django+mysql的web项目

工具准备：

python，pycharm，django，mysql，git全部安装并能正常使用

搭建环境：

1. 创建一个文件夹，cmdcd到文件夹创建虚拟环境：python -m venv wlenv（名字）
2. 激活虚拟环境：wlenv\Scripts\activate
3. 安装django：pip install django==版本号
4. 创建django项目：django-admin startporject worklog（名字）
5. 同步数据库：python manage.py migrate
6. 测试django环境：python manage.py runserver（可选ip+端口号）
7. 切库，在mysql创建一个数据库eg：worklog\_db设置字符集为utf-8
8. 切库，安装mysqlclient：pip install mysqlclient
9. 切库，配置settings文件：在DATABASES里将engine数据库改为mysql，填上数据 库名字，用户名，密码，host
10. 切库，同步数据库：python manage.py migrate
11. 切库，查看数据库中是否创建了表格，runserver服务查看是否切库成功
12. 创建app：python manage.py startapp worklogweb
13. 配置settings文件，把新创建的APP加入INSTALLED\_APP中
14. 修改语言，settings文件里最后LANGUAGE\_CODE:’zh-Hans’
15. 修改时区，settings文件最后TIME\_ZONE:中国范围内有Asia/Shanghai, Asia/Harbin, Asia/Chongqing, Asia/Urumqi, Asia/Kashgar

web类项目概述

web标准框架：MVC（model，view，controller）

model 模型，控制数据库表结构

view 视图，负责具体前端输出的内容，eg颜色，文字，表单

controller 控制器，具体负责实现的功能，eg增删改查，加减乘除

Router路由

用户访问地址的地址，通过router映射到controller的功能

Template渲染模板

用户将要看的页面，html

django框架：MTV（model，template，view）对应MVC，只是名字不同



用户发出访问请求—>django router（URL）将请求匹配path路径到views里的具体函数—>view根据被匹配的函数执行代码—>可能执行model调取数据库内的数据—>将所有数据渲染成模板发送到用户的浏览器展示

项目文件

urls.py

HTTP请求进入django时有限调用该文件，文件里包含所有路由，第一个参数是访问的地址，第二个参数是要调用的views.py里的映射文件。

settings.py

项目配置文件

wsgi.py

正式启用django项目时配置该网关文件

views.py

文件包含调用函数，让程序调用template里的html页面渲染并发送

setting.py

公有配置

BASE\_DIR=os.path(...)项目绝对路径，注册使用别的文件夹或文件时会用到

debug

启动模式，True为调试模式

1. 检测代码改动后立刻重启服务
2. 报错页面

False为正式模式

ALLOWED\_HOSTS=[ ]

请求头HOST头，只接受请求头包含其中的请求，有过滤作用

局域网内其他主机访问django项目服务

1. 启动方式为

python manage.py runserver 0.0.0.0:8000 （四个0代表服务器所有网卡IP都能被访问）

1. 添加服务器内网IP至ALLOWED\_HOSTS=[内网IP]

INSTALLED\_APPS

配置django应用

MIDDLEWARE

中间件，高级应用

ROOT\_URLCONF

配置主路由文件

TIME\_ZONE

时区设置

USA\_TZ

True时存入数据库的时间为美国时间，但是调用时显示会根据TIME\_ZONE修改，False时存入的时间和TIME\_ZONE一致

自定义配置

1. 大写
2. 个性化，不要与公有配置重名

URL

定义：统一资源定位符

作用：用来表示互联网上某个资源的地址

一般语法格式为：protocol://hostname[:port]/path[?query][#fragment] [ ]内可省略，共6大部分。

1. protocol（协议）指使用的什么协议
2. http通过HTTP访问该资源
3. https通过安全的HTTPS访问该资源
4. file资源是本地计算机上的文件。格式file:///
5. bostname（主机名）是指存放资源的服务器的域名系统（DNS）主机名、域名或IP
6. port（端口号）

1）整数、可选（范围为1024—65535，用户可用的端口号）

1. path（路由地址）

由0个或多个“/”符号隔开的字符串，一般用来表示主机上的一个目录或文件地址。路由地址决定了服务器端如何处理这个请求

1. query（查询）以问号开头，问号后都是查询字符串

可选，用于给动态网页传递参数，可有多个参数，用“&”符号隔开，每个参数名和值用“=”隔开

1. fragment（信息片段，锚点）以#号开始，类似书签，前端技术，输入带锚点的地址浏览器将直接定位到锚点所在地方
2. 相对地址开头加 / 表示从根地址起始，不加则从当前地址起始

处理URL请求

eg：浏览器地址栏输入http://127.0.0.1:8000/page/2003/

1. django从settings.py文件中根据ROOT\_URLCONF找到主路由，默认情况下该文件在项目同名目录下的urls.py
2. django加载主路由文件中的urlpatterns变量：一个包含很多path的数组
3. 依次匹配urlpatterns中的path，匹配到第一个合适的path后中断后续匹配
4. 匹配成功—调用对应视图函数处理请求，返回响应
5. 匹配失败—返回404响应

视图函数

用于接收浏览器请求（HttpResquest对象）并通过HttpResquest对象返回响应的函数。此函数可以接收浏览器请求并根据业务逻辑返回响应的内容给浏览器

语法

def XXX\_view（request[,其他参数...]）:

return HttpResponse对象

locals( )函数

将视图函数中的数据打包成字典传递，单个对象也会被打包成字典

路由配置

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，匹配请求路径
2. views：指定路径所对应的试图处理函数的名称
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

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

eg：path(‘page’/<int:page>’, views.xxx)

转换器类型：

1. str匹配除了/之外的非空字符串
2. int匹配0或任何正整数，返回一个int
3. 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
4. path匹配非空字段，包括路径分隔符

views.py里的函数要指定自定义的转换器的名字，相当于传参进来

eg：def pagex(request, page):

re\_path（）函数（精准匹配）

在url的匹配过程中可以施工正则表达式

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传参的方式传递给视图函数

请求和响应

请求是指浏览器通过HTTP协议发送给服务器端的数据

响应是指服务器端接收到请求后做相应的处理后再回复给浏览器端的数据



HTTP1.0定义了三种请求方法：GET、POST、HEAD（开发最常用网民用不到）

HTTP1.1新增了五种请求方法：OPTIONS、PUT、DELETE、TRACE、CONNECT

GET和POST传递的都是QueryDict类似于字典的对象





put：更新

delete：删除

connect：翻墙用

django处理请求的流程

1. 请求在django中实则就是视图函数的第一个参数，即HttpRequest对象
2. django接收到http协议的请求后，会根据请求数据报文创建HttpRequest对象
3. HttpRequest对象通过属性描述了请求的所有相关信息





django响应



HTTP状态码的英文为HTTP Status Code

200：请求成功

301：永久重定向—资源（网页等）被永久转移到其他URL

302：临时重定向（登录即临时重定向）

404：请求的资源（网页等）不存在

500：内部服务器错误（一定是代码有错）

HTTP状态码共分为5类

1\*\*：信息，服务器收到请求，需要请求者继续执行操作

2\*\*：成功，操作被成功接收并处理

3\*\*：重定向，许哟进一步的操作以完成请求

4\*\*：客户端错误，请求包含语法错误或无法完成请求

5\*\*：服务器错误，服务器在处理请求的过程中发生了错误

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

常用的content\_type：

1. text/html（默认的，html文件）
2. text/plain（纯文本）
3. text/css（css文件）
4. text/javascript（js文件）
5. multipart/form-data（文件提交）
6. application/json（json传输）
7. application/xml（xml文件）

