《数据库应用系统》详细设计

项目名称：数据库应用系统

子系统名称：线上图书商城

当前版本：V.12.13.8

最后修改时间：2023 年 12 月 13 日

修订记录

修订版本号	修订日期	修订描述	作者
V.12.10.0	2023/12/10	主体内容撰写	刘子晗
V.12.12.0	2023/12/12	构件撰写 1	吴程翔
V.12.12.1	2023/12/12	构件撰写 2	杨舜杰
V.12.12.2	2023/12/12	构件撰写 3	于珈尉
V.12.13.0	2023/12/13	后端部分撰写	刘征昊
V.12.13.1	2023/12/13	主体内容合并	刘子晗
V.12.13.2	2023/12/13	全文排版与修订	刘征昊

目 录

引言.....8

1.1 系统标识.....8

1.2 编写目的.....8

1.3 本文的读者群.....9

1.4 专门术语及缩略词定义.....9

1.5 参考文献.....9

2. 概要设计.....10

2.1 设计注意事项、原则、目标及项目规范.....10

2.1.1 设计方法.....10

2.1.2 开发框架选择依据.....10

2.1.3 协议/接口要求.....11

2.1.4 编程语言与代码规范.....11

2.1.5 模块之间的数据交互.....12

2.1.6 系统设计目标.....12

2.2 总体结构设计概述.....12

2.3 构件设计.....13

2.3.1 构件 1：用户登录.....13

2.3.2 构件 2：用户注册.....15

2.3.3 构件 3：高级搜索.....17

2.3.4 构件 4：书籍浏览.....19

2.3.5 构件 5：书籍详情弹出界面.....22

2.3.6 构件 6：获取收藏夹.....25

2.3.7 构件 7：删除收藏夹中的内容.....27

2.3.8 构件 8：个人信息中心.....28

2.3.9 构件 9：个人购物历史.....29

2.3.10 构件 10：获取购物车信息.....32

2.3.11 构件 11：添加购物车.....34

2.3.12 构件 12：支付.....37

2.4 构件合作模型.....39
2.5 图形用户接口设计.....39
2.5.1 导航区域：.....40
2.5.2 用户登录与注册界面.....40
2.5.3 首页.....41
2.5.4 高级搜索界面.....41
2.5.5 书籍浏览界面.....42
2.5.6 用户收藏夹界面.....43
2.5.7 用户购物车界面.....43
2.5.8 用户个人中心界面.....44
2.6 数据库设计.....44
2.6.1 对象永久存储策略.....44
2.6.2 数据库表设计.....44
2.6.3 存储过程.....51
2.6.4 触发器设计.....51
2.7 系统使用方法设计.....51
2.8 公共函数库.....51
2.8.1 Python.....51
2.8.2 Vue.....52
2.9 非功能性需求.....52
2.9.1 安全性.....52
2.9.2 易用性.....52
2.9.3 资源使用率.....52
2.9.4 兼容性需求.....52
2.10 辅助工具.....52
3. 系统详细设计.....53
3.1 开发环境概述.....53
3.1.1 版本控制工具.....53
3.1.2 编译环境.....53

3.1.3 源程序目录.....53
3.1.4 文档存储目录.....55
3.2 构件详细设计.....55
3.2.1 构件 1：用户登录.....55
3.2.2 构件 2：用户注册.....57
3.2.3 构件 3：高级搜索.....60
3.2.4 构件 4：书籍浏览.....64
3.2.5 构件 5：书籍详情弹出窗口.....72
3.2.6 构件 6：获取收藏夹.....74
3.2.7 构件 7：删除收藏夹中的内容.....75
3.2.8 构件 8-9：个人中心.....76
3.2.9 构件 10：获取购物车信息.....78
3.2.10 构件 11：添加购物车.....79
3.2.11 构件 12：支付.....80

插图目录

图 1 系统整体架构图.....13
图 2 书籍浏览页面构件合作.....39
图 3 登录界面.....40
图 4 注册界面.....40
图 5 首页界面.....41
图 6 弹出书籍详情窗口.....41
图 7 高级搜索界面.....42
图 8 书籍浏览界面.....42
图 9 用户收藏夹界面.....43
图 10 用户购物车界面.....43
图 11 用户个人信息界面.....44

表 格 目 录

表 1 用户登录请求参数.....15
表 2 用户注册请求参数.....16
表 3 高级搜索数据请求参数.....18
表 4 书籍浏览请求参数.....21
表 5 书籍加入购物车请求参数.....25
表 6 数据加入收藏夹请求参数.....25
表 7 获取收藏夹数据请求参数.....26
表 8 删除收藏夹请求参数.....28
表 9 获取个人信息请求参数.....29
表 10 获取购物历史请求参数.....31
表 11 获取购物车数据参数.....33
表 12 查询购物车指定数据.....35
表 13 创建购物车数据请求参数.....36
表 14 更新购物车数据请求参数.....36
表 15 修改书籍销量请求参数.....38
表 16 删除购物车数据请求参数.....39
表 17 books 设计表.....45
表 18 authors 设计表.....46
表 19 publishers 设计表.....47
表 20 category 表设计.....47
表 21 users 设计表.....48
表 22 shoppinghistory 设计表.....49
表 23 shoppingcarts 设计表.....50
表 24 collection 设计表.....50

引言

1.1 系统标识

项目名称：数据库应用系统——线上图书商城

（Database Application System——Online BookStore）

项目简称：图书商城（DBAS——OBS）

项目组成员：

姓名	职务	联系方法
刘征昊	开发工程师（后端）	21371445
刘子晗	开发工程师（数据库）	21371440
于珈尉	开发工程师（前端）	21371048
杨舜杰	开发工程师（前端）	21373140
吴程翔	开发工程师（前端）	21373305

1.2 编写目的

人们现在的生活方式因为网络的普及发生了巨大变化，由于电子商务在人们的视野中出现,人们对电子商务额外的关注。人们可以足不出户买到世界各地的图书,网上商城可以销售各式各样的图书,其中包括虚拟商品、电子商品、日常生活用品等等。我们的目标不只是在网上展示我们的图书，更重要的是，让更多的客户了解图书创造更多的商机。所以我们目前的挑战是前台界面的设计，要把顾客的眼球吸引住，选则比较人性化的界面设计，要更直观的表现，从而上顾客买到喜欢的图书。

本系统的主要意义在于，全力以赴为用户提供一个操作方便，界面简洁，信息直观的网上交易系统。使用该系统的用户，可以先浏览到图书信息、系统公告，并可以注册成为本网站的用户，可以利用购物车选择自己想买的图书，然后向商家提交订单，从而完成网上的交易流程。

1.3 本文的读者群

从事计算机科学领域的相关人员（如老师、学生、工程师等）。

知识储备：MySQL 操作（包括但不限于增删改查）；Django 框架及使用方法；Node.js 及 Vite 搭建 Web 服务器；Vue.js 框架；前端实现；前后端分离架构的配置与跨域处理。

1.4 专门术语及缩略词定义

无

1.5 参考文献

[1] https://www.djangoproject.com/

[2] https://www.django-rest-framework.org/

[3] https://cn.vuejs.org/

[4] https://element-plus.org/zh-CN/

[5] https://www.mysqlzh.com/

2. 概要设计

2.1 设计注意事项、原则、目标及项目规范

2.1.1 设计方法

图书商城系统的设计遵循“低耦合，高内聚”的设计原则，将系统功能分解为前端的信息展示与用户操作、后端的操作请求处理及数据库交互以及数据库管理系统的数据增删改查实现三个较为独立的部分。

具体方法上，故本项目采用前后端分离的方法，将三个部分分别交由 Vue 框架、Django 框架、MySQL 数据库来完成。

2.1.2 开发框架选择依据

Vue 作为一款轻量级前端框架，工程搭建简单，只需几行命令符，且主体语言为 JS/TS，开发者可以灵活地将其他框架的项目迁移到 Vue，具有很高的集成能力；Vue 框架将组成页面的 HTML、CSS 和 TS/JS 合并到一个组件中，可以被其它组件或页面重复利用，这种组件化的特性可以很好地将一个庞大复杂前端工程拆分为一个一个的组件，重复利用的性质也大大提高了开发效率；MVVM 为 Vue 框架提供了数据的双向绑定，减少了 DOM 操作，更高效地实现了视图和数据的交互，同时 MVVM 使界面、交互和数据层分离，便于设计人员负责设计界面、后端开发人员提供接口、前端开发人员专注于业务交互逻辑的实现；Vue 有许多 npm 扩展包和开发工具，可以在一个文件夹下统一管理所有外部插件的全局使用。基于上述理由，本项目采用 Vue 作为前端框架

Django 的主体语言为 Python，Python 语言使用简单，相较于其它编程语言（如 C++、Java），可以使用更少的代码来完成更多的功能，同时有很多功能丰富且安装便捷的第三方库；Django 支持 ORM，可以让开发人员更加轻松地操作数据库，而不用写繁琐的 SQL 语句；Django 提供了完整的 MVC 架构，让开发人员可以更加清晰地组织和管理代码；Django 还提供了大量的内置功能（比如表单处理、用户认证等等）。基于上述理由，本项目采用 Django 作为后端框架。

MySQL 数据库体积小，且优化了 SQL 查询算法，本项目数据规模并不算大（最大的数据表约为 300 条数据），采用 MySQL 数据库可以满足更高的性能需

求；MySQL 为多种编程语言提供了 API，比如本项目所使用的 Python；MySQL 提供了多语言支持，常见的编码如中文的 GB2312、国际的 UTF8 等都可以用作数据表名和数据列名。基于上述利用，本项目采用 MySQL 作为数据库存储。

2.1.3 协议/接口要求

REST 全称是 Representational State Transfer，中文意思是表征性状态转移。它首次出现在 2000 年 Roy Fielding 的博士论文中，Roy Fielding 是 HTTP 规范的主要编写者之一。 他在论文中提到："我这篇文章的写作目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。REST 指的是一组架构约束条件和原则。" 如果一个架构符合 REST 的约束条件和原则，我们就称它为 RESTful 架构。

REST 本身并没有创造新的技术、组件或服务，而隐藏在 RESTful 背后的理念就是使用 Web 的现有特征和能力， 更好地使用现有 Web 标准中的一些准则和约束。虽然 REST 本身受 Web 技术的影响很深， 但是理论上 REST 架构风格并不是绑定在 HTTP 上，只不过目前 HTTP 是唯一与 REST 相关的实例。 所以描述的 REST 也是通过 HTTP 实现的 REST。

本项目遵循 RESTful API 接口规范。

2.1.4 编程语言与代码规范

编程语言上，前端采用 Vue+Typescript，后端采用 Python。具体而言，前端采用 Vue 框架，后端采用 Django 框架。

代码规范上，总体而言：变量命名中前后端统一采用驼峰命名法，数据库内表的属性名统一采用下划线命名法；缩进中，如果地位相等则不需要缩进，如果属于某一个代码的内部代码则需要缩进；代码中添加必要的注释。

对于不同的编程语言，具体如下：

Python 语言规则：只对软件包/模块使用 import 语句，而不对独立的类型、类或函数使用；对每一个模块使用完整路径来导入；避免可变的全局声明；仅对简单情况使用 lambda 表达式；对支持它们的类型使用默认的迭代器，如列表、字典等；如果可能的话，使用“隐式”False；当有明显的优势时，明智地使用装饰器；避免和限制地使用 staticmethod classmethod；根据 PEP-484 使用类型提示

注释 Python 代码，并在构建时对代码进行类型检查；

Python 风格规则：不要用分号终止一行；最大行长度为 80 个字符；用 4 个空格缩进代码块；更多地使用 f 字符串，不要用格式化%format；

Typescript 语法规范：一般情况下，标识符不使用$；标识符禁止使用_作为前缀或后缀；命名应当具有描述性且易于读者理解；省略对于 Typescript 多余的注释；不要使用@override；调用构造函数时必须使用括号；不要使用#private 语法声明私有成员；

Typescript 语言特性： 必须使用 const 或 let 声明变量，尽可能使用 const；实例化异常对象时，必须使用 new Error()；对对象迭代时，必须使用 if 语句对对象的属性进行过滤；

2.1.5 模块之间的数据交互

对于前端与后端的数据交互，本系统统一使用 JSON 格式来编码数据，并将数据封装在 HTTP 协议的 GET/POST 请求中，在前后端之间进行传输。

对于后端与数据库的数据交互，则通过 Django 原生 ORM 和 pymysql 包所提供的 SQL 编程接口实现。

2.1.6 系统设计目标

遵循简洁、易用的设计目标，为用户提供跟为友善的环境。

该系统主要实现以下目标：构造构造数据库、实现数据查询功能、实现数据统计功能、数据添加功能、数据删除功能。以上功能将通过 B/S 架构来实现。

此外，本系统还应具有身份验证、阻止非法行为等功能，实现较高的安全性。

2.2 总体结构设计概述

由于功能的体现都在前端上，故我们根据前端的页面进行功能的划分；由于除登录、注册外的页面，每个页面又集成了多个功能，故又将每个页面进行功能的细分，抽象出该系统的所有构件，形成了如图 1 所示的系统整体架构图。

