

# 银行管理系统

---

## 系统目标

实现一个银行管理系统，涉及银行信息、客户信息、账户信息、贷款信息、银行部门信息、员工信息相关实体。

选择采用B/S架构，实现语言及web框架是Python+Flask，后端数据库是MySQL

仓库地址：[https://github.com/baqiyansheng/Bank\\_system.git](https://github.com/baqiyansheng/Bank_system.git)

## 需求分析

### 实体属性需求

登录账号信息：登录银行管理系统需要账号和密码。账号作为主键。

银行信息：银行存在多个支行，每个支行由Bank\_ID唯一确定，ID的格式为Bxxx。每个支行有各自的名称、地址、联系电话、营业时间。

银行部门信息：每个银行都存在自己的部门。部门由Department\_ID唯一确定，ID格式为Dxxx。每个部门有自己的名称，电话，部门经理。

员工信息：员工包括普通员工和部门经理。员工由Staff\_ID唯一确定，ID格式为Sxxx。每个员工有自己的姓名，职位(普通员工/部门经理)，住址，联系方式，照片(文件路径)。

客户信息：客户由Customer\_ID唯一确定，ID格式为Cxxx。每个客户有自己的姓名，电话，地址。

账户信息：账户由Account\_ID唯一确定，ID格式为Axxx。其属性包括开户日期，账户余额

记录信息：流水记录账户的收入支出信息。由Record\_ID唯一确定，ID格式为Rxxx。其属性包括：交易时间，净增值，交易类型，交易备注。

贷款信息：贷款由Loan\_ID唯一确定，ID格式为Lxxx。其属性包括剩余未还金额，贷款期限，贷款状态(是否已经还清)

### 系统功能需求

系统登录：登录账号及注册账号。登录成功后进入主页，进行后续的系统操作。系统分为管理端和客户端，当通过管理员账号admin登录时，进入管理员的页面，否则进入客户端页面。只能注册客户端账号，并且注册时会记录客户的信息(姓名等)，将账号作为客户的ID。

银行管理：管理端可以增加，删除，修改，查询支行信息。客户只能查询支行信息。

银行部门管理：管理端可以增加、删除、修改、查询部门信息。同时需要显示经理的ID、姓名和头像。通过指定经理ID可以选择任命谁当经理。客户端不能进行部门管理。

员工管理：管理端可以增加、删除、修改、查询员工信息。同时需要显示员工的头像。当在部门管理页面更换经理后，员工管理页面也会发生相应的变化。客户端不能进行员工管理。

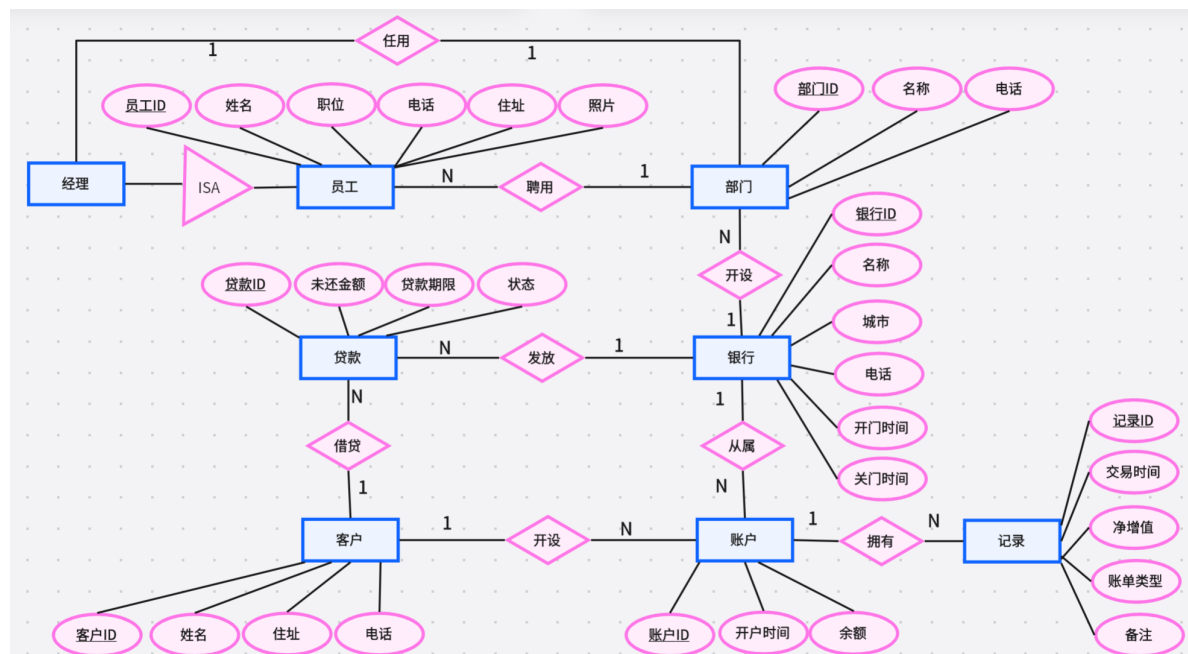
客户信息：管理端可以增加、删除、修改、查询客户信息。用户端只能删除、修改、查询自己的信息。当管理端添加客户时，同时创建登录账号。当删除客户信息时，同时删除登录账号。

账户信息：管理端和客户端都可以增加、删除、查询账户信息。客户端孩子只能对自己的账户做操作，也只能查询自己的账户。

记录信息：管理端不可见，只有客户端可见。客户只能看到自己名下的账户的记录。也可以发起事务，存款，取款，转账，并记录信息。除非发起事务，不然不能添加记录，记录不可修改，可以删除，可以查询。

贷款信息：管理端和客户端可以增加删除查询贷款信息。客户端只可见自己名下的贷款信息。贷款信息不能手动修改。客户端还有还贷功能。

## ER图



## 逻辑设计

根据ER图可以设计如下的关系数据库模式：

- 支行(银行ID, 名称, 地址, 电话, 开门时间, 关门时间)
- 部门(部门ID, 名称, 电话)
- 员工(员工ID, 姓名, 职位, 电话, 住址)
- 客户(客户ID, 姓名, 电话, 住址)
- 账户(账户ID, 开户时间, 余额)
- 记录(记录ID, 交易时间, 净增值, 账单类型, 备注)
- 贷款(贷款ID, 未还金额, 贷款期限, 贷款状态)

结合关系，得到如下的关系数据库模式：

- 支行(银行ID, 名称, 地址, 电话, 开门时间, 关门时间)
- 部门(部门ID, 所属银行ID, 名称, 电话)
- 员工(员工ID, 姓名, 职位, 所属部门ID, 电话, 住址)
- 客户(客户ID, 姓名, 电话, 住址)
- 账户(账户ID, 所属客户ID, 所属银行ID, 开户时间, 余额)
- 记录(记录ID, 所属账户ID, 交易时间, 净增值, 账单类型, 备注)
- 贷款(贷款ID, 所属客户ID, 所属银行ID, 未还金额, 贷款期限, 贷款状态)

在每个关系模式中，非主属性都是完全且非传递的依赖于主码，因此满足该关系数据库模式已经满足3NF，不需要进一步模式分解

# 数据库实现

## 登录账户

```
CREATE Table login (  
    username VARCHAR(20) PRIMARY KEY,  
    password VARCHAR(20) NOT NULL  
);  
# 管理员账号  
INSERT INTO login (username, password) VALUES ('admin', 'admin');  
INSERT INTO login (username, password) VALUES ('c001', '123456');  
INSERT INTO login (username, password) VALUES ('c002', '123456');
```