HttpResponse子类



重定向

1. xx=redirect(‘url’) 通过urls.py重定向到views.py的函数

return xx

1. xx=HttpResponseRedirect(‘url’) 通过urls.py重定向到views.py的函数

return xx

GET和POST请求

统一由视图函数接收请求，通过判断request.method区分具体的请求动作

样例：

if request.method == ‘GET’:

处理GET请求的代码块

elif request.method == ‘POST’:

处理POST请求的代码块（处理用户提交的数据）

elif XXXXX == ‘XXXX’:

......

GET，一般用于向服务器获取数据

能够产生GET请求的场景：

1. 浏览器地址栏中输入URL，回车后
2. <a href=”地址?参数=值&参数=值”>
3. form表单中的method为GET

GET请求中，如果有数据需要传递给服务器，通常用查询字符串Query String 注：不要传递敏感数据

url格式：xxx?参数1=值1&参数2=值2...

服务器端接收参数，获取客户端请求GET提交的数据：

request.GET[‘参数名’]

request.GET.get(‘参数名’, ‘默认值’)

request.GET.getlist(‘参数名’)

单个key可以对应多个value，用数组盛放，用get取只取数组中的最后一个值，用getlist拿能拿到全部值（直接拿一个数组）。

典型应用场景：问卷调查的多选

浏览器传递非敏感数据给服务器方式

1. 通过url传递
2. 通过GET请求的查询字符串Query String传递

POST，一般向服务器提交大量/隐私数据

客户端通过表单等POST请求将数据传递给服务器

<form method = ‘POST’ action = “/login”>

姓名：<input type = “text” name = ”username”>

<input type = ‘submit’ value = ’登录’>

</form>

action：指明form表单将数据发给谁，浏览器会将ip和action里的值组合起来作为地址发送数据

name：传递数据时作为key名

服务器端接收参数，通过request.method来判断是否为post请求

if request.method == ‘POST’:

处理POST请求的代码块

else:

...

request.POST[‘参数名’]

request.POST.get(‘参数名’, ‘’)

request.PSOT.getlist(‘参数名’)

取消csrf验证，否则django将会拒绝客户端发来的POST请求，报403

注释掉settings.py中MIDDLEWARE中的CsrfViewsMiddleWare的中间件

MTV模式

T：模板

模板层（Templates）（html页面）

1. 模板是可以根据字典数据动态变化的html网页
2. 模板可以根据视图中传递的字典数据动态生成相应的html网页

模板配置

1. 创建模板文件夹<项目名>/templates
2. 在settings.py中TEMPATES配置项
3. BACKEND：指定模板引擎
4. DIRS：模板的搜索目录（可以是一个或多个）
5. APP\_DIRS：是否要在应用中的templates文件夹中搜索模板文件
6. OPTIONS：有关模板的选项
7. 配置中需要修改的部分

设置DIRS - ‘DIRS’: [os. path. join(BASE\_DIR, ‘templates’)],

模板加载方式

1. 通过loader获取模板，通过HttpResponse进行响应

在视图函数中

from django.template import loader

1. 通过loader加载模板

t=loader.get\_template(“模板文件夹”)

1. 将t转换为html字符串

html=t.render(字典数据)

1. 用响应对象将转换的字符串内容返回给浏览器

return HttpResponse(html)

1. 使用render()直接加载并响应模板

在views.py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（dic）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)

1. 模板中，可以用{{ 变量名 }}的语法调用视图传进来的变量

模板的变量

能传递到模板中的数据类型

str-字符串 int-整型

list-数组 tuple-元组

dict-字典 func-方法

obj-类实例化的对象

在模板中使用变量语法

{{ 变量名 }}

{{ 变量名.index }} 调用list容器中的某一个值

{{ 变量名.key }} 调用字典中某个键的值

{{ 对象.方法 }} 调用对象的方法

{{ 函数名 }} 调用函数

模板（html）标签

将一些服务器端的功能嵌入到模板（html）中，例如流程控制（for循环，if控制）等

标签语法

{% 标签 %}

...

{% 结束标签 %}

if标签

{% if 条件表达式1 %}

...

{% elif 条件表达式2 %}

...

{% else %}

...

{% endif %}

if标签配和selected可以实现记录用户上次选择的功能，视频p10 33min

for标签

{% 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 %}

内置变量-forloop

变量 描述

forloop.counter 循环的当前迭代（从1开始索引）

forloop.counter0 循环的当前迭代（从0开始索引）

forloop.revcounter counter值得倒序

forloop.revcounter0 revcounter值的倒序

forloop.first 如果这是第一次通过循环，则为真

forloop.last 如果这是最后一次循环，则为真

forloop.parentloop 当前嵌套循环，parentloop表示外层循环

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’... }}

常用过滤器：

lower 将字符串全部转换为小写

upper 将字符串全部转换为大写

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’ 如果字符串字符多于指定的字符数量，那么会被截断。截 断的字符串将以可翻译的省略号序列(“...”)结尾

模板层继承

模板继承可以使父模板的内容重用，子模板直接继承父模板的全部内容并可以覆盖父模板中相应的块

语法—父模板中：

1. 定义父模板中的块block标签
2. 标识出哪些在子模块中时允许被修改的
3. block标签：在父模板中定义，可以在子模版中覆盖

语法—子模板

1. 继承父模板extends标签（写在模板文件第一行）

例如{% extends ‘base.html’ %}

1. 子模板重写父模板中的内容块

{% block block\_name %}

子模板块用来覆盖父模板中 block\_name 块的内容

{% endblock block\_name %}

重写覆盖规则

1. 不重写，将按照父模板的效果显示
2. 重写，则按照重写的效果显示

注：子模板继承不了父模板的变量

url地址

绝对地址：

<http://127.0.0.1:8000/page/1>

相对地址：

1. ’/page/1’ / 开头的相对地址，浏览器会把当前地址栏里的协议，ip和端口加上这个地址，作为最终访问地址。

eg：当前页面地址栏为http://127.0.0.1:8000/page/3，当前相对地址结果为http://127.0.0.1:8000/page/1

1. ‘page/1’ 没有 / 开头相对地址，浏览器会根据当前url的最后一个/之前的内容加上该相对地址作为最终访问地址。

eg：当前页面地址栏为http://127.0.0.1:8000/topic/detail，当前相对地址结果为http://127.0.0.1:8000/topic/page/1

url反向解析

指在视图或模板中，用path定义的名称来动态查找或计算出相应的路由

path函数的语法

path(route, views, name=”别名”)

eg：path(‘page’,views.page\_view, name=”page\_url” )

