Django学习笔记

`python manage.py runserver`

项目结构 - settings.py

- `settings.py` 包含 `Django` 项目启动的所有配置项
- 配置项分为 共有配置 和 自定义配置
- 配置项格式例： `BASE_DIR = xxxx`
- 公有配置 - `Django` 官方提供的基础配置
- **引入方式：** `from django.conf import settings`

settings.py解析

```
"""
Django settings for DjangoTest_mysite1 project.

Generated by 'django-admin startproject' using Django 4.0.1.

For more information on this file, see
https://docs.djangoproject.com/en/4.0/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/4.0/ref/settings/
"""

from pathlib import Path



# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
```

```
"""
用于绑定当前项目的绝对路径（动态计算出来的），所有文件夹都可以依赖此路径
Path(__file__)  -----  项目的绝对路径
Path(__file__).resolve().parent  -----  项目的上一级目录
Path(__file__).resolve().parent.parent  ----- 项目的上上一级目录
"""



# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/4.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-o=35s#xm29qaa-
73nmg^v2ei4=jvkra$96gxhvms1bl_!p387='



# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True
"""
用于配置Django项目的启动模式，取值
True ----- 调试模式
1. 检测代码改动后，立刻重启服务
2. 报错页面

False ----- 正式启动模式 / 上线模式
"""



ALLOWED_HOSTS = []
"""
设置允许访问到本项目的Host头，可以有效限制一些脏请求进行过滤，限制域名
[]-----空列表，表示只有请求头中host为127.0.0.1，localhost能访问本项目 -- Debug = True时
有效
['*']-----表示任何请求头的host都能访问到当前项目
['127.0.0.1','192.168.1.3']-----表示只有当前两个host头的值可以访问当前项目

示例：如果要在局域网其他主机也可以访问此主机的Django服务，启动方式如下：
        python manage.py runserver 0.0.0.0:5000
        指定网络设备如果内网环境下其他住建局想正常访问该站点，需要添加['内网ip']
        访问内网ip ----- ipconfig
"""

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',#session启用
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
"""
```

```
配置Django的应用
"""




MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',#session启用
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',        #csrf验证
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
"""
用于注册中间件
"""


ROOT_URLCONF = 'DjangoTest_mysite1.urls'
"""
主路由的位置
"""




TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [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',
            ],
        },
    },
]
"""
模板

"""



WSGI_APPLICATION = 'DjangoTest_mysite1.wsgi.application'
"""
正式启动会用
"""
```

```
# Database
# https://docs.djangoproject.com/en/4.0/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}

'''
配置数据库，我们使用mysql
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        #'NAME': BASE_DIR / 'db.mysql',
        'NAME' : 'DjangoTest_mysite1',
        'USER' :  'root',
        'PASSWORD' : '88888888',
        'HOST' : '127.0.0.1',
        'PORT' : '3306'
    }
}

'''


# Password validation
# https://docs.djangoproject.com/en/4.0/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]


# Internationalization
# https://docs.djangoproject.com/en/4.0/topics/i18n/

LANGUAGE_CODE = 'en-us'
#中文是'zh-Hans'
TIME_ZONE = 'UTC'
```

```
#东八区是'Asia/Shanghai'
USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.0/howto/static-files/

STATIC_URL = 'static/'

# Default primary key field type
# https://docs.djangoproject.com/en/4.0/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

URL 和 视图函数简介

URL组成部分

- 定义 - 即统一资源定位符 Uniform Resource Locator
- 作用 - 用来表示互联网上某个资源的地址
- URL的一般语法格式
- `protocol : // hostname[:port] / path [?query][#fragment]`
- protocol ---- 协议（http/https）
- hostname ----- 域名，主机名
- port --- 端口号 默认80
- path ----- 路由
- **query ----- 查询 。可选，用于给动态网络传递参数，可有多个参数，用"&"符号隔开，每一个参数的名和值用"="符号隔开**
- fragment ----- 信息片段，锚点 。 字符串，用于指定网络资源中的片段。例如一个网页中有很多名词解释，**可使用fragment直接定位到某一个名词解释**

处理URL请求过程

1. `Django`从配置文件中根据ROOT——URLCONF 找到主路由文件；默认情况下，该文件在 项目同名目录下的urls
2. `Django` 加载 主路由 文件中的 `urlpatterns` 变量 [很多路由的数组]
3. 依次匹配 `urlpatterns` 中的 `path`，匹配到第一个合适的中断后续匹配
4. 匹配成功--调用对应的视图函数处理请求，返回响应
5. 匹配失败--返回404响应

```
from django.contrib import admin
from django.urls import path
from student.views import *

urlpatterns = [
    path('', home, name='home'),
    path('register/', register, name='register'),
    path('logout/', logout, name='logout'),
    path('login/', login, name='login'),
    path('admin/', admin.site.urls),
]
```

视图函数简介

- 视图函数是用于接受一个浏览器请求（HttpRequest对象）并通过HttpResponse对象返回响应的函数。此函数可以接受浏览器请求并根据业务逻辑返回相应的响应内容给浏览器
- 语法

