Django 中文文档 1.8

wizardforcel

Published with GitBook

目錄

Django 中文文档 1.8

译者：Django 文档协作翻译小组，来源：Django documentation。

Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

下载

- PDF下载
- EPUB下载
- MOBI下载

新手入门

初次接触 Django 或编程吗？ 从这里开始吧！

从零开始

Django 初探

由于Django是在一个快节奏的新闻编辑室环境下开发出来的，因此它被设计成让普通的网站开发工作简单而快 捷。以下简单介绍了如何用 Django 编写一个数据库驱动的Web应用程序。

本文档的目标是给你描述足够的技术细节能让你理解Django是如何工作的，但是它并不表示是一个新手指南或参考目录 – 其实这些我们都有! 当你准备新建一个项目，你可以 从新手指南开始 或者 深入阅读详细的文档.

设计你的模型(model)

尽管你在 Django 中可以不使用数据库，但是它提供了一个完善的可以用 Python 代码描述你的数据库结构的对象关联映射(ORM)。

数据模型语法 提供了许多丰富的方法来展现你的模型 – 到目前为止，它已经解决了两个多年积累下来数据库架构问题。下面是个简单的例子，可能被保存为 mysite/news/models.py:

```
class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __unicode__(self):
        return self.full_name

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

    def __unicode__(self):
        return self.headline
```

安装它

下一步，运行 Django 命令行工具来自动创建数据库表：

```
manage.py syncdb
```

syncdb 命令会查找你所有可用的模型(models)然后在你的数据库中创建还不存在的数据库表。

享用便捷的 API

接着，你就可以使用一个便捷且功能丰富的 *Python API* 来访问你的数据。API 是动态生成的，不需要代码生成:

```
# 导入我们在 "news "应用中创建的模型。
>>> from news.models import Reporter, Article

# 在系统中还没有 reporters 。
>>> Reporter.objects.all()
[]

# 创建一个新的 Reporter 。
>>> r = Reporter(full_name='John Smith')

# 将对象保存到数据库。你需要显示的调用 save() 方法。
>>> r.save()

# 现在它拥有了一个ID。
>>> r.id
1

# 现在新的 reporter 已经存在数据库里了。
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# 字段被表示为一个 Python 对象的属性。
>>> r.full_name
'John Smith'

# Django 提供了丰富的数据库查询 API。
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist. Lookup parameters were {'id': 2}

# 创建一个 article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()

# 现在 article 已经存在数据库里了。
>>> Article.objects.all()
[<Article: Django is cool>]

# Article 对象有 API 可以访问到关联到 Reporter 对象。
>>> r = a.reporter
>>> r.full_name
'John Smith'

# 反之亦然：Reporter 对象也有访问 Article 对象的API。
>>> r.article_set.all()
[<Article: Django is cool>]

# API 会在幕后高效的关联表来满足你的关联查询的需求。
# 以下例子是找出名字开头为 "John" 的 reporter 的所有 articles 。
>>> Article.objects.filter(reporter__full_name__startswith="John")
[<Article: Django is cool>]

# 通过更改一个对象的属性值，然后再调用 save() 方法来改变它。
>>> r.full_name = 'Billy Goat'
>>> r.save()

# 调用 delete() 方法来删除一个对象。
>>> r.delete()
```

一个动态的管理接口：它不仅仅是个脚手架 – 还是个完整的房子

一旦你的 models 被定义好，Django 能自动创建一个专业的，可以用于生产环境的 管理界面 – 一个可让授权用户添加，修改和删除对象的网站。它使用起来非常简单只需在你的 admin site 中注册你的模型即可。:

```
# In models.py...

from django.db import models

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)


# In admin.py in the same directory...

import models
from django.contrib import admin

admin.site.register(models.Article)
```

这种设计理念是你的网站一般是由一个员工,或者客户，或者仅仅是你自己去编辑 – 而你应该不会想要仅仅为了管理内容而去创建后台界面。

在一个创建 Django 应用的典型工作流中，首先需要创建模型并尽可能快地启动和运行 admin sites， 让您的员工(或者客户)能够开始录入数据。然后,才开发展现数据给公众的方式。

设计你的 **URLs**

一个干净的，优雅的 URL 方案是一个高质量 Web 应用程序的重要细节。 Django 鼓励使用漂亮的 URL 设计，并且不鼓励把没必要的东西放到 URLs 里面，像 .php 或 .asp.

为了给一个 app 设计 URLs，你需要创建一个 Python 模块叫做 URLconf。这是一个你的 app 内容目录， 它包含一个简单的 URL 匹配模式与 Python 回调函数间的映射关系。这有助于解耦 Python 代码和 URLs 。

这是针对上面 Reporter/Article 例子所配置的 URLconf 大概样子:

```
from django.conf.urls import patterns

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

上面的代码映射了 URLs，从一个简单的正则表达式，到 Python 回调函数(“views”)所在的位置。 正则表达式通过圆括号来“捕获” URLs 中的值。当一个用户请求一个页面时， Django 将按照顺序去匹配每一个模式，并停在第一个匹配请求的 URL 上。(如果没有匹配到， Django 将会展示一个404的错误页面。) 整个过程是极快的，因为在加载时正则表达式就进行了编译。

一旦有一个正则表达式匹配上了，Django 将导入和调用对应的视图，它其实就是一个简单的 Python 函数。每个视图将得到一个 request 对象 – 它包含了 request 的 meta 信息 – 和正则表达式所捕获到的值。

例如：如果一个用户请求了个 URL “/articles/2005/05/39323/”, Django 将会这样调用函数 news.views.article_detail(request, '2005', '05', '39323').

编写你的视图(views)

每个视图只负责两件事中的一件：返回一个包含请求页面内容的 HttpResponse 对象; 或抛出一个异常如 Http404 。至于其他就靠你了。

通常，一个视图会根据参数来检索数据，加载一个模板并且根据该模板来呈现检索出来的数据。 下面是个接上例的 year_archive 例子

```
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    return render_to_response('news/year_archive.html', {'year': year, 'article_list': a_
```

这个例子使用了 Django 的 模板系统，该模板系统功能强大且简单易用，甚至非编程人员也会使用。

设计你的模板(templates)

上面的例子中载入了 news/year_archive.html 模板。

Django 有一个模板搜索路径板，它让你尽可能的减少冗余而重复利用模板。在你的 Django设置中，你可以指定一个查找模板的目录列表。如果一个模板没有在这个 列表中，那么它会去查找第二个，然后以此类推。

假设找到了模板 news/year_archive.html 。下面是它大概的样子:

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

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

变量使用双花括号包围。`{{ article.headline }} 表示 “输出 article 的 headline 属性”。而点符号不仅用于表示属性查找，还可用于字典的键值查找、索引查找和函数调用。

注意 `{{ article.pub_date|date:"F j, Y" }}` 使用了 Unix 风格的“管道”(“|”符合)。这就是所谓的模板过滤器，一种通过变量来过滤值的方式。本例中，Python datetime 对象被过滤成指定的格式(在 PHP 的日期函数中可以见到这种变换)。

你可以无限制地串联使用多个过滤器。你可以编写自定义的过滤器。你可以定制自 己的模板标记，在幕后运行自定义的 Python 代码。

最后，Django 使用了“模板继承”的概念：这就是 `{% extends "base.html" %}` 所做的事。它意味着 “首先载入名为 ‘base’ 的模板中的内容到当前模板，然后再处理本模板中的其余内容。”总之，模板继承让你在模板间大大减少冗余内容：每一个模板只需要定义它独特的部分即可。

下面是使用了 静态文件 的 “base.html” 模板的大概样子:

```
{% load staticfiles %}
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="{% static "images/sitelogo.png" %}" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

简单地说，它定义了网站的外观（含网站的 logo ），并留下了个“洞”让子模板来填充。这使站点的重新设计变得非常容易，只需改变一个文件 – “base.html” 模板。

它也可以让你创建一个网站的多个版本，不同的基础模板，而重用子模板。 Django 的创建者已经利用这一技术来创造了显著不同的手机版本的网站 – 只需创建一个新的基础模板。

请注意，如果你喜欢其他模板系统，那么你可以不使用 Django 的模板系统。 虽然 Django 的模板系统特别集成了 Django 的模型层，但并没有强制你使用它。同理，你也可以不使用 Django 的数据库 API。您可以使用其他数据库抽象层，您可以读取 XML 文件，你可以从磁盘

中读取文件，或任何你想要的方法去操作数据。 Django 的每个组成部分： 模型、视图和模板都可以解耦，以后会谈到。

这仅仅是一点皮毛

这里只是简要概述了 Django 的功能。以下是一些更有用的功能：

一个 缓存框架 可以与 memcached 或其他后端缓存集成。 一个 聚合框架 可以让创建 RSS 和 Atom 的 feeds 同写一个小小的 Python 类一样容易。 更性感的自动创建管理站点功能 – 本文仅仅触及了点皮毛。 显然，下一步你应该 下载 Django，阅读 入门教程 并且加入 社区. 感谢您的关注！

> 译者：Django 文档协作翻译小组，原文：Overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

快速安装指南

在你开始使用 Django 之前，你需要先安装它。我们有一个 完整安装指南 它涵盖了所有的安装步骤和可能遇到的问题；本指南将会给你一个最简单、简洁的安装指引。

安装 Python

作为一个 Web 框架，Django 需要使用 Python 。它适用 2.6.5 到 2.7 的所有 Python 版本。它还具有 3.2 和 3.3 版本的实验性支持。所有这些 Python 版本都包含一个轻量级的数据库名叫 SQLite 。因此你现在还不需要建立一个数据库。

在 http://www.python.org 获取 Python 。如果你使用 Linux 或者 Mac OS X，那很可能已经安装了 Python 。

> 在 **Jython** 使用 **Django**
>
> 如果你使用 *Jython* (一个在 Java 平台上实现的 Python)，你需要遵循一些额外的步骤。查看 在 *Jyton* 上运行 *Python* 获取详细信息。

在你的终端命令行(shell)下输入 python 来验证是否已经安装 Python ; 你将看到如下类似的提示信息:

```
Python 3.3.3 (default, Nov 26 2013, 13:33:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

建立一个数据库

若你需要一个“大”数据库引擎，例如：PostgreSQL，MySQL，或 Oracle，那此步骤是需要的。 想要安装这样一个数据库，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 在安装新版本之前先卸载旧版本的 *Django*.

安装 Django

你可以使用下面这简单的三个方式来安装 Django:

- 安装 你的操作系统所提供供的发行包 。 对于操作系统提供了 Django 安装包的人来说，这是最快捷的安装方法。
- 安装官方正式发布的版本 。 这是对于想要安装一个稳定版本而不介意运行一个稍旧版本的 Django 的人来说是最好的方式。
- 安装最新的开发版本 。 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。

> 总是参考你所使用的对应版本的 **Django** 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在开发版中新增的标记。这个标记表明这个特性仅适用开发版的 Django ， 而他们可能不在官方版本发布。

验证安装

为了验证 Django 被成功的安装到 Python 中，在你的终端命令行 (shell) 下输入 python 。 然后在 Python 提示符下，尝试导入 Django:

```
>>> import django
>>> print(django.get_version())
1.8
```

你可能已安装了其他版本的 Django 。

安装完成！

安装完成 – 现在你可以 学习入门教程.

> 译者：Django 文档协作翻译小组，原文：Installation。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

教程

编写你的第一个 Django 程序 第1部分

让我们通过例子来学习。

在本教程中，我们将引导您创建一个基本的投票应用。

它将包含两部分：

- 一个公共网站，可让人们查看投票的结果和让他们进行投票。
- 一个管理网站，可让你添加、修改和删除投票项目。

我们假设你已经 安装了 Django 。你可以运行以下命令来验证是否已经安装了 Django 和运行着的版本号：

```
python -c "import django; print(django.get_version())"
```

你应该看到你安装的 Django 版本或一个提示你 “No module named django” 的错误。此外，还应该检查下你的版本与本教程的版本是否一致。 若不一致，你可以参考 Django 版本对应的教程或者更新 Django 到最新版本。

请参考 如何安装 Django 中的意见先删除旧版本的 Django 再安装一个新的。

> 在哪里可以获得帮助：
>
> 如果您在学习本教程中遇到问题，请在 django-users 上发贴或者在 #django on irc.freenode.net 上与其他可能会帮助您的 Django 用户交流。

创建一个项目

如果这是你第一次使用 Django ，那么你必须进行一些初始设置。也就是通过自动生成代码来建立一个 Django 项目 project – 一个 Django 项目的设置集，包含了数据库配置、 Django 详细选项设置和应用特性配置。

在命令行中，使用 cd 命令进入你想存储代码所在的目录，然后运行以下命令：

```
django-admin.py startproject mysite
```

这将在当前目录创建一个 mysite 目录。如果失败了，请查看 Problems running django-admin.py.

> Note
>
> 你需要避免使用 python 保留字或 Django 组件名作为项目的名称。尤其是你应该避免使用的命名如： django (与 Django 本身会冲突) 或者 test (与 Python 内置的包名会冲突).
>
> 这段代码应该放在哪里？
>
> 如果你有一般 PHP 的编程背景（未使用流行的框架），可能会将你的代码放在 Web 服务器的文档根目录下（例如： `/var/www`）。而在 Django 中，你不必这么做。将任何 Python 代码放在你的 Web 服务器文档根目录不会是一个好主意，因为这可能会增加人们通过 Web 方式查看到你的代码的风险。这不利于安全。

将你的代码放在你的文档根目录 以外 的某些目录, 例如 /home/mycode 。

让我们来看看 startproject 都创建了些什么:

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

> 和你看到的不一样？
>
> 默认的项目布局最近刚刚改变过。如果你看到的是一个“扁平”结构的目录布局（没有内层 mysite/ 目录），你很可能正在使用一个和本教程版本不一致的 Django 版本。你需要切换到对应的旧版教程或者使用较新的 Django 版本。

这些文件是：

- 外层 mysite/ 目录只是你项目的一个容器。对于 Django 来说该目录名并不重要; 你可以重命名为你喜欢的。
- manage.py: 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互。 你可以在 django-admin.py and manage.py 中查看关于 manage.py 所有的细节。
- 内层 mysite/ 目录是你项目中的实际 Python 包。该目录名就是 Python 包名，通过它你可以导入它里面的任何东西。 (e.g. import mysite.settings).
- mysite/**init**.py: 一个空文件，告诉 Python 该目录是一个 Python 包。(如果你是 Python 新手，请查看官方文档了解 关于包的更多内容 。)
- mysite/settings.py: 该 Django 项目的设置/配置。请查看 Django settings 将会告诉你如何设置。
- mysite/urls.py: 该 Django 项目的 URL 声明; 一份由 Django 驱动的网站“目录”。请查看 URL dispatcher 可以获取更多有关 URL 的信息。
- mysite/wsgi.py: 一个 WSGI 兼容的 Web 服务器的入口，以便运行你的项目。请查看 How to deploy with WSGI 获取更多细节。

开发用服务器

让我们来验证是否工作。从外层 mysite 目录切换进去，若准备好了就运行命令 `python manage.py runserver`。你将会看到命令行输出如下内容：

```
Performing system checks...

0 errors found
May 13, 2015 - 15:50:53
Django version 1.8, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

你已经启动了 Django 开发服务器，一个纯粹的由 Python 编写的轻量级 Web 服务器。我们在 Django 内包含了这个服务器，这样你就可以迅速开发了，在产品投入使用之前不必去配置一台生产环境下的服务器 – 例如 Apache 。

现在是一个很好的提示时机：不要 在任何类似生产环境中使用此服务器。它仅适用于开发环境。(我们提供的是 Web 框架的业务，而不是 Web 服务器。)

现在服务器正在运行中，请在你的 Web 浏览器中访问 http://127.0.0.1:8000/ 。 你会看到一个令人愉悦的，柔和的淡蓝色 “Welcome to Django” 页面。它工作正常！

> 更改端口号
>
> 默认情况下，runserver 命令启动的开发服务器只监听本地 IP 的 8000 端口。
>
> 如果你想改变服务器的端口，把它作为一个命令行参数传递即可。例如以下命令启动的服务器将监听 8080 端口：
>
> ```
> python manage.py runserver 8080
> ```
>
> 如果你想改变服务器 IP，把它和端口号一起传递即可。因此，要监听所有公共 IP 地址（如果你想在其他电脑上炫耀你的工作），请使用：
>
> ```
> python manage.py runserver 0.0.0.0:8000
> ```
>
> 有关开发服务器的完整文档可以在 runserver 内参考。

数据库设置

现在，编辑 mysite/settings.py 。 这是一个普通的 Python 模块，包含了代表 Django 设置的模块级变量。 更改 DATABASES 中 'default' 下的以下键的值，以匹配您的数据库连接设置。

- ENGINE – 从 'django.db.backends.postgresql_psycopg2', 'django.db.backends.mysql',

'django.db.backends.sqlite3', 'django.db.backends.oracle' 中选一个， 至于其他请查看 also available.
- NAME – 你的数据库名。如果你使用 SQLite，该数据库将是你计算机上的一个文件；在这种情况下，NAME 将是一个完整的绝对路径，而且还包含该文件的名称。如果该文件不存在，它会在第一次同步数据库时自动创建（见下文）。

当指定路径时， 总是使用正斜杠，即使是在 Windows 下(例如： `C:/homes/user/mysite/sqlite3.db`)。

- USER – 你的数据库用户名 (SQLite 下不需要) 。
- PASSWORD – 你的数据库密码 (SQLite 下不需要) 。
- HOST – 你的数据库主机地址。如果和你的数据库服务器是同一台物理机器，请将此处保留为空 (或者设置为 127.0.0.1) (SQLite 下不需要) 。查看 HOST 了解详细信息。

如果你是新建数据库，我们建议只使用 SQLite ，将 ENGINE 改为 'django.db.backends.sqlite3' 并且将 NAME 设置为你想存放数据库的地方。 SQLite 是内置在 Python 中的，因此你不需要安装任何东西来支持你的数据库。

> Note
>
> 如果你使用 PostgreSQL 或者 MySQL，确保你已经创建了一个数据库。还是通过你的数据库交互接口中的 “CREATE DATABASE database_name;” 命令做到这一点的。 如果你使用 SQLite ， 你不需要事先创建任何东西 - 在需要的时候，将会自动创建数据库文件。当你编辑 settings.py 时，将 TIME_ZONE 修改为你所在的时区。默认值是美国中央时区（芝加哥）。

同时，注意文件底部的 INSTALLED_APPS 设置。它保存了当前 Django 实例已激活的所有 Django 应用。每个应用可以被多个项目使用，而且你可以打包和分发给其他人在他们的项目中使用。

默认情况下，INSTALLED_APPS 包含以下应用，这些都是由 Django 提供的：

- django.contrib.auth – 身份验证系统。
- django.contrib.contenttypes – 内容类型框架。
- django.contrib.sessions – session 框架。
- django.contrib.sites – 网站管理框架。
- django.contrib.messages – 消息框架。
- django.contrib.staticfiles – 静态文件管理框架。

这些应用在一般情况下是默认包含的。

所有这些应用中每个应用至少使用一个数据库表，所以在使用它们之前我们需要创建数据库中的表。要做到这一点，请运行以下命令：

```
python manage.py syncdb
```

syncdb 命令参照 INSTALLED_APPS 设置，并在你的 settings.py 文件所配置的数据库中创建必要的数据库表。每创建一个数据库表你都会看到一条消息，接着你会看到一个提示询问你是否想要在身份验证系统内创建个超级用户。按提示输入后结束。

如果你感兴趣，可以在你的数据库命令行下输入：`dt` (PostgreSQL), SHOW TABLES; (MySQL), 或 .schema (SQLite) 来列出 Django 所创建的表。

> 极简主义者
>
> 就像我们上面所说的，一般情况下以上应用都默认包含在内，但不是每个人都需要它们。如果不需要某些或全部应用，在运行 syncdb 命令前可从 INSTALLED_APPS 内随意注释或删除相应的行。syncdb 命令只会为 INSTALLED_APPS 内的应用创建表。

创建模型

现在你的项目开发环境建立好了， 你可以开工了。

你通过 Djaong 编写的每个应用都是由 Python 包组成的，这些包存放在你的 Python path 中并且遵循一定的命名规范。 Django 提供了个实用工具可以自动生成一个应用的基本目录架构，因此你可以专注于编写代码而不是去创建目录。

> 项目 (Projects) vs. 应用 (apps)
>
> 项目与应用之间有什么不同之处？应用是一个提供功能的 Web 应用 – 例如：一个博客系统、一个公共记录的数据库或者一个简单的投票系统。 项目是针对一个特定的 Web 网站相关的配置和其应用的组合。一个项目可以包含多个应用。一个应用可以在多个项目中使用。

你的应用可以存放在 Python path 中的任何位置。在本教材中，我们将通过你的 manage.py 文件创建我们的投票应用，以便它可以作为顶层模块导入，而不是作为 mysite 的子模块。

要创建你的应用，请确认与 manage.py 文件在同一的目录下并输入以下命令：

```
python manage.py startapp polls
```

这将创建一个 polls 目录，其展开的样子如下所示：:

```
polls/
    __init__.py
    models.py
    tests.py
    views.py
```

此目录结构就是投票应用。

在 Django 中编写一个有数据库支持的 Web 应用的第一步就是定义你的模型 – 从本质上讲就是数据库设计及其附加的元数据。

> 哲理
>
> 模型是有关你数据的唯一且明确的数据源。它包含了你所要存储的数据的基本字段和行为。 Django 遵循 DRY 原则 。目标是为了只在一个地方定义你的数据模型就可从中自动获取数据。

在这简单的投票应用中，我们将创建两个模型： Poll 和 Choice 。 Poll 有问题和发布日期两个字段。 `Choice` 有两个字段： 选项 (choice) 的文本内容和投票数。每一个 Choice 都与一个 Poll 关联。

这些概念都由简单的 Python 类来表现。编辑 polls/models.py 文件后如下所示：

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

代码很简单。每个模型都由继承自 django.db.models.Model 子类的类来描述。 每个模型都有一些类变量，每一个类变量都代表了一个数据库字段。

每个字段由一个 Field 的实例来表现 – 比如 CharField 表示字符类型的字段和 DateTimeField 表示日期时间型的字段。这会告诉 Django 每个字段都保存了什么类型的数据。

每一个 Field 实例的名字就是字段的名字（如： question 或者 pub_date ），其格式属于亲和机器式的。在你的 Python 的代码中会使用这个值，而你的数据库会将这个值作为表的列名。

你可以在初始化 Field 实例时使用第一个位置的可选参数来指定人类可读的名字。这在Django 的内省部分中被使用到了，而且兼作文档的一部分来增强代码的可读性。若字段未提供该参数，Django 将使用符合机器习惯的名字。在本例中，我们仅定义了一个符合人类习惯的字段名 Poll.pub_date 。对于模型中的其他字段，机器名称就已经足够替代人类名称了。

一些 Field 实例是需要参数的。 例如 CharField 需要你指定 `~django.db.models.CharField.max_length` 。这不仅适用于数据库结构，以后我们还会看到也用于数据验证中。

一个 Field 实例可以有不同的可选参数； 在本例中，我们将 votes 的 default 的值设为 0 。

最后，注意我们使用了 ForeignKey 定义了一个关联。它告诉 Django 每一个 `Choice` 关联一个 Poll 。 Django 支持常见数据库的所有关联：多对一（ many-to-ones ），多对多（ many-to-manys ） 和 一对一 （ one-to-ones ）。

激活模型

刚才那点模型代码提供给 Django 大量信息。有了这些 Django 就可以做：

为该应用创建对应的数据库架构 (CREATE TABLE statements)。 为 Poll 和 Choice 对象创建 Python 访问数据库的 API。 但首先，我们需要告诉我们的项目已经安装了 polls 应用。

> 哲理
>
> Django 应用是“可插拔的”：你可以在多个项目使用一个应用，你还可以分发应用，因为它们没有被捆绑到一个给定的 Django 安装环境中。

再次编辑 settings.py 文件，在 INSTALLED_APPS 设置中加入 'polls' 字符。因此结果如下所示：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

现在 Django 已经知道包含了 polls 应用。让我们运行如下命令：

```
python manage.py sql polls
```

你将看到类似如下所示内容 (有关投票应用的 CREATE TABLE SQL 语句)：

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id") DEFERRABLE INITIALLY DEFERR
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

请注意如下事项：

- 确切的输出内容将取决于您使用的数据库会有所不同。
- 表名是自动生成的，通过组合应用名 (polls) 和小写的模型名 – poll 和 choice 。 (你可以

重写此行为。)
- 主键 (IDs) 是自动添加的。(你也可以重写此行为。)
- 按照惯例，Django 会在外键字段名上附加 "_id" 。 (是的，你仍然可以重写此行为。)
- 外键关系由 REFERENCES 语句显示声明。
- 生成 SQL 语句时针对你所使用的数据库，会为你自动处理特定于数据库的字段，例如 auto_increment (MySQL), serial (PostgreSQL), 或 or integer primary key (SQLite) 。 在引用字段名时也是如此 – 比如使用双引号或单引号。 本教材的作者所使用的是 PostgreSQL，因此例子中输出的是 PostgreSQL 的语法。
- 这些 sql 命令其实并没有在你的数据库中运行过 - 它只是在屏幕上显示出来，以便让你了解 Django 认为什么样的 SQL 是必须的。 如果你愿意，可以把 SQL 复制并粘帖到你的数据库命令行下去执行。 但是，我们很快就能看到， Django 提供了一个更简单的方法来执行此 SQL 。

如果你感兴趣，还可以运行以下命令：

- python manage.py validate – 检查在构建你的模型时是否有错误。
- python manage.py sqlcustom polls – 输出为应用定义的任何 custom SQL statements (例如表或约束的修改) 。
- python manage.py sqlclear polls – 根据存在于你的数据库中的表 (如果有的话) ，为应用输出必要的 DROP TABLE 。
- python manage.py sqlindexes polls – 为应用输出 CREATE INDEX 语句。
- python manage.py sqlall polls – 输出所有 SQL 语句：sql, sqlcustom, 和 sqlindexes 。

看看这些输出的命令可以帮助你理解框架底层实际上处理了些什么。

现在，再次运行 syncdb 命令在你的数据库中创建这些模型对应的表：

```
python manage.py syncdb
```

syncdb 命令会给在 INSTALLED_APPS 中有但数据库中没有对应表的应用执行 sqlall 操作。该操作会为你上一次执行 syncdb 命令以来在项目中添加的任何应用创建对应的表、初始化数据和创建索引。 syncdb 命令只要你喜欢就可以任意调用，并且它仅会创建不存在的表。

请阅读 django-admin.py documentation 文档了解 manage.py 工具更多的功能。

玩转 API

现在，我们进入 Python 的交互式 shell 中玩弄 Django 提供给你的 API 。要调用 Python sell ，使用如下命令：

```
python manage.py shell
```

我们当前使用的环境不同于简单的输入 “python” 进入的 shell 环境，因为 manage.py 设置了 DJANGO_SETTINGS_MODULE 环境变量，该变量给定了 Django 需要导入的 settings.py 文件所在路径。

> 忽略 manage.py
>
> 若你不想使用 manage.py ，也是没有问题的。仅需要将 DJANGO_SETTINGS_MODULE 环境变量值设为 mysite.settings 并在与 manage.py 文件所在同一目录下运行 python （或确保目录在 Python path 下，那 import mysite 就可以了）。
>
> 想了解更多的信息，请参考 django-admin.py 文档 。

一旦你进入了 shell，就可通过 database API 来浏览数据：:

```
>>> from polls.models import Poll, Choice   # Import the model classes we just wrote.

#  系统中还没有 polls 。
>>> Poll.objects.all()
[]

# 创建一个新 Poll 。
# 在默认配置文件中时区支持配置是启用的，
# 因此 Django 希望为 pub_date 字段获取一个 datetime  with tzinfo 。使用了 timezone.now()
# 而不是 datetime.datetime.now() 以便获取正确的值。
>>> from django.utils import timezone
>>> p = Poll(question="What's new?", pub_date=timezone.now())

# 保存对象到数据库中。你必须显示调用 save() 方法。
>>> p.save()

# 现在对象拥有了一个ID 。请注意这可能会显示 "1L" 而不是 "1"，取决于
# 你正在使用的数据库。 这没什么大不了的，它只是意味着你的数据库后端
# 喜欢返回的整型数作为 Python 的长整型对象而已。
>>> p.id
1

# 通过 Python 属性访问数据库中的列。
>>> p.question
"What's new?"
>>> p.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# 通过改为属性值来改变值，然后调用 save() 方法。
>>> p.question = "What's up?"
>>> p.save()

# objects.all() 用以显示数据库中所有的 polls 。
>>> Poll.objects.all()
[<Poll: Poll object>]
```

请稍等。`<Poll: Poll object>` 这样显示对象绝对是无意义的。让我们编辑 polls 模型（在 polls/models.py 文件中） 并且给 Poll 和 Choice 都添加一个 **unicode**() 方法来修正此错误：

```
class Poll(models.Model):
    # ...
    def __unicode__(self):
        return self.question

class Choice(models.Model):
    # ...
    def __unicode__(self):
        return self.choice_text
```

给你的模型添加 **unicode**() 方法是很重要的， 不仅是让你在命令行下有明确提示，而且在 Django 自动生成的管理界面中也会使用到对象的呈现。

> 为什么是 **unicode**() 而不是 **str**()?
>
> 如果你熟悉 Python，那么你可能会习惯在类中添加 **str**() 方法而不是 **unicode**() 方法。We use 我们在这里使用 **unicode**() 是因为 Django 模型默认处理的是 Unicode 格式。当所有存储在数据库中的数据返回时都会转换为 Unicode 的格式。
>
> Django 模型有个默认的 **str**() 方法 会去调用 **unicode**() 并将结果转换为 UTF-8 编码的字符串。这就意味着 unicode(p) 会返回一个 Unicode 字符串，而 str(p) 会返回一个以 UTF-8 编码的普通字符串。
>
> 如果这让你感觉困惑，那么你只要记住在模型中添加 **unicode**() 方法。 运气好的话，这些代码会正常运行。

请注意这些都是普通的 Python 方法。让我们来添加个自定义方法，为了演示而已：

```
import datetime
from django.utils import timezone
# ...
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

请注意，增加了 import datetime 和 from django.utils import timezone, 是为了分别引用 Python 的标准库 datetime 模块和 Django 的 django.utils.timezone 中的 time-zone-related 实用工具 。如果你不熟悉在 Python 中处理时区，你可以在 时区支持文档 学到更多。

保存这些更改并且再次运行 python manage.py shell 以开启一个新的 Python shell:

```
>>> from polls.models import Poll, Choice

# 确认我们附加的  __unicode__() 正常运行。
>>> Poll.objects.all()
[<Poll: What's up?>]

# Django 提供了一个丰富的数据库查询 API ，
# 完全由关键字参数来驱动。
>>> Poll.objects.filter(id=1)
[<Poll: What's up?>]
>>> Poll.objects.filter(question__startswith='What')
[<Poll: What's up?>]
```

```
# 获取今年发起的投票。
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Poll.objects.get(pub_date__year=current_year)
<Poll: What's up?>

# 请求一个不存在的 ID ，这将引发一个异常。
>>> Poll.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Poll matching query does not exist. Lookup parameters were {'id': 2}

# 根据主键查询是常见的情况，因此 Django 提供了一个
# 主键精确查找的快捷方式。
# 以下代码等同于 Poll.objects.get(id=1).
>>> Poll.objects.get(pk=1)
<Poll: What's up?>

# 确认我们自定义方法正常运行。
>>> p = Poll.objects.get(pk=1)
>>> p.was_published_recently()
True

# 给 Poll 设置一些 Choices 。通过 create 方法调用构造方法去创建一个新
# Choice 对象实例，执行 INSERT 语句后添加该 choice 到
# 可用的 choices 集中并返回这个新建的 Choice 对象实例。 Django 创建了
# 一个保存外键关联关系的集合 ( 例如 poll 的 choices) 以便可以通过 API
# 去访问。
>>> p = Poll.objects.get(pk=1)

# 从关联对象集中显示所有 choices  -- 到目前为止还没有。
>>> p.choice_set.all()
[]

# 创建三个 choices 。
>>> p.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice 对象拥有访问它们关联的 Poll 对象的 API 。
>>> c.poll
<Poll: What's up?>

# 反之亦然： Poll 对象也可访问 Choice 对象。
>>> p.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> p.choice_set.count()
3

# 只要你需要 API 会自动连续关联。
# 使用双下划线来隔离关联。
# 只要你想要几层关联就可以有几层关联，没有限制。
# 寻找和今年发起的任何 poll 有关的所有 Choices
# ( 重用我们在上面建立的 'current_year' 变量 )。
>>> Choice.objects.filter(poll__pub_date__year=current_year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# 让我们使用 delete() 删除 choices 中的一个。
>>> c = p.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

欲了解更多有关模型关系的信息，请查看 访问关联对象 。欲了解更多有关如何使用双下划线来通过 API 执行字段查询的，请查看 字段查询 。 如需完整的数据库 API 信息，请查看我们的 数据库 API 参考 。

当你对 API 有所了解后, 请查看 教程 第2部分 来学习 Django 的自动生成的管理网站是如何工作的。

> 译者：Django 文档协作翻译小组，原文：Part 1: Models。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第2部分

本教程上接 教程 第1部分 。 我们将继续开发 Web-poll 应用，并且专注在 Django 的 自动生成的管理网站上。

> 哲理
>
> 为你的员工或客户生成添加、修改和删除内容的管理性网站是个单调乏味的工作。 出于这个原因，Django 根据模型完全自动化创建管理界面。
>
> Django 是在新闻编辑室环境下编写的，“内容发表者”和“公共”网站之间有 非常明显的界线。网站管理员使用这个系统来添加新闻、事件、体育成绩等等， 而这些内容会在公共网站上显示出来。Django 解决了为网站管理员创建统一 的管理界面用以编辑内容的问题。
>
> 管理界面不是让网站访问者使用的。它是为网站管理员准备的。

启用管理网站

- 默认情况下 Django 管理网站是不启用的 – 它是可选的。 要启用管理网站，需要做三件事：
- 在 INSTALLED_APPS 设置中取消 "django.contrib.admin" 的注释。
- 运行 python manage.py syncdb 命令。既然你添加了新应用到 INSTALLED_APPS 中，数据库表就需要更新。
- 编辑你的 mysite/urls.py 文件并且将有关管理的行取消注释 – 共有三行取消了注释。该文件是 URLconf ；我们将在下一个教程中深入探讨 URLconfs 。现在，你需要知道的是它将 URL 映射到应用。最后你拥有的 urls.py 文件看起来像这样:

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', '{{ project_name }}.views.home', name='home'),
    # url(r'^{{ project_name }}/', include('{{ project_name }}.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

(粗体显示的行就是那些需要取消注释的行。)

启动开发服务器

让我们启动开发服务器并浏览管理网站。

回想下教程的第一部分，像如下所示启动你的开发服务器：

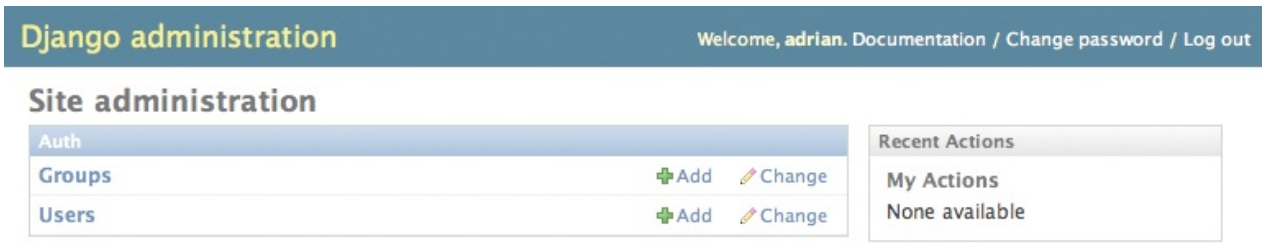
```
python manage.py runserver
```

现在，打开一个浏览器并在本地域名上访问 “/admin/” – 例如 http://127.0.0.1:8000/admin/ 。你将看到管理员的登录界面：

> 和你看到的不一样？
>
> 如果看到这，而不是上面的登录界面，那你应该得到一个类似如下所示的错误页面报告：
>
> ```
> ImportError at /admin/ cannot import name patterns ...
> ```
>
> 那么你很可能使用的 Django 版本不符合本教程的版本。 你可以切换到对应的旧版本教程去或者更新到较新的 Django 版本。

进入管理网站

现在尝试登录进去。（还记得吗？在本教程的第一部分时你创建过一个超级用户的帐号。如果你没有创建或忘记了密码，你可以 另外创建一个 。） 你将看到 Djaong 的管理索引页：



你将看到一些可编辑的内容，包括 groups， users 和 sites 。这些都是 Django 默认情况下自带的核心功能。

使 poll 应用的数据在管理网站中可编辑

但是 poll 应用在哪？ 它可是没有在管理网站的首页上显示啊。

只需要做一件事：我们需要告诉管理网站 Poll 对象要有一个管理界面。为此，我们在你的 polls 目录下创建一个名为 admin.py 的文件，并添加如下内容：:

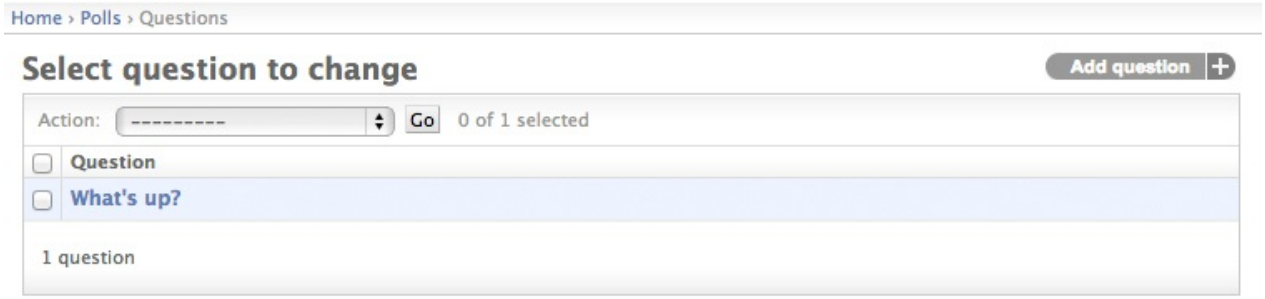
```
from django.contrib import admin
from polls.models import Poll
admin.site.register(Poll)
```

你需要重启开发服务器才能看到变化。通常情况下，你每次修改过一个文件后开发 服务器都会自动载入，但是创建一个新文件却不会触发自动载入的逻辑。

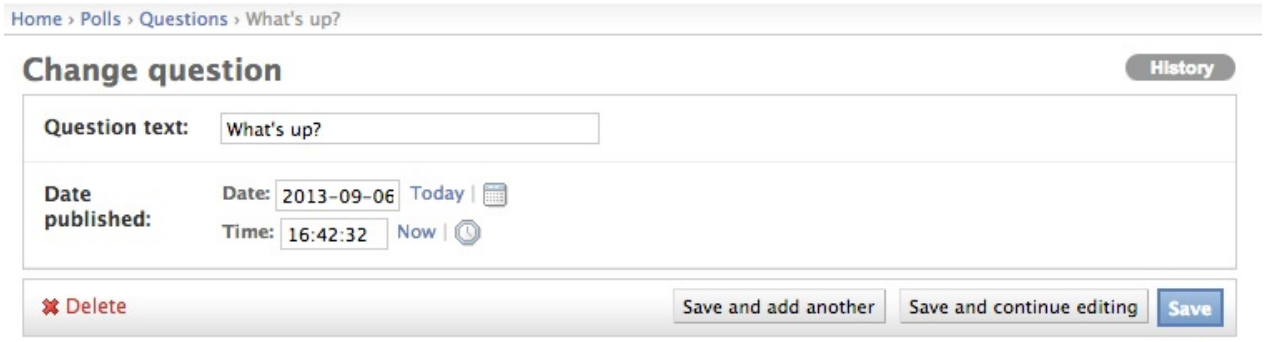
探索管理功能

现在我们已经注册了 Poll，那 Django 就知道了要在管理网站的首页上显示出来：

点击 “Polls” 。现在你在 polls 的 “更改列表” 页。该页 显示了数据库中所有的 polls 可让你选中一个进行编辑。 有个 “What’s up?” poll 是我们在第一个教程中创建的：



点击这个”What’s up?” 的 poll 进行编辑：



这有些注意事项：

- 这的表单是根据 Poll 模型自动生成的。
- 不同模型的字段类型 (DateTimeField, CharField) 会对应的相应的 HTML 输入控件。 每一种类型的字段 Djaong 管理网站都知道如何显示它们。
- 每个 DateTimeField 都会有个方便的 JavaScript 快捷方式。日期有一个 “Today” 快捷方式和弹出式日历，而时间有个 “Now” 快捷方式和一个列出了常用时间选项的弹出式窗口。

在页面的底部还为你提供了几个选项：

- Save – 保存更改并返回到当前类型的对象的更改列表页面。
- Save and continue editing – 保存更改并重新载入当前对象的管理界面。
- Save and add another – 保存更改并载入当前对象类型的新的空白表单。
- Delete – 显示删除确认页。

如果 “Date published” 的值与你在第一部分教程时创建的 poll 的时间不符，这可能 意味着你忘记了将 TIME_ZONE 设置成正确的值了。修改正确后再重启载入页面 来检查值是否正确。

分别点击 “Today” 和 “Now” 快捷方式来修改 “Date published” 的值。 然后点击 “Save and continue editing” 。最后点击右上角的 “History” 。 你将看到一页列出了通过 Django 管理界面对此对象所做的全部更改的清单的页面， 包含有时间戳和修改人的姓名等信息：

自定义管理表单

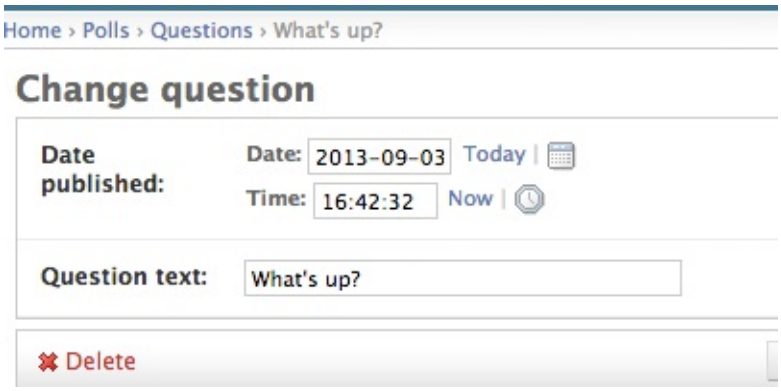
花些时间感叹一下吧，你没写什么代码就拥有了这一切。通过 admin.site.register(Poll) 注册了 Poll 模型，Django 就能构造一个默认的 表单。通常情况下，你将要自定义管理表单的外观和功能。这样的话你就需要在注册对象 时告诉 Django 对应的配置。

让我们来看看如何在编辑表单上给字段重新排序。将 admin.site.register(Poll) 这行替换成：:

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

你将遵循这个模式 – 创建一个模型的管理对象，将它作为 admin.site.register() 方法的第二个参数传入 – 当你需要为一个对象做管理界面配置的时候。

上面那特定的更改使得 “Publication date” 字段在 “Question” 字段之前:

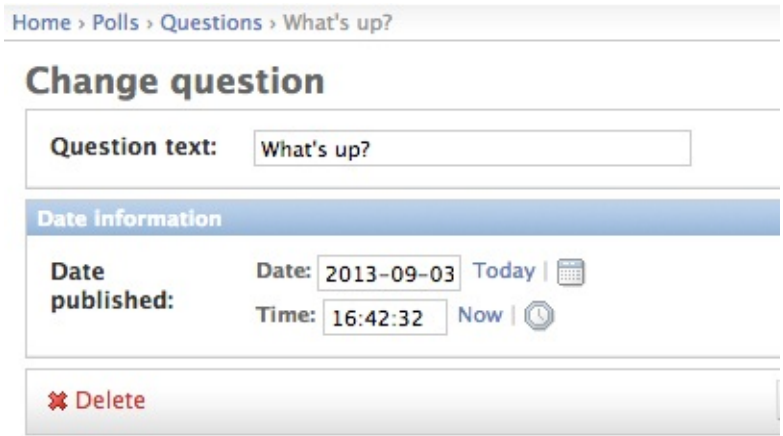


仅有两个字段不会令你印象深刻，但是对于有许多字段的管理表单时，选择一个直观 的排序方式是一个重要的实用细节。

刚才所说的有许多字段的表单，你可能想将表单中的字段分割成 fieldsets ：:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Poll, PollAdmin)
```

在 fieldsets 中每一个 tuple 的第一个元素就是 fieldset 的标题。 下面是我们表单现在的样子：



你可以为每个 fieldset 指定 THML 样式类。Django 提供了一个 "collapse" 样式类用于显示初始时是收缩的 fieldset 。 当你有一个包含一些不常用的长窗体时这是非常有用的

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
Fieldset is initially collapsed
```

添加关联对象

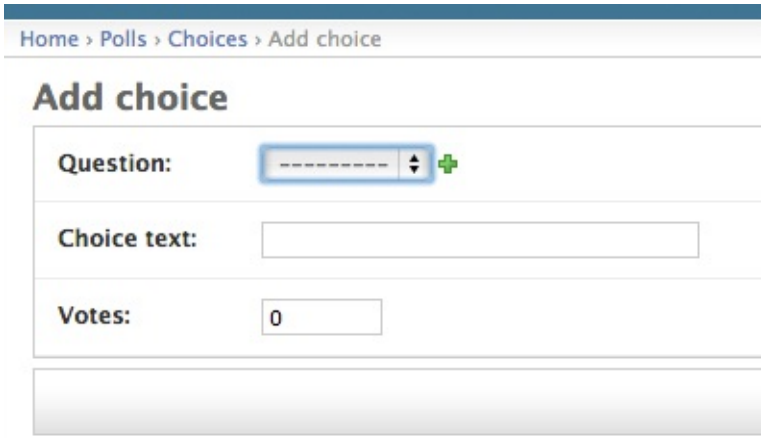
Ok，现在我们有了 Poll 的管理页面。但是一个 Poll 拥有多个 Choices ，而该管理页面并没有显示对应的 choices 。

是的。

我们有两种方法来解决这个问题。第一种就像刚才 Poll 那样在管理网站上 注册 Choice 。这很简单：

```
from polls.models import Choice

admin.site.register(Choice)
```

现在 “Choices” 在 Django 管理网站上是一个可用的选项了。”Add choice” 表单 看起来像这样：



该表单中， `Poll` 字段是一个包含了数据库中每个 poll 的选择框。 Django 知道 ForeignKey 在管理网站中以 `<select>` 框显示。在本例中，选择框中仅存在一个 poll 。

另外请注意 Poll 旁边的 “Add Another” 链接。每个有 ForeignKey 的对象关联到其他对象都会得到这个链接。 当点击 “Add Another” 时，你将会获得一个 “Add poll” 表单的弹出窗口。 如果你在窗口中添加了一 poll 并点击了 “Save” 按钮， Django 会将 poll 保存至数据库中并且动态的添加为你正在查看的 “Add choice” 表单中的 已选择项。

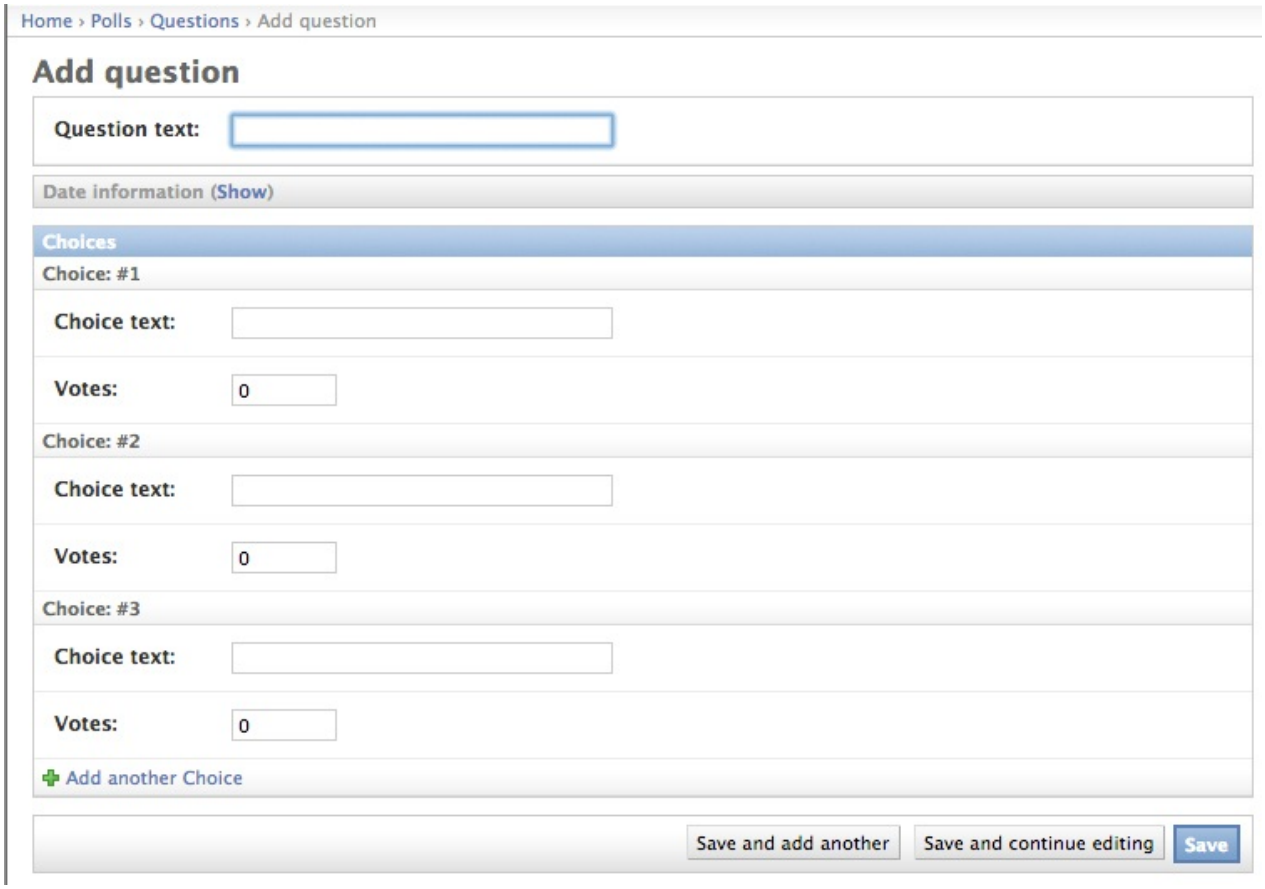
但是，这真是一个低效的将 Choice 对象添加进系统的方式。 如果在创建 Poll 对象时能够直接添加一批 Choices 那会更好。 让我们这样做吧。

移除对 Choice 模型的 register() 方法调用 。然后，将 Poll 的注册代码 编辑为如下所示：

```
from django.contrib import admin
from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

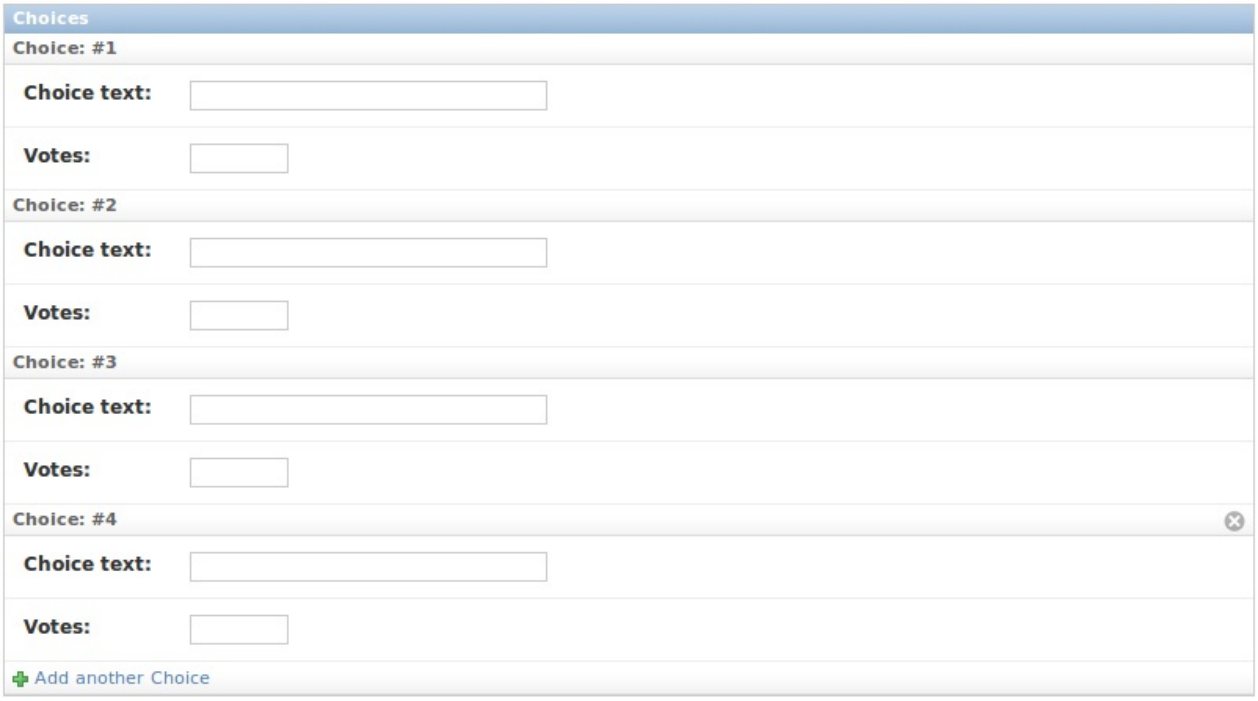
这将告诉 Django: “Choice 对象在 Poll 管理页面中被编辑。 默认情况下，提供 3 个 choices 的字段空间。

载入 “Add poll” 页面来看看，你可能需要重启你的开发服务器：



它看起来像这样：多了三个为关联 Choices 提供的输入插槽 – 由 extra 指定 – 并且每次你在 “Change” 页修改已经创建的对象时，都会另外获得三个额外插槽。

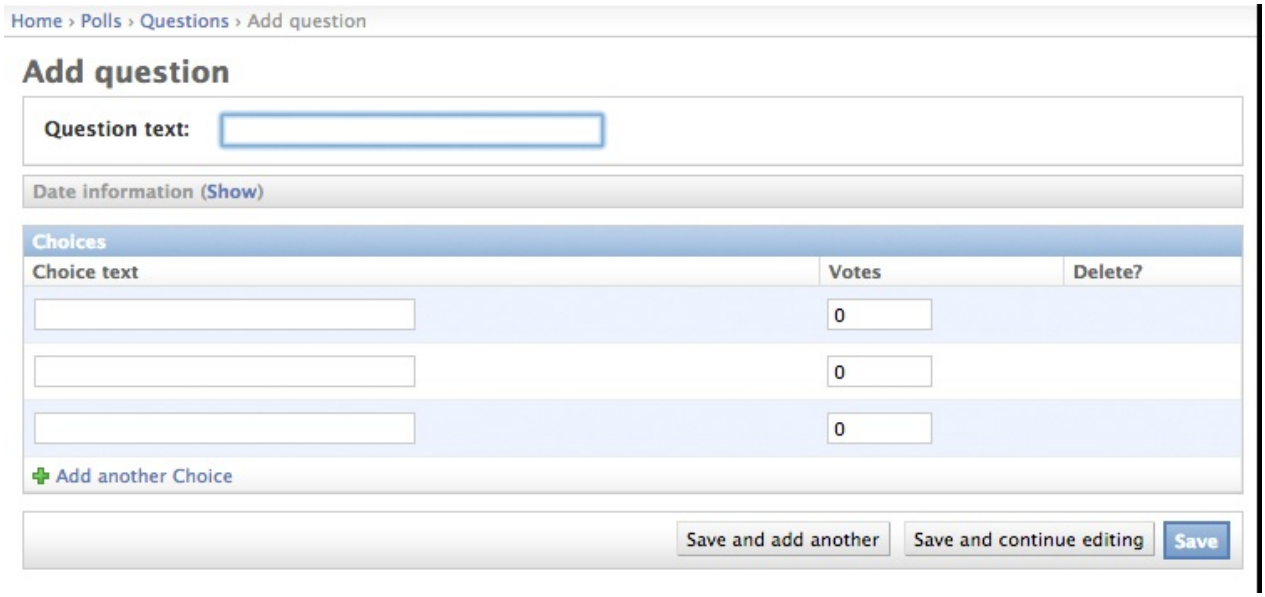
在现有的三个插槽的底部，你会发现一个 “Add another Choice” 链接。 如果你点击它，一个新的插槽会被添加。如果想移除添加的插槽， 你可以点击所添加的插槽的右上方的 X 。注意你不能移除原有的三个插槽。 此图片中显示了新增的插槽：



还有个小问题。为了显示所有关联 Choice 对象的字段需要占用大量的 屏幕空间。为此，Django 提供了一个以表格方式显示内嵌有关联对象的方式； 你只需要将 ChoiceInline 声明改为如下所示：

```
class ChoiceInline(admin.TabularInline):
    #...
```

使用了 TabularInline 后(而不是 StackedInline)， 基于表的格式下相关 对象被显示的更紧凑了：

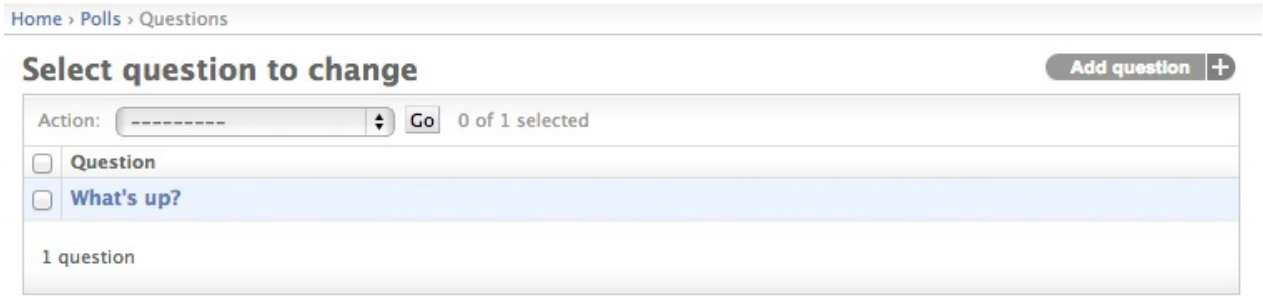


需要注意的是有个额外的 “Delete?” 列允许保存时移除已保存过的行。

自定义管理界面的变更列表

现在 Poll 的管理界面看起来不错了，让我们给 “chang list” 页面做些调整 – 显示系统中所有 polls 的页面。

下面是现在的样子：



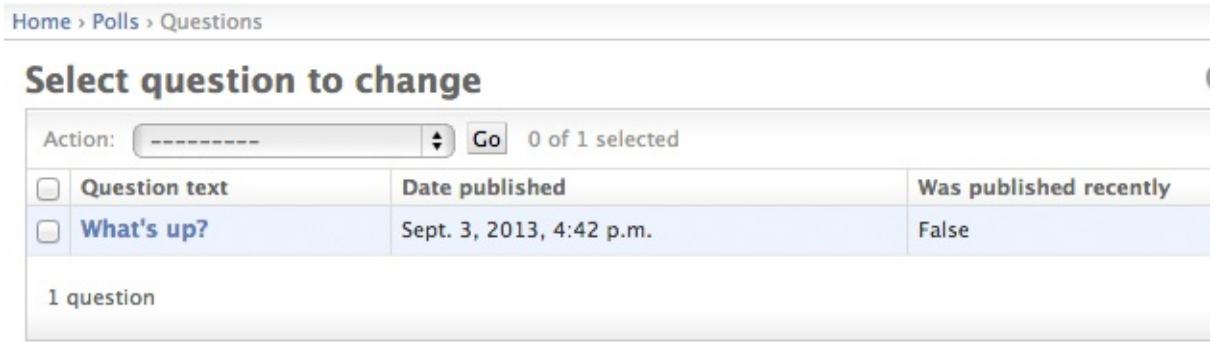
默认情况下， Django 显示的是每个对象 str() 的结果。但是若是我们能够显示每个字段的话有时会更有帮助的。要做到这一点，需要使用 list_display 管理选项，这是一个 tuple ，包含了要显示的字段名， 将会以列的形式在该对象的 chang lsit 页上列出来：:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

效果再好的点话，让我们把在第一部分教程中自定义的方法 was_published_recently 也包括进来:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_recently')
```

现在 poll 的变更列表页看起来像这样：



你可以点击列的标题对这些值进行排序 – 除了 was_published_recently 这一列，因为不支持根据方法输出的内容的排序。还要注意的是默认情况下列的标题是 was_published_recently ，就是方法名（将下划线替换为空格），并且每一行以字符串形式输出。

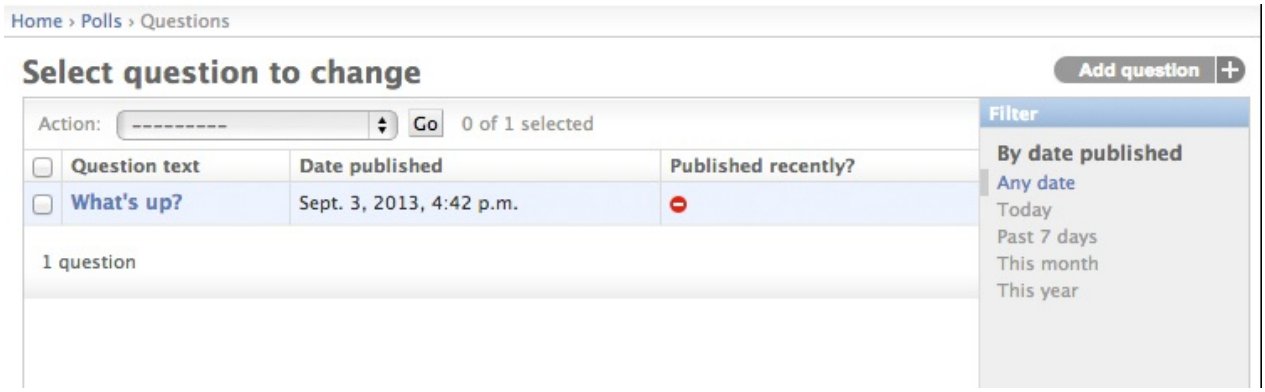
你可以通过给该方法 （在 models.py 内 ） 添加一些属性来改善显示效果，如下所示：:

```
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

再次编辑你的 admin.py 文件并添加一个改进 Poll 的 change list 页面效果的功能： 筛选 (Filters) 。在 PollAdmin 内添加一行如下所示的代码：:

```
list_filter = ['pub_date']
```

这就增加了一个 “筛选” 的侧边栏，让人们通过 pub_date 字段的值来筛选 change list 显示的内容：



显示筛选的类型取决于你需要筛选的字段类型。 因为 pub_date 是一个 DateTimeField 的实例，Django 知道提供对应的筛选选项：”Any date,” “Today,” “Past 7 days,” “This month,” “This year.”

为了效果更好。让我们来加上搜索功能：:

```
search_fields = ['question']
```

在 chang list 页的顶部增加了一个搜索框。当有人输入了搜索条件， Django 将搜索 question 字段。 虽然你可以使用任意数量的字段，如你希望的那样 – 但是因为它在后台用 LIKE 查询，为了保持数据库的性能请合理使用。

最后，因为 Poll 对象有日期字段，根据日期来向下钻取记录将会很方便。 添加下面这一行代码：:

```
date_hierarchy = 'pub_date'
```

这会在 change list 页的顶部增加了基于日期的分层导航功能。 在最顶层，显示所有可用年份。然后可钻取到月份，最终到天。

现在又是一个好时机，请注意 change lists 页面提供了分页功能。默认情况下每一页显示 100 条记录。 Change-list 分页，搜索框，筛选，日期分层和列标题排序如你所原地在一起运行了。

自定义管理界面的外观

显而易见，在每一个管理页面顶部有 “Django administration” 是无语的。虽然它仅仅是个占位符。

不过使用 Django 的模板系统是很容易改变的。Django 管理网站有 Django 框架自身的功能，可以通过 Django 自身的模板系统来修改界面。

自定义你的 项目 模板

在你的项目目录下创建一个 templates 目录。模板可以放在你的文件系统的任何地方，Diango 都能访问。 (Django 能以任何用户身份在你的服务器上运行。) 然后，在你的项目中保存模板是一个好习惯。

默认情况下，TEMPLATE_DIRS 值是空的。因此，让我们添加一行代码，来告诉 Django 我们的模板在哪里：:

```
TEMPLATE_DIRS = (
    '/path/to/mysite/templates', # 将此处改为你的目录。
)
```

现在从 Django 源代码中自带的默认 Django 管理模板的目录 (django/contrib/admin/templates) 下复制 admin/base_site.html 模板到你正在使用的 TEMPLATE_DIRS 中任何目录的子目录 admin 下。例如：如果你的 TEMPLATE_DIRS 中包含 '/path/to/mysite/templates' 目录， 如上所述，复制 django/contrib/admin/templates/admin/base_site.html 模板到 /path/to/mysite/templates/admin/base_site.html 。不要忘了是 admin 子目录。

> Django 的源代码在哪里？
>
> 如果在你的文件系统中很难找到 Django 源代码，可以运行如下命令：
>
> ```
> python -c "
> import sys
> sys.path = sys.path[1:]
> import django
> print(django.__path__)"
> ```

然后，只需要编辑该文件并将通用的 Djangot 文字替换为你认为适合的属于你自己的网站名。

该模板包含了大量的文字，比如 `{% block branding %}` 和 `{{ title }}`。 `{%` 和 `{{` 标记是 Django 模板语言的一部分。 当 Django 呈现 admin/base_site.html 时，根据模板语言生成最终的 HTML 页面。 Don't worry if you can't make any sense of the template right now – 如果你现在不能理解模板的含义先不用担心 – 我们将在教程 3 中深入探讨 Django' 的模板语言。

请注意 Django 默认的管理网站中的任何模板都是可覆盖的。 要覆盖一个模板，只需要像刚才处理 base_site.html 一样 – 从默认的目录下复制到你的自定义目录下，并修改它。

自定义你的 应用 模板

细心的读者会问：如果 TEMPLATE_DIRS 默认的情况下是空值， 那 Django 是如何找到默认的管理网站的模板的？ 答案就是在默认情况下， Django 会自动在每一个应用的包内查找 templates/ 目录，作为备用使用。 （不要忘记 django.contrib.admin 是一个应用）。

我们的 poll 应用不是很复杂并不需要自定义管理模板。但是如果它变得更复杂 而且为了一些功能需要修改 Django 的标准管理模板，修改应用模板将是更 明智的选择，而不是修改项目模板。通过这种方式，你可以在任何新项目包括 polls 应用中自定义模板并且放心会找到需要的自定义的模板的。

有关 Django 怎样找到它的模板的更多信息，请参考 模板加载文档 。

自定义管理网站的首页

于此类似，你可能还想自定义 Django 管理网站的首页。

默认情况下，首页会显示在 INSTALLED_APPS 中所有注册了管理功能的应用， 并按字母排序。你可能想在页面布局上做大修改。总之，首页可能是管理网站中最重要的页面， 因此它应该很容易使用。

你需要自定义的模板是 admin/index.html 。 （同先前处理 admin/base_site.html 一样 – 从默认目录下复制到你自定义的模板目录下。） 编辑这个文件，你将看到一个名为 app_list 的模板变量。这个变量包含了每一个 已安装的 Django 应用。你可以通过你认为最好的方法硬编码链接到特定对象的管理页面，而不是使用默认模板。 再次强调，如果你不能理解模板语言的话不用担心 – 我们将在教程 3 中详细介绍。

当你熟悉了管理网站的功能后，阅读 教程 第3部分 开始开发公共 poll 界面。

> 译者：Django 文档协作翻译小组，原文：Part 2: The admin site。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第3部分

本教程上接 教程 第2部分 。我们将继续 开发 Web-poll 应用并且专注在创建公共界面 – “视图（views ）”。

哲理

在 Django 应用程序中，视图是一“类”具有特定功能和模板的网页。 例如，在一个博客应用程序中，你可能会有以下视图：

- 博客首页 – 显示最新发表的博客。
- 博客详细页面 – 一篇博客的独立页面。
- 基于年份的归档页 – 显示给定年份中发表博客的所有月份。
- 基于月份的归档页 – 显示给定月份中发表博客的所有日期。
- 基于日期的归档页 – 显示给定日期中发表的所有的博客。
- 评论功能 – 为一篇给定博客发表评论。

在我们的 poll 应用程序中，将有以下四个视图：

- Poll “index” 页 – 显示最新发布的民意调查。
- Poll “detail” 页 – 显示一项民意调查的具体问题，不显示该项的投票结果但可以进行投票的 form 。
- Poll “results” 页 – 显示一项给定的民意调查的投票结果。
- 投票功能 – 为一项给定的民意调查处理投票选项。

在 Django 中，网页及其他内容是由视图来展现的。而每个视图就是一个简单的 Python 函数（或方法， 对于基于类的视图情况下）。Django 会通过检查所请求的 URL （确切地说是域名之后的那部分 URL）来匹配一个视图。

平时你上网的时候可能会遇到像 “ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B” 这种如此美丽的 URL。 但是你会很高兴知道 Django 允许我们使用比那优雅的 URL 模式 来展现 URL。

URL 模式就是一个简单的一般形式的 URL - 比如: `/newsarchive/<year>/<month>/` .

Django 是通过 ‘URLconfs’ 从 URL 获取到视图的。而 URLconf 是将 URL 模式 (由正则表达式来描述的) 映射到视图的一种配置。

本教程中介绍了使用 URLconfs 的基本指令，你可以查阅 django.core.urlresolvers 来获取更多信息。

编写你的第一个视图

让我们编写第一个视图。打开文件 polls/views.py 并在其中输入以下 Python 代码

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the poll index.")
```

在 Django 中这可能是最简单的视图了。为了调用这个视图，我们需要将它映射到一个 URL – 为此我们需要配置一个URLconf 。

在 polls 目录下创建一个名为 urls.py 的 URLconf 文档。 你的应用目录现在看起来像这样

```
polls/
    __init__.py
    admin.py
    models.py
    tests.py
    urls.py
    views.py
```

在 polls/urls.py 文件中输入以下代码：

```
from django.conf.urls import patterns, url

from polls import views

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

下一步是将 polls.urls 模块指向 root URLconf 。在 mysite/urls.py 中插入一个 include() 方法，最后的样子如下所示

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

现在你在 URLconf 中配置了 index 视图。通过浏览器访问 http://localhost:8000/polls/ ，如同你在 index 视图中定义的一样，你将看到 “Hello, world. You’re at the poll index.” 文字。

url() 函数有四个参数，两个必须的： regex 和 `view` ， 两个可选的： `kwargs` ， 以及 `name` 。 接下来，来探讨下这些参数的意义。

url() 参数: regex

regex 是 regular expression 的简写，这是字符串中的模式匹配的一种语法， 在 Django 中就是是 url 匹配模式。 Django 将请求的 URL 从上至下依次匹配列表中的正则表达式，直到匹配到一个为止。

需要注意的是，这些正则表达式不会匹配 GET 和 POST 参数，以及域名。 例如：针对 http://www.example.com/myapp/ 这一请求，URLconf 将只查找 myapp/ 。而在 http://www.example.com/myapp/?page=3 中 URLconf 也仅查找 myapp/ 。

如果你需要正则表达式方面的帮助，请参阅 Wikipedia's entry 和本文档中的 re 模块。 此外，O'Reilly 出版的由 Jeffrey Friedl 著的 "Mastering Regular Expressions" 也是不错的。 但是，实际上，你并不需要成为一个正则表达式的专家，仅仅需要知道如何捕获简单的模式。 事实上，复杂的正则表达式会降低查找性能，因此你不能完全依赖正则表达式的功能。

最后有个性能上的提示：这些正则表达式在 URLconf 模块第一次加载时会被编译。 因此它们速度超快 (像上面提到的那样只要查找的不是太复杂)。

url() 参数： view

当 Django 匹配了一个正则表达式就会调用指定的视图功能，包含一个 HttpRequest 实例作为第一个参数和正则表达式 "捕获" 的一些值的作为其他参数。 如果使用简单的正则捕获，将按顺序位置传参数；如果按命名的正则捕获，将按关键字传参数值。 有关这一点我们会给出一个例子。

url() 参数： kwargs

任意关键字参数可传一个字典至目标视图。在本教程中，我们并不打算使用 Django 这一特性。

url() 参数： name

命名你的 URL，让你在 Django 的其他地方明确地引用它，特别是在模板中。 这一强大的功能可允许你通过一个文件就可全局修改项目中的 URL 模式。

编写更多视图

现在让我们添加一些视图到 polls/views.py 中去。这些视图与之前的略有不同，因为 它们有一个参数：:

```
def detail(request, poll_id):
    return HttpResponse("You're looking at poll %s." % poll_id)

def results(request, poll_id):
    return HttpResponse("You're looking at the results of poll %s." % poll_id)

def vote(request, poll_id):
    return HttpResponse("You're voting on poll %s." % poll_id)
```

将新视图按如下所示的 url() 方法添加到 polls.urls 模块中去：:

```
from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<poll_id>\d+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
)
```

在你的浏览器中访问 http://localhost:8000/polls/34/ 。将运行 detail() 方法并且显示你在 URL 中提供的任意 ID 。试着访问 http://localhost:8000/polls/34/results/ 和 http://localhost:8000/polls/34/vote/ – 将会显示对应的结果页及投票页。

当有人访问你的网站页面如 “ /polls/34/ ” 时，Django 会加载 mysite.urls 模块，这是因为 ROOT_URLCONF 设置指向它。接着在该模块中寻找名为 `urlpatterns` 的变量并依次匹配其中的正则表达式。 include() 可让我们便利地引用其他 URLconfs 。请注意 include() 中的正则表达式没有 $ (字符串结尾的匹配符 match character) 而尾部是一个反斜杠。当 Django 解析 include() 时，它截取匹配的 URL 那部分而把剩余的字符串交由 加载进来的 URLconf 作进一步处理。

include() 背后隐藏的想法是使 URLs 即插即用。 由于 polls 在自己的 URLconf(polls/urls.py) 中，因此它们可以被放置在 “/polls/” 路径下，或 “/fun_polls/” 路径下，或 “/content/polls/” 路径下，或者其他根路径，而应用仍可以运行。

以下是当用户访问 “/polls/34/” 路径时系统中将发生的事：

- Django 将寻找 '^polls/' 的匹配
- 接着，Django 截取匹配文本 ("polls/") 后剩余的文本 – "34/" – 传递到 ‘polls.urls’ URLconf 中作进一步处理， 再将匹配 r'^(?P\d+)/$' 的结果作为参数传给 detail() 视图

```
detail(request=<HttpRequest object>, poll_id='34')
```

poll_id='34' 这部分就是来自 (?P\d+) 匹配的结果。 使用括号包围一个 正则表达式所“捕获”的文本可作为一个参数传给视图函数； `?P<poll_id>` 将会定义名称用于标识匹配的内容； 而 \d+ 是一个用于匹配数字序列（即一个数字）的正则表达式。

因为 URL 模式是正则表达式，所以你可以毫无限制地使用它们。但是不要加上 URL 多余的部分如 .html – 除非你想，那你可以像下面这样：:

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

真的，不要这样做。这很傻。

在视图中添加些实际的功能

每个视图只负责以下两件事中的一件：返回一个 HttpResponse 对象，其中包含了所请求页面的内容， 或者抛出一个异常，例如 Http404 。剩下的就由你来实现了。

你的视图可以读取数据库记录，或者不用。它可以使用一个模板系统，例如 Django 的 – 或者第三方的 Python 模板系统 – 或不用。它可以生成一个 PDF 文件，输出 XML， 即时创建 ZIP 文件， 你可以使用你想用的任何 Python 库来做你想做的任何事。

而 Django 只要求是一个 HttpResponse 或一个异常。

因为它很方便，那让我们来使用 Django 自己的数据库 API 吧， 在 教程 第1部分 中提过。修改下 index() 视图， 让它显示系统中最新发布的 5 个调查问题，以逗号分割并按发布日期排序：:

```
from django.http import HttpResponse

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

在这就有了个问题， 页面的设计是硬编码在视图中的。如果你想改变页面的外观，就必须修改这里的 Python 代码。因此，让我们使用 Django 的模板系统创建一个模板给视图用，就使页面设计从 Python 代码中 分离出来了。

首先，在 polls 目录下创建一个 templates 目录。 Django 将会在那寻找模板。

Django 的 TEMPLATE_LOADERS 配置中包含一个知道如何从各种来源导入模板的可调用的方法列表。 其中有一个默认值是 django.template.loaders.app_directories.Loader ， Django 就会在每个 INSTALLED_APPS 的 “templates” 子目录下查找模板 - 这就是 Django 知道怎么找到 polls 模板的原因，即使我们 没有修改 TEMPLATE_DIRS, 还是如同在 教程 第2部分 那样。

> 组织模板
>
> 我们 能够 在一个大的模板目录下一起共用我们所有的模板，而且它们会运行得很好。但是，此模板属于 polls 应用，因此与我们在上一个教程中创建的管理模板不同， 我们要把这个模板放在应用的模板目录 (polls/templates) 下而不是项目的模板目录 (templates)。 我们将在 可重用的应用教程 中详细讨论我们 为什么 要这样做。

在你刚才创建的 `templates` 目录下，另外创建个名为 polls 的目录，并在其中创建一个 index.html 文件。换句话说，你的模板应该是 polls/templates/polls/index.html 。由于知道如上所述的 app_directories 模板加载器是 如何运行的，你可以参考 Django 内的模板简单的作为 polls/index.html 模板。

> 模板命名空间
>
> 现在我们 也许 能够直接把我们的模板放在 polls/templates 目录下 (而不是另外创建 polls 子目录)， 但它实际上是一个坏注意。 Django 将会选择第一个找到的按名称匹配的模板， 如果你在 不同 应用中有相同的名称的模板，Django 将无法区分它们。 我们想要让 Django 指向正确的模板，最简单的方法是通过 命名空间 来确保是 他们的模板。也就是说，将模板放在 另一个 目录下并命名为应用本身的名称。

将以下代码添加到该模板中:

```
{% if latest_poll_list %}
    <ul>
    {% for poll in latest_poll_list %}
        <li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

现在让我们在 index 视图中使用这个模板：

```
from django.http import HttpResponse
from django.template import Context, loader

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(template.render(context))
```

代码将加载 polls/index.html 模板并传递一个 context 变量。 The context is a dictionary mapping template variable names to Python 该 context 变量是一个映射了 Python 对象到模板变量的字典。

在你的浏览器中加载 “/polls/” 页，你应该看到一个列表，包含了在教程 第1部分 中创建的 “What’s up” 调查。而链接指向 poll 的详细页面。

快捷方式: render()

这是一个非常常见的习惯用语，用于加载模板，填充上下文并返回一个含有模板渲染结果的 HttpResponse 对象。 Django 提供了一种快捷方式。这里重写完整的 index() 视图

```
from django.shortcuts import render

from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    context = {'latest_poll_list': latest_poll_list}
    return render(request, 'polls/index.html', context)
```

请注意，一旦我们在所有视图中都这样做了，我们就不再需要导入 loader， Context 和 HttpResponse (如果你仍然保留了 detail, `resutls` , 和 `vote` 方法，你还是需要保留 HttpResponse) 。

render() 函数中第一个参数是 request 对象，第二个参数是一个模板名称，第三个是一个字典类型的可选参数。 它将返回一个包含有给定模板根据给定的上下文渲染结果的 HttpResponse 对象。

抛出 404 异常

现在让我们解决 poll 的详细视图 – 该页显示一个给定 poll 的详细问题。 视图代码如下所示：:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        poll = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render(request, 'polls/detail.html', {'poll': poll})
```

在这有个新概念：如果请求的 poll 的 ID 不存在，该视图将抛出 Http404 异常。

我们稍后讨论如何设置 polls/detail.html 模板，若是你想快速运行上面的例子， 在模板文件中添加如下代码：

```
{{ poll }}
```

现在你可以运行了。

快捷方式: get_object_or_404()

这很常见，当你使用 get() 获取对象时 对象却不存在时就会抛出 Http404 异常。对此 Django 提供了一个快捷操作。如下所示重写 detail() 视图：

```
from django.shortcuts import render, get_object_or_404
# ...
def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/detail.html', {'poll': poll})
```

get_object_or_404() 函数需要一个 Django 模型类作为第一个参数以及 一些关键字参数，它将这些参数传递给模型管理器中的 get() 函数。 若对象不存在时就抛出 Http404 异常。

> 哲理
>
> 为什么我们要使用一个 get_object_or_404() 辅助函数 而不是在更高级别自动捕获 ObjectDoesNotExist 异常， 或者由模型 API 抛出 Http404 异常而不是 ObjectDoesNotExist 异常？
>
> 因为那样会使模型层与视图层耦合在一起。Django 最重要的设计目标之一 就是保持松耦合。一些控制耦合在 django.shortcuts 模块中介绍。

还有个 get_list_or_404() 函数，与 get_object_or_404() 一样 – 不过执行的是 filter() 而不是 get() 。若返回的是空列表将抛出 Http404 异常。

编写一个 404 (页面未找到) 视图

当你在视图中抛出 Http404 时，Django 将载入一个特定的视图来处理 404 错误。Django 会根据你的 root URLconf (仅在你的 root URLconf 中；在其他任何地方设置 handler404 都无效) 中设置的 handler404 变量来查找该视图，这个变量是个 Python 包格式字符串 – 和标准 URLconf 中的回调函数格式是一样的。 404 视图本身没有什么特殊性：它就是一个普通的视图。

通常你不必费心去编写 404 视图。若你没有设置 handler404 变量，默认情况下会使用内置的 django.views.defaults.page_not_found() 视图。或者你可以在你的模板目录下的根目录中 创建一个 404.html 模板。当 DEBUG 值是 False (在你的 settings 模块中) 时， 默认的 404 视图将使用此模板来显示所有的 404 错误。 如果你创建了这个模板，至少添加些如“页面未找到” 的内容。

一些有关 404 视图需要注意的事项：

- 如果 DEBUG 设为 True (在你的 settings 模块里) 那么你的 404 视图将永远不会被使用 (因此 404.html 模板也将永远不会被渲染) 因为将要显示的是跟踪信息。
- 当 Django 在 URLconf 中不能找到能匹配的正则表达式时 404 视图也将被调用。 编写一

个 500 (服务器错误) 视图

类似的，你可以在 root URLconf 中定义 handler500 变量，在服务器发生错误时 调用它指向的视图。服务器错误是指视图代码产生的运行时错误。

同样，你在模板根目录下创建一个 500.html 模板并且添加些像“出错了”的内容。

使用模板系统

回到我们 poll 应用的 detail() 视图中，指定 poll 变量后， `polls/detail.html` 模板可能看起来这样：

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

模板系统使用了“变量.属性”的语法访问变量的属性值。 例如 `{{ poll.question }}` ， 首先 Django 对 poll 对象做字典查询。 否则 Django 会尝试属性查询 – 在本例中属性查询成功了。如果属性查询还是失败了，Django 将尝试 list-index 查询。

在 `{% for %}` 循环中有方法调用: poll.choice_set.all 就是 Python 代码 poll.choice_set.all(), 它将返回一组可迭代的 Choice 对象，可以用在 `{% for %}` 标签中。

请参阅 模板指南 来了解模板的更多内容。

移除模板中硬编码的 URLS

记得吗? 在 polls/index.html 模板中，我们链接到 poll 的链接是硬编码成这样子的：

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

问题出在硬编码， 紧耦合使得在大量的模板中修改 URLs 成为富有挑战性的项目。 不过，既然你在 polls.urls 模块中的 url() 函数中定义了 命名参数，那么就可以在 url 配置中使用 `{% url %}` 模板标记来移除特定的 URL 路径依赖:

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

> Note
>
> 如果 `{% url 'detail' poll.id %}` (含引号) 不能运行，但是 `{% url detail poll.id %}` (不含引号) 却能运行，那么意味着你使用的 Djang 低于 < 1.5 版。这样的话，你需要在模板文件的顶部添加如下的声明：:
>
> ```
> {% load url from future %}
> ```

其原理就是在 polls.urls 模块中寻找指定的 URL 定义。 你知道命名为 'detail' 的 URL 就如下所示那样定义的一样：:

```
...
# 'name' 的值由 {% url %} 模板标记来引用
url(r'^(?P<poll_id>\d+)/$', views.detail, name='detail'),
...
```

如果你想将 polls 的 detail 视图的 URL 改成其他样子，或许像 polls/specifics/12/ 这样子，那就不需要在模板（或者模板集）中修改而只要在 polls/urls.py 修改就行了:

```
...
# 新增 'specifics'
url(r'^specifics/(?P<poll_id>\d+)/$', views.detail, name='detail'),
...
```

URL 名称的命名空间

本教程中的项目只有一个应用： `polls` 。在实际的 Django 项目中，可能有 5、10、20 或者更多的应用。Django 是如何区分它们的 URL 名称的呢？比如说， `polls` 应用有一个 detail 视图，而可能会在同一个项目中是一个博客应用的视图。Django 是如何知道 使用 `{% url %}` 模板标记创建应用的 url 时选择正确呢？

答案是在你的 root URLconf 配置中添加命名空间。在 mysite/urls.py 文件 (项目的 `urls.py` ，不是应用的) 中，修改为包含命名空间的定义:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls', namespace="polls")),
    url(r'^admin/', include(admin.site.urls)),
)
```

现在将你的 polls/index.html 模板中原来的 detail 视图:

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

修改为包含命名空间的 detail 视图:

```
<li><a href="{% url 'polls:detail' poll.id %}">{{ poll.question }}</a></li>
```

当你编写视图熟练后，请阅读 教程 第4部分 来学习如何处理简单的表单和通用视图。

> 译者：Django 文档协作翻译小组，原文：Part 3: Views and templates。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个 Django 程序 第4部分

本教程上接 教程 第3部分 。我们将 继续开发 Web-poll 应用并且关注在处理简单的窗体和优化我们的代码。

编写一个简单的窗体

让我们把在上一篇教程中编写的 poll 的 detail 模板更新下，在模板中包含 HTML 的

组件:

```
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' poll.id %}" method="post">
{% csrf_token %}
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.i
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

简单的总结下:

- 上面的模板中为每个投票选项设置了一个单选按钮。每个单选按钮的 value 是投票选项对应的 ID 。每个单选按钮的 name 都是 `"choice"` 。这意味着，当有人选择了一个单选按钮并提交了表单，将会发送 的 POST 数据是 `choice=3` 。这是 HTML 表单中的基本概念。
- 我们将 form 的 action 设置为 `{% url 'polls:vote' poll.id %}`，以及设置了 method="post"` 。使用 method="post" (而不是 method="get") 是非常重要的，因为这种提交表单的方式会改变服务器端的数据。 当你创建一个表单为了修改服务器端的数据时，请使用 method="post" 。这不是 Django 特定的技巧；这是优秀的 Web 开发实践。
- forloop.counter 表示 for 标签在循环中已经循环过的次数
- 由于我们要创建一个POST form (具有修改数据的功能)，我们需要担心跨站点请求伪造（ Cross Site Request Forgeries ）。 值得庆幸的是，你不必太担心这一点，因为 Django 自带了一个非常容易使用的系统来防御它。 总之，所有的 POST form 针对内部的 URLs 时都应该使用 `{% 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。 记得吗？在 教程 第3部分 中，我们为 polls 应用创建了一个 URLconf 配置中包含有这一行代码:

```
url(r'^(?P<poll_id>\d+)/vote/$', views.vote, name='vote'),
```

我们还创建了一个虚拟实现的 vote() 函数。让我们创建一个真实版本吧。在 polls/views.py 中添加如下代码:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from polls.models import Choice, Poll
# ...
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render(request, 'polls/detail.html', {
            'poll': p,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(p.id,)))
```

在这代码中有些内容还未在本教程中提到过:

request.POST 是一个类似字典的对象，可以让你 通过关键字名称来获取提交的数据。在本例中， request.POST['choice'] 返回了所选择的投票项目的 ID ，以字符串的形式。 request.POST 的值永远是字符串形式的。

请注意 Django 也同样的提供了通过 request.GET 获取 GET 数据的方法 – 但是在代码中我们明确的使用了 request.POST 方法，以确保数据是通过 POST 方法来修改的。

如果 choice 未在 POST 数据中提供 request.POST['choice'] 将抛出 KeyError 当未给定 choice 对象时上面的代码若检测到抛出的是 KeyError 异常就会向 poll 显示一条错误信息。

在增加了投票选项的统计数后，代码返回一个 HttpResponseRedirect 对象而不是常见的 HttpResponse 对象。 HttpResponseRedirect 对象需要一个参数：用户将被重定向的 URL (请继续看下去在这情况下我们是如何构造 URL) 。

就像上面用 Python 作的注释那样，当成功的处理了 POST 数据后你应该总是返回一个 HttpResponseRedirect 对象。 这个技巧不是特定于 Django 的；它是优秀的 Web 开发实践。

在本例中，我们在 HttpResponseRedirect 的构造方法中使用了 reverse() 函数。 此函数有助于避免在视图中硬编码 URL 的功能。它指定了我们想要的跳转的视图函数名以及视图函数中 URL 模式相应的可变参数。在本例中，我们使用了教程 第3部分中的 URLconf 配置，reverse() 将会返回类似如下所示的字符串

```
'/polls/3/results/'
```

... 在此 3 就是 p.id 的值。该重定向 URL 会调用 'results' 视图并显示最终页面。

正如在教程 第3部分提到的，`request` 是一个 HttpRequest 对象。想了解 HttpRequest 对象更多的内容，请参阅 request 和 response 文档 。

当有人投票后，`vote()` 视图会重定向到投票结果页。让我们来编写这个视图

```
def results(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render(request, 'polls/results.html', {'poll': poll})
```

这几乎和 教程 第3部分 中的 detail() 视图完全一样。 唯一的区别就是模板名称。 稍后我们会解决这个冗余问题。

现在，创建一个 polls/results.html 模板:

```
<h1>{{ poll.question }}</h1>

<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</
{% endfor %}
</ul>

<a href="{% url 'polls:detail' poll.id %}">Vote again?</a>
```

现在，在浏览器中访问 /polls/1/ 并完成投票。每次投票后你将会看到结果页数据都有更新。如果你没有选择投票选项就提交了，将会看到错误的信息。

使用通用视图：优化代码

detail() (在 教程 第3部分 中) 和 results() 视图 都很简单 – 并且还有上面所提到的冗余问题。`index()` 用于显示 polls 列表的 index() 视图 (也在教程 第3部分中)，也是存在类似的问题。

这些视图代表了基本的 Web 开发中一种常见的问题： 根据 URL 中的参数从数据库中获取数据，加载模板并返回渲染后的内容。由于这类现象很 常见，因此 Django 提供了一种快捷方式，被称之为“通用视图”系统。

通用视图抽象了常见的模式，以至于你不需要编写 Python 代码来编写一个应用。

让我们把 poll 应用修改成使用通用视图系统的应用，这样我们就能删除删除一些我们自己的代码了。 我们将采取以下步骤来进行修改：

- 修改 URLconf 。
- 删除一些旧的，不必要的视图。
- 修正 URL 处理到对应的新视图。

请继续阅读了解详细的信息。

> 为什么要重构代码？
>
> 通常情况下，当你编写一个 Django 应用时，你会评估下通用视图是否适合解决你的问题， 如果适合你就应该从一开始就使用它，而不是进行到一半才重构你的代码。 但是本教程直到现在都故意集中介绍“硬编码”视图，是为了专注于核心概念上。
>
> 就像你在使用计算器前需要知道基本的数学知识一样。

修改 **URLconf**

首先，打开 polls/urls.py 的 URLconf 配置文件并修改成如下所示样子

```
from django.conf.urls import patterns, url
from django.views.generic import DetailView, ListView
from polls.models import Poll

urlpatterns = patterns('',
    url(r'^$',
        ListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='polls/index.html'),
        name='index'),
    url(r'^(?P<pk>\d+)/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/detail.html'),
        name='detail'),
    url(r'^(?P<pk>\d+)/results/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/results.html'),
        name='results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote', name='vote'),
)
```

修改 **views**

在这我们将使用两个通用视图： ListView 和 DetailView 。这两个视图分别用于显示两种抽象概念 “显示一系列对象的列表” 和 “显示一个特定类型的对象的详细信息页”。

- 每个视图都需要知道使用哪个模型数据。因此需要提供将要使用的 model 参数。
- DetailView 通用视图期望从 URL 中捕获名为 "pk" 的主键值，因此我们将 poll_id 改为 pk 。

默认情况下， DetailView 通用视图使用名为 <应用名>/<模型名>_detail.html 的模板。在我们的例子中，将使用名为 "polls/poll_detail.html" 的模板。 template_name 参数是告诉 Django 使用指定的模板名，而不是使用自动生成的默认模板名。 我们也指定了 results 列表视图的

template_name – 这确保了 results 视图和 detail 视图渲染时会有不同的外观，虽然它们有一个 DetailView 隐藏在幕后。

同样的，~django.views.generic.list.ListView 通用视图使用的默认模板名为 <应用名>/<模型名>_list.html ；我们指定了 template_name 参数告诉 ListView 使用已经存在的 "polls/index.html" 模板。

在之前的教程中，模板提供的上下文中包含了 poll 和 latest_poll_list 上下文变量。在 DetailView 中 poll 变量是自动提供的 – 因为我们使用了一个 Django 模型 (Poll)，Django 能够为上下文变量确定适合的名称。另外 ListView 自动生成的上下文变量名是 poll_list 。若要覆盖此变量我们需要提供 context_object_name 选项，我们想要使用 latest_poll_list 来替代它。作为一种替代方式，你可以改变你的模板来匹配新的默认的上下文变量 – 但它是一个非常容易地告诉 Django 使用你想要的变量的方式。

现在你可以在 polls/views.py 中删除 index()， detail() 和 results() 视图了。我们不需要它们了 – 它们已替换为通用视图了。你也可以删除不再需要的 HttpResponse 导入包了。

运行服务器，并且使用下基于通用视图的新投票应用。

有关通用视图的完整详细信息，请参阅 通用视图文档.

当你熟悉了窗体和通用视图后，请阅读 教程 第5部分 来学习测试我们的投票应用。

> 译者：Django 文档协作翻译小组，原文：Part 4: Forms and generic views。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个Django应用，第5部分

本教程上接教程第4部分。 我们已经建立一个网页投票应用，现在我们将为它创建一些自动化测试。

自动化测试简介

什么是自动化测试？

测试是检查你的代码是否正常运行的简单程序。

测试可以划分为不同的级别。 一些测试可能专注于小细节（某一个模型的方法是否会返回预期的值？）， 其他的测试可能会检查软件的整体运行是否正常（用户在对网站进行了一系列的操作后，是否返回了正确的结果？）。这些其实和你早前在教程 1中做的差不多， 使用shell来检测一个方法的行为，或者运行程序并输入数据来检查它的行为方式。

自动化测试的不同之处就在于这些测试会由系统来帮你完成。你创建了一组测试程序，当你修改了你的应用，你就可以用这组测试程序来检查你的代码是否仍然同预期的那样运行，而无需执行耗时的手动测试。

为什么你需要创建测试

那么，为什么要创建测试？而且为什么是现在？

你可能感觉学习Python/Django已经足够，再去学习其他的东西也许需要付出巨大的努力而且没有必要。 毕竟，我们的投票应用已经活蹦乱跳了； 将时间运用在自动化测试上还不如运用在改进我们的应用上。 如果你学习Django就是为了创建一个投票应用，那么创建自动化测试显然没有必要。 但如果不是这样，现在是一个很好的学习机会。

测试将节省你的时间

在某种程度上，‘检查起来似乎正常工作’将是一种令人满意的测试。 在更复杂的应用中，你可能有几十种组件之间的复杂的相互作用。

这些组件的任何一个小的变化，都可能对应用的行为产生意想不到的影响。 检查起来‘似乎正常工作’可能意味着你需要运用二十种不同的测试数据来测试你代码的功能，仅仅是为了确保你没有搞砸某些事 —— 这不是对时间的有效利用。

尤其是当自动化测试只需要数秒就可以完成以上的任务时。 如果出现了错误，测试程序还能够帮助找出引发这个异常行为的代码。

有时候你可能会觉得编写测试程序将你从有价值的、创造性的编程工作里带出，带到了单调乏味、无趣的编写测试中，尤其是当你的代码工作正常时。

然而，比起用几个小时的时间来手动测试你的程序，或者试图找出代码中一个新引入的问题的原因，编写测试程序还是令人惬意的。

测试不仅仅可以发现问题，它们还能防止问题

将测试看做只是开发过程中消极的一面是错误的。

没有测试，应用的目的和意图将会变得相当模糊。 甚至在你查看自己的代码时，也不会发现这些代码真正干了些什么。

测试改变了这一切； 它们使你的代码内部变得明晰，当错误出现后，它们会明确地指出哪部分代码出了问题 —— 甚至你自己都不会料到问题会出现在那里。

测试使你的代码更受欢迎

你可能已经创建了一个堪称辉煌的软件，但是你会发现许多其他的开发者会由于它缺少测试程序而拒绝查看它一眼；没有测试程序，他们不会信任它。 Jacob Kaplan-Moss，Django最初的几个开发者之一，说过“不具有测试程序的代码是设计上的错误。”

你需要开始编写测试的另一个原因就是其他的开发者在他们认真研读你的代码前可能想要查看一下它有没有测试。

测试有助于团队合作

之前的观点是从单个开发人员来维护一个程序这个方向来阐述的。 复杂的应用将会被一个团队来维护。 测试能够减少同事在无意间破坏你的代码的机会（和你在不知情的情况下破坏别人的代码的机会）。 如果你想在团队中做一个好的Django开发者，你必须擅长测试！

基本的测试策略

编写测试有很多种方法。

一些开发者遵循一种叫做“由测试驱动的开发”的规则；他们在编写代码前会先编好测试。 这似乎与直觉不符，尽管这种方法与大多数人经常的做法很相似：人们先描述一个问题，然后创建一些代码来解决这个问题。 由测试驱动的开发可以用Python测试用例将这个问题简单地形式化。

更常见的情况是，刚接触测试的人会先编写一些代码，然后才决定为这些代码创建一些测试。 也许在之前就编写一些测试会好一点，但什么时候开始都不算晚。

有时候很难解决从什么地方开始编写测试。 如果你已经编写了数千行Python代码，挑选它们中的一些来进行测试不会是太容易的。 这种情况下，在下次你对代码进行变更，或者添加一个新功能或者修复一个bug时，编写你的第一个测试，效果会非常好。

现在，让我们马上来编写一个测试。

编写我们的第一个测试

我们找出一个错误

幸运的是，polls应用中有一个小错误让我们可以马上来修复它：如果Question在最后一个天发布，Question.was_published_recently() 方法返回True（这是对的），但是如果Question的pub_date 字段是在未来，它还返回True（这肯定是不对的）。

你可以在管理站点中看到这一点； 创建一个发布时间在未来的一个Question； 你可以看到Question 的变更列表声称它是最近发布的。

你还可以使用shell看到这点：

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently
>>> future_question.was_published_recently()
True
```

由于将来的事情并不能称之为‘最近’，这确实是一个错误。

创建一个测试来暴露这个错误

我们需要在自动化测试里做的和刚才在shell里做的差不多，让我们来将它转换成一个自动化测试。

应用的测试用例安装惯例一般放在该应用的tests.py文件中；测试系统将自动在任何以test开头的文件中查找测试用例。

将下面的代码放入polls应用下的tests.py文件中：

```
polls/tests.py
import datetime

from django.utils import timezone
from django.test import TestCase

from .models import Question


class QuestionMethodTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questions whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertEqual(future_question.was_published_recently(), False)
```

我们在这里做的是创建一个django.test.TestCase子类，它具有一个方法可以创建一个pub_date在未来的Question实例。然后我们检查was_published_recently()的输出 —— 它应该是 False.

