

第一章 创建博客应用

欢迎来到Django 2 by example的教程。你看到的是目前全网唯一翻译该书的教程。

本书将介绍如何创建可以用于生产环境的完整Django项目。如果你还没有安装Django，本章在第一部分中将介绍如何安装Django，之后的内容还包括创建一个简单的博客应用。本章的目的是让读者对Django的整体运行有一个概念，理解Django的各个组件如何交互运作，知道创建一个应用的基础方法。本书将指导你创建一个个完整的项目，但不会对所有细节进行详细阐述，Django各个组件的内容会在全书的各个部分内进行解释。

本章的主要内容有：

- 安装Django并创建第一个项目
- 设计数据模型和进行模型迁移（migrations）
- 为数据模型创建管理后台
- 使用QuerySet和模型管理器
- 创建视图、模板和URLs
- 给列表功能的视图添加分页功能
- 使用基于类的视图

1 安装Django

如果已经安装了Django，可以跳过本部分到[创建第一个Django项目](#)小节。Django是Python的一个包（模块），所以可以安装在任何Python环境。如果还没有安装Django，本节是一个用于本地开发的快速安装Django指南。

Django 2.0需要Python解释器的版本为3.4或更高。在本书中，采用Python 3.6.5版本，如果使用Linux或者macOS X，系统中也许已经安装Python（部分Liunx发行版初始安装Python2.7），对于Windows系统，从

<https://www.python.org/downloads/windows/> 下载Python安装包。

译者在此强烈建议使用基于UNIX的系统进行开发。

如果不确定系统中是否已经安装了Python，可以尝试在系统命令行中输入 `python` 然后查看输出结果，如果看到类似下列信息，则说明Python已经安装：

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果安装的版本低于3.4，或者没有安装Python，从 <https://www.python.org/downloads/> 下载并安装。

由于我们使用Python 3，所以暂时不需要安装数据库，因为Python 3自带一个轻量级的SQLite3数据库可以用于Django开发。如果打算在生产环境中部署Django项目，需要使用更高级的数据库，比如PostgreSQL，MySQL或者Oracle数据库。关于如何在Django中使用数据库，可以看官方文档

<https://docs.djangoproject.com/en/2.0/topics/install/#database-installation>。

译者注：在翻译本书和实验代码的时候，译者的开发环境是Centos 7.5 1804 + Python 3.7.0 + Django 2.1.0（最后一章升级到Django 2.1.2），除了后文会提到的一个旧版本第三方库插件冲突的问题之外，未发现任何兼容性问题。

1.1 创建独立的Python开发环境

推荐使用 `virtualenv` 创建独立的开发环境，这样可以对不同的项目应用不同版本的模块，比将这些模块直接安装为系统Python的第三方库要灵活很多。另一个使用 `virtualenv` 的优点是安装Python模块的时候不需要任何管理员权限。在系统命令行中输入如下命令来安装 `virtualenv`：

```
pip install virtualenv
```

在安装完 `virtualenv` 之后，通过以下命令创建一个独立环境：

```
virtualenv my_env
```

译者注：需要将 `virtualenv` 的所在路径添加到系统环境变量 `PATH` 中，对于Django也是如此，不然无法直接执行 `django-admin` 命令。

这个命令会在当前目录下创建一个 `my_env/` 目录，其中放着一个Python虚拟环境。在虚拟环境中安装的Python包实际会被安装到 `my_env/lib/python3.6/site-packages` 目录中。

如果操作系统中安装的是Python 2.X，必须再安装Python 3.X，还需要设置 `virtualenv` 虚拟Python 3.X的环境。

可以通过如下命令查找Python 3的安装路径，然后创建虚拟环境：

```
zenx$ which python3  
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3  
zenx$ virtualenv my_env -p /Library/Frameworks/Python.framework/Versions/3.6/bin/python3
```

根据Linux发行版的不同，上边的代码也会有所不同。在创建了虚拟环境对应的目录之后，使用如下命令激活虚拟环境：

```
source my_env/bin/activate
```

激活之后，在命令行模式的提示符前会显示括号包围该虚拟环境的名称，如下所示：

```
(my_env)laptop:~ zenx$
```

开启虚拟环境后，随时可以通过在命令行中输入 `deactivate` 来退出虚拟环境。

关于 `virtualenv` 的更多内容可以查看 <https://virtualenv.pypa.io/en/latest/>。

`virtualenvwrapper` 这个工具可以方便的创建和管理系统中的所有虚拟环境，需要在系统中先安装 `virtualenv`，可以到 <https://virtualenvwrapper.readthedocs.io/en/latest/> 下载。

1.2 使用PIP安装Django

推荐使用 `pip` 包安装Django。Python 3.6已经预装了 `pip`，也可以在 <https://pip.pypa.io/en/stable/installing/> 找到 `pip` 的安装指南。

使用下边的命令安装Django：

```
pip install Django==2.0.5
```

译者这里安装的是2.1版。

Django会被安装到虚拟环境下的 `site-packages/` 目录中。

现在可以检查Django是否已经成功安装，在系统命令行模式运行 `python`，然后导入Django，检查版本，如下：

```
>>> import django
>>> django.get_version()
'2.0.5'
```

如果看到了这个输出，就说明Django已经成功安装了。

Django的其他安装方式，可以查看官方文档完整的安装指南：

<https://docs.djangoproject.com/en/2.0/topics/install/>。

2 创建第一个Django项目

本书的第一个项目是创建一个完整的博客项目。Django提供了一个创建项目并且初始化其中目录结构和文件的命令，在命令行模式中输入：

```
django-admin startproject mysite
```

这会创建一个项目，名称叫做`mysite`。

避免使用Python或Django的内置名称作为项目名称。

看一下项目目录的结构：

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

这些文件解释如下：

- `manage.py`：是一个命令行工具，可以通过这个文件管理项目。其实是一个 `django-admin.py` 的包装器，这个文件在创建项目过程中不需要编辑。
- `mysite/`：这是项目目录，由以下文件组成：
 - `__init__.py`：一个空文件，告诉Python将 `mysite` 看成一个包。
 - `settings.py`：这是当前项目的设置文件，包含一些初始设置
 - `urls.py`：这是 `URL patterns` 的所在地，其中的每一行URL，表示URL地址与视图的一对一映射关系。
 - `wsgi.py`：这是自动生成的当前项目的WSGI程序，用于将项目作为一个WSGI程序启动。

自动生成的 `settings.py` 是当前项目的配置文件，包含一个用于使用SQLite 3 数据库的设置，以及一个叫做 `INSTALLED_APPS` 的列表。`INSTALLED_APPS` 包含Django默认添加到一个新项目中的所有应用。在之后的 [项目设置](#) 部分会接触到这些应用。

为了完成项目创建，还必须在数据库里创建起 `INSTALLED_APPS` 中的应用所需的数据表，打开系统命令行输入下列命令：

```
cd mysite
python manage.py migrate
```

会看到如下输出：

```
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 auth.0009_alter_user_last_name_max_length... OK
Applying sessions.0001_initial... OK
```

这些输出表示Django刚刚执行的数据库迁移（migrate）工作，在数据库中创建了这些应用所需的数据表。在本章的[创建和执行迁移](#)部分会详细介绍migrate命令。

2.1 运行开发中的站点

Django提供了一个轻量级的Web服务程序，无需在生产环境即可快速测试开发中的站点。启动这个服务之后，会检查所有的代码是否正确，还可以在代码被修改之后，自动重新载入修改后的代码，但部分情况下比如向项目中加入了新的文件，还需要手工关闭服务再重新启动。

在命令行中输入下列命令就可以启动站点：

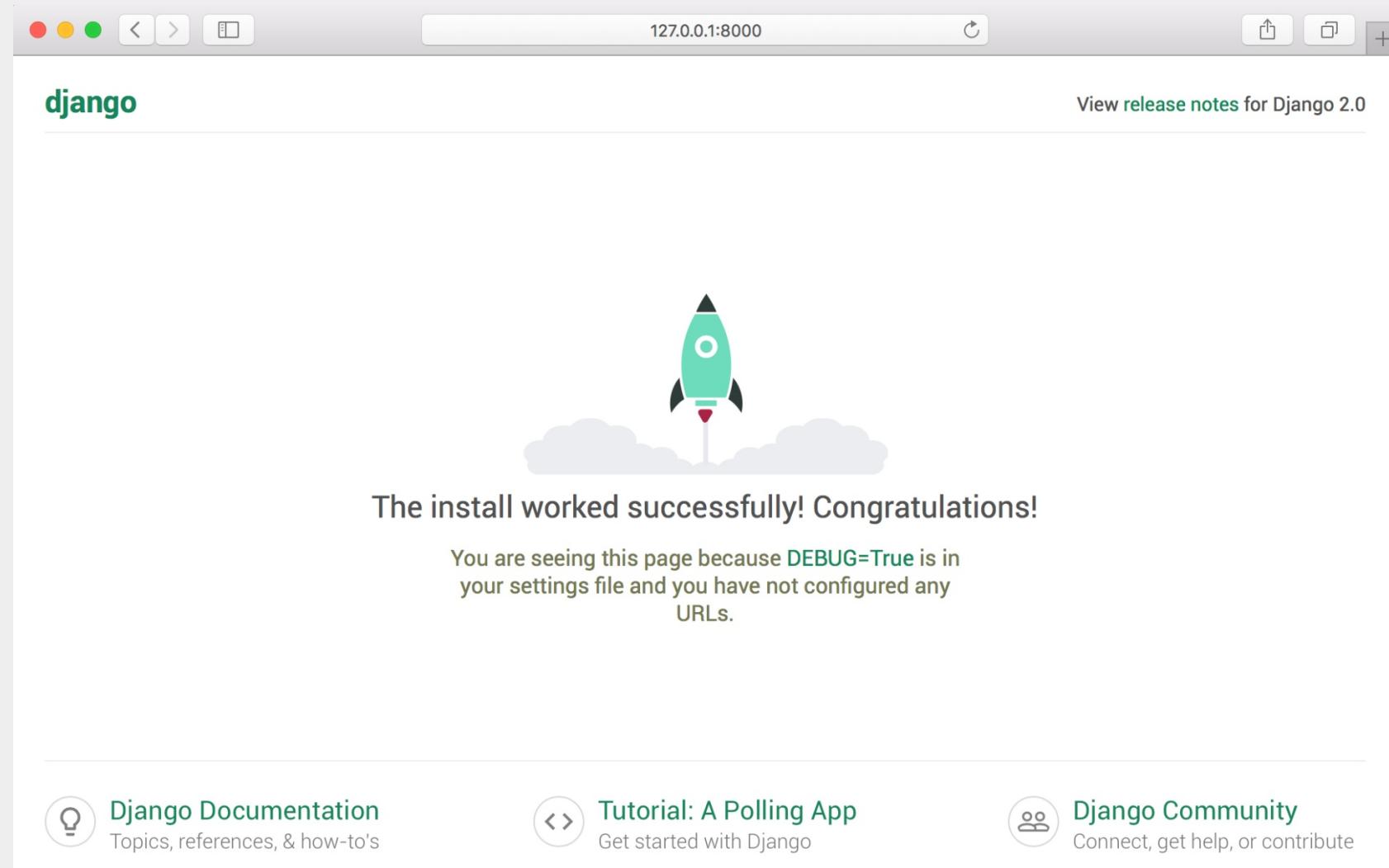
```
python manage.py runserver
```

应该会看到下列输出：

```
Performing system checks...

System check identified no issues (0 silenced).
May 06, 2018 - 17:17:31
Django version 2.0.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

现在可以在浏览器中打开 <http://127.0.0.1:8000/>，会看到成功运行站点的页面，如下图所示：



能看到这个页面，说明Django正在运行，如果此时看一下刚才启动站点的命令行窗口，可以看到浏览器的GET请求：

```
[15/May/2018 17:20:30] "GET / HTTP/1.1" 200 16348
```

站点接受的每一个HTTP请求，都会显示在命令行窗口中，如果站点发生错误，也会将错误显示在该窗口中。

在启动站点的时候，还可以指定具体的主机地址和端口，或者使用另外一个配置文件，例如：

```
python manage.py runserver 127.0.0.1:8001 --settings=mysite.settings
```

如果站点需要在不同环境下运行，单独为每个环境创建匹配的配置文件。

当前这个站点只能用作开发测试，不能够配置为生产用途。想要将Django配置到生产环境中，必须通过一个Web服务程序比如Apache，Gunicorn或者uWSGI，将Django作为一个WSGI程序运行。使用不同web服务程序部署Django请参考：<https://docs.djangoproject.com/en/2.0/howto/deployment/wsgi/>。本书的**第十三章上线**会介绍如何配置生产环境。

2.2 项目设置

打开 `settings.py` 看一下项目设置，其中列出了一些设置，但这只是Django所有设置的一部分。可以在 <https://docs.djangoproject.com/en/2.0/ref/settings/> 查看所有的设置和初始值。

文件中的以下设置值得注意：

- `DEBUG` 是一个布尔值，控制DEBUG模式的开启或关闭。当设置为 `True` 时，Django会将所有的日志和错误信息都打印在窗口中。在生产环境中则必须设置为 `False`，否则会导致信息泄露。
- `ALLOWED_HOSTS` 在本地开发的时候，无需设置。在生产环境中，`DEBUG` 设置为 `False` 时，必须将主机名/IP地址填入该列表中，以让Django为该主机/IP提供服务。
- `INSTALLED_APPS` 列出了每个项目当前激活的应用，Django默认包含下列应用：
 - `django.contrib.admin`：管理后台应用
 - `django.contrib.auth`：用户身份认证

- `django.contrib.contenttypes`：追踪ORM模型与应用的对应关系
 - `django.contrib.sessions`：session应用
 - `django.contrib.messages`：消息应用
 - `django.contrib.staticfiles`：管理站点静态文件
- `MIDDLEWARE` 是中间件列表。
 - `ROOT_URLCONF` 指定项目的根URL patterns配置文件。
 - `DATABASE` 是一个字典，包含不同名称的数据库及其具体设置，必须始终有一个名称为 `default` 的数据库， 默认使用SQLite 3数据库。
 - `LANGUAGE_CODE` 站点默认的语言代码。
 - `USE_TZ` 是否启用时区支持，Django可以支持根据时区自动切换时间显示。如果通过 `startproject` 命令创建站点，该项默认被设置为 `True`。

如果目前对这些设置不太理解也没有关系，在之后的章节中这里的设置都会使用到。

2.3 项目 (projects) 与应用 (applications)

在整本书中，这两个词会反复出现。在Django中，像我们刚才那样的一套目录结构和其中的设置就是一个Django可识别的项目。应用指的就是一组Model（数据模型）、Views（视图）、Templates（模板）和URLs的集合。Django框架通过使用应用，为站点提供各种功能，应用还可以被复用在不同的项目中。你可以将一个项目理解为一个站点，站点中包含很多功能，比如博客，wiki，论坛，每一种功能都可以看作是一个应用。

2.4 创建一个应用

我们将从头开始创建一个博客应用，进入项目根目录（`manage.py` 文件所在的路径），在系统命令行中输入以下命令创建第一个Django应用：

```
python manage.py startapp blog
```

这条命令会在项目根目录下创建一个如下结构的应用：

```
blog/
__init__.py
admin.py
apps.py
migrations/
__init__.py
models.py
tests.py
views.py
```

这些文件的含义为：

- `admin.py`：用于将模型注册到管理后台，以便在Django的管理后台（Django administration site）查看。管理后台也是一个可选的应用。
- `apps.py`：当前应用的主要配置文件
- `migrations` 这个目录包含应用的数据迁移记录，用来追踪数据模型的变化然后和数据库同步。
- `models.py`：当前应用的数据模型，所有的应用必须包含一个 `models.py` 文件，但其中内容可以是空白。
- `test.py`：为应用增加测试代码的文件
- `views.py`：应用的业务逻辑部分，每一个视图接受一个HTTP请求，处理这个请求然后返回一个HTTP响应。

3 设计博客应用的数据架构 (data schema)

schema是一个数据库名词，一般指的是数据在数据库中的组织模式或者说架构。我们将通过在Django中定义数据模型来设计我们博客应用在数据库中的数据架构。一个数据模型，是指一个继承了 `django.db.models.Model` 的Python类。

Django会在 `models.py` 文件中定义的每一个类，在数据库中创建对应的数据表。Django为创建和操作数据模型提供了一系列便捷的API (Django ORM)：

我们首先来定义一个 `Post` 类，在 `blog` 应用下的 `models.py` 文件中添加下列代码：

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

class Post(models.Model):
    STATUS_CHOICES = (('draft', 'Draft'), ('published', 'Published'))
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250, unique_for_date='publish')
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now())
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='draft')

    class Meta:
        ordering = ('-publish',)

    def __str__(self):
        return self.title
```

这是我们为了博客中每一篇文章定义的数据模型：

- `title`：这是文章标题字段。这个字段被设置为 `Charfield` 类型，在SQL数据库中对应 `VARCHAR` 数据类型

- `slug`：该字段通常在URL中使用。`slug`是一个短的字符串，只能包含字母，数字，下划线和减号。将使用`slug`字段构成优美的URL，也方便搜索引擎搜索。其中的`unique_for_date`参数表示不允许两条记录的`publish`字段日期和`title`字段全都相同，这样就可以使用文章发布的日期与`slug`字段共同生成一个唯一的URL标识该文章。
- `author`：是一个外键字段。通过这个外键，告诉Django一篇文章只有一个作者，一个作者可以写多篇文章。对于这个字段，Django会在数据库中使用外键关联到相关数据表的主键上。在这个例子中，这个外键关联到Django内置用户验证模块的`User`数据模型上。`on_delete`参数表示删除外键关联的内容时候的操作，这个并不是Django特有的定义，而是SQL数据库的标准操作；将其设置为`CASCADE`意味着如果删除一个作者，将自动删除所有与这个作者关联的文章，对于该参数的设置，可以查看https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.ForeignKey.on_delete。
`related_name`参数设置了从`User`到`Post`的反向关联关系，用`blog_posts`为这个反向关联关系命名，稍后会学习到该关系的使用。
- `body`：是文章的正文部分。这个字段是一个文本域，对应SQL数据库的`TEXT`数据类型。
- `publish`：文章发布的时间。使用了`django.utils.timezone.now`作为默认值，这是一个包含时区的时间对象，可以将其认为是带有时区功能的Python标准库中的`datetime.now`方法。
- `created`：表示创建该文章的时间。`auto_now_add`表示当创建一行数据的时候，自动用创建数据的时间填充。
- `updated`：表示文章最后一次修改的时间，`auto_now`表示每次更新数据的时候，都会用当前的时间填充该字段。
- `status`：这个字段表示该文章的状态，使用了一个`choices`参数，所以这个字段的值只能为一系列选项中的值。

Django提供了很多不同类型的字段可以用于数据模型，具体可以参考：

<https://docs.djangoproject.com/en/2.0/ref/models/fields/>。

在数据模型中的Meta类表示存放模型的元数据。通过定义`ordering = ('-publish',)`，指定了Django在进行数据库查询的时候，默认按照发布时间的逆序将查询结果排序。逆序通过加在字段名前的减号表示。这样最近发布的文章就会排在前边。

`__str__()`方法是Python类的功能，供显示给人阅读的信息，这里将其设置为文章的标题。Django在很多地方比如管理后台中都调用该方法显示对象信息。

如果你之前使用的是Python 2.X，注意在Python 3中，所有的字符串都已经是原生Unicode格式，所以只需要定义 `__str__()` 方法，`__unicode__()` 方法已被废弃。

3.1 激活应用

为了让Django可以为应用中的数据模型创建数据表并追踪数据模型的变化，必须在项目里激活应用。要激活应用，编辑 `settings.py` 文件，添加 `blog.apps.BlogConfig` 到 `INSTALLED_APPS` 设置中：

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
]
```

`BlogConfig` 类是我们应用的配置类。现在Django就已经知道项目中包含了一个新应用，可以载入这个应用的数据模型了。

3.2 创建和执行迁移

创建好了博客文章的数据模型，之后需要将其变成数据库中的数据表。Django提供数据迁移系统，用于追踪数据模型的变动，然后将变化写入到数据库中。我们之前执行过的 `migrate` 命令会对 `INSTALLED_APPS` 中的所有应用进行扫描，根据数据模型和已经存在的迁移数据执行数据库同步操作。

首先，我们需要来为 `Post` 模型创建迁移数据，进入项目根目录，输入下列命令：

```
python manage.py makemigrations blog
```

会看到如下输出：

```
Migrations for 'blog':  
  blog/migrations/0001_initial.py  
    - Create model Post
```

该命令执行后会在 `blog` 应用下的 `migrations` 目录里新增一个 `0001_initial.py` 文件，可以打开该文件看一下迁移数据是什么样子的。一个迁移数据文件里包含了与其他迁移数据的依赖关系，以及实际要对数据库执行的操作。

为了了解Django实际执行的SQL语句，可以使用 `sqlmigrate` 加上迁移文件名，会列出要执行的SQL语句，但不会实际执行。在命令行中输入下列命令然后观察数据迁移的指令：

```
python manage.py sqlmigrate blog 0001
```

输出应该如下所示：

```
BEGIN;  
--  
-- Create model Post  
--  
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
"title" varchar(250) NOT NULL, "slug" varchar(250) NOT NULL, "body" text NOT  
NULL, "publish" datetime NOT NULL, "created" datetime NOT NULL, "updated"  
datetime NOT NULL, "status" varchar(10) NOT NULL, "author_id" integer NOT  
NULL REFERENCES "auth_user" ("id"));  
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
```

```
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

具体的输出根据你使用的数据库会有变化。上边的输出针对SQLite数据库。可以看到表名被设置为应用名加上小写的类名（`blog_post`）也可以通过在`Meta`类中使用`db_table`属性设置表名。Django自动为每个模型创建了主键，也可以通过设置某个模型字段参数`primary_key=True`来指定主键。默认的主键列名叫做`id`，和这个列同名的`id`字段会自动添加到你的数据模型上。（即`Post`类被Django添加了`Post.id`属性）。

然后来让数据库与新的数据模型进行同步，在命令行中输入下列命令：

```
python manage.py migrate
```

会看到如下输出：

```
Applying blog.0001_initial... OK
```

这样就对`INSTALLED_APPS`中的所有应用执行完了数据迁移过程，包括我们的`blog`应用。在执行完迁移之后，数据库中的数据表就反映了我们此时的数据模型。

如果之后又编辑了`models.py`文件，对已经存在的数据模型进行了增删改，或者又添加了新的数据模型，必须重新执行`makemigrations`创建新的数据迁移文件然后执行`migrate`命令同步数据库。

4 为数据模型创建管理后台站点 (administration site)

定义了`Post`数据模型之后，可以为方便的管理其中的数据创建一个简单的管理后台。Django内置了一个管理后台，这个管理后台动态的读入数据模型，然后创建一个完备的管理界面，从而可以方便的管理数据。这是一个可以“拿来就用”的方便工具。

管理后台功能其实也是一个应用叫做 `django.contrib.admin`，默认包含在 `INSTALLED_APPS` 设置中。

4.1 创建超级用户

要使用管理后台，需要先注册一个超级用户，输入下列命令：

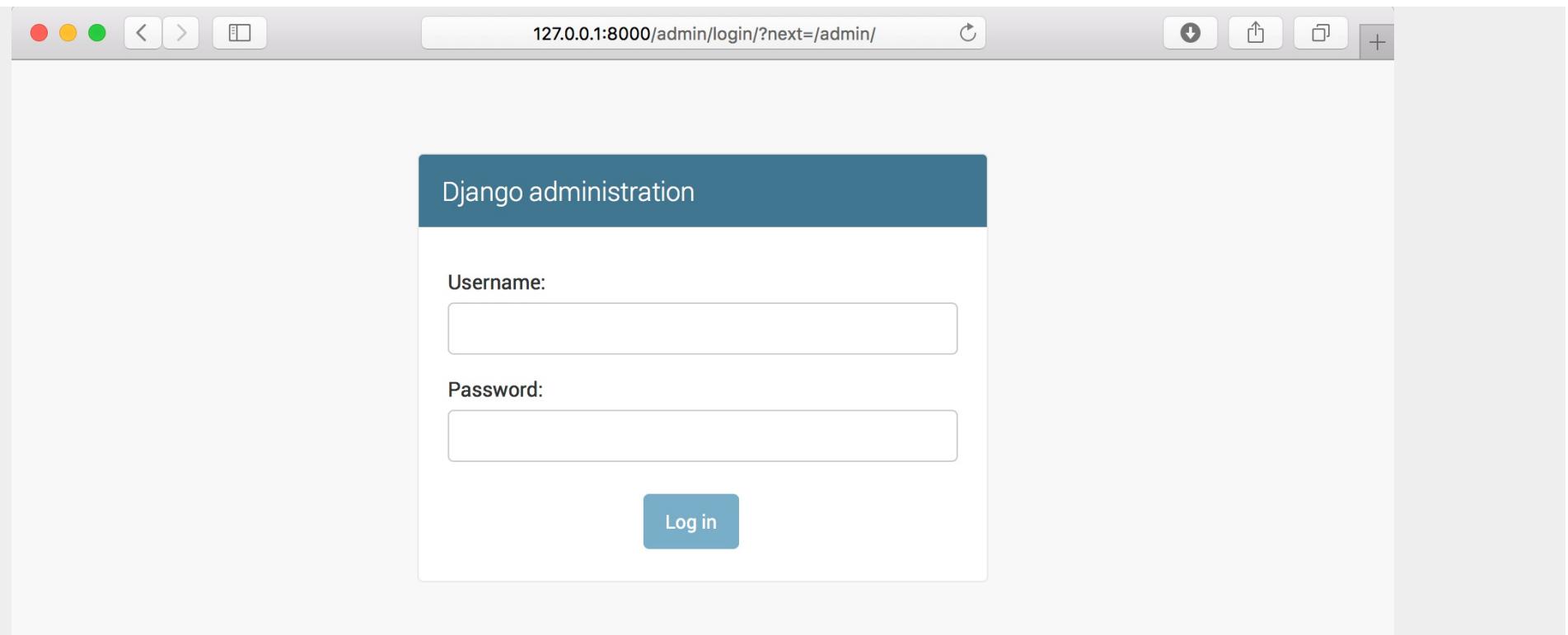
```
python manage.py createsuperuser
```

会看到下列输出，输入用户名、密码和邮件：

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: *****
Password (again): *****
Superuser created successfully.
```

4.2 Django 管理后台

使用 `python manage.py runserver` 启动站点，然后打开 <http://127.0.0.1:8000/admin/>，可以看到如下的管理后台登录页面：



输入刚才创建的超级用户的用户名和密码，可以看到管理后台首页，如下所示：

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	 Add	 Change
Users	 Add	 Change

Recent actions

My actions

None available

Group 和 User 已经存在于管理后台中，这是因为设置中默认启用了 django.contrib.auth 应用的原因。如果你点击 Users，可以看到刚刚创建的超级用户。还记得 blog 应用的 Post 模型与 User 模型通过 author 字段产生外键关联吗？

4.3 向管理后台内添加模型

我们把 Post 模型添加到管理后台中，编辑 blog 应用的 admin.py 文件为如下这样：

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

之后刷新管理后台页面，可以看到 Post 类出现在管理后台中：

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	 Add	 Change
Users	 Add	 Change

BLOG

Posts	 Add	 Change
-----------------------	----------------------	-------------------------

Recent actions

My actions

None available

看上去好像很简单。每当在管理后台中注册一个模型，就能迅速在管理后台中看到它，还可以对其进行增删改查。

点击Posts右侧的Add链接，可以看到Django根据模型的具体字段动态的生成了添加页面，如下所示：

Add post

Title:

Slug:

Author:



Body:

Publish:

Date: 2017-12-14

Today | 

Time: 08:54:24

Now | 

Note: You are 2 hours ahead of server time.

Status:

Draft



[Save and add another](#)

[Save and continue editing](#)

[SAVE](#)

Django对于每个字段使用不同的表单插件（form widgets，控制该字段实际在页面上对应的HTML元素）。即使是比较复杂的字段比如 `DateTimeField`，也会以简单的界面显示出来，类似于一个JavaScript的时间控件。

填写完这个表单然后点击SAVE按钮，被重定向到文章列表页然后显示一条成功信息，像下面这样：

The screenshot shows the Django administration interface. At the top, it says "Django administration" and "WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below that, the breadcrumb navigation shows "Home > Blog > Posts". A green success message box contains the text "The post \"Who was Django Reinhardt?\" was added successfully." In the main content area, there is a heading "Select post to change" and a "ADD POST +". Below this, there is a "Action:" dropdown set to "-----" and a "Go" button. Underneath, there is a list of posts with checkboxes: one for "POST" and another for "Who was Django Reinhardt?". A summary at the bottom indicates "1 post".

可以再录入一些文章数据，为之后数据库相关操作做准备。

4.4 自定义模型在管理后台的显示

现在我们来看一下如何自定义管理后台，编辑 `blog` 应用的 `admin.py`，修改成如下：

```
from django.contrib import admin
from .models import Post
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish', 'status')
```

这段代码的意思是将我们的模型注册到管理后台中，并且创建了一个类继承 `admin.ModelAdmin` 用于自定义模型的展示方式和行为。`list_display` 属性指定那些字段在详情页中显示出来。`@admin.register()` 装饰器的功能与之前的 `admin.site.register()` 一样，用于将 `PostAdmin` 类注册成 `Post` 的管理类。

再继续添加一些自定义设置，如下所示：

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish', 'status',)
    list_filter = ('status', 'created', 'publish', 'author',)
    search_fields = ('title', 'body',)
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ('author',)
    date_hierarchy = 'publish'
    ordering = ('status', 'publish',)
```

回到浏览器，刷新一下posts的列表页，会看到如下所示：

Select post to change

[ADD POST +](#)

Search

Search

< 2017 December 14

Action:



Go

0 of 1 selected

 TITLE

SLUG

AUTHOR

PUBLISH

2 ▲ STATUS 1 ▲

 Who was Django Reinhardt?

who-was-django-reinhardt

admin

Dec. 14, 2017, 8:54 a.m.

Draft

1 post

FILTER

By status

All

Draft

Published

By created

Any date

Today

Past 7 days

This month

This year

By publish

Any date

Today

Past 7 days

This month

This year

可以看到在该页面上显示的字段就是 `list_display` 中的字段。页面出现了一个右侧边栏用于筛选结果，这个功能由 `list_filter` 属性控制。页面上方出现了一个搜索栏，这是因为在 `search_fields` 中定义了可搜索的字段。在搜索栏的下方，出现了时间层级导航条，这是在 `date_hierarchy` 中定义的。还可以看到文章默认通过 `Status` 和 `Publish` 字段进行排序，这是由 `ordering` 属性设置的。

这个时候点击Add Post，可以发现也有变化。当输入文章标题时，`slug` 字段会根据标题自动填充，这是因为设置了 `prepopulated_fields` 属性中 `slug` 字段与 `title` 字段的对应关系。现在 `author` 字段旁边出现了一个搜索图标，并且可以按照ID来查找和显示作者，如果在用户数量很大的时候，这就方便太多了。

通过短短几行代码，就可以自定义模型在管理后台中的显示方法，还有很多自定义管理后台和扩展管理后台功能的方法，会在以后的各章中逐步遇到。

5 使用QuerySet和模型管理器（managers）

现在我们有了一个功能齐备的管理后台用于管理博客的内容数据，现在可以来学习如何从数据库中查询数据并且对结果进行操作了。Django具有一套强大的API，可以供你轻松的实现增删改查的功能，这就是Django Object-relational-mapper即Django ORM，可以兼容MySQL，PostgreSQL，SQLite和Oracle，可以在 `settings.py` 的 `DATABASES` 中修改数据库设置。可以通过编辑数据库的路由设置让Django同时使用多个数据库。

一旦你创建好了数据模型，Django就提供了一套API供你操作数据模型，详情可以参考
<https://docs.djangoproject.com/en/2.0/ref/models/>。

5.1 创建数据对象

打开系统的终端窗口，运行如下命令：

```
python manage.py shell
```

然后录入如下命令：

```
>>>from django.contrib.auth.models import User  
>>>from blog.models import Post
```

```
>>>user = User.objects.get(username='admin')
>>>post = Post(title='Another post', slug='another-post', body='Post body', author = user)
>>>post.save()
```

让我们来分析一下这段代码做的事情：我们先通过用户名 `admin` 取得 `user` 对象，就是下边这条命令：

```
user = User.objects.get(username='admin')
```

`get()`方法允许从数据库中取出单独一个数据对象。如果找不到对应数据，会抛出 `DoseNotExist` 异常，如果结果超过一个，会抛出 `MultipleObjectsReturn` 异常，这两个异常都是被查找的类的属性。

然后我们通过下边这条命令，使用了标题，简称和文章内容，以及指定 `author` 字段为刚取得的 `User` 对象，新建了一个 `Post` 对象：

```
post = Post(title='Another post', slug='another-post', body='Post body', author = user)
```

这个对象暂时保存在内存中，没有被持久化（写入）到数据库中。

最后，我们通过 `save()` 方法将 `Post` 对象写入到数据库中：

```
post.save()
```

这条命令实际会转化成一条 `INSERT` SQL语句。现在我们已经知道了如何在内存中先创建一个数据对象然后将其写入到数据库中的方法，我们还可以使用 `create()` 方法一次性创建并写入数据库，像这样：

```
Post.objects.create(title='One more post', slug='One more post', body='Post body', author=user)
```

5.2 修改数据对象

现在，修改刚才的post对象的标题：

```
>>> post.title = 'New title'  
>>> post.save()
```

这次 `save()` 方法实际转化为一个 `UPDATE` SQL语句。

对数据对象做的修改直到调用 `save()` 方法才会被存入数据库。

5.3 查询数据

Django ORM的全部使用都基于`QuerySet`（查询结果集对象，由于该术语使用频繁，因此在之后的文章中不再进行翻译）。一个查询结果集是一系列从数据库中取得的数据对象，经过一系列的过滤条件，最终组合到一起构成的一个对象。

之前已经了解了使用 `Post.objects.get()` 方法从数据库中取出一个单独的数据对象，每个模型都有至少一个管理器，默认的管理器叫做 `objects`。通过使用一个模型管理器，可以得到一个`QuerySet`，想得到一个数据表里的所有数据对象，可以使用默认模型管理器的 `all()` 方法，像这样：

```
>>> all_posts = Post.objects.all()
```

这样就取得了一个包含数据库中全部post的`Queryset`，值得注意的是，`QuerySet`还没有被执行（即执行SQL语句），因为`QuerySet`是惰性求值的，只有在确实要对其进行表达式求值的时候，`QuerySet`才会被执行。惰性求值特性使得`QuerySet`非常有用。如果我们不是把`QuerySet`的结果赋值给一个变量，而是直接写在Python命令行中，对应的SQL语句就会立刻被执行，因为会强制对其求值：

```
>>> Post.objects.all()
```

译者注：原书一直没有非常明确的指出这几个概念，估计是因为本书不是面向Django初学者所致。这里译者总结一下：
数据模型Model类=数据表，数据模型类的实例=数据表的一行数据（不一定是来自于数据库的，也可能是内存中创建的），查询结果集=包装一系列数据模型类实例的对象。

5.3.1 使用 filter() 方法

可以使用模型管理器的 `filter()` 过滤所需的数据，例如，可以过滤出所有2017年发布的博客文章：

```
Post.objects.filter(publish__year=2017)
```

还可以同时使用多个字段过滤，比如选出所有 `admin` 作者于2017年发布的文章：

```
Post.objects.filter(publish__year=2017, author__username='admin')
```

这和链式调用QuerySet的结果一样：

```
Post.objects.filter(publish__year=2017).filter(author__username='admin')
```

QuerySet中使用的条件查询采用双下划线写法，比如例子中的 `publish__year`，双下划线还有一个用法是从关联的模型中取其字段，例如 `author__username`。

5.3.2 使用 exclude() 方法

使用模型管理器的 `exclude()` 从结果集中去除符合条件的数据。例如选出2017年发布的所有标题不以 `Why` 开头的文章：

```
Post.objects.filter(publish__year=2017).exclude(title__startswith='Why')
```

5.3.3 使用 `order_by()` 方法

对于查询出的结果，可以使用 `order_by()` 方法按照不同的字段进行排序。例如选出所有文章，使其按照 `title` 字段排序：

```
Post.objects.order_by('title')
```

默认会采用升序排列，如果需要使用降序排列，在字符串格式的字段名前加一个减号：

```
Post.objects.order_by('-title')
```

译者注：如果不指定`order_by`的排序方式，但在Meta中指定了顺序，则默认会优先以Meta中的顺序列出。

5.4 删除数据

如果想删除一个数据，可以对一个数据对象直接调用 `delete()` 方法：

```
post = Post.objects.get(id=1)
post.delete()
```

当外键中的 `on_delete` 参数被设置为 `CASCADE` 时，删除一个对象会同时删除所有对其有依赖关系的对象，比如删除作者的时候该作者的文章会一并删除。

译者注：`filter()`, `exclude()`, `all()` 这三个方法都返回一个QuerySet对象，所以可以任意链式调用。

5.5 QuerySet何时会被求值

可以对一个QuerySet串联任意多的过滤方法，但只有到该QuerySet实际被求值的时候，才会进行数据库查询。

QuerySet仅在下列时候才被实际执行：

- 第一次迭代QuerySet
- 执行切片操作，例如 `Post.objects.all()[:3]`
- pickled或者缓存QuerySet的时候
- 调用QuerySet的 `repr()` 或者 `len()` 方法
- 显式对其调用 `list()` 方法将其转换成列表
- 将其用在逻辑判断表达式中。比如 `bool()`, `or`, `and` 和 `if`

如果对结构化程序设计中的表达式求值有所了解的话，就可以知道只有表达式被实际求值的时候，QuerySet才会被执行。译者在这里推荐伯克利大学的 [CS 61A: Structure and Interpretation of Computer Programs](#) Python教程。

5.6 创建模型管理器

像之前提到的那样，类名后的 `.objects` 就是默认的模型管理器，所有的ORM方法都通过模型管理器操作。除了默认的管理器之外，我们还可以自定义这个管理器。我们要创建一个管理器，用于获取所有 `status` 字段是 `published` 的文章。

自行编写模型管理器有两种方法：一是给默认的管理器增加新的方法，二是修改默认的管理器。第一种方法就像是给你提供了一个新的方法例如：`Post.objects.my_manager()`，第二种方法则是直接使用新的管理器例如：`Post.my_manager.all()`。我们想实现的方式是：`Post.published.all()` 这样的管理器。

在 `blog` 的 `models.py` 里增加自定义的管理器：

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super(PublishedManager, self).get_queryset().filter(status='published')

class Post(models.Model):
    # .....
    objects = models.Manager() # 默认的管理器
    published = PublishedManager() # 自定义管理器
```

模型管理器的 `get_queryset()` 方法返回后续方法要操作的QuerySet，我们重写了该方法，以让其返回所有过滤后的结果。现在我们已经自定义好了管理器并且将其添加到了 `Post` 模型中，现在可以使用这个管理器进行数据查询，来测试一下：

启动包含Django环境的Python命令行模式：

```
python manage.py shell
```

现在可以取得所有标题开头是 `Who`，而且已经发布的文章（实际的查询结果根据具体数据而变）：

```
Post.published.filter(title__startswith="Who")
```

6 创建列表和详情视图函数

在了解了ORM的相关知识以后，就可以来创建视图了。视图是一个Python中的函数，接受一个HTTP请求作为参数，返回一个HTTP响应。所有返回HTTP响应的业务逻辑都在视图中完成。

首先，我们会创建应用中的视图，然后会为每个视图定义一个匹配的URL路径，最后，会创建HTML模板将视图生成的结果展示出来。每一个视图都会向模板传递参数并且渲染模板，然后返回一个包含最终渲染结果的HTTP响应。

6.1 创建视图函数

来创建一个视图用于列出所有的文章。编辑 `blog` 应用的 `views.py` 文件：

```
from django.shortcuts import render, get_object_or_404
from .models import Post

def post_list(request):
    posts = Post.published.all()
    return render(request, 'blog/post/list.html', {'posts': posts})
```

我们创建了第一个视图函数--文章列表视图。`post_list` 目前只有一个参数 `request`，这个参数对于所有的视图都是必需的。在这个视图中，取得了所有已经发布（使用了 `published` 管理器）的文章。

最后，使用由 `django.shortcuts` 提供的 `render()` 方法，使用一个HTML模板渲染结果。`render()` 方法的参数分别是 `request`，HTML模板的位置，传给模板的变量名与值。`render()` 方法返回一个带有渲染结果（HTML文本）的 `HttpResponse` 对象。`render()` 方法还会将 `request` 对象携带的变量也传给模板，在模板中可以访问所有模板上下文管理器设置的变量。模板上下文管理器就是将变量设置到模板环境的可调用对象，会在第三章学习到。

再写一个显示单独一篇文章的视图，在 `views.py` 中添加下列函数：

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post, status="published", publish__year=year, publish__month=month,
                           publish__day=day)
    return render(request, 'blog/post/detail.html', {'post': post})
```

这就是我们的文章详情视图。这个视图需要 `year`，`month`，`day` 和 `post` 参数，用于获取一个指定的日期和简称的文章。还记得之前创建模型时设置 `slug` 字段的 `unique_for_date` 参数，这样通过日期和简称可以找到唯一的一篇文章（或者找不到）。使用 `get_object_or_404()` 方法来获取文章，这个方法返回匹配的一个数据对象，或者在找不到的情况下返回一个HTTP 404错误（not found）。最后使用 `render()` 方法通过一个模板渲染页面。

6.2 为视图配置URL

URL pattern的作用是将URL映射到视图上。一个URL pattern由一个正则字符串，一个视图和可选的名称（该名称必须唯一，可以在整个项目环境中使用）组成。Django接到对于某个URL的请求时，按照顺序从上到下试图匹配URL，停在第一个匹配成功的URL处，将 `HttpRequest` 类的一个实例和其他参数传给对应的视图并调用视图处理本次请求。

在 `blog` 应用下目录下边新建一个 `urls.py` 文件，然后添加如下内容：

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

app_name = 'blog'
urlpatterns = [
    # post views
    path('', views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/', views.post_detail, name='post_detail'),
]
```

上边的代码中，通过 `app_name` 定义了一个命名空间，方便以应用为中心组织URL并且通过名称对应到URL上。然后使用 `path()` 设置了两条具体的URL pattern。第一条没有任何的参数，对应 `post_list` 视图。第二条需要如下四个参数并且对应到 `post_detail` 视图：

- `year`：需要匹配一个整数
- `month`：需要匹配一个整数

- `day`：需要匹配一个整数
- `post`：需要匹配一个slug形式的字符串

我们使用了一对尖括号从URL中获取这些参数。任何URL中匹配上这些内容的文本都会被捕捉为这个参数的对应的类型值。例如`<int:year>`会匹配到一个整数形式的字符串然后会给模板传递名称为`int`的变量，其值为捕捉到的字符串转换为整数后的值。而`<slug:post>`则会被转换成一个名称为`post`，值为slug类型（仅有ASCII字符或数字，减号，下划线组成的字符串）的变量传给视图。

对于URL匹配的类型，可以参考<https://docs.djangoproject.com/en/2.0/topics/http/urls/#path-converters>

如果使用`path()`无法满足需求，则可以使用`re_path()`，通过Python正则表达式匹配复杂的URL。参考https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.re_path了解`re_path()`的使用方法，参考<https://docs.python.org/3/howto/regex.html>了解Python中如何使用正则表达式。

为每个视图创建单独的`urls.py`文件是保持应用可被其他项目重用的最好方式。

现在我们必须把`blog`应用的URL包含在整个项目的URL中，到`mysite`目录下编辑`urls.py`：

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

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

这行新的URL使用`include`方法导入了`blog`应用的所有URL，使其位于`blog/` URL路径下，还指定了命名空间`blog`。

URL命名空间必须在整个项目中唯一。之后我们方便的通过使用命名空间来快速指向具体的URL，例如`blog:post_list`

和 `blog:post_detail`。关于URL命名空间可以参考
<https://docs.djangoproject.com/en/2.0/topics/http/urls/#url-namespaces>。

6.3 规范模型的URL

可以使用在上一节创建的 `post_detail` URL来为Post模型的每一个数据对象创建规范化的URL。通常的做法是给模型添加一个 `get_absolute_url()` 方法，该方法返回对象的URL。我们将使用 `reverse()` 方法通过名称和其他参数来构建URL。编辑 `models.py` 文件

```
from django.urls import reverse

class Post(models.Model):
    # .....
    def get_absolute_url():
        return reverse('blog:post_detail', args=[self.publish.year, self.publish.month, self.publish.day, self.slug])
```

之后在模板中，就可以使用 `get_absolute_url()` 创建超链接到具体数据对象。

译者注：原书这里写得很简略，实际上反向解析URL是创建结构化站点非常重要的内容，可以参考Django 1.11版本的[Django进阶-路由系统](#)了解原理，Django 2.0此部分变化较大，需研读官方文档。

7 为视图创建模板

已经为 `blog` 应用配置好了URL pattern，现在需要将内容通过模板展示出来。

在 `blog` 应用下创建如下目录：

```
templates/
  blog/
    base.html
    post/
      list.html
      detail.html
```

这就是模板的目录结构。`base.html` 包含页面主要的HTML结构，并且将结构分为主体内容和侧边栏两部分。

`list.html` 和 `detail.html` 会分别继承 `base.html` 并渲染各自的内容。

Django提供了强大的模板语言用于控制数据渲染，由 **模板标签(template tags)**， **模板变量(template variables)**， **模板过滤器(template filters)** 组成：

- `template tags`：进行渲染控制，类似 `{% tag %}`
- `template variables`：可认为是模板标签的一种特殊形式，即只是一个变量，渲染的时候只替换内容，类似 `{{ variable }}`
- `template filters`：附加在模板变量上改变变量最终显示结果，类似 `{{ variable|filter }}`

所有内置的模板标签和过滤器可以参考 <https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>。

编辑 `base.html`，添加下列内容：

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>{% block title %}{% endblock %}</title>
  <link rel="stylesheet" href="{% static "css/blog.css" %}">
</head>
<body>
  <div id="content">
```

```
{% block content %}  
    {% endblock %}  
</div>  
<div id="sidebar">  
    <h2>My blog</h2>  
    <p>This is my blog.</p>  
</div>  
</body>  
</html>
```

{% load static %} 表示导入由 django.contrib.staticfiles 应用提供的 static 模板标签，导入之后，在整个当前模板中都可以使用 {% static %} 标签从而导入静态文件例如 blog.css（可在本书配套源码 blog 应用的 static/ 目录下找到，将其拷贝到你的项目的相同位置）。

还可以看到有两个 {% block %} 表示这个标签的开始与结束部分定义了一个块，继承该模板的模板将用具体内容替换这两个块。这两个块的名称是 title 和 content。

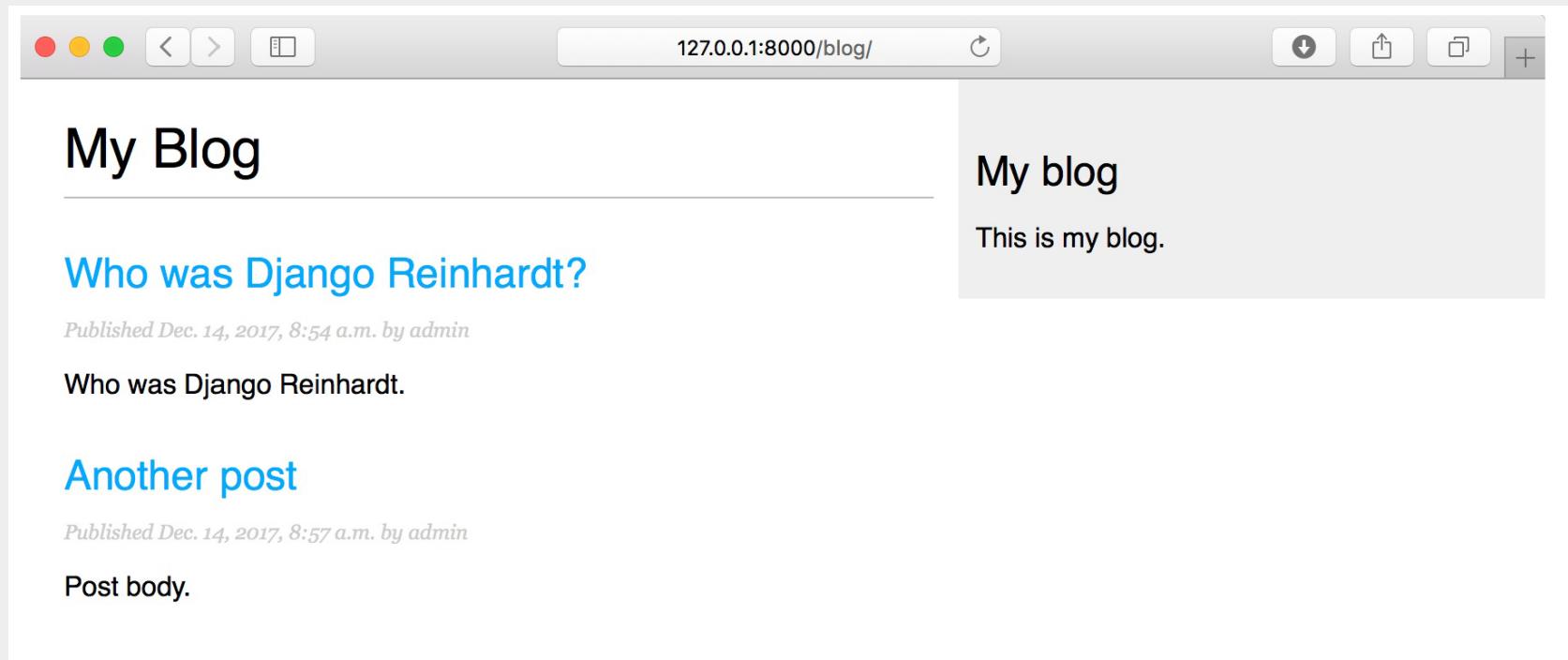
编辑 post/list.html：

```
{% extends "blog/base.html" %}  
{% block title %}My Blog{% endblock %}  
{% block content %}  
    <h1>My Blog</h1>  
    {% for post in posts %}  
        <h2>  
            <a href="{{ post.get_absolute_url }}">  
                {{ post.title }}  
            </a>  
        </h2>  
        <p class="date">  
            Published {{ post.publish }} by {{ post.author }}  
        </p>
```

```
{% post.body|truncatewords:30|linebreaks %}  
{% endfor %}  
{% endblock %}
```

通过使用 `{% extends %}`，让该模板继承了母版 `blog/base.html`，然后用实际内容填充了 `title` 和 `content` 块。通过迭代所有的文章，展示文章标题，发布日期，作者、正文及一个链接到文章的规范化URL。在正文部分使用了两个filter：`truncatewords` 用来截断指定数量的文字，`linebreaks` 将结果带上一个HTML换行。filter可以任意连用，每个都在上一个的结果上生效。

打开系统命令行输入 `python manage.py runserver` 启动站点，然后在浏览器中访问 <http://127.0.0.1:8000/blog/>，可以看到如下页面（如果没有文章，通过管理后台添加一些）：



然后编辑 `post/detail.html` :

```
{% extends 'blog/base.html' %}  
{% block title %}  
{{ post.title }}  
{% endblock %}  
  
{% block content %}  
    <h1>{{ post.title }}</h1>  
    <p class="date">  
        Published {{ post.publish }} by {{ post.author }}  
    </p>  
    {{ post.body|linebreaks }}  
{% endblock %}
```

现在可以回到刚才的页面，点击任何一篇文章可以看到详情页：



看一下此时的URL，应该类似 `/blog/2017/12/14/who-was-djangoreinhardt/`。这就是我们生成的规范化的URL。

8 添加分页功能

当输入一些文章后，你会很快意识到需要将所有的文章分页进行显示。Django自带了一个分页器可以方便地进行分页。

编辑 blog 应用的 `views.py` 文件，修改 `post_list` 视图：

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # 每页显示3篇文章
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # 如果page参数不是一个整数就返回第一页
        posts = paginator.page(1)
    except EmptyPage:
        # 如果页数超出总页数就返回最后一页
        posts = paginator.page(paginator.num_pages)
    return render(request, 'blog/post/list.html', {'page': page, 'posts': posts})
```

分页器相关代码解释如下：

1. 使用要分页的内容和每页展示的内容数量，实例化 `Paginator` 类得到 `paginator` 对象
2. 通过 `get()` 方法获取 `page` 变量，表示当前的页码
3. 调用 `paginator.page()` 方法获取要展示的数据
4. 如果 `page` 参数不是一个整数就返回第一页，如果页数超出总页数就返回最后一页
5. 把页码和要展示的内容传给页面。

现在需要为分页功能创建一个单独的模板，以让该模板可以包含在任何使用分页功能的页面中，在 `blog` 应用的 `templates/` 目录中新建 `pagination.html`，添加如下代码：

```
<div class="pagination">
    <span class="step-links">
        {% if page.has_previous %}
            <a href="?page={{ page.previous_page_number }}>Previous</a>
        {% endif %}
        <span class="current">
            Page {{ page.number }} of {{ page.paginator.num_pages }}.
        </span>
        {% if page.has_next %}
            <a href="?page={{ page.next_page_number }}>Next</a>
        {% endif %}
        </span>
    </div>
```

这个用于分页的模板接受一个名称为 `Page` 的对象，然后显示前一页，后一页和总页数。为此，回到 `blog/post/list.html` 文件，在 `{% content %}` 中的最下边增加一行：

```
{% block content %}
    # .....
    {% include 'pagination.html' with page = posts %}
{% endblock %}
```

由于视图传递给列表页的 `Page` 对象的名称叫做 `posts`，所以通过 `with` 重新指定了变量名称以让分页模板也能正确接收该对象。

打开浏览器到 <http://127.0.0.1:8000/blog/>，可以看到页面如下：

The screenshot shows a Django blog application running in a web browser. The URL in the address bar is 127.0.0.1:8000/blog/. The main content area displays three blog posts:

- Miles Davis favourite songs**
Published Dec. 14, 2017, 10:01 p.m. by admin
Miles Dewey Davis III was an American jazz trumpeter, bandleader, and composer.
- Notes on Duke Ellington**
Published Dec. 14, 2017, 9:58 p.m. by admin
Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.
- Another post**
Published Dec. 14, 2017, 11:45 a.m. by admin
Post body.

At the bottom left, there is a navigation link: **Page 1 of 2. Next**. On the right side of the main content area, there is a sidebar with the title **My blog** and the text **This is my blog.**

9 使用基于类的视图

Python中类可以取代函数，视图是一个接受HTTP请求并返回HTTP响应的可调用对象，所以基于函数的视图（FBV）也可以通过基于类的视图（CBV）来实现。Django为CBV提供了基类 `View`，包含请求分发功能和其他一些基础功能。

CBV相比FBV有如下优点

- 可编写单独的方法对应不同的HTTP请求类型如GET, POST, PUT等请求, 不像FBV一样需要使用分支
- 使用多继承创建可复用的类模块(也叫做 **mixins**)

可以看一下关于CBV的介绍: <https://docs.djangoproject.com/en/2.0/topics/class-based-views/intro/>。

我们用Django的内置CBV类 `ListView` 来改写 `post_list` 视图, `ListView` 的作用是列出任意类型的数据。编辑 `blog` 应用的 `views.py` 文件, 添加下列代码:

```
from django.views.generic import ListView
class PostListView(ListView):
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

这个CBV和 `post_list` 视图函数的功能类似, 在上边的代码里做了以下工作:

- 使用 `queryset` 变量查询所有已发布的文章。实际上, 可以不使用这个变量, 通过指定 `model = Post`, 这个CBV就会去进行 `Post.objects.all()` 查询获得全部文章。
- 设置 `posts` 为模板变量的名称, 如果不设置 `context_object_name` 参数, 默认的变量名称是 `object_list`
- 设置 `paginate_by` 为每页显示3篇文章
- 通过 `template_name` 指定需要渲染的模板, 如果不指定, 默认使用 `blog/post_list.html`

打开 `blog` 应用的 `urls.py` 文件, 注释掉刚才的 `post_list` URL pattern, 为 `PostListView` 类增加一行:

```
urlpatterns = [
    # post views
```

```
# path('', views.post_list, name='post_list'),
path('',views.PostListView.as_view(),name='post_list'),
path('<int:year>/<int:month>/<int:day>/<slug:post>/', views.post_detail, name='post_detail'),
]
```

为了正常使用分页功能，需要使用正确的变量名称，Django内置的 `ListView` 返回的变量名称叫做 `page_obj`，所以必须修改 `post/list.html` 中导入分页模板的那行代码：

```
{% include 'pagination.html' with page=page_obj %}
```

在浏览器中打开 <http://127.0.0.1:8000/blog/>，看一下是否和原来使用 `post_list` 的结果一样。这是一个简单的CBV示例，会在 **第十章** 更加深入的了解CBV的使用。

总结

这一章通过创建一个简单的博客应用，学习了基础的Django框架使用方法：设计了数据模型并且进行了数据模型迁移，创建了视图，模板和URLs，还学习了分页功能。下一章将学习给博客增加评论系统和标签分类功能，以及通过邮件分享文章链接的功能。

第二章 增强博客功能

在之前的章节创建基础的博客应用，现在可以变成具备通过邮件分享文章，带有评论和标签系统等功能的完整博客。在这一章会学习到如下内容：

- 使用Django发送邮件
- 创建表单并且通过视图控制表单
- 通过模型生成表单
- 集成第三方应用
- 更复杂的ORM查询

1 通过邮件分享文章

首先来制作允许用户通过邮件分享文章链接的功能。在开始之前，先想一想你将如何使用视图、URLs和模板来实现这个功能，然后看一看你需要做哪些事情：

- 创建一个表单供用户填写名称和电子邮件地址收件人，以及评论等。
- 在 `views.py` 中创建一个视图控制这个表单，处理接收到的数据然后发送电子邮件
- 为新的视图在 `urls.py` 中配置URL
- 创建展示表单的模板

1.1 使用Django创建表单

Django内置一个表单框架，可以简单快速的创建表单。表单框架具有自定义表单字段、确定实际显示的方式和验证数据的功能。

Django使用两个类创建表单：

- `Form`：用于生成标准的表单
- `ModelForm`：用于从模型生成表单

在 `blog` 应用中创建一个 `forms.py` 文件，然后编写：

```
from django import forms

class EmailPostForm(forms.Form):
    name = forms.CharField(max_length=25)
    email = forms.EmailField()
    to = forms.EmailField()
    comments = forms.CharField(required=False, widget=forms.Textarea)
```

这是使用`forms`类创建的第一个标准表单，通过继承内置 `Form` 类，然后设置字段为各种类型，用于验证数据。

表单可以编写在项目的任何位置，但通常将其编写在对应应用的 `forms.py` 文件中。

`name` 字段是 `Charfield` 类型，会被渲染为 `<input type="text">` HTML标签。每个字段都有一个默认的 `widget` 参数决定该字段被渲染成的HTML元素类型，可以通过 `widget` 参数改写。在 `comments` 字段中，使用了 `widget=forms.Textarea` 令该字段被渲染为一个 `<textarea>` 元素，而不是默认的 `<input>` 元素。

字段验证也依赖于字段属性。例如：`email` 和 `to` 字段都是 `EmailField` 类型，两个字段都接受一个有效的电子邮件格式的字符串，否则这两个字段会抛出 `forms.ValidationError` 错误。表单里还存在的验证是：`name` 字段的最大长度 `maxlength` 是25个字符，`comments` 字段的 `required=False` 表示该字段可以没有任何值。所有的这些设置都会影响到表单验证。本表单只使用了很少一部分的字段类型，关于所有表单字段可以参考 <https://docs.djangoproject.com/en/2.0/ref/forms/fields/>。

1.2 通过视图控制表单

现在需要写一个视图，用于处理表单提交来的数据，当表单成功提交的时候发送电子邮件。编辑 blog 应用的 `views.py` 文件：

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # 通过id 获取 post 对象
    post = get_object_or_404(Post, id=post_id, status='published')
    if request.method == "POST":
        # 表单被提交
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # 验证表单数据
            cd = form.cleaned_data
            # 发送邮件.....
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post, 'form': form})
```

这段代码的逻辑如下：

- 定义了 `post_share` 视图，参数是 `request` 对象和 `post_id`。
- 使用 `get_object_or_404()` 方法，通过 ID 和 `published` 取得所有已经发布的文章中对应 ID 的文章。
- 这个视图同时用于显示空白表单和处理提交的表单数据。我们先通过 `request.method` 判断当前请求是 `POST` 还是 `GET` 请求。如果是 `GET` 请求，展示一个空白表单；如果是 `POST` 请求，需要处理表单数据。

处理表单数据的过程如下：

1. 视图收到 `GET` 请求，通过 `form = EmailPostForm()` 创建一个空白的 `form` 对象，展示在页面中是一个空白的表单供用户填写。

2. 用户填写并通过 POST 请求提交表单，视图使用 `request.POST` 中包含的表单数据创建一个表单对象：

```
if request.method == 'POST':  
    # 表单被提交  
    form = EmailPostForm(request.POST)
```

3. 在上一步之后，调用表单对象的 `is_valid()` 方法。这个方法会验证表单中所有的数据是否有效，如果全部通过验证会返回 `True`，任意一个字段未通过验证，`is_valid()` 就会返回 `False`。如果返回 `False`，此时可以在 `form.errors` 属性中查看错误信息。

4. 如果表单验证失败，我们将这个表单对象渲染回页面，页面中会显示错误信息。

5. 如果表单验证成功，可以通过 `form.cleaned_data` 属性访问表单内所有通过验证的数据，这个属性类似于一个字典，包含字段名与值构成的键值对。

如果表单验证失败，`form.cleaned_data` 只会包含通过验证的数据。

现在就可以来学习如何使用Django发送邮件了。

1.3 使用Django发送邮件

使用Django发送邮件比较简单，需要一个本地或者外部的SMTP服务器，然后在 `settings.py` 文件中加入如下设置：

- `EMAIL_HOST`：邮件主机，默认是 `localhost`
- `EMAIL_PORT`：SMTP服务端口，默认是25
- `EMAIL_HOST_USER`：SMTP服务器的用户名
- `EMAIL_HOST_PASSWORD`：SMTP服务器的密码
- `EMAIL_USE_TLS`：是否使用TLS进行连接
- `EMAIL_USE_SSL`：是否使用SSL进行连接

如果无法使用任何SMTP服务器，则可以将邮件打印在命令行窗口中，在 `settings.py` 中加入下列这行：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

这样会把所有的邮件内容显示在控制台，非常便于测试。

如果没有本地SMTP服务器，可以使用很多邮件服务供应商提供的SMTP服务，以下是使用Google的邮件服务示例：

```
EMAIL_HOST = 'smtp.gmail.com'  
EMAIL_HOST_USER = 'your_account@gmail.com'  
EMAIL_HOST_PASSWORD = 'your_password'  
EMAIL_PORT = 587  
EMAIL_USE_TLS = True
```

输入 `python manage.py shell`，在命令行环境中试验一下发送邮件的指令：

```
from django.core.mail import send_mail  
send_mail('Django mail', 'This e-mail was sent with Django.', 'your_account@gmail.com', ['your_account@gmail.com'],
```

`send_mail()` 方法的参数分别是邮件标题、邮件内容、发件人和收件人地址列表，最后一个参数 `fail_silently=False` 表示如果发送失败就抛出异常。如果看到返回1，就说明邮件成功发送。

如果采用以上设置无法成功使用Google的邮件服务，需要到 <https://myaccount.google.com/lesssecureapps>，启用“允许不够安全的应用”，如下图所示：

Some apps and devices use less secure sign-in technology, which makes your account more vulnerable. You can **turn off** access for these apps, which we recommend, or **turn on** access if you want to use them despite the risks. [Learn more](#)

Allow less secure apps: ON



现在我们把发送邮件的功能加入到视图中，编辑 `views.py` 中的 `post_share` 视图函数：

```
def post_share(request, post_id):
    # 通过id 获取 post 对象
    post = get_object_or_404(Post, id=post_id, status='published')
    sent = False

    if request.method == "POST":
        # 表单被提交
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # 表单字段通过验证
            cd = form.cleaned_data
            post_url = request.build_absolute_uri(post.get_absolute_url())
            subject = '{} ({}) recommends you reading "{}"'.format(cd['name'], cd['email'], post.title)
            message = 'Read "{}" at {}\n\n{}\'s comments:{}'.format(post.title, post_url, cd['name'], cd['comments'])
            send_mail(subject, message, 'lee0709@vip.sina.com', [cd['to']])
            sent = True

    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post, 'form': form, 'sent': sent})
```

声明了一个 `sent` 变量用于向模板返回邮件发送的状态，当邮件发送成功的时候设置为 `True`。稍后将使用该变量显示一条成功发送邮件的消息。由于要在邮件中包含连接，因此使用了 `get_absolute_url()` 方法获取被分享文章的URL，然后将其作为 `request.build_absolute_uri()` 的参数转为完整的URL，再加上表单数据创建邮件正文，最后将邮件发送给 `to` 字段中的收件人。

还需要给视图配置URL，打开 `blog` 应用中的 `urls.py`，加一条 `post_share` 的URL pattern：

```
urlpatterns = [
    # ...
    path('<int:post_id>/share/', views.post_share, name='post_share'),
]
```

1.4 在模板中渲染表单

在创建表单，视图和配置好URL之后，现在只剩下模板了。在 `blog/templates/blog/post/` 目录内创建 `share.html`，添加如下代码：

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

{% block title %}Share a post{% endblock %}

{% block content %}
    {% if sent %}
        <h1>E-mail successfully sent</h1>
        <p>
            "{{ post.title }}" was successfully sent to {{ form.cleaned_data.to }}.
        </p>
    {% else %}
```

```
<h1>Share "{{ post.title }}" by e-mail</h1>
<form action="." method="post">
{{ form.as_p }}
    {% csrf_token %}
    <input type="submit" value="Send e-mail">
</form>
{% endif %}
{% endblock %}
```

这个模板在邮件发送成功的时候显示一条成功信息，否则则显示表单。你可能注意到了，创建了一个HTML表单元素并且指定其通过POST请求提交：

```
<form action="." method="post">
```

之后渲染表单实例，通过使用 `as_p` 方法，将表单中的所有元素以`<p>`元素的方式展现出来。还可以使用 `as_ul` 和 `as_table` 分别以列表和表格的形式显示。如果想分别渲染每个表单元素，可以迭代表单对象中的每个元素，例如这样：

```
{% for field in form %}
<div>
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
</div>
{% endfor %}
```

`{% csrf_token %}` 在页面中显示为一个隐藏的`input`元素，是一个自动生成的防止跨站请求伪造 (CSRF) 攻击的 token。跨站请求伪造是一种冒充用户在已经登录的 Web 网站上执行非用户本意操作的一种攻击方式，可能由其他网站或一段程序发起。关于 CSRF 的更多信息可以参考 [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))。

例子生成的隐藏字段类似如下：

```
<input type='hidden' name='csrfmiddlewaretoken' value='26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR' />
```

Django默认会对所有 POST 请求进行CSRF检查，在所有 POST 方式提交的表单中，都要添加 csrf_token。

修改 blog/post/detail.html 将下列链接增加到 {{ post.body|linebreaks }} 之后：

```
<p>
    <a href="{% url "blog:post_share" post.id %}">Share this post</a>
</p>
```

这里的 {% url %} 标签，其功能和在视图中使用的 reverse() 方法类似，使用URL的命名空间 blog 和URL命名 post_share，再传入一个ID作为参数，就可以构建出一个URL。在页面渲染时， {% url %} 就会被渲染成反向解析出的 URL。

现在使用 python manage.py runserver 启动站点，打开 <http://127.0.0.1:8000/blog/>，点击任意文章查看详情页，在文章的正文下会出现分享链接，如下所示：

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

[Share this post](#)

My blog

This is my blog.

点击 Share this post 链接，可以看到分享页面：

Share "Notes on Duke Ellington" by e-mail

Name:

Email:

To:

Comments:

SEND E-MAIL

My blog

This is my blog.

这个表单的CSS样式表文件位于 [static/css/blog.css](#)。当你点击SEND E-MAIL按钮的时候，就会提交表单并验证数据，如果有错误，可以看到页面如下：

Share "Notes on Duke Ellington" by e-mail

Name:

Antonio

- Enter a valid email address.

Email:

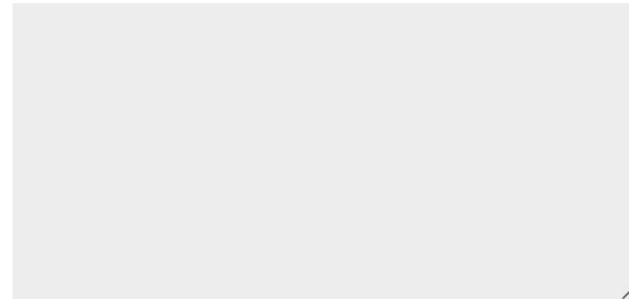
invalid

- This field is required.

To:



Comments:



SEND E-MAIL

My blog

This is my blog.

在某些现代浏览器上，很有可能浏览器会阻止你提交表单，提示必须完成某些字段，这是因为浏览器在提交根据表单的HTML元素属性先进行了验证。现在通过邮件分享链接的功能制作完成了，下一步是创建一个评论系统。

关闭浏览器验证的方法是给表单添加 `novalidate` 属性：`<form action="" novalidate>`

2 创建评论系统

现在，我们要创建一个评论系统，让用户可以对文章发表评论。创建评论系统需要进行以下步骤：

1. 创建一个模型用于存储评论
2. 创建一个表单用于提交评论和验证数据
3. 创建一个视图用于处理表单和将表单数据存入数据库
4. 编辑文章详情页以展示评论和提供增加新评论的表单

首先来创建评论对应的数据模型，编辑 `blog` 应用的 `models.py` 文件，添加以下代码：

```
class Comment(models.Model):  
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')  
    name = models.CharField(max_length=80)  
    email = models.EmailField()  
    body = models.TextField()  
    created = models.DateTimeField(auto_now_add=True)  
    updated = models.DateTimeField(auto_now=True)  
    active = models.BooleanField(default=True)  
  
    class Meta:  
        ordering = ("created",)  
  
    def __str__(self):  
        return 'Comment by {} on {}'.format(self.name, self.post)
```

`Comment` 模型包含一个外键（`ForeignKey`）用于将评论与一个文章联系起来，定义了文章和评论的一对多关系，即一个文章下边可以有多个评论；外键的 `related_name` 参数定义了在通过文章查找其评论的时候引用该关联关系的名称。这样定义了该外键之后，可以通过 `comment.post` 获得一条评论对应的文章，通过 `post.comments.all()` 获得一个文章

对应的所有评论。如果不定义 `related_name`，Django会使用模型的小写名加上 `_set` (`comment_set`) 来作为反向查询的管理器名称。

关于一对多关系的可以参考 https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_one/。

模型还包括一个 `active` 布尔类型字段，用于手工关闭不恰当的评论；还指定了排序方式为按照 `created` 字段进行排序。

新的 `Comment` 模型还没有与数据库同步，执行以下命令创建迁移文件：

```
python manage.py makemigrations blog
```

会看到如下输出：

```
Migrations for 'blog':  
  blog/migrations/0002_comment.py  
    - Create model Comment
```

Django在 `migrations/` 目录下创建了 `0002_comment.py` 文件，现在可以执行实际的迁移命令将模型写入数据库：

```
python manage.py migrate
```

会看到以下输出：

```
Applying blog.0002_comment... OK
```

数据迁移的过程结束了，数据库中新创建了名为 `blog_comment` 的数据表。

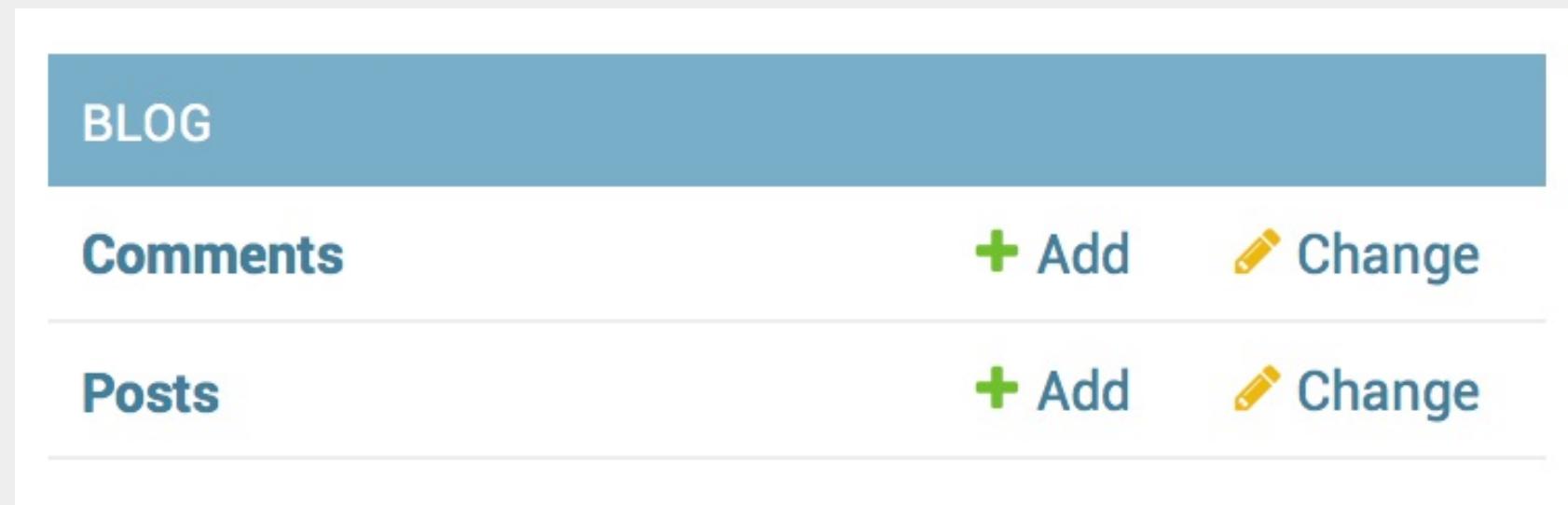
译者注：数据迁移的部分在原书中重复次数太多，而且大部分无实际意义，如无需要特殊说明的地方，以下翻译将略过类似的部分，以“执行数据迁移”或类似含义的字样替代。

创建了模型之后，可以将其加入管理后台。打开 `blog` 应用中的 `admin.py` 文件，导入 `Comment` 模型，然后增加如下代码：

```
from .models import Post, Comment

@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
    list_display = ('name', 'email', 'post', 'created', 'active')
    list_filter = ('active', 'created', 'updated')
    search_fields = ('name', 'email', 'body')
```

启动站点，到 <http://127.0.0.1:8000/admin/> 查看管理站点，会看到新的模型已经被加入到管理后台中：



2.1 根据模型创建表单

在发送邮件的功能里，采用继承 `forms.Form` 类的方式，自行编写各个字段创建了一个表单。Django对于表单有两个类：`Form` 和 `ModelForm`。这次我们使用 `ModelForm` 动态的根据 `Comment` 模型生成表单。编辑 `blog` 应用的 `forms.py` 文件：

```
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

依据模型创建表单，只需要在 `Meta` 类中指定基于哪个类即可。Django会自动内省该类然后创建对应的表单。我们对于模型字段的设置会影响到表单数据的验证规则。默认情况下，Django对每一个模型字段都创建一个对应的表单元素。然而，可以显示的通过 `Meta` 类中的 `fields` 属性指定需要创建表单元素的字段，或者使用`exclude`属性指定需要排除的字段。对于我们的 `CommentForm` 类，我们指定了表单只需要包含 `name`，`email` 和 `body` 字段即可。

2.2 在视图中处理表单

由于提交评论的动作在文章详情页发生，所以把处理表单的功能整合进文章详情视图中会让代码更简洁。编辑 `views.py` 文件，导入 `Comment` 模型然后修改 `post_detail` 视图：

```
from .models import Post, Comment
from .forms import EmailPostForm, CommentForm

def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post, status="published", publish__year=year, publish__month=month, publish__day=day)
    # 列出文章对应的所有活动的评论
```

```
comments = post.comments.filter(active=True)

new_comment = None

if request.method == "POST":
    comment_form = CommentForm(data=request.POST)
    if comment_form.is_valid():
        # 通过表单直接创建新数据对象，但是不要保存到数据库中
        new_comment = comment_form.save(commit=False)
        # 设置外键为当前文章
        new_comment.post = post
        # 将评论数据对象写入数据库
        new_comment.save()
else:
    comment_form = CommentForm()
return render(request, 'blog/post/detail.html',
             {'post': post, 'comments': comments, 'new_comment': new_comment, 'comment_form': comment_form})
```

现在post_detail视图可以显示文章及其评论，在视图中增加了一个获得当前文章对应的全部评论的QuerySet，如下：

```
comments = post.comments.filter(active=True)
```

以 Comments 类中定义的外键的 related_name 属性的名称作为管理器，对 post 对象执行查询从而得到了所需的 QuerySet。

同时还为这个视图增加了新增评论的功能。初始化了一个 new_comment 变量为 None，用于标记一个新评论是否被创建。如果是 GET 请求，使用 comment_form = CommentForm() 创建空白表单；如果是 POST 请求，使用提交的数据生成表单对象并调用 is_valid() 方法进行验证。如果表单未通过验证，使用当前表单渲染页面以提供错误信息。如果表单通过验证，则进行如下工作：

1. 调用当前表单的 `save()` 方法生成一个 `Comment` 实例并且赋给 `new_comment` 变量，就是下边这一行：

```
new_comment = comment_form.save(commit=False)
```

表单对象的 `save()` 方法会返回一个由当前数据构成的，表单关联的数据类的对象，并且会将这个对象写入数据库。如果指定 `commit=False`，则数据对象会被创建但不会被写入数据库，便于在保存到数据库之前对对象进行一些操作。

2. 将 `comment` 对象的外键关联指定为当前文章： `new_comment.post = post`，这样就明确了当前的评论是属于这篇文章的。

3. 最后，调用 `save()` 方法将新的评论对象写入数据库： `new_comment.save()`

注意 `save()` 方法仅对 `ModelForm` 生效，因为 `Form` 类没有关联到任何数据模型。

2.3 为文章详情页面添加评论

已经创建了用于管理一个文章的评论的视图，现在需要修改 `post/detail.html` 来做以下的事情：

- 展示当前文章的评论总数
- 列出所有评论
- 展示表单供用户添加新评论

首先要增加评论总数，编辑 `post/detail.html`，将下列内容追加在 `content` 块内的底部：

```
{% with comments.count as total_comments %}  
  <h2>  
    {{ total_comments }} comment{{ total_comments|pluralize }}  
  </h2>  
{% endwith %}
```

我们在模板里使用了Django ORM，执行了 `comments.count()`。在模板中执行一个对象的方法时，不需要加括号；也正因为如此，不能够执行必须带有参数的方法。`{% with %}` 标签表示在 `{% endwith %}` 结束之前，都可以使用一个变量来代替另外一个变量或者值。

`{% with %}` 标签经常用于避免反复对数据库进行查询和向模板传入过多变量。

这里使用了 `pluralize` 模板过滤器，用于根据 `total_comments` 的值显示复数词尾。将在下一章详细讨论模板过滤器。

如果值大于1，`pluralize` 过滤器会返回一个带复数词尾"s"的字符串，实际渲染出的字符串会是 `0 comments`，`1 comment`，`2 comments` 或者 `N comments`。

然后来增加评论列表的部分，在 `post/detail.html` 中上述代码之后继续追加：

```
{% for comment in comments %}
    <div class="comment">
        <p class="info">
            Comment {{ forloop.counter }} by {{ comment.name }}
            {{ comment.created }}
        </p>
        {{ comment.body|linebreaks }}
    </div>
{% empty %}
    <p>There are no comments yet.</p>
{% endfor %}
```

这里使用了 `{% for %}` 标签，用于循环所有的评论数据对象。如果 `comments` 对象为空，则显示一条信息提示用户没有评论。使用 `{{ forloop.counter }}` 可以在循环中计数。然后，显示该条评论的发布者，发布时间和评论内容。

最后是显示表单或者一条成功信息的部分，在上述代码后继续追加：

```
{% if new_comment %}
    <h2>Your comment has been added.</h2>
{% else %}
    <h2>Add a new comment</h2>
    <form action="." method="post">
        {{ comment_form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Add comment"></p>
    </form>
{% endif %}
```

这段代码的逻辑很直白：如果new_comment对象存在，显示一条成功信息，其他情况下则用 `as_p` 方法渲染整个表单以及CSRF token。在浏览器中打开 <http://127.0.0.1:8000/blog/>，可以看到如下页面：

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

[Share this post](#)

0 comments

There are no comments yet.

Add a new comment

Name:

My blog

This is my blog.

Email:

Body:

ADD COMMENT

使用表单添加一些评论，然后刷新页面，应该可以看到评论以发布的时间排序：

2 comments

Comment 1 by Antonio Dec. 14, 2017, 10:08 p.m.

It's very interesting.

Comment 2 by Bienvenida Dec. 14, 2017, 10:09 p.m.

I didn't know that.

在浏览器中打开 <http://127.0.0.1:8000/admin/blog/comment/>，可以在管理后台中看到所有评论，点击其中的一个进行编辑，取消掉Active字段的勾，然后点击SAVE按钮。然后会跳回评论列表，Active栏会显示一个红色叉号表示该评论未被激活，如下图所示：

Select comment to change

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	user1@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.	X
<input type="checkbox"/>	Bienvenida	user2@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.	✓

2 comments

此时返回文章详情页，可以看到被设置为未激活的评论不会显示出来，也不会被统计到评论总数中。由于有了这个 `active` 字段，可以非常方便的控制评论显示与否而不需要实际删除。

3 添加标签功能

在完成了评论系统之后，我们将来给文章加上标签系统。标签系统通过集成 `django-taggit` 第三方应用模块到我们的 Django 项目来实现，`django-taggit` 提供一个 `Tag` 数据模型和一个管理器，可以方便的给任何模型加上标签。`django-taggit` 的源代码位于：<https://github.com/alex/django-taggit>。

通过 `pip` 安装 `django-taggit`：

```
pip install django_taggit==0.22.2
```

译者注：如果安装了 `django 2.1` 或更新版本，请下载最新版 `django-taggit`。原书的 `0.22.2` 版只能和 `Django 2.0.5` 版搭配使用。新版使用方法与 `0.22.2` 版没有任何区别。

之后在 `setting.py` 里的 `INSTALLED_APPS` 设置中增加 `taggit` 以激活该应用：

```
INSTALLED_APPS = [
    # ...
    'blog.apps.BlogConfig',
    'taggit',
]
```

打开 `blog` 应用下的 `models.py` 文件，将 `django-taggit` 提供的 `TaggableManager` 模型管理器加入到 `Post` 模型中：

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # .....
    tags=TaggableManager()
```

这个管理器可以对 `Post` 对象的标签进行增删改查。然后执行数据迁移。

现在数据库也已经同步完成了，先学习一下如何使用 `django-taggit` 模块和其 `tags` 管理器。使用 `python manage.py shell` 进入Python命令行然后输入下列命令：

之后来看如何使用，先到命令行里：

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

然后给这个文章增加一些标签，然后再获取这些标签看一下是否添加成功：

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>, <Tag: django>]>
```

删除一个标签再检查标签列表：

```
>>> post.tags.remove('django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>]>
```

操作很简单。启动站点然后到 <http://127.0.0.1:8000/admin/taggit/tag/>，可以看到列出 taggit 应用中 Tag 对象的管理页面：

Django administration

WELCOME, **ADMIN**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Taggit > Tags

Select Tag to change

[ADD TAG](#) +



Search

Action: -----



Go

0 of 3 selected

<input type="checkbox"/>	NAME	SLUG	<input type="checkbox"/>
<input type="checkbox"/>	django	django	<input type="checkbox"/>
<input type="checkbox"/>	jazz	jazz	<input type="checkbox"/>
<input type="checkbox"/>	music	music	<input type="checkbox"/>

3 Tags

到 <http://127.0.0.1:8000/admin/blog/post/> 点击一篇文章进行修改，可以看到文章现在包含了一个标签字段，如下所示：

Tags:

jazz, music

A comma-separated list of tags.

现在还需要在页面上展示标签，编辑 `blog/post/list.html`，在显示文章的标题下边添加：

```
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
```

`join` 过滤器的功能和Python字符串的 `join()` 方法很类似，打开 <http://127.0.0.1:8000/blog/>，就可以看到在每个文章的标题下方列出了标签：

Who was Django Reinhardt?

Tags: jazz, music

Published Dec. 14, 2017, 8:54 a.m. by admin

现在来编辑 `post_list` 视图，让用户可以根据一个标签列出具备该标签的所有文章，打开 `blog` 应用的 `views.py` 文件，从 `django-taggit` 中导入 `Tag` 模型，然后修改 `post_list` 视图，让其可以额外的通过标签来过滤文章：

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    tag = None
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])
    paginator = Paginator(object_list, 3) # 3 posts in each page
    # .....
```

post_list视图现在工作如下：

1. 多接收一个 `tag_slug` 参数，默认值为 `None`。这个参数将通过URL传入
2. 在视图中，创建了初始的QuerySet用于获取所有的已发布的文章，然后判断如果传入了 `tag_slug`，就通过 `get_object_or_404()` 方法获取对应的 `Tag` 对象
3. 然后过滤初始的QuerySet，条件为文章的标签中包含选出的 `Tag` 对象，由于这是一个多对多关系，所以将 `Tag` 对象放入一个列表内选择。

QuerySet是惰性的，直到模板渲染过程中迭代 `posts` 对象列表的时候，QuerySet才被求值。

最后修改，视图底部的 `render()` 方法，把 `tag` 变量也传入模板。完整的视图如下：

```
def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])

    paginator = Paginator(object_list, 3) # 3 posts in each page
```

```
page = request.GET.get('page')
try:
    posts = paginator.page(page)
except PageNotAnInteger:
    posts = paginator.page(1)
except EmptyPage:
    posts = paginator.page(paginator.num_pages)

return render(request, 'blog/post/list.html', {'page': page, 'posts': posts, 'tag': tag})
```

打开 blog 应用的 `urls.py` 文件，注释掉 `PostListView` 那一行，取消 `post_list` 视图的注释，像下边这样：

```
path('', views.post_list, name='post_list'),
# path('', views.PostListView.as_view(), name='post_list'),
```

再增加一行通过标签显示文章的URL：

```
path('tag/<slug:tag_slug>', views.post_list, name='post_list_by_tag'),
```

可以看到，两个URL指向了同一个视图，但命名不同。第一个URL不带任何参数去调用 `post_list` 视图，第二个URL则会带上 `tag_slug` 参数调用 `post_list` 视图。使用了一个 `<slug:tag_slug>` 获取参数。

由于我们将CBV改回为FBV，所以在 `blog/post/list.html` 里将 `include` 语句的变量改回FBV的 `posts`：

```
{% include "pagination.html" with page=posts %}
```

再增加显示文章标签的 `{% for %}` 循环的代码：

```
{% if tag %}  
    <h2>Posts tagged with "{{ tag.name }}"</h2>  
{% endif %}
```

如果用户访问博客，可以看到全部的文章列表；如果用户点击某个具体标签，就可以看到具备该标签的文章。现在还需改变一下标签的显示方式：

```
<p class="tag">  
    Tags:  
    {% for tag in post.tags.all %}  
        <a href="{% url "blog:post_list_by_tag" tag.slug %}">{{ tag.name }}</a>  
    {% if not forloop.last %}, {% endif %}  
    {% endfor %}  
</p>
```

我们通过迭代所有标签，将标签设置为一个链接，指向通过该标签对应的所有文章。通过`{% url "blog:post_list_by_tag" tag.slug %}`反向解析出了链接。

现在到<http://127.0.0.1:8000/blog/>，然后点击任何标签，就可以看到该标签对应的文章列表：



Who was Django Reinhardt?