```
def xxx_view(request[,其他参数...])
    return HttpResponse对象
```

路由配置 - path

- path() 函数
- 导入 - `from django.urls import path`
- 语法 - `path(route, views , name = None)`
- 参数:

1. route : 字符串类型，匹配的请求路径
2. views : 指定路径所对应的视图处理函数的名称,**注意不要加括号**
3. name : 为地址起别名， 在模板中地址反向解析时使用

path - 转换器

- path转换器
- 语法 : `<转换器类型:自定义名>`
- 作用： 若转换器类型匹配到对应类型的数据，则将**数据按照关键字传参的方式**传递给视图函数
- 例子： `path('page/<int:page>',views.xxx)`

转换器类型	作用
str	匹配除了'/'之外的非空字符串
int	匹配0或任何正整数。返回一个int
slug	匹配任意由ASCII字母或数字以及连字符和下划线组成的短标签
path	匹配非空字段，包括路径分隔符'/'

路由配置 - re_path（可以用字符串+.split()解决）

- re_path()函数
- 在url的匹配过程中可以使用正则表达式进行精确匹配
- 语法：
  - re_path(reg,view,name=xxx)
  - 正则表达式为命名分组模式`(?P<name>pattern)`；匹配提取参数后**用关键字的传参方式**传递给视图函数

HTTP 请求和响应 与 视图函数的深造

请求中的方法

- 根据HTTP标准，HTTP请求可以使用多种请求方法。
- HTTP1.0定义了三种请求方法：GET,POST和HEAD方法（最常用）
- HTTP1.1新增了五种请求方法：OPTIONS,PUT,DELETE,TRACE和CONNECT方法

方法	描述
GET	**请求指定的页面信息，并返回实体主体**
HEAD	类似于get请求，只不过返回的相应没有具体的内容，用于获取报头
POST	**向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改**
PUT	从客户端向服务器传送的数据取代指定的文档的内容（**更新**）
DELETE	请求服务器删除指定的页面
CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器
OPTIONS	允许客户端查看服务器的性能
TRACE	回显服务器收到的请求，主要用于测试或者诊断

Django中的请求

- 请求在Django中实则就是视图函数的第一个参数，即HttpRequest对象
- Django收到http协议的请求后，会根据请求数据报文创建HttpRequest对象
- HttpRequest对象 通过属性 描述了请求的说有相关信息

  HttpRequest内含的属性:

属性	简介
path_info	URL字符串
method	**字符串，表示HTTP请求方法，常用值：'GET','POST'**
GET	**QueryDict查询字典的对象，包含get请求方式的所有数据**
POST	**QueryDict查询字典的对象，包含post请求方式的所有数据**
FILES	类似于字典的对象，包含所有的上传文件信息
COOKIES	Python字典，包含所有的cookie，键和值都为字符串
session	**类似于字典的对象，表示当前的会话**
body	字符串，请求体的内容（POST或PUT）
scheme	请求协议（'http'/'https'）

属性	简介
request.get_ful_path()	请的完整路径
request.META	请求中的元数据（消息头）

Django中的响应对象

构造函数格式：

```
HttpResponse(content = 响应体， content_type = 响应体数据类型 ， status = 状态码)
```

作用：

向客户端浏览器返回响应，同时携带响应体内容

重定向： `HttpResponseRedirect('/分路由');`

GET和POST请求

- 无论是GET还是POST，**统一都由视图函数接受请求**，通过判断**request.method区分**具体的请求动作

```
if request.method == 'GET' :
    pass
elif request.method == 'POST' :
    #处理用户提交的数据
    pass
else :
```

GET处理

- GET请求动作，一般用于向服务器获取数据，一般是用如下方法：
  - **request.GET.get('参数','默认值')，request.GET.getlist('参数')**

- 能够产生GET请求的场景：
  - **浏览器地址蓝中输入URL，回车后**
  -
  - form表单中的method为get

- GET请求方式中，如果有数据需要传递给服务器，通常会用查询字符串（Query String）传递。**注意：不要传递敏感信息**
- URL格式：地址？参数 = 值 & 参数 = 值

POST处理

- POST请求动作，一般用于向服务器提交大量/隐私数据

Django的设计模式及模板层

模板配置

在settings.py中TEMPLATES配置项

1. BACKEND: 指定模板引擎
2. DIRS：模板的搜索目录（可以是一个或者多个）
3. APP_DIRS：是否要在应用中的templates文件夹中搜索模板文件
4. **OPTIONS**：有关模板的选项

- 配置项中需要修改的部分:
- **设置DIRS -** `'DIRS' : [os.path.join(BASE_DIR,'template')],`

模板的加载方式

- 方案一 ------- 通过loader获取模板，通过HttpResponse进行相应

在视图函数中

```
from django.template import loader
# 1. 通过loader加载模板
t = loader.get_template("模板文件名")
# 2. 将t转换为 HTML 字符串
html = t.render(字典数据)
# 3. 用响应对象将转换的字符串内容返回给浏览器
return HttpResponse(htmp)
```

- **方案2 ------ 使用render()直接加载并响应模板**
- 在视图函数中：

```
from django.shortcuts import render
return render(request,'模板文件名', 字典数据)
```

视图层与模板层之间的交互

1. 视图函数中可以将python变量封装到字典中传递到模板

样例:

```
def xxx_view(request):
    dic = {
        "变量1" :  "值1",
        "变量2" :  "值2"
    }
    return render(request , "xxx.html", dic)
```

2. 模板中，我们**可以用{{变量名}}的语法 调用视图传进来的变量**

模板层-变量和标签

在**模板中使用变量**的语法:

```
{{变量名}}
{{变量名.index}}
{{变量名.key}}
{{对象.方法}}
{{函数名}}
```

在**模板中使用标签**的语法:

作用：将一些服务器端的功能嵌入到模板中，例如流程控制等等

标签语法:

```
{% 标签 %}
...
{% 结束标签 %}
###############
{% if %}

{% elif %}

{% endif %}
############
{% for 变量 in 可迭代对象 %}
    ...循环语句
{% empty %}
    ...可迭代对象无数据时填充的语句
{% endfor %}
```

模板层-过滤器和继承

模板过滤器

定义：在变量输出时对变量的值进行处理

作用：可以通过使用 过滤器来该改变变量的输出显示

语法：{{变量 | 过滤器1 : '参数值1' | 过滤器2 ： '参数值2' ....}}

过滤器	说明
lower	将字符串转换为全部小写
upper	将字符串转换为全部大写
safe	默认不对变量内的字符串进行html转义
add : "n"	将value的值增加 n
truncatechars : 'n'	如果字符串字符多余指定的祖父数量，那么会被截断。阶段的字符串将以可翻译的省略号序列结尾
...	

模板的继承

语法 - 父模板：

1. 定义父模板中的块block标签
2. 表示出哪些在子模块中式允许被修改的
3. bolck标签： 在父模板中定义，可以在子模版中覆盖

语法 - 子模版：

1. 继承模板 extends 标签（**写在模板文件的第一行**）

   例如 {% extends 'base.html' %}

2. 子模版重写父模板中的内容块

   {% block block_name %}

   子模版块用来覆盖父模板中 block_name 块的内容

   {% endblock block_name %}

URL反向解析

代码中URL出现的位置

1. 模板html中
   1. `<a href = 'url'>超链接</a>` 点击后页面跳转
   2. `<form action = 'url' method = 'post'>` form表单中的数据 用post方法提交至url
2. 视图函数中 - 302跳转 `HttpResponseRedirect('url')`；将用户地址栏中的地址跳转到url中

URL反向解析

url反向解析式指在视图或者模板中，用path定义的名称来动态查找或计算出相应的路由

path 函数的用法

- `path(route, views, name = '别名')`
- 根据path中的'name = '关键字传参给url却似那个了一个唯一确定的名字，在模板或视图中，可以通过这个名字反向推断出此url信息

模板中 - 通过url标签实现地址的反向解析