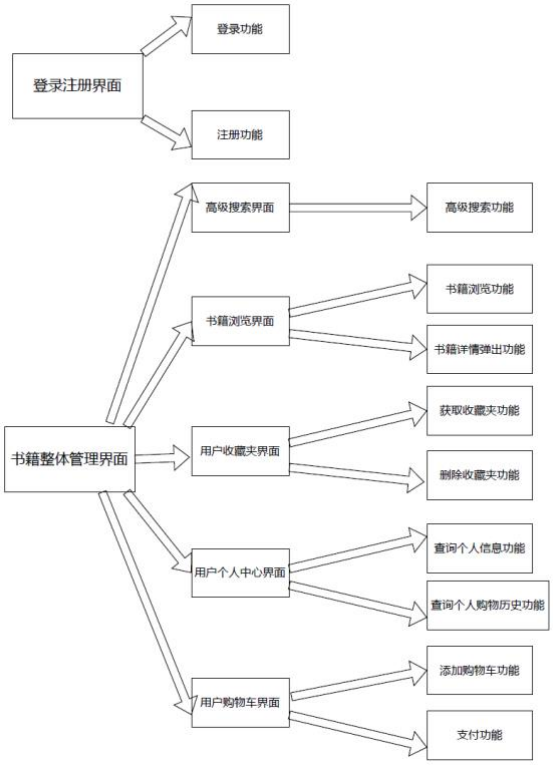

图 1 系统整体架构图

2.3 构件设计

2.3.1 构件 1：用户登录

2.3.1.1 功能描述

用户输入自己的用户名和密码，登录系统

2.3.1.2 对象模型

前端模型：

```
1. interface LoginInfo {
2.     username: string;
3.     password: string;
4. }
5. const param = reactive<LoginInfo>({
6.     username: ‘’,
7.     password: ‘’,
8. });
```

后端模型

```
1. class Users(Model):
2.     uid = IntegerField(primary_key=True, help_text="用户序号")
```

```
    uname = CharField(max_length=100, null=False, help_text="用户昵称")
    pwdhash = CharField(max_length=255, null=False, blank=False, help_text="用户密码的哈希值")
    email = CharField(max_length=100, null=True, blank=True, help_text="用户邮箱")
    tel = CharField(max_length=11, null=True, blank=True, help_text="用户电话")

    class Meta:
        db_table = 'users'

class UsersSerializer(ModelSerializer):
    class Meta:
        model = Users
        fields = '__all__'
```

后端主要函数：

```
def get_custom_filter(model):
    @action(methods=['get'], detail=False, schema=ManualSchema(
        description="自定义查询",
        fields=generate_fields_from_model(model)
    ))
    def custom_filter(self, request):
        # 获取前端提供的参数集合
        filter_params = dict(request.query_params)
        filter_params = {key: value[0] for key, value in filter_params.items()}
        # 在这里根据参数集合进行自定义筛选逻辑
        queryset = model.objects.all().filter(**filter_params)
        # 序列化结果并返回
        serializer = self.get_serializer(queryset, many=True)
        return Response(serializer.data)

    return custom_filter
```

设计简述：后端构件完整的用户模型；由于该构件的主要功能为登录，故前端返回的参数为用户名 username 和密码 password，后端主要函数 get_custom_filter 对输入内容进行过滤和判别，判断用户名与密码是否对应的上，判断成功后返回该用户的全部信息

2.3.1.3 接口规范

请求的 URL：/bookStore/api/users/custom_filter

请求方式：GET

请求参数：如下表 1 用户登录请求参数

参数名称	参数类型	参数描述	是否必须
uname	string	用户昵称	Y
passwd	string	用户密码	Y

表 1 用户登录请求参数

返回参数：

```
[
    {
        “uid”: int,
        “uname”: string,
        “pwdhash”: string,
        “email”: string,
        “tel”: string
    }
]
```

2.3.2 构件 2：用户注册

2.3.2.1 功能描述

对于一个未在该系统中注册的用户，可以在该系统中注册一个账号，输入必要的用户名和密码，可以选择输入邮箱、电话等信息。

2.3.2.2 对象模型

前端对象：

```
interface RuleForm {
  username: string
  userpwd: string
  address: string
  email: string
  telenum: string
}
const ruleForm = reactive<RuleForm>({
  username: '',
  userpwd: '',
  address: '',
  email: '',
  telenum: '',
})
```

后端模型：Users

后端主要视图：

```
1.  class UsersViewSet(ModelViewSet):
2.    queryset = Users.objects.all()
3.    serializer_class = UsersSerializer
4.    last = get_last(Users)
5.    get_id = get_id_by_name(Users, 'uid', 'uname')
6.    custom_filter = get_custom_filter(Users)
```

设计简述：

后端构建完整的用户模型；由于该构件的主要功能为注册，需要用户的较为完整的信息，其中用户名和密码是必须项，邮箱、电话为可选项。后端 User 视图集默认 create 方法利用前端发送的请求参数创建一条新的用户数据，在数据库中进行存储。

2.3.2.3 请求参数

请求的 URL：/bookStore/api/users/

请求方式：POST

请求参数：如下表 2 用户注册请求参数

参数名称	参数类型	参数描述	是否必须
uname	string	用户昵称	Y
passwd	string	用户密码	Y
email	string	用户邮箱	N
tel	string	用户电话	N

表 2 用户注册请求参数

返回参数：

```
1. {
2.    “uid”: int,
3.    “uname”: string,
4.    “pwdhash”: string,
5.    “email”: string/null,
6.    “tel”: string/null
7. }
```

2.3.3 构件 3：高级搜索

2.3.3.1 功能描述

获取全部的书籍信息（设置分页：20 条/页）

2.3.3.2 对象模型

前端模型：

```
interface RuleForm {
  name: string
  bookName: string
  Publisher: string
  StartTime: string
  EndTime: string
  type: string
}
const ruleForm = reactive<RuleForm>({
  name: '',
  bookName: '',
  Publisher: '',
  StartTime: '',
  EndTime: '',
  type: '',
})
```

后端模型：

```
class Books(Model):
    book_id = IntegerField(primary_key=True, help_text="书的序号")
    bname = CharField(max_length=255, null=False, help_text='书名')
    author = ForeignKey(Authors, on_delete=CASCADE, db_column='author_id')
    publisher = ForeignKey(Publishers, on_delete=CASCADE, db_column='pub_id')
    category = ForeignKey(Category, on_delete=CASCADE, db_column='category_id')
    price = DecimalField(max_digits=10, decimal_places=2, null=False, help_text="价格
")
    pub_year = IntegerField(null=False, help_text='出版年份')
    url = CharField(max_length=255, null=False, help_text="书籍图片链接")
    isbn = CharField(max_length=10, null=False, help_text="书籍 isbn 号")
    sales = IntegerField(null=True, default=0, help_text="书籍销售量")
    rate = DecimalField(max_digits=2, decimal_places=1, null=False, help_text="书籍评
分")

    @property
    def name(self):
```

```
16.         return self.bname
17.
18.     class Meta:
19.         db_table = 'books'
20.
21. class BooksSerializer(ModelSerializer):
22.     author = AuthorsSerializer(read_only=True)
23.     author_id = PrimaryKeyRelatedField(queryset=Authors.objects.all(),
write_only=True, help_text="作者序号")
24.     publisher = PublishersSerializer(read_only=True)
25.     pub_id = PrimaryKeyRelatedField(queryset=Publishers.objects.all(),
write_only=True, help_text="出版社序号")
26.     category = CategorySerializer(read_only=True)
27.     category_id = PrimaryKeyRelatedField(queryset=Category.objects.all(),
write_only=True, help_text="种类序号")
28.
29.     class Meta:
30.         model = Books
31.         fields = '__all__'
```

后端主要视图：booksViewSet

设计简述：

前端构建完整的书籍模型；由于该构件的主要功能为查询所有的书籍信息，而所有的书籍信息存在数据库中，故前端只需要构建一个书籍模型来接受后端返回的参数即可；而由于所有书籍信息是分页返回，故前端可以传递一个页数作为参数。后端利用 Books 视图集的 list 方法将指定页数的书籍信息返回。

2.3.3.3 接口规范

请求的 URL：/bookStore/api/books/

请求方式：GET

请求参数：如下表 3 高级搜索数据请求参数

参数名称	参数类型	参数描述	是否必须
page	int	搜寻的页数	Y

表 3 高级搜索数据请求参数

返回参数：

```
1. {
2.     “count”: int,
3.     “next”: string,
4.     “previous”: string/null,
```

```
    "result": [
        {
            "book_id": int,
            "authors": {
                "author_id": int,
                "aname": string
            },
            "publishers": {
                "publisher_id": int,
                "pname": string
            },
            "category": {
                "category_id": int,
                "category_name": string
            },
            "bname": string,
            "price": double,
            "pub_year": int,
            "url": string,
            "isbn": string,
            "sales": int,
            "rate": double
        },
    ]
}
```

2.3.4 构件 4：书籍浏览

2.3.4.1 功能描述

提供卡片形式的书籍浏览和筛选功能，对书籍进行筛选（按分类、价格、出版日期），对书籍展示进行排序（随机推荐、按价格排序、按出版日期排序、按销量排序），同时提供统计结果 csv 格式输出功能。用户点击卡片，会弹出窗口展示书籍详细信息，见构件 12。

2.3.4.2 对象模型

BooksApiRaw 是对书籍后端响应体 ApiRaw 的建模，接受时会被转换为前端书籍建模 Book，即将嵌套数据类型扁平化。BookService 是对界面中图书浏览、筛选、排序、统计等逻辑的集成服务，其调用两个组合类：书籍筛选器、书籍排序器。

```ts
export interface Book {
  id: number;
  title: string;
  author: string;
  publisher: string;
  category: string;
  year: number;
  isbn: string;
  price: number;
  sales: number;
  url: string;
  rate: number;
}

export interface BooksApiRaw {
  count: number;
  next: string | null;
  preivous: string | null;
  results: any[];
}

export class BookService {
  total: number = 0; // database 中书总数
  private books: Book[] = [];
  private currentPage: number = 1;
  async AddBooks(pageReq: number = 5) {
  }
  getBooks(): Book[] {
    return this.books;
  }
  bFilter(): BookFilter {
    return new BookFilter(this.books);
  }
  bSort(): BookSorter {
    return new BookSorter(this.books);
  }
  static output2CSV(filename: string, books: Book[]) {
}

class BookFilter {
  // 书籍筛选器，支持链式调用
  private books: Book[];
  constructor(books: Book[]) {
    this.books = books;
```

```
51.   }
52.   byYear(startYear: number, endYear: number): BookFilter {
53.   }
54.   byTitle(title: string): BookFilter {
55.   }
56.   byCategory(categories: string[] | null): BookFilter {
66.   }
68.   byPrice(minPrice: number, maxPrice: number): BookFilter {
69.   }
70.   getBooks(): Book[] {
71.   }
72. }

75. class BookSorter {
76.   // 书籍排序器，支持链式调用
77.   private books: Book[];
78.
79.   constructor(books: Book[]) {
80.     this.books = books;
81.   }
82.   byPrice(ascending: boolean = true) {
83.   }
84.   bySales(ascending: boolean = true) {
85.   }
86.   byYear(ascending: boolean = true) {
87.   }
88.   byRandom() {
89.   }
90. }
```

2.3.4.3 接口规范

请求的 URL：/bookStore/api/books/

请求方式：GET

请求参数：如下表 4 书籍浏览请求参数

参数名称	参数类型	参数描述	是否必须
page	int	搜寻的页数	Y

表 4 书籍浏览请求参数

返回参数：

```
1. {
2.     “count”: int,
3.     “next”: string,
```

```
    "previous": string/null,
    "result": [
        {
            "book_id": int,
            "authors": {
                "author_id": int,
                "aname": string
            },
            "publishers": {
                "publisher_id": int,
                "pname": string
            },
            "category": {
                "category_id": int,
                "category_name": string
            },
            "bname": string,
            "price": double,
            "pub_year": int,
            "url": string,
            "isbn": string,
            "sales": int,
            "rate": double
        },
    ]
}
```

2.3.5 构件 5：书籍详情弹出界面

2.3.5.1 功能描述

用户点击书籍卡片后，弹出该界面。展示书籍详细信息，同时允许用户将其加入购物车（指定数量），以及加入收藏夹。

2.3.5.2 对象模型

前端建模：用户 User 属性，以及用户相关服务：由用户名获取用户信息、用户认证、添加购物车、添加收藏夹。

```
export interface User{
  id: number;
  name: string;
  pwdhash: string;
```

```
  email: string;
  tel: string;
}
export class UserService {
  private me!: User;
  constructor(username: string) {
    this.getUser(username);
  }
  async getUser(username: string) {
}
  async addToCart(bookID: number, amount: number = 1) {
  }
  async addtoFavors(bookID: number) {
}
```

后端模型：

```
class Collection(Model):
user = ForeignKey(Users, on_delete=CASCADE, help_text="用户序号", db_column='uid')
book = ForeignKey(Books, on_delete=CASCADE, help_text="书的序号",
db_column='book_id')
class Meta:
constraints = [
UniqueConstraint(fields=['user', 'book'], name='collection_id'),
    ]
db_table = 'collection'
class CollectionSerializer(ModelSerializer):
user = UsersSerializer(read_only=True)
uid = PrimaryKeyRelatedField(queryset=Collection.objects.all(), write_only=True,
help_text="用户序号")
book = BooksSerializer(read_only=True)
book_id = PrimaryKeyRelatedField(queryset=Collection.objects.all(), write_only=True,
help_text="书的序号")
class Meta:
model = Collection
fields = '__all__'

class Shoppingcarts(Model):
user = ForeignKey(Users, on_delete=CASCADE, help_text="用户序号", db_column='uid')
book = ForeignKey(Books, on_delete=CASCADE, help_text="书的序号",
db_column='book_id')
amount = IntegerField(null=False, default=1, help_text="购买数量")
class Meta:
constraints = [
UniqueConstraint(fields=['user', 'book'], name='shoppingcarts_id'),
CheckConstraint(check=Q(amount__gt=0), name="amount__gt=0")
```

```
25. ]
26. db_table = 'shoppingcarts'
27. class ShoppingcartsSerializer(ModelSerializer):
28. user = UsersSerializer(read_only=True)
29. uid = PrimaryKeyRelatedField(queryset=Shoppingcarts.objects.all(), write_only=True,
help_text="用户序号")
30. book = BooksSerializer(read_only=True)
31. book_id = PrimaryKeyRelatedField(queryset=Shoppingcarts.objects.all(),
write_only=True, help_text="书的序号")
32. class Meta:
33. model = Shoppingcarts
34. fields = '__all__'
35.
```