运行测试

在终端中，我们可以运行我们的测试：

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

你将看到类似下面的输出：

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_fut
    self.assertEqual(future_question.was_published_recently(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

发生了如下这些事：

- python manage.py test polls查找polls 应用下的测试用例
- 它找到 django.test.TestCase 类的一个子类
- 它为测试创建了一个特定的数据库
- 它查找用于测试的方法 —— 名字以test开始

- 它运行test_was_published_recently_with_future_question创建一个pub_date为未来30天的 Question实例
- ... 然后利用assertEqual()方法，它发现was_published_recently() 返回True，尽管我们希望它返回False

这个测试通知我们哪个测试失败，甚至是错误出现在哪一行。

修复这个错误

我们已经知道问题是什么：Question.was_published_recently() 应该返回 False，如果它的pub_date是在未来。在models.py中修复这个方法，让它只有当日期是在过去时才返回True：

```
polls/models.py
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

再次运行测试：

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

在找出一个错误之后，我们编写一个测试来暴露这个错误，然后在代码中更正这个错误让我们的测试通过。

未来，我们的应用可能会出许多其它的错误，但是我们可以保证我们不会无意中再次引入这个错误，因为简单地运行一下这个测试就会立即提醒我们。 我们可以认为这个应用的这一小部分会永远安全了。

更加综合的测试

在这里，我们可以使was_published_recently() 方法更加稳定；事实上，在修复一个错误的时候引入一个新的错误将是一件很令人尴尬的事。

在同一个类中添加两个其它的测试方法，来更加综合地测试这个方法：

```
polls/tests.py
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions whose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertEqual(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions whose
    pub_date is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertEqual(recent_question.was_published_recently(), True)
```

现在我们有三个测试来保证无论发布时间是在过去、现在还是未来Question.was_published_recently()都将返回合理的数据。

再说一次，polls 应用虽然简单，但是无论它今后会变得多么复杂以及会和多少其它的应用产生相互作用，我们都能保证我们刚刚为它编写过测试的那个方法会按照预期的那样工作。

测试一个视图

这个投票应用没有区分能力：它将会发布任何一个Question，包括 pub_date字段位于未来。我们应该改进这一点。 设定pub_date在未来应该表示Question在此刻发布，但是直到那个时间点才会变得可见。

视图的一个测试

当我们修复上面的错误时，我们先写测试，然后修改代码来修复它。 事实上，这是由测试驱动的开发的一个简单的例子，但做的顺序并不真的重要。

在我们的第一个测试中，我们专注于代码内部的行为。 在这个测试中，我们想要通过浏览器从用户的角度来检查它的行为。

在我们试着修复任何事情之前，让我们先查看一下我们能用到的工具。

Django测试客户端

Django提供了一个测试客户端来模拟用户和代码的交互。我们可以在tests.py 甚至在shell 中使用它。

我们将再次以shell开始，但是我们需要做很多在tests.py中不必做的事。首先是在 shell中设置测试环境：

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

setup_test_environment()安装一个模板渲染器，可以使我们来检查响应的一些额外属性比如response.context，否则是访问不到的。请注意，这种方法不会建立一个测试数据库，所以以下命令将运行在现有的数据库上，输出的内容也会根据你已经创建的Question不同而稍有不同。

下一步我们需要导入测试客户端类（在之后的tests.py 中，我们将使用django.test.TestCase类，它具有自己的客户端，将不需要导入这个类）：

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

这些都做完之后，我们可以让这个客户端来为我们做一些事：

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # note - you might get unexpected results if your ``TIME_ZONE``
>>> # in ``settings.py`` is not correct. If you need to change it,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?", pub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> response = client.get('/polls/')
>>> response.content
'\n\n\n    <ul>\n    \n        <li><a href="/polls/1/">Who is your favorite Beatle?</a></
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

改进我们的视图

投票的列表显示还没有发布的投票（即pub_date在未来的投票）。让我们来修复它。

在教程 4中，我们介绍了一个继承ListView的基于类的视图：

```
polls/views.py
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]
```

response.context_data['latest_question_list'] 取出由视图放置在context 中的数据。

我们需要修改get_queryset方法并让它将日期与timezone.now()进行比较。首先我们需要添加一行导入：

```
polls/views.py
from django.utils import timezone
```

然后我们必须像这样修改get_queryset方法：

```
polls/views.py
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) 返回一个查询集，包含pub_date小于等于timezone.now的Question。

测试我们的新视图

启动服务器、在浏览器中载入站点、创建一些发布时间在过去和将来的Questions，然后检验只有已经发布的Question会展示出来，现在你可以对自己感到满意了。你不想每次修改可能与这相关的代码时都重复这样做 —— 所以让我们基于以上shell会话中的内容，再编写一个测试。

将下面的代码添加到polls/tests.py：

```
polls/tests.py
from django.core.urlresolvers import reverse
```

我们将创建一个快捷函数来创建Question，同时我们要创建一个新的测试类：

```
polls/tests.py
def create_question(question_text, days):
    """
    Creates a question with the given `question_text` published the given
    number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text,
                                   pub_date=time)


class QuestionViewTests(TestCase):
    def test_index_view_with_no_questions(self):
        """
        If no questions exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_a_past_question(self):
        """
        Questions with a pub_date in the past should be displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.",
                            status_code=200)
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past question 1.>']
        )
```

让我们更详细地看下以上这些内容。

第一个是Question的快捷函数create_question，将重复创建Question的过程封装在一起。

test_index_view_with_no_questions不创建任何Question，但会检查消息“No polls are available.” 并验证latest_question_list为空。注意django.test.TestCase类提供一些额外的断言方法。在这些例子中，我们使用assertContains() 和 assertQuerysetEqual()。

在test_index_view_with_a_past_question中，我们创建一个Question并验证它是否出现在列表中。

在test_index_view_with_a_future_question中，我们创建一个pub_date 在未来的Question。数据库会为每一个测试方法进行重置，所以第一个Question已经不在那里，因此首页面里不应该有任何Question。

等等。 事实上，我们是在用测试模拟站点上的管理员输入和用户体验，检查针对系统每一个状态和状态的新变化，发布的是预期的结果。

测试 DetailView

一切都运行得很好； 然而，即使未来发布的Question不会出现在index中，如果用户知道或者猜出正确的URL依然可以访问它们。所以我们需要给DetailView添加一个这样的约束：

```
polls/views.py
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

当然，我们将增加一些测试来检验pub_date 在过去的Question 可以显示出来，而pub_date在未来的不可以：

```
polls/tests.py
class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        The detail view of a question with a pub_date in the future should
        return a 404 not found.
        """
        future_question = create_question(question_text='Future question.',
                                          days=5)
        response = self.client.get(reverse('polls:detail',
                                   args=(future_question.id,)))
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        The detail view of a question with a pub_date in the past should
        display the question's text.
        """
        past_question = create_question(question_text='Past Question.',
                                        days=-5)
        response = self.client.get(reverse('polls:detail',
                                   args=(past_question.id,)))
        self.assertContains(response, past_question.question_text,
                            status_code=200)
```

更多的测试思路

我们应该添加一个类似get_queryset的方法到ResultsView并为该视图创建一个新的类。这将与我们刚刚创建的非常类似；实际上将会有许多重复。

我们还可以在其它方面改进我们的应用，并随之不断增加测试。例如，发布一个没有Choices的Questions就显得傻傻的。所以，我们的视图应该检查这点并排除这些 Questions。我们的测试应该创建一个不带Choices 的 Question然后测试它不会发布出来， 同时创建一个类似的带有 Choices的Question 并验证它会 发布出来。

也许登陆的用户应该被允许查看还没发布的 Questions，但普通游客不行。 再说一次：无论添加什么代码来完成这个要求，需要提供相应的测试代码，无论你是否是先编写测试然后让这些代码通过测试，还是先用代码解决其中的逻辑然后编写测试来证明它。

从某种程度上来说，你一定会查看你的测试，然后想知道是否你的测试程序过于臃肿，这将我们带向下面的内容：

测试越多越好

看起来我们的测试代码的增长正在失去控制。 以这样的速度，测试的代码量将很快超过我们的应用，对比我们其它优美简洁的代码，重复毫无美感。

没关系。让它们继续增长。最重要的是，你可以写一个测试一次，然后忘了它。 当你继续开发你的程序时，它将继续执行有用的功能。

有时，测试需要更新。 假设我们修改我们的视图使得只有具有Choices的 Questions 才会发布。在这种情况下，我们许多已经存在的测试都将失败 —— 这会告诉我们哪些测试需要被修改来使得它们保持最新，所以从某种程度上讲，测试可以自己照顾自己。

在最坏的情况下，在你的开发过程中，你会发现许多测试现在变得冗余。 即使这样，也不是问题；对测试来说，冗余是一件好 事。

只要你的测试被合理地组织，它们就不会变得难以管理。 从经验上来说，好的做法是：

- 每个模型或视图具有一个单独的TestClass
- 为你想测试的每一种情况建立一个单独的测试方法
- 测试方法的名字可以描述它们的功能

进一步的测试

本教程只介绍了一些基本的测试。 还有很多你可以做，有许多非常有用的工具可以随便使用来你实现一些非常聪明的做法。

例如，虽然我们的测试覆盖了模型的内部逻辑和视图发布信息的方式，你可以使用一个“浏览器”框架例如Selenium来测试你的HTML文件在浏览器中真实渲染的样子。 这些工具不仅可以让你检查你的Django代码的行为，还能够检查你的JavaScript的行为。 它会启动一个浏览器，并开始与你的网站进行交互，就像有一个人在操纵一样，非常值得一看！ Django 包含一个LiveServerTestCase来帮助与Selenium 这样的工具集成。

如果你有一个复杂的应用，你可能为了实现continuous integration，想在每次提交代码后对代码进行自动化测试，让代码自动 —— 至少是部分自动 —— 地来控制它的质量。

发现你应用中未经测试的代码的一个好方法是检查测试代码的覆盖率。 这也有助于识别脆弱的甚至死代码。 如果你不能测试一段代码，这通常意味着这些代码需要被重构或者移除。 Coverage将帮助我们识别死代码。 查看与coverage.py 集成来了解更多细节。

Django 中的测试有关于测试更加全面的信息。

下一步？

关于测试的完整细节，请查看Django 中的测试。

当你对Django 视图的测试感到满意后，请阅读本教程的第6部分来 了解静态文件的管理。

> 译者：Django 文档协作翻译小组，原文：Part 5: Testing。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写你的第一个Django应用，第6部分

本教程上接教程 5。 我们已经建立一个测试过的网页投票应用，现在我们将添加一张样式表和一张图片。

除了由服务器生成的HTML文件外，网页应用一般需要提供其它必要的文件 —— 比如图片文件、JavaScript脚本和CSS样式表 —— 来为用户呈现出一个完整的网站。 在Django中，我们将这些文件称为“静态文件”。

对于小型项目，这不是个大问题，因为你可以将它们放在你的网页服务器可以访问到的地方。 然而，在大一点的项目中 —— 尤其是那些由多个应用组成的项目 —— 处理每个应用提供的多个静态文件集合开始变得很难。

这正是django.contrib.staticfiles的用途：它收集每个应用（和任何你指定的地方）的静态文件到一个单独的位置，这个位置在线上可以很容易维护。

自定义你的应用的外观

首先在你的polls中创建一个static目录。Django将在那里查找静态文件，与Django如何polls/templates/内部的模板类似。

Django 的 STATICFILES_FINDERS 设置包含一个查找器列表，它们知道如何从各种源找到静态文件。 其中默认的一个是AppDirectoriesFinder，它在每个INSTALLED_APPS下查找“static”子目录，就像刚刚在polls中创建的一样。管理站点也为它的静态文件使用相同的目录结构。

在你刚刚创建的static目录中，创建另外一个目录polls并在它下面创建一个文件style.css。换句话讲，你的样式表应该位于polls/static/polls/style.css。因为AppDirectoriesFinder 静态文件查找器的工作方式，你可以通过polls/style.css在Django中访问这个静态文件，与你如何访问模板的路径类似。

> 静态文件的命名空间
>
> 与模板类似，我们可以家那个我们的静态文件直接放在polls/static（而不是创建另外一个 polls 子目录），但实际上这是一个坏主意。Django将使用它所找到的第一个文件名符合要求的静态文件，如果在你的不同应用中存在两个同名的静态文件，Django将无法区分它们。 我们需要告诉Django该使用其中的哪一个，最简单的方法就是为它们添加命名空间。 也就是说，将这些静态文件放进以它们所在的应用的名字命名的另外一个目录下。

将下面的代码放入样式表中 (polls/static/polls/style.css)：

```
polls/static/polls/style.css
li a {
    color: green;
}
```

下一步，在polls/templates/polls/index.html的顶端添加如下内容 ：

```
polls/templates/polls/index.html
{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

`{% load staticfiles %}` 从staticfiles模板库加载 `{% static %}` 模板标签。 `{% static %}` 模板标签会生成静态文件的绝对URL。

这就是你在开发过程中，所需要对静态文件做的所有处理。 重新加载 http://localhost:8000/polls/ ，你应该会看到Question的超链接变成了绿色（Django的风格！），这意味着你的样式表被成功导入。

添加一张背景图片

下一步，我们将创建一个子目录来存放图片。 在polls/static/polls/目录中创建一个 images 子目录。在这个目录中，放入一张图片background.gif。换句话，将你的图片放在 polls/static/polls/images/background.gif。

然后，向你的样式表添加（polls/static/polls/style.css）：

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

重新加载 http://localhost:8000/polls/ ，你应该在屏幕的右下方看到载入的背景图片。

> 警告：
>
> 当然， `{% static %}` 模板标签不能用在静态文件（比如样式表）中，因为他们不是由Django生成的。 你应该永远使用相对路径来相互链接静态文件，因为这样你可以改变STATIC_URL （ static模板标签用它来生成URLs）而不用同时修改一大堆静态文件的路径。

这些知识基础。关于静态文件设置的更多细节和框架中包含的其它部分，参见静态文件 howto 和静态文件参考。部署静态文件讨论如何在真实的服务器上使用静态文件。

下一步？

新手教程到此结束。 在这期间，你可能想要在如何查看文档中了解文档的结构和查找相关信息方法。

如果你熟悉Python 打包的技术，并且对如何将投票应用制作成一个“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。

> 译者：Django 文档协作翻译小组，原文：Part 6: Static files。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级教程

高级教程：如何编写可重用的应用

本高级教程上接教程 6。我们将把我们的网页投票转换成一个独立的Python包，这样你可以在其它项目中重用或者分享给其它人。

如果你最近没有完成教程1–6，我们建议你阅读它们使得你的示例项目与下面描述的相匹配。

可重用很重要

设计、构建、测试和维护一个网页应用有许多工作要做。许多Python 和 Django 项目都有常见的共同问题。如果我们可以节省一些这些重复的工作会不会很棒？

可重用性是Python 中一种生活的态度。Python包索引 (PyPI) 具有广泛的包，你可以在你自己的Python程序中使用。调查一下Django Packages中已经存在的可重用的应用，你可以结合它们到你的项目。Django 自身也只是一个Python 包。这意味着你可以获取已经存在的Python包和Django应用并将它们融合到你自己的网页项目。你只需要编写你项目的独特的部分。

比如说，你正在开始一个新的项目，需要一个像我们正在编写的投票应用。你如何让该应用可重用？幸运的是，你已经在正确的道路上。在教程 3中，我们看到我们可以如何使用include将投票应用从项目级别的URLconf 解耦。在本教程中，我们将更进一步，让你的应用在新的项目中容易地使用并随时可以发布给其它人安装和使用。

> 包？应用？
>
> Python 包 提供的方式是分组相关的Python 代码以容易地重用。一个包包含一个或多个Python代码（也叫做“模块”）。
>
> 包可以通过import foo.bar 或from foo import bar 导入。如果一个目录（例如polls）想要形成一个包，它必须包含一个特殊的文件**init**.py，即使这个文件为空。
>
> 一个Django 应用 只是一个Python包，它特意用于Django项目中。一个应用可以使用常见的Django 约定，例如具有models、tests、urls和views 子模块。
>
> 后面我们使用打包这个词来描述将一个Python包变得让其他人易于安装的过程。我们知道，这可能有点绕人。

你的项目和你的可重用的应用

经过前面的教程之后，我们的项目应该看上去像这样：

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    polls/
        __init__.py
        admin.py
        migrations/
            __init__.py
            0001_initial.py
        models.py
        static/
            polls/
                images/
                    background.gif
                style.css
        templates/
            polls/
                detail.html
                index.html
                results.html
        tests.py
        urls.py
        views.py
    templates/
        admin/
            base_site.html
```

你在教程 2中创建了mysite/templates，在教程 3中创建了polls/templates。现在你可能更加清晰为什么我们为项目和应用选择单独的模板目录：属于投票应用的部分全部在polls中。它使得该应用自包含且更加容易丢到一个新的项目中。

现在可以拷贝polls目录到一个新的Django项目并立即使用。然后它还不能充分准备好到可以立即发布。由于这点，我们需要打包这个应用来让它对其他人易于安装。

安装一些前提条件

Python 打包的目前状态因为有多种工具而混乱不堪。对于本教程，我们打算使用setuptools来构建我们的包。它是推荐的打包工具（已经与distribute 分支合并）。我们还将使用pip来安装和卸载它。现在你应该安装这两个包。如果你需要帮助，你可以参考如何使用pip安装Django。你可以使用同样的方法安装setuptools。

打包你的应用

Python packaging refers to preparing your app in a specific format that can be easily installed and used. Django 自己是以非常相似的方式打包起来的。对于一个像polls这样的小应用，这个过程不是太难。

首先，在你的Django项目之外，为polls创建一个父目录。称这个目录为django-polls。

> 为你的应用选择一个名字
>
> 让为你的包选择一个名字时，检查一下PyPI中的资源以避免与已经存在的包有名字冲突。当创建一个要发布的包时，在你的模块名字前面加上django-通常很有用。 这有助于其他正在查找Django应用的人区分你的应用是专门用于Django的。
>
> 应用的标签（应用的包的点分路径的最后部分）在INSTALLED_APPS中必须唯一。避免使用与Django的contrib 包 中任何一个使用相同的标签，例如auth、admin和messages。

将polls 目录移动到django-polls目录。

创建一个包含一些内容的文件django-polls/README.rst：

```
django-polls/README.rst
=====
Polls
=====

Polls is a simple Django app to conduct Web-based polls. For each
question, visitors can choose between a fixed number of answers.

Detailed documentation is in the "docs" directory.

Quick start
-----------

1. Add "polls" to your INSTALLED_APPS setting like this::

    INSTALLED_APPS = (
        ...
        'polls',
    )

2. Include the polls URLconf in your project urls.py like this::

    url(r'^polls/', include('polls.urls')),

3. Run `python manage.py migrate` to create the polls models.

4. Start the development server and visit http://127.0.0.1:8000/admin/
   to create a poll (you'll need the Admin app enabled).

5. Visit http://127.0.0.1:8000/polls/ to participate in the poll.
```

创建一个django-polls/LICENSE文件。选择License超出本教程的范围，但值得一说的是公开发布的代码如果没有License是毫无用处的。Django和许多与Django兼容的应用以BSD License 发布；然而，你可以随便挑选自己的License。只需要知道你的License的选则将影响谁能够使用你的代码。

下一步我们将创建一个setup.py 文件，它提供如何构建和安装该应用的详细信息。该文件完整的解释超出本教程的范围，setuptools 文档 有很好的解释。创建一个文件django-polls/setup.py，其内容如下：

```
django-polls/setup.py
import os
from setuptools import setup

with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme:
    README = readme.read()

# allow setup.py to be run from any path
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__), os.pardir)))

setup(
    name='django-polls',
    version='0.1',
    packages=['polls'],
    include_package_data=True,
    license='BSD License',  # example license
    description='A simple Django app to conduct Web-based polls.',
    long_description=README,
    url='http://www.example.com/',
    author='Your Name',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Replace these appropriately if you are stuck on Python 2.
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

默认只有Python模块和包会包含进包中。如果需要包含额外的文件，我们需要创建一个MANIFEST.in文件。上一步提到的setuptools 文档对这个文件有更详细的讨论。如果要包含模板、README.rst和我们的LICENSE 文件，创建一个文件django-polls/MANIFEST.in，其内容如下：

```
django-polls/MANIFEST.in
include LICENSE
include README.rst
recursive-include polls/static *
recursive-include polls/templates *
```

将详细的文档包含进你的应用中，它是可选的，但建议你这样做。创建一个空的目录django-polls/docs用于将来存放文档。向django-polls/MANIFEST.in添加另外一行：

```
recursive-include docs *
```

注意docs不会包含进你的包中除非你添加一些文件到它下面。许多Django应用还通过类似readthedocs.org这样的站点提供它们的在线文档.

试着通过python setup.py sdist 构建你的包（从django-polls的内部运行）。这创建一个dist目录并构建一个新包django-polls-0.1.tar.gz。

更多关于打包的信息，参见Python 的 打包和分发项目的教程。

使用你自己的包

因为，我们将polls 目录移到项目的目录之外，它不再工作了。我们将通过安装我们的新的django-polls包来修复它。

> 安装成某个用户的库
>
> 以下的步骤将安装django-polls 成某个用户的库。根据用户安装相比系统范围的安装具有许多优点，例如用于没有管理员权限的系统上以及防止你的包影响系统的服务和机器上的其它用户。
>
> 注意根据用户的安装仍然可以影响以该用户身份运行的系统工具，所以virtualenv 是更健壮的解决办法（见下文）。

安装这个包，使用pip（你已经安装好它了，对吧？）：

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

如果幸运，你的Django 项目现在应该可以再次正确工作。请重新运行服务器以证实这点。

若要卸载这个包，使用pip：

```
pip uninstall django-polls
```

发布你的应用：

既然我们已经打包并测试过django-polls，是时候与世界共享它了！要不是它仅仅是个例子，你现在可以：

- 将这个包用邮件发送给朋友。
- 上传这个包到你的网站上。
- 上传这个包到一个公开的仓库，例如Python 包索引 (PyPI)。packaging.python.org has a good tutorial for doing this.

使用 **virtualenv** 安装**Python** 包

前面，我们将poll 安装成一个用户的库。它有一些缺点：

- 修改这个用户的库可能影响你的系统上的其它Python 软件。
- 你将不可以运行这个包的多个版本（或者具有相同名字的其它包）。

特别是一旦你维护几个Django项目，这些情况就会出现。如果确实出现，最好的解决办法是使用virtualenv。这个工具允许你维护多个分离的Python环境，每个都具有它自己的库和包的命名空间。

> 译者：Django 文档协作翻译小组，原文：How to write reusable apps。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

为Django编写首个补丁

介绍

有兴趣为社区做出点贡献吗？也许你会在Django中发现你想要修复的漏洞，或者你希望为它添加一个小特征。

为Django作贡献这件事本身就是使你的顾虑得到解决的最好方式。一开始这可能会使你怯步，但事实上是很简单的。整个过程中我们会一步一步为你解说，所以你可以通过例子学习。

Who's this tutorial for?

使用教程前，我们希望你至少对于Django的运行方式有基础的了解。这意味着你可以自如地在写你自己的Django app时使用教程。 除此之外，你应该对于Python本身有很好的了解。如果您并不太了解， 我们为您推荐Dive Into Python，对于初次使用Python的程序员来说这是一本很棒（而且免费）的在线电子书。

对于版本控制系统及Trac不熟悉的人来说，这份教程及其中的链接所包含的信息足以满足你们开始学习的需求。然而，如果你希望定期为Django贡献，你可能会希望阅读更多关于这些不同工具的信息。

当然对于其中的大部分内容，Django会尽可能做出解释以帮助广大的读者。

> 何处获得帮助:
>
> 如果你在使用本教程时遇到困难，你可以发送信息给django开发者 或者登陆 #django-dev on irc.freenode.net 向其他Django使用者需求帮助。

教程包含的内容

一开始我们会帮助你为Django编写补丁，在教程结束时，你将具备对于工具和所包含过程的基本了解。准确来说，我们的教程将包含以下几点：

- 安装Git。
- 如何下载Django的开发复本
- 运行Django的测试组件
- 为你的补丁编写一个测试
- 为你的补丁编码。
- 测试你的补丁。

- 为你所做的改变写一个补丁文件。
- 去哪里寻找更多的信息。

一旦你完成了这份教程，你可以浏览剩下的Django's documentation on contributing. 它包含了大量信息。任何想成为Django的正式贡献者必须去阅读它。如果你有问题，它也许会给你答案

安装Git

使用教程前，你需要安装好Git，下载Django的最新开发版本并且为你作出的改变生成补丁文件

为了确认你是否已经安装了Git, 输入 git 进入命令行。如果信息提示命令无法找到, 你就需要下载并安装Git, 详情阅读 Git's download page.

模型层

Django 提供了一个抽象层（模型），对您的Web 应用中的数据进行构建及操作。通过以下内容来了解更多:

模型

模型

模型是你的数据的唯一的、权威的信息源。它包含你所储存数据的必要字段和行为。通常，每个模型对应数据库中唯一的一张表。

基础：

- 每个模型都是 `django.db.models.Model` 的一个Python 子类。
- 模型的每个属性都表示数据库中的一个字段。
- Django 提供一套自动生成的用于数据库访问的API；详见执行查询。

简短的例子

这个例子定义一个 `Person` 模型，它有 `first_name` 和 `last_name` 两个属性：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` 和 `last_name` 是模型的两个字段。每个字段都被指定成一个类属性，每个属性映射到一个数据库的列。

上面的 `Person` 模型会在数据库中创建这样一张表：

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

一些技术上的注意事项：

- 这个表的名称 `myapp_person`，是根据 模型中的元数据自动生成的，也可以覆写为别的名称，详见*Table names*。
- `id` 字段是自动添加的，但这个行为可以被重写。详见自增主键字段。
- 这个例子中的 `CREATE TABLE` SQL 语句使用PostgreSQL 语法格式，要注意的是Django 会根据设置文件 中指定的数据库类型来使用相应的SQL 语句。

使用模型

定义好模型之后，你需要告诉Django 使用这些模型。你要做的就是修改配置文件中的 `INSTALLED_APPS` 设置，在其中添加 `models.py` 所在应用的名称。

例如，如果你的应用的模型位于 `myapp.models` 模块（ `manage.py startapp` 脚本为一个应用创建的包结构），`INSTALLED_APPS` 部分看上去应该是：

```
INSTALLED_APPS = (
    #...
    'myapp',
    #...
)
```

当你在 `INSTALLED_APPS` 中添加新的应用名时，请确保运行命令 `manage.py migrate`，可以首先使用 `manage.py makemigrations` 来为它们生成迁移脚本。

字段

模型中不可或缺且最为重要的，就是字段集，它是一组数据库字段的列表。字段被指定为类属性。 要注意选择的字段名称不要和模型 *API* 冲突，比如 `clean`、 `save` 或者 `delete`。

例如：

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

字段类型

模型中的每个字段都是 `Field` 子类的某个实例。Django 根据字段类的类型确定以下信息：

- 数据库当中的列类型 (比如， `INTEGER`, `VARCHAR`)。
- 渲染表单时使用的默认HTML 部件（例如， `<input type="text">`, `<select>` ）。
- 最低限度的验证需求，它被用在 Django 管理站点和自动生成的表单中。

Django 自带数十种内置的字段类型；完整字段类型列表可以在模型字段参考 中找到。如果内置类型仍不能满足你的要求，你可以自由地编写符合你要求的字段类型； 详见编写自定义的模型字段。

字段选项

每个字段有一些特有的参数，详见模型字段参考。例如，`CharField`（和它的派生类）需要`max_length`参数来指定`VARCHAR`数据库字段的大小。

还有一些适用于所有字段的通用参数。这些参数在参考中有详细定义，这里我们只简单介绍一些最常用的：

`null`

如果为`True`，Django 将用`NULL`来在数据库中存储空值。默认值是`False`.

`blank`

如果为`True`，该字段允许不填。默认为`False`。

要注意，这与`null`不同。`null`纯粹是数据库范畴的，而`blank`是数据验证范畴的。如果一个字段的`blank=True`，表单的验证将允许该字段是空值。如果字段的`blank=False`，该字段就是必填的。

`choices`

由二元组组成的一个可迭代对象（例如，列表或元组），用来给字段提供选择项。如果设置了choices，默认的表单将是一个选择框而不是标准的文本框，而且这个选择框的选项就是 choices 中的选项。

这是一个关于 choices 列表的例子：

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

每个元组中的第一个元素，是存储在数据库中的值；第二个元素是在管理界面或 ModelChoiceField 中用作显示的内容。在一个给定的 model 类的实例中，想得到某个 choices 字段的显示值，就调用`get_FOO_display`方法(这里的 FOO 就是 choices 字段的名称)。例如：

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`default`

字段的默认值。可以是一个值或者可调用对象。如果可调用，每有新对象被创建它都会被调用。

`help_text`

表单部件额外显示的帮助内容。即使字段不在表单中使用，它对生成文档也很有用。

`primary_key`

如果为 `True`，那么这个字段就是模型的主键。

如果你没有指定任何一个字段的 `primary_key=True`，Django 就会自动添加一个 `IntegerField` 字段做为主键，所以除非你想覆盖默认的主键行为，否则没必要设置任何一个字段的 `primary_key=True`。详见自增主键字段。

主键字段是只读的。如果你在一个已存在的对象上面更改主键的值并且保存，一个新的对象将会在原有对象之外创建出来。例如：

```
from django.db import models

class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name='Apple')
>>> fruit.name = 'Pear'
>>> fruit.save()
>>> Fruit.objects.values_list('name', flat=True)
['Apple', 'Pear']
```

`unique`

如果该值设置为 `True`，这个数据字段的值在整张表中必须是唯一的

再说一次，这些仅仅是常用字段的简短介绍， 要了解详细内容，请查看 通用 model 字段选项参考(*common model field option reference*).

自增主键字段

默认情况下，Django 会给每个模型添加下面这个字段：

```
id = models.AutoField(primary_key=True)
```

这是一个自增主键字段。

如果你想指定一个自定义主键字段，只要在某个字段上指定 `primary_key=True` 即可。如果 Django 看到你显式地设置了 `Field.primary_key`，就不会自动添加 `id` 列。

每个模型只能有一个字段指定 `primary_key=True`（无论是显式声明还是自动添加）。

字段的自述名

除 `ForeignKey`、`ManyToManyField` 和 `OneToOneField` 之外，每个字段类型都接受一个可选的位置参数 —— 字段的自述名。如果没有给定自述名，Django 将根据字段的属性名称自动创建自述名 —— 将属性名称的下划线替换成空格。

在这个例子中，自述名是 `"person's first name"`：

```
first_name = models.CharField("person's first name", max_length=30)
```

在这个例子中，自述名是 `"first name"`：

```
first_name = models.CharField(max_length=30)
```

`ForeignKey`、`ManyToManyField` 和 `OneToOneField` 都要求第一个参数是一个模型类，所以要使用 `verbose_name` 关键字参数才能指定自述名：

```
poll = models.ForeignKey(Poll, verbose_name="the related poll")
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(Place, verbose_name="related place")
```

习惯上，`verbose_name` 的首字母不用大写。Django 在必要的时候会自动大写首字母。

关系

显然，关系数据库的威力体现在表之间的相互关联。 Django 提供了三种最常见的数据库关系：多对一(many-to-one)，多对多(many-to-many)，一对一(one-to-one)。

多对一关系

Django 使用 `django.db.models.ForeignKey` 定义多对一关系。和使用其它 `字段` 类型一样：在模型当中把它做为一个类属性包含进来。

`ForeignKey` 需要一个位置参数：与该模型关联的类。

比如，一辆 `Car` 有一个 `Manufacturer` —— 但是一个 `Manufacturer` 生产很多Car，每一辆 `Car` 只能有一个 `Manufacturer` —— 使用下面的定义：

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    # ...
```

你还可以创建递归的关联关系（对象和自己进行多对一关联）和 与尚未定义的模型的关联关系；详见模型字段参考。

建议你用被关联的模型的小写名称做为 `ForeignKey` 字段的名字（例如，上面 `manufacturer` ）。当然，你也可以起别的名字。例如：

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

另见

`ForeignKey` 字段还接受许多别的参数，在模型字段参考有详细介绍。这些选项帮助定义关联关系应该如何工作；它们都是可选的参数。

访问反向关联对象的细节，请见*Following relationships backward example*。

示例代码，请见多对一关系模型示例)。

多对多关系

`ManyToManyField` 用来定义多对多关系，用法和其他 `Field` 字段类型一样：在模型中做为一个类属性包含进来。

`ManyToManyField` 需要一个位置参数：和该模型关联的类。

例如，一个 `Pizza` 可以有多种 `Topping` —— 一种 `Topping` 可以位于多个Pizza 上，而且每个 `Pizza` 可以有多种Topping —— 如下：

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

和使用 `ForeignKey` 一样，你也可以创建递归的关联关系（对象与自己的多对多关联）和与尚未定义关系的模型的关联关系；详见模型字段参考。

建议你以被关联模型名称的复数形式做为 `ManyToManyField` 的名字（例如上例中的 `toppings`）。

在哪个模型中设置 `ManyToManyField` 并不重要，在两个模型中任选一个即可 —— 不要两个模型都设置。

通常， `ManyToManyField` 实例应该位于可以编辑的表单中。在上面的例子中， `toppings` 位于 `Pizza` 中（而不是在 `Topping` 里面设置 `pizzas` 的 `ManyToManyField` 字段），因为设想一个Pizza 有多种Topping 比一个Topping 位于多个Pizza 上要更加自然。按照上面的方式，在 `Pizza` 的表单中将允许用户选择不同的Toppings。

另见

完整的示例参见多对多关系模型示例。

`ManyToManyField` 字段还接受别的参数，在模型字段参考中有详细介绍。这些选项帮助定义关系应该如何工作；它们都是可选的。

多对多关系中的其他字段

处理类似搭配 pizza 和 topping 这样简单的多对多关系时，使用标准的 `ManyToManyField` 就可以了。但是，有时你可能需要关联数据到两个模型之间的关系上。

例如，有这样一个应用，它记录音乐家所属的音乐小组。我们可以用一个 `ManyToManyField` 表示小组和成员之间的多对多关系。但是，有时你可能想知道更多成员关系的细节，比如成员是何时加入小组的。

对于这些情况，Django 允许你指定一个模型来定义多对多关系。 你可以将其他字段放在中介模型里面。源模型的 `ManyToManyField` 字段将使用 `through` 参数指向中介模型。对于上面的音乐小组的例子，代码如下：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

在设置中介模型时，要显式指定外键并关联到多对多关系涉及的模型。这个显式声明定义两个模型之间是如何关联的。

中介模型有一些限制：

- 中介模型必须有且只有一个外键到源模型（上面例子中的 `Group`），或者你必须使用 `ManyToManyField.through_fields` 显式指定Django 应该使用的外键。如果你的模型中存在超个一个的外键，并且 `through_fields` 没有指定，将会触发一个无效的错误。 对目标模型的外键有相同的限制（上面例子中的 `Person`）。
- 对于通过中介模型与自己进行多对多关联的模型，允许存在到同一个模型的两个外键，但它们将被作为多对多关联关系的两个（不同的）方面。如果有超过 两个外键，同样你必须像上面一样指定 `through_fields`，否则将引发一个验证错误。
- 使用中介模型定义与自身的多对多关系时，你必须设置 `symmetrical=False`（详见模型字段参考）。

Changed in Django 1.7:

在Django 1.6 及之前的版本中，中介模型禁止包含多于一个的外键。

既然你已经设置好 `ManyToManyField` 来使用中介模型（在这个例子中就是 `Membership`），接下来你要开始创建多对多关系。你要做的就是创建中介模型的实例：

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

与普通的多对多字段不同，你不能使用 `add`、 `create` 和赋值语句（比如， `beatles.members = [...]`）来创建关系：

```
# THIS WILL NOT WORK
>>> beatles.members.add(john)
# NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
# AND NEITHER WILL THIS
>>> beatles.members = [john, paul, ringo, george]
```

为什么不能这样做？这是因为你不能只创建 `Person` 和 `Group` 之间的关联关系，你还要指定 `Membership` 模型中所需要的所有信息；而简单的 `add`、`create` 和赋值语句是做不到这一点的。所以它们不能在使用中介模型的多对多关系中使用。此时，唯一的办法就是创建中介模型的实例。

`remove()` 方法被禁用也是出于同样的原因。但是 `clear()` 方法却是可用的。它可以清空某个实例所有的多对多关系：

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
[]
```

通过创建中介模型的实例来建立对多对多关系后，你就可以执行查询了。 和普通的多对多字段一样，你可以直接使用被关联模型的属性进行查询：

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

如果你使用了中介模型，你也可以利用中介模型的属性进行查询：

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr]
```

如果你需要访问一个成员的信息，你可以直接获取 `Membership` 模型：

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

另一种获取相同信息的方法是，在 `Person` 对象上查询多对多反转关系：

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

一对一关系

`OneToOneField` 用来定义一对一关系。 用法和其他 `字段` 类型一样：在模型里面做为类属性包含进来。

当某个对象想扩展自另一个对象时，最常用的方式就是在这个对象的主键上添加一对一关系。

`OneToOneField` 要一个位置参数：与模型关联的类。

例如，你想建一个“places” 数据库，里面有一些常用的字段，比如address、 phone number 等等。 接下来，如果你想在Place 数据库的基础上建立一个Restaurant 数据库，而不想将已有的字段复制到 `Restaurant` 模型，那你可以在 `Restaurant` 添加一个 `OneToOneField` 字段，这个字段指向 `Place` （因为Restaurant 本身就是一个Place；事实上，在处理这个问题的时候，你应该使用一个典型的 继承，它隐含一个一对一关系)。

和使用 `ForeignKey` 一样，你可以定义 递归的关联关系和引用尚未定义关系的模型。详见模型字段参考。

另见

在一对一关系的模型例子 中有一套完整的例子。

`OneToOneField` 字段也接受一个特定的可选的 `parent_link` 参数，在模型字段参考 中有详细介绍。

在以前的版本中， `OneToOneField` 字段会自动变成模型 的主键。不过现在已经不这么做了(不过要是你愿意的话，你仍可以传递 `primary_key` 参数来创建主键字段)。所以一个 模型 中可以有多个 `OneToOneField` 字段。

跨文件的模型

访问其他应用的模型是非常容易的。 在文件顶部你定义模型的地方，导入相关的模型来实现它。然后，无论在哪里需要的话，都可以引用它。例如：

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(ZipCode)
```

字段命名的限制

Django 对字段的命名只有两个限制：

1. 字段的名称不能是Python 保留的关键字，因为这将导致一个Python 语法错误。例如：

   ```
   class Example(models.Model):
       pass = models.IntegerField() # 'pass' is a reserved word!
   ```

2. 由于Django 查询语法的工作方式，字段名称中连续的下划线不能超过一个。例如：

```
class Example(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
```

这些限制有变通的方法，因为没有要求字段名称必须与数据库的列名匹配。参 `db_column` 选项。

SQL 的保留字例如 `join`、`where` 和 `select`，可以用作模型的字段名，因为Django 会对底层的SQL 查询语句中的数据库表名和列名进行转义。 它根据你的数据库引擎使用不同的引用语法。

自定义字段类型

如果已有的模型字段都不合适，或者你想用到一些很少见的数据库列类型的优点，你可以创建你自己的字段类型。创建你自己的字段在编写自定义的模型字段中有完整讲述。

元选项

使用内部的 `class Meta` 定义模型的元数据，例如：

```
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

模型元数据是“任何不是字段的数据”，比如排序选项（`ordering`），数据表名（`db_table`）或者人类可读的单复数名称（`verbose_name` 和 `verbose_name_plural`）。在模型中添加 `class Meta` 是完全可选的，所有选项都不是必须的。

所有 `元` 选项的完整列表可以在模型选项参考找到。

模型的属性

`objects`

The most important attribute of a model is the `Manager`. It's the interface through which database query operations are provided to Django models and is used to *retrieve the instances* from the database. If no custom `Manager` is defined, the default name is `objects`. Managers are only accessible via model classes, not the model instances.

模型的方法

可以在模型上定义自定义的方法来给你的对象添加自定义的“底层”功能。`Manager` 方法用于“表范围”的事务，模型的方法应该着眼于特定的模型实例。

这是一个非常有价值的技术，让业务逻辑位于同一个地方 —— 模型中。

例如，下面的模型具有一些自定义的方法：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

这个例子中的最后一个方法是一个*property*。

模型实例参考 具有一个完整的为模型自动生成的方法 列表。你可以覆盖它们 —— 参见下文 覆盖模型预定义的方法 —— 但是有些方法你会始终想要重新定义：

`__str__()` (Python 3)

Python 3 equivalent of `__unicode__()`.

`__unicode__()` (Python 2)

一个Python “魔法方法”，返回对象的Unicode “表示形式”。当模型实例需要强制转换并显示为普通的字符串时，Python 和Django 将使用这个方法。最明显是在交互式控制台或者管理站点显示一个对象的时候。

将将永远想要定义这个方法；默认的方法几乎没有意义。

`get_absolute_url()`

它告诉Django 如何计算一个对象的URL。Django 在它的管理站点中使用到这个方法，在其它任何需要计算一个对象的URL 时也将用到。

任何具有唯一标识自己的URL 的对象都应该定义这个方法。

覆盖预定义的模型方法

还有另外一部分封装数据库行为的模型方法，你可能想要自定义它们。特别是，你将要经常改变 `save()` 和 `delete()` 的工作方式。

你可以自由覆盖这些方法（和其它任何模型方法）来改变它们的行为。

覆盖内建模型方法的一个典型的使用场景是，你想在保存一个对象时做一些其它事情。例如（参见 `save()` 中关于它接受的参数的文档）：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
        do_something_else()
```

你还可以阻止保存：

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
```

必须要记住调用超类的方法—— `super(Blog, self).save(*args, **kwargs)` —— 来确保对象被保存到数据库中。如果你忘记调用超类的这个方法，默认的行为将不会发生且数据库不会有任何改变。

还要记住传递参数给这个模型方法 —— 即 `*args, **kwargs` 。 Django 未来将一直会扩展内建模型方法的功能并添加新的参数。如果在你的方法定义中使用 `*args, **kwargs` ，将保证你的代码自动支持这些新的参数。

Overridden model methods are not called on bulk operations

注意，当使用查询集批量删除对象时，将不会为每个对象调用 `delete()` 方法。为确保自定义的删除逻辑得到执行，你可以使用 `pre_delete` 和/或 `post_delete` 信号。

不幸的是，当批量 `creating` 或 `updating` 对象时没有变通方法，因为不会调用 `save()` 、 `pre_save` 和 `post_save` 。

执行自定义的SQL

另外一个常见的需求是在模型方法和模块级别的方法中编写自定义的SQL 语句。关于使用原始SQL 语句的更多细节，参见使用原始 *SQL* 的文档。

模型继承

Django 中的模型继承与 Python 中普通类继承方式几乎完全相同，但是本页头部列出的模型基本的要求还是要遵守。这表示自定义的模型类应该继承 `django.db.models.Model` 。

你唯一需要作出的决定就是你是想让父模型具有它们自己的数据库表，还是让父模型只持有一些共同的信息而这些信息只有在子模型中才能看到。

在Django 中有3中风格的继承。

1. 通常，你只想使用父类来持有一些信息，你不想在每个子模型中都敲一遍。这个类永远不会单独使用，所以你使用抽象基类。
2. 如果你继承一个已经存在的模型且想让每个模型具有它自己的数据库表，那么应该使用多表继承。
3. 最后，如果你只是想改变模块Python 级别的行为，而不用修改模型的字段，你可以使用代理模型。

抽象基类

当你想将一些常见信息存储到很多model的时候，抽象化类是十分有用的。你编写完基类之后，在 *Meta*类中设置 `abstract=True` ，该类就不能创建任何数据表。取而代之的是，当它被用来作为一个其他model的基础类时，它将被加入那一子类中。如果抽象化基础类和它的子类有相同的项，那么将会出现error（并且Django将返回一个exception）。

一个例子

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

`Student` 模型将有三个项： `name` ， `age` 和 `home_group` 。 `CommonInfo` 模型无法像一般的Django模型一样使用，因为它是一个抽象化基础类。它无法生成数据表单或者管理器，并且不能实例化或者储存。

对很多用户来说, 这种类型的模型继承就是你想要的。它提供一种在 Python 语言层级上提取公共信息的方式，但在数据库层级上，各个子类仍然只创建一个数据库。

元 继承

当一个抽象类被创建的时候, Django会自动把你在基类中定义的 *Meta* 作为子类的一个属性。如果子类没有声明自己的*Meta* 类, 他将会继承父类的*Meta*. 如果子类想要扩展父类 的，可以继承父类的 *Meta* 即可，例如

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

继承时，Django 会对基类的 *Meta*类做一个调整：在安装 *Meta*属性之前，Django 会设置 `abstract=False` 。这意味着抽象基类的子类不会自动变成抽象类。 当然，你可以让一个抽象类继承另一个抽象基类，不过每次都要显式地设置 `abstract=True` 。

对于抽象基类而言，有些属性放在 *Meta* 内嵌类里面是没有意义的。例如，包含 `db_table` 将意味着所有的子类(是指那些没有指定自己的 *Meta* 类的子类)都使用同一张数据表，一般来说，这并不是我们想要的。

小心使用 `related_name`

如果你在 `ForeignKey` 或 `ManyToManyField` 字段上使用 `related_name` 属性，你必须总是为该字段指定一个唯一的反向名称。但在抽象基类上这样做就会引发一个很严重的问题。因为 Django 会将基类字段添加到每个子类当中，而每个子类的字段属性值都完全相同 (这里面就包括 `related_name`)。

当你在(且仅在)抽象基类中使用 `related_name` 时，如果想绕过这个问题，名称中就要包含 `'%(app_label)s'` 和 `'%(class)s'` 。

- `'%(class)s'` 会替换为子类的小写加下划线格式的名称，字段在子类中使用。
- `'%(app_label)s'` 会替换为应用的小写加下划线格式的名称，应用包含子类。每个安装的应用名称都应该是唯一的，而且应用里每个模型类的名称也应该是唯一的，所以产生的名称应该彼此不同。

例如，假设有一个app叫做 `common/models.py` ：

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="%(app_label)s_%(class)s_relate

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

以及另一个应用 `rare/models.py` ：

```
from common.models import Base

class ChildB(Base):
    pass
```

ChildA.m2m 字段的反向名称是 childa_related，而 ChildB.m2m 字段的反向名称是 childb_related。这取决于你如何使用 `'%(class)s'` 和 `'%(app_label)s` 来构造你的反向名称。如果你没有这样做，Django 就会在验证 model (或运行 `migrate`) 时抛出错误。

果你没有在抽象基类中为某个关联字段定义 `related_name` 属性，那么默认的反向名称就是子类名称加上 `'_set'` ，它能否正常工作取决于你是否在子类中定义了同名字段。例如，在上面的代码中，如果去掉 `related_name` 属性，在 `ChildA` 中， `m2m` 字段的反向名称就是 `childa_set` ；而 `ChildB` 的 m2m 字段的反向名称就是 `childb_set` 。

多表继承

这是 Django 支持的第二种继承方式。使用这种继承方式时，同一层级下的每个子 model 都是一个真正意义上完整的 model 。 每个子 model 都有专属的数据表，都可以查询和创建数据表。 继承关系在子 model 和它的每个父类之间都添加一个链接 (通过一个自动创建的 `OneToOneField` 来实现)。 例如：

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

`Place` 里面的所有字段在 `Restaurant` 中也是有效的，只不过数据保存在另外一张数据表当中。所以下面两个语句都是可以运行的：

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

如果你有一个 `Place` ，那么它同时也是一个 `Restaurant` ， 那么你可以使用子 model 的小写形式从 Pl `Place` 对象中获得与其对应的 `Restaurant` 对象：

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

但是，如果上例中的 `p` 并不是 `Restaurant` (比如它仅仅只是 `Place` 对象，或者它是其他类的父类)，那么在引用 `p.restaurant` 就会抛出 `Restaurant.DoesNotExist` 异常。

多表继承中的 `Meta`

在多表继承中，子类继承父类的 *Meta* 类是没什么意义的。所有的 *Meta* 选项已经对父类起了作用，再次使用只会起反作用。(这与使用抽象基类的情况正好相反，因为抽象基类并没有属于它自己的内容)

所以子 model 并不能访问它父类的 *Meta* 类。但是在某些受限的情况下，子类可以从父类继承某些 Meta ：如果子类没有指定 `ordering` 属性或 `get_latest_by` 属性，它就会从父类中继承这些属性。

如果父类有了排序设置，而你并不想让子类有任何排序设置，你就可以显式地禁用排序：

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

继承与反向关联

因为多表继承使用了一个隐含的 `OneToOneField` 来链接子类与父类，所以象上例那样，你可以用父类来指代子类。但是这个 OnetoOneField 字段默认的 `related_name` 值与 `ForeignKey` 和 `ManyToManyField` 默认的反向名称相同。如果你与其他 model 的子类做多对一或是多对多关系，你就必须在每个多对一和多对多字段上强制指定 `related_name` 。如果你没这么做，Django 就会在你运行 验证(validation) 时抛出异常。

例如，仍以上面 `Place` 类为例，我们创建一个带有 `ManyToManyField` 字段的子类：

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

这会产生一个错误：

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

像下面那样，向 `customers` 字段中添加 `related_name` 可以解决这个错误：`models.ManyToManyField(Place, related_name='provider')` 。

指定链接父类的字段

之前我们提到，Django 会自动创建一个 `OneToOneField` 字段将子类链接至非抽象的父 model 。如果你想指定链接父类的属性名称，你可以创建你自己的 `OneToOneField` 字段并设置 `parent_link=True` ，从而使用该字段链接父类。

代理模型

使用 多表继承时，model 的每个子类都会创建一张新数据表，通常情况下，这正是我们想要的操作。这是因为子类需要一个空间来存储不包含在基类中的字段数据。 但有时，你可能只想更改 model 在 Python 层的行为实现。比如：更改默认的 manager ，或是添加一个新方法。

而这，正是代理 model 继承方式要做的：为原始 model 创建一个代理 。你可以创建，删除，更新代理 model 的实例，而且所有的数据都可以象使用原始 model 一样被保存。 不同之处在于：你可以在代理 model 中改变默认的排序设置和默认的 manager ，更不会对原始 model 产生影响。

声明代理 model 和声明普通 model 没有什么不同。 设置 `Meta` 类中 `proxy` 的值为 `True` ，就完成了对代理 model 的声明。

举个例子，假设你想给 Django 自带的标准 `Person` model添加一个方法。你可以这样做：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

`MyPerson` 类和它的父类 `Person` 操作同一个数据表。特别的是， `Person` 的任何实例也可以通过 `MyPerson` 访问，反之亦然：

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

你也可以使用代理 model 给 model 定义不同的默认排序设置。 你可能并不想每次都给 `Person` 模型排序，但是使用代理的时候总是按照 `last_name` 属性排序。这非常容易：

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

现在，普通的 `Person` 查询时无序的，而 `OrderedPerson` 查询会按照 `last_name` 排序。

查询集始终返回请求的模型

也就是说，没有办法让DJango在查询 `Person` 对象时返回 `MyPerson` 对象。 `Person` 对象的查询集会返回相同类型的对象。代理对象的要点是，依赖于原生 `Person` 对象的代码仍然使用它，而你可以使用你添加进来的扩展对象（它不会依赖其它任何代码）。而并不是将 `Person` 模型（或者其它）在所有地方替换为其它你自己创建的模型。

基类的限制

代理 模型必须继承自一个非抽象基类。 你不能继承自多个非抽象基类，这是因为一个代理 model 不能连接不同的数据表。 代理 model 也可以继承任意多个抽象基类，但前提是它们没有 定义任何 model 字段。

代理模型的管理器

如果你没有在代理 模型中定义任何 管理器，代理模型就会从父类中继承 管理器。 如果你在代理 模型中定义了一个 管理器，它就会变成默认的管理器，不过定义在父类中的管理器仍然有效。

继续上面的例子，当你查询 `Person` 模型的时候，你可以改变默认 管理器，例如：

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass

class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

如果你想要向代理中添加新的管理器，而不是替换现有的默认管理器，你可以使用自定义管理器管理器文档中描述的技巧：创建一个含有新的管理器的基类，并且在主基类之后继承它：

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

你可能不需要经常这样做，但这样做是可行的。

代理 模型与非托管 模型之间的差异

代理 model 继承看上去和使用 `Meta` 类中的 `managed` 属性的非托管 model 非常相似。但两者并不相同，你应当考虑选用哪种方案。

一个不同之处是你可以在 `Meta.managed=False` 的 model 中定义字段(事实上，是必须指定，除非你真的想得到一个空 model)。在创建非托管 model 时要谨慎设置 `Meta.db_table` ，这是因为创建的非托管 model 映射某个已存在的 model ，并且有自己的方法。因此，如果你要保证这两个 model 同步并对程序进行改动，那么就会变得繁冗而脆弱。

另一个不同之处是两者对 管理器的处理方式不同。 代理 model 要与它所代理的 model 行为相似，所以代理 model 要继承父 model 的 managers ，包括它的默认 manager 。 但在普通的多表继承中，子类不能继承父类的 manager ，这是因为在处理非基类字段时，父类的 manager 未必适用。 后一种情况在 管理器文档有详细介绍。

我们实现了这两种特性之后，曾尝试把两者结合到一起。 结果证明，宏观的继承关系和微观的 管理器揉在一起，不仅导致 API 复杂难用，而且还难以理解。 由于任何场合下都可能需要这两个选项，所以目前二者仍是各自独立使用的。

所以，一般规则是：

1. 如果你要借鉴一个已有的 模型或数据表，且不想涉及所有的原始数据表的列，那就令 `Meta.managed=False` 。通常情况下， 对数据库视图创建 模型或是数据表不需要由 Django 控制时，就使用这个选项。
2. 如果你想对 model 做 Python 层级的改动，又想保留字段不变，那就令 `Meta.proxy=True` 。因此在数据保存时，代理 model 相当于完全复制了原始 模型的存储结构。

多重继承

就像Python的子类那样，DJango的模型可以继承自多个父类模型。切记一般的Python名称解析规则也会适用。出现特定名称的第一个基类(比如*Meta*)是所使用的那个。例如，这意味着如果多个父类含有 *Meta*类，只有第一个会被使用，剩下的会忽略掉。

一般来说，你并不需要继承多个父类。多重继承主要对“mix-in”类有用：向每个继承mix-in的类添加一个特定的、额外的字段或者方法。你应该尝试将你的继承关系保持得尽可能简洁和直接，这样你就不必费很大力气来弄清楚某段特定的信息来自哪里。

Changed in Django 1.7.

Django 1.7之前，继承多个含有 `id` 主键字段的模型不会抛出异常，但是会导致数据丢失。例如，考虑这些模型（由于 `id` 字段的冲突，它们不再有效）：

```
class Article(models.Model):
    headline = models.CharField(max_length=50)
    body = models.TextField()

class Book(models.Model):
    title = models.CharField(max_length=50)

class BookReview(Book, Article):
    pass
```

这段代码展示了如何创建子类的对象，并覆写之前创建的父类对象中的值。

```
>>> article = Article.objects.create(headline='Some piece of news.')
>>> review = BookReview.objects.create(
...     headline='Review of Little Red Riding Hood.',
...     title='Little Red Riding Hood')
>>>
>>> assert Article.objects.get(pk=article.pk).headline == article.headline
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError
>>> # the "Some piece of news." headline has been overwritten.
>>> Article.objects.get(pk=article.pk).headline
'Review of Little Red Riding Hood.'
```

你可以在模型基类中使用显式的 `AutoField` 来合理使用多重继承：

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

或者是使用一个公共的祖先来持有 `AutoField` ：

```
class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
    ...

class BookReview(Book, Article):
    pass
```

Field name “hiding” is not permitted

普通的 Python 类继承允许子类覆盖父类的任何属性。 但在 Django 中，重写 `Field` 实例是不允许的(至少现在还不行)。如果基类中有一个 `author` 字段，你就不能在子类中创建任何名为 `author` 的字段。

重写父类的字段会导致很多麻烦，比如：初始化实例(指定在 `Model.__init__` 中被实例化的字段) 和序列化。而普通的 Python 类继承机制并不能处理好这些特性。所以 Django 的继承机制被设计成与 Python 有所不同，这样做并不是随意而为的。

这些限制仅仅针对做为属性使用的 `Field` 实例，并不是针对 Python 属性，Python 属性仍是可以被重写的。 在 Python 看来，上面的限制仅仅针对字段实例的名称：如果你手动指定了数据库的列名称，那么在多重继承中，你就可以在子类和某个祖先类当中使用同一个列名称。(因为它们使用的是两个不同数据表的字段)。

如果你在任何一个祖先类中重写了某个 model 字段，Django 都会抛出 `FieldError` 异常。

另见

The Models Reference

Covers all the model related APIs including model fields, related objects, and `QuerySet`.

> 译者：Django 文档协作翻译小组，原文：Model syntax。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

模型元选项

这篇文档阐述了所有可用的元选项，你可以在你模型的Meta类中设置他们。

可用的元选项

abstract

Options.abstract

如果 abstract = True， 就表示模型是 抽象基类 (abstract base class).

app_label

Options.app_label

如果你的模型定义在默认的 models.py 之外(比如，你现在用的模型在 myapp.models 子模块当中)，你必须告诉 Django 该模型属于哪个应用：

```
app_label = 'myapp'
```

```
Django 1.7中新增：
一个应用中，定义在models 模块以外的模型，不再需要app_label。
```

db_table

Options.db_table

该模型所用的数据表的名称：

```
db_table = 'music_album'
```

数据表名称

为了节省时间，Django 会根据模型类的名称和包含它的app名称自动指定数据表名称，一个模型的数据表名称，由这个模型的“应用标签”（在 manage.py startapp中使用的名称）之间加上下划线组成。

举个例子， bookstore应用(使用 manage.py startapp bookstore 创建)，里面有个名为 Book 的模型，那数据表的名称就是 bookstore_book 。

使用 Meta类中的 db_table 参数来覆写数据表的名称。

数据表名称可以是 SQL 保留字，也可以包含不允许出现在 Python 变量中的特殊字符，这是因为 Django 会自动给列名和表名添加引号。

> 在 MySQL中使用小写字母为表命名
>
> 当你通过db_table覆写表名称时，强烈推荐使用小写字母给表命名，特别是如果你用了MySQL作为后端。详见MySQL注意事项 。
>
> Oracle中表名称的引号处理
>
> 为了遵从Oracle中30个字符的限制，以及一些常见的约定，Django会缩短表的名称，而且会把它全部转为大写。在db_table的值外面加上引号来避免这种情况：
>
> ```
> db_table = '"name_left_in_lowercase"'
> ```
>
> 这种带引号的名称也可以用于Django所支持的其他数据库后端，但是除了Oracle，引号不起任何作用。详见 Oracle 注意事项 。

db_tablespace

Options.db_tablespace

当前模型所使用的数据库表空间 的名字。默认值是项目设置中的DEFAULT_TABLESPACE，如果它存在的话。如果后端并不支持表空间，这个选项可以忽略。

default_related_name

Options.default_related_name

```
Django 1.8中新增：
```

这个名字会默认被用于一个关联对象到当前对象的关系。默认为 _set。

由于一个字段的反转名称应该是唯一的，当你给你的模型设计子类时，要格外小心。为了规避名称冲突，名称的一部分应该含有'%(app_label)s'和'%(model_name)s'，它们会被应用标签的名称和模型的名称替换，二者都是小写的。详见抽象模型的关联名称。

get_latest_by

Options.get_latest_by

模型中某个可排序的字段的名称，比如DateField、DateTimeField或者IntegerField。它指定了Manager的latest()和earliest()中使用的默认字段。

例如：

```
get_latest_by = "order_date"
```

详见latest() 文档。

managed

Options.managed

默认为True，意思是Django在migrate命令中创建合适的数据表，并且会在 flush 管理命令中移除它们。换句话说，Django会管理这些数据表的生命周期。

如果是False，Django 就不会为当前模型创建和删除数据表。如果当前模型表示一个已经存在的，通过其它方法建立的数据库视图或者数据表，这会相当有用。这是设置为managed=False时唯一的不同之处。. 模型处理的其它任何方面都和平常一样。这包括：

- 如果你不声明它的话，会向你的模型中添加一个自增主键。为了避免给后面的代码读者带来混乱，强烈推荐你在使用未被管理的模型时，指定数据表中所有的列。
- 如果一个带有managed=False的模型含有指向其他未被管理模型的ManyToManyField，那么多对多连接的中介表也不会被创建。但是，一个被管理模型和一个未被管理模型之间的中介表会被创建。

如果你需要修改这一默认行为，创建中介表作为显式的模型（设置为managed），并且使用ManyToManyField.through为你的自定义模型创建关联。

对于带有managed=False的模型的测试，你要确保在测试启动时建立正确的表。

如果你对修改模型类在Python层面的行为感兴趣，你可以设置 managed=False ，并且创建一个已经存在模型的部分。但是这种情况下使用代理模型才是更好的方法。

order_with_respect_to

Options.order_with_respect_to

按照给定的字段把这个对象标记为”可排序的“。这一属性通常用到关联对象上面，使它在父对象中有序。比如，如果Answer和 Question相关联，一个问题有至少一个答案，并且答案的顺序非常重要，你可以这样做：

```
from django.db import models

class Question(models.Model):
    text = models.TextField()
    # ...

class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

当order_with_respect_to 设置之后，模型会提供两个用于设置和获取关联对象顺序的方法：get_RELATED_order() 和set_RELATED_order()，其中RELATED是小写的模型名称。例如，假设一个 Question 对象有很多相关联的Answer对象，返回的列表中含有相关联Answer对象的主键：

```
>>> question = Question.objects.get(id=1)
>>> question.get_answer_order()
[1, 2, 3]
```

与Question对象相关联的Answer对象的顺序，可以通过传入一格包含Answer 主键的列表来设置：

```
>>> question.set_answer_order([3, 1, 2])
```

相关联的对象也有两个方法， get_next_in_order() 和get_previous_in_order()，用于按照合适的顺序访问它们。假设Answer对象按照 id来排序：

```
>>> answer = Answer.objects.get(id=2)
>>> answer.get_next_in_order()
<Answer: 3>
>>> answer.get_previous_in_order()
<Answer: 1>
```

> 修改 order_with_respect_to
>
> order_with_respect_to属性会添加一个额外的字段（/数据表中的列）叫做_order，所以如果你在首次迁移之后添加或者修改了order_with_respect_to属性，要确保执行和应用了合适的迁移操作。

ordering

Options.ordering

对象默认的顺序，获取一个对象的列表时使用：

```
ordering = ['-order_date']
```

它是一个字符串的列表或元组。每个字符串是一个字段名，前面带有可选的“-”前缀表示倒序。前面没有“-”的字段表示正序。使用"?"来表示随机排序。

例如，要按照pub_date字段的正序排序，这样写：

```
ordering = ['pub_date']
```

按照pub_date字段的倒序排序，这样写：

```
ordering = ['-pub_date']
```

先按照pub_date的倒序排序，再按照 author 的正序排序，这样写：

```
ordering = ['-pub_date', 'author']
```

> 警告
>
> 排序并不是没有任何代价的操作。你向ordering属性添加的每个字段都会产生你数据库的开销。你添加的每个外键也会隐式包含它的默认顺序。

permissions

Options.permissions

设置创建对象时权限表中额外的权限。增加、删除和修改权限会自动为每个模型创建。这个例子指定了一种额外的权限，can_deliver_pizzas：

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

它是一个包含二元组的元组或者列表，格式为 (permission_code, human_readable_permission_name)。

default_permissions

Options.default_permissions

```
Django 1.7中新增：
```

默认为('add', 'change', 'delete')。你可以自定义这个列表，比如，如果你的应用不需要默认权限中的任何一项，可以把它设置成空列表。在模型被migrate命令创建之前，这个属性必须被指定，以防一些遗漏的属性被创建。

proxy

Options.proxy

如果proxy = True, 作为该模型子类的另一个模型会被视为代理模型。

select_on_save

Options.select_on_save

该选项决定了Django是否采用1.6之前的 django.db.models.Model.save()算法。旧的算法使用SELECT来判断是否存在需要更新的行。而新式的算法直接尝试使用 UPDATE。在一些小概率的情况中，一个已存在的行的UPDATE操作并不对Django可见。比如PostgreSQL的ON UPDATE触发器会返回NULL。这种情况下，新式的算法会在最后执行 INSERT 操作，即使这一行已经在数据库中存在。

通常这个属性不需要设置。默认为False。

关于旧式和新式两种算法，请参见django.db.models.Model.save()。

unique_together

Options.unique_together

用来设置的不重复的字段组合：

```
unique_together = (("driver", "restaurant"),)
```

它是一个元组的元组，组合起来的时候必须是唯一的。它在Django后台中被使用，在数据库层上约束数据(比如，在 CREATE TABLE 语句中包含 UNIQUE语句)。

为了方便起见，处理单一字段的集合时，unique_together 可以是一维的元组：

```
unique_together = ("driver", "restaurant")
```

ManyToManyField不能包含在unique_together中。（这意味着什么并不清楚！）如果你需要验证ManyToManyField关联的唯一性，试着使用信号或者显式的贯穿模型(explicit through model)。

```
Django 1.7中修改：
当unique_together的约束被违反时，模型校验期间会抛出ValidationError异常。
```

index_together

Options.index_together

用来设置带有索引的字段组合：

```
index_together = [
    ["pub_date", "deadline"],
]
```

列表中的字段将会建立索引（例如，会在CREATE INDEX语句中被使用）。

```
Django 1.7中修改：
```

为了方便起见，处理单一字段的集合时，index_together可以是一个一维的列表。

```
index_together = ["pub_date", "deadline"]
```

verbose_name

Options.verbose_name

对象的一个易于理解的名称，为单数：

```
verbose_name = "pizza"
```

如果此项没有设置，Django会把类名拆分开来作为自述名，比如CamelCase 会变成camel case，

verbose_name_plural

Options.verbose_name_plural

该对象复数形式的名称：

```
verbose_name_plural = "stories"
```

如果此项没有设置，Django 会使用 verbose_name + "s"。

Model 类参考

这篇文档覆盖 `Model` 类的特性。关于模型的更多信息，参见Model 完全参考指南。

属性

objects

`Model.objects`

每个非抽象的 `Model` 类必须给自己添加一个 `Manager` 实例。Django 确保在你的模型类中至少有一个默认的 `Manager`。如果你没有添加自己的 `Manager`，Django 将添加一个属性 `objects`，它包含默认的 `Manager` 实例。如果你添加自己的 `Manager` 实例的属性，默认值则不会出现。思考一下下面的例子：

```
from django.db import models

class Person(models.Model):
    # Add manager with another name
    people = models.Manager()
```

关于模型管理器的更多信息，参见Managers 和 Retrieving objects。

> 译者：Django 文档协作翻译小组，原文：Model class。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

查询集

执行查询

一旦你建立好数据模型之后，django会自动生成一套数据库抽象的API，可以让你执行增删改查的操作。这篇文档阐述了如何使用这些API。关于所有模型检索选项的详细内容，请见数据模型参考。

在整个文档（以及参考）中，我们会大量使用下面的模型，它构成了一个博客应用。

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

创建对象

为了把数据库表中的数据表示成python对象，django使用一种直观的方式：一个模型类代表数据库的一个表，一个模型的实例代表数据库表中的一条特定的记录。

使用关键词参数实例化一个对象来创建它，然后调用**save()**把它保存到数据库中。

假设模型存放于文件**mysite/blog/models.py**中，下面是一个例子：

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

上面的代码在背后执行了sql的**INSERT**操作。在你显式调用**save()**之前，django不会访问数据库。

save()方法没有返回值。

> 请参见
>
> **save()**方法带有一些高级选项，它们没有在这里给出，完整的细节请见**save()**文档。
>
> 如果你想只用一条语句创建并保存一个对象，使用**create()**方法。

保存对象的改动

调用**save()**方法，来保存已经存在于数据库中的对象的改动。

假设一个**Blog**的实例**b5**已经被保存在数据库中，这个例子更改了它的名字，并且在数据库中更新它的记录：

```
>>> b5.name = 'New name'
>>> b5.save()
```

上面的代码在背后执行了sql的**UPDATE**操作。在你显式调用**save()**之前，django不会访问数据库。

保存**ForeignKey**和**ManyToManyField**字段

更新**ForeignKey**字段的方式和保存普通字段相同--只是简单地把一个类型正确的对象赋值到字段中。下面的例子更新了**Entry**类的实例**entry**的**blog**属性，假设**Entry**的一个合适的实例以及**Blog**已经保存在数据库中（我们可以像下面那样获取他们）：

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

更新**ManyToManyField**的方式有一些不同--使用字段的**add()**方法来增加关系的记录。这个例子向**entry**对象添加**Author**类的实例**joe**：

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

为了在一条语句中，向**ManyToManyField**添加多条记录，可以在调用**add()**方法时传入多个参数，像这样：

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django将会在你添加错误类型的对象时抛出异常。

获取对象

通过模型中的**Manager**构造一个**QuertSet**，来从你的数据库中获取对象。

QuerySet表示你数据库中取出来的一个对象的集合。它可以含有零个、一个或者多个过滤器，过滤器根据所给的参数限制查询结果的范围。在sql的角度，**QuerySet**和**SELECT**命令等价，过滤器是像**WHERE**和**LIMIT**一样的限制子句。

你可以从模型的**Manager**那里取得**QuerySet**。每个模型都至少有一个**Manager**，它通常命名为**objects**。通过模型类直接访问它，像这样：

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

> 注意
>
> 管理器通常只可以通过模型类来访问，不可以通过模型实例来访问。这是为了强制区分表级别和记录级别的操作。

对于一个模型来说，**Manager**是**QuerySet**的主要来源。例如， **Blog.objects.all()** 会返回持有数据库中所有**Blog**对象的一个**QuerySet**。

获取所有对象

获取一个表中所有对象的最简单的方式是全部获取。使用**Manager**的**all()**方法：

```
>>> all_entries = Entry.objects.all()
```

all()方法返回包含数据库中所有对象的**QuerySet**。

使用过滤器获取特定对象

all()方法返回的结果集中包含全部对象，但是更普遍的情况是你需要获取完整集合的一个子集。

要创建这样一个子集，需要精炼上面的结果集，增加一些过滤器作为条件。两个最普遍的途径是：

filter(kwargs)** 返回一个包含对象的集合，它们满足参数中所给的条件。

exclude(kwargs)** 返回一个包含对象的集合，它们不满足参数中所给的条件。

查询参数（上面函数定义中的**kwargs）需要满足特定的格式，字段检索一节中会提到。

举个例子，要获取年份为2006的所有文章的结果集，可以这样使用**filter()**方法：

```
Entry.objects.filter(pub_date__year=2006)
```

在默认的管理器类中，它相当于：

```
Entry.objects.all().filter(pub_date__year=2006)
```

链式过滤

QuerySet的精炼结果还是**QuerySet**，所以你可以把精炼用的语句组合到一起，像这样：

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 30)
... )
```

最开始的**QuerySet**包含数据库中的所有对象，之后增加一个过滤器去掉一部分，在之后又是另外一个过滤器。最后的结果的一个**QuerySet**，包含所有标题以”word“开头的记录，并且日期是2005年一月，日为当天的值。

过滤后的结果集是独立的

每次你筛选一个结果集，得到的都是全新的另一个结果集，它和之前的结果集之间没有任何绑定关系。每次筛选都会创建一个独立的结果集，可以被存储及反复使用。

例如：

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

这三个 QuerySets 是不同的。 第一个 QuerySet 包含大标题以"What"开头的所有记录。第二个则是第一个的子集，用一个附加的条件排除了出版日期 pub_date 是今天的记录。 第三个也是第一个的子集，它只保留出版日期 pub_date 是今天的记录。 最初的 QuerySet (q1) 没有受到筛选的影响。

查询集是延迟的

QuerySets 是惰性的 -- 创建 QuerySet 的动作不涉及任何数据库操作。你可以一直添加过滤器，在这个过程中，Django 不会执行任何数据库查询，除非 QuerySet 被执行. 看看下面这个例子:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.now())
>>> q = q.exclude(body_text__icontains="food")
>>> print q
```

虽然上面的代码看上去象是三个数据库操作，但实际上只在最后一行 (print q) 执行了一次数据库操作，。一般情况下， QuerySet 不能从数据库中主动地获得数据，得被动地由你来请求。对 QuerySet 求值就意味着 Django 会访问数据库。想了解对查询集何时求值，请查看 何时对查询集求值 (When QuerySets are evaluated).

其他查询集方法

大多数情况使用 all(), filter() 和 exclude() 就足够了。 但也有一些不常用的；请查看 查询API 参考 (QuerySet API Reference) 中完整的 QuerySet 方法列表。

限制查询集范围

可以用 python 的数组切片语法来限制你的 QuerySet 以得到一部分结果。它等价于SQL中的 LIMIT 和 OFFSET 。

例如，下面的这个例子返回前五个对象 (LIMIT 5):

```
>>> Entry.objects.all()[:5]
```

这个例子返回第六到第十之间的对象 (OFFSET 5 LIMIT 5):

```
>>> Entry.objects.all()[5:10]
```

Django 不支持对查询集做负数索引 (例如 Entry.objects.all()[-1]) 。

一般来说，对 QuerySet 切片会返回新的 QuerySet -- 这个过程中不会对运行查询。不过也有例外，如果你在切片时使用了 "step" 参数，查询集就会被求值，就在数据库中运行查询。举个例子，使用下面这个这个查询集返回前十个对象中的偶数次对象，就会运行数据库查询:

```
>>> Entry.objects.all()[:10:2]
```

要检索单独的对象，而非列表 (比如 SELECT foo FROM bar LIMIT 1)，可以直接使用索引来代替切片。举个例子，下面这段代码将返回大标题排序后的第一条记录 Entry:

```
>>> Entry.objects.order_by('headline')[0]
```

大约等价于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

要注意的是：如果找不到符合条件的对象，第一种方法会抛出 IndexError ，而第二种方法会抛出 DoesNotExist。详看 get() 。

字段筛选条件

字段筛选条件就是 SQL 语句中的 WHERE 从句。就是 Django 中的 QuerySet 的 filter(), exclude() 和 get() 方法中的关键字参数。

筛选条件的形式是 field__lookuptype=value 。 (注意：这里是双下划线)。例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

大体可以翻译为如下的 SQL 语句:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

这是怎么办到的？

Python 允许函式接受任意多 name-value 形式的参数，并在运行时才确定name和value的值。详情请参阅官方Python教程中的 关键字参数(Keyword Arguments)。

如果你传递了一个无效的关键字参数，会抛出 TypeError 导常。

数据库 API 支持24种查询类型；可以在 字段筛选参考(field lookup reference) 查看详细的列表。为了给您一个直观的认识，这里我们列出一些常用的查询类型：

exact

"exact" 匹配。例如：

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

会生成如下的 SQL 语句：

```
SELECT ... WHERE headline = 'Man bites dog';
```

如果你没有提供查询类型 -- 也就是说关键字参数中没有双下划线，那么查询类型就会被指定为 exact。

举个例子，这两个语句是相等的：

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)         # __exact is implied
```

这样做很方便，因为 exact 是最常用的。

iexact

忽略大小写的匹配。所以下面的这个查询:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

会匹配标题是 "Beatles Blog", "beatles blog", 甚至 "BeAtlES blOG" 的 Blog

contains

大小写敏感的模糊匹配。 例如：

```
Entry.objects.get(headline__contains='Lennon')
```

大体可以翻译为如下的 SQL：

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

要注意这段代码匹配大标题 'Today Lennon honored' ， 而不能匹配 'today lennon honored'。

它也有一个忽略大小写的版本，就是 icontains。

startswith, endswith

分别匹配开头和结尾，同样也有忽略大小写的版本 istartswith 和 iendswith。 再强调一次，这仅仅是简短介绍。完整的参考请参见 字段筛选条件参考(field lookup reference)。

跨关系查询

Django 提供了一种直观而高效的方式在查询(lookups)中表示关联关系，它能自动确认 SQL JOIN 联系。要做跨关系查询，就使用两个下划线来链接模型(model)间关联字段的名称，直到最终链接到你想要的 model 为止。

这个例子检索所有关联 Blog 的 name 值为 'Beatles Blog' 的所有 Entry 对象：

```
>>> Entry.objects.filter(blog__name__exact='Beatles Blog')
```

跨关系的筛选条件可以一直延展。

关系也是可逆的。可以在目标 model 上使用源 model 名称的小写形式得到反向关联。

下面这个例子检索至少关联一个 Entry 且大标题 headline 包含 'Lennon' 的所有 Blog 对象：

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

如果在某个关联 model 中找不到符合过滤条件的对象，Django 将视它为一个空的 (所有的值都是 NULL), 但是可用的对象。这意味着不会有异常抛出，在这个例子中：

```
Blog.objects.filter(entry__author__name='Lennon')
```

(假设关联到 Author 类), 如果没有哪个 author 与 entry 相关联，Django 会认为它没有 name 属性，而不会因为不存在 author 抛出异常。通常来说，这正是你所希望的机制。唯一的例外是使用 isnull 的情况。如下:

```
Blog.objects.filter(entry__author__name__isnull=True)
```

这段代码会得到 author 的 name 为空的 Blog 或 entry 的 author为空的 Blog。 如果不嫌麻烦，可以这样写：

```
Blog.objects.filter (entry__author__isnull=False,
        entry__author__name__isnull=True)
```

跨一对多／多对多关系(Spanning multi-valued relationships)

这部分是Django 1.0中新增的： 请查看版本记录 如果你的过滤是基于 ManyToManyField 或是逆向 ForeignKeyField 的，你可能会对下面这两种情况感兴趣。回顾 Blog/Entry 的关系 (Blog 到 Entry 是一对多关系)，如果要查找这样的 blog：它关联一个大标题包含"Lennon"，且在2008年出版的 entry ；或者要查找这样的 blogs：它关联一个大标题包含"Lennon"的 entry ，同时它又关联另外一个在2008年出版的 entry 。因为一个 Blog 会关联多个的Entry，所以上述两种情况在现实应用中是很有可能出现的。

同样的情形也出现在 ManyToManyField 上。例如，如果 Entry 有一个 ManyToManyField 字段，名字是 tags，我们想得到 tags 是"music"和"bands"的 entries，或者我们想得到包含名为"music" 的标签而状态是"public"的 entry。

针对这两种情况，Django 用一种很方便的方式来使用 filter() 和 exclude()。对于包含在同一个 filter() 中的筛选条件，查询集要同时满足所有筛选条件。而对于连续的 filter()，查询集的范围是依次限定的。但对于跨一对多／多对多关系查询来说，在第二种情况下，筛选条件针对的是主 model 所有的关联对象，而不是被前面的 filter() 过滤后的关联对象。

这听起来会让人迷糊，举个例子会讲得更清楚。要检索这样的 blog：它要关系一个大标题中含有 "Lennon" 并且在2008年出版的 entry (这个 entry 同时满足这两个条件)，可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon',
        entry__pub_date__year=2008)
```

要检索另外一种 blog：它关联一个大标题含有"Lennon"的 entry，又关联一个在2008年出版的 entry （一个 entry 的大标题含有 Lennon，同一个或另一个 entry 是在2008年出版的）。可以这样写：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
        entry__pub_date__year=2008)
```

在第二个例子中，第一个过滤器(filter)先检索与符合条件的 entry 的相关联的所有 blogs。第二个过滤器在此基础上从这些 blogs 中检索与第二种 entry 也相关联的 blog。第二个过滤器选择的 entry 可能与第一个过滤器所选择的完全相同，也可能不同。 因为过滤项过滤的是 Blog，而不是 Entry。

上述原则同样适用于 exclude()：一个单独 exclude() 中的所有筛选条件都是作用于同一个实例 (如果这些条件都是针对同一个一对多／多对多的关系)。连续的 filter() 或 exclude() 却根据同样的筛选条件，作用于不同的关联对象。

在过滤器中引用 model 中的字段(Filters can reference fields on the model)

这部分是 Django 1.1 新增的: 请查看版本记录 在上面所有的例子中，我们构造的过滤器都只是将字段值与某个常量做比较。如果我们要对两个字段的值做比较，那该怎么做呢？

Django 提供 F() 来做这样的比较。F() 的实例可以在查询中引用字段，来比较同一个 model 实例中两个不同字段的值。

例如：要查询回复数(comments)大于广播数(pingbacks)的博文(blog entries)，可以构造一个 F() 对象在查询中引用评论数量：

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments'))
```

Django 支持 F() 对象之间以及 F() 对象和常数之间的加减乘除和取模的操作。例如，要找到广播数等于评论数两倍的博文，可以这样修改查询语句：

```
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments') * 2)
```

要查找阅读数量小于评论数与广播数之和的博文，查询如下:

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

你也可以在 F() 对象中使用两个下划线做跨关系查询。F() 对象使用两个下划线引入必要的关联对象。例如，要查询博客(blog)名称与作者(author)名称相同的博文(entry)，查询就可以这样写：

```
>>> Entry.objects.filter(author__name=F('blog__name'))
```

主键查询的简捷方式

为使用方便考虑，Django 用 pk 代表主键"primary key"。

以 Blog 为例, 主键是 id 字段，所以下面三个语句都是等价的：

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

pk 对 __exact 查询同样有效，任何查询项都可以用 pk 来构造基于主键的查询：

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

pk 查询也可以跨关系，下面三个语句是等价的：

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)        # __exact is implied
>>> Entry.objects.filter(blog__pk=3)        # __pk implies __id__exact
```

在LIKE语句中转义百分号%和下划线_

字段筛选条件相当于 LIKE SQL 语句 (iexact, contains, icontains, startswith, istartswith, endswith 和 iendswith)，它会自动转义两个特殊符号 -- 百分号%和下划线。*(*在 *LIKE* 语句中，百分号%表示多字符匹配，而下划线表示单字符匹配。)

这就意味着我们可以直接使用这两个字符，而不用考虑他们的 SQL 语义。例如，要查询大标题中含有一个百分号%的 entry：

```
>>> Entry.objects.filter(headline__contains='%')
```

Django 会处理转义；最终的 SQL 看起来会是这样：

```
SELECT ... WHERE headline LIKE '%\%%';
```

下划线_和百分号%的处理方式相同，Django 都会自动转义。

缓存和查询

每个 QuerySet 都包含一个缓存，以减少对数据库的访问。要编写高效代码，就要理解缓存是如何工作的。

一个 QuerySet 时刚刚创建的时候，缓存是空的。 QuerySet 第一次运行时，会执行数据库查询，接下来 Django 就在 QuerySet 的缓存中保存查询的结果，并根据请求返回这些结果(比如，后面再次调用这个 QuerySet 的时候)。再次运行 QuerySet 时就会重用这些缓存结果。

要牢住上面所说的缓存行为，否则在使用 QuerySet 时可能会给你造成不小的麻烦。例如，创建下面两个 QuerySet，并对它们求值，然后释放：

```
>>> print [e.headline for e in Entry.objects.all()]
>>> print [e.pub_date for e in Entry.objects.all()]
```

这就意味着相同的数据库查询将执行两次，事实上读取了两次数据库。而且，这两次读出来的列表可能并不完全相同，因为存在这种可能：在两次读取之间，某个 Entry 被添加到数据库中，或是被删除了。

要避免这个问题，只要简单地保存 QuerySet 然后重用即可：

```
>>> queryset = Poll.objects.all()
>>> print [p.headline for p in queryset] # Evaluate the query set.
>>> print [p.pub_date for p in queryset] # Re-use the cache from the evaluation.
```

用 Q 对象实现复杂查找 (Complex lookups with Q objects)

在 filter() 等函式中关键字参数彼此之间都是 "AND" 关系。如果你要执行更复杂的查询(比如，实现筛选条件的 OR 关系)，可以使用 Q 对象。

Q 对象(django.db.models.Q)是用来封装一组查询关键字的对象。这里提到的查询关键字请查看上面的 "Field lookups"。

例如，下面这个 Q 对象封装了一个单独的 LIKE 查询：

```
Q(question__startswith='What')
```

Q 对象可以用 & 和 | 运算符进行连接。当某个操作连接两个 Q 对象时，就会产生一个新的等价的 Q 对象。

例如，下面这段语句就产生了一个 Q，这是用 "OR" 关系连接的两个 "question__startswith" 查询：

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

上面的例子等价于下面的 SQL WHERE 从句：

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

你可以用 & 和 | 连接任意多的 Q 对象，而且可以用括号分组。Q 对象也可以用 ~ 操作取反，而且普通查询和取反查询(NOT)可以连接在一起使用：

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

每种查询函式(比如 filter(), exclude(), get()) 除了能接收关键字参数以外，也能以位置参数的形式接受一个或多个 Q 对象。如果你给查询函式传递了多个 Q 对象，那么它们彼此间都是 "AND" 关系。例如：

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... 大体可以翻译为下面的 SQL：

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

查找函式可以混用 Q 对象和关键字参数。查询函式的所有参数(Q 关系和关键字参数) 都是 "AND" 关系。但是，如果参数中有 Q 对象，它必须排在所有的关键字参数之前。例如：

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')
```

... 是一个有效的查询。但下面这个查询虽然看上去和前者等价：

```
# INVALID QUERY
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

... 但这个查询却是无效的。

> 参见
>
> 在 Django 的单元测试 OR查询实例(OR lookups examples) 中展示了 Q 的用例。

对象比较

要比较两个对象，就和 Python 一样，使用双等号运算符：==。实际上比较的是两个 model 的主键值。

以上面的 Entry 为例，下面两个语句是等价的：

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

如果 model 的主键名称不是 id，也没关系。Django 会自动比较主键的值，而不管他们的名称是什么。例如，如果一个 model 的主键字段名称是 name，那么下面两个语句是等价的：

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

对象删除

删除方法就是 delete()。它运行时立即删除对象而不返回任何值。例如：

```
e.delete()
```

你也可以一次性删除多个对象。每个 QuerySet 都有一个 delete() 方法，它一次性删除 QuerySet 中所有的对象。

例如，下面的代码将删除 pub_date 是2005年的 Entry 对象：

```
Entry.objects.filter(pub_date__year=2005).delete()
```

要牢记这一点：无论在什么情况下，QuerySet 中的 delete() 方法都只使用一条 SQL 语句一次性删除所有对象，而并不是分别删除每个对象。如果你想使用在 model 中自定义的 delete() 方法，就要自行调用每个对象的delete 方法。(例如，遍历 QuerySet，在每个对象上调用 delete()方法)，而不是使用 QuerySet 中的 delete()方法。

在 Django 删除对象时，会模仿 SQL 约束 ON DELETE CASCADE 的行为，换句话说，删除一个对象时也会删除与它相关联的外键对象。例如：

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

要注意的是： delete() 方法是 QuerySet 上的方法，但并不适用于 Manager 本身。这是一种保护机制，是为了避免意外地调用 Entry.objects.delete() 方法导致 所有的 记录被误删除。如果你确认要删除所有的对象，那么你必须显式地调用：

```
Entry.objects.all().delete()
```

一次更新多个对象 (Updating multiple objects at once)

这部分是 Django 1.0 中新增的： 请查看版本文档 有时你想对 QuerySet 中的所有对象，一次更新某个字段的值。这个要求可以用 update() 方法完成。例如：

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

这种方法仅适用于非关系字段和 ForeignKey 外键字段。更新非关系字段时，传入的值应该是一个常量。更新 ForeignKey 字段时，传入的值应该是你想关联的那个类的某个实例。例如：

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.all().update(blog=b)
```

update() 方法也是即时生效，不返回任何值的(与 delete() 相似)。 在 QuerySet 进行更新时，唯一的限制就是一次只能更新一个数据表，就是当前 model 的主表。所以不要尝试更新关联表和与此类似的操作，因为这是不可能运行的。

要小心的是： update() 方法是直接翻译成一条 SQL 语句的。因此它是直接地一次完成所有更新。它不会调用你的 model 中的 save() 方法，也不会发出 pre_save 和 post_save 信号(这些信号在调用 save() 方法时产生)。如果你想保存 QuerySet 中的每个对象，并且调用每个对象各自的 save() 方法，那么你不必另外多写一个函式。只要遍历这些对象，依次调用 save() 方法即可：

```
for item in my_queryset:
    item.save()
```

这部分是在 Django 1.1 中新增的： 请查看版本文档 在调用 update 时可以使用 F() 对象 来把某个字段的值更新为另一个字段的值。这对于自增记数器是非常有用的。例如，给所有的博文 (entry) 的广播数 (pingback) 加一：

