

Table of Contents

美多商城	1.1
项目准备	1.2
开发流程介绍	1.2.1
项目介绍	1.2.2
项目需求分析	1.2.2.1
项目架构设计	1.2.2.2
工程创建和配置	1.2.3
创建工程	1.2.3.1
配置开发环境	1.2.3.2
配置Jinja2模板引擎	1.2.3.3
配置MySQL数据库	1.2.3.4
配置Redis数据库	1.2.3.5
配置工程日志	1.2.3.6
配置前端静态文件	1.2.3.7
用户注册	1.3
展示用户注册页面	1.3.1
创建用户模块子应用	1.3.1.1
追加导包路径	1.3.1.2
展示用户注册页面	1.3.1.3
用户模型类	1.3.2
定义用户模型类	1.3.2.1
迁移用户模型类	1.3.2.2
用户注册业务实现	1.3.3
用户注册业务逻辑分析	1.3.3.1
用户注册接口设计和定义	1.3.3.2
用户注册前端逻辑	1.3.3.3
用户注册后端逻辑	1.3.3.4
状态保持	1.3.3.5
用户名重复注册	1.3.3.6
手机号重复注册	1.3.3.7
验证码	1.4
图形验证码	1.4.1
图形验证码逻辑分析	1.4.1.1

图形验证码接口设计和定义	1.4.1.2
图形验证码后端逻辑	1.4.1.3
图形验证码前端逻辑	1.4.1.4
短信验证码	1.4.2
短信验证码逻辑分析	1.4.2.1
容联云通讯短信平台	1.4.2.2
短信验证码后端逻辑	1.4.2.3
短信验证码前端逻辑	1.4.2.4
补充注册时短信验证逻辑	1.4.2.5
避免频繁发送短信验证码	1.4.2.6
pipeline操作Redis数据库	1.4.2.7
异步方案Celery	1.4.3
生产者消费者设计模式	1.4.3.1
Celery介绍和使用	1.4.3.2
用户登录	1.5
账号登录	1.5.1
用户名登录	1.5.1.1
多账号登录	1.5.1.2
首页用户名展示	1.5.1.3
退出登录	1.5.1.4
判断用户是否登录	1.5.1.5
QQ登录	1.5.2
QQ登录开发文档	1.5.2.1
定义QQ登录模型类	1.5.2.2
QQ登录工具QQLoginTool	1.5.2.3
OAuth2.0认证获取openid	1.5.2.4
openid是否绑定用户的处理	1.5.2.5
openid绑定用户实现	1.5.2.6
用户中心	1.6
用户基本信息	1.6.1
用户基本信息逻辑分析	1.6.1.1
查询并渲染用户基本信息	1.6.1.2
添加和验证邮箱	1.6.2
添加邮箱后端逻辑	1.6.2.1
Django发送邮件的配置	1.6.2.2
发送邮箱验证邮件	1.6.2.3

验证邮箱后端逻辑	1.6.2.4
收货地址	1.6.3
省市区三级联动	1.6.3.1
新增地址后端逻辑	1.6.3.2
展示地址后端逻辑	1.6.3.3
修改地址后端逻辑	1.6.3.4
删除地址后端逻辑	1.6.3.5
设置默认地址	1.6.3.6
修改地址标题	1.6.3.7
修改密码	1.6.4
商品	1.7
商品数据库表设计	1.7.1
SPU和SKU	1.7.1.1
首页广告数据库表分析	1.7.1.2
商品信息数据库表分析	1.7.1.3
准备商品数据	1.7.2
文件存储方案FastDFS	1.7.2.1
容器化方案Docker	1.7.2.2
Docker和FastDFS上传和下载文件	1.7.2.3
录入商品数据	1.7.2.4
首页广告	1.7.3
展示首页商品分类	1.7.3.1
展示首页商品广告	1.7.3.2
自定义Django文件存储类	1.7.3.3
商品列表页	1.7.4
商品列表页分析	1.7.4.1
列表页面包屑导航	1.7.4.2
列表页分页和排序	1.7.4.3
列表页热销排行	1.7.4.4
商品搜索	1.7.5
全文检索方案Elasticsearch	1.7.5.1
Haystack扩展建立索引	1.7.5.2
渲染商品搜索结果	1.7.5.3
商品详情页	1.7.6
商品详情页分析和准备	1.7.6.1
展示详情页数据	1.7.6.2

统计分类商品访问量	1.7.6.3
用户浏览记录	1.7.7
设计浏览记录存储方案	1.7.7.1
保存和查询浏览记录	1.7.7.2
购物车	1.8
购物车存储方案	1.8.1
购物车管理	1.8.2
添加购物车	1.8.2.1
展示购物车	1.8.2.2
修改购物车	1.8.2.3
删除购物车	1.8.2.4
全选购物车	1.8.2.5
合并购物车	1.8.2.6
展示商品页面简单购物车	1.8.3
订单	1.9
结算订单	1.9.1
提交订单	1.9.2
创建订单数据库表	1.9.2.1
保存订单基本信息和订单商品信息	1.9.2.2
展示提交订单成功页面	1.9.2.3
使用事务保存订单数据	1.9.2.4
使用乐观锁并发下单	1.9.2.5
我的订单	1.9.3
支付	1.10
支付宝介绍	1.10.1
对接支付宝系统	1.10.2
订单支付功能	1.10.2.1
保存订单支付结果	1.10.2.2
评价订单商品	1.10.3
评价订单商品	1.10.3.1
详情页展示评价信息	1.10.3.2
性能优化	1.11
页面静态化	1.11.1
首页广告页面静态化	1.11.1.1
商品详情页面静态化	1.11.1.2
MySQL读写分离	1.11.2

MySQL主从同步	1.11.2.1
Django实现MySQL读写分离	1.11.2.2

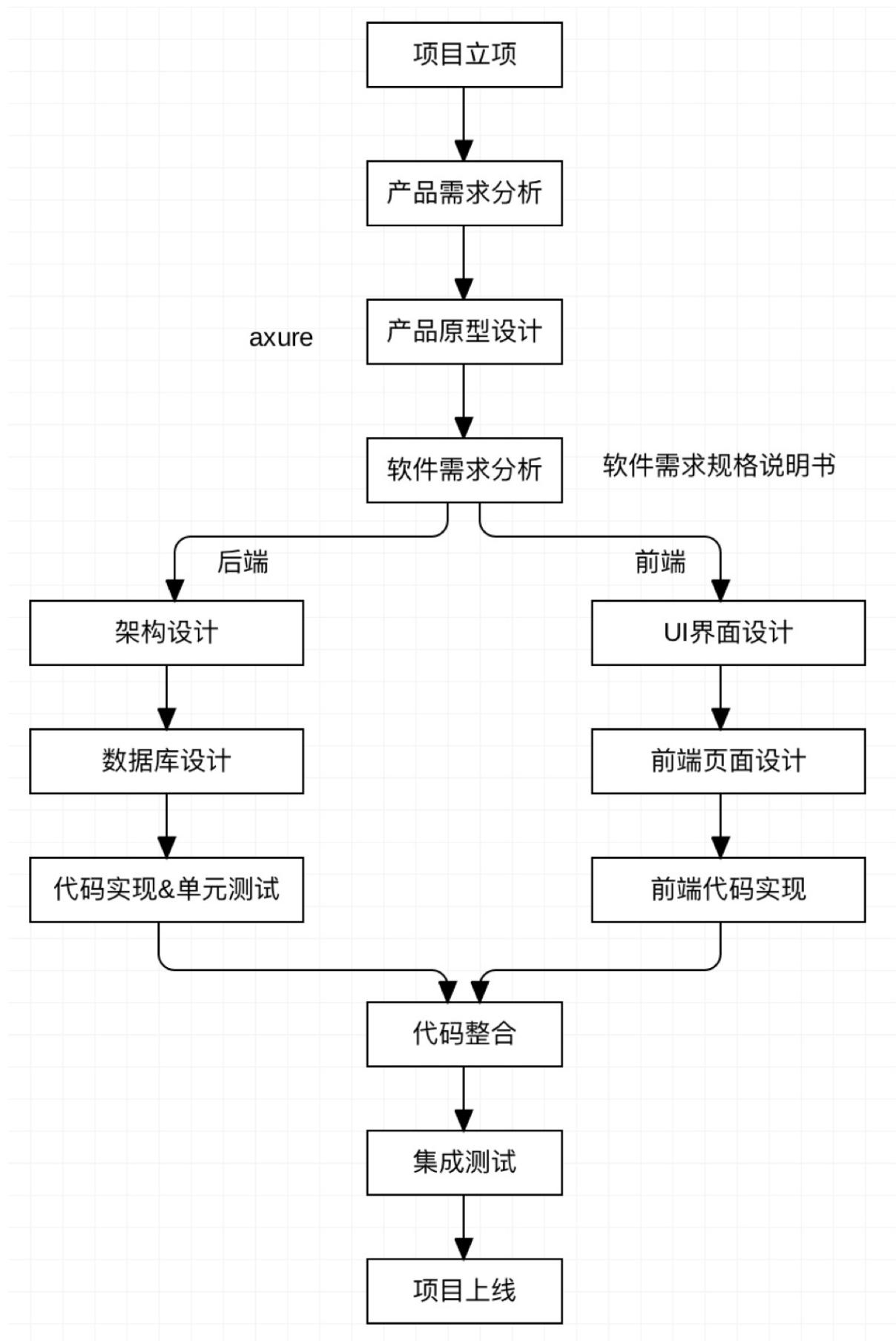
欢迎来到美多商城！

Screenshot of the Meiduo Mall homepage:

- Header:** Shows the URL "美多商城-首页" (Meiduo Mall - Home) and "① 不安全 | www.meiduo.site". It also includes a search bar, user status ("欢迎您: itcast"), and navigation links like "退出" (Logout), "用户中心" (User Center), "我的购物车" (My Shopping Cart), and "我的订单" (My Orders).
- Left Sidebar (商品分类):** Lists categories such as 手机, 相机, 数码; 电脑, 办公, 家用电器; 居家, 家具, 家装, 厨具; 男装, 女装, 童装, 内衣; 女鞋, 箱包, 钟表, 珠宝; 男鞋, 运动, 户外; 房产, 汽车, 汽车用品; 母婴, 玩具乐器; 食品, 酒类, 生鲜, 特产; 图书, 音像, 电子书; 机票, 酒店, 旅游, 生活.
- Center Banner:** Features a large advertisement for kitchen and bathroom products with the text "厨房卫浴品质315 · 满 / 额 / 可 / 省 1000元".
- Right Sidebar (快讯):** Displays news items like "I7顽石低至4199元", "奥克斯专场 正1匹空调1313元抢", etc.
- Bottom Section (1F 手机通讯):** Shows a grid of five smartphone models with their names and prices: 360手机 N6 Pro 全网通 (¥ 2699.00), iPhone X (¥ 7788.00), 荣耀 畅玩7A 全网通 极光蓝 (¥ 749.00), 魅蓝 S6 全网通 (¥ 1199.00), and 红米5Plus 全网通 浅蓝 (¥ 1299.00). There are also tabs for "时尚新品", "畅想低价", and "手机配件".

项目准备

开发流程介绍



说明：

1. 架构设计

- 分析可能用到的技术点
- 前后端是否分离
- 前端使用哪些框架
- 后端使用哪些框架
- 选择什么数据库
- 如何实现缓存
- 是否搭建分布式服务
- 如何管理源代码

2. 数据库设计

- 数据库表的设计至关重要
- 根据项目需求，设计合适的数据库表
- 数据库表在前期如果设计不合理，后期随需求增加会变得难以维护

3. 集成测试

- 在测试阶段要留意测试反馈平台的bug报告

项目介绍

项目需求分析

需求分析原因：

- 可以整体的了解项目的业务流程和主要的业务需求。
- 项目中，需求驱动开发。即开发人员需要以需求为目标来实现业务逻辑。

需求分析方式：

- 企业中，借助 **产品原型图** 分析需求。
- 需求分析完后，前端按照产品原型图开发前端页面，后端开发对应的业务及响应处理。

需求分析内容：

- 页面及其业务流程和业务逻辑。

提示：

- 我们现在借助 **示例网站** 作为原型图来分析需求。

1. 项目主要页面介绍

1. 首页广告

The screenshot shows the homepage of Meiduo Mall. At the top, there is a navigation bar with links for '首页' (Home), '真划算' (Great Deals), and '抽奖' (Lottery). Below the navigation is a search bar with placeholder text '搜索商品' (Search商品) and a red '搜索' (Search) button. To the right of the search bar is a shopping cart icon with a '2' notification. The main content area features a large red promotional banner for '厨房卫浴品质315' (Kitchen and Bath Quality 315) with a '满额可省1000元' (Save up to 1000 yuan with full amount) offer. The banner also highlights '专利金圭特护内胆' (Patented Gold Seal Special Protection Inner Tank). On the left side, there is a sidebar with '商品分类' (Product Categories) including '手机 相机 数码' (Mobile phones, cameras, electronics), '电脑 办公 家用电器' (Computers, office, home appliances), '家居 家具 家装 厨具' (Home, furniture, decoration, kitchenware), and other categories like '男装 女装 童装 内衣' (Men's, women's, children's, undergarments). At the bottom, there is a grid of mobile phones with their names and prices: '荣耀V10' (Huawei Honor V10) with a '优惠200' (Discount 200) offer, '360手机N6 Pro' (360 Phone N6 Pro) at 2699.00, 'iPhone X' at 7788.00, '荣耀畅玩7A' (Honor Play 7A) at 749.00, '魅蓝S6' (Meizu M6 Note) at 1199.00, and '红米5Plus' (Xiaomi Redmi 5 Plus) at 1299.00.

2. 注册

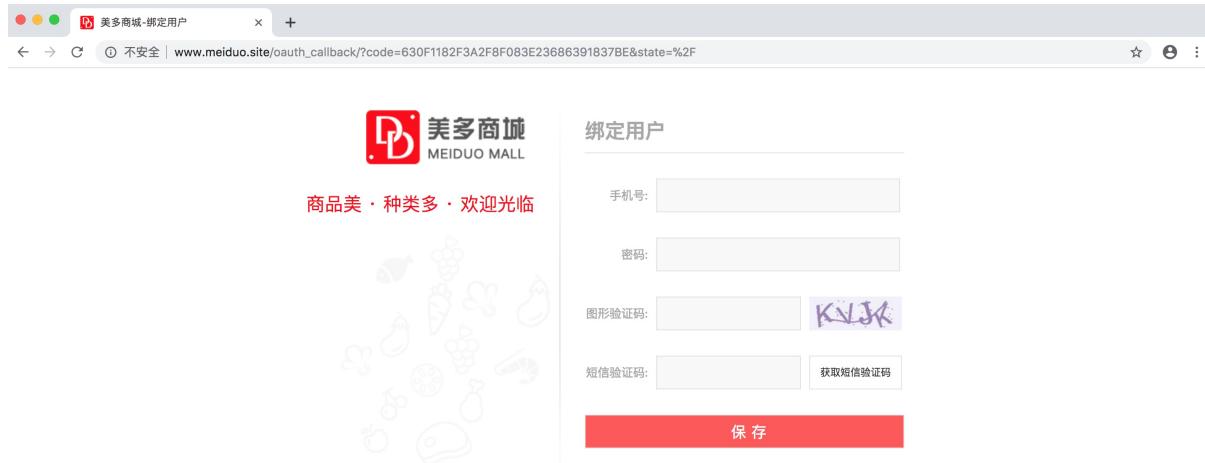
The screenshot shows the user registration interface for Meiduo Mall. At the top, there's a header bar with the title '美多商城-注册' and a URL 'www.meiduo.site/register/'. Below the header is the Meiduo Mall logo and a slogan '商品美·种类多·欢迎光临'. To the right of the logo is a '用户注册' (User Registration) section. It includes fields for '用户名' (Username), '密码' (Password), '确认密码' (Confirm Password), '手机号' (Mobile Number), '图形验证码' (Captcha), and '短信验证码' (SMS Verification Code). There's also a checkbox for accepting the user agreement and a red '注册' (Register) button at the bottom.

3. 登录



4.QQ登录





5.个人信息

This screenshot shows the 'User Center' page of the Meiduo Mall website. The header includes the title 'Meiduo Mall - User Center' and a welcome message 'Welcome to Meiduo Mall!'. On the left, there's a sidebar with links for '个人信息' (Personal Information), '收货地址' (Shipping Address), '全部订单' (All Orders), and '修改密码' (Change Password). The main content area has a search bar with placeholder text 'Search products' and a 'Search' button. Below the search bar, there's a promotional banner for 'Sony Weidian' with a discount of 15 yuan. The 'Basic Information' section displays the user's name 'itcast', contact number '17600992166', and an email input field with a 'Save' and 'Cancel' button. Under the 'Recent Browsing' heading, five product thumbnails are shown: Apple iPhone 8 Plus (A186...), Apple iPhone 8 Plus (A186...), Apple MacBook Pro 13.3..., Apple iPhone 8 Plus (A186...), and Apple iPhone 8 Plus (A186...). Each item has its price (e.g., ¥6499.00) and a 'View Details' button.

6.收货地址

The screenshot shows the Meiduo Mall user center interface. At the top, there's a navigation bar with links like '搜索商品' (Search Product), '搜索' (Search), '搜索结果' (Search Results), '我的购物车' (My Shopping Cart), and '我的订单' (My Orders). Below the navigation is a search bar with placeholder text '索尼微单 优惠15元 美妆个护 买2免1'. The main content area is titled '新增收货地址' (Add New Shipping Address) and displays two address entries:

- 传智播客** (Default Address)
收货人: itcast
所在地区: 北京市 北京市 昌平区
地址: 建材城西路
手机: 17600992166
固定电话:
电子邮箱:
操作按钮: 编辑
- 张小厨**
收货人: 张小厨
所在地区: 北京市 北京市 昌平区
地址: 建材城西路
手机: 17600992166
固定电话:
电子邮箱:
操作按钮: 设为默认 编辑

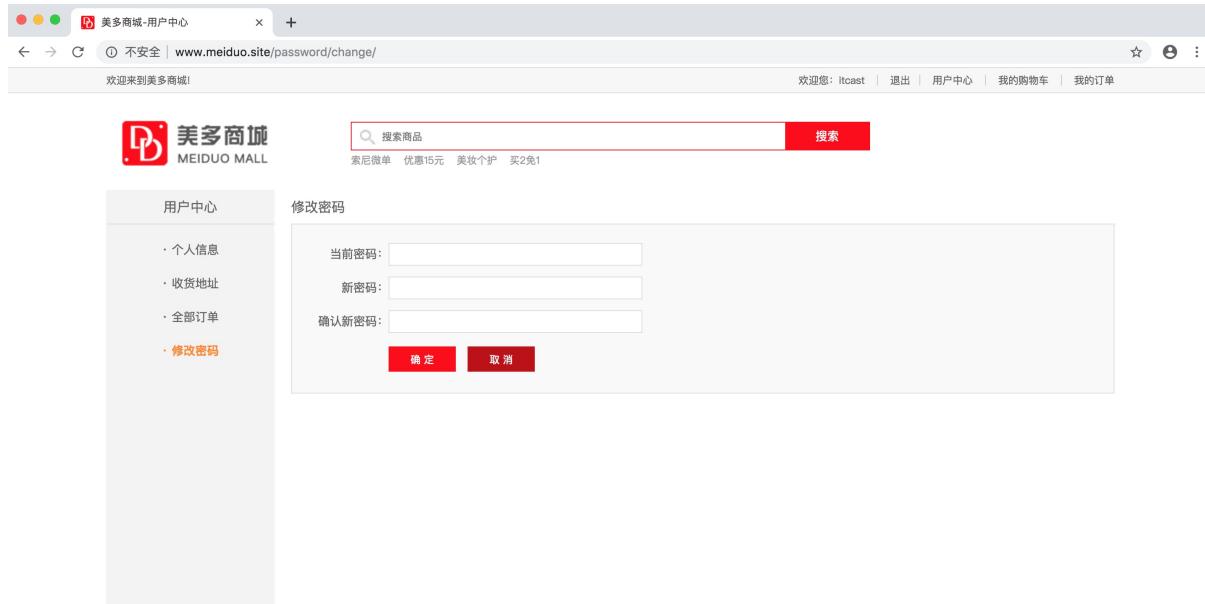
7.我的订单

The screenshot shows the Meiduo Mall user center interface for managing orders. At the top, there's a navigation bar with links like '搜索商品' (Search Product), '搜索' (Search), '搜索结果' (Search Results), '我的购物车' (My Shopping Cart), and '我的订单' (My Orders). Below the navigation is a search bar with placeholder text '索尼微单 优惠15元 美妆个护 买2免1'. The main content area is titled '全部订单' (All Orders) and displays two order lists:

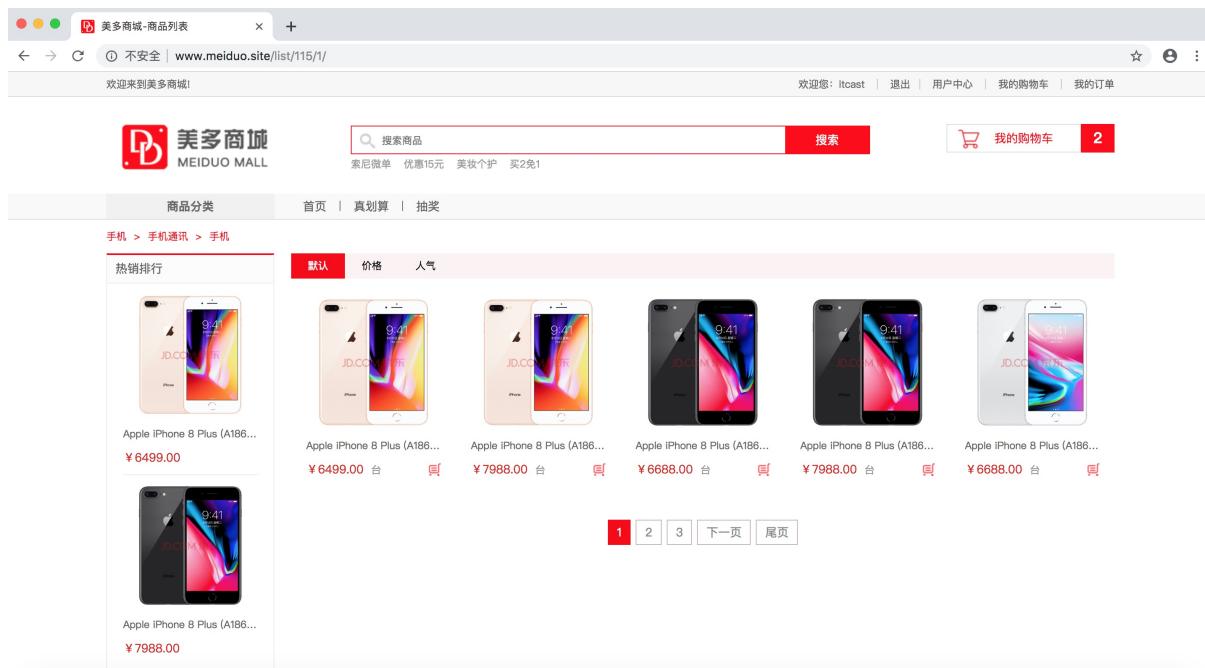
订单号	下单时间	商品信息	单价	数量	总价	运费	支付方式	状态
0746000000000005	07:46:10	Apple MacBook Pro 13.3英寸笔...	11388.00元	1	11388.00元	17897.00元 含运费: 10.00元	支付宝	待支付
0730500000000005	07:31:00	Apple iPhone 8 Plus (A1864) 64...	6499.00元	1	6499.00元	17897.00元 含运费: 10.00元	支付宝	待评价

Page navigation buttons: 1 2 3 4 5 下一页 尾页

8.修改密码



9.商品列表



10.商品搜索

项目需求分析

This screenshot shows the search results for 'iphone' on the Meiduo Mall website. It displays five iPhone 8 Plus products, each with a thumbnail, name, price, and rating. The first product is an iPhone 8 Plus (A1864) in gold with a price of ¥6499.00 and 2 reviews. The second is an iPhone 8 Plus (A1864) in gold with a price of ¥7988.00 and 0 reviews. The third is an iPhone 8 Plus (A1864) in silver with a price of ¥7988.00 and 0 reviews. The fourth is an iPhone 8 Plus (A1864) in silver with a price of ¥6688.00 and 0 reviews. The fifth is an iPhone 8 Plus (A1864) in black with a price of ¥7988.00 and 1 review. Below the products is a navigation bar with page numbers 1, 2, 下一页 (Next Page), and 尾页 (Last Page).

11.商品详情

This screenshot shows the detailed product page for an iPhone 8 Plus (A1864) in gold. The product image is displayed prominently. The title is 'Apple iPhone 8 Plus (A1864) 64GB 金色 移动联通电信4G手机'. A note below the title says '选【移动优惠购】新机配新卡，198优质靓号，流量不限量！'. The price is listed as ¥6499.00 with a red '1人评价' (1 review) badge. Below the price are dropdown menus for quantity (1), color (金色 - selected, 深空灰, 银色), and storage (64GB - selected, 256GB). The total price is shown as '总价: 6499.00元'. A large red '加入购物车' (Add to Cart) button is at the bottom. At the very bottom of the page, there are tabs for '热销排行', '商品详情' (selected), '规格与包装', '售后服务', and '商品评价(1)'.

12.购物车

This screenshot shows the shopping cart page with two items: an Apple MacBook Pro 13.3英寸笔记本 (银色) and an iPhone 8 Plus (A1864) 64GB 金色. The total price is listed as '合计(不含运费): ¥ 17887.00' and '共计 2 件商品'. A large red '去结算' (Go to Checkout) button is at the bottom right. The cart table includes columns for 商品名称 (Product Name), 商品价格 (Product Price), 数量 (Quantity), 小计 (Subtotal), and 操作 (Operations).

13. 结算订单

确认收货地址

寄送到:

- 北京市 北京市 昌平区 (Itcast 收) 17600992166
- 北京市 北京市 昌平区 (张小厨 收) 17600992166

编辑收货地址

支付方式

- 货到付款
- 支付宝

商品列表

	商品名称	商品单位	商品价格	数量	小计
1	Apple MacBook Pro 13.3英寸笔记本 银色	台	11388.00元	1	11388.00元
2	Apple iPhone 8 Plus (A1864) 64GB 金色 移动联...	台	6499.00元	1	6499.00元

总金额结算

共 2 件商品, 总金额 17887.00元
运费: 10.00元
实付款: 17887.00元

提交订单

14. 提交订单

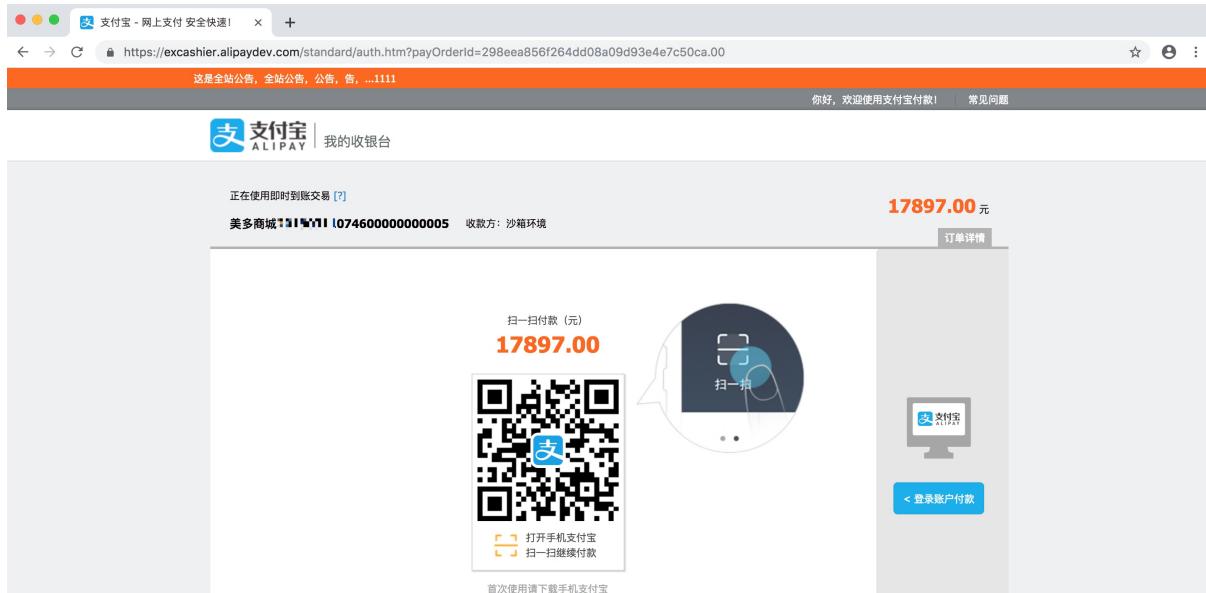
订单提交成功, 订单总价¥17897.00

您的订单已成功生成, 选择您想要的支付方式, 订单号: 07460000000005

您可以在【用户中心】->【我的订单】查看该订单

去支付

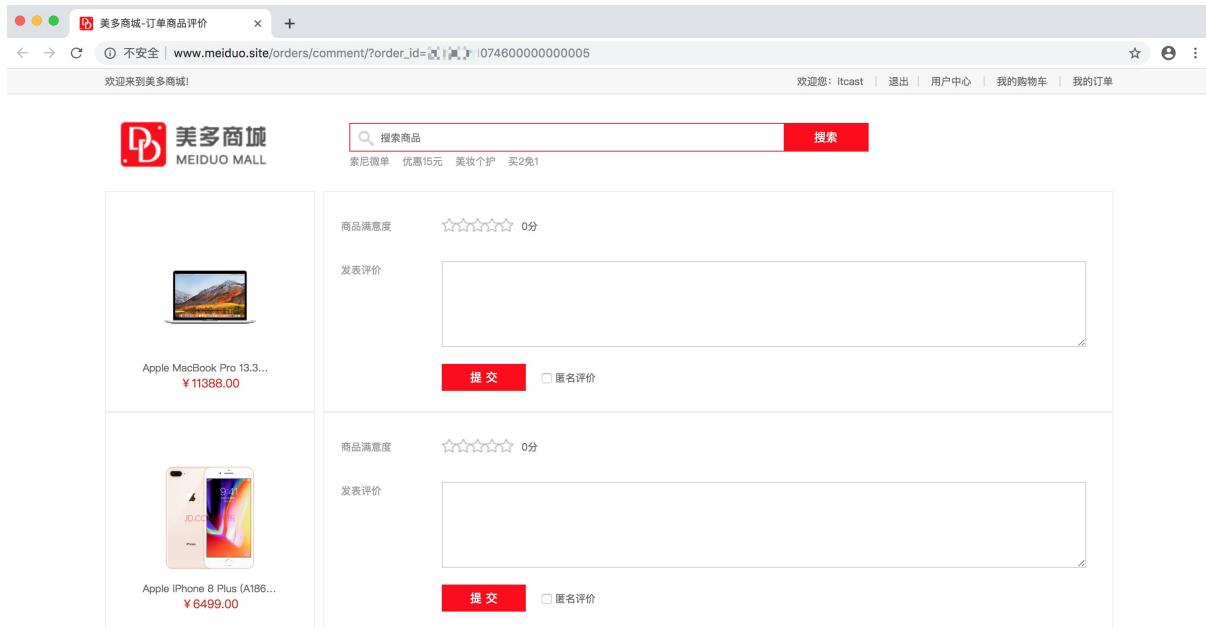
15. 支付宝支付



16.支付结果处理



17.订单商品评价



2. 归纳项目主要模块

为了方便项目管理及多人协同开发，我们根据需求将功能划分为不同的模块。

将来在项目中，每个模块都会对应一个子应用进行管理和解耦。

模块	功能
用户	注册、登录、用户中心
验证	图形验证、短信验证
第三方登录	QQ登录
首页广告	首页广告
商品	商品列表、商品搜索、商品详情
购物车	购物车管理、购物车合并
订单	确认订单、提交订单
支付	支付宝支付、订单商品评价
MIS系统	数据统计、用户管理、权限管理、商品管理、订单管理

3. 知识要点

1. 需求分析原因：需求驱动开发。
2. 需求分析方式：企业中，使用产品原型图。
3. 需求分析内容：页面及业务逻辑。
4. 需求分析结果：划分业务模块，明确每个模块下的主要功能，并以子应用的形式进行管理。

项目架构设计

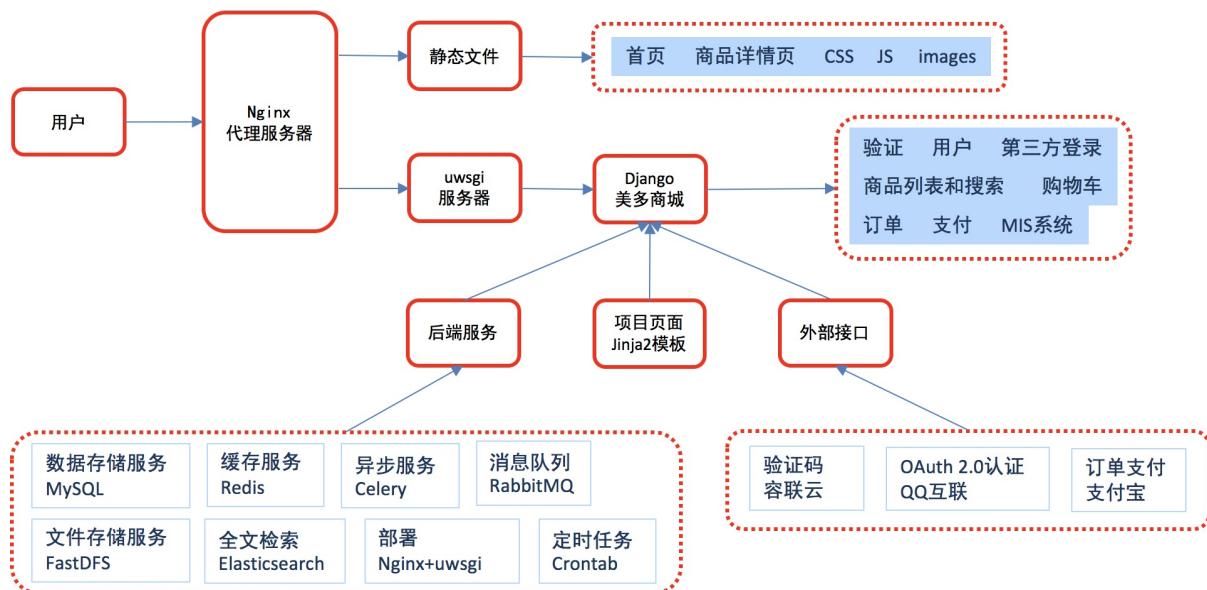
1. 项目开发模式

选项	技术选型
开发模式	前后端不分离
后端框架	Django + Jinja2模板引擎
前端框架	Vue.js

说明：

- 前后端不分离的开发模式，是为了提高搜索引擎排名，即**SEO**。特别是首页，详情页和列表页。
 - 搜索引擎爬虫请求到的页面数据就是渲染好的完整页面，搜索引擎可以直接解析建立索引。
- 页面需要整体刷新：我们会选择使用Jinja2模板引擎来实现。
- 页面需要局部刷新：我们会选择使用Vue.js来实现。

2. 项目运行机制



3. 知识要点

1. 项目开发模式
 - 前后端不分离，方便**SEO**。
 - 采用Django + Jinja2模板引擎 + Vue.js实现前后端逻辑。
2. 项目运行机制
 - 代理服务：Nginx服务器（反向代理）

- 静态服务：Nginx服务器（静态首页、商品详情页、...）
- 动态服务：uwsgi服务器（美多商场业务场景）
- 后端服务：MySQL、Redis、Celery、RabbitMQ、Docker、FastDFS、Elasticsearch、Crontab
- 外部接口：容联云、QQ互联、支付宝

工程创建和配置

创建工程

美多商城项目源代码采用 **远程仓库托管**。

1. 准备项目代码仓库

1. 源码托管网站

- 码云 (<https://gitee.com/>)

2. 创建源码远程仓库: meiduo_project

仓库名称 meiduo_project

归属 ITManager

路径 https://gitee.com/zjsharp/ meiduo_project

仓库介绍 非必填
传智播客

是否开源 公开 私有

选择语言 Python 添加 .gitignore Python 添加开源许可证 MIT License

使用Readme文件初始化这个仓库
 使用Issue模板文件初始化这个仓库
 使用Pull Request模板文件初始化这个仓库
 导入已有仓库

创建

2. 克隆项目代码仓库

1. 进入本地项目目录

```
$ mkdir ~/projects
$ cd projects/
```

2. 克隆仓库

```
$ git clone https://gitee.com/zjsharp/meiduo_project.git
```

3. 忽略 .idea



3. 创建美多商城工程

1.进入本地项目仓库

```
$ cd ~/projects/meiduo_project/
```

2.创建美多商城虚拟环境，安装Django框架

```
$ mkvirtualenv -p python3 meiduo_mall  
$ pip install django==1.11.11
```

3.创建美多商城Django工程

```
$ django-admin startproject meiduo_mall
```

创建工作完成后：运行程序，测试结果。

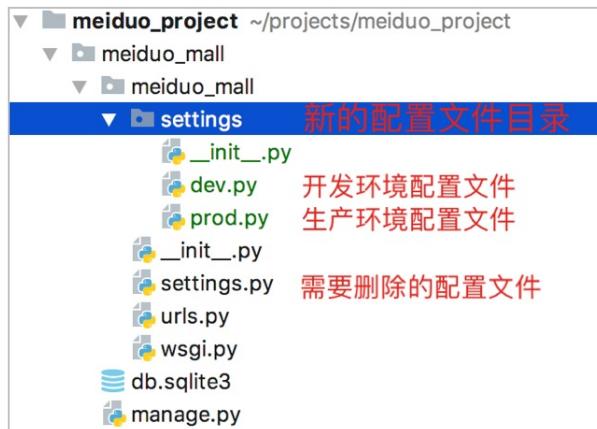
配置开发环境

美多商城项目的环境分为 `开发环境` 和 `生产环境`。

- 开发环境：用于编写和调试项目代码。
- 生产环境：用于项目线上部署运行。

1. 新建配置文件

1. 准备配置文件目录
 - 新建包，命名为`settings`，作为配置文件目录
2. 准备开发和生产环境配置文件
 - 在配置包`settings`中，新建开发和生产环境配置文件
3. 准备开发环境配置内容
 - 将默认的配置文件`settings.py`中内容拷贝至`dev.py`



2. 指定开发环境配置文件

```
manage.py
1 #!/usr/bin/env python
2 import os
3 import sys
4
5 if __name__ == "__main__":
6     # os.environ.setdefault("DJANGO_SETTINGS_MODULE", "meiduo_mall.settings") # 默认的开发环境配置文件
7     os.environ.setdefault("DJANGO_SETTINGS_MODULE", "meiduo_mall.settings.dev") # 新的开发环境配置文件
8     try:
9         from django.core.management import execute_from_command_line
10    except ImportError:
11        # The above import may fail for some other reason. Ensure that the
12        # issue is really that Django is missing to avoid masking other
13        # exceptions on Python 2.
14        try:
15            import django
16        except ImportError:
17            raise ImportError(
18                "Module import error: Are you sure it's installed and "
19                "'available on your PYTHONPATH environment variable? Did you '"
20                "'forget to activate a virtual environment?'"
21            )
22    raise
23 execute_from_command_line(sys.argv)
```

配置完成后：运行程序，测试结果。

配置Jinja2模板引擎

美多商城的模板采用 `Jinja2模板引擎`。

1. 安装Jinja2扩展包

```
$ pip install Jinja2
```

2. 配置Jinja2模板引擎

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2', # jinja2模板引擎
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

3. 补充Jinja2模板引擎环境

确保可以使用模板引擎中的`{{ static('') }}`、`{{ url('') }}`这类语句

```
<link rel="stylesheet" type="text/css" href="{{ static('css/reset.css') }}>
<link rel="stylesheet" type="text/css" href="{{ static('css/main.css') }}>
<script type="text/javascript" src="{{ static('js/vue-2.5.16.js') }}></script>
<script type="text/javascript" src="{{ static('js/axios-0.18.0.min.js') }}></script>

<a href="{{ url('users:login') }}">登录</a>
<span>|</span>
<a href="{{ url('users:register') }}">注册</a>
```

1. 创建Jinja2模板引擎环境配置文件



2. 编写Jinja2模板引擎环境配置代码

```
from jinja2 import Environment
from django.contrib.staticfiles.storage import staticfiles_storage
from django.urls import reverse

def jinja2_environment(**options):
    env = Environment(**options)
    env.globals.update({
        'static': staticfiles_storage.url,
        'url': reverse,
    })
    return env
```

确保可以使用模板引擎中的{{ static('') }}、{{ url('') }}这类语句

3. 加载Jinja2模板引擎环境

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.jinja2.Jinja2', # jinja2模板引擎
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
        # 补充Jinja2模板引擎环境
        'environment': 'meiduo_mall.utils.jinja2_env.jinja2_environment',
    },
},
```

]

配置完成后：运行程序，测试结果。

配置MySQL数据库

美多商城数据存储服务采用 MySQL数据库。

1. 新建MySQL数据库

1.新建MySQL数据库: meiduo_mall

```
$ create database meiduo charset=utf8;
```

2.新建MySQL用户

```
$ create user itheima identified by '123456';
```

3.授权 itcast 用户访问 meiduo_mall 数据库

```
$ grant all on meiduo.* to 'itheima'@'%';
```

4.授权结束后刷新特权

```
$ flush privileges;
```

2. 配置MySQL数据库

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # 数据库引擎
        'HOST': '127.0.0.1', # 数据库主机
        'PORT': 3306, # 数据库端口
        'USER': 'itheima', # 数据库用户名
        'PASSWORD': '123456', # 数据库用户密码
        'NAME': 'meiduo' # 数据库名字
    },
}
```

可能出现的错误

- Error loading MySQLdb module: No module named 'MySQLdb'.