后端主要视图：

```
1. class ShoppingcartsViewSet(ModelViewSet):
2. queryset = Shoppingcarts.objects.all()
3. serializer_class = ShoppingcartsSerializer
4. create = get_create(Shoppingcarts)
5.     def update(self, request, *args, **kwargs):
6.         partial = kwargs.pop('partial', False)
7.         try:
8.             instance = self.get_object()
9.             instance.uid = request.data.pop('uid')
10.            instance.book_id = request.data.pop('book_id')
11.            instance.amount = request.data.pop('amount')
12.            instance.save()
13.        except Exception as e:
14.            return Response({'error': str(e)}, status=HTTP_400_BAD_REQUEST)
15.        serializer = self.get_serializer(instance, partial=partial)
16.        return Response(serializer.data)
17.
18.    last = get_last(Shoppingcarts)
19.    custom_filter = get_custom_filter(Shoppingcarts)
20. class CollectionViewSet(ModelViewSet):
21.    queryset = Collection.objects.all()
22.    serializer_class = CollectionSerializer
23.
24.    create = get_create(Collection)
25.    last = get_last(Collection)
26.    custom_filter = get_custom_filter(Collection)
```

设计简述：

后端利用 Shoppingcarts 和 Collection 视图集的 create 方法，将前端参数转换

并存入数据库中。

2.3.5.3 接口规范

将书籍加入用户购物车：

请求的 URL：/bookStore/api/shoppingcarts/

请求方式：POST

请求参数：如下表 5 书籍加入购物车请求参数

字段名称	类型	允许空	缺省值	主外键	描述	约束
book_id	NUMBER	N		PK	书序号	
uid	STRING	N		PK	用户序号	
amount	NUMBER	N			添加数量	>0

表 5 书籍加入购物车请求参数

将书籍加入用户收藏夹：

请求的 URL：/bookStore/api/collection/

请求方式：POST

请求参数：如下表 6 数据加入收藏夹请求参数

字段名称	类型	允许空	缺省值	主外键	描述	约束
book_id	NUMBER	N		PK	书序号	
uid	NUMBER	N		PK	用户序号	

表 6 数据加入收藏夹请求参数

返回参数：无

2.3.6 构件 6：获取收藏夹

2.3.6.1 功能描述

根据用户的输入来返回用户期望查询的个人收藏夹内容。

2.3.6.2 对象模型

前端模型：

```
1. interface collectionUnit {
2.    collection_id: string,
3.    book_id: string,
4.    bname: string,
```

```
    author: string,
    category: string,
    price: string,
    pub_year: string,
    pname: string,
    rate: string,
    ImgAddress: string,
}
const collectionList = reactive<collectionUnit[]>([]);

```

后端模型：Collection

设计简述：

前端构建完整的收藏夹模型；由于该构件的主要功能为查询当前用户收藏夹内所有的书籍信息，而所有的书籍信息存在数据库中，故前端只需要构建一个收藏夹模型来接受后端返回的参数即可。由于收藏夹与用户绑定，因此需要向后端输入当前用户的 user_id。

2.3.6.3 接口规范

请求的 URL：/bookStore/api/collection/custom_filter

请求方式：GET

请求参数：如下表 7 获取收藏夹数据请求参数

参数名称	参数类型	参数描述	是否必须
id	int	自增主键	N
uid	int	用户的 id	Y
book_id	int	收藏的书的 id	N

表 7 获取收藏夹数据请求参数

返回参数：

```
[{
    “id”: int,
    “user”:{
            "uid": int,
            "uname": string,
            "pwdhash": string,
            "email": string,
            "tel": string,
            },
    “book”:{
```

```
            "book_id": int,
            "authors": {
                "author_id": int,
                "aname": string
            },
            "publishers": {
                "publisher_id": int,
                "pname": string
            },
             "category": {
                "category_id": int,
                "category_name": string
            },
            "bname": string,
            "price": double,
            "pub_year": int,
            "url": string,
            "isbn": string,
            "sales": int,
            "rate": double
            },
]
```

2.3.7 构件 7：删除收藏夹中的内容

2.3.7.1 功能描述

对于收藏夹中的东西，用户可能随时会想将某一本书移出收藏夹，该功能便可以帮助用户便捷地将待操作的书籍移出收藏夹。

2.3.7.2 对象模型

前端模型：

```
interface collectionUnit {
    collection_id: string,
    book_id: string,
    bname: string,
    author: string,
    category: string,
    price: string,
    pub_year: string,
    pname: string,
    rate: string,
    ImgAddress: string,
```

```
12. }
13. const collectionList = reactive<collectionUnit[]>([]);
14.
```

后端模型：Collection

设计简述：

前端根据页面删除按钮得到的 index 参数，找到收藏夹列表的指定元素，取出其 book_id,即 collectionList[index]. collection_id。再向后端接口进行删除请求，从而删除数据库收藏夹表中的该条记录。输入只需要 collection_id 即可。

2.3.7.3 接口规范

请求的 URL：/bookStore/api/collection/{id}/

请求方式：DELETE

请求参数：表 8 删除收藏夹请求参数

参数名称	参数类型	参数描述	是否必须
id	int	在收藏夹中的序号	Y

表 8 删除收藏夹请求参数

返回参数：无

2.3.8 构件 8：个人信息中心

2.3.8.1 功能描述

根据当前用户，获取用户的个人信息。

2.3.8.2 对象模型

前端模型：

```
1. interface UserData {
 2.   name: string
 3.   phoneNumber: string
 4.   email: string
 5.   address: string
 6. }
 7. const userdata = reactive<UserData>({
 8.   name: '',
 9.   phoneNumber: '',
 9.   email: '',
10.   address: '',
```

```
11.})
```

后端模型：Users

设计简述：

前端构建完整的用户信息模型；前端根据当前用户的 ID，向后端发送指定 uid 的用户信息访问请求，再收到后端 Users 视图集返回的数据后，填入变量，进行显示。输入仅用户 ID。

2.3.8.3 接口规范

请求的 URL：/bookStore/api/users/{uid}/

请求方式：GET

请求参数：如下表 9 获取个人信息请求参数

参数名称	参数类型	参数描述	是否必须
uid	int	用户 ID	Y

表 9 获取个人信息请求参数

返回参数：

```
{
2.    “uid”: int,
3.    “uname”: string,
4.    “pwdhash”: string/null,
5.    “email”: string/null,
6.    “tel”: string/null,
7. }
```

2.3.9 构件 9：个人购物历史

2.3.9.1 功能描述

根据当前用户的需求，返回其购物历史

2.3.9.2 对象模型

前端模型：

```
1. interface shoppingHistoryUnit {
 2.   book_id: string,
 3.   bname: string,
 4.   author: string,
 5.   category: string,
 6.   price: string,
```

```
  pub_year: string,
  pname: string,
  rate: string,
  ImgAddress: string,
  buy_date: string,
  amount: string,
}
const shoppingHistoryList= reactive<shoppingHistoryUnit>([])
```

后端模型：

```
class Shoppinghistory(Model):
    user = ForeignKey(Users, on_delete=CASCADE, help_text="用户序号", db_column='uid')
    book = ForeignKey(Books, on_delete=CASCADE, help_text="书的序号",
db_column='book_id')
    date = DateTimeField(auto_now_add=True, help_text="购买日期")
    amount = IntegerField(null=False, default=1, help_text="购买数量")

    class Meta:
        db_table = 'shoppinghistory'
class ShoppinghistorySerializer(ModelSerializer):
    user = UsersSerializer(read_only=True)
    uid = PrimaryKeyRelatedField(queryset=Shoppinghistory.objects.all(),
write_only=True, help_text="用户序号")
    book = BooksSerializer(read_only=True)
    book_id = PrimaryKeyRelatedField(queryset=Shoppinghistory.objects.all(),
write_only=True, help_text="书的序号")

    class Meta:
        model = Shoppinghistory
        fields = '__all__'
```

设计简述：

前端构建完整的访问历史列表模型；前端根据当前用户的 ID，向后端发送指定 uid 的访问历史查询请求，再收到后端利用 Shoppinghistory 视图集返回的数据后，填入列表，由页面显示即可。输入仅需要用户 ID。

2.3.9.3 接口规范

请求的 URL：/bookStore/api/shoppinghistory/custom_filter/

请求方式：GET

请求参数：如下表 10 获取购物历史请求参数

参数名称	参数类型	参数描述	是否必须

user_id	int	用户 ID	Y

表 10 获取购物历史请求参数

返回参数：

```
1.[{
 2.    “id”: int,
 3.    “user”:{
 4.          "uid": int,
 5.          "uname": string,
 6.          "pwdhash": string,
 7.          "email": string,
 8.          "tel": string,
 9.          },
10.    “book”:{
11.           “book_id”: int,
12.           “authors”: {
13.               “author_id”: int,
14.               “aname”: string
15.           },
16.           “publishers”: {
17.               “publisher_id”: int,
18.               “pname”: string
19.           },
20.            “category”: {
21.               “category_id”: int,
22.               “category_name”: string
23.           },
24.           “bname”: string,
25.           “price”: double,
26.           “pub_year”: int,
27.           “url”: string,
28.           “isbn”: string,
29.           “sales”: int,
30.           “rate”: double
31.    },
32.    “date”: datetime
33.    “amount”: int
34. },
35.]
```

2.3.10 构件 10：获取购物车信息

2.3.10.1 功能描述

用户可查询自己的购物车中的内容

2.3.10.2 对象模型

前端对象：

```
interface tableUnit {
  basket_id: string
  book_id: string
  bname: string
  author: string
  author_id: string
  pub_id: string
  category_id: string
  category: string
  price: string
  pub_year: string
  pname: string
  rate: string
  ImgAddress: string
  isbn: string
  sales: string
  amount: string
}
const tableData = reactive<tableUnit[]>([])
```

后端模型：Shoppingcarts

设计简述：

前端构建了一个存储购物车信息的列表模型，该列表内的每个对象又存储了具体的书籍信息。根据用户 ID，前端向后端发送购物车表查询请求后，后端 Shoppinghistory 视图集返还数据，前端将收到的数据依次填入表中，用于显示以及后续的删除、支付。

2.3.10.3 接口规范

请求的 URL：/bookStore/api/shoppingcarts/custom_filter/

请求方式：POST

请求参数：如下表 11 获取购物车数据参数

参数名称	参数类型	参数描述	是否必须
id	int	该条记录在购物车中的序号	N
user_id	int	用户的 id	Y
book_id	int	购买的书籍的 id	Y
amount	int	购买的数量	N

表 11 获取购物车数据参数

返回参数：

```
[
    {
        “id”: int,
        “user”: {
            “uid”: int,
            “uname”: string,
            “pwdhash”: string,
            “email”: string,
            “tel”: string
        },
        “book”: {
            “book_id”: int,
            “author”:{
                “author_id”: int,
                “aname”: string
            },
            “publisher”:{
                “pub_id”: int,
                “pname”: string
            },
            “category”:{
                “category_id”: int,
                “category_name”: string
            },
            “bname”: string,
            “price”: double,
            “pub_year”: int,
            “url”: string,
            “isbn”: string,
            “sales”: int,
            “rate”: double
        },
        “amount”: int
    }
```

```
35. ]
```

2.3.11 构件 11：添加购物车

2.3.11.1 功能描述

该构件使得用户在浏览书籍时可将想购买的书籍添加进购物车。

2.3.11.2 对象模型

前端模型：

```
1. interface ShowBookData {
 2.   id: string,
 3.   name: string,
 4.   ImgAddress: string,
 5.   author: string,
 6.   publisher: string,
 7.   date: string,
 8.   price: number,
 9.   buyNumber: number,
10.   type: string,
11.   rate: number,
12. }
13. const ShowingBook = reactive<ShowBookData>({
14.   id: '',
15.   name: '',
16.   ImgAddress: '',
17.   author: '',
18.   publisher: '',
19.   date: '',
20.   price: 0,
21.   buyNumber: 1,
22.   type: '',
23.   rate: 0,
24. })
25.
```

后端模型：Shoppingcarts

设计简述：

前端构建了一个当前显示的书籍模型，该模型包含弹窗显示书籍所需的必要信息。其中的 id，即书籍 ID，结合当前用户 ID，即可向后端请求将该书籍添加到当前用户的购物车中。具体流程中，前端会先根据书籍 ID，用户 ID 查询购物

车中是否已有相关记录，如果没有，则调用后端的 create 接口创建新的购物车记录；反之，则调用后端的 update 接口，将当前购买数量加到原来的数量上。

2.3.11.3 接口规范

(a)首先查询购物车表中是否存在指定用户 ID，书籍 ID 记录：

请求的 URL：/bookStore/api/shoppingcarts/custom_filter/

请求方式：GET

请求参数：如下表 12 查询购物车指定数据

参数名称	参数类型	参数描述	是否必须
user_id	string	用户 ID	Y
book_id	string	书籍 ID	Y

表 12 查询购物车指定数据

返回参数(列表)：

```
[{
    “id”: int,
    “user”:{
          "uid": int,
          "uname": string,
          "pwdhash": string,
          "email": string,
          "tel": string,
          },
    “book”:{
           “book_id”: int,
           “authors”: {
               “author_id”: int,
               “aname”: string
           },
           “publishers”: {
               “publisher_id”: int,
               “pname”: string
           },
            “category”: {
               “category_id”: int,
               “category_name”: string
           },
           “bname”: string,
           “price”: double,
           “pub_year”: int,
```

```
27.            “url”: string,
28.            “isbn”: string,
29.            “sales”: int,
30.            “rate”: double
31.            },
32.    “amount”: int
33. },
34.]
```