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

但是，与 F() 对象在查询时所不同的是，在filter 和 exclude子句中，你不能在 F() 对象中引入关联关系(NO-Join)，你只能引用当前 model 中要更新的字段。如果你在 F() 对象引入了Join关系object，就会抛出 FieldError 异常：

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

对象关联

当你定义在 model 定义关系时 (例如， ForeignKey, OneToOneField, 或 ManyToManyField)，model 的实例自带一套很方便的API以获取关联的对象。

以最上面的 models 为例，一个 Entry 对象 e 能通过 blog 属性获得相关联的 Blog 对象：e.blog。

(在场景背后，这个功能是由 Python 的 descriptors 实现的。如果你对此感兴趣，可以了解一下。)

Django 也提供反向获取关联对象的 API，就是由从被关联的对象得到其定义关系的主对象。例如，一个 Blog 类的实例 b 对象通过 entry_set 属性得到所有相关联的 Entry 对象列表: b.entry_set.all()。

这一节所有的例子都使用本页顶部所列出的 Blog, Author 和 Entry model。

一对多关系

正向

如果一个 model 有一个 ForeignKey字段，我们只要通过使用关联 model 的名称就可以得到相关联的外键对象。

例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

你可以设置和获得外键属性。正如你所期望的，改变外键的行为并不引发数据库操作，直到你调用 save()方法时，才会保存到数据库。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

如果外键字段 ForeignKey 有一个 null=True 的设置(它允许外键接受空值 NULL)，你可以赋给它空值 None 。例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

在一对多关系中，第一次正向获取关联对象时，关联对象会被缓存。其后根据外键访问时这个实例，就会从缓存中获得它。例如：

```
>>> e = Entry.objects.get(id=2)
>>> print e.blog  # Hits the database to retrieve the associated Blog.
>>> print e.blog  # Doesn't hit the database; uses cached version.
```

要注意的是，QuerySet 的 select_related() 方法提前将所有的一对多关系放入缓存中。例如：

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print e.blog  # Doesn't hit the database; uses cached version.
>>> print e.blog  # Doesn't hit the database; uses cached version.
```

逆向关联

如果 model 有一个 ForeignKey外键字段，那么外联 model 的实例可以通过访问 Manager 来得到所有相关联的源 model 的实例。默认情况下，这个 Manager 被命名为 FOO_set, 这里面的 FOO 就是源 model 的小写名称。这个 Manager 返回 QuerySets，它是可过滤和可操作的，在上面 "对象获取(Retrieving objects)" 有提及。

例如：

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

你可以通过在 ForeignKey() 的定义中设置 related_name 的值来覆写 FOO_set 的名称。例如，如果 Entry model 中做一下更改： blog = ForeignKey(Blog, related_name='entries')，那么接下来就会如我们看到这般：

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

你不能在一个类当中访问 ForeignKey Manager ；而必须通过类的实例来访问：

```
>>> Blog.entry_set
Traceback:
    ...
AttributeError: "Manager must be accessed via instance".
```

除了在上面 "对象获取Retrieving objects" 一节中提到的 QuerySet 方法之外，ForeignKey Manager 还有如下一些方法。下面仅仅对它们做一个简短介绍，详情请查看 related objects reference。

`add(obj1, obj2, ...)`

将某个特定的 model 对象添加到被关联对象集合中。

`create(**kwargs)`

创建并保存一个新对象，然后将这个对象加被关联对象的集合中，然后返回这个新对象。

`remove(obj1, obj2, ...)`

将某个特定的对象从被关联对象集合中去除。

`clear()`

清空被关联对象集合。 想一次指定关联集合的成员，那么只要给关联集合分配一个可迭代的对象即可。它可以包含对象的实例，也可以只包含主键的值。例如：

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

在这个例子中，e1 和 e2 可以是完整的 Entry 实例，也可以是整型的主键值。

如果 clear() 方法是可用的，在迭代器(上例中就是一个列表)中的对象加入到 entry_set 之前，已存在于关联集合中的所有对象将被清空。如果 clear() 方法 不可用，原有的关联集合中的对象就不受影响，继续存在。

这一节提到的每一个 "reverse" 操作都是实时操作数据库的，每一个添加，创建，删除操作都会及时保存将结果保存到数据库中。

多对多关系

在多对多关系的任何一方都可以使用 API 访问相关联的另一方。多对多的 API 用起来和上面提到的 "逆向" 一对多关系关系非常相象。

唯一的差虽就在于属性的命名： ManyToManyField 所在的 model (为了方便，我称之为源 model A) 使用字段本身的名称来访问关联对象；而被关联的另一方则使用 A 的小写名称加上 '_set' 后缀(这与逆向的一对多关系非常相象)。

下面这个例子会让人更容易理解：

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

与 ForeignKey 一样, ManyToManyField 也可以指定 related_name。在上面的例子中，如果 Entry 中的 ManyToManyField 指定 related_name='entries'，那么接下来每个 Author 实例的 entry_set 属性都被 entries 所代替。

一对一关系

相对于多对一关系而言，一对一关系不是非常简单的。如果你在 model 中定义了一个 OneToOneField 关系，那么你就可以用这个字段的名称做为属性来访问其所关联的对象。

例如：

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

与 "reverse" 查询不同的是，一对一关系的关联对象也可以访问 Manager 对象，但是这个 Manager 表现一个单独的对象，而不是一个列表：

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

如果一个空对象被赋予关联关系，Django 就会抛出一个 DoesNotExist 异常。

和你定义正向关联所用的方式一样，类的实例也可以赋予逆向关联方系：

```
e.entrydetail = ed
```

关系中的反向连接是如何做到的？

其他对象关系的映射(ORM)需要你在关联双方都定义关系。而 Django 的开发者则认为这违背了 DRY 原则 (Don't Repeat Yourself)，所以 Django 只需要你在一方定义关系即可。

但仅由一个 model 类并不能知道其他 model 类是如何与它关联的，除非是其他 model 也被载入，那么这是如何办到的？

答案就在于 INSTALLED_APPS 设置中。任何一个 model 在第一次调用时，Django 就会遍历所有的 INSTALLED_APPS 的所有 models，并且在内存中创建中必要的反向连接。本质上来说，INSTALLED_APPS 的作用之一就是确认 Django 完整的 model 范围。

在关联对象上的查询

包含关联对象的查询与包含普通字段值的查询都遵循相同的规则。为某个查询指定某个值的时候，你可以使用一个类实例，也可以使用对象的主键值。

例如，如果你有一个 Blog 对象 b ，它的 id=5, 下面三个查询是一样的：

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

直接使用**SQL**

如果你发现某个 SQL 查询用 Django 的数据库映射来处理会非常复杂的话，你可以使用直接写 SQL 来完成。

建议的方式是在你的 model 自定义方法或是自定义 model 的 manager 方法来运行查询。虽然 Django 不要求数据操作必须在 model 层中执行。但是把你的商业逻辑代码放在一个地方，从代码组织的角度来看，也是十分明智的。详情请查看 执行原生SQL查询(Performing raw SQL queries).

最后，要注意的是，Django的数据操作层仅仅是访问数据库的一个接口。你可以用其他的工具，编程语言，数据库框架来访问数据库。对你的数据库而言，没什么是非用 Django 不可的。

> 译者：Django 文档协作翻译小组，原文：Executing queries。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

查找 API 参考

```
New in Django 1.7.
```

这篇文档是查找 API 的参考，Django 用这些API 构建数据库查询的 `WHERE` 子句。若要学习如何使用 查找，参见执行查询；若要了解如何创建 新的查找，参见自定义查找。

查找 API 由两个部分组成： `RegisterLookupMixin` 类，它用于注册查找；查询表达式API，它是一个方法集，类必须实现它们才可以注册成一个查找。

Django 有两个类遵循查询表达式API，且Django 所有内建的查找都继承自它们：

- `Lookup` ：用于查找一个字段（例如 `field_name__exact` 中的 `exact` ）
- `Transform` ：用于转换一个字段

查找表达式由三部分组成：

- 字段部分（例如， `Book.objects.filter(author__best_friends__first_name...)` ；
- 转换部分（可以省略）（例如， `__lower__first3chars__reversed` ）；
- 查找部分（例如， `__icontains` ），如果省略则默认为 `__exact` 。

注册 API

Django 使用 `RegisterLookupMixin` 来为类提供接口，注册它自己的查找。两个最突出的例子是 `Field` （所有模型字段的基类）和 `Aggregate` （Django 所有聚合函数的基类）。

`class lookups.RegisterLookupMixin`

一个mixin，实现一个类上的查找API。

`classmethod register_lookup(lookup)`

在类中注册一个新的查找。例如， `DateField.register_lookup(YearExact)` 将在 `DateField` 上注册一个 `YearExact` 查找。它会覆盖已存在的同名查找。

`get_lookup(lookup_name)`

返回类中注册的名为 `lookup_name` 的 `Lookup` 。默认的实现会递归查询所有的父类，并检查它们中的任何一个是否具有名称为 `lookup_name` 的查找，并返回第一个匹配。

`get_transform(transform_name)`

返回一个名为 `transform_name` 的 `Transform` 。默认的实现会递归查找所有的父类，并检查它们中的任何一个是否具有名称为 `transform_name` 的查找，并返回第一个匹配。

一个类如果想要成为查找，它必须实现查询表达式API。 `Lookup` 和 `Transform` 一开始就遵循这个API。

查询表达式API

查询表达式API是一个通用的方法集，在查询表达式中可以使用定义了这些方法的类，来将它们自身转换为SQL表达式。直接的字段引用，聚合，以及 `Transform` 类都是遵循这个API的示例。当一个对象实现以下方法时，就被称为遵循查询表达式API：

`as_sql(self, compiler, connection)`

负责从表达式中产生查询字符串和参数。 `compiler` 是一个 `SQLCompiler` 对象，它拥有可以编译其它表达式的 `compile()` 方法。 `connection` 是用于执行查询的连接。

调用 `expression.as_sql()` 一般是不对的 -- 而是应该调用 `compiler.compile(expression)` 。 `compiler.compile()` 方法应该在调用表达式的供应商特定方法时格外小心。

`as_vendorname(self, compiler, connection)`

和 `as_sql()` 的工作方式类似。当一个表达式经过 `compiler.compile()` 编译之后， Django会首先尝试调用 `as_vendorname()` ，其中 `vendorname` 是用于执行查询的后端供应商。对于Django内建的后端， `vendorname` 是 `postgresql` ， `oracle` ， `sqlite` ，或者 `mysql` 之一。

`get_lookup(lookup_name)`

必须返回名称为 `lookup_name` 的查找。例如，通过返回 `self.output_field.get_lookup(lookup_name)` 来实现。

`get_transform(transform_name)`

必须返回名称为 `transform_name的` 查找。例如，通过返回 `self.output_field.get_transform(transform_name)` 来实现。

`output_field`

定义 `get_lookup()` 方法所返回的类的类型。必须为 `Field` 的实例。

Transform 类参考

`class Transform`

`Transform` 是用于实现字段转换的通用类。一个显然的例子是 `__year` 会把 `DateField` 转换为 `IntegerField` 。

在表达式中执行查找的标记是 `Transform<expression>__<transformation>` (例如 `date__year`)。

这个类遵循查询表达式API，也就是说你可以使用 `<expression>__<transform1>__<transform2>` 。

`bilateral`

> New in Django 1.8.

一个布尔值，表明是否对 `lhs` 和 `rhs` 都应用这个转换。如果对两侧都应用转换，应用在 `rhs` 的顺序和在查找表达式中的出现顺序相同。默认这个属性为 `False` 。使用方法的实例请见自定义查找。

`lhs`

在左边，也就是被转换的东西。必须遵循查询表达式API。

`lookup_name`

查找的名称，用于在解析查询表达式的时候识别它。

`output_field`

为这个类定义转换后的输出。必须为 `Field` 的实例。默认情况下和 `lhs.output_field` 相同。

`as_sql()`

需要被覆写；否则抛出 `NotImplementedError` 异常。

`get_lookup(lookup_name)`

和 `get_lookup()` 相同。

`get_transform(transform_name)`

和 `get_transform()` 相同。

Lookup 类参考

`class Lookup`

`Lookup` 是实现查找的通用的类。查找是一个查询表达式，它的左边是 `lhs` ，右边是 `rhs` ； `lookup_name` 用于构造 `lhs` 和 `rhs` 之间的比较，来产生布尔值，例如 `lhs in rhs` 或者 `lhs > rhs` 。

在表达式中执行查找的标记是 `<lhs>__<lookup_name>=<rhs>` 。

这个类并不遵循查询表达式API，因为在它构造的时候出现了 `=<rhs>` ：查找总是在查找表达式的最后。

`lhs`

在左边，也就是被查找的东西。这个对象必须遵循查询表达式API。

`rhs`

在右边，也就是用来和 `lhs` 比较的东西。它可以是个简单的值，也可以是在SQL中编译的一些东西，比如 `F()` 对象或者 `QuerySet` 。

`lookup_name`

查找的名称，用于在解析查询表达式的时候识别它。

`process_lhs(compiler, connection[, lhs=None])`

返回元组 `(lhs_string, lhs_params)` ，和 `compiler.compile(lhs)` 所返回的一样。这个方法可以被覆写，来调整 `lhs` 的处理方式。

`compiler` 是一个 `SQLCompiler` 对象，可以像 `compiler.compile(lhs)` 这样使用来编译 `lhs` 。 `connection` 可以用于编译供应商特定的SQL语句。 `lhs` 如果不为 `None` ，会代替 `self.lhs` 作为处理后的 `lhs` 使用。

`process_rhs(compiler, connection)`

对于右边的东西，和 `process_lhs()` 的行为相同。

> 译者：Django 文档协作翻译小组，原文：Lookup expressions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

模型的实例

模型实例参考

该文档详细描述模型 的API。它建立在模型 和执行查询 的资料之上， 所以在阅读这篇文档之前，你可能会想要先阅读并理解那两篇文档。

我们将用执行查询中所展现的 博客应用模型 来贯穿这篇参考文献。

创建对象

要创建模型的一个新实例，只需要像其它Python 类一样实例化它：

```
class Model(**kwargs)
```

关键字参数就是在你的模型中定义的字段的名字。注意，实例化一个模型不会访问数据库；若要保存，你需要save() 一下。

> 注
>
> 也许你会想通过重写 `__init__` 方法来自定义模型。无论如何，如果你这么做了，小心不要改变了调用签名——任何改变都可能阻碍模型实例被保存。尝试使用下面这些方法之一，而不是重写**init**：
>
> 1. 在模型类中增加一个类方法：

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)

    @classmethod
    def create(cls, title):
        book = cls(title=title)
        # do something with the book
        return book

book = Book.create("Pride and Prejudice")
```

> 2. 在自定义管理器中添加一个方法（推荐）：

```
class BookManager(models.Manager):
    def create_book(self, title):
        book = self.create(title=title)
        # do something with the book
        return book

class Book(models.Model):
    title = models.CharField(max_length=100)

    objects = BookManager()

book = Book.objects.create_book("Pride and Prejudice")
```

自定义模型加载

`classmethod Model.from_db(db, field_names, values)`

> New in Django 1.8.

`from_db()` 方法用于自定义从数据库加载时模型实例的创建。

`db` 参数包含数据库的别名，`field_names` 包含所有加载的字段的名称，`values` 包含 `field_names` 中每个字段加载的值。`field_names` 与 `values` 的顺序相同，所以可以使用 `cls(**(zip(field_names, values)))` 来实例化对象。如果模型的所有字段都提供，会保证 `values` 的顺序与 `__init__()` 所期望的一致。这表示此时实例可以通过 `cls(*values)` 创建。可以通过 `cls._deferred` 来检查是否提供所有的字段 —— 如果为 `False`，那么所有的字段都已经从数据库中加载。

除了创建新模型之前，`from_db()` 必须设置新实例 `_state` 属性中的 `adding` 和 `db` 标志位。

下面的示例演示如何保存从数据库中加载进来的字段原始值：

```
@classmethod
def from_db(cls, db, field_names, values):
    # default implementation of from_db() (could be replaced
    # with super())
    if cls._deferred:
        instance = cls(**zip(field_names, values))
    else:
        instance = cls(*values)
    instance._state.adding = False
    instance._state.db = db
    # customization to store the original field values on the instance
    instance._loaded_values = zip(field_names, values)
    return instance

def save(self, *args, **kwargs):
    # Check how the current values differ from ._loaded_values. For example,
    # prevent changing the creator_id of the model. (This example doesn't
    # support cases where 'creator_id' is deferred).
    if not self._state.adding and (
            self.creator_id != self._loaded_values['creator_id']):
        raise ValueError("Updating the value of creator isn't allowed")
    super(...).save(*args, **kwargs)
```

上面的示例演示 `from_db()` 的完整实现。当然在这里的 `from_db()` 中完全可以只用 `super()` 调用。

从数据库更新对象

`Model.refresh_from_db(using=None, fields=None, **kwargs)`

> New in Django 1.8.

如果你需要从数据库重新加载模型的一个值，你可以使用 `refresh_from_db()` 方法。当不带参数调用这个方法时，将完成以下的动作：

模型的所有非延迟字段都更新成数据库中的当前值。 之前加载的关联实例，如果关联的值不再合法，将从重新加载的实例中删除。例如，如果重新加载的实例有一个外键到另外一个模型 `Author`，那么如果 `obj.author_id != obj.author.id`， `obj.author` 将被扔掉并在下次访问它时根据 `obj.author_id` 的值重新加载。 注意，只有本模型的字段会从数据库重新加载。其它依赖数据库的值不会重新加载，例如聚合的结果。

重新加载使用的数据库与实例加载时使用的数据库相同，如果实例不是从数据库加载的则使用默认的数据库。可以使用 `using` 参数来强制指定重新加载的数据库。

可以回使用 `fields` 参数强制设置加载的字段。

例如，要测试 `update()` 调用是否得到预期的更新，可以编写类似下面的测试：

```
def test_update_result(self):
    obj = MyModel.objects.create(val=1)
    MyModel.objects.filter(pk=obj.pk).update(val=F('val') + 1)
    # At this point obj.val is still 1, but the value in the database
    # was updated to 2. The object's updated value needs to be reloaded
    # from the database.
    obj.refresh_from_db()
    self.assertEqual(obj.val, 2)
```

注意，当访问延迟的字段时，延迟字段的加载会通过这个方法加载。所以可以自定义延迟加载的行为。下面的实例演示如何在重新加载一个延迟字段时重新加载所有的实例字段：

```
class ExampleModel(models.Model):
    def refresh_from_db(self, using=None, fields=None, **kwargs):
        # fields contains the name of the deferred field to be
        # loaded.
        if fields is not None:
            fields = set(fields)
            deferred_fields = self.get_deferred_fields()
            # If any deferred field is going to be loaded
            if fields.intersection(deferred_fields):
                # then load all of them
                fields = fields.union(deferred_fields)
        super(ExampleModel, self).refresh_from_db(using, fields, **kwargs)
```

`Model.get_deferred_fields()`

```
New in Django 1.8.
```

一个辅助方法，它返回一个集合，包含模型当前所有延迟字段的属性名称。

验证对象

验证一个模型涉及三个步骤：

1. 验证模型的字段 —— `Model.clean_fields()`
2. 验证模型的完整性 —— `Model.clean()`
3. 验证模型的唯一性 —— `Model.validate_unique()`

当你调用模型的 `full_clean()` 方法时，这三个方法都将执行。

当你使用 `ModelForm` 时， `is_valid()` 将为表单中的所有字段执行这些验证。更多信息参见 `ModelForm` 文档。 如果你计划自己处理验证出现的错误，或者你已经将需要验证的字段从 `ModelForm` 中去除掉，你只需调用模型的 `full_clean()` 方法。

`Model.full_clean(exclude=None, validate_unique=True)`

该方法按顺序调用 `Model.clean_fields()`、 `Model.clean()` 和 `Model.validate_unique()` （如果 `validate_unique` 为 `True` ），并引发一个 `ValidationError` ，该异常的 `message_dict` 属性包含三个步骤的所有错误。

可选的 `exclude` 参数用来提供一个可以从验证和清除中排除的字段名称的列表。 `ModelForm` 使用这个参数来排除表单中没有出现的字段，使它们不需要验证，因为用户无法修正这些字段的错误。

注意，当你调用模型的 `save()` 方法时， `full_clean()` 不会 自动调用。如果你想一步就可以为你手工创建的模型运行验证，你需要手工调用它。例如：

```
from django.core.exceptions import ValidationError
try:
    article.full_clean()
except ValidationError as e:
    # Do something based on the errors contained in e.message_dict.
    # Display them to a user, or handle them programmatically.
    pass
```

`full_clean()` 第一步执行的是验证每个字段。

`Model.clean_fields(exclude=None)`

这个方法将验证模型的所有字段。可选的 `exclude` 参数让你提供一个字段名称列表来从验证中排除。如果有字段验证失败，它将引发一个 `ValidationError` 。

`full_clean()` 第二步执行的是调用 `Model.clean()` 。如要实现模型自定义的验证，应该覆盖这个方法。

`Model.clean()`

应该用这个方法来提供自定义的模型验证，以及修改模型的属性。例如，你可以使用它来给一个字段自动提供值，或者用于多个字段需要一起验证的情形：

```
import datetime
from django.core.exceptions import ValidationError
from django.db import models

class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError('Draft entries may not have a publication date.')
        # Set the pub_date for published items if it hasn't been set already.
        if self.status == 'published' and self.pub_date is None:
            self.pub_date = datetime.date.today()
```

然而请注意，和 `Model.full_clean()` 类似，调用模型的 `save()` 方法时不会引起 `clean()` 方法的调用。

在上面的示例中， `Model.clean()` 引发的 `ValidationError` 异常通过一个字符串实例化，所以它将被保存在一个特殊的错误字典键 `NON_FIELD_ERRORS` 中。这个键用于整个模型出现的错误而不是一个特定字段出现的错误：

```
from django.core.exceptions import ValidationError, NON_FIELD_ERRORS
try:
    article.full_clean()
except ValidationError as e:
    non_field_errors = e.message_dict[NON_FIELD_ERRORS]
```

若要引发一个特定字段的异常，可以使用一个字典实例化 `ValidationError` ，其中字典的键为字段的名称。我们可以更新前面的例子，只引发 `pub_date` 字段上的异常：

```
class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError({'pub_date': 'Draft entries may not have a publication
        ...
```

最后， `full_clean()` 将检查模型的唯一性约束。

`Model.validate_unique(exclude=None)`

该方法与 `clean_fields()` 类似，只是验证的是模型的所有唯一性约束而不是单个字段的值。可选的 `exclude` 参数允许你提供一个字段名称的列表来从验证中排除。如果有字段验证失败，将引发一个 `ValidationError` 。

注意，如果你提供一个 `exclude` 参数给 `validate_unique()` ，任何涉及到其中一个字段的 `unique_together` 约束将不检查。

对象保存

将一个对象保存到数据库，需要调用 `save()` 方法：

```
Model.save([force_insert=False, force_update=False, using=DEFAULT_DB_ALIAS, update_fields=
```

如果你想要自定义保存的动作，你可以重写 `save()` 方法。请看 重写预定义的模型方法 了解更多细节。

模型保存过程还有一些细节的地方要注意；请看下面的章节。

自增的主键

如果模型具有一个 `AutoField` —— 一个自增的主键 —— 那么该自增的值将在第一次调用对象的 `save()` 时计算并保存：

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b doesn't have an ID yet.
>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
```

在调用 `save()` 之前无法知道ID 的值，因为这个值是通过数据库而不是Django 计算。

为了方便，默认情况下每个模型都有一个 `AutoField` 叫做 `id`，除非你显式指定模型某个字段的 `primary_key=True` 。更多细节参见 `AutoField` 的文档。

pk 属性

```
Model.pk
```

无论你是自己定义还是让Django 为你提供一个主键字段， 每个模型都将具有一个属性叫做 `pk` 。它的行为类似模型的一个普通属性，但实际上是模型主键字段属性的别名。你可以读取并设置它的值，就和其它属性一样，它会更新模型中正确的值。

显式指定自增主键的值

如果模型具有一个 `AutoField`，但是你想在保存时显式定义一个新的对象 `ID`，你只需要在保存之前显式指定它而不用依赖 `ID` 自动分配的值：

```
>>> b3 = Blog(id=3, name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b3.id     # Returns 3.
>>> b3.save()
>>> b3.id     # Returns 3.
```

如果你手工赋值一个自增主键的值，请确保不要使用一个已经存在的主键值！如果你使用数据库中已经存在的主键值创建一个新的对象，Django 将假设你正在修改这个已存在的记录而不是创建一个新的记录。

接着上面的' `Cheddar Talk` ' 博客示例，下面这个例子将覆盖数据库中之前的记录：

```
b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese.')
b4.save()  # Overrides the previous blog with ID=3!
```

出现这种情况的原因，请参见下面的Django 如何知道是UPDATE 还是INSERT。

显式指定自增主键的值对于批量保存对象最有用，但你必须有信心不会有主键冲突。

当你保存时，发生了什么？

当你保存一个对象时，Django 执行以下步骤：

1. 发出一个 `pre-save` 信号。 发送一个 `django.db.models.signals.pre_save` 信号，以允许监听该信号的函数完成一些自定义的动作。

2. 预处理数据。 如果需要，对对象的每个字段进行自动转换。

大部分字段不需要预处理 —— 字段的数据将保持原样。预处理只用于具有特殊行为的字段。例如，如果你的模型具有一个 `auto_now=True` 的 `DateField` ，那么预处理阶段将修改对象中的数据以确保该日期字段包含当前的时间戳。（我们的文档还没有所有具有这种“特殊行为”字段的一个列表。）

3. 准备数据库数据。 要求每个字段提供的当前值是能够写入到数据库中的类型。

大部分字段不需要数据准备。简单的数据类型，例如整数和字符串，是可以直接写入的 Python 对象。但是，复杂的数据类型通常需要一些改动。

例如， `DateField` 字段使用Python 的 `datetime` 对象来保存数据。数据库保存的不是 `datetime` 对象，所以该字段的值必须转换成ISO兼容的日期字符串才能插入到数据库中。

4. 插入数据到数据库中。 将预处理过、准备好的数据组织成一个SQL 语句用于插入数据库。

5. 发出一个post-save 信号。 发送一个 `django.db.models.signals.post_save` 信号，以允许监听听信号的函数完成一些自定义的动作。

Django 如何知道是UPDATE 还是INSERT

你可能已经注意到Django 数据库对象使用同一个 `save()` 方法来创建和改变对象。Django 对 `INSERT` 和 `UPDATE` SQL 语句的使用进行抽象。当你调用 `save()` 时，Django 使用下面的算法：

- 如果对象的主键属性为一个求值为 `True` 的值（例如，非 `None` 值或非空字符串），Django 将执行 `UPDATE` 。
- 如果对象的主键属性没有设置或者 `UPDATE` 没有更新任何记录，Django 将执行 `INSERT` 。

现在应该明白了，当保存一个新的对象时，如果不能保证主键的值没有使用，你应该注意不要显式指定主键值。关于这个细微差别的更多信息，参见上文的显示指定主键的值 和下文的强制使用 `INSERT` 或 `UPDATE` 。

在Django 1.5 和更早的版本中，在设置主键的值时，Django 会作一个 `SELECT` 。如果 `SELECT` 找到一行，那么Django 执行 `UPDATE` ，否则执行 `INSERT` 。旧的算法导致 `UPDATE` 情况下多一次查询。有极少数的情况，数据库不会报告有一行被更新，即使数据库包含该对象的主键值。有个例子是PostgreSQL 的 `ON UPDATE` 触发器，它返回NULL。在这些情况下，可能要通过将 `select_on_save` 选项设置为 `True` 以启用旧的算法。

强制使用INSERT 或UPDATE

在一些很少见的场景中，需要强制 `save()` 方法执行SQL 的 `INSERT` 而不能执行 `UPDATE` 。或者相反：更新一行而不是插入一个新行。在这些情况下，你可以传递 `force_insert=True` 或 `force_update=True` 参数给 `save()` 方法。显然，两个参数都传递是错误的：你不可能同时插入和更新！

你应该极少需要使用这些参数。Django 几乎始终会完成正确的事情，覆盖它将导致错误难以跟踪。这个功能只用于高级用法。

使用 `update_fields` 将强制使用类似 `force_update` 的更新操作。

基于已存在字段值的属性更新

有时候你需要在一个字段上执行简单的算法操作，例如增加或者减少当前值。实现这点的简单方法是像下面这样：

```
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold += 1
>>> product.save()
```

如果从数据库中读取的旧的 `number_sold` 值为10，那么写回到数据库中的值将为11。

通过将更新基于原始字段的值而不是显式赋予一个新值，这个过程可以避免竞态条件而且更快。Django 提供 `F 表达式` 用于这种类型的相对更新。利用 `F 表达式` ，前面的示例可以表示成：

```
>>> from django.db.models import F
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold = F('number_sold') + 1
>>> product.save()
```

更多细节，请参见 `F 表达式` 和它们在更新查询中的用法。

指定要保存的字段

如果传递给 `save()` 的 `update_fields` 关键字参数一个字段名称列表，那么将只有该列表中的字段会被更新。如果你想更新对象的一个或几个字段，这可能是你想要的。不让模型的所有字段都更新将会带来一些轻微的性能提升。例如：

```
product.name = 'Name changed again'
product.save(update_fields=['name'])
```

`update_fields` 参数可以是任何包含字符串的可迭代对象。空的 `update_fields` 可迭代对象将会忽略保存。如果为 `None` 值，将执行所有字段上的更新。

指定 `update_fields` 将强制使用更新操作。

当保存通过延迟模型加载（ `only()` 或 `defer()` ）进行访问的模型时，只有从数据库中加载的字段才会得到更新。这种情况下，有个自动的 `update_fields` 。如果你赋值或者改变延迟字段的值，该字段将会添加到更新的字段中。

删除对象

`Model.delete([using=DEFAULT_DB_ALIAS])`

发出一个SQL `DELETE` 操作。它只在数据库中删除这个对象；其Python 实例仍将存在并持有各个字段的数据。

更多细节，包括如何批量删除对象，请参见删除对象。

如果你想自定义删除的行为，你可以覆盖 `delete()` 方法。详见覆盖预定义的模型方法。

Pickling 对象

当你 `pickle` 一个模型时，它的当前状态是pickled。当你unpickle 它时，它将包含pickle 时模型的实例，而不是数据库中的当前数据。

> 你不可以在不同版本之间共享pickles
>
> 模型的Pickles 只对于产生它们的Django 版本有效。如果你使用Django 版本N pickle，不能保证Django 版本N+1 可以读取这个pickle。Pickles 不应该作为长期的归档策略。
>
> `New in Django 1.8.`
>
> 因为pickle 兼容性的错误很难诊断例如一个悄无声息损坏的对象，当你unpickle 模型使用的Django 版本与pickle 时的不同将引发一个 `RuntimeWarning` 。

其它的模型实例方法

有几个实例方法具有特殊的目的。

> 注
>
> 在Python 3 上，因为所有的字段都原生被认为是Unicode，只需使用 `__str__()` 方法（ `__unicode__()` 方法被废弃）。如果你想与Python 2 兼容，你可以使用 `python_2_unicode_compatible()` 装饰你的模型类。

__unicode__

`Model.__unicode__()`

`__unicode__()` 方法在每当你对一个对象调用unicode() 时调用。Django 在许多地方都使用 `unicode(obj)` （或者相关的函数 `str(obj)` ）。最明显的是在Django 的Admin 站点显示一个对象和在模板中插入对象的值的时候。所以，你应该始终让 `__unicode__()` 方法返回模型的一个友好的、人类可读的形式。

例如：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

如果你定义了模型的 `__unicode__()` 方法且没有定义 `__str__()` 方法，Django 将自动提供一个 `__str__()` ，它调用 `__unicode__()` 并转换结果为一个UTF-8 编码的字符串。下面是一个建议的开发实践：只定义 `__unicode__()` 并让Django 在需要时负责字符串的转换。

__str__

`Model.__str__()`

`__str__()` 方法在每当你对一个对象调用 `str()` 时调用。在Python 3 中，Django 在许多地方使用 `str(obj)` 。 最明显的是在Django 的Admin 站点显示一个对象和在模板中插入对象的值的时候。 所以，你应该始终让 `__str__()` 方法返回模型的一个友好的、人类可读的形式。

例如：

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

在Python 2 中，Django 内部对 `__str__` 的直接使用主要在随处可见的模型的 `repr()` 输出中（例如，调试时的输出）。如果已经有合适的 `__unicode__()` 方法就不需要 `__str__()` 了。

前面 `__unicode__()` 的示例可以使用 `__str__()` 这样类似地编写：

```
from django.db import models
from django.utils.encoding import force_bytes

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        # Note use of django.utils.encoding.force_bytes() here because
        # first_name and last_name will be unicode strings.
        return force_bytes('%s %s' % (self.first_name, self.last_name))
```

__eq__

`Model.__eq__()`

定义这个方法是为了让具有相同主键的相同实类的实例是相等的。对于代理模型，实类是模型第一个非代理父类；对于其它模型，它的实类就是模型类自己。

例如：

```
from django.db import models

class MyModel(models.Model):
    id = models.AutoField(primary_key=True)

class MyProxyModel(MyModel):
    class Meta:
        proxy = True

class MultitableInherited(MyModel):
    pass

MyModel(id=1) == MyModel(id=1)
MyModel(id=1) == MyProxyModel(id=1)
MyModel(id=1) != MultitableInherited(id=1)
MyModel(id=1) != MyModel(id=2)
```

```
Changed in Django 1.7:

在之前的版本中，只有类和主键都完全相同的实例才是相等的。
```

__hash__

`Model.__hash__()`

`__hash__` 方法基于实例主键的值。它等同于hash(obj.pk)。如果实例的主键还没有值，将引发一个 `TypeError`（否则，`__hash__` 方法在实例保存的前后将返回不同的值，而改变一个实例的 `__hash__` 值在Python 中是禁止的）。

```
Changed in Django 1.7:

在之前的版本中，主键没有值的实例是可以哈希的。
```

get_absolute_url

`Model.get_absolute_url()`

`get_absolute_url()` 方法告诉Django 如何计算对象的标准URL。对于调用者，该方法返回的字符串应该可以通过HTTP 引用到这个对象。

例如：

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

（虽然这段代码正确又简单，这并不是编写这个方法可移植性最好的方式。通常使用 `reverse()` 函数是最好的方式。）

例如：

```
def get_absolute_url(self):
    from django.core.urlresolvers import reverse
    return reverse('people.views.details', args=[str(self.id)])
```

Django 使用 `get_absolute_url()` 的一个地方是在Admin 应用中。如果对象定义该方法，对象编辑页面将具有一个“View on site”链接，可以将你直接导入由 `get_absolute_url()` 提供的对象公开视图。

类似地，Django 的另外一些小功能，例如syndication feed 框架 也使用 `get_absolute_url()` 。 如果模型的每个实例都具有一个唯一的URL 是合理的，你应该定义 `get_absolute_url()` 。

> 警告
>
> 你应该避免从没有验证过的用户输入构建URL，以减少有害的链接和重定向：

```
def get_absolute_url(self):
    return '/%s/' % self.name
```

> 如果 `self.name` 为' `/example.com` '，将返回 ' `//example.com/` '， 而它是一个合法的相对URL而不是期望的' `/%2Fexample.com/` '。

在模板中使用 `get_absolute_url()` 而不是硬编码对象的URL 是很好的实践。例如，下面的模板代码很糟糕：

```
<!-- BAD template code. Avoid! -->
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

下面的模板代码要好多了：

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

如果你改变了对象的URL 结构，即使是一些简单的拼写错误，你不需要检查每个可能创建该 URL 的地方。在 `get_absolute_url()` 中定义一次，然后在其它代码调用它。

> 注
>
> `get_absolute_url()` 返回的字符串必须只包含ASCII 字符（URI 规范RFC 2396 的要求），并且如需要必须要URL-encoded。
>
> 代码和模板中对 `get_absolute_url()` 的调用应该可以直接使用而不用做进一步处理。你可能想使用 `django.utils.encoding.iri_to_uri()` 函数来帮助你解决这个问题，如果你正在使用ASCII 范围之外的Unicode 字符串。

额外的实例方法

除了 `save()` 、 `delete()` 之外，模型的对象还可能具有以下一些方法：

`Model.get_FOO_display()`

对于每个具有 `choices` 的字段，每个对象将具有一个 `get_FOO_display()` 方法，其中 `FOO` 为该字段的名称。这个方法返回该字段对“人类可读”的值。

例如：

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=2, choices=SHIRT_SIZES)
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`Model.get_next_by_FOO(**kwargs)`

`Model.get_previous_by_FOO(**kwargs)`

如果 `DateField` 和 `DateTimeField` 没有设置 `null=True`，那么该对象将具有 `get_next_by_FOO()` 和 `get_previous_by_FOO()` 方法，其中 `FOO` 为字段的名称。它根据日期字段返回下一个和上一个对象，并适时引发一个 `DoesNotExist` 。

这两个方法都将使用模型默认的管理器来执行查询。如果你需要使用自定义的管理器或者你需要自定义的筛选，这个两个方法还接受可选的参数，它们应该用字段查询 中提到的格式。

注意，对于完全相同的日期，这些方法还将利用主键来进行查找。这保证不会有记录遗漏或重复。这还意味着你不可以在未保存的对象上使用这些方法。

其它属性

DoesNotExist

exception `Model.DoesNotExist`

ORM 在好几个地方会引发这个异常，例如 `QuerySet.get()` 根据给定的查询参数找不到对象时。

Django 为每个类提供一个 `DoesNotExist` 异常属性是为了区别找不到的对象所属的类，并让你可以利用 `try/except` 捕获一个特定模型的类。这个异常是 `django.core.exceptions.ObjectDoesNotExist` 的子类。

> 译者：Django 文档协作翻译小组，原文：Instance methods。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

关联对象参考

class RelatedManager

"关联管理器"是在一对多或者多对多的关联上下文中使用的管理器。它存在于下面两种情况：

ForeignKey关系的“另一边”。像这样：

```
from django.db import models

class Reporter(models.Model):
    # ...
    pass

class Article(models.Model):
    reporter = models.ForeignKey(Reporter)
```

在上面的例子中，管理器reporter.article_set拥有下面的方法。

ManyToManyField关系的两边：

```
class Topping(models.Model):
    # ...
    pass

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

这个例子中，topping.pizza_set 和pizza.toppings都拥有下面的方法。

add(obj1[, obj2, ...])

把指定的模型对象添加到关联对象集中。

例如：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

在上面的例子中，对于ForeignKey关系，e.save()由关联管理器调用，执行更新操作。然而，在多对多关系中使用add()并不会调用任何 save()方法，而是由QuerySet.bulk_create()创建关系。如果你需要在关系被创建时执行一些自定义的逻辑，请监听m2m_changed信号。

create(**kwargs)

创建一个新的对象，保存对象，并将它添加到关联对象集之中。返回新创建的对象：

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )

# No need to call e.save() at this point -- it's already been saved.
```

这完全等价于（不过更加简洁于）：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
...     blog=b,
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )
>>> e.save(force_insert=True)
```

要注意我们并不需要指定模型中用于定义关系的关键词参数。在上面的例子中，我们并没有传入blog参数给create()。Django会明白新的 Entry对象blog 应该添加到b中。

remove(obj1[, obj2, ...])

从关联对象集中移除执行的模型对象：

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassociates Entry e from Blog b.
```

和add()相似，上面的例子中，e.save()可会执行更新操作。但是，多对多关系上的remove()，会使用QuerySet.delete()删除关系，意思是并不会有任何模型调用save()方法：如果你想在一个关系被删除时执行自定义的代码，请监听m2m_changed信号。

对于ForeignKey对象，这个方法仅在null=True时存在。如果关联的字段不能设置为None (NULL)，则这个对象在添加到另一个关联之前不能移除关联。在上面的例子中，从b.entry_set()移除e等价于让e.blog = None，由于blog的ForeignKey没有设置null=True，这个操作是无效的。

对于ForeignKey对象，该方法接受一个bulk参数来控制它如果执行操作。如果为True（默认值），QuerySet.update()会被使用。而如果bulk=False，会在每个单独的模型实例上调用save()方法。这会触发pre_save和post_save，它们会消耗一定的性能。

clear()

从关联对象集中移除一切对象。

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

注意这样不会删除对象 —— 只会删除他们之间的关联。

就像 remove() 方法一样，clear()只能在 null=True的ForeignKey上被调用，也可以接受bulk关键词参数。

> 注意
>
> 注意对于所有类型的关联字段，add()、create()、remove()和clear()都会马上更新数据库。换句话说，在关联的任何一端，都不需要再调用save()方法。
>
> 同样，如果你再多对多关系中使用了中间模型，一些关联管理的方法会被禁用。

直接赋值

通过赋值一个新的可迭代的对象，关联对象集可以被整体替换掉。

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set = new_list
```

如果外键关系满足null=True，关联管理器会在添加new_list中的内容之前，首先调用clear()方法来解除关联集中一切已存在对象的关联。否则， new_list中的对象会在已存在的关联的基础上被添加。

迁移

模式编辑器

`class BaseDatabaseSchemaEditor[source]`

Django的迁移系统分为两个部分；计算和储存应该执行什么操作的逻辑（`django.db.migrations`），以及用于把“创建模型”或者“删除字段”变成SQL语句的数据库抽象层 -- 后者是模式编辑器的功能。

你可能并不想像一个普通的开发者使用Django那样，直接和模型编辑器进行交互，但是如果你编写自己的迁移系统，或者有更进一步的需求，这样会比编写SQL语句更方便。

每个Django的数据库后端都提供了它们自己的模式编辑器，并且总是可以通过 `connection.schema_editor()` 上下文管理器来访问。

```
with connection.schema_editor() as schema_editor:
    schema_editor.delete_model(MyModel)
```

它必须通过上下文管理器来使用，因为这样可以管理一些类似于事务和延迟SQL（比如创建 `ForeignKey` 约束）的东西。

它会暴露所有可能的操作作为方法，这些方法应该按照执行修改的顺序调用。可能一些操作或者类型并不可用于所有数据库 -- 例如，MyISAM引擎不支持外键约束。

如果你在为Django编写一个三方的数据库后端，你需要提供 `SchemaEditor` 实现来使用1.7的迁移功能 -- 然而，只要你的数据库在SQL的使用和关系设计上遵循标准，你就应该能够派生Django内建的 `SchemaEditor` 之一，然后简单调整一下语法。同时也要注意，有一些新的数据库特性是迁移所需要的： `can_rollback_ddl` 和 `supports_combined_alters` 都很重要。

方法

execute

`BaseDatabaseSchemaEditor.execute(sql, params=[])[source]`

执行传入的 SQL语句，如果提供了参数则会带上它们。这是对普通数据库游标的一个简单封装，如果用户希望的话，它可以从 `.sql` 文件中获取SQL。

create_model

`BaseDatabaseSchemaEditor.create_model(model)[source]`

为提供的模型在数据库中创建新的表，带有所需的任何唯一性约束或者索引。

delete_model

`BaseDatabaseSchemaEditor.delete_model(model)[source]`

删除数据库中的模型的表，以及它带有的任何唯一性约束或者索引。

alter_unique_together

`BaseDatabaseSchemaEditor.alter_unique_together(model, old_unique_together, new_unique_toge`

修改模型的 `unique_together` 值；这会向模型表中添加或者删除唯一性约束，使它们匹配新的值。

alter_index_together

`BaseDatabaseSchemaEditor.alter_index_together(model, old_index_together, new_index_togethe`

修改模型的 `index_together` 值；这会向模型表中添加或者删除索引，使它们匹配新的值。

alter_db_table

`BaseDatabaseSchemaEditor.alter_db_table(model, old_db_table, new_db_table)[source]`

重命名模型的表，从 `old_db_table` 变成 `new_db_table`。

alter_db_tablespace

`BaseDatabaseSchemaEditor.alter_db_tablespace(model, old_db_tablespace, new_db_tablespace)[`

把模型的表从一个表空间移动到另一个中。

add_field

`BaseDatabaseSchemaEditor.add_field(model, field)[source]`

向模型的表中添加一列（或者有时几列），表示新增的字段。如果该字段带有 `db_index=True` 或者 `unique=True`，同时会添加索引或者唯一性约束。

如果字段为 `ManyToManyField` 并且缺少 `through` 值，会创建一个表来表示关系，而不是创建一列。如果提供了 `through` 值，就什么也不做。

如果字段为 `ForeignKey`，同时会向列上添加一个外键约束。

remove_field

`BaseDatabaseSchemaEditor.remove_field(model, field)[source]`

从模型的表中移除代表字段的列，以及列上的任何唯一性约束，外键约束，或者索引。

如果字段是 `ManyToManyField` 并且缺少 `through` 值，会移除创建用来跟踪关系的表。如果提供了 `through` 值，就什么也不做。

alter_field

`BaseDatabaseSchemaEditor.alter_field(model, old_field, new_field, strict=False)[source]`

这会将模型的字段从旧的字段转换为新的。这包括列名称的修改（`db_column` 属性）、字段类型的修改（如果修改了字段类）、字段 `NULL` 状态的修改、添加或者删除字段层面的唯一性约束和索引、修改主键、以及修改 `ForeignKey` 约束的目标。

最普遍的一个不能实现的转换，是把 `ManyToManyField` 变成一个普通的字段，反之亦然；Django不能在不丢失数据的情况下执行这个转换，所以会拒绝这样做。作为替代，应该单独调用 `remove_field()` 和 `add_field()`。

如果数据库满足 `supports_combined_alters`，Django会尽可能在单次数据库调用中执行所有这些操作。否则对于每个变更，都会执行一个单独的 `ALTER` 语句，但是如果不需要做任何改变，则不执行 `ALTER`（就像South经常做的那样）。

属性

除非另有规定，所有属性都应该是只读的。

connection

`SchemaEditor.connection`

一个到数据库的连接对象。`alias` 是 `connection` 的一个实用的属性，它用于决定要访问的数据库的名字。

当你在多种数据库之间执行迁移的时候，这是非常有用的。

> 译者：Django 文档协作翻译小组，原文：SchemaEditor。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写数据库迁移

这一节介绍你可能遇到的在不同情况下如何分析和编写数据库迁移. 有关迁移的入门资料，请查看 the topic guide.

数据迁移和多数据库

在使用多个数据库时，需要解决是否针对某个特定数据库运行迁移。例如，你可能 只 想在某个特定数据库上运行迁移。

为此你可以在RunPython中通过查看schema_editor.connection.alias 属性来检查数据库连接别名：

```
from django.db import migrations

def forwards(apps, schema_editor):
    if not schema_editor.connection.alias == 'default':
        return
    # Your migration code goes here

class Migration(migrations.Migration):

    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards),
    ]
```

```
Django 1.8 中新增。
```

你也可以提供一个提示作为 **hints参数传递到数据库路由的allow_migrate() 方法：

```
myapp/dbrouters.py
class MyRouter(object):

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        if 'target_db' in hints:
            return db == hints['target_db']
        return True
```

然后，要在你的迁移中利用，执行以下操作：

```
from django.db import migrations

def forwards(apps, schema_editor):
    # Your migration code goes here

class Migration(migrations.Migration):

    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards, hints={'target_db': 'default'}),
    ]
```

如果你的RunPython或者RunSQL操作只对一个模型有影响，最佳实践是将model_name作为提示传递，使其尽可能对路由可见。这对可复用的和第三方应用极其重要。

添加唯一字段的迁移

如果你应用了一个“朴素”的迁移，向表中一个已存在的行中添加了一个唯一的非空字段，会产生错误，因为位于已存在行中的值只会生成一次。所以需要移除唯一性的约束。

所以，应该执行下面的步骤。在这个例子中，我们会以默认值添加一个非空的UUIDField字段。你可以根据你的需要修改各个字段。

- 把default=...和unique=True参数添加到你模型的字段中。在这个例子中，我们默认使用uuid.uuid4。
- 运行 makemigrations 命令。
- 编辑创建的迁移文件。

生成的迁移类看上去像这样：

```
class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0003_auto_20150129_1705'),
    ]

    operations = [
        migrations.AddField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(max_length=32, unique=True, default=uuid.uuid4),
        ),
    ]
```

你需要做三处更改：

- 从已生成的迁移类中复制，添加第二个AddField操作，并改为AlterField。
- 在第一个AddField操作中，把unique=True改为 null=True，这会创建一个中间的null字段。

- 在两个操作之间，添加一个RunPython或RunSQL操作为每个已存在的行生成一个唯一值（例如UUID）。

最终的迁移类应该看起来是这样：

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import migrations, models
import uuid

def gen_uuid(apps, schema_editor):
    MyModel = apps.get_model('myapp', 'MyModel')
    for row in MyModel.objects.all():
        row.uuid = uuid.uuid4()
        row.save()

class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0003_auto_20150129_1705'),
    ]

    operations = [
        migrations.AddField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(default=uuid.uuid4, null=True),
        ),
        # omit reverse_code=... if you don't want the migration to be reversible.
        migrations.RunPython(gen_uuid, reverse_code=migrations.RunPython.noop),
        migrations.AlterField(
            model_name='mymodel',
            name='uuid',
            field=models.UUIDField(default=uuid.uuid4, unique=True),
        ),
    ]
```

现在你可以像平常一样使用migrate命令应用迁移。

注意如果你在这个迁移运行时让对象被创建，就会产生竞争条件(race condition)。在AddField之后， RunPython之前创建的对象会覆写他们原始的uuid。

高级

管理器

class Manager

管理器是一个接口，数据库查询操作通过它提供给django的模型。django应用的每个模型至少拥有一个 管理器。

管理器类的工作方式在 执行查询文档中阐述，而这篇文档涉及了自定义管理器行为的模型选项。

管理器的名字

通常，django为每个模型类添加一个名为objects的管理器。然而，如果你想将objects用于字段名称，或者你想使用其它名称而不是objects访问管理器，你可以在每个模型类中重命名它。在模型中定义一个值为models.Manager()的属性，来重命名管理器。例如：

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

使用例子中的模型， Person.objects会抛出AttributeError异常，而Person.people.all()会返回一个包含所有Person对象的列表。

自定义管理器

在一个特定的模型中，你可以通过继承管理器类来构建一个自定义的管理器，以及实例化你的自定义管理器。

你有两个原因可能会自己定义管理器：向器类中添加额外的方法，或者修改管理器最初返回的查询集。

添加额外的管理器方法

为你的模型添加表级(table-level)功能时，采用添加额外的管理器方法是更好的处理方式。如果要添加行级功能——就是说该功能只对某个模型的实例对象起作用。在这种情况下，使用模型方法 比使用自定义的管理器方法要更好。）

自定义的管理器 方法可以返回你想要的任何数据，而不只是查询集。

例如，下面这个自定义的 管理器提供了一个 with_counts() 方法，它返回所有 OpinionPoll 对象的列表，而且列表中的每个对象都多了一个名为 num_responses的属性，这个属性保存一个聚合查询(COUNT*)的结果：

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY p.id, p.question, p.poll_date
            ORDER BY p.poll_date DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

在这个例子中，你已经可以使用 OpinionPoll.objects.with_counts() 得到所有含有 num_responses属性的 OpinionPoll对象。

这个例子要注意的一点是： 管理器方法可以访问 self.model来得到它所用到的模型类。

修改管理器初始的查询集

管理器自带的 查询集返回系统中所有的对象。例如，使用下面这个模型：

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

... Book.objects.all() 语句将返回数据库中所有的 Book 对象。

你可以通过重写 Manager.get_queryset() 的方法来覆盖 管理器自带的 查询集。 get_queryset() 会根据你所需要的属性返回 查询集。

例如，下面的模型有两个 管理器，一个返回所有的对象， 另一个则只返回作者是 Roald Dahl 的对象：

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super(DahlBookManager, self).get_queryset().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

在这个简单的例子中，Book.objects.all()将返回数据库中所有的图书。而 Book.dahl_objects.all() 只返回作者是 Roald Dahl 的图书。

由于 get_queryset() 返回的是一个 查询集 对象，所以你仍可以对它使用 filter(), exclude()和其他 查询集的方法。所以下面这些例子都是可用的：

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

这个例子还展示了另外一个很有意思的技巧：在同一个模型中使用多个管理器。你可以随你所意在一个模型里面添加多个 Manager() 实例。下面就用很简单的方法，给模型添加通用过滤器：

例如：

```
class AuthorManager(models.Manager):
    def get_queryset(self):
        return super(AuthorManager, self).get_queryset().filter(role='A')

class EditorManager(models.Manager):
    def get_queryset(self):
        return super(EditorManager, self).get_queryset().filter(role='E')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Author')), ('E', _('Editor'))
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()
```

在这个例子中，你使用 Person.authors.all(), Person.editors.all(),以及 Person.people.all(), 都会得到和名称相符的结果。

默认管理器

如果你使用了自定义 管理器对象，要注意 Django 中的第一个 管理器 (按照模型中出现的顺序而定) 拥有特殊的地位。Django 会将模型中定义的管理器解释为默认的 管理器，并且 Django 中的一部分应用(包括数据备份)会使用默认的管理器，除了前面那个模型。因此，要决定默认

的管理器时，要小心谨慎，仔细考量，这样才能避免重写 get_queryset() 导致无法正确地获得数据。

使用管理器访问关联对象

默认情况下，在访问相关对象时（例如choice.poll），Django 并不使用相关对象的默认管理器，而是使用一个"朴素"管理器类的实例来访问。这是因为 Django 要能从关联对象中获得数据，但这些数据有可能被默认管理器过滤掉，或是无法进行访问。

如果普通的朴素管理器类(django.db.models.Manager)并不适用于你的应用，那么你可以通过在管理器类中设置 use_for_related_fields ，强制 Django 在你的模型中使用默认的管理器。这部分内容在 下面有 详细介绍。

调用自定义的查询集

虽然大多数标准查询集的方法可以从管理器中直接访问到，但是这是一个例子，访问了定义在自定义 查询集上的额外方法，如果你也在管理器上面实现了它们：

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role='A')

    def editors(self):
        return self.filter(role='E')

class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)

    def authors(self):
        return self.get_queryset().authors()

    def editors(self):
        return self.get_queryset().editors()

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Author')), ('E', _('Editor'))
    people = PersonManager()
```

这个例子展示了如何直接从管理器 Person.people调用authors() 和 editors()。

创建管理器

django 1.7 中新增

对于上面的例子，同一个方法需要在查询集 和 管理器上创建两份副本，作为替代，QuerySet.as_manager()可以创建一个管理器的实例，它拥有自定义查询集的方法：

```
class Person(models.Model):
    ...
    people = PersonQuerySet.as_manager()
```

通过QuerySet.as_manager()创建的管理器 实例，实际上等价于上面例子中的PersonManager。

并不是每个查询集的方法都在管理器层面上有意义。比如 QuerySet.delete()，我们有意防止它复制到管理器 中。

方法按照以下规则进行复制：

- 公共方法默认被复制。
- 私有方法（前面带一个下划线）默认不被复制。
- 带queryset_only 属性，并且值为False的方法总是被复制。
- 带 queryset_only 属性，并且值为True 的方法不会被复制。

例如：

```
class CustomQuerySet(models.QuerySet):
    # Available on both Manager and QuerySet.
    def public_method(self):
        return

    # Available only on QuerySet.
    def _private_method(self):
        return

    # Available only on QuerySet.
    def opted_out_public_method(self):
        return
    opted_out_public_method.queryset_only = True

    # Available on both Manager and QuerySet.
    def _opted_in_private_method(self):
        return
    _opted_in_private_method.queryset_only = False
```

from_queryset

```
classmethod from_queryset(queryset_class)
```

在进一步的使用中，你可能想创建一个自定义管理器和一个自定义查询集。你可以调用Manager.from_queryset()，它会返回管理器的一个子类，带有自定义查询集所有方法的副本：

```
class BaseManager(models.Manager):
    def manager_only_method(self):
        return

class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return

class MyModel(models.Model):
    objects = BaseManager.from_queryset(CustomQueryset)()
```

你也可以在一个变量中储存生成的类：

```
CustomManager = BaseManager.from_queryset(CustomQueryset)

class MyModel(models.Model):
    objects = CustomManager()
```

自定义管理器和模型继承

类继承和模型管理器两者之间配合得并不是很好。 管理器一般只对其定义所在的类起作用，在子类中对其继承绝对不是一个好主意。 而且，因为第一个 管理器会被 Djange 声明为默认的管理器，所以对默认的管理器 进行控制是非常必要的。下面就是 Django 如何处理自定义管理器和模型继承(model inheritance)的：

- 定义在非抽象基类中的管理器是 不会 被子类继承的。如果你想从一个非抽象基类中重用管理器，只能在子类中重定义管理器。 这是因为这种管理器与定义它的模型 绑定得非常紧密，所以继承它们经常会导致异常的结果（特别是默认管理器运行的时候）。 因此，它们不应继承给子类。
- 定义在抽象基类中的管理器总是被子类继续的，是按 Python 的命名解析顺序解析的（首先是子类中的命名覆盖所有的，然后是第一个父类的，以此类推）。 抽象类用来提取子类中的公共信息和行为，定义公共管理器也是公共信息的一部分。
- 如果类当中显示定义了默认管理器，Django 就会以此做为默认管理器；否则就会从第一个抽象基类中继承默认管理器； 如果没有显式声明默认管理器，那么 Django 就会自动添加默认管理器。

如果你想在一组模型上安装一系列自定义管理器，上面提到的这些规则就已经为你的实现提供了必要的灵活性。你可以继承一个抽象基类，但仍要自定义默认的管理器。例如，假设你的基类是这样的：

```
class AbstractBase(models.Model):
    # ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

如果你在基类中没有定义管理器，直接使用上面的代码，默认管理器就是从基类中继承的 objects：

```
class ChildA(AbstractBase):
    # ...
    # This class has CustomManager as the default manager.
    pass
```

如果你想从 AbstractBase继承，却又想提供另一个默认管理器，那么你可以在子类中定义默认管理器：

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

在这个例子中， default_manager就是默认的 管理器。从基类中继承的 objects 管理器仍是可用的。只不过它不再是默认管理器罢了。

最后再举个例子，假设你想在子类中再添加一个额外的管理器，但是很想使用从 AbstractBase继承的管理器做为默认管理器。那么，你不在直接在子类中添加新的管理器，否则就会覆盖掉默认管理器，而且你必须对派生自这个基类的所有子类都显示指定管理器。解决办法就是在另一个基类中添加新的管理器，然后继承时将其放在默认管理器所在的基类之后。例如：

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
    pass
```

注意在抽象模型上面定义一个自定义管理器的时候，不能调用任何使用这个抽象模型的方法。就像：

```
ClassA.objects.do_something()
```

是可以的，但是：

```
AbstractBase.objects.do_something()
```

会抛出一个异常。这是因为，管理器被设计用来封装对象集合管理的逻辑。由于抽象的对象中并没有一个集合，管理它们是毫无意义的。如果你写了应用在抽象模型上的功能，你应该把功能放到抽象模型的静态方法，或者类的方法中。

实现上的注意事项

无论你向自定义管理器中添加了什么功能，都必须可以得到 管理器实例的一个浅表副本：例如，下面的代码必须正常运行：

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django 在一些查询中会创建管理器的浅表副本；如果你的管理器不能被复制，查询就会失败。

这对于大多数自定义管理器不是什么大问题。如果你只是添加一些简单的方法到你的管理器中，不太可能会把你的管理器实例变为不可复制的。但是，如果你覆盖了__getattr__，或者其它管理器中控制对象状态的私有方法，你应该确保不会影响到管理器的复制。

控制自动管理器的类型

这篇文档已经提到了Django创建管理器类的一些位置：默认管理器和用于访问关联对象的“朴素” 管理器。在 Django 的实现中也有很多地方用到了临时的朴素管理器。 正常情况下，django.db.models.Manager 类的实例会自动创建管理器。

在整个这一节中，我们将那种由 Django 为你创建的管理器称之为 "自动管理器"，既有因为没有管理器而被 Django 自动添加的默认管理器， 也包括在访问关联模型时使用的临时管理器。

有时，默认管理器也并非是自动管理器。 一个例子就是 Django 自带的django.contrib.gis 应用，所有 gis模型都必须使用一个特殊的管理器类(GeoManager)，因为它们需要运行特殊的查询集(GeoQuerySet)与数据库进行交互。这表明无论自动管理器是否被创建，那些要使用特殊的管理器的模型仍要使用这个特殊的管理器类。

Django 为自定义管理器的开发者提供了一种方式：无论开发的管理器类是不是默认的管理器，它都应该可以用做自动管理器。 可以通过在管理器类中设置 use_for_related_fields 属性来做到这点：

```
class MyManager(models.Manager):
    use_for_related_fields = True
    # ...
```

如果在模型中的默认 管理器(在这些情况中仅考虑默认管理器)中设置了这个属性，那么无论它是否需要被自动创建，Django 都会自动使用它。否则 Django 就会使用 django.db.models.Manager.

> 历史回顾
>
> 从它使用目的来看，这个属性的名称(use_for_related_fields)好象有点古怪。原本，这个属性仅仅是用来控制访问关联字段的管理器的类型，这就是它名字的由来。 后来它的作用更加拓宽了，但是名称一直未变。 因为要保证现在的代码在 Django 以后的版本中仍可以正常工作(continue to work)，这就是它名称不变的原因。

在自动管理器实例中编写正确的管理器

在上面的django.contrib.gis 已经提到了， use_for_related_fields这个特性是在需要返回一个自定义查询集子类的管理器中使用的。要在你的管理器中提供这个功能，要注意以下几点。

不要在这种类型的管理器子类中过滤掉任何结果

一个原因是自动管理器是用来访问关联模型 的对象。 在这种情况下，Django 必须要能看到相关模型的所有对象，所以才能根据关联关系得到任何数据 。

如果你重写了 get_queryset() 方法并且过滤掉了一些行数据，Django 将返回不正确的结果。不要这么做！ 在 get_queryset()方法中过滤掉数据，会使得它所在的管理器不适于用做自动管理器。

设置 use_for_related_fields

use_for_related_fields属性必须在管理器类中设置，而不是在类的 实例中设置。上面已经有例子展示如何正确地设置，下面这个例子就是一个错误的示范：

```
# BAD: Incorrect code
class MyManager(models.Manager):
    # ...
    pass

# Sets the attribute on an instance of MyManager. Django will
# ignore this setting.
mgr = MyManager()
mgr.use_for_related_fields = True

class MyModel(models.Model):
    # ...
    objects = mgr

# End of incorrect code.
```

你也不应该在模型中使用这个属性之后，在类上改变它。这是因为在模型类被创建时，这个属性值马上就会被处理，而且随后不会再读取这个属性值。 这节的第一个例子就是在第一次定义的时候在管理器上设置use_for_related_fields属性，所有的代码就工作得很好。

进行原始的**sql**查询

在模型查询*API*不够用的情况下，你可以使用原始的sql语句。django提供两种方法使用原始sql进行查询：一种是使用**Manager.raw()**方法，进行原始查询并返回模型实例；另一种是完全避开模型层，直接执行自定义的*sql*语句。

> 警告
>
> 编写原始的sql语句时，应该格外小心。每次使用的时候，都要确保转义了参数中的任何控制字符，以防受到sql注入攻击。更多信息请参阅防止*sql*注入。

进行原始查询

raw()方法用于原始的sql查询，并返回模型的实例：

```
Manager.raw(raw_query, params=None, translations=None)
```

这个方法执行原始的sql查询之后，返回**django.db.models.query.RawQuerySet**的实例。**RawQuerySet**实例可以像一般的**QuerySet**那样，通过迭代来提供对象的实例。

这里最好通过例子展示一下，假设存在以下模型：

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

你可以像这样执行自定义的sql语句：

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

当然，这个例子不是特别有趣，和直接使用**Person.objects.all()**的结果一模一样。但是，**raw()**拥有其它更强大的使用方法。

> 模型表的名称
>
> 在上面的例子中，**Person**表的名称是从哪里得到的？
>
> 通常，Django通过将模型的名称和模型的“应用标签”（你在**manage.py startapp**中使用的名称）进行关联，用一条下划线连接他们，来组合表的名称。在这里我们假定**Person**模型存在于一个叫做**myapp**的应用中，所以表就应该叫做**myapp_person**。
>
> 更多细节请查看**db_table**选项的文档，它也可以让你自定义表的名称。
>
> 警告
>
> 传递给**raw()**方法的sql语句并没有任何检查。django默认它会返回一个数据集，但这不是强制性的。如果查询的结果不是数据集，则会产生一个错误。
>
> 警告
>
> 如果你在mysql上执行查询，注意在类型不一致的时候，mysql的静默类型强制可能导致意想不到的结果发生。如果你在一个字符串类型的列上查询一个整数类型的值，mysql会在比较前强制把每个值的类型转成整数。例如，如果你的表中包含值**'abc'**和**'def'**，你查询**'where mycolumn=0'**，那么两行都会匹配。要防止这种情况，在查询中使用值之前，要做好正确的类型转换。
>
> 警告
>
> 虽然**RawQuerySet**可以像普通的**QuerySet**一样迭代，**RawQuerySet**并没有实现可以在**QuerySet**上使用的所有方法。例如，**__bool__()**和**__len__()**在**RawQuerySet**中没有被定义，所以所有**RawQuerySet**转化为布尔值的结果都是**True**。**RawQuerySet**中没有实现他们的原因是，在没有内部缓存的情况下会导致性能下降，而且增加内部缓存不向后兼容。

将查询字段映射到模型字段

raw()方法自动将查询字段映射到模型字段。

字段的顺序并不重要。换句话说，下面两种查询的作用相同：

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM myapp_person')
...
```

Django会根据名字进行匹配。这意味着你可以使用sql的**as**子句来映射二者。所以如果在其他的表中有一些**Person**数据，你可以很容易地把它们映射成**Person**实例。

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                              last AS last_name,
...                              bd AS birth_date,
...                              pk AS id,
...                         FROM some_other_table''')
```

只要名字能对应上，模型的实例就会被正确创建。 又或者，你可以在**raw()**方法中使用翻译参数。翻译参数是一个字典，将表中的字段名称映射为模型中的字段名称、例如，上面的查询可以写成这样：

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date', 'pk': 'id
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

索引访问

raw()方法支持索引访问，所以如果只需要第一条记录，可以这样写：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

然而，索引和切片并不在数据库层面上进行操作。如果数据库中有很多的**Person**对象，更加高效的方法是在sql层面限制查询中结果的数量：

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person LIMIT 1')[0]
```

延迟加载模型字段

字段也可以被省略：

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

查询返回的Person对象是一个延迟的模型实例（请见 **defer()**）。这意味着被省略的字段，在访问时才被加载。例如:

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be retrieved by the original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

从表面上来看，看起来这个查询获取了**first_name**和**last_name**。然而，这个例子实际上执行了3次查询。只有**first_name**字段在**raw()**查询中获取，**last_name**字符按在执行打印命令时才被获取。

只有一种字段不可以被省略，就是主键。Django 使用主键来识别模型的实例，所以它在每次原始查询中都必须包含。如果你忘记包含主键的话，会抛出一个**InvalidQuery**异常。

增加注解

你也可以在查询中包含模型中没有定义的字段。例如，我们可以使用**PostgreSQL**的**age()**函数来获得一群人的列表，带有数据库计算出的年龄。

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

向 raw() 方法中传递参数

如果你需要参数化的查询，可以向**raw()** 方法传递**params**参数。

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

params是存放参数的列表或字典。你可以在查询语句中使用**%s**占位符，或者对于字典使用**%(key)**占位符（**key**会被替换成字典中键为key的值），无论你的数据库引擎是什么。这样的占位符会被替换成参数表中正确的参数。

> 注意
>
> SQLite后端不支持字典，你必须以列表的形式传递参数。
>
> 警告
>
> 不要在原始查询中使用字符串格式化！
>
> 它类似于这种样子：
>
> ```
> >> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
> >> Person.objects.raw(query)
> ```
>
> 使用参数化查询可以完全防止sql注入，一种普遍的漏洞使攻击者可以向你的数据库中注入任何sql语句。如果你使用字符串格式化，早晚会受到sql输入的攻击。只要你记住默认使用参数化查询，就可以免于攻击。

直接执行自定义sql

有时**Manager.raw()**方法并不十分好用，你不需要将查询结果映射成模型，或者你需要执行**UPDATE**、**INSERT**以及**DELETE**查询。

在这些情况下，你可以直接访问数据库，完全避开模型层。

django.db.connection对象提供了常规数据库连接的方式。为了使用数据库连接，调用**connection.cursor()**方法来获取一个游标对象之后，调用**cursor.execute(sql, [params])**来执行sql语句，调用**cursor.fetchone()**或者**curser.fetchall()**来返回结果行。

例如:

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])

    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

注意如果你的查询中包含百分号字符，你需要写成两个百分号字符，以便能正确传递参数：

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s", [self.id])
```

如果你使用了不止一个数据库，你可以使用**django.db.connections**来获取针对特定数据库的连接（以及游标）对象。**django.db.connections**是一个类似于字典的对象，允许你通过它的别名获取特定的连接

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
```

通常，Python DB API会返回不带字段的结果，这意味着你需要以一个列表结束，而不是一个字典。花费一点性能之后，你可以返回一个字典形式的结果，像这样：

```
def dictfetchall(cursor):
    "Returns all rows from a cursor as a dict"
    desc = cursor.description
    return [
        dict(zip([col[0] for col in desc], row))
        for row in cursor.fetchall()
    ]
```

下面是一个体现二者区别的例子:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982L, None), (54360880L, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id': 54360880L}]
```

连接和游标

连接和游标主要实现PEP 249中描述的Python DB API标准，除非它涉及到事务处理。

如果你不熟悉Python DB-API，注意**cursor.execute()**中的sql语句使用占位符**"%s"**，而不是直接在sql中添加参数。如果你使用它，下面的数据库会在必要时自动转义你的参数。

也要注意Django使用**"%s"**占位符，而不是SQLite Python绑定的**"?"**占位符。这是一致性和可用性的缘故。

```
Django 1.7中的改变。
```

PEP 249并没有说明游标是否可以作为上下文管理器使用。在python2.7之前，游标可以用作上下文管理器，由于魔术方法lookups中意想不到的行为(Python ticket #9220)。Django 1.7显式添加了对允许游标作为上下文管理器使用的支持。

将游标作为上下文管理器使用:

```
with connection.cursor() as c:
    c.execute(...)
```

等价于：

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

聚合

Django数据库抽象API描述了使用Django查询来增删查改单个对象的方法。然而，你有时候会想要获取从一组对象导出的值或者是聚合一组对象。这份指南描述了通过Django查询来生成和返回聚合值的方法。

整篇指南我们都将引用以下模型。这些模型用来记录多个网上书店的库存。

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

class Publisher(models.Model):
    name = models.CharField(max_length=300)
    num_awards = models.IntegerField()

class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    pubdate = models.DateField()

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
    registered_users = models.PositiveIntegerField()
```

速查表

急着用吗？以下是在上述模型的基础上，进行一般的聚合查询的方法:

```
# Total number of books.
>>> Book.objects.count()
2452

# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name='BaloneyPress').count()
73

# Average price across all books.
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

# Max price across all books.
>>> from django.db.models import Max
>>> Book.objects.all().aggregate(Max('price'))
{'price__max': Decimal('81.20')}

# Cost per page
>>> Book.objects.all().aggregate(
...    price_per_page=Sum(F('price')/F('pages'), output_field=FloatField()))
{'price_per_page': 0.4470664529184653}

# All the following queries involve traversing the Book<->Publisher
# many-to-many relationship backward

# Each publisher, each with a count of books as a "num_books" attribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book'))
>>> pubs
[<Publisher BaloneyPress>, <Publisher SalamiPress>, ...]
>>> pubs[0].num_books
73

# The top 5 publishers, in order by number of books.
>>> pubs = Publisher.objects.annotate(num_books=Count('book')).order_by('-num_books')[:5]
>>> pubs[0].num_books
1323
```

在查询集上生成聚合

Django提供了两种生成聚合的方法。第一种方法是从整个查询集生成统计值。比如，你想要计算所有在售书的平均价钱。Django的查询语法提供了一种方式描述所有图书的集合。

```
>>> Book.objects.all()
```

我们需要在QuerySet.对象上计算出总价格。这可以通过在QuerySet后面附加aggregate() 子句来完成。

```
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}
```

all()在这里是多余的，所以可以简化为：

```
>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 34.35}
```

aggregate()子句的参数描述了我们想要计算的聚合值，在这个例子中，是Book 模型中price字段的平均值。查询集参考中列出了聚合函数的列表。

aggregate()是QuerySet 的一个终止子句，意思是说，它返回一个包含一些键值对的字典。键的名称是聚合值的标识符，值是计算出来的聚合值。键的名称是按照字段和聚合函数的名称自动生成出来的。如果你想要为聚合值指定一个名称，可以向聚合子句提供它。

```
>>> Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 34.35}
```

如果你希望生成不止一个聚合，你可以向aggregate()子句中添加另一个参数。所以，如果你也想知道所有图书价格的最大值和最小值，可以这样查询：

```
>>> from django.db.models import Avg, Max, Min
>>> Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min': Decimal('12.99')}
```

为查询集的每一项生成聚合

生成汇总值的第二种方法，是为QuerySet中每一个对象都生成一个独立的汇总值。比如，如果你在检索一列图书，你可能想知道有多少作者写了每一本书。每本书和作者是多对多的关系。我们想要汇总QuerySet.中每本书里的这种关系。

逐个对象的汇总结果可以由annotate()子句生成。当annotate()子句被指定之后，QuerySet中的每个对象都会被注上特定的值。

这些注解的语法都和aggregate()子句所使用的相同。annotate()的每个参数都描述了将要被计算的聚合。比如，给图书添加作者数量的注解：

```
# Build an annotated queryset
>>> from django.db.models import Count
>>> q = Book.objects.annotate(Count('authors'))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

和使用 aggregate()一样，注解的名称也根据聚合函式的名称和聚合字段的名称得到的。你可以在指定注解时，为默认名称提供一个别名：

```
>>> q = Book.objects.annotate(num_authors=Count('authors'))
>>> q[0].num_authors
2
>>> q[1].num_authors
1
```

与 aggregate() 不同的是， annotate() 不是一个终止子句。annotate()子句的返回结果是一个查询集 (QuerySet)；这个 QuerySet可以用任何QuerySet方法进行修改，包括 filter(), order_by(), 甚至是再次应用annotate()。

> 有任何疑问的话，请检查 SQL query！
>
> 要想弄清楚你的查询到底发生了什么，可以考虑检查你QuerySet的 query 属性。
>
> 例如，在annotate() 中混入多个聚合将会得出错误的结果，因为多个表上做了交叉连接，导致了多余的行聚合。

连接和聚合

至此，我们已经了解了作用于单种模型实例的聚合操作， 但是有时，你也想对所查询对象的关联对象进行聚合。

在聚合函式中指定聚合字段时，Django 允许你使用同样的 双下划线 表示关联关系，然后 Django 在就会处理要读取的关联表，并得到关联对象的聚合。

例如，要得到每个书店的价格区别，可以使用如下注解：

```
>>> from django.db.models import Max, Min
>>> Store.objects.annotate(min_price=Min('books__price'), max_price=Max('books__price'))
```

这段代码告诉 Django 获取书店模型，并连接(通过多对多关系)图书模型，然后对每本书的价格进行聚合，得出最小值和最大值。

同样的规则也用于 aggregate() 子句。如果你想知道所有书店中最便宜的书和最贵的书价格分别是多少：

```
>>> Store.objects.aggregate(min_price=Min('books__price'), max_price=Max('books__price'))
```

关系链可以按你的要求一直延伸。 例如，想得到所有作者当中最小的年龄是多少，就可以这样写：

```
>>> Store.objects.aggregate(youngest_age=Min('books__authors__age'))
```

遵循反向关系

和 跨关系查找的方法类似，作用在你所查询的模型的关联模型或者字段上的聚合和注解可以遍历"反转"关系。关联模型的小写名称和双下划线也用在这里。

例如，我们可以查询所有出版商，并注上它们一共出了多少本书（注意我们如何用 'book'指定 Publisher -> Book 的外键反转关系）：

```
>>> from django.db.models import Count, Min, Sum, Avg
>>> Publisher.objects.annotate(Count('book'))
```

QuerySet结果中的每一个Publisher都会包含一个额外的属性叫做book__count。

我们也可以按照每个出版商，查询所有图书中最旧的那本：

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min('book__pubdate'))
```

（返回的字典会包含一个键叫做 'oldest_pubdate'。如果没有指定这样的别名，它会更长一些，像 'book**pubdate**min'。）

这不仅仅可以应用挂在外键上面。还可以用到多对多关系上。例如，我们可以查询每个作者，注上它写的所有书（以及合著的书）一共有多少页（注意我们如何使用 'book'来指定 Author -> Book的多对多的反转关系）：

```
>>> Author.objects.annotate(total_pages=Sum('book__pages'))
```

（每个返回的QuerySet中的Author 都有一个额外的属性叫做total_pages。如果没有指定这样的别名，它会更长一些，像 book**pages**sum。）

或者查询所有图书的平均评分，这些图书由我们存档过的作者所写：

```
>>> Author.objects.aggregate(average_rating=Avg('book__rating'))
```

（返回的字典会包含一个键叫做'average**rating'**。如果没有指定这样的别名，它会更长一些，像**'book**rating__avg'。）

聚合和其他查询集子句

filter() 和 exclude()

聚合也可以在过滤器中使用。 作用于普通模型字段的任何 filter()(或 exclude()) 都会对聚合涉及的对象进行限制。

使用annotate() 子句时，过滤器有限制注解对象的作用。例如，你想得到以 "Django" 为书名开头的图书作者的总数：

```
>>> from django.db.models import Count, Avg
>>> Book.objects.filter(name__startswith="Django").annotate(num_authors=Count('authors'))
```

使用aggregate()子句时，过滤器有限制聚合对象的作用。例如，你可以算出所有以 "Django" 为书名开头的图书平均价格：

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg('price'))
```

对注解过滤

注解值也可以被过滤。 像使用其他模型字段一样，注解也可以在filter()和exclude() 子句中使用别名。

例如，要得到不止一个作者的图书，可以用：

```
>>> Book.objects.annotate(num_authors=Count('authors')).filter(num_authors__gt=1)
```

这个查询首先生成一个注解结果，然后再生成一个作用于注解上的过滤器。

annotate() 的顺序

编写一个包含 annotate() 和 filter() 子句的复杂查询时，要特别注意作用于 QuerySet的子句的顺序。

当一个annotate() 子句作用于某个查询时，要根据查询的状态才能得出注解值，而状态由 annotate() 位置所决定。以这就导致filter() 和 annotate() 不能交换顺序，下面两个查询就是不同的：

```
>>> Publisher.objects.annotate(num_books=Count('book')).filter(book__rating__gt=3.0)
```

另一个查询：

```
>>> Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))
```

两个查询都返回了至少出版了一本好书(评分大于 3 分)的出版商。 但是第一个查询的注解包含其该出版商发行的所有图书的总数；而第二个查询的注解只包含出版过好书的出版商的所发行的图书总数。 在第一个查询中，注解在过滤器之前，所以过滤器对注解没有影响。 在第

二个查询中，过滤器在注解之前，所以，在计算注解值时，过滤器就限制了参与运算的对象的范围。

order_by()

注解可以用来做为排序项。 在你定义 order_by() 子句时，你提供的聚合可以引用定义的任何别名做为查询中 annotate()子句的一部分。

例如，根据一本图书作者数量的多少对查询集 QuerySet进行排序：

```
>>> Book.objects.annotate(num_authors=Count('authors')).order_by('num_authors')
```

values()

通常，注解会添加到每个对象上 —— 一个被注解的QuerySet会为初始QuerySet的每个对象返回一个结果集。但是，如果使用了values()子句，它就会限制结果中列的范围， 对注解赋值的方法就会完全不同。不是在原始的 QuerySet返回结果中对每个对象中添加注解，而是根据定义在values() 子句中的字段组合对先结果进行唯一的分组，再根据每个分组算出注解值，这个注解值是根据分组中所有的成员计算而得的：

例如，考虑一个关于作者的查询， 查询出每个作者所写的书的平均评分：

```
>>> Author.objects.annotate(average_rating=Avg('book__rating'))
```

这段代码返回的是数据库中所有的作者以及他们所著图书的平均评分。

但是如果你使用了values()子句， 结果是完全不同的：

```
>>> Author.objects.values('name').annotate(average_rating=Avg('book__rating'))
```

在这个例子中，作者会按名称分组，所以你只能得到某个唯一的作者分组的注解值。 这意味着如果你有两个作者同名，那么他们原本各自的查询结果将被合并到同一个结果中；两个作者的所有评分都将被计算为一个平均分。

annotate() 的顺序

和使用 filter() 子句一样，作用于某个查询的annotate() 和 values() 子句的使用顺序是非常重要的。如果values() 子句在 annotate() 之前，就会根据 values() 子句产生的分组来计算注解。

但是，如果 annotate() 子句在 values()子句之前，就会根据整个查询集生成注解。在这种情况下，values() 子句只能限制输出的字段范围。

举个例子，如果我们互换了上个例子中 values()和 annotate() 子句的顺序：

```
>>> Author.objects.annotate(average_rating=Avg('book__rating')).values('name', 'average_r
```

这段代码将给每个作者添加一个唯一的字段，但只有作者名称和average_rating 注解会返回在输出结果中。

你也应该注意到 average_rating 显式地包含在返回的列表当中。之所以这么做的原因正是因为values() 和 annotate() 子句。

如果 values() 子句在 annotate() 子句之前，注解会被自动添加到结果集中；但是，如果 values() 子句作用于annotate() 子句之后，你需要显式地包含聚合列。

与默认排序或order_by()交互

在查询集中的order_by() 部分(或是在模型中默认定义的排序项) 会在选择输出数据时被用到，即使这些字段没有在values() 调用中被指定。这些额外的字段可以将相似的数据行分在一起，也可以让相同的数据行相分离。在做计数时，就会表现地格外明显：

通过例子中的方法，假设有一个这样的模型：

```
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()

    class Meta:
        ordering = ["name"]
```

关键的部分就是在模型默认排序项中设置的name字段。如果你想知道每个非重复的data值出现的次数，可以这样写：

```
# Warning: not quite correct!
Item.objects.values("data").annotate(Count("id"))
```

...这部分代码想通过使用它们公共的 data 值来分组 Item对象，然后在每个分组中得到 id 值的总数。但是上面那样做是行不通的。这是因为默认排序项中的 name也是一个分组项，所以这个查询会根据非重复的 (data, name) 进行分组，而这并不是你本来想要的结果。所以，你应该这样改写：

```
Item.objects.values("data").annotate(Count("id")).order_by()
```

...这样就清空了查询中的所有排序项。 你也可以在其中使用 data，这样并不会有副作用，这是因为查询分组中只有这么一个角色了。

这个行为与查询集文档中提到的 distinct() 一样，而且生成规则也一样：一般情况下，你不想在结果中由额外的字段扮演这个角色，那就清空排序项，或是至少保证它仅能访问 values()中的字段。

> 注意
>
> 你可能想知道为什么 Django 不删除与你无关的列。主要原因就是要保证使用 distinct()和其他方法的一致性。Django 永远不会 删除你所指定的排序限制(我们不能改动那些方法的行为，因为这会违背 API stability 原则)。

聚合注解

你也可以在注解的结果上生成聚合。 当你定义一个 aggregate() 子句时，你提供的聚合会引用定义的任何别名做为查询中 annotate() 子句的一部分。

例如，如果你想计算每本书平均有几个作者，你先用作者总数注解图书集，然后再聚合作者总数，引入注解字段：

```
>>> from django.db.models import Count, Avg
>>> Book.objects.annotate(num_authors=Count('authors')).aggregate(Avg('num_authors'))
{'num_authors__avg': 1.66}
```

多数据库

这篇主题描述Django 对多个数据库的支持。大部分Django 文档假设你只和一个数据库打交道。如果你想与多个数据库打交道，你将需要一些额外的步骤。

定义你的数据库

在Django中使用多个数据库的第一步是告诉Django 你将要使用的数据库服务器。这通过使用 `DATABASES` 设置完成。该设置映射数据库别名到一个数据库连接设置的字典，这是整个Django 中引用一个数据库的方式。字典中的设置在 `DATABASES` 文档中有完整描述。

你可以为数据库选择任何别名。然而， `default` 这个别名具有特殊的含义。当没有选择其它数据库时，Django 使用具有 `default` 别名的数据库。

下面是 `settings.py` 的一个示例片段，它定义两个数据库 —— 一个默认的PostgreSQL 数据库和一个叫做 `users` 的MySQL 数据库：

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'USER': 'postgres_user',
        'PASSWORD': 's3krit'
    },
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te'
    }
}
```

如果 `default` 数据库在你的项目中不合适，你需要小心地永远指定是想使用的数据库。Django 要求 `default` 数据库必须定义，但是其参数字典可以保留为空如果不使用它。若要这样做，你必须为你的所有的应用的模型建立 `DATABASE_ROUTERS` ，包括正在使用的 `contrib` 中的应用和第三方应用，以使得不会有查询被路由到默认的数据库。下面是 `settings.py` 的一个示例片段，它定义两个非默认的数据库，其中 `default` 有意保留为空：

```
DATABASES = {
    'default': {},
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'superS3cret'
    },
    'customers': {
        'NAME': 'customer_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_cust',
        'PASSWORD': 'veryPriv@ate'
    }
}
```

如果你试图访问在 `DATABASES` 设置中没有定义的数据库，Django 将抛出一个 `django.db.utils.ConnectionDoesNotExist` 异常。

同步你的数据库

`migrate` 管理命令一次操作一个数据库。默认情况下，它在 `default` 数据库上操作，但是通过提供一个 `--database` 参数，你可以告诉 `migrate` 同步一个不同的数据库。因此，为了同步所有模型到我们示例中的所有数据库，你将需要调用：

```
$ ./manage.py migrate
$ ./manage.py migrate --database=users
```

如果你不想每个应用都被同步到同一台数据库上，你可以定义一个数据库路由，它实现一个策略来控制特定模型的访问性。

使用其它管理命令

其它 `django-admin` 命令与数据库交互的方式与 `migrate` 相同 —— 它们都一次只操作一个数据库，并使用 `--database` 来控制使用的数据库。

数据库自动路由

使用多数据库最简单的方法是建立一个数据库路由模式。默认的路由模式确保对象'粘滞'在它们原始的数据库上（例如，从 `foo` 数据库中获取的对象将保存在同一个数据库中）。默认的路由模式还确保如果没有指明数据库，所有的查询都回归到 `default` 数据库中。

你不需要做任何事情来激活默认的路由模式 —— 它在每个Django项目上'直接'提供。然而，如果你想实现更有趣的数据库分配行为，你可以定义并安装你自己的数据库路由。

数据库路由

数据库路由是一个类，它提供4个方法：

```
db_for_read(model, **hints)
```

建议model类型的对象的读操作应该使用的数据库。

如果一个数据库操作能够提供其它额外的信息可以帮助选择一个数据库，它将在 `hints` 字典中提供。合法的 `hints` 的详细信息在下文给出。

如果没有建议，则返回 `None` 。

```
db_for_write(model, **hints)
```

建议Model 类型的对象的写操作应该使用的数据库。

如果一个数据库操作能够提供其它额外的信息可以帮助选择一个数据库，它将在 `hints` 字典中提供。 合法的 `hints` 的详细信息在下文给出。

如果没有建议，则返回None。

```
allow_relation(obj1, obj2, **hints)
```

如果 `obj1` 和 `obj2` 之间应该允许关联则返回 `True` ，如果应该防止关联则返回 `False` ，如果路由无法判断则返回 `None` 。这是纯粹的验证操作，外键和多对多操作使用它来决定两个对象之间是否应该允许一个关联。

```
allow_migrate(db, app_label, model_name=None, **hints)
```

定义迁移操作是否允许在别名为 `db` 的数据库上运行。如果操作应该运行则返回 `True` ，如果不应该运行则返回 `False` ，如果路由无法判断则返回 `None` 。

位置参数 `app_label` 是正在迁移的应用的标签。

大部分迁移操作设置 `model_name` 的值为正在迁移的模型的 `model._meta.model_name` （模型的 `__name__` 的小写）。对于RunPython和RunSQL 操作它的值为 `None` ，除非这两个操作使用hint 提供它。

`hints` 用于某些操作来传递额外的信息给路由。

当设置了 `model_name` 时， `hints` 通常通过键' `model` '包含该模型的类。注意，它可能是一个历史模型，因此不会有自定的属性、方法或管理器。你应该只依赖 `_meta` 。

这个方法还可以用来决定一个给定数据库上某个模型的可用性。

注意，如果这个方法返回 `False` ，迁移将默默地不会在模型上做任何操作。这可能导致你应用某些操作之后出现损坏的外键、表多余或者缺失。

```
Changed in Django 1.8:

The signature of allow_migrate has changed significantly from previous versions. See the
```

路由不必提供所有这些方法 —— 它可以省略一个或多个。如果某个方法缺失，在做相应的检查时Django 将忽略该路由。

Hints

`Hint` 由数据库路由接收，用于决定哪个数据库应该接收一个给定的请求。

目前，唯一一个提供的hint 是instance，它是一个对象实例，与正在进行的读或者写操作关联。This might be the instance that is being saved, or it might be an instance that is being added in a many-to-many relation. In some cases, no instance hint will be provided at all. The router checks for the existence of an instance hint, and determine if that hint should be used to alter routing behavior.

使用路由

数据库路由使用 `DATABASE_ROUTERS` 设置安装。这个设置定义一个类名的列表，其中每个类表示一个路由，它们将被主路由（ `django.db.router` ）使用。

Django 的数据库操作使用主路由来分配数据库的使用。每当一个查询需要知道使用哪一个数据库时，它将调用主路由，并提供一个模型和一个 `Hint` （可选）。Django 然后依次测试每个路由直至找到一个数据库的建议。如果找不到建议，它将尝试 `Hint` 实例的当前 `_state.db` 。如果没有提供Hint 实例，或者该实例当前没有数据库状态，主路由将分配 `default` 数据库。

一个例子

> 只是为了示例！
>
> 这个例子的目的是演示如何使用路由这个基本结构来改变数据库的使用。它有意忽略一些复杂的问题，目的是为了演示如何使用路由。
>
> 如果 `myapp` 中的任何一个模型包含与其它 数据库之外的模型的关联，这个例子将不能工作。跨数据的关联引入引用完整性问题，Django目前还无法处理。
>
> `Primary/replica` （在某些数据库中叫做 `master/slave` ）配置也是有缺陷的 —— 它不提供任何处理Replication lag 的解决办法（例如，因为写入同步到replica 需要一定的时间，这会引入查询的不一致）。It also doesn't consider the interaction of transactions with the database utilization strategy.

