大作业项目总结报告

题目： B. The Taste of Beihang - What to Eat at Beihang University

小组成员：

2023 年 7 月

一、功能简介

1. 支持用户的登录和注册，提供登录验证，注册重复验证功能。
2. 支持用户个人信息管理，设置用户支持中心。
3. 支持对菜品的增删改操作，区别管理员用户与普通用户。
4. 支持用户对菜品、柜台、餐厅的收藏管理。
5. 支持用户记录吃过的菜肴，提供添加，修改，删除和搜索功能。
6. 支持用户口味分析，最爱菜肴分析。
7. 支持用户对菜肴发表评论，并支持多个用户之间的互动交流。
8. 支持菜肴搜索，以及根据关键词提供菜肴建议。
9. 支持根据时间、用户口味偏好和以往用餐记录进行膳食推荐，提供滚动显示榜单，必吃榜和热门榜单，设置趣味小游戏。

二、已完成任务

必做任务完成情况（7/7）

1. 实现食堂大厅、食品柜台、菜品的 CRUD（创建、读取、更新、删除）操作。至少对 150 种真实的菜肴进行编目，包括早餐、饮料和主食。
2. 允许用户记录自己吃过的菜肴（支持添加、删除和修改）。
3. 允许用户收藏特定菜肴、食品柜台和食堂大厅。
4. 根据早餐、午餐和晚餐的时间、用户口味偏好和以往用餐记录提供膳食推荐。
5. 显示“必吃”排行榜，根据用户购买菜肴的次数进行更新。
6. 允许用户对菜肴发表评论，并支持多个用户之间的互动交流。
7. 建立用户支持中心，支持注册、登录和个人信息管理。

选做任务完成情况（4/4）

选做任务自行设计的部分，请标名自定义。

1. 共添加 600 道菜。

2. 提供单词建议，根据用户输入单词，进行与单词相关的菜肴推荐。

3. 据用户收藏与用餐记录，提供用户对于酸甜苦辣咸五味的口味分析和最爱分析排行榜（自定义）

4. 增加趣味小游戏和”帮你想想吃什么“小组件（自定义）

三、总体设计方案

1.功能 1

1.1 功能描述

登录和注册界面主要分为前后端两个部分，前端负责 UI 搭建和向后端发送信息，后端负责连接服务器。用户在登录时向后端发送输入的用户名和密码，根据后端的验证判断是否能够登录系统，只有在登录成功后才能进入主界面。注册时，前端首先会检查用户所输入信息是否合法，如果是，向后端发送用户注册所需信息，根据后端返回判断是否注册成功。

1.2 实现逻辑与核心代码

登录界面在 main.py 中创建一个 MyLogin 对象，并调用 show 功能，即实现了登录界面的呈现。接下来登录验证部分，是与登录按钮的按下事件所联系的，其主要实现如下：

```
def go_to_mainWindow(self):
    # 检验用用户名密码是否正确
    account = self.userName.text()
    password = self.password.text()
```

```
    database = DBOperator()
    ok = database.sign_in(account, password)
    if ok:
        MainWindow = MyMainWindow(account=account)
        MainWindow.show()
        self.close()
    else:
        self.createErrorInfoBar('用户名或密码错误')
```

其中，首先会将用户输入的用户名和密码调用后端实现的 sign_in 函数，如果成功则将用户名作为参数传递给主界面的构造函数进入主界面，并且关闭登录界面。否则，会出现"用户名或密码错误的弹窗"，这里我使用了 PyQt_Fluent_Widgets 中的 InfoBar 实现弹窗的效果。

在后端部分，我们在数据库中使用了一个表来存储用户的用户名、密码、昵称等信息。在用户登录时，会先查找表中是否含有包含输入用户名和密码的项，若包含则登录成功，不包含则表明用户不存在或用户名密码不匹配，具体函数实现如下：

```
def sign_in(self, name, passwd):
    return len(self.execute(f"select * from people where name = '{name}' and passwd = '{passwd}';")) != 0
```

注册界面对象的创建是和“注册新用户”按键的按下事件联系的，当信息输入不全时，会使用 InfoBar 弹窗提醒请补全信息，输入的两次密码不相同时，会提

醒用户输入相同的密码，这两个功能是前端就可以判断的。当信息输入完全，会将信息调用 sign_up 发送给后端判断当前用户名是否注册过信息，如果是新用户注册成功，否则会弹窗提醒。提交信息并异常处理的具体实现代码如下：

```
def register_and_errorCatch(self):
    try:
        database = DBOperator()
        # 信息不全处理
        if self.userName.text() == '' or self.nickName.text()
== '' or self.password.text() == '' or self.password_2.text()
== '':
            self.createErrorInfoBar('请补全信息')
        # 两次密码输入不统一处理
        elif self.password.text() != self.password_2.text():
            self.createErrorInfoBar('请输入相同的密码')
        # 成功注册
        else:
            userName = self.userName.text()
            nickName = self.nickName.text()
            password = self.password.text()
            ok = database.sign_up(userName, nickName,
password)
            # 重复注册处理，查询数据库是否有这个用户名
```

```
                if ok:
                    # 输出信息，恢复标签
                    self.createSuccessInfoBar('恭喜你，注册成功！
')
                    self.userName.clear()
                    self.userName.setPlaceholderText('example
@example.com')
                    self.nickName.clear()
                    self.nickName.setPlaceholderText('Free to
yourself')
                    self.password.clear()
                    self.password.setPlaceholderText('••••••••••
•')
                    self.password_2.clear()
                    self.password_2.setPlaceholderText('••••••••••
•••')
                else:
                    self.createErrorInfoBar('用户已存在')
        except Exception as e:
            print(e)
```

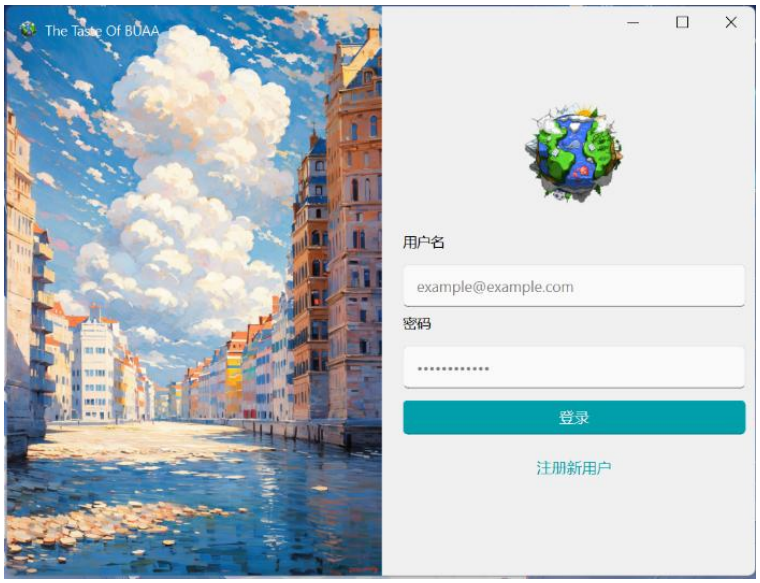
在后端部分，当注册函数被调用时，会先在表中查找有没有和输入相同的用户名，如果有则注册失败，否则把新增的用户名和密码添加入表内，实现用户注

册，具体函数实现如下：

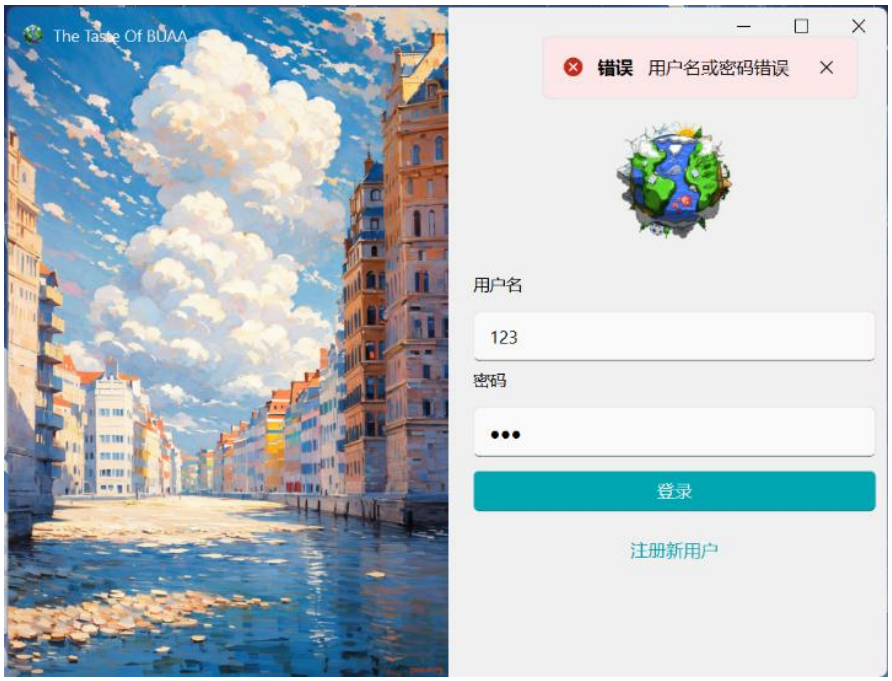
```
def sign_up(self, name, nick, passwd):
    name = "'" + name + "'"
    exist = self.execute(f'select * from people where name
= {name};')
    if len(exist) > 0:
        return False
    self.execute(
        f"insert into people (name, nick, passwd, sex, birth,
fav, ates) values ({name}, '{nick}', '{passwd}', 0, '', 0, 0);")
    return True
```