(b)若购物车表中无指定用户 ID，书籍 ID 记录：

请求的 URL：/bookStore/api/shoppingcarts/

请求方式：POST

请求参数：如下表 13 创建购物车数据请求参数

参数名称	参数类型	参数描述	是否必须
uid	string	用户 ID	Y
book_id	string	书籍 ID	Y
amount	string	购买数量	Y

表 13 创建购物车数据请求参数

返回参数：无

(c)若购物车表中已经指定用户 ID，书籍 ID 记录（重复）：

请求的 URL：/bookStore/api/shoppingcarts/{id}/

请求方式：PUT

请求参数：如下表 14 更新购物车数据请求参数

参数名称	参数类型	参数描述	是否必须
uid	string	用户 ID	Y
book_id	string	书籍 ID	Y
amount	string	购买数量	Y
id	string	购物车中条目 ID	Y

表 14 更新购物车数据请求参数

返回参数：无

2.3.12 构件 12：支付

2.3.12.1 功能描述

该构件可帮助用户在选好书籍后可购买该书籍。在购物车中删除该条目，后端触发器会自动将其添加入购物历史表单中。

2.3.12.2 对象模型

前端模型：

```
interface tableUnit {
  basket_id: string
  book_id: string
  bname: string
  author: string
  author_id: string
  pub_id: string
  category_id: string
  category: string
  price: string
  pub_year: string
  pname: string
  rate: string
  ImgAddress: string
  isbn: string
  sales: string
  amount: string
}
const tableData = reactive<tableUnit[]>([])
```

后端模型：Shoppingcarts、Shoppinghistory

后端主要函数：

```
def update(self, request, *args, **kwargs):
    partial = kwargs.pop('partial', False)
    try:
        instance = self.get_object()
        instance.uid = request.data.pop('uid')
        instance.book_id = request.data.pop('book_id')
        instance.amount = request.data.pop('amount')
        instance.save()
    except Exception as e:
        return Response({'error': str(e)}, status=HTTP_400_BAD_REQUEST)
    serializer = self.get_serializer(instance, partial=partial)
```

```
12.     return Response(serializer.data)
```

设计简述：

支付含三个部分：

1. 删除用户购物车中的指定条目。前端只需要知道该条目在购物车表的序号，即 basket_id。向后端指定接口发送该信息，即可删除该条目。

2. 在 shopping_history 表中记录此次支付操作。该部分主要由数据库的触发器自动实现。

3. 修改购买书籍的销量；前端需要修改指定 book_id 的数据库信息，即进行 update 操作。

2.3.12.3 接口规范

(a)修改所购买书籍的销量

请求的 URL：/bookStore/api/books/{book_id}/

请求方式：PUT

请求参数：如下表 15 修改书籍销量请求参数

参数名称	参数类型	参数描述	是否必须
book_id	String	书籍 ID	Y
author_id	String	作者 ID	Y
pub_id	String	出版社 ID	Y
category_id	String	分类 ID	Y
bname	String	书名	Y
price	String	价格	Y
pub_year	String	出版年份	Y
url	String	图片网址	Y
isbn	String	ISBN	Y
sales	int	销售量	N
rate	String	评分	Y

表 15 修改书籍销量请求参数

返回参数：无

(b)删除购物车中指定条目

请求的 URL：/bookStore/api/shoppingcarts/{id}/

请求方式：DELETE

请求参数：如下

参数名称	参数类型	参数描述	是否必须
id	String	购物车中条目 ID	Y

表 16 删除购物车数据请求参数

返回参数：无

2.4 构件合作模型

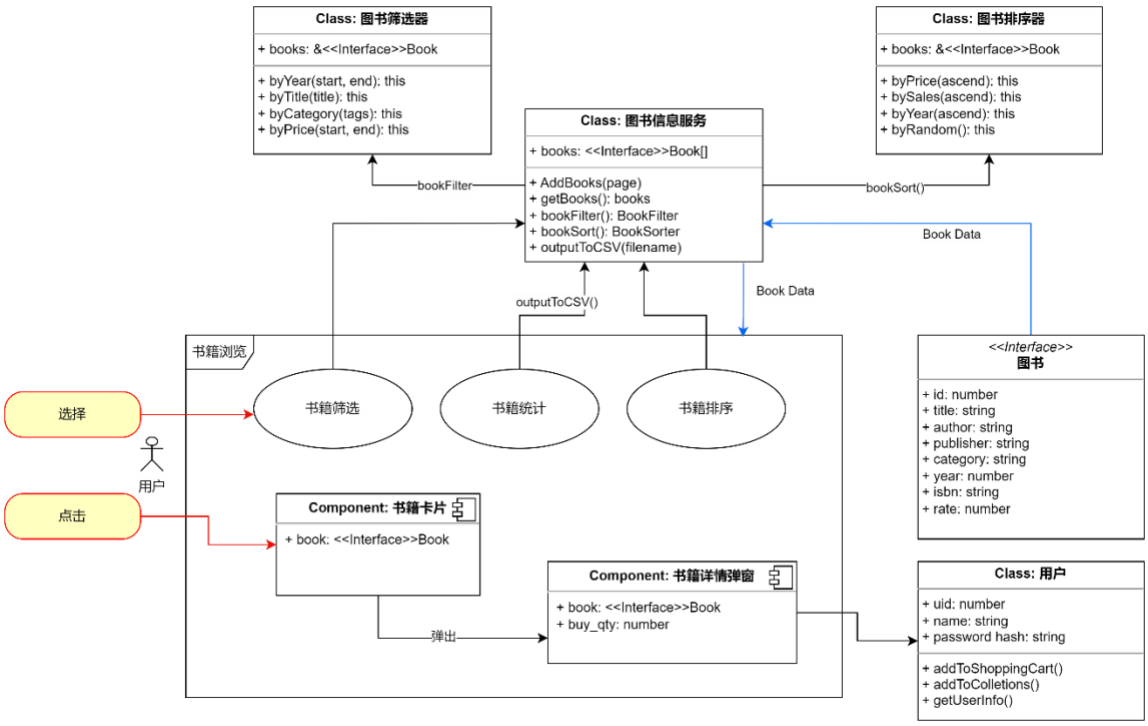

图 2 书籍浏览页面构件合作

2.5 图形用户接口设计

前端图形界面分为 7 大功能页面：用户登录与注册、首页、高级搜索界面、书籍浏览界面、收藏夹、个人中心界面、购物车界面；除登录界面外，所有界面均有一致的导航头部区域。用户应首先注册登录系统，进入首页。

软件图形界面使用同一蓝白色调，同一字体与样式，保证视觉一致性。以用户为中心设计功能组件，保证功能的直观和无歧义，同时提供充分的交互式设计，用户使用流程逻辑流畅，较好展示了本作品核心数据库的功能。

2.5.1 导航区域：

导航栏设于界面顶部，中央位置醒目展示 LOGO，该 LOGO 作为首页入口链接，用户可通过点击回到主页或导航至其他页面。

2.5.2 用户登录与注册界面

用户登录页面（图 3 登录界面）有两个输入栏，分别输入用户名和密码（会实时隐藏），以及两个功能按钮：登录和注册。

点击注册即可跳转用户注册界面（图 4 注册界面）。

图 3 登录界面

图 4 注册界面

2.5.3 首页

在主页（图 5 首页界面）上，系统随机推荐书籍，并以交互式卡片形式展现，每张卡片包含书籍封面图和基础信息。用户点击任一卡片即可激活显示书籍详细信息的弹窗（），弹窗内提供购买数量选择输入框及“加入购物车”和“加入收藏夹”功能键。

图 5 首页界面

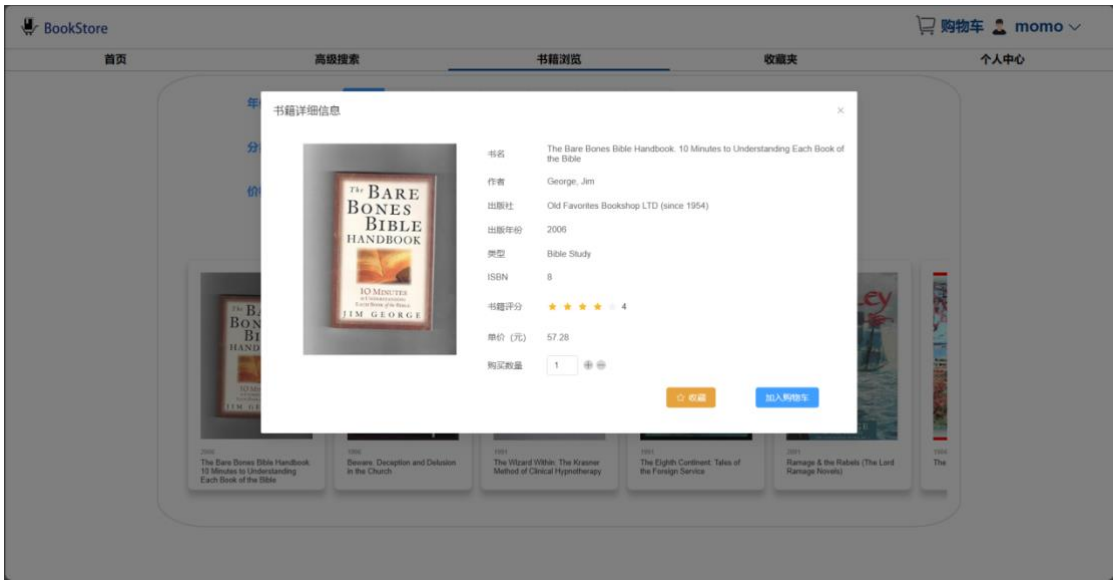

图 6 弹出书籍详情窗口

2.5.4 高级搜索界面

用户可在高级搜索界面（图 7 高级搜索界面）中利用多个输入栏（如作者、书名、类别等）细分搜索图书。该界面还设有时间范围选择器，以及“提交搜索”

和“清空搜索条件”功能键。搜索结果在点击提交按钮后显示在页面下方。

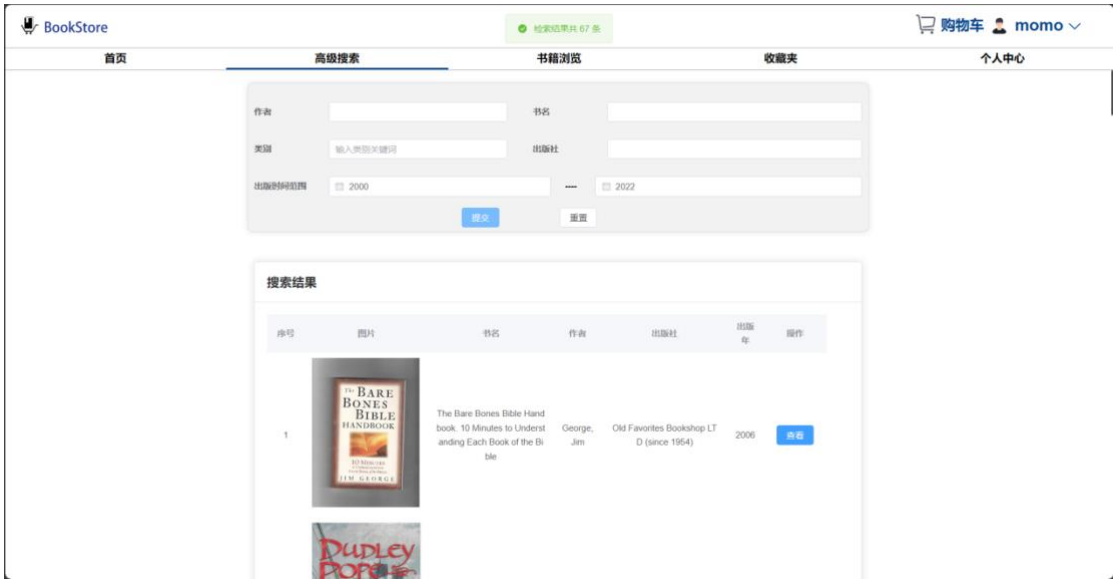

图 7 高级搜索界面

2.5.5 书籍浏览界面

如图 8 书籍浏览界面，三个筛选栏（互斥功能按钮），用户可分别选择年份（入 1980 年代、1990 年代）、分类（历史、人文等）、价格（30-50 元、50-100 元）等筛选范围，筛选结果实时更新在下方图书展廊区域。用户可点击全部按钮来重置筛选结果。

