CS307 数据库原理 Project2

成员信息与分工

1. 小组成员：陈长信（12210731）、赵欣瞳（12212727）
2. 实验课：Thursday 3-4
3. 任务分配占比 **成员贡献比：陈长信（60%）赵欣瞳（40%）**

Task	Author(Name)
所有基础部分API函数的编写	陈长信、赵欣瞳
GUI交互界面的设计	陈长信
Bonus中部分高级API的编写	赵欣瞳
封装成服务器，使用ORM映射、连接池、后端框架	陈长信
代码包的管理以及应用HTTP/RESTful Web	陈长信
不同等级用户的创建以及用户权限的测试	陈长信
MySQL实验	陈长信
报告撰写	陈长信、赵欣瞳

基础API函数的实现思路

Model.py 这里用了peewee的orm 框架来把数据库中的表结构映射到python的类上，从而实现通过操作这些类来间接操作数据库表。Line 类对应 lines 表，Station 类对应 station 表，LineStation 类对应 line_stations 表，BusInfo 类对应 bus_info 表，OutInfo 类对应 out_info 表，Card 类对应 card 表，Passenger类对应passenger表，PassengerRide类对应passenger表，CardRide类对应card_ride表。对于数据库里的外键，主键以及各组长属性，我们在model里都做了相应的映射。

db_handler.py 这里实现了基本功能相关的方法。这些方法都位于DatabaseHandler类里。Delete_line实现了删除一条地铁线的功能，输入地铁线的名字就可以帮你删除名字相同的地铁线。

Find_line 实现了查找line的网址，输入地铁线名就会返回地铁线的网址。

Add_line实现了加入一条地铁线，要输入line_info,如果加入成功，会返回加入成功信息。Modify_line实现了修改地铁线信息，输入line_info，这条地铁线的信息就会相应修改。Add_station实现了加入一个地铁站，输入station_info，如果加入成功，会返回加入成功信息。

Delete_station实现了删除一个地铁站，输入地铁站名（英文名），会帮你删除相应地铁站。删除成功会返回成功信息。(以下作为例子简单呈现，其他的简易API函数构造原理相似，本文档暂且略去)

```
def delete_line(self, line_name):
    try:
        line = Line.get(Line.line_name == line_name)
        line.delete_instance()
        return f'Line {line_name} deleted successfully'
    except Line.DoesNotExist:
        raise NotFoundException(f'Line {line_name} not found')
```

Modify_station 实现修改地铁站信息，输入station_info，修改成功会返回成功信息。Insert_station_before实现在指定位置插入地铁站，输入line_name,before_station_name(插入在此站之后)，station_name(插入站名称)。

Remove_station_from_line实现了将一个地铁站从线路上移除，输入line_name,station_name即可，会帮你移除相应地铁站。

Find_n_stations实现了查找一个站向前或向后数第n个站的名字，输入line_name,station_name(从哪个站开始数)，is_forward(是否向后数),n(数几站)。会返回找到站点的名字。

Passenger_board实现了乘客（卡）上车功能，输入id,start_station_name，会将这条记录加入ride里，并且返回上车信息。

Passenger_alight实现了乘客（卡）下车功能，输入id,end_station_name。会帮相应ride加入下车时间地点并打印票价，下车成功信息。

Find_ticket_price是上述方法的一个辅助方法，通过读入上车站下车站返回相应票价。

Get_ongoing_passenger_rides实现了获取当前上车但还未下车的乘客相应乘车记录。返回相应乘车信息。

Get_ongoing_card_rides实现了获取当前上车但还未下车的卡相应乘车记录。返回相应乘车信息。

高级API的实现思路

Bonus2.1 find_path 实现了通过输入起始站名和终点站名，返回从起点到终点的最短路径。Add_stations_and_edges_for_line是这个方法的辅助方法，将站作为节点添加到gragh中，并为同一条地铁线上的地铁间添加edge，find_path通过dijkstra算法找到了最短路径并返回。

```
//导入Graph并且根据交通网构造添加边
metro_network = nx.Graph()
    for line in Line.select():
        self.add_stations_and_edges_for_line(metro_network, line)
    try:
        # 使用 Dijkstra 算法找到最短路径
        shortest_path = nx.dijkstra_path(metro_network, source=start_station,
target=end_station
        return shortest_path
    except nx.NetworkXNoPath:
        # 如果没有找到路径，则返回 None 或抛出异常
        return None
```

