期末项目——在线商城

学号	姓名	分工
16337113	劳马东	数据库设计、web 前端
16337109	柯司博	数据库设计、数据库后端

一、 项目介绍

我们的项目是一个在线商城系统，我们用 python 的 django 编写前端，mysql 编写后端数据库。

系统功能：

顾客端：

- 商品分类：

系统提供根据不同商家进行商品分类的功能，顾客可以先选定商家再获取对应的商品列表。

- 商品信息获取：

顾客可以获取商品的名字、价格、描述等属性。

- 购物车与购买商品：

顾客可以把自己喜欢的商品添加指定数量进购物车，可以在选好需要的商品后为购物车里的商品结算，结算前需要填写个人名字、邮箱等信息。当然也可以删除掉购物车内未结算的商品。

商家端：

- 上架商品：

商家可以为自己的商店增加新的商品，自定义商品属性。

- 下架商品：

商家可以下架某种商品让其不可被购买，也可以直接删除某种商品。

- 修改商品信息

商家可以修改商品价格、数量、描述等信息。

- 删除用户订单

当商家觉得某些订单不合理时可以选择删除这些顾客订单。

二、 实验环境

系统	Windows 10
SQL	MySQL 8.0
工具	MySQL Workbench Power Designer Django

三、 实验过程

(一)数据库设计

1. 概念模型设计

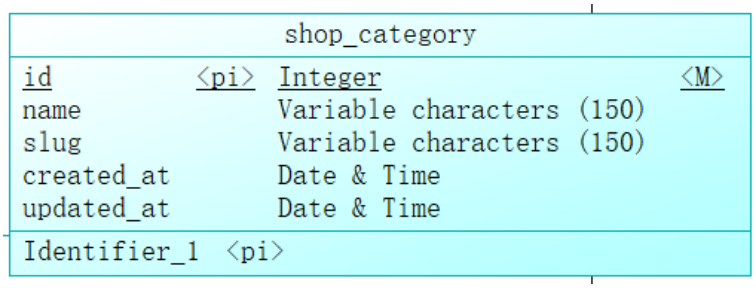
我们从实际生活中的交易出发，交易需要顾客和商家两端。

从用户的角度出发：我们设计了订单表 orders_order 和订单项表 orders_orderitem，一条订单项用于记录用户的一次购买记录，订单表用于记录多个订单。

从商家的角度出发：我们设计了商家表 shop_category 和商品表 product，商品表用于记录不同商品，商家表用于记录不同的商家。

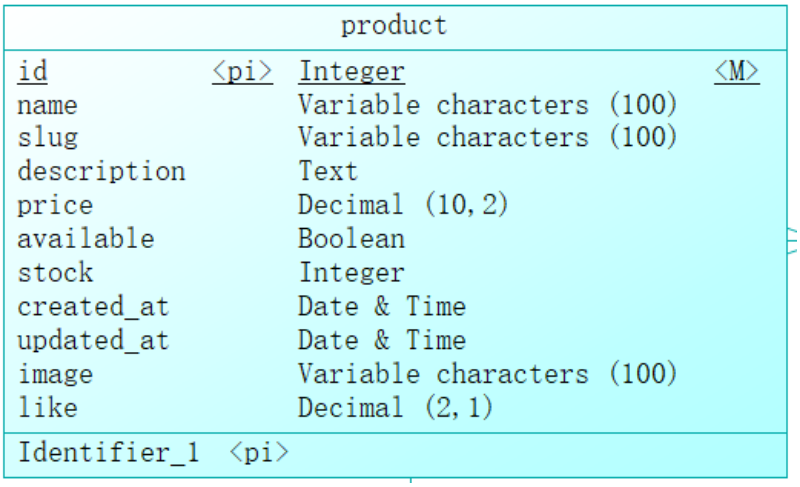
以下为 4 个表的设计：

shop_category:



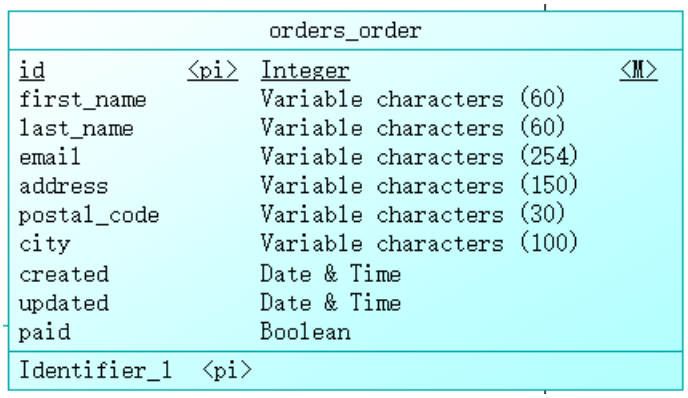
id	商家 id
name	商家名字
slug	商家网址
created_at	创建商家时间
updated_at	更新商家时间

product:



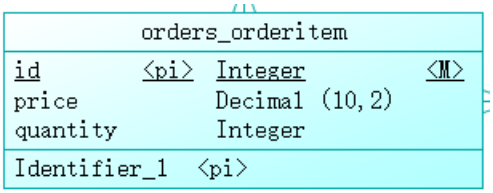
id	商品 id
name	商品名字
slug	商品网址
description	商品描述
price	商品价格
available	商品是否可以购买
stock	商品剩余数量
created_at	商品上架时间
updated_at	商品更新时间
image	商品图片保存路径
like	商家对自己产品的喜欢程度

orders_order:



id	订单号
first_name	顾客的 first name
last_name	顾客的 last name
email	顾客的 email
address	顾客的地址
postal_code	邮政编码
city	顾客所在的城市
created	订单创建时间
updated	订单更新时间
paid	顾客是否付钱了

orders_orderitem:



id	订单项 id
price	订单价格
quantity	购买产品数量

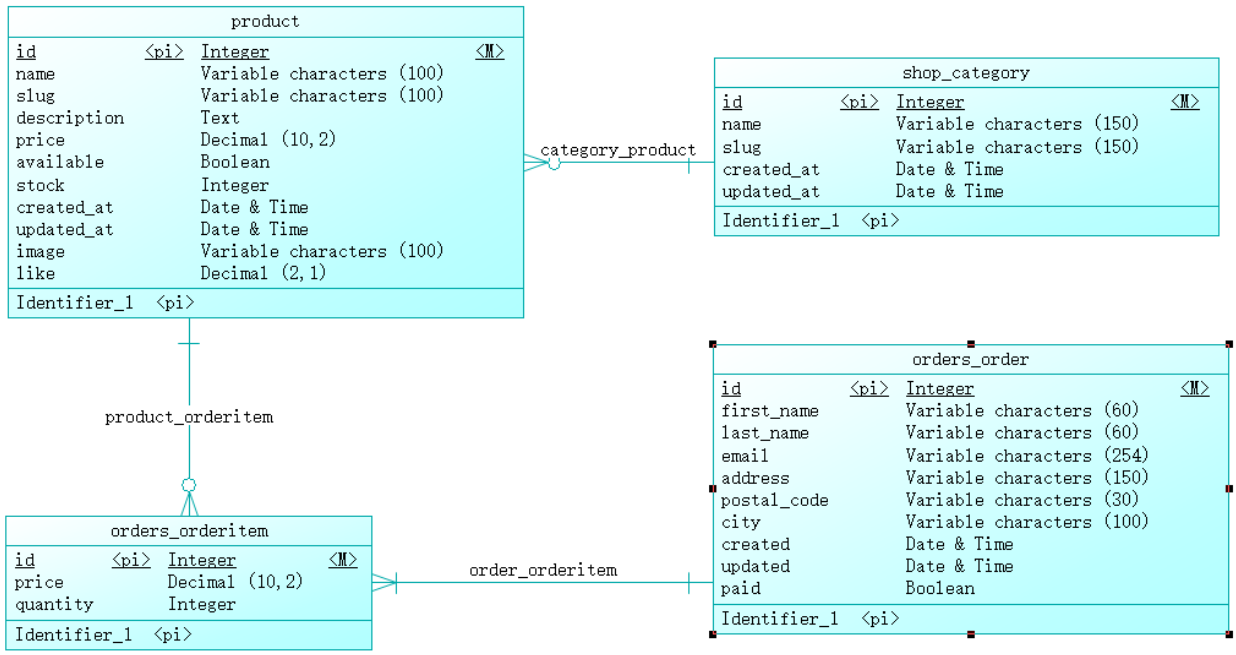
联系集设计：

一个商家可以有 0 种或 n 种商品，而一种商品必须只属于一个商家，因此商家表 shop_category 到商品表 product 是一对多的关系；

一种商品可以属于 0 条或多条订单项，而一条订单项必须只对应一种商品，因此商品表 product 到订单项 orders_orderitem 是一对多的关系；

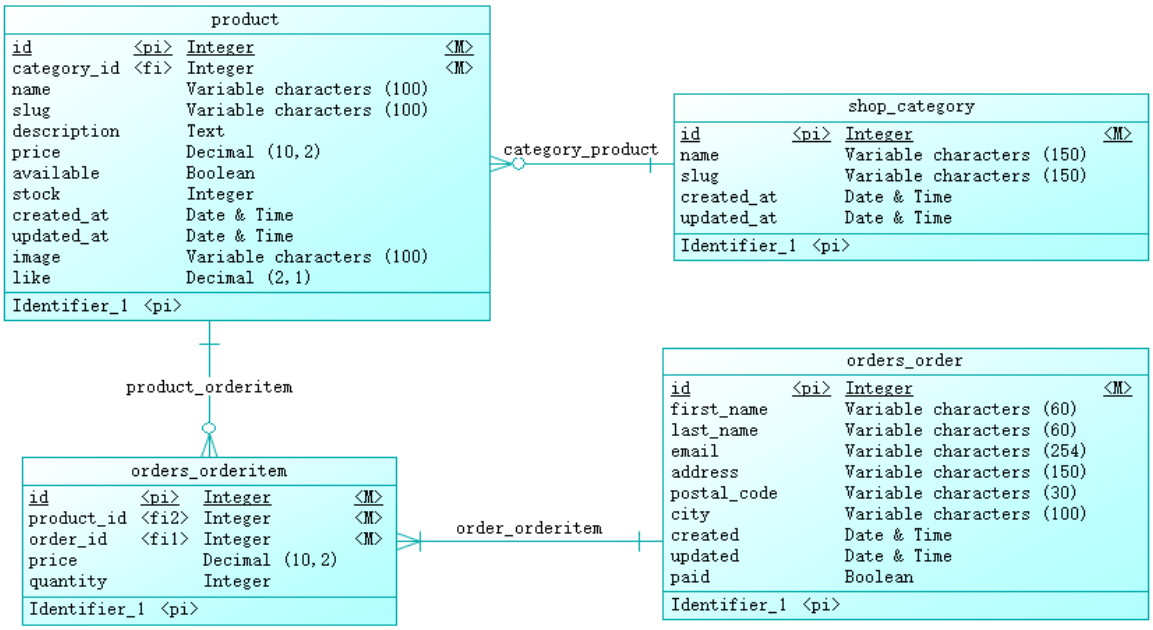
一个订单可以包含 1 个或多个订单项（因为没有订单项时是不会创建订单的），而一条订单项必须属于一个订单，因此订单到订单项是一对多的关系。

