

# 目 录

致谢

[README](#)

初识 Django

快速安装指南

创建你的第一个 Django 项目，第一部分

创建你的第一个 Django 项目，第二部分

创建你的第一个 Django 项目，第三部分

创建你的第一个 Django 项目，第四部分

创建你的第一个 Django 项目，第五部分

创建你的第一个 Django 项目，第六部分

创建你的第一个 Django 项目，第七部分

进阶内容：编写可重用的应用

接下来如何学习？

编写你的第一个 Django 补丁

# 致谢

当前文档《Django 官方教程翻译项目》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-25。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN)，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/Django-intro-zh>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# README

- [Django-intro-zh Django 官方教程翻译项目](#)
  - [目前版本：1.11](#)
  - [官方文档目录](#)
  - [中文版文档](#)
  - [翻译进度](#)
  - [我也想一起翻译](#)

## Django-intro-zh Django 官方教程翻译项目

---

### 目前版本：1.11

---

这个项目的目的是将 [Django 官方教程的 intro 部分](#) 翻译成中文版。

起因是在看完 Django Book 之后觉得有点过时，随后看了官方文档，还是觉得官方文档写的比较通俗易懂。为了方便更多想要学习 Django 的人（顺便翻译一遍也能更深入的理解文档），就有了这个项目。

这一项目离不开辛勤帮助翻译的[小伙伴们](#)，没有他们这个项目也无法完成。

特别感谢 [@Zoctan](#)，将翻译版本从 1.8 升级到了 1.11 ([pr](#))，跟上了 Django 的发展。

### 官方文档目录

---

官方文档 [在此](#)。

以下是官方文档的目录：

- [Django at a glance](#)
- [Quick install guide](#)
- [Writing your first Django app, part 1](#)
- [Writing your first Django app, part 2](#)
- [Writing your first Django app, part 3](#)
- [Writing your first Django app, part 4](#)
- [Writing your first Django app, part 5](#)
- [Writing your first Django app, part 6](#)
- [Writing your first Django app, part 7](#)
- [Advanced tutorial: How to write reusable apps](#)
- [What to read next](#)

- [Writing your first patch for Django](#)

## 中文版文档

中文版文档请到 [github page](#) 查看，下方目录在 [github page](#) 点击才有效。

- [初识 Django](#)
- [快速安装指南](#)
- [创建你的第一个 Django 项目， 第一部分](#)
- [创建你的第一个 Django 项目， 第二部分](#)
- [创建你的第一个 Django 项目， 第三部分](#)
- [创建你的第一个 Django 项目， 第四部分](#)
- [创建你的第一个 Django 项目， 第五部分](#)
- [创建你的第一个 Django 项目， 第六部分](#)
- [创建你的第一个 Django 项目， 第七部分](#)
- [进阶内容：编写可重用的应用](#)
- [接下来如何学习？](#)
- [编写你的第一个 Django 补丁](#)

## 翻译进度

1. <a href="#">Django</a> at a glance	[=====] 100/100
2. <a href="#">Quick</a> install guide	[=====] 100/100
3. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 1	[=====] 100/100
4. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 2	[=====] 100/100
5. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 3	[=====] 100/100
6. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 4	[=====] 100/100
7. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 5	[=====] 100/100
8. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 6	[=====] 100/100
9. <a href="#">Writing</a> your first <a href="#">Django</a> app, part 7	[=====] 100/100
10. <a href="#">Advanced</a> tutorial: <a href="#">How</a> to write reusable apps	[=====] 100/100
11. <a href="#">What</a> to read next	[=====] 100/100
12. <a href="#">Writing</a> your first patch for <a href="#">Django</a>	[=====] 100/100

## 我也想一起翻译

欢迎一切有时间有能力的小伙伴一起来翻译。

流程：

1. 到[任务大厅](#)认领任务。

2. 任务申请被接受后，Fork 本项目。
3. 仔细阅读 正在讨论中的[画风设定集](#)（我改改改改的累死了啊啊啊啊）。
4. 如果对上述草稿有话想说，请参与讨论。
5. 可以开始翻译咯。
6. PR。

# 初识 Django

- [初识 Django](#)
  - [设计模型](#)
  - [创建模型](#)
  - [享用便捷的 API](#)
  - [动态生成的管理页面：并非徒有其表](#)
  - [规划 URL](#)
  - [编写视图](#)
  - [设计模板](#)
  - [这只是冰山一角](#)

## 初识 Django

Django 最初被设计用于具有快速开发需求的新闻类站点，目的是要实现简单快捷的网站开发。以下内容简要介绍了如何使用 Django 实现一个数据库驱动的 Web 应用。

为了让您充分理解 Django 的工作原理，这份文档为您详细描述了相关的技术细节，不过这并不是是一份入门教程或者是参考文档（我们当然也为您准备了这些）。如果您想要马上开始一个项目，可以从 [实例教程（zh）](#) 开始入手，或者直接开始阅读详细的[参考文档](#)。

## 设计模型

Django 无需数据库就可以使用，它提供了[对象关系映射器（ORM）](#)。通过此技术，你可以使用 Python 代码来描述数据库结构。

[数据模型语法](#)提供了很多方法来描述你的数据，这解决了多年来在数据库模式中的难题。以下是一个简明的例子：

```
1. # mysite/news/models.py
2.
3. from django.db import models
4.
5. class Reporter(models.Model):
6.     full_name = models.CharField(max_length=70)
7.
8.     def __str__(self):          # Python 2 下请使用 __unicode__
9.         return self.full_name
10.
11. class Article(models.Model):
12.     pub_date = models.DateField()
13.     headline = models.CharField(max_length=200)
```

```

14.     content = models.TextField()
15.     reporter = models.ForeignKey(Reporter)
16.
17.     def __str__(self):          # Python 2 下请使用 __unicode__
18.         return self.headline

```

## 创建模型

然后，运行 Django 命令行工具来创建数据库表。

```
1. $ python manage.py migrate
```

**migrate** 命令会查找所有可用的模型，如果数据库中没有与之对应的表，则会为其自动创建。Django 也提供了其他[更丰富的控制方式](#)。

## 享用便捷的 API

接下来，你就可以使用一套便捷而丰富的 **Python API** 用于访问你的数据。这些 API 是自动即时创建的，你不用编写其他任何代码。

```

1. # 从我们的 news 应用里导入模型（译注：记者和文章模型）。
2. >>> from news.models import Reporter, Article
3.
4. # 现在系统中还没有记者。
5. >>> Reporter.objects.all()
6. <QuerySet []>
7.
8. # 创建一个 Reporter 对象。
9. >>> r = Reporter(full_name='John Smith')
10.
11. # 将对象保存到数据库。save()方法需要被显式调用。
12. >>> r.save()
13.
14. # 现在它有了id。
15. >>> r.id
16. 1
17.
18. # 现在这个新人记者已经在数据库里了。
19. >>> Reporter.objects.all()
20. <QuerySet [<Reporter: John Smith>]>
21.
22. # 字段被表示成 Python 对象中的属性。
23. >>> r.full_name

```

```

24. 'John Smith'
25.
26. # Django 提供了一套丰富的数据库查找 API。
27. >>> Reporter.objects.get(id=1)
28. <Reporter: John Smith>
29. >>> Reporter.objects.get(full_name__startswith='John')
30. <Reporter: John Smith>
31. >>> Reporter.objects.get(full_name__contains='mith')
32. <Reporter: John Smith>
33. >>> Reporter.objects.get(id=2)
34. Traceback (most recent call last):
35. ...
36. DoesNotExist: Reporter matching query does not exist.
37.
38. # 创建一篇文章。（译注：reporter放入了前面创建的r对象）
39. >>> from datetime import date
40. >>> a = Article(pub_date=date.today(), headline='Django is cool',
41. ...             content='Yeah.', reporter=r)
42. >>> a.save()
43.
44. # 现在文章在数据库里了。
45. >>> Article.objects.all()
46. <QuerySet [<Article: Django is cool>]>
47.
48. # 通过 Article 对象可以访问与其关联的 Reporter 对象。
49. >>> r = a.reporter
50. >>> r.full_name
51. 'John Smith'
52.
53. # 反之亦然：Reporter 对象也可以访问到与其关联的 Article 对象。
54. >>> r.article_set.all()
55. <QuerySet [<Article: Django is cool>]>
56.
57. # 这个 API 接受你所需的查询条件，并在后台高效地执行 JOIN 数据库操作。
58. # 这个操作会查找所有以 "John" 开头的记者发表的文章。
59. >>> Article.objects.filter(reporter__full_name__startswith='John')
60. <QuerySet [<Article: Django is cool>]>
61.
62. # 通过变更对象的属性值来修改对象，然后调用save()存入数据库。
63. >>> r.full_name = 'Billy Goat'
64. >>> r.save()
65.
66. # 用 delete() 来删除一个对象。
67. >>> r.delete()

```

## 动态生成的管理页面：并非徒有其表



当你的模型完成定义，Django 就会自动生成一个专业的生产级[管理页面](#) - 一个可以让已认证用户进行添加、更改和删除对象的 Web 站点。你只需简单的在 admin 站点上注册你的模型即可。

```
1. # mysite/news/models.py
2.
3. from django.db import models
4.
5. class Article(models.Model):
6.     pub_date = models.DateField()
7.     headline = models.CharField(max_length=200)
8.     content = models.TextField()
9.     reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

```
1. # mysite/news/admin.py
2.
3. from django.contrib import admin
4.
5. from . import models
6.
7. admin.site.register(models.Article)
```

这样设计所遵循的理念是，站点编辑人员可以是你的员工、你的客户、或者就是你自己——而你大概不会乐意去废半天劲创建一个只有内容管理功能的后台管理界面。

创建 Django 应用的典型流程是：先建立数据模型，然后搭建管理站点，尽可能快的跑起来。那样你的团队（或者客户）就可以向网站里填充数据了。后面我们会谈到如何展示这些数据。

## 规划 URL

简洁优雅的 URL 规划对于一个高质量 Web 应用来说至关重要。Django 推崇优美的 URL 设计，所以不要把诸如 `.php` 和 `.asp` 之类的冗余的后缀放到 URL 里。

为了设计你自己的 URL，你需要创建一个叫做 `URLconf` 的 Python 模块。一张包含 URL 匹配模式和 Python 回调函数之间的映射表。URLconf 也有利于将 Python 代码与 URL 解耦合（译注：使各个模块分离，独立）。

下面这个 URLconf 适用于前面 **Reporter/Article** 的例子：

```
1. # mysite/news/urls.py
2.
3. from django.conf.urls import url
4.
5. from . import views
6.
```

```

7. urlpatterns = [
8.     url(r'^articles/([0-9]{4})/$', views.year_archive),
9.     url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
10.    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
11. ]

```

上面的代码将 URL 的[正则表达式](#)映射到 views 里的回调函数。正则表达式通过括号来提取 URL 中的参数值。当一个用户请求页面时，Django 会顺序遍历这些匹配模式，直至模式和请求的 URL 成功匹配。（如果全部模式都无法匹配，Django 会返回一个404视图。）这个过程会在瞬间完成，因为这些正则表达式在启动时就被编译了。

一旦其中一个正则表达式匹配成功，Django 就会导入并调用指定的视图——那是一个简单的 Python 函数。视图会被传进一个请求（request）对象——其中包含了请求元数据——和正则表达式匹配到的那些参数值。

比如，如果用户请求了“/articles/2005/05/39323/”这样的 URL，Django 就会这样调用函数：`news.views.article_detail(request, '2005', '05', '39323')`。

## 编写视图

视图函数的执行结果只可能有两种：返回一个包含请求页面内容的 [HttpResponse](#) 对象；或者是抛出 [Http404](#) 这类异常。至于视图接下来还要做什么则由你决定。

通常来说，一个视图的工作就是：从参数获取数据，加载模板，然后模板进行带数据的渲染。下面是一个 `year_archive` 的视图例子：

```

1. # mysite/news/views.py
2.
3. from django.shortcuts import render
4.
5. from .models import Article
6.
7. def year_archive(request, year):
8.     a_list = Article.objects.filter(pub_date__year=year)
9.     context = {'year': year, 'article_list': a_list}
10.    return render(request, 'news/year_archive.html', context)

```

这个例子使用了 Django 的[模板系统](#)，它有着很多强大的功能，而且使用起来足够简单，即使不是程序员也可轻松使用。

## 设计模板

上面的代码加载了 `news/year_archive.html` 这个模板。

Django 允许设置搜索模板路径，这样可以最小化模板之间的冗余。在 Django 设置中，你可以通过 **DIRS** 参数指定目录列表来检索模板。如果模板不在第一个目录中，就继续检查第二个，以此类推。

比如 `news/year_archive.html` 模板找到了，它可能是这样的：

```
1. # mysite/news/templates/news/year_archive.html
2.
3. {% extends "base.html" %}
4.
5. {% block title %}Articles for {{ year }}{% endblock %}
6.
7. {% block content %}
8. <h1>Articles for {{ year }}</h1>
9.
10. {% for article in article_list %}
11.     <p>{{ article.headline }}</p>
12.     <p>By {{ article.reporter.full_name }}</p>
13.     <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
14. {% endfor %}
15. {% endblock %}
```

变量都被双花括号括起来了。 `{{ article.headline }}` 的意思是“输出 `article` 的 `headline` 属性值”。这个“点”不止用于查找属性，还可以查找字典键值、索引和函数调用。

注意： `{{ article.pub_date|date:"F j, Y" }}` 使用了 Unix 风格的“管道符”（“|”字符）。这是一个模板过滤器，用于过滤变量值。在这里过滤器将一个 Python `datetime` 对象转化为指定的格式（就像 PHP 中的日期函数那样）。

你可以将多个过滤器连在一起使用。你还可以[自定义模板过滤器](#)。你甚至可以[自定义模板标签](#)，相关的 Python 代码会在使用标签时在后台运行。

Django 使用了“模板继承”的概念。这就是 `{% extends "base.html" %}` 的作用。它的含义是“先加载名为 `base` 的模板作为基类，并且用下面的标记块对模板中定义的标记块进行填充”。简而言之，模板继承可以使模板间的冗余内容最小化：每个模板只需包含与其他文档有区别的内容。

下面是 `base.html` 可能的样子，它使用了[静态文件](#)：

```
1. # mysite/templates/base.html
2.
3. {% load static %}
4. <html>
5. <head>
6.     <title>{% block title %}{% endblock %}</title>
7. </head>
```

```
8. <body>
9.     
10.     {% block content %}{% endblock %}
11. </body>
12. </html>
```

简而言之，它定义了这个网站的外观（还有网站的 logo），并且给子模板们挖好了可以填的坑。这也让网站的改版变得简单无比——你只需更改这个base基类模板文件即可。

它也可以用来创建网站的多个版本，多个基类模板可以重用同一套子模板。Django 的创始人就用这种技术建立了网站的移动端适配版——只需建立一个新的基类模板。

注意，你并不是非得使用 Django 的模板系统，你可以使用其他你喜欢的模板系统。尽管 Django 的模板系统良好地集成了模型层，但这并不意味着你必须使用它。同样，你可以不使用 Django 的数据库 API。你可以用其他的数据库抽象层，像是直接读取 XML 文件，亦或直接读取磁盘文件，你可以使用任何方式。Django 的任何组成——模型、视图和模板——都是解耦的。

## 这只是冰山一角

以上只是 Django 的功能性概述。Django 还有更多实用的特性：

- [缓存框架](#)可以与 memcached 或其他后端集成。
- [聚合器框架](#)可以通过简单编写一个 Python 类来推送 RSS 和 Atom。
- 更多令人心动的自动化管理功能：概述里面仅仅浅尝辄止。

接下来您可以[下载 Django \(zh\)](#)，阅读 [实例教程 \(zh\)](#)，然后加入 [Django 社区](#)！感谢您的关注！

# 快速安装指南

- [快速安装指南](#)
  - [安装 Python](#)
  - [配置数据库](#)
  - [删除旧版本的 Django](#)
  - [安装 Django](#)
  - [验证安装](#)
  - [安装完成](#)

# 快速安装指南

你需要先安装 Django 才可以使用它。我们有一份[完整安装指南](#)，它涵盖了所有可能遇到的问题。本指南将会帮助你完成一个简单、最小化的安装。

## 安装 Python

作为一个 Python Web 框架，Django 依赖 Python。从 [Django 适用于哪些版本的 Python](#) 可以获取更多信息。较新版本的 Python 内置一个轻量级的数据库 [SQLite](#)，所以你暂时不需要配置数据库。

可以从 [Python 官网](#) 或者系统的包管理工具获取到最新版的 Python。

### *Jython 上的 Django*

如果你使用的是 Jython (一种 Java 平台的 Python 实现)，你需要做一些额外的步骤。查看在 [Jython 上运行 Django](#) 获取详细信息。

你可以在终端下输入命令 **python** 来验证是否已经安装 Python；你应该看到下面的信息：

```
1. Python 3.4.x
2. [GCC 4.x] on linux
3. Type "help", "copyright", "credits" or "license" for more information.
4. >>>
```

## 配置数据库

只有当你需要使用“大型”数据库例如 PostgreSQL、MySQL 或 Oracle 时，才需要这一步。若要安装这样的数据库，请参考[数据库安装信息](#)。

## 删除旧版本的 Django

---

如果你是从旧版本的 Django 升级安装，你将需要在[安装新版本之前卸载旧版本的 Django](#)。

## 安装 Django

---

你可以按下面三个简单的方式来安装 Django：

- [安装官方发布版本](#)。对大多数用户来说这是最好的方式。
- [安装操作系统所提供的发行包](#)。
- [安装最新的开发版](#)。这对于那些想要尝试最新最棒的特性而不担心运行崭新代码的用户来说是最好的。你可能会遇到一些 bug，但向 Django 报告这些 bug 将有助于他们的开发。此外，第三方包的很可能不兼容最新的开发版。

务必参考与你所使用的 *Django* 版本相对应的文档！

如果采用了前两种方式进行安装，你需要注意在文档中标明 [开发版新增](#) 的标记。这个标记表明这个特性仅适用开发版的 *Django*，而它们可能无法在当前版本工作。

## 验证安装

---

如果想验证是否成功安装了 Django，可以在终端输入 **python**。然后在 Python 提示符下，尝试导入 Django：

```
1. >>> import django
2. >>> print(django.get_version())
3. 1.11
```

如果版本和上面不一样，那你可能安装了其他版本的 Django 。

## 安装完成

---

安装完成！现在你可以开始通过 [实例学习 \(zh\)](#) 了。

# 创建你的第一个 Django 项目，第一部分

- [创建你的第一个 Django 项目，第一部分](#)
  - [创建项目](#)
  - [用于开发的服务器](#)
  - [创建 Polls 投票应用](#)
  - [编写你的第一个视图](#)
    - [url 参数: regex](#)
    - [url 参数: view](#)
    - [url 参数: kwargs](#)
    - [url 参数: name](#)

## 创建你的第一个 Django 项目，第一部分

来跟着实际项目学习 Django 吧。

在本教程中，我们将创建一个基础的投票网站。

它包含两个部分：

- 一个让公众查看投票内容并进行投票的公共站点。
- 一个能让你增加、修改和删除投票的管理界面。

我们假设你已经成功 [安装 Django \(zh\)](#)。如果你不清楚是否已经安装 Django 或不清楚安装的是哪个版本，请运行以下命令：

```
1. $ python -m django --version
```

如果已安装，你会看到安装的版本号；如果还没安装，你会看到错误提示：“No module named django”。

本教程的目标版本是 Django 1.11 和 Python 3.4 或更高版本。如果 Django 版本不匹配，你可以通过点击页面右下角的切换版本按钮来转到适合你版本的教程，或者你可以选择将 Django 升级到最新版本。如果你还在用 Python 2.7，你将需要对教程中的代码作一些微调，微调内容会被写在代码的注释里。

你可以查看文档 [快速安装指南 \(zh\)](#) 来获得关于移除旧版本，安装新版本的建议。

哪里可以获得帮助：

如果你在阅读或实践本教程中遇到困难，请发消息给 [django-users](#) 或加入IRC频道 [django on irc.freenode.net](#) 来与其他 Django 用户进行交流，他们也许能帮到你。

# 创建项目

如果这是你第一次使用 Django，你还需要进行一些初始化设置。也就是说，你需要通过自动生成代码来建立一个 Django 项目（一个 Django 项目实例需要的设置项集合，包括数据库配置，Django 选项和应用程序的具体设置）。

打开命令行，`cd` 切换到一个你想存放代码的目录，然后运行以下命令：

```
1. $ django-admin startproject mysite
```

这行代码将会在当前目录下创建一个 `mysite` 目录。如果命令不起作用，请看文档 [Problems running django-admin](#)。

## 注意

你得避免使用 *Python* 或 *Django* 的内部保留字来命名你的项目。具体地说，你得避免使用像 `django`（会和 *Django* 自己冲突）或 `test`（会和 *Python* 的内置模块冲突）这样的名字。

我的代码该放在哪？

如果你曾经是老式 *PHP* 程序员（没有使用过现代框架），你可能会习惯地把代码放在 *Web* 服务器的文档根目录（比如 `/var/www`）。但使用 *Django* 时你不用这样做。而且把所有 *Python* 代码放在 *Web* 服务器的根目录不是个好主意，因为这样会有风险。比如人们可能会在网站上看到你的代码。这不利于网站的安全。

你可以把代码放在文档根目录以外的地方，比如 `/home/mycode`。

让我们看看 `startproject` 这命令创建了什么：