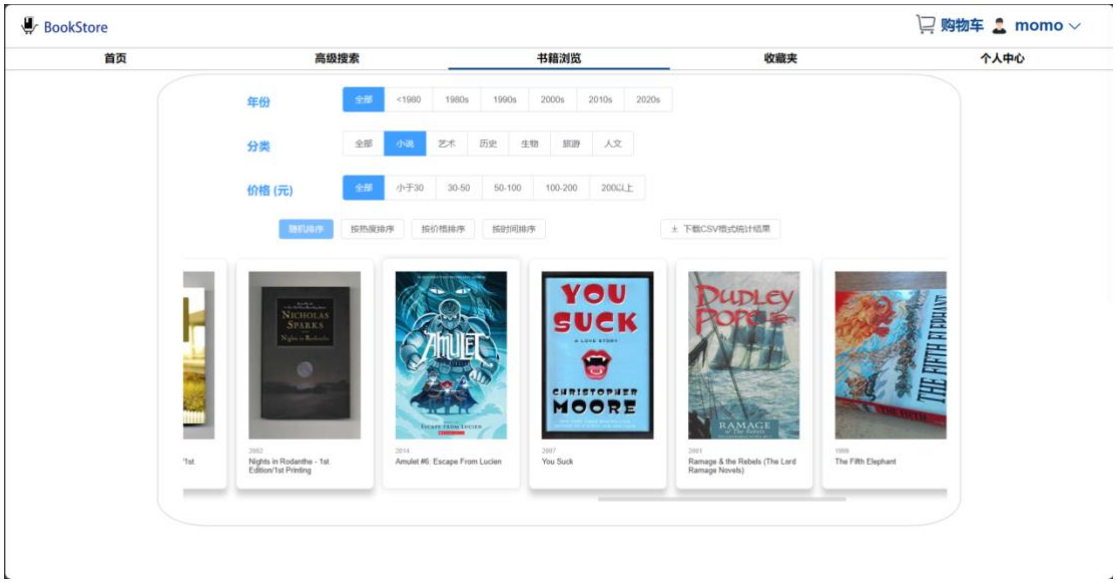

图 8 书籍浏览界面

四个排序功能按钮，分别为随机排序、按销量排序、按价格排序、按时间排序。

一个统计功能按钮，允许用户将当前筛选与排序结果导出为 CSV 文件。

图书展廊区域由多个图书卡片构成，卡片包括书籍封面图片、年份以及标题，用边缘阴影变化强化点击效果，点击后弹出书籍详情窗口（构件 12）。

2.5.6 用户收藏夹界面

用户在书籍详情弹窗中添加对书籍的收藏后，此处会显示当前用户的收藏的图书，以及图书一些简要信息。用户可以查看和删除条目，如图 9 用户收藏夹界面。

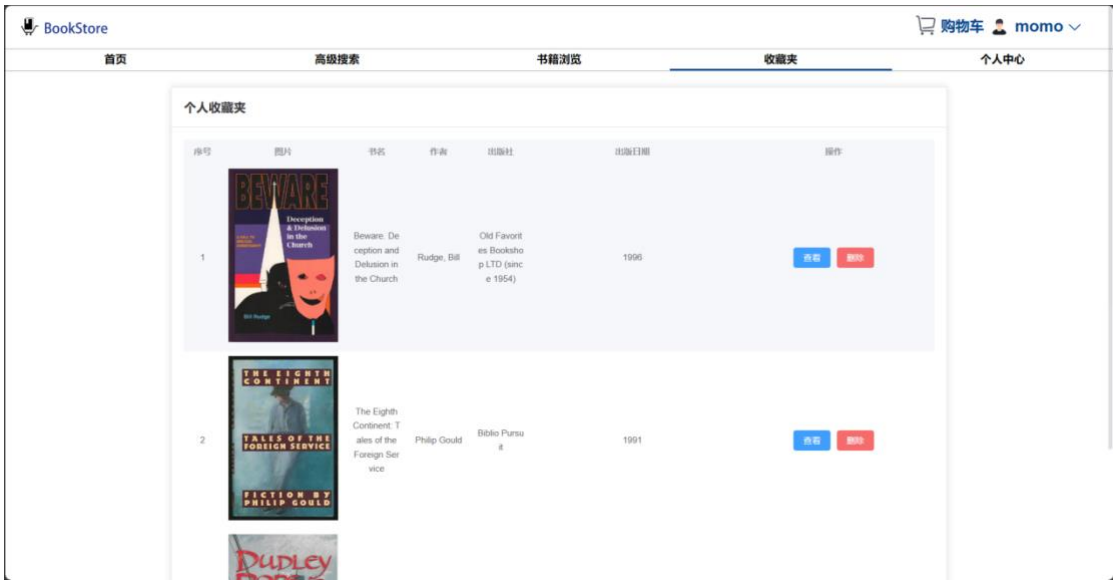

图 9 用户收藏夹界面

2.5.7 用户购物车界面

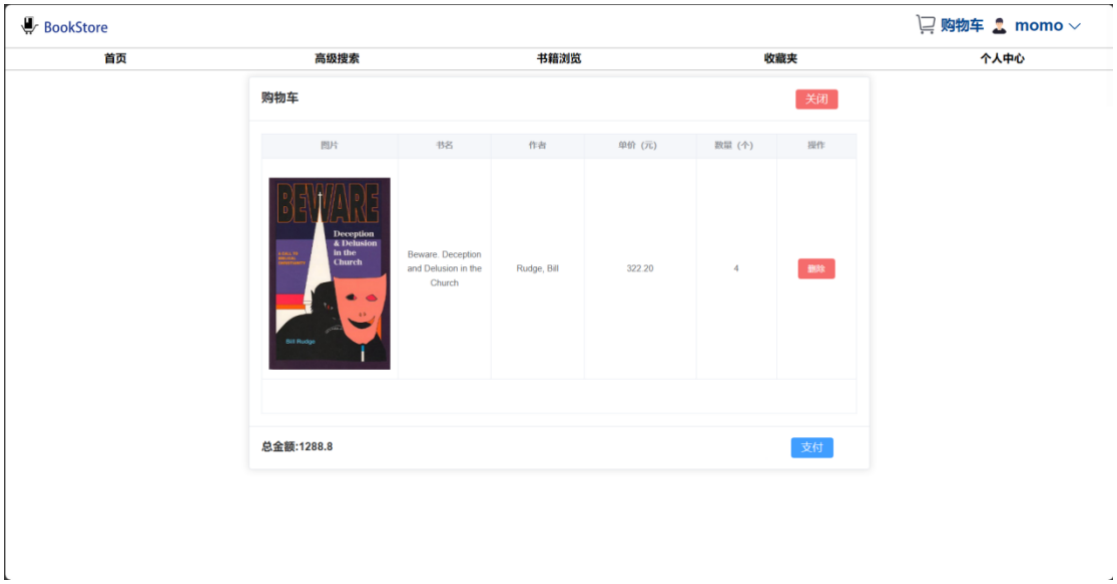

图 10 用户购物车界面

用户在书籍详情弹窗中将书籍添加入购物车，此处会显示购物车全部书籍以

及用户选择的购买数量。有功能按钮：关闭和支付，以及删除购物车中某个书籍条目。

2.5.8 用户个人中心界面

如图 11 用户个人信息界面，用户信息卡片展示用户详细信息，包括头像、用户名、电话邮箱等。

列表显示当前用户收藏的书单，列表显示用户的书籍购买历史。

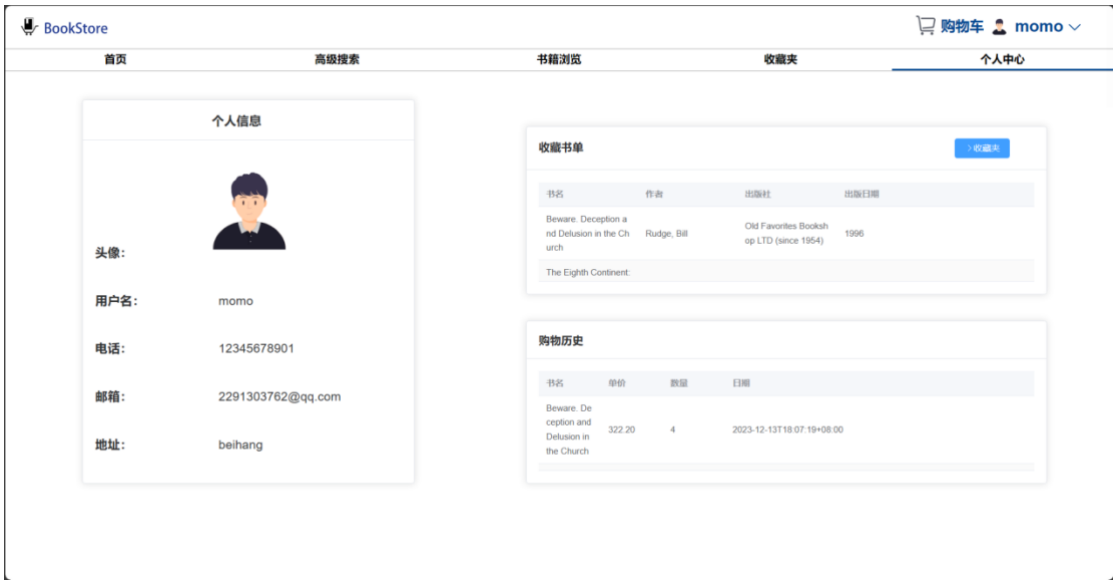

图 11 用户个人信息界面

2.6 数据库设计

2.6.1 对象永久存储策略

本项目中，采用 MySQL 数据库进行对象的永久存储。

2.6.2 数据库表设计

books：如下表 17 books 设计表

字段名称	类型	允许空	缺省值	主外键	描述	约束
book_id	INT	N		PK	序号	
bname	VARCHAR(255)	N			书名	
author_id	INT	N		FK	作者	
pub_id	INT	N		FK	出版社	

category_id	INT	N		FK	种类	
price	DECIMAL(10,2)	N			价格	
pub_year	INT	N			出版年	
url	VARCHAR(255)	N			网址	
isbn	CHAR(10)	N			isbn 号	
sales	INT	Y	0		销售量	
rate	DECIMAL(2,1)	N			评分	

表 17 books 设计表

- book_id：
  - 字段描述：书籍的 id，用来方便标识每个书
  - 字段格式：整数
  - 取值范围：[0，100000]
- bname：
  - 字段描述：书籍名称
  - 字段格式：字符串
  - 取值范围：无
- author_id：
  - 字段描述：作者 id，用来标识每个书的作者，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- pub_id：
  - 字段描述：出版社 id，用来标识每个书的出版社，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- category_id：
  - 字段描述：种类 id，用来标识每个书所属的种类，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- price：
  - 字段描述：该书在市面上对应的价格，以人民币为单位

- 字段格式：浮点数，保留两位小数
- 取值范围：[0.01，3000000.99]

- pub_year：
  - 字段描述：出版年份
  - 字段格式：合法的 4 位数年份
  - 取值范围：[1900，2023]
- url：
  - 字段描述：该书的图片对应的网址
  - 字段格式：https://pictures.abebooks.com/xxxxxx.jpg
  - 取值范围：无
- isbn：
  - 字段描述：该书对应的 10 位 isbn 号
  - 字段格式：10 位长的字符串，将前 9 位数字依序分别乘以 10 到 2 的数目，将其乘积相加，综合用 11 去除，若无余数则检查号码为 0，否则，以 11 减去余数，所得差数即为检查好，若差数为 10 则以 X 表示
  - 取值范围：无
- sales：
  - 字段描述：该书的销售量
  - 字段格式：整数
  - 取值范围：[0，1000000]
- rate：

  字段描述：该书的全网评分

  字段格式：浮点数，保留 1 位小数

  取值范围：[0.0，5.0]

authors：如下

字段名称	类型	允许空	缺省值	主外键	描述	约束
author_id	INT	N		PK	序号	
aname	VARCHAR(255)	N			作者名	

表 18 authors 设计表

- author_id：

- 字段描述：作者的 id，用来方便标识每本书的作者
- 字段格式：整数
- 取值范围：[0，100000]

- aname：
  - 字段描述：作者的姓名
  - 字段格式：字符串
  - 取值范围：无

publishers：如下表 19 publishers 设计表

字段名称	类型	允许空	缺省值	主外键	描述	约束
publisher_id	INT	N		PK	序号	
pname	VARCHAR(255)	N			出版社名	

表 19 publishers 设计表

- publisher_id：
  - 字段描述：出版社的 id，用来方便标识每本书的出版社
  - 字段格式：整数
  - 取值范围：[0，100000]

- pname：
  - 字段描述：出版社的名称
  - 字段格式：字符串
  - 取值范围：无

category：如下表 20 category 表设计

字段名称	类型	允许空	缺省值	主外键	描述	约束
category_id	INT	N		PK	序号	
category_name	VARCHAR(255)	N			类名	

表 20 category 表设计

- category_id：
  - 字段描述：种类的 id，用来方便标识每本书所属的种类
  - 字段格式：整数
  - 取值范围：[0，100000]

- category_name：

- 字段描述：种类的名称
- 字段格式：字符串
- 取值范围：无

users：如下表 21 users 设计表

字段名称	类型	允许空	缺省值	主外键	描述	约束
uid	INT	N		PK	序号	
uname	VARCHAR(100)	N			用户名	
pwdhash	VARCHAR(255)	N			口令哈希	
email	VARCHAR(100)	Y			邮箱	
tel	CHAR(11)	Y			电话	

表 21 users 设计表

- uid：
  - 字段描述：用户 id，用来方便标识每个用户
  - 字段格式：整数
  - 取值范围：[0，100000]
- uname：
  - 字段描述：用户昵称
  - 字段格式：字符串
  - 取值范围：无
- pwdhash：
  - 字段描述：用户登录密码的哈希值，哈希算法采用 sha256
  - 字段格式：长度为 64 位的 16 进制串（即字符出只包含 0-9 的数字和 A-F 的大写字母）
  - 取值范围：无
- email：
  - 字段描述：用户的邮箱
  - 字段格式：符合邮箱格式的字符串，本系统的合法邮箱后缀包括：@163.com、@qq.com、@gmail.com、@mail.hk.com、@yahoo.co.id
  - 取值范围：无
- tel：

- 字段描述：用户的电话
- 字段格式：符合中国电话号码格式的 11 位字符串
- 取值范围：无

shoppinghistorys：如下表 22 shoppinghistory 设计表

字段名称	类型	允许空	缺省值	主外键	描述	约束
id	INT	N		PK	标记	
uid	INT	N		FK	用户	
book_id	INT	N		FK	书	
date	DATETIME	N			日期	
amount	INT	N			购买量	

表 22 shoppinghistory 设计表

- id：
  - 字段描述：用来标识每一条购物记录的主键，方便前后端的操作
  - 字段格式：整数
  - 取值范围：[0，100000]
- uid：
  - 字段描述：用户 id，用来标识该购物历史中买书用户，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- book_id：
  - 字段描述：书 id，用来标识该购物历史中用户所买的书，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- date：
  - 字段描述：购买日期
  - 字段格式：符合 mysql 的 datetime 类型，为 YYYY-MM-DD HH:MM:SS
  - 取值范围：无
- amount：

