Django

ju7ran

目　录

9-阿里云部署

阿里云部署

空白目录

外键和表关系

在MySQL中，表有两种引擎，一种是InnoDB，另外一种是myisam。如果使用的是InnoDB引擎，是支持外键约束的。外键的存在使得ORM框架在处理表关系的时候异常的强大。因此这里我们首先来介绍下外键在Django中的使用。

类定义为class ForeignKey(to,on_delete,**options)。第一个参数是引用的是哪个模型，第二个参数是在使用外键引用的模型数据被删除了，这个字段该如何处理，比如有CASCADE、SET_NULL等。这里以一个实际案例来说明。比如有一个Category和一个Article两个模型。一个Category可以有多个文章，一个Article只能有一个Category，并且通过外键进行引用。

```
class Category(models.Model):
    name = models.CharField(max_length=100)

class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    # author = models.ForeignKey("User",on_delete=models.CASCADE)
    category = models.ForeignKey("Category",on_delete=models.CASCADE)
```

以上使用ForeignKey来定义模型之间的关系。即在article的实例中可以通过author属性来操作对应的User模型。这样使用起来非常的方便。

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Article,Category
# Create your views here.

def index(request):
    category = Category(name="news")
    category.save()
    article = Article(title='PHP',content='123')
    article.category = category
    article.save()
    article = Article.objects.first()

    # 获取文章分类名称
    article = Article.objects.first()
    print(article.category.name)
```

为什么使用了ForeignKey后，就能通过author访问到对应的user对象呢。因此在底层，Django为Article表添加了一个属性名_id的字段（比如author的字段名称是author_id），这个字段是一个外键，记录着对应的作者的主键。以后通过article.author访问的时候，实际上是先通过author_id找到对应的数据，然后再提取User表中的这条数据，形成一个模型。

如果想要引用另外一个app的模型，那么应该在传递to参数的时候，使用app.model_name进行指定。以上例为例，如果User和Article不是在同一个app中

```
# User模型在user这个app中
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

# Article模型在article这个app中
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    author = models.ForeignKey("user.User",on_delete=models.CASCADE)
```

如果模型的外键引用的是本身自己这个模型，那么to参数可以为'self'，或者是这个模型的名字。在论坛开发中，一般评论都可以进行二级评论，即可以针对另外一个评论进行评论，那么在定义模型的时候就需要使用外键来引用自身

```
class Comment(models.Model):
    content = models.TextField()
    origin_comment = models.ForeignKey('self',on_delete=models.CASCADE,null=True
)
    # 或者
    # origin_comment = models.ForeignKey('Comment',on_delete=models.CASCADE,null
=True)
```

外键删除操作

如果一个模型使用了外键。那么在对方那个模型被删掉后，该进行什么样的操作。可以通过on_delete来指定。可以指定的类型如下：

1.CASCADE：级联操作。如果外键对应的那条数据被删除了，那么这条数据也会被删除。

2.PROTECT：受保护。即只要这条数据引用了外键的那条数据，那么就不能删除外键的那条数据。

3.SET_NULL：设置为空。如果外键的那条数据被删除了，那么在本条数据上就将这个字段设置为空。如果设置这个选项，前提是要指定这个字段可以为空。

4.SET_DEFAULT：设置默认值。如果外键的那条数据被删除了，那么本条数据上就将这个字段设置为默认值。如果设置这个选项，前提是要指定这个字段一个默认值。

5.SET()：如果外键的那条数据被删除了。那么将会获取SET函数中的值来作为这个外键的值。SET函数可以接收一个可以调用的对象（比如函数或者方法），如果是可以调用的对象，那么会将这个对象调用后的结果作为值返回回去。

6.DO_NOTHING：不采取任何行为。一切全看数据库级别的约束。

以上这些选项只是Django级别的，数据级别依旧是RESTRICT！

1-Django前导知识

1-1-虚拟环境

为什么需要虚拟环境：

到目前位置，我们所有的第三方包安装都是直接通过pip install xx的方式进行安装的，这样安装会将那个包安装到你的系统级的Python环境中。但是这样有一个问题，就是如果你现在用Django 1.10.x写了个网站，然后你的领导跟你说，之前有一个旧项目是用Django 0.9开发的，让你来维护，但是Django 1.10不再兼容Django 0.9的一些语法了。这时候就会碰到一个问题，我如何在我的电脑中同时拥有Django 1.10和Django 0.9两套环境呢？这时候我们就可以通过虚拟环境来解决这个问题。

virtualenv

安装virtualenv

```
virtualenv是用来创建虚拟环境的软件工具，我们可以通过pip或者pip3来安装

pip install virtualenv
pip3 install virtualenv
```

创建虚拟环境

```
virtualenv [虚拟环境的名字]

创建虚拟环境的时候指定Python解释器
virtualenv -p C:\Python36\python.exe [virutalenv name]
```

进入虚拟环境

```
虚拟环境创建好了以后，那么可以进入到这个虚拟环境中，然后安装一些第三方包，进入虚拟环境在不同的
操作系统中有不同的方式，一般分为两种，
第一种是Windows
第二种是*nix：
```

```
windows进入虚拟环境：进入到虚拟环境的Scripts文件夹中，然后执行activate。

*nix进入虚拟环境：
cd  virtualenv/bin
source  activate

一旦你进入到了这个虚拟环境中，你安装包，卸载包都是在这个虚拟环境中，不会影响到外面的环境。
```

退出虚拟环境

```
退出虚拟环境很简单，通过一个命令就可以完成：deactivate。
```

virtualenvwrapper

virtualenvwrapper这个软件包可以让我们管理虚拟环境变得更加简单。不用再跑到某个目录下通过virtualenv来创建虚拟环境，并且激活的时候也要跑到具体的目录下去激活。

安装virtualenvwrapper

```
Linux:pip install virtualenvwrapper

编辑家目录下面的.bashrc文件，添加下面两行。
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh

使用source .bashrc使其生效一下。

Windows:pip install virtualenvwrapper-win
```

创建虚拟环境

```
mkvirtualenv [virutalenv name]
mkvirtualenv -p python3  [virutalenv name]  # 指定Python版本
```

切换到某个虚拟环境：

```
workon [virutalenv name]
```

退出当前虚拟环境

```
deactivate
```

列出所有虚拟环境

```
lsvirtualenv
```

删除某个虚拟环境

```
rmvirtualenv [virutalenv name]
```

创建虚拟环境的时候指定Python版本

```
mkvirtualenv --python==C:\Python36\python.exe [virutalenv name]
```

修改mkvirtualenv的默认路径

默认安装到电脑中C盘的当前登录用户的Envs目录下
在我的电脑->右键->属性->高级系统设置->环境变量->系统变量中添加一个参数WORKON_HOME，将这个参数的值设置为你需要的路径。

pipenv

Windows安装pipenv

```
pip install pipenv
```

Mac安装

```
brew install pipenv
```

Linux安装

```
pip install pipenv
```

pipfile和pipfile.lock

Pipfile文件

```
[[source]]
name = "pypi"
url = "https://mirrors.aliyun.com/pypi/simple/"
verify_ssl = true

[dev-packages]

[packages]
requests = "*"
django = "*"

[requires]
python_version = "3.6"
```

```
url           # 指定国内pip源,不然下载库会很慢

dev-packages  # 开发环境


packages      # 生产环境
django = "*"  # *表示最新版本

requires      # Python版本
```

pipfile.lock,详细记录环境依赖,并且利用了Hash算法保证了它完整对应关系

如果需要指定Python版本的话,前提是电脑中已经安装了python2和Python3

```
pipenv --three  # 泛指Python3的版本
pipenv --two    # 泛指Python2的版本
pipenv --python 3.7 # 指定Python版本
```

进入/退出/删除虚拟环境

```
pipenv shell    # 进入虚拟环境
exit            # 退出虚拟环境
pipenv --rm     # 删除整个环境  不会删除pipfile
```

管理开发环境

安装在开发环境下

```
pipenv install --dev itchat
```

在虚拟环境中运行命令,使用run参数

```
pipenv run python manage.py runserver
```

pipenv有个缺点,lock不稳定而且时间非常长,所以安装包的时候记得加上--skip-lock,最后开发完成要提价到仓库的时候在pipenv lock

```
pipenv install django --skip-lock
```

1-2-Django框架介绍与环境搭建

Django框架介绍

Django也遵循MVC思想，但是有自己的一个名词，叫做MVT

Django，发音为[`dʒæŋgəʊ]，Django诞生于2003年秋天，2005年发布正式版本，由Simon和Andrian开发。

Django版本和Python版本

https://docs.djangoproject.com/zh-hans/2.1/faq/

我应该使用哪个版本的 Python 来配合 Django?

Django 版本	Python 版本
1.11	2.7, 3.4, 3.5, 3.6, 3.7 (1.11.17添加)
2.0	3.4, 3.5, 3.6, 3.7
2.1, 2.2	3.5, 3.6, 3.7

Django开发原则

快速开发和DRY原则。Do not repeat yourself.不要自己去重复一些工作。

官网手册介绍

```
Django的官网：https://www.djangoproject.com/
Django Book2.0版本的中文文档：http://djangobook.py3k.cn/2.0/chapter01/
```

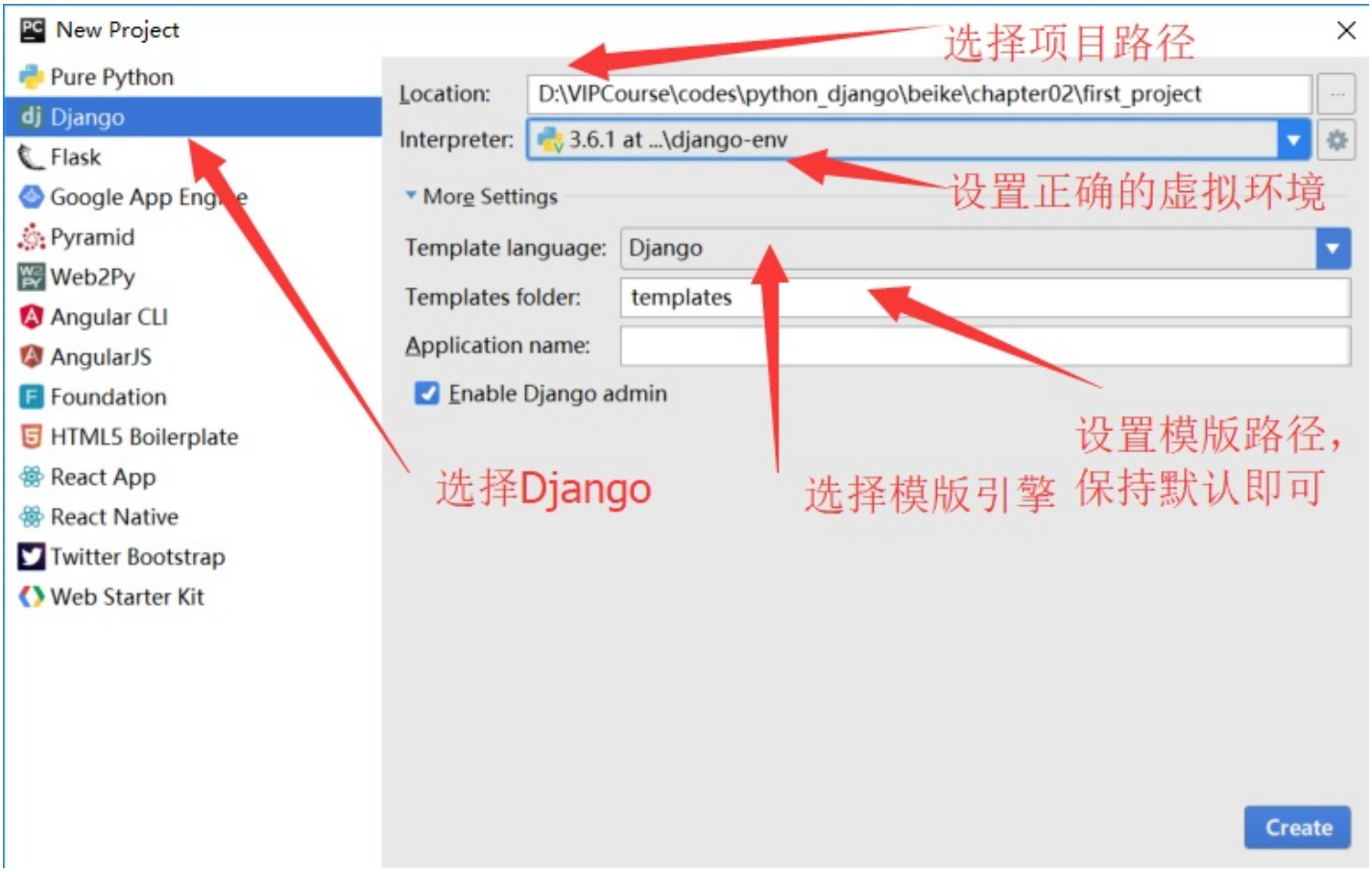
Django项目创建

1.用命令行的方式

```
创建项目：打开终端，使用命令：django-admin startproject [项目名称]
比如:django-admin startproject first_project。
```

2.用pycharm的方式

如果pycharm是专业版的话,可以用pycharm安装Django



运行Django项目

- 通过命令行的方式：python manage.py runserver。这样可以在本地访问你的网站，默认端口号是8000，这样就可以在浏览器中通过http://127.0.0.1:8000/来访问你的网站啦。如果想要修改端口号，那么在运行的时候可以指定端口号，python manage.py runserver 9000这样就可以通过9000端口来访问啦。
- 通过pycharm运行Django项目,在edit configurations中设置

项目结构介绍

manage.py：以后和项目交互基本上都是基于这个文件。一般都是在终端输入python manage.py [子命令]。可以输入python manage.py help看下能做什么事情。除非你知道你自己在做什么，一般情况下不应该编辑这个文件。

settings.py：本项目的设置项，以后所有和项目相关的配置都是放在这个里面。

urls.py：这个文件是用来配置URL路由的。比如访问http://127.0.0.1/news/是访问新闻列表页，这些东西就需要在这个文件中完成。

wsgi.py：项目与WSGI协议兼容的web服务器入口，部署的时候需要用到的，一般情况下也是不需要修改的。

project和app的关系

app是django项目的组成部分。一个app代表项目中的一个模块，所有URL请求的响应都是由app来处理。比如豆瓣，里面有图书，电影，音乐，同城等许许多多的模块，如果站在django的角度来看，图书，电影这些模块就是app，图书，电影这些app共同组成豆瓣这个项目。因此这里要有一个概念，django项目由许多app组成，一个app可以被用到其他项目，django也能拥有不同的app。

创建app

```
python manage.py startapp [app名称]
```

app中的文件

```
__init__.py 说明目录是一个Python模块
models.py   写和数据库相关的内容
views.py    接收请求，处理数据   与M和T进行交互
tests.py    写测试代码的文件(暂时不需要关心)
admin.py    网站后台管理相关的
```

应用注册

建立应用和项目之间的联系，需要对应用进行注册。

修改settings.py中的INSTALLED_APPS配置项。

Django初体验

```
from django.http import HttpResponse
from book.views import book
from move.views import move

def index(request):
    return HttpResponse("首页")

def book(request):
    return HttpResponse("图书首页")

def move(request):
    return HttpResponse("电影首页")

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',index),
    path("book",book),
    path("move",move)
]
```

DEBUG模式

- 开启了debug模式,那么修改代码,然后按下ctrl+s,那么Django会自动重启项目
- Django项目中代码出现了问题,在浏览器中和控制台中会打印错误信息
- 如果项目上线了,关闭debug模式,不然有很大的安全隐患
- 关闭DEBUG模式,在setting文件中,将DEBUG = False

2-URL与视图

2-1-URL与视图

视图

视图一般都写在app的views.py中。并且视图的第一个参数永远都是request（一个HttpRequest）对象。这个对象存储了请求过来的所有信息，包括携带的参数以及一些头部信息等。在视图中，一般是完成逻辑相关的操作。比如这个请求是添加一篇博客，那么可以通过request来接收到这些数据，然后存储到数据库中，最后再把执行的结果返回给浏览器。视图函数的返回结果必须是HttpResponseBase对象或者子类的对象。示例代码如下：

news/views.py

```
from django.http import HttpResponse
def news(request):
    return HttpResponse("新闻！")
```

urls.py

```
from news import views

urlpatterns = [
    path("news",views.news)
]
```

URL映射

视图写完后，要与URL进行映射，也即用户在浏览器中输入什么url的时候可以请求到这个视图函数。在用户输入了某个url，请求到我们的网站的时候，django会从项目的urls.py文件中寻找对应的视图。在urls.py文件中有一个urlpatterns变量，以后django就会从这个变量中读取所有的匹配规则。匹配规则需要使用django.urls.path函数进行包裹，这个函数会根据传入的参数返回URLPattern或者是URLResolver的对象。示例代码如下：

```
from django.contrib import admin
from django.urls import path
from book import views
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',views.book_list)
]
```

URL中添加参数

有时候，url中包含了一些参数需要动态调整。比如简书某篇文章的详情页的url，是https://www.jianshu.com/p/a5aab9c4978e后面的a5aab9c4978e就是这篇文章的id，那么简书的文章详情页面的url就可以写成https://www.jianshu.com/p/，其中id就是文章的id。那么如何在django中实现这种需求呢。这时候我们可以在path函数中，使用尖括号的形式来定义一个参数。比如我现在想要获取一本书籍的详细信息，那么应该在url中指定这个参数。示例代码如下：

```
from django.contrib import admin
from django.urls import path
from book import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',views.book_list),
    path('book/<book_id>/',views.book_detail)
]
----------------------------------------------------------------------
views.py中的代码如下：
def book_detail(request,book_id):
    text = "您输入的书籍的id是：%s" % book_id
    return HttpResponse(text)
```

也可以通过查询字符串的方式传递一个参数过去。示例代码如下：

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',views.book_list),
    path('book/detail/',views.book_detail)
]

----------------------------------------------------------------------
views.py中的代码如下：
def book_detail(request):
    book_id = request.GET.get("id")
    text = "您输入的书籍id是：%s" % book_id
    return HttpResponse(text)
```

