**1 - 搭建开发环境**

1 月，2 周前 字数 5685 阅读 1400 评论 17

**本教程使用的开发环境**

本教程写作时开发环境的系统平台为 Windows 10 （64 位），Python 版本为 3.5.2 （64 位），Django 版本为 1.10.6。

建议尽可能地与教程的开发环境保持一致（尤其是 Python 与 Django 版本），避免不必要的麻烦。推荐的 Python 版本为 Python 3.4 或以上，Django 版本号必须为 Django 1.10.x。

**注意：本教程将尽量保证兼容 Python 2，但不会在 Python 2 环境下做测试。如果你因为某种原因必须使用 Python 2，你可以在 Python 2 环境下尝试运行教程里的代码，应该能够顺利运行。**

**安装 Python**

Windows 下安装 Python 非常简单，去 Python 官方网站找到 Python 3 的下载地址，根据你的系统选择 32 位或者 64 位的安装包，下载好后双击安装即可。

安装完后检测 Python 是否可以正常运行。在命令行输入 python -v ，如果输出了 Python 的版本号，说明 Python 已安装成功。

C:\WINDOWS\system32>python -V

Python 3.5.2

如果提示命令未找到，而你又确定已经安装了 Python，多半是因为没有把 Python 添加到环境变量。可搜索如何把 Python 添加到环境变量的相关教程，将安装的Python 添加到环境变量即可。

**使用虚拟环境 Virtualenv**

**强烈推荐在 Virtualenv 下进行 Django 的开发。**Virtualenv 是一个 Python 工具，使用它可以创建一个独立的 Python 环境。

为什么要使用 Virtualenv 呢？举个例子，假设你已经在系统中安装了 Python，并且在阅读此教程前你已经进行过一些 Django 的学习，但那时候安装的 Django 还是 1.8 版本。我们教程使用的是最新版的 Django 1.10.6 版本，你可能不愿意删除掉旧版的 Django 1.8，因为那可能导致你以前的项目无法运行。既想让原本项目在 Django 1.8 环境下运行，又想再安装 Django 1.10.6 来开启本教程的项目，怎么办呢？使用 Virtualenv 就能够完美解决这个问题。

Virtualenv 帮我们从系统的 Python 环境中克隆一个全新的 Python 环境出来，这个环境独立于原来的 Python 环境。我们可以在这个新克隆的环境下安装 Django 1.10.6，并且在这个新环境下运行我们的新项目。

Virtualenv 的使用非常简单，首先安装 Virtualenv，打开命令行工具，输入 pip install virtualenv 命令即可安装 Virtualenv。

C:\WINDOWS\system32>pip install virtualenv

安装成功后就可以开始创建虚拟环境，指定一个你喜欢的目录，Virtualenv 会把这个新的虚拟环境装到你指定目录下。例如我把它装到 C:\Users\yangxg\Envs 目录下，并将虚拟环境命名为 blogproject\_env（也可以取任何你喜欢的名字）。在命令栏运行如下命令：

C:\WINDOWS\system32>virtualenv C:\Users\yangxg\Envs\blogproject\_env

虚拟环境已经创建好了，我们需要激活这个环境，运行 blogproject\_env\Scripts\ 目录下的 activate 程序激活它：

C:\WINDOWS\system32>C:\Users\yangxg\Envs\blogproject\_env\Scripts\activate

(blogproject\_env) C:\WINDOWS\system32>

**注意 Linux 下没有 Scripts\ 这个目录，取而代之的是 bin/ 目录。且激活命令为：**

$ source blogproject\_env/bin/activate

可以看到命令提示符前面多了 (blogproject\_env)，说明我们已经成功激活了虚拟环境，接下来就可以开始安装 Django 了。

**安装 Django**

Django 的官方文档对 [如何安装 Django](https://docs.djangoproject.com/en/1.10/intro/install/#install-django) 给出了详细且明确的指导，不过我们目前用不上这些，只需使用 pip 命令就可以解决问题。

(blogproject\_env) C:\WINDOWS\system32>pip install django==1.10.6

我们用 django==1.10.6 来安装指定的 Django 版本以保证和教程中的一致。如果直接 pip install django 的话有可能安装最新的 Django 发行版本，而不是 Django 1.10.6。

**注意命名提示符前的 (blogproject\_env) 以确保你始终处在虚拟环境中，如果不小心退出了虚拟环境，先按上面的步骤重新进入再安装 Django。**

测试一下安装是否成功，先在命令行输入 python 以打开 Python 自带的命令栏，然后输入 import django，如果没有报错就说明 Django 安装成功。通过运行 print(django.get\_version()) 打印出 Django 的版本号，确保安装了正确版本的 Django。

(blogproject\_env) C:\WINDOWS\system32>python

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" **or** "license" **for** more information.

>>> **import** **django**

>>> **print**(django.get\_version())

1.10.6

>>>

**建立 Django 工程**

万事已经具备了，让我们来建立 Django 项目工程。

Django 工程（Project）是我们项目代码的容器，例如我们博客项目中所有的代码（包括 Django 为我们自动生成的以及我们自己写的）都包含在这个工程里。其实说通俗一点就是用一个文件夹把一系列 Python 代码文件和 Django 配置文件包裹起来，这个文件夹就可以看做一个 Django 工程。我们不必亲自动手新建这个文件夹和代码文件，Django 的内置命令已经帮我们做了这些事情。例如我把博客工程的代码放在 C:\Users\yangxg\Workspace\ ，工程名我把它叫做 blogproject 。

首先在命令行进入到 C:\Users\yangxg\Workspace 这个目录（如果你还停留在 Python 的命令栏里，先多按几次 Ctrl + c 键跳出来）：

(blogproject\_env) C:\WINDOWS\system32>cd C:\Users\yangxg\Workspace

然后运行如下命令创建工程：

(blogproject\_env) C:\Users\yangxg\Workspace>django-admin startproject blogproject

**再次申明一定要在虚拟环境下运行这些命令，虚拟环境的标志就是命令提示符前有 (blogproject\_env) 这样的标记。以后几乎所有和 Django 有关的命令都是在虚拟环境下运行，因此在运行前一定确保先开启了虚拟环境。**

进入工程所在目录 C:\Users\yangxg\Workspace\（你可能设置在其它路径），会发现多了一个 blogproject\ 的目录，其内部的文件结构如下：

blogproject\

manage.py

blogproject\

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

最顶层的 blogproject\ 目录是我们刚刚指定的工程目录。blogproject\ 目录下面有一个 manage.py 文件，manage 是管理的意思，顾名思义 manage.py 就是 Django 为我们生成的管理这个项目的 Python 脚本文件，以后用到时会再次介绍。与 manage.py 同级的还有一个 blogproject\ 的目录，这里面存放了一些 Django 的配置文件，例如 settings.py、urls.py 等等，以后用到时会详细介绍。

**Hello Django**

网站需要运行在一个 Web 服务器上，Django 已经为我们提供了一个用于本地开发的 Web 服务器。在命令行工具里进入到 manage.py 所在目录，即**最外层**的 blogproject\ 目录下。运行 python manage.py runserver 命令就可以在本机上开启一个 Web 服务器：

(blogproject\_env) C:\Users\yangxg\Workspace>cd C:\Users\yangxg\Workspace\blogproject

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>python manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

December 21, 2016 - 20:23:07

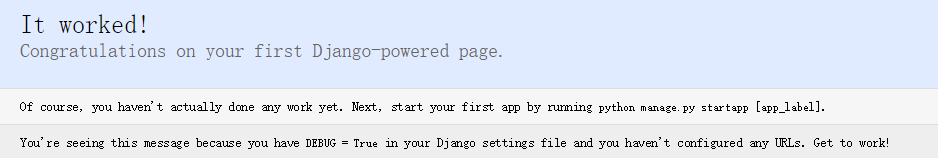
Django version 1.10.4, using settings 'blogproject.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

看到这样的信息表明我们的服务器开启成功。

在浏览器输入 http://127.0.0.1:8000/ ，看到如下的页面提示信息：



It worked! Django 工作了！

**注意：如果在浏览器输入 http://127.0.0.1:8000/ 后显示无法访问该网站，请检查是不是浏览器代理的问题。比如开启了某些 VPN 代理服务等，将它们全部关闭即可。**

这是 manage.py 的第一个用法，运行它的 runserver 命令开启本地开发服务器，以后我们还会遇到更多的命令。

命令栏工具下按 Ctrl + c 可以退出开发服务器（按一次没用的话连续多按几次）。重新开启则再次运行 python manage.py runserver 。

Django 默认的语言是英语，所以显示给我们的欢迎页面是英文的。我们在 Django 的配置文件里稍作修改，让它支持中文。用任何一个文本编辑器打开 settings.py 文件，找到如下的两行代码：

blogproject/blogproject/settings.py

*## 其它配置代码...*

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'UTC'

*## 其它配置代码...*

把 LANGUAGE\_CODE 的值改为 zh-hans，TIME\_ZONE 的值改为 Asia/Shanghai：

blogproject/blogproject/settings.py

*## 其它配置代码...*

*# 把英文改为中文*

LANGUAGE\_CODE = 'zh-hans'

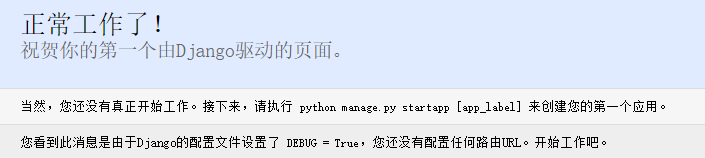
*# 把国际时区改为中国时区*

TIME\_ZONE = 'Asia/Shanghai'

*## 其它配置代码...*

**保存更改后**关闭 settings.py 文件。

再次运行开发服务器，并在浏览器打开 http://127.0.0.1:8000/，可以看到 Django 已经支持中文了。



一切准备就绪，开始进入我们的 Django 博客开发之旅吧！

**2 - 建立 Django 博客应用**

1 月，2 周前 字数 1932 阅读 1185 评论 0

**建立博客应用**

我们已经建立了 Django 博客的项目工程，并且成功地运行了它。不过到目前为止这一切都还只是 Django 为我们创建的项目初始内容，Django 不可能为我们初始化生成博客代码，这些功能性代码都得由我们自己编写。

Django 鼓励我们把自己编写的代码组织到应用（Application）里，并且最好是一个应用只提供一种功能。例如我们要开发的 Django 博客，相关的代码都放在 blog 这个应用里。其实应用也没什么复杂的，不过是把功能相关的代码组织到一个文件夹里，这个文件夹就成了一个应用（姑且可以这样理解）。

我们可以在 Django 工程目录下手动地建立一个文件夹，为它取个名字，并且告诉 Django 说这是一个应用。但是我们不必做这些无聊的工作，Django 已经为我们提供了自动创建应用的命令。

现在就来创建我们的 Django 博客应用，我把它命名为 blog。**激活虚拟环境**，进入到 manage.py 文件所在的目录下，运行 python manage.py startapp blog 命令即可建立一个 blog 应用：

C:\WINDOWS\system32>C:\Users\yangxg\Envs\blogproject\_env\Scripts\activate

(blogproject\_env) C:\WINDOWS\system32>cd C:\Users\yangxg\Workspace\blogproject

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>python manage.py startapp blog

前面已经说过，manage.py 是我们的工程管理助手。上一章中我们使用了它的 runserver 命令开启了一个本地开发服务器，在这里我们使用 manage.py 的 startapp 命令创建了一个 blog 应用。

**应用的目录结构**

用资源管理器打开 manage.py 文件所在的目录，可以看到多了一个 blog\ 的目录，该目录下面的文件结构如下：

blog\

\_\_init\_\_.py

admin.py

apps.py

migrations\

\_\_init\_\_.py

models.py

tests.py

views.py

不同名称的文件用于存放特定功能的代码，这些将会在后面详细介绍。总之这个应用的文件夹结构 Django 已经为我们建立好了，但它还只是包含各种文件的一个文件夹而已，Django 目前还不知道这是一个应用。我们得告诉 Django 这是我们建立的应用，专业一点说就是在 Django 的配置文件中注册这个应用。

打开 blogproject\ 目录下的 settings.py 文件，看名字就知道 settings.py 是一个设置文件（setting 意为设置），找到 INSTALLED\_APPS 设置项，将 blog 应用添加进去。

blogproject/blogproject/settings.py

*## 其他配置项...*

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog', *# 注册 blog 应用*

]

*## 其他配置项...*

可以看到 Django 已经为我们提供了一些内置的应用，这些应用提供了各种各样的功能。这就是 Django 强大的地方，通用的功能 Django 都帮我们写好了，我们只需编写与自身业务相关的功能代码即可。

# 3 - 创建 Django 博客的数据库模型

1 月，2 周前 字数 6862 阅读 1548 评论 1

## 设计博客的数据库表结构

博客最主要的功能就是展示我们写的文章，它需要从某个地方获取博客文章数据才能把文章展示出来，通常来说这个地方就是数据库。我们把写好的文章永久地保存在数据库里，当用户访问我们的博客时，Django 就去数据库里把这些数据取出来展现给用户。

博客的文章应该含有标题、正文、作者、发表时间等数据。一个更加现代化的博客文章还希望它有分类、标签、评论等。为了更好地存储这些数据，我们需要合理地组织数据库的表结构。

我们的博客初级版本主要包含博客文章，文章会有分类以及标签。一篇文章只能有一个分类，但可以打上很多标签。

数据库存储的数据其实就是表格的形式，例如存储博客文章的数据库表长这个样子：

| **文章 id** | **标题** | **正文** | **发表时间** | **分类** | **标签** |
| --- | --- | --- | --- | --- | --- |
| 1 | title 1 | text 1 | 2016-12-23 | Django | Django 学习 |
| 2 | title 2 | text 2 | 2016-12-24 | Django | Django 学习 |
| 3 | title 3 | text 3 | 2016-12-26 | Python | Python 学习 |

其中文章 ID 是一个数字，唯一对应着一篇文章。当然还可以有更多的列以存储更多相关数据，这只是一个最基本的示例。

数据库表设计成这样其实已经可以了，但是稍微分析一下我们就会发现一个问题，这 3 篇文章的分类和标签都是相同的，这会产生很多重复数据，当数据量很大时就浪费了存储空间。

不同的文章可能它们对应的分类或者标签是相同的，所以我们把分类和标签提取出来，做成单独的数据库表，再把文章和分类、标签关联起来。下面分别是分类和标签的数据库表：

| **分类 id** | **分类名** |
| --- | --- |
| 1 | Django |
| 2 | Python |
| **标签 id** | **标签名** |
| 1 | Django 学习 |
| 2 | Python 学习 |

## 编写博客模型代码

以上是自然语言描述的表格，数据库也和编程语言一样，有它自己的一套规定的语法来生成上述的表结构，这样我们才能把数据存进去。一般来说这时候我们应该先去学习数据库创建表格的语法，再回来写我们的 Django 博客代码了。但是 Django 告诉我们不用这么麻烦，它已经帮我们做了一些事情。Django 把那一套数据库的语法转换成了 Python 的语法形式，我们只要写 Python 代码就可以了，Django 会把 Python 代码翻译成对应的数据库操作语言。用更加专业一点的说法，就是 Django 为我们提供了一套 ORM（Object Relational Mapping）系统。

例如我们的分类数据库表，Django 只要求我们这样写：

blog/models.py

**from** **django.db** **import** models

**class** **Category**(models.Model):

name = models.CharField(max\_length=100)

Category 就是一个标准的 Python 类，它继承了 models.Model 类，类名为 Category 。Category 类有一个属性 name，它是 models.CharField 的一个实例。

这样，Django 就可以把这个类翻译成数据库的操作语言，在数据库里创建一个名为 category 的表格，这个表格的一个列名为 name。还有一个列 id，Django 则会自动创建。可以看出从 Python 代码翻译成数据库语言时其规则就是一个 Python 类对应一个数据库表格，类名即表名，类的属性对应着表格的列，属性名即列名。

我们需要 3 个表格：文章（Post）、分类（Category）以及标签（Tag），下面就来分别编写它们对应的 Python 类。模型的代码通常写在相关应用的 models.py 文件里。已经在代码中做了详细的注释，说明每一句代码的含义。但如果你在移动端下阅读不便的话，也可以跳到代码后面看正文的里的讲解。

**注意：代码中含有中文注释，如果你直接 copy 代码到你的文本编辑器且使用了 Python 2 开发环境的话，会得到一个编码错误。因此请在文件最开始处加入编码声明：# coding: utf-8。**

blog/models.py

**from** **django.db** **import** models

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

**class** **Category**(models.Model):

*"""*

*Django 要求模型必须继承 models.Model 类。*

*Category 只需要一个简单的分类名 name 就可以了。*

*CharField 指定了分类名 name 的数据类型，CharField 是字符型，*

*CharField 的 max\_length 参数指定其最大长度，超过这个长度的分类名就不能被存入数据库。*

*当然 Django 还为我们提供了多种其它的数据类型，如日期时间类型 DateTimeField、整数类型 IntegerField 等等。*

*Django 内置的全部类型可查看文档：*

*https://docs.djangoproject.com/en/1.10/ref/models/fields/#field-types*

*"""*

name = models.CharField(max\_length=100)

**class** **Tag**(models.Model):

*"""*

*标签 Tag 也比较简单，和 Category 一样。*

*再次强调一定要继承 models.Model 类！*

*"""*

name = models.CharField(max\_length=100)

**class** **Post**(models.Model):

*"""*

*文章的数据库表稍微复杂一点，主要是涉及的字段更多。*

*"""*

*# 文章标题*

title = models.CharField(max\_length=70)

*# 文章正文，我们使用了 TextField。*

*# 存储比较短的字符串可以使用 CharField，但对于文章的正文来说可能会是一大段文本，因此使用 TextField 来存储大段文本。*

body = models.TextField()

*# 这两个列分别表示文章的创建时间和最后一次修改时间，存储时间的字段用 DateTimeField 类型。*

created\_time = models.DateTimeField()

modified\_time = models.DateTimeField()

*# 文章摘要，可以没有文章摘要，但默认情况下 CharField 要求我们必须存入数据，否则就会报错。*

*# 指定 CharField 的 blank=True 参数值后就可以允许空值了。*

excerpt = models.CharField(max\_length=200, blank=True)

*# 这是分类与标签，分类与标签的模型我们已经定义在上面。*

*# 我们在这里把文章对应的数据库表和分类、标签对应的数据库表关联了起来，但是关联形式稍微有点不同。*

*# 我们规定一篇文章只能对应一个分类，但是一个分类下可以有多篇文章，所以我们使用的是 ForeignKey，即一对多的关联关系。*

*# 而对于标签来说，一篇文章可以有多个标签，同一个标签下也可能有多篇文章，所以我们使用 ManyToManyField，表明这是多对多的关联关系。*

*# 同时我们规定文章可以没有标签，因此为标签 tags 指定了 blank=True。*

*# 如果你对 ForeignKey、ManyToManyField 不了解，请看教程中的解释，亦可参考官方文档：*

*# https://docs.djangoproject.com/en/1.10/topics/db/models/#relationships*

category = models.ForeignKey(Category)

tags = models.ManyToManyField(Tag, blank=True)

*# 文章作者，这里 User 是从 django.contrib.auth.models 导入的。*

*# django.contrib.auth 是 Django 内置的应用，专门用于处理网站用户的注册、登录等流程，User 是 Django 为我们已经写好的用户模型。*

