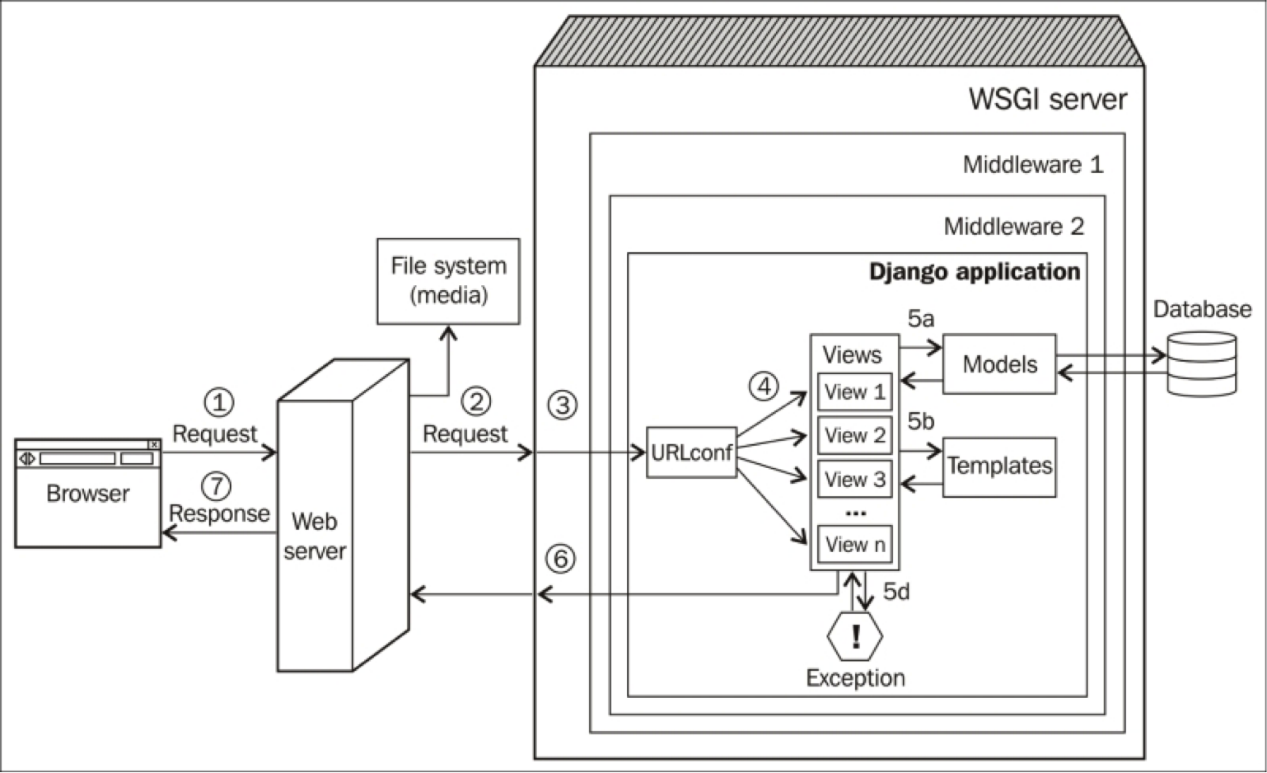
# Django基础



1. 浏览器发送请求（基本上是字节类型的字符串）到web服务器。

2. web服务器（比如，Nginx）把这个请求转交到一个WSGI（比如，uWSGI），或者直接地文件系统能够取出一个文件（比如，一个CSS文件）。

3. 不像web服务器那样，WSGI服务器可以直接运行Python应用。请求生成一个被称为environ的Ptyhon字典，而且，可以选择传递过去几个中间件的层，最终，达到Django应用。

4. URLconf中含有属于应用的urls.py选择一个视图处理基于请求的URL的那个请求，这个请求就已经变成了HttpRequest——一个Python字典对象。

5. 被选择的那个视图通常要做下面所列出的一件或者更多件事情：

* + 通过模型与数据库对话。
  + 使用模板渲染HTML或者任何格式化过的响应。
  + 返回一个纯文本响应（不被显示的）。
  + 抛出一个异常。

6. HttpResponse对象离开Django后，被渲染为一个字符串。

7. 在浏览器见到一个美化的，渲染后的web页面。

## 组成及结构

MVT

Django是一款python的web开发框架，与MVC有所不同，属于MVT框架

* m表示model，负责与数据库交互
* v表示view，是核心，负责接收请求、获取数据、返回结果
* t表示template，负责呈现内容到浏览器

组成结构

1. 管理工具（management）:内置的创建站点、迁移数据、维护静态文件的命令工具
2. 模型（model）：提供数据访问接口和模块，包括数据字段，元数据、数据关系等定义和操作
3. 视图（view）：视图层封装了HTTP request和response的一系列操作和数据流，主要功能包括URL的映射机制、绑定模板等
4. 表单（template）：通过内置的数据类型和控件生成HTML表单
5. 管理站（admin）：通过声明需要管理的model，快速生成后台数据管理网站

## 虚拟环境搭建

### virtualenv

1. 创建：virtualenv [虚拟环境名称]
2. 删除：rmvirtualenv [虚拟环境名称]
3. 进入：activate (scripts路径加入系统变量)
4. 退出：deactivate
5. x虚拟环境希望使用到y虚拟环境中所安装的所有的包