URL模块化

URL中包含另外一个urls模块：

在我们的项目中，不可能只有一个app，如果把所有的app的views中的视图都放在urls.py中进行映射，肯定会让代码显得非常乱。因此django给我们提供了一个方法，可以在app内部包含自己的url匹配规则，而在项目的urls.py中再统一包含这个app的urls。使用这个技术需要借助include函数。示例代码如下：

```
# django_first/urls.py文件：

from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/',include("book.urls"))
]
```

在urls.py文件中把所有的和book这个app相关的url都移动到app/urls.py中了，django_first/urls.py中，通过include函数包含book.urls，以后在请求book相关的url的时候都需要加一个book的前缀。

Django内置转换器

```
from django.urls import converters
```

UUID:https://www.cnblogs.com/franknihao/p/7307224.html

url命名与反转

- 1.为什么需要URL命名

  因为在项目开发的过程中URL地址可能经常变动，如果写死会经常去修改
- 2.如何给一个URL指定名称

  path("",views.index,name="index")
- 3.应用命名空间

  在多个app之间可能产生同名的URL，这时候为了避免这种情况，可以使用命名空间来加以区分。在urls.py中添加app_name即可。

应用命名空间和实例命名空间

一个app,可以创建多个实例。可以使用多个URL映射同一个App。在做反转的时候,如果使用应用命名空间,就会发生混淆,为了避免这个问题,可以使用实例命名空间,实例命名空间使用,namespace='实例命名空间'

```
urls.py

from django.contrib import admin
```

```
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('cms1/', include("cms.urls",namespace='cms1')),
    path('cms2/', include("cms.urls",namespace='cms2')),
    path('front/', include("front.urls")),
]
```

URL反转传递参数

如果这个url中需要传递参数，那么可以通过kwargs来传递参数。

```
reverse("book:detail",kwargs={"book_id":1})
```

因为django中的reverse反转url的时候不区分GET请求和POST请求，因此不能在反转的时候添加查询字符串的参数。如果想要添加查询字符串的参数，只能手动的添加。

```
login_url = reverse("front:singin") + "?name=jr"
return redirect(login_url)
```

指定默认的参数

```
article/views.py
-----------------------------------------------------------------------
from django.http import HttpResponse

# Create your views here.

article_lists = ["a","b","c"]

def article(request):
    return HttpResponse(article_lists[0])

def page(request,page_id=0):
    return HttpResponse(article_lists[page_id])

-----------------------------------------------------------------------
article/urls.py

from django.urls import re_path,path
from . import views

urlpatterns = [
    path("",views.article),
```

```
    path("page/",views.page),
    path("page/<int:page_id>",views.page),
]
```

re_path函数

有时候我们在写url匹配的时候，想要写使用正则表达式来实现一些复杂的需求，那么这时候我们可以使用re_path来实现。re_path的参数和path参数一模一样，只不过第一个参数也就是route参数可以为一个正则表达式。

```
article/urls.py
-----------------------------------------------------------------------
from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r"^$",views.article),
    re_path(r"^article_list/(?P<year>\d{4})/",views.article_list)
    re_path(r"^article_list/(?P<mouth>\d{2})/",views.article_mouth)
]
```

3-模板

3-1-模板介绍

模板介绍

在之前的章节中，视图函数只是直接返回文本，而在实际生产环境中其实很少这样用，因为实际的页面大多是带有样式的HTML代码，这可以让浏览器渲染出非常漂亮的页面。DTL是Django Template Language三个单词的缩写，也就是Django自带的模板语言。当然也可以配置Django支持Jinja2等其他模板引擎，但是作为Django内置的模板语言，和Django可以达到无缝衔接而不会产生一些不兼容的情况。

DTL与普通的HTML文件的区别

DTL模板是一种带有特殊语法的HTML文件，这个HTML文件可以被Django编译，可以传递参数进去，实现数据动态化。在编译完成后，生成一个普通的HTML文件，然后发送给客户端。

渲染模板

渲染模板有多种方式。这里讲下两种常用的方式。

1.render_to_string：找到模板，然后将模板编译后渲染成Python的字符串格式。

最后再通过HttpResponse类包装成一个HttpResponse对象返回回去。

```
from django.template.loader import render_to_string
from django.http import HttpResponse

def book_detail(request,book_id):
    html = render_to_string("index.html")
    return HttpResponse(html)


错误信息：django.template.exceptions.TemplateDoesNotExist: index.html
解决办法:需要在settings.py里面设置：'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

2.以上方式虽然已经很方便了。但是django还提供了一个更加简便的方式，直

接将模板渲染成字符串和包装成HttpResponse对象一步到位完成。

```
from django.shortcuts import render
def book_list(request):
    return render(request,'index.html')
```

模板查找路径配置

在项目的settings.py文件中。有一个TEMPLATES配置，这个配置包含了模板引擎的配置，模板查找路径的配置，模板上下文的配置等。模板路径可以在两个地方配置。

1.DIRS：这是一个列表，在这个列表中可以存放所有的模板路径，以后在视图中使用render或者render_to_string渲染模板的时候，会在这个列表的路径中查找模板。

2.APP_DIRS：默认为True，这个设置为True后，会在INSTALLED_APPS的安装了的APP下的templates文件加中查找模板。settings.py中INSTALLED_APPS数组中添加你的app名字。

3.查找顺序：比如代码render('list.html')。先会在DIRS这个列表中依次查找路径下有没有这个模板，如果有，就返回。如果DIRS列表中所有的路径都没有找到，那么会先检查当前这个视图所处的app是否已经安装，如果已经安装了，那么就先在当前这个app下的templates文件夹中查找模板，如果没有找到，那么会在其他已经安装了的app中查找。如果所有路径下都没有找到，那么会抛出一个TemplateDoesNotExist的异常。

3-2-模板变量

DTL模板语法

模板中可以包含变量，Django在渲染模板的时候，可以传递变量对应的值过去进行替换。变量的命名规范和Python非常类似，只能是阿拉伯数字和英文字符以及下划线的组合，不能出现标点符号等特殊字符。变量需要通过视图函数渲染，视图函数在使用render或者render_to_string的时候可以传递一个context的参数，这个参数是一个字典类型。

```
# views.py代码
def profile(request):
    return render(request,'profile.html',context={'username':'juran'})

class Person(object):
    def __init__(self,username):
        self.username = username
def index(request):

    p = Person("居然")
    content = {
        'person':p
    }

    content = {
        'persons':[
            'a',
            'b',
            'c'
        ]
    }
    return render(request,"index.html",context=content)

# profile.html模板代码
<h1>{{ username }}</h2>
<h2>{{person.username}}</h2>
<h3>{{ persons.0 }}</h3>
```

模板中的变量同样也支持点(.)的形式。在出现了点的情况，比如person.username，模板是按照以下方式进行解析的：

```
1.如果person是一个字典，那么就会查找这个字典的username这个key对应的值。
2.如果person是一个对象，那么就会查找这个对象的username属性，或者是username这个方法。
3.如果出现的是person.1，会判断persons是否是一个列表或者元组或者任意的可以通过下标访问的对象，

如果是的话就取这个列表的第1个值。如果不是就获取到的是一个空的字符串。
```

注意

> 不能通过中括号的形式访问字典和列表中的值，比如dict['key']和list[1]是不支持的！

> 因为使用点（.）语法获取对象值的时候，可以获取这个对象的属性，如果这个对象是一个字典，也可以获取这个字典的值。所以在给这个字典添加key的时候，千万不能和字典中的一些属性重复。比如items，items是字典的方法，那么如果给这个字典添加一个items作为key，那么以后就不能再通过item来访问这个字典的键值对了。

3-3-常用标签

常用的模板标签

1.if标签：if标签相当于Python中的if语句，有elif和else相对应，但是所有的标签都需要用标签符号（{%%}）进行包裹。if标签中可以使用==、!=、<、<=、>、>=、in、not in、is、is not等判断运算符。

```
{% if age > 18 %}
    <p>您是成年人了</p>
{% elif age == 18 %}
    <p>您刚满18岁</p>
{% else %}
    <p>您是未成年人</p>
{% endif %}

{% if "张三" in persons %}
    <p>张三</p>
{% else %}
    <p>李四</p>
{% endif %}
```

用in的时候会判断类型是否相同，如果类型不相同也会判定不在列表里面

2.for...in...标签：for...in...类似于Python中的for...in...。可以遍历列表、元组、字符串、字典等一切可以遍历的对象。

```
{% for book in books %}
    <p>{{ book }}</p>
{% endfor %}

# 反向遍历
{% for book in books reversed %}
    <p>{{ book }}</p>
{% endfor %}
```

遍历字典的时候，需要使用items、keys和values等方法。在DTL中，执行一个方法不能使用圆括号的形式。

```
{% for key,value in person.items %}
    <p>key：{{ key }}</p>
    <p>value：{{ value }}</p>
{% endfor %}
```

在for循环中，DTL提供了一些变量可供使用。

```
forloop.counter：当前循环的下标。以1作为起始值。
forloop.counter0：当前循环的下标。以0作为起始值。
forloop.revcounter：当前循环的反向下标值。比如列表有5个元素，那么第一次遍历这个属性是等于5，第二次是4，以此类推。并且是以1作为最后一个元素的下标。
forloop.revcounter0：类似于forloop.revcounter。不同的是最后一个元素的下标是从0开始。
forloop.first：是否是第一次遍历。
forloop.last：是否是最后一次遍历。
forloop.parentloop：如果有多个循环嵌套，那么这个属性代表的是上一级的for循环。
```

3.for...in...empty标签：这个标签使用跟for...in...是一样的，只不过是在遍历的对象如果没有元素的情况下，会执行empty中的内容。

```
{% for person in persons %}
    <li>{{ person }}</li>
{% empty %}
    暂时还没有任何人
{% endfor %}
```

注意:在for循环中，break，continue语句是用不了的。

4.url标签：在模版中，我们经常要写一些url，比如某个a标签中需要定义href属性。当然如果通过硬编码的方式直接将这个url写死在里面也是可以的。但是这样对于以后项目维护可能不是一件好事。因此建议使用这种反转的方式来实现，类似于django中的reverse一样。

```
<a href="{% url 'book:list' %}">图书列表页面</a>
```

如果url反转的时候需要传递参数，那么可以在后面传递。但是参数分位置参数和关键字参数。位置参数和关键字参数不能同时使用。

```
# path部分
path('detail/<book_id>/',views.book_detail,name='detail')

# url反转，使用位置参数
<a href="{% url 'book:detail' 1 %}">图书详情页面</a>
```

```
# url反转，使用关键字参数
<a href="{% url 'book:detail' book_id=1 %}">图书详情页面</a>
```

如果想要在使用url标签反转的时候要传递查询字符串的参数，那么必须要手动在在后面添加。

```
<a href="{% url 'book:detail' book_id=1 %}?page=1">图书详情页面</a>
```

如果需要传递多个参数，那么通过空格的方式进行分隔。

```
<a href="{% url 'book:detail' book_id=1 page=2 %}">图书详情页面</a>
```

更多标签:https://docs.djangoproject.com/en/2.0/ref/templates/builtins/

3-4-模板常用过滤器

模版常用过滤器

在模版中，有时候需要对一些数据进行处理以后才能使用。一般在Python中我们是通过函数的形式来完成的。而在模版中，则是通过过滤器来实现的。过滤器使用的是|来使用。

add

将传进来的参数添加到原来的值上面。这个过滤器会尝试将值和参数转换成整形然后进行相加。如果转换成整形过程中失败了，那么会将值和参数进行拼接。如果是字符串，那么会拼接成字符串，如果是列表，那么会拼接成一个列表。

```
{{ value|add:"2" }}
```

如果value是等于4，那么结果将是6。如果value是等于一个普通的字符串，比如abc，那么结果将是abc2。

cut

移除值中所有指定的字符串。类似于python中的replace(args,"")。

```
{{ value|cut:" " }}
```

date

将一个日期按照指定的格式，格式化成字符串。

```
# 数据
context = {
    "birthday": datetime.now()
}

# 模版
```

```
{{ birthday|date:"Y/m/d" }}
```

那么将会输出2018/02/01。其中Y代表的是四位数字的年份，m代表的是两位数字的月份，d代表的是两位数字的日。

格式字符	描述	示例
Y	四位数字的年份	2018
m	两位数字的月份	01-12
n	月份，1-9前面没有0前缀	1-12
d	两位数字的天	01-31
j	天，但是1-9前面没有0前缀	1-31
g	小时，12小时格式的，1-9前面没有0前缀	1-12
h	小时，12小时格式的，1-9前面有0前缀	01-12
G	小时，24小时格式的，1-9前面没有0前缀	1-23
H	小时，24小时格式的，1-9前面有0前缀	01-23
i	分钟，1-9前面有0前缀	00-59
s	秒，1-9前面有0前缀	00-59

default

如果值被评估为False。比如[]，""，None，{}等这些在if判断中为False的值，都会使用default过滤器提供的默认值。

```
{{ value|default:"nothing" }}
```

如果value是等于一个空的字符串。比如""，那么以上代码将会输出nothing。

first

返回列表/元组/字符串中的第一个元素。

```
{{ value|first }}
```

如果value是等于['a','b','c']，那么输出将会是a。

last

返回列表/元组/字符串中的最后一个元素。

```
{{ value|last }}
```

如果value是等于['a','b','c']，那么输出将会是c。

floatformat

使用四舍五入的方式格式化一个浮点类型。如果这个过滤器没有传递任何参数。那么只会在小数点后保留一个小数，如果小数后面全是0，那么只会保留整数。当然也可以传递一个参数，标识具体要保留几个小数。

```
<li>{{ 34.32|floatformat }}</li>          34.3
<li>{{ 34.35|floatformat }}</li>          34.4
<li>{{ 34.353333|floatformat:3}}</li>     34.353
```

join

类似与Python中的join，将列表/元组/字符串用指定的字符进行拼接.

```
{{ value|join:"/" }}
```

如果value是等于['a','b','c']，那么以上代码将输出a/b/c。

length

获取一个列表/元组/字符串/字典的长度。

```
{{ value|length }}
```

如果value是等于['a','b','c']，那么以上代码将输出3。如果value为None，那么以上将返回0。

lower

将值中所有的字符全部转换成小写。

```
{{ value|lower }}
```

如果value是等于Hello World。那么以上代码将输出hello world。

upper

类似于lower，只不过是将指定的字符串全部转换成大写。

random

在被给的列表/字符串/元组中随机的选择一个值。

```
{{ value|random }}
```

如果value是等于['a','b','c']，那么以上代码会在列表中随机选择一个。

safe

标记一个字符串是安全的。也即会关掉这个字符串的自动转义。

```
{{value|safe}}
```

如果value是一个不包含任何特殊字符的字符串，比如这种，那么以上代码就会把字符串正常的输入。如果value是一串html代码，那么以上代码将会把这个html代码渲染到浏览器中。

slice

类似于Python中的切片操作。

```
{{ some_list|slice:"2:" }}
```

以上代码将会给some_list从2开始做切片操作。

stringtags

删除字符串中所有的html标签。

```
{{ value|striptags }}
```

如果value是hello world，那么以上代码将会输出hello world。

truncatechars

如果给定的字符串长度超过了过滤器指定的长度。那么就会进行切割，并且会拼接三个点来作为省略号。

```
{{ value|truncatechars:5 }}
```

如果value是等于北京欢迎你，那么输出的结果是北京欢迎你。如果长度大于5，会截取到长度为4的位置，后面用....来代替

更多可以查看Django源码：

```
from django.template import defaultfilters,defaulttags
```

3-5-模板结构优化

模版结构优化

引入模版

有时候一些代码是在许多模版中都用到的。如果我们每次都重复的去拷贝代码那肯定不符合项目的规范。一般我们可以把这些重复性的代码抽取出来，就类似于Python中的函数一样，以后想要使用这些代码的时候，就通过include包含进来。这个标签就是include。

```
# header.html
<p>我是header</p>

# footer.html
<p>我是footer</p>

# main.html
{% include 'header.html' %}
<p>我是main内容</p>
{% include 'footer.html' %}
```

include标签寻找路径的方式。也是跟render渲染模板的函数是一样的。

默认include标签包含模版，会自动的使用主模版中的上下文，也即可以自动的使用主模版中的变量。如果想传入一些其他的参数，那么可以使用with语句

```
# header.html
<p>用户名：{{ username }}</p>

# main.html
{% include "header.html" with username='juran' %}
```

模板继承

在前端页面开发中。有些代码是需要重复使用的。这种情况可以使用include标签来实现。也可以使用另外一个比较强大的方式来实现，那就是模版继承。模版继承类似于Python中的类，在父类中可以先定义好一些变量和方法，然后在子类中实现。模版继承也可以在父模版中先定义好一些子模版需要用到的代码，然后子模版直接继承就可以了。并且因为子模版肯定有自己的不同代码，因此可以在父模版中定义一个block接口，然后子模版再去实现。

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="{% static 'style.css' %}" />
    <title>{% block title %}我的站点{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">首页</a></li>
            <li><a href="/blog/">博客</a></li>
        </ul>
        {% endblock %}
    </div>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

这个模版，我们取名叫做base.html，定义好一个简单的html骨架，然后定义好两个block接口，让子模版来根据具体需求来实现。子模板然后通过extends标签来实现.