```
1. mysite/
2.     manage.py
3.     mysite/
4.         __init__.py
5.         settings.py
6.         urls.py
7.         wsgi.py
```

这些目录和文件的用处是：

- 最外层的 `mysite/` 根目录只是你项目的容器，Django 不关心它的名字，你可以将它重命名为任何你喜欢的名字。
- `manage.py`：一个让你可以用各种方式管理该 Django 项目的命令行工具。你可以阅读 [django-admin and manage.py](#) 来获取关于 `manage.py` 的更多细节。
- 里面一层的 `mysite/` 目录就是你项目的实际 Python 包。它的名字就是当你引用它内部任何东西时需要用到的 Python 包名（比如：`mysite.urls`）。
- `mysite/__init__.py`：一个用于指明此目录是 Python 包的空白文件。（如果你刚开始学习 Python，请阅读 Python 官方文档中的 [more about packages](#)。）



- **mysite/settings.py**: 该 Django 项目的配置文件。如果你想知道这个文件是如何工作的，请看文档 [Django settings](#)。
- **mysite/urls.py**: 该 Django 项目的 URL 声明，就像是你网站的“目录”。阅读 [URL dispatcher](#) 文档来获取更多关于 URL 的内容。
- **mysite/wsgi.py**: 当你部署项目到一个兼容 WSGI 的服务器上时所需要的入口点。[How to deploy with WSGI](#) 文档内有更多关于这个文件的细节。

## 用于开发的服务器

让我们验证下项目是否可用。

切换到最外层的 **mysite** 目录下，如果你已经在这个目录下了，那就运行下面的命令：

```
1. $ python manage.py runserver
```

你会看到命令行输出：

```
1. Performing system checks...
2.
3. System check identified no issues (0 silenced).
4.
5. You have unapplied migrations; your app may not work properly until they are applied.
6. Run 'python manage.py migrate' to apply them.
7.
8. August 02, 2017 - 15:50:53
9. Django version 1.11, using settings 'mysite.settings'
10. Starting development server at http://127.0.0.1:8000/
11. Quit the server with CONTROL-C.
```

### 注意

现在请先忽略关于没有应用数据库迁移的警告，我们将在下一章解决它。

现在你已经开启了 Django 开发服务器 — 一个纯 Python 编写的轻量级 Web 服务器。我们已经在 Django 里包含了这项功能，所以你可以快速的开发网站，而不用去配置生产环境的服务器（比如 Apache），直到你做好了网站并准备投入生产环境。

注意：不要 在任何与生产环境相关的地方使用这个开发服务器，因为这只是为了开发所需。（我们的业务只是开发 Web 框架，而不是 Web 服务器）

现在服务器已经在运行了，在浏览器里访问 <http://127.0.0.1:8000/>。如果你看到一个写着“Welcome to Django”的令人赏心悦目的浅蓝色粉彩页面，那就是可以了！

### 更换端口

默认情况下，**runserver** 命令会将服务器设置为监听本机内部 IP 的 8000 端口。

如果你想更换服务器监听的端口，请使用命令行参数。比如，让服务器监听 `8080` 端口：

```
1. $ python manage.py runserver 8080
```

如果你想更换服务器监听的 `IP`，可以将它和端口号写在一起作为参数。比如，监听所有公网 `IP`（这尤其有用，当你想通过网络展示给其他人时），像这样：

```
1. $ python manage.py runserver 0:8000
```

`0` 是 `0.0.0.0` 的快捷方式。想了解更多关于这个用于开发的服务器的内容可以参考 [runserver](#)。

会自动重载的服务器

用于开发的服务器在需要的情况下会对每一次的访问请求重新载入一遍 `Python` 代码。所以你不需要为了让修改的代码生效而频繁地重启服务器。但是像添加新文件这样的动作，不会触发自动重载，这时你就需要手动重启服务器了。

## 创建 Polls 投票应用

现在你的开发环境 — 一个“项目” — 已经设置好了，你可以开始工作了。

在 Django 中，每一个应用都是一个 Python 包，并且遵循着一定的约定。Django 自带的工具，可以帮你自动生成应用的基础目录结构，这样你就能专注于编写代码，而不是创建目录了。

### 项目 vs 应用

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

你的应用可以放在 `Python path` 中的任何目录里。在本教程中，我们将直接在 `manage.py` 所在的目录里创建投票应用，这样它就能作为顶级模块被引入，而不是作为 `mysite` 的子模块。

请确定你现在处于 `manage.py` 所在的目录下，然后运行这行命令来创建一个应用：

```
1. $ python manage.py startapp polls
```

这将会创建一个 `polls` 目录，它的目录结构大致如下：

```
1. polls/
2.     __init__.py
3.     admin.py
4.     migrations/
5.         __init__.py
6.     models.py
7.     tests.py
8.     views.py
```

这个目录结构包括了投票应用的全部内容。

## 编写你的第一个视图

创建并打开 `polls/views.py`，然后写进 Python 代码：

```
1. # polls/views.py
2. from django.http import HttpResponse
3.
4. def index(request):
5.     return HttpResponse("Hello, world. You're at the polls index.")
```

这也许是 Django 中最简单的视图了。为了调用这个视图，我们需要在它和一个 URL 之间做映射，这就需要 `URLconf`。

为了在 `polls` 目录下创建一个 `URLconf`，需要先创建 `urls.py` 文件，你的应用目录现在应该是这样的：

```
1. polls/
2.     __init__.py
3.     admin.py
4.     apps.py
5.     migrations/
6.         __init__.py
7.     models.py
8.     tests.py
9.     urls.py
10.    views.py
```

`polls/urls.py` 文件里包含以下代码：

```
1. # polls/urls.py
2. from django.conf.urls import url
3.
4. from . import views
5.
6. urlpatterns = [
7.     url(r'^$', views.index, name='index'),
8. ]
```

下一步就是将根目录下的 `URLconf` 指向 `polls.urls` 模块。在 `mysite/urls.py` 中导入 `django.conf.urls.include`，并且在 `urlpatterns` 列表中插入 `include()`，像下面这样：

```

1. # mysite/urls.py
2. from django.conf.urls import include, url
3. from django.contrib import admin
4.
5. urlpatterns = [
6.     url(r'^polls/', include('polls.urls')),
7.     url(r'^admin/', admin.site.urls),
8. ]

```

**include()** 函数允许引用其他的 URLconf。要注意到 **include()** 函数并没有 **\$**(字符串匹配结束)，取而代之的是尾部的一个斜杠。每当 Django 遇到 **include()**，它就会排除正则匹配的部分，并将剩下的字符串发送到引用的 URLconf 中做进一步的处理。

**include()** 背后的想法是想使得 URL 的即插即用变得简单。polls 是在它们自己的 URLconf 中 (**polls/urls.py**)，它们是可以被放在 `"/polls/"`、`"/fun_polls/"`、`"/content/polls/"`、或者其他任何路径，而应用仍然是可以运行的。

什么时候使用 **include()**

当你要包含其他 URL 匹配模式时，你应该一直使用 **include()**。 **admin.site.urls** 在这里是一个例外。

不符合你所想看到的？

如果你看到的是 **include(admin.iste.urls)** 而不是 **admin.site.urls**，可能是你正在使用的 Django 版本和本教程的目标版本 (1.11) 不一致。那你就需要切换到旧版本的教程或者是安装较新的 Django。

现在你已经将 **index** 视图和 URLconf 连接在一起了。让我们验证下是不是生效了，运行以下命令：

```
1. $ python manage.py runserver
```

在你的浏览器中打开 <http://localhost:8000/polls/>。你应该能看到文字 “Hello, world. You’re at the polls index.” — 这是你在 **index** 视图中定义的。

**url** 函数有四个参数，两个必需参数：**regex** 正则和 **view** 视图；两个选项参数：**kwargs**字典和 **name** 名字。在这点上，值得再看下这些参数到底是干什么的。

## url参数：regex

术语 “regex” 是正则表达式 “regular expression” 的缩写，是匹配字符串的一段语法，像这里例子的是 url 匹配模式。Django 从列表的第一个正则表达式开始，按顺序匹配请求的 URL，直到找到与之匹配的。

注意，这些正则表达式不会去匹配 GET 和 POST 请求的参数值，或者域名。比如

<https://www.example.com/myapp/> 这个请求，URLconf 会找

[myapp/](https://www.example.com/myapp/)；<https://www.example.com/myapp/?page=3> 这个请求，URLconf 同样只会找

mysqpp/。

如果你需要正则表达式的帮助，可以看 [Wikipedia's entry](#) 和 关于 `re` 模块的文档。还有，由 Jeffrey Friedl 写的书 《掌握正则表达式》 也是很棒的。实际上，你并不需要成为正则表达式方面的专家，你真正要会的是如何使用简单捕获模式。因为复杂的正则可能会有不尽人意的查找性能，所以你不应该全依赖于正则匹配。

最后，一个性能注意点：这些正则表达式在 `URLconf` 模块加载后的第一时间就被编译了。它们都是非常快的（只要查找的不是特别复杂就像上面举例的）。

## url 参数：view

当 Django 发现一个正则表达式匹配时，Django 就会调用指定的视图函数，`HttpRequest` 对象作为第一个参数，正则表达式捕获的值作为其他参数。如果正则使用简单捕获，值会作为位置参数传递；如果使用命名捕获，值会作为关键字传递。我们稍后会给出一个例子。

## url 参数：kwargs

任意的关键字参数都可以作为字典传递到目标视图。但我们不准备在本教程里使用 Django 的这个特性。

## url 参数：name

命名你的 URL 可以让你在 Django 的别处明白引用的是什么，特别是在模版里。这个强大的特性允许你在项目里对一个文件操作就能对 URL 模式做全局改变。

当你对基本的请求和响应流都明白时，你就可以阅读 [教程第二部分 \(zh\)](#) 开始使用数据库了。

# 创建你的第一个 Django 项目，第二部分

- [创建你的第一个 Django 项目，第二部分](#)
  - [数据库的建立](#)
  - [创建模型](#)
  - [激活模型](#)
  - [初试 API](#)
  - [介绍下 Django 的管理站点](#)
  - [创建管理员账户](#)
  - [启动用于开发的服务器](#)
  - [进入管理页面](#)
  - [向管理页面中加入投票应用](#)
  - [体验便捷的管理功能](#)

## 创建你的第一个 Django 项目，第二部分

这一篇从 [第一部分 \(zh\)](#) 结尾的地方继续讲起。本节我们将建立数据库，创建你的第一个模型，然后是快速介绍一下 Django 自动生成的管理站点。

## 数据库的建立

现在，打开 `mysite/settings.py`，这是个普通的 Python 模块，用模块级别的变量表示 Django 设置。

默认情况下，配置的数据库是 SQLite，如果你对数据库不太熟，或者你只是对尝试 Django 感兴趣，这是最简单的选择。SQLite 内嵌在 Python 里，所以你不用再安装其他东西来支持你的数据库。但是当你开始做第一个实际的项目时，你也许想使用一个可扩展的数据库，比如 PostgreSQL 来避免令人头痛地切换数据库问题。

如果你希望使用其他数据库，你需要安装合适的 [database bingings](#) 和在 `DATABASES` ‘default’ 默认项里改变一些键值，以匹配你的数据库设置：

- **引擎 (ENGINE)** - ‘`django.db.backends.sqlite3`’、‘`django.db.backends.postgresql`’、‘`django.db.backends.mysql`’ 或者 ‘`django.db.backends.oracle`’。 其他的后端 [也可以参考](#)。
- **名字 (NAME)** - 你数据库的名字。如果你正在使用 SQLite，数据库将以文件形式保存在你的电脑；在这种情况下，**名字** 应该是绝对路径，包括文件名。默认 `os.path.join(BASE_DIR, ‘db.sqlite3’)` 将把文件保存在你项目的目录下。

如果你不使用 SQLite 作为你的数据库，那就必须额外设置下比如 **USER**，**PASSWORD** 和 **HOST**。

若想查看更多详情，可以参考文档 [DATABASES](#)。

对于 `SQLite` 以外的数据库

如果你使用除 `SQLite` 以外的数据库，请确认你已经创建了数据库。在你的数据库交互提示里用 `"CREATE DATABASE database_name;"` 创建数据库。

同样要确认在 `mysite/settings.py` 中的数据库用户拥有创建数据库的权限。这可以允许自动创建 [测速数据库](#) — 后面的教程需要。

如果你在使用 `SQLite`，你不需要在这之前创建什么 — 数据库文件会在需要的时候自动创建。

当你编辑 `mysite/settings` 的时候，记得把时区 `TIME_ZONE` 设成你要的时区。

同样的，注意 `INSTALLED_APPS` 应该设置在文件的较顶端处。它放着这个 Django 实例激活的所有 Django 应用程序。应用可以被用在多个项目中，你可以把它们打包分发，供其他人在项目中使用。

默认情况下，`INSTALLED_APPS` 包含着下面这些应用，它们都来自 Django：

- `django.contrib.admin` — 管理站点。你可以快捷地使用它。
- `django.contrib.auth` — 认证系统。
- `django.contrib.contenttypes` — 内容类型框架。
- `django.contrib.sessions` — session 框架。
- `django.contrib.messages` — 消息框架。
- `django.contrib.staticfiles` — 静态文件管理框架。

通常情况下为了方便，这些应用默认已被包含。

其中一些应用使用了至少一张数据库表，所以在使用它们之前，我们需要先在数据库中创建这些表。可以运行这行命令做些事：

```
1. $ python manage.py migrate
```

`migrate` 命令在 `mysite/settings.py` 文件的 `INSTALLED_APPS` 设置中寻找，并根据数据库设置创建一些必要的数据库表和随应用迁移的数据库（我们将会稍后介绍）。你将看到应用于各个迁移的消息。如果你感兴趣，可以运行一下你的数据库客户端，然后输入（`\dt`（PostgreSQL），`SHOW TABLES;`（MySQL），`.schema`（SQLite），`SELECT TABLE_NAME FROM USER_TABLES;`（Oracle））来看下 Django 创建的这些表。

给极简主义者

就像我们上面说的，通常情况下，这些默认应用都被包含了，但不是人人都需要它们的。如果你不需要其中一些，或者不需要它们全部，在运行 `migrate` 之前，可以随心把它们从 `INSTALLED_APPS` 中注释或删除。`migrate` 命令只会迁移那些在 `INSTALLED_APPS` 中激活的应用的数据库。

## 创建模型

现在我们将用额外的元数据来定义你的模型 —— 本质上是你的数据库布局。

#### 设计哲学

模型是你数据的简单明确的描述。它包含了储存的数据所必要的字段和行为。*Django* 遵循 **DRY 原则**。它的目标是让你只需要在一个地方定义数据模型，*Django* 就能自动从中导出迁移代码。

来介绍一下迁移 - 举个例子，不像 *Ruby On Rails*，*Django* 的迁移代码全部都是从你的模型文件导出的，它本质上只是个历史记录，*Django* 可以通过滚动更新数据库来匹配你当前的模型。

在这个简单的投票应用中，我们将创建两个模型：问题 **Question** 和选项 **Choice**。**Question** 模型包括问题描述和发布时间。**Choice** 模型有两个字段：选项描述和当前票数。每个选项属于一个问题。

这些概念可以通过一个简单的 Python 类来表示。像下面那样编辑 `polls/models.py` 文件：

```
1. # polls/models.py
2.
3. from django.db import models
4.
5. class Question(models.Model):
6.     question_text = models.CharField(max_length=200)
7.     pub_date = models.DateTimeField('date published')
8.
9. class Choice(models.Model):
10.    question = models.ForeignKey(Question, on_delete=models.CASCADE)
11.    choice_text = models.CharField(max_length=200)
12.    votes = models.IntegerField(default=0)
```

代码非常直白。每个模型都被表示为 `django.db.models.Model` 类的子类。每个模型都有些类变量，每一个都表示为模型里的一个数据库字段。

每个字段都是 **Field** 类的实例 - 比如，字符字段被表示为 **CharField**，日期时间字段被表示为 **DateTimeField**。这告诉 Django 每个要处理的字段是什么数据类型。

每个 **Field** 类实例变量的名字（比如 `question_text` 或 `pub_date`）都是字段名，这是对机器友好的格式。你将会在 Python 代码里使用它们，而数据库会将它们作为列名。

你可以使用可选的选项来为 **Field** 定义一个人类可读的名字。这个功能在很多 Django 内部组成部分中都被使用了，而且作为文档的一部分。如果某个字段没有被提供，Django 将会使用对机器友好的名称（也就是变量名）。在上面的例子中，我们只为 `Question.pub_date` 定义了对人类可读的名字。对于模型内的其他字段，它们的机器可读名也会被作为人类可读名使用。

定义某些 **Field** 类实例需要参数。例如 **CharField** 需要参数 `max_length`。该参数不止用于定义数据库结构，也用于验证数据，我们稍后将会看到这方面的内容。

**Field** 能够接收多个可选参数；在上面的例子中：我们已经将 `votes` 的 `default` 默认值设为0。



最后，还要注意我们使用了 **ForeignKey** 外键来定义一个关系。这会告诉 Django 每个 **Choice** 对象都和一个 **Question** 对象相关联。Django 支持所有常用的数据库关系：多对一、多对多和一对一。

## 激活模型

上面的一小段用于创建模型的代码给了 Django 很多信息，通过这些信息，Django 可以：

- 为这个应用创建数据库结构（生成 **CREATE TABLE** 语句）。
- 创建可以与 **Question** 和 **Choice** 对象进行交互的 Python 数据库 API。

但是我们首先要告诉项目 **polls** 应用要被安装进来。

### 设计哲学

*Django* 应用是“可插拔”的。你可以在多个项目中使用同一个应用。除此之外，你还可以分发自己的应用，因为它们并不会被绑定到当前安装的 *Django* 上。

要在项目中包含应用，我们需要在 **INSTALLED\_APPS** 设置里添加这个应用的设置类。这个设置类 **PollsConfig** 在 **polls/apps.py** 文件里，它的点分路径是 **'polls.apps.PollsConfig'**。编辑 **mysite/settings.py**，然后在 **INSTALLED\_APPS** 设置里添加这个点分路径，使其包含字符串 **polls**。看起来应该像这样：

```
1. # mysite/settings.py
2. INSTALLED_APPS = [
3.     'polls.apps.PollsConfig',
4.     'django.contrib.admin',
5.     'django.contrib.auth',
6.     'django.contrib.contenttypes',
7.     'django.contrib.sessions',
8.     'django.contrib.messages',
9.     'django.contrib.staticfiles',
10. ]
```

现在 Django 项目知道了要包含 **polls** 应用。接着运行下面的命令

```
1. $ python manage.py makemigrations polls
```

你将会看到类似于下面的输出：

```
1. Migrations for 'polls':
2.   polls/migrations/0001_initial.py:
3.     - Create model Choice
4.     - Create model Question
5.     - Add field question to choice
```

通过运行 **makemigrations** 命令，你告诉了 Django 你对模型文件做了些修改（在上面例子，你创建了新的模型），修改的部分被储存为了一次 迁移。

迁移是 Django 对于模型（也就是你的数据库结构）的修改的储存形式 - 它们其实也只是些磁盘上的文件。如果你想的话，你可以阅读一下你模型的迁移数据，它被储存在 **polls/migrations/0001\_initial.py** 里。别担心，你不需要每次都阅读迁移文件，但是它们被设计成人类可编辑的形式，这是为了你能手动对它们进行微调。

Django 有一个自动执行数据库迁移并同步管理你的数据库结构的命令 - 这个命令是 **migrate**，我们马上就会接触它 - 但是首先，让我们看看迁移命令会执行哪些 SQL 语句。**sqlmigrate** 命令接收一个迁移的名称，然后返回对应的 SQL：

```
1. $ python manage.py sqlmigrate polls 0001
```

你应该会看到类似下面这样的输出（我们已经把输出重组成了可读的格式）：

```
1. BEGIN;
2. --
3. -- 创建 Choice 模型
4. --
5. CREATE TABLE "polls_choice" (
6.     "id" serial NOT NULL PRIMARY KEY,
7.     "choice_text" varchar(200) NOT NULL,
8.     "votes" integer NOT NULL
9. );
10. --
11. -- 创建 Question 模型
12. --
13. CREATE TABLE "polls_question" (
14.     "id" serial NOT NULL PRIMARY KEY,
15.     "question_text" varchar(200) NOT NULL,
16.     "pub_date" timestamp with time zone NOT NULL
17. );
18. --
19. -- 对 choice 表添加 question 字段
20. --
21. ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
22. ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
23. CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
24. ALTER TABLE "polls_choice"
25.     ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
26.     FOREIGN KEY ("question_id")
27.     REFERENCES "polls_question" ("id")
28.     DEFERRABLE INITIALLY DEFERRED;
29.
```

```
30. COMMIT;
```

请注意以下几点：

- 输出的内容和你使用的数据库有关，上面的输出示例使用的是 PostgreSQL。
- 数据库的表名是由应用名（**polls**）和模型名的小写形式（**question** 和 **choice**）连接而来。（如果需要，你可以自定义此行为。）
- 主键（ID）会被自动添加。（当然，你也可以自定义。）
- 惯例下，Django 会在外键字段名后追加字符串 “\_id”。（同样，这也可以自定义。）
- 外键关系由 **FOREIGN KEY** 生成。别担心 **DEFERRABLE** 部分，它只是告诉 PostgreSQL，请在事务全都执行完之后再创建外键关系。
- 生成的 SQL 语句是为你所用的数据库定制的，所以那些和数据库有关的字段类型，比如 **auto\_increment**（MySQL）、**serial**（PostgreSQL）和 **integer primary key**（SQLite），都会为你自动处理的。那些和引号相关的事情 - 例如，是使用单引号还是双引号 - 也一样会被自动处理。
- **sqlmigrate** 命令并没有真正在你的数据库中的执行迁移 - 它只是把命令输出到屏幕上，让你看看 Django 认为需要执行哪些 SQL 语句。这在你想看看 Django 到底准备做什么，或者当你是数据库管理员，需要写脚本来批量处理数据库时会很有用。

如果你感兴趣，你也可以尝试运行 **python manage.py check**；这个命令帮助你检查项目中的问题，并且在检查过程中不会对数据库进行任何操作。

现在，再次运行 **migrate** 命令，在数据库里创建模型的数据表：

```
1. $ python manage.py migrate
2. Operations to perform:
3.   Apply all migrations: admin, auth, contenttypes, polls, sessions
4. Running migrations:
5.   Rendering model states... DONE
6.   Applying polls.0001_initial... OK
```

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

迁移是非常强大的功能，它能让你在开发过程中持续的改变数据库结构而不需要重新删除和创建表 - 它专注于使数据库平滑升级而不会丢失数据。我们会在后面的教程中更加深入的学习这部分内容，现在，你只需要记住，改变模型需要这三步：

- 编辑 **models.py** 文件，改变模型。
- 运行 **python manage.py makemigrations** 为模型的改变生成迁移文件。
- 运行 **python manage.py migrate** 来应用数据库迁移。

数据库迁移被分解成生成和应用两个命令是为了让你能够在代码控制系统上提交迁移数据并使其能在

多个应用里使用；这不仅仅会让开发更加简单，也给别的开发者和生产环境中的使用带来方便。

通过阅读 [Django-admin 文档](#)，你可以获得关于 `manage.py` 工具的更多信息。

## 初试 API

现在让我们进入交互式 Python 命令行，尝试一下 Django 为你创建的各种 API。通过以下命令打开 Python 命令行：