Tags: [jazz](#) , [music](#)

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

Page 1 of 1.

4 通过相似性获取文章

在为博客添加了标签功能之后，可以使用标签来做一些有趣的事情。一些相同主题的文章会具有相同的标签，可以创建一个功能给用户按照共同标签数量的多少推荐文章。

为了实现该功能，需要如下几步：

1. 获得当前文章的所有标签

2. 拿到所有具备这些标签的文章
3. 把当前文章从这个文章列表里去掉以避免重复显示
4. 按照具有相同标签的多少来排列
5. 如果文章具有相同数量的标签，按照时间来排列
6. 限制总推荐文章数目

这几个步骤会使用到更复杂的QuerySet，打开 `blog` 应用的 `views.py` 文件，在最上边增加一行：

```
from django.db.models import Count
```

这是从Django ORM中导入的 `Count` 聚合函数，这个函数可以按照分组统计某个字段的数量，`django.db.models` 还包含下列聚合函数：

- `Avg`：计算平均值
- `Max`：取最大值
- `Min`：取最小值
- `Count`：计数
- `Sum`：求和

译者注：作者在此处有所保留，没有写 `Sum` 函数，此外还有Q查询

聚合查询的官方文档在 <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>。

修改 `post_detail` 视图，在 `render()` 上边加上这一段内容，缩进与 `render()` 行同级：

```
def post_detail(request, year, month, day, post):  
    # .....  
    # 显示相近Tag的文章列表
```

```
post_tags_ids = post.tags.values_list('id', flat=True)
similar_tags = Post.published.filter(tags__in=post_tags_ids).exclude(id=post.id)
similar_posts = similar_tags.annotate(same_tags=Count('tags')).order_by('-same_tags', '-publish')[:4]
return render(.....
```

以上代码解释如下：

1. `values_list` 方法返回指定的字段的值构成的元组，通过指定 `flat=True`，让其结果变成一个列表比如 [1, 2, 3, ...]
2. 选出所有包含上述标签的文章并且排除当前文章
3. 使用 `Count` 对每个文章按照标签计数，并生成一个新字段 `same_tags` 用于存放计数的结果
4. 按照相同标签的数量，降序排列结果，然后截取前四个结果作为最终传入模板的数据对象。

最后修改 `render()` 函数将新生成的 `similar_posts` 传给模板：

```
return render(request,
    'blog/post/detail.html',
    {'post': post,
     'comments': comments,
     'new_comment': new_comment,
     'comment_form': comment_form,
     'similar_posts': similar_posts})
```

然后编辑 `blog/post/detail.html`，将以下代码添加到评论列表之前：

```
<h2>Similar posts</h2>
{% for post in similar_posts %}
<p>
    <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
</p>
{% empty %}
```

```
There are no similar posts yet.  
{% endfor %}
```

现在的文章详情页面示例如下：

Who was Django Reinhardt?

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

[Share this post](#)

Similar posts

[Miles Davis favourite songs](#)

[Notes on Duke Ellington](#)

现在已经实现了该功能。`django-taggit` 模块包含一个 `similar_objects()` 模型管理器也可以实现这个功能，可以在 <https://django-taggit.readthedocs.io/en/latest/api.html> 查看 `django-taggit` 的所有模型管理器使用方法。

还可以用同样的方法在文章详情页为文章添加标签显示。

总结

在这一章里了解了如何使用Django的表单和模型表单，为博客添加了通过邮件分享文章的功能和评论系统。第一次使用了Django的第三方应用，通过 `django-taggit` 为博客增添了基于标签的功能。最后进行复杂的聚合查询实现了通过标签相似性推荐文章。

下一章会学习如何创建自定义的模板标签和模板过滤器，创建站点地图和RSS feed，以及对博客文章实现全文检索功能。

第三章 扩展博客功能

在上一章里，使用了基本的表单提交数据与处理表单，进行复杂的分组查询，还学习了集成第三方应用。

这一章涵盖如下的内容：

- 自定义模板标签和模板过滤器
- 给站点增加站点地图和订阅功能
- 使用PostgreSQL数据库实现全文搜索

1 自定义模板标签和过滤器

Django提供很多内置的模板标签和过滤器，例如`{% if %}`或`{% block %}`，在之前已经在模板中使用过它们。关于完整的内置模板标签和过滤器可以看<https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>。

Django也允许你创建自己的模板标签，用来在页面中进行各种操作。当你想在模板中实现Django没有提供的功能时，自定义模板标签是一个好的选择。

1.1 自定义模板标签

Django提供下边的两个函数可以简单快速地创建自定义模板标签：

- `simple_tag`: 处理数据并且返回字符串
- `inclusion_tag`: 处理数据并返回一个渲染的模板

所有的自定义标签，只能够在模板中使用。

在 `blog` 应用目录里新建一个目录 `templatetags`，然后在其中创建一个空白的 `__init__.py`，再创建一个文件 `blog_tags.py`，文件结构如下：

```
blog/
  __init__.py
  models.py
  ...
  templatetags/
    __init__.py
    blog_tags.py
```

注意这里的命名很关键，一会在模板内载入自定义标签的时候就需要使用这个包的名称（`templatetags`）。

先创建一个简单的标签，在刚刚创建的 `blog_tags.py` 里写如下代码：

```
from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def total_posts():
    return Post.published.count()
```

我们创建了一个标签返回已经发布的文章总数。每个模板标签的模块内需要一个 `register` 变量，是 `template.Library` 的实例，用于注册自定义的标签。然后创建了一个Python函数 `total_posts`，用 `@register.simple_tag` 装饰器将其注册为一个简单标签。Django会使用这个函数的名称作为标签名称，如果想使用其他的名称，可以通过 `name` 属性指定，例如 `@register.simple_tag(name='my_tag')`。

在添加了新的自定义模板标签或过滤器之后，必须重新启动django服务才能在模板中生效。

在模板内使用自定义标签之前，需要使用 `{% load %}` 在模板中引入自定义的标签，像之前提到的那样，使用创建的名字作为 `load` 的参数。

打开 `blog/templates/base.html` 模板，在最上边添加 `{% load blog_tags %}`，然后使用自定义标签 `{% total_posts %}`，这个模板最后看起来像这样：

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}{% endblock %}</title>
        <link href="{% static "css/blog.css" %}" rel="stylesheet">
    </head>
    <body>
        <div id="content">
            {% block content %}
            {% endblock %}
        </div>
        <div id="sidebar">
            <h2>My blog</h2>
            <p>This is my blog. I've written {% total_posts %} posts so far.</p>
        </div>
    </body>
</html>
```

启动站点然后到 `http://127.0.0.1:8000/blog/`，应该可以看到总文章数被显示在了侧边栏：

My blog

This is my blog. I've written 4 posts so far.

自定义标签威力强大之处在于不通过视图就可以处理数据和添加到模板中。

现在再来创建一个用于在侧边栏显示最新发布的文章的自定义标签。这次通过 `inclusion_tag` 渲染一段HTML代码。编辑 `blog_tags.py` 文件，添加如下内容：

```
@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}
```

在上边的代码里，使用 `@register.inclusion_tag` 装饰器装饰了自定义函数 `show_latest_posts`，同时指定了要渲染的模板为 `blog/post/latest_posts.html`。我们的模板标签还接受一个参数 `count`，通过 `Post.published.order_by('-publish')[:count]` 切片得到指定数量的最新发布的文章。注意这个自定义函数返回的是一个字典对象而不是一个具体的值。`inclusion_tag` 必须返回一个类似给模板传入变量的字典，用于在 `blog/post/latest_posts.html` 中取得数据并渲染模板。刚刚创建的这一切在模板中以类似 `{% show_latest_posts 3 %}` 的形式来使用。

那么在模板里如何使用呢，这是一个带参数的tag，就像之前使用内置的那样，在标签后边加参数：
 `{% show_latest_posts 3 %}`

在 `blog/post/` 目录下创建 `latest_posts.html` 文件，添加下列代码：

```
<ul>
  {% for post in latest_posts %}
    <li>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </li>
  {% endfor %}
</ul>
```

在上边的代码里，使用 `latest_posts` 显示出了一个未排序的最新文章列表。然后编辑 `blog/base.html`，加入新标签以显示最新的三篇文章：

```
<div id="sidebar">
  <h2>My blog</h2>
  <p>This is my blog. I've written {% total_posts %} posts so far.</p>
  <h3>Latest posts</h3>
  {% show_latest_posts 3 %}
</div>
```

模板中调用了自定义标签，然后传入了一个参数3，之后这个标签的位置会被替换成被渲染的模板。

现在返回浏览器，刷新页面，可以看到侧边栏显示如下：

My blog

This is my blog. I've written 4 posts so far.

Latest posts

- [Miles Davis favourite songs](#)
- [Notes on Duke Ellington](#)
- [Another post](#)

最后再来创建一个 `simple_tag`，将数据存放在这个标签内，而不是像我们创建的第一个标签一样直接展示出来。我们使用这种方法来显示评论最多的文章。编辑 `blog_tags.py`，添加下列代码：

```
from django.db.models import Count  
  
@register.simple_tag
```

```
def get_most_commented_posts(count=5):
    return Post.published.annotate(total_comments=Count('comments')).order_by('-total_comments')[:count]
```

在上边的代码里，使用 `annotate`，对每篇文章的评论进行计数然后按照 `total_comments` 字段降序排列，之后使用 `[:count]` 切片得到评论数量最多的特定篇文章，

除了 `Count` 之外，Django提供了其他聚合函数 `Avg`，`Max`，`Min` 和 `Sum`，聚合函数的详情可以查看 <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>。

编辑 `blog/base.html` 把以下代码追加到侧边栏 `<div>` 元素内部：

```
<h3>Most commented posts</h3>
{% get_most_commented_posts as most_commented_posts %}
<ul>
    {% for post in most_commented_posts %}
        <li>
            <a href="{{ post.get_absolute_url }}>{{ post.title }}</a>
        </li>
    {% endfor %}
</ul>
```

在这里使用了 `as` 将我们的模板标签保存在一个叫做 `most_commented_posts` 变量中，然后展示其中的内容。

现在打开浏览器刷新页面，可以看到新的页面如下：

The screenshot shows a Django application running at `127.0.0.1:8000/blog/`. The main content area displays three blog posts:

- Miles Davis favourite songs**: Published on Dec. 14, 2017, 10:01 p.m. by admin. Content: Miles Dewey Davis III was an American jazz trumpeter, bandleader, and composer.
- Notes on Duke Ellington**: Published on Dec. 14, 2017, 9:58 p.m. by admin. Content: Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.
- Another post**: Published on Dec. 14, 2017, 11:45 a.m. by admin. Content: Post body.

The sidebar on the right contains:

- My blog**: This is my blog. I've written 4 posts so far.
- Latest posts**:
 - Miles Davis favourite songs
 - Notes on Duke Ellington
 - Another post
- Most commented posts**:
 - Notes on Duke Ellington
 - Who was Django Reinhardt?
 - Another post
 - Miles Davis favourite songs

关于自定义模板标签的详情可以查看 <https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/>。

1.2 自定义模板过滤器

Django内置很多模板过滤器用于在模板内修改变量。模板过滤器实际上是Python函数，接受1或2个参数-其中第一个参数是变量，第二个参数是一个可选的变量，然后返回一个可供其他模板过滤器操作的值。一个模板过滤器类似这样：
`{{ variable|my_filter }}`，带参数的模板过滤器类似：`{{ variable|my_filter:"foo" }}`，可以连用过滤器，例如：`{{ variable|filter1|filter2 }}`。

我们来通过自定义过滤器使我们的博客文章可以支持Markdown语法，然后将其转换成对应的HTML格式。Markdown是一种易于使用的轻型标记语言而且可以方便的转为HTML。可以在[这里查看Markdown语法的详情](https://daringfireball.net/projects/markdown/basics)：
<https://daringfireball.net/projects/markdown/basics>。

先通过pip安装Python的 Markdown 模块：

```
pip install Markdown==2.6.11
```

然后编辑 `blog_tags.py`，添加如下内容：

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

我们使用和模板标签类似的方式注册了模板过滤器，为了不使我们的函数和 `markdown` 模块重名，将我们的函数命名为 `markdown_format`，但是指定了模板中的标签名称为 `markdown`，这样就可以通过 `{{ variable|markdown }}` 来使用标签了。`mark_safe` 用来告诉Django该段HTML代码是安全的，可以将其渲染到最终页面中。默认情况下，Django对于生成的HTML代码都会进行转义而不会当成HTML代码解析，只有对 `mark_safe` 标记的内容才会正常解析，这是为了避免在页面中出现危险代码（如添加外部JavaScript文件的代码）。

然后在 `blog/post/list.html` 和 `blog/post/detail.html` 中的 `{% extends %}` 之后引入自定义模板的模块：

```
{% load blog_tags %}
```

在 `post/detail.html` 中，找到下边这行：

```
{% post.body|linebreaks %}
```

将其替换成：

```
{% post.body|markdown %}
```

然后在 `post/list.html` 中，找到下边这行：

```
{% post.body|truncatewords:30|linebreaks %}
```

将其替换成：

```
{% post.body|markdown|truncatewords_html:30 %}
```

`truncatewords_html` 过滤器不会截断未闭合的HTML标签。

浏览器中打开 <http://127.0.0.1:8000/admin/blog/post/add/> 然后写一段使用Markdown语法的正文：

```
This is a post formatted with markdown
-----
*This is emphasized* and **this is more emphasized**.
Here is a list:

* One
* Two
* Three

And a [link to the Django website](https://www.djangoproject.com/)
```

然后在浏览器中查看刚添加的文章，可以看到如下的结果：

Markdown post

Published Dec. 15, 2017, 8:42 a.m. by admin

This is a post formatted with markdown

This is emphasized and this is more emphasized.

Here is a list:

- One
- Two
- Three

And a [link to the Django website](#)

可以看到，自定义模板过滤器在需要自定义格式的时候非常好用。可在

<https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/#writing-custom-template-filters> 找到更多关于自定义过滤器的信息。

2 创建站点地图

Django带有站点地图功能框架，可以根据网站内容动态的生成站点地图。站点地图是一个XML文件，用于给搜索引擎提供信息，可以帮助搜索引擎爬虫索引站点的内容。

Django的站点地图框架是 `django.contrib.sites`。如果使用同一个Django项目运行多个站点，站点地图功能允许为每个站点创建单独的站点地图。为了使用站点地图功能，需要启用 `django.contrib.sites` 和 `jango.contrib.sitemaps`，将这两个应用添加到 `settings.py` 的 `INSTALLED_APPS` 设置中：

```
SITE_ID = 1
# Application definition
INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    'django.contrib.sitemaps',
]
```

由于添加了新应用，需要执行数据迁移。迁移完成之后，在 `blog` 应用目录内创建 `sitemaps.py` 文件并添加如下代码：

```
from django.contrib.sitemaps import Sitemap
from .models import Post


class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9

    def items(self):
        return Post.published.all()
```

```
def lastmod(self, obj):
    return obj.updated
```

我们通过继承 `django.contrib.sitemaps` 的 `Sitemap` 类创建了一个站点地图对象。`changefreq` 和 `priority` 属性表示文章页面更新的频率和这些文章与站点的相关性（最大相关性为1）。使用 `items()` 方法返回这个站点地图所需的 `QuerySet`，Django默认会调用数据对象的 `get_absolute_url()` 获取对应的URL，如果想手工指定具体的URL，可以为 `PostSitemap` 添加一个 `location` 方法。`lastmod` 方法接收 `items()` 返回的每一个数据对象然后返回其更新时间。`changefreq` 和 `priority` 可以通过定义方法也可以作为属性名进行设置。站点地图的详细使用可以看官方文档：
<https://docs.djangoproject.com/en/2.0/ref/contrib/sitemaps/>。

最后就是配置站点地图对应的URL，打开项目的根 `urls.py`，添加如下代码：

```
from django.urls import path, include
from django.contrib import admin
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = {'posts': PostSitemap,}

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps}, name='django.contrib.sitemaps.views.sitemap')
]
```

在上述代码中，导入了需要的库并且定义了一个站点地图的字典。将 `sitemap.xml` 的路径匹配到 `sitemap` 视图。`sitemaps` 字典会被传递给 `sitemap` 视图。现在启动站点然后在浏览器中打开 <http://127.0.0.1:8000/sitemap.xml>，可以看到如下的输出：

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
<url>
    <loc>http://example.com/blog/2017/12/15/markdown-post/</loc>
    <lastmod>2017-12-15</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
</url>
<url>
    <loc>
        http://example.com/blog/2017/12/14/who-was-django-reinhardt/
    </loc>
    <lastmod>2017-12-14</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
</url>
</urlset>
```

其中的每个文章的URL都是由 `get_absolute_url()` 方法生成的。`lastmod` 标签的内容是最后更新的时间，和在类中定义的一样。`changefreq` 和 `priority` 标签也包含对应的值。可以看到站点名为 `example.com`，这个名称来自于数据库存储的 `Site` 对象，这是我们在为站点地图应用进行数据迁移的时候默认生成的一个对象。打开 <http://127.0.0.1:8000/admin/sites/site/>，可以看到类似下边的界面：

Select site to change

[ADD SITE +](#) Action: 0 of 1 selected

DOMAIN NAME

DISPLAY NAME



example.com

example.com

1 site

上边的截图里包含刚才使用的的主机名，可以修改成自己想要的主机名。可以将其修改成 `localhost:8000` 以使用本地地址生成URL。如下图所示：

Change site

Domain name:

localhost:8000

Display name:

localhost:8000

设置之后，URL就会使用本地地址。在生产环境中，需要在此处设置正常的主机和站点名。

3 创建订阅功能

Django内置一些功能，采用和创建站点地图类似的方法为站点增加RSS或者Atom订阅信息。订阅信息是一个特定的数据格式，通常是XML文件，用于向用户提供这个网站的更新数据，用户通过一个订阅代理程序，订阅这个网站的feed，就可以接收到新的内容通知。

在 `blog` 应用目录下新建 `feeds.py` 文件并添加如下代码：

```
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords
from .models import Post

class LatestPostFeed(Feed):
```

```
title = 'My blog'
link = '/blog/'
description = 'New posts of my blog.'

def items(self):
    return Post.published.all()[:5]

def item_title(self, item):
    return item.title

def item_description(self, item):
    return truncatewords(item.body, 30)
```

译者注：`item_description(self, item)` 这个函数并没有对 `post.body` 进行处理，所以会返回未经处理的 markdown 代码，在不支持 markdown 的 Feed 阅读器里会出现问题，读者可以修改该函数，调用 markdown 库输出转换后的字符串。

这段代码首先继承了内置的 `Feed` 类，`title`，`link` 和 `description` 属性分别对应 XML 文件的 `<title>`、`<link>` 和 `<description>` 标签。

`items()` 方法用于获得订阅信息要使用的数据对象，这里只取了最新发布的 5 篇文章。`item_title()` 和 `item_description()` 方法接收每一个数据对象并且分别返回标题和正文前 30 个字符。

现在为配置订阅路径，编辑 `blog/urls.py`，导入 `LatestPostsFeed` 然后配置一条新路由：

```
from .feeds import LatestPostsFeed

urlpatterns = [
    # ...
    path('feed/', LatestPostsFeed(), name='post_feed'),
]
```

打开地址 <http://127.0.0.1:8000/blog/feed/> 即可看到feed内容，包含最新的5篇文章：

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
<channel>
    <title>My blog</title>
    <link>http://localhost:8000/blog/</link>
    <description>New posts of my blog.</description>
    <atom:link href="http://localhost:8000/blog/feed/" rel="self"/>
    <language>en-us</language>
    <lastBuildDate>Fri, 15 Dec 2017 09:56:40 +0000</lastBuildDate>
    <item>
        <title>Who was Django Reinhardt?</title>
        <link>http://localhost:8000/blog/2017/12/14/who-was-djangoreinhardt/</
        link>
        <description>Who was Django Reinhardt.</description>
        <guid>http://localhost:8000/blog/2017/12/14/who-was-djangoreinhardt/</
        guid>
    </item>
    ...
</channel>
</rss>
```

如果用一个RSS阅读器打开这个链接，就可以在其界面里看到对应信息。

最后一步是在侧边栏添加订阅本博客的链接，在 `blog/base.html` 里的侧边栏`<div>`里追加：

```
<p><a href='{% url "blog:post_feed" %}'>Subscribe to my RSS feed</a></p>
```

之后就可以在 <http://127.0.0.1:8000/blog/> 看到订阅链接，类似下图：

My blog

This is my blog. I've written 5 posts so far.

[Subscribe to my RSS feed](#)

4 增加全文搜索功能

现在可以为博客添加搜索功能。Django ORM可以使用 `contains` 或类似的 `icontains` 过滤器执行简单的匹配任务。比如：

```
from blog.models import Post
Post.objects.filter(body__contains='framework')
```

然而，如果要执行更加复杂的搜索，比如通过权重或者相似性，就必须使用一个 [全文搜索引擎 \(full-text search engine\)](#)。

Django的全文检索功能基于PostgreSQL数据库的全文搜索特性，所以这个全文检索功能不能用于Django ORM支持的其他种类的数据库。PostgreSQL的全文搜索介绍在 <https://www.postgresql.org/docs/10/static/textsearch.html>。

虽然Django ORM通过面向对象抽象，可以不依赖于具体的数据库，但是用于PostgreSQL的一部分功能无法用于其他数据库。

4.1 自定义模板过滤器

现在 `blog` 项目使用的是Python自带的SQLite数据库，对于开发而言已经足够。在生产环境中，需要使用诸如MySQL，PostgreSQL和Oracle等更强力的数据库。为了实现全文搜索功能，我们将转而使用PostgreSQL。

在Linux环境下，需要先安装PostgreSQL和Python的相关依赖：

```
sudo apt-get install libpq-dev python-dev
```

之后使用下列命令安装PostgreSQL：

```
sudo apt-get install postgresql postgresql-contrib
```

如果使用MacOS X或者Windows，到 <https://www.postgresql.org/download/> 查看安装说明。

在安装完之后，还需要为python安装 `psycopg2` 模块：

```
pip install psycopg2==2.7.4
```

译者注：使用 `pip install psycopg2-binary` 命令安装 `psycopg2` 最新版模块。

在PostgreSQL中创建一个名叫 `blog` 的用户，供项目使用。在系统命令行中输入下列命令：

```
su postgres  
createuser -dP blog
```

会被提示输入密码。创建用户成功之后，创建一个名叫 `blog` 的数据库并将所有权设置给 `blog` 用户：

```
createdb -E utf8 -U blog blog
```

译者注：PostgreSQL在Linux安装后会创建一个`postgres`用户，使用该用户身份可以登陆PostgreSQL数据库进行操作。

之后编辑 `settings.py` 文件中的 `DATABASES` 设置：

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'blog',  
        'USER': 'blog',  
        'PASSWORD': '*****',  
    }  
}
```

这里我们将默认的数据库修改成了PostgreSQL，之后执行数据迁移和创建超级用户。然后可以在 <http://127.0.0.1:8000/admin/> 登录管理后台。由于更换了数据库，博客应用内没有任何文章数据，录入一些数据，为之后使用全文搜索做准备。

4.2 执行简单搜索

编辑 `settings.py` 文件，将 `django.contrib.postgres` 加入到 `INSTALLED_APPS` 中：

```
INSTALLED_APPS = [
    # ...
    'django.contrib.postgres',
]
```

激活还应用后，现在可以通过search参数进行搜索：

```
from blog.models import Post
Post.objects.filter(body__search='django')
```

这个QuerySet会使用PostgreSQL为 `body` 字段创建内容是'django'字符串的搜索向量和一个查询，通过匹配查询和结果向量，返回最后的结果。

4.3 执行简单搜索

可能想在多个字段中进行检索。在这种情况下，需要定义一个 `SearchVector` 搜索向量对象，来创建一个针对 `Post` 模型的 `title` 和 `body` 进行搜索的向量：

```
from django.contrib.postgres.search import SearchVector
from blog.models import Post

Post.objects.annotate(search=SearchVector('title', 'body')).filter(search='poem')
```

使用分组函数然后定义了两个字段的向量，之后使用查询，就可以得到最终的结果。

全文搜索是一个密集计算过程，如果要检索的数据多于几百行，最好创建一个匹配搜索向量的索引，Django提供了一个 `SearchVectorField` 字段在模型中定义搜索向量。具体可以参考
<https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/#performance>。

4.4 创建搜索视图

现在我们可以创建一个视图用于让用户执行搜索。首先需要一个表单让用户输入要查询的数据，编辑 `blog` 应用的 `forms.py` 增加下面的表单：

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

`query` 字段用于输入查询内容，然后编辑 `blog` 应用的 `views.py` 文件，然后添加如下代码：

```
from django.contrib.postgres.search import SearchVector
from .forms import EmailPostForm, CommentForm, SearchForm

def post_search(request):
    form = SearchForm()
    query = None
    results = []
    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            results = Post.objects.annotate(search=SearchVector('title', 'slug', 'body')).filter(search=query)
    return render(request, 'blog/post/search.html', {'query': query, "form": form, 'results': results})
```

这个视图先初始化空白 `SearchForm` 表单，通过 `GET` 请求附加 URL 参数的方式提交表单。如果 `request.GET` 字典中存在 `query` 参数且通过了表单验证，就执行搜索并返回结果。

视图编写完毕，需要编写对应的模板，在 `/blog/post/` 目录下创建 `search.html` 文件，添加如下代码：

```
{% extends 'blog/base.html' %}

{% block title %}
Search
{% endblock %}

{% block content %}
{% if query %}
    <h1>Post containing {{ query }}</h1>
    <h3>
        {% with results.count as total_results %}
            Found {{ total_results }} result{{ total_results|pluralize }}
        {% endwith %}
    </h3>
    {% for post in results %}
        <h4>
            <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
        </h4>
        {{ post.body|truncatewords:5 }}
    {% empty %}
        <p>There are no results for your query.</p>
    {% endfor %}
    <p><a href="{% url 'blog:post_search' %}">Search again</a></p>
{% else %}
    <h1>Search for posts</h1>
    <form action="." method="get">
        {{ form.as_p }}
        <input type="submit" value="Search">
    </form>
{% endif %}
{% endblock %}
```

就像在视图中的逻辑一样，我们通过query参数存在与否判断表单是否提交。默认不提交表单的页面，显示表单和一个搜索按钮，进行搜索后则显示结果总数和搜索到的文章列表。

由于表单里配置了反向解析，所以编辑 blog 应用的 urls.py：

```
path('search/', views.post_search, name='post_search'),
```

现在启动站点，到 <http://127.0.0.1:8000/blog/search/> 可以看到表单页面如下：

The screenshot shows a search interface with the following elements:

- A large title "Search for posts" centered at the top.
- A label "Query:" followed by a redacted input field.
- A blue button labeled "SEARCH" below the input field.

输入查询内容然后点击搜索按钮，可以看到查询结果，如下所示：

Posts containing "music"

Found 2 results

[Another post more](#)

Post body.

[Who was Django Reinhardt?](#)

The Django web framework was ...

[Search again](#)

现在我们就创建了全文搜索功能了。

My blog

This is my blog. I've written 4 posts so far.

[Subscribe to my RSS feed](#)

Latest posts

- [Another post more](#)
- [New title](#)
- [Who was Django Reinhardt?](#)

Most commented posts

- [Who was Django Reinhardt?](#)
- [New title](#)
- [Another post more](#)
- [Old](#)

Django提供了一个 `SearchQuery` 类将一个查询词语转换成一个查询对象，默认会通过词干提取算法（ stemming algorithms ）转换成查询对象，用于更好的进行匹配。在查询时候还可能会依据相关性进行排名。PostgreSQL提供了一个排名功能，按照被搜索内容在一条数据里出现的次数和频率进行排名。

编辑 `blog` 应用的 `views.py` 文件：

```
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank
```

然后找到下边这行：

```
results = Post.objects.annotate(search=SearchVector('title', 'body')).filter(search=query)
```

替换成如下内容：

```
search_vector = SearchVector('title', 'body')
search_query = SearchQuery(query)
results = Post.objects.annotate(search=search_vector,
                                rank=SearchRank(search_vector, search_query))
                                .filter(search=search_query).order_by('-rank')
```

在上边的代码中，先创建了一个 `SearchQuery` 对象，用其过滤结果。之后使用`SearchRank`方法将结果按照相关性排序。

打开 <http://127.0.0.1:8000/blog/search/> 并且用不同的词语来测试搜索。下边是使用'django'搜索的示例：

Posts containing "django"

Found 3 results

[Django, Django, Django](#)

Django is the Web Framework ...

[Django twice](#)

Django offers full text search ...

[Django once](#)

A Python web framework.

[Search again](#)

4.6 搜索权重

当按照相关性进行搜索时，可以给不同的向量赋予不同的权重，从而影响搜索结果。例如，可以对在标题中搜索到的结果给予比正文中搜索到的结果更大的权重。编辑 `blog` 应用的 `views.py` 文件：

```
search_vector = SearchVector('title', weight='A') + SearchVector('body', weight='B')
results = Post.objects.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(rank__gte=0.3).order_by('-rank')
```

在上边的代码中，给 `title` 和 `body` 字段的搜索向量赋予了不同的权重。默认的权重 D, C, B, A 分别对应 0.1, 0.2, 0.4 和 1。我们给 `title` 字段的搜索向量赋予权重 1.0，给 `body` 字段的搜索向量的权重是 0.4，说明文章标题的重要性要比正文更重要，最后设置了只显示综合权重大于 0.3 的搜索结果。

4.7 三元相似性搜索

还有一种搜索方式是三元相似性搜索。三元指的是三个连续的字符。通过比较两个字符串里，有多少个三个连续的字符相同，可以检测这两个字符串的相似性。这种搜索方式对于不同语言中的相近单词很高效。

如果要在 PostgreSQL 中使用三元检索，必须安装一个 `pg_trgm` 扩展。

在系统命令行执行下列命令连接到数据库：

```
psql blog
```

然后输入下列数据库指令：

```
CREATE EXTENSION pg_trgm
```

然后编辑视图来增加三元相似搜索功能，编辑 `blog` 应用的 `views.py`，这一次需要导入的新组件：

```
from django.contrib.postgres.search import TrigramSimilarity
```

然后将Post搜索查询对象替换成如下这样：

```
results = Post.objects.annotate(
    similarity=TrigramSimilarity('title',query),
).filter(similarity__gte=0.1).order_by('-similarity')
```

打开 <http://127.0.0.1:8000/blog/search/>，然后试验不同的三元相似锁边，下边的例子显示了使用 `yango` 想搜索 `django` 的结果：

Posts containing "yango"

Found 1 result

Django Django

A Python web framework.

现在就为我们的博客创建了一个强力的搜索引擎。关于在Django中使用PostgreSQL的全文搜索，可以参考
<https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/>。

4.8 使用其他全文搜索引擎

除了常用的PostgreSQL之外，还有Solr和Elasticsearch等常用的全文搜索引擎，可以使用Haystack来将其集成到Django中。Haystack是一个Django应用，作为一个搜索引擎的抽象层工作。提供了与Django的QuerySet非常类似的API供执行搜索操作。关于Haystack的详情可以查看：<http://haystacksearch.org/>。

总结

在这一章学习了创建自定义的模板标签和过滤器，用于提供自定义的功能。还创建了站点地图用于搜索引擎优化和RSS feed为用户提供订阅功能。之后将站点数据库改用PostgreSQL从而实现了全文搜索功能。

在下一章中，将使用Django内置验证模块创建一个社交网站，创建用户信息和进行第三方认证登录。

第四章 创建社交网站

在之前的章节学习了如何创建站点地图、订阅信息和创建一个全文搜索引擎。这一章我们来开发一个社交网站。会创建用户登录、登出、修改和重置密码功能，为用户创建额外的用户信息，以及使用第三方身份认证登录。

本章包含以下内容：

- 使用Django内置验证模块
- 创建用户注册视图
- 使用自定义的用户信息表扩展用户模型
- 添加第三方身份认证系统

我们来创建本书的第二个项目。

1 社交网站

我们将创建一个社交网站，让用户可以把网上看到的图片分享到网站来。这个社交网站包含如下功能：

- 一个供用户注册、登录、登出、修改和重置密码的用户身份验证系统，还能够让用户自行填写用户信息
- 关注系统，让用户可以关注其他用户
- 一个JS小书签工具，让用户可以将外部的图片分享（上传）到本站
- 一个追踪系统，让用户可以看到他所关注的用户的上传内容

本章涉及到其中的第一个内容：用户身份验证系统。

1.1 启动社交网站项目

启动系统命令行，输入下列命令创建并激活一个虚拟环境：

```
mkdir env  
virtualenv env/bookmarks  
source env/bookmarks/bin/activate
```

终端会显示当前的虚拟环境，如下：

```
(bookmarks)laptop:~ zenx$
```

在终端中安装Django并启动 `bookmarks` 项目：

```
pip install Django==2.0.5  
django-admin startproject bookmarks
```

然后到项目根目录内创建 `account` 应用：

```
cd bookmarks/  
django-admin startapp account
```

然后在 `settings.py` 中的 `INSTALLED_APPS` 设置中激活该应用：

```
INSTALLED_APPS = [  
    'account.apps.AccountConfig',  
    # ...  
]
```

这里将我们的应用放在应用列表的最前边，原因是：我们稍后会为自己的应用编写验证系统的模板，Django内置的验证系统自带了一套模板，如此设置可以让我们的模板覆盖其他应用中的模板设置。Django按照 `INSTALLED_APPS` 中的顺序寻找模板。

之后执行数据迁移过程。

译者注：新创建的Django项目默认依然使用Python的SQLite数据库，建议读者为每个项目配置一个新创建的数据库。推荐使用上一章的PostgreSQL，因为本书之后还会使用PostgreSQL。

2 使用Django内置验证框架

Django提供了一个验证模块框架，具备用户验证，会话控制（session），权限和用户组功能并且自带一组视图，用于控制常见的用户行为如登录、登出、修改和重置密码。

验证模块框架位于 `django.contrib.auth`，也被其他Django的 `contrib` 库所使用。在第一章里创建超级用户的时候，就使用到了验证模块。

使用 `startproject` 命令创建一个新项目时，验证模块默认已经被设置并启用，包括 `INSTALLED_APPS` 设置中的 `django.contrib.auth` 应用，和 `MIDDLEWARE` 设置中的如下两个中间件：

- `AuthenticationMiddleware`：将用户与HTTP请求联系起来
- `SessionMiddleware`：处理当前HTTP请求的session

中间件是一个类，在接收HTTP请求和发送HTTP响应的阶段被调用，在本书的部分内容中会使用中间件，第十三章上线中会学习开发自定义中间件。

验证模块还包括如下数据模型：

- `User`：一个用户数据表，包含如下主要字段：`username`, `password`, `email`, `first_name`, `last_name` 和 `is_active`。
- `Group`：一个用户组表格
- `Permission`：存放用户和组的权限清单

验证框架还包括默认的验证视图以及对应表单，稍后会使用到。

2.1 创建登录视图

从这节开始使用Django的验证模块，一个登录视图需要如下功能：

- 通过用户提交的表单获取用户名和密码
- 将用户名和密码与数据库中的数据进行匹配
- 检查用户是否处于活动状态
- 通过在HTTP请求上附加session，让用户进入登录状态

首先需要创建一个登录表单，在 `account` 应用内创建 `forms.py` 文件，添加以下内容：

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

这是用户输入用户名和密码的表单。由于一般密码框不会明文显示，这里采用了 `widget=forms.PasswordInput`，令其在页面上显示为一个 `type="password"` 的 `INPUT` 元素。

然后编辑 `account` 应用的 `views.py` 文件，添加如下代码：

```
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth import authenticate, login
from .forms import LoginForm

def user_login(request):
    if request.method == "POST":
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(request, username=cd['username'], password=cd['password'])
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponseRedirect("Authenticated successfully")
                else:
                    return HttpResponseRedirect("Disabled account")
            else:
                return HttpResponseRedirect("Invalid login")
        else:
            form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

这是我们的登录视图，其基本逻辑是：当视图接受一个 `GET` 请求，通过 `form = LoginForm()` 实例化一个空白表单；如果接收到 `POST` 请求，则进行如下工作：

1. 通过 `form = LoginForm(request.POST)`，使用提交的数据实例化一个表单对象。
2. 通过调用 `form.is_valid()` 验证表单数据。如果未通过，则将当前表单对象展示在页面中。
3. 如果表单数据通过验证，则调用内置 `authenticate()` 方法。该方法接受 `request` 对象，`username` 和 `password` 三个参数，之后到数据库中进行匹配，如果匹配成功，会返回一个 `User` 数据对象；如果未找到匹配数据，返回 `None`。在匹配失败的情况下，视

图返回一个登陆无效信息。

4. 如果用户数据成功通过匹配，则根据 `is_active` 属性检查用户是否为活动用户，这个属性是Django内置 `User` 模型的一个字段。
如果用户不是活动用户，则返回一个消息显示不活动用户。
5. 如果用户是活动用户，则调用 `login()` 方法，在会话中设置用户信息，并且返回登录成功的消息。

注意区分内置的 `authenticate()` 和 `login()` 方法。`authenticate()` 仅到数据库中进行匹配并且返回 `User` 数据对象，其工作类似于进行数据库查询。而 `login()` 用于在当前会话中设置登录状态。二者必须搭配使用才能完成用户名和密码的数据验证和用户登录的功能。

现在需要为视图设置路由，在 `account` 应用下创建 `urls.py`，添加如下代码：

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

urlpatterns = [
    path('login/', views.user_login, name='login'),
]
```

然后编辑项目的根 `ulrs.py` 文件，导入 `include` 并且增加一行转发到account应用的二级路由配置：

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

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

之后需要配置模板。由于项目还没有任何模板，可以先创建一个母版，在 `account` 应用下创建如下目录和文件结构：

```
templates/
    account/
        login.html
    base.html
```

编辑 `base.html`，添加下列代码：

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        <span class="logo">Bookmarks</span>
    </div>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
</body>
</html>
```

这是这个项目使用的母版。和上一个项目一样使用了CSS文件，你需要把 `static` 文件夹从源码复制到 `account` 应用目录下。这个母版有一个 `title` 块和一个 `content` 块用于继承。

译者注：原书第一章使用了 `{% load static %}`，这里的模板使用了 `{% load staticfiles %}`，作者并没有对这两者的差异进行说明，读者可以参考 [What is the difference between `{% load staticfiles %}` and `{% load static %}`？](#)

之后编写 `account/login.html` :

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

{% block title %}Log-in{% endblock %}

{% block content %}
<h1>Log-in</h1>
<p>Please, use the following form to log-in:</p>
<form action="." method="post">
{{ form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Log in"></p>
</form>
{% endblock %}
```

这是供用户填写登录信息的页面，由于表单通过 `Post` 请求提交，所以需要 `{% csrf_token %}`。

我们的站点还没有任何用户，建立一个超级用户，然后使用超级用户到 <http://127.0.0.1:8000/admin/> 登录，会看到默认的管理后台：

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups

[+ Add](#) [Change](#)

Users

[+ Add](#) [Change](#)

Recent actions

My actions

None available

使用管理后台添加一个用户，然后打开 <http://127.0.0.1:8000/account/login/>，可以看到如下登录界面：

A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:8000/account/login/`. The main content area has a green header bar with the word "Bookmarks". Below this, the page title is "Log-in". A horizontal line separates the title from the form. The form instructions say "Please, use the following form to log-in:". It contains two input fields, one for "Username" and one for "Password", both of which are currently empty and shaded gray. Below the password field is a green rectangular button with the text "LOG IN" in white capital letters.

Please, use the following form to log-in:

Username:

Password:

LOG IN

填写刚创建的用户信息并故意留空表单然后提交，可以看到错误信息如下：

The screenshot shows a large, bold, dark blue text "Username:" centered on the page. This is likely a placeholder or a large error message indicating that the required field was not filled out.

Username:

test

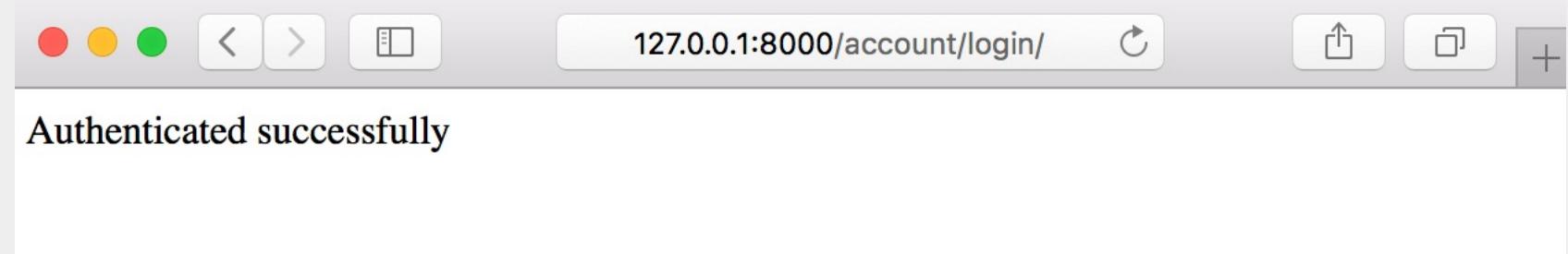
This field is required.

Password:



注意和第一章一样，很可能一些现代浏览器会阻止表单提交，修改模板关闭表单的浏览器验证即可。

再进行一些实验，如果输入不存在的用户名或密码，会得到无效登录的提示，如果输入了正确的信息，就会看到如下的登录成功信息：



2.2 使用内置验证视图

Django内置很多视图和表单可供直接使用，上一节的登录视图就是一个很好的例子。在大多数情况下都可以使用Django内置的验证模块而无需自行编写。

Django在 `django.contrib.auth.views` 中提供了如下基于类的视图供使用：

- `LoginView`：处理登录表单填写和登录功能（和我们写的功能类似）
- `LogoutView`：退出登录
- `PasswordChangeView`：处理一个修改密码的表单，然后修改密码
- `PasswordChangeDoneView`：成功修改密码后执行的视图
- `PasswordResetView`：用户选择重置密码功能执行的视图，生成一个一次性重置密码链接和对应的验证token，然后发送邮件给用户
- `PasswordResetDoneView`：通知用户已经发送给了他们一封邮件重置密码
- `PasswordResetConfirmView`：用户设置新密码的页面和功能控制
- `PasswordResetCompleteView`：成功重置密码后执行的视图

上边的视图列表按照一般处理用户相关功能的顺序列出相关视图，在编写带有用户功能的站点时可以参考使用。这些内置视图的默认值可以被修改，比如渲染的模板位置和使用的表单等。

可以通过官方文档 <https://docs.djangoproject.com/en/2.0/topics/auth/default/#all-authentication-views> 了解更多内置验证视图的信息。

2.3 登录与登出视图

由于直接使用内置视图和内置数据模型，所以不需要编写模型与视图，来为内置登录和登出视图配置URL，编辑 `account` 应用的 `urls.py` 文件，注释掉之前的登录方法，改成内置方法：

```
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views

urlpatterns = [
    # path('login/', views.user_login, name='login'),
    path('login', auth_views.LoginView.as_view(), name='login'),
    path('logout', auth_views.LogoutView.as_view(), name='logout'),
]
```

现在我们把登录和登出的URL导向了内置视图，然后需要为内置视图建立模板

在 `templates` 目录下新建 `registration` 目录，这个目录是内置视图默认到当前应用的模板目录里寻找具体模板的位置。

`django.contrib.admin` 模块中自带一些验证模板，用于管理后台使用。我们在 `INSTALLED_APPS` 中将 `account` 应用放到 `admin` 应用的上边，令django默认使用我们编写的模板。

在 `templates/registration` 目录下创建 `login.html` 并添加如下代码：

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

{% block title %}Log-in{% endblock %}

{% block content %}
<h1>Log-in</h1>
{% if form.errors %}


Your username and password didn't match.  

Please try again.


{% else %}
```

```
<p>Please, use the following form to log-in:</p>
{%- endif %}

<div class="login-form">
    <form action="{% url 'login' %}" method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <input type="hidden" name="next" value="{{ next }}>
        <p><input type="submit" value="Log-in"></p>
    </form>
</div>

{%- endblock %}
```

这个模板和刚才自行编写登录模板很类似。内置登录视图默认使用 `django.contrib.auth.forms` 里的 `AuthenticationForm` 表单，通过检查 `{% if form.errors %}` 可以判断验证信息是否错误。注意我们添加了一个 `name` 属性为 `next` 的隐藏 `<input>` 元素，这是内置视图通过 `Get` 请求获得并记录 `next` 参数的位置，用于返回登录前的页面，例如 `http://127.0.0.1:8000/account/login/?next=/account/`。

`next` 参数必须是一个URL地址，如果具有这个参数，登录视图会在登录成功后将用户重定向到这个参数的URL。

在 `registration` 目录下创建 `logged_out.html`：

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

{% block title %}
Logged out
{% endblock %}

{% block content %}
<h1>Logged out</h1>
```

```
<p>You have been successfully logged out. You can <a href="{% url 'login' %}">log-in again</a>.</p>
{% endblock %}
```

这是用户登出之后显示的提示页面。

现在我们的站点已经可以使用用户登录和登出的功能了。现在还需要为用户制作一个登录成功后自己的首页，打开 `account` 应用的 `views.py` 文件，添加如下代码：

```
from django.contrib.auth.decorators import login_required
@login_required
def dashboard(request):
    return render(request, 'account/dashboard.html', {'section': 'dashboard'})
```

使用 `@login_required` 装饰器，表示被装饰的视图只有在用户登录的情况下才会被执行，如果用户未登录，则会将用户重定向至 `Get` 请求附加的 `next` 参数指定的URL。这样设置之后，如果用户在未登录的情况下，无法看到首页。

还定义了一个参数 `section`，可以用来追踪用户当前所在的功能板块。

现在可以创建首页对应的模板，在 `templates/account/` 目录下创建 `dashboard.html`：

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

{% block title %}
Dashboard
{% endblock %}

{% block content %}
<h1>Dashboard</h1>
<p>Welcome to your dashboard.</p>
{% endblock %}
```

然后在 `account` 应用的 `urls.py` 里增加新视图对应的URL：

```
urlpatterns = [
    # ...
    path('', views.dashboard, name='dashboard'),
]
```

还需要在 `settings.py` 里增加如下设置：

```
LOGIN_REDIRECT_URL = 'dashboard'
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
```

这三个设置分别表示：

- 如果没有指定 `next` 参数，登录成功后重定向的URL
- 用户需要登录的情况下被重定向到的URL地址（例如 `@login_required` 重定向到的地址）
- 用户需要登出的时候被重定向到的URL地址

这里都使用了 `path()` 方法中的 `name` 属性，以动态的返回链接。在这里也可以硬编码URL。

总结一下我们现在做过的工作：

- 为项目添加内置登录和登出视图
- 为两个视图编写模板并编写了首页视图和对应模板
- 为三个视图配置了URL

最后需要在母版上添加登录和登出相关的展示。为了实现这个功能，必须根据当前用户是否登录，决定模板需要展示的内容。在内置函数 `LoginView` 成功执行之后，验证模块的中间件在 `HttpRequest` 对象上设置了用户对象 `User`，可以通

过 `request.user` 访问用户信息。在用户未登录的情况下，`request.user` 也存在，是一个 `AnonymousUser` 类的实例。判断当前用户是否登录最好的方式就是判断 `User` 对象的 `is_authenticated` 只读属性。

编辑 `base.html`，修改ID为 `header` 的 `<div>` 标签：

```
<div id="header">
<span class="logo">Bookmarks</span>
  {% if request.user.is_authenticated %}
    <ul class="menu">
      <li {% if section == 'dashboard' %}class="selected"{% endif %}><a href="{% url 'dashboard' %}">My dashboard</a>
      <li {% if section == 'images' %}class="selected"{% endif %}><a href="#">Images</a></li>
      <li {% if section == 'people' %}class="selected"{% endif %}><a href="#">People</a></li>
    </ul>
  {% endif %}

  <span class="user">
    {% if request.user.is_authenticated %}
      Hello {{ request.user.first_name }},{{ request.user.username }},<a href="{% url 'logout' %}">Logout</a>
    {% else %}
      <a href="{% url 'login' %}">Log-in</a>
    {% endif %}
  </span>
</div>
```

上边的视图只显示站点的菜单给已登录用户。还添加了根据 `section` 的内容为 `` 添加CSS类 `selected` 的功能，用于显示高亮当前的板块。最后对登录用户显示名称和登出链接，对未登录用户则显示登录链接。

现在启动项目，到 <http://127.0.0.1:8000/account/login/>，会看到登录页面，输入有效的用户名和密码并点击登录按钮，之后会看到如下页面：

Dashboard

Welcome to your dashboard.

可以看到当前的 My dashboard 应用了 `selected` 类的CSS样式。当前用户的信息显示在顶部的右侧，点击登出链接，会看到如下页面：

Logged out

You have been successfully logged out. You can [log-in again](#).

可以看到用户已经登出，顶部的菜单栏已经不再显示，右侧的链接变为登录链接。

如果这里看到Django内置的管理站点样式的页面，检查 `settings.py` 文件中的 `INSTALLED_APPS` 设置，确保 `account` 应用在 `django.contrib.admin` 应用的上方。由于内置的视图和我们自定义的视图使用了相同的相对路径，Django的模板加载器会使用先找到的模板。

2.4 修改密码视图

在用户登录之后需要允许用户修改密码，我们在项目中集成Django的内置修改密码相关的视图。编辑 `account` 应用的 `urls.py` 文件，添加如下两行URL：

```
path('password_change', auth_views.PasswordChangeView.as_view(), name='password_change'),
path('password_change/done/', auth_views.PasswordChangeDoneView.as_view(), name='password_change_done'),
```

`passwordChangeView` 视图会控制渲染修改密码的页面和表单，`PasswordChangeDoneView` 视图在成功修改密码之后显示成功消息。

之后要为两个视图创建模板，在 `templates/registration/` 目录下创建 `password_change_form.html`，添加如下代码：

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

{% block title %}
Change your password
{% endblock %}

{% block content %}
<h1>Change your password</h1>
<p>Use the form below to change your password.</p>
<form action="." method="post" novalidate>
{{ form.as_p }}
<p><input type="submit" value="Change"></p>
{% csrf_token %}
```

```
</form>
{% endblock %}
```

`password_change_form.html` 模板包含修改密码的表单，再在同一目录下创建 `password_change_done.html`：

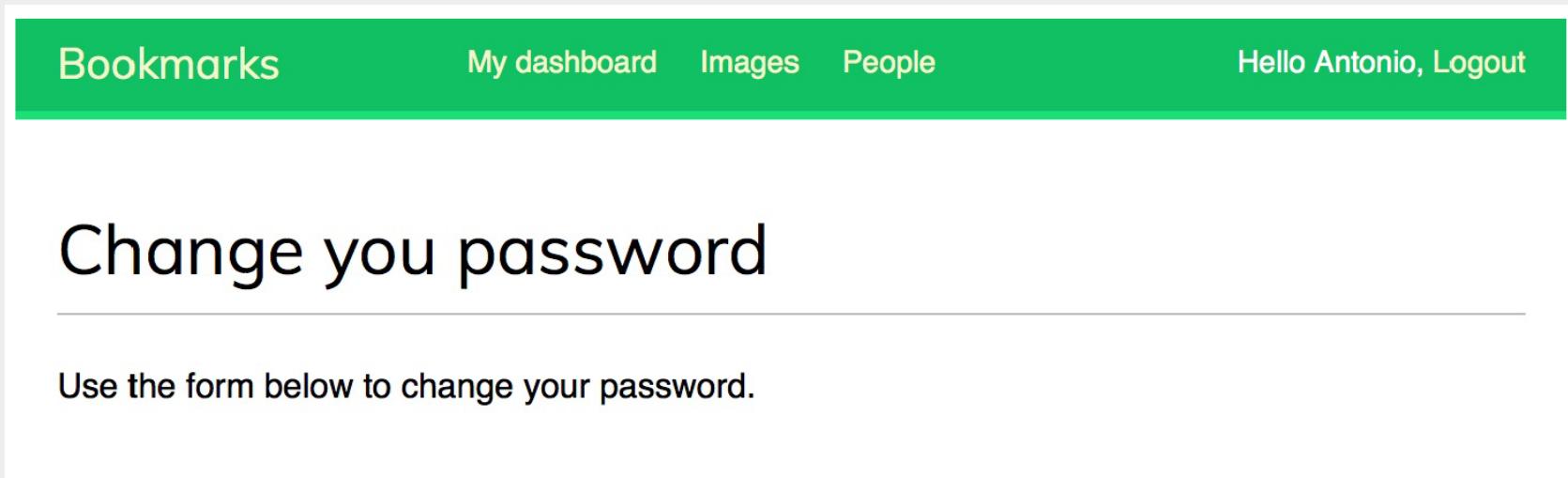
```
{% extends 'base.html' %}

{% block title %}
Password changed
{% endblock %}

{% block content %}
<h1>Password changed</h1>
<p>Your password has been successfully changed.</p>
{% endblock %}
```

`password_change_done.html` 模板包含成功创建密码后的提示消息。

启动服务，到 http://127.0.0.1:8000/account/password_change/，成功登录之后可看到如下页面：



Old password:

New password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

New password confirmation:

CHANGE

填写表单并修改密码，之后可以看到成功消息：

Password changed

Your password has been successfully changed.

之后登出再登录，验证是否确实成功修改密码。

2.5 重置密码视图

编辑 `account` 应用的 `urls.py` 文件，添加如下对应到内置视图的URL：

```
path('password_reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(), name='password_reset_done'),
path('reset/<uidb64>/<token>', auth_views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
path('reset/done/', auth_views.PasswordResetCompleteView.as_view(), name='password_reset_complete'),
```

然后在 `account` 应用的 `templates/registration/` 目录下创建 `password_reset_form.html`：

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

{% block title %}
```

```
Reset your password
{% endblock %}

{% block content %}
<h1>Forgotten your password?</h1>
<p>Enter your e-mail address to obtain a new password.</p>
<form action="." method="post" novalidate>
{{ form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Send e-mail"></p>
</form>
{% endblock %}
```

在同一目录下创建发送邮件的页面 `password_reset_email.html`，添加如下代码：

```
Someone asked for password reset for email {{ email }}. Follow the link
below:
{{ protocol }}://{{ domain }}{% url "password_reset_confirm" uidb64=uid token=token %}
Your username, in case you've forgotten: {{ user.get_username }}
```

这个模板用来渲染向用户发送的邮件内容。

之后在同一目录再创建 `password_reset_done.html`，表示成功发送邮件的页面：

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

{% block title %}
Reset your password
{% endblock %}

{% block content %}
<h1>Reset your password</h1>
```

```
<p>We've emailed you instructions for setting your password.</p>
<p>If you don't receive an email, please make sure you've entered the
address you registered with.</p>
{% endblock %}
```

然后创建重置密码的页面 `password_reset_confirm.html`，这个页面是用户从邮件中打开链接后经过视图处理后返回的页面：

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

{% block title %}Reset your password{% endblock %}

{% block content %}
    <h1>Reset your password</h1>
    {% if validlink %}
        <p>Please enter your new password twice:</p>
        <form action"." method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <p><input type="submit" value="Change my password"/></p>
        </form>
    {% else %}
        <p>The password reset link was invalid, possibly because it has
            already been used. Please request a new password reset.</p>
    {% endif %}
{% endblock %}
```

这个页面里有一个变量 `validlink`，表示用户点击的链接是否有效，由 `PasswordResetConfirmView` 视图传入模板。如果有效就显示重置密码的表单，如果无效就显示一段文字说明链接无效。

在同一目录内建立 `password_reset_complete.html`：

```
{% extends "base.html" %}  
{% block title %}Password reset{% endblock %}  
{% block content %}  
<h1>Password set</h1>  
<p>Your password has been set. You can <a href="{% url "login" %}">log in  
now</a></p>  
{% endblock %}
```

最后编辑 `registration/login.html`，在 `<form>` 元素之后加上如下代码，为页面增加重置密码的链接：

```
<p><a href="{% url 'password_reset' %}">Forgotten your password?</a></p>
```

之后在浏览器中打开 <http://127.0.0.1:8000/account/login/>，点击 `Forgotten your password ?` 链接，会看到如下页面：

Forgotten your password?

Enter your e-mail address to obtain a new password.

Email:

SEND E-MAIL

这里必须在 `settings.py` 中配置SMTP服务器，在第二章中已经学习过配置STMP服务器的设置。如果确实没有SMTP服务器，可以增加一行：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

以让Django将邮件内容输出到命令行窗口中。

返回浏览器，填入一个已经存在的用户的电子邮件地址，之后点SEND E-MAIL按钮，会看到如下页面：

The screenshot shows a web browser window with a green header bar containing the text "Bookmarks" on the left and "Log-in" on the right. Below the header, the main content area has a large heading "Reset your password". Underneath the heading, there is a message: "We've emailed you instructions for setting your password." Below this message, another line of text reads: "If you don't receive an email, please make sure you've entered the address you registered with."

此时看一下启动Django站点的命令行窗口，会打印如下邮件内容（或者到信箱中查看实际收到的电子邮件）：

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: user@domain.com
Date: Fri, 15 Dec 2017 14:35:08 -0000
Message-ID: <20150924143508.62996.55653@zenx.local>
Someone asked for password reset for email user@domain.com. Follow the link
below:
http://127.0.0.1:8000/account/reset/MQ/45f-9c3f30caaf523055fcc/
Your username, in case you've forgotten: zenx
```

这个邮件的内容就是 `password_reset_email.html` 经过渲染之后的实际内容。其中的URL指向视图动态生成的链接，将这个URL复制到浏览器中打开，会看到如下页面：

Reset your password

Please enter your new password twice:

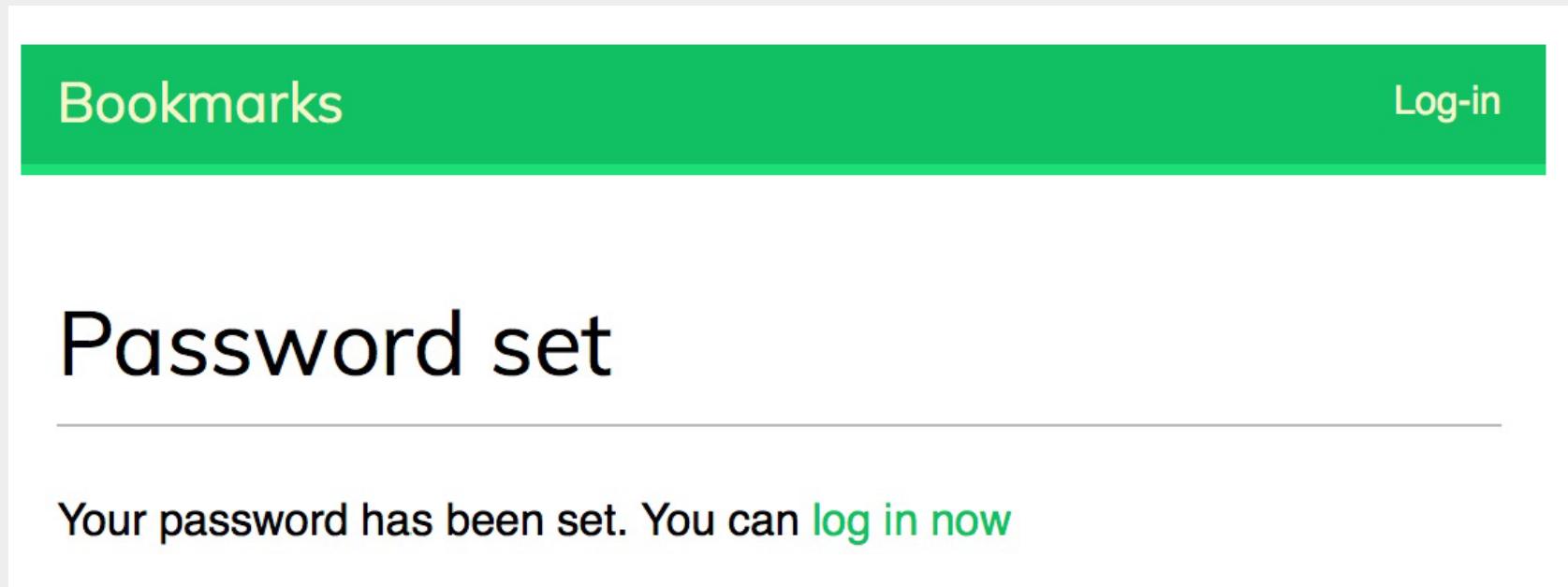
New password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

New password confirmation:

CHANGE MY PASSWORD

这个页面使用 `password_reset_confirm.html` 模板生成，填入一个新密码然后点击CHANGE MY PASSWORD按钮，Django会用你输入的内容生成加密后的密码保存在数据库中，然后会看到如下页面：



现在就可以使用新密码登录了。这里生成的链接只能使用一次，如果反复打开该链接，会收到无效链接的错误。

我们现在已经集成了Django内置验证模块的主要功能，在大部分情况下，可以直接使用内置验证模块。也可以自行编写所有的验证程序。

在第一个项目中，我们提到为应用配置单独的二级路由，有助于应用的复用。现在的 `account` 应用的 `urls.py` 文件中所有配置到内置视图的URL，可以用如下一行来代替：

```
urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
]
```

可以在github上看到 `django.contrib.auth.urls` 的源代码：

<https://github.com/django/django/blob/stable/2.0.x/django/contrib/auth/urls.py>。

3 用户注册与用户信息

已经存在的用户现在可以登录、登出、修改和重置密码了。现在需要建立一个功能让用户注册。

3.1 用户注册

为用户注册功能创建一个简单的视图：先建立一个供用户输入用户名、姓名和密码的表单。编辑 `account` 应用的 `forms.py` 文件，添加如下代码：

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

class userRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='password', widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password', widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ('username', 'first_name', 'email')

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError(r"Password don't match.")
        return cd['password2']
```

这里通过用户模型建立了一个模型表单，只包含 `username`，`first_name` 和 `email` 字段。这些字段会根据 `User` 模型中的设置进行验证，比如如果输入了一个已经存在的用户名，则验证不会通过，因为 `username` 字段被设置了 `unique=True`。添加了两个新的字段 `password` 和 `password2`，用于用户输入并且确认密码。定义了一个 `clean_password2()` 方法用于检查两个密码是否一致，这个方法是一个验证器方法，会在调用 `is_valid()` 方法的时候执行。可以对任意的字段采用 `clean_<fieldname>()` 方法名创建一个验证器。Forms类还拥有一个 `clean()` 方法用于验证整个表单，可以方便的验证彼此相关的字段。

译者注：这里必须了解表单的验证顺序。`clean_password2()` 方法中使用了 `cd['password2']`；为什么验证器还没有执行完毕的时候，`cleaned_data` 中已经存在 `password2` 数据了呢？[这里](#) 有一篇介绍django验证表单顺序的文章，可以看到，在执行自定义验证器之前，已经执行了每个字段的 `clean()` 方法，这个方法仅针对字段本身的属性进行验证，只要这个通过了，`cleaned_data` 中就有了数据，之后才执行自定义验证器，最后执行 `form.clean()` 完成验证。如果过程中任意时候抛出 `ValidationError`，`cleaned_data` 里就会只剩有效的值，`errors` 属性内就有了错误信息。

关于用户注册，Django提供了一个位于 `django.contrib.auth.forms` 的 `UserCreationForm` 表单供使用，和我们自行编写的表单非常类似。

编辑 `account` 应用的 `views.py` 文件，添加如下代码：

```
from .forms import LoginForm, UserRegistrationForm

def register(request):
    if request.method == "POST":
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # 建立新数据对象但是不写入数据库
            new_user = user_form.save(commit=False)
            # 设置密码
            new_user.set_password(user_form.cleaned_data['password'])
            # 保存User对象
```

```
        new_user.save()
    return render(request, 'account/register_done.html', {'new_user': new_user})
else:
    user_form = UserRegistrationForm()
return render(request, 'account/register.html', {'user_form': user_form})
```

这个视图逻辑很简单，我们使用了 `set_password()` 方法设置加密后的密码。

再配置 `account` 应用的 `urls.py` 文件，添加如下的URL匹配：

```
path('register/', views.register, name='register'),
```

在 `templates/account/` 目录下创建模板 `register.html`，添加如下代码：

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

{% block title %}
Create an account
{% endblock %}

{% block content %}
<h1>Create an account</h1>
<p>Please, sign up using the following form:</p>
<form action="." method="post" novalidate>
{{ user_form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Register"></p>
</form>
{% endblock %}
```

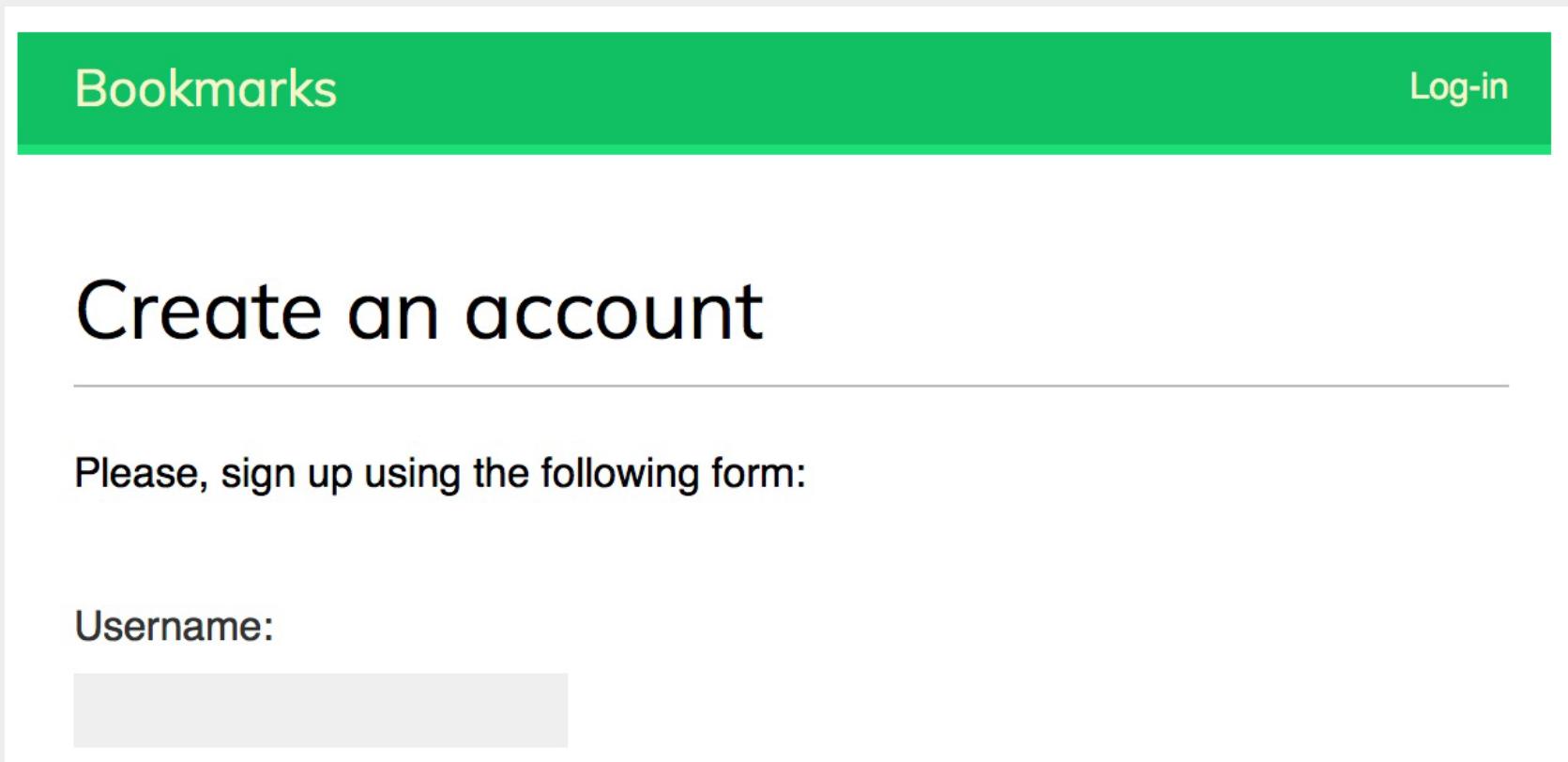
在同一目录下创建 `register_done.html` 模板，用于显示注册成功后的信息：

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

{% block title %}
Welcome
{% endblock %}

{% block content %}
<h1>Welcome {{ new_user.first_name }}!</h1>
<p>Your account has been successfully created. Now you can <a href="{% url 'login' %}">log in</a>.</p>
{% endblock %}
```

现在可以打开 <http://127.0.0.1:8000/account/register/>，看到注册界面如下：



Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

First name:

Email address:

Password:

Repeat password:

CREATE MY ACCOUNT

填写表单并点击CREATE MY ACCOUNT按钮，如果表单正确提交，会看如下成功页面：

Welcome Paloma!

Your account has been successfully created. Now you can [log in](#).

3.2 扩展用户模型

Django内置验证模块的User模型只有非常基础的字段信息，可能需要额外的用户信息。最好的方式是建立一个用户信息模型，然后通过一对关联字段，将用户信息模型和用户模型联系起来。

编辑 `account` 应用的 `models.py` 文件，添加以下代码：

```
from django.db import models
from django.conf import settings

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='user/%Y/%m/%d/', blank=True)

    def __str__(self):
        return "Profile for user {}".format(self.user.username)
```

为了保持代码通用性，使用 `get_user_model()` 方法来获取用户模型；当定义其他表与内置 `User` 模型的关系时使用 `settings.AUTH_USER_MODEL` 指代 `User` 模型。

这个 `Profile` 模型的 `user` 字段是一个一对一关联到用户模型的关系字段。将 `on_delete` 设置为 `CASCADE`，当用户被删除时，其对应的信息也被删除。这里还有一个图片文件字段，必须安装Python的 `Pillow` 库才能使用图片文件字段，在系统命令行中输入：

```
pip install Pillow==5.1.0
```

由于我们要允许用户上传图片，必须配置Django让其提供媒体文件服务，在 `settings.py` 中加入下列内容：

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

`MEDIA_URL` 表示存放和提供用户上传文件的URL路径，`MEDIA_ROOT` 表示实际媒体文件的存放目录。这里都采用相对地址动态生成URL。

来编辑一下 `bookmarks` 项目的根 `urls.py`，修改其中的代码如下：

```
from django.contrib import admin  
from django.urls import path, include  
from django.conf import settings  
from django.conf.urls.static import static  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('account/', include('account.urls')),  
]
```

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

这样设置后，Django开发服务器在 `DEBUG=True` 的情况下会提供媒体文件服务。

`static()` 方法仅用于开发环境，在生产环境中，不要用Django提供静态文件服务（而是用Web服务程序比如NGINX等提供静态文件服务）。

建立了新的模型之后需要执行数据迁移过程。之后将新的模型加入到管理后台，编辑 `account` 应用的 `admin.py` 文件，将 `Profile` 模型注册到管理后台中：

```
from django.contrib import admin  
from .models import Profile  
  
@admin.register(Profile)  
class ProfileAdmin(admin.ModelAdmin):  
    list_display = ['user', 'date_of_birth', 'photo']
```

启动站点，打开 <http://127.0.0.1:8000/admin/>，可以在管理后台中看到新增的模型：



现在需要让用户填写额外的用户信息，为此需要建立表单，编辑 `account` 应用的 `forms.py` 文件：

```
from .models import Profile

class UserEditForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

class ProfileEditForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('date_of_birth', 'photo')
```

这两个表单解释如下：

- `UserEditForm`：这个表单依据 `User` 类生成，让用户输入姓，名和电子邮件。
- `ProfileEditForm`：这个表单依据 `Profile` 类生成，可以让用户输入生日和上传一个头像。

之后建立视图，编辑 `account` 应用的 `views.py` 文件，导入 `Profile` 模型：

```
from .models import Profile
```

然后在 `register` 视图的 `new_user.save()` 下增加一行：

```
Profile.objects.create(user=new_user)
```

当用户注册的时候，会自动建立一个空白的用户信息关联到用户。在之前创建的用户，则必须在管理后台中手工为其添加对应的 `Profile` 对象

还必须让用户可以编辑他们的信息，在同一个文件内添加下列代码：

```
from .forms import LoginForm, UserRegistrationForm, UserEditForm, ProfileEditForm

@login_required
def edit(request):
    if request.method == "POST":
        user_form = UserEditForm(instance=request.user, data=request.POST)
        profile_form = ProfileEditForm(instance=request.user.profile, data=request.POST, files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(instance=request.user.profile)

    return render(request, 'account/edit.html', {'user_form': user_form, 'profile_form': profile_form})
```

这里使用了`@login_required`装饰器，因为用户必须登录才能编辑自己的信息。我们使用**UserEditForm**表单存储内置的**User**类的数据，用**ProfileEditForm**存放**Profile**类的数据。然后调用`is_valid()`验证两个表单数据，如果全部都通过，将使用`save()`方法写入数据库。

译者注：原书没有解释`instance`参数。`instance`用于指定表单类实例化为某个具体的数据对象。在这个例子里，将**UserEditForm**的`instance`指定为`request.user`表示该对象是数据库中当前登录用户那一行的数据对象，而不是一个空白的数据对象，**ProfileEditForm**的`instance`属性指定为当前用户对应的**Profile**类中的那行数据。这里如果不指定`instance`参数，则变成向数据库中增加两条新记录，而不是修改原有记录。

之后编辑**account**应用的**urls.py**文件，为新视图配置URL：