```
{% extends "base.html" %}

{% block title %}博客列表{% endblock %}

{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

需要注意的是：extends标签必须放在模版的第一行。

子模板中的代码必须放在block中，否则将不会被渲染。

如果在某个block中需要使用父模版的内容，那么可以使用{{block.super}}来继承。比如上例，{%block title%}，如果想要使用父模版的title，那么可以在子模版的title block中使用{{ block.super }}来实现。

在定义block的时候，除了在block开始的地方定义这个block的名字，还可以在block结束的时候定义名字。比如{% block title %}{% endblock title %}。这在大型模版中显得尤其有用，能让你快速的看到block包含在哪里。

3-6-加载静态文件

加载静态文件

在一个网页中，不仅仅只有一个html骨架，还需要css样式文件，js执行文件以及一些图片等。因此在DTL中加载静态文件是一个必须要解决的问题。在DTL中，使用static标签来加载静态文件。要使用static标签，首先需要{% load static %}。加载静态文件的步骤如下：

1.首先确保django.contrib.staticfiles已经添加到settings.INSTALLED_APPS中。

2.确保在settings.py中设置了STATIC_URL。

3.在已经安装了的app下创建一个文件夹叫做static，然后再在这个static文件夹下创建一个当前app的名字的文件夹，再把静态文件放到这个文件夹下。

例如你的app叫做book，有一个静态文件叫做logo.jpg，那么路径为book/static/book/logo.jpg。（为什么在app下创建一个static文件夹，还需要在这个static下创建一个同app名字的文件夹呢？原因是如果直接把静态文件放在static文件夹下，那么在模版加载静态文件的时候就是使用logo.jpg，如果在多个app之间有同名的静态文件，这时候可能就会产生混淆。而在static文件夹下加了一个同名app文件夹，在模版中加载的时候就是使用app/logo.jpg，这样就可以避免产生混淆。）

4.如果有一些静态文件是不和任何app挂钩的。那么可以在settings.py中添加STATICFILES_DIRS，以后DTL就会在这个列表的路径中查找静态文件。比如可以设置为:

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR,"static")
]
```

5.在模版中使用load标签加载static标签。比如要加载在项目的static文件夹下的style.css的文件。

```
{% load static %}
<link rel="stylesheet" href="{% static 'style.css' %}">
```

6.如果不想每次在模版中加载静态文件都使用load加载static标签，那么可以在settings.py中的TEMPLATES/OPTIONS添加'builtins':['django.templatetags.static']，这样以后在模版中就可以直接使用static标签，而不用手动的load了。

7.如果没有在settings.INSTALLED_APPS中添加django.contrib.staticfiles。

那么我们就需要手动的将请求静态文件的url与静态文件的路径进行映射了。

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

urlpatterns = [
    # 其他的url映射
] + static(settings.STATIC_URL, document_root=settings.STATICFILES_DIRS[0])
```

4-数据库

4-1-操作数据库

MySQL驱动程序安装

我们使用Django来操作MySQL，实际上底层还是通过Python来操作的。因此我们想要用Django来操作MySQL，首先还是需要安装一个驱动程序。在Python3中，驱动程序有多种选择。比如有pymysql以及mysqlclient等。这里我们就使用mysqlclient来操作。mysqlclient安装非常简单。只需要通过pip install mysqlclient即可安装。

常见MySQL驱动介绍：

1. MySQL-python：也就是MySQLdb。是对C语言操作MySQL数据库的一个简单封装。遵循了Python DB API v2

   但是只支持Python2，目前还不支持Python3。
2. mysqlclient：是MySQL-python的另外一个分支。支持Python3 并且修复了一些bug。
3. pymysql：纯Python实现的一个驱动。因为是纯Python编写的，因此执行效率不如MySQL-python。并且也因为是纯Python编写的，因此可以和Python代码无缝衔接。
4. MySQL Connector/Python：MySQL官方推出的使用纯Python连接MySQL的驱动。因为是纯Python开发的。效率不高。

Django配置连接数据库

在操作数据库之前，首先先要连接数据库。这里我们以配置MySQL为例来讲解。Django连接数据库，不需要单独的创建一个连接对象。只需要在settings.py文件中做好数据库相关的配置就可以了

```
DATABASES = {
    'default': {
        # 数据库引擎（是mysql还是oracle等）
        'ENGINE': 'django.db.backends.mysql',
        # 数据库的名字
        'NAME': 'logic',
        # 连接mysql数据库的用户名
```

```
        'USER': 'root',
        # 连接mysql数据库的密码
        'PASSWORD': 'root',
        # mysql数据库的主机地址
        'HOST': '127.0.0.1',
        # mysql数据库的端口号
        'PORT': '3306',
    }
}
```

连接Linux服务器MySQL问题:https://blog.csdn.net/qq473179304/article/details/56665364

在Django中操作数据库

在Django中操作数据库有两种方式。第一种方式就是使用原生sql语句操作，第二种就是使用ORM模型来操作。在Django中使用原生sql语句操作其实就是使用python db api的接口来操作。如果你的mysql驱动使用的是pymysql，那么你就是使用pymysql来操作的，只不过Django将数据库连接的这一部分封装好了，我们只要在settings.py中配置好了数据库连接信息后直接使用Django封装好的接口就可以操作了

```
# 使用django封装好的connection对象，会自动读取settings.py中数据库的配置信息
from django.db import connection

# 获取游标对象
cursor = connection.cursor()
# 拿到游标对象后执行sql语句
cursor.execute("select * from book")
# 获取所有的数据
rows = cursor.fetchall()
# 遍历查询到的数据
for row in rows:
    print(row)
```

以上的execute以及fetchall方法都是Python DB API规范中定义好的。任何使用Python来操作MySQL的驱动程序都应该遵循这个规范。所以不管是使用pymysql或者是mysqlclient或者是mysqldb，他们的接口都是一样的。

Python DB API下规范下cursor对象常用接口

1. `description`：如果`cursor`执行了查询的`sql`代码。那么读取`cursor.description`属性的时候，将返回一个列表，这个列表中装的是元组，元组中装的分别是`(name,type_code,display_size,internal_size,precision,scale,null_ok)`，其中`name`代表的是查找出来的数据的字段名称，其他参数暂时用处不大。
2. `rowcount`：代表的是在执行了`sql`语句后受影响的行数。
3. `close`：关闭游标。关闭游标以后就再也不能使用了，否则会抛出异常。

4. `execute(sql[,parameters])` ：执行某个 `sql` 语句。如果在执行 `sql` 语句的时候还需要传递参数，那么可以传给 `parameters` 参数。示例代码如下：

```
cursor.execute("select * from article where id=%s",(1,))
```

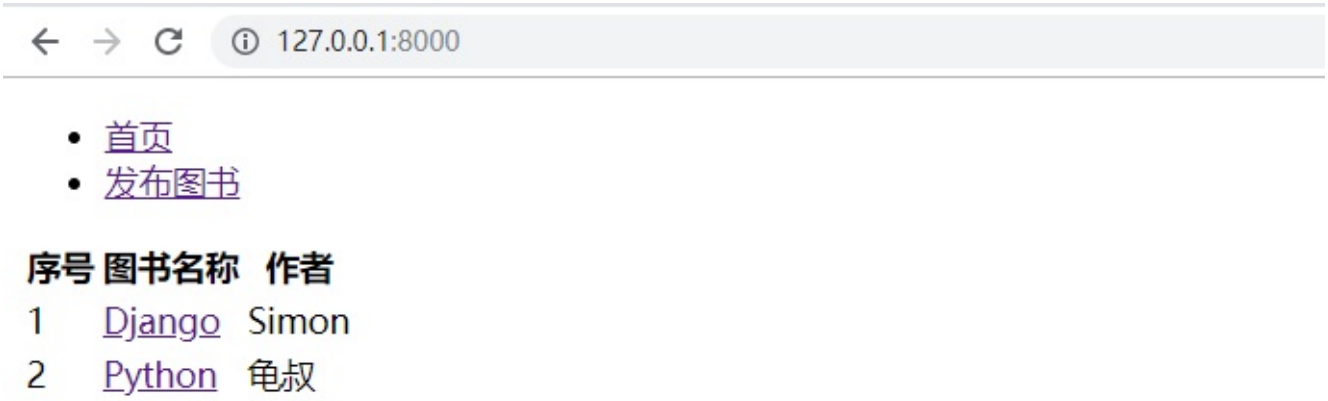
5. `fetchone` ：在执行了查询操作以后，获取第一条数据。
6. `fetchmany(size)` ：在执行查询操作以后，获取多条数据。具体是多少条要看传的 `size` 参数。如果不传 `size` 参数，那么默认是获取第一条数据。
7. `fetchall` ：获取所有满足 `sql` 语句的数据。

4-2-图书管理系统

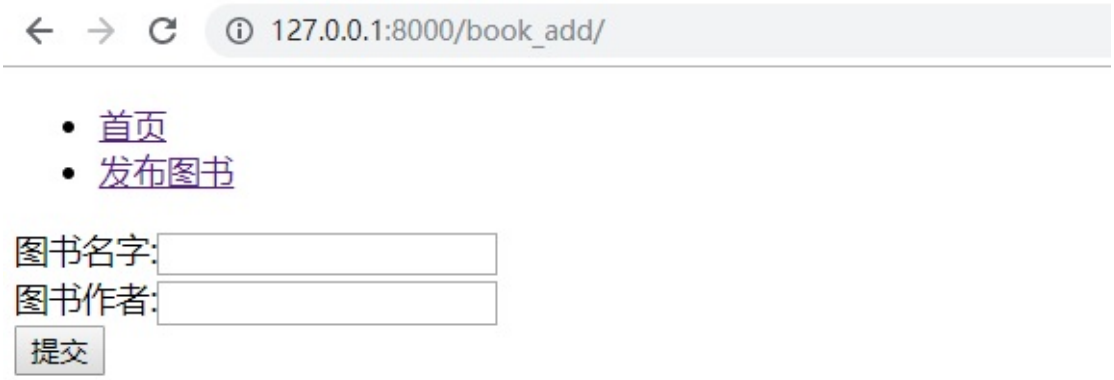
需求

完成图书的增删改查，以及前端的页面的展示

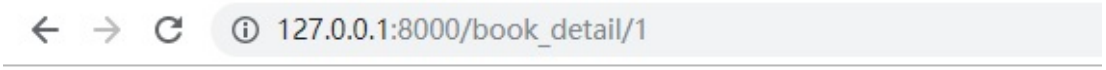
首页



图书添加页面



图书详情页面



- 首页
- 发布图书

图书详情

序号：1

书名：Django

删除图书

4-3-ORM模型介绍

ORM模型介绍

随着项目越来越大，采用写原生SQL的方式在代码中会出现大量的SQL语句，那么问题就出现了：

1.SQL语句重复利用率不高，越复杂的SQL语句条件越多，代码越长。会出现很多相近的SQL语句。

2.很多SQL语句是在业务逻辑中拼出来的，如果有数据库需要更改，就要去修改这些逻辑，这会很容易漏掉对某些SQL语句的修改。

3.写SQL时容易忽略web安全问题，给未来造成隐患。SQL注入。

```
select * from user where username = 'juran'#' and  password = ''

select * from user where username = 'juran'-- ' and  password = ''

select * from sqldb_book where id = -1 or 1=1
```

ORM，全称Object Relational Mapping，中文叫做对象关系映射，通过ORM我们可以通过类的方式去操作数据库，而不用再写原生的SQL语句。通过把表映射成类，把行作实例，把字段作为属性，ORM在执行对象操作的时候最终还是会把对应的操作转换为数据库原生语句。

```
from django.db import models
# 创建一个模型,对应数据库中的一张表
class Book(models.Model):
    id = models.AutoField()
    name = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    price = models.FloatField()

# 一个模型的对象,对应数据库表中的一条数据
book = Book(name="Python",author='龟叔',price=89)
# save方法,保存
book.save()
```

```
# delete方法，删除
book.delete()
```

使用ORM有许多优点

1.易用性：使用ORM做数据库的开发可以有效的减少重复SQL语句的概率，写出来的模型也更加直观、清晰。

2.性能损耗小：ORM转换成底层数据库操作指令确实会有一些开销。但从实际的情况来看，这种性能损耗很少（不足5%），只要不是对性能有严苛的要求，综合考虑开发效率、代码的阅读性，带来的好处要远远大于性能损耗，而且项目越大作用越明显。

3.设计灵活：可以轻松的写出复杂的查询。

4.可移植性：Django封装了底层的数据库实现，支持多个关系数据库引擎，包括流行的MySQL、PostgreSQL和SQLite。可以非常轻松的切换数据库。

创建ORM模型

ORM模型一般都是放在app的models.py文件中。每个app都可以拥有自己的模型。并且如果这个模型想要映射到数据库中，那么这个app必须要放在settings.py的INSTALLED_APP中进行安装。以下是写一个简单的书籍ORM模型。

```
from django.db import models
class Book(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=20,null=False)
    author = models.CharField(max_length=20,null=False)
    pub_time = models.DateTimeField(default=datetime.now)
    price = models.FloatField(default=0)
```

以上便定义了一个模型。这个模型继承自django.db.models.Model，如果这个模型想要映射到数据库中，就必须继承自这个类。这个模型以后映射到数据库中，表名是模型名称的小写形式，为book。在这个表中，有四个字段，一个为name，这个字段是保存的是书的名称，是varchar类型，最长不能超过20个字符，并且不能为空。第二个字段是作者名字类型，同样也是varchar类型，长度不能超过20个。第三个是出版时间，数据类型是datetime类型，默认是保存这本书籍的时间。第五个是这本书的价格，是浮点类型。

还有一个字段我们没有写，就是主键id，在django中，如果一个模型没有定义主键，那么将会自动生成一个自动增长的int类型的主键，并且这个主键的名字就叫做id。

映射模型到数据库中

将ORM模型映射到数据库中，总结起来就是以下几步：

1.在settings.py中，配置好DATABASES，做好数据库相关的配置。

2.在app中的models.py中定义好模型，这个模型必须继承自django.db.models。

3.将这个app添加到settings.py的INSTALLED_APP中。

4.在命令行终端，进入到项目所在的路径，然后执行命令python manage.py makemigrations来生成迁移脚本文件。

5.同样在命令行中，执行命令python manage.py migrate来将迁移脚本文件映射到数据库中。

4-4-ORM模型的增删改查

ORM的增删改查

添加数据

```
views.py
-----------------------------------------------------------------------
from django.shortcuts import render

from django.db import connection
from .models import Book
from django.http import HttpResponse

def add_book(request):
    book = Book(name="Python",author="JR",price=78)
    book.save()
    return HttpResponse("书籍添加成功")
```

查询数据

```
1.根据主键进行查找
book = Book.objects.get(pk=1)
print(book)

2.根据其他条件来查找
book = Book.objects.filter(name="Python")
print(book)

3.查询所有
book = Book.objects.all()
```

删除数据

```
book = Book.objects.get(pk=1)
```

```
book.delete()
```

修改数据

```
book = Book.objects.get(pk=2)
book.price = 200
book.save()
```

4-5-模型常用属性

常用字段

AutoField

映射到数据库中是int类型，可以有自动增长的特性。一般不需要使用这个类型，如果不指定主键，那么模型会自动的生成一个叫做id的自动增长的主键。如果你想指定一个其他名字的并且具有自动增长的主键，使用AutoField也是可以的。

BigAutoField

64位的整形，类似于AutoField，只不过是产生的数据的范围是从1-9223372036854775807。

BooleanField

在模型层面接收的是True/False。在数据库层面是tinyint类型。如果没有指定默认值，默认值是None。

CharField

在数据库层面是varchar类型。在Python层面就是普通的字符串。这个类型在使用的时候必须要指定最大的长度，也即必须要传递max_length这个关键字参数进去。
最大长度计算:https://www.cnblogs.com/canger/p/9850727.html

DateTimeField

日期时间类型，不仅仅可以存储日期，还可以存储时间。映射到数据库中是datetime类型。

```
TIME_ZONE = 'Asia/Shanghai'

from django.utils.timezone import localtime,now

auto_now：在每次这个数据保存的时候，都使用当前的时间。
比如作为一个记录修改日期的字段，可以将这个属性设置为True。

auto_now_add：在每次数据第一次被添加进去的时候，都使用当前的时间。
```

比如作为一个记录第一次入库的字段，可以将这个属性设置为True。

EmailField

类似于CharField。在数据库底层也是一个varchar类型。最大长度是254个字符。

FileField

用来存储文件的。

ImageField

用来存储图片文件的。

FloatField

浮点类型。映射到数据库中是float类型。

IntegerField

整形。值的区间是-2147483648——2147483647。

BigIntegerField

大整形。值的区间是-9223372036854775808——9223372036854775807。

PositiveIntegerField

正整形。值的区间是0——2147483647。

SmallIntegerField

小整形。值的区间是-32768——32767。

PositiveSmallIntegerField

正小整形。值的区间是0——32767。

TextField

大量的文本类型。映射到数据库中是longtext类型。

UUIDField

只能存储uuid格式的字符串。uuid是一个32位的全球唯一的字符串，一般用来作为主键。

URLField

类似于CharField，只不过只能用来存储url格式的字符串。并且默认的max_length是200。

Field的常用参数

null

如果设置为True，Django将会在映射表的时候指定是否为空。默认是为False。在使用字符串相关的Field（CharField/TextField）的时候，官方推荐尽量不要使用这个参数，也就是保持默认值False。因为Django在处理字符串相关的Field的时候，即使这个Field的null=False，如果你没有给这个Field传递任何值，那么Django也会使用一个空的字符串""来作为默认值存储进去。因此如果再使用null=True，Django会产生两种空值的情形（NULL或者空字符串）。如果想要在表单验证的时候允许这个字符串为空，那么建议使用blank=True。如果你的Field是BooleanField，那么对应的可空的字段则为NullBooleanField。

db_column

这个字段在数据库中的名字。如果没有设置这个参数，那么将会使用模型中属性的名字。

default

默认值。可以为一个值，或者是一个函数，但是不支持lambda表达式。并且不支持列表/字典/集合等可变的数据结构。

primary_key

是否为主键。默认是False。

unique

在表中这个字段的值是否唯一。一般是设置手机号码/邮箱等。
更多Field参数请参考官方文档：https://docs.djangoproject.com/zh-hans/2.2/ref/models/fields/