Bonus2.4 build_bus_lines_to_stations_mapping是一个辅助方法，将公交线上的公交站添加至此线路上。

Find_stations_by_bus_line输入公交线的名称，返回公交线上的公交站。

Find_nearby_stations_by_metro_station 输入地铁站的名字，通过查找相应地铁站附近的公交站，返回附近公交站名字的集合。

这三个方法综合实际上是建立了完整的地铁交通线路图

```
//在初始化时候就构建映射
self.bus_lines_to_stations = defaultdict(set)

# 构建映射
self._build_bus_lines_to_stations_mapping()

//然后在方法中直接使用set容器查找
metro_station = metro_stations.get()

# 初始化一个集合来存储所有找到的公交站名
bus_names = set()

# 查找与该地铁站点关联的所有公交信息
bus_infos = BusInfo.select(BusInfo.busname).where(BusInfo.station == metro_station)
```

Bonus2.5 search_rides 输入查询的参数（可以是1-n个参数），返回满足要求条件的passenger_rides,card_rides.可以输入id,start/end station,start/end_time from以及start/end_time to,注意time_from和time_to必须一起输入，来为起始或结束时间划定一个区间。Passenger_rides和card_rides里面是ride对象的集合，可以打印相关的属性来查看结果。

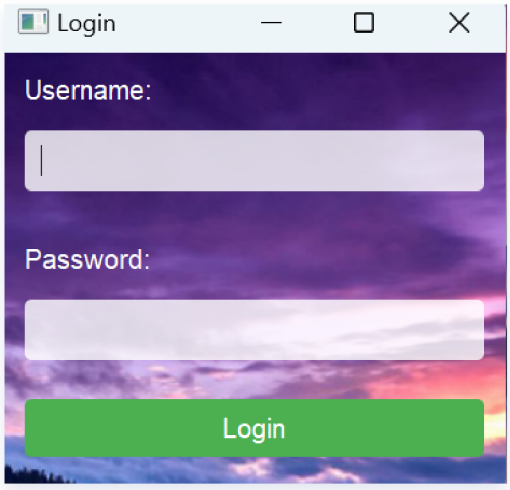
```
if 'ride_id' in search_params and search_params['ride_id'] != '/':
    passenger_ride_query = passenger_ride_query.where(PassengerRide.ride_id ==
search_params['ride_id'])
    card_ride_query = card_ride_query.where(CardRide.ride_id ==
search_params['ride_id'])
//其他变量的筛选

//最后进行类型转换，变成可序列化的类型
passenger_rides_dicts = list(passenger_ride_query.dicts())
passenger_rides_json = json.dumps(passenger_rides_dicts, ensure_ascii=False,
cls=DecimalDatetimeEncoder)
```

GUI交互界面的设计

为了方便API与数据库的交互，我们小组使用python语言搭建了GUI交互界面，方便用户与数据库的交互，在我们GUI模块中包含一下模块。

```
gui/
    src
    chooser.py
    function.py
    login_window.py
    main.py
```