```
{% url  '别名'  %}
{% url  '别名' '参数值1' '参数值2'  %}
ex:
```

静态文件

- 静态文件配置 - settings.py中

1. **配置静态文件的访问路径【该配置默认存在】**
    1. 通过哪个url地址找静态文件
    2. STATIC_URL = 'static/'
    3. 说明，指定范文静态文件时时需要通过/static/xxx（xxx表示具体静态资源位置）
2. **配置静态文件的存储路径 STATICFILES_DIRS**
    - STATICFILES_DIRS保存的时静态文件在服务端的存储位置

```
# file : setting.py
STATICFILES_DIRS = (
    os.path.join(BASE_DIR,"static"),
)
```

Django应用和分布式路由

应用的创建

步骤一：

用manage.py中的子命令startapp创建应用文件夹

`python manage.py startapp user`

步骤二：

在settings.py的INSTALLED_APPS列表中配置安装此应用

分布式路由

Django中，主路由配置文件（url.py）可以不处理用户具体路由，主路由配置文件可以做请求的分发（分布式请求处理）。具体的请求可以由各自的应用来进行处理

配置分布式路由

步骤一：**主路由中调用include函数**

- **语法：** `include('app名字.url模块名')`
- 作用：用于将当前路由转到哥哥应用的路由配置文件的urlpatterns进行分布式处理

模型层及ORM介绍

cmd数据库指令

```
cd C:\Program Files\MySQL\MySQL Server 8.0\bin

DjangoTest_mysite1          #项目名称

mysql -u root -p            #登录

show databases;             #显示所有数据库

show global variables like 'port';  #显示端口号

use DjangoTest_mysite1; #打开某一个数据库

show tables;    #查看数据库中的表

desc bookstore_book; # 查看表结构

create database 数据库名 default charset utf8;   #创建数据库

drop database 数据库名;  #删除数据库
```

- 模型层 -- 负责跟数据库之间进行通信

Django配置mysql

- 创建数据库
- 进入mysql数据库 执行
  - create database 数据库名 **default charset utf8**
  - 通常数据库名跟项目名保持一致
- settings.py里面进行数据库的配置
  - 修改DATABASES配置想的内容，由sqlite3 变为 mysql

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        #'NAME': BASE_DIR / 'db.mysql',
        'NAME' : 'DjangoTest_mysite1',
        'USER' :  'root',
        'PASSWORD' : '88888888',
        'HOST' : '127.0.0.1',
        'PORT' : '3306'
    }
}
```

什么是模型

- 模型是一个Pyhton类，它是由django.db.models.Model派生出来的子类
- **一个模型类代表数据库中的一张数据表**
- 模型类中**每一个类属性都代表数据库中的一个字段**
- 模型是数据交互的接口，是表示和操作数据库的方式和方法

ORM框架

1. 建立模型类和表之间的对于关系，允许我们通过面向对象的方式来操作数据库
2. 根据涉及的模型类生成数据库中表格
3. 通过简单的配置就可以进行数据库的切换

模型的编写

- 模型类代码示例：

```
from django.db import models


class Book(models.Model):
    #字符串类型，一定要填max_length
    title = models.CharField("书名", max_length=50, default='')
    #小数点类型，max_digits表示小数总显示数的位数，decimal_places为小数位数
    price = models.DecimalField('定价', max_digits=7, decimal_places=2,
default=0.0)
```

数据库迁移

- 迁移时Django同步你对模型所做更改（添加字段，删除模型等）到你的数据库模式的方式

  生成迁移文件 ------ 执行 `python manage.py makemigrations`

  将应用下的models.py文件生成一个中间文件，并保存再migrations文件夹中

- **执行迁移脚本程序 ----- 执行** `python manage.py migrate`

执行迁移程序实现迁移。将每个应用下的migrations目录的中间文件同步回数据库

字段类型

1. AutoField 根据 ID 自增长的 IntegerField 字段。通常用于主键ID，无需使用。
2. IntegerField 32位整数，可自定义选项（具体方法见视频）。
3. BooleanField 一个布尔值(true/false)字段。
4. **CharField 一个字符串字段，对小字符串和大字符串都适用。对于大量文本建议使用TextField 。必须参数：max_length,字段的最大字符数。**
5. **DateField 利用 Python 的 datetime.date 实例来表示日期。 可选参数DateField.auto_now：每一次保存对象时，Django 都会自动将该字段的值 设置为当前时间。一般用来表示 "最后修改" 时间。 可选参数DateField.auto_now_add：在第一次创建对象时，Django 自动将该字段的 值设置为当前时间，一般用来表示对象创建时间。**
6. **DateTimeField 利用 datetime.datetime 实例表示日期和时间。该字段所按受的参数和 DateField 一 样。**
7. **DecimalField 使用 Decimal 实例表示固定精度的十进制数的字段。 必须参数 DecimalField.max_digits：数字允许的最大位数 必须参数DecimalField.decimal_places：小数的最大位数**
8. EmailField 可以看做带有 email 合法性检测的CharField 。 默认max_length=75。
9. TextField 超大文本字段。
10. FileField 文件字段
11. ImageField 继承于FileField，确保是有效图

字段选项

字段选项，指定创建的列的额外信息

- 允许出现多个字段选项，多个选项之间使用，隔开
- primary_key:
  - **如果设置为True，表示该列为主键，如果指定一个字段为主键，则此数库表不会创建id字段**
- blank
  - 设置为True时，字段可以为空，设置为False时，字段是必须填写的
- null
  - 如果设置为True,表示该列值允许为空
  - 默认为False，如果选项为False，建议加入default选项来设置默认值
- default
  - **设置所在列的默认值，如果字段选项null=False建议添加此选项**
- db_index
  - 如果设置为True，表示为该列增加索引
- unique
  - **如果设置为True,表示该字段在数据库中的值必须时唯一的（不能重复出现）**
- db_column
  - 指定列的名称，如果不指定的话则采用属性名作为列名
- verbose_name
  - 设置此字段在admin界面上的显示名称

内部类 -- Meta类

- 使用内部Meta类 来给模型赋予属性，Meta类下有很多内建的类属性，可以对模型类做一些控制

```
class Book(models.Model):

    title = models.CharField('书名', max_length=50, default='')

    price = models.DecimalField('定价', max_digits=7, decimal_places=2,
default=0.0)

    info = models.CharField('信息',max_length=100,default='')

    class Meta:
        db_table = 'book'    #改变当前模型类对应的表名（修改之后需要立马同步数据库）
        verbose_name = '单数名'     #给模型对象一个易于理解的名称（单数），用于显示在/admin
管理界面中
        verbose_name--plural = '复数名'     #给模型对象一个易于理解的名称（复数），用于显
示在/admin管理界面中
```

