第一章　需求建模

1.1　系统功能性需求

本网站目标是为社区用户提供一个十五分钟生活圈临期食品智慧社区平台，通过建立网站，让用户既能作为卖发布临期食品信息，又可以作为买家及时获取优惠食品资源，购买商品，从而促进社区食品资源的高效流通，实现节约资源、便利生活的目标。

本网站用户的基本功能、以及买家和卖家的主要功能有：

基本功能：注册、登录、修改个人信息，充值，查看订单，评价订单。

作为买家：查看、购买商品，查看卖家信息，变更物流状态（取消订单，确认收货）。

作为卖家：上传商品信息，变更物流状态（取消订单，完成发货）。

基本功能、作为卖家、作为卖家的组织架构图如图 3-1 所示：

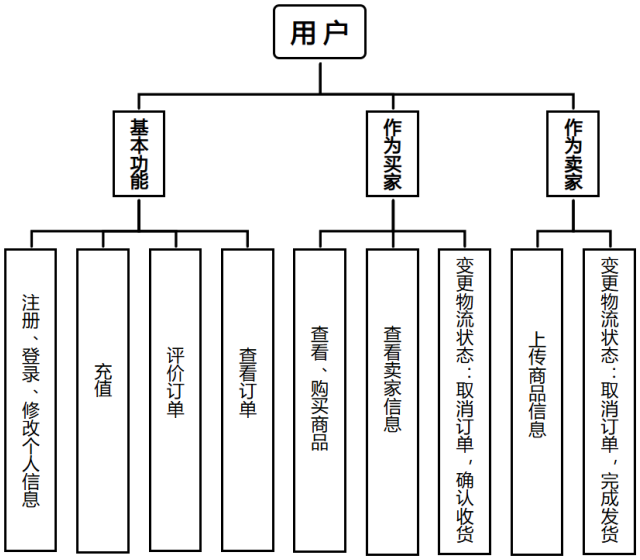


图 3-1：用户组织架构图

1.2　系统用例图（todo)

系统有三种用户：管理员、教师、学生，他们的功能性需求各不相同，根据系统功能性需求可以绘制出系统用例图，如图 3-2 所示：

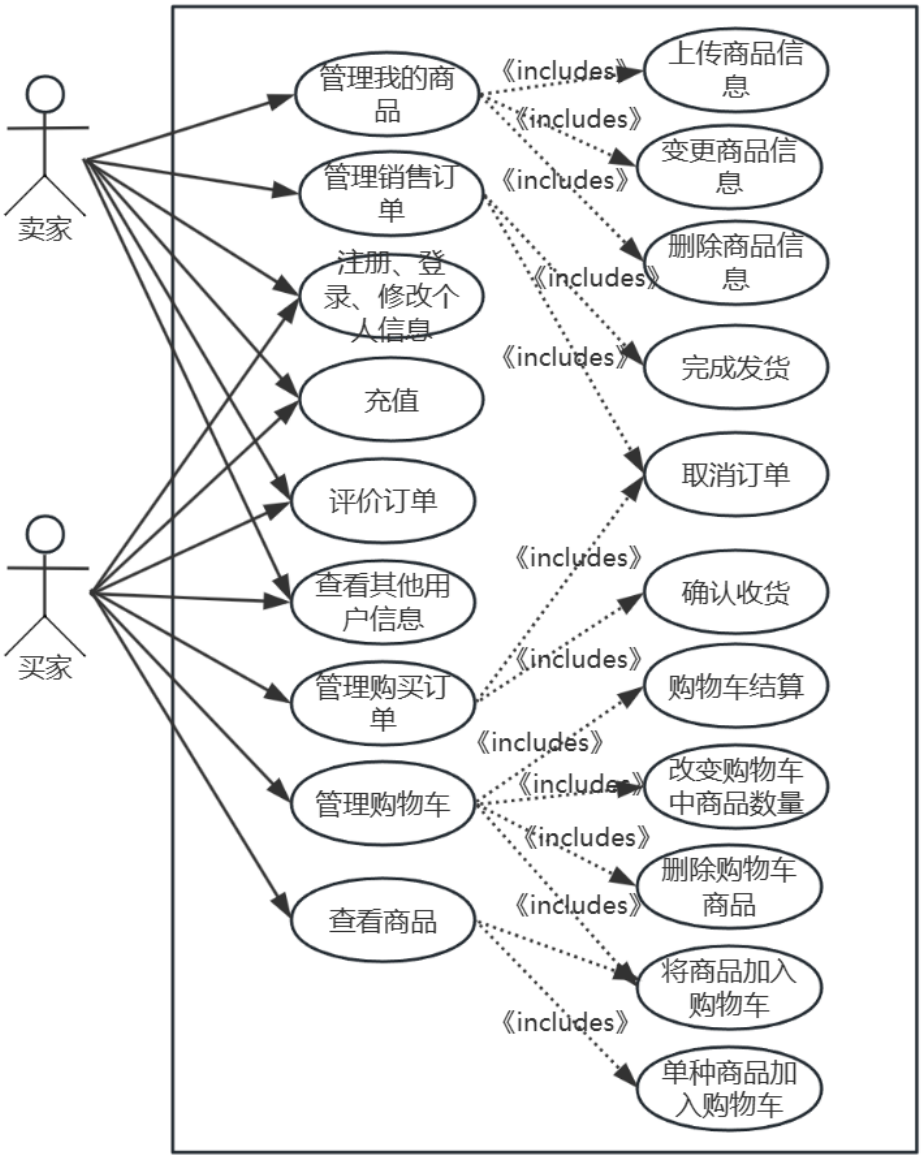


图 3-2：系统用例图

1.3　系统核心用例

本系统涉及到的用例比较多，本节主要介绍以下三个核心用例，它们分别是：发布商品、创建订单（根据购物车批量创建与单种商品创建），商品加入购物车

1.3.1 发布商品(todo)

用例：发布商品

范围：TODO 网站

级别：用户目标

主要参与者：买家

涉众及其关注点：

1. 卖家： 希望商品被准确记录并扣减库存，订单状态能及时反映交易进度。
2. 买家： 希望订单创建过程快速、准确，扣款和商品信息无误。**成功保证**：

主成功场景：

1. 用户已登录系统并具备正常的账户状态（余额充足、未被冻结等）。
2. 商品或购物车中的商品库存足够支持订单需求。

扩展：

1a. 卖家所输入的帐号或密码不正确：

1. 系统提示用户帐号或密码错误，要求用户重新输入。

4a. 卖家填写的临期日期格式错误：

1. 系统提示填写的日期格式错误，要求用户输入 YYYY-MM-DD 格式的日期。

4b. 卖家上传文件不是系统所规定的格式：

1. 系统提示文件格式错误，要求用户上传正确的文件格式。

5a. 卖家未填写必填商品信息：

1. 系统提示添加失败，说明失败原因是学生不在系统中，请用户联系管理员。

5b. 要添加的学生已经存在课程名单中：

1. 系统提示添加失败，说明失败原因是学生已经存在课程学生名单中。

特殊需求：

1. 系统确保上传过程的稳定性，支持大多数主流图片格式（如 JPEG、PNG）。
2. 商品信息的实时性和一致性保障，确保数据同步更新到买家可见的商品列表中。

发生频率：经常发生，卖家需要频繁更新或上传商品。

1.3.2 上传作业(todo)

用例：上传作业

范围：TODO 网站

级别：用户目标

主要参与者：买家

涉众及其关注点：

1. 卖家： 希望商品被准确记录并扣减库存，订单状态能及时反映交易进度。
2. 买家： 希望订单创建过程快速、准确，扣款和商品信息无误。

前置条件：

1. 用户已登录系统并具备正常的账户状态（余额充足、未被冻结等）。
2. 商品或购物车中的商品库存足够支持订单需求。

成功保证：

1. 单一商品直接购买时，订单和交易数据被正确创建，库存实时更新，余额准确扣减。
2. 批量结算购物车商品时，所有商品订单成功创建，库存逐项扣减，余额总价准确扣除。
3. Idempotency Key 确保请求的幂等性，避免重复订单创建。

主成功场景：

场景一：直接购买单一商品

1. 买家通过商品详情页面选择购买商品。
2. 系统校验 Idempotency Key，验证是否重复请求。
3. 系统校验商品库存是否足够满足购买需求。
4. 系统校验买家余额是否足够支付商品总价。
5. 系统扣减商品库存、买家余额，并创建订单记录。
6. 系统生成订单并返回成功信息，包括订单详情（商品名称、数量、总价等）。

场景二：购物车批量结算

1. 买家从购物车中选择结算的商品列表。
2. 系统校验 Idempotency Key，验证是否重复请求。

3. 系统校验每个商品的库存是否足够满足需求：
4. 若某商品库存不足，终止结算并返回失败提示。
5. 系统校验买家余额是否足够支付所有商品总价：
6. 若余额不足，终止结算并返回失败提示。
7. 系统逐项扣减商品库存，并为每个商品生成订单记录。
8. 系统总计支付金额，扣减买家余额，删除已结算购物车商品项。
9. 系统返回成功信息，包括订单总价和商品详情。

扩展

1. 直接购买单一商品：

1a. 买家未提供 Idempotency Key：系统提示缺少幂等性标识，要求重新提交请求。

3a. 商品库存不足：系统返回失败信息，提示买家选择其他商品或减少购买数量。

4a. 买家余额不足：系统返回失败信息，提示余额不足以支付总价。

2. 购物车批量结算：

3a. 某商品库存不足：

系统终止结算并标明具体商品库存不足信息。

4a. 买家余额不足：

系统终止结算并返回余额不足提示，要求买家调整购物车内容。

特殊需求：

1. 系统支持高并发场景下的幂等性操作，避免重复创建订单或扣款。
2. 对批量订单操作的事务处理能力强，确保所有商品操作的原子性：若任意商品库存不足或余额扣款失败，整体回滚操作。

发生频率：经常发生，用户需要日常交易或批量结算购物车商品。。

1.3.3 下载学生作业(todo)

用例：加入购物车

范围：TODO 网站

级别：用户目标

主要参与者：买家

涉众及其关注点：

买家： 希望能够快速将商品添加到购物车，并准确显示商品信息及数量。

前置条件：

1. 用户已登录系统并具备正常的账户状态。
2. 商品存在并可被正常访问。
3. 商品库存充足，能够满足购物车的添加需求。

成功保证：

1. 商品被成功添加到买家的购物车，购物车界面显示正确的商品信息（名称、价格、数量等）。
2. 系统校验库存，确保不会超出可用库存数量。
3. 系统避免重复添加同一商品而累积多个记录（应合并为一条记录并更新数量）。

主成功场景：

1. 买家通过商品详情页面点击 “加入购物车” 按钮。
2. 系统校验商品是否存在并检查库存数量。
3. 如果商品已存在于购物车中，系统更新该商品的数量。
4. 如果商品不存在于购物车中，系统创建新的购物车项记录。
5. 系统返回成功提示，购物车界面显示最新商品信息和数量。

扩展：

场景：库存不足

3a. 用户添加的商品数量超出当前库存：系统返回失败提示，并显示当前可用库存数量，提示用户减少选择的数量。

场景：商品已下架或被删除

2a. 系统发现商品已下架或不存在：系统返回失败提示，并从商品列表中移除该商品（如适用）。

场景：未登录用户操作

1a. 用户未登录直接尝试加入购物车：系统提示用户登录后才能将商品加入购物车，并跳转至登录界面。

特殊需求：

1. 系统支持高并发情况下购物车操作的性能，确保快速响应。
2. 同一商品在购物车中不能重复添加，而是合并数量。
3. 针对未登录用户，支持临时购物车数据的缓存，当用户登录后，自动同步数据至账户的购物车中。