那么 —— 在实际应用中这以为着什么？让我们看一下另外一个配置的例子。这个配置将有几个数据库：一个用于 `auth` 应用，所有其它应用使用一个具有两个读 `replica` 的 `primary/replica` 。下面是表示这些数据库的设置：

```
DATABASES = {
    'auth_db': {
        'NAME': 'auth_db',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'swordfish',
    },
    'primary': {
        'NAME': 'primary',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'spam',
    },
    'replica1': {
        'NAME': 'replica1',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'eggs',
    },
    'replica2': {
        'NAME': 'replica2',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'bacon',
    },
}
```

现在我们将需要处理路由。首先，我们需要一个路由，它知道发送 `auth` 应用的查询到 `auth_db`：

```
class AuthRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def db_for_write(self, model, **hints):
        """
        Attempts to write auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def allow_relation(self, obj1, obj2, **hints):
        """
        Allow relations if a model in the auth app is involved.
        """
        if obj1._meta.app_label == 'auth' or \
           obj2._meta.app_label == 'auth':
           return True
        return None

    def allow_migrate(self, db, app_label, model=None, **hints):
        """
        Make sure the auth app only appears in the 'auth_db'
        database.
        """
        if app_label == 'auth':
            return db == 'auth_db'
        return None
```

我们还需要一个路由，它发送所有其它应用的查询到 `primary/replica` 配置，并随机选择一个 `replica` 来读取：

```
import random

class PrimaryReplicaRouter(object):
    def db_for_read(self, model, **hints):
        """
        Reads go to a randomly-chosen replica.
        """
        return random.choice(['replica1', 'replica2'])

    def db_for_write(self, model, **hints):
        """
        Writes always go to primary.
        """
        return 'primary'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the primary/replica pool.
        """
        db_list = ('primary', 'replica1', 'replica2')
        if obj1._state.db in db_list and obj2._state.db in db_list:
            return True
        return None

    def allow_migrate(self, db, app_label, model=None, **hints):
        """
        All non-auth models end up in this pool.
        """
        return True
```

最后，在设置文件中，我们添加如下内容（替换 `path.to.` 为该路由定义所在的真正路径）：

```
DATABASE_ROUTERS = ['path.to.AuthRouter', 'path.to.PrimaryReplicaRouter']
```

路由处理的顺序非常重要。路由的查询将按照DATABASE_ROUTERS设置中列出的顺序进行。在这个例子中， `AuthRouter` 在 `PrimaryReplicaRouter` 之前处理，因此 `auth` 中的模型的查询处理在其它模型之前。如果 `DATABASE_ROUTERS` 设置按其它顺序列出这两个路由， `PrimaryReplicaRouter.allow_migrate()` 将先处理。 `PrimaryReplicaRouter` 中实现的捕获所有的查询，这意味着所有的模型可以位于所有的数据库中。

建立这个配置后，让我们运行一些Django 代码：

```
>>> # This retrieval will be performed on the 'auth_db' database
>>> fred = User.objects.get(username='fred')
>>> fred.first_name = 'Frederick'

>>> # This save will also be directed to 'auth_db'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a replica database
>>> dna = Person.objects.get(name='Douglas Adams')

>>> # A new object has no database allocation when created
>>> mh = Book(title='Mostly Harmless')

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the primary database...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on a replica
>>> mh = Book.objects.get(title='Mostly Harmless')
```

手动选择一个数据库

Django 还提供一个API，允许你在你的代码中完全控制数据库的使用。人工指定的数据库的优先级高于路由分配的数据库。

为QuerySet手动选择一个数据库

你可以在 `QuerySet` “链”的任意节点上为 `QuerySet` 选择数据库 。只需要 在`QuerySet` 上调用 `using()` 就可以让 `QuerySet` 使用一个指定的数据库。

`using()` 接收单个参数：你的查询想要运行的数据库的别名。例如：

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

为save() 选择一个数据库

对 `Model.save()` 使用 `using` 关键字来指定数据应该保存在哪个数据库。

例如，若要保存一个对象到legacy_users 数据库，你应该使用：

```
>>> my_object.save(using='legacy_users')
```

如果你不指定 `using`， `save()` 方法将保存到路由分配的默认数据库中。

将对象从一个数据库移动到另一个数据库

如果你已经保存一个实例到一个数据库中，你可能很想使用 `save(using=...)` 来迁移该实例到一个新的数据库中。然而，如果你不使用正确的步骤，这可能导致意外的结果。

考虑下面的例子：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')  # (statement 1)
>>> p.save(using='second') # (statement 2)
```

在statement 1中，一个新的 `Person` 对象被保存到 `first` 数据库中。此时 `p` 没有主键，所以Django 发出一个 `SQL INSERT` 语句。这会创建一个主键，且Django 将此主键赋值给 `p`。

当保存在statement 2中发生时， `p` 已经具有一个主键，Django 将尝试在新的数据库上使用该主键。如果该主键值在 `second` 数据库中没有使用，那么你不会遇到问题 —— 该对象将被复制到新的数据库中。

然而，如果 `p` 的主键在 `second` 数据库上已经在使用 `second` 数据库中的已经存在的对象将在 `p` 保存时被覆盖。

你可以用两种方法避免这种情况。首先，你可以清除实例的主键。如果一个对象没有主键，Django 将把它当做一个新的对象，这将避免 `second` 数据库上数据的丢失：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.pk = None # Clear the primary key.
>>> p.save(using='second') # Write a completely new object.
```

第二种方法是使用 `force_insert` 选项来 `save()` 以确保Django 使用一个 `INSERT SQL`：

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.save(using='second', force_insert=True)
```

这将确保名称为 `Fred` 的 `Person` 在两个数据库上具有相同的主键。在你试图保存到 `second` 数据库，如果主键已经在使用，将会引抛出发一个错误。

选择一个数据库用于删除表单

默认情况下，删除一个已存在对象的调用将在与获取对象时使用的相同数据库上执行：

```
>>> u = User.objects.using('legacy_users').get(username='fred')
>>> u.delete() # will delete from the `legacy_users` database
```

要指定删除一个模型时使用的数据库，可以对 `Model.delete()` 方法使用 `using` 关键字参数。这个参数的工作方式与 `save()` 的 `using` 关键字参数一样。

例如，你正在从 `legacy_users` 数据库到 `new_users` 数据库迁移一个 `User` ，你可以使用这些命令：

```
>>> user_obj.save(using='new_users')
>>> user_obj.delete(using='legacy_users')
```

多个数据库上使用管理器

在管理器上使用 `db_manager()` 方法来让管理器访问非默认的数据库。

例如，你有一个自定义的管理器方法，它访问数据库时候用 —— `User.objects.create_user()` 。因为 `create_user()` 是一个管理器方法，不是一个 `QuerySet` 方法，你不可以使用 `User.objects.using('new_users').create_user()` 。（ `create_user()` 方法只能在 `User.objects` 上使用，而不能在从管理器得到的 `QuerySet` 上使用）。解决办法是使用 `db_manager()` ，像这样：

```
User.objects.db_manager('new_users').create_user(...)
```

`db_manager()` 返回一个绑定在你指定的数据上的一个管理器。

多数据库上使用get_queryset()

如果你正在覆盖你的管理器上的 `get_queryset()` ，请确保在其父类上调用方法（使用 `super()` ）或者正确处理管理器上的 `_db` 属性（一个包含将要使用的数据库名称的字符串）。

例如，如果你想从 `get_queryset` 方法返回一个自定义的 `QuerySet` 类，你可以这样做：

```
class MyManager(models.Manager):
    def get_queryset(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Django 的管理站点中使用多数据库

Django 的管理站点没有对多数据库的任何显式的支持。如果你给数据库上某个模型提供的管理站点不想通过你的路由链指定，你将需要编写自定义的 `ModelAdmin` 类用来将管理站点导向一个特殊的数据库。

`ModelAdmin` 对象具有5个方法，它们需要定制以支持多数据库：

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = 'other'

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)

    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' database
        obj.delete(using=self.using)

    def get_queryset(self, request):
        # Tell Django to look for objects on the 'other' database.
        return super(MultiDBModelAdmin, self).get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_foreignkey(db_field, request=

    def formfield_for_manytomany(self, db_field, request=None, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_manytomany(db_field, request=
```

这里提供的实现实现了一个多数据库策略，其中一个给定类型的所有对象都将保存在一个特定的数据库上（例如，所有的 `User` 保存在 `other` 数据库中）。如果你的多数据库的用法更加复杂，你的 `ModelAdmin` 将需要反映相应的策略。

`Inlines` 可以用相似的方式处理。它们需要3个自定义的方法：

```
class MultiDBTabularInline(admin.TabularInline):
    using = 'other'

    def get_queryset(self, request):
        # Tell Django to look for inline objects on the 'other' database.
        return super(MultiDBTabularInline, self).get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_foreignkey(db_field, reque

    def formfield_for_manytomany(self, db_field, request=None, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_manytomany(db_field, reque
```

一旦你写好你的模型管理站点的定义，它们就可以使用任何 `Admin` 实例来注册：

```
from django.contrib import admin

# Specialize the multi-db admin objects for use with specific models.
class BookInline(MultiDBTabularInline):
    model = Book

class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

othersite = admin.AdminSite('othersite')
othersite.register(Publisher, MultiDBModelAdmin)
```

这个例子建立两个管理站点。在第一个站点上， `Author` 和 `Publisher` 对象被暴露出来； `Publisher` 对象具有一个表格的内联，显示该出版社出版的书籍。第二个站点只暴露 `Publishers` ，而没有内联。

多数据库上使用原始游标

如果你正在使用多个数据库，你可以使用 `django.db.connections` 来获取特定数据库的连接（和游标）： `django.db.connections` 是一个类字典对象，它允许你使用别名来获取一个特定的连接：

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
```

多数据库的局限

跨数据库关联

Django 目前不提供跨多个数据库的外键或多对多关系的支持。如果你使用一个路由来路由分离到不同的数据库上，这些模型定义的任何外键和多对多关联必须在单个数据库的内部。

这是因为引用完整性的原因。为了保持两个对象之间的关联，Django 需要知道关联对象的主键是合法的。如果主键存储在另外一个数据库上，判断一个主键的合法性不是很容易。

如果你正在使用Postgres、Oracle或者MySQ 的InnoDB，这是数据库完整性级别的强制要求 —— 数据库级别的主键约束防止创建不能验证合法性的关联。

然而，如果你正在使用SQLite 或MySQL的MyISAM 表，则没有强制性的引用完整性；结果是你可以‘伪造’跨数据库的外键。但是Django 官方不支持这种配置。

Contrib 应用的行为

有几个Contrib 应用包含模型，其中一些应用相互依赖。因为跨数据库的关联是不可能的，这对你如何在数据库之间划分这些模型带来一些限制：

- `contenttypes.ContentType`、`sessions.Session` 和 `sites.Site` 可以存储在分开存储在不同的数据库中，只要给出合适的路由
- `auth` 模型 —— `User`、`Group` 和 `Permission` —— 关联在一起并与 `ContentType` 关联，所以它们必须与 `ContentType` 存储在相同的数据库中。
- `admin` 依赖 `auth`，所以它们的模型必须与 `auth` 在同一个数据库中。
- `flatpages` 和 `redirects` 依赖 `sites`，所以它们必须与 `sites` 在同一个数据库中。

另外，一些对象在migrate在数据库中创建一张表后自动创建：

- 一个默认的 `Site`，
- 为每个模型创建一个 `ContentType`（包括没有存储在同一个数据库中的模型），
- 为每个模型创建3个 `Permission`（包括不是存储在同一个数据库中的模型）。

对于常见的多数据库架构，将这些对象放在多个数据库中没有什么用处。常见的数据库架构包括 `primary/replica` 和连接到外部的数据库。因此，建议写一个数据库路由，它只允许同步这3个模型到一个数据中。对于不需要将表放在多个数据库中的Contrib 应用和第三方应用，可以使用同样的方法。

> 警告
>
> 如果你将 `Content Types` 同步到多个数据库中，注意它们的主键在数据库之间可能不一致。这可能导致数据损坏或数据丢失。

> 译者：Django 文档协作翻译小组，原文：Multiple databases。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

自定义查找

```
New in Django 1.7.
```

Django为过滤提供了大量的内建的查找（例如，`exact` 和 `icontains`）。这篇文档阐述了如何编写自定义查找，以及如何修改现存查找的功能。关于查找的API参考，详见查找API参考。

一个简单的查找示例

让我们从一个简单的自定义查找开始。我们会编写一个自定义查找 `ne`，提供和 `exact` 相反的功能。`Author.objects.filter(name__ne='Jack')` 会转换成下面的SQL：

```
"author"."name" <> 'Jack'
```

这条SQL是后端独立的，所以我们并不需要担心不同的数据库。

实现它需要两个步骤。首先我们需要实现这个查找，然后我们需要告诉Django它的信息。实现是十分简单直接的：

```
from django.db.models import Lookup

class NotEqual(Lookup):
    lookup_name = 'ne'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s <> %s' % (lhs, rhs), params
```

我们只需要在我们想让查找应用的字段上调用 `register_lookup`，来注册 `NotEqual` 查找。这种情况下，查找在所有 `Field` 的子类都起作用，所以我们直接使用 `Field` 注册它。

```
from django.db.models.fields import Field
Field.register_lookup(NotEqual)
```

也可以使用装饰器模式来注册查找：

```
from django.db.models.fields import Field

@Field.register_lookup
class NotEqualLookup(Lookup):
    # ...
```

```
Changed in Django 1.8:

新增了使用装饰器模式的能力。
```

我们现在可以为任何 `foo` 字段使用 `foo__ne` 。你需要确保在你尝试创建使用它的任何查询集之前完成注册。你应该把实现放在 `models.py` 文件中，或者在 `AppConfig` 的 `ready()` 方法中注册查找。

现在让我们深入观察这个实现，首先需要的属性是 `lookup_name` 。这需要让ORM理解如何去解释 `name__ne` ，以及如何使用 `NotEqual` 来生成SQL。按照惯例，这些名字一般是只包含字母的小写字符串，但是唯一硬性的要求是不能够包含字符串 `__` 。

然后我们需要定义 `as_sql` 方法。这个方法需要传入一个 `SQLCompiler` 对象，叫做 `compiler` ，以及活动的数据库连接。 `SQLCompiler` 对象并没有记录，但是我们需要知道的唯一一件事就是他们拥有 `compile()` 方法，这个方法返回一个元组，含有SQL字符串和要向字符串插入的参数。在多数情况下，你并不需要世界使用它，并且可以把它传递给 `process_lhs()` 和 `process_rhs()` 。

`Lookup` 作用于两个值，lhs和rhs，分别是左边和右边。左边的值一般是个字段的引用，但是它可以是任何实现了查询表达式API的对象。右边的值由用户提供。在例子 `Author.objects.filter(name__ne='Jack')` 中，左边的值是 `Author` 模型的 `name` 字段的引用，右边的值是 `'Jack'` 。

我们可以调用 `process_lhs` 和 `process_rhs` 来将它们转换为我们需要的SQL值，使用之前我们描述的 `compiler` 对象。

最后我们用 `<>` 将这些部分组合成SQL表达式，然后将所有参数用在查询中。然后我们返回一个元组，包含生成的SQL字符串以及参数。

一个简单的转换器示例

上面的自定义转换器是极好的，但是一些情况下你可能想要把查找放在一起。例如，假设我们构建一个应用，想要利用 `abs()` 操作符。我们有用一个 `Experiment` 模型，它记录了起始值，终止值，以及变化量（起始值 - 终止值）。我们想要寻找所有变化量等于一个特定值的实验（ `Experiment.objects.filter(change__abs=27)` ），或者没有达到指定值的实验（ `Experiment.objects.filter(change__abs__lt=27)` ）。

> 注意
>
> 这个例子一定程度上很不自然，但是很好地展示了数据库后端独立的功能范围，并且没有重复实现Django中已有的功能。

我们从编写 `AbsoluteValue` 转换器来开始。这会用到SQL函数 `ABS()` ，来在比较之前转换值。

```
from django.db.models import Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "ABS(%s)" % lhs, params
```

接下来，为 `IntegerField` 注册它：

```
from django.db.models import IntegerField
IntegerField.register_lookup(AbsoluteValue)
```

我们现在可以执行之前的查询。`Experiment.objects.filter(change__abs=27)` 会生成下面的SQL：

```
SELECT ... WHERE ABS("experiments"."change") = 27
```

通过使用 `Transform` 来替代 `Lookup`，这说明了我们能够把以后更多的查找放到一起。所以 `Experiment.objects.filter(change__abs__lt=27)` 会生成以下的SQL：

```
SELECT ... WHERE ABS("experiments"."change") < 27
```

注意在没有指定其他查找的情况中，Django会将 `change__abs=27` 解释为 `change__abs__exact=27`。

当寻找在 `Transform` 之后，哪个查找可以使用的时候，Django使用 `output_field` 属性。因为它并没有修改，我们在这里并不指定，但是假设我们在一些字段上应用AbsoluteValue，这些字段代表了一个更复杂的类型（比如说与原点（origin）相关的一个点，或者一个复数（complex number））。之后我们可能想指定，转换要为进一步的查找返回 `FloatField` 类型。这可以通过向转换添加 `output_field` 属性来实现：

```
from django.db.models import FloatField, Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "ABS(%s)" % lhs, params

    @property
    def output_field(self):
        return FloatField()
```

这确保了更进一步的查找，像 `abs__lte` 的行为和对 `FloatField` 表现的一样。

编写高效的 `abs__lt` 查找

当我们使用上面编写的 `abs` 查找的时候，在一些情况下，生成的SQL并不会高效使用索引。尤其是我们使用 `change__abs__lt=27` 的时候，这等价于 `change__gt=-27 AND change__lt=27`。（对于 `lte` 的情况，我们可以使用 SQL子句 `BETWEEN`）。

所以我们想让 `Experiment.objects.filter(change__abs__lt=27)` 生成以下SQL:

```
SELECT .. WHERE "experiments"."change" < 27 AND "experiments"."change" > -27
```

它的实现为：

```
from django.db.models import Lookup

class AbsoluteValueLessThan(Lookup):
    lookup_name = 'lt'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = compiler.compile(self.lhs.lhs)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params + lhs_params + rhs_params
        return '%s < %s AND %s > -%s' % (lhs, rhs, lhs, rhs), params

AbsoluteValue.register_lookup(AbsoluteValueLessThan)
```

有一些值得注意的事情。首先， `AbsoluteValueLessThan` 并不调用 `process_lhs()`。而是它跳过了由 `AbsoluteValue` 完成的 `lhs`，并且使用原始的 `lhs`。这就是说，我们想要得到 `27` 而不是 `ABS(27)`。直接引用 `self.lhs.lhs` 是安全的，因为 `AbsoluteValueLessThan` 只能够通过 `AbsoluteValue` 查找来访问，这就是说 `lhs` 始终是 `AbsoluteValue` 的实例。

也要注意，就像两边都要在查询中使用多次一样，参数也需要多次包含 `lhs_params` 和 `rhs_params`。

最终的实现直接在数据库中执行了反转 (27变为 -27)。这样做的原因是如果 `self.rhs` 不是一个普通的整数值（比如是一个 `F()` 引用），我们在Python中不能执行这一转换。

> 注意
>
> 实际上，大多数带有__abs的查找都实现为这种范围查询，并且在大多数数据库后端中它更可能执行成这样，就像你可以利用索引一样。然而在PostgreSQL中，你可能想要向 abs(change) 中添加索引，这会使查询更高效。

一个双向转换器的示例

我们之前讨论的， `AbsoluteValue` 的例子是一个只应用在查找左侧的转换。可能有一些情况，你想要把转换同时应用在左侧和右侧。比如，你想过滤一个基于左右侧相等比较操作的查询集，在执行一些SQL函数之后它们是大小写不敏感的。

让我们测试一下这一大小写不敏感的转换的简单示例。这个转换在实践中并不是十分有用，因为Django已经自带了一些自建的大小写不敏感的查找，但是它是一个很好的，数据库无关的双向转换示例。

我们定义使用SQL 函数 `UPPER()` 的 `UpperCase` 转换器，来在比较前转换这些值。我们定义了 `bilateral = True` 来表明转换同时作用在 `lhs` 和 `rhs` 上面：

```
from django.db.models import Transform

class UpperCase(Transform):
    lookup_name = 'upper'
    bilateral = True

    def as_sql(self, compiler, connection):
        lhs, params = compiler.compile(self.lhs)
        return "UPPER(%s)" % lhs, params
```

接下来，让我们注册它：

```
from django.db.models import CharField, TextField
CharField.register_lookup(UpperCase)
TextField.register_lookup(UpperCase)
```

现在，查询集 `Author.objects.filter(name__upper="doe")` 会生成像这样的大小写不敏感查询：

```
SELECT ... WHERE UPPER("author"."name") = UPPER('doe')
```

为现存查找编写自动的实现

有时不同的数据库供应商对于相同的操作需要不同的SQL。对于这个例子，我们会为MySQL重新编写一个自定义的， `NotEqual` 操作的实现。我们会使用 `!=` 而不是 `<>` 操作符。（注意实际上几乎所有数据库都支持这两个，包括所有Django支持的官方数据库）。

我们可以通过创建带有 `as_mysql` 方法的 `NotEqual` 的子类来修改特定后端上的行为。

```
class MySQLNotEqual(NotEqual):
    def as_mysql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s != %s' % (lhs, rhs), params

Field.register_lookup(MySQLNotEqual)
```

我们可以在 `Field` 中注册它。它取代了原始的 `NotEqual` 类，由于它具有相同的 `lookup_name` 。

当编译一个查询的时候，Django首先寻找 `as_%s % connection.vendor` 方法，然后回退到 `as_sql` 。内建后端的供应商名称是 sqlite，postgresql， oracle 和mysql。

Django如何决定使用查找还是转换

有些情况下，你可能想要动态修改基于传递进来的名称， `Transform` 或者 `Lookup` 哪个会返回，而不是固定它。比如，你拥有可以储存搭配（ coordinate）或者任意一个维度（dimension）的字段，并且想让类似于 `.filter(coords__x7=4)` 的语法返回第七个搭配值为4的对象。为了这样做，你可以用一些东西覆写 `get_lookup` ，比如：

```
class CoordinatesField(Field):
    def get_lookup(self, lookup_name):
        if lookup_name.startswith('x'):
            try:
                dimension = int(lookup_name[1:])
            except ValueError:
                pass
            finally:
                return get_coordinate_lookup(dimension)
        return super(CoordinatesField, self).get_lookup(lookup_name)
```

之后你应该合理定义 `get_coordinate_lookup` 。来返回一个 `Lookup` 的子类，它处理 `dimension` 的相关值。

有一个名称相似的方法叫做 `get_transform()` 。 `get_lookup()` 应该始终返回 `Lookup` 的子类，而 `get_transform()` 返回 `Transform` 的子类。记住 `Transform` 对象可以进一步过滤，而 `Lookup` 对象不可以，这非常重要。

过滤的时候，如果还剩下只有一个查找名称要处理，它会寻找 `Lookup` 。如果有多个名称，它会寻找 `Transform` 。在只有一个名称并且 Lookup找不到的情况下，会寻找 `Transform` ，之后寻找在 `Transform` 上面的 `exact` 查找。所有调用的语句都以一个 `Lookup` 结尾。解释一下：

- `.filter(myfield__mylookup)` 会调用 `myfield.get_lookup('mylookup')` 。
- `.filter(myfield__mytransform__mylookup)` 会调用 `myfield.get_transform('mytransform')` ，然后调用 `mytransform.get_lookup('mylookup')` 。
- `.filter(myfield__mytransform)` 会首先调用 `myfield.get_lookup('mytransform')` ，这样会失败，所以它会回退来调用 `myfield.get_transform('mytransform')` ，之后是 `mytransform.get_lookup('exact')` 。

> 译者：Django 文档协作翻译小组，原文：Custom lookups。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

条件表达式

```
New in Django 1.8.
```

条件表达式允许你在过滤器、注解、聚合和更新操作中使用 `if ... elif ... else` 的逻辑。条件表达式为表中的每一行计算一系列的条件，并且返回匹配到的结果表达式。条件表达式也可以像其它 表达式一样混合和嵌套。

条件表达式类

我们会在后面的例子中使用下面的模型：

```
from django.db import models

class Client(models.Model):
    REGULAR = 'R'
    GOLD = 'G'
    PLATINUM = 'P'
    ACCOUNT_TYPE_CHOICES = (
        (REGULAR, 'Regular'),
        (GOLD, 'Gold'),
        (PLATINUM, 'Platinum'),
    )
    name = models.CharField(max_length=50)
    registered_on = models.DateField()
    account_type = models.CharField(
        max_length=1,
        choices=ACCOUNT_TYPE_CHOICES,
        default=REGULAR,
    )
```

When

`class When(condition=None, then=None, **lookups)[source]`

`When()` 对象用于封装条件和它的结果，为了在条件表达式中使用。使用 `When()` 对象和使用 `filter()` 方法类似。条件可以使用字段查找 或者 `Q` 来指定。结果通过使用 `then` 关键字来提供。

一些例子：

```
>>> from django.db.models import When, F, Q
>>> # String arguments refer to fields; the following two examples are equivalent:
>>> When(account_type=Client.GOLD, then='name')
>>> When(account_type=Client.GOLD, then=F('name'))
>>> # You can use field lookups in the condition
>>> from datetime import date
>>> When(registered_on__gt=date(2014, 1, 1),
...      registered_on__lt=date(2015, 1, 1),
...      then='account_type')
>>> # Complex conditions can be created using Q objects
>>> When(Q(name__startswith="John") | Q(name__startswith="Paul"),
...      then='name')
```

要注意这些值中的每一个都可以是表达式。

> 注意
>
> 由于then 关键字参数为 When()的结果而保留，如果Model有名称为 then的字段，会有潜在的冲突。这可以用以下两种办法解决：

```
>>> from django.db.models import Value
>>> When(then__exact=0, then=1)
>>> When(Q(then=0), then=1)
```

Case

`class Case(*cases, **extra)[source]`

`Case()` 表达式就像是Python中的 `if ... elif ... else` 语句。每个提供的 `When()` 中的 `condition` 按照顺序计算，直到得到一个真值。返回匹配 `When()` 对象的 `result` 表达式。

一个简单的例子：

```
>>>
>>> from datetime import date, timedelta
>>> from django.db.models import CharField, Case, Value, When
>>> Client.objects.create(
...     name='Jane Doe',
...     account_type=Client.REGULAR,
...     registered_on=date.today() - timedelta(days=36))
>>> Client.objects.create(
...     name='James Smith',
...     account_type=Client.GOLD,
...     registered_on=date.today() - timedelta(days=5))
>>> Client.objects.create(
...     name='Jack Black',
...     account_type=Client.PLATINUM,
...     registered_on=date.today() - timedelta(days=10 * 365))
>>> # Get the discount for each Client based on the account type
>>> Client.objects.annotate(
...     discount=Case(
...         When(account_type=Client.GOLD, then=Value('5%')),
...         When(account_type=Client.PLATINUM, then=Value('10%')),
...         default=Value('0%'),
...         output_field=CharField(),
...     ),
... ).values_list('name', 'discount')
[('Jane Doe', '0%'), ('James Smith', '5%'), ('Jack Black', '10%')]
```

`Case()` 接受任意数量的 `When()` 对象作为独立的参数。其它选项使用关键字参数提供。如果没有条件为 `TRUE`，表达式会返回提供的 `default` 关键字参数。如果没有提供 `default` 参数，会使用 `Value(None)`。

如果我们想要修改之前的查询，来获取基于 `Client` 跟着我们多长时间的折扣，我们应该这样使用查找：

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Get the discount for each Client based on the registration date
>>> Client.objects.annotate(
...     discount=Case(
...         When(registered_on__lte=a_year_ago, then=Value('10%')),
...         When(registered_on__lte=a_month_ago, then=Value('5%')),
...         default=Value('0%'),
...         output_field=CharField(),
...     )
... ).values_list('name', 'discount')
[('Jane Doe', '5%'), ('James Smith', '0%'), ('Jack Black', '10%')]
```

> 注意
>
> 记住条件按照顺序来计算，所以上面的例子中，即使第二个条件匹配到了 Jane Doe 和 Jack Black，我们也得到了正确的结果。这就像Python中的if ... elif ... else语句一样。

高级查询

条件表达式可以用于注解、聚合、查找和更新。它们也可以和其它表达式混合和嵌套。这可以让你构造更强大的条件查询。

条件更新

假设我们想要为客户端修改 `account_type` 来匹配它们的注册日期。我们可以使用条件表达式和 `update()` 放啊来实现：

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Update the account_type for each Client from the registration date
>>> Client.objects.update(
...     account_type=Case(
...         When(registered_on__lte=a_year_ago,
...              then=Value(Client.PLATINUM)),
...         When(registered_on__lte=a_month_ago,
...              then=Value(Client.GOLD)),
...         default=Value(Client.REGULAR)
...     ),
... )
>>> Client.objects.values_list('name', 'account_type')
[('Jane Doe', 'G'), ('James Smith', 'R'), ('Jack Black', 'P')]
```

条件聚合

如果我们想要弄清楚每个 `account_type` 有多少客户端，要怎么做呢？我们可以在聚合函数中嵌套条件表达式来实现：

```
>>> # Create some more Clients first so we can have something to count
>>> Client.objects.create(
...     name='Jean Grey',
...     account_type=Client.REGULAR,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='James Bond',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='Jane Porter',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> # Get counts for each value of account_type
>>> from django.db.models import IntegerField, Sum
>>> Client.objects.aggregate(
...     regular=Sum(
...         Case(When(account_type=Client.REGULAR, then=1),
...              output_field=IntegerField())
...     ),
...     gold=Sum(
...         Case(When(account_type=Client.GOLD, then=1),
...              output_field=IntegerField())
...     ),
...     platinum=Sum(
...         Case(When(account_type=Client.PLATINUM, then=1),
...              output_field=IntegerField())
...     )
... )
{'regular': 2, 'gold': 1, 'platinum': 3}
```

> 译者：Django 文档协作翻译小组，原文：Conditional Expressions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

数据库函数

```
New in Django 1.8.
```

下面记述的类为用户提供了一些方法，来在Django中使用底层数据库提供的函数用于注解、聚合或者过滤器等操作。函数也是表达式，所以可以像聚合函数一样混合使用它们。

我们会在每个函数的实例中使用下面的模型：

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    age = models.PositiveIntegerField(null=True, blank=True)
    alias = models.CharField(max_length=50, null=True, blank=True)
    goes_by = models.CharField(max_length=50, null=True, blank=True)
```

我们并不推荐在CharField上允许null=True，以后那位这会允许字段有两个“空值”，但是对于下面的Coalesce示例来说它很重要。

Coalesce

`class Coalesce(*expressions, **extra)[source]`

接受一个含有至少两个字段名称或表达式的列表，返回第一个非空的值（注意空字符串不被认为是一个空值）。每个参与都必须是相似的类型，所以掺杂了文本和数字的列表会导致数据库错误。

使用范例：

```
>>> # Get a screen name from least to most public
>>> from django.db.models import Sum, Value as V
>>> from django.db.models.functions import Coalesce
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie')
>>> author = Author.objects.annotate(
...     screen_name=Coalesce('alias', 'goes_by', 'name')).get()
>>> print(author.screen_name)
Maggie

>>> # Prevent an aggregate Sum() from returning None
>>> aggregated = Author.objects.aggregate(
...     combined_age=Coalesce(Sum('age'), V(0)),
...     combined_age_default=Sum('age'))
>>> print(aggregated['combined_age'])
0
>>> print(aggregated['combined_age_default'])
None
```

Concat

`class Concat(*expressions, **extra)[source]`

接受一个含有至少两个文本字段的或表达式的列表，返回连接后的文本。每个参数都必须是文本或者字符类型。如果你想把一个 `TextField()` 和一个 `CharField()` 连接， 一定要告诉 Django `output_field` 应该为 `TextField()` 类型。在下面连接 `Value` 的例子中，这也是必需的。

这个函数不会返回 `null` 。在后端中，如果一个 `null` 参数导致了整个表达式都是 `null` ，Django会确保把每个 `null` 的部分转换成一个空字符串。

使用范例：

```
>>> # Get the display name as "name (goes_by)"
>>> from django.db.models import CharField, Value as V
>>> from django.db.models.functions import Concat
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie')
>>> author = Author.objects.annotate(
...     screen_name=Concat('name', V(' ('), 'goes_by', V(')'),
...     output_field=CharField())).get()
>>> print(author.screen_name)
Margaret Smith (Maggie)
```

Length

`class Length(expression, **extra)[source]`

接受一个文本字段或表达式，返回值的字符个数。如果表达式是 `null` ，长度也会是 `null` 。

使用范例：

```
>>> # Get the length of the name and goes_by fields
>>> from django.db.models.functions import Length
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(
...     name_length=Length('name'),
...     goes_by_length=Length('goes_by')).get()
>>> print(author.name_length, author.goes_by_length)
(14, None)
```

Lower

`class Lower(expression, **extra)[source]`

接受一个文本字符串或表达式，返回它的小写表示形式。

使用范例：

```
>>> from django.db.models.functions import Lower
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_lower=Lower('name')).get()
>>> print(author.name_lower)
margaret smith
```

Substr

`class Substr(expression, pos, length=None, **extra)[source]`

返回这个字段或者表达式的，以 `pos` 位置开始，长度为 `length` 的子字符串。位置从下标为1开始，所以必须大于0。如果 `length` 是 `None`，会返回剩余的字符串。

使用范例：

```
>>> # Set the alias to the first 5 characters of the name as lowercase
>>> from django.db.models.functions import Substr, Lower
>>> Author.objects.create(name='Margaret Smith')
>>> Author.objects.update(alias=Lower(Substr('name', 1, 5)))
1
>>> print(Author.objects.get(name='Margaret Smith').alias)
marga
```

Upper

`class Upper(expression, **extra)[source]`

接受一个文本字符串或表达式，返回它的大写表示形式。

使用范例：

```
>>> from django.db.models.functions import Upper
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_upper=Upper('name')).get()
>>> print(author.name_upper)
MARGARET SMITH
```

其它

将遗留数据库整合到Django

虽然Django最适合用来开发新的应用，但也可以将它整合到遗留的数据库中。Django包含了很多工具，尽可能自动化解决这类问题。

这篇文章假设你了解Django的基础部分，它们在教程中提及。

一旦你的Django环境建立好之后，你可以按照这个大致的流程，整合你的现有数据库。

向Django提供你的数据库参数

你需要告诉Django你的数据库连接参数，以及数据库的名称。请修改DATABASES设置，为'默认' 连接的以下键赋值：

- NAME
- ENGINE
- USER
- PASSWORD
- HOST
- PORT

自动生成模型

Django自带叫做inspectdb的工具，可以按照现有的数据库创建模型。你可以运行以下命令，并查看输出：

```
$ python manage.py inspectdb
```

通过重定向Unix标准输出流来保存文件：

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

这个特性是一个快捷方式，并不是一个确定的模型生成器。详见inspectdb文档 。

一旦你创建好了你的模型，把文件命名为models.py，然后把它放到你应用的Python包中。然后把应用添加到你的INSTALLED_APPS 设置中。

默认情况下，inspectdb创建未被管理的模型。这就是说，模型的Meta类中的managed = False告诉Django不要管理每个表的创建、修改和删除：

```
class Person(models.Model):
    id = models.IntegerField(primary_key=True)
    first_name = models.CharField(max_length=70)
    class Meta:
       managed = False
       db_table = 'CENSUS_PERSONS'
```

如果你希望Django管理表的生命周期，你需要把managed选项改为 True（或者简单地把它移除，因为True是默认值）。

安装Django核心表

接下来，运行migrate命令来安装所有所需的额外的数据库记录，比如后台权限和内容类型：

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

测试和调整

上面就是所有基本的步骤了 —— 到目前为止你会想要调整Django自动生成的模型，直到他们按照你想要的方式工作。尝试通过Django数据库API访问你的数据，并且尝试使用Django后台页面编辑对象，以及相应地编辑模型文件。

为模型提供初始数据

当你首次建立一个应用的时候，为你的数据库预先安装一些硬编码的数据，是很有用处的。有几种方法可以让Django自动创建这些数据：你可以通过fixtures提供初始数据，或者提供一个包含初始数据的sql文件。

通常来讲，使用fixtrue更加简洁，因为它是数据库无关的，而使用sql初始化更加灵活。

提供初始数据的**fixtures**

fixture是数据的集合，让Django了解如何导入到数据库中。创建fixture的最直接的方式，是使用manage.py dumpdata命令，如果数据库中已经有了一些数据。或者你可以手写fixtures。fixtures支持JSON、XML或者YAML（需要安装PyYAML）文档。序列化文档中详细阐述了每一种所支持的序列化格式。

下面这个例子展示了一个简单的Person 模型的fixtrue，看起来很像JSON：

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

下面是它的YAML格式：

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

你可以把这些数据储存在你应用的fixtures目录中。

加载数据很简单：只要调用manage.py loaddata <fixturename>就好了，其中<fixturename>是你所创建的fixture文件的名字。每次你运行loaddata的时候，数据都会从fixture读出，并且重复加载进数据库。注意这意味着，如果你修改了fixtrue创建的某一行，然后再次运行了loaddata，你的修改将会被抹掉。

自动加载初始数据的**fixtures**

```
1.7中废除：

如果一个应用使用了迁移，将不会自动加载fixtures。由于Django 1.9中，迁移将会是必要的，这一行为经权衡之后被
```

如果你创建了一个命名为 initial_data.[xml/yaml/json]的fixtrue，在你每次运行migrate命令时，fixtrue都会被加载。这非常方面，但是要注意：记住数据在你每次运行migrate命令后都会被刷新。So don't use initial_data for data you'll want to edit.

Django在哪里寻找**fixture**文件

通常，Django 在每个应用的fixtures目录中寻找fixture文件。你可以设置FIXTURE_DIRS选项为一个额外目录的列表，Django会从里面寻找。

运行manage.py loaddata命令的时候，你也可以指定一个fixture文件的目录，它会覆盖默认设置中的目录。

> 另见
>
> fixtrues也被用于测试框架来搭建一致性的测试环境。

提供初始**SQL**数据

```
1.7中废除：

如果一个应用使用迁移，初始SQL数据将不会加载（包括后端特定的SQL数据）。由于Django 1.9中，迁移将会是必须的
```

Django为数据库无关的SQL提供了一个钩子，当你运行migrate命令时，CREATE TABLE语句执行之后就会执行它。你可以使用这个钩子来建立默认的记录，或者创建SQL函数、视图、触发器以及其它。

钩子十分简单：Django会在你应用的目录中寻找叫做sql/<modelname>.sql的文件，其中<modelname>是小写的模型名称。

所以如果在myapp应用中存在Person模型，你应该在myapp目录的文件sql/person.sql中添加数据库无关的SQL。下面的例子展示了文件可能会包含什么：

```
INSERT INTO myapp_person (first_name, last_name) VALUES ('John', 'Lennon');
INSERT INTO myapp_person (first_name, last_name) VALUES ('Paul', 'McCartney');
```

每个提供的SQL文件，都应该含有用于插入数据的有效的SQL语句（例如，格式适当的INSERT语句，用分号分隔）。

这些SQL文件可被manage.py中的 sqlcustom和sqlall命令阅读。详见manage.py文档。

注意如果你有很多SQL数据文件，他们执行的顺序是不确定的。唯一可以确定的是，在你的自定义数据文件被执行之前，所有数据表都被创建好了。

> 初始SQL数据和测试
>
> 这一技巧不能以测试目的用于提供初始数据。Django的测试框架在每次测试后都会刷新测试数据库的内容。所以，任何使用自定义SQL钩子添加的数据都会丢失。
>
> 如果你需要在测试用例中添加数据，你应该在测试fixture中添加它，或者在测试用例的setUp()中添加。

数据库后端特定的SQL数据

没有钩子提供给后端特定的SQL数据。例如，你有分别为PostgreSQL和SQLite准备的初始数据文件。对于每个应用，Django都会寻找叫做<app_label>/sql/<modelname>.<backend>.sql的文件，其中<app_label>是小写的模型名称，<modelname>是小写的模型名称，<backend>是你的设置文件中由ENGINE提供的模块名称的最后一部分（例如，如果你定义了一个数据库，ENGINE的值为django.db.backends.sqlite3，Django会寻找<app_label>/sql/<modelname>.sqlite3.sql）。

后端特定的SQL数据会先于后端无关的SQL数据执行。例如，如果你的应用包含了sql/person.sql 和sql/person.sqlite3.sql文件，而且你已经安装了SQLite应用，Django会首先执行 sql/person.sqlite3.sql的内容，其次才是sql/person.sql。

数据库访问优化

Django的数据库层提供了很多方法来帮助开发者充分的利用他们的数据库。这篇文档收集了相关文档的一些链接，添加了大量提示，并且按照优化数据库使用的步骤的概要来组织。

性能优先

作为通用的编程实践，性能的重要性不用多说。弄清楚你在执行什么查询以及你的开销花在哪里。你也可能想使用外部的项目，像django-debug-toolbar，或者直接监控数据库的工具。

记住你可以优化速度、内存占用，甚至二者一起，这取决于你的需求。一些针对其中一个的优化会对另一个不利，但有时会对二者都有帮助。另外，数据库进程做的工作，可能和你在Python代码中做的相同工作不具有相同的开销。决定你的优先级是什么，是你自己的事情，你必须要权衡利弊，按需使用它们，因为这取决于你的应用和服务器。

对于下面提到的任何事情，要记住在任何修改后验证一下，确保修改是有利的，并且足够有利，能超过你代码中可读性的下降。下面的所有建议都带有警告，在你的环境中大体原则可能并不适用，或者会起到相反的效果。

使用标准数据库优化技巧

...包括：

- 索引。在你决定哪些索引应该添加 之后，这一条具有最高优先级。使用Field.db_index或者Meta.index_together在Dhango中添加它们。考虑在你经常使用filter()、exclude()、order_by()和其它方法查询的字段上面添加索引，因为索引有助于加速查找。注意，设计最好的索引方案是一个复杂的、数据库相关的话题，它取决于你应用的细节。持有索引的副作用可能会超过查询速度上的任何收益。
- 合理使用字段类型。

我们假设你已经完成了上面这些显而易见的事情。这篇文档剩下的部分，着重于讲解如何以不做无用功的方式使用Django。这篇文档也没有强调用在开销大的操作上其它的优化技巧，像general purpose caching。

理解查询集

理解查询集(QuerySets) 是通过简单的代码获取较好性能至关重要的一步。特别是：

理解查询集计算

要避免性能问题，理解以下几点非常重要：

- QuerySets是延迟的。
- 什么时候它们被计算出来。
- 数据在内存中如何存储。

理解缓存属性

和整个QuerySet的缓存相同，ORM对象的属性的结果中也存在缓存。通常来说，不可调用的属性会被缓存。例如下面的博客模型示例：

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog   # Blog object is retrieved at this point
>>> entry.blog   # cached version, no DB access
```

但是通常来讲，可调用的属性每一次都会访问数据库。

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()   # query performed
>>> entry.authors.all()   # query performed again
```

要小心当你阅读模板代码的时候 —— 模板系统不允许使用圆括号，但是会自动调用callable对象，会隐藏上述区别。

要小心使用你自定义的属性 —— 实现所需的缓存取决于你，例如使用cached_property装饰符。

使用with模板标签

要利用QuerySet的缓存行为，你或许需要使用with模板标签。

使用iterator()

当你有很多对象时，QuerySet的缓存行为会占用大量的内存。这种情况下，采用iterator()解决。

在数据库中而不是Python中做数据库的工作

比如：

- 在最基础的层面上，使用过滤器和反向过滤器对数据库进行过滤。
- 使用F 表达式在相同模型中基于其他字段进行过滤。
- 使用数据库中的注解和聚合。

如果上面那些都不够用，你可以自己生成SQL语句：

使用QuerySet.extra()

extra()是一个移植性更差，但是功能更强的方法，它允许一些SQL语句显式添加到查询中。如果这些还不够强大：

使用原始的SQL

编写你自己的自定义SQL语句，来获取数据或者填充模型。使用 django.db.connection.queries来了解Django为你编写了什么，以及从这里开始。

用唯一的被或索引的列来检索独立对象

有两个原因在get()中，用带有unique或者db_index的列检索独立对象。首先，由于查询经过了数据库的索引，所以会更快。其次，如果很多对象匹配查询，查询会更慢一些；列上的唯一性约束确保这种情况永远不会发生。

所以，使用博客模型的例子：

```
>>> entry = Entry.objects.get(id=10)
```

会快于：

```
>>> entry = Entry.object.get(headline="News Item Title")
```

因为id被数据库索引，而且是唯一的。

下面这样做会十分缓慢：

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

首先， headline没有被索引，它会使查询变得很慢：

其次，这次查找并不确保返回唯一的对象。如果查询匹配到多于一个对象，它会在数据库中遍历和检索所有这些对象。如果记录中返回了成百上千个对象，代价是非常大的。如果数据库运行在分布式服务器上，网络开销和延迟也是一大因素，代价会是它们的组合。

一次性检索你需要的任何东西

在不同的位置多次访问数据库，一次获取一个数据集，通常来说不如在一次查询中获取它们更高效。如果你在一个循环中执行查询，这尤其重要。有可能你会做很多次数据库查询，但只需要一次就够了。所以：

使用QuerySet.select_related()和prefetch_related()

充分了解并使用select_related()和prefetch_related()：

- 在视图的代码中，
- 以及在适当的管理器和默认管理器中。要意识到你的管理器什么时候被使用和不被使用；有时这很复杂，所以不要有任何假设。

不要获取你不需要的东西

使用QuerySet.values()和values_list()

当你仅仅想要一个带有值的字典或者列表，并不需要使用ORM模型对象时，可以适当使用values()。对于在模板代码中替换模型对象，这样会非常有用 —— 只要字典中带有的属性和模板中使用的一致，就没问题。

使用QuerySet.defer()和only()

如果一些数据库的列你并不需要（或者大多数情况下并不需要），使用defer()和only()来避免加载它们。注意如果你确实要用到它们，ORM会在另外的查询之中获取它们。如果你不能够合理地使用这些函数，不如不用。

另外，当建立起一个带有延迟字段的模型时，要意识到一些（小的、额外的）消耗会在Django内部产生。不要不分析数据库就盲目使用延迟字段，因为数据库必须从磁盘中读取大多数非text和VARCHAR数据，在结果中作为单独的一行，即使其中的列很少。 defer()和only()方法在你可以避免加载大量文本数据，或者可能要花大量时间处理而返回给Python的字段时，特别有帮助。像往常一样，应该先写出个大概，之后再优化。

使用QuerySet.count()

...如果你想要获取大小，不要使用 len(queryset)。

使用QuerySet.exists()

...如果你想要知道是否存在至少一个结果，不要使用if queryset。

但是：

不要过度使用 count() 和 exists()

如果你需要查询集中的其他数据，就把它加载出来。

例如，假设Email模型有一个body属性，并且和User有多对多的关联，下面的的模板代码是最优的：

```
{% if display_inbox %}
  {% with emails=user.emails.all %}
    {% if emails %}
      <p>You have {{ emails|length }} email(s)</p>
      {% for email in emails %}
        <p>{{ email.body }}</p>
      {% endfor %}
    {% else %}
      <p>No messages today.</p>
    {% endif %}
  {% endwith %}
{% endif %}
```

这是因为：

- 因为查询集是延迟加载的，如果'display_inbox'为False，不会查询数据库。
- 使用with意味着我们为了以后的使用，把user.emails.all储存在一个变量中，允许它的缓存被复用。
- {% if emails %}的那一行调用了QuerySet.**bool**()，它导致user.emails.all()查询在数据库上执行，并且至少在第一行以一个ORM对象的形式返回。如果没有任何结果，会返回False，反之为True。
- {{ emails|length }}调用了QuerySet.**len**()方法，填充了缓存的剩余部分，而且并没有执行另一次查询。
- for循环的迭代器访问了已经缓存的数据。

总之，这段代码做了零或一次查询。唯一一个慎重的优化就是with标签的使用。在任何位置使用QuerySet.exists()或者QuerySet.count()都会导致额外的查询。

使用QuerySet.update()和delete()

通过QuerySet.update()使用批量的SQL UPDATE语句，而不是获取大量对象，设置一些值再单独保存。与此相似，在可能的地方使用批量deletes。

但是要注意，这些批量的更新方法不会在单独的实例上面调用save()或者delete()方法，意思是任何你向这些方法添加的自定义行为都不会被执行，包括由普通数据库对象的信号驱动的任何方法。

直接使用外键的值

如果你仅仅需要外键当中的一个值，要使用对象上你已经取得的外键的值，而不是获取整个关联对象再得到它的主键。例如，执行：

```
entry.blog_id
```

而不是：

```
entry.blog.id
```

不要做无谓的排序

排序并不是没有代价的；每个需要排序的字段都是数据库必须执行的操作。如果一个模型具有默认的顺序（Meta.ordering），并且你并不需要它，通过在查询集上无参调用order_by()来移除它。

向你的数据库添加索引可能有助于提升排序性能。

整体插入

创建对象时，尽可能使用bulk_create()来减少SQL查询的数量。例如：

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...更优于：

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

注意该方法有很多注意事项，所以确保它适用于你的情况。

这也可以用在ManyToManyFields中，所以：

```
my_band.members.add(me, my_friend)
```

...更优于：

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...其中Bands和Artists具有多对多关联。

视图层

Django 具有“视图”的概览，用于封装负责处理用户请求及返回响应的逻辑。通过下面的链接可以找到你需要知道的所有关于视图的内容：

基础

URL调度器

简洁、优雅的URL 模式在高质量的Web 应用中是一个非常重要的细节。Django 允许你任意设计你的URL，不受框架束缚。

不要求有 `.php` 或 `.cgi`，更不会要求类似 `0,2097,1-1-1928,00` 这样无意义的东西。

参见万维网的发明者Berners-Lee 的Cool URIs don't change，里面有关于为什么URL 应该保持整洁和有意义的卓越的论证。

概览

为了给一个应用设计URL，你需要创建一个Python 模块，通常称为URLconf（URL configuration）。这个模块是纯粹的Python 代码，包含URL 模式（简单的正则表达式）到Python 函数（你的视图）的简单映射。

映射可短可长，随便你。它可以引用其它的映射。而且，因为它是纯粹的Python 代码，它可以动态构造。

Django 还提供根据当前语言翻译URL 的一种方法。更多信息参见国际化文档。

Django 如何处理一个请求

当一个用户请求Django 站点的一个页面，下面是Django 系统决定执行哪个Python 代码使用的算法：

1. Django 决定要使用的根 `URLconf` 模块。通常，这个值就是 `ROOT_URLCONF` 的设置，但是如果进来的 `HttpRequest` 对象具有一个 `urlconf` 属性（通过中间件 `request processing` 设置），则使用这个值来替换 `ROOT_URLCONF` 设置。
2. Django 加载该Python 模块并寻找可用的 `urlpatterns`。它是 `django.conf.urls.url()` 实例的一个Python 列表。
3. Django 依次匹配每个URL 模式，在与请求的URL 匹配的第一个模式停下来。
4. 一旦其中的一个正则表达式匹配上，Django 将导入并调用给出的视图，它是一个简单的Python 函数（或者一个基于类的视图）。视图将获得如下参数:
   - 一个 `HttpRequest` 实例。
   - 如果匹配的正则表达式没有返回命名的组，那么正则表达式匹配的内容将作为位置参数提供给视图。
   - 关键字参数由正则表达式匹配的命名组组成，但是可以被 `django.conf.urls.url()` 的可选参数 `kwargs` 覆盖。
5. 如果没有匹配到正则表达式，或者如果过程中抛出一个异常，Django 将调用一个适当的

错误处理视图。请参见下面的错误处理。

例子

下面是一个简单的 URLconf：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

注：

- 若要从URL 中捕获一个值，只需要在它周围放置一对圆括号。
- 不需要添加一个前导的反斜杠，因为每个URL 都有。例如，应该是 `^articles` 而不是 `^/articles`。
- 每个正则表达式前面的'r' 是可选的但是建议加上。它告诉Python 这个字符串是“原始的” —— 字符串中任何字符都不应该转义。参见Dive Into Python 中的解释。

一些请求的例子：

- `/articles/2005/03/` 请求将匹配列表中的第三个模式。Django 将调用函数 `views.month_archive(request, '2005', '03')`。
- `/articles/2005/3/` 不匹配任何URL 模式，因为列表中的第三个模式要求月份应该是两个数字。
- `/articles/2003/` 将匹配列表中的第一个模式不是第二个，因为模式按顺序匹配，第一个会首先测试是否匹配。请像这样自由插入一些特殊的情况来探测匹配的次序。
- `/articles/2003` 不匹配任何一个模式，因为每个模式要求URL 以一个反斜线结尾。
- `/articles/2003/03/03/` 将匹配最后一个模式。Django 将调用函数 `views.article_detail(request, '2003', '03', '03')`。

命名组

上面的示例使用简单的、没有命名的正则表达式组（通过圆括号）来捕获URL 中的值并以位置 参数传递给视图。在更高级的用法中，可以使用命名的正则表达式组来捕获URL 中的值并以关键字 参数传递给视图。

在Python 正则表达式中，命名正则表达式组的语法是 `(?P<name>pattern)`，其中 `name` 是组的名称，`pattern` 是要匹配的模式。

下面是以上URLconf 使用命名组的重写：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$', views.ar
]
```

这个实现与前面的示例完全相同，只有一个细微的差别：捕获的值作为关键字参数而不是位置参数传递给视图函数。例如：

- `/articles/2005/03/` 请求将调用 `views.month_archive(request, year='2005', month='03')` 函数，而不是 `views.month_archive(request, '2005', '03')`。
- `/articles/2003/03/03/` 请求将调用函数 `views.article_detail(request, year='2003', month='03', day='03')`。

在实际应用中，这意味你的URLconf 会更加明晰且不容易产生参数顺序问题的错误 —— 你可以在你的视图函数定义中重新安排参数的顺序。当然，这些好处是以简洁为代价；有些开发人员认为命名组语法丑陋而繁琐。

匹配/分组算法

下面是URLconf 解析器使用的算法，针对正则表达式中的命名组和非命名组：

1. 如果有命名参数，则使用这些命名参数，忽略非命名参数。
2. 否则，它将以位置参数传递所有的非命名参数。

根据传递额外的选项给视图函数（下文），这两种情况下，多余的关键字参数也将传递给视图。

URLconf 在什么上查找

URLconf 在请求的URL 上查找，将它当做一个普通的Python 字符串。不包括GET和POST参数以及域名。

例如，http://www.example.com/myapp/ 请求中，URLconf 将查找 `myapp/`。

在 http://www.example.com/myapp/?page=3 请求中，URLconf 仍将查找 `myapp/`。

URLconf 不检查请求的方法。换句话讲，所有的请求方法 —— 同一个URL 的 `POST`、`GET`、`HEAD` 等等 —— 都将路由到相同的函数。

捕获的参数永远是字符串

每个捕获的参数都作为一个普通的Python 字符串传递给视图，无论正则表达式使用的是什么匹配方式。例如，下面这行URLconf 中：

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
```

... `views.year_archive()` 的 `year` 参数将是一个字符串，即使 `[0-9]{4}` 值匹配整数字符串。

指定视图参数的默认值

有一个方便的小技巧是指定视图参数的默认值。 下面是一个URLconf 和视图的示例：

```
# URLconf
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>[0-9]+)/$', views.page),
]

# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    ...
```

在上面的例子中，两个URL模式指向同一个视图 `views.page` —— 但是第一个模式不会从URL中捕获任何值。如果第一个模式匹配， `page()` 函数将使用 `num` 参数的默认值"1"。如果第二个模式匹配， `page()` 将使用正则表达式捕获的 `num` 值。

性能

`urlpatterns` 中的每个正则表达式在第一次访问它们时被编译。这使得系统相当快。

urlpatterns 变量的语法

`urlpatterns` 应该是 `url()` 实例的一个Python 列表。

错误处理

当Django 找不到一个匹配请求的URL 的正则表达式时，或者当抛出一个异常时，Django 将调用一个错误处理视图。

这些情况发生时使用的视图通过4个变量指定。它们的默认值应该满足大部分项目，但是通过赋值给它们以进一步的自定义也是可以的。

完整的细节请参见自定义错误视图。

这些值可以在你的根URLconf 中设置。在其它URLconf 中设置这些变量将不会生效果。

它们的值必须是可调用的或者是表示视图的Python 完整导入路径的字符串，可以方便地调用它们来处理错误情况。

这些值是：

- `handler404` —— 参见 `django.conf.urls.handler404`。
- `handler500` —— 参见 `django.conf.urls.handler500`。
- `handler403` —— 参见 `django.conf.urls.handler403`。
- `handler400` —— 参见 `django.conf.urls.handler400`。

包含其它的URLconfs

在任何时候，你的urlpatterns 都可以包含其它URLconf 模块。这实际上将一部分URL 放置与其它URL 下面。

例如，下面是URLconf for the Django 网站自己的URLconf 中一个片段。它包含许多其它URLconf：

```
from django.conf.urls import include, url

urlpatterns = [
    # ... snip ...
    url(r'^community/', include('django_website.aggregator.urls')),
    url(r'^contact/', include('django_website.contact.urls')),
    # ... snip ...
]
```

注意，这个例子中的正则表达式没有包含$（字符串结束匹配符），但是包含一个末尾的反斜杠。每当Django 遇到 `include()` （ `django.conf.urls.include()` ）时，它会去掉URL 中匹配的部分并将剩下的字符串发送给包含的URLconf 做进一步处理。

另外一种包含其它URL 模式的方式是使用一个url() 实例的列表。例如，请看下面的URLconf：

```
from django.conf.urls import include, url

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    url(r'^reports/(?P<id>[0-9]+)/$', credit_views.report),
    url(r'^charge/$', credit_views.charge),
]

urlpatterns = [
    url(r'^$', main_views.homepage),
    url(r'^help/', include('apps.help.urls')),
    url(r'^credit/', include(extra_patterns)),
]
```

在这个例子中，`/credit/reports/` URL将被 `credit.views.report()` 这个Django 视图处理。

这种方法可以用来去除URLconf 中的冗余，其中某个模式前缀被重复使用。例如，考虑这个URLconf：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/history/$', views.history),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/edit/$', views.edit),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/discuss/$', views.discuss),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/permissions/$', views.permissions),
]
```

我们可以改进它，通过只声明共同的路径前缀一次并将后面的部分分组：

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/', include([
        url(r'^history/$', views.history),
        url(r'^edit/$', views.edit),
        url(r'^discuss/$', views.discuss),
        url(r'^permissions/$', views.permissions),
    ])),
]
```

捕获的参数

被包含的URLconf 会收到来之父URLconf 捕获的任何参数，所以下面的例子是合法的：

```
# In settings/urls/main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
]

# In foo/urls/blog.py
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.blog.index),
    url(r'^archive/$', views.blog.archive),
]
```

在上面的例子中，捕获的"`username`"变量将被如期传递给包含的 URLconf。

嵌套的参数

正则表达式允许嵌套的参数，Django 将解析它们并传递给视图。当反查时，Django 将尝试填满所有外围捕获的参数，并忽略嵌套捕获的参数。考虑下面的URL 模式，它带有一个可选的`page`参数：

```
from django.conf.urls import url

urlpatterns = [
    url(r'blog/(page-(\d+)/)?$', blog_articles),                  # bad
    url(r'comments/(?:page-(?P<page_number>\d+)/)?$', comments),  # good
]
```

两个模式都使用嵌套的参数，其解析方式是：例如`blog/page-2/`将匹配`blog_articles`并带有两个位置参数`page-2/`和2。第二个`comments`的模式将匹配`comments/page-2/`并带有一个值为2 的关键字参数`page_number`。这个例子中外围参数是一个不捕获的参数`(?:...)`。

`blog_articles`视图需要最外层捕获的参数来反查，在这个例子中是`page-2/`或者没有参数，而`comments`可以不带参数或者用一个`page_number`值来反查。

嵌套捕获的参数使得视图参数和URL 之间存在强耦合，正如`blog_articles`所示：视图接收URL（`page-2/`）的一部分，而不只是视图感兴趣的值。这种耦合在反查时更加显著，因为反查视图时我们需要传递URL 的一个片段而不只是page 的值。

作为一个经验的法则，当正则表达式需要一个参数但视图忽略它的时候，只捕获视图需要的值并使用非捕获参数。

传递额外的选项给视图函数

URLconfs 具有一个钩子，让你传递一个Python 字典作为额外的参数传递给视图函数。

`django.conf.urls.url()` 函数可以接收一个可选的第三个参数，它是一个字典，表示想要传递给视图函数的额外关键字参数。

例如：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

在这个例子中，对于 `/blog/2005/` 请求，Django 将调用 `views.year_archive(request, year='2005', foo='bar')` 。

这个技术在Syndication 框架 中使用，来传递元数据和选项给视图。

> 处理冲突
>
> URL 模式捕获的命名关键字参数和在字典中传递的额外参数有可能具有相同的名称。当这种情况发生时，将使用字典中的参数而不是URL 中捕获的参数。

传递额外的选项给include()

类似地，你可以传递额外的选项给include()。当你传递额外的选项给include() 时，被包含的URLconf 的每一 行将被传递这些额外的选项。

例如，下面两个URLconf 设置功能上完全相同：

设置一次：

```
# main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^blog/', include('inner'), {'blogid': 3}),
]

# inner.py
from django.conf.urls import url
from mysite import views

urlpatterns = [
    url(r'^archive/$', views.archive),
    url(r'^about/$', views.about),
]
```

设置两次：

```
# main.py
from django.conf.urls import include, url
from mysite import views

urlpatterns = [
    url(r'^blog/', include('inner')),
]

# inner.py
from django.conf.urls import url

urlpatterns = [
    url(r'^archive/$', views.archive, {'blogid': 3}),
    url(r'^about/$', views.about, {'blogid': 3}),
]
```

注意，额外的选项将永远传递给被包含的URLconf 中的每一行，无论该行的视图实际上是否认为这些选项是合法的。由于这个原因，该技术只有当你确定被包含的URLconf 中的每个视图都接收你传递给它们的额外的选项。

URL 的反向解析

在使用Django 项目时，一个常见的需求是获得URL 的最终形式，以用于嵌入到生成的内容中（视图中和显示给用户的URL等）或者用于处理服务器端的导航（重定向等）。

人们强烈希望不要硬编码这些URL（费力、不可扩展且容易产生错误）或者设计一种与URLconf 毫不相关的专门的URL 生成机制，因为这样容易导致一定程度上产生过期的URL。

换句话讲，需要的是一个DRY 机制。除了其它有点，它还允许设计的URL 可以自动更新而不用遍历项目的源代码来搜索并替换过期的URL。

获取一个URL 最开始想到的信息是处理它视图的标识（例如名字），查找正确的URL 的其它必要的信息有视图参数的类型（位置参数、关键字参数）和值。

Django 提供一个办法是让URL 映射是URL 设计唯一的地方。你填充你的URLconf，然后可以双向使用它：

- 根据用户/浏览器发起的URL 请求，它调用正确的Django 视图，并从URL 中提取它的参数需要的值。
- 根据Django 视图的标识和将要传递给它的参数的值，获取与之关联的URL。

第一种方式是我们在前面的章节中一直讨论的用法。第二种方式叫做反向解析URL、反向URL 匹配、反向URL 查询或者简单的URL 反查。

在需要URL 的地方，对于不同层级，Django 提供不同的工具用于URL 反查：

- 在模板中：使用url 模板标签。
- 在Python 代码中：使用 `django.core.urlresolvers.reverse()` 函数。
- 在更高层的与处理Django 模型实例相关的代码中：使用 `get_absolute_url()` 方法。

例子

考虑下面的URLconf：

```
from django.conf.urls import url

from . import views

urlpatterns = [
    #...
    url(r'^articles/([0-9]{4})/$', views.year_archive, name='news-year-archive'),
    #...
]
```

根据这里的设计，某一年nnnn对应的归档的URL是 `/articles/nnnn/` 。

你可以在模板的代码中使用下面的方法获得它们：

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>

<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

在Python 代码中，这样使用：

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2006
    # ...
    return HttpResponseRedirect(reverse('news-year-archive', args=(year,)))
```

如果出于某种原因决定按年归档文章发布的URL应该调整一下，那么你将只需要修改URLconf中的内容。

在某些场景中，一个视图是通用的，所以在URL 和视图之间存在多对一的关系。对于这些情况，当反查URL 时，只有视图的名字还不够。请阅读下一节来了解Django 为这个问题提供的解决办法。

命名URL 模式

为了完成上面例子中的URL 反查，你将需要使用命名的URL 模式。URL 的名称使用的字符串可以包含任何你喜欢的字符。不只限制在合法的Python 名称。

当命名你的URL 模式时，请确保使用的名称不会与其它应用中名称冲突。如果你的URL 模式叫做 `comment`，而另外一个应用中也有一个同样的名称，当你在模板中使用这个名称的时候不能保证将插入哪个URL。

在URL 名称中加上一个前缀，比如应用的名称，将减少冲突的可能。我们建议使用 `myapp-comment` 而不是 `comment`。

URL 命名空间

简介

URL 命名空间允许你反查到唯一的命名URL 模式，即使不同的应用使用相同的URL 名称。第三方应用始终使用带命名空间的URL 是一个很好的实践（我们在教程中也是这么做的）。类似地，它还允许你在一个应用有多个实例部署的情况下反查URL。换句话讲，因为一个应用的多个实例共享相同的命名URL，命名空间将提供一种区分这些命名URL 的方法。

在一个站点上，正确使用URL 命名空间的Django 应用可以部署多次。例如， `django.contrib.admin` 具有一个 `AdminSite` 类，它允许你很容易地部署多个管理站点的实例。在下面的例子中，我们将讨论在两个不同的地方部署教程中的polls 应用，这样我们可以为两种不同的用户（作者和发布者）提供相同的功能。

一个URL 命名空间有两个部分，它们都是字符串：

应用命名空间

它表示正在部署的应用的名称。一个应用的每个实例具有相同的应用命名空间。例如，可以预见Django 的管理站点的应用命名空间是' `admin` '。

实例命名空间

它表示应用的一个特定的实例。实例的命名空间在你的全部项目中应该是唯一的。但是，一个实例的命名空间可以和应用的命名空间相同。它用于表示一个应用的默认实例。例如，Django 管理站点实例具有一个默认的实例命名空间'admin'。 URL 的命名空间使用':' 操作符指定。例如，管理站点应用的主页使用' `admin:index` '。它表示' `admin` ' 的一个命名空间和' `index` ' 的一个命名URL。

命名空间也可以嵌套。命名URL' `sports:polls:index` ' 将在命名空间' `polls` '中查找' `index` '，而 `poll` 定义在顶层的命名空间' `sports` ' 中。

反查带命名空间的URL

当解析一个带命名空间的URL（例如' `polls:index` '）时，Django 将切分名称为多个部分，然后按下面的步骤查找：

1. 首先，Django 查找匹配的应用的命名空间(在这个例子中为'`polls`'）。这将得到该应用实例的一个列表。
2. 如果有定义当前 应用，Django 将查找并返回那个实例的URL 解析器。当前 应用可以通过请求上的一个属性指定。希望可以多次部署的应用应该设置正在处理的`request`上的`current_app`属性。

```
Changed in Django 1.8:

在以前版本的Django 中，你必须在用于渲染模板的每个`Context` 或 `RequestContext`上设置`current_app`
```

当前应用还可以通过`reverse()`函数的一个参数手工设定。

1. 如果没有当前应用。Django 将查找一个默认的应用实例。默认的应用实例是实例命名空间 与应用命名空间 一致的那个实例（在这个例子中，`polls`的一个叫做'`polls`'的实例）。
2. 如果没有默认的应用实例，Django 将该应用挑选最后部署的实例，不管实例的名称是什么。
3. 如果提供的命名空间与第1步中的应用命名空间 不匹配，Django 将尝试直接将此命名空间作为一个实例命名空间查找。

如果有嵌套的命名空间，将为命名空间的每个部分重复调用这些步骤直至剩下视图的名称还未解析。然后该视图的名称将被解析到找到的这个命名空间中的一个URL。

例子

为了演示解析的策略，考虑教程中polls 应用的两个实例：'`author-polls`'和'`publisher-polls`'。假设我们已经增强了该应用，在创建和显示投票时考虑了实例命名空间。

```
#urls.py

from django.conf.urls import include, url

urlpatterns = [
    url(r'^author-polls/', include('polls.urls', namespace='author-polls', app_name='poll
    url(r'^publisher-polls/', include('polls.urls', namespace='publisher-polls', app_name
]
```

```
#polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
    ...
]
```

根据以上设置，可以使用下面的查询：

- 如果其中一个实例是当前实例 —— 如果我们正在渲染'`author-polls`'实例的`detail`页面 ——'`polls:index`'将解析成'`author-polls`'实例的主页面；例如下面两个都将解析成"`/author-polls/`"。

在基于类的视图的方法中：

```
reverse('polls:index', current_app=self.request.resolver_match.namespace)
```

和在模板中：

```
{% url 'polls:index' %}
```

注意，在模板中的反查需要添加`request`的`current_app`属性，像这样：

```
def render_to_response(self, context, **response_kwargs):
    self.request.current_app = self.request.resolver_match.namespace
    return super(DetailView, self).render_to_response(context, **response_kwargs)
```

- 如果没有当前实例 —— 如果我们在站点的其它地方渲染一个页面 ——'`polls:index`'将解析到最后注册的`polls`的一个实例。因为没有默认的实例（命名空间为'polls'的实例），将使用注册的`polls`的最后一个实例。它将是'`publisher-polls`'，因为它是在`urlpatterns`中最后一个声明的。
- '`author-polls:index`'将永远解析到'`author-polls`'实例的主页（'`publisher-polls`'类似）。

如果还有一个默认的实例 —— 例如，一个名为'`polls`'的实例 —— 上面例子中唯一的变化是当没有当前实例的情况（上述第二种情况）。在这种情况下'`polls:index`'将解析到默认实例而不是`urlpatterns`中最后声明的实例的主页。

URL 命名空间和被包含的URLconf

被包含的URLconf 的命名空间可以通过两种方式指定。

首先，在你构造你的URL 模式时，你可以提供 应用 和 实例的命名空间给 `include()` 作为参数。例如：

```
url(r'^polls/', include('polls.urls', namespace='author-polls', app_name='polls')),
```

这将包含 `polls.urls` 中定义的URL 到应用命名空间 ' `polls` '中，其实例命名空间为' `author-polls` '。

其次，你可以include 一个包含嵌套命名空间数据的对象。如果你 `include()` 一个 `url()` 实例的列表，那么该对象中包含的URL 将添加到全局命名空间。然而，你还可以 `include()` 一个3个元素的元组：

```
(<list of url() instances>, <application namespace>, <instance namespace>)
```

例如：

```
from django.conf.urls import include, url

from . import views

polls_patterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
]

url(r'^polls/', include((polls_patterns, 'polls', 'author-polls'))),
```

这将include 命名的URL 模式到给定的应用和实例命名空间中。

例如，Django 的管理站点部署的实例叫 `AdminSite` 。 `AdminSite` 对象具有一个 `urls` 属性：一个3元组，包含管理站点中的所有URL 模式和应用的命名空间' `admin` '以及管理站点实例的名称。你 `include()` 到你项目的 `urlpattern` s 中的是这个 `urls` 属性。

请确保传递一个元组给 `include()` 。如果你只是传递3个参数： `include(polls_patterns, 'polls', 'author-polls')` ，Django 不会抛出一个错误，但是根据 `include()` 的功能，' `polls` ' 将是实例的命名空间而' `author-polls` ' 将是应用的命名空间，而不是反过来的。

> 译者：Django 文档协作翻译小组，原文：URLconfs。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写视图

一个视图函数，或者简短来说叫做视图，是一个简单的Python函数，它接受web请求，并且返回web响应。响应可以是一张网页的HTML内容，一个重定向，一个404错误，一个XML文档，或者一张图片... 是任何东西都可以。无论视图本身包含什么逻辑，都要返回响应。代码写在哪里也无所谓，只要它在你的Python目录下面。除此之外没有更多的要求了——可以说"没有什么神奇的地方"。为了能够把代码放在某个地方，惯例是把视图放在叫做views.py的文件中，然后把它放到你的项目或者应用目录里。

一个简单的视图

下面是一个返回当前日期和时间作为HTML文档的视图：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

让我们逐行阅读上面的代码：

- 首先，我们从 django.http模块导入了HttpResponse类，以及Python的datetime库。
- 接着，我们定义了current_datetime函数。它是一个视图函数。每个视图函数都应接收HttpRequest对象作为第一个参数，一般叫做request。
- 注意视图函数的名称并不重要；不需要用一个统一的命名方式来命名，以便让Django识别它。我们将其命名为current_datetime，是因为这个名称能够精确地反映出它的功能。
- 这个视图会返回一个HttpResponse对象，其中包含生成的响应。每个视图函数都要返回HttpResponse对象。（有例外，我们接下来会讲。）

> Django中的时区
>
> Django中包含一个TIME_ZONE设置，默认为America/Chicago。可能并不是你住的地方，所以你可能会在设置文件里修改它。

把你的URL映射到视图

所以，再重复一遍，这个视图函数返回了一个包含当前日期和时间的HTML页面。你需要创建URLconf来展示在特定的URL这一视图；详见URL 分发器。

返回错误

在Django中返回HTTP错误是相当容易的。有一些HttpResponse的子类代表不是200（“OK”）的HTTP状态码。你可以在request/response文档中找到所有可用的子类。你可以返回那些子类的一个实例，而不是普通的HttpResponse，来表示一个错误。例如：

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

由于一些状态码不太常用，所以不是每个状态码都有一个特化的子类。然而，如HttpResponse文档中所说的那样，你也可以向HttpResponse的构造器传递HTTP状态码，来创建你想要的任何状态码的返回类。例如：

```
from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

由于404错误是最常见的HTTP错误，所以处理这一错误的方式非常便利。

Http404异常

class django.http.Http404

当你返回一个像HttpResponseNotFound这样的错误时，它会输出这个错误页面的HTML作为结果：

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

为了便利起见，也因为你的站点有个一致的404页面是个好主意，Django提供了Http404异常。如果你在视图函数中的任何地方抛出Http404异常，Django都会捕获它，并且带上HTTP404错误码返回你应用的标准错误页面。

像这样：

```
from django.http import Http404
from django.shortcuts import render_to_response
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render_to_response('polls/detail.html', {'poll': p})
```

为了尽可能利用 Http404，你应该创建一个用来在404错误产生时展示的模板。这个模板应该叫做404.html，并且在你的模板树中位于最顶层。

如果你在抛出Http404异常时提供了一条消息，当DEBUG为True时它会出现在标准404模板的展示中。你可以将这些消息用于调试；但他们通常不适用于404模板本身。

自定义错误视图

Django中默认的错误视图对于大多数web应用已经足够了，但是如果你需要任何自定义行为，重写它很容易。只要在你的URLconf中指定下面的处理器（在其他任何地方设置它们不会有效）。

handler404覆盖了page_not_found()视图：

```
handler404 = 'mysite.views.my_custom_page_not_found_view'
```

handler500覆盖了server_error()视图：

```
handler500 = 'mysite.views.my_custom_error_view'
```

handler403覆盖了permission_denied()视图：

```
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

handler400覆盖了bad_request()视图：

```
handler400 = 'mysite.views.my_custom_bad_request_view'
```

Django 的快捷函数

`django.shortcuts` 收集了“跨越”多层MVC 的辅助函数和类。 换句话讲，这些函数/类为了方便，引入了可控的耦合。

render

```
render(request, template_name[, context][, context_instance][, content_type][, status][, c
```

结合一个给定的模板和一个给定的上下文字典，并返回一个渲染后的 `HttpResponse` 对象。

`render()` 与以一个强制使用 `RequestContext` 的 `context_instance` 参数调用 `render_to_response()` 相同。

Django 不提供返回 `TemplateResponse` 的快捷函数，因为 `TemplateResponse` 的构造与 `render()` 提供的便利是一个层次的。

必选的参数

`request`

用于生成响应的请求对象。

`template_name`

要使用的模板的完整名称或者模板名称的一个序列。

可选的参数

`context`

添加到模板上下文的一个字典。默认是一个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。

```
Django 1.8 的改变：

context 参数之前叫做dictionary。这个名字在Django 1.8 中废弃并将在Django 2.0 中删除。
```

`context_instance`

渲染模板的上下文实例。默认情况下，模板将使用 `RequestContext` 实例（值来自 `request` 和 `context`）渲染。

版本 1.8 以后废弃：

废弃context_instance 参数。仅仅使用context。

`content_type`

生成的文档要使用的MIME 类型。默认为 `DEFAULT_CONTENT_TYPE` 设置的值。

`status`

响应的状态码。默认为200。

`current_app`

指示哪个应用包含当前的视图。更多信息，参见带命名空间的URL 的解析。

版本1.8 以后废弃：

废弃current_app 参数。你应该设置request.current_app。

`using`

用于加载模板使用的模板引擎的名称。

Changed in Django 1.8:

增加using 参数。

Changed in Django 1.7:

增加dirs 参数。

Deprecated since version 1.8:

废弃dirs 参数。

示例

下面的示例渲染模板 `myapp/index.html`，MIME 类型为 `application/xhtml+xml`：

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {"foo": "bar"},
        content_type="application/xhtml+xml")
```

这个示例等同于：

```
from django.http import HttpResponse
from django.template import RequestContext, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = RequestContext(request, {'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

render_to_response

`render_to_response(template_name[, context][, context_instance][, content_type][, status][`

根据一个给定的上下文字典渲染一个给定的目标，并返回渲染后的HttpResponse。

必选的参数

`template_name`

使用的模板的完整名称或者模板名称的序列。如果给出的是一个序列，将使用存在的第一个模板。关于如何查找模板的更多信息请参见 模板加载的文档 。

可选的参数

`context`

添加到模板上下文中的字典。默认是个空字典。如果字典中的某个值是可调用的，视图将在渲染模板之前调用它。

```
Changed in Django 1.8:

context 参数之前叫做dictionary。 这个名字在Django 1.8 中废弃并将在Django 2.0 中删除。
```

`context_instance`

渲染模板使用的上下文实例。默认情况下，模板将 `Context` 实例（值来自 `context` ）渲染。如果你需要使用上下文处理器，请使用 `RequestContext` 实例渲染模板。你的代码看上去像是这样：

```
return render_to_response('my_template.html',
                          my_context,
                          context_instance=RequestContext(request))
```

```
版本1.8 以后废弃：

废弃context_instance 参数。 仅仅使用context。
```

`content_type`

生成的文档使用的MIME 类型。默认为 `DEFAULT_CONTENT_TYPE` 设置的值。

`status`

相应的状态码。默认为200。

`using`

加载模板使用的模板引擎的名称。

```
Changed in Django 1.8:

添加status 和using 参数。
```

```
Changed in Django 1.7:

增加dirs 参数。
```

```
Deprecated since version 1.8:

废弃dirs 参数。
```

示例

下面的示例渲染模板 `myapp/index.html`， MIIME 类型为 `application/xhtml+xml` ：

```
from django.shortcuts import render_to_response

def my_view(request):
    # View code here...
    return render_to_response('myapp/index.html', {"foo": "bar"},
        content_type="application/xhtml+xml")
```

这个示例等同于：

```
from django.http import HttpResponse
from django.template import Context, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = Context({'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

redirect

`redirect(to, [permanent=False, ]*args, **kwargs)[source]`

为传递进来的参数返回HttpResponseRedirect 给正确的URL 。

参数可以是：

- 一个模型：将调用模型的 `get_absolute_url()` 函数
- 一个视图，可以带有参数：将使用 `urlresolvers.reverse` 来反向解析名称
- 一个绝对的或相对的URL，将原样作为重定向的位置。

默认返回一个临时的重定向；传递 `permanent=True` 可以返回一个永久的重定向。

```
Django 1.7 中的改变：

增加使用相对URL 的功能。
```

示例

你可以用多种方式使用 `redirect()` 函数。

通过传递一个对象；将调用 `get_absolute_url()` 方法来获取重定向的URL：

```
from django.shortcuts import redirect

def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)
```

通过传递一个视图的名称，可以带有位置参数和关键字参数；将使用 `reverse()` 方法反向解析URL：

```
def my_view(request):
    ...
    return redirect('some-view-name', foo='bar')
```

传递要重定向的一个硬编码的URL：

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

完整的URL 也可以：

```
def my_view(request):
    ...
    return redirect('http://example.com/')
```

默认情况下， `redirect()` 返回一个临时重定向。以上所有的形式都接收一个 `permanent` 参数；如果设置为 `True`，将返回一个永久的重定向：

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

get_object_or_404

`get_object_or_404(klass, *args, **kwargs)[source]`

在一个给定的模型管理器上调用 `get()`，但是引发 `Http404` 而不是模型的 `DoesNotExist` 异常。

必选的参数

`klass`

获取该对象的一个 `Model` 类， `Manager` 或 `QuerySet` 实例。

`**kwargs`

查询的参数，格式应该可以被 `get()` 和 `filter()` 接受。

示例

下面的示例从 `MyModel` 中使用主键1 来获取对象：

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

这个示例等同于：

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```

最常见的用法是传递一个 `Model`，如上所示。然而，你还可以传递一个 `QuerySet` 实例：

```
queryset = Book.objects.filter(title__startswith='M')
get_object_or_404(queryset, pk=1)
```

上面的示例有点做作，因为它等同于：

```
get_object_or_404(Book, title__startswith='M', pk=1)
```

但是如果你的queryset 来自其它地方，它就会很有用了。

最后你还可以使用一个管理器。如果你有一个自定义的管理器，它将很有用：

```
get_object_or_404(Book.dahl_objects, title='Matilda')
```

你还可以使用关联的 管理器：

```
author = Author.objects.get(name='Roald Dahl')
get_object_or_404(author.book_set, title='Matilda')
```

注：与get()一样，如果找到多个对象将引发一个 `MultipleObjectsReturned` 异常。

get_list_or_404

`get_list_or_404(klass, *args, **kwargs)[source]`

返回一个给定模型管理器上filter() 的结果，并将结果映射为一个列表，如果结果为空则返回Http404。

必选的参数

`klass`

获取该列表的一个 `Model` 、 `Manager` 或 `QuerySet` 实例。

`**kwargs`

查寻的参数，格式应该可以被 `get()` 和 `filter()` 接受。

示例

下面的示例从 `MyModel` 中获取所有发布出来的对象：

```
from django.shortcuts import get_list_or_404

def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

这个示例等同于：

```
from django.http import Http404

def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404("No MyModel matches the given query.")
```

> 译者：Django 文档协作翻译小组，原文：Shortcuts。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

视图装饰器

Django为视图提供了数个装饰器，用以支持相关的HTTP服务。

允许的HTTP 方法

`django.views.decorators.http` 包里的装饰器可以基于请求的方法来限制对视图的访问。若条件不满足会返回 `django.http.HttpResponseNotAllowed` 。

`require_http_methods` (request_method_list)[source]

限制视图只能服务规定的http方法。用法：

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

@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

注意，方法名必须大写。

`require_GET()`

只允许视图接受GET方法的装饰器。

`require_POST()`

只允许视图接受POST方法的装饰器。

`require_safe()`

只允许视图接受 GET 和 HEAD 方法的装饰器。 这些方法通常被认为是安全的，因为方法不该有请求资源以外的目的。

> 注
>
> Django 会自动清除对HEAD 请求的响应中的内容而只保留头部，所以在你的视图中你处理HEAD 请求的方式可以完全与GET 请求一致。因为某些软件，例如链接检查器，依赖于HEAD 请求，所以你可能应该使用 `require_safe` 而不是 `require_GET` 。

可控制的视图处理

`django.views.decorators.http` 中的以下装饰器可以用来控制特定视图的缓存行为。

`condition` (etag_func=None, last_modified_func=None)[source]

`etag` (etag_func)[source]

`last_modified` (last_modified_func)[source]

这些装饰器可以用于生成ETag 和Last-Modified 头部；参考 conditional view processing.

GZip 压缩

`django.views.decorators.gzip` 里的装饰器基于每个视图控制其内容压缩。

`gzip_page()`

如果浏览器允许gzip 压缩，这个装饰器将对内容进行压缩。它设置相应的 `Vary` 头部，以使得缓存根据 `Accept-Encoding` 头来存储信息。

Vary 头部

`django.views.decorators.vary` 可以用来基于特定的请求头部来控制缓存。

`vary_on_cookie` (func)[source]

`vary_on_headers` (*headers)[source]

到当构建缓存的键时，Vary 头部定义一个缓存机制应该考虑的请求头。

参见使用vary 头部。

> 译者：Django 文档协作翻译小组，原文：Decorators。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

参考

内建的视图

有几个Django 的内建视图在编写视图 中讲述，文档的其它地方也会有所讲述。

开发环境中的文件服务器

static.serve(request, path, document_root, show_indexes=False)

在本地的开发环境中，除了你的项目中的静态文件，可能还有一些文件，出于方便，你希望让Django 来作为服务器。serve() 视图可以用来作为任意目录的服务器。（该视图不能用于生产环境，应该只用于开发时辅助使用；在生产环境中你应该使用一个真实的前端Web 服务器来服务这些文件）。

最常见的例子是用户上传文档到 `MEDIA_ROOT` 中。 `django.contrib.staticfiles` 用于静态文件且没有对用户上传的文件做处理，但是你可以通过在URLconf 中添加一些内容来让Django 作为MEDIA_ROOT 的服务器：

```
from django.conf import settings
from django.views.static import serve

# ... the rest of your URLconf goes here ...

if settings.DEBUG:
    urlpatterns += [
        url(r'^media/(?P<path>.*)$', serve, {
            'document_root': settings.MEDIA_ROOT,
        }),
   ]
```

注意，这里的代码片段假设你的 `MEDIA_URL` 的值为 `'/media/'` 。它将调用 `serve()` 视图，传递来自URLconf 的路径和（必选的） `document_root` 参数。

因为定义这个URL 模式显得有些笨拙，Django 提供一个小巧的URL 辅助函数 `static()` ，它接收 `MEDIA_URL` 这样的参数作为前缀和视图的路径如 `'django.views.static.serve'` 。其它任何函数参数都将透明地传递给视图。

错误视图

Django 原生自带几个默认视图用于处理HTTP 错误。若要使用你自定义的视图覆盖它们，请参见自定义错误视图。

404 (page not found) 视图

defaults.page_not_found(request, template_name='404.html')

当你在一个视图中引发Http404 时，Django 将加载一个专门的视图用于处理404 错误。默认为 `django.views.defaults.page_not_found()` 视图，它产生一个非常简单的“Not Found” 消息或者渲染 `404.html` 模板，如果你在根模板目录下创建了它的话。

默认的404 视图将传递一个变量给模板：`request_path`，它是导致错误的URL。

关于404 视图需要注意的3点：

- 如果Django 在检测URLconf 中的每个正则表达式后没有找到匹配的内容也将调用404 视图。
- 404 视图会被传递一个 `RequestContext` 并且可以访问模板上下文处理器提供的变量（例如 `MEDIA_URL`）。
- 如果DEBUG 设置为 `True`（在你的 `settings` 模块中），那么将永远不会调用404 视图，而是显示你的URLconf 并带有一些调试信息。

500 (server error) 视图

defaults.server_error(request, template_name='500.html')

类似地，在视图代码中出现运行时错误，Django 将执行特殊情况下的行为。如果一个视图导致异常，Django 默认情况下将调用 `django.views.defaults.server_error` 视图，它产生一个非常简单的“Server Error” 消息或者渲染 `500.html`，如果你在你的根模板目录下定义了它的话。

默认的500 视图不会传递变量给 `500.html` 模板，且使用一个空 `Context` 来渲染以减少再次出现错误的可能性。

如果 `DEBUG` 设置为 `True`（在你的 `settings` 模块中），那么将永远不会调用500 视图，而是显示回溯并带有一些调试信息。

403 (HTTP Forbidden) 视图

defaults.permission_denied(request, template_name='403.html')

与404 和500 视图一样，Django 具有一个处理403 Forbidden 错误的视图。如果一个视图导致一个403 视图，那么Django 将默认调用 `django.views.defaults.permission_denied` 视图。

该视图加载并渲染你的根模板目录下的 `403.html`，如果这个文件不存在则根据RFC 2616（HTTP 1.1 Specification）返回“403 Forbidden”文本。

`django.views.defaults.permission_denied` 通过 `PermissionDenied` 异常触发。若要拒绝访问一个视图，你可以这样视图代码：

```
from django.core.exceptions import PermissionDenied

def edit(request, pk):
    if not request.user.is_staff:
        raise PermissionDenied
    # ...
```

400 (bad request) 视图

defaults.bad_request(request, template_name='400.html')

当Django 中引发一个 `SuspiciousOperation` 时，它可能通过Django 的一个组件处理（例如重设会话的数据）。如果没有特殊处理，Django 将认为当前的请求时一个'bad request' 而不是一个server error。

`django.views.defaults.bad_request` 和server_error 视图非常相似，除了返回400 状态码来表示错误来自客户端的操作。

`bad_request` 视图同样只是在 `DEBUG` 为 `False` 时使用。

> 译者：Django 文档协作翻译小组，原文：Built-in Views。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

TemplateResponse 和 SimpleTemplateResponse

标准的 `HttpResponse` 对象是静态的结构。在构造的时候提供给它们一个渲染之前的内容，但是当内容改变时它们却不能很容易地完成相应的改变。

然而，有时候允许装饰器或者中间件在响应被构造之后修改它是很有用的。例如，你可能想改变使用的模板，或者添加额外的数据到上下文中。

`TemplateResponse` 提供了实现这一点的方法。与基本的HttpResponse 对象不同， `TemplateResponse` 对象会记住视图提供的模板和上下文的详细信息来计算响应。响应的最终结果在后来的响应处理过程中直到需要时才计算。

SimpleTemplateResponse 对象

`class SimpleTemplateResponse[source]`

属性

`SimpleTemplateResponse.template_name`

渲染的模板的名称。接收一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

例如： `['foo.html', 'path/to/bar.html']`

```
Deprecated since version 1.8:
template_name 以前只接受一个 Template。
```

`SimpleTemplateResponse.context_data`

渲染模板时用到的上下文数据。它必须是一个 `dict` 。

例如： `{'foo': 123}`

```
Deprecated since version 1.8:
context_data 以前只接受一个 Context。
```

`SimpleTemplateResponse.rendered_content[source]`

使用当前的模板和上下文数据渲染出来的响应内容。

`SimpleTemplateResponse.is_rendered[source]`

一个布尔值，表示响应内容是否已经渲染。

方法

`SimpleTemplateResponse.__init__(template, context=None, content_type=None, status=None, ch`

使用给定的模板、上下文、Content-Type、HTTP 状态和字符集初始化一个 `SimpleTemplateResponse` 对象。

`template`

一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

> Deprecated since version 1.8:
>
> template 以前只接受一个Template。

`context`

一个 `dict`，包含要添加到模板上下文中的值。它默认是一个空的字典。

> Deprecated since version 1.8:
>
> context 以前只接受一个Context。

`content_type`

HTTP `Content-Type` 头部包含的值，包含MIME 类型和字符集的编码。 如果指定 `content_type`，则使用它的值。 否则，使用 `DEFAULT_CONTENT_TYPE`。

`status`

响应的HTTP 状态码。

`charset`

响应编码使用的字符集。 如果没有给出则从 `content_type` 中提取，如果提取不成功则使用 `DEFAULT_CHARSET` 设置。

`using`

加载模板使用的模板引擎的名称。

> Changed in Django 1.8:
>
> 添加charset 和using 参数。

`SimpleTemplateResponse.resolve_context(context)[source]`

预处理即将用于渲染模板的上下文数据。接受包含上下文数据的一个 `dict` 。默认返回同一个 `dict` 。

若要自定义上下文，请覆盖这个方法。

```
Changed in Django 1.8:

resolve_context 返回一个dict。它以前返回一个Context。
```

```
Deprecated since version 1.8:

resolve_context 不再接受Context。
```

`SimpleTemplateResponse.resolve_template(template)[source]`

解析渲染用到的模板实例。接收一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

返回将被渲染的模板对象。

若要自定义模板的加载，请覆盖这个方法。

```
Changed in Django 1.8:

resolve_template 返回一个与后端有关的模板对象。它以前返回一个Template。
```

```
Deprecated since version 1.8:

resolve_template 不再接受一个 Template。
```

`SimpleTemplateResponse.add_post_render_callback()[source]`

添加一个渲染之后调用的回调函数。这个钩子可以用来延迟某些特定的处理操作（例如缓存）到渲染之后。

如果 `SimpleTemplateResponse` 已经渲染，那么回调函数将立即执行。

调用的时只传递给回调函数一个参数 —— 渲染后的 `SimpleTemplateResponse` 实例。

如果回调函数返回非 `None` 值，它将用作响应并替换原始的响应对象（以及传递给下一个渲染之后的回调函数，以此类推）。

`SimpleTemplateResponse.render()[source]`

设置 `response.content` 为 `SimpleTemplateResponse.rendered_content` 的结果，执行所有渲染之后的回调函数，最后返回生成的响应对象。

`render()` 只在第一次调用它时其作用。以后的调用将返回第一次调用的结果。

TemplateResponse 对象

`class TemplateResponse[source]`

`TemplateResponse` 是 `SimpleTemplateResponse` 的子类，而且能知道当前的 `HttpRequest`。

方法

`TemplateResponse.__init__(request, template, context=None, content_type=None, status=None,`

使用给定的模板、上下文、 `Content-Type`、HTTP 状态和字符集实例化一个 `TemplateResponse` 对象。

`request`

An `HttpRequest` instance.