```
path('edit/', views.edit, name='edit'),
```

最后，在**templates/account/**目录下创建**edit.html**，添加如下代码：

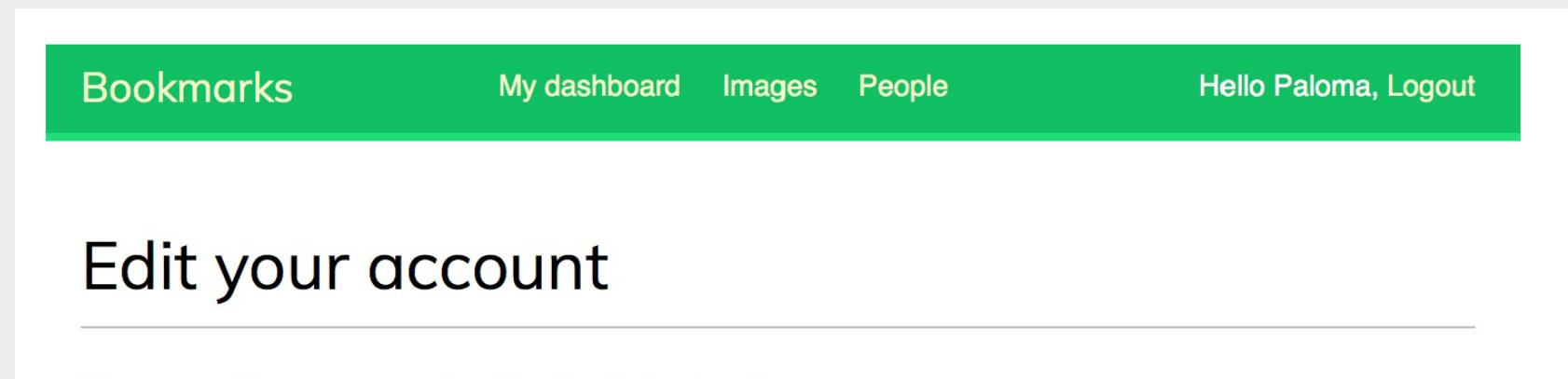
```
{#edit.html#}
{% extends 'base.html' %}

{% block title %}
Edit your account
{% endblock %}

{% block content %}
<h1>Edit your account</h1>
<p>You can edit your account using the following form:</p>
<form action="." method="post" enctype="multipart/form-data" novalidate>
{{ user_form.as_p }}
{{ profile_form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Save changes"></p>
</form>
{% endblock %}
```

由于这个表单可能处理用户上传头像文件，所以必须设置 `enctype="multipart/form-data"`。我们采用一个HTML表单同时提交 `user_form` 和 `profile_form` 表单。

启动站点，注册一个新用户，然后打开 <http://127.0.0.1:8000/account/edit/>，可以看到页面如下：



You can edit your account using the following form:

First name:

Paloma

Last name:

Melé

Email address:

paloma@zenxit.com

Date of birth:

1981-04-14

Photo:

no file selected

SAVE CHANGES

现在可以在用户登录后的首页加上修改用户信息的链接了，打开 [account/dashboard.html](#)，找到下边这行：

```
<p>Welcome to your dashboard.</p>
```

将其替换为：

```
<p>Welcome to your dashboard. You can <a href="{% url 'edit' %}">edit your profile</a> or <a href="{% url "password
```

用户现在可以通过登录后的首页修改用户信息，打开 <http://127.0.0.1:8000/account/> 然后可以看到新增了修改用户信息的链接，页面如下：

Dashboard

Welcome to your dashboard. You can **edit your profile** or **change your password**.

3.2.1 使用自定义的用户模型

Django提供了使用自定义的模型替代内置 `User` 模型的方法，需要编写自定义的类继承 `AbstractUser` 类。这个 `AbstractUser` 类提供了默认的用户模型的完整实现，作为一个抽象类供其他类继承。关于模型的继承将在本书最后一个项目中学习。可以在 <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#substituting-a-custom-user-model> 找到关于自定义用户模型的详细信息。

使用自定义用户模型比起默认内置用户模型可以更好的满足开发需求，但需要注意的是会影响一些使用Django内置用户模型的第三方应用。

3.3 使用消息框架

当用户在我们的站点执行各种操作时，在一些关键操作可能需要通知用户其操作是否成功。Django有一个内置消息框架可以给用户发送一次性的通知。

消息模块位于 `django.contrib.messages`，并且已经被包含在初始化的 `INSTALLED_APPS` 设置中，还有一个默认启用的中间件叫做 `django.contrib.messages.middleware.MessageMiddleware`，共同构成了消息系统。

消息框架提供了非常简单的方法向用户发送通知：默认在cookie中存储消息内容（根据session的存储设置），然后会在下一次HTTP请求的时候在对应的响应上附加该信息。导入消息模块并且在视图中使用很简单的语句就可以发送消息，例如：

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

这样就在请求上附加了一个错误信息。可以使用 `add_message()` 或如下的方法创建消息：

- `success()`：一个动作成功之后发送的消息
- `info()`：通知性质的消息
- `warning()`：警告性质的内容，所谓警告就是还没有失败但很可能失败的情况
- `error()`：错误信息，通知操作失败
- `debug()`：除错信息，给开发者展示，在生产环境中需要被移除

在我们的站点中增加消息内容。由于消息是贯穿整个网站的，所以打算将消息显示的部分设置在母版中，编辑 `base.html`，在ID为 `header` 的 `<div>` 标签和ID为 `content` 的 `<div>` 标签之间增加下列代码：

```
{% if messages %}
<ul class="messages">
```

```
{% for message in messages %}
    <li class="{{ message.tags }}>{{ message|safe }}<a href="#" class="close">X</a></li>
{% endfor %}
</ul>
{% endif %}
```

在模板中使用了 `messages` 变量，在后文可以看到视图并未向模板传入该变量。这是因为在 `settings.py` 中的 `TEMPLATES` 设置中，`context_processors` 的设置中包含 `django.contrib.messages.context_processors.messages` 这个上下文管理器，从而为模板传入了 `messages` 变量，而无需经过视图。默认情况下可以看到还有 `debug`，`request` 和 `auth` 三个上下文处理器。其中后两个就是我们在模板中可以直接使用 `request.user` 而无需传入该变量，也无需为 `request` 对象添加 `user` 属性的原因。

之后来修改 `account` 应用的 `views.py` 文件，导入 `messages`，然后编辑 `edit` 视图：

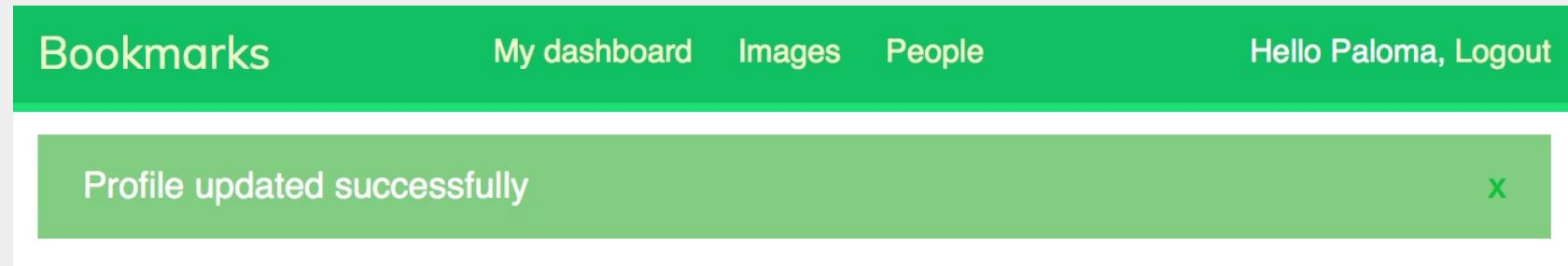
```
from django.contrib import messages

@login_required
def edit(request):
    if request.method == "POST":
        user_form = UserEditForm(instance=request.user, data=request.POST)
        profile_form = ProfileEditForm(instance=request.user.profile, data=request.POST, files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Profile updated successfully')
        else:
            messages.error(request, "Error updating your profile")
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(instance=request.user.profile)

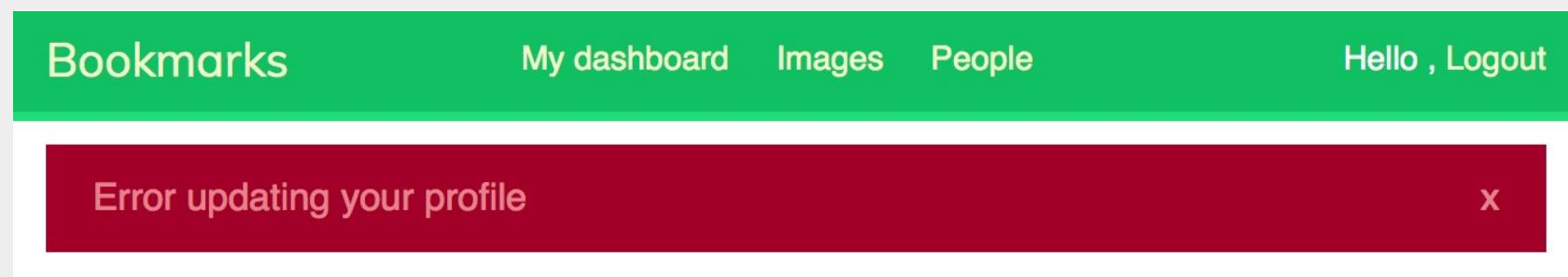
    return render(request, 'account/edit.html', {'user_form': user_form, 'profile_form': profile_form})
```

为视图增加了两条语句，分别在成功登录之后显示成功信息，在表单验证失败的时候显示错误信息。

浏览器中打开 <http://127.0.0.1:8000/account/edit/>，编辑用户信息，之后可以看到成功信息如下：



故意填写通不过验证的数据，则可以看到错误信息如下：



关于消息框架的更多信息，可以查看官方文档：<https://docs.djangoproject.com/en/2.0/ref/contrib/messages/>。

4 创建自定义验证后端

Django允许对不同的数据来源采用不同的验证方式。在 `settings.py` 里有一个 `AUTHENTICATION_BACKENDS` 设置列出了项目中可使用的验证后端。其默认是：

```
['django.contrib.auth.backends.ModelBackend']
```

默认的 `ModelBackend` 通过 `django.contrib.auth` 后端进行验证，这对于大部分项目已经足够。然而我们也可以创建自定义的验证后端，用于满足个性化需求，比如LDAP目录或者来自于其他系统的验证。

关于自定义验证后端可以参考官方文档：

<https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#other-authentication-sources>。

每次使用内置的 `authenticate()` 函数时，Django会按照 `AUTHENTICATION_BACKENDS` 设置中列出的顺序，依次执行其中的验证后端进行验证工作，直到有一个验证后端返回成功为止。如果列表中的后端全部返回失败，则这个用户就不会被认证通过。

Django提供了一个简单的规则用于编写自定义验证后端：一个验证后端必须是一个类，至少提供如下两个方法：

- `authenticate()`：参数为 `request` 和用户验证信息，如果用户验证信息有效，必须返回一个 `user` 对象，否则返回 `None`。
`request` 参数必须是一个 `HttpRequest` 对象或者是 `None`
- `get_user()`：参数为用户的ID，返回一个 `user` 对象

我们来编写一个采用电子邮件（而不是 `username` 字段）和密码登录的验证后端，编写验证后端就和编写一个Python的类没有什么区别：

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

class EmailAuthBackend:
    """
    Authenticate using an e-mail address.
    """

```

```
def authenticate(self, request, username=None, password=None):
    try:
        user = User.objects.get(email=username)
        if user.check_password(password):
            return user
        return None
    except User.DoesNotExist:
        return None

def get_user(self, user_id):
    try:
        return User.objects.get(id=user_id)
    except User.DoesNotExist:
        return None
```

以上代码是一个简单的验证后端。`authenticate()` 方法接受 `request` 对象和 `username` 及 `password` 作为可选参数，这里可以用任何自定义的参数名称，我们使用 `username` 及 `password` 是为了可以与内置验证框架配合工作。两个方法工作流程如下：

- `authenticate()`：尝试使用电子邮件和密码获取用户对象，采用 `check_password()` 方法验证加密后的密码。
- `get_user()`：通过 `user_id` 参数获取用户ID，在会话存续Django会使用内置的验证后端去验证并取得 `User` 对象。

编辑 `settings.py` 文件增加：

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

在上边的设置里，我们将自定义验证后端加到了默认验证的后边。打开 <http://127.0.0.1:8000/account/login/>，注意 Django 尝试使用所有的验证后端，所以我们现在可以使用用户名或者电子邮件来登录，填写的信息会先交给

`ModelBackend` 进行验证，如果没有得到用户对象，就会使用我们的 `EmailAuthBackend` 进行验证。

`AUTHENTICATION_BACKENDS` 中的顺序很重要，如果一个用户信息对于多个验证后端都有效，Django会停止在第一个成功验证的后端处。

5 第三方认证登录

很多社交网站除了注册用户之外，提供了链接可以快速的通过第三方平台的用户信息进行登录，我们也可以为自己的站点添加例如Facebook，Twitter或Google的第三方认证登录功能。Python Social Auth是一个提供第三方认证登录的模块。使用这个模块可以让用户以第三方网站的信息进行登录，而无需先注册本网站的用户。这个模块的源码在 <https://github.com/python-social-auth>。

这个模块支持很多不同的Python Web框架，其中也包括Django，通过以下命令安装：

```
pip install social-auth-app-django==2.1.0
```

然后将应用名 `social_django` 添加到 `settings.py` 文件的 `INSTALLED_APPS` 设置中：

```
INSTALLED_APPS = [
    ...
    'social_django',
]
```

该应用自带了数据模型，所以需要执行数据迁移过程。执行之后可以在数据库中看到新增 `social_auth` 开头的一系列数据表。Python 的social auth模块具体支持的第三方验证服务，可以查看官方文档：<https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends>。

译者注：Facebook, Twitter和Google的第三方验证均通过OAuth2认证，而且操作方式基本相同。以下仅以Google为例子进行翻译：

需要先把第三方认证的URL添加到项目中，编辑 `bookmarks` 项目的根 `urls.py`：

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/', include('social_django.urls', namespace='social')),
]
```

一些网站的第三方验证接口不允许将验证后的地址重定向到类似 `127.0.0.1` 或者 `localhost` 这种本地地址，为了正常使用第三方验证服务，需要一个正式域名，可以通过修改Hosts文件。如果是Linux或macOS X下，可以编辑 `/etc/hosts` 加入一行：

```
127.0.0.1 mysite.com
```

这样会将 `mysite.com` 域名对应到本机地址。如果是Windows环境，可以在 `C:\Windows\System32\Drivers\etc\hosts` 找到 `hosts` 文件。

为了测试该设置是否生效，启动站点然后在浏览器中打开 <http://mysite.com:8000/account/login/>，会得到如下错误信息：

DisallowedHost at /account/login/

Invalid HTTP_HOST header: 'mysite.com:8000'. You may need to add 'mysite.com' to ALLOWED_HOSTS.

这是因为Djanog在 `settings.py` 中的 `ALLOWED_HOSTS` 设置中，仅允许对此处列出的域名提供服务，这是为了防止 HTTP请求头攻击。关于该设置可以参考官方文档： <https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>。

编辑 `settings.py` 文件然后修改 `ALLOWED_HOSTS` 为如下：

```
ALLOWED_HOSTS = ['mysite.com', 'localhost', '127.0.0.1']
```

在 `mysite.com` 之外，我们增加了 `localhost` 和 `127.0.0.1`，其中 `localhost` 是在 `DEBUG=True` 和 `ALLOWED_HOSTS` 留空情况下的默认值，现在就可以通过 <http://mysite.com:8000/account/login/> 正常访问开发网站了。

5.1 使用Google第三方认证

Google提供OAuth2认证，详细文档可以参考： <https://developers.google.com/identity/protocols/OAuth2>。

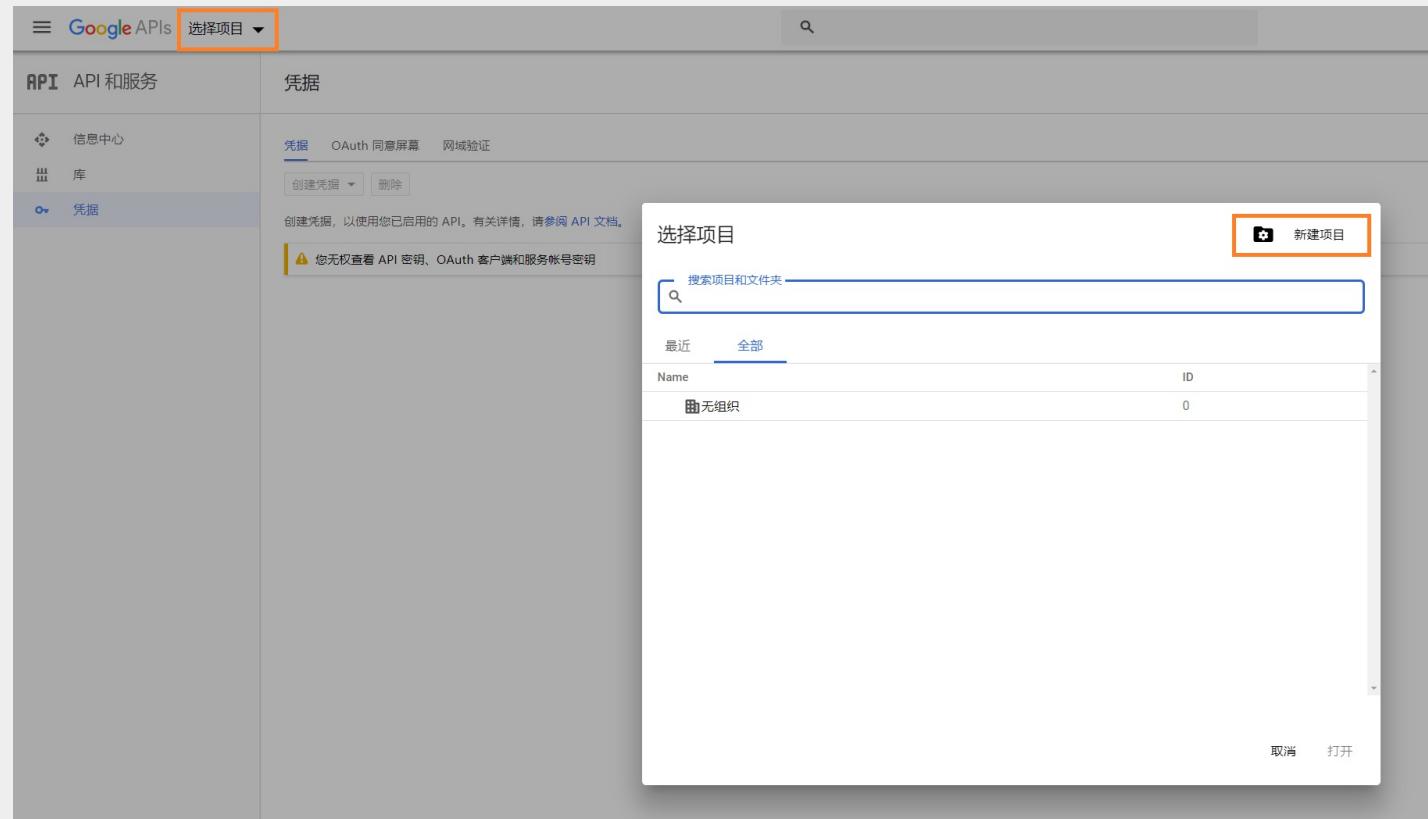
为使用Google的第三方认证服务，将以下验证后端添加到 `settings.py` 的 `AUTHENTICATION_BACKENDS` 中：

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
    'social_core.backends.facebook.FacebookAppOAuth2',
]
```

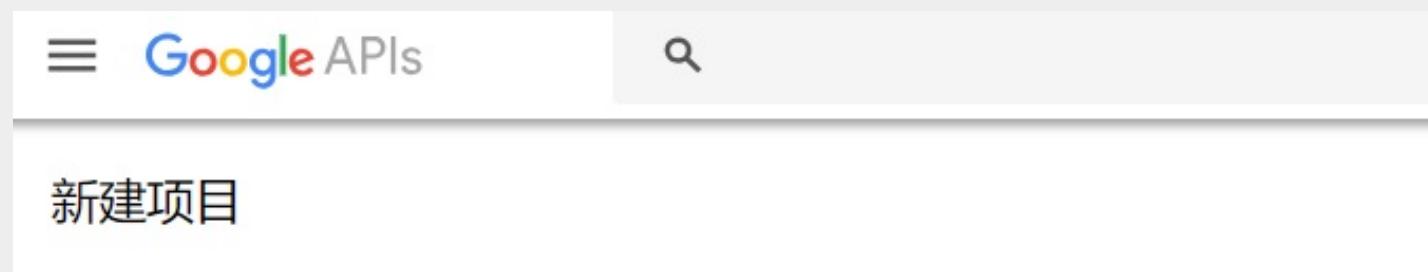
译者注：由于Google API的界面在原书成书后已经改变，以下在Google网站的操作步骤和截图来自于译者实际操作过程。

需要到Google开发者网站创建一个API key，按照以下步骤操作：

1. 打开 <https://console.developers.google.com/apis/credentials>，点击屏幕左上方 Google APIs 字样右边的 选择项目，会弹出项目对话框，点击右上方的新建项目，如图所示：



2. 填写新建项目的信息，项目名称为 Bookmarks，位置可以不选，之后点击创建按钮，如下图所示：



⚠ 您的配额中还剩 12projects。您可以申请提高配额或删除项目。

[了解详情](#)

MANAGE QUOTAS

项目名称 * ?

项目 ID: **bookmarks-219204**。它日后无法更改。 [修改](#)

位置 * 浏览

父级组织或文件夹

创建 [取消](#)

3. 之后与步骤1中的步骤类似，点开 **选择项目**，选中刚建立的 **Bookmarks** 项目，然后点击右下方的 **打开**。

4. 会自动跳转到一个页面提示尚未创建API凭据，点击页面中的 **创建凭据** 按钮，并选择第二项 **OAuth客户端ID**，如下图所示：

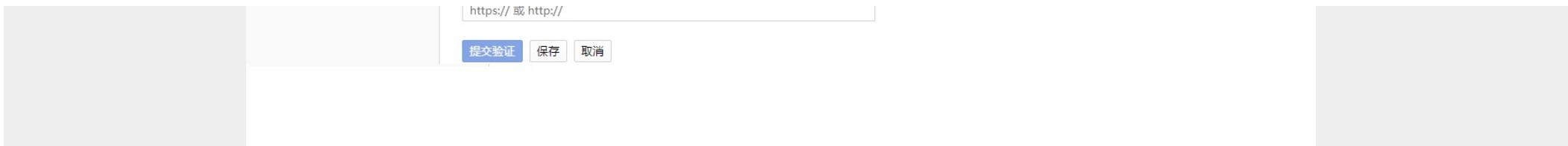
The screenshot shows the Google APIs console interface. On the left, there's a sidebar with 'API 和服务' (API Services) and three main categories: '信息中心' (Information Center), '库' (Library), and '凭据' (Credentials). The '凭据' category is currently selected and highlighted with a blue background. The main content area has a title '凭据' and three tabs at the top: '凭据' (selected), 'OAuth 同意屏幕', and '网域验证'. Below the tabs, there's a section titled 'API凭据' with a brief description: '您必须要有凭据才能使用 API。请启用您要使用的 API，然后创建 API 所需的凭据。您可能需要拥有 API 密钥、服务帐号或 OAuth 2.0 客户端 ID，具体取决于 API。有关详情，请参阅 API 文档。' A prominent blue button labeled '创建凭据' with a dropdown arrow is centered. A dropdown menu is open over this button, listing four options: 'API 密钥' (selected, shown in light blue), 'OAuth 客户端 ID' (highlighted in grey), '服务帐号密钥', and '帮我选择' (Help me choose). The 'OAuth 客户端 ID' option is described as '征得用户同意，让您的应用有权访问用户数据'.

5. 之后会进入一个界面，要求必须配置OAuth同意屏幕，如下图所示：

点击右侧的 **配置同意屏幕** 按钮。

6. 之后进入到OAuth同意屏幕，里边有一系列设置。在应用名称中填入 **Bookmarks**，默认 **支持电子邮件** 为你的电子邮件地址，可以修改为其他地址，在 **已获授权的网域** 中填入 **mysite.com**，之后点击保存，如图所示：

API 和服务	凭据
信息中心	凭据 OAuth 同意屏幕 网域验证
库	
凭据	<p>在您的用户进行身份验证之前，此同意屏幕可让他们选择是否要授予您对其私有数据的访问权限，同时向他们提供指向您的服务条款和隐私权政策的链接。此页面用于为这个项目中的所有应用配置同意屏幕。</p> <p>验证状态 未发布</p> <p>应用名称 ? 向用户征求同意的应用的名称 Bookmarks</p> <p>应用徽标 ? 同意屏幕上显示的一张图片，用以让用户认出您的应用 要上传的本地文件 浏览</p> <p></p> <p>支持电子邮件地址 ? 显示在同意屏幕上，用以提供用户支持服务 <input type="text"/></p> <p>Google API 的范围 范围让您的应用可以访问用户的私有数据。 了解详情 如果您添加敏感范围（例如向您赋予 Gmail 或云端硬盘的完整访问权限的范围），Google 会在您的同意屏幕发布之前先对其进行验证。</p> <p>email profile openid</p> <p>添加范围</p> <p>已获授权的网域 ? 为了保护您和您的用户，Google 仅允许那些通过 OAuth 进行身份验证的应用使用已获授权的网域。应用的链接必须托管在已获授权的网域上。 了解详情</p> <p>mysite.com  example.com</p> <p>应用首页链接 在同意屏幕上显示，必须托管在已获授权的网域上。 <input type="text"/> https:// 或 http://</p> <p>应用隐私权政策链接 在同意屏幕上显示，必须托管在已获授权的网域上。 <input type="text"/> https:// 或 http://</p> <p>应用服务条款链接 (可选) 在同意屏幕上显示，必须托管在已获授权的网域上。 <input type="text"/></p>



7. 此时会跳转到步骤5的问题页面，选择 **网页应用**，之后会被要求填写辅助信息，在**名称**中填写 **Bookmarks**，**已获授权的重定向 URI** 中填写 **http://mysite.com:8000/socialauth/complete/google-oauth2/**，如下图所示：

对于使用 OAuth 2.0 协议调用 Google API 的应用，您可以使用 OAuth 2.0 客户端 ID 生成访问令牌。该令牌包含唯一标识符。请参阅[设置 OAuth 2.0](#)以了解详情。

应用类型

网页应用
 Android [了解详情](#)
 Chrome 应用 [了解详情](#)
 iOS [了解详情](#)
 PlayStation 4
 其他

名称 [?](#)

Bookmarks

限制

输入 JavaScript 来源和/或重定向 URI[了解详情](#)

来源和重定向网域必须添加到 OAuth 同意设置中已获授权的网域列表中。

已获授权的 JavaScript 来源

适合搭配来自浏览器的请求使用。这是客户端应用的来源 URI，其中不得包含通配符 (`https://*.example.com`) 或路径 (`https://example.com/subdir`)。如果您使用的是非标准端口，则必须在来源 URI 中包含该端口。



8. 点击 **创建** 按钮，即可在页面中看到当前API的ID和密钥，如图所示：

The screenshot shows the Google Cloud Platform API library interface. On the left, the sidebar shows "API 和服务" (APIs & Services) selected. The main area displays the "凭据" (Credentials) section. Under "凭据", the "OAuth 2.0 客户端 ID" tab is selected. It lists one client entry: "名称" (Name) is empty, "创建日期" (Created at) is "2018年10月12日" (October 12, 2018), "类型" (Type) is "网页应用" (Web Application), and "客户端 ID" (Client ID) is "1064109389279-sdukdtigg3hue53bms76remvt8itr69k.apps.googleusercontent.com". A modal window titled "OAuth 客户端" (OAuth Client) is overlaid on the page. It contains a note: "您可以随时通过“API 与服务中的凭据”访问客户端 ID 和密钥" (You can随时通过“API 与服务中的凭据”访问客户端 ID 和密钥). Below this is a warning: "在发布 OAuth 同意屏幕之前，OAuth 的敏感范围登录数量将被限制为最多 100 个。发布此屏幕可能需要经由验证流程，而该流程需要数天的时间。" (Before publishing the OAuth consent screen, the number of sensitive range logins for OAuth will be limited to a maximum of 100. Publishing this screen may require a verification process, which may take several days.). The modal also shows the "客户端 ID" (Client ID) as "1064109389279-sdukdtigg3hue53bms" and the "客户端密钥" (Client Key) as "Qcx414TAnn0t". At the bottom right of the modal is a "确定" (OK) button.

9. 将API ID 和密钥填写到settings.py文件中，增加如下两行：

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'XXX' # API ID  
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXX' # 密钥
```

10. 点击 **确认** 关闭对话框，之后在左侧菜单的 **凭据** 菜单内可以回到此处查看ID和密钥。现在点击左侧菜单的 **库**，会跳转到欢迎使用新版API库的界面，在其中找到Google+ API，如图所示：

← API 库

G Suite (18)

Machine learning (7)

Maps (17)

Mobile (14)

Monitoring (4)

Networking (3)

Security (1)

Social (4)

Storage (4)

YouTube (4)

Other (6)

G Suite

查看全部 (18)



Google Drive API

Google

The Google Drive API allows clients to access resources from Google Drive



Google Calendar API

Google

Integrate with Google Calendar using the Calendar API.



Gmail API

Google

Flexible, RESTful access to the user's inbox



Google Sheets API

Google

The Sheets API gives you full control over the content and appearance of your spreadsheet

YouTube



YouTube Data API v3

Google

The YouTube Data API v3 is an API that provides access to YouTube data, such as videos,



YouTube Analytics API

Google

Retrieves your YouTube Analytics data.



YouTube Reporting API

Google

Schedules reporting jobs containing your YouTube Analytics data and downloads



YouTube Ads Reach API

Google

Retrieve YouTube ads reach per market, demographics, campaign budget and other

Social

查看全部 (4)



Google+ API

Google

The Google+ API enables developers to build on top of the Google+ platform.



Blogger API v3

Google

The Blogger API provides access to posts, comments and pages of a Blogger blog.



Google+ Domains API

Google

The Google+ Domains API enables developers to build on top of the Google+ platform for



Google People API

Google

Provides access to information about profiles and contacts.

11. 点击Google+ API，在弹出的页面中选择启用，如图所示：

The screenshot shows the Google API Library interface. At the top, there are navigation links for 'Google APIs' and 'Bookmarks', a search bar, and a user profile icon. Below the header, a breadcrumb trail says 'API 库'. The main content area features a circular icon with the Google+ logo. To its right, the title 'Google+ API' and the provider 'Google' are displayed. A brief description states: 'The Google+ API enables developers to build on top of the Google+ platform.' Below the description are two buttons: a blue '启用' (Enable) button and a white '试用此 API' (Try this API) button with a magnifying glass icon. On the left side of the main content, there's a sidebar with the following information:

类型	概览
API 和服务	The Google+ API enables developers to build on top of the Google+ platform.
上次更新日期	Google简介
2018/8/1 下午12:47	Google's mission is to organize the world's information and make it universally accessible and useful. Through products and platforms like Search, Maps, Gmail, Android, Google Play, Chrome and YouTube, Google plays a meaningful role in the daily lives of billions of people.
类别	教程和文档
社交	Learn more
服务名称	服务条款
plus.googleapis.com	使用此产品即表示您同意以下许可的条款及条件: Google APIs Terms of Service , Google+ API

在Google中的配置就全部结束了，生成了一个OAuth2认证的ID和密钥，之后我们就将采用这些信息与Google进行通信。

然后编辑 `account` 应用的 `registration/login.html` 模板，在 `content` 块的内部最下方增加用于进行Google第三方认证登录的链接：

```
<div class="social">
  <ul>
    <li class="google"><a href="{% url 'social:begin' 'google-oauth2' %}">Log in with Google</a></li>
  </ul>
</div>
```

打开 <http://mysite.com:8000/account/login/>，可以看到如下页面：

The screenshot shows a login interface. At the top, there's a green header bar with the word "Bookmarks" on the left and "Log-in" on the right. Below the header, the word "Log-in" is displayed in large, bold, black font. A horizontal line separates this from the main content area. In the content area, the text "Please, use the following form to log-in. If you don't have an account [register here](#)" is centered. Below this, there are two input fields: one for "Username" and one for "Password", both represented by light gray rectangular boxes. To the right of the password field is a red rectangular button with the text "Login with Google". At the bottom left of the form is a green rectangular button with the text "LOG-IN" in white.

点击Login with Google按钮，使用Google账户登录后，就会被重定向到我们网站的登录首页。

我们现在就为项目增加了第三方认证登录功能，即使是没有在本站注册的用户，也可以快捷的进行登录了。

译者注：这里有一个小问题，就是通过第三方登录进来的用户，检查 `auth_user` 表会发现其实用户信息已经被写入到了该表里，但是 `Profile` 表没有写入对应的外键字段，导致第三方认证用户在修改用户信息时会报错。很多网站的做法是：通过第三方验证进来的用户，必须捆绑到本站已经存在的账号中。这里我们简化一下处理，当用户修改字段的 `Get` 请求进来时，检测 `Profile` 表中该用户的外键是不是存在，如果不存在，就新建对应该用户的 `Profile` 对象，然后再用这个数据对象返回表单实例供填写。修改后的 `edit` 视图如下：

```
@login_required
def edit(request):
    if request.method == "POST":
        user_form = UserEditForm(instance=request.user, data=request.POST)
        profile_form = ProfileEditForm(instance=request.user.profile, data=request.POST, files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Profile updated successfully')
        else:
            messages.error(request, "Error updating your profile")
    else:
        try:
            Profile.objects.get(user=request.user)
        except Profile.DoesNotExist:
            Profile.objects.create(user=request.user)
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(instance=request.user.profile)

    return render(request, 'account/edit.html', {'user_form': user_form, 'profile_form': profile_form})
```

总结

这一章学习了使用内置框架快捷的建立用户验证系统，以及建立自定义的用户信息，还学习了为网站添加第三方认证。

下一章中将学习建立一个图片分享系统，生成图片缩略图，以及在Djanog中使用AJAX技术。

第五章 内容分享功能

在上一章我们使用内置验证框架迅速的建立了整个网站的用户相关功能，还学习了如何通过一对一段落扩展用户信息，以及为网站添加第三方认证登录功能。

这一章会学习使用JavaScript小书签程序，将其他网站的图片内容分享到本站，还将学习使用jQuery在Django中使用AJAX技术。本章包含如下要点

- 创建多对多关系
- 自定义表单行为
- 在Django中使用jQuery
- 创建jQuery小书签程序
- 使用sorl-thumbnail创建缩略图
- 使用jQuery发送AJAX请求和创建AJAX视图
- 创建视图的自定义装饰器
- AJAX动态加载页面

1 创建图片分享功能

我们的站点将让用户可以收藏然后分享他们在互联网上看到的图片到本站来，为此将要做以下工作：

- 用一个数据类存放图片和相关信息
- 建立表单和视图用于处理图片上传
- 需要建立一个系统，让用户将外站图片贴到本站来。

这是一个独立与用户验证系统的新功能，为此新建一个应用 `images`：

```
django-admin startapp images
```

然后在 `settings.py` 中激活该应用：

```
INSTALLED_APPS = [
    # ...
    'images.apps.ImagesConfig',
]
```

1.1 创建图片模型

编辑 `images` 应用的 `models.py` 文件，添加如下代码：

```
from django.db import models
from django.conf import settings

class Image(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, related_name='images_created', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, blank=True)
    url = models.URLField()
    image = models.ImageField(upload_to='images/%Y/%m/%d')
    description = models.TextField(blank=True)
    created = models.DateTimeField(auto_now_add=True, db_index=True)

    def __str__(self):
        return self.title
```

这是我们用于存储图片的模型，来看一下具体的字段：

- `user`：这是一个连接到 `User` 模型的外键，体现了用户与图片的一对多关系，即一个用户可以上传多个图片。
- `title`：图片的名称
- `slug`：该图片的简称，用于动态建立该图片的URL
- `image`：图片文件字段，用于存放图片
- `description`：可选的关于图片的描述
- `created`：图片分享到本站来的时间，使用了 `auto_now_add` 自动生成创建时间，并且使用了 `db_index=True` 创建索引

数据库索引可以有效的提高数据库查询效率。对于频繁使用 `filter()`, `exclude()` 或者 `order_by()` 等方法的字段推荐创建字段。`ForeignKey` 和设置了 `unique=True` 的字段默认会被创建索引。还可以使用 `Meta.index_together` 创建联合索引。

译者注：为 `created` 字段创建索引是常用做法。

这里我们需要自定义该模型的行为，重写 `Image` 模型的 `save()` 方法，使图片在保存到数据库时，自动根据 `title` 字段生成 `slug` 字段的内容。导入 `slugify()` 然后为 `Image` 模型添加一个 `save()` 方法：

```
from django.utils.text import slugify

class Image(models.Model):
    # .....

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Image, self).save(*args, **kwargs)
```

译者注：原书代码缩进有误，此处已经修改为正确版本。

在这段代码里，使用了Django内置的 `slugify()` 自动生成了 `slug` 字段的内容。之后调用超类的方法保存图片，这样用户无需手工输入。

1.2 创建多对多关系

我们将在 `Image` 模型中再添加一个外键，用于存储哪些用户喜欢该图片。由于一个用户可能喜欢多个图片，一个图片也可能被多个用户喜欢，因此图片和用户之间多对多的关系，需要修改 `Image` 模型添加如下字段：

```
users_like = models.ManyToManyField(settings.AUTH_USER_MODEL, related_name='images_liked', blank=True)
```

当定义了 `ManyToManyField` 多对多外键字段时，Django会创建一张中间表，中间表分别通过外键关联到当前的模型和 `ManyToManyField()` 的第一个参数对应的模型，多对多关系可以用于任意两个有关系的模型。

与 `ForeignKey` 一样，`related_name` 属性定义了多对多字段反向查询的名称，多对多字段提供了一个多对多模型管理器来进行查询，类似 `image.users_like.all()`，如果是从 `user` 对象查询，则类似 `user.images_liked.all()`。

之后进行 `Image` 类的数据迁移。

1.3 添加图片模型至管理后台

编辑 `images` 应用的 `admin.py` 文件，将 `Image` 类添加至管理后台：

```
from django.contrib import admin
from .models import Image

@admin.register(Image)
class ImageAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'image', 'created']
    list_filter = ['created']
```

启动站点，打开 <http://127.0.0.1:8000/admin/>，可以看到 `Image` 已经被加入管理后台，如图所示：

2 从外站分享内容至本站

我们实现用户将外站图片分享到本站的方式是：用户提供图片的URL，一个标题和可选的秒数，我们的站点会将该图片下载下来，建立一个对应的新 `Image` 对象，然后保存进数据库。

已经建立完了图片模型，这里我们需要建立一个表单供用户提交图片信息。在 `Images` 应用下建立 `forms.py` 文件，然后添加如下代码：

```
from django import forms
from .models import Image

class ImageCreateForm(forms.ModelForm):
    class Meta:
        model = Image
        fields = ('title', 'url', 'description',)
        widgets = {
            'url': forms.HiddenInput,
        }
```

这里使用了 `ModelForm` 类，基于 `Image` 模型创建了表单，仅包含 `title`，`url` 和 `description` 字段。用户无需直接在表单中输入图片URL，我们将使用一个JavaScript小书签程序来从外站选择一个图片并将其URL作为 `Get` 请求的参数，然后访问我们的站点。所以我们使用了 `HiddenInput` 小插件替代了默认的 `url` 字段的设置。我们这么做是希望这个字段不被用户看到。

2.1 验证表单字段

为了验证这个URL是一个图片，需要检查URL中的文件名是否以 `.jpg` 或 `.jpeg` 扩展名结尾。像在之前章节那样，我们将针对 `url` 字段编写一个自定义验证器 `clean_url()`，这样表单对象调用 `is_valid()` 时，我们的验证器就可以修改数据或者报错。添加如下方法到 `ImageCreateForm`：

```
def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['jpg', 'jpeg']
    extension = url.rsplit('.', 1)[1].lower()
    if extension not in valid_extensions:
        raise forms.ValidationError('The given URL does not match valid image extensions.')
    return url
```

在上边的代码中，定义了 `clean_URL()` 方法来验证 `url` 字段，该方法解释如下：

1. 从 `cleaned_data` 中获取 `url` 字段的值
2. 将URL通过从右边开始的第一个 `.` 进行切分，然后取切分结果的第二个元素，也就是扩展名进行比较。如果验证失败，则抛出一个 `ValidationError` 错误。这里我们采用的验证方式比较简陋，而且仅支持 `jpg` 类型图片，你可以采用正则表达式或者其他高级方法来验证URL是否是一个有效的图片文件地址。
- 3.

除了验证URL之外，我们还必须在验证成功的时候将图片下载并保存到数据库中。我们可以使用处理该表单的视图来完成这个操作，但更常用的方式是重写表单的 `save()` 来实现此功能。

2.2 重写表单的 `save()` 方法

在之前已经知道，`ModelForm` 有一个 `save()` 方法，将当前的模型数据存储到数据库中并且返回该对象。这个方法还接受一个 `commit` 布尔值参数，用于确定是否实际将数据持久化到数据库中。如果 `commit=False`，则 `save()` 方法仅返回当前的数据对象，但不执行数据库写入操作。因此我们可以重写 `save()` 方法，让其下载图片之后，再将数据对象写入数据库。

添加如下导入语句到 `forms.py` 文件：

```
from urllib import request
from django.core.files.base import ContentFile
from django.utils.text import slugify
```

之后添加下列 `save()` 方法至 `ImageCreateForm` 类中：

```
def save(self, force_insert=False, force_update=False, commit=True):
    image = super(ImageCreateForm, self).save(commit=False)
    image_url = self.cleaned_data['url']
    image_name = '{}.{}'.format(slugify(image.title), image_url.rsplit('.', 1)[1].lower())

    # 根据URL下载图片
    response = request.urlopen(image_url)
    image.image.save(image_name, ContentFile(response.read()), save=False)

    if commit:
        image.save()
    return image
```

我们重写了 `save()` 方法，保持与原来方法一样的默认参数设置。重写的方法工作逻辑如下：

1. 先调用父类的 `save()` 方法，使用现有表单数据建立一个新的 `image` 数据对象但不保存
2. 从 `cleaned_data` 中获取URL
3. 将 `image.slug` 与扩展名拼成新的文件名
4. 使用Python的 `urllib` 模块下载图片，然后使用 `image` 字段的 `save()` 方法保存到 `MEDIA` 目录中。`image` 字段的 `save()` 方法的参数之一 `ContentFile` 是下载的图片内容，这里使用了 `save=False` 防止直接将字段写入数据库。
5. 为了和原 `save()` 方法的行为保持一致，仅当 `commit=True` 的时候写入数据库。

译者注：本章到现在为止出现了模型的 `save()` 方法，表单的 `save()` 方法和 `image` 字段的 `save()` 方法，读者不要混淆。

之后来编写处理表单的视图，编辑 `images` 应用的 `views.py` 文件，添加如下代码：

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .forms import ImageCreateForm

@login_required
def image_create(request):
    if request.method == "POST":
        # 表单被提交
        form = ImageCreateForm(request.POST)
        if form.is_valid():
            # 表单验证通过
            cd = form.cleaned_data
            new_item = form.save(commit=False)
            # 将当前用户附加到数据对象上
            new_item.user = request.user
            new_item.save()
            messages.success(request, 'Image added successfully')
```

```
# 重定向到新创建的数据对象的详情视图
    return redirect(new_item.get_absolute_url())
else:
    # 根据GET请求传入的参数建立表单对象
    form = ImageCreateForm(data=request.GET)

return render(request, 'images/image/create.html', {'section': 'images', 'form': form})
```

使用`@login_required`装饰器令`image_create`视图仅供登录后的用户使用，这个视图工作逻辑如下：

1. 我们通过一个`Get`请求附加的参数创建表单对象，参数会带着`url`和`title`字段对应的内容。这个`Get`请求是由之后我们创建的JavaScript小书签程序发起的，现在，我们就假设该表单已经被初始化而且被用户确认并提交。
2. 表单提交后，如果验证通过，那么建立一个新的`Image`对象，但是不存入数据库。
3. 取得当前的用户，赋给`Image`对象的外键后进行保存，这样就可以知道该图片由哪个用户上传。
4. 将图片写入数据库。
5. 创建一个成功保存图片的消息，然后将用户重定向到规范化的图片对象的URL，现在还没有为`Image`模型创建`get_absolute_url()`方法，稍后会进行创建。

在`images`应用中建立`urls.py`文件，添加如下代码：

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

app_name = 'images'

urlpatterns = [
    path('create/', views.image_create, name='create'),
]
```

然后编辑bookmarks项目的根`urls.py`文件，为`images`应用增加一条二级路由匹配：

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/', include('social_django.urls', namespace='social')),
    path('images/', include('images.urls', namespace='images')),
]
```

最后来建立对应的模板，在images应用的目录下创建如下目录和文件结构：

```
templates/
  images/
    image/
      create.html
```

然后编辑刚刚创建的 `create.html` 文件，添加如下代码：

```
{# create.html #-}
{% extends "base.html" %}
{% block title %}Bookmark an image{% endblock %}
{% block content %}
  <h1>Bookmark an image</h1>
  
  <form action"." method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <input type="submit" value="Bookmark it!">
  </form>
{% endblock %}
```

现在启动站点，输入类似 `http://127.0.0.1:8000/images/create/?title=...&url=...` 的链接，其中包含 `title` 和 `url` 两个参数，分别表示图片的名称和URL地址。可以使用下边这个测试地址：

http://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=http://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg

应该可以看到下面的页面：

Bookmark an image



Title:

Django and Duke

Description:

BOOKMARK IT!

在description内输入一些内容，然后点击BOOKMARK IT!按钮，一个新的Image对象会被存入数据库。由于此时 `get_absolute_url()` 方法还未编写，所以会报错如下：

AttributeError at /images/create/

'Image' object has no attribute 'get_absolute_url'

此时不用担心这个错误信息，通过刚才编写的视图可以知道，执行到这里报错说明图片已经成功存入数据库，打开<http://127.0.0.1:8000/admin/images/image/> 即可看到该图片的信息，如下图所示：

Action:	-----	Go	0 of 1 selected	
<input type="checkbox"/>	TITLE	SLUG	IMAGE	CREATED
<input type="checkbox"/>	Django and Duke	django-and-duke	images/2017/11/05/django-and-duke.jpg	Dec. 16, 2017

2.3 使用jQuery创建小书签程序

[小书签程序](#) 是一段JavaScript代码，可以被浏览器保存为书签，在点击该小书签时，其中的JavaScript代码被执行，从而实现一些功能。

一些比较知名的站点，如Pinterest，使用小书签程序让用户可以从其他网站将内容分享到其网站上。我们建立的程序和这个小书签程序类似，让用户将图片分享到我们的站点来。

我们将使用jQuery建立小书签程序，jQuery是一个得到广泛使用的JavaScript库，可以快速开发基于JavaScript的程序，可以访问其官方站点<https://jquery.com/> 了解更多信息。

用户将会这样使用我们的小书签：

1. 用户将我们网站上的一个链接拖到浏览器的书签栏中，这个链接的 `href` 属性中保存着JS代码，这个链接被保存到浏览器书签成为一个可点击的书签
2. 用户在其他网站上看到想分享的图片，点击这个小书签，小书签里边的程序被运行，让用户选择要分享的图片然后自动以GET请求访问我们的网站。

由于小书签程序保存在用户的浏览器上，在用户第一次保存后，想要更新该程序就很困难，所以一般小书签程序实际上是一个程序启动器，实际执行的程序位于我们的网站上。这就是我们创建小书签的方法解说，现在来实现：

在 `images/templates/` 目录下创建一个文件，叫做 `bookmarklet_launcher.js`，添加如下JavaScript代码：

```
(function () {
    if (window.myBookmarklet !== undefined) {
        myBookmarklet()
    }
    else {
        document.body.appendChild(document.createElement('script')).src = 'http://127.0.0.1:8000/static/js/bookmarklet_launcher.js?r=' + Math.random();
    }
})();
```

这段JavaScript代码首先检查 `myBookmarklet` 这个名称是否存在于当前环境，这样用户反复点击小书签程序也不会多次运行相同程序。如果名称不存在，就在当前的页面中增加一个 `<script>` 标签，也就是导入了我们网站的一段JavaScript程序并且执行。之后的r参数生成了一段随机数，目的是让浏览器每次都去请求实际的JavaScript文件，而不从缓存中直接读取。

新增的 `<script>` 标签的src属性为 `"http://127.0.0.1:8000/static/js/bookmarklet_launcher.js?r=xxxxxxxxxxxxxxxxxxxx"`，指向我们网站自己的JavaScript程序文件，这样小程序每次执行的时候，都会将我们网站上的JavaScript程序在当前页面

执行。下边我们把小程序链接加入到用户登录首页，以让用户可以将其保存成书签。

这就是一个启动器，用于加载实际上位于我们站点上的bookmarklet.js然后在当前页面运行。

编辑 account 应用的模板目录中的 `account/dashboard.html`，让其看起来像下边这样：

```
{% extends "base.html" %}  
{% block title %}Dashboard{% endblock %}  
{% block content %}  
    <h1>Dashboard</h1>  
  
    {% with total_images_created=request.user.images_created.count %}  
        <p>Welcome to your dashboard. You have bookmarked {{ total_images_created }} image{{ total_images_created|pluralize }}.{% endwith %}  
  
        <p>Drag the following button to your bookmarks toolbar to bookmark images from other websites <a href="">Bookmarklet</a>.</p>  
  
        <p>You can also <a href="{% url "edit" %}">edit your profile</a> or <a href="{% url "password_change" %}">change your password</a>.  
    {% endblock %}
```

现在首页已经当前用户已经分享了多少图片到本站，使用了 `{% with %}` 标签用于设置一个变量名给图片总数，可以避免反复查询数据库。然后包含了一个 `href` 属性是小标签启动器程序的链接，供用户将其拖动到浏览器的书签栏上。这里使用了 `include` 将JavaScript文件的内容导入。

译者注：这里灵活使用了 `include` 标签，可见引入的模板文件不需要是HTML文件，只要是文本文件即可，这里就通过该标签将 `bookmarklet_launcher.js` 文件引入，避免了在此处硬编码JavaScript代码。

在浏览器中打开 <http://127.0.0.1:8000/account/>，可以看到如下页面：

Dashboard

Welcome to your dashboard. You have bookmarked 1 image.

Drag the following button to your bookmarks toolbar to bookmark images from other websites → [BOOKMARK IT](#)

You can also [edit your profile](#) or [change your password](#).

现在开始来编写实际执行的JavaScript程序，在images应用下建立如下目录和文件结构：

```
static/  
  js/  
    bookmarklet.js
```

在随书代码中可以看到 `images` 应用目录下有 `static/css/` 目录，将其中的 `css/` 目录拷贝到你的应用的 `static/` 目录下，小书签程序将要使用其中的 `bookmarklet.css` 文件。

打开刚建立的 `bookmarklet.js` 文件，添加如下代码：

```
(function () {  
  let jquery_version = '3.3.1';  
  let site_url='http://127.0.0.1:8000/';  
  let static_url = site_url + 'static/';  
  let min_width = 100;
```

```
let min_height = 100;
function bookmarklet(msg){
    //这里是分享图片的代码
}

// 检查页面是否加载了jQuery，如果没有就进行加载，尝试15次
if(typeof window.jQuery !== 'undefined'){
    bookmarklet();
}
else {
    let conflict = typeof window.$ !== 'undefined';
    let script = document.createElement('script');
    script.src = '//ajax.googleapis.com/ajax/libs/jquery/' + jquery_version + '/jquery.min.js';
    document.head.appendChild(script);
    let attempts = 15;
    (function(){
        if(typeof window.jQuery === 'undefined'){
            if(--attempts>0){
                window.setTimeout(arguments.callee, 250)
            }else {
                alert("An error occurred while loading jQuery")
            }
        }else {
            bookmarklet()
        }
    })();
}
})();
```

这是加载jQuery的代码。如果jQuery已经在当前页面加载，则会使用当前页面的jQuery，如果没有加载，则将jQuery位于google的CDN地址加入到页面中。当jQuery被成功加载的时候，就去执行 `bookmarklet()` 函数，该函数含有实际的分享图片代码。在文件开始的地方还定义了如下几个全局变量：

- `jquery_version` : jQuery的版本号
- `site_url` 和 `static_url` : 我们网站的地址和静态文件地址
- `min_width` 和 `min_height` : 用于控制程序寻找的最小图片宽高，小于这个宽或高的图片不会出现在供分享的清单中。

现在来编写 `bookmarklet()` 函数，编辑文件里的 `bookmarklet()` 函数的代码如下：

```
function bookmarklet(msg){  
    // 加载CSS文件  
    let css = jQuery('<link>');  
    css.attr({  
        rel:'stylesheet',  
        type:'text/css',  
        href:static_url + 'css/bookmarklet.css?r=' + Math.floor(Math.random()*9999999999999999)  
    });  
    jQuery('head').append(css);  
  
    // 加载HTML结构  
    box_html = '<div id="bookmarklet"><a href="#" id="close">x</a><h1>Select an image to bookmark:</h1><div class="image-list"></div>'  
    jQuery('body').append(box_html);  
  
    // 关闭事件  
    jQuery('#bookmaklet #close').click(function () {  
        jQuery("#bookmarklet").remove();  
    });  
};
```

这段代码的逻辑如下：

1. 加载 `bookmarklet.css`，使用随机数确保浏览器不从缓存中读取
2. 加入一块HTML结构代码到当前页面的 `<body>` 标签中，在页面的右上方显示一个浮动的图片列表区域

3. 加入了一个事件，用户点击新增的区域的关闭按钮时，将我们添加的HTML结构代码从当前页面中删除。使用jQuery，通过父元素ID为 bookmarklet 的 #bookmarklet 和 #close 选择器定位我们的HTML元素。关于jQuery的选择器，可以参考 <https://api.jquery.com/category/selectors/>。

在加载了HTML结构和对应的CSS样式后，接下来要添加分享功能，将如下代码追加在 `bookmarklet()` 函数的内部：

```
// 寻找页面内所有图片然后显示在新增的HTML结构中
jQuery.each(jQuery('img[src$="jpg"]'), function(index, image) {
  if (jQuery(image).width() >= min_width && jQuery(image).height() >= min_height)
  {
    image_url = jQuery(image).attr('src');
    jQuery('#bookmarklet .images').append('<a href="#"></a>');
  }
});
```

这段代码使用了 `img[src$="jpg"]` 选择器来选择所有jpg格式的 `` 元素，然后使用 `each()` 方法，对其中每个图片检查是否大于最小宽高，如果大于就将其加入到我们HTML结构的 `<div class="images">` 标签中。

在开始试验编写的功能之前，还必须进行最后的设置。现在HTTPS协议使用的很广泛，为了安全起见，浏览器一般不会允许HTTP协议的小书签程序运行，因此必须给我们自己的网站一个HTTPS地址，但是Django的测试服务器无法自动支持HTTPS，为了测试小书签的功能，使用 [Ngrok](#) 可以建立一个隧道将自己的本机通过HTTP和HTTPS地址向外提供服务。

在 <https://ngrok.com/download> 下载Ngrok，之后在系统命令行里运行如下命令：

```
./ngrok http 8000
```

Ngrok建立一个隧道连接到本机的8000端口，然后为其分配一个域名，可以看到窗口里显示：

```
ngrok by @inconshreveable
```

```
(Ctrl+C to copy)
```

```
Session Status          online
Session Expires        7 hours, 58 minutes
Version                2.2.8
Region                 United States (us)
Web Interface          http://127.0.0.1:4040
Forwarding             http://d0de3ca5.ngrok.io -> localhost:8000
Forwarding             https://d0de3ca5.ngrok.io -> localhost:8000

Connections            ttl     opn     rt1     rt5     p50     p90
                        0       0       0.00   0.00   0.00   0.00
```

其中的 `https://d0de3ca5.ngrok.io` 就是可以访问到本机Django服务的HTTPS地址，把这个地址加入到 `settings.py` 文件的 `ALLOWED_HOSTS` 里：

```
ALLOWED_HOSTS = [
    'mysite.com',
    'localhost',
    '127.0.0.1',
    'd0de3ca5.ngrok.io'
]
```

译者注：最好按照Ngrok官网的教程注册一个用户再使用，否则HTTPS的域名很快过期，需要重新启动Ngrok并进行相关配置。

启动站点，然后访问这个HTTPS地址，应该可以看到站点的登录页面，说明HTTPS服务正常。

获得HTTPS地址之后，编辑 `bookmarklet_launcher.js` 文件，将其中的 `http://127.0.0.1:8000/` 替换为新获得的 HTTPS地址：

```
(function () {
    if (window.myBookmarklet !== undefined) {
        myBookmarklet()
    }
    else {
        document.body.appendChild(document.createElement('script')).src = 'https://d0de3ca5.ngrok.io/static/js/bookmarklet.js';
    }
})();
```

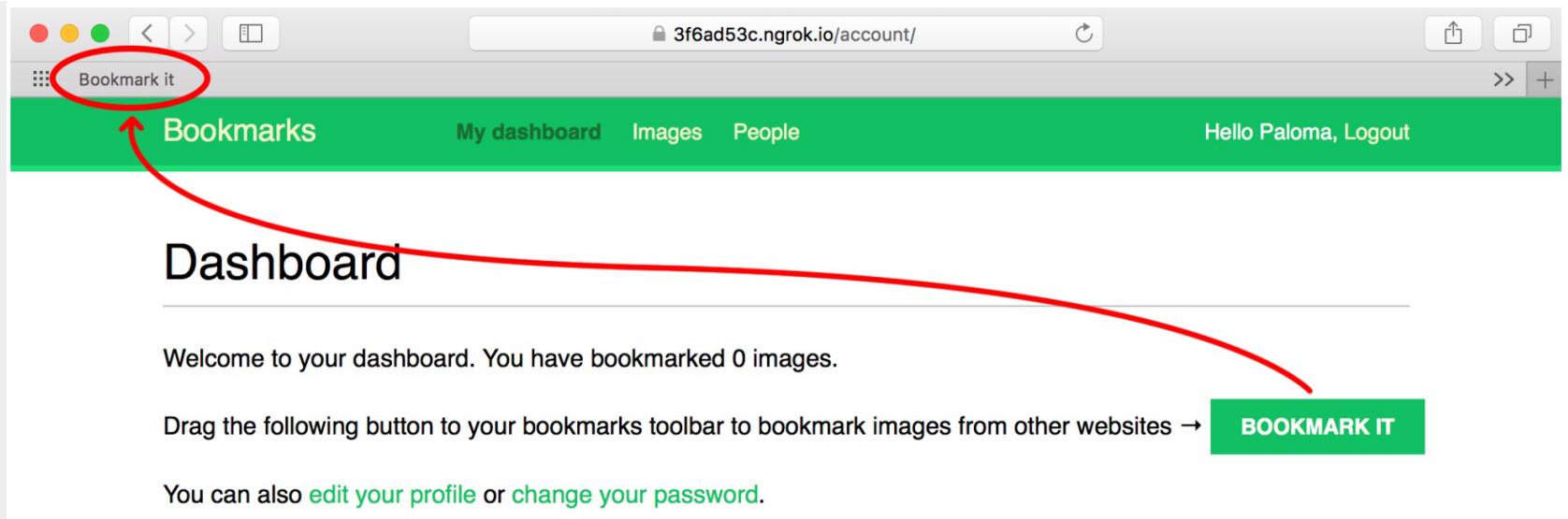
再将 `js/bookmarklet.js` 文件中的这一行：

```
let site_url='http://127.0.0.1:8000/';
```

修改为：

```
let site_url='https://d0de3ca5.ngrok.io/';
```

然后打开 <https://d0de3ca5.ngrok.io/account/>，将页面上的BOOKMART IT的绿色按钮拖到浏览器的书签栏上，如图所示：



打开任意一个图片比较多的网站，点击小书签，应该可以看到屏幕右上方显示一块新区域，里边列出了当前站点可供分享的图片，如下所示：

NEW & INTERESTING FINDS ON AMAZON

EXPLORE

Select an image to bookmark:

1-16 of 11,256 results for "django reinhardt"

Show results for

CDs & Vinyl

- Jazz
- Pop
- European Jazz
- Swing Jazz
- Gypsy Music
- [See more](#)

Books

- Jazz Music
- [See more](#)

[See All 21 Departments](#)

Refine by

International Shipping (What's this?)

Ship to Spain

Amazon Prime

prime

Eligible for Free Shipping

music unlimited

Introducing Amazon Music Unlimited. Listen to any song, anywhere.

[Learn More about Amazon Music Unlimited](#)

Showing most relevant results. See all results for django reinhardt.

Sponsored ⓘ

Solo Flight: The Music of Django Reinhardt by J.P. McShane and Django Reinhardt

\$15⁰⁰ prime

FREE Shipping on eligible orders
Only 2 left in stock - order soon.

The Essential DJANGO REINHARDT **The Essential Django Reinhardt** Mar 15, 2011

我们希望用户点击一张图片，就可以将该图片分享到我们的网站，进入之前编写的视图对应的表单填写页面上，编辑 `js/bookmarklet.js` 文件，在 `bookmarklet()` 函数底部追加：

```
// 点击图片时按照指定URL访问我们的网站
jQuery('#bookmarklet .images a').click(function(e){
    let selected_image = jQuery(this).children('img').attr('src');
    // hide bookmarklet
```

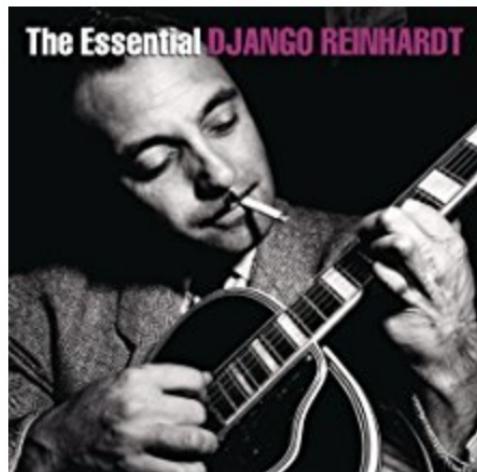
```
jQuery('#bookmarklet').hide();
// open new window to submit the image
window.open(site_url +'images/create/?url='
            + encodeURIComponent(selected_image)
            + '&title='
            + encodeURIComponent(jQuery('title').text()),
            '_blank');
});
```

这个函数的逻辑如下：

1. 为每个图片元素绑定一个 `click()` 事件
2. 当用户点击一个图片时，设置一个变量 `selected_image`，是这个图片的URL地址。
3. 之后隐藏新增的HTML结构，使用 `selected_image` 和网站的 `<title>` 的内容外加我们的网站地址，生成一个链接然后在新窗口中打开链接，实现GET请求附带参数访问我们自己的网站。

打开一个网站，然后点击小书签，在右上方出现的窗口中点击一张图片，会被重定向到我们网站的图片创建页面，如下所示：

Bookmark an image



Title:

Django Reinhardt

Description:

BOOKMARK IT!

撒花庆祝，我们实现了第一个小书签程序，然后将其集成到了我们的Django项目中。

3 创建图片详情视图

完成了图片分享并保存的功能之后，现在需要建立一个详情视图用来展示具体图片，编辑 `images` 应用的 `views.py` 文件，添加如下代码：

```
from django.shortcuts import get_object_or_404
from .models import Image

def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    return render(request, 'images/image/detail.html', {'section': 'images', 'image': image})
```

这是一个简单的用于展示某个图片详情的视图，编辑 `images` 应用的 `urls.py` 文件为该视图添加一行URL：

```
path('detail/<int:id>/<slug:slug>/', views.image_detail, name='detail'),
```

有过上个项目的经验，此时可以知道必须编写 `Image` 类的 `get_absolute_url()` 方法用于生成规范化链接，打开 `images` 应用的 `models.py` 文件，添加 `get_absolute_url()` 方法如下：

```
from django.urls import reverse

class Image(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('images:detail', args=[self.id, self.slug])
```

记住在每个编写的模型中加入该方法，以快捷的生成对应的URL。

译者注：在django 2里， urls.py 文件中使用 `include()` 方法并通过 `namespace` 参数指定命名空间，还需要在对应的下一级 urls.py 里写上 `app_name = 'namespace'` 来设置命名空间。如果 `include()` 方法中设置了命名空间，其对应的 urls.py 文件中的 `app_name` 必须一致，否则会报错。如果 `include()` 方法未设置命名空间，则以 `app_name` 的设置为准。

最后就是建立模板了，在 images 应用的模板目录中的 `/images/image/` 路径下创建 `detail.html` 文件并添加如下代码：

```
{#/templates/images/image/detail.html#}
{% extends 'base.html' %}

{% block title %}
    {{ image.title }}
{% endblock %}

{% block content %}
    <h1>{{ image.title }}</h1>
    
    {% with total_likes=image.users_like.count %}
        <div class="image-info">
            <div>
                <span class="count">
                    {{ total_likes }} like{{ total_likes|pluralize }}
                </span>
            </div>
            {{ image.description|linebreaks }}
        </div>
        <div class="image-likes">
            {% for user in image.users_like.all %}
                <div>
                    
                    <p>{{ user.first_name }}</p>
                </div>
            {% empty %}
        
```

```
Nobody likes this image yet.  
    {% endfor %}  
  </div>  
  {% endwith %}  
{% endblock %}
```

这是展示具体某个图片的模板，其中使用 `{% with %}` 保存查询结果到 `total_likes` 变量中避免了查询两次数据库。然后展示图片的 `description` 字段，之后迭代 `image.users_like.all`，显示出所有喜欢该图片的用户。

在一个模板中反复使用某一个QuerySet时，可以通过 `{% with %}` 将其查询结果保存到一个变量中，避免重复查询。

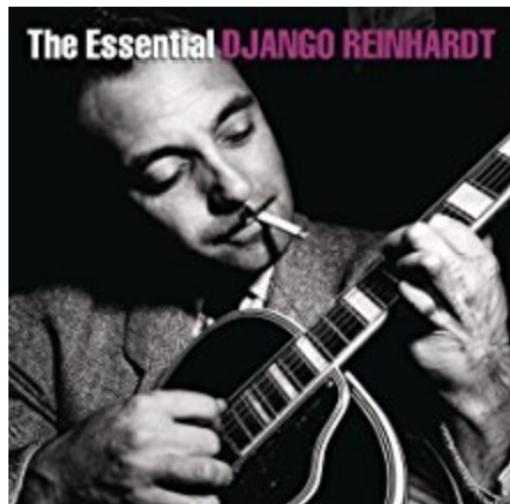
译者注：`image.image.url` 和 `user.profile.photo.url`：这两个字段不是 `Image` 类中的 `url` 字段，而是在定义 `Imagefield` 字段时 `upload_to` 的路径名称。

现在可以通过小书签程序再导入一个新图片，保存成功之后，会被重定向到图片的详情页，如下所示：

Image added successfully

X

Django Reinhardt



0 likes

The Essential Django Reinhardt.

Nobody likes this image yet.

4 创建图片缩略图

现在我们的图片详情页展示的是原始的图片，但是图片的尺寸可能差异很大，而且原始图片的大小可能会很大，载入时间较长。一般网站需要大量展示图片的通用做法是生成图片的缩略图然后展示缩略图。我们使用一个第三方应用 `sorl-thumbnail` 来生成缩略图。

在系统命令行中输入以下命令安装 `sorl-thumbnail`：

```
pip install sorl-thumbnail==12.4.1
```

然后在 `settings.py` 文件中激活该应用：

```
INSTALLED_APPS = [
    # ...
    'sorl.thumbnail',
]
```

之后按照惯例执行数据迁移程序，可以看到数据库中增添了该应用的一个数据表。

这个模块采用了两种方法显示缩略图：一是提供了新的模板标签 `{% thumbnail %}` 直接在模板内显示缩略图，二是基于 `Imagefield` 自定义的图片字段，用于在模型内设置缩略图字段。这两种方式都可以显示缩略图。

我们采用模板标签的方式。编辑 `images/image/detail.html`，找到如下这行：

```

```

将其替换成下列代码：

```
{% load thumbnail %}
{% thumbnail image.image "300" as im %}
```

```
<a href="{{ image.image.url }}>
    
</a>
{% endthumbnail %}
```

这里我们定义了个固定宽度为300像素的缩略图，当用户第一次打开图片详情页时，一个缩略图会被创建在静态文件夹下，页面的原图片链接会被缩略图链接所代替。启动站点然后打开某个图片详情页，可以在项目根目录的 `media/cache/` 找到该图片对应的缩略图。

`sorl-thumbnail` 可以使用很多算法生成各种缩略图。如果生成不了缩略图，在 `settings.py` 里增加一行 `THUMBNAIL_DEBUG=True`，之后在命令行窗口中可以看到debug信息。具体文档可以看 <https://sorl-thumbnail.readthedocs.io/>。

5 使用jQuery发送AJAX请求

现在要给站点增加AJAX相关功能，AJAX是Asynchronous JavaScript and XML的简称，这个技术使用一系列方式实现异步HTTP请求，可以从服务器异步取得数据并无需重载全部页面。不像名字里边必须采取XML格式，发送和收取数据可以采用JSON，HTML甚至纯文本。

AJAX的相关内容可以参考 [在Django中使用jQuery发送AJAX请求](#) 和 [使用原生JS发送AJAX请求的方法](#)。

我们将要给图片详情页面增加一个按钮，让用户可以点击该按钮表示喜欢该图片，之后再点击该按钮可以取消喜欢该图片。首先我们先为这个功能建立视图函数，编写 `images` 应用的 `views.py` 文件，添加如下代码：

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

@login_required
@require_POST
```

```
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == "like":
                image.users_like.add(request.user)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status': 'ok'})
        except:
            pass
    return JsonResponse({'status': 'ko'})
```

这个视图使用了两个装饰器，`@login_required`的作用是仅供已登录用户使用，`@require_POST`的作用是让该视图仅接受`POST`请求，否则返回一个`HttpResponseNotAllowed`对象，即HTTP 405错误。Django还提供了一个`@require_GET`装饰器用于只接受`GET`请求，还提供了一个`@require_http_methods`装饰器，可以指定允许哪些类型的HTTP请求。

在这个视图中，我们还是用了两个`Post.get`取得数据：

- `image_id`：用户正在喜欢/不喜欢的图片的ID
- `action`：用户执行的动作，用字符串`like`表示喜欢，`unlike`表示不喜欢

这里还使用了多对多字段的管理器`users_like`查询图片与喜欢用户之间的关系，然后使用`add()`和`remove()`方法用于添加和去除多对多关系。`add()`方法即使传入已经存在的数据对象，也不会重复建立关系，`remove()`即使传入不存在的对象，也不会报错。还有一个`clear()`方法可以快速的从关联表中全部清除多对多关系。

最后，使用了`JsonResponse`类，这个类的作用是将一个HTTP请求附加上`application/json`请求头，并将其中的内容序列化为JSON格式的字符串

编辑 `images` 应用的 `urls.py`，为该视图配置URL：

```
path('like/', views.image_like, name='like'),
```

5.1 加载jQuery

我们将使用jQuery来发送AJAX请求，为此需要在页面内加载jQuery，为了可以让jQuery在所有的模板内都生效，将其加载代码放入base.html文件中，编辑account应用的base.html文件，在`</body>`之前增加下列代码：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script>
$(document).ready(function () {
    {% block domready %}
    {% endblock %}
});
</script>
```

我们从Google CDN中加载了jQuery，可以直接在<https://jquery.com/> 下载jQuery并将其放入本应用的 `static` 文件夹内。

在引入jQuery之后，增加了一个`<script>`标签，定义了一个`$(document).ready()`，这是一个jQuery方法，在DOM加载完毕后会执行该方法。DOM是Document Object Model的简称，由浏览器在加载页面时生成，以树形结构保存当前页面的所有节点数据。这样保证了JS代码执行时，其要操作的对象已经全部生成。

`domready` 块，用于存放在DOM加载完毕后执行的JS代码，我们将在需要执行JS代码的具体模板中编写该块内容。

注意不要混淆JavaScript代码和Django模板标签。Django的模板语言在服务端进行处理，转换最终的HTML字节流，浏览器取得HTML字节流创建页面和DOM对象，并执行JavaScript代码。有时候动态的生成JavaScript代码非常方便。

在这一章里，我们直接将JS代码通过模板内块的形式编写进来，这是为了教学方便。最好的方式是从静态文件中导入 .js 文件，以做到有效解耦HTML与JS。

5.2 AJAX中使用CSRF

在第二章中已经了解到 POST 请求中需要包含 `{% csrf_token %}` 生成的token数据，以防止跨站伪造请求攻击。不过在AJAX中发送CSRF token有点不方便，所以Django允许在AJAX请求中设置一个 `X-CSRFToken` 请求头，其中包含CSRF token的数据。jQuery在发送AJAX请求的时候设置上该请求头，就可以完成CSRF的发送了。

为了在AJAX请求中设置CSRF token，需要做如下事情：

1. 从 `csrftoken` cookie中取得CSRF token，如果开启了CSRF中间件，cookie中一直会有CSRF token数据
2. 将CSRF token数据设置在AJAX请求的 `X-CSRFToken` 请求头中

可以在 <https://docs.djangoproject.com/en/2.0/ref/csrf/#ajax> 阅读更多关于Django中CSRF与AJAX的信息。

修改刚刚在 `base.html` 中增加的JS代码部分，修改成下边这样：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/js-cookie@2/src/js.cookie.min.js"></script>
<script>
    let csrftoken = Cookies.get('csrftoken');

    function csrfSafeMethon(method) {
        // 如下的HTTP请求不需要设置CSRF信息
        return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
    }

    $.ajaxSetup({
        beforeSend: function (xhr, settings) {
            if (!csrfSafeMethon(settings.type) && !this.crossDomain) {
```

```
        xhr.setRequestHeader("X-CSRFToken", csrftoken);
    }
}
});
$(document).ready(function () {
{% block domready %}
{% endblock %}
});
</script>
```

以上代码解释如下：：

1. 通过外部CDN导入了一个JS库 `js-cookie`--一个轻量级的操作cookie的第三方库，可以在 <https://github.com/js-cookie/js-cookie> 找到该库的详细信息。
2. 通过 `Cookies.get()` 方法拿到 `csrftoken` 的值
3. 创建 `csrfSafeMethod()` 函数，使用正则验证HTTP请求种类，`GET`, `HEAD`, `OPTIONS`和`TRACE`类型的请求无需添加CSRF信息
4. 调用 `$.ajaxSetup()` 方法，在AJAX请求发送之前，为请求设置 `X-CSRFToken` 请求头信息，这个设置会影响到所有jQuery发送的AJAX请求。

这样所有的不安全的HTTP请求，比如 `GET` 或 `PUT`，都会被添加上CRSF信息。

5.3 jQuery发送AJAX请求

编辑 `images` 应用的 `images/image/detail.html` 文件，找到下边这行：

```
{% with total_likes=image.users_like.count %}
```

将其修改成：

```
{% with total_likes=image.users_like.count users_like=image.users_like.all %}
```

然后修改 `<div class="image-info">` 其中的内容，如下：

```
<div class="image-info">
  <div>
    <span class="count">
      <span class="total">{{ total_likes }}</span>
      like{{ total_likes|pluralize }}
    </span>
    <a href="#" data-id="{{ image.id }}" data-action="{% if request.user in users_like %}un{% endif %}like" class="button"
       {% if request.user not in users_like %}
         Like
       {% else %}
         Unlike
       {% endif %}
    </a>
  </div>
  {{ image.description|linebreaks }}
</div>
```

模板内首先通过 `{% with %}` 指定了新的变量 `users_like`，用于存放所有喜欢该图片的用户，可以避免反复查询。然后显示总的喜欢该图片的人数，还包含一个按钮样式的 `<a>` 标签。这个按钮根据当前用户是否在 `users_like` 中，显示 `like` 或 `unlike`，还为 `<a>` 标签设置了两个HTML5自定义属性：

1. `data-id`：当前页面显示图片的ID
2. `data-action`：用户的动作，喜欢或者不喜欢，值是 `like` 或 `unlike`

我们将把这两个HTML5自定义属性的值通过AJAX发送给 `image_like` 视图，当用户点击喜欢/不喜欢按钮的时候，我们需要在客户端做如下操作：

1. 调用AJAX视图，传入两个参数：`id` 和 `action`
2. 如果AJAX请求成功返回，更新按钮的 `data-action` 属性为相反的操作（原来是 `like` 则更新为 `unlike`，反之亦反）
3. 更新喜欢当前图片的用户总数

为此来编写页面所需的JS代码，在 `images/image/detail.html` 中添加 `domready` 块的内容：

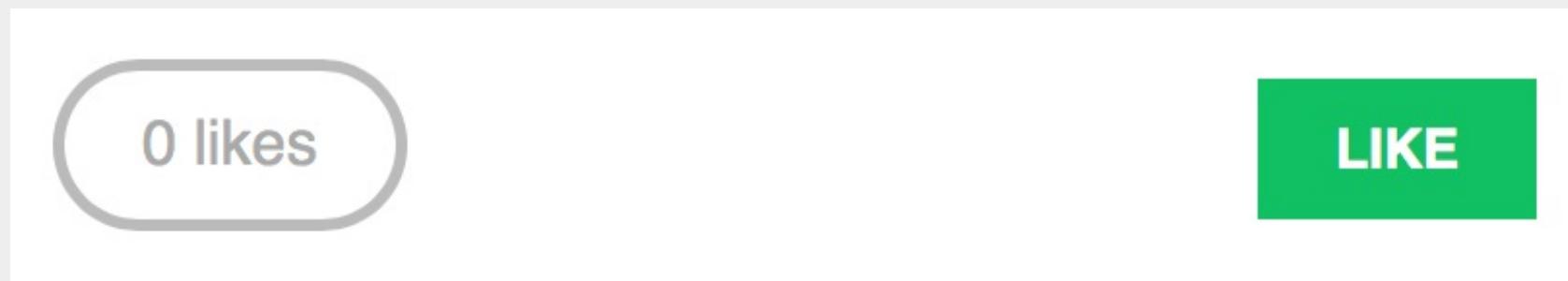
```
{% block domready %}  
$('a.like').click(function (e) {  
    e.preventDefault();  
    $.post('{% url 'images:like' %}',  
    {  
        id: $(this).data('id'),  
        action: $(this).data('action'),  
    },  
    function (data) {  
        if (data['status'] === 'ok') {  
            let previous_action = $('a.like').data('action');  
            //切换 data-action 属性  
            $('a.like').data('action', previous_action === 'like' ? 'unlike' : 'like');  
            //切换按钮文本  
            $('a.like').text(previous_action === 'like' ? 'Unlike' : 'Like');  
            //更新总的喜欢人数  
            let previous_likes = parseInt($('span.count.total').text());  
            $('span.count.total').text(previous_action === 'like' ? previous_likes + 1 : previous_likes - 1);  
        }  
    }  
});  
{% endblock %}
```

这段代码的逻辑解释如下：

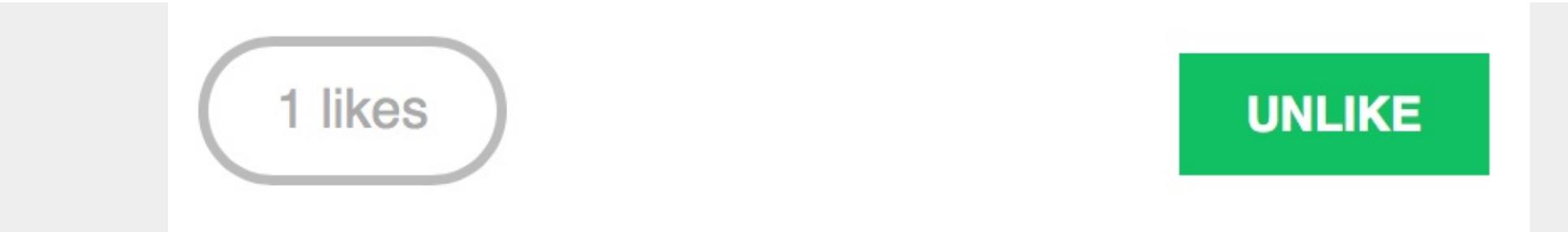
1. 使用 `$('.a.like')` 选择所有属于 `like` 类的 `<a>` 标签
2. 给 `<a>` 标签绑定 `click` 事件，每次点击就发送AJAX请求。
3. 在事件处理函数内，使用 `e.preventDefault()` 阻止 `<a>` 的默认功能，即阻止打开新的超链接
4. 使用 `$.post()` 发送异步的 `POST` 请求。jQuery还提供了 `$.get()` 用于发送异步的 `GET` 请求，和一个更底层的 `$.ajax()` 方法。
5. 使用 `{% url %}` 反向解析出AJAX的请求目标地址
6. 创建要发送的数据字典，通过 `<a>` 标签的 `data-id` 和 `data-action` 设置 `id` 和 `action` 键值对。
7. 设置回调函数，当成功收到AJAX响应时执行，响应数据被包含在对象 `data` 中。
8. 根据 `data` 中的 `status` 判断值是否为 `ok`，如果是则切换 `data-action` 和按钮文本。
9. 根据刚才执行的结果，对总喜欢人数增加1或者减少1

译者注：原书这里的逻辑是为了让读者可以迅速看出操作结果。在多用户的环境中，不能如此简单的增减1，因为每次执行动作后，该人数的变化未必是1。

打开任意图片详情页，可以看到新增的总人数和按钮，如下所示：



点击一下LIKE按钮，可以看到如下所示：



1 likes

UNLIKE

如果再点击UNLIKE按钮，可以看到按钮变回LIKE，人数也减少1

如果提示 `The 'photo' attribute has no file associated with it` 错误，原书作者在这里没有讲清楚，错误原因是 `detail.html` 页面用了 `user.profile.photo.url`，但没有上传用户头像。在管理后台给每个用户上传头像，再访问任意详情图片页，就不会报错了。直接修改多对多的关系再查看这张表，就能发现显示出同样喜欢了这张图的用户头像和名称。这里如果要完善的话，应该判断用户是否上传头像，如果没有就用默认头像代替。

当编写JavaScript代码发送AJAX请求时，为了方便调试，推荐使用开发工具而不是在Django中编写代码。现代浏览器都带有开发工具用于调试页面和JavaScript代码，通常可以按F12或者在页面上右击选“检查”来启动开发工具。

6 创建自定义装饰器

在AJAX视图中使用了 `@require_POST` 装饰器以限制视图仅接受 `POST` 请求，这显然还不够，需要让这个视图仅接受AJAX请求才行。Django对于HTTP请求对象提供了一个 `is_ajax()` 方法，通过HTTP请求头部字段 `HTTP_X_REQUESTED_WITH` 判断该请求是否是一个 `XMLHttpRequest` 对象，即一个AJAX请求。

我们准备自行编写一个装饰器，用于检查HTTP请求的 `HTTP_X_REQUESTED_WITH` 头部信息，从而限制我们的视图仅接受AJAX请求。Python中的装饰器是接受一个函数为参数的函数，为参数函数附加执行额外功能而不改变原函数的功能。如果对装饰器不太了解，可以参考Python官方文档：<https://www.python.org/dev/peps/pep-0318/>。

我们准备编写的装饰器是通用的，所以在 `bookmarks` 项目根目录下建立一个 `common` 包，其中的文件如下：