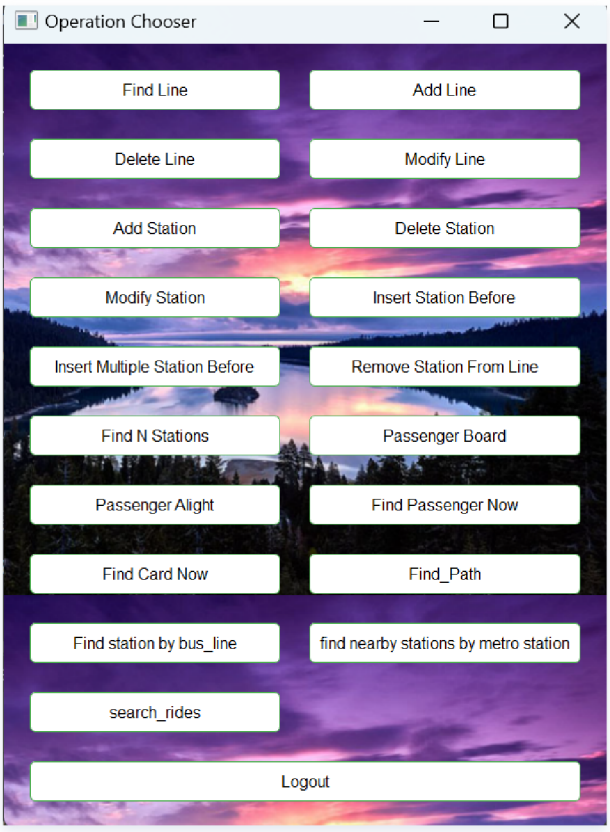
其中登录界面login_window即用户输入账户名、账户密码的界面，输入正确的用户信息之后，进入chooser界面，即提供对API功能的选择，选择对应的功能之后可以跳转到function模块中对应的GUI界面，对对应的API交互功能进行具体实现。(部分截图见后文)



本项目主要借助python中 `PyQt5` 库来搭建一个主要以按钮构成的GUI界面，包括但不限于 `QWidget` ，`QVBoxLayout` ，`QLabel` ，`QPushButton` 等核心模块。

登录界面主要在调用类时候对 `_initUI_` 默认初始化函数进行调用，其中采取了以下一些操作，包括设置按钮、标签以及设置了排版布局，另外在美化部分我还进行了字体、背景图片、字号等一些个性化的操作，让登陆界面更加美观。具体样式见下图所示。

```
def initUI(self):
    self.setWindowTitle('Login')
    self.setGeometry(100, 100, 700, 500)  # Increase window size
    # 设置背景图片
    self.set_background_image('background.png')
    # 添加间隔以调整布局
    layout.addSpacerItem(QSpacerItem(20, 40, QSizePolicy.Minimum,
QSizePolicy.Expanding))
    self.label_username = QLabel('Username:', self)
    self.label_username.setFont(QFont("Arial", 20))
    layout.addWidget(self.label_username)
    /...其余接受输入同理
    //设置button
    self.button.clicked.connect(self.login)
    layout.addWidget(self.button)
    # 设置样式表
    self.setStyleSheet("......")
    //图片框中置
    self.center()
```



同理，完成登陆之后 `chooser.py` 中主要进行的操作的选择操作，这里由于篇幅的限制原因，下面只进行一些简单的解析，具体运行的结果请见具体代码。完成chooser的选择之后，系统会跳转到对应的 `function.py` 中的GUI 模块进行操作。

在function模块中，对应的API会接受需要接受的输入，然后把这些输入通过一些格式的转换，传给我后续通过HTTP思路实现的 `routes.py` ，里面记录的是各个服务器访问路径需要执行的具体函数，由于篇幅的限制，下面仅以 `AddStation` 作为例子进行解释。

我把每一个方法作为一个类 `class` 来记录，这样方便对方法内的操作进行管理，另外由于 `function.py` 中很多具体的实现模块在按钮的创建等地方有相似的地方，所以也进行了 `class UIUtils` 类的建立，里面主要写了一些可以重复利用的工具。

```
class AddStation(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        //界面设计
        ....
        //
        self.label_name = UIUtils.add_label(layout, 'Station Name:', self)
        self.station_name_edit = UIUtils.add_edit(layout, self)
        //创建district等其他的输入框
        ....
        //如果按下按钮就进行执行
        self.add_button = UIUtils.add_button(layout, 'Add Station', self.add_station,
self)
    //返回按钮
    def go_back(self):
        self.main_window = chooser.MainWindow()
        self.main_window.show()
        self.close()
    //真正的执行
    def add_station(self):
        station_name = self.station_name_edit.text()
        ....
        if not all([station_name, district, intro, chinese_name]):
            QMessageBox.warning(self, 'Input Error', 'Please fill in all fields.')
            return
        data = {
            'station_name': station_name,
            'district': district,
            'intro': intro,
            'chinese_name': chinese_name
        }
        //如果读到了所有数据就把这些数据以json的格式传给对应的route，进行操作，这里是当然
是"post"操作
        try:
            response = requests.post('http://127.0.0.1:5000/add_station', json=data)
            if response.status_code == 200:
                QMessageBox.information(self, 'Success', 'Station added
successfully.')
                self.clear_fields()
            else:
```

```
                QMessageBox.warning(self, 'Error', f"Failed to add Station:
{response.text}")
        except requests.exceptions.RequestException as e:
            QMessageBox.critical(self, 'Error', str(e))
```