1.3 具体实现效果

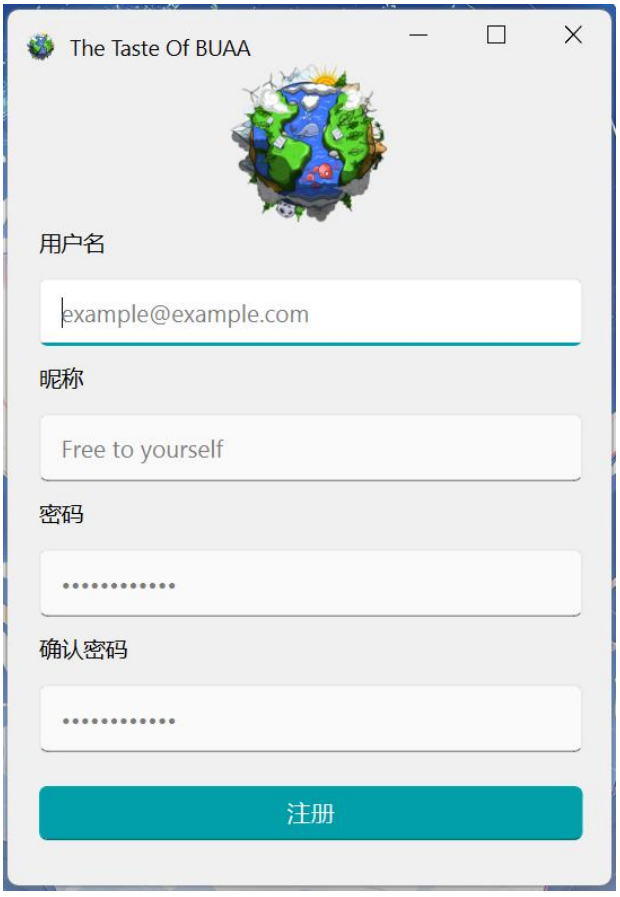
登录界面



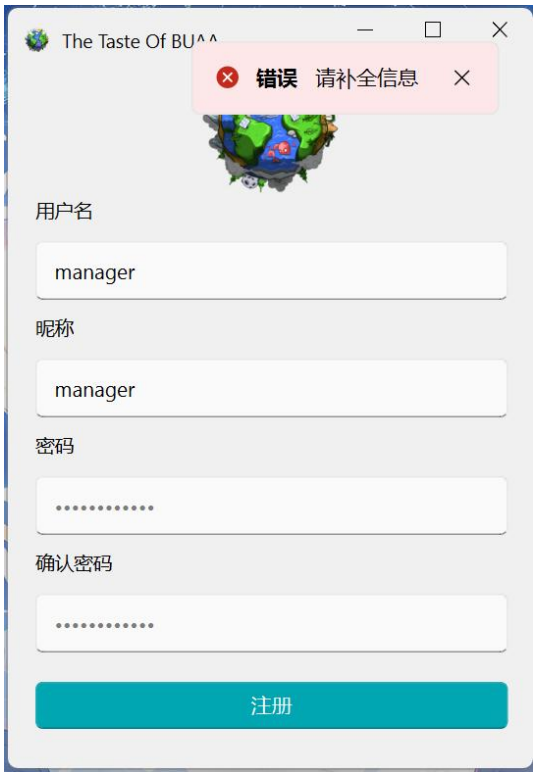
登录失败效果



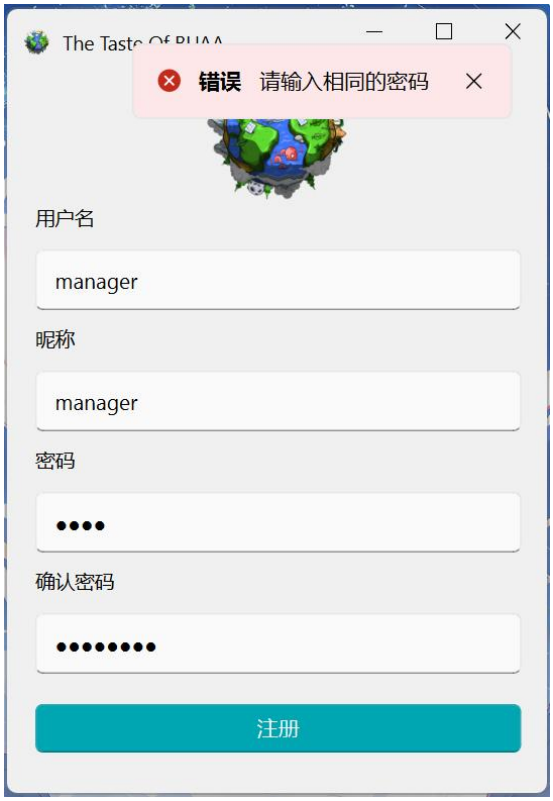
注册界面



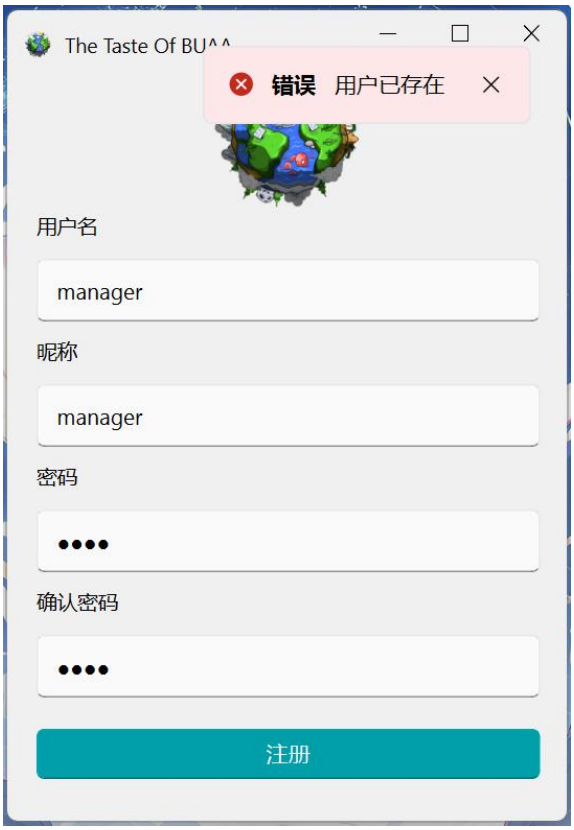
注册失败（信息不完整）



注册失败（两次输入密码不同）



注册失败（用户已注册）



2.功能 2

2.1 功能描述

登录进入主界面后，可以对于用户个人信息进行修改，在注册时并未设置用户的性别、生日、头像等属性，在用户支持中心可以进行设置和修改。完成登录后，此时我们已经拥有了当前登录用户的用户名，在个人信息呈现呈现板块，通过从向后端发起请求，获取当前用户名的昵称、性别、生日，并设置页面 Label 的 text 显示，在个人信息编辑页面，通过向后端发送将要修改的信息，可以选择单独设置昵称、性格、生日、密码的一项或多项，在修改密码时，也会检验原密码是否正确，确保用户的个人信息安全。当注册成功时会出现成功的提示弹窗，并且刷新界面的显示。

2.2 实现逻辑与核心代码

个人信息界面和编辑界面通过 StackWidget 进行布局。登录进入主界面后，

通过用户 account 设置个人信息各个标签的显示效果，并且将 StackWidget 设置到第一页，即个人信息页，实现代码如下：

```
def initUserInfo(self):
    # 设置个人信息页为首页
    self.stackedWidget.setCurrentIndex(0)
    ......
    # 设置个人信息,头像,昵称，用户名，性别，生日，收藏数，吃过数
    # (name,nick, passwd,sex,birth,fav,ates,photo)
    database = DBOperator()
    person = database.get_person(self.account)
    name = person[0]
    nick = person[1]
    if person[3] == 0:
        sex = 'Not edited yet'
    elif person[3] == 1:
        sex = '男'
    else:
        sex = '女'
    birth = person[4] if person[4] != '' else 'Not edited yet'
    fav = person[5]
    ates = person[6]
```

```
    self.userNameShow.setText(name)
    self.nickNameShow.setText(nick)
    self.UserName.setText(nick)
    self.genderShow.setText(sex)
    self.birthdayShow.setText(birth)
    self.favouriteNum.setText(str(fav))
    self.eatenNum.setText(str(ates))
```

将从后端获得的个人信息设置给相应的标签，即完成了个人信息的呈现部分。

在个人信息修改界面，我们使用了 LineEditLabel 作为个人信息的输入框，在框中输入个人信息后，可以通过 Label 的 text()功能获取用户的输入，之后就可以调用后端提供的 update_person 功能，通过传入用户名，要修改的字段和待修改的值，即可完成用户个人信息的修改，另外，错误弹窗也是用到了之前提到的 InfoBar。

对于每位用户，我们会在数据库中保存其用户名、昵称、密码、性别、生日、收藏菜数、已吃菜数和头像，其中用户名、昵称和密码是在注册时必须填写的，其他可在后面于个人主页中更新，更新函数实现如下，该函数接收更新字段为参数，支持用户灵活的更新单个字段：

```
def update(self, name, field, value):
    if field in ['nick', 'passwd', 'birth']:
        value = "'" + value + "'"
    self.execute(
```

```
        f"update people set {field} = {value} where name =
'{name}';")
```

