

车辆租赁管理系统实验报告

林杰泓 22336137

刘艺凡 22336162

2025 年 1 月 2 日

引言

本任务关注设计车辆租赁管理系统，设计目的在于为用户和管理人员提供一个功能全面、操作便捷的车辆租赁平台，既能满足用户快速获取租车服务的需求，也能帮助管理人员在后台轻松实现对租车业务的高效管理。租车系统有三大核心的要求，车辆信息管理、租赁管理、客户管理。(1) 车辆信息管理要求记录和维护所有车辆的基本信息、租赁状态以及被租赁信息。(2) 租赁管理要求对租车订单的创建、查询、修改等功能，需要能够高效处理订单状态。(3) 客户管理要求对客户的基本信息、租车信息等数据进行管理，确保客户信息的安全性和准确性。此外，车辆租赁管理系统还需要提供良好的交互页面，注重用户交互的便捷，确保前端与数据库之间的数据交互高效可靠。同时，系统需要设计合理的权限管理机制，确保客户与管理员能够在各自权限范围内安全使用系统数据。最后，我们还实现了基于**时间戳排序协议的并发访问控制**并且在发生并发访问冲突时实现了简易的**恢复系统**。

在车辆租赁管理系统的开发中，我们采用 Django Web 框架作为后端技术栈，使用 PostgreSQL 作为数据库管理系统的后端，采用 HTML, CSS, JavaScript 框架开发前端交互页面。由于条件限制，本系统只支持本地部署。该系统开发人员为刘艺凡 (22336162) 和林杰泓 (22336137)。系统需求分析，结构设计及功能模块设计由本组开发人员共同分析设计。并由林杰泓负责用户信息管理和用户租赁管理的前端和后端开发和设计以及日志记录与审计等设计和测试，由刘艺凡负责车辆信息管理，车辆租赁信息管理和并发管理控制的前端和后端的设计和开发。系统交互页面的设计及优化由两人共同完成。同时，我们共同负责了系统的安全性设计，包括对用户，车辆信息的保护，以及对用户，管理员的权限设置及授权。

我们在附件中提供了开发系统的相关代码。同时，为了便于查看和了解该车辆租赁管理系统的成果，我们提供了 demo——**demo.gif**，展示了该系统的功能。此外，为了方便本地部署该系统，我们提供脚本 **init.sh** 用于构建简化对系统数据库的构建，同时也提供了 **README.md** 作为指引，讲述部署系统以及使用系统的方法。更进一步的，当系统部署成功后，我们可以通过访问链接<http://127.0.0.1:8000/management/register>进入系统的注册页面，以用户的视角开始体验该车辆租赁管理系统。

目录

1 实验题目	4
2 概要设计	4
2.1 需求分析	4
2.1.1 功能需求	4
2.1.2 非功能需求	4
2.2 系统结构	4
2.3 系统功能模块	5
3 详细设计	6
3.1 E-R 图	6
3.1.1 实体及属性	7
3.1.2 实体之间的关系	7
3.2 数据库模式	7
3.2.1 表设计	9
3.2.2 数据库模式特点	10
3.3 车辆信息管理模块设计与开发	10
3.3.1 实现流程	10
3.3.2 主要算法	11
3.4 租赁管理模块设计与开发	15
3.4.1 实现流程	16
3.4.2 租车功能的算法实现	16
3.4.3 还车功能的算法实现	18
3.5 客户管理模块设计与开发	19
3.5.1 用户注册功能 (Register)	19
3.5.2 用户登录功能 (Login)	20
3.5.3 系统中的用户信息记录	21
3.6 安全性与完备性	23
3.6.1 用户认证与权限管理	23
3.6.2 数据完整性约束	24
3.6.3 数据保护与加密	24
3.6.4 日志记录与审计	24
3.6.5 最小权限原则	26
4 调试与运行结果	26
4.1 问题与调试	26

4.1.1	request.user 为匿名用户	26
4.1.2	Customer 模型与 User 模型的关系未正确配置	27
4.1.3	用户信息在模板中无法访问	27
4.1.4	@login_required 装饰器没有正确应用	28
4.2	事务的并发访问控制与恢复	29
4.2.1	问题分析	29
4.2.2	并发访问控制	29
4.2.3	恢复系统	32
4.3	最终结果展示	32
4.3.1	注册与登录	33
4.3.2	用户首页	34
4.3.3	租车查询	34
4.3.4	并发访问控制	35
5	总结	36

1 实验题目

设计一个车辆租赁管理系统，包括车辆信息管理、租赁管理、客户管理等功能。车辆信息管理负责车辆信息的添加、修改和查询；租赁管理负责租赁信息的录入、修改和查询；客户管理负责客户信息的添加、修改和查询。

2 概要设计

2.1 需求分析

2.1.1 功能需求

- 车辆信息管理：包括车辆的添加、修改、查询。每辆车有编号、品牌、型号、车牌号、租金等信息。
- 租赁管理：包括租赁记录的管理，租赁客户、租赁车辆等。
- 客户管理：管理客户信息，包含客户 ID、姓名、联系方式等。

2.1.2 非功能需求

- 安全性：对用户的权限进行控制，确保只有管理员可以进行修改操作。
- 系统响应时间：保证系统能快速响应用户操作，尤其是查询操作。
- 界面友好性：设计直观的用户界面，保证用户能够方便地操作和管理信息。

2.2 系统结构

车辆租赁管理系统采用分层架构设计，主要分为表现层、业务逻辑层和数据访问层，各层次的功能和作用如下：

- 表现层（前端）：

表现层是系统与用户交互的部分，负责用户界面的显示和操作处理，包括车辆信息的查询、客户信息录入、租赁信息的修改等功能。具体技术采用 HTML、CSS 等前端技术，确保界面美观和用户体验友好。

- 业务逻辑层（后端）：

业务逻辑层位于表现层与数据访问层之间，是系统功能实现的核心部分，主要负责接收前端请求、处理业务逻辑并与数据库交互。车辆管理、租赁管理和客户管理的功能均在该层实现。本系统采用 Django 框架开发业务逻辑层，支持高效的请求响应和模块化开发。

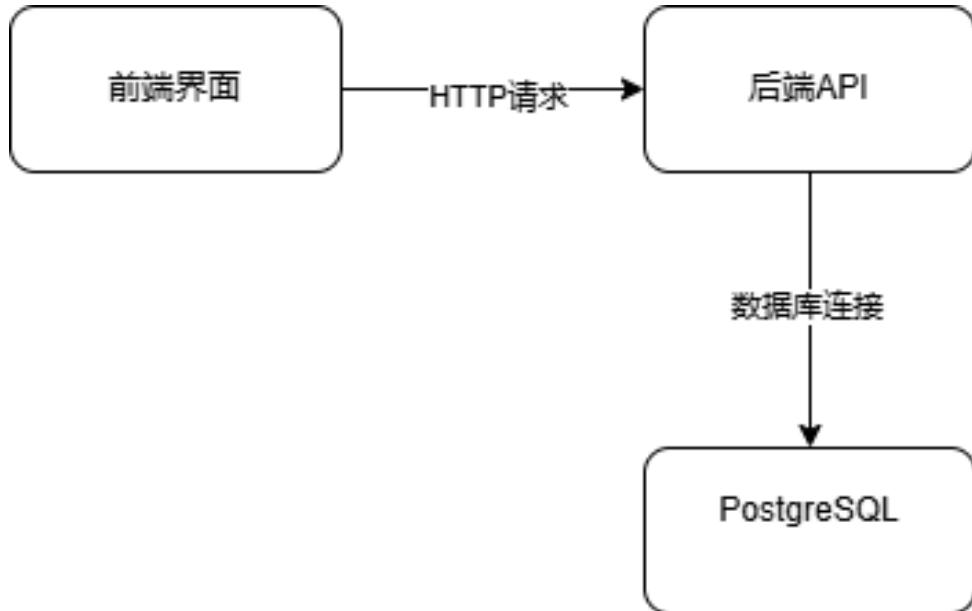


图 1: 系统结构图

- 数据访问层（数据库）：

数据访问层是系统的数据存储和管理部分，负责与数据库交互，执行 SQL 查询操作以实现数据的增删改查。本系统选用 PostgreSQL 作为数据库管理系统，设计了满足第三范式（3NF）的数据库结构，确保数据存储的规范性和完整性。

各层通过接口进行数据交互：表现层将用户请求发送给业务逻辑层，业务逻辑层根据功能需求调用数据访问层与数据库交互，最终返回结果至表现层供用户查看和操作。这种分层设计提升了系统的可维护性和扩展性。