```
1. $ python manage.py shell
```

我们使用这个命令而不是简单的使用“Python”是因为 `manage.py` 会设置 `DJANGO_SETTINGS_MODULE` 环境变量，这个变量会让 Django 根据 `mysite/settings.py` 文件来设置 Python 包的导入路径。

我就是不想用 `manage.py`

如果你不想使用 `manage.py`，没问题，你只要手动将 `DJANGO_SETTINGS_MODULE` 环境变量设置为 `mysite.settings` 就行。打开一个普通的 Python 命令行，然后输入以下命令来配置 Django：

```
1. >> import django
2. >> django.setup()
```

如果抛出 `AttributeError` 错误，说明你使用的 Django 版本可能和本教程不一致。你可以切换到旧版本的教程或者把 Django 升级至最新版本。

你必须在 `manage.py` 所在目录中运行 `python` 命令，或者确保这个目录在 `Python path` 里，因为只有这样 `import mysite` 才能被正确的执行。

阅读 [Django-admin 文档](#) 获取更多信息。

当你成功进入命令行后，来试试 [数据库 API](#) 吧：

```
1. >>> from polls.models import Question, Choice # 导入刚刚创建的模型类
2.
3. # 现在系统里还没有 Question 对象
4. >>> Question.objects.all()
5. <QuerySet []>
6.
7. # 创建新 Question
8. # 在 settings 文件里，时区支持被设为开启状态，所以
9. # pub_date 字段要求一个带有时区信息 (tzinfo)
10. # 的 datetime 数据。请使用 timezone.now() 代替
11. # datetime.datetime.now()，这样就能获取正确的时间。
12. >>> from django.utils import timezone
13. >>> q = Question(question_text="What's new?", pub_date=timezone.now())
```

```

14.
15. # 想将对象保存到数据库中，必须显式的调用 save()。
16. >>> q.save()
17.
18. # 现在它被分配了一个 ID。注意有可能你的结果是“1L”而不是“1”，
19. # 这取决于你在使用哪种数据库。这不是什么大问题；只是表明
20. # 你所用的数据库后端倾向于将整数转换为 Python 的
21. # long integer 对象。
22. >>> q.id
23. 1
24.
25. # 通过属性来获取模型字段的值
26. >>> q.question_text
27. "What's new?"
28. >>> q.pub_date
29. datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)
30.
31. # 通过改变属性值来改变模型字段，然后调用 save()。
32. >>> q.question_text = "What's up"
33. >>> q.save()
34.
35. # objects.all() 显示数据库中所有 question。
36. >>> Question.objects.all()
37. <QuerySet [ <Question: Question object> ]>

```

等等。用 \ 表示这个对象没什么帮助信息，它无法告诉我们这个对象的详细信息。让我们通过编辑 **Question** 模型的代码 (**polls/models.py** 文件)，给 **Question** 和 **Choice** 增加 **\_\_str\_\_()** 方法来改善这个问题：

```

1. # polls/models.py
2. from django.db import models
3. from django.utils.encoding import python_2_unicode_compatible
4.
5. @python_2_unicode_compatible # 如果你想支持 Python 2
6. class Question(models.Model):
7.     # ...
8.     def __str__(self):
9.         return self.question_text
10.
11. @python_2_unicode_compatible # 如果你想支持 Python 2
12. class Choice(models.Model):
13.     # ...
14.     def __str__(self):
15.         return self.choice_text

```

给模型增加 **\_\_str\_\_()** 方法是很重要的，这不仅仅能给你在交互式的命令行里使用带来方便，而且

Django 自动生成的 admin 里也是使用这个方法来表示对象的。

注意这些只是 Python 的普通方法。我们可以向模型里添加自定义方法，示范：

```

1. # polls/models.py
2.
3. import datetime
4.
5. from django.db import models
6. from django.utils import timezone
7.
8. class Question(models.Model):
9.     # ...
10.     def was_published_recently(self):
11.         return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

```

注意，新加入的 `import datetime` 和 `from django.utils import timezone` 分别导入了 Python 的标准 `datetime` 模块和 Django 中和时区相关的 `django.utils.timezone` 工具模块。如果你不太熟悉 Python 中的时区处理，可以看看 [时区支持文档](#)。

保存文件然后通过 `python manage.py shell` 命令再次打开 Python 交互式命令行：

```

1. >>> from polls.models import Question, Choice
2.
3. # 确认添加的 __str__() 是否正常工作。
4. >>> Question.objects.all()
5. <QuerySet [<Question: What's up?>]>
6.
7. # Django 提供了丰富的数据库查找 API,
8. # 通过关键字参数就能轻松使用。
9. >>> Question.objects.filter(id=1)
10. <QuerySet [<Question: What's up?>]>
11. >>> Question.objects.filter(question_text__startswith='What')
12. <QuerySet [<Question: What's up?>]>
13.
14. # 获取今年发布的问题
15. >>> from django.utils import timezone
16. >>> current_year = timezone.now().year
17. >>> Question.objects.get(pub_date__year=current_year)
18. <Question: What's up?>
19.
20. # 查找一个不存在的 ID 将会引发异常
21. >>> Question.objects.get(id=2)
22. Traceback (most recent call last):
23.     ...
24. DoesNotExist: Question matching query does not exist.
25.

```

```

26. # 通过主键来查找数据是非常常见的需求，所以 Django
27. # 为这种需求专门制定了一个参数。
28. # 以下代码等同于 Question.objects.get(id=1)。
29. >>> Question.objects.get(pk=1)
30. <Question: What's up?>
31.
32. # 确认我们自定义的方法正常工作。
33. >>> q = Question.objects.get(pk=1)
34. >>> q.was_published_recently()
35. True
36.
37. # 给这个问题添加几个选项。create 函数会创建一个新的
38. # Choice 对象，执行 INSERT 语句，将 Choice 添加到
39. # Question 的选项列表中，最后返回刚刚创建的
40. # Choice 对象。Django 创建了一个集合 API 来使你可以从
41. # 外键关系的另一方管理关联的数据。
42. # （例如，可以获取问题的选项列表）
43. >>> q = Question.objects.get(pk=1)
44.
45. # 显示所有和当前问题关联的选项列表，现在是空的。
46. >>> q.choice_set.all()
47. <QuerySet []>
48.
49. # 创建三个选项。
50. >>> q.choice_set.create(choice_text='Not much', votes=0)
51. <Choice: Not much>
52. >>> q.choice_set.create(choice_text='The sky', votes=0)
53. <Choice: The sky>
54. >>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)
55.
56. # Choice 对象能通过 API 获取关联到的 Question 对象。
57. >>> c.question
58. <Question: What's up?>
59.
60. # 反过来，Question 对象也可以获取 Choice 对象
61. >>> q.choice_set.all()
62. [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
63. >>> q.choice_set.count()
64. 3
65.
66. # 查找 API 的关键字参数可以自动调用关系函数。
67. # 只需使用双下划线来分隔关系函数。
68. # 只要你想，这个调用链可以无限长。
69. # 例如查找所有「所在问题的发布日期是今年」的选项
70. # （重用我们之前创建的 'current_year' 变量）
71. >>> Choice.objects.filter(question__pub_date__year=current_year)
72. [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
73.

```

```

74. # 试试删除一个选项，使用 delete() 函数。
75. >>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
76. >>> c.delete()

```

阅读 [Accessing related objects](#) 文档可以获取关于数据库关系的更多内容。想知道关于双下划线的更多用法，参见 [Field Lookup](#) 文档。数据库 API 的所有细节可以在 [数据库 API 参考](#) 文档中找到。

## 介绍下 Django 的管理站点

### 设计哲学

为你的团队和客户创建一个用于添加、修改和删除网站内容的管理页面是一项乏味的工作，而且不需要太多的创造力。因为这些原因，*Django* 提供完全自动地为模型创建管理接口的功能。

*Django* 产生于一个公众页面和内容发布者页面完全分离的新闻类站点的开发过程中。站点管理人员使用管理系统来添加新闻、事件和体育时讯等，这些添加的内容被显示在公众页面上。*Django* 通过为站点管理人员创建统一的内容编辑界面解决了这个问题。

管理界面不是为了网站的访问者，而是为管理者准备的。

## 创建管理员账户

首先，我们得创建一个能登录管理页面的用户。请运行下面的命令：

```
1. $ python manage.py createsuperuser
```

输入你想使用的用户名，然后回车。

```
1. Username: admin
```

接着，你会被提示要求输入邮箱：

```
1. Email address: admin@example.com
```

最后一步是输入密码。你会被要求输入两次密码，第二次的目的是为了确认第一次输入的确实是你想要的密码。

```

1. Password: *****
2. Password(again): *****
3. Superuser created successfully.

```



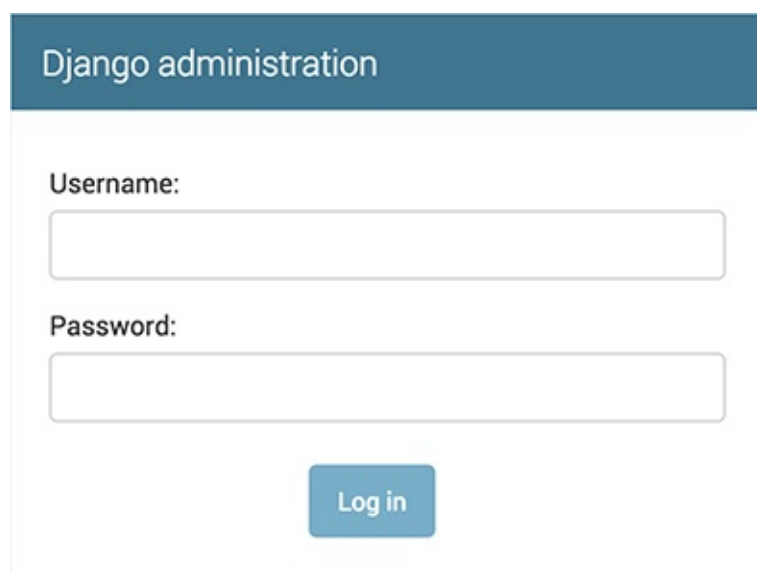
## 启动用于开发的服务器

Django 的管理界面默认就是启用的。让我们启动开发服务器，看看它到底是什么样的。

如果服务器还没运行，那就运行下：

```
1. $ python manage.py runserver
```

现在，打开浏览器，转到你本地域名的 “/admin/” 目录，比如 “<http://127.0.0.1:8000/admin>”。你应该会看见管理员登录界面：

The image shows the Django administration login page. It has a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". Below the password field, there is a blue button labeled "Log in".

Django administration

Username:

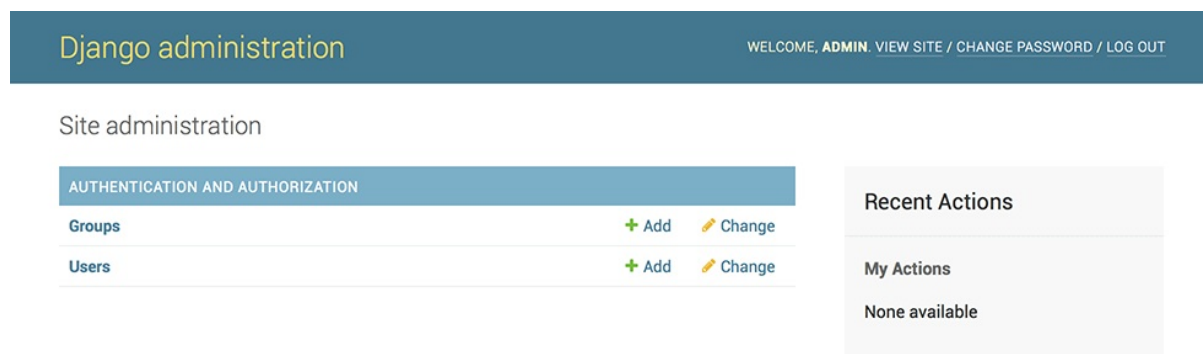
Password:

Log in

因为 [翻译](#) 功能默认是开着的，所以登录界面可能会使用你的语言，取决于你的浏览器设置和是否 Django 已被翻译成你的语言。

## 进入管理页面

现在，试着使用你在上一步中创建的超级用户来登录。然后你将会看到 Django 管理页面的索引页：



你将会看到几种可编辑的内容：组和用户。它们由 `django.contrib.auth` 提供，这是 Django 开发的认证框架。

## 向管理页面中加入投票应用

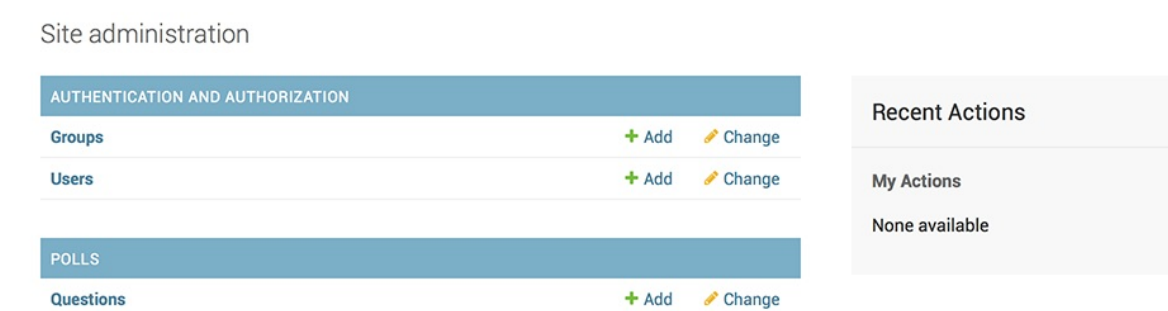
但是我们的投票应用在哪呢？它没在索引页面里显示。

只需要做一件事：我们得告诉管理页面，`Question` 对象需要被管理。打开 `polls/admin.py` 文件，把它编辑成下面这样：

```
1. # polls/admin.py
2.
3. from django.contrib import admin
4.
5. from .models import Question
6.
7. admin.site.register(Question)
```

## 体验便捷的管理功能

现在我们向管理页面注册了 `Question` 类。Django 知道它应该被显示在索引页里：



点击 “Question”。现在看到是 Question 对象的修改列表。这个界面会显示所有数据库里的 Question 对象，你可以选择一个来修改。这里现在有我们在上一部分中创建的 “What’s up?” 问题。

Home > Polls > Questions

Select question to change ADD QUESTION +

Action:   0 of 1 selected

<input type="checkbox"/>	QUESTION
<input type="checkbox"/>	What's up?

1 question

点击 “What’s up?” 来编辑这个 Question 对象：

Home > Polls > Questions > What's up?

Change question HISTORY

Question text:

Date published:

Date:  Today

Time:  Now

有些事情需要注意：

- 这个表单是从 **Question** 模型中自动生成的。
- 不同的字段类型 日期时间字段 (**DateTimeField**)、字符字段 (**CharField**) 会生成对应的 HTML 输入控件。每个类型的字段都知道它们该如何在管理页面里显示自己。
- 每个 日期时间字段 (**DateTimeField**) 都有 JavaScript 写的快捷按钮。日期有转到今天 (Today) 的快捷按钮和一个弹出式日历界面。时间有设为现在 (Now) 的快捷按钮和一个列出常用时间的方便的弹出式列表。

页面的底部提供了几个选项：

- 保存 (Save) - 保存改变，然后返回对象列表。
- 保存并继续编辑 (Save and continue editing) - 保存改变，然后重新载入当前对象的修改界面。
- 保存并新增 (Save and add another) - 保存改变，然后添加一个新的空对象并载入修改界面。

- 删除 (Delete) - 显示一个确认删除页面。

如果显示的“发布日期 (Date Published)”和你在教程第一部分里创建它们的时间不一致，这意味着你可能没有正确的设置 `TIME_ZONE`。改变设置，然后重新载入页面看看是否显示了正确的值。

通过点击“今天” (Today) 和“现在 (Now)”按钮改变“发布日期 (Date Published)”。然后点击“保存并继续编辑 (Save and add another)”按钮。然后点击右上角的“历史 (History)”按钮。你会看到一个列出了所有通过 Django 管理页面对当前对象进行的改变的页面，其中列出了时间戳和进行修改操作的用户名：

Home › Polls › Questions › What's up? › History

Change history: What's up?

DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

当你明白了模型的 API 和熟悉了你自己的管理站点后，你就可以开始阅读 [教程第三部分 \(zh\)](#)，学习如何在投票应用中添加更多的视图。

## 创建你的第一个 Django 项目，第三部分

- [创建你的第一个 Django 项目，第三部分](#)
  - [总概](#)
  - [编写更多的视图](#)
  - [写一个真正有用的视图](#)
  - [一个快捷函数：render\(\)](#)
  - [抛出 404 错误](#)
  - [一个快捷函数：get\\_object\\_of\\_404\(\)](#)
  - [使用模板系统](#)
  - [去除模板中的硬编码 URL](#)
  - [为 URL 名称添加命名空间](#)

## 创建你的第一个 Django 项目，第三部分

这一篇从 [第二部分 \(zh\)](#) 结尾的地方继续。我们将继续编写投票应用，并且着力于如何创建公用接口 — “视图”。

### 总概

Django 中的视图的概念是「一类具有相同功能和模板的网页的集合」。比如，在一个博客应用中，你可能会创建如下几个视图：

- 博客首页 — 展示最近的一些记录。
- 内容详情页 — 详细展示某项内容。
- 以年为单位的归档页 — 展示选中的年份里各个月份创建的内容。
- 以月为单位的归档页 — 展示选中的月份里各天创建的内容。
- 以天为单位的归档页 — 展示选中天里创建的所有内容。
- 评论页 — 用于响应为一项内容添加评论的操作。

而在投票应用中，我们需要下列几个视图：

- 问题索引页 — 展示最近的几个投票问题。
- 问题详情页 — 展示某个投票的问题和不带结果的选项列表。
- 问题结果页 — 展示某个投票的结果。
- 投票页 — 用于响应用户为某个问题的特定选项投票的操作。

在 Django 中，网页和其他内容都是从视图派生而来。每一个视图表现为一个简单的 Python 函数（或者说方法，如果是在基于类的视图里的话）。Django 将会根据用户请求的 URL 来选择使用哪个视图（更准确的说，是根据 URL 中域名之后的部分）。

在你上网的过程中，很可能看见过像这样美丽的 URL：“ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B”。别担心，Django 里的 URL 要比这优雅的多！

一个 URL 模式定义了某种 URL 的基本格式 —— 举个例子：

`/newsarchive/<year>/<month>/`

为了将 URL 和视图关联起来，Django 使用了“URLconfs”来配置。URLconf 将 URL 模式（表现为一个正则表达式）映射到视图。

本教程只会介绍 URLconf 的基础内容，你可以查看 [django.core.urlresolvers](https://docs.djangoproject.com/en/1.11/topics/http/urls/) 以获取更多内容。

## 编写更多的视图

现在让我们添加更多的视图进 `polls/view.py`，这些视图都有点不一样，因为它们都接受一个参数：

```
1. # polls/views.py
2.
3. def detail(request, question_id):
4.     return HttpResponse("You're looking at question %s." % question_id)
5.
6. def results(request, question_id):
7.     response = "You're looking at the results of question %s."
8.     return HttpResponse(response % question_id)
9.
10. def vote(request, question_id):
11.     return HttpResponse("You're voting on question %s." % question_id)
```

要把这些新视图添加进 `polls.urls` 模块里，只需添加几个 `url()` 函数调用：

```
1. # polls/urls.py
2.
3. from django.conf.urls import url
4.
5. from . import views
6.
7. urlpatterns = [
8.     # ex: /polls/
9.     url(r'^$', views.index, name='index'),
10.    # ex: /polls/5/
11.    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
12.    # ex: /polls/5/results/
```

```

13.     url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
14.     # ex: /polls/5/vote/
15.     url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
16. ]

```

然后在你的浏览器里转到 `"/polls/34/"`，Django 将会运行 `detail()` 方法并展示你在 URL 里提供的问题 ID。再试试 `"/polls/34/results"` 和 `"/polls/34/vote/"` — 你将会看到暂时用于占位的结果和投票页。

当某人请求你网站的某一页面时——比如说，`"/polls/34/"`，Django 将会载入 `mysite.urls` 模块，因为配置项 `ROOT_URLCONF` 说要载入它。然后 Django 寻找名为 `urlpatterns` 变量并且按序遍历正则表达式。Django 找到匹配的正则表达式 `'^polls/'` 然后 Django 将会去除被匹配的部分 (`polls/`)，然后发送剩下的文本 — `"34/"` — 给 `"polls.urls"` 这个 `URLconf` 做进一步处理。然后找到匹配的正则表达式 `r'^(?P[0-9]+)/$'`，随后用以下方式调用 `detail()` 函数：

```

1. detail(request=<HttpRequest object>, question_id='34')

```

`question_id='34'` 这一部分是由 `(?P[0-9+])` 产生的。使用括号来包围一部分模式，就可以“捕获”这部分所匹配到的文本，随后作为参数被传递给视图函数；`?P` 用于定义匹配部分的名字；`[0-9+]` 是用于匹配一连串数字（也就是所有整数）的正则表达式。

因为 URL 模式本质上是正则表达式，所以不会有规定限制你如何使用它们。还有，没必要为每个 URL 加上不必要的东西，例如 `.html`。不过如果你非要加的话，也是可以的：