此外，用户还可以自定义自己的头像，点击头像，会出现一个选择本地图像的弹窗，这是通过 QFileDialog 实现的，通过将本地的图片地址利用 update_person 传递给后端，即可实现头像的更新，在之后的每次登录后也都会显示最新的头像，更新头像的代码实现如下：

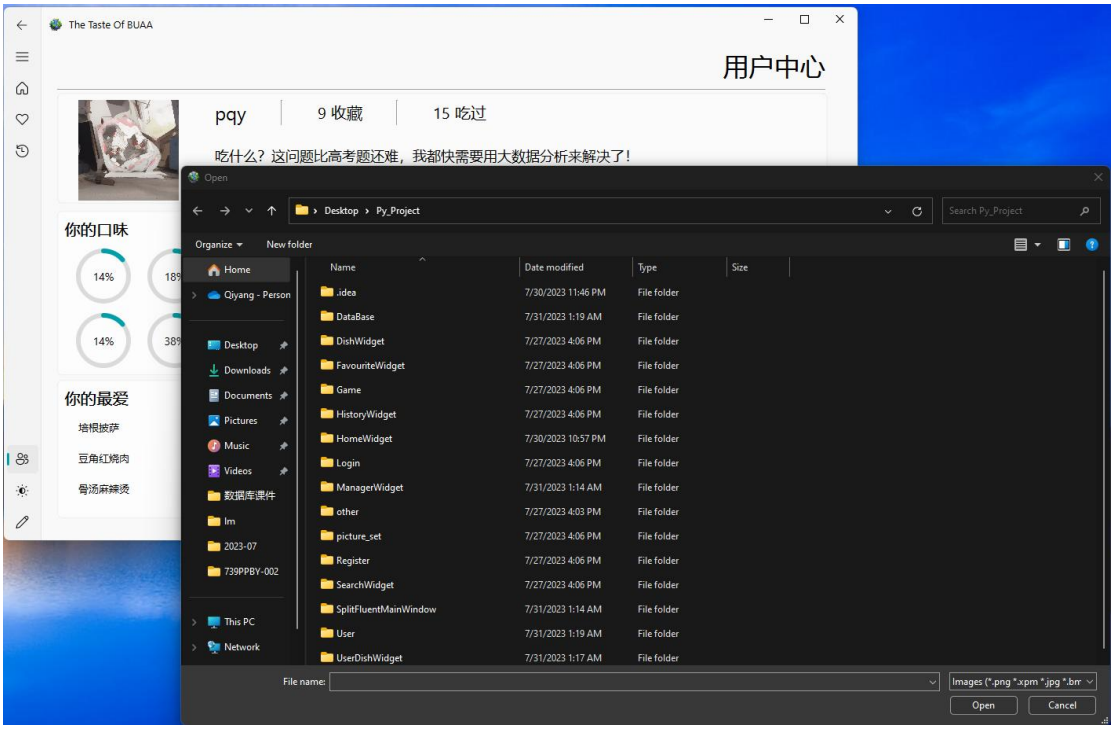
```
def uploadProfilePhoto(self):
    file_dialog = QFileDialog()
    file_dialog.setFileMode(QFileDialog.ExistingFile)
    file_dialog.setNameFilter('Images (*.png *.xpm *.jpg
*.bmp)')
    if file_dialog.exec_():
        file_path = file_dialog.selectedFiles()[0]
        pixmap = QPixmap(file_path)
        self.ImageLabel.setPixmap(pixmap.scaled(self.Imag
eLabel.width(), self.ImageLabel.height()))
        # 上传图片路径到云端
        dataBase = DBOperator()
        dataBase.update_person(self.account, 'photo',
file_path)
        self.createSuccessInfoBar('头像上传成功')
```

2.3 具体实现效果

个人信息显示

个人信息修改

头像上传



3. 功能 3

3.1 功能描述

对于管理员用户，拥有对于菜品的增加和删除权限，当管理员登录进入时，会开放管理菜品的入口，在菜品管理界面，菜品会以餐厅->柜台->菜品的形式组织成为一个树状结构，可以对于餐厅、柜台、菜品进行删除操作。

3.2 实现逻辑与核心代码

在管理员界面，我们将所有菜品组织成为嵌套的树形结构，将餐厅柜台和菜品组织为{餐厅:{柜台:[菜品、]},}的形式，以这样的形式，我们就结合了 TreeWidget 实现菜品的树形显示，以树形组织能够很直观的观念菜品，对于添加菜品和删除菜品也很方便，我们将树形结构存在数据库中，每次登录时可以通过遍历树形表，以此构建 TreeWidget，在增添菜后，也会及时更新数据库中的树形结构，初始化树形结构的代码如下：

```
def init_dish_graph(self):
```

```
    database = DBOperator()
    dish_graph = database.execute('select * from globe;')
    if dish_graph[0][0] == '':
        self.dic = {}
    else:
        self.dic = dict(ast.literal_eval(dish_graph[0][0]))
        for key in self.dic.keys():
            restaurant_item =
QtWidgets.QTreeWidgetItem(self.resturantWidget)
            restaurant_item.setText(0, key)
            for key_1 in self.dic[key].keys():
                counter_item =
QtWidgets.QTreeWidgetItem(restaurant_item)
                counter_item.setText(0, key_1)
                for dish in self.dic[key][key_1]:
                    dish_item =
QtWidgets.QTreeWidgetItem(counter_item)
                    dish_item.setText(0, dish)
        self.resturantWidget.expandAll()
```

在管理员界面，添加餐厅功能和添加柜台功能的主要实现逻辑是在{餐厅:{柜台:[菜品、]},}插入新的餐厅和在已有的餐厅中插入新的柜台，简单来说就是在字典和子字典中插入新的项。在点击删除时，可以点击相应条目去删除餐厅、

柜台和或菜品，前两者通过树形结构递归的进行删除操作，菜品删除的核心代码如下：

```
def deleteItem(self):
    database = DBOperator()
    selected_item = self.resturantWidget.currentItem()
    if selected_item:
        parent_item = selected_item.parent()
        if parent_item:
            parent_parent_item = parent_item.parent()
            if parent_parent_item:  # 说明就是直接是菜
                # 删除 treeWidget 中的项
                parent_item.removeChild(selected_item)
                # 删除大字符串中的菜
                self.dic[parent_parent_item.text(0)][parent_item.text(0)].remove(selected_item.text(0))
                # 删除总的菜单中的菜
                dish_id = database.get_id(selected_item.text(0), parent_item.text(0), parent_parent_item.text(0))
                database.del_dish(dish_id)
            else:  # 说明是柜台
                parent_item.removeChild(selected_item)
```

```
                # 删除大字符串中的柜台
                self.dic[parent_item.text(0)].pop(selected_ite
m.text(0))
                # 将他的孩子(菜)也一块删除,递归删除
                try:
                    self.deleteCounter(parent_item.text(0),
selected_item.text(0))
                except Exception as e:
                    print(e)

        else:
            index =
self.resturantWidget.indexOfTopLevelItem(selected_item)
            self.resturantWidget.takeTopLevelItem(index)
            # 删除大字符串中的餐厅
            self.dic.pop(selected_item.text(0))
            # 递归删除餐馆里所有的菜
            self.deleteRestaurant(selected_item.text(0))
        graph = str(self.dic)
        database = DBOperator()
        dst_graph = '"' + graph + '"'
```

```
        database.execute(f'update globe set total =
{dst_graph};')
```

在点击添加菜品时，会弹出添加菜品的框，在填写菜品的相关信息后，会调用后端提供的 add_dish 函数，这样就完成了菜品的添加，add_dish 的定义如下：

```
def add_dish(self, dish: str, tp: int, heat: int, taste: int, bar:
str, hall: str, img: str):
    self.dishOp.add(dish, tp, heat, taste, bar, hall, img)
```

当处于用户模式下，我们不会对用户开放删除和添加菜品的权限，只对用户开放查看**菜品总数**和**菜品详情**的功能，实现管理员和用户分离。在用户可见界面，我们也是通过调用前面提到的 init_dish_graph 初始化树形结构，同时我们调用 get_all_id 获取所有 id，并通过表的长度获取当前所拥有的所有菜。当我们选择某一个菜品时，可以点击查看详情按钮，这时会弹出菜品的详情页面，便于用户更细致地了解菜品。具体的菜品展示代码如下：

```
def seeInfo(self):
    database = DBOperator()
    selected_item = self.resturantWidget.currentItem()
    if selected_item:
        parent_item = selected_item.parent()
        if parent_item:
            parent_parent_item = parent_item.parent()
            # 说明就是直接是菜
            if parent_parent_item:
```

```
                # 得到菜的 id
                dish_id =
database.get_id(selected_item.text(0), parent_item.text(0),
parent_parent_item.text(0))
                dish_window =
DishDetailWindow(dish_id=dish_id, account=self.account)
                self.objectBase.append(dish_window)
                dish_window.show()
```

在数据库中，我们用一个菜品表记录菜品信息，具体包括菜品 id、名称、时间段类型（早餐、正餐、饮料）、冷热、酸甜苦辣咸五味、柜台、大厅、评论、图片。为每个菜品添加唯一的 id 是为了以简洁的方式索引菜品，为之后的已吃和收藏的记录提供便利。我们为了实现菜品 id 的自动获取，另加入了一个 n 行 1 列的菜品 id 分配表，表的每一行值为当前行的行号，代表了当前最大分配的 id，每当新加入一个菜品时就会给这个 id 分配表加一行并给菜品分配一个 id。而对于菜品的图片，我们在数据库中采用了 LONGBLOB 的格式来存储和获取，在上传时，图片会以二进制的形式被读取，而查询获得的二进制图片会在后端转换成 PIL.Image 格式，以便在前端中被 QImage 等数据结构使用。