发生频率：经常发生，买家在浏览商品时会频繁使用加入购物车功能。

1.4 领域模型

本系统共有 8 个模型：用户、产品、购物车、订单、账单、评论、幂等密钥和图片。它们的关系如下：

1. 用户（User）

用户可以出售多个产品，用户与产品的关系是 1:0..n。

用户可以购买多个订单，用户与订单的关系是 1:0..n。

用户可以作为卖家接受多个订单，用户与订单的关系是 1:0..n。

用户可以添加多个产品到购物车，用户与购物车的关系是 1:0..n。

用户可以写多个评论，用户与评论的关系是 1:0..n（评论者）。

用户可以收到多个评论，用户与评论的关系是 1:0..n（被评论者）。

用户可以有多个幂等密钥，用于幂等操作，用户与幂等密钥的关系是 1:0..n。

2. 产品（Product）

产品属于一个卖家，产品与用户的关系是 n:1。

产品可以出现在多个订单中，产品与订单的关系是 1:0..n。

产品可以被添加到多个购物车，产品与购物车的关系是 1:0..n。

3. 购物车（Cart）

一个购物车条目对应一个用户和一个产品。

用户与购物车的关系是 1:0..n。

产品与购物车的关系是 1:0..n。

4. 订单（Order）

一个订单由买家（用户）发起，订单与买家的关系是 1:1。

一个订单由卖家（用户）处理，订单与卖家的关系是 1:1。

一个订单包含一个产品，订单与产品的关系是 1:1。

5. **账单（Bill）**

一个账单由用户发起，账单与用户的关系是 1:1。

一个账单可能与一个订单相关，账单与订单的关系是 0..1:1。

6. 评论（Review）

一个评论有评论者和被评论者（用户），评论与用户的关系是 1:1（双向）。

一个评论对应一个订单，评论与订单的关系是 1:1。

7. 幂等密钥（IdempotencyKey）

一个幂等密钥属于一个用户，幂等密钥与用户的关系是 1:1。

8. 图片（Image）

一个图片属于一个商品，图片与商品的关系时 1:1

据此，可以画出系统的领域模型，如图 3-3 所示：

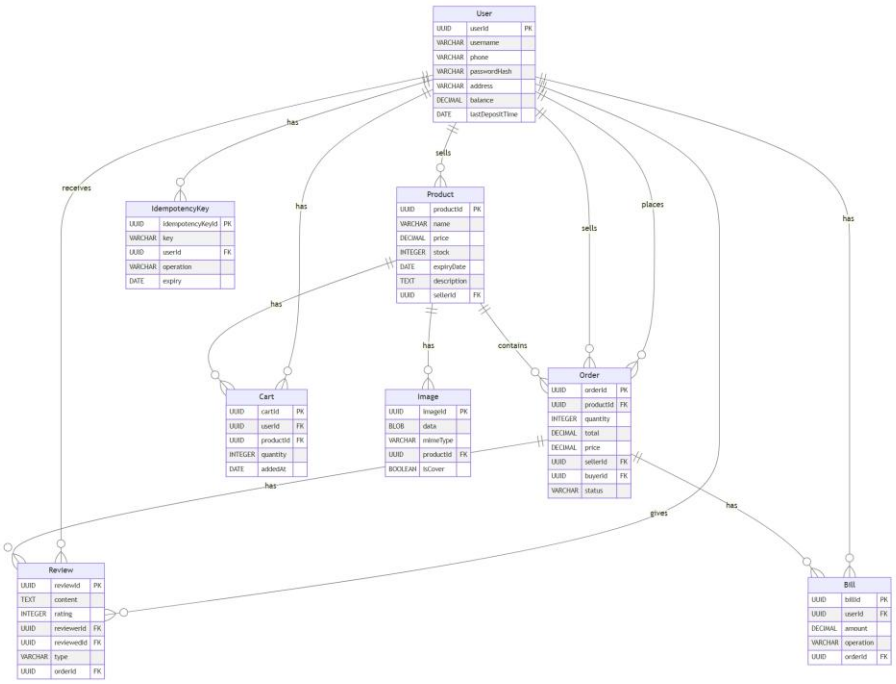


图 3-3：系统领域模型图

第二章　架构设计

2.1　系统架构及原理

本系统采用 MVC（Model-View-Controller）的设计模式，为更清晰地体现系统的职责分工，我们将后端 Model 层细分为 **Domain 层** 和 **Foundation 层**，使系统具备良好的扩展性与模块化特性。

1. 视图层（UI 层）

位置：架构的最顶层。

功能：负责与用户交互，展示界面和处理用户操作。

技术：我们使用 React 框架和 Material-UI 构建动态界面，结合 Bootstrap 实现跨平台自适应设计，保证界面友好、操作便捷。React 同时通过 Axios 调用后端提供的 RESTful API，与后端数据进行交互。

2. 控制层（Controller 层）

位置：架构的第二层。

功能：连接 UI 层和 Domain 层，处理来自前端的请求并调用相应的业务逻辑，返回结果给视图层。

技术：使用 Express.js 框架搭建路由和控制器。控制器主要负责请求的处理，包括参数校验、调用 Domain 层的逻辑、以及返回 HTTP 响应。

3. 领域层（Domain 层）

位置：架构的第三层。

功能：实现系统的核心业务逻辑，处理与业务模型相关的复杂计算和数据操作。

技术：该层使用 Sequelize 定义系统的核心数据模型（如用户、订单、产品等）。每个模型负责实现与该业务对象相关的操作，如创建、更新、查询或删除。通过模块化的方式对模型类和业务逻辑进行分组管理。

4. 基础层（Foundation 层）

位置：架构的最底层。

功能：提供系统运行所需的基础服务，如数据库连接与框架支持。

技术：使用 Sequelize ORM 实现对 MySQL/PostgreSQL 等关系型数据库的支持，提供简洁的接口实现数据表与领域层模型的映射。使用 Express.js 中间件处理日志、错误管理、鉴权等通用功能。配置数据库连接池、环境变量管理等基础服务。

我们的系统架构具有如下特点

- **前后端分离**：前端使用 React 负责界面展示与用户交互，后端通过 RESTful API 提供数据支持，便于开发和维护。
- **模块化分层设计**：领域层与基础层的分离，使得系统可以灵活更换数据库类型或升级框架而不影响业务逻辑层。
- **扩展性与维护性**：通过 MVC 模式和服务抽象层，降低了各模块间的耦合性，便于系统功能的扩展和后续维护。

这种架构充分利用了 Node.js 异步非阻塞的特点，结合 React 的组件化开发模式和 Sequelize 的 ORM 优势，实现了高效、易扩展的系统设计。

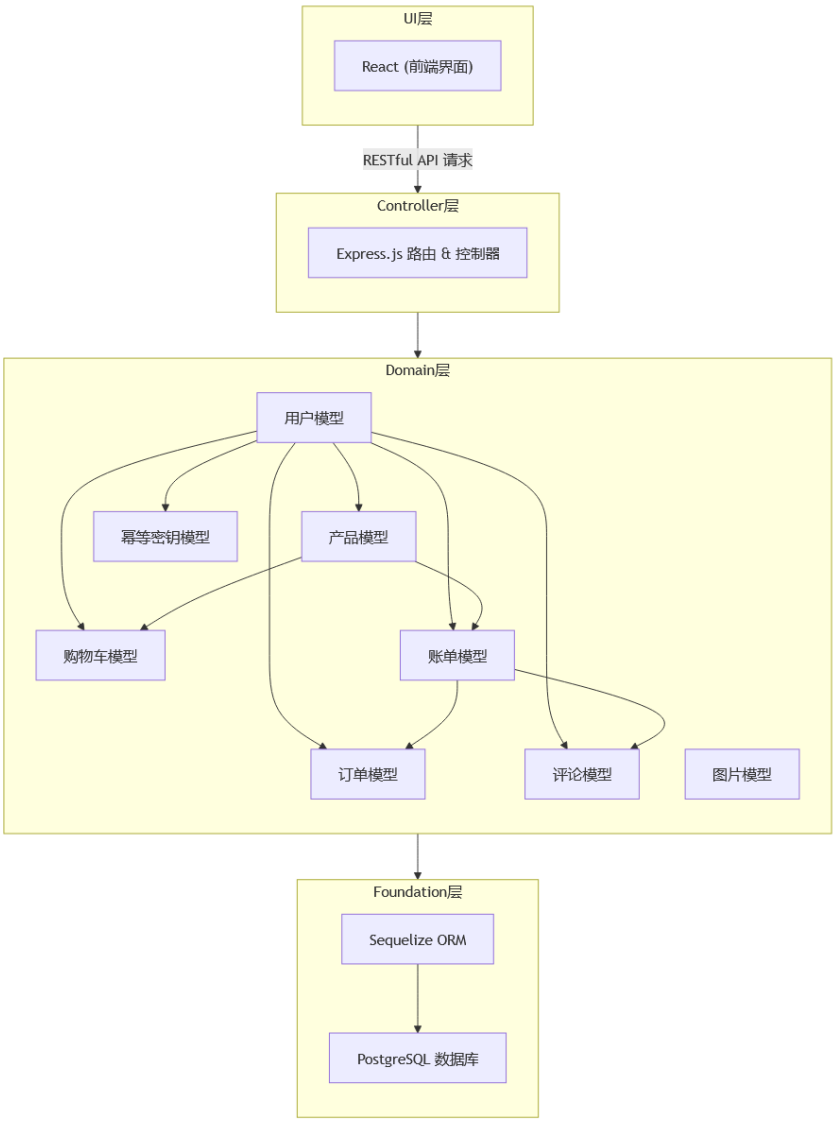


图 4-1：系统包图

2.2　业务用例的实现

本系统涉及到的业务用例比较多，本节选取其中的三个用例进行详细的分析。它们分别是：发布商品，创建订单（根据购物车批量创建与单种商品创建），商品加入购物车。

2.2.1 发布商品

发布商品是卖家的功能，其与系统的交互过程如下：

1. 用户在登录界面输入用户名和密码登录系统。

2. 系统验证用户的用户名和密码。
3. 用户成功登录进入用户的主页。
4. 用户选择发布商品功能。
5. 系统接收请求，返回商品发布表单页面。
6. 用户填写商品信息并提交请求。
7. 系统接收请求，验证商品信息。
8. 系统将商品信息存储到数据库中。
9. 系统返回包含商品信息的页面。