针对需求分析，我们得到了系统结构图，如图1所示。

2.3 系统功能模块

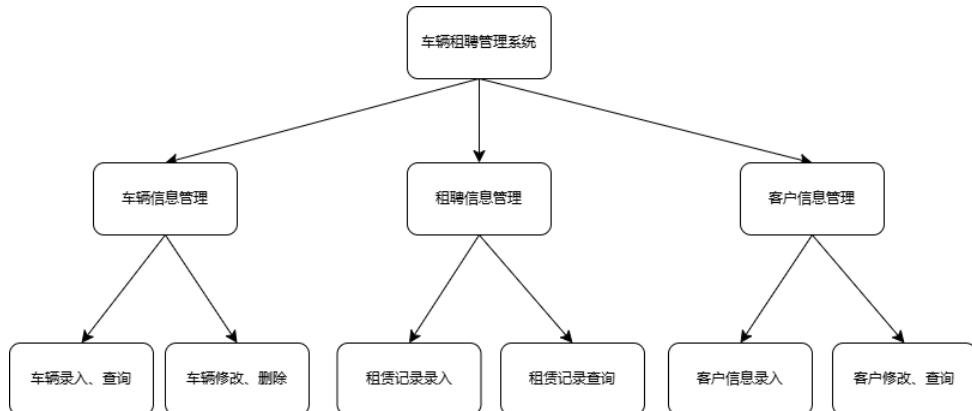


图 2: 系统功能模块图

车辆租赁管理系统的功能模块图以系统需求为基础，分为三个主要功能模块：车辆信息管理模块、租赁管理模块和客户信息管理模块。各模块的功能和相互关系描述如下：

- 车辆信息管理模块：实现车辆信息的添加、修改等功能。
添加功能用于录入车辆基本信息，如车辆编号、车型、租赁价格等。
修改功能用于更新车辆状态或属性，如车辆的租赁状态、等。
- 租赁管理模块：实现租赁交易的管理，包括租赁信息的录入、修改和查询。
录入功能记录租赁交易信息，如租赁车辆、客户、起始日期、结束日期及费用等。
修改功能允许调整租赁的相关信息，如租赁时间或费用的变更。
查询功能支持客户查找租赁记录，方便查看历史交易。
- 客户信息管理模块：实现客户信息的管理，包括添加、修改和查询功能。
添加功能用于录入新客户的信息，如姓名、联系方式、身份证号等。
修改功能用于更新客户信息，如更改联系方式或地址。
查询功能支持按客户姓名或其他条件检索客户信息，便于用户管理客户数据。

针对需求分析，我们得到了系统的功能模块图，如图2所示。

3 详细设计

3.1 E-R 图

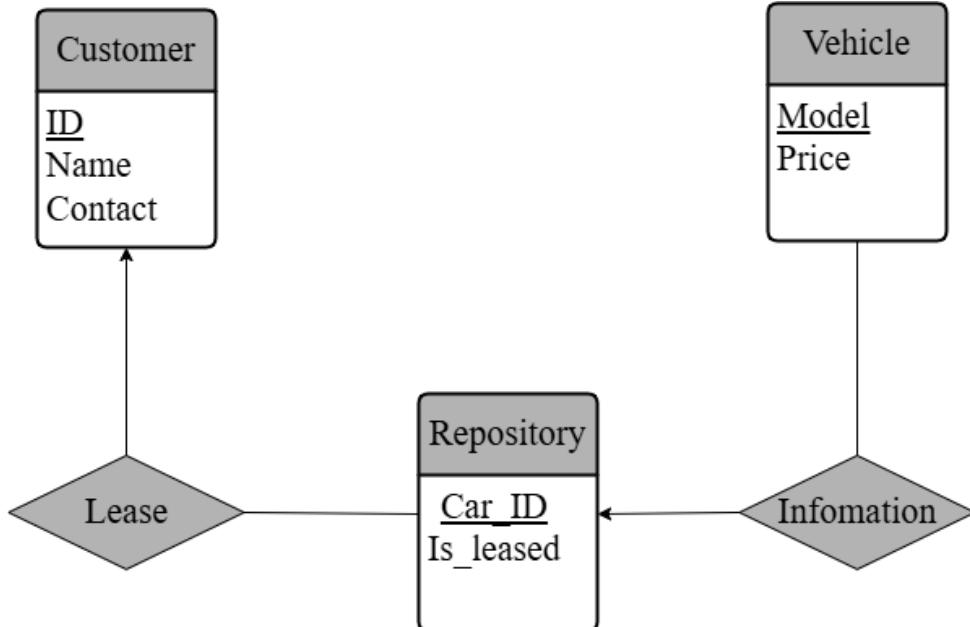


图 3: E-R 图

车辆租赁管理系统的 E-R 图描述了系统中的主要实体及其之间的关系，反映了系统的数据结构和逻辑关系 [1]。根据系统需求分析，设计了以下实体及其属性：

3.1.1 实体及属性

- 客户 (Customer) 属性:

用户 (User): 与 Django 的内置用户模型进行一对一关联，扩展用户信息。

ID (主键): 客户唯一标识符。

姓名 (name): 客户姓名。

联系方式 (contact): 客户的联系方式。

- 车辆 (Vehicle) 属性:

型号 (Model, 主键): 车辆的唯一标识。

价格 (Price): 车辆的租赁价格。

- 车辆库 (Repository) 属性:

车辆 ID (Car_ID, 主键): 标识库中每辆车的唯一编号。

是否租赁 (Is_leased): 表示车辆是否已被租赁。

- 车辆信息 (Info) 属性:

车辆 ID (Car_ID, 外键): 引用车辆库中的车辆 ID。

型号 (Model, 外键): 引用车辆表中的型号信息。

- 租赁记录 (Lease) 属性:

车辆 ID (Car_ID, 外键): 关联车辆库中的车辆。

客户 ID (ID, 外键): 关联客户表中的客户。

3.1.2 实体之间的关系

- 客户与租赁记录: 一个客户可以进行多次租赁记录 (一对多)。
- 车辆与车辆库: 车辆库中的每辆车属于一个车辆型号 (多对一)。
- 车辆库与租赁记录: 每辆车在租赁记录中最多出现一次 (一对一)。
- 车辆库与车辆信息: 每辆车在车辆信息表中有对应的详细记录 (一对一)。

针对需求分析，画出 E-R 图表示的概念模型，如图3所示。

3.2 数据库模式

首先我们这个关系模型是满足 3NF 的，即第三范式。具体满足了一下条件：

满足第一范式 (1NF)

- 所有字段的值都是原子值，没有重复的组或数组。比如，`Customer` 类中的字段 `name` 和 `contact` 都是单一值，没有包含集合或数组。
- 每一列的值都是不可分割的。

满足第二范式（2NF）

- 在满足第一范式的基础上，所有非主键属性完全依赖于主键。
- 例如，`Customer` 类中的 `name` 和 `contact` 完全依赖于主键 `ID`。这个表中的每个字段都与主键 `ID` 相关，而不是仅仅部分依赖。
- 在 `Lease` 类中，`Car_ID` 和 `ID` 外键的组合确保了每条记录唯一地标识了一个租赁。`ID` 外键依赖于 `Customer` 的 `ID`，而 `Car_ID` 外键依赖于 `Repository` 的 `Car_ID`。

满足第三范式（3NF）

- 除了满足第二范式外，还要求所有非主键字段必须直接依赖于主键，而不能依赖于其他非主键字段。
- 在 `Vehicle` 类中，字段 `Model` 是唯一标识一个车辆类型的字段，字段 `Price` 完全依赖于 `Model`。所以，`Vehicle` 表中的每个字段直接依赖于主键 `Model`。
- `Repository` 类中的 `Car_ID` 是主键，并且字段 `Is_leased` 完全依赖于主键 `Car_ID`，没有任何传递依赖。
- `Info` 类通过外键将 `Car_ID` 和 `Model` 与 `Repository` 和 `Vehicle` 表关联。`Car_ID` 和 `Model` 作为外键，确保了数据表之间的关联关系符合规范。
- 在 `Lease` 类中，`Car_ID` 和 `ID` 外键分别引用了 `Repository` 和 `Customer` 表，确保了租赁关系的唯一性，而不会引入冗余数据。

具体范式分析

- Customer:**
 - 主键：`ID`，所有字段（`name` 和 `contact`）直接依赖于主键，符合第二范式。
 - 该表没有任何字段间的传递依赖，符合第三范式。
- Vehicle:**
 - 主键：`Model`，字段 `Price` 完全依赖于 `Model`，没有传递依赖，符合第二范式和第三范式。
- Repository:**