出现错误的原因:

- Django中操作MySQL数据库需要驱动程序MySQLdb
- 目前项目虚拟环境中没有驱动程序MySQLdb

解决办法：

- 安装PyMySQL扩展包
- 因为MySQLdb只适用于Python2.x的版本，Python3.x的版本中使用PyMySQL替代MySQLdb

3. 安装PyMySQL扩展包

1. 安装驱动程序

```
$ pip install PyMySQL
```

2. 在工程同名子目录的 `__init__.py` 文件中，添加如下代码：

```
from pymysql import install_as_MySQLdb

install_as_MySQLdb()
```

配置完成后：运行程序，测试结果。

配置Redis数据库

美多商城数据缓存服务采用 Redis数据库。

1. 安装django-redis扩展包

1.安装django-redis扩展包

```
$ pip install django-redis
```

2.django-redis使用说明文档

[点击进入文档](#)

2. 配置Redis数据库

```
CACHES = {
    "default": { # 默认
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/0",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    },
    "session": { # session
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    },
}
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "session"
```

default:

- 默认的Redis配置项，采用0号Redis库。

session:

- 状态保持的Redis配置项，采用1号Redis库。

SESSION_ENGINE

- 修改 session存储机制 使用Redis保存。

SESSION_CACHE_ALIAS:

- 使用名为"session"的Redis配置项存储 session数据。

配置完成后：运行程序，测试结果。

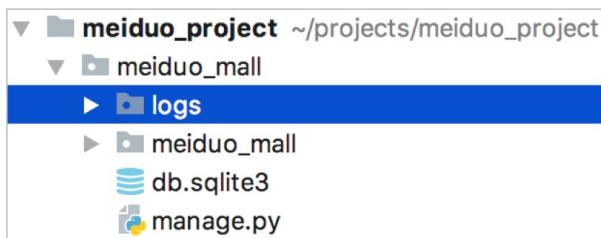
配置工程日志

美多商城的日志记录采用 logging模块。

1. 配置工程日志

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False, # 是否禁用已经存在的日志器
    'formatters': { # 日志信息显示的格式
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(lineno)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(module)s %(lineno)d %(message)s'
        },
    },
    'filters': { # 对日志进行过滤
        'require_debug_true': { # django在debug模式下才输出日志
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'handlers': { # 日志处理方法
        'console': { # 向终端中输出日志
            'level': 'INFO',
            'filters': ['require_debug_true'],
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'file': { # 向文件中输出日志
            'level': 'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': os.path.join(os.path.dirname(BASE_DIR), 'logs/meiduo.log'),
            # 日志文件的位置
            'maxBytes': 300 * 1024 * 1024,
            'backupCount': 10,
            'formatter': 'verbose'
        },
    },
    'loggers': { # 日志器
        'django': { # 定义了一个名为django的日志器
            'handlers': ['console', 'file'], # 可以同时向终端与文件中输出日志
            'propagate': True, # 是否继续传递日志信息
            'level': 'INFO', # 日志器接收的最低日志级别
        },
    }
}
```

2. 准备日志文件目录



3. 日志记录器的使用

```

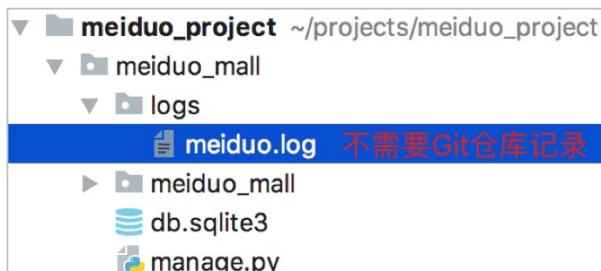
import logging

# 创建日志记录器
logger = logging.getLogger('django')
# 输出日志
logger.debug('测试logging模块debug')
logger.info('测试logging模块info')
logger.error('测试logging模块error')
  
```

4. Git管理工程日志

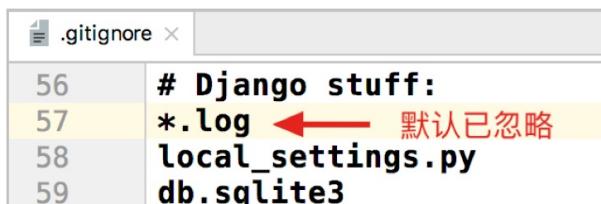
提示1：

- 开发过程中，产生的日志信息不需要代码仓库进行管理和记录。



提示2：

- 建立代码仓库时，生成的忽略文件中已经默认忽略掉了 *.log 。

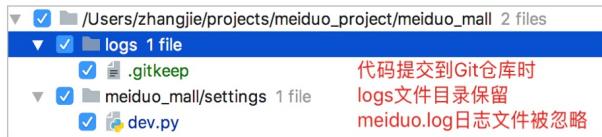


问题：

- logs文件目录需求被Git仓库记录和管理。
- 当把 *.log 都忽略掉后，logs文件目录为空。
- 但是，Git是不允许提交一个空的目录到版本库上的。

解决：

- 在空文件目录中建立一个 `.gitkeep` 文件，然后即可提交。



配置完成后：运行程序，测试结果。

5. 知识要点

1. 本项目最低日志等级设置为：**INFO**

2. 创建日志记录器的方式：

```
logger = logging.getLogger('django')
```

3. 日志记录器的使用：

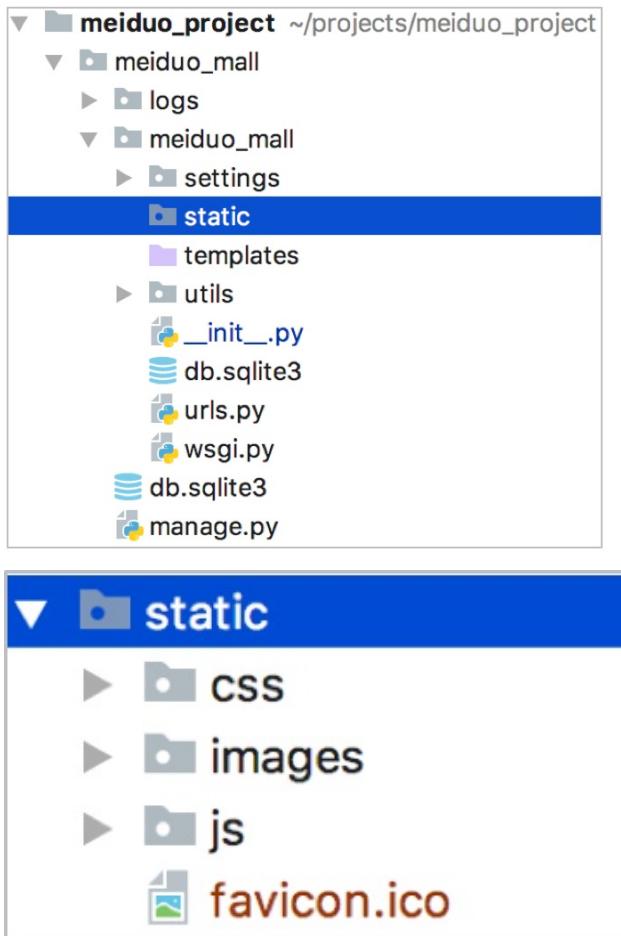
```
logger.info('测试logging模块info')
```

4. 在日志 `loggers` 选项中可以指定多个日志记录器

配置前端静态文件

美多商城项目中需要使用静态文件，比如 css、images、js 等等。

1. 准备静态文件



2. 指定静态文件加载路径

```
STATIC_URL = '/static/'

# 配置静态文件加载路径
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

配置完成后：运行程序，测试结果。

- <http://127.0.0.1:8000/static/images/adv01.jpg>

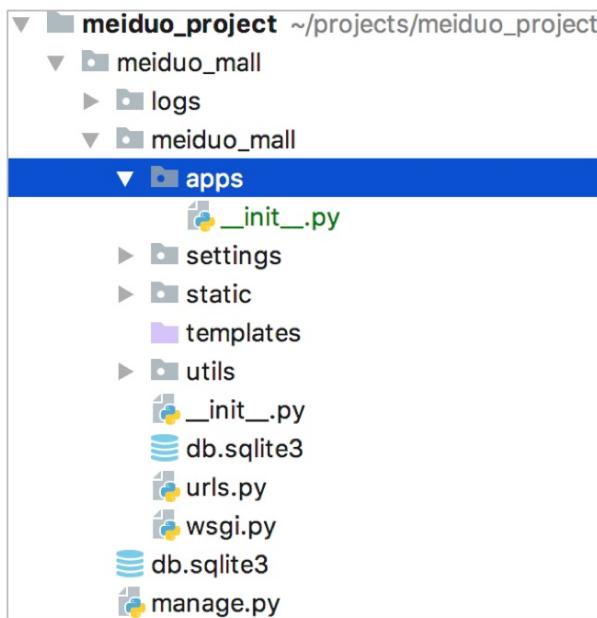
用户注册

展示用户注册页面

创建用户模块子应用

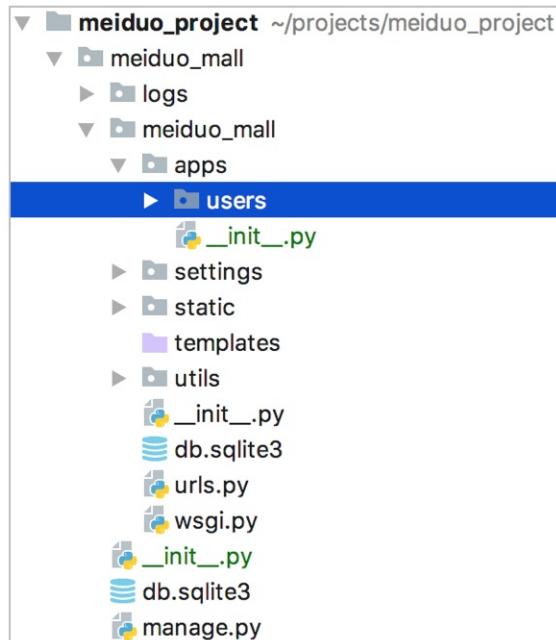
1. 创建用户模块子应用

1. 准备 apps 包，用于管理所有应用



2. 在 apps 包下创建应用 users

```
$ cd ~/projects/meiduo_project/meiduo_mall/meiduo_mall/apps  
$ python ../../manage.py startapp users
```



2. 查看项目导包路径

重要提示：

- 若要知道如何导入users应用并完成注册，需要知道**项目导包路径**



```

13 import os, sys
14
15
16 # 查看项目导包路径
17 print(sys.path)
18 [
19     '~/projects/meiduo_project/meiduo_mall',
20     '~/projects/meiduo_project',
21     './virtualenvs/meiduo_mall/lib/python3.6/site-packages',
22     '....'
23 ]
24
25

```

已知导包路径

- `meiduo_project/meiduo_mall`

已知 'users' 应用所在目录

- `meiduo_project/meiduo_mall/meiduo_mall/apps/users`

得到导入'users'应用的导包路径是：`meiduo_mall/apps/users`

3. 注册用户模块子应用

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'meiduo_mall.apps.users', # 用户模块应用
]

```

注册完users应用后，运行测试程序。

追加导包路径

思考：

- 是否可以将注册users应用做的更加简便？
- 按照如下形式，直接以应用名users注册

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users', # 用户模块应用
]
```

分析：

- 已知导包路径
 - meiduo_project/meiduo_mall
- 已知'users'应用所在目录
 - meiduo_project/meiduo_mall/meiduo_mall/apps/users
- 若要直接以应用名'users'注册
 - 需要一个导包路径： meiduo_project/meiduo_mall/meiduo_mall/apps

解决办法

* 追加导包路径：`meiduo_project/meiduo_mall/meiduo_mall/apps`

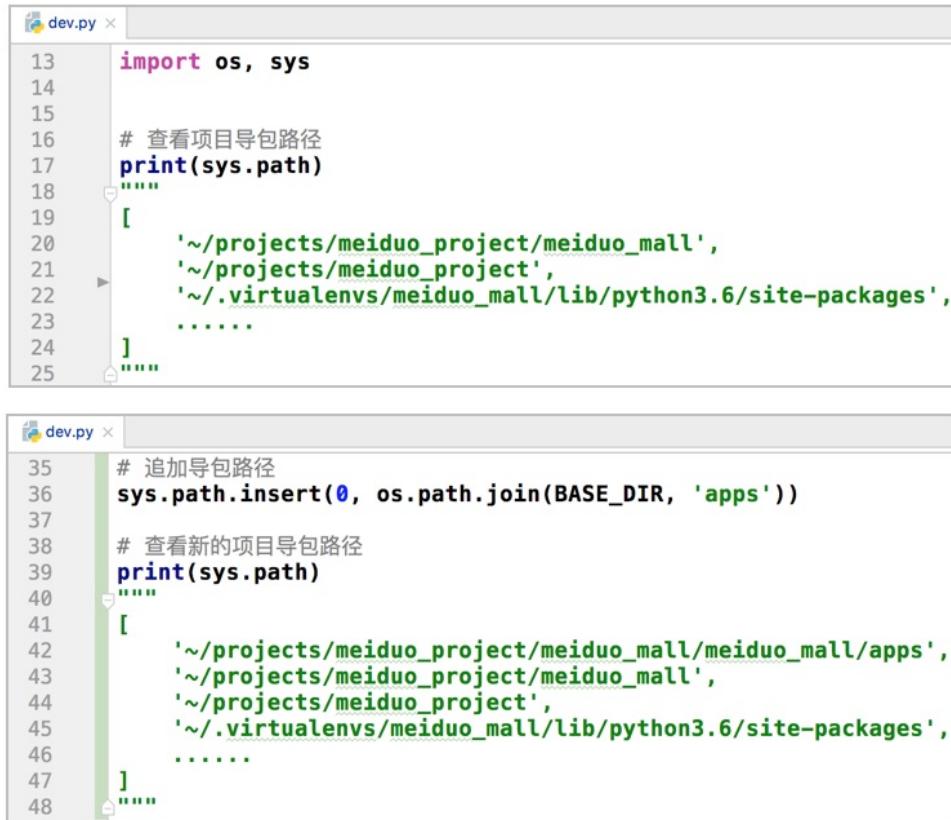
1. 追加导包路径

1. 查看项目BASE_DIR



```
dev.py <
29 # Build paths inside the project like this: os.path.join(BASE_DIR, ...)
30 BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
31 print(BASE_DIR)
32 """
33 ~'/projects/meiduo_project/meiduo_mall/meiduo_mall
34 """
```

2. 追加导包路径



```

dev.py
13 import os, sys
14
15
16 # 查看项目导包路径
17 print(sys.path)
18 """
19 [
20     '~/projects/meiduo_project/meiduo_mall',
21     '~/projects/meiduo_project',
22     '~/virtualenvs/meiduo_mall/lib/python3.6/site-packages',
23     .....
24 ]
25 """

```



```

dev.py
35 # 追加导包路径
36 sys.path.insert(0, os.path.join(BASE_DIR, 'apps'))
37
38 # 查看新的项目导包路径
39 print(sys.path)
40 """
41 [
42     '~/projects/meiduo_project/meiduo_mall/meiduo_mall/apps',
43     '~/projects/meiduo_project/meiduo_mall',
44     '~/projects/meiduo_project',
45     '~/virtualenvs/meiduo_mall/lib/python3.6/site-packages',
46     .....
47 ]
48 """

```

2. 重新注册用户模块应用

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users', # 用户模块应用
]

```

重新注册完users应用后，运行测试程序。

3. 知识要点

1. 查看导包路径
 - 通过查看导包路径，可以快速的知道项目中各个包该如何的导入。
 - 特别是接手老项目时，可以尽快的适应项目导包的方式。
2. 追加导包路径
 - 通过追加导包路径，可以简化某些目录复杂的导包方式。

展示用户注册页面

1. 准备用户注册模板文件



加载页面静态文件

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>美多商城-注册</title>
    <link rel="stylesheet" type="text/css" href="{{ static('css/reset.css') }}">
    <link rel="stylesheet" type="text/css" href="{{ static('css/main.css') }}">
</head>
```

2. 定义用户注册视图

```
class RegisterView(View):
    """用户注册"""

    def get(self, request):
        """
        提供注册页面
        :param request: 请求对象
        :return: 注册页面
        """
        return render(request, 'register.html')
```

3. 定义用户注册路由

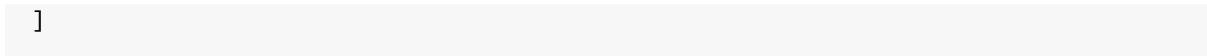
1. 总路由

```
urlpatterns = [
    # users
    url(r'^', include('users.urls', namespace='users')),
]
```

2. 子路由

```
urlpatterns = [
    # 注册
    url(r'^register/$', views.RegisterView.as_view(), name='register'),
```

]



用户模型类

定义用户模型类

1. Django默认用户认证系统

- Django自带用户认证系统
 - 它处理用户账号、组、权限以及基于cookie的用户会话。
- Django认证系统位置
 - `django.contrib.auth` 包含认证框架的核心和默认的模型。
 - `django.contrib.contenttypes` 是Django内容类型系统，它允许权限与你创建的模型关联。
- Django认证系统同时处理认证和授权
 - 认证：验证一个用户是否它声称的那个人，可用于账号登录。
 - 授权：授权决定一个通过了认证的用户被允许做什么。
- Django认证系统包含的内容
 - 用户：**用户模型类**、用户认证。
 - 权限：标识一个用户是否可以做一个特定的任务，MIS系统常用到。
 - 组：对多个具有相同权限的用户进行统一管理，MIS系统常用到。
 - 密码：一个可配置的密码哈希系统，设置密码、密码校验。

2. Django默认用户模型类

- Django认证系统中提供了用户模型类User保存用户的数据。
 - **User对象是认证系统的中心。**
- Django认证系统用户模型类位置
 - `django.contrib.auth.models.User`

```
class User(AbstractUser):
    """
    Users within the Django authentication system are represented by this
    model.

    Username, password and email are required. Other fields are optional.

    Meta:
        swappable = 'AUTH_USER_MODEL'
```

- 父类**AbstractUser**介绍

- User对象基本属性
 - 创建用户(注册用户)必选：`username`、`password`
 - 创建用户(注册用户)可选：`email`、`first_name`、`last_name`、`last_login`、`date_joined`、`is_active`、`is_staff`、`is_superuser`
 - 判断用户是否通过认证(是否登录)：`is_authenticated`
- 创建用户(注册用户)的方法

```
user = User.objects.create_user(username, email, password, **extra_fields)
```

- 用户认证(用户登录)的方法

```
from django.contrib.auth import authenticate
```

```
user = authenticate(username=username, password=password, **kwargs)
```

- 处理密码的方法
 - 设置密码: `set_password(raw_password)`
 - 校验密码: `check_password(raw_password)`

3. 自定义用户模型类

思考：为什么要自定义用户模型类？

- 观察注册界面会发现，美多商城 注册数据 中 必选用户mobile信息。
- 但是Django默认用户模型类中没有mobile字段，所以要自定义用户模型类。

如何自定义用户模型类？

- 继承自**AbstractUser**（可通过阅读Django默认用户模型类的源码得知）。
- 新增 `mobile` 字段。

```
from django.db import models
from django.contrib.auth.models import AbstractUser

# Create your models here.

class User(AbstractUser):
    """自定义用户模型类"""
    mobile = models.CharField(max_length=11, unique=True, verbose_name='手机号')

    class Meta:
        db_table = 'tb_users'
        verbose_name = '用户'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.username
```

4. 知识要点

1. Django自带用户认证系统，核心就是**User**对象，并封装了一系列可用的方法和属性。
2. Django用户认证系统包含了一系列对用户的操作，比如：模型类，认证，权限，分组，密码处理等。
3. Django用户认证系统中的用户模型类可以自定义，继承自**AbstractUser**。
4. [Django用户认证系统说明文档](#)

迁移用户模型类

1. 指定用户模型类

思考：为什么Django默认用户模型类是User？

- 阅读源代码：'django.conf.global_settings'

```
AUTH_USER_MODEL = 'auth.User'
```

结论：

- Django用户模型类是通过全局配置项 **AUTH_USER_MODEL** 决定的

配置规则：

```
AUTH_USER_MODEL = '应用名.模型类名'
```

```
# 指定本项目用户模型类
AUTH_USER_MODEL = 'users.User'
```

2. 迁移用户模型类

1. 创建迁移文件

- python manage.py makemigrations

```
Migrations for 'users':
meiduo_mall/apps/users/migrations/0001_initial.py
  - Create model User
```

2. 执行迁移文件

- python manage.py migrate

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002.Alter_permission_name_max_length... OK
  Applying auth.0003.Alter_user_email_max_length... OK
  Applying auth.0004.Alter_user_username_opts... OK
  Applying auth.0005.Alter_user_last_login_null... OK
  Applying auth.0006.Require_contenttypes_0002... OK
  Applying auth.0007.Alter_validators_add_error_messages... OK
  Applying auth.0008.Alter_user_username_max_length... OK
  Applying users.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying sessions.0001_initial... OK
```

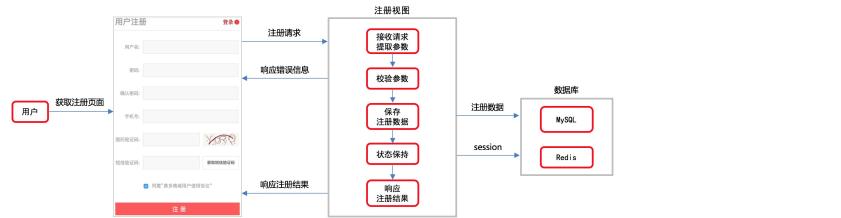
3. 知识要点

1. 用户认证系统中的用户模型类，是通过全局配置项 **AUTH_USER_MODEL** 决定的。
2. 如果迁移自定义用户模型类，必须先配置 **AUTH_USER_MODEL**。

用户注册业务实现

用户注册业务逻辑分析

1. 用户注册业务逻辑分析



用户注册接口设计和定义

1. 设计接口基本思路

- 对于接口的设计，我们要根据具体的业务逻辑，设计出适合业务逻辑的接口。
- 设计接口的思路：
 - 分析要实现的业务逻辑：
 - 明确在这个业务中涉及到几个相关子业务。
 - 将每个子业务当做一个接口来设计。
 - 分析接口的功能任务，明确接口的访问方式与返回数据：
 - 请求方法（如GET、POST、PUT、DELETE等）。
 - 请求地址。
 - 请求参数（如路径参数、查询字符串、表单、JSON等）。
 - 响应数据（如HTML、JSON等）。

2. 用户注册接口设计

1. 请求方式

选项	方案
请求方法	POST
请求地址	/register/

2. 请求参数：表单参数

参数名	类型	是否必传	说明
username	string	是	用户名
password	string	是	密码
password2	string	是	确认密码
mobile	string	是	手机号
sms_code	string	是	短信验证码
allow	string	是	是否同意用户协议

3. 响应结果：HTML

响应结果	响应内容
注册失败	响应错误提示
注册成功	重定向到首页

3. 用户注册接口定义

1.注册视图

```
class RegisterView(View):
    """用户注册"""

    def get(self, request):
        """
        提供注册界面
        :param request: 请求对象
        :return: 注册界面
        """
        return render(request, 'register.html')

    def post(self, request):
        """
        实现用户注册
        :param request: 请求对象
        :return: 注册结果
        """
        pass
```

2.总路由

```
urlpatterns = [
    # users
    url(r'^', include('users.urls', namespace='users')),
]
```

3.子路由

```
urlpatterns = [
    # 注册
    url(r'^register/$', views.RegisterView.as_view(), name='register'),
]
```

用户注册前端逻辑

为了学会使用Vue.js的双向绑定实现用户的交互和页面局部刷新效果。

1. 用户注册页面绑定Vue数据

1.准备div盒子标签

```
<body>
  <div id="app">
    .....
  </div>
</body>
```

2.register.html

- 绑定内容：变量、事件、错误提示等

```
<form method="post" class="register_form" @submit="on_submit" v-cloak>
  {{ csrf_input }}
  <ul>
    <li>
      <label>用户名:</label>
      <input type="text" v-model="username" @blur="check_username" name="username" id="user_name">
      <span class="error_tip" v-show="error_name">{{ error_name_message }}</span>
    </li>
    <li>
      <label>密码:</label>
      <input type="password" v-model="password" @blur="check_password" name="password" id="pwd">
      <span class="error_tip" v-show="error_password">请输入8-20位的密码</span>
    </li>
    <li>
      <label>确认密码:</label>
      <input type="password" v-model="password2" @blur="check_password2" name="password2" id="cpwd">
      <span class="error_tip" v-show="error_password2">两次输入的密码不一致</span>
    </li>
    <li>
      <label>手机号:</label>
      <input type="text" v-model="mobile" @blur="check_mobile" name="mobile" id="phone">
      <span class="error_tip" v-show="error_mobile">{{ error_mobile_message }}</span>
    </li>
  </ul>
</form>
```

```

        </li>
        <li>
            <label>图形验证码:</label>
            <input type="text" name="image_code" id="pic_code" class="msg_input">
            
            <span class="error_tip">请填写图形验证码</span>
        </li>
        <li>
            <label>短信验证码:</label>
            <input type="text" name="sms_code" id="msg_code" class="msg_input">
            <a href="javascript:;" class="get_msg_code">获取短信验证码</a>
            <span class="error_tip">请填写短信验证码</span>
        </li>
        <li class="agreement">
            <input type="checkbox" v-model="allow" @change="check_allow" name="allow" id="allow">
            <label>同意“美多商城用户使用协议”</label>
            <span class="error_tip2" v-show="error_allow">请勾选用户协议</span>
        </li>
        <li class="reg_sub">
            <input type="submit" value="注 册">
        </li>
    </ul>
</form>

```

2. 用户注册JS文件实现用户交互

1. 导入Vue.js库和ajax请求的库

```

<script type="text/javascript" src="{{ static('js/vue-2.5.16.js') }}"></script>
<script type="text/javascript" src="{{ static('js/axios-0.18.0.min.js') }}"></script>

```

2. 准备register.js文件

```
<script type="text/javascript" src="{{ static('js/register.js') }}"></script>
```

绑定内容：变量、事件、错误提示等

```

let vm = new Vue({
    el: '#app',
    // 修改Vue读取变量的语法
    delimiters: ['[[',']]'],
    data: {
        username: '',

```

```

password: '',
password2: '',
mobile: '',
allow: '',

error_name: false,
error_password: false,
error_password2: false,
error_mobile: false,
error_allow: false,

error_name_message: '',
error_mobile_message: '',
},
methods: {
    // 校验用户名
    check_username() {
    },
    // 校验密码
    check_password() {
    },
    // 校验确认密码
    check_password2() {
    },
    // 校验手机号
    check_mobile() {
    },
    // 校验是否勾选协议
    check_allow() {
    },
    // 监听表单提交事件
    on_submit() {
    },
}
});

```

3. 用户交互事件实现

```

methods: {
    // 校验用户名
    check_username() {
        let re = /^[a-zA-Z0-9_-]{5,20}$/;
        if (re.test(this.username)) {
            this.error_name = false;
        } else {
            this.error_name_message = '请输入5-20个字符的用户名';
            this.error_name = true;
        }
    },
    // 校验密码
}

```

```

check_password() {
    let re = /^[0-9A-Za-z]{8,20}$/;
    if (re.test(this.password)) {
        this.error_password = false;
    } else {
        this.error_password = true;
    }
},
// 校验确认密码
check_password2() {
    if (this.password != this.password2) {
        this.error_password2 = true;
    } else {
        this.error_password2 = false;
    }
},
// 校验手机号
check_mobile() {
    let re = /^1[3-9]\d{9}$/;
    if (re.test(this.mobile)) {
        this.error_mobile = false;
    } else {
        this.error_mobile_message = '您输入的手机号格式不正确';
        this.error_mobile = true;
    }
},
// 校验是否勾选协议
check_allow() {
    if (!this.allow) {
        this.error_allow = true;
    } else {
        this.error_allow = false;
    }
},
// 监听表单提交事件
on_submit() {
    this.check_username();
    this.check_password();
    this.check_password2();
    this.check_mobile();
    this.check_allow();

    if (this.error_name == true || this.error_password == true || this.error_pa
    ssword2 == true
        || this.error_mobile == true || this.error_allow == true) {
        // 禁用表单的提交
        window.event.returnValue = false;
    }
},
}

```

4. 知识要点

1. Vue绑定页面的套路
 - 导入Vue.js库和ajax请求的库
 - 准备div盒子标签
 - 准备js文件
 - html页面绑定变量、事件等
 - js文件定义变量、事件等
2. 错误提示
 - 如果错误提示信息是固定的，可以把错误提示信息写死，再通过v-show控制是否展示
 - 如果错误提示信息不是固定的，可以使用绑定的变量动态的展示错误提示信息，再通过v-show控制是否展示
3. 修改Vue变量的读取语法，避免和Django模板语法冲突
 - `delimiters: ['[[',']]'`
4. 后续的页面中如果有类似的交互和刷新效果，也可按照此套路实现

用户注册后端逻辑

1. 接收参数

提示：用户注册数据是从注册表单发送过来的，所以使用 `request.POST` 来提取。

```
username = request.POST.get('username')
password = request.POST.get('password')
password2 = request.POST.get('password2')
mobile = request.POST.get('mobile')
allow = request.POST.get('allow')
```

2. 校验参数

前端校验过的后端也要校验，后端的校验和前端的校验是一致的

```
# 判断参数是否齐全
# 判断用户名是否是5-20个字符
# 判断密码是否是8-20个数字
# 判断两次密码是否一致
# 判断手机号是否合法
# 判断是否勾选用户协议

# 判断参数是否齐全
if not all([username, password, password2, mobile, allow]):
    return http.HttpResponseForbidden('缺少必传参数')
# 判断用户名是否是5-20个字符
if not re.match(r'^[a-zA-Z0-9_-]{5,20}$', username):
    return http.HttpResponseForbidden('请输入5-20个字符的用户名')
# 判断密码是否是8-20个数字
if not re.match(r'^[0-9A-Za-z]{8,20}$', password):
    return http.HttpResponseForbidden('请输入8-20位的密码')
# 判断两次密码是否一致
if password != password2:
    return http.HttpResponseForbidden('两次输入的密码不一致')
# 判断手机号是否合法
if not re.match(r'^1[3-9]\d{9}$', mobile):
    return http.HttpResponseForbidden('请输入正确的手机号码')
# 判断是否勾选用户协议
if allow != 'on':
    return http.HttpResponseForbidden('请勾选用户协议')
```

提示：这里校验的参数，前端已经校验过，如果此时参数还是出错，说明该请求是非正常渠道发送的，所以直接禁止本次请求。

3. 保存注册数据

- 这里使用Django认证系统用户模型类提供的 `create_user()` 方法创建新的用户。
- 这里 `create_user()` 方法中封装了 `set_password()` 方法加密密码。

```
# 保存注册数据
try:
    User.objects.create_user(username=username, password=password, mobile=mobile)
except DatabaseError:
    return render(request, 'register.html', {'register_errmsg': '注册失败'})

# 响应注册结果
return HttpResponse('注册成功，重定向到首页')
```

如果注册失败，我们需要在页面上渲染出注册失败的提示信息。

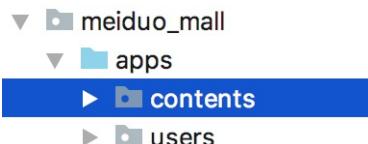
```
{% if register_errmsg %}
    <span class="error_tip2">{{ register_errmsg }}</span>
{% endif %}
```

4. 响应注册结果

- 重要提示：注册成功，重定向到首页

1. 创建首页广告应用：contents

```
$ cd ~/projects/meiduo_project/meiduo_mall/meiduo_mall/apps
$ python ../../manage.py startapp contents
```



2. 定义首页广告视图：IndexView

```
class IndexView(View):
    """首页广告"""

    def get(self, request):
        """提供首页广告页面"""
        return render(request, 'index.html')
```

3. 配置首页广告路由：绑定命名空间

```
# contents
url(r'^', include('contents.urls', namespace='contents')),
```

```
# 首页广告
url(r'^$', views.IndexView.as_view(), name='index'),
```

4. 测试首页广告是否可以正常访问

```
http://127.0.0.1:8000/
```

5. 响应注册结果：重定向到首页

```
# 响应注册结果
return redirect(reverse('contents:index'))
```

5. 知识要点

1. 后端逻辑编写套路：
 - 业务逻辑分析
 - 接口设计和定义
 - 接收和校验参数
 - 实现主体业务逻辑
 - 响应结果
2. 注册业务逻辑核心思想：
 - 保存用户注册数据

状态保持

说明：

- 如果需求是注册成功后即表示用户登入成功，那么此时可以在注册成功后实现状态保持
- 如果需求是注册成功后不表示用户登入成功，那么此时不用在注册成功后实现状态保持

美多商城的需求是：注册成功后即表示用户登入成功

1. login()方法介绍

1. 用户登入本质：

- 状态保持
- 将通过认证的用户的唯一标识信息（比如：用户ID）写入到当前浏览器的 cookie 和服务端的 session 中。

2. login()方法：

- Django用户认证系统提供了 `login()` 方法。
- 封装了写入session的操作，帮助我们快速登入一个用户，并实现状态保持。

3. `login()`位置：

- `django.contrib.auth.__init__.py` 文件中。

```
login(request, user, backend=None)
```

4. 状态保持 session 数据存储的位置：**Redis数据库的1号库**

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "session"
```

2. login()方法登入用户

```
# 保存注册数据
try:
    user = User.objects.create_user(username=username, password=password, mobile=mobile)
except DatabaseError:
    return render(request, 'register.html', {'register_errmsg': '注册失败'})

# 登入用户，实现状态保持
login(request, user)

# 响应注册结果
return redirect(reverse('contents:index'))
```

3. 查看状态保持结果

The screenshot shows the Network tab of a browser developer tools interface. A request to '127.0.0.1' is selected, and the 'Cookies' section is expanded. It lists two cookies: 'csrftoken' and 'sessionid'. The 'sessionid' cookie is highlighted with a gray background.

名称	内容	域名
sessionid	0p4e5naanv4swkbvelbo2sz6p3fpkto8	127.0.0.1

```
127.0.0.1:6379[1]> keys *
1) ":1:django.contrib.sessions.cache0p4e5naanv4swkbvelbo2sz6p3fpkto8"
```

4. 知识要点

1. 登入用户，并实现状态保持的方式： `login(request, user, backend=None)`

用户名重复注册

1. 用户名重复注册逻辑分析



2. 用户名重复注册接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/usernames/(?P<username>[a-zA-Z0-9_-]{5,20})/count/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
username	string	是	用户名

3. 响应结果：JSON

响应结果	响应内容
code	状态码
errmsg	错误信息
count	记录该用户名的个数

3. 用户名重复注册后端逻辑

```

class UsernameCountView(View):
    """判断用户名是否重复注册"""

    def get(self, request, username):
        """
        :param request: 请求对象
        :param username: 用户名
        """
  
```

```

    :return: JSON
    """
    count = User.objects.filter(username=username).count()
    return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'count': count})
}

```

4. 用户名重复注册前端逻辑

```

if (this.error_name == false) {
    let url = '/usernames/' + this.username + '/count/';
    axios.get(url, {
        responseType: 'json'
    })
    .then(response => {
        if (response.data.count == 1) {
            this.error_name_message = '用户名已存在';
            this.error_name = true;
        } else {
            this.error_name = false;
        }
    })
    .catch(error => {
        console.log(error.response);
    })
}

```

5. 知识要点

1. 判断用户名重复注册的核心思想：
 - 使用用户名查询该用户名对应的记录是否存在，如果存在，表示重复注册了，反之，没有重复注册。
2. axios发送异步请求套路：
 - 处理用户交互
 - 收集请求参数
 - 准备请求地址
 - 发送异步请求
 - 得到服务器响应
 - 控制界面展示效果

手机号重复注册

1. 手机号重复注册逻辑分析



2. 手机号重复注册接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/mobiles/(?P<mobile>1[3-9]\d{9})/count/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
mobile	string	是	手机号

3. 响应结果：JSON

响应结果	响应内容
code	状态码
errmsg	错误信息
count	记录该用户名的个数

3. 手机号重复注册后端逻辑

```

class MobileCountView(View):
    """判断手机号是否重复注册"""

    def get(self, request, mobile):
        ...
        :param request: 请求对象
        :param mobile: 手机号
  
```

```
:return: JSON
    ...
    count = User.objects.filter(mobile=mobile).count()
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'count': count})
```

4. 手机号重复注册前端逻辑

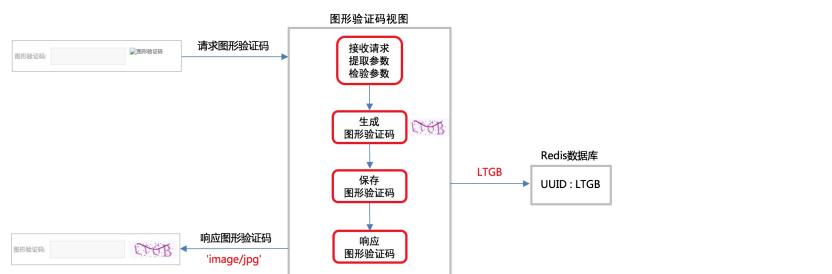
```
if (this.error_mobile == false) {
    let url = '/mobiles/' + this.mobile + '/count/';
    axios.get(url, {
        responseType: 'json'
    })
    .then(response => {
        if (response.data.count == 1) {
            this.error_mobile_message = '手机号已存在';
            this.error_mobile = true;
        } else {
            this.error_mobile = false;
        }
    })
    .catch(error => {
        console.log(error.response);
    })
}
```

验证码

验证码

图形验证码

图形验证码逻辑分析



需要新建应用 verifications

知识要点

1. 后端
 - 生成、保存、响应图形验证码。
 - 将图形验证码的文字信息保存到Redis数据库，是为发送短信验证码前的校验做准备。
 - UUID 用于唯一区分该图形验证码属于哪个用户，也可使用其他唯一标识信息来实现。
2. 前端
 - 处理用户对于图形验证码的交互和校验。

图形验证码接口设计和定义

1. 图形验证码接口设计

1. 请求方式

选项	方案
请求方法	GET
请求地址	/image_codes/(?P<uuid>[\w-]+)/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
uuid	string	是	唯一编号

3. 响应结果：image/jpg



2. 图形验证码接口定义

1. 图形验证码视图

```
class ImageCodeView(View):
    """图形验证码"""

    def get(self, request, uuid):
        """
        :param request: 请求对象
        :param uuid: 唯一标识图形验证码所属于的用户
        :return: image/jpg
        """
        pass
```

2. 总路由

```
# verifications
url(r'^', include('verifications.urls')),
```

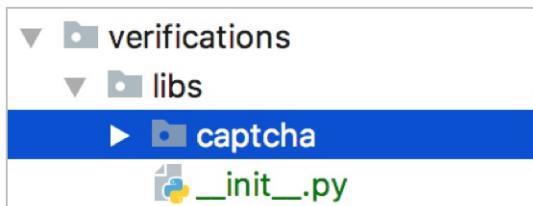
3. 子路由

```
# 图形验证码
url(r'^image_codes/(?P<uuid>[\w-]+)/$', views.ImageCodeView.as_view()),
```


图形验证码后端逻辑

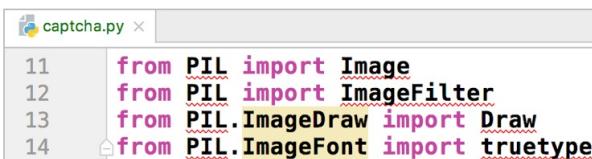
1. 准备captcha扩展包

提示：captcha 扩展包用于后端生成图形验证码



可能出现的错误

- 报错原因：环境中没有Python处理图片的库：PIL



解决办法

- 安装Python处理图片的库： pip install Pillow

2. 准备Redis数据库

准备Redis的2号库存储验证码数据

```

"verify_code": { # 验证码
    "BACKEND": "django_redis.cache.RedisCache",
    "LOCATION": "redis://127.0.0.1:6379/2",
    "OPTIONS": {
        "CLIENT_CLASS": "django_redis.client.DefaultClient",
    }
},

```

3. 图形验证码后端逻辑实现

```

class ImageCodeView(View):
    """图形验证码"""

    def get(self, request, uuid):
        """
        :param request: 请求对象
        :param uuid: 唯一标识图形验证码所属于的用户
        """

```

```
:return: image/jpg
"""
# 生成图形验证码
text, image = captcha.generate_captcha()

# 保存图形验证码
redis_conn = get_redis_connection('verify_code')
redis_conn.setex('img_%s' % uuid, constants.IMAGE_CODE_REDIS_EXPIRES, text)

# 响应图形验证码
return http.HttpResponse(image, content_type='image/jpg')
```

图形验证码前端逻辑

1. Vue实现图形验证码展示

1.register.js

```

mounted(){
    // 生成图形验证码
    this.generate_image_code();
},
methods: {
    // 生成图形验证码
    generate_image_code(){
        // 生成UUID。generateUUID()：封装在common.js文件中，需要提前引入
        this.uuid = generateUUID();
        // 拼接图形验证码请求地址
        this.image_code_url = "/image_codes/" + this.uuid + "/";
    },
    .....
}

```

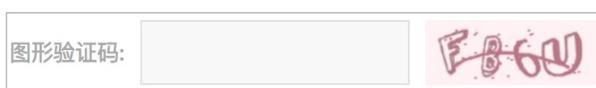
2.register.html

```

<li>
    <label>图形验证码:</label>
    <input type="text" name="image_code" id="pic_code" class="msg_input">
    
    <span class="error_tip">请填写图形验证码</span>
</li>

```

3.图形验证码展示和存储效果



```

127.0.0.1:6379[2]> keys *
1) "img_55b57522-2362-42be-b603-26c8667e2299"
127.0.0.1:6379[2]> get img_55b57522-2362-42be-b603-26c8667e2299
"FB6U"

```

2. Vue实现图形验证码校验

1.register.html

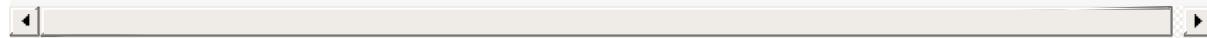
```

<li>
    <label>图形验证码:</label>

```

```
<input type="text" v-model="image_code" @blur="check_image_code" name="image_code" id="pic_code" class="msg_input">

<span class="error_tip" v-show="error_image_code">[[ error_image_code_message ]]</span>
</li>
```



2.register.js

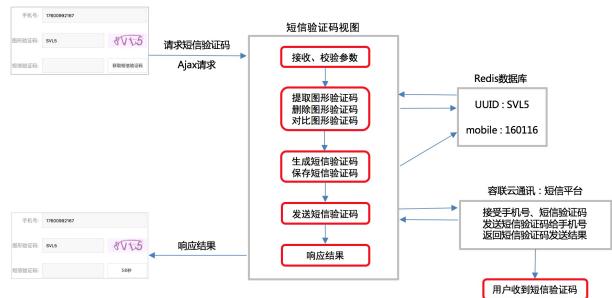
```
check_image_code(){
  if(this.image_code.length != 4) {
    this.error_image_code_message = '请填写图片验证码';
    this.error_image_code = true;
  } else {
    this.error_image_code = false;
  }
},
```

3.图形验证码校验效果



短信验证码

短信验证码逻辑分析



知识要点

1. 保存短信验证码是为注册做准备的。
2. 为了避免用户使用图形验证码恶意测试，后端提取了图形验证码后，立即删除图形验证码。
3. Django不具备发送短信的功能，所以我们借助第三方的容联云通讯短信平台来帮助我们发送短信验证码。

容联云通讯短信平台

1. 容联云通讯短信平台介绍

1.1. 容联云官网

- 容联云通讯网址: <https://www.yuntongxun.com/>
- 注册并登陆

The registration page (left) has fields for Email Address, Password, Confirm Password, Mobile Number, and Captcha, with a 'Register' button at the bottom. The login page (right) has fields for Email/Phone and Password, with a 'Remember Username' checkbox, a 'Forgot Password?' link, and a 'Login' button.

2. 容联云管理控制台

开发者主账号

ACCOUNT SID: 8aaf070862181ad5016236f3bcc811d5
AUTH TOKEN: 4e8*****16ef [查看](#)
Rest URL(生产): <https://app.cloopen.com:8883>
AppID(默认): 8aaf070868747811016883f12ef3062c [未上线](#) [APP TOKEN](#) (APP TOKEN 请到应用管理中获取)
鉴权IP: 尚未开启 (开启后可有效提升账户安全) [开启鉴权IP \(推荐\)](#)

3. 容联云创建应用

创建应用

* 应用名称: meiduo_mall 不超过12个汉字/英文字母/数字

启用回调地址 勾选启用

启用IVR 勾选启用 (收费功能)

启用TTS 勾选启用 (收费功能)

* 使用功能 (可多选)

<input checked="" type="checkbox"/> 短信验证码	<input checked="" type="checkbox"/> 语音验证码	<input checked="" type="checkbox"/> IM 即时通讯
<input checked="" type="checkbox"/> 语音通话	<input checked="" type="checkbox"/> 视频通话	<input checked="" type="checkbox"/> 会议
<input checked="" type="checkbox"/> 呼叫中心	<input checked="" type="checkbox"/> 流量	<input checked="" type="checkbox"/> 电话通知
<input checked="" type="checkbox"/> 短信通知		

确定

应用列表

应用名称	状态	创建时间	操作
meiduo_mall	未上线	2018-11-14 14:14:14	应用管理 编辑 上线 删除

4. 应用申请上线，并进行资质认证

申请上线

您还未完成实名认证，未完成实名认证的应用只能在有限的环境中开发测试，如需正式发布上线请先完成实名认证

请前往：「帐号管理」--「认证信息」

[现在去认证](#) [取消](#)

资质认证信息

开发者类型	开发者
真实姓名	T...
证件类型	身份证
证件号码	...

5. 完成资质认证，应用成功上线

开发者主账号

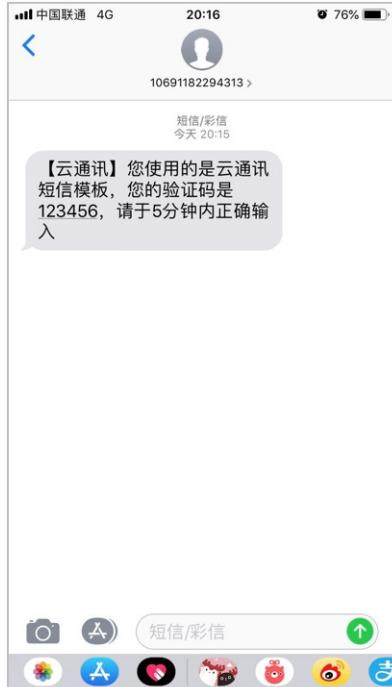
ACCOUNT SID: 8aaf070862181ad5016236f3bcc811d5
AUTH TOKEN: 4e8*****16ef [查看](#)
Rest URL(生产): <https://app.cloopen.com:8883> (APP TOKEN 请到应用管理中获取)
AppID(默认): 8aaf070868747811016883f12ef3062c 运营中 (APP TOKEN 请到应用管理中获取)
鉴权IP: 尚未开启 (开启后可有效提升账户安全) [开启鉴权IP \(推荐\)](#)

应用列表

应用名称	状态	创建时间	操作
meiduo_mall	运营中	2018-11-14 14:14:14	应用管理 编辑 暂停 删除

6. 短信模板

模板类型	模板内容
验证码	您的验证码为{1}，请于{2}内正确输入，如非本人操作，请忽略此短信。
	验证码：{1}，打死都不要告诉别人哦！
	登录验证码：{1}，如非本人操作，请忽略此短信。
	验证码为{1}，您正在修改登录密码，请确认是本人操作。



7.添加测试号码

平台测试号码

①	中国(+86)	17600992168	可移除
②	中国(+86)	17600992166	去验证
③	中国(+86)		

测试号码

费用说明：

1、新注册开发者拥有默认赠送一定额度的金额供开发应用期间调试使用，超出部分计费按照标准资费执行
 2、未上线应用不产生每月最低消费，计费按照标准资费执行
 3、上线应用请先配置套餐结算档次
 4、未充值和通过身份认证帐号只允许测试，不允许正式上线。测试必须在沙盒环境内进行

2. 容联云通讯短信SDK测试

1. 模板短信SDK下载

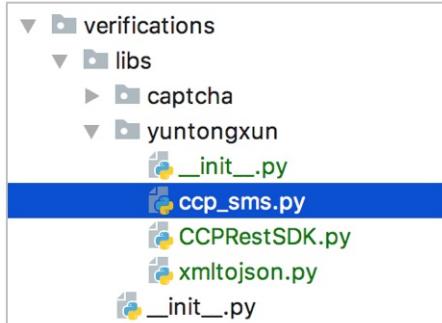
- https://www.yuntongxun.com/doc/ready/demo/1_4_1_2.html

2.模板短信SDK使用说明

- <http://doc.yuntongxun.com/p/5a533e0c3b8496dd00dce08c>

3.集成模板短信SDK

- `CCPRestSDK.py` : 由容联云通讯开发者编写的官方SDK文件，包括发送模板短信的方法
- `ccp_sms.py` : 调用发送模板短信的方法



4.模板短信SDK测试

- `ccp_sms.py` 文件中

```

# -*- coding:utf-8 -*-

from verifications.libs.yuntongxun.CCPRestSDK import REST

# 说明：主账号，登陆云通讯网站后，可在"控制台-应用"中看到开发者主账号ACCOUNT SID
_accountSid = '8aad070862181ad5016236f3bcc811d5'

# 说明：主账号Token，登陆云通讯网站后，可在控制台-应用中看到开发者主账号AUTH TOKEN
_accountToken = '4e831592bd464663b0de944df13f16ef'

# 请使用管理控制台首页的APPID或自己创建应用的APPID
_appId = '8aad070868747811016883f12ef3062c'

# 说明：请求地址，生产环境配置成app.cloopen.com
_serverIP = 'sandboxapp.cloopen.com'

# 说明：请求端口，生产环境为8883
_serverPort = "8883"

# 说明：REST API版本号保持不变
_softVersion = '2013-12-26'

# 云通讯官方提供的发送短信代码实例
# 发送模板短信
# @param to 手机号码
# @param datas 内容数据 格式为数组 例如: {'12', '34'}，如不需替换请填 ''
# @param $tempId 模板Id
def sendTemplateSMS(to, datas, tempId):
    
```

```

# 初始化REST SDK
rest = REST(_serverIP, _serverPort, _softVersion)
rest.setAccount(_accountSid, _accountToken)
rest.setAppId(_appId)

result = rest.sendTemplateSMS(to, datas, tempId)
print(result)

if __name__ == '__main__':
    # 注意： 测试的短信模板编号为1
    sendTemplateSMS('17600992168', ['123456', 5], 1)

```

5. 模板短信SDK返回结果说明

```

{
    'statusCode': '000000', // 状态码。'000000'表示成功，反之，失败
    'templateSMS':
    {
        'smsMessageSid': 'b5768b09e5bc4a369ed35c444c13a1eb', // 短信唯一标识符
        'dateCreated': '20190125185207' // 短信发送时间
    }
}

```

3. 封装发送短信单例类

- 单例模式：是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。
当你希望在整个系统中，某个类只能出现一个实例时，就可以使用单例设计模式。

1. 封装发送短信单例类

```

class CCP(object):
    """发送短信的单例类"""

    def __new__(cls, *args, **kwargs):
        # 判断是否存在类属性__instance，__instance是类CCP的唯一对象，即单例
        if not hasattr(CCP, "__instance"):
            cls.__instance = super(CCP, cls).__new__(cls, *args, **kwargs)
            cls.__instance.rest = REST(_serverIP, _serverPort, _softVersion)
            cls.__instance.rest.setAccount(_accountSid, _accountToken)
            cls.__instance.rest.setAppId(_appId)
        return cls.__instance

```

2. 封装发送短信单例方法

```

def send_template_sms(self, to, datas, temp_id):
    """
    发送模板短信单例方法

```

```
:param to: 注册手机号
:param datas: 模板短信内容数据, 格式为列表, 例如: ['123456', 5], 如不需替换请填 ''
:param temp_id: 模板编号, 默认免费提供id为1的模板
:return: 发短信结果
"""

result = self.rest.sendTemplateSMS(to, datas, temp_id)
if result.get("statusCode") == "000000":
    # 返回0, 表示发送短信成功
    return 0
else:
    # 返回-1, 表示发送失败
    return -1
```

3. 测试单例类发送模板短信结果

```
if __name__ == '__main__':
    # 注意: 测试的短信模板编号为1
    CCP().send_template_sms('17600992168', ['123456', 5], 1)
```

4. 知识要点

1. 容联云通讯只是发送短信的平台之一, 还有其他云平台可用, 比如, 阿里云等, 实现套路都是相通的。
2. 将发短信的类封装为单例, 属于性能优化的一种方案。

短信验证码后端逻辑

1. 短信验证码接口设计

1. 请求方式

选项	方案
请求方法	GET
请求地址	/sms_codes/(?P<mobile>1[3-9]\d{9})/

2. 请求参数：路径参数和查询字符串

参数名	类型	是否必传	说明
mobile	string	是	手机号
image_code	string	是	图形验证码
uuid	string	是	唯一编号

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

2. 短信验证码接口定义

```
class SMSCodeView(View):
    """短信验证码"""

    def get(self, request, mobile):
        """
        :param request: 请求对象
        :param mobile: 手机号
        :return: JSON
        """
        pass
```