*# 这里我们通过 ForeignKey 把文章和 User 关联了起来。*

*# 因为我们规定一篇文章只能有一个作者，而一个作者可能会写多篇文章，因此这是一对多的关联关系，和 Category 类似。*

author = models.ForeignKey(User)

## 博客模型代码代码详解

首先是 Category 和 Tag 类，它们均继承自 model.Model 类，这是 Django 规定的。Category 和 Tag 类均有一个name 属性，用来存储它们的名称。由于分类名和标签名一般都是用字符串表示，因此我们使用了 CharField 来指定 name 的数据类型，同时 max\_length 参数则指定 name 允许的最大长度，超过该长度的字符串将不允许存入数据库。除了 CharField ，Django 还为我们提供了更多内置的数据类型，比如时间类型 DateTimeField、整数类型 IntegerField 等等。

**在本教程中我们会教你这些类型的使用方法，但以后你开发自己的项目时，你就需要通过阅读Django 官方文档**[**关于字段类型的介绍**](https://docs.djangoproject.com/en/1.10/ref/models/fields/)**来了解有哪些数据类型可以使用以及如何使用它们。**

Post 类也一样，必须继承自 model.Model 类。文章的数据库表稍微复杂一点，主要是列更多，我们指定了这些列：

* title。这是文章的标题，数据类型是 CharField，允许的最大长度 max\_length = 70。
* body。文章正文，我们使用了 TextField。比较短的字符串存储可以使用 CharField，但对于文章的正文来说可能会是一大段文本，因此使用 TextField 来存储大段文本。
* created\_time、modified\_time。这两个列分别表示文章的创建时间和最后一次修改时间，存储时间的列用 DateTimeField 数据类型。
* excerpt。文章摘要，可以没有文章摘要，但默认情况下 CharField 要求我们必须存入数据，否则就会报错。指定 CharField 的 blank=True 参数值后就可以允许空值了。
* category 和 tags。这是分类与标签，分类与标签的模型我们已经定义在上面。我们把文章对应的数据库表和分类、标签对应的数据库表关联了起来，但是关联形式稍微有点不同。我们规定一篇文章只能对应一个分类，但是一个分类下可以有多篇文章，所以我们使用的是 ForeignKey，即一对多的关联关系。而对于标签来说，一篇文章可以有多个标签，同一个标签下也可能有多篇文章，所以我们使用 ManyToManyField，表明这是多对多的关联关系。同时我们规定文章可以没有标签，因此为标签 tags 指定了 blank=True。
* author。文章作者，这里 User 是从 django.contrib.auth.models 导入的。django.contrib.auth 是 Django 内置的应用，专门用于处理网站用户的注册、登录等流程。其中 User 是 Django 为我们已经写好的用户模型，和我们自己编写的 Category 等类是一样的。这里我们通过 ForeignKey 把文章和 User关联了起来，因为我们规定一篇文章只能有一个作者，而一个作者可能会写多篇文章，因此这是一对多的关联关系，和 Category 类似。

## 理解多对一和多对多两种关联关系

我们分别使用了两种关联数据库表的形式：ForeignKey 和 ManyToManyField。

### ForeignKey

ForeignKey 表明一种一对多的关联关系。比如这里我们的文章和分类的关系，一篇文章只能对应一个分类，而一个分类下可以有多篇文章。反应到数据库表格中，它们的实际存储情况是这样的：

| **文章 ID** | **标题** | | **正文** | **分类 ID** |
| --- | --- | --- | --- | --- |
| 1 | title 1 | | body 1 | 1 |
| 2 | title 2 | | body 2 | 1 |
| 3 | title 3 | | body 3 | 1 |
| 4 | title 4 | | body 4 | 2 |
| **分类 ID** | | **分类名** | |  |
| 1 | | Django | |  |
| 2 | | Python | |  |

可以看到文章和分类实际上是通过文章数据库表中 **分类 ID** 这一列关联的。当要查询文章属于哪一个分类时，只需要查看其对应的分类 ID 是多少，然后根据这个分类 ID 就可以从分类数据库表中找到该分类的数据。例如这里文章 1、2、3 对应的分类 ID 均为 1，而分类 ID 为 1 的分类名为 Django，所以文章 1、2、3 属于分类 Django。同理文章 4 属于分类 Python。

反之，要查询某个分类下有哪些文章，只需要查看对应该分类 ID 的文章有哪些即可。例如这里 Django 的分类 ID 为 1，而对应分类 ID 为 1 的文章有文章 1、2、3，所以分类 Django 下有 3 篇文章。

希望这个例子能帮助你加深对多对一关系，以及它们在数据库中是如何被关联的理解，更多的例子请看文末给出的 Django 官方参考资料。

### ManyToManyField

ManyToManyField 表明一种多对多的关联关系，比如这里的文章和标签，一篇文章可以有多个标签，而一个标签下也可以有多篇文章。反应到数据库表格中，它们的实际存储情况是这样的：

| **文章 ID** | **标题** | | **正文** |
| --- | --- | --- | --- |
| 1 | title 1 | | body 1 |
| 2 | title 2 | | body 2 |
| 3 | title 3 | | body 3 |
| 4 | title 4 | | body 4 |
| **标签 ID** | | **标签名** | |
| 1 | | Django 学习 | |
| 2 | | Python 学习 | |

| **文章 ID** | **标签 ID** |
| --- | --- |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 3 | 2 |

多对多的关系无法再像一对多的关系中的例子一样在文章数据库表加一列 **分类 ID** 来关联了，因此需要额外建一张表来记录文章和标签之间的关联。例如**文章 ID** 为 1 的文章，既对应着 **标签 ID** 为 1 的标签，也对应着 **标签 ID** 为 2 的标签，即文章 1 既属于标签 1：Django 学习，也属于标签 2：Python 学习。

反之，**标签 ID** 为 1 的标签，既对应着 **文章 ID** 为 1 的文章，也对应着 **文章 ID** 为 2 的文章，即标签 1：Django 学习下有两篇文章。

希望这个例子能帮助你加深对多对多关系，以及它们在数据库中是如何被关联的理解，更多的例子请看文末给出的 Django 官方参考资料。

假如你对多对一关系和多对多关系还存在一些困惑，强烈建议阅读官方文档对这两种关系的说明以及更多官方的例子以加深理解：

* [Django ForeignKey 简介](https://docs.djangoproject.com/en/1.10/topics/db/models/#relationships)
* [Django ForeignKey 详细示例](https://docs.djangoproject.com/en/1.10/topics/db/examples/many_to_one/)
* [Django ManyToManyField 简介](https://docs.djangoproject.com/en/1.10/topics/db/models/#many-to-many-relationships)
* [Django ManyToManyField 详细示例](https://docs.djangoproject.com/en/1.10/topics/db/examples/many_to_many/)

# 4 - 让 Django 完成翻译：迁移数据库

1 月，1 周前 字数 8734 阅读 1354 评论 6

我们已经编写了博客数据库模型的代码，但那还只是 Python 代码而已，Django 还没有把它翻译成数据库语言，因此实际上这些数据库表还没有真正的在数据库中创建。

## 迁移数据库

为了让 Django 完成翻译，创建好这些数据库表，我们再一次请出我的工程管理助手 manage.py。**激活虚拟环境**，切换到 manage.py 文件所在的目录下，分别运行 python manage.py makemigrations 和 python manage.py migrate 命令：

C:\WINDOWS\system32>C:\Users\yangxg\Envs\blogproject\_env\Scripts\activate

(blogproject\_env) C:\WINDOWS\system32>cd C:\Users\yangxg\Workspace\blogproject

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>python manage.py makemigrations

Migrations for 'blog':

blog\migrations\0001\_initial.py:

- Create model Category

- Create model Post

- Create model Tag

- Add field tags to post

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, blog, contenttypes, sessions

Running migrations:

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying auth.0008\_alter\_user\_username\_max\_length... OK

Applying blog.0001\_initial... OK

Applying sessions.0001\_initial... OK

**注意：如果代码中含有中文注释，且你使用的是 Python 2 开发环境的话，会得到一个编码错误。因此请在含有中文注释的文件最开始处加入编码声明：# coding: utf-8。**

当我们执行了 python manage.py makemigrations 后，Django 在 blog 应用的 migrations\ 目录下生成了一个 0001\_initial.py 文件，这个文件是 Django 用来记录我们对模型做了哪些修改的文件。目前来说，我们在 models.py 文件里创建了 3 个模型类，Django 把这些变化记录在了 0001\_initial.py 里。

不过此时还只是告诉了 Django 我们做了哪些改变，为了让 Django 真正地为我们创建数据库表，接下来又执行了 python manage.py migrate 命令。Django 通过检测应用中 migrations\ 目录下的文件，得知我们对数据库做了哪些操作，然后它把这些操作翻译成数据库操作语言，从而把这些操作作用于真正的数据库。

你可以看到命令的输出除了 Applying blog.0001\_initial... OK 外，Django 还对其它文件做了操作。这是因为除了我们自己建立的 blog 应用外，Django 自身还内置了很多应用，这些应用本身也是需要存储数据的。可以在 settings.py 的 INSTALLED\_APP 设置里看到这些应用，当然我们目前不必关心这些。

blogproject/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog',

]

对于了解数据库语言的人，你可以运行下面的命令看看 Django 究竟为我们做了什么：

python manage.py sqlmigrate blog 0001

你将看到输出了经 Django 翻译后的数据库表创建语句，这有助于你理解 Django ORM 的工作机制。

## 选择数据库版本

我们没有安装任何的数据库软件，Django 就帮我们迁移了数据库。这是因为我们使用了 Python 内置的 SQLite3 数据库。

SQLite3 是一个十分轻巧的数据库，它仅有一个文件。你可以看一到项目根目录下多出了一个 db.sqlite3 的文件，这就是 SQLite3 数据库文件，Django 博客的数据都会保存在这个数据库文件里。

Django 在 settings.py 里为我们做了一些默认的数据库配置：

blogproject/settings.py

*## 其它配置选项...*

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

*## 其它配置选项...*

可以看到默认的数据库引擎就是使用的 SQLite3。

当然一些人倾向于使用 MySQL 等大型数据库，至于 Django 如何配置 MySQL 这里就不赘述了，你可以自行使用搜索引擎或者查阅 Django 的官方文档解决。对于一个小型博客而言，SQLite3 数据库足以胜任。

## 用 Django 的方式操作数据库

数据库最主要的操作就是往里面存入数据、从中取出数据、修改已保存的数据和删除不再需要的数据。和创建数据库表一样，Django 为这些操作提供了一整套方法，从而把我们从数据库语言中解放出来。我们不用学习如何利用数据库语言去完成这些操作，只要简单地调用几个 Python 函数就可以满足我们的需求。

### 存数据

先在命令行中来探索一下这些函数，感受一下如何用 Django 的方式来操作数据库。在 manage.py 所在目录下运行 python manage.py shell 命令：

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>python manage.py shell

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

>>>

这打开了一个交互式命令行。

首先我们来创建一个分类和一个标签：

>>> **from** **blog.models** **import** Category, Tag, Post

>>> c = Category(name='category test')

>>> c.save()

>>> t = Tag(name='tag test')

>>> t.save()

我们首先导入 3 个之前写好的模型类，然后实例化了一个 Category 类和一个 Tag 类，为他们的属性 name 赋了值。为了让 Django 把这些数据保存进数据库，调用实例的 save 方法即可。

再创建一篇文章试试，但创建文章之前，我们需要先创建一个 User，用于指定文章的作者。创建 User 的命令 Django 已经帮我们写好了，依然是通过 manage.py 来运行。首先按住 Ctrl + c 退出命令交互栏（一次退不出就连续多按几次），运行 python manage.py createsuperuser 命令并根据提示创建用户：

(blogproject\_env) **C**:\Users\yangxg\Workspace\blogproject>python manage.py createsuperuser

Username (leave blank to use 'zmrenwu@163.com'): myuser

Email **address**: a@aa.com

**Password**:

Password (again):

Superuser created successfully.

运行 python manage.py createsuperuser 开始创建用户，之后会提示你输入用户名、邮箱、密码和确认密码，按照提示输入即可。**注意一点的是密码输入过程中不会有任何字符显示，不要误以为你的键盘出问题了，正常输入即可。**最后出现 Superuser created successfully. 说明用户创建成功了。

再次运行 python manage.py shell 进入 Python 命令交互栏，开始创建文章：

>>> **from** **blog.models** **import** Category, Tag, Post

>>> **from** **django.utils** **import** timezone

>>> **from** **django.contrib.auth.models** **import** User

>>> user = User.objects.get(username='myuser')

>>> c = Category.objects.get(name='category test')

>>> p = Post(title='title test', body='body test', created\_time=timezone.now(), modified\_time=timezone.now(), category=c, author=user)

>>> p.save()

由于我们重启了 shell，因此需要重新导入了 Category、Tag、Post 以及 User。我们还导入了一个 Django 提供的辅助模块 timezone，这是因为我们需要调用它的 now() 方法为 created\_time 和 modified\_time 指定时间，容易理解 now 方法返回当前时间。然后我们根据用户名和分类名，通过 get 方法取出了存在数据库中的 User 和 Category（取数据的方法将在下面介绍）。接着我们为文章指定了 title、body 、created\_time、modified\_time值，并把它和前面创建的 Category 以及 User 关联了起来。允许为空 excerpt、tags 我们就没有为它们指定值了。

**注意：我们这里使用 get 方法根据 Category 的 name 属性的值获取分类的一条记录。Category.objects.get(name='category test') 的含义是从数据库中取出 name 的值为 category test 的分类记录。确保数据库中只有一条值为 category test 的记录，否则 get 方法将返回一个 MultipleObjectsReturned 异常。如果你不小心已经存了多条记录，请删掉多余的记录。如何删除数据请看下文。**

### 取数据

数据已经存入数据库了，现在要把它们取出来看看：

>>> Category.objects.all()

<QuerySet [<Category: Category object>]>

>>> Tag.objects.all()

<QuerySet [<Tag: Tag object>]>

>>> Post.objects.all()

<QuerySet [<Post: Post object>]>

>>>

objects 是我们的模型管理器，它为我们提供一系列从数据库中取数据方法，这里我们使用了 all 方法，表示我们要把对应的数据全部取出来。可以看到 all方法都返回了数据，这些数据应该是我们之前存进去的，但是显示的字符串有点奇怪，无法看出究竟是不是我们之前存入的数据。为了让显示出来的数据更加人性化一点，我们为 3 个模型分别增加一个 \_\_str\_\_ 方法：

blog/models.py

**from** **django.utils.six** **import** python\_2\_unicode\_compatible

*# python\_2\_unicode\_compatible 装饰器用于兼容 Python2*

**@python\_2\_unicode\_compatible**

**class** **Category**(models.Model):

...

**def** \_\_str\_\_(self):

**return** self.name

**@python\_2\_unicode\_compatible**

**class** **Tag**(models.Model):

...

**def** \_\_str\_\_(self):

**return** self.name

**@python\_2\_unicode\_compatible**

**class** **Post**(models.Model):

...

**def** \_\_str\_\_(self):

**return** self.title

定义好 \_\_str\_\_ 方法后，解释器显示的内容将会是 \_\_str\_\_ 方法返回的内容。这里 Category 返回分类名 name ，Tag 返回标签名，而 Post 返回它的 title。

python\_2\_unicode\_compatible 装饰器用于兼容 Python2。如果你使用的 Python3 开发环境，去掉这个装饰器不会有任何影响。如果你使用的 Python2 开发环境，而又不想使用这个装饰器，则将 \_\_str\_\_ 方法改为 \_\_unicode\_\_ 方法即可。

先按 Ctrl + c 退出 Shell，再重新运行 python manage.py shell 进入 Shell。

>>> **from** **blog.models** **import** Category, Tag, Post

>>> Category.objects.all()

<QuerySet [<Category: category test>]>

>>> Tag.objects.all()

<QuerySet [<Tag: tag test>]>

>>> Post.objects.all()

<QuerySet [<Post: title test>]>

>>> Post.objects.get(title='title test')

<Post: title test>

可以看到返回的是我们之前存入的数据。

此外我们在创建文章时提到了通过 get 方法来获取数据，这里 all 方法和 get方法的区别是：all 方法返回全部数据，是一个类似于列表的数据结构（QuerySet）；而 get 返回一条记录数据，如有多条记录或者没有记录，get方法均会抛出相应异常。

### 改数据

尝试修改数据：

>>> c = Category.objects.get(name='category test')

>>> c.name = 'category test new'

>>> c.save()

>>> Category.objects.all()

<QuerySet [<Category: test category new>]>

首先通过 get 方法根据分类名 name 获取值为 category test 到分类，修改它的 name 属性为新的值 category test new，然后调用 save 方法把修改保存到数据库，之后可以看到数据库返回的数据已经是修改后的值了。Tag、Post 的修改也一样。

### 删数据

删除掉数据：

>>> p = Post.objects.get(title='title test')

>>> p

<Post: title test>

>>> p.delete()

(1, {'blog.Post\_tags': 0, 'blog.Post': 1})

>>> Post.objects.all()

<QuerySet []>

先根据标题 title 的值从数据库中取出 Post，保存在变量 p 中，然后调用它的delete 方法，最后看到 Post.objects.all() 返回了一个空的 QuerySet（类似于一个列表），表明数据库中已经没有 Post，Post 已经被删除了。

这就是 Django 对数据库增、删、改、查的操作。除了上述演示的方法外，Django 还为我们提供了大量其它的方法，这些方法有一部分会在教程中使用，用到时我会讲解它们的用法。但以后你开发自己的项目时，你就需要通过阅读 [Django 的官方文档](https://docs.djangoproject.com/en/1.10/ref/models/querysets/) 来了解有哪些方法可用以及如何使用它们

# 5 - Django 博客首页视图

1 月，1 周前 字数 7083 阅读 1242 评论 0

## Django 处理 HTTP 请求

Web 应用的交互过程其实就是 HTTP 请求与响应的过程。无论是在 PC 端还是移动端，我们通常使用浏览器来上网，上网流程大致来说是这样的：

1. 我们打开浏览器，在地址栏输入想访问的网址，比如 http://zmrenwu.com/（当然你也可能从收藏夹里直接打开网站，但本质上都是一样的）。
2. 浏览器知道我们想要访问哪个网址后，它在后台帮我们做了很多事情。主要就是把我们的访问意图包装成一个 HTTP 请求，发给我们想要访问的网址所对应的服务器。通俗点说就是浏览器帮我们通知网站的服务器，说有人来访问你啦，访问的请求都写在 HTTP 里了，你按照要求处理后告诉我，我再帮你回应他！
3. 服务器处理了HTTP 请求，然后生成一段 HTTP 响应给浏览器。浏览器解读这个响应，把相关的内容在浏览器里显示出来，于是我们就看到了网站的内容。比如你访问了我的博客主页 http://zmrenwu.com/，服务器接收到这个请求后就知道用户访问的是首页，首页显示的是全部文章列表，于是它从数据库里把文章数据取出来，生成一个写着这些数据的 HTML 文档，包装到 HTTP 响应里发给浏览器，浏览器解读这个响应，把 HTML 文档显示出来，我们就看到了文章列表的内容。

因此，Django 作为一个 Web 框架，它的使命就是处理流程中的第二步。即接收浏览器发来的 HTTP 请求，返回相应的 HTTP 响应。于是引出这么几个问题：