根据path中的name关键字传参给url定义一个唯一的名字，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’ %}

eg：{% url ‘pagen’ ‘400’ %} 位置传参

{% url ‘person’ age=’18’ name=’xxx’ %} 关键字传参

在视图函数中可调用django中的reverse方法进行反向解析

from django.urls import reverse

reverse(‘别名’, args=[ ], kwargs={ }) args为位置传参 kwargs为关键字传参

eg：reverse(‘pagen’, args=[300])

reverse(‘person’, kwargs={‘name’ : ‘xxx’, ‘age’ : 18 })

静态文件

1. 配置settings.py中静态文件访问路径

TEMPLATE的DIRS中添加os.path.join(BASE\_DIR, ‘templates’)

SRTATIC\_URL=’/static/’

1. 配置settings.py中存储路径STATICFILES\_DIRS

STATICFILES\_DIRS保存的时静态文件在服务器端的存储位置

STATICFILES\_DIRS=(os.path.join(BASE\_DIR, “static”),)

模板中访问静态文件—img标签为例

1. 直接拼接访问路径

<img src=”/static/images/lena.jpg”>

<img src=”http://127.0.0.1:8000/static/images/lena.jpg”>

1. 通过{% static %}标签访问静态文件（推荐）
2. 加载static：{% load static %}，在TEMPLATE的OPTIONS中添加'builtins':['django.templatetags.static']
3. 使用静态资源：{% static ‘静态资源路径’ %}
4. eg：<img src=”{% static ‘images/lena.jpg’ %}”>

标签访问静态文件更动态

应用层的静态文件可以放在最外层static里的同app名的文件夹中，这样结构简洁，逻辑清晰。也可以每个app里创建一个static，但是比较混乱

应用

应用在django项目中是一个独立的业务模块，可包含自己的路由，视图，模板，模型

创建应用：python manage.py startapp xxx 创建应用文件夹

配置应用：在settings.py的INSTALLED\_APPS列表中配置安装此应用

文件结构（应用下）

migrations文件夹：数据库迁移用

admin.py后台管理文件，写相关代码让非技术人员可以增删改查

apps.py应用相关配置

models.py模型层入口，DB相关

test.py测试

views.py视图

分布式路由

主路由配置文件（urls.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

1. 主路由中调用include函数

include(‘app名字.url模块’)

作用：用于将当前路由转到各个引用的路由配置文件的urlpatterns进行分布式处理

1. 应用下配置urls.py，手动创建urls.py，内容结构和主路由一致

应用下的模板

1. 应用下手动创建tempates文件夹
2. settings.py中开启应用模板功能

TEMPLATE配置项中的’APP\_DIRS’值为True即可

应用下和外层都存在templates文件夹时，django的查找规则为

1. 优先查找外层templates目录下的模板
2. 按INSTALLED\_APPS配置下的应用顺序逐层查找

问题：会出现外层html文件截胡内层同名html的情况

解决：在内层templates文件夹下创建与app同名的文件夹，将html文件放入，并修改app下的views.py内函数的参数：xxx.html改为app/xxx.html

M：模型层

mysql配置

1. 在虚拟环境中pip install mysqlclient
2. 在DATABASES中写入mysql相关配置信息：engine，name，user，password，host

数据库迁移

迁移是django同步使用者对模型所做更改（添加字段，删除模型等）到数据库模式的方式

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，并保存在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目录中的中间文件同步到数据库

模型

是一个python类，由django.db.models.Model派生出的子类

1. 一个模型类代表数据库中的一张数据表
2. 模型类中每一个类属性都代表数据库中的一个字段
3. 模型时数据交互的接口，时表示和操作数据库的方法和方式
4. 新增字段必须设置默认值

模型类创建

from django.db import models

class 模型类名(model.Model):

字段名=models.字段类型(字段选项)

模型创建总流程

1. 创建应用
2. 在应用下的models.py中编写模型类
3. 迁移同步数据库 makemigrations&migrate

注：任何关于表结构的修改，务必在对应模型类上修改

1. 模型类中添加对应类属性
2. 执行数据库迁移

字段类型（models.字段类型）

BooleanField()

数据库类型：tinyint(1)

编程语言中：True或False

数据库中：1或0

CharField

数据库类型：varchar

注：必须要指定max\_length

DateField

数据库类型：date

作用：表示日期

参数：

1. auto\_now：每次保存对象时，自动设置该字段为当前时间（取值True/False）
2. auto\_now\_add：当对象第一次被创建时自动设置当前时间（取值True/False）
3. default：设置当前时间（取值：字符串格式时间如：’2021-6-19’）

注：以上三个参数只能三选一

DateTimeField()

数据库类型：datetime(6)

作用：表示日期和时间

参数同DateField

FloatField()

数据库类型double

编程语言中和数据库中都使用小数表示值

DecimalField()

数据库类型：decimal(x, y)

编程语言中：使用小数表示该列的值

数据库中：使用小数

参数：

1. max\_digits：位数总数，包括小数点后的位数。该值必须大于等于decimal\_places
2. decimal\_places：小数点后的数字数量

EmailField()

数据库类型：varchar

编程语言和数据库中使用字符串

实现：django模型中内置了一些正则表达式来检查用户输入的邮箱格式正不正确

IntegerField()

数据库类型：int

编程语言和数据库中使用整数

ImageField()

数据库类型：varchar(100)

作用：数据库中保存的是图片路径

编程语言中和数据库中使用字符串

TextField()

数据库类型：longtext

作用：表示不定长的字符数据

字段选项

创建的列的额外的信息

多个字段选项之间使用 , 隔开

修改字段选项后要执行迁移操作makemigrations&migrate

primary\_ley

主键，设置为True后表示该列为主键，用户自定义主键列后django不会创建 id列作为主键

blank

设置为True后，字段可以为空，设置为False后，字段必须填写。控制admin 后台的选项，不是mysql里的null

null

设置为True后，表示该列值允许为空。默认为False，若设置为False建议配 合default选项来设置默认值。控制mysql里的值是否可以为空

default

设置所在列的默认值。新增字段必须设置默认值

db\_index

设置为True后，表示为该列增加索引

unique

设置为True后，表示该字段在数据库中的值必须是唯一的（不能出现重复的）

db\_column

指定列的名称，如果不指定则采用属性名作为列名

verbose\_name（即字段函数中的第一个字符串参数）

设置此字段在admin界面上的显示名称

max\_length

设置最大字符数

模型类—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，可对模型类做一些控制

class ...(models.Model):

....

class Meta:

db\_table=’更改表的名字’

类属性：

db\_table=’数据表名’

设置数据表名，设置完需要更新数据库

verbose\_name=’xxx’

设置数据表在admin中的显示名

verbose\_name\_plural=’xxx’