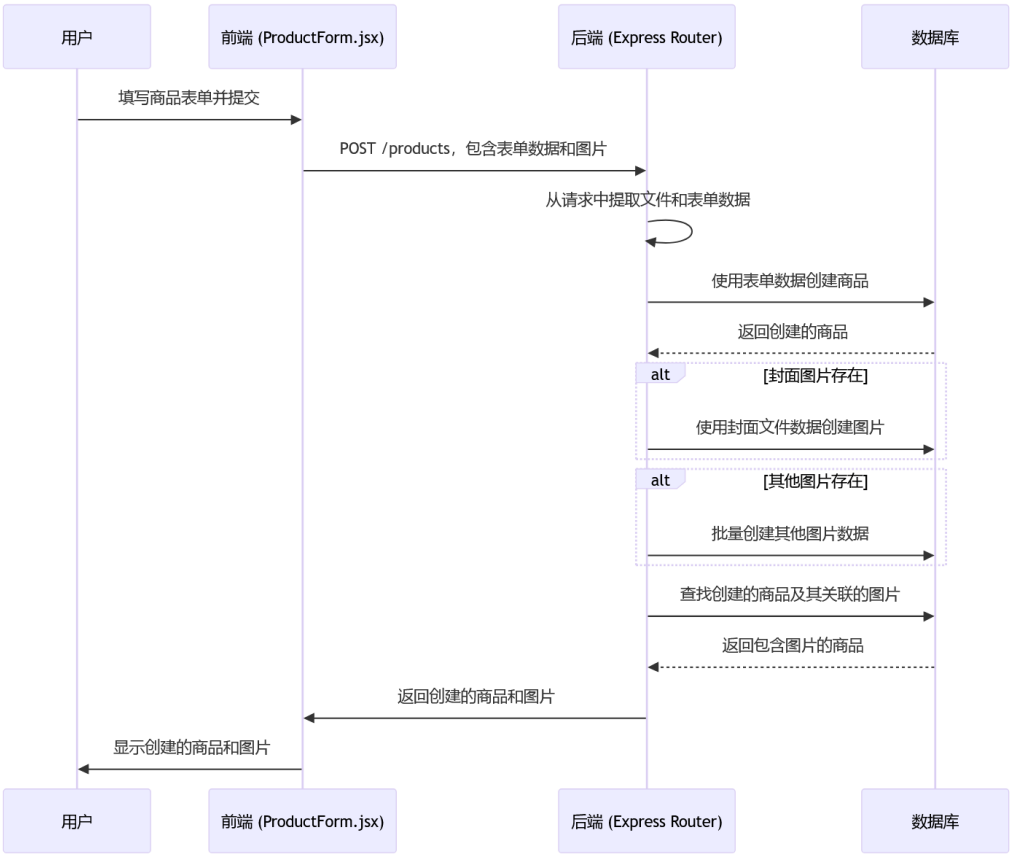


图 4-2：增加课程学生主要顺序图

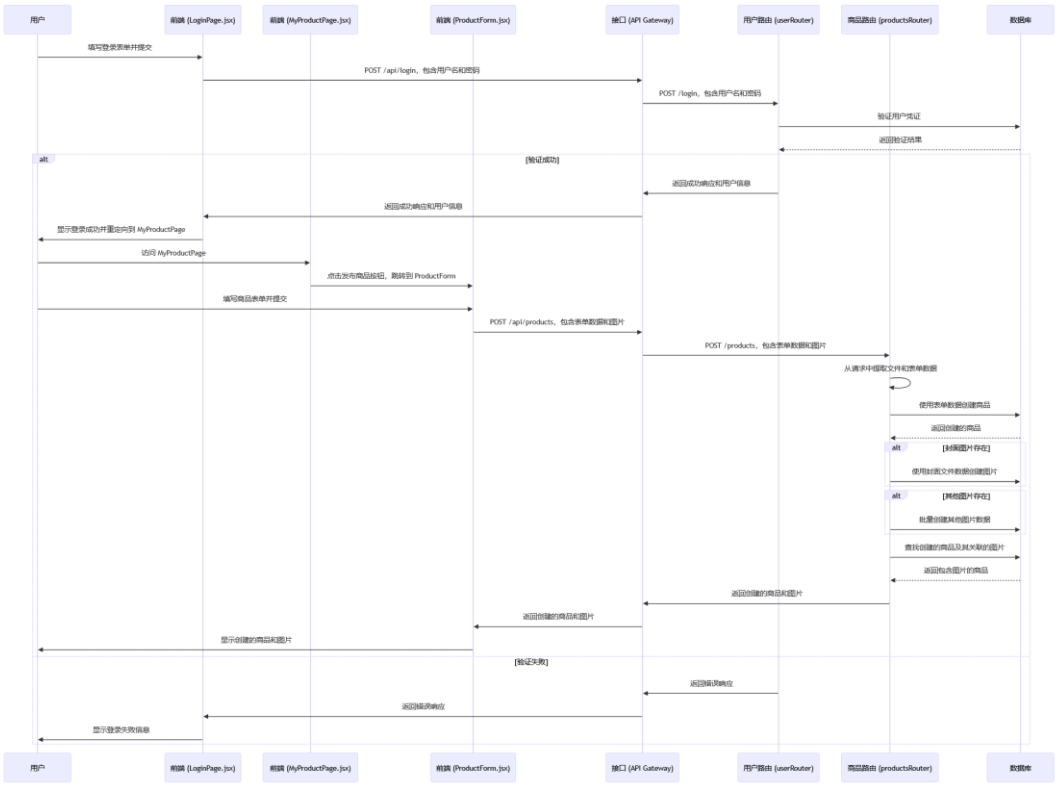


图 4-3：增加课程学生顺序图

2.2.2 创建订单

2.2.2.1 根据单一商品创建订单

根据单一商品创建订单是买家用户的功能，其与系统的交互过程如下：

1. 用户在登录界面输入用户名和密码登录系统。
2. 系统验证用户的用户名和密码。
3. 用户成功登录进入商品页面。
4. 用户在商品页面点击购买按钮。
5. 系统显示结账对话框。
6. 用户确认购买并提交请求。
7. 系统接收请求，验证商品信息和用户余额。
8. 系统将订单信息存储到数据库中。
9. 系统返回订单信息。

根据单种商品创建订单的顺序图如图 4-4 所示：

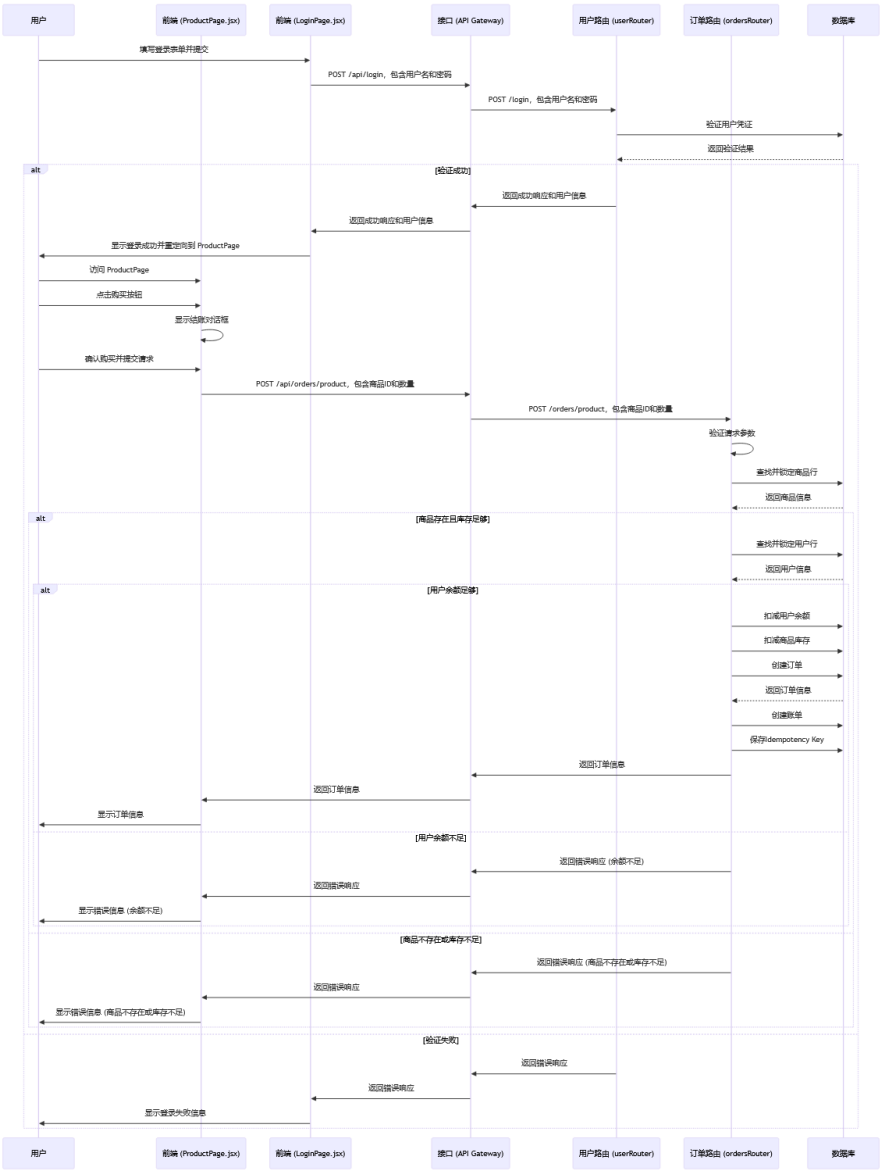


图 4-4：根据单一商品创建订单顺序图

2.2.2.2 根据购物车批量创建订单

根据购物车批量创建订单是买家用户的功能，其与系统的交互过程如下：

1. 用户在登录界面输入用户名和密码登录系统。
2. 系统验证用户的用户名和密码。
3. 用户成功登录进入购物车页面。
4. 用户在购物车页面选择要结账的商品。
5. 用户点击结账按钮。
6. 系统显示结账对话框。
7. 用户确认购买并提交请求。

8. 系统接收请求，验证商品信息和用户余额。
9. 系统将订单信息存储到数据库中。
10. 系统返回订单信息。

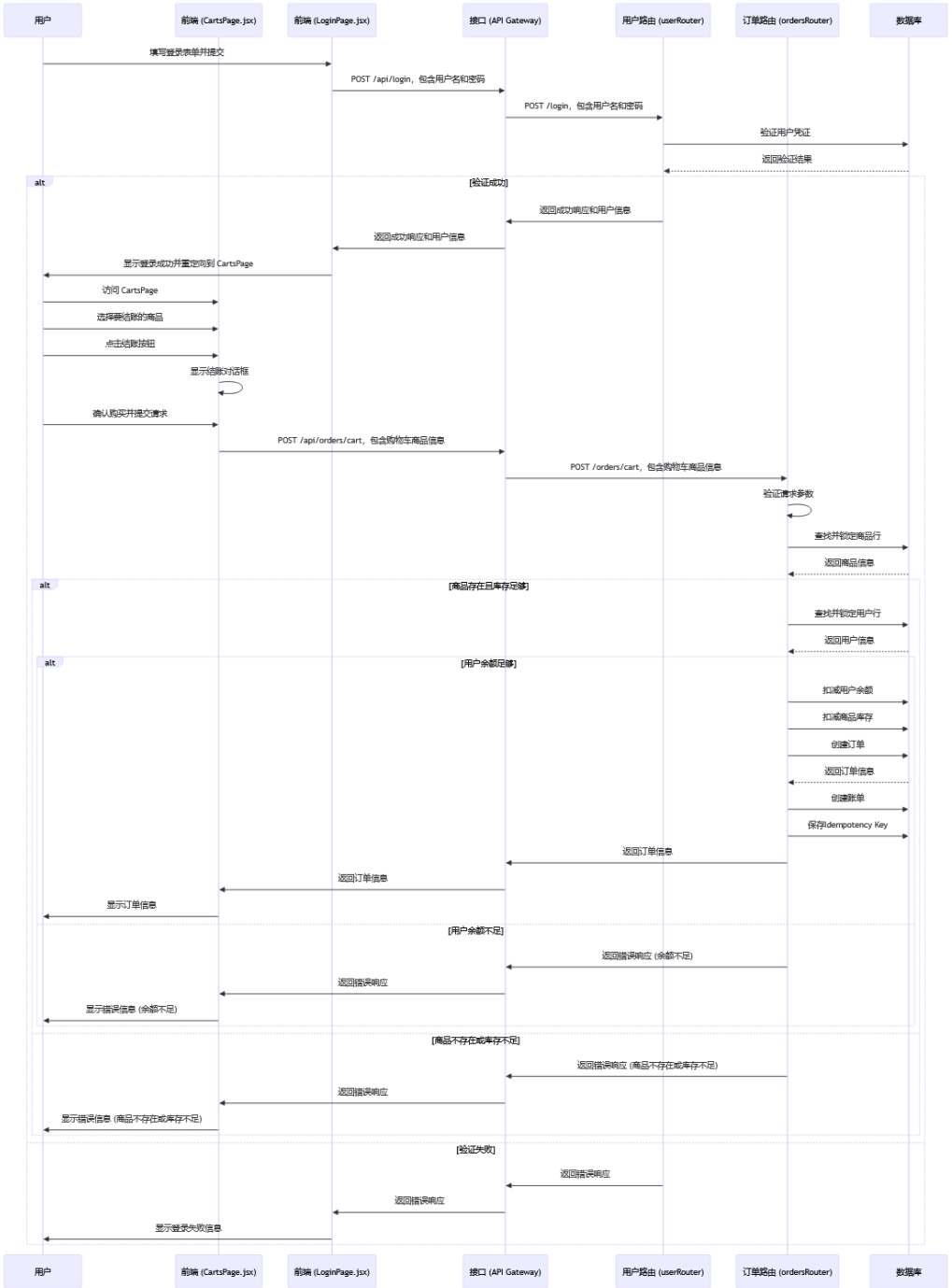


图 4-5：根据购物车批量创建订单序图

2.2.3 将商品加入购物车

将商品加入购物车是教师用户的功能，其与系统的交互过程如下：

1. 用户在登录界面输入用户名和密码登录系统。
2. 系统验证用户的用户名和密码。
3. 用户成功登录进入商品页面。
4. 用户在商品页面选择商品并点击加入购物车按钮。
5. 系统检查购物车中是否已有该商品。
6. 如果购物车中没有该商品，系统将商品添加到购物车。
7. 如果购物车中已有该商品，系统更新购物车中该商品的数量。
8. 系统返回操作结果并显示通知。下载学生作业的顺序图如图 4-6 所示：