`template`

一个与后端有关的模板对象（例如 `get_template()` 返回的对象）、模板的名称或者一个模板名称列表。

```
Deprecated since version 1.8:
template 以前只接受一个Template。
```

`context`

一个 `dict`，包含要添加到模板上下文中的值。 它默认是一个空的字典。

```
Deprecated since version 1.8:
context 以前只接受一个Context。
```

`content_type`

HTTP `Content-Type` 头部包含的值，包含MIME 类型和字符集的编码。如果指定 `content_type`，则使用它的值。否则，使用 `DEFAULT_CONTENT_TYPE`。

`status`

响应的HTTP 状态码。

`current_app`

包含当前视图的应用。更多信息，参见带命名空间的URL 解析策略。

```
Deprecated since version 1.8:
废弃current_app 参数。你应该去设置request.current_app。
```

`charset`

响应编码使用的字符集。如果没有给出则从content_type中提取，如果提取不成功则使用 DEFAULT_CHARSET 设置。

`using`

加载模板使用的模板引擎的名称。

```
Changed in Django 1.8:

添加charset 和using 参数。
```

渲染的过程

在 `TemplateResponse` 实例返回给客户端之前，它必须被渲染。渲染的过程采用模板和上下文变量的中间表示形式，并最终将它转换为可以发送给客户端的字节流。

三种情况下会渲染 `TemplateResponse` ：

- 当使用SimpleTemplateResponse.render() 方法显示渲染TemplateResponse 实例的时候。
- 当通过给response.content 赋值显式设置响应内容的时候。
- 在应用模板响应中间件之后，应用响应中间件之前。

`TemplateResponse` 只能渲染一次。 `SimpleTemplateResponse.render()` 的第一次调用设置响应的内容；以后的响应不会改变响应的内容。

然而，当显式给 `response.content` 赋值时，修改会始终生效。如果你想强制重新渲染内容，你可以重新计算渲染的内容并手工赋值给响应的内容：

```
# Set up a rendered TemplateResponse
>>> from django.template.response import TemplateResponse
>>> t = TemplateResponse(request, 'original.html', {})
>>> t.render()
>>> print(t.content)
Original content

# Re-rendering doesn't change content
>>> t.template_name = 'new.html'
>>> t.render()
>>> print(t.content)
Original content

# Assigning content does change, no render() call required
>>> t.content = t.rendered_content
>>> print(t.content)
New content
```

渲染后的回调函数

某些操作 —— 例如缓存 —— 不可以在没有渲染的模板上执行。它们必须在完整的渲染后的模板上执行。

如果你正在使用中间件，解决办法很容易。中间件提供多种在从视图退出时处理响应的机会。如果你向响应中间件添加一些行为，它们将保证在模板渲染之后执行。

然而，如果正在使用装饰器，就不会有这样的机会。装饰器中定义的行为会立即执行。

为了补偿这一点（以及其它类似的使用情形） `TemplateResponse` 允许你注册在渲染完成时调用的回调函数。使用这个回调函数，你可以延迟某些关键的处理直到你可以保证渲染后的内容是可以访问的。

要定义渲染后的回调函数，只需定义一个接收一个响应作为参数的函数并将这个函数注册到模板响应中：

```
from django.template.response import TemplateResponse

def my_render_callback(response):
    # Do content-sensitive processing
    do_post_processing()

def my_view(request):
    # Create a response
    response = TemplateResponse(request, 'mytemplate.html', {})
    # Register the callback
    response.add_post_render_callback(my_render_callback)
    # Return the response
    return response
```

`my_render_callback()` 将在 `mytemplate.html` 渲染之后调用，并将被传递一个 `TemplateResponse` 实例作为参数。

如果模板已经渲染，回调函数将立即执行。

使用TemplateResponse 和 SimpleTemplateResponse

`TemplateResponse` 对象和普通的 `django.http.HttpResponse` 一样可以用于任何地方。它可以用来作为 `render()` 和 `render_to_response()` 的另外一种选择。

例如，下面这个简单的视图使用一个简单模板和包含查询集的上下文返回一个 `TemplateResponse` ：

```
from django.template.response import TemplateResponse

def blog_index(request):
    return TemplateResponse(request, 'entry_list.html', {'entries': Entry.objects.all()})
```

> 译者：Django 文档协作翻译小组，原文：TemplateResponse objects。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

文件上传

文件上传

当Django在处理文件上传的时候，文件数据被保存在 `request. FILES` （更多关于 `request` 对象的信息 请查看 请求和响应对象）。这篇文档阐述了文件如何上传到内存和硬盘，以及如何自定义默认的行为。

> 警告
>
> 允许任意用户上传文件是存在安全隐患的。更多细节请在用户上传的内容中查看有关安全指导的话题。

基本的文件上传

考虑一个简单的表单，它含有一个 `FileField` ：

```
# In forms.py...
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

处理这个表单的视图会在 `request` 中接受到上传文件的数据。 `FILES` 是个字典，它包含每个 `FileField` 的键 （或者 `ImageField` ， `FileField` 的子类）。这样的话就可以用 `request.FILES['file']` 来存放表单中的这些数据了。

注意 `request.FILES` 会仅仅包含数据，如果请求方法为 `POST` ，并且发送请求的 `<form>` 拥有 `enctype="multipart/form-data"` 属性。否则 `request.FILES` 为空。

大多数情况下，你会简单地从 `request` 向表单中传递数据，就像绑定上传文件到表单描述的那样。这样类似于：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render_to_response('upload.html', {'form': form})
```

注意我们必须向表单的构造器中传递 `request.FILES` 。这是文件数据绑定到表单的方法。

这里是一个普遍的方法，可能你会采用它来处理上传文件：

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

遍历 `UploadedFile.chunks()` ，而不是使用 `read()` ，能确保大文件并不会占用系统过多的内存。

`UploadedFile` 对象也拥有一些其他的方法和属性；完整参考请见UploadedFile。

使用模型处理上传文件

如果你在 `Model` 上使用 `FileField` 保存文件，使用 `ModelForm` 可以让这个操作更加容易。调用 `form.save()` 的时候，文件对象会保存在相应的 `FileField` 的 `upload_to` 参数指定的地方。

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = ModelFormWithFileField(request.POST, request.FILES)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = ModelFormWithFileField()
    return render(request, 'upload.html', {'form': form})
```

如果你手动构造一个对象，你可以简单地把文件对象从 `request.FILE` 赋值给模型：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES['file'])
            instance.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

上传处理器

当用户上传一个文件的时候，Django会把文件数据传递给上传处理器 – 一个小型的类，会在文件数据上传时处理它。上传处理器在 `FILE_UPLOAD_HANDLERS` 中定义，默认为：

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

`MemoryFileUploadHandler` 和 `TemporaryFileUploadHandler` 一起提供了Django的默认文件上传行为，将小文件读取到内存中，大文件放置在磁盘中。

你可以编写自定义的处理器，来定制Django如何处理文件。例如，你可以使用自定义处理器来限制用户级别的配额，在运行中压缩数据，渲染进度条，甚至是向另一个储存位置直接发送数据，而不把它存到本地。关于如何自定义或者完全替换处理器的行为，详见编写自定义的上传处理器。

上传数据在哪里储存

在你保存上传文件之前，数据需要储存在某个地方。

通常，如果上传文件小于2.5MB，Django会把整个内容存到内存。这意味着，文件的保存仅仅涉及到从内存读取和写到磁盘，所以非常快。

但是，如果上传的文件很大，Django会把它写入一个临时文件，储存在你系统的临时目录中。在类Unix的平台下，你可以认为Django生成了一个文件，名称类似于 `/tmp/tmpzfp6I6.upload` 。如果上传的文件足够大，你可以观察到文件大小的增长，由于Django向磁盘写入数据。

这些特定值 – 2.5 MB， `/tmp` ，以及其它 -- 都仅仅是"合理的默认值"，它们可以自定义，这会在下一节中描述。

更改上传处理器的行为

Django的文件上传处理器的行为由一些设置控制。详见文件上传设置。

在运行中更改上传处理器

有时候一些特定的视图需要不同的上传处理器。在这种情况下，你可以通过修改 `request.upload_handlers` ，为每个请求覆盖上传处理器。通常，这个列表会包含 `FILE_UPLOAD_HANDLERS` 提供的上传处理器，但是你可以把它修改为其它列表。

例如，假设你编写了 `ProgressBarUploadHandler` ，它会在上传过程中向某类AJAX控件提供反馈。你可以像这样将它添加到你的上传处理器中：

```
request.upload_handlers.insert(0, ProgressBarUploadHandler())
```

在这中情况下你可能想要使用 `list.insert()` （而不是 `append()` ），因为进度条处理器需要在任何其他处理器 之前执行。要记住，多个上传处理器是按顺序执行的。

如果你想要完全替换上传处理器，你可以赋值一个新的列表：

```
request.upload_handlers = [ProgressBarUploadHandler()]
```

> 注意
>
> 你只可以在访问 `request.POST` 或者 `request.FILES` 之前修改上传处理器-- 在上传处理工作执行之后再修改上传处理就毫无意义了。如果你在读取 `request.FILES` 之后尝试修改 `request.upload_handlers` ，Django会抛出异常。
>
> 所以，你应该在你的视图中尽早修改上传处理器。
>
> `CsrfViewMiddleware` 也会访问 `request.POST` ，它是默认开启的。意思是你需要在你的视图中使用 `csrf_exempt()` ，来允许你修改上传处理器。接下来在真正处理请求的函数中，需要使用 `csrf_protect()` 。注意这意味着处理器可能会在CSRF验证完成之前开始接收上传文件。例如：

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler())
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Process request
```

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

File对象

`django.core.files` 模块及其子模块包含了一些用于基本文件处理的内建类。

File类

`class File(file_object)`

`File` 类是Python file 对象的一个简单封装，并带有Django特定的附加功能。需要表示文件的时候，Django内部会使用这个类。

File对象拥有下列属性和方法：

`name`

含有 `MEDIA_ROOT` 相对路径的文件名称。

`size`

文件的字节数。

`file`

这个类所封装的，原生的file 对象。

`mode`

文件的读写模式。

`open([mode=None])`

打开或者重新打开文件（同时会执行 `File.seek(0)` ）。 `mode` 参数的值和Python内建的 `open()` 相同。

重新打开一个文件时，无论文件原先以什么模式打开， `mode` 都会覆盖； `None` 的意思是以原先的模式重新打开。

`read([num_bytes=None])`

读取文件内容。可选的 `size` 参数是要读的字节数；没有指定的话，文件会一直读到结尾。

`__iter__()`

迭代整个文件，并且每次生成一行。

```
Changed in Django 1.8:

File现在使用[通用的换行符](https://www.python.org/dev/peps/pep-0278)。以下字符会识别为换行符：Uni
```

`chunks([chunk_size=None])`

迭代整个文件，并生成指定大小的一部分内容。`chunk_size` 默认为64 KB。

处理大文件时这会非常有用，因为这样可以把他们从磁盘中读取出来，而避免将整个文件存到内存中。

`multiple_chunks([chunk_size=None])`

如果文件足够大，需要按照提供的 `chunk_size` 切分成几个部分来访问到所有内容，则返回 `True` 。

`write([content])`

将指定的内容字符串写到文件。取决于底层的储存系统，写入的内容在调用 `close()` 之前可能不会完全提交。

`close()`

关闭文件。

除了这些列出的方法， `File` 暴露了 `file` 对象的以下属性和方法： `encoding` , `fileno` , `flush` , `isatty` , `newlines` , `read` , `readinto` , `readlines` , `seek` , `softspace` , `tell` , `truncate` , `writelines` , `xreadlines` 。

ContentFile 类

`class ContentFile(File)[source]`

`ContentFile` 类继承自 `File` ，但是并不像 `File` 那样，它操作字符串的内容（也支持字节集），而不是一个实际的文件。例如：

```
from __future__ import unicode_literals
from django.core.files.base import ContentFile

f1 = ContentFile("esta sentencia está en español")
f2 = ContentFile(b"these are bytes")
```

ImageFile 类

`class ImageFile(file_object)[source]`

Django特地为图像提供了这个内建类。 `django.core.files.images.ImageFile` 继承了 `File` 的所有属性和方法，并且额外提供了以下的属性：

`width`

图像的像素单位宽度。

`height`

图像的像素单位高度。

附加到对象的文件的额外方法

任何关联到一个对象（比如下面的 `Car.photo` ）的 `File` 都会有一些额外的方法：

`File.save(name, content[, save=True])`

以提供的文件名和内容保存一个新文件。这样不会替换已存在的文件，但是会创建新的文件，并且更新对象来指向它。如果 `save` 为 `True` ，模型的 `save()` 方法会在文件保存之后调用。这就是说，下面两行：

```
>>> car.photo.save('myphoto.jpg', content, save=False)
>>> car.save()
```

等价于：

```
>>> car.photo.save('myphoto.jpg', content, save=True)
```

要注意 `content` 参数必须是 `File` 或者 `File` 的子类的实例，比如 `ContentFile` 。

`File.delete([save=True])`

从模型实例中移除文件，并且删除内部的文件。如果 `save` 是 `True` ，模型的 `save()` 方法会在文件删除之后调用。

> 译者：Django 文档协作翻译小组，原文：File objects。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

文件储存API

获取当前的储存类

Django提供了两个便捷的方法来获取当前的储存类：

`class DefaultStorage[source]`

`DefaultStorage` 提供对当前的默认储存系统的延迟访问，像 `DEFAULT_FILE_STORAGE` 中定义的那样。`DefaultStorage` 内部使用了 `get_storage_class()` 。

`get_storage_class([import_path=None])[source]`

返回实现储存API的类或者模块。

当没有带着 `import_path` 参数调用的时候， `get_storage_class` 会返回当前默认的储存系统，像 `DEFAULT_FILE_STORAGE` 中定义的那样。如果提供了 `import_path` ，`get_storage_class` 会尝试从提供的路径导入类或者模块，并且如果成功的话返回它。如果导入不成功会抛出异常。

FileSystemStorage类

`class FileSystemStorage([location=None, base_url=None, file_permissions_mode=None, directo`

`FileSystemStorage` 类在本地文件系统上实现了基本的文件存储功能。它继承自 `Storage` ，并且提供父类的所有公共方法的实现。

`location`

储存文件的目录的绝对路径。默认为 `MEDIA_ROOT` 设置的值。

`base_url`

在当前位置提供文件储存的URL。默认为 `MEDIA_URL` 设置的值。

`file_permissions_mode`

文件系统的许可，当文件保存时会接收到它。默认为 `FILE_UPLOAD_PERMISSIONS` 。

> New in Django 1.7:
>
> 新增了file_permissions_mode属性。之前，文件总是会接收到FILE_UPLOAD_PERMISSIONS许可。

`directory_permissions_mode`

文件系统的许可，当目录保存时会接收到它。默认为 `FILE_UPLOAD_DIRECTORY_PERMISSIONS` 。

```
New in Django 1.7:

新增了directory_permissions_mode属性。之前，目录总是会接收到FILE_UPLOAD_DIRECTORY_PERMISSIONS许
```

> 注意
>
> `FileSystemStorage.delete()` 在提供的文件名称不存在的时候并不会抛出任何异常。

Storage 类

`class Storage[source]`

`Storage` 类为文件的存储提供了标准化的API，并带有一系列默认行为，所有其它的文件存储系统可以按需继承或者复写它们。

> 注意
>
> 对于返回原生 `datetime` 对象的方法，所使用的有效时区为 `os.environ['TZ']` 的当前值。要注意它总是可以通过Django的 `TIME_ZONE` 来设置。

`accessed_time(name)[source]`

返回包含文件的最后访问时间的原生 `datetime` 对象。对于不能够返回最后访问时间的储存系统，会抛出 `NotImplementedError` 异常。

`created_time(name)[source]`

返回包含文件创建时间的原生 `datetime` 对象。对于不能够返回创建时间的储存系统，会抛出 `NotImplementedError` 异常。

`delete(name)[source]`

删除 `name` 引用的文件。如果目标储存系统不支持删除操作，会抛出 `NotImplementedError` 异常。

`exists(name)[source]`

如果提供的名称所引用的文件在文件系统中存在，则返回 `True`，否则如果这个名称可用于新文件，返回 `False`。

`get_available_name(name, max_length=None)[source]`

返回基于 `name` 参数的文件名称，它在目标储存系统中可用于写入新的内容。

如果提供了 `max_length`，文件名称长度不会超过它。如果不能找到可用的、唯一的文件名称，会抛出 `SuspiciousFileOperation` 异常。

如果 `name` 命名的文件已存在，一个下划线加上随机7个数字或字母的字符串会添加到文件名称的末尾，扩展名之前。

```
Changed in Django 1.7:

之前，下划线和一位数字（比如"_1"，"_2"，以及其他）会添加到文件名称的末尾，直到目标目录中发现了可用的名称。
```

```
Changed in Django 1.8:

新增了max_length参数。
```

`get_valid_name(name)[source]`

返回基于 `name` 参数的文件名称，它适用于目标储存系统。

`listdir(path)[source]`

列出特定目录的所有内容，返回一个包含2元组的列表；第一个元素是目录，第二个是文件。对于不能够提供列表功能的储存系统，抛出 `NotImplementedError` 异常。

`modified_time(name)[source]`

返回包含最后修改时间的原生 `datetime` 对象。对于不能够返回最后修改时间的储存系统，抛出 `NotImplementedError` 异常。

`open(name, mode='rb')[source]`

通过提供的 `name` 打开文件。注意虽然返回的文件确保为 `File` 对象，但可能实际上是它的子类。在远程文件储存的情况下，这意味着读写操作会非常慢，所以警告一下。

`path(name)[source]`

本地文件系统的路径，文件可以用Python标准的 `open()` 在里面打开。对于不能从本地文件系统访问的储存系统，抛出 `NotImplementedError` 异常。

`save(name, content, max_length=None)[source]`

使用储存系统来保存一个新文件，最好带有特定的名称。如果名称为 `name` 的文件已存在，储存系统会按需修改文件名称来获取一个唯一的名称。返回被储存文件的实际名称。

max_length参数会传递给 `get_available_name()` 。

`content` 参数必须为 `django.core.files.File` 或者 `File` 子类的实例。

```
Changed in Django 1.8:

新增了max_length参数。
```

`size(name)[source]`

返回 `name` 所引用的文件的总大小，以字节为单位。对于不能够返回文件大小的储存系统，抛出 `NotImplementedError` 异常。

`url(name)[source]`

返回URL，通过它可以访问到 `name` 所引用的文件。对于不支持通过URL访问的储存系统，抛出 `NotImplementedError` 异常。

> 译者：Django 文档协作翻译小组，原文：Storage API。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

管理文件

这篇文档描述了Django为那些用户上传文件准备的文件访问API。底层的API足够通用，你可以使用为其它目的来使用它们。如果你想要处理静态文件（JS，CSS，以及其他），参见管理静态文件（CSS和图像）。

通常，Django使用 `MEDIA_ROOT` 和 `MEDIA_URL` 设置在本地储存文件。下面的例子假设你使用这些默认值。

然而，Django提供了一些方法来编写自定义的 文件储存系统，允许你完全自定义Django在哪里以及如何储存文件。这篇文档的另一部分描述了这些储存系统如何工作。

在模型中使用文件

当你使用 `FileField` 或者 `ImageField` 的时候，Django为你提供了一系列的API用来处理文件。

考虑下面的模型，它使用 `ImageField` 来储存一张照片：

```
from django.db import models

class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to='cars')
```

任何 `Car` 的实例都有一个 `photo` 字段，你可以通过它来获取附加图片的详细信息：

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: chevy.jpg>
>>> car.photo.name
'cars/chevy.jpg'
>>> car.photo.path
'/media/cars/chevy.jpg'
>>> car.photo.url
'http://media.example.com/cars/chevy.jpg'
```

例子中的 `car.photo` 对象是一个 `File` 对象，这意味着它拥有下面描述的所有方法和属性。

> 注意
>
> 文件保存是数据库模型保存的一部分，所以磁盘上真实的文件名在模型保存之前并不可靠。

例如，你可以通过设置文件的 `name` 属性为一个和文件储存位置（ `MEDIA_ROOT` ，如果你使用默认的 `FileSystemStorage` ）相关的路径，来修改文件名称。

```
>>> import os
>>> from django.conf import settings
>>> initial_path = car.photo.path
>>> car.photo.name = 'cars/chevy_ii.jpg'
>>> new_path = settings.MEDIA_ROOT + car.photo.name
>>> # Move the file on the filesystem
>>> os.rename(initial_path, new_path)
>>> car.save()
>>> car.photo.path
'/media/cars/chevy_ii.jpg'
>>> car.photo.path == new_path
True
```

File

当Django需要表示一个文件的时候，它在内部使用 `django.core.files.File` 实例。这个对象是 Python 内建文件对象的一个简单封装，并带有一些Django特定的附加功能。

大多数情况你可以简单地使用Django提供给你的 `File` 对象（例如像上面那样把文件附加到模型，或者是上传的文件）。

如果你需要自行构造一个 `File` 对象，最简单的方法是使用Python内建的 `file` 对象来创建一个：

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open('/tmp/hello.world', 'w')
>>> myfile = File(f)
```

现在你可以使用 `File` 类的任何文档中记录的属性和方法了。

注意这种方法创建的文件并不会自动关闭。以下步骤可以用于自动关闭文件：

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statement
>>> with open('/tmp/hello.world', 'w') as f:
...     myfile = File(f)
...     myfile.write('Hello World')
...
>>> myfile.closed
True
>>> f.closed
True
```

在处理大量对象的循环中访问文件字段时，关闭文件极其重要。如果文件在访问之后没有手动关闭，会有消耗完文件描述符的风险。这可能导致如下错误：

```
IOError: [Errno 24] Too many open files
```

文件储存

在背后，Django需要决定在哪里以及如何将文件储存到文件系统。这是一个对象，它实际上理解一些东西，比如文件系统，打开和读取文件，以及其他。

Django的默认文件储存由 `DEFAULT_FILE_STORAGE` 设置提供。如果你没有显式提供一个储存系统，就会使用它。

关于内建的默认文件储存系统的细节，请参见下面一节。另外，关于编写你自己的文件储存系统的一些信息，请见编写自定义的文件系统。

储存对象

大多数情况你可能并不想使用 `File` 对象（它向文件提供适当的存储功能），你可以直接使用文件储存系统。你可以创建一些自定义文件储存类的实例，或者 – 大多数情况更加有用的 – 你可以使用全局的默认储存系统：

```
>>> from django.core.files.storage import default_storage
>>> from django.core.files.base import ContentFile

>>> path = default_storage.save('/path/to/file', ContentFile('new content'))
>>> path
'/path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

关于文件储存API，参见 文件储存API。

内建的文件系统储存类

Django自带了 `django.core.files.storage.FileSystemStorage` 类，它实现了基本的本地文件系统中的文件储存。

例如，下面的代码会在 `/media/photos` 目录下储存上传的文件，无论 `MEDIA_ROOT` 设置是什么：

```
from django.db import models
from django.core.files.storage import FileSystemStorage

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

自定义储存系统 以相同方式工作：你可以把它们作为 `storage` 参数传递给 `FileField` 。

> 译者：Django 文档协作翻译小组，原文：Managing files。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

编写自定义存储系统

如果你需要提供自定义文件存储 – 一个普遍的例子是在某个远程系统上储存文件 – 你可以通过定义一个自定义的储存类来实现。你需要遵循以下步骤：

1. 你的自定义储存类必须是 `django.core.files.storage.Storage` 的子类：

```
from django.core.files.storage import Storage

class MyStorage(Storage):
    ...
```

2. Django必须能够不带任何参数来实例化你的储存类。这意味着任何设置都应该从 `django.conf.settings` 中获取。

```
from django.conf import settings
from django.core.files.storage import Storage

class MyStorage(Storage):
    def __init__(self, option=None):
        if not option:
            option = settings.CUSTOM_STORAGE_OPTIONS
        ...
```

3. 你的储存类必须实现 `_open()` 和 `_save()` 方法，以及任何适合于你的储存类的其它方法。更多这类方法请见下文。

另外，如果你的类提供本地文件存储，它必须覆写 `path()` 方法。

4. 你的储存类必须是 可以析构的，所以它在迁移中的一个字段上使用的时候可以被序列化。只要你的字段拥有自己可以序列化的参数，你就可以为它使用 `django.utils.deconstruct.deconstructible` 类装饰器（这也是Django用在 `FileSystemStorage` 上的东西）。

默认情况下，下面的方法会抛出 `NotImplementedError` 异常，并且必须覆写它们。

- Storage.delete()
- Storage.exists()
- Storage.listdir()
- Storage.size()
- Storage.url()

然而要注意，并不是这些方法全部都需要，可以故意省略一些。可以不必实现每个方法而仍然能拥有一个可以工作的储存类。

比如，如果在特定的储存后端中，列出内容的开销比较大，你可以决定不实现 `Storage.listdir` 。

另一个例子是只处理写入文件的后端。这种情况下，你不需要实现上面的任意一种方法。

根本上来说，需要实现哪种方法取决于你。如果不去实现一些方法，你会得到一个不完整（可能是不能用的）的接口。

你也会经常想要使用特意为自定义储存对象设计的钩子。它们是：

`_open(name, mode='rb')`

必需的。

被 `Storage.open()` 调用，这是储存类用于打开文件的实际工具。它必须返回 `File` 对象，在大多数情况下，你会想要返回一些子类，它们实现了后端储存系统特定的逻辑。

`_save(name, content)`

被 `Storage.save()` 调用。 `name` 必须事先通过 `get_valid_name()` 和 `get_available_name()` 过滤，并且 `content` 自己必须是一个 `File` 对象。

应该返回被保存文件的真实名称（通常是传进来的 `name` ，但是如果储存需要修改文件名称，则返回新的名称来代替）。

`get_valid_name(name)`

返回适用于当前储存系统的文件名。传递给该方法的 `name` 参数是发送给服务器的原始文件名称，并移除了所有目录信息。你可以覆写这个方法，来自定义非标准的字符将会如何转换为安全的文件名称。

`Storage` 提供的代码只会保留原始文件名中的数字和字母字符、英文句号和下划线，并移除其它字符。

`get_available_name(name, max_length=None)`

返回在储存系统中可用的文件名称，可能会顾及到提供的文件名称。传给这个方法的name参数需要事先过滤为储存系统有效的文件名称，根据上面描述的 `get_valid_name()` 方法。

如果提供了 `max_length` ，文件名称长度不会超过它。如果不能找到可用的、唯一的文件名称，会抛出 `SuspiciousFileOperation` 异常。

如果 `name` 命名的文件已存在，一个下划线加上随机7个数字或字母的字符串会添加到文件名称的末尾，扩展名之前。

```
Changed in Django 1.7:

之前，下划线和一位数字（比如"_1", "_2"，以及其他）会添加到文件名称的末尾，直到目标目录中发现了可用的名称。
```

```
Changed in Django 1.8:

新增了max_length参数。
```

自定义储存系统 以相同方式工作：你可以把它们作为 `storage` 参数传递给 `FileField` 。

> 译者：Django 文档协作翻译小组，原文：Custom storage。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的视图

基于类的视图

视图是一个可调用对象，它接收一个请求然后返回一个响应。这个可调用对象可以不只是函数，Django 提供一些可以用作视图的类。它们允许你结构化你的视图并且利用继承和混合重用代码。后面我们将介绍一些用于简单任务的通用视图，但你可能想要设计自己的可重用视图的结构以适合你的使用场景。完整的细节，请参见基于类的视图的参考文档。

基本的示例

Django 提供基本的视图类，它们适用于广泛的应用。所有的视图类继承自 `View` 类，它负责连接视图到URL、HTTP 方法调度和其它简单的功能。 `RedirectView` 用于简单的HTTP 重定向， `TemplateView` 扩展基类来渲染模板。

在URLconf 中的简单用法

使用通用视图最简单的方法是在URLconf 中创建它们。如果你只是修改基于类的视图的一些简单属性，你可以将它们直接传递给 `as_view()` 方法调用：

```
from django.conf.urls import url
from django.views.generic import TemplateView

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

传递给 `as_view()` 的参数将覆盖类中的属性。在这个例子中，我们设置 `TemplateView` 的 `template_name` 。可以使用类似的方法覆盖 `RedirectView` 的 `url` 属性。

子类化通用视图

第二种，功能更强一点的使用通用视图的方式是继承一个已经存在的视图并在子类中覆盖其属性（例如 `template_name` ）或方法（例如 `get_context_data` ）以提供新的值或方法。例如，考虑只显示一个模板 `about.html` 的视图。Django 有一个通用视图 `TemplateView` 来做这件事，所以我们可以简单地子类化它，并覆盖模板的名称：

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

然后我们只需要添加这个新的视图到我们的URLconf 中。`TemplateView`是一个类不是一个函数，所以我们将URL 指向类的`as_view()`方法，它让基于类的视图提供一个类似函数的入口：

```
# urls.py
from django.conf.urls import url
from some_app.views import AboutView

urlpatterns = [
    url(r'^about/', AboutView.as_view()),
]
```

关于如何使用内建的通用视图的更多信息，参考下一主题通用的基于类的视图。

支持其它HTTP 方法

假设有人想通过HTTP 访问我们的书库，它使用视图作为API。这个API 客户端将随时连接并下载自上次访问以来新出版的书籍的数据。如果没有新的书籍，仍然从数据库中获取书籍、渲染一个完整的响应并发送给客户端将是对CPU 和带宽的浪费。如果有个API 用于查询书籍最新发布的时间将会更好。

我们在URLconf 中映射URL 到书籍列表视图：

```
from django.conf.urls import url
from books.views import BookListView

urlpatterns = [
    url(r'^books/$', BookListView.as_view()),
]
```

下面是这个视图：

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest('publication_date')
        response = HttpResponse('')
        # RFC 1123 date format
        response['Last-Modified'] = last_book.publication_date.strftime('%a, %d %b %Y %H:
        return response
```

如果该视图从GET 请求访问，将在响应中返回一个普通而简单的对象列表（使用 `book_list.html` 模板）。但如果客户端发出一个 `HEAD` 请求，响应将具有一个空的响应体而 `Last-Modified` 头部会指示最新发布的书籍的时间。基于这个信息，客户端可以下载或不下载完整的对象列表。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的内建通用视图

编写Web应用可能是单调的，因为你需要不断的重复某一种模式。 Django尝试从model和template层移除一些单调的情况，但是Web开发者依然会在view（视图）层经历这种厌烦。

Django的通用视图被开发用来消除这一痛苦。它们采用某些常见的习语和在开发过 程中发现的模式然后把它们抽象出来，以便你能够写更少的代码快速的实现基础的视图。

我们能够识别一些基础的任务，比如展示对象的列表，以及编写代码来展示任何对象的 列表。此外，有问题的模型可以作为一个额外的参数传递到URLconf中。

Django通过通用视图来完成下面一些功能：

- 为单一的对象展示列表和一个详细页面。 如果我们创建一个应用来管理会议，那么 一个TalkListView (讨论列表视图)和一个 RegisteredUserListView （ 注册用户列表视图）就是列表视图的一个例子。一个单独的讨论信息页面就是我们称 之为 "详细" 视图的例子。
- 在年/月/日归档页面，以及详细页面和“最后发表”页面中，展示以数据库为基础的对象。
- 允许用户创建，更新和删除对象 -- 以授权或者无需授权的方式。

总的来说，这些视图提供了一些简单的接口来完成开发者遇到的大多数的常见任务。

扩展通用视图

使用通用视图可以极大的提高开发速度，是毫无疑问的。 然而在大多数工程中， 总会遇到通用视图无法满足需求的时候。的确，大多数来自Django开发新手 的问题是如何能使得通用视图的使用范围更广。

这是通用视图在1.3发布中被重新设计的原因之一 - 之前，它们仅仅是一些函数视图加上 一列令人疑惑的选项；现在，比起传递大量的配置到URLconf中，更推荐的扩展通用视图的 方法是子类化它们，并且重写它们的属性或者方法。

这就是说，通用视图有一些限制。如果你将你的视图实现为通用视图的子类，你就会发现这样能够更有效地编写你想要的代码，使用你自己的基于类或功能的视图。

在一些三方的应用中，有更多通用视图的示例，或者你可以自己按需编写。

对象的通用视图

TemplateView确实很有用，但是当你需要 呈现你数据库中的内容时Django的通用视图才真的会脱颖而出。因为这是如此常见 的任务，Django提供了一大把内置的通用视图，使生成对象的展示列表和详细视图 的变得极其容易。

让我们来看一下这些通用视图中的"对象列表"视图。

我们将使用下面的模型：

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

现在我们需要定义一个视图：

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

最后将视图解析到你的url上：

```
# urls.py
from django.conf.urls import url
from books.views import PublisherList

urlpatterns = [
    url(r'^publishers/$', PublisherList.as_view()),
]
```

上面就是所有我们需要写的Python代码了。

> 注意
>
> 所以，当（例如）DjangoTemplates后端的APP_DIRS选项在TEMPLATES中设置为True时，模板的位置应该为：/path/to/project/books/templates/books/publisher_list.html。

这个模板将会依据于一个上下文(context)来渲染，这个context包含一个名为object_list 包含所有publisher对象的变量。一个非常简单的模板可能看起来像下面这样：

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

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

这确实就是全部代码了。 所有通用视图中有趣的特性来自于修改被传递到通用视图中的"信息" 字典。generic views reference文档详细 介绍了通用视图以及它的选项；本篇文档剩余的部分将会介绍自定义以及扩展通用 视图的常见方法。

编写“友好的”模板上下文

你可能已经注意到了，我们在publisher列表的例子中把所有的publisher对象 放到 object_list 变量中。虽然这能正常工作，但这对模板作者并不是 "友好的"。他们只需要知道在这里要处理publishers就行了。

因此，如果你在处理一个模型(model)对象，这对你来说已经足够了。 当你处理 一个object或者queryset时，Django能够使用你定义对象显示用的自述名（verbose name，或者复数的自述名，对于对象列表）来填充上下文(context)。提供添加到默认的 object_list 实体中，但是包含完全相同的数据，例如publisher_list。

如果自述名(或者复数的自述名) 仍然不能很好的符合要求，你 可以手动的设置上下文(context)变量的名字。在一个通用视图上的context_object_name属性指定了要使用的定了上下文变量：

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favorite_publishers'
```

提供一个有用的context_object_name总是个好主意。和你一起工作的设计 模板的同事会感谢你的。

添加额外的上下文

多数时候，你只是需要展示一些额外的信息而不是提供一些通用视图。 比如，考虑到每个publisher 详细页面上的图书列表的展示。DetailView通用视图提供了一个publisher对象给context，但是我们如何在模板中添加附加信息呢？

答案是派生DetailView，并且在get_context_data方法中提供你自己的实现。默认的实现只是简单的 给模板添加了要展示的对象，但是你这可以这样覆写来展示更多信息：

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

> 注意
>
> 通常来说，get_context_data会将当前类中的上下文数据，合并到所有超类中的上下文数据。要在你自己想要改变上下文的类中保持这一行为，你应该确保在超类中调用了get_context_data。如果没有任意两个类尝试定义相同的键，会返回异常的结果。然而，如果任何一个类尝试在超类持有一个键的情况下覆写它（在调用超类之后），这个类的任何子类都需要显式于超类之后设置它，如果你想要确保他们覆写了所有超类的话。如果你有这个麻烦，复查你视图中的方法调用顺序。

查看对象的子集

现在让我们来近距离查看下我们一直在用的 model参数。model参数指定了视图在哪个数据库模型之上进行操作，这适用于所有的需要 操作一个单独的对象或者一个对象集合的通用视图。然而，model参数并不是唯一能够指明视图要基于哪个对象进行操作的方法 -- 你同样可以使用queryset参数来指定一个对象列表：

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

指定model = Publisher等价于快速声明的queryset = Publisher.objects.all()。然而，通过使用queryset来定义一个过滤的对象列表，你可以更加详细 的了解哪些对象将会被显示的视图中(参见执行查询来获取更多关于查询集对象的更对信息，以及参见 基于类的视图参考来获取全部 细节)。

我们可能想要对图书列表按照出版日期进行排序来选择一个简单的例子，并且把 最近的放到前面：

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

这是个非常简单的列子，但是它很好的诠释了处理思路。 当然，你通常想做的不仅仅只是 对对象列表进行排序。如果你想要展现某个出版商的所有图书列表，你可以使用 同样的手法：

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='Acme Publishing')
    template_name = 'books/acme_list.html'
```

注意，除了经过过滤之后的查询集，一起定义的还有我们自定义的模板名称。如果我们不这么做，通过视图会使用和 "vanilla" 对象列表名称一样的模板，这可 能不是我们想要的。

另外需要注意，这并不是处理特定出版商的图书的非常优雅的方法。 如果我们 要创建另外一个出版商页面，我们需要添加另外几行代码到URLconf中，并且再多几个 出版商就会觉得这么做不合理。我们会在下一个章节处理这个问题。

> 注意
>
> 如果你在访问 /books/acme/时出现404错误，检查确保你确实有一个名字为“ACME Publishing”的出版商。通用视图在这种情况下拥有一个allow_empty 的参数。详见基于类的视图参考。

动态过滤

另一个普遍的需求是在给定的列表页面中根据URL中的关键字来过滤对象。 前面我们把出版商的名字硬编码到URLconf中，但是如果我们想要编写一个视图来展示任何publisher的所有图书，应该如何处理？

相当方便的是， ListView 有一个get_queryset() 方法来供我们重写。在之前，它只是返回一个queryset属性值，但是现在我们可以添加更多的逻辑。

让这种方式能够工作的关键点，在于当类视图被调用时，各种有用的对象被存储在self上；同request()(self.request)一样，其中包含了从URLconf中获取到的位置参数 (self.args)和基于名字的参数(self.kwargs)(关键字参数)。

这里，我们拥有一个带有一组供捕获的参数的URLconf：

```
# urls.py
from django.conf.urls import url
from books.views import PublisherBookList

urlpatterns = [
    url(r'^books/([\w-]+)/$', PublisherBookList.as_view()),
]
```

接着，我们编写了PublisherBookList视图::

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

如你所见，在queryset区域添加更多的逻辑非常容易；如果我们想的话，我们可以 使用self.request.user来过滤当前用户，或者添加其他更复杂的逻辑。

同时我们可以把出版商添加到上下文中，这样我们就可以在模板中使用它：

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**kwargs)
    # Add in the publisher
    context['publisher'] = self.publisher
    return context
```

执行额外的工作

我们需要考虑的最后的共同模式在调用通用视图之前或者之后会引起额外的开销。

想象一下，在我们的Author对象上有一个last_accessed字段，这个字段用来 跟踪某人最后一次查看了这个作者的时间。

```
# models.py
from django.db import models

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

通用的DetailView类，当然不知道关于这个字段的事情，但我们可以很容易 再次编写一个自定义的视图，来保持这个字段的更新。

首先，我们需要添加作者详情页的代码配置到URLconf中，指向自定义的视图：

```
from django.conf.urls import url
from books.views import AuthorDetailView

urlpatterns = [
    #...
    url(r'^authors/(?P<pk>[0-9]+)/$', AuthorDetailView.as_view(), name='author-detail'),
]
```

然后，编写我们新的视图 -- get_object是用来获取对象的方法 -- 因此我们简单的 重写它并封装调用：

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

> 注意
>
> 这里URLconf使用参数组的名字pk - 这个名字是DetailView用来查找主键的值的默认名称，其中主键用于过滤查询集。
>
> 如果你想要调用参数组的其它方法，你可以在视图上设置pk_url_kwarg。详见DetailView参考。

使用基于类的视图处理表单

表单的处理通常有3 个步骤：

- 初始的的GET （空白或预填充的表单）
- 带有非法数据的POST（通常重新显示表单和错误信息）
- 带有合法数据的POST（处理数据并重定向）

你自己实现这些功能经常导致许多重复的样本代码（参见在视图中使用表单）。为了避免这点，Django 提供一系列的通用的基于类的视图用于表单的处理。

基本的表单

根据一个简单的联系人表单：

```
#forms.py

from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

可以使用 `FormView` 来构造其视图：

```
#views.py

from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

注：

- `FormView` 继承 `TemplateResponseMixin` 所以这里可以使用 `template_name` 。
- `form_valid()` 的默认实现只是简单地重定向到 `success_url` 。

模型的表单

通用视图在于模型一起工作时会真正光芒四射。这些通用的视图将自动创建一个 ModelForm，只要它们能知道使用哪一个模型类：

- 如果给出 `model` 属性，则使用该模型类。
- 如果 `get_object()` 返回一个对象，则使用该对象的类。
- 如果给出 `queryset`，则使用该查询集的模型。

模型表单提供一个 `form_valid()` 的实现，它自动保存模型。如果你有特殊的需求，可以覆盖它；参见下面的例子。

你甚至不需要为 `CreateView` 和 `UpdateView` 提供 `success_url` —— 如果存在它们将使用模型对象的 `get_absolute_url()`。

如果你想使用一个自定义的 `ModelForm`（例如添加额外的验证），只需简单地在你的视图上设置 `form_class`。

> 注
>
> 当指定一个自定义的表单类时，你必须指定模型，即使 `form_class` 可能是一个 `ModelForm`。

首先我们需要添加 `get_absolute_url()` 到我们的 `Author` 类中：

```
#models.py

from django.core.urlresolvers import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

然后我们可以使用 `CreateView` 机器伙伴来做实际的工作。注意这里我们是如何配置通用的基于类的视图的；我们自己没有写任何逻辑：

```
#views.py

from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

> 注
>
> 这里我们必须使用 `reverse_lazy()` 而不是 `reverse`，因为在该文件导入时URL 还没有加载。

`fields` 属性的工作方式与 `ModelForm` 的内部 `Meta` 类的 `fields` 属性相同。除非你用另外一种方式定义表单类，该属性是必须的，如果没有将引发一个 `ImproperlyConfigured` 异常。

如果你同时指定 `fields` 和 `form_class` 属性，将引发一个 `ImproperlyConfigured` 异常。

```
Changed in Django 1.8:

省略fields 属性在以前是允许的，但是导致表单带有模型的所有字段。
```

```
Changed in Django 1.8:

以前，如果fields 和form_class 两个都指定，会默默地忽略 fields。
```

最后，我我们来将这些新的视图放到URLconf 中：

```
#urls.py

from django.conf.urls import url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = [
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author_add'),
    url(r'author/(?P<pk>[0-9]+)/$', AuthorUpdate.as_view(), name='author_update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$', AuthorDelete.as_view(), name='author_delete'),
]
```

> 注
>
> 这些表单继承 `SingleObjectTemplateResponseMixin`，它使用 `template_name_suffix` 并基于模型来构造 `template_name`。
>
> 在这个例子中：
>
> - `CreateView` 和 `UpdateView` 使用 `myapp/author_form.html`
> - `DeleteView` 使用 `myapp/author_confirm_delete.html`
>
> 如果你希望分开 `CreateView` 和 `UpdateView` 的模板，你可以设置你的视图类的 `template_name` 或 `template_name_suffix`。

模型和request.user

为了跟踪使用CreateView 创建一个对象的用户，你可以使用一个自定义的ModelForm 来实现这点。首先，向模型添加外键关联：

```
#models.py

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

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

    # ...
```

在这个视图中，请确保你没有将 `created_by` 包含进要编辑的字段列表，并覆盖 `form_valid()` 来添加这个用户：

```
#views.py

from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

注意，你需要使用 `login_required()` 来装饰这个视图，或者在 `form_valid()` 中处理未认证的用户。

AJAX 示例

下面是一个简单的实例，展示你可以如何实现一个表单，使它可以同时为AJAX 请求和'普通的'表单POST 工作：

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    """
    Mixin to add AJAX support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """
    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_invalid(form)
        if self.request.is_ajax():
            return JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super(AjaxableResponseMixin, self).form_valid(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return JsonResponse(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

> 译者：Django 文档协作翻译小组，原文：Built-in editing views。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

内建基于类的视图的API

基于类的视图的API 参考。另请参见基于类的视图 的简介。

说明

由基于类的视图处理的每个请求都具有一个独立的状态；所以，在实例中保存状态变量是安全的（例如， `self.foo = 3` 是线程安全的操作）。

基于类的视图在URL 模式中的部署使用 `as_view()` 类方法：

```
urlpatterns = [
    url(r'^view/$', MyView.as_view(size=42)),
]
```

> 视图参数的线程安全性
>
> 传递给视图的参数在视图的每个实例之间共享。这表示不应该使用列表、字典或其它可变对象作为视图的参数。如果你真这么做而且对共享的对象做过修改，某个用户的行为可能对后面访问同一个视图的用户产生影响。

传递给 `as_view()` 的参数将赋值给服务请求的实例。利用前面的例子，这表示对 `MyView` 的每个请求都可以使用 `self.size` 。参数必须对应于在类中已经存在的属性（ `hasattr` 检查可以返回 `True` ）。

基础视图 VS. 通用视图

基于类的基础视图可以认为是父视图，它们可以直接使用或者继承它们。它们不能满足项目中所有的需求，在这种情况下有Mixin 可以扩展基础视图的功能。

Django 的通用视图建立在基础视图之上，用于作为经常用到的功能的快捷方式，例如显示对象的详细信息。它们提炼视图开发中常见的风格和模式并将它们抽象，这样你可以快速编写常见的视图而不用重复你自己。

大部分通常视图需要 `queryset` 键，它是一个 `查询集` 实例；关于 `查询集` 对象的更多信息，请参见执行查询。

> 译者：Django 文档协作翻译小组，原文：API reference。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

基于类的通用视图 —— 索引

这里的索引提供基于类的视图的另外一种组织形式。对于每个视图，在类继承树中有效的属性和方法都显示在该视图的下方。按照行为进行组织的文档，参见基于类的视图。

简单的通用视图

View

`class View`

属性 （以及访问它们的方法）：

- `http_method_names`

方法

- `as_view()`
- `dispatch()`
- `head()`
- `http_method_not_allowed()`

TemplateView

`class TemplateView`

属性 （以及访问它们的方法）：

- `content_type`
- `http_method_names`
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `head()`
- `http_method_not_allowed()`

- `render_to_response()`

RedirectView

`class RedirectView`

属性 （以及访问它们的方法）：

- `http_method_names`
- `pattern_name`
- `permanent`
- `query_string`
- `url` [`get_redirect_url()`]

方法

- `as_view()`
- `delete()`
- `dispatch()`
- `get()`
- `head()`
- `http_method_not_allowed()`
- `options()`
- `post()`
- `put()`

明细视图

DetailView

`class DetailView`

属性 （以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`

- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `render_to_response()`

清单视图

ListView

`class ListView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`

- `get()`
- `get_context_data()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

编辑视图

FormView

`class FormView`

属性 （以及访问它们的方法）：

- `content_type`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `prefix` [`get_prefix()`]
- `response_class` [`render_to_response()`]
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `http_method_not_allowed()`
- `post()`
- `put()`

CreateView

`class CreateView`

属性 （以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `fields`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `model`
- `pk_url_kwarg`
- `prefix` [`get_prefix()`]
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `post()`
- `put()`
- `render_to_response()`

UpdateView

`class UpdateView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `fields`
- `form_class` [`get_form_class()`]
- `http_method_names`
- `initial` [`get_initial()`]
- `model`
- `pk_url_kwarg`
- `prefix` [`get_prefix()`]
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `form_invalid()`
- `form_valid()`
- `get()`
- `get_context_data()`
- `get_form()`
- `get_form_kwargs()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `post()`
- `put()`
- `render_to_response()`

DeleteView

`class DeleteView`

属性（以及访问它们的方法）：

- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `http_method_names`
- `model`
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `success_url` [`get_success_url()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`

方法

- `as_view()`
- `delete()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_object()`
- `head()`
- `http_method_not_allowed()`
- `post()`
- `render_to_response()`

基于日期的视图

ArchiveIndexView

`class ArchiveIndexView`

属性（以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]

- `date_field` [`get_date_field()`]
- `http_method_names`
- `model`
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

YearArchiveView

`class YearArchiveView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `make_object_list` [`get_make_object_list()`]
- `model`

- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

MonthArchiveView

`class MonthArchiveView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]

- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

WeekArchiveView

`class WeekArchiveView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `http_method_names`
- `model`

- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `week` [`get_week()`]
- `week_format` [`get_week_format()`]
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_paginator()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

DayArchiveView

`class DayArchiveView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]
- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]

- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_day()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

TodayArchiveView

`class TodayArchiveView`

属性 （以及访问它们的方法）：

- `allow_empty` [`get_allow_empty()`]

- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]
- `ordering` [`get_ordering()`]
- `paginate_by` [`get_paginate_by()`]
- `paginate_orphans` [`get_paginate_orphans()`]
- `paginator_class`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_date_list()`
- `get_dated_items()`
- `get_dated_queryset()`
- `get_next_day()`
- `get_next_month()`
- `get_paginator()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `paginate_queryset()`
- `render_to_response()`

DateDetailView

`class DateDetailView`

属性 （以及访问它们的方法）：

- `allow_future` [`get_allow_future()`]
- `content_type`
- `context_object_name` [`get_context_object_name()`]
- `date_field` [`get_date_field()`]
- `day` [`get_day()`]
- `day_format` [`get_day_format()`]
- `http_method_names`
- `model`
- `month` [`get_month()`]
- `month_format` [`get_month_format()`]
- `pk_url_kwarg`
- `queryset` [`get_queryset()`]
- `response_class` [`render_to_response()`]
- `slug_field` [`get_slug_field()`]
- `slug_url_kwarg`
- `template_engine`
- `template_name` [`get_template_names()`]
- `template_name_field`
- `template_name_suffix`
- `year` [`get_year()`]
- `year_format` [`get_year_format()`]

方法

- `as_view()`
- `dispatch()`
- `get()`
- `get_context_data()`
- `get_next_day()`
- `get_next_month()`
- `get_object()`
- `get_previous_day()`
- `get_previous_month()`
- `head()`
- `http_method_not_allowed()`
- `render_to_response()`

> 译者：Django 文档协作翻译小组，原文：Flattened index。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级

使用Django输出CSV

这篇文档阐述了如何通过使用Django视图动态输出CSV (Comma Separated Values)。 你可以使用Python CSV 库或者Django的模板系统来达到目的。

使用Python CSV库

Python自带了CSV库， `csv` 。在Django中使用它的关键是， `csv` 模块的CSV创建功能作用于类似于文件的对象，并且Django的 `HttpResponse` 对象就是类似于文件的对象。

这里是个例子：

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"])

    return response
```

代码和注释是不用多说的，但是一些事情需要提醒一下：

- 响应对象获得了一个特殊的MIME类型， `text/csv` 。这会告诉浏览器，文档是个CSV文件而不是HTML文件。如果你把它去掉，浏览器可能会把输出解释为HTML，会在浏览器窗口中显示一篇丑陋的、可怕的官样文章。
- 响应对象获取了附加的 `Content-Disposition` 协议头，它含有CSV文件的名称。文件名可以是任意的；你想把它叫做什么都可以。浏览器会在”另存为“对话框中使用它，或者其它。
- 钩住CSV生成API非常简单：只需要把 `response` 作为第一个参数传递给 `csv.writer` 。 `csv.writer` 函数接受一个类似于文件的对象，而 `HttpResponse` 对象正好合适。
- 对于你CSV文件的每一行，调用 `writer.writerow` ，向它传递一个可迭代的对象比如列表或者元组。
- CSV模板会为你处理引用，所以你不用担心没有转义字符串中的引号或者逗号。只需要向 `writerow()` 传递你的原始字符串，它就会执行正确的操作。

> 在Python 2中处理Unicode
>
> Python2的 `csv` 模块不支持Unicode输入。由于Django在内部使用Unicode，这意味着从一些来源比如 `HttpRequest` 读出来的字符串可能导致潜在的问题。有一些选项用于处理它：
>
> - 手动将所有Unicode对象编码为兼容的编码。
> - 使用csv模块示例章节中提供的 `UnicodeWriter` 类。
> - 使用python-unicodecsv 模块，它作为 `csv` 模块随时可用的替代方案，能够优雅地处理Unicode。
>
> 更多信息请见 `csv` 模块的Python文档。

流式传输大尺寸**CSV**文件

当处理生成大尺寸响应的视图时，你可能想要使用Django的 `StreamingHttpResponse` 类。例如，通过流式传输需要长时间来生成的文件，可以避免负载均衡器在服务器生成响应的时候断掉连接。

在这个例子中，我们利用Python的生成器来有效处理大尺寸CSV文件的拼接和传输：

```
import csv

from django.utils.six.moves import range
from django.http import StreamingHttpResponse

class Echo(object):
    """An object that implements just the write method of the file-like
    interface.
    """
    def write(self, value):
        """Write the value by returning it, instead of storing in a buffer."""
        return value

def some_streaming_csv_view(request):
    """A view that streams a large CSV file."""
    # Generate a sequence of rows. The range is based on the maximum number of
    # rows that can be handled by a single sheet in most spreadsheet
    # applications.
    rows = (["Row {}".format(idx), str(idx)] for idx in range(65536))
    pseudo_buffer = Echo()
    writer = csv.writer(pseudo_buffer)
    response = StreamingHttpResponse((writer.writerow(row) for row in rows),
                                     content_type="text/csv")
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'
    return response
```

使用模板系统

或者，你可以使用Django模板系统来生成CSV。比起便捷的Python `csv` 模板来说，这样比较低级，但是为了完整性，这个解决方案还是在这里展示一下。

它的想法是，传递一个项目的列表给你的模板，并且让模板在 `for` 循环中输出逗号。

这里是一个例子，它像上面一样生成相同的CSV文件：

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    # The data is hard-coded here, but you could load it from a database or
    # some other source.
    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({
        'data': csv_data,
    })
    response.write(t.render(c))
    return response
```

这个例子和上一个例子之间唯一的不同就是，这个例子使用模板来加载，而不是CSV模块。代码的结果 -- 比如 `content_type='text/csv'` -- 都是相同的。

然后，创建模板 `my_template_name.txt`，带有以下模板代码：

```
{% for row in data %}"{{ row.0|addslashes }}", "{{ row.1|addslashes }}", "{{ row.2|addsla
{% endfor %}
```

这个模板十分基础。它仅仅遍历了提供的数据，并且对于每一行都展示了一行CSV。它使用了 `addslashes` 模板过滤器来确保没有任何引用上的问题。

其它基于文本的格式

要注意对于 CSV来说，这里并没有什么特别之处 -- 只是特定了输出格式。你可以使用这些技巧中的任何一个，来输出任何你想要的，基于文本的格式。你也可以使用相似的技巧来生成任意的二进制数据。例子请参见在Django中输出PDF。

> 译者：Django 文档协作翻译小组，原文：Generating CSV。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

使用Django输出PDF

这篇文档阐述了如何通过使用Django视图动态输出PDF。这可以通过一个出色的、开源的Python PDF库ReportLab来实现。

动态生成PDF文件的优点是，你可以为不同目的创建自定义的PDF -- 这就是说，为不同的用户或者不同的内容。

例如，Django在kusports.com上用来为那些参加March Madness比赛的人，生成自定义的，便于打印的 NCAA 锦标赛晋级表作为PDF文件。

安装ReportLab

ReportLab库在PyPI上提供。也可以下载到用户指南 （PDF文件，不是巧合）。 你可以使用 `pip` 来安装ReportLab：

```
$ pip install reportlab
```

通过在Python交互解释器中导入它来测试你的安装：

```
>>> import reportlab
```

若没有抛出任何错误，则已安装成功。

编写你的视图

使用Django动态生成PDF的关键是，ReportLab API作用于类似于文件的对象，并且Django的 `HttpResponse` 对象就是类似于文件的对象。

这里是一个 “Hello World”的例子：

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="somefilename.pdf"'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

代码和注释是不用多说的，但是一些事情需要提醒一下：

- 响应对象获得了一个特殊的MIME类型， `application/pdf` 。这会告诉浏览器，文档是个PDF文件而不是HTML文件。 如果你把它去掉，浏览器可能会把输出解释为HTML，会在浏览器窗口中显示一篇丑陋的、可怕的官样文章。
- 响应对象获取了附加的 `Content-Disposition` 协议头，它含有PDF文件的名称。 文件名可以是任意的；你想把它叫做什么都可以。浏览器会在”另存为“对话框中使用它，或者其它。
- 在这个例子中， `Content-Disposition` 协议头以 `'attachment;'` 开头。 这样就强制让浏览器弹出对话框来提示或者确认，如果机器上设置了默认值要如何处理文档。如果你去掉了 `'attachment;'` ，无论什么程序或控件被设置为用于处理PDF，浏览器都会使用它。代码就像这样：

```
response['Content-Disposition'] = 'filename="somefilename.pdf"'
```

- 钩住ReportLab API 非常简单：只需要向 `canvas.Canvas` 传递 `response` 作为第一个参数。 `Canvas` 函数接受一个类似于文件的对象，而 `HttpResponse` 对象正好合适。
- 注意所有随后的PDF生成方法都在PDF对象（这个例子是p）上调用，而不是 `response` 对象上。
- 最后，在PDF文件上调用 `showPage()` 和 `save()` 非常重要。

> 注意
>
> ReportLab并不是线程安全的。一些用户报告了一些奇怪的问题，在构建生成PDF的Django视图时出现，这些视图在同一时间被很多人访问。

复杂的PDF

如果你使用ReportLab创建复杂的PDF文档，考虑使用 `io` 库作为你PDF文件的临时保存地点。这个库提供了一个类似于文件的对象接口，非常实用。这个是上面的“Hello World”示例采用 `io` 重写后的样子：

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="somefilename.pdf"'

    buffer = BytesIO()

    # Create the PDF object, using the BytesIO object as its "file."
    p = canvas.Canvas(buffer)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the BytesIO buffer and write it to the response.
    pdf = buffer.getvalue()
    buffer.close()
    response.write(pdf)
    return response
```

更多资源

- PDFlib与Python捆绑的另一个PDF生成库。在Django中使用它的方法和这篇文章所阐述的相同。
- Pisa XHTML2PDF是另一个PDF生成库。Pisa自带了如何将 Pisa 集成到 Django的例子。
- HTMLdoc是一个命令行脚本，它可以把HTML转换为PDF。它并没有Python接口，但是你可以使用 `system` 或者 `popen`，在控制台中使用它，然后再Python中取回输出。

其它格式

要注意在这些例子中并没有很多PDF特定的东西 -- 只是使用了 `reportlab`。你可以使用相似的技巧来生成任何格式，只要你可以找到对应的Python库。关于用于生成基于文本的格式的其它例子和技巧，另见使用Django输出CSV。

> 译者：Django 文档协作翻译小组，原文：Generating PDF。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

中间件

中间件

中间件是一个介入Django的请求和响应的处理过程中的钩子框架。它是一个轻量级，底层的“插件”系统，用于在全局修改Django的输入或输出。

中间件组件责任处理某些特殊的功能。例如，Django包含一个中间件组件，AuthenticationMiddleware ，使用会话将用户和请求关联。

这篇文档讲解了中间件如何工作，如何激活中间件，以及如何编写自己的中间件。Django集成了一些内置的中间件可以直接开箱即用。它们被归档在 内置中间件参考.

激活中间件

要激活一个中间件组件，需要把它添加到你Django配置文件中的MIDDLEWARE_CLASSES列表中。

在MIDDLEWARE_CLASSES中，每一个中间件组件用字符串的方式描述：一个完整的Python全路径加上中间件的类名称。例如，使用 django-admin startproject创建工程的时候生成的默认值：

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
)
```

Django的程序中，中间件不是必需的 — 只要你喜欢，MIDDLEWARE_CLASSES可以为空 — 但是强烈推荐你至少使用CommonMiddleware。

MIDDLEWARE_CLASSES中的顺序非常重要，因为一个中间件可能依赖于另外一个。例如，AuthenticationMiddleware在会话中储存已认证的用户。所以它必须在SessionMiddleware之后运行。一些关于Django中间件类的顺序的常见提示，请见Middleware ordering。

钩子和应用顺序

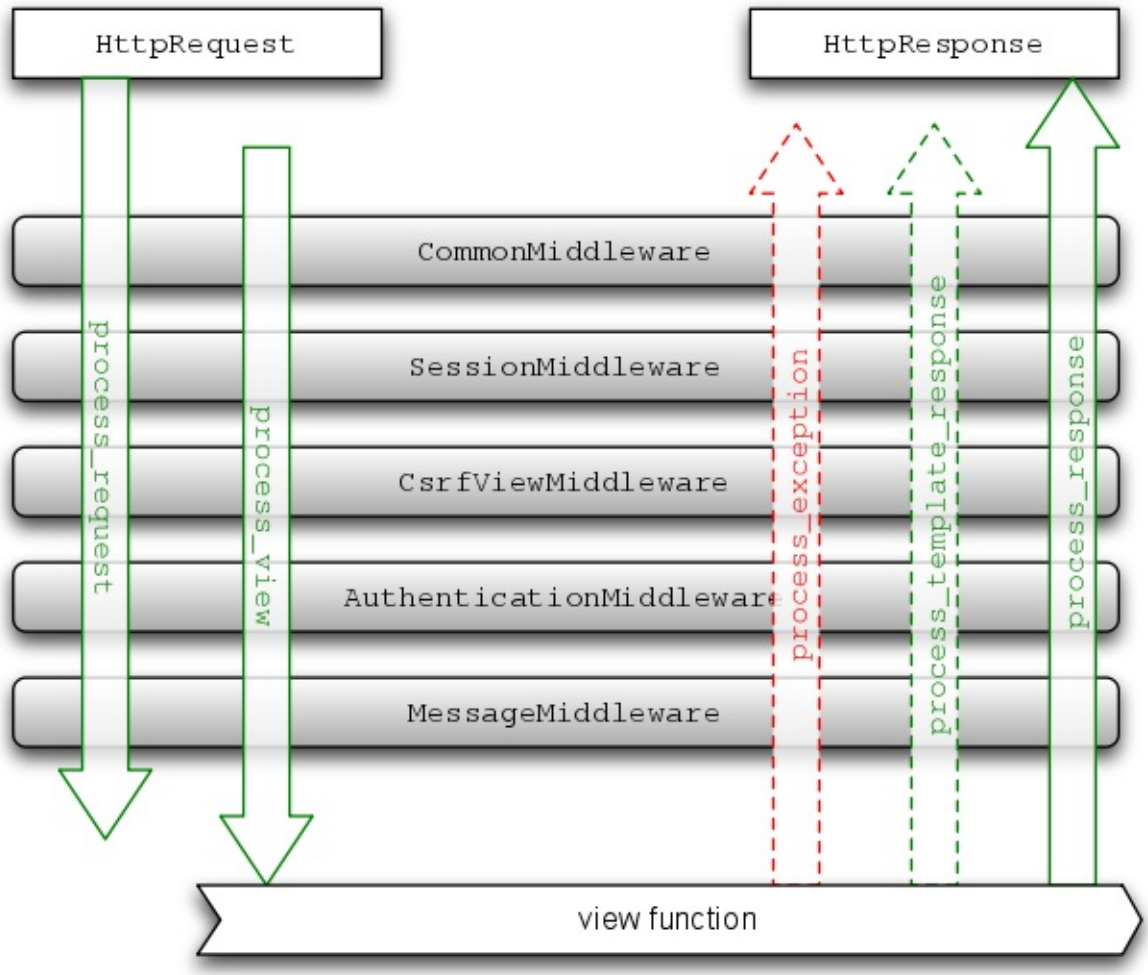
在请求阶段中，调用视图之前，Django会按照MIDDLEWARE_CLASSES中定义的顺序自顶向下应用中间件。会用到两个钩子：

- process_request()

- process_view()

在响应阶段中，调用视图之后，中间件会按照相反的顺序应用，自底向上。会用到三个钩子：

- process_exception() （仅当视图抛出异常的时候）
- process_template_response() （仅用于模板响应）
- process_response()



如果你愿意的话，你可以把它想象成一颗洋葱：每个中间件都是包裹视图的一层“皮”。

每个钩子的行为接下来会描述。

编写自己的中间件

编写自己的中间件很容易的。每个中间件组件是一个单独的Python的class，你可以定一个或多个下面的这些方法：

process_request

process_request(request)

request是一个HttpRequest 对象。

在Django决定执行哪个视图(view)之前，process_request()会被每次请求调用。

它应该返回一个None 或一个HttpResponse对象。如果返回 None, Django会继续处理这个请求，执行其他process_request()中间件，然后process_view()中间件显示对应的视图。如果它返回一个HttpResponse对象，Django便不再会去调用其他的请求(request), 视图(view)或其他中间件，或对应的视图；处理HttpResponse的中间件会处理任何返回的响应(response)。

process_view

process_view(request, view_func, view_args, view_kwargs)

request是一个HttpRequest对象。view_func是 Django会调用的一个Python的函数。(它确实是一个函数对象，不是函数的字符名称。) view_args是一个会被传递到视图的位置参数列表，而view_kwargs 是一个会被传递到视图的关键字参数字典。 view_args和 view_kwargs 都不包括第一个视图参数(request)。

process_view()会在Django调用视图(view)之前被调用。

它将返回None 或一个HttpResponse 对象。如果返回 None，将会继续处理这个请求，执行其他的process_view() 中间件，然后显示对应的视图。如果返回HttpResponse对象，Django就不再会去调用其他的视图（view），异常中间件（exception middleware）或对应的视图；它会把响应中间件应用到HttpResponse上，并返回结果。

> 注意
>
> 在中间件内部，从process_request或者process_view方法中访问request.POST或者request.REQUEST将会阻碍该中间 件之后的所有视图无法修改request的上传处理程序， 一般情况要避免这样使用。
>
> 类CsrfViewMiddleware可以被认为是个例外， 因为它提供了csrf_exempt() 和csrf_protect()两个允许视图来精确控制 在哪个点需要开启CSRF验证。

process_template_response

process_template_response(request, response)

request是一个HttpRequest对象。response是一个TemplateResponse对象（或等价的对象），由Django视图或者中间件返回。

如果响应的实例有render()方法，process_template_response()在视图刚好执行完毕之后被调用，这表明了它是一个TemplateResponse对象（或等价的对象）。

这个方法必须返回一个实现了render方法的响应对象。它可以修改给定的response对象，通过修改 response.template_name和response.context_data或者它可以创建一个全新的TemplateResponse或等价的对象。

你不需要显式渲染响应 —— 一旦所有的模板响应中间件被调用，响应会自动被渲染。

在一个响应的处理期间，中间件以相反的顺序运行，这包括process_template_response()。

process_response

process_response(request, response)

request是一个HttpRequest对象。response是Django视图或者中间件返回的HttpResponse或者StreamingHttpResponse对象。

process_response()在所有响应返回浏览器之前被调用。

这个方法必须返回HttpResponse或者StreamingHttpResponse对象。它可以改变已有的response，或者创建并返回新的HttpResponse或StreamingHttpResponse对象。

不像 process_request()和process_view()方法，即使同一个中间件类中的process_request()和process_view()方法会因为前面的一个中间件返回HttpResponse而被跳过，process_response()方法总是会被调用。特别是，这意味着你的process_response()方法不能依赖于process_request()方法中的设置。

最后，记住在响应阶段中，中间件以相反的顺序被应用，自底向上。意思是定义在MIDDLEWARE_CLASSES最底下的类会最先被运行。

处理流式响应

不像HttpResponse，StreamingHttpResponse并没有content属性。所以，中间件再也不能假设所有响应都带有content属性。如果它们需要访问内容，他们必须测试是否为流式响应，并相应地调整自己的行为。

```
if response.streaming:
    response.streaming_content = wrap_streaming_content(response.streaming_content)
else:
    response.content = alter_content(response.content)
```

> 注意
>
> 我们需要假设streaming_content可能会大到在内存中无法容纳。响应中间件可能会把它封装在新的生成器中，但是一定不要销毁它。封装一般会实现成这样：

```
def wrap_streaming_content(content):
    for chunk in content:
        yield alter_content(chunk)
```

process_exception

process_exception(request, exception)

request是一个HttpRequest对象。exception是一个被视图中的方法抛出来的 Exception对象。

当一个视图抛出异常时，Django会调用process_exception()来处理。process_exception()应该返回一个None 或者一个HttpResponse对象。如果它返回一个HttpResponse对象，模型响应和响应中间件会被应用，响应结果会返回给浏览器。Otherwise, default exception handling kicks in.

再次提醒，在处理响应期间，中间件的执行顺序是倒序执行的，这包括process_exception。如果一个异常处理的中间件返回了一个响应，那这个中间件上面的中间件都将不会被调用。

__init__

大多数的中间件类都不需要一个初始化方法，因为中间件的类定义仅仅是为process_*提供一个占位符。如果你确实需要一个全局的状态那就可以通过__init__来加载。然后要铭记如下两个警告：

Django初始化你的中间件无需任何参数，因此不要定义一个有参数的__init__方法。 不像process_*每次请求到达都要调用__init__只会被调用一次，就是在Web服务启动的时候。

标记中间件不被使用

有时在运行时决定是否一个中间件需要被加载是很有用的。 在这种情况下，你的中间件中的__init__方法可以抛出一个django.core.exceptions.MiddlewareNotUsed异常。Django会从中间件处理过程中移除这部分中间件，并且当DEBUG为True的时候在django.request记录器中记录调试信息。

```
1.8中的修改：

之前 MiddlewareNotUsed异常不会被记录。
```

指导准则

- 中间件的类不能是任何类的子类。
- 中间件可以存在与你Python路径中的任何位置。 Django所关心的只是被包含在MIDDLEWARE_CLASSES中的配置。
- 将Django's available middleware作为例子随便看看。
- 如果你认为你写的中间件组建可能会对其他人有用，那就把它共享到社区！ 让我们知道它，我们会考虑把它添加到Django中。

中间件

这篇文档介绍了Django自带的所有中间件组件。 要查看关于如何使用它们以及如何编写自己的中间件，请见中间件使用指导。

可用的中间件

缓存中间件

class UpdateCacheMiddleware[source]

class FetchFromCacheMiddleware[source]

开启全站范围的缓存。 如果开启了这些缓存，任何一个由Django提供的页面将会被缓存，缓存时长是由你在CACHE_MIDDLEWARE_SECONDS配置中定义的。详见缓存文档。

"常用"的中间件

class CommonMiddleware[source]

给完美主义者增加一些便利条件：

- 禁止访问DISALLOWED_USER_AGENTS中设置的用户代理，这项配置应该是一个已编译的正则表达式对象的列表。
- 基于APPEND_SLASH和PREPEND_WWW的设置来重写URL。

如果APPEND_SLASH设为True并且一开始的的URL没有以斜线结尾，并且在URLconf中也没找到对应定义，这时形成一个一个斜线结尾新的URL。如果这个新的URL存在于URLconf，这时Django会重定向请求到这个新URL上，否则，一开始的URL按正常情况处理。

比如，foo.com/bar将会被重定向到foo.com/bar/，如果你没有为foo.com/bar定义有效的正则，但是为foo.com/bar/定义了有效的正则。

如果PREPEND_WWW设为True，前面缺少 "www."的url将会被重定向到相同但是以一个"www."开头的url。

两种选项都是为了规范化url。其中的哲学就是，任何一个url应该在一个地方仅存在一个。技术上来讲，url foo.com/bar 区别于foo.com/bar/ -- 搜索引擎索引会把这里分开处理 -- 因此，最佳实践就是规范化url。

- 基于 USE_ETAGS 设置来处理ETag。如果设置USE_ETAGS为True，Django会通过MD5-hashing处理页面的内容来为每一个页面请求计算Etag，并且如果合适的话，它将

会发送携带Not Modified的响应。

CommonMiddleware.response_redirect_class

```
Django 1.8中新增
```

默认为HttpResponsePermanentRedirect。它继承了CommonMiddleware，并覆写了属性来自定义中间件发出的重定向。

class BrokenLinkEmailsMiddleware[source]

- 向MANAGERS发送死链提醒邮件（详见错误报告）。

GZip中间件

class GZipMiddleware[source]

> 警告
>
> 安全研究员最近发现，当压缩技术（包括GZipMiddleware）用于一个网站的时候，网站会受到一些可能的攻击。此外，这些方法可以用于破坏Django的CSRF保护。在你的站点使用GZipMiddleware之前，你应该先仔细考虑一下你的站点是否容易受到这些攻击。如果你不确定是否会受到这些影响，应该避免使用 GZipMiddleware。详见the BREACH paper (PDF)和breachattack.com。

为支持GZip压缩的浏览器（一些现代的浏览器）压缩内容。

建议把这个中间件放到中间件配置列表的第一个，这样压缩响应内容的处理会到最后才 发生。

如果满足下面条件的话，内容不会被压缩：

- 消息体的长度小于200个字节。
- 响应已经设置了Content-Encoding协议头。
- 请求（浏览器）没有发送包含gzip的Accept-Encoding协议头。

你可以通过这个gzip_page()装饰器使用独立的GZip压缩。

带条件判断的GET中间件

class ConditionalGetMiddleware[source]

处理带有条件判断状态GET操作。 如果一个请求包含 ETag 或者Last-Modified协议头，并且请求包含If-None-Match或If-Modified-Since，这时响应会被 替换为HttpResponseNotModified。

另外，它会设置Date和Content-Length响应头。

本地中间件

class LocaleMiddleware[source]

基于请求中的数据开启语言选择。 它可以为每个用户进行定制。 详见国际化文档。

LocaleMiddleware.response_redirect_class

默认为HttpResponseRedirect。继承自LocaleMiddleware并覆写了属性来自定义中间件发出的重定向。

消息中间件

class MessageMiddleware[source]

开启基于cookie或会话的消息支持。详见消息文档。

安全中间件

> 警告
>
> 如果你的部署环境允许的话，让你的前端web服务器展示SecurityMiddleware提供的功能是个好主意。这样一来，如果有任何请求没有被Django处理（比如静态媒体或用户上传的文件），他们会拥有和向Django应用的请求相同的保护。

class SecurityMiddleware[source]

```
Django 1.8中新增
```

django.middleware.security.SecurityMiddleware为请求/响应循环提供了几种安全改进。每一种可以通过一个选项独立开启或关闭。

- SECURE_BROWSER_XSS_FILTER
- SECURE_CONTENT_TYPE_NOSNIFF
- SECURE_HSTS_INCLUDE_SUBDOMAINS
- SECURE_HSTS_SECONDS
- SECURE_REDIRECT_EXEMPT
- SECURE_SSL_HOST
- SECURE_SSL_REDIRECT

HTTP Strict Transport Security (HSTS)

对于那些应该只能通过HTTPS访问的站点，你可以通过设置HSTS协议头，通知现代的浏览器，拒绝用不安全的连接来连接你的域名。这会降低你受到SSL-stripping的中间人（MITM）攻击的风险。

如果你将SECURE_HSTS_SECONDS设置为一个非零值，SecurityMiddleware会在所有的HTTPS响应中设置这个协议头。

开启HSTS的时候，首先使用一个小的值来测试它是个好主意，例如，让SECURE_HSTS_SECONDS = 3600为一个小时。每当浏览器在你的站点看到HSTS协议头，都会在提供的时间段内绝对使用不安全（HTTP）的方式连接到你的域名。一旦你确认你站点上的所有东西都以安全的方式提供（例如，HSTS并不会干扰任何事情），建议你增加这个值，这样不常访问你站点的游客也会被保护（比如，一般设置为31536000秒，一年）。

另外，如果你将 SECURE_HSTS_INCLUDE_SUBDOMAINS设置为True,，SecurityMiddleware会将includeSubDomains标签添加到Strict-Transport-Security协议头中。强烈推荐这样做（假设所有子域完全使用HTTPS），否则你的站点仍旧有可能由于子域的不安全连接而受到攻击。

> 警告
>
> HSTS策略在你的整个域中都被应用，不仅仅是你所设置协议头的响应中的url。所以，如果你的整个域都设置为HTTPS only，你应该只使用HSTS策略。
>
> 适当遵循HSTS协议头的浏览器，会通过显示警告的方式，拒绝让用户连接到证书过期的、自行签署的、或者其他SSL证书无效的站点。如果你使用了HSTS，确保你的证书处于一直有效的状态！
>
> 注意
>
> 如果你的站点部署在负载均衡器或者反向代理之后，并且Strict-Transport-Security协议头没有添加到你的响应中，原因是Django有可能意识不到这是一个安全连接。你可能需要设置SECURE_PROXY_SSL_HEADER。

X-Content-Type-Options: nosniff

一些浏览器会尝试猜测他们所得内容的类型，而不是读取Content-Type协议头。虽然这样有助于配置不当的服务器正常显示内容，但也会导致安全问题。

如果你的站点允许用户上传文件，一些恶意的用户可能会上传一个精心构造的文件，当你觉得它无害的时候，文件会被浏览器解释成HTML或者Javascript。

欲知更多有关这个协议头和浏览器如何处理它的内容，你可以在IE安全博客中读到它。

要防止浏览器猜测内容类型，并且强制它一直使用 Content-Type协议头中提供的类型，你可以传递X-Content-Type-Options: nosniff协议头。SecurityMiddleware将会对所有响应这样做，如果SECURE_CONTENT_TYPE_NOSNIFF 设置为True。

注意在大多数Django不涉及处理上传文件的部署环境中，这个设置不会有任何帮助。例如，如果你的MEDIA_URL被前端web服务器直接处理（例如nginx和Apache），你可能想要在那里设置这个协议头。而在另一方面，如果你使用Django执行为了下载文件而请求授权之类的事情，并且你不能使用你的web服务器设置协议头，这个设置会很有用。

X-XSS-Protection: 1; mode=block

一些浏览器能够屏蔽掉出现XSS攻击的内容。通过寻找页面中GET或者POST参数中的JavaScript内容来实现。如果JavaScript在服务器的响应中被重放，页面就会停止渲染，并展示一个错误页来取代。

X-XSS-Protection协议头用来控制XSS过滤器的操作。

要在浏览器中启用XSS过滤器，并且强制它一直屏蔽可疑的XSS攻击，你可以在协议头中传递X-XSS-Protection: 1; mode=block。 如果SECURE_BROWSER_XSS_FILTER设置为True，SecurityMiddleware会在所有响应中这样做。

> 警告
>
> 浏览器的XSS过滤器是一个十分有效的手段，但是不要过度依赖它。它并不能检测到所有的XSS攻击，也不是所有浏览器都支持这一协议头。确保你校验和过滤了所有的输入来防止XSS攻击。

SSL重定向

如果你同时提供HTTP和HTTPS连接，大多数用户会默认使用不安全的（HTTP）链接。为了更高的安全性，你应该讲所有HTTP连接重定向到HTTP连接。

如果你将SECURE_SSL_REDIRECT设置为True，SecurityMiddleware会将HTTP链接永久地（HTTP 301，permanently）重定向到HTTPS连接。

> 注意
>
> 由于性能因素，最好在Django外面执行这些重定向，在nginx这种前端负载均衡器或者反向代理服务器中执行。SECURE_SSL_REDIRECT专门为这种部署情况而设计，当这不可选择的时候。

如果SECURE_SSL_HOST设置有一个值，所有重定向都会发到值中的主机，而不是原始的请求主机。

如果你站点上的一些页面应该以HTTP方式提供，并且不需要重定向到HTTPS，你可以SECURE_REDIRECT_EXEMPT设置中列出匹配那些url的正则表达式。

> 注意
>
> 如果你在负载均衡器或者反向代理服务器后面部署应用，而且Django不能辨别出什么时候一个请求是安全的，你可能需要设置SECURE_PROXY_SSL_HEADER。

会话中间件

class SessionMiddleware[source]

开启会话支持。详见会话文档。

站点中间件

class CurrentSiteMiddleware[source]

```
Django 1.7中新增
```

向每个接收到的HttpRequest对象添加一个site属性，表示当前的站点。详见站点文档。

认证中间件

class AuthenticationMiddleware[source]

向每个接收到的HttpRequest对象添加user属性，表示当前登录的用户。详见web请求中的认证。

class RemoteUserMiddleware[source]

使用web服务器提供认证的中间件。详见使用REMOTE_USER进行认证。

class SessionAuthenticationMiddleware[source]

```
Django 1.7中新增
```

当用户修改密码的时候使用户的会话失效。详见密码更改时的会话失效。在MIDDLEWARE_CLASSES中，这个中间件必须出现在django.contrib.auth.middleware.AuthenticationMiddleware之后。

CSRF保护中间件

class CsrfViewMiddleware[source]

添加跨站点请求伪造的保护，通过向POST表单添加一个隐藏的表单字段，并检查请求中是否有正确的值。详见CSRF保护文档。

X-Frame-Options中间件

class XFrameOptionsMiddleware[source]

通过X-Frame-Options协议头进行简单的点击劫持保护。

中间件的排序

下面是一些关于Django中间件排序的提示。

UpdateCacheMiddleware

放在修改大量协议头的中间件(SessionMiddleware, GZipMiddleware, LocaleMiddleware)之前。

GZipMiddleware

放在任何可能修改或使用响应消息体的中间件之前。

放在UpdateCacheMiddleware之后：会修改大量的协议头。

ConditionalGetMiddleware

放在CommonMiddleware之前：当USE_ETAGS = True时会使用它的Etag 协议头。

SessionMiddleware

放在UpdateCacheMiddleware之后：会修改 大量协议头。

LocaleMiddleware

放在SessionMiddleware（由于使用会话数据）和 CacheMiddleware（由于要修改大量协议头）之后的最上面。

CommonMiddleware

放在任何可能修改相应的中间件之前（因为它会生成ETags）。

在GZipMiddleware之后，不会在压缩后的内容上再去生成ETag。

尽可能放在靠上面的位置，因为APPEND_SLASH或者PREPEND_WWW设置为 True时会被重定向。

CsrfViewMiddleware

放在任何假设CSRF攻击被处理的视图中间件之前。

AuthenticationMiddleware

放在SessionMiddleware之后：因为它使用会话存储。

MessageMiddleware

放在SessionMiddleware之后：会使用基于会话的存储。

FetchFromCacheMiddleware

放在任何修改大量协议头的中间件之后：协议头被用来从缓存的哈希表中获取值。

FlatpageFallbackMiddleware

应该放在最底下，因为他是中间件中的底牌。

RedirectFallbackMiddleware

应该放在最底下，因为他是中间件中的底牌。

模板层

模板层提供了设计友好的语法来展示信息给用户。 了解其语法可以让设计师知道如何使用，让程序员知道如何扩展：

基础

面向设计师

Django模版语言

本文将介绍Django模版系统的语法。如果您需要更多该系统如何工作的技术细节，以及希望扩展它，请浏览 *The Django template language: for Python programmers*.

Django模版语言的设计致力于在性能和简单上取得平衡。 它的设计使习惯于使用HTML的人也能够自如应对。如果您有过使用其他模版语言的经验，像是 Smarty 或者 Jinja2, 那么您将对Django的模版语言感到一见如故。

理念

如果您有过编程背景，或者您使用过一些在HTML中直接混入程序代码的语言，那么现在您需要记住，Django的模版系统并不是简单的将Python嵌入到HTML中。 设计决定了：模版系统致力于表达外观，而不是程序逻辑。

Django的模版系统提供了和一些程序结构功能类似的标签——用于布尔判断的 `if` 标签, 用于循环的 `for` 标签等等。——但是这些都不是简单的作为Python代码那样来执行的，并且，模版系统也不会随意执行Python表达式。只有下面列表中的标签、过滤器和语法才是默认就被支持的。 （但是您也可以根据需要添加 您自己的扩展 到模版语言中）。

模版

模版是纯文本文件。它可以产生任何基于文本的的格式（HTML，XML，CSV等等）。

模版包括在使用时会被值替换掉的 变量，和控制模版逻辑的 标签。

下面是一个小模版，它说明了一些基本的元素。后面的文档中会解释每个元素。

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

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

理念

为什么要使用基于文本的模版，而不是基于XML的（比如Zope的TAL）呢？我们希望Django的模版语言可以用在更多的地方，而不仅仅是XML/HTML模版。在线上世界，我们在email、Javascript和CSV中使用它。你可以在任何基于文本的格式中使用这个模版语言。

还有，让人类编辑HTML简直是施虐狂的做法！

变量

变量看起来就像是这样： `{{ variable }}` . 当模版引擎遇到一个变量，它将计算这个变量，然后用结果替换掉它本身。变量的命名包括任何字母数字以及下划线 (`"_"`)的组合。点 (`"."`) 也在会变量部分中出现，不过它有特殊的含义，我们将在后面说明。重要的是， 你不能在变量名称中使用空格和标点符号。

使用点 (`.`) 来访问变量的属性。

幕后

从技术上来说，当模版系统遇到点，它将以这样的顺序查询：

- 字典查询（Dictionary lookup）
- 属性或方法查询（Attribute or method lookup）
- 数字索引查询（Numeric index lookup）

如果计算结果的值是可调用的，它将被无参数的调用。调用的结果将成为模版的值。

这个查询顺序，会在优先于字典查询的对象上造成意想不到的行为。例如，思考下面的代码片段，它尝试循环 `collections.defaultdict`：

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

因为字典查询首先发生，行为奏效了并且提供了一个默认值，而不是使用我们期望的 `.iteritems()` 方法。在这种情况下，考虑首先转换成字典。

在前文的例子中， `{{ section.title }}` 将被替换为 `section` 对象的 `title` 属性。

如果你使用的变量不存在， 模版系统将插入 `string_if_invalid` 选项的值， 它被默认设置为 `''` (空字符串) 。

注意模版表达式中的“bar”， 比如 `{{ foo.bar }}` 将被逐字直译为一个字符串，而不是使用变量“bar”的值，如果这样一个变量在模版上下文中存在的话。

过滤器

您可以通过使用 过滤器来改变变量的显示。

过滤器看起来是这样的：`{{ name|lower }}`。这将在变量 `{{ name }}` 被过滤器 `lower` 过滤后再显示它的值，该过滤器将文本转换成小写。使用管道符号 (`|`)来应用过滤器。

过滤器能够被“串联”。一个过滤器的输出将被应用到下一个。`{{ text|escape|linebreaks }}` 就是一个常用的过滤器链，它编码文本内容，然后把行打破转成 `<p>` 标签。

一些过滤器带有参数。过滤器的参数看起来像是这样： `{{ bio|truncatewords:30 }}`。这将显示 `bio` 变量的前30个词。

过滤器参数包含空格的话，必须被引号包起来；例如，连接一个有逗号和空格的列表，你需要使用 `{{ list|join:", " }}`。

Django提供了大约六十个内置的模版过滤器。你可以在 内置过滤器参考手册中阅读全部关于它们的信息。为了体验一下它们的作用，这里有一些常用的模版过滤器：

`default`

如果一个变量是false或者为空，使用给定的默认值。否则，使用变量的值。例如：

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

如果 `value` 没有被提供，或者为空， 上面的例子将显示“ `nothing` ”。

`length`

返回值的长度。它对字符串和列表都起作用。例如：

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

如果 `value` 是 `['a', 'b', 'c', 'd']`，那么输出是 `4`。

`filesizeformat`

将值格式化为一个 “人类可读的” 文件尺寸 （例如 `'13 KB'`，`'4.1 MB'`，`'102 bytes'`，等等）。例如：

```
{{ value|filesizeformat }}
```

如果 `value` 是 123456789，输出将会是 `117.7 MB`。

再说一下，这仅仅是一些例子；查看 内置过滤器参考手册 来获取完整的列表。

您也可以创建自己的自定义模版过滤器；参考 自定义模版标签和过滤器。

更多

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

标签

标签看起来像是这样的： `{% tag %}` 。标签比变量更加复杂：一些在输出中创建文本，一些通过循环或逻辑来控制流程，一些加载其后的变量将使用到的额外信息到模版中。

一些标签需要开始和结束标签 （例如 `{% tag %} ... 标签` 内容 ... {% endtag %}）。

Django自带了大约24个内置的模版标签。你可以在 内置标签参考手册中阅读全部关于它们的内容。为了体验一下它们的作用，这里有一些常用的标签：

`for`

循环数组中的每个元素。例如，显示 `athlete_list` 中提供的运动员列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

`if` , `elif` , and `else`

计算一个变量，并且当变量是“true”是，显示块中的内容：

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

在上面的例子中，如果 `athlete_list` 不是空的，运动员的数量将显示为 `{{ athlete_list|length }}` 的输出。另一方面， 如果 `athlete_in_locker_room_list` 不为空， 将显示 “Athletes should be out...” 这个消息。如果两个列表都是空的，将显示 “No athletes.” 。

您也可以在 `if` 标签中使用过滤器和多种运算符：

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

当上面的例子工作时，需要注意，大多数模版过滤器返回字符串，所以使用过滤器做数学的比较通常都不会像您期望的那样工作。 `length` 是一个例外。

`block` and `extends`

Set up template inheritance (see below), a powerful way of cutting down on “boilerplate” in templates.

再说一下，上面的仅仅是整个列表的一部分；查看 内置标签参考手册 来获取完整的列表。

您也可以创建您自己的自定义模版标签；参考 自定义模版标签和过滤器。

更多

Django’s admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

注释

要注释模版中一行的部分内容，使用注释语法 `{# #}`.