模型中Meta配置

对于一些模型级别的配置。我们可以在模型中定义一个类，叫做Meta。然后在这个类中添加一些类属性来控制模型的作用。比如我们想要在数据库映射的时候使用自己指定的表名，而不是使用模型的名称。那么我们可以在Meta类中添加一个db_table的属性。

```
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    desc = models.CharField(max_length=100,name='description',db_column="descri
ption1")

    class Meta:
        db_table = 'book_model'
```

db_table

这个模型映射到数据库中的表名。如果没有指定这个参数，那么在映射的时候将会使用模型名来作为默认的表名。

ordering

设置在提取数据的排序方式。比如我想在查找数据的时候根据添加的时间排序

```
class Book(models.Model):
    name = models.CharField(max_length=20,null=False)
    desc = models.CharField(max_length=100,name='description',db_column="descri
ption1")
    pub_date = models.DateTimeField(auto_now_add=True)

    class Meta:
        db_table = 'book_model'
        ordering = ['pub_date']     # 正序
        ordering = ['-pub_date']    # 倒序
```

4-6-外键和表

外键和表关系

在MySQL中，表有两种引擎，一种是InnoDB，另外一种是myisam。如果使用的是InnoDB引擎，是支持外键约束的。外键的存在使得ORM框架在处理表关系的时候异常的强大。因此这里我们首先来介绍下外键在Django中的使用。

类定义为class ForeignKey(to,on_delete,**options)。第一个参数是引用的是哪个模型，第二个参数是在使用外键引用的模型数据被删除了，这个字段该如何处理，比如有CASCADE、SET_NULL等。这里以一个实际案例来说明。比如有一个Category和一个Article两个模型。一个Category可以有多个文章，一个Article只能有一个Category，并且通过外键进行引用。

```
class Category(models.Model):
    name = models.CharField(max_length=100)

class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    # author = models.ForeignKey("User",on_delete=models.CASCADE)
    category = models.ForeignKey("Category",on_delete=models.CASCADE)
```

以上使用ForeignKey来定义模型之间的关系。即在article的实例中可以通过author属性来操作对应的User模型。这样使用起来非常的方便。

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Article,Category
# Create your views here.

def index(request):
    category = Category(name="news")
    category.save()
```

```
    article = Article(title='PHP',content='123')
    article.category = category
    article.save()
    article = Article.objects.first()

    # 获取文章分类名称
    article = Article.objects.first()
    print(article.category.name)
```

为什么使用了ForeignKey后，就能通过author访问到对应的user对象呢。因此在底层，Django为Article表添加了一个属性名_id的字段（比如author的字段名称是author_id），这个字段是一个外键，记录着对应的作者的主键。以后通过article.author访问的时候，实际上是先通过author_id找到对应的数据，然后再提取User表中的这条数据，形成一个模型。

如果想要引用另外一个app的模型，那么应该在传递to参数的时候，使用app.model_name进行指定。以上例为例，如果User和Article不是在同一个app中

```
# User模型在user这个app中
class User(models.Model):
    username = models.CharField(max_length=20)
    password = models.CharField(max_length=100)

# Article模型在article这个app中
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    author = models.ForeignKey("user.User",on_delete=models.CASCADE)
```

如果模型的外键引用的是本身自己这个模型，那么to参数可以为'self'，或者是这个模型的名字。在论坛开发中，一般评论都可以进行二级评论，即可以针对另外一个评论进行评论，那么在定义模型的时候就需要使用外键来引用自身

```
class Comment(models.Model):
    content = models.TextField()
    origin_comment = models.ForeignKey('self',on_delete=models.CASCADE,null=True
)
    # 或者
    # origin_comment = models.ForeignKey('Comment',on_delete=models.CASCADE,null
=True)
```

外键删除操作

如果一个模型使用了外键。那么在对方那个模型被删掉后，该进行什么样的操作。可以通过on_delete来指

定。可以指定的类型如下：

1.CASCADE：级联操作。如果外键对应的那条数据被删除了，那么这条数据也会被删除。

2.PROTECT：受保护。即只要这条数据引用了外键的那条数据，那么就不能删除外键的那条数据。

3.SET_NULL：设置为空。如果外键的那条数据被删除了，那么在本条数据上就将这个字段设置为空。如果设置这个选项，前提是要指定这个字段可以为空。

4.SET_DEFAULT：设置默认值。如果外键的那条数据被删除了，那么本条数据上就将这个字段设置为默认值。如果设置这个选项，前提是要指定这个字段一个默认值。

5.SET()：如果外键的那条数据被删除了。那么将会获取SET函数中的值来作为这个外键的值。SET函数可以接收一个可以调用的对象（比如函数或者方法），如果是可以调用的对象，那么会将这个对象调用后的结果作为值返回回去。

6.DO_NOTHING：不采取任何行为。一切全看数据库级别的约束。

以上这些选项只是Django级别的，数据级别依旧是RESTRICT！

4-7-查询操作

查询操作

查找是数据库操作中一个非常重要的技术。查询一般就是使用filter、exclude以及get三个方法来实现。我们可以在调用这些方法的时候传递不同的参数来实现查询需求。在ORM层面，这些查询条件都是使用field+__+condition的方式来使用的。

查询条件

exact

使用精确的=进行查找。如果提供的是一个None，那么在SQL层面就是被解释为NULL

```
article = Article.objects.get(id__exact=14)
article = Article.objects.get(id__exact=None)

以上的两个查找在翻译为SQL语句

select * from article where id=14;
select * from article where id IS NULL;

article.query,可以得到Django执行的SQL语句。但是只能作用于QuerySet对象上。
```

iexact

使用like进行查找。

```
article = Article.objects.filter(title__iexact='hello world')

那么以上的查询就等价于以下的SQL语句：

select * from article where title like 'hello world';
```

注意上面这个sql语句，因为在MySQL中，没有一个叫做ilike的。所以exact和iexact的区别实际上就是LIKE和=

的区别，在大部分collation=utf8_general_ci情况下都是一样的（collation是用来对字符串比较的）。

contains

大小写敏感，判断某个字段是否包含了某个数据。

```
articles = Article.objects.filter(title__contains='hello')

在翻译成SQL语句为如下：

select * where title like binary '%hello%';
```

要注意的是，在使用contains的时候，翻译成的sql语句左右两边是有百分号的，意味着使用的是模糊查询。而exact翻译成sql语句左右两边是没有百分号的，意味着使用的是精确的查询。

icontains

大小写不敏感的匹配查询。

```
articles = Article.objects.filter(title__icontains='hello')
在翻译成SQL语句为如下：

select * where title like '%hello%';
```

in

提取那些给定的field的值是否在给定的容器中。容器可以为list、tuple或者任何一个可以迭代的对象，包括QuerySet对象。

```
articles = Article.objects.filter(id__in=[1,2,3])
以上代码在翻译成SQL语句为如下：

select *from articles where id in (1,2,3)

当然也可以传递一个QuerySet对象进去。
# 查找标题为hello的文章分类
articles = Article.objects.filter(title__icontains="hello")
category = Category.objects.filter(article__in=articles)

# 查找文章ID为1, 2, 3的文章分类
category = Category.objects.filter(article__id__in=[1,2,3])
```

根据关联的表进行查询

```
想要获取文章标题中包含"hello"的所有的分类。那么可以通过以下代码来实现：

categories = Category.object.filter(article__title__contains("hello"))
```

比较运算

gt

```
某个field的值要大于给定的值。
# 将所有id大于4的文章全部都找出来。
articles = Article.objects.filter(id__gt=4)
将翻译成以下SQL语句：
select * from articles where id > 4;
```

gte

类似于gt，是大于等于。

lt

类似于gt是小于。

lte

类似于lt，是小于等于。

range

判断某个field的值是否在给定的区间中。

```
start_date = datetime(year=2019, month=12, day=1,hour=10,minute=0,second=0)
end_date = datetime(year=2019, month=12, day=30,hour=10,minute=0,second=0)
date_range = Common.objects.filter(test_date__range=(start_date,end_date))

以上代码的意思是提取所有发布时间在2019/1/1到2019/12/12之间的文章。
将翻译成以下的SQL语句：
SELECT `user_common`.`id`, `user_common`.`content`, `user_common`.`pid`, `user_
common`.`test_date` FROM `user_common` WHERE `user_common`.`test_date` BETWEEN
2019-12-01 02:00:00 AND 2019-12-30 02:00:00。
```

date

针对某些date或者datetime类型的字段。可以指定date的范围。并且这个时间过滤，还可以使用链式调用。

```
date_test = Common.objects.filter(test_date__date=datetime(year=2018,month=12,d
ay=19))

print(date_test.query)
print(date_test)

翻译成SQL语句:
SELECT `user_common`.`id`, `user_common`.`content`, `user_common`.`pid`, `user_
common`.`test_date` FROM `user_common` WHERE DATE(`user_common`.`test_date`) =
```

```
2018-12-19
```

year

根据年份进行查找。

```
articles = Article.objects.filter(pub_date__year=2018)
articles = Article.objects.filter(pub_date__year__gte=2017)
以上的代码在翻译成SQL语句为如下：

select ... where pub_date between '2018-01-01' and '2018-12-31';
select ... where pub_date >= '2017-01-01';
```

time

根据时间进行查找。示例代码如下：

```
articles = Article.objects.filter(pub_date__time=time(hour=15,minute=21,second=
10))
以上的代码是获取每一天中15点21分10秒发表的所有文章。

# 查询10秒到11秒之间的
start_date = time(hour=17,minute=21,second=10)
end_date = time(hour=17,minute=21,second=11)
date_test = Common.objects.filter(test_date__time__range = (start_date,end_date)
)
```

聚合函数

如果你用原生SQL，则可以使用聚合函数来提取数据。比如提取某个商品销售的数量，那么可以使用Count，如果想要知道商品销售的平均价格，那么可以使用Avg。

聚合函数是通过aggregate方法来实现的。

```
from django.db import models

class Author(models.Model):
    """作者模型"""
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField()

    class Meta:
        db_table = 'author'
```

```
class Publisher(models.Model):
    """出版社模型"""
    name = models.CharField(max_length=300)

    class Meta:
        db_table = 'publisher'


class Book(models.Model):
    """图书模型"""
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.FloatField()
    rating = models.FloatField()
    author = models.ForeignKey(Author,on_delete=models.CASCADE)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)

    class Meta:
        db_table = 'book'


class BookOrder(models.Model):
    """图书订单模型"""
    book = models.ForeignKey("Book",on_delete=models.CASCADE)
    price = models.FloatField()

    class Meta:
        db_table = 'book_order'
```

聚合函数的使用

1.Avg：求平均值。比如想要获取所有图书的价格平均值。那么可以使用以下代码实现。

```
from django.db.models import Avg
from django.db import connection
result = Book.objects.aggregate(Avg('price'))
print(connection.queries)       # 打印SQL语句
print(result)
```

以上的打印结果是：

```
{"price__avg":23.0}
```

其中price__avg的结构是根据field__avg规则构成的。如果想要修改默认的名字，那么可以将Avg赋值给一个关键字参数。

```
result = Book.objects.aggregate(my_avg=Avg('price'))
print(result)
```

那么以上的结果打印为：

```
{"my_avg":23}
```

2.Count：获取指定的对象的个数。

```
from django.db.models import Count
result = Book.objects.aggregate(book_num=Count('id'))
以上的result将返回Book表中总共有多少本图书。

Count类中，还有另外一个参数叫做distinct，默认是等于False，如果是等于True，那么将去掉那些重复的值。

比如要获取作者表中所有的不重复的邮箱总共有多少个。
from djang.db.models import Count
result = Author.objects.aggregate(count=Count('email',distinct=True))

# 统计每本图书的销量
result = Book.objects.annotate(book_nums=Count("bookorder"))
for book in result:
    print("%s/%s"%(book.name,book.book_nums))
```

3.Max和Min：获取指定对象的最大值和最小值。比如想要获取Author表中，最大的年龄和最小的年龄分别是多少。

```
from django.db.models import Max,Min
result = Author.objects.aggregate(Max('age'),Min('age'))
如果最大的年龄是88,最小的年龄是18。那么以上的result将为：
{"age__max":88,"age__min":18}

# 统计每本售卖图书的最大值和最小值
request = Book.objects.annotate(max=Max("bookorder__price"),min=Min("bookorder__price"))
print(request)
```

4.Sum：求指定对象的总和。比如要求图书的销售总额。

```
from djang.db.models import Sum

result = Book.objects.aggregate(total=Sum("price"))

# 每一本图书的销售总额
result = Book.objects.annotate(total=Sum("bookorder__price"))

# 统计2019年，销售总额
result = BookOrder.objects.filter(create_time__year=2019).aggregate(total=Sum("price"))
```

aggregate和annotate的区别

1·aggregate：返回使用聚合函数后的字段和值。

2.annotate：在原来模型字段的基础之上添加一个使用了聚合函数的字段，并且在使用聚合函数的时候，会使用当前这个模型的主键进行分组（group by）。

```
# 求每一本图书销售的平均价格
result = Book.objects.aggregate(avg=Avg("bookorder__price"))
print(result)
print(connection.queries)

result = Book.objects.annotate(avg=Avg("bookorder__price"))
print(result)
```

F表达式和Q表达式

F表达式

F表达式是用来优化ORM操作数据库的。比如我们要将公司所有员工的薪水都增加1000元，如果按照正常的流程，应该是先从数据库中提取所有的员工工资到Python内存中，然后使用Python代码在员工工资的基础之上增加1000元，最后再保存到数据库中。这里面涉及的流程就是，首先从数据库中提取数据到Python内存中，然后在Python内存中做完运算，之后再保存到数据库中。

```
employees = Employee.objects.all()
for employee in employees:
    employee.salary += 1000
    employee.save()
```

而我们的F表达式就可以优化这个流程，他可以不需要先把数据从数据库中提取出来，计算完成后再保存回去，他可以直接执行SQL语句，就将员工的工资增加1000元。

```
from djang.db.models import F
Employee.objects.update(salary=F("salary")+1000)
F表达式并不会马上从数据库中获取数据，而是在生成SQL语句的时候，动态的获取传给F表达式的值。
```

比如如果想要获取作者中，name和email相同的作者数据。如果不使用F表达式。

```
authors = Author.objects.all()
for author in authors:
    if author.name == author.email:
        print(author)
如果使用F表达式，那么一行代码就可以搞定。示例代码如下：
```

```
from django.db.models import F
authors = Author.objects.filter(name=F("email"))
```

Q表达式

如果想要实现所有价格高于100元，并且评分达到9.0以上评分的图书。

```
books = Book.objects.filter(price__gte=100,rating__gte=9)
```

以上这个案例是一个并集查询，可以简单的通过传递多个条件进去来实现。

但是如果想要实现一些复杂的查询语句，比如要查询所有价格低于10元，或者是评分低于9分的图书。那就没有办法通过传递多个条件进去实现了。这时候就需要使用Q表达式来实现了。

```
from django.db.models import Q
books = Book.objects.filter(Q(price__lte=10) | Q(rating__lte=9))
```

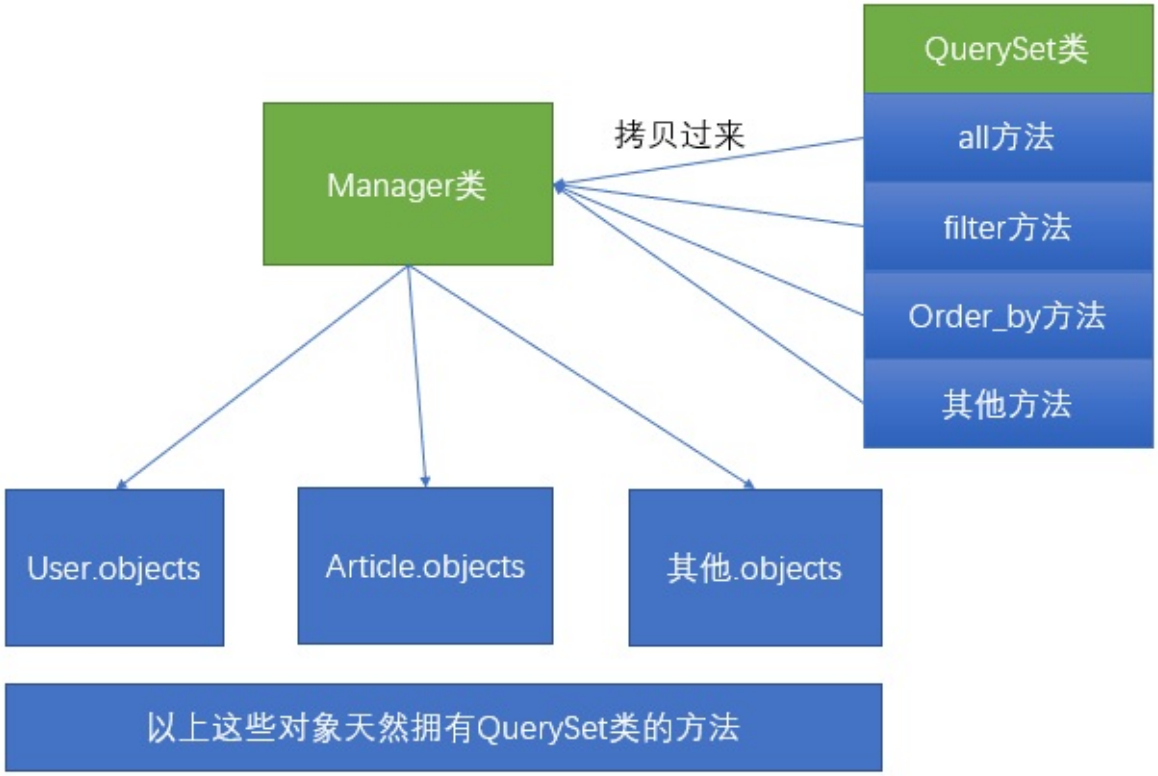
以上是进行或运算，当然还可以进行其他的运算，比如有&和~（非）等。一些用Q表达式的例子如下：