- 字段描述：本次购买的购买量
- 字段格式：大于 0 的整数
- 取值范围：>0

shoppingcarts：如下表 23 shoppingcarts 设计表

字段名称	类型	允许空	缺省值	主外键	描述	约束
uid	INT	N		PK	用户	
book_id	INT	N		PK	书	
amount	INT	N			量	>0

表 23 shoppingcarts 设计表

- uid：
  - 字段描述：用户 id，用来标识该购物车所属的用户，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- book_id：
  - 字段描述：书 id，用来标识该购物车中用户待购买的书，方便建立外键连接
  - 字段格式：整数
  - 取值范围：[0，100000]
- amount：
  - 字段描述：购买量
  - 字段格式：整数
  - 取值范围：>0

collections：如下

字段名称	类型	允许空	缺省值	主外键	描述	约束
uid	INT	N		PK	用户	
book_id	INT	N		PK	书	

表 24 collection 设计表

- uid：
  - 字段描述：用户 id，用来标识该收藏夹所属用户，方便建立外键连接
  - 字段格式：整数

    - 取值范围：[0，100000]
- book_id：
    - 字段描述：书 id，用来表示该收藏夹中用户收藏的书籍，方便建立外键连接
    - 字段格式：整数
    - 取值范围：[0，100000]

2.6.3 存储过程

在数据库建立过程中，通过自行编写爬虫脚本将书籍数据信息存储到本地的 csv 文件中，并对数据进行处理，将处理好的数据录入 MySQL 中完成数据初始化；在后续的使用过程中，通过数据库管理员来对数据进行增删操作。

2.6.4 触发器设计

设计目的：为了保证用户在删除一条购物车信息后，自动将对应书籍添加到购买历史中。

设计方法：在 shoppingcarts 表中添加一个 after delete 的触发器 shop，并将删除的数据中的 uid、book_id、amount 以及删除时的日期添加到表 shoppinghistory 中。

2.7 系统使用方法设计

1. git clone 本项目至本地。
2. 启动 MySQL 服务器，创建数据库 bookStore。
3. 进入前端项目 bookStore_vue，运行 npm run dev，启动前端服务器。
4. 进入后端项目 django，运行 python3 manage.py runserver 启动后端服务器。

2.8 公共函数库

2.8.1 Python

- Django
- Django-rest-framework
- coreapi

- pymysql

2.8.2 Vue

- axios@1.4.0
- element-plus@2.3.8
- vue@3.3.4

2.9 非功能性需求

2.9.1 安全性

本项目应满足的安全性包括：保密性、防泄露、访问控制。

2.9.2 易用性

本项目应易于用户使用，即前端页面不可过于复杂。

2.9.3 资源使用率

本项目的 CPU 占用率应小于等于 50%，内存占用应小于等于 50%。

2.9.4 兼容性需求

系统应支持 IOS、Android、Windows、MacOS 操作系统；最多只有 5%的系统实现需要具体到操作系统；替换关系数据库系统的平均时间不超过 2 小时，并且保证没有数据丢失。

2.10辅助工具

- Navicate
- Drawio
- Github

3. 系统详细设计

3.1 开发环境概述

3.1.1 版本控制工具

本项目中采用 GIT 作为版本控制工具。

3.1.2 编译环境

- 硬件平台：
  - 前端采用 AMD Ryzen 7 5800H with Radeon Graphics 及 NVIDIA GeForce RTX 3060 Laptop GPU
  - 后端及数据库采用 MacBook Pro 及 Apple M1 Pro。
- 操作系统：
  - 前端采用 Windows 11 Version 22H2
  - 后端及数据库采用 Sonoma 14.1.2。
- 开发工具与环境：
  - 开发工具：PyCharm 2023.2 、WebStorm 2023.2
  - 环境包括：Node.js v21.3.0、npm@10.2.4、Python3.10、mysql Ver 8.0.33

3.1.3 源程序目录

- 前端程序位于项目的/forend 文件夹中，目录结构如下：

```
bookstore_vues
├── LICENSE # 前端项目 license
├── README.md # 前端项目 README
├── public
├── src
│   ├── App.vue
│   ├── api # 常用 api
│   ├── assets
```

```
│   ├── components # 常用构件
│   ├── main.ts
│   ├── router # 路由配置
│   ├── store
│   ├── style.css
│   ├── utils # 基础服务
│   ├── views # 视图文件
│   └── vite-env.d.ts
├── index.html
├── package.json
├── tsconfig.json
└── vite.config.ts
```

- 后端程序位于项目的/backend/django 文件夹中，目录结构如下：

```
bookstore_django # 后端项目文件
├── backend_django # 后端项目基础目录
│   ├── asgi.py
│   ├── settings.py # 后端项目配置文件
│   ├── urls.py # 后端项目总 url
│   └── wsgi.py
├── bookStore # Django 应用
│   ├── migrations # 迁移文件
│   ├── admin.py
│   ├── apps.py
│   ├── models.py # 模型文件
│   ├── serializers.py # 序列化器文件
│   ├── tests.py
│   ├── urls.py # 应用 url
│   └── views.py # 视图文件
├── data_django.json # Django 可用的数据库数据文件
```

```
├── manage.py # Django 项目命令文件
├── LICENSE # 后端项目 license
└── README.md # 后端项目 README 文件
```

3.1.4 文档存储目录

各文档存储目录位于/docs 文件夹中，目录结构如下：

```
docs
├── sql_docs
│   ├── 概念设计.pdf
│   ├── 逻辑设计.pdf
│   └── 物理设计.xlsx
├── ~$系统详细设计.docx
└── 系统详细设计.docx
```

3.2 构件详细设计

3.2.1 构件 1：用户登录

本构建实现对用户输入账号密码的有效性进行检查，并在核验其有效后将数据放入本地缓存，进入主页面。此外还提供到注册页面的跳转功能。

3.2.1.1 源程序列表

ForeEnd/src/views/login.vue

3.2.1.2 设计思想

通过用 v-model 绑定的方式获取用户在网页输入的用户名和密码。随后根据用户名构造 GET 请求，向后端发起查询，确认是否存在这一用户。若用户存在，则从返回值中获取到对应的密码哈希值，并在本地对用户输入的密码进行哈希，若二者相同，则通过验证，将用户的 UID 等信息载入本地缓存，并跳转至主页面。

3.2.1.3 实现描述

以下是对主要函数 submitForm 的描述：

函数原型：submitForm = (formEl: FormInstance | undefined) => {}

功能描述：根据用户输入构建 GET 请求，向后端发送请求，并根据返回数据判断登录信息是否有效。

参数：输入参数为从后端获取用户信息的请求 URL。

输入情况：输入为表单实例类型变量 formEL。

输出情况：无返回值。

异常处理：若因网络原因，无法获取后端响应，则在控制台打印错误信息；若用户输入无效，则用弹窗的方式给出错误信息如“用户不存在”。

完整算法：

```
const submitForm = (formEl: FormInstance | undefined) => {
    if (!formEl) return;

    formEl.validate((valid: boolean) => {
        if (valid) {
            visitsUp();
            localStorage.setItem('ms_username', param.username);
            request.query.uname = param.username
            requestData(request)!.then((res) => {
              // 核验用户名是否存在
              if (res.data.length == 0){
                ElMessage.error('用户不存在！ ');
              } else{
                // 核验密码是否正确
                localStorage.setItem('ms_uid', res.data[0].uid);
                localStorage.setItem('ms_telnum', res.data[0].tel);
                console.log(localStorage.getItem('ms_uid'))
                impsha256(param.password)!.then((inputhash) =>{
                  console.log('input hash: ', inputhash)
                  console.log('backend hash: ', res.data[0].pwdhash)
                  if (inputhash == res.data[0].pwdhash){
                    ElMessage.success('登录成功！ ')
                    router.push('/home');
                  } else{
                    ElMessage.error('用户名或密码错误！ ')
                  }
                }).catch((hasherror) => {
                  console.log('Error in Calculating Hash')
                });
              }
            }).catch((error) => {
```

```
32.             // 请求出错处理
33.             console.log('Error : ', error)
34.           });
35.       } else {
36.
37.           ElMessage.error('操作失败');
38.           return false;
39.       }
40.   });
41. };
```

3.2.2 构件 2：用户注册

该构件获取用户注册输入信息，并核验用户输入是否有效，核验通过后向后端发送请求，往数据库中添加新的账户信息。

3.2.2.1 源程序列表

ForeEnd/src/views/register.vue

3.2.2.2 设计思想

通过 v-model 绑定的方式获取用户输入的账户信息。随后对用户输入合法性进行校验。校验通过后，构造 GET 请求并发送至后端，根据响应结果判断是否存在重名用户，若否，则用新账户信息构造 POST 请求发至后端，从而添加新的用户信息

3.2.2.3 实现描述

函数 1:

以下是对名称合法性检查函数 nameCheck 的介绍，在本构件中，需要进行内容合法性检查的的还有密码、电话号、邮箱，各自的检查逻辑都是一样的，区别仅在于使用的正则表达式不同，因此这里仅选其一进行介绍。

函数原型：const nameCheck = (rule: any, value: string, callback: (arg0: Error | undefined) => void)=> {}

功能描述：对用户注册使用的账户名进行合法性检查，合法的账户名应只包含英文字母与数字。

参数：输入参数为向后端进行查询的 URL、向后端获取 UID 的查询 URL，

以及向后端提交新用户信息的 URL。

输入情况：输入为规则对象 rule，字符串 value，回调函数 callback。

输出情况：无返回值。

完整算法：

```
1. const nameCheck = (rule: any, value: string, callback: (arg0: Error | undefined) =>
void)=> {
2.   const reg = /^[a-zA-Z0-9]+$/;
3.   if (reg.test(value)) {
4.     callback();
5.   } else {
6.     callback(new Error("用户名只能包含字母和数字"));
7.   }
8. }
```

函数 2:

以下是对注册函数 submitForm 的介绍：

函数原型：submitForm = (formEl: FormInstance | undefined) => {}

功能描述：根据用户输入信息，构建 GET 请求发向后端，根据返回结果判断是否存在重名用户，若不存在，则向后端请求新用户的 UID，并结合这一 UID 和输入的注册信息，向后端发送 POST 请求，以创建新用户。

输入情况：输入为表单实例类型变量 formEL。

输出情况：无返回值。

异常处理：若因网络原因，无法获取后端响应，则在控制台打印错误信息；若已存在同名用户，则用弹窗的方式给出提示信息。

完整算法：

```
1. const submitForm = async (formEl: FormInstance | undefined) => {
2.   if (!formEl) return
3.   console.log('input: ', ruleForm)
4.   await formEl.validate(async (valid: any, fields: any) => {
5.     if (valid) {
6.       // 查询是否存在重名
7.       request.query.uname = ruleForm.username;
8.       requestData(request)!.then((req_res) => {
9.         console.log('重名查询结果:  ', req_res);
10.        if (req_res.data.length > 0) {
11.          ElMessage({
12.            message: '用户重名，请重新输入',
```

```
            type: 'warning',
          });
        } else {
          // 查询 uid，然后 create
          getid.query.uname = ruleForm.username;
          requestData(getid)!.then((idres) => {
            console.log('get id: ', idres.data);
            const uidstring = idres.data.uid;
            const pwdstring = ruleForm.userpwd;
            const namestring = ruleForm.username;
            const telstring = ruleForm.telenum;
            // const addstring = ruleForm.address;
            const emailstring = ruleForm.email;
            console.log('reach 1 ');
            impsha256(pwdstring)!.then((hashValue) => {
              console.log('uid: ', uidstring);
              console.log('uname: ', namestring);
              console.log('password: ', pwdstring, ' SHA-256 Hash:', hashValue);
              console.log('email: ', emailstring);
              console.log('tel: ', telstring);
              createuser.query.uid = uidstring;
              createuser.query.uname = namestring;
              createuser.query.pwdhash = hashValue;
              createuser.query.email = emailstring;
              createuser.query.tel = telstring;
              requestData(createuser)!.then((creatres) => {
                console.log('post info: ', createuser);
                console.log('post response: ', creatres)
                ElMessage({
                  message: '注册成功! ',
                  type: 'success',
                });
                router.push('/login');
              }).catch((error) => {
                console.error('Error creating ', error)
              });
            }).catch((error) => {
              console.error('Error:', error)
            });
          }).catch((error) => {
            console.error('Error:', error);
          });
        }
      });
```

```
    }
  }).catch((error) => {
    console.error('Error:', error);
  });
}
```

3.2.3 构件 3：高级搜索

本构件支持用户通过包括作者名、书名、出版社、出版时间等多个条件对数据库中的书籍进行检索，将检索结果的简略信息陈列在页面，并让用户能够逐个查看检索结果的详细信息

3.2.3.1 源程序列表

ForeEnd/src/views/AdvancedSearch.vue

3.2.3.2 设计思想

在页面首次加载时构造 GET 请求并发送至后端，从后端获取到所有书籍的信息。通过 v-model 绑定的方式获取用户输入的筛选信息。当用户点击搜索后，获取用户输入的各个检索信息，将其用于筛选从前端获取到的书籍资料，并将筛选后的结果呈现到页面上。

3.2.3.3 实现描述

函数 1:

函数原型：const funcData = async () => {}

功能描述：向后端发起针对书籍信息列表的 GET 请求，并将结果写入 AllResults 变量。这一函数被挂载在 onMounted 中，以便让网页初次加载时就能获取到书籍信息。