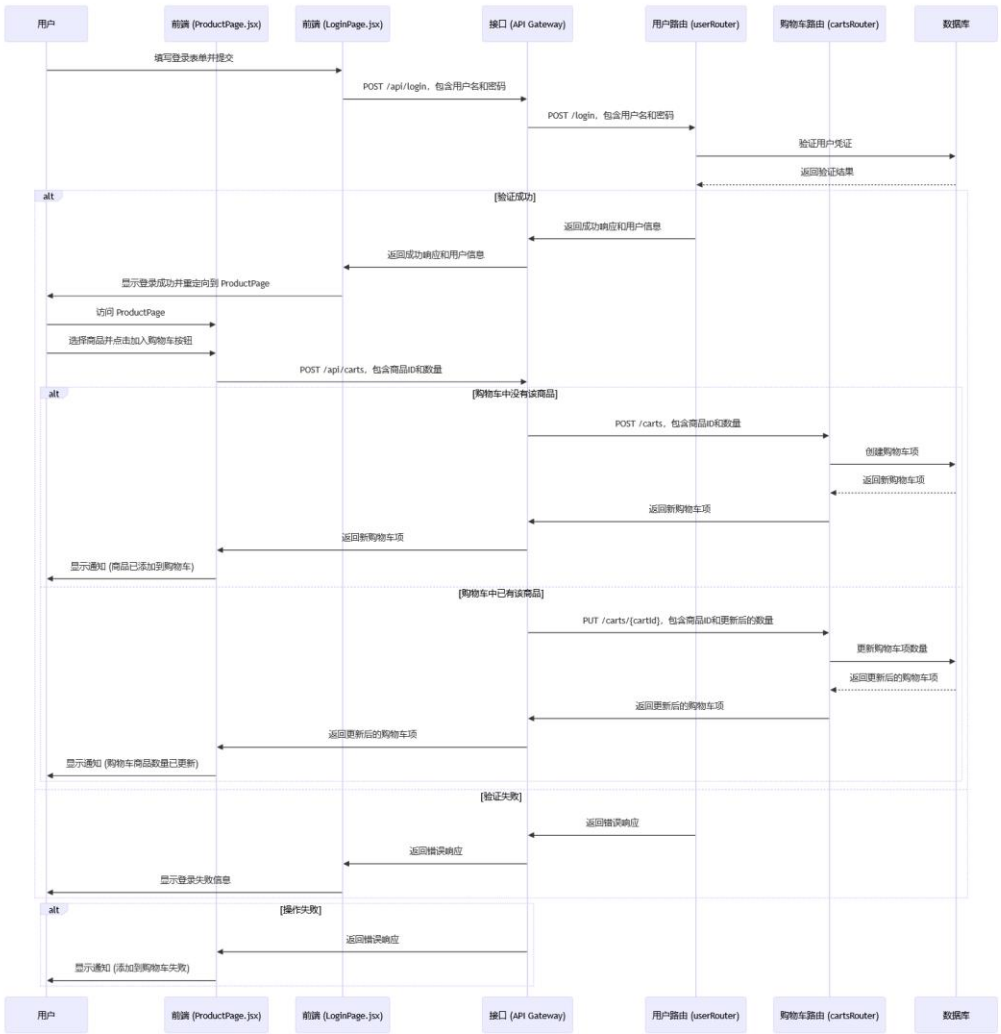


图 4-4：下载学生作业顺序图

2.3 数据库设计

（todo)本系统数据库采用 Python 自带的 SQLite 数据库，这是一个轻量级的高效易用的关系型数据库。

为了方便对系统三种角色的操作，本系统将用户设计成三张表，管理员、教师和学生各成一张表。另外，本系统涉及到文件的存储和下载，为了方便操作和数据库记录，在数据库表中只记录文件存储的地址，文件上传时创建对应的地址并保存在对应的数据表中，文件需要下载时则可以根据表中的地址直接获取对应的文件。

2.3.1 ER 图

本系统的 ER 图如图 4-5 所示：

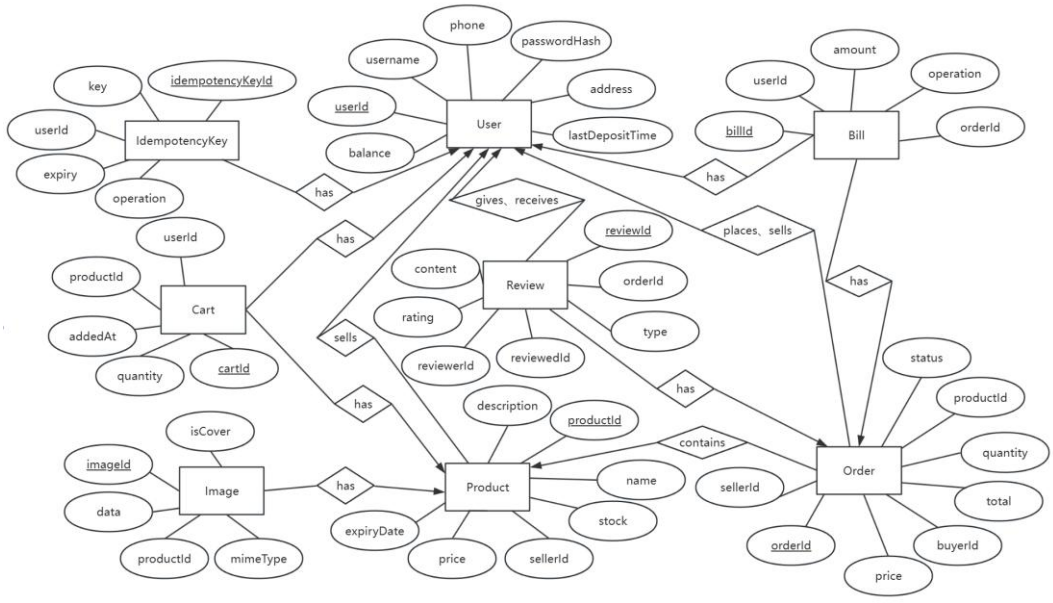


图 4-5：系统 ER 图

2.3.2 数据库表设计

本系统共有 8 张数据库表，分别是：bill、Cart、Idem potency Key、Image、order、Product、review、user。

表 4-1 对数据库表进行了整体的描述。

表 4-1：数据库表汇总

表名	说明
bills	账单表
Cart	购物车表
IdempotencyKey	幂等键表，防止重复操作
Image	存放商品图片的表
order	订单表
Product	商品表
review	评价表
user	用户信息表

表 4-2 描述了 bills 表的详细信息。

表 4-2：账单表

字段	数据类型	主键、外键	允许空	说明
billId	UUID	主键	否	账单编号
userId	UUID	外键（user.userId）	否	用户编号
amount	DECIMAL (10, 2)		否	账单金额，最多 10 位数字，其中 2 位为小数位
operation	STRING		否	限制值只有 4 种，deposit,payment,income,refund
orderId	UUID	外键 (order.orderId)	是	订单编号

表 4-3 描述了 Carts 表的详细信息。

表 4-3：购物车表

字段	数据类型	主键、外键	允许空	说明
cartId	UUID	主键	否	购物车编号
userId	UUID	外键 (users.userId)	否	用户编号
productId	UUID	外键	是	商品编号

		(products.productId)		
quantity	INTEGER		否	添加到购物车的商品数量
addedAt	DATE		否	添加到购物车时间，默认为当前

表 4-4 描述了 IdempotencyKey 表的详细信息。

表 4-4：幂等键表

字段	数据类型	主键、外键	允许空	说明
idempotencyKeyId	UUID	主键	否	幂等键表编号
key	STRING		否	幂等键的值，用于唯一标识一次操作。
userId	UUID	外键 (users.userId)	否	用户编号
operation	STRING		否	操作类型
expiry	DATE		否	幂等键过期时间，当前

表 4-5 描述了 Image 表的详细信息。

表 4-5：商品图片表

字段	数据类型	主键、外键	允许空	说明
imageId	UUID	主键	否	图片编号
data	BLOB('long')	无	否	存储图片的二进制数据。
mimeType	STRING	无	否	图片的 MIME 类型。
productId	UUID	外键	否	产品编号

		(products.productId)		
isCover	BOOLEAN	无	否	表示该图片是否为产品的封面图

表 4-6 描述了 order 表的详细信息。

表 4-6：订单表

字段	数据类型	主键、外键	允许空	说明
orderId	UUID	主键	否	订单编号
productId	UUID	外键（product.productId)	是	商品编号
quantity	INTEGER	否	否	订单中商品数量
total	DECIMAL(10, 2)	否	否	订单的总金额
price	DECIMAL(10, 2)	否	否	单个产品的价格
sellerId	UUID	外键（user.userId)	否	卖家编号
buyerId	UUID	外键(user.userId)	否	买家编号
status	STRING	否	否	订单的状态

表 4-7 描述了 Product 表的详细信息。

表 4-7：商品表

字段	数据类型	主键、外键	允许空	说明
productId	UUID	主键	否	商品编号
name	STRING	否	否	商品名称
price	DECIMAL(10, 2)	否	否	商品的价格，使用精确浮点类型，精度为小数点后 2 位。

stock	INTEGER	否	否	商品的库存数量
expiryDate	DATE	否	是	商品的到期日期
description	TEXT	否	否	商品的详细描述
sellerId	UUID	外键（user.userId）	否	卖家编号

表 4-8 描述了 review 表的详细信息。

表 4-8：评价表

字段	数据类型	主键、外键	允许空	说明
reviewId	UUID	主键	否	评价编号
content	TEXT	无	否	评价的内容
rating	INTEGER	无	否	评分
reviewerId	UUID	外键（user.userId）	否	评价发起者
reviewedId	UUID	外键（user.userId）	否	被评价者
type	STRING	无	否	评价类型，表示评价是从买家到卖家（buyerToSeller）还是从卖家到买家（sellerToBuyer）。
orderId	UUID	外键（order.orderId）	否	订单编号

表 4-9 描述了 user 表的详细信息

表 4-9：用户信息表

字段	数据类型	主键、外键	允许空	说明
userId	UUID	主键	否	用户编号
username	STRING		否	用户名
phone	STRING		否	用户的手机号，仅允许数字

passwordHash	STRING		否	加密后的用户密码，用于登录验证。
address	STRING		否	用户的居住地址
balance	DECIMAL(10, 2)		否	用户账户余额，保留两位小数，默认为20。
lastDepositTime	DATE	无	是	用户最后一次充值时间

第三章　模块设计

本程序按照功能划分为多个模块，每个模块包含多个子模块。通过这种模块化设计，系统各部分功能清晰分工，便于维护和扩展，同时提升了开发效率和代码的可读性。主要模块包括订单模块、购物车模块、账单模块和图片管理模块。比如订单模块负责处理用户的订单相关操作，主要子模块包括：订单创建子模块、订单管理子模块和订单查询子模块：提供订单搜索和过滤功能，方便用户查找特定订单。本章将详细介绍订单模块中的订单创建模块（两种方式）和评价订单的对方（比如买家评价卖家，卖家评价买家）的模块。

3.1　订单创建模块（由单种商品创建订单）

创建订单是买家的功能，买家可以选择对单个选择的商品结账或者对购物车内的商品批量结账，此处是买家对单个商品进行结账的场景。具体步骤如下：

1. 检查请求体中是否包含了 idempotencyKey。我们涉及了一个幂等性 Key 用于防止客户端因为网络抖动或其他原因重复发送相同的请求，导致重复购买。如果缺少这个 key，则无法保证请求的幂等性。

2. 检查购买数量是否有效: 购买数量必须是正数，并且必须没有实际意义。

```
router.post('/product', userExtractor, async (req, res, next) => {
  const { idempotencyKey, productId, quantity } = req.body;

  if (!idempotencyKey) {
    return res.status(400).json({ error: 'Idempotency-Key is required' });
  }

  if (quantity <= 0) {
    return res.status(400).json({ error: 'Invalid quantity' });
  }
```

3. 开启数据库事务，处理用户创建订单事物: 事务保证了多个数据库操作的原子性，即这些操作要么全部成功，要么全部失败。这对于涉及资金和库存变更的关键操作至关重要，防止数据不一致。事务包括以下步骤

1. 检查是否存在相同的幂等性 Key：如果存在，回滚事务并返回冲突响应。

```
const t = await sequelize.transaction();
try {
  const existingKey = await IdempotencyKey.findOne({ where: { key: idempotencyKey }, transaction: t })
  if (existingKey) {
    await t.rollback();
    return res.status(409).json({ error: 'Duplicate request' });
  }
```

图 5-1：订单创建模块（由单种商品创建订单）步骤 1

2. 从请求对象中获取当前用户:

3. 获取商品信息并锁定该行: 对查找到的商品记录加上一个排他锁。这意味着在当前事务完成之前，其他事务无法修改这条商品记录，从而避免了并发修改导致的数据不一致性问题。因为需要获取最新的商品信息（特别是库存）来进行校验，并防止在购买过程中库存被其他请求修改。之后检查商品是否存在和验证商品库存是否足够，如果不够则报错。

```
const user = req.user;

// 获取商品信息并锁定行
const product = await Product.findByPk(productId, {
  transaction: t,
  lock: t.LOCK.UPDATE
});
if (!product) throw new Error('商品不存在');

// 验证库存是否足够
if (product.stock < quantity) {
  throw new Error('库存不足');
}
```

图 5-2：订单创建模块（由单种商品创建订单）步骤 2 与步骤 3

4. 计算订单总价，重新加载用户信息并锁定该行，防止在扣款过程中用户的余额被其他事务修改。使扣款前的用户信息使最新的用户信息，同时再次锁定可以确保使用的是最新的用户数据，并防止并发修改。

5. 再次检查用户余额是否足够 (在获取锁之后):用户余额可能在之前的检查和现在的扣款操作之间被其他事务修改。加锁后再次检查可以确保扣款操作的准确性。

```
const totalPrice = product.price * quantity;

// Acquire a row-level lock on the user
await user.reload({ transaction: t, lock: t.LOCK.UPDATE });

// Recalculate balance after locking
if (user.balance < totalPrice) {
  throw new Error('余额不足'); // 'Insufficient balance'
}
```

图 5-3：订单创建模块（由单种商品创建订单）步骤 4 与步骤 5

6. 原子性地扣除用户余额并原子性地减少商品库存

```
// Deduct the total price from user balance atomically
await user.decrement('balance', { by: totalPrice, transaction: t });

// 减少商品库存 atomically
await product.decrement('stock', { by: quantity, transaction: t });
```

图 5-4：订单创建模块（由单种商品创建订单）步骤 6

7. 创建订单记录，创建账单记录，并保存幂等性 Key（ 标记这个请求已经被处理过，用于后续的幂等性检查）。

```
// 创建订单
const order = await Order.create({
  buyerId: user.userId,
  sellerId: product.sellerId,
  productId: product.productId,
  total: totalPrice,
  price: product.price,
  quantity: quantity,
  status: 'Pending',
}, { transaction: t });

// 创建账单
await user.createBill({
  orderId: order.orderId,
  amount: - totalPrice,
  operation: 'payment',
}, { transaction: t });

// 保存Idempotency Key
await IdempotencyKey.create({
  key: idempotencyKey,
  userId: user.userId,
  operation: 'POST /product',
  expiry: new Date(Date.now() + 24 * 60 * 60 * 1000), // 24 hours expiry
}, { transaction: t });
```

图 5-5：订单创建模块（由单种商品创建订单）步骤 7

如果所有操作都成功完成，提交数据库事务。提交后，所有的数据变更将永久保存到数据库中，并且前端返回 201 Created 状态码，表示资源创建成功，并返回新创建的订单信息。

```
await t.commit();
res.status(201).json(order);
```

图 5-6：订单创建模块（由单种商品创建订单）事务提交

如果上述过程中发生任何错误，则回滚事务，并返回错误相应。这可以保证数据的一致性，撤销所有未完成的数据库操作。

```
} catch (error) {
  await t.rollback();
  // throw(error)
  return res.status(400).json({ error: error.message });
}
```

图 5-7：订单创建模块（由单种商品创建订单）步骤 4 与步骤 5

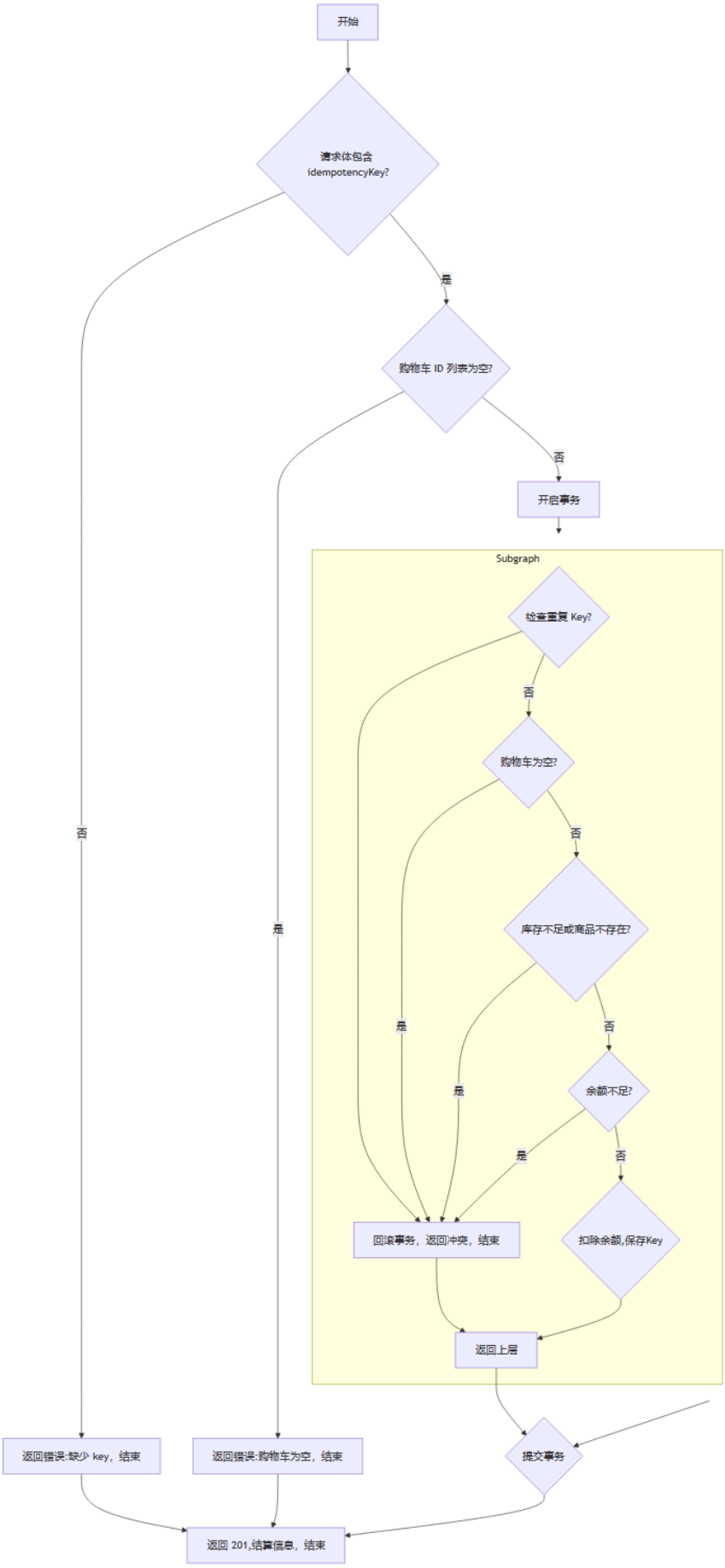

开始
请求体包含 idempotencyKey?
是
否
购物车 ID 列表为空?
否
是
开启事务
Subgraph
检查重复 Key?
否
是
购物车为空?
否
是
库存不足或商品不存在?
否
是
余额不足?
否
是
回滚事务，返回冲突，结束
扣除余额,保存Key
返回上层
提交事务
返回错误:缺少 key，结束
返回错误:购物车为空，结束
返回 201,结算信息，结束

图 5-8：用户由单个商品创建订单流程图（后端）

3.2 订单创建模块（由购物车批量创建订单）

创建订单是买家的功能，买家可以选择对单个选择的商品结账或者对购物车内的商品批量结账，此处是买家对购物车内商品进行结账的场景。具体步骤如下：

1. 检查请求体中是否包含了 idempotencyKey。我们涉及了一个幂等性 Key 用于防止客户端因为网络抖动或其他原因重复发送相同的请求，导致重复购买。如果缺少这个 key，则无法保证请求的幂等性。

2. 检查购物车 ID 列表是否为空。 如果 cartIds 为空或未提供，则表示购物车中没有商品需要结算，此时应该返回错误。

```
const { idempotencyKey, cartIds } = req.body;
if (!idempotencyKey) {
  return res.status(400).json({ error: 'Idempotency-Key is required' });
}

if (!cartIds || cartIds.length === 0) {
  return res.status(400).json({ error: '购物车为空' });
}
```

图 5-9：订单创建模块（由购物车批量创建订单）步骤 1 与步骤 2

3. 开启数据库事务，处理用户结算购物车事务。 事务保证了多个数据库操作的原子性，即这些操作要么全部成功，要么全部失败。这对于涉及多个商品、资金和库存变更的关键操作至关重要，防止数据不一致。事务包括以下步骤：

1. 检查是否存在相同的幂等性 Key。 查询 IdempotencyKey 表，判断当前请求是否是重复请求。如果存在，回滚事务并返回冲突响应。 如果找到了相同的 idempotencyKey，说明这是一个重复请求，为了保证幂等性，需要终止当前操作并告知客户端。

2. 从请求对象中获取当前用户。 通过中间件提取当前认证的用户信息。

```
const t = await sequelize.transaction();
try {
  const existingKey = await IdempotencyKey.findOne({ where: { key: idempotencyKey }, transaction: t });
  if (existingKey) {
    await t.rollback();
    return res.status(409).json({ error: 'Duplicate request' });
  }

  const user = req.user;
```

图 5-10：订单创建模块（由购物车批量创建订单）步骤 3-1 与步骤 3-2

3. 获取购物车项。 根据 cartIds 和用户 ID 查询购物车中的商品项，并预加载关联的商品信息。检查购物车是否为空。 确认根据提供的 cartIds 实际找到了购物车项，如果找不到则说明购物车为空。

```
const carts = await Cart.findAll({
  where: {
    cartId: cartIds,
    userId: user.userId,
  },
  include: [{
    model: Product,
    as: 'Product'
  }],
  transaction: t
});

if (carts.length === 0) throw new Error('购物车为空');
```

图 5-10：订单创建模块（由购物车批量创建订单）步骤 3-3

4. 验证所有商品库存是否足够。 遍历购物车中的每个商品项：

```
// 验证所有商品库存是否足够
for (const cart of carts) {
  // 获取商品信息并锁定行
  const product = await Product.findByPk(cart.productId, {
    transaction: t,
    lock: t.LOCK.UPDATE
  });
  const quantity = cart.quantity;

  if (product.stock < quantity) {
    throw new Error(`商品 ${product.name} 库存不足`);
  }

  // 减少商品库存 atomically
  await product.decrement('stock', { by: quantity, transaction: t });
}
```

图 5-12：订单创建模块（由购物车批量创建订单）步骤 4

1. 获取商品信息并锁定该行。对查找到的商品记录加上一个排他锁。这意味着在当前事务完成之前，其他事务无法修改这条商品记录，从而避免了并发修改导致的数据不一致性问题。因为需要获取最新的商品信息（特别是库存）来进行校验，并防止在购买过程中库存被其他请求修改。

2. 检查商品是否存在：确保根据购物车项中的 `productId` 能够找到对应的商品。

3. 检查商品库存是否足够。判断当前商品的库存是否满足购物车项中的购买数量。

4. 原子性地减少商品库存。 使用数据库的原子操作减少商品库存。

5. 计算订单总价。 初始化订单总价为 0，并在后续步骤中累加。

```
let totalPrice = 0;
```

图 5-13：订单创建模块（由购物车批量创建订单）步骤 5

6. 遍历购物车项，创建订单记录，创建账单记录，并更新总价。 对于购物车中的每个商品项：

```
// 遍历购物车项，创建订单商品关联并更新库存
for (const cart of carts) {
  const product = cart.Product;
  const quantity = cart.quantity;

  const order = await Order.create({
    productId: product.productId,
    buyerId: user.userId,
    sellerId: product.sellerId,
    quantity: quantity,
    price: product.price,
    total: product.price * quantity,
  }, { transaction: t });

  // 创建账单
  await user.createBill({
    orderId: order.orderId,
    amount: - product.price * quantity,
    operation: 'payment',
  }, { transaction: t });

  totalPrice += product.price * quantity;

  // 删除已结算的购物车项
  await cart.destroy({ transaction: t });
}
```

图 5-14：订单创建模块（由购物车批量创建订单）步骤 6

1. 为用户创建一条新的订单记录和一个账单记录，记录这笔支付。
2. 累加总价。将当前商品的总价累加到订单总价中。
3. 删除已结算的购物车项。从购物车中移除已成功结算的商品项。