```
from django.db.models import Q
# 获取id等于3的图书
books = Book.objects.filter(Q(id=3))
# 获取id等于3，或者名字中包含文字"传"的图书
books = Book.objects.filter(Q(id=3)|Q(name__contains="传"))
# 获取价格大于100，并且书名中包含"传"的图书
books = Book.objects.filter(Q(price__gte=100)&Q(name__contains="传"))
# 获取书名包含"传"，但是id不等于3的图书
books = Book.objects.filter(Q(name__contains='传') & ~Q(id=3))
```

4-8-QuerySet的方法

QuerySet API

我们通常做查询操作的时候，都是通过模型名字.objects的方式进行操作。其实模型名字.objects是一个django.db.models.manager.Manager对象，而Manager这个类是一个“空壳”的类，他本身是没有任何的属性和方法的。他的方法全部都是通过Python动态添加的方式，从QuerySet类中拷贝过来的



所以我们如果想要学习ORM模型的查找操作，首先要学会QuerySet上的一些API的使用。

QuerySet的方法

在使用QuerySet进行查找操作的时候，可以提供多种操作。比如过滤完后还要根据某个字段进行排序，那么这一系列的操作我们可以通过一个非常流畅的链式调用的方式进行。比如要从文章表中获取标题为123，并且提取后要将结果根据发布的时间进行排序，那么可以使用以下方式来完成

```
articles = Article.objects.filter(title='123').order_by('create_time')
```

可以看到order_by方法是直接在filter执行后调用的。这说明filter返回的对象是一个拥有order_by方法的对象。而这个对象正是一个新的QuerySet对象。因此可以使用order_by方法。

那么以下将介绍在那些会返回新的QuerySet对象的方法。

1. `filter` ：将满足条件的数据提取出来，返回一个新的 `QuerySet`
2. `exclude` ：排除满足条件的数据，返回一个新的 `QuerySet`

```
提取那些标题不包含`hello`的图书
Article.objects.exclude(title__contains='hello')
```

3. `annotate` ：给 `QuerySet` 中的每个对象都添加一个使用查询表达式（聚合函数、F表达式、Q表达式、Func表达式等）的新字段

```
将在每个对象中都添加一个`author__name`的字段，用来显示这个文章的作者的年龄
articles = Article.objects.annotate(author_name=F("author__name"))
```

4. `order_by` ：指定将查询的结果根据某个字段进行排序。如果要倒叙排序，那么可以在这个字段的前面加一个负号

```
 # 根据创建的时间正序排序(从小到大，默认排序规则)
 articles = Article.objects.order_by("create_time")

 # 根据创建的时间倒序排序
 articles = Article.objects.order_by("-create_time")

 # 根据作者的名字进行排序
 articles = Article.objects.order_by("author__name")

 # 首先根据创建的时间进行排序，如果时间相同，则根据作者的名字进行排序
 articles = Article.objects.order_by("create_time",'author__name')

 # 根据图书订单的评分来排序
articles = BookOrder.objects.order_by("book__rating")
```

5. `values` ：用来指定在提取数据出来，需要提取哪些字段。默认情况下会把表中所有的字段全部都提取出来，可以使用 `values` 来进行指定，并且使用了 `values` 方法后，提取出的 `QuerySet` 中的数据类型不是模型，而是在 `values` 方法中指定的字典和值形成的字典

```
articles = Article.objects.values("title",'content')
for article in articles:
```

```
    print(article)
```

以上打印出来的 `article` 是类似于 `{"title":"abc","content":"xxx"}` 的形式

6. `values_list` ：类似于 `values` 。只不过返回的 `QuerySet` 中，存储的不是字典，而是元组

```
articles = Article.objects.values_list("id","title")
print(articles)
```

那么在打印 `articles` 后，结果为 `<QuerySet [(1,'abc'),(2,'xxx'),...]>` 等

7. `all` ：获取这个 `ORM` 模型的 `QuerySet` 对象。

8. `select_related` ：在提取某个模型的数据的同时，也提前将相关联的数据提取出来。比如提取文章数据，可以使用 `select_related` 将 `author` 信息提取出来，以后再次使用 `article.author` 的时候就不需要再次去访问数据库了。可以减少数据库查询的次数

```
article = Article.objects.get(pk=1)
>> article.author # 重新执行一次查询语句
article = Article.objects.select_related("author").get(pk=2)
>> article.author # 不需要重新执行查询语句了
```

`selected_related` 只能用在 `一对多` 或者 `一对一` 中，不能用在 `多对多` 或者 `多对一` 中。比如可以提前获取文章的作者，但是不能通过作者获取这个作者的文章，或者是通过某篇文章获取这个文章所有的标签

9. `prefetch_related` ：这个方法和 `select_related` 非常的类似，就是在访问多个表中的数据的时候，减少查询的次数。这个方法是为了解决 `多对一` 和 `多对多` 的关系的查询问题。比如要获取标题中带有 `hello` 字符串的文章以及他的所有标签

```
from django.db import connection
articles = Article.objects.prefetch_related("tag_set").filter(title__contains='
hello')
print(articles.query) # 通过这条命令查看在底层的SQL语句
for article in articles:
    print("title:",article.title)
    print(article.tag_set.all())
```

10. `create` ：创建一条数据，并且保存到数据库中。这个方法相当于先用指定的模型创建一个对象，然后再调用这个对象的 `save` 方法

```
article = Article(title='abc')
article.save()
# 下面这行代码相当于以上两行代码
article = Article.objects.create(title='abc')
```

11. `get_or_create` ：根据某个条件进行查找，如果找到了那么就返回这条数据，如果没有查找到，那么就创建一个

```
obj,created= Category.objects.get_or_create(title='默认分类')
```

如果有标题等于 `默认分类` 的分类，那么就会查找出来，如果没有，则会创建并且存储到数据库中。
这个方法的返回值是一个元组，元组的第一个参数 `obj` 是这个对象，第二个参数 `created` 代表是否创建的。

12. `exists` ：判断某个条件的数据是否存在。如果要判断某个条件的元素是否存在，那么建议使用 `exists` ，这比使用 `count` 或者直接判断 `QuerySet` 更有效得多。

```
if Article.objects.filter(title__contains='hello').exists():
    print(True)
比使用count更高效：
if Article.objects.filter(title__contains='hello').count() > 0:
    print(True)
也比直接判断QuerySet更高效：
if Article.objects.filter(title__contains='hello'):
    print(True)
```

13. `update` ：执行更新操作，在 `SQL` 底层走的也是 `update` 命令。比如要将所有 `category` 为空的 `article` 的 `article` 字段都更新为默认的分类

```
Article.objects.filter(category__isnull=True).update(category_id=3)
```

注意这个方法走的是更新的逻辑。所以更新完成后保存到数据库中不会执行 `save` 方法，因此不会更新 `auto_now` 设置的字段

14.切片操作：有时候我们查找数据，有可能只需要其中的一部分

```
books = Book.objects.all()[1:3]
for book in books:
    print(book)
```

切片操作并不是把所有数据从数据库中提取出来再做切片操作。而是在数据库层面使用 `LIMIE` 和 `OFFSET` 来帮我们完成。所以如果只需要取其中一部分的数据的时候，建议大家使用切片操作。

将QuerySet转换为SQL去执行

生成一个QuerySet对象并不会马上转换为SQL语句去执行

```
from django.db import connection
books = Book.objects.all()
print(connection.queries)
```

我们可以看到在打印connection.quries的时候打印的是一个空的列表。说明上面的QuerySet并没有真正的执行。

在以下情况下QuerySet会被转换为SQL语句执行

1. 迭代：在遍历QuerySet对象的时候，会首先先执行这个SQL语句，然后再把这个结果返回进行迭代。比如以下代码就会转换为SQL语句：

```
for book in Book.objects.all():
    print(book)
```

2. 使用步长做切片操作：QuerySet可以类似于列表一样做切片操作。做切片操作本身不会执行SQL语句，但是如果如果在做切片操作的时候提供了步长，那么就会立马执行SQL语句。需要注意的是，做切片后不能再执行filter方法，否则会报错。
3. 调用len函数：调用len函数用来获取QuerySet中总共有多少条数据也会执行SQL语句。
4. 调用list函数：调用list函数用来将一个QuerySet对象转换为list对象也会立马执行SQL语句。
5. 判断：如果对某个QuerySet进行判断，也会立马执行SQL语句。

4-9-ORM模型练习

假设有以下ORM模型

```
from django.db import models

class Student(models.Model):
    """学生表"""
    name = models.CharField(max_length=100)
    gender = models.SmallIntegerField()

    class Meta:
        db_table = 'student'

class Course(models.Model):
    """课程表"""
    name = models.CharField(max_length=100)
    teacher = models.ForeignKey("Teacher",on_delete=models.SET_NULL,null=True)
    class Meta:
        db_table = 'course'

class Score(models.Model):
    """分数表"""
    student = models.ForeignKey("Student",on_delete=models.CASCADE)
    course = models.ForeignKey("Course",on_delete=models.CASCADE)
    number = models.FloatField()

    class Meta:
        db_table = 'score'

class Teacher(models.Model):
    """老师表"""
    name = models.CharField(max_length=100)

    class Meta:
        db_table = 'teacher'
```

使用之前学到过的操作实现下面的查询操作：

1.查询平均成绩大于60分的同学的id和平均成绩；

2.查询所有同学的id、姓名、选课的数量、总成绩；

3.查询姓“李”的老师的个数；

4.查询没学过“李老师”课的同学的id、姓名；

5.查询学过课程id为1和2的所有同学的id、姓名；

6.查询所有课程成绩小于60分的同学的id和姓名；

7.查询没有学全所有课的同学的id、姓名；

8.查询所有学生的姓名、平均分，并且按照平均分从高到低排序；

9.查询各科成绩的最高和最低分，以如下形式显示：课程ID，课程名称，最高分，最低分；

10.统计总共有多少女生，多少男生；

11.将“黄老师”的每一门课程都在原来的基础之上加5分；

12.查询两门以上不及格的同学的id、姓名、以及不及格课程数；

13.查询每门课的选课人数；

4-10-ORM模型迁移

迁移命令

1.makemigrations：将模型生成迁移脚本。模型所在的app，必须放在settings.py中的INSTALLED_APPS中。这个命令有以下几个常用选项：

- app_label：后面可以跟一个或者多个app，那么就只会针对这几个app生成迁移脚本。
  如果没有任何的app_label，那么会检查INSTALLED_APPS中所有的app下的模型，针对每一个app都生成响应的迁移脚本。
- --name：给这个迁移脚本指定一个名字。
- --empty：生成一个空的迁移脚本。如果你想写自己的迁移脚本，可以使用这个命令来实现一个空的文件，然后自己再在文件中写迁移脚本。

2.migrate：将新生成的迁移脚本。映射到数据库中。创建新的表或者修改表的结构。以下一些常用的选项：

- app_label：将某个app下的迁移脚本映射到数据库中。如果没有指定，那么会将所有在INSTALLED_APPS中的app下的模型都映射到数据库中。
- app_label migrationname：将某个app下指定名字的migration文件映射到数据库中。
- --fake：可以将指定的迁移脚本名字添加到数据库中。但是并不会把迁移脚本转换为SQL语句，修改数据库中的表。
- --fake-initial：将第一次生成的迁移文件版本号记录在数据库中。但并不会真正的执行迁移脚本。

3.showmigrations：查看某个app下的迁移文件。如果后面没有app，那么将查看INSTALLED_APPS中所有的迁移文件。

```
python manage.py showmigrations [app名字]
```

4.sqlmigrate：查看某个迁移文件在映射到数据库中的时候，转换的SQL语句

```
python manage.py sqlmigrate book 0001_initial
```

migrations中的迁移版本和数据库中的迁移版本对不上怎么办？

1.找到哪里不一致，然后使用python manage.py --fake [版本名字]，将这个版本标记为已经映射。
2.删除指定app下migrations和数据库表django_migrations中和这个app相关的版本号，然后将模型中的字段和数据库中的字段保持一致，再使用命令python manage.py makemigrations重新生成一个初始化的迁移脚本，之后再使用命令python manage.py makemigrations --fake-initial来将这个初始化的迁移脚本标记为已经映射。以后再修改就没有问题了。

根据已有的表自动生成模型

在实际开发中，有些时候可能数据库已经存在了。如果我们用Django来开发一个网站，读取的是之前已经存在的数据库中的数据。那么该如何将模型与数据库中的表映射呢？根据旧的数据库生成对应的ORM模型，需要以下几个步骤：
1.Django给我们提供了一个inspectdb的命令，可以非常方便的将已经存在的表，自动的生成模型。想要使用inspectdb自动将表生成模型。首先需要在settings.py中配置好数据库相关信息。不然就找不到数据库。

```
DATABASES = {
 'default': {
     'ENGINE': 'django.db.backends.mysql',
     'NAME': "migrations_demo",
     'HOST': '127.0.0.1',
     'PORT': '3306',
     'USER': 'root',
     'PASSWORD': 'root'
 }
}

python manage.py inspectdb  > models.py
```

2.修正模型：新生成的ORM模型有些地方可能不太适合使用。比如模型的名字，表之间的关系等等

- 2.1 模型名：自动生成的模型，是根据表的名字生成的，可能不是你想要的。这时候模型的名字你可以改成任何你想要的。
- 2.2 模型所属app：根据自己的需要，将相应的模型放在对应的app中。放在同一个app中也是没有任何问题的。只是不方便管理。
- 2.3 模型外键引用：将所有使用ForeignKey的地方，模型引用都改成字符串。这样不会产生模型顺序的问题。另外，如果引用的模型已经移动到其他的app中了，那么还要加上这个app的前缀。
- 2.4 让Django管理模型：将Meta下的managed=False删掉，如果保留这个，那么以后这个模型有任何的修改，使用migrate都不会映射到数据库中。

- 2.5 当有多对多的时候，应该也要修正模型。将中间表注视了，然后使用ManyToManyField来实现多对多。并且，使用ManyToManyField生成的中间表的名字可能和数据库中那个中间表的名字不一致，这时候肯定就不能正常连接了。那么可以通过db_table来指定中间表的名字。

```
class Article(models.Model):
    title = models.CharField(max_length=100, blank=True, null=True)
    content = models.TextField(blank=True, null=True)
    author = models.ForeignKey('front.User', models.SET_NULL, blank=True, null=
True)
    tags = models.ManyToManyField("Tag",db_table='article_tag')

    class Meta:
        db_table = 'article'
```

- 2.6 表名：切记不要修改表的名字。不然映射到数据库中，会发生找不到对应表的错误

3 执行命令python manage.py makemigrations生成初始化的迁移脚本。方便后面通过ORM来管理表。这时候还需要执行命令python manage.py migrate --fake-initial，因为如果不使用--fake-initial，那么会将迁移脚本会映射到数据库中。这时候迁移脚本会新创建表，而这个表之前是已经存在了的，所以肯定会报错。此时我们只要将这个0001-initial的状态修改为已经映射，而不真正执行映射，下次再migrate的时候，就会忽略他。
4 将Django的核心表映射到数据库中：Django中还有一些核心的表也是需要创建的。不然有些功能是用不了的。比如auth相关表。如果这个数据库之前就是使用Django开发的，那么这些表就已经存在了。可以不用管了。如果之前这个数据库不是使用Django开发的，那么应该使用migrate命令将Django中的核心模型映射到数据库中。

5-视图高级

1-Django限制请求method

常用的请求method

1.GET请求：GET请求一般用来向服务器索取数据，但不会向服务器提交数据，不会对服务器的状态进行更改。比如向服务器获取某篇文章的详情。

2.POST请求：POST请求一般是用来向服务器提交数据，会对服务器的状态进行更改。比如提交一篇文章给服务器。

限制请求装饰器

Django内置的视图装饰器可以给视图提供一些限制。比如这个视图只能通过GET的method访问等。以下将介绍一些常用的内置视图装饰器

1. `django.views.decorators.http.require_http_methods` ：

这个装饰器需要传递一个允许访问的方法的列表

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET"])
def my_view(request):
    pass
```

2.django.views.decorators.http.require_GET：

这个装饰器相当于是require_http_methods(['GET'])的简写形式，只允许使用GET的method来访问视图

```
from django.views.decorators.http import require_GET

@require_GET
def my_view(request):
    pass
```

3.django.views.decorators.http.require_POST：

这个装饰器相当于是require_http_methods(['POST'])的简写形式，只允许使用POST的method来访问视图

```
from django.views.decorators.http import require_POST

@require_POST
def my_view(request):
    pass
```

4.django.views.decorators.http.require_safe：

这个装饰器相当于是require_http_methods(['GET','HEAD'])的简写形式，

只允许使用相对安全的方式来访问视图。因为GET和HEAD不会对服务器产生增删改的行为

```
from django.views.decorators.http import require_safe

@require_safe
def my_view(request):
     pass
```

2-页面重定向

重定向

重定向分为永久性重定向和暂时性重定向，在页面上体现的操作就是浏览器会从一个页面自动跳转到另外一个页面。比如用户访问了一个需要权限的页面，但是该用户当前并没有登录，因此我们应该给他重定向到登录页面。

- 永久性重定向：http的状态码是301，多用于旧网址被废弃了要转到一个新的网址确保用户的访问，最经典的就是京东网站，你输入www.jingdong.com的时候，会被重定向到www.jd.com，因为jingdong.com这个网址已经被废弃了，被改成jd.com，所以这种情况下应该用永久重定向。
- 暂时性重定向：http的状态码是302，表示页面的暂时性跳转。比如访问一个需要权限的网址，如果当前用户没有登录，应该重定向到登录页面，这种情况下，应该用暂时性重定向。