按照以上的方法，GUI交互界面就基本上设计完毕。

后端服务器的设计

与GUI相似，我的后端服务器设计server有着类似的结构，接下来我会根据我的构造展开进行解释。

```
server/
    src
    config.py    //记录数据库用户的配置
    models.py    //使用ORM映射实现的对象关系映射
    db_handler.py //直接对models中也就是数据库中数据进行操作的执行者
    routes.py //定义了各种http_web格式的链接，来对应具体的db_handler中的数据库操作
    exceptions.py   //定义了一些可能出现的异常情况
```

使用ORM映射

我定义了模型类 `models.py` ，直接将数据库表与python联系起来。我们主要使用的是Python中Peewee库实现映射关系，具体实现如下所示（仅以Station类的创建为例子）。

```
from peewee import (......)
class BaseModel(Model):
    class Meta:
        database = db
class Station(BaseModel):
    station_name = CharField(primary_key=True)  # 假设station_name是唯一的
    district = CharField()
    intro = TextField()
    chinese_name = CharField()
    //对应具体数据库的某一个表
    class Meta:
        db_table = 'station'
```

这样在操作时候可以更加方便的抓取数据以及对数据进行操作，同样也避免了直接使用SQL语句对数据库操作，取而代之的是用Python进行对对应的类进行数据操作，我们认为有以下优点：

1. **简化数据库操作**：ORM将数据库表映射到Python类，可以直接操作类实例来进行数据库操作，避免了繁琐的SQL语句。

2. **提高代码可读性和维护性**：使用ORM，数据库操作更接近于面向对象的编程风格，使代码更加直观和易读。
3. **数据库无关性**：Peewee支持多种数据库（如SQLite、MySQL、PostgreSQL），使用ORM可以轻松切换数据库而无需修改大量代码。
4. **简化关联关系处理**：通过定义外键和关联关系，可以轻松地进行跨表查询和操作，简化了复杂查询的实现。

总的来说，使用ORM可以大大简化数据库操作，提升代码的清晰度和可维护性，同时减少手写SQL带来的错误风险。Peewee作为一个轻量级的ORM库，提供了丰富的功能来支持各种数据库操作需求。

HTTP/RESTful Web服务

在服务器的设计中，我也使用了Flask框架来定义Web API路由，并与数据库进行交互。主要在 `routes.py` 进行实现，同样以下进行简析：

```
@app.route('/add_station', methods=['POST'])
def add_station():
    username = db_credentials.get('username')
    password = db_credentials.get('password')
    //判断用户是否有权限对数据库进行操作
    if not username or not password:
        return jsonify({'error': 'Unauthorized'}), 401
    data = request.json
    db_handler = DatabaseHandler(username, password)
    try:
        result = db_handler.add_station(data)
        db_handler.close_connection()
        return jsonify({'message': result}), 200
    // ...抛出其他异常
```

通过对 `@app.route` 中的定义，可以对add_station进行定义并最终通过db_handler进行具体操作，然而这样HTTP服务的特点就是可以定义了多个端点，如 `/login` 、 `/find_line/<line_name>` 、 `/add_line` 等，通过装饰器 `@app.route` 来指定路由路径和允许的方法（如GET、POST、DELETE），RESTful服务提供统一的接口来操作资源，最后通过Flask路由和函数来实现这些接口。另外，可以在这里设置一个数据过滤站，进行数据正确性的检查，返回如200，404，401，500这样的约定的response信号，方便后端服务器反应，给GUI 反馈。