此外，与个人信息功能的处理类似，后端对于菜品的更新函数也是有弹性的，允许管理员更改菜品任意字段的信息。对于菜品表，后端部分自定了一个 DishesTb 类来提供其增删改查操作，类中主要函数如下：

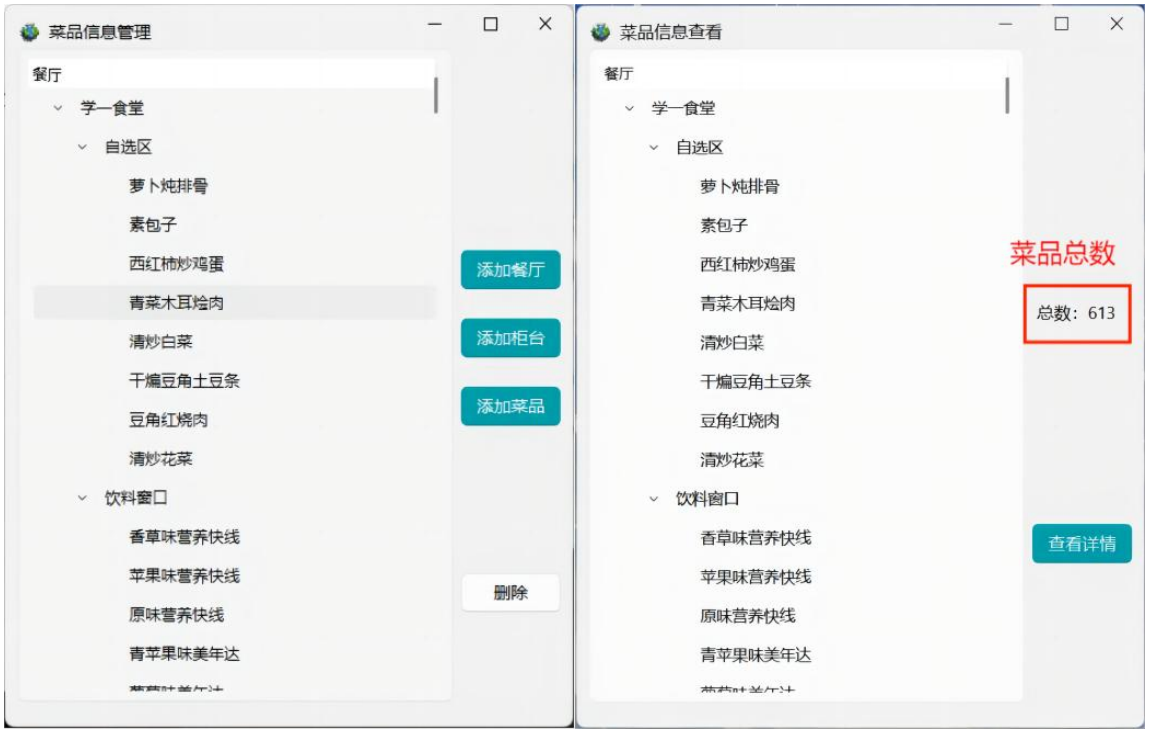
```
class DishesTb:
    def delete(self, dish_id):
```

```
        self.execute(f"delete from dishes where id =
{dish_id};")
        self.execute(f'update mapper set valid = 0 where id
= {dish_id};')

    def add(self, name, tp, heat, taste, bar, hall, img):
        dish_id = self.execute(f'select count(*) from
mapper;')[0][0]
        self.execute(f'insert into mapper (id, valid)
values({dish_id}, 1);')
        with open(img, 'rb') as f:
            image = f.read()
        query = "insert into dishes (id, name, tp, heat, taste,
bar, hall, img) values (%s, %s, %s, %s, %s, %s, %s, %s);"
        self.execute(query, (dish_id, name, tp, heat, taste,
bar, hall, image))

    def update(self, dish_id, field, value):
        if field in ['name', 'bar', 'hall', 'com']:
            value = "'" + value + "'"
        if field == 'img':
            with open(value, 'rb') as f:
```

```
                value = f.read()
            query = 'update dishes set img = %s where id
= %s'
            self.execute(query, (value, dish_id))
        else:
            self.execute(
                f"update dishes set {field} = {value} where id
= {dish_id};")

    def get_id(self, dish, bar, hall):
        return self.execute(
            'select * from dishes where name = %s and bar
= %s and hall = %s', (dish, bar, hall))[0][0]
```

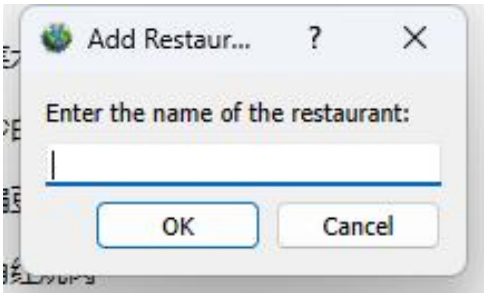
3.3 具体实现效果

树形结构


菜品信息管理
餐厅
学一食堂
自选区
萝卜炖排骨
素包子
西红柿炒鸡蛋
青菜木耳烩肉
清炒白菜
干煸豆角土豆条
豆角红烧肉
清炒花菜
饮料窗口
香草味营养快线
苹果味营养快线
原味营养快线
青苹果味美年达
添加餐厅
添加柜台
添加菜品
删除
菜品信息查看
餐厅
学一食堂
自选区
萝卜炖排骨
素包子
西红柿炒鸡蛋
青菜木耳烩肉
清炒白菜
干煸豆角土豆条
豆角红烧肉
清炒花菜
饮料窗口
香草味营养快线
苹果味营养快线
原味营养快线
青苹果味美年达
菜品总数
总数：613
查看详情

添加菜、柜台、餐厅


菜品添加
菜品名称
类型
餐厅
柜台
口感
冷
热
酸
甜
苦
辣
咸
添加



删除菜

4. 功能 4

4.1 功能描述

在菜品详情界面，我们设置了对于当前菜品的收藏按钮，同时设置对于当前菜品的所在柜台和所在餐厅的收藏按钮，在主界面，我们设置了用户的收藏管理中心，可以查看收藏的餐厅、柜台和菜品，同时支持对于它们的取消收藏操作。

4.2 实现逻辑与核心代码

为了实现对用户收藏的管理，后端采用了菜品收藏表、柜台收藏表和大厅收藏表，并在用户的个人信息中维护其收藏菜数量。收藏表中的每一行为用户名称和收藏的内容，代表了一位用户的一次收藏记录。

由于菜品、柜台、大厅收藏这三个表在结构上具有高度相似性（唯一区别是菜品用 INT 类型的 id 记录，其他二者用 VARCHAR 记录），我们使用了一个 FavTb 类来管理三个表，其中提供增删改查功能的函数接收收藏对象的类型为额外参数以确定修改哪个表。

该类的主要函数如下：

```
class FavTb:
    def add(self, name, tp, value):
        if tp != 'dish':
            value = "'" + value + "'"
        name = "'" + name + "'"
        self.execute(
            f"insert into fav_{tp} (name, {tp}) values({name},
{value});")

    def delete(self, name, tp, value):
        name = "'" + name + "'"
        if tp != 'dish':
            value = "'" + value + "'"
        self.execute(
```

```
            f"delete from fav_{tp} where name = {name} and
{tp} = {value};")


    def get(self, name, tp):
        name = "'" + name + "'"
        temp = self.execute(f'select * from fav_{tp} where
name = {name};')
        return [i[1] for i in temp]
```

对于每一个菜品详情界面，在构造函数里我传入了当前用户的 account 和当前菜品的 id，利用菜品的 id 向后端查询菜品信息进行设置详情页的各种信息，当点击收藏按钮时，相应地调用不同的处理函数以完成收藏效果，代码如下：

```
# 收藏菜品
def set_favourite_button(self):
    database = DBOperator()
    if self.dish_id not in
database.get_fav_dish(self.account):
        database.add_fav_dish(self.account, self.dish_id)
# 收藏餐厅
def set_restaurant_button(self):
    database = DBOperator()
    dish = database.get_dish(self.dish_id)
    if dish[6] not in database.get_fav_hall(self.account):
```

```
            database.add_fav_hall(self.account, dish[6])
    # 收藏柜台
    def set_counter_button(self):
        database = DBOperator()
        dish = database.get_dish(self.dish_id)
        if dish[5] not in database.get_fav_bar(self.account):
            database.add_fav_bar(self.account, dish[5])
```

在收藏呈现界面，我们使用了两个 ListWidget 用来呈现用户收藏的餐厅和柜台，使用一个 ScrollArea 用来存放每一个收藏菜品的条目，具体呈现方式，只需从后端获取用户收藏信息，调用 ListWidget 和 ScrollArea 的添加项的函数即可，同时支持删除操作，我们在本地维护了一个收藏表，我们可以查表找到要删除的是哪一项，之后就可以调用删除功能进行删除。

除此以外，我们的收藏功能具有动态刷新效果，即在菜品界面点击收藏，之后便会在收藏管理界面呈现而不是等到下一次重新登录进入系统时才进行刷新，这里我们使用了多线程和信号结合的方法实现刷新效果，收藏管理界面拥有两个进程，一个进程用来作为收藏管理页面的呈现，另一个进程作为更新提醒进程，当达到要求的时间间隔，就会向主进程发送更新的信号，此时就是调用主进程里的更新函数对于餐厅、柜台和菜品的收藏进行刷新，做到及时相应的效果。