在Django中，重定向是使用redirect(to, *args, permanent=False, **kwargs)来实现的。to是一个url，permanent代表的是这个重定向是否是一个永久的重定向，默认是False。

```
from django.shortcuts import reverse,redirect
def profile(request):
    if request.GET.get("username"):
        return HttpResponse("%s, 欢迎来到个人中心页面！")
    else:
        return redirect(reverse("user:login"))
```

3-HttpRequest对象

WSGIRequest对象

Django在接收到http请求之后，会根据http请求携带的参数以及报文信息创建一个WSGIRequest对象，并且作为视图函数第一个参数传给视图函数。也就是我们经常看到的request参数。在这个对象上我们可以找到客户端上传上来的所有信息。这个对象的完整路径是django.core.handlers.wsgi.WSGIRequest。

WSGIRequest对象常用属性

WSGIRequest对象上大部分的属性都是只读的。因为这些属性是从客户端上传上来的，没必要做任何的修改。

1. `path` ：请求服务器的完整“路径”，但不包含域名和参数。比如 `http://www.baidu.com/xxx/yyy/` ，那么 `path` 就是 `/xxx/yyy/` 。
2. `method` ：代表当前请求的 `http` 方法。比如是 `GET` 还是 `POST` 。
3. `GET` ：一个 `django.http.request.QueryDict` 对象。操作起来类似于字典。这个属性中包含了所有以 `?xxx=xxx` 的方式上传上来的参数。
4. `POST` ：也是一个 `django.http.request.QueryDict` 对象。这个属性中包含了所有以 `POST` 方式上传上来的参数。
5. `FILES` ：也是一个 `django.http.request.QueryDict` 对象。这个属性中包含了所有上传的文件。
6. `COOKIES` ：一个标准的Python字典，包含所有的 `cookie` ，键值对都是字符串类型。
7. `session` ：一个类似于字典的对象。用来操作服务器的 `session` 。
8. `META` ：存储的客户端发送上来的所有 `header` 信息。
9. `CONTENT_LENGTH` ：请求的正文的长度（是一个字符串）。
10. `CONTENT_TYPE` ：请求的正文的MIME类型。
11. `HTTP_ACCEPT` ：响应可接收的Content-Type。
12. `HTTP_ACCEPT_ENCODING` ：响应可接收的编码。
13. `HTTP_ACCEPT_LANGUAGE` ： 响应可接收的语言。
14. `HTTP_HOST` ：客户端发送的HOST值。

15. `HTTP_REFERER` ：在访问这个页面上一个页面的url。
16. `QUERY_STRING` ：单个字符串形式的查询字符串（未解析过的形式）。
17. `REMOTE_ADDR` ：客户端的IP地址。如果服务器使用了 `nginx` 做反向代理或者负载均衡，那么这个值返回的是 `127.0.0.1` ，这时候可以使用 `HTTP_X_FORWARDED_FOR` 来获取，所以获取 `ip` 地址的代码片段如下：

```
if request.META.has_key('HTTP_X_FORWARDED_FOR'):
      ip =  request.META['HTTP_X_FORWARDED_FOR']
  else:
      ip = request.META['REMOTE_ADDR']
```

18. `REMOTE_HOST` ：客户端的主机名。
19. `REQUEST_METHOD` ：请求方法。一个字符串类似于 `GET` 或者 `POST` 。
20. `SERVER_NAME` ：服务器域名。
21. `SERVER_PORT` ：服务器端口号，是一个字符串类型。

WSGIRequest对象常用方法

1. `is_secure()` ：是否是采用 `https` 协议。
2. `is_ajax()` ：是否采用 `ajax` 发送的请求。原理就是判断请求头中是否存在 `X-Requested-With:XMLHttpRequest` 。
3. `get_host()` ：服务器的域名。如果在访问的时候还有端口号，那么会加上端口号。比如 `www.baidu.com:9000` 。
4. `get_full_path()` ：返回完整的path。如果有查询字符串，还会加上查询字符串。比如 `/music/bands/?print=True` 。
5. `get_raw_uri()` ：获取请求的完整 `url` 。

4-HttpResponse对象

HttpResponse对象

Django服务器接收到客户端发送过来的请求后，会将提交上来的这些数据封装成一个HttpRequest对象传给视图函数。那么视图函数在处理完相关的逻辑后，也需要返回一个响应给浏览器。而这个响应，我们必须返回HttpResponseBase或者他的子类的对象。而HttpResponse则是HttpResponseBase用得最多的子类

常用属性

1.content：返回的内容。

```
response = HttpResponse()
response.content = "首页"
return response
```

2.status_code：返回的HTTP响应状态码。

3.content_type：返回的数据的MIME类型，默认为text/html。浏览器会根据这个属性，来显示数据。如果是text/html，那么就会解析这个字符串，如果text/plain，那么就会显示一个纯文本。常用的Content-Type如下：

```
    text/html（默认的，html文件）
    text/plain（纯文本）
    text/css（css文件）
    text/javascript（js文件）
    multipart/form-data（文件提交）
    application/json（json传输）
    application/xml（xml文件）
```

4.设置请求头：
`response['X-Access-Token'] = 'xxxx'。`

常用方法

1.set_cookie：用来设置cookie信息。

2.delete_cookie：用来删除cookie信息。

3.write：HttpResponse是一个类似于文件的对象，可以用来写入数据到数据体（content）中。

JsonResponse类

用来对象dump成json字符串，然后返回将json字符串封装成Response对象返回给浏览器。并且他的Content-Type是application/json。

```
from django.http import JsonResponse
def index(request):
    return JsonResponse({"username":"juran","age":18})
```

默认情况下JsonResponse只能对字典进行dump，如果想要对非字典的数据进行dump，那么需要给JsonResponse传递一个safe=False参数。

```
from django.http import JsonResponse
def index(request):
    persons = ['张三','李四','王五']
    return JsonResponse(persons)
```

以上代码会报错，应该在使用HttpResponse的时候，传入一个safe=False参数

```
return JsonResponse(persons,safe=False,json_dumps_params={'ensure_ascii':False})
```

5-类视图

类视图

在写视图的时候，Django除了使用函数作为视图，也可以使用类作为视图。使用类视图可以使用类的一些特性，比如继承等。

View

django.views.generic.base.View是主要的类视图，所有的类视图都是继承自他。如果我们写自己的类视图，也可以继承自他。然后再根据当前请求的method，来实现不同的方法。比如这个视图只能使用get的方式来请求，那么就可以在这个类中定义get(self,request,*args,**kwargs)方法。以此类推，如果只需要实现post方法，那么就只需要在类中实现post(self,request,*args,**kwargs)。

```
from django.views import View
class BookDetailView(View):
    def get(self,request,*args,**kwargs):
        return render(request,'detail.html')
```

类视图写完后，还应该在urls.py中进行映射，映射的时候就需要调用View的类方法as_view()来进行转换。自动查找指定方法。

```
urlpatterns = [
    path("detail/<book_id>/",views.BookDetailView.as_view(),name='detail')
]
```

除了get方法，View还支持以下方法['get','post','put','patch','delete','head','options','trace']。
如果用户访问了View中没有定义的方法。比如你的类视图只支持get方法，而出现了post方法，那么就会把这个请求转发给http_method_not_allowed(request,*args,**kwargs)。

```
class AddBookView(View):
    def post(self,request,*args,**kwargs):
        return HttpResponse("书籍添加成功！")

    def http_method_not_allowed(self, request, *args, **kwargs):
        return HttpResponse("您当前采用的method是：%s，本视图只支持使用post请求！" %
request.method)
```

urls.py中的映射如下

```
path("addbook/",views.AddBookView.as_view(),name='add_book')
```

TemplateView

django.views.generic.base.TemplateView，这个类视图是专门用来返回模版的。在这个类中，有两个属性是经常需要用到的，一个是template_name，这个属性是用来存储模版的路径，TemplateView会自动的渲染这个变量指向的模版。另外一个是get_context_data，这个方法是用来返回上下文数据的，也就是在给模版传的参数的。

```
from django.views.generic.base import TemplateView

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['username'] = "juran"
        return context
```

在urls.py中的映射代码如下

```
from django.urls import path

from myapp.views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

如果在模版中不需要传递任何参数，那么可以直接只在urls.py中使用TemplateView来渲染模版。

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    path('about/', TemplateView.as_view(template_name="about.html")),
]
```

ListView

在网站开发中，经常会出现需要列出某个表中的一些数据作为列表展示出来。比如文章列表，图书列表等等。在Django中可以使用ListView来帮我们快速实现这种需求。

```
class ArticleListView(ListView):
    model = Article
    template_name = 'article_list.html'
    paginate_by = 10
    context_object_name = 'articles'
    ordering = 'create_time'
    page_kwarg = 'page'

    def get_context_data(self, **kwargs):
        context = super(ArticleListView, self).get_context_data(**kwargs)
        print(context)
        return context

    def get_queryset(self):
        return Article.objects.filter(id__lte=89)
```

对以上代码进行解释:

1. 首先 `ArticleListView` 是继承自 `ListView` 。
2. `model` ：重写 `model` 类属性，指定这个列表是给哪个模型的。
3. `template_name` ：指定这个列表的模板。
4. `paginate_by` ：指定这个列表一页中展示多少条数据。
5. `context_object_name` ：指定这个列表模型在模板中的参数名称。
6. `ordering` ：指定这个列表的排序方式。
7. `page_kwarg` ：获取第几页的数据的参数名称。默认是 `page` 。
8. `get_context_data` ：获取上下文的数据。
9. `get_queryset` ：如果你提取数据的时候，并不是要把所有数据都返回，那么你可以重写这个方法。将一些不需要展示的数据给过滤掉。

Paginator和Page类

Paginator和Page类都是用来做分页的。他们在Django中的路径为django.core.paginator.Paginator和django.core.paginator.Page。以下对这两个类的常用属性和方法做解释：

Paginator常用属性和方法

```
count：总共有多少条数据。
num_pages：总共有多少页。
page_range：页面的区间。比如有三页，那么就range(1,4)。
```

Page常用属性和方法

```
has_next：是否还有下一页。
has_previous：是否还有上一页。
next_page_number：下一页的页码。
previous_page_number：上一页的页码。
number：当前页。
start_index：当前这一页的第一条数据的索引值。
end_index：当前这一页的最后一条数据的索引值。
```

示例分页代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
</head>
<body>
    <ul>
        {% for i in articles %}
        <li>{{ i.id }}-{{ i.name }}</li>
        {% endfor %}
    </ul>

    <ul class="pagination">
        {% if page_obj.has_previous %}
        <li>
          <a href="{% url 'list' %}?page={{ page_obj.previous_page_number }}" aria-label="Previous">
            <span aria-hidden="true">&laquo;</span>
          </a>
        </li>
```

```
        {% else %}
        <li class="disabled">
          <a href="javascript:void(0)" aria-label="Previous">
            <span aria-hidden="true">&laquo;</span>
          </a>
        </li>
        {% endif %}
        {% for i in paginator.page_range %}
            {% if page_obj.number == i %}
                <li class="active"><a href="{% url 'list' %}?page={{ i }}">{{ i
}}</a></li>
            {% else %}
                <li><a href="{% url 'list' %}?page={{ i }}">{{ i }}</a></li>
            {% endif %}
        {% endfor %}
        <!--<li><a href="#">2</a></li>-->
        <!--<li><a href="#">3</a></li>-->
        <!--<li><a href="#">4</a></li>-->
        <!--<li><a href="#">5</a></li>-->
        {% if page_obj.has_next %}
        <li>
          <a href="{% url 'list' %}?page={{ page_obj.next_page_number }}" aria-
label="Next">
            <span aria-hidden="true">&raquo;</span>
          </a>
        </li>
        {% else %}
        <li class="disabled">
          <a href="#" aria-label="Next">
            <span aria-hidden="true">&raquo;</span>
          </a>
        </li>
        {% endif %}
    </ul>
</body>
</html>
```

通用分页代码

article1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/d
ist/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww
```

```
7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
</head>
<body>
    <ul>
        {% for i in articles %}
        <li>{{ i.id }}-{{ i.name }}</li>
        {% endfor %}
    </ul>

    <ul class="pagination">
        {% if page_obj.has_previous %}
        <li>
          <a href="{% url 'list' %}?page={{ page_obj.previous_page_number }}" a
ria-label="Previous">
            <span aria-hidden="true">&laquo;</span>
          </a>
        </li>
        {% else %}
        <li class="disabled">
          <a href="javascript:void(0)" aria-label="Previous">
            <span aria-hidden="true">&laquo;</span>
          </a>
        </li>
        {% endif %}

        {% if left_has_more %}
            <li><a href="{% url 'list' %}?page=1">1</a></li>
            <li><a href="javascript:void(0)">...</a></li>
        {% endif %}

        {% for i in left_range %}
            <li><a href="{% url 'list' %}?page={{ i }}">{{ i }}</a></li>
        {% endfor %}

        <li><a href="">{{ page_obj.number }}</a></li>

        {% for i in right_range %}
            <li><a href="{% url 'list' %}?page={{ i }}">{{ i }}</a></li>
        {% endfor %}

        {% if right_has_more %}
            <li><a href="javascript:void(0)">...</a></li>
            <li><a href="{% url 'list' %}?page={{ paginator.num_pages }}">{{ pa
ginator.num_pages }}</a></li>
        {% endif %}


        {% if page_obj.has_next %}
        <li>
          <a href="{% url 'list' %}?page={{ page_obj.next_page_number }}" aria-
```

```
label="Next">
            <span aria-hidden="true">&raquo;</span>
          </a>
        </li>
        {% else %}
        <li class="disabled">
          <a href="#" aria-label="Next">
            <span aria-hidden="true">&raquo;</span>
          </a>
        </li>
        {% endif %}
    </ul>
</body>
</html>
```

views.py

```
class ArticleListVies(ListView):
    model = Publisher
    template_name = 'article_list1.html'
    paginate_by = 5
    context_object_name = 'articles'
    # ordering = 'create_time'
    page_kwarg = 'page'

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super(ArticleListVies, self).get_context_data(**kwargs)
        paginator = context.get("paginator")
        page_obj = context.get("page_obj")
        # print(paginator.count)
        # print(page_obj.has_next())
        paginator_date = self.get_page(paginator,page_obj)
        context.update(paginator_date)
        return context

    def get_page(self,paginator,page_obj,page_offset=2):
        current_page = page_obj.number
        left_has_more = False
        right_has_more = False
        # 3 4 5 6 7
        if current_page <= page_offset + 2:
            left_range = range(1,current_page)
        else:
            left_has_more = True
            left_range = range(current_page-page_offset,current_page)

        #  7     10
        if current_page >= paginator.num_pages - page_offset - 1:
            right_range = range(current_page+1, paginator.num_pages+1)
```

```
        else:
            right_has_more = True
            right_range = range(current_page+1,current_page+page_offset+1)

        return {
            'left_range':left_range,
            'right_range':right_range,
            'right_has_more':right_has_more,
            'left_has_more':left_has_more
        }
```

6-错误处理

错误处理

在一些网站开发中。经常会需要捕获一些错误，然后将这些错误返回比较优美的界面，或者是将这个错误的请求做一些日志保存。

常用的错误码

```
404：服务器没有指定的url。
403：没有权限访问相关的数据。
405：请求的method错误。
400：bad request，请求的参数错误。
500：服务器内部错误，一般是代码出bug了。
502：一般部署的时候见得比较多，一般是nginx启动了，然后uwsgi有问题
```

自定义错误模板

在碰到比如404，500错误的时候，想要返回自己定义的模板。那么可以直接在templates文件夹下创建相应错误代码的html模板文件。那么以后在发生相应错误后，会将指定的模板返回回去。

```
修改配置文件

DEBUG = False
ALLOWED_HOSTS = ["127.0.0.1"]
```

错误处理的解决方案

对于404和500这种自动抛出的错误。我们可以直接在templates文件夹下新建相应错误代码的模板文件。而对于其他的错误，我们可以专门定义一个app，用来处理这些错误

```
views.py
from django.http import HttpResponse
from django.shortcuts import render
```

```
def view_405(request):
    return render(request,"errors/405.html",status=405)

urls.py
from django.urls import path
from . import views
urlpatterns = [
    path("405",views.view_405,name="405")
]
```

6-表单

1-用表单验证数据

表单

HTML中的表单：

单纯从前端的html来说，表单是用来提交数据给服务器的,不管后台的服务器用的是Django还是PHP语言还是其他语言。只要把input标签放在form标签中，然后再添加一个提交按钮，那么以后点击提交按钮，就可以将input标签中对应的值提交给服务器了。

Django中的表单

Django中的表单丰富了传统的HTML语言中的表单。在Django中的表单，主要做以下两件事

1. 渲染表单模板。
2. 表单验证数据是否合法。

Django中表单使用流程

在讲解Django表单的具体每部分的细节之前。我们首先先来看下整体的使用流程。
首先我们在后台服务器定义一个表单类，继承自django.forms.Form

```
# forms.py
class MessageBoardForm(forms.Form):
    title = forms.CharField(max_length=3,label='标题',min_length=2,error_messag
es={"min_length":'标题字符段不符合要求！'})
    content = forms.CharField(widget=forms.Textarea,label='内容',error_messages=
{"required":'content字段必须填写！'})
    email = forms.EmailField(label='邮箱')
    reply = forms.BooleanField(required=False,label='回复')
```

然后在视图中，根据是GET还是POST请求来做相应的操作。如果是GET请求，那么返回一个空的表单，如果是POST请求，那么将提交上来的数据进行校验。

```
from .forms import MessageForm
from django.views import View
from django.forms.utils import ErrorDict

class IndexView(View):
    def get(self,request):
        form = MessageBoardForm()
        return render(request,'index.html',{'form':form})

    def post(self,request):
        form = MessageBoardForm(request.POST)
        if form.is_valid():
            title = form.cleaned_data.get('title')
            content = form.cleaned_data.get('content')
            email = form.cleaned_data.get('email')
            reply = form.cleaned_data.get('reply')
            return HttpResponse('success')
        else:
            print(form.errors)
            return HttpResponse('fail')
```