3. 短信验证码后端逻辑实现

```
class SMSCodeView(View):
    """短信验证码"""

    def get(self, request, mobile):
```

```

    """
    :param request: 请求对象
    :param mobile: 手机号
    :return: JSON
    """

    # 接收参数
    image_code_client = request.GET.get('image_code')
    uuid = request.GET.get('uuid')

    # 校验参数
    if not all([image_code_client, uuid]):
        return http.JsonResponse({'code': RETCODE.NECESSARYPARAMERR, 'errmsg': '缺少必传参数'})

    # 创建连接到redis的对象
    redis_conn = get_redis_connection('verify_code')
    # 提取图形验证码
    image_code_server = redis_conn.get('img_%s' % uuid)
    if image_code_server is None:
        # 图形验证码过期或者不存在
        return http.JsonResponse({'code': RETCODE.IMAGECODEERR, 'errmsg': '图形验证码失效'})
    # 删除图形验证码，避免恶意测试图形验证码
    redis_conn.delete('img_%s' % uuid)
    # 对比图形验证码
    image_code_server = image_code_server.decode() # bytes转字符串
    if image_code_client.lower() != image_code_server.lower(): # 转小写后比较
        return http.JsonResponse({'code': RETCODE.IMAGECODEERR, 'errmsg': '输入图形验证码有误'})

    # 生成短信验证码：生成6位数验证码
    sms_code = '%06d' % random.randint(0, 999999)
    logger.info(sms_code)
    # 保存短信验证码
    redis_conn.setex('sms_%s' % mobile, constants.SMS_CODE_REDIS_EXPIRES, sms_code)
    # 发送短信验证码
    CCP().send_template_sms(mobile, [sms_code, constants.SMS_CODE_REDIS_EXPIRES // 60], constants.SEND_SMS_TEMPLATE_ID)

    # 响应结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '发送短信成功'})

```

短信验证码前端逻辑

1. axios请求短信验证码

1. 展示倒计时60秒的效果

```
<li>
    <label>短信验证码:</label>
    <input type="text" name="sms_code" id="msg_code" class="msg_input">
    <a @click="send_sms_code" class="get_msg_code">[[ sms_code_tip ]]</a>
    <span class="error_tip">请填写短信验证码</span>
</li>
```

```
send_sms_code(){
    // 避免重复点击
    if (this.sending_flag == true) {
        return;
    }
    this.sending_flag = true;

    // 校验参数
    this.check_mobile();
    this.check_image_code();
    if (this.error_mobile == true || this.error_image_code == true) {
        this.sending_flag = false;
        return;
    }

    // 请求短信验证码
    let url = '/sms_codes/' + this.mobile + '?image_code=' + this.image_code + '&uid=' + this.uuid;
    axios.get(url, {
        responseType: 'json'
    })
    .then(response => {
        if (response.data.code == '0') {
            // 倒计时60秒
            let num = 60;
            let t = setInterval(() => {
                if (num == 1) {
                    clearInterval(t);
                    this.sms_code_tip = '获取短信验证码';
                    this.generate_image_code();
                    this.sending_flag = false;
                } else {
                    num -= 1;
                    // 展示倒计时信息
                }
            }, 1000)
        }
    })
}
```

```

        this.sms_code_tip = num + '秒';
    }
},
),
} else {
    if (response.data.code == '4001') {
        this.error_image_code_message = response.dataerrmsg;
        this.error_image_code = true;
    } else { // 4003 缺少必传参数
        console.log(response.data)
    }
    this.sending_flag = false;
}
})
.catch(error => {
    console.log(error.response);
    this.sending_flag = false;
})
},

```

2.发送短信验证码效果展示

手机号: 17600992168

图形验证码: weab

短信验证码: 50秒

2. 短信验证码用户交互和校验

1.register.html

```

<li>
    <label>短信验证码:</label>
    <input type="text" v-model="sms_code" @blur="check_sms_code" name="sms_code" id="msg_code" class="msg_input">
    <a @click="send_sms_code" class="get_msg_code">[[ sms_code_tip ]]</a>
    <span class="error_tip" v-show="error_sms_code">[[ error_sms_code_message ]]</span>
</li>

```

2.register.js

```

check_sms_code(){
    if(this.sms_code.length != 6){
        this.error_sms_code_message = '请填写短信验证码';
    }
}

```

```
    this.error_sms_code = true;
} else {
    this.error_sms_code = false;
}
},
```

补充注册时短信验证逻辑

1. 补充注册时短信验证后端逻辑

1. 接收短信验证码参数

```
# 接收参数
sms_code_client = request.POST.get('sms_code')

# 判断参数是否齐全
if not all([username, password, password2, mobile, sms_code_client, allow]):
    return http.HttpResponseForbidden('缺少必传参数')
```

2. 保存注册数据之前，对比短信验证码

```
redis_conn = get_redis_connection('verify_code')
sms_code_server = redis_conn.get('sms_%s' % mobile)
if sms_code_server is None:
    return render(request, 'register.html', {'sms_code_errmsg': '无效的短信验证码'})
if sms_code_client != sms_code_server.decode():
    return render(request, 'register.html', {'sms_code_errmsg': '输入短信验证码有误'})
```

2. 补充注册时短信验证前端逻辑

1.register.html

```
<li>
    <label>短信验证码:</label>
    <input type="text" v-model="sms_code" @blur="check_sms_code" name="sms_code" id="msg_code" class="msg_input">
    <a @click="send_sms_code" class="get_msg_code">[ [ sms_code_tip ] ]</a>
    <span v-show="error_sms_code" class="error_tip">[ [ error_sms_code_message ] ]</span>
</li>
```

2.register.js

```
on_submit(){
    this.check_username();
    this.check_password();
    this.check_password2();
```

```
this.check_mobile();
this.check_sms_code();
this.check_allow();

if(this.error_name == true || this.error_password == true || this.error_passwor
d2 == true
    || this.error_mobile == true || this.error_allow == true) {
    // 注册参数不全: 禁用表单
    window.event.returnValue = false;
}
},
```

避免频繁发送短信验证码

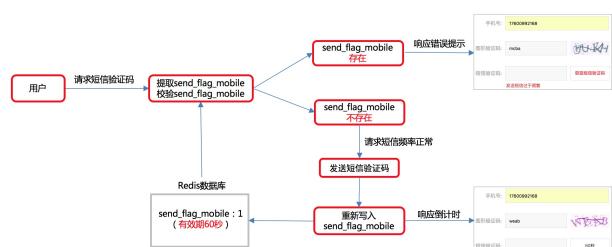
存在的问题：

- 虽然我们在前端界面做了60秒倒计时功能。
- 但是恶意用户可以绕过前端面向后端频繁请求短信验证码。

解决办法：

- 在后端也要限制用户请求短信验证码的频率。60秒内只允许一次请求短信验证码。
- 在Redis数据库中缓存一个数值，有效期设置为60秒。

1. 避免频繁发送短信验证码逻辑分析



2. 避免频繁发送短信验证码逻辑实现

1. 提取、校验send_flag

```

send_flag = redis_conn.get('send_flag_%s' % mobile)
if send_flag:
    return http.JsonResponse({'code': RETCODE.THROTTLINGERR, 'errmsg': '发送短信过于频繁'})
  
```

2. 重新写入send_flag

```

# 保存短信验证码
redis_conn.setex('sms_%s' % mobile, constants.SMS_CODE_REDIS_EXPIRES, sms_code)
# 重新写入send_flag
redis_conn.setex('send_flag_%s' % mobile, constants.SEND_SMS_CODE_INTERVAL, 1)
  
```

3. 界面渲染频繁发送短信提示信息

```

if (response.data.code == '4001') {
    this.error_image_code_message = response.data errmsg;
    this.error_image_code = true;
} else if (response.data.code == '4002') {
    this.error_sms_code_message = response.data errmsg;
    this.error_sms_code = true;
} else { // 4003 缺少必传参数
  
```

```
    console.log(response.data)
}
```

pipeline操作Redis数据库

Redis的 C - S 架构：

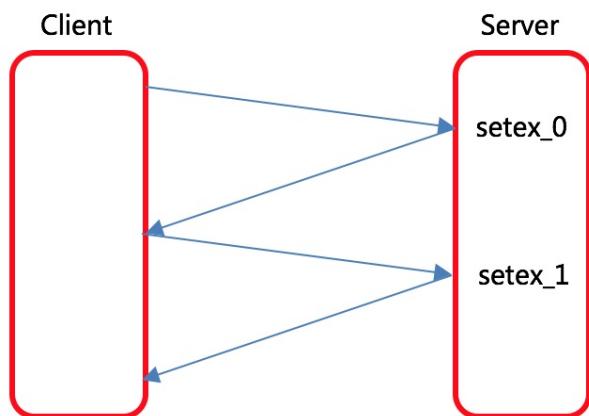
- 基于客户端-服务端模型以及请求/响应协议的TCP服务。
- 客户端向服务端发送一个查询请求，并监听Socket返回。
- 通常是以阻塞模式，等待服务端响应。
- 服务端处理命令，并将结果返回给客户端。

存在的问题：

- 如果Redis服务端需要同时处理多个请求，加上网络延迟，那么服务端利用率不高，效率降低。

解决的办法：

- 管道pipeline



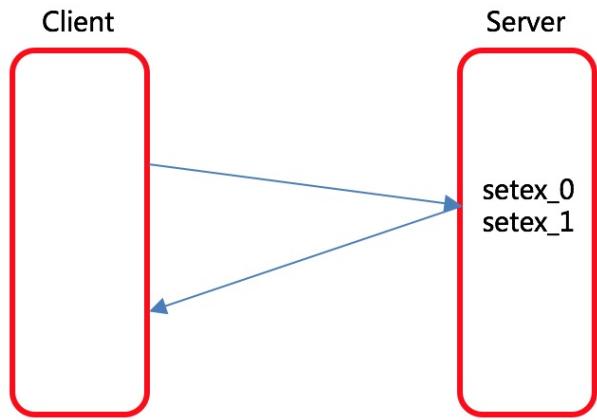
1. pipeline的介绍

管道pipeline

- 可以一次性发送多条命令并在执行完后一次性将结果返回。
- pipeline通过减少客户端与Redis的通信次数来实现降低往返延时时间。

实现的原理

- 实现的原理是队列。
- Client可以将三个命令放到一个tcp报文一起发送。
- Server则可以将三条命令的处理结果放到一个tcp报文返回。
- 队列是先进先出，这样就保证数据的顺序性。



2. pipeline操作Redis数据库

1. 实现步骤

1. 创建Redis管道
2. 将Redis请求添加到队列
3. 执行请求

2. 代码实现

```
# 创建Redis管道
pl = redis_conn.pipeline()
# 将Redis请求添加到队列
pl.setex('sms_%s' % mobile, constants.SMS_CODE_EXPIRES, sms_code)
pl.setex('send_flag_%s' % mobile, constants.SEND_SMS_CODE_INTERVAL, 1)
# 执行请求
pl.execute()
```

异步方案Celery

生产者消费者设计模式

思考：

- 下面两行代码存在什么问题？

```
# 发送短信验证码
CCP().send_template_sms(mobile,[sms_code, constants.SMS_CODE_REDIS_EXPIRES // 60], constants.SEND_SMS_TEMPLATE_ID)
# 响应结果
return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '发送短信成功'})
```

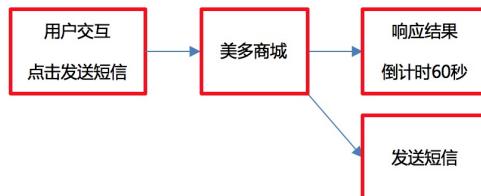
问题：

- 我们的代码是自上而下同步执行的。
- 发送短信是耗时的操作。如果短信被阻塞住，用户响应将会延迟。
- 响应延迟会造成用户界面的倒计时延迟。



解决：

- 异步发送短信
- 发送短信和响应分开执行，将 `发送短信` 从主营业务中 `解耦` 出来。



思考：

- 如何将 `发送短信` 从主营业务中 `解耦` 出来。

生产者消费者设计模式介绍

- 为了将 `发送短信` 从主营业务中 `解耦` 出来，我们引入 **生产者消费者设计模式**。
- 它是最常用的解耦方式之一，寻找中间人(**broker**)搭桥，保证两个业务没有直接关联。



总结：

- 生产者生成消息，缓存到消息队列中，消费者读取消息队列中的消息并执行。
- 由美多商城生成发送短信消息，缓存到消息队列中，消费者读取消息队列中的发送短信消息并执行。

Celery介绍和使用

思考：

- 消费者取到消息之后，要消费掉（执行任务），需要我们去实现。
- 任务可能出现高并发的情况，需要补充多任务的方式执行。
- 耗时任务很多种，每种耗时任务编写的生产者和消费者代码有重复。
- 取到的消息什么时候执行，以什么样的方式执行。

结论：

- 实际开发中，我们可以借助成熟的工具 celery 来完成。
- 有了 Celery，我们在使用生产者消费者模式时，只需要关注任务本身，极大的简化了程序员的开发流程。

1. Celery介绍

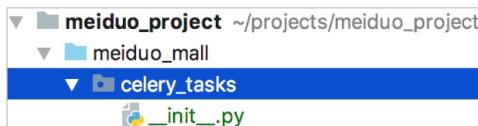
- Celery介绍：
 - 一个简单、灵活且可靠、处理大量消息的分布式系统，可以在一台或者多台机器上运行。
 - 单个 Celery 进程每分钟可处理数以百万计的任务。
 - 通过消息进行通信，使用 消息队列（broker） 在 客户端 和 消费者 之间进行协调。
- 安装Celery：

```
$ pip install -U Celery
```

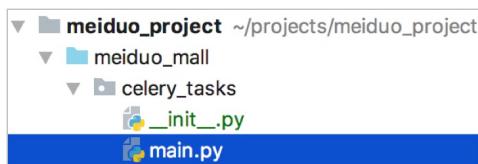
- [Celery官方文档](#)

2. 创建Celery实例并加载配置

1.定义Celery包



2.创建Celery实例



celery_tasks.main.py

```
# celery启动文件
```

```
from celery import Celery

# 创建celery实例
celery_app = Celery('meiduo')
```

3. 加载Celery配置



celery_tasks.config.py

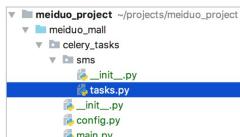
```
# 指定消息队列的位置
broker_url = "redis://127.0.0.1/10"
```

celery_tasks.main.py

```
# celery启动文件
from celery import Celery

# 创建celery实例
celery_app = Celery('meiduo')
# 加载celery配置
celery_app.config_from_object('celery_tasks.config')
```

3. 定义发送短信任务



1. 定义任务: celery_tasks.sms.tasks.py

```
@celery_app.task(name='ccp_send_sms_code')
def ccp_send_sms_code(mobile, sms_code):
    """
    发送短信异步任务
    :param mobile: 手机号
    :param sms_code: 短信验证码
    :return: 成功0 或 失败-1
    """

    send_ret = CCP().send_template_sms(mobile, [sms_code, constants.SMS_CODE_EXPIRES // 60], constants.SEND_SMS_TEMPLATE_ID)
    return send_ret
```

2. 注册任务: celery_tasks.main.py

```
# celery启动文件
from celery import Celery

# 创建celery实例
celery_app = Celery('meiduo')
# 加载celery配置
celery_app.config_from_object('celery_tasks.config')
# 自动注册celery任务
celery_app.autodiscover_tasks(['celery_tasks.sms'])
```

4. 启动Celery服务

```
$ cd ~/projects/meiduo_project/meiduo_mall
$ celery -A celery_tasks.main worker -l info
```

- -A 指对应的应用程序, 其参数是项目中 Celery 实例的位置。
- worker 指这里要启动的 worker。
- -l 指日志等级, 比如 info 等级。



5. 调用发送短信任务

```
# 发送短信验证码
# CCP().send_template_sms(mobile, [sms_code, constants.SMS_CODE_REDIS_EXPIRES // 60]
, constants.SEND_SMS_TEMPLATE_ID)
# Celery异步发送短信验证码
ccp_send_sms_code.delay(mobile, sms_code)
```

```
[2019-07-14 14:27:55,558: INFO/MainProcess] Received task: ccp_send_sms_code[4aecf9b0-dbdc-4c8c-b747-6358bf521835]
[2019-07-14 14:27:55,883: INFO/ForkPoolWorker-8] Task ccp_send_sms_code[4aecf9b0-dbdc-4c8c-b747-6358bf521835] succeeded in 0.241200231s
4700485: 8
```

用户登录

账号登录

用户名登录

1. 用户名登录逻辑分析



2. 用户名登录接口设计

1. 请求方式

选项	方案
请求方法	POST
请求地址	/login/

2. 请求参数：表单

参数名	类型	是否必传	说明
username	string	是	用户名
password	string	是	密码
remembered	string	否	是否记住用户

3. 响应结果：HTML

字段	说明
登录失败	响应错误提示
登录成功	重定向到首页

3. 用户名登录接口定义

```

class LoginView(View):
    """用户名登录"""

    def get(self, request):
        """
        提供用户登录页面
        :param request: 请求对象
        :return: 登录界面
        """
    
```

```

    pass

def post(self, request):
    """
    实现登录逻辑
    :param request: 请求对象
    :return: 登录结果
    """

    pass

```

4. 用户名登录后端逻辑

```

class LoginView(View):
    """用户名登录"""

    def get(self, request):
        """
        提供登录界面
        :param request: 请求对象
        :return: 登录界面
        """

        return render(request, 'login.html')

    def post(self, request):
        """
        实现登录逻辑
        :param request: 请求对象
        :return: 登录结果
        """

        # 接受参数
        username = request.POST.get('username')
        password = request.POST.get('password')
        remembered = request.POST.get('remembered')

        # 校验参数
        # 判断参数是否齐全
        if not all([username, password]):
            return http.HttpResponseForbidden('缺少必传参数')

        # 判断用户名是否是5-20个字符
        if not re.match(r'^[a-zA-Z0-9_-]{5,20}$', username):
            return http.HttpResponseForbidden('请输入正确的用户名或手机号')

        # 判断密码是否是8-20个数字
        if not re.match(r'^[0-9A-Za-z]{8,20}$', password):
            return http.HttpResponseForbidden('密码最少8位，最长20位')

        # 认证登录用户
        user = authenticate(username=username, password=password)

```

```
if user is None:
    return render(request, 'login.html', {'account_errmsg': '账号或密码错误'})
)

# 实现状态保持
login(request, user)
# 设置状态保持的周期
if remembered != 'on':
    # 没有记住用户：浏览器会话结束就过期
    request.session.set_expiry(0)
else:
    # 记住用户：None表示两周后过期
    request.session.set_expiry(None)

# 响应登录结果
return redirect(reverse('contents:index'))
```

5. 知识要点

1. 登录的核心思想：认证和状态保持
 - 通过用户的认证，确定该登录用户是美多商场的注册用户。
 - 通过状态保持缓存用户的唯一标识信息，用于后续是否登录的判断。

多账号登录

- Django自带的用户认证后端默认是使用用户名实现用户认证的。
- 用户认证后端位置： django.contrib.auth.backends.ModelBackend。
- 如果想实现用户名和手机号都可以认证用户，就需要自定义用户认证后端。
- 自定义用户认证后端步骤
 - 在users应用中新建utils.py文件
 - 新建类，继承自ModelBackend
 - 重写认证authenticate()方法
 - 分别使用用户名和手机号查询用户
 - 返回查询到的用户实例

1. 自定义用户认证后端

users.utils.py

```
from django.contrib.auth.backends import ModelBackend
import re
from .models import User

def get_user_by_account(account):
    """
    根据account查询用户
    :param account: 用户名或者手机号
    :return: user
    """
    try:
        if re.match('^\d{3}-\d{8}$', account):
            # 手机号登录
            user = User.objects.get(mobile=account)
        else:
            # 用户名登录
            user = User.objects.get(username=account)
    except User.DoesNotExist:
        return None
    else:
        return user

class UsernameMobileAuthBackend(ModelBackend):
    """自定义用户认证后端"""

    def authenticate(self, request, username=None, password=None, **kwargs):
        pass
```

```

    """
    重写认证方法，实现多账号登录
    :param request: 请求对象
    :param username: 用户名
    :param password: 密码
    :param kwargs: 其他参数
    :return: user
    """

    # 根据传入的username获取user对象。username可以是手机号也可以是账号
    user = get_user_by_account(username)
    # 校验user是否存在并校验密码是否正确
    if user and user.check_password(password):
        return user
    else:
        return None

```

2. 配置自定义用户认证后端

1.Django自带认证后端源码

```

django > conf > global_settings.py
global_settings.py <
507 AUTHENTICATION_BACKENDS = ['django.contrib.auth.backends.ModelBackend']

```

2.配置自定义用户认证后端

```

# 指定自定义的用户认证后端
AUTHENTICATION_BACKENDS = ['users.utils.UsernameMobileAuthBackend']

```

3. 测试自定义用户认证后端



4. 知识要点

1. Django自带的用户认证系统只会使用用户名去认证一个用户。
 - 所以我们为了实现多账号登录，就可以自定义认证后端，采用其他的唯一信息去认证一个用户。

首页用户名展示

欢迎您：itcast | 退出 | 用户中心 | 我的购物车 | 我的订单

1. 首页用户名展示方案

方案一

- 模板中 **request** 变量直接渲染用户名
- 缺点：不方便做首页静态化

```
{% if user.is_authenticated %}  
    <div class="login_btn f1">  
        欢迎您: <em>{{ user.username }}</em>  
        <span>|</span>  
        <a href="#">退出</a>  
    </div>  
    {% else %}  
        <div class="login_btn f1">  
            <a href="login.html">登录</a>  
            <span>|</span>  
            <a href="register.html">注册</a>  
        </div>  
    {% endif %}
```

方案二

- 发送ajax请求获取用户信息
- 缺点：需要发送网络请求

```
<div class="login_btn f1">  
    {# ajax渲染 #}  
</div>
```

方案三

- Vue读取cookie渲染用户信息

```
<div v-if="username" class="login_btn f1">  
    欢迎您: <em>[[ username ]]</em>  
    <span>|</span>  
    <a href="#">退出</a>  
</div>  
<div v-else class="login_btn f1">  
    <a href="login.html">登录</a>
```

```
<span>|</span>
<a href="register.html">注册</a>
</div>
```

结论：

- 对比此三个方案，我们在本项目中选择 方案三

实现步骤：

- 注册或登录后，用户名写入到cookie
- Vue渲染主页用户名

2. 用户名写入到cookie

```
# 响应注册结果
response = redirect(reverse('contents:index'))

# 注册时用户名写入到cookie，有效期15天
response.set_cookie('username', user.username, max_age=3600 * 24 * 15)

return response
```

```
# 响应登录结果
response = redirect(reverse('contents:index'))

# 登录时用户名写入到cookie，有效期15天
response.set_cookie('username', user.username, max_age=3600 * 24 * 15)

return response
```

3. Vue渲染首页用户名

1.index.html

```
<div v-if="username" class="login_btn f1">
    欢迎您: <em>[[ username ]]</em>
    <span>|</span>
    <a href="#">退出</a>
</div>
<div v-else class="login_btn f1">
    <a href="login.html">登录</a>
    <span>|</span>
    <a href="register.html">注册</a>
</div>
```

2.index.js

```
data: {  
    username: getCookie('username'),  
    .....  
},
```

退出登录

1. logout()方法介绍

1. 退出登录：

- 回顾登录：将通过认证的用户的唯一标识信息，写入到当前session会话中
- 退出登录：正好和登录相反（清理session会话信息）

2. logout()方法：

- Django用户认证系统提供了 `logout()` 方法
- 封装了清理session的操作，帮助我们快速实现登出一个用户

3. logout()位置：

- `django.contrib.auth.__init__.py` 文件中

```
logout(request)
```

2. logout()方法使用

```
class LogoutView(View):
    """退出登录"""

    def get(self, request):
        """实现退出登录逻辑"""
        # 清理session
        logout(request)
        # 退出登录，重定向到登录页
        response = redirect(reverse('contents:index'))
        # 退出登录时清除cookie中的username
        response.delete_cookie('username')

    return response
```

3. 知识要点

1. 退出登录的核心思想就是清理登录时缓存的状态保持信息。
2. 由于首页中用户名是从cookie中读取的。所以退出登录时，需要将cookie中用户名清除。

判断用户是否登录

1. 展示用户中心界面

```
class UserInfoView(View):
    """用户中心"""

    def get(self, request):
        """提供个人信息界面"""
        return render(request, 'user_center_info.html')
```

需求：

- 当用户登录后，才能访问用户中心。
- 如果用户未登录，就不允许访问用户中心，将用户引导到登录界面。

实现方案：

- 需要判断用户是否登录，根据是否登录的结果，决定用户是否可以访问用户中心。
- 使用Django自定义的扩展类 `LoginRequiredMixin` 判断用户是否登录

2. `LoginRequiredMixin` 判断用户是否登录

介绍：

- Django用户认证系统提供了方法 `request.user.is_authenticated()` 来判断用户是否登录。如果通过登录验证则返回**True**。反之，返回**False**。
- `LoginRequiredMixin` 封装了判断用户是否登录的操作。

使用：

```
from django.contrib.auth.mixins import LoginRequiredMixin

class UserInfoView(LoginRequiredMixin, View):
    """用户中心"""

    def get(self, request):
        """提供个人信息界面"""
        return render(request, 'user_center_info.html')
```

相关配置：

- 搭配 `LoginRequiredMixin` 表示当用户未通过登录验证时，将用户重定向到登录页面。

```
LOGIN_URL = '/login/'
```

3. 登录时next参数的使用

1.next参数的效果

<http://127.0.0.1:8000/login/?next=/info/>



2.next参数的作用

- 由Django用户认证系统提供，搭配 login_required装饰器 使用。
- 记录了用户未登录时访问的地址信息，可以帮助我们实现在用户登录成功后直接进入未登录时访问的地址。

```
# 响应登录结果
next = request.GET.get('next')
if next:
    response = redirect(next)
else:
    response = redirect(reverse('contents:index'))
```



4. 知识要点

- 判断用户是否登录依然使用状态保持信息实现。
- 登录时next参数的作用是为了方便用户从哪里进入到登录页面，登录成功后就回到哪里。

QQ登录

QQ登录

QQ登录开发文档

QQ登录：即我们所说的第三方登录，是指用户可以不在本项目中输入密码，而直接通过第三方的验证，成功登录本项目。

1. QQ互联开发者申请步骤

若想实现QQ登录，需要成为QQ互联的开发者，审核通过才可实现。

- 相关连
接：<http://wiki.connect.qq.com/%E6%88%90%E4%B8%BA%E5%BC%80%E5%8F%91%E8%80%85>

2. QQ互联应用申请步骤

成为QQ互联开发者后，还需创建应用，即获取本项目对应与QQ互联的应用ID。

- 相关连接：http://wiki.connect.qq.com/_trashed-2

3. 网站对接QQ登录步骤

QQ互联提供有开发文档，帮助开发者实现QQ登录。

- 相关连
接：http://wiki.connect.qq.com/%E5%87%86%E5%A4%87%E5%B7%A5%E4%BD%9C_oauth2-0

4. QQ登录流程分析



5. 知识要点

- 当我们在对接第三方平台的接口时，一定要认真阅读第三方平台提供的文档。文档中一定会有接口的使用说明，方便我们开发。

定义QQ登录模型类

QQ登录成功后，我们需要将QQ用户和美多商城用户关联到一起，方便下次QQ登录时使用，所以我们选择使用MySQL数据库进行存储。

1. 定义模型类基类

为了给项目中模型类补充数据 `创建时间` 和 `更新时间` 两个字段，我们需要定义模型类基类。在 `meiduo_mall.utils/models.py` 文件中创建模型类基类。

```
from django.db import models

class BaseModel(models.Model):
    """为模型类补充字段"""

    create_time = models.DateTimeField(auto_now_add=True, verbose_name="创建时间")
    update_time = models.DateTimeField(auto_now=True, verbose_name="更新时间")

    class Meta:
        abstract = True # 说明是抽象模型类，用于继承使用，数据库迁移时不会创建BaseModel的表
```

2. 定义QQ登录模型类

创建一个新的应用 `oauth`，用来实现QQ第三方认证登录。

```
# oauth
url(r'^', include('oauth.urls')),
```

在 `oauth/models.py` 中定义QQ身份（`openid`）与用户模型类User的关联关系

```
from django.db import models

from meiduo_mall.utils.models import BaseModel
# Create your models here.

class OAuthQQUser(BaseModel):
    """QQ登录用户数据"""
    user = models.ForeignKey('users.User', on_delete=models.CASCADE, verbose_name='用户')
    openid = models.CharField(max_length=64, verbose_name='openid', db_index=True)

    class Meta:
        db_table = 'tb_oauth_qq'
        verbose_name = 'QQ登录用户数据'
```

```
verbose_name_plural = verbose_name
```

3. 迁移QQ登录模型类

```
$ python manage.py makemigrations  
$ python manage.py migrate
```



id	create_time	update_time	openid	user_id
1	2018-09-09 09:00:51.976388	2018-09-09 09:00:51.976599	173F141D1EE7DE3558E910446..	10

QQ登录工具QQLoginTool

1. QQLoginTool介绍

- 该工具封装了QQ登录时对接QQ互联接口的请求操作。可用于快速实现QQ登录。

2. QQLoginTool安装

```
pip install QQLoginTool
```

3. QQLoginTool使用说明

1.导入

```
from QQLoginTool.QQtool import OAuthQQ
```

2.初始化 OAuthQQ对象

```
oauth = OAuthQQ(client_id=settings.QQ_CLIENT_ID, client_secret=settings.QQ_CLIENT_SECRET, redirect_uri=settings.QQ_REDIRECT_URI, state=next)
```

3.获取QQ登录扫码页面，扫码后得到 Authorization Code

```
login_url = oauth.get_qq_url()
```

4.通过 Authorization Code 获取 Access Token

```
access_token = oauth.get_access_token(code)
```

5.通过 Access Token 获取 OpenID

```
openid = oauth.get_open_id(access_token)
```

OAuth2.0认证获取openid

1. 获取QQ登录扫码页面

1. 请求方式

选项	方案
请求方法	GET
请求地址	/qq/login/

2. 请求参数：查询参数

参数名	类型	是否必传	说明
next	string	否	用于记录QQ登录成功后进入的网址

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息
login_url	QQ登录扫码页面链接

4. 后端逻辑实现

```
class QQAuthURLView(View):
    """提供QQ登录页面网址
    https://graph.qq.com/oauth2.0/authorize?response_type=code&client_id=xxx&redirect_uri=xxx&state=xxx
    """
    def get(self, request):
        # next表示从哪个页面进入到的登录页面，将来登录成功后，就自动回到那个页面
        next = request.GET.get('next')

        # 获取QQ登录页面网址
        oauth = OAuthQQ(client_id=settings.QQ_CLIENT_ID, client_secret=settings.QQ_CLIENT_SECRET,
                        redirect_uri=settings.QQ_REDIRECT_URI, state=next)
        login_url = oauth.get_qq_url()

        return http.JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'login_url': login_url})
```

5. QQ登录参数

```
QQ_CLIENT_ID = '101518219'
```

```
QQ_CLIENT_SECRET = '418d84ebdc7241efb79536886ae95224'
QQ_REDIRECT_URI = 'http://www.meiduo.site:8000/oauth_callback'
```

QQ登录扫码后，待处理的业务逻辑

```
# 提取code请求参数
# 使用code向QQ服务器请求access_token
# 使用access_token向QQ服务器请求openid
# 使用openid查询该QQ用户是否在美多商城中绑定过用户
# 如果openid已绑定美多商城用户，直接登入到美多商城系统
# 如果openid没绑定美多商城用户，创建用户并绑定到openid
```

2. 接收Authorization Code

提示：

- 用户在QQ登录成功后，QQ会将用户重定向到我们配置的回调网址。
- 在QQ重定向到回调网址时，会传给我们一个 Authorization Code。
- 我们需要拿到 Authorization Code 并完成OAuth2.0认证获取openid。
- 在本项目中，我们申请QQ登录开发资质时配置的回调网址为：
 - `http://www.meiduo.site:8000/oauth_callback`
- QQ互联重定向的完整网址为：
 - `http://www.meiduo.site:8000/oauth_callback/?code=AE263F12675FA79185B54870D79730A7&state=%2F`

```
class QQAuthUserView(View):
    """用户扫码登录的回调处理"""

    def get(self, request):
        """OAuth2.0认证"""
        # 接收Authorization Code
        code = request.GET.get('code')
        if not code:
            return http.HttpResponseForbidden('缺少code')
        pass
```

```
url(r'^oauth_callback/$', views.QQAuthUserView.as_view()),
```

3. OAuth2.0认证获取openid

- 使用code向QQ服务器请求access_token
- 使用access_token向QQ服务器请求openid

```
class QQAuthUserView(View):
    """用户扫码登录的回调处理"""
```

```

def get(self, request):
    """OAuth2.0认证"""
    # 提取code请求参数
    code = request.GET.get('code')
    if not code:
        return http.HttpResponseForbidden('缺少code')

    # 创建工具对象
    oauth = OAuthQQ(client_id=settings.QQ_CLIENT_ID, client_secret=settings.QQ_CLIENT_SECRET, redirect_uri=settings.QQ_REDIRECT_URI)

    try:
        # 使用code向QQ服务器请求access_token
        access_token = oauth.get_access_token(code)

        # 使用access_token向QQ服务器请求openid
        openid = oauth.get_open_id(access_token)
    except Exception as e:
        logger.error(e)
        return http.HttpResponseServerError('OAuth2.0认证失败')
    pass

```

4. 本机绑定www.meiduo.site域名

1.ubuntu系统或者Mac系统

编辑 /etc/hosts

```

127.0.0.1 localhost
127.0.1.1 ubuntu

# The following lines are desirable for IPv6 capable hosts
::1      ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters
0.0.0.0 account.jetbrains.com
127.0.0.1 www.meiduo.site                                         ubuntu系统

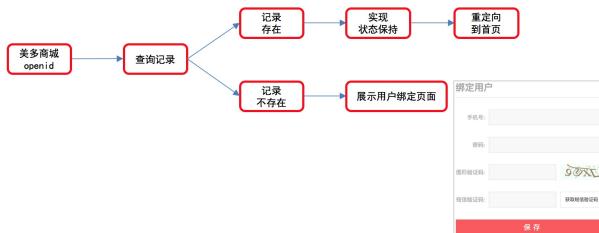
## 
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1      localhost
255.255.255.255 broadcasthost
::1            localhost
0.0.0.0 account.jetbrains.com
127.0.0.1      www.meiduo.site                                     Mac系统

```

2.Windows系统

编辑 C:\Windows\System32\drivers\etc\hosts

openid是否绑定用户的处理



1. 判断openid是否绑定过用户

使用openid查询该QQ用户是否在美多商城中绑定过用户。

```

try:
    oauth_user = OAuthQQUser.objects.get(openid=openid)
except OAuthQQUser.DoesNotExist:
    # 如果openid没绑定美多商城用户
    pass
else:
    # 如果openid已绑定美多商城用户
    pass

```

2. openid已绑定用户的处理

如果openid已绑定美多商城用户，直接生成状态保持信息，登录成功，并重定向到首页。

```

try:
    oauth_user = OAuthQQUser.objects.get(openid=openid)
except OAuthQQUser.DoesNotExist:
    # 如果openid没绑定美多商城用户
    pass
else:
    # 如果openid已绑定美多商城用户
    # 实现状态保持
    qq_user = oauth_user.user
    login(request, qq_user)

    # 响应结果
    next = request.GET.get('state')
    response = redirect(next)

    # 登录时用户名写入到cookie，有效期15天
    response.set_cookie('username', qq_user.username, max_age=3600 * 24 * 15)

return response

```

3. openid未绑定用户的处理

- 为了能够在后续的绑定用户操作中前端可以使用openid，在这里将openid签名后响应给前端。
- openid属于用户的隐私信息，所以需要将openid签名处理，避免暴露。

```

try:
    oauth_user = OAuthQQUser.objects.get(openid=openid)
except OAuthQQUser.DoesNotExist:
    # 如果openid没绑定美多商城用户
    access_token_openid = generate_eccess_token(openid)
    context = {'access_token_openid': access_token_openid}
    return render(request, 'oauth_callback.html', context)
else:
    # 如果openid已绑定美多商城用户
    # 实现状态保持
    qq_user = oauth_user.user
    login(request, qq_user)

    # 响应结果
    next = request.GET.get('state')
    response = redirect(next)

    # 登录时用户名写入到cookie，有效期15天
    response.set_cookie('username', qq_user.username, max_age=3600 * 24 * 15)

return response

```

oauth_callback.html 中渲染 access_token

```
<input type="hidden" name="access_token_openid" value="{{ access_token_openid }}>
```

4. 补充itsdangerous的使用

- itsdangerous 模块的参考资料链接 <http://itsdangerous.readthedocs.io/en/latest/>
- 安装： pip install itsdangerous
- TimedJSONWebSignatureSerializer 的使用
 - 使用 TimedJSONWebSignatureSerializer 可以生成带有有效期的 token

```
In [1]: from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
In [2]: s = Serializer('secret_key', 600) ② 创建序列化器对象
        1.导入安装的itsdangerous
In [3]: data = {'openid': '426459AD133CBF5B062AEA2AE89DB365'} ③ 准备要序列化的数据
In [4]: token = s.dumps(data) ④ 序列化数据

In [5]: token ⑤ 序列化后的数据
Out[5]: b'eyJhbGciOiJIUzI1NiIsInlhdcIEMTU1NzgwODA4OSwiZXhwIjoxNTU3ODA4Njg5fQ.eyJvcGVuaWQiO
J9.bTZJvy1gWj3Kts9o_2svg-F2Kfx958b8-hwK-DmYoh6H4IDgcGyh12EnlwbtSUzh_u9mayKhr9sr7Nks039zow'

In [6]: s = Serializer('secret_key', 600) ⑥ 创建相同的序列化器对象
In [7]: data = s.loads(token) ⑦ 反序列化数据

In [8]: data ⑧ 反序列化后的数据
Out[8]: {'openid': '426459AD133CBF5B062AEA2AE89DB365'}
```

补充：openid签名处理

- oauth.utils.py

```
def generate_eccess_token(openid):
    """
    签名openid
    :param openid: 用户的openid
    :return: access_token
    """
    serializer = Serializer(settings.SECRET_KEY, expires_in=constants.ACCESS_TOKEN_EXPIRES)
    data = {'openid': openid}
    token = serializer.dumps(data)
    return token.decode()
```

openid绑定用户实现

类似于用户注册的业务逻辑

- 当用户输入的手机号对应的用户已存在
 - 直接将该已存在用户跟 openid 绑定
- 当用户输入的手机号对应的用户不存在
 - 新建一个用户，并跟 openid 绑定

```
class QQAuthUserView(View):
    """用户扫码登录的回调处理"""

    def get(self, request):
        """Oauth2.0认证"""
        .....

    def post(self, request):
        """美多商城用户绑定到openid"""
        # 接收参数
        mobile = request.POST.get('mobile')
        password = request.POST.get('password')
        sms_code_client = request.POST.get('sms_code')
        access_token_openid = request.POST.get('access_token_openid')

        # 校验参数
        # 判断参数是否齐全
        if not all([mobile, password, sms_code_client]):
            return http.HttpResponseForbidden('缺少必传参数')
        # 判断手机号是否合法
        if not re.match(r'^1[3-9]\d{9}$', mobile):
            return http.HttpResponseForbidden('请输入正确的手机号码')
        # 判断密码是否合格
        if not re.match(r'^[0-9A-Za-z]{8,20}$', password):
            return http.HttpResponseForbidden('请输入8-20位的密码')
        # 判断短信验证码是否一致
        redis_conn = get_redis_connection('verify_code')
        sms_code_server = redis_conn.get('sms_%s' % mobile)
        if sms_code_server is None:
            return render(request, 'oauth_callback.html', {'sms_code_errmsg': '无效的短信验证码'})
        if sms_code_client != sms_code_server.decode():
            return render(request, 'oauth_callback.html', {'sms_code_errmsg': '输入短信验证码有误'})
        # 判断openid是否有效：错误提示放在sms_code_errmsg位置
        openid = check_access_token(access_token_openid)
        if not openid:
            return render(request, 'oauth_callback.html', {'openid_errmsg': '无效的openid'})
```

```

# 保存注册数据
try:
    user = User.objects.get(mobile=mobile)
except User.DoesNotExist:
    # 用户不存在,新建用户
    user = User.objects.create_user(username=mobile, password=password, mobile=mobile)
else:
    # 如果用户存在, 检查用户密码
    if not user.check_password(password):
        return render(request, 'oauth_callback.html', {'account_errmsg': '用户名或密码错误'})

# 将用户绑定openid
try:
    OAuthQQUser.objects.create(openid=openid, user=user)
except DatabaseError:
    return render(request, 'oauth_callback.html', {'qq_login_errmsg': 'QQ登录失败'})

# 实现状态保持
login(request, user)

# 响应绑定结果
next = request.GET.get('state')
response = redirect(next)

# 登录时用户名写入到cookie, 有效期15天
response.set_cookie('username', user.username, max_age=3600 * 24 * 15)

return response

```

反序列化access_token_openid

```

def check_access_token(access_token_openid):
    """
    反解、反序列化access_token_openid
    :param access_token_openid: openid密文
    :return: openid明文
    """

    # 创建序列化器对象: 序列化和反序列化的对象的参数必须是一模一样的
    s = Serializer(settings.SECRET_KEY, constants.ACCESS_TOKEN_EXPIRES)

    # 反序列化openid密文
    try:
        data = s.loads(access_token_openid)
    except BadData: # openid密文过期
        return None
    else:

```

```
# 返回openid明文  
return data.get('openid')
```

用户中心

用户基本信息

用户基本信息逻辑分析

1. 用户基本信息逻辑分析

基本信息

用户名:	itcast
联系方式:	17600992166
Email:	<input type="text"/> 保存 取消

基本信息

用户名:	itcast
联系方式:	17600992166
Email:	<input type="text" value="zhangesharp@163.com"/> 待验证 已发送验证邮件

基本信息

用户名:	itcast
联系方式:	17600992166
Email:	<input type="text" value="zhangesharp@163.com"/> 待验证 重新发送验证邮件

基本信息

用户名:	itcast
联系方式:	17600992166
Email:	<input type="text" value="zhangesharp@163.com"/> 已验证

以下是要实现的后端逻辑

1. 用户模型补充 `email_active` 字段
2. 查询并渲染用户基本信息
3. 添加邮箱
4. 发送邮箱验证邮件
5. 验证邮箱

提示:

- 用户添加邮箱时，界面的局部刷新，我们选择使用 `Vue.js` 来实现。

查询并渲染用户基本信息

1. 用户模型补充email_active字段

- 由于在渲染用户基本信息时，需要渲染用户邮箱验证的状态，所以需要给用户模型补充 email_active 字段
- 补充完字段后，需要进行迁移。

```
$ python manage.py makemigrations
$ python manage.py migrate
```

```
class User(AbstractUser):
    """自定义用户模型类"""
    mobile = models.CharField(max_length=11, unique=True, verbose_name='手机号')
    email_active = models.BooleanField(default=False, verbose_name='邮箱验证状态')

    class Meta:
        db_table = 'tb_users'
        verbose_name = '用户'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.username
```

2. 查询用户基本信息

```
class UserInfoView(LoginRequiredMixin, View):
    """用户中心"""

    def get(self, request):
        """提供个人信息界面"""
        context = {
            'username': request.user.username,
            'mobile': request.user.mobile,
            'email': request.user.email,
            'email_active': request.user.email_active
        }
        return render(request, 'user_center_info.html', context)
```

3. 渲染用户基本信息

1. 将后端模板数据传递到Vue.js

- 为了方便实现用户添加邮箱时的界面局部刷新

- 我们将后端提供的用户数据传入到 `user_center_info.js` 中

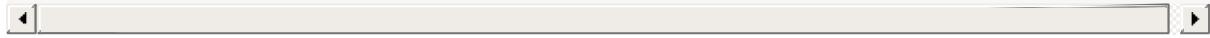
```
<script type="text/javascript">
    let username = "{{ username }}";
    let mobile = "{{ mobile }}";
    let email = "{{ email }}";
    let email_active = "{{ email_active }}";
</script>
<script type="text/javascript" src="{{ static('js/common.js') }}"></script>
<script type="text/javascript" src="{{ static('js/user_center_info.js') }}"></script>
>
```

```
data: {
    username: username,
    mobile: mobile,
    email: email,
    email_active: email_active,
},
```

2.Vue渲染用户基本信息: `user_center_info.html`

```
<div class="info_con clearfix" v-cloak>
    <h3 class="common_title2">基本信息</h3>
    <ul class="user_info_list">
        <li><span>用户名: </span>[[ username ]]</li>
        <li><span>联系方式: </span>[[ mobile ]]</li>
        <li>
            <span>Email: </span>
            <div v-if="set_email">
                <input v-model="email" @blur="check_email" type="email" name="email" class="email">
                <input @click="save_email" type="button" name="" value="保 存">
                <input @click="cancel_email" type="reset" name="" value="取 消">
                <div v-show="error_email" class="error_email_tip">邮箱格式错误</div>
            </div>
            <div v-else>
                <input v-model="email" type="email" name="email" class="email" read
only>
                <div v-if="email_active">
                    已验证
                </div>
                <div v-else>
                    待验证<input @click="save_email" :disabled="send_email_btn_disabled" type="button" :value="send_email_tip">
                </div>
            </div>
        </li>
    </ul>
</div>
```

```
</ul>
</div>
```



添加和验证邮箱

添加邮箱后端逻辑

1. 添加邮箱接口设计和定义

1. 请求方式

选项	方案
请求方法	PUT
请求地址	/emails/

2. 请求参数

参数名	类型	是否必传	说明
email	string	是	邮箱

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息

2. 添加邮箱后端逻辑实现

```

class EmailView(LoginRequiredMixin, View):
    """添加邮箱"""

    def put(self, request):
        """实现添加邮箱逻辑"""
        # 接收参数
        json_str = request.body.decode()
        json_dict = json.loads(json_str)
        email = json_dict.get('email')

        # 校验参数
        if not re.match(r'^[a-zA-Z][\w\.-]*@[a-zA-Z\-.]+\.\w{2,5}\w{1,2}$', email):
            return http.HttpResponseForbidden('参数email有误')

        # 赋值email字段
        try:
            request.user.email = email
            request.user.save()
        except Exception as e:
            logger.error(e)
            return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '添加邮箱失败'})
    
```

```

    })

    # 响应添加邮箱结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '添加邮箱成功'})

```

3. 判断用户是否登录并返回JSON

重要提示：

- 只有用户登录时才能让其绑定邮箱。
- 此时前后端交互的数据类型是JSON，所以需要判断用户是否登录并返回JSON给用户。

实现方案：自定义 `LoginRequiredJSONMixin` 扩展类

- 新建 `meiduo_mall/meiduo_mall/utils/views.py` 文件

```

class LoginRequiredJSONMixin(LoginRequiredMixin):
    """Verify that the current user is authenticated."""

    def handle_no_permission(self):
        return http.JsonResponse({'code': RETCODE.SESSIONERR, 'errmsg': '用户未登录'})
)

```

`handle_no_permission`说明：我们只需要改写父类中的处理方式 至于如何判断用户是否登录 在父类中已经判断了

`LoginRequiredJSONMixin` 的使用

```

from meiduo_mall.utils.views import LoginRequiredJSONMixin

class EmailView(LoginRequiredJSONMixin, View):
    """添加邮箱"""

    def put(self, request):
        """实现添加邮箱逻辑"""
        # 判断用户是否登录并返回JSON
        pass

```

Django发送邮件的配置



1. Django发送邮件流程分析



send_mail() 方法介绍

- 位置：
 - 在 `django.core.mail` 模块提供了 `send_mail()` 来发送邮件。
- 方法参数：
 - `send_mail(subject, message, from_email, recipient_list, html_message=None)`

`subject` 邮件标题
`message` 普通邮件正文，普通字符串
`from_email` 发件人
`recipient_list` 收件人列表
`html_message` 多媒体邮件正文，可以是html字符串

2. 准备发邮件服务器

1.点击进入《设置》界面



2.点击进入《客户端授权密码》界面



3.开启《授权码》，并完成验证短信



4.填写《授权码》



5.完成《授权码》设置



6.配置邮件服务器

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' # 指定邮件后端
EMAIL_HOST = 'smtp.163.com' # 发邮件主机
EMAIL_PORT = 25 # 发邮件端口
EMAIL_HOST_USER = 'hmmeiduo@163.com' # 授权的邮箱
EMAIL_HOST_PASSWORD = 'hmmeiduo123' # 邮箱授权时获得的密码，非注册登录密码
EMAIL_FROM = '美多商城<hmmeiduo@163.com>' # 发件人抬头
```

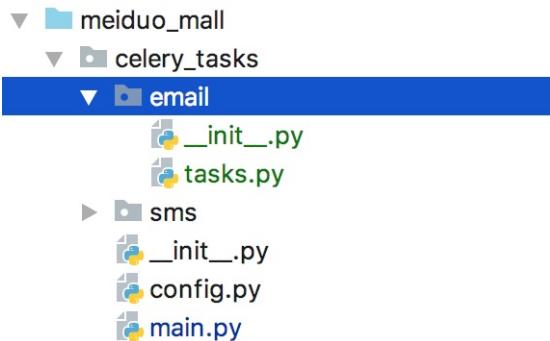
发送邮箱验证邮件

重要提示：

- 发送邮箱验证邮件是耗时的操作，不能阻塞美多商城的响应，所以需要异步发送邮件。
- 我们继续使用Celery实现异步任务。

1. 定义和调用发送邮件异步任务

1. 定义发送邮件任务