以下是发送信号的线程，为了避免卡顿，我们设置了五秒刷新的时差：

```
class BackendThread(QObject):
    # 通过类成员对象定义信号
    update_date = pyqtSignal()
```

```
    # 每隔五秒发送更新信号
    def run(self):
        while 1:
            # 刷新 1-10
            for i in range(1, 11):
                self.update_date.emit()
                time.sleep(5)
```

当我们在菜品详情页面点击收藏时，会在数据库中增加一条新的记录，但此时收藏管理页面还可能没有显示，当我们在收藏管理页面删除一条收藏时，数据库和收藏管理界面会同时删除这条记录，因此数据库一定是领先于本地的，因此我们在更新时只需要得到数据库比本地多了什么再添加上去即可，而不是全部清空之后再设置，这样可以省去不小的开销，更新以更新餐厅收藏为例，柜台和菜品同理：

```
def update(self):
    database = DBOperator()
    new_list = database.get_fav_hall(self.account)
    added_elements = [x for x in new_list if x not in
self.restaurantList]
    if len(added_elements) != 0:
        self.favouriteRestaurant.addItems(added_elements
)
```

```
            self.restaurantList.extend(added_elements)
            added_elements.clear()
        new_list = list(database.get_fav_bar(self.account))
        added_elements = [x for x in new_list if x not in
self.counterList]
        if len(added_elements) != 0:
            self.favouriteCounter.addItems(added_elements)
            self.counterList.extend(added_elements)
            added_elements.clear()
```

5.3 具体实现效果

收藏菜、餐厅、柜台

取消收藏菜、餐厅、柜台

5. 功能 5

5.1 功能描述

在菜品详情界面，我们设置了吃过按钮，用于记录吃过当前的菜品，在主页面中我们设置了一个历史管理界面，可以在里面进行对于吃过菜品的删除、查询和修改。

5.2 实现逻辑与核心代码

为了记录用户的已吃信息，我们在数据库中维护了一个已吃表，其中每一行为一次用户进食的姓名、菜品 id 和时间。

此外，我们还会在用户添加已吃时更新在用户信息表中的已吃菜品次数。由于已吃表的操作方法与上面收藏菜品的操作方式较为相似，这里不再呈现其函数。

在菜品详情界面，我们设置了吃过按钮，通过点击按钮即可调用后端提供的 add_ates 函数，将当前菜品加上当前时间加入到用户的历史记录当中，实现函数具体如下：

```
def set_eaten_button(self):
    database = DBOperator()
    database.add_ates(self.account, self.dish_id,
QDateTime.currentDateTime().toString("yyyy-MM-dd\nH
H:mm:ss"))
```

当我们点击吃过按钮后，历史记录很快就会在历史记录界面显示，这里运用的是和收藏界面一样的多线程和增加多的更新方法，就不再赘述了。

对于历史记录界面的呈现，首先最外层我们设置了可以上下滚动的 ScrollArea，接下来我们设计了历史记录的表项，对于它的构造函数，我们需要向里面传入菜品的 id 和时间，同时我们会本地维护一个历史记录表项，用于记录当前的历史记录的所有表项，当我们要删除历史记录时，首先会调用 del_ates 删除用户的当前历史记录，同时对于表项自身调用自带的 deleteLater 方法销毁自己，最后将本地的历史记录表项表中删除。当需要全部清空时，依托历史记录表项表，分别对每一项进行之前的操作，即可完成删除操作，删除表项的代码如下：

```
def delete_history(self):
    # 删除表项表中的自己那一项
    for item in self.item_list:
        if item.time == self.time and item.dish_id ==
self.dish_id:
            self.item_list.remove(item)
    # 根据传入的 personId 删掉这条历史
    database = DBOperator()
```

```
    database.del_ates(self.account, self.dish_id, self.time)
    self.deleteLater()
```

对于历史记录的查询，我们依托于本地维护的历史记录表项，如果搜索关键词出现在菜品的餐厅、柜台、菜品名称和时间中，就会将搜索结果作为一个个表项展示在我们设计的搜索界面 UI 中，搜索历史记录的实现代码如下：

```
def show_search_result(self):
    # 清空上次记录
    self.TableWidget.setRowCount(0)
    try:
        search_content = self.SearchLineEdit.text()
        if search_content != '':
            for item in self.source_list:
                if search_content in item.dish_name.text() or
\
                        search_content in
item.restaurant_name.text() or \
                        search_content in
item.counter_name.text() or \
                        search_content in
item.history_time.text():
                    self.addTableRow([item.dish_name.text(),
item.restaurant_name.text(), item.counter_name.text(),
```

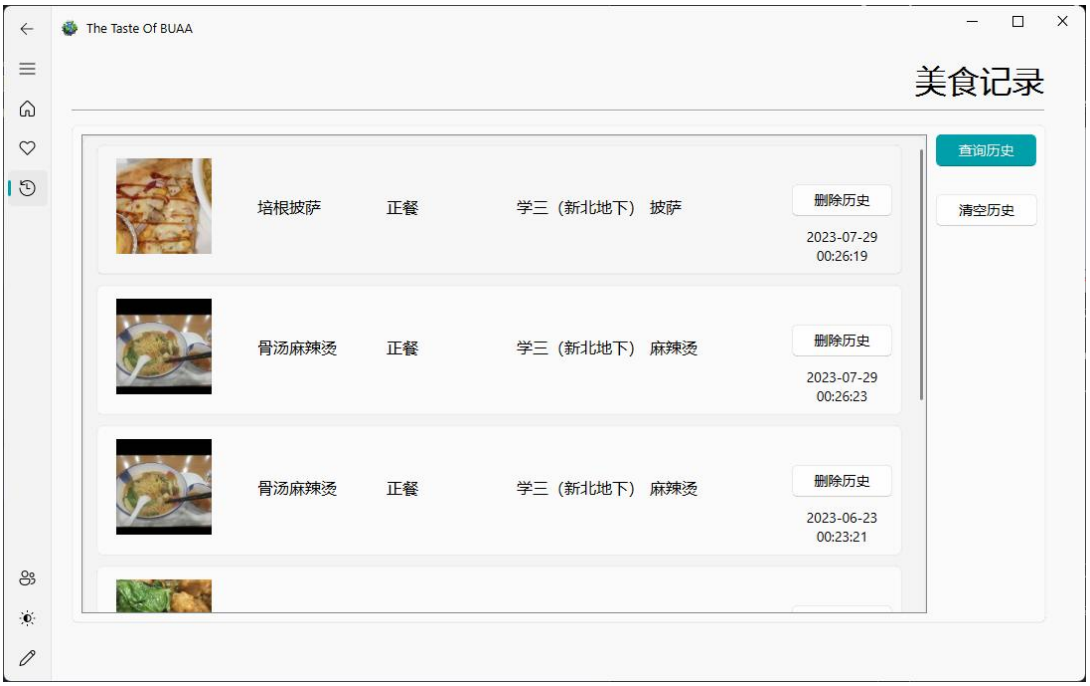
```
                                        item.history_time.text()])
    except Exception as e:
        print(e)
```

由于餐品是管理员所管理的，因此在对于历史记录的修改中我们仅允许用户对于历史记录的时间进行修改，点击历史记录项的时间，即可弹出输入框，可以输入待改变的时间。在这里，前端使用了 QInputDialog 用作输入框口的调用，在修改过程中调用了后端提供的 update_ates 方法，用于更新本条记录的时间信息。修改时间的主要代码实现如下：

```
def on_item_click(self, evnet):
    old = self.history_time.text()
    time, ok = QInputDialog.getText(
        self, "Change Time", "Enter the new
time(yyyy-MM-dd HH:mm:ss):", text=old
    )
    if time and ok:
        database = DBOperator()
        tmp = time.split(' ')
        time = tmp[0] + '\n' + tmp[1]
        self.time = time
        self.history_time.setText(time)
        database.update_ates(self.account, self.dish_id, old,
time)
```

5.3 具体实现效果

查看已吃记录



添加已吃

6. 功能 6

6.1 功能描述

在用户个人中心，我们为用户设置了酸甜苦辣咸五味比例分析，同时为用户展示最喜欢的菜品。

6.2 实现逻辑与核心代码

在菜品部分，我们使用 5 位二进制数表示其酸甜苦辣咸五味属性，例如糖醋里脊可能为 0b11000。在统计用户的口味偏好时，我们会从数据库的已吃表中获得用户所有的进食记录，并统计其吃过的菜品的酸甜苦辣咸各种口味的总次数。之后，我们会对这些口味的总次数使用 softmax 函数得到用户对于每种口味的偏爱程度，具体的偏好计算函数如下：

```
def get_person_weight(self, name):
    ates = [i[0] for i in self.get_ates(name)]
    ate_dish = [self.get_dish(dish_id) for dish_id in ates]
    ate_weight = np.array([0.0, 0, 0, 0, 0, 0, 0])
    for dish in ate_dish:
        taste = dish[4]
        heat = dish[3]
        ate_weight += np.array(
            [(heat >> 1) & 1, heat & 1, (taste >> 4) & 1,
(taste >> 3) & 1, (taste >> 2) & 1, (taste >> 1) & 1,
             taste & 1])
    ate_weight /= float(len(ates) + 0.000001)
```

```
    fav = self.get_fav_dish(name)
    fav_weight = np.array([0.0, 0, 0, 0, 0, 0, 0])
    fav_dish = [self.get_dish(dish_id) for dish_id in fav]
    for dish in fav_dish:
        taste = dish[4]
        heat = dish[3]
        fav_weight += np.array(
            [(heat >> 1) & 1, heat & 1, (taste >> 4) & 1,
(taste >> 3) & 1, (taste >> 2) & 1, (taste >> 1) & 1,
             taste & 1])
    fav_weight /= float(len(fav) + 0.000001)
    return (fav_weight + ate_weight) / 2


def get_softmax_weight(self, name):
    taste_weight = self.get_person_weight(name)[2:]
    exp_x = np.exp(taste_weight - np.max(taste_weight))
    return (exp_x / np.sum(exp_x)).tolist()
```