- 主键: Car_ID, 字段 Is_leased 完全依赖于 Car_ID, 符合第二范式和第三范式。

- **Info:**

- 外键 Car_ID 引用 Repository 表, 外键 Model 引用 Vehicle 表。所有字段依赖于外键, 不存在传递依赖, 符合第三范式。

- **Lease:**

- 外键 Car_ID 引用 Repository, 外键 ID 引用 Customer。这些字段是唯一标识租赁记录的, 因此不会引入冗余数据, 符合第三范式。

该设计通过外键建立了表与表之间的关系, 并且每个表中的字段都直接依赖于主键, 避免了冗余数据, 符合第三范式的要求。

根据车辆租赁管理系统的具体需求和 E-R 图, 将概念模型转换为关系模型, 设计出满足第三范式(上面已经给予证明)的数据库模式。具体如下:

3.2.1 表设计

表 1: 客户表 (Customer)

字段名	数据类型	约束	说明
ID	VARCHAR(10)	PRIMARY KEY	客户唯一标识符
user_id	INTEGER	UNIQUE, NOT NULL	关联 Django 用户表
name	VARCHAR(50)	NOT NULL	客户姓名
contact	VARCHAR(15)	NOT NULL	客户联系方式

表 2: 车辆表 (Vehicle)

字段名	数据类型	约束	说明
Model	VARCHAR(50)	PRIMARY KEY	车辆型号
Price	INTEGER	NOT NULL	租赁价格

表 3: 车辆库表 (Repository)

字段名	数据类型	约束	说明
Car_ID	VARCHAR(10)	PRIMARY KEY	车辆唯一标识符
Is_leased	BOOLEAN	DEFAULT FALSE	是否被租赁

表 4: 车辆信息表 (Info)

字段名	数据类型	约束	说明
Car_ID	VARCHAR(10)	FOREIGN KEY (Repository.Car_ID), NOT NULL	关联车辆库表
Model	VARCHAR(50)	FOREIGN KEY (Vehicle.Model), NOT NULL	关联车辆型号

表 5: 租赁记录表 (Lease)

字段名	数据类型	约束	说明
ID	VARCHAR(10)	FOREIGN KEY (Customer.ID), NOT NULL	关联客户表
Car_ID	VARCHAR(10)	FOREIGN KEY (Repository.Car_ID), NOT NULL	关联车辆库表

3.2.2 数据库模式特点

1. 满足第三范式 (3NF)

- 消除冗余：每个表只存储一种实体的属性，避免数据冗余。
- 数据依赖明确：所有非主键字段完全依赖于主键。

2. 完整性约束

- 主键约束：每个表都有明确的主键，保证数据唯一性。
- 外键约束：通过外键建立表间关系，保证数据一致性。

3. 扩展性

- 模块化表设计便于功能扩展，如增加车辆维修管理、客户等级管理等功能。

3.3 车辆信息管理模块设计与开发

车辆信息管理模块需要维护数据中的所有车辆，同时还需要提供良好的交互界面，便于用户使用和了解车辆信息。对此，该模块的前端设计需要提供方便客户查询车辆信息和租赁的交互界面，而后端则需要维护车辆的信息。该模块的总体设计如图 4所示，在前端，用户可以方便地查看仓库中车辆的品牌，在了解相应的车品牌后，可以查看该品牌相关的车辆并了解车辆的信息。而后端则需要对用户的行为做出合理的反应，即使反馈用户所查看的信息。

3.3.1 实现流程

当用户进入交互页面后，用户可以在侧边栏点击租车键，然后租车键下会展开车辆品牌信息，供用户查看。这些展开的车辆品牌也是可交互的，用户可以点击感兴趣的品 牌进一步查看相应品牌的车辆有哪些，然后查看车辆的具体信息。而用户的一系列点击行为，在后端会引发相应的数据库查询。

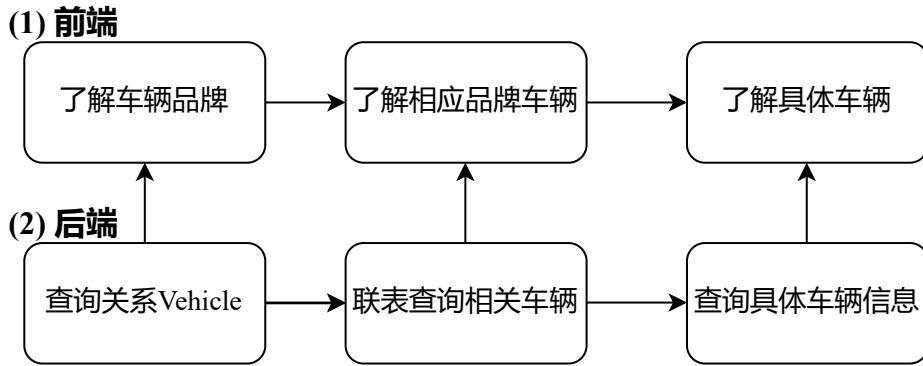


图 4: 车辆信息管理模块图

3.3.2 主要算法

实现这一模块的核心思想在于，通过用户点击 HTML 页面中的相关按钮，触发相应的 JavaScript 操作，而 JavaScript 函数进一步通过指定的 URL 调用后端服务以执行相应的查询逻辑。

在 HTML 文件中，添加一个租车标签，并设置 onclick 属性，用来指定点击该元素时要执行的 JavaScript 函数 fetchVehicles(event)，通过该函数动态填充车辆品牌列表。

```

1 <li>
2     <a href="#" onclick="fetchVehicles(event)">租 车 </a>
3     <ul id="vehicle-list" style="padding-left: 20px; display:
4         none;"> </ul>
5 </li>

```

编写相关的 JavaScript 函数 fetchVehicles(event) 查询相应的车辆品牌。在函数中，我们通过 URL 连接后端端口得到车辆品牌，然后再函数中填充车辆品牌列表。

```

1 function fetchVehicles(event) {
2     event.preventDefault();
3     fetch('/management/get-vehicles/')
4         .then(response => response.json())
5         .then(data => {
6             const vehicleList = document.getElementById('
7                 vehicle-list');
8             vehicleList.innerHTML = '';
9
10            if (data.status === "success") {
11                let html = '';
12                data.data.forEach(vehicle => {

```

3 详细设计

```
13             <a href="#" onclick="
14                 fetchVehicleDetails('${vehicle.Model}')">
15                 ${vehicle.Model}
16             </a>
17         </li>`;
18     );
19     vehicleList.innerHTML = html;
20     vehicleList.style.display = 'block'; // 显示车
21     辆列表
22 } else {
23     vehicleList.innerHTML = '<li>暂无车辆信息</li>';
24     ;
25     vehicleList.style.display = 'block';
26 }
27 )
28 .catch(error => {
29     console.error("Error fetching vehicles:", error);
30 });
31 }
```

在后端 views.py 中编写该函数对应的相关行为，即 select Model from Vehicle 并返回给前端。

```
1 def get_vehicles(request):
2     vehicles = Vehicle.objects.all()
3     if vehicles.exists():
4         data = list(vehicles.values("Model"))
5         return JsonResponse({"status": "success", "data": data})
6     else:
7         return JsonResponse({"status": "empty", "message": "暂
8         无车辆信息"})
```

为了使用户可以根据车辆品牌了解车辆信息，我们需要将这些车辆品牌的标签设置为可跳转可查询的标签。在函数 fetchVehicles 中，我们在车辆品牌的标签处添加 onclick 行为，使其在被点击后可以反馈用户相关的车辆信息。

```
1 <a href="#" onclick="fetchVehicleDetails('${vehicle.Model}')">
2     ${vehicle.Model}
```

3 详细设计

3

在点击车辆品牌后，网页会触发 JavaScript 函数 fetchVehicleDetails，这个函数会根据点击的车辆品牌返回相关的车辆和车辆信息。这个函数的实现也是类似的，通过访问相关的 URL 然后触发后端的数据库查询然后得到相关数据。

```
1 function fetchVehicleDetails(name) {
2
3   fetch(`/management/get-vehicle-details/?name=${name}`)
4     .then(response => response.json())
5     .then(data => {
6       const container =
7         document.getElementById('vehicle-details');
8       container.innerHTML = '';
9
10      if (data.status === "success") {
11        let html = '';
12        data.data.forEach(item => {
13          html += `
14            <tr>
15              <td>${item.Car_ID}</td>
16              <td>
17                ${item.Is_leased ? "已租赁" : "可租赁"}
18              </td>
19              <td>¥ ${item.Price}</td>
20              <td>
21                ${item.Is_leased
22                  ? '<button disabled class="action-button" style="background-color: gray; cursor: not-allowed;">已租赁
23                  </button>'
24                  : `<button class="action-button" onclick="leaseVehicle
25                    ('${item.Car_ID}', '${name}')">租赁 </button>`}
26              </td>
27            </tr>
28          `;
29        });
30        container.innerHTML = html;
31      } else {
```

