数据库大作业-银行管理系统 实验报告

PB21081601 张芷苒

实验环境：Windows 11 + Python 3.11 + Pycharm + Django

1 实验目标

本次实验的目标是完成一个银行管理系统的数据库应用开发。该系统涉及多个实体，包括银行信息、客户信息、账户信息、贷款信息、部门信息、员工信息等。

在本次实验中，需要完成以下任务：

1. 需求/功能分析：对银行管理系统的功能进行分析和需求确定，明确系统需要实现的功能和业务逻辑。
2. 架构和语言：选择适合的架构和编程语言，用于实现银行管理系统的后端逻辑和数据库设计。
3. 界面设计：设计银行管理系统的用户界面，包括登录页面、主页面、数据展示页面等，以提供用户友好的操作界面。
4. 后端逻辑实现（数据库设计）：根据需求分析设计数据库模型，包括定义实体的属性和关系，确定实体之间的关联和约束。编写后端逻辑代码，包括模型定义、视图函数和路由设计等，用于处理用户请求和与数据库交互。
5. 软件测试：对银行管理系统进行测试，包括单元测试和集成测试，以确保系统功能的正确性和稳定性。
6. 说明文档：编写说明文档，详细描述银行管理系统的设计和实现过程，包括需求分析、概念设计、逻辑设计、系统架构与界面设计、后端逻辑实现、展示结果等内容。

通过完成以上任务，最终目标是实现一个完整的银行管理系统，能够对银行信息、客户信息、账户信息、贷款信息、部门信息、员工信息等进行管理和操作，提供用户友好的界面和功能。该系统能够满足银行管理的需求，包括客户管理、账户管理、贷款管理、员工管理等多种功能。

2 需求分析

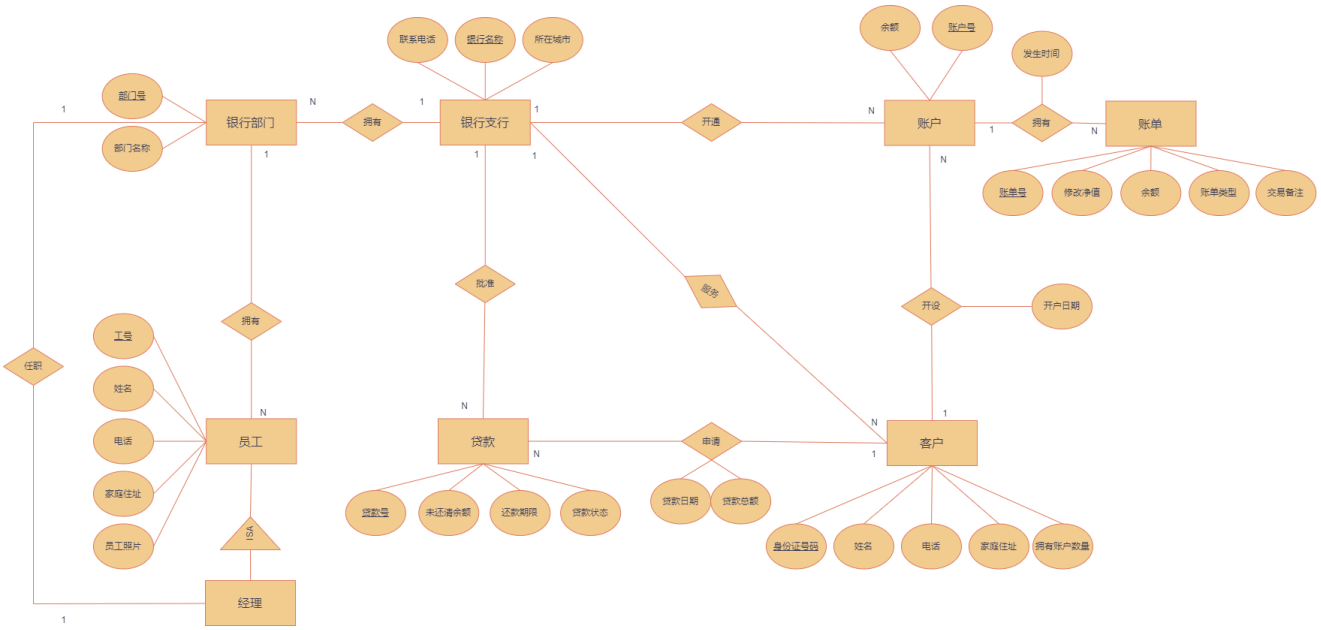
2.1 数据需求

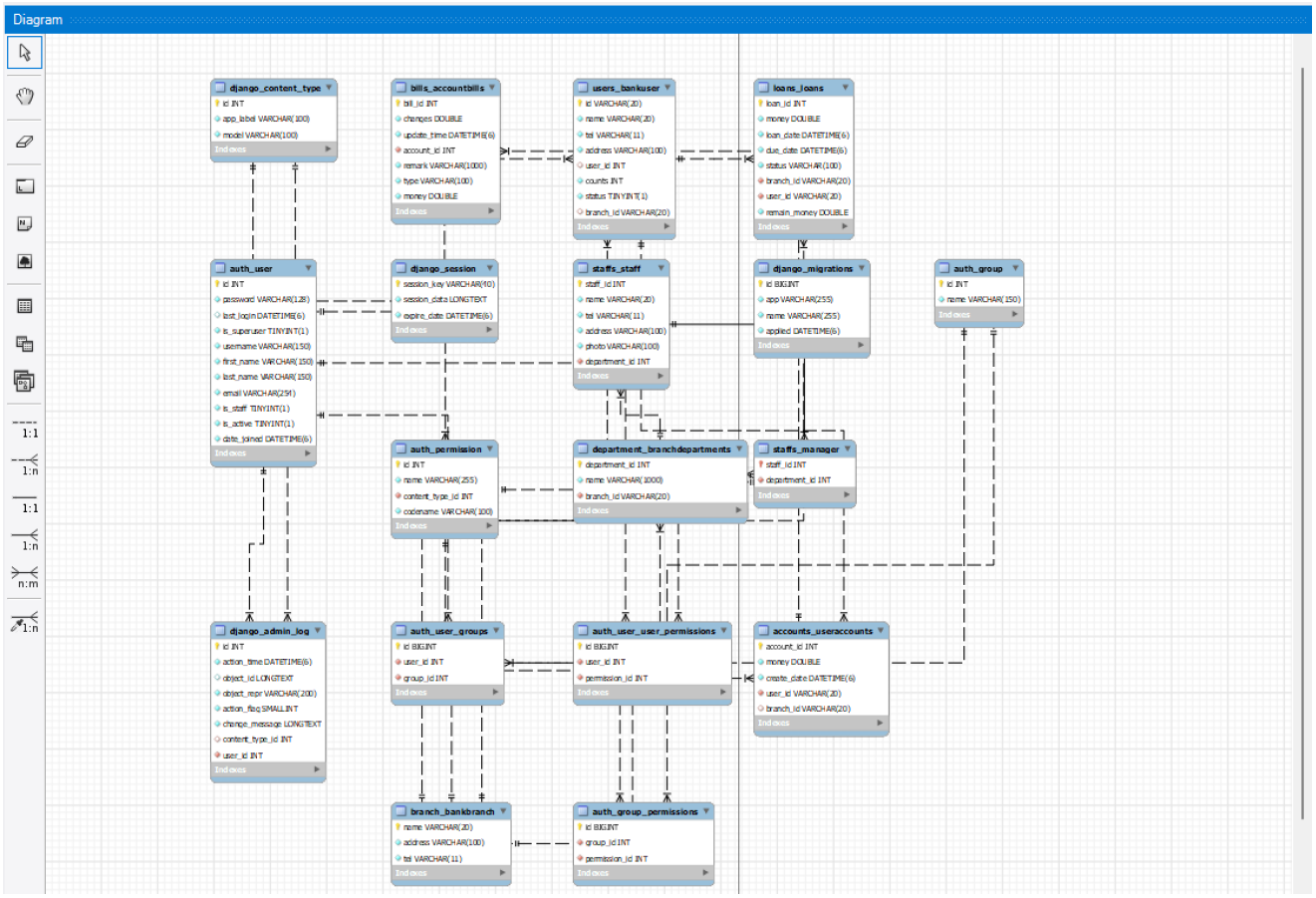
一家银行在多个城市都开设了支行，每个银行支行可以用银行名字唯一确认，每个银行支行都有一个联系电话。每个支行银行都拥有多个银行部门，可以用部门号确认，每个银行部门都有唯一非空的部门号，部门信息还包括部门名称和部门经理。银行部门分别管理各自的员工，员工可以用工号来唯一确认，其他信息包括姓名，电话，家庭住址和员工照片，全部信息都不能为空，员工中还包括一部分的部门经理，与银行部门存在一对一的关系。每个客户都需要保存姓名，电话，身份证号码与家庭住址，还有客户开通的账户的数目与客户状态，客户可以由身份证号码来确认，且要求姓名，电话，身份证号码不能为空。客户可以在某个支行银行开通账户并将开户时间记录到账户，这个账户包含开户者的开户日期，账户号与账户余额，账户由账户号唯一确定。客户在每个账户进行的存取历史记录都被保存在账单记录中，账单记录保存了发生时间，修改净值，余额，账单类型以及交易备注，并且使用账单号来唯一标识账单。客户还可以在支行银行贷款，贷款由客户申请、支行银行批准，贷款信息包括贷款号，贷款总额，未还清余额，贷款状态，贷款日期以及还款期限，每笔贷款由贷款号唯一确定。