设置数据的表名并且在admin中为单数显示

常见问题解决

1. 插入的列没设定default

解决：推荐在models.py中添加相应代码

1. migration文件混乱导致的错误

解决： 1>删除migrations里所有的000x\_xxxx.py（\_init\_.py除外）

1. 删除数据库：sql>drop database xxx;
2. 重新创建数据库：sql>create database xxx default charset....;
3. 重新生成migrations里所有的000?\_xxxx.py
4. 重新更新数据库：python manage.py migrate
5. migrate报错：该表不存在

解决： 1>删除migrations里的000x\_xxxx.py

1. 删除数据库django\_migrations表内相关app的migrate记录
2. 关闭runserver界面
3. 执行migrate

视频p18

ORM

object relatinal mapping 即对象关系映射，是一种程序技术，允许使用类和对象对数据库进行操作，从而避免通过SQL语句操作数据库

作用：

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

优点：实现了数据模型与数据的解耦，屏蔽了不同数据库操作上的差异

缺点：有性能损失



ORM操作

基本操作包括增删改查（CRUD）增加Create、读取查询Read、更新Update、删除Delete

ORM CRUD核心：模型类.管理器对象

每个继承自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

class Mymodel(mdoels.Model):

Mymodel.objects.create(...)

django shell

django提供的交互式操作项目，能够子交互模式用项目工程的代码执行相应的操作，利用django shell可以代替编写view的代码来惊醒直接操作

注：项目代码发生变化时，重新进入django shell

启动方式：python manage.py shell

增Create

方案1

赋值写入库一并完成

Mymodel.objects.create(属性1=值1, 属性2=值2...)

成功：返回实体对象

失败：报错

方案2

创建Mymodel实例对象，并调用save( )进行保存，先赋值后入库

obj=Mymodel(属性=值, 属性=值) 属性可填可空，可后期添加

obj.属性=值 后期添加的属性和值

obj.save( )

1. from worklog\_web.models import Userinfo
2. Userinfo.objects.create(....)

查Read

使用管理器对象进行

通过Mymodel.objects管理器方法调用查询方法

返回值的类型及用法

返回对象分为两类：单一实例object（get方法）、QuerySet实例组（典型为filter）

QuerySet实例组包含多个实例对象，想要取用单个实例的属性值需要从实例组中取 出实例对象：实例组名.索引.属性 即单个实例的属性值

all() 查询全部记录，返回QuerySet查询对象

get() 查询符合条件的单条记录

filter() 查询复合条件的多条纪录

exclude() 查询扶额和条件之外的全部记录

all( )

用法：Mymodel.objects.all()

作用：查询Mymodel实体中的所有数据

等同于select \* from tabel

返回值：QuerySet容器内存对象（类似数组），内部存放Mymodel实例

def \_str\_(self):

可以在模型中定义\_str\_方法，自定义QuerySet中的输出格式

eg：def \_str\_(self):

return ‘%s\_%s\_%s\_%s’%(self.title, self.price, self.pub,....)

django shell中可得到如下显示输出:

<QuerySet [ <xxx: python\_20.00\_清华大学出版社\_....>, .....]>

values(‘列1’, ‘列2’...)

用法：Mymodel.objects.values(...)

作用：查询部分列的数据并返回

等同于select 列1, 列2 from xxx

返回值：QuerySet容器内存字典，每个字典代表一条数据，格式为：{‘列1’: 值1, ‘列2’: 值2}, {‘列1’: 值2, ‘列2’: 值2}

调用：html页面中，{{ 单个字典 . 键名(列名) }}

values\_list(‘列1’, ‘列2’...)

用法：Mymodel.objects.values\_list(...)

作用：查询部分列的数据并返回元组形式的结果

等同于select 列1, 列2 from xxx

返回值：QuerySet容器，内部存放元组，会将查询出来的数据封装到元组中，再封装到QuerySet中

调用：html页面中，{{ 单个元组 . 索引 }}

order\_by()

用法：Mymodel.objects.order\_by(‘-列’, ‘列’)

作用：与all()方法不同，他会用SQL语句的order by子句对查询结果进行根据某个字段选择性的进行排序

顺序：默认按照升序排序，降序需要再列前加’ - ’来表示

本质：为all().order\_by()

query不带括号

用法：数据对象.query，前提必须是QuerySet对象

作用：展示数据对象所执行的SQL语句

条件查询

filter(条件)

语法：Mymodel.objects.filter(属性1=值1, 属性2=值2, ....)

作用：返回包含此条件的全部数据集

返回值：QuerySet容器对象，内部存放Mymodel实例

说明：当多个属性在一起时为and关系

样例：

#查询所在科室是信息科的用户

from worklog\_web impor Userinfo

users=Userinfo.object.filter(keshi=’信息科’)

for user in users:

print(‘所在科室：’, user.keshi)

exclude(条件)

语法：Mymodel.objects.exclude(条件)

作用：返回不包含此条件的全部数据集，与filter对立

get(条件)

语法：Mymodel.objects.get(条件)

作用：返回满足条件的唯一一条数据（object实例）

说明：该方法只能返回一条数据

若查询结果多于一条则报错MultipleObjectReturned异常

查询结果没有数据也报错DoesNotExist异常

查询谓词

定义：做更灵活的条件查询时需要使用查询谓词

说明：每一个查询谓词是一个独立的查询功能

字段\_\_exact：等值匹配 （双下划线）

示例：

xxxx.objects.filter(id\_\_exact=1)

\_\_contains：包含指定值，等同于SQL中like ‘%w%’

\_\_startswith：以xxx开始

\_\_endswith：以xxx结束

\_\_gt：大于指定值

\_\_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

样例：xxx.objects.filter(country\_\_in=[‘中国’, ’美国’, ....]) in后跟列表

\_\_range：查找数据是否在指定的区间范围内

样例：xxx.objects.filter(age\_\_range=(20,30)) range后跟元组

\_\_year：查找包含在指定年内的数据

\_\_month：查找包含在指定月份内的数据

\_\_day：查找包含在指定日子内的数据

改Update

方案一：单条数据修改，分三步

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 a.name=’xxx’
3. 保存：通过对象.save()保存数据 a.save()

方案二：调用QuerySet的update(属性=值)实现批量修改

样例：

1. abc=xxx.objects.filter(....)
2. abc.update(xxx=xx)

删Delete

单条删除：

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

try:

abc=xxx.objects.get(条件)

abc.delete( )

except:

print(‘删除失败’)

批量删除：

1. 查找查询结果集中满足条件的全部QuerySet查询集合对象
2. 调用查询集合对象的delete()方法实现删除

abc=xxx.objects.filter(条件)

abc.delete( )