3 详细设计

```
30     container.innerHTML = ` 
31         <tr>
32             <td colspan="4" style="color: red;">${data.
33                 message}</td>
34         </tr>
35     `;
36 }
37 .catch(error => {
38     console.error("Error fetching vehicle details:", error)
39     ;
40 });
41 }
```

在后端，后端根据点击的车辆品牌 name 进行数据库查询，即发生如下查询行为：

```
1 SELECT a.Car_ID, a.Is_leased, b.Price
2 FROM Repository a, Vehicle b
3 JOIN Info c ON b.Model = c.Model_id
4 WHERE a.Car_ID = c.Car_ID_id AND b.Model = name
```

而我们在 views.py 需要将这个查询嵌入到 python 函数之后，具体实现如下

```
1 def get_vehicle_details(request):
2     if request.method == "GET":
3         car_name = request.GET.get("name")
4         if not car_name:
5             return JsonResponse({"status": "error", "message": "车辆名称不能为空"})
6
7         query = """
8             SELECT a."Car_ID", a."Is_leased", b."Price"
9             FROM management_repository a, management_vehicle b
10            JOIN management_info c ON b."Model" = c."Model_id"
11           WHERE a."Car_ID" = c."Car_ID_id" AND b."Model" = %s
12           """
13
14         with connection.cursor() as cursor:
15             cursor.execute(query, [car_name])
16             rows = cursor.fetchall()
```

```
16
17     data = [
18         {"Car_ID": row[0], "Is_leased": row[1], "Price": row[2]}
19         for row in rows
20     ]
21
22     if data:
23         return JsonResponse({"status": "success", "data": data})
24     else:
25         return JsonResponse({"status": "empty", "message": "未找到车辆详情"})
```

至此，我们实现了车辆信息管理模块。

3.4 租赁管理模块设计与开发

租赁管理模块需要在交互界面实现租车和还车功能，这一功能需要便于客户的使用。同时，当客户进行租赁行为时，后端需要即使更新车辆租赁的状态，并将新的车辆租赁信息显示到用户交互页面。总体设计如图 5 所示，

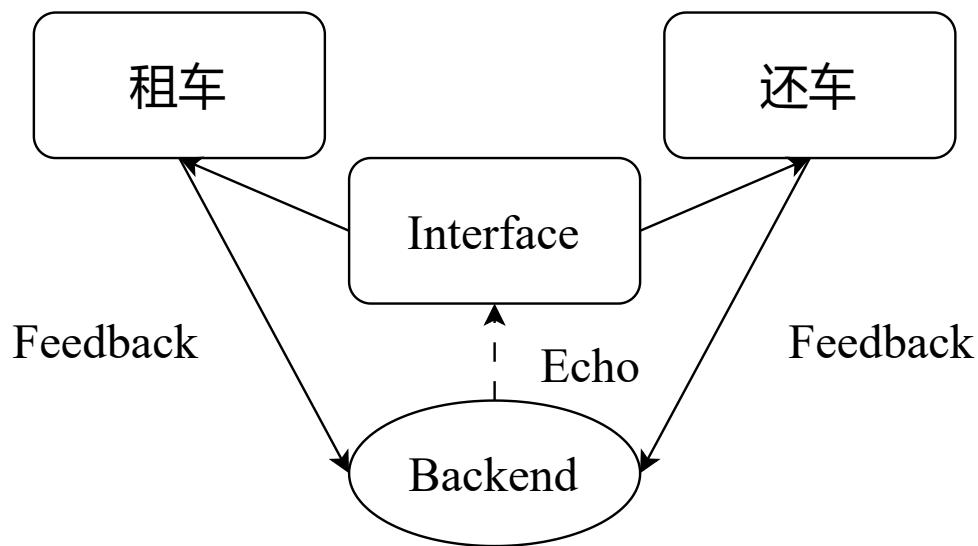


图 5: 用户管理模块图

3.4.1 实现流程

客户在展示车辆详细信息的页面中可以点击租车进行租车操作，然后可以在客户个人首页中点击还车按键进行还车操作。这些客户的行为会触发后端对数据库中车辆的租赁状态进行更新并反馈到交互页面中。然而，由于这个模块是多用户并发访问的，所以如果不对租车事务加以控制则会带来事务的隔离性和原子性不能保证的问题。在 4.2 一节中，我们针对这个问题进行了详细的讨论。

3.4.2 租车功能的算法实现

在车辆详细信息处添加租车按键，当点击按键时会触发 JavaScript 函数 leaseVehicle 进行租车行为。同时，对于被租车辆需要禁止租车行为的发生。

```

1 ${item.Is_leased
2 ? '<button disabled class="action-button" style="background-
  color: gray; cursor: not-allowed;">已租赁</button>'
3 : `<button class="action-button" onclick="leaseVehicle('${item.
  Car_ID}', '${name}')">租赁</button>`}
```

对于函数 leaseVehicle，当用户在交互页面进行租车行为触发该函数时，该函数会向租车 URL 提交一个表单，传递租车的客户和租赁的具体车辆。当数据库更新成功后将刷新页面，更新交互页面的信息显示。

```

1 function leaseVehicle(carId, name) {
2   fetch('/management/lease-vehicle', {
3     method: 'POST',
4     headers: {
5       'Content-Type': 'application/json',
6       'X-CSRFToken': getCookie('csrftoken')
7     },
8     body: JSON.stringify({ car_id: carId })
9   })
10   .then(response => response.json())
11   .then(data => {
12     if (data.status === "success") {
13       fetchVehicleDetails(name);
14     } else {
15       alert(data.message);
16     }
17   })
18   .catch(error => {
```

3 详细设计

```
19         console.error("Error leasing vehicle:", error);
20         alert("租赁失败，请稍后再试。");
21     });
22 }
```

在后端，当其接收到客户提交的表单时，后端根据提交的客户信息 ID 和车辆信息 CarID 更新数据库中的关系，即 select:

```
1 UPDATE Repository
2 SET Is_leased = TRUE
3 WHERE Car_ID = CarID;
4 INSERT INTO Lease (ID,Car_ID)
5 VALUES (ID, CarID);
```

最后将这个 SQL 语句嵌入到 view.py 的相应处理函数中：

```
1 @login_required
2 def lease_vehicle(request):
3     if request.method == "POST":
4         try:
5             data = json.loads(request.body)
6             car_id = data.get("car_id")
7             user_id = request.user.username
8             if not car_id:
9                 return JsonResponse({"status": "error", "message": "车辆ID不能为空"})
10
11             update_query = """
12                 UPDATE management_repository
13                     SET "Is_leased" = TRUE
14                     WHERE "Car_ID" = %s
15             """
16
17             insert_query = """
18                 INSERT INTO management_lease ("ID_id",
19                                         "Car_ID_id")
20                     VALUES (%s, %s)
21             """
22             print(car_id)
```

3 详细设计

```
23     with connection.cursor() as cursor:
24         cursor.execute(update_query, [car_id])
25         cursor.execute(insert_query, [user_id, car_id])
26
27
28     return JsonResponse({"status": "success", "message":
29                         : "车辆租赁成功!"})
30
31     except Exception as e:
32         print(f"Error leasing vehicle: {e}")
33         return JsonResponse({"status": "error", "message":
34                         : "租赁失败，请稍后再试。"})
35
36
37     return JsonResponse({"status": "error", "message": "无效的
38                         请求方法"})
```

3.4.3 还车功能的算法实现

系统尝试查找与当前登录用户相关联的租赁记录，查找条件包括：车辆 ID 和用户 ID。如果没有找到匹配的租赁记录（即用户没有租过这辆车），则抛出 Lease.DoesNotExist 异常，并向用户显示错误信息，提示其没有租用此车辆，并将其重定向到主页。

```
1   try:
2       lease = Lease.objects.get(Car_ID__Car_ID=vehicle_id,
3                                   ID__user=request.user)
4   except Lease.DoesNotExist:
5       messages.error(request, "You have not rented this
6                       vehicle.")
7   return redirect('homepage')
```

通过 lease 对象获取到租赁记录中关联的仓库车辆对象 Car_ID，然后将 Is_leased 字段更新为 False，表示该车辆已经被归还并且重新可用。之后调用 save() 方法保存更新。

```
1   repository = lease.Car_ID
2   repository.Is_leased = False # 标记车辆为未租赁
3   repository.save()
```