2.2 功能需求

- 客户管理：提供客户信息的增加、修改和查阅功能。提供一个支持客户注册、登录以及退出的系统。
- 账户管理：提供账户信息的增加、修改、删除和查阅功能，账户号不能修改。 并且支持不同帐户间的转账。
- 贷款管理：提供贷款信息的增加、修改、删除和查阅的功能，未归还状态的贷款记录不能删除。客户可以使用某个账户来还清贷款。
- 员工管理：提供员工信息的增加、修改、删除和查阅功能，其中工号不能修改。
- 经理管理：提供部门经理的设置与取消功能。
- 账单管理：提供账单的增加和修改功能。

3 ER图





4 逻辑设计

将 ER 模型转换成关系数据库模式：

首先将每个实体转换为一个关系模式，实体的属性为关系模式的属性，实体的标识成为关系模式的主码：

银行支行(银行名称，联系电话，所在城市)
银行部门(部门号，部门名称)
员工(工号，姓名，电话，家庭住址，员工照片)
客户(身份证号码，姓名，电话，家庭住址，拥有账户数量)
账户(账户号，余额)
账单(账单号，修改净值，余额，账单类型，交易备注)
贷款(贷款号，未还清余额，还款期限，贷款状态)

对子类 "经理" 进行转化；
进行联系转换；
对于银行支行和客户的 1 到 N 联系，将银行支行的实体标识加入到客户关系模式中；
对于银行支行和账户的 1 到 N 联系，将银行支行的实体标识加入到账户关系模式中；
对于银行支行和贷款的 1 到 N 联系，将银行支行的实体标识加入到贷款关系模式中；
对于银行支行和银行部门的 1 到 N 联系，将银行支行的实体标识加入到银行部门关系模式中；
对于客户和贷款的 1 到 N 联系，将客户的实体标识以及联系属性加到贷款关系模式中；
对于客户和账户的 1 到 N 联系，将客户的实体标识以及联系属性加到账户关系模式中；
对于账户和账单的 1 到 N 联系，将账户的实体标识以及联系属性加到账单关系模式中；
对于银行部门和员工的 1 到 N 联系，将银行部门的实体标识加到员工关系模式中；
对于银行部门和经理的 1 到 1 联系，将银行部门的实体标识加到经理关系模式中。

最后得到的关系数据库模式如下：

银行支行(银行名称，联系电话，所在城市)
银行部门(部门号，银行名称，部门名称)
员工(工号，部门号，姓名，电话，家庭住址，员工照片)
经理(工号，部门号)
客户(身份证号码，银行名称，姓名，电话，家庭住址，拥有账户数量)
账户(账户号，身份证号码，银行名称，余额，开户日期)
账单(账单号，账户号，发生时间，修改净值，余额，账单类型，交易备注)
贷款(贷款号，身份证号码，银行名称，贷款总额，未还清余额，贷款日期，还款期限，贷款状态)

接着进行关系数据库模式的规范化处理。在本次实验的设计中，可以很清楚的发现每个关系模式中，所有属性都是完全且非传递的依赖于主码。其所有的函数依赖集如下：

$$
\begin{aligned}
&\text{函数依赖集}_{\text{银行支行}} = \{\text{银行名称} \rightarrow \text{联系电话}, \text{银行名称} \rightarrow \text{所在城市}\} \\
&\text{函数依赖集}_{\text{银行部门}} = \{\text{部门号} \rightarrow \text{银行名称}, \text{部门号} \rightarrow \text{部门名称}\} \\
&\text{函数依赖集}_{\text{员工}} = \{\text{工号} \rightarrow \text{部门号}, \text{工号} \rightarrow \text{姓名}, \text{工号} \rightarrow \text{电话}, \text{工号} \rightarrow \text{家庭住址}, \text{工号} \rightarrow \text{员工照片}\} \\
&\text{函数依赖集}_{\text{客户}} = \{\text{身份证号码} \rightarrow \text{银行名称}, \text{身份证号码} \rightarrow \text{姓名}, \text{身份证号码} \rightarrow \text{电话} \\
&\qquad \text{身份证号码} \rightarrow \text{家庭住址}, \text{身份证号码} \rightarrow \text{拥有账户数量}\} \\
&\text{函数依赖集}_{\text{账户}} = \{\text{账户号} \rightarrow \text{身份证号码}, \text{账户号} \rightarrow \text{银行名称}, \text{账户号} \rightarrow \text{余额}, \text{账户号} \rightarrow \text{开户日期}\} \\
&\text{函数依赖集}_{\text{账单}} = \{\text{账单号} \rightarrow \text{账户号}, \text{账单号} \rightarrow \text{发生时间}, \text{账户号} \rightarrow \text{修改净值}, \\
&\qquad \text{账单号} \rightarrow \text{余额}, \text{账单号} \rightarrow \text{账单类型}, \text{账单号} \rightarrow \text{交易备注}\} \\
&\text{函数依赖集}_{\text{贷款}} = \{\text{贷款号} \rightarrow \text{身份证号码}, \text{贷款号} \rightarrow \text{银行名称}, \text{贷款号} \rightarrow \text{贷款总额}, \\
&\qquad \text{贷款号} \rightarrow \text{未还清余额}, \text{贷款号} \rightarrow \text{贷款日期}, \text{贷款号} \rightarrow \text{还款期限}, \text{贷款号} \rightarrow \text{贷款状态}\}
\end{aligned}
$$

因此，经过逻辑设计的规范化分析，确认该关系数据库模式已满足第三范式 3NF 的要求，并不需要进行进一步的模式分解。在后续的系统设计和实现中，经过规范化检查后的关系数据库模式完全满足需求分析，并且功能和性能也能够满足预期需求。因此，无需进行模式修正。通过这一过程，确认了数据库设计的合理性，并确保了系统的功能性和性能的满足。

5 系统设计与实现

本次实验采用的是 B/S 架构。采用的实现语言以及 Web 框架是 Python + Django，采用的后端数据库为 MySQL。后端开发过程中，主要根据软件的功能需求提供数据接口：从数据库或其他数据源写入、读取和处理数据；前端开发过程中，根据软件的功能需求设计界面：采用简单的 Bootstrap 样式来优化应用界面。

在实现银行管理系统过程中，为了保证信息的安全性和合规性，针对不同的用户角色和操作设置了访问权限控制。这意味着用户只能访问其具有权限的内容和功能，而对于没有权限的内容和功能，系统会显示相应的错误信息以提示用户。

当用户尝试访问其无权访问的内容时，系统会检测到该操作并生成相应的错误信息。这个错误信息会向用户传达访问被拒绝的信息，并提醒用户该操作无法执行。错误信息的内容通常会描述具体的原因，例如"无法为他人创建账户"或"无法删除他人贷款"等，以帮助用户理解问题所在。

5.1 分支银行

```
class BankBranch(models.Model):
        objects = models.Manager()
        name = models.CharField(max_length=20, primary_key=True)
        address = models.CharField(max_length=100)
        tel = models.CharField(max_length=11)
```

该模型类定义了银行分支机构的基本属性，可以用于存储和检索分支机构的信息。

```
def branches(request):
        branches_lists = BankBranch.objects.all()
        paginator = Paginator(branches_lists, 4)
        page = request.GET.get('page')
        branches_list = paginator.get_page(page)
        context = {'branches': branches_list}
        return render(request, 'branches/branches.html', context)
```

`branches` 函数是一个视图函数，用于显示银行分支机构的列表页面。通过 `BankBranch.objects.all()` 从数据库中获取所有的银行分支机构对象，存储在 `branches_lists` 中。使用分页器（Paginator）对分支机构列表进行分页处理，每页显示 4 个分支机构。获取请求中的页码。根据页码获取当前页的分支机构列表。构建上下文（context）字典，包含分支机构列表。渲染分支机构列表页面，并将上下文传递给模板进行渲染。

总体来说，该函数实现了显示银行分支机构列表的功能，包括分页处理和将数据传递给模板进行渲染。用户访问该视图时，将显示所有分支机构的列表，并支持分页浏览。

5.2 银行部门

```
class BranchDepartments(models.Model):
        objects = models.Manager()
        branch = models.ForeignKey(BankBranch,
on_delete=models.CASCADE,
        related_name='BranchDepartments')
        department_id = models.AutoField(primary_key=True)
        name = models.CharField(max_length=1000)
```