其中 get_person_weight()还会计算用户的冷热偏好，但在此处的统计中只使用其五味偏好。

此外，我们还会根据用户的已吃记录和收藏记录给出“你的最爱”统计。具体的统计方法使用了功能 9 推荐算法中必吃榜的推荐函数，并加上了仅使用对当前用户记录的限制，具体函数将在后面第 9 部分说明。

在前端我们调用已经实现的 get_softmax_weight 获得酸甜苦辣咸每种口味的所占比例，以此设置用于展示分布的 ProcessRing 的数值，由于口味分析会随着时间变化而不断更新，因此这里我们也使用了和之前一样的使用多线程刷新的方法。对于你的最爱板块，通过调用调用后端提供的 recommend 方法，获取用户最喜欢的菜品，最后加入到最喜欢的菜品表项当中。

6.3 具体实现效果

口味分析与“你的最爱”

7. 功能 7

7.1 功能描述

在菜品详情界面，不同用户可以发表评论，同时支持不同用户之间的回复，并且实时刷新，他人评论和回复评论能够做到实时响应。

7.2 实现逻辑与核心代码

对于每一个菜品的评论，我们将它处理为一个大的 List，它的每一项是 {'username': username, 'comment': comment_text, 'time': current_time,'reply_to':

self.reply_to_user}，将列表整体转化为一个大的字符串后就作为菜品的一个属性存储起来。当我们从数据库获取菜品的评论时，首先会把它存储在本地。

当我们发送一条评论时，会首先将将新的评论插入到本地的评论表中，之后会调用后端提供的 commit 函数将更新后的函数发送到后台数据库，同时我们还需要考虑更新问题，这个的主要框架还是利用多线程进行更新，但是这里与之前不同的是，可能本地与数据库都存在最新的消息，因此我们不能单纯从数据库中找到新的评论设置在本地，同时我们也应该把本地新增的数据加入到数据库后台当中，因此我们设计了云端和本地相结合的评论处理方式进行评论的提交和更新。以下是一个大致的示意图：

在这里我们使用了锁的机制，我们将本地的评论表作为共享对象，并且为其加上锁，提交评论和更新评论时需要获得这把锁，这样就能保证两个过程的互斥效果，在它们获得锁的之后的过程基本相同，首先从云端获取评论，将其和本地的评论合并并且去重，最后由于合并时候可能存在顺序的打乱，就再依据时间从近到远的顺序进行排序，最后将排序后的评论传递到数据库，这样就安全的实现了本地和云端的同步更新。

以下是将评论进行合并并排序的操作：

```
def merge_and_remove_duplicates(self, list1, list2):
    # 将两个列表合并
    merged_list = list1 + list2

    # 合并并去重字典
    merged_dict = {}
```

```
    for d in merged_list:
        key = (d['username'], d['comment'], d['time'],
d['reply_to'])
        merged_dict[key] = d

    # 将合并后的字典转换回列表
    merged_list_no_duplicates = list(merged_dict.values())
    # 根据时间字段对列表进行排序，从近到远
    merged_list_no_duplicates.sort(key=lambda x:
datetime.strptime(x['time'], '%Y-%m-%d %H:%M:%S'),
reverse=True)
    return merged_list_no_duplicates
```

以下是更新时获取锁并更新评论的操作，提交评论与之基本相同：

```
def update(self):
    lock.acquire()
    database = DBOperator()
    dish = database.get_dish(self.dish_id)
    if dish[7] == '':
        new_comments = []
    else:
        new_comments = ast.literal_eval(dish[7])
```

```
            self.comments =
self.merge_and_remove_duplicates(new_comments,
self.comments)
            self.update_comments(submit=True)
        lock.release()
```

最后我们的评论就会被转换为字符串进行保存，在解析时也只需要调用eval()即可将字符串转化为 List。

由于对于评论的接收和解析都已经在前端完成并转化为一个大字符串，后端只在数据库中开了一个一行一列的表来存储评论内容，更新评论的函数如下：

```
def comment(self, dish_id, content):
    self.update_dish(dish_id, 'com', content)
```

7.3 具体实现效果

用户评论与回复

8. 功能 8

8.1 功能描述

在主界面，我们设计了搜索小组件，当我们输入关键词时即可弹出搜索框，罗列出搜索的结果。

8.2 实现逻辑与核心代码

对于菜品的搜索，我们提供了两种方式，一种是根据菜品名称搜索，另一种是使用描述菜品的形容词搜索。在具体实现中，面对一个搜索输入，我们会先按照菜品名称进行搜索，然后按照形容词搜索。

按照名称搜索时，我们会判断输入是否为某一菜品名称或者柜台或大厅名称的子串。按照形容词搜索时，我们意识到诸如“温暖”、“香甜”等形容词都可以映射

到冷热和酸甜苦辣咸等形容词上，所以我们枚举了几十个常见的菜品形容词，并把它们映射到冷热和酸甜苦辣咸这些存储在菜品表的属性中，在搜索时，我们会依据这些映射后的属性在菜品表中进行查找、而当一个输入通过以上两种方式都没有搜索结果时，我们会返回当前热门的菜（评价标准在后面讲述）作为搜索结果，这一点和美团类似。

相关函数实现如下：

```
def search_by_name(name, dishes):
    return [dish[0] for dish in dishes if name in dish[1]]


def search_by_adj(adj, dishes, mapping: dict):
    k = mapping[adj]
    fits = []
    for dish in dishes:
        fixed_attr = dish[4] | (dish[3] << 5) | (dish[2] << 7)
        if fixed_attr & k > 0:
            fits.append(dish[0])
    return fits
def search(self, k):
    d = self.execute('select * from dishes;')
    temp = search_by_name(k, d)
    if len(temp) == 0 and k in self.mapping:
```

```
        temp = search_by_adj(k, d, self.mapping)
    if len(temp) == 0:
        return [i[0] for i in self.get_popularity(50)]
    return temp
```

具体界面的实现，仍旧调用的是历史搜索板块的搜索界面，没有其他差异。

9. 功能 9

9.1 功能描述

在首页我们结合时间、用户口味偏好和以往用餐记录设置了用户必吃榜，热门榜单和个性化推荐界面，会依据不同的推荐算法对用户进行膳食推荐，点击每一个榜单内的项，可以呈现菜品的详情页面，同时，这三个推荐模块会以固定的周期进行更新，以保证最具时效性的推荐。同时，我们设计了两个小游戏，用作提升整个产品的趣味性。

9.2 实现逻辑与核心代码

在首页上我们设计了三个推荐模块，分别采用不同的算法进行推荐。

对于必吃榜，我们从被吃次数和收藏次数两个方面统计菜品的权重并进行综合。在收藏次数统计中，我们使用单个菜品收藏次数减去所有菜品收藏次数的平均数再除以总收藏次数来得到菜品的收藏权重，对于已吃权重也是如此。之后由于这两个权重都通过除以对应类型总记录数的方式进行了标准化，因此可以把它们直接相加获得菜品的综合权重。主要代码实现如下：

```
def get_recommendation(ate, fav, dishes):
    eaten_weight = get_most_eaten(ate, dishes)
    fav_weight = get_most_fav(fav, dishes)
```

```
    ew_sum = sum(eaten_weight.values())
    ew_avg = ew_sum / len(eaten_weight)
    fw_sum = sum(fav_weight.values())
    fw_avg = fw_sum / len(fav_weight)
    cnt = dict()
    for dish_id, ew in eaten_weight.items():
        cnt[dish_id] = (ew - ew_avg) / (ew_sum + 0.00001)
+ (fav_weight[dish_id] - fw_avg) / (fw_sum + 0.00001)
    info = [(dish_id, times) for dish_id, times in cnt.items()]
    info.sort(key=lambda x: x[1], reverse=True)
    return [i[0] for i in info]
```

对于热榜，我们依据菜品被吃记录进行推进，具体而言我们会获得当前时间和每一条记录中菜品被吃时间，并为每个菜品记录一个“热度”作为推荐权重。

我们定义一个超参数衰减率α为 0.05，并记菜品被吃记录时间点到当前时间的时间差为Δt，则$\alpha^{\Delta t}$为一次菜品被吃记录的权重，可以看出该权重值是随时间差变长而减小的，符合“热门的定义”。而每一个菜品的热度就是它的多有被吃记录的权重和。主要代码实现如下：

```
def get_popularity(self, num):
    ids = [dish[0] for dish in self.execute('select * from
dishes;')]
    records = [i[1:] for i in self.execute('select * from ates;')]
    popularity_dict = dict()
```

```
    for id in ids:
        popularity_dict[id] = [0, 0]
    current_time = time.time()
    decay_factor = 0.05
    for dish_id, dining_time in records:
        dining_time = dining_time.replace('\n', ' ')
        popularity_dict[dish_id][0] += 1
        dining_timestamp =
int(time.mktime(time.strptime(dining_time,
"%Y-%m-%d %H:%M:%S")))
        time_difference = current_time - dining_timestamp
        time_weight = pow(decay_factor, time_difference /
1000)
        popularity_dict[dish_id][1] += time_weight
    popularity_result = {}
    for dish_id, (dining_count, time_weight_sum) in
popularity_dict.items():
        popularity_score = dining_count *
time_weight_sum
        popularity_result[dish_id] = popularity_score
    return sorted(popularity_result.items(), key=lambda x:
x[1], reverse=True)[:num]
```

