

CS307 Project2

小组成员：陈明志 12211414，邱天润 12210829

源码托管于 GitHub，将在项目 ddl 结束后基于 **MIT License** 协议开源，访问链接：

<https://github.com/RoderickQiu/CS307-Project2>

成员分工及贡献百分比

陈明志：

- 构建Cards和Passengers相关API（基础要求5，6，7）。
- 关于API的扩展要求（站点状况，商务车厢，多参数查询）。
- 处理Price.xlsx，准备测试数据。
- 基于Flask的对后端的封装，RESTful API、连接池、ORM映射的实现。
- 通过Sqlalchemy实现触发器以及Postgres用户权限的配置。
- 项目报告撰写。

邱天润：

- 构建Lines和Stations相关API（基础要求1，2，3，4）。
- 使用Vue等工具构建一个现代化的前端界面进行数据展示和测试，满足好用、优雅的扩展要求。
- MySQL数据库测试。
- 基于Flask的对后端的封装，RESTful API、连接池、ORM映射的实现，以及前后端的包管理。
- 项目报告撰写。

贡献百分比相同，均为 50%。

项目源码文件夹结构：

```
1 | └─ backend
2 |   └─ __init__.py
3 |   └─ config.py
4 |   └─ controllers.py
5 |   └─ models.py
6 |   └─ urls.py
7 | └─ frontend
8 |   └─ index.html
9 |   └─ jsconfig.json
10 |   └─ package.json
11 |   └─ postcss.config.js
12 |   └─ src
13 |     └─ App.vue
14 |     └─ assets
15 |       └─ base.css
16 |       └─ layout.css
17 |       └─ main.css
18 |     └─ components
19 |     └─ Cards.vue
```

```
20 | | | | | | Lines.vue
21 | | | | | | Stations.vue
22 | | | | | | golden
23 | | | | | | | GlTemplate.vue
24 | | | | | | | GoldenLayout.vue
25 | | | | | | | SlotExtr.vue
26 | | | | | | | index.js
27 | | | | | | |
28 | | | | | | | stores
29 | | | | | | | util.js
30 | | | | | | |
31 | | | | | | |
32 | | | | | | |
33 | | | | | | |
34 | | | | | | |
```

主要文件介绍：

- [app.py](#): 包含应用程序的主要运行逻辑
- [config.py](#): 包含应用程序的配置信息，如数据库连接字符串、密钥等
- [controllers.py](#): 包含处理请求和响应的控制器函数
- [models.py](#): 定义应用程序的数据模型，与数据库表格对应
- [urls.py](#): 定义应用程序的URL路由访问规则
- [Process.py](#): 处理或转换数据的脚本，将 `票价.xlsx` 转换为csv文件后再转换为可直接用的数据
- [frontend/*](#): 基于Vue，使用现代技术和前后端分离的思想，实现的一个优雅、易用前端界面，用于进行数据展示和测试

使用指南：

本地配置

后端配置

1. 按照 `requirement.txt` 中的版本安装 `Flask` `Flask_Migrate` `flask_sqlalchemy` `SQLAlchemy` 等PyPi包。
2. `cd backend` 切换到backend目录下后运行 `flask run` 即可开启后端服务器。

前端配置

1. `cd frontend` 切换到frontend目录下后，使用 `yarn` 或 `npm install` 命令安装所需NPM包。
2. 输入 `yarn dev` 或 `npm run dev`，运行Vue服务，进行测试。

基础部分（默认本地路径：<http://127.0.0.1:5000>）

1. 线路操作

获取所有线路或创建线路

- 请求路径: `/lines`
- 请求方法: `GET`, `POST`
- 描述: 获取所有线路的列表或创建新的线路。对于POST方法, 需要在Body字段中按照字典的格式添加每一个字段所对应的信息。(`line_name` `business_carriage` `start_time` `end_time` `intro` `mileage` `color` `first_opening` `url`)

获取、更新或删除指定线路

- 请求路径: `/lines/<line_id>`
- 请求方法: `GET`, `PUT`, `DELETE`
- 描述: 获取、更新或删除指定ID的线路。对于PUT方法, 需要在Body字段中按照字典的格式添加每一个字段更新后所对应的信息。(`line_name` `business_carriage` `start_time` `end_time` `intro` `mileage` `color` `first_opening` `url`)