```

# bind: 保证task对象会作为第一个参数自动传入
# name: 异步任务别名
# retry_backoff: 异常自动重试的时间间隔 第n次(retry_backoff×2^(n-1))s
# max_retries: 异常自动重试次数的上限
@celery_app.task(bind=True, name='send_verify_email', retry_backoff=3)
def send_verify_email(self, to_email, verify_url):
    """
    发送验证邮箱邮件
    :param to_email: 收件人邮箱
    :param verify_url: 验证链接
    :return: None
    """

    subject = "美多商城邮箱验证"
    html_message = '<p>尊敬的用户您好! </p>' \
                  '<p>感谢您使用美多商城。</p>' \
                  '<p>您的邮箱为: %s 。请点击此链接激活您的邮箱: </p>' \
                  '<p><a href="%s">%s</a></p>' % (to_email, verify_url, verify_url)

    try:
        send_mail(subject, "", settings.EMAIL_FROM, [to_email], html_message=html_message)
    except Exception as e:
        logger.error(e)
        # 有异常自动重试三次
        raise self.retry(exc=e, max_retries=3)
  
```

2. 注册发邮件的任务：main.py

- 在发送邮件的异步任务中，我们用到了Django的配置文件。
- 所以我们需要修改celery的启动文件main.py。
- 在其中指明celery可以读取的Django配置文件。
- 最后记得注册新添加的email的任务

```
# celery启动文件
from celery import Celery

# 为celery使用django配置文件进行设置
import os
if not os.getenv('DJANGO_SETTINGS_MODULE'):
    os.environ['DJANGO_SETTINGS_MODULE'] = 'meiduo_mall.settings.dev'

# 创建celery实例
celery_app = Celery('meiduo')

# 加载celery配置
celery_app.config_from_object('celery_tasks.config')

# 自动注册celery任务
celery_app.autodiscover_tasks(['celery_tasks.sms', 'celery_tasks.email'])
```

3.调用发送邮件异步任务

```
# 赋值email字段
try:
    request.user.email = email
    request.user.save()
except Exception as e:
    logger.error(e)
    return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '添加邮箱失败'})

# 异步发送验证邮件
verify_url = '邮件验证链接'
send_verify_email.delay(email, verify_url)

# 响应添加邮箱结果
return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '添加邮箱成功'})
```

4.启动Celery

```
$ celery -A celery_tasks.main worker -l info
```

2.生成邮箱验证链接

邮箱验证链接

- <http://www.meiduo.site:8000/emails/verification/?token=eyJhbGciOiJIUzUxMjlsImhdCl6MTU1Nzg5NTA2NSwiZXhwIjoxNTU3OTgxNDY1fQ.eyJ1c2VyX2lkIjoxLCJlbWFpbCI6InpoYW5namllc2hhcnBAMTYzLmNvbSJ9.JBRjoAgZMbMrfrTdmhPyJy2gVMPvRe9bAsxmQr5uzADZo3mZhr9d5MjsVrSI9BJagg31UwpvlvuL5iZdPRi4qw>

1. 定义生成邮箱验证链接方法

```
def generate_verify_email_url(user):
    """
    生成邮箱验证链接
    :param user: 当前登录用户
    :return: verify_url
    """

    serializer = Serializer(settings.SECRET_KEY, expires_in=constants.VERIFY_EMAIL_TOKEN_EXPIRES)
    data = {'user_id': user.id, 'email': user.email}
    token = serializer.dumps(data).decode()
    verify_url = settings.EMAIL_VERIFY_URL + '?token=' + token
    return verify_url
```

2. 配置相关参数

```
# 邮箱验证链接
EMAIL_VERIFY_URL = 'http://www.meiduo.site:8000/emails/verification/'
```

3. 使用邮箱验证链接

```
verify_url = generate_verify_email_url(request.user)
send_verify_email.delay(email, verify_url)
```

3. 补充celery worker的工作模式

- 默认是进程池方式，进程数以当前机器的CPU核数为参考，每个CPU开四个进程。
- 如何自己指定进程数： celery worker -A proj --concurrency=4
- 如何改变进程池方式为协程方式： celery worker -A proj --concurrency=1000 -P eventlet -c 1000

```
# 安装eventlet模块
$ pip install eventlet

# 启用 Eventlet 池
$ celery -A celery_tasks.main worker -l info -P eventlet -c 1000
```

```
(root@ocean:~/) python3.6/projects/miduo_project/miduo/miduo/celery -A celery_tasks.main worker -l info -f /tmp/celery.log --loglevel=info
--> celery@ocean:~/local/v2.2.1 (celery[main])
--> * * * * * --> Service[27.3.8-0.0.0-2348-44911 2018-02-21 15:47:42]
--> * * * * * --> [config]
--> * * * * * --> [transport: amqp://guest:@192.168.100.100:5672//]
--> * * * * * --> [consumers: 1000 (current)]
--> * * * * * --> [task events: OFF (enable <-t> to monitor tasks in this worker)]
--> * * * * * --> [exchange: celery] [direct] key=celery
(tasks)
--> [celery_send_sms_code]
[2018-02-21 15:47:23,423: INFO:miduoProcess] Connected to amqp://guest:@192.168.100:5672// (vhost:None)
[2018-02-21 15:47:23,423: INFO:miduoProcess] Single connection for celery
[2018-02-21 15:47:23,423: INFO:miduoProcess] Using connection to amqp://guest:@192.168.100:5672// (vhost:None)
[2018-02-21 15:47:23,423: INFO:miduoProcess] celery@ocean.local ready
```

验证邮箱后端逻辑

1. 验证邮箱接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/emails/verification/

2. 请求参数：查询参数

参数名	类型	是否必传	说明
token	string	是	邮箱激活链接

3. 响应结果：HTML

字段	说明
邮箱验证失败	响应错误提示
邮箱验证成功	重定向到用户中心

2. 验证链接提取用户信息

```

def check_verify_email_token(token):
    """
    验证token并提取user
    :param token: 用户信息签名后的结果
    :return: user, None
    """

    serializer = Serializer(settings.SECRET_KEY, expires_in=constants.VERIFY_EMAIL_TOKEN_EXPIRES)
    try:
        data = serializer.loads(token)
    except BadData:
        return None
    else:
        user_id = data.get('user_id')
        email = data.get('email')
        try:
            user = User.objects.get(id=user_id, email=email)
        except User.DoesNotExist:
            return None
        else:
            return user
    
```

3. 验证邮箱后端逻辑实现

验证邮箱的核心：就是将用户的 `email_active` 字段设置为 `True`

```
class VerifyEmailView(View):
    """验证邮箱"""

    def get(self, request):
        """实现邮箱验证逻辑"""
        # 接收参数
        token = request.GET.get('token')

        # 校验参数：判断token是否为空和过期，提取user
        if not token:
            return http.HttpResponseBadRequest('缺少token')

        user = check_verify_email_token(token)
        if not user:
            return http.HttpResponseForbidden('无效的token')

        # 修改email_active的值为True
        try:
            user.email_active = True
            user.save()
        except Exception as e:
            logger.error(e)
            return http.HttpResponseServerError('激活邮件失败')

        # 返回邮箱验证结果
        return redirect(reverse('users:info'))
```

收货地址

新增收货地址 你已创建了2个收货地址，最多可创建20个

传智播客 北京		
收货人：传智播客		
所在地址：北京市昌平区		
地址：建材城西路		
手机：158****0001		
固定电话：78912345		
电子邮箱：hmmeliduo@163.com		

编辑

传智播客 北京		
收货人：传智播客		
所在地址：北京市昌平区		
地址：建材城西路		
手机：158****0001		
固定电话：78912345		
电子邮箱：hmmeliduo@163.com		

设为默认 **编辑**

新增收货地址 你已创建了2个收货地址，最多可创建20个

传智播客 北京			
收货人：传智播客	新增收货地址		
所在地址：北京	<input type="text"/>	北京	<input type="text"/>
地址：	<input type="text"/>		
手机：15	<input type="text"/>	<input type="text"/>	<input type="text"/>
固定电话：78	<input type="text"/>	<input type="text"/>	<input type="text"/>
电子邮箱：h	<input type="text"/>	<input type="text"/>	<input type="text"/>

新增 **取消**

传智播客 北京		
收货人：传智播客		
所在地址：北京		
地址：建材城西路		
手机：158****0001		
固定电话：78912345		
电子邮箱：hmmeliduo@163.com		

设为默认 **编辑**

用户地址的主要业务逻辑有：

1. 展示省市区数据
 2. 用户地址的增删改查处理
 3. 设置默认地址
 4. 设置地址标题

省市区三级联动

1. 展示收货地址页面

提示：

- 省市区数据是在收货地址界面展示的，所以我们先渲染出收货地址界面。
- 收货地址界面中基础的交互已经提前实现。

```
class AddressView(LoginRequiredMixin, View):
    """用户收货地址"""

    def get(self, request):
        """提供收货地址页面"""
        return render(request, 'user_center_site.html')
```

2. 准备省市区模型和数据

```
class Area(models.Model):
    """省市区"""
    name = models.CharField(max_length=20, verbose_name='名称')
    parent = models.ForeignKey('self', on_delete=models.SET_NULL, related_name='subs',
                               null=True, blank=True, verbose_name='上级行政区划')

    class Meta:
        db_table = 'tb_areas'
        verbose_name = '省市区'
        verbose_name_plural = '省市区'

    def __str__(self):
        return self.name
```

模型说明：

- 自关联字段的外键指向自身，所以 `models.ForeignKey('self')`
- 使用 `related_name` 指明父级查询子级数据的语法
 - 默认 `Area`模型类对象.`area_set` 语法
- `related_name='subs'`
 - 现在 `Area`模型类对象.`subs` 语法

导入省市区数据

```
mysql -h数据库ip地址 -u数据库用户名 -p数据库密码 数据库 < areas.sql
mysql -h127.0.0.1 -uroot -pmysql meiduo_mall < areas.sql
```

3. 查询省市区数据

1. 请求方式

选项	方案
请求方法	GET
请求地址	/areas/

2. 请求参数：查询参数

- 如果前端没有传入 `area_id`，表示用户需要省份数据
- 如果前端传入了 `area_id`，表示用户需要市或区数据

参数名	类型	是否必传	说明
<code>area_id</code>	string	否	地区ID

3. 响应结果：JSON

- 省份数据

```
{
  "code": "0",
  "errmsg": "OK",
  "province_list": [
    {
      "id": 110000,
      "name": "北京市"
    },
    {
      "id": 120000,
      "name": "天津市"
    },
    {
      "id": 130000,
      "name": "河北省"
    },
    .....
  ]
}
```

- 市或区数据

```
{
  "code": "0",
  "errmsg": "OK",
  "sub_data": {
    "id": 130000,
    "name": "河北省",
  }
}
```

```

    "subs": [
      {
        "id": 130100,
        "name": "石家庄市"
      },
      .....
    ]
}

```

4.查询省市区数据后端逻辑实现

- 如果前端没有传入 `area_id`，表示用户需要省份数据
- 如果前端传入了 `area_id`，表示用户需要市或区数据

```

class AreasView(View):
    """省市区数据"""

    def get(self, request):
        """提供省市区数据"""
        area_id = request.GET.get('area_id')

        if not area_id:
            # 提供省份数据
            try:
                # 查询省份数据
                province_model_list = Area.objects.filter(parent__isnull=True)

                # 序列化省级数据
                province_list = []
                for province_model in province_model_list:
                    province_list.append({'id': province_model.id, 'name': province_model.name})

                # 响应省份数据
                return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'province_list': province_list})
            except Exception as e:
                logger.error(e)
                return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '省份数据错误'})
        else:
            # 提供市或区数据
            try:
                parent_model = Area.objects.get(id=area_id) # 查询市或区的父级
                sub_model_list = parent_model.subs.all()

                # 序列化市或区数据
                sub_list = []
                for sub_model in sub_model_list:

```

```

        sub_list.append({'id': sub_model.id, 'name': sub_model.name})

    sub_data = {
        'id': parent_model.id, # 父级pk
        'name': parent_model.name, # 父级name
        'subs': sub_list # 父级的子集
    }

    # 响应市或区数据
    return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'sub_data':
        sub_data})
except Exception as e:
    logger.error(e)
    return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '城市或区数据错
误'})

```

4. Vue渲染省市区数据

1. user_center_site.js 中

```

mounted() {
    // 获取省份数据
    this.get_provinces();
},

// 获取省份数据
get_provinces(){
    let url = '/areas/';
    axios.get(url, {
        responseType: 'json'
    })
    .then(response => {
        if (response.data.code == '0') {
            this.provinces = response.data.province_list;
        } else {
            console.log(response.data);
            this.provinces = [];
        }
    })
    .catch(error => {
        console.log(error.response);
        this.provinces = [];
    })
},

```

```

watch: {
    // 监听到省份id变化
}

```

```

'form_address.province_id': function(){
    if (this.form_address.province_id) {
        let url = '/areas/?area_id=' + this.form_address.province_id;
        axios.get(url, {
            responseType: 'json'
        })
        .then(response => {
            if (response.data.code == '0') {
                this.cities = response.data.sub_data.subs;
            } else {
                console.log(response.data);
                this.cities = [];
            }
        })
        .catch(error => {
            console.log(error.response);
            this.cities = [];
        })
    }
},
// 监听到城市id变化
'form_address.city_id': function(){
    if (this.form_address.city_id){
        let url = '/areas/?area_id=' + this.form_address.city_id;
        axios.get(url, {
            responseType: 'json'
        })
        .then(response => {
            if (response.data.code == '0') {
                this.districts = response.data.sub_data.subs;
            } else {
                console.log(response.data);
                this.districts = [];
            }
        })
        .catch(error => {
            console.log(error.response);
            this.districts = [];
        })
    }
},
}
,
```

2. user_center_site.html 中

```

<div class="form_group">
    <label>*所在地区: </label>
    <select v-model="form_address.province_id">
        <option v-for="province in provinces" :value="province.id">[ [ province.name
    ]]</option>

```

```

</select>
<select v-model="form_address.city_id">
    <option v-for="city in cities" :value="city.id">[ [ city.name ] ]</option>
</select>
<select v-model="form_address.district_id">
    <option v-for="district in districts" :value="district.id">[ [ district.name ] ]</option>
</select>
</div>

```

5. 缓存省市区数据

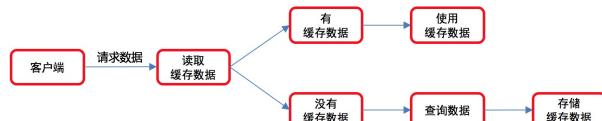
提示：

- 省市区数据是我们动态查询的结果。
- 但是省市区数据不是频繁变化的数据，所以没有必要每次都重新查询。
- 所以我们可以选择对省市区数据进行缓存处理。

1. 缓存工具

- from django.core.cache import cache
- 存储缓存数据：cache.set('key', 内容, 有效期)
- 读取缓存数据：cache.get('key')
- 删除缓存数据：cache.delete('key')
- 注意：存储进去和读取出来的数据类型相同，所以读取出来后可以直接使用。

2. 缓存逻辑



3. 缓存逻辑实现

- 省份缓存数据
 - cache.set('province_list', province_list, 3600)
- 市或区缓存数据
 - cache.set('sub_area_' + area_id, sub_data, 3600)

```

class AreasView(View):
    """省市区数据"""

    def get(self, request):
        """提供省市区数据"""
        area_id = request.GET.get('area_id')

        if not area_id:
            # 读取省份缓存数据
            province_list = cache.get('province_list')

```

```

    if not province_list:
        # 提供省份数据
        try:
            # 查询省份数据
            province_model_list = Area.objects.filter(parent__isnull=True)

            # 序列化省级数据
            province_list = []
            for province_model in province_model_list:
                province_list.append({'id': province_model.id, 'name': province_model.name})

            # 缓存省份数据
            cache.set('province_list', province_list, 3600)
        except Exception as e:
            logger.error(e)
            return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '省份数据错误'})

    # 响应省份数据
    return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'province_list': province_list})
else:
    # 读取市或区缓存数据
    sub_data = cache.get('sub_area_' + area_id)

    if not sub_data:
        # 提供市或区数据
        try:
            parent_model = Area.objects.get(id=area_id) # 查询市或区的父级
            sub_model_list = parent_model.subs.all()

            # 序列化市或区数据
            sub_list = []
            for sub_model in sub_model_list:
                sub_list.append({'id': sub_model.id, 'name': sub_model.name})
        except Exception as e:
            logger.error(e)
            return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '城市或区数据错误'})

        sub_data = {
            'id': parent_model.id, # 父级pk
            'name': parent_model.name, # 父级name
            'subs': sub_list # 父级的子集
        }
    except Exception as e:
        logger.error(e)
        return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '城市或区数据错误'})

    # 缓存市或区数据
    cache.set('sub_area_' + area_id, sub_data, 3600)

```

```
# 响应市或区数据
return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'sub_data': su
b_data})
```



新增地址后端逻辑

1. 定义用户地址模型类

1. 用户地址模型类

```
class Address(BaseModel):
    """用户地址"""
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='addresses', verbose_name='用户')
    title = models.CharField(max_length=20, verbose_name='地址名称')
    receiver = models.CharField(max_length=20, verbose_name='收货人')
    province = models.ForeignKey('areas.Area', on_delete=models.PROTECT, related_name='province_addresses', verbose_name='省')
    city = models.ForeignKey('areas.Area', on_delete=models.PROTECT, related_name='city_addresses', verbose_name='市')
    district = models.ForeignKey('areas.Area', on_delete=models.PROTECT, related_name='district_addresses', verbose_name='区')
    place = models.CharField(max_length=50, verbose_name='地址')
    mobile = models.CharField(max_length=11, verbose_name='手机')
    tel = models.CharField(max_length=20, null=True, blank=True, default='', verbose_name='固定电话')
    email = models.CharField(max_length=30, null=True, blank=True, default='', verbose_name='电子邮箱')
    is_deleted = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_address'
        verbose_name = '用户地址'
        verbose_name_plural = verbose_name
        ordering = ['-update_time']
```

2. Address 模型类说明

- Address 模型类中的外键指向 areas/models 里面的 Area。指明外键时，可以使用 应用名.模型类名 来定义。
- ordering 表示在进行 Address 查询时，默认使用的排序方式。
 - ordering = ['-update_time']：根据更新的时间倒序。

3. 补充用户模型默认地址字段

```
class User(AbstractUser):
    """自定义用户模型类"""
    mobile = models.CharField(max_length=11, unique=True, verbose_name='手机号')
    email_active = models.BooleanField(default=False, verbose_name='邮箱验证状态')
    default_address = models.ForeignKey('Address', related_name='users', null=True, blank=True, on_delete=models.SET_NULL, verbose_name='默认地址')
```

```

class Meta:
    db_table = 'tb_users'
    verbose_name = '用户'
    verbose_name_plural = verbose_name

    def __str__(self):
        return self.username

```

2. 新增地址接口设计和定义

1. 请求方式

选项	方案
请求方法	POST
请求地址	/addresses/create/

2. 请求参数: JSON

参数名	类型	是否必传	说明
receiver	string	是	收货人
province_id	string	是	省份ID
city_id	string	是	城市ID
district_id	string	是	区县ID
place	string	是	收货地址
mobile	string	是	手机号
tel	string	否	固定电话
email	string	否	邮箱

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息
id	地址ID
receiver	收货人
province	省份名称
city	城市名称
district	区县名称
place	收货地址

mobile	手机号
tel	固定电话
email	邮箱

3. 新增地址后端逻辑实现

提示：

- 用户地址数量有上限，最多20个，超过地址数量上限就返回错误信息

```
class CreateAddressView(LoginRequiredJSONMixin, View):
    """新增地址"""

    def post(self, request):
        """实现新增地址逻辑"""
        # 判断是否超过地址上限：最多20个
        # Address.objects.filter(user=request.user).count()
        count = request.user.addresses.count()
        if count >= constants.USER_ADDRESS_COUNTS_LIMIT:
            return http.JsonResponse({'code': RETCODE.THROTTLINGERR, 'errmsg': '超过地址数量上限'})

        # 接收参数
        json_dict = json.loads(request.body.decode())
        receiver = json_dict.get('receiver')
        province_id = json_dict.get('province_id')
        city_id = json_dict.get('city_id')
        district_id = json_dict.get('district_id')
        place = json_dict.get('place')
        mobile = json_dict.get('mobile')
        tel = json_dict.get('tel')
        email = json_dict.get('email')

        # 校验参数
        if not all([receiver, province_id, city_id, district_id, place, mobile]):
            return http.HttpResponseForbidden('缺少必传参数')
        if not re.match(r'^1[3-9]\d{9}$', mobile):
            return http.HttpResponseForbidden('参数mobile有误')
        if tel:
            if not re.match(r'^(\d{2,3}-)?(\d{2-3}\d{6,7})+(-\d{1,4})?$', tel):
                return http.HttpResponseForbidden('参数tel有误')
        if email:
            if not re.match(r'^[a-zA-Z][\w\.-]*@[a-zA-Z\-.]+\.\w{2,5}(\,\w{2,5})?$', email):
                return http.HttpResponseForbidden('参数email有误')

        # 保存地址信息
        try:
```

```

        address = Address.objects.create(
            user=request.user,
            title = receiver,
            receiver = receiver,
            province_id = province_id,
            city_id = city_id,
            district_id = district_id,
            place = place,
            mobile = mobile,
            tel = tel,
            email = email
        )

        # 设置默认地址
        if not request.user.default_address:
            request.user.default_address = address
            request.user.save()

    except Exception as e:
        logger.error(e)
        return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '新增地址失败'})

    # 新增地址成功，将新增的地址响应给前端实现局部刷新
    address_dict = {
        "id": address.id,
        "title": address.title,
        "receiver": address.receiver,
        "province": address.province.name,
        "city": address.city.name,
        "district": address.district.name,
        "place": address.place,
        "mobile": address.mobile,
        "tel": address.tel,
        "email": address.email
    }

    # 响应保存结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '新增地址成功', 'address': address_dict})

```

4. 新增地址前端逻辑实现

```

<form>
    <div class="form_group">
        <label>*收货人: </label>
        <input v-model="form_address.receiver" @blur="check_receiver" type="text" class="receiver">
        <span v-show="error_receiver" class="receiver_error">请填写收件人</span>
    </div>

```

```

<div class="form_group">
    <label>*所在地区: </label>
    <select v-model="form_address.province_id">
        <option v-for="province in provinces" :value="province.id">[[ province.name ]]</option>
    </select>
    <select v-model="form_address.city_id">
        <option v-for="city in cities" :value="city.id">[[ city.name ]]</option>
    </select>
    <select v-model="form_address.district_id">
        <option v-for="district in districts" :value="district.id">[[ district.name ]]</option>
    </select>
</div>
<div class="form_group">
    <label>*详细地址: </label>
    <input v-model="form_address.place" @blur="check_place" type="text" class="place">
        <span v-show="error_place" class="place_error">请填写地址信息</span>
</div>
<div class="form_group">
    <label>*手机: </label>
    <input v-model="form_address.mobile" @blur="check_mobile" type="text" class="mobile">
        <span v-show="error_mobile" class="mobile_error">手机信息有误</span>
</div>
<div class="form_group">
    <label>固定电话: </label>
    <input v-model="form_address.tel" @blur="check_tel" type="text" class="tel">
        <span v-show="error_tel" class="tel_error">固定电话有误</span>
</div>
<div class="form_group">
    <label>邮箱: </label>
    <input v-model="form_address.email" @blur="check_email" type="text" class="email">
        <span v-show="error_email" class="email_error">邮箱信息有误</span>
</div>
    <input @click="save_address" type="button" name="" value="新增" class="info_submit">
    <input @click="is_show_edit=false" type="reset" name="" value="取消" class="info_submit info_reset">
</form>

```

```

save_address(){
    if (this.error_receiver || this.error_place || this.error_mobile || this.error_email || !this.form_address.province_id || !this.form_address.city_id || !this.form_address.district_id ) {

```

```
    alert('信息填写有误! ');
} else {
    // 新增地址
    let url = '/addresses/create/';
    axios.post(url, this.form_address, {
        headers: {
            'X-CSRFToken':getCookie('csrftoken')
        },
        responseType: 'json'
    })
    .then(response => {
        if (response.data.code == '0') {
            // 局部刷新界面：展示所有地址信息

        } else if (response.data.code == '4101') {
            location.href = '/login/?next=/addresses/';
        } else {
            alert(response.dataerrmsg);
        }
    })
    .catch(error => {
        console.log(error.response);
    })
},
},
```

展示地址后端逻辑

1. 展示地址接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/addresses/

2. 请求参数

无

3. 响应结果: HTML

user_center_site.html

2. 展示地址后端逻辑实现

```
class AddressView(LoginRequiredMixin, View):
    """用户收货地址"""

    def get(self, request):
        """提供收货地址页面"""
        # 获取用户地址列表
        login_user = request.user
        addresses = Address.objects.filter(user=login_user, is_deleted=False)

        address_list = []
        for address in addresses:
            address_dict = {
                "id": address.id,
                "title": address.title,
                "receiver": address.receiver,
                "province": address.province.name,
                "city": address.city.name,
                "district": address.district.name,
                "place": address.place,
                "mobile": address.mobile,
                "tel": address.tel,
                "email": address.email
            }
            address_list.append(address_dict)
```

```

context = {
    'default_address_id': login_user.default_address_id,
    'addresses': address_list
}

return render(request, 'user_center_site.html', context)

```

3. 展示地址前端逻辑实现

1. 将后端模板数据传递到Vue.js

```

<script type="text/javascript">
    let addresses = {{ addresses | safe }};
    let default_address_id = {{ default_address_id }};
</script>

```

```

data: {
    addresses: JSON.parse(JSON.stringify(addresses)),
    default_address_id: default_address_id,
},

```

2. user_center_site.html 中渲染地址信息

```

<div class="right_content clearfix" v-cloak>
    <div class="site_top_con">
        <a @click="show_add_site">新增收货地址</a>
        <span>你已创建了<b>[[ addresses.length ]]</b>个收货地址, 最多可创建<b>20</b>个</span>
    </div>
    <div class="site_con" v-for="(address, index) in addresses">
        <div class="site_title">
            <div v-if="edit_title_index==index">
                <input v-model="new_title" type="text" name="">
                <input @click="save_title(index)" type="button" name="" value="保存">
            </div>
            <div>
                <h3>[[ address.title ]]</h3>
                <a @click="show_edit_title(index)" class="edit_title"></a>
            </div>
            <em v-if="address.id==default_address_id">默认地址</em>
            <span @click="delete_address(index)">x</span>
        </div>
        <ul class="site_list">
            <li><span>收货人: </span><b>[[ address.receiver ]]</b></li>
        </ul>
    </div>
</div>

```

```
<li><span>所在地区: </span><b>[[ address.province ]] [[address.city]] [[  
address.district ]]</b></li>  
<li><span>地址: </span><b>[[ address.place ]]</b></li>  
<li><span>手机: </span><b>[[ address.mobile ]]</b></li>  
<li><span>固定电话: </span><b>[[ address.tel ]]</b></li>  
<li><span>电子邮箱: </span><b>[[ address.email ]]</b></li>  
</ul>  
<div class="down_btn">  
    <a v-if="address.id!=default_address_id" @click="set_default(index)">设  
为默认</a>  
    <a @click="show_edit_site(index)" class="edit_icon">编辑</a>  
</div>  
</div>
```

3.完善 user_center_site.js 中成功新增地址后的局部刷新

```
if (response.data.code == '0') {  
    // 局部刷新界面: 展示所有地址信息, 将新的地址添加到头部  
    this.addresses.splice(0, 0, response.data.address);  
    this.is_show_edit = false;  
}
```

修改地址后端逻辑

1. 修改地址接口设计和定义

1. 请求方式

选项	方案
请求方法	PUT
请求地址	/addresses/(?P<address_id>\d+)/

2. 请求参数：路径参数 和 JSON

参数名	类型	是否必传	说明
address_id	string	是	要修改的地址ID（路径参数）
receiver	string	是	收货人
province_id	string	是	省份ID
city_id	string	是	城市ID
district_id	string	是	区县ID
place	string	是	收货地址
mobile	string	是	手机号
tel	string	否	固定电话
email	string	否	邮箱

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息
id	地址ID
receiver	收货人
province	省份名称
city	城市名称
district	区县名称
place	收货地址
mobile	手机号
tel	固定电话
email	邮箱

2. 修改地址后端逻辑实现

提示

- 修改地址后端逻辑和新增地址后端逻辑非常的相似，都是更新用户地址模型类，需要保存用户地址信息。

```
class UpdateDestroyAddressView(LoginRequiredMixin, View):
    """修改和删除地址"""

    def put(self, request, address_id):
        """修改地址"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        receiver = json_dict.get('receiver')
        province_id = json_dict.get('province_id')
        city_id = json_dict.get('city_id')
        district_id = json_dict.get('district_id')
        place = json_dict.get('place')
        mobile = json_dict.get('mobile')
        tel = json_dict.get('tel')
        email = json_dict.get('email')

        # 校验参数
        if not all([receiver, province_id, city_id, district_id, place, mobile]):
            return http.HttpResponseForbidden('缺少必传参数')
        if not re.match(r'^1[3-9]\d{9}$', mobile):
            return http.HttpResponseForbidden('参数mobile有误')
        if tel:
            if not re.match(r'^(\d{2,3}-)?(\d{2-9}[\d]{6,7})+(-\d{1,4})?$', tel):
                return http.HttpResponseForbidden('参数tel有误')
        if email:
            if not re.match(r'^[a-zA-Z][\w.\-]*@[a-zA-Z\-.]+\.\w{2,5}\w{1,2}$', email):
                return http.HttpResponseForbidden('参数email有误')

        # 判断地址是否存在，并更新地址信息
        try:
            Address.objects.filter(id=address_id).update(
                user = request.user,
                title = receiver,
                receiver = receiver,
                province_id = province_id,
                city_id = city_id,
                district_id = district_id,
                place = place,
                mobile = mobile,
                tel = tel,
                email = email
        
```

```

        )
    except Exception as e:
        logger.error(e)
        return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '更新地址失败'})
    }

    # 构造响应数据
    address = Address.objects.get(id=address_id)
    address_dict = {
        "id": address.id,
        "title": address.title,
        "receiver": address.receiver,
        "province": address.province.name,
        "city": address.city.name,
        "district": address.district.name,
        "place": address.place,
        "mobile": address.mobile,
        "tel": address.tel,
        "email": address.email
    }

    # 响应更新地址结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '更新地址成功', 'address': address_dict})

```

3. 修改地址前端逻辑实现

1. 添加修改地址的标记

- 新增地址和修改地址的交互不同。
- 为了区分用户是新增地址还是修改地址，我们可以选择添加一个变量，作为标记。
- 为了方便得到正在修改的地址信息，我们可以选择展示地址时对应的序号作为标记。

```

data: {
    editing_address_index: '',
},

```

2. 实现 编辑 按钮对应的事件

```

show_edit_site(index){
    this.is_show_edit = true;
    this.clear_all_errors();
    this.editing_address_index = index.toString();
},

```

```

<div class="down_btn">
    <a v-if="address.id!=default_address_id" @click="set_default(index)">设为默认</a>

```

```
<a @click="show_edit_site(index)" class="edit_icon">编辑</a>
</div>
```

```
<div class="site_pop_title">
  <h3 v-if="editing_address_index">编辑收货地址</h3>
  <h3 v-else>新增收货地址</h3>
  <a @click="is_show_edit=false">x</a>
</div>
```

3. 展示要重新编辑的数据

```
show_edit_site(index){
  this.is_show_edit = true;
  this.clear_all_errors();
  this.editing_address_index = index.toString();
  // 只获取要编辑的数据
  this.form_address = JSON.parse(JSON.stringify(this.addresses[index]));
},
```

4. 发送修改地址请求

- 重要提示：
 - `0 == ''` 返回 `true`
 - `0 === ''` 返回 `false`
 - 为了避免第0个索引出错，我们选择 `this.editing_address_index === ''` 的方式进行判断

```
if (this.editing_address_index === '') {
  // 新增地址
  .....
} else {
  // 修改地址
  let url = '/addresses/' + this.addresses[this.editing_address_index].id + '/';
  axios.put(url, this.form_address, {
    headers: {
      'X-CSRFToken': getCookie('csrftoken')
    },
    responseType: 'json'
  })
  .then(response => {
    if (response.data.code == '0') {
      this.addresses[this.editing_address_index] = response.data.address;
      this.is_show_edit = false;
    } else if (response.data.code == '4101') {
      location.href = '/login/?next=/addresses/';
    } else {
      // 处理其他错误情况
    }
  })
  .catch(error => {
    console.error('Error updating address:', error);
  });
}
```

```
        alert(response.data.errmsg);
    }
})
.catch(error => {
    alert(error.response);
})
}
```

删除地址后端逻辑

1. 删除地址接口设计和定义

1. 请求方式

选项	方案
请求方法	DELETE
请求地址	/addresses/(?P<address_id>\d+)/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
address_id	string	是	要修改的地址ID（路径参数）

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

2. 删除地址后端逻辑实现

提示：

- 删除地址不是物理删除，是逻辑删除。

```
class UpdateDestroyAddressView(LoginRequiredMixin, View):
    """修改和删除地址"""

    def put(self, request, address_id):
        """修改地址"""
        .....

    def delete(self, request, address_id):
        """删除地址"""
        try:
            # 查询要删除的地址
            address = Address.objects.get(id=address_id)

            # 将地址逻辑删除设置为True
            address.is_deleted = True
            address.save()
        except Exception as e:
            logger.error(e)
```

```

        return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '删除地址失败'})
    }

    # 响应删除地址结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '删除地址成功'})

```

3. 删除地址前端逻辑实现

```

delete_address(index){
    let url = '/addresses/' + this.addresses[index].id + '/';
    axios.delete(url, {
        headers: {
            'X-CSRFToken':getCookie('csrftoken')
        },
        responseType: 'json'
    })
    .then(response => {
        if (response.data.code == '0') {
            // 删除对应的标签
            this.addresses.splice(index, 1);
        } else if (response.data.code == '4101') {
            location.href = '/login/?next=/addresses/';
        }else {
            alert(response.data errmsg);
        }
    })
    .catch(error => {
        console.log(error.response);
    })
},

```

```

<div class="site_title">
    <h3>[[ address.title ]]</h3>
    <a href="javascript:;" class="edit_icon"></a>
    <em v-if="address.id==default_address_id">默认地址</em>
    <span @click="delete_address(index)" class="del_site">x</span>
</div>

```

设置默认地址

1. 设置默认地址接口设计和定义

1. 请求方式

选项	方案
请求方法	PUT
请求地址	/addresses/(?P<address_id>\d+)/default/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
address_id	string	是	要修改的地址ID（路径参数）

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

2. 设置默认地址后端逻辑实现

```

class DefaultAddressView(LoginRequiredMixin, View):
    """设置默认地址"""

    def put(self, request, address_id):
        """设置默认地址"""
        try:
            # 接收参数, 查询地址
            address = Address.objects.get(id=address_id)

            # 设置地址为默认地址
            request.user.default_address = address
            request.user.save()
        except Exception as e:
            logger.error(e)
            return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '设置默认地址失败'})

        # 响应设置默认地址结果
        return JsonResponse({'code': RETCODE.OK, 'errmsg': '设置默认地址成功'})
    
```

3. 设置默认地址前端逻辑实现

```
set_default(index){
    let url = '/addresses/' + this.addresses[index].id + '/default/';
    axios.put(url, {}, {
        headers: {
            'X-CSRFToken':getCookie('csrf_token')
        },
        responseType: 'json'
    })
    .then(response => {
        if (response.data.code == '0') {
            // 设置默认地址标签
            this.default_address_id = this.addresses[index].id;
        } else if (response.data.code == '4101') {
            location.href = '/login/?next=/addresses/';
        } else {
            alert(response.dataerrmsg);
        }
    })
    .catch(error => {
        console.log(error.response);
    })
},

```

```
<div class="down_btn">
    <a v-if="address.id!=default_address_id" @click="set_default(index)">设为默认</a>

    <a @click="show_edit_site(index)" class="edit_icon">编辑</a>
</div>
```

修改地址标题

1. 修改地址标题接口设计和定义

1. 请求方式

选项	方案
请求方法	PUT
请求地址	/addresses/(?P<address_id>\d+)/title/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
address_id	string	是	要修改的地址ID（路径参数）
title	string	是	要修改的地址标题（请求体参数）

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

2. 修改地址标题后端逻辑实现

```

class UpdateTitleAddressView(LoginRequiredMixin, View):
    """设置地址标题"""

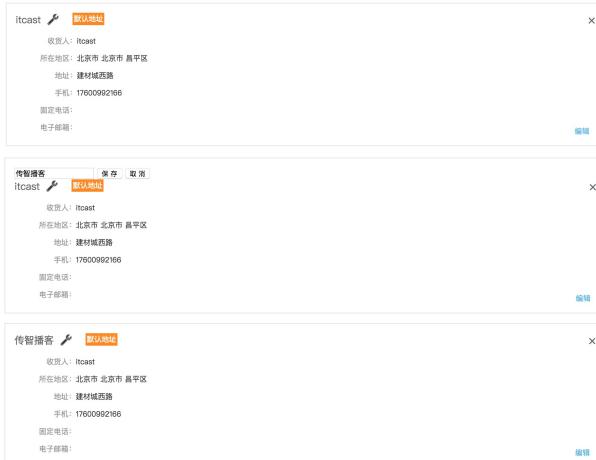
    def put(self, request, address_id):
        """设置地址标题"""
        # 接收参数：地址标题
        json_dict = json.loads(request.body.decode())
        title = json_dict.get('title')

        try:
            # 查询地址
            address = Address.objects.get(id=address_id)

            # 设置新的地址标题
            address.title = title
            address.save()
        except Exception as e:
            logger.error(e)
            return JsonResponse({'code': RETCODE.DBERR, 'errmsg': '设置地址标题失败'})
    
```

```
# 响应删除地址结果
return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '设置地址标题成功'})
```

3. 修改地址标题前端逻辑实现



```
<div class="site_title">
    <div v-if="edit_title_index === index">
        <input v-model="new_title" type="text" name="">
        <input @click="save_title(index)" type="button" name="" value="保 存">
        <input @click="cancel_title(index)" type="reset" name="" value="取 消">
    </div>
    <div>
        <h3>[[ address.title ]]</h3>
        <a @click="show_edit_title(index)" class="edit_title"></a>
    </div>
    <em v-if="address.id === default_address_id">默认地址</em>
    <span @click="delete_address(index)">x</span>
</div>
```

```
data: {
    edit_title_index: '',
    new_title: '',
},
```

```
// 展示地址title编辑框
show_edit_title(index){
    this.edit_title_index = index;
},
// 取消保存地址title
cancel_title(){
    this.edit_title_index = '';
    this.new_title = '';
},
```

```
// 修改地址title
save_title(index){
    if (!this.new_title) {
        alert("请填写标题后再保存！");
    } else {
        let url = '/addresses/' + this.addresses[index].id + '/title/';
        axios.put(url, {
            title: this.new_title
        }, {
            headers: {
                'X-CSRFToken':getCookie('csrftoken')
            },
            responseType: 'json'
        })
        .then(response => {
            if (response.data.code == '0') {
                // 更新地址title
                this.addresses[index].title = this.new_title;
                this.cancel_title();
            } else if (response.data.code == '4101') {
                location.href = '/login/?next=/addresses/';
            } else {
                alert(response.dataerrmsg);
            }
        })
        .catch(error => {
            console.log(error.response);
        })
    }
},
```

修改密码

1. 修改密码后端逻辑

提示：

- 修改密码前需要校验原始密码是否正确，以校验修改密码的用户身份。
- 如果原始密码正确，再将新的密码赋值给用户。

```
class ChangePasswordView(LoginRequiredMixin, View):
    """修改密码"""

    def get(self, request):
        """展示修改密码界面"""
        return render(request, 'user_center_pass.html')

    def post(self, request):
        """实现修改密码逻辑"""
        # 接收参数
        old_password = request.POST.get('old_password')
        new_password = request.POST.get('new_password')
        new_password2 = request.POST.get('new_password2')

        # 校验参数
        if not all([old_password, new_password, new_password2]):
            return http.HttpResponseForbidden('缺少必传参数')
        try:
            request.user.check_password(old_password)
        except Exception as e:
            logger.error(e)
            return render(request, 'user_center_pass.html', {'origin_pwd_errmsg': '原始密码错误'})
        if not re.match(r'^[0-9A-Za-z]{8,20}$', new_password):
            return http.HttpResponseForbidden('密码最少8位，最长20位')
        if new_password != new_password2:
            return http.HttpResponseForbidden('两次输入的密码不一致')

        # 修改密码
        try:
            request.user.set_password(new_password)
            request.user.save()
        except Exception as e:
            logger.error(e)
```

```
        return render(request, 'user_center_pass.html', {'change_pwd_errmsg': '修改密码失败'})  
  
    # 清理状态保持信息  
    logout(request)  
    response = redirect(reverse('users:login'))  
    response.delete_cookie('username')  
  
    # # 响应密码修改结果：重定向到登录界面  
    return response
```

商品

商品数据库表设计

SPU和SKU

在电商中对于商品，有两个重要的概念：**SPU**和**SKU**

1. SPU介绍

- **SPU = Standard Product Unit**（标准产品单位）
 - SPU是商品信息聚合的最小单位，是一组可服用、易检索的标准化信息的集合，该集合描述了一个产品的特性。
 - 通俗的讲，属性值、特性相同的商品就可以归类到一类SPU。
 - 例如：
 - iPhone X 就是一个**SPU**，与商家、颜色、款式、规格、套餐等都无关。



2. SKU介绍

- **SKU = Stock Keeping Unit**（库存量单位）
 - SKU即库存进出计量的单位，可以是以件、盒等为单位，是物理上不可分割的最小存货单元。
 - 通俗的讲，SKU是指一款商品，每款都有一个SKU，便于电商品牌识别商品。
 - 例如：
 - iPhone X 全网通 黑色 256G 就是一个**SKU**，表示了具体的规格、颜色等信息。

Apple iPhone 8 (A1863) 64GB 深空灰色 移动联通电信4G手机



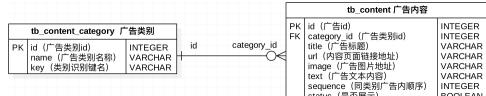
思考

SPU和SKU是怎样的对应关系？

- 一对一？
- 一对多？

首页广告数据库表分析

1. 首页广告数据库表分析



2. 定义首页广告模型类

定义在首页广告模块子应用中

```

class ContentCategory(BaseModel):
    """广告内容类别"""
    name = models.CharField(max_length=50, verbose_name='名称')
    key = models.CharField(max_length=50, verbose_name='类别键名')

    class Meta:
        db_table = 'tb_content_category'
        verbose_name = '广告内容类别'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.name


class Content(BaseModel):
    """广告内容"""
    category = models.ForeignKey(ContentCategory, on_delete=models.PROTECT, verbose_name='类别')
    title = models.CharField(max_length=100, verbose_name='标题')
    url = models.CharField(max_length=300, verbose_name='内容链接')
    image = models.ImageField(null=True, blank=True, verbose_name='图片')
    text = models.TextField(null=True, blank=True, verbose_name='内容')
    sequence = models.IntegerField(verbose_name='排序')
    status = models.BooleanField(default=True, verbose_name='是否展示')

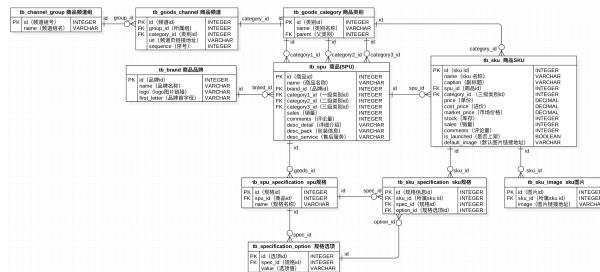
    class Meta:
        db_table = 'tb_content'
        verbose_name = '广告内容'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.category.name + ': ' + self.title

```


商品信息数据库表分析

1. 商品信息数据库表分析



2. 定义商品信息模型类

定义在商品模块子应用中

```

class GoodsCategory(BaseModel):
    """商品类别"""
    name = models.CharField(max_length=10, verbose_name='名称')
    parent = models.ForeignKey('self', related_name='subs', null=True, blank=True,
        on_delete=models.CASCADE, verbose_name='父类别')

    class Meta:
        db_table = 'tb_goods_category'
        verbose_name = '商品类别'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.name


class GoodsChannelGroup(BaseModel):
    """商品频道组"""
    name = models.CharField(max_length=20, verbose_name='频道组名')

    class Meta:
        db_table = 'tb_channel_group'
        verbose_name = '商品频道组'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.name


class GoodsChannel(BaseModel):
    """商品频道"""
    group = models.ForeignKey(GoodsChannelGroup, verbose_name='频道组名')

```

```

category = models.ForeignKey(GoodsCategory, on_delete=models.CASCADE, verbose_name='顶级商品类别')
url = models.CharField(max_length=50, verbose_name='频道页面链接')
sequence = models.IntegerField(verbose_name='组内顺序')

class Meta:
    db_table = 'tb_goods_channel'
    verbose_name = '商品频道'
    verbose_name_plural = verbose_name

def __str__(self):
    return self.category.name


class Brand(BaseModel):
    """品牌"""
    name = models.CharField(max_length=20, verbose_name='名称')
    logo = models.ImageField(verbose_name='Logo图片')
    first_letter = models.CharField(max_length=1, verbose_name='品牌首字母')

    class Meta:
        db_table = 'tb_brand'
        verbose_name = '品牌'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.name


class SPU(BaseModel):
    """商品SPU"""
    name = models.CharField(max_length=50, verbose_name='名称')
    brand = models.ForeignKey(Brand, on_delete=models.PROTECT, verbose_name='品牌')
    category1 = models.ForeignKey(GoodsCategory, on_delete=models.PROTECT, related_name='cat1_spu', verbose_name='一级类别')
    category2 = models.ForeignKey(GoodsCategory, on_delete=models.PROTECT, related_name='cat2_spu', verbose_name='二级类别')
    category3 = models.ForeignKey(GoodsCategory, on_delete=models.PROTECT, related_name='cat3_spu', verbose_name='三级类别')
    sales = models.IntegerField(default=0, verbose_name='销量')
    comments = models.IntegerField(default=0, verbose_name='评价数')
    desc_detail = models.TextField(default='', verbose_name='详细介绍')
    desc_pack = models.TextField(default='', verbose_name='包装信息')
    desc_service = models.TextField(default='', verbose_name='售后服务')

    class Meta:
        db_table = 'tb_spu'
        verbose_name = '商品SPU'
        verbose_name_plural = verbose_name

    def __str__(self):

```

```

        return self.name

class SKU(BaseModel):
    """商品SKU"""
    name = models.CharField(max_length=50, verbose_name='名称')
    caption = models.CharField(max_length=100, verbose_name='副标题')
    spu = models.ForeignKey(SPU, on_delete=models.CASCADE, verbose_name='商品')
    category = models.ForeignKey(GoodsCategory, on_delete=models.PROTECT, verbose_name='从属类别')
    price = models.DecimalField(max_digits=10, decimal_places=2, verbose_name='单价')
    cost_price = models.DecimalField(max_digits=10, decimal_places=2, verbose_name='进价')
    market_price = models.DecimalField(max_digits=10, decimal_places=2, verbose_name='市场价')
    stock = models.IntegerField(default=0, verbose_name='库存')
    sales = models.IntegerField(default=0, verbose_name='销量')
    comments = models.IntegerField(default=0, verbose_name='评价数')
    is_launched = models.BooleanField(default=True, verbose_name='是否上架销售')
    default_image = models.ImageField(max_length=200, default='', null=True, blank=True, verbose_name='默认图片')

    class Meta:
        db_table = 'tb_sku'
        verbose_name = '商品SKU'
        verbose_name_plural = verbose_name

    def __str__(self):
        return '%s: %s' % (self.id, self.name)

class SKUImage(BaseModel):
    """SKU图片"""
    sku = models.ForeignKey(SKU, on_delete=models.CASCADE, verbose_name='sku')
    image = models.ImageField(verbose_name='图片')

    class Meta:
        db_table = 'tb_sku_image'
        verbose_name = 'SKU图片'
        verbose_name_plural = verbose_name

    def __str__(self):
        return '%s %s' % (self.sku.name, self.id)

class SPUSpecification(BaseModel):
    """商品SPU规格"""
    spu = models.ForeignKey(SPU, on_delete=models.CASCADE, related_name='specs', verbose_name='商品SPU')
    name = models.CharField(max_length=20, verbose_name='规格名称')

```