1. Django 如何接收 HTTP 请求？
2. Django 如何处理这个 HTTP 请求？
3. Django 如何生成 HTTP 响应？

对于如何处理这些问题，Django 有其一套规定的机制。我们按照 Django 的规定，就能开发出所需的功能。

## Hello 视图函数

我们先以一个最简单的 Hello World 为例来看看 Django 处理上述问题的机制是怎么样的。

### 绑定 URL 与视图函数

首先 Django 需要知道当用户访问不同的网址时，应该如何处理这些不同的网址（即所说的路由）。Django 的做法是把不同的网址对应的处理函数写在一个 urls.py 文件里，当用户访问某个网址时，Django 就去会这个文件里找，如果找到这个网址，就会调用和它绑定在一起的处理函数（叫做视图函数）。

下面是具体的做法，**首先在 blog 应用的目录下创建一个 urls.py 文件**，这时你的目录看起来是这样：

blog\

\_\_init\_\_.py

admin.py

apps.py

migrations\

0001\_initial.py

\_\_init\_\_.py

models.py

tests.py

views.py

urls.py

在 blog\urls.py 中写入这些代码：

blog/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

urlpatterns = [

url(r'^$', views.index, name='index'),

]

我们首先从 django.conf.urls 导入了 url 函数，又从当前目录下导入了 views 模块。然后我们把网址和处理函数的关系写在了 urlpatterns 列表里。

绑定关系的写法是把网址和对应的处理函数作为参数传给 url 函数（第一个参数是网址，第二个参数是处理函数），另外我们还传递了另外一个参数 name，这个参数的值将作为处理函数 index 的别名，这在以后会用到。

注意这里我们的网址是用正则表达式写的，Django 会用这个正则表达式去匹配用户实际输入的网址，如果匹配成功，就会调用其后面的视图函数做相应的处理。

比如说我们本地开发服务器的域名是 http://127.0.0.1:8000，那么当用户输入网址 http://127.0.0.1:8000 后，Django 首先会把协议 http、域名 127.0.0.1 和端口号 8000 去掉，此时只剩下一个空字符串，而 r'^$' 的模式正是匹配一个空字符串（这个正则表达式的意思是以空字符串开头且以空字符串结尾），于是二者匹配，Django 便会调用其对应的 views.index 函数。

**注意：在项目根目录的 blogproject\ 目录下（即 settings.py 所在的目录），原本就有一个 urls.py 文件，这是整个工程项目的 URL 配置文件。而我们这里新建了一个 urls.py 文件，且位于 blog 应用下。这个文件将用于 blog 应用相关的 URL 配置。不要把两个文件搞混了。**

### 编写视图函数

第二步就是要实际编写我们的 views.index 视图函数了，按照惯例视图函数定义在 views.py 文件里：

blog/views.py

**from** **django.http** **import** HttpResponse

**def** index(request):

**return** HttpResponse("欢迎访问我的博客首页！")

我们前面说过，Web 服务器的作用就是接收来自用户的 HTTP 请求，根据请求内容作出相应的处理，并把处理结果包装成 HTTP 响应返回给用户。

这个两行的函数体现了这个过程。它首先接受了一个名为 request 的参数，这个request 就是 Django 为我们封装好的 HTTP 请求，它是类 HttpRequest 的一个实例。然后我们便直接返回了一个 HTTP 响应给用户，这个 HTTP 响应也是 Django 帮我们封装好的，它是类 HttpResponse 的一个实例，只是我们给它传了一个自定义的字符串参数。

浏览器接收到这个响应后就会在页面上显示出我们传递的内容 ：欢迎访问我的博客首页！

### 配置项目 URL

还差最后一步了，我们前面建立了一个 urls.py 文件，并且绑定了 URL 和视图函数 index，但是 Django 并不知道。Django 匹配 URL 模式是在 blogproject\ 目录（即 settings.py 文件所在的目录）的 urls.py 下的，所以我们要把 blog 应用下的 urls.py 文件包含到 blogproject\urls.py 里去，打开这个文件看到如下内容：

blogproject/urls.py

*"""*

*一大段注释*

*"""*

**from** **django.conf.urls** **import** url

**from** **django.contrib** **import** admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

]

修改成如下的形式：

- **from** **django.conf.urls** **import** url

+ **from** **django.conf.urls** **import** url, include

**from** **django.contrib** **import** admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

+ url(r'', include('blog.urls')),

]

这里 - 表示删掉这一行，+ 表示添加这一行。

我们这里导入了一个 include 函数，然后利用这个函数把 blog 应用下的 urls.py 文件包含了进来。此外 include 前还有一个 r''，这是一个空字符串。这里也可以写其它字符串，Django 会把这个字符串和后面 include 的 urls.py 文件中的 URL 拼接。比如说如果我们这里把 r'' 改成 r'blog/'，而我们在 blog.urls 中写的 URL 是 r'^$'，即一个空字符串。那么 Django 最终匹配的就是 blog/ 加上一个空字符串，即 blog/。

### 运行结果

**激活虚拟环境**，运行 python manage.py runserver 打开开发服务器，在浏览器输入开发服务器的地址 http://127.0.0.1:8000/，可以看到 Django 返回的内容了。

欢迎访问我的博客首页！

## 使用 Django 模板系统

这基本上就上 Django 的开发流程了，写好处理 HTTP 请求和返回 HTTP 响应的视图函数，然后把视图函数绑定到相应的 URL 上。

但是等一等！我们看到在视图函数里返回的是一个 HttpResponse 类的实例，我们给它传入了一个希望显示在用户浏览器上的字符串。但是我们的博客不可能只显示这么一句话，它有可能会显示很长很长的内容。比如我们发布的博客文章列表，或者一大段的博客文章。我们不能每次都把这些大段大段的内容传给 HttpResponse。

Django 对这个问题给我们提供了一个很好的解决方案，叫做模板系统。Django 要我们把大段的文本写到一个文件里，然后 Django 自己会去读取这个文件，再把读取到的内容传给 HttpResponse。让我们用模板系统来改造一下上面的例子。

首先在我们的项目**根目录**（即 manage.py 文件所在目录）下建立一个名为 templates 的文件夹，用来存放我们的模板。然后在 templates\ 目录下建立一个名为 blog 的文件夹，用来存放和 blog 应用相关的模板。

当然模板存放在哪里是无关紧要的，只要 Django 能够找到的就好。但是我们建立这样的文件夹结构的目的是把不同应用用到的模板隔离开来，这样方便以后维护。我们在 templates\blog 目录下建立一个名为 index.html 的文件，此时你的目录结构应该是这样的：

blogproject\

manage.py

blogproject\

\_\_init\_\_.py

settings.py

...

blog\

\_\_init\_\_.py

models.py

,,,

templates\

blog\

index.html

**再一次强调 templates\ 目录位于项目根目录，而 index.html 位于 templates\blog 目录下，而不是 blog 应用下，如果弄错了你可能会得到一个TemplateDoesNotExist 异常。如果遇到这个异常，请回来检查一下模板目录结构是否正确。**

在 templates\blog\index.html 文件里写入下面的代码：

<!DOCTYPE html>

<**html** lang="en">

<**head**>

<**meta** charset="UTF-8">

<**title**>{{ title }}</**title**>

</**head**>

<**body**>

<**h1**>{{ welcome }}</**h1**>

</**body**>

</**html**>

这是一个标准的 HTML 文档，只是里面有两个比较奇怪的地方：{{ title }}，{{ welcome }}。这是 Django 规定的语法。用 {{ }} 包起来的变量叫做模板变量。Django 在渲染这个模板的时候会根据我们传递给模板的变量替换掉这些变量。最终在模板中显示的将会是我们传递的值。

**注意：index.html 必须以 UTF-8 的编码格式保存，且小心不要往里面添加一些特殊字符，否则极有可能得到一个 UnicodeDecodeError 这样的错误。**

模板写好了，还得告诉 Django 去哪里找模板，在 settings.py 文件里设置一下模板文件所在的路径。在 settings.py 找到 TEMPLATES 选项，它的内容是这样的：

blogproject/settings.py

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

其中 DIRS 就是设置模板的路径，在 [] 中写入 os.path.join(BASE\_DIR, 'templates')，即像下面这样：

blogproject/settings.py

TEMPLATES = [

{

...

'DIRS': [os.path.join(BASE\_DIR, 'templates')],

...

},

]

这里 BASE\_DIR 是 settings.py 在配置开头前面定义的变量，记录的是工程根目录 blogproject\ 的值（注意是最外层的 blogproject\ 目录）。在这个目录下有模板文件所在的目录 templates\，于是利用os.path.join 把这两个路径连起来，构成完整的模板路径，Django 就知道去这个路径下面找我们的模板了。

视图函数可以改一下了：

blog/views.py

**from** **django.http** **import** HttpResponse

**from** **django.shortcuts** **import** render

**def** index(request):

**return** render(request, 'blog/index.html', context={

'title': '我的博客首页',

'welcome': '欢迎访问我的博客首页'

})

这里我们不再是直接把字符串传给 HttpResponse 了，而是调用 Django 提供的 render 函数。这个函数根据我们传入的参数来构造 HttpResponse。

我们首先把 HTTP 请求传了进去，然后 render 根据第二个参数的值 blog/index.html 找到这个模板文件并读取模板中的内容。之后 render 根据我们传入的 context 参数的值把模板中的变量替换为我们传递的变量的值，{{ title }} 被替换成了 context 字典中 title 对应的值，同理 {{ welcome }} 也被替换成相应的值。

最终，我们的 HTML 模板中的内容字符串被传递给 HttpResponse 对象并返回给浏览器（Django 在 render 函数里隐式地帮我们完成了这个过程），这样用户的浏览器上便显示出了我们写的 HTML 模板的内容。

# 6 - 真正的 Django 博客首页视图

1 月，1 周前 字数 11252 阅读 1426 评论 2

在此之前我们已经编写了 Blog 的首页视图，并且配置了 URL 和模板，让 Django 能够正确地处理 HTTP 请求并返回合适的 HTTP 响应。不过我们仅仅在首页返回了一句话：欢迎访问我的博客。这是个 Hello World 级别的视图函数，我们需要编写真正的首页视图函数，当用户访问我们的博客首页时，他将看到我们发表的博客文章列表，就像 [演示项目](http://demo.zmrenwu.com/) 里展示的这样。

## 首页视图函数

上一节我们阐明了 Django 的开发流程。即首先配置 URL，把 URL 和相应的视图函数绑定，一般写在 urls.py 文件里，然后在工程的 urls.py 文件引入。其次是编写视图函数，视图中需要渲染模板，我们也在 settings.py 中进行了模板相关的配置，让 Django 能够找到需要渲染的模板。最后把渲染完成的 HTTP 响应返回就可以了。相关的配置和准备工作都在之前完成了，这里我们只需专心编写视图函数，让它实现我们想要的功能即可。

首页的视图函数其实很简单，代码像这样：

blog/views.py

**from** **django.shortcuts** **import** render

**from** **.models** **import** Post

**def** index(request):

post\_list = Post.objects.all().order\_by('-created\_time')

**return** render(request, 'blog/index.html', context={'post\_list': post\_list})

我们曾经在前面的章节讲解过模型管理器 objects 的使用。这里我们使用 all()方法从数据库里获取了全部的文章，存在了 post\_list 变量里。all 方法返回的是一个 QuerySet（可以理解成一个类似于列表的数据结构），由于通常来说博客文章列表是按文章发表时间倒序排列的，即最新的文章排在最前面，所以我们紧接着调用了 order\_by 方法对这个返回的 queryset 进行排序。排序依据的字段是 created\_time，即文章的创建时间。- 号表示逆序，如果不加 - 则是正序。 接着如之前所做，我们渲染了 blog\index.html 模板文件，并且把包含文章列表数据的 post\_list 变量传给了模板。

## 处理静态文件

我们的项目使用了从网上下载的一套博客模板（[点击这里下载全套模板](https://github.com/zmrenwu/django-blog-tutorial-templates)）。这里面除了 HTML 文档外，还包含了一些 CSS 文件和 JavaScript 文件以让网页呈现出我们现在看到的样式。同样我们需要对 Django 做一些必要的配置，才能让 Django 知道如何在开发服务器中引入这些 CSS 和 JavaScript 文件，这样才能让博客页面的 CSS 样式生效。

按照惯例，我们把 CSS 和 JavaScript 文件放在 **blog 应用**的 static\ 目录下。因此，先在 **blog 应用**下建立一个 static 文件夹。同时，为了避免和其它应用中的 CSS 和 JavaScript 文件命名冲突（别的应用下也可能有和 blog 应用下同名的 CSS 、JavaScript 文件），我们再在 static\ 目录下建立一个 blog 文件夹，把下载的博客模板中的 css 和 js 文件夹连同里面的全部文件一同拷贝进这个目录。最终我们的 blog 应用目录结构应该是这样的：

blog\

\_\_init\_\_.py

static\

blog\

css\

.css 文件...

js\

.js 文件...

admin.py

apps.py

migrations\

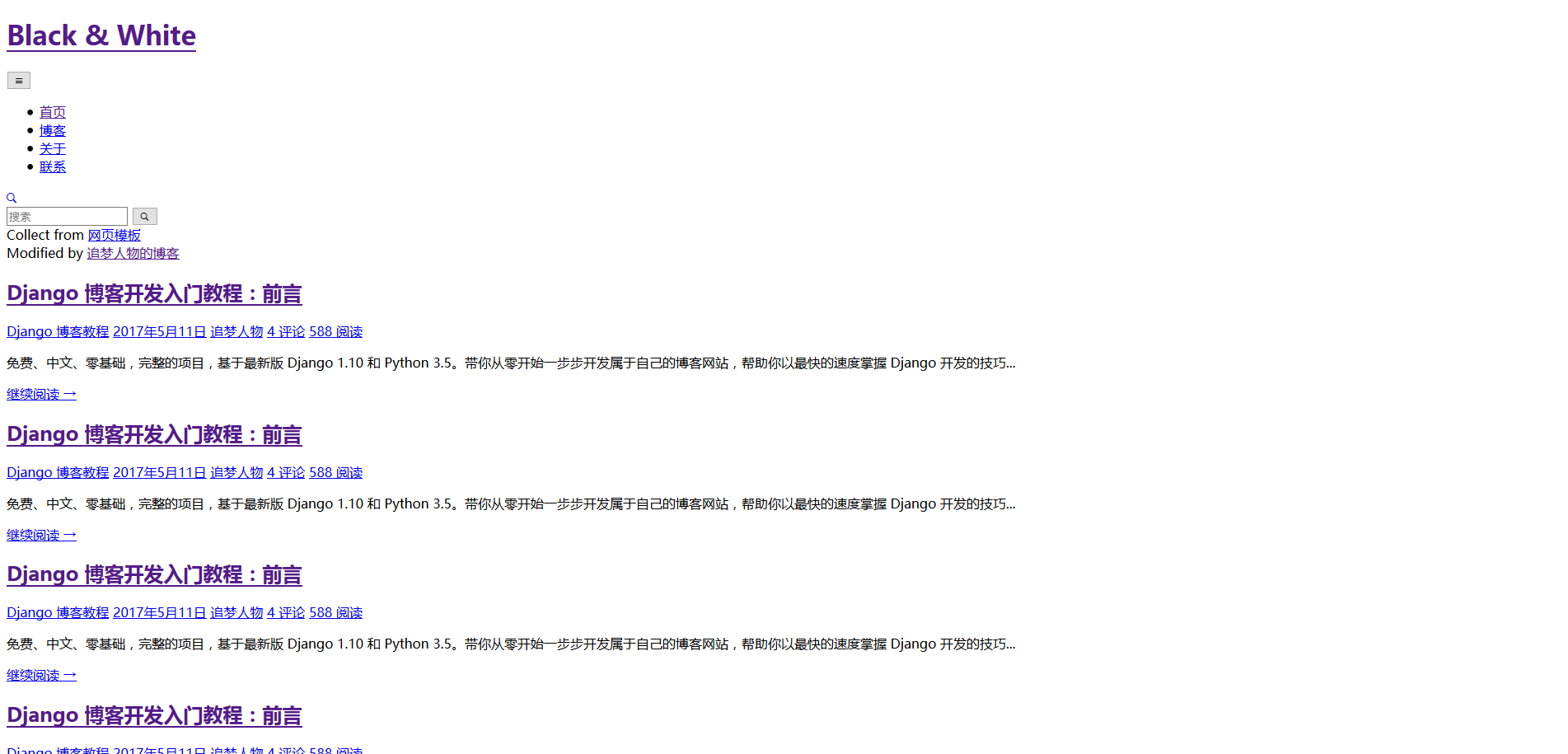
\_\_init\_\_.py

models.py

tests.py

views.py

用下载的博客模板中的 index.html 文件替换掉之前我们自己写的 index.html 文件。如果你好奇，现在就可以运行开发服务器，看看首页是什么样子。



如图所示，你会看到首页显示的样式非常混乱，原因是浏览器无法正确加载 CSS 等样式文件。需要以 Django 的方式来正确地处理 CSS 和 JavaScript 等静态文件的加载路径。CSS 样式文件通常在 HTML 文档的 head 标签里引入，打开 index.html 文件，在文件的开始处找到 head 标签包裹的内容，大概像这样：

templates/blog/index.html

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Black **&amp;** White</**title**>

*<!-- meta -->*

<**meta** charset="UTF-8">

<**meta** name="viewport" content="width=device-width, initial-scale=1">

*<!-- css -->*

<**link** rel="stylesheet" href="css/bootstrap.min.css">

<**link** rel="stylesheet" href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css">

<**link** rel="stylesheet" href="css/pace.css">

<**link** rel="stylesheet" href="css/custom.css">

*<!-- js -->*

<**script** src="js/jquery-2.1.3.min.js"></**script**>

<**script** src="js/bootstrap.min.js"></**script**>

<**script** src="js/pace.min.js"></**script**>

<**script** src="js/modernizr.custom.js"></**script**>

</**head**>

<**body**>

*<!-- 其它内容 -->*

<**script** src="js/script.js"></**script**>

</**body**>

</**html**>

CSS 样式文件的路径在 link 标签的 href 属性里，而 JavaScript 文件的路径在 script 标签的 src 属性里。可以看到诸如 `href="css/bootstrap.min.css" 或者 src="js/jquery-2.1.3.min.js" 这样的引用，由于引用文件的路径不对，所以浏览器引入这些文件失败。我们需要把它们改成正确的路径。把代码改成下面样子，正确地引入 static 文件下的 CSS 和 JavaScript 文件：

templates/blog/index.html

+ {% load staticfiles %}

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Black **&amp;** White</**title**>

*<!-- meta -->*

<**meta** charset="UTF-8">

<**meta** name="viewport" content="width=device-width, initial-scale=1">

*<!-- css -->*

- <**link** rel="stylesheet" href="css/bootstrap.min.css">

<**link** rel="stylesheet" href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css">