此外，我们还有针对每个人的个性化推荐，这个个性化推荐会结合基于内容推荐和协同过滤推荐生成 9 个推荐菜品。基于内容推荐是根据菜品的酸甜苦辣咸五种口味和冷热属性，为每个菜品构建特征向量。将口味和属性映射为数值，形成一个特征向量。然后，对于每个用户，将他们吃过的菜品的特征向量进行平均，得到用户的偏好向量。在推荐时，根据用户的偏好向量和其他菜品的特征向量之间的余弦相似度，推荐与用户偏好相似的菜品。

协同过滤推荐是基于拥有相似口味偏好的用户可能也会喜欢其他人吃过的菜，即使这个菜的口味与他的偏好不相近这个假设。我们对于每个用户，根据他们的吃过的菜品，计算用户之间余弦的相似度。然后，对于每个用户，找到与其相似度较高的其他用户，并推荐这些用户喜欢的菜品中，用户没有吃过的菜品。

综合来看，我们通过协同过滤方法向用户推荐 3 个菜，通过基于内容方法向用户推荐 6 个菜，实现综合推荐。具体的函数实现如下：

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
def content_based_recommendation(self, weight, num):
    time_flag = self.get_time_need()
    similarity_scores = {}
    for dish in self.execute('select * from dishes;'):
        taste = dish[4]
        heat = dish[3]
        tp = dish[2]
        if tp & time_flag == 0:
```

```
                similarity_scores[dish[0]] = 0
                continue
            temp = np.array(
                [(heat >> 1) & 1, heat & 1, (taste >> 4) & 1,
(taste >> 3) & 1, (taste >> 2) & 1, (taste >> 1) & 1,
                 taste & 1])
            similarity_scores[dish[0]] =
cosine_similarity([weight], [temp])[0][0]
    recommended_dishes =
sorted(similarity_scores.items(), key=lambda x: x[1],
reverse=True)
    return [i for i, _ in recommended_dishes][:num]

def collaborative_filtering_recommendation(self, weight,
tar_name):
    time_flag = self.get_time_need()
    similar_person = {}
    ates = [i[0] for i in self.get_ates(tar_name)]
    names = [person[0] for person in self.execute('select
name from people;')]
    for name in names:
        if name != tar_name:
```

```
            similar_person[name] =
cosine_similarity([weight],
[self.get_person_weight(name)])[0][0]
    recommend_people = [i[0] for i in
sorted(similar_person.items(), key=lambda x: x[1],
reverse=False)]
    res = []
    for name in recommend_people:
        other_ates = [i[0] for i in self.get_ates(name)]
        for dish_id in other_ates:
            tp = self.get_dish(dish_id)[2]
            if len(res) > 2:
                break
            if dish_id not in ates and dish_id not in res and
(time_flag & tp != 0):
                res.append(dish_id)
    return res

def personalized_recommendation(self, name):
    weight = self.get_person_weight(name)
```

```
    rec1 =
self.collaborative_filtering_recommendation(weight,
name)
    rec2 = self.content_based_recommendation(weight,
100)
    for dish_id in rec2:
        if len(rec1) >= 9:
            break
        if dish_id not in rec1:
            rec1.append(dish_id)
    return rec1
```

对于三个榜单，我们都设置了点击条目就可以呈现菜品的详情界面，这个是和条目的选择函数所绑定的，只需要创建一个菜品详情界面的对象并展示接口。在刷新方面，我们对于三个表的不同特点进行不同频率的刷新，我们使用多线程实现了不同周期的刷新，通过不同的时间间隔发送不同的信号，给三个表的更新发指令。对于个性化推荐榜单，我们实现了图片循环滚动展示的效果，我们设置了一个 PopUpAniStackedWidget 作为基底，设置三页，每页设置三个菜的呈现小窗口，通过多线程的信号发送进行当前展示页面的变化，就实现了个性化推荐页面的滚动展示效果。这里的线程设置如下：

```
class BackendThread(QObject):
    # 滚动 roll 效果
    update_date = pyqtSignal(int)
```

```python
# 更新必吃榜 5min 更新一次
update_must_list = pyqtSignal()
# 更新滚动推荐 30min 更新一次
update_roll = pyqtSignal()
# 更新热门榜单 1h 更新一次
update_hot_list = pyqtSignal()

# 处理业务逻辑
def run(self):
    timer = 0
    while 1:
        # 刷新展示页面 1-3 页
        for i in range(3):
            self.update_date.emit(i)
            time.sleep(5)
            timer += 1
            if (timer % 60) == 0:
                self.update_must_list.emit()
            if (timer % 360) == 0:
                self.update_roll.emit()
            if (timer % 720) == 0:
                self.update_hot_list.emit()
```