```

1. url(r'^polls/latest\.html$', views.index),

```

但是，别这样做，这太傻了。

## 写一个真正有用的视图

每个视图必须要做的只有两件事：返回一个包含被请求页面内容的 `HttpResponse` 对象，或者抛出一个异常，比如 `Http404`。至于你还想干什么，都随你。

你的视图可以从数据库里读取记录，可以使用一个模板引擎（比如 Django 自带的，或者其他第三方的），可以生成一个 PDF 文件，可以输出一个 XML，创建一个 ZIP 文件，你可以做任何你想做的事，使用任何你想用的 Python 库。

Django 只需要一个 `HttpResponse`，或者一个异常。

因为 Django 自带的数据库 API 很方便（我们在 [在第二部分 \(zh\)](#) 里介绍过），所以我们试试在视图里使用它。我们在 `index()` 函数里插入了一些新内容，让它能展示数据库里按发布日期倒序的

最近五个投票问题，以逗号分割：

```
1. # polls/views.py
2.
3. from django.http import HttpResponse
4.
5. from .models import Question
6.
7.
8. def index(request):
9.     latest_question_list = Question.objects.order_by('-pub_date')[:5]
10.    output = ', '.join([q.question_text for q in latest_question_list])
11.    return HttpResponse(output)
12.
13. # 暂时不管剩下的视图 (detail, results, vote)
```

这里有个问题：页面的设计是硬编码在视图函数的代码里的。如果你想改变页面的样子，你就要编辑 Python 代码。所以让我们使用 Django 的模板系统，只要创建一个视图，就可以将页面的设计从代码中分离出来。

首先，在你的 `polls` 目录里创建一个 `templates` 目录。Django 将会在这个目录里查找模板文件。

你项目的 `TEMPLATES` 配置项描述了 Django 如何加载和渲染模板。默认的设置文件设置了 `DjangoTemplates` 后端，并将 `APP_DIRS` 设置成了 `True`。这一选项将会让 `DjangoTemplates` 在每个 `INSTALLED_APPS` 中激活的应用文件夹中寻找 “templates” 子目录。

`templates` 目录建好后，在里面再创建一个目录 `polls`，在这个 `polls` 的目录里再建立一个文件 `index.html`。换句话说，你的模版路径应该是 `polls/templates/polls/index.html`。由于 `app_directories` 模版加载器的工作原理如上所述，所以你使用简单的 `polls/index.html` 就能引用这模版了。

#### 模板命名空间

虽然我们现在可以将模板文件直接放在 `polls/templates` 文件夹中（而不是再建立一个 `polls` 子文件夹），但这不是个好办法。*Django* 将会选择第一个匹配的模板文件，如果你有一个模板文件正好和另一个应用中的某个模板文件重名，*Django* 没有办法区分它们。我们需要 *Django* 选择正确的模板，最简单的方法就是把他们放入各自的命名空间中，也就是把这些模板放入一个和自身应用重名的子文件夹里。

将下面的代码输入到刚刚创建的模板文件中：

```
1. <!-- polls/templates/polls/index.html -->
2.
3. {% if latest_question_list %}
4.     <ul>
5.         {% for question in latest_question_list %}
```



```

6.         <li><a href="/polls/{ question.id }/">{{ question.question_text }}</a></li>
7.     {% endfor %}
8. </ul>
9. {% else %}
10.    <p>No polls are available.</p>
11. {% endif %}

```

为了使用这个模版，让我们更新一下在 `polls/views.py` 里的 `index` 视图：

```

1. # polls/views.py
2.
3. from django.http import HttpResponse
4. from django.template import loader
5.
6. from .models import Question
7.
8.
9. def index(request):
10.     latest_question_list = Question.objects.order_by('-pub_date')[:5]
11.     template = loader.get_template('polls/index.html')
12.     context = {
13.         'latest_question_list': latest_question_list,
14.     }
15.     return HttpResponse(template.render(context, request))

```

上述代码的作用是，加载 `polls/index.html` 模板，并且向它传递一个上下文环境（context）。这个上下文是一个字典，它将模板内的变量映射为 Python 对象。

在你的浏览器访问 `"/polls/"`，你将会看见一个无序列表，列出了我们在 [教程二 \(zh\)](#) 中添加的“`What's up`”投票问题，它链接到这个问题的详情页。

## 一个快捷函数：`render()`

「载入模板，填充上下文，再返回由它生成的 `HttpResponse` 对象」是一个如此常用的操作流程。于是 Django 提供了一个快捷函数，我们用它来重写 `index()` 视图：

```

1. # polls/views.py
2.
3. from django.shortcuts import render
4.
5. from .models import Question
6.
7.
8. def index(request):
9.     latest_question_list = Question.objects.order_by('-pub_date')[:5]

```

```

10.     context = {'latest_question_list': latest_question_list}
11.     return render(request, 'polls/index.html', context)

```

注意，我们不再需要导入 `loader` 和 `HttpResponse`。不过如果你还有其他函数（比如说 `detail`，`results` 和 `vote`）需要用到它的话，就需要保持 `HttpResponse` 的导入。

`render()` 函数把请求(`HttpRequest`)对象作为第一个参数，加载的模版名字作为第二个参数，用于渲染模板的上下文字典作为可选的第三个参数。函数返回一个 `HttpResponse` 对象，内容为指定模板用指定上下文渲染后的结果。

## 抛出 404 错误

现在，让我们来处理下投票详情视图 — 它会显示指定投票的问题标题。下面是这个视图的代码：

```

1. # polls/views.py
2.
3. from django.http import Http404
4. from django.shortcuts import render
5.
6. from .models import Question
7. # ...
8. def detail(request, question_id):
9.     try:
10.         question = Question.objects.get(pk=question_id)
11.     except Question.DoesNotExist:
12.         raise Http404("Question does not exist")
13.     return render(request, 'polls/detail.html', {'question': question})

```

这里有个新概念：如果指定问题 ID 所对应的问题不存在，这个视图就会抛出一个 `Http404` 异常。

我们稍后再讨论你需要在 `polls/details.html` 里输入什么，但是如果你想试试上面这段代码是否正常工作的话，你可以暂时把下面这段输进去：

```

1. <!-- polls/templates/polls/details.html -->
2.
3. {{ question }}

```

这样你就能测试了。

## 一个快捷函数：get\_object\_of\_404()

「用 `get()` 函数获取对象，抛出 `Http404` 错误」也是一个常用流程。Django 也提供了一个快捷函数，下面是重写的 `detail()` 视图：

```
1. # polls/views.py
2.
3. from django.shortcuts import get_object_or_404, render
4.
5. from .models import Question
6. # ...
7. def detail(request, question_id):
8.     question = get_object_or_404(Question, pk=question_id)
9.     return render(request, 'polls/detail.html', {'question': question})
```

`get_object_or_404()` 函数的第一个参数是一个 Django 模型。在此之后可以有任意个的关键字参数，他们会被直接传递给模型的 `get()` 函数。如果对象并不存在，此快捷函数将会抛出一个 `Http404` 异常。

#### 设计哲学

为什么我们使用辅助函数 `get_object_or_404()` 而不是自己捕获 `ObjectDoesNotExist` 异常呢？或者，为什么模型 API 不直接抛出 `Http404` 而是抛出 `ObjectDoesNotExist` 呢？

因为这样做会增加模型层和视图层的耦合度。指导 Django 设计的最重要的思想之一就是要保证松散耦合。一些受控的耦合将会被包含在 `django.shortcuts` 模块中。

也有 `get_list_or_404()` 函数，工作原理和 `get_object_or_404()` 一样，除了 `get()` 函数被换成了 `filter()` 函数。如果列表为空的话会抛出 `Http404` 异常。

## 使用模板系统

回过头去看看我们的 `detail()` 视图。它向模板传递了上下文变量 `question`。下面是 `polls/detail.html` 模板里的代码：

```
1. <!-- polls/templates/polls/detail.html -->
2.
3. <h1>{{ question.question_text }}</h1>
4. <ul>
5. {% for choice in question.choice_set.all %}
6.     <li>{{ choice.choice_text }}</li>
7. {% endfor %}
8. </ul>
```

模板系统统一使用「点」符号来访问变量的属性。在示例 `{{ question.question_text }}` 中，首先 Django 尝试对 `Question` 对象使用字典查找（也就是使用 `obj.get(str)` 操作），如果失败了就尝试属性查找（也就是 `obj.str` 操作），结果是成功了。如果这一操作也失败的话，将会

尝试列表查找（也就是 `obj[int]` 操作）。

在 `{% for %}` 循环中发生的函数调用：

`question.choice_set.all` 被解释为 Python 代码 `question.choice_set.all()`，将会返回一个可迭代的 `Choice` 对象，这一对象可以在 `{% for %}` 标签内部使用。

查看 [模板指南](#) 可以了解关于模板的更多信息。

## 去除模板中的硬编码 URL

还记得吗，我们在 `polls/index.html` 里编写投票链接时，链接是硬编码的：

```
1. <li><a href="/polls/{{ question.id }}">{{ question.question_text }}</a></li>
```

问题在于，硬编码和强耦合的链接，对于一个包含很多应用的项目来说，修改起来是十分困难的。然而，因为你在 `polls.urls` 的 `urls()` 函数中通过 `name` 参数为 URL 定义了名字，你可以使用 `{% url %}` 标签代替它：

```
1. <li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

这个标签的工作方式是在 `polls.urls` 模块的 URL 定义中寻具有指定名字的条目。你可以回忆一下，具有名字 “detail” 的 URL 是这样被定义的：

```
1. ...
2. # name 变量被 {% url %} 标签调用
3. url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
4. ...
```

如果你想改变投票详情视图的 URL，比如想改成 `polls/specifcics/12/`，你不用在模板里修改任何东西，只要在 `polls/urls.py` 里稍微修改一下就行：

```
1. ...
2. # 增加 specifics
3. url(r'^specifics/(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
4. ...
```

## 为 URL 名称添加命名空间

本教程的项目只有一个应用，`polls`。在一个真实的 Django 项目中，可能会有五个，十个，二十个甚至更多应用。Django 如何分辨重名的 URL 呢？举个例子，`polls` 应用有 `detail` 视图，可

能另一个博客应用也有同名的视图。Django 如何知道 `{% url %}` 标签到底对应哪一个应用的 URL 呢？

答案是：在 `URLconf` 中添加命名空间。在 `polls/urls.py` 文件中添加 `app_name` 变量作为应用的命名空间：

```
1. # polls/urls.py
2.
3. from django.conf.urls import url
4.
5. from . import views
6.
7. app_name = 'polls'
8. urlpatterns = [
9.     url(r'^$', views.index, name='index'),
10.    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
11.    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
12.    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
13. ]
```

现在，修改 `polls/index.html` 模版文件，从：

```
1. <!-- polls/templates/polls/index.html -->
2.
3. <li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

修改成：

```
1. <!-- polls/templates/polls/index.html -->
2.
3. <li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

当你弄懂如何编写视图之后，就可以去看教程的 [教程第四部分 \(zh\)](#)，来学习关于表单处理和视图类的相关内容。

## 创建你的第一个 Django 项目，第四部分

- [创建你的第一个 Django 项目，第四部分](#)
  - [编写一个简单的表单](#)
  - [使用通用视图：代码还是少点好](#)
    - [改良 URLconf](#)
    - [改良视图](#)

## 创建你的第一个 Django 项目，第四部分

这一篇从 [第三部分 \(zh\)](#) 结尾的地方继续讲起。我们将继续编写投票应用，本章着力于简单的表单处理和精简我们的代码。

## 编写一个简单的表单

让我们更新一下在上一个教程中编写的投票详细页面的模板（“polls/detail.html”），让它包含一个 HTML `<form>` 元素：

```
1. <!-- polls/templates/polls/detail.html -->
2.
3. <h1>{{ question.question_text }}</h1>
4.
5. {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
6.
7. <form action="{% url 'polls:vote' question.id %}" method="post">
8.   {% csrf_token %}
9.   {% for choice in question.choice_set.all %}
10.     <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}"
11.     />
12.     <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
13.   {% endfor %}
14.   <input type="submit" value="Vote" />
15. </form>
```

简要说明：

- 上面的模板在 Question 的每个 Choice 前添加一个单选按钮。每个单选按钮的 **value** 属性是对应的各个 Choice 的 ID。每个单选按钮的 **name** 是 “**choice**”。这意味着，当有人选择一个单选按钮并提交表单提交时，它将发送一个 POST 数据 `choice=#`，其中 # 为选择的 Choice 的 ID。这是 HTML 表单的基本概念。
- 我们设置表单的 **action** 为 `{% url 'polls:vote' question.id %}`，并设置

`method="post"`。使用 `method="post"`（与其相对的是`method="get"`）是非常重要的，因为这个提交表单的行为会改变服务器端的数据。 无论何时，当你需要创建一个改变服务器端数据的表单时，请使用 `method="post"`。这不是 Django 的特定技巧；这是优秀的网站开发实践。

- `forloop.counter` 指示 `for` 标签已经循环多少次。
- 由于我们创建一个 POST 表单（它具有修改数据的作用），所以我们需要小心跨站点请求伪造。谢天谢地，你不必太过担心，因为 Django 已经拥有一个用来防御它的非常容易使用的系统。简而言之，所有针对内部 URL 的 POST 表单都应该使用 `{% csrf_token %}` 模板标签。

现在，让我们来创建一个 Django 视图来处理提交的数据。记住，在教程 [第三部分 \(zh\)](#) 中，我们为投票应用创建了一个 `URLconf`，包含这一行：

```
1. # polls/urls.py
2.
3. url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
```

我们还创建了一个 `vote()` 函数的虚拟实现。让我们来创建一个真实的版本。 将下面的代码添加到 `polls/views.py`：

```
1. # polls/views.py
2.
3. from django.shortcuts import get_object_or_404, render
4. from django.http import HttpResponseRedirect, HttpResponse
5. from django.urls import reverse
6.
7. from .models import Choice, Question
8. # ...
9. def vote(request, question_id):
10.     question = get_object_or_404(Question, pk=question_id)
11.     try:
12.         selected_choice = question.choice_set.get(pk=request.POST['choice'])
13.     except (KeyError, Choice.DoesNotExist):
14.         # 重新显示问题的投票表单
15.         return render(request, 'polls/detail.html', {
16.             'question': question,
17.             'error_message': "You didn't select a choice.",
18.         })
19.     else:
20.         selected_choice.votes += 1
21.         selected_choice.save()
22.         # 成功处理之后 POST 数据之后，总是返回一个 HttpResponseRedirect。防止因为用户点击了后退按钮而提交了两次。
23.         return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

以上代码中有些内容还未在本教程中提到过：

- `request.POST` 是一个类字典对象，让你可以通过关键字的名字获取提交的数据。这个例子中，`request.POST['choice']` 以字符串形式返回选择的 Choice 的 ID。`request.POST` 的值永远是字符串。

注意，Django 还以同样的方式提供 `request.GET` 用于访问 GET 数据 — 但我们在代码中显式地使用 `request.POST`，以保证数据只能通过POST调用改动。

- 如果在 POST 数据中没有提供 `choice`，`request.POST['choice']` 将引发一个 `KeyError`。上面的代码检查 `KeyError`，如果没有给出 `choice` 将重新显示Question表单和一个错误信息。
- 在增加Choice的得票数之后，代码返回一个 `HttpResponseRedirect` 而不是常用的 `HttpResponse`。`HttpResponseRedirect` 只接收一个参数：用户将要被重定向的 URL（请继续看下去，我们将会解释如何构造这个例子中的 URL）。正如上面的Python注释指出的，你应该在成功处理 POST 数据后总是返回一个 `HttpResponseRedirect`。这不是 Django 的特定技巧；这是那些优秀网站在开发实践中形成的共识。
- 在这个例子中，我们在 `HttpResponseRedirect` 的构造函数中使用 `reverse()` 函数。这个函数避免了我们在视图函数中硬编码 URL。它需要我们给出我们想要跳转的视图的名字和该视图所对应的URL模式中需要给该视图提供的参数。在本例中，使用在 第三部分 (zh) 中设定的 `URLconf`，`reverse()` 调用将返回一个这样的字符串：

```
1. '/polls/3/results/'
```

其中 3 是 `question.id` 的值。重定向的 URL 将调用 `'results'` 视图来显示最终的页面。

正如在 第三部分 (zh) 中提到的，`request` 是一个 `HttpRequest` 对象。更多关于 `HttpRequest` 对象的内容，请参见 [请求和响应的文档](#)。

当有人对 Question 进行投票后，`vote()` 视图将请求重定向到 Question 的结果界面。让我们来编写这个视图：

```
1. # polls/views.py
2.
3. from django.shortcuts import get_object_or_404, render
4.
5. def results(request, question_id):
6.     question = get_object_or_404(Question, pk=question_id)
7.     return render(request, 'polls/results.html', {'question': question})
```

这和 第三部分 (zh) 中的 `detail()` 视图几乎一模一样。唯一的不同是模板的名字。我们将在稍后解决这个冗余问题。



现在，创建一个 `polls/results.html` 模板：

```
1. <!-- polls/templates/polls/results.htm -->
2.
3. <h1>{{ question.question_text }}</h1>
4.
5. <ul>
6. {% for choice in question.choice_set.all %}
7.     <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
8. {% endfor %}
9. </ul>
10.
11. <a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

现在，在你的浏览器中访问 `/polls/1/` 然后为 Question 投票。你应该看到一个投票结果页面，并且在你每次投票之后都会更新。如果你提交时没有选择任何Choice，你应该看到错误信息。

#### 注意

我们的 `vote()` 视图代码有点小问题。它首先从数据库中得到 `selected_choice` 对象，然后计算新的票数 (`votes`)，最后把新的票数存回数据库中。但如果两个用户几乎在同一时间在我们的网站上投票就会出现错误：同票数，比如说 42 张票。然后两个用户计算和保存的票数都会是 43，而不是我们期待的 44。

这就是竞争条件 (*race condition*)，如果你感兴趣，可以阅读 [使用 F\(\) 来避免竞争条件](#)，学一下如何解决这个问题。

## 使用通用视图：代码还是少点好

`detail()` (在 [第三部分 \(zh\)](#) 中) 和 `results()` 视图都很简单 — 并且，像上面提到的那样，存在冗余问题。用来显示一个议题列表的 `index()` 视图 (也在 [第三部分 \(zh\)](#) 中) 和它们类似。

这些视图反映基本的 web 开发中的一个常见情况：根据 URL 中的参数从数据库中获取数据、载入模板文件然后返回渲染后的模板。由于这种情况特别常见，Django 提供一种快捷方式，叫做“通用视图”系统。

通用视图将常见的模式抽象化，可以使你在编写应用时甚至不需要编写 Python 代码。

让我们将我们的投票应用转换成使用通用视图系统，这样我们可以删除许多我们的代码。我们仅仅需要做以下几步来完成转换：我们将：

1. 转换 URLconf。
2. 删除一些旧的、不再需要的代码。
3. 引进基于 Django 通用视图的新视图。

请继续阅读来了解详细信息。

为什么要重构代码？

一般来说，当编写一个 *Django* 应用时，你应该先评估一下通用视图是否可以解决你的问题，你应该在一开始使用它，而不是进行到一半时重构代码。本教程目前为止是有意将重点放在以“艰难的方式”编写视图，这是为将重点放在核心概念上。

就像在使用计算器之前你需要知道基本的数学一样。

## 改良 URLconf

首先，打开 `polls/urls.py` 这个 URLconf 并将它修改成：

```
1. # polls/urls.py
2.
3. from django.conf.urls import url
4.
5. from . import views
6.
7. app_name = 'polls'
8. urlpatterns = [
9.     url(r'^$', views.IndexView.as_view(), name='index'),
10.    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
11.    url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(), name='results'),
12.    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
13. ]
```

注意在第二个和第三个模式的正则表达式中，匹配的模式的名字由 `<question_id>` 变成 `<pk>`。

## 改良视图

下一步，我们将删除旧的 `index`、`detail` 和 `results` 视图，并用 Django 的通用视图代替。打开 `polls/views.py` 文件，并将它修改成：

```
1. # polls/views.py
2.
3. from django.shortcuts import get_object_or_404, render
4. from django.http import HttpResponseRedirect
5. from django.urls import reverse
6. from django.views import generic
7.
8. from .models import Choice, Question
9.
10.
11. class IndexView(generic.ListView):
12.     template_name = 'polls/index.html'
13.     context_object_name = 'latest_question_list'
14.
```

```

15.     def get_queryset(self):
16.         """ 返回最近时间的五个问题 """
17.         return Question.objects.order_by('-pub_date')[:5]
18.
19.
20. class DetailView(generic.DetailView):
21.     model = Question
22.     template_name = 'polls/detail.html'
23.
24.
25. class ResultsView(generic.DetailView):
26.     model = Question
27.     template_name = 'polls/results.html'
28.
29.
30. def vote(request, question_id):
31.     ... # 像上面一样保存

```

我们在这里使用两个通用视图：`ListView` 和 `DetailView`。这两个视图分别抽象“显示一个对象列表”和“显示一个特定类型对象的详细信息页面”这两种概念。

- 每个通用视图需要知道它将作用于哪个模型。这由 `model` 属性提供。
- `DetailView` 期望从 URL 中捕获名为“pk”的主键值，所以我们为通用视图把 `question_id` 改成 `pk`。

默认情况下，通用视图 `DetailView` 使用一个叫做“<app name>/<model name>\_detail.html”的模板。在我们的例子中，它将使用“polls/question\_detail.html”模板。`template_name` 属性是用来告诉 Django 使用一个指定的模板名字，而不是自动生成的默认名字。我们也为 `results` 列表视图指定了 `template_name` — 这确保 `results` 视图和 `detail` 视图在渲染时具有不同的外观，即使它们在后台都是同一个 `DetailView`。

类似地，`ListView` 使用一个叫做“<app name>/<model name>\_detail.html”的默认模板；我们使用 `template_name` 来告诉 `ListView` 使用已经存在的“polls/index.html”模板。

在之前的教程中，提供模板文件时都带有一个包含 `question` 和 `latest_question_list` 变量的 `context`。对于 `DetailView`，`question` 变量会自动提供 — 因为我们使用 Django 的模型 (`Question`)，Django 能够为 `context` 变量决定一个合适的名字。然而对于 `ListView`，自动生成的 `context` 变量是 `question_list`。为了覆盖这个行为，我们提供 `context_object_name` 属性，表示我们想使用 `latest_question_list`。作为一种替换方案，你可以改变你的模板来匹配新的 `context` 变量 — 但它告诉 Django 使用你想使用的变量名更容易多了。