- <**link** rel="stylesheet" href="css/pace.css">

- <**link** rel="stylesheet" href="css/custom.css">

+ <**link** rel="stylesheet" href="{% static 'blog/css/bootstrap.min.css' %}">

+ <**link** rel="stylesheet" href="{% static 'blog/css/pace.css' %}">

+ <**link** rel="stylesheet" href="{% static 'blog/css/custom.css' %}">

*<!-- js -->*

- <**script** src="js/jquery-2.1.3.min.js"></**script**>

- <**script** src="js/bootstrap.min.js"></**script**>

- <**script** src="js/pace.min.js"></**script**>

- <**script** src="js/modernizr.custom.js"></**script**>

+ <**script** src="{% static 'blog/js/jquery-2.1.3.min.js' %}"></**script**>

+ <**script** src="{% static 'blog/js/bootstrap.min.js' %}"></**script**>

+ <**script** src="{% static 'blog/js/pace.min.js' %}"></**script**>

+ <**script** src="{% static 'blog/js/modernizr.custom.js' %}"></**script**>

</**head**>

<**body**>

*<!-- 其它内容 -->*

- <**script** src="js/script.js' %}"></**script**>

+ <**script** src="{% static 'blog/js/script.js' %}"></**script**>

</**body**>

</**html**>

这里 - 表示删掉这一行，而 + 表示增加这一行。

我们把引用路径放在了一个奇怪的符号里，例如：href="{% static 'blog/css/bootstrap.min.css' %}"。用 {% %} 包裹起来的叫做模板标签。我们前面说过用 {{ }} 包裹起来的叫做模板变量，其作用是在最终渲染的模板里显示由视图函数传过来的变量值。而这里我们使用的模板标签的功能则类似于函数，例如这里的 static 模板标签，它把跟在后面的字符串 'css/bootstrap.min.css' 转换成正确的文件引入路径。这样 css 和 js 文件才能被正确加载，样式才能正常显示。

**为了能在模板中使用 {% static %} 模板标签，别忘了在最顶部 {% load staticfiles %} 。static 模板标签位于 staticfiles 模块中，只有通过 load 模板标签将该模块引入后，才能在模板中使用 {% static %} 标签。**

替换完成后你可以刷新页面并看看网页的源代码，看一看 {% static %} 模板标签在页面渲染后究竟被替换成了什么样的值。例如我们可以看到

<**link** rel="stylesheet" href="{% static 'blog/css/pace.css' %}">

这一部分最终在浏览器中显示的是：

<**link** rel="stylesheet" href="/static/blog/css/pace.css">

这正是 pace.css 文件所在的路径，其它的文件路径也被类似替换。可以看到 {% static %} 标签的作用实际就是把后面的字符串加了一个 /static/ 前缀，比如 {% static 'blog/css/pace.css' %} 最终渲染的值是 /static/blog/css/pace.css。而 /static/ 前缀是我们在 settings.py 文件中通过 STATIC\_URL = '/static/' 指定的。事实上，如果我们直接把引用路径写成 /static/blog/css/pace.css 也是可以的，那么为什么要使用 {% static %} 标签呢？想一下，目前 URL 的前缀是 /static/，如果哪一天因为某些原因，我们需要把 /static/ 改成 /resource/，如果你是直接写的引用路劲而没有使用 static 模板标签，那么你可能需要改 N 个地方。如果你使用了 static 模板标签，那么只要在 settings.py 处改一个地方就可以了，即把 STATIC\_URL = '/static/' 改成 STATIC\_URL = '/resource/'。

**有时候按 F5 刷新后页面还是很乱，这可能是因为浏览器缓存了之前的结果。按 Shift + F5（有些浏览器可能是 Ctrl + F5）强制刷新浏览器页面即可。**

注意这里有一个 CSS 文件的引入

<**link** rel="stylesheet" href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css">

我们没有使用模板标签，因为这里的引用的文件是一个外部文件，不是我们项目里 static\blog\css\ 目录下的文件，因此无需使用模板标签。

正确引入了静态文件后样式显示正常了。



## 修改模板

目前我们看到的只是模板中预先填充的一些数据，我们得让它显示从数据库中获取的文章数据。下面来稍微改造一下模板：

在模板 index.html 中你会找到一系列 article 标签：

templates/blog/index.html

...

<**article** class="post post-1">

...

</**article**>

<**article** class="post post-2">

...

</**article**>

<**article** class="post post-3">

...

</**article**>

...

这里面包裹的内容显示的就是文章数据了。我们前面在视图函数 index 里给模板传了一个 post\_list 变量，它里面包含着从数据库中取出的文章列表数据。就像 Python 一样，我们可以在模板中循环这个列表，把文章一篇篇循环出来，然后一篇篇显示文章的数据。要在模板中使用循环，需要使用到前面提到的模板标签，这次使用 {% for %} 模板标签。将 index.html 中多余的 article 标签删掉，只留下一个 article 标签，然后写上下列代码：

templates/blog/index.html

...

{% for post in post\_list %}

<**article** class="post post-{{ post.pk }}">

...

</**article**>

{% empty %}

<**div** class="no-post">暂时还没有发布的文章！</**div**>

{% endfor %}

...

可以看到语法和 Python 的 for 循环类似，只是被 {% %} 这样一个模板标签符号包裹着。{% empty %} 的作用是当 post\_list 为空，即数据库里没有文章时显示 {% empty %} 下面的内容，最后我们用 {% endfor %} 告诉 Django 循环在这里结束了。

你可能不太理解模板中的 post 和 post\_list 是什么。post\_list 是一个 QuerySet（类似于一个列表的数据结构），其中每一项都是之前定义在 blog\models.py 中的 Post 类的实例，且每个实例分别对应着数据库中每篇文章的记录。因此我们循环遍历 post\_list ，每一次遍历的结果都保存在 post 变量里。所以我们使用模板变量来显示 post 的属性值。例如这里的 {{ post.pk }}（pk 是 primary key 的缩写，即 post 对应于数据库中记录的 id 值，该属性尽管我们没有显示定义，但是 Django 会自动为我们添加）。

现在我们可以在循环体内通过 post 变量访问单篇文章的数据了。分析 article 标签里面的 HTML 内容，h1 显示的是文章的标题，

<**h1** class="entry-title">

<**a** href="single.html">Adaptive Vs. Responsive Layouts And Optimal Text Readability</**a**>

</**h1**>

我们把标题替换成 post 的 title 属性值。注意要把它包裹在模板变量里，因为它最终要被替换成实际的 title 值。

<**h1** class="entry-title">

<**a** href="single.html">{{ post.title }}</**a**>

</**h1**>

下面这 5 个 span 标签里分别显示了分类（category）、文章发布时间、文章作者、评论数、阅读量。

<**div** class="entry-meta">

<**span** class="post-category"><**a** href="#">Django 博客教程</**a**></**span**>

<**span** class="post-date"><**a** href="#"><**time** class="entry-date"

datetime="2012-11-09T23:15:57+00:00">2017年5月11日</**time**></**a**></**span**>

<**span** class="post-author"><**a** href="#">追梦人物</**a**></**span**>

<**span** class="comments-link"><**a** href="#">4 评论</**a**></**span**>

<**span** class="views-count"><**a** href="#">588 阅读</**a**></**span**>

</**div**>

再次替换掉一些数据，由于评论数和阅读量暂时没法替换，因此先留着，我们在之后实现了这些功能后再来修改它，目前只替换分类、文章发布时间、文章作者：

<**div** class="entry-meta">

<**span** class="post-category"><**a** href="#">{{ post.category.name }}</**a**></**span**>

<**span** class="post-date"><**a** href="#"><**time** class="entry-date"

datetime="{{ post.created\_time }}">{{ post.created\_time }}</**time**></**a**></**span**>

<**span** class="post-author"><**a** href="#">{{ post.author }}</**a**></**span**>

<**span** class="comments-link"><**a** href="#">4 评论</**a**></**span**>

<**span** class="views-count"><**a** href="#">588 阅读</**a**></**span**>

</**div**>

这里 p 标签里显示的是摘要

<**div** class="entry-content clearfix">

<**p**>免费、中文、零基础，完整的项目，基于最新版 Django 1.10 和 Python 3.5。带你从零开始一步步开发属于自己的博客网站，帮助你以最快的速度掌握 Django

开发的技巧...</**p**>

<**div** class="read-more cl-effect-14">

<**a** href="#" class="more-link">继续阅读 <**span** class="meta-nav">→</**span**></**a**>

</**div**>

</**div**>

替换成 post 的摘要：

<**div** class="entry-content clearfix">

<**p**>{{ post.excerpt }}</**p**>

<**div** class="read-more cl-effect-14">

<**a** href="#" class="more-link">继续阅读 <**span** class="meta-nav">→</**span**></**a**>

</**div**>

</**div**>

再次访问首页，它显示：暂时还没有发布的文章！好吧，做了这么多工作，但是数据库中其实还没有任何数据呀！接下来我们就实际写几篇文章保存到数据库里，看看显示的效果究竟如何。

# 7 - 在 Django Admin 后台发布文章

1 月，1 周前 字数 2036 阅读 1153 评论 4

在此之前我们完成了 Django 博客首页视图的编写，我们希望首页展示发布的博客文章列表，但是它却抱怨：暂时还没有发布的文章！如它所言，我们确实还没有发布任何文章，本节我们将使用 Django 自带的 Admin 后台来发布我们的博客文章。

## 创建 Admin 后台管理员账户

要想进入Django Admin 后台，首先需要创建一个超级管理员账户。我们在 [让 Django 完成翻译：迁移数据库](http://zmrenwu.com/post/6/) 中已经创建了一个后台账户，但如果你没有按照前面的步骤创建账户的话，可以运行 python manage.py createsuperuser 命令新建一个：

python manage.py createsuperuser

Username (leave blank to use 'zmrenwu@163.com'): admin

Email **address**: admin@example.com

**Warning**: Password input may be echoed.

**Password**: \*\*\*\*\*\*

**Warning**: Password input may be echoed.

Password (again): \*\*\*\*\*\*

Superuser created successfully.

**注意：在命令行输入密码时可能不会显示输入的字符，不要以为键盘坏了，照正常的方式输入密码即可。**

## 在 Admin 后台注册模型

要在后台注册我们自己创建的几个模型，这样 Django Admin 才能知道它们的存在，注册非常简单，只需要在 blog\admin.py 中加入下面的代码：

blog/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Post, Category, Tag

admin.site.register(Post)

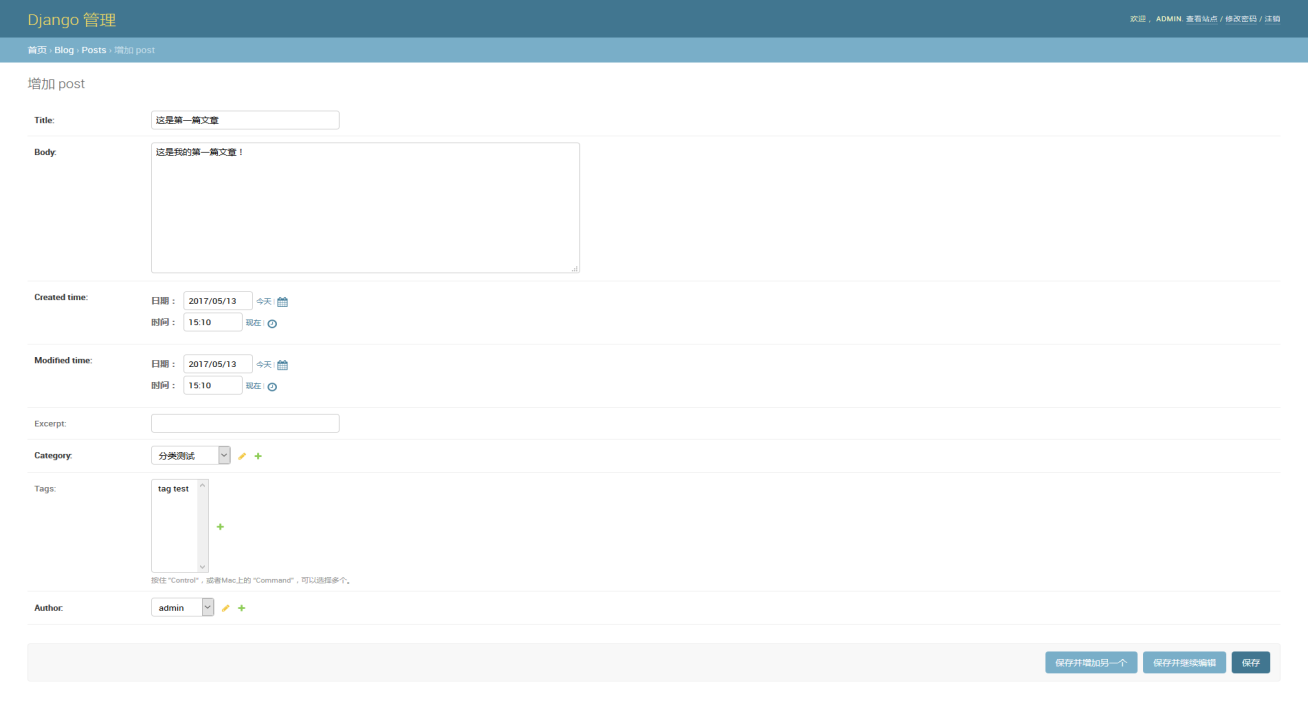
admin.site.register(Category)

admin.site.register(Tag)

**激活虚拟环境**，运行开发服务器，访问 http://127.0.0.1:8000/admin/ ，就进入了到了Django Admin 后台登录页面，输入刚才创建的管理员账户密码就可以登录到后台了。



可以看到我们刚才注册的三个模型了，点击 Posts 后面的**增加**按钮，将进入添加 Post 的页面，也就是新增博客文章。然后在相关的地方输入一些测试用的内容，增加完后点击保存，这样文章就添加完毕了，你也可以多添加几篇看看效果。注意每篇文章必须有一个分类，在添加文章时你可以选择已有分类。如果数据库中还没有分类，在选择分类时点击 Category 后面的 + 按钮新增一个分类即可。



**你可能想往文章内容中添加图片，但目前来说还做不到。在支持 Markdown 语法部分中将介绍如何在文章中插入图片的方法。**

访问 http://127.0.0.1:8000/ 首页，你就可以看到你添加的文章列表了，下面是我所在环境的效果图：



## 定制 Admin 后台

在 admin post 列表页面，我们只看到了文章的标题，但是我们希望它显示更加详细的信息，这需要我们来定制 Admin 了，在 admin.py 添加如下代码：

blog/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Post, Category, Tag

**class** **PostAdmin**(admin.ModelAdmin):

list\_display = ['title', 'created\_time', 'modified\_time', 'category', 'author']

*# 把新增的 PostAdmin 也注册进来*

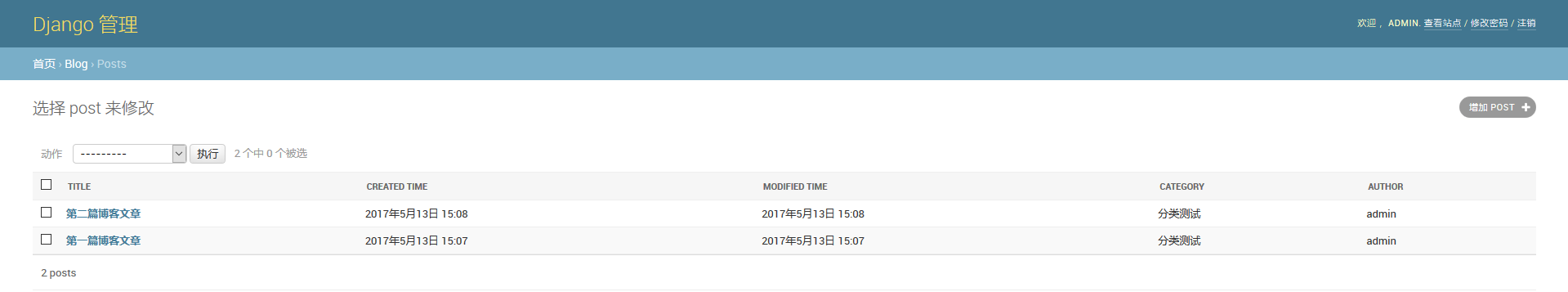
admin.site.register(Post, PostAdmin)

admin.site.register(Category)

admin.site.register(Tag)

由于如何定制 Admin 后台不是本教程的重点，这里只是为了简单地到达期望效果，因此不再对上述代码做过多解释。

刷新 Admin Post 列表页面，可以看到显示的效果好多了。



# 8 - 博客文章详情页

1 月，1 周前 字数 10102 阅读 1223 评论 8

首页展示的是所有文章的列表，当用户看到感兴趣的文章时，他点击文章的标题或者继续阅读的按钮，应该跳转到文章的详情页面来阅读文章的详细内容。现在让我们来开发博客的详情页面，有了前面的基础，开发流程都是一样的了：首先配置 URL，即把相关的 URL 和视图函数绑定在一起，然后实现视图函数，编写模板并让视图函数渲染模板。

## 设计文章详情页的 URL

回顾一下我们首页视图的 URL，在 blog\urls.py 文件里，我们写了：

blog/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

urlpatterns = [

url(r'^$', views.index, name='index'),

]

首页视图匹配的 URL 去掉域名后其实就是一个空的字符串。对文章详情视图而言，每篇文章对应着不同的 URL。比如我们可以把文章详情页面对应的视图设计成这个样子：当用户访问 <网站域名>/post/1/ 时，显示的是第一篇文章的内容，而当用户访问 <网站域名>/post/2/ 时，显示的是第二篇文章的内容，这里数字代表了第几篇文章，也就是数据库中 Post 记录的 id 值。下面依照这个规则来绑定 URL 和视图：

blog/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

app\_name = 'blog'

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^post/(?P<pk>[0-9]+)/$', views.detail, name='detail'),

]

Django 使用正则表达式来匹配用户访问的网址。这里 r'^post/(?P<pk>[0-9]+)/$' 整个正则表达式刚好匹配我们上面定义的 URL 规则。这条正则表达式的含义是，以 post/ 开头，后跟一个至少一位数的数字，并且以 / 符号结尾，如 post/1/、 post/255/ 等都是符合规则的，[0-9]+ 表示一位或者多位数。此外这里 (?P<pk>[0-9]+) 表示命名捕获组，其作用是从用户访问的 URL 里把括号内匹配的字符串捕获并作为关键字参数传给其对应的视图函数 detail。比如当用户访问 post/255/ 时（注意 Django 并不关心域名，而只关心去掉域名后的相对 URL），被括起来的部分 (?P<pk>[0-9]+) 匹配 255，那么这个 255 会在调用视图函数 detail 时被传递进去，实际上视图函数的调用就是这个样子：detail(request, pk=255)。我们这里必须从 URL 里捕获文章的 id，因为只有这样我们才能知道用户访问的究竟是哪篇文章。

**可能上述的正则表达式你有点难以理解，关于正则表达式的部分并非 Django 相关的内容，而是 Python 的内容。Django 只是在这里使用了 Python 处理正则表达式的 re 模块。因此如果想更好地理解 Python 中正则表达式的相关知识，请自行查看 Python 官方文档中 re 模块的文档。**

**此外我们通过 app\_name='blog' 告诉 Django 这个 urls.py 模块是属于 blog 应用的，这种技术叫做视图函数命名空间。我们看到 blog\urls.py 目前有两个视图函数，并且通过 name 属性给这些视图函数取了个别名，分别是 index、detail。但是一个复杂的 Django 项目可能不止这些视图函数，例如一些第三方应用中也可能有叫 index、detail 的视图函数，那么怎么把它们区分开来，防止冲突呢？方法就是通过 app\_name 来指定命名空间，命名空间具体如何使用将在下面介绍。如果你忘了在 blog\urls.py 中添加这一句，接下来你可能会得到一个 NoMatchReversed 异常。**

为了方便地生成上述的 URL，我们在 Post 类里定义一个 get\_absolute\_url 方法，注意 Post 本身是一个 Python 类，在类中我们是可以定义任何方法的。

blog/models.py

**from** **django.db** **import** models

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

**from** **django.urls** **import** reverse

**from** **django.utils.six** **import** python\_2\_unicode\_compatible

**@python\_2\_unicode\_compatible**

**class** **Post**(models.Model):

...