```
timer = 0
```

在小游戏方面，第一个是经典的俄罗斯方框，第二个是我们设计的”今天吃什么“，通过点击按钮，小游戏会随机挑选出一道菜供你选择，增加了菜品选择的丰富性。

四、项目运行过程

1. 安装依赖

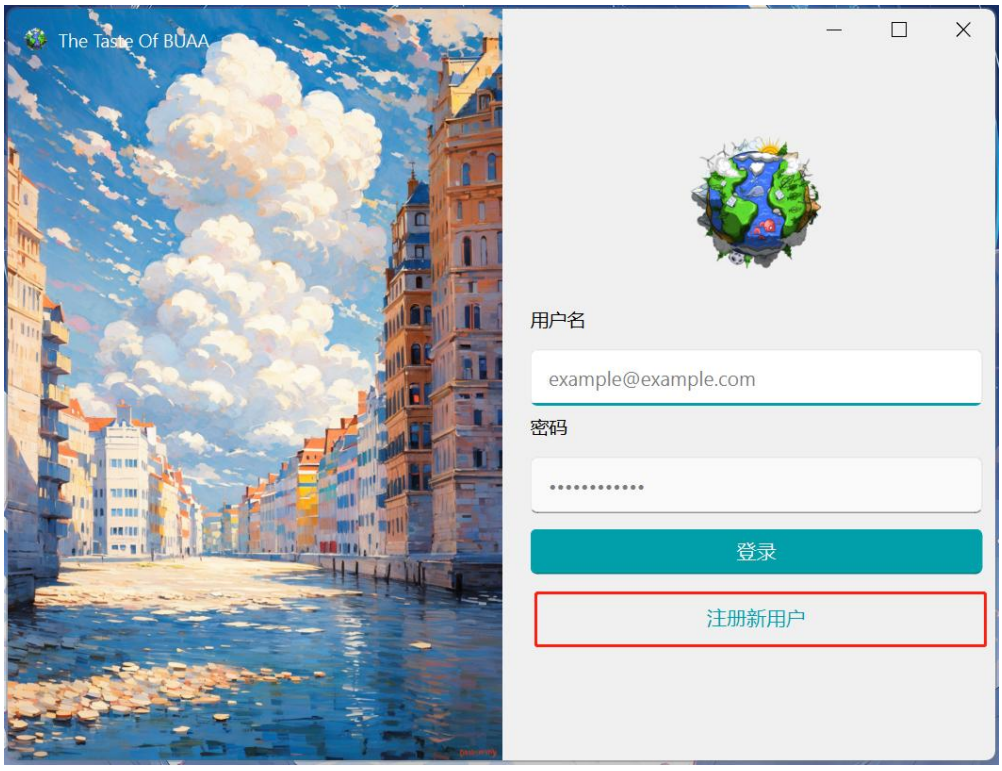
在**项目目录**下打开终端，执行:

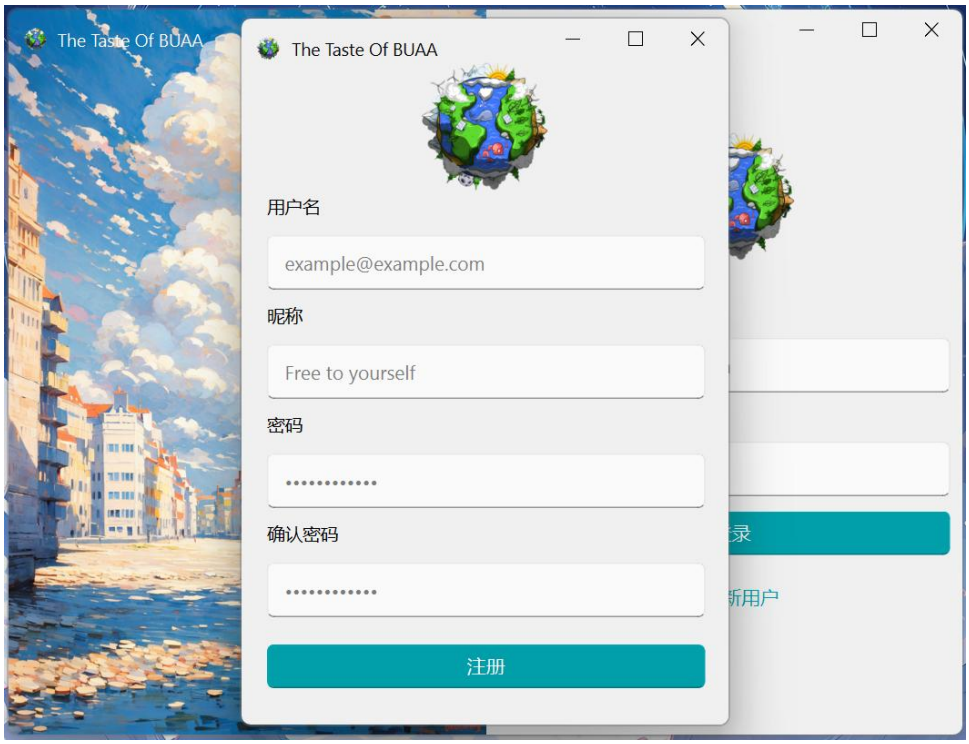
```
pip install -r requirements.txt
```

以此安装项目所需要的依赖。

2. 用户注册

运行 main.py，在登陆界面点击**注册新用户**进行注册，在注册界面进行**注册**。

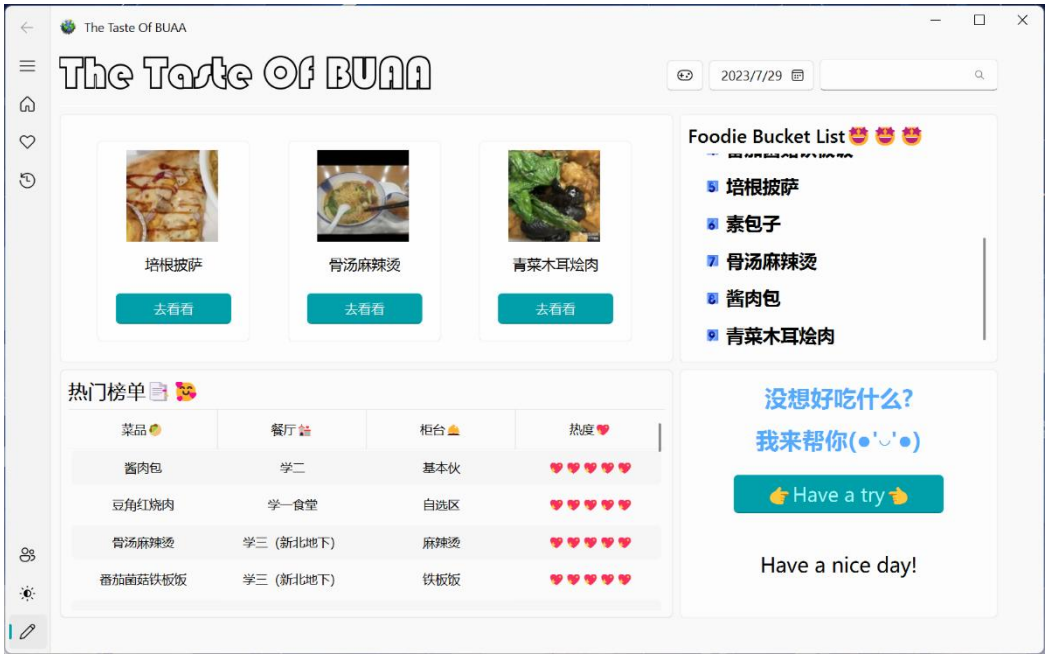




这里提供了注册好的普通用户和管理员用户：

普通用户	管理员用户
用户名：user_X	用户名：manager
密码：123456789	密码：buaa

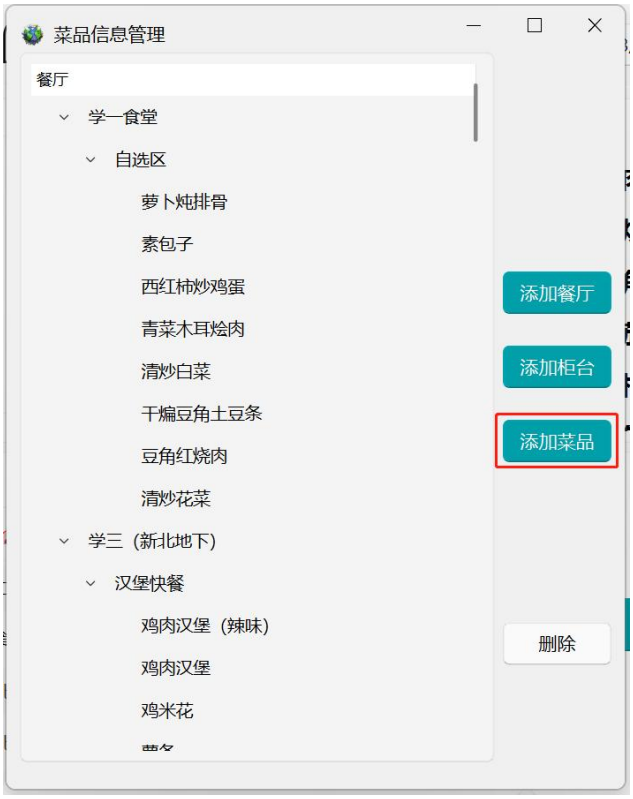
输入账号密码，登录后即可进入主界面。



3. 菜品上传

以管理员身份登录进入系统，点击**左下角编辑样式**按钮，进入菜品管理页面。

点击**添加菜品**，会弹出菜品添加的页面。



输入菜品相关信息，点击界面右上角的图片可以上传菜品图片，最后**点击添加**即可完成菜品上传。

五、项目总结

总结来说，我们的项目是一个前后端结合项目，前端负责呈现数据并与用户交互，后端负责对数据进行增删改查和基于此的统计与推荐。

本项目中前端部分的主要工作是搭建各个界面的 UI，同时设置后端数据的呈现接口。我们使用了 Qtdesigner 作为趁手的工具搭建界面，相较于纯手工写代码使得前端的界面更加地美观。同时在搭建过程中和后端积极沟通，实现了接口的统一，为之后的合并减少了很大的工作量。除此以外，在本项目的许多样式呈现我们都创新性地提出了简便但有效的数据结构，在菜单的树形结构中，我们设计了字典嵌套字典的结构，在用户评论的结构中，我们设计了字典列表的结构，这两处都体现了数据的降维，将有层次的数据转化为可以以字符串呈现的形式，为后端的存储和设计也提供了很大的便利。

本项目中后端部分的代码编写的追求是具体实现简单，接口提供丰富。对于前一方面，我们对操作数据库各个表所需要的功能进行了抽象提炼和简化，设计了很多低耦合的函数，也减少了代码量，比如提供通用的函数修改表中某一字段，或者把与数据库交互的语句进行包装复用。 这些对于每个表操作的对象和函数分别在与表名相同的 dishes.py, ates.py, fav.py...几个文件中。而对于后一方面，为了屏蔽掉底层数据结构的复杂性对前端工作的影响，我们在 database.py 中设立的 DBOperator 对象来提供针对数据的所有函数，包括增删菜品、记录已吃、提供推荐等等。 里面丰富的函数对原来精炼的实现进行了包装并配以符合直觉的函数名和参数类型，为前端实现丰富的功能提供有力的支持，这也使得在前后端分别编写完成的情况下，我们很快的就完成了代码的合并，基本符合我们的预期。

六、课程学习总结

1、我们小组四位成员在之前均只了解过一点 python 对于其语法也没有特别深入、系统的了解，大作业所涉及的内容也基本完全没有了解。在写大作业的过程中，前后端是分开由不同同学来做的。

前端的难点在在于 UI 的搭建，原有的 PyQt 的美观性设置难以进行，因此就不得不使用拓展的 PyQt 包进行编辑，但是由于非官方库对于各个组件和函数的功能的描述不太细致，只能通过不断尝试加看示例的方式进行制作，同时由于有些拓展的函数会和原有的函数发生冲突，或者已经被原库删除，会导致一些奇怪的 bug，因此对于前端的 debug 也是一大难点。

后端部分的难点是数据库中表的设计以及推荐算法。我们使用的是 mysql 关系型数据库，而代码本身的逻辑是按照对象存储数据的，把这样的数据逻辑映射到表中确实有难度。而推荐算法则需要用到统计学的知识，我们在查阅足够资料以后才能给出较为合理的推荐方式。

2、李老师讲课讲的很清楚也很细致，四节课认真听下来对于 python 零基础的同学感觉收获还是非常的大的，不仅掌握了 python 基础语法、达到会写简单的 python 程序的程度，还了解了语言背后的原理。建议和希望介绍的内容见后面第四部分。

3、几位助教在给我们答疑时都非常认真。在做平时练习和完成大作业时遇到了一些困难，助教也很耐心地帮我们解决了。

4、本课程针对的主要都是 6 系大二的学生，应该已经有一定的编程基础，很多基本的语法、基础知识以及和其他语言相差不大的地方应该都已经比较熟悉了，其实这些内容让同学们通过每次课下作业及课上测试去自学大概就可以较好

的掌握了。同时，感觉授课内容及平时的课下作业与最后的大作业也有些脱节，大作业用到的绝大部分知识，像是针对数据库的操作或者是前端界面的编写都需要同学们去自学。我们认为老师可能可以在课上把基础语法知识的部分快速带过，节省下的时间用来讲一讲大作业有关的知识或者编写这种大型项目的基本方法和大致框架，让同学们在第一次自己动手时有些头绪。

七、主要参考资料

PyQt 中文教程

白月黑羽

PyQt 之旅

PyQt-Fluent-Widgets

PyQt 从零开始

阿里巴巴矢量图标库

pyqt 显示图片的两种方法

PyQt5 实时刷新数据

MySQL Documents

八、项目功能实际展示视频（不超过 5min）