删除租赁记录，以便用户不再与这辆车有关联，并且租赁记录从数据库中移除。

```
1   lease.delete()
```

3.5 客户管理模块设计与开发

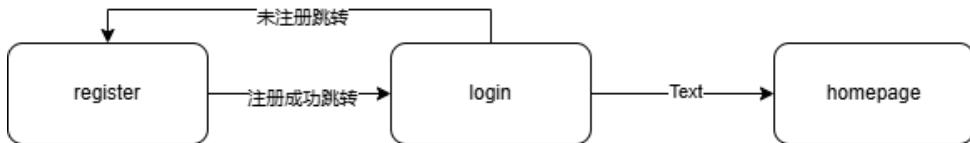


图 6: 用户管理模块图

总体设计如图6所示。

客户管理模块包括用户注册、登录、车辆租赁等核心功能，旨在为系统提供完整的客户管理与操作流程。

3.5.1 用户注册功能 (Register)

注册功能实现了用户创建账户的操作，并且在创建用户时，将其信息保存到自定义的 Customer 表中，确保系统能够关联用户和客户之间的关系。

实现流程:

- **GET 请求:** 用户访问注册页面时，渲染 register.html 模板。
- **POST 请求:** 用户提交注册表单数据时，首先进行简单校验（确保所有字段填写）。如果数据不完整或者用户名已存在，返回错误提示。
- 然后，使用 Django 的 `transaction.atomic()` 来保证数据的一致性，在同一个事务中创建 User 和 Customer 实例。
- 如果一切正常，返回成功信息并重定向到登录页面。

主要算法:

- **字段校验:** 确保用户输入的所有字段有效。
- **事务控制:** 通过 `transaction.atomic()` 保证 User 和 Customer 的数据一致性。
- **异常捕获:** 通过 `try-except` 结构处理注册过程中可能出现的异常，并记录日志。

代码片段:

```

1 # POST 请求时
2 if request.method == 'POST':
3     username = request.POST.get('username', '').strip()
4     password = request.POST.get('password', '').strip()
5     contact = request.POST.get('contact', '').strip()
6
  
```

```
7 # 校验必填字段
8 if not username or not password or not contact:
9     messages.error(request, 'All fields are required.')
10    return render(request, 'management/register.html')
11
12 # 检查用户名是否已存在
13 if User.objects.filter(username=username).exists():
14     messages.error(request, 'Username already exists.
15     Please choose another one.')
16
17 # 创建用户和客户信息
18 try:
19     with transaction.atomic():
20         user = User.objects.create_user(username=username,
21                                         password=password)
22         Customer.objects.create(user=user, ID=username,
23                                  name=username, contact=contact)
24
25         messages.success(request, 'Registration successful!
26         Please log in.')
27         return redirect('login')
28
29 except Exception as e:
30     logger.error(f"Error during registration: {e}")
31     messages.error(request, 'An error occurred during
32                     registration. Please try again.')
33     return render(request, 'management/register.html')
```

3.5.2 用户登录功能 (Login)

登录功能允许已注册的用户通过用户名和密码登录。系统会验证用户的凭证，如果正确则跳转到主页 homepage，否则提示登录失败。

实现流程：

- 用户提交用户名和密码后，使用 `authenticate()` 验证其凭证。
- 如果验证成功，使用 `login()` 登录用户并重定向到主页。

- 如果验证失败，记录错误日志并返回错误信息。

主要算法：

- 身份验证**：通过 `authenticate()` 校验用户名和密码。
- 登录操作**：通过 `login()` 将用户登录状态保存在会话中。

代码片段：

```
1 def login_view(request):  
2     if request.method == 'POST':  
3         username = request.POST['username']  
4         password = request.POST['password']  
5         user = authenticate(request, username=username,  
6                             password=password)  
7         if user is not None:  
8             login(request, user)  
9             logger.info(f"User {username} logged in  
10                 successfully.")  
11             return redirect('homepage') # 登录成功后跳转到用户  
12                 仪表盘  
13         else:  
14             logger.error(f"Failed login attempt for username {  
15                         username}.")  
16             messages.error(request, 'Invalid username or  
17                             password.')  
18             return render(request, 'management/login.html')
```

3.5.3 系统中的用户信息记录

通过上述登录功能的实现，我们就可以在租车系统中的任何位置获取到当前登录的用户信息（如用户名、联系方式等），以便于后续的租车操作和信息管理。

我们可以通过 `view` 函数中的 `request` 对象获取当前登录的用户信息，如下所示：

```
1     user_id = request.user.username # 获取当前用户 ID
```

因为 Django 默认会将 `request.user` 传递到模板中，所以我们也可以通过直接在模板中访问它，而不需要在视图中显式传递，如下所示：

```
1     {% if user.is_authenticated %}  
2         <p style="font-size: 24px; font-family: 'Dancing Script',  
3             cursive; color: #7b1efc; font-weight: bold;">
```

3 详细设计

```
3     Welcome, <strong>{{ user.username }}</strong>
4   </p>
```

当然，如果没有登录，我们会判断出来，通过按钮跳转到 login 或者 register 页面。

```
1  {% else %}
2    <p>Please log in or register to start renting vehicles
3      .</p>
4    <a href="{% url 'login' %}" class="login-button">Login
5      </a>
6    <a href="{% url 'register' %}" class="register-button">
7      Register</a>
8  {% endif %}
```

用户应该在看到自己的账号名下有什么已经租聘的车辆。正如我们在 3.4.3 的还车算法中看到的，我们只需要在前端将当前用户名下的车辆渲染出来即可。

```
1  <!-- 我的租赁部分 -->
2  <div class="card-container">
3    <div class="card">
4      <h3 style="text-align: center;">我的租赁 </h3>
5      <div class="table-container">
6        <table>
7          <thead>
8            <tr>
9              <th>车牌号 </th>
10             <th>车型 </th>
11             <th>价格 </th>
12             <th>操作 </th>
13           </tr>
14         </thead>
15         <tbody>
16           {% for car in rented_cars %}
17             <tr>
18               <td>{{ car.license_plate
19                 }}</td>
20               <td>{{ car.model }}</td>
21               <td>¥ {{ car.price }}</td>
22               <td>
23                 <!-- 还车按钮 -->
```

```
23          <form action="{% url 'return_vehicle' %}"  
24              method="POST" style=  
25                  "display:inline;">  
26              {% csrf_token %}  
27              <input type="hidden"  
28                  " name="  
29                      vehicle_id"  
30                  value="{{ car.  
31                      license_plate }}"  
32                  " >  
33              <button type="submit" class="  
34                  btn-danger">还车  
35              </button>  
36          </form>  
37      </td>  
38  </tr>
```

```
{% empty %}  
<tr>  
    <td colspan="4">暂无租赁车  
辆</td>  
</tr>  
{% endfor %}  
</tbody>  
</table>  
</div>  
</div>
```

3.6 安全性与完备性

为了确保车辆租赁管理系统的数据库安全性与完备性，采取了以下设计和实现措施：

3.6.1 用户认证与权限管理

- **用户认证：**通过 Django 内置的用户认证系统（Authentication System），对系统用户进行登录验证，确保只有授权用户能够访问系统。

- **权限控制:** 基于角色分配权限，不同用户（如管理员和普通用户）拥有不同的数据库访问权限。例如：

- 管理员：拥有添加、修改、删除数据的权限。
- 普通用户：仅能查看车辆信息和提交租赁请求。

3.6.2 数据完整性约束

- **主键约束:** 每张表都有主键，确保每条记录具有唯一标识符。
- **外键约束:** 外键关系保证了数据的一致性，例如租赁记录表中的车辆 ID 和客户 ID 必须合法且存在。
- **非空约束:** 关键字段（如客户姓名、联系方式等）设置为非空，确保数据完整。
- **默认值约束:** 为布尔字段（如车辆是否租赁）设置默认值，减少人为错误。

3.6.3 数据保护与加密

- **敏感数据加密:** 对于用户的敏感信息（如密码），使用 Django 提供的密码哈希功能进行加密存储。

Django 的 User 模型已经内置了密码哈希功能，因此只需要使用 `create_user` 方法来处理用户密码。下面是我们实现的思路：

```
1 # 创建用户并加密密码
2 user = User.objects.create_user(username=username, password=password)
```

3.6.4 日志记录与审计

- 系统记录所有数据库操作的日志，包括数据添加、修改、删除等关键操作。
- 日志可用于追踪用户操作行为，及时发现和响应潜在的安全威胁。

```
1 LOGGING = {
2     'version': 1,
3     'disable_existing_loggers': False,
4     'formatters': {
5         'verbose': {
6             'format': '{levelname} {asctime} {module} {message}'
7         },
8     },
9 }
```

3 详细设计

```
7      'style': '{',
8    },
9      'simple': {
10        'format': '{levelname} {message}',
11        'style': '{',
12      },
13    },
14    'handlers': {
15      'console': {
16        'level': 'DEBUG',
17        'class': 'logging.StreamHandler',
18        'formatter': 'simple',
19      },
20      'file': {
21        'level': 'INFO',
22        'class': 'logging.FileHandler',
23        'filename': 'myapp.log',
24        'formatter': 'verbose',
25      },
26    },
27    'loggers': {
28      'django': {
29        'handlers': ['console'],
30        'level': 'DEBUG',
31        'propagate': True,
32      },
33      'myapp': {
34        'handlers': ['console', 'file'],
35        'level': 'DEBUG',
36        'propagate': True,
37      },
38    },
39  }
```