例如，这个模版将被渲染为 `'hello'` ：

```
{# greeting #}hello
```

注释可以包含任何模版代码，有效的或者无效的都可以。例如：

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

这个语法只能被用于单行注释 （在 `{#` 和 `#}` 分隔符中，不允许有新行）。如果你需要注释掉模版中的多行内容，请查看 `comment` 标签。

模版继承

Django模版引擎中最强大也是最复杂的部分就是模版继承了。模版继承可以让您创建一个基本的“骨架”模版，它包含您站点中的全部元素，并且可以定义能够被子模版覆盖的 **blocks** 。

通过从下面这个例子开始，可以容易的理解模版继承：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{%/span> endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

这个模版，我们把它叫作 `base.html`， 它定义了一个可以用于两列排版页面的简单HTML骨架。“子模版”的工作是用它们的内容填充空的blocks。

在这个例子中， `block` 标签定义了三个可以被子模版内容填充的block。 `block` 告诉模版引擎： 子模版可能会覆盖掉模版中的这些位置。

子模版可能看起来是这样的：

```
{% extends "base.html" %}/span>

{% block title %}My amazing blog{% endblock %}

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

`extends` 标签是这里的关键。它告诉模版引擎，这个模版“继承”了另一个模版。当模版系统处理这个模版时，首先，它将定位父模版——在此例中，就是“base.html”。

那时，模版引擎将注意到 `base.html` 中的三个 `block` 标签，并用子模版中的内容来替换这些block。根据 `blog_entries` 的值，输出可能看起来是这样的：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

请注意，子模版并没有定义 `sidebar` block，所以系统使用了父模版中的值。父模版的 `{% block %}` 标签中的内容总是被用作备选内容（fallback）。

您可以根据需要使用多级继承。使用继承的一个常用方式是类似下面的三级结构：

- 创建一个 `base.html` 模版来控制您整个站点的主要视觉和体验。
- 为您的站点的每一个“部分”创建一个 `base_SECTIONNAME.html` 模版。 例如，`base_news.html`，`base_sports.html`。这些模版都继承自 `base.html`，并且包含了每部分特有的样式和设计。
- 为每一种页面类型创建独立的模版，例如新闻内容或者博客文章。这些模版继承了有关的部分模版（section template）。

这种方式使代码得到最大程度的复用，并且使得添加内容到共享的内容区域更加简单，例如，部分范围内的导航。

这里是使用继承的一些提示：

- 如果你在模版中使用 `{% extends %}` 标签，它必须是模版中的第一个标签。其他的任何情况下，模版继承都将无法工作。

- 在base模版中设置越多的 `{% block %}` 标签越好。请记住，子模版不必定义全部父模版中的blocks，所以，你可以在大多数blocks中填充合理的默认内容，然后，只定义你需要的那一个。多一点钩子总比少一点好。

- 如果你发现你自己在大量的模版中复制内容，那可能意味着你应该把内容移动到父模版中的一个 `{% block %}` 中。

- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.
- 为了更好的可读性，你也可以给你的 `{% endblock %}` 标签一个 名字 。例如：

```
{% block content %}
...
{% endblock content %}
```

  在大型模版中，这个方法帮你清楚的看到哪一个 `{% block %}` 标签被关闭了。

最后，请注意您并不能在一个模版中定义多个相同名字的 `block` 标签。这个限制的存在是因为block标签的作用是“双向”的。这个意思是，block标签不仅提供了一个坑去填，它还在 父模版中定义了填坑的内容。如果在一个模版中有两个名字一样的 `block` 标签，模版的父模版将不知道使用哪个block的内容。

自动HTML转义

当从模版中生成HTML时，总会有这样一个风险：值可能会包含影响HTML最终呈现的字符。例如，思考这个模版片段：

```
Hello, {{ name }}
```

首先，它看起来像是一个无害的方式来显示用户的名字，但是设想一下，如果用户像下面这样输入他的名字，会发生什么：

```
<script>alert('hello')</script>
```

使用这个名字值，模版将会被渲染成这样：

```
Hello, <script>alert('hello')</script>
```

...这意味着，浏览器将会弹出一个Javascript警示框！

类似的，如果名字包含一个 `'<'` 符号（比如下面这样），会发生什么呢？

```
<b>username
```

这将会导致模版呗渲染成这样：

```
Hello, <b>username
```

...进而这将导致网页的剩余部分都被加粗！

显然，用户提交的数据都被应该被盲目的信任，并且被直接插入到你的网页中，因为一个怀有恶意的用户可能会使用这样的漏洞来做一些可能的坏事。这种类型的安全问题被叫做 跨站脚本（Cross Site Scripting） (XSS) 攻击。

为避免这个问题，你有两个选择：

- 第一， 你可以确保每一个不被信任的值都通过 `escape` 过滤器（下面的文档中将提到）运行，它将把潜在的有害HTML字符转换成无害的。This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- 第二，你可以利用Django的自动HTML转义。 本节描述其余部分描述的是自动转义是如何工作的

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- `<` 会转换为 `<`
- `>` 会转换为 `>`
- `'` (单引号) 会转换为 `&#39;`
- `"` (双引号)会转换为 `&quot;`
- `&` 会转换为 `&`

我们要再次强调这是默认行为。如果你使用Django的模板系统，会处于保护之下。

如果关闭它

如果你不希望数据自动转义，在站点、模板或者变量级别，你可以使用几种方法来关闭它。

然而你为什么想要关闭它呢？由于有时，模板变量含有一些你打算渲染成原始HTML的数据，你并不想转义这些内容。例如，你可能会在数据库中储存一些HTML代码，并且直接在模板中嵌入它们。或者，你可能使用Django的模板系统来生成_不是_HTML的文本 -- 比如邮件信息。

用于独立变量

使用 `safe` 过滤器来关闭独立变量上的自动转移：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

*safe*是*safe from further escaping*或者*can be safely interpreted as HTML*的缩写。 在这个例子中，如果 `data` 含有 `'<b>'`，输出会是：

```
This will be escaped: <b>
This will not be escaped: <b>
```

用于模板代码块

要控制模板上的自动转移，将模板（或者模板中的特定区域）包裹在 `autoescape` 标签 中，像这样：

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

`autoescape` 标签接受 `on` 或者 `off` 作为它的参数。有时你可能想在自动转移关闭的情况下强制使用它。下面是一个模板的示例：

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

自动转移标签作用于扩展了当前模板的模板，以及通过 `include` 标签包含的模板，就像所有block标签那样。例如：

base.html

```
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

child.html

```
{% extends "base.html" %}
{% block title %}This &amp; that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

由于自动转义标签在base模板中关闭，它也会在child模板中关闭，导致当 `greeting` 变量含有 `<b>Hello!</b>` 字符串时，会渲染HTML。

```
<h1>This &amp;amp; that</h1>
<b>Hello!</b>
```

注释

通常，模板的作用并不非常担心自动转义。Python一边的开发者（编写视图和自定义过滤器的人）需要考虑数据不应被转移的情况，以及合理地标记数据，让这些东西在模板中正常工作。

如果你创建了一个模板，它可能用于你不确定自动转移是否开启的环境，那么应该向任何需要转移的变量添加 `escape` 过滤器。当自动转移打开时， `escape` 过滤器双重过滤数据没有任何危险 -- `escape` 过滤器并不影响自动转义的变量。

字符串字面值和自动转义

像我们之前提到的那样，过滤器参数可以是字符串：

```
{{ data|default:"This is a string literal." }}
```

所有字面值字符串在插入模板时都 不会带有任何自动转义 -- 它们的行为类似于通过 `safe` 过滤器传递。背后的原因是，模板作者可以控制字符串字面值得内容，所以它们可以确保在模板编写时文本经过正确转义。

也即是说你可以编写

```
{{ data|default:"3 < 2" }}
```

...而不是：

```
{{ data|default:"3 < 2" }}  {# Bad! Don't do this. #}
```

这并不影响来源于模板自身的数据。模板内容在必要时仍然会自动转移，因为它们不受模板作者的控制。

访问方法调用

大多数对象上的方法调用同样可用于模板中。这意味着模板必须拥有对除了类属性（像是字段名称）和从视图中传入的变量之外的访问。例如，Django ORM提供了*“entry_set”* 语法用于查找关联到外键的对象集合。所以，提供一个模型叫做“comment”，并带有一个关联到 “task” 模型的外键，你就可以遍历给定任务附带的所有评论，像这样：

```
{% for comment in task.comment_set.all %}
    {{ comment }}
{% endfor %}
```

与之类似，*QuerySets*提供了 `count()` 方法来计算含有对象的总数。因此，你可以像这样获取所有关于当前任务的评论总数：

```
{{ task.comment_set.all.count }}
```

当然，你可以轻易访问已经显式定义在你自己模型上的方法：

models.py

```
class Task(models.Model):
    def foo(self):
        return "bar"
```

template.html

```
{{ task.foo }}
```

由于Django有意限制了模板语言中逻辑处理的总数，不能够在模板中传递参数来调用方法。数据应该在视图中处理，然后传递给模板用于展示。

自定义标签和过滤器库

特定的应用提供自定义的标签和过滤器库。要在模板中访问它们，确保应用在 `INSTALLED_APPS` 之内（在这个例子中我们添加了 `'django.contrib.humanize'`），之后在模板中使用 `load` 标签：

```
{% load humanize %}

{{ 45000|intcomma }}
```

上面的例子中， `load` 标签加载了 `humanize` 标签库，之后我们可以使用 `intcomma` 过滤器。如果你开启了 `django.contrib.admindocs`，你可以查询admin站点中的文档部分，来寻找你的安装中的自定义库列表。

`load` 标签可以接受多个库名称，由空格分隔。例如：

```
{% load humanize i18n %}
```

关于编写你自己的自定义模板库，详见自定义模板标签和过滤器。

自定义库和模板继承

当你加载一个自定义标签或过滤器库时，标签或过滤器只在当前模板中有效 -- 并不是带有模板继承关系的任何父模板或者子模版中都有效。

例如，如果一个模板 `foo.html` 带有 `{% load humanize %}`，子模版（例如，带有 `{% extends "foo.html" %}`）中不能访问humanize模板标签和过滤器。子模版需要添加自己的 `{% load humanize %}`。

这个特性是可维护性和逻辑性的缘故。

另见

The Templates Reference

Covers built-in tags, built-in filters, using an alternative template, language, and more.

> 译者：Django 文档协作翻译小组，原文：Language overview。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

django.contrib.humanize

一系列Django的模板过滤器，有助于向数据添加“人文关怀”。

把'django.contrib.humanize'添加到INSTALLED_APPS设置来激活这些过滤器。 执行以上步骤之后，在模板中使用{% load humanize %}，你就可以访问到下面的过滤器了。

基数词

对于数字1~9，返回拼写出来的数字。否则返回数字本身。这样遵循了出版的格式。

例如：

- 1 会变成one。
- 2 会变成 two。
- 10 会变成 10。

你可以传递整数，或者整数的字符串形式。

整数间的逗号

将整数转化为字符串，每三位之间带一个逗号。

例如：

- 4500 会变成 4,500。
- 45000 会变成 45,000
- 450000 会变成 450,000。
- 4500000 会变成 4,500,000。

如果启动了格式本地化，将会被遵循。例如，在德语（'de'）中：

- 45000 会变成 '45.000'。
- 450000 会变成 '450.000'。

你可以传递整数，或者整数的字符串形式。

整数词组

将一个大的整数转化为友好的文字表示形式。适用于超过一百万的数字。

例如：

- 1000000 会变成 1.0 million。
- 1200000 会变成 1.2 million。
- 1200000000 会变成 1.2 billion。

支持高达10的100次方 (Googol) 的整数。

如果启动了格式本地化将会被遵循。例如，在德语（'de'）中：

- 1000000 会变成 '1,0 Million'。
- 1200000 会变成 '1,2 Million'。
- 1200000000 会变成 '1,2 Milliarden'。

你可以传递整数，或者整数的字符串形式。

自然日期

对于当天或者一天之内的日期， 返回“今天”，“明天”或者“昨天”，视情况而定。否则，使用传进来的格式字符串给日期格式化。

参数：日期的格式字符串在date标签中描述。

例如（其中“今天”是2007年2月17日）：

- 16 Feb 2007 会变成 yesterday。
- 17 Feb 2007 会变成 today。
- 18 Feb 2007 会变成 tomorrow。

其他日期按照提供的参数格式化，如果没提供参数的话，将会按照DATE_FORMAT 设置。

自然时间

对于日期时间的值，返回一个字符串来表示多少秒、分钟或者小时之前 —— 如果超过一天之前，则回退为使用timesince格式。如果是未来的日期时间，返回值会自动使用合适的文字表述。

例如（其中“现在”是2007年2月17日16时30分0秒）：

- 17 Feb 2007 16:30:00 会变成 now。
- 17 Feb 2007 16:29:31 会变成 29 seconds ago。
- 17 Feb 2007 16:29:00 会变成 a minute ago。
- 17 Feb 2007 16:25:35 会变成 4 minutes ago。
- 17 Feb 2007 15:30:29 会变成 59 minutes ago。
- 17 Feb 2007 15:30:01 会变成 59 minutes ago。
- 17 Feb 2007 15:30:00 会变成 an hour ago。

- 17 Feb 2007 13:31:29 会变成 2 hours ago。
- 16 Feb 2007 13:31:29 会变成 1 day, 2 hours ago。
- 16 Feb 2007 13:30:01 会变成 1 day, 2 hours ago。
- 16 Feb 2007 13:30:00 会变成 1 day, 3 hours ago。
- 17 Feb 2007 16:30:30 会变成 30 seconds from now。
- 17 Feb 2007 16:30:29 会变成 29 seconds from now。
- 17 Feb 2007 16:31:00 会变成 a minute from now。
- 17 Feb 2007 16:34:35 会变成 4 minutes from now。
- 17 Feb 2007 17:30:29 会变成 an hour from now。
- 17 Feb 2007 18:31:29 会变成 2 hours from now。
- 18 Feb 2007 16:31:29 会变成 1 day from now。
- 26 Feb 2007 18:31:29 会变成 1 week, 2 days from now。

序数词

将一个整数转化为它的序数词字符串。

例如：

- 1 会变成 1st。
- 2 会变成 2nd。
- 3 会变成 3rd。

你可以传递整数，或者整数的字符串形式。

面向程序员

表单

Django 提供了一个丰富的框架可便利地创建表单及操作表单数据。

基础

使用表单

> 关于这页文档
>
> 这页文档简单介绍Web 表单的基本概念和它们在Django 中是如何处理的。关于表单API 某方面的细节，请参见表单 API、表单的字段和表单和字段的检验。

除非你计划构建的网站和应用只是发布内容而不接受访问者的输入，否则你将需要理解并使用表单。

Django 提供广泛的工具和库来帮助你构建表单来接收网站访问者的输入，然后处理以及响应输入。

HTML 表单

在HTML中，表单是位于 `<form>...</form>` 之间的元素的集合，它们允许访问者输入文本、选择选项、操作对象和控制等等，然后将信息发送回服务器。

某些表单的元素 —— 文本输入和复选框 —— 非常简单而且内建于HTML 本身。其它的表单会复杂些；例如弹出一个日期选择对话框的界面、允许你移动滚动条的界面、使用JavaScript 和CSS 以及HTML 表单 `<input>` 元素来实现操作控制的界面。

与 `<input>` 元素一样，一个表单必须指定两样东西：

- where：响应用户输入的URL
- how：HTTP 方法

例如，Django Admin 站点的登录表单包含几个 `<input>` 元素： `type="text"` 用于用户名， `type="password"` 用于密码， `type="submit"` 用于“Log in" 按钮。它还包含一些用户看不到的隐藏的文本字段，Django 使用它们来决定下一步的行为。

它还告诉浏览器表单数据应该发往 `<form>` 的 `action` 属性指定的URL —— `/admin/` ，而且应该使用 `method` 属性指定的HTTP 方法 —— `post` 。

当触发 `<input type="submit" value="Log in">` 元素时，数据将发送给 `/admin/` 。

GET 和 POST

处理表单时候只会用到 `GET` 和 `POST` 方法。

Django 的登录表单使用POST 方法，在这个方法中浏览器组合表单数据、对它们进行编码以用于传输、将它们发送到服务器然后接收它的响应。

相反，GET 组合提交的数据为一个字符串，然后使用它来生成一个URL。这个URL 将包含数据发送的地址以及数据的键和值。如果你在Django 文档中做一次搜索，你会立即看到这点，此时将生成一个 `https://docs.djangoproject.com/search/?q=forms&release=1` 形式的URL。

`GET` 和 `POST` 用于不同的目的。

用于改变系统状态的请求 —— 例如，给数据库带来变化的请求 —— 应该使用 `POST` 。 `GET` 只应该用于不会影响系统状态的请求。

`GET` 还不适合密码表单，因为密码将出现在URL 中，以及浏览器的历史和服务器的日志中，而且都是以普通的文本格式。它还不适合数据量大的表单和二进制数据，例如一张图片。使用GET 请求作为管理站点的表单具有安全隐患：攻击者很容易模拟表单请求来取得系统的敏感数据。 `POST` ，如果与其它的保护措施结合将对访问提供更多的控制，例如Django 的CSRF保护。

另一个方面， `GET` 适合网页搜索这样的表单，因为这种表示一个 `GET` 请求的URL 可以很容易地作为书签、分享和重新提交。

Django 在表单中的角色

处理表单是一件很复杂的事情。考虑一下Django 的Admin 站点，不同类型的大量数据项需要在一个表单中准备好、渲染成HTML、使用一个方便的界面编辑、返回给服务器、验证并清除，然后保存或者向后继续处理。

Django 的表单功能可以简化并自动化大部分这些工作，而且还可以比大部分程序员自己所编写的代码更安全。

Django 会处理表单工作中的三个显著不同的部分：

- 准备并重新构造数据
- 为数据创建HTML 表单
- 接收并处理客户端提交的表单和数据

可以手工编写代码来实现，但是Django 可以帮你完成所有这些工作。

Django 中的表单

我们已经简短讲述HTML 表单，但是HTML的 `<form>` 只是其机制的一部分。

在一个Web 应用中，‘表单’可能指HTML `<form>` 、或者生成它的Django 的 `Form` 、或者提交时发送的结构化数据、或者这些部分的总和。

Django 的Form 类

表单系统的核心部分是Django 的 `Form` 类。Django 的模型描述一个对象的逻辑结构、行为以及展现给我们的方式，与此类似， `Form` 类描述一个表单并决定它如何工作和展现。

模型类的字典映射到数据库的字典，与此类似，表单类的字段映射到HTML 的表单 `<input>` 元素。（ `ModelForm` 通过一个 `Form` 映射模型类的字段到HTML 表单的 `<input>` 元素；Django 的Admin 站点就是基于这个）。

表单的字段本身也是类；它们管理表单的数据并在表单提交时进行验证。 `DateField` 和 `FileField` 处理的数据类型差别很大，必须完成不同的事情。

表单字段在浏览器中呈现给用户的是一个HTML 的“widget” —— 用户界面的一个片段。每个字段类型都有一个合适的默认Widget 类，需要时可以覆盖。

实例化、处理和渲染表单

在Django 中渲染一个对象时，我们通常：

1. 在视图中获得它（例如，从数据库中获取）
2. 将它传递给模板上下文
3. 使用模板变量将它扩展为HTML 标记

在模板中渲染表单和渲染其它类型的对象几乎一样，除了几个关键的差别。

在模型实例不包含数据的情况下，在模板中对它做处理很少有什么用处。但是渲染一个未填充的表单却非常有意义 —— 我们希望用户去填充它。

所以当我们在视图中处理模型实例时，我们一般从数据库中获取它。当我们处理表单时，我们一般在视图中实例化它。

当我们实例化表单时，我们可以选择让它为空还是预先填充它，例如使用：

- 来自一个保存后的模型实例的数据（例如用于编辑的管理表单）
- 我们从其它地方获得的数据
- 从前面一个HTML 表单提交过来的数据

最后一种情况最令人关注，因为它使得用户可以不只是阅读一个网站，而且可以给网站返回信息。

构建一个表单

需要完成的工作

假设你想在你的网站上创建一个简单的表单，以获得用户的名字。你需要类似这样的模板：

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

这告诉浏览器发送表单的数据到URL `/your-name/`，并使用 `POST` 方法。它将显示一个标签为"Your name:"的文本字段，和一个"OK"按钮。如果模板上下文包含一个 `current_name` 变量，它将用于预填充 `your_name` 字段。

你将需要一个视图来渲染这个包含HTML 表单的模板，并提供合适的 `current_name` 字段。

当表单提交时，发往服务器的 `POST` 请求将包含表单数据。

现在你还需要一个对应 `/your-name/` URL 的视图，它在请求中找到正确的键/值对，然后处理它们。

这是一个非常简单的表单。实际应用中，一个表单可能包含几十上百个字段，其中大部分需要预填充，而且我们预料到用户将来回编辑-提交几次才能完成操作。

我们可能需要在表单提交之前，在浏览器端作一些验证。我们可能想使用非常复杂的字段，以允许用户做类似从日历中挑选日期这样的事情，等等。

这个时候，让Django 来为我们完成大部分工作是很容易的。

在Django 中构建一个表单

Form 类

我们已经计划好了我们的 HTML 表单应该呈现的样子。在Django 中，我们的起始点是这里：

```
#forms.py

from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

它定义一个 `Form` 类，只带有一个字段（ `your_name` ）。我们已经对这个字段使用一个友好的标签，当渲染时它将出现在 `<label>` 中（在这个例子中，即使我们省略它，我们指定的 `label` 还是会自动生成）。

字段允许的最大长度通过 `max_length` 定义。它完成两件事情。首先，它在HTML 的 `<input>` 上放置一个 `maxlength="100"` （这样浏览器将在第一时间阻止用户输入多于这个数目的字符）。它还意味着当Django 收到浏览器发送过来的表单时，它将验证数据的长度。

`Form` 的实例具有一个 `is_valid()` 方法，它为所有的字段运行验证的程序。当调用这个方法时，如果所有的字段都包含合法的数据，它将：

- 返回 `True`
- 将表单的数据放到 `cleaned_data` 属性中。

完整的表单，第一次渲染时，看上去将像：

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100">
```

注意它不包含 `<form>` 标签和提交按钮。我们必须自己在模板中提供它们。

视图

发送给Django 网站的表单数据通过一个视图处理，一般和发布这个表单的是同一个视图。这允许我们重用一些相同的逻辑。

当处理表单时，我们需要在视图中实例化它：

```
#views.py

from django.shortcuts import render
from django.http import HttpResponseRedirect

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

如果访问视图的是一个 `GET` 请求，它将创建一个空的表单实例并将它放置到要渲染的模板的上下文中。这是我们在第一个访问该URL 时预期发生的情况。

如果表单的提交使用 `POST` 请求，那么视图将再次创建一个表单实例并使用请求中的数据填充它：`form = NameForm(request.POST)`。这叫做”绑定数据至表单“（它现在是一个绑定的表单）。

我们调用表单的 `is_valid()` 方法；如果它不为 `True`，我们将带着这个表单返回到模板。这时表单不再为空（未绑定），所以HTML 表单将用之前提交的数据填充，然后可以根据要求编辑并改正它。

如果 `is_valid()` 为 `True`，我们将能够在 `cleaned_data` 属性中找到所有合法的表单数据。在发送HTTP 重定向给浏览器告诉它下一步的去向之前，我们可以用这个数据来更新数据库或者做其它处理。

模板

我们不需要在name.html 模板中做很多工作。最简单的例子是：

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

根据 `{{ form }}`，所有的表单字段和它们的属性将通过Django 的模板语言拆分成HTML 标记。

> 表单和跨站请求伪造的防护
>
> Django 原生支持一个简单易用的跨站请求伪造的防护。当提交一个启用CSRF 防护的 `POST` 表单时，你必须使用上面例子中的 `csrf_token` 模板标签。然而，因为CSRF 防护在模板中不是与表单直接捆绑在一起的，这个标签在这篇文档的以下示例中将省略。
>
> HTML5 输入类型和浏览器验证
>
> 如果你的表单包含 `URLField`、`EmailField` 和其它整数字段类似，Django 将使用 `url`、`email` 和 `number` 这样的HTML5 输入类型。默认情况下，浏览器可能会对这些字段进行它们自身的验证，这些验证可能比Django 的验证更严格。如果你想禁用这个行为，请设置 `form` 标签的 `novalidate` 属性，或者指定一个不同的字段，如 `TextInput`。

现在我们有了一个可以工作的网页表单，它通过Django Form 描述、通过视图处理并渲染成一个HTML `<form>`。

这是你入门所需要知道的所有内容，但是表单框架为了提供了更多的内容。一旦你理解了上面描述的基本处理过程，你应该可以理解表单系统的其它功能并准备好学习更多的底层机制。

Django Form 类详解

所有的表单类都作为 `django.forms.Form` 的子类创建，包括你在Django 管理站点中遇到的 `ModelForm`。

> 模型和表单
>
> 实际上，如果你的表单打算直接用来添加和编辑Django 的模型，ModelForm 可以节省你的许多时间、精力和代码，因为它将根据 `Model` 类构建一个表单以及适当的字段和属性。

绑定的和未绑定的表单实例

绑定的和未绑定的表单 之间的区别非常重要：

- 未绑定的表单没有关联的数据。当渲染给用户时，它将为空或包含默认的值。
- 绑定的表单具有提交的数据，因此可以用来检验数据是否合法。如果渲染一个不合法的绑定的表单，它将包含内联的错误信息，告诉用户如何纠正数据。

表单的 `is_bound` 属性将告诉你一个表单是否具有绑定的数据。

字段详解

考虑一个比上面的迷你示例更有用的一个表单，我们可以用它来在一个个人网站上实现“联系我”功能：

```
#forms.py

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

我们前面的表单只使用一个字段 `your_name`，它是一个 `CharField`。在这个例子中，我们的表单具有四个字段：`subject`、`message`、`sender` 和 `cc_myself`。共用到三种字段类型：`CharField`、`EmailField` 和 `BooleanField`；完整的字段类型列表可以在表单字段中找到。

Widgets

每个表单字段都有一个对应的 `Widget` 类，它对应一个HTML 表单 `Widget`，例如 `<input type="text">`。

在大部分情况下，字段都具有一个合理的默认Widget。例如，默认情况下，`CharField` 具有一个 `TextInput Widget`，它在HTML 中生成一个 `<input type="text">`。如果你需要 `<textarea>`，在定义表单字段时你应该指定一个合适的 `Widget`，例如我们定义的 `message` 字段。

字段的数据

不管表单提交的是什么数据，一旦通过调用 `is_valid()` 成功验证（`is_valid()` 返回 `True`），验证后的表单数据将位于 `form.cleaned_data` 字典中。这些数据已经为你转换好为Python 的类型。

> 注
>
> 此时，你依然可以从 `request.POST` 中直接访问到未验证的数据，但是访问验证后的数据更好一些。

在上面的联系表单示例中，`cc_myself` 将是一个布尔值。类似地，`IntegerField` 和 `FloatField` 字段分别将值转换为Python 的 `int` 和 `float`。

下面是在视图中如何处理表单数据：

```
#views.py

from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

> 提示
>
> 关于Django 中如何发送邮件的更多信息，请参见发送邮件。

有些字段类型需要一些额外的处理。例如，使用表单上传的文件需要不同地处理（它们可以从 `request.FILES` 获取，而不是 `request.POST`）。如何使用表单处理文件上传的更多细节，请参见绑定上传的文件到一个表单。

使用表单模板

你需要做的就是将表单实例放进模板的上下文。如果你的表单在 `Contex t` 中叫做 `form`，那么 `{{ form }}` 将正确地渲染它的 `<label>` 和 `<input>` 元素。

表单渲染的选项

> 表单模板的额外标签
>
> 不要忘记，表单的输出不 包含 `<form>` 标签，和表单的 `submit` 按钮。你必须自己提供它们。

对于 `<label>/<input>` 对，还有几个输出选项：

- `{{ form.as_table }}` 以表格的形式将它们渲染在 `<tr>` 标签中
- `{{ form.as_p }}` 将它们渲染在 `<p>` 标签中
- `{{ form.as_ul }}` 将它们渲染在 `<li>` 标签中

注意，你必须自己提供 `<table>` 或 `<ul>` 元素。

下面是我们的 `ContactForm` 实例的输出 `{{ form.as_p }}`：

```
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
    <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
    <input type="email" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

注意，每个表单字段具有一个 `ID` 属性并设置为 `id_<field-name>`，它被一起的 `label` 标签引用。它对于确保屏幕阅读软件这类的辅助计算非常重要。你还可以自定义label 和 id 生成的方式。

更多信息参见 输出表单为HTML。

手工渲染字段

我们没有必要非要让Django 来分拆表单的字段；如果我们喜欢，我们可以手工来做（例如，这样允许重新对字段排序）。每个字段都是表单的一个属性，可以使用 `{{ form.name_of_field }}` 访问，并将在Django 模板中正确地渲染。例如：

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

完整的 `<label>` 元素还可以使用 `label_tag()` 生成。例如：

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

渲染表单的错误信息

当然，这个便利性的代价是更多的工作。直到现在，我们没有担心如何展示错误信息，因为 Django 已经帮我们处理好。在下面的例子中，我们将自己处理每个字段的错误和表单整体的各种错误。注意，表单和模板顶部的 `{{ form.non_field_errors }}` 查找每个字段的错误。

使用 `{{ form.name_of_field.errors }}` 显示表单错误的一个清单，并渲染成一个 `ul`。看上去可能像：

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

这个 `ul` 有一个 `errorlist` CSS 类型，你可以用它来定义外观。如果你希望进一步自定义错误信息的显示，你可以迭代它们来实现：

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

空字段错误（以及使用 `form.as_p()` 时渲染的隐藏字段错误）将渲染成一个额外的CSS 类型 `nonfield` 以帮助区分每个字段的错误信息。例如， `{{ form.non_field_errors }}` 看上去会像：

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

```
Changed in Django 1.8:

添加上面示例中提到的nonfield CSS 类型。
```

参见Forms API 以获得关于错误、样式以及在模板中使用表单属性的更多内容。

迭代表单的字段

如果你为你的表单使用相同的HTML，你可以使用 `{% for %}` 循环迭代每个字段来减少重复的代码：

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

`{{ field }}` 中有用的属性包括：

`{{ field.label }}`

字段的 `label`，例如 `Email address`。

`{{ field.label_tag }}`

包含在HTML `<label>` 标签中的字段 `Label`。它包含表单的 `label_suffix`。例如，默认的 `label_suffix` 是一个冒号：

```
<label for="id_email">Email address:</label>
```

`{{ field.id_for_label }}`

用于这个字段的 `ID`（在上面的例子中是 `id_email`）。如果你正在手工构造 `label`，你可能想使用它代替 `label_tag`。如果你有一些内嵌的JavaScript 并且想避免硬编码字段的 `ID`，这也是有用的。

`{{ field.value }}`

字段的值，例如 `someone@example.com`。

`{{ field.html_name }}`

输入元素的 `name` 属性中将使用的名称。它将考虑到表单的前缀。

`{{ field.help_text }}`

与该字段关联的帮助文档。

`{{ field.errors }}`

输出一个 `<ul class="errorlist">`，包含这个字段的验证错误信息。你可以使用 `{% for error in field.errors %}` 自定义错误的显示。 这种情况下，循环中的每个对象只是一个包含错误信息的简单字符串。

`{{ field.is_hidden }}`

如果字段是隐藏字段，则为 `True`，否则为 `False`。作为模板变量，它不是很有用处，但是可以用于条件测试，例如：

```
{% if field.is_hidden %}

{% endif %}
```

`{{ field.field }}`

表单类中的 `Field` 实例，通过 `BoundField` 封装。你可以使用它来访问 `Field` 属性，例如 `{% char_field.field.max_length %}`。

迭代隐藏和可见的字段

如果你正在手工布局模板中的一个表单，而不是依赖Django 默认的表单布局，你可能希望将 `<input type="hidden">` 字段与非隐藏的字段区别对待。例如，因为隐藏的字段不会显示，在该字段旁边放置错误信息可能让你的用户感到困惑 —— 所以这些字段的错误应该有区别地来处理。

Django 提供两个表单方法，它们允许你独立地在隐藏的和可见的字段上迭代：`hidden_fields()` 和 `visible_fields()`。下面是使用这两个方法对前面一个例子的修改：

```
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}

{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

这个示例没有处理隐藏字段中的任何错误信息。通常，隐藏字段中的错误意味着表单被篡改，因为正常的表单填写不会改变它们。然而，你也可以很容易地为这些表单错误插入一些错误信息显示出来。

可重用的表单模板

如果你的网站在多个地方对表单使用相同的渲染逻辑，你可以保存表单的循环到一个单独的模板中来减少重复，然后在其它模板中使用 `include` 标签来重用它：

```
# In your form template:
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

如果传递到模板上下文中的表单对象具有一个不同的名称，你可以使用 `include` 标签的 `with` 参数来对它起个别名：

```
{% include "form_snippet.html" with form=comment_form %}
```

如果你发现自己经常这样做，你可能需要考虑一下创建一个自定义的 `inclusion` 标签。

更深入的主题

这里只是基础，表单还可以完成更多的工作：

- 表单集
  - 在表单集中使用初始化数据
  - 限制表单的最大数目
  - 表单集的验证
  - 验证表单集中表单的数目
  - 处理表单的排序和删除
  - 添加额外的字段到表单中
  - 在视图和模板中视图表单集
- 从模型中创建表单
  - ModelForm
  - 模型表单集
  - Inline formsets
- 表单集（`Media` 类）
  - Assets as a static definition
  - `Media` as a dynamic property
  - Paths in asset definitions
  - `Media` 对象
  - 表单中的 `Media`

> 另见
>
> 表单参考 覆盖完整的API 参考，包括表单字段、表单Widget 以及表单和字段的验证。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

表单 API

关于这篇文档

这篇文档讲述Django 表单API 的详细细节。你应该先阅读表单简介。

绑定的表单和未绑定的表单

`表单` 要么是绑定的，要么是未绑定的。

- 如果是绑定的，那么它能够验证数据，并渲染表单及其数据成HTML。
- 如果是未绑定的，那么它不能够完成验证（因为没有可验证的数据！），但是仍然能渲染空白的表单成HTML。

class `Form`

若要创建一个未绑定的 `表单` 实例，只需简单地实例化该类：

```
>>> f = ContactForm()
```

若要绑定数据到表单，可以将数据以字典的形式传递给 `表单` 类的构造函数的第一个参数：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
```

在这个字典中，键为字段的名称，它们对应于 `表单` 类中的属性。值为需要验证的数据。它们通常为字符串，但是没有强制要求必须是字符串；传递的数据类型取决于 `字段`，我们稍后会看到。

`Form.``is_bound`

如果运行时刻你需要区分绑定的表单和未绑定的表单，可以检查下表单 `is_bound` 属性的值：

```
>>> f = ContactForm()
>>> f.is_bound
False
>>> f = ContactForm({'subject': 'hello'})
>>> f.is_bound
True
```

注意，传递一个空的字典将创建一个带有空数据的绑定的表单：

```
>>> f = ContactForm({})
>>> f.is_bound
True
```

如果你有一个绑定的 `表单` 实例但是想改下数据，或者你想绑定一个未绑定的 `表单` 表单到某些数据，你需要创建另外一个 `表单` 实例。 `Form` 实例的数据没有办法修改。 `表单` 实例一旦创建，你应该将它的数据视为不可变的，无论它有没有数据。

使用表单来验证数据

`Form.``clean` ()

当你需要为相互依赖的字段添加自定义的验证时，你可以实现 `表单` 的 `clean()` 方法。示例用法参见*Cleaning and validating fields that depend on each other*。

`Form.``is_valid` ()

`表单` 对象的首要任务就是验证数据。对于绑定的 `表单` 实例，可以调用 `is_valid()` 方法来执行验证并返回一个表示数据是否合法的布尔值。

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

让我们试下非法的数据。下面的情形中， `subject` 为空（默认所有字段都是必需的）且 `sender` 是一个不合法的邮件地址：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

`Form.``errors`

访问 `errors` 属性可以获得错误信息的一个字典：

```
>>> f.errors
{'sender': ['Enter a valid email address.'], 'subject': ['This field is required.']}
```

在这个字典中，键为字段的名称，值为表示错误信息的Unicode 字符串组成的列表。错误信息保存在列表中是因为字段可能有多个错误信息。

你可以在调用 `is_valid()` 之前访问 `errors` 。表单的数据将在第一次调用 `is_valid()` 或者访问 `errors` 时验证。

验证将值调用一次，无论你访问 `errors` 或者调用 `is_valid()` 多少次。这意味着，如果验证过程有副作用，这些副作用将只触发一次。

`Form.errors.``as_data` ()

New in Django 1.7.

返回一个 `字典` ，它映射字段到原始的 `ValidationError` 实例。

```
>>> f.errors.as_data()
{'sender': [ValidationError(['Enter a valid email address.'])],
'subject': [ValidationError(['This field is required.'])]}
```

每当你需要根据错误的 `code` 来识别错误时，可以调用这个方法。它可以用来重写错误信息或者根据特定的错误编写自定义的逻辑。它还可以用来序列化错误为一个自定义的格式（例如，XML）；`as_json()` 就依赖于 `as_data()` 。

需要 `as_data()` 方法是为了向后兼容。以前， `ValidationError` 实例在它们渲染后 的错误消息一旦添加到 `Form.errors` 字典就立即被丢弃。理想情况下， `Form.errors` 应该已经保存 `ValidationError` 实例而带有 `as_` 前缀的方法可以渲染它们，但是为了不破坏直接使用 `Form.errors` 中的错误消息的代码，必须使用其它方法来实现。

`Form.errors.``as_json` (*escape_html=False*)

New in Django 1.7.

返回JSON 序列化后的错误。

```
>>> f.errors.as_json()
{"sender": [{"message": "Enter a valid email address.", "code": "invalid"}],
"subject": [{"message": "This field is required.", "code": "required"}]}
```

默认情况下， `as_json()` 不会转义它的输出。如果你正在使用AJAX 请求表单视图，而客户端会解析响应并将错误插入到页面中，你必须在客户端对结果进行转义以避免可能的跨站脚本攻击。使用一个JavaScript 库比如jQuery 来做这件事很简单 —— 只要使用 `$(el).text(errorText)` 而不是 `.html()` 就可以。

如果由于某种原因你不想使用客户端的转义，你还可以设置 `escape_html=True` ，这样错误消息将被转义而你可以直接在HTML 中使用它们。

`Form.``add_error` (*field*, *error*)

New in Django 1.7.

这个方法允许在 `Form.clean()` 方法内部或从表单的外部一起给字段添加错误信息；例如从一个视图中。

`field` 参数为字段的名称。如果值为 `None`，error 将作为 `Form.non_field_errors()` 返回的一个非字段错误。

`error` 参数可以是一个简单的字符串，或者最好是一个 `ValidationError` 实例。引发 *ValidationError* 中可以看到定义表单错误时的最佳实践。

注意，`Form.add_error()` 会自动删除 `cleaned_data` 中的相关字段。

`Form.``has_error` (*field*, *code=None*)

New in Django 1.8.

这个方法返回一个布尔值，指示一个字段是否具有指定错误 `code` 的错误。当 `code` 为 `None` 时，如果字段有任何错误它都将返回 `True`。

若要检查非字段错误，使用 `NON_FIELD_ERRORS` 作为 `field` 参数。

`Form.``non_field_errors` ()

这个方法返回 `Form.errors` 中不是与特定字段相关联的错误。它包含在 `Form.clean()` 中引发的 `ValidationError` 和使用 `Form.add_error(None, "...")` 添加的错误。

未绑定表单的行为

验证没有绑定数据的表单是没有意义的，下面的例子展示了这种情况：

```
>>> f = ContactForm()
>>> f.is_valid()
False
>>> f.errors
{}
```

动态的初始值

`Form.``initial`

表单字段的初始值使用 `initial` 声明。例如，你可能希望使用当前会话的用户名填充 `username` 字段。

使用 `Form` 的 `initial` 参数可以实现。该参数是字段名到初始值的一个字典。只需要包含你期望给出初始值的字段；不需要包含表单中的所有字段。例如：

```
>>> f = ContactForm(initial={'subject': 'Hi there!'})
```

这些值只显示在没有绑定的表单中，即使没有提供特定值它们也不会作为后备的值。

注意，如果 字段 有定义 initial ， 而实例化 表单 时也提供 initial ，那么后面的 initial 将优先。在下面的例子中， initial 在字段和表单实例化中都有定义，此时后者具有优先权：

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='class')
...     url = forms.URLField()
...     comment = forms.CharField()
>>> f = CommentForm(initial={'name': 'instance'}, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="instance" /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

检查表单数据是否改变

Form.``has_changed ()

当你需要检查表单的数据是否从初始数据发生改变时，可以使用 表单 的 has_changed() 方法。

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data, initial=data)
>>> f.has_changed()
False
```

当提交表单时，我们可以重新构建表单并提供初始值，这样可以实现比较：

```
>>> f = ContactForm(request.POST, initial=data)
>>> f.has_changed()
```

如果 request.POST 中的数据与 initial 中的不同， has_changed() 将为 True ，否则为 False 。 计算的结果是通过调用表单每个字段的 Field.has_changed() 得到的。

从表单中访问字段

Form.``fields

你可以从 表单 实例的 fields 属性访问字段：

```
>>> for row in f.fields.values(): print(row)
...
<django.forms.fields.CharField object at 0x7ffaac632510>
<django.forms.fields.URLField object at 0x7ffaac632f90>
<django.forms.fields.CharField object at 0x7ffaac3aa050>
>>> f.fields['name']
<django.forms.fields.CharField object at 0x7ffaac6324d0>
```

可你可以修改 `表单` 实例的字段来改变字段在表单中的表示：

```
>>> f.as_table().split('\n')[0]
'<tr><th>Name:</th><td><input name="name" type="text" value="instance" /></td></tr>'
>>> f.fields['name'].label = "Username"
>>> f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value="instance" /></td></tr>'
```

注意不要改变 `base_fields` 属性，因为一旦修改将影响同一个Python 进程中接下来所有的 `ContactForm` 实例：

```
>>> f.base_fields['name'].label = "Username"
>>> another_f = CommentForm(auto_id=False)
>>> another_f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value="class" /></td></tr>'
```

访问“清洁”的数据

`Form.``cleaned_data`

`表单` 类中的每个字段不仅负责验证数据，还负责“清洁”它们 —— 将它们转换为正确的格式。这是个非常好用的功能，因为它允许字段以多种方式输入数据，并总能得到一致的输出。

例如， `DateField` 将输入转换为Python 的 `datetime.date` 对象。无论你传递的是 `'1994-07-15'` 格式的字符串、 `datetime.date` 对象、还是其它格式的数字， `DateField` 将始终将它们转换成 `datetime.date` 对象，只要它们是合法的。

一旦你创建一个 `表单` 实例并通过验证后，你就可以通过它的 `cleaned_data` 属性访问清洁的数据：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@example.com', 'subject': 'hello
```

注意，文本字段 —— 例如， `CharField` 和 `EmailField` —— 始终将输入转换为Unicode 字符串。我们将在这篇文档的后面将是编码的影响。

如果你的数据没有 通过验证， `cleaned_data` 字典中只包含合法的字段：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there'}
```

`cleaned_data` 始终只 包含 `表单` 中定义的字段，即使你在构建 `表单` 时传递了额外的数据。在下面的例子中，我们传递一组额外的字段给 `ContactForm` 构造函数，但是 `cleaned_data` 将只包含表单的字段：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True,
...         'extra_field_1': 'foo',
...         'extra_field_2': 'bar',
...         'extra_field_3': 'baz'}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data # Doesn't contain extra_field_1, etc.
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@example.com', 'subject': 'hello
```

当 `表单` 合法时， `cleaned_data` 将包含所有字段的键和值，即使传递的数据不包含某些可选字段的值。在下面的例子中，传递的数据字典不包含 `nick_name` 字段的值，但是 `cleaned_data` 任然包含它，只是值为空：

```
>>> from django.forms import Form
>>> class OptionalPersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
...     nick_name = CharField(required=False)
>>> data = {'first_name': 'John', 'last_name': 'Lennon'}
>>> f = OptionalPersonForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'nick_name': '', 'first_name': 'John', 'last_name': 'Lennon'}
```

在上面的例子中， `cleaned_data` 中 `nick_name` 设置为一个空字符串，这是因为 `nick_name` 是 `CharField` 而 `CharField` 将空值作为一个空字符串。每个字段都知道自己的“空”值 —— 例如， `DateField` 的空值是 `None` 而不是一个空字符串。关于每个字段空值的完整细节，参见“内建的 `Field` 类”一节中每个字段的“空值”提示。

你可以自己编写代码来对特定的字段（根据它们的名字）或者表单整体（考虑到不同字段的组合）进行验证。更多信息参见表单和字段验证。

输出表单为HTML

`表单` 对象的第二个任务是将它渲染成HTML。很简单， `print` 它：

```
>>> f = ContactForm()
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="tex
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender"
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name=
```

如果表单是绑定的，输出的HTML 将包含数据。例如，如果字段是 `<input type="text">` 的形式，其数据将位于 `value` 属性中。如果字段是 `<input type="checkbox">` 的形式，HTML 将包含 `checked="checked"` ：

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="tex
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender"
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name=
```

默认的输出时具有两个列的HTML 表格，每个字段对应一个 `<tr>` 。注意事项：

- 为了灵活性，输出不包含 `<table>` 和 `</table>` 、 `<form>` 和 `</form>` 以及 `<input type="submit">` 标签。你需要添加它们。
- 每个字段类型有一个默认的HTML 表示。 `CharField` 表示为一个 `<input type="text">` ， `EmailField` 表示为一个 `<input type="email">` 。 `BooleanField` 表示为一个 `<input type="checkbox">` 。注意，这些只是默认的表示；你可以使用Widget 指定字段使用哪种HTML，我们将稍后解释。
- 每个标签的HTML `name` 直接从 `ContactForm` 类中获取。
- 每个字段的文本标签 —— 例如 `'Subject:'` 、 `'Message:'` 和 `'Cc myself:'` 通过将所有的下划线转换成空格并大写第一个字母生成。再次提醒，这些只是默认的表示；你可以手工指定标签。
- 每个文本标签周围有一个HTML `<label>` 标签，它指向表单字段的 `id` 。这个 `id` ，是通过在字段名称前面加上 `'id_'` 前缀生成。 `id` 属性和 `<label>` 标签默认包含在输出中，但你可以改变这一行为。

虽然 `print` 表单时 `<table>` 是默认的输出格式，但是还有其它格式可用。每个格式对应于表单对象的一个方法，每个方法都返回一个Unicode 对象。

as_p()

`Form.``as_p` ()

`as_p()` 渲染表单为一系列的 `<p>` 标签，每个 `<p>` 标签包含一个字段：

```
>>> f = ContactForm()
>>> f.as_p()
'<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="sub
>>> print(f.as_p())
<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subj
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_mess
<p><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender
<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" i
```

as_ul()

`Form.``as_ul` ()

`as_ul()` 渲染表单为一系列的 `<li>` 标签，每个 `<li>` 标签包含一个字段。它不包含 `<ul>` 和 `</ul>`，所以你可以自己指定 `<ul>` 的任何HTML 属性：

```
>>> f = ContactForm()
>>> f.as_ul()
'<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="su
>>> print(f.as_ul())
<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="sub
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_mes
<li><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sende
<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself"
```

as_table()

`Form.``as_table` ()

最后， `as_table()` 输出表单为一个HTML `<table>` 。它与 `print` 完全相同。事实上，当你 `print` 一个表单对象时，在后台调用的就是 `as_table()` 方法：

```
>>> f = ContactForm()
>>> f.as_table()
'<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="te
>>> print(f.as_table())
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="tex
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender"
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name=
```

表单必填行和错误行的样式

`Form.``error_css_class`

`Form.``required_css_class`

将必填的表单行和有错误的表单行定义不同的样式特别常见。例如，你想将必填的表单行以粗体显示、将错误以红色显示。

`表单` 类具有一对钩子，可以使用它们来添加 `class` 属性给必填的行或有错误的行：只需简单地设置 `Form.error_css_class` 和/或 `Form.required_css_class` 属性：

```
from django.forms import Form

class ContactForm(Form):
    error_css_class = 'error'
    required_css_class = 'required'

    # ... and the rest of your fields here
```

一旦你设置好，将根据需要设置行的 `"error"` 和/或 `"required"` CSS 类型。 其HTML 看上去将类似：

```
>>> f = ContactForm(data)
>>> print(f.as_table())
<tr class="required"><th><label class="required" for="id_subject">Subject:</label>    ...
<tr class="required"><th><label class="required" for="id_message">Message:</label>    ...
<tr class="required error"><th><label class="required" for="id_sender">Sender:</label>
<tr><th><label for="id_cc_myself">Cc myself:<label> ...
>>> f['subject'].label_tag()
<label class="required" for="id_subject">Subject:</label>
>>> f['subject'].label_tag(attrs={'class': 'foo'})
<label for="id_subject" class="foo required">Subject:</label>
```

Changed in Django 1.8:

`required_css_class` 添加到 `<label>` 标签，如上面所看到的。

配置表单元素的HTML `id` 属性和 `<label>` 标签

`Form.``auto_id`

默认情况下，表单的渲染方法包含：

- 表单元素的HTML `id` 属性
- 对应的 `<label>` 标签。HTML `<label>` 标签指示标签文本关联的表单元素。这个小小的改进让表单在辅助设备上具有更高的可用性。使用 `<label>` 标签始终是个好想法。

`id` 属性值通过在表单字段名称的前面加上 `id_` 生成。但是如果你想改变 `id` 的生成方式或者完全删除 HTML `id` 属性和 `<label>` 标签，这个行为是可配置的。

`id` 和label 的行为使用 `表单` 构造函数的 `auto_id` 参数控制。这个参数必须为 `True`、 `False` 或者一个字符串。

如果 `auto_id` 为 `False`，那么表单的输出将不包含 `<label>` 标签和 `id` 属性：

```
>>> f = ContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="email" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="email" name="sender" /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

如果 `auto_id` 设置为 `True`，那么输出的表示将 包含 `<label>` 标签并简单地使用字典名称作为每个表单字段的 `id`：

```
>>> f = ContactForm(auto_id=True)
>>> print(f.as_table())
<tr><th><label for="subject">Subject:</label></th><td><input id="subject" type="text" nam
<tr><th><label for="message">Message:</label></th><td><input type="text" name="message" i
<tr><th><label for="sender">Sender:</label></th><td><input type="email" name="sender" id=
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc
>>> print(f.as_ul())
<li><label for="subject">Subject:</label> <input id="subject" type="text" name="subject"
<li><label for="message">Message:</label> <input type="text" name="message" id="message"
<li><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" /><
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id=
>>> print(f.as_p())
<p><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" m
<p><label for="message">Message:</label> <input type="text" name="message" id="message" /
<p><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" /></
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="
```

如果 `auto_id` 设置为包含格式字符 `'%s'` 的字符串，那么表单的输出将包含 `<label>` 标签，并将根据格式字符串生成 `id` 属性。例如，对于格式字符串 `'field_%s'`，名为 `subject` 的字段的 `id` 值将是 `'field_subject'`。继续我们的例子：

```
>>> f = ContactForm(auto_id='id_for_%s')
>>> print(f.as_table())
<tr><th><label for="id_for_subject">Subject:</label></th><td><input id="id_for_subject" t
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text" name="mes
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="email" name="send
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type="checkbox" n
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" n
<li><label for="id_for_message">Message:</label> <input type="text" name="message" id="id
<li><label for="id_for_sender">Sender:</label> <input type="email" name="sender" id="id_f
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myse
>>> print(f.as_p())
<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" na
<p><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_
<p><label for="id_for_sender">Sender:</label> <input type="email" name="sender" id="id_fo
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_mysel
```

如果 `auto_id` 设置为任何其它的真值 —— 例如不包含 `%s` 的字符串 —— 那么其行为将类似 `auto_id` 等于 `True`。

默认情况下， `auto_id` 设置为 `'id_%s'`。

`Form.``label_suffix`

一个字符串（默认为英文的 `:`），表单渲染时将附加在每个label 名称的后面。

使用 `label_suffix` 参数可以自定义这个字符，或者完全删除它：

```
>>> f = ContactForm(auto_id='id_for_%s', label_suffix='')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject</label> <input id="id_for_subject" type="text" na
<li><label for="id_for_message">Message</label> <input type="text" name="message" id="id_
<li><label for="id_for_sender">Sender</label> <input type="email" name="sender" id="id_fo
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name="cc_mysel
>>> f = ContactForm(auto_id='id_for_%s', label_suffix=' ->')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject -></label> <input id="id_for_subject" type="text"
<li><label for="id_for_message">Message -></label> <input type="text" name="message" id="
<li><label for="id_for_sender">Sender -></label> <input type="email" name="sender" id="id
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name="cc_my
```

注意，该标签后缀只有当label 的最后一个字符不是表单符号（`.`，`!`，`?` 和 `:`）时才添加。

New in Django 1.8.

字段可以定义自己的 `label_suffix`。而且将优先于 `Form.label_suffix`。在运行时刻，后缀可以使用 `label_tag()` 的 `label_suffix` 参数覆盖。

字段的顺序

在 `as_p()`、`as_ul()` 和 `as_table()` 中，字段以表单类中定义的顺序显示。例如，在 `ContactForm` 示例中，字段定义的顺序为 `subject`，`message`，`sender`，`cc_myself`。若要重新排序HTML 中的输出，只需改变字段在类中列出的顺序。

错误如何显示

如果你渲染一个绑定的 `表单` 对象，渲染时将自动运行表单的验证，HTML 输出将在出错字段的附近以 `<ul class="errorlist">` 形式包含验证的错误。错误信息的位置与你使用的输出方法有关：

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><ul class="errorlist"><li>This field is required.</li></ul><inpu
<tr><th>Message:</th><td><input type="text" name="message" value="Hi there" /></td></tr>
<tr><th>Sender:</th><td><ul class="errorlist"><li>Enter a valid email address.</li></ul><
<tr><th>Cc myself:</th><td><input checked="checked" type="checkbox" name="cc_myself" /></
>>> print(f.as_ul())
<li><ul class="errorlist"><li>This field is required.</li></ul>Subject: <input type="text
<li>Message: <input type="text" name="message" value="Hi there" /></li>
<li><ul class="errorlist"><li>Enter a valid email address.</li></ul>Sender: <input type="
<li>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p><ul class="errorlist"><li>This field is required.</li></ul></p>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<p><ul class="errorlist"><li>Enter a valid email address.</li></ul></p>
<p>Sender: <input type="email" name="sender" value="invalid email address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

自定义错误清单的格式

默认情况下，表单使用 `django.forms.utils.ErrorList` 来格式化验证时的错误。如果你希望使用另外一种类来显示错误，可以在构造时传递（在Python 2 中将 `__str__` 替换为 `__unicode__`）：

```
>>> from django.forms.utils import ErrorList
>>> class DivErrorList(ErrorList):
...     def __str__(self):              # __unicode__ on Python 2
...         return self.as_divs()
...     def as_divs(self):
...         if not self: return ''
...         return '<div class="errorlist">%s</div>' % ''.join(['<div class="error">%s</d
>>> f = ContactForm(data, auto_id=False, error_class=DivErrorList)
>>> f.as_p()
<div class="errorlist"><div class="error">This field is required.</div></div>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<div class="errorlist"><div class="error">Enter a valid email address.</div></div>
<p>Sender: <input type="email" name="sender" value="invalid email address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Changed in Django 1.7:

`django.forms.util` 重命名为 `django.forms.utils` 。

更细粒度的输出

`as_p()` 、 `as_ul()` 和 `as_table()` 方法是为懒惰的程序员准备的简单快捷方法 —— 它们不是显示表单的唯一方式。

class `BoundField`

用于显示HTML 表单或者访问 `表单` 实例的一个属性。

其 `__str__()` （Python 2 上为 `__unicode__` ）方法显示该字段的HTML。

以字段的名称为键，用字典查询语法查询表单，可以获取一个 `BoundField` ：

```
>>> form = ContactForm()
>>> print(form['subject'])
<input id="id_subject" type="text" name="subject" maxlength="100" />
```

迭代表单可以获取所有的 `BoundField` ：

```
>>> form = ContactForm()
>>> for boundfield in form: print(boundfield)
<input id="id_subject" type="text" name="subject" maxlength="100" />
<input type="text" name="message" id="id_message" />
<input type="email" name="sender" id="id_sender" />
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

字段的输出与表单的 `auto_id` 设置有关：

```
>>> f = ContactForm(auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f = ContactForm(auto_id='id_%s')
>>> print(f['message'])
<input type="text" name="message" id="id_message" />
```

若要获取字段的错误列表，可以访问字段的 `errors` 属性。

`BoundField.``errors`

一个类列表对象，打印时以HTML `<ul class="errorlist">` 形式显示：

```
>>> data = {'subject': 'hi', 'message': '', 'sender': '', 'cc_myself': ''}
>>> f = ContactForm(data, auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f['message'].errors
['This field is required.']
>>> print(f['message'].errors)
<ul class="errorlist"><li>This field is required.</li></ul>
>>> f['subject'].errors
[]
>>> print(f['subject'].errors)

>>> str(f['subject'].errors)
''
```

`BoundField.``label_tag` (*contents=None*, *attrs=None*, *label_suffix=None*)

可以调用 `label_tag` 方法单独渲染表单字段的label 标签：

```
>>> f = ContactForm(data)
>>> print(f['message'].label_tag())
<label for="id_message">Message:</label>
```

如果你提供一个可选的 `contents` 参数，它将替换自动生成的label 标签。另外一个可选的 `attrs` 参数可以包含 `<label>` 标签额外的属性。

生成的HTML 包含表单的 `label_suffix` （默认为一个冒号），或者当前字段的 `label_suffix` 。可选的 `label_suffix` 参数允许你覆盖之前设置的后缀。例如，你可以使用一个空字符串来隐藏已选择字段的label。如果在模板中需要这样做，你可以编写一个自定义的过滤器来允许传递参数给 `label_tag` 。

Changed in Django 1.8:

如果可用，label 将包含 `required_css_class` 。

`BoundField.``css_classes` ()

当你使用Django 的快捷的渲染方法时，习惯使用CSS 类型来表示必填的表单字段和有错误的字段。如果你是手工渲染一个表单，你可以使用 `css_classes` 方法访问这些CSS 类型：

```
>>> f = ContactForm(data)
>>> f['message'].css_classes()
'required'
```

除了错误和必填的类型之外，如果你还想提供额外的类型，你可以用参数传递它们：

```
>>> f = ContactForm(data)
>>> f['message'].css_classes('foo bar')
'foo bar required'
```

`BoundField.``value` ()

这个方法用于渲染字段的原始值，与用 `Widget` 渲染的值相同：

```
>>> initial = {'subject': 'welcome'}
>>> unbound_form = ContactForm(initial=initial)
>>> bound_form = ContactForm(data, initial=initial)
>>> print(unbound_form['subject'].value())
welcome
>>> print(bound_form['subject'].value())
hi
```

`BoundField.``id_for_label`

使用这个属性渲染字段的ID。例如，如果你在模板中手工构造一个 `<label>`（尽管 `label_tag()` 将为你这么做）：

```
<label for="{{ form.my_field.id_for_label }}">...</label>{{ my_field }}
```

默认情况下，它是在字段名称的前面加上 `id_`（上面的例子中将是“ `id_my_field` ”）。你可以通过设置字段Widget 的 `attrs` 来修改ID。例如，像这样声明一个字段：

```
my_field = forms.CharField(widget=forms.TextInput(attrs={'id': 'myFIELD'}))
```

使用上面的模板，将渲染成：

```
<label for="myFIELD">...</label><input id="myFIELD" type="text" name="my_field" />
```

绑定上传的文件到表单

处理带有 `FileField` 和 `ImageField` 字段的表单比普通的表单要稍微复杂一点。

首先，为了上传文件，你需要确保你的 `<form>` 元素正确定义 `enctype` 为 `"multipart/form-data"`：

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

其次，当你使用表单时，你需要绑定文件数据。文件数据的处理与普通的表单数据是分开的，所以如果表单包含 `FileField` 和 `ImageField`，绑定表单时你需要指定第二个参数。所以，如果我们扩展ContactForm 并包含一个名为 `mugshot` 的 `ImageField`，我们需要绑定包含 mugshot 图片的文件数据：

```
# Bound form with an image field
>>> from django.core.files.uploadedfile import SimpleUploadedFile
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> file_data = {'mugshot': SimpleUploadedFile('face.jpg', <file data>)}
>>> f = ContactFormWithMugshot(data, file_data)
```

实际上，你一般将使用 `request.FILES` 作为文件数据的源（和使用 `request.POST` 作为表单数据的源一样）：

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithMugshot(request.POST, request.FILES)
```

构造一个未绑定的表单和往常一样 —— 将表单数据和文件数据同时省略：

```
# Unbound form with an image field
>>> f = ContactFormWithMugshot()
```

测试**multipart** 表单

`Form.``is_multipart` ()

如果你正在编写可重用的视图或模板，你可能事先不知道你的表单是否是一个multipart 表单。 `is_multipart()` 方法告诉你表单提交时是否要求multipart：

```
>>> f = ContactFormWithMugshot()
>>> f.is_multipart()
True
```

下面是如何在模板中使用它的一个示例：

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="/foo/">
{% else %}
    <form method="post" action="/foo/">
{% endif %}
{{ form }}
</form>
```

子类化表单

如果你有多个 `表单` 类共享相同的字段，你可以使用子类化来减少冗余。

当你子类化一个自定义的 `表单` 类时，生成的子类将包含父类中的所有字段，以及在子类中定义的字段。

在下面的例子中，`ContactFormWithPriority` 包含 `ContactForm` 中的所有字段，以及另外一个字段 `priority`。排在前面的是 `ContactForm` 中的字段：

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="email" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
<li>Priority: <input type="text" name="priority" /></li>
```

可以子类化多个表单，将表单作为“mix-ins”。在下面的例子中，`BeatleForm` 子类化 `PersonForm` 和 `InstrumentForm`，所以它的字段列表包含两个父类的所有字段：

```
>>> from django.forms import Form
>>> class PersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
>>> class InstrumentForm(Form):
...     instrument = CharField()
>>> class BeatleForm(PersonForm, InstrumentForm):
...     haircut_type = CharField()
>>> b = BeatleForm(auto_id=False)
>>> print(b.as_ul())
<li>First name: <input type="text" name="first_name" /></li>
<li>Last name: <input type="text" name="last_name" /></li>
<li>Instrument: <input type="text" name="instrument" /></li>
<li>Haircut type: <input type="text" name="haircut_type" /></li>
```

New in Django 1.7.

- 在子类中，可以通过设置名字为 `None` 来删除从父类中继承的 `字段`。例如：

  ```
  >>> from django import forms

  >>> class ParentForm(forms.Form):
  ...     name = forms.CharField()
  ...     age = forms.IntegerField()

  >>> class ChildForm(ParentForm):
  ...     name = None

  >>> ChildForm().fields.keys()
  ... ['age']
  ```

表单前缀

`Form.``prefix`

你可以将几个Django 表单放在一个 `<form>` 标签中。为了给每个 `表单` 一个自己的命名空间，可以使用 `prefix` 关键字参数：

```
>>> mother = PersonForm(prefix="mother")
>>> father = PersonForm(prefix="father")
>>> print(mother.as_ul())
<li><label for="id_mother-first_name">First name:</label> <input type="text" name="mother
<li><label for="id_mother-last_name">Last name:</label> <input type="text" name="mother-l
>>> print(father.as_ul())
<li><label for="id_father-first_name">First name:</label> <input type="text" name="father
<li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-l
```

Widgets

Widget 是Django 对HTML 输入元素的表示。Widget 负责渲染HTML和提取GET/POST 字典中的数据。

小贴士

不要将Widget 与表单字段搞混淆。表单字段负责验证输入并直接在模板中使用。Widget 负责渲染网页上HTML 表单的输入元素和提取提交的原始数据。但是，Widget 需要赋值给表单的字段。

指定Widget

每当你指定表单的一个字段的时候，Django 将使用适合其数据类型的默认Widget。若要查找每个字段使用的Widget，参见内建的字段文档。

然而，如果你想要使用一个不同的Widget，你可以在定义字段时使用 `widget` 参数。例如：

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

这将使用一个 `Textarea` Widget来设置表单的评论，而不是默认的 `TextInput` Widget。

设置Widget 的参数

很多Widget 都有可选的参数；它们可以在定义字段的Widget 时设置。在下面的示例中，设置了 `SelectDateWidget` 的 `years` 属性：

```
from django import forms
from django.forms.extras.widgets import SelectDateWidget

BIRTH_YEAR_CHOICES = ('1980', '1981', '1982')
FAVORITE_COLORS_CHOICES = (('blue', 'Blue'),
                           ('green', 'Green'),
                           ('black', 'Black'))

class SimpleForm(forms.Form):
    birth_year = forms.DateField(widget=SelectDateWidget(years=BIRTH_YEAR_CHOICES))
    favorite_colors = forms.MultipleChoiceField(required=False,
        widget=forms.CheckboxSelectMultiple, choices=FAVORITE_COLORS_CHOICES)
```

可用的Widget 以及它们接收的参数，参见内建的*Widget*。

继承自Select 的Widget

继承自 `Select` 的Widget 负责处理HTML 选项。它们呈现给用户一个可以选择的选项列表。不同的Widget 以不同的方式呈现选项； `Select` 使用HTML 的列表形式 `<select>`，而 `RadioSelect` 使用单选按钮。

`ChoiceField` 字段默认使用 `Select` 。Widget 上显示的选项来自 `ChoiceField`，对 `ChoiceField.choices` 的改变将更新 `Select.choices` 。例如：

```
>>> from django import forms
>>> CHOICES = (('1', 'First',), ('2', 'Second',))
>>> choice_field = forms.ChoiceField(widget=forms.RadioSelect, choices=CHOICES)
>>> choice_field.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices = ()
>>> choice_field.choices = (('1', 'First and only',),)
>>> choice_field.widget.choices
[('1', 'First and only')]
```

提供 `choices` 属性的Widget 也可以用于不是基于选项的字段， 例如 `CharField` —— 当选项与模型有关而不只是Widget 时，建议使用基于 `ChoiceField` 的字段。

自定义Widget 的实例

当Django 渲染Widget 成HTML 时，它只渲染最少的标记 —— Django 不会添加class 的名称和特定于Widget 的其它属性。这表示，网页上所有 `TextInput` 的外观是一样的。

有两种自定义Widget 的方式：基于每个*Widget 实例*和基于每个*Widget 类*。

设置Widget 实例的样式

如果你想让某个Widget 实例与其它Widget 看上去不一样，你需要在Widget 对象实例化并赋值给一个表单字段时指定额外的属性（以及可能需要在你的CSS 文件中添加一些规则）。

例如下面这个简单的表单：

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

这个表单包含三个默认的 `TextInput` Widget，以默认的方式渲染 —— 没有CSS 类、没有额外的属性。这表示每个Widget 的输入框将渲染得一模一样：

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

在真正得网页中，你可能不想让每个Widget 看上去都一样。你可能想要给comment 一个更大的输入元素，你可能想让'name' Widget 具有一些特殊的CSS 类。可以指定'type' 属性来利用新式的HTML5 输入类型。在创建Widget 时使用 `Widget.attrs` 参数可以实现：

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

Django 将在渲染的输出中包含额外的属性：

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="special"/></td></tr>
<tr><th>Url:</th><td><input type="url" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40"/></td></tr>
```

你还可以使用 `attrs` 设置HTML `id`。参见 `BoundField.id_for_label` 示例。

设置Widget 类的样式

可以添加（`css` 和 `javascript`）给Widget，以及深度定制它们的外观和行为。

概况来讲，你需要子类化Widget 并定义一个*"Media"* 内联类 或 创建一个*"media"* 属性。

这些方法涉及到Python 高级编程，详细细节在表单*Assets* 主题中讲述。

Widget 的基类

`Widget` 和 `MultiWidget` 是所有内建*Widget* 的基类，并可用于自定义Widget 的基类。

class `Widget` (*attrs=None*)

这是个抽象类，它不可以渲染，但是提供基本的属性 `attrs`。你可以在自定义的Widget 中实现或覆盖 `render()` 方法。

`attrs`

包含渲染后的Widget 将要设置的HTML 属性。

```
>>> from django import forms
>>> name = forms.TextInput(attrs={'size': 10, 'title': 'Your name',})
>>> name.render('name', 'A name')
'<input title="Your name" type="text" name="name" value="A name" size="10" />'
```

Changed in Django 1.8:

如果你给一个属性赋值 `True` 或 `False`，它将渲染成一个HTML5 风格的布尔属性：

```
>>> name = forms.TextInput(attrs={'required': True})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" required />'
>>>
>>> name = forms.TextInput(attrs={'required': False})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" />'
```

`render`(*name*, *value*, *attrs=None*)

返回Widget 的HTML，为一个Unicode 字符串。子类必须实现这个方法，否则将引发 `NotImplementedError`。

它不会确保给出的'value' 是一个合法的输入，因此子类的实现应该防卫式地编程。

`value_from_datadict`(*data*, *files*, *name*)

根据一个字典和该Widget 的名称，返回该Widget 的值。`files` may contain data coming from `request.` FILES. 如果没有提供value，则返回 `None`。 在处理表单数据的过程中，`value_from_datadict` 可能调用多次，所以如果你自定义并添加额外的耗时处理时，你应该自己实现一些缓存机制。

class `MultiWidget`(*widgets*, *attrs=None*)

由多个Widget 组合而成的Widget。`MultiWidget` 始终与 `MultiValueField` 联合使用。

`MultiWidget` 具有一个必选参数：

`widgets`

一个包含需要的Widget 的可迭代对象。

以及一个必需的方法：

`decompress`(*value*)

这个方法接受来自字段的一个“压缩”的值，并返回“解压”的值的一个列表。可以假设输入的值是合法的，但不一定是非空的。

子类必须实现 这个方法，而且因为值可能为空，实现必须要防卫这点。

“解压”的基本原理是需要“分离”组合的表单字段的值为每个Widget 的值。

有个例子是， `SplitDateTimeWidget` 将 `datetime` 值分离成两个独立的值分别表示日期和时间：

```
from django.forms import MultiWidget

class SplitDateTimeWidget(MultiWidget):

    # ...

    def decompress(self, value):
        if value:
            return [value.date(), value.time().replace(microsecond=0)]
        return [None, None]
```

小贴士

注意， `MultiValueField` 有一个 `compress()` 方法用于相反的工作 —— 将所有字段的值组合成一个值。

其它可能需要覆盖的方法：

`render` (*name*, *value*, *attrs=None*)

这个方法中的 `value` 参数的处理方式与 `Widget` 子类不同，因为需要弄清楚如何为了在不同widget中展示分割单一值。

渲染中使用的 `value` 参数可以是二者之一：

- 一个 `列表` 。
- 一个单一值（比如字符串），它是 `列表` 的“压缩”表现形式。

如果 `value` 是个列表， `render()` 的输出会是一系列渲染后的子widget。如果 `value` 不是一个列表，首先会通过 `decompress()` 方法来预处理，创建列表，之后再渲染。

`render()` 方法执行HTML渲染时，列表中的每个值都使用相应的widget来渲染 -- 第一个值在第一个widget中渲染，第二个值在第二个widget中渲染，以此类推。

不像单一值的widget， `render()` 方法并不需要在子类中实现。

`format_output` (*rendered_widgets*)

接受选然后的widget（以字符串形式）的一个列表，返回表示全部HTML的Unicode字符串。

这个钩子允许你以任何你想要的方式，格式化widget的HTML设计。

下面示例中的Widget 继承 `MultiWidget` 以在不同的选择框中显示年、月、日。这个Widget 主要想用于 `DateField` 而不是 `MultiValueField` ，所以我们实现了 `value_from_datadict()` ：

```
from datetime import date
from django.forms import widgets

class DateSelectorWidget(widgets.MultiWidget):
    def __init__(self, attrs=None):
        # create choices for days, months, years
        # example below, the rest snipped for brevity.
        years = [(year, year) for year in (2011, 2012, 2013)]
        _widgets = (
            widgets.Select(attrs=attrs, choices=days),
            widgets.Select(attrs=attrs, choices=months),
            widgets.Select(attrs=attrs, choices=years),
        )
        super(DateSelectorWidget, self).__init__(_widgets, attrs)

    def decompress(self, value):
        if value:
            return [value.day, value.month, value.year]
        return [None, None, None]

    def format_output(self, rendered_widgets):
        return ''.join(rendered_widgets)

    def value_from_datadict(self, data, files, name):
        datelist = [
            widget.value_from_datadict(data, files, name + '_%s' % i)
            for i, widget in enumerate(self.widgets)]
        try:
            D = date(day=int(datelist[0]), month=int(datelist[1]),
                     year=int(datelist[2]))
        except ValueError:
            return ''
        else:
            return str(D)
```

构造器在一个元组中创建了多个 `Select` widget。 `超` 类使用这个元组来启动widget。

`format_output()` 方法相当于在这里没有干什么新的事情（实际上，它和 `MultiWidget` 中默认实现的东西相同），但是这个想法是，你可以以自己的方式在widget之间添加自定义的HTML。

必需的 `decompress()` 方法将 `datetime.date` 值拆成年、月和日的值，对应每个widget。注意这个方法如何处理 `value` 为 `None` 的情况。

`value_from_datadict()` 的默认实现会返回一个列表，对应每一个 `Widget` 。当和 `MultiValueField` 一起使用 `MultiWidget` 的时候，这样会非常合理，但是由于我们想要和拥有单一值得 `DateField` 一起使用这个widget，我们必须覆写这一方法，将所有子widget的数据组装成 `datetime.date` 。这个方法从 `POST` 字典中获取数据，并且构造和验证日期。如果日期有效，会返回它的字符串，否则会返回一个空字符串，它会使 `form.is_valid` 返回 `False` 。

内建的Widget

Django 提供所有基本的HTML Widget，并在 `django.forms.widgets` 模块中提供一些常见的 Widget 组，包括文本的输入、各种选择框、文件上传和多值输入。

处理文本输入的Widget

这些Widget 使用HTML 元素 `input` 和 `textarea`。

TextInput

class `TextInput`

文本输入：`<input type="text" ...>`

NumberInput

class `NumberInput`

文本输入：`<input type="number" ...>`

注意，不是所有浏览器的 `number` 输入类型都支持输入本地化的数字。Django 将字段的 `localize` 属性设置为 `True` 以避免字段使用它们。

EmailInput

class `EmailInput`

文本输入：`<input type="email" ...>`

URLInput

class `URLInput`

文本输入：`<input type="url" ...>`

PasswordInput

class `PasswordInput`

密码输入：`<input type='password' ...>`

接收一个可选的参数：

`render_value`

决定在验证错误后重新显示表单时，Widget 是否填充（默认为 `False`）。

HiddenInput

class `HiddenInput`

隐藏的输入：`<input type='hidden' ...>`

注意，还有一个 `MultipleHiddenInput` Widget，它封装一组隐藏的输入元素。

DateInput

class `DateInput`

日期以普通的文本框输入：`<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `DATE_INPUT_FORMATS` 中找到的第一个格式。

DateTimeInput

class `DateTimeInput`

日期/时间以普通的文本框输入：`<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `DATETIME_INPUT_FORMATS` 中找到的第一个格式。

TimeInput

class `TimeInput`

时间以普通的文本框输入：`<input type='text' ...>`

接收的参数与 `TextInput` 相同，但是带有一些可选的参数：

`format`

字段的初始值应该显示的格式。

如果没有提供 `format` 参数，默认的格式为参考本地化格式在 `TIME_INPUT_FORMATS` 中找到的第一个格式。

Textarea

class `Textarea`

文本区域：`<textarea>...</textarea>`

选择和复选框Widget

CheckboxInput

class `CheckboxInput`

复选框：`<input type='checkbox' ...>`

接受一个可选的参数：

`check_test`

一个可调用的对象，接收 `CheckboxInput` 的值并如果复选框应该勾上返回 `True`。

Select

class `Select`

Select widget：`<select><option ...>...</select>`

`choices`

当表单字段没有 `choices` 属性时，该属性是随意的。如果字段有choice 属性，当 `字段` 的该属性更新时，它将覆盖你在这里的任何设置。

NullBooleanSelect

class `NullBooleanSelect`

Select Widget，选项为‘Unknown’、‘Yes’ 和‘No’。

SelectMultiple

class `SelectMultiple`

类似 `Select`，但是允许多个选择：`<select multiple='multiple'>...</select>`

RadioSelect

class `RadioSelect`

类似 `Select`，但是渲染成 `<li>` 标签中的一个单选按钮列表：

```
<ul>
  <li><input type='radio' name='...'></li>
  ...
</ul>
```

你可以迭代模板中的单选按钮来更细致地控制生成的HTML。假设表单 `myform` 具有一个字段 `beatles`，它使用 `RadioSelect` 作为Widget：

```
{% for radio in myform.beatles %}
<div class="myradio">
    {{ radio }}
</div>
{% endfor %}
```

它将生成以下HTML：

```
<div class="myradio">
    <label for="id_beatles_0"><input id="id_beatles_0" name="beatles" type="radio" value=
</div>
<div class="myradio">
    <label for="id_beatles_1"><input id="id_beatles_1" name="beatles" type="radio" value=
</div>
<div class="myradio">
    <label for="id_beatles_2"><input id="id_beatles_2" name="beatles" type="radio" value=
</div>
<div class="myradio">
    <label for="id_beatles_3"><input id="id_beatles_3" name="beatles" type="radio" value=
</div>
```

这包括 `<label>` 标签。你可以使用单选按钮的 `tag` 、 `choice_label` 和 `id_for_label` 属性进行更细的控制。例如，这个模板：

```
{% for radio in myform.beatles %}
    <label for="{{ radio.id_for_label }}">
        {{ radio.choice_label }}
        <span class="radio">{{ radio.tag }}</span>
    </label>
{% endfor %}
```

... 将生成下面的HTML：

```
<label for="id_beatles_0">
    John
    <span class="radio"><input id="id_beatles_0" name="beatles" type="radio" value="john"
</label>

<label for="id_beatles_1">
    Paul
    <span class="radio"><input id="id_beatles_1" name="beatles" type="radio" value="paul"
</label>

<label for="id_beatles_2">
    George
    <span class="radio"><input id="id_beatles_2" name="beatles" type="radio" value="georg
</label>

<label for="id_beatles_3">
    Ringo
    <span class="radio"><input id="id_beatles_3" name="beatles" type="radio" value="ringo
</label>
```

如果你不迭代单选按钮 —— 例如，你的模板只是简单地包含 `{{ myform.beatles }}` —— 它们将以 `<ul>` 中的 `<li>` 标签输出，就像上面一样。

外层的 `<ul>` 将带有定义在Widget 上的 `id` 属性。

Changed in Django 1.7:

当迭代单选按钮时， `label` 和 `input` 标签分别包含 `for` 和 `id` 属性。每个单项按钮具有一个 `id_for_label` 属性来输出元素的ID。

CheckboxSelectMultiple

class `CheckboxSelectMultiple`

类似 `SelectMultiple`，但是渲染成一个复选框列表：

```
<ul>
  <li><input type='checkbox' name='...' ></li>
  ...
</ul>
```

外层的 `<ul>` 具有定义在Widget 上的 `id` 属性。

类似 `RadioSelect`，你可以迭代列表的每个复选框。更多细节参见 `RadioSelect` 的文档。

Changed in Django 1.7:

当迭代单选按钮时， `label` 和 `input` 标签分别包含 `for` 和 `id` 属性。 每个单项按钮具有一个 `id_for_label` 属性来输出元素的ID。

文件上传Widget

FileInput

class `FileInput`

文件上传输入： `<input type='file' ...>`

ClearableFileInput

class `ClearableFileInput`

文件上传输入： `<input type='file' ...>`， 带有一个额外的复选框，如果该字段不是必选的且有初始的数据，可以清除字段的值。

复合Widget

MultipleHiddenInput

class `MultipleHiddenInput`

多个 `<input type='hidden' ...>` Widget。

一个处理多个隐藏的Widget 的Widget，用于值为一个列表的字段。

`choices`

当表单字段没有 `choices` 属性时，这个属性是可选的。如果字段有choice 属性，当 `字段` 的该属性更新时，它将覆盖你在这里的任何设置。

SplitDateTimeWidget

class `SplitDateTimeWidget`

封装（使用 `MultiWidget` ）两个Widget： `DateInput` 用于日期， `TimeInput` 用于时间。

`SplitDateTimeWidget` 有两个可选的属性：

`date_format`

类似 `DateInput.format`

`time_format`

类似 `TimeInput.format`

SplitHiddenDateTimeWidget

class `SplitHiddenDateTimeWidget`

类似 `SplitDateTimeWidget` ，但是日期和时间都使用 `HiddenInput` 。

SelectDateWidget

class `SelectDateWidget` [source]

封装三个 `Select` Widget：分别用于年、月、日。注意，这个Widget 与标准的Widget 位于不同的文件中。

接收一个可选的参数：

`years`

一个可选的列表/元组，用于”年“选择框。默认为包含当前年份和未来9年的一个列表。

`months`

New in Django 1.7.

一个可选的字典，用于”月“选择框。

字典的键对应于月份的数字（从1开始），值为显示出来的月份：

```
MONTHS = {
    1:_('jan'), 2:_('feb'), 3:_('mar'), 4:_('apr'),
    5:_('may'), 6:_('jun'), 7:_('jul'), 8:_('aug'),
    9:_('sep'), 10:_('oct'), 11:_('nov'), 12:_('dec')
}
```

`empty_label`

New in Django 1.8.

如果 `DateField` 不是必选的， `SelectDateWidget` 将有一个空的选项位于选项的顶部（默认为 `---`）。你可以通过 `empty_label` 属性修改这个文本。 `empty_label` 可以是一个 `字符串` 、 `列表` 或 `元组` 。当使用字符串时，所有的选择框都带有这个空选项。如果 `empty_label` 为具有3个字符串元素的 `列表` 或 `元组` ，每个选择框将具有它们自定义的空选项。空选项应该按这个顺序 `('year_label', 'month_label', 'day_label')` 。

```
# A custom empty label with string
field1 = forms.DateField(widget=SelectDateWidget(empty_label="Nothing"))

# A custom empty label with tuple
field1 = forms.DateField(widget=SelectDateWidget(
    empty_label=("Choose Year", "Choose Month", "Choose Day"))
```

> 译者：Django 文档协作翻译小组，原文：Built-in widgets。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

高级

表单素材(`Media` 类)

渲染有吸引力的、易于使用的web表单不仅仅需要HTML -- 同时也需要CSS样式表，并且，如果你打算使用奇妙的web2.0组件，你也需要在每个页面包含一些JavaScript。任何提供的页面都需要CSS和JavaScript的精确配合，它依赖于页面上所使用的组件。

这就是素材定义所导入的位置。Django允许你将一些不同的文件 -- 像样式表和脚本 -- 与需要这些素材的表单和组件相关联。例如，如果你想要使用日历来渲染DateField，你可以定义一个自定义的日历组件。这个组件可以与渲染日历所需的CSS和JavaScript关联。当日历组件用在表单上的时候，Django可以识别出所需的CSS和JavaScript文件，并且提供一个文件名的列表，以便在你的web页面上简单地包含这些文件。

素材和Django Admin

Django的Admin应用为日历、过滤选择等一些东西定义了一些自定义的组件。这些组件定义了素材的需求，DJango Admin使用这些自定义组件来代替Django默认的组件。Admin模板只包含在提供页面上渲染组件所需的那些文件。

如果你喜欢Django Admin应用所使用的那些组件，可以在你的应用中随意使用它们。它们位于 `django.contrib.admin.widgets` 。

选择哪个JavaScript工具包？

现在有许多JavaScript工具包，它们中许多都包含组件（比如日历组件），可以用于提升你的应用。Django 有意避免去称赞任何一个JavaScript工具包。每个工具包都有自己的有点和缺点 -- 要使用适合你需求的任何一个。Django 有能力集成任何JavaScript工具包。

作为静态定义的素材

定义素材的最简单方式是作为静态定义。如果使用这种方式，定义在 `Media` 内部类中出现，内部类的属性定义了需求。

这是一个简单的例子：

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

上面的代码定义了 `CalendarWidget`，它继承于 `TextInput`。每次CalendarWidget在表单上使用时，表单都会包含CSS文件 `pretty.css`，以及JavaScript文件 `animations.js` 和 `actions.js`。

静态定义在运行时被转换为名为 `media` 的组件属性。`CalendarWidget` 实例的素材列表可以通过这种方式获取：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
```

下面是所有可能的 `Media` 选项的列表。它们之中没有必需选项。

css

各种表单和输出媒体所需的，描述CSS的字典。

字典中的值应该为文件名称的列表或者元组。对于如何指定这些文件的路径，详见路径的章节。

字典中的键位输出媒体的类型。它们和媒体声明中CSS文件接受的类型相同： ‘all’, ‘aural’, ‘braille’, ‘embossed’, ‘handheld’, ‘print’, ‘projection’, ‘screen’, ‘tty’ 和‘tv’。如果你需要为不同的媒体类型使用不同的样式表，要为每个输出媒体提供一个CSS文件的列表。下面的例子提供了两个CSS选项 -- 一个用于屏幕，另一个用于打印：

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

如果一组CSS文件适用于多种输出媒体的类型，字典的键可以为输出媒体类型的逗号分隔的列表。在下面的例子中，TV和投影仪具有相同的媒体需求：

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

如果最后的CSS定义即将被渲染，会变成下面的HTML：

```
<link href="http://static.example.com/pretty.css" type="text/css" media="screen" rel="sty
<link href="http://static.example.com/lo_res.css" type="text/css" media="tv,projector" re
<link href="http://static.example.com/newspaper.css" type="text/css" media="print" rel="s
```

js

所需的JavaScript文件由一个元组来描述。如何制定这些文件的路径，详见路径一节。

extend

一直布尔值，定义了 `Media` 声明的继承行为。

通常，任何使用静态 `Media` 定义的对象都会继承所有和父组件相关的素材。无论父对象如何定义它自己的需求，都是这样。例如，如果我们打算从上面的例子中扩展我们的基础日历控件：

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<link href="http://static.example.com/fancy.css" type="text/css" media="all" rel="stylesh
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

FancyCalendar 组件继承了所有父组件的素材。如果你不想让 `Media` 以这种方式被继承，要向 `Media` 声明中添加 `extend=False` 声明：

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/fancy.css" type="text/css" media="all" rel="stylesh
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

如果你需要对继承进行更多控制，要使用动态属性来定义你的素材。动态属性可以提供更多的控制，来控制继承哪个文件。

Media as a dynamic property

如果你需要对素材需求进行更多的复杂操作，你可以直接定义 `media` 属性。这可以通过定义一个返回 `forms.Media` 实例的组件属性来实现。 `forms.Media` 的构造器接受 `css` 和 `js` 关键字参数，和在静态媒体定义中的格式相同。

例如，我们的日历组件的静态定义可以定义成动态形式：

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

对于如何构建动态 `media` 属性的的返回值，详见媒体对象一节。

素材定义中的路径

用于指定素材的路径可以是相对的或者绝对的。如果路径以 `/` ， `http://` 或者 `https://` 开头，会被解释为绝对路径。所有其它的路径会在开头追加合适前缀的值。

作为 *staticfiles app* 的简介的一部分，添加了两个新的设置，它们涉及到渲染完整页面所需的“静态文件”： `STATIC_URL` 和 `STATIC_ROOT` 。

Django 会检查是否 `STATIC_URL` 设置不是 `None` ，来寻找合适的前缀来使用，并且会自动回退使用 `MEDIA_URL` 。例如，如果你站点的 `MEDIA_URL` 是 `'http://uploads.example.com/'` 并且 `STATIC_URL` 是 `None` ：

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('/css/pretty.css',),
...         }
...         js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://uploads.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

但如果 `STATIC_URL` 为 `'http://static.example.com/'` ：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

`Media` 对象

当你访问表单上的一个组件的 `media` 属性时，返回值是一个 `forms.Media` 对象。就像已经看到的那样，表示 `Media` 对象的字符串，是在你的HTML页面的 `<head>` 代码段包含相关文件所需的HTML。

然而， `Media` 对象具有一些其它的有趣属性。

素材的子集

如果你仅仅想得到特定类型的文件，你可以使用下标运算符来过滤出你感兴趣的媒体。例如：

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>

>>> print(w.media['css'])
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
```

当你使用下标运算符的时候，返回值是一个新的 `Media` 对象，但是只含有感兴趣的媒体。

组合 `Media` 对象

`Media` 对象可以添加到一起。添加两个 `Media` 的时候，产生的 `Media` 对象含有二者指定的素材的并集：

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('pretty.css',)
...         }
...         js = ('animations.js', 'actions.js')

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ('whizbang.js',)

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

表单上的 `Media`

组件并不是唯一拥有 `media` 定义的对象 -- 表单可以定义 `media` 。在表单上定义 `media` 的规则和组件上面一样：定义可以为静态的或者动态的。声明的路径和继承规则也严格一致。

无论是否你定义了 `media`， 所有表单对象都有 `media` 属性。这个属性的默认值是，向所有属于这个表单的组件添加 `media` 定义的结果。

```
>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

如果你打算向表单关联一些额外的素材 -- 例如，表单布局的CSS -- 只是向表单添加 `Media` 声明就可以了：

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...
...     class Media:
...         css = {
...             'all': ('layout.css',)
...         }

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="styles
<link href="http://static.example.com/layout.css" type="text/css" media="all" rel="styles
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

> 译者：Django 文档协作翻译小组，原文：Integrating media。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

开发过程

学习各种组件和工具帮助你对Django应用进行改进和测试：

设置

Django 的设置

Django 的设置文件包含你安装的Django 的所有配置。这页文档解释设置是如何工作以及有哪些设置。

基础

设置文件只是一个Python 模块，带有模块级别的变量。

下面是一些示例设置：

```
ALLOWED_HOSTS = ['www.example.com']
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

> 注
>
> 如果你设置 `DEBUG` 为 `False`，那么你应该正确设置 `ALLOWED_HOSTS` 的值。

因为设置文件是一个Python 模块，所以适用以下情况：

- 不允许出现Python 语法错误。
- 它可以使用普通的Python 语法动态地设置。例如：

```
MY_SETTING = [str(i) for i in range(30)]
```

- 它可以从其它设置文件导入值。

指定设置文件

`DJANGO_SETTINGS_MODULE`

当你使用Django 时，你必须告诉它你正在使用哪个设置。这可以使用环境变量 `DJANGO_SETTINGS_MODULE` 来实现。

`DJANGO_SETTINGS_MODULE` 的值应该使用Python 路径的语法，例如 `mysite.settings`。注意，设置模块应该在Python 的导入查找路径 中。

django-admin 工具

当使用 `django-admin` 时， 你可以设置只设置环境变量一次，或者每次运行该工具时显式传递设置模块。

例如（Unix Bash shell）：

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

例如（Windows shell）：

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

使用--settings 命令行参数可以手工指定设置：

```
django-admin runserver --settings=mysite.settings
```

在服务器上(mod_wsgi)

在线上服务器环境中，你需要告诉WSGI 的application 使用哪个设置文件。可以使用 os.environ 实现：

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

阅读Django mod_wsgi 文档 以获得关于Django WSGI application 的更多和其它常见信息。

默认的设置

Django 的设置文件不需要定义所有的设置。每个设置都有一个合理的默认值。这些默认值位于 `django/conf/global_settings.py` 模块中。

下面是Django 用来编译设置的算法：

- 从 `global_settings.py` 中加载设置。
- 从指定的设置文件中加载设置，如有必要则覆盖全局的设置。

注意，设置文件不 应该从global_settings 中导入，因为这是多余的。

查看改变的设置

有一个简单的方法可以查看哪些设置与默认的设置不一样了。python `manage.py diffsettings` 命令显示当前的设置文件和Django 默认设置之间的差异。

获取更多信息，查看 `diffsettings` 的文档。

在Python 代码中使用设置

在Django 应用中，可以通过导入 `django.conf.settings` 对象来使用设置。例如：

```
from django.conf import settings

if settings.DEBUG:
    # Do something
```

注意， `django.conf.settings` 不是一个模块 —— 它是一个对象。所以不可以导入每个单独的设置：

```
from django.conf.settings import DEBUG  # This won't work.
```

还要注意，你的代码不应该 从 `global_settings` 或你自己的设置文件中导入。 `django.conf.settings` 抽象出默认设置和站点特定设置的概念；它表示一个单一的接口。它还可以将代码从你的设置所在的位置解耦出来。

运行时改变设置

请不要在应用运行时改变设置。例如，不要在视图中这样做：

```
from django.conf import settings

settings.DEBUG = True   # Don't do this!
```

给设置赋值的唯一地方是在设置文件中。

安全

因为设置文件包含敏感的信息，例如数据库密码，你应该尽一切可能来限制对它的访问。例如，修改它的文件权限使得只有你和Web 服务器使用者可以读取它。这在共享主机的环境中特别重要。

可用的设置

完整的可用设置清单，请参见设置参考。

创建你自己的设置

没有什么可以阻止你为自己的Django 应用创建自己的设置。只需要遵循下面的一些惯例：

- 设置名称全部是大写
- 不要使用一个已经存在的设置

对于序列类型的设置，Django 自己使用元组而不是列表，但这只是一个习惯。

不用DJANGO_SETTINGS_MODULE 设置

有些情况下，你可能想绕开 `DJANGO_SETTINGS_MODULE` 环境变量。例如，如果你正在使用自己的模板系统，而你不想建立指向设置模块的环境变量。

这些情况下，你可以手工配置Django 的设置。实现这点可以通过调用：

`django.conf.settings.configure(default_settings, **settings)`

例如：

```
from django.conf import settings

settings.configure(DEBUG=True)
```

可以传递 `configure()` 给任意多的关键字参数，每个关键字参数表示一个设置及其值。每个参数的名称应该都是大写，与上面讲到的设置名称相同。如果某个设置没有传递给 `configure()` 而且在后面需要使用到它，Django 将使用其默认设置的值。

当你在一个更大的应用中使用到Django 框架的一部分，有必要以这种方式配置Django ——而且实际上推荐这么做。

所以，当通过 `settings.configure()` 配置时，Django 不会对进程的环境变量做任何修改（参见 `TIME_ZONE` 文档以了解为什么会发生）。在这些情况下，它假设你已经完全控制你的环境变量。

自定义默认的设置

如果你想让默认值来自其它地方而不是 `django.conf.global_settings`，你可以传递一个提供默认设置的模块或类作为 `default_settings` 参数（或第一个位置参数）给 `configure()` 调用。

在下面的示例中，默认的设置来自 `myapp_defaults`， 并且设置 `DEBUG` 为 `True`，而不论它在 `myapp_defaults` 中的值是什么：

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

下面的示例和上面一样，只是使用 `myapp_defaults` 作为一个位置参数：

```
settings.configure(myapp_defaults, DEBUG=True)
```

正常情况下，你不需要用这种方式覆盖默认值。Django 的默认值以及足够好使，你可以安全地使用它们。注意，如果你传递一个新的默认模块，你将完全取代 Django 的默认值，所以你必须指定每个可能用到的设置的值。完整的设置清单，参见 `django.conf.settings.global_settings` 。

configure() 和DJANGO_SETTINGS_MODULE 两者必居其一

如果你没有设置 `DJANGO_SETTINGS_MODULE` 环境变量，你 必须 在使用到读取设置的任何代码之前调用 `configure()` 。

如果你没有设置 `DJANGO_SETTINGS_MODULE` 且没有调用 `configure()` ，在首次访问设置时 Django 将引发一个 `ImportError` 异常。

如果你设置了 `DJANGO_SETTINGS_MODULE` ，并访问了一下设置，然后 调用 `configure()` ，Django 将引发一个 `RuntimeError` 表示该设置已经有配置。有个属性正好可以用于这个情况：

例如：

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

另外，多次调用 `configure()` 或者在设置已经访问过之后调用 `configure()` 都是错误的。

归结为一点：只使用 `configure()` 或 `DJANGO_SETTINGS_MODULE` 中的一个。不可以两个都用和都不用。

> 另见
>
> 设置参考 包含完整的核心设置和contrib 应用设置的列表。

应用程序

异常

Django异常

DJango会抛出一些它自己的异常，以及Python的标准异常。

Django核心异常

Django核心异常类定义在 `django.core.exceptions` 中。

ObjectDoesNotExist

exception `ObjectDoesNotExist` [source]

`DoesNotExist` 异常的基类；对 `ObjectDoesNotExist` 的 `try/except` 会为所有模型捕获到所有 `DoesNotExist` 异常。

`ObjectDoesNotExist` 和 `DoesNotExist` 的更多信息请见 `get()` 。

FieldDoesNotExist

exception `FieldDoesNotExist` [source]

当被请求的字段在模型或模型的父类中不存在时， `FieldDoesNotExist` 异常由模型的 `_meta.get_field()` 方法抛出。

Changed in Django 1.8:

之前的版本中，异常只在 `django.db.models.fields` 中定义，并不是公共API的一部分。

MultipleObjectsReturned

exception `MultipleObjectsReturned` [source]

`MultipleObjectsReturned` 异常由查询产生，当预期只有一个对象，但是有多个对象返回的时候。这个异常的一个基础版本在 `django.core.exceptions` 中提供。每个模型类都包含一个它的子类版本，它可以用于定义返回多个对象的特定的对象类型。

详见 `get()` 。

SuspiciousOperation

exception `SuspiciousOperation` [source]

当用户进行的操作在安全方面可疑的时候，抛出 `SuspiciousOperation` 异常，例如篡改会话 cookie。 `SuspiciousOperation` 的子类包括：

- `DisallowedHost`
- `DisallowedModelAdminLookup`
- `DisallowedModelAdminToField`
- `DisallowedRedirect`
- `InvalidSessionKey`
- `SuspiciousFileOperation`
- `SuspiciousMultipartForm`
- `SuspiciousSession`

如果 `SuspiciousOperation` 异常到达了WSGI处理器层，它会在 `Error` 层记录，并导致 `HttpResponseBadRequest` 异常。 详见日志文档。

PermissionDenied

exception `PermissionDenied` [source]

`PermissionDenied` 异常当用户不被允许来执行请求的操作时产生。

ViewDoesNotExist

exception `ViewDoesNotExist` [source]

当所请求的视图不存在时， `ViewDoesNotExist` 异常由 `django.core.urlresolvers` 产生。

MiddlewareNotUsed

exception `MiddlewareNotUsed` [source]

当中间件没有在服务器配置中出现时， 产生 `MiddlewareNotUsed` 异常。

ImproperlyConfigured

exception `ImproperlyConfigured` [source]

DJango配置不当时产生 `ImproperlyConfigured` 异常 -- 例如， `settings.py` 中的值不正确或者不可解析。

FieldError

exception `FieldError` [source]

`FieldError` 异常当模型字段上出现问题时产生。它会由以下原因造成：

- 模型中的字段与抽象基类中相同名称的字段冲突。
- 排序造成了一个死循环。
- 关键词不能由过滤器参数解析。
- 字段不能由查询参数中的关键词决定。
- 连接（join）不能在指定对象上使用。
- 字段名称不可用。
- 查询包含了无效的 order_by参数。

ValidationError

exception `ValidationError` [source]

当表单或模型字段验证失败时抛出 `ValidationError` 异常。关于验证的更多信息，请见表单字段验证, 模型字段验证 和 验证器参考。

NON_FIELD_ERRORS

`NON_FIELD_ERRORS`

在表单或者模型中不属于特定字段的 `ValidationError` 被归类为 `NON_FIELD_ERRORS`。This constant is used as a key in dictionaries that otherwise map fields to their respective list of errors.

URL解析器异常

URL解析器异常定义在 `django.core.urlresolvers` 中。

Resolver404

exception `Resolver404` [source]

当向 `resolve()` 传递的路径不映射到视图的时候， `Resolver404` 异常由 `django.core.urlresolvers.resolve()` 产生。 它是 `django.http.Http404` 的子类。

NoReverseMatch

exception `NoReverseMatch` [source]

当你的URLconf中的一个匹配的URL不能基于提供的参数识别时， `NoReverseMatch` 异常由 `django.core.urlresolvers` 产生。

Database Exceptions

数据库异常由 `django.db` 导入。

Django封装了标准的数据库异常，以便确保你的DJango代码拥有这些类的通用实现。

exception `Error`

exception `InterfaceError`

exception `DatabaseError`

exception `DataError`

exception `OperationalError`

exception `IntegrityError`

exception `InternalError`

exception `ProgrammingError`

exception `NotSupportedError`

Django数据库异常的包装器的行为和底层的数据库异常一样。详见**PEP 249**，Python 数据库 API 说明 v2.0。

按照 **PEP 3134**， `__cause__` 属性会在原生（底层）的数据库异常中设置，允许访问所提供的任何附加信息。（注意这一属性在Python 2和 3下面都可用，虽然 **PEP 3134**通常只用于 Python 3。）

exception `models.``ProtectedError`

使用 `django.db.models.PROTECT` 时，抛出异常来阻止所引用对象的删除。 `models.` ProtectedError is a subclass of `IntegrityError` .

Http异常

HTTP异常由 `django.http` 导入。

UnreadablePostError

exception `UnreadablePostError`

用户取消上传时抛出 `UnreadablePostError` 异常。

事务异常

事务异常定义在 `django.db.transaction` 中。

TransactionManagementError

exception `TransactionManagementError` [source]

对于数据库事务相关的任何问题，抛出 `TransactionManagementError` 异常。

测试框架异常

由DJango `django.test` 包提供的异常。

RedirectCycleError

exception `client.``RedirectCycleError`

New in Django 1.8.