因此概念模型（ER 图）最终设计如下：



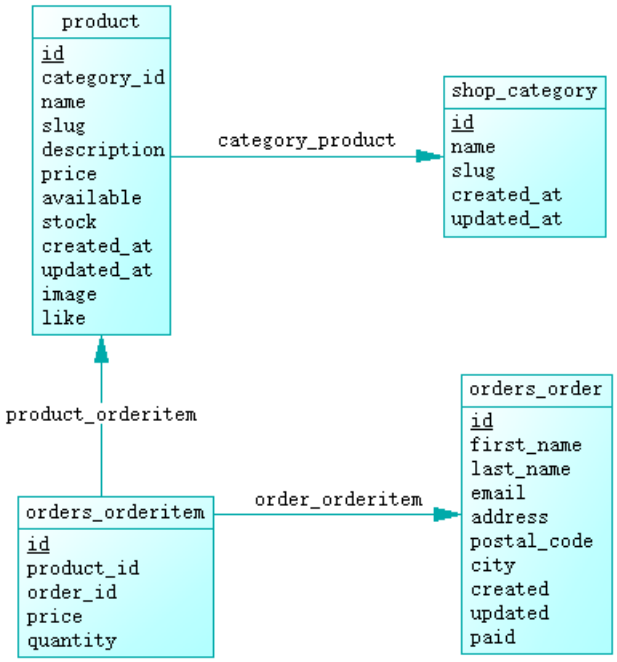
2. 逻辑模型设计

用 powerdesigner 从概念模型生成逻辑模型如下：



3. 物理模型设计

用 powerdesigner 从逻辑模型生成物理模型如下：



4. SQL 语句生成

从物理模型中生成建表 sql 语言：

product:

```
create table product
(
    product_id          int not null,
    category_id         int not null,
    product_name        varchar(100),
    product_slug        varchar(100),
    description         text,
    price               decimal(10,2),
    available           bool,
      stock               int,
    product_created_at  datetime,
    product_updated_at  datetime,
    image               varchar(100),
    "like"              decimal(2,1),
    primary key (product_id)
);
 alter table product add constraint FK_category_product foreign key (category_id)
       references shop_category (category_id) on delete restrict on update restrict;
```

shop_category:

```
create table shop_category
(
    category_id          int not null,
    category_name        varchar(150),
    category_slug        varchar(150),
    category_created_at  datetime,
    category_updated_at  datetime,
    primary key (category_id)
);
```

orders_orderitem:

```
create table orders_orderitem
(
    item_id              int not null,
    product_id           int not null,
    order_id             int not null,
    price                decimal(10,2),
    quantity             int,
    primary key (item_id)
);

 alter table orders_orderitem add constraint FK_order_orderitem foreign key (order_id)
       references orders_order (order_id) on delete restrict on update restrict;

 alter table orders_orderitem add constraint FK_product_orderitem foreign key (product_id)
       references product (product_id) on delete restrict on update restrict;
```

orders_order:

```
create table orders_order
(
    order_id             int not null,
    first_name           varchar(60),
    last_name            varchar(60),
    email                varchar(254),
    address              varchar(150),
    postal_code          varchar(30),
    city                 varchar(100),
```

```
    created             datetime,
    updated             datetime,
    paid                bool,
    primary key (order_id)
);
```

(二) 导入数据

有两个表需要导入数据的：商店类型表（商家表）shop_category，商品表 product。我们的数据主要为“华为”、“联想”等电子产品店铺以及它们的产品“手机”和“电脑”，因为数据量不大因此由我们手动输入 tbl 文件，根据两个表各自的属性构造不同的数据，数据格式如下：

以 shop_category 为例：

```
1|联想专卖店|联想专卖店|2018-12-26 14:20:19.792778|2018-12-26 14:20:19.792778CRLF
2|华硕专卖店|华硕专卖店|2018-12-26 14:20:19.792778|2018-12-26 14:20:19.792778CRLF
3|苹果专卖店|苹果专卖店|2018-12-26 14:20:19.792778|2018-12-26 14:20:19.792778CRLF
4|华为专卖店|华为专卖店|2018-12-26 14:20:19.792778|2018-12-26 14:20:19.792778CRLF
```