```
common/
__init__.py
decorators.py
```

编辑 `decorators.py` 文件，添加下列代码：

```
from django.http import HttpResponseBadRequest

def ajax_required(func):
    def wrap(request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseBadRequest()
        else:
            return func(request, *args, **kwargs)
    wrap.__doc__ = func.__doc__
    wrap.__name__ = func.__name__
    return wrap
```

这段代码就是自定义的 `ajax_required` 装饰器函数。其中定义了一个 `wrap` 函数，如果请求不是AJAX请求，就返回 `HttpResponseBadRequest` 即HTTP 400错误。如果是AJAX请求，则原来视图的功能正常执行。

然后编辑 `images` 应用的 `views.py` 文件，导入新的包然后为视图添加自定义装饰器：

```
from common.decorators import ajax_required

@ajax_required
@login_required
@require_POST
```

```
def image_like(request):
    # .....
```

如果用浏览器直接访问 <http://127.0.0.1:8000/images/like/>，会得到400错误。（未添加该装饰器之前，得到的是由 `@require_POST` 返回的405错误）。

如果你发现项目中的很多视图对同一个条件做判断，可以考虑将该判断逻辑编写为一个自定义装饰器。

7 AJAX分页

我们将制作一个图片列表页，用于列出我们网站所有的图片。这里将使用AJAX动态的发送图片数据，即当页面滚动到底部的时候，就会继续显示新的图片，直到全部图片都显示完毕。

为此我们将编写一个图片列表视图，同时处理普通的HTTP请求和AJAX请求。当用户一开始以 `GET` 请求方式访问图片列表页时，会显示第一页图片。当用户滚动到页面底部时，通过AJAX发送请求给该视图，返回下一页图片显示在页面底部；如此反复直到所有图片都显示完毕。

编辑 `images` 应用的 `views.py` 文件，创建一个新的视图：

```
from django.http import HttpResponse
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

@login_required
def image_list(request):
    images = Image.objects.all()
    paginator = Paginator(images, 8)
    page = request.GET.get('page')
    try:
        images = paginator.page(page)
    except PageNotAnInteger:
```

```
# 如果页数不是整数，就返回第一页
images = paginator.page(1)
except EmptyPage:
    # 如果是不存在的页数，而且请求是AJAX请求，返回空字符串
    if request.is_ajax():
        return HttpResponse('')
# 如果页数超范围，显示最后一页
images = paginator.page(paginator.num_pages)
if request.is_ajax():
    return render(request, 'images/image/list_ajax.html', {'section': 'images', 'images': images})
return render(request, 'images/image/list.html', {'section': 'images', 'images': images})
```

在这个视图中，先查询所有图片，然后使用内置的分页功能创建 `Paginator` 对象，按照8个图片一页进行分组。当HTTP请求的页面不存在的时候捕捉 `EmptyPage` 异常，判断此时请求的种类，如果是AJAX请求，说明页面到了底部，返回空字符串即可。我们将结果渲染到两个不同的模板中：

1. 对于AJAX请求，渲染 `list_ajax.html` 模板，这个模板仅包含图片内容。
2. 对于普通请求，渲染 `list.html`，这个模板会继承 `base.html`，并且 `include list_ajax.html` 模板

编辑 `images` 应用的 `urls.py` 文件，为新视图添加一行URL：

```
path('', views.image_list, name='list'),
```

最后来创建前述的两个模板，在 `images/image/` 模板目录下创建 `list_ajax.html`，添加如下代码：

```
{% load thumbnail %}

{% for image in images %}
<div class="image">
    <a href="{{ image.get_absolute_url }}">
        {% thumbnail image.image "300x300" crop="100%" as im %}
```

```
<a href="{{ image.get_absolute_url }}>
    
    <a href="{{ image.get_absolute_url }}" class="title">
        {{ image.title }}
    </a>
</div>
</div>
{%- endfor %}
```

上述模板显示图片列表，将使用这个模板渲染AJAX请求返回的结果。在相同目录下创建 `list.html` 文件并添加如下代码：

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

{% block title %}
Images bookmarked
{% endblock %}

{% block content %}
<h1>Images bookmarked</h1>
<div id="image-list">
    {% include 'images/image/list_ajax.html' %}
</div>
{% endblock %}
```

这个页面继承 `base.html`，同时包含了 `list_ajax.html`，这个模板中还必须包含发送AJAX的JS代码，所以继续在其中编写 `domready` 块的内容：

```
{% block domready %}
let page = 1;
let empty_page = false;
let block_request = false;
$(window).scroll(
    function () {
        let margin = $(document).height() - $(window).height() - 200;
        if ($(window).scrollTop() > margin && empty_page === false && block_request === false) {
            block_request = true;
            page += 1;
            $.get("?page=" + page, function (data) {
                if (data === "") {
                    empty_page = true;
                }
                else {
                    block_request = false;
                    $('#image-list').append(data)
                }
            });
        }
    }
);
{% endblock %}
```

这段代码实现了滚动加载功能，其中的逻辑解释如下：

1. 首先创建如下变量：

1. `page`：存储当前页数
2. `empty_page`：判断是否已经到达页面底部。如果已经到达底部，阻止发送AJAX请求
3. `block_request`：在已经发送AJAX请求但还未收到响应时阻止再发送AJAX请求

2. 使用 `$(window).scroll()` 方法监听滚动事件

3. 计算页面高度和窗口高度的差，记录在 `margin` 变量中，表示未显示的页面的高度。再减去200表示当滚动到离窗口底部还有200像素的时候发送AJAX请求。
4. 判断 `block_request` 和 `empty_page` 同时为 `False` 的情况下发送AJAX请求。
5. 发送AJAX请求之后将 `block_request` 设置为 `True`，避免再次发送，同时将 `page` 增加1，下一次发送的时候就获取下一个分页结果。
6. 使用 `$.get()` 方法发送类型为 `GET` 的AJAX请求，将响应数据保存到 `data` 中，然后处理以下两种情况：
 1. 响应数据中无内容：说明视图返回了空字符串，已经没有更多的分页结果可以加载，此时将 `empty_page` 设置为 `True`，阻止后续所有AJAX请求发送
 2. 响应数据中有数据：说明得到了新的分页结果，将其中的内容追加到id属性为 `image-list` 的元素内部，页面下方增加出新的图片。

在浏览器中打开 <http://127.0.0.1:8000/images/>，可以看到如下页面（需要自行添加一些图片）：

Images bookmarked



Louis Armstrong



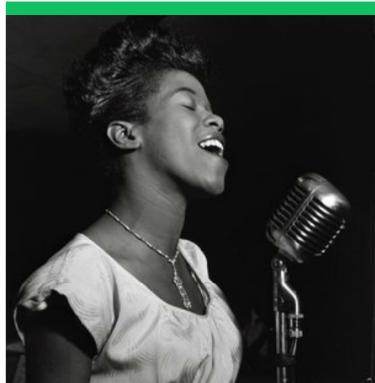
Chick Corea



Al Jarreau



Al Jarreau



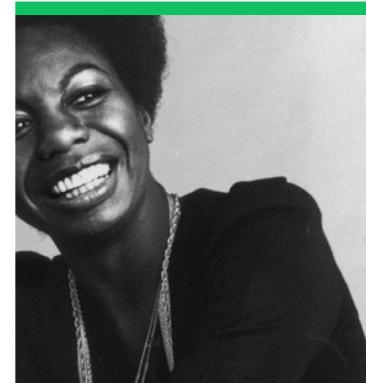
Ella Fitzgerald



Glenn Miller



Charlie Parker



Nina Simone

滚动该页面到底部，确保在数据库中添加了超过8张图片，会看到额外的图片被加载并显示出来

最后修改 `base.html` 文件中顶部导航栏的连接，添加下列代码：

```
<li {% if section == "images" %}class="selected"{% endif %}>
    <a href="{% url "images:list" %}">Images</a>
</li>
```

现在就可以通过用户首页访问图片清单页面了。

总结

这一章建立了一个小书签程序，用于分享图片到本站，还实现了jQuery发送AJAX请求和使用AJAX动态加载页面。

下一章将学习建立关注系统，涉及到模型的通用关系，信号功能和数据库的非规范化等知识，还将学习到在Django中使用Redis数据库。

第六章 追踪用户行为

在之前的章节里完成了小书签将外站图片保存至本站的功能，并且实现了通过jQuery发送AJAX请求，让用户可以对图片进行喜欢/不喜欢操作。

这一章将学习如何创建一个用户关注系统和创建用户行为流数据，还将学习Django的信号框架使用和集成Redis数据库到Django中。主要的内容有：

- 通过中间模型建立多对多关系
- 创建关注系统
- 创建行为流应用（显示用户最近的行为列表）
- 为模型添加通用关系
- 优化QuerySet查找外键关联模型
- 使用signal模块对数据库进行非规范化改造
- 在Redis中存取内容

1 创建关注系统

所谓关注系统，就是指用户可以关注其他用户，并且可以看到所关注用户的行为。关注关系在用户之间是多对多的关系，一个用户可以关注很多用户，也可以被很多用户关注。

1.1 通过中间模型创建多对多关系

在之前的章节中，通过 `ManyToManyField` 创建了多对多关系，然后让Django创建了数据表。对于大多数情况，直接使用多对多字段已经足够。在需要为多对多关系存储额外的信息时（比如创建多对多关系的时间字段，描述多对多关系性质

的字段），可能需要自定义一个模型作为多对多关系的中间模型。

我们将创建一个中间模型用来建立用户之间的多对多关系，原因是：

- 我们将使用内置的 `User` 模型，但不想修改它
- 想存储一个用户关注另外一个用户的时间

在 `account` 应用的 `models.py` 中建立新 `Contact` 类：

```
class Contact(models.Model):  
    user_from = models.ForeignKey(settings.AUTH_USER_MODEL, related_name='rel_from_set', on_delete=models.CASCADE)  
    user_to = models.ForeignKey(settings.AUTH_USER_MODEL, related_name='rel_to_set', on_delete=models.CASCADE)  
    created = models.DateTimeField(auto_now_add=True, db_index=True)  
  
    class Meta:  
        ordering = ('-created',)  
  
    def __str__(self):  
        return '{} follows {}'.format(self.user_from, self.user_to)
```

这个 `Contact` 类将用来记录用户关注关系，包含如下字段：

- `user_from`：发起关注的用户外键
- `user_to`：被关注的用户外键
- `created`：该关注关系创建的时间，使用 `auto_now_add=True` 自动记录时间

数据库对于外键会自动创建索引，这里还使用了 `db_index=True` 为 `created` 字段创建了索引。

使用ORM的时候，如果 `user1` 关注了 `user2`，实际操作的语句可以写成这样：

```
user1 = User.objects.get(id=n)
user2 = User.objects.get(id=m)
Contact.objects.create(user_from=user1, user_to=user2)
```

基于 `Contact` 模型，可以通过为两个外键字段设置的名称 `rel_from_set` 和 `rel_to_set` 作为管理器名称进行查询。为了从 `User` 模型中也可以进行查询，`User` 模型应该有一个多对多关系关联到其自己，类似这样：

```
following = models.ManyToManyField('self',
    through=Contact,
    related_name='followers',
    symmetrical=False)
```

在上边这行代码里，我们 `through=Contact` 告诉 Django 以 `Contact` 类作为中间表格建立多对多关系，这是一个 `User` 模型与自己的多对多关系，其中的 '`self`' 参数表示模型自己。

当需要在多对多关系中记录额外数据时，创建一个关联到两个模型的中间表格，然后手动指定 `ManyToManyField` 的 `through` 参数，将中间表格作为多对多关系的中间表。

如果 `User` 模型是我们自定义的模型，可以很方便的为其添加 `following` 字段，但我们不想修改 `User` 类，这里可以采用一个动态的方法为其添加字段。在 `account` 应用里的 `models.py` 里增加如下内容：

```
from django.contrib.auth.models import User
User.add_to_class('following',
    models.ManyToManyField('self', through=Contact, related_name='followers', symmetrical=False))
```

这里用了一个 `add_to_class()` 方法给 `User` 打了一个 猴子补丁，不推荐使用该方法。但是在这里使用主要考虑如下原因：

- 通过这个方法简化了查询，通过使用 `user.followers.all()` 和 `user.following.all()` 可以迅速查询。如果通过一对关系定义在 `Profile` 模型上，查询就要复杂很多。
- 通过这种方法添加的多对多字段实际是通过 `Contact` 模型生效，不会实际修改数据库中的 `User` 数据表
- 也无需建立自定义的 `User` 模型替换原 `User` 模型

这里需要在此强调的是，在大部分情况下需要为内置数据模型增加额外数据时，优先通过一对一的方式如 `Profile` 模型进行扩展，将额外信息和关系字段都添加在扩展的数据上；其次是自定义新的数据模型取代原数据模型，而不是直接通过猴子补丁。否则给后续开发和测试带来很大困难。关于自定义用户模型可以参考

<https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#specifying-a-custom-user-model>。

这里还有一个参数是 `symmetrical=False` 对称参数，当创建一个 **关联到自身的多对多字段** 的时候，Django默认关系是对称的，即A关注了B，会自动添加B也关注A的记录，这与实际情况不符，所以必须设置为 `False`。

使用中间表格作为多对多关系的中间表时，一些管理器的内置方法如 `add()`，`create()`，`remove()` 等无法使用，必须编写直接操作中间表的代码。

定义好中间表后，执行数据迁移过程。现在模型已经建好，我们需要建立展示用户关注关系的列表和详情视图。

1.2 创建用户关注关系的列表和详情视图

在account应用的views.py里添加如下内容：

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request, 'account/user/list.html', {'section': 'people', 'users': users})
```

```
@login_required
def user_detail(request, username):
    user = get_object_or_404(User, username=username, is_active=True)
    return render(request, 'account/user/detail.html', {'section': 'people', 'user': user})
```

这是两个简单的展示所有用列表户和某个具体用户信息的视图，如果用户较多，还可以为 `user_list` 添加分页功能。

`user_detail` 使用了 `get_object_or_404` 方法，如果找不到用户就会返回一个404错误。

编辑 `account` 应用的 `urls.py` 文件，为这两个视图配置URL：

```
path('users/', views.user_list, name='user_list'),
path('users/<username>/', views.user_detail, name='user_detail'),
```

这里我们看到，需要通过URL传参数给视图，需要建立规范化URL，为模型添加 `get_absolute_url()`，除了通过自定义的方法之外，对于User这种内置的模型，还有一种方法是设置 `ABSOLUTE_URL_OVERRIDES`。

修改项目的 `settings.py` 文件：

```
from django.urls import reverse_lazy

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])}
```

Django动态的为所有 `ABSOLUTE_URL_OVERRIDES` 中列出的模型添加 `get_absolute_url()` 方法，这个方法按照设置中的结果返回规范化URL。这里通过一个匿名函数返回规范化URL，这个匿名函数被绑定在对象上，作为调用 `get_absolute_url()` 时候实际调用的函数。

配置好了以后我们先来实验一下，打开命令行模式：

```
>>> from django.contrib.auth.models import User  
>>> user = User.objects.latest('id')  
>>> str(user.get_absolute_url())  
>>> '/account/users/caidaye/'
```

可以看到解析出了地址，之后需要建立模板，在 `account` 应用的 `templates/account/` 目录下建立如下目录和文件结构：

```
/user/  
    detail.html  
    list.html
```

之后编写其中的 `list.html`：

```
{#list.html#}  
{% extends "base.html" %}  
{% load thumbnail %}  
{% block title %}People{% endblock %}  
{% block content %}  
    <h1>People</h1>  
    <div id="people-list">  
        {% for user in users %}  
            <div class="user">  
                <a href="{{ user.get_absolute_url }}>  
                    {% thumbnail user.profile.photo "180x180" crop="100%" as im %}  
                    
```

```
        <a href="{{ user.get_absolute_url }}" class="title">
            {{ user.get_full_name }}
        </a>
    </div>
</div>
{% endfor %}
</div>
{% endblock %}
```

这个模板中用一个循环列出了视图返回的所有活跃用户，分别显示每个用户的名称和头像，使用`{% thumbnail %}`显示缩略图。

在`base.html`中添加这个模板的路径，作为用户关注系统的链接首页：

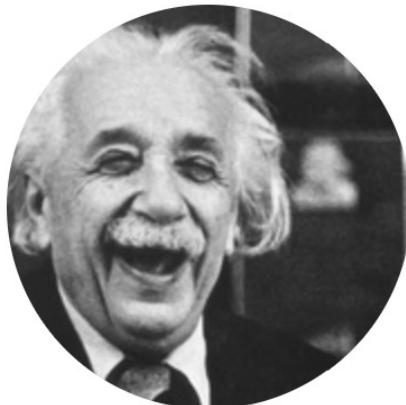
```
<li {% if section == 'people' %}class="selected"{% endif %}><a href="{% url 'user_list' %}">People</a></li>
```

之后启动网站，到`http://127.0.0.1:8000/account/users/`可以看到显示出了用户列表页面，示例如下：

People



Tesla



Einstein



Turing

如果无法显示缩略图，记得在 `settings.py` 中设置 `THUMBNAIL_DEBUG = True`，在命令行窗口中查看错误信息。

编写 `account/user/detail.html` 来展示具体用户：

```
{% extends "base.html" %}  
{% load thumbnail %}  
{% block title %}{{ user.get_full_name }}{% endblock %}  
{% block content %}  
    <h1>{{ user.get_full_name }}</h1>
```

```
<div class="profile-info">
    {% thumbnail user.profile.photo "180x180" crop="100%" as im %}
        
    {% endthumbnail %}
</div>
{% with total_followers=user.followers.count %}
    <span class="count">
<span class="total">{{ total_followers }}</span>
follower{{ total_followers|pluralize }}
    </span>
    <a href="#" data-id="{{ user.id }}" data-action="{% if request.user in user.followers.all %}unfollow{% endif %}{% if request.user not in user.followers.all %}follow{% else %}unfollow{% endif %}>
        {% if request.user not in user.followers.all %}
            Follow
        {% else %}
            Unfollow
        {% endif %}
    </a>
    <div id="image-list" class="image-container">
        {% include "images/image/list_ajax.html" with images=user.images_created.all %}
    </div>
    {% endwith %}
{% endblock %}
```

在这个详情页面，同样展示用户名和使用 `{% thumbnail %}` 展示用户头像缩略图。此外还展示了关注该用户的人数，以及提供了一个按钮供当前用户关注/取消关注该用户。和上一章类似，我们将使用AJAX技术来完成关注/取消关注行为，为此在 `<a>` 标签中增加了 `data-id` 和 `data-action` 属性用于保存用户ID和初始动作。还通过引入 `images/image/list_ajax.html` 展示了该用户上传的所有图片。

启动站点，点击某个具体的用户，可以看到用户详情页面的示例如下：

Tesla



0 followers

FOLLOW



Django and Duke



Louis Armstrong



Chick Corea

1.3 创建用户关注行为的AJAX视图

编辑 account 应用的 `views.py` 文件：

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from common.decorators import ajax_required
from .models import Contact
```

```
@ajax_required
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == "follow":
                Contact.objects.get_or_create(user_from=request.user, user_to=user)
            else:
                Contact.objects.filter(user_from=request.user, user_to=user).delete()
            return JsonResponse({'status': 'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status': 'ko'})

    return JsonResponse({'status': 'ko'})
```

这个视图与之前喜欢/不喜欢图片的功能如出一辙。由于我们使用了自定义的中间表作为多对多字段中间表，无法通过 `User` 模型直接使用管理器的 `add()` 和 `remove()` 方法，因此这里直接操作 `Contact` 模型。

编辑 `account` 应用的 `urls.py` 文件，添加一行

```
path('users/follow/', views.user_follow, name="user_follow"),
```

注意这一行一定要在 `user_detail` 的URL配置之前，否则所有访问 `/users/follow/` 路径的请求都会被路由至 `user_detail` 视图。记住Django匹配URL的顺序是从上到下停在第一个匹配成功的地方。

修改 `account` 应用的 `user/detail.html`，添加发送AJAX请求的JavaScript代码：

```
{% block domready %}
$('a.follow').click(function (e) {
    e.preventDefault();
    $.post('{% url 'user_follow' %}', {
        id: $(this).data('id'),
        action: $(this).data('action')
    },
    function (data) {
        if (data['status'] === 'ok') {
            let previous_action = $('a.follow').data('action');
            // 切换 data-action 属性
            $('a.follow').data('action', previous_action === 'follow' ? 'unfollow' : 'follow');
            // 切换按钮文字
            $('a.follow').text(previous_action === 'follow' ? 'unfollow' : 'follow');
            // 更新关注人数
            let previous_followers = parseInt($('span.count .total').text());
            $('span.count .total').text(previous_action === 'follow' ? previous_followers + 1 : previous_followers - 1);
        }
    });
});
{% endblock %}
```

这个函数的逻辑也和上一章的喜欢/不喜欢功能很相似。用户点击按钮时，首先将用户ID和行为发送至视图，根据返回的结果，相应切换行为属性和显示的文字，同时更新关注人数。尝试打开一个用户详情页面并且点击喜欢，之后可以看到显示如下：

1 followers

UNFOLLOW

译者注：这个函数和之前的AJAX函数一样，更新关注人数的逻辑比较简单粗暴，关注人数最好从数据库中取followers的总数。原书明显是为了让读者看到立竿见影的效果。

2 创建通用行为流应用

许多社交网站向其用户展示其他用户的行为流，供用户追踪其他用户最近在网站中做了什么。一个行为流是一个用户或者一组用户最近进行的所有活动的列表。例如Facebook界面的News Feed就是一个行为流。对于我们的网站来说，X用户上传了Y图片或者X用户关注了Y用户，都是行为流中的一个数据。我们也准备创建一个行为流应用，让用户可以看到他们所关注的用户最近的所有活动。为了实现这个功能，我们需要建立一个模型，用于保存一个用户最近在网站上做过的所有事情，及向模型中添加行为记录的方法。

新建一个叫 `actions` 应用然后添加到 `settings.py` 里，如下所示：

```
INSTALLED_APPS = [
    # ...
    'actions.apps.ActionsConfig',
]
```

在 `action` 应用中编辑 `models.py`：

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.user', related_name='actions', db_index=True, on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        ordering = ('-created',)
```

上边的代码建立了一个 `Action` 模型，用于存放用户的所有行为记录，模型的字段有这些：

- `user`：进行行为的主体，即用户，采用了 `ForeignKey` 关联至内置的 `User` 模型
- `verb`：行为的动词，描述用户进行了什么行为
- `created`：记录用户执行行为的时间，采用 `auto_now_add=True` 自动记录创建该条数据的时间

使用这个模型，我们目前只能记录行为的主体和行为动词，即 `用户X关注了...` 或者 `用户X上传了...`，还缺少行为的目标对象。显然我们还需要一个外键关联到用户操作的具体对象上，这样才能够展示出类似 `用户X关注了用户Y` 这样的行为流。在之前我们已经知道，一个 `ForeignKey` 字段只能关联到一个模型，很显然无法满足我们的需求。目标对象必须可以是任意一个已经存在的模型的对象，这个时候Django的content types框架就该登场了。

2.1 使用contenttypes框架

`django.contrib.contenttypes` 模块中提供了一个contenttypes框架，这个框架可以追踪当前项目内所有已激活的应用中的所有模型，并且提供一个通用的接口可以操作模型。

`django.contrib.contenttypes` 同时也是一个应用，在默认设置中已经包含在 `INSTALLED_APPS` 中，其他 `contrib` 包中的程序也使用这个框架，比如内置认证模块和管理后台。

`contenttypes` 应用中包含一个 `ContentType` 模型。这个模型的实例代表项目中一个实际的数据模型。当项目中每新建一个模型时，`ContentType` 的新实例会自动增加一个，对应该新增模型。`ContentType` 模型包含如下字段：

- `app_label`：数据模型所属的应用名称，这个来自模型内的 `Meta` 类里的 `app_label` 属性。我们的 `Image` 模型就属于 `images` 应用
- `model`：模型的名称
- `name`：给人类阅读的名称，这个来自模型内的 `Meta` 类的 `verbose_name` 属性。

来看一下如何使用 `ContentType` 对象，打开系统命令行窗口，可以通过指定 `app_label` 和 `model` 属性，在 `ContentType` 模型中查询得到一个具体对象：

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(app_label='images', model='image')
>>> image_type
<ContentType: image>
```

还可以对刚获得的 `ContentType` 对象调用 `model_class()` 方法查看类型：

```
>>> image_type.model_class()
<class 'images.models.Image'>
```

还可以直接通过具体的类名获取对应的 `ContentType` 对象：

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: image>
```

这是几个简单的例子，还有更多的方法可以操作，详情可以阅读官方文档：

<https://docs.djangoproject.com/en/2.0/ref/contrib/contenttypes/>。

2.2 为模型添加通用关系

通常来说，通过获取ContentType模型的实例，就可以与整个项目中任何一个模型建立关系。为了建立通用关系，需要如下三个字段：

- 一个关联到 ContentType 模型的 ForeignKey，这会用来反映与外键所在模型关联的具体模型。
- 一个存储具体的模型的主键的字段，通常采用 PositiveIntegerField 字段，以匹配主键自增字段，这个字段用于从相关的具体模型中确定一个对象。
- 一个使用前两个字段，用于管理通用关系的字段，content types框架提供了一个 GenericForeignKey 专门用于管理通用关系。

编辑 actions 应用的 models.py 文件：

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.user', related_name='actions', db_index=True, on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    target_ct = models.ForeignKey(ContentType, blank=True, null=True, related_name='target_obj',
                                  on_delete=models.CASCADE)
    target_id = models.PositiveIntegerField(null=True, blank=True, db_index=True)
    target = GenericForeignKey('target_ct', 'target_id')
    created = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        ordering = ('-created',)
```

我们将下列字段增加到了 Action 模型中：

- `target_ct`：一个外键字段，关联到 `ContentType` 模型
- `target_id`：一个 `PositiveIntegerField` 字段，用于存储相关模型的主键
- `target`：一个 `GenericForeignKey` 字段，通过组合前两个字段得到

Django并不会为 `GenericForeignKey` 创建数据表中的字段，只有 `target_ct` 和 `target_id` 会被写入数据表。这两个字段都设置了 `blank=True` 和 `null=True`，这样新增 `Action` 对象的时候不会强制要有关联的目标对象。

如果确实需要的话，建立通用关系比使用外键可以创建更灵活的关系。

创建完模型之后，执行数据迁移程序，然后将Action模型添加到管理后台中，编辑 `actions` 应用的 `admin.py` 文件：

```
from django.contrib import admin
from .models import Action

@admin.register(Action)
class ActionAdmin(admin.ModelAdmin):
    list_display = ('user', 'verb', 'target', 'created')
    list_filter = ('created',)
    search_fields = ('verb',)
```

加入管理后台之后，打开 <http://127.0.0.1:8000/admin/actions/action/add/>，可以看到如下界面：

Add action

User:  

Target ct:

Target id:

Verb:

[Save and add another](#) [Save and continue editing](#) **SAVE**

这里可以看到，只有 `target_id` 和 `target_ct` 出现，`GenericForeignKey` 并没有出现在表单中。`target_ct` 字段允许选择项目中的所有模型，可以使用 `limit_choices_to` 属性来限制可以选择的模型。

在 `actions` 应用中新建 `utils.py` 文件，在其中将编写一个函数用来快捷的建立新 `Action` 对象：

```
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

这个 `create_action()` 函数的参数有一个 `target`，就是行为所关联的目标对象，可以在任意地方导入该文件然后使用这个函数来快速为行为流添加新行为对象。

2.3 避免添加重复的行为

有些时候，用户可能在短期内连续点击同一类型的事件，比如取消又关注，关注再取消，如果即使保存所有行为，会造成大量重复的数据。为了避免这种情况，需要修改一下刚刚建立的 `utils.py` 文件中的 `create_action()` 函数：

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action


def create_action(user, verb, target=None):
    # 检查最后一分钟内的相同动作
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id, verb=verb, created__gte=last_minute)

    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(target_ct=target_ct, target_id=target.id)

    if not similar_actions:
        # 最后一分钟内找不到相似的记录
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```

我们修改了 `create_action()` 函数避免在一分钟内重复保存相同动作，并且返回一个布尔值以表示是否成功保存。这个函数的逻辑解释如下：

- 首先，通过 `timezone.now()` 获取当前时间，这个方法与 `datetime.datetime.now()` 相同，但是返回一个 `timezone-aware` 对象。Django 使用 `USE_TZ` 设置控制是否支持时区，使用 `startapp` 命令建立的项目默认 `USE_TZ=True`
- 使用 `last_minute` 变量保存之前一分钟，然后获取之前一分钟到现在，当前用户进行的所有动词相同的行为。
- 如果没有找到任何相同的行为，就直接创建 `Action` 对象，并返回 `True`，否则返回 `False`。

2.4 向行为流中添加行为

现在需要编辑视图，添加一些功能来创建行为流。我们将对下边的行为创建行为流：

- 任意用户上传了图片
- 任意用户喜欢了一张图片
- 任意用户创建账户
- 任意用户关注其他用户

编辑 `images` 应用的 `views.py` 文件：

```
from actions.utils import create_action
```

在 `image_create` 视图中，在保存图片之后添加 `create_action()` 语句：

```
new_item.save()
create_action(request.user, 'bookmarked image', new_item)
```

在 `image_like` 视图中，在将用户添加到 `users_like` 关系之后添加 `create_action()` 语句：

```
image.users_like.add(request.user)
create_action(request.user, 'likes', image)
```

编辑 `account` 应用的 `views.py` 文件，添加如下导入语句：

```
from actions.utils import create_action
```

在 `register` 视图里，在创建 `Profile` 对象之后添加 `create_action()` 语句：

```
Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
```

在 `user_follow` 视图里也添加 `create_action()`：

```
Contact.objects.get_or_create(user_from=request.user, user_to=user)
create_action(request.user, 'is following', user)
```

从上边的代码中可以看到，由于建立好了 `Action` 模型，可以方便的添加各种行为。

2.5 展示用户行为流

最后，需要展示每个用户的行为流，我们将在用户的登录后页面中展示行为流。编辑 `account` 应用的 `views.py` 文件，修改 `dashboard` 视图，如下：

```
from actions.models import Action

@login_required
def dashboard(request):
```

```
# 默认展示所有行为，不包含当前用户
actions = Action.objects.exclude(user=request.user)
following_ids = request.user.following.value_list('id', flat=True)

if following_ids:
    # 如果当前用户有关注的用户，仅展示被关注用户的行为
    actions = actions.filter(user_id__in=following_ids)
actions = actions[:10]
return render(request, 'account/dashboard.html', {'section': 'dashboard', 'actions': actions})
```

在上边代码中，首先从数据库中获取除了当前用户之外的全部行为流数据。如果当前用户有关注其他用户，则在所有的行为流中筛选出属于关注用户的行为流。最后限制展示的数量为10条。在QuerySet中我们并没有使用 `order_by()` 方法，因为默认已经按照 `ordering='{-created}'` 进行了排序。

2.6 优化QuerySet查询关联对象

现在我们每次获取一个 `Action` 对象时，都会去查询关联的 `User` 对象，然后还会去查询 `User` 对象关联的 `Profile` 对象，要查询两次。Django ORM提供了一种简便的方法获取相关联的对象，而无需反复查询数据库。

2.6.1 使用 `select_related()`

Django提供了 `select_related()` 方法用于一对多字段查询关联对象。这个方法实际上会得到一个更加复杂的QuerySet，然而却避免了反复查询关联对象。`select_related()` 方法仅能用于 `ForeignKey` 和 `OneToOneField`，其实际生成的SQL语句是 `JOIN` 连表查询，方法的参数则是 `SELECT` 语句之后的字段名。

为了使用 `select_related()`，修改下边这行代码：

```
actions = actions[:10]
```

将其修改成：

```
actions = actions.select_related('user', 'user__profile')[:10]
```

我们使用 `user__profile` 在查询中将Profile数据表进行了连表查询。如果不给 `select_related()` 传任何参数，会将所有该表外键关联的表格都进行连表操作。最好每次都指定具体要关联的表。

进行连表操作的时候注意避免不需要的额外连表，以减少查询时间。

2.6.2 使用 `prefetch_related()`

`select_related()` 仅能用于一对一和一对多关系，不能用于多对多（`ManyToMany`）和多对一关系（反向的 `ForeignKey` 关系）。Django提供了QuerySet的 `prefetch_related()` 方法用于多对多和多对一关系查询，这个方法会对每个对象的关系进行一次单独查询，然后再把结果连接起来。这个方法还支持查询 `GenericRelation` 和 `GenericForeignKey` 字段。

编辑 `account` 应用的 `views.py` 文件，为 `GenericForeignKey` 增加 `prefetch_related()` 方法：

```
actions = actions.select_related('user', 'user__profile').prefetch_related('target')[:10]
```

现在我们就完成了优化查询的工作。

2.7 创建行为流模板

现在来创建展示用户行为的页面，在 `actions` 应用下创建 `templates` 目录，添加如下文件结构：

```
actions/
  action/
    detail.html
```

编辑 `actions/action/detail.html` 模板，添加如下内容：

```
{% load thumbnail %}
{% with user=action.user profile=action.user.profile %}
    <div class="action">
        <div class="images">
            {% if profile.photo %}
                {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
                    <a href="{{ user.get_absolute_url }}>
                        
                    </a>
                {% endthumbnail %}
            {% endif %}
            {% if action.target %}
                {% with target=action.target %}
                    {% if target.image %}
                        {% thumbnail target.image "80x80" crop="100%" as im %}
                            <a href="{{ target.get_absolute_url }}>
                                
                            </a>
                        {% endthumbnail %}
                    {% endif %}
                {% endwith %}
            {% endif %}
        </div>
        <div class="info">
            <p>
                <span class="date">{{ action.created|timesince }} ago</span>
                <br/>
                <a href="{{ user.get_absolute_url }}>
                    {{ user.first_name }}
                </a>
            </p>
        </div>
    </div>

```

```
    {{ action.verb }}
    {% if action.target %}
        {% with target=action.target %}
            <a href="{{ target.get_absolute_url }}">{{ target }}</a>
        {% endwith %}
    {% endif %}
    </p>
</div>
</div>
{% endwith %}
```

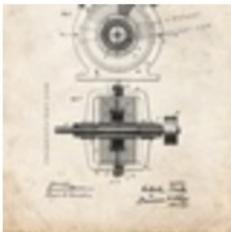
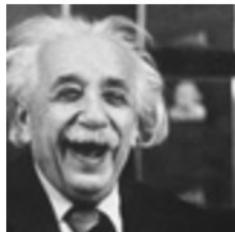
这是展示 `Action` 对象的模板。首先我们使用 `{% with %}` 标签存储当前用户和当前用户的 `Profile` 对象；然后如果 `Action` 对象存在关联的目标对象而且有图片，就展示这个目标对象的图片；最后，展示执行这个行为的用户的链接，动词，和目标对象。

然后编辑 `account` 应用里的 `dashboard.html`，把这个页面包含到 `content` 块的底部：

```
<h2>What's happening</h2>
<div id="action-list">
    {% for action in actions %}
        {% include 'actions/action/detail.html' %}
    {% endfor %}
</div>
```

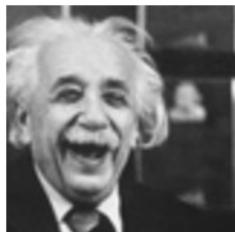
启动站点，打开 <http://127.0.0.1:8000/account/>，使用已经存在的用户登录，然后进行一些行为。再更换另外一个用户登录，关注之前的用户，然后到登录后页面看一下行为流，如下图所示：

What's happening



3 minutes ago

Einstein likes Alternating electric current generator



5 minutes ago

Einstein bookmarked image Turing Machine



2 days, 2 hours ago

Tesla likes Chick Corea

我们就建立了一个完整的行为流应用，可以方便的添加用户行为。还可以为这个页面添加之前的AJAX动态加载页面的效果。

3 使用signals非规范化数据

有些时候你可能需要非规范化数据库。 [非规范化 \(Denormalization\)](#) 是一种数据库方面的名词，指通过向数据库中添加冗余数据以提高效率。非规范化只有在确实必要的情况下再考虑使用。使用非规范化数据的最大问题是如何保持非规范化数据始终更新。

我们将通过一个例子展示如何通过非规范化数据提高查询效率，缺点就是必须额外编写代码以保持数据更新。我们将非规范化 `Image` 模型并通过Django的信号功能保持数据更新。

译者注：规范化简单理解就是不会存储对象非必要的额外信息，就像我们现在为止的所有设计，来自于对象基础信息以外的额外信息（如求和，分组）都通过设计良好的表间关系和查询手段获得，而且这些基础信息都在对应的视图内得到操作和更新。非规范化是与规范化相反的手段，添加冗余数据用于提高数据库的效率。这是结构化程序设计思想中的运行时间与占用空间关系在数据库结构方面的反映。

3.1 使用signal功能

Django提供一个信号模块，可以让receiver函数在某种动作发生的时候得到通知。信号功能在实现每当发生什么动作就执行一些代码的时候很有用，也可以创建自定义的信号用于通知其他程序

Django在 `django.db.models.signals` 中提供了一些信号功能，其中有如下的信号：

- `pre_save` 和 `post_save`，在调用 `save()` 方法之前和之后发送信号
- `pre_delete` 和 `post_delete`，在调用 `delete()` 方法之前和之后发送信号
- `m2m_changed` 在多对多字段发生变动的时候发送信号

这只是部分信号功能，完整的内置信号功能见官方文档 <https://docs.djangoproject.com/en/2.0/ref/signals/>。

举个例子来看如何使用信号功能。如果在图片列表页，想给图片按照受欢迎的程度排序，可以使用聚合函数，对喜欢该图片的用户合计总数，代码是这样：

```
from django.db.models import Count
from images.models import Image
images_by_popularity = Image.objects.annotate(total_likes=Count('users_like')).order_by('-total_likes')
```

在性能上来说，通过合计 `users_like` 字段，生成临时表再进行排序的操作，远没有直接通过一个字段排序的效率高。我们可以直接在 `Image` 模型上增加一个字段，用于保存图片的被喜欢数合计，这样虽然使数据库非规范化，但显著的提高了查询效率。现在的问题是，如何保持这个字段始终为最新值？

先到 `images` 应用的 `models.py` 中，为 `Image` 模型增加一个字段 `total_likes`：

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(db_index=True, default=0)
```

`total_likes` 用来存储喜欢该图片的用户总数，这个非规范化的字段在查询和排序的时候非常简便。

在使用非规范化手段之前，还有几种方法可以提高效率，比如使用索引，优化查询和使用缓存。

添加完字段之后执行数据迁移程序。

之后需要给 `m2m_changed` 信号设置一个 `receiver` 函数，在 `images` 应用目录内新建一个 `signals.py` 文件，添加如下代码：

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image

@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
```

```
instance.total_likes = instance.users_like.count()  
instance.save()
```

首先，使用 `@receiver` 装饰器，将 `users_like_changed` 函数注册为一个事件的接收 `receiver` 函数，然后将其设置为监听 `m2m_changed` 类型的信号，并且设置信号来源为 `Image.users_like.through`，这表示来自于 `Image.users_like` 字段的变动会触发该接收函数。除了如此设置之外，还可以采用 `Signal` 对象的 `connect()` 方法进行设置。

Django的信号是同步阻塞的，不要将信号和异步任务的概念搞混。可以将二者有效结合，让程序在收到某个信号的时候启动异步任务。

配置好 `receiver` 接收函数之后，还必须将函数导入到应用中，这样就可以在每次发送信号的时候调用函数。推荐的做法是在应用配置类的 `ready()` 方法中，导入接收函数。这就需要再了解一下应用配置类。

3.2 应用配置类

Django允许为每个应用设置一个单独的应用配置类。当使用 `startapp` 命令创建一个应用时，Django会在应用目录下创建一个 `apps.py` 文件，并在其中自动设置一个名称为“首字母大写的应用名+Config”并继承 `AppConfig` 类的应用配置类。

使用应用配置类可以存储这个应用的元数据，应用配置和提供自省功能。应用配置类的官方文档

<https://docs.djangoproject.com/en/2.0/ref/applications/>。

我们已经使用 `@receiver` 装饰器注册好了信号接收函数，这个函数应该在应用一启动的时候就可以进行调用，所以要注册在应用配置类中，其他类似的需要在应用初始化阶段就调用的功能也要注册在应用配置类中。编辑 `images` 应用的 `apps.py` 文件：

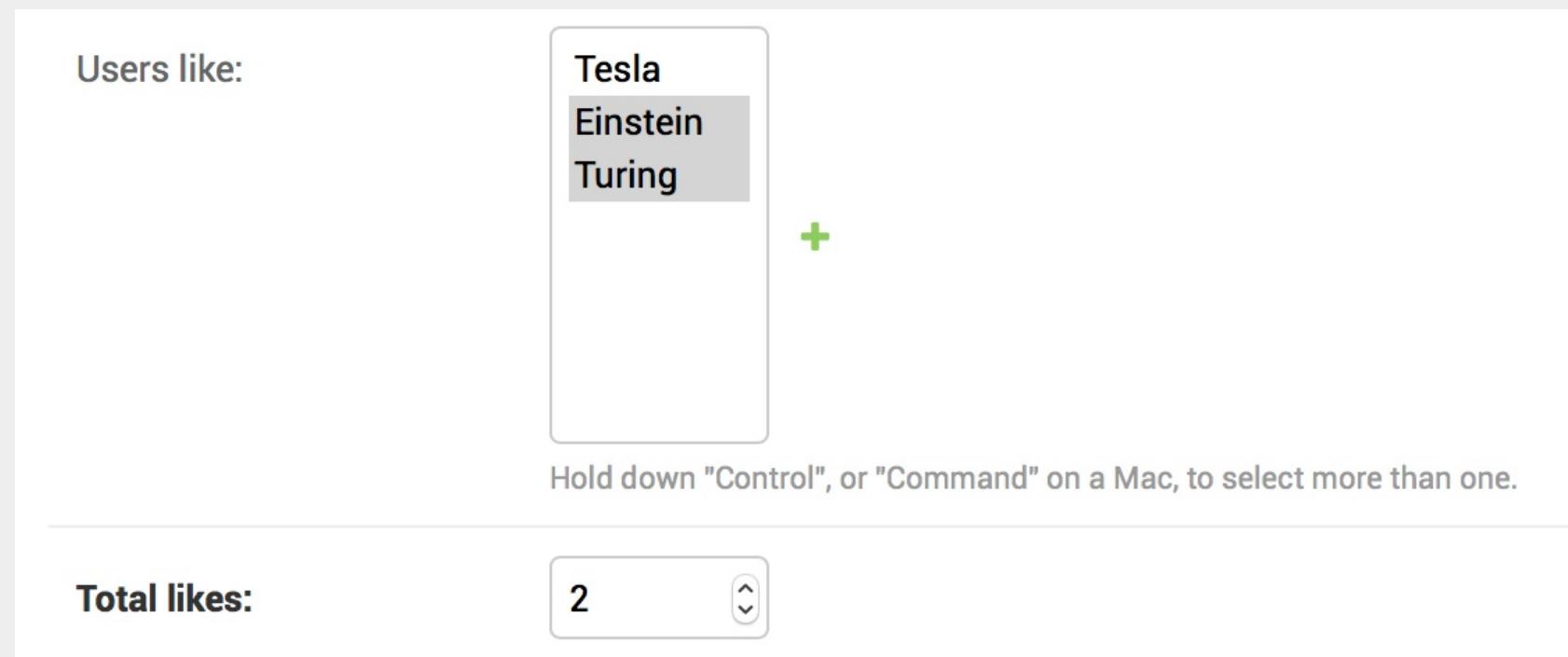
```
from django.apps import AppConfig  
  
class ImagesConfig(AppConfig):  
    name = 'images'
```

```
def ready(self):
    # 导入信号接收函数
    import images.signals
```

通过 `ready()` 方法导入之后，在 `images` 应用加载的时候该函数就会被导入。

启动程序，选中一张图片并点击LIKE按钮，然后到管理站点查看该图片，例如

<http://127.0.0.1:8000/admin/images/image/1/change/>，可以看到新增的 `total_likes` 字段。还可以看到 `total_likes` 字段已经得到了更新，如图所示：



现在可以用 `total_likes` 字段排序图片并且显示总数量，避免复杂的查询。看一下本章开头的查询语句：

```
from django.db.models import Count

images_by_popularity = Image.objects.annotate(likes=Count('users_like')).order_by('-likes')
```

现在上边的查询可以改成下边这样：

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

现在这个查询的开销要比原来小很多，这是一个使用信号的例子。

使用信号功能会让控制流变得更加难以追踪，在很多情况下，如果明确知道需要进行什么操作，无需使用信号功能。

对于已经存在表内的对象，`total_likes` 字段中还没有任何数据，需要为所有对象设置当前的值，通过 `python manage.py shell` 进入带有当前Django环境的Python命令行，并输入下列命令：

```
>>>from images.models import Image
>>>for image in Image.objects.all():
>>>    image.total_likes = image.users_like.count()
>>>    image.save()
```

现在每个图片的 `total_likes` 字段已被更新。

4 使用Redis数据库

Redis是一个先进的键值对数据库，可以存储多种类型的数据并提供高速存取服务。Redis运行时的数据保存在内存中，也可以定时将数据持久化到磁盘中或者通过日志输出。Redis相比普通的键值对存储，具有一系列强力的命令支持不同的数据格式，比如字符串、哈希值、列表、集合和有序集合，甚至是位图或HyperLogLogs数据。

尽管SQL数据库依然是保存结构化数据的最佳选择，对于迅速变化的数据、反复使用的数据和缓存需求，采用Redis有着独特的优势。本节来看一看如何通过Redis为我们的项目增加一个新功能。

4.1 安装Redis

在 <https://redis.io/download> 下载最新的Redis数据库，解压 `tar.gz` 文件，进入 `redis` 目录，然后使用 `make` 命令编译安装Redis：

```
cd redis-4.0.9  
make
```

在安装完成后，在命令行中输入如下命令来初始化Redis服务：

```
src/redis-server
```

可以看到如下输出：

```
# Server initialized  
* Ready to accept connections
```

说明Redis服务已经启动。Redis默认监听 [6379](#) 端口。可以使用 `--port` 参数指定端口，例如 `redis-server --port 6655`。

保持Redis服务运行，新开一个系统终端窗口，启动Redis客户端：

```
src/redis-cli
```

可以看到如下提示：

```
127.0.0.1:6379>
```

说明已经进入Redis命令行模式，可以直接执行Redis命令，我们来试验一些命令：

译者注：Redis官方未提供Windows版本，可以在<https://github.com/MicrosoftArchive/redis/releases>找到Windows版，安装好之后默认已经添加Redis服务，默认端口号和Linux系统一样是6379。进入cmd，输入redis-cli进入Redis命令行模式。

使用SET命令保存一个键值对：

```
127.0.0.1:6379> SET name "Peter"
OK
```

上边的命令创建了一个name键，值是字符串"Peter"。OK表示这个键值对已被成功存储。可以使用GET命令取出该键值对：

```
127.0.0.1:6379> GET name
"Peter"
```

使用EXIST命令检测某个键是否存在，返回整数1表示True，0表示False：

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

使用 `EXPIRE` 设置一个键值对的过期秒数。还可以使用 `EXPIREAT` 以UNIX时间戳的形式设置过期时间。过期时间对于将 Redis 作为缓存时很有用：

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

等待超过2秒钟，然后尝试获取该键：

```
127.0.0.1:6379> GET name
(nil)
```

(`nil`) 说明是一个null响应，即没有找到该键。使用 `DEL` 命令可以删除键和值，如下：

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total
(integer) 1
127.0.0.1:6379> GET total
(nil)
```

这是Redis的基础操作，Redis对于各种数据类型有很多命令，可以在 <https://redis.io/commands> 查看命令列表，Redis所有支持的数据格式在 <https://redis.io/topics/data-types>。

译者注：特别要看一下Redis中有序集合这个数据类型，以下会使用到。

4.2 通过Python操作Redis

同使用PostgreSQL一样，在Python安装支持该数据库的模块 `redis-py`：

```
pip install redis==2.10.6
```

该模块的文档可以在 <https://redis-py.readthedocs.io/en/latest/> 找到。

`redis-py` 提供了两大功能模块，`StrictRedis` 和 `Redis`，功能完全一样。区别是前者只支持标准的Redis命令和语法，后者进行了一些扩展。我们使用严格遵循标准Redis命令的 `StrictRedis` 模块，打开Python命令行界面，输入以下命令：

```
>>> import redis  
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
```

上述命令使用本机地址和端口和数据库编号实例化数据库连接对象，在Redis内部，数据库的编号是一个整数，共有0-16号数据库，默认客户端连接到的数据库是 `0` 号数据库，可以通过修改 `redis.conf` 更改默认数据库。

通过Python存入一个键值对：

```
>>> r.set('foo', 'bar')  
True
```

返回 `True` 表示成功存入键值对，通过 `get()` 方法取键值对：

```
>>> r.get('foo')  
b'bar'
```

可以看到，这些方法源自同名的标准Redis命令。

了解Python中使用Redis之后，需要把Redis集成到Django中来。编辑 `bookmarks` 应用的 `settings.py` 文件，添加如下设置：

```
REDIS_HOST = 'localhost'  
REDIS_PORT = 6379  
REDIS_DB = 0
```

这是Redis服务的相关设置。

4.3 在Redis中存储图片浏览次数

我们需要存储一个图片被浏览过的总数。如果我们使用Django ORM来实现，每次展示一个图片时，需要通过视图执行SQL的 `UPDATE` 语句并写入磁盘。如果我们使用Redis，只需要每次对保存在内存中的一个数字增加1，相比之下Redis的速度要快很多。

编辑 `images` 应用的 `views.py` 文件，在最上边的导入语句后边添加如下内容：

```
import redis  
from django.conf import settings  
  
r = redis.StrictRedis(host=settings.REDIS_HOST, port=settings.REDIS_PORT, db=settings.REDIS_DB)
```

通过上述语句，在视图文件中实例化了一个Redis数据库连接对象，等待其他视图的调用。编辑`image_detail`视图，让其看起来如下：

```
@login_required  
def image_detail(request, id, slug):  
    image = get_object_or_404(Image, id=id, slug=slug)  
    # 浏览数+1
```

```
total_views = r.incr('image:{}:views'.format(image.id))
return render(request, 'images/image/detail.html',
             {'section': 'images', 'image': image, 'total_views': total_views})
```

这个视图使用了 `incr` 命令，将该键对应的值增加1。如果键不存在，会自动创建该键（初始值为0）然后将值加1。`incr()`方法返回增加1这个操作之后的结果，也就是最新的浏览总数。然后用 `total_views` 存储浏览总数并传入模板。我们采用Redis的常用格式创建键名，如 `object-type:id:field`（例如 `image:33:id`）。

Redis数据库的键常用冒号分割的字符串来创建类似于带有命名空间一样的键值，这样的键名易于阅读，而且在其名字中有共同的部分，便于对应至具体对象和查找。

编辑 `images/image/detail.html`，在 `` 之后追加：

```
<span class="count">
  {{ total_views }} view{{ total_views|pluralize }}
</span>
```

打开一个图片的详情页面，然后按F5刷新几次，能够看到访问数“* views”不断上升，如下图所示：

Django and Duke



0 likes

16 views

LIKE

Django and Duke image.

Nobody likes this image yet.

现在我们就将Redis集成到Django中并用其显示数量了。

4.4 在Redis中存储排名

现在用Redis来实现一个更复杂一些的功能：创建一个排名，按照图片的访问量将图片进行排名。为了实现这个功能，将使用Redis的有序集合数据类型。有序集合是一个不重复的字符串集合，其中的每一个字符串都对应一个分数，按照分数的大小进行排序。

编辑 `images` 应用里的 `views.py` 文件，继续修改 `image_detail` 视图：

```
@login_required  
def image_detail(request, id, slug):
```

```
image = get_object_or_404(Image, id=id, slug=slug)
total_views = r.incr('image:{}:views'.format(image.id))
# 在有序集合image_ranking里，把image.id的分数增加1
r.zincrby('image_ranking', image.id, 1)
return render(request, 'images/image/detail.html',
{'section': 'images', 'image': image, 'total_views': total_views})
```

使用 `zincrby` 方法创建一个 `image_ranking` 有序集合对象，在其中存储图片的id，然后将对应的分数加1。这样就可以在每次图片被浏览之后，更新该图片被浏览的次数以及所有图片被浏览的次数的排名。

在当前的 `views.py` 文件里创建一个新的视图用于展示图片排名：

```
@login_required
def image_ranking(request):
    # 获得排名前十的图片ID列表
    image_ranking = r.zrange('image_ranking', 0, -1, desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # 取排名最高的图片然后排序
    most_viewed = list(Image.objects.filter(id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request, 'images/image/ranking.html', {'section': 'images', 'most_viewed': most_viewed})
```

这个 `image_ranking` 视图工作逻辑如下：

1. 使用 `zrange()` 命令从有序集合中取元素，后边的两个参数表示开始和结束索引，给出 `0` 到 `-1` 的范围表示取全部元素，`desc=True` 表示将这些元素降序排列。最后使用 `[:10]` 切片列表前10个元素。
2. 使用列表推导式，取得了键名对应的整数构成的列表，存入 `image_ranking_ids` 中。然后查询id属于该列表中的所有 `Image` 对象。由于要按照 `image_ranking_ids` 中的顺序对查询结果进行排序，所以使用 `list()` 将查询结果列表化。
3. 按照每个 `Image` 对象的id在 `image_ranking_ids` 中的顺序，对查询结果组成的列表进行排序。

在 `images/image/` 模板目录内创建 `ranking.html`，添加下列代码：

```
{% extends 'base.html' %}  
{% block title %}  
    Images Ranking  
{% endblock %}  
  
{% block content %}  
    <h1>Images Ranking</h1>  
    <ol>  
        {% for image in most_viewed %}  
            <li>  
                <a href="{{ image.get_absolute_url }}">{{ image.title }}</a>  
            </li>  
        {% endfor %}  
    </ol>  
{% endblock %}
```

这个页面很简单，迭代 `most_viewed` 中的每个 `Image` 对象，展示图片内容、名称和对应的详情链接。

最后为新的视图配置URL，编辑 `images` 应用的 `urls.py` 文件，增加一行：

```
path('ranking/', views.image_ranking, name='ranking'),
```

译者注：原书此处有误，`name` 参数的值设置成了 `create`，按作者的一贯写法，应该为 `'ranking'`。

之后启动站点，访问不同图片的详情页，反复刷新拖杆次，然后打开 <http://127.0.0.1:8000/images/ranking/>，即可看到排名页面：

Images ranking

1. [Chick Corea](#)
2. [Louis Armstrong](#)
3. [Al Jarreau](#)
4. [Django Reinhardt](#)
5. [Django and Duke](#)

4.5 进一步使用Redis

Redis无法替代SQL数据库，但其使用内存存储的特性可以用来完成模型具体任务，把Redis加入到你的工具库里，在必要的时候就可以使用它。下边是一些适合使用Redis的场景：

- 计数：从我们的例子可以看出，使用Redis管理计数非常便捷，`incr()` 和 `incrby()` 方法可以方便的实现计数功能。
- 存储最新的项目：使用`lpush()` 和 `rpush()` 可以向一个队列的开头和末尾追加数据，`lpop()` 和 `rpop()` 则是从队列开始和末尾弹出元素。如果操作造成队列长度改变，还可以用`ltrim()` 保持队列长度。
- 队列：除了上边的`pop` 和 `push` 系列方法，Redis还提供了阻塞队列的方法
- 缓存：`expire()` 和 `expireat()` 方法让用户可以把Redis当做缓存来使用，还可以找到一些第三方开发的将Redis配置为Django缓存后端的模块。
- 订阅/发布：Redis提供订阅/发布消息模式，可以向一些频道发送消息，订阅该频道的Redis客户端可以接受到该消息。

- 排名和排行榜：Redis的有序集合可以方便的创建排名相关的数据。
- 实时跟踪：Redis的高速I/O可以用在实时追踪并更新数据方面。

总结

这一章里完成了两大任务，一个是用户之间的互相关注系统，一个是用户行为流系统。还学习了使用Django的信号功能，和将Redis集成至Django。

在下一章，我们将开始一个新的项目，创建一个电商网站。将学习创建商品品类，通过session创建购物车，以及使用Celery启动异步任务。

第七章 创建电商网站

在上一章里，创建了用户关注系统和行为流应用，还学习了使用Django的信号功能与使用Redis数据库存储图片浏览次数和排名。这一章将学习如何创建一个基础的电商网站。本章将学习创建商品品类目录，通过session实现购物车功能。还将学习创建自定义上下文管理器和使用Celery执行异步任务。

本章的要点有：

- 创建商品品类目录
- 使用session创建购物车
- 管理客户订单
- 使用Celery异步向用户发送邮件通知

1 创建电商网站项目

我们要创建一个电商网站项目。用户能够浏览商品品类目录，然后将具体商品加入购物车，最后还可以通过购物车生成订单。本章电商网站的如下功能：

- 创建商品品类模型并加入管理后台，创建视图展示商品品类
- 创建购物车系统，用户浏览网站时购物车中一直保存着用户的商品
- 创建提交订单的页面
- 订单提交成功后异步发送邮件给用户

打开系统命令行窗口，为新项目配置一个新的虚拟环境并激活：

```
mkdir env  
virtualenv env/myshop  
source env/myshop/bin/activate
```

然后在虚拟环境中安装Django：

```
pip install Django==2.0.5
```

新创建一个项目叫做 `myshop`，之后创建新应用叫 `shop`：

```
django-admin startproject myshop  
cd myshop/  
django-admin startapp shop
```

编辑 `settings.py` 文件，激活 `shop` 应用：

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
]
```

现在应用已经激活，下一步是设计数据模型。

1.1 创建商品品类模型

我们的商品品类模型包含一系列商品大类，每个商品大类中包含一系列商品。每一个商品都有一个名称，可选的描述，可选的图片，价格和是否可用属性。编辑 `shop` 应用的 `models.py` 文件：

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True, unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name


class Product(models.Model):
    category = models.ForeignKey(Category, related_name='category', on_delete=models.CASCADE)
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('name',)
        index_together = (('id', 'slug'),)

    def __str__(self):
        return self.name
```

这是我们的 `Category` 和 `Product` 模型。 `Category` 包含 `name` 字段和设置为不可重复的 `slug` 字段（`unique` 同时也意味着创建索引）。 `Product` 模型的字段如下：

- `category`：关联到 `Category` 模型的外键。这是一个多对一关系，一个商品必定属于一个品类，一个品类包含多个商品。
- `name`：商品名称。
- `slug`：商品简称，用于创建规范化URL。
- `image`：可选的商品图片。
- `description`：可选的商品图片。
- `price`：该字段使用了Python的 `decimal.Decimal` 类，用于存储商品的金额，通过 `max_digits` 设置总位数，`decimal_places=2` 设置小数位数。
- `available`：布尔值，表示商品是否可用，可以用于切换该商品是否可以购买。
- `created`：记录商品对象创建的时间。
- `updated`：记录商品对象最后更新的时间。

这里需要特别说明的是 `price` 字段，使用 `DecimalField`，而不是 `FloatField`，以避免小数尾差。

凡是涉及到金额相关的数值，使用 `DecimalField` 字段。`FloatField` 的后台使用Python的 `float` 类型，而 `DecimalField` 字段后台使用Python的 `Decimal` 类，可以避免出现浮点数的尾差。

在 `Product` 模型的 `Meta` 类中，使用 `index_together` 设置 `id` 和 `slug` 字段建立联合索引，这样在同时使用两个字段的索引时会提高效率。

由于使用了 `ImageField`，还需要安装 `Pillow` 库：

```
pip install Pillow==5.1.0
```

之后执行数据迁移程序，创建数据表。

1.2 将模型注册到管理后台

将我们的模型都添加到管理后台中，编辑 `shop` 应用的 `admin.py` 文件：

```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}


@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

我们使用了 `prepopulated_fields` 用于让 `slug` 字段通过 `name` 字段自动生成，在之前的项目中可以看到这么做很简便。在 `ProductAdmin` 中使用 `list_editable` 设置了可以编辑的字段，这样可以一次性编辑多行而不用点开每一个对象。注意所有在 `list_editable` 中的字段必须出现在 `list_display` 中。

之后创建超级用户。打开 <http://127.0.0.1:8000/admin/shop/product/add/>，使用管理后台添加一个新的商品品类和该品类中的一些商品，页面如下：

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Shop > Products

✓ The product "Green tea" was added successfully.

Select product to change

Action: ----- Go 0 of 1 selected

	NAME	SLUG	PRICE	STOCK	AVAILABLE	CREATED	UPDATED
<input type="checkbox"/>	Green tea	green-tea	30	22	<input checked="" type="checkbox"/>	Dec. 5, 2017, 6:17 p.m.	Dec. 5, 2017, 6:17 p.m.

1 product

ADD PRODUCT +

FILTER

By available

- All
- Yes
- No

By created

- Any date
- Today
- Past 7 days
- This month
- This year

By updated

- Any date
- Today
- Past 7 days
- This month
- This year

译者注：这里图片上有一个 `stock` 字段，这是上一版的程序使用的字段。在本书内程序已经修改，但图片依然使用了上一版的图片。本项目中后续并没有使用 `stock` 字段。

1.3 创建商品品类视图

为了展示商品，我们创建一个视图，用于列出所有商品，或者根据品类显示某一品类商品，编辑 `shop` 应用的 `views.py` 文件：

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(categories, slug=category_slug)
        products = products.filter(category=category)
    return render(request, 'shop/product/list.html',
                  {'category': category, 'categories': categories, 'products': products})
```

这个视图逻辑较简单，使用了 `available=True` 筛选所有可用的商品。设置了一个可选的 `category_slug` 参数用于选出特定的品类。

还需要一个展示单个商品详情的视图，继续编辑 `views.py` 文件：

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id, slug=slug, available=True)
    return render(request, 'shop/product/detail.html', {'product': product})
```

`product_detail` 视图需要 `id` 和 `slug` 两个参数来获取商品对象。只通过ID可以获得商品对象，因为ID是唯一的，这里增加了 `slug` 字段是为了对搜索引擎优化。

在创建了上述视图之后，需要为其配置URL，在 `shop` 应用内创建 `urls.py` 文件并添加如下内容：

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

app_name = 'shop'
```

```
urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('<slug:category_slug>/', views.product_list, name='product_list_by_category'),
    path('<int:id>/<slug:slug>/', views.product_detail, name='product_detail'),
]
```

我们为 `product_list` 视图定义了两个不同的URL，一个名称是 `product_list`，不带任何参数，表示展示全部品类的全部商品；一个名称是 `product_list_by_category`，带参数，用于显示指定品类的商品。还为 `product_detail` 视图配置了传入 `id` 和 `slug` 参数的URL。

这里要解释的就是 `product_list` 视图带一个默认值参数，所以默认路径进来后就是展示全部品类的页面。加上了具体某个品类，就展示那个品类的商品。详情页的URL使用 `id` 和 `slug` 来进行参数传递。

还需要编写项目的一级路由，编辑 `myshop` 项目的根 `urls.py` 文件：

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

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

我们为 `shop` 应用配置了名为 `shop` 的二级路由。

由于URL中有参数，就需要配置URL反向解析，编辑 `shop` 应用的 `models.py` 文件，导入 `reverse()` 函数，然后为 `Category` 和 `Product` 模型编写 `get_absolute_url()` 方法：

```
from django.urls import reverse

class Category(models.Model):
    # .....
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category', args=[self.slug])

class Product(models.Model):
    # .....
    def get_absolute_url(self):
        return reverse('shop:product_detail', args=[self.id, self.slug])
```

这样就为模型的对象配置好了用于反向解析URL的方法，我们已经知道，`get_absolute_url()` 是很好的获取具体对象规范化URL的方法。

1.4 创建商品品类模板

现在需要创建模板，在`shop`应用下建立如下目录和文件结构：

```
templates/
    shop/
        base.html
    product/
        list.html
        detail.html
```

像以前的项目一样，`base.html` 是母版，让其他的模板继承母版。编辑`base.html`：

```
{% load static %}
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="utf-8"/>
    <title>{% block title %}My shop{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        <a href="/" class="logo">My shop</a>
    </div>
    <div id="subheader">
        <div class="cart">Your cart is empty.</div>
    </div>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
</body>
</html>
```

这是这个项目的母版。其中使用的CSS文件可以从随书源代码中复制到 `shop` 应用的 `static/` 目录下。

然后编辑 `shop/product/list.html` :

```
{% extends "shop/base.html" %}
{% load static %}
{% block title %}
    {% if category %}{{ category.name }}{% else %}Products{% endif %}
{% endblock %}
{% block content %}
    <div id="sidebar">
        <h3>Categories</h3>
        <ul>
            <li {% if not category %}class="selected"{% endif %}>
                <a href="{% url "shop:product_list" %}">All</a>
```

```
</li>
    {% for c in categories %}
        <li {% if category.slug == c.slug %}class="selected"
            {% endif %}>
            <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
        </li>
    {% endfor %}
</ul>
</div>
<div id="main" class="product-list">
    <h1>{% if category %}{{ category.name }}{% else %}Products
    {% endif %}</h1>
    {% for product in products %}
        <div class="item">
            <a href="{{ product.get_absolute_url }}">
                <img src=""
                    {% if product.image %}{{ product.image.url }}{% else %}{% static "img/no_image.png" %}{%
                    endif %}
                </a>
            <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
            <br>
            ${{ product.price }}
        </div>
    {% endfor %}
</div>
{% endblock %}
```

这是展示商品列表的模板，继承了 `base.html`，使用 `categories` 变量在侧边栏显示品类的列表，在页面主体部分通过 `products` 变量展示商品清单。展示所有商品和具体某一类商品都采用这个模板。如果 `Product` 对象的 `image` 字段为空，我们显示一张默认的图片，可以在随书源码中找到 `img/no_image.png`，将其拷贝到对应的目录。

由于使用了 Imagefield，还需要对媒体文件进行一些设置，编辑 `settings.py` 文件加入下列内容：

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

`MEDIA_URL` 是保存用户上传的媒体文件的目录，`MEDIA_ROOT` 是存放媒体文件的目录，通过 `BASE_DIR` 变量动态建立该目录。

为了让Django提供静态文件服务，还必须修改 `shop` 应用的 `urls.py` 文件：

```
from django.conf import settings  
from django.conf.urls.static import static  
urlpatterns = [  
    # ...  
]  
if settings.DEBUG:  
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

注意仅在开发阶段才能如此设置。在生产环境中不能使用Django提供静态文件。使用管理后台增加一些商品，然后打开 <http://127.0.0.1:8000/>，可以看到如下页面：

Your cart is empty.

Products

Categories

All

Tea



Green tea
\$30



Red tea
\$45.5



Tea powder
\$21.2

如果没有给商品上传图片，则会显示 no_image.png，如下图：

NO IMAGE
AVAILABLE



Green tea
\$30



Red tea
\$45.5

Tea powder
\$21.2

然后编写商品详情页 `shop/product/detail.html` :

```
{% extends "shop/base.html" %}  
{% load static %}  
{% block title %}  
    {{ product.name }}  
{% endblock %}  
{% block content %}  
    <div class="product-detail">  
          
        <h1>{{ product.name }}</h1>  
        <h2><a href="{{ product.category.get_absolute_url }}">{{ product.category }}</a></h2>  
        <p class="price">${{ product.price }}</p>  
        {{ product.description|linebreaks }}  
    </div>  
{% endblock %}
```

在模板中调用 `get_absolute_url()` 方法用于展示对应类的商品，打开 <http://127.0.0.1:8000/>，然后点击任意一个商品，详情页如下：



The screenshot shows a product detail page for 'Red tea'. At the top left is a navigation bar with 'My shop'. On the right, it says 'Your cart is empty.' Below the image, the product name 'Red tea' is displayed in large bold letters, followed by its category 'Tea' in a smaller blue font. The price '\$45.5' is shown in large bold letters below that. A detailed product description follows: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'

现在已经将商品品类和展示功能创建完毕。

2 创建购物车功能

在建立商品品类之后，下一步是创建一个购物车，让用户可以将指定的商品及数量加入购物车，而且在浏览整个网站并且下订单之前，购物车都会维持其中的信息。为此，我们需要将购物车数据存储在当前用户的session中。

由于session通用翻译成会话，而在本章中很多时候session指的是Django的session模块或者session对象，所以不再进行翻译。

我们将使用Django的session框架来存储购物车数据。直到用户生成订单，商品信息都存储在购session中，为此我们还需要为购物车和其中的商品创建一个模型。

2.1 使用Django的session模块

Django 提供了一个session模块，用于进行匿名或登录用户会话，可以为每个用户保存独立的数据。session数据存储在服务端，通过在cookie中包含session ID就可以获取到session，除非将session存储在cookie中。session中间件管理具体的cookie信息，默认的session引擎将session保存在数据库内，也可以切换不同的session引擎。

要使用session，需要在 `settings.py` 文件的 `MIDDLEWARE` 设置中启

用 `'django.contrib.sessions.middleware.SessionMiddleware'`，这个管理session中间件在使用 `startproject` 命令创建项目时默认已经被启用。

这个中间件在 `request` 对象中设置了 `session` 属性用于访问session数据，类似于一个字典一样，可以存储任何可以被序列化为JSON的Python数据类型。可以像这样存入数据：

```
request.session['foo'] = 'bar'
```

获取键对应的值：

```
request.session.get('foo')
```

删除一个键值对：

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

可以将`request.session`当成字典来操作。

当用户登录到一个网站的时候，服务器会创建一个新的用于登录用户的session信息替代原来的匿名用户session信息，这意味着原session信息会丢失。如果想保存原session信息，需要在登录的时候将原session信息存为一个新的session数据。

2.2 session设置

Django中可以配置session模块的一些参数，其中最重要的是`SESSION_ENGINE`设置，即设置session数据具体存储在何处。默认情况下，Django通过`django.contrib.session`应用的`Session`模型，将session数据保存在数据库中的`django_session`数据表中。

Django提供了如下几种存储session数据的方法：

- Database sessions：session数据存放于数据库中，为默认设置，即将session数据存放到`settings.py`中的`DATABASES`设置中的数据库内。
- File-based sessions：保存在一个具体的文件中
- Cached sessions：基于缓存的session存储，使用Django的缓存系统，可以通过`CACHES`设置缓存后端。这种情况下效率最高。
- Cached database sessions：先存到缓存再持久化到数据库中。取数据时如果缓存内无数据，再从数据库中取。
- Cookie-based sessions：基于cookie的方式，session数据存放在cookie中。

为了提高性能，使用基于缓存的session是好的选择。Django直接支持基于Memcached的缓存和如Redis的第三方缓存后端。

还有其他一系列的session设置，以下是一些主要的设置：

- `SESSION_COOKIE_AGE` : session过期时间，为秒数，默认为 1209600 秒，即两个星期。
- `SESSION_COOKIE_DOMAIN` : 默认为 `None`，设置为某个域名可以启用跨域cookie。
- `SESSION_COOKIE_SECURE` : 布尔值，默认为 `False`，表示是否只允许HTTPS连接下使用session
- `SESSION_EXPIRE_AT_BROWSER_CLOSE` : 布尔值，默认为 `False`，表示是否一旦浏览器关闭，session就失效
- `SESSION_SAVE_EVERY_REQUEST` : 布尔值，默认为 `False`，设置为 `True` 表示每次HTTP请求都会更新session，其中的过期时间相关设置也会一起更新。

可以在 <https://docs.djangoproject.com/en/2.0/ref/settings/#sessions> 查看所有的session设置和默认值。

2.3 session过期

特别需要提的是 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置。该设置默认为 `False`，此时session有效时间采用 `SESSION_COOKIE_AGE` 中的设置。

如果将 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `True`，则session在浏览器关闭后就失效，`SESSION_COOKIE_AGE` 设置不起作用。

还可以使用 `request.session.set_expiry()` 方法设置过期时间。

2.4 在session中存储购物车数据

我们需要创建一个简单的数据结构，可以被JSON序列化，用于存放购物车数据。购物车中必须包含如下内容：

- `Product` 对象的ID
- 商品的数量
- 商品的单位价格

由于商品的价格会变化，我们在将商品加入购物车的同时存储当时商品的价格，如果商品价格之后再变动，也不进行处理。

现在需要实现创建购物车和为session添加购物车的功能，购物车按照如下方式工作：

1. 当需要创建一个购物车的时候，先检查session中是否存在自定义的购物车键，如果存在说明当前用户已经使用了购物车，如果不存在，就新建一个购物车键。
2. 对于接下来的HTTP请求，都要重复第一步，并且从购物车中保存的商品ID到数据库中取得对应的 `Product` 对象数据。

编辑 `settings.py` 里新增一行：

```
CART_SESSION_ID = 'cart'
```

这就是我们的购物车键名称，由于session对于每个用户都通过中间件管理，所以可以在所有用户的session里都使用统一的这个名称。

然后新建一个应用来管理购物车，启动系统命令行并创建新应用 `cart`：

```
python manage.py startapp cart
```

然后在 `settings.py` 中激活该应用：

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
]
```

在 `cart` 应用中创建 `cart.py`，添加如下代码：

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart:

    def __init__(self):
        """
        初始化购物车对象
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # 向session中存入空白购物车数据
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
```

这是我们用于管理购物车的Cart类，使用request对象进行初始化，使用 `self.session = request.session` 让类中的其他方法可以访问session数据。首先，使用 `self.session.get(settings.CART_SESSION_ID)` 尝试获取购物车对象。如果不存在购物车对象，通过为购物车键设置一个空白字段对象从而新建一个购物车对象。我们将使用商品ID作为字典中的键，其值又是一个由数量和价格构成的字典，这样可以保证不会重复生成同一个商品的购物车数据，也简化了取出购物车数据的方式。

创建将商品添加到购物车和更新数量的方法，为Cart类添加 `add()` 和 `save()` 方法：

```
class Cart:
    # .....
    def add(self, product, quantity=1, update_quantity=False):
        """
```

向购物车中增加商品或者更新购物车中的数量

```
"""
product_id = str(product.id)
if product_id not in self.cart:
    self.cart[product_id] = {'quantity': 0, 'price': str(product.price)}
if update_quantity:
    self.cart[product_id]['quantity'] = quantity
else:
    self.cart[product_id]['quantity'] += quantity
self.save()

def save(self):
    # 设置session.modified的值为True，中间件在看到这个属性的时候，就会保存session
    self.session.modified = True
```

`add()` 方法接受以下参数：

- `product`：要向购物车内添加或更新的 `product` 对象
- `quantity`：商品数量，为整数，默认值为1
- `update_quantity`：布尔值，为 `True` 表示要将商品数量更新为 `quantity` 参数的值，为 `False` 表示将当前数量增加 `quantity` 参数的值。

我们把商品的ID转换成字符串形式然后作为购物车中商品键名，这是因为Django使用JSON序列化session数据，而JSON只允许字符串作为键名。商品价格也被从 `decimal` 类型转换为字符串，同样是为了序列化。最后，使用 `save()` 方法把购物车数据保存进session。

`save()` 方法中修改了 `session.modified = True`，中间件通过这个判断session已经改变然后存储session数据。

我们还需要从购物车中删除商品的方法，为 `Cart` 类添加以下方法：

```
class Cart:  
    # .....  
    def remove(self, product):  
        """  
        从购物车中删除商品  
        """  
        product_id = str(product.id)  
        if product_id in self.cart:  
            del self.cart[product_id]  
        self.save()
```

`remove()` 根据id从购物车中移除对应的商品，然后调用 `save()` 方法保存session数据。

为了使用方便，我们会需要遍历购物车内的所有商品，用于展示等操作。为此需要在 `Cart` 类内定义 `__iter__()` 方法，生成迭代器，供将for循环使用。

```
class Cart:  
    # .....  
    def __iter__(self):  
        """  
        遍历所有购物车中的商品并从数据库中取得商品对象  
        """  
        product_ids = self.cart.keys()  
        # 获取购物车内的所有商品对象  
        products = Product.objects.filter(id__in=product_ids)  
  
        cart = self.cart.copy()  
        for product in products:  
            cart[str(product.id)]['product'] = product  
  
        for item in cart.values():  
            item['price'] = Decimal(item['price'])
```

```
    item['total_price'] = item['price'] * item['quantity']
    yield item
```

在 `__iter__` 方法中，获取了当前购物车中所有商品的Product对象。然后浅拷贝了一份 `cart` 购物车数据，并为其中的每个商品添加了键为 `product`，值为商品对象的键值对。最后迭代所有的值，为把其中的价格转换为 `decimal` 类，增加一个 `total_price` 键来保存总价。这样我们就可以迭代购物车对象了。

还需要显示购物车中有几件商品。当执行 `len()` 方法的时候，Python会调用对象的 `__len__()` 方法，为 `Cart` 类添加如下的 `__len__()` 方法：

```
class Cart:
    # .....
    def __len__(self):
        """
        购物车内一共有几种商品
        """
        return sum(item['quantity'] for item in self.cart.values())
```

这个方法返回所有商品的数量的合计。

再编写一个计算购物车商品总价的方法：

```
class Cart:
    # .....
    def get_total_price(self):
        return sum(Decimal(item['price'])*item['quantity'] for item in self.cart.values())
```

最后，再编写一个清空购物车的方法：