当测试客户端检测到重定向的循环或者过长的链时，抛出 `RedirectCycleError` 异常。

Python异常

Django在适当的时候也会抛出Python的内建异常。进一步的信息请见内建的异常的Python文档。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

django-admin 和 manage.py

编写自定义的django-admin命令

应用可以通过 `manage.py` 注册它们自己的动作。例如，你可能想为你正在发布的Django应用添加一个 `manage.py` 动作。在本页文档中，我们将为教程中的 `polls` 应用构建一个自定义的 `closepoll` 命令。

要做到这点，只需向该应用添加一个 `management/commands` 目录。Django将为该目录中名字没有以下划线开始的每个Python模块注册一个 `manage.py` 命令。例如：

```
polls/
    __init__.py
    models.py
    management/
        __init__.py
        commands/
            __init__.py
            _private.py
            closepoll.py
    tests.py
    views.py
```

在Python 2上，请确保 `management` 和 `management/commands` 两个目录都包含 `__init__.py` 文件，否则将检测不到你的命令。

在这个例子中， `closepoll` 命令对任何项目都可使用，只要它们在 `INSTALLED_APPS` 里包含 `polls` 应用。

`_private.py` 将不可以作为一个管理命令使用。

`closepoll.py` 模块只有一个要求 – 它必须定义一个 `Command` 类并扩展自 `BaseCommand` 或其子类。

> 独立的脚本
>
> 自定义的管理命令主要用于运行独立的脚本或者UNIX crontab和Windows周期任务控制面板周期性执行的脚本。

要实现这个命令，需将 `polls/management/commands/closepoll.py` 编辑成这样：

```
from django.core.management.base import BaseCommand, CommandError
from polls.models import Poll

class Command(BaseCommand):
    help = 'Closes the specified poll for voting'

    def add_arguments(self, parser):
        parser.add_argument('poll_id', nargs='+', type=int)

    def handle(self, *args, **options):
        for poll_id in options['poll_id']:
            try:
                poll = Poll.objects.get(pk=poll_id)
            except Poll.DoesNotExist:
                raise CommandError('Poll "%s" does not exist' % poll_id)

            poll.opened = False
            poll.save()

            self.stdout.write('Successfully closed poll "%s"' % poll_id)
```

```
Changed in Django 1.8:

在Django 1.8之前，管理命令基于optparse模块，位置参数传递给*args，可选参数传递给**options。现在，管理
```

> 注
>
> 当你使用管理命令并希望提供控制台输出时，你应该写到 `self.stdout` 和 `self.stderr`，而不能直接打印到 `stdout` 和 `stderr`。通过使用这些代理方法，测试你自定义的命令将变得非常容易。还请注意，你不需要在消息的末尾加上一个换行符，它将被自动添加，除非你指定 `ending` 参数：
>
> ```
> self.stdout.write("Unterminated line", ending='')
> ```

新的自定义命令可以使用 `python manage.py closepoll <poll_id>` 调用。

`handle()` 接收一个或多个 `poll_ids` 并为他们中的每个设置 `poll.opened` 为 `False`。如果用户访问任何不存在的 `polls`，将引发一个 `CommandError`。`poll.opened` 属性在教程中并不存在，只是为了这个例子将它添加到 `polls.models.Poll` 中。

接收可选参数

通过接收额外的命令行选项，可以简单地修改 `closepoll` 来删除一个给定的 `poll` 而不是关闭它。这些自定义的选项可以像下面这样添加到 `add_arguments()` 方法中：

```
class Command(BaseCommand):
    def add_arguments(self, parser):
        # Positional arguments
        parser.add_argument('poll_id', nargs='+', type=int)

        # Named (optional) arguments
        parser.add_argument('--delete',
            action='store_true',
            dest='delete',
            default=False,
            help='Delete poll instead of closing it')

    def handle(self, *args, **options):
        # ...
        if options['delete']:
            poll.delete()
        # ...
```

```
Changed in Django 1.8:

之前，只支持标准的optparse库，你必须利用optparse.make_option()扩展命令option_list变量。
```

选项（在我们的例子中为 `delete`）在 `handle` 方法的 `options` 字典参数中可以访问到。更多关于 `add_argument` 用法的信息，请参考 `argparse` 的Python 文档。

除了可以添加自定义的命令行选项， 管理命令还可以接收一些默认的选项，例如 `--verbosity` 和 `--traceback`。

管理命令和区域设置

默认情况下， `BaseCommand.execute()` 方法使转换失效，因为某些与Django一起的命令完成的任务要求一个与项目无关的语言字符串（例如，面向用户的内容渲染和数据库填入）。

```
Changed in Django 1.8:

在之前的版本中，Django强制使用"en-us"区域设置而不是使转换失效。
```

如果，出于某些原因，你的自定义的管理命令需要使用一个固定的区域设置，你需要在你的 `handle()` 方法中利用I18N支持代码提供的函数手工地启用和停用它：

```
from django.core.management.base import BaseCommand, CommandError
from django.utils import translation

class Command(BaseCommand):
    ...
    can_import_settings = True

    def handle(self, *args, **options):

        # Activate a fixed locale, e.g. Russian
        translation.activate('ru')

        # Or you can activate the LANGUAGE_CODE # chosen in the settings:
        from django.conf import settings
        translation.activate(settings.LANGUAGE_CODE)

        # Your command logic here
        ...

        translation.deactivate()
```

另一个需要可能是你的命令只是简单地应该使用设置中设置的区域设置且Django应该保持不让它停用。你可以使用 `BaseCommand.leave_locale_alone` 选项实现这个功能。

虽然上面描述的场景可以工作，但是考虑到系统管理命令对于运行非统一的区域设置通常必须非常小心，所以你可能需要：

- 确保运行命令时 `USE_I18N` 设置永远为 `True`（this is a good example of the potential problems stemming from a dynamic runtime environment that Django commands avoid offhand by deactivating translations）。
- Review the code of your command and the code it calls for behavioral differences when locales are changed and evaluate its impact on predictable behavior of your command.

测试

关于如何测试自定义管理命令的信息可以在测试文档中找到。

Command 对象

`class BaseCommand`

所有管理命令最终继承的基类。

如果你想获得解析命令行参数并在响应中如何调用代码的所有机制，可以使用这个类；如果你不需要改变这个行为，请考虑使用它的子类。

继承 `BaseCommand` 类要求你实现 `handle()` 方法。

属性

所有的属性都可以在你派生的类中设置，并在 `BaseCommand` 的子类中使用。

`BaseCommand.args`

一个字符串，列出命令接收的参数，适合用于帮助信息；例如，接收一个应用名称列表的命令可以设置它为‘ `<app_label app_label ...>` ’。

```
Deprecated since version 1.8:
现在，应该在add_arguments()方法中完成，通过调用parser.add_argument()方法。参见上面的closepoll例子。
```

`BaseCommand.can_import_settings`

一个布尔值，指示该命令是否需要导入Django的设置的能力；如果为 `True` ， `execute()` 将在继续之前验证这是否可能。默认值为 `True` 。

`BaseCommand.help`

命令的简短描述，当用户运行 `python manage.py help <command>` 命令时将在帮助信息中打印出来。

`BaseCommand.missing_args_message`

```
New in Django 1.8.
```

如果你的命令定义了必需的位置参数，你可以自定义参数缺失时返回的错误信息。默认是由 `argparse` 输出的 (“too few arguments”)。

`BaseCommand.option_list`

这是optparse选项列表，将赋值给命令的OptionParser用于解析命令。

```
Deprecated since version 1.8:
现在，你应该覆盖`add_arguments()`方法来添加命令行接收的自定义参数。参见上面的例子。
```

`BaseCommand.output_transaction`

一个布尔值，指示命令是否输出SQL语句；如果为 `True` ，输出将被自动用 `BEGIN;` 和 `COMMIT;` 封装。默认为 `False` 。

`BaseCommand.requires_system_checks`

```
New in Django 1.7.
```

一个布尔值；如果为 `True` ，在执行该命令之前将检查整个Django项目是否有潜在的问题。如果 `requires_system_checks` 缺失，则使用 `requires_model_validation` 的值。如果后者的值也缺失，则使用默认值（ `True` ）。同时定

义 `requires_system_checks` 和 `requires_model_validation` 将导致错误。

`BaseCommand.requires_model_validation`

> Deprecated since version 1.7:
>
> 被requires_system_checks代替

一个布尔值；如果为 `True`，将在执行命令之前作安装的模型的验证。默认为 `True`。若要验证一个单独应用的模型而不是全部应用的模型，可以调用在 `handle()` 中调用 `validate()`。

`BaseCommand.leave_locale_alone`

一个布尔值，指示设置中的区域设置在执行命令过程中是否应该保持而不是强制设成'en-us'。

默认值为 `False`。

如果你决定在你自定义的命令中修改该选项的值，请确保你知道你正在做什么。 如果它创建对区域设置敏感的数据库内容，这种内容不应该包含任何转换（比如 `django.contrib.auth` 权限发生的情况），因为将区域设置变成与实际上默认的'en-us' 不同可能导致意外的效果。更进一步的细节参见上面的管理命令和区域设置一节。

当 `can_import_settings` 选项设置为 `False` 时，该选项不可以也为 `False`，因为尝试设置区域设置需要访问 `settings`。这种情况将产生一个 `CommandError`。

方法

`BaseCommand` 有几个方法可以被覆盖，但是只有 `handle()` 是必须实现的。

> 在子类中实现构造函数
>
> 如果你在 `BaseCommand的` 子类中实现 `__init__`，你必须调用 `BaseCommand` 的 `__init__`：
>
> ```
> class Command(BaseCommand):
>     def __init__(self, *args, **kwargs):
>         super(Command, self).__init__(*args, **kwargs)
>         # ...
> ```

`BaseCommand.add_arguments(parser)`

> New in Django 1.8.

添加解析器参数的入口，以处理传递给命令的命令行参数。自定义的命令应该覆盖这个方法以添加命令行接收的位置参数和可选参数。当直接继承 `BaseCommand` 时不需要调用 `super()`。

`BaseCommand.get_version()`

返回Django的版本，对于所有内建的Django命令应该都是正确的。用户提供的命令可以覆盖这个方法以返回它们自己的版本。

`BaseCommand.execute(*args, **options)`

执行这个命令，如果需要则作系统检查（通过 `requires_system_checks` 属性控制）。如果该命令引发一个 `CommandError`，它将被截断并打印到标准错误输出。

> 在你的代码中调用管理命令
>
> 不应该在你的代码中直接调用 `execute()` 来执行一个命令。请使用 `call_command`。

`BaseCommand.handle(*args, **options)`

命令的真正逻辑。子类必须实现这个方法。

`BaseCommand.check(app_configs=None, tags=None, display_num_errors=False)`

```
New in Django 1.7.
```

利用系统的检测框架检测全部Django项目的潜在问题。严重的问题将引发 `CommandError`；警告会输出到标准错误输出；次要的通知会输出到标准输出。

如果 `app_configs` 和 `tags` 都为 `None`，将进行所有的系统检查。 `tags` 可以是一个要检查的标签列表，比如 `compatibility` 或 `models`。

`BaseCommand.validate(app=None, display_num_errors=False)`

```
Deprecated since version 1.7:
被check命令代替
```

如果 `app` 为 `None`，那么将检查安装的所有应用的错误。

BaseCommand 的子类

`class AppCommand`

这个管理命令接收一个或多个安装的应用标签作为参数，并对它们每一个都做一些动作。

子类不用实现 `handle()`，但必须实现 `handle_app_config()`，它将会为每个应用调用一次。

`AppCommand.handle_app_config(app_config, **options)`

对 `app_config` 完成命令行的动作，其中 `app_config` 是 `AppConfig` 的实例，对应于在命令行上给出的应用标签。

> Changed in Django 1.7:
>
> 以前，AppCommand子类必须实现 `handle_app(app, **options)`，其中 `app` 是一个模型模块。新的API可以不需要模型模块来处理应用。迁移的最快的方法如下：
>
> ```
> def handle_app_config(app_config, **options):
>     if app_config.models_module is None:
>         return                                  # Or raise an exception.
>     app = app_config.models_module
>     # Copy the implementation of handle_app(app_config, **options) here.
> ```
>
> 然而，你可以通过直接使用 `app_config` 的属性来简化实现。

`class LabelCommand`

这个管理命令接收命令行上的一个或多个参数（标签），并对它们每一个都做一些动作。

子类不用实现 `handle()`，但必须实现 `handle_label()`，它将会为每个标签调用一次。

`LabelCommand.handle_label(label, **options)`

对 `label` 完成命令行的动作， `label` 是命令行给出的字符串。

`class NoArgsCommand`

> Deprecated since version 1.8:
>
> 使用BaseCommand代替，它默认也不需要参数。

这个命令不接收命令行上的参数。

子类不需要实现 `handle()`，但必须实现 `handle_noargs()`； `handle()` 本身已经被覆盖以保证不会有参数传递给命令。

`NoArgsCommand.handle_noargs(**options)`

完成这个命令的动作

Command 的异常

`class CommandError`

异常类，表示执行一个管理命令时出现问题。

如果这个异常是在执行一个来自命令行控制台的管理命令时引发，它将被捕获并转换成一个友好的错误信息到合适的输出流（例如，标准错误输出）；因此，引发这个异常（并带有一个合理的错误描述）是首选的方式来指示在执行一个命令时某些东西出现错误。

如果管理命令从代码中通过call_command调用，那么需要时捕获这个异常由你决定。

> 译者：Django 文档协作翻译小组，原文：Adding custom commands。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

测试

Django中的测试

自动化测试对于现代web开发者来说，是非常实用的除错工具。你可以使用一系列测试-- 测试套件 -- 来解决或者避免大量问题：

- 当你编写新代码的时候，你可以使用测试来验证你的代码是否像预期一样工作。
- 当你重构或者修改旧代码的时候，你可以使用测试来确保你的修改不会在意料之外影响到你的应用的应为。

测试web应用是个复杂的任务，因为web应用由很多的逻辑层组成 -- 从HTTP层面的请求处理，到表单验证和处理，到模板渲染。使用Django的测试执行框架和各种各样的工具，你可以模拟请求，插入测试数据，检查你的应用的输出，以及大体上检查你的代码是否做了它应该做的事情。

最好的一点是，它非常简单。

在Django中编写测试的最佳方法是，使用构建于Python标准库的unittest模块。这在编写和运行测试 文档中会详细介绍。

你也可以使用任何其它 Python 的测试框架；Django为整合它们提供了API和工具。这在高级测试话题的使用不同的测试框架 一节中描述。

- 编写和运行测试
- 测试工具
- 高级测试话题

> 译者：Django 文档协作翻译小组，原文：Introduction。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

部署

部署 Django

虽然Django 满满的便捷性让Web 开发人员活得轻松一些，但是如果不能轻松地部署你的网站，这些工具还是没有什么用处。Django 起初，易于部署就是一个主要的目标。有许多优秀的方法可以轻松地来部署Django：

- 如何使用WSGI 部署
- 部署的检查清单

FastCGI 的支持已经废弃并将在Django 1.9 中删除。

- 如何使用FastCGI、SCGI 和AJP 部署Django

如果你是部署Django 和/或 Python 的新手，我们建议你先试试 mod_wsgi。 在大部分情况下，这将是最简单、最迅速和最稳当的部署选择。

> 另见
>
> Django Book（第二版）的第12 章 更详细地讨论了部署，尤其是可扩展性。但是请注意，这个版本是基于Django 1.1 版本编写，而且在 `mod_python` 废弃并于Django 1.5 中删除之后一直没有更新。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

如何使用WSGI 部署

Django 首要的部署平台是WSGI，它是Python Web 服务器和应用的标准。

Django 的 `startproject` 管理命名为你设置一个简单的默认WSGI 配置，你可以根据你项目的需要做调整并指定任何与WSGI 兼容的应用服务器使用。

Django 包含以下WSGI 服务器的入门文档：

- 如何使用Apache 和mod_wsgi 部署Django
- 从Apache 中利用Django 的用户数据库进行认证
- 如何使用Gunicorn 部署Django (100%)
- 如何使用uWSGI 部署Django (100%)

application 对象

使用WSGI 部署的核心概览是 `application` 可调用对象，应用服务器使用它来与你的代码进行交换。在Python 模块中，它通常一个名为 `application` 的对象提供给服务器使用。

`startproject` 命令创建一个 `<project_name>/wsgi.py` 文件，它就包含这样一个 `application` 可调用对象。

它既可用于Django 的开发服务器，也可以用于线上WSGI 的部署。

WSGI 服务器从它们的配置中获得 `application` 可调用对象的路径。Django 内建的服务器，叫做 `runserver` 和 `runfcgi` 命令，是从 `WSGI_APPLICATION` 设置中读取它。默认情况下，它设置为 `<project_name>.wsgi.application`，指向 `<project_name>/wsgi.py` 中的 `application` 可调用对象。

配置settings 模块

当WSGI 服务器加载你的应用时，Django 需要导入settings 模块 —— 这里是你的全部应用定义的地方。

Django 使用 `DJANGO_SETTINGS_MODULE` 环境变量来定位settings 模块。它包含settings 模块的路径，以点分法表示。对于开发环境和线上环境，你可以使用不同的值；这完全取决于你如何组织你的settings。

如果这个变量没有设置，默认的 `wsgi.py` 设置为 `mysite.settings`，其中 `mysite` 为你的项目的名称。这是 `runserver` 如何找到默认的 `settings` 文件的机制。

> 注
>
> 因为环境变量是进程范围的，当你在同一个进程中运行多个Django 站点时，它将不能工作。使用 `mod_wsgi` 就是这个情况。
>
> 为了避免这个问题，可以使用mod_wsgi 的守护进程模式，让每个站点位于它自己的守护进程中，或者在 `wsgi.py` 中通过强制使用 `os.environ["DJANGO_SETTINGS_MODULE"] = "mysite.settings"` 来覆盖这个值。

运用WSGI 中间件

你可以简单地封装application 对象来运用 WSGI 中间件。 例如，你可以在 `wsgi.py` 的底下添加以下这些行：

```
from helloworld.wsgi import HelloWorldApplication
application = HelloWorldApplication(application)
```

如果你结合使用 Django 的application 与另外一个WSGI application 框架，你还可以替换Django WSGI 的application 为一个自定义的WSGI application。

> 注
>
> 某些第三方的WSGI 中间件在处理完一个请求后不调用响应对象上的 `close` —— most notably Sentry's error reporting middleware up to version 2.0.7。这些情况下，不会发送 `request_finished` 信号。这可能导致数据库和memcache 服务的空闲连接。

> 译者：Django 文档协作翻译小组，原文：WSGI servers。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

部署静态文件

> 另见
>
> `django.contrib.staticfiles` 的用法简介，请参见管理静态文件（CSS、images）。

在线上环境部署静态文件

放置静态文件到线上环境的基本步骤很简单：当静态文件改变时，运行 `collectstatic` 命令，然后安排将收集好的静态文件的目录(`STATIC_ROOT`) 搬到静态文件服务器上。取决于 `STATICFILES_STORAGE` ，这些文件可能需要手工移动到一个新的位置或者 `Storage` 类的 `post_process` 方法可以帮你。

当然，与所有的部署任务一样，魔鬼隐藏在细节中。每个线上环境的建立都会有所不同，所以你需要调整基本的纲要以适应你的需求。下面是一些常见的方法，可能有所帮助。

网站和静态文件位于同一台服务器上

如果你的静态文件和网站位于同一台服务器，流程可能像是这样：

- 将你的代码推送到部署的服务器上。
- 在这台服务器上，运行 `collectstatic` 来收集所有的静态文件到 `STATIC_ROOT` 。
- 配置Web 服务器来托管URL `STATIC_URL` 下的 `STATIC_ROOT` 。 例如，这是如何使用 Apache 和mod_wsgi 来完成它。

你可能想自动化这个过程，特别是如果你有多台Web 服务器。有许多种方法来完成这个自动化，但是许多Django 开发人员喜欢 Fabric。

在一下的小节中，我们将演示一些示例的Fabric 脚本来自动化不同选择的文件部署。Fabric 脚本的语法相当简单，但这里不会讲述；参见Fabric 的文档 以获得其语法的完整解释。

所以，一个部署静态文件来多台Web 服务器上的Fabric 脚本大概会是：

```
from fabric.api import *

# Hosts to deploy onto
env.hosts = ['www1.example.com', 'www2.example.com']

# Where your project code lives on the server
env.project_root = '/home/www/myproject'

def deploy_static():
    with cd(env.project_root):
        run('./manage.py collectstatic -v0 --noinput')
```

静态文件位于一台专门的服务器上

大部分大型的Django 站点都使用一台单独的Web 服务器来存放静态文件 —— 例如一台不运行Django 的服务器。这种服务器通常运行一种不同类型的服务器 —— 更快但是功能很少。一些常见的选择有：

- Nginx
- 裁剪版的Apache

配置这些服务器在这篇文档范围之外；查看每种服务器各自的文档以获得说明。

既然你的静态文件服务器不会允许Django，你将需要修改的部署策略，大概会是这样：

- 当静态文件改变时，在本地运行 `collectstatic` 。
- 将你本地的 `STATIC_ROOT` 推送到静态文件服务器相应的目录中。在这一步，常见的选择 rsync ，因为它只传输静态文件改变的部分。

下面是Fabric 脚本大概的样子：

```
from fabric.api import *
from fabric.contrib import project

# Where the static files get collected locally. Your STATIC_ROOT setting.
env.local_static_root = '/tmp/static'

# Where the static files should go remotely
env.remote_static_root = '/home/www/static.example.com'

@roles('static')
def deploy_static():
    local('./manage.py collectstatic')
    project.rsync_project(
        remote_dir = env.remote_static_root,
        local_dir = env.local_static_root,
        delete = True
    )
```

静态文件位于一个云服务或CDN 上

两位一个常见的策略是放置静态文档到一个云存储提供商比如亚马逊的S3 和/或一个CDN(Content Delivery Network)上。这让你可以忽略保存静态文件的问题，并且通常可以加快网页的加载（特别是使用CDN 的时候）。

当使用这些服务时，除了不是使用rsync 传输你的静态文件到服务器上而是到存储提供商或CDN 上之外，基本的工作流程和上面的差不多。

有许多方式可以实现它，但是如果提供商具有API，那么自定义的文件存储后端 将使得这个过程相当简单。如果你已经写好或者正在使用第三方的自定义存储后端，你可以通过设置 `STATICFILES_STORAGE` 来告诉 `collectstatic` 来使用它。

例如，如果你已经在 `myproject.storage.S3Storage` 中写好一个S3 存储的后端，你可以这样使用它：

```
STATICFILES_STORAGE = 'myproject.storage.S3Storage'
```

一旦完成这个，你所要做的就是运行 `collectstatic`，然后你的静态文件将被你的存储后端推送到S3 上。如果以后你需要切换到一个不同的存储提供商，你只需简单地修改你的 `STATICFILES_STORAGE` 设置。

关于如何编写这些后端的细节，请参见编写一个自定义的存储系统。有第三方的应用提供存储后端，它们支持许多常见的文件存储API。一个不错的入口是djangopackages.com 的概览。

了解更多

关于 `django.contrib.staticfiles` 中包含的设置、命令、模板标签和其它细节，参见 staticfiles 参考。

> 译者：Django 文档协作翻译小组，原文：Deploying static files。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

错误报告

当你运行一个公开站点时，你应该始终关闭 `DEBUG` 设置。这会使你的服务器运行得更快，也会防止恶意用户看到由错误页面展示的一些应用细节。

但是，运行在 `DEBUG` 为 `False` 的情况下，你不会看到你的站点所生成的错误 -- 每个人都只能看到公开的错误页面。你需要跟踪部署的站点上的错误，所以可以配置Django来生成带有错误细节的报告。

报告邮件

服务器错误

`DEBUG` 为 `False` 的时候，无论什么时候代码产生了未处理的异常，并且出现了服务器内部错误（HTTP状态码 500），Django 都会给 `ADMINS` 设置中的用户发送邮件。这会向管理员提供任何错误的及时通知。 `ADMINS` 会得到一份错误的描述，完整的Python traceback，以及HTTP请求和导致错误的详细信息。

注意

为了发送邮件，DJango需要一些设置来告诉它如何连接到邮件服务器。最起码，你需要指定 `EMAIL_HOST` ，可能需要 `EMAIL_HOST_USER` 和 `EMAIL_HOST_PASSWORD` ，尽管所需的其他设置可能也依赖于你的邮件服务器的配置。邮件相关设置的完整列表请见 *Django设置文档*。

Django通常从root@localhost发送邮件。但是一些邮件提供商会拒收所有来自这个地址的邮件。修改 `SERVER_EMAIL` 设置可以使用不同的发信人地址。

将收信人的邮箱地址放入 `ADMINS` 设置中来激活这一行为。

另见

服务器错误邮件使用日志框架来发送，所以你可以通过 自定义你的日志配置自定义这一行为。

404错误

也可以配置Django来发送关于死链的邮件（404"找不到页面"错误）。Django在以下情况发送404错误的邮件：

- `DEBUG` 为 `False` ；
- 你的 `MIDDLEWARE_CLASSES` 设置含有 `django.middleware.common.BrokenLinkEmailsMiddleware` 。

如果符合这些条件，无论什么时候你的代码产生404错误，并且请求带有referer， Django 都会给 `MANAGERS` 中的用户发送邮件。 (It doesn't bother to email for 404s that don't have a referer – those are usually just people typing in broken URLs or broken Web 'bots).

注意

`BrokenLinkEmailsMiddleware` 必须出现在其它拦截404错误的中间件之前，比如 `LocaleMiddleware` 或者 `FlatpageFallbackMiddleware` 。把它放在你的 `MIDDLEWARE_CLASSES` 设置的最上面。

你可以通过调整 `IGNORABLE_404_URLS` 设置，告诉Django停止报告特定的404错误。它应该为一个元组，含有编译后的正则表达式对象。例如：

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'\.(php|cgi)$'),
    re.compile(r'^/phpmyadmin/'),
)
```

在这个例子中，任何以 `.php` 或者 `.cgi` 结尾URL的404错误都不会报告。任何以 `/phpmyadmin/` 开头的URL也不会。

下面的例子展示了如何排除一些浏览器或爬虫经常请求的常用URL：

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'^/apple-touch-icon.*\.png$'),
    re.compile(r'^/favicon\.ico$'),
    re.compile(r'^/robots\.txt$'),
)
```

（要注意这些是正则表达式，所以需要在句号前面添加反斜线来对它转义。）

如果你打算进一步自定义 `django.middleware.common.BrokenLinkEmailsMiddleware` 的行为（比如忽略来自web爬虫的请求），你应该继承它并覆写它的方法。

另见

404错误使用日志框架来记录。通常，日志记录会被忽略，但是你可以通过编写合适的处理器和配置日志，将它们用于错误报告。

过滤错误报告

过滤敏感的信息

错误报告对错误的调试及其有用，所以对于这些错误，通常它会尽可能多的记录下相关信息。例如，通常DJango会为产生的异常记录完整的traceback，traceback 帧的每个局部变量，以及 `HttpRequest` 的属性。

然而，有时特定的消息类型十分敏感，并不适合跟踪消息，比如用户的密码或者信用卡卡号。所以Django提供一套函数装饰器，来帮助你控制需要在生产环境（也就是 `DEBUG` 为 `False` 的情况）中的错误报告中过滤的消息：`sensitive_variables()` 和 `sensitive_post_parameters()`。

`sensitive_variables`(**variables*)[source]

如果你的代码中一个函数（视图或者常规的回调）使用可能含有敏感信息的局部变量，你可能需要使用 `sensitive_variables` 装饰器，来阻止错误报告包含这些变量的值。

```
from django.views.decorators.debug import sensitive_variables

@sensitive_variables('user', 'pw', 'cc')
def process_info(user):
    pw = user.pass_word
    cc = user.credit_card_number
    name = user.name
    ...
```

在上面的例子中，`user`，`pw` 和 `cc` 变量的值会在错误报告中隐藏并且使用星号(**) 来代替，虽然 `name` 变量的值会公开。

要想有顺序地在错误报告中隐藏一个函数的所有局部变量，不要向 `sensitive_variables` 装饰器提供任何参数：

```
@sensitive_variables()
def my_function():
    ...
```

使用多个装饰器的时候

如果你想要隐藏的变量也是一个函数的参数（例如，下面例子中的 `user`），并且被装饰的函数有多个装饰器，你需要确保将 `@sensitive_variables` 放在装饰器链的顶端。这种方法也会隐藏函数参数，尽管它通过其它装饰器传递：

```
@sensitive_variables('user', 'pw', 'cc')
@some_decorator
@another_decorator
def process_info(user):
    ...
```

`sensitive_post_parameters`(**parameters*)[source]

如果你的代码中一个视图接收到了可能带有敏感信息的，带有 `POST 参数` 的 `HttpRequest` 对象，你可能需要使用 `sensitive_post_parameters` 装饰器，来阻止错误报告包含这些参数的值。

```
from django.views.decorators.debug import sensitive_post_parameters

@sensitive_post_parameters('pass_word', 'credit_card_number')
def record_user_profile(request):
    UserProfile.create(user=request.user,
                       password=request.POST['pass_word'],
                       credit_card=request.POST['credit_card_number'],
                       name=request.POST['name'])
    ...
```

在上面的例子中，`pass_word` 和 `credit_card_number` POST参数的值会在错误报告中隐藏并且使用星号(**) 来代替，虽然 `name` 变量的值会公开。

要想有顺序地在错误报告中隐藏一个请求的所有POST 参数，不要向 `sensitive_post_parameters` 装饰器提供任何参数：

```
@sensitive_post_parameters()
def my_view(request):
    ...
```

所有POST参数按顺序被过滤出特定 `django.contrib.auth.views` 视图的错误报告（`login`，`password_reset_confirm`，`password_change`，`add_view` 和 `auth` 中的 `user_change_password`），来防止像是用户密码这样的敏感信息的泄露。

自定义错误报告

所有 `sensitive_variables()` 和 `sensitive_post_parameters()` 分别用敏感变量的名字向被装饰的函数添加注解，以及用POST敏感参数的名字向 `HttpRequest` 对象添加注解，以便在错误产生时可以随后过滤掉报告中的敏感信息。Django的默认错误包告过滤器 `django.views.debug.SafeExceptionReporterFilter` 会完成实际的过滤操作。产生错误报告的时候，这个过滤器使用装饰器的注解来将相应的值替换为星号 (**)。如果你希望为你的整个站点覆写或自定义这一默认的属性，你需要定义你自己的过滤器类，并且通过 `DEFAULT_EXCEPTION_REPORTER_FILTER` 设置来让Django使用它。

```
DEFAULT_EXCEPTION_REPORTER_FILTER = 'path.to.your.CustomExceptionReporterFilter'
```

你也可能会以更精细的方式来控制在提供的视图中使用哪种过滤器，通过设置 `HttpRequest` 的 `exception_reporter_filter` 属性。

```
def my_view(request):
    if request.user.is_authenticated():
        request.exception_reporter_filter = CustomExceptionReporterFilter()
    ...
```

你的自定义过滤器类需要继承自 `django.views.debug.SafeExceptionReporterFilter`，并且可能需要覆写以下方法：

class `SafeExceptionReporterFilter` [source]

`SafeExceptionReporterFilter.``is_active` (*request*)[source]

如果其它方法中操作的过滤器已激活，返回 `True`。如果 `DEBUG` 为 `False`，通常过滤器是激活的。

`SafeExceptionReporterFilter.``get_request_repr` (*request*)

Returns the representation string of the request object, that is, the value that would be returned by `repr(request)`, except it uses the filtered dictionary of POST parameters as determined by `SafeExceptionReporterFilter.get_post_parameters()`.

`SafeExceptionReporterFilter.``get_post_parameters` (*request*)[source]

返回过滤后的POST参数字典。通常它会把敏感参数的值以星号 (**)替换。

`SafeExceptionReporterFilter.``get_traceback_frame_variables` (*request*, *tb_frame*)[source]

返回过滤后的，所提供traceback帧的局部变量的字典。通常它会把敏感变量的值以星号 (**) 替换。

另见

你也可以通过编写自定义的*exception middleware*来建立自定义的错误报告。如果你编写了自定义的错误处理器，模拟Django内建的错误处理器，只在 `DEBUG` 为 `False` 时报告或记录错误是个好主意。

> 译者：Django 文档协作翻译小组，原文：Tracking code errors by email。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Admin

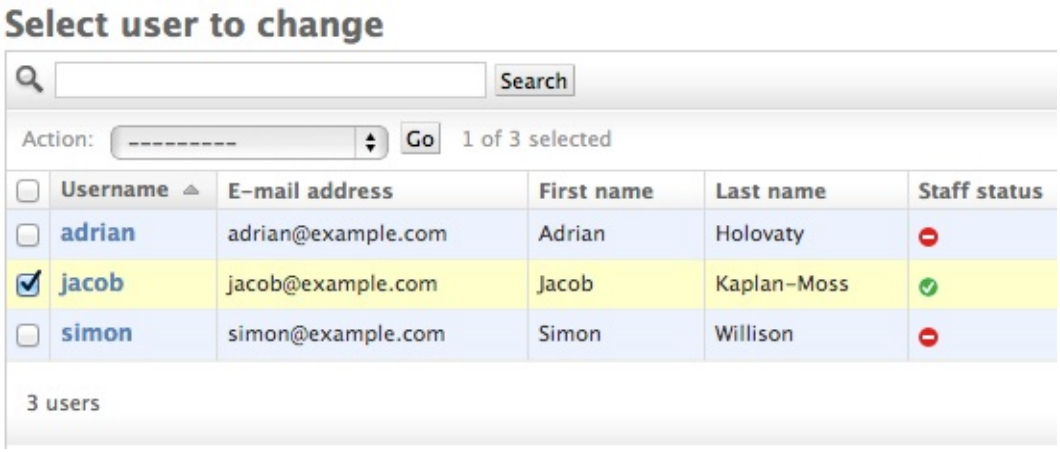
Django 最受欢迎的特性之一 —— 自动生成的Admin 界面的所有内容：

管理操作

简而言之，Django管理后台的基本流程是，“选择一个对象并改变它”。在大多数情况下，这是非常适合的。然而当你一次性要对多个对象做相同的改变，这个流程是非常的单调乏味的。

在这些例子中，Django管理后台可以让你实现和注册“操作” —— 仅仅只是一个以已选中对象集合为参数的回调函数。

在Django自带的管理页面中都能看到这样的例子。Django在所有的模型中自带了一个“删除所选对象”操作。例如，下面是 `django.contrib.auth` app 在Django's创建的用户模型:



警告

“删除所选对象”的操作由于性能因素使用了 `QuerySet.delete()`，这里有个附加说明：它不会调用你模型的 `delete()` 方法。

如果你想覆写这一行为，编写自定义操作，以你的方式实现删除就可以了 -- 例如，对每个已选择的元素调用 `Model.delete()`。

关于整体删除的更多信息，参见对象删除的文档。

继续阅读，来弄清楚如何向列表添加你自己的操作。

编写操作

通过示例来解释操作最为简单，让我们开始吧。

操作的一个最为普遍的用例是模型的整体更新。考虑带有 `Article` 模型的简单新闻应用：

```
from django.db import models

STATUS_CHOICES = (
    ('d', 'Draft'),
    ('p', 'Published'),
    ('w', 'Withdrawn'),
)

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()
    status = models.CharField(max_length=1, choices=STATUS_CHOICES)

    def __str__(self):              # __unicode__ on Python 2
        return self.title
```

我们可能在模型上执行的一个普遍任务是，将文章状态从“草稿”更新为“已发布”。我们在后台一次处理一篇文章非常轻松，但是如果我们想要批量发布一些文章，会非常麻烦。所以让我们编写一个操作，可以让我们将一篇文章的状态修改为“已发布”。

编写操作 函数

首先，我们需要定义一个函数，当后台操作被点击触发的时候调用。操作函数，跟普通的函数一样，需要接收三个参数：

- 当前的 `ModelAdmin`
- 表示当前请求的 `HttpRequest`
- 含有用户所选的对象集合的 `QuerySet`

我们用于发布这些文章的函数并不需要 `ModelAdmin` 或者请求对象，但是我们会用到查询集：

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
```

注意

为了性能最优，我们使用查询集的*update 方法*。其它类型的操作可能需要分别处理每个对象；这种情况下我们需要对查询集进行遍历：

```
for obj in queryset:
    do_something_with(obj)
```

编写操作的全部内容实际上就这么多了。但是，我们要进行一个可选但是有用的步骤，在后台给操作起一个“非常棒”的标题。通常，操作以“Make published”的方式出现在操作列表中 -- 所有空格被下划线替换后的函数名称。这样就很好了，但是我们可以提供一个更好、更人性化的名称，通过向 `make_published` 函数添加 `short_description` 属性：

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"
```

注意

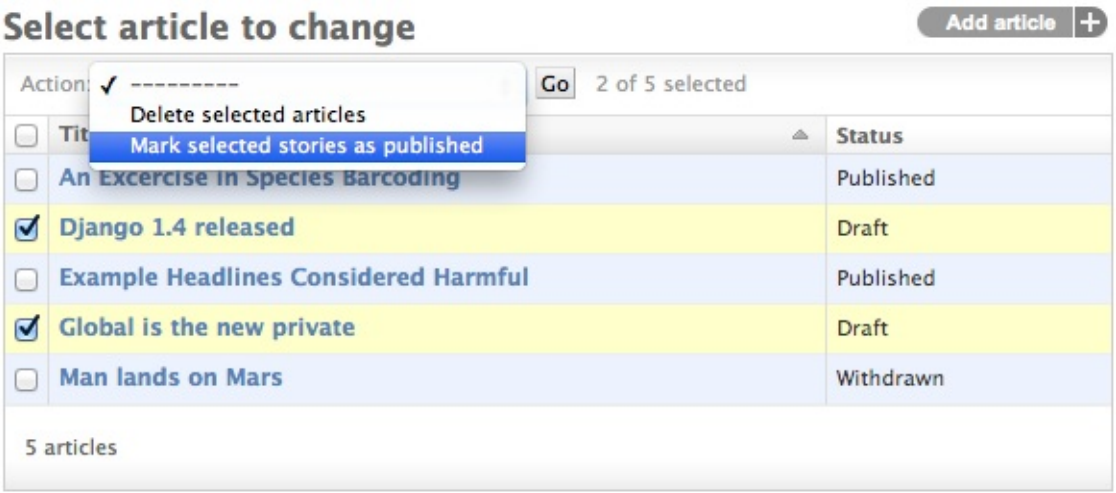
这看起来可能会有点熟悉；admin的 `list_display` 选项使用同样的技巧，为这里注册的回掉函数来提供人类可读的描述。

添加操作到 `ModelAdmin`

接下来，我们需要把操作告诉 `ModelAdmin` 。它和其他配置项的工作方式相同。所以，带有操作及其注册的完整的 `admin.py` 看起来像这样：

```
from django.contrib import admin
from myapp.models import Article

def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"

class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'status']
    ordering = ['title']
    actions = [make_published]

admin.site.register(Article, ArticleAdmin)
```

这段代码会向我们提供admin的更改列表，看起来像这样：



这就是全部内容了。如果你想编写自己的操作，你现在应该知道怎么开始了。这篇文档的剩余部分会介绍更多高级技巧。

在操作中处理错误

如果你预见到，运行你的操作时可能出现一些错误，你应该以优雅的方式向用户通知这些错误。也就是说，异常处理和使用 `django.contrib.admin.ModelAdmin.message_user()` 可以在响应中展示用户友好的问题描述。

操作的高级技巧

对于进一步的选择，你可以使用一些额外的选项。

ModelAdmin上的操作 `ModelAdmin`

上面的例子展示了定义为一个简单函数的 `make_published` 操作。这真是极好的，但是以视图的代码设计角度来看，它并不完美：由于操作与 `Article` 紧密耦合，不如将操作直接绑定到 `ArticleAdmin` 对象上更有意义。

这样做十分简单：

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    actions = ['make_published']

    def make_published(self, request, queryset):
        queryset.update(status='p')
    make_published.short_description = "Mark selected stories as published"
```

首先注意，我们将 `make_published` 放到一个方法中，并重命名 `modeladmin` 为 `self`，其次，我们现在将 `'make_published'` 字符串放进了 `actions`，而不是一个直接的函数引用。这样会让 `ModelAdmin` 将这个操作视为方法。

将操作定义为方法，可以使操作以更加直接、符合语言习惯的方式来访问 `ModelAdmin`，调用任何admin提供的方法。

例如，我们可以使用 `self` 来向用户发送消息，告诉她操作成功了：

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    def make_published(self, request, queryset):
        rows_updated = queryset.update(status='p')
        if rows_updated == 1:
            message_bit = "1 story was"
        else:
            message_bit = "%s stories were" % rows_updated
        self.message_user(request, "%s successfully marked as published." % message_bit)
```

这会使动作与后台在成功执行动作后做的事情相匹配：

提供中间页面的操作

通常，在执行操作之后，用户会简单地通过重定向返回到之前的修改列表页面中。然而，一些操作，尤其是更加复杂的操作，需要返回一个中间页面。例如，内建的删除操作，在删除选中对象之前需要向用户询问来确认。

要提供中间页面，只要从你的操作返回 `HttpResponse` （或其子类）就可以了。例如，你可能编写了一个简单的导出函数，它使用了Django的序列化函数来将一些选中的对象转换为JSON：

```
from django.http import HttpResponse
from django.core import serializers

def export_as_json(modeladmin, request, queryset):
    response = HttpResponse(content_type="application/json")
    serializers.serialize("json", queryset, stream=response)
    return response
```

通常，上面的代码的实现方式并不是很好。大多数情况下，最佳实践是返回 `HttpResponseRedirect` ，并且使用户重定向到你编写的视图中，向GET查询字符串传递选中对象的列表。这需要你在中间界面上提供复杂的交互逻辑。例如，如果你打算提供一个更加复杂的导出函数，你会希望让用户选择一种格式，以及可能在导出中包含一个含有字段的列表。最佳方式是编写一个小型的操作，简单重定向到你的自定义导出视图中：

```
from django.contrib import admin
from django.contrib.contenttypes.models import ContentType
from django.http import HttpResponseRedirect

def export_selected_objects(modeladmin, request, queryset):
    selected = request.POST.getlist(admin.ACTION_CHECKBOX_NAME)
    ct = ContentType.objects.get_for_model(queryset.model)
    return HttpResponseRedirect("/export/?ct=%s&ids=%s" % (ct.pk, ",".join(selected)))
```

就像你看到的那样，这个操作是最简单的部分；所有复杂的逻辑都在你的导出视图里面。这需要处理任何类型的对象，所以需要处理 `ContentType` 。

这个视图的编写作为一个练习留给读者。

在整个站点应用操作

`AdminSite.``add_action`(*action*[, *name*])

如果一些操作对管理站点的任何对象都可用的话，是非常不错的 -- 上面所定义的导出操作是个不错的备选方案。你可以使用 `AdminSite.add_action()` 让一个操作在全局都可以使用。例如：

```
from django.contrib import admin

admin.site.add_action(export_selected_objects)
```

这样， `export_selected_objects` 操作可以在全局使用，名称为“export*selected_objects*”。你也可以显式指定操作的名称 – 如果你想以编程的方式*移除这个操作*_](#disabling-admin-actions) – 通过向 `AdminSite.add_action()` 传递第二个参数：

```
admin.site.add_action(export_selected_objects, 'export_selected')
```

禁用操作

有时你需要禁用特定的操作 -- 尤其是注册的站点级操作 -- 对于特定的对象。你可以使用一些方法来禁用操作：

禁用整个站点的操作

`AdminSite.``disable_action`(*name*)

如果你需要禁用站点级操作，你可以调用 `AdminSite.disable_action()`。

例如，你可以使用这个方法来移除内建的“删除选中的对象”操作：

```
admin.site.disable_action('delete_selected')
```

一旦你执行了上面的代码，这个操作不再对整个站点中可用。

然而，如果你需要为特定的模型重新启动在全局禁用的对象，把它显式放在 `ModelAdmin.actions` 列表中就可以了：

```
# Globally disable delete selected
admin.site.disable_action('delete_selected')

# This ModelAdmin will not have delete_selected available
class SomeModelAdmin(admin.ModelAdmin):
    actions = ['some_other_action']
    ...

# This one will
class AnotherModelAdmin(admin.ModelAdmin):
    actions = ['delete_selected', 'a_third_action']
    ...
```

为特定的ModelAdmin禁用所有操作 `ModelAdmin`

如果你想批量移除所提供 `ModelAdmin` 上的所有操作，可以把 `ModelAdmin.actions` 设置为 `None` ：

```
class MyModelAdmin(admin.ModelAdmin):
    actions = None
```

这样会告诉 `ModelAdmin` ，不要展示或者允许任何操作，包括站点级操作。

按需启用或禁用操作

`ModelAdmin.``get_actions` (*request*)

最后，你可以通过覆写 `ModelAdmin.get_actions()` ，对每个请求（每个用户）按需开启或禁用操作。

这个函数返回包含允许操作的字典。字典的键是操作的名称，值是 `(function, name, short_description)` 元组。

多数情况下，你会按需使用这一方法，来从超类中的列表移除操作。例如，如果我只希望名称以'J'开头的用户可以批量删除对象，我可以执行下面的代码：

```
class MyModelAdmin(admin.ModelAdmin):
    ...

    def get_actions(self, request):
        actions = super(MyModelAdmin, self).get_actions(request)
        if request.user.username[0].upper() != 'J':
            if 'delete_selected' in actions:
                del actions['delete_selected']
        return actions
```

> 译者：Django 文档协作翻译小组，原文：Admin actions。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Django管理文档生成器

Django的 `admindocs` 应用从模型、视图、模板标签以及模板过滤器中，为任何 `INSTALLED_APPS` 中的应用获取文档。并且让文档可以在 `Django admin` 中使用。

在某种程度上，你可以使用 `admindocs` 来快为你自己的代码生成文档。这个应用的功能十分有限，然而它主要用于文档模板、模板标签和过滤器。例如，需要参数的模型方法在文档中会有意地忽略，因为它们不能从模板中调用。这个应用仍旧有用，因为它并不需要你编写任何额外的文档（除了 `docstrings` ），并且在 `Django admin` 中使用很方便。

概览

要启用 `admindocs` ，你需要执行以下步骤：

- 向 `INSTALLED_APPS` 添加 `django.contrib.admindocs` 。
- 向你的 `urlpatterns` 添加(`r'^admin/doc/'` , `include('django.contrib.admindocs.urls')`)。 确保它在 `r'^admin/'` 这一项 之前包含，以便 `/admin/doc/` 的请求不会被后面的项目处理。
- 安装 `docutils` Python 模块 (http://docutils.sf.net/)。
- 可选的： 使用 `admindocs` 的书签功能需要安装 `django.contrib.admindocs.middleware.XViewMiddleware` 。

一旦完成这些步骤，你可以开始通过你的 `admin` 接口和点击在页面右上方的“Documentation”链接来浏览文档。

文档助手

下列特定的标记可以用于你的 `docstrings` ，来轻易创建到其他组件的超链接：

Django Component	reStructuredText roles
Models	:model:`app_label.ModelName`
Views	:view:`app_label.view_name`
Template tags	:tag:`tagname`
Template filters	:filter:`filtername`
Templates	:template:`path/to/template.html`

模型参考

`admindocs` 页面的 `models` 部分描述了系统中每个模型，以及所有可用的字段和方法（不带任何参数）。虽然模型的属性没有任何参数，但他们没有列出。和其它模型的关联以超链接形式出现。描述由字段上的 `help_text` 属性，或者从模型方法的 `docstrings` 导出。

带有有用文档的模型看起来像是这样：

```
class BlogEntry(models.Model):
    """
    Stores a single blog entry, related to :model:`blog.Blog` and
    :model:`auth.User`.

    """
    slug = models.SlugField(help_text="A short label, generally used in URLs.")
    author = models.ForeignKey(User)
    blog = models.ForeignKey(Blog)
    ...

    def publish(self):
        """Makes the blog entry live on the site."""
        ...
```

视图参考

你站点中的每个URL都在·页面中有一个单独的记录，点击提供的URL会向你展示相应的视图。有一些有用的东西，你可以在你的视图函数的·中记录：

- 视图所做工作的一个简短的描述。
- 上下文，或者是视图的模板中可用变量的列表。
- 用于当前视图的模板的名称。

例如：

```
from django.shortcuts import render

from myapp.models import MyModel

def my_view(request, slug):
    """
    Display an individual :model:`myapp.MyModel`.

    **Context**

    ``mymodel``
        An instance of :model:`myapp.MyModel`.

    **Template:**

    :template:`myapp/my_template.html`

    """
    context = {'mymodel': MyModel.objects.get(slug=slug)}
    return render(request, 'myapp/my_template.html', context)
```

模板标签和过滤器参考

`admindocs` 的 `tags` 和 `filters` 部分描述了Django自带的所有标签和过滤器（事实上，内建的标签参考 和 内建的过滤器参考文档直接来自于那些页面）。你创建的，或者由三方应用添加的任何标签或者过滤器，也会在这一部分中展示。

模板参考

虽然 `admindocs` 并不包含一个地方来保存模板，但如果你在结果页面中使用 `:template:`path/to/template.html`` 语法，会使用Django的模板加载器来验证该模板的路径。这是一个非常便捷的方法，来检查是否存在特定的模板，以及展示模板在文件系统的何处存放。

包含的书签

`admindocs` 页面上有一些很有用的书签：

Documentation for this page

Jumps you from any page to the documentation for the view that generates that page.

Show object ID

Shows the content-type and unique ID for pages that represent a single object.

Edit this object

Jumps to the admin page for pages that represent a single object.

为使用这些书签，你需要用带有 `is_staff` 设置为 `True` 的 `User` 登录 `Django admin`，或者安装了 `XViewMiddleware` 并且你通过 `INTERNAL_IPS` 中的IP地址访问站点。

> 译者：Django 文档协作翻译小组，原文：Admin documentation generator。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

安全

安全在Web应用开发中是一项至关重要的话题，Django提供了多种保护手段和机制：

Django安全

这份文档是 Django 的安全功能的概述。 它包括给 Django 驱动的网站一些加固建议。

跨站脚本 (XSS) 防护

XSS攻击允许用户注入客户端脚本到其他用户的浏览器里。 这通常是通过存储在数据库中的恶意脚本，它将检索并显示给其他用户，或者通过让用户点击一个链接，这将导致攻击者的 JavaScript 被用户的浏览器执行。 然而，XSS 攻击可以来自任何不受信任的源数据，如 Cookie 或 Web 服务，任何没有经过充分处理就包含在网页中的数据。

使用 Django 模板保护你免受多数 XSS 攻击。 然而，重要的是要了解它提供了什么保护及其局限性。

Django 模板会 编码特殊字符 ，这些字符在 HTML 中都是特别危险的。 虽然这可以防止大多数恶意输入的用户，但它不能完全保证万无一失。 例如，它不会防护以下内容：

```
<style class=>...</style>
```

如果 `var` 设置为 `'class1 onmouseover=javascript:func()'` ，这可能会导致在未经授权的 JavaScript 的执行，取决于浏览器如何呈现不完整的 HTML。 （对属性值使用引号可以修复这种情况。）

同样重要的是 `is_safe` 要特别小心的用在 自定义模板标签， `safe` 模板标签， `mark_safe` ，还有 autoescape 被关闭的时候。

此外，如果您使用的是模板系统输出 HTML 以外的东西，可能会有完全不同的字符和单词需要编码。

你也应该在数据库中存储 HTML 的时候要非常小心，尤其是当 HTML 被检索然后展示出来。

跨站请求伪造 (CSRF) 防护

CSRF 攻击允许恶意用户在另一个用户不知情或者未同意的情况下，以他的身份执行操作。

Django 对大多数类型的 CSRF 攻击有内置的保护，在适当情况下你可以开启并使用它 。 然而，对于任何解决技术，都有它的局限性。 例如，CSRF 模块可以在全局范围内或为特定视图被禁用 。 您应该只在您知道在做什么的情况下操作。 还有其他 限制 如果你的网站有子域名并且在你的控制之外。

CSRF 防护 是通过检查每个 POST 请求的一个随机数（nonce）来工作。这确保了恶意用户不能简单“回放”你网站上面表单的POST，以及让另一个登录的用户无意中提交表单。恶意用户必须知道这个随机数，它是用户特定的（存在cookie里）。

使用 *HTTPS*来部署的时候，`CsrfViewMiddleware` 会检查HTTP referer协议头是否设置为同源的URL（包括子域和端口）。因为HTTPS提供了附加的安全保护，转发不安全的连接请求时，必须确保链接使用 HTTPS，并使用HSTS支持的浏览器。

使用 `csrf_exempt` 装饰器来标记视图时，要非常小心，除非这是极其必要的。

SQL 注入保护

SQl注入是一种攻击类型，恶意用户可以在系统数据库中执行任意SQL代码。这可能会导致记录删除或者数据泄露。

通过使用Django的查询集，产生的SQL会由底层数据库驱动正确地转义。然而，Django也允许开发者编写原始查询或者执行自定义 *sql*。这些功能应该谨慎使用，并且你应该时刻小心正确转义任何用户可以控制的参数。另外，你在使用 `extra()` 的时候应该谨慎行事。

点击劫持保护

点击劫持是一类攻击，恶意站点在一个frame中包裹了另一个站点。这类攻击可能导致用户被诱导在目标站点做出一些无意识的行为。

Django在 `X-Frame-Options 中间件` 的表单中中含有 点击劫持保护 ，它在支持的浏览器中可以保护站点免于在frame中渲染。也可以在每个视图中禁止这一保护，或者配置要发送的额外的协议头。

对于任何不需要将页面包装在三方站点的frame中，或者只需要包含它的一部分的站点，都强烈推荐启用这一中间件。

SSL/HTTPS

把你的站点部署在HTTPS下总是更安全的，尽管在所有情况下不都有效。如果不这样，恶意的网络用户可能会嗅探授权证书，或者其他在客户端和服务端之间传输的信息，或者一些情况下 -- 活跃的网络攻击者 -- 会修改在两边传输的数据。

如果你想要使用HTTPS提供的保护，并且在你的服务器上开启它，你需要遵循一些额外的步骤：

- 如果必要的话，设置 `SECURE_PROXY_SSL_HEADER` ，确保你已经彻底了解警告。未能实现它会导致CSRF方面的缺陷，也是很危险的！

- 设置重定向，以便HTTP下的请求可以重定向到HTTPS。

  这可以通过自定义的中间件来实现。请注意 `SECURE_PROXY_SSL_HEADER` 下的警告。对于反向代理的情况，配置web主服务器来重定向到HTTPS或许是最简单也许是最安全的做法。

- 使用“安全的”cookie。

  如果浏览器的连接一开始通过HTTP，这是大多数浏览器的通常情况，已存在的cookie可能会被泄露。因此，你应该将 `SESSION_COOKIE_SECURE` 和 `CSRF_COOKIE_SECURE` 设置为 `True` 。这会使浏览器只在HTTPS连接中发送这些cookie。要注意这意味着会话在HTTP下不能工作，并且CSRF保护功能会在HTTP下阻止接受任何POST数据（如果你把所有HTTP请求都重定向到HTTPS之后就没问题了）。

- 使用 HTTP 强制安全传输 (HSTS)

  HSTS 是一个HTTP协议头，它通知浏览器，到特定站点的所有链接都一直使用HTTPS。通过和重定向HTTP请求到HTTPS一起使用，确保连接总是享有附加的SSL安全保障，由一个已存在的成功的连接提供。HSTS通常在web服务器上面配置。

Host 协议头验证

在某些情况下，Django使用客户端提供的 `Host` 协议头来构造URL。虽然这些值可以被审查，来防止跨站脚本攻击（XSS），但是一个假的 `Host` 值可以用于跨站请求伪造（CSRF），有害的缓存攻击，以及email中的有害链接。

Because even seemingly-secure web server configurations are susceptible to fake `Host` headers, Django validates `Host` headers against the `ALLOWED_HOSTS` setting in the `django.http.HttpRequest.get_host()` method.

验证只通过 `get_host()` 来应用；如果你的代码从 `request.META` 中直接访问 `Host` 协议头，就会绕开这一安全防护。

详见完整的 `ALLOWED_HOSTS` 文档。

Warning

Previous versions of this document recommended configuring your web server to ensure it validates incoming HTTP `Host` headers. While this is still recommended, in many common web servers a configuration that seems to validate the `Host` header may not in fact do so. For instance, even if Apache is configured such that your Django site is served from a non-default virtual host with the `ServerName` set, it is still possible for an HTTP request to match this virtual host and supply a fake `Host` header. Thus, Django now requires that you set `ALLOWED_HOSTS` explicitly rather than relying on web server configuration.

另外，就像1.3.1，如果你的配置需要它的话，Django 需要你显式开启对 `X-Forwarded-Host` 协议头的支持(通过 `USE_X_FORWARDED_HOST` 这只)。

Session 会话安全

类似于部署在站点上的*CSRF 限制* 使不受信任的用户不能访问任何子域， `django.contrib.sessions` 也有一些限制。详见安全中会话的话题指南。

用户上传的内容

注意

考虑在云服务器或*CDN*上面部署静态文件来避免一些此类问题。

- 如果你的站点接受上传文件，强烈推荐你在web服务器配置中，将这些上传限制为合理的大小，来避免拒绝服务（DOS）攻击。在Apache中，这可以简单地使用LimitRequestBody指令。
- 如果你自己处理静态文件，确保像Apache的 `mod_php` 的处理器已关闭，它会将静态文件执行为代码。你并不希望用户能够通过上传和请求一个精心构造的文件来执行任意代码。
- Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML. This file will pass verification of the library that Django uses for `ImageField` image processing (Pillow). When this file is subsequently displayed to a user, it may be displayed as HTML depending on the type and configuration of your web server.

  No bulletproof technical solution exists at the framework level to safely validate all user uploaded file content, however, there are some other steps you can take to mitigate these attacks:

  - One class of attacks can be prevented by always serving user uploaded content from a distinct top-level or second-level domain. This prevents any exploit blocked by same-origin policy protections such as cross site scripting. For example, if your site runs on `example.com`, you would want to serve uploaded content (the `MEDIA_URL` setting) from something like `usercontent-example.com`. It's *not* sufficient to serve content from a subdomain like `usercontent.example.com`.
  - 除此之外，应用可以选择为用户上传的文件定义一个允许的文件扩展名的白名单，并且配置web服务器直处理这些文件。

额外的安全话题

虽然Django提供了开箱即用的，良好的安全保护，但是合理地部署你的应用，以及利用web服务器、操作系统和其他组件的安全保护仍然很重要。

- 确保你的Python代码在web服务器的根目录外。这会确保你的Python代码不会意外被解析为纯文本（或者意外被执行）。
- 小心处理任何用户上传的文件。
- Django并不限制验证用户的请求。要保护对验证系统的暴力破解攻击，你可以考虑部署一个DJango的插件或者web服务器模块来限制这些请求。
- 秘密保存 `SECRET_KEY` 。
- 使用防火墙来限制缓存系统和数据库的访问是个好主意。

> 译者：Django 文档协作翻译小组，原文：Security overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

安全问题归档

Django的开发小组坚定地承诺，为报告和公开安全相关问题负责，这在*Django的安全问题*中列出。

作为承诺的一部分，我们保留了下面的问题的历史列表，这些问题已经被解决和公开。对于每个问题，下面的列表包含日期、简短的描述、CVE 标识符、受影响的版本列表、完整的页面链接以及相应补丁的连接。

有一些重要的附加说明：

- 列出的受影响版本只包含了在漏洞公开时期的Django稳定的安全支持发行版。这意味着，老的版本（安全支持已经过期），以及预发行版本（alpha/beta/RC）在漏洞公开的时期也可能会受影响，但是没有列出。
- Django项目偶尔会发布安全公告，指出潜在的安全问题，可能会由不合理的配置或其他Django本身以外的问题产生。这些公告中有一些收到了CVE；这种情况下，它们会在这里列出来，但是没有任何附加的补丁或者发行版，只有描述、公开信息和CVE。

Issues prior to Django's security process

一些安全问题在Django具有规范化的安全处理流程之前被修复。对于这些问题，可能不会发布新的发行版，也不会分配CVE。

August 16, 2006 - CVE-2007-0404

CVE-2007-0404: 翻译框架中的文件名验证问题。Full description

Versions affected

- Django 0.90 (patch)
- Django 0.91 (patch)
- Django 0.95 (patch) (released January 21 2007)

January 21, 2007 - CVE-2007-0405

CVE-2007-0405: 已认证用户的可见“缓存”。Full description

Versions affected

- Django 0.95 (patch)

Issues under Django's security process

所有其它的安全问题都已经在Django安全处理流程下的版本中解决。下面会列出来：

October 26, 2007 - CVE-2007-5712

CVE-2007-5712: 通过任意大尺寸 `Accept-Language` 协议头的拒绝服务攻击。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

May 14, 2008 - CVE-2008-2302

CVE-2008-2302: 通过admin登录重定向的XSS。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

September 2, 2008 - CVE-2008-3909

CVE-2008-3909: 通过在admin登录状态下保存POST数据的CSRF。Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

July 28, 2009 - CVE-2009-2659

CVE-2009-2659: 开发服务器的媒体处理器上的拒绝服务攻击。Full description

Versions affected

- Django 0.96 (patch)
- Django 1.0 (patch)

October 9, 2009 - CVE-2009-3965

CVE-2009-3965: 通过执行异常正则表达式的拒绝服务攻击。Full description

Versions affected

- Django 1.0 (patch)
- Django 1.1 (patch)

September 8, 2010 - CVE-2010-3082

CVE-2010-3082: 通过不安全cookie值的XSS。Full description

Versions affected

- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4534

CVE-2010-4534: 管理界面上的信息泄露。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4535

CVE-2010-4535: 密码重置机制上的拒绝服务攻击。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0696

CVE-2011-0696: 通过伪造HTTP协议头的XSS。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0697

CVE-2011-0697: 通过未检查的名称或者上传文件的XSS。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0698

CVE-2011-0698: Windows上通过不正确的目录分隔符处理的目录遍历。Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

September 9, 2011 - CVE-2011-4136

CVE-2011-4136:使用memory-cache-backed会话时的会话操纵。Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4137

CVE-2011-4137: 通过 `URLField.verify_exists` 的拒绝服务攻击。Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4138

CVE-2011-4138: 通过 `URLField.verify_exists` 的信息泄露/任何请求发布。Full description

Versions affected

- Django 1.2: (patch)
- Django 1.3: (patch)

September 9, 2011 - CVE-2011-4139

CVE-2011-4139: `Host` 协议头缓存污染。 Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4140

CVE-2011-4140:通过 `Host` 协议头的潜在CSRF威胁。Full description

Versions affected

这个通知只是一个公告，没有任何补丁发布。

- Django 1.2
- Django 1.3

July 30, 2012 - CVE-2012-3442

CVE-2012-3442: 通过验证重定向模式失败的XSS。Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3443

CVE-2012-3443: 通过压缩的图像文件的拒绝服务u攻击。Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3444

CVE-2012-3444:通过大尺寸图像文件的拒绝服务攻击。Full description

Versions affected

- Django 1.3 (patch)

- Django 1.4 (patch)

October 17, 2012 - CVE-2012-4520

CVE-2012-4520: `Host` 协议头污染。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 1

对 `Host` 协议头处理的额外加固。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 2

对重定向验证的额外加固。Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

February 19, 2013 - No CVE

对 `Host` 协议头处理的额外加固。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-1664/1665

CVE-2013-1664 and CVE-2013-1665: 对Python XML库的基于实体的攻击。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0305

CVE-2013-0305: 通过admin历史记录的信息泄露。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0306

CVE-2013-0306: 通过表单集 `max_num` 的拒绝服务攻击。Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

August 13, 2013 - Awaiting CVE 1

(CVE not yet issued): 通过admin受信任的 `URLField` 值的XSS。Full description

Versions affected

- Django 1.5 (patch)

August 13, 2013 - Awaiting CVE 2

(CVE not yet issued):可能的XSS漏洞，通过未验证的URL重定向模式。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 10, 2013 - CVE-2013-4315

CVE-2013-4315 通过 `ssi` 模板标签的目录遍历。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 14, 2013 - CVE-2013-1443

CVE-2013-1443: 通过长密码的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch and Python compatibility fix)
- Django 1.5 (patch)

April 21, 2014 - CVE-2014-0472

CVE-2014-0472: 使用 `reverse()` 的非预期代码执行。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0473

CVE-2014-0473: 匿名页面的缓存可能会泄露CSRF标识。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0474

CVE-2014-0474: MySQL类型转换产生非预期的查询结果。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)

- Django 1.7 (patch)

May 18, 2014 - CVE-2014-1418

CVE-2014-1418: 缓存可能允许存储和处理私人数据。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

May 18, 2014 - CVE-2014-3730

CVE-2014-3730: 来源于用户输入的错误格式URL的不正确验证。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0480

CVE-2014-0480: reverse() 可能会生成指向其它域名的URL。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0481

CVE-2014-0481: 文件上传的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0482

CVE-2014-0482: RemoteUserMiddleware会话劫持。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0483

CVE-2014-0483: admin中查询集操作产生的数据泄露。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0219

CVE-2015-0219: 通过下划线或者破折号合并产生的WSGI协议头欺骗。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0220

CVE-2015-0220: 通过用户提供的重定向URL的可能的XSS攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)

- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0221

CVE-2015-0221: `django.views.static.serve()` 上的拒绝服务攻击。Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0222

CVE-2015-0222: 使用 `ModelMultipleChoiceField` 的数据库拒绝服务攻击。Full description

Versions affected

- Django 1.6 (patch)
- Django 1.7 (patch)

> 译者：Django 文档协作翻译小组，原文：Disclosed security issues in Django。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

点击劫持保护

点击劫持中间件和装饰器提供了简捷易用的，对点击劫持的保护。这种攻击在恶意站点诱导用户点击另一个站点的被覆盖元素时出现，另一个站点已经加载到了隐藏的 `frame` 或 `iframe` 中。

点击劫持的示例

假设一个在线商店拥有一个页面，已登录的用户可以点击“现在购买”来购买一个商品。用户为了方便，可以选择一直保持商店的登录状态。一个攻击者的站点可能在他们自己的页面上会创建一个“我喜欢Ponies”的按钮，并且在一个透明的 `iframe` 中加载商店的页面，把“现在购买”的按钮隐藏起来覆盖在“我喜欢Ponies”上。如果用户访问了攻击者的站点，点击“我喜欢Ponies”按钮会触发对“现在购买”按钮的无意识的点击，不知不觉中购买了商品。

点击劫持的防御

现代浏览器遵循X-Frame-Options协议头，它表明一个资源是否允许加载到 `frame` 或者 `iframe` 中。如果响应包含值为 `SAMEORIGIN` 的协议头，浏览器会在 `frame` 中只加载同源请求的的资源。如果协议头设置为 `DENY`，浏览器会在加载 `frame` 时屏蔽所有资源，无论请求来自于哪个站点。

Django提供了一些简单的方法来在你站点的响应中包含这个协议头：

- 一个简单的中间件，在所有响应中设置协议头。
- 一系列的视图装饰器，可以用于覆盖中间件，或者只用于设置指定视图的协议头。

如何使用

为所有响应设置X-Frame-Options

要为你站点中所有的响应设置相同的 `X-Frame-Options` 值，将 `'django.middleware.clickjacking.XFrameOptionsMiddleware'` 设置为 `MIDDLEWARE_CLASSES`：

```
MIDDLEWARE_CLASSES = (
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
)
```

这个中间件可以在startproject生成的设置文件中开启。

通常，这个中间件会为任何开放的 `HttpResponse` 设置 `X-Frame-Options` 协议头为 `SAMEORIGIN`。如果你想用 `DENY` 来替代它，要设置 `X_FRAME_OPTIONS`：

```
X_FRAME_OPTIONS = 'DENY'
```

使用这个中间件时可能会有一些视图，你并不想为它设置 `X-Frame-Options` 协议头。对于这些情况，你可以使用一个视图装饰器来告诉中间件不要设置协议头：

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe to load in a frame on any site.")
```

为每个视图设置 X-Frame-Options

Django提供了以下装饰器来为每个基础视图设置 `X-Frame-Options` 协议头。

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_deny
from django.views.decorators.clickjacking import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display in any frame!")

@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a frame if it's from the same origin as me.")
```

注意你可以在中间件的连接中使用装饰器。使用装饰器来覆盖中间件。

限制

`X-Frame-Options` 协议头只在现代浏览器中保护点击劫持。老式的浏览器会忽视这个协议头，并且需要 其它点击劫持防范技巧。

支持 X-Frame-Options 的浏览器

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

另见

浏览器对 `X-Frame-Options` 支持情况的完整列表。

> 译者：Django 文档协作翻译小组，原文：Clickjacking protection。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

加密签名

web应用安全的黄金法则是，永远不要相信来自不可信来源的数据。有时通过不可信的媒介来传递数据会非常方便。密码签名后的值可以通过不受信任的途径传递，这样是安全的，因为任何篡改都会检测的到。

Django提供了用于签名的底层API，以及用于设置和读取被签名cookie的上层API，它们是web应用中最常使用的签名工具之一。

你可能会发现，签名对于以下事情非常有用：

- 生成用于“重置我的账户”的URL，并发送给丢失密码的用户。
- 确保储存在隐藏表单字段的数据不被篡改，
- 生成一次性的秘密URL，用于暂时性允许访问受保护的资源，例如用户付费的下载文件。

保护 SECRET_KEY

当你使用 startproject创建新的Django项目时，自动生成的 settings.py文件会得到一个随机的 SECRET_KEY值。这个值是保护签名数据的密钥 -- 它至关重要，你必须妥善保管，否则攻击者会使用它来生成自己的签名值。

使用底层 API

Django的签名方法存放于 `django.core.signing` 模块。首先创建一个 `Signer` 的实例来对一个值签名：

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

这个签名会附加到字符串末尾，跟在冒号后面。你可以使用 `unsign` 方法来获取原始的值：

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

如果签名或者值以任何方式改变，会抛出 `django.core.signing.BadSignature` 异常：

```
>>> from django.core import signing
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

通常，`Signer` 类使用 `SECRET_KEY` 设置来生成签名。你可以通过向 `Signer` 构造器传递一个不同的密钥来使用它：

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

`class Signer(key=None, sep=':', salt=None)[source]`

返回一个 `signer`，它使用 `key` 来生成签名，并且使用 `sep` 来分割值。`sep` 不能是 [URL安全的base64字母表(http://tools.ietf.org/html/rfc4648#section-5)]中的字符。字母表含有数字、字母、连字符和下划线。

使用salt参数

如果你不希望对每个特定的字符串都生成一个相同的签名哈希值，你可以在 `Signer` 类中使用可选的 `salt` 参数。使用 `salt` 参数会同时用它和 `SECRET_KEY` 初始化签名哈希函数：

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
'My string'
```

以这种方法使用 `salt` 会把不同的签名放在不同的命名空间中。来自于单一命名空间（一个特定的 `salt` 值）的签名不能用于在不同的命名空间中验证相同的纯文本字符串。不同的命名空间使用不同的 `salt` 设置。这是为了防止攻击者使用在一个地方的代码中生成的签名后的字符串，作为使用不同 `salt` 来生成（和验证）签名的另一处代码的输入。

不像你的 `SECRET_KEY`，你的 `salt` 参数可以不用保密。

验证带有时间戳的值

`TimestampSigner` 是 `Signer` 的子类，它向值附加一个签名后的时间戳。这可以让你确认一个签名后的值是否在特定时间段之内被创建：

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

`class TimestampSigner(key=None, sep=':', salt=None)[source]`

`sign(value)[source]`

签名 `value`，并且附加当前的时间戳。

`unsign(value, max_age=None)[source]`

检查 `value` 是否在少于 `max_age` 秒之前被签名，如果不是则抛出 `SignatureExpired` 异常。 `max_age` 参数接受一个整数或者 `datetime.timedelta` 对象。

```
Changed in Django 1.8:

在此之前， max_age参数只接受整数。
```

保护复杂的数据结构

如果你希望保护一个列表、元组或字典，你可以使用签名模块的 `dumps` 和 `loads` 函数来实现。它们模仿了Python的 `pickle` 模块，但是在背后使用了 `JSON` 序列化。 `JSON` 可以确保即使你的 `SECRET_KEY` 被盗取，攻击者并不能利用 `pickle` 的格式来执行任意的命令：

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1NMg1b:zGcDE4-TCkaeGzLeW9UQwZesciI'
>>> signing.loads(value)
{'foo': 'bar'}
```

由于 `JSON` 的本质（列表和元组之间没有原生的区别），如果你传进来一个元组，你会从 `signing.loads(object)` 得到一个列表：

```
>>> from django.core import signing
>>> value = signing.dumps(('a','b','c'))
>>> signing.loads(value)
['a', 'b', 'c']
```

`dumps(obj, key=None, salt='django.core.signing', compress=False)[source]`

返回URL安全，sha1签名的base64压缩的JSON字符串。序列化的对象使用 `TimestampSigner` 来签名。

`loads(string, key=None, salt='django.core.signing', max_age=None)[source]`

`dumps()` 的反转，如果签名失败则抛出 `BadSignature` 异常。如果提供了 `max_age` 则会检查它（以秒为单位）。

> 译者：Django 文档协作翻译小组，原文：Cryptographic signing。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

国际化和本地化

Django 提供了一种健壮的国际化和本地化框架来帮助你实现多种语言和世界区域范围的开发。

国际化和本地化

概述

国际化和本地化的目的就是让一个网站应用能做到根据用户语种和指定格式的不同而提供不同的内容。

Django 对文本翻译, 日期、时间和数字的格式化，以及时区提供了完善的支持。

实际上，Django做了两件事：

- 由开发者和模板作者指定应用的哪些部分应该翻译，或是根据本地语种和文化进行相应的格式化。
- 根据用户的偏好设置，使用钩子将web应用本地化。

很显然，翻译取决于用户所选语言，而格式化通常取决于用户所在国家。 这些信息由浏览器通过 `Accept-Language` 协议头提供。不过确定时区就不是这么简单了。

定义

国际化和本地化通常会被混淆，这里我们对其进行简单的定义和区分：

国际化

让软件支持本地化的准备工作，通常由开发者完成。

本地化

编写翻译和本地格式，通常由翻译者完成。

更多细节详见W3C Web Internationalization FAQ、Wikipedia article和GNU gettext documentation。

> 警告
>
> 是否启用翻译和格式化分别由配置项 `USE_I18N` 和 `USE_L10N` 决定。 但是，这两个配置项都同时影响国际化和本地化。 这种情况是Django的历史因素所致。

下面几项可帮助我们更好地处理某种语言：

本地化名称

表示地域文化的名称，可以是 `ll` 格式的语种代码，也可以是 `ll_CC` 格式的语种和国家组合代码。例如：`it`，`de_AT`，`es`，`pt_BR` 。语种部分总是小写而国家部分则应是大写，中间以下划线(_)连接。

语言代码

表示语言的名称。浏览器会发送带有语言代码的 `Accept-Language` HTTP报头给服务器。例如：`it`，`de-at`，`es`，`pt-br`. 语种和国家部分都是小写，中间以破折线(-)连接。

消息文件

消息文件是纯文本文件，包含某种语言下所有可用的翻译字符串及其对应的翻译结果。消息文件以 `.po` 做为文件扩展名。

翻译字符串

可以被翻译的文字。

格式文件

针对某个地域定义数据格式的Python模块。

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

格式本地化

概览

Django的格式化系统可以在模板中使用当前地区特定的格式，来展示日期、时间和数字。也可以处理表单中输入的本地化。

当它被开启时，访问相同内容的两个用户可能会看到以不同方式格式化的日期、时间和数字，这取决于它们的当前地区的格式。

格式化系统默认是禁用的。需要在你的设置文件中设置 `USE_L10N = True` 来启用它。

注意

为了方便起见， `django-admin startproject创建的默认的` settings.py `文件包含了  [` USE_L10N = True `](../../ref/settings.html#std:setting-USE_L10N) 的设置。` 但是要注意，要开启千位分隔符的数字格式化，你需要在你的设置文件中设置 `USE_THOUSAND_SEPARATOR = True` 。或者，你也可以在你的模板中使用 `intcomma` 来格式化数字。

注意

`USE_I18N` 是另一个独立的并且相关的设置，它控制着Django是否应该开启翻译。详见翻译。

表单中的本地化识别输入

格式化开启之后，Django可以在表单中使用本地化格式来解析日期、时间和数字。也就是说，在表单上输入时，它会尝试不同的格式和地区来猜测用户使用的格式。

注意

Django对于展示数据，使用和解析数据不同的格式。尤其是，解析日期的格式不能使用 `%a` （星期名称的缩写）， `%A` （星期名称的全称）， `%b` （月份名称的缩写）， `%B` （月份名称的全称），或者 `%p` （上午/下午）。

只是使用 `localize` 参数，就能开启表单字段的本地化输入和输出：

```
class CashRegisterForm(forms.Form):
   product = forms.CharField()
   revenue = forms.DecimalField(max_digits=4, decimal_places=2, localize=True)
```

在模板中控制本地化

当你使用 `USE_L10N` 来开启格式化的时候，Django会尝试使用地区特定的格式，无论值在模板的什么位置输出。

然而，这对于本地化的值不可能总是十分合适，如果你在输出JavaScript或者机器阅读的XML，你会想要使用去本地化的值。你也可能想只在特定的模板中使用本地化，而不是任何位置都使用。

DJango提供了 `l10n` 模板库，包含以下标签和过滤器，来实现对本地化的精细控制。

模板标签

localize

在包含的代码块内开启或关闭模板变量的本地化。

这个标签可以对本地化进行比 `USE_L10N` 更加精细的操作。

这样做来为一个模板激活或禁用本地化：

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

注意

在 `{% localize %}` 代码块内并不遵循f `USE_L10N` 的值。

对于在每个变量基础上执行相同工作的模板过滤器，参见 `localize` 和 `unlocalize`。

模板过滤器

localize

强制单一值的本地化。

例如：

```
{% load l10n %}

{{ value|localize }}
```

使用 `unlocalize` 来在单一值上禁用本地化。使用 `localize` 模板标签来在大块的模板区域内控制本地化。

unlocalize

强制单一值不带本地化输出。

例如：

```
{% load l10n %}
{{ value|unlocalize }}
```

使用 `localize` 来强制单一值的本地化。使用 `localize` 模板标签来在大块的模板区域内控制本地化。

创建自定义的格式文件

Django为许多地区提供了格式定义，但是有时你可能想要创建你自己的格式，因为你的的确并没有现成的格式文件，或者你想要覆写其中的一些值。

Changed in Django 1.8:

添加了指定 `FORMAT_MODULE_PATH` 为列表的功能。之前只支持单一的字符串值。

指定你首先放置格式文件的位置来使用自定义格式。把你的 `FORMAT_MODULE_PATH` 设置设置为格式文件存在的包名来使用它，例如：

```
FORMAT_MODULE_PATH = [
    'mysite.formats',
    'some_app.formats',
]
```

文件并不直接放在这个目录中，而是放在和地区名称相同的目录中，文件也必须名为 `formats.py` 。

需要这样一个结构来自定义英文格式：

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

其中 `formats.py` 包含自定义的格式定义。例如：

```
from __future__ import unicode_literals

THOUSAND_SEPARATOR = '\xa0'
```

使用非间断空格(Unicode `00A0`)作为千位分隔符，来代替英语中默认的逗号。

提供本地化格式的限制

一些地区对数字使用上下文敏感的格式，Django的本地化系统不能自动处理它。

瑞士(德语)

瑞士的数字格式化取决于被格式化的数字类型。对于货币值，使用逗号作为千位分隔符，以及使用小数点作为十进制分隔符。对于其它数字，逗号用于十进制分隔符，空格用于千位分隔符。Django提供的本地格式使用通用的分隔符，即逗号用于十进制分隔符，空格用于千位分隔符。

> 译者：Django 文档协作翻译小组，原文：ocalized Web UI formatting and form input。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

"本地特色"附加功能

由于历史因素，Django自带了 `django.contrib.localflavor` -- 各种各样的代码片段，有助于在特定的国家地区或文化中使用。为了便于维护以及减少Django代码库的体积，这些代码现在在Django之外单独发布。

详见官方文档：

> https://django-localflavor.readthedocs.org/

这些代码托管在GIthub上面，https://github.com/django/django-localflavor。

如何迁移

如果你使用了老版本的 `django.contrib.localflavor` 包，或者 `django-localflavor-*` 的模板之一，执行这两个简单的步骤就可以更新你的代码：

- 在PyPI中安装第三方的 `django-localflavor` 包。
- 修改你应用的导入语句来引用新的包。

  例如，将：

  ```
  from django.contrib.localflavor.fr.forms import FRPhoneNumberField
  ```

  ...改为：

  ```
  from localflavor.fr.forms import FRPhoneNumberField
  ```

新的包中的代码和以前一样(它是直接从Django中复制出来的)，所以你并不用担心功能上的向后兼容问题。只需要修改导入语句。

弃用政策

在 Django 1.5中，导入 `django.contrib.localflavor` 会产生 `DeprecationWarning` 异常。也就是说你的代码还可以继续工作，但是你应该尽快修改它。

在Django 1.6中，导入 `django.contrib.localflavor` 将不会继续工作。

> 译者：Django 文档协作翻译小组，原文：“Local flavor”。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

常见的网站应用工具

Django 提供了多种工具用于开发Web应用程序

认证

Django 中的用户认证

Django从开始就带有一个用户认证系统。它处理用户账号、组、权限以及基于cookie的用户会话。本节文档解释默认的实现如何直接使用，以及如何扩展和定制它以适合你项目的需要。

概览

Django认证系统同时处理认证和授权。简单地讲，认证验证一个用户是它们声称的那个人，授权决定一个认证通过的用户允许做什么。这里的词语认证同时指代这两项任务。

认证系统包含：

- 用户
- 权限：二元（是/否）标志指示一个用户是否可以做一个特定的任务。
- 组：对多个用户运用标签和权限的一种通用的方式。
- 一个可配置的密码哈希系统
- 用于登录用户或限制内容的表单和视图
- 一个可插拔的后台系统

Django中的认证系统的目标是非常通用且不提供在web认证系统中某些常见的功能。某些常见问题的解决方法已经在第三方包中实现：

- 密码强度检查
- 登录尝试的制约
- 第三方认证（例如OAuth）

安装

认证的支持作为Django的一个contrib模块，打包于 `django.contrib.auth` 中。默认情况下，要求的配置已经包含在 `django-admin startproject` 生成的 `settings.py` 中，它们的组成包括 `INSTALLED_APPS` 设置中的两个选项：

1. '`django.contrib.auth`'包含认证框架的核心和默认的模型。
2. '`django.contrib.contenttypes`'是Django内容类型系统，它允许权限与你创建的模型关联。 和 `MIDDLEWARE_CLASSES` 设置中的两个选项：
3. `SessionMiddleware` 管理请求之间的会话。
4. `AuthenticationMiddleware` 使用会话将用户与请求管理起来。

有了这些设置，运行 `manage.py migrate` 命令将为认证相关的模型创建必要的数据库表并为你的应用中定义的任意模型创建权限。

使用

使用Django默认的实现

- 使用User对象
- 权限和授权
- Web 请求中的认证
- 在admin 中管理用户

默认实现的API参考

自定义Users和认证

Django中的密码管理

> 译者：Django 文档协作翻译小组，原文：Overview。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

使用Django认证系统

这篇文档解释默认配置下Django认证系统的使用。这些配置已经逐步可以满足大部分常见项目对的需要，可以处理范围非常广泛的任务，且具有一套细致的密码和权限实现。对于需要与默认配置不同需求的项目，Django支持扩展和自定义认证。

Django的认证同时提供认证和授权，并通常统一称为认证系统，因为这些功能某些地方是耦合的。

User对象

`User` 对象是认证系统的核心。它们通常表示与你的站点进行交互的用户，并用于启用限制访问、注册用户信息和关联内容给创建者等。在Django的认证框架中只存在一种类型的用户，因此诸如 `'superusers'` 或管理员 `'staff'` 用户只是具有特殊属性集的user对象，而不是不同类型的user对象。

默认user的基本属性有：

- username
- password
- email
- first_name
- last_name

完整的参考请参阅 `full API documentation`，该文档更偏重特定的任务。

创建users

创建users最直接的方法是使用 `create_user()` 辅助函数：

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')

# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.last_name = 'Lennon'
>>> user.save()
```

如果你已经安装了Django admin，你也可以间接地创建*users*.

创建superusers

使用 `createsuperuser` 命令创建superusers：

```
$ python manage.py createsuperuser --username=joe --email=joe@example.com
```

将会提示你输入一个密码。在你输入一个密码后，该user将会立即创建。如果不带 `--username` 和 `--email` 选项，将会提示你输入这些值。

修改密码

Django不会在user模型上存储原始的（明文）密码，而只是一个哈希（完整的细节参见文档：密码是如何管理的）。因为这个原因，不要尝试直接操作user的password属性。这也是为什么创建一个user时要使用辅助函数。

若要修改一个用户的密码，你有几种选择：

`manage.py changepassword *username*` 提供一种从命令行修改User密码的方法。它提示你修改一个给定user的密码，你必须输入两次。如果它们匹配，新的密码将会立即修改。如果你没有提供user，命令行将尝试修改与当前系统用户匹配的用户名的密码。

你也可以通过程序修改密码，使用 `set_password()` ：

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

如果你安装了Django admin，你还可以在认证系统的*admin*页面修改user的密码。

Django还提供*views*和*forms*用于允许user修改他们自己密码。

New in Django 1.7.

如果启用了 `SessionAuthenticationMiddleware` ，修改user的密码将会登出他们所有的会话。详细信息请参阅密码修改后会话失效。

认证Users

`authenticate` (***credentials*)[source]

认证一个给定用户名和密码，请使用 `authenticate()` 。它以关键字参数形式接收凭证，对于默认的配置它是 `username` 和 `password`，如果密码对于给定的用户名有效它将返回一个 `User` 对象。如果密码无效， `authenticate()` 返回 `None` 。例子：

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # the password verified for the user
    if user.is_active:
        print()
    else:
        print()
else:
    # the authentication system was unable to verify the username and password
    print()
```

注

这是认证一系列凭证的低级的方法；例如，它被 `RemoteUserMiddleware` 使用。除非你正在编写你自己的认证系统，你可能不会使用到它。当然如果你在寻找一种登录user的方法，请参见 `login_required()` 装饰器。

权限和授权

Django从开始就带有一个简单的权限系统。它提供一种分配权限给特定的用户和用户组的方法。

它被Django的admin站点使用，但欢迎你在你自己的代码中使用。

Django admin 站点使用如下的权限：

- 查看"add"表单并添加一个只限具有该类型对象的“add”权限的用户对象。
- 查看修改列表、查看“change”表单以及修改一个只限具有该类型对象的“change”权限的用户对象。
- 删除一个只限具有该类型对象的“delete”权限的用户对象。

权限不但可以根据每个对象的类型，而且可以根据特定的对象实例设置。通过使用 `ModelAdmin` 类提供的 `has_add_permission()` 、 `has_change_permission()` 和 `has_delete_permission()` 方法，可以针对相同类型的不同对象实例自定义权限。

`User` 对象具有两个多对多的字段： `groups` 和 `user_permissions` 。 `User` 对象可以用和其它 *Django* 模型一样的方式访问它们相关的对象：

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

默认的权限

当 `django.contrib.auth` 在你的 `INSTALLED_APPS` 设置中列出时，它将确保为你安装的应用中的每个Django模型创建3个默认的权限 – add、change和delete。

这些权限将在你运行 `manage.py migrate` 时创建；在添加 `django.contrib.auth` 到 `INSTALLED_APPS` 中之后，当你第一次运行 `migrate` 时，将会为之前安装的模型创建默认的权限，包括与其同时正在安装的新的模型。之后，每当你运行 `manage.py migrate` 时，它都将为新的模型创建默认的权限。

假设你有个应用的 `app_label` 是 `foo` 和一个名为 `Bar` 的模型，要测试基本的权限，你应该使用：

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`

很少直接访问 `Permission` 模型。

组

`django.contrib.auth.models.Group` 模型是用户分类的一种通用的方式，通过这种方式你可以应用权限或其它标签到这些用户。一个用户可以属于任意多个组。

组中某个用户自动具有赋给那个组的权限。例如，如果组 `Site editors` 具有权限 `can_edit_home_page`，那么该组中的任何用户都具有该权限。

出权限之外，组还是给用户分类的一种方便的方法以给他们某些标签或扩展的功能。例如，你可以创建一个组 `'Special users'`，然后你可以这样写代码，给他们访问你的站点仅限会员的部分，或者给他们发仅限于会员的邮件。

用程序创建权限

虽然*custom permissions*可以定义在 `Meta` 类中，你还可以直接创建权限。例如，你可以为 `myapp` 中的 `BlogPost` 创建 `can_publish` 权限：

```
from myapp.models import BlogPost
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BlogPost)
permission = Permission.objects.create(codename='can_publish',
                                       name='Can Publish Posts',
                                       content_type=content_type)
```

然后该权限可以通过 `user_permissions` 属性分配给一个 `User`，或者通过 `permissions` 属性分配给 `Group`。

权限的缓存

`ModelBackend` 在第一次需要访问 `User` 对象来检查权限时会缓存它们的权限。这对于请求-响应循环还是比较好的，因为在权限添加进来之后并不会立即检查（例如在admin中）。如果你正在添加权限并需要立即检查它们，例如在一个测试或视图中，最简单的解决办法是从数据库中重新获取 `User`。 例如：

```
from django.contrib.auth.models import Permission, User
from django.shortcuts import get_object_or_404

def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # any permission check will cache the current set of permissions
    user.has_perm('myapp.change_bar')

    permission = Permission.objects.get(codename='change_bar')
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm('myapp.change_bar')  # False

    # Request new instance of User
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm('myapp.change_bar')  # True

    ...
```

Web请求中的认证

Django使用会话和中间件来拦截 `request 对象` 到认证系统中。

它们在每个请求上提供一个 `request.user` 属性，表示当前的用户。如果当前的用户没有登入，该属性将设置成 `AnonymousUser` 的一个实例，否则它将是 `User` 的实例。

你可以通过 `is_authenticated()` 区分它们，像这样：

```
if request.user.is_authenticated():
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

如何登入一个用户

如果你有一个认证了的用户，你想把它附带到当前的会话中 - 这可以通过 `login()` 函数完成。

`login` ()[source]

从视图中登入一个用户，请使用 `login()`。它接受一个 `HttpRequest` 对象和一个 `User` 对象。 `login()` 使用Django的会话框架保存用户的ID在会话中。

注意任何在匿名会话中设置的数据都会在用户登入后的会话中都会记住。

这个例子展示你可能如何使用 `authenticate()` 和 `login()`：

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirect to a success page.
        else:
            # Return a 'disabled account' error message
            ...
    else:
        # Return an 'invalid login' error message.
        ...
```

先调用 `authenticate()`：

当你是手工登入一个用户时，你必须在调用 `login()` 之前通过 `authenticate()` 成功地认证该用户。 `authenticate()` 在 `User` 上设置一个属性标识哪种认证后台成功认证了该用户（细节参见后台的文档），且该信息在后面登录的过程中是需要的。如果你视图登入一个直接从数据库中取出的用户，将会抛出一个错误。

如何登出一个用户

`logout`()[source]

若要登出一个已经通过 `django.contrib.auth.login()` 登入的用户，可以在你的视图中使用 `django.contrib.auth.logout()`。 它接收一个 `HttpRequest` 对象且没有返回值。例如：

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

注意，即使用户没有登入 `logout()` 也不会抛出任何错误。

当你调用 `logout()` 时，当前请求的会话数据将被完全清除。所有存在的数据都将清除。这是为了防止另外一个人使用相同的Web浏览器登入并访问前一个用户的会话数据。如果你想在用户登出之后>可以立即访问放入会话中的数据，请在调用 `django.contrib.auth.logout()` 之后放入。

限制访问给登陆后的用户

原始的方法

限制页面访问的简单、原始的方法是检查 `request.user.is_authenticated()` 并重定向到一个登陆页面：

```
from django.conf import settings
from django.shortcuts import redirect

def my_view(request):
    if not request.user.is_authenticated():
        return redirect('%s?next=%s' % (settings.LOGIN_URL, request.path))
    # ...
```

...或者显示一个错误信息：

```
from django.shortcuts import render

def my_view(request):
    if not request.user.is_authenticated():
        return render(request, 'myapp/login_error.html')
    # ...
```

login_required 装饰器

`login_required` ([*redirect_field_name=REDIRECT_FIELD_NAME*, *login_url=None*])[source]

作为一个快捷方式，你可以使用便捷的 `login_required()` 装饰器：

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

`login_required()` 完成下面的事情：

- 如果用户没有登入，则重定向到settings.LOGIN_URL，并传递当前查询字符串中的绝对路径。例如：`/accounts/login/?next=/polls/3/`。
- 如果用户已经登入，则正常执行视图。视图的代码可以安全地假设用户已经登入。

默认情况下，在成功认证后用户应该被重定向的路径存储在查询字符串的一个叫做 `)带有一个可选的` redirect_field_name`参数：

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

注意，如果你提供一个值给 `redirect_field_name`，你非常可能同时需要自定义你的登录模板，因为存储重定向路径的模板上下文变量将使用 `redirect_field_name` 值作为它的键，而不是默认的 `"next"`。

`login_required()` 还带有一个可选的 `login_url` 参数。例如：

```
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

注意，如果你没有指定 `login_url` 参数，你需要确保 `settings.LOGIN_URL` 并且你的登录视图正确关联。例如，使用默认值，可以添加下面几行到你的URLconf中：

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

url(r'^accounts/login/$', auth_views.login),
```

`settings.LOGIN_URL` 同时还接收视图函数名和命名的*URL*模式。这允许你自由地重新映射你的URLconf中的登录视图而不用更新设置。

注

login_required装饰器不检查user的is_active标志位。

给已验证登录的用户添加访问限制

基于特定的权限和其他方式来限制访问，你最好按照前面所叙述的那样操做。

简单的方法就是在视图中直接运行你对 `request.user` 的测试。例如，视图检查用户的邮件属于特定的地址（例如@example.com），若不是，则重定向到登录页面。

```
from django.shortcuts import redirect

def my_view(request):
    if not request.user.email.endswith('@example.com'):
        return redirect('/login/?next=%s' % request.path)
    # ...
```

`user_passes_test` (*func*[, *login_url=None*, *redirect_field_name=REDIRECT_FIELD_NAME*]) [source]

你可以用方便的 `user_passes_test` 装饰器，当回掉函数返回 `False` 时会执行一个重定向操作：

```
from django.contrib.auth.decorators import user_passes_test

def email_check(user):
    return user.email.endswith('@example.com')

@user_passes_test(email_check)
def my_view(request):
    ...
```

`user_passes_test()` 要求一个以 `User` 对象为参数的回掉函数，若用户允许访问此视图，返回 `True`。注意，`user_passes_test()` 不会自动检查 `User` 是否是不是匿名对象。

`user_passes_test()` 接收两个额外的参数：

`login_url`

让你指定那些没有通过检查的用户要重定向至哪里。若不指定其值，它可能是默认的 `settings.LOGIN_URL`。

`redirect_field_name`

与 `login_required()` 的参数相同。把它设置为 `None` 来把它从 URL 中移除，当你想把通不过检查的用户重定向到没有next page 的非登录页面时。

例如：

```
@user_passes_test(email_check, login_url='/login/')
def my_view(request):
    ...
```

permission_required 装饰器

`permission_required` (*perm*[, *login_url=None*, *raise_exception=False*])[source]

检查一个用户是否有指定的权限是相对常见的需求。因此，Django 提供了一个快捷方式：`permission_required()` 装饰器：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

`has_perm()` 方法, 权限名称采用如下方法 `"<app label>.<permission codename>"` (例如 `polls.can_vote` 表示在 `polls` 应用下一个模块的权限。

要注意 `permission_required()` 也接受一个可选的 `login_url` 参数。例如：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url='/loginpage/')
def my_view(request):
    ...
```

在 `login_required()` 装饰器中，`login_url` 默认为 `settings.` LOGIN_URL。

如果提供了 `raise_exception` 参数，装饰器抛出 `PermissionDenied` 异常，使用 *the 403 (HTTP Forbidden) 视图* 而不是重定向到登录页面。

Changed in Django 1.7:

`permission_required()` 装饰器既可以接收一个权限序列也可以接收一个单个的权限。

对普通的视图使用权限

若要对一个基于类的普通视图使用权限，可以在该类上装饰 `View.dispatch` 方法。详细细节参见*Decorating the class*。另外一个方法是编写一个封装*as_view()*的*mixin*。

密码更改后的会话失效

New in Django 1.7.