```

class Meta:
    db_table = 'tb_spu_specification'
    verbose_name = '商品SPU规格'
    verbose_name_plural = verbose_name

    def __str__(self):
        return '%s: %s' % (self.spu.name, self.name)

class SpecificationOption(BaseModel):
    """规格选项"""
    spec = models.ForeignKey(SPUSpecification, related_name='options', on_delete=models.CASCADE, verbose_name='规格')
    value = models.CharField(max_length=20, verbose_name='选项值')

    class Meta:
        db_table = 'tb_specification_option'
        verbose_name = '规格选项'
        verbose_name_plural = verbose_name

        def __str__(self):
            return '%s - %s' % (self.spec, self.value)

class SKUSpecification(BaseModel):
    """SKU具体规格"""
    sku = models.ForeignKey(SKU, related_name='specs', on_delete=models.CASCADE, verbose_name='sku')
    spec = models.ForeignKey(SPUSpecification, on_delete=models.PROTECT, verbose_name='规格名称')
    option = models.ForeignKey(SpecificationOption, on_delete=models.PROTECT, verbose_name='规格值')

    class Meta:
        db_table = 'tb_sku_specification'
        verbose_name = 'SKU规格'
        verbose_name_plural = verbose_name

        def __str__(self):
            return '%s: %s - %s' % (self.sku, self.spec.name, self.option.value)

```

准备商品数据

提示：

- 数据库表有了以后，我们现在需要准备商品信息数据和商品图片数据，以便查询和展示。
- 商品信息数据：比如商品编号等都是字符串类型的，可以直接存储在MySQL数据库。
- 商品图片数据：MySQL通常存储的是图片的地址字符串信息。
 - 所以图片数据需要进行其他的物理存储。

id	image	sku_id
6	group1/M00/00/02/CtM3BVrPB5CALKn6AADq-Afr0eE1672090	1
7	group1/M00/00/02/CtM3BVrPCA0AIKRBAAGvaeRBMfc0463515	2

图片物理存储思考：

- 需要提供图片上传和下载的机制。
- 需要解决图片备份和扩容的问题。
- 需要解决图片重名的问题等等。

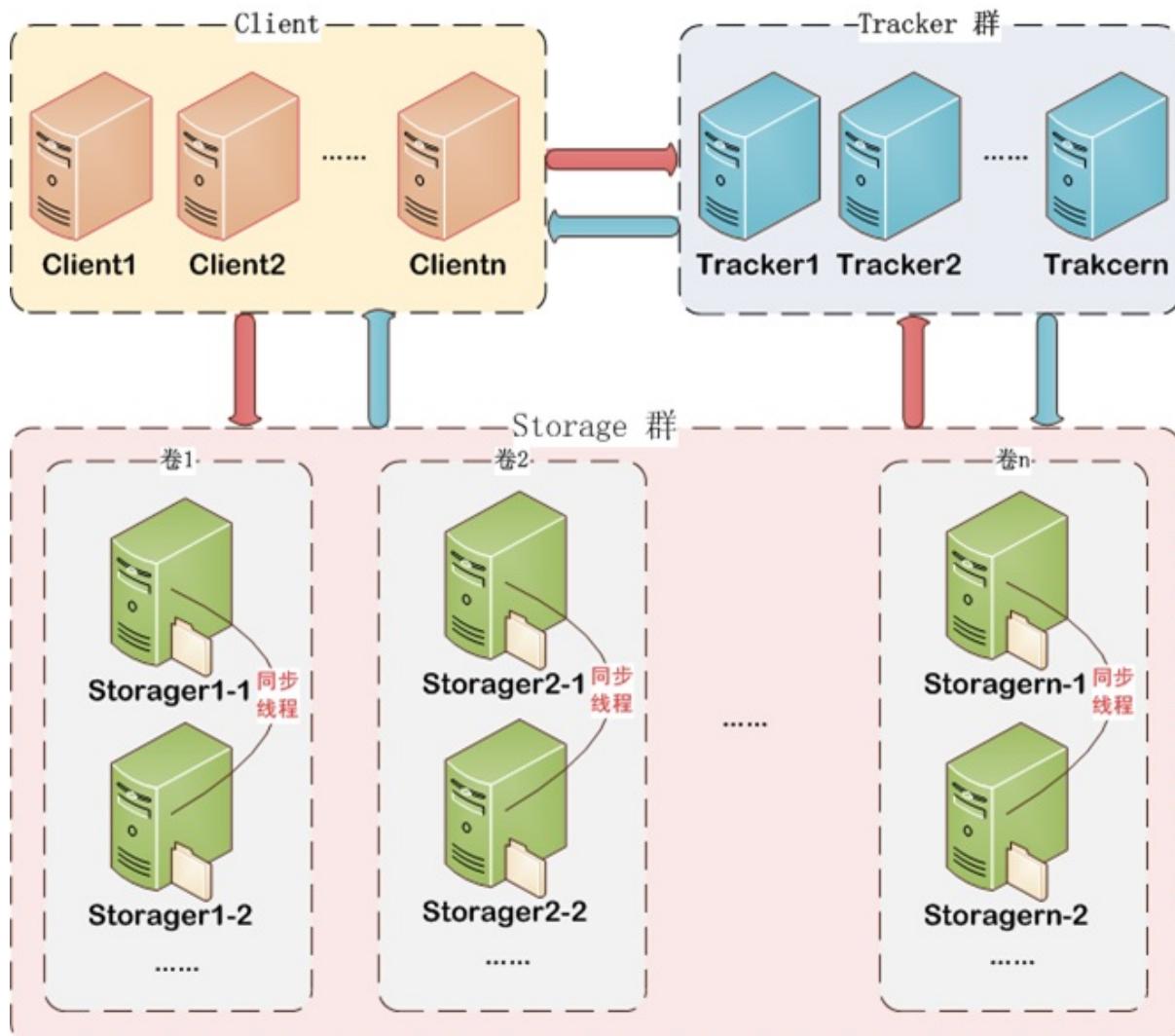
图片物理存储方案：

- **FastDFS**

文件存储方案FastDFS

1. FastDFS介绍

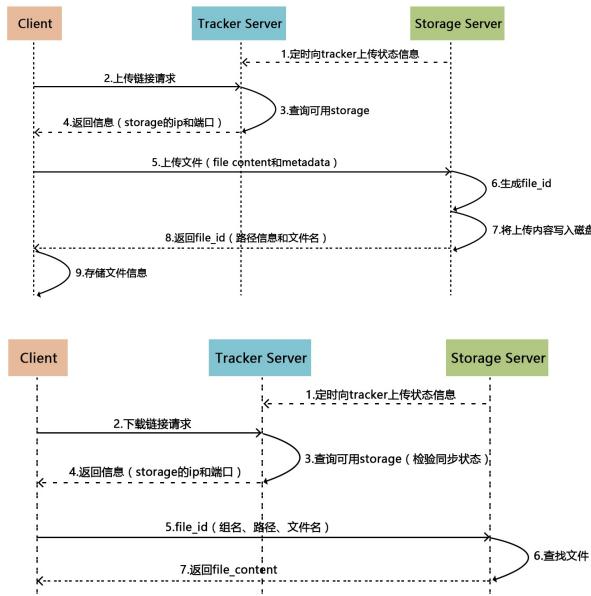
- 用 c语言 编写的一款开源的轻量级分布式文件系统。
- 功能包括：文件存储、文件访问（文件上传、文件下载）、文件同步等，解决了大容量存储和负载均衡的问题。特别适合以文件为载体的在线服务，如相册网站、视频网站等等。
- 为互联网量身定制，充分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标。
- 可以帮助我们搭建一套高性能的文件服务器集群，并提供文件上传、下载等服务。



- **FastDFS架构** 包括 Client 、 Tracker server 和 Storage server 。
 - Client 请求 Tracker 进行文件上传、下载， Tracker 再调度 Storage 完成文件上传和下载。
- **Client**: 客户端，业务请求的发起方，通过专有接口，使用TCP/IP协议与 Tracker 或 Storage 进行数据交互。FastDFS提供了 upload 、 download 、 delete 等接口供客户端使用。

- **Tracker server**: 跟踪服务器，主要做调度工作，起负载均衡的作用。在内存中记录集群中所有存储组和存储服务器的状态信息，是客户端和数据服务器交互的枢纽。
- **Storage server**: 存储服务器（存储节点或数据服务器），文件和文件属性都保存到存储服务器上。Storage server直接利用OS的文件系统调用管理文件。
 - Storage群中的横向可以扩容，纵向可以备份。

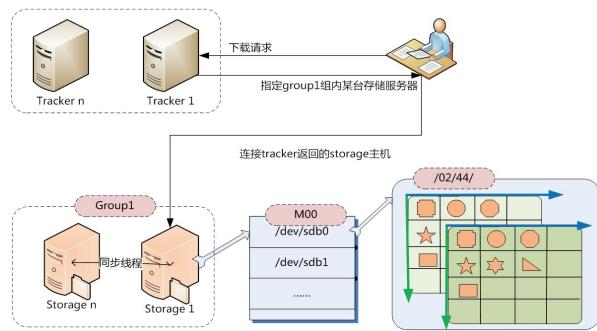
2. FastDFS上传和下载流程



3. FastDFS文件索引

group1/M00/00/00/wKhnnlw_gmAcoWmAAEXU5wmjPs35.jpg

- FastDFS上传和下载流程可以看出都涉及到一个数据叫文件索引 (**file_id**) 。
 - 文件索引 (**file_id**) 是客户端上传文件后Storage返回给客户端的一个字符串，是以后访问该文件的索引信息。
- 文件索引 (**file_id**) 信息包括：组名、虚拟磁盘路径、数据两级目录、文件名等信息。
 - 组名：文件上传后所在的 Storage 组名称。
 - 虚拟磁盘路径：Storage 配置的虚拟路径，与磁盘选项 `store_path*` 对应。如果配置了 `store_path0` 则是 M00，如果配置了 `store_path1` 则是 M01，以此类推。
 - 数据两级目录：Storage 服务器在每个虚拟磁盘路径下创建的两级目录，用于存储数据文件。
 - 文件名：由存储服务器根据特定信息生成，文件名包含：源存储服务器IP地址、文件创建时间戳、文件大小、随机数和文件拓展名等信息。



容器化方案Docker

思考：

- FastDFS的安装步骤非常的多，涉及的依赖包也很多，当新的机器需要安装FastDFS时，是否需要从头开始安装。
- 我们在学习时拿到ubuntu系统的镜像，在VM虚拟机中运行这个镜像后，为什么就可以直接进行开发，而不需要重新搭建开发环境。
- 在工作中，如何高效的保证开发人员写代码的开发环境与应用程序要部署的生产环境一致性。如果要部署一台新的机器，是否需要从头开始部署。

结论：

- 上述思考的问题，都涉及到相同的工作是否需要重复做。
- 避免相同的工作重复做是容器化技术应用之一。

容器化方案：

- Docker
- Docker的目标之一就是缩短代码从开发、测试到部署、上线运行的周期，让我们的应用程序具备可移植性、易于构建、并易于协作。

1. Docker介绍

- [Docker中文社区文档](#)
- Docker 是一个开源的软件部署解决方案。
- Docker 也是轻量级的应用容器框架。
- Docker 可以打包、发布、运行任何的应用。
- Docker 就像一个盒子，里面可以装很多物件，如果需要某些物件，可以直接将该盒子拿走，而不需要从该盒子中一件一件的取。
- Docker 是一个 客户端-服务端(C/S) 架构程序。
 - 客户端只需要向服务端发出请求，服务端处理完请求后会返回结果。

Docker 包括三个基本概念：

- 镜像 (Image)
 - Docker的镜像概念类似于虚拟机里的镜像，是一个只读的模板，一个独立的文件系统，包括运行容器所需的数据，可以用来创建新的容器。
 - 例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了MySQL或用户需要的其它应用程序。
- 容器 (Container)
 - Docker容器是由Docker镜像创建的运行实例，类似VM虚拟机，支持启动，停止，删除等。
 - 每个容器间是相互隔离的，容器中会运行特定的应用，包含特定应用的代码及所需的依赖文件。
- 仓库 (Repository)

- Docker的仓库功能类似于Github，是用于托管镜像的。

2. Docker安装（ubuntu 16.04）

1. 源码安装Docker CE

```
$ cd docker源码目录
$ sudo apt-key add gpg
$ sudo dpkg -i docker-ce_17.03.2-ce-0~ubuntu-xenial_amd64.deb
```

2. 检查Docker CE是否安装正确

```
$ sudo docker run hello-world
```

出现如下信息，表示安装成功

3. 启动与停止

- 安装完成Docker后，默认已经启动了docker服务。

```
# 启动docker
$ sudo service docker start
# 重启docker
$ sudo service docker restart
# 停止docker
$ sudo service docker stop
```

3. Docker镜像操作

1. 镜像列表

```
$ sudo docker image ls
```

```
python@ubuntu:~$ sudo docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
hello-world         latest   fce289e99eb9    7 weeks ago   1.84 kB
```

- * REPOSITORY: 镜像所在的仓库名称
- * TAG: 镜像标签
- * IMAGEID: 镜像ID
- * CREATED: 镜像的创建日期(不是获取该镜像的日期)
- * SIZE: 镜像大小

2. 从仓库拉取镜像

```
# 官方镜像
```

```
$ sudo docker image pull 镜像名称 或者 sudo docker image pull library/镜像名称
$ sudo docker image pull ubuntu 或者 sudo docker image pull library/ubuntu
$ sudo docker image pull ubuntu:16.04 或者 sudo docker image pull library/ubuntu:16.04

# 个人镜像
$ sudo docker image pull 仓库名称/镜像名称
$ sudo docker image pull itcast/fastdfs
```

```
python@ubuntu:~$ sudo docker image pull ubuntu:16.04
16.04: Pulling from library/ubuntu
7b722c1d79cd: Downloading [=====] 6.671 MB/43.52 MB
5fbf74d6b1ff: Download complete
python@ubuntu:~$ sudo docker image pull delron/fastdfs
Using default tag: latest
latest: Pulling from delron/fastdfs
469cfcc7a4b3: Downloading [=====] 12.8 MB/73.17 MB
4bf08b0d0171: Downloading [=====] 31.68 MB/88.7 MB
95eeff997896: Download complete
python@ubuntu:~$ sudo docker image ls
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
ubuntu          16.04        7e87e2b3bf7a   4 weeks ago    117 MB
hello-world     latest        fce289e99eb9   7 weeks ago    1.84 kB
delron/fastdfs  latest        8487e86fc6ee   9 months ago   464 MB
```

3.删除镜像

```
$ sudo docker image rm 镜像名或镜像ID
$ sudo docker image rm hello-world
$ sudo docker image rm fce289e99eb9
```

```
python@ubuntu:~$ sudo docker image ls
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
ubuntu          16.04        7e87e2b3bf7a   4 weeks ago    117 MB
delron/fastdfs  latest        8487e86fc6ee   9 months ago   464 MB
```

4. Docker容器操作

1.容器列表

```
# 查看正在运行的容器
$ sudo docker container ls
# 查看所有的容器
$ sudo docker container ls --all
```

```
python@ubuntu:~$ sudo docker container ls
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS          NAMES
495986899999:~      "ubuntu:16.04"   "/bin/bash"   10 seconds ago   Up 9 seconds
python@ubuntu:~$ sudo docker container ls --all
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS          NAMES
495986899999:~      "ubuntu:16.04"   "/bin/bash"   18 seconds ago   Up 17 seconds
a4550801ff6d        "ubuntu:16.04"   "/bin/bash"   About a minute ago   Exited (0) About a minute ago
ubuntu16.04
```

2.创建容器

```
$ sudo docker run [option] 镜像名 [向启动容器中传入的命令]
```

常用可选参数说明：

- * -i 表示以《交互模式》运行容器。
- * -t 表示容器启动后会进入其命令行。加入这两个参数后，容器创建就能登录进去。即分配一个伪终端。
- * --name 为创建的容器命名。
- * -v 表示目录映射关系，即宿主机目录:容器中目录。注意：最好做目录映射，在宿主机上做修改，然后共享到容器上。
- * -d 会创建一个守护式容器在后台运行(这样创建容器后不会自动登录容器)。
- * -p 表示端口映射，即宿主机端口:容器中端口。

* `--network=host` 表示将主机的网络环境映射到容器中，使容器的网络与主机相同。

3.交互式容器

```
$ sudo docker run -it --name=ubuntu1 ubuntu /bin/bash
```

```
python@ubuntu:~$ sudo docker run -it --name=ubuntu1 ubuntu:16.04 /bin/bash
root@cd216c0f285c:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin
root@cd216c0f285c:/# exit
exit
python@ubuntu:~$
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1117929d8e1	ubuntu:16.04	/bin/bash	12 seconds ago	Exited (0) 3 seconds ago		ubuntu1

在容器中可以随意执行linux命令，就是一个ubuntu的环境。

当执行 `exit` 命令退出时，该容器随之停止。

4.守护式容器

```
# 开启守护式容器
```

```
$ sudo docker run -dit --name=ubuntu2 ubuntu
```

```
python@ubuntu:~$ sudo docker run -dit --name=ubuntu2 ubuntu:16.04 /bin/bash
python@ubuntu:~$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
9f8cd43ad5e        ubuntu:16.04       "/bin/bash"         21 seconds ago   Up 20 seconds
1117929d8e1        ubuntu:16.04       "/bin/bash"         3 minutes ago    Exited (0) 3 minutes ago
python@ubuntu:~$
```

进入到容器内部交互环境

\$ sudo docker exec -it 容器名或容器id 进入后执行的第一个命令

\$ sudo docker exec -it ubuntu2 /bin/bash

```
python@ubuntu:~$ sudo docker exec -it ubuntu2 /bin/bash
root@9f8cd43ad5e:~# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin sys usr var
root@9f8cd43ad5e:~# exit
exit
python@ubuntu:~$ sudo docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
9f8cd43ad5e        ubuntu:16.04       "/bin/bash"         40 seconds ago   Up 39 seconds
1117929d8e1        ubuntu:16.04       "/bin/bash"         3 minutes ago    Exited (0) 3 minutes ago
python@ubuntu:~$
```

如果对于一个需要长期运行的容器来说，我们可以创建一个守护式容器。

在容器内部执行 `exit` 命令退出时，该容器也随之停止。

5.停止和启动容器

```
# 停止容器
$ sudo docker container stop 容器名或容器id
# kill掉容器
$ sudo docker container kill 容器名或容器id
# 启动容器
$ sudo docker container start 容器名或容器id
```



6.删除容器

- 正在运行的容器无法直接删除。

```
$ sudo docker container rm 容器名或容器id
```

```
python@ubuntu:~$ sudo docker container ls -all
python@ubuntu:~$ sudo docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
9f8cd43ab5e        ubuntu:16.04       "/bin/bash"        12 minutes ago   Up 3 minutes          ubuntu2
```

7.容器制作成镜像

- 为保证已经配置完成的环境可以重复利用，我们可以将容器制作成镜像。

```
# 将容器制作成镜像
$ sudo docker commit 容器名 镜像名
```

```
python@ubuntu:~$ sudo docker commit ubuntu2 ubuntu2_1mp
sha256:d085a8652fe08bf63990091c46a2e0c9daf2f69e63de2b3b2e6b720d2a46f
python@ubuntu:~$ sudo docker image ls
REPOSITORY          TAG      IMAGE ID            CREATED             SIZE
ubuntu2_1mp         latest   ed085a8652fe        1 second ago       117 MB
ubuntu              16.04    f687eb2b3bf7a      4 weeks ago        117 MB
deltar0n/fastdfs   latest   6487edfcdee       9 months ago       464 MB
```

```
# 镜像打包备份
$ sudo docker save -o 保存的文件名 镜像名
```

```
python@ubuntu:~$ sudo docker save -o ubuntu2_1mp.tar ubuntu2_1mp
[core] Desktop [Downloads] [etc] [lib] [media] [mysql] [mysql-clave] [Pictures] [PycharmProjects] [rabbitmq] [Templates] [ubuntu2_1mp.tar]
```

```
# 镜像解压
$ sudo docker load -i 文件路径/备份文件
```

```
python@ubuntu:~$ sudo docker image rm ubuntu2_1mp
Deleted: sha256:9e885a8652fe08bf63990091c46a2e0c9daf2f69e63de2b3b2e6b720d2a46f
python@ubuntu:~$ sudo docker image ls
REPOSITORY          TAG      IMAGE ID            CREATED             SIZE
ubuntu              16.04    f687eb2b3bf7a      4 weeks ago        117 MB
ubuntu              latest   ed085a8652fe        1 second ago       117 MB
deltar0n/fastdfs   latest   6487edfcdee       9 months ago       464 MB
```

Docker和FastDFS上传和下载文件

1. Docker安装运行FastDFS

1.获取FastDFS镜像

```
# 从仓库拉取镜像
$ sudo docker image pull delron/fastdfs
# 解压教学资料中本地镜像
$ sudo docker load -i 文件路径/fastdfs_docker.tar
```

2.开启tracker容器

- 我们将 tracker 运行目录映射到宿主机的 /var/fdfs/tracker 目录中。

```
$ sudo docker run -dit --name tracker --network=host -v /var/fdfs/tracker:/var/fdfs
delron/fastdfs tracker
```

3.开启storage容器

- TRACKER_SERVER=Tracker的ip地址:22122 (Tracker的ip地址不要使用127.0.0.1)
- 我们将 storage 运行目录映射到宿主机的 /var/fdfs/storage 目录中。

```
$ sudo docker run -dti --name storage --network=host -e TRACKER_SERVER=192.168.103.
158:22122 -v /var/fdfs/storage:/var/fdfs delron/fastdfs storage
```

4.查看宿主机映射路径

```
python@ubuntu:/var/fdfs$ ls
storage  tracker
```

注意：如果无法重启storage容器，可以删除 /var/fdfs/storage/data 目录下的 fdfs_storaged.pid 文件，然后重新运行storage。

2. FastDFS客户端上传文件

- Python版本的FastDFS客户端使用参考文档

1.安装FastDFS客户端扩展

- 安装准备好的 fdfs_client-py-master.zip 到虚拟环境中

```
$ pip install fdfs_client-py-master.zip
```

```
$ pip install mutagen
$ pip install requests
```

2.准备FastDFS客户端扩展的配置文件

- meiduo_mall.utils.fastdfs.client.conf



base_path=FastDFS客户端存放日志文件的目录

tracker_server=运行Tracker服务的机器ip:22122

3.FastDFS客户端实现文件存储

```
# 使用 shell 进入 Python交互环境
$ python manage.py shell
```

```
# 1. 导入FastDFS客户端扩展
from fdfs_client.client import Fdfs_client
# 2. 创建FastDFS客户端实例
client = Fdfs_client('meiduo_mall/utils/fastdfs/client.conf')
# 3. 调用FastDFS客户端上传文件方法
ret = client.upload_by_filename('/Users/zhangjie/Desktop/kk.jpeg')
```

```
ret = {
    'Group name': 'group1',
    'Remote file_id': 'group1/M00/00/00/wKhnnlxw_gmAcowmAAEXU5wmjPs35.jpeg',
    'Status': 'Upload successed.',
    'Local file name': '/Users/zhangjie/Desktop/kk.jpeg',
    'Uploaded size': '69.00KB',
    'Storage IP': '192.168.103.158'
}
```

```
ret = {
    'Group name': 'Storage组名',
    'Remote file_id': '文件索引，可用于下载',
    'Status': '文件上传结果反馈',
    'Local file name': '上传文件全路径',
    'Uploaded size': '文件大小',
    'Storage IP': 'Storage地址'
}
```

```
python@ubuntu:/var/fdfs/storage/data$ cd 00/00/
python@ubuntu:/var/fdfs/storage/data/00/00$ ls
wKhnnlxw_gmAcoWmAAEXU5wmjPs35.jpeg
```

3. 浏览器下载并渲染图片

思考：如何才能找到在Storage中存储的图片？

- 协议：
 - http
- IP地址：192.168.103.158
 - Nginx 服务器的IP地址。
 - 因为 FastDFS 擅长存储静态文件，但是不擅长提供静态文件的下载服务，所以我们一般会将 Nginx 服务器绑定到 Storage，提升下载性能。
- 端口：8888
 - Nginx 服务器的端口。
- 路径：group1/M00/00/00/wKhnnlxw_gmAcoWmAAEXU5wmjPs35.jpeg
 - 文件在Storage上的文件索引。
- 完整图片下载地址
 - http://192.168.103.158:8888/group1/M00/00/00/wKhnnlxw_gmAcoWmAAEXU5wmjPs35.jpeg

编写测试代码：meiduo_mall.utils.fdfs_t.html

```

```

录入商品数据和图片数据

1. SQL脚本录入商品数据

```
$ mysql -h127.0.0.1 -uroot -pmysql meiduo_mall < 文件路径/goods_data.sql
```

2. FastDFS服务器录入图片数据

1.准备新的图片数据压缩包

```
python@ubuntu:~$ cd Desktop/
python@ubuntu:~/Desktop$ ls
data.tar.gz  projects
```

2.删除 Storage 中旧的 data 目录

```
python@ubuntu:~/Desktop$ cd /var/fdfs/storage/
python@ubuntu:/var/fdfs/storage$ ls
data  logs
python@ubuntu:/var/fdfs/storage$ sudo rm -rf data/
python@ubuntu:/var/fdfs/storage$ ls
logs
```

3.拷贝新的图片数据压缩包到 Storage，并解压

```
# 解压命令
sudo tar -zxvf data.tar.gz
```

```
python@ubuntu:/var/fdfs/storage$ sudo cp ~/Desktop/data.tar.gz ./
python@ubuntu:/var/fdfs/storage$ ls
data.tar.gz  logs
python@ubuntu:/var/fdfs/storage$ sudo tar -zxvf data.tar.gz
```

4.查看新的 data 目录

```
python@ubuntu:/var/fdfs/storage$ ls
data  data.tar.gz  logs

python@ubuntu:/var/fdfs/storage$ ls
CtM3BVni03-ANUDwAAAqv27px4k9203075  CtM3BVni4RWAU5g1AAAljHPuXJg3689968
CtM3BVniow6AuajbAAAmv27px4k0365964  CtM3BVni4s6AH_heAD0akkXnF07197106
CtM3BVniH2ARPiXAAAqv27px4k6813735  CtM3BVni4sCAJlzHAAETwxK_pso9622268
```

首页广告

首页数据由 商品频道分类 和 广告 组成

商品分类 首页 | 真划算 | 抽奖

手机 办公 数码
电器 家具 家居 厨具
男鞋 女包 帽饰 内衣
女鞋 钱包 钟表 陈设
男鞋 服饰 户外
房产 汽车 汽车用品
母婴 时尚服饰
食品 消费 生鲜 特产
图书 音像 电子书
机票 酒店 旅游 生活

快讯 更多 >

门类手机最低499元
真划算专享 正价低至319元起
实惠商品最低 限购服务立减22...
更多权益会员特权
门类手机最低499元
真划算专享 正价低至319元起

好友联播 双双领

1F 手机通讯

荣耀V10优惠200
360手机 N6 Pro 全网通 6... ¥ 2699.00
iPhoneX N6 Pro 全网通 6G... ¥ 7788.00
360手机 N6 Pro 全网通 6... ¥ 1988.00
360手机 N6 Pro 全网通 6... ¥ 3688.00
360手机 N6 Pro 全网通 6... ¥ 4288.80

展示首页商品分类

1. 分析首页商品分类数据结构



```
{
  "1": {
    "channels": [
      {"id": 1, "name": "手机", "url": "http://shouji.jd.com/"},
      {"id": 2, "name": "相机", "url": "http://www.itcast.cn/"}
    ],
    "sub_cats": [
      {
        "id": 38,
        "name": "手机通讯",
        "sub_cats": [
          {"id": 115, "name": "手机"}, {"id": 116, "name": "游戏手机"}
        ]
      },
      {
        "id": 39,
        "name": "手机配件",
        "sub_cats": [
          {"id": 119, "name": "手机壳"}, {"id": 120, "name": "贴膜"}
        ]
      }
    ]
  },
  "2": {
    "channels": [],
    "sub_cats": []
  }
}
```

2. 查询首页商品分类

```
class IndexView(View):
    """首页广告"""

```

```

def get(self, request):
    """提供首页广告界面"""
    # 查询商品频道和分类
    categories = OrderedDict()
    channels = GoodsChannel.objects.order_by('group_id', 'sequence')
    for channel in channels:
        group_id = channel.group_id # 当前组

        if group_id not in categories:
            categories[group_id] = {'channels': [], 'sub_cats': []}

        cat1 = channel.category # 当前频道的类别

        # 追加当前频道
        categories[group_id]['channels'].append({
            'id': cat1.id,
            'name': cat1.name,
            'url': channel.url
        })
        # 构建当前类别的子类别
        for cat2 in cat1.subs.all():
            cat2.sub_cats = []
            for cat3 in cat2.subs.all():
                cat2.sub_cats.append(cat3)
            categories[group_id]['sub_cats'].append(cat2)

    # 渲染模板的上下文
    context = {
        'categories': categories,
    }
    return render(request, 'index.html', context)

```

3. 渲染首页商品分类

index.html

```

<ul class="sub_menu">
    {% for group in categories.values() %}
    <li>
        <div class="level1">
            {% for channel in group.channels %}
            <a href="{{ channel.url }}>{{ channel.name }}</a>
            {% endfor %}
        </div>
        <div class="level2">
            {% for cat2 in group.sub_cats %}
            <div class="list_group">
                <div class="group_name fl">{{ cat2.name }} &gt;</div>

```

```

        <div class="group_detail f1">
            {% for cat3 in cat2.sub_cats %}
                <a href="/list/{{ cat3.id }}/1/">{{ cat3.name }}</a>
            {% endfor %}
        </div>
    {% endfor %}
</div>
</li>
{% endfor %}
</ul>

```

4. 封装首页商品分类

1. 封装首页商品频道分类到 contents.utils.py 文件

```

def get_categories():
    """
    提供商品频道和分类
    :return 菜单字典
    """

    # 查询商品频道和分类
    categories = OrderedDict()
    channels = GoodsChannel.objects.order_by('group_id', 'sequence')
    for channel in channels:
        group_id = channel.group_id  # 当前组

        if group_id not in categories:
            categories[group_id] = {'channels': [], 'sub_cats': []}

        cat1 = channel.category  # 当前频道的类别

        # 追加当前频道
        categories[group_id]['channels'].append({
            'id': cat1.id,
            'name': cat1.name,
            'url': channel.url
        })

        # 构建当前类别的子类别
        for cat2 in cat1.subs.all():
            cat2.sub_cats = []
            for cat3 in cat2.subs.all():
                cat2.sub_cats.append(cat3)
            categories[group_id]['sub_cats'].append(cat2)

    return categories

```

2. contents.view.py 中使用 contents.utils.py 文件

```
class IndexView(View):
    """首页广告"""

    def get(self, request):
        """提供首页广告界面"""
        # 查询商品频道和分类
        categories = get_categories()

        # 渲染模板的上下文
        context = {
            'categories': categories,
        }
        return render(request, 'index.html', context)
```

展示首页商品广告

1. 分析首页商品广告数据结构



	id	name	key
1	...	轮播图	index_lbt
2	...	快讯	index_kx
3	...	页头广告	index_ytgg
5	...	1楼Logo	index_1f_logo
6	...	1楼频道	index_1f_pd
7	...	1楼标签	index_1f_bq
8	...	1楼时尚新品	index_1f_ssxp
10	...	1楼畅享低价	index_1f_cxdj
11	...	1楼手机配件	index_1f_sjpj
13	...	2楼Logo	index_2f_logo

Id	title	url	image	sequence	category_id
1	美图M8s	http://www.itcast.cn/group1/M00/00/01/CtM3BVRlmc-AJdVSAEI5Wm7zaw8639396		1	1
2	黑色星期五	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		2	1
3	周末365	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		3	1
4	双十一返场一元购	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		4	1
5	17银石低至4199元	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		1	2
6	奥克斯专场 正品...	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		2	2
7	荣耀9青春版 商配...	http://www.itcast.cn/group1/M00/00/01/CtM3BVRl...		3	2

```
{
  "index_lbt": [
    {
      "id": 1,
      "category": 1,
      "title": "美图M8s",
      "url": "http://www.itcast.cn",
      "image": "group1/M00/00/01/CtM3BVRlmc-AJdVSAEI5Wm7zaw8639396",
      "text": "",
      "sequence": 1,
      "status": 1
    },
    {
      "id": 2,
      "category": 1,
      "title": "黑色星期五",
      "url": "http://www.itcast.cn",
      "image": "group1/M00/00/01/CtM3BVRlmiKANEeLAAFFMRWFbY86177278",
      "text": "",
      "sequence": 2,
      "status": 1
    }
  ]
}
```

```

"index_kx": [
    {
        "id": 5,
        "category": 2,
        "title": "i7顽石低至4199元",
        "url": "http://www.itcast.cn",
        "image": "",
        "text": "",
        "sequence": 1,
        "status": 1
    },
    {
        "id": 5,
        "category": 2,
        "title": "奥克斯专场 正1匹空调1313元抢",
        "url": "http://www.itcast.cn",
        "image": "",
        "text": "",
        "sequence": 2,
        "status": 1
    }
]
}

```

结论：

- 首页商品广告数据由广告分类和广告内容组成。
- 广告分类带有标识符 `key`，可以利用它确定广告展示的位置。
- 确定广告展示的位置后，再查询和渲染出该位置的广告内容。
- 广告的内容还有内部的排序字段，决定了广告内容的展示顺序。

2. 查询首页商品广告

```

class IndexView(View):
    """首页广告"""

    def get(self, request):
        """提供首页广告界面"""
        # 查询商品频道和分类
        .....

        # 广告数据
        contents = {}
        content_categories = ContentCategory.objects.all()
        for cat in content_categories:
            contents[cat.key] = cat.content_set.filter(status=True).order_by('sequence')

        # 渲染模板的上下文

```

```

context = {
    'categories': categories,
    'contents': contents,
}
return render(request, 'index.html', context)

```

3. 渲染首页商品广告

1. 轮播图广告

```

<ul class="slide">
    {% for content in contents.index_lbt %}
        <li><a href="{{ content.url }}"></a></li>
    {% endfor %}
</ul>

```

2. 快讯和页头广告

```

<div class="news">
    <div class="news_title">
        <h3>快讯</h3>
        <a href="#">更多 &gt;</a>
    </div>
    <ul class="news_list">
        {% for content in contents.index_kx %}
            <li><a href="{{ content.url }}">{{ content.title }}</a></li>
        {% endfor %}
    </ul>
    {% for content in contents.index_ytgg %}
        <a href="{{ content.url }}" class="advs"></a>
    {% endfor %}
</div>

```

3. 楼层广告（一楼）

```

<div class="list_model">
    <div class="list_title clearfix">
        <h3 class="f1" id="model01">1F 手机通讯</h3>
        <div class="subtitle fr">
            <a @mouseenter="f1_tab=1" :class="f1_tab==1?'active':''">时尚新品</a>
            <a @mouseenter="f1_tab=2" :class="f1_tab==2?'active':''">畅想低价</a>
            <a @mouseenter="f1_tab=3" :class="f1_tab==3?'active':''">手机配件</a>
        </div>
    </div>
    <div class="goods_con clearfix">
        <div class="goods_banner f1">
            

```

```

<div class="channel">
    {% for content in contents.index_1f_pd %}
        <a href="{{ content.url }}">{{ content.title }}</a>
    {% endfor %}
</div>
<div class="key_words">
    {% for content in contents.index_1f_bq %}
        <a href="{{ content.url }}">{{ content.title }}</a>
    {% endfor %}
</div>
<div class="goods_list_con">
    <ul v-show="f1_tab==1" class="goods_list fl">
        {% for content in contents.index_1f_ssxp %}
            <li>
                <a href="{{ content.url }}" class="goods_pic"></a>
                <h4><a href="{{ content.url }}" title="{{ content.title }}">{{ content.title }}</a></h4>
                <div class="price">{{ content.text }}</div>
            </li>
        {% endfor %}
    </ul>
    <ul v-show="f1_tab==2" class="goods_list fl">
        {% for content in contents.index_1f_cxdj %}
            <li>
                <a href="{{ content.url }}" class="goods_pic"></a>
                <h4><a href="{{ content.url }}" title="{{ content.title }}">{{ content.title }}</a></h4>
                <div class="price">{{ content.text }}</div>
            </li>
        {% endfor %}
    </ul>
    <ul v-show="f1_tab==3" class="goods_list fl">
        {% for content in contents.index_1f_sjbj %}
            <li>
                <a href="{{ content.url }}" class="goods_pic"></a>
                <h4><a href="{{ content.url }}" title="{{ content.title }}">{{ content.title }}</a></h4>
                <div class="price">{{ content.text }}</div>
            </li>
        {% endfor %}
    </ul>
</div>
</div>
</div>

```

4.楼层广告（二楼）

```

<div class="list_model model02">
    <div class="list_title clearfix">
        <h3 class="f1" id="model01">2F 电脑数码</h3>
        <div class="subtitle fr">
            <a @mouseenter="f2_tab=1" :class="f2_tab==1?'active':''">加价换购</a>
            <a @mouseenter="f2_tab=2" :class="f2_tab==2?'active':''">畅享低价</a>
        </div>
    </div>
    <div class="goods_con clearfix">
        <div class="goods_banner f1">
            
                {% for content in contents.index_2f_pd %}
                    <a href="{{ content.url }}>{{ content.title }}</a>
                {% endfor %}
            </div>
            <div class="key_words">
                {% for content in contents.index_2f_bq %}
                    <a href="{{ content.url }}>{{ content.title }}</a>
                {% endfor %}
            </div>
        </div>
        <div class="goods_list_con">
            <ul v-show="f2_tab==1" class="goods_list f1">
                {% for content in contents.index_2f_cxdj %}
                    <li>
                        <a href="{{ content.url }}" class="goods_pic">{{ content.title }}</a></h4>
                        <div class="price">{{ content.text }}</div>
                    </li>
                {% endfor %}
            </ul>
            <ul v-show="f2_tab==2" class="goods_list f1">
                {% for content in contents.index_2f_jjhg %}
                    <li>
                        <a href="{{ content.url }}" class="goods_pic">{{ content.title }}</a></h4>
                        <div class="price">{{ content.text }}</div>
                    </li>
                {% endfor %}
            </ul>
        </div>
    </div>
</div>

```

5.楼层广告（三楼）

```

<div class="list_model model03">
    <div class="list_title clearfix">
        <h3 class="f1" id="model01">3F 家居家装</h3>
        <div class="subtitle fr">
            <a @mouseenter="f3_tab=1" :class="f3_tab==1?'active':''">生活用品</a>
            <a @mouseenter="f3_tab=2" :class="f3_tab==2?'active':''">厨房用品</a>
        </div>
    </div>
    <div class="goods_con clearfix">
        <div class="goods_banner f1">
            
                {% for content in contents.index_3f_pd %}
                    <a href="{{ content.url }}>{{ content.title }}</a>
                {% endfor %}
            </div>
            <div class="key_words">
                {% for content in contents.index_3f_bq %}
                    <a href="{{ content.url }}>{{ content.title }}</a>
                {% endfor %}
            </div>
        </div>
        <div class="goods_list_con">
            <ul v-show="f3_tab==1" class="goods_list fl">
                {% for content in contents.index_3f_shyp %}
                    <li>
                        <a href="{{ content.url }}" class="goods_pic">{{ content.title }}</a></h4>
                        <div class="price">{{ content.text }}</div>
                    </li>
                {% endfor %}
            </ul>
            <ul v-show="f3_tab==2" class="goods_list fl">
                {% for content in contents.index_3f_cfyp %}
                    <li>
                        <a href="{{ content.url }}" class="goods_pic">{{ content.title }}</a></h4>
                        <div class="price">{{ content.text }}</div>
                    </li>
                {% endfor %}
            </ul>
        </div>
    </div>
</div>

```


自定义Django文件存储类

思考：

- 下图首页页面中图片无法显示的原因。

The screenshot shows a product listing interface. At the top, there's a navigation bar with tabs like '首页' (Home), '商品' (Products), '服务' (Services), etc. Below the navigation, there's a search bar and a filter section with dropdown menus for '类别' (Category), '品牌' (Brand), '价格' (Price), and '关键词' (Keywords). The main content area displays five products in a grid:

- 1. 美丽M8s - ￥2699.00
- 2. 黑色星期五 - ￥7788.00
- 3. 前卫365 - ￥749.00
- 4. 17层石凳至4199元 - ￥199.00
- 5. 梅丽斯专场 正品 - ￥1299.00

Below the grid, there's a slide-in menu with the following code:

```

<ul class="slide">
    <li><a href="http://www.itcast.cn"></a></li>
    <li><a href="http://www.itcast.cn"></a></li>
    <li><a href="http://www.itcast.cn"></a></li>
    <li><a href="http://www.itcast.cn"></a></li>
</ul>

```

At the bottom, there's a table showing the database records:

ID	Title	url	image	group	group_id	sequence	category_id
1	美丽M8s	http://www.itcast.cn/group1/M00/00/01/CM3BVrLw-AzgRVgAaP13Mw7xaw6539396		1	1	1	1
2	黑色星期五	http://www.itcast.cn/group1/M00/00/01/CM3BVrLm1mXANtGtaAAxFn0WPY8617721B		2	1	1	1
3	前卫365	http://www.itcast.cn/group1/M00/00/01/CM3BVrLmkaAP7IKhAAN8CUTQah41642702		3	1	1	1
4	17层石凳至4199元	http://www.itcast.cn/group1/M00/00/01/CM3BVrLmnaAhtSRhAOlxtuk7uk4998927		4	1	1	1
5	梅丽斯专场 正品	http://www.itcast.cn/group1/M00/00/01/CM3BVrLmnaAhtSRhAOlxtuk7uk4998927		1	2	1	2
6	梅丽斯专场 正品	http://www.itcast.cn/group1/M00/00/01/CM3BVrLmnaAhtSRhAOlxtuk7uk4998927		2	2	1	2
7	梅丽斯专场 正品	http://www.itcast.cn/group1/M00/00/01/CM3BVrLmnaAhtSRhAOlxtuk7uk4998927		3	2	1	2

结论：

- 通过FastDFS上传文件后返回的 'Remote file_id' 字段是文件索引。
- 文件索引会被我们存储到MySQL数据库。所以将来读取出来的也是文件索引，导致界面无法下载到图片。

解决：

- 重写Django文件存储类的url()方法。
- 在重写时拼接完整的图片下载地址（协议、IP、端口、文件索引）

1. Django文件存储类url()方法介绍

```

<img alt="Diagram showing the inheritance path from ImageField to ImageStorage. It starts with 'class ImageField(Field)', which has a dashed arrow pointing to 'class ImageStorage(Storage)'. This indicates that ImageStorage inherits from Field and overrides its methods like 'url()'." data-bbox="118 86 267 180"/>

```

结论：

- 文件存储类 `url()` 方法的作用：返回 `name` 所代表的文件内容的URL。
- 文件存储类 `url()` 方法的触发：`content.image.url`
 - 虽然表面上调用的是 `ImageField` 的 `url` 方法。但是内部会去调用文件存储类的 `url()` 方法。
- 文件存储类 `url()` 方法的使用：
 - 我们可以通过自定义Django文件存储类达到重写 `url()` 方法的目的，从而得到图片全路径。

2. 自定义Django文件存储类

[自定义文件存储类的官方文档](#)

```

class FastDFSStorage(Storage):
    """自定义文件存储系统"""

    def _open(self, name, mode='rb'):
        """
        用于打开文件
        :param name: 要打开的文件的名字
        :param mode: 打开文件方式
        :return: None
        """

        # 打开文件时使用的，此时不需要，而文档告诉说明必须实现，所以pass
        pass

    def _save(self, name, content):
        """
        用于保存文件
        :param name: 要保存的文件名字
        :param content: 要保存的文件的内容
        :return: None
        """

        # 保存文件时使用的，此时不需要，而文档告诉说明必须实现，所以pass
        pass

```

3. 重写Django文件存储类url()方法

[1.重写 url\(\) 方法](#)

```

class FastDFSStorage(Storage):
    """自定义文件存储系统，修改存储的方案"""
    def __init__(self, fdfs_base_url=None):
        """
        构造方法，可以不带参数，也可以携带参数
        :param base_url: Storage的IP
        """
        self.fdfs_base_url = fdfs_base_url or settings.FDFS_BASE_URL

    def _open(self, name, mode='rb'):
        .....

    def _save(self, name, content):
        .....

    def url(self, name):
        """
        返回name所指文件的绝对URL
        :param name: 要读取文件的引用:group1/M00/00/00/wKhnnlxw_gmAcowmAAEXU5wmjPs35.j
peg
        :return: http://192.168.103.158:8888/group1/M00/00/00/wKhnnlxw_gmAcowmAAEXU
5wmjPs35.jpeg
        """
        # return 'http://192.168.103.158:8888/' + name
        # return 'http://image.meiduo.site:8888/' + name
        return self.fdfs_base_url + name

```

2.相关配置参数

```

# 指定自定义的Django文件存储类
DEFAULT_FILE_STORAGE = 'meiduo_mall.utils.fastdfs.fdfs_storage.FastDFSStorage'

# FastDFS相关参数
# FDFS_BASE_URL = 'http://192.168.103.158:8888/'
FDFS_BASE_URL = 'http://image.meiduo.site:8888/'

```

3.添加访问图片的域名

- 在 /etc/hosts 中添加访问Storage的域名

```

$ Storage的IP      域名
$ 192.168.103.158  image.meiduo.site

```

4.文件存储类 url() 方法的使用

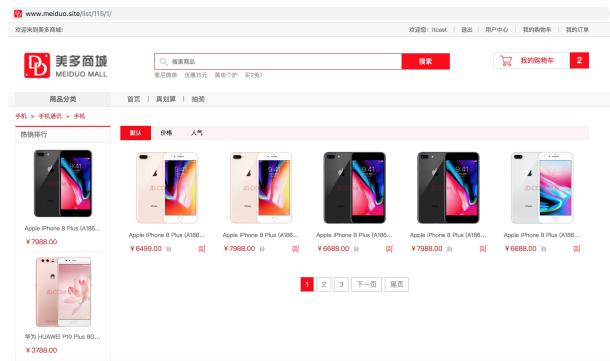
- 以图片轮播图为例： content.image.url

```
<ul class="slide">
    {% for content in contents.index_lbt %}
        <li><a href="{{ content.url }}></a></li>
    {% endfor %}
</ul>
```