步骤：

1. 将安装在x虚拟环境中的包导入到一个文件中去,如果在home目录中执行的操作，就到home中找这个文件

pip freeze > output.txt

1. 在y虚拟环境中找到这个文件，并进行安装

pip install -r output.txt

1. 等待安装，并检查y虚拟环境中是否已成功安装工具包

pip freeze

### pipenv

1. pipenv –three
2. pipenv shell

## 安装

pip3 install django

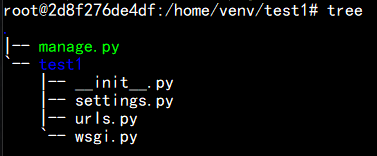
查看版本

import django

django.get\_version()

## 创建项目

django-admin startproject 项目名



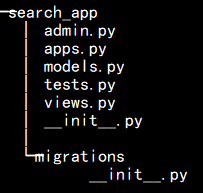
**目录说明：**

* test1: 项目的容器。
* [manage.py](https://docs.djangoproject.com/zh-hans/2.1/ref/django-admin/): 命令行工具，以各种方式与Django 项目交互。不需要编辑
* 里面一层的 mysite/ 目录包含你的项目，它是一个纯 Python 包。它的名字就是当你引用它内部任何东西时需要用到的 Python 包名。
* test1/\_\_init\_\_.py: 一个空文件，告诉 Python 该目录是一个 Python 包。
* test1/[settings.py](https://docs.djangoproject.com/zh-hans/2.1/topics/settings/): 项目的设置/配置。
* test1/[urls.py](https://docs.djangoproject.com/zh-hans/2.1/topics/http/urls/): 项目的 URL 声明; 指出了什么样的 URL 调用什么的视图
* test1/[wsgi.py](https://docs.djangoproject.com/zh-hans/2.1/howto/deployment/wsgi/): 一个 WSGI 兼容的 Web 服务器的入口，定义wsgi接口信息，与其他web服务器集成，一般无需改动。

## 建立应用

为了在项目中开发符合架构的实际应用程序，需要建立应用，每个项目可以包含多个应用。应用是一组拥有特定功能的Python包。

python manage.py startapp 应用名称



─search\_app

│ admin.py 管理站点模型的声明文件

│ apps.py 应用信息定义文件，其中生成了类AppConfig，用于定义应用名等Meta数据

│ models.py 添加模型层数据类的文件

│ tests.py 测试代码文件

│ views.py 定义URL响应函数

│ \_\_init\_\_.py 一个空文件，告诉 Python 该目录是一个 Python 包。

│

└─migrations 用于定义引用迁移功能

\_\_init\_\_.py

项目 VS 应用

应用是一个专门做某件事的网络应用程序——比如博客系统，或者公共记录的数据库，或者简单的投票程序。项目则是一个网站使用的配置和应用的集合。项目可以包含很多个应用。应用可以被很多个项目使用。

## 应用配置

1. 修改工程配置文件 Search\_Engine/setting.py
2. INSTALLED\_APPS 添加应用名称

[INSTALLED\_APPS](https://docs.djangoproject.com/zh-hans/2.1/ref/settings/#std:setting-INSTALLED_APPS) 默认包括了以下 Django 的自带应用：

* [django.contrib.admin](https://docs.djangoproject.com/zh-hans/2.1/ref/contrib/admin/#module-django.contrib.admin) -- 管理员站点。
* [django.contrib.auth](https://docs.djangoproject.com/zh-hans/2.1/topics/auth/#module-django.contrib.auth) -- 认证授权系统。
* [django.contrib.contenttypes](https://docs.djangoproject.com/zh-hans/2.1/ref/contrib/contenttypes/#module-django.contrib.contenttypes) -- 内容类型框架。
* [django.contrib.sessions](https://docs.djangoproject.com/zh-hans/2.1/topics/http/sessions/#module-django.contrib.sessions) -- 会话框架。
* [django.contrib.messages](https://docs.djangoproject.com/zh-hans/2.1/ref/contrib/messages/#module-django.contrib.messages) -- 消息框架。
* [django.contrib.staticfiles](https://docs.djangoproject.com/zh-hans/2.1/ref/contrib/staticfiles/#module-django.contrib.staticfiles) -- 管理静态文件的框架。

INSTALLED\_APPS = [  
 'search\_app'  
]

1. MIDDLEWARE 注释 csrf 校验

MIDDLEWARE = [  
 # 'django.middleware.csrf.CsrfViewMiddleware',]

1. DATABASES  修改连库配置默认 sqlite3 改mysql

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.mysql',  
 'NAME': 'python',  
 'USER': 'root',  
 'PASSWORD': 'zkyr1006',  
 'HOST': '127.0.0.1',  
 'PORT': '3306',  
 }  
}

1. 修改时间和语言

LANGUAGE\_CODE = 'zh-Hans'  
TIME\_ZONE = 'Asia/Shanghai'

1. 允许所有IP访问

ALLOWED\_HOSTS = ['\*']

1. 定义用户信息的数据模型 search\_app/ models.py

定义字段名和数据类型

1. 初始化模型数据库并生成数据库文件

python manage.py migrate

migrate检查 INSTALLED\_APPS 设置，为其中的每个应用创建需要的数据表

1. 显示注册信息修改默认标题 search\_app/admin.py

*from* django.contrib *import* admin  
*from* search\_app.models *import* User  
  
# Register your models here.  
*class* HostAdmin(admin.ModelAdmin):  
 list\_display = [  
 'name',  
 'age',  
 'birthday',  
 'gender',  
 'account',  
 ]  
 search\_fields = ('name',)  
  
admin.site.register(User,HostAdmin)  
admin.AdminSite.site\_header='运维系统管理后台'  
admin.AdminSite.site\_title='运维系统'