```
class Cart:  
    # .....  
    def clear(self):  
        del self.session[settings.CART_SESSION_ID]  
        self.save()
```

现在就编写完了用于管理购物车的 `Cart` 类。

译者注，原书的代码采用 `class Cart(object)` 的写法，译者将其修改为 Python 3 的新式类编写方法。

2.5 创建购物车视图

现在我们拥有了管理购物车的 `Cart` 类，需要创建如下的视图来添加、更新和删除购物车中的商品

- 添加商品的视图，可以控制增加或者更新商品数量
- 删除商品的视图
- 详情视图，显示购物车中的商品和总金额等信息

2.5.1 购物车相关视图

为了向购物车内增加商品，显然需要一个表单让用户选择数量并按下添加到购物车的按钮。在 `cart` 应用中创建 `forms.py` 文件并添加如下内容：

```
from django import forms  
  
PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]  
  
class CartAddProductForm(forms.Form):
```

```
quantity = forms.TypedChoiceField(choices=PRODUCT_QUANTITY_CHOICES, coerce=int)
update = forms.BooleanField(required=False, initial=False, widget=forms.HiddenInput)
```

使用该表单添加商品到购物车，这个CartAddProductForm表单包含如下两个字段：

- `quantity`：限制用户选择的数量为1-20个。使用 `TypedChoiceField` 字段，并且设置 `coerce=int`，将输入转换为整型字段。
- `update`：用于指定当前数量是增加到原有数量（`False`）上还是替代原有数量（`True`），把这个字段设置为 `HiddenInput`，因为我们不需要用户看到这个字段。

创建向购物车中添加商品的视图，编写 `cart` 应用中的 `views.py` 文件，添加如下代码：

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .form import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product, quantity=cd['quantity'], update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

这是添加商品的视图，使用 `@require_POST` 使该视图仅接受 `POST` 请求。这个视图接受商品ID作为参数，ID取得商品对象之后验证表单。表单验证通过后，将商品添加到购物车，然后跳转到购物车详情页面对应的 `cart_detail` URL，稍后我们会来编写 `cart_detail` URL。

再来编写删除商品的视图，在 `cart` 应用的 `views.py` 中添加如下代码：

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

删除商品视图同样接受商品ID作为参数，通过ID获取 `Product` 对象，删除成功之后跳转到 `cart_detail` URL。

还需要一个展示购物车详情的视图，继续在 `cart` 应用的 `views.py` 文件中添加下列代码：

```
def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

`cart_detail` 视图用来展示当前购物车中的详情。现在已经创建了添加、更新、删除及展示的视图，需要配置URL，在 `cart` 应用里新建 `urls.py`：

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

app_name = 'cart'
urlpatterns = [
    path('', views.cart_detail, name='cart_detail'),
    path('add/<int:product_id>/', views.cart_add, name='cart_add'),
    path('remove/<int:product_id>/', views.cart_remove, name='cart_remove'),
]
```

然后编辑项目的根 `urls.py`，配置URL：

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('cart//', include('cart.urls', namespace='cart')),
    path('', include('shop.urls', namespace='shop')),
]
```

注意这一条路由需要增加在 `shop.urls` 路径之前，因为这一条比下一条的匹配路径更加严格。

2.5.2 创建展示购物车的模板

`cart_add` 和 `cart_remove` 视图并未渲染模板，而是重定向到 `cart_detail` 视图，我们需要为编写展示购物车详情的模板。

在 `cart` 应用内创建如下文件目录结构：

```
templates/
  cart/
    detail.html
```

编辑 `cart/detail.html`，添加下列代码：

```
{% extends 'shop/base.html' %}

{% load static %}

{% block title %}
  Your shopping cart
{% endblock %}

{% block content %}
```

```
<h1>Your shopping cart</h1>
<table class="cart">
    <thead>
        <tr>
            <th>Image</th>
            <th>Product</th>
            <th>Quantity</th>
            <th>Remove</th>
            <th>Unit price</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        {% for item in cart %}
            {% with product=item.product %}
                <tr>
                    <td>
                        <a href="{{ product.get_absolute_url }}>
                            
                        </a>
                    </td>
                    <td>{{ product.name }}</td>
                    <td>{{ item.quantity }}</td>
                    <td>
                        <a href="{% url 'cart:cart_remove' product.id %}>Remove</a>
                    </td>
                    <td class="num">${{ item.price }}</td>
                    <td class="num">${{ item.total_price }}</td>
                </tr>
            {% endwith %}
        {% endfor %}

        <tr class="total">
```

```
<td>total</td>
<td colspan="4"></td>
<td class="num">${{ cart.get_total_price }}</td>
</tr>
</tbody>
</table>
<p class="text-right">
    <a href="{% url 'shop:product_list' %}" class="button light">Continue shopping</a>
    <a href="#" class="button">Checkout</a>
</p>
{% endblock %}
```

这是展示购物车详情的模板，包含了一个表格用于展示具体商品。用户可以通过表单修改之中的数量，并将其发送至 `cart_add` 视图。还提供了一个删除链接供用户删除商品。

2.5.3 添加商品至购物车

需要修改商品详情页，增加一个Add to Cart按钮。编辑 `shop` 应用的 `views.py` 文件，把 `CartAddProductForm` 添加到 `product_detail` 视图中：

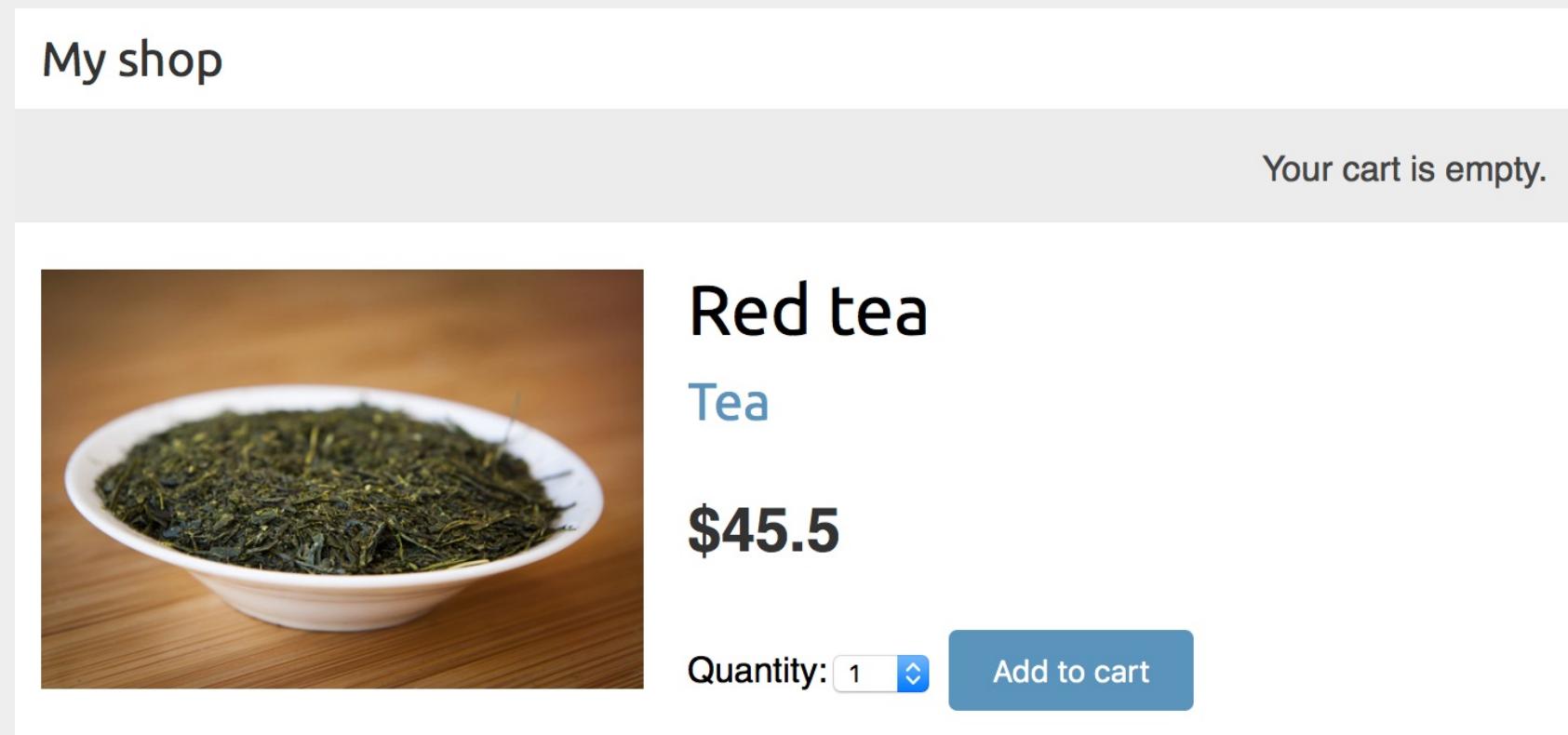
```
from cart.forms import CartAddProductForm

def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id, slug=slug, available=True)
    cart_product_form = CartAddProductForm()
    return render(request, 'shop/product/detail.html', {'product': product, 'cart_product_form': cart_product_form})
```

编辑对应的 `shop/templates/shop/product/detail.html` 模板，在展示商品价格之后添加如下内容：

```
<p class="price">${{ product.price }}</p>
<form action="{% url 'cart:cart_add' product.id %}" method="post">
    {{ cart_product_form }}
    {% csrf_token %}
    <input type="submit" value="Add to cart">
</form>
{{ product.description|linebreaks }}
```

启动站点，到 <http://127.0.0.1:8000/>，进入任意一个商品的详情页，可以看到商品详情页内增加了按钮，如下图：



选择一个数量，然后点击Add to cart按钮，即可购物车详情界面，如下图：

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2	Remove	\$45.5	\$91.0
Total					\$91.0

[Continue shopping](#)[Checkout](#)

2.5.4 更新商品数量

当用户在浏览购物车详情时，在下订单前很可能会修改购物车的中商品的数量，我们必须允许用户在购物车详情页修改数量。

编辑 `cart` 应用中的 `views.py` 文件，修改其中的 `cart_detail` 视图：

```
def cart_detail(request):  
    cart = Cart(request)  
    for item in cart:
```

```
for item in cart:
    item['update_quantity_form'] = CartAddProductForm(initial={'quantity': item['quantity'], 'update': True})
return render(request, 'cart/detail.html', {'cart': cart})
```

这个视图为每个购物车的商品对象添加了一个 `CartAddProductForm` 对象，这个表单使用当前数量初始化，然后将 `update` 字段设置为 `True`，这样在提交表单时，当前的数字直接覆盖原数字。

编辑 `cart` 应用的 `cart/detail.html` 模板，找到下边这行

```
<td>{{ item.quantity }}</td>
```

将其替换成：

```
<td>
<form action="{% url 'cart:cart_add' product.id %}" method="post">
    {{ item.update_quantity_form.quantity }}
    {{ item.update_quantity_form.update }}
    <input type="submit" value="Update">
    {% csrf_token %}
</form>
</td>
```

之后启动站点，到 <http://127.0.0.1:8000/cart/>，可以看到如下所示：

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	<input type="button" value="2"/> <input type="button" value="▼"/> <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$45.5	\$91.0
Total					\$91.0

[Continue shopping](#)

[Checkout](#)

修改数量然后点击Update按钮来测试新的功能，还可以尝试从购物车中删除商品。

2.6 创建购物车上下文处理器

你可能在实际的电商网站中会注意到，购物车的详细情况一直显示在页面上方的导航部分，在购物车为空的时候显示特殊的为空的字样，如果购物车中有商品，则会显示数量或者其他内容。这种展示购物车的方法与之前编写的处理购物车的视图没有关系，因此我们可以通过创建一个上下文处理器，将购物车对象作为 `request` 对象的一个属性，而不用去管是不是通过视图操作。

2.6.1 上下文处理器

Django中的上下文管理器，就是能够接受一个 `request` 请求对象作为参数，返回一个要添加到 `request` 上下文的字典的 Python函数。

当默认通过 `startproject` 启动一个项目的时候，`settings.py` 中的 `TEMPLATES` 设置中的 `context_processors` 部分，就是给模板附加上下文的上下文处理器，有这么几个：

- `django.template.context_processors.debug`：这个上下文处理器附加了布尔类型的 `debug` 变量，以及 `sql_queries` 变量，表示请求中执行的SQL查询
- `django.template.context_processors.request`：这个上下文处理器设置了 `request` 变量
- `django.contrib.auth.context_processors.auth`：这个上下文处理器设置了 `user` 变量
- `django.contrib.messages.context_processors.messages`：这个上下文处理器设置了 `messages` 变量，用于使用消息框架

除此之外，django还启用了 `django.template.context_processors.csrf` 来防止跨站请求攻击。这个组件没有写在 `settings.py` 里，强制启用，无法进行设置和关闭。有关所有上下文管理器的详情请参见
<https://docs.djangoproject.com/en/2.0/ref/templates/api/#built-in-template-context-processors>。

2.6.2 将购物车设置到request上下文中

现在我们就来设置一个自定义上下文处理器，以在所有模板内访问购物车对象。

在 `cart` 应用内新建一个 `context_processors.py` 文件，同视图，模板以及其他内容一样，django内的程序可以写在应用内的任何地方，但为了结构良好，将其单独写成一个文件：

```
from .cart import Cart
def cart(request):
    return {'cart': Cart(request)}
```

Django规定的上下文处理器，就是一个函数，接受 `request` 请求作为参数，然后返回一个字典。这个字典的键值对被 `RequestContext` 设置为所有模板都可以使用的变量及对应的值。在我们的上下文处理器中，我们使用 `request` 对象初始化了 `cart` 对象

之后在 `settings.py` 里将我们的自定义上下文处理器加到 `TEMPLATES` 设置中：

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')]

    ,
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            .....
            'cart.context_processors.cart'
        ],
    },
},
]
```

定义了上下文管理器之后，只要一个模板被 `RequestContext` 渲染，上下文处理器就会被执行然后附加上变量名 `cart`。

所有使用 `RequestContext` 的请求过程中都会执行上下文处理器。对于不是每个模板都需要的变量，一般情况下首先考虑的是使用自定义模板标签，特别是涉及到数据库查询的变量，否则会极大的影响网站的效率。

修改 `base.html`，找到下面这部分：

```
<div class="cart">
Your cart is empty.
```

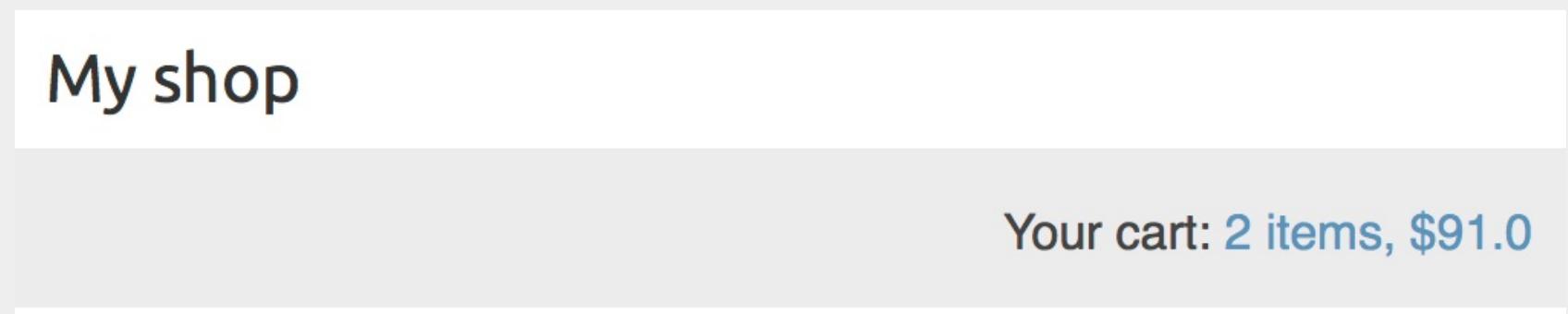
```
</div>
```

将其修改成：

```
<div class="cart">
  {% with total_items=cart|length %}
    {% if cart|length > 0 %}
      Your cart:
      <a href="{% url 'cart:cart_detail' %}">{{ total_items }} items{{ total_items|pluralize }},  

      ${{ cart.get_total_price }}
      </a>
    {% else %}
      Your cart is empty.
    {% endif %}
  {% endwith %}
</div>
```

启动站点，到 <http://127.0.0.1:8000/>，添加一些商品到购物车，在网站的标题部分可以显示出购物车的信息：



3 生成客户订单

当用户准备对一个购物车内的商品进行结账的时候，需要生成一个订单数据保存到数据库中。订单必须保存用户信息和用户所购买的商品信息。

为了实现订单功能，新创建一个订单应用：

```
python manage.py startapp orders
```

然后在 `settings.py` 中的 `INSTALLED_APPS` 中进行激活：

```
INSTALLED_APPS = [
    # ...
    'orders.apps.OrdersConfig',
]
```

3.1 创建订单模型

我们用一个模型存储订单的详情，然后再用一个模型保存订单内的商品信息，包括价格和数量。编辑 `orders` 应用的 `models.py` 文件：

```
from django.db import models
from shop.models import Product

class Order(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
```

```
created = models.DateTimeField(auto_now_add=True)
updated = models.DateTimeField(auto_now=True)
paid = models.BooleanField(default=False)

class Meta:
    ordering = ('-created',)

def __str__(self):
    return 'Order {}'.format(self.id)

def get_total_cost(self):
    return sum(item.get_cost() for item in self.items.all())

class OrderItem(models.Model):
    order = models.ForeignKey(Order, related_name='items', on_delete=models.CASCADE)
    product = models.ForeignKey(Product, related_name='order_items', on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return '{}'.format(self.id)

    def get_cost(self):
        return self.price * self.quantity
```

`Order` 模型包含一些存储用户基础信息的字段，以及一个是否支付的布尔字段 `paid`。稍后将在支付系统中使用该字段区分订单是否已经付款。还定义了一个获得总金额的方法 `get_total_cost()`，通过该方法可以获得当前订单的总金额。

`OrderItem` 存储了生成订单时候的价格和数量。然后定义了一个 `get_cost()` 方法，返回当前商品的总价。

之后执行数据迁移，过程不再赘述。

3.2 将订单模型加入管理后台

编辑 `orders` 应用的 `admin.py` 文件：

```
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                   'address', 'postal_code', 'city', 'paid',
                   'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
```

我们让 `OrderItem` 类继承了 `admin.TabularInline` 类，然后在 `OrderAdmin` 类中使用了 `inlines` 参数指定 `OrderItemInline`，通过该设置，可以将一个模型显示在相关联的另外一个模型的编辑页面中。

启动站点到 `http://127.0.0.1:8000/admin/orders/order/add/`，可以看到如下的页面：

Add order

First name:

Last name:

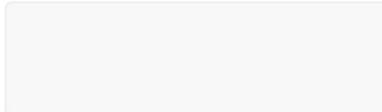
Email:

Address:

Postal code:

City:

Paid

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
<input type="text"/> 	<input type="text"/>	<input type="text" value="1"/>  	
<input type="text"/> 	<input type="text"/>	<input type="text" value="1"/>  	
<input type="text"/> 	<input type="text"/>	<input type="text" value="1"/>  	
+ Add another Order item			
		Save and add another	Save and continue editing
		SAVE	

3.3 创建客户订单视图和模板

在用户提交订单的时候，我们需要用刚创建的订单模型来保存用户当时购物车内的信息。创建一个新的订单的步骤如下：

1. 提供一个表单供用户填写
2. 根据用户填写的内容生成一个新 `Order` 类实例，然后将购物车中的商品放入 `OrderItem` 实例中并与 `Order` 实例建立外键关系
3. 清理全部购物车内容，然后重定向用户到一个操作成功页面。

首先利用内置表单功能建立订单表单，在 `orders` 应用中新建 `forms.py` 文件并添加如下代码：

```
from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address', 'postal_code', 'city']
```

采用内置的模型表单创建对应 `order` 对象的表单，现在要建立视图来控制表单，编辑 `orders` 应用中的 `views.py`：

```
from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart

def order_create(request):
    cart = Cart(request)
    if request.method == "POST":
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save()
            for item in cart:
```

```
        OrderItem.objects.create(order=order, product=item['product'], price=item['price'],
                                quantity=item['quantity'])
    # 成功生成OrderItem之后清除购物车
    cart.clear()
    return render(request, 'orders/order/created.html', {'order': order})

else:
    form = OrderCreateForm()
return render(request, 'orders/order/create.html', {'cart': cart, 'form': form})
```

在这个 `order_create` 视图中，我们首先通过 `cart = Cart(request)` 获取当前购物车对象；之后根据HTTP请求种类的不同，视图进行以下工作：

- GET请求：初始化空白的 `OrderCreateForm`，并且渲染 `orders/order/created.html` 页面。
- POST请求：通过POST请求中的数据生成表单并且验证，验证通过之后执行 `order = form.save()` 创建新订单对象并写入数据库；然后遍历购物车的所有商品，对每一种商品创建一个 `OrderItem` 对象并存入数据库。最后清空购物车，渲染 `orders/order/created.html` 页面。

在 `orders` 应用里建立 `urls.py` 作为二级路由：

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

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

配置好了 `order_create` 视图的路由，再配置 `myshop` 项目的根 `urls.py` 文件，在 `shop.urls` 之前增加下边这条：

```
path('orders/', include('orders.urls', namespace='orders')),
```

编辑购物车详情页 `cart/detail.html`，找到下边这行：

```
<a href="#" class="button">Checkout</a>
```

将这个结账按钮的链接修改为 `order_create` 视图的URL：

```
<a href="{% url 'orders:order_create' %}" class="button">Checkout</a>
```

用户现在可以通过购物车详情页来提交订单，我们要为订单页制作模板，在 `orders` 应用下建立如下文件和目录结构：

```
templates/
  orders/
    order/
      create.html
      created.html
```

编辑确认订单的页面 `orders/order/create.html`，添加如下代码：

```
{% extends 'shop/base.html' %}

{% block title %}
Checkout
{% endblock %}

{% block content %}
<h1>Checkout</h1>
```

```
<div class="order-info">
    <h3>Your order</h3>
    <ul>
        {% for item in cart %}
        <li>
            {{ item.quantity }} x {{ item.product.name }}
            <span>${{ item.total_price }}</span>
        </li>
        {% endfor %}
    </ul>
    <p>Total: ${{ cart.get_total_price }}</p>
</div>

<form action="." method="post" class="order-form" novalidate>
    {{ form.as_p }}
    <p><input type="submit" value="Place order"></p>
    {% csrf_token %}
</form>
{% endblock %}
```

这个模板，展示购物车内的商品和总价，之后提供空白表单用于提交订单。

再来编辑订单提交成功后跳转到的页面 `orders/order/created.html`：

```
{% extends 'shop/base.html' %}

{% block title %}
Thank you
{% endblock %}

{% block content %}
    <h1>Thank you</h1>
    <p>Your order has been successfully completed. Your order number is <strong>{{ order.id }}</strong>. </p>
{% endblock %}
```

这是订单成功页面。启动站点，添加一些商品到购物车中，然后在购物车详情页面中点击CHECKOUT按钮，之后可以看到如下页面：

The screenshot shows a checkout page for a shop. At the top left is the shop name "My shop". To the right, it displays "Your cart: 3 items, \$112.2". The main title "Checkout" is centered above a form with five input fields: "First name:", "Last name:", "Email:", "Address:", and "Postal code:". To the right, a summary box titled "Your order" lists the items: "1x Tea powder" and "2x Red tea", with a total of "\$112.2".

My shop

Your cart: 3 items, \$112.2

Checkout

First name:

Last name:

Email:

Address:

Postal code:

Your order

- 1x Tea powder \$21.2
- 2x Red tea \$91.0

Total: \$112.2

City:

Place order

填写表单然后点击Place order按钮，订单被创建，然后重定向至创建成功页面：

Thank you

Your order has been successfully completed. Your order number is 1.

现在可以到管理后台去看一看相关的信息了。

4 使用Celery启动异步任务

在一个视图内执行的所有操作，都会影响到响应时间。很多情况下，尤其视图中有一些非常耗时或者可能会失败，需要重试的操作，我们希望尽快给用户先返回一个响应而不是等到执行结束，而让服务器去继续异步执行这些任务。例如：很多视频分享网站允许用户上传视频，在上传成功之后服务器需花费一定时间转码，这个时候会先返回一个响应告知用户视频已经成功上传，正在进行转码，然后异步进行转码。还有一个例子是向用户发送邮件。如果站点中有一个视图的操作是发送邮件，SMTP连接很可能失败或者速度比较慢，这个时候采用异步的方式就能有效的避免阻塞。

Celery是一个分布式任务队列，采取异步的方式同时执行大量的操作，支持实施操作和计划任务，可以方便的批量创建异步任务并且执行，也可以设定为计划执行。Celery的文档在
<http://docs.celeryproject.org/en/latest/index.html>。

4.1 安裝Celery

通过 `pip` 安裝Celery：

```
pip install celery==4.1.0
```

Celery需要一个消息代理程序来处理外部的请求，这个代理把要处理的请求发送到Celery worker，也就是实际处理任务的模块。所以还需要安装一个消息代理程序：

4.2 安裝RabbitMQ

Celery的消息代理程序有很多选择，Redis数据库也可以作为Celery的消息代理程序。这里我们使用RabbitMQ，因为它是Celery官方推荐的消息代理程序。

如果是Linux系统，通过如下命令安装RabbitMQ：

```
apt-get install rabbitmq
```

如果使用macOS X或者Windows，可以在<https://www.rabbitmq.com/download.html> 下载RabbitMQ。

安装之后使用下列命令启动RabbitMQ服务：

```
rabbitmq-server
```

之后会看到：

```
Starting broker... completed with 10 plugins.
```

就说明RabbitMQ已经就绪，等待接受消息。

译者注：Windows下安装RabbitMQ，必须先安装[Erlang OPT平台](#)，然后安装从官网下载回来的RabbitMQ windows installer。之后需要手工把Erlang安装目录下的bin目录和RabbitMQ安装目录下的sbin目录设置到PATH中。之后安装参见[这里](#)。

4.3 在项目中集成Celery

需要为项目使用的Celery实例进行一些配置，在 `settings.py` 文件的相同目录下创建 `celery.py` 文件：

```
import os
from celery import Celery

# 为celery程序设置环境为当前项目的环境
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')

app = Celery('myshop')

app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

这段程序解释如下：

1. 导入 `DJANGO_SETTINGS_MODULE` 环境变量，为Celery命令行程序创造运行环境。
2. 实例化一个 `app` 对象，是一个Celery程序实例

3. 调用 `config_from_object()` 方法，从我们项目的设置文件中读取环境设置。`namespace` 属性指定了在我们的 `settings.py` 文件中，所有和 Celery 相关的配置都以 `CELERY` 开头，例如 `CELERY_BROKER_URL`。
4. 调用 `autodiscover_tasks()`，让 Celery 自动发现所有的异步任务。Celery 会在每个 `INSTALLED_APPS` 中列出的应用中寻找 `task.py` 文件，在里边寻找定义好的异步任务然后执行。

还需要在项目的 `__init__.py` 文件中导入 `celery` 模块，以让项目启动时 Celery 就运行，编辑 `myshop/__init__.py`：

```
# import celery
from .celery import app as celery_app
```

现在就可以为应用启动异步任务了。

`CELERY_ALWAYS_EAGER` 设置可以让 Celery 在本地以同步的方式直接执行任务，而不会去把任务加到队列中。这常用来进行测试或者检查 Celery 的配置是否正确。

4.4 为应用添加异步任务

我们准备在用户提交订单的时候异步发送邮件。一般的做法是在应用目录下建立一个 `task` 模块专门用于编写异步任务，在 `orders` 应用下建立 `task.py` 文件，添加如下代码：

```
from celery import task
from django.core.mail import send_mail
from .models import Order

@task
def order_created(order_id):
    """
    当一个订单创建完成后发送邮件通知给用户
    """
```

```
order = Order.objects.get(id=order_id)
subject = 'Order {}'.format(order.id)
message = 'Dear {},\n\nYou have successfully placed an order. Your order id is {}.'.format(order.first_name,
                                         order_id)
mail_sent = send_mail(subject, message, 'lee0709@vip.sina.com', [order.email])
print(mail_sent, type(mail_sent))
return mail_sent
```

将 `order_created` 函数通过装饰器 `@task` 定义为异步任务，可以看到，只要用 `@task` 装饰就可以把一个函数变成 Celery 异步任务。这里我们给异步函数传入 `order_id`，推荐仅传入 ID，让异步任务启动的时候再去检索数据库。最后拼接好标题和正文后使用 `send_mail()` 发送邮件。

在第二章已经学习过如何发送邮件，如果没有 SMTP 服务器，在 `settings.py` 里将邮件配置为打印到控制台上：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

在实际应用中，除了耗时比较大的功能之外，还可以将其他容易失败需要重试的功能，即使耗时较短，也推荐设置为异步任务。

设置好了异步任务之后，还需要修改原来的视图 `order_created`，以便在订单完成的时候，调用 `order_created` 异步函数。编辑 `orders` 应用的 `views.py` 文件：

```
from .task import order_created

def order_create(request):
    .....
    if request.method == "POST":
        .....
        if form.is_valid():
            .....
```

```
cart.clear()
# 启动异步任务
order_created.delay(order.id)
#.....
```

调用 `delay()` 方法即表示异步执行该任务，任务会被加入队列然后交给执行程序执行。

启动另外一个shell（必须是导入了当前环境的命令行窗口，比如Pycharm中启动的terminal），使用如下命令启动Celery worker：

```
celery -A myshop worker -l info
```

现在Celery worker已经启动并且准备处理任务。启动站点，然后添加一些商品到购物车，提交订单。在启动了Celery worker的窗口应该能看到类似下边的输出：

```
[2017-12-17 17:43:11,462: INFO/MainProcess] Received task:
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e]
[2017-12-17 17:43:11,685: INFO/ForkPoolWorker-4] Task
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e] succeeded in
0.22019841300789267s: 1
```

表示任务已经被执行，应该可以收到邮件了。

译者注：Windows平台下，在发送邮件的时候，有可能出现错误信息如下：

```
not enough values to unpack (expected 3, got 0)
```

这是因为Celery 4.x 在win10版本下运行存在问题，解决方案为：先安装Python的`eventlet`模块：

```
pip install eventlet
```

然后在启动Celery worker的时候，加上参数-P eventlet，命令行如下：

```
celery -A myshop worker -l info -P eventlet
```

即可解决该错误。在linux下应该不会发生该错误。参考Celery项目在Github上的问题：[Unable to run tasks under Windows #4081](#)

4.5 监控Celery

如果想要监控异步任务的执行情况，可以安装Python的Flower模块：

```
pip install flower==0.9.2
```

之后在新的终端窗口输入：

```
celery -A myshop flower
```

之后在浏览器中打开<http://localhost:5555/dashboard>，即可看到图形化监控的Celery情况：

Active: 0

Processed: 0

Failed: 0

Succeeded: 0

Retried: 0

Search:

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@MacBook-Air-de-Antonio.local	Online	0	0	0	0	0	2.2, 2.4, 2.36

Showing 1 to 1 of 1 entries

可以在 <https://flower.readthedocs.io/> 查看Flower的文档。

总结

这一章里创建了一个基础的电商网站。为网站创建了商品品类和详情展示，通过session创建了购物车应用。实现了一个自定义的上下文处理器用于将购物车对象附加到所有模板上，还实现了创建订单的功能。最后还学习了使用Celery启动异步任务。

在下一章将学习集成支付网关，为管理后台增加自定义操作，将数据导出为CSV格式，以及动态的生成PDF文件。

第八章 管理支付与订单

上一章制作了一个带有商品品类展示和购物车功能的电商网站雏形，同时也学到了如何使用Celery给项目增加异步任务。本章将学习为网站集成支付网关以让用户通过信用卡付款，还将为管理后台扩展两项功能：将数据导出为CSV以及生成PDF发票。

本章的主要内容有：

- 集成支付网关到项目中
- 将订单数据导出成CSV文件
- 为管理后台创建自定义视图
- 动态生成PDF发票

1 集成支付网关

支付网关是一种处理在线支付的网站或者程序，使用支付网关，就可以管理用户的订单，然后将支付过程交给一个可信赖且安全的第三方，而无需在我们自己的站点上处理支付信息。

支付网关有很多可供选择，我们将要集成的是叫做“Braintree”的支付网关。Braintree使用较为广泛，是Uber和Airbnb的支付服务提供商。Braintree提供了一套API用于支持信用卡，PayPal，Android Pay和Apple Pay等支付方式，官方网站在 <https://www.braintreepayments.com/>。

Braintree提供了很多集成的方法，最简单的集成方式就是Drop-in集成，包含一个预先建立好的支付表单。但是为了自定义一些支付过程中的内容，这里选择使用高级的**Hosted Field**（字段托管）方式进行集成。在 <https://developers.braintreepayments.com/guides/hosted-fields/overview/javascript/v3> 可以看到详细的帮助文档。

支付表单中包含的信用卡号，CVV码，过期日期等信息必须要得到安全处理，Hosted Field集成方式将这些字段展示给用户的时候，在页面中渲染的是一个iframe框架。我们可以来自定义该字段的外观，但必须要遵循 [Payment Card Industry \(PCI\) 安全支付](#) 的要求。由于可以修改外观，用户并不会注意到页面使用了iframe。

译者注：原书在这里说的不是很清晰。Hosted Fields的意思是敏感字段由我们页面中的Braintree JavaScript客户端通过Braintree服务器生成并填入到页面中，而不是在模板中直接编写 `input` 字段。简单的说就是信用卡等敏感信息的字段是由Braintree托管生成，而不是我们自行编写。

1.1 注册Braintree沙盒测试账户

需要注册一个Braintree账户。才能使用集成支付功能。我们先注册一个Braintree沙盒账户用于开发和测试。打开<https://www.braintreepayments.com/sandbox>，如下图所示：

Test Everything Braintree

Entering our sandbox allows you to get a feel for the Braintree experience before applying for a merchant account or going to production.

Already in the sandbox? [Sign in.](#)

Sign up for the sandbox

Full name

First name

Last name

Company name

Where is your business located?

Spain

Email address

me@example.com

Try the sandbox

填写表单创建用户，之后会收到电子邮件验证，验证通过之后在 <https://sandbox.braintreegateway.com/login> 进行登录。可以得到自己的商户ID和私有/公开密钥如下图所示：

Sandbox Keys & Configuration

Here are the keys to your Sandbox Account. Once you're ready to start taking payments with a production Braintree Account you'll have to update your code, replacing these with your production Braintree Account keys.

Merchant ID: 9xtfhm7sv733jznk

Public Key: q8fxx6fwkjx8dfkw

Private Key: XXXXXXXXXXXXXXXXXXXXXXXXX

这些信息与使用Braintree API进行验证交易有关，注意保存好私钥，不要泄露给他人。

1.2 安裝Braintree的Python模块

Braintree为Python提供了一个模块操作其API，源代码地址在 https://github.com/braintree/braintree_python。我们将把这个 `braintree` 模块集成到站点中。

使用命令行安装 `braintree` 模块：

```
pip install braintree==3.45.0
```

之后在 `settings.py` 里配置：

```
# Braintree支付网关设置
BRAINTREE_MERCHANT_ID = 'XXX' # 商户ID
BRAINTREE_PUBLIC_KEY = 'XXX' # 公钥
BRAINTREE_PRIVATE_KEY = 'XXX' # 私钥

from braintree import Configuration, Environment

Configuration.configure(
    Environment.Sandbox,
    BRAINTREE_MERCHANT_ID,
    BRAINTREE_PUBLIC_KEY,
    BRAINTREE_PRIVATE_KEY
)
```

将 `BRAINTREE_MERCHANT_ID` , `BRAINTREE_PUBLIC_KEY` , `BRAINTREE_PRIVATE_KEY` 的值替换成你自己的实际信息。

注意此处的设置 `Environment.Sandbox` , 表示我们当前集成的是沙盒环境。如果站点正式上线并且获取了正式的 Braintree 账户，必须修改成 `Environment.Production`。Braintree 对于正式账号会有新的商户ID和公钥/私钥。

Braintree的基础设置结束了，下一步是将支付网关和支付过程结合起来。

1.3 集成支付网关

结账过程是这样的：

1. 将商品加入到购物车
2. 从购物车中选择结账
3. 输入信用卡信息并且支付

针对支付功能，我们建立一个新的应用叫做 `payment` :

```
python manage.py startapp payment
```

编辑 `settings.py` 文件，激活该应用：

```
INSTALLED_APPS = [  
    # ...  
    'payment.apps.PaymentConfig',  
]
```

`payment` 现在已经被激活。

客户成功提交订单后，必须将该页面重定向到一个支付过程页面（目前是重定向到一个简单的成功页面）。编辑 `orders` 应用中的 `views.py`，增加如下导入：

```
from django.urls import reverse  
from django.shortcuts import render, redirect
```

在同一个文件内，将 `order_create` 视图的如下部分：

```
# 启动异步任务  
order_created.delay(order.id)  
return render(request, 'orders/order/created.html', locals())
```

替换成：

```
# 启动异步任务  
order_created.delay(order.id)  
# 在session中加入订单id
```

```
request.session['order_id'] = order.id  
# 重定向到支付页面  
return redirect(reverse('payment:process'))
```

这样修改后，在成功创建订单之后，session中就保存了订单ID的变量 `order_id`，然后用户被重定向至 `payment:process` URL，这个URL稍后会编写。

注意必须为 `order_created` 视图启动Celery。

每次我们向Braintree中发送一个交易请求的时候，会生成一个唯一的交易ID号。因此我们在 `Order` 模型中增加一个字段用于存储这个交易ID号，这样可以将订单与Braintree交易联系起来。

编辑 `orders` 应用的 `models.py` 文件，为 `Order` 模型新增一行：

```
class Order(models.Model):  
    # ...  
    braintree_id = models.CharField(max_length=150, blank=True)
```

之后执行数据迁移程序，每一个订单都会保存与其关联的交易ID。目前准备工作都已经做完，剩下就是在支付过程中使用支付网关。

1.3.1 使用Hosted Fields进行支付

Hosted Fields方式允许我们创建自定义的支付表单，使用自定义样式和表现形式。Braintree JavaScript SDK会在页面中动态的添加iframe框体用于展示Host Fields支付字段。当用户提交表单的时候，Hosted Fields会安全地提取用户的信用卡等信息，生成一个特征字符串（ tokenize，令牌化）。如果令牌化过程成功，就可以使用这个特征字符串（ token），通过视图中的 `braintree` 模块发起一个支付申请。

为此需要建立一个支付视图。这个视图的工作流程如下：

1. 用户提交订单时，视图通过 `braintree` 模块生成一个token，这个token用于Braintree JavaScript 客户端生成支付表单，并不是最终发送给支付网关的token。为了方便以下把这个token称为临时token，把最终提交给Braintree网站的token叫做交易token。
2. 视图渲染支付表单所在的模板。页面中的Braintree JavaScript 客户端使用临时token来生成页面中的支付表单。
3. 用户输入信用卡信息并且提交支付表单后，Braintree JavaScript 客户端会生成交易token，将这个交易token通过 `POST` 请求发送到视图
4. 视图获取交易token之后，通过 `braintree` 模块向网站提交交易请求。

了解了工作流程之后，来编写相关视图，编辑 `payment` 应用中的 `views.py` 文件，添加下列代码：

```
import braintree
from django.shortcuts import render, redirect, get_object_or_404
from orders.models import Order

def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)

    if request.method == "POST":
        # 获得交易token
        nonce = request.POST.get('payment_method_nonce', None)
        # 使用交易token和附加信息，创建并提交交易信息
        result = braintree.Transaction.sale(
            {
                'amount': '{:2f}'.format(order.get_total_cost()),
                'payment_method_nonce': nonce,
                'options': {
                    'submit_for_settlement': True,
                }
            }
        )
```

```
if result.is_success:
    # 标记订单状态为已支付
    order.paid = True
    # 保存交易ID
    order.braintree_id = result.transaction.id
    order.save()
    return redirect('payment:done')
else:
    return redirect('payment:canceled')

else:
    # 生成临时token交给页面上的JS程序
    client_token = braintree.ClientToken.generate()
    return render(request,
                  'payment/process.html',
                  {'order': order,
                   'client_token': client_token})
```

这个 `payment_process` 视图管理支付过程，工作流程如下：

1. 从 `session` 中取出由 `order_create` 视图设置的 `order_id` 变量。
2. 获取 `Order` 对象，如果没找到，返回 `404 Not Found` 错误
3. 如果接收到 `POST` 请求，获取交易 token `payment_method_nonce`，使用交易 token 和 `braintree.Transaction.sale()` 方法生成新的交易，该方法的几个参数解释如下：
 1. `amount`：总收款金额
 2. `payment_method_nonce`：交易 token，由页面中的 Braintree JavaScript 客户端生成。
 3. `options`：其他选项，`submit_for_settlement` 设置为 `True` 表示生成交易信息完毕的时候就立刻提交。
4. 如果交易成功，通过设置 `paid` 属性为 `True`，将订单标记为已支付，将交易 ID 存储到 `braintree_id` 属性中，之后重定向至 `payment:done`，如果交易失败就重定向至 `payment:canceled`。
5. 如果视图接收到 `GET` 请求，生成临时 token 交给页面中的 Braintree JavaScript 客户端。

下边建立支付成功和失败时的处理视图，在 payment 应用的views.py中添加下列代码：

```
def payment_done(request):
    return render(request, 'payment/done.html')
def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

然后在 payment 目录下建立 urls.py，为上述视图配置路由：

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('done/', views.payment_done, name='done'),
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

这是支付流程的路由，配置了如下URL模式：

- `process`：处理支付的视图
- `done`：支付成功的视图
- `canceled`：支付未成功的视图

编辑 myshop 项目的根 urls.py 文件，为 payment 应用配置二级路由：

```
urlpatterns = [
    # ...
```

```
    path('payment/', include('payment.urls', namespace='payment')),
    path('', include('shop.urls', namespace='shop')),
]
```

依然要注意这一行要放到 `shop.urls` 上边，否则无法被解析到。

之后是建立视图，在`payment`目录下建立`templates/payment/`目录，并在其中建立 `process.html`, `done.html`, `canceled.html`三个模板。先来编写`process.html`:

在 `payment` 应用内建立下列目录和文件结构:

```
templates/
  payment/
    process.html
    done.html
    canceled.html
```

编辑 `payment/process.html`，添加下列代码:

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

{% block title %}Pay by credit card{% endblock %}

{% block content %}
  <h1>Pay by credit card</h1>
  <form action"." id="payment" method="post">

    <label for="card-number">Card Number</label>
    <div id="card-number" class="field"></div>

    <label for="cvv">CVV</label>
    <div id="cvv" class="field"></div>
```

```
<label for="expiration-date">Expiration Date</label>
<div id="expiration-date" class="field"></div>

<input type="hidden" id="nonce" name="payment_method_nonce" value="">
{%
  csrf_token
}
<input type="submit" value="Pay">
</form>
<!-- Load the required client component. --&gt;
&lt;script src="https://js.braintreegateway.com/web/3.29.0/js/client.min.js"&gt;&lt;/script&gt;
<!-- Load Hosted Fields component. --&gt;
&lt;script src="https://js.braintreegateway.com/web/3.29.0/js/hosted-fields.min.js"&gt;&lt;/script&gt;
&lt;script&gt;
  var form = document.querySelector('#payment');
  var submit = document.querySelector('input[type="submit"]');

  braintree.client.create({
    authorization: '{{ client_token }}'
  }, function (clientErr, clientInstance) {
    if (clientErr) {
      console.error(clientErr);
      return;
    }

    braintree.hostedFields.create({
      client: clientInstance,
      styles: {
        'input': {'font-size': '13px'},
        'input.invalid': {'color': 'red'},
        'input.valid': {'color': 'green'}
      },
      fields: {
        number: {selector: '#card-number'},
        cvv: {selector: '#cvv'},
        expirationDate: {selector: '#expiration-date'}
      }
    });
  });
&lt;/script&gt;</pre>
```

```
        }
    }, function (hostedFieldsErr, hostedFieldsInstance) {
        if (hostedFieldsErr) {
            console.error(hostedFieldsErr);
            return;
        }

        submit.removeAttribute('disabled');

        form.addEventListener('submit', function (event) {
            event.preventDefault();

            hostedFieldsInstance.tokenize(function (tokenizeErr, payload) {
                if (tokenizeErr) {
                    console.error(tokenizeErr);
                    return;
                }
                // set nonce to send to the server
                document.getElementById('nonce').value = payload.nonce;
                // submit form
                document.getElementById('payment').submit();
            });
        }, false);
    });
});
</script>
{% endblock %}
```

这是用户填写信用卡信息并且提交支付的模板，我们用 `<div>` 替代 `<input>` 使用在信用卡号，CVV码和过期日期字段上。这些字段就是Braintree JavaScript客户端渲染的iframe字段。还使用了一个名称为 `payment_method_nonce` 的 `<input>` 元素用于提交交易ID到后端。

在模板中还导入了Braintree JavaScript SDK的 `client.min.js` 和Hosted Fields组件 `hosted-fields.min.js`，然后执行了下列JS代码：

1. 使用 `braintree.client.create()` 方法，传入 `client_token` 即 `payment_process` 视图里生成的临时token，实例化Braintree JavaScript 客户端。
2. 使用 `braintree.hostedFields.create()` 实例化Hosted Field组件
3. 给 `input` 字段应用自定义样式
4. 给 `cardnumber`，`cvv`，和 `expiration-date` 字段设置 `id` 选择器
5. 给表单的 `submit` 行为绑定一个事件，当表单被点击提交时，Braintree SDK 使用表单中的信息，生成交易token放入 `payment_method_nonce` 字段中，然后提交表单。

编辑 `payment/done.html` 文件，添加下列代码：

```
{% extends "shop/base.html" %}  
{% block content %}  
    <h1>Your payment was successful</h1>  
    <p>Your payment has been processed successfully.</p>  
{% endblock %}
```

这是订单成功支付时用户被重定向的页面。

编辑 `canceled.html`，添加下列代码：

```
{% extends "shop/base.html" %}  
{% block content %}  
    <h1>Your payment has not been processed</h1>  
    <p>There was a problem processing your payment.</p>  
{% endblock %}
```

这是订单未支付成功时用户被重定向的页面。之后我们来试验一下付款。

1.4 测试支付

打开系统命令行窗口然后运行RabbitMQ:

```
rabbitmq-server
```

再启动一个命令行窗口，启动Celery worker:

```
celery -A myshop worker -l info
```

再启动一个命令行窗口，启动站点:

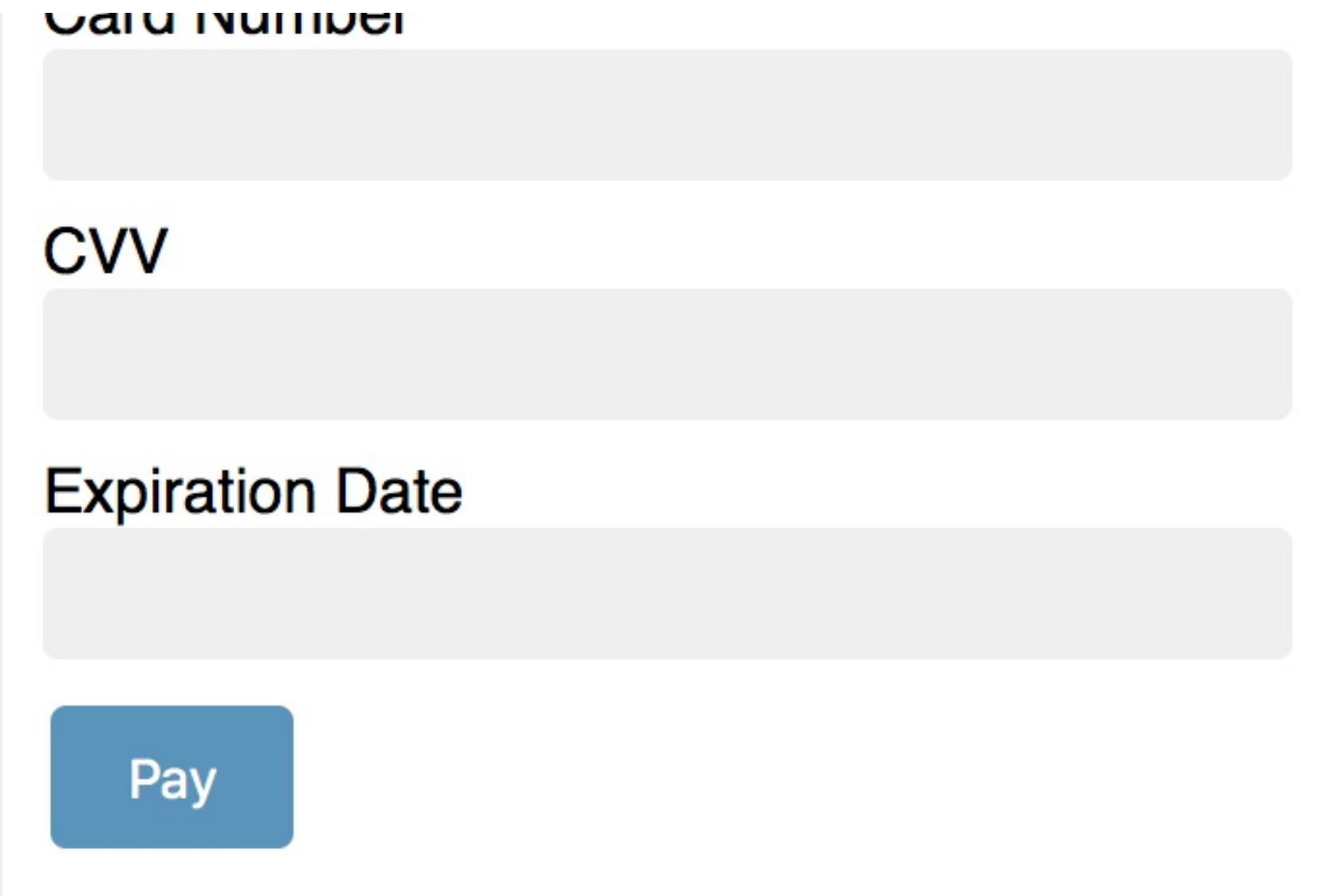
```
python manage.py runserver
```

之后在浏览器中打开 <http://127.0.0.1:8000/> 加入一些商品到购物车，提交订单，当按下PLACE ORDER按钮后，订单信息被保存进数据库，订单ID被附加到session上，然后进入支付页面。

支付页面从session中取得订单id然后在iframe中渲染Hosted Fields，像下图所示：



The screenshot shows a large title "Pay by credit card" at the top. Below it, there is a form field labeled "Card Number".



可以看一下页面的HTML代码，从而理解什么是Hosted Fields。

针对沙盒测试环境，Braintree提供了一些测试用的信用卡资料，可以进行付款成功或失败的测试，可以在
<https://developers.braintreepayments.com/guides/credit-cards/testing-go-live/python> 找到，我们来使用

4111 1111 1111 1111 这个信用卡号，在CVV码中填入 123，到期日期填入未来的某一天比如 12/20：

Pay by credit card

Card Number

4111 1111 1111 1111

CVV

123

Expiration Date

12 / 20



Pay

之后点击Pay，应该可以看到成功页面：

My shop

Your payment was successful

Your payment has been processed successfully.

说明付款已经成功。可以在<https://sandbox.braintreegateway.com/login>登录，然后在左侧菜单选Transaction里搜索最近的交易，可以看到如下信息：

ID	Transaction Date	Type	Status	Customer	Payment Information	Amount
2bwkx5b6	02/05/2018 07:45:23 PM CST	Sale	Submitted For Settlement		411111*****1111	21,20 € EUR

译者注：Braintree网站在成书后有部分改版，读者看到的支付详情页面可能与上述图片有一些区别。

然后再查看管理站点 <http://127.0.0.1:8000/admin/orders/order/> 中的对应记录，该订单应该已经被标记为已支付，而且记录了交易ID，如下图所示：



我们现在就成功集成了支付功能。

1.5 正式上线

在沙盒环境中测试通过之后，需要正式上线的话，需要到 <https://www.braintreepayments.com> 创建正式账户。

在部署到生产环境时，需要将 `settings.py` 中的商户ID和公钥私钥更新为正式账户的对应信息，然后将其中的 `Environment.Sandbox` 修改为 `Environment.Production`。正式上线的具体步骤可以参考：
<https://developers.braintreepayments.com/start/go-live/python>。

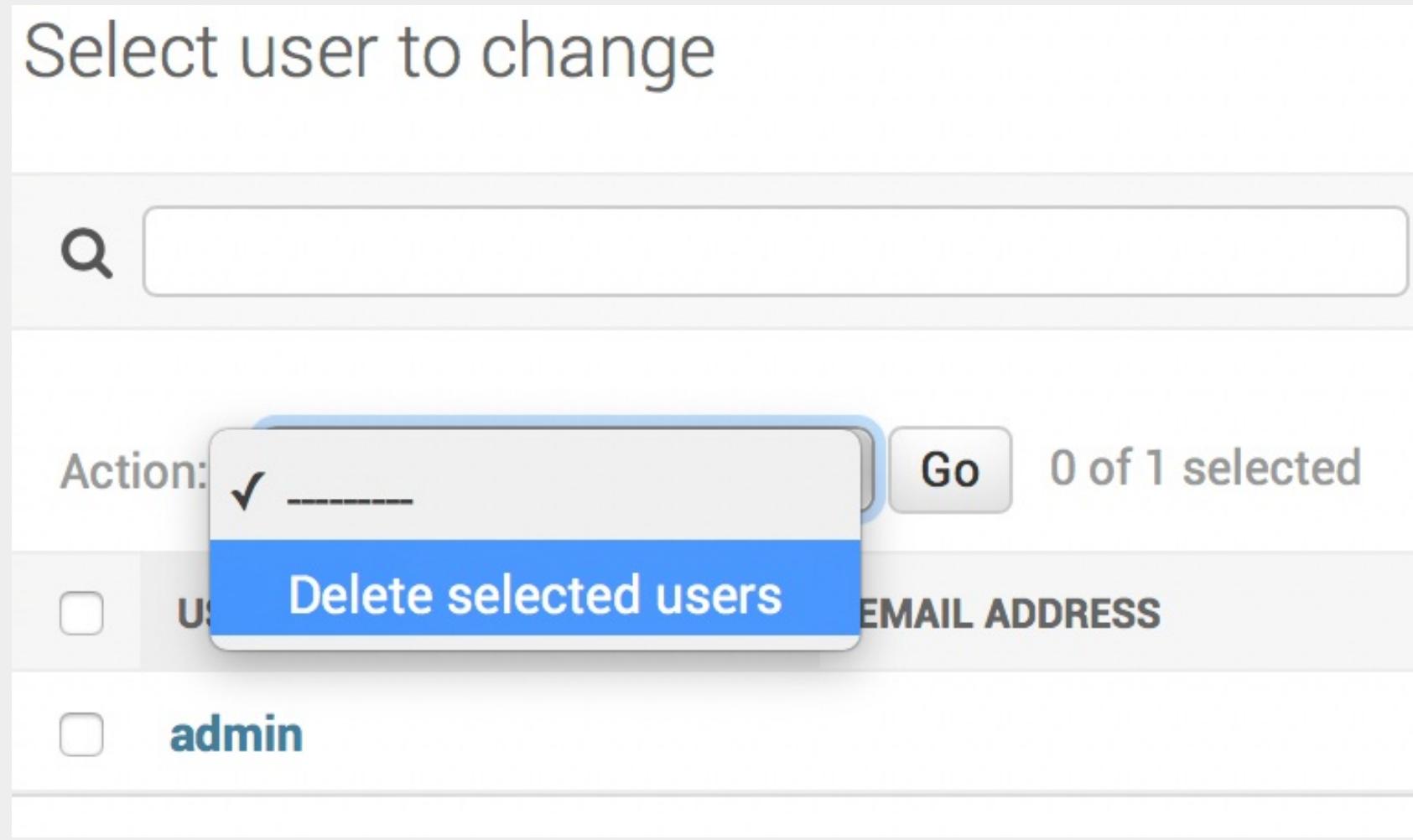
2 导出订单为CSV文件

有时我们想将某个模型中的数据导出为一个文件，用于在其他系统导入。常用的一种数据交换格式是CSV（逗号分隔数据）文件。CSV文件是一个纯文本文件，包含很多条记录。通常一行是一条记录，用特定的分隔符（一般是逗号）分隔每个字段的值。我们准备自定义管理后台，增加导出CSV文件的功能。

2.1 给管理后台增加自定义管理行为 (actions)

Django允许对管理后台的很多内容进行自定义修改。我们准备在列出具体数据的视图内增加导出CSV文件的功能。

一个管理行为是指如下操作：用户从管理后台列出某个模型中具体记录的页面内，使用复选框选中要操作的记录，然后从下拉菜单中选择一项操作，之后就会针对所有选中的记录执行操作。这个action的位置如下图所示：



创建自定义管理行为可以让管理员批量对记录进行操作。

可以通过写一个符合要求的自定义函数作为一项管理行为，这个函数要接受如下参数：

- 当前显示的 `ModelAdmin` 类
- 当前的 `request` 对象，是一个 `HttpResponse` 实例
- 用户选中的内容组成的 `QuerySet`

在选中一个 `action` 选项然后点击旁边的 Go 按钮的时候，该函数就会被执行。

我们就准备在下拉 `action` 清单里增加一项导出 CSV 数据的功能，为此先来修改 `orders` 应用中的 `admin.py`，将下列代码加在 `OrderAdmin` 类定义之前：

```
import csv
import datetime
from django.http import HttpResponseRedirect

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    response = HttpResponseRedirect(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename={}.csv'.format(opts.verbose_name)
    writer = csv.writer(response)

    fields = [field for field in opts.get_fields() if not field.many_to_many and not field.one_to_many]
    writer.writerow([field.verbose_name for field in fields])

    for obj in queryset:
        data_row = []
        for field in fields:
            value = getattr(obj, field.name)
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)
        writer.writerow(data_row)
```

```
        data_row.append(value)
    writer.writerow(data_row)
return response

export_to_csv.short_description = 'Export to CSV'
```

在这个函数里我们做了如下事情：

1. 创建一个 `HttpResponse` 对象，将其内容类型设置为 `text/csv`，以告诉浏览器将其视为一个CSV文件。还为请求头附加了 `Content-Disposition` 头部信息，告诉浏览器这个请求带有一个附加文件。
2. 创建一个CSV的 `writer` 对象，用于向Http响应对象 `response` 中写入CSV文件数据。
3. 通过 `_meta` 的 `get_fields()` 方法获取所有字段名，动态获取 `model` 的字段，排除了所有一对多和多对多字段。
4. 将字段名写入到响应的CSV数据中，作为第一行数据，即表头
5. 迭代QuerySet，将其中每一个对象的数据写入一行中，注意特别对 `datetime` 采用了格式化功能，以转换成字符串。
6. 最后设置了该函数对象的 `short_description` 属性，该属性的值为在action列表中显示的功能名称。

这样我们就创建了一个通用的管理功能，可以操作任何 `ModelAdmin` 对象。

之后在 `OrderAdmin` 类中增加这个新的 `export_to_csv` 功能：

```
class OrderAdmin(admin.ModelAdmin):
    WeasyPrint    # ...
    actions = [export_to_csv]
```

在浏览器中打开 <http://127.0.0.1:8000/admin/orders/order/> 查看订单类，页面如下：

Select order to change

Action: **Export to CSV** Go 1 of 19 selected

<input type="checkbox"/>	ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS
<input checked="" type="checkbox"/>	19	Antonio	Melé	antonio.mele@gmail.com	Bank Street
<input type="checkbox"/>	18	Django	Reinhardt	email@domain.com	Music Street

选择一些订单，然后选择上边的Export to CSV功能，然后点击Go按钮，浏览器就会下载一个 `order.csv` 文件。

译者注：此处下载的文件名可能不是 `order.csv`，这是因为原书没有在 `orders` 应用的 `models.py` 中为 `Order` 类的 `meta` 类增加 `verbose_name` 属性，手工增加 `verbose_name` 的值为 `order`，这样才能下载到和原书里写的名称一样的 `order.csv` 文件。

使用文本编辑器打开刚下载的CSV文件，可以看到里边的内容类似：

```
ID,first name,last name,email,address,postal  
code,city,created,updated,paid,braintree id  
3,Antonio,Melé,antonio.mele@gmail.com,Bank Street,WS  
J11,London,25/02/2018,25/02/2018,True,2bwkx5b6
```

可以看到，实现管理功能的方法很直接。Django中将数据输出为CSV的说明可以参考
<https://docs.djangoproject.com/en/2.0/howto/outputting-csv/>。

3 用自定义视图扩展管理后台的功能

不仅仅是配置 `ModelAdmin`，创建管理行为和覆盖内置模板，有时候可能需要对管理后台进行更多的自定义。这时你需要创建自定义的管理视图。使用管理视图，就可以实现自己想要的功能，要注意的只是自定义管理视图应该只允许管理员进行操作，同时继承内置模板以保持风格一致性。

我们这次来修改一下管理后台，增加一个自定义的功能用于显示一个订单的信息。修改 `orders` 应用中的 `views.py` 文件，增加如下内容：

```
from django.contrib.admin.views.decorators import staff_member_required
from django.shortcuts import get_object_or_404
from .models import Order

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request, 'admin/orders/order/detail.html', {'order': order})
```

`@staff_member_required` 装饰器只允许 `is_staff` 和 `is_active` 字段同时为 `True` 的用户才能使用被装饰的视图。在这个视图中，通过传入的 `id` 取得对应的 `Order` 对象。

然后配置 `orders` 应用的 `urls.py` 文件，增加一条路由：

```
path('admin/order/<int:order_id>', views.admin_order_detail, name='admin_order_detail')
```

然后在order应用的templates/目录下创建如下文件目录结构：

```
admin/
  orders/
    order/
      detail.html
```

编辑这个 `detail.html`，添加下列代码：

```
{% extends "admin/base_site.html" %}
{% load static %}
{% block extrastyle %}
    <link rel="stylesheet" type="text/css" href="{% static "css/admin.css" %}" />
{% endblock %}
{% block title %}
    Order {{ order.id }} {{ block.super }}
{% endblock %}
{% block breadcrumbs %}
    <div class="breadcrumbs">
        <a href="{% url "admin:index" %}">Home</a> ›
        <a href="{% url "admin:orders_order_changelist" %}">Orders</a>
        ›
        <a href="{% url "admin:orders_order_change" order.id %}">Order {{ order.id }}</a>
        › Detail
    </div>
{% endblock %}
{% block content %}
    <h1>Order {{ order.id }}</h1>
    <ul class="object-tools">
        <li>
            <a href="#" onclick="window.print();">Print order</a>
        </li>
    </ul>
</div>
```

```
<table>
  <tr>
    <th>Created</th>
    <td>{{ order.created }}</td>
  </tr>
  <tr>
    <th>Customer</th>
    <td>{{ order.first_name }} {{ order.last_name }}</td>
  </tr>
  <tr>
    <th>E-mail</th>
    <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
  </tr>
  <tr>
    <th>Address</th>
    <td>{{ order.address }}, {{ order.postal_code }} {{ order.city }}</td>
  </tr>
  <tr>
    <th>Total amount</th>
    <td>${{ order.get_total_cost }}</td>
  </tr>
  <tr>
    <th>Status</th>
    <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
  </tr>
</table>
<div class="module">
  <div class="tabular inline-related last-related">
    <table>
      <caption>Items bought</caption>
      <thead>
        <tr>
          <th>Product</th>
          <th>Price</th>
          <th>Quantity</th>
```

```
        <th>Total</th>
    </tr>
</thead>
<tbody>
{% for item in order.items.all %}
    <tr class="row{{ cycle "1" "2" }}">
        <td>{{ item.product.name }}</td>
        <td class="num">${{ item.price }}</td>
        <td class="num">{{ item.quantity }}</td>
        <td class="num">${{ item.get_cost }}</td>
    </tr>
{% endfor %}
<tr class="total">
    <td colspan="3">Total</td>
    <td class="num">${{ order.get_total_cost }}</td>
</tr>
</tbody>
</table>
</div>
</div>
{% endblock %}
```

这个模板用于在管理后台显示订单详情。模板继承了 `admin/base_site.html` 母版，这个母版包含Django管理站点的基础结构和CSS类，然后还加载了自定义的样式表 `css/admin.css`。

自定义CSS样式表在随书代码中，像之前的项目一样将其复制到对应目录。

我们使用的块名称都定义在母版中，在其中编写了展示订单详情的部分。

当你需要继承Django的内置模板时，必须了解内置模板的结构，在

<https://github.com/django/django/tree/2.1/django/contrib/admin/templates/admin> 可以找到内置模板的信息。

如果需要覆盖内置模板，需要将自己编写的模板命名为与原来模板相同，然后复制到 `templates` 下，设置与内置模板相同的相对路径和名称。管理后台就会优先使用当前项目下的模板。

最后，还需要再管理后台中为每个 `Order` 对象增加一个链接到我们自行编写的视图，编辑 `orders` 应用的 `admin.py` 文件，在 `OrderAdmin` 类之前增加如下代码：

```
from django.urls import reverse
from django.utils.safestring import mark_safe

def order_detail(obj):
    return mark_safe('<a href="{}">View</a>'.format(reverse('orders:admin_order_detail', args=[obj.id])))
```

这个函数接受一个 `Order` 对象作为参数，返回一个解析后的 `admin_order_detail` 名称对应的 URL，由于 Django 默认会将 HTML 代码转义，所以加上 `mark_safe`。

使用 `mark_safe` 可以不让 HTML 代码转义。使用 `mark_safe` 的时候，确保对于用户的输入依然要进行转义，以防止跨站脚本攻击。

然后编辑 `OrderAdmin` 类来显示链接：

```
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email', 'address', 'postal_code', 'city', 'paid', 'created',
                   'updated', order_detail]
```

然后启动站点，访问 <http://127.0.0.1:8000/admin/orders/order/>，可以看到新增了一列：

PAID	CREATED	▼	UPDATED	ORDER DETAIL
	Feb. 6, 2018, 1:35 a.m.		Feb. 6, 2018, 1:45 a.m.	View

点击任意一个订单的View链接查看详情，会进入Django管理后台风格的订单详情页：

Django administration

Home > Orders > Order 19 > Detail

Order 19

[PRINT ORDER](#)

Created Feb. 6, 2018, 1:35 a.m.

Customer Antonio Melé

E-mail antonio.mele@gmail.com

Address Jazz Street, 28027 Madrid

Total amount \$21.2

Status Paid

Items bought

PRODUCT	PRICE	QUANTITY	TOTAL
Tea powder	\$21.2	1	\$21.2
Total			\$21.2

4 动态生成PDF发票

我们现在已经实现了完整的结账和支付功能，可以为每个订单生成一个PDF发票。有很多Python库都可以用来生成PDF，常用的是Reportlab库，该库也是django 2.0官方推荐使用的库，可以在<https://docs.djangoproject.com/en/2.0/howto/outputting-pdf/> 查看详情。

大部分情况下，PDF文件中都要包含一些样式和格式，这个时候通过渲染后的HTML模板生成PDF更加方便。我们在Django中集成一个模块来按照上述方法转换PDF。这里我们使用WeasyPrint库，这个库用来从HTML模板生成PDF文件。

4.1 安装WeasyPrint

需要先按照<http://weasyprint.org/docs/install/#platforms> 的指引安装相关依赖，之后通过 pip 安装WeasyPrint：

```
pip install WeasyPrint==0.42.3
```

译者注：WeasyPrint在Windows下的配置非常麻烦，还未必能够成功，推荐Windows用户使用Linux虚拟机。

4.2 创建PDF模板

需要创建一个模板作为输入给WeasyPrint的数据。我们创建一个带有订单内容和CSS样式的模板，通过Django渲染，将最终生成的页面传给WeasyPrint。

在 orders 应用的 `templates/orders/order/` 目录下创建 `pdf.html` 文件，添加下列代码：

```
<html>
<body>
<h1>My Shop</h1>
<p>
    Invoice no. {{ order.id }}<br>
    <span class="secondary">
```

```
  {{ order.created|date:"M d, Y" }}  
  </span>  
  </p>  
  <h3>Bill to</h3>  
  <p>  
    {{ order.first_name }} {{ order.last_name }}<br>  
    {{ order.email }}<br>  
    {{ order.address }}<br>  
    {{ order.postal_code }}, {{ order.city }}  
  </p>  
  <h3>Items bought</h3>  
  <table>  
    <thead>  
      <tr>  
        <th>Product</th>  
        <th>Price</th>  
        <th>Quantity</th>  
        <th>Cost</th>  
      </tr>  
    </thead>  
    <tbody>  
      {% for item in order.items.all %}  
        <tr class="row{{ cycle "1" "2" }}">  
          <td>{{ item.product.name }}</td>  
          <td class="num">${{ item.price }}</td>  
          <td class="num">{{ item.quantity }}</td>  
          <td class="num">${{ item.get_cost }}</td>  
        </tr>  
      {% endfor %}  
      <tr class="total">  
        <td colspan="3">Total</td>  
        <td class="num">${{ order.get_total_cost }}</td>  
      </tr>  
    </tbody>  
  </table>
```

```
<span class="{% if order.paid %}paid{% else %}pending{% endif %}">  
  {% if order.paid %}Paid{% else %}Pending payment{% endif %}  
</span>  
</body>  
</html>
```

译者注：注意第五行标红的部分，原书错误的写成了`
`。

这个模板的内容很简单，使用一个`<table>`元素展示订单的用户信息和商品信息，还添加了消息显示订单是否已支付。

4.3 创建渲染PDF的视图

我们来创建在管理后台内生成订单PDF文件的视图，在`orders`应用的`views.py`文件内增加下列代码：

```
from django.conf import settings  
from django.http import HttpResponse  
from django.template.loader import render_to_string  
import weasyprint  
  
@staff_member_required  
def admin_order_pdf(request, order_id):  
    order = get_object_or_404(Order, id=order_id)  
    html = render_to_string('orders/order/pdf.html', {'order': order})  
    response = HttpResponse(content_type='application/pdf')  
    response['Content-Disposition'] = 'filename="order_{}.pdf"'.format(order.id)  
    weasyprint.HTML(string=html).write_pdf(response, stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')])  
    return response
```

这是生成PDF文件的视图，使用了`@staff_member_required`装饰器使该视图只能由管理员访问。通过ID获取Order对象，然后使用`render_to_string()`方法渲染`orders/order/pdf.html`，渲染后的模板以字符串形式保存在`html`变量

中。然后创建一个新的 `HttpResponse` 对象，并为其附加 `application/pdf` 和 `Content-Disposition` 请求头信息。使用 WeasyPrint 从字符串形式的 HTML 中转换 PDF 文件并写入 `HttpResponse` 对象。这个生成的 PDF 会带有 `STATIC_ROOT` 路径下的 `css/pdf.css` 中的样式，最后返回响应。

如果发现文件缺少 CSS 样式，记得把 CSS 文件从随书目录中放入 `shop` 应用的 `static/` 目录下。

我们这里还没有配置 `STATIC_ROOT` 变量，这个变量规定了项目的静态文件存放的路径。编辑 `myshop` 项目的 `settings.py` 文件，添加下面这行：

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

然后运行如下命令：

```
python manage.py collectstatic
```

之后会看到：

```
120 static files copied to 'code/myshop/static'.
```

这个命令会把所有已经激活的应用下的 `static/` 目录中的文件，复制到 `STATIC_ROOT` 指定的目录中。每个应用都可以有自己的 `static/` 目录存放静态文件，还可以在 `STATICFILES_DIRS` 中指定其他的静态文件路径，当执行 `collectstatic` 时，会把所有 `STATICFILES_DIRS` 目录内的文件都复制过来。如果再次执行 `collectstatic`，会提示是否需要覆盖已经存在的静态文件。

译者注：虽然将静态文件分开存放在每个应用的 `static/` 下可以正常运行开发中的站点，但在正式上线的最好统一静态文件的存放地址，以方便配置 Web 服务程序。

编辑 `orders` 应用的 `urls.py` 文件，为视图配置URL：

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/' ,views.admin_order_pdf ,name='admin_order_pdf') ,
]
```

像导出CSV一样，我们要在管理后台的展示页面中增加一个链接到这个视图的URL。打开 `orders` 应用的 `admin.py` 文件，在 `OrderAdmin` 类之前增加：

```
def order_pdf(obj):
    return mark_safe('<a href="{% url "orders:admin_order_pdf" %}">View</a>'.format(reverse('orders:admin_order_pdf', args=[obj.id])))
order_pdf.short_description = 'Invoice'
```

如果指定了 `short_description` 属性，Django就会用该属性的值作为列名。

为 `OrderAdmin` 的 `list_display` 增加这个新的字段 `order_pdf`：

```
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email', 'address', 'postal_code', 'city', 'paid', 'created',
                    'updated', order_detail, order_pdf]
```

在浏览器中打开 <http://127.0.0.1:8000/admin/orders/order/>，可以看到新增了一列字段用于转换PDF：

UPDATED

ORDER DETAIL

INVOICE

Feb. 11, 2018, 3:17 p.m.

[View](#)

[PDF](#)

点击PDF链接，浏览器应该会下载一个PDF文件，如果是尚未支付的订单，样式如下：

My Shop

Invoice no. 16

Feb 01, 2018

Bill to

Antonio Mele

antonio.mele@gmail.com

Jazz Street

28033, Madrid

Items bought

Product	Price	Quantity	Cost
Green tea	\$30	1	\$30
Total	\$30		



PENDING PAYMENT

已经支付的订单，则类似这样：

My Shop

Invoice no. 19

Feb 06, 2018

Bill to

Antonio Melé

antonio.melo@gmail.com

antonio.mete@gmail.com

Jazz Street
28027, Madrid

Items bought

Product	Price	Quantity	Cost
Tea powder	\$21.2	1	\$21.2
Total	\$21.2		



4.4 使用电子邮件发送PDF文件

当支付成功的时，我们发送带有PDF发票的邮件给用户。编辑 payment 应用中的 `views.py` 视图，添加如下导入语句：

```
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
from django.conf import settings
```

```
import weasyprint
from io import BytesIO
```

在 `payment_process` 视图中，`order.save()` 这行之后，以相同的缩进添加下列代码：

```
def payment_process(request):
    # .....
    if request.method == "POST":
        # .....
        if result.is_success:
            # .....
            order.save()

            # 创建带有PDF发票的邮件
            subject = 'My Shop - Invoice no. {}'.format(order.id)
            message = 'Please, find attached the invoice for your recent purchase.'
            email = EmailMessage(subject, message, 'admin@myshop.com', [order.email])

            # 生成PDF文件
            html = render_to_string('orders/order/pdf.html', {'order': order})
            out = BytesIO()
            stylesheets = [weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
            weasyprint.HTML(string=html).write_pdf(out, stylesheets)

            # 附加PDF文件作为邮件附件
            email.attach('order_{}.pdf'.format(order.id), out.getvalue(), 'application/pdf')

            # 发送邮件
            email.send()

            return redirect('payment:done')
    else:
        return redirect('payment:canceled')
```

```
else:  
    # .....
```

这里使用了 `EmailMessage` 类创建一个邮件对象 `email`，然后将模板渲染到 `html` 变量中，然后通过 `WeasyPrint` 将其写入一个 `BytesIO` 二进制字节对象，之后使用 `attach` 方法，将这个字节对象的内容设置为 `EmailMessage` 的附件，同时设置文件类型为 PDF，最后发送邮件。

记得在 `settings.py` 中设置 SMTP 服务器，可以参考第二章。

现在，可以尝试完成一个新的付款，并且在邮箱内接收 PDF 发票。

总结

在这一章中，我们集成了支付网关，自定义了 Django 管理后台，还学习了如何将数据以 CSV 文件格式导出和动态生成 PDF 文件。

下一章我们将深入了解 Django 项目的国际化和本地化设置，还将创建一个优惠码功能和商品推荐引擎。

第九章 扩展商店功能

在上一章里，为电商站点集成了支付功能，然后可以生成PDF发票发送给用户。在本章，我们将为商店添加优惠码功能。此外，还会学习国际化和本地化的设置和建立一个推荐商品的系统。

本章涵盖如下要点：

- 建立一个优惠券系统，可以实现折扣功能
- 给项目增加国际化功能
- 使用Rosetta来管理翻译
- 使用Django-parler翻译模型
- 建立商品推荐系统

1 优惠码系统

很多电商网站，会向用户发送电子优惠码，以便用户在购买时使用，以折扣价进行结算。一个在线优惠码通常是一个字符串，然后还规定了有效期限，一次性有效或者可以反复使用。

我们将为站点添加优惠码功能。我们的优惠码带有有效期，但是不限制使用次数，输入之后，就会影响用户购物车中的总价。为了实现这个需求，需要建立一个数据模型来存储优惠码，有效期和对应的折扣比例。

为 `myshop` 项目创建新的应用 `coupons`：

```
python manage.py startapp coupons
```

然后在 `settings.py` 内激活该应用：

```
INSTALLED_APPS = [
    # ...
    'coupons.apps.CouponsConfig',
]
```

1.1 创建优惠码数据模型

编辑 `coupons` 应用的 `models.py` 文件，创建一个 `Coupon` 模型：

```
from django.db import models
from django.core.validators import MinValueValidator, MaxValueValidator

class Coupon(models.Model):
    code = models.CharField(max_length=50, unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(validators=[MinValueValidator(0), MaxValueValidator(100)])
    active = models.BooleanField()

    def __str__(self):
        return self.code
```

这是用来存储优惠码的模型，`Coupon` 模型包含以下字段：

- `code`：用于存放码的字符串
- `valid_from`：优惠码有效期的开始时间。
- `valid_to`：优惠码有效期的结束时间。
- `discount`：该券对应的折扣，是一个百分比，所以取值为 `0-100`，我们使用了内置验证器控制该字段的取值范围。
- `active`：表示该码是否有效

之后执行数据迁移程序。然后将 `Coupon` 模型加入到管理后台，编辑 `coupons` 应用的 `admin.py` 文件：

```
from django.contrib import admin
from .models import Coupon

class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to', 'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
    search_fields = ['code']

admin.site.register(Coupon, CouponAdmin)
```

现在启动站点，到 <http://127.0.0.1:8000/admin/coupons/coupon/add/> 查看 `Coupon` 模型：

Add coupon

Code:

Valid from:

Date: Today |

Time: Now |

Note: You are 1 hour ahead of server time.

Valid to:

Date: Today |

Time: Now |

Note: You are 1 hour ahead of server time.

Discount:

Active

[Save and add another](#)

[Save and continue editing](#)

SAVE

输入一个优惠码记录，有效期设置为当前日期，不要忘记勾上Active然后点击SAVE按钮。

1.2 为购物车增加优惠码功能

创建数据模型之后，可以查询和获得优惠码对象。现在我们必须增添使用户可以输入优惠码从而获得折扣价的功能。这个功能将按照如下逻辑进行操作：

1. 用户添加商品到购物车
2. 用户能通过购物车详情页面的表单输入一个优惠码
3. 输入优惠码并提交表单之后，需要来判断该码是否在数据库中存在、当前时间是否在 `valid_from` 和 `valid_to` 有效时间之间、`active` 属性是否为 `True`。
4. 如果优惠码通过上述检查，将优惠码的信息保存在 `session` 中，用折扣重新计算价格并更新购物车中的商品价格
5. 用户提交订单时，将优惠码保存在订单对象中。

在 `coupons` 应用里建立 `forms.py` 文件，添加下列代码：

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

这个表单用于用户输入优惠码。然后来编辑 `coupons` 应用的 `views.py` 文件：

```
from django.shortcuts import render, redirect
from django.utils import timezone
from django.views.decorators.http import require_POST
from .models import Coupon
from .forms import CouponApplyForm

@require_POST
def coupon_apply(request):
    now = timezone.now()
```

```
form = CouponApplyForm(request.POST)
if form.is_valid():
    code = form.cleaned_data['code']
    try:
        coupon = Coupon.objects.get(code__iexact=code, valid_from__lte=now, valid_to__gte=now, active=True)
        request.session['coupon_id'] = coupon.id
    except Coupon.DoesNotExist:
        request.session['coupon_id'] = None
return redirect('cart:cart_detail')
```

这个 `coupon_apply` 视图验证优惠码并将其存储在session中，使用了 `@require_POST` 装饰器令该视图仅接受 `POST` 请求。

这个视图的业务逻辑如下：

1. 使用请求中的数据初始化 `CouponApplyForm`
2. 如果表单通过验证，从表单的 `cleaned_data` 获取 `code`，然后使用 `code` 查询数据库得到 `coupon` 对象，这里使用了过滤参数 `iexact`，进行完全匹配；使用 `active=True` 过滤出有效的优惠码；使用 `timezone.now()` 获取当前时间，`valid_from` 和 `valid_to` 分别采用 `lte`（小于等于）和 `gte`（大于等于）过滤查询以保证当前时间位于有效期内。
3. 将优惠码ID存入当前用户的session。
4. 重定向到 `cart_detail` URL对应的购物车详情页，以显示应用了优惠码之后的金额。

需要为 `coupon_apply` 视图配置URL，在 `coupons` 应用中建立 `urls.py` 文件，添加下列代码：

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

app_name = 'coupons'
urlpatterns = [
    path('apply/', views.coupon_apply, name='apply'),
]
```