**def** \_\_str\_\_(self):

**return** self.title

*# 自定义 get\_absolute\_url 方法*

*# 记得从 django.urls 中导入 reverse 函数*

**def** get\_absolute\_url(self):

**return** reverse('blog:detail', kwargs={'pk': self.pk})

注意到 URL 配置中的 url(r'^post/(?P<pk>[0-9]+)/$', views.detail, name='detail') ，我们设定的 name='detail' 在这里派上了用场。看到这个 reverse 函数，它的第一个参数的值是 'blog:detail'，意思是 blog 应用下的 name=detail 的函数，由于我们在上面通过 app\_name = 'blog' 告诉了 Django 这个 URL 模块是属于 blog 应用的，因此 Django 能够顺利地找到 blog 应用下 name 为 detail 的视图函数，于是 reverse 函数会去解析这个视图函数对应的 URL，我们这里 detail 对应的规则就是 post/(?P<pk>[0-9]+)/ 这个正则表达式，而正则表达式部分会被后面传入的参数 pk 替换，所以，如果 Post 的 id（或者 pk，这里 pk 和 id 是等价的） 是 255 的话，那么 get\_absolute\_url 函数返回的就是 /post/255/ ，这样 Post 自己就生成了自己的 URL。

## 编写 detail 视图函数

接下来就是实现我们的 detail 视图函数了：

blog/views.py

**from** **django.shortcuts** **import** render, get\_object\_or\_404

**from** **.models** **import** Post

**def** index(request):

*# ...*

**def** detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

**return** render(request, 'blog/detail.html', context={'post': post})

视图函数很简单，它根据我们从 URL 捕获的文章 id（也就是 pk，这里 pk 和 id 是等价的）获取数据库中文章 id 为该值的记录，然后传递给模板。注意这里我们用到了从 django.shortcuts 模块导入的 get\_object\_or\_404 方法，其作用就是当传入的 pk 对应的 Post 在数据库存在时，就返回对应的 post，如果不存在，就给用户返回一个 404 错误，表明用户请求的文章不存在。

## 编写详情页模板

接下来就是书写模板文件，从下载的博客模板（如果你还没有下载，请 [点击这里](https://github.com/zmrenwu/django-blog-tutorial-templates) 下载）中把 single.html 拷贝到 templates\blog 目录下（和 index.html 在同一级目录），然后改名为 detail.html。此时你的目录结构应该像这个样子：

blogproject\

manage.py

blogproject\

\_\_init\_\_.py

settings.py

...

blog/

\_\_init\_\_.py

models.py

,,,

templates\

blog\

index.html

detail.html

在 index 页面博客文章列表的**标题**和**继续阅读按钮**写上超链接跳转的链接，即文章 post 对应的详情页的 URL，让用户点击后可以跳转到 detail 页面：

templates/blog/index.html

<**article** class="post post-1">

<**header** class="entry-header">

<**h1** class="entry-title">

<**a** href="{{ post.get\_absolute\_url }}">{{ post.title }}</**a**>

</**h1**>

...

</**header**>

<**div** class="entry-content clearfix">

...

<**div** class="read-more cl-effect-14">

<**a** href="{{ post.get\_absolute\_url }}" class="more-link">继续阅读 <**span** class="meta-nav">→</**span**></**a**>

</**div**>

</**div**>

</**article**>

{% empty %}

<**div** class="no-post">暂时还没有发布的文章！</**div**>

{% endfor %}

这里我们修改两个地方，第一个是文章标题处：

<**h1** class="entry-title">

<**a** href="{{ post.get\_absolute\_url }}">{{ post.title }}</**a**>

</**h1**>

我们把 a 标签的 href 属性的值改成了 {{ post.get\_absolute\_url }}。回顾一下模板变量的用法，由于 get\_absolute\_url 这个方法（我们定义在 Post 类中的）返回的是 post 对应的 URL，因此这里 {{ post.get\_absolute\_url }} 最终会被替换成该 post 自身的 URL。

同样，第二处修改的是继续阅读按钮的链接：

<**a** href="{{ post.get\_absolute\_url }}" class="more-link">继续阅读 <**span** class="meta-nav">→</**span**>

</**a**>

这样当我们点击首页文章的标题或者继续阅读按钮后就会跳转到该篇文章对应的详情页面了。然而如果你尝试跳转到详情页后，你会发现样式是乱的。这在 [真正的 Django 博客首页](http://zmrenwu.com/post/8/) 时讲过，由于我们是直接复制的模板，还没有正确地处理静态文件。我们可以按照介绍过的方法修改静态文件的引入路径，但很快你会发现在任何页面都是需要引入这些静态文件，如果每个页面都要修改会很麻烦，而且代码都是重复的。下面就介绍 Django 模板继承的方法来帮我们消除这些重复操作。

## 模板继承

我们看到 index.html 文件和 detail.html 文件除了 main 标签包裹的部分不同外，其它地方都是相同的，我们可以把相同的部分抽取出来，放到 base.html 里。首先在 templates\ 目录下新建一个 base.html 文件，这时候你的项目目录应该变成了这个样子：

blogproject\

manage.py

blogproject\

\_\_init\_\_.py

settings.py

...

blog\

\_\_init\_\_.py

models.py

,,,

templates\

base.html

blog\

index.html

detail.html

把 index.html 的内容全部拷贝到 base.html 文件里，然后删掉 main 标签包裹的内容，替换成如下的内容。

templates/base.html

...

<**main** class="col-md-8">

{% block main %}

{% endblock main %}

</**main**>

<**aside** class="col-md-4">

{% block toc %}

{% endblock toc %}

...

</**aside**>

...

这里 block 也是一个模板标签，其作用是占位。比如这里的 {% block main %}{% endblock main %} 是一个占位框，main 是我们给这个 block 取的名字。下面我们会看到 block 标签的作用。同时我们也在 aside 标签下加了一个 {% block toc %}{% endblock toc %} 占位框，因为 detail.html 中 aside 标签下会多一个目录栏。当 {% block toc %}{% endblock toc %} 中没有任何内容时，{% block toc %}{% endblock toc %} 在模板中不会显示。但当其中有内容是，模板就会显示 block 中的内容。

在 index.html 里，我们在文件**最顶部**使用 {% extends 'base.html' %} 继承 base.html，这样就把 base.html 里的代码继承了过来，另外在 {% block main %}{% endblock main %} 包裹的地方填上 index 页面应该显示的内容：

templates/blog/index.html

{% extends 'base.html' %}

{% block main %}

{% for post in post\_list %}

<**article** class="post post-1">

...

</**article**>

{% empty %}

<**div** class="no-post">暂时还没有发布的文章！</**div**>

{% endfor %}

*<!-- 简单分页效果*

*<div class="pagination-simple">*

*<a href="#">上一页</a>*

*<span class="current">第 6 页 / 共 11 页</span>*

*<a href="#">下一页</a>*

*</div>*

*-->*

<**div** class="pagination">

...

</**div**>

{% endblock main %}

这样 base.html 里的代码加上 {% block main %}{% endblock main %} 里的代码就和最开始 index.html 里的代码一样了。这就是模板继承的作用，公共部分的代码放在 base.html 里，而其它页面不同的部分通过替换 {% block main %}{% endblock main %} 占位标签里的内容即可。

如果你对这种模板继承还是有点糊涂，可以把这种继承和 Python 中类的继承类比。base.html 就是父类，index.html 就是子类。index.html 继承了 base.html 中的全部内容，同时它自身还有一些内容，这些内容就通过 “覆写” {% block main %}{% endblock main %}（把 block 看做是父类的属性）的内容添加即可。

detail 页面处理起来就简单了，同样继承 base.html ，在 {% block main %}{% endblock main %} 里填充 detail.html 页面应该显示的内容，以及在 {% block toc %}{% endblock toc %} 中填写 base.html 中没有的目录部分的内容。不过目前的目录只是占位数据，我们在以后会实现如何从文章中自动摘取目录。

templates/blog/detail.html

{% extends 'base.html' %}

{% block main %}

<**article** class="post post-1">

...

</**article**>

<**section** class="comment-area">

...

</**section**>

{% endblock main %}

{% block toc %}

<**div** class="widget widget-content">

<**h3** class="widget-title">文章目录</**h3**>

<**ul**>

<**li**>

<**a** href="#">教程特点</**a**>

</**li**>

<**li**>

<**a** href="#">谁适合这个教程</**a**>

</**li**>

<**li**>

<**a** href="#">在线预览</**a**>

</**li**>

<**li**>

<**a** href="#">资源列表</**a**>

</**li**>

<**li**>

<**a** href="#">获取帮助</**a**>

</**li**>

</**ul**>

</**div**>

{% endblock toc %}

修改 article 标签下的一些内容，让其显示文章的实际数据：

<**article** class="post post-{{ post.pk }}">

<**header** class="entry-header">

<**h1** class="entry-title">{{ post.title }}</**h1**>

<**div** class="entry-meta">

<**span** class="post-category"><**a** href="#">{{ post.category.name }}</**a**></**span**>

<**span** class="post-date"><**a** href="#"><**time** class="entry-date"

datetime="{{ post.created\_time }}">{{ post.created\_time }}</**time**></**a**></**span**>

<**span** class="post-author"><**a** href="#">{{ post.author }}</**a**></**span**>

<**span** class="comments-link"><**a** href="#">4 评论</**a**></**span**>

<**span** class="views-count"><**a** href="#">588 阅读</**a**></**span**>

</**div**>

</**header**>

<**div** class="entry-content clearfix">

{{ post.body }}

</**div**>

</**article**>

再次从首页点击一篇文章的标题或者继续阅读按钮跳转到详情页面，可以看到预期效果了！



# 9 - 支持 Markdown 语法和代码高亮

1 月，1 周前 字数 4613 阅读 1009 评论 21

为了让博客文章具有良好的排版，显示更加丰富的格式，我们使用 Markdown 语法来书写我们的博文。Markdown 是一种 HTML 文本标记语言，只要遵循它约定的语法格式，Markdown 的渲染器就能够把我们写的文章转换为标准的 HTML 文档，从而让我们的文章呈现更加丰富的格式，例如标题、列表、代码块等等 HTML 元素。由于 Markdown 语法简单直观，不用超过 5 分钟就可以掌握常用的标记语法，因此大家青睐使用 Markdown 书写 HTML 文档。下面让我们的博客也支持使用 Markdown 书写。

## 安装 Python Markdown

将 Markdown 格式的文本渲染成标准的 HTML 文档是一个复杂的工作，好在已有好心人帮我们完成了这些工作，我们直接使用即可。首先安装 Markdown，这是一个 Python 第三方库，**激活虚拟环境**，然后使用命令 pip install markdown 安装即可。

## 在 detail 视图中渲染 Markdown

将 Markdown 格式的文本渲染成 HTML 文本非常简单，只需调用这个库的 markdown 方法即可。我们书写的博客文章内容存在 Post 的 body 属性里，回到我们的详情页视图函数，对 post 的 body 的值做一下渲染，把 Markdown 文本转为 HTML 文本再传递给模板：

blog/views.py

**import** **markdown**

**from** **django.shortcuts** **import** render, get\_object\_or\_404

**from** **.models** **import** Post

**def** detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

*# 记得在顶部引入 markdown 模块*

post.body = markdown.markdown(post.body,

extensions=[

'markdown.extensions.extra',

'markdown.extensions.codehilite',

'markdown.extensions.toc',

])

**return** render(request, 'blog/detail.html', context={'post': post})

这样我们在模板中展示 {{ post.body }} 的时候，就不再是原始的 Markdown 文本了，而是渲染过后的 HTML 文本。注意这里我们给 markdown 渲染函数传递了额外的参数 extensions，它是对 Markdown 语法的拓展，这里我们使用了三个拓展，分别是 extra、codehilite、toc。extra 本身包含很多拓展，而 codehilite 是语法高亮拓展，这为我们后面的实现代码高亮功能提供基础，而 toc 则允许我们自动生成目录（在以后会介绍）。

来测试一下效果，进入后台，这次我们发布一篇用 Markdown 语法写的测试文章看看，你可以使用以下的 Markdown 测试代码进行测试，也可以自己书写你喜欢的 Markdown 文本。假设你是 Markdown 新手参考一下这些教程，一定学一下，保证你可以在 5 分钟内掌握常用的语法格式，而以后对你写作受用无穷。可谓充电五分钟，通话 2 小时。以下是我学习中的一些参考资料：

* [Markdown——入门指南](http://www.jianshu.com/p/1e402922ee32/)
* [Markdown 语法说明 (简体中文版)](http://www.appinn.com/markdown/)

# 一级标题

## 二级标题

### 三级标题

- 列表项1

- 列表项2

- 列表项3

> 这是一段引用

​```python

def detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

post.body = markdown.markdown(post.body,

extensions=[

'markdown.extensions.extra',

'markdown.extensions.codehilite',

'markdown.extensions.toc',

])

return render(request, 'blog/detail.html', context={'post': post})

​```

**如果你发现无法显示代码块，即代码无法换行，请检查代码块的语法是否书写有误。代码块的语法如上边的测试文本中最后一段所示。**

你可能想在文章中插入图片，目前能做的且推荐做的是使用外链引入图片。比如将图片上传到七牛云这样的云存储服务器，然后通过 Markdown 的图片语法将图片引入。Markdown 引入图片的语法为：![图片说明](图片链接)。

## safe 标签

我们在发布的文章详情页没有看到预期的效果，而是类似于一堆乱码一样的 HTML 标签，这些标签本应该在浏览器显示它本身的格式，但是 Django 出于安全方面的考虑，任何的 HTML 代码在 Django 的模板中都会被转义（即显示原始的 HTML 代码，而不是经浏览器渲染后的格式）。为了解除转义，只需在模板标签使用 safe 过滤器即可，告诉 Django，这段文本是安全的，你什么也不用做。在模板中找到展示博客文章主体的 {{ post.body }} 部分，为其加上 safe 过滤器，{{ post.body|safe }}，大功告成，这下看到预期效果了。

safe 是 Django 模板系统中的过滤器（Filter），可以简单地把它看成是一种函数，其作用是作用于模板变量，将模板变量的值变为经过滤器处理过后的值。例如这里 {{ post.body|safe }}，本来 {{ post.body }} 经模板系统渲染后应该显示 body 本身的值，但是在后面加上 safe 过滤器后，渲染的值不再是body 本身的值，而是由 safe 函数处理后返回的值。过滤器的用法是在模板变量后加一个 | 管道符号，再加上过滤器的名称。可以连续使用多个过滤器，例如 {{ var|filter1|filter2 }}。



## 代码高亮

程序员写博客免不了要插入一些代码，Markdown 的语法使我们容易地书写代码块，但是目前来说，显示的代码块里的代码没有任何颜色，很不美观，也难以阅读，要是能够像我们的编辑器里一样让代码高亮就好了。虽然我们在渲染时使用了 codehilite 拓展，但这只是实现代码高亮的第一步，还需要简单的几步才能达到我们的最终目的。

### 安装 Pygments

首先我们需要安装 Pygments，**激活虚拟环境**，运行： pip install Pygments 安装即可。

搞定了，虽然我们除了安装了一下 Pygments 什么也没做，但 Markdown 使用 Pygments 在后台为我们做了很多事。如果你打开博客详情页，找到一段代码段，在浏览器查看这段代码段的 HTML 源代码，可以发现 Pygments 的工作原理是把代码切分成一个个单词，然后为这些单词添加 css 样式，不同的词应用不同的样式，这样就实现了代码颜色的区分，即高亮了语法。为此，还差最后一步，引入一个样式文件来给这些被添加了样式的单词定义颜色。

### 引入样式文件

在项目的 blog\static\blog\css\highlights\ 目录下应该能看到很多 .css 样式文件，这些文件是用来提供代码高亮样式的。选择一个你喜欢的样式文件，在 base.html 引入即可（别忘了使用 static 模板标签）。比如我比较喜欢 github.css 的样式，那么引入这个文件：

templates/base.html

...

<**link** rel="stylesheet" href="{% static 'blog/css/pace.css' %}">

<**link** rel="stylesheet" href="{% static 'blog/css/custom.css' %}">

...