7. 重新加载用户信息并锁定该行。防止在扣款过程中用户的余额被其他事务修改。使扣款前的用户信息使最新的用户信息，同时再次锁定可以确保使用的是最新的用户数据，并防止并发修改。再次检查用户余额是否足够 (在获取锁之后)。 用户余额可能在之前的检查和现在的扣款操作之间被其他事务修改。加锁后再次检查可以确保扣款操作的准确性。

```
// Acquire a row level lock on the user
await user.reload({ transaction: t, lock: t.LOCK.UPDATE });

// Recalculate balance after locking
if (user.balance < totalPrice) {
  throw new Error('余额不足'); // 'Insufficient balance'
}

// Deduct the total price from user balance atomically
await user.decrement('balance', { by: totalPrice, transaction: t });
```

图 5-15：订单创建模块（由购物车批量创建订单）步骤 7

8. 原子性地扣除用户余额。使用数据库的原子操作从用户账户中扣除总价。结算完成后保存 Idempotency Key。 标记这个请求已经被处理过，用于后续的幂等性检查。

```
// 结算完成后保存Idempotency Key
await IdempotencyKey.create({
  key: idempotencyKey,
  userId: user.userId,
  operation: 'POST /cart',
  expiry: new Date(Date.now() + 24 * 60 * 60 * 1000), // 24 hours expiry
}, { transaction: t });
```

图 5-18：订单创建模块（由购物车批量创建订单）步骤 8

如果所有操作都成功完成，提交数据库事务。 提交后，所有的数据变更将永久保存到数据库中。前端返回 201 Created 状态码，表示资源创建成功，并返回结算信息。 告知客户端结算成功，并返回包含用户 ID 和总价的结算信息。

如果上述过程中发生任何错误，则回滚事务。 保证数据的一致性，撤销所有未完成的数据库操作并返回错误相应。 将错误信息返回给客户端，说明结算失败的原因。

```
    await t.commit();
    res.status(201).json({
      message: '结算成功',
      buyerId: user.userId,
      totalPrice: totalPrice,
    });
  } catch (error) {
    if (!t.finished) await t.rollback();
    // throw(error)
    res.status(400).json({ error: error.message });
  }
});
```

any

图 5-19：订单创建模块（由购物车批量创建订单）事务提交或者回滚

流程图如下所示：

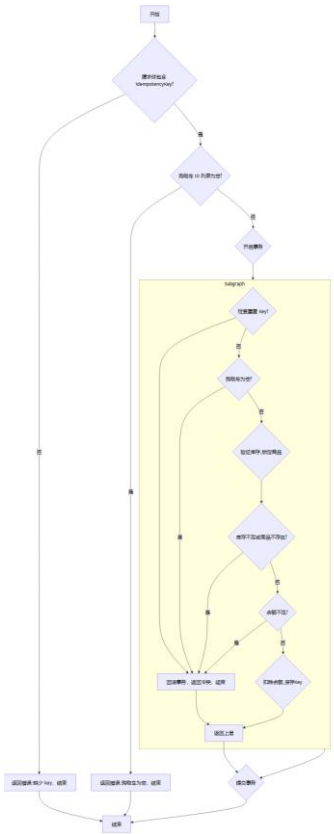


图 5-9：订单创建模块（由购物车批量创建订单）后端流程图

3.3 评价订单的对方（比如买家评价卖家，卖家评价买家）

评价订单是买家和卖家共有的功能，当卖家和买家存在订单关系后，他们便可以评价对方，但是一个订单尽可以评价一次对方。步骤如下：

1. 从请求体中提取评价内容、评分和订单 ID。

2. 根据订单 ID 查找对应的订单。评价是关联到特定订单的，需要确保该订单存在，并且后续需要使用订单信息来确定评价类型和被评价者。

3. 根据订单的信息判断评价者是买家还是卖家，并设置评价类型和被评价者 ID。评价只能由订单的买家或卖家发起，并且需要区分评价的方向。如果评价者既不是买家也不是卖家，则无权评价。

```
router.post('/', userExtractor, async (req, res) => {
  const { content, rating, orderId } = req.body
  const reviewerId = req.user.userId
  const order = await Order.findOne({ where: { orderId } })
  if (!order) {
    return res.status(404).json({ error: 'Order not found' })
  }
  let type, reviewedId
  if (order.buyerId === reviewerId) {
    type = 'buyerToSeller'
    reviewedId = order.sellerId
  }
  else if (order.sellerId === reviewerId) {
    type = 'sellerToBuyer'
    reviewedId = order.buyerId
  } else {
    return res.status(403).json({ error: 'Unauthorized' })
  }
```

图 5-16：评价订单的对方（比如买家评价卖家，卖家评价买家）步骤 1、步骤 2 与步骤 3

4. 检查是否已存在相同的评价。 在 Review 模型中查找是否已存在具有相同评价者 ID、被评价者 ID、订单 ID 和评价类型的评价记录，用于防止用户对同一订单的同一方进行重复评价。

5. 创建新的评价记录。则在 Review 模型中创建一条新的评价记录，包含评价内容、评分、评价者 ID、被评价者 ID、评价类型和订单 ID。返回 201 状

态码，表示资源创建成功，并返回新创建的评价信息。

```
const existingReview = await Review.findOne({
  where: { reviewerId, reviewedId, orderId, type }
});

if (existingReview) {
  return res.status(429).json({ error: 'Review already exists' });
}

const review = await Review.create({ content, rating, reviewerId,
  reviewedId, type, orderId })
res.status(201).json(review)
```

图 5-17：评价订单的对方（比如买家评价卖家，卖家评价买家）步骤 4 与步骤 5

流程图如下所示：

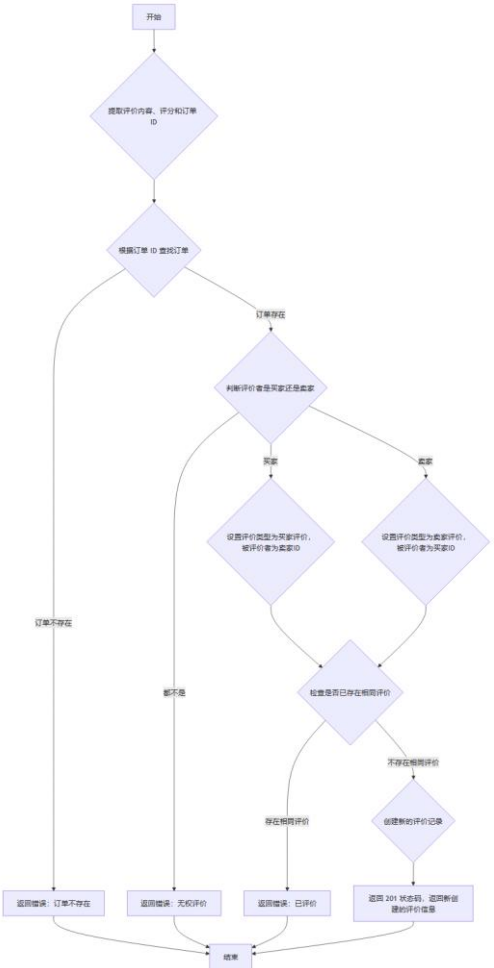


图 5-18：评价订单的对方（比如买家评价卖家，卖家评价买家）流程图

第四章　部署与应用

4.1　部署图

本系统采用 B/S 模式，当服务器端完全部署应用后，客户端即可通过浏览器进行相应的访问。

- **客户端 (Browser)：**用浏览器（如 Chrome、Firefox）即可完成访问。
- **前端服务 (Frontend)**：通过 Vite 打包生成静态文件，由服务器进行托管并提供访问。
- **后端服务 (Backend)**：基于 Node.js，使用 Express 框架，连接 Postgresql 数据库数据库并提供 API 服务。
- **数据库服务 (Database)**：使用 Postgresql 数据库作为主要数据存储。数据库通过 Docker 部署，或直接运行于远程服务器。

系统的部署图如图 6-1 所示：

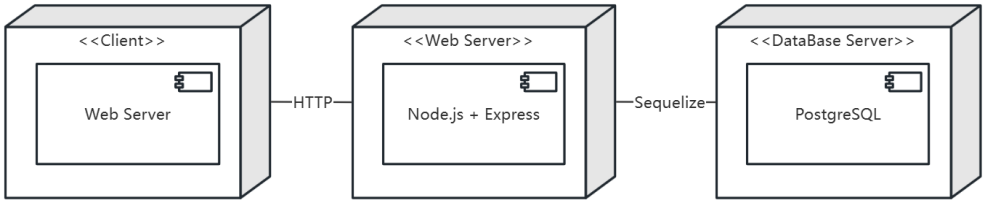


图 6-1：系统部署图

4.2　部署过程

4.2.1 Node.js 环境配置

本系统需要运行在 Node.js 环境下，可以按照下面的步骤进行：

1. 下载并安装最新版本的 Node.js，https://nodejs.org/。
2. 安装 Node.js 后，npm 会自动安装，可以使用以下命令更新 npm 到最新版本：`npm install npm@latest -g`

4.2.2 前端

在终端打开实验文件，然后依次执行以下命令运行前端：

```
$ cd frontend
```

```
# Install dependencies
$ npm install

# Start the application in dev environment
$ npm run dev

# Build the application for production
$ npm run build
```

```
# Preview the production build
$ npm run preview
```

```
# Lint the code
$ npm run lint
```

4.2.3 后端

在终端打开实验文件，然后按照以下步骤运行后端：

1. 进入到后端 backend 文件夹，在终端输入命令：

```
$ cd backend
```

2. 准备一个 PostgreSQL 数据库，例如用 dockers 运行 PostgreSQL：

(1) 参考 docker 安装文档 https://docs.docker.com/engine/install/ 安装 dockers。

(2) 在命令行中启动 PostgreSQL 的 docker 镜像：

```
docker run -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432
postgres
```

(3) 在 backend/.env 文件中配置 DATABASE_URL：

```
DATABASE_URL=postgres://postgres:mysecretpassword@localhost:5432
/postgres
```

<注>我们运行开发的方案是用 docker 运行 PostgreSQL 映像，如使用其他方式运行 PostgreSQL 数据库，可能要修改 backend/src/utils/db.js 中 sequelize 的初始化。其中连接 docker、PostgreSQL 映像的方式，连接远程 azure 远程数据库，heroku 远程数据库种 sequelize 的初始化文件已在 backend/src/utils/db.js 给出。

3. 在终端依次输入命令行运行后端：

```
# Install dependencies
$ npm install
```

```
  # Create a .env file and put there the DATABASE_URL for connecting
to your PostgreSQL database
  $ echo "DATABASE_URL=<YOUR-DATABASE-URL>" > .env

  # Set a variable ACCESS_TOKEN_SECRET which is a digital signature
ensures that only parties who know the secret can generate a valid
token.
  $ echo "ACCESS_TOKEN_SECRET=youraccesstokensecretphrase" >> .env

  # Initialize the database
  $ npm run init

  # Rollback the last migration
  $ npm run migration:down

  # Start the application in dev environment
  $ npm run dev

  # Start the application in test environment and run tests
  $ npm test
```

4.2.4 运行

使用 https://for-database-use.azurewebsites.net/ （此网站保留至 1 月 31 日）来访问网站或者本地运行程序在后端使用 npm start 命令 在 http://localhost:3000 访问网站