登录账户只需要两个属性，用户名和密码。用户名是主键，密码是非空的。插入一个管理员账号和两个客户账号，方便后面使用。

## 支行信息

```
CREATE TABLE bank (  
    Bank_ID VARCHAR(20) PRIMARY KEY,  
    name VARCHAR(20) NOT NULL,  
    city VARCHAR(20),  
    phone VARCHAR(20),  
    open_time TIME DEFAULT '09:00:00',  
    close_time TIME DEFAULT '17:00:00'  
);  
INSERT INTO bank (Bank_ID, name, city, phone, open_time, close_time)  
VALUES ('B001', '黄山路支行', '安徽合肥', '1234567890', '08:00:00', '18:00:00');  
INSERT INTO bank (Bank_ID, name, city, phone, open_time, close_time)  
VALUES ('B002', '肥西路支行', '安徽合肥', '9876543210', '09:00:00', '17:00:00');
```

支行ID作为主键，支行名是非空的，开门时间关门时间默认为朝九晚五。

## 部门信息

```
CREATE TABLE department (  
    Department_ID VARCHAR(20) PRIMARY KEY,  
    name VARCHAR(20) NOT NULL,  
    Bank_ID VARCHAR(20) NOT NULL,  
    phone VARCHAR(20) ,  
    FOREIGN KEY (Bank_ID) REFERENCES bank (Bank_ID)  
);  
INSERT INTO department (Department_ID, name, Bank_ID, phone)  
VALUES ('D001', '柜员部', 'B001', '1313131313');  
INSERT INTO department ( Department_ID, name, Bank_ID, phone )  
VALUES ('D002', '接待部', 'B001', '1414141414');  
INSERT INTO department (Department_ID, name, Bank_ID, phone)  
VALUES ('D003', '柜员部', 'B002', '1515151515');  
INSERT INTO department ( Department_ID, name, Bank_ID, phone )  
VALUES ('D004', '接待部', 'B002', '1616161616');
```

部门ID是主键，部门名称和所属银行ID非空，银行ID作为外键。

## 员工信息

```
CREATE TABLE stuff (  
    Stuff_ID VARCHAR(20) PRIMARY KEY,  
    name VARCHAR(20) NOT NULL,  
    Department_ID VARCHAR(20) NOT NULL,  
    work VARCHAR(20) NOT NULL DEFAULT '员工',  
    phone VARCHAR(20),  
    address VARCHAR(50),  
    figure VARCHAR(50) NOT NULL DEFAULT 'default.jpg',  
    FOREIGN KEY (Department_ID) REFERENCES department (Department_ID)  
);  
  
INSERT INTO stuff (Stuff_ID, name, Department_ID, work, phone, address, figure)  
VALUES ('S001', '钢铁侠', 'D001', '经理', '13000000000', '浙江杭州', '钢铁侠.jpg');  
INSERT INTO stuff (Stuff_ID, name, Department_ID, work, phone, address, figure)  
VALUES ('S002', '美国队长', 'D002', '员工', '13100000000', '上海', '美国队长.jpg');  
INSERT INTO stuff (Stuff_ID, name, Department_ID, work, phone, address, figure)  
VALUES ('S003', '浩克', 'D003', '员工', '13200000000', '北京', '浩克.jpg');  
INSERT INTO stuff (Stuff_ID, name, Department_ID, work, phone, address, figure)  
VALUES ('S004', '死侍', 'D004', '经理', '13300000000', '江苏南京', '死侍.jpg');
```

员工ID作为主键，姓名、所属部门ID，职位，图像都非空，职位默认为员工，图像默认为default图像的名称。所属部门ID作为外键

## 客户信息

```
CREATE TABLE customer (  
    Customer_ID VARCHAR(20) PRIMARY KEY,  
    name VARCHAR(20) NOT NULL,  
    phone VARCHAR(20),  
    address VARCHAR(50)  
);  
  
INSERT INTO customer (Customer_ID, name, phone, address)  
VALUES ('C001', '张三', '1300000789', '安徽合肥');  
INSERT INTO customer (Customer_ID, name, phone, address)  
VALUES ('C002', '李四', '1310000789', '湖南长沙');
```

客户ID作为主键，姓名非空。

## 账户信息

```
CREATE TABLE account (  
    Account_ID VARCHAR(20) PRIMARY KEY,  
    Customer_ID VARCHAR(20) NOT NULL,  
    Bank_ID VARCHAR(20) NOT NULL,  
    create_time DATE NOT NULL,  
    Balance DECIMAL(18, 2) DEFAULT 0,  
    FOREIGN KEY (Customer_ID) REFERENCES customer (Customer_ID),  
    FOREIGN KEY (Bank_ID) REFERENCES bank (Bank_ID)  
);  
  
INSERT INTO account (Account_ID, Customer_ID, Bank_ID, create_time, Balance)  
VALUES ('A001', 'C001', 'B001', '2020-01-01', 10000);  
INSERT INTO account (Account_ID, Customer_ID, Bank_ID, create_time, Balance)
```

```
VALUES ('A002', 'C001', 'B002', '2021-01-01', 100000);
INSERT INTO account (Account_ID, Customer_ID, Bank_ID, create_time, Balance)
VALUES ('A003', 'C002', 'B001', '2020-06-01', 5000);
INSERT INTO account (Account_ID, Customer_ID, Bank_ID, create_time, Balance)
VALUES ('A004', 'C002', 'B002', '2021-11-11', 500);
```

账户ID作为主键，客户ID，银行ID为外键，所有属性都非空，余额默认为0。余额的属性设置为decimal(18,2)，代表着最多18，其中2位是小数的十进制数。之所以不使用float是因为float容易出现精度问题。而银行余额这种需要精度比较高的数据需要用decimal

## 记录信息

```
CREATE TABLE record (
    Record_ID VARCHAR(20) PRIMARY KEY,
    Account_ID VARCHAR(20) NOT NULL,
    time DATE NOT NULL,
    increasement DECIMAL(18, 2) NOT NULL,
    type VARCHAR(20),
    detail VARCHAR(50),
    FOREIGN KEY (Account_ID) REFERENCES account (Account_ID)
);
INSERT INTO record (Record_ID, Account_ID, time, increasement, type, detail)
VALUES ('R001', 'A001', '2020-01-01', 1000, '收入', '存钱');
INSERT INTO record (Record_ID, Account_ID, time, increasement, type, detail)
VALUES ('R002', 'A001', '2020-01-02', -500, '支出', '取钱');
INSERT INTO record (Record_ID, Account_ID, time, increasement, type, detail)
VALUES ('R003', 'A001', '2021-01-03', -500, '支出', '向A002转账');
INSERT INTO record (Record_ID, Account_ID, time, increasement, type, detail)
VALUES ('R004', 'A001', '2021-01-04', 1000, '收入', 'A002的转账');
```