在 views.py 中，我们可以记录用户操作日志，如下所示（举例说明）：

```
1 if User.objects.filter(username=username).exists():
2     logger.warning(f"Username {username} already exists")
```

```
    .")
messages.error(request, 'Username already exists.
Please choose another one.')
return render(request, 'management/register.html')
```

然后我们操作后就可以在 myapp.log 文件中看到相关日志。

```
1   WARNING 2024-12-25 10:36:47,223 views Username demo already
exists.
2   INFO 2024-12-25 10:41:41,707 views User demo logged in
successfully.
3   INFO 2024-12-25 10:41:43,616 views User demo returned
vehicle with ID XPENG001.
4   INFO 2024-12-25 10:41:54,603 views User demo returned
vehicle with ID XPENG001.
```

3.6.5 最小权限原则

- 数据库账户的权限设置遵循最小权限原则，只赋予完成任务所需的最低权限，避免权限过高导致的安全隐患。

通过以上安全性设计，系统能够有效保护数据免受未授权访问、篡改或丢失，确保数据库的完整性、保密性和可用性。

4 调试与运行结果

4.1 问题与调试

4.1.1 request.user 为匿名用户

如果用户没有登录，`request.user` 会是一个 `AnonymousUser` 实例，而不是 `User` 实例。在这种情况下，`request.user` 的属性（如 `username`、`email`）可能会导致错误或不符合预期。

解决方法：

- 使用 `request.user.is_authenticated` 判断用户是否登录，以避免访问匿名用户的属性。
- 如果用户未登录，确保有对应的处理策略。

```
1   user_id = request.user.username # 获取当前用户 ID
```

4.1.2 Customer 模型与 User 模型的关系未正确配置

在代码中，如果通过 User 对象来查找 Customer 对象时，确保 User 和 Customer 模型之间的关系正确配置（例如，Customer 是通过 User 外键关联的）。

解决方法：

- 确保在 Customer 模型中有一个外键字段指向 User 模型，并且关系设置正确。

```
1 class Customer(models.Model):
2     user = models.OneToOneField(User, on_delete=models.CASCADE)
3     ID = models.CharField(max_length=10, primary_key=True) # 使用 'ID' 作为主键
4     name = models.CharField(max_length=50)
5     contact = models.CharField(max_length=15)
6
7     def __str__(self):
8         return self.name
```

- 使用 `get_object_or_404` 来获取关联的 `Customer.DoesNotExist` 异常。

```
1 from django.shortcuts import get_object_or_404
2
3 try:
4     # 获取当前登录的用户来查找关联的 Customer 实例
5     customer = get_object_or_404(Customer, user=request.user)
```

4.1.3 用户信息在模板中无法访问

如果在模板中无法访问 `request.user` 或 `user` 变量，可能是由于模板没有正确获取上下文中的用户信息。

解决方法：

- 确保模板中有 `if user.is_authenticated` 判断语句，并正确引用 `user` 变量。
- 如果自定义了中间件或修改了上下文，需要确保 `user` 被传递到模板中。

```
1 <div class="user-info">
2     {% if user.is_authenticated %}
3         <p style="font-size: 24px; font-family: 'Dancing
4             Script', cursive; color: #7b1efc; font-weight:
5                 bold;">
6             Welcome, <strong>{{ user.username }}</strong>
7         </p>
8     {% else %}
9         <p>Please log in or register to start renting
10            vehicles.</p>
11         <a href="{% url 'login' %}" class="login-button">
12             Login</a>
13         <a href="{% url 'register' %}" class="register-
14             button">Register</a>
15     {% endif %}
16 </div>
```

4.1.4 @login_required 装饰器没有正确应用

@login_required 装饰器要求用户登录才能访问视图，如果没有正确应用，可能会导致未登录用户能够访问需要登录才能访问的视图。

解决方法：

- 确保视图函数上正确应用了 @login_required 装饰器。

```
1 from django.contrib.auth.decorators import login_required
2 @login_required
3 def rent_car_view(request):
```

- 如果 @login_required 装饰器不适用，用户会被自动重定向到登录页面。可以通过设置 LOGIN_URL 来修改重定向的登录页面：

```
1 LOGIN_URL = '/management/login/' # 登录页面
```

4.2 事务的并发访问控制与恢复

4.2.1 问题分析

本系统是允许用户同时访问的，也就是说，在这个系统中用户的行为是并发的。而注意到，与车辆详细信息的关系表是可以被不同用户访问的，同时车辆的租赁状态也可以被不同用户的租车事务所改变，而对于其他的关系表，用户则只能访问与之有关的数据，不能访问其他数据。因此，在这个系统中只有当事务访问与车辆信息有关的关系表时，才有可能出现事务的隔离性被破坏的情况。

更进一步的，在这个系统中，会改变车辆租赁状态的事务为，访问车辆信息并租赁车辆信息。当有多用户同时访问同一个车辆信息时，如果有用户租用了该车，这时因为其他用户看到的车辆信息是租用车辆前的车辆状态信息，所以他们仍然可以租用该车。这导致了一辆车同时被多个用户租用，也就导致了数据的不一致，使得事务的一致性，隔离性和原子性被破坏了。

如图 7 所示，对同一辆车，前一个用户租用后，第二个用户因为网页页面没有刷新，导致得到车辆仍未被租的错误信息，从而两个用户租用了同一辆车，发生了的错误。

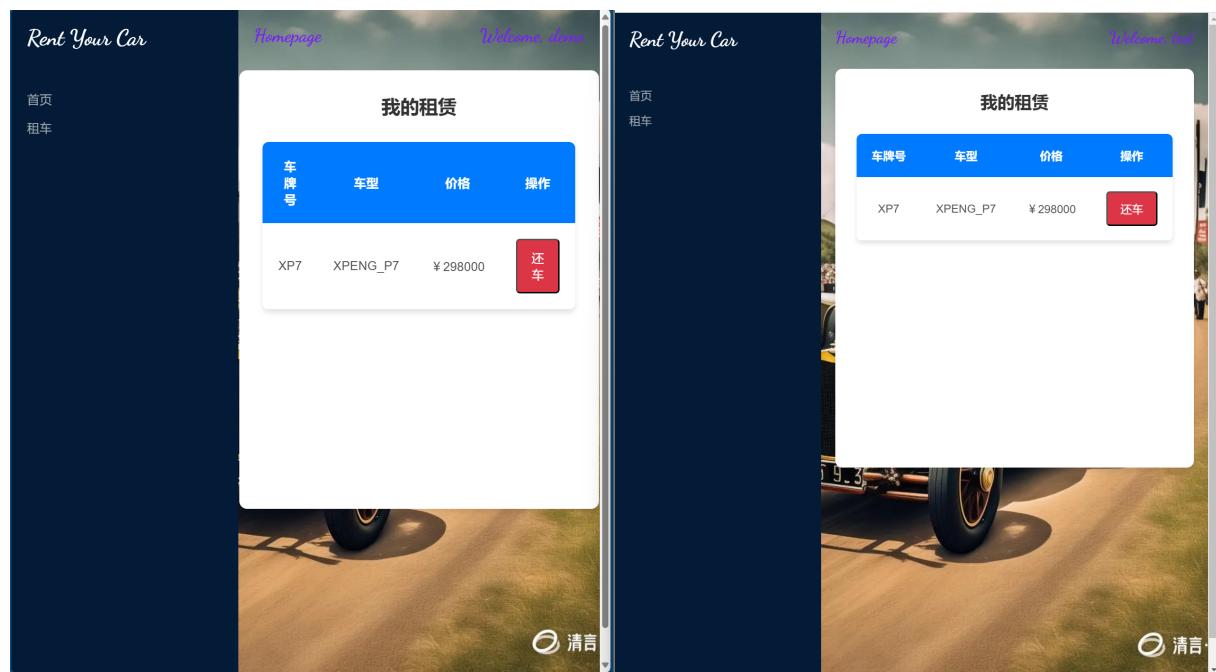


图 7: 并发访问冲突示例

4.2.2 并发访问控制

在我们的系统中，我们使用时间戳排序协议来进行并发访问控制，而不采用封锁机制。这是因为，用户并不是在访问车辆信息后马上租用车辆的，相反的，用户往往总是处于浏览车辆信息的阶段，并在长时间后有可能租借车辆。如果我们引入共享锁和排他锁进行并发访问控制，就会因为总是有用户在浏览网页而占有共享锁，而使得一个用户