伪删除：

在表中添加布尔型字段is\_active，默认是True，执行删除时，将该字段设置为 False

注：用伪删除时，确保展示数据的地方都添加了is\_active=True的过滤

F对象和Q对象

F对象

一个F对象代表数据库中某条记录的字段的信息，具有行锁，同时间大量 请求更新不会出错，防止资源竞争。详解见视频P23

作用：

1. 通常是对数据库中的字段值在不获取的情况下进行操作
2. 用于类属性（字段）之间的比较

语法：

from django.db.models import F

F(‘列名’)

示例：更新Book实例中所有的零售价涨10元

Book.objects.all( ).update(market\_price=F(‘market\_price’)+10)

Q对象

当在获取查询结果集使用复杂的逻辑或 |、逻辑非 ~等操作时可以借助Q对象进行操作

示例：想找出定价低于20元或清华大学出版社的全部书

from django.db.models import Q

Book.objects.filter(Q(price\_\_lt=20) | Q(pub=”清华大学出版社”))

作用：在条件中用来实现除and(&)以外的or( | )或not(~)操作，可以灵活组合

运算符：

& 与

| 或

~ 非

语法：

from django.db.models import Q

Q(条件1) | Q(条件2) 条件1成立或条件2成立

Q(条件1) & Q(条件2) 条件1和条件2同时成立

Q(条件1) &~ Q(条件2) 条件1成立且条件2不成立

拼接：

Q对象可以进行拼接操作

1. 定义一个Q对象
2. 与+=类似，逻辑运算符和等号

q=Q( )

q &= Q(条件)

q |= Q(条件)

q ~= Q(条件)

聚合查询

聚合查询是指对一个数据表中的一个字段的数据进行部分或全部的统计查询，比 如查询bookstore\_book数据表中的全部书的平均价格，查询所有书的总个数等，都要使 用聚合查询，聚合查询又分为：整表聚合、分组聚合

整表聚合：

将整表全部数据进行集中统计查询，聚合函数需要导入

from django.db.models import \*

Sum、Avg、Count、Max、Min （聚合函数）

语法：

Mymodel.objects.aggregate(结果变量名=聚合函数(‘列’))

返回值：

结果变量名和值组成的字典

格式：

{“结果变量名”: 值}

分组聚合

分组聚合是指通过计算查询结果中每一个对像所关联的对象集合，从而得出总 计值（也可以是平均值或总和），即为查询集的每一项生成聚合

语法：

QuerySet.annotate(结果变量名=聚合函数(‘列’))

返回值:

QuerySet

1. 通过先用查询结果Mymodels.objects.values查找查询要分组聚合的列

Mymodel.objects.values(‘列1’, ‘列2’)

1. 通过返回结果的QuerySet.annotate方法分组聚合得到分组结果

QuerySet.annotate(结果名=聚合函数(‘列’))

原生数据库操作（不推荐使用）

django也支持直接用SQL语句的方式通信数据库

raw()（不推荐使用）

查询：使用Mymodel.objects.raw()进行数据库查询操作，只用于查询

语法：Mymodel.objects.raw(SQL语句，拼接参数)

返回值：RawQuerySet集合对象（只支持基础操作，比如循环）

示例：

books=models.Book.objects.raw(‘select \* from bookstore\_book’)

for book in books:

print(book)

注：raw()使用不当会有被SQL注入的风险，详解见视频P24

错误示范：s1=Book .objects.raw(‘select \* from bookstore\_book where id=%s’ %(‘1 or 1=1’))

正确示范：s1=Book.objects.raw(‘select ......id=%s’, [‘1 or 1=1’]) 将参数写在raw()的第二个参数中

cursor（不推荐使用）

完全跨过模型类操作数据库：查询/更新/删除

1. 导入cursor所在的包

from django.db import connection

1. 用创建cursor类的构造函数创建cursor对象，再使用cursor对象，为保证在出现异常时能释放cursor资源，通常使用with语句进行创建操作

from django.db import connection

with connection.cursor() as cur:

cur.execute(‘执行SQL语句’, ‘拼接参数’)

admin管理后台

django提供了比较完善的后台管理数据库的接口，可供开发过程中调用和测试使用

django后搜集所有已注册的模型类，为这些模型类提供数据管理界面，供开发者使用

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

输入用户名（账号）

输入邮箱

输入密码

再次输入密码

创建成功

注册自定义模型类

若要自己定义的模型类也能在/admin后台管理界中显示和管理，需要将自己的类注册到后台管理界面

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

1. 调用admin.site.register方法进行注册

如：admin.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里的ModelAdmin类

使用方法：

1. 在app的admin.py里定义模型管理类

class xxxManager(admin.ModelAdmin):

....

1. 绑定注册模型管理器和模型类

from django.contrib import admin

from .models import \*

admin.site.register(yyy, xxxManager) # 绑定yyy模型类与管理器类xxxManager

样例：

class UserinfoManager(admin,ModelAdmin):

list\_display=[‘id’, ‘name’, ‘sex’,.....]

admin.site.register(Userinfo, UserinfoManager)

管理器类属性：

显示设置好的列名

list\_display=[‘列名1’, ‘列名2’,.....]

将链接放到指定列

list\_display\_links=[‘列名’]

过滤器

list\_filter=[‘列名1’, ‘列名2’, ....]

搜索框（支持模糊查询）

search\_fields=[‘列名’]

添加可在列表页编辑的字段与links冲突

list\_editable=[‘列名’]

关系映射

一对一（外键）

数据之间存在一对一的对应关系

语法：OneToOneField(类名, on\_delete=xxx)

class A(models.Model):

....

class B(models.Model):

属性=models.OneToOneField(A, on\_delete=xxx)

注：OneToOne函数写在哪个类，哪个类就是含有外键值的类

特殊字段选项

on\_delete级联删除

1. models.CASCADE级联删除。django模拟SQL约束ON DELETE CASCADE的行为，并删除包含ForeignKey的对象。先删外键指向的数据，再删本体
2. models.PROTECT抛出ProtectedError以阻止被引用对象的删除
3. SET\_NULL设置ForeignKey null，需要指定null=True，可以删，外键指向null
4. SET\_DEFAULT将ForeignKey设置为其默认值，可以删，必须设置ForeignKey的默认值

创建数据

无外键的模型类：

b=xxx.objects.create(...)

无特殊操作，按原样创建

有外键的模型类（两种方式）

1. a=xxx.objects.create(.... , 关联类属性名=object)

关联一个实例名

1. a=xxx.objects.create(... , 外键名=值) 还可以写成：外键名=实例 . 主键值

关联实例对象的主键值

查询数据

1. 正向查询：直接通过外键属性查询

from .models import A

a=A.objects.get(....)

a.关联类属性名.name就是被关联对象b的名字数据，即b.name。

2. 反向查询：没有外键属性的一方，可以调用反向属性查询到关联的另一方

反向关联属性为：实例对象. 小写引用类名

b=B.objects.get(....)

b.关联类属性名.name就是关联对象a的名字值，即a.name

一对多

数据间存在一对多的对应关系

一对多需要明确出具体角色，在多表上设置外键

创建模型：

当一个A类对象可以关联多个B类对象时

class A(models.Model):

...

class B(models.Model):

属性=models.ForeignKey(A, on\_delete=xxx)

创建数据：

先创建“一”，再创建“多”

from .models import \*

a=A.objects.create(.....)