在使用GET请求的时候，我们传了一个form给模板，那么以后模板就可以使用form来生成一个表单的html代码。在使用POST请求的时候，我们根据前端上传上来的数据，构建一个新的表单，这个表单是用来验证数据是否合法的，如果数据都验证通过了，那么我们可以通过cleaned_data来获取相应的数据。在模板中渲染表单的HTML

```
<form action="" method="post">
    <table>
        {{ form.as_table }}
        <tr>
            <td></td>
            <td><input type="submit" value="提交"></td>
        </tr>
    </table>
</form>
```

我们在最外面给了一个form标签，然后在里面使用了table标签来进行美化，在使用form对象渲染的时候，使用的是table的方式，当然还可以使用ul的方式（as_ul），也可以使用p标签的方式（as_p），并且在后面我们还加上了一个提交按钮。这样就可以生成一个表单了

常用的Field

使用Field可以是对数据验证的第一步。你期望这个提交上来的数据是什么类型，那么就使用什么类型的Field。

CharField

用来接收文本。

```
参数：
max_length：这个字段值的最大长度。
min_length：这个字段值的最小长度。
required：这个字段是否是必须的。默认是必须的。
error_messages：在某个条件验证失败的时候，给出错误信息。
```

EmailField

```
用来接收邮件，会自动验证邮件是否合法。
错误信息的key：required、invalid。
```

FloatField

用来接收浮点类型，并且如果验证通过后，会将这个字段的值转换为浮点类型。

```
参数：
max_value：最大的值。
min_value：最小的值。
错误信息的key：required、invalid、max_value、min_value。
```

IntegerField

用来接收整形，并且验证通过后，会将这个字段的值转换为整形。

```
参数：
max_value：最大的值。
min_value：最小的值。
错误信息的key：required、invalid、max_value、min_value。
```

URLField

```
用来接收url格式的字符串。
错误信息的key：required、invalid。
```

常用验证器

在验证某个字段的时候，可以传递一个validators参数用来指定验证器，进一步对数据进行过滤。验证器有很多，但是很多验证器我们其实已经通过这个Field或者一些参数就可以指定了。比如EmailValidator，我们可以通过EmailField来指定，比如MaxValueValidator，我们可以通过max_value参数来指定。以下是一些常用的验证器：

```
MaxValueValidator：验证最大值。
MinValueValidator：验证最小值。
MinLengthValidator：验证最小长度。
MaxLengthValidator：验证最大长度。
EmailValidator：验证是否是邮箱格式。
URLValidator：验证是否是URL格式。
RegexValidator：如果还需要更加复杂的验证，那么我们可以通过正则表达式的验证器：RegexValida
tor。比如现在要验证手机号码是否合格，那么我们可以通过以下代码实现：
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678
]\d{9}",message='请输入正确格式的手机号码！')])
```

自定义验证

有时候对一个字段验证，不是一个长度，一个正则表达式能够写清楚的，还需要一些其他复杂的逻辑，那么我们可以对某个字段，进行自定义的验证。比如在注册的表单验证中，我们想要验证手机号码是否已经被注册过了，那么这时候就需要在数据库中进行判断才知道。对某个字段进行自定义的验证方式是，定义一个方法，这个方法的名字定义规则是：clean_fieldname。如果验证失败，那么就抛出一个验证错误。比如要验证用户表中手机号码之前是否在数据库中存在，那么可以通过以下代码实现：

```
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678
]\d{9}",message='请输入正确格式的手机号码！')])

    def clean_telephone(self):
        telephone = self.cleaned_data.get('telephone')
        exists = User.objects.filter(telephone=telephone).exists()
        if exists:
            raise forms.ValidationError("手机号码已经存在！")
        return telephone
```

以上是对某个字段进行验证，如果验证数据的时候，需要针对多个字段进行验证，那么可以重写clean方法。比如要在注册的时候，要判断提交的两个密码是否相等。那么可以使用以下代码来完成：

```
class MyForm(forms.Form):
    telephone = forms.CharField(validators=[validators.RegexValidator("1[345678
]\d{9}",message='请输入正确格式的手机号码！')])
    pwd1 = forms.CharField(max_length=12)
    pwd2 = forms.CharField(max_length=12)

    def clean(self):
        cleaned_data = super().clean()
        pwd1 = cleaned_data.get('pwd1')
        pwd2 = cleaned_data.get('pwd2')
        if pwd1 != pwd2:
```

```
            raise forms.ValidationError('两个密码不一致！')
```

提取错误信息

如果验证失败了，那么有一些错误信息是我们需要传给前端的。这时候我们可以通过以下属性来获取：

1.form.errors：这个属性获取的错误信息是一个包含了html标签的错误信息。

2.form.errors.get_json_data()：这个方法获取到的是一个字典类型的错误信息。将某个字段的名字作为key，错误信息作为值的一个字典。

3.form.as_json()：这个方法是将form.get_json_data()返回的字典dump成json格式的字符串，方便进行传输。

4.上述方法获取的字段的错误值，都是一个比较复杂的数据。比如以下：

```
{'username': [{'message': 'Enter a valid URL.', 'code': 'invalid'}, {'message': 'Ensure this value has at most 4 characters (it has 22).', 'code': 'max_length'}]}
```

那么如果我只想把错误信息放在一个列表中，而不要再放在一个字典中。这时候我们可以定义一个方法，把这个数据重新整理一份。

```
class MyForm(forms.Form):
    username = forms.URLField(max_length=4)

    def get_errors(self):
        errors = self.errors.get_json_data()
        new_errors = {}
        for key,message_dicts in errors.items():
            messages = []
            for message in message_dicts:
                messages.append(message['message'])
            new_errors[key] = messages
        return new_errors
```

这样就可以把某个字段所有的错误信息直接放在这个列表中。

2-ModelForm

ModelForm

大家在写表单的时候，会发现表单中的Field和模型中的Field基本上是一模一样的，而且表单中需要验证的数据，也就是我们模型中需要保存的。那么这时候我们就可以将模型中的字段和表单中的字段进行绑定。
比如现在有个Article的模型。

```
from django.db import models
from django.core import validators
class Article(models.Model):
    title = models.CharField(max_length=10,validators=[validators.MinLengthVali
dator(limit_value=3)])
    content = models.TextField()
    author = models.CharField(max_length=100)
    category = models.CharField(max_length=100)
    create_time = models.DateTimeField(auto_now_add=True)
```

那么在写表单的时候，就不需要把Article模型中所有的字段都一个个重复写一遍了。

```
from django import forms
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = "__all__"
```

MyForm是继承自forms.ModelForm，然后在表单中定义了一个Meta类，在Meta类中指定了model=Article，以及fields="all"，这样就可以将Article模型中所有的字段都复制过来，进行验证。如果只想针对其中几个字段进行验证，那么可以给fields指定一个列表，将需要的字段写进去。比如只想验证title和content，那么可以使用以下代码实现

```
from django import forms
```

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = ['title','content']
```

如果要验证的字段比较多，只是除了少数几个字段不需要验证，那么可以使用exclude来代替fields。比如我不想验证category，那么示例代码如下：

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ['category']
```

自定义错误消息

使用ModelForm，因为字段都不是在表单中定义的，而是在模型中定义的，因此一些错误消息无法在字段中定义。那么这时候可以在Meta类中，定义error_messages，然后把相应的错误消息写到里面去。

```
class MyForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ['category']
        error_messages  ={
            'title':{
                'max_length': '最多不能超过10个字符！',
                'min_length': '最少不能少于3个字符！'
            },
            'content': {
                'required': '必须输入content！',
            }
        }
```

save方法

ModelForm还有save方法，可以在验证完成后直接调用save方法，就可以将这个数据保存到数据库中了

```
form = MyForm(request.POST)
if form.is_valid():
    form.save()
    return HttpResponse('succes')
else:
    print(form.get_errors())
    return HttpResponse('fail')
```

这个方法必须要在clean没有问题后才能使用，如果在clean之前使用，会抛出异常。另外，我们在调用save方法

的时候，如果传入一个commit=False，那么只会生成这个模型的对象，而不会把这个对象真正的插入到数据库中。比如表单上验证的字段没有包含模型中所有的字段，这时候就可以先创建对象，再根据填充其他字段，把所有字段的值都补充完成后，再保存到数据库中。

```
form = MyForm(request.POST)
if form.is_valid():
    article = form.save(commit=False)
    article.category = 'Python'
    article.save()
    return HttpResponse('succes')
else:
    print(form.get_errors())
    return HttpResponse('fail')
```

3-文件上传

文件上传

前端HTML代码实现

1.在前端中，我们需要填入一个form标签，然后在这个form标签中指定enctype="multipart/form-data"，不然就不能上传文件。

2.在form标签中添加一个input标签，然后指定input标签的name，以及type="file"。

```
<form action="" method="post" enctype="multipart/form-data">
    <input type="file" name="myfile">
</form>
```

后端的代码实现

后端的主要工作是接收文件。然后存储文件。接收文件的方式跟接收POST的方式是一样的，只不过是通过FILES来实现。

```
def save_file(file):
    with open('somefile.txt','wb') as fp:
        for chunk in file.chunks():
            fp.write(chunk)

    # with open("abc.txt","w") as f:
        # f.write(files.read().decode())

def index(request):
    if request.method == 'GET':
        form = MyForm()
        return render(request,'index.html',{'form':form})
    else:
        myfile = request.FILES.get('myfile')
        save_file(myfile)
        return HttpResponse('success')
```

以上代码通过request.FILES接收到文件后，再写入到指定的地方。这样就可以完成一个文件的上传功能了。

使用模型来处理上传的文件

在定义模型的时候，我们可以给存储文件的字段指定为FileField，这个Field可以传递一个upload_to参数，用来指定上传上来的文件保存到哪里。比如我们让他保存到项目的files文件夹下，那么示例代码如下：

```
# models.py
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.FileField(upload_to="files")


# views.py
def index(request):
    if request.method == 'GET':
        return render(request,'index.html')
    else:
        title = request.POST.get('title')
        content = request.POST.get('content')
        thumbnail = request.FILES.get('thumbnail')
        article = Article(title=title, content=content, thumbnail=thumbnail)
        article.save()
        return HttpResponse('success')
```

调用完article.save()方法，就会把文件保存到files下面，并且会将这个文件的路径存储到数据库中。

指定MEDIA_ROOT和MEDIA_URL

以上我们是使用了upload_to来指定上传的文件的目录。我们也可以指定MEDIA_ROOT，就不需要在FielField中指定upload_to，他会自动的将文件上传到MEDIA_ROOT的目录下。

```
MEDIA_ROOT = os.path.join(BASE_DIR,'media')
MEDIA_URL = '/media/'
```

然后我们可以在urls.py中添加MEDIA_ROOT目录下的访问路径。示例代码如下：

```
from django.urls import path
from front import views
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
```

```
    path('', views.index),
] + static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

如果我们同时指定MEDIA_ROOT和upload_to，那么会将文件上传到MEDIA_ROOT下的upload_to文件夹中。示例代码如下：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.FileField(upload_to="%Y/%m/%d/")
```

限制上传的文件拓展名

如果想要限制上传的文件的拓展名，那么我们就需要用到表单来进行限制。我们可以使用普通的Form表单，也可以使用ModelForm，直接从模型中读取字段。

```
# models.py
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnial = models.FileField(upload_to='%Y/%m/%d/',validators=[validators.F
ileExtensionValidator(['txt','pdf'])])
    files = models.ImageField(upload_to="%Y%m%d")

# forms.py
class BookForms(forms.Form):
    files = forms.FileField(validators=[validators.FileExtensionValidator(['txt'
],message="必须是TXT")])
    # files = forms.ImageField(error_messages={"invalid_image":"格式不对"})

# views.py
class uploadfile(View):
    def get(self,request):
        return render(request,"upload.html")


    def post(self,request):
        form = BookForms(request.POST,request.FILES)
        if form.is_valid():
            title = request.POST.get('title')
            files = request.FILES.get('files')
            FilesUpload.objects.create(title=title,files=files)
            return HttpResponse("success")
        else:
            print(form.errors.get_json_data())
            return HttpResponse("fail")
```

上传图片

上传图片跟上传普通文件是一样的。只不过是上传图片的时候Django会判断上传的文件是否是图片的格式（除了判断后缀名，还会判断是否是可用的图片）。如果不是，那么就会验证失败。我们首先先来定义一个包含ImageField的模型。

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    thumbnail = models.ImageField(upload_to="%Y/%m/%d/")
```

因为要验证是否是合格的图片，因此我们还需要用一个表单来进行验证。表单我们直接就使用ModelForm就可以了

```
class BookForms(forms.Form):

    files = forms.ImageField(error_messages={"invalid_image":"格式不对"})
```

注意：使用ImageField，必须要先安装Pillow库：pip install pillow

7-session和cookie

1-session和cookie

cookie和session

1.cookie：在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的cookie数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。cookie存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用cookie只能存储一些小量的数据
2.session: session和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，cookie是存储在本地浏览器，session是一个思路、一个概念、一个服务器存储授权信息的解决方案，不同的服务器，不同的框架，不同的语言有不同的实现。虽然实现不一样，但是他们的目的都是服务器为了方便存储数据的。session的出现，是为了解决cookie存储数据不安全的问题的。
3.cookie和session使用：web开发发展至今，cookie和session的使用已经出现了一些非常成熟的方案。在如今的市场或者企业里，一般有两种存储方式

> 3.1 存储在服务端：通过cookie存储一个sessionid，然后具体的数据则是保存在session中。如果用户已经登录，则服务器会在cookie中保存一个sessionid，下次再次请求的时候，会把该sessionid携带上来，服务器根据sessionid在session库中获取用户的session数据。就能知道该用户到底是谁，以及之前保存的一些状态信息。这种专业术语叫做server side session。Django把session信息默认存储到数据库中，当然也可以存储到其他地方，比如缓存中，文件系统中等。存储在服务器的数据会更加的安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，一些session信息还是绰绰有余的。

> 3.2 将session数据加密，然后存储在cookie中。这种专业术语叫做client side session。flask框架默认采用的就是这种方式，但是也可以替换成其他形式。

在django中操作cookie和session

操作cookie

设置cookie

设置cookie是设置值给浏览器的。因此我们需要通过response的对象来设置，设置cookie可以通过response.set_cookie来设置，这个方法的相关参数如下：

```
key：这个cookie的key。
value：这个cookie的value。
max_age：最长的生命周期。单位是秒。
expires：过期时间。跟max_age是类似的，只不过这个参数需要传递一个具体的日期，比如datetime或者是符合日期格式的字符串。如果同时设置了expires和max_age，那么将会使用expires的值作为过期时间。
path：对域名下哪个路径有效。默认是对域名下所有路径都有效。
domain：针对哪个域名有效。默认是针对主域名下都有效，如果只要针对某个子域名才有效，那么可以设置这个属性.
secure：是否是安全的，如果设置为True，那么只能在https协议下才可用。
httponly：默认是False。如果为True，那么在客户端不能通过JavaScript进行操作
```

```
from datetime import datetime
from django.utils.timezone import make_aware

def cookie_test(request):
    response = HttpResponse("index")
    expires = make_aware(datetime(year=2018,month=12,day=27,hour=3,minute=20,se
cond=0))
    response.set_cookie("username","juran",expires=expires,path="/cms/")

    return response

def get_cookie_test(request):
    cookies = request.COOKIES
    username = cookies.get('username')
    return HttpResponse(username)
```

删除cookie

通过delete_cookie即可删除cookie。实际上删除cookie就是将指定的cookie的值设置为空的字符串，然后使用将他的过期时间设置为0，也就是浏览器关闭后就过期。

```
def delete_cookie(request):
    response = HttpResponse('delete')
    response.delete_cookie('username')
    return response
```

获取cookie

获取浏览器发送过来的cookie信息。可以通过request.COOKIES来或者。这个对象是一个字典类型。比如获取所有的cookie

```
cookies = request.COOKIES
for cookie_key,cookie_value in cookies.items():
   print(cookie_key,cookie_value)
```

操作session

django中的session默认情况下是存储在服务器的数据库中的，在表中会根据sessionid来提取指定的session数据，然后再把这个sessionid放到cookie中发送给浏览器存储，浏览器下次在向服务器发送请求的时候会自动的把所有cookie信息都发送给服务器，服务器再从cookie中获取sessionid，然后再从数据库中获取session数据。但是我们在操作session的时候，这些细节压根就不用管。我们只需要通过request.session即可操作。

```
def index(request):
    request.session['username'] = 'jr'
    request.session.get('username')
    return HttpResponse('index')
```

session常用的方法如下：

1. `get` ：用来从 `session` 中获取指定值。
2. `pop` ：从 `session` 中删除一个值。
3. `keys` ：从 `session` 中获取所有的键。
4. `items` ：从 `session` 中获取所有的值。
5. `clear` ：清除当前这个用户的 `session` 数据。
6. `flush` ：删除 `session` 并且删除在浏览器中存储的 `session_id` ，一般在注销的时候用得比较多。
7. `set_expiry(value)` ：设置过期时间。

   整形：代表秒数，表示多少秒后过期。

   `0` ：代表只要浏览器关闭， `session` 就会过期。

   `None` ：会使用全局的 `session` 配置。在 `settings.py` 中可以设置 `SESSION_COOKIE_AGE` 来配置全局的过期时间。默认是 `1209600` 秒，也就是2周的时间。

   -1:代表已经过期
8. `clear_expired` ：清除过期的 `session` 。 `Django` 并不会清除过期的 `session` ，需要定期手动的清理，或者是在终端，使用命令行 `python manage.py clearsessions` 来清除过期的 `session` 。

修改session的存储机制

默认情况下，session数据是存储到数据库中的。当然也可以将session数据存储到其他地方。可以通过设置SESSION_ENGINE来更改session的存储位置，这个可以配置为以下几种方案

1. django.contrib.sessions.backends.db：使用数据库。默认就是这种方案。
2. django.contrib.sessions.backends.file：使用文件来存储session。
3. django.contrib.sessions.backends.cache：使用缓存来存储session。想要将数据存储到缓存中，前提是你必须要在settings.py中配置好CACHES，并且是需要使用Memcached，而不能使用纯内存作为缓存。
4. django.contrib.sessions.backends.cached_db：在存储数据的时候，会将数据先存到缓存中，再存到数据库中。这样就可以保证万一缓存系统出现问题，session数据也不会丢失。在获取数据的时候，会先从缓存中获取，如果缓存中没有，那么就会从数据库中获取。
5. django.contrib.sessions.backends.signed_cookies：将session信息加密后存储到浏览器的cookie中。这种方式要注意安全，建议设置SESSION_COOKIE_HTTPONLY=True，那么在浏览器中不能通过js来操作session数据，并且还需要对settings.py中的SECRET_KEY进行保密，因为一旦别人知道这个SECRET_KEY，那么就可以进行解密。另外还有就是在cookie中，存储的数据不能超过4k

配置文件