记录ID作为主键，账户ID、银行ID作为外键，净增值非空。

## 贷款信息

```
REATE TABLE loan (
    Loan_ID VARCHAR(20) PRIMARY KEY,
    Customer_ID VARCHAR(20) NOT NULL,
    Bank_ID VARCHAR(20) NOT NULL,
    RemainingAmount DECIMAL(18, 2) NOT NULL,
    term DATE NOT NULL,
    status INT NOT NULL DEFAULT 0, # 0表示未还清,1表示已还清
    FOREIGN KEY (Customer_ID) REFERENCES customer (Customer_ID),
    FOREIGN KEY (Bank_ID) REFERENCES bank (Bank_ID)
);
INSERT INTO loan (Loan_ID, Customer_ID, Bank_ID, RemainingAmount, term, status)
VALUES ('L001', 'C001', 'B001', 10000, '2022-01-01', 1);
INSERT INTO loan (Loan_ID, Customer_ID, Bank_ID, RemainingAmount, term, status)
VALUES ('L002', 'C001', 'B002', 100000, '2023-01-01', 0);
```

贷款ID作为主键，客户ID和银行ID作为外键。剩余未还金额、贷款期限、贷款状态非空。

## 数据库模型

结合在MySQL中创建的表的信息，使用Flask-SQLAlchemy扩展，定义了多个数据库模型，也相当于实体的类。不做过多解释。

其中\_\_tablename\_\_表示对应的表名

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

# 用户管理
class User(db.Model):
    __tablename__ = 'login'

    username = db.Column(db.String(15, 'utf8mb4_0900_ai_ci'), primary_key=True)
    password = db.Column(db.String(50), nullable=False)

# 支行
class Bank(db.Model):
    __tablename__ = 'bank'

    Bank_ID = db.Column(db.String(20), primary_key=True)
    name = db.Column(db.String(20), nullable=False)
    city = db.Column(db.String(20))
    phone = db.Column(db.String(20))
    open_time = db.Column(db.Time, default='09:00:00')
    close_time = db.Column(db.Time, default='17:00:00')

# 部门
class Department(db.Model):
    __tablename__ = 'department'

    Department_ID = db.Column(db.String(20), primary_key=True)
    name = db.Column(db.String(20), nullable=False)
    Bank_ID = db.Column(db.String(20),
                        db.ForeignKey('bank.Bank_ID'),
                        nullable=False)
    phone = db.Column(db.String(20))
    bank = db.relationship('Bank', backref=db.backref('departments'))

# 员工
class Stuff(db.Model):
    __tablename__ = 'stuff'

    Stuff_ID = db.Column(db.String(20), primary_key=True)
    name = db.Column(db.String(20), nullable=False)
    Department_ID = db.Column(db.String(20),
                              db.ForeignKey('department.Department_ID'),
                              nullable=False)
    work = db.Column(db.String(20), nullable=False, default='员工')
```

```
phone = db.Column(db.String(20))
address = db.Column(db.String(50))
figure = db.Column(db.String(50), nullable=False, default='default.jpg')
department = db.relationship('Department', backref=db.backref('stuffs'))
```

## # 客户

```
class Customer(db.Model):
    __tablename__ = 'customer'

    Customer_ID = db.Column(db.String(20), primary_key=True)
    name = db.Column(db.String(20))
    phone = db.Column(db.String(20))
    address = db.Column(db.String(50))
```

## # 账户

```
class Account(db.Model):
    __tablename__ = 'account'

    Account_ID = db.Column(db.String(20), primary_key=True)
    Customer_ID = db.Column(db.String(20),
                             db.ForeignKey('customer.Customer_ID'),
                             nullable=False)
    Bank_ID = db.Column(db.String(20),
                         db.ForeignKey('bank.Bank_ID'),
                         nullable=False)
    create_time = db.Column(db.Date, nullable=False)
    Balance = db.Column(db.DECIMAL(18, 2), default=0)
    customer = db.relationship('Customer', backref=db.backref('accounts'))
    bank = db.relationship('Bank', backref=db.backref('accounts'))
```

## # 交易记录

```
class Record(db.Model):
    __tablename__ = 'record'

    Record_ID = db.Column(db.String(20), primary_key=True)
    Account_ID = db.Column(db.String(20),
                           db.ForeignKey('account.Account_ID'),
                           nullable=False)
    time = db.Column(db.Date, nullable=False)
    increasement = db.Column(db.DECIMAL(18, 2), nullable=False)
    type = db.Column(db.String(50), nullable=False)
    detail = db.Column(db.String(50))
    account = db.relationship('Account', backref=db.backref('records'))
```

## # 贷款

```
class Loan(db.Model):  
    __tablename__ = 'loan'  
  
    Loan_ID = db.Column(db.String(20), primary_key=True)  
    Customer_ID = db.Column(db.String(20),  
                             db.ForeignKey('customer.Customer_ID'),
```

```

        nullable=False)
    Bank_ID = db.Column(db.String(20),
                        db.ForeignKey('bank.Bank_ID'),
                        nullable=False)
    RemainingAmount = db.Column(db.DECIMAL(18, 2), nullable=False)
    term = db.Column(db.Date, nullable=False)
    status = db.Column(db.Integer, nullable=False, default=0)
    customer = db.relationship('Customer', backref=db.backref('loans'))
    bank = db.relationship('Bank', backref=db.backref('loans'))

```

```
class DepartmentManagerView:
```

```

    def __init__(self, department, stuff):
        self.department_id = department.Department_ID
        self.department_name = department.name
        self.bank_id = department.Bank_ID
        self.phone = department.phone
        if (stuff):
            self.manager_id = stuff.Stuff_ID
            self.manager_name = stuff.name
            self.manager_photo = stuff.figure
        else:
            self.manager_photo = 'default.jpg'

```

增加了一个类，用于显示部门及经理的信息。包含部门的所有信息以及经理的ID，姓名和照片。

## 网页的实现

### 连接数据库及创建实例

```

app = Flask(__name__)
app.config.from_object(config)
db = SQLAlchemy(app)

```

先创建Flask应用实例，然后从config文件中加载配置，最后创建了一个 SQLAlchemy 对象，并将 Flask 应用实例传递给它。这样，SQLAlchemy 就能在 Flask 应用中使用并连接到数据库。

```

SQLALCHEMY_DATABASE_URI =
'mysql+pymysql://root:2003523@localhost:3306/bank_system' #
root:password@localhost:3306/database

SQLALCHEMY_ECHO = True

```

上面是config文件，配置了数据库的相关信息。

### Flask和SQLAlchemy接口：

这一部分讲解后文经常使用到的一些接口，避免重复介绍。



## 从前端获取数据

后端通过 `request.form.get()` 获取表单的对应数据，然后存储在对应变量中方便后面的使用。

事务的设置：事务的设置统一为以下格式，后面的代码里不再赘述：

对数据库的所有增加、删除、修改功能都设置为事务，以保持一致性。

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/login', methods=['GET', 'POST'])
def login():
    global authorized
    global cur_ID
    if request.method == 'GET':
        return render_template('login.html', authorized=authorized)
    else:
        if request.form.get('type') == 'signup':
            try:
                with db.session.begin():
                    ID = request.form.get('ID')
                    key = request.form.get('password')
                    confirm = request.form.get('password2')
                    name = request.form.get('name')
                    phone = request.form.get('phone')
```

```

city = request.form.get('city')
UserNotExist = db.session.query(User).filter_by(
    username=ID).scalar() is None
if not UserNotExist:
    return jsonify({
        'success': False,
        'error_message': '用户名已存在'
    })
if key != confirm:
    return jsonify({
        'success': False,
        'error_message': '两次密码不一致'
    })
newUser = User(
    username=ID,
    password=key,
)
db.session.add(newUser)
newCustomer = Customer(Customer_ID=ID, name=name)
if phone:
    newCustomer.phone = phone
if city:
    newCustomer.address = city
db.session.add(newCustomer)
db.session.commit()
return jsonify({'success': True})
except Exception as e:
    db.session.rollback()
    return jsonify({'success': False, 'error_message': str(e)})
elif request.form.get('type') == 'login':

    name = request.form.get('name')
    key = request.form.get('password')
    UserNotExist = db.session.query(User).filter_by(
        username=name).scalar() is None

    if UserNotExist == 1:
        error_title = '登录错误'
        error_message = '用户名不存在'
        return render_template('404.html',
                                error_title=error_title,
                                error_message=error_message)

    user_result = db.session.query(User).filter_by(
        username=name).first()
    if user_result.password == key:
        if name == 'admin':
            authorized = True
            cur_ID = name
        else:
            authorized = False
            cur_ID = name
        return render_template('index.html', authorized=authorized)
    else:
        error_title = '登录错误'