B.objects.create(.... , 关联类属性名=object实例)

或

B.onjects.create(... , 外键名=值)

查询数据

1.正向查询（B查A，即有外键的查无外键的为正向）

通过A属性查询即可

b.A

b=B.objects.get(...)

b.关联类属性名.属性

2.反向查询（A查B，即无外键的查有外键的）

a=A.objects.get(...)

bs=a.关联类名\_set.all( ) # 获取所有与a有关的B中的object对象

或

bs=B.objects.filter(关联类属性名=object对象)

for b in bs:

print(b.属性)

多对多映射

数据之间存在多对多的关系

mysql中创建第三张表来实现

django中无需手动创建第三张表，django自动完成

模型创建：在关联的两个类中的任意一个类中，增加：

属性=models.ManyToManyField(Mymodel)

示例：

class A(models.Model):

....

class B(models.Model):

属性=models.ManyToManyField(A) # 属性名为A的小写名字

创建数据

方案1先创建a对象再关联模型B

a1=A.objects.create(...)

a2=A.objects.create(...)

b1=a1.B关联模型表名\_set.create(...) # 关联A模型的a1

a2.B关联模型表名\_set.add(b1) # 关联A模型的a2

方案2先创建b对象再关联模型A

b=B.objects.create(...)

a3=b.objects.create(...)

b.A关联属性名.add(object对象)

查询数据

正向查询

多对多属性等价于objects

写有关联属性的模型查关联模型中的数据

b1.A关联属性名.all() 获取b1对应的所有的author的信息

b1.A关联属性名.filter(...) 获取b1对应的条件查询后的数据

反向查询

没写关联属性的模型查关联模型中的数据

a1.关联模型表名\_set.all()

a1.关联模型表名\_set.filter(....)

cookies和session

会话

从打开浏览器访问一个网站，到关闭浏览器结束此次访问，称之为一次会话

HTTP协议是无状态的，导致会话状态难以保持

cookies和session就是为了保持会话状态的两个存储技术

cookies

是保存在客户端浏览器上的存储空间

开发者工具->应用->cookies

特点：

cookies在浏览器上是以键值对的形式存储的，键和值都是以ASCII字符串的形式存储（不能是中文字符串）

存储的数据带有生命周期

cookies中的数据是按域存储隔离的，不同的域之间无法访问

cookies的内部数据会在每次访问此网址时都会携带到服务器端，如果cookies过大会降低响应速度

存储：

从响应出发

HttpResponse.set\_cookie(key, value=’xxx’, max\_age=None, expires=None)

key:：cookie的名字

value：cookie的值

max\_age：cookie寿命，秒为单位

expires：具体过期日期

当不指定max\_age和expires时，关闭浏览器时此数据失效

添加cookie

需要一个HttpResponse对象，该对象为目标网页

responds=HttpResponse(“已添加 my\_varl，值为123”)

responds.set\_cookie(‘my\_varl’, 123, 3600) #参数分别为： cookie名；值；存活时间

return responds

修改cookie

responds=HttpResponse(“已修改 my\_var1，值为456”)

responds.set\_cookie(‘my\_var1’, 456, 3600\*2)

return responds

获取cookies

通过request.COOKIES绑定的字典（dict）获取客户端的cookies数据

value=request.COOKIES.get(‘cookie名’, ‘默认值’)

删除cookies

HttpResponse.delete\_cookie(key)

删除指定的key的cookie，如果key不存在则什么都不发生

session

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

django中的session数据存储在数据库中，使用session前需要确保执行过migrate

实现方式

需要在浏览器客户端启动cookies，且在cookies中存储sessionid

每个客户端都可以在服务器端有一个独立的session

不同的请求者之间不会共享该数据，与请求者一一对应

使用：

session对象是一个类似于字典的SessionStore类型的对象，可以用类似于字典的方式进行操作

session能够存储字符串、整型、字典、列表等

1. 保存session的值到服务器

request.session[‘KEY’]=VALUE

1. 获取session的值

value=request.session[‘KEY’]

value=request.session.get(‘KEY’, ‘默认值’)

1. 删除session

del request.session[‘KEY’]

session在settings.py中的设置

1. SESSION\_COOKIE\_AGE

作用：指定session在cookies中的保存时长（默认两周）

语法：SESSION\_COOKIE\_AGE=60\*60\*24\*7\*2 （两周）

1. SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE=True

只要浏览器关闭，session就失效，默认为False

django中session的问题：

1. django\_session表是单表设计，且该表数据量持续增长
2. 可以执行python manage.py claersessions来清除过期的session数据

cookies和session对比：

种类 存储 安全性

cookies 浏览器 相对不安全

session 服务器 相对安全

哈希算法的应用

特点：

1. 定长输出：不管明文长度为多少，哈希值皆为定长的。MD5为32位16进制
2. 不可逆：无法反向计算出对应的明文
3. 雪崩效应：输入改变，输出必变且变化很大

使用场景：

1. 密码处理
2. 文件完整性校验

使用：

import hashlib

m=hashlib.md5() # 创建哈希算法对象

m.update(b‘xxxxx’) # 传入待计算的字节串，变量需要调用encode()方法处理

m.hexdigest() # 生成哈希值，可见16进制摘要

注：计算新的哈希值需要新建一个对象，在update中二次输入会添加到上一次输入的末尾，如上一次为123456，再次update中输入123，最终为123456123

装饰器

缓存：

定义：缓存是一类可以过快的读取数据的介质统称，也指其他可以加快数据读取的存储方式。一般用来存储临时数据，常用介质是读取速度很快的内存

意义：视图渲染有一定成本，数据库的频繁查询过高，所以对于低频变动的页面可以考虑使用缓存技术，减少实际渲染次数，用户拿到响应的时间成本会更低

缓存场景：数据变动频率较少的页面

缓存配置：settings.py中

1. 数据库缓存

说明：尽管存储介质没有更换，但是当把一次负责查询的结果直接存储到表里，比如多个条件的过滤查询结果，可避免重复进行复杂查询，提升效率

配置：

CACHES={

‘default’:{

‘BACKEND’: ‘django.core.cache.backends.db.DatabaseCache’,

‘LOCATION’: ‘存储表名’,

‘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, #缓存条数达到最大值时删除1/x的缓存

}

}

}

1. 本地内存缓存

配置：

CACHES={

‘default’: {

‘BACKEND’: ‘django.core.cache.backends.locmem.LocMemCache’,

‘LOCATION’: ‘unique-snowflake’,

}

}

1. 文件系统缓存

将缓存的数据存储到本地文件中

配置：

CACHES={

‘default’: {

‘BACKEND’: ‘django.core.cache.backends.filebased.FileBasedCache’,

‘LOCATION’: ‘/va/tmp/django\_cache’, #文件路径windows：c:\a\cache

}

}

引入cache对象

方式1：使用caches[‘CACHE配置key’]导入具体对象

from django.core.cache import caches

cache1 = caches[‘myalias’]

cache2 = caches[‘myalias\_2’]