然后编辑项目的根路由，增加一行：

```
urlpatterns = [
    # ...
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('', include('shop.urls', namespace='shop')),
]
```

依然记得要把这一行放在 `shop.urls` 上方。

编辑 `cart` 应用中的 `cart.py` 文件，添加下列导入：

```
from coupons.models import Coupon
```

然后在 `cart` 类的 `__init__()` 方法的最后添加从session中获得优惠码ID的语句：

```
class Cart(object):
    def __init__(self, request):
        # ...
        # store current applied coupon
        self.coupon_id = self.session.get('coupon_id')
```

在 `Cart` 类中，我们需要通过 `coupon_id` 获取优惠码信息并将其保存在 `Cart` 对象内，为 `Cart` 类添加如下方法：

```
class Cart(object):
    # ...
    @property
    def coupon(self):
        if self.coupon_id:
```

```
        return Coupon.objects.get(id=self.coupon_id)
    return None

    def get_discount(self):
        if self.coupon:
            return (self.coupon.discount / Decimal('100')) * self.get_total_price()
        return Decimal('0')

    def get_total_price_after_discount(self):
        return self.get_total_price() - self.get_discount()
```

这些方法解释如下：

- `coupon()`：我们使用 `@property` 将该方法定义为属性，如果购物车包含一个 `coupon_id` 属性，会返回该id对应的 `Coupon` 对象
- `get_discount()`：如果包含优惠码id，计算折扣价格，否则返回0。
- `get_total_price_after_discount()`：返回总价减去折扣价之后的折扣后价格。

现在 `Cart` 类就具备了根据优惠码计算折扣价的功能。

现在还需要修改购物车详情视图函数，以便在页面中应用表单和展示折扣金额，修改 `cart` 应用的 `views.py` 文件，增加导入代码：

```
from coupons.forms import CouponApplyForm
```

然后修改 `cart_detail` 视图，添加表单：

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(initial={'quantity': item['quantity'], 'update': True})
    coupon_apply_form = CouponApplyForm()
```

```
return render(request, 'cart/detail.html', {'cart': cart, 'coupon_apply_form': coupon_apply_form})
```

修改 `cart` 应用的购物车模板 `cart/detail.html`，找到如下几行：

```
<tr class="total">
    <td>total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
</tr>
```

替换成如下代码：

```
{% if cart.coupon %}
    <tr class="subtotal">
        <td>Subtotal</td>
        <td colspan="4"></td>
        <td class="num">${{ cart.get_total_price_after_discount }}</td>
    </tr>
    <tr>
        <td>"{{ cart.coupon.code }}" coupon ({{ cart.coupon.discount }}% off)</td>
        <td colspan="4"></td>
        <td class="num neg">- ${{ cart.get_discount|floatformat:"2" }}</td>
    </tr>
{% endif %}

<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price_after_discount|floatformat:"2" }}</td>
</tr>
```

这是新的购物车模板。如果包含一个优惠券，就展示一行购物车总价，再展示一行优惠券信息，最后通过 `get_total_price_after_discount()` 展示折扣后价格。

在同一个文件内，在 `</table>` 后增加下列代码：

```
{# 在紧挨着</table>标签之后插入： #}
<p>Apply a coupon:</p>
<form action="{% url 'coupons:apply' %}" method="post">
    {{ coupon_apply_form }}
    <input type="submit" value="Apply">
    {% csrf_token %}
</form>
```

上边这段代码展示输入优惠码的表单。

在浏览器中打开 <http://127.0.0.1:8000/>，向购物车内加入一些商品，然后进入购物车页面输入优惠码并提交，可以看到如下所示：

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	<input type="button" value="1"/> <input type="button" value="Update"/>	Remove	\$21.2	\$21.2
Subtotal					\$21.20
"SUMMER" coupon (10% off)					- \$2.12
Total					\$19.08
Apply a coupon:					
Code:	<input type="text"/>	<input type="button" value="Apply"/>			
			Continue shopping	<input type="button" value="Checkout"/>	

之后来修改订单模板 `orders/order/create.html`，在其中找到如下部分：

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }} x {{ item.product.name }}
```

```
<span>${{ item.total_price }}</span>
</li>
{% endfor %}
</ul>
```

替换成：

```
<ul>
{% for item in cart %}
<li>
    {{ item.quantity }}x {{ item.product.name }}
    <span>${{ item.total_price|floatformat:"2" }}</span>
</li>
{% endfor %}
{% if cart.coupon %}
<li>
    "{{ cart.coupon.code }}" ({{ cart.coupon.discount }}% off)
    <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
</li>
{% endif %}
</ul>
```

如果有优惠码，现在的订单页面就展示优惠码信息了。继续找到下边这行：

```
<p>Total: ${{ cart.get_total_price }}</p>
```

替换成：

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
```

这样总价也变成了折扣后价格。

在浏览器中打开 <http://127.0.0.1:8000/>，添加商品到购物车然后生成订单，可以看到订单页面的价格现在是折扣后的价格了：

Your order

- 1x Tea powder \$21.20
- "SUMMER" (10% off) - \$2.12

Total: \$19.08

1.3 在订单中记录优惠码信息

像之前说的，我们需要将优惠码信息保存至 `order` 对象中，为此需要修改 `Order` 模型。编辑

编辑 `orders` 应用的 `models.py` 文件，增加导入部分的代码：

```
from decimal import Decimal
from django.core.validators import MinValueValidator, MaxValueValidator
from coupons.models import Coupon
```

然后为 `Order` 模型增加下列字段：

```
class Order(models.Model):
    coupon = models.ForeignKey(Coupon, related_name='orders', null=True, blank=True, on_delete=models.SET_NULL)
    discount = models.IntegerField(default=0, validators=[MinValueValidator(0), MaxValueValidator(100)])
```

这两个字段用于存储优惠码信息。虽然折扣信息保存在Coupon对象中，但这里还是用 `discount` 字段保存了当前的折扣，以免未来优惠码折扣发生变化。为 `coupon` 字段设置了 `on_delete=models.SET_NULL`，优惠码删除时，该外键字段会变成空值。

增加好字段后数据迁移程序。回到 `models.py` 文件，需要修改 `Order` 类中的 `get_total_cost()` 方法：

```
class Order(models.Model):
    # ...
    def get_total_cost(self):
        total_cost = sum(item.get_cost() for item in self.items.all())
        return total_cost - total_cost * (self.discount / Decimal('100'))
```

修改后的 `get_total_cost()` 方法会把折扣也考虑进去。之后还需要修改 `orders` 应用里的 `views.py` 文件中的 `order_create` 视图，以便在生成订单的时候，存储这两个新增的字段。找到下边这行：

```
order = form.save()
```

将其替换成如下代码：

```
order = form.save(commit=False)
if cart.coupon:
    order.coupon = cart.coupon
    order.discount = cart.coupon.discount
order.save()
```

在修改后代码中，通过调用 `OrderCreateForm` 表单对象的 `save()` 方法，创建一个 `order` 对象，使用 `commit=False` 暂不存入数据库。如果购物车对象中有折扣信息，就保存折扣信息。然后将 `order` 对象存入数据库。

启动站点，在浏览器中访问 <http://127.0.0.1:8000/>，使用一个自己创建的优惠码，在完成购买之后，可以到 <http://127.0.0.1:8000/admin/orders/order/> 查看包含优惠码和折扣信息的订单：

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
21 3 <input type="text" value="3"/> <input type="button" value="Tea powder"/>	21,2 <input type="text" value="21,2"/>	1 <input type="text" value="1"/>	<input type="checkbox"/>

还可以修改管理后台的订单详情页和PDF发票，以使其包含优惠码和折扣信息。下边我们将为站点增加国际化功能。

译者注：这里有一个问题：用户提交了订单并清空购物车后，如果再向购物车内添加内容，再次进入购物车详情页面可以发现自动使用了上次使用的优惠券。此种情况的原因是作者把优惠券信息附加到了session上，在提交订单的时候没有

清除。cart对象实例化的时候又取到了相同的优惠券信息。所以需要对程序进行一下改进。

修改orders应用的order_create视图，在生成OrderItem并清空购物车的代码下增加一行：

```
def order_create(request):
    cart = Cart(request)
    if request.method == "POST":
        form = OrderCreateForm(request.POST)
        # 表单验证通过就对购物车内每一条记录生成OrderItem中对应的一条记录
        if form.is_valid():
            order = form.save(commit=False)
            if cart.coupon:
                order.coupon = cart.coupon
                order.discount = cart.coupon.discount
            order.save()
            for item in cart:
                OrderItem.objects.create(order=order, product=item['product'], price=item['price'],
                                         quantity=item['quantity'])
            # 成功生成OrderItem之后清除购物车
            cart.clear()

            # 清除优惠券信息
            request.session['coupon_id'] = None

            # 成功完成订单后调用异步任务发送邮件
            order_created.delay(order.id)
            # 在session中加入订单id
            request.session['order_id'] = order.id
            # 重定向到支付页面
            return redirect(reverse('payment:process'))

    else:
        form = OrderCreateForm()
    return render(request, 'orders/order/create.html', {'cart': cart, 'form': form})
```

2 国际化与本地化

Django对于国际化和本地化提供了完整的支持，允许开发者将站点内容翻译成多种语言，而且可以处理本地化的时间日期数字和时区格式等本地化的显示内容。在开始之前，先需要区分一下 [国际化和本地化](#) 两个概念。国际化和本地化都是一种软件开发过程。国际化（Internationalization，通常缩写为i18n），是指一个软件可以被不同的国家和地区使用，而不会局限于某种语言。本地化（Localization，缩写为l10n）是指对国际化的软件将其进行翻译或者其他本地化适配，使之变成适合某一个国家或地区使用的软件的过程。Django通过自身的国际化框架，可以支持超过50种语言。

2.1 国际化与本地化设置

Django的国际化框架可以让开发者很方便的在Python代码和模板中标注需要翻译的字符串，这个框架依赖于GNU gettext开源软件来生成和管理[消息文件](#)（message file）。消息文件是一个纯文本文件，代表一种语言的翻译，存放着在站点应用中找到的部分或者所有需要翻译的字符串以及对应的某种语言的翻译，就像一个字典一样。消息文件的后缀名是 `.po`。

一旦完成翻译，可以把消息文件编译，以快速访问翻译内容，编译后的消息文件的后缀名是 `.mo`。

2.1.1 国际化与本地化设置

Django提供了一些国际化和本地化的设置，下边一些设置是最重要的：

- `USE_I18N`：布尔值，是否启用国际化功能，默认为 `True`
- `USE_L10N`：布尔值，设置本地化功能是否启用，设置为 `True` 时，数字和日期将采用本地化显示。默认为 `False`
- `USE_TZ`：布尔值，指定时间是否根据时区进行调整，当使用 `startproject` 创建项目时，默认为 `True`

- `LANGUAGE_CODE`：项目的默认语言代码，采用标准的语言代码格式，例如`'en-us'`表示美国英语，`'en-gb'`表示英国英语。这个设置需要 `USE_I18N` 设置为 `True` 才会生效。在 <http://www.i18nguy.com/unicode/language-identifiers.html> 可以找到语言代码清单。
- `LANGUAGES`：一个包含项目所有可用语言的元组，其中每个元素是语言代码和语言名称构成的二元组。可以在 `django.conf.global_settings` 查看所有可用的语言。这个属性可设置的值必须是 `django.conf.global_settings` 中列出的值。
- `LOCALE_PATHS`：一个目录列表，目录内存放项目的翻译文件。
- `TIME_ZONE`：字符串，代表项目所采用的时区。如果使用 `startproject` 启动项目，该值被设置为 `'UTC'`。可以按照实际情况将其设置为具体时区，如 `'Europe/Madrid'`。中国的时区是 `'Asia/Shanghai'`，大小写敏感。

以上是常用的国际化和本地化设置，完整设置请参见

<https://docs.djangoproject.com/en/2.1/ref/settings/#globalization-i18n-l10n>。

2.1.2 国际化和本地化管理命令

Django包含了用于管理翻译的命令如下：

- `makemessages`：运行该命令，会找到项目中所有标注要翻译的字符串，建立或者更新 `locale` 目录下的 `.po` 文件，每种语言会生成单独的 `.po` 文件。
- `compilemessages`：编译所有的 `.po` 文件为 `.mo` 文件。

需要使用GNU gettext工具来执行上述过程，大部分linux发行版自带有该工具。如果在使用mac OSX，可以通过<http://brew.sh/> 使用命令 `brew install gettext` 来安装，之后使用 `brew link gettext --force` 强制链接。对于Windows下的安装，参考<https://docs.djangoproject.com/en/2.0/topics/i18n/translation/#gettext-on-windows> 中的步骤。

2.1.3 如何为项目增加翻译文件

先来看一下增加翻译需要进行的流程：

1. 在Python代码和模板中标注出需要翻译的字符串
2. 运行 `makemessages` 命令建立消息文件
3. 在消息文件中将字符串翻译成另外一种语言，然后运行 `compilemessages` 命令编译消息文件

2.1.4 Django如何确定当前语言

Django使用中间件 `django.middleware.locale.LocaleMiddleware` 来检查HTTP请求中所使用的本地语言。这个中间件做的工作如下：

1. 如果使用 `i18_patterns` (django特殊的一种URL方式，里边包含语言前缀)，中间件会在请求的URL中寻找特定语言的前缀
2. 如果在URL中没有发现语言前缀，会在session中寻找一个键 `LANGUAGE_SESSION_KEY`
3. 如果session中没有该键，会在cookie中寻找一个键。可以通过 `LANGUAGE_COOKIE_NAME` 自定义该cookie的名称，默认是 `django_language`
4. 如果cookie中未找到，找HTTP请求头的 `Accept-Language` 键
5. 如果 `Accept-Language` 头部信息未指定具体语言，则使用 `LANGUAGE_CODE` 设置

注意这个过程只有在开启了该中间件的时候才会得到完整执行，如果未开启中间件，Django直接使用 `LANGUAGE_CODE` 中的设置。

2.2 为项目使用国际化进行准备

我们准备为电商网站增添各种语言的支持，增添英语和西班牙语的支持。编辑 `settings.py` 文件，加入 `LANGUAGES` 设置，放在 `LANGUAGE_CODE` 的旁边：

```
LANGUAGES = (
    ('en', 'English'),
    ('es', 'Spanish'),
)
```

`LANGUAGES` 设置包含两个语言代码和名称组成的元组。语言代码可以指定具体语言如 `en-us` 或 `en-gb`，也可以更模糊，如 `en`。通过这个设置，我们定义了我们的网站仅支持英语和西班牙语。如果不定义 `LANGUAGES` 设置，默认支持所有 django 支持的语言。

设置 `LANGUAGE_CODE` 为如下：

```
LANGUAGE_CODE = 'en'
```

添加 `django.middleware.locale.LocaleMiddleware` 到 `settings.py` 的中间件设置中，位置在 `SessionMiddleware` 中间件之后，`CommonMiddleware` 中间件之前，因为 `LocaleMiddleware` 需要使用 `session`，而 `CommonMiddleware` 需要一种可用语言来解析 URL，`MIDDLEWARE` 设置成如下：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
]
```

django 中间件设置的顺序很重要，中间件会在请求上附加额外的数据，某个中间件会依赖于另外一个中间件附加的数据才能正常工作。

在manage.py文件所在的项目根目录下创建如下目录：

```
locale/  
  en/  
  es/
```

locale 目录是用来存放消息文件的目录，编辑 settings.py 文件加入如下设置：

```
LOCALE_PATH = (  
    os.path.join(BASE_DIR, 'locale/'),  
)
```

LOCALE_PATH 指定了Django寻找消息文件的路径，可以是一系列路径，最上边的路径优先级最高。

当使用 makemessages 命令的时候，消息文件会在我们创建的 locale/ 目录中创建，如果某个应用也有 locale/ 目录，那个应用中的翻译内容会优先在那个应用的目录中创建。

2.3 翻译Python代码中的字符串

为了翻译Python代码中的字符串字面量，需要使用 django.utils.translation 模块中的 gettext() 方法来标注字符串。这个方法返回翻译后的字符串，通常做法是导入该方法然后命名为一个下划线"_"。可以在

<https://docs.djangoproject.com/en/2.0/topics/i18n/translation/> 查看文档。

2.3.1 标记字符串

标记字符串的方法如下：

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

2.3.2 惰性翻译

Django对于所有的翻译函数都有惰性版本，后缀为 `_lazy()`。使用惰性翻译函数的时候，字符串只有被访问的时候才会进行翻译，而不是在翻译函数调用的时候。当字符串位于模块加载的时候才生成的路径中时候特别有效。

使用 `gettext_lazy()` 代替 `gettext()` 方法，只有在该字符串被访问的时候才会进行翻译，所有的翻译函数都有惰性版本。。

2.3.3 包含变量的翻译

被标注的字符串中还可以带有占位符，以下是一个占位符的例子：

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month, 'day': day}
```

通过使用占位符，可以使用字符串变量。例如，上边这个例子的英语如果是 "*Today is April 14*"，翻译成的西班牙语就是 "*Hoy es 14 de Abril*"。当需要翻译的文本中存在变量的时候，推荐使用占位符。

2.3.4 复数的翻译

对于复数形式的翻译，可以采用 `ngettext()` 和 `ngettext_lazy()`。这两个函数根据对象的数量来翻译单数或者复数。使用例子如下：

```
output = ngettext('there is %(count)d product', 'there are %(count)d products', count) % {'count': count}
```

现在我们了解了Python中翻译字面量的知识，可以来为我们的项目添加翻译功能了。

2.3.5 为项目翻译Python字符串字面量

编辑 `settings.py`，导入 `gettext_lazy()`，然后修改 `LANGUAGES` 设置：

```
from django.utils.translation import gettext_lazy as _

LANGUAGES = (
    ('en', _('English')),
    ('es', _('Spanish')),
)
```

这里导入了 `gettext_lazy()` 并使用了别名"`_`"来避免重复导入。将显示的名称也进行了翻译，这样对于不同的语言的人来说，可以看懂并选择他自己的语言。

然后打开系统命令行窗口，输入如下命令：

```
django-admin makemessages --all
```

可以看到如下输出：

```
processing locale en
processing locale es
```

然后查看项目的 `locale` 目录，可以看到如下文件和目录结构：

```
en/
LC_MESSAGES/
    django.po
es/
LC_MESSAGES/
    django.po
```

每个语言都生成了一个 `.po` 消息文件，使用文本编辑器打开 `es/LC_MESSAGES/django.po` 文件，在末尾可以看到如下内容：

```
#: .\myshop\settings.py:107
msgid "English"
msgstr ""

#: .\myshop\settings.py:108
msgid "Spanish"
msgstr ""
```

每一部分的第一行表示在那个文件的第几行发现了需翻译的内容，每个翻译包含两个字符串：

- `msgid`：源代码中的字符串
- `msgstr`：被翻译成的字符串，默认为空，需要手工添加。

添加好翻译之后的文件如下：

```
#: myshop/settings.py:117
msgid "English"
msgstr "Inglés"

#: myshop/settings.py:118
```

```
msgid "Spanish"
msgstr "Español"
```

保存这个文件，之后执行命令编译消息文件：

```
django-admin compilemessages
```

可以看到输出如下：

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

这表明已经编译了翻译文件，此时查看 `locale` 目录，其结构如下：

```
en/
  LC_MESSAGES/
    django.mo
    django.po
es/
  LC_MESSAGES/
    django.mo
    django.po
```

可以看到每种语言都生成了 `.mo` 文件。

我们已经翻译好了语言名称本身。现在我们来试着翻译一下 `Order` 模型的所有字段，修改 `orders` 应用的 `models.py` 文件：

```
from django.utils.translation import gettext_lazy as _

class Order(models.Model):
    first_name = models.CharField(_('frist name'), max_length=50)
    last_name = models.CharField(_('last name'), max_length=50)
    email = models.EmailField(_('e-mail'))
    address = models.CharField(_('address'), max_length=250)
    postal_code = models.CharField(_('postal code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    ....
```

我们为每个显示出来的字段标记了翻译内容，也可以使用 `verbose_name` 属性来命名字段。在 `orders` 应用中建立如下目录：

```
locale/
  en/
  es/
```

通过创建 `locale` 目录，当前应用下的翻译内容会优先保存到这个目录中，而不是保存在项目根目录下的 `locale` 目录中。这样就可以为每个应用配置独立的翻译文件。

在系统命令行中执行：

```
django-admin makemessages --all
```

输出为：

```
processing locale es
processing locale en
```

使用文本编辑器打开 `locale/es/LC_MESSAGES/django.po`，可以看到Order模型的字段翻译，在 `msgid` 中为对应的 `msgid` 字符串加上西班牙语的翻译：

```
#: orders/models.py:10
msgid "first name"
msgstr "nombre"

#: orders/models.py:11
msgid "last name"
msgstr "apellidos"

#: orders/models.py:12
msgid "e-mail"
msgstr "e-mail"

#: orders/models.py:13
msgid "address"
msgstr "dirección"

#: orders/models.py:14
msgid "postal code"
msgstr "código postal"

#: orders/models.py:15
msgid "city"
msgstr "ciudad"
```

添加完翻译之后保存文件。

除了常用的文本编辑软件，还可以考虑使用Poedit编辑翻译内容，该软件同样依赖gettext，支持Linux，Windows和macOS X。可以在 <https://poedit.net/> 下载该软件。

下边来翻译项目使用的表单。`OrderCreateForm`这个表单类无需翻译，因为它会自动使用`Order`类中我们刚刚翻译的`verbose_name`。现在我们去翻译`cart`和`coupons`应用中的内容。

在`cart`应用的`forms.py`文件中，导入翻译函数，为`CartAddProductForm`类的`quantity`字段增加一个参数`label`，代码如下：

```
from django import forms
from django.utils.translation import gettext_lazy as _
PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(choices=PRODUCT_QUANTITY_CHOICES, coerce=int, label=_('Quantity'))
    update = forms.BooleanField(required=False, initial=False, widget=forms.HiddenInput)
```

译者注：红字部分是本书上一版的遗留，无任何作用，读者可以忽略。

之后修改`coupons`应用的`forms.py`文件，为`CouponApplyForm`类增加翻译：

```
from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))
```

我们为`code`字段增加了一个`label`标签用于展示翻译后的字段名称。

2.4 翻译模板

Django为翻译模板内容提供了`{% trans %}`和`{% blocktrans %}`两个模板标签用于翻译内容，如果要启用这两个标签，需要在模板顶部加入`{% load i18n %}`。

2.4.1 使用`{% trans %}`模板标签

`{% trans %}`标签用来标记一个字符串，常量或者变量用于翻译。Django内部也是该文本执行`gettext()`等翻译函数。标记字符串的例子是：

```
{% trans "Text to be translated" %}
```

也可以像其他标签变量一样，使用`as`将翻译后的结果放入一个变量中，在其他地方使用。下面的例子使用了一个变量`greeting`：

```
{% trans "Hello!" as greeting %}
<h1>{{ greeting }}</h1>
```

这个标签用于比较简单的翻译，但不能用于带占位符的文字翻译。

2.4.2 使用`{% blocktrans %}`模板标签

`{% blocktrans %}`标签可以标记包含常量和占位符的内容用于翻译，下边的例子展示了使用一个`name`变量的翻译：

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

可以使用`with`，将具体的表达式设置为变量的值，此时在`blocktrans`块内部不能够再继续访问表达式和对象的属性，下面是一个使用了`capfirst`装饰器的例子：

```
{% blocktrans with name=user.name|capfirst %}  
    Hello {{ name }}!  
{% endblocktrans %}
```

如果翻译内容中包含变量，使用 `{% blocktrans %}` 代替 `{% trans %}`。

2.4.3 翻译商店模板

编辑 `shop` 应用的 `base.html`，在其顶部加入 `i18n` 标签，然后标注如下要翻译的部分：

```
{% load i18n %}  
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8"/>  
    <title>{% block title %}{% trans "My shop" %}{% endblock %}</title>  
    <link href="{% static "css/base2.css" %}" rel="stylesheet">  
</head>  
<body>  
<div id="header">  
    <a href="/" class="logo">{% trans "My shop" %}</a>  
</div>  
<div id="subheader">  
    <div class="cart">  
        {% with total_items=mycart|length %}  
            {% if mycart|length > 0 %}  
                {% trans "Your cart" %}:  
                <a href="{% url 'cart:cart_detail' %}">  
                    {% blocktrans with total_items_plural=total_items|pluralize total_price=cart.get_total_price %}  
                    {{ total_items }} items{{ total_items_plural }}, ${{ total_price }}  
                    {% endblocktrans %}  
                </a>  
            {% endif %}  
        {% endwith %}  
    </div>  
</div>
```

```
        </a>
    {% else %}
        {% trans "Your cart is empty." %}
    {% endif %}
    {% endwith %}
</div>
</div>
<div id="content">
    {% block content %}
    {% endblock %}
</div>
</body>
</html>
```

注意`{% blocktrans %}`展示购物车总价部分的方法，在原来的模板中，我们使用了：

```
{{ total_items }} item{{ total_items|pluralize }},
${{ cart.get_total_price }}
```

现在改用`{% blocktrans with ... %}`来为`total_items|pluralize`（使用了过滤器）和`cart.get_total_price`（访问对象的方法）创建占位符：

编辑`shop`应用的`shop/product/detail.html`，紧接着`{% extends %}`标签导入`i18n`标签：

```
{% load i18n %}
```

之后找到下边这一行：

```
<input type="submit" value="Add to cart">
```

将其替换成：

```
<input type="submit" value="{% trans "Add to cart" %}">
```

现在来翻译 orders 应用，编辑 `orders/order/create.html`，标记如下翻译内容：

```
{% extends 'shop/base.html' %}
{% load i18n %}
{% block title %}
    {% trans "Checkout" %}
{% endblock %}

{% block content %}
    <h1>{% trans "Checkout" %}</h1>

    <div class="order-info">
        <h3>{% trans "Your order" %}</h3>
        <ul>
            {% for item in cart %}
                <li>
                    {{ item.quantity }}x {{ item.product.name }}
                    <span>${{ item.total_price|floatformat:"2" }}</span>
                </li>
            {% endfor %}
            {% if cart.coupon %}
                <li>
                    {% blocktrans with code=cart.coupon.code discount=cart.coupon.discount %}
                        "{{ code }}" ({{ discount }}% off)
                    {% endblocktrans %}
                    <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
                </li>
            {% endif %}
        </ul>
    </div>
</div>
```

```
<p>{% trans "Total" %}: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
</div>

<form action="." method="post" class="order-form" novalidate>
{{ form.as_p }}
<p><input type="submit" value="{% trans "Place order" %}"></p>
{% csrf_token %}
</form>
{% endblock %}
```

到现在我们完成了如下文件的翻译：

- `shop` 应用的 `shop/product/list.html` 模板
- `orders` 应用的 `orders/order/created.html` 模板
- `cart` 应用的 `cart/detail.html` 模板

之后来更新消息文件，打开命令行窗口执行：

```
django-admin makemessages --all
```

此时 `myshop` 项目下的 `locale` 目录内有了对应的 `.po` 文件，而 `orders` 应用的翻译文件优先存放在应用内部的 `locale` 目录中。

编辑所有 `.po` 文件，在 `msgstr` 属性内添加西班牙语翻译。你也可以直接复制随书代码内对应文件的内容。

执行命令编译消息文件：

```
django-admin compilemessages
```

可以看到如下输出：

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
processing file django.po in myshop/orders/locale/en/LC_MESSAGES
processing file django.po in myshop/orders/locale/es/LC_MESSAGES
```

针对每一个 .po 文件都会生成对应的 .mo 文件。

2.5 使用Rosetta翻译界面

Rosetta是一个第三方应用，通过Django管理后台编辑所有翻译内容，让 .po 文件的管理变得更加方便，先通过 pip 安装该模块：

```
pip install django-rosetta==0.8.1
```

之后在 `settings.py` 中激活该应用：

```
INSTALLED_APPS = [
    # ...
    'rosetta',
]
```

然后需要为Rosetta配置相应的URL，其二级路由已经配置好，修改项目根路由增加一行：

```
urlpatterns = [
    # ...
    path('rosetta/', include('rosetta.urls')),
```

```
    path('', include('shop.urls', namespace='shop')),  
]
```

这条路径也需要在 `shop.urls` 上边。

然后启动站点，使用管理员身份登录 <http://127.0.0.1:8000/rosetta/>，再转到 <http://127.0.0.1:8000/rosetta/>，点击右上的THIRD PARTY以列出所有的翻译文件，如下图所示：

The screenshot shows the Rosetta translation interface. At the top, there's a navigation bar with 'Home' and 'Language selection'. Below it, a filter bar has 'Filter' set to 'PROJECT' and 'THIRD PARTY' selected. The main area is divided into two sections: 'English' and 'Spanish'. Each section has a table with columns: APPLICATION, PROGRESS, MESSAGES, TRANSLATED, FUZZY, OBSOLETE, and FILE.

APPLICATION	PROGRESS	MESSAGES	TRANSLATED	FUZZY	OBSOLETE	FILE
Myshop	0%	12	0	0	0	/Users/zenx/dbe/myshop/locale/en/LC_MESSAGES/django.po
Orders	0%	13	0	0	0	/Users/zenx/dbe/myshop/orders/locale/en/LC_MESSAGES/django.po

APPLICATION	PROGRESS	MESSAGES	TRANSLATED	FUZZY	OBSOLETE	FILE
Myshop	100%	12	12	0	0	/Users/zenx/dbe/myshop/locale/es/LC_MESSAGES/django.po
Orders	100%	13	13	0	0	/Users/zenx/dbe/myshop/orders/locale/es/LC_MESSAGES/django.po
Rosetta	73%	37	28	1	2	/Users/zenx/env/dbe3/lib/python3.6/site-packages/rosetta/locale/es/LC_MESSAGES/django.po

点开Spanish下边的Myshop链接，可以看到列出了所有需要翻译的内容：

Translate into Spanish

Display: UNTRANSLATED ONLY TRANSLATED ONLY FUZZY ONLY ALL

ORIGINAL	SPANISH	FUZZY	OCCURRENCES(S)
Quantity	Cantidad	<input type="checkbox"/>	cart/forms.py:12
Coupon	Cupón	<input type="checkbox"/>	coupons/forms.py:6
English	Inglés	<input type="checkbox"/>	myshop/settings.py:117
Spanish	Español	<input type="checkbox"/>	myshop/settings.py:118
My shop	Mi tienda	<input type="checkbox"/>	shop/templates/shop/base.html:7 shop/templates/shop/base.html:12
Your cart	Tu carro	<input type="checkbox"/>	shop/templates/shop/base.html:18

可以手工编辑需要翻译的地方，OCCURRENCES(S)栏显示了该翻译所在的文件名和行数，对于那些占位符翻译的内容，显示为这样：

<code>%(total_items)s item%</code> <code>(total_items_plural)s,</code> <code>\$%(total_price)s</code>	<code>%(total_items)s producto%</code> <code>(total_items_plural)s,</code> <code>\$%(total_price)s</code>	<input type="checkbox"/>	shop/templates/shop/base.html:20
---	---	--------------------------	----------------------------------

Rosetta对占位符使用了不同的背景颜色，在手工输入翻译内容的时候注意不要破坏占位符的结构，例如要翻译下边这一行：

```
%({total_items)s item%({total_items_plural)s, ${%(total_price)s
```

应该输入：

```
%({total_items)s producto%({total_items_plural)s, ${%(total_price)s
```

可以参考本章随书代码中的西班牙语翻译来录入翻译内容。

结束输入的时候，点击一下Save即可将当前翻译的内容保存到 `.po` 文件中，当保存之后，Rosetta会自动进行编译，所以无需执行 `compilemessages` 命令。然而要注意Rosetta会直接读写 `locale` 目录，注意要给予其相应的权限。

如果需要其他用户来编辑翻译内容，可以到 <http://127.0.0.1:8000/admin/auth/group/add/> 新增一个用户组叫 `translators`，然后到 <http://127.0.0.1:8000/admin/auth/user/> 编辑用户的权限以给予其修改翻译的权限，将该用户加入到 `translators` 用户组内。仅限超级用户和 `translators` 用户组内的用户才能使用Rosetta。

Rosetta的官方文档在 <https://django-rosetta.readthedocs.io/en/latest/>。

特别注意的是，当Django已经在生产环境运行时，如果修改和新增了翻译，在运行了 `compilemessages` 命令之后，只有重新启动Django才会让新的翻译生效。

2.6 待校对翻译Fuzzy translations

你可能注意到了，Rosetta页面上有一列叫做Fuzzy。这不是Rosetta的功能，而是 `gettext` 提供的功能。如果将fuzzy设置为true，则该条翻译不会包含在编译后的消息文件中。这个字段用来标记需要由用户进行检查的翻译内容。当 `.po` 文

件更新了新的翻译字符串时，很可能一些翻译被自动标成了fuzzy。这是因为：在 `gettext` 发现一些 `msgid` 被修改过的时候，`gettext` 会将其与它认为的旧有翻译进行匹配，然后标注上fuzzy。看到fuzzy出现的时候，人工翻译者必须检查该条翻译，然后取消fuzzy，之后再行编译。

2.7 国际化URL

Django提供两种国际化URL的特性：

- **Language prefix in URL patterns** 语言前缀URL模式：在URL的前边加上不同的语言前缀构成不同的基础URL
- **Translated URL patterns** 翻译URL模式：基础URL相同，把基础URL按照不同语言翻译给用户得到不同语言的URL

使用翻译URL模式的优点是对搜索引擎友好。如果采用语言前缀URL，则必须要为每一种语言进行索引，使用翻译URL模式，则一条URL就可以匹配全部语言。下边来看一下两种模式的使用：

2.7.1 语言前缀URL模式

Django可以为不同语言在URL前添加前缀，例如我们的网站，英语版以 `/en/` 开头，而西班牙语版以 `/es/` 开头。

要使用语言前缀URL模式，需要启用 `LocaleMiddleware` 中间件，用于从不同的URL中识别语言，在之前我们已经添加过该中间件。

我们来为URL模式增加前缀，现在需要修改项目的根 `urls.py` 文件：

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
```

```
path('pyament/', include('payment.urls', namespace='payment')),
path('coupons/', include('coupons.urls', namespace='coupons')),
path('rosetta/', include('rosetta.urls')),
path('', include('shop.urls', namespace='shop')),
)
```

可以混用未经翻译的标准URL与 `i18n_patterns` 类型的URL，使部分URL带有语言前缀，部分不带前缀。但最好只使用翻译URL，以避免把翻译过的URL匹配到未经翻译过的URL模式上。

现在启动站点，到 <http://127.0.0.1:8000/>，Django的语言中间件会按照之前介绍的顺序来确定本地语言，然后重定向到带有语言前缀的URL。现在看一下浏览器的地址栏，应该是 <http://127.0.0.1:8000/en/>。当前语言是由请求头 `Accept-Language` 所设置，或者就是 `LANGUAGE_CODE` 的设置。

2.7.2 翻译URL模式

Django支持在URL模式中翻译字符串。针对不同的语言，可以翻译出不同的URL。在 `urls.py` 中，使用 `gettext_lazy()` 来标注字符串。

编辑 `myshop` 应用的根 `urls.py`，为 `cart`，`orders`，`payment` 和 `coupons` 应用配置URL：

```
from django.utils.translation import gettext_lazy as _

urlpatterns = i18n_patterns(
    path(_('admin/'), admin.site.urls),
    path(_('cart/'), include('cart.urls', namespace='cart')),
    path(_('orders/'), include('orders.urls', namespace='orders')),
    path(_('payment/'), include('payment.urls', namespace='payment')),
    path(_('coupons/'), include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

编辑 `orders` 应用的 `urls.py` 文件，修改成如下：

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('create/'), views.order_create, name='order_create'),
    # ...
]
```

修改 `payment` 应用的 `urls.py` 文件，修改成如下：

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('process/'), views.payment_process, name='process'),
    path(_('done/'), views.payment_done, name='done'),
    path(_('canceled/'), views.payment_canceled, name='canceled'),
]
```

对于 `shop` 应用的URL不需要修改，因为其URL是动态建立的。

执行命令进行编译，更新消息文件：

```
django-admin makemessages --all
```

启动站点，访问 <http://127.0.0.1:8000/en/rosetta/>，点击Spanish下的Myshop，可以看到出现了URL对应的翻译。

可以点击Untranslated查看所有尚未翻译的字符串，然后输入翻译内容。

2.8 允许用户切换语言

在之前的工作中，我们配置好了英语和西班牙语的翻译，应该给用户提供切换语言的选项，为此准备给网站增加一个语言选择器，列出所有支持的语言，显示为一系列链接。

编辑 `shop` 应用下的 `base.html`，找到下边这三行：

```
<div id="header">
    <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

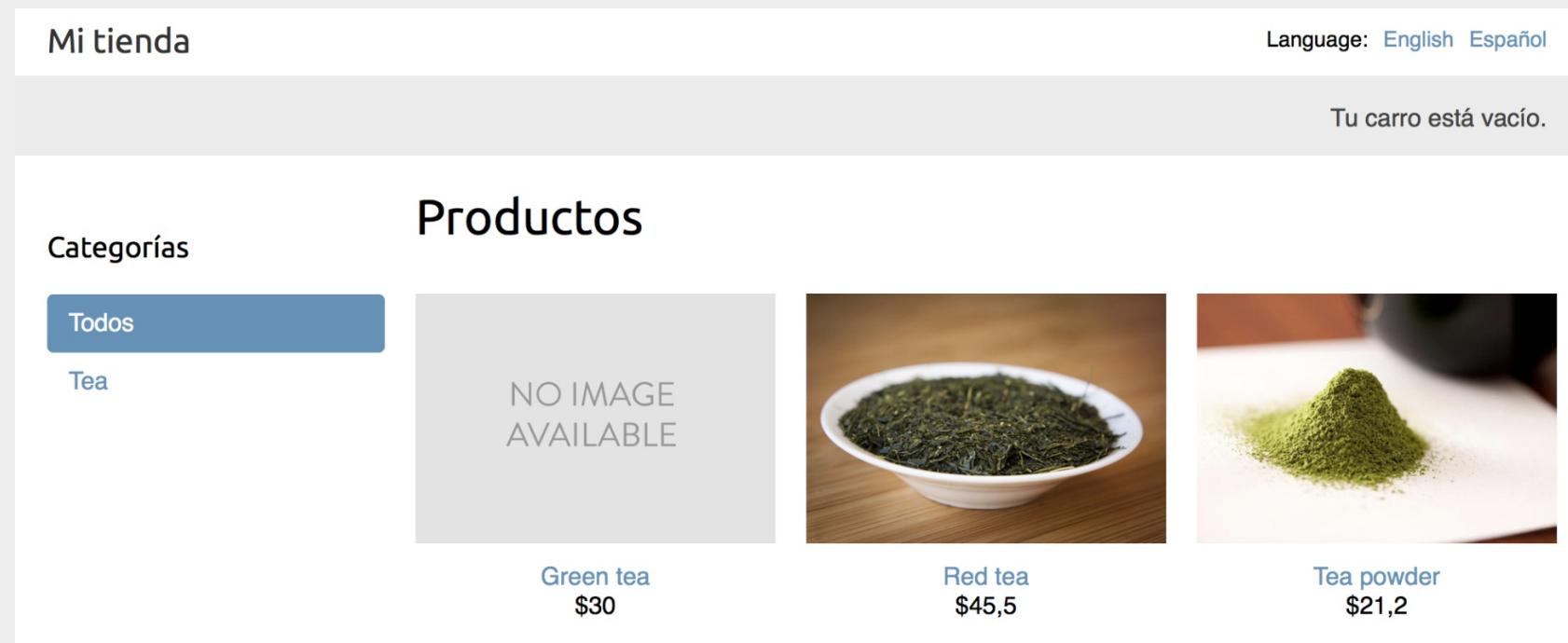
将其替换成：

```
<div id="header">
    <a href="/" class="logo">{% trans "My shop" %}</a>
    {% get_current_language as LANGUAGE_CODE %}
    {% get_available_languages as LANGUAGES %}
    {% get_language_info_list for LANGUAGES as languages %}
    <div class="languages">
        <p>{% trans "Language" %}:</p>
        <ul class="languages">
            {% for language in languages %}
                <li>
                    <a href="/{{ language.code }}/" 
                        {% if language.code == LANGUAGE_CODE %} class="selected"{% endif %}>
                        {{ language.name_local }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </div>
</div>
```

这个就是我们的语言选择器，逻辑如下：

1. 页面的最上方加载`{% load i18n %}`
2. 使用`{% get_current_language %}`标签用于获取当前语言
3. 使用`{% get_available_languages %}`标签用于从`LANGUAGES`里获取所有可用的支持语言
4. 使用`{% get_language_info_list %}`是为了快速获取语言的属性而设置的变量
5. 用循环列出了所有可支持的语言，对于当前语言设置CSS类为`select`

启动站点到<http://127.0.0.1:8000/>，可以看到页面右上方出现了语言选择器，如下图：



2.9 使用django-parler翻译模型

Django没有提供直接可用的模型翻译功能，必须采用自己的方式实现模型翻译。有一些第三方工具可以翻译模型字段，每个工具存储翻译的方式都不相同。其中一个工具叫做 `django-parler`，提供了高效的翻译管理，还能够与管理后台进行集成。

`django-parler` 的工作原理是为每个模型建立一个对应的翻译数据表，表内每条翻译记录通过外键连到翻译文字所在的模型，表内还有一个 `language` 字段，用于标记是何种语言。

2.9.1 安装django-parler

使用 `pip` 安装 `django-parler`：

```
pip install django-parler==1.9.2
```

在 `settings.py` 内激活该应用：

```
INSTALLED_APPS = [
    # ...
    'parler',
]
```

继续添加下列设置：

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
    }
}
```

```
        'hide_untranslated': False,  
    }  
}
```

该配置的含义是指定了 `django-parler` 的可用语言为 `en` 和 `es`，然后指定了默认语言为 `en`，然后指定 `django-parler` 不要隐藏未翻译的内容。

2.9.2 翻译模型字段

我们为商品品类添加翻译。 `django-parler` 提供一个 `TranslatableModel` 类（此处作者原文有误，写成了 `TranslatedModel`）和 `TranslatedFields` 方法来翻译模型的字段。编辑 `shop` 应用的 `models.py` 文件，添加导入语句：

```
from parler.models import TranslatableModel, TranslatedFields
```

然后修改 `Category` 模型的 `name` 和 `slug` 字段：

```
class Category(TranslatableModel):  
    translations = TranslatedFields(  
        name=models.CharField(max_length=200, db_index=True),  
        slug=models.SlugField(max_length=200, db_index=True, unique=True)  
    )
```

`Category` 类现在继承了 `TranslatableModel` 类，而不是原来的 `models.Model`，`name` 和 `slug` 字段被包含在了 `TranslatedFields` 包装器里。

编辑 `Product`，`name`，`slug`，`description`，和上边一样的方式：

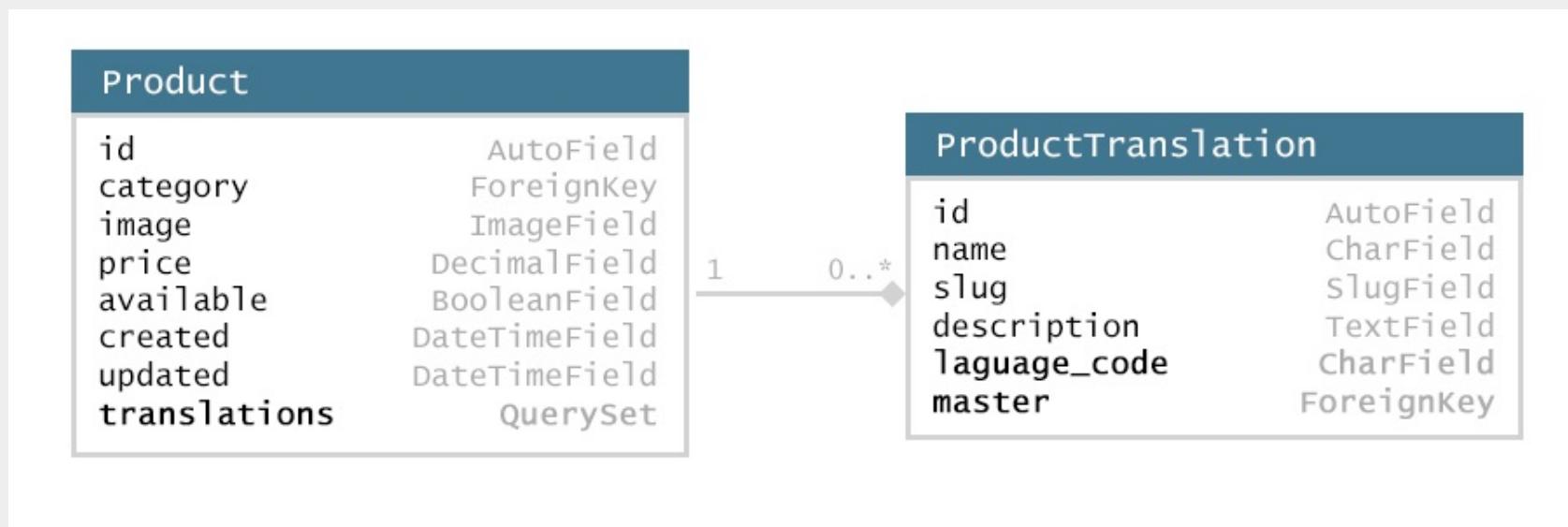
```
class Product(TranslatableModel):  
    translations = TranslatedFields(  
        name=models.CharField(max_length=200, db_index=True),  
        slug=models.SlugField(max_length=200, db_index=True, unique=True),  
        description=models.TextField()  
    )
```

```

        name=models.CharField(max_length=200, db_index=True),
        slug=models.SlugField(max_length=200, db_index=True),
        description=models.TextField(blank=True)
    )
category = models.ForeignKey(Category, related_name='products')
image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)
price = models.DecimalField(max_digits=10, decimal_places=2)
available = models.BooleanField(default=True)
created = models.DateTimeField(auto_now_add=True)
updated = models.DateTimeField(auto_now=True)

```

`django-parler` 通过新创建模型为其他模型提供翻译，在下图可以看到 `Product` 与其对应的翻译模型 `ProductTranslation` 之间的关系：



译者注：此时如果运行站点，一些IDE会提示模型的字段找不到，这个对于实际运行程序没有影响，该字段依然可用。

`django-parler` 生成的 `ProductTranslation` 类包含 `name`, `slug`, `description`, 和一个 `language_code` 字段, 还有一个外键连接到 `Product` 类, 针对一个 `Product` 模型, 会按照每种语言生成一个对应的 `ProductTranslation` 对象。

由于翻译的部分和原始的类是独立的两个模型, 因此一些ORM的功能无法使用, 比如不能在 `Product` 类中根据一个翻译后的字段进行排序, 也不能在 `Meta` 类的 `ordering` 属性中使用翻译的字段。

所以编辑 `shop` 应用的 `models.py` 文件, 注释掉 `ordering` 设置:

```
class Category(TranslatableModel):
    # ...
    class Meta:
        # ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

对于 `Product` 类, 也要注释掉 `ordering`, 还需要注释掉 `index_together`, 这是因为目前的 `django-parler` 不支持联合索引的验证关系。如下图:

```
class Product(TranslatableModel):
    # ...
    class Meta:
        pass
        # ordering = ('name',)
        # index_together = (('id', 'slug'),)
```

译者注: 原书在这里遗漏了 `pass`, 不要忘记加上。

关于 `django-parler` 的兼容性, 可以在 <https://django-parler.readthedocs.io/en/latest/compatibility.html> 查看。

2.9.3 将 django-parler 集成到管理后台

django-parler 易于集成到django管理后台中，包含一个 `TranslatableAdmin` 类代替了原来的 `ModelAdmin` 类。

编辑 `shop` 应用的 `admin.py` 文件，导入该类：

```
from parler.admin import TranslatableAdmin
```

修改 `CategoryAdmin` 和 `ProductAdmin` 类，使其继承 `TranslatableAdmin` 而不是 `ModelAdmin` 类，`django-parler` 不支持 `prepopulated_fields` 属性，但支持相同功能的 [`get_prepopulated_fields\(\)`方法](#)，因此将两个类修改如下：

```
from django.contrib import admin
from .models import Category, Product
from parler.admin import TranslatableAdmin

@admin.register(Category)
class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}


@admin.register(Product)
class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'price', 'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}
```

现在在管理后台内也能进行对翻译模型的管理了。现在可以执行数据迁移程序。

2.9.4 迁移翻译模型数据

打开shell执行下列命令：

```
python manage.py makemigrations shop --name "translations"
```

会看到如下输出：

```
Migrations for 'shop':
shop\migrations\0002_translations.py
- Create model CategoryTranslation
- Create model ProductTranslation
- Change Meta options on category
- Change Meta options on product
- Remove field name from category
- Remove field slug from category
- Alter index_together for product (0 constraint(s))
- Add field master to producttranslation
- Add field master to categorytranslation
- Remove field description from product
- Remove field name from product
- Remove field slug from product
- Alter unique_together for producttranslation (1 constraint(s))
- Alter unique_together for categorytranslation (1 constraint(s))
```

`django-parler` 动态地创建了 `CategoryTranslation` 和 `ProductTranslation`。注意，原模型中需要翻译的字段从原模型中删除了，这意味着这几个字段的数据全都丢失了，必须启动站点后重新录入。

之后运行数据迁移：

```
python manage.py migrate shop
```

可以看到下列输出：

```
Applying shop.0002_translations... OK
```

现在数据已经和数据库同步好了。

启动站点，访问 <http://127.0.0.1:8000/en/admin/shop/category/>，可以看到已经存在的模型失去了那些需要翻译的字段。点击一个 `category` 对象进行修改，可以看到包含了两个不同的表格，一个对应英语，一个对应西班牙语，如下图所示：

Change category (English)

[HISTORY](#)[VIEW ON SITE ➔](#)[English](#)[Spanish](#)**Name:**

Tea

Slug:

tea

[Delete](#)[Save and add another](#)[Save and continue editing](#)**SAVE**

为所有已存在category记录都添加名称和简称，再为其添加西班牙语的名称和简称，然后点击SAVE按钮，确保在切换标签之前点击了SAVE按钮，否则数据不会被保存。

之后到 <http://127.0.0.1:8000/en/admin/shop/product/> 进行同样的工作：补充每个商品的名称、简称、描述以及对应的西班牙语翻译。

2.9.5 视图中加入翻译功能

为了正常使用翻译后的模型，必须让 shop 应用的视图对翻译后的字段也能够获取QuerySet，终端内输入 `python manage.py shell` 进入带Django环境的命令行模式来试验一下经过翻译后的查询操作：

看一下如何查询翻译后的字段。为了获取某种语言的查询结果集，需要使用Django的 `activate()` 函数：

```
>>> from shop.models import Product
>>> from django.utils.translation import activate
>>> activate('es')
>>> product=Product.objects.first()
>>> product.name
'Té verde'
```

另外一种根据不同语言查询的方式是使用 `django-parler` 提供的 `language()` 模型管理器：

```
>>> product=Product.objects.language('en').first()
>>> product.name
'Green tea'
```

当查询翻译字段时，会根据所指定的语言返回结果。可以通过设置管理器的属性得到不同语言的结果，类似这样：

```
>>> product.set_current_language('es')
>>> product.name
'Té verde'
>>> product.get_current_language()
'es'
```

如果需要使用 `filter` 功能，需要使用 `translations__` 语法，例子如下：

```
>>> Product.objects.filter(translations__name='Green tea')
<TranslatableQuerySet [<Product: Té verde>]>
```

了解了基础操作，可以来修改我们自己的视图中的查询方法了，修改 `shop` 应用中的 `views.py`，找到 `product_list` 视图中如下这行：

```
category = get_object_or_404(Category, slug=category_slug)
```

替换成如下内容：

```
language = request.LANGUAGE_CODE  
category = get_object_or_404(Category, translations__language_code=language, translations__slug=category_slug)
```

然后编辑 `product_detail` 视图，找到下边这行：

```
product = get_object_or_404(Product, id=id, slug=slug, available=True)
```

替换成如下内容：

```
language = request.LANGUAGE_CODE  
product = get_object_or_404(Product, id=id, translations__language_code=language, translations__slug=slug,  
                             available=True)
```

`product_list` 和 `product_detail` 现在都具备了根据翻译字段查询数据库的功能。启动站点，到 <http://127.0.0.1:8000/es/>，应该可以看到商品名称全部都变成了西班牙语，如下图：

Tu carro está vacío.

Productos

Categorías

[Todos](#)[Té](#)NO IMAGE
AVAILABLE

Té verde
\$30



Té rojo
\$45,5

Té en polvo
\$21,2

可以看到通过每个商品的 `slug` 字段生成的URL也变成了西班牙语。比如一个商品的URL在西班牙语下是 <http://127.0.0.1:8000/es/2/te-rojo/>，在英语里则是 <http://127.0.0.1:8000/en/2/red-tea/>。如果到一个商品详情页，能够看到翻译后的URL和内容如下：

Tu carro está vacío.



Té rojo

Té

\$45,5

Cantidad: 1

Añadir al carro

El té pu-erh es conocido en Occidente también como té rojo (en chino: 普洱茶, pinyin: pǔ'ěrchá) y su nombre proviene de la región de Pu'er de Yunnan China, de donde procede. Se trata de un té inusual en China, siendo este el mayor productor del té rojo o pu-erh del mundo.

在 <https://django-parler.readthedocs.io/en/latest/> 可以查看 django-parler 的文档。

现在已经知道了如何翻译Python代码，模板，URL和模型的字段，站点已经可以提供不同语言的服务了。为了完成国际化和本地化的过程，还需要对本地的日期，时间，数字格式进行设置。

2.10 本地格式化

根据用户的国家和地区，需要以不同的格式显示日期，时间和数字。本地化格式可以通过 `settings.py` 里的 `USE_L10N` 设置为 `True` 来开启。

当 `USE_L10N` 设置为开启的时候，Django在渲染模板的时候，会尽可能的尝试使用当前本地化的方式进行输出。可以看到我们的站点的小数点是一个圆点显示的，切换到西班牙语的时候，小数点显示为一个逗号。这是通过对每种语言进行不同的格式设置实现的，对于支持的每种语言的格式，Django都有对应的配置文件，例如针对西班牙语的配置文件可以查看 <https://github.com/django/django/blob/stable/2.0.x/django/conf/locale/es/formats.py>。

通常情况下，只要设置 `USE_L10N` 为 `True`，Django就会自动应用本地化格式。然而，站点内可能有些内容并不想使用本地化格式，尤其那些标准数据例如代码或者是JSON字符串的内容。

Django提供了一个 `{% localize %}` 模板标签，用于控制模板或者模板片段开启或关闭本地化输出。为了使用这个标签，必须在模板开头使用 `{% load l10n %}` 标签。下边是一个如何在模板中控制开启/关闭本地化输出的例子：

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

Django还提供了两个模板过滤器用于控制本地化，分别是 `localize` 和 `unlocalize`，用来强制让一个值开启/关闭本地化显示。用法如下：

```
{{ value|localize }}
{{ value|unlocalize }}
```

除了这两个方法之外，还可以采取自定义格式文件方式，具体看

<https://docs.djangoproject.com/en/2.0/topics/i18n/formatting/#creating-custom-format-files>。

2.11 用django-localflavor验证表单字段

`django-localflavor` 是一个第三方模块，包含一系列特别针对本地化验证的工具，比如为每个国家单独设计的表单和模型字段，对于验证某些国家的地区，电话号码，身份证件，社会保险号码等非常方便。这个模块是按照ISO 3166国家代码标准编写的。

安装 `django-localflavor`：

```
pip install django-localflavor==2.0
```

在 `settings.py` 中激活该应用：

```
INSTALLED_APPS = [
    # ...
    'localflavor',
]
```

为了使用该模块，我们给订单增加一个美国邮编字段和对应验证，必须是一个有效的美国邮编才能建立订单。

编辑 `orders` 应用的 `forms.py` 文件，修改成如下：

```
from localflavor.us.forms import USZipCodeField

class OrderCreateForm(forms.ModelForm):
    postal_code = USZipCodeField()
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address', 'postal_code', 'city']
```

从 `localflavor` 的 `us` 模块中导入 `USZipCodeField` 字段类型，将 `OrderCreateForm` 类的 `postal_code` 字段设置为该类型。

运行站点，到 <http://127.0.0.1:8000/en/orders/create/>，输入一些不符合美国邮编的邮政编码，可以看到表单的错误提示：

Enter a zip code in the format XXXXX or XXXXX-XXXX.

这只是一个针对给字段附加本地化验证的一个简单例子。`localflavor` 提供的组件对于将站点快速适配到某些国家非常有用。可以在 <https://django-localflavor.readthedocs.io/en/latest/> 阅读 `django-flavor` 的官方文档。

现在就结束了所有国际化和本地化配置的工作，下一步是建立一个商品推荐系统。

3 创建商品推荐系统

商品推荐系统可以预测用户对一个商品的喜爱程度或者评价高低，根据用户的行为和收集到的用户数据，选择可能和用户相关的产品推荐给用户。在电商行业，推荐系统使用的非常广泛。推荐系统可以帮助用户从浩如烟海的商品中选出自己感兴趣的商品。好的推荐系统可以增加用户粘性，对电商平台则意味着销售额的提高。

我们准备建立一个简单但是强大的商品推荐系统，用于推荐经常被一起购买的商品，这些商品基于用户过去的购买数据来给用户进行推荐。我们打算在两个页面向用户推荐商品：

- 首先是商品详情页。我们会在此展示一些与当前商品一起购买的商品。展示的文字类似：Users who bought this also bought X, Y, Z. 所以我们需要一个数据结构来存放所有与该商品一同购买的次数。
- 其次是购物车详情页。这时将不同商品与购物车中所有商品的关联购买次数进行求和再进行排名。

我们将使用Redis数据库记录一起购买的商品。我们在第六章已经使用过Redis，如果还没有安装Redis，可以参考该章节的内容。

3.1 根据之前的购买记录推荐商品

现在，需要根据用户加入到购物车内的商品计算排名。对于我们网站每一个被售出的商品，在Redis中存一个键。这个商品键对应的值是一个有序集合，就为同订单的其他商品在当前商品键对应的有序集合中的分数加1。

当一个订单成功支付时，我们为订单每个购买的商品存储一个有序集合，这个有序集合将记录一起购买的商品分数。

安装 `redis-py` 模块：

```
pip install redis==2.10.6
```

之后在 `settings.py` 里配置Redis：

```
REDIS_HOST = 'localhost'  
REDIS_PORT = 6379  
REDIS_DB = 1
```

这是用于建立和Redis服务通信的设置。在 `shop` 应用目录下新建 `recommender.py` 文件，添加下列代码：

```
import redis  
from django.conf import settings  
from .models import Product  
  
# 连接到Redis  
r = redis.StrictRedis(host=settings.REDIS_HOST, port=settings.REDIS_PORT, db=settings.REDIS_DB)
```

```
class Recommender:

    def get_product_key(self, product_id):
        return 'product:{}:purchased_with'.format(product_id)

    def products_bought(self, products):
        product_ids = [p.id for p in products]
        # 针对订单里的每一个商品，将其他商品在当前商品的有序集合中增加1
        for product_id in product_ids:
            for with_id in product_ids:
                if product_id != with_id:
                    r.zincrby(self.get_product_key(product_id), with_id, amount=1)
```

这个 `Recommender` 类用来存储订单购买时的相关信息和根据一个指定的对象获取相关的推荐。 `get_product_key()` 方法获取一个 `Product` 对象的id，然后创建对应的有序集合，其中的键看起来像这样： `product:[id]:purchased_with`。

`product_bought()` 方法接受属于同一个订单的 `Product` 对象的列表，然后做如下操作：

1. 获取所有 `Product` 对象的ID
2. 针对每一个ID遍历一次全部的ID，跳过内外循环ID相同的部分，这样就针对其中每个商品都遍历了与其一同购买的商品
3. 使用 `get_product_id()` 方法得到每个商品的Redis键名。例如针对ID为33的商品，返回的键名是 `product:33:purchased_with`，这个键将用于操作有序集合
4. 在该商品对应的有序序列将同一订单内的其他商品的分数增加1

我们现在有了一个保存商品相关信息的方法。还需要一个方法来从Redis中获得推荐的商品，继续编写 `Recommender` 类，增加 `suggest_products_for()` 方法：

```
class Recommender:
    # .....
    def suggest_products_for(self, products, max_results=6):
        product_ids = [p.id for p in products]
```

```
# 如果当前列表只有一个商品:
if len(product_ids) == 1:
    suggestions = r.zrange(self.get_product_key(product_ids[0]), 0, -1, desc=True)[:max_results]
else:
    # 生成一个临时的key, 用于存储临时的有序集合
    flat_ids = ''.join([str(id) for id in product_ids])
    tmp_key = 'tmp_{}'.format(flat_ids)
    # 对于多个商品, 取所有商品的键名构成keys列表
    keys = [self.get_product_key(id) for id in product_ids]
    # 合并有序集合到临时键
    r.zunionstore(tmp_key, keys)
    # 删除与当前列表内商品相同的键。
    r.zrem(tmp_key, *product_ids)
    # 获得排名结果
    suggestions = r.zrange(tmp_key, 0, -1, desc=True)[:max_results]
    # 删除临时键
    r.delete(tmp_key)
# 获取关联商品并通过相关性排序
suggested_products_ids = [int(id) for id in suggestions]
suggested_products = list(Product.objects.filter(id__in=suggested_products_ids))
suggested_products.sort(key=lambda x: suggested_products_ids.index(x.id))
return suggested_products
```

`suggest_products_for()` 方法接受两个参数：

- `products`：表示为哪些商品进行推荐，可以包含一个或多个商品
- `max_results`：整数值，表示最大推荐几个商品

在这个方法里我们做了如下的事情：

1. 获取所有 `Product` 对象的ID

2. 如果仅有一个商品，直接查询这个id对应的有序集合，按降序返回结果。为了实现查询，使用了Redis的 `ZRANGE` 命令。我们使用 `max_results` 属性指定返回的最大数量。
3. 如果商品数量多于1个，通过ID创建一个临时键名。
4. 通过Redis的 `ZUNIONSTORE` 命令合并所有商品的有序集合。 `ZUNIONSTORE` 合并所有的有序集合中相同键的分数，然后将新生成的有序集合存入临时键。关于该命令可以参考 <https://redis.io/commands/ZUNIONSTORE>。
5. 由于已经在当前购物车内的商品无需被推荐，因此使用 `ZREM` 命令从临时键的有序集合中删除与当前订单内商品id相同的键。
6. 从临时键中获取商品ID，使用 `ZRANGE` 命令按照分数排序，通过 `max_results` 控制返回数量，之后删除临时键。
7. 根据ID获取 `Product` 对象，然后按照与取出的ID相同的顺序进行排列。

为了更加实用，再给 `Recommender` 类添加一个清除推荐商品的方法：

```
class Recommender:  
    # .....  
    def clear_purchases(self):  
        for id in Product.objects.values_list('id', flat=True):  
            r.delete(self.get_product_key(id))
```

我们来测试一下推荐引擎是否正常工作。确保 `Product` 数据表中有一些商品信息，然后先启动Redis：

```
src/redis-server
```

通过 `python manage.py shell` 进入带有Django项目环境的shell中：

```
from shop.models import Product  
black_tea = Product.objects.get(translations__name='Black tea')  
red_tea = Product.objects.get(translations__name='Red tea')  
green_tea = Product.objects.get(translations__name='Green tea')  
tea_powder = Product.objects.get(translations__name='Tea powder')
```

之后增加一些测试购买数据：

```
from shop.recommender import Recommender
r = Recommender()
r.products_bought([black_tea, red_tea])
r.products_bought([black_tea, green_tea])
r.products_bought([red_tea, black_tea, tea_powder])
r.products_bought([green_tea, tea_powder])
r.products_bought([black_tea, tea_powder])
r.products_bought([red_tea, green_tea])
```

进行完上述操作后，我们实际为四个商品保存的有序集合是：

```
black_tea: red_tea (2), tea_powder (2), green_tea (1)
red_tea: black_tea (2), tea_powder (1), green_tea (1)
green_tea: black_tea (1), tea_powder (1), red_tea(1)
tea_powder: black_tea (2), red_tea (1), green_tea (1)
```

下边测试一下通过翻译字段获取推荐商品信息：

```
>>> from django.utils.translation import activate
>>> activate('en')
>>> r.suggest_products_for([black_tea])
[<Product: Tea powder>, <Product: Red tea>, <Product: Green tea>]
>>> r.suggest_products_for([red_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Red tea>]
>>> r.suggest_products_for([tea_powder])
[<Product: Black tea>, <Product: Red tea>, <Product: Green tea>]
```

如果看到商品是按照它们的分数进行降序排列的，就说明引擎工作正常了。再测试一下多个商品的推荐：

```
>>> r.suggest_products_for([black_tea, red_tea])
[<Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea, red_tea])
[<Product: Black tea>, <Product: Tea powder>]
>>> r.suggest_products_for([tea_powder, black_tea])
[<Product: Red tea>, <Product: Green tea>]
```

可以实际计算一下是否符合合并有序集合后的结果，例如针对第一条程序，`tea_powder` 的分数是2+1，`green_tea` 的分数是1+1等

测试之后说明我们的推荐算法正常工作，下一步就是将该功能集成到站点中，在商品详情页和购物车清单页进行展示。先修改 `shop` 应用的 `views.py` 文件中的 `product_detail` 视图：

```
from .recommender import Recommender

def product_detail(request, id, slug):
    language = request.LANGUAGE_CODE
    product = get_object_or_404(Product, id=id, translations__language_code=language, translations__slug=slug,
                               available=True)

    cart_product_form = CartAddProductForm()

    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)

    return render(request, 'shop/product/detail.html', {'product': product, 'cart_product_form': cart_product_form,
                                                       'recommended_products': recommended_products})
```

编辑 `shop/product/detail.html` 模板，增加下列代码到 `{{ product.description|linebreaks }}` 之后：

```
{% if recommended_products %}
<div class="recommendations">
    <h3>{% trans "People who bought this also bought" %}</h3>
    {% for p in recommended_products %}
        <div class="item">
            <a href="{{ p.get_absolute_url }}">
                
            </a>
            <p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
        </div>
    {% endfor %}
</div>
{% endif %}
```

然后运行站点，点击商品进入详情页，可以看到类似下图的商品推荐：



Tea powder

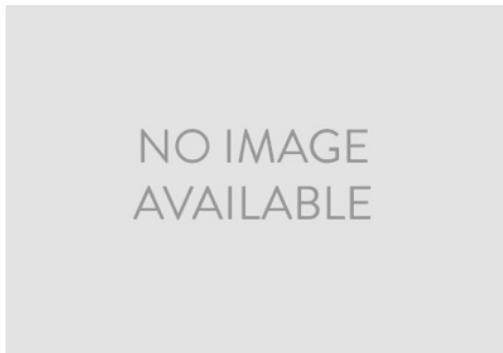
Tea

\$21.2

Quantity:

Add to cart

People who bought this also bought



Black tea



Red tea



Green tea

我们还需要在购物车详情页增加推荐功能，编辑 `cart` 应用的 `views.py` 文件中的 `cart_detail` 视图：

```
from shop.recommender import Recommender

def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(initial={'quantity': item['quantity'], 'update': True})
    coupon_apply_form = CouponApplyForm()

    r = Recommender()
    cart_products = [item['product'] for item in cart]
    recommended_products = r.suggest_products_for(cart_products, max_results=4)

    return render(request, 'cart/detail.html',
                  {'cart': cart, 'coupon_apply_form': coupon_apply_form, 'recommended_products': recommended_products})
```

然后修改对应的模板 `cart/detail.html`, 在 `</table>` 之后增加下列代码:

```
{% if recommended_products %}
    <div class="recommendations cart">
        <h3>{% trans "People who bought this also bought" %}</h3>
        {% for p in recommended_products %}
            <div class="item">
                <a href="{{ p.get_absolute_url }}>
                    
                </a>
                <p><a href="{{ p.get_absolute_url }}>{{ p.name }}</a></p>
            </div>
        {% endfor %}
    </div>
{% endif %}
```

译者注，由于上述内容使用了`{% trans %}`模板标签，不要忘记在页面上方加入`{% load i18n %}`，原书这里没有加，会导致报错。

在浏览器中打开 <http://127.0.0.1:8000/en/>。将一些商品加入购物车，然后至 <http://127.0.0.1:8000/en/cart/> 查看购物车详情，可以看到出现了推荐商品：

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	<input type="button" value="1"/> <input type="button" value="Up"/> <input type="button" value="Down"/> <input type="button" value="Update"/>	Remove	\$21.2	\$21.2
	Green tea	<input type="button" value="1"/> <input type="button" value="Up"/> <input type="button" value="Down"/> <input type="button" value="Update"/>	Remove	\$30	\$30
Total					\$51.20

People who bought this also bought

Apply a coupon:

NO IMAGE
AVAILABLE



Black tea

Red tea

Coupon:

Apply

Continue shopping

Checkout

现在我们就使用Redis配合Django完成了一个推荐系统。

译者注，原书其实并没有将功能写完。可以发现，目前的购买数据（调用 `Recommender` 类的 `products_bought()` 方法）是在我们测试的时候通过命令行添加的，而不是通过网站功能自动添加。按照一开始的分析，应该在付款成功的时候，更新Redis的数据。需要在 `payment` 应用的 `views.py` 文件中，在 `payment_process` 视图中付款响应成功，保存交易id和 `paid` 字段之后，发送PDF发票之前，添加如下代码：

```
from shop.recommender import Recommender

def payment_process(request):
    .....
    if request.method == "POST":
        .....
        if result.is_success:
            order.paid = True
            order.braintree_id = result.transaction.id
            order.save()

            # 更新Redis中本次购买的商品分数
            r = Recommender()
            order_items = [order_item.product for order_item in order.items.all()]
            r.products_bought(order_items)
```

总结

在这一章，学习了创建优惠码系统和国际化与本地化配置工作。还基于Redis创建了一个商品推荐系统。

在下一章，我们将创建一个新的项目：在线教育平台，里边将使用Django的CBV技术，还会创建一个内容管理系统。

第十章 创建在线教育平台

在上一章，我们为电商网站项目添加了国际化功能，还创建了优惠码和商品推荐系统。在本章，会建立一个新的项目：一个在线教育平台，并创内容管理系统CMS (Content Management System)。

本章的具体内容有

- 为模型建立 [fixtures](#)
- 使用模型的继承关系
- 创建自定义模型字段
- 使用CBV和 [mixin](#)
- 建立表单集formsets
- 管理用户组与权限
- 创建CMS

1 创建在线教育平台项目

我们最后一个项目就是这个在线教育平台。在这个项目中，我们将建立一个灵活的CMS系统，让讲师可以创建课程并且管理课程的内容。

为本项目建立一个虚拟环境，在终端输入如下命令：

```
mkdir env  
virtualenv env/educa  
source env/educa/bin/activate
```

在虚拟环境中安装Django与Pillow：

```
pip install Django==2.0.5  
pip install Pillow==5.1.0
```

之后新建项目 `educa`：

```
django-admin startproject educa
```

进入 `educa` 目录然后新建名为 `courses` 的应用：

```
cd educa  
django-admin startapp courses
```

编辑 `settings.py`，将应用激活并且放在最上边一行：

```
INSTALLED_APPS = [  
    'courses.apps.CoursesConfig',  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

之后的第一步工作，依然是定义数据模型。

2 创建课程模型

我们的在线教育平台会提供很多不同主题 (subject) 的课程，每一个课程会被划分为一定数量的课程章节 (module)，每个章节里边又有一定数量的内容 (content)。对于一个课程来说，里边使用到的内容类型很多，包含文本，文件，图片甚至视频，下边的是一个课程的例子：

```
Subject 1
Course 1
    Module 1
        Content 1 (image)
        Content 2 (text)
    Module 2
        Content 3 (text)
        Content 4 (file)
        Content 5 (video)
....
```

来建立课程的数据模型，编辑 courses 应用下的 `models.py` 文件：

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

class Subject(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ['title']

    def __str__(self):
        return self.title
```

```
class Course(models.Model):
    owner = models.ForeignKey(User, related_name='course_created', on_delete=models.CASCADE)
    subject = models.ForeignKey(Subject, related_name='courses', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    overview = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['-created']

    def __str__(self):
        return self.title

class Module(models.Model):
    course = models.ForeignKey(Course, related_name='modules', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)

    def __str__(self):
        return self.title
```

这是初始的 `Subject`，`Course` 和 `Module` 模型。`Course` 模型的字段如下：

1. `owner`：课程讲师，也是课程创建者
2. `subject`：课程的主体，外键关联到 `Subject` 模型
3. `title`：课程名称
4. `slug`：课程slug名称，将来用在生成URL
5. `overview`：课程简介
6. `created`：课程建立时间，生成数据行时候自动填充

`Module` 从属于一个具体的课程，所以 `Module` 模型中有一个外键连接到 `Course` 模型。

之后进行数据迁移，不再赘述。

2.1 在管理后台注册上述模型

编辑 `course` 应用的 `admin.py` 文件，添加如下代码：

```
from django.contrib import admin
from .models import Subject, Course, Module

@admin.register(Subject)
class SubjectAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug']
    prepopulated_fields = {'slug': ('title',)}


class ModuleInline(admin.StackedInline):
    model = Module


@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ['title', 'subject', 'created']
    list_filter = ['created', 'subject']
    search_fields = ['title', 'overview']
    prepopulated_fields = {'slug': ('title',)}
    inlines = [ModuleInline]
```

这就注册好了应用里的全部模型，记住 `@admin.register()` 用于将模型注册到管理后台中。

2.2 使用fixture为模型提供初始化数据

有些时候，需要使用原始数据来直接填充数据库，这比每次建立项目之后手工录入原始数据要方便很多。Django提供了fixtures（可以理解为一个预先格式化好的数据文件）功能，可以方便的从数据库中读取数据到fixture中，或者把fixture中的数据导入至数据库。

Django支持使用JSON，XML或YAML等格式来使用fixture。来建立一个包含一些初始化的Subject对象的fixture：

首先创建超级用户：

```
python manage.py createsuperuser
```

之后运行站点：

```
python manage.py runserver
```

进入<http://127.0.0.1:8000/admin/courses/subject/> 可以看到如下界面（需要先输入一些数据）：

Select subject to change

ADD SUBJECT +

Action: ----- 0 of 4 selected

<input type="checkbox"/>	TITLE	SLUG
<input type="checkbox"/>	Mathematics	mathematics
<input type="checkbox"/>	Music	music
<input type="checkbox"/>	Physics	physics
<input type="checkbox"/>	Programming	programming

4 subjects

在shell中执行如下命令：

```
python manage.py dumpdata courses --indent=2
```

可以看到如下输出：

```
[  
 {  
     "model": "courses.subject",  
     "pk": 1,  
     "fields": {  
         "title": "Mathematics",  
         "slug": "mathematics"  
     }  
 },
```

```
{  
    "model": "courses.subject",  
    "pk": 2,  
    "fields": {  
        "title": "Music",  
        "slug": "music"  
    }  
},  
{  
    "model": "courses.subject",  
    "pk": 3,  
    "fields": {  
        "title": "Physics",  
        "slug": "physics"  
    }  
},  
{  
    "model": "courses.subject",  
    "pk": 4,  
    "fields": {  
        "title": "Programming",  
        "slug": "programming"  
    }  
}  
]
```

`dumpdata` 命令采取默认的JSON格式，将 `Course` 类中的数据序列化并且输出。JSON中包含了模型的名称，主键，字段与对应的值。设置了`indent=2`是表示每行的缩进。

可以通过向命令行提供应用名和模块名，例如 `app.Model`，让数据直接输出到这个模型中；还可以通过 `--format` 参数控制输出的数据格式，默认是使用JSON格式。还可以通过 `--output` 参数指定输出到具体文件。

对于 `dumpdata` 的详细参数，可以使用命令 `python manage.py dumpdata --help` 查看。

使用如下命令把这个dump结果保存到 `courses` 应用的一个 `fixture/` 目录中：

```
mkdir courses/fixtures  
python manage.py dumpdata courses --indent=2 --output=courses/fixtures/subjects.json
```

译者注，原书写成了在 `orders` 应用下的 `fixture/` 目录，显然是将应用名写错了。

现在进入管理后台，将 `Subject` 表中的数据全部删除，之后执行下列语句，从fixture中加载数据：

```
python manage.py loaddata subjects.json
```

可以发现，所有删除的数据都都回来了。

默认情况下Django会到每个应用里的 `fixtures/` 目录内寻找指定的文件名，也可以在 `settings.py` 中设置 `FIXTURE_DIRS` 来告诉Django到哪里寻找fixture。

fixture除了初始化数据库之外，还可以方便的为应用提供测试数据。

有关fixture的详情可以查看 <https://docs.djangoproject.com/en/2.0/topics/testing/tools/#fixture-loading>。

如果在进行数据模型移植的时候就加载fixture生成初始数据，可以查看
<https://docs.djangoproject.com/en/2.0/topics/migrations/#data-migrations>。

3 创建不同类型内容的模型

在课程中会向用户提供不同类型的内容，包括文字，图片，文件和视频等。我们必须采用一个能够存储各种文件类型的通用模型。在第六章中，我们学会了使用通用关系来创建与项目内任何一个数据模型的关系。这里我们建立一个

Content模型，用于存放章节中的内容，定义一个通用关系来连接任何类型的内容。

编辑 courses 应用的 `models.py` 文件，增加下列内容：

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

之后在文件末尾添加下列内容：

```
class Content(models.Model):
    module = models.ForeignKey(Module, related_name='contents', on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
```

这就是 Content 模型，设置外键关联到了 Module 模型，同时设置了与 ContentType 模型的通用关联关系，可以从获取任意模型的内容。复习一下创建通用关系的所需的三个字的：

1. `content_type`：一个外键用于关联到 ContentType 模型。
2. `object_id`：对象的id，使用 `PositiveIntegerField` 字段。
3. `item`：通用关联关系字段，通过合并上两个字段来进行关联。

`content_type`, `object_id` 两个字段会实际生成在数据库中，`item` 字段的关系是ORM引擎构建的，不真正被写进数据库中。

下一步的工作是建立每种具体内容类型的数据库，这些数据库有一些相同的字段用于标识基本信息，也有不同的字段存放该模型独特的信息。

3.1 模型的继承

Django支持数据模型之间的继承关系，这和Python程序的类继承关系很相似，Django提供了以下三种继承的方式：

1. **Abstarct model**: 接口模型继承，用于方便的向不同的数据模型中添加相同的信息，这种继承方式中的基类不会在数据库中建立数据表，子类会建立数据表。
2. **Multi-table model inheritance**: 多表模型继承，在继承关系中的每个表都被认为是一个完整的模型时采用此方法，继承关系中的每一个表都会实际在数据库中创建数据表。
3. **Proxy models**: 代理模型继承，在继承的时候需要改变模型的行为时使用，例如加入额外的方法，修改默认的模型管理器或使用新的Meta类设置，此种继承不会在数据库中创建数据表。

让我们详细看一下这三种方式。

3.1.1 Abstract models 抽象基类继承

接口模型本质上是一个基类类，其中定义了所有需要包含在子模型中的字段。Django不会为接口模型创建任何数据库中的数据表。继承接口模型的子模型必须将这些字段完善，每一个子模型会创建数据表，表中的字段包括继承自接口模型的字段和子模型中自定义的字段。

为了标记一个模型为接口模型，在其Meta设置中，必须设置 `abstract = True`，django就会认为该模型是一个接口模型，不会创建数据表。子模型只需要继承该模型即可。

下边的例子是如何建立一个接口模型 `Content` 和子模型 `Text`：

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
```

```
created = models.DateTimeField(auto_now_add=True)

class Meta:
    abstract = True

class Text(BaseContent):
    body = models.TextField()
```

在这个例子中，实际在数据库中创建的是 `Text` 类对应的数据表，包含 `title`，`created` 和 `body` 字段。

3.1.2 Multi-table model inheritance 多表继承

多表继承关系中的每一个表都是完整的数据模型。对于继承关系，Django会自动在子模型中创建一个一对一关系的外键连接到父模型。

要使用该种继承方式，必须继承一个已经存在的模型，django会把父模型和子模型都写入数据库，下边是一个例子：

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django会将两张表都写入数据库，`Text` 表中除了 `body` 字段，还有一个一对一的外键关联到 `BaseContent` 表。