```

```
error_message = '密码错误'
return render_template('404.html',
                       error_title=error_title,
                       error_message=error_message)

return render_template('login.html', authorized=authorized)
```

`app.route()` 指定路由，接受 GET 和 POST 请求

`authorized` 和 `cur_ID` 作为全局变量，用于记录当前登录的是管理员还是用户，以及用户的ID。

对于 GET 请求：通过 `render_template()` 显示 `login.html`。

对于 POST 请求：如果是注册，先检查客户ID是否已经存在，如果存在则显示错误信息；再检查密码和确认密码是否一致；最后结合输入的账号密码信息创建账号，结合输入的客户信息创建客户。如果是登录，先检查账号是否存在，再检测密码是否正确，最后根据账号设置 `authorized` 和 `cur_ID`，然后显示主页 `index.html`

数据库系统及应用Lab

银行业务管理系统

Version 1.0

用户名: admin

密码: .....

注册 登录

江岩 PB21111727

这是登录界面，给出注册和登录两个按钮，输入的密码会被隐藏

数据库系统及应用Lab

银行业务管理系统

Version 1.0

注册

用户名(必填)

密码(必填)

确认密码(必填)

姓名(必填)

联系电话(选填)

居住城市(选填)

确认 取消

江岩 PB21111727

注册表单会指引需要填写的内容。如果不填写必填项，点击确认会提示填写必填项。

由于客户端的客户信息界面不能创建客户，因此在创建账号的同时会创建客户信息。

## 主页

```
@app.route('/index')
def index():
    return render_template('index.html', authorized=authorized)
```

根据 `authorized` 的值决定显示的是管理端还是客户端。

下面是管理端的页面显示：



下面是客户端的页面显示：



通过在html文件中使用if else可以由 `authorized` 决定要显示的内容。管理端没有账户明细查询，客户端没有部门管理和员工管理。且侧面菜单的名称也略有不同。

## 银行信息页面

```
# 支行管理
@app.route('/bank', methods=['GET', 'POST'])
def bank():
    labels = ['支行号', '支行名', '城市', '电话', '开放时间', '关闭时间']

    if request.method == 'GET':
        result_query = db.session.query(Bank)
        result = result_query.all()
        return render_template('bank.html',
                               labels=labels,
                               content=result,
                               authorized=authorized)

    else:
        if request.form.get('type') == 'query':
            bank_id = request.form.get('id')
            bank_name = request.form.get('name')
            bank_city = request.form.get('city')

            # 构建查询条件
            global filters
            filters = []
            if bank_id:
                filters.append(Bank.Bank_ID == bank_id)
            if bank_name:
                filters.append(Bank.name == bank_name)
            if bank_city:
                filters.append(Bank.city.like(f'%{bank_city}%'))
            result_query = db.session.query(Bank)
            # 应用查询条件
            if filters:
                result_query = result_query.filter(and_(*filters))
            result = result_query.all()

            return render_template('bank.html',
                                   labels=labels,
                                   content=result,
                                   authorized=authorized)

        elif request.form.get('type') == 'update':
            try:
                with db.session.begin():
                    old_id = request.form.get('key')
                    bank_name = request.form.get('bank_name')
                    bank_city = request.form.get('bank_city')
                    bank_phone = request.form.get('bank_phone')
                    bank_open_time = request.form.get('bank_open_time')
                    bank_close_time = request.form.get('bank_close_time')
                    # 创建一个更新语句
                    update_stmt = (update(Bank).where(
                        Bank.Bank_ID == old_id).values(
                            name=bank_name,
                            city=bank_city,
                            phone=bank_phone,
```

```

        open_time=bank_open_time,
        close_time=bank_close_time))

    # 执行更新操作
    db.session.execute(update_stmt)
    db.session.commit()
except Exception as e:
    db.session.rollback()
    return render_template('404.html',
                           error_title='支行信息更新错误',
                           error_message=str(e._message()))
elif request.form.get('type') == 'delete':
    try:
        with db.session.begin():
            bank_id = request.form.get('key')
            # 检查银行是否存在关联的账户
            if db.session.query(Account).filter(
                Account.Bank_ID == bank_id, Account.Balance
                != 0).first() is not None:
                error_title = '删除错误'
                error_message = '支行存在关联账户'
                return render_template('404.html',
                                       error_title=error_title,
                                       error_message=error_message)

            # 检查银行是否存在未还清的贷款
            if db.session.query(Loan).filter(
                Loan.Bank_ID == bank_id,
                Loan.status == 0).first() is not None:
                error_title = '删除错误'
                error_message = '支行存在未还清的贷款'
                return render_template('404.html',
                                       error_title=error_title,
                                       error_message=error_message)

            # 如果通过了所有检查，执行删除操作
            # 先删除银行下的账户
            account_result = db.session.query(Account).filter_by(
                Bank_ID=bank_id).all()
            for account in account_result:
                db.session.delete(account)
            # 再删除银行下的贷款
            loan_result = db.session.query(Loan).filter_by(
                Bank_ID=bank_id).all()
            for loan in loan_result:
                db.session.delete(loan)
            # 查询银行下的部门
            departments = db.session.query(Department).filter_by(
                Bank_ID=bank_id).subquery()
            # 删除部门下的员工
            db.session.query(Stuff).filter(
                Stuff.Department_ID.in_(departments)).delete()
            # 删除部门
            db.session.query(Department).filter_by(
                Bank_ID=bank_id).delete()

```

```

        # 最后删除银行
        bank_result = db.session.query(Bank).filter_by(
            Bank_ID=bank_id).first()
        db.session.delete(bank_result)
        db.session.commit()
    except Exception as e:
        db.session.rollback()
        return render_template('404.html',
                                error_title='支行信息删除错误',
                                error_message=str(e._message()))
elif request.form.get('type') == 'insert':
    try:
        with db.session.begin():
            bank_id = request.form.get('id')
            bank_name = request.form.get('name')
            bank_city = request.form.get('address')
            bank_phone = request.form.get('phone')
            bank_open_time = request.form.get('open_time')
            bank_close_time = request.form.get('close_time')

            # 检查每个属性是否存在, 如果存在则设置相应的值, 否则设置为 None
            new_bank = Bank()
            if bank_id:
                new_bank.Bank_ID = bank_id
            if bank_name:
                new_bank.name = bank_name
            if bank_city:
                new_bank.city = bank_city
            if bank_phone:
                new_bank.phone = bank_phone
            if bank_open_time:
                new_bank.open_time = bank_open_time
            if bank_close_time:
                new_bank.close_time = bank_close_time
            if db.session.query(Bank).filter_by(
                Bank_ID=bank_id).first():
                error_title = '插入错误'
                error_message = '支行编号重复'
                return render_template('404.html',
                                        error_title=error_title,
                                        error_message=error_message)

            db.session.add(new_bank)
            db.session.commit()
            filters = []
    except Exception as e:
        db.session.rollback()
        return render_template('404.html',
                                error_title='支行信息插入错误',
                                error_message=str(e._message()))

result_query = db.session.query(Bank)
if filters:
    result_query = result_query.filter(and_(*filters))
result = result_query.all()
return render_template('bank.html',
                        labels=labels,