方式2：直接引入CACHES配置项中的’default’项

from django.core.cache import cache

缓存api

1. caches.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，单位秒，默认为CACHES中的TIMEOUT值

返回值：None，成功

1. cache.get(key)获取缓存

key：缓存的key

返回值：为key的具体值，如果没有数据，则返回None

1. cache.add(key,value)存储缓存，只在key不存在时生效

返回值：True（存储成功）或False（存储失败）

1. cache.get\_or\_set(key,value,timeout)如果未获取到数据则执行set操作

返回值：value

1. cache.set\_many(dict,timeout)批量存储缓存

dict：key和value的字典

timeout：存储时间

返回值：插入不成功的key的数组

1. cache.get\_many(key\_list)批量获取缓存数据

key\_list:包含key的数组

返回值：取到的key和value的字典

1. cache.delete(key)删除key的缓存数据

返回值：None

1. cache.delete\_many(key\_list)批量删除

返回值：None

浏览器缓存策略

强缓存：不会向服务器发送请求，直接从缓存中读取资源

1. 响应头—Expires（绝对过期时间）

定义：缓存过期时间，用来指定资源到期的时间，是服务器端的具体的时间点

样例：Expires: Thu, 02 Apr 2030 05:14:03 GMT

1. 响应头—Cache-Control（相对过期时间）

在HTTP/1.1中，Cache-Control主要用于控制网页缓存。比如当’Cache-Control: max-age=120’ 代表请求创建时间后的120秒，缓存失效

说明：目前服务器都会带着这两个头同时响应给浏览器，浏览器优先使用Cache-Control

协商缓存：当强缓存到期后，浏览器会和服务器协商旧缓存还能否使用，可以则继续使用，不可以则缓存新的

1. Last-Modified响应头和If-Modified-Since请求头
2. Last-Modified为文件的最近修改时间，浏览器第一次请求静态文件时，服务器如果返回Last-Modified响应头，则代表该资源为需要协商的缓存
3. 当缓存到期后，浏览器将获取到的Last-Modified值作为请求头If-Modified-Since的值，与服务器发请求协商，服务器端返回304响应码[响应体为空]，代表缓存继续使用，200响应码代表缓存不可用[响应体为最新资源]
4. ETag响应头和If-None-Match请求头
5. ETag是服务器响应请求时，返回当前资源文件的一个唯一标识（由服务器生成），只要资源有变化，ETag就会重新生成（本质为哈希值，极高精准度）
6. 缓存到期后，浏览器将Etag响应头的值作为If-None-Match请求头的值，给服务器发请求协商，服务器接受到请求头后比对文件标识，不一致则认为资源不可用，返回200响应码[响应体为最新资源]，可用则返回304响应码

中间件：

定义：中间件是django请求/响应处理的钩子框架。是一个轻量级的、低级的“插件系统”，用于全局改变djang的输入或输出

中间件以类的形式体现

每个中间件组件负责做一些特定的功能。如django包含一个AuthenticationMiddleware，它使用会话将用户与请求关联起来

中间件类继承自django.utils.deprecation.MiddlewareMixin类

中间件类须实现下列五个方法中的一个或多个

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

1. process\_view(self,request,callback,callback\_args,callback\_kwargs)

调用试图之前被调用，在每个请求上调用，返回None或HttpResponse对象

1. 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HttpResponse对象

1. process\_exception(self,request,exception)

当处理过程中抛出异常时调用，返回一个HttpResponse对象

1. process\_template\_response(self,request,response)

在视图函数执行完毕试图返回的对象中包含render方法时被调用，该方法需要返回实现了render方法的响应对象

注：中间件中的大多数方法在返回None时表示忽略当前操作进入下一项事件，当返回HttpResponse对象时表示此请求结束直接返回给客户端

编写中间件

1. 在项目目录下创建名为middlewarep的python文件夹包
2. 创建一个py文件，在其中写中间件

注册中间件

settings.py中需要注册一下自定义的中间件

MIDDLEWARE=[

....

]

注：配置为数组，中间件被调用时以先“由上到下”再“由下到上”的顺序调用

中间件经典应用场景：过滤访问IP

获取远程客户端的IP：request.META[‘REMOTE\_ADDR’]

获取客户端访问的请求路由信息：request.path\_info

django流程图：视频中间件P2

CSRF防范（主要防范POST）

django采用‘比对暗号’机制防范攻击

cookies中存储暗号1，模板中表单里藏着暗号2，用户只有在本网站下提交数据，暗号2才会随表单提交给服务器，django对比两个暗号，对比成功则认为时合法请求，否则是违法请求—403响应码

配置步骤：

1. settings.py中确认MIDDELWARE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验证：

如果某个视图不需要django进行csrf保护，可以用装饰器关闭该视图的csrf验证

样例：

from django.views.decorators.csrf import csrf\_exempt

@csrf\_exempt

def my\_view(request):

return HttpResponse(‘xxx’)

分页

指web页面由大量数据需要显示，为了阅读方便在每个页面中只显示部分数据。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

优点：

1. 方便阅读
2. 减少数据提取量，减轻服务器压力

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)

参数：

object\_list：需要分类数据的对象列表，可以直接扔QuerySet进去

per\_page每页数据个数

返回值：

Paginator的对象

Paginator属性（宏观）

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（微观）

paginator对象.page(number) number为有效页码

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InvaildPage异常

InvalidPage：总的异常基类，包含以下两个异常子类

PageNotAnInteger：当向page( )传入一个不是整数的值时抛出

EmptyPage：当向page( )提供一个有效值，但是那个页面上没有任何对象时抛出

page对象

负责具体某一页的数据管理

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：当前页上所有数据对象的列表

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\_other\_pages()：如果有上一页或下一页返回True

next\_page\_number()：返回下一页的页码，如果下一页不存在，报InvalidPage异常

previous\_page\_number()：返回上一页的页码，如果上一页不存在，报Inavlidpage

CSV文件

逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分割值，因为分割字符也可以不是逗号），其文件以纯文本的形式存储表格数据（数字和文本）