1. 添加应用的url访问 Search\_Engine/urls.py

*from* django.contrib *import* admin  
*from* django.urls *import* path  
  
urlpatterns = [  
 path('admin/',admin.site.urls)  
]

1. mysql创建表

## 创建、激活模型

### 创建模型

编辑app中的model.py文件

*class* Question(models.Model):  
 question\_text = models.CharField(max\_length=200)  
 pub\_date = models.DateTimeField('date published')  
*class* Choice(models.Model):  
 question = models.ForeignKey(Question, on\_delete=models.CASCADE)  
 choice\_text = models.CharField(max\_length=200)  
 votes = models.IntegerField(default=0)

每个模型被表示为 [django.db.models.Model](https://docs.djangoproject.com/zh-hans/2.1/ref/models/instances/#django.db.models.Model) 类的子类。每个模型有一些类变量，它们都表示模型里的一个数据库字段。

每个字段都是 Field 类的实例 - 比如，字符字段被表示为 CharField ，日期时间字段被表示为 DateTimeField。这将告诉 Django 每个字段要处理的数据类型。

每个 Field 类实例变量的名字（例如 question\_text 或 pub\_date ）也是字段名，数据库会将它们作为列名。

使用 ForeignKey 定义了一个关系。这将告诉 Django，每个 Choice 对象都关联到一个 Question 对象。Django 支持所有常用的数据库关系：多对一、多对多和一对一。

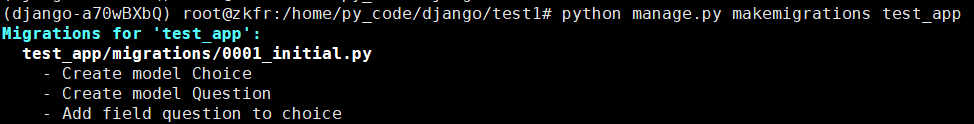
### 激活模型

1. 在项目test1下修改settings.py，NSTALLED\_APPS 子项添加点式路径app的Config

INSTALLED\_APPS = [  
 'test\_app.apps.TestAppConfig'  
]

1. 为模型的改变生成迁移文件

**python manage.py makemigrations test\_app**



通过运行 makemigrations 命令，Django 会检测你对模型文件的修改（在这种情况下，你已经取得了新的），并且把修改的部分储存为一次 *迁移*。

迁移是 Django 对于模型定义（也就是你的数据库结构）的变化的储存形式，其实只是一些磁盘上的文件。被储存在 test\_app/migrations/0001\_initial.py 里

## 应用数据库生成/迁移

**python manage.py migrate**

* migrate 命令可以在SQL等数据库中创建与models.py代码对应的表
* 选中所有还没有执行过的迁移（Django 通过在数据库中创建一个特殊的表 django\_migrations 来跟踪执行过哪些迁移）并应用在数据库上 - 也就是将你对模型的更改同步到数据库结构上。

迁移是非常强大的功能，它能让你在开发过程中持续的改变数据库结构而不需要重新删除和创建表 - 它专注于使数据库平滑升级而不会丢失数据。

tip：

**python manage.py check**

这个命令帮助你检查项目中的问题，并且在检查过程中不会对数据库进行任何操作。

清空数据库

**python manage.py flush**

## 运行简易开发服务器

**python manage.py runserver** #默认端口8000

**python manage.py runserver 8080**  #指定端口8080

**python manage.py runserver 0:8000** # 0.0.0.0:8000

django开发服务是可用在开发期间的，一个内建的，轻量的web服务。在准备发布产品之前，无需进行产品级 Web 服务器（比如 Apache）的配置工作。

不要在正式的应用布署环境中使用它。 在同一时间，该服务器只能可靠地处理一次单个请求，并且没有进行任何类型的安全审计。

runserver在需要的情况下会对每一次的访问请求重新载入一遍 Python 代码。所以你不需要为了让修改的代码生效而频繁的重新启动服务器。然而，一些动作，比如添加新文件，将不会触发自动重新加载，这时你得自己手动重启服务器。

## Django命令行

**python manage.py shell**

manage.py 会设置 DJANGO\_SETTINGS\_MODULE 环境变量，这个变量会让 Django 根据 test1/settings.py 文件来设置 Python 包的导入路径。

终端上输入 **python manage.py** 可以看到详细的列表，在忘记子名称的时候特别有用。

## 管理页面

1. 创建一个管理员账号

**python manage.py createsuperuser**

邮箱可以为空

1. 修改用户密码

**python manage.py changepassword username**

1. 向管理页面加入应用

修改app的admin.py文件

*from* django.contrib *import* admin  
*from* .models *import* Question  
admin.site.register(Question)

* 应用的表单是在model文件中自动生成的
* 不同的字段类型（日期时间字段 DateTimeField 、字符字段 CharField）会生成对应的 HTML 输入控件。每个类型的字段都知道它们该如何在管理页面里显示自己。

## 复用、打包

APP目录可被拷贝到一个新的Django工程复用，也可打包应用方便其他人安装使用

### 打包步骤

1. 在Django项目目录外创建一个文件夹用于放置app，不要和已存在的包名冲突
2. 将app目录移动到创建的文件夹
3. 创建一个README.rst文件，内容如下，根据实际情况修改

=====

Test\_app

=====

Test\_app is a simple Django app to conduct Web-based test\_app. For each

question, visitors can choose between a fixed number of answers.