+ <**link** rel="stylesheet" href="{% static 'blog/css/highlights/github.css' %}">

这里 + 号表示添加这行代码。好了，看看效果，大功告成，终于可以愉快地贴代码了。



**注意**：如果你按照教程中的方法做完后发现代码依然没有高亮，请依次检查以下步骤：

1. 确保在渲染文本时添加了 markdown.extensions.codehilite 拓展，详情见上文。
2. 确保安装了 Pygments。
3. 确保代码块的 Markdown 语法正确，**特别是指明该代码块的语言类型**，具体请参见上文中 Markdown 的语法示例。
4. 在浏览器端代码块的源代码，看代码是否被 pre 标签包裹，并且代码的每一个单词都被 span 标签包裹，且有一个 class 属性值。如果没有，极有可能是前三步中某个地方出了问题。
5. 确保用于代码高亮的样式文件被正确地引入，具体请参见上文中引入样式文件的讲解。
6. 有些样式文件可能对代码高亮没有作用，首先尝试用 github.css 样式文件做测试。

**10 - 页面侧边栏：使用自定义模板标签**

1 月，1 周前 字数 5812 阅读 792 评论 1

我们的博客侧边栏有四项内容：最新文章、归档、分类和标签云。这些内容相对比较固定，且在各个页面都会显示，如果像文章列表或者文章详情一样，从视图函数中获取然后传递给模板，则每个页面对应的视图函数里都要写一段获取这些内容的代码，这会导致很多重复代码。更好的解决方案是直接在模板中获取，为此，我们使用 Django 的一个新技术：自定义模板标签来完成任务。

**使用模板标签的解决思路**

我们前面已经接触过一些 Django 内置的模板标签，比如比较简单的 {% static %} 模板标签，这个标签帮助我们在模板中引入静态文件。还有比较复杂的如 {% for %} {% endfor%} 标签。这里 我们希望自己定义一个模板标签，例如名为 get\_recent\_posts 的模板标签，它可以这样工作：我们只要在模板中写入 {% get\_recent\_posts as recent\_post\_list %}，那么模板中就会有一个从数据库获取的最新文章列表，并通过 as 语句保存到 recent\_post\_list 模板变量里。这样我们就可以通过 {% for %} {% endfor%} 模板标签来循环这个变量，显示最新文章列表了，这和我们在编写博客首页面视图函数是类似的。首页视图函数中从数据库获取文章列表并保存到 post\_list 变量，然后把这个 post\_list 变量传给模板，模板使用 for 模板标签循环这个文章列表变量，从而展示一篇篇文章。这里唯一的不同是我们从数据库获取文章列表的操作不是在视图函数中进行，而是在模板中通过自定义的 {% get\_recent\_posts %} 模板标签进行。

以上就是解决思路，但模板标签不是我们随意写的，必须遵循 Django 的规范我们才能在 Django 的模板系统中使用自定义的模板标签，下面我们就依照这些规范来实现我们的需求。

**模板标签目录结构**

首先在我们的 **blog 应用**下创建一个 templatetags 文件夹。然后在这个文件夹下创建一个 \_\_init\_\_.py 文件，使这个文件夹成为一个 Python 包，之后在 templatetags\ 目录下创建一个 blog\_tags.py 文件，这个文件存放自定义的模板标签代码。

此时你的目录结构应该是这样的：

blog\

\_\_init\_\_.py

admin.py

apps.py

migrations\

\_\_init\_\_.py

models.py

static\

templatetags\

\_\_init\_\_.py

blog\_tags.py

tests.py

views.py

**编写模板标签代码**

接下来就是编写各个模板标签的代码了，自定义模板标签代码写在 blog\_tags.py 文件中。其实模板标签本质上就是一个 Python 函数，因此按照 Python 函数的思路来编写模板标签的代码就可以了，并没有任何新奇的东西或者需要新学习的知识在里面。

**最新文章模板标签**

打开 blog\_tags.py 文件，开始写我们的最新文章模板标签。

blog/templatetags/blog\_tags.py

**from** **..models** **import** Post

**def** get\_recent\_posts(num=5):

**return** Post.objects.all().order\_by('-created\_time')[:num]

这个函数的功能是获取数据库中前 num 篇文章，这里 num 默认为 5。函数就这么简单，但目前它还只是一个纯 Python 函数，Django 在模板中还不知道该如何使用它。为了能够通过 {% get\_recent\_posts %} 的语法在模板中调用这个函数，必须按照 Django 的规定注册这个函数为模板标签，方法如下：

blog/templatetags/blog\_tags.py

**from** **django** **import** template

**from** **..models** **import** Post

register = template.Library()

**@register.simple\_tag**

**def** get\_recent\_posts(num=5):

**return** Post.objects.all().order\_by('-created\_time')[:num]

这里我们首先导入 template 这个模块，然后实例化了一个 template.Library类，并将函数 get\_recent\_posts 装饰为 register.simple\_tag。这样就可以在模板中使用语法 {% get\_recent\_posts %} 调用这个函数了。

**注意 Django 1.9 后才支持 simple\_tag 模板标签，如果你使用的 Django 版本小于 1.9，你将得到一个错误。Django 1.9 以前的版本如何自定义模板标签这里不再赘述。**

**归档模板标签**

和最新文章模板标签一样，先写好函数，然后将函数注册为模板标签即可。

blog/templatetags/blog\_tags.py

**@register.simple\_tag**

**def** archives():

**return** Post.objects.dates('created\_time', 'month', order='DESC')

这里 dates 方法会返回一个列表，列表中的元素为每一篇文章（Post）的创建时间，且是 Python 的 date 对象，精确到月份，降序排列。接受的三个参数值表明了这些含义，一个是 created\_time ，即 Post 的创建时间，month 是精度，order='DESC' 表明降序排列（即离当前越近的时间越排在前面）。例如我们写了 3 篇文章，分别发布于 2017 年 2 月 21 日、2017 年 3 月 25 日、2017 年 3 月 28 日，那么 dates 函数将返回 2017 年 3 月 和 2017 年 2 月这样一个时间列表，且降序排列，从而帮助我们实现按月归档的目的。

**分类模板标签**

过程还是一样，先写好函数，然后将函数注册为模板标签。注意分类模板标签函数中使用到了 Category 类，其定义在 blog.models.py 文件中，使用前记得先导入它，否则会报错。

blog/templatetags/blog\_tags.py

**from** **..models** **import** Post, Category

**@register.simple\_tag**

**def** get\_categories():

*# 别忘了在顶部引入 Category 类*

**return** Category.objects.all()

尽管侧边栏有 4 项内容（还有一个标签云），但是这里我们只实现最新文章、归档和分类数据的显示，还有一个标签云没有实现。因为标签云的实现稍有一点不同，所以将在接下来的教程中专门介绍。这里你也可以尝试着自己解决，如果遇到问题，可以通过官方文档或者搜索引擎求助。独立思考并寻求解决方案以及善用搜索引擎是一个开发者必须培养的能力，只有这样你才能成为一个独立的开发者，独立地解决别人可能从来没有遇到过的问题。

**使用自定义的模板标签**

打开 base.html，为了使用模板标签，我们首先需要在模板中导入存放这些模板标签的模块，这里是 blog\_tags.py 模块。当时我们为了使用 static 模板标签时曾经导入过 {% load staticfiles %}，这次在 {% load staticfiles %} 下再导入 blog\_tags：

templates/base.html

{% load staticfiles %}

{% load blog\_tags %}

<!DOCTYPE html>

<**html**>

...

</**html**>

然后找到最新文章列表处，把里面的列表修改一下：

templates/base.html

<**div** class="widget widget-recent-posts">

<**h3** class="widget-title">最新文章</**h3**>

{% get\_recent\_posts as recent\_post\_list %}

<**ul**>

{% for post in recent\_post\_list %}

<**li**>

<**a** href="{{ post.get\_absolute\_url }}">{{ post.title }}</**a**>

</**li**>

{% empty %}

暂无文章！

{% endfor %}

</**ul**>

</**div**>

这里我们通过使用 get\_recent\_posts 模板标签获取到最新文章列表，然后我们通过 as 语法（Django 模板系统的语法）将获取的文章列表保存进了 recent\_post\_list 模板变量中，之后就可以通过 for 循环来循环显示文章列表数据了，这和我们在写首页视图时是一样的。

然后是归档部分：

templates/base.html

<**div** class="widget widget-archives">

<**h3** class="widget-title">归档</**h3**>

{% archives as date\_list %}

<**ul**>

{% for date in date\_list %}

<**li**>

<**a** href="#">{{ date.year }} 年 {{ date.month }} 月</**a**>

</**li**>

{% empty %}

暂无归档！

{% endfor %}

</**ul**>

</**div**>

同样，这里我们调用 archives 模板标签自动获取一个已发表文章的日期列表，精确到月份，降序排列，然后通过 as 语法将其保存在 date\_list 模板变量里。由于日期列表中的元素为 Python 的 date 对象，因此可以通过其 year 和 month属性分别获取年和月的信息，<a href="#">{{ date.year }} 年 {{ date.month }} 月</a> 反应了这个事实。

分类部分也一样：

<**div** class="widget widget-category">

<**h3** class="widget-title">分类</**h3**>

{% get\_categories as category\_list %}

<**ul**>

{% for category in category\_list %}

<**li**>

<**a** href="#">{{ category.name }} <**span** class="post-count">(13)</**span**></**a**>

</**li**>

{% empty %}

暂无分类！

{% endfor %}

</**ul**>

</**div**>

<span class="post-count">(13)</span> 显示的是该分类下的文章数目，这个特性会在接下来的教程中讲解如何实现，目前暂时用占位数据代替吧。

现在运行开发服务器，可以看到侧边栏显示的数据已经不再是之前的占位数据，而是我们保存在数据库中的数据了。

**注意**：如果你按照教程的步骤做完后发现报错，请按以下顺序检查。

1. 检查目录结构是否正确。确保 templatetags\ 位于 blog\ 目录下，且目录名必须为 templatetags。具体请对照上文给出的目录结构。
2. 确保 templatetags\ 目录下有 \_\_init\_\_.py 文件。
3. 确保使用的 Django 版本不小于 1.9。
4. 确保通过 register = template.Library() 和 @register.simple\_tag 装饰器将函数装饰为一个模板标签。
5. 确保在使用模板标签以前导入了 blog\_tags，即 {% load blog\_tags %}。**注意要在使用任何 blog\_tags 下的模板标签以前导入它。**
6. 确保模板标签的语法使用正确，即 {% load blog\_tags %}，注意 { 和 % 以及 % 和 } 之间没有任何空格。

# 11 - 分类与归档

1 月，1 周前 字数 4462 阅读 688 评论 0

侧边栏已经正确地显示了最新文章列表、归档、分类等信息。现在来完善归档和分类功能，当用户点击归档下的某个日期或者分类下的某个分类时，跳转到文章列表页面，显示该日期或者分类下的全部文章。

## 归档页面

要显示某个归档日期下的文章列表，思路和显示主页文章列表是一样的，回顾一下主页视图的代码：

blog/views.py

**def** index(request):

post\_list = Post.objects.all().order\_by('-created\_time')

**return** render(request, 'blog/index.html', context={'post\_list': post\_list})

主页视图函数中我们通过 Post.objects.all() 获取全部文章，而在我们的归档和分类视图中，我们不再使用 all 方法获取全部文章，而是使用 filter 来根据条件过滤。先来看归档视图：

blog/views.py

**def** archives(request, year, month):

post\_list = Post.objects.filter(created\_time\_\_year=year,

created\_time\_\_month=month

).order\_by('-created\_time')

**return** render(request, 'blog/index.html', context={'post\_list': post\_list})

这里我们使用了模型管理器（objects）的 filter 函数来过滤文章。由于是按照日期归档，因此这里根据文章发表的年和月来过滤。具体来说，就是根据 created\_time 的 year 和 month 属性过滤，筛选出文章发表在对应的 year 年和 month 月的文章。注意这里 created\_time 是 Python 的 date 对象，其有一个 year 和 month 属性，我们在 [页面侧边栏：使用自定义模板标签](http://zmrenwu.com/post/12/) 使用过这个属性。Python 中类实例调用属性的方法通常是 created\_time.year，但是由于这里作为函数的参数列表，所以 Django 要求我们把点替换成了两个下划线，即 created\_time\_\_year。同时和 index 视图中一样，我们对返回的文章列表进行了排序。此外由于归档的下的文章列表的显示和首页是一样的，因此我们直接渲染了index.html 模板。

写好视图函数后就是配置好 URL：

blog/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

app\_name = 'blog'

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^post/(?P<pk>[0-9]+)/$', views.detail, name='detail'),

+ url(r'^archives/(?P<year>[0-9]{4})/(?P<month>[0-9]{1,2})/$', views.archives, name='archives'),

]

这个归档视图对应的 URL 的正则表达式和 detail 视图函数对应的 URL 是类似的，这在之前我们讲过。两个括号括起来的地方是两个命名组参数，Django 会从用户访问的 URL 中自动提取这两个参数的值，然后传递给其对应的视图函数。例如如果用户想查看 2017 年 3 月下的全部文章，他访问 /archives/2017/3/，那么 archives 视图函数的实际调用为：archives(request, year=2017, month=3)。

在模板找到归档列表部分的代码，修改超链接的 href 属性，让用户点击超链接后跳转到文章归档页面：

templates/base.html

{% for date in date\_list %}

<**li**>

<**a** href="{% url 'blog:archives' date.year date.month %}">

{{ date.year }} 年 {{ date.month }} 月

</**a**>

</**li**>

{% endfor %}

这里 {% url %} 这个模板标签的作用是解析视图函数 blog:archives 对应的 URL 模式，并把 URL 模式中的年和月替换成 date.year，date.month 的值。例如 blog:archives 表示 blog 应用下的 archives 函数，这个函数对应的 URL 模式为 ^archives/(?P<year>[0-9]{4})/(?P<month>[0-9]{1,2})/$，假设 date.year=2017，date.month=5，那么 {% url 'blog:archives' date.year date.month %} 模板标签返回的值为/archives/2017/5/。

为什么要使用 {% url %} 模板标签呢？事实上，我们把超链接的 href 属性设置为 /archives/{{ date.year }}/{{ date.month }}/ 同样可以达到目的，但是这种写法是硬编码的。虽然现在 blog:archives 视图函数对应的 URL 模式是这种形式，但是如果哪天这个模式改变了呢？如果使用了硬编码的写法，那你需要把每一处 /archives/{{ date.year }}/{{ date.month }}/ 修改为新的模式。但如果使用了 {% url %} 模板标签，则不用做任何修改。

测试一下，点击侧边栏归档的日期，跳转到归档页面，发现报了个错误，提示没有安装 pytz。**激活虚拟环境**，使用 pip install pytz 安装即可。

重启一下开发服务器，再次测试，发现可以显示归档下的文章列表了。

## 分类页面

同样的写好分类页面的视图函数：

blog/views.py

**import** **markdown**

**from** **django.shortcuts** **import** render, get\_object\_or\_404

*# 引入 Category 类*

**from** **.models** **import** Post, Category

**def** category(request, pk):

*# 记得在开始部分导入 Category 类*

cate = get\_object\_or\_404(Category, pk=pk)

post\_list = Post.objects.filter(category=cate).order\_by('-created\_time')

**return** render(request, 'blog/index.html', context={'post\_list': post\_list})

这里我们首先根据传入的 pk 值（也就是被访问的分类的 id 值）从数据库中获取到这个分类。get\_object\_or\_404 函数和 detail 视图中一样，其作用是如果用户访问的分类不存在，则返回一个 404 错误页面以提示用户访问的资源不存在。然后我们通过 filter 函数过滤出了该分类下的全部文章。同样也和首页视图中一样对返回的文章列表进行了排序。

URL 配置如下：

blog/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

app\_name = 'blog'

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^post/(?P<pk>[0-9]+)/$', views.detail, name='detail'),

url(r'^archives/(?P<year>[0-9]{4})/(?P<month>[0-9]{1,2})/$', views.archives, name='archives'),

+ url(r'^category/(?P<pk>[0-9]+)/$', views.category, name='category'),

]

这个分类页面对应的 URL 模式和文章详情页面对应的 URL 模式十分类似，你可以自己分析分析它是如何工作的，在此就不赘述了。

修改相应模板：

templates/base.html

{% for category in category\_list %}

<**li**>

<**a** href="{% url 'blog:category' category.pk %}">{{ category.name }}</**a**>

</**li**>

{% endfor %}

同样，{% url %} 模板标签的用法和写归档页面时的用法是一样的。现在尝试点击相应的链接，就可以跳转到归档或者分类页面了。

# 12 - 评论

1 月，1 周前 字数 11610 阅读 930 评论 8

## 创建评论应用

相对来说，评论其实是另外一个比较独立的功能。Django 提倡，如果功能相对比较独立的话，最好是创建一个应用，把相应的功能代码写到这个应用里。我们的第一个应用叫 blog，它里面放了展示博客文章列表和细节等相关功能的代码。而这里我们再创建一个应用，名为 comments，这里面将存放和评论功能相关的代码。首先**激活虚拟环境**，然后输入如下命令创建一个新的应用：

python manage.py startapp comments

我们可以看到生成的 comments 应用目录结构和 blog 应用的目录是类似的。关于创建应用以及 Django 的目录结构在 [建立 Django 博客应用](http://zmrenwu.com/post/4/) 中已经有过介绍。创建新的应用后一定要记得在 settings.py 里注册这个应用，Django 才知道这是一个应用。

blogproject/settings.py

...

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog',

'comments', *# 注册新创建的 comments 应用*

]

...

## 设计评论的数据库模型

用户评论的数据必须被存储到数据库里，以便其他用户访问时 Django 能从数据库取回这些数据然后展示给访问的用户，因此我们需要为评论设计数据库模型，这和设计文章、分类、标签的数据库模型是一样的，如果你忘了怎么做，再回顾一下 [创建 Django 博客的数据库模型](http://zmrenwu.com/post/5/) 中的做法。我们的评论模型设计如下（评论模型的代码写在 comment\models.py 里）：

comments/models.py

**from** **django.db** **import** models

**from** **django.utils.six** **import** python\_2\_unicode\_compatible

*# python\_2\_unicode\_compatible 装饰器用于兼容 Python2*

**@python\_2\_unicode\_compatible**

**class** **Comment**(models.Model):

name = models.CharField(max\_length=100)

email = models.EmailField(max\_length=255)

url = models.URLField(blank=True)

text = models.TextField()

created\_time = models.DateTimeField(auto\_now\_add=True)

post = models.ForeignKey('blog.Post')

**def** \_\_str\_\_(self):

**return** self.text[:20]

这里我们会保存评论用户的 name（名字）、email（邮箱）、url（个人网站），用户发表的内容将存放在 text 字段里，created\_time 记录评论时间。最后，这个评论是关联到某篇文章（Post）的，由于一个评论只能属于一篇文章，一篇文章可以有多个评论，是一对多的关系，因此这里我们使用了 ForeignKey。关于 ForeKey 我们前面已有介绍，这里不再赘述。

同时注意我们为 DateTimeField 传递了一个 auto\_now\_add=True 的参数值。auto\_now\_add 的作用是，当评论数据保存到数据库时，自动把 created\_time 的值指定为当前时间。created\_time 记录用户发表评论的时间，我们肯定不希望用户在发表评论时还得自己手动填写评论发表时间，这个时间应该自动生成。

创建了数据库模型就要迁移数据库，迁移数据库的命令也在前面讲过。在**虚拟环境**下分别运行下面两条命令：

python manage.py makemigrations

python manage.py migrate

## 评论表单设计

这一节我们将学习一个全新的 Django 知识：表单。那么什么是表单呢？基本的 HTML 知识告诉我们，在 HTML 文档中这样的代码表示一个表单：

<**form** action="" method="post">

<**input** type="text" name="username" />

<**input** type="password" name="password" />

<**input** type="submit" value="login" />

</**form**>

为什么需要表单呢？表单是用来收集并向服务器提交用户输入的数据的。考虑用户在我们博客网站上发表评论的过程。当用户想要发表评论时，他找到我们给他展示的一个评论表单（我们已经看到在文章详情页的底部就有一个评论表单，你将看到表单呈现给我们的样子），然后根据表单的要求填写相应的数据。之后用户点击评论按钮，这些数据就会发送给某个 URL。我们知道每一个 URL 对应着一个 Django 的视图函数，于是 Django 调用这个视图函数，我们在视图函数中写上处理用户通过表单提交上来的数据的代码，比如验证数据的合法性并且保存数据到数据库中，那么用户的评论就被 Django 后台处理了。如果通过表单提交的数据存在错误，那么我们把错误信息返回给用户，并在前端重新渲染，并要求用户根据错误信息修正表单中不符合格式的数据，再重新提交。

Django 的表单功能就是帮我们完成上述所说的表单处理逻辑，表单对 Django 来说是一个内容丰富的话题，很难通过教程中的这么一个例子涵盖其全部用法。因此我们强烈建议你在完成本教程后接下来的学习中仔细阅读 Django 官方文档关于 [表单](https://docs.djangoproject.com/en/1.10/topics/forms/) 的介绍，因为表单在 Web 开发中会经常遇到。

下面开始编写评论表单代码。在 comments\ 目录下（和 models.py 同级）新建一个 forms.py 文件，用来存放表单代码，我们的表单代码如下：

comments/forms.py

**from** **django** **import** forms

**from** **.models** **import** Comment

**class** **CommentForm**(forms.ModelForm):

**class** **Meta**:

model = Comment

fields = ['name', 'email', 'url', 'text']

要使用 Django 的表单功能，我们首先导入 forms 模块。Django 的表单类必须继承自 forms.Form 类或者 forms.ModelForm 类。如果表单对应有一个数据库模型（例如这里的评论表单对应着评论模型），那么使用 ModelForm 类会简单很多，这是 Django 为我们提供的方便。之后我们在表单的内部类 Meta 里指定一些和表单相关的东西。model = Comment 表明这个表单对应的数据库模型是 Comment类。fields = ['name', 'email', 'url', 'text'] 指定了表单需要显示的字段，这里我们指定了 name、email、url、text 需要显示。