```

```
content=result,  
authorized=authorized)
```

GET 请求直接返回bank表的所有查询结果，相当于 `SELECT * FROM bank`。

POST 请求的处理取决于 `type` 的值。即增删改查中的一个。

查询时要把查询条件 `filters` 设置为全局变量，以保证更新后显示相同的条目。当输入的查询条件非空时，将相应的查询条件加入到 `filters` 中，最后查询数据库。

更新要设置为事务，读入输入的数据，然后用update语句更新数据库。

删除要设置为事务，先检查是银行下是否存在账户且其余额不为0，然后银行下是否存在未还清的贷款。删除过程是依次删除银行下的账户、银行下的贷款、银行下部门的员工，银行下的部门，最后删除银行。

插入时要设置为事务，当输入的内容非空时，设置插入属性。然后检查银行编号是否重复，最后插入数据。

管理端的页面：

数据库系统及应用Lab

银行管理系统

首页

支行管理

部门管理

员工管理

客户管理

账户管理

贷款管理

支行ID

支行名称

所在城市

查询

添加

支行信息

支行ID	支行名称	所在城市	联系电话	开门时间	关门时间	操作
B001	黄山路支行	安徽合肥	1234567890	08:00:00	18:00:00	<div>更新</div> <div>删除</div>
B002	肥西路支行	安徽合肥	9876543210	09:00:00	17:00:00	<div>更新</div> <div>删除</div>

客户端只能查询支行信息，不能增加、删除、修改支行信息。

客户端的页面：



银行客户系统

首页

支行查询

个人信息

账户查询

账户明细查询

贷款查询

支行ID

支行名称

所在城市

查询

支行信息

支行ID	支行名称	所在城市	联系电话	开门时间	关门时间
B001	黄山路支行	安徽合肥	1234567890	08:00:00	18:00:00
B002	肥西路支行	安徽合肥	9876543210	09:00:00	17:00:00

后面的页面的处理比较类似，只展示比较特殊的处理的代码。

## 部门信息页面

```
def department_manager_view(departments):
    department_manager = []
    for department in departments:
        manager = db.session.query(Stuff).filter_by(
            Department_ID=department.Department_ID, work='经理').first()
        department_manager.append(DepartmentManagerView(department, manager))
    return department_manager
```

为了构建视图查询结果，输入查询得到的部门信息，然后从Stuff表里查询每个部门的经理的信息。

```
@app.route('/department', methods=['GET', 'POST'])
def department():
    .....
    elif request.form.get('type') == 'update':
        try:
            with db.session.begin():
                old_id = request.form.get('Department_ID')
                department_name = request.form.get('name')
                bank_id = request.form.get('Bank_ID')
                phone = request.form.get('phone')
                stuff_id = request.form.get('stuff_id')
                BankNotExist = db.session.query(Bank).filter_by(
                    Bank_ID=bank_id).scalar() is None
                .....
                # 创建一个更新部门信息
                update_stmt = (update(Department).where(
                    Department.Department_ID == old_id).values(
                        name=department_name, Bank_ID=bank_id,
                        phone=phone))
                db.session.execute(update_stmt)
                # 把曾经的经理降级为员工
                old_manager = db.session.query(Stuff).filter_by(
                    Department_ID=old_id, work='经理').first()
```

```

if old_manager:
    old_manager_id = old_manager.Stuff_ID
    update_stmt = (update(Stuff).where(
        Stuff.Stuff_ID == old_manager_id).values(
            work='员工'))
    db.session.execute(update_stmt)
# 任命新的经理
update_stmt = (update(Stuff).where(
    Stuff.Stuff_ID == stuff_id).values(
        Department_ID=old_id, work='经理'))
db.session.execute(update_stmt)
db.session.commit()
except Exception as e:
    db.session.rollback()
return render_template('404.html',
    error_title='部门信息修改错误',
    error_message=str(e._message()))

.....

```

插入的处理：检查部门编号是否重复，银行是否存在。

删除的处理：先删除部门下的员工，再删除部门。

更新的处理：检查银行是否存在，检查经理的员工信息是否存在。如果之前这个部门已经有经理了，需要将他职位变为普通员工。最后更新新的经理的信息，包括他的职位和所属部门。

管理端页面如下：





数据库系统及应用Lab

银行业务管理系统

- 首页
- 支行管理
- 部门管理
- 员工管理
- 客户管理
- 账户管理
- 贷款管理

所在支行 部门号 部门名称 查询 添加

部门管理

部门号	名称	支行编号	电话	经理ID	经理姓名	照片	操作
D001	柜员部	B001	1313131313	S001	钢铁侠		<button>更新</button> <button>删除</button>
D002	接待部	B001	1414141414				<button>更新</button> <button>删除</button>
D003	柜员部	B002	1515151515				<button>更新</button> <button>删除</button>
D004	接待部	B002	1616161616	S004	死侍		<button>更新</button> <button>删除</button>

客户端不能查看部门信息。

## 员工信息页面

```

@app.route('/stuff', methods=['GET', 'POST'])
def stuff():
    .....
    elif form_type == 'insert':
        try:
            with db.session.begin():
                .....

```

```

name = request.form.get('name')
.....
new_stuff = Stuff()
# 检查每个属性是否存在，如果存在则设置相应的值，否则设置为 None
.....
if os.path.exists('Database/lab2/src/static/photo/' +
                  name + '.jpg'):
    new_stuff.figure = name + '.jpg'
.....
.....

```

插入的处理：根据输入的员工姓名，再图像文件夹下寻找是否存在名为姓名.jpg的照片文件，如果存在，则设置相应的照片路径为插入的条目的属性。然后检查员工编号是否重复，部门是否存在。

更新的处理：检查输入的新部门是否存在。

管理端页面如下：

数据库系统及应用Lab

银行业务管理系统

- 首页
- 支行管理
- 部门管理
- 员工管理
- 客户管理
- 账户管理
- 贷款管理

员工ID	姓名	部门编号	职位	联系电话	居住城市	照片	操作
S001	钢铁侠	D001	经理	13000000000	浙江杭州		<a href="#">更新</a> <a href="#">删除</a>
S002	美国队长	D002	员工	13100000000	上海		<a href="#">更新</a> <a href="#">删除</a>
S003	浩克	D003	员工	13200000000	北京		<a href="#">更新</a> <a href="#">删除</a>
S004	死侍	D004	经理	13300000000	江苏南京		<a href="#">更新</a> <a href="#">删除</a>

DBMS © 2023.