ORM基本操作

基本操作包括增删改查，即CRUD（Create,Read,Update,Delete）

- 核心： **模型类.管理器对象**

管理器对象

每一个继承自models.Model 的模型类，都会有一个 objects 对象被同样继承下来，这个对象叫做管理器对象

数据库的增删改查可以通过模型的管理器实现

创建数据

创建数据中的每一条记录就是创建一个数据对象

方案一：

MyModel.objects.create(属性1 = 值1 , 属性2 = 值2, ……)

成功：返回创建好的实体对象

失败：抛出异常

方案二：

创建MyModel实例对象，并调用save()进行保存

```
obj - MyModel(属性=值,属性=值,...)
obj.属性=值
obj,save()       #很重要
```

查询数据

- 数据库的查询需要使用管理器对象进行
- **通过MyModel.objects 管理器方法调用查询方法**

普通查询方法	**说明**
all()	查询**全部列**记录，返回QuerySet查询对象
values('列1','列2'..)	查询**部分列**的数据并全部返回，返回的结构**内存字典**
values_list('列1','列2'..)	查询**部分列**的数据并全部返回，返回的结构**内存元组**
order_by('-列','列')	加在上面的三种方法的后面，默认升序，前面加-表示降序

条件查询方法	**说明**
get()	查询符合条件的单一记录，**只能返回一条数据，多余一条或者没有都会报错**
filter(属性1=值1,属性2=值2,...)	查询符合条件的多条记录，**多个属性在一起时为'与'**
exclude(属性1=值1,属性2=值2,...)	查询符合条件之外的全部记录

查询谓词（用于非等值查询）

- 做更灵活的条件查询时需要使用查询谓词
- 每一个查询谓词时一个独立的查询功能

- __exact ： 等值查询
  - ```
    Book.objects.filter(id__exact=1)
    ```

- **__contains ： 包含指定值（模糊查询）**
  - ```
    Book.objects.filter(name__contains='w')
    ```
- __startswith ： 以xxx开始
- __endswith ： 以xxx结束

- __gt ： 大于指定值
  - ```
    Book.objects.filter(id__gt=50)  #id>50
    ```
- __gte ： 大于等于
    ```

- __lt ： 小于
- __lte ： 小于等于

- __in ： 查找数据是否在指定范围内
  - ```
    Book.objects.filter(pub__in = ['清华大学出版社','北京航空航天大学出版社'])
    ```
- __range ： 查找数据是否在指定的区间范围内
  - ```
    Book.objects.filter(id__range=(35,50))
    ```

更新数据

- 修改**单个实体**的某些字段值的步骤：
  1. 查： 通过get()得到要修改的实体对象
  2. 改： 通过 对象.属性 的方式修改数据
  3. 保存： 通过 对象.save()保存数据
- 批量更新数据
  - 直接调用QuerySet的update(属性 = 值) 实现批量修改
  - ```
    # 实例
    books = Book.objects.filter(id__gt = 3)
    books.update(price = 0)
    ```

删除操作

- 单个数据的删除
  1. 查找查询结果对应的一个数据对象
  2. 调用这个数据对象的delete()方法实现删除

```
try :
    book = Book.objects.get(id=1)
    book.delete()
except :
    print('删除失败')
```

- 批量删除

```
try :
    books = Book.objects.filter(id__gt = 3)
    books.delete()
except :
    print('删除失败')
```
```

伪删除

- 通常不会轻易删掉，取而代之的时做伪删除，即在表中添加一个布尔字段（is_active），默认是True;执行删除时，将欲删除数据的is_active字段设置伪False
- 注意：用伪删除时，确保显示数据的地方，均加了is_active=True的过滤查询

F对象和Q对象

F对象(处理资源竞争的并发问题，同时有大量请求需要更新)

- 一个F对象代表数据库中某条记录的字段的信息
- 作用：
  - 通常是对数据库中的字段值在**不获取的情况下进行操作**
  - 用于类属性（字段）之间的比较
- 语法：

```
from django.db.models import F
F('列名')

#例一：对数据库中两个字段的值进行比较，列出哪些书的零售价高于定价
books = Book.objects.filter(market_price__gt = F('price'))
#例二：更新Book是咧中所有的零售价涨10元
Book.objects.all().update(market_price = F('market_price') + 10)
```

Q对象（处理查询结果集有负责的逻辑或，逻辑非等操作）

- 语法：

```
from django.db.models import Q
Q('列名')

#例一：查询定价低于20 或清华大学出版社的全部图书
books = Book.objects.filter(Q(price__lt = 20 )  |  Q(pub = "清华大学出版社"))
```

聚合查询和原生数据库查询

聚合查询（整表聚合，分组聚合）

- 整表聚合：不带分组的聚合查询是指导将全部数据集中统计查询
- 聚合函数【需要导入】：
  - **导入方法：** `from django.db.models import *`
  - 聚合函数： `Sum , Avg , Count , Max , Min`
- 语法： `MyModel.objects.aggregate( 结果变量名 = 聚合函数('列') )`
  - 返回结果：结果变量名和值组成的字典
  - 格式为（字典）： {"结果变量名"：值}

- 分组聚合
    1. **通过先用查询结果`MyModels.objects.values`查找查询要分组聚合的列**
    2. 通过返回的结果的QuerySet.annotate 方法分组聚合得到分组结果
    3. `QuerySet.annotate(名 = 聚合函数('列') )`

Admin后台管理层

- 创建后台管理账号 - 该账号为管理后台最高权限账号

```
python manage.py createsuperuser
```

注册自定义模型类

注册步骤：

1. **在应用app中的admin.py中导入注册要管理的模型models类**，如：

   `from .models import Book`

2. **调用 admin.site.register方法进行注册**，如：

   `addmin.site.register(自定义模型类)`

模型管理器类

- 作用：为后台管理界面添加便于操作的新功能
- 说明：后台管理器类须继承自`django.contrib.admin`里面的ModelAdmin类

- 使用方法：
    1. **在/admin.py 里定义模型管理器类**

```
class xxxManager(admin.ModelAdmin):

    # 列表页显示那些字段的列
    list_display = ['id','title','pub','price','market_price']
    # 控制list_display中的字段 哪些可以链接到修改页
    list_display_links = ['title']
    # 添加过滤器（很常用）
    list_filter = ['pub']
    # 添加搜索框[模糊查询]
    search_fields = ['title','pub']
    # 添加可在列表页编辑的字段[与list_display_links中的互斥]
    list_editable = ['price']
```

    2. **绑定注册模型管理器和模型类**