## 关于表单进一步的解释

Django 为什么要给我们提供一个表单类呢？为了便于理解，我们可以把表单和前面讲过的 Django ORM 系统做类比。回想一下，我们使用数据库保存我们创建的博客文章，但是我们从头到尾没有写过任何和数据库有关的代码（要知道数据库自身也有一门数据库语言），这是因为 Django 的 ORM 系统内部帮我们做了一些事情。我们遵循 Django 的规范写的一些 Python 代码，例如创建 Post、Category 类，然后通过运行数据库迁移命令将这些代码反应到数据库。

Django 的表单和这个思想类似，正常的前端表单代码应该是和本文开头所提及的那样，但是我们目前并没有写这些代码，而是写了一个 CommentForm 这个 Python 类。通过调用这个类的一些方法和属性，Django 将自动为我们创建常规的表单代码，接下来的教程我们就会看到具体是怎么做的。

## 评论视图函数

当用户提交表单中的数据后，Django 需要调用相应的视图函数来处理这些数据，下面开始写我们视图函数处理逻辑：

comments/views.py

**from** **django.shortcuts** **import** render, get\_object\_or\_404, redirect

**from** **blog.models** **import** Post

**from** **.models** **import** Comment

**from** **.forms** **import** CommentForm

**def** post\_comment(request, post\_pk):

*# 先获取被评论的文章，因为后面需要把评论和被评论的文章关联起来。*

*# 这里我们使用了 Django 提供的一个快捷函数 get\_object\_or\_404，*

*# 这个函数的作用是当获取的文章（Post）存在时，则获取；否则返回 404 页面给用户。*

post = get\_object\_or\_404(Post, pk=post\_pk)

*# HTTP 请求有 get 和 post 两种，一般用户通过表单提交数据都是通过 post 请求，*

*# 因此只有当用户的请求为 post 时才需要处理表单数据。*

**if** request.method == 'POST':

*# 用户提交的数据存在 request.POST 中，这是一个类字典对象。*

*# 我们利用这些数据构造了 CommentForm 的实例，这样 Django 的表单就生成了。*

form = CommentForm(request.POST)

*# 当调用 form.is\_valid() 方法时，Django 自动帮我们检查表单的数据是否符合格式要求。*

**if** form.is\_valid():

*# 检查到数据是合法的，调用表单的 save 方法保存数据到数据库，*

*# commit=False 的作用是仅仅利用表单的数据生成 Comment 模型类的实例，但还不保存评论数据到数据库。*

comment = form.save(commit=False)

*# 将评论和被评论的文章关联起来。*

comment.post = post

*# 最终将评论数据保存进数据库，调用模型实例的 save 方法*

comment.save()

*# 重定向到 post 的详情页，实际上当 redirect 函数接收一个模型的实例时，它会调用这个模型实例的 get\_absolute\_url 方法，*

*# 然后重定向到 get\_absolute\_url 方法返回的 URL。*

**return** redirect(post)

**else**:

*# 检查到数据不合法，重新渲染详情页，并且渲染表单的错误。*

*# 因此我们传了三个模板变量给 detail.html，*

*# 一个是文章（Post），一个是评论列表，一个是表单 form*

*# 注意这里我们用到了 post.comment\_set.all() 方法，*

*# 这个用法有点类似于 Post.objects.all()*

*# 其作用是获取这篇 post 下的的全部评论，*

*# 因为 Post 和 Comment 是 ForeignKey 关联的，*

*# 因此使用 post.comment\_set.all() 反向查询全部评论。*

*# 具体请看下面的讲解。*

comment\_list = post.comment\_set.all()

context = {'post': post,

'form': form,

'comment\_list': comment\_list

}

**return** render(request, 'blog/detail.html', context=context)

*# 不是 post 请求，说明用户没有提交数据，重定向到文章详情页。*

**return** redirect(post)

这个评论视图相比之前的一些视图复杂了很多，主要是处理评论的过程更加复杂。具体过程在代码中已有详细注释，这里仅就视图中出现了一些新的知识点进行讲解。

首先我们使用了 redirect 函数。这个函数位于 django.shortcuts 模块中，它的作用是对 HTTP 请求进行重定向（即用户访问的是某个 URL，但由于某些原因，服务器会将用户重定向到另外的 URL）。redirect 既可以接收一个 URL 作为参数，也可以接收一个模型的实例作为参数（例如这里的 post）。如果接收一个模型的实例，那么这个实例必须实现了 get\_absolute\_url 方法，这样 redirect 会根据 get\_absolute\_url 方法返回的 URL 值进行重定向。

另外我们使用了 post.comment\_set.all() 来获取 post 对应的全部评论。 Comment 和Post 是通过 ForeignKey 关联的，回顾一下我们当初获取某个分类 cate 下的全部文章时的代码：Post.objects.filter(category=cate)。这里 post.comment\_set.all() 也等价于 Comment.objects.filter(post=post)，即根据 post 来过滤该 post 下的全部评论。但既然我们已经有了一个 Post 模型的实例 post（它对应的是 Post 在数据库中的一条记录），那么获取和 post 关联的评论列表有一个简单方法，即调用它的 xxx\_set 属性来获取一个类似于 objects 的模型管理器，然后调用其 all 方法来返回**这个 post** 关联的全部评论。 其中 xxx\_set 中的 xxx 为关联模型的类名（小写）。例如 Post.objects.filter(category=cate) 也可以等价写为 cate.post\_set.all()。

## 绑定 URL

视图函数需要和 URL 绑定，这里我们在 comment 应用中再建一个 urls.py 文件，写上 URL 模式：

comments/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

app\_name = 'comments'

urlpatterns = [

url(r'^comment/post/(?P<post\_pk>[0-9]+)/$', views.post\_comment, name='post\_comment'),

]

别忘了给这个评论的 URL 模式规定命名空间，即 app\_name = 'comments'。

最后要在项目的 blogprokect\ 目录的 urls.py 里包含 comments\urls.py 这个文件：

blogproject/urls.py

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'', include('blog.urls')),

+ url(r'', include('comments.urls')),

]

## 更新文章详情页面的视图函数

我们可以看到评论表单和评论列表是位于文章详情页面的，处理文章详情页面的视图函数是 detail，相应地需要更新 detail，让它生成表单和从数据库获取文章对应的评论列表数据，然后传递给模板显示：

blog/views.py

**import** **markdown**

**from** **django.shortcuts** **import** render, get\_object\_or\_404

+ **from** **comments.forms** **import** CommentForm

**from** **.models** **import** Post, Category

**def** detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

post.body = markdown.markdown(post.body,

extensions=[

'markdown.extensions.extra',

'markdown.extensions.codehilite',

'markdown.extensions.toc',

])

*# 记得在顶部导入 CommentForm*

form = CommentForm()

*# 获取这篇 post 下的全部评论*

comment\_list = post.comment\_set.all()

*# 将文章、表单、以及文章下的评论列表作为模板变量传给 detail.html 模板，以便渲染相应数据。*

context = {'post': post,

'form': form,

'comment\_list': comment\_list

}

**return** render(request, 'blog/detail.html', context=context)

## 在前端渲染表单

使用 Django 表单的一个好处就是 Django 能帮我们自动渲染表单。我们在表单的视图函数里传递了一个 form 变量给模板，这个变量就包含了自动生成 HTML 表单的全部数据。在 detail.html 中通过 form 来自动生成表单。删掉原来用于占位的 HTML 评论表单代码，即下面这段代码：

<**form** action="#" method="post" class="comment-form">

<**div** class="row">

<**div** class="col-md-4">

<**label** for="id\_name">名字：</**label**>

<**input** type="text" id="id\_name" name="name" required>

</**div**>

...

</**div**> *<!-- row -->*

</**form**>

替换成如下的代码：

<**form** action="{% url 'comments:post\_comment' post.pk %}" method="post" class="comment-form">

{% csrf\_token %}

<**div** class="row">

<**div** class="col-md-4">

<**label** for="{{ form.name.id\_for\_label }}">名字：</**label**>

{{ form.name }}

{{ form.name.errors }}

</**div**>

<**div** class="col-md-4">

<**label** for="{{ form.email.id\_for\_label }}">邮箱：</**label**>

{{ form.email }}

{{ form.email.errors }}

</**div**>

<**div** class="col-md-4">

<**label** for="{{ form.url.id\_for\_label }}">URL：</**label**>

{{ form.url }}

{{ form.url.errors }}

</**div**>

<**div** class="col-md-12">

<**label** for="{{ form.text.id\_for\_label }}">评论：</**label**>

{{ form.text }}

{{ form.text.errors }}

<**button** type="submit" class="comment-btn">发表</**button**>

</**div**>

</**div**> *<!-- row -->*

</**form**>

{{ form.name }}、{{ form.email }}、{{ form.url }} 等将自动渲染成表单控件，例如 <input> 控件。

{{ form.name.errors }}、{{ form.email.errors }} 等将渲染表单对应字段的错误（如果有的话），例如用户 email 格式填错了，那么 Django 会检查用户提交的 email 的格式，然后将格式错误信息保存到 errors 中，模板便将错误信息渲染显示。

## 显示评论内容

在 detail 视图函数我们获取了全部评论数据，并通过 comment\_list 传递给了模板。和处理 index 页面的文章列表方式是一样的，我们在模板中通过 {% for %} 模板标签来循环显示文章对应的全部评论内容。

删掉占位用的评论内容的 HTML 代码，即如下的代码：

<**ul** class="comment-list list-unstyled">

<**li** class="comment-item">

<**span** class="nickname">追梦人物</**span**>

<**time** class="submit-date">2017年3月12日 14:56</**time**>

<**div** class="text">

文章观点又有道理又符合人性，这才是真正为了表达观点而写，不是为了迎合某某知名人士粉丝而写。我觉得如果琼瑶是前妻，生了三孩子后被一不知名的女人挖了墙角，我不信谁会说那个女人是追求真爱，说同情琼瑶骂小三的女人都是弱者。

</**div**>

</**li**>

...

</**ul**>

替换成如下的代码：

<**ul** class="comment-list list-unstyled">

{% for comment in comment\_list %}

<**li** class="comment-item">

<**span** class="nickname">{{ comment.name }}</**span**>

<**time** class="submit-date">{{ comment.created\_time }}</**time**>

<**div** class="text">

{{ comment.text }}

</**div**>

</**li**>

{% empty %}

暂无评论

{% endfor %}

</**ul**>

接下来尝试在详情页下的评论表单提交一些评论数据，可以看到详情页的评论列表处渲染了你提交的评论数据。



# 13 - 已知小问题修正

1 月前 字数 2692 阅读 552 评论 0

## 在模型中指定排序

为了让文章（Post）按发布时间逆序排列，即最新发表的文章排在文章列表的最前面，我们对返回的文章列表进行了排序，即各个视图函数中都有类似于 Post.objects.all().order\_by('-created\_time') 这样的代码，这导致了很多重复。因为只要是返回的文章列表，基本都是逆序排列的，因此我们可以在 Post 模型中指定 Post 的自然排序方式。

Django 允许我们在 models.Model 的子类里定义一个 Meta 的内部类，这个内部类通过指定一些属性来规定这个类该有的一些特性，例如在这里我们要指定 Post 的排序方式。首先看到 Post 的代码：

blog/models.py

**@python\_2\_unicode\_compatible**

**class** **Post**(models.Model):

...

created\_time = models.DateTimeField()

...

**def** \_\_str\_\_(self):

...

**def** get\_absolute\_url(self):

...

现在在 Post 类的内部定义一个 Meta 类，并指定排序属性：

blog/models.py

**@python\_2\_unicode\_compatible**

**class** **Post**(models.Model):

...

created\_time = models.DateTimeField()

...

**def** \_\_str\_\_(self):

...

**def** get\_absolute\_url(self):

...

**class** **Meta**:

ordering = ['-created\_time']

ordering 属性用来指定文章排序方式，['-created\_time'] 指定了依据哪个属性的值进行排序，这里指定为按照文章发布时间排序，且负号表示逆序排列。列表中可以用多个项，比如 ordering = ['-created\_time', 'title'] ，那么首先依据 created\_time 排序，如果 created\_time 相同，则再依据 title 排序。这样指定以后所有返回的文章列表都会自动按照 Meta 中指定的顺序排序，因此可以删掉视图函数中对文章列表中返回结果进行排序的代码了。

## 完善跳转链接

导航栏有一个 Black & White 的 Logo，我们希望点击它就能回到首页面，只修修改一下超链接即可。打开 base.html，修改 Logo 处的超链接：

<**header** id="site-header">

<**div** class="row">

<**div** class="logo">

- <**h1**><**a** href="index.html"><**b**>Black</**b**> **&amp;** White</**a**></**h1**>

+ <**h1**><**a** href="{% url 'blog:index' %}"><**b**>Black</**b**> **&amp;** White</**a**></**h1**>

</**div**>

...

</**div**>

</**header**>

另外导航栏还有一个首页导航按钮，也希望点击它就能回到首页面，修改的任务作为练习交给你了。

当然还有一些跳转可以完善，比如文章标题下有分类、发布时间、作者、评论量、阅读量等信息，可以设置点击分类跳转到分类页面；点击评论量就跳转到文章详情页的评论处等，这些细节部分不涉及新知识，就当做练习交给你自己完成了。

## 显示正确的评论量

有两处地方显示的评论量，显示评论量的方法很简单。回顾一下我们是如何获取某篇 post 的下的评论列表的？我们使用的是 post.comment\_set.all()。all 方法返回该 post 关联的评论列表。此外模型管理器（comment\_set 是一个特殊的模型管理器）还有一个 count 方法，返回的是数量，即 post 下有多少条评论，我们可以直接在模板中调用这个方法：{{ post.comment\_set.count }}。将评论量替换成该模板变量就可以正确显示文章的评论数了。

## 结束了么

我们通过一个博客实战项目，了解了 Django 基本的开发技术。包括如何编写模型（Model）、如何编写视图函数（View）、如何使用 Django 内置的模板系统（Template）以及如何配置路由（URL），这四大模块是 Django 开发的核心所在，现在我们已经能够基本掌握这些模块的使用方法了。

但是，还没完...

Django 提供的不仅仅是这些，我们的博客也不仅仅只有这些功能。如何实现标签云效果？如何给博客提供 RSS 订阅服务？如果实现文章搜索？如果网站需要提供用户系统，如何实现用户的注册登录？如何部署到服务器上让他人通过公网访问？这些需求都可以利用 Django 提供的 API 来实现。

另外，Django 还有海量的第三方应用来提供更加丰富的功能。比如当他人评论了我的文章时，如何收到通知提醒？网站需要提供新浪微博、微信等社交账号的登录等等，这些都可以借助 Django 第三方应用快速完成，而我们自己只需要写很少量的代码就可以了。

所以，让我们再接再厉，进入到 [Django 博客开发进阶教程](http://zmrenwu.com/category/django-advanced-blog-tutorial/)，学习更多的 Django 开发技巧，为博客提供更多的功能吧！

# 14 - 使用 Nginx 和 Gunicorn 部署 Django 博客

3 周，1 日前 字数 9720 阅读 1534 评论 18

我们博客的基础功能已经开发的基本差不多了，虽然还有很多地方可以完善，但我们还是希望早点把博客部署到服务器上，让他人可以通过外网访问。至于有待完善的地方，可以等部署完后一点点地迭代和改进。现在就让我们来把博客部署到服务器上吧！

**注意：本文的每一个步骤都在真实环境下验证无误。除非你知道自己在做什么，否则我们建议每一步均严格按照教程的指导来，这样能保证你顺利完成部署。**

## 部署前准备

我们将使用比较流行的 Nginx + Gunicorn 的方式将 Django 开发的博客部署到自己的服务器，让别人能够通过域名访问你的博客。至于 Nginx、Gunicorn 是什么暂时放到一边，读完本教程后你就会知道它们的作用和使用方法了。

为了部署我们的博客，需要满足以下两个条件：

1. 有一台可以通过外网访问的服务器。
2. 有一个域名。

如果你已经满足以上条件，可以直接跳到后面的搭建服务器部分。这里简单介绍一下我目前所知的以最低成本满足以上两个条件的方式。

### 购买服务器

如果你是学生，推荐购买阿里云服务器，学生优惠价是 9.9 元/月，而且服务器性能比较高。购买地址：[阿里云服务器学生专区](https://promotion.aliyun.com/ntms/campus2017.html)。具体的购买步骤这里就不赘述了，根据网站的指引相信你肯定能够购买成功。只是注意一点的是在选服务器类型的时候选择**公共镜像**，这样系统比较纯净。操作系统建议选 **ubuntu 14.04 64位**，这是本教程使用的服务器环境。

如果你不是学生，推荐购买搬瓦工 vps。目前最便宜的是 19.9美元/年，缺点是服务器性能没有阿里云高，但优点是顺带可以用它来搭梯子，从此访问 google、youtube 不是梦（基于 shadowsocks 只需简单几步就可以搭建起自己的梯子服务器）。同样购买的过程就不赘述了，[搬瓦工 vps 中文网](http://banwagong.cn/) 有超级详细的指引。只是注意安装操作系统时建议选 **ubuntu 14.04 64位**，这是本教程使用的服务器环境。

如果你不差那点钱，随意选择一个云服务器提供商购买一个云服务器即可。

### 购买域名

域名服务商很多，我这里使用的是 [阿里云域名注册系统](https://wanwang.aliyun.com/domain/)。域名是网站的门牌，如果打算长期运营这个网站建议多考虑考虑，选一个适当的域名。如果只是为了测试，随便注册一个域名即可，一些非常见后缀的域名非常便宜，一般 10元/年就能搞定。但注意一点根据工信部规定，以下后缀的域名需要实名认证后才能使用：

.cn/.com/.net/.top/.xyz/.vip/.club/.ren/.wang/.shop/.xin/.中国/.信息/.公司/.网络/.广东/.佛山

如果你购买的是上述后缀的域名，意味着需要提交个人的身份资料实名认证后才能正常使用，这通常需要花费几天的时间。所以如果只为了测试和学习部署的话，最好避开上述后缀的域名。

## 搭建服务器

本教程使用的本地环境为 Windows 10，服务器环境为 ubuntu 14.04（64 位）。**如果你的环境和我的有所差异导致一些命令无法执行，将这些命令转换为你所在环境的命令执行即可。**

### 远程登录到服务器

服务器通常位于云端，需要使用远程登录工具登录后才能对服务器进行操作。我使用的是 Xshell，Windows 下百度 Xshell 下载安装即可，软件对学校和个人用户是免费的。

如何远程登录到服务器这里就不赘述了，相信你参考网上的一些教程肯定能够顺利登录。假如你和我一样使用 Xshell 的话，这里有一篇很详细的教程可以参考：[教你怎么使用xshell远程连接linux服务器](http://jingyan.baidu.com/article/ab69b270b0ca3d2ca7189fdc.html)。

### 安装软件

顺利连接到远程服务器了。如果是一台全新服务器的话，通常我们是以 root 用户登录的。在 root 下部署代码不安全，最好是建一个新用户（如果你已经以非 root 用户登录的话可以跳过这一步）。下面的一些列命令将创建一个拥有超级权限的新用户：