2. 车站操作

获取所有车站或创建车站

- 请求路径: `/stations`
- 请求方法: `GET`, `POST`
- 描述: 获取所有车站的列表或创建新的车站。
 - 对于POST方法, 需要在Body字段中按照字典的格式添加每一个字段所对应的信息, Status字段包括三个状态 `opening` `closed` `under`。(`English_name` `Chinese_name` `District` `Status` `Introduction`)
 - 对于GET方法, 我们进行分页, 在GET参数中添加 `page` 和 `elem_per_page` 字段, 表示当前的页数和每页长度; 返回值为这样的形式:

```
1 {
2   "page": "1",
3   "total": "2000",
4   "result": { RESPONSE }
5 }
```

获取、更新或删除指定车站

- 请求路径: `/stations/<station_id>`
- 请求方法: `GET`, `PUT`, `DELETE`
- 描述: 获取、更新或删除指定ID的车站。对于PUT方法, 需要在Body字段中按照字典的格式添加每一个字段更新后所对应的信息。 , Status字段包括三个状态 `opening` `closed` `under`。(`English_name` `Chinese_name` `District` `Status` `Introduction`)

3-4. 线路和车站操作

获取线路上的所有车站

- 请求路径: `/lines/<line_id>/stations`
- 请求方法: `GET`
- 描述: 获取指定线路上的所有车站。

获取、添加或删除线路上的指定车站

- 请求路径: `/lines/<line_id>/stations/<station_id>`
- 请求方法: `GET`, `POST`, `DELETE`
- 描述:
 - 获取、添加或删除线路上的指定车站。在POST方法中, 需要在Body字段中按照字典的格式添加每一个字段所对应的信息(`line_num`)。
 - 另外, 对于POST方法, 我们可以指定`station_id`为一个数组, 从而可以一次放入多个车站, 使用格式如 `stations/[1,2,3]`, 表示在在`line_num`位置先后添加`station_id`为1, 2, 3的三个车站。若只导入单个车站, 使用 `stations/1` 格式即可。

获取线路上指定车站的前后n个车站

- 请求路径: `/lines/<line_id>/stations/<station_id>/n/<n>`
- 请求方法: `GET`
- 描述: 获取线路上指定车站的前后n个车站。

注意: 在以上的路径中, `<line_id>`、`<station_id>` 和 `<n>` 需要替换为实际的线路ID、车站ID和车站数量。

5-6. 乘客和公交卡操作

获取所有卡行程或创建卡行程

- 请求路径: `/card_rides`
- 请求方法: `GET`, `POST`
- 描述: 获取所有卡行程的列表或创建新的卡行程。对于POST方法(上车), 需要在Body字段中按照字典的格式添加每一个字段所对应的信息。(`card_id` `from_station` `start_time` `business_carriage`)

获取、更新或删除指定卡行程

- 请求路径: `/card_rides/<ride_id>`
- 请求方法: `GET`, `PUT`, `DELETE`
- 描述: 获取、更新或删除指定ID的卡行程。对于PUT方法(下车), 需要在Body字段中按照字典的格式添加每一个字段更新后所对应的信息。(`to_station` `end_time`)

获取所有用户行程或创建用户行程

- 请求路径: `/user_rides`
- 请求方法: `GET`, `POST`
- 描述: 获取所有用户行程的列表或创建新的用户行程。对于POST方法(上车), 需要在Body字段中按照字典的格式添加每一个字段所对应的信息。(`user_id` `from_station` `start_time` `business_carriage`)

获取、更新或删除指定用户行程

- 请求路径: `/user_rides/<ride_id>`
- 请求方法: `GET`, `PUT`, `DELETE`
- 描述: 获取、更新或删除指定ID的用户行程。对于PUT方法(下车), 需要在Body字段中按照字典的格式添加每一个字段更新后所对应的信息。(`to_station` `end_time`)
- 注意: 在以上的路径中, `<ride_id>` 需要替换为实际的行程ID。

7. 在线人数操作

获取在线人数（即已经上车还未出站的人数）

- 请求路径： `/online`
- 请求方法： `GET`
- 描述： 获取当前地铁线上还未下车的人数以及其具体信息。

进阶部分

MySQL数据库配置

在生成Postgres数据库DDL的同时，我们学习MySQL的DDL语法，并同步制作了一个MySQL的数据库副本。

由于我们使用了SQLAlchemy库来通过ORM方式访问数据库，我们使用了pymysql的连接来替代用于Postgres的psycopg2连接器，便捷地实现了MySQL数据库的支持。

具体使用时，仅需要在 `/backend/.env` 文件中，修改 `DEVELOPMENT_DATABASE_URL` 为 `mysql+pymysql://<username>:<password>@<host>:<port>/<database>`，即可实现MySQL切换，实现一套代码、两个数据库系统皆可用。