```
from django.contrib import admin
from .models import *
admin.site.register(YYYYY,XXXXXManager)      #绑定YY模型类与管理器类XXManager
```

关系映射(外键)

一对一映射

模型类创建

- 语法：`OneToOneField(类名 ， on_delete = xxx)`

```
class A(model.Model):
    pass
class B(model.Model):
    属性 = models.OneToOneField(A, on_delete = xxx)
```

- **on_delete - 级联删除**

1. **models.CASCADE 级联删除。 Django模拟MySQL约束ON DELETE CASCADE的行为，并删除包含ForeignKey的对象**
2. **models.PROTECT 抛出ProtectedError 以阻止被引用对象安定删除**
3. SET_NULL 设置ForeignKey null; 需要指定null = True
4. SET_DEFAULT 将ForeignKey设置为其默认值；必须设置ForeignKey的默认值

实例对象创建

- 无外键的模型类[Author] :
  - `author1 = Author.objects.create(name = '王老师')`
- **有外键的模型类[Wife]:**
  1. `wife1 = Wife.objects.create(name = '王夫人' ， author = author1) #关联王老师obj`
  2. `wife1 = Wife.objects.create(name = '王夫人' ， author_id = 1)    #关联王老师对应主键值`

实例对象查询

- 正向查询：**直接通过外键属性查询，称为正向查询**

```
# 通过 wife 找 author
from .models import *
wife = Wife.objects.get(name='王夫人')
print(wife.writer.name)
```

- 反向查询：没有外键属性的一方，可以通过反向属性查询到关联的另一方

- **反向关联属性为`实例对象.引用类名（小写）`,如作家的反向引用为`作家对象.wife`**
- 当反向引用不存在的时候，会触发异常

```
writer1 = Writer.objects.get(name = '王老师')
writer1.wife.name
```

一对多映射

- 一对多需要明确出具体角色，在多表上设置外键

模型类创建

A类对象可以关联多个B类对象(**ForeignKey 必须指定om_delete模式**)

```
class A(model.Model):
    pass
class B(model.Model):
    属性 = models.ForeignKey("一"的模型类, on_delete = xx)
```

实例对象创建

- 先创建一，再创建多

```
from .models import *
pub1 = Publisher.objects.create(name = '清华大学出版社')
Book.objects.create(title = 'C++',publisher = pub1)
Book.objects.create(title = 'Java',publisher = pub1)
```

实例对象查询

- 正向查询：**直接通过外键属性查询，称为正向查询**

```
# 通过 book 找 publisher
from .models import *
book = Book.objects.get(id=1)
print(book.title)
```

- 反向查询：需要用到反向属性（类名小写_set.all()）

```
pub1.book_set.all() #返回的是和pub1关联的所有书的集合,book_set其实就是objects，后面可以用ORM查询的任何操作
```

多对多

模型类创建

- 语法：在关联的两个类中的任意一个类中，增加：

```
# 属性 = models.ManyToManyField(MyModel)
class Author(models.Model):
    pass
class Book(models.Model):
    ...
    authors = models.ManyToManyField(Author)
```

创建数据

```
# 方案一 先创建 author 再关联 book
a1 = Author.objects.create(name = '吕老师')
a2 = Author.objects.create(name = '王老师')
book1 = a1.book_set.create(titile = 'Pyhton')
a2.book_set.add(book1)

# 方案二 先创建 book 再关联 author
book = Book.objects.create(title = 'Python1')
a3 = book.authors.create(name = 'guoxiaonao')
book.authors.add(a1)
```

查询数据

- 正向查询：有多对多属性的对象查另一方
- 通过Book 查询对应的所有的Author，**此时多对多属性等价于objects**

```
book.authors.all()
book,authors.filter(age__gt = 80)
```

- 反向查询：**利用反向属性**book_set

```
author.book_set.all()
author.book_set.filter()
```

cookies和session

会话

- Cookies和Session就是为了**保持会话状态**而诞生的两个**存储技术**
- Cookies存浏览器上，Session存服务器上

Cookies的使用

存储和修改

- 通过key-value（键值对）实现存储

```
HttpResponse.set_cookie(key,value = '',max_age = None, expires = None)
'''
key: cookie的名字
value: cookie的值
max_age: cookie存活时间，秒为单位
expires: 具体过期时间
当不指定max_age，expires（二选一）时，关比浏览器时此数据失效

def set_cookies(request):
    resp = HttpResponse('set cookies is ok')
    resp.set_cookie()
    return resp
'''
```

获取Cookies

```
request.COOKIES 绑定字典（dict）获取客户端的COOKIES数据
value = request.COOKIES.get('Cookies名','默认值')

def get_cookies(request):

    value = request.COOKIES.get('usename')
    return HttpResponse('value is %s' % value)
```

删除Cookies

```
HttpResponse.delete_cookie(key)
# 删除指定的key的Cookie。如果key不存在则什么也不发生
```

Session

- session是在**服务器上开辟**一段空间用于保留浏览器和服务器交互时的重要数据
- 实现方式：
    - 使用session需要在**浏览器客户端启动cookie,且在cookie中存储sessionid**
    - **每个客户端都可以在服务器端有一个独立的Session**
    - 注意：不同的请求者直接不会共享这个数据，与请求者一一对应

session 初始配置

settings.py中配置session

1. 向INSTALLED_APPS列表中添加

```
INSTALLED_APPS = [
    'django.contrib.sessions',#session启用
]

MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',#session启用
]
```

session的使用

- 和字典的形式基本一致

1. 保持session的值到服务器

```
request.session['KEY'] = VALUE
```

2. 获取session的值

```
value = request.session['KEY']
value = request.session.get('KEY',默认值)
```

3. 删除session

```
del request.session['KEY']
```

- settings.py中相关的配置项

1. **SESSION_COOKIE_AGE**

   作用：指定sessionid在cookies中保存是时常（默认是两周）

2. SESSION_EXPIRE_AT_BROWSER_CLOSE = True

   设置只要浏览器关闭时,session就会失效（默认为False）

- **注意:Django中的session数据存储在数据库中,所以使用session前需要确保已经执行过migrate**

session的问题

1. 由于是单表设计，该数据量会不断上升
2. **可以选择每晚执行`python manage.py clearsessions`，该命令可以删除以及过期的数据**

Django高级技巧

缓存

- 大量数据库查询访问很耗时，可以考虑访问内存
- 缓存的地方，数据变动频率较少

数据库缓存

- 需要在settings.py进行CACHE的配置,配置完毕需要执行`python manage.py createcachetable`

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
        'TIMEOUT': 300,      #缓存保存时间，单位秒，默认300
        'OPTIONS':{
            'MAX_ENTRIES': 300, #缓存最大数据条数
            'CULL_FREQUENCY':2, #缓存条数到达最大值时，删除1/x的缓存数据
        }
    }
}
```