4 调试与运行结果

在想租车时迟迟不能获得排他锁，而一直不能租车，导致**饥饿 (starvation)**发生。因此，我们选择使用时间戳排序协议。我们采用如下的时间戳排序协议。系统为每辆车 Q 维护以下两个时间戳：

- $R - timestamp(Q)$: 表示最后一次成功读取 Q 的事务时间戳。
- $W - timestamp(Q)$: 表示最后一次成功写入 Q 的事务时间戳。

当一个用户尝试进行一个访问租车事务时，这个事务会得到一个时间戳 $TS(T_i)$ ，并且这个事务在执行操作的过程中需要遵循以下规则

- 访问车辆信息 (读操作)
 - 如果 $TS(T_i) < W - timestamp(Q)$ ，拒绝操作，回滚事务。
 - 如果 $TS(T_i) \geq W - timestamp(Q)$ ，允许操作，更新 $R - timestamp(Q) = \max(R - timestamp(Q), TS(T_i))$ 。
- 租用车辆 (写操作)
 - 如果 $TS(T_i) < R - timestamp(Q)$ ，拒绝操作，回滚事务。
 - 如果 $TS(T_i) < W - timestamp(Q)$ ，拒绝操作，回滚事务。
 - 其他情况下，允许写操作，更新 $W - timestamp(Q)$ 。

我们在原有代码的基础上进行修改实现该协议下的车辆租赁管理系统，为了获得一个时间戳，我们构建如下函数生成以毫秒为单位的时间戳

```
1 import time
2 def generate_timestamp():
3     return int(time.time() * 1000)
```

然后，我们在关系表 Repository 中添加两个时间戳属性

```
1 r_timestamp = models.BigIntegerField(default=0)
2 w_timestamp = models.BigIntegerField(default=0)
```

在 JavaScript 中添加变量管理每个车辆的时间戳

```
1 let TS = new Map();
```

当事务尝试访问车辆信息时有

```
1 # 由于是第一次访问，所以先生成当前事务的时间戳
2 transaction_timestamp = generate_timestamp()
3 # 然后根据协议访问车辆信息
4 query = """
5     SELECT a."Car_ID", a."Is_leased", b."Price"
```

4 调试与运行结果

```
6     FROM management_repository a, management_vehicle b
7     JOIN management_info c ON b."Model" = c."Model_id"
8     WHERE a."Car_ID" = c."Car_ID_id" AND b."Model" = %s AND %s
9         >= a.w_timestamp
10    """
11
12    with connection.cursor() as cursor:
13        # 执行查询语句
14        cursor.execute(query, [car_name, transaction_timestamp])
15        rows = cursor.fetchall()
16
17    # 最后根据协议更新库中每个数据项的时间戳
18    update_query = """
19        UPDATE management_repository
20        SET r_timestamp = GREATEST(r_timestamp, %s)
21        WHERE (%s >= w_timestamp) AND "Car_ID" in (
22            SELECT a."Car_ID"
23            FROM management_repository a, management_vehicle b
24            JOIN management_info c ON b."Model" = c."Model_id"
25            WHERE a."Car_ID" = c."Car_ID_id" AND b."Model" = %s
26        )
27    """
28
29    with connection.cursor() as cursor:
30        # 执行更新语句
31        cursor.execute(update_query, [transaction_timestamp,
32                                       transaction_timestamp, car_name])
33
34    # 最后将产生的时间戳返回给该事务
35    return JsonResponse({"status": "success", "data": data, "TS": transaction_timestamp})
```

获得时间戳后，事务维护其时间戳

```
1 const timestamp = data.TS;
2 # 循环处理每个车辆的时间戳（这里省略了循环的代码）
3 TS.set(item.Car_ID, timestamp);
```

当用户想要租车时，事务将其时间戳传递到租车函数中

```
1 fetch('/management/lease-vehicle/', {
2     method: 'POST',
3     headers: {
4         //
```

```
5     },
6     body: JSON.stringify({ car_id: carId, ts: TS.get(carId) })
7 })
```

在租车函数中，函数根据事务提供的时间戳依照协议进行写操作

```
1 # 获得当前事务传递的时间戳
2 transaction_timestamp = data.get("ts")
3 data_object = Repository.objects.get(Car_ID=car_id)
4 # 时间戳协议检查
5 if transaction_timestamp < data_object.r_timestamp:
6     return JsonResponse({"message": "租车失败，车辆租赁信息已更
7         改"}, status=409)
8 if transaction_timestamp < data_object.w_timestamp:
9     return JsonResponse({"message": "租车失败，车辆租赁信息已更
10        改"}, status=409)
11 data_object.w_timestamp = transaction_timestamp
12 data_object.save()
13 # 当时间戳符合条件时进行写操作，否则函数返回状态数409提示系统发
14        生了冲突，租赁操作不可以继续，并中断该操作
15 #
```

4.2.3 恢复系统

该本系统中，我们只考虑由于时间戳排序协议带来的事务故障。在这种情况下，当故障发生时，我们可以知道这个事务的日志此时还没有写入数据，因此，我们不需要进行回滚，仅仅是中止该事务即可。当回滚事务后，我们选择 redo 这个事务。这时，这个事务获得一个新的时间戳并重新开始可以读取到新的数据。注意到，这个事务重新开始意味着重新读入数据，于是我们可以通过刷新页面的方式来获得新的时间戳并按照此时间戳重新开始事务。代码如下

```
1 alert(data.message); // 显示错误消息
2 fetchVehicleDetails(name); // 刷新车辆列表
```

4.3 最终结果展示

经过精细的设计，我们实现了具备有注册登录功能的全面的精美的租车管理系统。在这个阶段中，我们将以用户视角展示我们的车辆租赁管理系统。

4.3.1 注册与登录

图8展示了该系统的注册页面，用户在注册页面中填入用户名，密码和邮箱，在不发生用户冲突的情况下即可注册成功。注册成功后页面会跳转到登录界面便于用户登录。由于对用户信息的保护这个过程是加密的，非明文的，所以这个过程对客户来说时安全的，无需担心的。



图8: 用户注册界面

图9展示了该系统的登录界面，用户可以主动访问该界面，或者在注册成功后跳转到该界面进行登录。用户只需填写刚才注册的用户名和密码，在信息校验成功后即登录成功，将会进入专属该用户的首页中。在登录页面，我们还加入了注册界面跳转连接，当用户发现没有账号时可以在登录页面便利地跳转到注册页面进行注册。

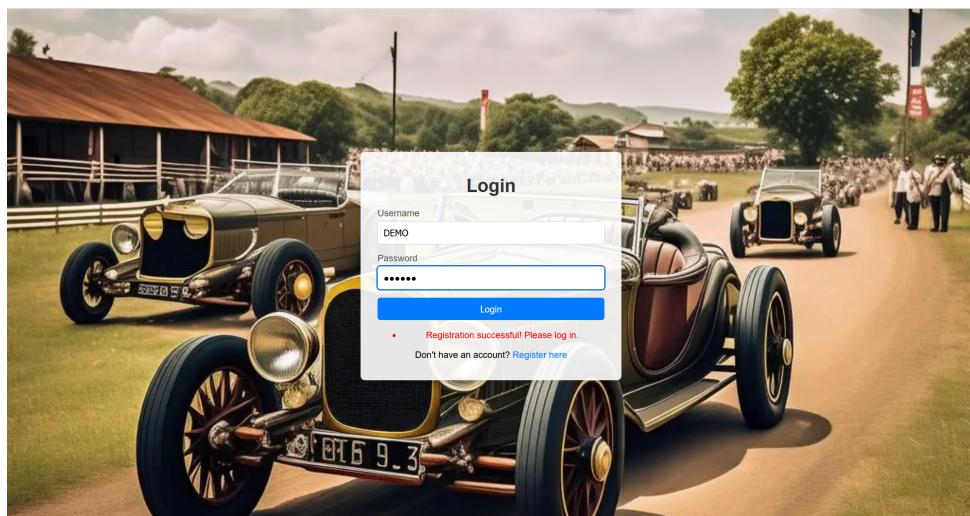


图9: 用户登录界面

4.3.2 用户首页

登录成功进入用户首页后，用户可以查看自己当前的租车状态。在该侧面的侧边栏由两个按键，一个是首页，用户可以通过这个按键访问首页即图片现实的内容。另一个是租车，用户可以通过租车键查看车辆品牌，并进一步访问详细车辆信息。