该模型类定义了银行分支机构部门的基本属性，与 BankBranch 模型类建立了一对多的关联关系，可以用于存储和检索分支机构的部门信息。

5.2.1 查看银行部门信息

```
def departments(request):
        name = None
        if request.user.is_superuser:
                name = request.user.username
        if name:
                departments_lists =
BranchDepartments.objects.filter(branch_id=name)
        else:
                departments_lists = BranchDepartments.objects.all()
                paginator = Paginator(departments_lists, 4)
```

```
                page = request.GET.get('page')
                departments_list = paginator.get_page(page)
        # get managers
        managers = Manager.objects.all()
        for department in departments_list:
                for manager in managers:
                        if department.department_id ==
manager.department.department_id:
                        department.manager = manager.staff
        context = {'departments': departments_list}
        return render(request, 'departments/departments.html', context)
```

首先，通过判断当前登录用户是否为超级用户来确定是否需要限制只显示特定分支机构的部门。如果是超级用户，则通过 request.user.username 获取当前部门的名称，存储在 name 变量中。接下来，根据 name 的值来获取相应的部门信息。如果 name 有值，则使用 filter() 方法根据分支机构的名称过滤部门列表，否则获取所有的部门信息。然后，使用 Paginator 类将部门列表分页，每页显示 4 个部门。对于每个部门，通过遍历 managers 对象获取部门的经理信息。遍历过程中，如果部门的department_id 与经理的 department.department_id 相匹配，则将经理信息赋值给部门的manager 属性。最后，将部门列表存储在 context 字典中，传递给模板 departments/departments.html 进行渲染。函数最后返回渲染后的响应结果。

该视图函数用于在页面中显示部门信息，并将部门经理与部门进行关联显示。

5.2.2 查看部门员工

```
def department_staffs(request, department_id):
        branch_name = None
        if request.user.is_superuser:
                branch_name = request.user.username
        branch =

BranchDepartments.objects.get(department_id=department_id).branch_id
        if not (branch_name and branch_name == branch):
                messages.error(request, '你没有权限查看该部门员工')
        return render(request, 'frontend/error.html')
        staffs_lists =
Staff.objects.filter(department_id=department_id)
        paginator = Paginator(staffs_lists, 4)
        page = request.GET.get('page')
        staffs_list = paginator.get_page(page)
        context = {'staffs': staffs_list}
        return render(request, 'departments/staffs.html', context)
```

首先，通过判断当前登录用户是否为超级用户来确定是否需要限制只显示特定分支机构的部门员工信息。如果是超级用户，则通过 request.user.username 获取当前支行名称，存储在 branch_name 变量中。接下来，根据传入的 department_id 参数获取部门对应的分支机构名称，并存储在 branch 变量中。然后，进行权限验证，判断当前用户是否具有查看该部门员工的权限。如果不满足条件，则显示错误信息并返回错误页面。如果满足权限要求，则根据部门 ID 过滤出该部门的员工信息。使用Paginator 类将员工列表分页，每页显示 4 个员工。最后，将员工列表存储在 context 字典中，传递给模板 departments/staffs.html 进行渲染。函数最后返回渲染后的响应结果。

该视图函数用于在页面中显示特定部门的员工信息，并进行权限验证以确保只有具有权限的用户才能查看.

5.3 员工与经理

```
class Staff(models.Model):
        objects = models.Manager()
        staff_id = models.AutoField(primary_key=True)
        department = models.ForeignKey(BranchDepartments,
on_delete=models.CASCADE,
        related_name='DepartmentStaff')
        name = models.CharField(max_length=20)
        tel = models.CharField(max_length=11)
        address = models.CharField(max_length=100)
        photo = models.ImageField(upload_to='photos/%Y%m%d/',
        default='photos/date/default.png')
```

该模型表示员工的基本信息，与部门进行关联，可以通过部门字段获取员工所属的部门信息。照片字段用于存储员工的照片文件，并指定了上传路径和默认图片。

```
class Manager(models.Model):
        objects = models.Manager()
        staff = models.OneToOneField(Staff, on_delete=models.CASCADE,
        related_name='StaffManager', primary_key=True)
        department = models.ForeignKey(BranchDepartments,
on_delete=models.CASCADE,
        related_name='DepartmentManager')
```

这个模型表示每个部门和经理是个一对一之间的关系。

5.3.1 查看员工

```
def staffs(request):
        staffs_lists = Staff.objects.all()
```

```
        paginator = Paginator(staffs_lists, 4)
        page = request.GET.get('page')
        staffs_list = paginator.get_page(page)
        context = {'staffs': staffs_list}
        return render(request, 'staffs/staffs.html', context)
```

通过 Staff.objects.all() 查询所有员工的列表。使用 Paginator 对员工列表进行分页，每页显示4 条员工记录。从请求的 GET 参数中获取当前页数。从分页器中获取当前页的员工列表。创建一个字典context ，将员工列表存储在键 'staffs' 下。使用 render 函数将员工列表数据传递给名为'staffs/staffs.html' 的模板，并返回渲染后的 HTML 响应。

这个视图函数用于展示员工列表页面，并将分页后的员工数据传递给模板进行渲染。

5.3.2 增加员工

```
class Manager(models.Model):
        objects = models.Manager()
        staff = models.OneToOneField(Staff, on_delete=models.CASCADE,
        related_name='StaffManager', primary_key=True)
        department = models.ForeignKey(BranchDepartments,
on_delete=models.CASCADE,
        related_name='DepartmentManager')
        def staffs(request):
                staffs_lists = Staff.objects.all()
                paginator = Paginator(staffs_lists, 4)
                page = request.GET.get('page')
                staffs_list = paginator.get_page(page)
                context = {'staffs': staffs_list}
                return render(request, 'staffs/staffs.html', context)
        def create_staff(request, department_id):
                name = None
                if request.user.is_superuser:
                        name = request.user.username
                        branch =

BranchDepartments.objects.get(department_id=department_id).branch_id
                if not (name and name == branch or name == 'admin'):
                        messages.error(request, '你没有权限操作该部门')
                return render(request, 'frontend/error.html')
                # judge if user is superuser
                if not request.user.is_superuser:
                        messages.error(request, '你没有权限创建员工')
                return render(request, 'frontend/error.html')
                department =
BranchDepartments.objects.get(department_id=department_id)
                if request.method ≠ 'POST':
```

```
                        form = StaffCreateForm(initial={'department':
department})
                else:
                        form = StaffCreateForm(initial={'department':
department},
                data=request.POST, files=request.FILES)
                if form.is_valid():
                        name = form.cleaned_data.get("name")
                        tel = form.cleaned_data.get("tel")
                        address = form.cleaned_data.get("address")
                        staff =
Staff.objects.create(department=department, name=name,
                        tel=tel, address=address)
                if 'photo' in request.FILES:
                        staff.photo = form.cleaned_data.get("photo")
                        staff.save()
                return redirect('staffs:staffs')
        context = {'form': form, 'department': department}
        return render(request, 'staffs/create_staff.html', context)
```

通过 request.user.is_superuser 判断当前用户是否为超级用户。获取指定部门的分支名称。根据部门 ID 获取部门对象。如果请求的方法不是 POST，则创建一个空的员工创建表单 StaffCreateForm 并传入初始值为指定部门。如果请求的方法是 POST，则创建一个带有请求数据和文件的员工创建表单 StaffCreateForm 。验证表单数据是否有效，如果有效则创建员工记录并保存。如果请求中包含上传的照片文件，则将照片赋值给员工对象的 photo 字段。重定向到员工列表页面。创建一个字典 context ，将员工创建表单和部门对象存储在相应的键下。使用 render 函数将数据传递给名为 `staffs/create_staff.html` 的模板，并返回渲染后的 HTML 响应。

这个视图函数用于展示员工创建页面，并处理员工创建表单的提交。只有超级用户有权限创建员工记录，并且只能在对应部门的分支下进行创建。创建成功后，将重定向到员工列表页面。

5.3.3 修改员工信息

```
def edit_staff(request, staff_id):
    # judge if user is superuser
    if not request.user.is_superuser:
        messages.error(request, '你没有权限修改员工信息')
        return render(request, 'frontend/error.html')
    staff = Staff.objects.get(staff_id=staff_id)
    old_department = staff.department
    if request.method != 'POST':
        form = StaffEditForm(instance=staff)
    else:
```

```
        form = StaffEditForm(instance=staff, data=request.POST,
files=request.FILES)
        if form.is_valid():
            staff.department = form.cleaned_data.get("department")
            # if staff is the old department manager
            # and old department is not the new department, delete the
manager
            if old_department ≠ staff.department and
Manager.objects.filter(department=old_department):
                manager = Manager.objects.get(staff=staff,
department=old_department)
                manager.delete()
            staff.name = form.cleaned_data.get("name")
            staff.tel = form.cleaned_data.get("tel")
            staff.address = form.cleaned_data.get("address")
            if 'photo' in request.FILES:
                # if old photo is not default photo, then delete old
photo
                if staff.photo and staff.photo.name ≠
'photos/20230601/default.png':
                    staff.photo.delete()
                staff.photo = form.cleaned_data.get("photo")
            staff.save()
            return redirect('staffs:staffs')
    context = {'form': form, 'staff': staff}
    return render(request, 'staffs/edit_staff.html', context)
```