4.3 部分功能截图

网站注册页面如图 6-0 所示：

图 6-0：网站注册页面

网站登录页面如图 6-1 所示：

图 6-1：网站登录页面

登录成功后显示的页面-商品列表页如图 6-2 所示：

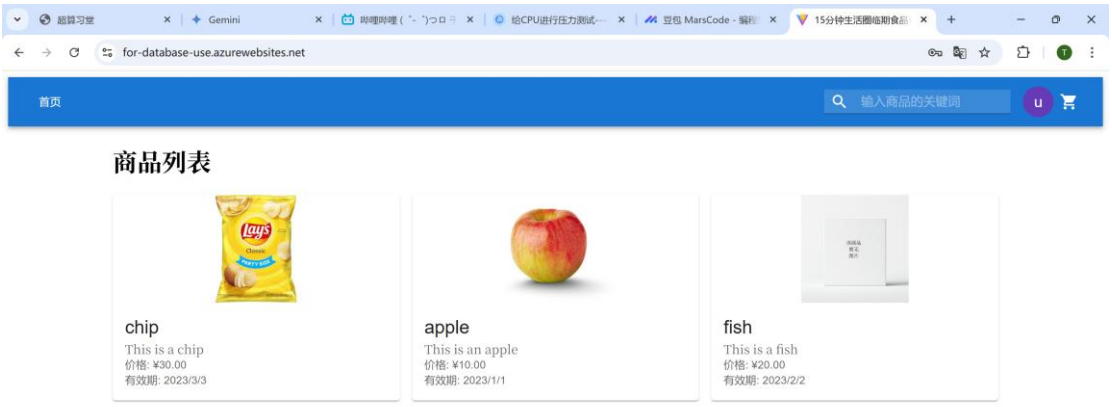


图 6-2：商品列表页面

点击商品，显示商品详情页，如图 6-3 所示：

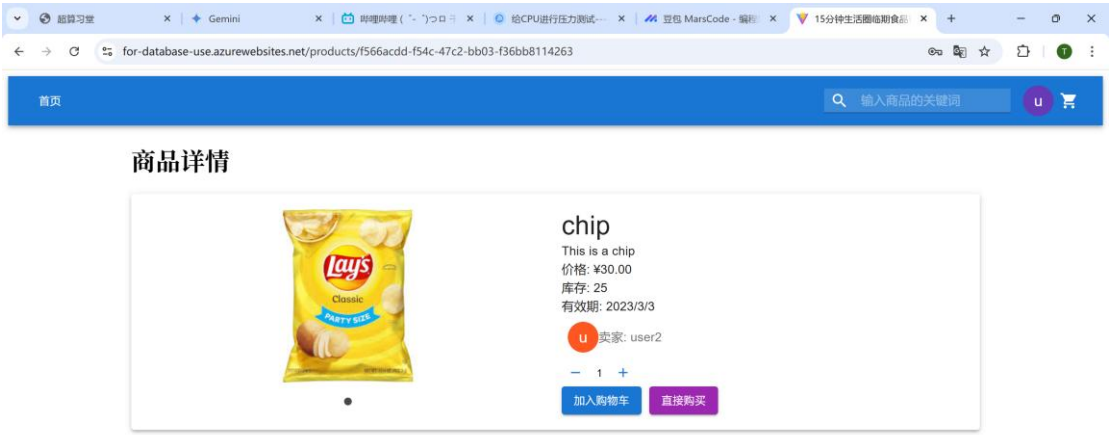


图 6-3：商品详情页

点击卖家，显示卖家信息页，如图 6-4 所示：

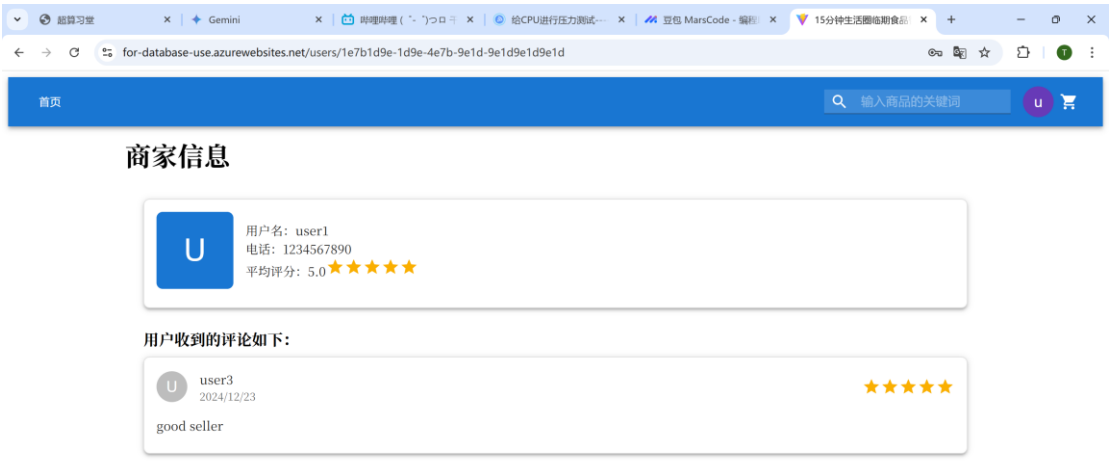
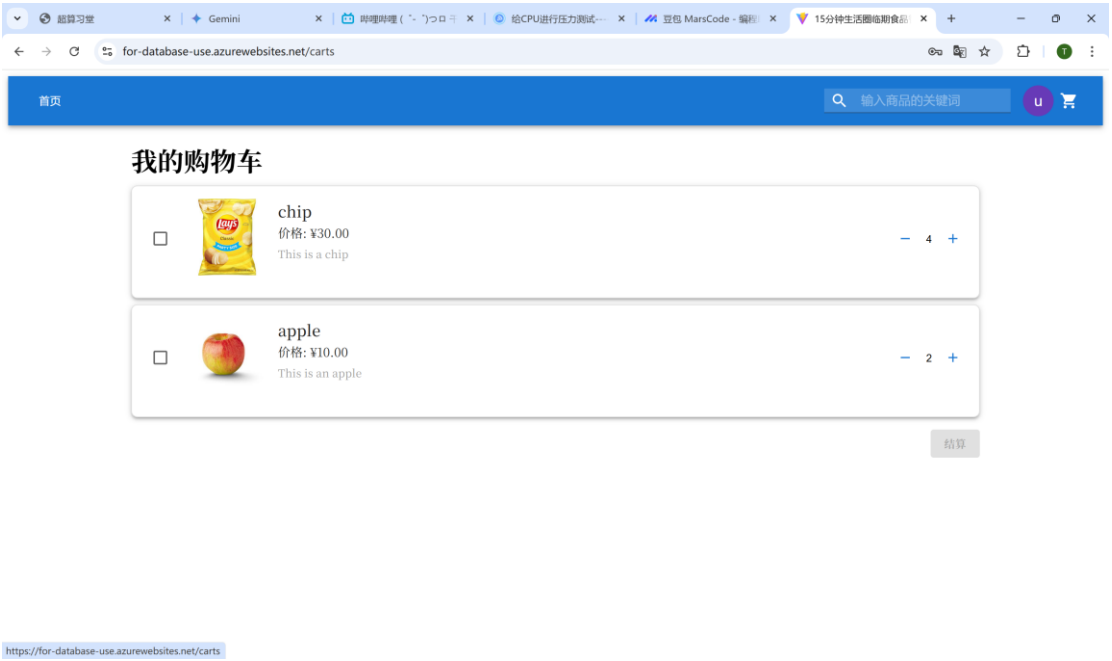