参数：输入参数为从后端获取书籍信息的 URL，输出参数为前端的书籍信息列表变量 AllResults。

完整算法：

```
const funcData = async () => {
    let page = 1;

  while (page < 10) {
      // 修改请求的 page 参数
```

```
      request.query.page = String(page);

      try {
        // 发送请求
        const res = await requestData(request);

        console.log(res.data);
        // 处理返回的数据，这里可以将数据合并到 allbooks 数组中
        allbooks = allbooks.concat(res.data);
        // 增加 page 值
        page++;
      } catch (error) {
        // 发生错误时进行处理
        console.error('Error fetching data:', error);
        console.log(allbooks);
        ElMessage.error('Error Fetching Data');
        break;
      }
    }

  for (const book of allbooks) {
    // 检查每个子数组是否包含 results 属性
    if (book.results && Array.isArray(book.results)) {
      // 将当前子数组的 results 拼接到 allResults 中
      allresults = allresults.concat(book.results);
    }
}
    console.log(allresults);
}
```

函数 2:

以下是对提交函数 submitForm 的介绍：

函数原型：submitForm = (formEl: FormInstance | undefined) => {}

功能描述：根据用户输入信息，构件检索条件，对先前获取到的书籍数据进行筛选，并将筛选结果呈现到页面中。

输入情况：输入为表单实例类型变量 formEL。

输出情况：无返回值。

异常处理：若用户输入的检索条件不合法（如时间筛选中，起始时间晚于结束时间），则在页面上进行报错提示；若因网络原因无法从后端获取书籍数据，则通过弹窗的方式提示无法从后端获取数据；若检索出错，则通过弹窗提示错误

信息，并将其打印到控制台。

完整算法：

```
const submitForm = async (formEl: FormInstance | undefined) => {
  if (!formEl) return
  await formEl.validate(async (valid: any, fields: any) => {
    if (valid) {
      console.log('submit')
      const searchCriteria: { [key: string]: string } = {};

      // 遍历表单对象的属性
      for (const key in ruleForm) {
        if (ruleForm[key] !== undefined && ruleForm[key] !== null &&
ruleForm[key] !== '') {
          // 如果属性有值，将其添加到搜索条件中
          searchCriteria[key] = ruleForm[key];
          console.log("搜索条件：  ", searchCriteria)
        }
      }
      const searchResults = searchBooks(searchCriteria);
      const convertedResults = searchResults.map((result: any) => ({
        id: result.book_id,
        ImgAddress: result.url,
        name: result.bname,
        author: result.author.aname,
        publisher: result.publisher.pname,
        date: result.pub_year,
        price: result.price,
        type: result.category.category_name,
        rate: result.rate,
      }));

      BookListRef.value = convertedResults;
      console.log('检索结果（后端原始数据：', searchResults);
      ElMessage({
        message: '检索结果共 '+String(BookListRef.value.length)+' 条',
        type: 'success',
        })
    } else {
      console.log('error submit!', fields)
      ElMessage({
        message: '检索出错！ ',
        type: 'warning',
        });
```

```
41.       }
42.   })
43. }
```

函数 3:

函数原型：const searchBooks = (criteria: RuleForm): Array<any> => {}

功能描述：根据用户输入的表单数据与前端获取到的书籍数据，进行筛选，从而获取搜索结果，将其返回。从而为函数 2 提供支持。

输入：检索条件 criteria，其类型为 RuleForm（在代码中定义的接口）。

输出：一个数组，表示检索结果。

完整算法：

```
1. const searchBooks = (criteria: RuleForm): Array<any> => {
2.   // 存储匹配的书籍
3.   const matchingBooks: Array<any> = allresults.filter((book) => {
4.     // 检查书名是否匹配
5.     const isBookNameMatch = !criteria.bookName ||
book.bname.toLowerCase().includes(criteria.bookName.toLowerCase());
6.
7.     // 检查作者是否匹配
8.     const isAuthorMatch = !criteria.name ||
book.authors.aname.toLowerCase().includes(criteria.name.toLowerCase());
9.
10.    // 检查出版社是否匹配
11.    const isPublisherMatch = !criteria.Publisher ||
book.publishers.pname.toLowerCase().includes(criteria.Publisher.toLowerCase());
12.
13.    // 检查类别关键字是否匹配
14.    const isTypeMatch = !criteria.type ||
book.category.category_name.toLowerCase().includes(criteria.type.toLowerCase());
15.
16.    // 检查出版年份是否在范围内
17.    const startYear = parseInt(new Date(criteria.StartTime).toLocaleDateString(),
10);
18.    const endYear = parseInt(new Date(criteria.EndTime).toLocaleDateString(), 10);
19.
20.    // 时间为 null 时对应变量值为 NaN
21.    const isYearInRange = (
22.      (!criteria.StartTime || Number(book.pub_year) >= startYear) &&
23.      (!criteria.EndTime || Number(book.pub_year) <= endYear)
24.    );
25.
26.    // 如果所有条件都匹配，返回 true
```

```
    return isBookNameMatch && isAuthorMatch && isPublisherMatch && isYearInRange &&
isTypeMatch;
  });

  // 返回匹配的书籍数组
  return matchingBooks;
};
```

3.2.4 构件 4：书籍浏览

3.2.4.1 源程序列表

ForeEnd/src/views/BookService.vue

ForeEnd/src/components/bookCard.vue

ForeEnd/src/components/bookPopup.vue

ForeEnd/src/api/models.ts

ForeEnd/src/api/services.ts

3.2.4.2 设计思想

用面向对象的方法将页面模块化，分层进行抽象。具体建模及功能如下：

对象建模：将后端嵌套格式 json 数据扁平化，构造前端对象。

- 用户：用户信息以及用户认证
- 图书：图书相关信息

服务建模：

- 用户服务：验证用户认证状态，修改用户相关信息。
- 图书信息服务：缓存后端图书数据，提供统计功能。
- 图书检索服务：组合类，用于筛选图书数据，可链式调用。
- 图书排序服务：组合类，用于排序图书数据，可链式调用。

页面组件建模：

- 图书展廊，及界面功能按钮
- 图书卡片：用于在展廊中展示单本书籍
- 弹出图书详情窗口：点击卡片后，弹出该图书的详细信息。

3.2.4.3 实现描述

图书服务：

功能描述：

向后端发起针对书籍信息列表的 GET 请求，并在类实例中全局缓存该数据。提供了统计、排序、筛选、添加图书数据等功能。

方法列表：

- AddBooks(page)：从后端数据库获取更多图书数据缓存入类中。算法中包括 http 请求、数据接口转换等。

- getBooks()：从类中获取图书缓存。

- bFilter()：获取一个图书筛选器（类 BookFilter），使用方法后述。

- bSort()：获取一个图书排序器（类 BookSorter），使用方法后述。

- output2CSV()：静态方法，将图书统计数据输出到 CSV 文件中。

```
export class BookService {
  total: number = 0; // database 中书总数
  private books: Book[] = [];
  private currentPage: number = 1;

  // 从 database 拿新页
  async AddBooks(pageReq: number = 10) {
    for (let i = 0; i < pageReq; i++) {
      try {
        const res = await requestData({
          url: 'books/',
          method: 'get',
          query: {
            page: (this.currentPage + i).toString()
          }
        });
        const data = RawToBook(res?.data);
        this.books.push(...data.books);
        this.total = data.totalBooks;
        this.currentPage += 1;
      } catch (error) {
        console.error("add books failed", error)
        break;
      }
    }
    this.currentPage += pageReq;
  }

  getBooks(): Book[] {
```

```
30.     return this.books;
31.   }
32.
33.   bFilter(): BookFilter {
34.     return new BookFilter(this.books);
35.   }
36.
37.   bSort(): BookSorter {
38.     return new BookSorter(this.books);
39.   }
40.
41.   static output2CSV(filename: string, books: Book[]) {
42.     // filename 无需后缀
43.     const csvContent = books.map(book => {
44.       return [
45.         book.id,
46.         `"${book.title.replace(/"/g, '""')}"`,
47.         book.author,
48.         book.publisher,
49.         book.category,
50.         book.year,
51.         book.isbn,
52.         book.price,
53.         book.sales,
54.         book.url,
55.         book.rate
56.       ].join(",");
57.     }).join("\n");
58.
59.     const header =
"ID,Title,Author,Publisher,Category,Year,ISBN,Price,Sales,URL,Rate\n";
60.     const csvFile = header + csvContent;
61.
62.     const blob = new Blob([csvFile], { type: 'text/csv;charset=utf-8;' });
63.
64.     const url = URL.createObjectURL(blob);
65.     const link = document.createElement('a');
66.     link.href = url;
67.     link.setAttribute('download', `${filename}.csv`); //filename
68.
69.     link.style.visibility = 'hidden'; //hide the link
70.     document.body.appendChild(link);
71.     link.click();
72.
```

```
    document.body.removeChild(link);
    URL.revokeObjectURL(url);
  }
}
```

图书检索服务：

功能描述：

获取外部图书数据引用，然后按特定筛选条件对图书进行筛选，支持链式调用，例如：myBookFilter.byYear().byTitle().byCategory().byPrice().getBooks()。一般用工厂模式声明该类，其链式调用特性使不同条件的组合变得更加清晰易懂，最终会返回原始数据的切片（浅拷贝）。

方法列表：

- byYear(start, end)：按始末年份筛选图书。

- byPrice(min, max)：按图书价格筛选图书。

- byCategory(categories[])：按图书分类过滤图书，支持一次传入多个分类析取查找。

- byTitle(title)：按图书标题过滤图书。

```
class BookFilter {
  // filter 会返回新切片，而不是修改原数据
  private books: Book[];

  constructor(books: Book[]) {
    this.books = books;
  }

  byYear(startYear: number, endYear: number): BookFilter {
    this.books = this.books.filter(book =>
      book.year >= startYear && book.year <= endYear);
    return this;
  }

  byTitle(title: string): BookFilter {
    this.books = this.books.filter(book =>
      book.title.toLowerCase().includes(title.toLowerCase())
    );
    return this;
  }

  byCategory(categories: string[]|null): BookFilter {
```

```
    // 如果传入 null，不进行过滤
    if (categories === null) {
      return this;
    }
    this.books = this.books.filter(book =>
      categories.some(category =>
        book.category.toLowerCase().includes(
          category.toLowerCase()
        )
      )
    );
    return this;
  }

  byPrice(minPrice: number, maxPrice: number): BookFilter {
    this.books = this.books.filter(book => {
      return book.price >= minPrice && book.price <= maxPrice;
    });
    return this;
  }

  getBooks(): Book[] {
    return this.books;
  }
}
```

图书排序服务：

功能描述：

获取外部图书数据引用，然后按特定条件对图书进行排序，支持链式调用，例如：myBookSorter.byYear().byTitle().bySales()。一般用工厂模式声明该类，其链式调用特性使不同筛选条件的组合变得更加清晰易懂，多条件排序时结果更加稳定，该类直接排序传入的列表引用。

方法列表：

- byYear()：按出版年份排序图书。
- byPrice()：按图书价格排序图书。
- bySales()：按图书销量（热度）排序图书。
- byRandom()：随机排序图书（洗牌算法）

```
class BookSorter {
  // 书籍排序器，支持链式调用
  private books: Book[];
```

```
4.
5.   constructor(books: Book[]) {
6.     // 直接修改外部引用
7.     this.books = books;
8.   }
9.
10.  byPrice(ascending: boolean = true) {
11.    this.books.sort((a, b) =>
12.      ascending ? a.price - b.price : b.price - a.price
13.    );
14.    return this;
15.  }
16.
17.  bySales(ascending: boolean = true) {
18.    this.books.sort((a, b) =>
19.      ascending ? a.sales - b.sales : b.sales - a.sales);
20.    return this;
21.  }
22.
23.  byYear(ascending: boolean = true) {
24.    this.books.sort((a, b) =>
25.      ascending ? a.year - b.year : b.year - a.year);
26.    return this;
27.  }
28.
29.    byRandom() {
30.    this.books.sort(() => Math.random() - 0.5);
31.    return this;
32.  }
33. }
```

“书籍浏览”界面逻辑：

功能描述：

界面“书籍浏览”的主要逻辑如下，将筛选和排序按钮映射为具体筛选值（BookFilter 与 BookSorter 中接口），然后在回调函数（reFilter，reSort）中对界面展示的书籍进行重新排序。

函数列表及描述：

- reFilter：根据用户选择的分类、年限、价格重新筛选展示图书。
- reSort：按随机、价格、销量、年份重新排序展示图书。、
- download：下载按当前排序与筛选后的图书信息的 csv 统计文件，该函数同

时对文件的用户名进行格式化。

```
const chosenType = ref('全部');
const types = [
    '全部', '小说', '艺术', '历史', '生物', '旅游', '人文',
]
const typeMap: { [key: string]: string[]|null } = {
    '全部': null,
    '小说': ['fiction', 'novel'],
    '艺术': ['art', 'music'],
    '历史': ['his'],
    '生物': ['biolo'],
    '旅游': ['travel'],
    '人文': ['human']
};

// price filter
const chosenPrice = ref('全部');
const prices = [
    '全部', '小于 30', '30-50', '50-100', '100-200', '200 以上',
]
const priceMap: { [key: string]: number[] } = {
    '全部': [0, 10000],
    '小于 30': [0, 30],
    '30-50': [30, 50],
    '50-100': [50,100],
    '100-200': [100, 200],
    '200 以上': [200, 10000],
};

// year filter
const chosenYear = ref('全部')
const years = [
    '全部', '<1980', '1980s', '1990s', '2000s', '2010s', '2020s',
]
const yearMap: { [key: string]: number[] } = {
    '全部': [0, 3000],
    '<1980': [0, 1980],
    '1980s': [1980, 1990],
    '1990s': [1990, 2000],
    '2000s': [2000, 2010],
    '2010s': [2010, 2020],
    '2020s': [2020, 2030],
};