以’|’为属性值之间的分隔符，以回车’\n’（图中的 CR LF 标记）为元组之间的分隔符。product 的数据格式同理，然后执行以下 mysql 指令导入数据：

```
LOAD DATA INFILE 'D:/data/shop_category.tbl' REPLACE INTO TABLE shop_category FIELDS TERMINATED BY '|' LINES TERMINATED BY '\n';
LOAD DATA INFILE 'D:/data/shop_product.tbl' REPLACE INTO TABLE product FIELDS TERMINATED BY '|' LINES TERMINATED BY '\n';
```

(三) 创建远程用户并授权（自主存储管理）

为了方便两台主机之间远程操控数据库，我们把一台主机作为服务器，然后另一台主机作为远程用户。

- 用根用户登录后创建名字为’laomd’、密码为 123456 的用户：

  ```
  CREATE USER 'laomd'@'%' IDENTIFIED BY '123456';
  ```

  其中’%’表示该用户可以远程访问。

- 为了更符合安全性的要求，只授予该用户对实验数据库’ecommerce’中所有表的所有操作权限：

  ```
  GRANT all ON ecommerce.* TO 'laomd'@'%';
  ```

  其中 all 表示所有权限，ecommerce.*表示 ecommerce 数据库的所有表。

(四)视图的使用

原始数据表 product 中字段名为'like'的属性表示的是商家对自己产品的喜欢程度（百分比），而我们在做实际数据库时为了客观并不打算把这点呈现出来，因此我们用视图的方式把除了这个属性外的其他属性提供给 python 接口，即只选取除了'like'属性外的其他属性来构建视图'shop_product'：

```
create view shop_product
as select
    id,name,slug,description,
    price,available,stock,created_at,
    updated_at,image,category_id
from product;
```

(五)触发器设计

当顾客购买商品、商家在后台删除一条购买记录或更新购买记录时，我们会在 orders_orderitem 表中插入、删除或更新对应元组，而这涉及到一个很重要的问题就是商品剩余数量是否足够，因此我们创建了 3 个 before 触发器来进行检查并更改数据。

- insert 触发器

  当顾客购买一定数量（quantity）的商品时，需要检查商品剩余数量 stock 是否足够，如果足够就允许插入并修改 shop_product 中的对应商品数量：

```
delimiter $
create trigger TRI_INSERT_ORDERITEM
before insert on orders_orderitem
    for each row
    begin
        -- 保存购买数量
        declare purchase_num int;
        -- 保存商品剩余数量
        declare remain_num int;
          set purchase_num = new.quantity;
          select stock into remain_num
          from shop_product
          where new.product_id = id;
          -- 只有当剩余数量大于购买数量时才允许插入并进行商品剩余数量的更新
          if (remain_num>=purchase_num) then
          begin
```

```
            update shop_product
            set stock = stock-purchase_num
            where id = new.product_id;
        end;
        -- 否则提示商品数量不足并拒绝插入
        else
            signal sqlstate '45000' set message_text = 'Remain quantity not enough!';
        end if;
    end$
delimiter ;
```

- delete 触发器

  当商户在后台删除购买记录时，不需要进行检查，直接触发增加 shop_product 中商品数量（stock）的功能即可：

```
delimiter $
create trigger TRI_DELETE_ORDERITEM
before delete on orders_orderitem
    for each row
    begin
        declare purchase_num int;
        set purchase_num = old.quantity;
        # 当初购买了多少数量现在就返回多少数量
        update shop_product
          set stock = stock + purchase_num
          where id = old.product_id;
    end$
delimiter ;
```

- update 触发器

  当商户要更新顾客的订单中商品数量时，需要检查商品剩余数量是否足够，主要分为两部分：

  如果新的数量更少，则直接返回减少的数量即可：

```
if (new.quantity <= old.quantity) then
    begin
        update shop_product
        set stock = stock+old.quantity-new.quantity
        where id = new.product_id;
    end ;
```

  而当更新后的数量更多了，则需要检查剩余商品数量是否足够抵扣多出来的部分：

```
else
    begin
```

```
        set increase_num = new.quantity - old.quantity;
        if (remain_num>=increase_num) then
        begin
            update shop_product
            set stock = stock-increase_num
            where id = new.product_id;
        end;
          else
              signal sqlstate '45000' set message_text = 'Remain quantity not enough!';
          end if;
    end;
```

(六) 商店模块

Django 按照 MTV 模式来开发 web 系统。M（Model）代表模型，负责业务对象和数据库的关系映射；T (Template)代表模板，负责如何把页面展示给用户；V（View）代表视图，负责业务逻辑。因此，我也会按照 MTV 三个部分来介绍项目中不同模块的设计。