启动服务器，使用一下基于通用视图的新投票应用。

更多关于通用视图的详细信息，请查看 [通用视图的文档](#)。

当你明白了这些表单和通用视图后，可以继续阅读 [教程第五部分 \(zh\)](#) 来了解如何测试我们的投票应用。

## 创建你的第一个 Django 项目， 第五部分

- 创建你的第一个 Django 项目， 第五部分
  - 自动化测试简介
    - 自动化测试是什么？
    - 为什么你需要写测试
      - 测试将节约你的时间
      - 测试不仅能发现错误，而且能预防错误
      - 测试使你的代码更有吸引力
      - 测试有利于团队协作
  - 基本测试策略
  - 第一个测试
    - 首先得有个 Bug
    - 写个测试来发现 Bug
    - 运行测试
    - 修复 Bug
    - 更全面的测试
  - 测试视图
    - 针对视图的测试
    - Django 测试工具之 Client
    - 改善视图代码
    - 测试新视图
    - 测试 DetailView
    - 更多的测试
  - 测试那是越多越好
  - 进一步测试
  - 然后呢？

## 创建你的第一个 Django 项目， 第五部分

这一篇从 [第四部分 \(zh\)](#) 结尾的地方继续讲起。我们在前几章成功的构建了一个在线投票应用，本章我们将为其创建一些自动化测试。

## 自动化测试简介

### 自动化测试是什么？

测试，是用来检查代码正确性的一些简单的程序。

测试在不同的层次中都存在。有些测试只关注某个很小的细节（某个模型的某个方法的返回值是否满足预期？），而另一些测试可能检查对某个软件的一系列操作（某一用户输入序列是否造成了预期的结果？）。其实这和我们在教程的 [第二部分\(zh\)](#) 里做的并没有什么不同，我们使用 `shell` 来测试某一方法的功能，或者运行某个应用并输入数据来检查它的行为。

真正不同的地方在于，自动化测试是由某个系统帮你自动完成的。当你创建好了一系列测试，每次修改应用代码后，就可以自动检查出修改后的代码是否还像你曾经预期的那样正常工作。你不需要花费大量时间来进行手动测试。

## 为什么你需要写测试

但是，为什么需要测试呢？又为什么是现在呢？

你可能觉得学 Python/Django 对你来说已经很充实了，再学一些新东西的话看起来有点负担过重并且没什么必要。毕竟，我们的投票应用看起来已经完美工作了。写一些自动测试并不能让它工作的更好。如果写一个投票应用是你想用 Django 完成的唯一工作，那你确实没必要学写测试。但是如果你还想写更复杂的项目，现在就是学习测试写法的最好时机了。

## 测试将节约你的时间

在某种程度上，能够「判断出代码是否正常工作」的测试，就称得上是个令人满意的了。

在更加复杂的应用中，各种组件之间的交互可能会及其的复杂。改变其中某一组件的行为，也有可能造成意想不到的结果。判断「代码是否正常工作」意味着你需要用大量的数据来完整的测试全部代码的功能，以确保你的小修改没有对应用整体造成破坏——这太费时间了。

尤其是当你发现自动化测试能在几秒钟之内帮你完成这件事时，就更会觉得手动测试实在是太浪费时间了。当某人写出错误的代码时，自动化测试还能帮助你定位错误代码的位置。

有时候你会觉得，和富有创造性和生产力的业务代码比起来，编写枯燥的测试代码实在是太无聊了，特别是当你知道你的代码完全没有问题的时候。

然而，编写测试还是要比花费几个小时手动测试你的应用，或者为了找到某个小错误而胡乱翻看代码要有意义的多。

## 测试不仅能发现错误，而且能预防错误

「测试是开发的对立面」，这种思想是不对的。

如果没有测试，整个应用的行为意图会变得更加的不清晰。甚至当你在看自己写的代码时也是这样，有时候你需要仔细研读一段代码才能搞清楚它有什么用。

而测试的出现改变了这种情况。测试就好像是从内部仔细检查你的代码，当有些地方出错时，这些地方将会变得很显眼——就算你自己没有意识到那里写错了。

## 测试使你的代码更有吸引力

你也许遇到过这种情况：你编写了一个绝赞的软件，但是其他开发者看都不看它一眼，因为它缺少测试。没有测试的代码不值得信任。Django 最初开发者之一的 Jacob Kaplan-Moss 说过：“项目规划时没有包含测试是不科学的。”

其他的开发者希望在正式使用你的代码前看到它通过了测试，这是你需要写测试的另一个重要原因。

## 测试有利于团队协作

前面的几点都是从单人开发的角度来说的。复杂的应用可能由团队维护。测试的存在保证了协作者不会不小心破坏了你的代码（也保证你不会不小心弄坏他们的）。如果你想作为一个 Django 程序员谋生的话，你必须擅长编写测试！

## 基本测试策略

---

测试有几种不同的应用方法。

一些开发者遵循 [测试驱动开发 \(test-driven development\)](#) 的原则，他们在写代码之前先写测试。这种方法看起来有点反直觉，但事实上，这和大多数人日常的做法是相吻合的。我们会先描述一个问题，然后写代码来解决它。「测试驱动」的开发方法只是将问题的描述抽象为了 Python 的测试样例。

更普遍的情况是，一个刚接触自动化测试的新手更倾向于先写代码，然后再写测试。虽然提前写测试可能更好，但是晚点写起码也比没有强。

有时候很难决定从测试该哪里开始下手。如果你已经写了几千行 Python 代码了，选择从哪里开始写测试确实不怎么简单。如果是这种情况，那么在你下次修改代码（比如加新功能，或者修复 Bug）之前写个测试是比较合理且有效的。

所以，我们现在就开始写吧。

## 第一个测试

---

### 首先得有个 Bug

幸运的是，我们的投票（polls）应用现在就有一个小 Bug 需要被修复：我们的要求是如果 **Question** 是在一天之内发布的，`Question.was_published_recently()` 方法将会返回 **True**，然而现在这个方法在 **Question** 的 `pub_date` 字段比当前时间还晚时也会返回 **True**（这是个 Bug）。

你能从管理页面确认这个 Bug。创建一个发布日期是将来的投票，在投票列表里你会看到它被标明为

最近发布 (published recently)。

也可以从 `shell` 里确认 Bug：

```
1. >>> import datetime
2. >>> from django.utils import timezone
3. >>> from polls.models import Question
4. >>> # 创建一个 30 天后发布的投票
5. >>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
6. >>> # 检查它是否是最近发布的
7. >>> future_question.was_published_recently()
8. True
```

因为将来发生的是肯定不是最近发生的，所以代码明显是错误的。

## 写个测试来发现 Bug

我们刚刚在 `shell` 里做的测试也就是自动化测试应该做的工作。所以我们来把它改写成自动化的吧。

按照惯例，Django 应用的测试因该写在应用的 `tests.py` 文件里。测试系统会自动的在所有以 `test` 开头的文件里寻找并执行测试代码。

在 `polls` 应用的 `tests.py` 文件里输入以下代码：

```
1. # polls/tests.py
2.
3. import datetime
4.
5. from django.utils import timezone
6. from django.test import TestCase
7.
8. from .models import Question
9.
10.
11. class QuestionModelTests(TestCase):
12.
13.     def test_was_published_recently_with_future_question(self):
14.         """
15.         was_published_recently() 应该对 pub_date 字段值是将来的那些问题返回 False。
16.         """
17.         time = timezone.now() + datetime.timedelta(days=30)
18.         future_question = Question(pub_date=time)
19.         self.assertIs(future_question.was_published_recently(), False)
```



我们创建了一个 `django.test.TestCase` 的子类，并添加了一个方法。在此方法中我们创建了一个 `pub_date` 字段值在将来的 `Question` 实例，然后检查它的 `was_published_recently()` 方法的返回值——它应该是 `False`。

## 运行测试

在终端中，我们通过输入以下代码运行测试：

```
1. $ python manage.py test polls
```

你将会看到运行结果：

```
1. Creating test database for alias 'default'...
2. System check identified no issues (0 silenced).
3. F
4. =====
5. FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionModelTests)
6. -----
7. Traceback (most recent call last):
8.   File "/path/to/mysite/polls/tests.py", line 16, in
     test_was_published_recently_with_future_question
9.     self.assertIs(future_question.was_published_recently(), False)
10. AssertionError: True is not False
11.
12. -----
13. Ran 1 test in 0.001s
14.
15. FAILED (failures=1)
16. Destroying test database for alias 'default'...
```

发生了什么呢？以下是自动化测试的运行过程：

- `python manage.py test polls` 将会寻找 `poll` 应用里的测试代码
- 它找到了一个 `django.test.TestCase` 的子类
- 它创建一个特殊的数据库供测试使用
- 它在类中寻找测试方法——以 `test` 开头的方法。
- 在 `test_was_published_recently_with_future_question` 方法中，它创建了一个 `pub_date` 值为未来第 30 天的 `Question` 实例。
- 然后使用 `assertIs()` 方法，发现 `was_published_recently()` 返回了 `True`，而我们希望它返回 `False`

测试系统通知我们哪些测试样例失败了，和造成测试失败的代码所在的行号。

## 修复 Bug

我们现在知道了，问题出在当 `pub_date` 为将来时，`Question.was_published_recently()` 应该返回 `False`。我们去修改 `models.py` 里的方法，让它只在日期是过去的时候才返回 `True`：

```
1. # polls/model.py
2.
3. def was_published_recently(self):
4.     now = timezone.now()
5.     return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

然后我们重新运行测试：

```
1. Creating test database for alias 'default'...
2. System check identified no issues (0 silenced).
3. .
4. -----
5. Ran 1 test in 0.001s
6.
7. OK
8. Destroying test database for alias 'default'...
```

在出现 Bug 之后，我们编写了能够发现这个 Bug 的自动化测试。在修复 Bug 之后，我们的代码顺利的通过了测试。

将来，我们的应用可能会出现其他的问题，但是我们可以肯定的是，一定不会再次出现这个 Bug，因为只要简单的运行一遍测试，就会立刻收到警告。我们可以认为应用的这一小部分代码永远是安全的。

## 更全面的测试

我们已经搞定一小部分了，现在可以考虑全面的测试 `was_published_recently()` 这个方法以确定它的安全性，然后就可以把这个方法稳定下来了。事实上，在修复一个 Bug 时不小心引入另一个 Bug 会是非常令人尴尬的。

我们在上次写的类里再增加两个测试，来更全面的测试这个方法：

```
1. # polls/tests.py
2.
3. def test_was_published_recently_with_old_question(self):
4.     """
5.     对于 pub_date 在一天以前的 Question, was_published_recently() 应该返回 False。
6.     """
7.     time = timezone.now() - datetime.timedelta(days=1, seconds=1)
8.     old_question = Question(pub_date=time)
9.     self.assertIs(old_question.was_published_recently(), False)
10.
```

```

11. def test_was_published_recently_with_recent_question(self):
12.     """
13.     对于 pub_date 在一天之内的 Question, was_published_recently() 应该返回 True。
14.     """
15.     time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
16.     recent_question = Question(pub_date=time)
17.     self.assertIs(recent_question.was_published_recently(), True)

```

现在，我们三个测试来确保 `Question.was_published_recently()` 方法对于过去，最近，和将来的三种情况都返回正确的值。

再次申明，尽管 `polls` 现在是个非常简单的应用，但是无论它以后成长到多么复杂，要和其他代码进行怎样的交互，我们都能保证进行过测试的那些方法的行为永远是符合预期的。

## 测试视图

我们的投票应用对所有问题都一视同仁：它将会发布所有的问题，也包括那些 `pub_date` 字段值是未来的问题。我们应该改善这一点。将 `pub_date` 设置为将来应该被解释为这个问题将在所填写的时间点才被发布，而在之前是不可见的。

## 针对视图的测试

为了修复上述 Bug，我们这次先编写测试，然后再去改代码。事实上，这是一个「测试驱动」开发模式的实例，但其实这两者的顺序不太重要。

在我们的第一个测试中，我们关注代码的内部行为。我们通过假装有用户使用浏览器访问被测试的应用来检查代码行为是否符合预期。

在我们动手之前，先看看需要用到的工具们。

## Django 测试工具之 Client

Django 提供了一个供测试使用的 `Client` 来模拟用户和视图层代码的交互。我们能在 `tests.py` 甚至是 `shell` 中使用它。

我们依照惯例从 `shell` 开始，首先我们要做一些在 `tests.py` 里并不需要的准备工作。第一步是在 `shell` 中配置测试环境：

```

1. >>> from django.test.utils import setup_test_environment
2. >>> setup_test_environment()

```

`setup_test_environment()` 安装了一个特殊的模板渲染器，它能让我们使用 `response` 的一

些在正常情况下不可用的附加属性，比如 `response.context`。注意，这个方法并不会配置测试数据库，所以接下来的代码将会当前存在的数据库上运行，输出的内容可能由于数据库内容的不同而不同。如果你在 `settings.py` 中的 `TIME_ZONE` 设置不对，你还有可能得到意想不到的结果。如果你不记得之前是否设置过，那在继续下面的步骤之前先检查一下。

然后我们需要导入 `client` 类（在后续 `tests.py` 的实例中我们将会使用 `django.test.TestCase` 类，这个类里包含了自己的 `client` 实例，所以不需要这一步）

```
1. >>> from django.test import Client
2. >>> # 创建一个 Client 实例
3. >>> client = Client()
```

搞定了之后，我们可以要求 `client` 来为我们工作了：

```
1. >>> # 获取 '/' 的响应
2. >>> response = client.get('/')
3. Not Found: /
4. >>> # 我们期望返回一个 404；
5. 如果你看到一个
6. >>> # “无效 HTTP_HOST_header” 错误 和 一个
7. 400 响应，那你可能
8. >>> # 忘记了这之前要调用的 setup_test_environment()
9. >>> response.status_code
10. 404
11. >>> # 访问 '/polls/' 的话，我们应该会得到一些有意义的响应
12. >>> # 我们使用 'reverse()' 方法而不是硬编码的 URL
13. >>> from django.urls import reverse
14. >>> response = client.get(reverse('polls:index'))
15. >>> response.status_code
16. 200
17. >>> response.content
18. b'\n    <ul>\n        \n        <li><a href="/polls/1/">What&#39;s up?</a></li>\n    \n    </ul>\n\n'
19. >>> response.context['latest_question_list']
20. <QuerySet [<Question: What's up?>]>
```

## 改善视图代码

现在的投票列表会显示将来的投票（`pub_date` 值是将来的那些）。我们来修复这个问题。

在教程的 [第四部分 \(zh\)](#) 里，我们介绍了基于 `ListView` 的视图类：

```
1. # polls/views.py
2.
3. class IndexView(generic.ListView):
```

```

4.     template_name = 'polls/index.html'
5.     context_object_name = 'latest_question_list'
6.
7.     def get_queryset(self):
8.         """返回最近发布的五个投票"""
9.         return Question.objects.order_by('-pub_date')[:5]

```

我们需要改进 `get_queryset()` 方法，让他它能通过将 `Question` 的 `pub_date` 属性与 `timezone.now()` 相比较来判断是否应该显示此 `Question`。首先我们需要一行 `import` 语句：

```

1. # polls/views.py
2.
3. from django.utils import timezone

```

然后我们把 `get_queryset` 方法改写成下面这样：

```

1. # polls/views.py
2.
3. def get_queryset(self):
4.     """返回最近发布的五个投票（不包括那些被设置为在将来发布的）"""
5.     return Question.objects.filter(
6.         pub_date__lte=timezone.now()
7.     ).order_by('-pub_date')[:5]

```

`Question.objects.filter(pub_date__lte=timezone.now())` 返回一个由 `pub_date` 小于或等于（也就是早于或等于）`timezone.now` 的 `Question` 组成的集合。

## 测试新视图

现在你可以通过创建一个将来发布的投票问题，然后执行 `runserver`，再在浏览器中检查主页里的列表来判断代码是否符合预期。你绝对不会愿意每次修改代码后都这么来一次的，所以让我们创建一些自动化测试吧。

在 `polls/tests.py` 里加入一句 `import`

```

1. # polls/tests.py
2.
3. from django.urls import reverse

```

然后我们写一个公用的快捷函数用于创建投票问题，再为视图创建一个测试类：

```

1. # polls/test.py
2.
3. def create_question(question_text, days):

```

```

4.     """
5.     创建一个以 question_text 为标题，pub_date 为 days 天之后的问题。
6.     days 为正表示将来，为负表示过去。
7.     """
8.     time = timezone.now() + datetime.timedelta(days=days)
9.     return Question.objects.create(question_text=question_text, pub_date=time)
10.
11.
12. class QuestionIndexViewTests(TestCase):
13.     def test_no_questions(self):
14.         """
15.         如果数据库里没有保存问题，应该显示一个合适的提示信息。
16.         """
17.         response = self.client.get(reverse('polls:index'))
18.         self.assertEqual(response.status_code, 200)
19.         self.assertContains(response, "No polls are available.")
20.         self.assertQuerysetEqual(response.context['latest_question_list'], [])
21.
22.     def test_past_question(self):
23.         """
24.         值 pub_date 是过去的，问题应该被显示在主页上。
25.         """
26.         create_question(question_text="Past question.", days=-30)
27.         response = self.client.get(reverse('polls:index'))
28.         self.assertQuerysetEqual(
29.             response.context['latest_question_list'],
30.             ['<Question: Past question.>']
31.         )
32.
33.     def test_future_question(self):
34.         """
35.         值 pub_date 是将来的，问题不应该被显示在主页上。
36.         """
37.         create_question(question_text="Future question.", days=30)
38.         response = self.client.get(reverse('polls:index'))
39.         self.assertContains(response, "No polls are available.")
40.         self.assertQuerysetEqual(response.context['latest_question_list'], [])
41.
42.     def test_future_question_and_past_question(self):
43.         """
44.         如果数据库里同时存有过去和将来的投票，那么只应该显示过去那些。
45.         """
46.         create_question(question_text="Past question.", days=-30)
47.         create_question(question_text="Future question.", days=30)
48.         response = self.client.get(reverse('polls:index'))
49.         self.assertQuerysetEqual(
50.             response.context['latest_question_list'],
51.             ['<Question: Past question.>']

```

```

52.         )
53.
54.     def test_two_past_questions(self):
55.         """
56.         问题索引页应该可以显示多个问题。
57.         """
58.         create_question(question_text="Past question 1.", days=-30)
59.         create_question(question_text="Past question 2.", days=-5)
60.         response = self.client.get(reverse('polls:index'))
61.         self.assertQuerysetEqual(
62.             response.context['latest_question_list'],
63.             ['<Question: Past question 2.>', '<Question: Past question 1.>']
64.         )

```

我们仔细分析一下上面的代码。

首先是一个快捷函数 `create_question`，它封装了创建问题（questions）的流程，减少了重复代码。

`test_no_questions` 方法里没有创建任何投票，它检查返回的网页上有没有 “No polls are available.” 这段消息和 `latest_question_list` 是否为空。注意到 `django.test.TestCase` 类提供了一些额外的 `assert` 方法，在这个例子中，我们使用了 `assertContains()` 和 `assertQuerysetEqual()`。

在 `test_past_question` 方法中，我们创建了一个投票并检查它是否出现在列表中。

在 `test_future_questionn` 方法中，我们创建了一个 `pub_date` 在将来的投票。数据库会在每次调用测试方法之前被重置，所以第一个方法里创建的投票已经没了，所以此时我们希望看到的是一个没有任何投票的目录页。

剩下的那些也都差不多。实际上，测试就是假装一些管理员的输入，然后通过用户端的表现是否符合预期来判断新加入的改变是否破坏了原有的系统状态。

## 测试 DetailView

我们的工作似乎已经很完美了？不，还有一个问题：就算在将来发布的那些投票不会在目录页里出现，但是用户还是能够通过猜测 URL 的方式访问到他们。所以我们得在 `DetailView` 里增加一些约束：

```

1. # polls/views.py
2.
3. class DetailView(generic.DetailView):
4.     ...
5.     def get_queryset(self):
6.         """
7.         过滤掉现在不应该被发布的投票。

```

```

8.         """
9.         return Question.objects.filter(pub_date__lte=timezone.now())

```

当然，我们还要增加一些测试，用于确认过去的 问题（**Question**）能被用户访问，而将来的则不能：