```

```
const reFilter = () => {
    // 结合三种筛选
    const [startYear, endYear] = yearMap[chosenYear.value];
    const [minPrice, maxPrice] = priceMap[chosenPrice.value];
    const types = typeMap[chosenType.value]
    books.value = oracle.bFilter()
        .byCategory(types)
        .byPrice(minPrice, maxPrice)
        .byYear(startYear, endYear)
        .getBooks();
}

// button
const chosenSortOrder = ref<string>('');
const sortOrders = [
    { label: '随机排序' },
    { label: '按热度排序' },
    { label: '按价格排序' },
    { label: '按时间排序' },
];

function reSort(label: string) {
    chosenSortOrder.value = label;
    if (label == '随机排序') {
        oracle.bSort().byRandom();
    } else if (label == '按价格排序') {
        oracle.bSort().byPrice();

    } else if (label == '按时间排序') {
        oracle.bSort().byYear();
    } else { // 按热度
        oracle.bSort().bySales();
    }
    // 对原始数据重新排序后，重新过滤一次，得到过滤数据
    reFilter();
}

// download csv
function download() {
    // 格式化文件名
    const now = new Date();
    const timestamp = `${now.getFullYear()}-${now.getMonth() + 1}-${now.getDate()}`;

    const bookPrice = "价格-"+chosenPrice.value.replace(/\s+/g, '-').toLowerCase();
```

```
89.     const bookType = "类型-"+chosenType.value.replace(/\s+/g, '-').toLowerCase();
90.     const bookSortOrder = "排序方式"+chosenSortOrder.value.replace(/\s+/g, '-
').toLowerCase();
91.     const bookYear = "年份-"+chosenYear.value.replace(/\s+/g, '-').toLowerCase();
92.
93.     const filename = [
94.         "筛选条件", bookPrice, bookType, bookYear, bookSortOrder, timestamp
95.     ].filter(Boolean).join('_');
96.     BookService.output2CSV(filename, books.value);
97. }
```

3.2.5 构件 5：书籍详情弹出窗口

3.2.5.1 源程序列表

ForeEnd/src/components/bookPopup.vue

ForeEnd/src/api/models.ts

ForeEnd/src/api/services.ts

3.2.5.2 设计思想

将界面逻辑与服务分离，构建后端数据接口和前端界面逻辑的中间服务层：UserService，用来处理后端用户数据，同时给前端易于使用的集成接口。

书籍弹出窗口组件接受外部组件传入的书籍信息以及当前用户信息，该窗口是对单本书籍实体的一层抽象，用户在此对单本书籍完成操作：添加该书籍入收藏夹；选择购买数量，并添加入购物车。操作由前端用户服务向后端发起具体请求。

3.2.5.3 实现描述

输入参数：

- 单本图书详细信息
- 用户名：用于初始化用户服务。

功能描述：

对用户逻辑的建模，获取用户信息后对当前用户相关数据进行操作。点击“书籍浏览界面”后，弹出书籍详情窗口（构件 12），其主要负责和当前用户进行交互，所以封装一个类 UserService 充当后端与界面逻辑的中间层。

方法列表：

- getUser(username)：根据用户输入的用户名获取用户详细信息。

- addToCart(book_id, amount)：向当前用户购物车中添加书籍。

- addToFavors(book_id)：向当前用户收藏夹中添加书籍。

```
export class UserService {
  private me!: User;
  constructor(username: string) {
    this.getUser(username);
  }

  async getUser(username: string) {
    // 其实应该是静态方法。。
    try {
      const idResponse = await requestData({
        url: 'users/get_id/',
        method: 'get',
        query: { uname: username },
      })
      const id = idResponse?.data.uid;
      const userResponse = await requestData({
        url: `/users/${id}`,
        method: 'get',
        query: { uid: id, }
      })
      this.me = userResponse?.data;
    } catch (error) {
      throw new Error('get user failed')
    }
  }

  async addToCart(bookID: number, amount: number = 1) {
    try {
      await requestData({
        url: 'shoppingcarts/',
        method: 'post',
        query: {
          uid: this.me.id,
          book_id: bookID,
          amount: amount,
        }
      });
    } catch (error) {
      throw new Error('update shopping cart failed')
    }
```

```
41.   }
42.
43.   async addtoFavors(bookID: number) {
44.     const myQuery = {
45.       uid: this.me.id,
46.       book_id: bookID.toString(),
47.     }
48.     requestData({
49.       url: 'collection/custom_filter/',
50.       method: 'get',
51.       query: myQuery,
52.     })!.then(res => {
53.       if (res.data.length > 0) {
54.         throw new Error('book already in collection');
55.       } else {
56.         try {
57.           requestData({
58.             url: 'collection/',
59.             method: 'post',
60.             query: myQuery,
61.           })
62.         } catch (error) {
63.           throw new Error('add to collection failed');
64.         }
65.       }
66.     })
67.   }
68. }
```

3.2.6 构件 6：获取收藏夹

该构件负责根据用户 ID 向后端获取对应的收藏夹信息，并存入变量中用于显示或函数使用

3.2.6.1 源程序列表

ForeEnd/src/views/usercollections.vue

3.2.6.2 设计思想

当用户访问收藏夹页面时，前端在 onMounted 中向后端发送 shoppingcarts/custom_filter/请求，获取当前用户收藏夹中的具体书籍信息，并将之存入变量中，用户 html 显示和事件处理

3.2.6.3 实现描述

前端主要函数：

```
onMounted(() => {
  let uid = localStorage.getItem('ms_uid')
  let collectionRequest = {
    url: 'collection/custom_filter/',
    method: 'get',
    query: {
      'user_id': uid
    },
  }
  requestData(collectionRequest)!.then(res => {
    for (let item of res.data) {
      collectionList.push({
        collection_id: item['id'],
        book_id: item['book']['book_id'],
        bname: item['book']['bname'],
        author: item['book']['author']['aname'],
        category: item['book']['category']['category_name'],
        price: item['book']['price'],
        pub_year: item['book']['pub_year'],
        pname: item['book']['publisher']['pname'],
        rate: item['book']['rate'],
        ImgAddress: item['book']['url'],
      })
    }
  });
})
```

3.2.7 构件 7：删除收藏夹中的内容

该构件负责删除当前用户在数据库收藏夹表中指定的书籍信息，取消收藏。

3.2.7.1 源程序列表

ForeEnd/src/views/usercollections.vue

3.2.7.2 设计思想

在本构建中，前端页面中存在删除按钮，当用户点击该按钮后，触发 DeleteBook 事件，前端根据获取到的信息向后端发送删除请求，删除指定的条目。

3.2.7.3 实现描述

前端主要函数：

```
const DeleteBook = (index: number) => {
  let deleteRequest = {
    url: 'collection/' + collectionList[index].collection_id + '/',
    method: 'delete',
    query: {},
  }
  requestData(deleteRequest)!.then(res => {})
  collectionList.splice(index, 1);
  tableKey++;
}
```

3.2.8 构件 8-9：个人中心

由于构件 8 和构件 9 源代码重合度较高，故这里放在一起说明

3.2.8.1 源程序列表

ForeEnd/src/views/UserInfo.vue

3.2.8.2 设计思想

在本构件中，前端页面在 onMounted 中向后端分别根据用户 ID（缓存在前端）发送获取用户信息、收藏夹信息、访问历史的请求，随后将之用变量保存，用于显示查看。

3.2.8.3 实现描述

前端主要函数：

```
onMounted(() => {
  let uid = localStorage.getItem('ms_uid')
  let UserReadRequest = {
    url: 'users/' + uid + '/',
    method: 'get',
    query: {},
  }
  requestData(UserReadRequest)!.then(res=>{
    userdata.phoneNumber = res.data['tel']
    userdata.email = res.data['email']
    if (res.data['address'] != undefined){
       userdata.address = res.data['address']
```

```
    }

  })
  let collectionRequest = {
    url: 'collection/custom_filter/',
    method: 'get',
    query: {
      'user_id': uid
    },
  }
  requestData(collectionRequest)!.then(res => {
    for (let item of res.data) {
      collectionList.push({
        book_id: item['book']['book_id'],
        bname: item['book']['bname'],
        author: item['book']['author']['aname'],
        category: item['book']['category']['category_name'],
        price: item['book']['price'],
        pub_year: item['book']['pub_year'],
        pname: item['book']['publisher']['pname'],
        rate: item['book']['rate'],
        ImgAddress: item['book']['url'],
      })
    }
  });
  let ShoppingHistoryRequest = {
    url: 'shoppinghistory/custom_filter/',
    method: 'get',
    query: {
      'user_id': uid
    },
  }
  requestData(ShoppingHistoryRequest)!.then(res => {
    for (let item of res.data) {
      shoppingHistoryList.push({
        book_id: item['book']['book_id'],
        bname: item['book']['bname'],
        author: item['book']['author']['aname'],
        category: item['book']['category']['category_name'],
        price: item['book']['price'],
        pub_year: item['book']['pub_year'],
        pname: item['book']['publisher']['pname'],
        rate: item['book']['rate'],
        ImgAddress: item['book']['url'],
```

```
        buy_date: item['date'],
        amount: item['amount']
      })
    }
  });
})
```

3.2.9 构件 10：获取购物车信息

3.2.9.1 源程序列表

ForeEnd/src/views/Basket.vue

3.2.9.2 设计思想

当用户访问购物车时，前端在 onMounted 中根据用户 ID 向后端请求购物车信息，收到数据后用变量保存。

3.2.9.3 实现描述

前端主要函数：

```
onMounted(() => {
  let uid = localStorage.getItem('ms_uid')
  let basketRequest = {
    url: 'shoppingcarts/custom_filter/',
    method: 'get',
    query: {
      'user_id': uid
    },
  }
  requestData(basketRequest)!.then(res => {
    for (let item of res.data) {
      tableData.push({
        basket_id: item['id'],
        book_id: item['book']['book_id'],
        pub_id: item['book']['publisher']['pub_id'],
        bname: item['book']['bname'],
        author_id: item['book']['author']['author_id'],
        author: item['book']['author']['aname'],
        category: item['book']['category']['category_name'],
        category_id: item['book']['category']['category_id'],
        isbn: item['book']['sales'],
        price: item['book']['price'],
        pub_year: item['book']['pub_year'],
```

```
        pname: item['book']['publisher']['pname'],
        rate: item['book']['rate'],
        ImgAddress: item['book']['url'],
        sales: item['book']['sales'],
        amount: item['amount'],
      })
    }
  });
})
```

3.2.10 构件 11：添加购物车

3.2.10.1 源程序列表

ForeEnd/src/views/Basket.vue

3.2.10.2 设计思想

当用户在书籍显示页面点击加入购物车按钮后，触发 AddToBasket，该事件中先根据用户 ID 和书籍 ID 向后端查询是否已有相关记录，若有则接着向后端发送修改请求，增加条目的购买数量。反之,在购物车表创建新的条目，记录此次信息。

3.2.10.3 实现描述

前端主要函数：

```
const AddToBasket = () => {
  const point: RegExp = /\./;
  if (point.test(ShowingBook.buyNumber.toString())) {
    showWarning.value = true
    return
  } else {
    if (ShowingBook.buyNumber < 1) {
      showWarning.value = true
      return;
    } else {
      showWarning.value = false
    }
  }
  let uid = localStorage.getItem('ms_uid');
  let filterToBasketRequest = {
    url: 'shoppingcarts/custom_filter/',
    method: 'get',
```

```
    query: {
      'user_id': uid,
      'book_id': ShowingBook.id,
    },
  }
  requestData(filterToBasketRequest)!.then(res => {
    if (res.data.length > 0) {
      let basket_id = res.data[0]['id']
      let old_amount = res.data[0]['amount']
      let new_amount = (Number(old_amount) + ShowingBook.buyNumber).toString()
      let updateBasketRequest = {
        url: 'shoppingcarts/' + basket_id + '/',
        method: 'put',
        query: {
          'uid': uid,
          'book_id': ShowingBook.id,
          'amount': new_amount,
        },
      }
      requestData(updateBasketRequest)!.then(res => {
        console.log('add to basket.')
      })
    } else {
        let createBasketRequest = {
        url: 'shoppingcarts/',
        method: 'post',
        query: {
          'uid': uid,
          'book_id': ShowingBook.id,
          'amount': ShowingBook.buyNumber.toString(),
        },
      }
      requestData(createBasketRequest)!.then(res => {
      })
    }
  })
  BookVisible.value = !BookVisible.value;
}
```

3.2.11 构件 12：支付

3.2.11.1 源程序列表

ForeEnd/src/views/Basket.vue

3.2.11.2 设计思想

当用户在购物车页面点击支付按钮，触发 buyBooks，该事件向后端发送书籍信息修改请求，修改所购买书籍的销量，紧接着删除购物车中条目，清空购物车。与此同时，数据库中的触发器会在访问历史表中记录此次操作。

3.2.11.3 实现描述

前端主要函数：

```
const buyBooks = ()=>{
  let uid = localStorage.getItem('ms_uid')
  for (let i=0;i<tableData.length;i++){
    let updateBookRequest ={
      url: 'books/' + tableData[i].book_id +'/',
    method: 'put',
    query: {
        book_id: tableData[i].book_id,
      author_id: tableData[i].author_id,
      pub_id: tableData[i].pub_id,
      category_id: tableData[i].category_id,
      bname: tableData[i].bname,
      price: tableData[i].price,
      pub_year: tableData[i].pub_year,
      url: tableData[i].ImgAddress,
      isbn: tableData[i].isbn,
      sales: (Number(tableData[i].sales)+ Number(tableData[i].amount)).toString(),
      rate:tableData[i].rate,
    },
    }
    requestData(updateBookRequest)!.then(() => {})
    let deleteRequest = {
    url: 'shoppingcarts/' + tableData[i].basket_id + '/',
    method: 'delete',
    query: {},
    }
    requestData(deleteRequest)!.then(() => {})
  }
  tableData.splice(0, tableData.length)
  tableKey++;
}
```