# 在 root 用户下运行这条命令创建一个新用户，yangxg 是用户名

# 因为我叫杨学光，所以我取的用户名是 yangxg

# 选择一个你喜欢的用户名，不一定非得和我的相同

root@localhost:~# useradd -m -s /bin/bash yangxg

# 把新创建的用户加入超级权限组

root@localhost:~# usermod -a -G sudo yangxg

# 为新用户设置密码

# 注意在输密码的时候不会有字符显示，不要以为键盘坏了，正常输入即可

root@localhost:~# passwd yangxg

# 切换到创建的新用户

root@localhost:~# su - yangxg

# 切换成功，@符号前面已经是新用户名而不是 root 了

yangxg@localhost:~$

新用户创建并切换成功了。如果是新服务器的话，最好先更新一下系统，避免因为版本太旧而给后面安装软件带来麻烦。运行下面的两条命令：

yangxg@localhost:~$ sudo apt-get update

yangxg@localhost:~$ sudo apt-get upgrade

接下来就可以安装必要的软件了，这里我们需要用到的软件有 Nginx、Pytohn3、Git、pip 和 virtualenv。

yangxg@localhost:~$ sudo apt-get install nginx

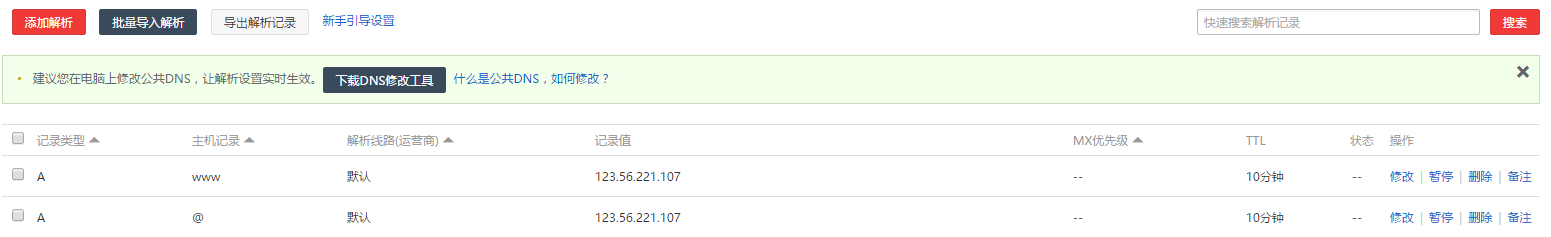
yangxg@localhost:~$ sudo apt-get install git python3 python3-pip

yangxg@localhost:~$ sudo pip3 install virtualenv

### 解析域名到服务器的 IP 地址

将域名和服务器的 IP 地址绑定后，用户就可以通过在浏览器输入域名来访问服务器了。

各大域名服务商都提供了域名解析服务，但其配置界面各有差异，请依据其指引完成域名解析。下面是我使用的阿里云域名解析页面。



### 启动 Nginx 服务

Nginx 是用来处理静态文件请求的。比如当我们访问一个博客文章详情页面时，服务器会接收到下面两种请求：

* 显示文章的详情信息，这些信息通常保存在数据库里，因此需要调用数据库获取数据。
* 图片、css、js 等存在服务器某个文件夹下的静态文件。

对于前一种请求，博客文章的数据需要借助 Django 从数据库中获取，Nginx 处理不了，它就会把这个请求转发给 Django，让 Django 去处理。而对于后一种静态文件的请求，只需要去这些静态文件所在的文件夹获取，Nginx 就会代为处理，不再麻烦 Django。

用 Django 去获取静态文件是很耗时的，但 Nginx 可以很高效地处理，这就是我们要使用 Nginx 的原因（当然其功能远不止这些）。

通过前面的步骤我们已经安装了 Nginx，并且已经把域名和服务器 IP 绑定了。运行下面的命令启动 Nginx 服务：

yangxg@localhost:~$ sudo service nginx start

在浏览器输入域名，看到如下页面说明 Nginx 启动成功了。



## 部署代码

### 部署前的项目配置

Django 项目中会有一些 CSS、JavaScript 等静态文件，为了能够方便地让 Nginx 处理这些静态文件的请求，我们把项目中的全部静态文件收集到一个统一的目录下，这个目录通常位于 Django 项目的根目录，并且命名为 static。为了完成这些任务，需要在项目的配置文件里做一些必要的配置：

blogproject/settings.py

*# 其他配置...*

STATIC\_URL = '/static/'

*# 加入下面的配置*

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

STATIC\_ROOT 指明了静态文件的收集目录，即项目根目录（BASE\_DIR）下的 static 文件夹。

为了安全起见，在生产环境下需要关闭 DEBUG 选项以及设置允许访问的域名。打开 settings.py 文件，找到 DEBUG 和 ALLOWED\_HOSTS 这两个选项，将它们设置成如下的值：

blogproject/settings.py

DEBUG = False

ALLOWED\_HOSTS = ['127.0.0.1', 'localhost ', '.zmrenwu.com']

ALLOWED\_HOSTS 是允许访问的域名列表，127.0.0.1 和 localhost 是本地访问的域名，.zmrenwu.com 是访问服务器的域名（**换成你自己的域名**）。域名前加一个点表示允许访问该域名下的子域名，比如 www.zmrenwu.com、test.zmrenwu.com 等二级域名同样允许访问。如果不加前面的点则只允许访问 zmrenwu.com。

项目还会依赖一些第三方 Python 库，为了方便在服务器上一次性安装，我们将全部依赖写入一个叫 requirements.txt 的文本文件中。激活**本地**的虚拟环境（如果你使用了虚拟环境的话），并进入项目的根目录，运行 pip freeze > requirements.txt 命令：

(blogproject\_env) C:\Users\yangxg\Workspace\blogproject>

pip freeze > requirements.txt

这时项目根目录下会生成了一个 requirements.txt 的文本文件，其内容记录了项目的全部依赖。

### 将代码上传到 GitHub

将代码上传到 GitHub 等代码托管平台，这样我们就可以方便地把代码拉取到服务器了。Git 和 GitHub 的使用相信你已经很熟悉了，这里就不赘述过程。如果不知道如何使用地话可以自行百度相关教程。

**注意数据库文件不要上传！**

### 设置服务器目录结构

接下来需要把代码上传到服务器了。我服务器上存放代码的目录结构一般是这样的：

/home/yangxg/

sites/

demo.zmrenwu.com/

env/

django-blog-tutorial/

一台服务器可能部署多个网站，所有网站代码都放在 sites/ 目录下。demo.zmrenwu.com/ 这个文件夹以网站的域名命名，便于区分。env/ 是 python 虚拟环境目录。django-blog-tutorial/ 是 Django 博客项目目录。

因此先来创建这个目录结构，**注意目录名替换为你自己的域名，以后涉及到 demo.zmrenwu.com 的地方通常都要替换你自己的域名，后面就不再一一指出了**，运行下面的命令，

yangxg@localhost:~$ mkdir -p ~/sites/demo.zmrenwu.com

这里 ~ 代表当前用户的 home 目录，即 /home/yangxg/。

接下来创建虚拟环境，先进入到 demo.zmrenwu.com 目录下，然后运行 virtualenv 命令创建虚拟环境：

yangxg@localhost:~$ cd ~/sites/demo.zmrenwu.com

yangxg@localhost:~/sites/demo.zmrenwu.com$ virtualenv --python=python3 env

注意这里使用 --python=python3 来指定克隆 Python3 的环境。因为 ubuntu 系统默认安装了 Python2，如果不特别指定的话 Virtualenv 默认克隆的是 Python2 的环境。

检查一下虚拟环境是否创建成功，运行 ls 命令列出当前目录下的文件和文件夹，看到 env 这个文件夹说明虚拟环境创建成功。

yangxg@localhost:~/sites/demo.zmrenwu.com$ ls

env

接着再从代码仓库把项目代码拉取过来，**把 git clone 后的地址换成你自己的 GitHub 仓库地址！**

yangxg@localhost:~/sites/demo.zmrenwu.com$ git clone https://github.com/zmrenwu/django-blog-tutorial.git

运行 ls 命令检查一下是否拉取成功：

yangxg@localhost:~/sites/demo.zmrenwu.com$ ls

django-blog-tutorial env

多了 django-blog-tutorial 文件夹（文件夹名称由你的 GitHub 仓库名决定），说明拉取成功了。

### 安装项目依赖

激活虚拟环境，再进入到项目根目录，即 requirements.txt 所在的目录，安装项目的全部依赖：

yangxg@localhost:~/sites/demo.zmrenwu.com$ source env/bin/activate

(env) yangxg@localhost:~/sites/demo.zmrenwu.com$ cd django-blog-tutorial/

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ pip install -r requirements.txt

### 收集静态文件

虚拟环境下继续运行 python manage.py collectstatic 命令收集静态文件到 static 目录下：

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ python manage.py collectstatic

### 生成数据库

虚拟环境下继续运行 python manage.py migrate 命令创建数据库文件：

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ python manage.py migrate

### 创建超级用户

虚拟环境下继续运行 python manage.py createsuperuser 命令创建一个超级用户，方便我们进入 Django 管理后台。这和本地开发时是一样的，具体请参照：[在 Django Admin 后台文章](http://www.zmrenwu.com/post/9/)。

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ python manage.py createsuperuser

## 配置 Nginx

接下是配置 Nginx 来处理用户请求。

先在服务器的 /etc/nginx/sites-available/ 目录下新建一个配置文件，文件名我一般就设置为域名。写上下面的配置内容：

/**etc**/**nginx**/**sites-available**/**demo**.**zmrenwu**.**com**

**server** {

charset utf-8;

listen 80;

server\_name demo.zmrenwu.com; ①

location /static { ②

alias /home/yangxg/sites/demo.zmrenwu.com/django-blog-tutorial/static;

}

**location** / { ③

proxy\_set\_header Host $host;

proxy\_pass http://unix:/tmp/demo.zmrenwu.com.socket;

}

}

① 服务的域名为 demo.zmrenwu.com。

② 所有URL 带有 /static 的请求均由 Nginx 处理，alias 指明了静态文件的存放目录。

③ 其它请求转发给 Django 处理。proxy\_pass 后面使用了 unix 套接字，其作用是防止端口冲突，这里就不再详述。

至于怎么在服务器新建文件和写文件，请自行学习一点点 vi 编辑器的用法，这里也不一一讲解了。

我们在 /etc/nginx/sites-available/ 放置了配置文件，接下来需要创建一个符号链接，把这个配置文件加入到启用的网站列表中去，被启用网站的目录在 /etc/nginx/sites-enabled/，你可以理解为从 sites-available/ 目录下发送了一个配置文件的快捷方式到 sites-enabled/ 目录。具体命令如下：

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ sudo ln -s /etc/nginx/sites-available/demo.zmrenwu.com /etc/nginx/sites-enabled/demo.zmrenwu.com

## 使用 Gunicorn

Gunicorn 一般用来管理多个进程，有进程挂了Gunicorn 可以把它拉起来，防止服务器长时间停止服务，还可以动态调整 worker 的数量，请求多的时候增加 worker 的数量，请求少的时候减少。

在虚拟环境下，安装 Gunicorn：

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ pip install gunicorn

用 Gunicorn 启动服务器进程：

(env) yangxg@localhost:~/sites/demo.zmrenwu.com/django-blog-tutorial$ gunicorn --bind unix:/tmp/demo.zmrenwu.com.socket blogproject.wsgi:application

浏览器输入域名，可以看到访问成功了！

### 自动启动 Gunicorn

现在 Gunicorn 是我们手工启动的，万一哪天服务器崩溃重启了又得重新手工启动。为此我们写一个自动启动脚本，这样当服务器重新启动后，脚本会帮我们重启 Gunicorn。先按 Ctrl + c 停止刚才启动的服务器进程。

写一个启动脚本，这样当服务器重启后能自动引导 Gunicorn 的启动。脚本位于 /etc/init/ 目录下，且脚本文件名必须以 .conf 结尾：

/etc/init/gunicorn-demo.zmrenwu.com.conf

start on net-device-up ①

stop on shutdown

respawn ②

setuid yangxg ③

chdir /home/yangxg/sites/demo.zmrenwu.com/django-blog-tutorial ④

exec ../env/bin/gunicorn --bind unix:/tmp/demo.zmrenwu.com.socket blogproject.wsgi:application ⑤

① start on net-device-up 确保只在服务器联网时才启动 Gunicorn。

② 如果进程崩溃了（比如服务器重启或者进程因为某些以外情况被 kill），respawn 将自动重启 Gunicorn。

③ setuid 确保以 yangxg 用户的身份（换成你自己的用户名）运行 Gunicorn 进程。

④ chdir 进入到指定目录，这里进入项目的根目录。

⑤ exec 执行进程，即开启服务器进程。

现在可以用 start 命令启动 Gunicorn 了：

sudo start gunicorn-demo.zmrenwu.com

以后如果更新了代码，只要运行下面的命令重启一下 Nginx 和 Gunicorn 就可以使新的代码生效了：

sudo service nginx reload

sudo restart gunicorn-demo.zmrenwu.com

## 使用 CDN 加快 Bootstrap 和 jQuery 的加载速度

我们的项目使用了 Bootstrap 和 jQuery，这两个文件我们是从本地加载的。如果服务器性能比较差的话，加载需要耗费很长的时间，网站打开的速度就变得无法忍受。我们使用 CDN 来加快加载速度。具体来说，替换 base.html 的几个静态文件的加载标签：

base.html

- <**link** rel="stylesheet" href="{% static 'blog/css/bootstrap.min.css' %}">

- <**script** src="{% static 'blog/js/jquery-2.1.3.min.js' %}"></**script**>

- <**script** src="{% static 'blog/js/bootstrap.min.js' %}"></**script**>

+ <**link** href="https://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">

+ <**script** src="https://cdn.bootcss.com/jquery/2.1.3/jquery.min.js"></**script**>

+ <**script** src="https://cdn.bootcss.com/bootstrap/3.3.7/js/bootstrap.min.js"></**script**>

这样网站访问的速度将大大提升！

## 部署过程自动化

在整个部署过程中我们运行了十几条命令，手动输入了 N 个字符。如果每次更新代码都要远程连接到服务器执行这些命令的话将变得非常麻烦。接下来的教程我们将介绍使用 Fabric 自动化整个部署过程。写好部署脚本后，只需要执行一条命令，就可以非常方便地自动完成整个部署。

**15 - 使用 Fabric 自动化部署**

3 周前 字数 2945 阅读 716 评论 4

在 [使用 Nginx 和 Gunicorn 部署 Django 博客](http://zmrenwu.com/post/20/) 中，我们通过手工方式将代码部署到了服务器。整个过程涉及到十几条命令，输了 N 个字符。一旦我们本地的代码有更新，整个过程又得重复来一遍，这将变得非常繁琐。

使用 Fabric 可以在服务器中自动执行命令。因为整个代码部署过程都是相同的，只要我们用 Fabric 写好部署脚本，以后就可以通过运行脚本自动完成部署了。

**安装 Fabric**

Fabric 目前仅支持 Python2，如果你的系统中只有 Python3 版本，请先安装一个 Python2 的版本，推荐安装 Python 2.7。Python3 和 Python2 可以共存于一个系统，所以不用担心同时安装两个版本的 Python 会发生冲突。

接下就可以简单地通过 pip 命令安装 Fabric 了。**注意要使用 Python2 环境下的 pip，这样才能把 Fabric 安装到 Python2 环境下。**

pip install fabric

**部署过程回顾**

在写 Fabric 脚本之前，我们先来回顾一下当我们在本地开发环境下更新了代码后，在服务器上的整个部署过程。

1. 远程连接服务器。
2. 进入项目根目录，从远程仓库拉取最新的代码。
3. 如果项目引入了新的依赖，需要执行 pip install -r requirement.txt 安装最新依赖。
4. 如果修改或新增了项目静态文件，需要执行 python manage.py collectstatic 收集静态文件。
5. 如果数据库发生了变化，需要执行 python manage.py migrate 迁移数据库。
6. 重启 Nginx 和 Gunicorn 使改动生效。

整个过程就是这样，把每一步操作翻译成 Fabric 对应的脚本代码，这样一个自动化部署脚本就完成了。

**编写 Fabric 脚本**

Fabric 脚本通常位于 fabfile.py 文件里，因此先在**项目根目录**下建一个 fabfile.py 文件。

根据上述过程编写的脚本代码如下：

blogproject/fabfile.py

**from** **fabric.api** **import** env, run

**from** **fabric.operations** **import** sudo

GIT\_REPO = "you git repository" ①

env.user = 'you host username' ②

env.password = 'you host password'

*# 填写你自己的主机对应的域名*

env.hosts = ['demo.zmrenwu.com']

*# 一般情况下为 22 端口，如果非 22 端口请查看你的主机服务提供商提供的信息*

env.port = '22'

**def** deploy():

source\_folder = '/home/yangxg/sites/zmrenwu.com/django-blog-tutorial' ③

run('cd *%s* && git pull' % source\_folder) ④

run("""

cd {} &&

../env/bin/pip install -r requirements.txt &&

../env/bin/python3 manage.py collectstatic --noinput &&

../env/bin/python3 manage.py migrate

""".format(source\_folder)) ⑤

sudo('restart gunicorn-demo.zmrenwu.com') ⑥

sudo('service nginx reload')

① 你的代码托管仓库地址。

② 配置一些服务器的地址信息和账户信息，各参数的含义分别为：

* env.user：用于登录服务器的用户名
* env.password：用户名对应的密码
* env.hosts：服务器的 IP 地址，也可以是解析到这个 IP 的域名
* env.port：SSH 远程服务器的端口号

③ 需要部署的项目根目录在服务器上的位置。

④ 通过 run 方法在服务器上执行命令，传入的参数为需要执行的命令，用字符串包裹。这里执行了两条命令，不同命令间用 && 符号连接：

1. cd 命令进入到需要部署的项目根目录
2. git pull 拉取远程仓库的最新代码

⑤ 对应上述部署过程中 3-5 的几条命令。因为启用了虚拟环境，所以运行的是虚拟环境 ../env/bin/ 下的 pip 和 python

⑥ 重启 Gunicorn 和 Nginx，由于这两条命令要在超级权限下运行，所以使用了 sudo 方法而不是 run 方法。

**注意全部的脚本代码要放在 deploy 函数里，Fabric 会自动检测 fabfile.py 脚本中的 deploy 函数并运行。**

**由于脚本中有登录服务器的用户名和密码等敏感信息，不要把 fabfile.py 文件也上传到公开的代码托管仓库。**

**执行 Fabric 自动部署脚本**

进入 fabfile.py 文件所在的目录，在 **Python2** 的环境下用 fab 命令运行这个脚本文件。

比如我的是 Windows 环境，Python2 安装在 C:\Python27\ 下，那么运行：

C:\Python27\Scripts\fab deploy

这时 Fabric 会自动检测到 fabfile.py 脚本中的 deploy 函数并运行，你会看到命令行输出了一系列字符串，如果在最后看到

Done. Disconnecting from zmrenwu.com... done.

说明脚本运行成功。

而如果看到

Aborting. Disconnecting from zmrenwu.com... done.

说明脚本运行中出错，检查一下命令行输入的错误信息，修复问题后重新运行脚本即可。以后当你在本地开发完相关功能后，只需要执行这一个脚本文件，就可以自动把最新代码部署到服务器了。