可被常见制表工具，如excel等直接进行读取

python中生成csv文件

python提供了内建库—csv，可直接通过该库操作csv文件

import csv

with open(‘eggs.csv’, ‘w’, newline=’ ’) as csvfile: # newline=’’模式不会转义特殊字符

writer=csv.writer(csvfile)

writer.writerow([‘a’, ‘b’, ‘c’]) # 每调用一次写一行

django实现下载csv

响应Content-Type类型修改为text/csv。告诉浏览器这是csv文件，而不是HTML文件

响应会获得一个额外的Content-Disposition标头，其中包括csv文件的名称。它将被浏览器用于开启“另存为”对话框

案例：

import csv

from django.http import HttpResponse

from .models import Book

def make\_csv\_view(request):

resp=HttpResponse(content\_type=’text/csv’)

filename=’书籍信息’+’%s’%(datetime.datetime.now())+’.csv’

filename=urlquote(filename) # 让中文名可用

resp[‘Content-Disposition’]=’attachment; filename=”xxx.csv”’

all\_book=Book.objects.all()

writer=csv.writer(resp)

writer.writerow([‘id’, ’title’])

for b in all\_book:

writer.writerow([b.id, b.title])

return resp

文件上传

前端

文件上传必须为POST提交方式

表单form中文件上传时必须有带有enctype=”multipart/form-data”时才会包含文件内容数据

表单中用<input type=”file” name=”xxx”>标签上传文件

后端

视图函数中用request.FILES取文件框的内容

file=request.FILES[‘XXX’]

说明：

1. FILES的key对应页面中file框的name值
2. file绑定文件流对象
3. file.name文件名
4. file.file文件的字节流数据

配置路由

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

settings.py

MEDIA\_URL = ’/media/’ #告诉django什么样的路由是想加载用户上传的资源

MEDIA\_ROOT = os.path.join(BASE\_DIR, ’media’) #上传资源所存储的路径

MEDIA\_URL和MEIDA\_ROOT需要手动绑定

主路由中添加路由

from django.conf import settings #官方要求的引用方式

from django.conf.urls.static import static

# 在原urlpatterns数组外写

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

文件写入

方案1：传统的open方式（重名文件会覆盖）

def upload\_file(request):

a\_file=request.FILES[‘xxx’]

print(‘上传的文件名：’, a\_file.name)

filepath=os.path.join(settings.MEDIA\_ROOT, a\_file.name)

with open(filepath, ‘wb’) as f:

data=a\_file.file.read()

f.write(data)

return HttpResponse(‘上传文件成功’)

方案2：ORM方式（重名文件会自动重命名）

字段：FileField(upload=’子目录名’) #用来存储文件的相对路径

class File(models.Model):

title=CharField(max\_length=30)

picture=FileField(upload=’pic’)

def upload\_file(request):

a\_file=request.FILES[‘XXX’]

file\_title=request.POST.get(‘title’)

File.objects.create(title=file\_title, picture=a\_file)

return HttpResponse(‘文件上传成功’)

media文件夹下会生成一个pic文件夹用来存放上传的文件

邮件发送

应用场景：业务告警、邮件验证、密码找回

SMTP：全称是Simple Mail Transfer Protocol即简单邮件传输协议，端口号25

它是一组用于从源地址到目的地址传输邮件的规范，通过它来控制邮件的中转

属于推送协议

IMAP：全称是Internet Mail Access Protocol即交互式邮件访问协议，是一个应用层协议，端口号143

用来从本地邮件客户端访问远程服务器上的邮件

属于拉取协议

POP3：Post Office Protocol的简称，即邮局协议的第三个版本，是TCP/IP协议族中的一员，端口号110

本协议主要用于支持客户端远程管理在服务器上的电子邮件

属于拉取协议

IMAP对比POP3

两者均为拉取型协议，负责从邮件服务器中下载邮件到本地客户端

IMAP具备摘要浏览功能，可预览部分摘要，再下载整个邮件

IMAP为双向协议，客户端造作可反馈给服务器

POP3必须下载全部邮件，无摘要功能

POP3为单向协议，客户端操作无法同步服务器



邮件服务流程

django发邮件

django中配置邮件功能，主要为SMTP协议，负责发邮件

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封装了电子邮件的自动发送SMTP协议

django配置

EMAIL\_BACKEND=’django.core.mail.backends.smtp.EmailBackend’

EMAIL\_HOST=’smtp.qq.com’ # 对应邮箱的smtp服务器地址

EMAIL\_PORT=25 # smtp服务的端口号

EMAIL\_HOST\_USER=’xxx@qq.com’ # 发送邮件的QQ邮箱

EMAIL\_HOST\_PASSWORD=’\*\*\*\*\*\*’ # 填入授权码

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=’xxx’, # 题目

message=’xxx’, # 内容

from\_email=’xxx’, # 发送者

recipient\_list=[‘xxx@qq.com’], # 接收者邮箱

）

项目部署

在软件开发完毕后，将开发机器上运行的软件实际安装到服务器上进行长期运行

1. 在安装机器上安装和配置同版本的环境：python、数据库等
2. django项目迁移
3. 用uWSGI代替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

WSGI（Web Server Gateway Interface）web服务器网关接口，是python应用程序或框架和web服务器之间的一种接口，被广泛使用

使用python manage.py runserver通常只在开发和测试环境中使用。

当开发结束后，完善的项目代码需要在一个高效稳定的环境中运行，这时可以使用WSGI



uWSGI

是WSGI的一种，它实现了http协议WSGI协议以及uWSGI协议，uWSGI功能完善，支持协议众多，在python web圈热度极高

uWSGI主要学习如何配置

django项目轻部署

压缩django项目文件夹和开发所用的python文件夹

在服务器上解压并正确配置各项路径

安装mysql，配置好密码，创建项目数据库

makemigrations

migrate

settings.py中配置服务器ip在allow列表里

debug改为False

runserver 0.0.0.0：端口号 --insecure

注：轻部署若关闭debug则无法加载CSS和media文件，在runserver后加--insecure解决

git分布式版本控制

代码提交到本地库：依次输入以下命令

git add . # 添加问文件到本地暂存区

git commit -m “xxx” #上传到本地版本库，xxx最好为有意义的东西，可以帮助记录版本

创建远程库

github上创建远程库，复制远程库url

本地库关联远程库

git remote add origin 远程库url

git remote -v # 查看是否关联成功

首次上传到远程库

git push -u origin master # 推送到远程版本库（github）

后续上传到远程库

git push

从远程库同步代码到本地

git pull

版本回滚

git reset --hard HEAD^ # HEAD^代表回滚到上个版本，HEAD^^代表回滚到上上版本，该命令只影响本地不改变远程版本库

git pull # 同步代码完成回滚

git push --force # 回滚远程版本库到指定版本

git分支

相当于复制一份master的代码用来隔离开发或实验，不影响线上主代码的业务

新建分支：

git branch 分支名

查阅分支：

git branch

更改当前工作分支

git checkout 分支名

合并分支

1. 切换到想合并的分支
2. git merge 分支名

git首次拉取远程库到本地

git clone 项目url（ssh或https）

gitSSH密钥

最好为每个工作站配置一对密钥

创建SSH密钥对，将公钥上传到github

ssh-keygen -t rsa -C “邮箱” # -t：指定加密模式；-C：添加注释，一般为邮箱

。。。。

项目组成员角色

产品/运营经理：负责产品功能细节的把控

开发：

前端：负责显示部分内容的开发（多）

后端：负责服务器部分的功能开发（少）

运维：管理linux服务器，组件化配置，安全问题

测试：负责找出产品功能的问题（BUG）

美术：负责产品素材方面的绘制

1212