在商店模块，我们希望实现这样的功能：

1) 用户访问我们的主页时，向他显示商店所有的商品；

2) 用户可按不同商品目录浏览商品；

3) 用户点击商品链接可查看商品详情。

1. 模型定义

显然，这部分需要两个实体——目录和商品。因此我们需要建立了两个对应的模型类——Category 和 Product，它们都继承自 Django 的 Model 类。

```
class Category(models.Model):
    name = models.CharField(max_length=150, db_index=True)
    slug = models.SlugField(max_length=150, unique=True, db_index=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('name', )
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

name 表示商品目录名，是一个字符串类型，对应 MySQL 中的 character；slug 属性用于 URL 查询，是一个 varchar；created_at 和 updated_at 分别是目录创建和修改的时间。MySQL 后端经常需要使用 name 属性进行查询，为了提高效率我们在 name 属性上创建索引；前端经常使用 slug 属性，因此它也建立了一个索引。此外，不同商品目录的 URL 应该是唯一的，于是 slug 属性需要时 unique。

Category 内部 Meta 类的 ordering 成员赋值表示按目录名排列目录，即在数据库表中每个元组按 name 属性排序。

```
class Product(models.Model):
    category = models.ForeignKey(Category, related_name='products',
                                 on_delete=models.CASCADE)
    name = models.CharField(max_length=100, db_index=True)
    slug = models.SlugField(max_length=100, db_index=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    stock = models.PositiveIntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)

    class Meta:
        ordering = ('name', )
        index_together = (('id', 'slug'),)
```

description 表示商品详情；available 表示商品是否在售；stock 是商品库存数量；image 是商品的图片，按上传日期存放在 products 目录下。

Meta 中的 index_together 表示将 id 和 slug 两个属性组合起来，创建一个索引。id 是 Model 类默认的主键。

2. 视图设计

这部分是与 HTML/CSS 交互，当客户点击一个 URL 时（如目录、商品），一个包含目录的 category_slug 或商品的 id 和 slug 的网络请求就会被 Post 到服务器。下面的

product_list 函数处理用户点击目录的请求，查询对应目录的商品并用上面的第一个模板返回网页；product_detail 处理用户点击商品的请求，根据商品 id 和 slug 查询数据库中的商品并返回页面。

```
def product_list(request, category_slug=None):
    category = None     # 默认为 None，表示显示所有商品
    categories = Category.objects.all()
    # 筛选出对应目录的商品
    if category_slug:
        category = get_object_or_404(Category, slug=category_slug)
        products = Product.objects.filter(category=category,
                                          available=True)
    else:
        products = Product.objects.filter(available=True)
    context = {     # HTML/CSS 的上下文变量
        'category': category,
        'categories': categories,
        'products': products
    }
    return render(request, 'shop/product/list.html', context)


def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id, slug=slug, available=True)
    context = {
        'product': product,
    }
    return render(request, 'shop/product/detail.html', context)
```

3. HTML 与 CSS 模板设计

在浏览器的右上方提供一个 sidebar，用户点击“所有”可以查看所有的商品，点击对应目录名可导航到该目录的所有商品。

```
{# 显示商品目录 sidebar #}
<div class="sidebar-module">
  <h4>商品目录</h4>
  <ol class="list-unstyled">
    <li class="active">
        <a href="{% url 'shop:product_list' %}">所有</a>
    </li>
   {% for c in categories %}
     <li class="active">
        <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
     </li>
   {% endfor %}
  </ol>
</div>
```

客户点击了一个商品目录，我们就需要给它列出对应目录的商品。下面的代码遍历属于对应目录的商品（查询结果在 products 中），给出商品的简介：图片、商品名字和价格。客户点击图片或商品名字都能导航到商品详情。

```
{# 显示商品简介 #}
{% for product in products %}
    <a href="{{ product.get_absolute_url }}">
        <img src="{{ product.image.url }}" >
    </a>
    <div class="caption">
       <h3 class="text-center">
       <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
       </h3>
       <p class="text-center">价格. ￥{{ product.price }}</p>
    </div>
% endfor %}
```

如下图，右边是商品目录，包括所有、华为专卖店、华硕专卖店、联想专卖店和苹果专卖店。客户点击“华为专卖店”，就会显示属于“华为专卖店”的商品。每种商品以一个小宫格显示它的图片、名字和价格。

现在，只要用户点击宫格的图片或商品名字，就会进入商品详情页面。下面的代码用户显示和渲染商品的具体信息，首先是显示商品的图片，然后是商品名字、价格，最后是商品详细的介绍。

```
{# 显示商品详情 #}
<div class="col-md-6 text-right">
    <img src="{{ product.image.url }}">
</div>
<div class="col-md-6" style="padding-left: 20px">
    <h3>{{ product.name }}</h3>
    ...
    <p class="text-muted">价格. ￥{{ product.price }}</p>
</div>
    <div style="text-align: left;">
    <br/>
    <br/>
    <p style="font-size:16px"><b>详情:</b><br/>
        {{ product.description|safe|linebreaksbr }}</p>
</div>
```

显示效果如下图，购物车功能将在下文介绍。

(七)购物车模块

在这一部分，我们希望实现这样的功能：

1) 用户将一件商品放入购物车；

2) 用户将购物车的商品删除；

3) 用户可以查看自己的购物车。

1. 视图设计

A. 商品加入购物车

首先根据商品的 id 去查询数据库中对应的商品，然后将其加入到购物车 Cart 对象中。添加之后，将用户重定位为购物车界面。

```
@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product, quantity=cd['quantity'],
                 update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