商品列表页

商品列表页分析



1. 商品列表页组成结构分析

1.商品频道分类

- 已经提前封装在 `contents.utils.py` 文件中，直接调用即可。

2.面包屑导航

- 可以使用三级分类ID，查询出该类型商品的三级分类数据。

3.排序和分页

- 无论如何排序和分页，商品的分类不能变。
- 排序时需要知道当前排序方式。
- 分页时需要知道当前分页的页码，且每页五条商品记录。

4.热销排行

- 热销排行中的商品分类要和排序、分页的商品分类一致。
- 热销排行是查询出指定分类商品销量前二的商品。
- 热销排行使用Ajax实现局部刷新的效果。

2. 商品列表页接口设计和定义

1.请求方式

选项	方案
请求方法	GET
请求地址	<code>/list/(?P<category_id>\d+)/(?P<page_num>\d+)/?sort=排序方式</code>

```
# 按照商品创建时间排序（默认）
http://www.meiduo.site:8000/list/115/1/
http://www.meiduo.site:8000/list/115/1/?sort=default
# 按照商品价格由低到高排序
```

```
# 按照商品销量由高到低排序
http://www.meiduo.site:8000/list/115/1/?sort=hot
# 用户随意传递排序规则
http://www.meiduo.site:8000/list/115/1/?sort=itcast
```

2.请求参数：路径参数 和 查询参数

参数名	类型	是否必传	说明
category_id	string	是	商品分类ID, 第三级分类
page_num	string	是	当前页码
sort	string	否	排序方式

3.响应结果：HTML

```
list.html
```

4.接口定义

```
class ListView(View):
    """商品列表页"""

    def get(self, request, category_id, page_num):
        """提供商品列表页"""

        # 校验参数category_id
        try:
            category = GoodsCategory.objects.get(id=category_id)
        except GoodsCategory.DoesNotExist:
            return http.HttpResponseNotFound('GoodsCategory does not exist')

        return render(request, 'list.html')
```

列表页面包屑导航

重要提示：路径参数category_id是商品第三级分类

1. 查询列表页面包屑导航数据

提示：对包屑导航数据的查询进行封装，方便后续直接使用。

goods.utils.py

```
def get_breadcrumb(category):
    """
    获取面包屑导航
    :param category: 商品类别
    :return: 面包屑导航字典
    """
    breadcrumb = dict(
        cat1='',
        cat2='',
        cat3=''
    )
    if category.parent is None:
        # 当前类别为一级类别
        breadcrumb['cat1'] = category
    elif category.subs.count() == 0:
        # 当前类别为三级
        breadcrumb['cat3'] = category
        cat2 = category.parent
        breadcrumb['cat2'] = cat2
        breadcrumb['cat1'] = cat2.parent
    else:
        # 当前类别为二级
        breadcrumb['cat2'] = category
        breadcrumb['cat1'] = category.parent

    return breadcrumb
```

```
class ListView(View):
    """商品列表页"""

    def get(self, request, category_id, page_num):
        """提供商品列表页"""
        # 判断category_id是否正确
        try:
            category = models.GoodsCategory.objects.get(id=category_id)
        except models.GoodsCategory.DoesNotExist:
            return http.HttpResponseNotFound('GoodsCategory does not exist')
```

```
# 查询商品频道分类:封装在contents.utils.py
categories = get_categories()
# 查询面包屑导航
breadcrumb = get_breadcrumb(category)

# 渲染页面
context = {
    'categories':categories,
    'breadcrumb':breadcrumb
}
return render(request, 'list.html', context)
```

2. 渲染列表页面包屑导航数据

```
<div class="breadcrumb">
    <a href="javascript:; ">{{ breadcrumb.cat1.name }}</a>
    <span>></span>
    <a href="javascript:; ">{{ breadcrumb.cat2.name }}</a>
    <span>></span>
    <a href="javascript:; ">{{ breadcrumb.cat3.name }}</a>
</div>
```

列表页分页和排序

```
# 按照商品创建时间排序（默认）
http://www.meiduo.site:8000/list/115/1/
http://www.meiduo.site:8000/list/115/1/?sort=default
# 按照商品价格由低到高排序
http://www.meiduo.site:8000/list/115/1/?sort=price
# 按照商品销量由高到低排序
http://www.meiduo.site:8000/list/115/1/?sort=hot
# 用户随意传递排序规则
http://www.meiduo.site:8000/list/115/1/?sort=itcast
```

1. 查询列表页分页和排序数据

```
class ListView(View):
    """商品列表页"""

    def get(self, request, category_id, page_num):
        """提供商品列表页"""
        # 判断category_id是否正确
        try:
            category = models.GoodsCategory.objects.get(id=category_id)
        except models.GoodsCategory.DoesNotExist:
            return http.HttpResponseNotFound('GoodsCategory does not exist')
        # 接收sort参数：如果用户不传，就是默认的排序规则
        sort = request.GET.get('sort', 'default')

        # 查询商品频道分类
        categories = get_categories()
        # 查询面包屑导航
        breadcrumb = get_breadcrumb(category)

        # 按照排序规则查询该分类商品SKU信息
        if sort == 'price':
            # 按照价格由低到高
            sort_field = 'price'
        elif sort == 'hot':
            # 按照销量由高到低
            sort_field = '-sales'
        else:
            # 'price'和'sales'以外的所有排序方式都归为'default'
            sort = 'default'
            sort_field = 'create_time'
        skus = models.SKU.objects.filter(category=category, is_launched=True).order_by(sort_field)
```

```

# 创建分页器: 每页N条记录
paginator = Paginator(skus, constants.GOODS_LIST_LIMIT)
# 获取每页商品数据
try:
    page_skus = paginator.page(page_num)
except EmptyPage:
    # 如果page_num不正确, 默认给用户404
    return http.HttpResponseNotFound('empty page')
# 获取列表页总页数
total_page = paginator.num_pages

# 渲染页面
context = {
    'categories': categories,      # 商品分类
    'breadcrumb': breadcrumb,     # 面包屑导航
    'sort': sort,                 # 排序字段
    'category': category,         # 第三级分类
    'page_skus': page_skus,       # 分页后数据
    'total_page': total_page,     # 总页数
    'page_num': page_num,         # 当前页码
}
return render(request, 'list.html', context)

```

2. 渲染列表页分页和排序数据

1. 渲染分页和排序数据

```

<div class="r_wrap fr clearfix">
    <div class="sort_bar">
        <a href="{{ url('goods:list', args=(category.id, 1)) }}?sort=default" {% if
sort == 'default' %}class="active"{% endif %}>默认</a>
        <a href="{{ url('goods:list', args=(category.id, 1)) }}?sort=price" {% if s
ort == 'price' %}class="active"{% endif %}>价格</a>
        <a href="{{ url('goods:list', args=(category.id, 1)) }}?sort=hot" {% if sort
== 'hot' %}class="active"{% endif %}>人气</a>
    </div>
    <ul class="goods_type_list clearfix">
        {% for sku in page_skus %}
        <li>
            <a href="detail.html"></a>
            <h4><a href="detail.html">{{ sku.name }}</a></h4>
            <div class="operate">
                <span class="price">¥{{ sku.price }}</span>
                <span class="unit">台</span>
                <a href="#" class="add_goods" title="加入购物车"></a>
            </div>
        </li>
        {% endfor %}
    </ul>

```

```
</div>
```



2.列表页分页器

首页 上一页 1 2 3 下一页 尾页

准备分页器标签

```
<div class="r_wrap fr clearfix">  
    ....  
    <div class="pagination">  
        <div id="pagination" class="page"></div>  
    </div>  
</div>
```

```
# 导入样式时放在最前面导入  
<link rel="stylesheet" type="text/css" href="{{ static('css/jquery.pagination.css') }}>
```

准备分页器交互

```
<script type="text/javascript" src="{{ static('js/jquery.pagination.min.js') }}></script>
```

```
<script type="text/javascript">  
$(function () {  
    $('#pagination').pagination({  
        currentPage: {{ page_num }},  
        totalPage: {{ total_page }},  
        callback: function (current) {  
            if(location.href = '/list/115/1/?sort=default';#)  
                location.href = '/list/{{ category.id }}/' + current + '?sort={{ s  
ort }}';  
        }  
    })  
});  
</script>
```

列表页热销排行

根据路径参数 category_id 查询出该类型商品销量前二的商品。

使用Ajax实现局部刷新的效果。

1. 查询列表页热销排行数据

1. 请求方式

选项	方案
请求方法	GET
请求地址	/hot/(?P<category_id>\d+)/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
category_id	string	是	商品分类ID, 第三级分类

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息
hot_skus[]	热销SKU列表
id	SKU编号
default_image_url	商品默认图片
name	商品名称
price	商品价格

```
{
    "code": "0",
    "errmsg": "OK",
    "hot_skus": [
        {
            "id": 6,
            "default_image_url": "http://image.meiduo.site:8888/group1/M00/00/02/CTM3BVrRbI2ARekNAAFZsBqChgk3141998",
            "name": "Apple iPhone 8 Plus (A1864) 256GB 深空灰色 移动联通电信4G手机",
            "price": "7988.00"
        },
        {
            "id": 14,
            "default_image_url": "http://image.meiduo.site:8888/group1/M00/00/02/CTM3BVrRbI2ARekNAAFZsBqChgk3141999",
            "name": "Apple iPhone X 256GB 深空灰色 移动联通电信4G手机",
            "price": "8999.00"
        }
    ]
}
```

```

        "default_image_url": "http://image.meiduo.site:8888/group1/M00/00/02/CTM
3BVrRdMSAAuDAAVs1h9vkK04466364",
        "name": "华为 HUAWEI P10 Plus 6GB+128GB 玫瑰金 移动联通电信4G手机 双卡双待",
        "price": "3788.00"
    }
]
}

```

4. 接口定义和实现

```

class HotGoodsView(View):
    """商品热销排行"""

    def get(self, request, category_id):
        """提供商品热销排行JSON数据"""
        # 根据销量倒序
        skus = models.SKU.objects.filter(category_id=category_id, is_launched=True)
        .order_by('-sales')[2]

        # 序列化
        hot_skus = []
        for sku in skus:
            hot_skus.append({
                'id': sku.id,
                'name': sku.name,
                'price': sku.price,
                'default_image_url': sku.default_image.url
            })

        return http.JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'hot_skus': hot_skus})

```

2. 渲染列表页热销排行数据

1. 模板数据 category_id 传递到Vue.js

```

<script type="text/javascript">
    let category_id = "{{ category.id }}";
</script>

```

```

data: {
    category_id: category_id,
},

```

2. Ajax请求商品热销排行JSON数据

```
get_hot_skus(){
```

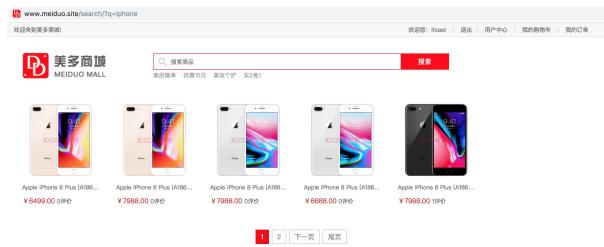
```
if (this.category_id) {
    let url = '/hot/' + this.category_id + '/';
    axios.get(url, {
        responseType: 'json'
    })
    .then(response => {
        this.hot_skus = response.data.hot_skus;
        for(let i=0; i<this.hot_skus.length; i++){
            this.hot_skus[i].url = '/detail/' + this.hot_skus[i].id + '/';
        }
    })
    .catch(error => {
        console.log(error.response);
    })
},
},
```

3.渲染商品热销排行界面

```
<div class="new_goods" v-cloak>
    <h3>热销排行</h3>
    <ul>
        <li v-for="sku in hot_skus">
            <a :href="sku.url"></a>
            <h4><a :href="sku.url">[[ sku.name ]]</a></h4>
            <div class="price">¥ [[ sku.price ]]</div>
        </li>
    </ul>
</div>
```

商品搜索

全文检索方案Elasticsearch



1. 全文检索和搜索引擎原理

商品搜索需求

- 当用户在搜索框输入商品关键字后，我们要为用户提供相关的商品搜索结果。

商品搜索实现

- 可以选择使用模糊查询 like 关键字实现。
- 但是 like 关键字的效率极低。
- 查询需要在多个字段中进行，使用 like 关键字也不方便。

全文检索方案

- 我们引入全文检索的方案来实现商品搜索。
- 全文检索即在指定的任意字段中进行检索查询。
- 全文检索方案需要配合搜索引擎来实现。

搜索引擎原理

- 搜索引擎**进行全文检索时，会对数据库中的数据进行一遍预处理，单独建立起一份索引结构数据。
- 索引结构数据类似新华字典的索引检索页，里面包含了关键词与词条的对应关系，并记录词条的位置。
- 搜索引擎进行全文检索时，将关键字在索引数据中进行快速对比查找，进而找到数据的真实存储位置。

100 难检字笔画索引										101 难检字笔画索引																			
(三) 难检字笔画索引																													
(字右边的号码指正文的页码; 带圆括号的字是繁体字或异体字。)																													
一画	丈	611	卫	501	不	39	下	28	635	冉	410	(逃)	87	年	353	更	152	(亞)	549										
○ 301	与	584	子	251	平	111	为	499	可	丝	44	朱	633	153	其	384	单	250											
乙 566		586	也	561	有	326		502	四	6	六画	丢	102	東	449	直	624	48											
二画		587	飞	125	牙	547	尹	573	生	141	戌	414	乔	395	两	294	(來)	83											
丁 100	万	341	习	514	屯	491	尺	53	生	436	考	259	丘	376	丽	284	喪	427											
620		496	乡	523		641		652	失	438	老	280	丘	364	丽	289	肅	458											
七 382	上	429	四画		互	190	大	164	右	583	乍	606	亚	549	向	525	来	275											
又 567		429	丰	130	(毋)	410	(弔)	98	布	丝	402	亘	152	肉	533	半	333	(東)	102										
匕 23	千	389	斤	383	中	630	丑	63	戊	60	𠂇	60	更	289	后	186	串	67	承	57									
九 244	乞	386	开	255	井	631	巴	8	平	冠	60	𠂇	624	再	598	角	310	𠂇	290										
刁 98	川	66	井	241	内	351	以	566	东	102	手	187	(夏)	152	亮	614	我	506	九画										
了 282	久	244	天	477	牛	509	予	584	(戊)	187	从	73	戌	539	舛	67	(鬼)	489	奏	649									
297	么	321	夫	133	壬	413		586	卡	187	用	579	在	598	产	49	圓	73	卖	322									
乃 347		327	十	134	升	436	书	446	255	𠂇	579	𠂇	624	𠂇	535	坐	653	哉	598										
也 337		557	元	590	夭	558	五画		510	𠂇	593	𠂇	624	死	454	兴	533	些	528										
355	丸	495	无	508	长	50	未	502	凸	50	乐	281	死	454	𠂇	535	龟	168	菴	435									
三画	及	207	云	595		611	末	341	归	187	𠂇	593	𠂇	624	成	56	农	357	暢	52									
三 421	亡	497	专	638	反	122	击	205	且	187	𠂇	593	𠂇	624	至	13	果	171	菴	177									
干 142	·	508	𠂇	142	爻	558	戈	218	13	𠂇	593	𠂇	624	𠂇	106	𠂇	403	菴	220										
145	丫	547	廿	354	乏	119	正	620	且	13	𠂇	593	𠂇	624	𠂇	236	卿	314	威	498									
于 65	义	567	五	509	氏	441		621	𠂇	13	𠂇	593	𠂇	624	𠂇	237	島	88	秉	34									
于 584	之	622	(市)	597		623	甘	143	电	13	𠂇	593	𠂇	624	𠂇	238	𠂇	585	面	335									
亏 270	已	566	支	623	丹	83	世	442	由	13	𠂇	593	𠂇	624	𠂇	239	𠂇	586	周	632									
才 40	巳	454	𠂇	335	鸟	507	本	21	史	13	𠂇	593	𠂇	624	𠂇	240	𠂇	587	𠂇	580									
下 518	子	231	卅	420		510	术	448	央	13	𠂇	593	𠂇	624	𠂇	241	𠂇	588	𠂇	581									

结论：

- 搜索引擎建立索引结构数据，类似新华字典的索引检索页，全文检索时，关键字在索引数据中进行快速对比查找，进而找到数据的真实存储位置。

2. Elasticsearch介绍

实现全文检索的搜索引擎，首选的是 Elasticsearch。

- Elasticsearch 是用 Java 实现的，开源的搜索引擎。
- 它可以快速地储存、搜索和分析海量数据。维基百科、Stack Overflow、Github 等都采用它。
- Elasticsearch 的底层是开源库 Lucene。但是，没法直接使用 Lucene，必须自己写代码去调用它的接口。

分词说明

- 搜索引擎在对数据构建索引时，需要进行分词处理。
- 分词是指将一句话拆解成多个单字 或 词，这些字或词便是这句话的关键词。
- 比如：我是中国人
 - 分词后：我、是、中、国、人、中国 等等都可以是这句话的关键词。
- Elasticsearch 不支持对中文进行分词建立索引，需要配合扩展 elasticsearch-analysis-ik 来实现中文分词处理。

3. 使用Docker安装Elasticsearch

1. 获取Elasticsearch-ik镜像

```
# 从仓库拉取镜像
$ sudo docker image pull delron/elasticsearch-ik:2.4.6-1.0
# 解压教学资料中本地镜像
$ sudo docker load -i elasticsearch-ik-2.4.6_docker.tar
```

2.配置Elasticsearch-ik

- 将教学资料中的 elasticsearc-2.4.6 目录拷贝到 home 目录下。
- 修改 /home/python/elasticsearc-2.4.6/config/elasticsearch.yml 第54行。
- 更改ip地址为本机真实ip地址。

```
52 # Set the bind address to a specific IP (IPv4 or IPv6):
53 #
54 network.host: 192.168.103.158
```

3.使用Docker运行Elasticsearch-ik

```
$ sudo docker run -dti --name=elasticsearch --network=host -v /home/python/elasticsearc-2.4.6/config:/usr/share/elasticsearch/config delron/elasticsearch-ik:2.4.6-1.0
```

```
python@ubuntu:~/elasticsearch-2.4.6/config$ sudo docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
ubuntu2_1mp         latest   ed085a8652fe  4 days ago    117 MB
ubuntu              16.04    7e87e2b3bf7a  5 weeks ago   117 MB
delron/fastdfs      latest   8487e8bfc0ee  18 months ago  464 MB
delron/elasticsearch-ik  2.4.6-1.0  095b6487fb77  18 months ago  689 MB

python@ubuntu:~/elasticsearch-2.4.6/config$ sudo docker container ls -all
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
2ffdfcfc688        delron/elasticsearch-ik:2.4.6-1.0   "/docker-entrypoint..."   3 minutes ago     Up 3 minutes          0.0.0.0:9200->9200/tcp   elasticsearch
fc5c0f01151        delron/fastdfs           "/usr/bin/start15..."   4 days ago       Up 4 days           0.0.0.0:2379->2379/tcp   fastdfs
```

Haystack扩展建立索引

提示：

- Elasticsearch 的底层是开源库 [Lucene](#)。但是没法直接使用 Lucene，必须自己写代码去调用它的接口。

思考：

- 我们如何对接 Elasticsearch服务端？

解决方案：

- Haystack

1. Haystack介绍和安装配置

1.Haystack介绍

- Haystack 是在Django中对接搜索引擎的框架，搭建了用户和搜索引擎之间的沟通桥梁。
 - 我们在Django中可以通过使用 Haystack 来调用 Elasticsearch 搜索引擎。
- Haystack 可以在不修改代码的情况下使用不同的搜索后端（比如 Elasticsearch 、 Whoosh 、 Solr 等等）。

2.Haystack安装

```
$ pip install django-haystack
$ pip install elasticsearch==2.4.1
```

3.Haystack注册应用和路由

```
INSTALLED_APPS = [
    'haystack', # 全文检索
]
```

```
url(r'^search/', include('haystack.urls')),
```

4.Haystack配置

- 在配置文件中配置Haystack为搜索引擎后端

```
# Haystack
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://192.168.103.158:9200/'}, # Elasticsearch服务器ip地址, 端口号固定
```

```

为9200
    'INDEX_NAME': 'meiduo_mall', # Elasticsearch建立的索引库的名称
},
}

# 当添加、修改、删除数据时，自动生成索引
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'

```

重要提示：

- **HAYSTACK_SIGNAL_PROCESSOR** 配置项保证了在Django运行起来后，有新的数据产生时， Haystack仍然可以让Elasticsearch实时生成新数据的索引

2. Haystack建立数据索引

1. 创建索引类

- 通过创建索引类，来指明让搜索引擎对哪些字段建立索引，也就是可以通过哪些字段的关键字来检索数据。
- 本项目中对SKU信息进行全文检索，所以在 `goods` 应用中新建 `search_indexes.py` 文件，用于存放索引类。

```

from haystack import indexes

from .models import SKU


class SKUIndex(indexes.SearchIndex, indexes.Indexable):
    """SKU索引数据模型类"""
    text = indexes.CharField(document=True, use_template=True)

    def get_model(self):
        """返回建立索引的模型类"""
        return SKU

    def index_queryset(self, using=None):
        """返回要建立索引的数据查询集"""
        return self.get_model().objects.filter(is_launched=True)

```

- 索引类SKUIndex说明：
 - 在 `SKUIndex` 建立的字段，都可以借助 `Haystack` 由 `Elasticsearch` 搜索引擎查询。
 - 其中 `text` 字段我们声明为 `document=True`，表明该字段是主要进行关键字查询的字段。
 - `text` 字段的索引值可以由多个数据库模型类字段组成，具体由哪些模型类字段组成，我们用 `use_template=True` 表示后续通过模板来指明。

2. 创建 `text` 字段索引值模板文件

- 在 `templates` 目录中创建 `text` 字段使用的模板文件

- 具体在 `templates/search/indexes/goods/sku_text.txt` 文件中定义

```
{{ object.id }}
{{ object.name }}
{{ object.caption }}
```

- 模板文件说明：当将关键词通过`text`参数名传递时
 - 此模板指明SKU的 `id`、`name`、`caption` 作为 `text` 字段的索引值来进行关键字索引查询。

3. 手动生成初始索引

```
$ python manage.py rebuild_index
```

```
(meiduo_mall) Python:~/projects/meiduo_project/meiduo_mall$ python manage.py rebuild_index
WARNING: This will irreparably remove EVERYTHING from your search index in connection 'default'.
Your choices after this are to restore from backups or rebuild via the `rebuild_index` command.
Are you sure you wish to continue? [y/N] y
Removing all documents from your index because you said so.
All documents removed.
Indexing 16 商品SKU
GET /meiduo_mall/_mapping [status:404 request:0.010s]
```

3. 全文检索测试

1. 准备测试表单

- 请求方法： `GET`
- 请求地址： `/search/`
- 请求参数： `q`



The screenshot shows a search interface with the following elements:

- A search bar with a magnifying glass icon and the placeholder "搜索商品".
- A red "搜索" (Search) button.
- Below the search bar, there are four small links: "索尼微单", "优惠15元", "美妆个护", and "买2免1".
- The page content area displays the following HTML code:

```
<div class="search_wrap fl">
    <form method="get" action="/search/" class="search_con">
        <input type="text" class="input_text fl" name="q" placeholder="搜索商品">
        <input type="submit" class="input_btn fr" name="" value="搜索">
    </form>
    <ul class="search_suggest fl">
        <li><a href="#">索尼微单</a></li>
        <li><a href="#">优惠15元</a></li>
        <li><a href="#">美妆个护</a></li>
        <li><a href="#">买2免1</a></li>
    </ul>
</div>
```

2. 全文检索测试结果

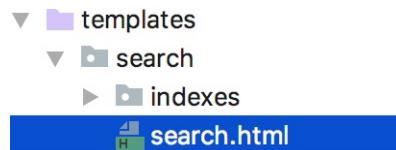


结论：

- 错误提示告诉我们在 templates/search/ 目录中缺少一个 search.html 文件
- search.html 文件作用就是接收和渲染全文检索的结果。

渲染商品搜索结果

1. 准备商品搜索结果页面



2. 渲染商品搜索结果

Haystack返回的数据包括：

- `query`：搜索关键字
- `paginator`：分页paginator对象
- `page`：当前页的page对象（遍历 `page` 中的对象，可以得到 `result` 对象）
- `result.objects`：当前遍历出来的SKU对象。

```
<div class="main_wrap clearfix">
    <div class="clearfix">
        <ul class="goods_type_list clearfix">
            {% for result in page %}
                <li>
                    {%# object取得才是sku对象 #}
                    <a href="detail.html">{{ result.object.name }}</a></h4>
                    <div class="operate">
                        <span class="price">¥{{ result.object.price }}</span>
                        <span>{{ result.object.comments }}评价</span>
                    </div>
                </li>
            {% else %}
                <p>没有找到您要查询的商品。</p>
            {% endfor %}
        </ul>
        <div class="pagination">
            <div id="pagination" class="page"></div>
        </div>
    </div>
</div>
```



3. Haystack搜索结果分页

1. 设置每页返回数据条数

- 通过 HAYSTACK_SEARCH_RESULTS_PER_PAGE 可以控制每页显示数量
- 每页显示五条数据： HAYSTACK_SEARCH_RESULTS_PER_PAGE = 5

2. 准备搜索页分页器

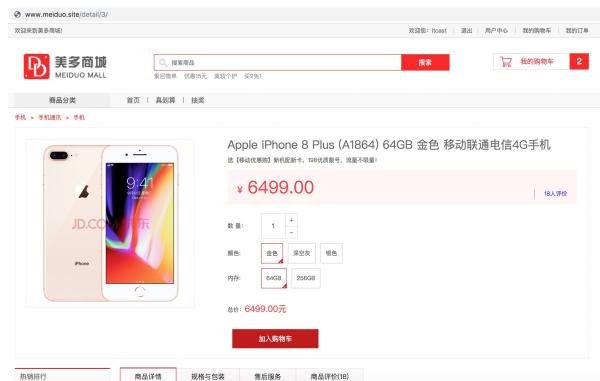
```
<div class="main_wrap clearfix">
    <div class="clearfix">
        .....
        <div class="pagenation">
            <div id="pagination" class="page"></div>
        </div>
    </div>
</div>
```

```
<script type="text/javascript">
$(function () {
    $('#pagination').pagination({
        currentPage: {{ page.number }},
        totalPage: {{ paginator.num_pages }},
        callback:function (current) {
            $('#window.location.href = '/search/?q=iphone&page=' + current;
            window.location.href = '/search/?q={{ query }}&page=' + current;
        }
    });
}</script>
```



商品详情页

商品详情页分析和准备



1. 商品详情页组成结构分析

1.商品频道分类

- 已经提前封装在 `contents.utils.py` 文件中，直接调用方法即可。

2.面包屑导航

- 已经提前封装在 `goods.utils.py` 文件中，直接调用方法即可。

3.热销排行

- 该接口已经在商品列表页中实现完毕，前端直接调用接口即可。

4.商品SKU信息(详情信息、详情介绍、规格与包装、售后服务)

- 通过 `sku_id` 可以找到SKU信息，然后渲染模板即可。

5.SKU规格信息

- 通过 `sku` 可以找到SPU规格和SKU规格信息。

6.商品评价

- 商品评价需要在生成了订单，对订单商品进行评价后再实现，商品评价信息是动态数据。
- 使用Ajax实现局部刷新效果。

2. 商品详情页接口设计和定义

1.请求方式

选项	方案
请求方法	GET
请求地址	/detail/(?P<sku_id>\d+)/

2.请求参数：路径参数

参数名	类型	是否必传	说明
sku_id	string	是	商品SKU编号

3.响应结果: HTML

```
detail.html
```

4.接口定义

```
class DetailView(View):
    """商品详情页"""

    def get(self, request, sku_id):
        """提供商品详情页"""
        return render(request, 'detail.html')
```

3. 商品详情页初步渲染

渲染商品频道分类、面包屑导航、商品热销排行

- 将原先在商品列表页实现的代码拷贝到商品详情页即可。
- 添加 detail.js

```
class DetailView(View):
    """商品详情页"""

    def get(self, request, sku_id):
        """提供商品详情页"""
        # 获取当前sku的信息
        try:
            sku = models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return render(request, '404.html')

        # 查询商品频道分类
        categories = get_categories()
        # 查询面包屑导航
        breadcrumb = get_breadcrumb(sku.category)

        # 渲染页面
        context = {
            'categories': categories,
            'breadcrumb': breadcrumb,
            'sku': sku
        }
        return render(request, 'detail.html', context)
```

提示：为了让前端在获取商品热销排行数据时，能够拿到商品分类ID，我们将商品分类ID从模板传入到Vue.js

```
<script type="text/javascript">
    let category_id = "{{ sku.category.id }}";
</script>
```

```
data: {
    category_id: category_id,
},
```

展示详情页数据

1. 查询和渲染SKU详情信息

```
# 渲染页面
context = {
    'categories':categories,
    'breadcrumb':breadcrumb,
    'sku':sku,
}
return render(request, 'detail.html', context)
```

```
<div class="goods_detail_con clearfix">
    <div class="goods_detail_pic fl"></div>
    <div class="goods_detail_list fr">
        <h3>{{ sku.name }}</h3>
        <p>{{ sku.caption }}</p>
        <div class="price_bar">
            <span class="show_pirce">¥<em>{{ sku.price }}</em></span>
            <a href="javascript:;" class="goods_judge">18人评价</a>
        </div>
        <div class="goods_num clearfix">
            <div class="num_name fl">数 量: </div>
            <div class="num_add fl">
                <input v-model="sku_count" @blur="check_sku_count" type="text" class="num_show fl">
                <a @click="on_addition" class="add fr">+</a>
                <a @click="on_minus" class="minus fr">-</a>
            </div>
        </div>
        {#...商品规格...#}
        <div class="total" v-cloak>总价: <em>[[ sku_amount ]]元</em></div>
        <div class="operate_btn">
            <a href="javascript:;" class="add_cart" id="add_cart">加入购物车</a>
        </div>
    </div>
</div>
```

提示：为了实现用户选择商品数量的局部刷新效果，我们将商品单价从模板传入到Vue.js

```
<script type="text/javascript">
    let sku_price = "{{ sku.price }}";
</script>
```

```
data: {
    sku_price: sku_price,
},
```

2. 渲染详情、包装和售后信息

商品详情、包装和售后信息被归类到商品SPU中，`sku.spu` 关联查询就可以找到该SKU的SPU信息。

```
<div class="r_wrap fr clearfix">
    <ul class="detail_tab clearfix">
        <li @click="on_tab_content('detail')" :class="tab_content.detail?'active':''">商品详情</li>
        <li @click="on_tab_content('pack')" :class="tab_content.pack?'active':''">规格与包装</li>
        <li @click="on_tab_content('service')" :class="tab_content.service?'active':''">售后服务</li>
        <li @click="on_tab_content('comment')" :class="tab_content.comment?'active':''">商品评价(18)</li>
    </ul>
    <div @click="on_tab_content('detail')" class="tab_content" :class="tab_content.detail?'current':''">
        <dl>
            <dt>商品详情: </dt>
            <dd>{{ sku.spu.desc_detail|safe }}</dd>
        </dl>
    </div>
    <div @click="on_tab_content('pack')" class="tab_content" :class="tab_content.pack?'current':''">
        <dl>
            <dt>规格与包装: </dt>
            <dd>{{ sku.spu.desc_pack|safe }}</dd>
        </dl>
    </div>
    <div @click="on_tab_content('service')" class="tab_content" :class="tab_content.service?'current':''">
        <dl>
            <dt>售后服务: </dt>
            <dd>{{ sku.spu.desc_service|safe }}</dd>
        </dl>
    </div>
    <div @click="on_tab_content('comment')" class="tab_content" :class="tab_content.comment?'current':''">
        <ul class="judge_list_con">
            {#...商品评价...#}
        </ul>
    </div>
</div>
```

3. 查询和渲染SKU规格信息

1. 查询SKU规格信息

```

class DetailView(View):
    """商品详情页"""

    def get(self, request, sku_id):
        """提供商品详情页"""
        # 获取当前sku的信息
        try:
            sku = models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return render(request, '404.html')

        # 查询商品频道分类
        categories = get_categories()
        # 查询面包屑导航
        breadcrumb = get_breadcrumb(sku.category)

        # 构建当前商品的规格键
        sku_specs = sku.specs.order_by('spec_id')
        sku_key = []
        for spec in sku_specs:
            sku_key.append(spec.option.id)
        # 获取当前商品的所有SKU
        skus = sku.spu.sku_set.all()
        # 构建不同规格参数（选项）的sku字典
        spec_sku_map = {}
        for s in skus:
            # 获取sku的规格参数
            s_specs = s.specs.order_by('spec_id')
            # 用于形成规格参数-sku字典的键
            key = []
            for spec in s_specs:
                key.append(spec.option.id)
            # 向规格参数-sku字典添加记录
            spec_sku_map[tuple(key)] = s.id
        # 获取当前商品的规格信息
        goods_specs = sku.spu.specs.order_by('id')
        # 若当前sku的规格信息不完整，则不再继续
        if len(sku_key) < len(goods_specs):
            return
        for index, spec in enumerate(goods_specs):
            # 复制当前sku的规格键
            key = sku_key[:]
            # 该规格的选项
            spec_options = spec.options.all()
            for option in spec_options:
                # 在规格参数sku字典中查找符合当前规格的sku

```

```
key[index] = option.id
option.sku_id = spec_sku_map.get(tuple(key))
spec.spec_options = spec_options

# 渲染页面
context = {
    'categories':categories,
    'breadcrumb':breadcrumb,
    'sku':sku,
    'specs': goods_specs,
}
return render(request, 'detail.html', context)
```

2.渲染SKU规格信息

```
{% for spec in specs %}
<div class="type_select">
    <label>{{ spec.name }}:</label>
    {% for option in spec.spec_options %}
        {% if option.sku_id == sku.id %}
            <a href="javascript:;" class="select">{{ option.value }}</a>
        {% elif option.sku_id %}
            <a href="{{ url('goods:detail', args=(option.sku_id, )) }}>{{ option.value }}</a>
        {% else %}
            <a href="javascript;">{{ option.value }}</a>
        {% endif %}
    {% endfor %}
</div>
{% endfor %}
```

统计分类商品访问量

提示：

- 统计分类商品访问量 是统计一天内该类别的商品被访问的次数。
- 需要统计的数据，包括商品分类，访问次数，访问时间。
- 一天内，一种类别，统计一条记录。

	id	count	date	category_id
1	1	2	2019-02-28	115
2	2	1	2019-02-28	157

1. 统计分类商品访问量模型类

模型类定义在 `goods.models.py` 中，然后完成迁移建表。

```
class GoodsVisitCount(BaseModel):
    """统计分类商品访问量模型类"""
    category = models.ForeignKey(GoodsCategory, on_delete=models.CASCADE, verbose_name='商品分类')
    count = models.IntegerField(verbose_name='访问量', default=0)
    date = models.DateField(auto_now_add=True, verbose_name='统计日期')

    class Meta:
        db_table = 'tb_goods_visit'
        verbose_name = '统计分类商品访问量'
        verbose_name_plural = verbose_name
```

2. 统计分类商品访问量后端逻辑

1. 请求方式

选项	方案
请求方法	POST
请求地址	/detail/visit/(?P<category_id>\d+)/

2. 请求参数：路径参数

参数名	类型	是否必传	说明
category_id	string	是	商品分类ID, 第三级分类

3. 响应结果：JSON

字段	说明
code	状态码

errmsg

错误信息

4.后端接口定义和实现,

- 如果访问记录存在，说明今天不是第一次访问，不新建记录，访问量直接累加。
- 如果访问记录不存在，说明今天是第一次访问，新建记录并保存访问量。

```
class DetailVisitView(View):
    """详情页分类商品访问量"""

    def post(self, request, category_id):
        """记录分类商品访问量"""
        try:
            category = models.GoodsCategory.objects.get(id=category_id)
        except models.GoodsCategory.DoesNotExist:
            return http.HttpResponseForbidden('缺少必传参数')

        # 获取今天的日期
        t = timezone.localtime()
        today_str = '%d-%02d-%02d' % (t.year, t.month, t.day)
        today_date = datetime.datetime.strptime(today_str, '%Y-%m-%d')
        try:
            # 查询今天该类别的商品的访问量
            # counts_data = category.goodsvisitcount_set.get(date=today_date)
            counts_data = GoodsVisitCount.objects.get(category=category, date=today_date)
        except models.GoodsVisitCount.DoesNotExist:
            # 如果该类别的商品在今天没有过访问记录，就新建一个访问记录
            counts_data = models.GoodsVisitCount()

        try:
            counts_data.category = category
            counts_data.count += 1
            counts_data.save()
        except Exception as e:
            logger.error(e)
            return http.HttpResponseServerError('服务器异常')

    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK'})
```

用户浏览记录

设计浏览记录存储方案

- 当登录用户在浏览商品的详情页时，我们就可以把详情页这件商品信息存储起来，作为该登录用户的浏览记录。
- 用户未登录，我们不记录其商品浏览记录。



1. 存储数据说明

- 虽然浏览记录界面上要展示商品的一些SKU信息，但是我们在存储时没有必要存很多SKU信息。
- 我们选择存储SKU信息的唯一编号（sku_id）来表示该件商品的浏览记录。
- 存储数据： `sku_id`

2. 存储位置说明

- 用户浏览记录是临时数据，且经常变化，数据量不大，所以我们选择内存型数据库进行存储。
- 存储位置： `Redis数据库 3号库`

```
CACHES = {
    "history": { # 用户浏览记录
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/3",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    },
}
```

3. 存储类型说明

- 由于用户浏览记录跟用户浏览商品详情的顺序有关，所以我们选择使用Redis中的 `list` 类型存储 `sku_id`
- 每个用户维护一条浏览记录，且浏览记录都是独立存储的，不能共用。所以我们需要对用户的浏览记录进行唯一标识。
- 我们可以使用登录用户的ID来唯一标识该用户的浏览记录。
- 存储类型： `'history_user_id' : [sku_id_1, sku_id_2, ...]`

```
[127.0.0.1:6379[3]> keys *
1) "history_5"
[127.0.0.1:6379[3]> LRANGE history_5 0 -1
1) "6"
2) "14"
3) "2"
4) "7"
5) "5"
```

4. 存储逻辑说明

- SKU信息不能重复。
- 最近一次浏览的商品SKU信息排在最前面，以此类推。
- 每个用户的浏览记录最多存储五个商品SKU信息。
- 存储逻辑：先去重，再存储，最后截取。

保存和查询浏览记录

1. 保存用户浏览记录

1. 请求方式

选项	方案
请求方法	POST
请求地址	/browse_histories/

2. 请求参数: JSON

参数名	类型	是否必传	说明
sku_id	string	是	商品SKU编号

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息

4. 后端接口定义和实现

```
class UserBrowseHistory(LoginRequiredMixin, View):
    """用户浏览记录"""

    def post(self, request):
        """保存用户浏览记录"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        sku_id = json_dict.get('sku_id')

        # 校验参数
        try:
            models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return http.HttpResponseForbidden('sku不存在')

        # 保存用户浏览数据
        redis_conn = get_redis_connection('history')
        pl = redis_conn.pipeline()
        user_id = request.user.id

        # 先去重
        pl.lrem('history_%s' % user_id, 0, sku_id)
        # 再存储
        pl.lpush('history_%s' % user_id, sku_id)
        pl.execute()
```

```

    pl.lpush('history_%s' % user_id, sku_id)
    # 最后截取
    pl.ltrim('history_%s' % user_id, 0, 4)
    # 执行管道
    pl.execute()

    # 响应结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK'})

```

2. 查询用户浏览记录

1. 请求方式

选项	方案
请求方法	GET
请求地址	/browse_histories/

2. 请求参数:

无

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息
skus[]	商品SKU列表数据
id	商品SKU编号
name	商品SKU名称
default_image_url	商品SKU默认图片
price	商品SKU单价

```

{
    "code": "0",
    "errmsg": "OK",
    "skus": [
        {
            "id": 6,
            "name": "Apple iPhone 8 Plus (A1864) 256GB 深空灰色 移动联通电信4G手机",
            "price": "7988.00",
            "default_image_url": "http://image.meiduo.site:8888/group1/M00/00/02/ctM
3BVrRbI2ARekNAAFZsBqChgk3141998"
        },
        .....
    ]
}

```

```
        ]
    }
```

4.后端接口定义和实现

```
class UserBrowseHistory(LoginRequiredJSONMixin, View):
    """用户浏览记录"""

    def get(self, request):
        """获取用户浏览记录"""
        # 获取Redis存储的sku_id列表信息
        redis_conn = get_redis_connection('history')
        sku_ids = redis_conn.lrange('history_%s' % request.user.id, 0, -1)

        # 根据sku_ids列表数据，查询出商品sku信息
        skus = []
        for sku_id in sku_ids:
            sku = models.SKU.objects.get(id=sku_id)
            skus.append({
                'id': sku.id,
                'name': sku.name,
                'default_image_url': sku.default_image.url,
                'price': sku.price
            })

        return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'skus': skus})
    }
```

Vue渲染用户浏览记录

```
<div class="has_view_list" v-cloak>
    <ul class="goods_type_list clearfix">
        <li v-for="sku in histories">
            <a :href="sku.url"></a>
            <h4><a :href="sku.url">[ [ sku.name ] ]</a></h4>
            <div class="operate">
                <span class="price">¥ [ [ sku.price ] ]</span>
                <span class="unit">台</span>
                <a href="javascript:;" class="add_goods" title="加入购物车"></a>
            </div>
        </li>
    </ul>
</div>
```

最近浏览



购物车

购物车存储方案

Apple iPhone 8 Plus (A1864) 64GB 金色 移动联通电信4G手机

选【移动优惠购】新机配新卡，199优质靓号，流量不限量！

¥ 6499.00

18人评价

数量: 1

颜色: 金色 深空灰 银色

内存: 64GB 256GB

总价: 6499.00元

[加入购物车](#)

全部商品 3 件					
	商品名称	商品价格	数量	小计	操作
<input checked="" type="checkbox"/>	Apple iPhone 8 Plus (A1864) 64GB 金...	6499.00元	- 1 +	6499.00元	删除
<input checked="" type="checkbox"/>	Apple iPhone 8 Plus (A1864) 256GB 深...	7999.00元	- 2 +	15998.00元	删除
<input checked="" type="checkbox"/>	全选			合计(不含运费): 22475.00	去结算

- 用户登录与未登录状态下，都可以保存购物车数据。
- 用户对购物车数据的操作包括：增、删、改、查、全选 等等
- 每个用户的购物车数据都要做唯一的标识。

1. 登录用户购物车存储方案

1. 存储数据说明

- 如何描述一条完整的购物车记录？
 - 用户 `itcast`，选择了两个 `iPhone8` 添加到了购物车中，状态为勾选
- 一条完整的购物车记录包括： 用户、商品、数量、勾选状态。
- 存储数据：`user_id`、`sku_id`、`count`、`selected`

2. 存储位置说明

- 购物车数据量小，结构简单，更新频繁，所以我们选择内存型数据库Redis进行存储。
- 存储位置：`Redis数据库 4号库`

```
"carts": { # 购物车
    "BACKEND": "django_redis.cache.RedisCache",
    "LOCATION": "redis://127.0.0.1:6379/4",
    "OPTIONS": {
        "CLIENT_CLASS": "django_redis.client.DefaultClient",
    }
},
```

3. 存储类型说明

- 提示：我们很难将 用户、商品、数量、勾选状态 存放到一条Redis记录中。所以我们要把购物车数据合理的分开存储。
- 用户、商品、数量：** hash
 - carts_user_id: {sku_id1: count, sku_id3: count, sku_id5: count, ...}
- 勾选状态：** set
 - 只将已勾选商品的sku_id存储到set中，比如，1号和3号商品是被勾选的。
 - selected_user_id: [sku_id1, sku_id3, ...]

```
[127.0.0.1:6379[4]> keys *
1) "selected_5"      5号用户的购物车数据
2) "carts_5"
[127.0.0.1:6379[4]> hgetall carts_5
1) "3"
2) "1"      3号商品1件
3) "6"      6号商品2件
4) "2"
[127.0.0.1:6379[4]> smembers selected_5
1) "3"      3号和6号商品已勾选
2) "6"
```

4. 存储逻辑说明

- 当要添加到购物车的商品已存在时，对商品数量进行累加计算。
- 当要添加到购物车的商品不存在时，向hash中新增field和value即可。

2. 未登录用户购物车存储方案

1. 存储数据说明

- 存储数据：** user_id 、 sku_id 、 count 、 selected

2. 存储位置说明

- 由于用户未登录，服务端无法拿到用户的ID，所以服务端在生成购物车记录时很难唯一标识该记录。
- 我们可以将未登录用户的购物车数据缓存到用户浏览器的 cookie 中，每个用户自己浏览器的 cookie 中存储属于自己的购物车数据。
- 存储位置：** 用户浏览器的cookie

3. 存储类型说明

- 提示：浏览器的cookie中存储的数据类型是字符串。
- 思考：如何在字符串中描述一条购物车记录？
- 结论：JSON字符串可以描述复杂结构的字符串数据，可以保证一条购物车记录不用分开存储。

```
{
  "sku_id1": {
    "count": "1",
    "selected": "True"
  },
  ...
}
```

```
"sku_id3":{  
    "count":"3",  
    "selected":"True"  
},  
"sku_id5":{  
    "count":"3",  
    "selected":"False"  
}  
}
```



4. 存储逻辑说明

- 当要添加到购物车的商品已存在时，对商品数量进行累加计算。
- 当要添加到购物车的商品不存在时，向JSON中新增field和value即可。

提示：

- 浏览器cookie中存储的是字符串明文数据。
 - 我们需要对购物车这类隐私数据进行密文存储。
- 解决方案：`pickle`模块 和 `base64`模块

5. pickle模块介绍

- `pickle`模块是Python的标准模块，提供了对Python数据的序列化操作，可以将数据转换为`bytes`类型，且序列化速度快。
- `pickle`模块使用：
 - `pickle.dumps()` 将Python数据序列化为`bytes`类型数据。
 - `pickle.loads()` 将`bytes`类型数据反序列化为python数据。

```
>>> import pickle

>>> dict = {'1': {'count': 10, 'selected': True}, '2': {'count': 20, 'selected': False}}
>>> ret = pickle.dumps(dict)
>>> ret
b'\x80\x03}q\x00(X\x01\x00\x00\x001q\x01}q\x02(X\x05\x00\x00\x00countq\x03K\nX\x08\
\x00\x00\x00selectedq\x04\x88uX\x01\x00\x00\x002q\x05}q\x06(h\x03K\x14h\x04\x89uu.'
>>> pickle.loads(ret)
{'1': {'count': 10, 'selected': True}, '2': {'count': 20, 'selected': False}}
```

6. base64模块介绍

- 提示：`pickle`模块序列化转换后的数据是`bytes`类型，浏览器cookie无法存储。
- `base64`模块是Python的标准模块，可以对`bytes`类型数据进行编码，并得到`bytes`类型的密文数据。
- `base64`模块使用：
 - `base64.b64encode()` 将`bytes`类型数据进行base64编码，返回编码后的`bytes`类型数据。
 - `base64.b64decode()` 将base64编码后的`bytes`类型数据进行解码，返回解码后的`bytes`类型数据。

```
>>> import base64
>>> ret
b'\x80\x03}q\x00(X\x01\x00\x00\x001q\x01}q\x02(X\x05\x00\x00\x00countq\x03K\nX\x08\
\x00\x00\x00selectedq\x04\x88uX\x01\x00\x00\x002q\x05}q\x06(h\x03K\x14h\x04\x89uu.'
```

```
>>> b = base64.b64encode(ret)
>>> b
b'gAN9cQoWAEAAAAxQF9cQIoWAUAAABjb3VudHEDSwpYCAAAAHN1bGVjdGVkcQSIdVgBAAAAMnEFFXEGK
GgDSxRoB1l1dS4='
>>> base64.b64decode(b)
b'\x80\x03}q\x00(X\x01\x00\x00\x001q\x01}q\x02(X\x05\x00\x00\x00countq\x03K\nX\x08\
\x00\x00\x00selectedq\x04\x88uX\x01\x00\x00\x002q\x05}q\x06(h\x03K\x14h\x04\x89uu.'
```



购物车管理

添加购物车

提示：在商品详情页添加购物车使用局部刷新的效果。

1. 添加购物车接口设计和定义

1. 请求方式

选项	方案
请求方法	POST
请求地址	/carts/

2. 请求参数：JSON

参数名	类型	是否必传	说明
sku_id	int	是	商品SKU编号
count	int	是	商品数量
selected	bool	否	是否勾选

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

4. 后端接口定义

```
class CartsView(View):
    """购物车管理"""

    def post(self, request):
        """添加购物车"""
        # 接收和校验参数
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            pass
        else:
            # 用户未登录，操作cookie购物车
            pass
```

2. 添加购物车后端逻辑实现

1.接收和校验参数

```

class CartsView(View):
    """购物车管理"""

    def post(self, request):
        """添加购物车"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        sku_id = json_dict.get('sku_id')
        count = json_dict.get('count')
        selected = json_dict.get('selected', True)

        # 判断参数是否齐全
        if not all([sku_id, count]):
            return http.HttpResponseForbidden('缺少必传参数')
        # 判断sku_id是否存在
        try:
            models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return http.HttpResponseForbidden('商品不存在')
        # 判断count是否为数字
        try:
            count = int(count)
        except Exception:
            return http.HttpResponseForbidden('参数count有误')
        # 判断selected是否为bool值
        if selected:
            if not isinstance(selected, bool):
                return http.HttpResponseForbidden('参数selected有误')

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            pass
        else:
            # 用户未登录，操作cookie购物车
            pass

```

2.添加购物车到Redis

```

class CartsView(View):
    """购物车管理"""

    def post(self, request):
        """添加购物车"""
        # 接收和校验参数
        .....