本地内存缓存

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
        }
}
```

Django中使用缓存

视图函数中

- 使用装饰器

```
from django.views.decorators.cache import cache_page

@cache_page(30)
def my_view(request):
    pass
```

路由中

- 使用装饰器

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    path('foo/',cache_page(60)(my_view))
]
```

缓存API的使用

- 先引入cache对象

1. 使用cache['CACHE配置key']导入具体对象

```
from django.core.cache import caches
cache1 = caches['myalias']
cache2 = caches['myalias_2']
```

2. `from django.core.cache import cache`相当于直接引入CACHES配置项中的default项

- 使用API

1. cache.set(key,value,timeout) - **存储缓存**
    - key : 缓存的key，字符串类型
    - value: Python对象
    - timeout:缓存存储时间（s），默认是CACHES中的TIMROUT值
2. cache.get(key) - **获取缓存**
    - key :缓存的key
3. cache.add(key,value) - **存储缓存**，只在key不存在时生效
    - 返回值：True or False
4. cache.get_or_set(key,value,timeout) - 如果为获取到数据，则执行set操作
    - 返回值： value
5. cache.set_many(dict,timeout) - 批量存储缓存
    - dict : key和value的字典
    - timeout: 存储时间
6. cache.get_many(dict,timeout) - 批量获取缓存
    - dict : key和value的字典
    - timeout: 存储时间
7. **cache.delete**(key) - 删除key缓存数据
8. cache.delete_many(key_list) - 批量删除

浏览器缓存策略

强缓存

- 不会像服务器发送请求，直接从缓存中读取资源

1. 响应头 - Expires
    - 定义：存储过期时间，用来指定资源到期的时间，是否服务器端的具体时间点
2. 响应头 - Cache-Control

中间件

- 中间件以类的形式体现
- **中间件类必须继承自`django.utils.deprecation.MiddlewareMixin`类**
- 中间件类必须实现下列五个方法中的一个或者多个：

1. process_request(self,request)

   执行**路由之前**被调用，在每个请求上调用，返回None或HttpResponse对象

2. process_view(self,request,callback,callback_args,callback_kwargs)

   调用**视图之前**被调用，在每个请求上调用，返回None或HttpResponse对象

3. process_response(self,request,response)

   所有响应返回浏览器被调用，在每个请求上调用，返回HttpRespose对象

4. process_exception(self,request,exception)

   当处理过程中**抛出异常**时使用，返回一个HttpResponse对象

注册中间件

- settings.py中需要注册一下自定义的中间件

```
from django.utils.deprecation import MiddlewareMixin


class MyMiddleWare(MiddlewareMixin):

    def process_request(self,request):

        print('MyMW process_request do ---')

    def process_view(self,request,callback,callback_args,callback_kwargs):

        print('MyMW process_view do ---')

    def process_response(self,request,response):

        print('MyMW process_response do ---')
        return response

    def process_exception(self,request,exception):

        print(traceback.format_exc())
        return HttpResponse('---对不起，当前网页有点忙')
```

```
# file : settings.py
MIDDLEWARE =[
    'middleware.mymiddleware.MyMiddleWare',
]
```

- 注意：配置为数组，中间件被调用时以'先上到下'再'由下到上'的顺序调用

中间件的一个小案例（强制某个IP只能向/test开头的地址发送5次请求）

- `request.META['REMOTE_ADDR']`可以得到原创客户端的IP地址
- `request.path_info` 可以得到客户端访问的请求路由信息

中间件写法：

```
from django.utils.deprecation import MiddlewareMixin
```

```
from django.http import *
import re

class VisitLimit(MiddlewareMixin):

    visit_times = {}

    def process_request(self, request):

        ip_address = request.META['REMOTE_ADDR']
        path_url = request.path_info

        if not re.match('^/test',path_url):
            return

        times = self.visit_times.get(ip_address,0)
        print('ip',ip_address,'已经访问',times)
        self.visit_times[ip_address] = times + 1

        if times < 5:
            return
        return HttpResponse('您已经访问过' + str(times) + '次，访问被禁止')
```

CSRF攻击/防范

配置步骤：

1. settings.py中确认MIDDLEWARE中django.middleware.csrf.CsrfViewMiddleware是否打开
2. 模板中，form标签下添加如下标签 {% csrf_token %}

- 局部关闭csrf检验，用装饰器关闭对此视图的检查

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def my_view(request):
    return HttpResponse('...')
```

分页

- 分页定义：为了阅读方便，在每个页的页中只显示部分数据
- 优点：
  - 方便阅读
  - 减少数据提取量，减轻服务器压力
- Django提供了**Paginator类**可以方便的实现分页功能
- **Paginator类位于`django.core.paginator`模块中**

Paginator对象

- 负责分页数据整体的管理
- 对象的构造方法：paginator = Paginator(object_list,per_page)

    - object_list 需要分页数据的对象列表
    - per_page 每页数据个数

Paginator 属性

- count：需要分页数据的对象总数
- num_pages：分页后的页面总数
- **page_range：从1开始的range对象，用于记录当前页码数**
- per_page：每页数据的个数

Paginator 方法

paginator 对象.**page(number)**

- 参数 number为页码信息（从1开始）
- 返回当前number页对应的页信息
- 如果提供的页码不存在，抛出InvalidPage异常

page对象

- 负责具体某一页的数据的管理
- 创建对象
    - Paginator 对象的page()方法返回Page对象
    - page = paginator.page(页码)

page对象属性

- object_list：当前页上多有数据对象的列表
- **number：当前页的序号，从1开始**
- paginator：当前page对象相关的Paginator 对象

page对象方法

- **has_next()：如果有下一页返回True**
- **has_previous()：如果有上一页返回True**
- has_outher_pages()：如果有上一页或者下一页返回True
- **next_page_number()：返回下一页的页码，如果下一页不存在，抛出InvalidPage异常**
- **previous_page_number()：返回上一页的页码，如果上一页不存在，抛出InvalidPage异常**

一个分页实例

- 视图层