```

1. # polls/tests.py
2.
3. class QuestionDetailViewTests(TestCase):
4.     def test_future_question(self):
5.         """
6.         访问将来发布的问题详情页应该会收到一个 404 错误。
7.         """
8.         future_question = create_question(question_text='Future question.', days=5)
9.         url = reverse('polls:detail', args=(future_question.id,))
10.        response = self.client.get(url)
11.        self.assertEqual(response.status_code, 404)
12.
13.    def test_past_question(self):
14.        """
15.        访问过去发布的问题详情页，页面上应该显示问题描述。
16.        """
17.        past_question = create_question(question_text='Past Question.', days=-5)
18.        url = reverse('polls:detail', args=(past_question.id,))
19.        response = self.client.get(url)
20.        self.assertContains(response, past_question.question_text)

```

## 更多的测试

我们应该给 **ResultsView** 也增加一个类似的 **get\_queryset** 方法，并且为它创建测试。这和我们之前干的差不多，事实上，基本就是重复一遍。

我们还可以从各个方面改进应用，但是测试会一直伴随我们。比方说，在目录页上显示一个没有选项（**Choices**）的问题就没什么意义。我们可以检查并排除这样的问题（**Questions**）。测试里则可以创建一个没有选项的问题，然后检查它是否被显示在目录上。当然也要创建一个有选项的投票，然后确认它确实被显示了。

恩，也许你想让登录的管理员能在目录上够看见未被发布的那些问题，但是其他的用户看不到。不管怎么说，如果你想要增加一个新功能，那么同时一定要为它编写测试。不过你是先写代码还是先写测试那就随你了。

在未来的某个时刻，你一定会去查看测试代码，然后开始怀疑：「这么多的测试不会使代码越来越复杂吗？」。别着急，我们马上就会谈到这一点。



## 测试那是越多越好

---

貌似我们的测试多的快要失去控制了。按照这样发展下去，测试代码就要变得比应用的实际代码还要多了。而且测试代码大多都是重复且不优雅的，特别是在和业务代码比起来的时候，这种感觉更加明显。

但是这没关系！就让测试代码继续肆意增长吧。大部分情况下，你写完一个测试之后就可以忘掉它了。在你继续开发的过程中，它会一直默默无闻地为你做贡献的。

但有时测试也需要更新。想象一下如果我们真的想让目录只显示有选项的那些投票，那么只前写的很多测试就都会失败。但是 这也明确地告诉了我们哪些测试需要被更新，所以增加的测试会自行测试向前的兼容性。

最坏的情况是，当你继续开发的时候，发现之前的一些测试现在看来是多余的。但是这也不是什么问题，冗余的测试也是件好事。

如果你对测试有个整体规划，那么它们就几乎不会变得混乱。下面有几条好的建议：

- 对于每个模型和视图都建立单独的测试类
- 每个测试方法之测试一个功能
- 给每个测试方法起个能描述其功能的名字

## 进一步测试

---

在本教程中，我们仅仅是了解了测试的基础知识。你能做的还有很多，而且世界上有很多有用的工具来帮你完成这些有意义的事。

举个例子，在我们上述的测试中，已经从代码逻辑和视图响应的角度检查了应用的输出，现在你可以从一个更加用户的角度来检查最终渲染出的 HTML 是否符合预期，使用 [Selenium](#) 可以很轻松的完成这件事。这个工具不仅可以测试 Django 框架里的代码，还可以检查其他部分，比如说你的 JavaScript。它假装成是一个正在和你站点进行交互的浏览器，就好像有个真人在访问网站一样！Django 它提供了 [LiveServerTestCase](#) 来和 Selenium 这样的工具进行交互。

如果你在开发一个很复杂的应用的话，你也许想在每次提交代码时自动运行测试，也就是我们所说的[持续整合](#)，这样的话就实现质量控制的自动化，起码是部分自动化。

一个找出代码中未被测试部分的方法是检查代码覆盖率。它有助于找出代码中的薄弱部分和无用部分。如果你无法测试一段代码，通常说明这段代码需要被重构或者删除。想知道代码覆盖率和无用代码的详细信息，请看文档 [和 coverage.py 结合使用](#)。

## 然后呢？

---

如果你想深入了解测试，就去看 [在 Django 中测试](#)。

当你已经比较熟悉该如何测试 Django 的视图之后，就可以继续于读 [教程第六部分 \(zh\)](#)，来学习关于静态文件管理的相关知识。

## 创建你的第一个 Django 项目，第六部分

- [创建你的第一个 Django 项目，第六部分](#)
  - [自定义应用的界面和风格](#)
  - [添加背景图片](#)

## 创建你的第一个 Django 项目，第六部分

这一篇从 [第五部分 \(zh\)](#) 结尾的地方继续讲起。我们已经为投票程序编写了测试，而现在我们要为它加上样式和图片。

除了服务端生成的 HTML 以外，网络应用通常需要一些其他的文件——比如图片，脚本和样式表——来帮助渲染网络页面。在 Django 中，我们把这些文件统称为“静态文件”。

对于小项目来说，这个问题没什么大不了的，因为你可以把这些静态文件随便放在哪，只要服务程序能够找到它们就行。然而在大项目——特别是由好几个应用组成的大项目中，处理不同应用所需要的静态文件的工作就显得有点麻烦了。

这就是 `django.contrib.staticfiles` 存在的意义：它将各个应用的静态文件（和一些你指明的目录里的文件）统一收集起来，这样一来，在生产环境中，这些文件就会集中在一个便于分发的地方。

## 自定义应用的界面和风格

首先，在 `polls` 目录下创建一个 `static` 目录。Django 将会从这里收集静态文件，就像 Django 在 `polls/templates` 里寻找模板一样。

Django 的 `STATICFILES_FINDERS` 设置项是一个列表，它包含了若干个知道如何从不同地点寻找静态文件的搜寻器。其中之一是 `AppDirectoriesFinder`，它会从 `INSTALLED_APPS` 中各个应用的“static”子目录中寻找文件，比如刚刚创建的 `polls/static`。之前的管理页面也使用了相同的目录结构来管理静态文件。

在刚创建的 `static` 目录下创建一个 `polls` 目录，再在其中新建文件 `style.css`。也就是说，你的样式文件的位置是 `polls/static/polls/style.css`。由于 `AppDirectoriesFinder` 的存在，你可以在 Django 中使用 `polls/style.css` 来引用这个文件，就像我们引用模板文件那样。

### 静态文件命名空间

和模板一样，虽然我们可以直接把样式表放在 `polls/static` 目录下（而不是再创建一个 `polls` 子目录），但是这并不是个好主意。Django 将会使用它找到的第一个和名称项匹配文件，当你在另一个应用里也有一个同名的文件的话，Django 将无法区分它们。我们想让 Django 能够找到正确的文件，最简单的方法就是使用命名空间。也就是把静态文件放到一个和应用同名的子目录中。

将以下代码写入样式表（`polls/static/polls/style.css`）中：

```
1. /* polls/static/polls/style.css */
2.
3. li a {
4.     color: green;
5. }
```

下一步，在 `polls/templates/polls/index.html` 的顶部加入以下内容：

```
1. <!-- polls/templates/polls/index.html -->
2.
3. {% load static %}
4.
5. <link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

`{% static %}` 语句会把对静态文件的引用转化为绝对地址。

这就是你需要做的所有工作了。现在重新载入 <http://localhost:8000/polls/>，你会看到问题链接变成了绿色（Django 使用的样式），这说明样式表已经正常工作了。

## 添加背景图片

接下来，我们要为图片新建一个子目录。在 `polls/static/polls` 目录下创建 `images` 目录。然后在其中放入背景图片 `background.gif`。也就是说，背景图片路径为 `polls/static/polls/images/background.gif`。

然后，在样式表文件（`polls/static/polls/style.css`）中加入：

```
1. /* polls/static/polls/style.css */
2.
3. body {
4.     background: white url("images/background.gif") no-repeat right bottom;
5. }
```

重新加载 <http://localhost:8000/polls/> 你将会看到背景图片被放置在了页面的右下角。

### 警告

显然 `{% static %}` 模板标签是无法在静态文件中使用的，比如样式表文件，因为它们不是 Django 生成的。在静态文件中，你应该使用 相对路径（*relative paths*）来互相引用。因为这样的话，你可以通过改变 `STATIC_URL`（`static` 模块用它来生成 URL）达到改变静态文件地址的目的，而不需要在大量的文件中逐一修改引用文件的地址。

这些都只是 基础（**basics**）。关于静态文件的各种设置，和框架中其余和静态文件有关的部分，请

参考 [the static files howto](#) 和 [the staticfiles reference](#)。另外，[Deploying static files](#) 讨论了在真实的服务器上组织静态文件的方法。

如果你明白静态文件，来阅读 [教程第七部分 \(zh\)](#) 学习如何自定义 Django 自动生成的管理站点吧！

## 创建你的第一个 Django 项目，第七部分

- [创建你的第一个 Django 项目，第七部分](#)
  - [自定义管理表单](#)
  - [添加关联对象](#)
  - [自定义对象列表](#)
  - [自定义管理页面样式](#)
    - [自定义项目模板](#)
    - [自定义应用模板](#)
  - [自定义管理页面索引页](#)
  - [下一步？](#)

## 创建你的第一个 Django 项目，第七部分

这一篇从 [第六部分 \(zh\)](#) 结尾的地方继续讲起。我们继续在投票程序上下功夫，本章着力于自定义 Django 自动生成的管理站点（在 [第二部分 \(zh\)](#) 已有涉及）。

### 自定义管理表单

通过使用 `admin.site.register(Question)` 注册 `问题 (Question)` 模型，Django 能构造一个出一个默认的表单样式。通常，你会想自定义表单的样式和作用。你能通过在注册对象时增加一些选项来达到这一目的。

来看看如何改变表单中字段的顺序。把 `admin.site.register(Question)` 替换成下面的代码：

```
1. # polls/admin.py
2.
3. from django.contrib import admin
4.
5. from .models import Question
6.
7.
8. class QuestionAdmin(admin.ModelAdmin):
9.     fields = ['pub_date', 'question_text']
10.
11. admin.site.register(Question, QuestionAdmin)
```

当你想改变管理页面的某些选项时，步骤一般是这样的：创建模型管理对象（model admin object），然后把它当作第二个参数传递给 `admin.site.register()` 函数。



上面的代码会使“发布日期 (Publication date)”出现在“问题说明 (Question text)”

字段之前。

Home > Polls > Questions > What's up?

## Change question

Date published:

Date: 2015-09-06 Today 
  
Time: 21:16:20 Now 

---

Question text:

What's up?

---

如果只有两个字段的话感觉不出什么，但是当表单有十几个字段时，选择一个直观的顺序是个可以提升用户体验细节。

如果真的有十几个字段的话，你可能想将它们分组，也即分为多个字段集（`fieldsets`）：

```

1. # polls/admin.py
2.
3. from django.contrib import admin
4.
5. from .models import Question
6.
7.
8. class QuestionAdmin(admin.ModelAdmin):
9.     fieldsets = [
10.         (None, {'fields': ['question_text']}),
11.         ('Date information', {'fields': ['pub_date']}),
12.     ]
13.
14. admin.site.register(Question, QuestionAdmin)

```

**fieldsets** 列表里的元组的第一个元素是分组名。下面是表单现在的样子：

Home > Polls > Questions > What's up?

## Change question

Question text:

What's up?

### Date information

Date published:

Date:

2015-09-06

Today



Time:

21:16:20

Now



## 添加关联对象

现在我们已经可以管理问题（**Question**）了。但是每个 问题（**Question**） 都有多个 选项（**Choices**），而且管理页面里并没有选项。

不过马上就有了。

有两个方法来解决这个问题。第一种是像注册 问题（**Question**） 一样去注册 选项（**Choices**）。这很简单：


```
1. # polls/admin.py
2.
3. from django.contrib import admin
4.
5. from .models import Choice, Question
6. # ...
7. admin.site.register(Choice)
```

现在，Django 的管理页面里就有 “选项（Choices）” 了。“添加选项（Add choices）” 的表单看起来就像这样：



Home › Polls › Choices › Add choice

## Add choice

Question:	<input type="text" value="-----"/>   
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>

在这个表单中，“问题（Question）”字段表现为一个包含所有数据库里储存着的问题（question）的选择框。Django 知道 [外键（ForeignKey）](#) 应该被表示为 选择框（\）。现在，列表里只有一个问题（question）。

“问题（Question）”旁边的“添加（Add Another）”按钮也值得注意一下。每一个 外键（ForeignKey） 字段都会有这个按钮。当你点击它时，会弹出一个带有“添加问题（Add question）”表单的窗口。如果使用它添加了问题（Question）并点击了“保存（Save）”，Django 会将问题（Question）对象储存在数据库中然后动态的将其加入到之前的“添加选项”表单中的问题（Question）选择框中。

但是，说实话，这样添加选项（Choices）到数据库中是非常低效的。如果当你添加一个问题（Question）时能够同时为它加上几个选项的话那就再好不过了。来看看怎么实现：

删掉 选项（Choices） 模型的 `register()` 语句，然后编辑 问题（Question） 的注册代码：

```

1. # polls/admin.py
2.
3. from django.contrib import admin
4.
5. from .models import Choice, Question
6.
7.
8. class ChoiceInline(admin.StackedInline):
9.     model = Choice
10.    extra = 3
11.
12.
13. class QuestionAdmin(admin.ModelAdmin):
14.     fieldsets = [
15.         (None, {'fields': ['question_text']}),
16.         ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),

```

```

17.     ]
18.     inlines = [ChoiceInline]
19.
20. admin.site.register(Question, QuestionAdmin)

```

增加的代码将会告诉 Django：“**Choice**（选项）对象将会在 问题（**Question**）的管理界面里被编辑。默认显示3个选项字段以供编辑。”

重新打开“添加问题（Add question）”页面：

CHOICES			
CHOICE TEXT		VOTES	DELETE?
<input type="text"/>		<input type="text" value="0"/>	
<input type="text"/>		<input type="text" value="0"/>	
<input type="text"/>		<input type="text" value="0"/>	
<a href="#">+ Add another Choice</a>			

现在有3个可以添加相关选项（choices）的单元 — “3” 是由代码里的 `extra` 所规定的——并且当你每次进入一个已经创建好的对象的修改界面时，总会多出另外三个单元让你可以添加选项。

在三个添加选项单元的最下方有一个“添加一个选项（Choices）”按钮。但你点击它时，上方会增加一个添加选项的单元。如果你想删掉添加的单元，可以点击单元右上角的X。请注意：你无法移除初始的那三个单元。下面这张图片显示新增了一个单元的效果：

CHOICES

Choice: #1

Choice text:

Votes:

Choice: #2

Choice text:

Votes:

Choice: #3

Choice text:

Votes:

Choice: #4 ✕

Choice text:

Votes:

[+ Add another Choice](#)

还有个小问题。显示所有的相关的选项（Choices）单元实在是太占屏幕空间了。由于这个原因，Django 提供了表格式的视图；你只需要改一下 `ChoiceInline` 的定义就行了：

```
1. # polls/admin.py
2.
3. class ChoiceInline(admin.TabularInline):
4.     #...
```

通过使用 `TabularInline`（而不是 `StackedInline`），相关的对象将会用更紧凑的，表格式的格式显示：

Home > Polls > Questions

Select question to change

Action:   0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

多出来的“删除？（Delete？）”列可以用来删除通过“添加选项（Add Another Choices）”按钮

添加的数据，尽管它们已经被储存了。

## 自定义对象列表

现在 Django 的对象修改页面已经很棒了，现在我们来调整一下对象列表页 - 也就是显示所有问题 (Question) 对象的页面。

目前它是这样的：

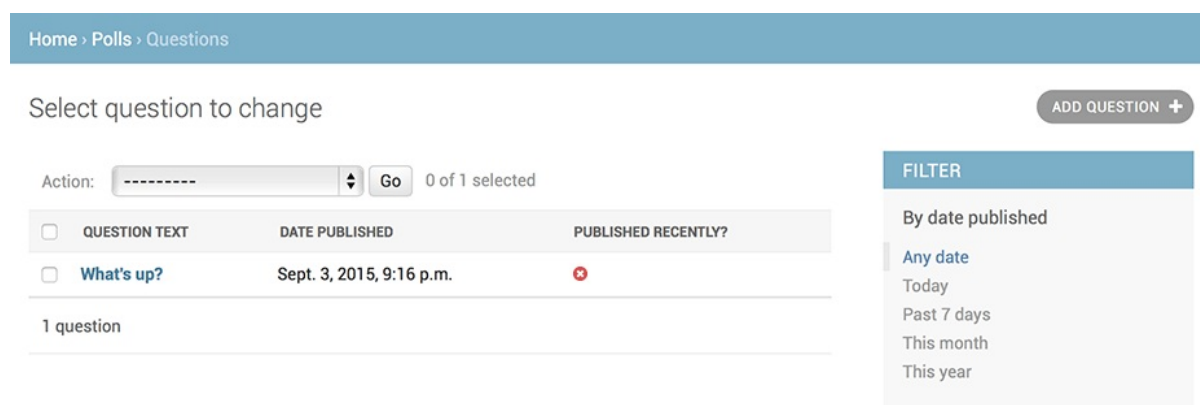
默认情况下，Django 使用 `str()` 方法来显示对象。但有时如果我们显示一些其他的字段会很有用。为此，我们可以使用 `list_display` 选项，它是由需要被显示的字段名组成的元组，这些字段将会作为额外的列显示在列表中。

```
1. # polls/admin.py
2.
3. class QuestionAdmin(admin.ModelAdmin):
4.     # ...
5.     list_display = ('question_text', 'pub_date')
```

为了方便检索，我们把在 [教程第二部分 \(zh\)](#) 中自定义的 `was_published_recently` 方法也加入进来：

```
1. # polls/admin.py
2.
3. class QuestionAdmin(admin.ModelAdmin):
4.     # ...
5.     list_display = ('question_text', 'pub_date', 'was_published_recently')
```

现在，问题 (Question) 的对象列表如下图：



通过点击表头，你可以让列表按所选的属性进行排序 - 除了 `was_published_recently` 函数，因为 Django 不支持通过随便定义的方法的输出结果进行排序。`was_published_recently` 方法所在列的列名默认使用了方法名（下划线被替换为空格），显示的值是对应方法返回值的字符串形式。

你能够通过给方法（在 `polls/models.py` 里）增加属性来扩展功能，看下面的示例：

```
1. # polls/models.py
2.
3. class Question(models.Model):
4.     # ...
5.     def was_published_recently(self):
6.         now = timezone.now()
7.         return now - datetime.timedelta(days=1) <= self.pub_date <= now
8.     was_published_recently.admin_order_field = 'pub_date'
9.     was_published_recently.boolean = True
10.    was_published_recently.short_description = 'Published recently?'
```

想了解更多的方法选项，请看文档：[list\\_display](#)。

再次编辑 `polls/admin.py` 文件。我们要为问题（Question）的对象列表增加一个新功能：通过 `list_filter` 过滤对象。向 `QuestionAdmin` 中添加以下内容：

```
1. list_filter = ['pub_date']
```

上面代码的作用是为对象列表添加了一个“快速过滤”侧边栏，可以通过 `pub_date` 字段来挑选问题（Question）。

过滤的类型取决于你要通过哪种字段进行过滤。因为 `pub_date` 是个 [日期时间字段（DateTimeField）](#)，所以 Django 会自动提供合适的过滤方法：“任何时候（Any date）”、“今天（Today）”、“过去7天（Past 7 days）”、“本月（This month）”、“今年（This year）”。

这个功能搞定了，让我们加一些搜索功能：

```
1. search_fields = ['question_text']
```

这将会在对象列表的顶部加一个搜索框。当有人键入内容时，Django 会搜索 `question_text` 字段。你想搜索多少字段都行 - 尽管搜索过程使用的是数据库查询语句里的 **LIKE** 关键字，但限制一下搜索结果数量将会让数据库搜索更加轻松。

默认情况下，Django 每100个对象为一页。[对象列表分页 \(Change list pagination\)](#)、[搜索框 \(search boxes\)](#)、[过滤器 \(filters\)](#)、根据 [时间分组 \(date-hierarchies\)](#) 和根据 [表头排序 \(column-header-ordering\)](#) 可以完美结合，一起工作。

## 自定义管理页面样式

显然，左上角有个“Django 管理系统 (Django administration)”看起来非常滑稽，它其实只是个占位文本而已。

你可以轻易地修改它 - 通过使用 Django 模板系统。Django 管理页面是 Django 自身提供的，它的界面使用的是 Django 的模板系统。

## 自定义项目模板

在你的项目文件夹 (`manage.py` 所在的文件夹) 里创建一个 `templates` 目录。模板 (Template) 可以放在 Django 拥有访问权的任何地方。(Django 和执行启动服务器操作的用户拥有相同的权限。) 但是，把模板放在项目目录是个惯例，遵守它有好处。

打开设置文件 (记住，是 `mysite/settings.py`)，向 **TEMPLATES** 设置中添加 **DIRS** 选项：

```
1. # mysite/settings.py
2.
3. TEMPLATES = [
4.     {
5.         'BACKEND': 'django.template.backends.django.DjangoTemplates',
6.         'DIRS': [os.path.join(BASE_DIR, 'templates')],
7.         'APP_DIRS': True,
8.         'OPTIONS': {
9.             'context_processors': [
10.                'django.template.context_processors.debug',
11.                'django.template.context_processors.request',
12.                'django.contrib.auth.context_processors.auth',
13.                'django.contrib.messages.context_processors.messages',
14.            ],
15.        },
16.    ],
```

17. ]

**DIRS** 是在载入 Django 模板时会搜索的文件系统上的目录集合。

组织下模版们

就像静态文件那样，我们可以把所有模版放在一个大的模版目录里，那会工作得很好。不过，如果模版属于某个特别的应用，那它们应该放在那个应用的模版目录下（比如 `polls/templates`）而不是项目的（`templates`）。我们将在 [可重用的应用](#) 进行详细的讨论。

现在在 `templates` 里创建 `admin` 目录，然后从 Django 代码目录里的默认 Django 管理页面模板目录（`django/contrib/admin/templates`）中将 `admin/base_site.html` 模板复制到新建的目录里。

Django 代码目录在哪

如果你不知道 Django 代码目录在哪，尝试以下命令：

```
1. $ python -c "import django; print(django.__path__)"
```

然后，只要把 `{{ siteheader|default:('Django administration') }}`（包括大括号）替换为你想要的字符就行了。你应该把它编辑成大概这个样子：

```
1. {% block branding %}
2. <h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
3. {% endblock %}
```

我们只是用这个例子来教你如何覆盖模板。在实际项目里，使用 `django.contrib.admin.AdminSite.site_header` 属性来设置自定义标题会更简单。

模板文件里含有很多像 `{% block branding %}` 和 `{{ title }}` 这样的文本。`{%}` 和 `{{` 标签是 Django 模板语言的一部分。当 Django 渲染 `admin/base_site.html` 时，模板语言将会被执行，执行的结果是产生一个 HTML 文档。就像 [教程第三部分 \(zh\)](#) 看到的那样。

Django 管理界面的所有默认模板都能够被覆盖。如果你想覆盖它们的话，就像你覆盖 `base_site.html` 一样，把原模板复制到你的目录中，然后修改它即可。

## 自定义应用模板

敏锐的读者将会思考：如果 **DIRS** 目录维持默认的空值的话，Django 怎么知道去哪找默认的模板呢？答案是，因为 **APP\_DIRS** 被设置为 **True**，Django 会自动搜索每个应用目录下的 `templates/` 子目录，将它们作为最后的备用选择。（别忘了，`django.contrib.admin` 也是个应用）

我们的投票应用并不太复杂，不需要自定义管理页面。但是如果它慢慢变成一个复杂的应用，为了增

加某些功能需要修改 Django 的管理页面标准模板，这时候，比起项目模板，修改 应用 模板会是更明智的选择。通过这种方式，你可以在新项目中使用投票应用并且保证它还能找到所需要的自定义模板。