```
CACHES = {
    'default':{
        'BACKEND':'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION':'127.0.0.1:11211'
    }
}

SESSION_ENGINE = 'django.contrib.sessions.backends.cached_db'
```

在memcached中查看

```
stats items
stats cachedump 4 0
```

session和cookie:https://www.cnblogs.com/sss4/p/7071334.html

8-memcached

1-memcached

memcached

什么是memcached

1.memcached之前是danga的一个项目，最早是为LiveJournal服务的，当初设计师为了加速LiveJournal访问速度而开发的，后来被很多大型项目采用。官网是www.danga.com或者是memcached.org。
2.Memcached是一个高性能的分布式的内存对象缓存系统，全世界有不少公司采用这个缓存项目来构建大负载的网站，来分担数据库的压力。Memcached是通过在内存里维护一个统一的巨大的hash表，memcached能存储各种各样的数据，包括图像、视频、文件、以及数据库检索的结果等。简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。
3.哪些情况下适合使用Memcached：存储验证码（图形验证码、短信验证码）、登录session等所有不是至关重要的数据。

memcache特性

1.保存内存中
2.重启服务，数据会丢失
3.LRU算法，根据最近使用的变量，将长时间没有使用的变量删除
4.memcache服务端是不安全的，
5.不适合单机使用，对内存的消耗比较大
6.格式简单，不支持list数据格式

安装和启动memcached

```
windows：
安装：memcached.exe -d install。
启动：memcached.exe -d start。

linux（ubuntu）：
安装：sudo apt install memcached
```

```
启动：
cd /usr/bin/memcached/
memcached -d start
```

可能出现的问题：

```
1.提示你没有权限：在打开cmd的时候，右键使用管理员身份运行。

2.提示缺少pthreadGC2.dll文件：将pthreadGC2.dll文件拷贝到windows/System32.

3.不要放在含有中文的路径下面
```

启动memcached：

```
-d：这个参数是让memcached在后台运行。
-m：指定占用多少内存。以M为单位，默认为64M。
-p：指定占用的端口。默认端口是11211。
-l：别的机器可以通过哪个ip地址连接到我这台服务器。如果是通过service memcached start的方式，那么只能通过本机连接。如果想要让别的机器连接，就必须设置-l 0.0.0.0
```

如果想要使用以上参数来指定一些配置信息，那么不能使用service memcached start，而应该使用/usr/bin/memcached的方式来运行。比如/usr/bin/memcached -u memcache -m 1024 -p 11222 start。

telnet操作memcached

telnet ip地址 [11211]

添加数据

```
set
  set key flas(是否压缩) timeout value_length
  value

set username 0 60 5
juran

add
  add key flas(0) timeout value_length
  value

add username 0 60 5
juran
```

set和add的区别：add是只负责添加数据，不会去修改数据。如果添加的数据的key已经存在了，则添加失败，

如果添加的key不存在，则添加成功。而set不同，如果memcached中不存在相同的key，则进行添加，如果存在，则替换。

获取数据

```
get key

get username
```

删除数据

```
delete key

delete username

flush_all：删除memcached中的所有数据。
```

自增自减

```
incr key nums
decr key nums
```

查看memcached的当前状态

```
stats
```

通过python操作memcached

安装：python-memcached：pip install python-memcached

建立连接

```
import memcache
mc = memcache.Client(['127.0.0.1:11211','192.168.174.130:11211'],debug=True)
```

设置数据

```
mc.set('username','hello world',time=60*5)
mc.set_multi({'email':'xxx@qq.com','telphone':'111111'},time=60*5)
```

获取数据

```
mc.get('telphone')
```

删除数据

```
mc.delete('email')
```

自增自减

```
mc.incr('read_count')
mc.decr('read_count')
```

分布式

```
mc = memcache.Client(["192.168.164.137:11211","192.168.164.137:11212"],debug=Tr
ue)

mc.set_multi({"pwd":'123',"age":18},time=60)
```

memcached尽管是“分布式”缓存服务器，但服务器端并没有分布式功能。各个memcached不会互相通信以共享信息,这完全取决于客户端的实现.

memcached的安全性

memcached的操作不需要任何用户名和密码，只需要知道memcached服务器的ip地址和端口号即可。因此memcached使用的时候尤其要注意他的安全性。这里提供两种安全的解决方案。

1.使用-l参数设置为只有本地可以连接：这种方式，就只能通过本机才能连接，别的机器都不能访问，可以达到最好的安全性。

2.使用防火墙，关闭11211端口，外面也不能访问。

```
ufw enable # 开启防火墙
ufw disable # 关闭防火墙
ufw default deny # 防火墙以禁止的方式打开，默认是关闭那些没有开启的端口
ufw deny 端口号 # 关闭某个端口
ufw allow 端口号 # 开启某个端口
```

在Django中使用memcached

首先需要在settings.py中配置好缓存

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
```

```
    }
}
```

如果想要使用多台机器，那么可以在LOCATION指定多个连接

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

配置好memcached的缓存后，以后在代码中就可以使用以下代码来操作memcached了：

```
from django.core.cache import cache

def index(request):
    cache.set('abc','juran',60)
    print(cache.get('abc'))
    response = HttpResponse('index')
    return response
```

需要注意的是，django在存储数据到memcached中的时候，不会将指定的key存储进去，而是会对key进行一些处理。比如会加一个前缀，会加一个版本号。如果想要自己加前缀，那么可以在settings.CACHES中添加KEY_FUNCTION参数：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'KEY_FUNCTION': lambda key,prefix_key,version:"django:%s"%key
    }
}
```

memcached 常用命令及使用说明:https://www.cnblogs.com/wayne173/p/5652034.html

9-阿里云部署

阿里云部署

阿里云部署

阿里云安全组规则

端口范围

80/80

3306/3306

6379/6379

23/23

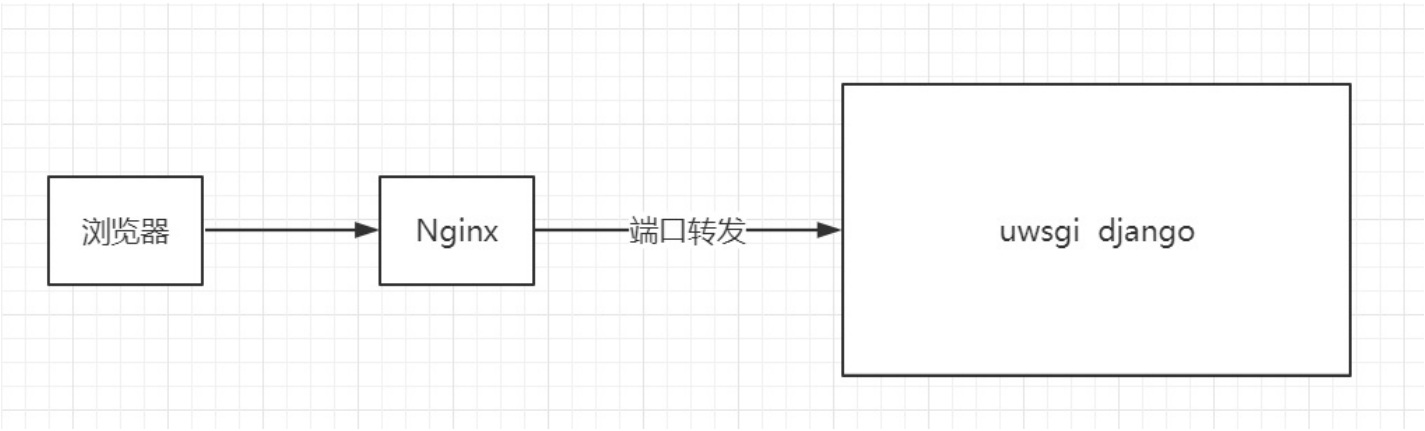
443/433

22/22

80/80

3389/3389

nginx + uwsgi



Nginx默认是80端口

1.安装Python3.7

1.安装依赖包

```
yum install opensll-devel bzip2-devel expat-devel gdbm-devel readline-devel sql
ite-devel gcc gcc-c++ opensll-devel libffi-devel python-devel mariadb-devel
```

2.下载Python源码

```
wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tgz
tar -xzvf Python-3.7.3.tgz -C /tmp
cd /tmp/Python-3.7.3
```

3.把Python3.7安装到 /usr/local目录

```
./configure --prefix=/usr/local
make
make altinstall  # 这一步比较耗时
```

4.更改/usr/bin/python链接

```
ln -s /usr/local/bin/python3.7 /usr/bin/python3
ln -s /usr/local/bin/pip3.7 /usr/bin/pip3
```

2.maridb和redis

mariadb跟MySQL是一样的,centos中已经集成了,安装非常简单

1.安装

```
sudo yum install mariadb-server
```

2.启动,重启

```
sudo systemctl start mariadb
sudo systemctl restart mariadb
```

设置安全规则,配置MySQL端口

```
访问MySQL  mysql -uroot
```

3.设置bind-ip

```
vim /etc/my.cnf
在[mysqld]: 下面加一行
```

```
bind-address = 0.0.0.0
```

4.设置外部ip可以访问

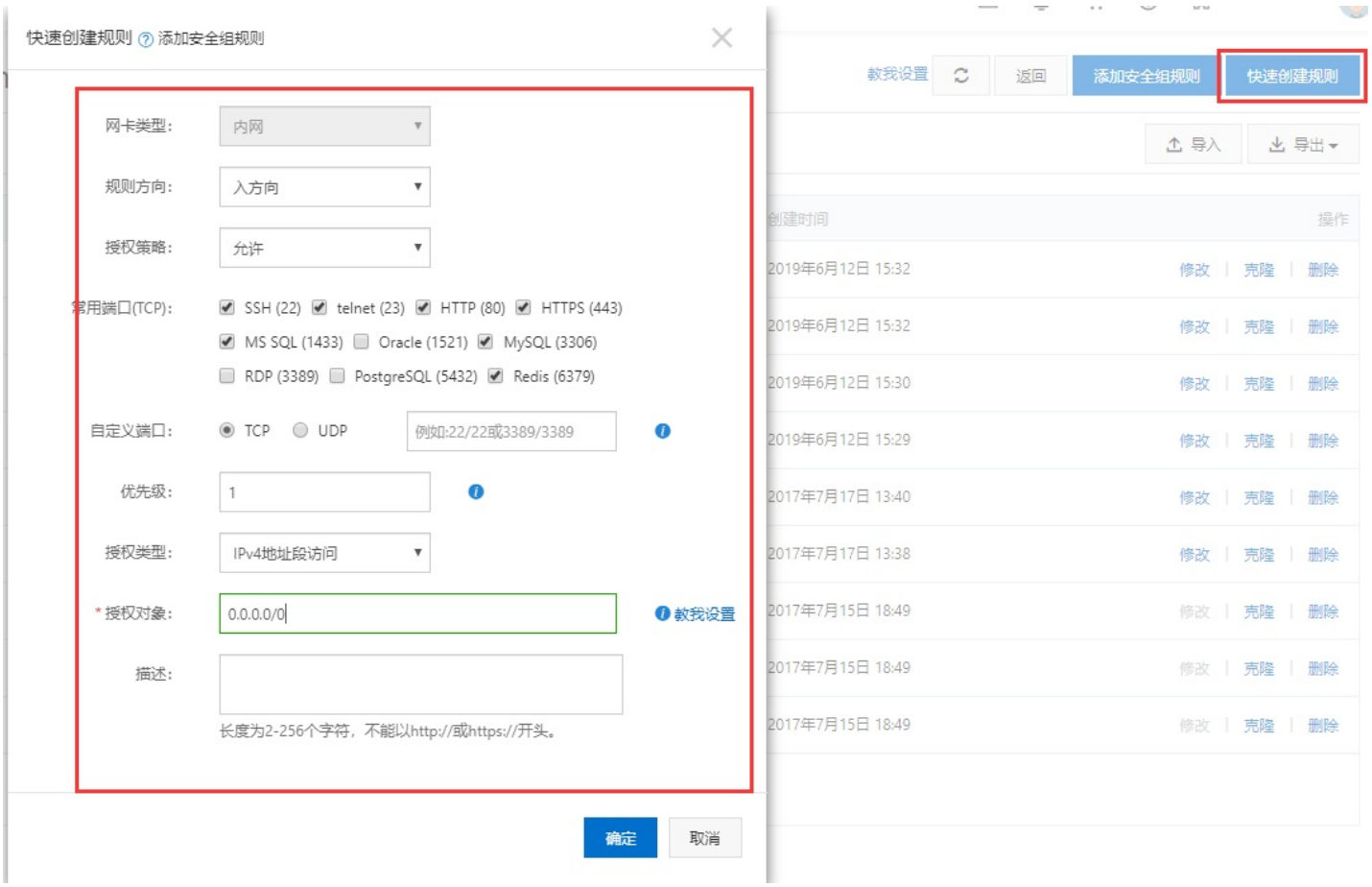
先进入MySQL才能运行下面命令

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '123456' WITH GRANT OPT
ION;

GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY '123456' WITH G
RANT OPTION;

FLUSH PRIVILEGES;
```

5.设置阿里云的对外端口



6.安装mysqlcilent出问题

```
centos 7: yum install python-devel mariadb-devel -y

pip install mysqlclient
```

将本地的数据库文件传到服务器上

6.安装redis

```
yum install redis

service redis start
```

3.安装Nginx

```
sudo yum install epel-release
sudo yum install nginx
sudo systemctl start nginx
```

4.安装virtualenvwrapper

```
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
```

```
yum install python-setuptools python-devel
pip install virtualenvwrapper
```

编辑.bashrc文件

```
vim ~/.bashrc

export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

重新加载.bashrc文件

```
source ~/.bashrc
```

如果报错,就查找一下文件,将查找到的文件路径复制到source中

```
sudo find / -name virtualenvwrapper.sh
```

新建虚拟环境

```
mkvirtualenv logic_online
mkvirtualenv -p python3 logic_online        # 用Python3新建虚拟环境
```

进入虚拟环境

```
workon logic_online
```

退出虚拟环境

```
deactivate
```

如果安装虚拟环境报错,先更新pip

```
pip3 install -i https://pypi.douban.com/simple --upgrade pip
```

安装pip包

```
将requireements.txt文件上传到服务器之后运行
pip install -r requireements.txt
安装依赖包
```

4.安装uwsgi

```
pip install uwsgi
```

6.测试uwsgi

```
uwsgi --http :8000 --module logic_online.wsgi
```

7.配置Nginx

```
新建uc_nginx.cong

# the upstream component nginx needs to connect to
upstream django {
# server unix:///path/to/your/mysite/mysite.sock; # for a file socket
server 127.0.0.1:8001; # for a web port socket (we'll use this first)
}
# configuration of the server

server {
# the port your site will be served on
listen      80;
# the domain name it will serve for
server_name 47.106.209.215;     # substitute your machine's IP address or FQDN
charset     utf-8;

# max upload size
client_max_body_size 75M;   # adjust to taste

# Django media
location /media  {
    alias /root/logic_online/media;  # 指向django的media目录
}

location /static {
    alias /root/logic_online/static; # 指向django的static目录
}

# Finally, send all non-media requests to the Django server.
location / {
    uwsgi_pass  django;
    include     uwsgi_params; # the uwsgi_params file you installed
}
}
```

8.将改配置文件加入到Nginx的启动配置文件中

```
sudo ln -s 你的目录/logic_online/conf/nginx/uc_nginx.conf  /etc/nginx/conf.d/
```

9.拉取所有需要的static file到同一目录

```
在django的setting文件中,添加下面一行内容
STATIC_ROOT = os.path.join(BASE_DIR, "static/")

运行命令
python manage.py collectstatic
```

10.运行Nginx

```
sudo /usr/sbin/nginx
```

这里注意一定是直接用Nginx命令启动,不要用systemctl启动Nginx不然会有权限问题

11.通过配置文件启动uwsgi

新建uwsgi.ini配置文件

```
# mysite_uwsgi.ini file
[uwsgi]

# Django-related settings
# the base directory (full path)
chdir           = /root/logic_online
# Django's wsgi file
module          = logic_online.wsgi
# the virtualenv (full path)

# process-related settings
# master
master          = true
# maximum number of worker processes
processes       = 10
# the socket (use the full path to be safe
socket          = 127.0.0.1:8001
# ... with appropriate permissions - may be needed
# chmod-socket    = 664
# clear environment on exit
vacuum          = true
virtualenv = /root/.virtualenvs/logic_online

#logto = /tmp/mylog.log
```

```
注：
chdir:表示需要操作的目录,也就是项目的目录
module:wsgi文件的路径
processes:进程数
virtualenv:虚拟环境的目录

workon logic_online

uwsgi -i 你的目录/logic_online/conf/uwsgi.ini &
```

12.访问

```
http://你的IP地址/
```