```

```

# 判断用户是否登录
user = request.user
if user.is_authenticated:
    # 用户已登录，操作redis购物车
    redis_conn = get_redis_connection('carts')
    pl = redis_conn.pipeline()
    # 新增购物车数据
    pl.hincrby('carts_%s' % user.id, sku_id, count)
    # 新增选中的状态
    if selected:
        pl.sadd('selected_%s' % user.id, sku_id)
    # 执行管道
    pl.execute()
    # 响应结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '添加购物车成功'})
)
else:
    # 用户未登录，操作cookie购物车
    pass

```

3.添加购物车到cookie

```

class CartsView(View):
    """购物车管理"""

    def post(self, request):
        """添加购物车"""
        # 接收和校验参数
        .....

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            .....

        else:
            # 用户未登录，操作cookie购物车
            cart_str = request.COOKIES.get('carts')
            # 如果用户操作过cookie购物车
            if cart_str:
                # 将cart_str转成bytes,再将bytes转成base64的bytes,最后将bytes转字典
                cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
            else: # 用户从没有操作过cookie购物车
                cart_dict = {}

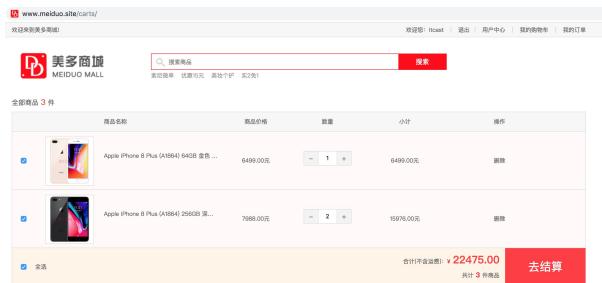
            # 判断要加入购物车的商品是否已经在购物车中,如有相同商品,累加求和,反之,直接赋值
            if sku_id in cart_dict:
                # 累加求和
                origin_count = cart_dict[sku_id]['count']

```

```
        count += origin_count
    cart_dict[sku_id] = {
        'count': count,
        'selected': selected
    }
    # 将字典转成bytes,再将bytes转成base64的bytes,最后将bytes转字符串
    cookie_cart_str = base64.b64encode(pickle.dumps(cart_dict)).decode()

    # 创建响应对象
    response = http.JsonResponse({'code': RETCODE.OK, 'errmsg': '添加购物车成功'})
    # 响应结果并将购物车数据写入到cookie
    response.set_cookie('carts', cookie_cart_str, max_age=constants.CARTS_COOKIE_EXPIRES)
    return response
```

展示购物车



1. 展示购物车接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/carts/

2. 请求参数:

无

3. 响应结果: HTML

cart.html

4. 后端接口定义

```
class CartsView(View):
    """购物车管理"""

    def get(self, request):
        """展示购物车"""
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询redis购物车
            pass
        else:
            # 用户未登录, 查询cookies购物车
            pass
```

2. 展示购物车后端逻辑实现

1. 查询Redis购物车

```

class CartsView(View):
    """购物车管理"""

    def get(self, request):
        """展示购物车"""
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询redis购物车
            redis_conn = get_redis_connection('carts')
            # 获取redis中的购物车数据
            redis_cart = redis_conn.hgetall('carts_%s' % user.id)
            # 获取redis中的选中状态
            cart_selected = redis_conn.smembers('selected_%s' % user.id)

            # 将redis中的数据构造成跟cookie中的格式一致, 方便统一查询
            cart_dict = {}
            for sku_id, count in redis_cart.items():
                cart_dict[int(sku_id)] = {
                    'count': int(count),
                    'selected': sku_id in cart_selected
                }
        else:
            # 用户未登录, 查询cookies购物车
            pass

```

2.查询cookie购物车

```

class CartsView(View):
    """购物车管理"""

    def get(self, request):
        """展示购物车"""
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询redis购物车
            .....
        else:
            # 用户未登录, 查询cookies购物车
            cart_str = request.COOKIES.get('carts')
            if cart_str:
                # 将cart_str转成bytes, 再将bytes转成base64的bytes, 最后将bytes转字典
                cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
            else:
                cart_dict = {}

```

3.查询购物车SKU信息

```

class CartsView(View):

```

```

"""购物车管理"""

def get(self, request):
    """展示购物车"""
    user = request.user
    if user.is_authenticated:
        # 用户已登录, 查询redis购物车
        .....
    else:
        # 用户未登录, 查询cookies购物车
        .....

    # 构造购物车渲染数据
    sku_ids = cart_dict.keys()
    skus = models.SKU.objects.filter(id__in=sku_ids)
    cart_skus = []
    for sku in skus:
        cart_skus.append({
            'id': sku.id,
            'count': cart_dict.get(sku.id).get('count'),
            'selected': str(cart_dict.get(sku.id).get('selected')), # 将True, 转'True', 方便json解析
            'name': sku.name,
            'default_image_url': sku.default_image.url,
            'price': str(sku.price), # 从Decimal('10.2')中取出'10.2', 方便json解析
            'amount': str(sku.price * cart_dict.get(sku.id).get('count'))
        })

    context = {
        'cart_skus': cart_skus
    }

    # 渲染购物车页面
    return render(request, 'cart.html', context)

```

4. 渲染购物车信息

```

<div class="total_count">全部商品<em>[[ total_count ]]</em>件</div>
<ul class="cart_list_th clearfix">
    <li class="col01">商品名称</li>
    <li class="col03">商品价格</li>
    <li class="col04">数量</li>
    <li class="col05">小计</li>
    <li class="col06">操作</li>
</ul>
<ul class="cart_list_td clearfix" v-for="(cart_sku, index) in carts" v-cloak>
    <li class="col01"><input type="checkbox" name="" v-model="cart_sku.selected" @change="update_selected(index)"></li>
    <li class="col02"></li>
    <li class="col03">[[ cart_sku.name ]]</li>

```

```
<li class="col05">[[ cart_sku.price ]]元</li>
<li class="col06">
    <div class="num_add">
        <a @click="on_minus(index)" class="minus f1">-</a>
        <input v-model="cart_sku.count" @blur="on_input(index)" type="text" class="num_show f1">
        <a @click="on_add(index)" class="add f1">+</a>
    </div>
</li>
<li class="col07">[[ cart_sku.amount ]]元</li>
<li class="col08"><a @click="on_delete(index)">删除</a></li>
</ul>
<ul class="settlements" v-cloak>
    <li class="col01"><input type="checkbox" name="" @change="on_selected_all" v-model="selected_all"></li>
    <li class="col02">全选</li>
    <li class="col03">合计(不含运费): <span>¥</span><em>[[ total_selected_amount ]]</em><br>共计<b>[[ total_selected_count ]]</b>件商品</li>
    <li class="col04"><a href="place_order.html">去结算</a></li>
</ul>
```

修改购物车

提示：在购物车页面修改购物车使用局部刷新的效果。

1. 修改购物车接口设计和定义

1.请求方式

选项	方案
请求方法	PUT
请求地址	/carts/

2.请求参数: JSON

参数名	类型	是否必传	说明
sku_id	int	是	商品SKU编号
count	int	是	商品数量
selected	bool	否	是否勾选

3.响应结果: JSON

字段	说明
sku_id	商品SKU编号
count	商品数量
selected	是否勾选

4.后端接口定义

```
class CartsView(View):
    """购物车管理"""

    def put(self, request):
        """修改购物车"""
        # 接收和校验参数
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，修改redis购物车
            pass
        else:
            # 用户未登录，修改cookie购物车
            pass
```

2. 修改购物车后端逻辑实现

1. 接收和校验参数

```

class CartsView(View):
    """购物车管理"""

    def put(self, request):
        """修改购物车"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        sku_id = json_dict.get('sku_id')
        count = json_dict.get('count')
        selected = json_dict.get('selected', True)

        # 判断参数是否齐全
        if not all([sku_id, count]):
            return http.HttpResponseForbidden('缺少必传参数')
        # 判断sku_id是否存在
        try:
            sku = models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return http.HttpResponseForbidden('商品sku_id不存在')
        # 判断count是否为数字
        try:
            count = int(count)
        except Exception:
            return http.HttpResponseForbidden('参数count有误')
        # 判断selected是否为bool值
        if selected:
            if not isinstance(selected, bool):
                return http.HttpResponseForbidden('参数selected有误')

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，修改redis购物车
            pass
        else:
            # 用户未登录，修改cookie购物车
            pass

```

2. 修改Redis购物车

```

class CartsView(View):
    """购物车管理"""

    def put(self, request):
        """修改购物车"""

```

```

# 接收和校验参数
.....
# 判断用户是否登录
user = request.user
if user.is_authenticated:
    # 用户已登录，修改redis购物车
    redis_conn = get_redis_connection('carts')
    pl = redis_conn.pipeline()
    # 因为接口设计为幂等的，直接覆盖
    pl.hset('carts_%s' % user.id, sku_id, count)
    # 是否选中
    if selected:
        pl.sadd('selected_%s' % user.id, sku_id)
    else:
        pl.srem('selected_%s' % user.id, sku_id)
    pl.execute()

# 创建响应对象
cart_sku = {
    'id':sku_id,
    'count':count,
    'selected':selected,
    'name': sku.name,
    'price': sku.price,
    'amount': sku.price * count,
    'default_image_url': sku.default_image.url
}
return http.JsonResponse({'code':RETCODE.OK, 'errmsg':'修改购物车成功',
    'cart_sku':cart_sku})
else:
    # 用户未登录，修改cookie购物车
    pass

```

3.修改cookie购物车

```

class CartsView(View):
    """购物车管理"""

    def put(self, request):
        """修改购物车"""
        # 接收和校验参数
        .....
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，修改redis购物车
            .....
        else:

```

```
# 用户未登录，修改cookie购物车
cart_str = request.COOKIES.get('carts')
if cart_str:
    # 将cart_str转成bytes,再将bytes转成base64的bytes,最后将bytes转字典
    cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
else:
    cart_dict = {}
# 因为接口设计为幂等的，直接覆盖
cart_dict[sku_id] = {
    'count': count,
    'selected': selected
}
# 将字典转成bytes,再将bytes转成base64的bytes,最后将bytes转字符串
cookie_cart_str = base64.b64encode(pickle.dumps(cart_dict)).decode()

# 创建响应对象
cart_sku = {
    'id': sku_id,
    'count': count,
    'selected': selected,
    'name': sku.name,
    'price': sku.price,
    'amount': sku.price * count,
    'default_image_url': sku.default_image.url
}
response = http.JsonResponse({'code': RETCODE.OK, 'errmsg': '修改购物车成功',
    'cart_sku': cart_sku})
# 响应结果并将购物车数据写入到cookie
response.set_cookie('carts', cookie_cart_str, max_age=constants.CARTS_COOKIE_EXPIRES)
return response
```



删除购物车

提示：在购物车页面删除购物车使用局部刷新的效果。

1. 删除购物车接口设计和定义

1. 请求方式

选项	方案
请求方法	DELETE
请求地址	/carts/

2. 请求参数：JSON

参数名	类型	是否必传	说明
sku_id	int	是	商品SKU编号

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

4. 后端接口定义

```
class CartsView(View):
    """购物车管理"""

    def delete(self, request):
        """删除购物车"""
        # 接收和校验参数
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，删除redis购物车
            pass
        else:
            # 用户未登录，删除cookie购物车
            pass
```

2. 删除购物车后端逻辑实现

1. 接收和校验参数

```

class CartsView(View):
    """购物车管理"""

    def delete(self, request):
        """删除购物车"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        sku_id = json_dict.get('sku_id')

        # 判断sku_id是否存在
        try:
            models.SKU.objects.get(id=sku_id)
        except models.SKU.DoesNotExist:
            return http.HttpResponseForbidden('商品不存在')

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 删除redis购物车
            pass
        else:
            # 用户未登录, 删除cookie购物车
            pass

```

2.删除Redis购物车

```

class CartsView(View):
    """购物车管理"""

    def delete(self, request):
        """删除购物车"""
        # 接收和校验参数
        .....

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 删除redis购物车
            redis_conn = get_redis_connection('carts')
            pl = redis_conn.pipeline()
            # 删除键, 就等价于删除了整条记录
            pl.hdel('carts_%s' % user.id, sku_id)
            pl.srem('selected_%s' % user.id, sku_id)
            pl.execute()

            # 删除结束后, 没有响应的数据, 只需要响应状态码即可
            return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '删除成功'})
        )
        else:

```

```
# 用户未登录，删除cookie购物车
pass
```

3.删除cookie购物车

```
class CartsView(View):
    """购物车管理"""

    def delete(self, request):
        """删除购物车"""
        # 接收和校验参数
        .....

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，删除redis购物车
            .....
        else:
            # 用户未登录，删除cookie购物车
            cart_str = request.COOKIES.get('carts')
            if cart_str:
                # 将cart_str转成bytes,再将bytes转成base64的bytes,最后将bytes转字典
                cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
            else:
                cart_dict = {}

            # 创建响应对象
            response = http.JsonResponse({'code': RETCODE.OK, 'errmsg': '删除成功'})
            if sku_id in cart_dict:
                del cart_dict[sku_id]
                # 将字典转成bytes,再将bytes转成base64的bytes,最后将bytes转字符串
                cookie_cart_str = base64.b64encode(pickle.dumps(cart_dict)).decode()
            )
            # 响应结果并将购物车数据写入到cookie
            response.set_cookie('carts', cookie_cart_str, max_age=constants.CART_COOKIE_EXPIRES)
        return response
```

全选购物车

提示：在购物车页面全选购物车使用局部刷新的效果。

1. 全选购物车接口设计和定义

1.请求方式

选项	方案
请求方法	PUT
请求地址	/carts/selection/

2.请求参数: JSON

参数名	类型	是否必传	说明
selected	bool	是	是否全选

3.响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息

4.后端接口定义

```
class CartsSelectAllView(View):
    """全选购物车"""

    def put(self, request):
        # 接收和校验参数
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            pass
        else:
            # 用户未登录，操作cookie购物车
            pass
```

2. 全选购物车后端逻辑实现

1.接收和校验参数

```
class CartsSelectAllView(View):
```

```
"""全选购物车"""

def put(self, request):
    # 接收参数
    json_dict = json.loads(request.body.decode())
    selected = json_dict.get('selected', True)

    # 校验参数
    if selected:
        if not isinstance(selected, bool):
            return http.HttpResponseForbidden('参数selected有误')

    # 判断用户是否登录
    user = request.user
    if user.is_authenticated:
        # 用户已登录，操作redis购物车
        pass
    else:
        # 用户未登录，操作cookie购物车
        pass
```

2.全选Redis购物车

```
class CartsSelectAllView(View):
    """全选购物车"""

    def put(self, request):
        # 接收和校验参数
        .....

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            redis_conn = get_redis_connection('carts')
            redis_cart = redis_conn.hgetall('carts_%s' % user.id)
            cart_sku_ids = redis_cart.keys()
            if selected:
                # 全选
                redis_conn.sadd('selected_%s' % user.id, *cart_sku_ids)
            else:
                # 取消全选
                redis_conn.srem('selected_%s' % user.id, *cart_sku_ids)
            return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '全选购物车成功'})
        )
        else:
            # 用户未登录，操作cookie购物车
            pass
```

3.全选cookie购物车

```
class CartsSelectAllView(View):
    """全选购物车"""

    def put(self, request):
        # 接收和校验参数
        .....

        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录，操作redis购物车
            .....

        else:
            # 用户未登录，操作cookie购物车
            cart_str = request.COOKIES.get('carts')
            response = http.JsonResponse({'code': RETCODE.OK, 'errmsg': '全选购物车成功'})
            if cart_str:
                cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
                for sku_id in cart_dict:
                    cart_dict[sku_id]['selected'] = selected
                cookie_cart_str = base64.b64encode(pickle.dumps(cart_dict)).decode()
            )
            response.set_cookie('carts', cookie_cart_str, max_age=constants.CARTS_COOKIE_EXPIRES)

    return response
```

合并购物车

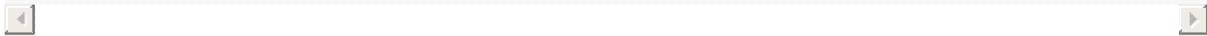
需求：用户登录时，将 cookie 购物车数据 合并 到 Redis 购物车数据中。

提示：

- QQ登录 和 账号登录 时都要进行购物车合并操作。

1. 合并购物车逻辑分析

1. 合并方向：cookie购物车数据合并到Redis购物车数据中。
2. 合并数据：购物车商品数据和勾选状态。
3. 合并方案：
 - 3.1 Redis数据库中的购物车数据保留。
 - 3.2 如果cookie中的购物车数据在Redis数据库中已存在，将cookie购物车数据覆盖Redis购物车数据。
 - 3.3 如果cookie中的购物车数据在Redis数据库中不存在，将cookie购物车数据新增到Redis。
 - 3.4 最终购物车的勾选状态以cookie购物车勾选状态为准。



2. 合并购物车逻辑实现

新建文件： carts.utils.py

```
def merge_cart_cookie_to_redis(request, user, response):
    """
    登录后合并cookie购物车数据到Redis
    :param request: 本次请求对象，获取cookie中的数据
    :param response: 本次响应对象，清除cookie中的数据
    :param user: 登录用户信息，获取user_id
    :return: response
    """

    # 获取cookie中的购物车数据
    cookie_cart_str = request.COOKIES.get('carts')
    # cookie中没有数据就响应结果
    if not cookie_cart_str:
        return response
    cookie_cart_dict = pickle.loads(base64.b64decode(cookie_cart_str.encode()))

    new_cart_dict = {}
    new_cart_selected_add = []
    new_cart_selected_remove = []
    # 同步cookie中购物车数据
    for sku_id, cookie_dict in cookie_cart_dict.items():
        new_cart_dict[sku_id] = cookie_dict['count']

        if cookie_dict['selected']:
            new_cart_selected_add.append(sku_id)
```

```
else:  
    new_cart_selected_remove.append(sku_id)  
  
    # 将new_cart_dict写入到Redis数据库  
    redis_conn = get_redis_connection('carts')  
    pl = redis_conn.pipeline()  
    pl.hmset('carts_%s' % user.id, new_cart_dict)  
    # 将勾选状态同步到Redis数据库  
    if new_cart_selected_add:  
        pl.sadd('selected_%s' % user.id, *new_cart_selected_add)  
    if new_cart_selected_remove:  
        pl.srem('selected_%s' % user.id, *new_cart_selected_remove)  
    pl.execute()  
  
    # 清除cookie  
    response.delete_cookie('carts')  
  
return response
```

3. 账号和QQ登录合并购物车

在 `users.views.py` 和 `oauth.views.py` 文件中调用合并购物车的工具方法

```
# 合并购物车  
response = merge_cart_cookie_to_redis(request=request, user=user, response=response  
)
```

展示商品页面简单购物车



需求：用户鼠标悬停在商品页面右上角购物车标签上，以下拉框形式展示当前购物车数据。

1. 简单购物车数据接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/carts/simple/

2. 请求参数：

无

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息
cart_skus[]	简单购物车SKU列表
id	购物车SKU编号
name	购物车SKU名称
count	购物车SKU数量
default_image_url	购物车SKU图片

```
{
    "code": "0",
    "errmsg": "OK",
    "cart_skus": [
        {
            "id": 1,
            "name": "Apple MacBook Pro 13.3英寸笔记本 银色",
            "count": 1,
            "default_image_url": "http://image.meiduo.site:8888/group1/M00/00/02/ctM3BVrPB4GAwkJT1AAGuN6wB9fU4220429"
        },
    ]
}
```

```
    .....
]
}
```

4.后端接口定义

```
class CartsSimpleView(View):
    """商品页面右上角购物车"""

    def get(self, request):
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询Redis购物车
            pass
        else:
            # 用户未登录, 查询cookie购物车
            pass

        # 构造简单购物车JSON数据
        pass
```

2. 简单购物车数据后端逻辑实现

1.查询Redis购物车

```
class CartsSimpleView(View):
    """商品页面右上角购物车"""

    def get(self, request):
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询Redis购物车
            redis_conn = get_redis_connection('carts')
            redis_cart = redis_conn.hgetall('carts_%s' % user.id)
            cart_selected = redis_conn.smembers('selected_%s' % user.id)
            # 将redis中的两个数据统一格式, 跟cookie中的格式一致, 方便统一查询
            cart_dict = {}
            for sku_id, count in redis_cart.items():
                cart_dict[int(sku_id)] = {
                    'count': int(count),
                    'selected': sku_id in cart_selected
                }
        else:
            # 用户未登录, 查询cookie购物车
            pass

        # 构造简单购物车JSON数据
```

```
pass
```

2.查询Redis购物车

```
class CartsSimpleView(View):
    """商品页面右上角购物车"""

    def get(self, request):
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询Redis购物车
            .....
        else:
            # 用户未登录, 查询cookie购物车
            cart_str = request.COOKIES.get('carts')
            if cart_str:
                cart_dict = pickle.loads(base64.b64decode(cart_str.encode()))
            else:
                cart_dict = {}


```

3.构造简单购物车JSON数据

```
class CartsSimpleView(View):
    """商品页面右上角购物车"""

    def get(self, request):
        # 判断用户是否登录
        user = request.user
        if user.is_authenticated:
            # 用户已登录, 查询Redis购物车
            .....
        else:
            # 用户未登录, 查询cookie购物车
            .....

        # 构造简单购物车 JSON数据
        cart_skus = []
        sku_ids = cart_dict.keys()
        skus = models.SKU.objects.filter(id__in=sku_ids)
        for sku in skus:
            cart_skus.append({
                'id': sku.id,
                'name': sku.name,
                'count': cart_dict.get(sku.id).get('count'),
                'default_image_url': sku.default_image.url
            })

        # 响应json列表数据

```

```
        return http.JsonResponse({'code':RETCODE.OK, 'errmsg':'OK', 'cart_skus':cart_skus})
```

3. 展示商品页面简单购物车

1.商品页面发送Ajax请求

- index.js 、 list.js 、 detail.js

```
get_carts(){
    let url = '/carts/simple/';
    axios.get(url, {
        responseType: 'json',
    })
    .then(response => {
        this.carts = response.data.cart_skus;
        this.cart_total_count = 0;
        for(let i=0;i<this.carts.length;i++){
            if (this.carts[i].name.length>25){
                this.carts[i].name = this.carts[i].name.substring(0, 25) + '...'
            }
            this.cart_total_count += this.carts[i].count;
        }
    })
    .catch(error => {
        console.log(error.response);
    })
},

```

2.商品页面渲染简单购物车数据

- index.html 、 list.html 、 detail.html

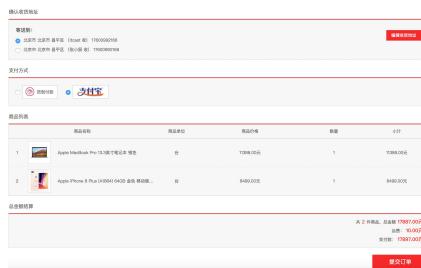
```
<div @mouseenter="get_carts" class="guest_cart fr" v-cloak>
    <a href="{{ url('carts:info') }}" class="cart_name fl">我的购物车</a>
    <div class="goods_count fl" id="show_count">[[ cart_total_count ]]</div>
    <ul class="cart_goods_show">
        <li v-for="sku in carts">
            
            <h4>[[ sku.name ]]</h4>
            <div>[[ sku.count ]]</div>
        </li>
    </ul>
</div>
```


订单

提示：

- 订单入口 在《购物车》页面的《去结算》。
- 《去结算》后进入到《结算订单》页面，展示出要结算的商品信息。

结算订单



1. 结算订单逻辑分析

结算订单是从Redis购物车中查询出被勾选的商品信息进行结算并展示。

2. 结算订单接口设计和定义

1. 请求方式

选项	方案
请求方法	GET
请求地址	/orders/settlement/

2. 请求参数:

无

3. 响应结果: HTML

place_order.html

4. 后端接口定义

```
class OrderSettlementView(LoginRequiredMixin, View):
    """结算订单"""

    def get(self, request):
        """提供订单结算页面"""
        return render(request, 'place_order.html')
```

3. 结算订单后端逻辑实现

```
class OrderSettlementView(LoginRequiredMixin, View):
    """结算订单"""

    def get(self, request):
        """提供订单结算页面"""
        # 获取登录用户
```

```

user = request.user
# 查询地址信息
try:
    addresses = Address.objects.filter(user=user, is_deleted=False)
except Address.DoesNotExist:
    # 如果地址为空, 渲染模板时会判断, 并跳转到地址编辑页面
    addresses = None

# 从Redis购物车中查询出被勾选的商品信息
redis_conn = get_redis_connection('carts')
redis_cart = redis_conn.hgetall('carts_%s' % user.id)
cart_selected = redis_conn.smembers('selected_%s' % user.id)
cart = {}
for sku_id in cart_selected:
    cart[int(sku_id)] = int(redis_cart[sku_id])

# 准备初始值
total_count = 0
total_amount = Decimal('0.00')
# 查询商品信息
skus = SKU.objects.filter(id__in=cart.keys())
for sku in skus:
    sku.count = cart[sku.id]
    sku.amount = sku.count * sku.price
    # 计算总数量和总金额
    total_count += sku.count
    total_amount += sku.amount
# 补充运费
freight = Decimal('10.00')

# 渲染界面
context = {
    'addresses': addresses,
    'skus': skus,
    'total_count': total_count,
    'total_amount': total_amount,
    'freight': freight,
    'payment_amount': total_amount + freight
}

return render(request, 'place_order.html', context)

```

4. 结算订单页面渲染

```

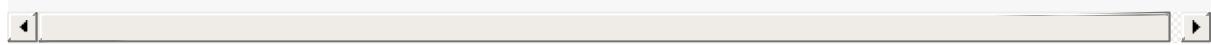
<h3 class="common_title">确认收货地址</h3>
<div class="common_list_con clearfix" id="get_site">
    <dl>
        {% if addresses %}
            <dt>寄送到: </dt>

```

```

        {% for address in addresses %}
            <dd @click="nowsite={{ address.id }}"><input type="radio" v-model="nowsite"
value="{{ address.id }}"{{ address.province }} {{ address.city }} {{ address.district }} ({{ address.receiver }} 收) {{ address.mobile }}</dd>
        {% endfor %}
        {% endif %}
    </dl>
    <a href="{{ url('users:address') }}" class="edit_site">编辑收货地址</a>
</div>
<h3 class="common_title">支付方式</h3>
<div class="common_list_con clearfix">
    <div class="pay_style_con clearfix">
        <input type="radio" name="pay_method" value="1" v-model="pay_method">
        <label class="cash">货到付款</label>
        <input type="radio" name="pay_method" value="2" v-model="pay_method">
        <label class="zhifubao"></label>
    </div>
</div>
<h3 class="common_title">商品列表</h3>
<div class="common_list_con clearfix">
    <ul class="goods_list_th clearfix">
        <li class="col01">商品名称</li>
        <li class="col02">商品单位</li>
        <li class="col03">商品价格</li>
        <li class="col04">数量</li>
        <li class="col05">小计</li>
    </ul>
    {% for sku in skus %}
        <ul class="goods_list_td clearfix">
            <li class="col01">{{loop.index}}</li>
            <li class="col02">{{ sku.name }}</li>
            <li class="col04">台</li>
            <li class="col05">{{ sku.price }}元</li>
            <li class="col06">{{ sku.count }}</li>
            <li class="col07">{{ sku.amount }}元</li>
        </ul>
    {% endfor %}
</div>
<h3 class="common_title">总金额结算</h3>
<div class="common_list_con clearfix">
    <div class="settle_con">
        <div class="total_goods_count">共<em>{{ total_count }}</em>件商品，总金额<b>{{ total_amount }}元</b></div>
        <div class="transit">运费: <b>{{ freight }}元</b></div>
        <div class="total_pay">实付款: <b>{{ payment_amount }}元</b></div>
    </div>
</div>
<div class="order_submit clearfix">
    <a @click="on_order_submit" id="order_btn">提交订单</a>
</div>

```



提交订单

提示：

- 确认了要结算的商品信息后，就可以去提交订单了。

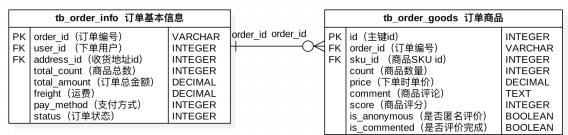
创建订单数据库表

生成的订单数据要做持久化处理，而且需要在《我的订单》页面展示出来。

1. 订单数据库表分析

注意：

- 订单号不再采用数据库自增主键，而是由后端生成。
- 一个订单中可以有多个商品信息，订单基本信息和订单商品信息是一对多的关系。



2. 订单模型类迁移建表

```

class OrderInfo(BaseModel):
    """订单信息"""
    PAY_METHODS_ENUM = {
        "CASH": 1,
        "ALIPAY": 2
    }
    PAY_METHOD_CHOICES = (
        (1, "货到付款"),
        (2, "支付宝"),
    )
    ORDER_STATUS_ENUM = {
        "UNPAID": 1,
        "UNSEND": 2,
        "UNRECEIVED": 3,
        "UNCOMMENT": 4,
        "FINISHED": 5
    }
    ORDER_STATUS_CHOICES = (
        (1, "待支付"),
        (2, "待发货"),
        (3, "待收货"),
        (4, "待评价"),
        (5, "已完成"),
        (6, "已取消"),
    )
    order_id = models.CharField(max_length=64, primary_key=True, verbose_name="订单号")
    user = models.ForeignKey(User, on_delete=models.PROTECT, verbose_name="下单用户")

```

```

address = models.ForeignKey(Address, on_delete=models.PROTECT, verbose_name="收货地址")
total_count = models.IntegerField(default=1, verbose_name="商品总数")
total_amount = models.DecimalField(max_digits=10, decimal_places=2, verbose_name="商品总金额")
freight = models.DecimalField(max_digits=10, decimal_places=2, verbose_name="运费")
pay_method = models.SmallIntegerField(choices=PAY_METHOD_CHOICES, default=1, verbose_name="支付方式")
status = models.SmallIntegerField(choices=ORDER_STATUS_CHOICES, default=1, verbose_name="订单状态")

class Meta:
    db_table = "tb_order_info"
    verbose_name = '订单基本信息'
    verbose_name_plural = verbose_name

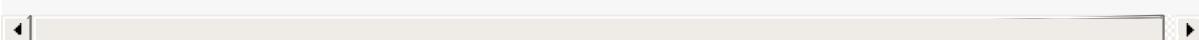
    def __str__(self):
        return self.order_id

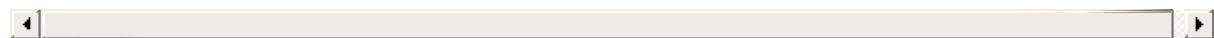

class OrderGoods(BaseModel):
    """订单商品"""
    SCORE_CHOICES = (
        (0, '0分'),
        (1, '20分'),
        (2, '40分'),
        (3, '60分'),
        (4, '80分'),
        (5, '100分'),
    )
    order = models.ForeignKey(OrderInfo, related_name='skus', on_delete=models.CASCADE, verbose_name="订单")
    sku = models.ForeignKey(SKU, on_delete=models.PROTECT, verbose_name="订单商品")
    count = models.IntegerField(default=1, verbose_name="数量")
    price = models.DecimalField(max_digits=10, decimal_places=2, verbose_name="单价")
    comment = models.TextField(default="", verbose_name="评价信息")
    score = models.SmallIntegerField(choices=SCORE_CHOICES, default=5, verbose_name='满意度评分')
    is_anonymous = models.BooleanField(default=False, verbose_name='是否匿名评价')
    is_commented = models.BooleanField(default=False, verbose_name='是否评价了')

    class Meta:
        db_table = "tb_order_goods"
        verbose_name = '订单商品'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.sku.name

```





保存订单基本信息和订单商品信息

1. 提交订单接口设计和定义

1. 请求方式

选项	方案
请求方法	POST
请求地址	/orders/commit/

2. 请求参数: JSON

参数名	类型	是否必传	说明
address_id	int	是	用户地址编号
pay_method	int	是	用户支付方式

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息
order_id	订单编号

4. 后端接口定义

```
class OrderCommitView(LoginRequiredMixin, View):
    """订单提交"""

    def post(self, request):
        """保存订单信息和订单商品信息"""
        pass
```

提示:

- 订单数据分为订单基本信息和订单商品信息，二者为一对多的关系。
- 保存到订单的数据是从Redis购物车中的已勾选的商品信息。

2. 保存订单基本信息

```
class OrderCommitView(LoginRequiredMixin, View):
    """提交订单"""

    def post(self, request):
```

```

"""保存订单信息和订单商品信息"""
# 接收参数
json_dict = json.loads(request.body.decode())
address_id = json_dict.get('address_id')
pay_method = json_dict.get('pay_method')
# 校验参数
if not all([address_id, pay_method]):
    return http.HttpResponseForbidden('缺少必传参数')
# 判断address_id是否合法
try:
    address = Address.objects.get(id=address_id)
except Address.DoesNotExist:
    return http.HttpResponseForbidden('参数address_id错误')
# 判断pay_method是否合法
if pay_method not in [OrderInfo.PAY_METHODS_ENUM['CASH'], OrderInfo.PAY_METHODS_ENUM['ALIPAY']]:
    return http.HttpResponseForbidden('参数pay_method错误')

# 获取登录用户
user = request.user
# 生成订单编号: 年月日时分秒+用户编号
order_id = timezone.localtime().strftime('%Y%m%d%H%M%S') + ('%09d' % user.id)
# 保存订单基本信息 OrderInfo (—)
order = OrderInfo.objects.create(
    order_id=order_id,
    user=user,
    address=address,
    total_count=0,
    total_amount=Decimal('0'),
    freight=Decimal('10.00'),
    pay_method=pay_method,
    status=OrderInfo.ORDER_STATUS_ENUM['UNPAID'] if pay_method == OrderInfo.PAY_METHODS_ENUM['ALIPAY'] else OrderInfo.ORDER_STATUS_ENUM['UNSEND'],
)
pass

```

3. 保存订单商品信息

```

class OrderCommitView(LoginRequiredJSONMixin, View):
    """提交订单"""

    def post(self, request):
        """保存订单信息和订单商品信息"""
        # 获取当前保存订单时需要的信息
        .....
        # 保存订单基本信息 OrderInfo (—)

```

```

.....
# 从redis读取购物车中被勾选的商品信息
redis_conn = get_redis_connection('carts')
redis_cart = redis_conn.hgetall('carts_%s' % user.id)
selected = redis_conn.smembers('selected_%s' % user.id)
carts = {}
for sku_id in selected:
    carts[int(sku_id)] = int(redis_cart[sku_id])
sku_ids = carts.keys()

# 遍历购物车中被勾选的商品信息
for sku_id in sku_ids:
    # 查询SKU信息
    sku = SKU.objects.get(id=sku_id)
    # 判断SKU库存
    sku_count = carts[sku.id]
    if sku_count > sku.stock:
        return JsonResponse({'code': RETCODE.STOCKERR, 'errmsg': '库存不足'})
    # SKU减少库存，增加销量
    sku.stock -= sku_count
    sku.sales += sku_count
    sku.save()

    # 修改SPU销量
    sku.goods.sales += sku_count
    sku.goods.save()

    # 保存订单商品信息 OrderGoods (多)
    OrderGoods.objects.create(
        order=order,
        sku=sku,
        count=sku_count,
        price=sku.price,
    )

    # 保存商品订单中总价和总数量
    order.total_count += sku_count
    order.total_amount += (sku_count * sku.price)

    # 添加邮费和保存订单信息
    order.total_amount += order.freight
    order.save()

    # 清除购物车中已结算的商品
    pl = redis_conn.pipeline()
    pl.hdel('carts_%s' % user.id, *selected)
    pl.srem('selected_%s' % user.id, *selected)
    pl.execute()

```

```
# 响应提交订单结果
return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '下单成功', 'order_id': order.order_id})
```

展示提交订单成功页面

支付方式：货到付款



支付方式：支付宝



1. 请求方式

选项	方案
请求方法	GET
请求地址	/orders/success/

2. 请求参数：

无

3. 响应结果：HTML

order_success.html

4. 后端接口定义和实现

```
class OrderSuccessView(LoginRequiredMixin, View):
    """提交订单成功"""

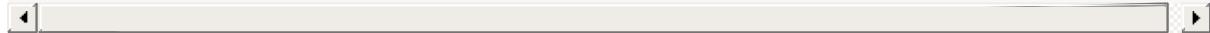
    def get(self, request):
        order_id = request.GET.get('order_id')
        payment_amount = request.GET.get('payment_amount')
        pay_method = request.GET.get('pay_method')

        context = {
            'order_id': order_id,
            'payment_amount': payment_amount,
            'pay_method': pay_method
        }
        return render(request, 'order_success.html', context)
```

5.渲染提交订单成功页面信息

```
<div class="common_list_con clearfix">
    <div class="order_success">
        <p><b>订单提交成功，订单总价<em>¥{{ payment_amount }}</em></b></p>
        <p>您的订单已成功生成，选择您想要的支付方式，订单号: {{ order_id }}</p>
        <p><a href="user_center_order.html">您可以在【用户中心】->【我的订单】查看该订单</a>
    </p>
    </div>
</div>

<div class="order_submit clearfix">
    {% if pay_method == '1' %}
        <a href="{{ url('contents:index') }}">继续购物</a>
    {% else %}
        <a @click="order_payment" class="payment">去支付</a>
    {% endif %}
</div>
```



使用事务保存订单数据

重要提示：

- 在保存订单数据时，涉及到多张表（OrderInfo、OrderGoods、SKU、SPU）的数据修改，对这些数据的修改应该是一个整体事务，即要么一起成功，要么一起失败。
- Django中对于数据库的事务，默认每执行一句数据库操作，便会自动提交。所以我们需要在保存订单中自己控制数据库事务的执行流程。

1. Django中事务的使用

1.Django中事务的使用方案

- 在Django中可以通过 `django.db.transaction` 模块 提供的 `atomic` 来定义一个事务。
- `atomic` 提供两种方案实现事务：

- 装饰器用法：

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # 这些代码会在一个事务中执行
    ....
```

- with语句用法：

```
from django.db import transaction

def viewfunc(request):
    # 这部分代码不在事务中，会被Django自动提交
    .....

    with transaction.atomic():
        # 这部分代码会在事务中执行
        .....
```

2.事务方案的选择：

- 装饰器用法：**整个视图中所有MySQL数据库的操作都看做一个事务，范围太大，不够灵活。而且无法直接作用于类视图。
- with语句用法：**可以灵活的有选择性的把某些MySQL数据库的操作看做一个事务。而且不用关心视图的类型。
- 综合考虑后我们选择 **with语句实现事务**

3.事务中的保存点：

- 在Django中，还提供了保存点的支持，可以在事务中创建保存点来记录数据的特定状态，数据库出现错误时，可以回滚到数据保存点的状态。

```
from django.db import transaction

# 创建保存点
save_id = transaction.savepoint()
# 回滚到保存点
transaction.savepoint_rollback(save_id)
# 提交从保存点到当前状态的所有数据库事务操作
transaction.savepoint_commit(save_id)
```

2. 使用事务保存订单数据

```
class OrderCommitView(LoginRequiredMixin, View):
    """订单提交"""

    def post(self, request):
        """保存订单信息和订单商品信息"""
        # 获取当前保存订单时需要的信息
        .....

        # 显式的开启一个事务
        with transaction.atomic():
            # 创建事务保存点
            save_id = transaction.savepoint()

            # 暴力回滚
            try:
                # 保存订单基本信息 OrderInfo (—)
                order = OrderInfo.objects.create(
                    order_id=order_id,
                    user=user,
                    address=address,
                    total_count=0,
                    total_amount=Decimal('0'),
                    freight=Decimal('10.00'),
                    pay_method=pay_method,
                    status=OrderInfo.ORDER_STATUS_ENUM['UNPAID'] if pay_method == 0
                else
                    OrderInfo.ORDER_STATUS_ENUM['UNSEND']
                )
            except Exception as e:
                # 从redis读取购物车中被勾选的商品信息
                redis_conn = get_redis_connection('carts')
                redis_cart = redis_conn.hgetall(f'carts_{user.id}')
                selected = redis_conn.smembers(f'selected_{user.id}')
                carts = []
                for sku_id in selected:
                    carts.append(...)
```

```

        carts[int(sku_id)] = int(redis_cart[sku_id])
        sku_ids = carts.keys()

        # 遍历购物车中被勾选的商品信息
        for sku_id in sku_ids:
            # 查询SKU信息
            sku = SKU.objects.get(id=sku_id)
            # 判断SKU库存
            sku_count = carts[sku.id]
            if sku_count > sku.stock:
                # 出错就回滚
                transaction.savepoint_rollback(save_id)
                return http.JsonResponse({'code': RETCODE.STOCKERR, 'errmsg': '库存不足'})

        # SKU减少库存，增加销量
        sku.stock -= sku_count
        sku.sales += sku_count
        sku.save()

        # 修改SPU销量
        sku.spu.sales += sku_count
        sku.spu.save()

        # 保存订单商品信息 OrderGoods (多)
        OrderGoods.objects.create(
            order=order,
            sku=sku,
            count=sku_count,
            price=sku.price,
        )

        # 保存商品订单中总价和总数量
        order.total_count += sku_count
        order.total_amount += (sku_count * sku.price)

        # 添加邮费和保存订单信息
        order.total_amount += order.freight
        order.save()
    except Exception as e:
        logger.error(e)
        transaction.savepoint_rollback(save_id)
        return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '下单失败'})
    }

    # 提交订单成功，显式的提交一次事务
    transaction.savepoint_commit(save_id)

    # 清除购物车中已结算的商品
    pl = redis_conn.pipeline()
    pl.hdel('carts_%s' % user.id, *selected)

```

```
    pl.srem('selected_%s' % user.id, *selected)
    pl.execute()

    # 响应提交订单结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '下单成功', 'order_id': order.order_id})
```

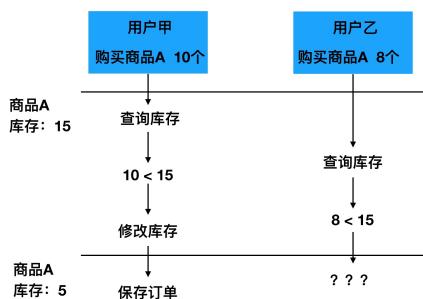


使用乐观锁并发下单

重要提示：

- 在多个用户同时发起对同一个商品的下单请求时，先查询商品库存，再修改商品库存，会出现资源竞争问题，导致库存的最终结果出现异常。

1. 并发下单问题演示和解决方案



解决办法：

• 悲观锁

- 当查询某条记录时，即让数据库为该记录加锁，锁住记录后别人无法操作，使用类似如下语法

```

select stock from tb_sku where id=1 for update;

SKU.objects.select_for_update().get(id=1)

```

- 悲观锁类似于我们在多线程资源竞争时添加的互斥锁，容易出现死锁现象，采用不多。

• 乐观锁

- 乐观锁并不是真实存在的锁，而是在更新的时候判断此时的库存是否是之前查询出的库存，如果相同，表示没人修改，可以更新库存，否则表示别人抢过资源，不再执行库存更新。类似如下操作

```

update tb_sku set stock=2 where id=1 and stock=7;

SKU.objects.filter(id=1, stock=7).update(stock=2)

```

• 任务队列

- 将下单的逻辑放到任务队列中（如celery），将并行转为串行，所有人排队下单。比如开启只有一个进程的Celery，一个订单一个订单的处理。

2. 使用乐观锁并发下单

思考：

- 下单成功的条件是什么？
 - 首先库存大于购买量，然后更新库存和销量时原始库存没变。

结论：

- 所以在用户库存满足的情况下，如果更新库存和销量时原始库存有变，那么继续给用户下单的机会。

```
class OrderCommitView(LoginRequiredMixin, View):
    """订单提交"""

    def post(self, request):
        """保存订单信息和订单商品信息"""
        # 获取当前保存订单时需要的信息
        .....