Detailed documentation is in the "docs" directory.

Quick start

-----------

1. Add "test\_app" to your INSTALLED\_APPS setting like this::

INSTALLED\_APPS = [

...

'test\_app',

]

2. Include the test\_app URLconf in your project urls.py like this::

path('test\_app/', include('test\_app.urls')),

3. Run `python manage.py migrate` to create the test\_app models.

4. Start the development server and visit http://127.0.0.1:8000/admin/

to create a poll (you'll need the Admin app enabled).

1. Visit http://127.0.0.1:8000/test\_app/ to participate in the poll.
2. 创建LICENSE文件，选择一个非本教程使用的授权协议，可以自己选择一个授权协议。只要确定你选择的协议能够限制未来会使用你的代码的人。
3. 创建 setup.py 用于说明如何构建和安装应用的细节。内容如下，根据实际情况修改

import os

from setuptools import find\_packages, setup

with open(os.path.join(os.path.dirname(\_\_file\_\_), 'README.rst')) as readme:

README = readme.read()

# allow setup.py to be run from any path

os.chdir(os.path.normpath(os.path.join(os.path.abspath(\_\_file\_\_), os.pardir)))

setup(

name='django-test\_app',

version='0.1',

packages=find\_packages(),

include\_package\_data=True,

license='BSD License', # example license

description='A simple Django app to conduct Web-based test\_app.',

long\_description=README,

url='https://www.example.com/',

author='Your Name',

author\_email='yourname@example.com',

classifiers=[

'Environment :: Web Environment',

'Framework :: Django',

'Framework :: Django :: X.Y', # replace "X.Y" as appropriate

'Intended Audience :: Developers',

'License :: OSI Approved :: BSD License', # example license

'Operating System :: OS Independent',

'Programming Language :: Python',

'Programming Language :: Python :: 3.5',

'Programming Language :: Python :: 3.6',

'Topic :: Internet :: WWW/HTTP',

'Topic :: Internet :: WWW/HTTP :: Dynamic Content',

],

)

1. 默认包中只包含 Python 模块和包。为了包含额外文件,创建MANIFEST.in 的文件。

内容如下，根据实际情况修改

include LICENSE

include README.rst

recursive-include test\_app/static \*

recursive-include test\_app/templates \*

1. 使用**python setup.py sdist**创建一个名为 ` dist `的目录并构建你自己的应用包django-test\_app-0.1.tar.gz。

### 安装包

进入虚拟环境，使用pip安装

**pip install django-test\_app/dist/django-test\_app-0.1.tar.gz**

## 开发流程

**step1:创建虚拟环境**

**step2：安装django包**

**step3：创建项目**

**step4：创建应用 需要将应用加入到 INSTALLED\_APPS**

**step5:根据表结构定义模型类 models.py 定义类属性 、（定义admin）**

**step6：定义视图 视图需要接收request参数（必须）,必须返回结果**

**step7：配置url**

**step8：创建模板，模板一般放到项目路径下，建立template/projectname**

# 环境搭建

## Vue环境搭建

1. 安装nodejs

<https://nodejs.org/en/>

1. cnpm

npm install -g cnpm --registry=https://registry.npm.taobao.org

1. 安装依赖

cnpm install

1. 运行

cnpm run dev

## 安装库

1. 依赖库

pip install -i https://pypi.douban.com/simple django==2.0.2

pip3 install markdown

pip3 install django-filter

pip3 install pillow

pip3 install pymysql

1. djangorestframework

pip3 install djangorestframework

1. 安装Mysqlclient

https://www.lfd.uci.edu/~gohlke/pythonlibs/

pip3 install mysqlclient-1.3.12-cp36-cp36m-win\_amd64

\_\_init\_\_.py添加代码：

import pymysql

pymysql.install\_as\_MySQLdb()

## xadmin

1. <https://github.com/sshwsfc/xadmin/tree/django2>

把zip文件放到pip3目录下，运行下面命令安装：

pip3 install xadmin-django2

pip3 install django-import-export

1. 源码xadmin文件夹放到mydjango/extra\_apps文件夹下面
2. 把extra\_apps右键mark为Source Root并在settings中加入

sys.path.insert(0,os.path.join(BASE\_DIR, 'extra\_apps'))

1. pip3 uninstall xadmin

## 富文本编辑器

1. https://github.com/twz915/DjangoUeditor3/

解压后，把DjangoUeditor文件夹拷贝到项目目录下面

1. settings中添加app

INSTALLED\_APPS = [

'DjangoUeditor',

]

1. mydjango/urls.py加入

# 富文本编辑器url

path('ueditor/',include('DjangoUeditor.urls' )),

1. drf

pip install coreapi drf的文档支持

pip install django-guardian drf对象级别的权限支持

pip install django-cors-headers drf跨域问题

1. json web token

pip install djangorestframework-jwt

# 模型

模型是数据唯一而且准确的信息来源。它包含正在储存的数据的重要字段和行为。一般来说，每一个模型都映射一个数据库表。

* 每个模型都是一个 Python 的类，这些类继承 django.db.models.Model
* 模型类的每个属性都相当于一个数据库的字段。

Django 给你一个自动生成访问数据库的 API

## ORM (Object Relational Mapping)

对象关系映射就是把我们定义的对象(类)映射到对应的数据库的表上。 所以ORM就是代码(软件)层面对于数据库表和关系的一种抽象。

Django的 Model就是orm的一个具体实现。

*from* django.db *import* models  
*class* Question(models.Model):  
 question\_text = models.CharField(max\_length=200)  
 pub\_date = models.DateTimeField('date published')