```python
def test_page(request):
    # path转换器(/test_page/<int:page_id>) 和
    # 查询字符串(/test_page?page=1) 都OK
    page_num = request.GET.get('page','1')
    all_data = ['a','b','c','d','e']
    # 初始化paginator
    paginator = Paginator(all_data,2)
    # 初始化 具体页码的 page 对象
    c_page = paginator.page(int(page_num))

    return render(request,'test_page.html',locals())
```

- 模板层

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>分页</title>
    <style>
        body{
            text-align: center;
        }
        table {margin: auto;}
     </style>
</head>
<body>

{% for p in c_page %}

    <p>
        {{ p }}
    </p>


{% endfor %}

{% if c_page.has_previous %}
    <a href="/test_page?page={{ c_page.previous_page_number }}">上一页</a>
{% endif %}

{% for p_num in paginator.page_range%}

    {% if p_num == c_page.num %}
        {{ p_num }}
    {% else %}
        <a href="/test_page?page={{ p_num }}">{{ p_num }}</a>
    {% endif %}
{% endfor %}

{% if c_page.has_next %}
    <a href="/test_page?page={{ c_page.next_page_number }}">下一页</a>
{% endif %}


</body>
```

```
</html>
```

CSV文件

- CSV文件可以被常见制表工具，如excel等直接进行读取
- Python提供了内建库 **-csv**;可以直接通过该库操作csv文件
- 一般生成csv文件案例如下：

```
import csv
with open('egg.csv','w',newline = '') as csvfile:
    writer = csc.write(csvfile)
    writer.writerow(['a','b','c'])
```

csv文件下载

在网站中，实现下载CSV，注意如下：

- 响应Content-Type类型需要改成 text/csv。这里告诉浏览器该文档是CSV文件，而不是HTML文件
- 响应会获得一个额外的Contebt-Disposition 标头，其中包含CSV文件的名称。它将被浏览器用于开启"另存为..."对话框

- 注意，如果要输出中文，需要单独添加响应头`response.write(codecs.BOM_UTF8)`

- 案例操作

```
from django.http import *
from django.shortcuts import render
from .models import Book
import csv,codecs

def make_csv_view(request):

    response = HttpResponse(content_type='/text/csv')
    response.write(codecs.BOM_UTF8)      #输出中文
    response['Content-Disposition'] = 'attachment;filename="mybook.csv"'
    book = Book.objects.all()
    writer = csv.writer(response)
    writer.writerow(['id','title'])
    for b in book:
        writer.writerow([b.id,b.title])
    return response
```

内建用户系统

- Django带有一个用户认证系统。它处理用户账号，组，权限以及基于cookie的用户会话
- 用户可以直接使用Django自带的用户表

基本字段

- 模型类位置 `from django.contrib.auth.models import User`

字段名	中文名
username	用户名
password	密码
email	邮箱
first_name	名
last_name	姓

字段名	中文名
is_superuser	是否是管理员账号（/admin）
is_staff	是否可以访问admin管理界面
is_active	是否是活跃用户，默认True。一般不删除用户，而是将用户的is_active设置为False
last_login	上一次登录时间
date_joined	用户创建的时间

基本模型操作

1. 创建普通用户 create_user

```
from django.contrib.auth.models import User

user = User.objects.create_user(username = '用户名',password = '密码', email = '邮箱'...)
```

2. 创建超级用户

```
from django.contrib.auth.models import User

user = User.objects.create_superuser(username = '用户名',password = '密码', email = '邮箱'...)
```

3. 删除用户

```
from django.contrib.auth.models import User

try:
    user = User.objects.get(username = '用户名')
    user.is_active = False
    user.save()
    printf("删除普通用户成功！")
except:
    printf("删除普通用户失败")
```

4. **校验密码**

```
from django.contrib.auth import authenticate

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

5. 修改密码

```
from django.contrib.auth.models import User

try:
    user = User.objects.get(username = 'Zhoues')
    user.set_password('654321')
    user.save()
    return HttpResponse("修改密码成功！")
except:
    return HttpResponse("修改密码失败！")
```

6. **登录状态的保持**

```
from django.contrib.auth import login

def login_view(request):
    user = authenticate(username = username,password = password)
    login(request,user)
```

7. 登录状态的校验

```
from django.contrib.auth.decorators import login_required

@login_required#必须登录才可以访问，未登录跳转至settings.LOGIN_URL（需要自己配置）
def index_voew(request):
    # 该视图必须为用户登录状态下才可以访问
    # 当前登录用户可通过request.user获取
    login_user = requesr.user    # 直接获取到登录用户的所有信息
```

8. 登录状态取消

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
```

内建用户表 - 拓展字段

- 继承 内建的抽象user模型类
- 步骤：
  - 添加新的应用
  - **定义模型类 继承AbstractUser**
  - **settings.py中 指明 AUTH_USER_MODEL = '应用名.类名'，注意此操作要在第一次Migrate之前进行**

文件上传

- 定义：用户可以通过浏览器将 图片等文件传至网站
- 场景：
- 用户上传头像
- 上传流程性的文档[pdf,txt等]

上传规则 - 前端[HTML]

- 文件上传必须使用POST提交方式
- 表单<form>中文件上传时**必须带有enctype = "multipart/form-data"时**才会包含文件内容数据。
- 标点中用<input type = 'file' name = 'xxx'>表情上传文件

上传规则 - 后端[Django]

- `file = request.FILES['XXX']  /   file = request.FILES.get('XXX')`

- 说明：
  - FILES的 key 对应页码中file框的 name值
  - file 绑定文件流对象
  - **file.name 文件名**
  - **file.file 文件的字节流数据**

- **需要配置 文件的访问路径和存储路径**
- 在settings.py中设置MEDIA相关配置；Django把用户上传的文件，统称为media资源

```
# file : settings.py
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR,'media')
```

- **MEDIA_URL和MEDIA_ROOT需要手动绑定**
- 步骤：主路由中添加路由

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns += static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

- 说明：等价于做了 MEDIA_URL开头的路由，Django接到该特征请求后去MEDIA_ROOT路径查找资源

解助ORM实现文件写入

- 首先在M层创建一个模型类，用于在数据库存储该文件的路径

```
# 字段：FileField(upload_to = '子目录名')
# 用于存储在绑定路径下的子目录里

class Content(models.Model):
    title = models.CharField('书名',max_length=11)
    picture = models.FileField(upload_to='picture')
```

Django发送邮件

- 业务场景：
  - 业务警告
  - 邮件验证
  - 密码找回
- 原理：
  - **给Django授权一个邮箱**
  - Django用该邮箱给对应收件人发送邮件
  - **django.core.mail 封装了 电子邮件自动发送的SMTP协议**