        # 显式的开启一个事务
        with transaction.atomic():
            # 创建事务保存点
            save_id = transaction.savepoint()

            # 暴力回滚
            try:
                # 保存订单基本信息 OrderInfo (—)
                order = OrderInfo.objects.create(
                    order_id=order_id,
                    user=user,
                    address=address,
                    total_count=0,
                    total_amount=Decimal('0'),
                    freight=Decimal('10.00'),
                    pay_method=pay_method,
                    status=OrderInfo.ORDER_STATUS_ENUM['UNPAID'] if pay_method == 0
                )
                if pay_method == OrderInfo.PAY_METHODS_ENUM['ALIPAY']:
                    order.status = OrderInfo.ORDER_STATUS_ENUM['UNSEND']
                else:
                    order.status = OrderInfo.ORDER_STATUS_ENUM['UNSEND']
            except:
                # 从redis读取购物车中被勾选的商品信息
                redis_conn = get_redis_connection('carts')
                redis_cart = redis_conn.hgetall('carts_%s' % user.id)
                selected = redis_conn.smembers('selected_%s' % user.id)
                carts = {}
                for sku_id in selected:
                    carts[int(sku_id)] = int(redis_cart[sku_id])
                sku_ids = carts.keys()

                # 遍历购物车中被勾选的商品信息
                for sku_id in sku_ids:
                    while True:
```

```

# 查询SKU信息
sku = SKU.objects.get(id=sku_id)

# 读取原始库存
origin_stock = sku.stock
origin_sales = sku.sales

# 判断SKU库存
sku_count = carts[sku.id]
if sku_count > origin_stock:
    # 事务回滚
    transaction.savepoint_rollback(save_id)
    return http.JsonResponse({'code': RETCODE.STOCKERR, 'errmsg': '库存不足'})

# 模拟延迟
# import time
# time.sleep(5)

# SKU减少库存， 增加销量
# sku.stock -= sku_count
# sku.sales += sku_count
# sku.save()

# 乐观锁更新库存和销量
new_stock = origin_stock - sku_count
new_sales = origin_sales + sku_count
result = SKU.objects.filter(id=sku_id, stock=origin_stock).update(stock=new_stock, sales=new_sales)
# 如果下单失败，但是库存足够时，继续下单，直到下单成功或者库存不足为止
if result == 0:
    continue

# 修改SPU销量
sku.spu.sales += sku_count
sku.spu.save()

# 保存订单商品信息 OrderGoods (多)
OrderGoods.objects.create(
    order=order,
    sku=sku,
    count=sku_count,
    price=sku.price,
)
# 保存商品订单中总价和总数量
order.total_count += sku_count
order.total_amount += (sku_count * sku.price)

# 下单成功或者失败就跳出循环
break

```

```

        # 添加邮费和保存订单信息
        order.total_amount += order.freight
        order.save()
    except Exception as e:
        logger.error(e)
        # 事务回滚
        transaction.savepoint_rollback(save_id)
        return http.JsonResponse({'code': RETCODE.DBERR, 'errmsg': '下单失败'})
    }

    # 保存订单数据成功，显式的提交一次事务
    transaction.savepoint_commit(save_id)

    # 清除购物车中已结算的商品
    pl = redis_conn.pipeline()
    pl.hdel('carts_%s' % user.id, *selected)
    pl.srem('selected_%s' % user.id, *selected)
    pl.execute()

    # 响应提交订单结果
    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '下单成功', 'order_id': order.order_id})

```

3. MySQL事务隔离级别

- 事务隔离级别指的是在处理同一个数据的多个事务中，一个事务修改数据后，其他事务何时能看到修改后的结果。
- MySQL数据库事务隔离级别主要有四种：
 - Serializable：串行化，一个事务一个事务的执行。
 - Repeatable read：可重复读，无论其他事务是否修改并提交了数据，在这个事务中看到的数据值始终不受其他事务影响。
 - Read committed：读取已提交，其他事务提交了对数据的修改后，本事务就能读取到修改后的数据值。
 - Read uncommitted：读取未提交，其他事务只要修改了数据，即使未提交，本事务也能看到修改后的数据值。
- 使用乐观锁的时候，如果一个事务修改了库存并提交了事务，那其他的事务应该可以读取到修改后的数据值，所以不能使用可重复读的隔离级别，应该修改为读取已提交（Read committed）。
- 修改方式：

```

python@ubuntu:~$ cd /etc/mysql/mysql.conf.d/
python@ubuntu:/etc/mysql/mysql.conf.d$ ls
mysqld.cnf  mysqld_safe_syslog.cnf
python@ubuntu:/etc/mysql/mysql.conf.d$ sudo vim mysqld.cnf

```

```
102 # ssl-ca=/etc/mysql/cacert.pem  
103 # ssl-cert=/etc/mysql/server-cert.pem  
104 # ssl-key=/etc/mysql/server-key.pem  
105 transaction-isolation=READ-COMMITTED
```

我的订单

The screenshot shows the Meiduo Mall user center interface. At the top, there's a navigation bar with links for '我的订单' (My Orders), '退出登录' (Logout), '用户中心' (User Center), '我的购物车' (Shopping Cart), and '我的订单' (My Orders). Below the navigation is a search bar with placeholder text '搜索商品' (Search Product) and a '搜索' (Search) button. The main content area is titled '全部订单' (All Orders) and shows two order lists. Order 1 (ID: 202304200000000005) contains an iPhone 8 Plus (64GB) at 6499.00元 and a MacBook Pro 13.3英寸 at 10988.00元, totaling 17487.00元. Order 2 (ID: 202304200000000005) contains a MacBook Pro 13.3英寸 at 10988.00元, totaling 10988.00元. Both orders have payment methods '支付宝' (Alipay) and '待支付' (Pending Payment). At the bottom of each order list is a page navigation bar with numbers 1, 2, 3, 4, 5, '下一页' (Next Page), and '尾页' (Last Page).

1. 请求方式

选项	方案
请求方法	GET
请求地址	/orders/info/(?P<page_num>\d+)/

2. 请求参数: 路径参数

参数名	类型	是否必传	说明
page_num	int	是	当前页码

3. 响应结果: HTML

```
user_center_order.html
```

4. 后端接口定义和实现

```
class UserOrderInfoView(LoginRequiredMixin, View):
    """我的订单"""

    def get(self, request, page_num):
        """提供我的订单页面"""
        user = request.user
        # 查询订单
        orders = user.orderinfo_set.all().order_by("-create_time")
        # 遍历所有订单
        for order in orders:
            # 绑定订单状态
            order.status_name = OrderInfo.ORDER_STATUS_CHOICES[order.status-1][1]
            # 绑定支付方式
            order.pay_method_name = OrderInfo.PAY_METHOD_CHOICES[order.pay_method-1][1]
            order.sku_list = []
            # 查询订单商品
            order_goods = order.skus.all()
```

```

# 遍历订单商品
for order_good in order_goods:
    sku = order_good.sku
    sku.count = order_good.count
    sku.amount = sku.price * sku.count
    order.sku_list.append(sku)

# 分页
page_num = int(page_num)
try:
    paginator = Paginator(orders, constants.ORDERS_LIST_LIMIT)
    page_orders = paginator.page(page_num)
    total_page = paginator.num_pages
except EmptyPage:
    return http.HttpResponseNotFound('订单不存在')

context = {
    "page_orders": page_orders,
    'total_page': total_page,
    'page_num': page_num,
}
return render(request, "user_center_order.html", context)

```

5.渲染我的订单信息

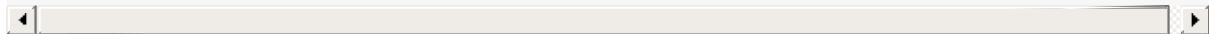
```

<div class="right_content clearfix">
    <h3 class="common_title2">全部订单</h3>
    {% for order in page_orders %}
        <ul class="order_list_th w978 clearfix">
            <li class="col01">{{ order.create_time.strftime('%Y-%m-%d %H:%M:%S') }}</li>

            <li class="col02">订单号: {{ order.order_id }}</li>
        </ul>
        <table class="order_list_table w980">
            <tbody>
                <tr>
                    <td width="55%">
                        {% for sku in order.sku_list %}
                            <ul class="order_goods_list clearfix">
                                <li class="col01"></li>
                                <li class="col02"><span>{{ sku.name }}</span><em>{{ sku.price }}元</em></li>
                                <li class="col03">{{ sku.count }}</li>
                                <li class="col04">{{ sku.amount }}元</li>
                            </ul>
                        {% endfor %}
                    </td>
                    <td width="15%">{{ order.total_amount }}元<br>含运费: {{ order.freight }}元</td>
                </tr>
            </tbody>
        </table>
    {% endfor %}
</div>

```

```
<td width="15%">{{ order.pay_method_name }}</td>
<td width="15%">
    <a @click="oper_btn_click('{{ order.order_id }}', {{ order.stats }})" class="oper_btn">{{ order.status_name }}</a>
</td>
</tr>
</tbody>
</table>
{% endfor %}
<div class="pagination">
    <div id="pagination" class="page"></div>
</div>
</div>
```



支付

提示：

- 如果用户选择的支付方式是 "支付宝"，在点击《去支付》时对接支付宝的支付系统。

支付宝介绍

支付宝开放平台入口

- <https://open.alipay.com/platform/home.htm>



1. 创建应用和沙箱环境

1. 创建应用

* 应用名称： 不超过32个字符，[查看命名规范](#)

应用图标：

请上传应用图标图片，支持.jpg,.jpeg,.png格式，建议320*320像素，小于3M

应用类型：
 网页应用 移动应用

网址url：
例：https://www.example.com/main或http://www.example.com/main

应用简介：

[确认创建](#)

2. 沙箱环境

支付宝提供给开发者的模拟支付的环境。跟真实环境是分开的。

- 沙箱应用：<https://openhome.alipay.com/platform/appDaily.htm?tab=info>

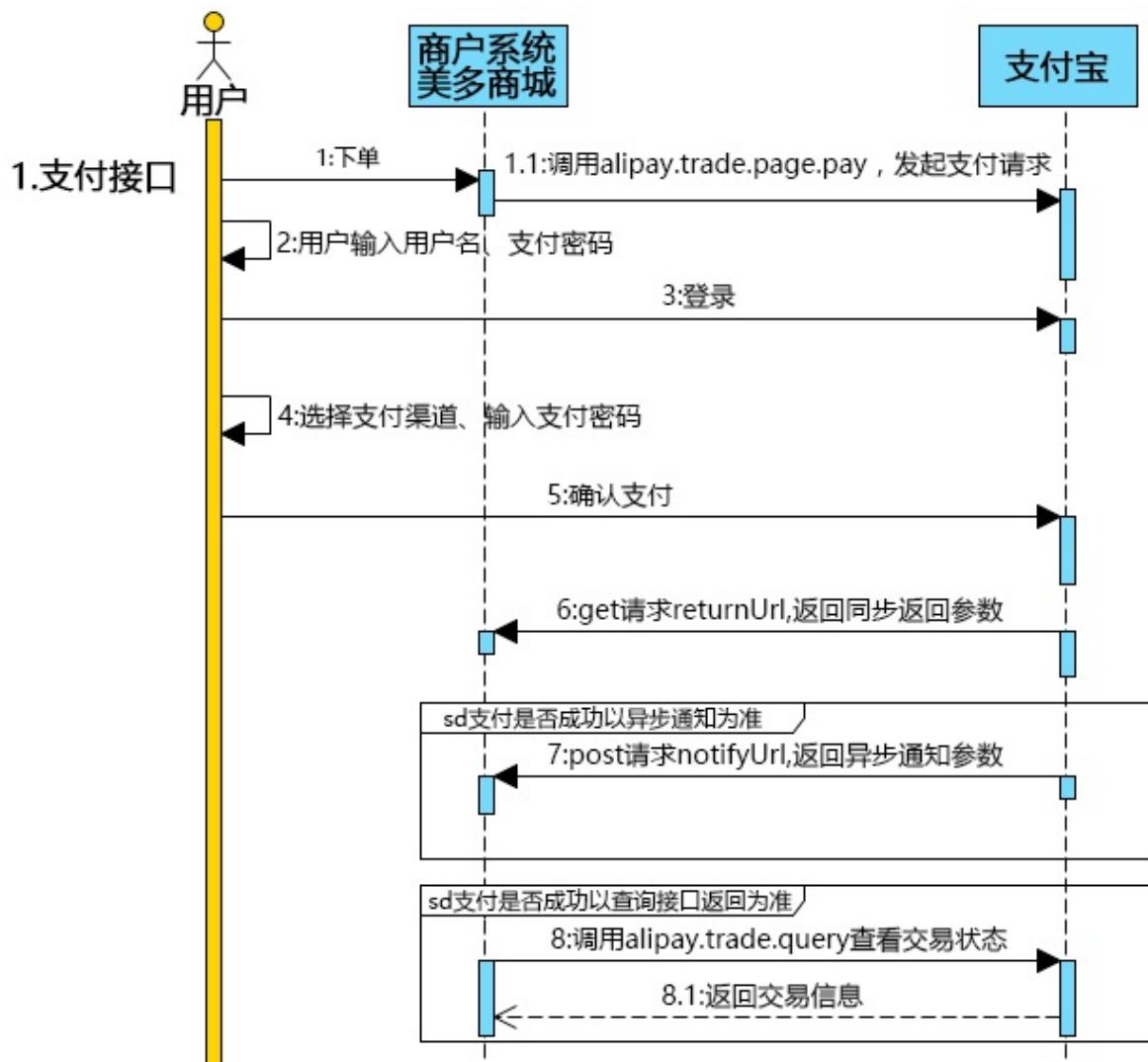


- 沙箱账号：<https://openhome.alipay.com/platform/appDaily.htm?tab=account>

2. 支付宝开发文档

- 文档主页: <https://openhome.alipay.com/developmentDocument.htm>
- 电脑网站支付产品介绍: <https://docs.open.alipay.com/270>
- 电脑网站支付快速接入: <https://docs.open.alipay.com/270/105899/>
- API列表: <https://docs.open.alipay.com/270/105900/>
- SDK文档: <https://docs.open.alipay.com/270/106291/>
- Python支付宝SDK: <https://github.com/fzlee/alipay/blob/master/README.zh-hans.md>
 - SDK安装: `pip install python-alipay-sdk --upgrade`

3. 电脑网站支付流程

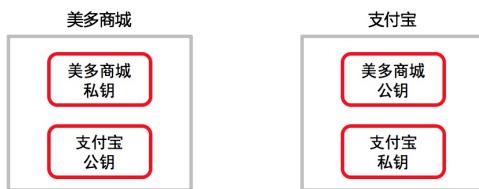


4. 配置RSA2公私钥

提示:

- 美多商城私钥加密数据，美多商城公钥解密数据。

- 支付宝私钥加密数据，支付宝公钥解密数据。



1.生成美多商城公私钥

```
$ openssl
$ OpenSSL> genrsa -out app_private_key.pem 2048 # 制作私钥RSA2
$ OpenSSL> rsa -in app_private_key.pem -pubout -out app_public_key.pem # 导出公钥

$ OpenSSL> exit
```

2.配置美多商城公私钥

- 配置美多商城私钥
 - 新建子应用 payment，在该子应用下新建文件夹 keys 用于存储公私钥。
 - 将制作的美多商城私钥 app_private_key.pem 拷贝到 keys 文件夹中。
- 配置美多商城公钥
 - 将 payment.keys.app_public_key.pem 文件中内容上传到支付宝。

3.配置支付宝公钥

- 将支付宝公钥内容拷贝到 payment.keys.alipay_public_key.pem 文件中。

A screenshot of a file explorer window. A red arrow points to a file named '支付宝公钥内容'. The file content is displayed below:

```
-----BEGIN PUBLIC KEY-----
支付宝公钥内容
-----END PUBLIC KEY-----
```

配置公私钥结束后



对接支付宝系统

订单支付功能

| 提示：

- 订单支付触发页面：《order_success.html》 和 《user_center_order.html》
- 我们实现订单支付功能时，只需要向支付宝获取登录链接即可，进入到支付宝系统后就是用户向支付宝进行支付的行为。



1. 请求方式

选项	方案
请求方法	GET
请求地址	/payment/({order_id})/

2. 请求参数: 路径参数

参数名	类型	是否必传	说明
order_id	int	是	订单编号

3. 响应结果: JSON

字段	说明
code	状态码
errmsg	错误信息
alipay_url	支付宝登录链接

4. 后端接口定义和实现

```
# 测试账号: pqcanx4910@sandbox.com
class PaymentView(LoginRequiredMixin, View):
    """订单支付功能"""

    def get(self, request, order_id):
        # 查询要支付的订单
        user = request.user
        try:
            order = OrderInfo.objects.get(order_id=order_id, user=user, status=OrderInfo.ORDER_STATUS_ENUM['UNPAID'])
        except OrderInfo.DoesNotExist:
            return http.HttpResponseForbidden('订单信息错误')

        # 创建支付宝支付对象
        alipay = AliPay(
            appid=settings.ALIPAY_APPID,
            app_notify_url=None, # 默认回调url
            app_private_key_path=os.path.join(os.path.dirname(os.path.abspath(__file__)), "keys/app_private_key.pem"),
            alipay_public_key_path=os.path.join(os.path.dirname(os.path.abspath(__file__)), "keys/app_public_key.pem"))
```

```

    file__)), "keys/alipay_public_key.pem"),
        sign_type="RSA2",
        debug=settings.ALIPAY_DEBUG
    )

    # 生成登录支付宝连接
    order_string = alipay.api_alipay_trade_page_pay(
        out_trade_no=order_id,
        total_amount=str(order.total_amount),
        subject="美多商城%s" % order_id,
        return_url=settings.ALIPAY_RETURN_URL
    )

    # 响应登录支付宝连接
    # 真实环境电脑网站支付网关: https://openapi.alipay.com/gateway.do? + order_stri
ng
    # 沙箱环境电脑网站支付网关: https://openapi.alipaydev.com/gateway.do? + order_s
tring
    alipay_url = settings.ALIPAY_URL + "?" + order_string
    return http.JsonResponse({ 'code': RETCODE.OK, 'errmsg': 'OK', 'alipay_url': alipay_url})

```

5.支付宝SDK配置参数

```

ALIPAY_APPID = '2016082100308405'
ALIPAY_DEBUG = True
ALIPAY_URL = 'https://openapi.alipaydev.com/gateway.do'
ALIPAY_RETURN_URL = 'http://www.meiduo.site:8000/payment/status/'

```

保存订单支付结果

1. 支付结果数据说明

- 用户订单支付成功后，支付宝会将用户重定向到
`http://www.meiduo.site:8000/payment/status/`，并携带支付结果数据。
- 参考统一收单下单并支付页面接口：<https://docs.open.alipay.com/270/alipay.trade.page.pay>

页面回跳参数

对于PC网站支付的交易，在用户支付完成之后，支付宝会根据API中商户传入的return_url参数，通过GET请求的形式将部分支付结果参数通知到商户系统。

公共参数：

参数	类型	是否必填	最大长度	描述	示例值
app_id	String	是	32	支付宝分配给开发者的应用ID	2016040501024706
method	String	是	128	接口名称	alipay.trade.page.pay/return
sign_type	String	是	10	签名算法类型，目前支持RSA2和RSA，推荐使用RSA2	RSA2
sign	String	是	256	支付宝对本次支付结果的签名，开发者必须使用支付宝公钥验签签名	详见示例
charset	String	是	10	编码格式，如UTF-8,Gbk,gb2312等	utf-8
timestamp	String	是	19	前台回调的时间，格式“yyyy-MM-dd HH:mm:ss”	2016-08-11 19:36:01
version	String	是	3	调用的接口版本，固定为：1.0	1.0
auth_app_id	String	是	32	授权方的appid值：由于本接口暂不开放第三方应用授权，因此auth_app_id=app_id	2016040501024706

业务参数：

参数	类型	是否必填	最大长度	描述	示例值
out_trade_no	String	是	64	商户网站唯一订单号	705011111115001111119
trade_no	String	是	64	该交易在支付系统中的交易流水号，最长64位。	2016081121001004630200142207
total_amount	Price	是	9	该笔订单的资金总额，单位为HMB-Yuan，取值范围为[0.01, 10000000.00]，精确到小数点后两位。	9.00
seller_id	String	是	16	收款支付宝账号对应的支付宝唯一用户号，以2088开头的纯16位数字	2088111111116894

提示：

我们需要将 `订单编号` 和 `交易流水号` 进行关联存储，方便用户和商家后续使用。

2. 定义支付结果模型类

```
class Payment(BaseModel):
    """支付信息"""
    order = models.ForeignKey(OrderInfo, on_delete=models.CASCADE, verbose_name='订单')
    trade_id = models.CharField(max_length=100, unique=True, null=True, blank=True, verbose_name="支付编号")

    class Meta:
        db_table = 'tb_payment'
        verbose_name = '支付信息'
        verbose_name_plural = verbose_name
```

3. 保存订单支付结果

1. 请求方式

选项	方案
请求方法	GET
请求地址	/payment/status/

2. 请求参数：路径参数

参考统一收单下单并支付页面接口中的《页面回跳参数》

3.响应结果: HTML

```
pay_success.html
```

4.后端接口定义和实现

注意: 保存订单支付结果的同时, 还需要修改订单的状态为 待评价

```
# 测试账号: pqcanx4910@sandbox.com
class PaymentStatusView(View):
    """保存订单支付结果"""

    def get(self, request):
        # 获取前端传入的请求参数
        query_dict = request.GET
        data = query_dict.dict()
        # 获取并从请求参数中剔除signature
        signature = data.pop('sign')

        # 创建支付宝支付对象
        alipay = AliPay(
            appid=settings.ALIPAY_APPID,
            app_notify_url=None,
            app_private_key_path=os.path.join(os.path.dirname(os.path.abspath(__file__)), "keys/app_private_key.pem"),
            alipay_public_key_path=os.path.join(os.path.dirname(os.path.abspath(__file__)), "keys/alipay_public_key.pem"),
            sign_type="RSA2",
            debug=settings.ALIPAY_DEBUG
        )
        # 校验这个重定向是否是alipay重定向过来的
        success = alipay.verify(data, signature)
        if success:
            # 读取order_id
            order_id = data.get('out_trade_no')
            # 读取支付宝流水号
            trade_id = data.get('trade_no')
            # 保存Payment模型类数据
            Payment.objects.create(
                order_id=order_id,
                trade_id=trade_id
            )

            # 修改订单状态为待评价
            OrderInfo.objects.filter(order_id=order_id, status=OrderInfo.ORDER_STATUS_ENUM['UNPAID']).update(
                status=OrderInfo.ORDER_STATUS_ENUM["UNCOMMENT"])

            # 响应trade_id
            context = {
```

```

        'trade_id': trade_id
    }
    return render(request, 'pay_success.html', context)
else:
    # 订单支付失败，重定向到我的订单
    return HttpResponseRedirect('非法请求')

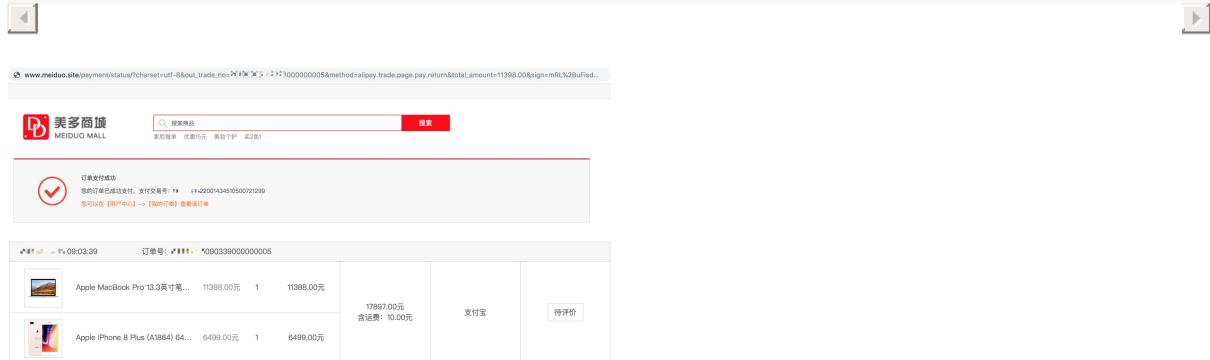
```

5.渲染支付成功页面信息

```

<div class="common_list_con clearfix">
    <div class="order_success">
        <p><b>订单支付成功</b></p>
        <p>您的订单已成功支付，支付交易号: {{ trade_id }}</p>
        <p><a href="{{ url('orders:info', args=(1, )) }}>您可以在【用户中心】->【我的订单】查看该订单</a></p>
    </div>
</div>

```



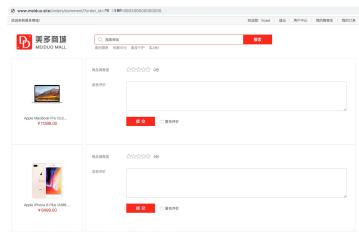
评价订单商品

提示：

点击《我的订单》页面中的《待评价》按钮，进入到订单商品评价页面。

评价订单商品

1. 展示商品评价页面



1.请求方式

选项	方案
请求方法	GET
请求地址	/orders/comment/

2.请求参数：查询参数

参数名	类型	是否必传	说明
order_id	int	是	订单编号

3.响应结果：HTML

goods_judge.html

4.后端接口定义和实现

```
class OrderCommentView(LoginRequiredMixin, View):
    """订单商品评价"""

    def get(self, request):
        """展示商品评价页面"""
        # 接收参数
        order_id = request.GET.get('order_id')
        # 校验参数
        try:
            OrderInfo.objects.get(order_id=order_id, user=request.user)
        except OrderInfo.DoesNotExist:
            return http.HttpResponseNotFound('订单不存在')

        # 查询订单中未被评价的商品信息
        try:
            uncomment_goods = OrderGoods.objects.filter(order_id=order_id, is_commented=False)
        except Exception:
            return http.HttpResponseServerError('订单商品信息出错')

        # 构造待评价商品数据
        uncomment_goods_list = []
        for goods in uncomment_goods:
            uncomment_goods_list.append({
```

```
'order_id':goods.order.order_id,
'sku_id':goods.sku.id,
'name':goods.sku.name,
'price':str(goods.price),
'default_image_url':goods.sku.default_image.url,
'comment':goods.comment,
'score':goods.score,
'is_anonymous':str(goods.is_anonymous),
})

# 渲染模板
context = {
    'uncomment_goods_list':uncomment_goods_list
}
return render(request, 'goods_judge.html', context)
```

2. 评价订单商品



1. 请求方式

选项	方案
请求方法	POST
请求地址	/orders/comment/

2. 请求参数：查询参数

参数名	类型	是否必传	说明
order_id	int	是	订单编号

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息

4. 后端接口定义和实现

```

class OrderCommentView(LoginRequiredMixin, View):
    """订单商品评价"""

    def get(self, request):
        """展示商品评价页面"""
        .....

    def post(self, request):
        """评价订单商品"""
        # 接收参数
        json_dict = json.loads(request.body.decode())
        order_id = json_dict.get('order_id')
        sku_id = json_dict.get('sku_id')
        score = json_dict.get('score')
        comment = json_dict.get('comment')
        is_anonymous = json_dict.get('is_anonymous')
        # 校验参数
        if not all([order_id, sku_id, score, comment]):
            return http.HttpResponseForbidden('缺少必传参数')
        try:
            OrderInfo.objects.filter(order_id=order_id, user=request.user, status=OrderInfo.ORDER_STATUS_ENUM['UNCOMMENT'])
        except OrderInfo.DoesNotExist:
            return http.HttpResponseForbidden('参数order_id错误')
        try:
            sku = SKU.objects.get(id=sku_id)
        except SKU.DoesNotExist:

```

```
        return http.HttpResponseForbidden('参数sku_id错误')
    if is_anonymous:
        if not isinstance(is_anonymous, bool):
            return http.HttpResponseForbidden('参数is_anonymous错误')

    # 保存订单商品评价数据
    OrderGoods.objects.filter(order_id=order_id, sku_id=sku_id, is_commented=False).update(
        comment=comment,
        score=score,
        is_anonymous=is_anonymous,
        is_commented=True
    )

    # 累计评论数据
    sku.comments += 1
    sku.save()
    sku.spu.comments += 1
    sku.spu.save()

    # 如果所有订单商品都已评价，则修改订单状态为已完成
    if OrderGoods.objects.filter(order_id=order_id, is_commented=False).count() == 0:
        OrderInfo.objects.filter(order_id=order_id).update(status=OrderInfo.ORDER_STATUS_ENUM['FINISHED'])

    return http.JsonResponse({'code': RETCODE.OK, 'errmsg': '评价成功'})
```

详情页展示评价信息



1. 请求方式

选项	方案
请求方法	POST
请求地址	/comments/(?P<sku_id>\d+)/

2. 请求参数：查询参数

参数名	类型	是否必传	说明
sku_id	int	是	商品SKU编号

3. 响应结果：JSON

字段	说明
code	状态码
errmsg	错误信息
comment_list[]	评价列表
username	发表评价的用户
comment	评价内容
score	分数

```
{
    "code": "0",
    "errmsg": "OK",
    "comment_list": [
        {
            "username": "itcast",
            "comment": "这是一个好手机！",
            "score": 4
        }
    ]
}
```

4.后端接口定义和实现

```
class GoodsCommentView(View):
    """订单商品评价信息"""

    def get(self, request, sku_id):
        # 获取被评价的订单商品信息
        order_goods_list = OrderGoods.objects.filter(sku_id=sku_id, is_commented=True).order_by('-create_time')[:30]
        # 序列化
        comment_list = []
        for order_goods in order_goods_list:
            username = order_goods.order.user.username
            comment_list.append({
                'username': username[0] + '***' + username[-1] if order_goods.is_anonymous else username,
                'comment': order_goods.comment,
                'score': order_goods.score,
            })
        return JsonResponse({'code': RETCODE.OK, 'errmsg': 'OK', 'comment_list': comment_list})
```

5.渲染商品评价信息

```
<div @click="on_tab_content('comment')" class="tab_content" :class="tab_content.comment?'current':''">
    <ul class="judge_list_con">
        <li class="judge_list fl" v-for="comment in comments">
            <div class="user_info fl">
                <b>[[comment.username]]</b>
            </div>
            <div class="judge_info fl">
                <div :class="comment.score_class"></div>
                <div class="judge_detail">[[comment.comment]]</div>
            </div>
        </li>
    </ul>
</div>
```

```
<li @click="on_tab_content('comment')" :class="tab_content.comment?'active':''>商品评价([[ comments.length ]])</li>
```

```
<div class="price_bar">
    <span class="show_pirce">¥<em>{{ sku.price }}</em></span>
    <a href="javascript:;" class="goods_judge">[[ comments.length ]]人评价</a>
</div>
```

提示：订单商品评价完成后，一个订单的流程就结束了，订单状态修改为 **已完成**。

订单号: 397****590390000000					
 Apple MacBook Pro 13.3英寸...	11388.00元	1	11388.00元		
 Apple iPhone 8 Plus (A1904) 64...	6499.00元	1	6499.00元	17897.00元 含运费: 10.00元	支付宝 已完成

性能优化

页面静态化

首页广告页面静态化

思考：

- 美多商城的首页访问频繁，而且查询数据量大，其中还有大量的循环处理。

问题：

- 用户访问首页会耗费服务器大量的资源，并且响应数据的效率会大大降低。

解决：

- 页面静态化

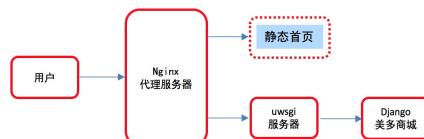
1. 页面静态化介绍

1.为什么要做页面静态化

- 减少数据库查询次数。
- 提升页面响应效率。

2.什么是页面静态化

- 将动态渲染生成的页面结果保存成html文件，放到静态文件服务器中。
- 用户直接去静态服务器，访问处理好的静态html文件。



3.页面静态化注意点

- 用户相关数据不能静态化：
 - 用户名、购物车等不能静态化。
- 动态变化的数据不能静态化：
 - 热销排行、新品推荐、分页排序数据等等。
- 不能静态化的数据处理：
 - 可以在用户得到页面后，在页面中向后端发送Ajax请求获取相关数据。
 - 直接使用模板渲染出来。
 - 其他合理的处理方式等等。

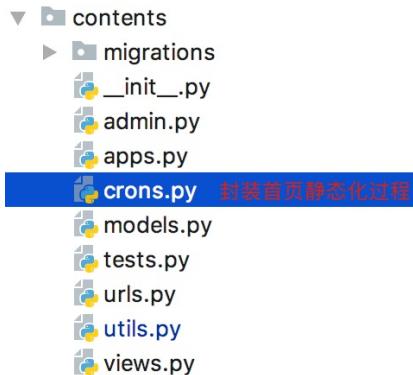
2. 首页页面静态化实现

1.首页页面静态化实现步骤

- 查询首页相关数据
- 获取首页模板文件
- 渲染首页html字符串

- 将首页html字符串写入到指定目录，命名'index.html'

2.首页页面静态化实现



```
def generate_static_index_html():
    """
    生成静态的主页html文件
    """

    print('%s: generate_static_index_html' % time.ctime())

    # 获取商品频道和分类
    categories = get_categories()

    # 广告内容
    contents = {}
    content_categories = ContentCategory.objects.all()
    for cat in content_categories:
        contents[cat.key] = cat.content_set.filter(status=True).order_by('sequence')

    # 渲染模板
    context = {
        'categories': categories,
        'contents': contents
    }

    # 获取首页模板文件
    template = loader.get_template('index.html')
    # 渲染首页html字符串
    html_text = template.render(context)
    # 将首页html字符串写入到指定目录，命名'index.html'
    file_path = os.path.join(settings.STATICFILES_DIRS[0], 'index.html')
    with open(file_path, 'w', encoding='utf-8') as f:
        f.write(html_text)
```

3.首页页面静态化测试效果

```
In [1]: from contents.crons import generate_static_index_html
In [2]: generate_static_index_html()
Tue Feb 26 03:22:46 2019: generate_static_index_html
```



提示：使用Python自带的 `http.server` 模块来模拟静态服务器，提供静态首页的访问测试。

```

# 进入到static目录
$ cd ~/projects/meiduo_project/meiduo_mall/meiduo_mall/static
# 开启测试静态服务器
$ python -m http.server 8080 --bind 127.0.0.1
  
```

```
(meiduo_mall) Python:~/projects/meiduo_project/meiduo_mall/meiduo_mall$ python -m http.server 8080 --bind 127.0.0.1
Serving HTTP on 127.0.0.1 port 8080 (http://127.0.0.1:8080/) ...
  
```

3. 定时任务crontab静态化首页

重要提示：

- 对于首页的静态化，考虑到页面的数据可能由多名运营人员维护，并且经常变动，所以将其做成定时任务，即定时执行静态化。
- 在Django执行定时任务，可以通过 `django-crontab` 扩展来实现。

1.安装 django-crontab

```
$ pip install django-crontab
```

2.注册 django-crontab 应用

```
INSTALLED_APPS = [
    'django_crontab', # 定时任务
]
```

3.设置定时任务

定时时间基本格式：

`* * * * *`

分 时 日 月 周 命令

M: 分钟（0-59）。每分钟用 * 或者 */1 表示

H: 小时（0-23）。（0表示0点）

D: 天（1-31）。

m: 月（1-12）。

d: 一星期内的天（0~6，0为星期天）。

定时任务分为三部分定义：

- 任务时间
- 任务方法
- 任务日志

```
CRONJOBS = [
    # 每1分钟生成一次首页静态文件
    ('*/1 * * * *', 'contents.crons.generate_static_index_html', '>> ' + os.path.join(os.path.dirname(BASE_DIR), 'logs/crontab.log'))
]
```

解决 crontab 中文问题

- 在定时任务中，如果出现非英文字符，会出现字符异常错误

```
CRONTAB_COMMAND_PREFIX = 'LANG_ALL=zh_cn.UTF-8'
```

4.管理定时任务

```
# 添加定时任务到系统中
$ python manage.py crontab add

# 显示已激活的定时任务
$ python manage.py crontab show

# 移除定时任务
$ python manage.py crontab remove
```

```
Tue Feb 26 06:48:01 2019: generate_static_index_html
Tue Feb 26 06:49:01 2019: generate_static_index_html
Tue Feb 26 06:50:01 2019: generate_static_index_html
```

商品详情页面静态化

提示：

- 商品详情页查询数据量大，而且是用户频繁访问的页面。
- 类似首页广告，为了减少数据库查询次数，提升页面响应效率，我们也要对详情页进行静态化处理。

静态化说明：

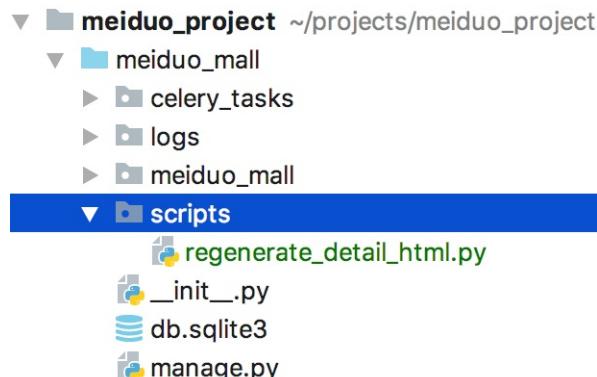
- 首页广告的数据变化非常的频繁，所以我们最终使用了 定时任务 进行静态化。
- 详情页的数据变化的频率没有首页广告那么频繁，而且是当SKU信息有改变时才要更新的，所以我们采用新的静态化方案。
 - 方案一：通过Python脚本手动一次性批量生成所有商品静态详情页。
 - 方案二：后台运营人员修改了SKU信息时，异步的静态化对应的商品详情页面。
 - 我们在这里先使用方案一来静态详情页。当有运营人员参与时才会补充方案二。

注意：

- 用户数据和购物车数据不能静态化。
- 热销排行和商品评价不能静态化。

1. 定义批量静态化详情页脚本文件

1.准备脚本目录和Python脚本文件



2.指定Python脚本解析器

```
#!/usr/bin/env python
```

3.添加Python脚本导包路径

```
#!/usr/bin/env python

import sys
sys.path.insert(0, '../')
```

4.设置Python脚本Django环境

```
#!/usr/bin/env python

import sys
sys.path.insert(0, '../')

import os
if not os.getenv('DJANGO_SETTINGS_MODULE'):
    os.environ['DJANGO_SETTINGS_MODULE'] = 'meiduo_mall.settings.dev'

import django
django.setup()
```

5.编写静态化详情页Python脚本代码

```
#!/usr/bin/env python

import sys
sys.path.insert(0, '../')

import os
if not os.getenv('DJANGO_SETTINGS_MODULE'):
    os.environ['DJANGO_SETTINGS_MODULE'] = 'meiduo_mall.settings.dev'

import django
django.setup()

from django.template import loader
from django.conf import settings

from goods import models
from contents.utils import get_categories
from goods.utils import get_breadcrumb


def generate_static_sku_detail_html(sku_id):
    """
    生成静态商品详情页面
    :param sku_id: 商品sku id
    """
    # 获取当前sku的信息
    sku = models.SKU.objects.get(id=sku_id)

    # 查询商品频道分类
    categories = get_categories()
    # 查询面包屑导航
    breadcrumb = get_breadcrumb(sku.category)
```

```

# 构建当前商品的规格键
sku_specs = sku.specs.order_by('spec_id')
sku_key = []
for spec in sku_specs:
    sku_key.append(spec.option.id)
# 获取当前商品的所有SKU
skus = sku.spu.sku_set.all()
# 构建不同规格参数（选项）的SKU字典
spec_sku_map = {}
for s in skus:
    # 获取SKU的规格参数
    s_specs = s.specs.order_by('spec_id')
    # 用于形成规格参数-SKU字典的键
    key = []
    for spec in s_specs:
        key.append(spec.option.id)
    # 向规格参数-SKU字典添加记录
    spec_sku_map[tuple(key)] = s.id
# 获取当前商品的规格信息
goods_specs = sku.spu.specs.order_by('id')
# 若当前SKU的规格信息不完整，则不再继续
if len(sku_key) < len(goods_specs):
    return
for index, spec in enumerate(goods_specs):
    # 复制当前SKU的规格键
    key = sku_key[:]
    # 该规格的选项
    spec_options = spec.options.all()
    for option in spec_options:
        # 在规格参数SKU字典中查找符合当前规格的SKU
        key[index] = option.id
        option.sku_id = spec_sku_map.get(tuple(key))
    spec.spec_options = spec_options

# 上下文
context = {
    'categories': categories,
    'breadcrumb': breadcrumb,
    'sku': sku,
    'specs': goods_specs,
}

template = loader.get_template('detail.html')
html_text = template.render(context)
file_path = os.path.join(settings.STATICFILES_DIRS[0], 'detail/' + str(sku_id) + '.html')
with open(file_path, 'w') as f:
    f.write(html_text)

if __name__ == '__main__':
    skus = models.SKU.objects.all()

```

```

for sku in skus:
    print(sku.id)
    generate_static_sku_detail_html(sku.id)

```

2. 执行批量静态化详情页脚本文件

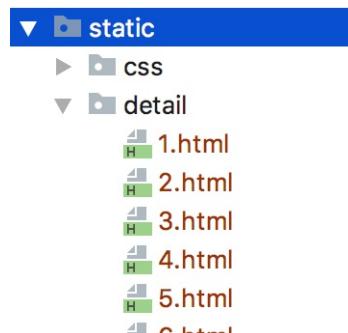
1. 添加Python脚本文件可执行权限

```
$ chmod +x regenerate_detail_html.py
```

```
(meiduo_mall) Python:~/projects/meiduo_project/meiduo_mall/scripts$ ls
regenerate_detail_html.py
(meiduo_mall) Python:~/projects/meiduo_project/meiduo_mall/scripts$ ls -l
total 8
-rwxr-xr-x 1 chenxl  staff  2555 7月 15 15:45 regenerate_detail_html.py
```

2. 执行批量静态化详情页脚本文件

```
$ cd ~/projects/meiduo_project/meiduo_mall/scripts
$ ./regenerate_detail_html.py
```



提示：跟测试静态首页一样的，使用Python自带的http.server模块来模拟静态服务器，提供静态首页的访问测试。

```
# 进入到static目录
$ cd ~/projects/meiduo_project/meiduo_mall/meiduo_mall/static
# 开启测试静态服务器
$ python -m http.server 8080 --bind 127.0.0.1
```

MySQL读写分离

提示：

我们的项目中已经存在非常多的数据库表了，数据量也会逐渐增多，所以我们需要做一些数据库的安全和性能的优化。

对于数据库的优化，我们选择使用MySQL读写分离实现。涉及内容包括 `主从同步` 和 `Django` 实现 `MySQL读写分离`。

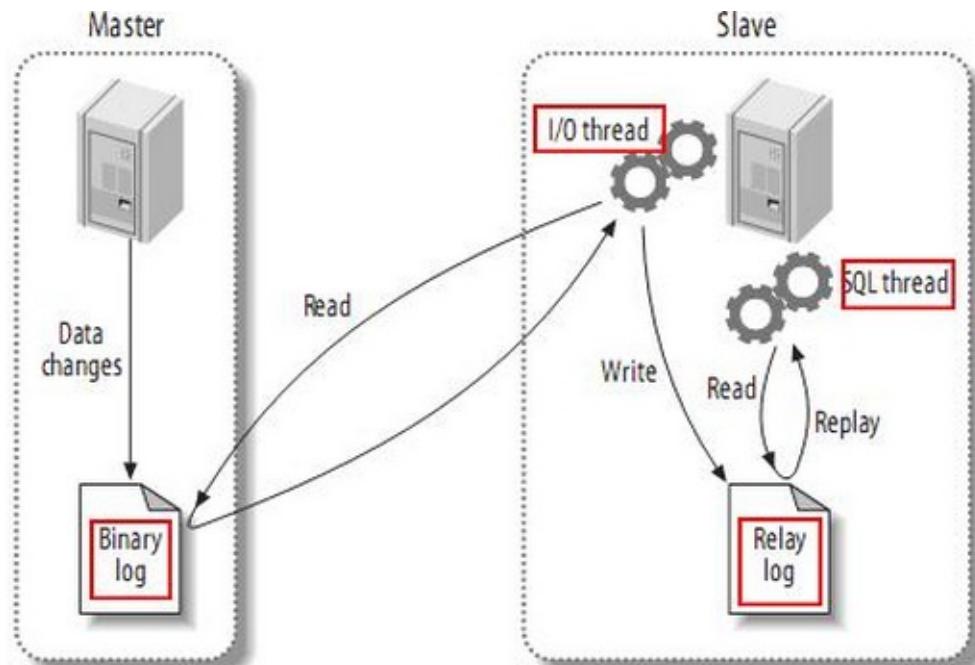
MySQL主从同步

1. 主从同步机制

1. 主从同步介绍和优点

- 在多台数据服务器中，分为主服务器和从服务器。一台主服务器对应多台从服务器。
- 主服务器只负责写入数据，从服务器只负责同步主服务器的数据，并让外部程序读取数据。
- 主服务器写入数据后，即刻将写入数据的命令发送给从服务器，从而使得主从数据同步。
- 应用程序可以随机读取某一台从服务器的数据，这样就可以分摊读取数据的压力。
- 当从服务器不能工作时，整个系统将不受影响；当主服务器不能工作时，可以方便地从从服务器选举一台来当主服务器
- 使用主从同步的优点：
 - 提高读写性能
 - 因为主从同步之后，数据写入和读取是在不同的服务器上进行的，而且可以通过增加从服务器来提高数据库的读取性能。
 - 提高数据安全
 - 因为数据已复制到从服务器，可以在从服务器上备份而不破坏主服务器相应数据。

2. 主从同步机制



MySQL服务器之间的主从同步是基于 二进制日志机制，主服务器使用二进制日志来记录数据库的变动情况，从服务器通过读取和执行该日志文件来保持和主服务器的数据一致。

2. Docker安装运行MySQL从机

提示：

- 本项目中我们搭建 一主一从 的主从同步。
- 主服务器：ubuntu操作系统中的MySQL。
- 从服务器：Docker容器中的MySQL。

1.获取MySQL镜像

- 主从同步尽量保证多台MySQL的版本相同或相近。

```
$ sudo docker image pull mysql:5.7.22
或
$ sudo docker load -i 文件路径/mysql_docker_5722.tar
```

2.指定MySQL从机配置文件

- 在使用Docker安装运行MySQL从机之前，需要准备好从机的配置文件。
- 为了快速准备从机的配置文件，我们直接把主机的配置文件拷贝到从机中。

```
# 进入到家目录
$ cd ~
# 新建mysql_slave目录
$ mkdir mysql_slave
# 进入到mysql_slave目录
$ cd mysql_slave
# 准备从机的配置文件：拷贝的主机的配置文件
$ cp -r /etc/mysql/mysql.conf.d ./
# 新建从机数据目录
$ mkdir data
```

3.修改MySQL从机配置文件

- 编辑 `~/mysql_slave/mysql.conf.d/mysqld.cnf` 文件。
- 由于主从机都在同一个电脑中，所以我们选择使用不同的端口号区分主从机，从机端口号是8306。

```
# 从机端口号
port = 8306
# 关闭日志
general_log = 0
# 从机唯一编号
server_id = 2
```

4.Docker安装运行MySQL从机

- `MYSQL_ROOT_PASSWORD`：创建 root 用户的密码为 mysql。

```
$ sudo docker run --name mysql-slave -e MYSQL_ROOT_PASSWORD=mysql -d --network=host
-v /home/python/mysql_slave/data:/var/lib/mysql -v /home/python/mysql_slave/mysql.
conf.d:/etc/mysql/mysql.conf.d mysql:5.7.22
```

5. 测试从机是否创建成功

```
$ mysql -uroot -pmysql -h 127.0.0.1 --port=8306
```

3. 主从同步实现

1. 配置主机（ubuntu中MySQL）

- 配置文件如有修改，需要重启主机。
 - `sudo service mysql restart`

```
# 进入到主机配置文件目录
cd /etc/mysql/mysql.conf.d/
# 开启日志
general_log_file = /var/log/mysql/mysql.log
general_log = 1
# 主机唯一编号
server-id = 1
# 二进制日志文件
log_bin = /var/log/mysql/mysql-bin.log
```

2. 从机备份主机原有数据

- 在做主从同步时，如果从机需要主机上原有数据，就要先复制一份到从机。

```
# 1. 收集主机原有数据
$ mysqldump -uroot -pmysql --all-databases --lock-all-tables > ~/master_db.sql

# 2. 从机复制主机原有数据
$ mysql -uroot -pmysql -h127.0.0.1 --port=8306 < ~/master_db.sql
```

3. 主从同步实现

- 1. 创建用于从服务器同步数据的帐号

```
# 登录到主机
$ mysql -uroot -pmysql
# 创建从机账号
$ GRANT REPLICATION SLAVE ON *.* TO 'slave'@'%' identified by 'slave';
# 刷新权限
$ FLUSH PRIVILEGES;
```

- 2. 展示ubuntu中MySQL主机的二进制日志信息

```
$ SHOW MASTER STATUS;
```



- 3.Docker中MySQL从机连接ubuntu中MySQL主机

```
# 登录到从机
$ mysql -uroot -pmysql -h 127.0.0.1 --port=8306
# 从机连接到主机
$ change master to master_host='127.0.0.1', master_user='slave', master_password='slave',master_log_file='mysql-bin.000250', master_log_pos=990250;
# 开启从机服务
$ start slave;
# 展示从机服务状态
$ show slave status \G
```

```
+----+-----+
|   | Slave_IO_Running: Yes
|   | Slave_SQL_Running: Yes
|   +-----+
| Master_Host: 127.0.0.1
| Master_Port: 3306
| Connect_Retry: 60
| Master_Log_File: mysql-bin.000250
| Read_Master_Log_Pos: 990250
| Relay_Master_Log_File: mysql-bin.000025
| Slave_SQL_Running: Yes
+----+
```

测试：

在主机中新建一个数据库后，直接在从机查看是否存在。

Django实现MySQL读写分离

1. 增加slave数据库的配置

```

DATABASES = {
    'default': { # 写 (主机)
        'ENGINE': 'django.db.backends.mysql', # 数据库引擎
        'HOST': '192.168.103.158', # 数据库主机
        'PORT': 3306, # 数据库端口
        'USER': 'itcast', # 数据库用户名
        'PASSWORD': '123456', # 数据库用户密码
        'NAME': 'meiduo_mall' # 数据库名字
    },
    'slave': { # 读 (从机)
        'ENGINE': 'django.db.backends.mysql',
        'HOST': '192.168.103.158',
        'PORT': 8306,
        'USER': 'root',
        'PASSWORD': 'mysql',
        'NAME': 'meiduo_mall'
    }
}

```

2. 创建和配置数据库读写路由

1. 创建数据库读写路由

- 在 `meiduo_mall.utils.db_router.py` 中实现读写路由

```

class MasterSlaveDBRouter(object):
    """数据库读写路由"""

    def db_for_read(self, model, **hints):
        """读"""
        return "slave"

    def db_for_write(self, model, **hints):
        """写"""
        return "default"

    def allow_relation(self, obj1, obj2, **hints):
        """是否运行关联操作"""
        return True

```

2. 配置数据库读写路由

```
DATABASE_ROUTERS = ['meiduo_mall.utils.db_router.MasterSlaveDBRouter']
```