通过 request.user.is_superuser 判断当前用户是否为超级用户，如果不是，则返回无权限错误页面。根据员工 ID 获取员工对象。如果请求的方法不是 POST，则创建一个员工编辑表单
StaffEditForm 并传入初始值为员工对象。如果请求的方法是 POST，则创建一个带有请求数据和文件的员工编辑表单 StaffEditForm 。验证表单数据是否有效，如果有效则更新员工对象的相关字段。如果员工的部门发生变化，并且旧部门存在相关的经理记录，则删除旧部门的经理记录。如果请求中包含上传的照片文件，则更新员工对象的 photo 字段。如果旧照片不是默认照片，则删除旧照片。保存更新后的员工对象。重定向到员工列表页面。创建一个字典 context ，将员工编辑表单和员工对象存储在相应的键下。使用 render 函数将数据传递给名为 `staffs/edit_staff.html` 的模板，并返回渲染后的 HTML 响应。

这个视图函数用于展示员工编辑页面，并处理员工编辑表单的提交。只有超级用户有权限修改员工信息。编辑成功后，将重定向到员工列表页面。如果旧部门有相关的经理记录，当员工所在的部门发生变化时，将删除旧部门的经理记录。如果上传了新的照片文件，将更新员工的照片字段，并删除旧照片（如果旧照片不是默认照片）。

5.3.4 删除员工

```
def delete_staff(request, staff_id):
    # judge if user is superuser
    if not request.user.is_superuser:
        messages.error(request, '你没有权限删除员工')
        return render(request, 'frontend/error.html')
    staff = Staff.objects.get(staff_id=staff_id)
    # if staff is a manager, delete it
    if Manager.objects.filter(staff=staff):
        manager = Manager.objects.get(staff=staff)
        manager.delete()
    # if staff photo is not the default photo, delete staff photo
    if staff.photo and staff.photo.name ≠
'photos/20230601/default.png':
        staff.photo.delete()
    staff.delete()
    return redirect('staffs:staffs')
```

通过 request.user.is_superuser 判断当前用户是否为超级用户，如果不是，则返回无权限错误页面。根据员工 ID 获取员工对象。如果员工是一个经理，先删除该经理记录。如果员工的照片不是默认照片，则删除员工的照片文件。删除员工对象。重定向到员工列表页面。

这个视图函数用于处理删除员工的请求。只有超级用户有权限删除员工。在删除员工之前，会检查员工是否是一个经理，如果是，则先删除经理记录。然后，会检查员工的照片是否为默认照片，如果不是，则删除照片文件。最后，删除员工对象，并重定向到员工列表页面。

5.3.5 设置经理

```
def set_manager(request, staff_id, department_id):
    name = None
    if request.user.is_superuser:
        name = request.user.username
    branch =
BranchDepartments.objects.get(department_id=department_id).branch_id
    if not (name and name == branch or name == 'admin'):
        messages.error(request, '你没有权限操作该部门')
        return render(request, 'frontend/error.html')
    # judge if user is superuser
    if not request.user.is_superuser:
        messages.error(request, '你没有权限设置经理')
        return render(request, 'frontend/error.html')
```

```
    staff = Staff.objects.get(staff_id=staff_id)
    department =
BranchDepartments.objects.get(department_id=department_id)
    # if there is a manager in this department, delete it
    if Manager.objects.filter(department=department):
        manager = Manager.objects.get(department=department)
        manager.delete()
    # create a new manager
    manager = Manager.objects.create(department=department,
staff=staff)
    manager.save()
    return redirect('departments:departments')
```

函数接受 staff_id 和 department_id 作为参数，表示要设置为经理的员工和对应的部门。函数首先进行权限验证，判断用户是否有权限操作该部门和设置经理的权限。然后根据传入的 staff_id 和department_id 获取相应的员工和部门对象。接下来，如果该部门已经有经理存在，先删除现有的经理。最后，创建一个新的经理对象，并将员工和部门关联起来，保存到数据库中。最后，重定向到部门列表页面。该函数用于在部门列表页面中设置经理，并返回到部门列表页面。

5.3.6 删除经理

```
def delete_manager(request, department_id):
    # judge if user is superuser
    if not request.user.is_superuser:
        messages.error(request, '你没有权限删除经理')
        return render(request, 'frontend/error.html')
    department =
BranchDepartments.objects.get(department_id=department_id)
    # if there is a manager in this department, delete it
    if Manager.objects.filter(department=department):
        manager = Manager.objects.get(department=department)
        manager.delete()
    return redirect('departments:departments')
```

函数接受 department_id 作为参数，表示要删除经理的部门。函数首先进行权限验证，判断用户是否有权限删除经理的权限。然后根据传入的 department_id 获取对应的部门对象。接下来，如果该部门存在经理，删除经理对象。最后，重定向到部门列表页面。该函数用于在部门列表页面中删除经理，并返回到部门列表页面。

该函数用于在部门列表页面中删除经理，并返回到部门列表页面。只有超级用户有权限删除经理。

5.4 客户

```
# Create Bank User Information
class BankUser(models.Model):
    objects = models.Manager()
    user = models.OneToOneField(User, on_delete=models.CASCADE,
related_name='BankUser', null=True)
    branch = models.ForeignKey(BankBranch, on_delete=models.CASCADE,
related_name='BranchUser', null=True)
    id = models.CharField(max_length=20, primary_key=True)
    name = models.CharField(max_length=20)
    tel = models.CharField(max_length=11)
    address = models.CharField(max_length=100)
    counts = models.IntegerField(default=0)
    status = models.BooleanField(default=True)
```

在实现客户模块的时候，将其与 Django 内置的 User 模块进行了一对一映射，从而可以方便的时候Django 内置的注册、登录以及退出系统，这样就不必自己实现一个会话来保持用户的登录状态。并且以内置的 User 模块来实现，可以利用其已有的安全检查系统，这样便可以保证客户系统的安全性。

该 BankUser 模型定义了银行用户的各个属性，包括用户与用户认证系统的关联、银行分行的关联以及一些其他信息字段。这些字段将在数据库中作为表的列，用于存储和检索银行用户的数据。

5.4.1 客户注册

```
def bank_user_register(request):
    if request.method == 'POST':
        form = UserCreationForm(data=request.POST)
        register_form = BankUserRegisterForm(data=request.POST)
        if form.is_valid() and register_form.is_valid():
            username = form.cleaned_data.get("username")
            password = form.cleaned_data.get("password1")
            id = register_form.cleaned_data.get("id")
            # # check if there is id
            # if BankUser.objects.filter(id=id):            #
messages.error(request, '该身份证号码已被注册')
            #     return render(request, 'frontend/error.html')
user = User.objects.create_user(username=username, password=password)
            name = register_form.cleaned_data.get("name")
            tel = register_form.cleaned_data.get("tel")
            address = register_form.cleaned_data.get("address")
            branch = register_form.cleaned_data.get("branch")
            bank_user = BankUser.objects.create(user=user, id=id,
name=name,
                                                tel=tel,
```

```
address=address, branch=branch)
            bank_user.save()
            login(request, user)
            return redirect('frontend:index')
        else:
            return HttpResponse('输入不合法或该用户名/身份证号码已注册')
    elif request.method == 'GET':
        form = UserCreationForm()
        register_form = BankUserRegisterForm()
        context = {'form': form, 'register_form': register_form}
        return render(request, 'registration/register.html', context)
```

首先通过表单获取客户注册的信息，然后将其解析： if form.is_valid() and register_form.is_valid(): ，这段代码会自动检查是否有信息重叠的客户已经存在系统内了。当输入的表单信息是合法时，这时候将会先创建一个 User 模块的 user ，然后再将其一对一映射绑定到自己定义的 BankUser 中。这样，在系统中就会存在 User 和 BankUser 两个表，而User 表主要负责客户的登录管理功能，而 BankUser 表主要负责数据库的需求功能。

5.4.2 客户登录

```
def bank_user_login(request):
    if request.method == 'POST':
        bank_user_login_form = BankUserLoginForm(data=request.POST)
        if bank_user_login_form.is_valid():
            data = bank_user_login_form.cleaned_data
            user = authenticate(username=data['username'],
password=data['password'])
            if user:
                login(request, user)
                return redirect('frontend:index')
            else:
                return HttpResponse('账号或密码输入错误，请重新输入')
        else:
            return HttpResponse('账号或密码输入不合法，请重新输入')
    elif request.method == 'GET':
        bank_user_login_form = BankUserLoginForm()
        context = {'form': bank_user_login_form}
        return render(request, 'registration/login.html', context)
```

客户登录的时候，使用的就是注册时采用的用户名 username ，然后通过 django.contrib.auth 中的authenticate() 来验证用户的信息是否正确，如果正确，则使用 django.contrib.auth 的login() 函数进行登录。

5.4.3 客户修改密码

```
@login_required
def change_pwd(request, user_id):
    # judge if the user is the owner
    user = User.objects.get(id=user_id)
    if user ≠ request.user:
        messages.error(request, '无法修改他人密码')
        return render(request, 'frontend/error.html')
    if request.method ≠ 'POST':
        form = PasswordChangeForm(user=request.user)
    else:
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            user = form.save()
            update_session_auth_hash(request, user)
            return redirect('frontend:index')
    context = {'form': form}
    return render(request, 'registration/change_pwd.html', context)
```