B. 删除购物车商品

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

C. 查看购物车

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['form'] = CartAddProductForm(initial={
            'quantity': item['quantity'],
            'update': True
        })
    return render(request, 'cart/detail.html', {'cart': cart})
```

2. HTML 与 CSS 模板设计

下面的代码主要展示了更新商品数量和从购物车删除商品的界面设计。更新操作是一个 post 方法，调用 cart_add，它由一个 submit 类型的按钮发起，按钮可输入 0-quantity 的数量。

删除操作调用 cart_remove，会将商品的 id 一同传递给 cart_remove 函数。

```
{# 显示购物车详情 #}
{% for item in cart %}
  {% with product=item.product %}
      <td>
        <form action="{% url "cart:cart_add" product.id %}" method="post">
          {% csrf_token %}
            {{ item.form.quantity }}    {{ item.form.update }}
            <input type="submit" value="更新" class="btn btn-info">
        </form>
      </td>
```

```
      <td>
        <a href="{% url "cart:cart_remove" product.id %}">删除</a>
      </td>
    </tr>
  {% endwith %}
{% endfor %}
```

下面是在浏览器中看到的购物车界面。用户买了 10 台“华为手机 p20”和 1 台“华为手机畅想 9plus”。每个列表项包含商品图片（可导航到商品）、商品名字、购买数量（旁边的“更新”按钮可改变数量）、单价，然后算出每种商品的总价。右下角的总价是所有商品总价之和。

(八)订单模块

在这一部分，我们需要实现的功能是：用户将结算购物车中的商品，给商家下单。

1. 模型定义

既然要下单，就需要由一个订单模型——Order。Order 包含了一个订单 id 和用户的基本信息（如姓名、地址 address、邮政编码 postal_code 等）。Meta 类的 ordering 表示订单按创建时间倒序排列，即最新的订单会放在前面。

```
class Order(models.Model):
    first_name = models.CharField(max_length=60)
    last_name = models.CharField(max_length=60)
    email = models.EmailField()
    address = models.CharField(max_length=150)
```

```
    postal_code = models.CharField(max_length=30)
    city = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)

    class Meta:
        ordering = ('-created', )
```

此外，我们创建一个 OrderItem 模型，表示订单明细。order 是一个外键引用，表示对应的订单号；product 也是外键引用，表示对应的商品 id；unit_price 是商品的单价，quantity 是购买该商品的数量。

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order, related_name='items',
on_delete=models.CASCADE)
    product = models.ForeignKey(Product, related_name='order_items',
on_delete=models.CASCADE)
    unit_price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)
```

2. 视图设计

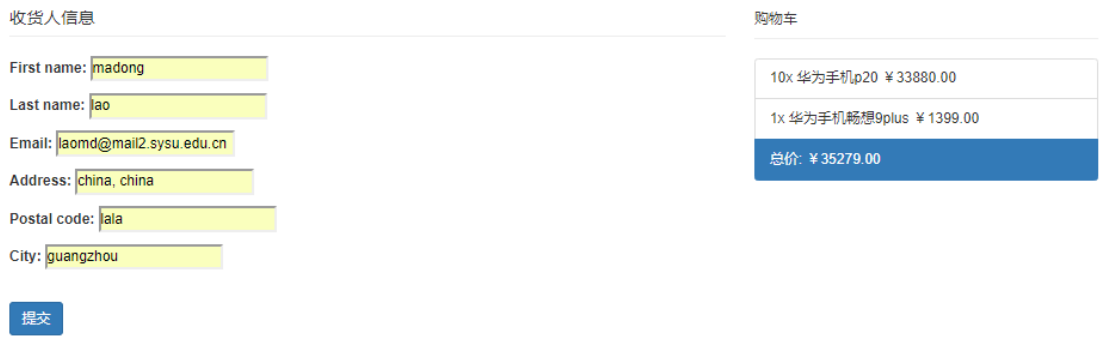
下面的订单创建函数会遍历购物车中的每种商品，单独给每种商品创建一个订单明细，但同一购物车的所有订单明细的订单号相同。

当数据库中商品的剩余数量不够时，OrderItem 的 insert 触发器会抛出异常，因此需要捕捉。但是在抛出异常之前，订单已经生成了，因此需要把对应的订单删除。该订单下已生成的订单明细也会全部被级联地删除。

```
def order_create(request):
    cart = Cart(request)
    form = OrderCreateForm(request.POST)
    try:
        order = form.save()      # 创建对应的订单
        for item in cart:   # 给每种商品创建一个明细
```

```
            OrderItem.objects.create(
                order=order,
                product=item['product'],
                price=item['price'],
                quantity=item['quantity']
            )
            cart.clear()  # 清空购物车
        return render(request, 'orders/created.html', {'order': order})
    except:
        # 数量不够时，OrderItem 的 insert 触发器抛出异常，但订单已经生成了
        # 因此需要删除对应的订单
        old_order = get_object_or_404(Order, id=order.id)
        old_order.delete()
        return HttpResponse('404 剩余数量不够')
```