更多API功能

我们另多实现了几种API设计，完成了更多的系统功能需求。

另外，由于部分数据量较大，为了提升访问效率，我们采用了分页输出的形式，较好地优化了性能，具体详见以下描述。

1. **地铁站状态**：增加并合理使用地铁站状态，例如：建设中、运营中、关闭中等。
 - 在 `/stations` 中增加 `status` 字段，表示地铁站状态。
2. **商务车厢信息**：增加商务车厢的信息，如乘坐商务车厢，价格翻倍。
 - 在 `/lines` 中增加 `business_carriage` 字段，表示是否有商务车厢。
 - 在 `/stations` 中增加 `business_carriage` 字段，表示是否有商务车厢。
 - 在 `/card_rides` 中增加 `business_carriage` 字段，表示是否乘坐商务车厢，如若乘坐商务车厢，价格翻倍。
3. **多参数搜索乘车记录功能**：通过地铁站、乘车人、时间段等实现1~n 参数搜索乘车记录功能。具体使用方法如下：
 - `/queryuser`：查询用户乘车记录。你可以在 POST 请求的表单数据中包含以下参数：
 - `business_carriage`：商务车厢
 - `from_station`：起始站
 - `to_station`：终点站
 - `user_id`：用户 ID
 - `on_the_ride`：是否在乘车
 - `price`：价格
 - `time`：时间（格式为 "YYYY-MM-DDTHH:MM:SS"）
 - `/querycard`：查询卡片乘车记录。你可以在 POST 请求的表单数据中包含以下参数：
 - `business_carriage`：商务车厢
 - `from_station`：起始站

- `to_station`: 终点站
- `card_id`: 卡片 ID
- `on_the_ride`: 是否在乘车
- `price`: 价格
- `time`: 时间 (格式为 "YYYY-MM-DDTHH:MM:SS")

4. 获取指定用户的所有行程: 获取指定用户的所有行程。

- 请求路径: `/user_rides/user/<user_id>`
- 请求方法: `GET`

5. 获取所有用户或创建用户: 获取所有用户的列表或创建新的用户。

- 请求路径: `/users`
- 请求方法: `GET, POST`
- 描述: 对于POST方法, 需要在Body字段中按照字典的格式添加每一个字段所对应的信息。
(`user_id_number` `name` `phone` `gender` `district`)
- 对于GET方法, 我们进行分页, 在GET参数中添加 `page` 和 `elem_per_page` 字段, 表示当前的页数和每页长度; 返回值为这样的形式:

```
1 {
2   "page": "1",
3   "total": "2000",
4   "result": { RESPONSE }
5 }
```

6. 获取所有卡或创建卡: 获取所有卡的列表或创建新的卡。

- 请求路径: `/cards`
- 请求方法: `GET, POST`
- 描述: 对于POST方法, 需要在Body字段中按照字典的格式添加每一个字段所对应的信息(`card_number` `money` `create_time`)
- 对于GET方法, 我们进行分页, 在GET参数中添加 `page` 和 `elem_per_page` 字段, 表示当前的页数和每页长度; 返回值为这样的形式:

```
1 {
2   "page": "1",
3   "total": "2000",
4   "result": { RESPONSE }
5 }
```

7. 获取指定卡的所有行程: 获取指定卡的所有行程。

- 请求路径: `/card_rides/card/<card_id>`
- 请求方法: `GET`

封装并实现一个真正的后端服务器

1. ORM映射：

- 使用SQLAlchemy实现ORM映射，将数据库表格映射为Python类，实现对数据库的操作；这也便利了Postgres和MySQL的自如切换。
- 在 `/backend/models.py` 中定义了 `Line`、`Station`、`User`、`Card`、`CardRide`、`UserRide` 等类，分别对应数据库中的各个表格。
- 在 `/backend/controllers.py` 中实现了对数据库的增删改查操作。