3.1.3 Proxy models 代理模型

代理模型用于改变类的行为，例如增加额外的方法或者不同的Meta设置。父模型和子模型操作一张相同的数据表。

Meta类中指定 proxy=True 就可以建立一个代理模型。

下边是一个创建代理模型的例子：

```
from django.db import models
from django.utils import timezone

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class OrderedContent(BaseContent):
    class Meta:
        proxy = True
        ordering = ['created']

    def created_delta(self):
        return timezone.now() - self.created
```

这里我们定义了一个 OrderedContent 模型，作为 BaseContent 模型的一个代理模型。这个代理模型提供了排序设置和一个新方法 created_delta()。OrderedContent 和 BaseContent 都是操作由 BaseContent 模型生成的数据表，但新增的排序和方法，只有通过 OrderedContent 对象才能使用。

这种方法就类似于经典的Python类继承方式。

3.2 创建内容的模型

courses 应用中的 Content 模型现在有着通用关系，可以取得任何模型的数据。我们要为每种内容建立不同的模型。所有的内容模型都有相同的字段也有不同的字段，这里就采取接口模型继承的方式来建立内容模型：

编辑 courses 应用中的 `models.py` 文件，添加下列代码：

```
class ItemBase(models.Model):
    owner = models.ForeignKey(User, related_name='%(class)s_related', on_delete=models.CASCADE)
    title = models.CharField(max_length=250)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True

    def __str__(self):
        return self.title


class Text(ItemBase):
    content = models.TextField()


class File(ItemBase):
    file = models.FileField(upload_to='files')


class Image(ItemBase):
    file = models.FileField(upload_to='images')


class Video(ItemBase):
    url = models.URLField()
```

在这段代码中，首先建立了一个接口模型 `ItemBase`，其中有四个字段，然后在 `Meta` 中设置了 `abstract=True` 以使该类为接口类。该类中定义了 `owner`, `title`, `created`, `updated` 四个字段，将在所有的内容模型中使用。`owner` 是关联到用户的外键，存放当前内容的创建者。由于这是一个基类，必须要为不同的模型指定不同的 `related_name`。Django 允许

在 `related_name` 属性中使用类似 `%(class)s` 之类的占位符。设置之后，`related_name` 就会动态生成。这里我们使用了 `'%(class)s_related'`，最后实际的名称是 `text_related`, `file_related`, `image_related` 和 `video_retaled`。

我们定义了四种类型的内容模型，均继承 `ItemBase` 抽象基类：

- `Text`：存储教学文本
- `File`：存储分发给用户的文件，比如PDF文件等教学资料
- `Image`：存储图片
- `Video`：存储视频，定义了一个 `URLField` 字段存储视频的路径。

每个子模型中都包含 `ItemBase` 中定义的字段。Django会针对四个子模型分别在数据库中创建数据表，但 `ItemBase` 类不会被写入数据库。

继续编辑 `courses` 应用的 `models.py` 文件，由于四个子模型的类名已经确定了，需要修改 `Content` 模型让其对应到这四个模型上，修改 `content_type` 字段如下：

```
class Content(models.Model):  
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE,  
                                    limit_choices_to={'model__in': ('text', 'file', 'image', 'video')})
```

这里使用了 `limit_choices_to` 属性，以使 `ContentType` 对象限于这四个模型中。如此定义之后，在查询数据库的时候还能够使用 `filter` 的参数例如 `model__in='text'` 来检索具体某个模型的对象。

建立好所有模型之后，执行数据迁移程序，不再赘述。

现在就已经建立了本项目所需要的基本数据表及其结构。然而我们的模型中还缺少一些内容：课程和课程的内容是按照一定顺序排列的，但用户建立课程和上传内容的时候未必是线性的，我们需要一个排序字段，通过字段可以把课程，章节和内容进行排序。

3.3 创建自定义字段

Django内置了很完善的模型字段供方便快捷的建立数据模型。然而依然有无法满足用户需求的地方，我们也可以自定义模型字段，来存储个性化的内容，或者修改内置字段的行为。

我们需要一个字段存储课程和内容组织的顺序。通常用于确定顺序可以方便的采用内置的 `PositiveIntegerField` 字段，采用一个正整数就可以方便的标记数据的顺序。这里我们继承 `PositiveIntegerField` 字段，然后增加额外的行为来完成我们的自定义排序。

我们要给自定义字段增加增加如下两个功能：

- 如果序号没有给出，则自动分配一个序号。当内容和课程表中存进一个新的数据对象的时候，如果用户给出了具体的序号，就将该序号存入到排序字段中。如果用户没有给出序号，应该自动按照最大的序号再加1。例如如果已经存在两个数据对象的序号是1和2，如果用户存入第三个数据但未给出序号，则应该自动给新数据对象分配序号3。
- 根据其他相关的内容排序：章节应该按照课程排序，而内容应该按照章节排序

在 `courses` 应用下建立 `fields.py` 文件，添加如下代码：

```
from django.db import models
from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):
    def __init__(self, for_fields=None, *args, **kwargs):
        self.for_fields = for_fields
        super(OrderField, self).__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):
        if getattr(model_instance, self.attname) is None:
            # 如果没有值，查询自己所在表的全部内容，找到最后一条字段，设置临时变量value = 最后字段的序号+1
            pass
```

```
try:  
    qs = self.model.objects.all()  
    if self.for_fields:  
        # 存在for_fields参数，通过该参数取对应的数据行  
        query = {field: getattr(model_instance, field) for field in self.for_fields}  
        qs = qs.filter(**query)  
    # 取最后一个数据对象的序号  
    last_item = qs.latest(self.attname)  
    value = last_item.order + 1  
except ObjectDoesNotExist:  
    value = 0  
setattr(model_instance, self.attname, value)  
return value  
else:  
    return super(OrderField, self).pre_save(model_instance, add)
```

这是自定义的字段类 `OrderField`，继承了内置的 `PositiveIntegerField` 类，还增加了额外的参数 `for_fields` 指定按照哪一个字段的顺序进行计算。

我们重写了 `pre_save()` 方法，这个方法是在将字段的值实际存入到数据库之前执行的。在这个方法里，执行了如下逻辑：

1. 检查当前字段是否存在值，`self.attname` 表示该字段对应的属性名，也就是字段属性。如果属性名是 `None`，说明用户没有设置序号。则按照以下逻辑进行计算：
 1. 建立一个QuerySet，查询这个字段所在的模型的全部数据行。访问字段所在的模型使用了 `self.model`
 2. 通过用户给出的 `for_fields` 参数，把上一步的QuerySet用其中的字段拆解之后过滤，这样就可以取得具体的用于计算序号的参考数据行。
 3. 然后从过滤过的QuerySet中使用 `last_item = qs.latest(self.attname)` 方法取出最新一行数据对应的序号。如果取不到，说明自己是第一行。就将临时变量设置为0
 4. 如果能够取到，就把取到的序号+1然后赋给 `value` 临时变量

5. 然后通过 `setattr()` 将临时变量 `value` 添加为字段名属性对应的值
2. 如果当前的字段已经有值，说明用户传入了序号，不需要做任何工作。

在自定义字段时，一定不要硬编码将内容写死，也需要像内置字段一样注意通用性。

关于自定义字段可以看 <https://docs.djangoproject.com/en/2.0/howto/custom-model-fields/>。

3.4 将自定义字段加入到模型中

建立好自定义的字段类之后，需要在各个模型中设置该字段，编辑 `courses` 应用的 `models.py` 文件，添加如下内容：

```
from .fields import OrderField

class Module(models.Model):
    # .....
    order = OrderField(for_fields=['course'], blank=True)
```

我们给自定义的排序字段起名叫 `order`，然后通过设置 `for_fields=['course']`，让该字段按照课程来排序。这意味着如果最新的某个 `Course` 对象关联的 `module` 对象的序号是3，为该 `Course` 对象其新增一个关联的 `module` 对象的序号就是4。

然后编辑 `Module` 模型的 `__str__()` 方法：

```
class Module(models.Model):
    def __str__(self):
        return '{}. {}'.format(self.order, self.title)
```

章节对应的内容也必须有序号，现在为 `Content` 模型也增加上 `OrderField` 类型的字段：

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

这样就指定了 `Content` 对象的序号根据其对应的 `module` 字段来排序，最后为两个模型添加默认的排序，为两个模型添加如下 `Meta` 类：

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']

class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

最终的 `Module` 和 `Content` 模型应该是这样：

```
class Module(models.Model):
    course = models.ForeignKey(Course, related_name='modules', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    order = OrderField(for_fields=['course'], blank=True)

    def __str__(self):
        return '{}. {}'.format(self.order, self.title)

    class Meta:
        ordering = ['order']
```

```
class Content(models.Model):
    module = models.ForeignKey(Module, related_name='contents', on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE,
                                    limit_choices_to={'model__in': ('text', 'video', 'image', 'file')})
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
    order = OrderField(for_fields=['module'], blank=True)

    class Meta:
        ordering = ['order']
```

模型修改好了，执行迁移命令 `python manage.py makemigrations courses`，可以发现提示如下：

```
Tracking file by folder pattern: migrations
You are trying to add a non-nullable field 'order' to content without a default; we can't do that (the database needs it).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

这个提示的意思是说不能添加值为 `null` 的新字段 `order` 到数据表中，必须提供一个默认值。如果字段有 `null=True` 属性，就不会提示此问题。我们有两个选择，选项1是输入一个默认值，作为所有已经存在的数据行该字段的值，选项2是放弃这次操作，在模型中为该字段添加 `default=xx` 属性来设置默认值。

这里我们输入1并按回车键，看到如下提示：

```
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
Type 'exit' to exit this prompt
```

系统提示我们输入值，输入0然后按回车，之后Django又会对 `Module` 模型询问同样的问题，依然选择第一项然后输入0。之后可以看到：

```
Migrations for 'courses':  
  courses\migrations\0003_auto_20181001_1344.py  
    - Change Meta options on content  
    - Change Meta options on module  
    - Add field order to content  
    - Add field order to module
```

表示成功，之后执行 `python manage.py migrate`。然后我们来测试一下排序，打开系统命令行窗口：

```
python manage.py shell
```

创建一个新课程：

```
>>> from django.contrib.auth.models import User  
>>> from courses.models import Subject, Course, Module  
>>> user = User.objects.last()  
>>> subject = Subject.objects.last()  
>>> c1 = Course.objects.create(subject=subject, owner=user, title='Course 1', slug='course1')
```

添加了一个新课程，现在我们来为新课程添加对应的章节，来看看是如何自动排序的。

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')  
>>> m1.order  
0
```

可以看到 `m1` 对象的序号字段的值被设置为0，因为这是针对课程的第一个 `Module` 对象，下边再增加一个 `Module` 对象：

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

可以看到随后增加的 `Module` 对象的序号自动被设置成了1，这次我们创建第三个对象，指定序号为5：

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

如果指定了序号，则序号就会是指定的数字。为了继续试验，再增加一个对象，不给出序号参数：

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

可以看到，序号会根据最后保存的数据继续增加1。`OrderField` 字段无法保证序号一定连续，但可以保证添加的内容的序号一定是从小到大排列的。

继续试验，我们再增加第二个课程，然后第二个课程添加一个 `Module` 对象：

```
>>> c2 = Course.objects.create(subject=subject, title='Course 2', slug='course2', owner=user)
>>> m5 = Module.objects.create(course=c2, title='Module 1')
>>> m5.order
0
```

可以看到序号又从0开始，该字段在生成序号的时候只会考虑同属于同一个外键字段下边的对象，第二个课程的第一个 `Module` 对象的序号又从0开始，正是由于 `order` 字段设置了 `for_fields=['course']` 所致。

祝贺你成功创建了第一个自定义字段。

4 创建内容管理系统CMS

在创建好了完整的数据模型之后，需要创建内容管理系统。内容管理系统能够让讲师创建课程然后管理课程资源。

我们的内容管理系统需要如下几个功能：

- 登录功能
- 列出讲师的全部课程
- 新建，编辑和删除课程
- 为课程增加章节
- 为章节增加不同的内容

4.1 为站点增加用户验证系统

这里我们使用Django内置验证模块为项目增加用户验证功能、所有的讲师和学生都是 `User` 模型的实例，都可以通过 `django.contrib.auth` 来管理用户。

编辑 `educa` 项目的根 `urls.py` 文件，添加连接到内置验证函数 `login` 和 `logout` 的路由：

```
from django.contrib import admin
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
```

```
    path('admin/', admin.site.urls),  
]
```

4.2 创建用户验证模板

在 `courses` 应用下建立如下目录和文件：

```
templates/  
    base.html  
registration/  
    login.html  
    logged_out.html
```

在编写登录登出和其他模板之前，先来编辑 `base.html` 作为母版，在其中添加如下内容：

```
{% load staticfiles %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8"/>  
    <title>{% block title %}Educa{% endblock %}</title>  
    <link href="{% static "css/base.css" %}" rel="stylesheet">  
</head>  
<body>  
<div id="header">  
    <a href="/" class="logo">Educa</a>  
    <ul class="menu">  
        {% if request.user.is_authenticated %}  
            <li><a href="{% url "logout" %}">Sign out</a></li>  
        {% else %}  
            <li><a href="{% url "login" %}">Sign in</a></li>  
        {% endif %}
```

```
</ul>
</div>
<div id="content">
    {% block content %}
    {% endblock %}
</div>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script>
    $(document).ready(function () {
        {% block domready %}
        {% endblock %}
    });
</script>
</body>
</html>
```

译者注：为了使用方便，这里将作者原书存放jQuery文件的的Google CDN换成了国内BootCDN的地址。下边很多地方都作类似处理。

在母版中，定义了几个块：

1. `title`：用于HEAD标签的TITLE标签使用
2. `content`：页面主体内容
3. `domready`：包含jQuery的 `$document.ready()` 代码，为页面DOM加载完成后执行的JS代码

这里还用到了CSS文件，在 `courses` 应用中建立 `static/css/` 目录并将随书源代码中的CSS文件复制过来。

有了母版之后，编辑 `registration/login.html`：

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

{% block title %}Log-in{% endblock %}
```

```
{% block content %}
    <h1>Log-in</h1>
    <div class="module">
        {% if form.errors %}
            <p>Your username and password didn't match. Please try again.</p>
        {% else %}
            <p>Please, use the following form to log-in:</p>
        {% endif %}
        <div class="login-form">
            <form action="{% url 'login' %}" method="post">
                {{ form.as_p }}
                {% csrf_token %}
                <input type="hidden" name="next" value="{{ next }}"/>
                <p><input type="submit" value="Log-in"></p>
            </form>
        </div>
    </div>
{% endblock %}
```

这是Django标准的用于内置 `login` 视图的模板。继续编写同目录下的 `logged_out.html` :

```
{% extends "base.html" %}
{% block title %}Logged out{% endblock %}
{% block content %}
    <h1>Logged out</h1>
    <div class="module">
        <p>You have been successfully logged out.
            You can <a href="{% url "login" %}">log-in again</a>.</p>
    </div>
{% endblock %}
```

这是用户登出之后展示的页面。启动站点，到 <http://127.0.0.1:8000/accounts/login/> 查看，页面如下：

Log-in

Please, use the following form to log-in:

Username:

Password:

LOG-IN

4.3 创建CBV

我们将来创建增加，编辑和删除课程的功能。这次使用基于类的视图进行编写，编辑 courses 应用的 `views.py` 文件：

```
from django.views.generic.list import ListView
from .models import Course

class ManageCourseListView(ListView):
```

```
model = Course
template_name = 'courses/manage/course/list.html'

def get_queryset(self):
    qs = super(ManageCourseListView, self).get_queryset()
    return qs.filter(owner=self.request.user)
```

这是 `ManageCourseListView` 视图，继承自内置的 `ListView` 视图。为了避免用户操作不属于该用户的内容，重写了 `get_queryset()` 方法以取得当前用户相关的课程，在其他增删改内容的视图中，我们同样需要重写 `get_queryset()` 方法。

如果想为一些CBV提供特定的功能和行为（而不是在每个类内重写某个方法），可以使用 `mixins`。

4.4 在CBV中使用mixin

对类来说，`Mixin` 是一种特殊的多继承方式。通过Mixin可以给类附加一系列功能，自定义类的行为。有两种情况一般都会使用mixins：

- 给类提供一系列可选的特性
- 在很多类中实现一种特定的功能

Django为CBV提供了一系列mixins用来增强CBV的功能，具体可以看

<https://docs.djangoproject.com/en/2.0/topics/class-based-views/mixins/>。

我们准备创建一个mixin，包含一个通用的方法，用于我们与课程相关的CBV中。修改 `courses` 应用的 `views.py` 文件，修改成下面这样：

```
from django.urls import reverse_lazy
from django.views.generic.list import ListView
```

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView

from .models import Course

class OwnerMixin:
    def get_queryset(self):
        qs = super(OwnerMixin, self).get_queryset()
        return qs.filter(owner=self.request.user)

class OwnerEditMixin:
    def form_valid(self, form):
        form.instance.owner = self.request.user
        return super(OwnerEditMixin, self).form_valid(form)

class OwnerCourseMixin(OwnerMixin):
    model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
    template_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):
    template_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
    pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
    pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
```

在上述代码中，创建了两个 mixin类 `OwnerMixin` 和 `OwnerEditMixin`，将这些 mixins和Django内置的 `ListView`，`CreateView`，`UpdateView`，`DeleteView`一起使用。

这里创建的 mixin类解释如下：

`OwnerMixin` 实现了下列方法：

- `get_queryset()`：这个方法是内置视图用于获取QuerySet的方法，我们的 mixin重写了该方法，让该方法只返回与当前用户 `request.user` 关联的查询结果。

`OwnerEditMixin` 实现下列方法：

- `form_valid()`：所有使用了Django内置的 `ModelFormMixin` 的视图，都具有该方法。这个方法具体工作机制是：如 `CreateView` 和 `UpdateView` 这种需要处理表单数据的视图，当表单验证通过时，就会执行 `form_valid()` 方法。该方法的默认行为是保存数据对象，然后重定向到一个保存成功的URL。这里重写了该方法，自动给当前的数据对象设置上 `owner` 属性对应的用户对象，这样我们就在保存过程中自动附加上用户信息。

`OwnerMixin` 可以用于任何带有owner字段的模型。

我们还定义了继承自 `OwnerMixin` 的 `OwnerCourseMixin`，然后指定了下列参数：

- `model`：进行查询的模型，可以被所有CBV使用。

定义了 `OwnerCourseEditMixin`，具有下列属性：

- `fields`：指定 `CreateView` 和 `UpdateView` 等处理表单的视图在建立表单对象的时候使用的字段。
- `success_url`： `CreateView` 和 `UpdateView` 视图在表单提交成功后的跳转地址，这里定义了一个URL名称 `manage_course_list`，稍后会在路由中配置该名称

最后我们创建了如下几个 `OwnerCourseMixin` 的子类

- `ManageCourseListView`：展示当前用户创建的课程，继承 `OwnerCourseMixin` 和 `ListView`
- `CourseCreateView`：使用一个模型表单创建一个新的 `Course` 对象，使用 `OwnerCourseEditMixin` 定义的字段，并且继承内置的 `CreateView`
- `CourseUpdateView`：允许编辑和修改已经存在的 `Course` 对象，继承 `OwnerCourseEditMixin` 和 `UpdateView`
- `CourseDeleteView`：继承 `OwnerCourseMixin` 和内置的 `DeleteView`，定义了成功删除对象之后跳转的 `success_url`

译者注：使用 mixin 时必须了解 Python 3 对于类继承的 MRO 查找顺序，想要确保 mixin 中重写的方法生效，必须在继承时把 mixin 放在内置 CBV 的左侧。对于刚开始使用 mixin 的读者，可以使用 Pycharm 专业版点击右键--Diagrams--Show Diagrams--Python Class Diagram 查看当前文件的类图来了解继承关系。

4.5 使用用户组和权限

我们已经创建好了所有管理课程的视图。目前任何已登录用户都可以访问这些视图。但是我们要限制课程相关的内容只能由创建者进行操作，Django 的内置用户验证模块提供了权限系统，用于向用户和用户组分派权限。我们准备针对讲师建立一个用户组，然后给这个用户组内用户授予增删改课程的权限。

启动站点，进入 <http://127.0.0.1:8000/admin/auth/group/add/>，然后创建一个新的 `Group`，名字叫做 `Instructors`，然后为其选择除了 `Subject` 模型之外，所有与 `courses` 应用相关的权限。如下图所示：

Add group

The screenshot shows the 'Add group' page in the Django Admin interface. The 'Name' field is filled with 'Instructors'. The 'Permissions' section is expanded, showing two lists: 'Available permissions' and 'Chosen permissions'. The 'Available permissions' list contains 14 items, mostly starting with 'auth' or 'admin'. The 'Chosen permissions' list contains 10 items, mostly starting with 'courses' or 'content'. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and a large blue 'SAVE' button.

Name:	Instructors																																																		
Permissions:	<table border="1"><thead><tr><th colspan="2">Available permissions</th></tr></thead><tbody><tr><td>Filter</td><td></td></tr><tr><td>admin log entry Can add log entry</td><td></td></tr><tr><td>admin log entry Can change log entry</td><td></td></tr><tr><td>admin log entry Can delete log entry</td><td></td></tr><tr><td>auth group Can add group</td><td></td></tr><tr><td>auth group Can change group</td><td></td></tr><tr><td>auth group Can delete group</td><td></td></tr><tr><td>auth permission Can add permission</td><td></td></tr><tr><td>auth permission Can change permission</td><td></td></tr><tr><td>auth permission Can delete permission</td><td></td></tr><tr><td>auth user Can add user</td><td></td></tr><tr><td>auth user Can change user</td><td></td></tr></tbody></table> <table border="1"><thead><tr><th colspan="2">Chosen permissions</th></tr></thead><tbody><tr><td>courses content Can add content</td><td></td></tr><tr><td>courses content Can change content</td><td></td></tr><tr><td>courses content Can delete content</td><td></td></tr><tr><td>courses course Can add course</td><td></td></tr><tr><td>courses course Can change course</td><td></td></tr><tr><td>courses course Can delete course</td><td></td></tr><tr><td>courses file Can add file</td><td></td></tr><tr><td>courses file Can change file</td><td></td></tr><tr><td>courses file Can delete file</td><td></td></tr><tr><td>courses image Can add image</td><td></td></tr><tr><td>courses image Can change image</td><td></td></tr></tbody></table> <p>Choose all</p> <p>Hold down "Control", or "Command" on a Mac, to select more than one.</p> <p>Save and add another Save and continue editing SAVE</p>	Available permissions		Filter		admin log entry Can add log entry		admin log entry Can change log entry		admin log entry Can delete log entry		auth group Can add group		auth group Can change group		auth group Can delete group		auth permission Can add permission		auth permission Can change permission		auth permission Can delete permission		auth user Can add user		auth user Can change user		Chosen permissions		courses content Can add content		courses content Can change content		courses content Can delete content		courses course Can add course		courses course Can change course		courses course Can delete course		courses file Can add file		courses file Can change file		courses file Can delete file		courses image Can add image		courses image Can change image	
Available permissions																																																			
Filter																																																			
admin log entry Can add log entry																																																			
admin log entry Can change log entry																																																			
admin log entry Can delete log entry																																																			
auth group Can add group																																																			
auth group Can change group																																																			
auth group Can delete group																																																			
auth permission Can add permission																																																			
auth permission Can change permission																																																			
auth permission Can delete permission																																																			
auth user Can add user																																																			
auth user Can change user																																																			
Chosen permissions																																																			
courses content Can add content																																																			
courses content Can change content																																																			
courses content Can delete content																																																			
courses course Can add course																																																			
courses course Can change course																																																			
courses course Can delete course																																																			
courses file Can add file																																																			
courses file Can change file																																																			
courses file Can delete file																																																			
courses image Can add image																																																			
courses image Can change image																																																			

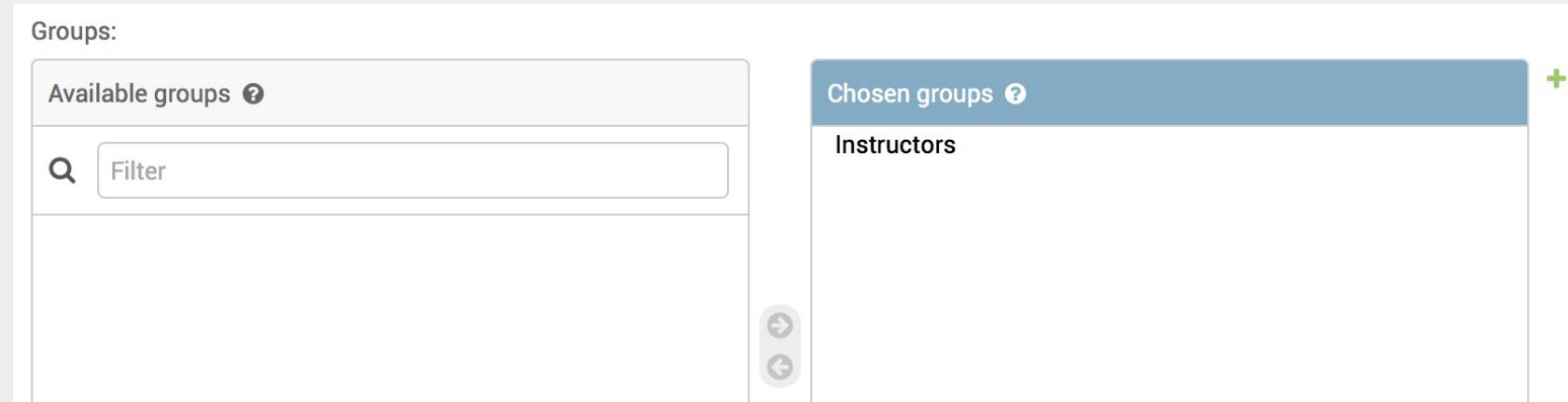
可以看到，对于每个应用中的每个模型，都有三个权限 **can add** , **can change** , **can delete** 。选好之后，点击SAVE按钮保存。

译者注：如果读者使用2.1或者更新版本的Django，权限还包括can view。

Django会为项目内的模型自动设置权限，如果需要的话，也可以编写自定义权限。具体可以查看

<https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#custom-permissions>。

打开 <http://127.0.0.1:8000/admin/auth/user/add/> 添加一个新用户，然后设置其为 `Instructors` 用户组的成员，如下图所示：



默认情况下，用户会继承其用户组设置的权限，也可以自行选择任意的其他单独权限。如果用户的 `is_superuser` 属性被设置为 `True`，则自动具有全部权限。

4.5.1 限制访问CBV

我们将限制用户对于视图的访问，使具有对应权限的用户才能进行增删改 `Course` 对象的操作。这里使用两个 `django.contrib.auth` 提供的mixins来限制对视图的访问：

1. `LoginRequiredMixin`: 与 `@login_required` 装饰器功能一样
2. `PermissionRequiredMixin`: 允许具有特定权限的用户访问该视图，超级用户具备所有权限。

编辑 `courses` 应用的 `views.py` 文件，新增如下导入代码：

```
from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin
```

让 `OwnerCourseMixin` 类继承 `LoginRequiredMixin` 类，然后添加属性：

```
class OwnerCourseMixin(OwnerMixin, LoginRequiredMixin):
    model = Course
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

然后为几个视图都配置一个 `permission_required` 属性：

```
class CourseCreateView(PermissionRequiredMixin, OwnerCourseEditMixin, CreateView):
    permission_required = 'courses.add_course'

class CourseUpdateView(PermissionRequiredMixin, OwnerCourseEditMixin, UpdateView):
    permission_required = 'courses.change_course'

class CourseDeleteView(PermissionRequiredMixin, OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
    permission_required = 'courses.delete_course'
```

`PermissionRequiredMixin` 会检查用户是否具备在 `permission_required` 参数里指定的权限。现在视图就只能供指定权限的用户使用了。

视图编写完毕之后，为视图配置路由，先在 `courses` 应用中新建 `urls.py` 文件，添加下列代码：

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

urlpatterns = [
    path('mine/', views.ManageCourseListView.as_view(), name='manage_course_list'),
    path('create/', views.CourseCreateView.as_view(), name='course_create'),
```

```
    path('<pk>/edit/', views.CourseUpdateView.as_view(), name='course_edit'),
    path('<pk>/delete/', views.CourseDeleteView.as_view(), name='course_delete'),
]
```

再来配置项目的根路由，将 `courses` 应用的路由作为二级路由：

```
from django.urls import path, include
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('admin/', admin.site.urls),
    path('course/', include('courses.urls')),
]
```

然后需要为视图创建模板，在 `courses` 应用的 `templates/` 目录下新建如下目录和文件：

```
courses/
  manage/
    course/
      list.html
      form.html
      delete.html
```

编辑其中的 `courses/manage/course/list.html`，添加下列代码：

```
{% extends "base.html" %}
{% block title %}My courses{% endblock %}
{% block content %}
<h1>My courses</h1>
```

```
<div class="module">
    {% for course in object_list %}
        <div class="course-info">
            <h3>{{ course.title }}</h3>
            <p>
                <a href="{% url "course_edit" course.id %}">Edit</a>
                <a href="{% url "course_delete" course.id %}">Delete</a>
            </p>
        </div>
    {% empty %}
    <p>You haven't created any courses yet.</p>
    {% endfor %}
    <p>
        <a href="{% url "course_create" %}" class="button">Create new
            course</a>
    </p>
</div>
{% endblock %}
```

这是供 `ManageCourseListView` 使用的视图。在这个视图里列出了所有的课程，然后生成对应的编辑和删除功能链接。

启动站点，到 <http://127.0.0.1:8000/accounts/login/?next=/course/mine/>，用一个在 `Instructors` 用户组内的用户登录，可以看到如下界面：

My courses

You haven't created any courses yet.

[CREATE NEW COURSE](#)

这个页面会显示当前用户创建的所有课程。

现在来创建新增和修改课程需要的模板，编辑 `courses/manage/course/form.html`，添加下列代码：

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

{% block title %}
    {% if object %}
        Edit course "{{ object.title }}"
    {% else %}
        Create a new course
    {% endif %}
{% endblock %}

{% block content %}

<h1>
    {% if object %}
        Edit course "{{ object.title }}"
    {% else %}
        Create a new course
    {% endif %}
</h1>

<div class="module">
    <h2>Course info</h2>
    <form action"." method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Save course"></p>
    </form>
</div>
{% endblock %}
```

这个模板由 `CourseCreateView` 和 `CourseUpdateView` 进行操作。在模板内先检查 `object` 变量是否存在，如果存在则显示针对该对象的修改功能。如果不存在就建立一个新的 `Course` 对象。

浏览器中打开 <http://127.0.0.1:8000/course/mine/>，点击CREATE NEW COURSE按钮，可以看到如下界面：

Create a new course

Course info

Subject:

A dropdown menu with a light gray background and a dark gray border, containing a single placeholder dash ('---'). It features a small downward-pointing arrow icon on its right side.

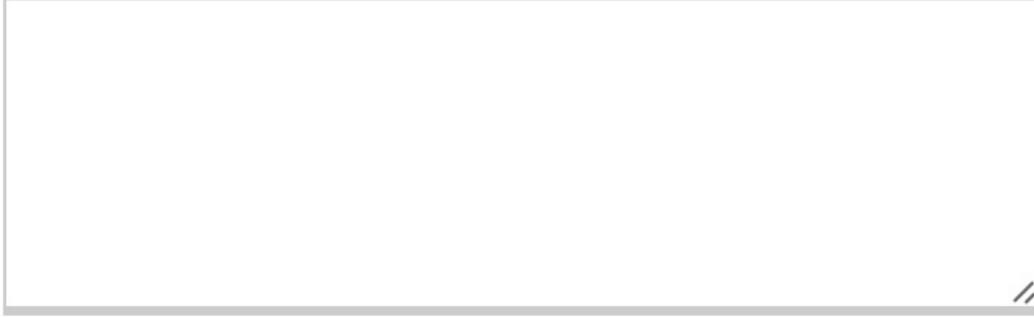
Title:

A rectangular input field with a light gray background and a dark gray border.

Slug:

A rectangular input field with a light gray background and a dark gray border.

Overview:



SAVE COURSE

填写表单后后点击SAVE COURSE进行保存，课程会被保存，然后重定向到课程列表页，可以看到如下界面：

My courses

Django course

[Edit](#) [Delete](#)

[CREATE NEW COURSE](#)

点击其中的Edit链接，可以在看到这个表单页面，但这次是修改已经存在的 Course 对象。

最后来编写 `courses/manage/course/delete.html`，添加下列代码：

```
{% extends "base.html" %}  
{% block title %}Delete course{% endblock %}  
{% block content %}  
    <h1>Delete course "{{ object.title }}"</h1>  
    <div class="module">  
        <form action="" method="post">  
            {% csrf_token %}  
            <p>Are you sure you want to delete "{{ object }}"?</p>  
            <input type="submit" class="button" value="Confirm">  
        </form>  
    </div>  
{% endblock %}
```

注意原书的代码在 `<input>` 元素的的 `class` 属性后边漏了一个"="号

这个模板由继承了 `DeleteView` 的 `CourseDeleteView` 视图操作，负责删除课程。

打开浏览器，点击刚才页面中的Delete链接，跳转到如下确认页面：

Delete course "Django course"

Are you sure you want to delete "Django course"?

CONFIRM

点击CONFIRM按钮，课程就会被删除，然后重定向至课程列表页。

讲师组用户现在可以增删改课程了。下边要做的是通过CMS让讲师组用户为课程添加章节和内容。

5 管理章节与内容

这一节里来建立一个管理课程中章节和内容的系统，将为同时管理课程中的多个章节及其中不同的内容建立表单。章节和内容都需要按照特定的顺序记录在我们的CMS中。

5.1 在课程模型中使用表单集 (formsets)

Django通过一个抽象层控制页面中的所有表单对象。一组表单对象被称为表单集。表单集由多个 `Form` 类或者 `ModelForm` 类的实例组成。表单集内的所有表单在提交的时候会一并提交，表单集可以控制显示的表单数量，对提交的最大表单数量做限制，同时对其中的全部表单进行验证。

表单集包含一个 `is_valid()` 方法用于一次验证所有表单。可以给表单集初始数据，也可以控制表单集显示的空白表单数量。普通的表单集官方文档可以看 <https://docs.djangoproject.com/en/2.0/topics/forms/formsets/>，由模型表单构成的model formset可以看 <https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/#model-formsets>。

由于一个课程由多个章节组成，方便运用表单集进行管理。在 `courses` 应用中建立 `forms.py` 文件，添加如下代码：

```
from django import forms
from django.forms.models import inlineformset_factory
from .models import Course, Module

ModuleFormSet = inlineformset_factory(Course, Module, fields=['title', 'description'], extra=2, can_delete=True)
```

我们使用内置的 `inlineformset_factory()` 方法 构建了表单集 `ModuleFormSet`。内联表单工厂函数是在普通的表单集之上的一个抽象。这个函数允许我们动态的通过与 `Course` 模型关联的 `Module` 模型创建表单集。

对这个表单集我们应用了如下字段：

- `fields`：表示表单集中每个表单的字段

- `extra`：设置每次显示表单集时候的表单数量
- `can_delete`：该项如果设置 `True`，Django会在每个表单内包含一个布尔字段（被渲染成为一个CHECKBOX类型的INPUT元素），供用户选中需要删除的表单

编辑 `courses` 应用的 `views.py` 文件，增加下列代码：

```
from django.shortcuts import redirect, get_object_or_404
from django.views.generic.base import TemplateResponseMixin, View
from .forms import ModuleFormSet

class CourseModuleUpdateView(TemplateResponseMixin, View):
    template_name = 'courses/manage/module/formset.html'
    course = None

    def get_formset(self, data=None):
        return ModuleFormSet(instance=self.course, data=data)

    def dispatch(self, request, pk):
        self.course = get_object_or_404(Course, id=pk, owner=request.user)
        return super(CourseModuleUpdateView, self).dispatch(request, pk)

    def get(self, request, *args, **kwargs):
        formset = self.get_formset()
        return self.render_to_response({'course': self.course, 'formset': formset})

    def post(self, request, *args, **kwargs):
        formset = self.get_formset(data=request.POST)
        if formset.is_valid():
            formset.save()
            return redirect('manage_course_list')
        return self.render_to_response({'course': self.course, 'formset': formset})
```

`CourseModuleUpdateView` 用于对一个课程的章节进行增删改。这个视图继承了以下的mixins和视图：

- `TemplateResponseMixin`：这个 mixin 提供的功能是渲染模块并且返回 HTTP 响应，需要一个 `template_name` 属性用于指定模板位置，提供了一个 `render_to_response()` 方法给模板传入上下文并且渲染模板
- `View`：基础的CBV视图，由Django内置提供。简单继承该类就可以得到一个基本的CBV。

在这个视图中，实现了如下的方法：

1. `get_formset()`：这个方法是创建formset对象的过程，为了避免重复编写所以写了一个方法。功能是根据获得的 `Course` 对象和可选的 `data` 参数来构建一个 `ModuleFormSet` 对象。
2. `dispatch()`：这个方法是 `View` 视图的方法，是一个分发器，HTTP请求进来之后，最先执行的是 `dispatch()` 方法。该方法把小写的HTTP请求的种类分发给同名方法：例如 `GET` 请求会被发送到 `get()` 方法进行处理，`POST` 请求会被发送到 `post()` 方法进行处理。在这个方法里。使用 `get_object_or_404()` 加一个 `id` 参数，从 `Course` 类中获取对象。把这段代码包含在 `dispatch()` 方法中是因为无论 `GET` 还是 `POST` 请求，都会使用 `Course` 对象。在请求一进来的时候，就把 `Course` 对象存入 `self.course`，供其他方法使用。
3. `get()`：处理 `GET` 请求。创建一个 `ModuleFormSet` 然后使用当前的 `Course` 对象渲染模板，使用了 `TemplateResponseMixin` 提供的 `render_to_response()` 方法
4. `post()`：处理 `POST` 请求，在这个方法中执行了如下动作：
 1. 使用请求附带的数据建立 `ModuleFormSet` 对象
 2. 执行 `is_valid()` 方法验证所有表单
 3. 验证通过则使用 `save()` 方法保存，这时增删改都会写入数据库。然后重定向到 `manage_course_list` URL。如果未通过验证，就返回当前表单对象以显示错误信息。

编辑 `courses` 应用中的 `urls.py` 文件，为刚写的视图配置URL：

```
path('<pk>/module/', views.CourseModuleUpdateView.as_view(), name='course_module_update'),
```

在模板目录 `courses/templates/` 下创建一个新目录，叫做 `module`，然后创建 `templates/courses/manage/module/formset.html` 文件，添加下列代码：

```
{% extends "base.html" %}  
{% block title %}  
    Edit "{{ course.title }}"  
{% endblock %}  
{% block content %}  
    <h1>Edit "{{ course.title }}"</h1>  
    <div class="module">  
        <h2>Course modules</h2>  
        <form action="" method="post">  
            {{ formset }}  
            {{ formset.management_form }}  
            {% csrf_token %}  
            <input type="submit" class="button" value="Save modules">  
        </form>  
    </div>  
{% endblock %}
```

在这个模板中，创建了一个表单元素 `<form>`，其中包含了 `formset` 表单集，还包含了一个管理表单 `{{ formset.management_form }}`。这个管理表单包含隐藏的字段用于控制显示起始，总计，最小和最大编号的表单。可以看到创建表单集很简单。

编辑 `courses/templates/course/list.html`，把 `course_module_update` 的链接加在编辑和删除链接之下：

```
<a href="{% url "course_edit" course.id %}">Edit</a>  
<a href="{% url "course_delete" course.id %}">Delete</a>  
<a href="{% url "course_module_update" course.id %}">Edit modules</a>
```

现在模板中有了编辑课程中章节的链接，启动站点，到 <http://127.0.0.1:8000/course/mine/> 创建一个课程然后点击 Edit modules链接，可以看到页面中的表单集如下：

Edit "Django course"

Course modules

Title:

Description:



Delete:



Title:



Description:



Delete:



SAVE MODULES

这个表单集合包含了该课程中的每个 `Module` 对象，然后还多出来2个空白的表单可供填写，这是因为我们为 `ModuleFormSet` 设置了 `extra=2`。输入两个新的章节内容，然后保存表单，再进编辑页面，可以看到又多出来了两个空白表单。

5.2 向课程中添加内容

现在要为章节添加具体的内容。在之前我们定义了四种内容对应四个模型：文字，图片，文件和视频。可能会考虑建立四个不同的视图操作这四个不同的类，但这里我们采用更加通用的方式：建立一个视图来对这四个类进行增删改。

编辑 `courses` 应用中的 `views.py` 文件，添加如下代码：

```
from django.forms.models import modelform_factory
from django.apps import apps
from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View):
    module = None
    model = None
    obj = None
    template_name = 'courses/manage/content/form.html'

    def get_model(self, model_name):
        if model_name in ['text', 'video', 'image', 'file']:
            return apps.get_model(app_label='courses', model_name=model_name)
        return None

    def get_form(self, model, *args, **kwargs):
        Form = modelform_factory(model, exclude=['owner', 'order', 'created', 'updated'])
        return Form(*args, **kwargs)

    def dispatch(self, request, module_id, model_name, id=None):
        self.module = get_object_or_404(Module, id=module_id, course__owner=request.user)
        self.model = self.get_model(model_name)
        if id:
            self.obj = get_object_or_404(self.model, id=id, owner=request.user)
        return super(ContentCreateUpdateView, self).dispatch(request, module_id, model_name, id)
```

这是 `ContentCreateUpdateView` 视图的第一部分。这个类用于建立和更新章节中的内容，这个类定义了如下方法：

1. `get_model()`：检查给出的名字是否在指定的四个类名中，然后用Django的 `apps` 模块，从 `courses` 应用中取出对应的模块，如果沒有找到，就返回 `None`

2. `get_form()`：使用内置的 `modelform_factory()` 方法建立表单集，去掉了四个指定的字段，使用剩下的字段建立。这么做，我们可以不考虑具体是哪个模型，只去掉通用的字段保留剩下的字段。

3. `dispatch()`：这个方法接收下列的URL参数，然后为当前对象设置 `module` 和 `model` 属性：

- `module_id`：章节的id
- `model_name`：内容模型的名称
- `id`：要更新的内容的id，默认值为 `None` 表示新建。

然后来编写该视图的 `get()` 和 `post()` 方法：

```
def get(self, request, module_id, model_name, id=None):
    form = self.get_form(self.model, instance=self.obj)
    return self.render_to_response({'form': form, 'object': self.obj})

def post(self, request, module_id, model_name, id=None):
    form = self.get_form(self.model, instance=self.obj, data=request.POST, files=request.FILES)
    if form.is_valid():
        obj = form.save(commit=False)
        obj.owner = request.user
        obj.save()
        if not id:
            # 新内容
            Content.objects.create(module=self.module, item=obj)
    return redirect('module_content_list', self.module.id)
    return self.render_to_response({'form': form, 'object': self.obj})
```

这两个方法解释如下：

- `get()`：处理 GET 请求。通过 `get_form()` 方法获取需要修改的四种内容之一生成的表单。如果没有 `id`，前置的 `dispatch` 方法里不设置 `self.obj`，所以 `instance=None`，表示新建

- `post()`：处理 POST 请求。通过传入的所有数据创建表单集对象，然后进行验证。如果验证通过，给当前对象设置上 `user` 属性，然后保存。如果没有传入 `id`，说明是新建内容，需要在 `Content` 中追加一条记录关联到 `module` 对象和新建的内容对象。

编辑 `courses` 应用的 `urls.py` 文件，为新视图配置URL：

```
path('module/<int:module_id>/content/<model_name>/create/', views.ContentCreateUpdateView.as_view(),
      name='module_content_create'),
path('module/<int:module_id>/content/<model_name>/<id>/', views.ContentCreateUpdateView.as_view(),
      name='module_content_update'),
```

这两条路由解释如下：

- `module_content_create`：用于建立新内容的URL，带有 `module_id` 和 `model_name` 两个参数，第一个是用来取得对应的 `module` 对象，第二个用来取得对应的内容数据模型。
- `module_content_update`：用于修改原有内容的URL，除了带有 `module_id` 和 `model_name` 两个参数之外，还带有 `id` 用于确定具体修改哪一个内容对象。
-

在 `courses/manage/` 目录下创建一个新目录叫 `content`，再创建 `courses/manage/content/form.html`，添加下列代码：

```
{% extends "base.html" %}
{% block title %}
  {% if object %}
    Edit content "{{ object.title }}"
  {% else %}
    Add a new content
  {% endif %}
{% endblock %}
{% block content %}
<h1>
```

```
{% if object %}
    Edit content "{{ object.title }}"
{% else %}
    Add a new content
{% endif %}
</h1>
<div class="module">
    <h2>Course info</h2>
    <form action="" method="post" enctype="multipart/form-data">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Save content"></p>
    </form>
</div>
{% endblock %}
```

这是视图 `ContentCreateUpdateView` 控制的模板。在这个模板里，使用了一个 `object` 变量，如果 `object` 变量不为 `None`，说明在修改一个已经存在的内容，否则就是新建一个内容。

`<form>` 标签中设置了属性 `enctype="multipart/form-data"`，因为 `File` 和 `Image` 模型中有文件字段。

启动站点，到 <http://127.0.0.1:8000/course/mine/>，点击任何一个已经存在的课程的Edit modules链接，之后新建一个module。

然后打开带有当前Django环境的Python命令行，来进行一些测试，首先取到最后一个建立的module对象：

```
>>> from courses.models import Module
>>> Module.objects.latest('id').id
6
```

取到了这个id之后，打开 <http://127.0.0.1:8000/course/module/6/content/image/create/>，把6替换成你实际取到的结果，可以看到创建 Image 对象的页面：

Add a new content

Course info

Title:

File:

Choose File

no file selected

SAVE CONTENT

现在还不要提交表单，如果提交会报错，因为我们还没有定义 `module_content_list` URL。

现在还需要一个视图用来删除内容。编辑 `courses` 应用的 `views.py` 文件：

```
class ContentDeleteView(View):
    def post(self, request, id):
        content = get_object_or_404(Content, id=id, module__course__owner=request.user)
        module = content.module
```

```
content.item.delete()
content.delete()
return redirect('module_content_list', module.id)
```

这个 `ContentDeleteView` 视图通过 ID 参数获取 `Content` 对象，然后删除相关的 `Text`、`Video`、`Image`、或 `File` 对象，再把 `Content` 对象删除，之后重定向到 `module_content_list` URL。

在就在 `courses` 应用的 `urls.py` 文件中设置该URL:

```
path('content/<int:id>/delete/', views.ContentDeleteView.as_view(), name='module_content_delete'),
```

现在讲师用户就可以增删改内容了。

5.3 管理章节与内容

在上一节里编写好了增删改的视图，现在需要一个视图将一个课程的全部章节和其中的内容展示出来的视图。

编辑 `courses` 应用的 `views.py` 文件，添加下列代码：

```
class ModuleContentListView(TemplateResponseMixin, View):
    template_name = 'courses/manage/module/content_list.html'

    def get(self, request, module_id):
        module = get_object_or_404(Module,
                                  id=module_id,
                                  course__owner=request.user)
        return self.render_to_response({'module': module})
```

这个 `ModuleContentView` 视图通过一个指定的 `Module` 对象的 ID 和当前用户，来获取 `Module` 对象，然后使用该对象渲染模板。

在 `courses` 应用的 `urls.py` 内加入该视图的路由：

```
path('module/<int:module_id>', views.ModuleContentView.as_view(), name='module_content_list'),
```

在 `templates/courses/manage/module/` 目录中新建 `content_list.html`，添加下列代码：

```
{% extends "base.html" %}
{% block title %}
    Module {{ module.order|add:1 }}: {{ module.title }}
{% endblock %}
{% block content %}
    {% with course=module.course %}
        <h1>Course "{{ course.title }}"</h1>
        <div class="contents">
            <h3>Modules</h3>
            <ul id="modules">
                {% for m in course.modules.all %}
                    <li data-id="{{ m.id }}" {% if m == module %}
                        class="selected"{% endif %}>
                        <a href="{% url "module_content_list" m.id %}">
                            <span>
                                Module <span class="order">{{ m.order|add:1 }}</span>
                            </span>
                            <br>
                            {{ m.title }}
                        </a>
                    </li>
                {% empty %}
                    <li>No modules yet.</li>
                
```

```
        {% endfor %}
    </ul>
    <p><a href="{% url "course_module_update" course.id %}">
        Edit modules</a></p>
</div>
<div class="module">
    <h2>Module {{ module.order|add:1 }}: {{ module.title }}</h2>
    <h3>Module contents:</h3>
    <div id="module-contents">
        {% for content in module.contents.all %}
            <div data-id="{{ content.id }}">
                {% with item=content.item %}
                    <p>{{ item }}</p>
                    <a href="#">Edit</a>
                    <form action="{% url "module_content_delete" content.id %}"
                        method="post">
                        <input type="submit" value="Delete">
                        {% csrf_token %}
                    </form>
                {% endwith %}
            </div>
        {% empty %}
        <p>This module has no contents yet.</p>
    {% endfor %}
</div>
<h3>Add new content:</h3>
<ul class="content-types">
    <li><a href="{% url "module_content_create" module.id "text" %}">
        Text</a></li>
    <li><a href="{% url "module_content_create" module.id "image" %}">
        Image</a></li>
    <li><a href="{% url "module_content_create" module.id "video" %}">
        Video</a></li>
    <li><a href="{% url "module_content_create" module.id "file" %}">
        File</a></li>
```

```
        </ul>
    </div>
{%
endwith %}
{% endblock %}
```

这是用来展示该课程中全部章节和内容的模板。迭代全部的章节显示在侧边栏中，然后针对每个章节的内容，通过 `content.item` 迭代其中的相关的所有内容进行展示，然后配上对应的链接。

我们想知道每个 `item` 对象究竟是 `text`, `video`, `image` 或者 `file` 的哪一种，因为我们需要模型的名称来创建修改数据的 URL。此外还需要在模板中按照类别单独把每个内容展示出来。对于一个数据对象，可以通过 `_meta_` 属性获取该数据所属的模型类，但 Django 不允许在视图中使用以下划线开头的模板变量或者属性，以防访问到私有属性或方法。可以通过编写一个自定义的模板过滤器来解决。

在 `courses` 应用中建立如下目录和文件：

```
templatetags/
__init__.py
course.py
```

在其中的 `course.py` 中编写：

```
from django import template

register = template.Library()

@register.filter
def model_name(obj):
    try:
        return obj._meta.model_name
```

```
except AttributeError:  
    return None
```

这是 `model_name` 模板过滤器，在模板里可以通过 `object|model_name` 来获得一个数据对象所属的模型名称。

编辑刚才的 `templates/courses/manage/module/content_list.html`，在 `{% extend %}` 的下一行添加：

```
{% load course %}
```

然后找到下边两行：

```
<p>{{ item }}</p>  
<a href="#">Edit</a>
```

替换成：

```
<p>{{ item }} ({{ item|model_name }})</p>  
<a href="{% url "module_content_update" module.id item|model_name item.id %}">Edit</a>
```

使用了自定义模板过滤器之后，我们在模板中显示内容对象时，就可以通过对对象所属模型的名称来生成URL链接了。编辑 `courses/manage/course/list.html`，添加一个列表页的链接：

```
<a href="{% url "course_module_update" course.id %}">Edit modules</a>  
{% if course.modules.count > 0 %}  
    <a href="{% url "module_content_list" course.modules.first.id %}">Manage contents</a>  
{% endif %}
```

这个新连接跳转到显示第一个章节的内容的页面。

打开 <http://127.0.0.1:8000/course/mine/>，可以看到页面中多出来了Manage contents链接，点击该链接后如下图所示：

The screenshot shows a web application interface for managing course contents. At the top, there is a green header bar with the word "EDUCA". Below it, the title "Course \"Django course\" is displayed. On the left, a sidebar has a dark background and contains the following text:

- Modules
- MODULE 1
Introduction to Django
- Edit modules**

The "Edit modules" link is highlighted in green. The main content area has a light gray background and displays the following information:

- Module 1: Introduction to Django**
- Module contents:**
- This module has no contents yet.**
- Add new content:**
- Four buttons labeled **Text**, **Image**, **Video**, and **File** are shown at the bottom.

在左侧边栏点击一个章节时，该章节的内容就显示在右侧。这个页面还带了链接到添加四种类型内容的页面。实际添加一些内容然后看一下页面效果，内容也会展示出来：

EDUCA

Course "Django course"

Modules

MODULE 1

Introduction to Django

MODULE 2

Configuring Django

[Edit modules](#)

Module 2: Configuring Django

Module contents:

Setting up Django (text)

[Edit](#)

[Delete](#)

Example settings.py (image)

[Edit](#)

[Delete](#)

Add new content:

Text

Image

Video

File

5.4 重新排列章节和内容的顺序

我们需要给用户提供一个简单的可以重新排序的方法。通过JavaScript的拖动插件，让用户通过拖动就可以重新排列章节和内容的顺序。在用户结束拖动的时候，我们使用AJAX来记录当前的新顺序。

5.4.1 使用django-braces模块中的mixins

django-braces 是一个第三方模块，包含了一系列通用的Mixin，为CBV提供额外的功能。可以查看其官方文档：

<https://django-braces.readthedocs.io/en/latest/> 来获得完整的 mixin 列表。

我们要使用 django-braces 中下列mixin：

- `CsrfExemptMixin`：在 POST 请求中不检查CSRF，无需生成 `csrf_token`
- `JsonRequestResponseMixin`：以JSON字符串形式解析请求中的数据，并且序列化响应数据为JSON格式，带有 `application/json` 头部信息

通过 pip 安装 django-braces：

```
pip install django-braces==1.13.0
```

我们需要一个视图，能够接受JSON格式的新的模块顺序。编辑 courses 应用的 `views.py` 文件，添加下列代码：

```
from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class ModuleOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):

    def post(self, request):
        for id, order in self.request_json.items():
            Module.objects.filter(id=id, course__owner=request.user).update(order=order)
        return self.render_json_response({'saved': 'OK'})
```

这个 `ModuleOrderView` 视图的逻辑是拿到JSON数据后，对于其中的每一条记录，更新 `module` 对象的 `order` 字段。

基于类似的逻辑，来编写章节内容的重新排列视图，继续在 `views.py` 中追加代码：

```
class ContentOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):
    def post(self, request):
        for id, order in self.request_json.items():
            Content.objects.filter(id=id, module__course__owner=request.user).update(order=order)
        return self.render_json_response({'saved': 'OK'})
```

然后编辑 `courses` 应用的 `urls.py`，为这两个视图配置URL：

```
path('module/order/', views.ModuleOrderView.as_view(), name='module_order'),
path('content/order/', views.ContentOrderView.as_view(), name='content_order'),
```

最后，需要在模板中实现拖动功能。使用jQuery UI库来完成这个功能。jQuery UI基于jQuery，提供了一系列的界面互动操作，效果和插件。我们使用其中的 `sortable` 元素。首先，需要把jQuery加载到母版中。打开 `base.html`，在加载jQuery的script标签之后加入jQuery UI。

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.css"></script>
```

这里使用了国内的CDN。由于jQueryUI依赖于jQuery，所以要在其后载入。之后编辑

[courses/manage/module/content_list.html](#)，在底部添加如下代码：

```
{% block domready %}
$('#modules').sortable({
    stop: function (event, ui) {
        let modules_order = {};
        $('#modules').children().each(function () {
            $(this).find('.order').text($(this).index() + 1);
            modules_order[$(this).data('id')] = $(this).index();
        });
        $.ajax({
            type: 'POST',
            url: '{% url "module_order" %}',
            contentType: 'application/json; charset=utf-8',
            dataType: 'json',
            data: JSON.stringify(modules_order)
        });
    }
});

$('#module-contents').sortable({
    stop: function (event, ui) {
        let contents_order = {};
        $('#module-contents').children().each(function () {
            contents_order[$(this).data('id')] = $(this).index();
        });
        $.ajax({
            type: 'POST',
            url: '{% url "content_order" %}',
        });
    }
});
```

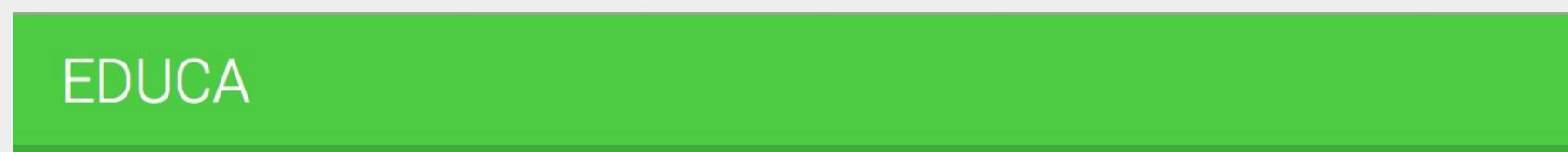
```
        contentType: 'application/json; charset=utf-8',
        dataType: 'json',
        data: JSON.stringify(contents_order),
    });
}
});
{% endblock %}
```

译者注：这里对原书的代码增加了let声明。

这段代码加载在 `{% domready %}` 块中，会在页面DOM加载完成后立刻执行。在代码中为所有的侧边栏中的章节列表定义了一个 `sortable` 方法，为内容也定义了一个同样功能的方法。这段代码做了下列工作：

1. 使用 `#modules` 选择器，为 `modules` 的HTML元素定义了 `sortable` 元素
2. 定义了一个 `stop` 事件处理函数，用户停止拖动后触发该事件
3. 建立了一个空字典 `modules_order` (JS里叫做对象)，其中的键是 `module` 的ID (`LI`元素的`data-id`属性的值)，值是重新排列后的顺序。
4. 遍历拖动后的 `#module` 的子元素，取得此时每个元素的 `data-id` 和此时在列表中的索引，用此时的id作为键，其顺序作为值，更新 `modules_order` 字典。
5. 通过AJAX发送 `POST` 请求到 `content_order` URL进行处理，请求中带有 `modules_order` JSON字符串，交给 `ModuleOrderView` 进行处理。

用于排序内容部分的 `sortable` 元素与上述这个相似。启动站点，重新加载编辑内容的页面，现在可以通过拖动重新排列章节和内容的顺序，如下图所示：



Course "Django course"

Modules

MODULE 1
Introduction to Django

MODULE 2
Configuring Django

Edit modules

Module 2: Configuring Django

contents:

Setting up Django (text)

Edit Delete

Example settings.py (image)

Edit Delete

Add new content:

Text **Image** **Video** **File**

现在我们就实现了拖动排序功能。

总结

这一章学习了如何建立一个CMS。使用了模型继承和创建自定义字段，同时使用了基于类的视图和mixins。还使用了表单集和实现了一个管理不同的内容的系统。

在下一章，将学习创建一个学生注册系统，以及在页面内渲染各种课程内容，以及学习Django缓存框架的使用。

第十一章 渲染和缓存课程内容

在上一章中，使用了模型继承和通用关系建立弹性的课程、章节和内容的关联数据模型，并且建立了一个CMS系统，在其中使用了CBV，表单集和AJAX管理课程内容。在这一章将要做的事情是：

- 创建公开对外展示课程的视图
- 创建学生注册系统
- 学生选课功能
- 渲染不同的课程内容
- 采用缓存框架缓存课程内容

我们就从建立一个课程目录，供学生们浏览和选课来开始本章。

1 展示课程

为了展示课程，我们需要实现如下功能：

1. 列出所有可用的课程，可以通过课程主题来进行筛选
2. 显示某个课程的具体内容

由于数据模型已经齐备，编辑 `courses` 应用的 `views.py` 文件，增加以下代码：

```
from django.db.models import Count
from .models import Subject

class CourseListView(TemplateResponseMixin, View):
    model = Course
```

```
template_name = 'courses/course/list.html'

def get(self, request, subject=None):
    subjects = Subject.objects.annotate(total_courses=Count('courses'))
    courses = Course.objects.annotate(total_modules=Count('modules'))
    if subject:
        subject = get_object_or_404(Subject, slug=subject)
        courses = courses.filter(subject=subject)
    return self.render_to_response({'subjects': subjects, 'subject': subject, 'courses': courses})
```

这个 `CourseListView` 继承了 `TemplateResponseMixin` 和 `View` 视图，执行了如下任务：

1. 取所有的主题，使用了 `annotate()` 分组和 `Count()` 聚合方法生成一个其中包含课程的数量字段。
2. 获得所有课程，同样进行了分组，增加了一个按照章节分组计数的字段。
3. 如果传入了某个具体的主题 `slug` 字段，就取得该 `slug` 对应的具体主题，并且将课程设置为该主题对应的课程，而不是全部课程。
4. 使用个 `TemplateResponseMixin` 类提供的 `render_to_response()` 方法将上边几个结果返回给模板。

再创建一个显示具体课程的视图，在 `views.py` 里增加如下内容：

```
from django.views.generic.detail import DetailView

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'
```

这个视图继承了Django内置的 `DetailView` 视图，为其指定模型 `model` 和模板 `template_name` 属性，该CBV会在模板上渲染其中该数据类的详情。 `DetailView` 需要一个 `slug` 或者主键 `pk` 来从指定的 `Course` 模型中获取具体信息，然后在 `template_name` 属性指定的模板中进行渲染。

编辑 `educa` 项目的根路由 `urls.py` 文件，增加以下代码：

```
from courses.views import CourseListView
urlpatterns = [
    # ...
    path('', CourseListView.as_view(), name='course_list'),
]
```

我们想让访问该站点的人默认就来到列表页，因此将 `course_list` 设置为匹配网站的根目录，将这行放在所有URL的最下边，其他课程相关的URL都带有 `/course/` 前缀。

然后编辑 `courses` 应用的 `urls.py` 文件，增加下边两条URL：

```
path('subject/<slug:subject>', views.CourseListView.as_view(), name='course_list_subject'),
path('<slug:slug>', views.CourseDetailView.as_view(), name='course_detail'),
```

我们添加了如下两条路由：

- `course_list_subject`：展示所有的或某个主题下的课程
- `course_detail`：展示某个课程的详情

来为这两个视图创建模板，在 `templates/courses/` 目录下创建：

```
course/
    list.html
    detail.html
```

编辑 `courses/course/list.html` 模板，添加下列代码：

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

{% block title %}
    {% if subject %}
        {{ subject.title }} courses
    {% else %}
        All courses
    {% endif %}
{% endblock %}

{% block content %}
<h1>
    {% if subject %}
        {{ subject.title }} courses
    {% else %}
        All courses
    {% endif %}
</h1>
<div class="contents">
    <h3>Subjects</h3>
    <ul id="modules">
        <li {% if not subject %}class="selected"{% endif %}>
            <a href="{% url "course_list" %}">All</a>
        </li>
        {% for s in subjects %}
            <li {% if subject == s %}class="selected"{% endif %}>
                <a href="{% url "course_list_subject" s.slug %}">
                    {{ s.title }}
                    <br><span>{{ s.total_courses }} courses</span>
                </a>
            </li>
        {% endfor %}
    </ul>
</div>
<div class="module">
    {% for course in courses %}
        {% with subject=course.subject %}
```

```
<h3><a href="{% url "course_detail" course.slug %}">
    {{ course.title }}</a></h3>
<p>
    <a href="{% url "course_list_subject" subject.slug %}">
        {{ subject }}</a>.
    {{ course.total_modules }} modules.
    Instructor: {{ course.owner.get_full_name }}
</p>
{% endwith %}
{% endfor %}
</div>
{% endblock %}
```

这个模板用来列出所有的课程。模板中创建了一个列表，展示所有的 `Subject` 对象和反向解析的链接 `course_list_subject`，使用判断来切换CSS类 `selected` 用于显示当前被选中的主题。然后迭代所有的 `Course` 对象，展示其中的总章节数目以及讲师的名字。

启动站点，打开 <http://127.0.0.1:8000/>，可以看到如下页面：

All courses

Subjects

- All
- Mathematics
1 COURSES
- Music
0 COURSES
- Physics
0 COURSES
- Programming
2 COURSES

Django course
Programming. 2 modules. Instructor: Antonio Melé

Python for beginners
Programming. 2 modules. Instructor: Laura Marlon

Algebra basics
Mathematics. 4 modules. Instructor: Laura Marlon

左侧边栏包含所有的主题以及主题中的课程数量，右侧在默认情况下，显示所有的主题其中的所有的课程。如果选择任
何主题，则只显示该主题对应的课程。

编辑 `courses/course/detail.html`，添加如下代码：

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

{% block title %}
    {{ object.title }}
{% endblock %}

{% block content %}
    {% with subject=course.subject %}

        <h1>
            {{ object.title }}
        </h1>

        <div class="module">
            <h2>Overview</h2>
            <p>
                <a href="{% url "course_list_subject" subject.slug %}">
                    {{ subject.title }}</a>.
                {{ course.modules.count }} modules.
                Instructor: {{ course.owner.get_full_name }}
            </p>
            {{ object.overview|linebreaks }}
        </div>
    {% endwith %}
{% endblock %}
```

这个模板中显示了一个课程的整体情况和其中的具体内容。在浏览器中打开 <http://127.0.0.1:8000/>，点击右侧任意一个课程，可以看到如下页面：

Django course

Overview

Programming. 2 modules. Instructor: Antonio Melé

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

我们已经建立好了公共的（不需要特别权限）的展示课程的页面，下一步，需要允许用户以学生身份注册并且选课。

2 增加学生注册功能

建立一个新的应用来管理学生注册功能：

```
python manage.py startapp students
```

编辑 `settings.py` 激活新应用：

```
INSTALLED_APPS = [
    # ...
    'students.apps.StudentsConfig',
]
```

2.1 创建注册视图

编辑 `students` 目录内的 `views.py` 文件，增加如下代码：

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login

class StudentRegistrationView(CreateView):
    template_name = 'students/student/registration.html'
    form_class = UserCreationForm
    success_url = reverse_lazy('student_course_list')

    def form_valid(self, form):
        result = super(StudentRegistrationView, self).form_valid(form)
        cd = form.cleaned_data
        user = authenticate(username=cd['username'], password=cd['password1'])
        login(self.request, user)
        return result
```

这是允许学生注册的视图，继承了内置的 `CreateView` 视图，该视图提供了创建模型对象的一般方法。这个视图需要如下属性：

1. `template_name`：需要渲染的模板位置
2. `form_class`：必须是一个 `ModelForm` 对象，这里指定为Django内置的建立新用户的 `UserCreationForm` 表单。

3. `success_url`：成功后跳转的URL，通过反向解析 `student_course_list` 获取，看名字就知道是给学生列出课程列表的URL，会在稍后配置该URL。

`form_valid()` 方法表单数据成功验证的时候执行，该方法必须返回一个HTTP响应。重写该方法以使用户在成功注册之后就登录。

在 `students` 应用中创建 `urls.py` 文件，并在其中设置该视图的URL：

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

urlpatterns = [
    path('register/', views.StudentRegistrationView.as_view(), name='student_registration'),
]
```

然后编辑 `educa` 项目的根 `urls.py`，为 `students` 应用配置二级路由：

```
urlpatterns = [
    # ...
    path('students/', include('students.urls')),
]
```

之后在 `students` 应用中创建如下目录和模板文件：

```
templates/
  students/
    student/
      registration.html
```

编辑 `students/student/registration.html` 模板，添加下列代码：

```
{% extends "base.html" %}  
{% block title %}  
    Sign up  
{% endblock %}  
{% block content %}  
    <h1>  
        Sign up  
    </h1>  
    <div class="module">  
        <p>Enter your details to create an account:</p>  
        <form action="" method="post">  
            {{ form.as_p }}  
            {% csrf_token %}  
            <p><input type="submit" value="Create my account"></p>  
        </form>  
    </div>  
{% endblock %}
```

启动站点，到 <http://127.0.0.1:8000/students/register/>，应该可以看到如下的注册表单：

Sign up

Enter your details to create an account:

Username: Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.

>Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

CREATE MY ACCOUNT

注意此时 `StudentRegistrationView` 视图的 `success_url` 属性中的 `student_course_list` URL 还没有配置，所以还不能提交表单，否则会报错。会在下一节中配置该URL。

2.2 选课功能

在用户注册成功之后，应该能够让其选课以便加入到某门课的学习中去。很显然，一个学生可以选择多门课程，一个课程也有多个学生，需要在 `Course` 模型和 `User` 模型之间添加一个多对多关系。

编辑 `courses` 应用的 `models.py` 文件，为 `Course` 模型添加一个字段：

```
class Course(models.Model):
    # .....
    students = models.ManyToManyField(User, related_name='courses_joined', blank=True)
```

之后立刻执行数据迁移过程，不再赘述

现在我们可以过多对多关系来设置学生与课程之间的关系了。之后需要编写一个视图用于实现选课功能。

在 `students` 应用内创建 `forms.py`：

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
    course = forms.ModelChoiceField(queryset=Course.objects.all(), widget=forms.HiddenInput)
```

这个表单是学生选课时候提交的表单。`course` 字段使用了 `ModelChoiceField`，供学生选择所有的课程，使用了 `HiddenInput` 插件不给学生展示该表单。这个表单将在 `CourseDetailView` 中使用，在页面上显示一个选课按钮让学生进行选课。

编辑 `students` 应用的 `views.py` 文件，增加如下代码：

```
from django.views.generic.edit import FormView
from django.contrib.auth.mixins import LoginRequiredMixin
from .forms import CourseEnrollForm

class StudentEnrollCourseView(LoginRequiredMixin, FormView):
    course = None
    form_class = CourseEnrollForm

    def form_valid(self, form):
        self.course = form.cleaned_data['course']
        self.course.students.add(self.request.user)
        return super(StudentEnrollCourseView, self).form_valid(form)

    def get_success_url(self):
        return reverse_lazy('student_course_detail', args=[self.course.id])
```

这是用于处理学生选课的 `StudentEnrollCourseView` 视图。该视图继承了内置 `LoginRequiredMixin` 类，一定要用户登录才能使用该功能。还继承了内置的 `FormView`，因为我们要处理表单提交。设置 `form_class` 属性为 `CourseEnrollForm` 类，设置了一个 `course` 属性用于保存学生选的课程对象。当表单验证通过的时候，取得当前的用户，设置多对多关系，然后调用父类的方法保存数据。

`get_success_url()` 方法返回了成功之后跳转的URL，这个方法和 `success_url` 属性的功能一样。该URL会在下一节中设置。

编辑 `students` 应用中的 `urls.py` 文件，为该视图配置URL：

```
path('enroll-course/', views.StudentEnrollCourseView.as_view(), name='student_enroll_course'),
```

然后在课程详情页面增加一个选课按钮，编辑 `courses` 应用中的 `views.py` 文件，找到 `CourseDetailView` 视图，修改成如下所示：

```
from students.forms import CourseEnrollForm

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'

    def get_context_data(self, **kwargs):
        context = super(CourseDetailView, self).get_context_data(**kwargs)
        context['enroll_form'] = CourseEnrollForm(initial={'course':self.object})
        return context
```

这里重写了 `get_context_data()` 方法，把这个表单添加到模板变量中，并且将表单中隐藏的字段内容初始化成了当前的课程对象，所以可以直接通过按钮提交表单，无需填写隐藏字段。

在 `courses/course/detail.html` 文件中找到如下一行：

```
{% object.overview|linebreaks %}
```

将其替换成如下代码：

```
{% object.overview|linebreaks %}
{% if request.user.is_authenticated %}
    <form action="{% url "student_enroll_course" %}" method="post">
        {{ enroll_form }}
        {% csrf_token %}
        <input type="submit" class="button" value="Enroll now">
    </form>
{% else %}
    <a href="{% url "student_registration" %}" class="button">
        Register to enroll
    </a>
{% endif %}
```

```
</a>  
{% endif %}
```

这样就给页面添加上了按钮，如果用户已登录，就展示该按钮，包含一个指向 `student_enroll_course` 的隐藏表单，如果未登录，展示一个登录链接供用户登录。

启动站点，在浏览器中打开 <http://127.0.0.1:8000/>，然后点击一个具体的课程，如果已经登录，可以看到该按钮，如下所示：

Overview

Programming. 2 modules. Instructor: Antonio Melé

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

ENROLL NOW

如果未登录，则看到的是一个REGISTER TO ENROLL的按钮。

3 访问课程内容

在学生选好课之后，还必须创建一个视图给学生展示课程中的章节和内容，以便让他们访问课程内容来进行具体学习。

编辑 `students` 应用的 `views.py` 文件，添加下列代码：

```
from django.views.generic.list import ListView
from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView):
    model = Course
    template_name = 'students/course/list.html'

    def get_queryset(self):
        qs = super(StudentCourseListView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])
```

这个视图用来给学生展示所有的课程。该视图继承了需要登录的 `LoginRequiredMixin`。还继承了内置的 `ListView` 用于展示一系列的 `Course` 对象。重写了 `get_queryset()` 方法，通过 `ManyToManyField` 过滤出当前学生的已选课程。

继续在 `views.py` 文件里添加显示详情的类：

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
    model = Course
    template_name = 'students/course/detail.html'

    def get_queryset(self):
        qs = super(StudentCourseDetailView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])

    def get_context_data(self, **kwargs):
        context = super(StudentCourseDetailView, self).get_context_data(**kwargs)
        course = self.get_object()

        if 'module_id' in self.kwargs:
            context['module'] = course.modules.get(id=self.kwargs['module_id'])
        else:
```

```
    context['module'] = course.modules.all()[0]
    return context
```

这个视图用于向学生展示他们选的课程和章节。依然重写了`get_queryset()`方法用于返回与当前学生已选课程。重写了`get_context_data()`方法，如果给了一个`module_id`，就将模板变量`module`设置为这个`module_id`对应的课程，如果没给出，默认为该课程的第一个章节。这样学生就能浏览整个课程的章节。

然后在`students`应用中的`urls.py`中为该视图配置URL：

```
path('course/',views.StudentCourseListView.as_view(),name='student_course_list'),
path('course/<pk>',views.StudentCourseDetailView.as_view(),name='student_course_detail'),
path('course/<pk>/<module_id>',views.StudentCourseDetailView.as_view(),name='student_course_detail_module'),
```

在`students`应用中的`templates/students/`目录下创建如下文件和目录结构：

```
course/
  detail.html
  list.html
```

编辑`students/course/list.html`：

```
{% extends "base.html" %}
{% block title %}My courses{% endblock %}
{% block content %}
  <h1>My courses</h1>
  <div class="module">
    {% for course in object_list %}
      <div class="course-info">
        <h3>{{ course.title }}</h3>
        <p><a href="{% url "student_course_detail" course.id %}">
```

```
        Access contents</a></p>
    </div>
    {% empty %}
    <p>
        You are not enrolled in any courses yet.
        <a href="{% url "course_list" %}">Browse courses</a>
        to enroll in a course.
    </p>
    {% endfor %}
</div>
{% endblock %}
```

这个模板用于展示用户所有选的课程。注意在上一小节里，学生注册成功之后，会被重定向至 `student_course_list` URL，但是如果在其他页面登录，会被导向内置的验证模块的相关URL，所以修改 `settings.py`：

```
from django.urls import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')
```

设置成这样之后，所有内置 `auth` 模块完成登录操作之后都跳转到该指定地址。现在所有的学生在注册成功之后都会跳转到 `student_course_list` URL，即显示该学生已选课程的页面。

再编辑 `students/course/detail.html`，添加下列代码：

```
{% extends "base.html" %}
{% block title %}
    {{ object.title }}
{% endblock %}
{% block content %}
    <h1>
        {{ module.title }}
    </h1>
```

```
<div class="contents">
    <h3>Modules</h3>
    <ul id="modules">
        {% for m in object.modules.all %}
            <li data-id="{{ m.id }}" {% if m == module %}class="selected"
                {% endif %}>
                <a href="{% url "student_course_detail_module" object.id m.id %}">
                    <span>Module <span class="order">{{ m.order|add:1 }}</span></span>
                    <br>
                    {{ m.title }}
                </a>
            </li>
        {% empty %}
        {% else %}<li>No modules yet.</li>
        {% endif %}
    </ul>
</div>
<div class="module">
    {% for content in module.contents.all %}
        {% with item=content.item %}
            <h2>{{ item.title }}</h2>
            {{ item.render }}
        {% endwith %}
    {% endfor %}
</div>
{% endblock %}
```

这是用于让学生具体学习已选课程内容的页面。首先我们创建了一个列表包含所有章节，并且高亮当前章节，然后迭代所有当前章节中的内容，使用了一个{{ item.render }}来展示具体的内容。

此时render()方法还没有编写，在下一节中就来为每个内容对象编写这个方法来展示不同种类的内容。

3.1 渲染各种课程内容

我们想为不同的内容编写一个统一的渲染方式。编辑 `courses` 应用的 `models.py` 文件，来为这四个类共同继承的基类 `ItemBase` 模型编写一个 `render()` 方法：

```
from django.template.loader import render_to_string
from django.utils.safestring import mark_safe

class ItemBase(models.Model):
    # .....
    def render(self):
        return render_to_string('courses/content/{}.html'.format(self._meta.model_name), {'item': self})
```

这个方法利用的内置的 `render_to_string()` 方法，传入一个模板名称和上下文，然后模板渲染成为一个HTML字符串。每种类型的内容采用不同名称的模板。使用 `self._meta.model_name` 获取当前的模型名字。通过这个 `render()` 方法，就得到了渲染内容的通用接口。

在 `courses` 应用的 `templates/courses/` 下边建立如下目录和文件结构：

```
content/
  text.html
  file.html
  image.html
  video.html
```

编辑 `courses/content/text.html`，添加如下代码：

```
{{ item.content|linebreaks|safe }}
```

编辑 `courses/content/file.html`，添加如下代码：

```
<p><a href="{{ item.file.url }}" class="button">Download file</a></p>
```

编辑 `courses/content/image.html`，添加如下代码：

```
<p></p>
```

由于 `ImageField` 和 `FileField` 都是文件字段，为了管理这两个字段，必须在 `settings.py` 中配置媒体文件的路径：

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

回忆一下，这里 `MEDIA_URL` 是指上传媒体文件的路径，`MEDIA_ROOT` 是指的查找媒体文件的路径。

编辑项目的根 `urls.py` 文件，在开头添加下列导入代码：

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

然后在末尾追加：

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

现在我们的站点就能够接受文件上传和提供文件存储了。在开发的过程中，Django会管理媒体文件。但在正式生产环境中，就不能如此配置了。我们将在第十三章学习生产环境的配置。

关于 `video.html`，有点额外的事情要做。将使用 `django-embed-video` 模块来集成视频内容。`django-embed-video` 是一个第三方模块，可以将来自 YouTube 或者 Vimeo 等来源的视频内容集成到模板中，只需为其提供视频的 URL 即可。

安装该模块：

```
pip install django-embed-video==1.1.2
```

然后在 `settings.py` 里激活该应用：

```
INSTALLED_APPS = [
    # ...
    'embed_video',
]
```

可以在 <https://django-embed-video.readthedocs.io/en/latest/> 找到这个模块的文档。

现在来编辑 `video.html`：

```
{% load embed_video_tags %}
{% video item.url "small" %}
```

现在启动站点，到 <http://127.0.0.1:8000/course/mine/>，以 `Instructors` 组内用户的身份登录，为一个课程添加各种课程内容，视频地址可以拷贝任意的 YouTube 链接比如 <https://www.youtube.com/watch?v=bgV39D1mZ2U>。

在添加完内容之后，到 <http://127.0.0.1:8000/> 点击刚创建的课程，再点击 `Enroll Now`，之后被重定向到课程列表，应该可以看到类似下面的页面：

Introduction to Django

Modules

MODULE 1

Introduction to Django

MODULE 2

Configuring Django

MODULE 3

Your first Django project

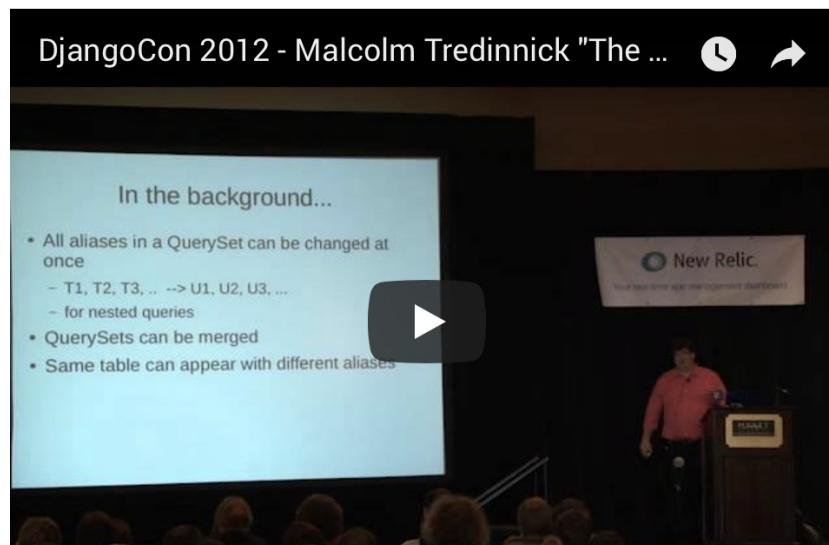
MODULE 4

Django URLs

Why Django?