@login_required 是一个装饰器，用于限制只有已登录的用户才能访问该视图。如果用户未登录，装饰器会将其重定向到登录页面。同时使用 if user != request.user 这行代码判断获取的用户对象是否与当前请求的用户对象相同。如果不同，意味着用户正在尝试修改其他用户的密码，那么会返回错误消息并渲染一个错误页面。通过这两个操作，可以保护用户的个人信息不被其他人修改。体现出了数据库的安全性。然后采用 django.contrib.auth 中的 PasswordChangeForm 来获取表单信息，并且通过 update_session_auth_hash 在保持用户登录的状态下，更新用户的密码，并且在数据库中保存用户密码的 hash 值，从而体现出了修改密码的安全性。

5.4.4 客户修改信息

```
@login_required
def bank_user_edit(request, user_id):
    user = User.objects.get(id=user_id)
    info = BankUser.objects.get(user_id=user_id)
    if user ≠ request.user and not request.user.is_superuser:
        messages.error(request, '无法修改他人信息')
        return render(request, 'frontend/error.html')

    if request.method ≠ 'POST':
        form = BankUserEditForm(instance=info)
    else:
        form = BankUserEditForm(instance=info, data=request.POST)
        if form.is_valid():
```

```
            info.id = form.cleaned_data.get("id")
            info.name = form.cleaned_data.get("name")
            info.tel = form.cleaned_data.get("tel")
            info.address = form.cleaned_data.get("address")
            info.branch = form.cleaned_data.get("branch")
            info.save()
            return redirect('accounts:accounts', user_id=user_id)

    context = {'form': form, 'user_id': user_id}
    return render(request, 'registration/edit.html', context)
```

与之前描述的相似，@login_required 和 user != request.user 可以确保用户个人信息不被他人修改。然后用户可以通过表单来修改其个人信息。

5.4.5 查看客户信息

```
@login_required
def get_users(request):
    branch_name = None
    if request.user.is_superuser:
        branch_name = request.user.username
    if branch_name:
        users_lists = BankUser.objects.filter(branch_id=branch_name)
        paginator = Paginator(users_lists, 4)
        page = request.GET.get('page')
        users = paginator.get_page(page)
        context = {'users': users}
        return render(request, 'registration/get_users.html', context)
    else:
        messages.error(request, '无法查看信息')
        return render(request, 'frontend/error.html')
```

这段代码根据用户身份决定是否显示银行用户列表。如果用户是超级用户，将根据用户所属的分行获取相应的用户列表，并在前端进行分页显示。如果用户不是超级用户，将显示一个错误页面，提示用户无法查看信息.

5.5 账户

```
class UserAccounts(models.Model):
    objects = models.Manager()
    user = models.ForeignKey(BankUser, on_delete=models.CASCADE,
related_name='UserAccounts')
    branch = models.ForeignKey(BankBranch, on_delete=models.CASCADE,
related_name='BranchAccounts', null=True)
```

```
    account_id = models.AutoField(primary_key=True)
    money = models.FloatField(default=0.0, validators=
[MinValueValidator(0.0)])
    create_date = models.DateTimeField(auto_now_add=True)
```

该 UserAccounts 模型定义了用户账户的各个属性，包括与银行用户和银行分行的关联，以及账户的唯一标识、金额和创建日期。这些字段将在数据库中作为表的列，用于存储和检索用户账户的数据。

5.5.1 创建账户

```
@login_required
def create_account(request, user_id):
    user = BankUser.objects.get(user_id=user_id)
    if request.user.id != user_id:
        messages.error(request, '无法为他人创建账户')
        return render(request, 'frontend/error.html')
    branch = BankBranch.objects.get(name=user.branch_id)
    if request.method != 'POST':
        form = UserAccountsForm(initial={'user': user, 'branch':
branch})
    else:
        form = UserAccountsForm(initial={'user': user, 'branch':
branch}, data=request.POST)
        if form.is_valid():
            money = form.cleaned_data.get("money")
            account = UserAccounts.objects.create(user=user,
branch=branch, money=money)
            account.save()
            # 触发器，自动更新用户的账户数
            user.counts = user.counts + 1
            user.save()
            # 账单
            bill = AccountBills.objects.create(account=account,
changes=money,
                                               type='收入', remark="创建
账户", money=money)
            bill.save()
            return redirect('accounts:accounts', user_id=user_id)
    context = {'form': form}
    return render(request, 'accounts/create_account.html', context)
```

首先，通过 user_id 获取对应的 BankUser 对象。检查当前登录用户的 ID 是否与 user_id 相等，以确定是否有权限为他人创建账户。如果没有权限，将显示错误消息并返回错误页面。获取与用户关联的银行分行对象。如果请求方法不是 POST，即第一次访问页面，将

创建一个空的账户表单对象，并显示在页面上。如果请求方法是 POST，即提交了表单数据，将根据提交的数据创建账户对象，并保存到数据库中。更新用户对象的账户数。创建账户变动的账单对象，并保存到数据库中。重定向到显示账户信息的页面。

总体来说，这段代码实现了通过表单创建用户账户的功能，包括权限验证、表单数据的验证和保存账户及相关信息到数据库中。

5.5.2 删除账户

```
@login_required
def delete_account(request, account_id):
    account = UserAccounts.objects.get(account_id=account_id)
    user = BankUser.objects.get(id=account.user_id)
    # judge if the user is the owner of the account
    if request.user.id ≠ user.user_id:
        messages.error(request, '无法删除他人账户')
        return render(request, 'frontend/error.html')
    if not account or account.money > 0:
        messages.error(request, '无法删除账户')
        return render(request, 'frontend/error.html')
    # 触发器，自动更新用户的账户数
    user.counts = user.counts - 1
    user.save()
    account.delete()
    return redirect('accounts:accounts', user_id=user.user_id)
```

首先，通过 account_id 获取对应的账户对象 UserAccounts 。通过账户对象获取关联的用户对象BankUser 。检查当前登录用户的 ID 是否与账户所属用户的 ID 相等，以确定是否有权限删除他人的账户。如果没有权限，将显示错误消息并返回错误页面。检查账户对象是否存在且账户余额是否大于零。如果账户不存在或账户余额大于零，则说明当前账户无法删除，将显示错误消息并返回错误页面。更新用户对象的账户数，将账户数减 1。保存更新后的用户对象到数据库。删除账户对象。重定向到显示用户账户信息的页面。

总体来说，这段代码实现了删除用户账户的功能，包括权限验证、账户余额检查、更新用户对象的账户数以及从数据库中删除账户对象。

5.5.3 查看账户

```
# show accounts information
@login_required
def accounts(request, user_id):
    # judge if the user is the owner of the account
    if request.user.id ≠ user_id and not request.user.is_superuser:
        messages.error(request, '无法查看他人账户')
```

```
        return render(request, 'frontend/error.html')
    account_user = BankUser.objects.get(user_id=user_id)
    accounts_lists =
UserAccounts.objects.filter(user_id=account_user.id)
    paginator = Paginator(accounts_lists, 4)
    page = request.GET.get('page')
    accounts_list = paginator.get_page(page)
    context = {'accounts': accounts_list, 'account_user': account_user}
    return render(request, 'accounts/accounts.html', context)
```

首先，检查当前登录用户的 ID 是否与 user_id 相等或是否为超级用户，以确定是否有权限查看他人的账户。如果没有权限，将显示错误消息并返回错误页面。通过 user_id 获取对应的 BankUser 对象，表示要查看账户的用户。通过用户对象获取与之关联的账户对象列表。使用分页器对账户列表进行分页处理，每页显示 4 个账户。从请求中获取当前页的页码。获取当前页的账户列表。创建一个包含账户列表和用户信息的上下文对象。渲染显示账户的页面，并将上下文对象传递给模板。

总体来说，这段代码实现了查看用户账户列表的功能，包括权限验证、获取用户对象、获取账户列表、分页处理以及将数据传递给模板进行页面渲染。

5.5.4 转账

```
# transfer money
@login_required
def transfer(request, account_id):
    account = UserAccounts.objects.get(account_id=account_id)
    user = BankUser.objects.get(id=account.user_id)
    # judge if the user is the owner of the account
    if request.user.id ≠ user.user_id:
        messages.error(request, '无法使用他人账户')
        return render(request, 'frontend/error.html')
    if request.method ≠ 'POST':
        form = AccountsTransferForm(initial={'account': account})
    else:
        form = AccountsTransferForm(initial={'account': account},
data=request.POST)
        if form.is_valid():
            money = form.cleaned_data.get("money")
            target_account = form.cleaned_data.get("target_account")
            # check if the account has enough money
            if account.money < money:
                messages.error(request, '余额不足')
                return render(request, 'frontend/error.html')
            # check if the target account exists
            if not
```

```
UserAccounts.objects.filter(account_id=target_account.account_id):
                messages.error(request, '目标账户不存在')
                return render(request, 'frontend/error.html')
            account.money = account.money - money
            account.save()
            # 账单
            remark = "转账给" + target_account.user.name + "的" +
str(target_account.account_id) + "账户"
            bill1 = AccountBills.objects.create(account=account,
changes=-money,  type='支出', remark=remark, money=account.money)
            bill1.save()
            target_account.money = target_account.money + money
            target_account.save()
            remark = "收到" + account.user.name + "的" +
str(account.account_id) + "账户转账"
            bill2 = AccountBills.objects.create(account=target_account,
changes=money, type='收入', remark=remark, money=target_account.money)
            bill2.save()
            return redirect('accounts:accounts', user_id=user.user_id)
    context = {'form': form, 'account': account}
    return render(request, 'accounts/transfer.html', context)
```