2. 连接池：

- 使用SQLAlchemy实现连接池，提高数据库的访问效率。
- 在 `/backend/config.py` 中配置了数据库连接字符串，实现了连接池。

3. Flask后端框架封装：

- 使用Flask框架实现后端服务器，实现了对请求的响应。
- 在 `/backend/app.py` 中实现了Flask应用程序的主要运行逻辑。

4. 代码包管理：

1. 后端层面

- 使用Python的包管理工具，将代码封装为包，方便管理。
- 在 `/backend/__init__.py` 中实现了包的初始化。

2. 前端层面

- 使用NPM/Yarn进行包管理，较好地实现了代码复用。

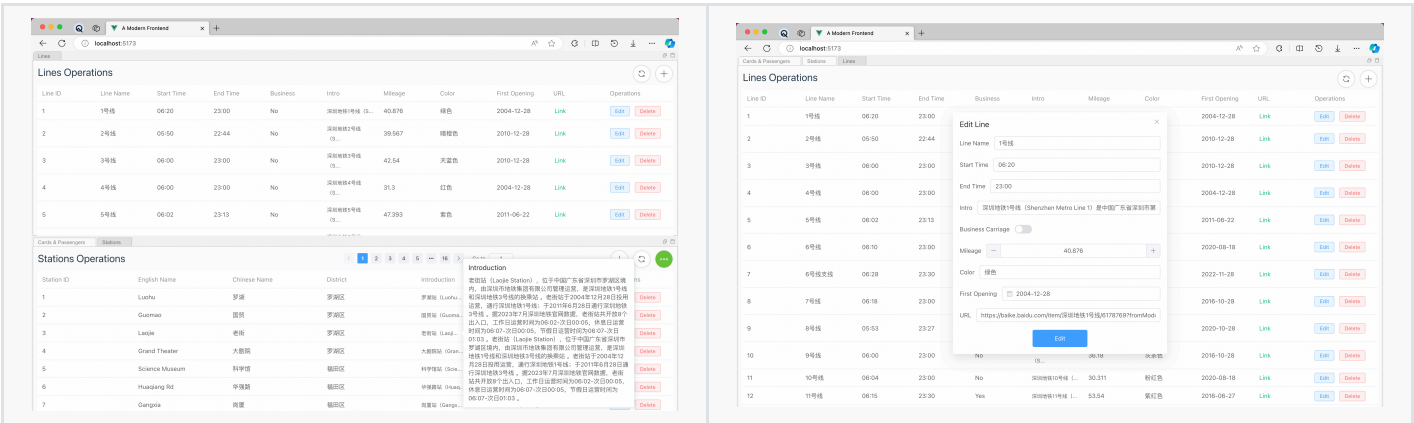
5. 套接字编程和RESTFul API支持：

- 使用Flask框架实现了套接字编程，实现了对请求的响应。
- 通信范式按照 RESTFul API 规范进行设计，确保了设计的通用性、规范性和可扩展性。
 - 通过 `PUT`、`GET`、`POST`、`DELETE` 的指令和FormData参数传送，实现了API的良好实现。
- 在 `/backend/app.py` 中实现了Flask应用程序的主要运行逻辑。

页面显示设计

我们实现了一个优雅、实用性强、代码规范的网页界面，基于Vue开发。

页面整体效果如下：



我们总结出该前端界面的以下特点，并因此认为这是一个具有良好效果的数据库应用系统管理界面：

- 基于包管理的现代开发方式，使用了Vite、Vue、Tailwind、ElementPlus、Axios、DayJS等流行的NPM库，使得代码可读性强、实现效果流畅。
- 支持后端API的所有功能，可以很好的进行展示和测试。

3. 是一套完整、自洽的系统，使用流畅，包含各种主要数据结构的列表展示和操作，并考虑到了多处的性能优化，使得只需要使用这套系统，就可以快速完整执行所有操作。
4. 美观优雅、风格统一，自定义程度高，整体使用ElementPlus统一设计风格，并通过GoldenLayout实现多个界面的自由组织。