模型定义了一个Question, 定义两个字段：question\_text和pub\_date，每个字段都被指定为一个类属性，并且每个属性映射为一个数据库列。

## 字段类型

模型中每一个字段都应该是相应类的实例，字段类型用以：

* 指定数据库数据类型（如：INTEGER, VARCHAR, TEXT）
* 默认的HTML表单输入框</ref/forms/widgets>（如：<input type="text"><select>）
* Django admin和自动生成表单的基本验证。

Django内置了多种字段类型；可以在<model-field-types>中看到完整列表。也可以[编写自定义的字段类型](https://docs.djangoproject.com/zh-hans/2.1/howto/custom-model-fields/)

### 数值型

1. AutoField int(ll)。 自增主键， Django Model 默认提供 ， 可以被重写。 它的完整定义是 id = models.AutoField(primary\_key=True)。
2. BooleanField tinyint(l)。 布尔类型字段，一般用于记录状态标记。
3. DecimalField decimal。 开发对数据精度要求较高的业务时考虑使用 ，比如做支付相关、金融相关。 定义时，需要指定精确到多少位，比如 cash =models.DecimalField(max\_digits=8,decimal\_places=2,default=0, verbose\_name="消费金额”) 就是定义总长度为 8 位、精度为 2 位的数字
4. IntegerField int(ll)。 它同 AutoField 一样，唯一的差别就是不自增
5. PositiveintegerField。 同IntergerField，只包含正整数。
6. SmallintegerField smallint。 小整数时一般会用到。

### 字符型

1. CharField varchar。 基础的 varchar 类型。
2. URLField。 继承自CharField，但是实现了对URL的特殊处理。
3. UUIDField char(32)。 除了在 PostgreSQL中使用的是 uuid类型外，在其他数据库中均是固定长度char(32) ，用来存放生成的唯一id。
4. EmailField。 同URLField继承自 CharField， 多了对E-mail 的特殊处理。
5. FileField。继承自 CharField，多了对文件的特殊处理。 当你定义一个字段为 FileField 时，admin部分展示时会自动生成一个可上传文件的按钮。
6. TextField longtext。 一般用来存放大量文本内容，比如新闻正文 、博客正文
7. ImageField。 继承自FileField，用来处理图片相关的数据，在展示上会有不同。

### 日期类型

1. DateField
2. DateTimeField
3. TimeField

### 关系类型

1. ForeignKey
2. OneToOneField
3. ManyToManyField

其中外键和一对一其实是一种，只是一对一在外键的字段上加了unique。 而多对多会创建 一个中间表，来进行多对多的关联

Django 提供了定义三种最常见的数据库关联关系的方法：多对一，多对多，一对一。

#### 多对一

使用 **django.db.models.ForeignKey** 类，在模型中添加一个值为该类的属性。

*class* Manufacturer(models.Model):  
 *pass  
class* Car(models.Model):  
 manufacturer = models.ForeignKey(Manufacturer, on\_delete=models.CASCADE)

多个car对应一个Manufacturer

#### 多对多

使用 **django.db.models.ManyToManyField** 类。在模型中添加一个值为该类的属性。

*class* Topping(models.Model):  
 *pass  
class* Pizza(models.Model):  
 toppings = models.ManyToManyField(Topping)

一个pizza含有多种topping，一个topping存在于多种pizza中

对于多对多关联关系的两个模型，可以在任何一个模型中添加 ManyToManyField 字段，但不能同时在两模型中添加该字段

额外字段

当牵扯到数据与两个模型之间的关系相关联的时候，将使用到额外字段

可以在中间模型上添加额外的字段。中间模型与ManyToManyField使用**through**参数指向将充当中介的模型相关联

中间模型有一些限制：

* 中间模型必须只有一个源模型的外键，或者显式指定Django应该用于关系的外键ManyToManyField.through\_fields。如果有多个外键through\_fields但未指定，则会引发验证错误。类似的限制适用于目标模型的外键
* 对于通过中间模型与自身具有多对多关系的模型，允许同一模型的两个外键，但它们将被视为多对多关系的两个（不同）侧。如果有更多的外键，还必须指定through\_fields
* 在使用中间模型定义从模型到自身的多对多关系时，必须使用 symmetrical=False

*class* Person(models.Model):  
 name = models.CharField(max\_length=128)  
 *def \_\_str\_\_*(self):  
 *return* self.name  
*class* Group(models.Model):  
 name = models.CharField(max\_length=128)  
 members = models.ManyToManyField(Person, through='Membership')  
 *def \_\_str\_\_*(self):  
 *return* self.name  
*class* Membership(models.Model):  
 person = models.ForeignKey(Person, on\_delete=models.CASCADE)  
 group = models.ForeignKey(Group, on\_delete=models.CASCADE)  
 date\_joined = models.DateField()  
 invite\_reason = models.CharField(max\_length=64)