首先，通过 account_id 获取对应的账户对象 UserAccounts 。通过账户对象获取关联的用户对象BankUser 。检查当前登录用户的 ID 是否与账户所属用户的 ID 相等，以确定是否有权限使用他人的账户。如果没有权限，将显示错误消息并返回错误页面。如果请求方法不是 POST，即第一次访问页面，将创建一个空的转账表单对象，并显示在页面上。如果请求方法是 POST，即提交了表单数据，将根据提交的数据进行转账操作。验证表单数据的有效性。检查账户余额是否足够进行转账。检查目标账户是否存在。更新转出账户的余额，减去转账金额。创建转出账户的支出账单对象。更新目标账户的余额，增加转账金额。创建目标账户的收入账单对象。重定向到显示用户账户信息的页面。

总体来说，这段代码实现了账户间转账的功能，包括权限验证、表单数据验证、账户余额检查、更新账户余额、创建账单对象以及重定向到账户信息页面。

5.6 账单

```
class AccountBills(models.Model):
    bill_type = [
        ('收入', '收入'),
        ('支出', '支出'),
    ]
    objects = models.Manager()
    account = models.ForeignKey(UserAccounts, on_delete=models.CASCADE,
related_name='AccountBills')
    bill_id = models.AutoField(primary_key=True)
```

```
    changes = models.FloatField()
    type = models.CharField(max_length=100, choices=bill_type,
default='收入')
    remark = models.CharField(max_length=1000, default='', blank=True)
    money = models.FloatField(default=0)
    update_time = models.DateTimeField(auto_now_add=True)
```

该模型用于记录账户的收入和支出情况，包括金额变化、类型、备注信息以及更新时间。通过与UserAccounts 模型的关联，可以将账单与特定账户进行关联。

5.6.1 创建账单

```
@login_required
def create_bill(request, account_id):
    account = UserAccounts.objects.get(account_id=account_id)
    # judge if the user is the owner of the account
    user_id = BankUser.objects.get(id=account.user_id).user_id
    if request.user.id ≠ user_id:
        messages.error(request, '无法为他人账户创建账单')
        return render(request, 'frontend/error.html')
    if request.method ≠ 'POST':
        form = AccountBillsForm(initial={'account': account})
    else:
        form = AccountBillsForm(initial={'account': account},
data=request.POST)
        if form.is_valid():
            old_money = account.money
            changes = form.cleaned_data.get("changes")
            bill_type = form.cleaned_data.get("type")
            remark = form.cleaned_data.get("remark")
            new_money = old_money + changes
            if new_money < 0:
                messages.error(request, '余额不足')
                return render(request, 'frontend/error.html')
            bill = AccountBills.objects.create(account=account,
changes=changes,
                                               type=bill_type,
remark=remark, money=new_money)
            bill.save()
            # 触发器，自动更新账户余额
            account.money = new_money
            account.save()
            return redirect('bills:bills',
account_id=account.account_id)
```

```
    context = {'form': form, 'account': account}
    return render(request, 'bills/create_bill.html', context)
```

首先，通过 account_id 获取对应的账户对象 UserAccounts 。通过账户对象获取关联的用户对象BankUser 的 ID。检查当前登录用户的 ID 是否与账户所属用户的 ID 相等，以确定是否有权限为他人账户创建账单。如果没有权限，将显示错误消息并返回错误页面。如果请求方法不是 POST，即第一次访问页面，将创建一个空的账单表单对象，并显示在页面上。如果请求方法是 POST，即提交了表单数据，将根据提交的数据创建一个新的账单。验证表单数据的有效性。获取账户的当前余额和表单中的变化金额。根据变化金额计算账户的新余额。检查新余额是否小于 0，如果是，则显示错误消息并返回错误页面。创建一个新的账单对象，并保存到数据库中。更新账户对象的余额为新余额。重定向到显示账户账单的页面。

总体来说，这段代码实现了创建账单的功能，包括权限验证、表单数据验证、计算账户余额、创建账单对象、更新账户余额以及重定向到账单页面。

5.6.2 查看账单

```
@login_required
def bills(request, account_id):
    account = UserAccounts.objects.get(account_id=account_id)
    # judge if the user is the owner of the account
    user_id = BankUser.objects.get(id=account.user_id).user_id
    if request.user.id != user_id:
        messages.error(request, '无法查看他人账单')
        return render(request, 'frontend/error.html')
    bills_lists = AccountBills.objects.filter(account_id=account_id)
    paginator = Paginator(bills_lists, 4)
    page = request.GET.get('page')
    bills_list = paginator.get_page(page)
    context = {'bills': bills_list, 'account': account}
    return render(request, 'bills/bills.html', context)
```

首先，通过 account_id 获取对应的账户对象 UserAccounts 。通过账户对象获取关联的用户对象BankUser 的 ID。检查当前登录用户的 ID 是否与账户所属用户的 ID 相等，以确定是否有权限查看他人账单。如果没有权限，将显示错误消息并返回错误页面。根据 account_id 从数据库中获取与该账户关联的账单对象 AccountBills 列表。使用分页器（Paginator）对账单列表进行分页处理。获取请求中的页码。根据页码获取当前页的账单列表。构建上下文（context）字典，包含账单列表和账户对象。渲染账单页面，并将上下文传递给模板。

总体来说，这段代码实现了查看账单的功能，包括权限验证、获取账单列表、分页处理以及将数据传递给模板进行渲染。

5.7 贷款

```
class Loans(models.Model):
    loan_choices = [
        ('未还清', '未还清'),
        ('已还清', '已还清'),
    ]
    objects = models.Manager()
    loan_id = models.AutoField(primary_key=True)
    user = models.ForeignKey(BankUser, on_delete=models.CASCADE,
related_name='UserLoans')
    branch = models.ForeignKey(BankBranch, on_delete=models.CASCADE,
related_name='BranchLoans')
    money = models.FloatField(default=0.0, validators=
[MinValueValidator(0.0)])
    remain_money = models.FloatField(default=0.0, validators=
[MinValueValidator(0.0)])
    loan_date = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField(default=datetime.now() +
timedelta(days=365))
    status = models.CharField(max_length=100, choices=loan_choices,
default='未还清')
```

该模型表示贷款记录的基本信息，与用户和分行进行关联。通过 user 字段可以获取与贷款相关的用户信息，通过 branch 字段可以获取与贷款相关的分行信息。 money 字段表示贷款金额， remain_money 字段表示剩余贷款金额， loan_date 字段记录了贷款创建的日期和时间， due_date字段表示贷款的到期日期。 status 字段表示贷款的状态，可以选择 "未还清" 或 "已还清"。

5.7.1 申请贷款

```
@login_required
def apply_loan(request, user_id):
    user = BankUser.objects.get(user_id=user_id)
    if request.user.id ≠ user_id:
        messages.error(request, '无法为他人创建账户')
        return render(request, 'frontend/error.html')
    branch = BankBranch.objects.get(name=user.branch_id)
    if request.method ≠ 'POST':
        form = LoanForm(initial={'user': user, 'branch': branch})
    else:
        form = LoanForm(initial={'user': user, 'branch': branch},
data=request.POST)
        if form.is_valid():
```

```
            money = form.cleaned_data.get("money")
            loan = Loans.objects.create(user=user, branch=branch,
money=money, remain_money=money)
            loan.save()
            # 触发器，自动更新用户的状态
            user.status = False
            user.save()
            return redirect('loans:loans', user_id=user_id)
    context = {'form': form}
    return render(request, 'loans/apply_loan.html', context)
```

首先通过 user_id 获取用户对象。判断当前用户是否为所申请贷款的用户，如果不是则返回无权限错误页面。根据用户的分行ID获取分行对象。如果请求的方法不是 POST，则创建一个贷款申请表单LoanForm，并传入初始值为用户对象和分行对象。如果请求的方法是 POST，则创建一个贷款申请表单 LoanForm，并传入初始值为用户对象、分行对象和请求数据。验证表单数据是否有效，如果有效则创建一个贷款对象 Loans，并保存到数据库中。更新用户的状态为 False，表示用户已申请贷款。重定向到贷款列表页面，显示该用户的贷款信息。创建一个字典 context，将贷款申请表单存储在相应的键下。使用渲染函数将数据传递给名为 `loans/apply_loan.html` 的模板，并返回渲染后的 HTML响应。