合理使用数据库用户权限以及触发器

1. 用户权限：

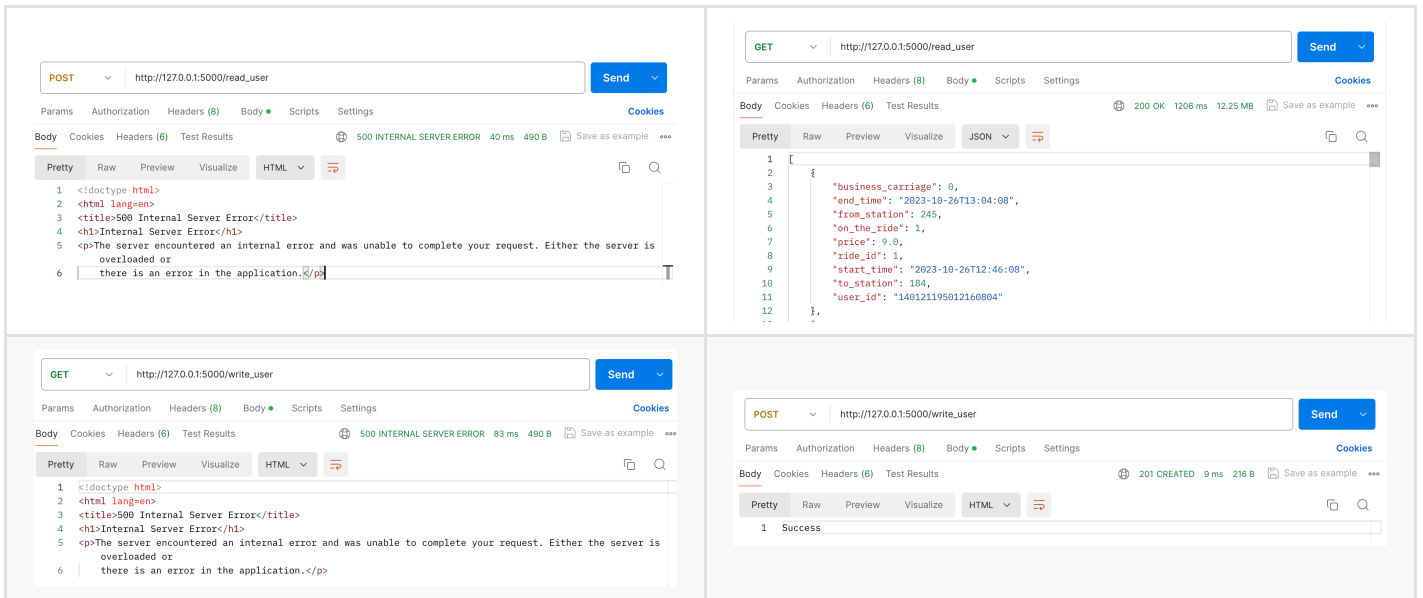
- 使用Postgres实现用户权限的配置，限制用户对数据库的访问权限。
- 首先在Postgres中执行以下代码

```
1 CREATE USER read_user WITH PASSWORD '123456';
2 GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_user;
3 CREATE USER write_user WITH PASSWORD '123456';
4 GRANT ALL PRIVILEGES ON DATABASE project1 TO write_user;
5 GRANT INSERT ON lines TO write_user;
6 GRANT UPDATE ON lines TO write_user;
7 GRANT DELETE ON lines TO write_user;
```

- 在 `/backend/config.py` 中配置了数据库连接字符串，实现了用户权限的配置。
- 在 `/backend/app.py` 中利用 `SQLAlchemy` 配置了数据库连接字符串，实现了用户权限的配置。

```
1 app.read_engine =
  create_engine('postgresql://read_user:123456@localhost/project1')
2 app.write_engine =
  create_engine('postgresql://write_user:123456@localhost/project1')
```

- 在 `/backend/controllers.py` 中实现了对数据库的增删改查操作，实现了用户权限的配置。
- 我们定义了两个路由 `/read_user` 和 `/write_user`，分别用于读取和写入用户数据。
- 对于 `/read_user` 路由，我们在GET请求中调用 `read_user_read_controller()` 函数来读取用户数据，而在POST请求中调用 `read_user_write_controller()` 函数来写入用户数据。
- 对于 `/write_user` 路由，我们在GET请求中调用 `write_user_read_controller()` 函数来读取用户数据，而在POST请求中调用 `write_user_write_controller()` 函数来写入用户数据。
- 如果接收到的请求方法不是GET或POST，我们将返回"Method is Not Allowed"的错误信息和405状态码。
- 测试时，你可以使用工具（如Postman）来发送GET或POST请求，查看返回的结果是否符合预期。
- 根据测试结果来看，我们成功地实现了用户权限的配置，限制了用户对数据库的访问权限。



2. 触发器：

- 我们使用SQLAlchemy的事件监听功能来实现触发器，对数据库的操作进行自动化处理。具体来说，我们在插入新的 `Line` 或 `Station` 记录之前，设置了一些默认值。
- 对于 `Line` 模型，我们在插入新的线路记录之前，如果没有指定 `business_carriage`（商务车厢），则默认为 `0`。这是通过 `before_insert` 事件监听器和 `default_business_carriage` 静态方法实现的。代码如下：

```
1 @staticmethod
2 def default_business_carriage(mapper, connection, target):
3     if target.business_carriage is None:
4         target.business_carriage = 0
5
6 event.listen(Line, 'before_insert', Line.default_business_carriage)
```

- 对于 `Station` 模型，我们在插入新的站点记录之前，如果没有指定 `status`（站点状态），则默认为 `opening`。这是通过 `before_insert` 事件监听器和 `default_status` 静态方法实现的。代码如下：

```
1 @staticmethod
2 def default_status(mapper, connection, target):
3     if target.status is None:
4         target.status = 'opening'
5
6 event.listen(Station, 'before_insert', Station.default_status)
```

- 这种方法的优点是，我们可以在不改变数据库结构的情况下，对数据进行预处理和验证，提高了数据的一致性和完整性。