- 授权步骤：以QQ号为例
  - 用QQ号登录QQ邮箱并修改设置
  - 用申请到的QQ号和密码登录到 `https://mail.qq.com/`
  - 修改 QQ邮箱->设置->账户->'POP3/IMAP...服务'

Django配置

- settings.py

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST =         #腾讯QQ邮箱SMTP服务器地址
EMAIL_PORT =                 #SMTP服务器端口号
EMAIL_HOST_USER = '# 发送邮件的QQ邮箱
EMAIL_HOST_PASSWORD =    #授权码
EMAIL_USE_TLS = False    #与SMTP服务器通信时，是否启动TLS链接（安全链接）默认False
```

函数调用

```
from django.core import mail
mail.send_mail(
                subject,    # 题目
                message,    # 消息内容
                from_email  # 发送者[当前配置邮箱]
                recipient_list = ['xxx@qq.com'] # 接受者邮件列表
                )
```

中间件范例 - 邮件提示报错

```
class Exception(MiddlewareMixin):

    def process_exception(self,request,exception):

        print(traceback.format_exc())
        mail.send_mail(subject='Django报错自动发送邮件测试',
                       message = traceback.format_exc(),
                       from_email = settings.EMAIL_HOST_USER,
                       recipient_list = settings.EX_EMAIL_MEMBER
                       )

        return HttpResponse('---对不起，当前网页有点忙')
```

项目部署

基础概念

1. 在安装机器上安装和配置同版本的环境[py,数据库等]
2. django项目迁移

```
sudo scp /文件路径 <sp> root@用户ip:服务器路径
```

3. 用uWSGI替代 `python manage.py runserver` 方法启动服务器
4. 配置nginx反向代理服务器
5. 用nginx配置静态文件路径，解决静态路径问题

配置uWSGI网关

- 主要以学习配置为主

- 添加配置文件 **项目同名文件夹/uwsgi.ini**
- 如mysite1/mysite1/uwsgi.ini
- 文件以**[uwsgi]**开头，有如下配置项：

1. 套接字方式的IP地址：端口号【此模式需要有nginx】
    - **socket=127.0.0.1:8000**
2. Http通信方式的IP地址：端口号
    - **http=127.0.0.1:8000**
3. 项目当前工作目录(绝对路径)
    - **chdir=/home/..../my_project**
4. 项目中wsgi.py文件的目录，相对于当前工作目录（相对路径，相对于上方的绝对路径）
    - **wsgi-file=my_project/wsgi.py**
5. 进程个数
    - process=4
6. 每个进程的线程个数
    - threads=2
7. 服务的pid记录文件
    - **pidfile=uwsgi.pid**
8. 服务的日志文件位置
    - **daemonize=uwsgi.log**
9. 开启主进程管理模式
    - **master=true**

- 特殊说明：Django的settings.py需要做如下配置
    1. **DEBUG=True改成DEBUG=False**
    2. **ALLOWED_HOSTS=[]改成ALLOWED_HOSTS=['网站域名']或者['服务监听的ip地址']**

具体流程：

1. 在项目的根目录下面打开终端，输入ls指令看看是否是项目根目录（有settings.py）
2. 输入指令 touch uwsgi.ini 创建uwsgi.ini文件
3. 打开之后编写配置项
4. 对于chdir，可以对项目右键copy path直接获取

uWSGI的运行管理

启动uwsgi

cd到uWSGI配置文件所在目录

uwsgi --ini uwsgi.ini

停止uwsgi

cd 到 uWSGI配置文件所在目录

uwsgi --stop uwsgi.pid

查看uwsgi的状态

ps aux|grep 'uwsgi'

强制杀死进程

sudo kill -9 状态码

配置nginx

- 修改nignx的配置文件/etc/nginx/sites-enabled/default;
- sudo vim该文件(先 sudo apt-get install vim-gtk), i进入修改，esc退出修改，:wq保存

```
# 在server节点下添加新的location项，指向uwsgi的ip与端口
server{
    ...
    location / {
        uwsgi_pass 127.0.0.1:8000; # 重定向到127.0.01的8000端口
        include /etc/nginx/uwsgi_params; # 将所有的参数转到uwsgi下
    }
    ...
}
```

启动/停止

```
$ sudo /etc/init.d/nginx start|stop|restart|status
 或
$ sudo service nginx start|stop|restart|status

启动 - sudo /etc/init.d/nginx start
停止 - sudo /etc/init.d/nginx stop
重启 - sudo /etc/init.d/nginx restart
# 注意：nginx配置只要修改，就需要进行重启，否则配置不生效
查看是否成功启动nginx - sudo nginx -t
```

修改 uWSGI配置

uWSGI需要以 socket 模式启动，**注意重启**

样例:

```
[uwsgi]
# 去掉如下
# http=127.0.0.1:8000
# 改为
socket=127.0.0.1:8000
```

常见问题排查

排查问题宗旨 -> 看日志！看日志！

nginx 日志位置(**同名目录下打开终端**):
  异常信息： /var/log/nginx/error.log
  正常访问信息： /var/log/nginx/access.log

uwsgi 日志位置:
  项目同名目录下， uwsgi.log

1. 访问127.0.0.1:80地址，502响应

   502响应 代表nginx反向代理配置成功，但是对应的uWSGI未启动

2. 访问127.0.0.1:80/url 404响应

   1. 路由的确不在django配置中
   2. nginx配置错误，未禁掉try_files

nginx静态文件配置

静态文件配置步骤:

1. 创建新路径文件夹 - 主要存放Django所有静态文件 如： /home/..../tedu_note_static
2. 在Django settings.py中添加新配置

   ```
   STATIC_ROOT = '/home/..../tedu_note_static/static'
   # 注意 此配置路径为  存放所有正式环境中需要的静态文件
   ```

3. 进入项目，执行**python3 manage.py collectstatic** 执行该命令后，Django将项目所有的静态文件复制到STATIC_ROOT中，包括Django内建的静态文件
4. Nginx配置中添加新配置

```
# /etc/nginx/sites-enabled/default;
# 新添加location /static 路由配置，重定向到指定的第一步创建的路径即可
server{
    ...
    location /static {
        # root 第一步传教文件夹的绝对路径
        root /home/..../项目名_static;
    }

}
```

部署实践

```
# 运行uwsgi
uwsgi --http :8000 --wsgi-file test.py
# 杀掉所有正在运行的uwsgi
ps -aux | grep uwsgi |awk '{print $2}'|xargs kill -9
```