下图是结算时的界面，左边填写收件人的信息，右边列出购物车里面的东西和总价。客户点击提交后，就会调用上面的 order_create 函数。



四、 实验结果

顾客进入商城：

可以看到商品的基本信息

商家列表中选择其中一个商家，将筛选出该商家的商品：

如选择华为专卖店：

如果想购买华为笔记本电脑 MateBook13 即最后一个商品，点进去：

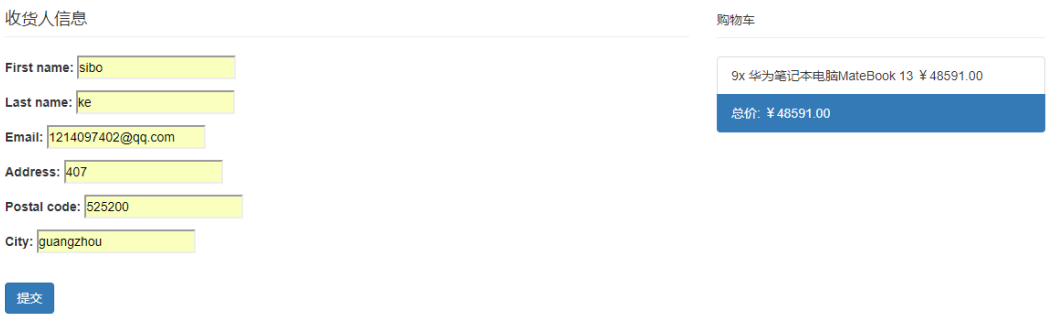
可以看到商品的更详细的信息，如详情

选择数量 9 并添加进购物车：

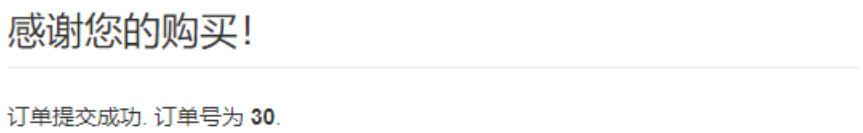
这时右上角会显示购物车的统计信息：

点击结算：

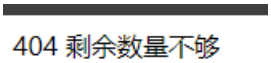
将要求填写个人信息：



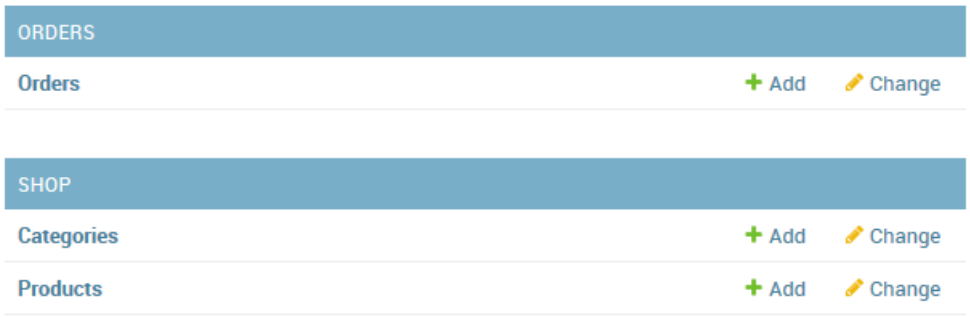
填好后提交即可完成购买：



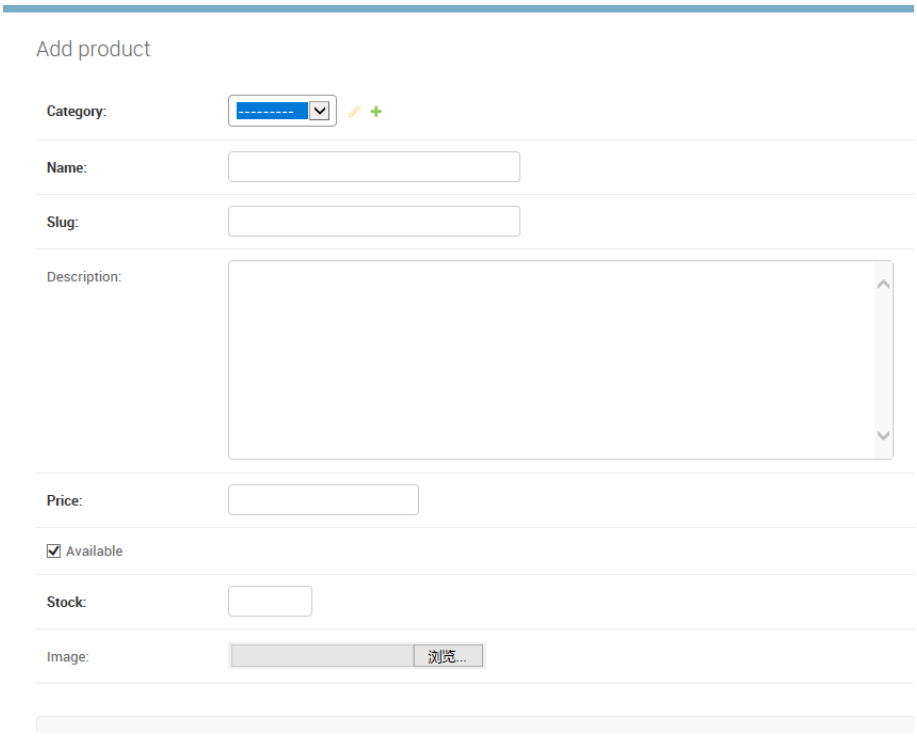
这里验证一下 `insert` 触发器的功能（其他触发器同理就不验证了），如果购买数量超过剩余数量，如总数 `100` 件但购买 `118` 件将出现如下提示：