图 6-4：卖家信息页

将商品添加至我的购物车，我的购物车及购物车的结算页面，如图 6-5 所示：



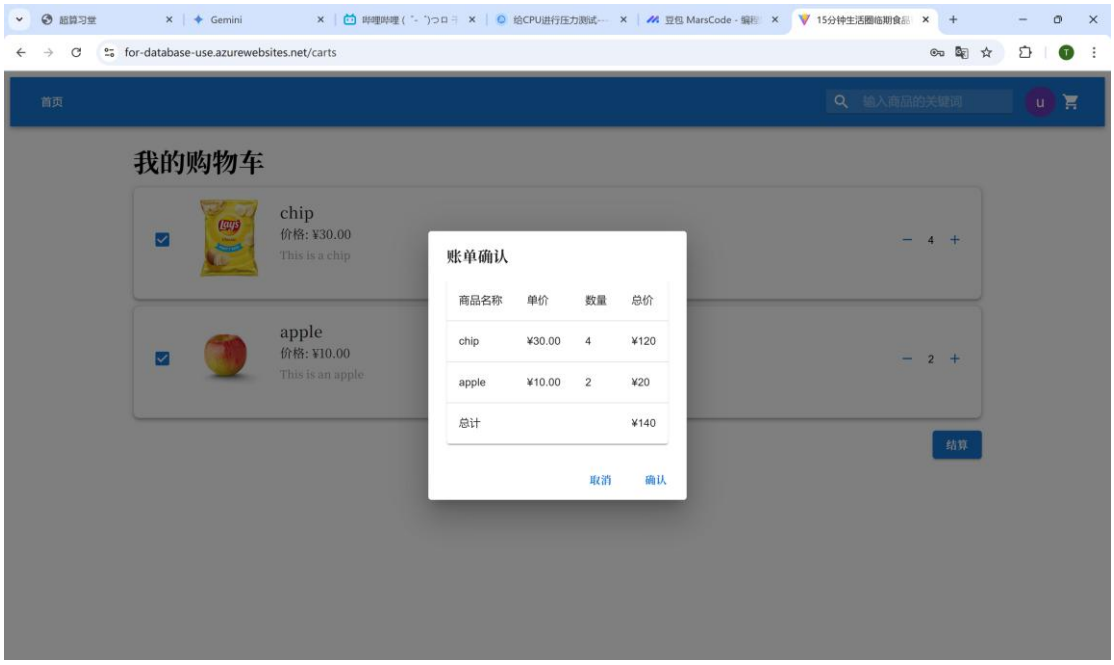


图 6-5：我的购物车及购物车中结算页面

我的订单页面，如图 6-6 和 6-7 所示：

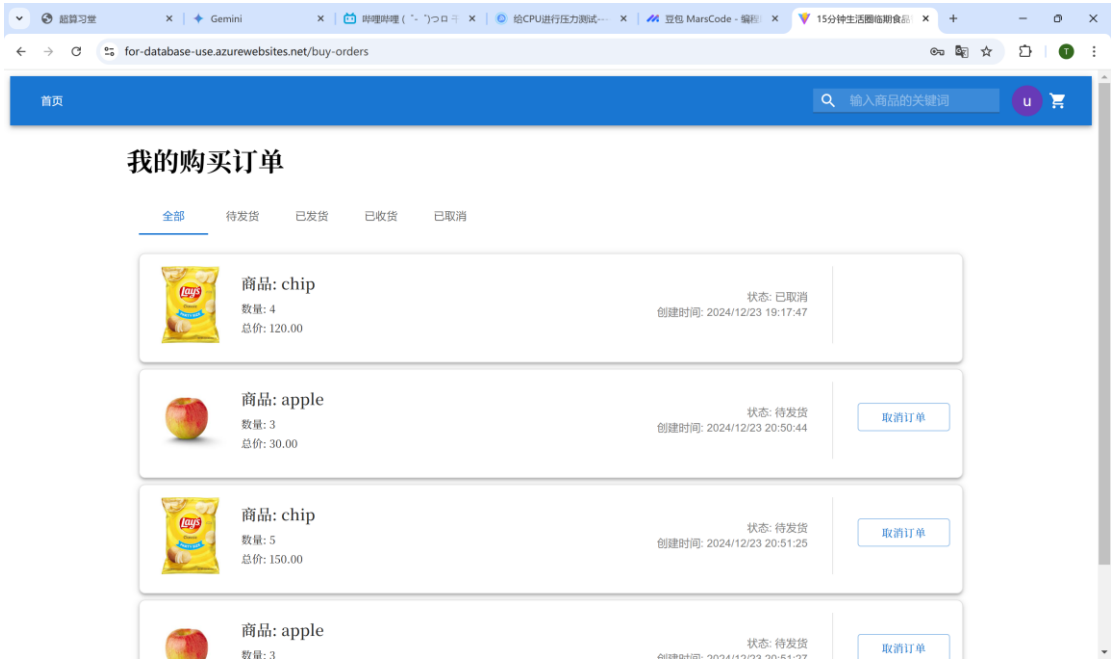


图 6-6：我的购买订单页面

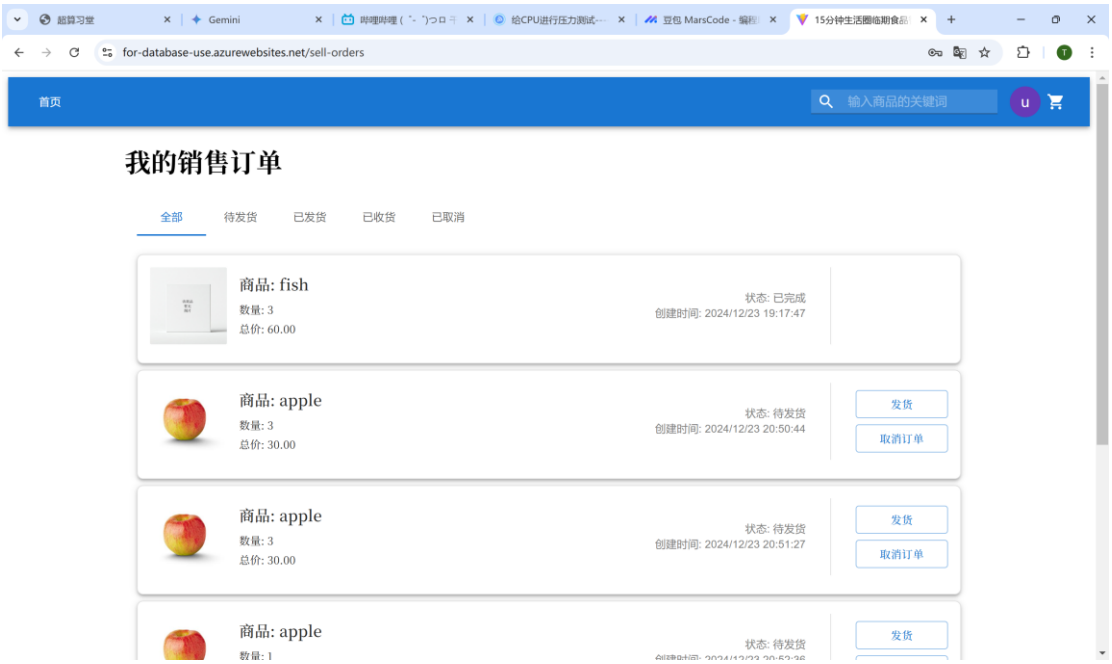


图 6-7：我的销售订单页面

我的商品页面，如图 6-8 所示：

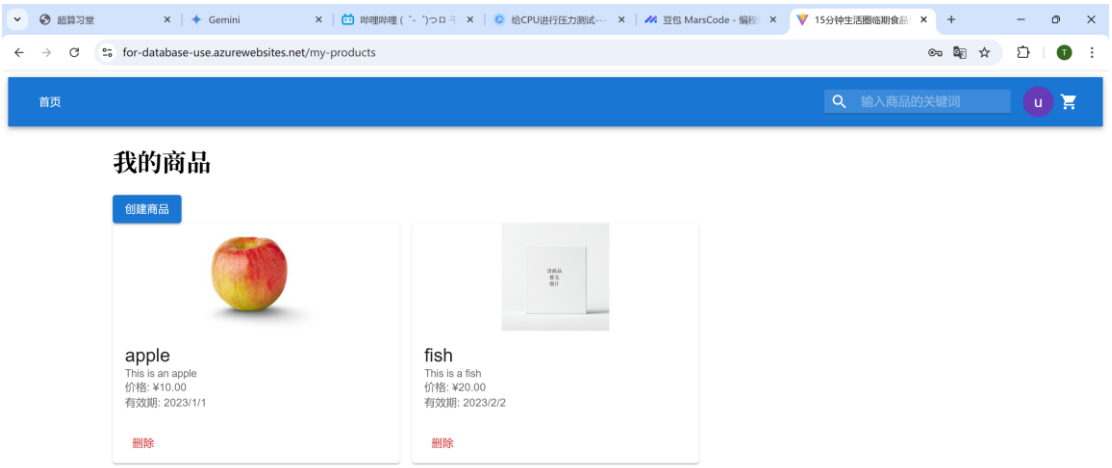


图 6-8：我的商品页面

我的商品-创建商品页面，如图 6-9 所示：

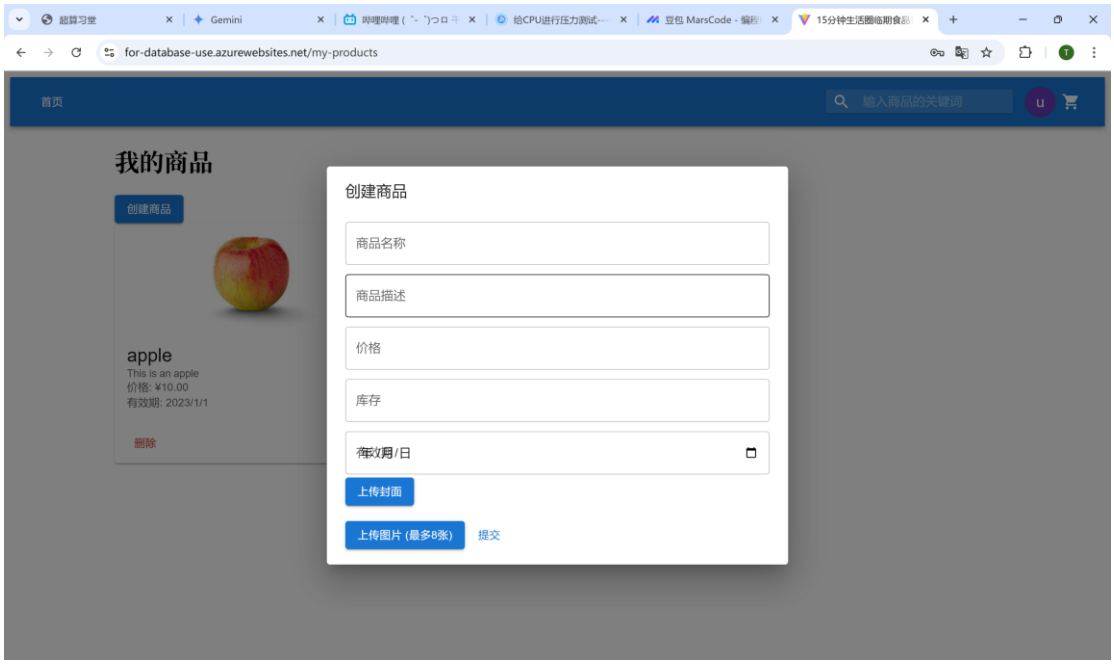
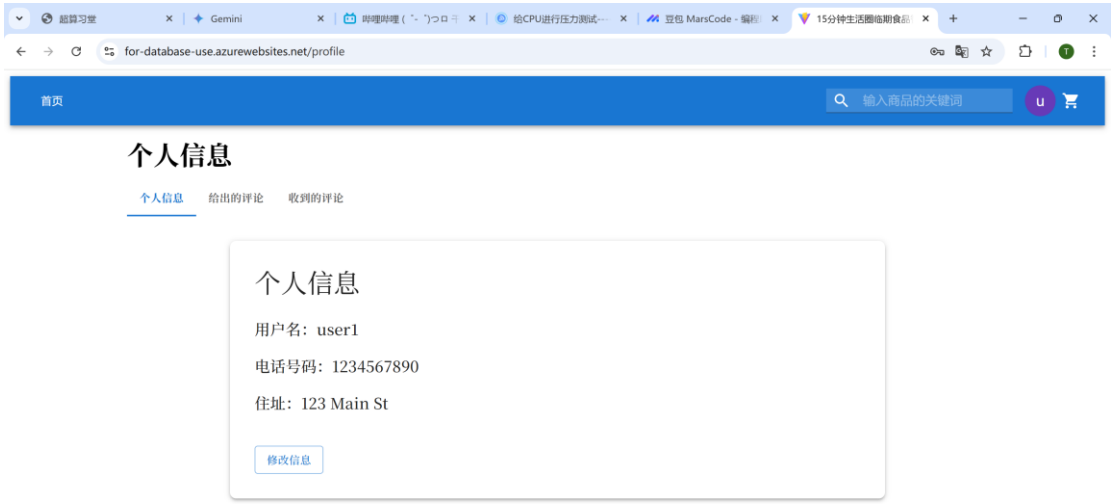


图 6-9：我的商品-创建商品页面

个人信息及修改个人信息页面，如图 6-10 所示：



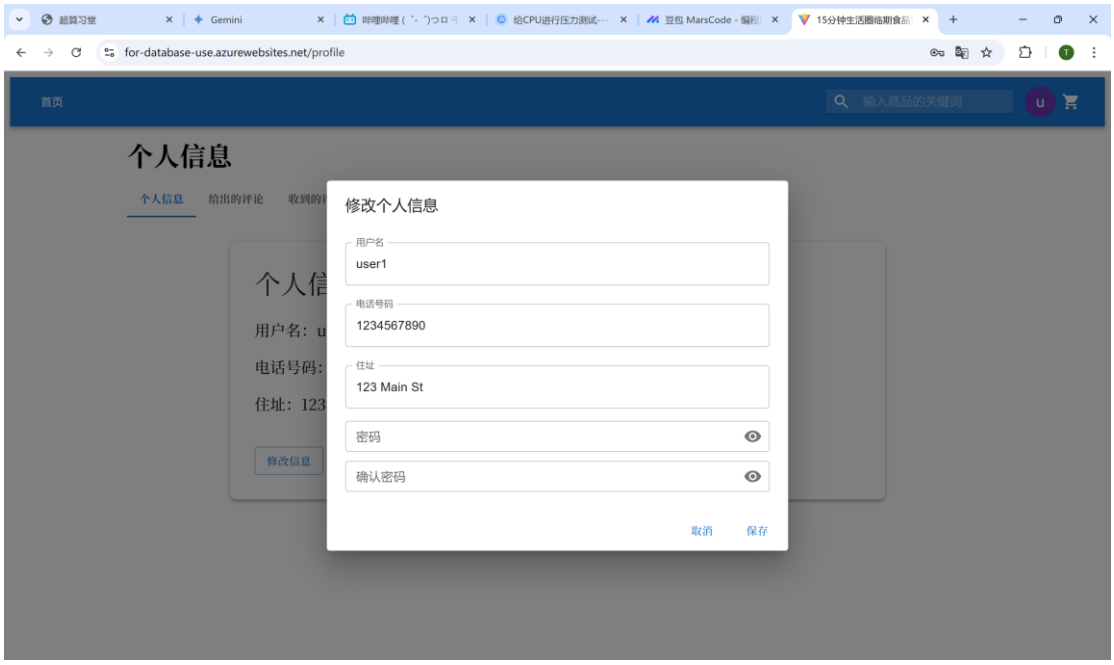


图 6-10：个人信息及修改个人信息页面

余额查看及充值页面，如图 6-11 所示：

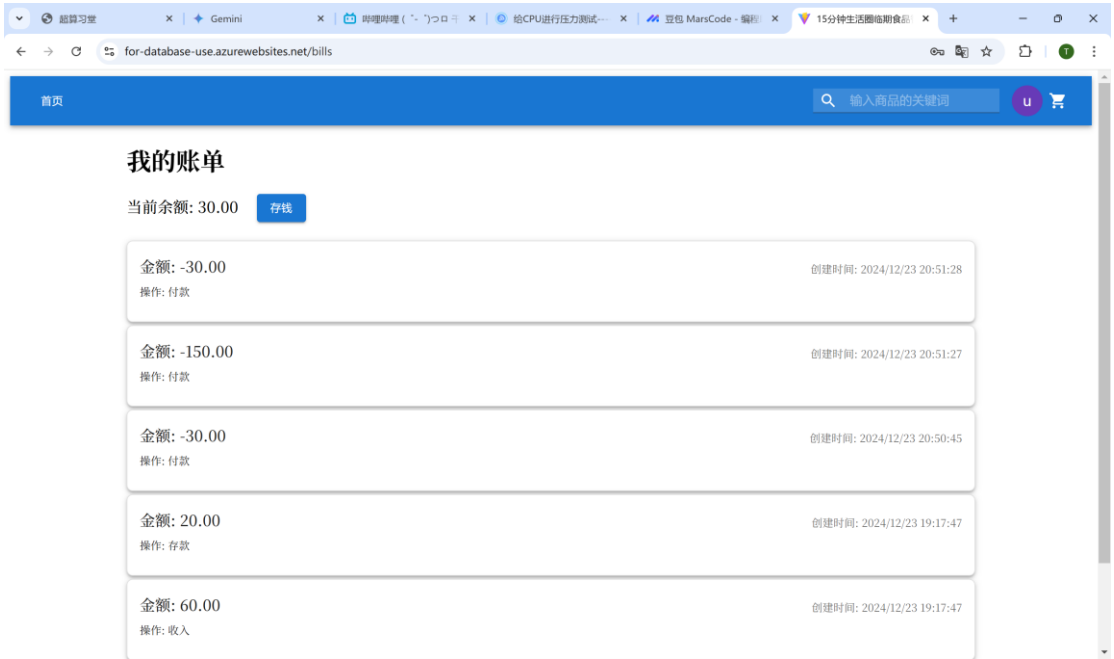


图 6-11：余额查看及充值页面