客户端不能查看员工信息。

## 客户信息页面

```

@app.route('/client', methods=['GET', 'POST'])
def client():
    labels = ['客户ID', '姓名', '联系电话', '地址']
    global filters
    if request.method == 'GET':
        filters = []
        if authorized:
            result = db.session.query(Customer).all()
        else:
            result = db.session.query(Customer).filter_by(Customer_ID=cur_ID)
    return render_template('client.html',
                           labels=labels,
                           content=result,
                           authorized=authorized,
                           cur_ID=cur_ID)
    elif request.method == 'POST':
        form_type = request.form.get('type')

```

```

if form_type == 'query':
    .....
elif form_type == 'insert':
    try:
        with db.session.begin():
            customer_id = request.form.get('customer_id')
            name = request.form.get('name')
            phone = request.form.get('phone')
            address = request.form.get('address')
            if not db.session.query(Customer).filter_by(
                Customer_ID=customer_id).first():
                new_customer = Customer()
                .....
                db.session.add(new_customer)
                # 同时创建登录账户
                new_user = User(username=customer_id,
                                password=customer_id)
                db.add(new_user)
                db.session.commit()
                filters = []
            else:
                .....
    elif form_type == 'delete':
        try:
            with db.session.begin():
                customer_id = request.form.get('key')
                .....
                # 删除客户
                # 先删除名下的账户
                account_result = db.session.query(Account).filter_by(
                    Customer_ID=customer_id).all()
                for account in account_result:
                    db.session.delete(account)
                # 再删除名下贷款
                loan_result = db.session.query(Loan).filter_by(
                    Customer_ID=customer_id).all()
                for loan in loan_result:
                    db.session.delete(loan)
                # 最后删除客户
                result = db.session.query(Customer).filter_by(
                    Customer_ID=customer_id).first()
                db.session.delete(result)
                # 还要删除登录账号
                result = db.session.query(User).filter_by(
                    username=customer_id).first()
                db.session.delete(result)
                db.session.commit()
                if not authorized:
                    return redirect(url_for('login'))
        except Exception as e:
            db.session.rollback()
            return render_template('404.html',
                                error_title='客户信息删除错误',
                                error_message=str(e._message()))
    .....

```

查询处理：管理端可以查询所有的客户信息，客户端只能根据全局变量 `cur_ID` 查询自己的客户信息。

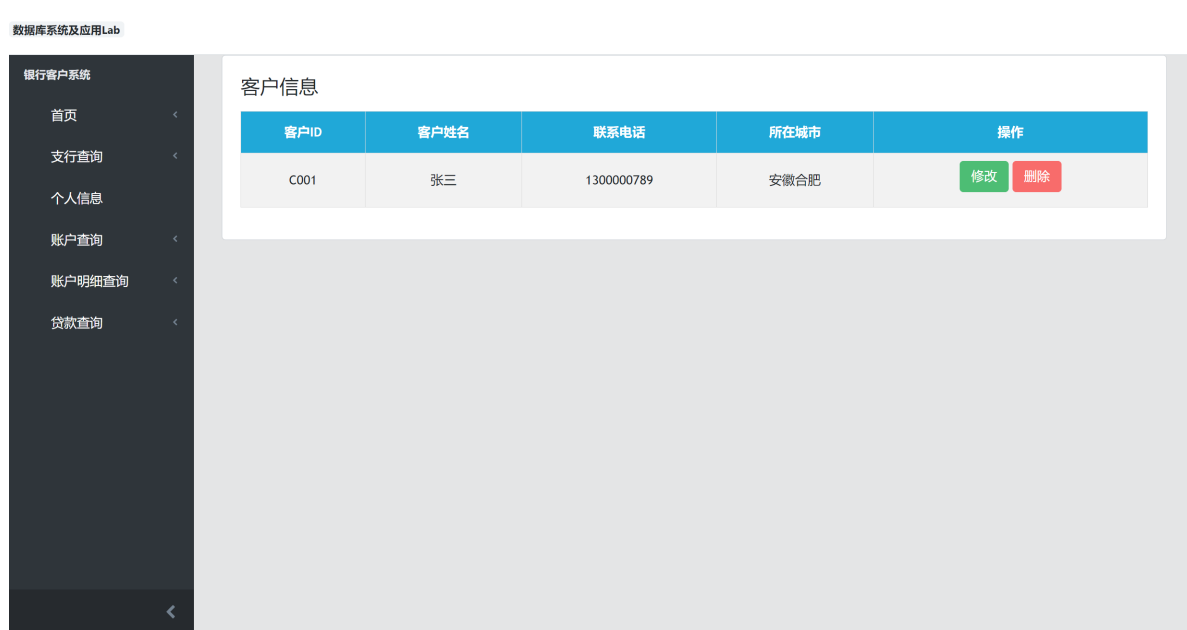
插入处理：只有管理端可以插入客户信息。先检查客户ID是否重复，然后在插入客户ID后，还要在login表中插入对应的账号信息，账号密码都设置为客户ID

删除处理：先检查名下账户是否有余额，然后检查名下贷款是否全部还清。删除时先删除名下账户，再删除名下贷款，最后删除客户及对应的登录账号信息。如果是客户端删除了自己的账号，要回到登录页面重新注册。

管理端页面如下：



客户端查询时默认客户ID为自己的ID，因此也不再需要查询功能。



## 账户信息页面

```
@app.route('/account', methods=['GET', 'POST'])
def account():
    labels = ['账户ID', '客户ID', '银行ID', '开户时间', '余额']
    global filters
    if request.method == 'GET':
```

```

filters = []
if authorized:
    result = db.session.query(Account).all()
else:
    result = db.session.query(Account).filter_by(Customer_ID=cur_ID)
return render_template('account.html',
                        labels=labels,
                        content=result,
                        cur_ID=cur_ID,
                        authorized=authorized)

elif request.method == 'POST':
    form_type = request.form.get('type')
    if form_type == 'query':
        filters = []
        account_id = request.form.get('account_id')
        if authorized:
            customer_id = request.form.get('customer_id')
        else:
            customer_id = cur_ID
        bank_id = request.form.get('bank_id')
        .....
    elif form_type == 'insert':
        try:
            with db.session.begin():
                account_id = request.form.get('account_id')
                if authorized:
                    customer_id = request.form.get('customer_id')
                else:
                    customer_id = cur_ID
                bank_id = request.form.get('bank_id')
            .....

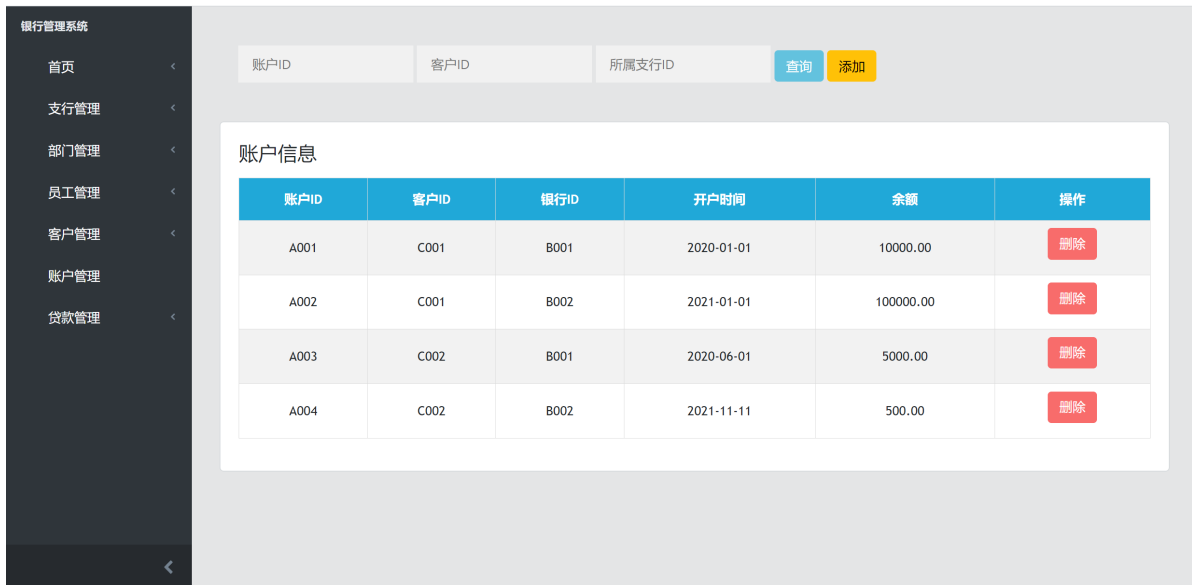
```