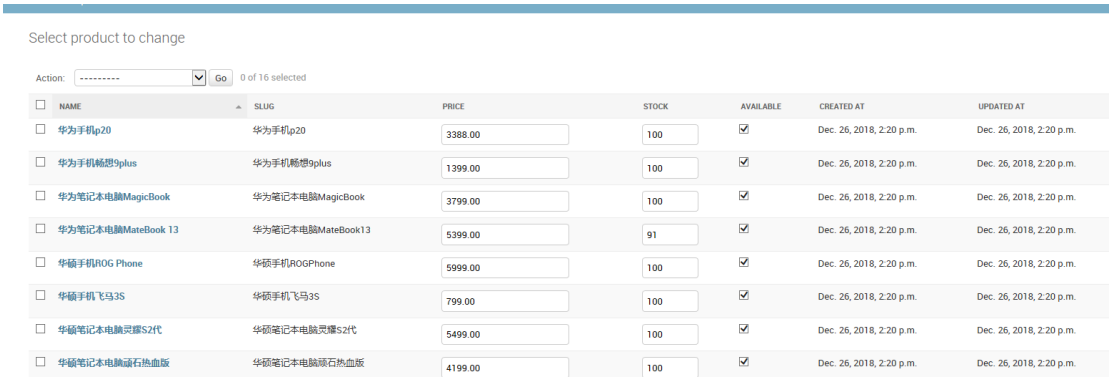
而当我们以商家身份登录进去时，我们可以看到：



点击其中的 products 的 add 是可以进行商品的上架的：



而点击 products 的 change 是可以修改商品信息的：



五、 实验总结

柯司博：

这次 project 做的在线商城涵盖了这学期几乎所有的实验核心内容，不仅复习了以前的知识也学会了应用知识，学会了怎样开发一个简单的数据库系统，从模型设计到整个系统的建立过程并不容易，但如果灵活运用一些最基本的知识如数据库的查询和更新操作等将很快能搭建起一个系统，然后再运用上一些高级的数据库知识如自主存储控制、视图、触发器等来逐步完善整个系统。而且要做一个数据库应用一般都包含前端和后台，前端和后台接口的适配也是需要认真对待的，只有这样整个系统性能才能稳定安全。

劳马东：

这次的数据库项目我们共同设计了系统的概念模型。

概念模型设计的一个重要前提是你要清楚自己的系统具有哪些功能。一个功能的描述通常有主语、宾语和动作，例如，“用户点击商品目录，查看对应商品”，在这句话里面，出现了“商品目录”和“商品”，于是它们就对应了两个实体集（值得一提的是，“用户”也可以做成一个实体机集），“查看对应”一词表示了商品目录和商品之间的包含关系，于是联系集也出来了。

得到了实体集，下一个步骤就是设计它的属性集。我们参考了一些著名网站（如亚马逊、淘宝）上显示的商品属性给商品设计了 name、price、stock、description 等属性，以及“显示”给计算机看的 id、slug 等属性。其余实体集的属性集设计过程同理。

数据库设计完毕后，我开始用 Django 着重开发前端。因为在上学期的《高级程序设计》课程上学习了 Django，也是用它做过一些小应用，因此 web 端开发的整个过程没有遇到很大的麻烦。此外，也得益于 Django 使用 MTV 模式（Model、Template、View）进行开发，最大限度把数据和业务解耦合，我能较快地开发系统的每个模块。例如，在开发商城（shop）模块时，需要给用户提供显示商品列表和商品详情的功能，于是就在 V 层设计了 product_list 和 product_detail 两个函数，它们去操作 M 层的数据库，然后利用 T 层的 HTML 模板文件构建网页并返回给浏览器。在 T 层就可以 DIY 商品列表和商品详情，做到对用户友好。

数据库的一个重要性质是它的一致性。在我们的在线商城系统中，我们经过不断调试发现了一个容易破坏一致性的地方——order 和 orderitem。一个购物车对应一个 order，购物车的每种商品对应一个 orderitem。当用户点击“结算”按钮时，一个

order 项和多个 orderitem 项就会被创建。问题是 orderitem 的插入操作会失败，因为商品的数量可能不够。于是，就会发生这样的情况——用户付了整个购物车的钱，但是发货的时候不发 orderitem 失败的那些商品！

为了解决这个问题，我们最初使用了 python 提供给 MySQL 的事务机制，即 commit 和 rollback。但是这样的代价比较大，因为系统需要定时地建立一些 checkpoint。有没有代价小一点的方法呢？破坏一致性的地方就是购物车的一些商品创建 orderitem 成功了，另一些没有。既然这样，只要购物车里面有商品没能创建 orderitem，那就不应该创建一个订单。这类似于事件原子性的定义——购物车的商品要么全部都创建 orderitem，要么全都不创建 orderitem！

以上就是我在这次期末项目中的工作和心得体会。