#### 一对一

使用 OneToOneField，在模型中添加一个值为该类的属性，

#### 跨文件

将模型与另一个应用程序中的模型相关联，在定义模型的文件顶部导入相关模型，然后引入其他模型

## 字段选项

### 字段参数

每一种字段都需要指定一些特定的参数，一些可选的参数是通用的，可以用于任何字段类型。

1. null。 可以同 blank对比考虑，其中 null 用于设定在数据库层面是否允许为空。
2. blank。 针对业务层面，该值是否允许为空。
3. choices。配置字段的 choices 后，admin页面上就可以看到对应的可选项展示
4. db\_column。 默认情况下，定义的 Field就是对应数据库中的字段名称 ，通过这个参数可以指定 Model中的某个字段对应数据库中的哪个字段
5. db\_index。索引配置。对于业务上需要经常作为查询条件的字段，应该配置此项。
6. default。 默认值配置。
7. editable。 是否可编辑，默认是 True。 如果不想将这个字段展示到页面上，可以配置为 False。
8. error\_messages。用来自定义字段值校验失败时的异常提示，字典格式。key可选项为null、blank、invalid、invalid\_choice、unique、unique\_for\_date。
9. help\_text。 字段提示语，配置这一项后，在页面对应字段的下方会展示此配置
10. primary\_key。 主键，一个 Model 只允许设置一个字段为 primary\_key。在一个模型中，如果没有对任何一个字段设置primary\_key=True选项。 Django 会自动添加一个IntegerField字段设置为主键，需要重写 Django 默认的主键设置，将该字段设置为 True，设置为该模型的主键。
11. unique。 唯一约束，当需要配置唯一值时，设置 unique=True，设置此项后，不需要设置 db\_index。
12. unique for date。针对 date(日期)的联合约束，并不是数据库层面的约束。
13. unique\_for\_month。 针对月份的联合约束。
14. unique\_for\_year。 针对年份的联合约束。
15. verbose\_name。 字段对应的展示文案。除了 ForeignKey`ManyToManyField 和 OneToOneField ，任何字段类型都接收一个可选的参数 verbose\_name ，如果未指定该参数值， Django 会自动使用该字段的属性名作为该参数值，并且把下划线转换为空格。ForeignKey, ManyToManyField and OneToOneField 接收的第一个参数为模型的类名，后面可以添加一个 verbose\_name 参数，一般情况下不需要将 verbose\_name 值首字母大写，必要时 Djanog 会自动把首字母转换为大写。
16. validators。 自定义校验逻辑,同 form 类似

## 字段名称限制

* 字段名称不能是Python保留字，会导致Python语法错误
* 字段名称不能在一行中包含多个下划线，例如foo\_\_bar=models.IntegerField()

## QuerySet

在 Model层中， Django通过给 Model增加一个 objects 属性来提供数据操作的接口。 比如，想要查询所有文章的数据，可以这么写 : Post.objects.all()，这样就能拿到 QuerySet 对象。 这个对象中包含了我们需要的数据， 当我们用到它时，它会去DB中获取数据。

Django 中的 QuerySet 本质上是一个懒加载的对象，上面的两行代码执行后，都不 会产生数据库查询操作，只是会返回一个 QuerySet 对象，等你真正用它时才会执行查询。

posts= Post.objects.all() #返回 一个 QuerySet 对象并赋位给 posts  
available\_posts = posts.filter(status=1) #继续返回 一个 QuerySet对象并赋位给available\_posts  
print(available\_posts) # 此时会根据上面的两个 条件执行数据查询操作，对应的 SQL 语句为 SELECT \* FROM blog\_post where status =1;

链式调用就是，执行一个对象中的方法之后得到的结果还是这个对象，这样可以接着执行对象上的 其他方法

posts = Post.objects.filter(status=1).filter(category\_id=2).filter(title\_icontains="the5fire")

在每个函数(或者方法)的执行结果上可以继续调用同样的方法，因为每个函数的返回值都 是它自己， 也就是 QuerySet

### 常用接口

#### 支持链式调用

支持链式调用的接口即返回 QuerySet 的接口

1. all。 相当于 SELECT \* FROM table name语句，用于查询所有数据。
2. filter。根据条件过滤数据，常用的条件基本上是字段等于、不等于、大于、小于。还有其他的，比如能改成产生LIKE查询的:Model.objects.filter (content\_contains=“条件”)。
3. exclude。 同 filter，只是相反的逻辑。
4. reverse 接口。 把 QuerySet 中的结果倒序排列。
5. distinct接口。 用来进行去重查询 ， 产生 SELECT DISTINCT 这样的SQL查询。
6. none接口。 返回空的QuerySet。

#### 不支持链式调用的接口

不支持链式调用的接口即返回值不是 QuerySet 的接口。

1. get。 比如Post.objects.get(id=l)用于查询id为l的文章: 如果存在， 则直 接返回对应的Post实例;如果不存在，则抛出 DoesNotExist异常:

*try*:   
 post=Post.objects.get(id=l)   
*except* Post.DoesNotExist:

*xxxx*

1. create。 用来直接创建一个 Model 对象，比如:post=Post.objects.create(title=”test”)。
2. update\_or\_create 接口。 同 get\_or\_create，只是用来做更新操作。
3. count。 返回QuerySe记录数量， 相当于SELECT COUNT(\*) FROM table\_1
4. latest。用于返回最新的一条记录，但是需要在Model的Meta中定义:get\_latest\_by = <用来排序的字段〉。
5. earliest。 同上，返回最早的一条记录。
6. first。 从当前QuerySet记录中获取第一条。
7. last接口。 同上，获取最后一条。
8. exists接口。 返回True或者False，在数据库层面执行SELECT (1) AS ”a” FROM  table\_name LIMIT l 的查询，如果只是需要判断 QuerySet 是否有数据，用这个接口 是最合适的方式。 不要用 count 或者 len(queryset)这样的操作来判断是否存在。 相反， 如果可以预期接下来会用到 QuerySet 中的数据 ，可以考虑使用 len(queryset) 的方式来做判断，这样可以减少一次 DB查询请求。
9. bulk\_create 接口。 同 create，用来批量创建记录。
10. in\_bulk。 批量查询， 接收两个参数 id\_list 和 filed\_name。 可以通过 Post.ob]ects.in\_bulk([1, 2, 3])查询出 id为 1、 2、 3的数据，返回结果是字典类型，字典类型的 key 为查询条件。 返回结果示例: {l:a, 2:b>, 3:c}
11. update。 用来根据条件批量更新记录 ，比如 :Post.objects.filter(owner\_name=‘a’).update(title=’b’)。
12. delete。 同 update，这个接口是用来根据条件批量删除记录。 需要注意的是 ， update 和 delete都会触发 Django的 signal ( signal的用法见附录 E)。
13. values 。 当我们明确知道只需要返回某个字段的值 ，不需要 Model实例时，可以使 用它，用法如下: title\_list= Post.objects.filter(category\_id=l).values(‘title’) #返回的结果包含dict的QuerySet，类似这样: <QuerySet [{'title’:xxx},]>
14. values\_list。 同 values，但是直接返回的是包含 tuple 的 QuerySet : titles\_list=Post.objects.filter(category=l).values\_list('title') #返回结果类似:<QuerySet[(‘标题’,)]> 如果只是一个字段 的话，可以通过增加 flat=True 参数， 便于我们后续处理 :

title\_list =Post.objects.filter(category=1).values\_list('title', flat=*True*)  
*for* title *in* title\_list :  
 print(title)

#### 进阶接口

1. defer接口。 把不需要展示的字段做延迟加载。 比如说， 需要获取到文章中除正文外的其他字段，就可以通过 posts = Post.objects.all().defer('content’)，这样拿 到的记录中就不会包含 content 部分。 但是当我们需要用到这个字段时，在使用时会去加载。
2. only接口。 同defer接口刚好相反，如果只想获取到所有的title记录，就可以使用only，只获取 title 的内容， 其他值在获取时会产生额外的查询。
3. 解决一对多的关联关系: 一般情况下由外键查询产生的N+1问题比较多 ，即一条查询请求返回N条数据,当我们操作数据时,又会产生额外的请求,所有的 ORM 框架都存在这样的问题

posts =Post.objects .all()  
*for* post *in* posts : #产生数据库查询  
 print(post.owner) #产 生额外的数据库查询

select\_related 接口用来解决外键产生的 N+1问题的方案 。

posts=Post.objects.all().select\_related('category')  
*for* post *in* posts: #产生数据库查询， category 数据也会一次性查询出未  
 print(post.category)

1. prefetch related 接口 。针对多对多关系的数据，可以通过这个接口来避免 N+l 查询 。 比如， post和tag的关系可以通过这种方式来避免:

posts = Post.objects.all().prefetch\_related('tag')  
*for* post *in* posts: #产生两条查询语句， 分别查询post和tag  
 print(post.tag.all())

### 常用字段查询

1. contains :包含， 用来进行相似查询 。
2. icontains:同 contains，只是忽略大小写。
3. exact: 精确匹配。
4. iexact:同 exact，忽略大小写。
5. in:指定某个集合 ，比如 Post.objects.filter(id\_\_in=[1,2,3])相当于 SELECT  \* FROM blog\_post WHERE IN (1, 2, 3 ); 。
6. gt:大于某个值。
7. gte: 大于等于某个值。
8. lt:小于某个值。
9. lte:小于等于某个值。
10. startswith:以某个字符串开头，与 contains 类似，会产生 LIKE ‘<关键词>%’  这样的 SQL。
11. istartswith:同 startswith，忽略大小写。
12. endswith:以某个字符串结尾。
13. iendswith:同 endswith，忽略大小写。
14. range:范围查询，多用于时间范围，如Post.objects.filter(created\_time range= (‘2018-05-01’,’2018-06-01’))会产生这样的查询: SELECT ... WHERE created\_time BETWEEN '2018-05-01’ AND ’2018-06-01’; 。

### 进阶字段查询

1. F． F 表达式常用来执行数据库层面的计算，从而避免出现竞争状态 。
2. Q． Q 表达式用来解决OR 查询，或者进行 AND (&)查询

*from* django .db .models *import* Q  
Post.objects.filter(Q(id=1)| Q(id=2))

1. count 。 用来做聚合查询
2. Sum。 同 Count 类似 ，只是它是用来做合计的, 还有Avg、 Min和Max等表达式， 均用来满足我们对SQL查询的 需求。

## meta选项

使用内部提供的class Meta元数据模型

*class* Ox(models.Model):  
 horn\_length = models.IntegerField()  
  
 *class* Meta:  
 ordering = ["horn\_length"]  
 verbose\_name\_plural = "oxen"

模型元数据是“任何不是字段的东西”， 例如排序选项（ordering），数据库表名（db\_table）或人类可读的单数和复数名称（verbose\_name和 verbose\_name\_plural）

## 模型属性

objects

模型最重要的属性是 Manager。为Django模型提供数据库查询操作的接口，用于从数据库中检索实例。如果Manager未定义自定义，则默认名称为 objects。管理员只能通过模型​​类访问，而不能通过模型​​实例访问。