查询处理：客户端不用输入客户ID，而是直接设置为自己的ID。

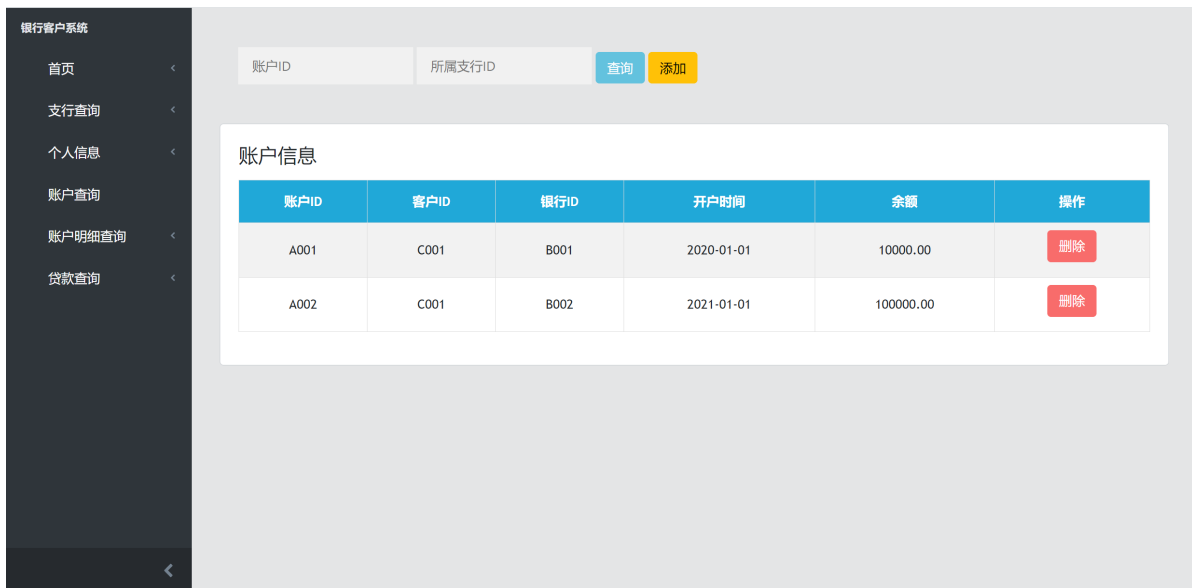
插入处理：客户端不用输入客户ID，而是直接设置为自己的ID。先检查账户ID是否重复，然后检查客户ID、银行ID是否存在。插入账户时要设置余额为0。

删除处理：先检查账户余额是否为0。

管理端页面如下：



客户端查询时不用输入客户ID



## 账户记录信息页面

```
@app.route('/detail', methods=['GET', 'POST'])
def detail():
    global filters
    global accounts
    labels = ['记录ID', '账户ID', '交易时间', '净增值', '交易类型', '交易明细']

    if request.method == 'GET':
        filters = []
        accounts = db.session.query(Account.Account_ID).filter(
            Account.Customer_ID == cur_ID).subquery()
        result = db.session.query(Record).filter(
            Record.Account_ID.in_(accounts)).all()
        return render_template('detail.html', labels=labels, content=result)

    .....

    if account_id:
```

```

AccountExist = db.session.query(Account).filter_by(
    Account_ID=account_id, Customer_ID=cur_ID).first()
if not AccountExist:
    error_title = '账户明细查询错误'
    error_message = '当前用户名下没有该账户'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)
filters.append(Record.Account_ID == account_id)

.....

if Decimal(amount) <= 0:
    error_title = '业务办理错误'
    error_message = '金额必须大于0'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

if action == '取款':
    # 余额不足
    if db.session.query(Account).filter_by(
        Account_ID=account_id).first().Balance < float(
            amount):
        error_title = '取款业务办理错误'
        error_message = '余额不足'
        return render_template('404.html',
                               error_title=error_title,
                               error_message=error_message)

    # 账户余额减少
    db.session.query(Account).filter_by(
        Account_ID=account_id).update(
        {'Balance': Account.Balance - Decimal(amount)})
    # 增加账户明细
    new_record = Record(
        Record_ID=generate_next_record_id(),
        Account_ID=account_id,
        time=datetime.datetime.now(),
        increasement=-Decimal(amount),
        type='支出',
        detail='取款')
    db.session.add(new_record)
elif action == '存款':
    # 账户余额增加
    db.session.query(Account).filter_by(
        Account_ID=account_id).update(
        {'Balance': Account.Balance + Decimal(amount)})
    # 增加账户明细
    new_record = Record(
        Record_ID=generate_next_record_id(),
        Account_ID=account_id,
        time=datetime.datetime.now(),
        increasement=Decimal(amount),
        type='收入',
        detail='存款')
    db.session.add(new_record)
elif action == '转账':
    # 余额不足

```



```

if db.session.query(Account).filter_by(
    Account_ID=account_id).first(
    ).Balance < Decimal(amount):
    error_title = '转账业务办理错误'
    error_message = '余额不足'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

# 没有填写目标账户
if not target_id:
    error_title = '转账业务办理错误'
    error_message = '没有填写目标账户'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

# 没有目标账户
TargetExist = db.session.query(Account).filter_by(
    Account_ID=target_id).first()
if not TargetExist:
    error_title = '转账业务办理错误'
    error_message = '没有目标账户'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

# 自己账户余额减少
db.session.query(Account).filter_by(
    Account_ID=account_id).update(
    {'Balance': Account.Balance - Decimal(amount)})

# 目标账户余额增加
db.session.query(Account).filter_by(
    Account_ID=target_id).update(
    {'Balance': Account.Balance + Decimal(amount)})

# 自己的账户增加账户明细
new_record = Record(
    Record_ID=generate_next_record_id(),
    Account_ID=account_id,
    time=datetime.datetime.now(),
    increasement=-Decimal(amount),
    type='支出',
    detail=f'向{target_id}转账')
db.session.add(new_record)

# 目标账户增加账户明细
new_target_record = Record(
    Record_ID=generate_next_record_id(),
    Account_ID=target_id,
    time=datetime.datetime.now(),
    increasement=Decimal(amount),
    type='收入',
    detail=f'{account_id}的转账')
db.session.add(new_target_record)
filters = []

else:
    error_title = '业务办理错误'
    error_message = '未知业务类型'
    return render_template('404.html',

```

```
error_title=error_title,  
error_message=error_message)
```

.....