连接池的使用

我们同样使用了连接池技术，代码中使用了 `PooledPostgresqlExtDatabase` 来创建连接池，这是一种通过Python实现数据库连接池的方式。

```
from playhouse.pool import PooledPostgresqlExtDatabase
db = PooledPostgresqlExtDatabase(
    //User information...
    max_connections=100,  # 最大连接数
    stale_timeout=600  # 空闲连接超时
)
```

- `max_connections=100`：指定连接池中最大连接数为100，这意味着同时最多可以有50个连接被创建并保持活跃状态。
- `stale_timeout=600`：设置空闲连接超时时间为600秒（10分钟）。超过这个时间的空闲连接将被自动关闭并移出连接池。

我们认为使用连接池非常显著的优势是**性能提升**，因为通过使用连接池，应用程序可以快速获取和释放数据库连接，减少了频繁建立连接的开销，从而提高了整体性能。

使用后端框架

本部分作为一个总结，实际上在前面提到HTTP Web服务时候就提到了我们使用的Flask框架。

在使用 Flask 后端框架的时候，我们发现确实是很方便的一个库：

1. **简洁性和易用性**：Flask 是一个轻量级的微框架，设计简单，易于快速开发 web 应用。
2. **可扩展性**：Flask 拥有丰富的扩展插件，可以轻松集成 ORM（如 Peewee）、表单验证、身份验证等功能。

```
from flask import Flask
from peewee import PostgresqlDatabase
app = Flask(__name__)
//配置其他文件...
if __name__ == '__main__':
    app = create_routes(app)
    app.run(debug=True)
```

作为main类即启动flask框架中http的“首页”网页地址即可使用，较为方便且直接。

其他额外功能设计

使用MySQL

基于Project1，我们同样使用了MySQL进行了实验，事先准备好数据集，在使用ORM时候仅仅只用修改部分语句，就可以直接对ORM映射之后类的数据进行实验，非常方便，由于篇幅有限，以下仅作部分简述。

其实由于我们使用了Peewee支持的ORM，相当于只需要把PostgreSQL的连接方式迁移到MySQL中，并且在db_handler中进行同样的修改，在config配置中更改为MySQL的配置就可以。

```
models.py// ...
db = PooledMySQLDatabase(
    // ..其他相关参数
    port=3306,  # MySQL 的默认端口
    // ...MySQL中相关连接池参数
)
```

配置完了之后同样同样的方式可以与MySQL中的数据进行交互。

打包生成.exe

最后为了方便运行，我将gui文件夹的Main函数打包成.exe文件，方便用户使用。

我应用cx_Freeze在gui目录下建立了setup.py文件

```
from cx_Freeze import setup, Executable
# 配置包含的文件和目录
build_exe_options = {
    "packages": ["os", "sys", "PyQt5"],
    "includes": ["chooser", "function", "login_window"],//添加relate文件
    "include_files": [("src/background.png", "src/background.png")],
}
# 配置生成的可执行文件
executables = [
    Executable(
        "main.py",
        base="Win32GUI",  # 指定为 GUI 应用程序，避免弹出控制台窗口
        target_name="main.exe"
    )
]
# 设置打包配置
setup(
    .. // ..其他参数
)
```

最后在build文件夹里找到main.exe文件，即可作为前端GUI直接使用，但是在运行之前必须得开启后端服务器端口才能正常收发数据运行。

项目总结

本项目总体分为后端服务器和前端的GUI显示模块，通过本项目的学习，我学习到了通过Python中的Flask框架搭建简易的服务器，并通过Python中PyQt5书写的前端GUI完成用户与后端服务器的交互。

项目存在的不足：

1. 前端GUI部分的设计有些过于冗长，有些可以公用的资源其实可以运用重写的方式进行调用，而非这样拷贝粘贴
2. 由于时间的关系，没有进行数据库用户权限等的测试。
3. 另外，我认为本项目的结构还可以更加优化，代码结构可以更加合理一些。