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers , it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Django video



这样就完成了展示内容的通用方法。

译者注：这里还有一些不完善的地方，比如选了某个课之后回到列表页面再次进入已经选过的课程，会看到Enroll Now还在，其实应该显示Start to Learn之类的词语。这只要在模板中检测一下当前的课程是否包含在用户已经选择的课程中就可以了。

此外在测试代码的时候还发现，如果一个课程内没有章节，则作者在 `StudentCourseDetailView` 中的最后一句：

```
context['module'] = course.modules.all()[0]
```

此处硬编码了返回第一个查询结果，就会报错，修改方法是做个判断，如果长度=0，就返回空就可以了，这样页面不会渲染出内容。

4 使用缓存框架

HTTP请求对我们的Web应用来说，意味着查询数据和处理业务逻辑和渲染模板等工作，这比返回一个静态的页面开销要大很多。

当站点的流量越来越大的时候，大量访问给后端带来的压力是巨大的。这个时候就是缓存系统大派用场的时刻。把一个HTTP请求导致的数据查询，业务逻辑处理结果，甚至渲染后的内容缓存起来，就可以避免在后续类似的请求中反复执行开销大的操作，会有效地提高网站的响应时间。

Django包含一个健壮的缓存系统，可以缓存不同粒度的数据。可以缓存一个查询，一个视图的返回结果，部分模板的渲染内容，甚至整个站点。在缓存系统中存储的数据有时效性，可以设置其过期的时间。

当应用接到一个HTTP请求的时候，通常按照如下的顺序使用缓存系统：

1. 在缓存系统中寻找HTTP请求需要的数据
2. 如果找到了，返回缓存的数据

3. 如果没有找到，按照如下顺序执行：

1. 进行数据查询或者处理，得到数据
2. 将数据保存在缓存内
3. 返回数据

关于详细的缓存机制，可以官方文档：<https://docs.djangoproject.com/en/2.0/topics/cache/>。

4.1 可用的缓存后端

就像数据库一样，Django的缓存机制可以使用多种缓存服务后端来完成，主要有这些：

1. `backends.memcached.MemcachedCache` 或 `backends.memcached.PyLibMCCache`：是基于Memcached服务的后端。具体使用哪种后端取决于采用哪种Python支持的Memcached模块。
2. `backends.db.DatabaseCache`：使用数据库作为缓存（还记得Redis吗）
3. `backends.filebased.FileBasedCache`：使用文件作为缓存，序列化每个缓存数据为一个单独的文件
4. `backends.locmem.LocMemCache`：本地内存缓存，这是默认值。
5. `backends.dummy.DummyCache`：伪缓存机制，仅用于开发。提供了缓存界面但实际上不缓存任何内容。每个进程的缓存互相独立而且线程安全。

对于优化性能而言，最好选取基于内存缓存的缓存机制比如Memcached后端。

4.2 安装Memcached服务

我们将使用Memcached缓存。Memcached在内存中运行，占用一定大小的内存作为缓冲区。当被分配的内存用光的时候，Memcached就会以新数据替代较老的数据。

在<https://memcached.org/downloads> 下载Memcached，如果是Linux系统，可以使用下列命令编译安装：

```
./configure && make && make test && sudo make install
```

如果使用MacOS X而且安装了Homebrew，可以直接通过 `brew install memcached` 安装，也可以在 <https://brew.sh/> 下载。安装了Memcached之后，可以通过一个命令启动服务：

```
memcached -l 127.0.0.1:11211
```

此时Memcached就会在默认的 `11211` 端口运行。还可以通过 `-l` 参数指定其他的主机和端口号。可以在 <https://memcached.org> 查看文档。

还需要安装Python的Memcached模块：

```
pip install python-memcached==1.59
```

4.3 缓存设置

Django提供了下列缓存设置：

- `CACHES`：一个字典，包含当前项目所有可用的缓存
- `CACHE_MIDDLEWARE_ALIAS`：缓存的别名
- `CACHE_MIDDLEWARE_KEY_PREFIX`：缓存键名的前缀，如果不同站点都用同一个Memcached服务，设置这个KEY可以避免发生键冲突
- `CACHE_MIDDLEWARE_SECONDS`：缓存页面的时间

通过 `CACHES` 可以设置项目的缓存系统。这个设置以字典的形式，可以配置多个缓存后端的设置。每个 `CACHES` 中的缓存后端有如下设置：

- **BACKEND**：缓存后端
- **KEY_FUNCTION**：生成键的函数，是一个字符串，包含一个可调用函数的位置，这个函数接受前缀，版本和键名为参数，返回一个最终的缓存键
- **KEY_PREFIX**：缓存键名的前缀
- **LOCATION**：缓存后端的位置，根据不同的后端配置，可能是一个目录，一个主机+端口或者一个内存缓存的名称
- **OPTIONS**：其他的向缓存后端传递的配置参数
- **TIMEOUT**：过期设置，单位是秒。默认是300秒=5分钟，如果设置为None，则缓存键不会过期。
- **VERSION**：缓存键的版本号，用于缓存版本信息

4.4 为项目配置Memcached缓存

编辑 `settings.py`，将上述的缓存设置加入到文件中：

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

这里指定了后端为 `Memcached`，然后指定了主机IP和端口号。如果有很多Memcached配置在不同主机上，可以给 `LOCATION` 传一个列表。

4.4.1 监控Memcached服务

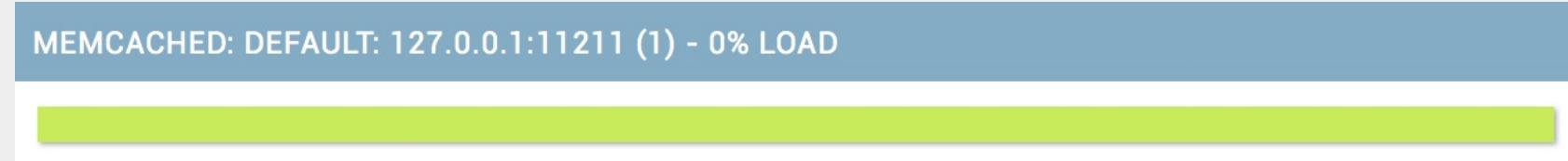
为了监控Memcached的服务，可以使用第三方模块 `django-memcache-status`，该模块可以在管理后台中显示 Memcached的统计情况。安装该模块：

```
pip install django-memcache-status==1.3
```

编辑 `setting.py`，激活该应用：

```
INSTALLED_APPS = [  
    # ...  
    'memcache_status',  
]
```

确保Memcached服务在运行，然后进入站点的管理后台，可以看到如下的内容：



绿色的条代表空闲的缓存容量，红色代表已经使用的内容。如果点击标题，可以展开一个详情页看具体内容。已经为项目配置好了Memcached服务，现在来缓存具体内容。

4.5 缓存级别

Django为不同粒度的数据提供了如下的缓存级别：

1. **Low-level cache API**: 粒度最细，缓存精确的查询或者计算结果
2. **Per-view cache**: 对单独的视图进行缓存
3. **Template cache**: 缓存模板片段
4. **Per-site cache**: 站点缓存，最高级别缓存。

在建立缓存系统之前，必须仔细考虑缓存策略。建议对不基于具体用户身份的，系统开销比较大的数据库查询和密集计算进行缓存

4.6 使用Low-level cache API

Low-level缓存API可以存储任意粒度的对象，该功能位于`django.core.cache`，可以导入进来，：

```
from django.core.cache import cache
```

这个缓存默认使用会使用配置中的`default`名称对应的缓存后端，类似于数据库，等于通过`caches['default']`获取缓存后端。

可以通过如下命令得到一个使用某个具体配置名称的缓存配置：

```
from django.core.cache import caches
my_cache = caches['alias']
```

在导入当前Django环境的Python命令行中里进行一些实验：

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

通过使用了`set(key,value,timeout)`方法，向默认的缓存后端存入了一个键名叫`'musician'`，值是`'Django Reinhardt'`，20秒过期。如果不给出具体时间，则Django使用`settings.py`中的默认设置。然后再输入：

```
>>> cache.get('musician')
'Django Reinhardt'
```

可以获取对应的值。等待20秒再执行上述命令：

```
>>> cache.get('musician')
```

说明该键已经过期，`get()`方法返回`None`。

不要在缓存中存储值为`None`的键，否则无法区分缓存命中与否。

再实验如下代码：

```
>>> from courses.models import Subject  
>>> subjects = Subject.objects.all()  
>>> cache.set('all_subjects', subjects)
```

这里先执行了一个`QuerySet`查询，然后将查询的结果缓存到`all_subjects`键中。来试试从缓存中取数：

```
>>> cache.get('all_subjects')  
<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>, <Subject: Programming>]>
```

现在我们知道如何使用缓存了。下一步就是给常用的视图增加缓存机制。打开`courses`应用的`views.py`文件，首先导入缓存：

```
from django.core.cache import cache
```

在`CourseListView`的`get()`方法里，找到下面这一行：

```
subjects = Subject.objects.annotate(total_courses=Count('courses'))
```

将其修改成：

```
subjects = cache.get('all_subjects')
if not subjects:
    subjects = Subject.objects.annotate(total_courses=Count('courses'))
    cache.set('all_subjects', subjects)
```

在这段代码里，我们先尝试去缓存中获取 `all_subjects` 这个键，如果结果为 `None`，说明缓存中没有，执行正常数据查询，然后将结果存入到缓存中。

启动站点，访问 <http://127.0.0.1:8000/>，只要访问这个路径，刚才配置的缓存就会启动，由于第一次执行，之前没有缓存内容，所以视图就会将查询结果放入缓存。此时进入管理后台查看Memcached的统计，可以看到如下内容：

MEMCACHED: DEFAULT: 127.0.0.1:11211 (1) - 0% LOAD



Miss Ratio 38%



Avg GET by item 1

Avg GET by seconds/minutes 0/0

Detailed Statistics:

Pid 12606

Uptime 0y, 0d, 0h, 43m, 12s

Time 03/30/18 17:13:02

Version 1.4.20

Libevent 2.0.21-stable

在Memcache的统计数据里找到Curr Item这一项，如果严格按照本文来进行，应该为1，除非之前存储了其他内容。这表示当前缓存中有一个键值对。Get Hits表示有多少次Get操作成功命中缓存数据，Get Miss则表示未命中的次数。最上方的Miss Ration使用这两个值计算得到。

现在打开浏览器，反复刷新 <http://127.0.0.1:8000/>，然后再去看Memcached的统计页面，看看统计数据的变化。

4.6.1 缓存动态数据

但有的时候，想缓存动态生成的数据。这就必须建立动态的键，用于唯一确定对具体的缓存数据。看以下例子：

编辑 courses 应用的 `views.py` 文件，修改 `CourseListView` 视图如下所示：

```
def get(self, request, subject=None):
    subjects = cache.get('all_subjects')
    if not subjects:
        subjects = Subject.objects.annotate(total_courses=Count('courses'))
        cache.set('all_subjects', subjects)
    all_courses = Course.objects.annotate(total_modules=Count('modules'))
    if subject:
        subject = get_object_or_404(Subject, slug=subject)
        key = 'subject_{}_courses'.format(subject.id)
        courses = cache.get(key)
        if not courses:
            courses = all_courses.filter(subject=subject)
            cache.set(key, courses)
    else:
        courses = cache.get('all_courses')
        if not courses:
            courses = all_courses
            cache.set('all_courses', courses)
    return self.render_to_response({'subjects': subjects,
                                    'subject': subject,
                                    'courses': courses})
```

在这个视图里，我们还保存了所有的课程和按主题过滤的课程。`all_courses` 键对应的是所有课程的查询结果集，动态生成的键名 `'subject_{}_courses'.format(subject.id)'` 对应着具体类别的查询结果集。

要注意的是，不能用从缓存中取出来的查询结果再去建立其他查询结果，也就是说下边的代码是不行的：

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

缓存只能用于存储最终可供页面直接使用的查询结果，不能在中间步骤缓存。所以这就是为什么要在视图开始的地方建立基础查询 `all_courses = Course.objects.annotate(total_modules=Count('modules'))`，然后再用 `courses = all_courses.filter(subject=subject)` 生成查询结果的原因。

4.7 缓存模板片段

缓存模板片段是比较高级别的缓存，需要在模板中加载缓存标签：`{% load cache %}`，然后使用 `{% cache %}` 来标记需要缓存的片段。实际使用像这样：

```
{% cache 300 fragment_name %}
...
{% endcache %}
```

如上边例子所示，`{% cache %}` 标签有两个可选的参数，第一个是过期秒数，第二个是为该片段起的名称。如果需要缓存动态生成的模板片段，可以再增加额外的参数用于生成唯一KEY。

编辑 `/students/course/detail.html`，为模板在 `{% extends %}` 标签后加上：

```
{% load cache %}
```

然后找到下列代码：

```
{% for content in module.contents.all %}
  {% with item=content.item %}
    <h2>{{ item.title }}</h2>
    {{ item.render }}
  {% endwith %}
{% endfor %}
```

替换成下列代码：

```
{% cache 600 module_contents module %}
  {% for content in module.contents.all %}
    {% with item=content.item %}
      <h2>{{ item.title }}</h2>
      {{ item.render }}
    {% endwith %}
  {% endfor %}
  {% endcache %}
```

这里使用了600秒的过期时间，指定了该片段的别名为 `module_contents`，然后用 `module` 变量动态创建键，这样就建立了独特的键以避免重复。

如果启用了国际化设置 `USE_I18N=True`，缓存中间件会考虑语言的影响。如果你在一个页面中使用了 `{% cache %}` 标签，下次想从缓存中拿到正确的数据，必须将特定语言的代码和缓存标签一起使用，才能得到正确的结果：例如 `{% cache 600 name request.LANGUAGE_CODE %}.`。

4.8 缓存视图

可以通过使用 `django.views.decorators.cache` 中的 `cache_page` 装饰器来缓存视图的输出结果，需要一个参数 `timeout`，是过期秒数。

在视图中使用该装饰器，编辑 `students` 应用的 `urls.py` 文件，先导入该装饰器：

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

然后把 `cache_page` 用于 `student_course_detail` 和 `student_course_detail_module` 两个URL上，如下：

```
path('course/<pk>/', cache_page(60 * 15)(views.StudentCourseDetailView.as_view()), name='student_course_detail'),
path('course/<pk>/<module_id>/', cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
     name='student_course_detail_module'),
```

这样配置之后，`StudentCourseDetailView` 的结果就会被缓存15分钟。

注意，缓存使用URL来构建缓存键，对同一个视图函数，来自不同URL路由的结果，会被分别缓存。

4.8.1 缓存动态数据

缓存站点是级别最高的缓存，允许缓存整个站点。

要启用站点缓存，需要编辑 `settings.py`，把 `UpdateCacheMiddleware` 和 `FetchFromCacheMiddleware` 中间件加入 `MIDDLEWARE` 设置中：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    # ...
]
```

中间件的顺序至关重要，中间件在HTTP请求进来的时候是按照从上到下的顺序执行，返回响应的时候按照从下到上的顺序执行。`UpdateCacheMiddleware` 必须放在 `CommonMiddleware` 的上边，因为 `UpdateCacheMiddleware` 只在响应的时候才执行。`FetchFromCacheMiddleware` 被放在 `CommonMiddleware` 之后，因为 `FetchFromCacheMiddleware` 需要 `CommonMiddleware` 处理过的请求数据。

然后还需要把下列设置加入到 `settings.py` 中：

```
CACHE_MIDDLEWARE_ALIAS = 'default'  
CACHE_MIDDLEWARE_SECONDS = 60 * 15 # 15 minutes  
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

在这些设置里，设置了使用 `default` 名称的缓存后端，15分钟过期时间，以及设置前缀避免重复。现在站点对所有的 `GET` 请求，都缓存和优先返回缓存的结果。

这样我们就设置好了整个站点的缓存，然而站点缓存对于我们这个站来说是不适合的，因为CMS系统里更改了数据之后，必须立刻返回更新后的数据。所以最佳的方法是缓存向学生返回课程内容的视图或者模板。

我们已经学习过了Django内的各种方法用于缓存数据。应该明智的设置缓存策略，优先缓存开销高的查询和计算。

总结

在这一章，我们创建了公开展示所有课程的页面，通过多对多关系建立了学生注册和选课系统，并且为站点安装了 Memcached 服务并缓存了各个级别的内容。

下一章我们将为项目构建一个RESTful API。

第十二章 创建API

在上一章里，创建了一个学生注册系统和选课系统。然后创建了展示课程内容的视图，以及学习了如何使用Django缓存框架。在这一章里有如下内容：

- 建立RESTful API
- 管理API视图的认证与权限
- 建立API视图集和路由

1 创建RESTful API

你可能会想建立一个接口（API），让其他应用程序和我们的网站进行交互。通过建立一个API，就可以让第三方应用程序自动化的操作和消费我们网站生产的数据。

译者注：使用API而不是模板渲染与前端进行交互，这就是前后端分离的思路。仅使用Django来进行Web开发的话前后端分离并不明显。读者在未来的Web开发中接触到前端框架就会对此有更深的了解。

有很多种方式可以建立这样一个API，推荐根据REST原则来建立这样一个API。REST是Representational State Transfer的简称。RESTful API是基于资源的，即URL用于表示网站所有的资源，HTTP的请求种类比如 `GET`, `POST`, `PUT` 或 `DELETE` 表示对应的行为，即获取，创建，更新和删除数据。不同的HTTP响应码表示这次动作的完成结果，例如 `2XX` 表示该操作成功，`4XX` 表示错误等。

RESTful API常用的数据交换格式是JSON或者XML，我们准备建立一个使用JSON进行数据交换的API。我们的API会提供以下功能：

- 获取主题

- 获取可用的课程
- 获取课程内容
- 在一个课程中注册

我们可以从0开始写视图来建立该API，也可以通过第三方应用简单的为项目建立API，在这方面最出名的第三方应用就是Django REST framework。

1.1 安装Django REST framework

Django REST framework可以让你简单地创建符合REST风格的API，其官方网站是<https://www.django-rest-framework.org/>。

打开系统命令行输入如下命令：

```
pip install djangorestframework==3.8.2
```

然后编辑`settings.py`激活`rest_framework`应用：

```
INSTALLED_APPS = [
    # ...
    'rest_framework',
]
```

再在`settings.py`中加入如下设置：

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly',
    ]
}
```

```
]  
}
```

原书代码这里少了一个左方括号。

`REST_FRAMEWORK` 用于具体设置该模块。`REST framework` 提供了很多设置：`DEFAULT_PERMISSION_CLASSES` 提供了对于增删改查行为的默认权限。我们设置了 `DjangoModelPermissionsOrAnonReadOnly` 作为唯一默认的权限类。

这个权限类依赖于Django的权限系统，让用户可以增删改查数据对象，同时让未登录用户只能进行只读操作。在下边的向视图增加权限一节中还会详细学习这部分功能。

Django REST 框架的全部设置可以在 <https://www.django-rest-framework.org/api-guide/settings/> 找到。

1.2 设置序列化器

在设置好框架后，还需要确定使用的序列化器。网站对外提供的数据应当是经过序列化的标准数据，同时还需要对外界输入的数据进行反序列化。REST框架提供了下列类用于对一个单独对象设置序列化器：

- `Serializer`：为普通的Python类实例提供序列化
- `ModelSerializer`：为数据模型的实例提供序列化
- `HyperlinkedModelSerializer`：与 `ModelSerializer` 的功能相同，但可以通过超链接来表示对象之间的关系，而不是通过主键关联。

让我们来实际建立一个序列化器。在 `courses` 应用中建立如下文件结构：

```
api/  
    __init__.py  
    serializers.py
```

我们创建了一个叫做 `api` 的包，然后打算在这个包里建立序列化器。编辑 `serializers.py` 文件，添加下列代码：

```
from rest_framework import serializers
from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Subject
        fields = ['id', 'title', 'slug']
```

这是继承了 `ModelSerializer` 类的，专门用于 `Subject` 模型的序列化器。定义序列化器的类使用起来和 `Form` 以及 `ModelForm` 类很类似： `Meta` 内的属性允许指定要序列化的类及字段。如果不设置具体的 `fields` 属性，则默认会包含该模型的全部字段。

来实验一下这个序列化器，进入带有Django环境的Python命令行模式：

```
python manage.py shell
```

输入以下命令：

```
>>> from courses.models import Subject
>>> from courses.api.serializers import SubjectSerializer
>>> subject = Subject.objects.latest('id')
>>> serializer = SubjectSerializer(subject)
>>> serializer.data
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

在这个例子里，先获取了一个 `Subject` 实例，然后创建了一个序列化器的实例 `SubjectSerializer` 并访问序列化之后的数据，可以看到模型的数据被序列化成了Python原生的字典类型数据。

1.3 理解解析器 (parser) 与渲染器 (renderer)

序列化的结果在通过HTTP响应返回之前，必须被渲染成为一个特殊的格式。同样，在从HTTP请求中获取数据的时候，也必须解析数据然后反序列化。REST框架包含了渲染器和解析器用于处理这些过程。

先来看看如何解析数据，在Python命令行模式中输入下列命令：

```
>>> from io import BytesIO
>>> from rest_framework.parsers import JSONParser
>>> data = b'{"id":4,"title":"Programming","slug":"programming"}'
>>> JSONParser().parse(BytesIO(data))
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

可以看到，给定一个二进制字节流形式的JSON字符串，使用 `JSONParser` 可以将其反序列化为Python的数据对象。

REST框架还包含渲染器 `Renderer` 类用于格式化API的响应。框架通过上下文内容协商机制来确定使用哪种渲染器，即渲染器会通过HTTP请求的 `Accept` 头部字段来确定这个请求所需要的内容类型来进行判断。还可以通过URL的格式化的前缀来判断，例如，一个请求返回JSON格式响应的访问可能会触发 `JSONRenderer` 渲染器。

再回到Python命令行模式中，在刚才的代码的基础上继续执行下列代码：

```
>>> from rest_framework.renderers import JSONRenderer
>>> JSONRenderer().render(serializer.data)
b'{"id":4,"title":"Programming","slug":"programming"}'
```

使用 `JSONRenderer` 可以将Python数据对象渲染成JSON字符串。REST框架提供了两个不同的渲染器：`JSONRenderer` 和 `BrowsableAPIRenderer`。后者提供了一个浏览API返回数据的web界面。可以在 `settings.py` 的 `REST_FRAMEWORK` 设置中的 `DEFAULT_RENDERER_CLASSES` 选项中设置默认的渲染器。

关于渲染器和解析器的更多说明可以看 <https://www.django-rest-framework.org/api-guide/renderers/> 和 <https://www.django-rest-framework.org/api-guide/parsers/>。

1.4 创建列表和详情视图

REST框架包含一系列内置的通用视图和mixins用于建立API视图，提供了增删改查数据模型对象的功能。关于所有的通用视图和mixin可以看 <https://www.django-rest-framework.org/api-guide/generic-views/>。

现在来建立一个获取 `Subject` 对象的视图，在 `courses/api/` 目录内新建 `views.py` 文件，在其中增加下列代码：

```
from rest_framework import generics
from ..models import Subject
from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer
```

在这段代码中，使用了REST框架提供的 `ListAPIView` 和 `RetrieveAPIView` 两个内置视图，在URL中包含一个主键参数，用于获取具体的数据对象。两个视图都有下列属性：

- `queryset`：基础的QuerySet，用于返回数据

- `serializer_class`：序列化器对象，指定要使用的序列化器

接下来为新的视图配置URL，在`courses/api/`下新建`urls.py`文件，然后编辑其中的内容：

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

app_name = 'courses'

urlpatterns = [
    path('subjects/', views.SubjectListView.as_view(), name='subject_list'),
    path('subjects/<pk>/', views.SubjectDetailView.as_view(), name='subject_detail'),
]
```

然后编辑`educa`项目的根`urls.py`，加上一行：

```
urlpatterns = [
    # .....
    path('api/', include('courses.api.urls', namespace='api')),
]
```

我们为API视图使用了`api`路由命名空间。启动站点，使用`curl`命令访问`http://127.0.0.1:8000/api/subjects/`：

```
curl http://127.0.0.1:8000/api/subjects/
```

会得到和下边很相似的响应：

```
[{"id":1,"title":"Mathematics","slug":"mathematics"}, {"id":2,"title":"Music","slug":"music"},
```

```
{"id":3,"title":"Physics","slug":"physics"},  
 {"id":4,"title":"Programming","slug":"programming"}  
]
```

这个HTTP响应包含一系列JSON格式的字符串，其内容是序列化后的所有 `Subject` 模型中的数据，包含指定的三个字段。如果系统中没有安装 `curl`，可以通过 <https://curl.haxx.se/dlwiz/> 进行安装。也可以通过其他浏览器扩展比如 Postman，在 <https://www.getpostman.com/> 进行安装。

现在不使用curl，而是直接在浏览器中打开 <http://127.0.0.1:8000/api/subjects/>，会看到如下页面：

The screenshot shows a Django REST framework interface. At the top, it says "Django REST framework". Below that, the title "Subject List" is displayed. To the right of the title are two buttons: "OPTIONS" and "GET ▾". Underneath the title, there is a button labeled "GET /api/subjects/". Below this, the response details are shown: "HTTP 200 OK", "Allow: GET, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body starts with a "[", followed by a "{" and then a JSON object: {"id": 1, "title": "Mathematics", "slug": "mathematics"}.

```
        },
        {
            "id": 2,
            "title": "Music",
            "slug": "music"
        },
        {
            "id": 3,
            "title": "Physics",
            "slug": "physics"
        },
        {
            "id": 4,
            "title": "Programming",
            "slug": "programming"
        }
    ]
}
```

这个界面就是由之前提到的 `BrowsableAPIRenderer` 渲染器提供的。页面内显示了结果的头部信息及API的返回信息。还可以通过在URL中包含具体的ID来获取一个 `Subject` 对象，访问 <http://127.0.0.1:8000/api/subjects/1/>，可以发现页面只展示了一个单独的JSON格式的对象数据。

1.5 创建嵌套的序列化器

我们再为 `Course` 模型创建一个序列化器，打开 `courses/api/serializers.py` 继续编辑：

```
from ..models import Course

class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']
```

之后看一下 `Course` 序列化器是如何工作的，进入 Python 命令行模式输入下列命令：

```
>>> from rest_framework.renderers import JSONRenderer
>>> from courses.models import Course
>>> from courses.api.serializers import CourseSerializer
>>> course = Course.objects.latest('id')
>>> serializer = CourseSerializer(course)
>>> JSONRenderer().render(serializer.data)
```

这个时候可以看到查询结果里，该课程包含的模块是一个主键列表的形式，类似这样：

```
"modules": [6, 7, 9, 10]
```

这样的数据意义不大，我们想在结果里包含每个 `Module` 的更多信息，所以还必须给 `Module` 模型也制作一个序列化器，编辑 `serializers.py`，修改成如下：

```
from rest_framework import serializers
from ..models import Module

class ModuleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Module
```

```
        fields = ['order', 'title', 'description']

class CourseSerializer(serializers.ModelSerializer):
    modules = ModuleSerializer(many=True, read_only=True)
    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']
```

首先为 `Module` 模型制作了一个序列化器，然后给 `CourseSerializer` 增加了一个 `modules` 属性，设置为 `Module` 类的序列化器，`many=True` 表示需要序列化多个对象，`read_only` 参数表示这个字段是只读的，不应该被包含在任何需要进行增删改的字段中。

重新启动Python命令行模式，再执行一遍上边Python命令行代码，使用 `JSONRenderer` 渲染序列化器实例的 `data` 属性，可以看到结果中关于 `modules` 的部分被嵌套的 `ModuleSerializer` 序列化成下面这样：

```
"modules": [
    {
        "order": 0,
        "title": "Introduction to overview",
        "description": "A brief overview about the Web Framework."
    },
    {
        "order": 1,
        "title": "Configuring Django",
        "description": "How to install Django."
    },
    ...
]
```

这样就完成了嵌套序列化的工作，关于序列化器的更多信息可以看 <https://www.django-rest-framework.org/api-guide/serializers/>。

1.6 创建自定义API视图

REST框架提供了一个 `APIView` 视图，基于Django内置的 `View` 视图基础上增加了RESTful API的功能，但与 `View` 不同的是， `APIView` 采用了REST框架自定义的处理 `Request` 和 `Response` 对象的方法，并且在返回HTTP响应的时候处理 `APIException` 错误，而且还包含内建的认证系统来管理对视图的访问。

下边通过 `APIView` 来创建一个视图供用户选课，编辑 `courses` 应用的 `api/views.py` 文件，增加如下代码：

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from ..models import Course

class CourseEnrollView(APIView):
    def post(self, request, pk, format=None):
        course = get_object_or_404(Course, pk=pk)
        course.students.add(request.user)
        return Response({'enrolled': True})
```

这个 `CourseEnrollView` 视图管理用户选课的功能。代码解释如下：

1. 创建一个视图继承 `APIView`
2. 在其中定义了 `post()` 方法用于处理POST请求，这个类不需要处理其他类型的HTTP请求。
3. 这个类需要接收一个 `pk` 参数，为课程的主键id，用于取得该课程对象。如果找不到就返回 `404` 错误。
4. 添加当前用户与 `Course` 对象的多对多关系，即选课。

编辑 `api/urls.py` 文件，为新的视图配置URL：

```
path('courses/<pk>/enroll/', views.CourseEnrollView.as_view(), name='course_enroll'),
```

现在理论上我们就可以发送一个POST请求来选课，而无需在页面中点击按钮。然而这么做需要区分用户身份，避免未认证的用户也来发送POST请求。下一节来看看API认证与权限管理是如何工作的。

1.7 处理身份认证

REST框架提供了一个认证类，用于鉴别提交HTTP请求的用户身份。如果认证通过，REST框架会在 `request.user` 中设置认证后的 `User` 对象，如果没有用户通过认证，则 `request` 被设置一个Django内置的 `AnonymousUser` 对象。

REST框架提供如下的认证后端：

- `BasicAuthentication`：这是基础的HTTP认证（BA认证），用户和密码存放在HTTP请求头的 `Authorization` 头部数据中，以Base64格式发送。关于BA认证的具体内容看 [这里](#)。
- `TokenAuthentication`：这是基于token的认证，一个 `Token` 模型用于存放用户的token，HTTP请求头中的 `Authorization` 信息中存储token数据用于验证。
- `SessionAuthentication`：使用Django的session后端进行验证，对于前端发来的AJAX请求一般使用该方式验证。
- `RemoteUserAuthentication`：允许使用web服务器代理认证，会设置一个 `REMOTE_USER` 变量。

除此之外，还可以继承REST框架提供的 `BaseAuthentication` 类并且重写 `authenticate()` 方法来创建自定义的验证后端。

通过 `DEFAULT_AUTHENTICATION_CLASSES` 还可以设置认证是基于每个视图的，还是全局认证。

认证 (Authentication) 只解决用户身份的问题，即识别发请求的用户身份，但不会允许或阻止用户访问视图，必须通过设置用户权限来限制访问视图。

在 <https://www.djangoproject.org/api-guide/authentication/> 可以找到所有认证相关的文档。

在视图中增加 `BasicAuthentication` 类，编辑 `api/views.py` 文件，为 `CourseEnrollView` 添加一行：

```
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    # .....
```

现在视图就可以通过HTTP请求头的 `Authorization` 头部信息进行用户身份认证了。

1.8 为视图增加权限控制

REST框架提供了一个权限系统用于控制对视图的访问。REST框架内建的部分权限有：

- `AllowAny`：完全开放权限，不管用户认证与否，都不做任何限制
- `IsAuthenticated`：仅允许通过认证的用户
- `IsAuthenticatedOrReadOnly`：认证用户具有完整权限，匿名用户只读（只能使用 `GET`，`HEAD`，`OPTIONS` 三种HTTP请求种类）。
- `DjangoModelPermissions`：使用 `django.contrib.auth` 的权限管理系统。视图需要一个 `queryset` 属性，只有认证的用户加上具备访问某个数据类的权限才能够进行操作
- `DjangoObjectPermissions`：也使用Django权限，是基于每个对象的单独权限设置。

如果用户因为权限问题操作失败，则通常会得到下列HTTP响应码和错误信息：

- `HTTP 401` : Unauthorized
- `HTTP 403` : Permission denied

可以在 <https://www.djangoproject.org/api-guide/permissions/> 中找到更多关于权限的信息。

继续编辑 `api/views.py` 文件，为 `CourseEnrollView` 添加一个属性 `permission_classes`：

```
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    permission_classes = (IsAuthenticated,)
    # .....
```

我们为视图加上了 `IsAuthenticated` 权限，意味着只有认证用户可以访问该视图。现在可以尝试给这个视图发一个 POST 请求。

启动站点，然后在系统命令行里输入下列命令：

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

应该会得到下列响应：

```
HTTP/1.1 401 Unauthorized
.....
{"detail": "Authentication credentials were not provided."}
```

结果得到了 401 响应，因为我们没有认证过。现在我们为请求头增加一个已经注册的用户的认证信息，将下列代码中的 `student:password` 替换成你网站中的实际用户名和密码，然后执行命令：

```
curl -i -X POST -u student:password http://127.0.0.1:8000/api/courses/1/enroll/
```

会得到如下响应：

```
HTTP/1.1 200 OK
.....
{"enrolled":true}
```

现在可以到管理后台查看数据库是否已经更新了该用户选课的数据。

1.9 创建视图集和路由

视图集 `Viewsets` 允许对API定义一系列的交互动作，并允许REST框架使用一个 `Router` 对象动态的建立URL。通过使用视图集，可以避免重复编写视图逻辑。REST框架中的视图集涵盖的经典的增删改查动作，包括 `list()`, `create()`, `retrieve()`, `update()`, `partial_update()`, 和 `destroy()`。

为 `Course` 模型创建一个视图集，编辑 `api/views.py` 文件，增加如下代码：

```
from rest_framework import viewsets
from .serializers import CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
```

创建了一个视图集并继承了 `ReadOnlyModelViewSet`，`ReadOnlyModelViewSet` 提供了只读的 `list()` 和 `retrieve()` 方法，可以返回一个对象集合或者单个对象。编辑 `api/urls.py`，为视图集配置URL：

```
from django.urls import path, include
from rest_framework import routers
```

```
from . import views

router = routers.DefaultRouter()
router.register('courses', views.CourseViewSet)

urlpatterns = [
    # .....
    path('', include(router.urls)),
]
```

建立了一个默认的路由对象 `DefaultRouter()`，然后将 `CourseViewSet` 视图注册到路由中，使用了前缀 `courses`，现在这个 `router` 对象就可以为视图集动态的生成URL。

打开 <http://127.0.0.1:8000/api/>，可以看到如下页面：

Api Root

OPTIONS

GET ▾

The default basic root view for DefaultRouter

GET /api/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{  
    "courses": "http://127.0.0.1:8000/api/courses/"  
}
```

这个时候可以访问 <http://127.0.0.1:8000/api/courses/>，就可以得到JSON格式的课程列表。这个路径中的 /courses/ 就是注册路由的时候使用的前缀 courses。

视图集的详细使用可以看 <https://www.djangoproject.org/api-guide/viewsets/>，路由的使用方法可以参考 <https://www.djangoproject.org/api-guide/routers/>。

1.10 为视图集添加额外功能

可以为视图集添加额外功能。让我们来把 CourseEnrollView 变成一个自定义的视图集。编辑 api/views.py 文件，修改 CourseViewSet 为下面这样：

```
from rest_framework.decorators import detail_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

    @detail_route(methods=['post'], authentication_classes=[BasicAuthentication], permission_classes=[IsAuthenticated])
    def enroll(self, request, *args, **kwargs):
        course = self.get_object()
        course.students.add(request.user)
        return Response({'enrolled': True})
```

为视图集增加了一个自定义的方法 `enroll()`，代表为视图集增加的新功能，解释如下：

1. 使用 `detail_router` 装饰器（该装饰器已经被Pycharm提醒要被删除，未来改用 `@action` 装饰器），定义了这是一个在单个对象上执行的功能。
2. 这个装饰器同时还允许添加参数，`methods` 设置为 `['post']` 表示该视图只接受POST请求，然后还设置了验证和权限。
3. 使用 `self.get_object()` 获取当前的 `Course` 对象。
4. 把当前的用户增加到多对多关系中（选课），返回一个自定义的成功响应

然后编辑 `api/urls.py` 文件，去掉下边这一行，因为通过 `@detail_route` 动态配置了新的路由，这一行无需再用：

```
path('courses/<pk>/enroll/', views.CourseEnrollView.as_view(), name='course_enroll'),
```

然后编辑 `api/views.py`，删除 `CourseEnrollView`，因为这个类的功能现在成为视图集的一部分。

现在选课功能的URL由 `router` 自动生成，实际的URL与刚才相同，因为使用了我们自定义的函数名称 `enroll`。

1.11 自定义权限

我们希望只有选了某课程的学生用户才可以访问该课程的全部内容。最好的方式就是自定义一个权限，REST框架（原书为Django，应该为REST）提供了一个 `BasePermission` 类允许重写下列方法：

- `has_permission()`：视图级别的权限检查
- `has_object_permission()`：对象级别的权限检查

这两个方法必须返回 `True` 表示具有权限或 `False` 表示不具有权限。在 `courses/api/` 目录下新建 `permissions.py` 文件，添加下列代码：

```
from rest_framework.permissions import BasePermission

class IsEnrolled(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.students.filter(id=request.user.id).exists()
```

这个 `IsEnrolled` 权限继承了 `BasePermission` 类然后重写了 `has_object_permission` 方法。由于这个方法是基于对象的，所以 `obj` 就是当前的课程。检查当前用户是否在已经选该课的所有用户里。之后就可以使用该权限类了。

1.12 序列化课程内容

现在已经把主题，课程和章节都序列化了。还必须序列化内容。`Content` 模型有一个通用外键关系，可以用于检索所有内容模型。而且我们还为所有内容模型添加了 `render()` 方法。可以使用这些关系和方法，来实现序列化。

编辑 `api/serializers.py` 文件，添加下列代码：

```
from ..models import Content

class ItemRelatedField(serializers.RelatedField):
    def to_representation(self, value):
```

```
        return value.render()

class ContentSerializer(serializers.ModelSerializer):
    item = ItemRelatedField(read_only=True)

    class Meta:
        model = Content
        fields = ['order', 'item']
```

在这段代码里，通过继承 `RelatedField` 定义了一个特别的字段 `ItemRelatedField`，然后重写了 `to_representation()` 方法。然后定义了内容序列化器 `ContentSerializer` 并且指定与原来通用外键同名的 `item` 属性为刚定义的 `ItemRelatedField` 字段。

我们还需要另外一个用于 `Module` 模型的序列化器，其中嵌套这个 `Content` 序列化器；还需要改造 `Course` 序列化器以让其也包含内容输出，编辑 `api/serializers.py` 文件添加下列代码：

```
class ModuleWithContentsSerializer(serializers.ModelSerializer):
    contents = ContentSerializer(many=True)

    class Meta:
        model = Module
        fields = ['order', 'title', 'description', 'contents']

class CourseWithContentsSerializer(serializers.ModelSerializer):
    modules = ModuleWithContentsSerializer(many=True)

    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']
```

这其实是一层一层从内到外嵌套序列化器，由于已经定义了 Content 的序列化器，就建立一个外层的 ModuleWithContent 序列化器，其中设置 contents 字段为 Content 序列化器，再往上一层的 CourseWithContent 序列化器也是类似来嵌套 ModuleWithContent。

再建立一个视图，模仿刚才的 `retrieve()` 行为，但是采用新的序列化器，编辑 `api/views.py` 文件，给 `CourseViewSet` 视图集添加下列代码：

```
from .permissions import IsEnrolled
from .serializers import CourseWithContentsSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    # ...
    @detail_route(methods=['get'],
                  serializer_class=CourseWithContentsSerializer,
                  authentication_classes=[BasicAuthentication],
                  permission_classes=[IsAuthenticated, IsEnrolled])
    def contents(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

这段代码解释如下：

- 使用 `@detail_route` 装饰器来定义该方法是针对一个单独数据对象的
- 该方法只接受GET请求
- 使用了 `CourseWithContentsSerializer` 这个新的序列器，用于返回包含具体内容数据的序列化后输出。
- 添加了用户认证 `IsAuthenticated` 和自定义权限 `IsEnrolled`
- 采用 `ReadOnlyModelViewSet` 提供的 `retrieve()` 方法来返回 `Course` 对象

然后打开 <http://127.0.0.1:8000/api/courses/1/contents/>。如果你的当前用户选了对应的课程，就可以看到课程，章节和内容嵌套渲染后的字符串以JSON的形式显示出来，类似下边这样：

```
{  
    "order": 0,  
    "title": "Introduction to Django",  
    "description": "Brief introduction to the Django Web Framework.",  
    "contents": [  
        {  
            "order": 0,  
            "item": "<p>Meet Django. Django is a high-level Python Web framework...</p>"  
        },  
        {  
            "order": 1,  
            "item": "\n<iframe width=\"480\" height=\"360\" src=\"http://www.youtube.com/embed/bgV39DlmZ2U?wmode=op  
        }  
    ]  
}
```

现在我们就建立了一个简单的符合RESTful风格的API，让网站自动化向外部提供数据。REST框架还可以使用 `ModelViewSet` 来创建和编辑数据对象。关于REST框架中的主要内容在本章都涉及到了，如果对于框架特性还需要详细了解，可以参考REST框架的官方文档： <https://www.django-rest-framework.org/>。

总结

在本章，为其他程序自动化使用本网站的程序，建立了一套API，方便与其他应用程序进行互动。

下一章将讨论如何通过uWSGI和NGINX配置生产环境。你还会学到如何实现一个自定义的中间件以及建立自定义的管理命令。

第十三章 上线

在上一章，为其他程序与我们的Web应用交互创建了RESTful API。本章将学习如何创建生产环境让我们的网站正式上线，主要内容有：

- 配置生产环境
- 创建自定义中间件
- 实现自定义管理命令

1 创建生产环境

现在该将Django项目正式部署到生产环境中了。我们将按照下列步骤将站点部署到生产环境中：

1. 为生产环境配置项目设置
2. 使用PostgreSQL数据库
3. 使用uWSGI和NGINX建立web服务器
4. 管理静态资源
5. 使用SSL加强站点安全管理

1.1 管理用于多个环境的配置

在实际的项目中，很可能要面对不同的环境。一般至少有一个本地开发环境和一个生产环境，也可能有其他环境比如测试环境，预上线环境等。对于不同的环境，有些设置是通用的，有些则因环境而异。让我们将项目设置为可以适合不同环境，又可以保证项目结构不会被改变。

在 `educa/educa/` 目录下建立 `settings` 目录(包)，与 `settings.py` 同级，将 `settings.py` 文件重命名为 `base.py` 然后移动到 `settings` 目录中来，再创建其他文件，`setting/` 目录如下所示：

```
settings/
__init__.py
base.py
local.py
pro.py
```

这些文件用途如下：

- `base.py`：基本的设置文件，包含通用的设置，是原来的`settings.py`
- `local.py`：本地环境的自定义设置
- `pro.py`：生产环境的自定义设置

编辑 `settings/base.py`，找到下列这行：

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

将其替换成本边这行：

```
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(os.path.join(__file__, os.pardir))))
```

由于我们将 `settings.py` 文件又往下级目录放了一级，必须让 `BASE_DIR` 指向正确的路径，所以使用了 `os.pardir` 指向父目录，来让最后的路径依然是原来的项目根目录。

编辑 `settings/local.py`，添加下列代码：

```
from .base import *

DEBUG = True
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

这是代表我们本地环境的配置文件。在其中导入了所有 `base.py` 中的设置内容，然后写了 `DEBUG` 和 `DATABASES` 两个设置，这两个设置会覆盖原来 `base.py` 中的设置，成为本文件中的设置。由于 `DEBUG` 设置和 `DATABASES` 设置在每个配置文件中都会修改，也可以将这两个设置从 `base.py` 中删除。

再来编辑 `settings/pro.py`，如下所示：

```
from .base import *

DEBUG = False
ADMINS = (
    ('Antonio M', 'email@mydomain.com'),
)
ALLOWED_HOSTS = ['*']
DATABASES = {
    'default': {
    }
}
```

这是生产环境的配置文件，来详细看一下其中的内容：

- **DEBUG**：设置 `DEBUG` 为 `False` 是生产环境的强制要求。如果不关闭，会将错误跟踪和敏感配置信息泄露给所有人。
- **ADMINS**：当 `DEBUG` 设置为 `False` 的时候，如果一个视图抛出异常，所有信息会以邮件形式发送到 `ADMINS` 配置中列出的所有人。
需要将其中的信息改成自己的名字和邮箱（还需要配置SMTP服务器）。
- **ALLOWED_HOSTS**：Django只会向这个设置中的地址或者主机名称提供Web服务。这是一个安全手段。我们使用了通配符*表示可以用于所有主机名称或者IP地址。在稍后的配置中会更详细的作出限制。
- **DATABASES**：生产环境的数据库设置，现在留空，后边会进行该设置。由于生产环境的数据库和非生产环境的数据库一般是隔离的，甚至生产环境数据库只有处于生产环境才能访问。所以该项需要单独配置。

在需要面对多种环境时，建立一个基础配置文件并为每种环境编写单独的配置文件。用于具体环境的配置文件继承基础配置并重写与环境相关的配置即可。

由于我们现在没有把配置文件放在原来 `settings.py` 所在的位置，所以无法运行 `manage.py`，必须为其指定 `settings` 模块的所在路径，即使用 `--settings` 参数或者设置环境变量 `DJANGO_SETTINGS_MODULE`。

打开系统命令行窗口输入：

```
export DJANGO_SETTINGS_MODULE=educa.settings.pro
```

这条命令会为当前的会话窗口设置 `DJANGO_SETTINGS_MODULE` 环境变量。如果不想要每次运行shell都执行一遍，可以把这条命令加入到shell配置文件如 `.bashrc` 或者 `.bash_profile` 中。

如果不想要对系统进行任何设置，那么在启动站点的时候必须加上 `--settings` 参数，如下：

```
python manage.py migrate --settings=educa.settings.pro
```

现在我们就为多环境做好了基础设置。

1.2 使用PostgreSQL数据库

在整本书中，我们大部分都使用了Python自带的SQLite数据库，只要在博客全文检索的时候推荐使用了PostgreSQL数据库。SQLite轻量而且易于使用，但对于生产环境而言太过简陋，必须需要一个更加强力的数据库比如PostgreSQL和MySQL或者Oracle。PostgreSQL的安装在第三章中已经介绍过，不再赘述。

让我们为我们的应用创建一个PostgreSQL用户，打开系统命令行输入如下命令：

```
su postgres  
createuser -dP educa
```

系统会提示输入用户密码和权限。输入密码并且给予用户权限，然后使用下列命令建立一个新的数据库：

```
createdb -E utf8 -U educa educa
```

这样就建立好了一个新的数据库并且将其分配给educa用户，之后编辑 `settings/pro.py`，修改数据库的设置如下：

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'educa',  
        'USER': 'educa',  
        'PASSWORD': '*****',  
    }  
}
```

将密码部分替换成为educa用户设置的密码。由于新数据库是空的，运行：

```
python manage.py migrate
```

然后创建一个超级用户：

```
python manage.py createsuperuser
```

译者注，安装PostgreSQL远没有这么简单，尤其是通过第三方程序远程管理PostgreSQL，需要修改PostgreSQL的配置文件，将认证方式修改为md5或者trust，然后启用允许访问的IP，建议查看官方文档和各种安装教程进行配置。

1.3 部署前检查

Django提供了一个 `check` 命令，可以在任何时候检查项目。通常检查过程包括检查所有注册的应用，输出所有错误和警告信息。如果包含 `--deploy` 参数，还会额外执行针对生产环境的检查。

打开系统终端然后输入如下命令进行检查：

```
python manage.py check --deploy
```

译者注：作者这里遗漏了配置文件的路径，应该写成 `python manage.py check --deploy --settings=educa.settings.***`，其中***为base，local或pro

如果站点编写正确的话，会看到没有错误输出，但是会有一些警告信息。这说明站点通过了检查，但这些警告信息应该得到处理，以让站点更加安全。本书不会深入这里的内容，但是要记得在正式部署之前一定要进行部署前检查。

1.4 通过WSGI程序提供Django服务

Django的主要部署平台就是WSGI，WSGI是 [Web Server Gateway Interface](#) 的简称，是基于Python的程序提供Web服务的标准格式。由于Django也是Python程序，也需要通过WSGI对外提供服务。

当通过 `startproject` 命令新建一个项目的时候，Django会在项目目录内新建一个 `wsgi.py`。这个文件包含了一个WSGI可调用函数，为我们的Django应用提供了一个接口。无论是我们之前采用本机8000端口的开发服务器，还是正式生产环境，都需要通过这个接口。关于WSGI的详细知识可以看 <https://wsgi.readthedocs.io/en/latest/> 及Python的 [PEP333](#)。

1.5 安装WSGI

直到本节之前，我们的所有开发都是在django在本地环境运行的开发服务器上进行的。在生产环境中，需要一个真正的web服务器才能部署django服务。

uWSGI是一个非常快的Python应用程序WSGI服务器，使用WSGI标准与Python应用进行通信。uWSGI把HTTP请求翻译成Django程序能够处理的格式。

安装uWSGI：

```
pip install uwsgi==2.0.17
```

在pip安装之后，会built uWSGI（编译安装），需要一个C编译器，比如GCC或者clang，在linux环境下可以输入命令：
`apt-get install build-essential`。

如果是MacOS X，可以通过Homebrew安装，执行命令：`brew install uwsgi`。如果在windows下安装，需要Cygwin
<https://www.cygwin.com>

。推荐在基于UNIX的操作系统上安装uWSGI。

UNIX环境下如果看到Successfully built uwsgi就说明成功安装了uWSGI。关于uWSGI的文档可以在 <https://uwsgi-docs.readthedocs.io/en/latest/> 找到。

1.6 配置uWSGI

可以通过命令行配置uWSGI，打开系统命令行模式，进入 `educa` 项目的根目录，然后输入：

```
sudo uwsgi --module=educa.wsgi:application --env=DJANGO_SETTINGS_MODULE=educa.settings.pro --master --pidfile=/tmp/
```

必须需要 `su` 权限才可以。通过这条命令，为本机上的uWSGI设置了如下的内容：

1. 使用 `educa.wsgi:application` 作为调用接口
2. 载入生产环境的设置文件
3. 使用 `virtualenv` 设置的虚拟环境，注意将 `/home/env/educa/` 替换为实际的虚拟环境所在路径。如果未使用虚拟环境，该配置可以不填。

如果不是在项目目录内执行的上述命令，需要额外加一个参数指定具体的项目目录 `--chdir=/path/to/educa/`，将其中的 `/path/to/educa/` 替换成 `educa` 的项目路径。

通过浏览器访问 <http://127.0.0.1:8000/>（无需启动django服务），可以看到站点内容显示了出来，但没有任何CSS样式，也无法显示图片，这是因为还没有配置uWSGI来提供静态文件服务。

uWSGI允许使用一个 `.ini` 配置文件进行自定义配置，比使用命令行要方便很多。在 `educa` 项目根目录下建立：

```
config/
  uwsgi.ini
```

编辑 `uwsgi.ini`，添加如下代码：

```
[uwsgi]
# variables
projectname = educa
base = /home/projects/educa

# configuration
master = true
virtualenv = /home/env/%(projectname)
pythonpath = %(base)
chdir = %(base)
env = DJANGO_SETTINGS_MODULE=%(projectname).settings.pro
module = educa.wsgi:application
socket = /tmp/%(projectname).sock
```

在这个 `.ini` 文件里我们定义了两个变量：

- `projectname`：Django项目的名称，是 `educa`
- `base`：`educa` 项目的绝对路径，将其替换成实际项目路径

上边定义的这两个变量是自定义变量，还可以定义任意其他变量，只要不和内置的名称冲突。接下来是具体设置的解释：

- `master`：表示启用主进程
- `virtualenv`：虚拟环境地址，将其替换成实际的路径所在（不包含 `bin/activate`）
- `pythonpath`：加入到Python PATH中的地址，一般就是项目的根目录
- `chdir`：项目的实际地址，uWSGI会在加载应用之前将工作目录变更到这个路径
- `env`：环境变量，设置为 `DJANGO_SETTINGS_MODULE`，具体路径指向生产环境的配置文件
- `module`：要使用的WSGI模块，指向项目中的 `wsgi.py` 中的调用函数。`application` 是该函数在项目中默认的命名。
- `socket`：绑定该服务的套接字。（是一个文件套接字，用于与NGINX通信）

其中的 `socket` 套接字是用于和第三方路由软件进行通信，比如NGINX。命令行模式中我们使用的 `--http 127.0.0.1:8000` 指的是让uWSGI自己接受HTTP请求并自己负责路由这些请求。我们需要把uWSGI作为socket启动（在 `.ini` 文件设置中并没有设置 `--http` 参数），因为我们要使用NGINX作为我们的web服务器，NGINX通过刚才设置的文件套接字与uWSGI进行通信。

关于uWSGI的详细设置可以看 <https://uwsgi-docs.readthedocs.io/en/latest/Options.html>。

现在可以通过使用配置文件来启动uWSGI（先关闭原来运行的uWSGI服务）：

```
uwsgi --ini config/uwsgi.ini
```

这样运行之后，可以发现暂时无法通过浏览器访问 <http://127.0.0.1:8000/>，因为此时uWSGI监听文件套接字而不是HTTP端口，我们还需要继续完善生产环境配置。

1.7 配置uWSGI

当启动一个Web服务的时候，很显然必须提供动态的内容服务，但也需要静态的文件服务，比如CSS，JavaScript文件，图像等。如果用uWSGI来管理静态文件，会为HTTP请求增加不必要的开销，所以最好在uWSGI之前加一个Web服务，比如NGINX。

NGINX是一个高并发，低内存占用的Web服务端，也具有反向代理功能，即接受一个HTTP请求，然后把这个请求路由给不同的后端。通常来说，你需要一个web服务端如NGINX，用于快速高效的提供静态文件，然后把动态的请求转发给uWSGI。通过使用NGINX，还可以设置其反向代理功能从而更好的提供web服务。

安装NGINX可以使用下列命令：

```
sudo apt-get install nginx
```

如果使用MacOS X，可以通过 `brew install nginx` 来安装。Windows下的NGINX可以通过 <https://nginx.org/en/download.html> 下载。

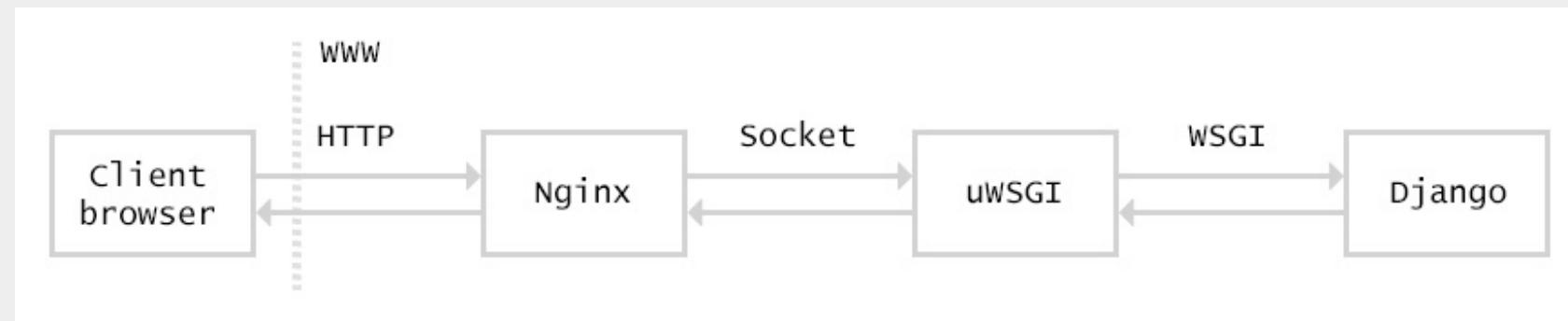
译者注：安装NGINX后不会立刻启动，译者使用的Centos 7.5 1804还需要启动NGINX服务和开机启动：

```
systemctl start nginx.service  
systemctl enable nginx.service
```

正常情况下在启动NGINX之后，直接访问本机IP地址，可以看到NGINX欢迎页面，表示基础配置成功运行，之后可以先停用NGINX服务，以配置生产环境。

1.8 生产环境

下面的图表示了我们最终配置的生产环境的结构：



当一个浏览器发起一个HTTP请求的时候，发生如下事情：

1. NGINX接收HTTP请求
2. 如果请求静态文件，NGINX直接提供服务。如果请求动态页面，NGINX通过SOCKET与uWSGI通信，将请求转交给uWSGI处理
3. uWSGI将请求转交给Django后端进行处理，返回的响应被传递给NGINX，NGINX再发回给浏览器。

1.9 配置NGINX

在 config/ 目录下创建 nginx.conf 文件，在其中添加如下代码：

```
# the upstream component nginx needs to connect to
upstream educa {
    server unix:///tmp/educa.sock;
}
server {
    listen 80;
    server_name www.educaproject.com educaproject.com;
    location / {
        include /etc/nginx/uwsgi_params;
        uwsgi_pass educa;
    }
}
```

这是NGINX的基础配置。我们建立了一个upstream名叫 `educa`，指定了uWSGI使用的socket名称，然后使用 `server` 指令，其中的设置有：

- `listen 80` 表示让NGINX监听 80 端口
- 设置主机名为 `www.educaproject.com` 和 `educaproject.com`，NGINX会为这两个主机地址提供服务
- 配置 `location` 参数，将所有在 '/' 路径下的URL转发给上边的upstream `educa`，也就是uWSGI的socket进行处理。还把NGINX自带的关于和uwsgi协同工作的参数设置也包含进去。

NGINX还有很多复杂的设置，文档可以参考 <https://nginx.org/en/docs/>。

NGINX主要的设置文件位于 `/etc/nginx/nginx.conf`，该文件包含 `/etc/nginx/sites-enabled/` 下的所有配置文件。为了让NGINX使用我们刚才编写的配置文件，打开系统命令行窗口建立一个软连接：

```
sudo ln -s /home/projects/educa/config/nginx.conf /etc/nginx/sites-enabled/educa.conf
```

将其中的 `/home/projects/educa/` 替换成实际的绝对路径。注意，这里如果没有 `/sites-enabled/` 目录，要先手工建立。

如果还没有运行uWSGI，打开系统命令行窗口，在 `educa` 项目根目录先运行uWSGI：

```
uwsgi --ini config/uwsgi.ini
```

当前窗口会被uWSGI占用，再开一个命令行窗口，然后执行：

```
service nginx start
```

由于我们使用了自定义的域名，还必须修改 `/etc/hosts`，添加如下两行：

```
127.0.0.1 educaproject.com  
127.0.0.1 www.educaproject.com
```

这样我们就把这两个域名都路由到本地回环地址上，由于我们是从本机访问本机，所以要更改HOSTS，实际生产环境不必做本步修改，因为生产环境会有固定的IP，域名和对应的DNS解析。

打开浏览器，输入 <http://educaproject.com/>，应该可以看到站点了，但是所有的静态文件依然没有被加载，没关系，即将完成生产环境的配置。

如果系统是Centos 7，这里显示502错误，查看 `/var/log/nginx/error.log`，如果其中的错误是 `[crit] 4036#4036: *1 connect() to unix:///tmp/educa.sock failed (13: Permission denied)`，就先执行 `/usr/sbin/sestatus` 查看SELINUX的状态，如果为开启，就编辑SELINUX的设置，将其关闭，如下：

```
vi /etc/selinux/config  
  
#SELINUX=enforcing  
SELINUX=disabled
```

之后 *reboot* 重启系统才行。之后应该就可以正常显示站点了。

之后为了安全起见，到 `settings/pro.py` 中，修改 `ALLOWED_HOSTS` 设置为NGINX配置文件中的两个域名：

```
ALLOWED_HOSTS = ['educaproject.com', 'www.educaproject.com']
```

现在Django就只为这两个主机名提供服务了。关于 `ALLOWED_HOSTS` 的更多信息可以看
<https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>。

1.10 让NGINX提供静态文件和媒体资源服务

NGINX提供静态文件的速度很快。刚才我们把所有的地址转发，都交给了uWSGI，现在要将所有的静态文件通过NGINX提供服务，对于我们站点来说，就是把所有的CSS JS文件和用户上传的媒体文件都交给NGINX来代理。

编辑 `settings/base.py`，增加下边一行：

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

这行表示存放站点静态文件的地址，还记得之前学习过使用 `python manage.py collectstatic` 吗？现在就需要将所有的静态文件收集过来放在此目录中，在命令行中输入：

```
python manage.py collectstatic --settings=educa.settings.pro
```

注意，原书的命令缺少了 `--settings=educa.settings.pro`

可以看到下列输出：

```
160 static files copied to '/educa/static'.
```

静态文件目录设置好了，现在需要将这个目录设置到NGINX中，编辑 `config/nginx.conf`，在 `server` 指令后的大括号中增加下列内容：

```
location /static/ {  
    alias /home/projects/educa/static/;  
}  
location /media/ {  
    alias /home/projects/educa/media/;  
}
```

将其中的 `/home/projects/educa/static/` 和 `/home/projects/educa/media/` 替换成你项目的实际 `static` 和 `media` 目录的绝对路径。这两个参数解释如下：

- `/static/`：这个路径是Django中设置的 `STATIC_URL`，表示当NGINX看到 `/static/` 的路径请求的时候，就到这个设置对应的路径中寻找所需文件。
- `/media/`：这个路径是Django中设置的 `MEDIA_URL` 路径，表示当NGINX看到 `/media/` 的路径请求的时候，就到这个设置对应的路径中寻找所需文件。

重新启动NGINX服务，以便让配置文件生效：

```
service nginx reload
```

在浏览器中打开 <http://educaproject.com/>，现在可以看到整个站点包含静态资源都正确的显示了。对于站点的静态文件请求，NGINX将绕开uWSGI，把文件直接返回给浏览器。

现在生产环境就初步配置完毕。整个站点现在可以说运行在生产环境之下了。

1.11 使用SSL安全连接

在配置完初步的生产环境之后，下一个话题是站点的安全性。[Secure Sockets Layer](#) 现在逐渐成为提供Web安全连接服务的规范。强烈建议对于正式的网站使用HTTPS协议，现在就在NGINX中配置SSL认证来让站点变得更加安全。

1.11.1 创建一个SSL认证

在 `educa` 项目根目录下建立一个 `ssl` 目录，然后通过 `openssl` 生成我们的SSL证书：

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ssl/educa.key -out ssl/educa.crt
```

用这条命令生成一个365天有效的2048位的SSL证书，然后系统会提示输入一些信息：

```
Country Name (2 letter code) [AU]:  
State or Province Name (full name) [Some-State]:  
Locality Name (eg, city) []:  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:  
Organizational Unit Name (eg, section) []:  
Common Name (e.g. server FQDN or YOUR name) []: educaproject.com  
Email Address []: email@domain.com
```

这其中最关键的是 `Common Name`，必须将主机域名名称输入：这里使用 `educaproject.com`

之后会在 `ssl/` 目录下生成两个文件，`educa.key` 是私钥，`educa.crt` 是实际的SSL证书。

1.11.2 配置NGINX使用SSL

编辑 config/nginx.conf，在 server 设置中加入下列内容：

```
server {  
listen 80;  
listen 443 ssl;  
ssl_certificate /home/projects/educa/ssl/educa.crt;  
ssl_certificate_key /home/projects/educa/ssl/educa.key;  
server_name www.educaproject.com educaproject.com;  
# ...  
}
```

将其中的路径都修改为SSL证书所在的实际绝对路径。

这么设置之后，NGINX将同时监听 80 端口（HTTP协议）和 443 端口（HTTPS协议），然后指定了SSL的验证信息 `ssl_certificate` 与对应的密钥 `ssl_certificate_key`。

现在重新启动NGINX服务，访问 <https://educaproject.com/>，会看到类似如下提示：



This Connection is Untrusted

You have asked Firefox to connect securely to educaproject.com, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

[Get me out of here!](#)

► Technical Details

▼ I Understand the Risks

If you understand what's going on, you can tell Firefox to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

[Add Exception...](#)

这个提示因浏览器而异。意思是警告当前站点并没有使用一个值得信任的验证方式，浏览器无法确定该站点安全与否。这是因为我们使用的SSL证书是由我们自行签发的，而不是从一个受信任的机构（Certification Authority）获得的证书。当我们有了实际的公开域名之后，就可以向一个受信任的证书颁发机构申请一个SSL证书，这样浏览器就能识别该站点的HTTPS认证。

如果想为实际的站点申请证书，可以使用Linux基金会Linux Foundation的 **Let's Encrypt** 项目。这是一个致力于免费获得和更新SSL证书的计划，该计划的站点在 <https://letsencrypt.org/>。

点击 "Add Exception" 按钮可以让浏览器知道可以信任该站点，这时浏览器的显示可能如下：



点击小锁按钮，就可以看到SSL的详细信息。

译者注：这里也因浏览器而异，有的浏览器依旧会提示证书不可信或者存在问题，毕竟这个证书是我们自行签发的。

1.11.3 配置Django使用SSL

Django也有针对SSL的配置，编辑 `settings/pro.py`，增加下边的代码：

```
SECURE_SSL_REDIRECT = True  
CSRF_COOKIE_SECURE = True
```

这两个设置的含义如下：

- `SECURE_SSL_REDIRECT`：是否所有的HTTP请求都必须被重定向到HTTPS
- `CSRF_COOKIE_SECURE`：是否建立加密cookie防止CSRF攻击

现在我们就配置好了一个高效的提供Web服务的生产环境。

2 自定义中间件

在之前我们已经了解了中间件 MIDDLEWARE 的设置，该设置包含项目中所有使用到的中间件。关于中间件，可以认为其是一个底层的插件系统，为在请求/响应的过程中提供 [钩子](#)。每一个中间件都负责一个特定的行为，会在HTTP请求和响应的过程中得到执行。

注意不要添加开销非常大的中间件，因为中间件会在项目的所有请求和响应的过程中被执行。

当一个HTTP请求进来的时候，中间件会按照其在 MIDDLEWARE 设置中从上到下的顺序执行，当HTTP响应被生成且发送的过程中，中间件会按照设置中从下到上的顺序执行。

一个符合标准的函数可以作为一个中间件被注册在 `settings.py` 中。类似下边的函数就可以作为一个中间件：

```
def my_middleware(get_response):
    def middleware(request):
        # 对于每个HTTP请求，在视图和之后的中间件执行之前执行的代码
        response = get_response(request)
        # 对于每个HTTP请求和响应，在视图执行之后执行的代码
        return response
    return middleware
```

一个中间件工厂函数接受一个 `get_response` 可调用对象，然后返回一个中间件函数。一个中间件接受一个请求然后返回一个响应，类似于视图。这里的 `get_response` 可以是下一个中间件，如果自己就是中间件列表中的最后一个，也可以是一个视图名称。

如果任何一个中间件在尚未调用 `get_response` 这个可调用对象之前就返回了一个响应，这个时候就会短路整个中间件链条的处理：其后的中间件不再被执行，这个响应开始从同级的中间件向上返回。

所以 `MIDDLEWARE` 设置中的中间件顺序非常重要，因为中间件依赖于上下中间件的数据进行工作。

在向 `MIDDLEWARE` 中添加一个中间件时必须注意将其放置在正确的位置，反复强调，中间件在HTTP请求进来的时候从上到下执行，HTTP响应发出的时候从下到上执行。

原书在这里只是比较简单的说了一下执行顺序，详细的中间件执行顺序请参考 [Django进阶-中间件](#) 以及 <https://docs.djangoproject.com/en/2.0/topics/http/middleware/>。

2.1 创建二级域名中间件

我们来创建一个自定义中间件，用于通过一个自定义的二级域名来访问课程资源。例如：某个显示课程的URL：

`https://educaproject.com/course/django/`，可以通过一个二级域名 `django.educaproject.com` 来访问。这样用户就可以使用二级域名作为快捷方式快速访问课程，也比较容易记忆该路径。所有发往这个二级域名的请求，都会被重定向到实际的 `educaproject.com/course/django/` 这个URL。

与视图，模型，表单等组件一样，中间件也可以写在项目的任何位置。推荐在应用目录内建立 `middleware.py` 文件来编写中间件。

在 `courses` 应用目录内创建 `middleware.py` 文件，并编写如下代码：

```
from django.urls import reverse
from django.shortcuts import get_object_or_404, redirect
from .models import Course

def subdomain_course_middleware(get_response):
    """
    为课程提供二级域名
    """

    def middleware(request):
        pass
```

```
host_parts = request.get_host().split('.')
if len(host_parts) > 2 and host_parts[0] != 'www':
    # 通过指定的二级域名查询课程对象
    course = get_object_or_404(Course, slug=host_parts[0])
    course_url = reverse('course_detail', args=[course.slug])
    # 将二级域名请求重定向至实际的URL
    url = '{}://{}{}'.format(request.scheme, '.'.join(host_parts[1:]), course_url)
    return redirect(url)
response = get_response(request)
return response
return middleware
```

当一个HTTP请求进来的时候，这个中间件执行如下任务：

1. 取得这个HTTP请求中的域名，然后将其分割成几部分；例如 `mycourse.educaproject.com` 会被分割得到一个列表 `['mycourse', 'educaproject', 'com']`
2. 检查这个域名是否包含二级域名，判断分割后的域名是否包含多于2个元素。如果包含，就取出第一个元素也就是二级域名，如果这个域名不是www，那就通过根据 `slug` 查询并取得该课程对象。
3. 如果找不到对应的课程，就返回404错误；如果找到了，就重定向到课程对象对应的规范化URL。

编辑 `settings/base.py`，把自定义中间件添加到 `MIDDLEWARE` 设置中：

```
MIDDLEWARE = [
    # .....
    'courses.middleware.subdomain_course_middleware',
]
```

还需要看一下 `ALLOWED_HOSTS` 中的域名设置，这里我们将其设置为可以是任何 `eduproject.com` 的二级域名：

```
ALLOWED_HOSTS = ['.educaproject.com']
```

`ALLOWED_HOSTS` 中以一个`.`开始的域名，例如`.educaproject.com`，会匹配`educaproject.com`及所有的`educaproject.com`的二级域名，比如`course.educaproject.com`和`django.educaproject.com`。

2.3 配置NGINX的二级域名

编辑`config/nginx.conf`，将以下这行：

```
server_name www.educaproject.com educaproject.com;
```

修改成：

```
server_name *.educaproject.com educaproject.com;
```

通过增加通配符设置，让NGINX也可以代理所有的二级域名，为了测试中间件，还必须在`etc/hosts`中配置相关内容，比如如果要测试二级域名`django.educaproject.com`，需要增加一行：

```
127.0.0.1 django.educaproject.com
```

然后启动站点到<https://django.educaproject.com/>，可以发现中间件现在将其重定向到<https://educaproject.com/course/django/>

3 实现自定义的管理命令

Django允许应用向`manage.py`管理工具中注册自定义的管理命令。所谓管理命令就是通过`manage.py`使用的指令，例如，我们曾经使用在第9章使用过`makemessages`和`compilemessages`命令。

一个管理命令由一个Python模块组成，这个模块里包含一个 `Command` 类，这个 `Command` 类继承 `django.core.management.base.BaseCommand` 或者 `BaseCommand` 的子类。我们可以创建一个简单的包含参数和选项的自定义命令。

对于每个在 `INSTALLED_APPS` 内注册的应用，Django会在应用目录下边的 `management/commands/` 目录下搜索管理命令，搜索到的每个命令模块，都会被注册成为一个同名的命令。

更多自定义管理命令的信息可以查看 <https://docs.djangoproject.com/en/2.0/howto/custom-management-commands/>。

我们准备来创建一个提醒学生至少选一个课程的命令。这个命令会向所有已经注册超过一定时间，但还没有选任何一门课程的学生发送一封邮件。

在 `student` 应用下建立如下的目录和文件结构：

```
management/
__init__.py
commands/
__init__.py
enroll_reminder.py
```

编辑 `enroll_reminder.py`，添加下列代码：

```
import datetime
from django.conf import settings
from django.core.management.base import BaseCommand
from django.core.mail import send_mass_mail
from django.contrib.auth.models import User
from django.db.models import Count
```

```
class Command(BaseCommand):
    help = 'Sends an e-mail reminder to users registered more than N days that are not enrolled into any courses yet'

    def add_arguments(self, parser):
        parser.add_argument('--days', dest='days', type=int)

    def handle(self, *args, **options):
        emails = []
        subject = 'Enroll in a course'
        date_joined = datetime.date.today() - datetime.timedelta(days=options['days'])
        users = User.objects.annotate(course_count=Count('courses_joined')).filter(course_count=0,
                                                                                     date_joined__lte=date_joined)
        for user in users:
            message = "Dear {},\n\n We noticed that you didn't enroll in any courses yet. What are you waiting for?".format(
                user.first_name)
            emails.append((subject, message, settings.DEFAULT_FROM_EMAIL, [user.email]))
        send_mass_mail(emails)
        self.stdout.write('Sent {} reminders'.format(len(emails)))
```

这是 `enroll_reminder` 命令，解释如下：

- `Command` 类继承 `BaseCommand` 类
- `Command` 类包含一个 `help` 属性，为命令提供帮助信息，运行 `python manage.py help enroll_reminder` 就可以看到这段信息。
- `add_arguments()` 用来设置可用的参数，这里设置了 `--days` 参数，指定其类型为整型。运行命令时这个参数用于指定天数，方便筛选出要向其发送邮件的学生。
- `handle()` 方法定义命令的实际业务逻辑。这里从命令行中获取解析后的 `days` 属性，然后查询注册时间超过该天数的用户，再通过分组计算这些用户的选课数量，从中选出未选课的用户。然后使用一个 `emails` 列表记录所有需要发送的邮件，最后通过 `send_mass_mail()` 方法发送邮件，这样可以使用一个SMTP链接发送大量邮件，而不用每发一次邮件就新开一个SMTP链接。

编写好上述代码后，打开系统命令行来运行命令：

```
python manage.py enroll_reminder --days=20
```

如果还没有配置SMTP服务器，可以参考第二章中的内容。如果确实没有SMTP服务器，可以在 `settings.py` 中加上：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

以让Django将邮件内容显示在控制台而不实际发送邮件。

还可以通过系统让这个命令每天早上8点运行，如果使用了基于UNIX的操作系统，可以打开系统命令行模式，输入 `crontab -e` 来编辑 `crontab`，在其中增加下边这行：

```
0 8 * * * python /path/to/educa/manage.py enroll_reminder --days=20 --settings=educa.settings.pro
```

将其中的 `/path/to/educa/manage.py` 替换成实际的 `manage.py` 所在的绝对路径。如果不熟悉cron的使用，可以参考 <http://www.unixgeeks.org/security/newbie/unix/cron-1.html>。

如果使用的是Windows，可以使用系统的计划任务功能，具体可以参考 <https://docs.microsoft.com/zh-cn/windows/desktop/TaskSchd/task-scheduler-start-page>。

还有一个方法是使用Celery定期执行任务。我们在第7章使用过Celery，可以使用Celery beat scheduler来建立定期执行的异步任务，具体可以参考 <https://celery.readthedocs.io/en/latest/userguide/periodic-tasks.html>。

对于想通过cron或者Windows的计划任务执行的单独脚本，都可以通过自定义管理命令的方式来进行。

Django还提供了一个使用Python执行管理命令的方法，可以通过Python代码来运行管理命令，例如：

```
from django.core import management  
management.call_command('enroll_reminder', days=20)
```

程序在执行到这里的时候，就会去运行这个命令。现在我们就可以为自己的应用定制管理命令并且计划运行了。

总结

这一章里使用uWSGI和NGINX配置完成了生产环境，还实现了自定义中间件和管理命令。

到这里本书已经结束。祝贺你，本书通过创建实际的项目和将其他软件与Django集成的方式，指引你学习使用Django建立Web应用所需的技能。无论一个简单的项目原型还是大型的Web应用，你现在都具备使用Django创建它们的能力。

祝你未来的Django之旅愉快！

其他感兴趣的书

如果你发现本书很有用，你可能还会对下列书籍感兴趣。

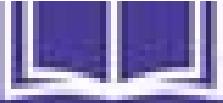


Python Programming Blueprints

Build nine projects by leveraging powerful frameworks such as Flask, Nameko, and Django



Dive into

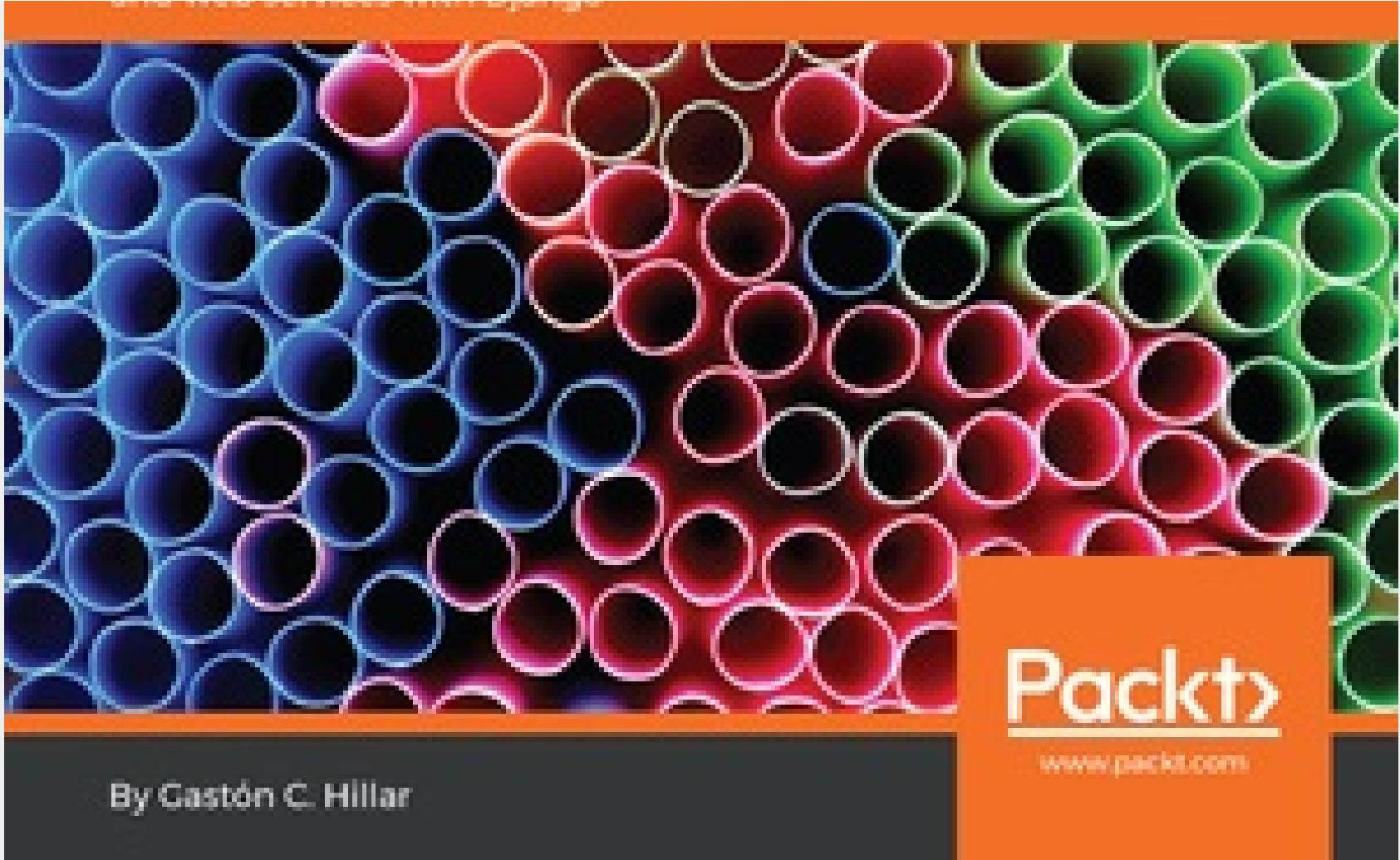


POCKET

[Python Programming Blueprints](#)

Django RESTful Web Services

The easiest way to build Python RESTful APIs
and web services with Django



[Django RESTful Web Services](#)