查看 [加载模版](#) 文档来获取有关 Django 如何寻找模板的细节信息。

## 自定义管理页面索引页

---

类似的。你有可能想自定义 Django 管理页面索引页的样式。

默认情况下，索引页会按字母顺序显示出所有在 `INSTALLED_APPS` 里并且已经向 admin 注册过的应用。你可能想对这个界面做一些大的修改。但是，要记住，索引页可能是众多管理页面里最重要的一个，它需要是简单且易用的，

需要自定义的是 `admin/index.html`。（和上面自定义 `admin/base_site.html` 的步骤类似 - 从默认模板目录复制一份到自定义模板目录）打开这个文件，你会看见它使用了一个叫做 `app_list` 的变量。这个变量包含所有被激活的 Django 应用。除了使用这个变量，你还可以使用你认为合适的任何方法，把特定对象的管理页面地址硬编在模板里。

## 下一步？

---

本新手教程到这里就结束了。与此同时，你可能还想得到一些指导，可以看[下一步](#)。

如果你熟悉 Python 的打包机制，并且对如何将投票应用转化为一个“可重用的应用”感兴趣，请看 [高级教程：如何编写可重用的应用](#)。



## 进阶内容：编写可重用的应用

- [进阶内容：如何编写可重用的应用](#)
  - [可重用性问题](#)
  - [你的项目和你可重用的应用](#)
  - [安装一些必备工具](#)
  - [打包你的应用](#)
  - [使用你的包](#)
  - [发布你的应用](#)
  - [用 `virtualenv` 安装 Python 包](#)

## 进阶内容：如何编写可重用的应用

本篇从 [教程第七部分 \(zh\)](#) 结束的地方开始。我们将把之前写的调查应用转化为一个可在新项目中重复使用，和能分享给其他人的独立 Python 包。

如果您还没有完成教程 1-7，我们鼓励你把它们完成一遍，以便使你的项目和下面的教程相匹配。

## 可重用性问题

设计，构建，测试和维护 Web 应用程序需要大量的工作。许多 Python 和 Django 的项目都会面临这个问题。如果我们能节省一些重复工作的话，那一定会很棒。

可重用性深入 Python 的设计理念中。[Python 包索引 \(PyPI\)](#) 里有大量可以在你自己的程序中使用的包。查看[用于 Django 的包](#)这个分类，里面包含着现有的可重用应用程序，你可以在自己的项目里使用它们。Django 本身也只是一个 Python 包。这意味着，你可以充分利用现有的 Python 包或者 Django 应用程序来帮助完成自己的 Web 项目。你只需要编写一些特殊的部分即可。

假设，你的新项目需要一个类似我们之前写过的那样的投票应用。如何使这个应用可重用呢？幸运的是，你已经在正确的路上了。在 [教程第三部分 \(zh\)](#) 中，我们了解了如何从通过在项目级别的 URL 配置文件中 **include** 来使投票应用和主项目分离。在本教程中，我们将采取进一步的措施，使应用能很容易在新项目中使用，并发布给别人安装和使用。

包？应用？

Python 包 (`package`) 提供了将 Python 代码按相关性分组的组织方式，这一方式提高了可重用性。包中包含一个或多个 Python 代码文件（也被称为“模块”）。

包可以通过 `import foo.bar` 或 `from foo import bar` 的方式被导入。对于由一个目录（如 `polls`）组织起来的包，目录里必须有一个特殊的文件 `__init__.py`，即使这个文件是空的。

Django 应用也只是一个 Python 包，不过是专门用于 Django 项目的。应用一般遵循常见的 Django 的公约，例如会包含 `models`（模型），`tests`（测试），`urls`（网址）和 `views`（视图）等子模块。

之后我们使用术语 `打包` 来描述“制作一个能被其他人使用的 *Python* 包”的过程。这可能会有点令人迷糊。（译注：原文中这句话的意思是后文将会把 *package* 同时作为动词和名词，可能造成阅读困难，但翻译成中文后并不会，因为我们有打包这个词 ^\_^。）

## 你的项目和你可重用的应用

在跟着做完前面的教程后，我们的项目应该是这样的：

```
1. mysite/
2.     manage.py
3.     mysite/
4.         __init__.py
5.         settings.py
6.         urls.py
7.         wsgi.py
8.     polls/
9.         __init__.py
10.        admin.py
11.        migrations/
12.            __init__.py
13.            0001_initial.py
14.        models.py
15.        static/
16.            polls/
17.                images/
18.                    background.gif
19.                style.css
20.        templates/
21.            polls/
22.                detail.html
23.                index.html
24.                results.html
25.        tests.py
26.        urls.py
27.        views.py
28.    templates/
29.        admin/
30.            base_site.html
```

你在 [教程第七部分 \(zh\)](#) 中创建了 `mysite/templates`，在 [教程第三部分 \(zh\)](#) 中创建了 `polls/templates`。现在你也许能更清楚的了解为什么我们选择了将项目和应用的模板目录分开：所有投票应用所用到的东西都在应用里。这使应用完全包含自身，更容易被新的项目所使用。

`polls` 目录现在可以被复制到一个新的 Django 项目，并可以立即被重用。但是对发布来说，还有一些地方需要准备。我们需要打包的应用程序，以方便他人进行安装。

## 安装一些必备工具

Python 打包的解决方案目前有点混乱，因为有各种不同的工具。在本教程中，我们将使用 `setuptools` 建立我们的包。这是推荐的打包工具（与 `distribute` 分支合并后）。可是使用 `pip` 来安装和卸载它。你现在应该安装这两个软件包。如果需要帮助，你可以参考[如何使用 pip 安装 Django](#)。您可以用相同的方式安装 `setuptools`。

## 打包你的应用

Python 的 打包 是指将你的应用制作成特定的格式，以便能被很方便地安装和使用。Django 本身就是按照这个规定打包的。对于一个小应用，比如我们的投票应用，这个过程不会太困难。

1. 首先，在你的 Django 项目外为 `polls` 目录创建一个父目录，命名为 `django-polls`。

选择你应用的名字

当为你的包选择名称时，记得检查 *PyPI* 上的内容以避免与现有的包产生冲突。以 `django-` 作为模块名称前缀是很实用的，这将有助于想寻找 Django 应用的人来识别哪些包是用于 Django 的。

应用标签（即，以点分隔的模块路径的最后一部分）必须在 `INSTALLED_APPS` 里是独一无二的。避免和 Django *contrib package* 使用相同的标签，例如 `auth`，`admin`，或 `message`。

2. 移动 `polls` 目录到 `django-polls` 目录下

3. 创建文件 `django-polls/README.rst`，写入下面的内容。

```

1. # django-polls/README.rst
2.
3. =====
4. Polls
5. =====
6.
7. Polls is a simple Django app to conduct Web-based polls. For each
8. question, visitors can choose between a fixed number of answers.
9.
10. Detailed documentation is in the "docs" directory.
11.
12. Quick start
13. -----
14.
15. 1. Add "polls" to your INSTALLED_APPS setting like this::
16.
17.     INSTALLED_APPS = [
18.         ...
19.         'polls',
20.     ]

```

```

21.
22. 2. Include the polls URLconf in your project urls.py like this::
23.
24.     url(r'^polls/', include('polls.urls')),
25.
26. 3. Run `python manage.py migrate` to create the polls models.
27.
28. 4. Start the development server and visit http://127.0.0.1:8000/admin/
29.     to create a poll (you'll need the Admin app enabled).
30.
31. 5. Visit http://127.0.0.1:8000/polls/ to participate in the poll.

```

4. 创建 **django-polls/LICENSE** 文件。选择许可证超出了本指南的范围，但可以说，没有许可证的公开发布的代码是 没有用的。Django 和许多 Django 兼容的应用程序是在 BSD 许可下发布的；然而，你可以自由选择你自己的许可证。只是要注意你的许可选择会影响到谁能使用你的代码。

5. 接下来我们将创建一个 **setup.py** 文件，这个文件提供了有关如何创建和安装应用的细节。对此文件的完整的解释已经超出了本文的范围，但 [setuptools docs](#) 文档对它有很棒的解释。创建包含以下内容的 **django-polls/setup.py** 文件：

```

1. # django-polls/setup.py
2.
3. import os
4. from setuptools import find_packages, setup
5.
6. with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme:
7.     README = readme.read()
8.
9. # 使 setup.py 能在任何地方运行
10. os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__), os.pardir)))
11.
12. setup(
13.     name='django-polls',
14.     version='0.1',
15.     packages=find_packages(),
16.     include_package_data=True,
17.     license='BSD License', # example license
18.     description='A simple Django app to conduct Web-based polls.',
19.     long_description=README,
20.     url='https://www.example.com/',
21.     author='Your Name',
22.     author_email='yourname@example.com',
23.     classifiers=[
24.         'Environment :: Web Environment',
25.         'Framework :: Django',
26.         'Framework :: Django :: X.Y', # replace "X.Y" as appropriate
27.         'Intended Audience :: Developers',

```

```

28.         'License :: OSI Approved :: BSD License', # example license
29.         'Operating System :: OS Independent',
30.         'Programming Language :: Python',
31.         # Replace these appropriately if you are stuck on Python 2.
32.         'Programming Language :: Python :: 3',
33.         'Programming Language :: Python :: 3.4',
34.         'Programming Language :: Python :: 3.5',
35.         'Topic :: Internet :: WWW/HTTP',
36.         'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
37.     ],
38. )

```

6. 生成的打包文件中默认只包含 Python 模块和包。如果想包括额外的文件，我们需要创建一个 **MANIFEST.in** 文件。上一步骤中提到的 **setuptools** 文档里有更多的细节介绍。为了包含模板、我们的 **README.rst** 和 **LICENSE** 文件，请创建包含以下内容的 **django-polls/MANIFEST.in** 文件：

```

1. # django-polls/MANIFEST.in
2.
3. include LICENSE
4. include README.rst
5. recursive-include polls/static *
6. recursive-include polls/templates *

```

7. 这是可选步骤，但是我们建议完成这个步骤。将应用的详细的文档一起打包进去。为以后的文档创建一个空目录 **django-polls/docs**，在 **django-polls/MANIFEST.in** 里添加一行：

```

1. recursive-include docs *

```

注意，如果 **docs** 目录里没有文件，那么它是不会被打包的。很多 Django 应用还通过托管网站，如 [readthedocs.org](http://readthedocs.org)，提供在线文档。

8. 尝试通过 **python setup.py sdist** 命令进行打包（在 **django/polls** 目录里运行）。这将创建一个名为 **dist** 的目录，并在其中创建打包文件 **django-polls-0.1.tar.gz**。

关于打包的更多信息，参见 Python 教程：[Tutorial on Packaging and Distributing Projects](#)。

## 使用你的包

自从我们把 **polls** 目录从项目中移走了，它就不再工作了。我们现在通过安装新的 **django/polls** 包解决这个问题。

作为用户库安装

以下步骤将把 *django/polls* 安装为用户库。和全局安装相比，单用户安装有很多好处，比如可以在没有管理员权限的系统中使用，以及防止安装的包影响系统服务或机器上的其他用户。

请注意，单用户安装仍然可能影响到其他用户的系统的工具，所以 *virtualenv* 是一个更强大的解决方案（见后文）。

1. 要安装这个包，得用到 `pip`（你已经安装了吧，对吗？）

```
1. pip install --user django-polls/dist/django-polls-0.1.tar.gz
```

2. 如果幸运的话，你的 Django 项目现在应该可以正常工作了。再次运行服务器以确认。

3. 想卸载这个包，用 `pip`。

```
1. pip uninstall django-polls
```

## 发布你的应用

现在，我们已经打包并测试了 *django-polls*，并准备与所有人共享它！如果这不是一个示例的话，你现在可以：

- 将这个包通过电子邮件发给朋友。
- 将包上传到你自己的网站上。
- 在公共库，如 [Python 包索引 \(PyPI\)](#)，上发布你的包。[packaging.python.org](http://packaging.python.org) 上有很棒的教程。

## 用 virtualenv 安装 Python 包

之前，我们将投票应用程序安装为用户库。这有一些缺点：

- 修改用户库可能影响你系统中的其他 Python 软件。
- 你将无法同时使用这个包的多个版本（或相同名字的其他包）。

通常，这些情况仅出现在你同时管理多个 Django 项目时。当出现这种情况时，最好的办法是使用 *virtualenv*。这个工具允许你维护多个分离的 Python 环境，每一个复制了一份自己专用的库和包命名空间（package namespace）。

## 接下来如何学习？

- [接下来如何学习？](#)
  - [如何查询文档](#)
  - [文档是如何组织的](#)
  - [文档是如何更新的](#)
  - [哪里可以获取文档](#)
    - [在线获取](#)
    - [纯文本](#)
    - [下载为本地HTML文件](#)
  - [版本之间的差异](#)

## 接下来如何学习？

看来您已经阅读完所有的[介绍文档](#)，并且决定继续使用 Django。但是我们前面只是进行了概括性的介绍（事实上，即使您全部浏览完，也只是涉及了所有文档 5% 左右的内容）。

所以接下来是？

没错，我们一直专注于通过实践来提升自身能力。基于这一点考虑，您应该已经掌握足够的知识，可以直接开始自己的个人项目。当您需要相关帮助时，届时再查询文档。

我们倾注了大量精力来使 Django 官方文档更加实用，易于阅读和尽可能地完备。本文其他部分更多的是关于 Django 官方文档的使用方法，这样您可以最有效地利用它。

是的，这篇文档是关于如何使用 Django 官方文档的。不过请放心，我们不会再撰写一份新文档来介绍如何使用这篇文档：)

## 如何查询文档

Django 拥有着大量的文档 – 大概在 450,000 单词左右 – 因此有时候查询您所需要的内容并非易事。一些查看文档的好地方是[搜索页面](#)和[索引页](#)。

或者您可以完整浏览！

## 文档是如何组织的

Django 官方文档的主体内容可以根据满足不同的需要分解成不同的部分：

- [介绍文档](#)通常是刚接触 Django 或者 Web 开发的人所设计的。它并不涉及太多高深的内容，

但相对地提供了一个宏观的概览视角帮助培养如何使用 Django 进行开发的“感觉”。

- 另一方面，[主题指南](#)对 Django 的每一块进行了深入讲解。主题指南包括了对于 Django 的[模型系统](#)，[模板引擎](#)，[表单框架](#)以及更多内容的完整指导。

这里可能是您花费时间最多的地方；如果您动手实践了这些指导文档包含的内容，那么您应该对 Django 非常熟悉了。

- Web 开发涉猎范围广，但是并不深入——遇到的问题可能跨越了不同的领域，为此我们撰写了一系列 [how-to guides](#)文档来回答常见诸如 “How do I..?”（我该如何）这类的问题。在这部分的文档里，您可以找到关于[使用 Django 生成 PDF](#)，[如何编写自定义的模板标签](#)等等的内容。

对于常见问题的答案可以在 [FAQ](#) 找到。

- 指导文档和 How-To 文档并没有完全覆盖到 Django 中的每一个类、函数、方法，那样的话内容会太多，不利于学习。实际上，每个类、函数、方法还有模块的实现细节都记录在[参考部分](#)中。那里才是当你需要查找函数细节或是其他具体细节的地方。
- 如果您对于部署 Django 项目到公共网络感兴趣的话，我们文档也提供了一些关于各种部署设置的[指导](#)，包括您所需要关注的[部署清单](#)。
- 最后，有一些“特殊”的文档通常与大多数开发者无关，比如[发行记录](#)以及针对于那些想为 Django 项目贡献力量的开发人员的[内部文档](#)，此外还包括了一些[不好分类的杂散文档](#)。

## 文档是如何更新的

---

正如 Django 代码每天都在开发和改进，我们的文档也在不断完善中。我们改进文档的理由如下：

- 修正文档内容，例如语法/排版错误。
- 为有需要扩展的章节增加内容和可能的示例。
- 记录之前尚未归档的 Django 特性。（这样的功能列表正在萎缩，但仍然存在）。
- 当 Django 增加新特性，以及 Django API 或者其行为发生改变时，会补充相关内容到文档中。

Django 文档和其代码使用同一份版本控制进行管理。它位于我们 Git 仓库的 [docs](#) 目录下。在 Git 仓库中，每一份在线文档都是独立的文本文件。

## 哪里可以获取文档

---

您可以通过多种方式阅读 Django 文档，根据优先级顺序排列如下：

### 在线获取



最新版本的 Django 官方文档位于 <https://docs.djangoproject.com/en/dev/>。这些HTML页面是通过源代码控制的文本文件自动生成的。这就意味着它们涉及了 Django “最新和最好”的方面——包括了最新的修正和新添加的内容，以及对于只面向使用 Django 最新版本的用户开放的新特性讨论。（参考下文“版本之间的差异”）

我们鼓励您通过[反馈系统](#)提交更新，修正和建议以促进文档质量的改善。Django的开发人员会积极地关注反馈系统，并根据您的反馈来改善文档。

请注意，无论如何，提交的反馈应该和文档密切相关，而不是涉及技术支持的问题。如果您需要额外的技术支持，请试试 [django user](#) 邮件列表或者 [#django IRC channel](#)。

## 纯文本

如果是想离线阅读或者仅仅只是为了方便，您可以直接阅读 Django 的纯文本文档。

如果您正在使用 Django 的官方发行版本，请注意对应发行版的说明文档都包含在了代码压缩包（tarball）的 `docs/` 的目录下。

如果您正在使用开发版本的 Django（又名“trunk 版本”），请注意 `docs/` 目录包含了所有的文档。您可以通过 `git checkout` 来获取最新更新。

一个稍微有点技术含量的查看文档的方法是通过 Unix 系统的 `grep` 命令来查找关键字搜索文档。例如，这将会展示 Django 文档中每一处提及“`max_length`”的地方：

```
1. $ grep -r max_length /path/to/django/docs/
```

## 下载为本地HTML文件

您也可以通过以下简单的步骤获取到官方文档的本地HTML文件：

- Django 的文档使用了 [Sphinx](#) 文档系统来进行从纯文本到 HTML 的转换。您需要通过 Sphinx 官网下载安装包，或者通过 `pip` 方式进行安装：

```
1. $ pip install Sphinx
```

- 然后使用文档目录下的 `Makefile` 进行纯文本到 HTML 格式的转换：

```
1. cd path/to/django/docs
2. $ make html
```

进行此操作，您需要安装 [GUN Make](#)。

如果您是在 Windows 系统的环境下，也可以选择使用目录中的批处理文件：

```
1. cd path\to\django\docs
2. make.bat html
```

- 最后生成的 HTML 格式的文档会位于 `docs/_build/html` 中。

## 版本之间的差异

正如前面提及的那样，我们 Git 仓库中的文档包含了许多“最新最好”的改变。这些改变通常包括了 Django 开发版本新增的一些特性。因此出于以上理由，有必要说明我们在处理不同 Django 版本文档时所遵循的准则。

我们遵循着如下准则：

- 在 *django project.com* 呈现的内容始终是 Git 仓库最新文档的 HTML 版本。这些文档总是对应 Django 官方最新版本，再加上在最新框架发布以来的特性的更改或者添加的内容。
- 当 Django 的开发版本中添加新特性的时候，我们会尽可能在同一份 git 提交中同步更新文档。
- 为了在文档中区分特性的更改或者添加，我们使用了如下说明文字：“ `New in version X.Y` ”，意思是将出现在接下来的发行版本中（因此，还处在开发的阶段中）。
- 在某些情况下，文档的修复和改进可能会在开发人员的同意下回迁到最后的发行分支。然而，一旦某个版本的 Django **不再被支持**，那么对应版本的文档也会停止进一步的更新。
- [文档主页面](#) 包含着所有旧版本的文档。请务必确保您阅读的文档版本对应着您正在使用的 Django ！

# 编写你的第一个 Django 补丁

- 为 Django 编写你的第一个补丁
  - 介绍
  - 本教程的目标对象？
  - 本教程覆盖了哪些知识点
  - 行为准则 (Code of Conduct)
  - 安装 Git
  - 下载 Django 开发版的拷贝
  - 回滚到先前的 Django 版本
  - 第一次运行 Django 的测试套件
  - 为你的补丁创建一个分支
  - 为你的任务编写一些测试
  - 为任务 #24788 编写一些测试
  - 运行你写的新测试
  - 为你的任务编写代码
    - 为任务 #24788 编写代码
    - 现在验证并通过你的测试
  - 第二次运行 Django 的测试套件
  - 写文档
  - 预览你的修改
  - 提交补丁中的修改
  - 推送提交和拉取请求 (pull request)
  - 下一步
    - 更多关于新贡献者的信息
    - 寻找你真正的第一次任务
    - 在创建完拉取请求后还要干什么呢？

## 为 Django 编写你的第一个补丁

### 介绍

对向社区回馈有点兴趣？也许你发现了 Django 的一个 bug，然后你想修复它，或者你想为 Django 添加一些功能。

回馈 Django 本身就是看到自己所关心的问题的最佳方式。也许初看会吓到你，但这实际上是很简单的。我们将带你走一遍整个流程，好让你可以从例子中学习。

### 本教程的目标对象？

参见

如果你正在为如何提交补丁，寻找参考指南，可以看[提交补丁](#)文档。

在本教程里，我们期望你至少能基本明白 Django 是如何工作的。这意味着你应该顺利地读完了[实例教程](#)。还有，你应该对 Python 本身也有了很好地了解。如果没有，可以在线看看这本有趣的[Dive Into Python](#)，它面向 Python 新手程序员。

不熟悉版本控制系统和 Trac 的人会发现本教程及其链接的信息刚刚好够开始。但是，如果你打算定期为 Django 做贡献，你可能会想了解更多关于这些不同工具的知识。

在大多数情况下，本教程会尽可能多地进行说明，以确保受众能接受。

哪里可以获得帮助：

如果你在阅读或实践本教程中遇到困难，请发消息给 [django-users](#) 或加入IRC频道 [django on irc.freenode.net](#) 来与其他 Django 用户进行交流，他们也许能帮到你。

## 本教程覆盖了哪些知识点

首先我们会带你走完为 Django 贡献补丁的流程。在本教程结束时，你应该对工具和流程都有了基本认识。特别地，我们将会介绍以下内容：

- 安装 Git
- 如何下载 Django 开发版的拷贝
- 运行 Django 的测试套件
- 为你的补丁编写测试程序
- 为你的补丁编写代码
- 测试你的补丁
- 提交拉去请求 (pull request)
- 哪里可以查看更多信息