该视图函数用于展示贷款申请页面，并处理贷款申请表单的提交。只有所申请贷款的用户才有权限访问该页面，成功提交贷款申请后将重定向到贷款列表页面。

5.7.2 删除贷款

```
@login_required
def delete_loan(request, loan_id):
    loan = Loans.objects.get(loan_id=loan_id)
    user = BankUser.objects.get(id=loan.user_id)
    # judge if the user is the owner of the account
    if request.user.id ≠ user.user_id:
        messages.error(request, '无法删除他人贷款')
        return render(request, 'frontend/error.html')
    if not loan or loan.status == '未还清':
        messages.error(request, '未还清贷款，无法删除贷款')
        return render(request, 'frontend/error.html')
    loan.delete()
    # if user has no loan, change the status to True
    if not Loans.objects.filter(user_id=user.id):
        user.status = True
        user.save()
    return redirect('loans:loans', user_id=user.user_id)
```

首先根据 loan_id 获取贷款对象。根据贷款对象的用户 ID 获取用户对象。判断当前用户是否为贷款的所有者，如果不是则返回无权限错误页面。判断贷款是否存在或状态为 '未还清'，如果是则返回相应的错误页面。然后删除贷款对象。如果用户没有其他贷款记录，则将用户的状态更新为 True，表示用户没有贷款。重定向到贷款列表页面，显示该用户的贷款信息。

该视图函数用于删除贷款，只有贷款的所有者才有权限删除。如果贷款不存在或状态为 '未还清'，则无法删除贷款。成功删除后将重定向到贷款列表页面。

5.7.3 还款

```
@login_required
def pay_loan(request, loan_id):
    loan = Loans.objects.get(loan_id=loan_id)
    user = BankUser.objects.get(id=loan.user_id)
    branch = BankBranch.objects.get(name=user.branch_id)
    accounts = UserAccounts.objects.filter(user_id=user.id)
    if request.user.id ≠ user.user_id:
        messages.error(request, '无法为他人还款')
        return render(request, 'frontend/error.html')
    if request.method ≠ 'POST':
        form = PayForm(initial={'loan': loan, 'user': user, 'branch':
branch})
    else:
        form = PayForm(initial={'loan': loan, 'user': user, 'branch':
branch}, data=request.POST)
        if form.is_valid():
            money = form.cleaned_data.get("money")
            account = form.cleaned_data.get("account")
            # if account has no enough money
            if account.money < money:
                messages.error(request, '账户余额不足')
                return render(request, 'frontend/error.html')
            loan.remain_money -= money
            if loan.remain_money < 0:
                messages.error(request, '还款金额超过贷款金额')
                return render(request, 'frontend/error.html')
            if loan.remain_money == 0:
                loan.status = '已还清'
            loan.save()
            # 修改账户余额
            account.money -= money
            account.save()
            remark = "还款" + str(loan.loan_id) + "号贷款"
            # 添加账单
```

```
            bill = AccountBills.objects.create(account=account,
changes=-money,
                                               type='支出',
remark=remark, money=account.money)
            bill.save()
            return redirect('loans:loans', user_id=user.user_id)
    context = {'form': form, 'loan': loan}
    return render(request, 'loans/pay_loan.html', context)
```

首先根据 loan_id 获取贷款对象，根据贷款对象的用户ID获取用户对象，根据用户对象的分行ID获取分行对象，根据用户ID获取用户的账户列表。然后判断当前用户是否为贷款的所有者，如果不是则返回无权限错误页面。如果请求的方法不是 POST，则创建一个还款表单 PayForm 并传入初始值为贷款、用户和分行对象。如果请求的方法是 POST，则创建一个带有请求数据的还款表单 PayForm 。接着验证表单数据是否有效，如果有效则执行还款操作：获取还款金额和账户信息，如果账户余额不足以支付还款金额，则返回错误页面。更新贷款剩余金额：如果贷款剩余金额小于0，则返回错误页面，如果贷款剩余金额等于0，则将贷款状态更新为 '已还清'。保存贷款对象，更新账户余额并且创建账单记录。最后重定向到贷款列表页面，显示该用户的贷款信息。

该视图函数用于进行贷款的还款操作，只有贷款的所有者才有权限进行还款。还款金额将从用户账户余额中扣除，并更新贷款的剩余金额和状态。成功删除后将重定向到贷款列表页面。

5.7.4 查看贷款

```
@login_required
def loans(request, user_id):
    user = BankUser.objects.get(user_id=user_id)
    if request.user.id ≠ user_id:
        messages.error(request, '无法查看他人贷款信息')
        return render(request, 'frontend/error.html')
    loans_lists = Loans.objects.filter(user_id=user.id)
    paginator = Paginator(loans_lists, 4)
    page = request.GET.get('page')
    loans_list = paginator.get_page(page)
    context = {'loans': loans_list, 'loan_user': user}
    return render(request, 'loans/loans.html', context)
```

首先根据 user_id 获取用户对象。然后判断当前用户是否为贷款信息的所有者，如果不是则返回无权限错误页面。根据用户ID获取该用户的贷款信息列表。使用分页器对贷款信息列表进行分页，并获取当前页数。将分页后的贷款信息列表和用户对象存储在上下文中。使用 renders 函数将数据传递给名为 `loans/loans.html` 的模板，并返回渲染后的 HTML 响应。

该视图函数用于显示用户的贷款信息列表，只有贷款信息的所有者才有权限查看。贷款信息通过分页显示，每页显示 4 条贷款记录。用户对象和分页后的贷款信息列表将传递给 `loans/loans.html` 模板进行渲染。

5.7.5 分行贷款

```
def branch_loans(request):
    name = None
    if request.user.is_superuser:
        name = request.user.username
    if name:
        loans_lists = Loans.objects.filter(branch_id=name)
        paginator = Paginator(loans_lists, 4)
        page = request.GET.get('page')
        loans_page = paginator.get_page(page)
        context = {'loans': loans_page}
        return render(request, 'loans/branch_loans.html', context)
    else:
        messages.error(request, '无法查看信息')
        return render(request, 'frontend/error.html')
```

首先获取当前分行的名称。如果当前用户是超级用户，则将用户名赋值给变量 name 。如果 name 不为空，则根据分行名称获取对应的贷款信息列表。使用分页器对贷款信息列表进行分页。获取当前页数。将分页后的贷款信息列表存储在上下文中。使用 render 函数将数据传递给名为'loans/branch_loans.html' 的模板，并返回渲染后的 HTML 响应。否则，返回无权限错误页面。

该视图函数用于显示某个分行的贷款信息列表。只有超级用户才有权限查看。

6 用户手册

6.1 系统启动

1. 在pycharm终端中确保安装了所需的包，例如 `pip install django mysqlclient`.
2. 连接到数据库 `db2024` （在settings.py中）。
3. 运行迁移：`python manage.py makemigrations` 和 `python manage.py migrate`
4. 创建超级用户：`python manage.py createsuperuser` (或者使用已经创建好的超级用户，这一步可以省略)。
5. 运行开发服务器：`python manage.py runserver`
6. 访问 Django 管理界面：`http://127.0.0.1:8000`