只有客户端可以查看账户明细信息。

GET请求处理：先从Account表中查询当前ID名下的账户，存储在变量 `accounts` 中，然后查询 `Account_ID` 在 `accounts` 中的记录。

查询处理：如果输入了账户ID作为筛选信息，先检查账户是否在自己名下。

插入处理：即为存款、取款、转账的事务的办理。`action` 记录事务类型。如果为取款，先检查余额是否足够，如果足够，则减少账户的余额，并插入记录信息。如果为存款，则增加账户余额，并插入记录信息。如果为转账，先检查自己余额是否足够，然后要检查是否填写了目标账户，以及目标账户是否存在，最后更新两个账户的余额，插入相应的记录。

客户端界面：

数据库系统及应用Lab

银行客户系统

首页

支行查询

个人信息

账户查询

账户明细查询

贷款查询

记录ID

账户ID

交易类型(收入/支出)

查询

业务办理

账户交易明细及业务办理

记录ID	账户ID	交易时间	净增值	交易类型	交易明细	操作
R001	A001	2020-01-01	1000.00	收入	存钱	删除
R002	A001	2020-01-02	-500.00	支出	取钱	删除
R003	A001	2021-01-03	-500.00	支出	向A002转账	删除
R004	A001	2021-01-04	1000.00	收入	A002的转账	删除

点击业务办理会弹出表格

数据库系统及应用Lab

银行客户系统

首页

支行查询

个人信息

账户查询

账户明细查询

贷款查询

记录ID

账户ID

业务办理

业务填写

账户ID(必填)

金额(必填)

目标账户ID(转账时必填,其他情况无效)

业务类型(存款/取款/转账)

确认

取消

账户交易明细及业务办理

记录ID	账户ID	交易时间	交易明细	操作
R001	A001	2020-01-01	存钱	删除
R002	A001	2020-01-02	取钱	删除
R003	A001	2021-01-03	向A002转账	删除
R004	A001	2021-01-04	A002的转账	删除

## 贷款信息页面

```
@app.route('/debt', methods=['GET', 'POST'])
def debt():
    labels = ['贷款ID', '客户ID', '银行ID', '未还贷款金额', '贷款期限', '贷款状态']
    global filters
    if request.method == 'GET':
        filters = []
        if authorized:
            result = db.session.query(Loan).all()
        else:
            result = db.session.query(Loan).filter_by(Customer_ID=cur_ID)
        return render_template('debt.html',
                               labels=labels,
                               content=result,
                               cur_ID=cur_ID,
                               authorized=authorized)
    elif request.method == 'POST':
        .....
        if Decimal(loan_amount) <= 0:
            error_title = '贷款信息插入错误'
            error_message = '贷款金额必须大于0'
            return render_template('404.html',
                                   error_title=error_title,
                                   error_message=error_message)
        if datetime.datetime.strptime(term, "%Y-%m-%d").date()
        ) <= datetime.datetime.now().date():
            error_title = '贷款信息插入错误'
            error_message = '贷款期限必须大于当前日期'
            return render_template('404.html',
                                   error_title=error_title,
                                   error_message=error_message)
        new_loan = Loan(Loan_ID=loan_id,
                        Customer_ID=customer_id,
                        Bank_ID=bank_id,
                        RemainingAmount=loan_amount,
                        term=term,
                        status=0)
        db.session.add(new_loan)
        db.session.commit()
        filters = []
    except Exception as e:
        error_title = '贷款信息插入错误'
        error_message = str(e)
        return render_template('404.html',
                               error_title=error_title,
                               error_message=error_message)
    .....

    elif form_type == 'update':
        try:
            with db.session.begin():
                loan_id = request.form.get('loan_id')
                amount = request.form.get('amount')
                loan = db.session.query(Loan).filter_by(
                    Loan_ID=loan_id).first()
```

```

if Decimal(amount) <= 0:
    error_title = '还贷错误'
    error_message = '还款金额必须大于0'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

if Decimal(amount) > loan.RemainingAmount:
    error_title = '还贷错误'
    error_message = '还款金额不能大于剩余未还金额'
    return render_template('404.html',
                           error_title=error_title,
                           error_message=error_message)

db.session.query(Loan).filter_by(Loan_ID=loan_id).update({
    'RemainingAmount':
        loan.RemainingAmount - Decimal(amount)
})

loan = db.session.query(Loan).filter_by(
    Loan_ID=loan_id).first()
if loan.RemainingAmount == 0:
    db.session.query(Loan).filter_by(
        Loan_ID=loan_id).update({'status': 1})
db.session.commit()
.....

```

查询处理：客户端只能查询自己名下的贷款信息。

插入处理：即为借贷处理，依次检查贷款ID是否重复、客户ID是否存在、银行ID是否存在、贷款金额是否为正数、贷款期限是否晚于今天。

删除处理：先检查贷款是否还完，才能删除贷款

更新处理：即为还贷处理，先检查还贷金额为正数，再检查还贷金额是否不超过剩余未还贷款金额。然后更新贷款剩余金额信息，最后根据剩余金额是否为0来更新贷款状态。

管理端界面如下：

数据库系统及应用Lab

银行管理系统

- 首页
- 支行管理
- 部门管理
- 员工管理
- 客户管理
- 账户管理
- 贷款管理

贷款ID

客户ID

所属支行ID

是否还清(是/否)

查询

添加

贷款信息

贷款ID	客户ID	银行ID	未还贷款金额	贷款期限	贷款状态	操作
L001	C001	B001	10000.00	2022-01-01	已还清	删除
L002	C001	B002	100000.00	2023-01-01	未还清	删除

客户端查询不用输入客户ID，增加还贷按钮



点击申请贷款会弹出表单：



点击还贷会弹出表单：



## 实验总结

本次实验综合性较强，也学到了很多东西。

首先是学会了Flask和SQLAlchemy的使用。用python语言来实现数据库的增删改查，以及存储过程和事务。

其次是学会了网页的编写，以及前后端的交互。前端向使用者展示页面，使用者通过填写表单，点击按钮来向后端传递信息。后端接受数据处理数据，更新数据库，并将新的数据传递给前端用于显示。

实验过程中，MySQL的实现没有感觉到难度，在构建好清晰的ER图和需求分析后就比较容易了。难点主要在于网页的实现，因为没有接触过网页的编写，基本上得在网上边学边用，很多时候的页面呈现和自己想要的并不一样，但是基本上也是熟能生巧，前几个页面比较困难，后面的网页只需要在其他网页基础上修改就好了。其次是一些异常情况的检测，什么情况下才能增删改，以及怎么对增删改做处理还不引起外键约束等异常，都需要考虑清楚，需要一边实现一边完善。

虽然功能基本实现完全，但是有些部分也不太符合实际情况，比如电话位数的限制。也还可以增加许多其他功能，比如员工照片的文件上传。

这个系统也存在一个缺陷，就是只能在本地访问页面，没办法在其他地方访问，但是上网查询发现要配置路由器防火墙等，感觉实现起来比较困难，就没有实现。

感觉工作量还是略大，听说往年都是组队实验，感觉还是组队要轻松一点，也能让整体实现不那么粗糙。

总体来看，这次实验还是很有用的，增加了自己的编程经验，既活用了上课学到的数据库知识，也学会了很多其他实用的能力，对未来的学习、工作有重要意义。