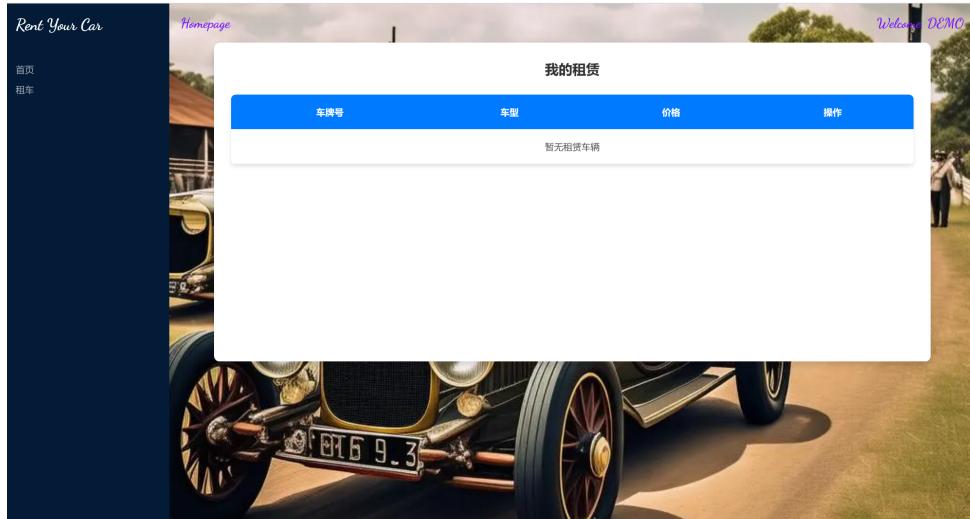


图 10: 用户首页

当用户租车后，该用户的租车信息会显示在首页中，如图 11所示 然后，用户通过点

我的租赁			
车牌号	车型	价格	操作
s6876	DYB	¥ 10	<button>还车</button>
y3425	Conda	¥ 400	<button>还车</button>

图 11: 用户租车信息列表

击还车键即可一键还车。换车后该车辆信息会从主页中移除，如图 12所示。

我的租赁			
车牌号	车型	价格	操作
暂无租赁车辆			

图 12: 用户还车后列表信息

4.3.3 租车查询

用户可以通过点击侧栏租车按键，查看车辆品牌信息，如图 13所示。进一步的，用

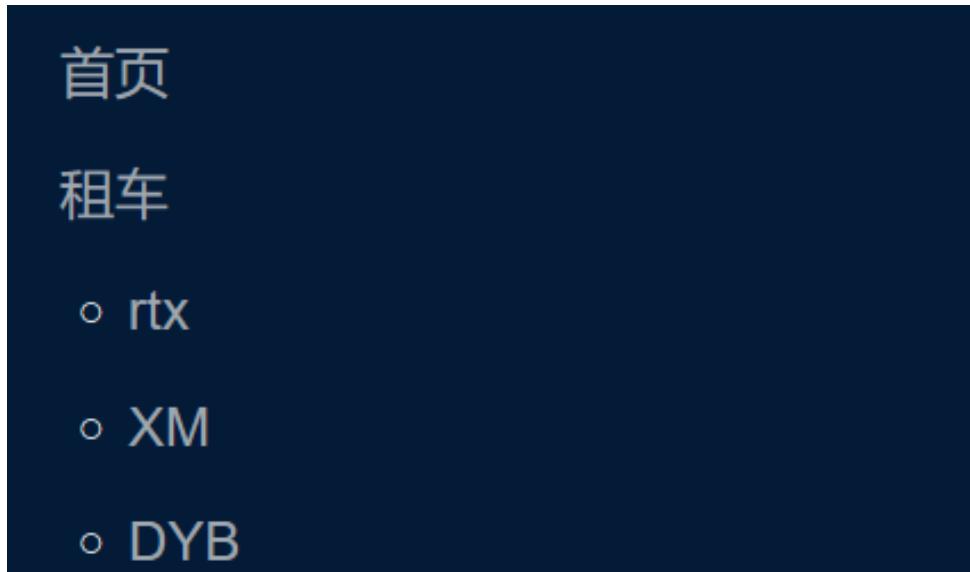


图 13: 侧栏车辆品牌信息

户可以点击自己感兴趣的车辆品牌，此时交互页面会跳转到含有该品牌车辆详细信息的界面供用户查看，如图 14 所示。对于想租的车辆，用户可以点击相应的租赁按钮进行租

车辆信息			
车牌号	租赁状态	价格	操作
s8765	可租赁	¥ 10	<button>租赁</button>
s6876	可租赁	¥ 10	<button>租赁</button>
g8975	可租赁	¥ 10	<button>租赁</button>
v9876	可租赁	¥ 10	<button>租赁</button>

图 14: 详细车辆信息

赁。当租赁成功后，页面会更新车辆状态，原按键变灰暂时不可用，如图 15 所示。

4.3.4 并发访问控制

为了模拟并发访问冲突的情况，我们进行如下实验。分别在两个浏览器中用不同用户身份进入租车页面并查看同一个车辆信息，此时两个用户都可以看到车辆是可以租赁的。这时一个用户租用一辆车，此时另外一个用户由于没有刷新页面所以仍然可以租用那辆被租的车。这时我们点击租赁那辆车时，由于时间戳排序协议的存在，那辆车的写时间戳已经大于事物的时间戳，所以租车的行为不能发生。如图 16 所示，用户点击后系统发出信息提示车辆的信息已经被更改了，租车失败。当点击确认后页面会自动刷新，此时用户看到车已经被其他用户租用了。

车辆信息			
车牌号	租赁状态	价格	操作
s8765	已租赁	¥ 10	已租赁
s6876	已租赁	¥ 10	已租赁
g8975	可租赁	¥ 10	租赁
v9876	可租赁	¥ 10	租赁

图 15: 详细车辆信息

车辆信息			
车牌号	租赁状态	价格	操作
g8975	可租赁	¥ 10	租赁
s6876	可租赁	¥ 10	租赁
s6876	127.0.0.1:8000 租车失败，车辆租赁信息已更改		租赁
v9876	可租赁	¥ 10	租赁

图 16: 并发访问控制

5 总结

本实验设计旨在通过分析需求、设计系统架构和数据库结构，开发一个功能完整的车辆租赁管理系统。设计过程中，首先明确了系统所需实现的核心功能，包括车辆信息管理、租赁管理、客户管理及系统安全等，并结合实际需求制定了详细的功能模块划分和技术架构。系统架构上，采用了前后端交互的设计模式，前端采用 HTML 和 CSS 等技术进行界面设计，后端则基于 Django 框架处理数据逻辑及业务流程，数据库部分使用了 PostgreSQL 以保证系统的数据安全性和一致性。数据库设计遵循关系型数据库的设计规范，通过 E-R 图及关系模式分析，合理设计了各数据表之间的关联关系，并且通过标准化原则（如第三范式）保证了数据存储的高效性与无冗余性。

系统的开发过程中，后端逻辑层通过 Django 框架实现了用户管理、车辆信息管理和租赁流程等功能模块，同时针对系统安全性进行了设计，主要包括用户身份验证、数据加密、权限控制等措施，保障了系统的基本安全要求。此外，还在系统中进行了并发访问控制，确保了多用户同时使用该系统时数据不会出现不一致的情况。在前端设计上，注重界面的简洁性和用户操作的便捷性，通过直观的布局与交互设计，确保了良好的用户体验。系统开发完成后，通过多轮功能测试和压力测试，验证了各模块的稳定性与正确性，确保了系统的高效运行和良好的用户体验。

尽管系统已初步实现了预定功能目标，但在实际应用中，仍存在一定的优化空间。在性能方面，数据库查询的效率有待提升，尤其是在面对大量数据和高并发访问时，可

以通过增加索引、优化 SQL 查询、引入缓存机制等方式来优化响应时间和系统吞吐量。在系统架构方面，未来可以考虑引入微服务架构，以便更好地应对系统扩展和高可用性要求。此外，安全性方面，尽管基本采用了加密和权限控制，但还可以进一步完善，如增加基于角色的访问控制（RBAC）、引入多因素身份验证（MFA）等机制来增强系统的安全性。功能层面，未来可通过引入数据分析模块，为系统提供数据驱动的决策支持，例如租赁趋势分析、客户行为预测等，进一步提升系统的智能化水平。在并发访问控制层面，虽然在当前系统中可能存在更简单的方法解决并发访问控制问题，但是我们的做法保证了更强大的泛化性，提供可以用于支持系统后续功能的并发访问控制。

综上所述，本实验设计实现了一个完整的车辆租赁管理系统，涵盖了从需求分析到系统测试的全过程。尽管系统具备了基本的功能和稳定的运行状态，但在性能优化、安全增强和功能扩展等方面，仍有较大的提升空间。未来可以通过持续优化架构设计、引入新技术和完善系统功能，进一步提升系统的整体性能和用户体验。

参考文献

- [1] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. Database system concepts. 2011.