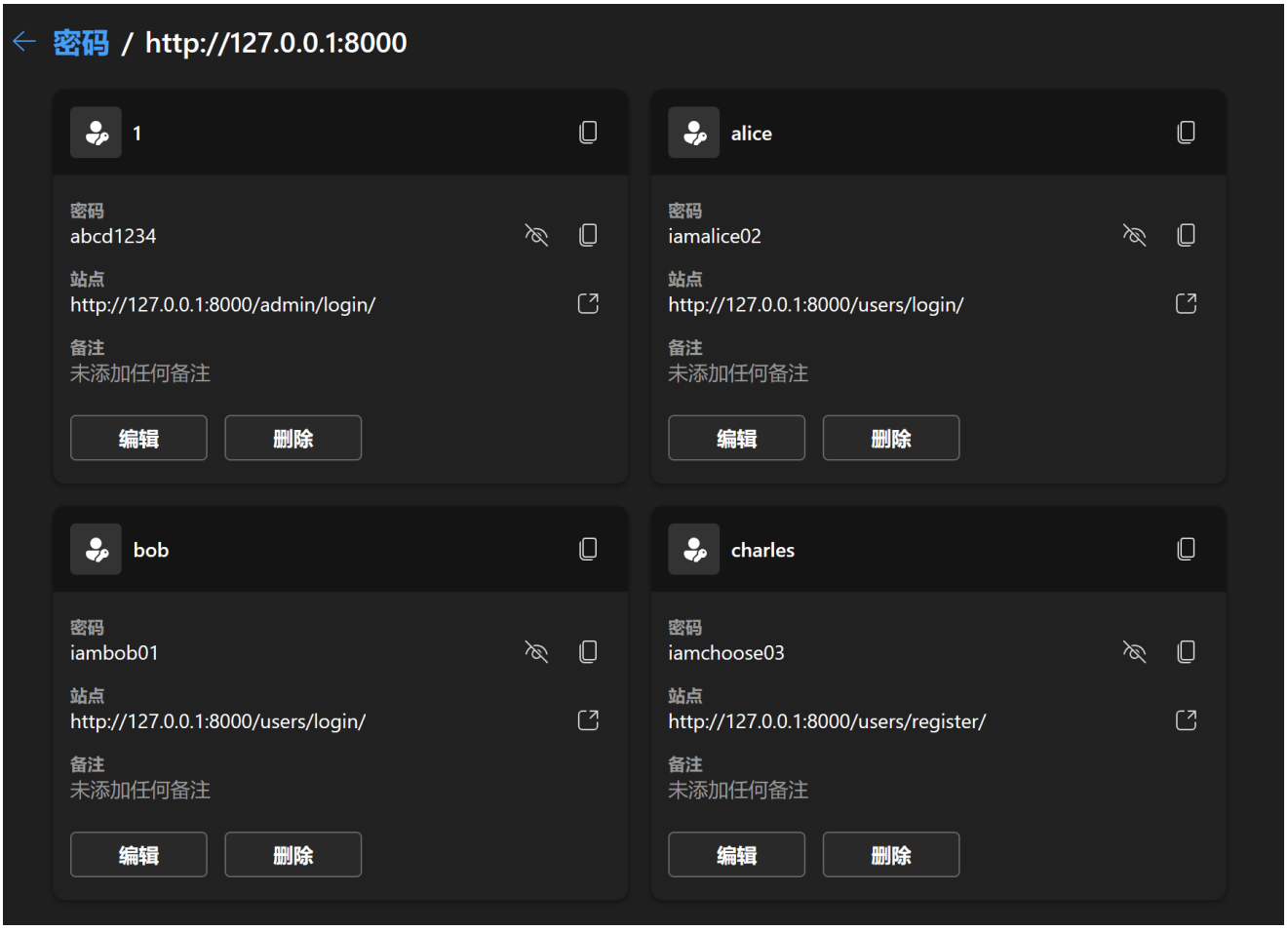
```
(base) D:\code\database\BankManagement
python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
June 22, 2024 - 22:22:54
Django version 5.0.6, using settings 'bank.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

6.2 用户登录

进入系统：

在调试过程中已经注册了如下账号，可以用于直接登录，也可以自己重新注册新账号。



其中1为超级用户，其他为普通用户。

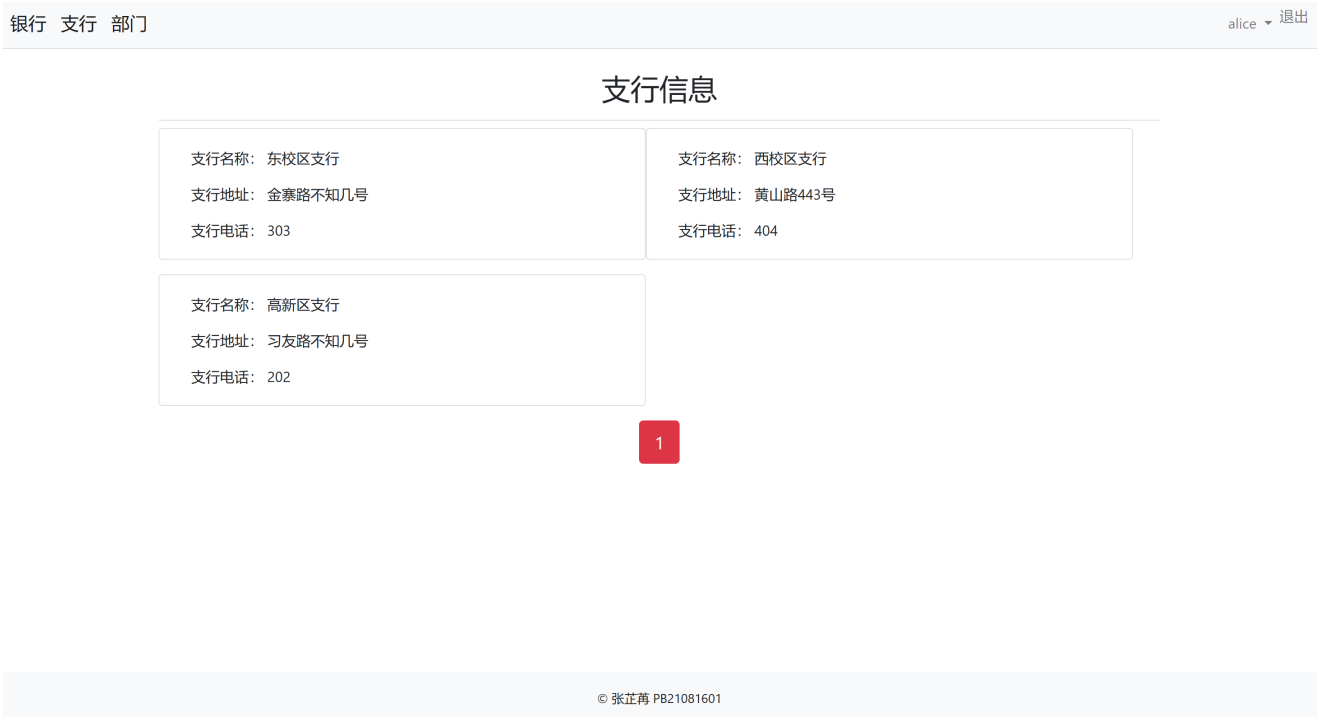
6.3 普通用户操作

在这里以alice为例，展示几个基本功能。

6.3.1 界面信息

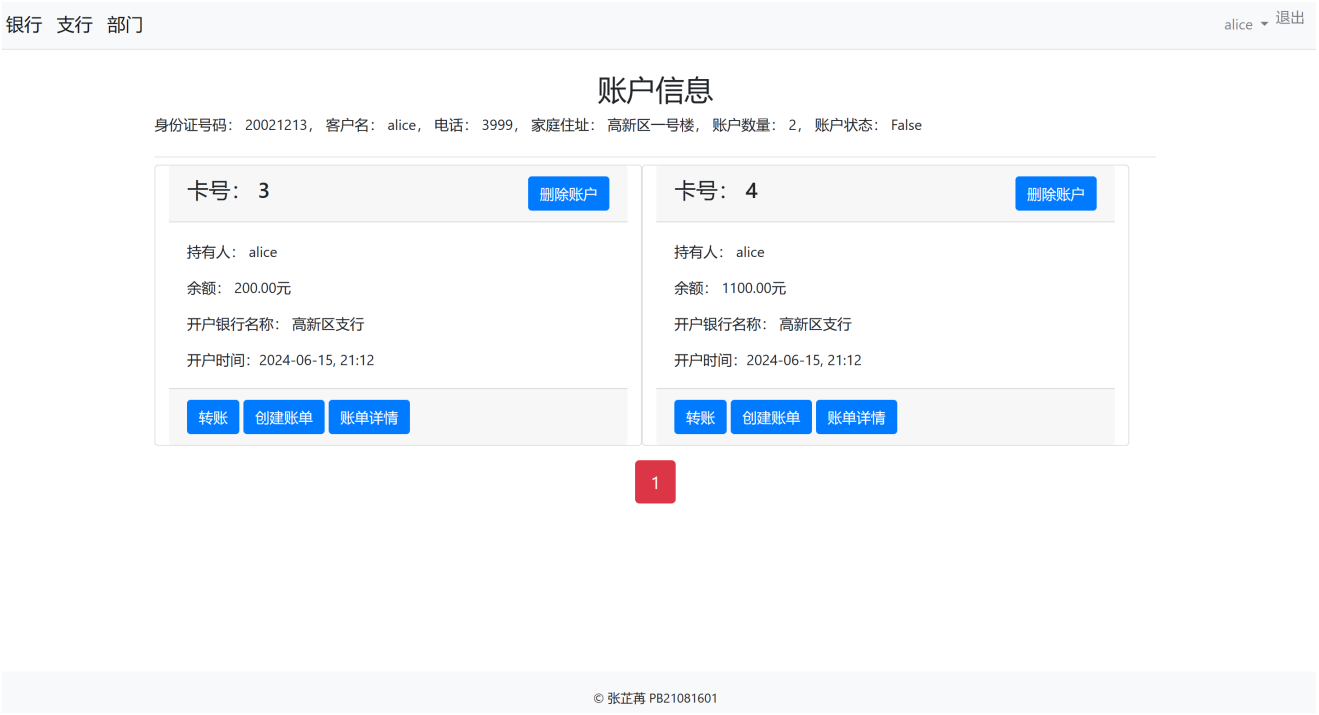
主界面展示：

用户alice可以看到银行、支行、部门的信息：



6.3.2 查看账单

alice可以查看自己的账户和每个账户下的流水：



6.3.3 借贷及还款

alice可以申请贷款，并还款：

银行 支行 部门

alice 退出

申请贷款

客户信息

20021213-alice

所属分行

高新区支行

借贷金额

借贷金额

申请

银行 支行 部门

alice 退出

贷款信息

身份证号码： 20021213， 客户名： alice， 电话： 3999， 家庭住址： 高新区一号楼， 账户数量： 2， 账户状态： False

贷款号： 2

删除贷款

申请人： alice

贷款金额： 600.00

批准银行： 高新区支行

申请时间：2024-06-15, 21:13

还款期限：2025-06-15, 21:10

未还金额： 0.00

贷款状态： 已还清

还款

1

© 张芷苒 PB21081601