一旦你完成了本教程，你就可以去看完为 [Django 做贡献文档](#)的剩下部分了。那里有很多信息，而且是那些想成为定期贡献者必看的。如果你有问题，你也可能在那得到答案。

需要 **Python 3**！

本教程假定你用的就是 Python 3。请到[Python 官网下载页](#)或者你系统的包管理安装最新版的 Python3。

给 Windows 用户

给 Windows 安装 Python 时，请确保把 `python.exe` 添加进了 `Path`，这样可以直接在命令行使用。

## 行为准则 (Code of Conduct)

作为一名贡献者，你可以帮助我们保持 Django 社区的开放和包容。请阅读和遵循我们的[行为准则 \(Code of Conduct\)](#)

## 安装 Git

本教程里，为了下载到最新的开发版 Django 和生成你修改过的补丁文件，你将需要安装 Git。

为了确认是否安装了 Git，你可以在命令行输入 `git`。如果提示说命令找不到，那你就需要下载安装了，[Git 下载页](#)。

给 Windows 用户

给 Windows 安装 Git 时，请确保给 “Git Bash” 打勾了，这样 Git 可以用它自己的 shell 运行。本教程假定你已经安装了它。

如果你对 Git 不太熟悉，你可以在命令行输入 `git help` 获取更多命令信息。

## 下载 Django 开发版的拷贝

第一步就是得到 Django 源码的拷贝。首先在[GitHub 里 fork Django项目](#)。然后在命令行里，使用 `cd` 切换到你想放 Django 本地拷贝的目录里。

使用下面的命令下载 Django 源码仓库：

```
1. $ git clone git@github.com:YourGitHubName/django.git
```

现在你有了一份 Django 的本地拷贝，你可以安装它，就像使用 `pip` 安装其他包那样。最方便的方式是使用 虚拟环境 (*virtual environment*) (或者 `virtualenv`)，这是 Python 内置的功能，允许您为每个项目单独设立已安装软件包的目录，让它们不会相互干扰。

最好是把你所有的虚拟环境 (`virtualenvs`) 都放在一个目录下，比如在你的 `home` 目录下的 `.virtualenvs/` 目录。如果还没创建：

```
1. $ mkdir ~/.virtualenvs
```

现在，运行以下命令创建新的虚拟环境 (`virtualenvs`)：

```
1. $ python3 -m venv ~/.virtualenvs/djangodev
```

路径就是新的虚拟环境，而它会被保存在你的电脑里。

给 Windows 用户

如果你在 *Windows* 下使用 *Git* 的 *shell*，使用内置的 *venv* 模块会无效，由于启动脚本是为系统 *shell* 创建的 (*.bat*) 和 *PowerShell* (*.ps1*)。使用 *virtualenv* 包代替 *venv* 模块：

```
1. $ pip install virtualenv
2. $ virtualenv ~/.virtualenvs/djangodev
```

给 *Ubuntu* 用户

在一些 *Ubuntu* 版本中，上面的命令可能会失败。

使用 *virtualenv* 包代替，确认你安装了 *pip3*：

```
1. $ sudo apt-get install python3-pip
2. # 如果下面的命令因为没有权限出错了，那就加上 sudo
3. $ pip3 install virtualenv
4. $ virtualenv --python=`which python3` ~/.virtualenvs/djangodev
```

最后一步就是让你的虚拟环境 (*virtualenvs*) 生效：

```
1. $ source ~/.virtualenvs/djangodev/bin/activate
```

如果 **source** 命令无效，你可以试试用一个“点”代替：

```
1. $ . ~/.virtualenvs/djangodev/bin/activate
```

给 *Windows* 用户

为了在 *Windows* 上让虚拟环境 (*virtualenvs*) 生效，运行下面命令：

```
1. $ source ~/virtualenvs/djangodev/Scripts/activate
```

无论什么时候，当你打开一个新的终端窗口时都要激活一下虚拟环境 (*virtualenvs*)。为了方便这种情况，*virtualenvwrapper* 是很有用的工具。

从现在开始，你用 **pip** 安装的任何东西都会被安装进你新的的虚拟环境 (*virtualenvs*)，这会隔绝其他环境和系统包。同样，当前激活的虚拟环境 (*virtualenvs*) 的名字会在命令行前面显示，这可以让你知道使用的是哪个虚拟环境。

继续，安装之前克隆 (*clone*) 下来的 *Django* 拷贝：

```
1. $ pip install -e /path/to/your/local/clone/django/
```

安装的 *Django* 现在指向了你的本地拷贝。你将马上看到你做的任何修改，这对你编写第一个补丁很有帮助。

## 回滚到先前的 Django 版本

本教程里，我们将会使用 [#24788](#) 任务 (ticket) 作为例子学习，所以在任务 (ticket) 补丁应用之前，我们将回退到 Django 的历史版本。这让我们可以走完所有步骤，包括从头开始写编写补丁，还有运行 Django 的测试套件。

请记住，为了下面的教程，我们将使用 Django 的主干旧版本，当你编写自己的补丁时，你应该始终使用 Django 当前的开发版本

### 注意

这个任务 (ticket) 的补丁已经由 *Paweł Marczewski* 编写了，也已经被应用到了 Django 中，[commit 4df7e8483b2679fc1cba3410f08960bac6f51115](#)。因此，我们将使用的是 Django 先前的版本，[commit 4ccfc4439a7add24f8db4ef3960d02ef8ae09887](#)。

切换到 Django 的根目录 (那里有 **django**, **docs**, **tests**, **AUTHORS** 等)，你可以检查下这个旧版本 Django：

```
1. $ git checkout 4ccfc4439a7add24f8db4ef3960d02ef8ae09887
```

## 第一次运行 Django 的测试套件

这非常重要：当向 Django 贡献时，你修改的代码不会将 bug 引到 Django 的其他地方。一个检查 Django 还可以工作的方法就是在你修改之后运行 Django 的测试套件。如果所有的测试都通过了，你就有理由确认你的修改不会完全破坏 Django。如果你之前没有运行过测试套件，最好在这之前运行熟悉下它的输出大概是什么样的。

在运行之前，**cd** 切换到 Django 的 **tests/** 目录，安装它的依赖关系：

```
1. $ pip install -r requirements/py3.txt
```

如果你在安装过程中遇到错误，你的系统可能有一些 Python 包的依赖关系缺失。查询一下安装失败的包的文档，或者在网上搜索一下你遇到的这个错误信息。

现在我们准备好运行测试套件了。如果你用的是 GNU/Linux, macOS 或者其他 Unix 风格的系统，运行：

```
1. $ ./runtests.py
```

现在坐下来歇会。Django 整个测试套件有超过 9,600 个不同的测试，所以需要 5 到 15 分钟的时间运行，这取决于你电脑的运算速度。

当 Django 测试套件运行的时候，你可以看到一段显示着每个正在运行的测试的状态的字符流。E

表示在测试期间抛出了一个错误，**F** 表示一个测试的断言 (assertions) 失败。这两者都可以被认为是测试失败了。同时，**x** 和 **s** 分别表示预期失败和跳过测试。“点”表示通过测试。

跳过测试是由于运行测试时缺少额外的库；可以查看[运行所有测试](#)所需要的一系列的依赖，确认和你修改的相关的一些测试依赖被安装了（本教程我们什么都不需要安装）。一些测试是特别针对特别的后台数据库的，如果不对那后台进行测试，就会被跳过。SQLite 是默认设置的后台数据库。如果是使用其他的后台数据库，为了运行测试，可以查看[使用其他设置模块](#)。

一旦测试完毕，你应该会收到一条关于测试套件是通过了还是失败了的信息。由于你还没有对 Django 的代码做修改，所以整个测试套件 应该是 (**should**) 通过。如果你得到的是失败或者错误的信息，请确保你之前走的所有步骤都正确。可以查看[运行单元测试](#)获得更多信息。如果你正在使用 Python3.5+，将会有些与不被赞成的警告 (deprecation warnings) 相关的失败，那些你可以忽略掉。这些失败已经被 Django 修复了。

注意，最新的 Django 主干 (trunk) 版本不总是稳定的。当开发版本遇上主干 (trunk) 版本，你可以检查 [Django 的持续集成构建 \(Django's continuous integration builds\)](#) 来确认是特别针对你的机器的失败还是已经存在在 Django 官方构建里的。如果你点击去看一个特别的构建，你可以看到 “配置矩阵 (Configuration Matrix)” 显示着由于 Python 版本和数据库后台问题出现的失败。

#### 注意

在本教程和我们正在工作的任务 (ticket) 里，测试使用 SQLite 足够了，但是，[运行其他数据库进行测试](#)是可能的（有时是必要的）。

## 为你的补丁创建一个分支

在修改之前，先为任务 (ticket) 创建一个新的分支：

```
1. $ git checkout -b ticket_24788
```

你可以为这个分支选择任何你想要的名字，例如 “ticket\_24788”。在这个分支做的所有修改都是特别针对这个任务的，并不会影响到我们先前克隆下来的主要代码。

## 为你的任务编写一些测试

在大多数情况下，被 Django 接收进来的补丁都必须测试过。针对 bug 修复的补丁，这意味着要编写回退测试代码来确保 bug 不会在 Django 的后续版本再次出现。一个回退测试应该是这样的：当 bug 存在时会失败，当 bug 被修复时会通过。对于包含新特性的补丁，你的测试需要能保证新特性是能正常工作的。当新特性不存在是，它们也是要失败的；一旦被应用又是能通过的。

一个好方法是：首先在修改代码前，编写你的新测试。这种开发风格被称为[测试驱动开发 \(test-driven development\)](#)，而且既能被应用在整个项目又能应用在单个补丁。在编写完你的测试



后，运行一下它们以确保它们确实是失败的（由于你还没有修复 bug 或者新增特性）。如果你的新测试没有失败，你就需要修复一下这些测试好让它们失败。毕竟，无论 bug 是否存在，回退测试都会通过，那在防止 bug 重新出现这事上是非常没有帮助的。

现在让我们动手做下实例。

## 为任务 #24788 编写一些测试

任务 #24788 推荐加点小功能：给 Form 类指定类级别（class level）属性加 前缀（prefix）的功能：

1. [...] forms which ship with apps could effectively namespace themselves such
2. that N overlapping form fields could be POSTed at once and resolved to the
3. correct form.

为了解决这个任务，我们将会为 **BaseForm** 类添加前缀属性。当实例化这个类时，传递一个前缀给 **init()** 方法也将会给被创建的实例设置这个前缀。但如果不传递一个前缀（或者传递 **None**）将会使用类级别的前缀。在我们修改之前，我们准备编写一些测试来验证我们的修改函数没有问题，在将来也没有问题。

切换到 Django 的 `tests/forms_tests/tests/` 文件夹里，打开 `test_forms.py` 文件。在第 1674 行 `test_forms_with_null_boolean` 函数之前添加以下代码：

```
1. # tests/forms_tests/tests/test_forms.py
2.
3. def test_class_prefix(self):
4.     # 前缀也可以在类级别指定
5.     class Person(Form):
6.         first_name = CharField()
7.         prefix = 'foo'
8.
9.     p = Person()
10.    self.assertEqual(p.prefix, 'foo')
11.
12.    p = Person(prefix='bar')
13.    self.assertEqual(p.prefix, 'bar')
```

这个新的测试：检查 [设置一个类级别前缀] 是否如预期一样地工作；并且在创建实例时传递一个前缀（prefix）参数，看是否也工作。

但这个测试的东西看起来特别困难.....

如果你之前从来没有接触过测试，第一眼看上去它们确实有点难。幸运的是，测试在电脑编程中是个 非常 大的主题，有很多出自这里的信息：

- [编写和运行测试文档](#) — 给 *Django* 编写测试。
- [单元测试介绍](#) — 《深入 Python》（一本给 *Python* 初学开发者的免费的在线书籍）。
- 在阅读完这些后，如果你求知若渴，还可以看看 *Python* 的官方文档 [unittest](#)。

## 运行你写的新测试

请记住，我们实际上没有对 **BaseForm** 做任何修改，所以我们的测试应该会失败。为了确保失败真的出现了，让我们在 **forms\_tests** 文件夹运行一下所有的测试。从命令行里，**cd** 切换进 Django 的 **tests/** 目录里，然后运行：

```
1. $ ./runtests.py forms_tests
```

如果测试运行正常，你应该看到和我们添加的测试方法相对应的一个失败。如果所有的测试都通过了，那么你要确认一下你已经将新测试添加到了上面合适的文件夹和类里。

## 为你的任务编写代码

下一步我们将给任务（ticket）[#24788](#) 添加函数描述。

### 为任务 **#24788** 编写代码

切换到 **django/django/forms/** 文件夹里，然后打开 **forms.py**。在第 72 行找到 **BaseForm** 类，然后在 **field\_order** 属性之后添加 前缀（**prefix**）类属性：

```
1. class BaseForm(object):
2.     # This is the main implementation of all the Form logic. Note that this
3.     # class is different than Form. See the comments by the Form class for
4.     # more information. Any improvements to the form API should be made to
5.     # *this* class, not to the Form class.
6.     field_order = None
7.     prefix = None
```

## 现在验证并通过你的测试

一旦你修改完了 Django，我们就需要确保之前写的测试都能通过，这样我们才能看到我们上面写的代码正确工作。为了在 **forms\_tests** 文件夹运行测试，**cd** 切换到 Django 的 **tests/** 目录里，然后运行：

```
1. $ ./runtests.py forms_tests
```

噢，我们写的那些测试好了！您应该仍然会看到一个例外失败：

```
1. AssertionError: None != 'foo'
```

我们忘记在 `init()` 方法里添加条件语句了。在 `django/forms/forms.py` 现在的第 87 行修改 `self.prefix = prefix`，添加一个条件语句：

```
1. if prefix is not None:
2.     self.prefix = prefix
```

重新运行测试，所有都应该是通过的。如果不是，请确认你像上面那样正确修改了 `BaseForm` 类，然后正确复制了新的测试。

## 第二次运行 Django 的测试套件

一旦你验证了你的补丁，你的测试都是正常工作的，运行整个测试套件是个好主意，去验证你的修改没有将 bug 引入到 Django 的其他地方。整个测试套件成功通过并不保证你的代码是没有 bug 的，尽管它确实有助于识别很多可能被忽视的错误和回退。

为了运行整个 Django 测试套件，`cd` 切换到 Django 的 `tests/` 目录里，然后运行：

```
1. $ ./runtests.py
```

只要你没看到任何失败，你就可以继续了。

## 写文档

这是个新功能，所以应该被记录到文档里。在 `django/docs/ref/forms/api.txt` 的第 1068 行（文件结尾）添加：

```
1. The prefix can also be specified on the form class::
2.
3.     >>> class PersonForm(forms.Form):
4.         ...
5.         prefix = 'person'
6.
7. .. versionadded:: 1.9
8.
9.     The ability to specify ``prefix`` on the form class was added.
```

由于这个新功能将被添加到还没发布的 Django 1.9 版本的发行说明中，在 `docs/releases/1.9.txt` 文件的 164 行的“Forms”部分：

1. \* A form prefix can be specified inside a form class, not only when
2. instantiating a form. See :ref:`form-prefix` for details.

有关编写文档的更多信息，包括有关 **versionadded** 的说明，可以查看[文档编写](#)。文章还包括了一篇关于怎样在本地建立文档副本的说明，好让你可以预览将被生成的 HTML。

## 预览你的修改

现在是时候看看我们的补丁做的所有修改了。为了显示你当前的 Django 副本（有你的修改）和你最初版本：

1. `$ git diff`

使用箭头键上下移动。

```
1. diff --git a/django/forms/forms.py b/django/forms/forms.py
2. index 509709f..d1370de 100644
3. --- a/django/forms/forms.py
4. +++ b/django/forms/forms.py
5. @@ -75,6 +75,7 @@ class BaseForm(object):
6.     # information. Any improvements to the form API should be made to *this*
7.     # class, not to the Form class.
8.     field_order = None
9. +    prefix = None
10.
11.     def __init__(self, data=None, files=None, auto_id='id_%s', prefix=None,
12.                  initial=None, error_class=ErrorList, label_suffix=None,
13. @@ -83,7 +84,8 @@ class BaseForm(object):
14.         self.data = data or {}
15.         self.files = files or {}
16.         self.auto_id = auto_id
17. -        self.prefix = prefix
18. +        if prefix is not None:
19. +            self.prefix = prefix
20.         self.initial = initial or {}
21.         self.error_class = error_class
22.         # Translators: This is the default suffix added to form field labels
23. diff --git a/docs/ref/forms/api.txt b/docs/ref/forms/api.txt
24. index 3bc39cd..008170d 100644
25. --- a/docs/ref/forms/api.txt
26. +++ b/docs/ref/forms/api.txt
27. @@ -1065,3 +1065,13 @@ You can put several Django forms inside one ``<form>`` tag. To give each
28.     >>> print(father.as_ul())
29.     <li><label for="id_father-first_name">First name:</label> <input type="text" name="father-
       first_name" id="id_father-first_name" /></li>
```

```

30.     <li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-
      last_name" id="id_father-last_name" /></li>
31. +
32. +The prefix can also be specified on the form class::
33. +
34. + >>> class PersonForm(forms.Form):
35. +     ...     ...
36. +     ...     prefix = 'person'
37. +
38. +.. versionadded:: 1.9
39. +
40. + The ability to specify ``prefix`` on the form class was added.
41. diff --git a/docs/releases/1.9.txt b/docs/releases/1.9.txt
42. index 5b58f79..f9bb9de 100644
43. --- a/docs/releases/1.9.txt
44. +++ b/docs/releases/1.9.txt
45. @@ -161,6 +161,9 @@ Forms
46. :attr:`~django.forms.Form.field_order` attribute, the ``field_order``
47. constructor argument , or the :meth:`~django.forms.Form.order_fields` method.
48.
49. +* A form prefix can be specified inside a form class, not only when
50. + instantiating a form. See :ref:`form-prefix` for details.
51. +
52. Generic Views
53. ^^^^^^^^^^^^^^
54.
55. diff --git a/tests/forms_tests/tests/test_forms.py b/tests/forms_tests/tests/test_forms.py
56. index 690f205..e07fae2 100644
57. --- a/tests/forms_tests/tests/test_forms.py
58. +++ b/tests/forms_tests/tests/test_forms.py
59. @@ -1671,6 +1671,18 @@ class FormsTestCase(SimpleTestCase):
60.         self.assertEqual(p.cleaned_data['last_name'], 'Lennon')
61.         self.assertEqual(p.cleaned_data['birthday'], datetime.date(1940, 10, 9))
62.
63. + def test_class_prefix(self):
64. +     # Prefix can be also specified at the class level.
65. +     class Person(Form):
66. +         first_name = CharField()
67. +         prefix = 'foo'
68. +
69. +     p = Person()
70. +     self.assertEqual(p.prefix, 'foo')
71. +
72. +     p = Person(prefix='bar')
73. +     self.assertEqual(p.prefix, 'bar')
74. +
75.     def test_forms_with_null_boolean(self):
76.         # NullBooleanField is a bit of a special case because its presentation (widget)
77.         # is different than its data. This is handled transparently, though.

```

当你预览完了这个补丁，按下 **q** 键回到命令行。如果补丁的内容看起来是对的，那就是时候 提交 (commit) 这些修改了。

## 提交补丁中的修改

为了提交修改：

```
1. $ git commit -a
```

为了输入提交的信息，会打开一个文本编辑器。根据[提交信息准则](#)，写下像这样的信息：

```
1. Fixed #24788 -- Allowed Forms to specify a prefix at the class level.
```

## 推送提交和拉取请求 (pull request)

在提交了补丁之后，把它发送到在 GitHub 上你 fork 下来的仓库（如果不一样，就把“ticket\_24788”替换为分支的名字）：

```
1. $ git push origin ticket_24788
```

通过 [Django](#) 的 [GitHub 页面](#)，你可以创建一个拉取请求。你会看到你的分支在 “你最近推送的分支 (Your recently pushed branches)” 里。点击旁边的 “对比和拉取请求 (Compare & pull request)”。

但在本教程里，请别那么做，在下一页里显示预览补丁，你可以点击 “创建拉取请求 (Create pull request)”

## 下一步

恭喜你，你已经学会了如何给 Django 创建拉取请求了！更多进阶技术细节你可以看[用 Git 和 GitHub 工作](#)。

现在，你可以通过帮助改进 Django 的代码库来使这些技能得到很好的使用了。

## 更多关于新贡献者的信息

在你给 Django 写补丁之前，这里有些关于贡献的信息，你应该看一下：

- 你应该确保读了 Django 的文档[认领任务和提交补丁](#)。它涵盖了 Trac 礼仪，如何申请自己的任务，期望的补丁代码风格以及很多其他的重要细节。
- 首次贡献者，也应该读下 Django 的[给第一次贡献者文档](#)。那里有很多好的建议。
- 读完这些后，如果你仍然渴望得到更多关于贡献的信息，你可以随时浏览 [Django 关于做贡献的文档](#)的剩下部分。那里包含了非常多的信息，也应该是回答你任何问题的第一来源。

## 寻找你真正的第一次任务

一旦你已经看完了所有的那些信息，你就准备好入门了，然后寻找属于你的任务，去编写补丁吧。特别留意那些写着“轻松 (easy pickings)”级别的任务。这些任务通常更简单，对于首次贡献者而言是非常棒的。一旦你熟悉了为 Django 做贡献，你就可以继续为更困难更复杂的任务编写补丁了。

如果你只是想在已经完成的任務上开始（没人会怪你），可以看看这个列表[需要补丁的简单任务](#)和[已有补丁但需要改进的简单任务](#)。如果你对编写测试很熟，你也可以看看这个列表[需要测试的简单任务](#)。要记得遵循 Django 的关于[认领任务和提交补丁](#)文档中提到的认领任务问题。

## 在创建完拉取请求后还要干什么呢？

在任务有了补丁之后，它需要被第二次审查。在提交了拉取请求后，通过设置标志在任务上，比如“有补丁了 (has patch)”、“不用测试了 (doesn't need tests)”等，来更新任务元数据，好让其他人为了审查而找到它。做贡献并不意味着总是从头开始写补丁。审查已经存在的补丁也是非常有帮助的一次贡献。查看 [Triaging tickets](#) 获取更多细节。