## 模型方法

**\_\_str\_\_()**

返回任何对象的字符串表示形式。这是Python和Django在模型实例需要被强制并显示为纯字符串时将使用的内容。

**get\_absolute\_url()**

这告诉Django如何计算对象的URL。Django在其管理界面中使用它，并且在需要的时候计算出对象的URL。

对于对象具有特殊URL定义方法的情况，需要定义这个方法

## 继承模型

* 抽象基类：使用父类来保存，不希望为每个子模型键入的信息。这个类不会被孤立使用
* 多表继承：是现有模型的子类（可能是完全来自另一个应用程序的东西），并希望每个模型都有自己的数据库表
* 代理模型：修改模型的Python级行为，而不以任何方式更改模型字段

## 使用

修改settings.py文件

# 管理后台

# 模板

模板是一个文本，用于分离文档的表现形式和内容

app文件夹下建立templates目录并建立HTML文件

## 模板语言

1. 变量 {{ variable }}

1.字典查询，例如：foo["bar"]

2.属性或方法查询，例如：foo.bar

3.数字索引查询，例如：foo[bar]）

在模板中调用方法时不能传递参数

1. 标签 { % 代码块 % }

作用：在输出中创建文本；控制循环或逻辑；加载外部信息到模板中供以后的变量使用

{% for hero in list %}

<li>{{ hero.hname }}</li>

{% endfor %}

1. 过滤器

语法：{ { 变量|过滤器 }}，例如{ { name|lower }}，表示将变量name的值变为小写输出

使用管道符号 (|)来应用过滤器

通过使用过滤器来改变变量的计算结果

可以在if标签中使用过滤器结合运算符

if list1|length > 1

过滤器能够被“串联”，构成过滤器链

name|lower|upper

过滤器可以传递参数，参数使用引号包起来 {% if forloop.counter|divisibleby:"2" %}

list|join:", "

default：如果一个变量没有被提供，或者值为false或空，则使用默认值，否则使用变量的值

value|default:"什么也没有"

date：根据给定格式对一个date变量格式化

value|date:'Y-m-d'

escape：详见“HTML转义”

1. 注释{# 代码或html #}

单行注释

{#...#}

注释可以包含任何模版代码，有效的或者无效的都可以

{# { % if foo % }bar{ % else % } #}

使用comment标签注释模版中的多行内容

1. for标签

{ %for ... in ...%} 循环逻辑 {{forloop.counter}}表示当前是第几次循环 { %empty%} 给出的列表为或列表不存在时，执行此处 { %endfor%}

1. if标签

{ %if ...%} 逻辑1 { %elif ...%} 逻辑2 { %else%} 逻辑3 { %endif%}

1. comment标签

{ % comment % } 多行注释 { % endcomment % }

1. include：加载模板并以标签内的参数渲染

{ %include "foo/bar.html" % }

1. url：反向解析

{ % url 'name' p1 p2 %}

1. csrf\_token：这个标签用于跨站请求伪造保护

{ % csrf\_token %}

1. 布尔标签：and、or，and比or的优先级高

block、extends：详见“模板继承”

autoescape：详见“HTML转义”

1. html转义：

{{ t1|safe }} {# 过滤器关闭转义 #}

<hr>

{% autoescape off %} {# 标签关闭转义 #}

{{ t1 }}

{% endautoescape %}

# 表单

## get

get用于接收用户的请求

# 表单  
*def* search\_form(request):  
 *return* render\_to\_response('search\_form.html')  
# 接收请求数据  
*def* search(*request*):   
 *request*.encoding='utf-8'  
 *if* 'q' *in request*.GET:  
 message = '你搜索的内容为: ' + *request*.GET['q'].encode('utf-8')  
 *else*:  
 message = '你提交了空表单'  
 *return* HttpResponse(message)

## post

提交数据时更常用POST方法。

post的HTML文档有一个{% csrf\_token %}的标签。csrf全称是Cross Site Request Forgery。这是Django提供的防止伪装提交请求的功能。POST方法提交的表格，必须有此标签。

# 视图

## [URL配置](https://blog.csdn.net/qq_40272386/article/details/78800507)

**path('booktest/**<*int***:id**>**',booktest\_views.show,name = 'booktest'),**

基本规则：

* 使用尖括号(<>)从url中捕获值。
* 捕获值中可以包含一个转化器类型（converter type），比如使用 <int:name> 捕获一个整数变量。若果没有转化器，将匹配任何字符串，当然也包括了 / 字符。
* 无需添加前导斜杠。
* Django默认支持以下5个转化器：
  + str,匹配除了路径分隔符（/）之外的非空字符串，这是默认的形式
  + int,匹配正整数，包含0。 自动转换为整型而不是字符串。
  + slug,匹配字母、数字以及横杠、下划线组成的字符串。
  + uuid,匹配格式化的uuid，如 075194d3-6885-417e-a8a8-6c931e272f00。
  + path,匹配任何非空字符串，包含了路径分隔符

**urlpatterns = [   
path('articles/2003/', views.special\_case\_2003),   
path('articles/**<*int***:year**>**/', views.year\_archive),   
path('articles/**<*int***:year**>**/**<*int***:month**>**/', views.month\_archive),   
path('articles/**<*int***:year**>**/**<*int***:month**>**/**<**slug**>**/', views.article\_detail),   
]**

* 使用正则表达式

如果上述的paths和converters还是无法满足需求，也可以使用正则表达式，这时应当使用 django.urls.re\_path函数。

在Python正则表达式中，命名式分组语法为 (?P<name>pattern) ，其中name为名称， pattern为待匹配的模式。