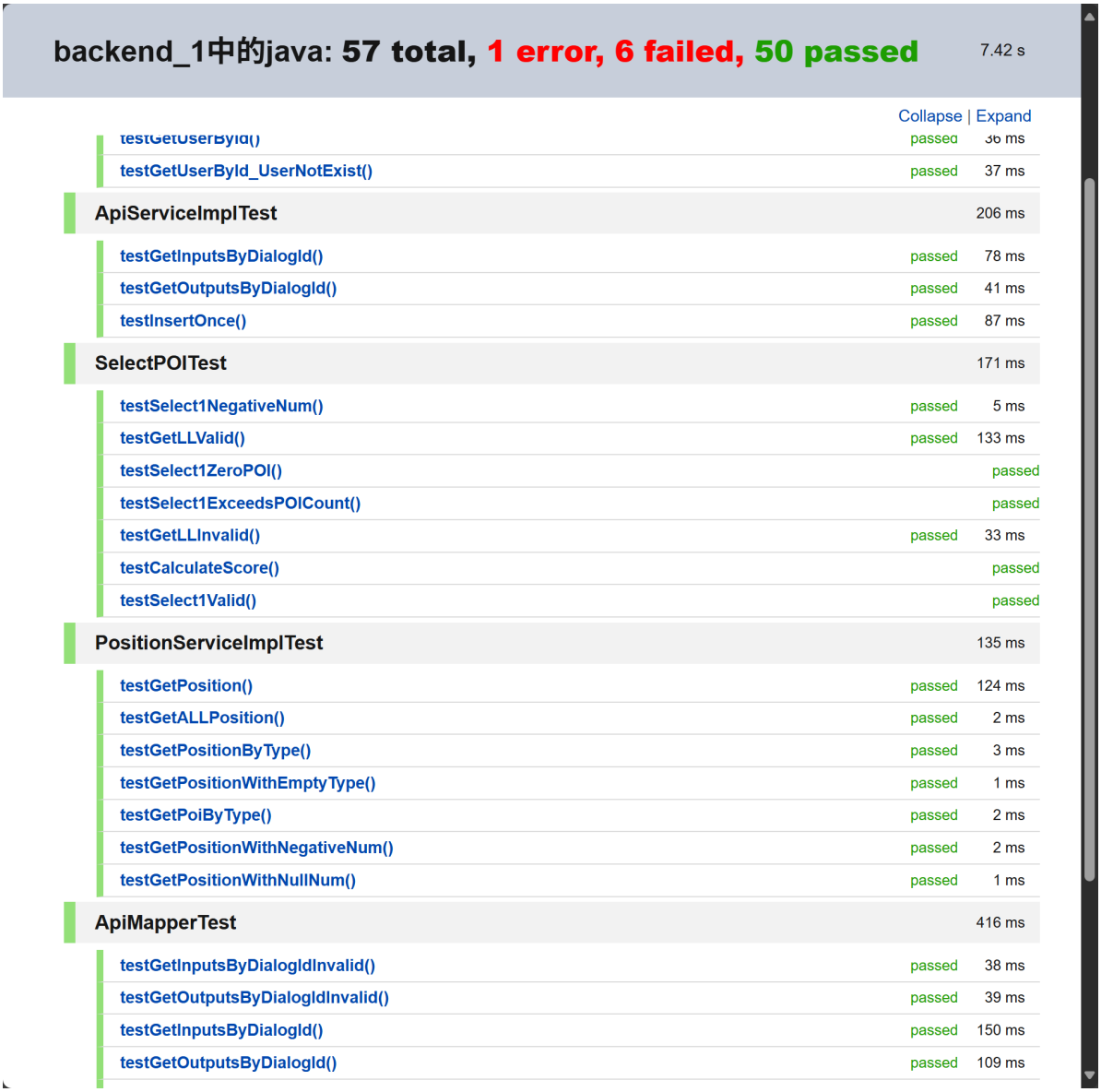
测试与验证

测试环境设置

为了确保系统的稳定性和可靠性，我们在本地搭建了一套完整的测试环境，包括模拟OceanBase集群、前后端服务器以及各种依赖服务。

测试用例

1. 用户注册与登录测试
2. 个人信息更新测试
3. 旅行路线规划测试
4. 地图可视化功能测试



测试结果

经过多次测试，OceanGuide系统表现良好，所有功能均能正常运行，特别是在并发访问情况下，OceanBase展现了出色的性能优势，保证了数据处理的高效性和准确性。