警告

这种保护只在 `MIDDLEWARE_CLASSES` 中 `SessionAuthenticationMiddleware` 开启的情况下应用。如果 `settings.py` 由Django ≥ 1.7. 的 `startproject` 生成，它会被包含进来。

在Django 2.0中，会话验证会变成强制性的， 无论是否开启了 `SessionAuthenticationMiddleware` 。 如果你拥有一个1.7之前的项目，或者使用不包含 `SessionAuthenticationMiddleware` 的模板生成的项目，考虑在阅读下面的升级说明之后开启它。

如果你的 `AUTH_USER_MODEL` 继承自 `AbstractBaseUser` ，或者实现了它自己的 `get_session_auth_hash()` 方法，验证后的会话会包含这个函数返回的哈希值。在 `AbstractBaseUser` 的情况中，这是密码字段的HMAC。如果开启了 `SessionAuthenticationMiddleware` ，Django会验证每个请求带有的哈希值是否匹配服务端计算出来的哈希值。这允许用户通过修改密码来登出所有的会话。

Django中包含的默认的密码修改视图，以及 `django.contrib.auth` 中的 `django.contrib.auth.views.password_change()` 和 `user_change_password` 视图 ，会使用新的密码哈希值升级会话，以便用户在修改密码是不会登出。如果你拥有自定义的密码修改视图，并且希望具有相似的行为，使用这个函数：

`update_session_auth_hash` (*request*, *user*)

这个函数接受当前请求，并且会在会话哈希值得到的地方升级用户对象，也会适当地升级会话哈希值。使用示例：

```
from django.contrib.auth import update_session_auth_hash

def password_change(request):
    if request.method == 'POST':
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

如果你在升级一个现存的站点，并且希望开启这一中间件，而不希望你的所有用户之后重新登录，你可以首先升级到DJango1.7并且运行它一段时间，以便所有会话在用户登录时自然被创建，它们包含上面描述的会话哈希。一旦你使用 `SessionAuthenticationMiddleware` 开始运行你的站点，任何没有登录并且会话使用验证哈希值升级过的用户的现有会话都会失效，并且需要重新登录。

注意

虽然 `get_session_auth_hash()` 给予 `SECRET_KEY`，使用新的私钥升级你的站点会使所有现有会话失效。

认证的视图

Django提供一些视图，你可以用来处理登录、登出和密码管理。它们使用*stock auth 表单*，但你也可以传递你自己的表单。

Django没有为认证视图提供默认的模板。你应该为你想要使用的视图创建自己的模板。模板的上下文定义在每个视图中，参见所有的认证视图.

使用视图

有几种不同的方法在你的项目中使用这些视图。最简单的方法是包含 `django.contrib.auth.urls` 中提供的URLconf到你自己的URLconf中，例如

```
urlpatterns = [
    url('^', include('django.contrib.auth.urls'))
]
```

这将包含进下面的URL模式：

```
^login/$ [name='login']
^logout/$ [name='logout']
^password_change/$ [name='password_change']
^password_change/done/$ [name='password_change_done']
^password_reset/$ [name='password_reset']
^password_reset/done/$ [name='password_reset_done']
^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$ [name
^reset/done/$ [name='password_reset_complete']
```

这些视图提供了一个简单易记的URL名称。使用命名URL模式的细节请参见*URL文档*。

如果你想更多地控制你的URL，你可以在你的URLconf中引用一个特定的视图：

```
urlpatterns = [
    url('^change-password/', 'django.contrib.auth.views.password_change')
]
```

这些视图具有可选的参数，你可以用来改变视图的行为。例如，如果你想修改一个视图使用的模板名称，你可以提供 `template_name` 参数。实现它的一种方法是在URLconf中提供一个关键字参数，它们将被传递到视图中。例如：

```
urlpatterns = [
    url(
        '^change-password/',
        'django.contrib.auth.views.password_change',
        {'template_name': 'change-password.html'}
    )
]
```

所有的视图都返回一个 `TemplateResponse` 实例，这允许你在渲染之前很容易自定义响应。实现它的一种方法是在你自己的视图中包装一个视图：

```
from django.contrib.auth import views

def change_password(request):
    template_response = views.password_change(request)
    # Do something with `template_response`
    return template_response
```

更多的细节，参见*TemplateResponse*文档。

所有的认证视图

下面列出了 `django.contrib.auth` 提供的所有视图。实现细节参见使用视图。

`login`(*request*[, *template_name*, *redirect_field_name*, *authentication_form*, *current_app*, *extra_context*])[source]

URL 名称： `login`

关于使用命名URL模式的细节参见*URL 文档*。

可选的参数：

- `template_name`：用于用户登录视图的模板名。默认为 `registration/login.html`。
- `redirect_field_name`：`GET` 字段的名称，包含登陆后重定向URL。默认为 `next`。
- `authentication_form`：用于认证的可调用对象（通常只是一个表单类）。默认为AuthenticationForm。
- `current_app`：指示包含当前视图的是哪个应用。更多信息参见命名URL的解析策略。
- `extra_context`：一个上下文数据的字典，将被添加到传递给模板的默认上下文数据中。

下面是 `django.contrib.auth.views.login` 所做的事情：

- 如果通过 `GET` 调用，它显示一个POST给相同URL的登录表单。后面有更多这方面的信息。
- 如果通过 `POST` 调用并带有用户提交的凭证，它会尝试登入该用户。如果登入成功，该视

图重定向到 `next` 中指定的URL。如果 `next` 没有提供，它重定向到settings.LOGIN_REDIRECT_URL（默认为 `/accounts/profile/`）。如果登入不成功，则重新显示登录表单。

你需要提供html模板给login，默认调用 `registration/login.html` 。模板会得到4个模板上下文变量：

- `form`：一个表示AuthenticationForm的Form对象。
- `next`：登入成功之后重定向的URL。它还可能包含一个查询字符串。
- `site`：如果你没有安装site框架，这将被设置成RequestSite的一个实例，它从当前的HttpRequest获得site名称和域名。
- `site_name`：`site.name` 的别名。如果你没有安装site框架，这将被设置成request.META['SERVER_NAME']的值。关于site 的更多信息，参见*"sites" 框架*。

如果你不喜欢调用 `registration/login.html` ，你可以通过额外的参数传递 `template_name` 参数给你URLconf中的视图。例如，下面URLconf中的行将使用 `myapp/login.html` ：

```
url(r'^accounts/login/$', auth_views.login, {'template_name': 'myapp/login.html'}),
```

通过传递 `redirect_field_name` 给视图，你还可以指定 `GET` 字段的值，它包含登入成功后的重定向的URL。默认情况下，该字段叫做 `next` 。

下面是一个 `registration/login.html` 模板的示例，你可以用它来作为起点。它假设你有一个定义了 `content` 块的 `base.html` 模板：

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

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

如果你自定义认证（参见*Customizing Authentication*），你可以通过 `authentication_form` 参数传递一个自定义的认证表单给登录视图。该表单必须在它的 `__init__` 方法中接收一个 `request` 关键字参数，并提供一个 `get_user` 方法，此方法返回认证过的用户对象（这个方法永远只在表单验证成功后调用）。

`logout`(*request*[, *next_page*, *template_name*, *redirect_field_name*, *current_app*, *extra_context*])[source]

登出一个用户。

URL名称： `logout`

可选的参数：

- `next_page`：登出之后要重定向的URL。
- `template_name`：用户登出之后，要展示的模板的完整名称。如果不提供任何参数，默认为 `registration/logged_out.html`。
- `redirect_field_name`：包含登出之后所重定向的URL的 `GET` 字段的名称。默认为 `next`。如果提供了 `GET` 参数，会覆盖 `next_page` URL。
- `current_app`：一个提示，表明哪个应用含有了当前视图。详见 命名空间下的URL解析策略。
- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

模板上下文：

- `title`：本地化的字符串“登出”。
- `site`：根据SITE_ID 设置的当前站点。如果你并没有安装站点框架，会设置为 RequestSite的示例，它从当前HttpRequest来获取站点名称和域名。
- `site_name`：`site.name` 的别名。如果没有安装站点框架，会设置为request.META['SERVER_NAME']。站点的更多信息请见“站点”框架。
- `current_app`：一个提示，表明哪个应用含有了当前视图。详见 命名空间下的URL解析策略。
- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

`logout_then_login`(*request*[, *login_url*, *current_app*, *extra_context*])[source]

登出一个用户，然后重定向到登录页面。

URL 名称： 没有提供默认的URL

可选的参数：

- `login_url`：登录页面要重定向的URL。如果没有提供，默认为settings.LOGIN_URL。
- `current_app`：一个提示，表明哪个应用含有了当前视图。详见 命名空间下的URL解析策

略。

- `extra_context`：一个上下文数据的字典，会被添加到向模板传递的默认的上下文数据中。

`password_change` (*request*[, *template_name*, *post_change_redirect*, *password_change_form*, *current_app*, *extra_context*])[source]

允许一个用户修改他的密码。

URL 名称：`password_change`

可选的参数：

- `template_name`：用来显示修改密码表单的template的全名。如果没有提供，默认为 `registration/password_change_form.html`。
- `post_change_redirect`：密码修改成功后重定向的URL。
- `password_change_form`：一个自定义的“修改密码”表单，必须接受 `user` 关键词参数。表单用于实际修改用户密码。默认为 PasswordChangeForm。
- `current_app`：一个提示，暗示哪个应用包含当前的视图。详见 命名空间下的URL解析策略。
- `extra_context`：上下文数据的字典，会添加到传递给模板的默认的上下文数据中。

模板上下文：

- `form`：密码修改表单（请见上面的 `password_change_form`）。

`password_change_done` (*request*[, *template_name*, *current_app*, *extra_context*])[source]

这个页面在用户修改密码之后显示。

URL 名称：`password_change_done`

可选参数：

- `template_name`：所使用模板的完整名称。如果没有提供，默认为 `registration/password_change_done.html`。
- `current_app`：一个提示，暗示哪个应用包含当前的视图。详见 命名空间下的URL解析策略。
- `extra_context`：上下文数据的字典，会添加到传递给模板的默认的上下文数据中。

`password_reset` (*request*[, *is_admin_site*, *template_name*, *email_template_name*, *password_reset_form*, *token_generator*, *post_reset_redirect*, *from_email*, *current_app*, *extra_context*, *html_email_template_name*])[source]

允许用户通过生成一次性的连接并发送到用户注册的邮箱地址中来重置密码。

如果提供的邮箱地址不在系统中存在，这个视图不会发送任何邮件，但是用户也不会收到任何错误信息。这会阻止数据泄露给潜在的攻击者。如果你打算在这种情况提供错误信息，你可以继承 `PasswordResetForm`，并使用 `password_reset_form` 参数。

用无效密码标记的用户（参见 `set_unusable_password()`）不允许请求重置密码，为了防止使用类似于LDAP的外部验证资源时的滥用。注意它们不会收到任何错误信息，因为这会暴露它们的账户，也不会发送任何邮件。

URL 名称： `password_reset`

可选参数：

- `template_name`: The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.
- `email_template_name`: The full name of a template to use for generating the email with the reset password link. Defaults to `registration/password_reset_email.html` if not supplied.
- `subject_template_name`: The full name of a template to use for the subject of the email with the reset password link. Defaults to `registration/password_reset_subject.txt` if not supplied.
- `password_reset_form`: Form that will be used to get the email of the user to reset the password for. Defaults to {{s.379}}.
- `token_generator`: Instance of the class to check the one time link. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `post_reset_redirect`: The URL to redirect to after a successful password reset request.
- `from_email`: A valid email address. By default Django uses the DEFAULT_FROM_EMAIL.
- `current_app`: A hint indicating which application contains the current view. See the {{s.385}} for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.
- `html_email_template_name`: The full name of a template to use for generating a `text/html` multipart email with the password reset link. By default, HTML email is not sent.

New in Django 1.7:

添加了 `html_email_template_name`。

Deprecated since version 1.8: `is_admin_site` 参数已被废弃，将在Django2.0中被移除。

模板上下文：

- `form`: The form (see `password_reset_form` above) for resetting the user's password.

Email模板上下文：

- `email`: An alias for `user.email`
- `user`: The current User, according to the `email` form field. Only active users are able to reset their passwords (`User.is_active is True`).
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of request.META['SERVER_NAME']. For more on sites, see *The "sites" framework*.
- `domain`: An alias for `site.domain`. If you don't have the site framework installed, this will be set to the value of `request.get_host()`.
- `protocol`: http or https
- `uid`: The user's primary key encoded in base 64.
- `token`: Token to check that the reset link is valid.

`registration/password_reset_email.html` 样例（邮件正文模板）：

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
```

主题模板使用了同样的模板上下文。主题必须是单行的纯文本字符串。

`password_reset_done` (*request*[, *template_name*, *current_app*, *extra_context*])[source]

这个页面在向用户发送重置密码的邮件后展示。如果 `password_reset()` 视图没有显式设置 `post_reset_redirect` URL，默认会调用这个视图。

URL名称： `password_reset_done`

注意

如果提供的email地址在系统中不存在，用户未激活，或者密码不可用，用户仍然会重定向到这个视图，但是不会发送邮件。

可选参数：

- `template_name`: The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.
- `current_app`: A hint indicating which application contains the current view. See the {{s.393}} for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

`password_reset_confirm` (*request*[, *uidb64*, *token*, *template_name*, *token_generator*, *set_password_form*, *post_reset_redirect*, *current_app*, *extra_context*])[source]

为输入新密码展示表单。

URL名称： `password_reset_confirm`

可选参数：

- `uidb64`: The user's id encoded in base 64. Defaults to `None`.
- `token`: Token to check that the password is valid. Defaults to `None`.
- `template_name`: The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `set_password_form`: Form that will be used to set the password. Defaults to {{s.395}}
- `post_reset_redirect`: URL to redirect after the password reset done. Defaults to `None`.
- `current_app`: A hint indicating which application contains the current view. See the {{s.400}} for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

Template context:

- `form`: The form (see `set_password_form` above) for setting the new user's password.
- `validlink`: Boolean, True if the link (combination of `uidb64` and `token`) is valid or unused yet.

`password_reset_complete` (*request*[, *template_name*, *current_app*, *extra_context*])[source]

展示一个视图，它通知用户密码修改成功。

URL名称： `password_reset_complete`

可选参数：

- `template_name`: The full name of a template to display the view. Defaults to `registration/password_reset_complete.html`.
- `current_app`: A hint indicating which application contains the current view. See the {{s.403}} for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

辅助函数

`redirect_to_login` (*next*[, *login_url*, *redirect_field_name*])[source]

重定向到登录页面，然后在登入成功后回到另一个URL。

必需的参数：

- `next`: The URL to redirect to after a successful login.

可选的参数：

- `login_url`: The URL of the login page to redirect to. Defaults to {{s.411}} if not supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next` if the given `GET` parameter is passed.

内建的表单

如果你不想用内建的视图，但是又不想编写针对该功能的表单，认证系统提供了几个内建的表单，位于 `django.contrib.auth.forms` ：

注

内建的验证表单对他们处理的用户模型做了特定假设。如果你使用了自定义的用户模型，可能需要为验证系统定义你自己的表单。更多信息请见 使用带有自定义用户模型的内建验证表单的文档。

class `AdminPasswordChangeForm` [source]

管理界面中使用的表单，用于修改用户密码。

接受 `user` 作为第一个参数。

class `AuthenticationForm` [source]

用于用户登录的表单。

接受 `request` 作为第一个参数，它储存在表单实例中，被子类使用。

`confirm_login_allowed` (*user*)[source]

New in Django 1.7.

通常， `AuthenticationForm` 会拒绝 `is_active` 标志是 `False` 的用户。你可以使用自定义政策覆盖这一行为，来决定哪些用户可以登录。使用一个继承 `AuthenticationForm` 并覆写 `confirm_login_allowed` 方法的自定义表单来实现它。如果提供的用户不能登录，这个方法应该抛出 `ValidationError` 异常。

例如，允许所有用户登录，不管“活动”状态如何：

```
from django.contrib.auth.forms import AuthenticationForm

class AuthenticationFormWithInactiveUsersOkay(AuthenticationForm):
    def confirm_login_allowed(self, user):
        pass
```

或者只允许一些活动用户登录进来：

```
class PickyAuthenticationForm(AuthenticationForm):
    def confirm_login_allowed(self, user):
        if not user.is_active:
            raise forms.ValidationError(
                _(),
                code='inactive',
            )
        if user.username.startswith('b'):
            raise forms.ValidationError(
                _(),
                code='no_b_users',
            )
```

class `PasswordChangeForm` [source]

一个表单，允许用户修改他们的密码。

class `PasswordResetForm` [source]

一个表单，用于生成和通过邮件发送一次性密码重置链接。

`send_email`(*subject_template_name*, *email_template_name*, *context*, *from_email*, *to_email*[, *html_email_template_name=None*])

New in Django 1.8.

使用参数来发送 `EmailMultiAlternatives` 。可以覆盖来自定义邮件如何发送给用户。

Parameters:	* **subject_template_name** – the template for the subject. * **email_template_name** – the template for the email body. * **context** – context passed to the `subject_template`, `email_template`, and `html_email_template` (if it is not `None`). * **from_email** – the sender's email. * **to_email** – the email of the requester. * **html_email_template_name** – the template for the HTML body; defaults to `None`, in which case a plain text email is sent.

通常， `save()` 位于 `context` 中，并带有 `password_reset()` 向它的email上下文传递的一些变量。

class `SetPasswordForm` [source]

允许用户不输入旧密码修改密码的表单。

class `UserChangeForm` [source]

用户管理界面中修改用户信息和许可的表单。

class `UserCreationForm` [source]

用于创建新用户的表单。

模板中的认证数据

当你使用 `RequestContext` 时，当前登入的用户和它们的权限在模板上下文中可以访问。

技术细节

技术上讲，这些变量只有在你使用 `RequestContext` 并启用了 `'django.contrib.auth.context_processors.auth'` 上下文处理器时才可以在模板上下文中访问到。它是默认产生的配置文件。更多信息，参见*RequestContext 文档*。

用户

当渲染 `RequestContext` 模板时，当前登录的用户，可能是 `User` 实例或者 `AnonymousUser` 实例，会存储在模板变量 `{{ user }}` 中：

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

如果使用的不是 `RequestContext`，则不可以访问该模板变量：

权限

当前登录的用户的权限存储在模板变量 `{{ perms }}` 中。这是个 `django.contrib.auth.context_processors` 实例的封装，他是一个对于模板友好的权限代理。

在 `{{ perms }}` 对象中，单一属性的查找是 `User.has_module_perms` 的代理。如果已登录的用户在 `foo` 应用中拥有任何许可，这个例子会显示 `True`：

```
{{ perms.foo }}
```

二级属性的查找是 `User.has_perm` 的代理。如果已登录的用户拥有 `foo.can_vote` 的许可，这个示例会显示 `True`：

```
{{ perms.foo.can_vote }}
```

所以，你可以用模板的 `{% if %}` 语句检查权限：

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

还可以通过 `{% if in %}` 语句查询权限。例如：

```
{% if 'foo' in perms %}
    {% if 'foo.can_vote' in perms %}
        <p>In lookup works, too.</p>
    {% endif %}
{% endif %}
```

在admin中管理用户

如果 `django.contrib.admin` 和 `django.contrib.auth` 这两个你都安装了，将可以通过admin方便地查看和管理用户、组和权限。可以像其它任何Django模型一样创建和删除用户。可以创建组，并分配权限给用户和组。admin中还会保存和显示对用户模型编辑的日志。

创建用户

在admin的主页，你应该可以在“Auth”部分看到“Users”链接。“Add user” 页面与标准admin页面不同点在于它要求你在编辑用户的其它字段之前先选择一个用户名和密码。

另请注意：如果你想使得一个用户能够使用Django的admin站点创建其它用户， 你需要给他添加用户和修改用户的权限（例如，"Add user” 和“Change user” 权限）。如果一个账号具有添加用户的权限但是没有权限修改他们，该账号将不能添加用户。为什么呢？因为如果你具有添加用户的权限，你将可以添加超级用户，这些超级用户将可以修改其他用户。所以Django同时要求添加权限和修改权限作为一种轻量的安全措施。

仔细考虑一下你是如何允许用户管理权限的。如果你了一个非超级用户编辑用户的能力，这和给他们超级用户的权限在最终效果上是一样的，因为他们将能够提升他们自己下面的用户的权限。

修改密码

用户密码不会显示在admin上（也不会存储在数据库中），但是会显示 密码存储的细节。 这个信息的显示中包含一条指向修改密码表单的链接，允许管理员修改用户的密码。

> 译者：Django 文档协作翻译小组，原文：Using the authentication system 。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

Django中的密码管理

密码管理在非必要情况下一般不会重新发明，Django致力于提供一套安全、灵活的工具集来管理用户密码。本文档描述Django存储密码和hash存储方法配置的方式，以及使用hash密码的一些实例。

另见

即使用户可能会使用强密码，攻击者也可能窃听到他们的连接。使用*HTTPS*来避免在HTTP连接上发送密码（或者任何敏感的数据），因为否则密码又被嗅探的风险。

Django如何储存密码

Django通常使用PBKDF2来提供灵活的密码储存系统。

`User` 对象的 `password` 属性是一个这种格式的字符串：

```
<algorithm>$<iterations>$<salt>$<hash>
```

那些就是用于储存用户密码的部分，以美元字符分分隔。它们由哈希算法、算法迭代次数（工作因数）、随机的salt、以及生成的密码哈希值组成。算法是Django可以使用的，单向哈希或者密码储存算法之一，请见下文。迭代描述了算法在哈希上执行的次数。salt是随机的种子值，哈希值是这个单向函数的结果。

通常，Django以SHA256的哈希值使用PBKDF2算法，由NIST推荐的一种密码伸缩机制。这对于大多数用户都很有效：它非常安全，需要大量的计算来破解。

然而，取决于你的需求，你可以选择一个不同的算法，或者甚至使用自定义的算法来满足你的特定的安全环境。不过，大多数用户并不需要这样做 -- 如果你不确定，最好不要这样。如果你打算这样做，请继续阅读：

DJango通过访问 `PASSWORD_HASHERS` 设置来选择要使用的算法。这里有一个列表，列出了Django支持的哈希算法类。列表的第一个元素 (即 `settings.` PASSWORD_HASHERS[0]) 会用于储存密码， 所有其它元素都是用于验证的哈希值，它们可以用于检查现有的密码。意思是如果你打算使用不同的算法，你需要修改 `PASSWORD_HASHERS` ，来将你最喜欢的算法在列表中放在首位。

`PASSWORD_HASHERS` 默认为：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

这意味着，Django会使用 PBKDF2 储存所有密码，但是支持使用 PBKDF2SHA1, bcrypt, SHA1等等算法来检查储存的密码。下一节会描述一些通用的方法，高级用户可能想通过它来修改这个设置。

在Django中使用bcrypt

Bcrypt是一种流行的密码储存算法，它特意被设计用于长期的密码储存。Django并没有默认使用它，由于它需要使用三方的库，但是由于很多人都想使用它，Django会以最小的努力来支持。

执行以下步骤来作为你的默认储存算法来使用Bcrypt：

1. 安装bcrypt 库。这可以通过运行 `pip install django[bcrypt]`，，或者下载并运行 `python setup.py install` 来实现。
2. 修改 `PASSWORD_HASHERS` ，将 `BCryptSHA256PasswordHasher` 放在首位。也就是说，在你的设置文件中应该：

   ```
   PASSWORD_HASHERS = (
       'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
       'django.contrib.auth.hashers.BCryptPasswordHasher',
       'django.contrib.auth.hashers.PBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
       'django.contrib.auth.hashers.SHA1PasswordHasher',
       'django.contrib.auth.hashers.MD5PasswordHasher',
       'django.contrib.auth.hashers.CryptPasswordHasher',
   )
   ```

   (你应该将其它元素留在列表中，否则Django不能升级密码；见下文)。

配置完毕 -- 现在Django会使用Bcrypt作为默认的储存算法。

BCryptPasswordHasher的密码截断

bcrypt的设计者会在72个字符处截断所有的密码，这意味着 `bcrypt(password_with_100_chars) == bcrypt(password_with_100_chars[:72])` 。原生的 `BCryptPasswordHasher` 并不会做任何的特殊处理， 所以它也会受到这一隐藏密码长度限制的约束。 `BCryptSHA256PasswordHasher` 通过事先使用 sha256生成哈希来解决这一问题。这样就可以防止密码截断了，所以你还是应该优先考虑 `BCryptPasswordHasher` 。这个截断带来的实际

效果很微不足道，因为大多数用户不会使用长度超过72的密码，并且即使在72个字符处截断，破解brypt所需的计算能力依然是天文数字。虽然如此，我们还是推荐使用 `BCryptSHA256PasswordHasher` ，根据 “有备无患”的原则。

其它 bcrypt 的实现

有一些其它的bcrypt 实现，可以让你在Django中使用它。Django的bcrypt 支持并不直接兼容这些实现。你需要修改数据库中的哈希值，改为 `bcrypt$(raw bcrypt output)` 的形式，来升级它们。例如： `bcrypt$$2a$12$NT0I31Sa7ihGEWpka9ASYrEFkhuTNeBQ2xfZskIiiJeyFXhRgS.Sy` 。

增加工作因数

PBKDF2 和bcrypt 算法使用大量的哈希迭代或循环。这会有意拖慢攻击者，使对哈希密码的攻击更难以进行。然而，随着计算机能力的不断增加，迭代的次数也需要增加。我们选了一个合理的默认值（并且在Django的每个发行版会不断增加），但是你可能想要调高或者调低它，取决于你的安全需求和计算能力。要想这样做，你可以继承相应的算法，并且覆写 `iterations` 参数。例如，增加PBKDF2算法默认使用的迭代次数：

1. 创建 `django.contrib.auth.hashers.PBKDF2PasswordHasher` 的子类：

   ```
   from django.contrib.auth.hashers import PBKDF2PasswordHasher

   class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
       """
       A subclass of PBKDF2PasswordHasher that uses 100 times more iterations.
       """
       iterations = PBKDF2PasswordHasher.iterations * 100
   ```

   把它保存在项目中的某个位置。例如，把它放在类似于 `myproject/hashers.py` 的文件中。

2. 将你的新的hasher作为第一个元素添加到 `PASSWORD_HASHERS` ：

   ```
   PASSWORD_HASHERS = (
       'myproject.hashers.MyPBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
       'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
       'django.contrib.auth.hashers.BCryptPasswordHasher',
       'django.contrib.auth.hashers.SHA1PasswordHasher',
       'django.contrib.auth.hashers.MD5PasswordHasher',
       'django.contrib.auth.hashers.CryptPasswordHasher',
   )
   ```

配置完毕 -- 现在DJango在储存使用PBKDF2的密码时会使用更多的迭代次数。

密码升级

用户登录之后，如果他们的密码没有以首选的密码算法来储存，Django会自动将算法升级为首选的那个。这意味着Django中旧的安装会在用户登录时自动变得更加安全，并且你可以随意在新的（或者更好的）储存算法发明之后切换到它们。

然而，Django只会升级在 `PASSWORD_HASHERS` 中出现的算法，所以升级到新系统时，你应该确保不要 移除列表中的元素。如果你移除了，使用列表中没有的算法的用户不会被升级。修改PBKDF2迭代次数之后，密码也会被升级。

Manually managing a user's password

`django.contrib.auth.hashers` 模块提供了一系列的函数来创建和验证哈希密码。你可以独立于 `User` 模型之外使用它们。

`check_password` (*password*, *encoded*)[source]

如果你打算通过比较纯文本密码和数据库中哈希后的密码来手动验证用户，要使用 `check_password()` 这一便捷的函数。它接收两个参数：要检查的纯文本密码，和数据库中用户的 `password` 字段的完整值。如果二者匹配，返回 `True` ，否则返回 `False` 。

`make_password` (*password*, *salt=None*, *hasher='default'*)[source]

以当前应用所使用的格式创建哈希密码。它接受一个必需参数：纯文本密码。如果你不想使用默认值（ `PASSWORD_HASHERS` 设置的首选项），你可以提供salt值和要使用的哈希算法，它们是可选的。当前支持的算法是： `'pbkdf2_sha256'` , `'pbkdf2_sha1'` , `'bcrypt_sha256'` (参见 *在 Django中使用Bcrypt*), `'bcrypt'` , `'sha1'` , `'md5'` , `'unsalted_md5'` (仅仅用于向后兼容) 和 `'crypt'` （如果你安装了 `crypt` 库）。如果password参数是 `None` ，会返回一个不可用的密码（它永远不会被 `check_password()` 接受）。

`is_password_usable` (*encoded_password*)[source]

检查提供的字符串是否是可以用 `check_password()` 验证的哈希密码。

> 译者：Django 文档协作翻译小组，原文：Password management。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

日志

日志快速入门

Django 使用Python 内建的 `logging` 模块打印日志。该模块的用法在Python 本身的文档中有详细的讨论。如果你从来没有使用过Python 的logging 框架（或者即使使用过），请参见下面的快速导论。

logging 的组成

Python 的logging 配置由四个部分组成：

- Loggers
- Handlers
- Filters
- Formatters

Loggers

Logger 为日志系统的入口。每个logger 是一个具名的容器，可以向它写入需要处理的消息。

每个logger 都有一个日志级别。日志级别表示该logger 将要处理的消息的严重性。Python 定义以下几种日志级别：

- `DEBUG`：用于调试目的的底层系统信息
- `INFO`：普通的系统信息
- `WARNING`：表示出现一个较小的问题。
- `ERROR`：表示出现一个较大的问题。
- `CRITICAL`：表示出现一个致命的问题。

写入logger 的每条消息都是一个日志记录。每个日志记录也具有一个日志级别，它表示对应的消息的严重性。每个日志记录还可以包含描述正在打印的事件的有用元信息。这些元信息可以包含很多细节，例如回溯栈或错误码。

当给一条消息给logger 时，会将消息的日志级别与logger 的日志级别进行比较。如果消息的日志级别大于等于logger 的日志级别，该消息将会往下继续处理。如果小于，该消息将被忽略。

Logger 一旦决定消息需要处理，它将传递该消息给一个*Handler*。

Handlers

Handler 决定如何处理logger 中的每条消息。它表示一个特定的日志行为，例如将消息写到屏幕上、写到文件中或者写到网络socket。

与logger 一样，handler 也有一个日志级别。如果消息的日志级别小于handler 的级别，handler 将忽略该消息。

Logger 可以有多个handler，而每个handler 可以有不同的日志级别。利用这种方式，可以根据消息的重要性提供不同形式的处理。例如，你可以用一个handler 将 `ERROR` 和 `CRITICAL` 消息发送给一个页面服务，而用另外一个hander 将所有的消息（包括 `ERROR` 和 `CRITICAL` 消息）记录到一个文件中用于以后进行分析。

Filters

Filter 用于对从logger 传递给handler 的日志记录进行额外的控制。

默认情况下，满足日志级别的任何消息都将被处理。通过安装一个filter，你可以对日志处理添加额外的条件。例如，你可以安装一个filter，只允许处理来自特定源的 `ERROR` 消息。

Filters 还可以用于修改将要处理的日志记录的优先级。例如，如果日志记录满足特定的条件，你可以编写一个filter 将日志记录从 `ERROR` 降为 `WARNING` 。

Filters 可以安装在logger 上或者handler 上；多个filter 可以串联起来实现多层filter 行为。

Formatters

最后，日志记录需要转换成文本。Formatter 表示文本的格式。Fomatter 通常由包含日志记录属性的Python 格式字符串组成；你也可以编写自定义的fomatter 来实现自己的格式。

使用logging

配置好logger、handler、filter 和formatter 之后，你需要在代码中放入logging 调用。使用 logging 框架非常简单。下面是个例子：

```
# import the logging library
import logging

# Get an instance of a logger
logger = logging.getLogger(__name__)

def my_view(request, arg1, arg):
    ...
    if bad_mojo:
        # Log an error message
        logger.error('Something went wrong!')
```

就是这样！每次满足 `bad_mojo` 条件，将写入一条错误日志记录。

命名logger

`logging.getLogger()` 调用获取（如有必要则创建）一个logger 的实例。Logger 实例通过名字标识。Logger 使用名称的目的是用于标识其配置。

Logger 的名称习惯上通常使用 `__name__`，即包含该logger 的Python 模块的名字。这允许你基于模块filter 和handle 日志调用。如果你想使用其它方式组织日志消息，可以提供点号分隔的名称来标识你的logger：

```
# Get an instance of a specific named logger
logger = logging.getLogger('project.interesting.stuff')
```

点号分隔的logger 名称定义一个层级。`project.interesting` logger 被认为是 `project.interesting.stuff` logger 的上一级；`project` logger 是 `project.interesting` logger 的上一级。

层级为何如此重要？因为可以设置logger 传播它们的logging 调用给它们的上一级。利用这种方式，你可以在根logger 上定义一系列的handler，并捕获子logger 中的所有logging 调用。在 `project` 命名空间中定义的handler 将捕获 `project.interesting` 和 `project.interesting.stuff` logger 上的所有日志消息。

这种传播行为可以基于每个logger 进行控制。如果你不想让某个logger 传播消息给它的上一级，你可以关闭这个行为。

logging 调用

Logger 实例为每个默认的日志级别提供一个入口方法：

- `logger.debug()`
- `logger.info()`
- `logger.warning()`
- `logger.error()`
- `logger.critical()`

还有另外两个调用：

- `logger.log()`：打印消息时手工指定日志级别。
- `logger.exception()`：创建一个 `ERROR` 级别日志消息，它封装当前异常栈的帧。

配置logging

当然，只是将logging 调用放入你的代码中还是不够的。你还需要配置logger、handler、filter 和formatter 来确保日志的输出是有意义的。

Python 的logging 库提供几种配置logging 的技术，从程序接口到配置文件。默认情况下，Django 使用dictConfig 格式。

为了配置logging，你需要使用 `LOGGING` 来定义字典形式的logging 设置。这些设置描述你的logging 设置的logger、handler、filter 和formatter，以及它们的日志等级和其它属性。

默认情况下， `LOGGING` 设置与*Django 的默认logging 配置*进行合并。

如果 `LOGGING` 中的 `disable_existing_loggers` 键为 `True` （默认值），那么默认配置中的所有logger 都将禁用。Logger 的禁用与删除不同；logger 仍然存在，但是将默默丢弃任何传递给它的信息，也不会传播给上一级logger。所以，你应该非常小心使用 `'disable_existing_loggers': True` ；它可能不是你想要的。你可以设置 `disable_existing_loggers` 为 `False` ，并重新定义部分或所有的默认loggers；或者你可以设置 `LOGGING_CONFIG` 为 `None` ，并 自己处理*logging 配置*。

Logging 的配置属于Django `setup()` 函数的一部分。所以，你可以肯定在你的项目代码中logger 是永远可用的。

示例

dictConfig 格式的完整文档是logging 字典配置最好的信息源。但是为了让你尝尝，下面是几个例子。

首先，下面是一个简单的配置，它将来自*django.request* logger 的所有日志请求写入到一个本地文件：

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': '/path/to/django/debug.log',
        },
    },
    'loggers': {
        'django.request': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

如果你使用这个示例，请确保修改 `'filename'` 路径为运行Django 应用的用户有权限写入的一个位置。

其次，下面这个示例演示如何让日志系统将Django 的日志打印到控制台。 `django.request` 和 `django.security` 不会传播日志给上一级。它在本地开发期间可能有用。

默认情况下，这个配置只会将 `INFO` 和更高级别的日志发送到控制台。Django 中这样的日志信息不多。可以设置环境变量 `DJANGO_LOG_LEVEL=DEBUG` 来看看Django 的debug 日志，它包含所有的数据库查询所以非常详尽。

```
import os

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
        },
    },
}
```

最后，下面是相当复杂的一个logging 设置：

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(mess
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'logging.NullHandler',
        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers': ['null'],
            'propagate': True,
            'level': 'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

这个logging 配置完成以下事情：

- 以‘dictConfig version 1’格式解析配置。目前为止，这是dictConfig 格式唯一的版本。
- 定义两个formatter：
    - `simple`，它只输出日志的级别（例如，`DEBUG`）和日志消息。

      `format` 字符串是一个普通的Python 格式化字符串，描述每行日志的细节。输出的完整细节可以在formatter 文档中找到。

    - `verbose`，它输出日志级别、日志消息，以及时间、进程、线程和生成日志消息的模块。

- 定义filter —— `project.logging.SpecialFilter`，并使用别名 `special`。如果filter 在构造时要求额外的参数，可以在filter 的配置字段中用额外的键提供。在这个例子中，在实例化 `SpecialFilter` 时，`foo` 参数的值将使用 `bar`。
- 定义三个handler：
    - `null`，一个NullHandler，它传递 `DEBUG`（和更高级）的消息给 `/dev/null`。
    - `console`，一个StreamHandler，它将打印 `DEBUG`（和更高级）的消息到stderr。这个handler 使用 `simple` 输出格式。
    - `mail_admins`，一个AdminEmailHandler，它将用邮件发送 `ERROR`（和更高级）的消息到站点管理员。这个handler 使用 `special` filter。
- 配置三个logger：
    - `django`，它传递所有 `INFO` 和更高级的消息给 `null` handler。
    - `django.request`，它传递所有 `ERROR` 消息给 `mail_admins` handler。另外，标记这个logger 不 向上传播消息。这表示写入 `django.request` 的日志信息将不会被 `django` logger 处理。
    - `myproject.custom`，它传递所有 `INFO` 和更高级的消息并通过 `special` filter 的消息给两个handler —— `console` 和 `mail_admins`。这表示所有 `INFO`（和更高级）的消息将打印到控制台上；`ERROR` 和 `CRITICAL` 消息还会通过邮件发送出来。

自定义logging 配置

如果你不想使用Python 的dictConfig 格式配置logger，你可以指定你自己的配置模式。

`LOGGING_CONFIG` 设置定义一个可调用对象，将它用来配置Django 的logger。默认情况下，它指向Python 的 `logging.config.dictConfig()` 函数。但是，如果你想使用不同的配置过程，你可以使用其它只接受一个参数的可调用对象。配置logging 时，将使用 `LOGGING` 的内容作为参数的值。

禁用logging 配置

如果你完全不想配置logging（或者你想使用自己的方法手工配置logging），你可以设置 `LOGGING_CONFIG` 为 `None`。这将禁用*Django 默认logging* 的配置过程。下面的示例禁用Django 的logging 配置，然后手工配置logging：

settings.py

```
LOGGING_CONFIG = None

import logging.config
logging.config.dictConfig(...)
```

设置 `LOGGING_CONFIG` 为 `None` 只表示禁用自动配置过程，而不是禁用logging 本身。如果你禁用配置过程，Django 仍然执行logging 调用，只是调用的是默认定义的logging 行为。

Django’s logging extensions

Django 提供许多工具用于处理在网站服务器环境中独特的日志需求。

Loggers

Django 提供几个内建的logger。

django

`django` 是一个捕获所有信息的logger。消息不会直接提交给这个logger。

django.request

记录与处理请求相关的消息。5XX 响应作为 `ERROR` 消息；4XX 响应作为 `WARNING` 消息。

这个logger 的消息具有以下额外的上下文：

- `status_code` ：请求的HTTP 响应码。
- `request` ：生成日志信息的请求对象。

django.db.backends

与数据库交互的代码相关的消息。例如，HTTP请求执行应用级别的SQL 语句将以 `DEBUG` 级别记录到该logger。

这个logger 的消息具有以下额外的上下文：

- `duration` ：执行SQL 语句花费的时间。
- `sql` ：执行的SQL 语句。
- `params` ：SQL 调用中用到的参数。

由于性能原因，SQL的日志只在 `设置` 之后开启。DEBUG 设置为 `True` ，无论日志级别或者安装的处理器是什么。

这里的日志不包含框架级别的的初始化（例如， `SET TIMEZONE` ）和事务管理查询（例如， `BEGIN` 、 `COMMIT` 和 `ROLLBACK` ）。如果你希望看到所有的数据库查询，可以打开数据库中的查询日志。

django.security.*

Security logger 将收到任何出现 `SuspiciousOperation` 的消息。SuspiciousOperation 的每个子类型都有一个子logger。日志的级别取决于异常处理的位置。大部分情况是一个warning 日志，而如果 `SuspiciousOperation` 到达WSGI handler 则记录为一个error。例如，如果请求中包含的HTTP `Host` 头部与 `ALLOWED_HOSTS` 不匹配，Django 将返回400 响应，同时将记录一个error 消息到 `django.security.DisallowedHost` logger。

默认情况下只会配置 `django.security` logger，其它所有的子logger 都将传播给上一级 logger。 `django.security` logger 的配置与 `django.request` logger 相同，任何error 消息将用邮件发送给站点管理员。由于 `SuspiciousOperation` 导致400 响应的请求不会在 `django.request` logger 中记录日志，而只在 `django.security` logger 中记录日志。

若要默默丢弃某种类型的SuspiciousOperation，你可以按照下面的示例覆盖其logger：

```
'loggers': {
    'django.security.DisallowedHost': {
        'handlers': ['null'],
        'propagate': False,
    },
},
```

django.db.backends.schema

New in Django 1.7.

当迁移框架执行的SQL 查询会改变数据库的模式时，则记录这些SQL 查询。注意，它不会记录 `RunPython` 执行的查询。

Handlers

在Python logging 模块提供的handler 基础之上，Django 还提供另外一个handler。

class `AdminEmailHandler` (*include_html=False*, *email_backend=None*)[source]

这个handler 将它收到的每个日志信息用邮件发送给站点管理员。

如果日志记录包含 `request` 属性，该请求的完整细节都将包含在邮件中。

如果日志记录包含栈回溯信息，该栈回溯也将包含在邮件中。

`AdminEmailHandler` 的 `include_html` 参数用于控制邮件中是否包含HTML 附件，这个附件包含 `DEBUG` 为 `True` 时的完整网页。若要在配置中设置这个值，可以将它包含在 `django.utils.log.AdminEmailHandler` handler 的定义中，像下面这样：

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    }
},
```

注意，邮件中的HTML 包含完整的回溯栈，包括栈每个层级局部变量的名称和值以及你的 Django 设置。这些信息可能非常敏感，你也许不想通过邮件发送它们。此时可以考虑使用类似Sentry 这样的东西，回溯栈的完整信息和安全信息不会 通过邮件发送。你还可以从错误报告中显式过滤掉特定的敏感信息 —— 更多信息参见过滤错误报告。

通过设置 `AdminEmailHandler` 的 `email_backend` 参数，可以覆盖handler 使用的*email backend*，像这样：

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'email_backend': 'django.core.mail.backends.filebased.EmailBackend',
    }
},
```

默认情况下，将使用 `EMAIL_BACKEND` 中指定的邮件后端。

`send_mail`(*subject*, *message*, **args*, ***kwargs*)[source]

New in Django 1.8.

发送邮件给管理员用户。若要自定它的行为，可以子类化 `AdminEmailHandler` 类并覆盖这个方法。

Filters

在Python logging 模块提供的过滤器的基础之上，Django 还提供两个过滤器。

class `CallbackFilter`(*callback*)[source]

这个过滤器接受一个回调函数（它接受一个单一参数，也就是要记录的东西），并且对每个传递给过滤器的记录调用它。如果回调函数返回False，将不会进行记录的处理。

例如，要从admin邮件中过滤掉 `UnreadablePostError`（只在用户取消上传时产生），你可以创建一个过滤器函数：

```
from django.http import UnreadablePostError

def skip_unreadable_post(record):
    if record.exc_info:
        exc_type, exc_value = record.exc_info[:2]
        if isinstance(exc_value, UnreadablePostError):
            return False
    return True
```

然后把它添加到logger的配置中：

```
'filters': {
    'skip_unreadable_posts': {
        '()': 'django.utils.log.CallbackFilter',
        'callback': skip_unreadable_post,
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['skip_unreadable_posts'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class `RequireDebugFalse` [source]

这个过滤器只在设置后传递记录。DEBUG 为 False。

这个过滤器遵循 `LOGGING` 默认的配置，以确保 `AdminEmailHandler` 只在 `DEBUG` 为 `False` 的时候发送错误邮件。

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse',
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class `RequireDebugTrue` [source]

这个过滤器类似于 `RequireDebugFalse`，除了记录只在 `DEBUG` 为 `True` 时传递的情况。

Django’s default logging configuration

默认情况下，Django 的logging 配置如下：

当 `DEBUG` 为 `True` 时：

- `django` 的全局logger会向控制台发送级别等于或高级 `INFO` 的所有消息。Django在这个时候并不会做任何日志调用（所有在 `DEBUG` 级别上的日志，或者被 `django.request` 和 `django.security` 处理的日志）。
- `py.warnings` logger，它处理来自 `warnings.warn()` 的消息，会向控制台发送消息。

当 `DEBUG` 为 `False` 时：

- `django.request` 和 `django.security` loggers 向 `AdminEmailHandler` 发送带有 `ERROR` 或 `CRITICAL` 级别的消息。这些logger 会忽略任何级别等于或小于 `WARNING` 的信息，被记录的日志不会传递给其他logger（它们不会传递给 `django` 的全局 logger，即使 `DEBUG` 为 `True`）。

另见配置日志来了解如何补充或者替换默认的日志配置。

> 译者：Django 文档协作翻译小组，原文：Logging。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

分页

Django提供了一些类来帮助你管理分页的数据 -- 也就是说，数据被分在不同页面中，并带有“上一页/下一页”标签。这些类位于 `django/core/paginator.py` 中。

示例

向 `Paginator` 提供对象的列表，以及你想为每一页分配的元素数量，它就会为你提供访问每一页上对象的方法：

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
4
>>> p.num_pages
2
>>> p.page_range
[1, 2]

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
>>> page2.previous_page_number()
1
>>> page2.start_index() # The 1-based index of the first item on this page
3
>>> page2.end_index() # The 1-based index of the last item on this page
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

注意

注意你可以向 `Paginator` 提供一个列表或元组，Django的 `QuerySet` ，或者任何带有 `count()` 或 `__len__()` 方法的对象。当计算传入的对象所含对象的数量时， `Paginator` 会首先尝试调用 `count()` ，接着如果传入的对象没有 `count()` 方法则回退调用 `len()` 。这样会使类似于Django的 `QuerySet` 的对象使用更加高效的 `count()` 方法，如果存在的话。

使用 `Paginator`

这里有一些复杂一点的例子，它们在视图中使用 `Paginator` 来为查询集分页。我们提供视图以及相关的模板来展示如何展示这些结果。这个例子假设你拥有一个已经导入的 `Contacts` 模型。

视图函数看起来像是这样：

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

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Show 25 contacts per page

    page = request.GET.get('page')
    try:
        contacts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer, deliver first page.
        contacts = paginator.page(1)
    except EmptyPage:
        # If page is out of range (e.g. 9999), deliver last page of results.
        contacts = paginator.page(paginator.num_pages)

    return render_to_response('list.html', {"contacts": contacts})
```

在 `list.html` 模板中，你会想要包含页面之间的导航，以及来自对象本身的任何有趣的信息：

```
<div class="pagination">
    <span class="step-links">

        <span class="current">
            Page  of .
        </span>

    </span>
</div>
```

Paginator objects

`Paginator` 类拥有以下构造器：

class `Paginator` (*object_list*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)[source]

所需参数

`object_list`

A list, tuple, Django `QuerySet`, or other sliceable object with a `count()` or `__len__()` method.

`per_page`

The maximum number of items to include on a page, not including orphans (see the `orphans` optional argument below).

可选参数

`orphans`

The minimum number of items allowed on the last page, defaults to zero. Use this when you don't want to have a last page with very few items. If the last page would normally have a number of items less than or equal to `orphans`, then those items will be added to the previous page (which becomes the last page) instead of leaving the items on a page by themselves. For example, with 23 items, `per_page=10`, and `orphans=3`, there will be two pages; the first page with 10 items and the second (and last) page with 13 items.

`allow_empty_first_page`

Whether or not the first page is allowed to be empty. If `False` and `object_list` is empty, then an `EmptyPage` error will be raised.

方法

`Paginator.``page`(*number*)[source]

返回在提供的下标处的 `Page` 对象，下标以1开始。如果提供的页码不存在，抛出 `InvalidPage` 异常。

属性

`Paginator.``count`

所有页面的对象总数。

注意

当计算 `object_list` 所含对象的数量时， `Paginator` 会首先尝试调用 `object_list.count()`。如果 `object_list` 没有 `count()` 方法， `Paginator` 接着会回退使用 `len(object_list)`。这样会使类似于Django's `QuerySet` 的对象使用更加便捷的 `count()` 方法，如果存在的话。

`Paginator.``num_pages`

页面总数。

`Paginator.``page_range`

页码的范围，从1开始，例如 `[1, 2, 3, 4]` 。

InvalidPage exceptions

exception `InvalidPage` [source]

异常的基类，当paginator传入一个无效的页码时抛出。

`Paginator.page()` 放回在所请求的页面无效（比如不是一个整数）时，或者不包含任何对象时抛出异常。通常，捕获 `InvalidPage` 异常就够了，但是如果你想更加精细一些，可以捕获以下两个异常之一：

exception `PageNotAnInteger` [source]

当向 `page()` 提供一个不是整数的值时抛出。

exception `EmptyPage` [source]

当向 `page()` 提供一个有效值，但是那个页面上没有任何对象时抛出。

这两个异常都是 `InvalidPage` 的子类，所以你可以通过简单的 `except InvalidPage` 来处理它们。

Page objects

你通常不需要手动构建 `Page` 对象 -- 你可以从 `Paginator.page()` 来获得它们。

class `Page` (*object_list*, *number*, *paginator*)[source]

当调用 `len()` 或者直接迭代一个页面的时候，它的行为类似于 `Page.object_list` 的序列。

方法

`Page.``has_next` ()[source]

Returns `True` if there's a next page.

`Page.``has_previous` ()[source]

如果有上一页，返回 `True` 。

`Page.``has_other_pages` ()[source]

如果有上一页或下一页，返回 `True` 。

`Page.``next_page_number` ()[source]

返回下一页的页码。如果下一页不存在，抛出 `InvalidPage` 异常。

`Page.``previous_page_number` ()[source]

返回上一页的页码。如果上一页不存在，抛出 `InvalidPage` 异常。

`Page.``start_index` ()[source]

返回当前页上的第一个对象，相对于分页列表的所有对象的序号，从1开始。比如，将五个对象的列表分为每页两个对象，第二页的 `start_index()` 会返回 `3` 。

`Page.``end_index` ()[source]

返回当前页上的最后一个对象，相对于分页列表的所有对象的序号，从1开始。 比如，将五个对象的列表分为每页两个对象，第二页的 `end_index()` 会返回 `4` 。

属性

`Page.``object_list`

当前页上所有对象的列表。

`Page.``number`

当前页的序号，从1开始。

`Page.``paginator`

相关的 `Paginator` 对象。

> 译者：Django 文档协作翻译小组，原文：Pagination。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

如何使用会话

Django 提供对匿名会话的完全支持。其会话框架让你根据各个站点的访问者存储和访问任意数据。它在服务器端存储数据并抽象Cookie 的发送和接收。Cookie 包含会话的ID —— 不是数据本身（除非你使用基于Cookie 的后端）。

启用会话

会话是通过一个中间件实现的。

为了启用会话功能，需要这样做：

编辑 `MIDDLEWARE_CLASSES` 设置并确保它包含' `django.contrib.sessions.middleware.SessionMiddleware` '。 `django-admin startproject` 创建的默认的 `settings.py` 已经启用 `SessionMiddleware` 。 如果你不想使用会话，你也可以从 `MIDDLEWARE_CLASSES` 中删除 `SessionMiddleware` 行，并从 `INSTALLED_APPS` 中删除' `django.contrib.sessions` '。它将节省一些性能消耗。

配置会话引擎

默认情况下，Django 存储会话到你的数据库中（使用 `django.contrib.sessions.models.Session` 模型）。虽然这很方便，但是在某些架构中存储会话在其它地方会更快，所以可以配置Django 来存储会话到你的文件系统上或缓存中。

使用数据库支持的会话

如果你想使用数据库支持的会话，你需要添加' `django.contrib.sessions` ' 到你的 `INSTALLED_APPS` 设置中。

在配置完成之后，请运行 `manage.py migrate` 来安装保存会话数据的一张数据库表。

使用基于缓存的会话

为了更好的性能，你可能想使用一个基于缓存的会话后端。

为了使用Django 的缓存系统来存储会话数据，你首先需要确保你已经配置好你的缓存；详细信息参见缓存的文档。

> 警告
>
> 你应该只在使用Memcached 缓存系统时才使用基于缓存的会话。基于本地内存的缓存系统不会长时间保留数据，所以不是一个好的选择，而且直接使用文件或数据库会话比通过文件或数据库缓存系统要快。另外，基于本地内存的缓存系统不是多进程安全的，所以对于生产环境可能不是一个好的选择。

如果你在 `CACHES` 中定义多个缓存，Django 将使用默认的缓存。若要使用另外一种缓存，请设置 `SESSION_CACHE_ALIAS` 为该缓存的名字。

配置好缓存之后，对于如何在缓存中存储数据你有两个选择：

- 对于简单的缓存会话存储，可以设置 `SESSION_ENGINE` 为"`django.contrib.sessions.backends.cache`"。此时会话数据将直接存储在你的缓存中。然而，缓存数据将可能不会持久：如果缓存填满或者缓存服务器重启，缓存数据可能会被清理掉。
- 若要持久的缓存数据，可以设置 `SESSION_ENGINE` 为"`django.contrib.sessions.backends.cached_db`"。它的写操作使用缓存 —— 对缓存的每次写入都将再写入到数据库。对于读取的会话，如果数据不在缓存中，则从数据库读取。

两种会话的存储都非常快，但是简单的缓存更快，因为它放弃了持久性。大部分情况下，`cached_db` 后端已经足够快，但是如果你需要榨干最后一点的性能，并且接收让会话数据丢失，那么你可使用 `cache` 后端。

如果你使用 `cached_db` 会话后端，你还需要遵循使用数据库支持的会话中的配置说明。

```
Changed in Django 1.7:

在1.7 版之前，`cached_db` 永远使用`default`缓存而不是`SESSION_CACHE_ALIAS`。
```

使用基于文件的缓存

要使用基于文件的缓存，请设置 `SESSION_ENGINE` 为"`django.contrib.sessions.backends.file`"。

你可能还想设置 `SESSION_FILE_PATH`（它的默认值来自 `tempfile.gettempdir()` 的输出，大部分情况是 `/tmp`）来控制Django在哪里存储会话文件。请保证你的Web 服务器具有读取和写入这个位置的权限。

使用基于Cookie 的会话

要使用基于Cookie 的会话，请设置 `SESSION_ENGINE` 为"`django.contrib.sessions.backends.signed_cookies`"。此时，会话数据的存储将使用Django的加密签名 工具和 `SECRET_KEY` 设置。

> 注
>
> 建议保留SESSION_COOKIE_HTTPONLY 设置为True 以防止从JavaScript 中访问存储的数据。
>
> 警告
>
> 如果 `SECRET_KEY` 没有保密并且你正在使用 `PickleSerializer`，这可能导致远端执行任意的代码。
>
> 拥有 `SECRET_KEY` 的攻击者不仅可以生成篡改的会话数据而你的站点将会信任这些数据，而且可以远程执行任何代码，就像数据是通过pickle 序列化过的一样。
>
> 如果你使用基于Cookie 的会话，请格外注意你的安全秘钥对于任何可以远程访问的系统都是永远完全保密的。
>
> 会话数据经过签名但没有加密。
>
> 如果使用基于Cookie的会话，则会话数据可以被客户端读取。
>
> MAC（消息认证码）被用来保护数据不被客户端修改，所以被篡改的会话数据将是变成不合法的。如果保存Cookie的客户端（例如你的浏览器）不能保存所有的会话Cookie或丢失数据，会话同样会变得不合法。尽管Django 对数据进行压缩，仍然完全有可能超过每个Cookie 常见的4096 个字节的限制。
>
> 没有更新保证
>
> 还要注意，虽然MAC可以保证数据的权威性（由你的站点生成，而不是任何其他人）和完整性（包含全部的数据并且是正确的），它不能保证是最新的，例如返回给你发送给客户端的最新的数据。这意味着对于某些会话数据的使用，基于Cookie 可能让你受到重放攻击。其它方式的会话后端在服务器端保存每个会话并在用户登出时使它无效，基于Cookie 的会话在用户登出时不会失效。因此，如果一个攻击者盗取用户的Cookie，它们可以使用这个Cookie 来以这个用户登录即使用户已登出。Cookies 只能被当做是“过期的”，如果它们比你的SESSION_COOKIE_AGE要旧。
>
> 性能
>
> 最后，Cookie 的大小对你的网站的速度 有影响。

在视图中使用会话

当 `SessionMiddleware` 激活时，每个 `HttpRequest` 对象 —— 传递给Django 视图函数的第一个参数 —— 将具有一个 `session` 属性，它是一个类字典对象。

你可以在你的视图中任何地方读取并写入 `request.session` 。你可以多次编辑它。

`class backends.base.SessionBase`

这是所有会话对象的基类。它具有以下标准的字典方法：

`__getitem__(key)`

例如： `fav_color = request.session['fav_color']`

`__setitem__(key, value)`

例如： `request.session['fav_color'] = 'blue'`

`__delitem__(key)`

例如： `del request.session['fav_color']` 。如果给出的key 在会话中不存在，将抛出 `KeyError` 。

`__contains__(key)`

例如： `'fav_color' in request.session`

`get(key, default=None)`

例如： `fav_color = request.session.get('fav_color', 'red')`

`pop(key)`

例如： `fav_color = request.session.pop('fav_color')`

`keys()`

`items()`

`setdefault()`

`clear()`

它还具有这些方法：

`flush()`

删除当前的会话数据并删除会话的Cookie。这用于确保前面的会话数据不可以再次被用户的浏览器访问（例如， `django.contrib.auth.logout()` 函数中就会调用它）。

```
Changed in Django 1.8:

删除会话Cookie 是Django 1.8 中的新行为。以前，该行为用于重新生成会话中的值，这个值会在Cookie 中发回给用
```

`set_test_cookie()`

设置一个测试的Cookie 来验证用户的浏览器是否支持Cookie。因为Cookie 的工作方式，只有到用户的下一个页面才能验证。更多信息参见下文的设置测试的Cookie。

`test_cookie_worked()`

返回 `True` 或 `False`，取决于用户的浏览器时候接受测试的Cookie。因为Cookie的工作方式，你必须在前面一个单独的页面请求中调用 `set_test_cookie()`。更多信息参见下文的设置测试的Cookie。

`delete_test_cookie()`

删除测试的Cookie。使用这个函数来自己清理。

`set_expiry(value)`

设置会话的超时时间。你可以传递一系列不同的值：

- 如果 `value` 是一个整数，会话将在这么多秒没有活动后过期。例如，调用 `request.session.set_expiry(300)` 将使得会话在5分钟后过期。
- 若果value 是一个 `datetime` 或 `timedelta` 对象，会话将在这个指定的日期/时间过期。注意 `datetime` 和 `timedelta` 值只有在你使用P `ickleSerializer` 时才可序列化。
- 如果 `value` 为0，那么用户会话的Cookie将在用户的浏览器关闭时过期。
- 如果 `value` 为 `None`，那么会话转向使用全局的会话过期策略。

过期的计算不考虑读取会话的操作。会话的过期从会话上次修改的时间开始计算。

`get_expiry_age()`

返回会话离过期的秒数。对于没有自定义过期的会话（或者设置为浏览器关闭时过期的会话），它将等于 `SESSION_COOKIE_AGE`。

该函数接收两个可选的关键字参数：

- `modification`：会话的最后一次修改时间，类型为一个 `datetime` 对象。默认为当前的时间。
- `expiry`：会话的过期信息，类型为一个 `datetime` 对象、一个整数（以秒为单位）或 `None`。默认为通过 `set_expiry()` 保存在会话中的值，如果没有则为 `None`。

`get_expiry_date()`

返回过期的日期。对于没有自定义过期的会话（或者设置为浏览器关闭时过期的会话），它将等于从现在开始 `SESSION_COOKIE_AGE` 秒后的日期。

这个函数接受与 `get_expiry_age()` 一样的关键字参数。

`get_expire_at_browser_close()`

返回 `True` 或 `False`，取决于用户的会话Cookie在用户浏览器关闭时会不会过期。

`clear_expired()`

从会话的存储中清除过期的会话。这个类方法被 `clearsessions` 调用。

`cycle_key()`

创建一个新的会话，同时保留当前的会话数据。`django.contrib.auth.login()` 调用这个方法来减缓会话的固定。

会话的序列化

在1.6 版以前，在保存会话数据到后端之前Django 默认使用pickle 来序列化它们。如果你使用的是签名的Cookie 会话后端 并且 `SECRET_KEY` 被攻击者知道（Django 本身没有漏洞会导致它被泄漏），攻击者就可以在会话中插入一个字符串，在unpickle 之后可以在服务器上执行任何代码。在因特网上这个攻击技术很简单并很容易查到。尽管Cookie 会话的存储对Cookie 保存的数据进行了签名以防止篡改，`SECRET_KEY` 的泄漏会立即使得可以执行远端的代码。

这种攻击可以通过JSON而不是pickle序列化会话数据来减缓。为了帮助这个功能，Django 1.5.3 引入一个新的设置，`SESSION_SERIALIZER`，来自定义会话序列化的格式。为了向后兼容，这个设置在Django 1.5.x 中默认为 `django.contrib.sessions.serializers.PickleSerializer`，但是为了增强安全性，在Django 1.6 中默认为 `django.contrib.sessions.serializers.JSONSerializer`。即使在编写你自己的序列化方法讲述的说明中，我们也强烈建议依然使用JSON 序列化，特别是在你使用的是Cookie 后端时。

绑定的序列化方法

`class serializers.JSONSerializer`

对 `django.core.signing` 中的JSON 序列化方法的一个包装。只可以序列基本的数据类型。

另外，因为JSON 只支持字符串作为键，注意使用非字符串作为 `request.session` 的键将不工作：

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following serialization & deserialization
>>> # of session data
>>> request.session[0]  # KeyError
>>> request.session['0']
'bar'
```

参见编写你自己的序列化器 一节以获得更多关于JSON 序列化的限制。

`class serializers.PickleSerializer`

支持任意Python 对象，但是正如上面描述的，可能导致远端执行代码的漏洞，如果攻击者知道了 `SECRET_KEY`。

编写你自己的序列化器

注意，与 `PickleSerializer` 不同， `JSONSerializer` 不可以处理任意的Python 数据类型。这是常见的情况，需要在便利性和安全性之间权衡。如果你希望在JSON 格式的会话中存储更高级的数据类型比如 `datetime` 和 `Decimal`，你需要编写一个自定义的序列化器（或者在保存它们到 `request.session` 中之前转换这些值到一个可JSON 序列化的对象）。虽然序列化这些值相当简单直接 （ `django.core.serializers.json.DateTimeAwareJSONEncoder` 可能帮得上忙），编写一个解码器来可靠地取出相同的内容却能困难。例如，返回一个 `datetime` 时，它可能实际上是与 `datetime` 格式碰巧相同的一个字符串）。

你的序列化类必须实现两个方法， `dumps(self, obj)` 和 `loads(self, data)` 来分别序列化和去序列化会话数据的字典。

会话对象指南

`在request.session` 上使用普通的Python 字符串作为字典的键。这主要是为了方便而不是一条必须遵守的规则。 以一个下划线开始的会话字典的键被Django保留作为内部使用。 不要新的对象覆盖 `request.session`，且不要访问或设置它的属性。要像Python 字典一样使用它。

例子

下面这个简单的视图在一个用户提交一个评论后设置 `has_commented` 变量为 `True`。它不允许一个用户多次提交评论：

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

登录站点一个“成员”的最简单的视图：

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

...下面是登出一个成员的视图，已经上面的login()：

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

标准的 `django.contrib.auth.logout()` 函数实际上所做的内容比这个要多一点以防止意外的数据泄露。它调用的 `request.session` 的 `flush()` 方法。我们使用这个例子来演示如何利用会话对象来工作，而不是一个完整的 `logout()` 实现。

设置测试的Cookie

为了方便，Django 提供一个简单的方法来测试用户的浏览器时候接受Cookie。只需在一个视图中调用 `request.session` 的 `set_test_cookie()` 方法，并在接下来的视图中调用 `test_cookie_worked()` —— 不是在同一个视图中调用。

由于Cookie的工作方式，在 `set_test_cookie()` 和 `test_cookie_worked()` 之间这种笨拙的分离是必要的。当你设置一个Cookie，直到浏览器的下一个请求你不可能真实知道一个浏览器是否接受了它。

使用 `delete_test_cookie()` 来自己清除测试的Cookie是一个很好的实践。请在你已经验证测试的Cookie 已经工作后做这件事。

下面是一个典型的使用示例：

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

在视图外使用会话

> 注
>
> 这一节中的示例直接从 `django.contrib.sessions.backends.db` 中导入 `SessionStore` 对象。在你的代码中，你应该从 `SESSION_ENGINE` 指定的会话引擎中导入 `SessionStore`，如下所示：

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
```

在视图的外面有一个API 可以使用来操作会话的数据：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not serializable in JSON.
>>> s['last_login'] = 1376587691
>>> s.save()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'

>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login']
1376587691
```

为了减缓会话固话攻击，不存在的会话的键将重新生成：

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='no-such-session-here')
>>> s.save()
>>> s.session_key
'ff882814010ccbc3c870523934fee5a2'
```

如果你使用的是 `django.contrib.sessions.backends.db` 后端，每个会话只是一个普通的 Django 模型。 `Session` 模型定义在 `django/contrib/sessions/models.py` 中。因为它是一个普通的模型，你可以使用普通的Django 数据库API 来访问会话：

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

注意，你需要调用 `get_decoded()` 以获得会话的字典。这是必需的，因为字典是以编码后的格式保存的：

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

会话何时保存

默认情况下，Django 只有在会话被修改时才会保存会话到数据库中 —— 即它的字典中的任何值被赋值或删除时：

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

上面例子的最后一种情况，我们可以通过设置会话对象的 `modified` 属性显式地告诉会话对象它已经被修改过：

```
request.session.modified = True
```

若要修改这个默认的行为，可以设置 `SESSION_SAVE_EVERY_REQUEST` 为 `True`。当设置为 `True` 时，Django 将对每个请求保存会话到数据库中。

注意会话的Cookie 只有在一个会话被创建或修改后才会发送。如果 `SESSION_SAVE_EVERY_REQUEST` 为 `True`，会话的Cookie 将在每个请求中发送。

类似地，会话Cookie 的 `expires` 部分在每次发送会话Cookie 时更新。

如果响应的状态码时500，则会话不会被保存。

浏览器时长的会话 VS. 持久的会话

你可以通过 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置来控制会话框架使用浏览器时长的会话，还是持久的会话。

默认情况下，`SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `False`，表示会话的Cookie 保存在用户的浏览器中的时间为 `SESSION_COOKIE_AGE`。如果你不想让大家每次打开浏览器时都需要登录时可以这样使用。

如果 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `True`，Django 将使用浏览器时长的Cookie —— 用户关闭他们的浏览器时立即过期。如果你想让大家在每次打开浏览器时都需要登录时可以这样使用。

这个设置是一个全局的默认值，可以通过显式地调 `request.session` 的 `set_expiry()` 方法来覆盖，在上面的在视图中使用会话中有描述。

> 注
>
> 某些浏览器（例如Chrome）提供一种设置，允许用户在关闭并重新打开浏览器后继续使用会话。在某些情况下，这可能干扰 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置并导致会话在浏览器关闭后不会过期。在测试启用 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置的Django应用时请注意这点。

清除存储的会话

随着用户在你的网站上创建新的会话，会话数据可能会在你的会话存储仓库中积累。如果你正在使用数据库作为后端， `django_session` 数据库表将持续增长。如果你正在使用文件作为后端，你的临时目录包含的文件数量将持续增长。

要理解这个问题，考虑一下数据库后端发生的情况。当一个用户登入时，Django 添加一行到 `django_session` 数据库表中。每次会话数据更新时，Django 将更新这行。如果用户手工登出，Django 将删除这行。但是如果该用户不登出，该行将永远不会删除。以文件为后端的过程类似。

Django 不提供自动清除过期会话的功能。因此，定期地清除会话是你的任务。Django 提供一个清除用的管理命令来满足这个目的： `clearsessions` 。建议定义调用这个命令，例如作为一个每天运行的Cron 任务。

注意，以缓存为后端不存在这个问题，因为缓存会自动删除过期的数据。以cookie 为后端也不存在这个问题，因为会话数据通过用户的浏览器保存。

设置

一些Django 设置 让你可以控制会话的行为：

- SESSION_CACHE_ALIAS
- SESSION_COOKIE_AGE
- SESSION_COOKIE_DOMAIN
- SESSION_COOKIE_HTTPONLY
- SESSION_COOKIE_NAME
- SESSION_COOKIE_PATH
- SESSION_COOKIE_SECURE
- SESSION_ENGINE
- SESSION_EXPIRE_AT_BROWSER_CLOSE
- SESSION_FILE_PATH
- SESSION_SAVE_EVERY_REQUEST

会话的安全

一个站点下的子域名能够在客户端为整个域名设置Cookie。如果子域名不收信任的用户控制且允许来自子域名的Cookie，那么可能发生会话固定。

例如，一个攻击者可以登录 `good.example.com` 并为他的账号获取一个合法的会话。如果该攻击者具有 `bad.example.com` 的控制权，那么他可以使用这个域名来发送他的会话ID给你，因为子域名允许在 `*.example.com` 上设置Cookie。当你访问 `good.example.com` 时，你将被登录成攻击者而没有注意到并输入你的敏感的个人信息（例如，信用卡信息）到攻击者的账号中。

另外一个可能的攻击是，如果 `good.example.com` 设置它的 `SESSION_COOKIE_DOMAIN` 为" `.example.com` "，这将导致来自该站点的会话Cookie 被发送到 `bad.example.com` 。

技术细节

- 当使用 `JSONSerializer` 时，会话字典接收任何可json 序列化的值，当使用 `PickleSerializer` 时接收任何pickleable 的Python对象。更多信息参见 `pickle` 模块。
- 会话数据存储在数据中名为 `django_session` 的表中。
- Django 只发送它需要的Cookie。如果你没有设置任何会话数据，它将不会发送会话Cookie。

URL 中的会话ID

Django 会话框架完全地、唯一地基于Cookie。它不像PHP一样，实在没办法就把会话的ID放在URL 中。这是一个故意的设计。这个行为不仅使得URL变得丑陋，还使得你的网站易于受到通过"Referer" 头部窃取会话ID的攻击。

> 译者：Django 文档协作翻译小组，原文：Sessions。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

验证器

编写验证器

验证器是一个可调用的对象，它接受一个值，并在不符合一些规则时抛出 `ValidationError` 异常。验证器有助于在不同类型的字段之间重复使用验证逻辑。

例如，这个验证器只允许偶数：

```
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError('%s is not an even number' % value)
```

你可以通过字段的 `validators` 参数将它添加到模型字段中：

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

由于值在验证器运行之前会转化为Python，你可以在表单上使用相同的验证器：

```
from django import forms

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```

你也可以使用带有 `__call__()` 方法的类，来实现更复杂或可配置的验证器。例如，`RegexValidator` 就用了这种技巧。如果一个基于类的验证器用于 `validators` 模型字段的选项，你应该通过添加*deconstruct()* 和 `__eq__()` 方法确保它可以被迁移框架序列化。

验证器如何运行

关于验证器如何在表单中运行，详见表单验证 。关于它们如何在模型中运行，详见 验证对象。要注意验证器不会在你保存模型时自动运行，但是如果你使用 `ModelForm`，它会在任何你表单包含的字段上运行你的验证器。关于模型验证器如何和表单交互，详见*ModelForm 文档*。

内建的验证器

`django.core.validators` 模块包含了一系列的可调用验证器，用于模型和表单字段。它们在内部使用，但是也可以用在你自己的字段上。它们可以用在 `field.clean()` 方法之外，或者代替它。

RegexValidator

class `RegexValidator` ([*regex=None*, *message=None*, *code=None*, *inverse_match=None*, *flags=0*])[source]

Parameters:	* **regex** – 如果不是`None`则覆写 [`regex`] (#django.core.validators.RegexValidator.regex "django.core.validators.RegexValidator.regex")。可以是一个正则表达式字符串，或者预编译的正则表达式对象。 * **message** – 如果不是`None`，则覆写 [`message`] (#django.core.validators.RegexValidator.message "django.core.validators.RegexValidator.message")。 * **code** – 如果不是`None`，则覆写[`code`] (#django.core.validators.RegexValidator.code "django.core.validators.RegexValidator.code")。 * **inverse_match** – 如果不是`None`，则覆写[`inverse_match`] (#django.core.validators.RegexValidator.inverse_match "django.core.validators.RegexValidator.inverse_match")。 * **flags** – 如果不是`None`，则覆写 [`flags`] (#django.core.validators.RegexValidator.flags "django.core.validators.RegexValidator.flags")。这种情况下，[`regex`] (#django.core.validators.RegexValidator.regex "django.core.validators.RegexValidator.regex")，必须是正则表达式字符串，否则抛出[`TypeError`] (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。

`regex`

用于搜索提供的 `value` 的正则表达式，或者是预编译的正则表达式对象。通常在找不到匹配时抛出带有 `message` 和 `code` 的 `ValidationError` 异常。这一标准行为可以通过设置 `inverse_match` 为 `True` 来反转，这种情况下，如果找到匹配则抛出 `ValidationError` 异常。通常它会匹配任何字符串（包括空字符串）。

`message`

验证失败时 `ValidationError` 所使用的错误信息。默认为 `"Enter a valid value"`。

`code`

验证失败时 `ValidationError` 所使用的错误代码。默认为 `"invalid"`。

`inverse_match`

New in Django 1.7.

`regex` 的匹配模式。默认为 `False`。

`flags`

New in Django 1.7.

编译正则表达式字符串 `regex` 时所用的标识。如果 `regex` 是预编译的正则表达式，并且覆写了 `flags`，会产生 `TypeError` 异常。默认为 *0*。

EmailValidator

class `EmailValidator` ([*message=None*, *code=None*, *whitelist=None*])[source]

Parameters:	* **message** – 如果不是 `None`，则覆写[`message`](#django.core.validators.EmailValidator.message "django.core.validators.EmailValidator.message")。 * **code** – 如果不是 `None`，则覆写[`code`](#django.core.validators.EmailValidator.code "django.core.validators.EmailValidator.code")。 * **whitelist** – 如果不是`None`，则覆写 [`whitelist`](#django.core.validators.EmailValidator.whitelist "django.core.validators.EmailValidator.whitelist")。

`message`

验证失败时 `ValidationError` 所使用的错误信息。默认为 `"Enter a valid email address"`。

`code`

验证失败时 `ValidationError` 所使用的错误代码。默认为 `"invalid"`。

`whitelist`

所允许的邮件域名的白名单。通常，正则表达式(`domain_regex` 属性) 用于验证 @ 符号后面的任何东西。但是，如果这个字符串在白名单里，就可以通过验证。如果没有提供，默认的白名单是 `['localhost']`。其它不包含点符号的域名不能通过验证，所以你需要按需将它们添加进白名单。

URLValidator

class `URLValidator` ([*schemes=None*, *regex=None*, *message=None*, *code=None*])[source]

`RegexValidator` 确保一个值看起来像是URL，并且如果不是的话产生 `'invalid'` 错误代码。

回送地址以及保留的IP空间被视为有效。同时也支持字面的IPv6地址 (**RFC 2732**) 以及unicode域名。

除了父类 `RegexValidator` 的可选参数之外， `URLValidator` 接受一个额外的可选属性：

`schemes`

需要验证的URL/URI模式列表。如果没有提供，默认为 `['http', 'https', 'ftp', 'ftps']`。IANA 网站提供了 有效的URI模式的完整列表作为参考。

Changed in Django 1.7:

添加了可选的 `schemes` 属性。

Changed in Django 1.8:

添加了对IPv6 地址, unicode 域名, 以及含有验证信息的URL的支持。

validate_email

`validate_email`

一个不带有任何自定义的 `EmailValidator` 实例。

validate_slug

`validate_slug`

一个 `RegexValidator` 实例，确保值只含有字母、数字、下划线和连字符。

validate_ipv4_address

`validate_ipv4_address`

一个 `RegexValidator` 的实例，确保值是IPv4地址。

validate_ipv6_address

`validate_ipv6_address` [source]

使用 `django.utils.ipv6` 来检查是否是 IPv6 地址。

validate_ipv46_address

`validate_ipv46_address` [source]

使用 `validate_ipv4_address` 和 `validate_ipv6_address` 值是有效的 IPv4 或 IPv6 地址。

validate_comma_separated_integer_list

`validate_comma_separated_integer_list`

一个 `RegexValidator` 的实例，确保值是整数的逗号分隔列表。

MaxValueValidator

class `MaxValueValidator`(*max_value*, *message=None*)[source]

如果 `value` 大于 `max_value`，抛出带有 `'max_value'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MinValueValidator

class `MinValueValidator`(*min_value*, *message=None*)[source]

如果 `value` 小于 `min_value`，抛出带有 `'min_value'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MaxLengthValidator

class `MaxLengthValidator`(*max_length*, *message=None*)[source]

如果 `value` 的长度大于 `max_length`，抛出带有 `'max_length'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

MinLengthValidator

class `MinLengthValidator`(*min_length*, *message=None*)[source]

如果 `value` 的长度小于 `min_length`，抛出带有 `'min_length'` 代码的 `ValidationError` 异常。

Changed in Django 1.8:

添加了 `message` 参数。

> 译者：Django 文档协作翻译小组，原文：Data validation。
>
>
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

其它核心功能

学习django框架的其他核心功能：

按需内容处理

HTTP客户端可能发送一些协议头来告诉服务端它们已经看过了哪些资源。这在获取网页（使用HTTP `GET` 请求）时非常常见，可以避免发送客户端已经获得的完整数据。然而，相同的协议头可用于所有HTTP方法(`POST` , `PUT` , `DELETE` , 以及其它)。

对于每一个Django从视图发回的页面（响应），都会提供两个HTTP协议头： `ETag` 和 `Last-Modified` 。这些协议头在HTTP响应中是可选的。它们可以由你的视图函数设置，或者你可以依靠 `CommonMiddleware` 中间件来设置 `ETag` 协议头。

当你的客户端再次请求相同的资源时，它可能会发送 If-modified-since 或者If-unmodified-since的协议头，包含之前发送的最后修改时间；或者 If-match 或If-none-match协议头，包含之前发送的 `ETag` 。如果页面的当前版本匹配客户端发送的 `ETag` ，或者如果资源没有被修改，会发回304状态码，而不是一个完整的回复，告诉客户端没有任何修改。根据协议头，如果页面被修改了，或者不匹配客户端发送的 `ETag` ，会返回412（先决条件失败，Precondition Failed）状态码。

当你需要更多精细化的控制时，你可以使用每个视图的按需处理函数。

Changed in Django 1.8:

向按需视图处理添加 `If-unmodified-since` 协议头的支持

The `condition`

有时（实际上是经常），你可以创建一些函数来快速计算出资源的ETag值或者最后修改时间，并不需要执行构建完整视图所需的所有步骤。Django可以使用这些函数来为视图处理提供一个“early bailout”的选项。来告诉客户端，内容自从上次请求并没有任何改动。

这两个函数作为参数传递到 `django.views.decorators.http.condition` 装饰器中。这个装时期使用这两个函数（如果你不能既快又容易得计算出来，你只需要提供一个）来弄清楚是否HTTP请求中的协议头匹配那些资源。如果它们不匹配，会生成资源的一份新的副本，并调用你的普通视图。

`condition` 装饰器的签名为i：

```
condition(etag_func=None, last_modified_func=None)
```

计算ETag的最后修改时间的两个函数，会以相同的顺序传入 `request` 对象和相同的参数，就像它们封装的视图函数那样。 `last_modified_func` 函数应该返回一个标准的datetime值，它制订了资源修改的最后时间，或者资源不存在为 `None` 。传递给 `etag` 装饰器的函数应该返回一

个表示资源Etag的字符串，或者资源不存在时为 `None` 。

用一个例子可以很好展示如何使用这一特性。假设你有这两个模型，表示一个简单的博客系统：

```
import datetime
from django.db import models

class Blog(models.Model):
    ...

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    published = models.DateTimeField(default=datetime.datetime.now)
    ...
```

如果头版展示最后的博客文章，仅仅在你添加新文章的时候修改，你可以非常快速地计算出最后修改时间。你需要这个博客每一篇文章的最后 `发布` 日期。实现它的一种方式是：

```
def latest_entry(request, blog_id):
    return Entry.objects.filter(blog=blog_id).latest("published").published
```

接下来你可以使用这个函数，来为你的头版视图事先探测未修改的页面：

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

@condition(last_modified_func=latest_entry)
def front_page(request, blog_id):
    ...
```

只计算一个值的快捷方式

一个普遍的原则是，如果你提供了计算 ETag和最后修改时间的函数，你应该这样做：你并不知道HTTP客户端会发给你哪个协议头，所以要准备好处理两种情况。但是，有时只有二者之一容易计算，并且Django只提供给你计算ETag或最后修改日期的装饰器。

`django.views.decorators.http.etag` 和 `django.views.decorators.http.last_modified` 作为 `condition` 装饰器，传入相同类型的函数。他们的签名是：

```
etag(etag_func)
last_modified(last_modified_func)
```

我们可以编写一个初期的示例，它仅仅使用最后修改日期的函数，使用这些装饰器之一：

```
@last_modified(latest_entry)
def front_page(request, blog_id):
    ...
```

...或者：

```
def front_page(request, blog_id):
    ...
front_page = last_modified(latest_entry)(front_page)
```

Use `condition`

如果你想要测试两个先决条件，把 `etag` 和 `last_modified` 装饰器链到一起看起来很不错。但是，这会导致不正确的行为：

```
# Bad code. Don't do this!
@etag(etag_func)
@last_modified(last_modified_func)
def my_view(request):
    # ...

# End of bad code.
```

第一个装饰器不知道后面的任何事情，并且可能发送“未修改”的响应，即使第二个装饰器会处理别的事情。 `condition` 装饰器同时更使用两个回调函数，来弄清楚哪个是正确的行为。

使用带有其它HTTP方法的装饰器

`condition` 装饰器不仅仅对 `GET` 和 `HEAD` 请求有用（ `HEAD` 请求在这种情况下和 `GET` 相同）。它也可以用于为 `POST` , `PUT` 和 `DELETE` 请求提供检查。在这些情况下，不是要返回一个“未修改（not modified，314）”的响应，而是要告诉服务端，它们尝试修改的资源在此期间被修改了。

例如，考虑以下客户端和服务端之间的交互：

1. 客户端请求 `/foo/` 。
2. 服务端回复一些带有 `"abcd1234"` ETag的内容。
3. 客户端发送HTTP `PUT` 请求到 `/foo/` 来更新资源。同时也发送了 `If-Match: "abcd1234"` 协议头来指定尝试更新的版本。
4. 服务端检查是否资源已经被修改，通过和 `GET` 上所做的相同方式计算ETag（使用相同的函数）。如果资源 已经 修改了，会返回412状态码，意思是“先决条件失败（precondition failed）”。
5. 客户端在接收到412响应之后，发送 `GET` 请求到 `/foo/` ，来在更新之前获取内容的新版本。

重要的事情是，这个例子展示了在所有情况下，ETag和最后修改时间值都采用相同函数计算。实际上，你 应该 使用相同函数，以便每次都返回相同的值。

使用中间件按需处理来比较

你可能注意到，Django已经通过 `django.middleware.http.ConditionalGetMiddleware` 和 `CommonMiddleware` .提供了简单和直接的 `GET` 的按需处理。这些中间件易于使用并且适用于多种情况，然而它们的功能有一些高级用法上的限制：

- 它们在全局上用于你项目中的所有视图。
- 它们不会代替你生成响应本身，这可能要花一些代价。
- 它们只适用于HTTP `GET` 请求。

在这里，你应该选择最适用于你特定问题的工具。如果你有办法快速计算出ETag和修改时间，并且如果一些视图需要花一些时间来生成内容，你应该考虑使用这篇文档描述的 `condition` 装饰器。如果一些都执行得非常快，坚持使用中间件在如果视图没有修改的条件下也会使发回客户端的网络流量也会减少。

> 译者：Django 文档协作翻译小组，原文：Conditional content processing。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

重定向应用

Django 原生自带一个可选的重定向应用。它将简单的重定向保存到数据库中并处理重定向。它默认使用HTTP 响应状态码 `301 Moved Permanently`。

安装

请依照下面的步骤安装重定向应用：

1. 确保 `django.contrib.sites` 框架已经安装。
2. 添加'`django.contrib.redirects`'到 `INSTALLED_APPS` 设置中。
3. 添加'`django.contrib.redirects.middleware.RedirectFallbackMiddleware`'到 `MIDDLEWARE_CLASSES` 设置中。
4. 运行命令 `manage.py migrate`。

它是如何工作的

`manage.py migrate` 在数据库中创建一张 `django_redirect` 表。它是一张简单的查询表，具有 `site_id`、 `old_path` 和 `new_path` 字段。

`RedirectFallbackMiddleware` 完成所有的工作。每当Django 的应用引发一个404 错误，该中间件将到重定向数据库中检查请求的URL。它会根据 `old_path` 和 `SITE_ID` 设置的站点ID 查找重定向的路径。

- 如果找到匹配的记录且 `new_path` 不为空，它将使用301(“Moved Permanently”)重定向到 `new_path`。你可以子类化 `RedirectFallbackMiddleware` 并设置 `response_redirect_class` 为 `django.http.HttpResponseRedirect` 来使用302 Moved Temporarily 重定向。
- 如果找到匹配的记录而 `new_path` 为空，它将发送一个410 (“Gone”) HTTP 头和空（没有内容的）响应。
- 如果没有找到匹配的记录，请求将继续正常处理。

这个中间件只针对404 错误启用 —— 不能用于500 或其它状态码。

注意 `MIDDLEWARE_CLASSES` 的顺序很重要。通常可以将 `RedirectFallbackMiddleware` 放在列表的最后，因为它最后执行。

更多的信息可以阅读中间件的文档。

如何添加、修改和删除重定向

通过Admin 接口

如果你已经启用Django 自动生成的 `Admin` 接口，你应该可以在 `Admin` 的主页看到“Redirects”部分。编辑这些重定向，就像编辑系统中的其它对象一样。

通过Python API

`class models.Redirect`

重定向通过一个标准的Django 模型表示，位于 `django/contrib/redirects/models.py` 。你可以通过Django 的数据库API 访问重定向对象。

中间件

`class middleware.RedirectFallbackMiddleware`

你可以通过创建 `RedirectFallbackMiddleware` 的子类并覆盖 `response_gone_class` 和/或 `response_redirect_class` 来修改中间件使用的 `HttpResponse` 类。

`response_gone_class`

```
New in Django 1.7.
```

`HttpResponse` 类，用于找不到请求路径的 `Redirect` 或找到的 `new_path` 值为空的时候。

默认为 `HttpResponseGone` 。

`response_redirect_class`

```
New in Django 1.7.
```

处理重定向的 `HttpResponse` 类。

默认为 `HttpResponsePermanentRedirect` 。

> 译者：Django 文档协作翻译小组，原文：Redirects。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

信号

Django包含一个“信号的分发器”，允许解耦的应用在信号出现在框架的任何地方时，都能获得通知。简单来说，信号允许指定的 发送器通知一系列的接收器，一些操作已经发生了。当一些代码会相同事件感兴趣时，会十分有帮助。

Django 提供了一系列的内建信号，允许用户的代码获得DJango的特定操作的通知。这包含一些有用的通知：

- `django.db.models.signals.pre_save` & `django.db.models.signals.post_save`

  在模型 `save()` 方法调用之前或之后发送。

- `django.db.models.signals.pre_delete` & `django.db.models.signals.post_delete`

  在模型 `delete()` 方法或查询集的 `delete()` 方法调用之前或之后发送。

- `django.db.models.signals.m2m_changed`

  模型上的 `ManyToManyField` 修改时发送。

- `django.core.signals.request_started` & `django.core.signals.request_finished`

  Django建立或关闭HTTP 请求时发送。

关于完整列表以及每个信号的完整解释，请见内建信号的文档 。

你也可以定义和发送你自己的自定义信号；见下文。

监听信号

你需要注册一个接收器函数来接受信号，它在信号使用 `Signal.connect()` 发送时被调用：

`Signal.``connect` (*receiver*[, *sender=None*, *weak=True*, *dispatch_uid=None*])

Parameters:	* **receiver** – 和这个信号连接的回调函数。详见[_接收器函数_](#receiver-functions)。 * **sender** – 指定一个特定的发送器，来从它那里接受信号。详见[_连接由指定发送器发送的信号_](#connecting-to-specific-signals)。 * **weak** – DJango通常以弱引用储存信号处理器。这就是说，如果你的接收器是个局部变量，可能会被垃圾回收。当你调用信号的`connect()`方法是，传递 `weak=False`来防止这样做。 * **dispatch_uid** – 一个信号接收器的唯一标识符，以防信号多次发送。详见[_防止重复的信号_](#preventing-duplicate-signals)。

让我们来看一看它如何通过注册在每次在HTTP请求结束时调用的信号来工作。我们将会连接到 `request_finished` 信号。

接收器函数

首先，我们需要定义接收器函数。接受器可以是Python函数或者方法：

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

注意函数接受 `sender` 函数，以及通配符关键字参数(`**kwargs`)；所有信号处理器都必须接受这些参数。

我们过一会儿再关注发送器，现在先看一看 `**kwargs` 参数。所有信号都发送关键字参数，并且可以在任何时候修改这些关键字参数。在 `request_finished` 的情况下，它被记录为不发送任何参数，这意味着我们可能需要像 `my_callback(sender)` 一样编写我们自己的信号处理器。

这是错误的 -- 实际上，如果你这么做了，Django会抛出异常。这是因为无论什么时候信号中添加了参数，你的接收器都必须能够处理这些新的参数。

连接接收器函数

有两种方法可以将一个接收器连接到信号。你可以采用手动连接的方法：

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

或者使用 `receiver()` 装饰器来自动连接：

`receiver` (*signal*)

Parameters:	**signal** – A signal or a list of signals to connect a function to.

下面是使用装饰器连接的方法：

```
from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

现在，我们的 `my_callback` 函数会在每次请求结束时调用。

这段代码应该放在哪里？

严格来说，信号处理和注册的代码应该放在你想要的任何地方，但是推荐避免放在应用的根模块和 `models` 模块中，以尽量减少产生导入代码的副作用。

实际上，信号处理通常定义在应用相关的 `signals` 子模块中。信号接收器在你应用配置类中的 `ready()` 方法中连接。如果你使用；额 `receiver()` 装饰器，只是在 `ready()` 内部导入 `signals` 子模块就可以了。

Changed in Django 1.7:

由于 `ready()` 并不在Django之前版本中存在，信号的注册通常在 `models` 模块中进行。

注意

`ready()` 方法会在测试期间执行多次，所以你可能想要防止重复的信号，尤其是打算在测试中发送它们的情况。

连接由指定发送器发送的信号

一些信号会发送多次，但是你只想接收这些信号的一个确定的子集。例如，考虑 `django.db.models.signals.pre_save` 信号，它在模型保存之前发送。大多数情况下，你并不需要知道 任何模型何时保存 -- 只需要知道一个特定的模型何时保存。

在这些情况下，你可以通过注册来接收只由特定发送器发出的信号。对于 `django.db.models.signals.pre_save` 的情况， 发送者是被保存的模型类，所以你可以认为你只需要由某些模型发出的信号：

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

`my_handler` 函数只在 `MyModel` 实例保存时被调用。

不同的信号使用不同的对象作为他们的发送器；对于每个特定信号的细节，你需要查看内建信号的文档。

防止重复的信号

在一些情况下，向接收者发送信号的代码可能会执行多次。这会使你的接收器函数被注册多次，并且导致它对于同一信号事件被调用多次。

如果这样的行为会导致问题（例如在任何时候模型保存时使用信号来发送邮件），传递一个唯一的标识符作为 `dispatch_uid` 参数来标识你的接收器函数。标识符通常是一个字符串，虽然任何可计算哈希的对象都可以。最后的结果是，对于每个唯一的 `dispatch_uid` 值，你的接收器函数都只被信号调用一次：

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

定义和发送信号

你的应用可以利用信号功能来提供自己的信号。

定义信号

class `Signal` ([*providing_args=list*])

所有信号都是 `django.dispatch.Signal` 的实例。 `providing_args` 是一个列表，包含参数的名字，它们由信号提供给监听者。理论上是这样，但是实际上并没有任何检查来保证向监听者提供了这些参数。

例如：

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

这段代码声明了 `pizza_done` 信号，它向接受者提供 `toppings` 和 `size` 参数。

要记住你可以在任何时候修改参数的列表，所以首次尝试的时候不需要完全确定API。

发送信号

Django中有两种方法用于发送信号。

`Signal.``send` (*sender*, ***kwargs*)

`Signal.``send_robust` (*sender*, ***kwargs*)

调用 `Signal.send()` 或者 `Signal.send_robust()` 来发送信号。你必须提供 `sender` 参数（大多数情况下它是一个类），并且可以提供尽可能多的关键字参数。

例如，这样来发送我们的 `pizza_done` 信号：

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
        ...
```

`send()` 和 `send_robust()` 都会返回一个含有二元组的列表 `[(receiver, response), ...]`，它代表了被调用的接收器函数和他们的响应值。

`send()` 与 `send_robust()` 在处理接收器函数产生的异常时有所不同。`send()` 不会 捕获任何由接收器产生的异常。它会简单地让错误往上传递。所以在错误产生的情况，不是所有接收器都会获得通知。

`send_robust()` 捕获所有继承自Python `Exception` 类的异常，并且确保所有接收器都能得到信号的通知。如果发生了错误，错误的实例会在产生错误的接收器的二元组中返回。

New in Django 1.8:

调用 `send_robust()` 的时候，所返回的错误的 `__traceback__` 属性上会带有 traceback。

断开信号

`Signal.``disconnect` ([*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*])

调用 `Signal.disconnect()` 来断开信号的接收器。`Signal.connect()` 中描述了所有参数。如果接收器成功断开，返回 `True`，否则返回 `False`。

`receiver` 参数表示要断开的已注册接收器。如果 `dispatch_uid` 用于定义接收器，可以为 `None`。

Changed in Django 1.8:

增加了返回的布尔值。

> 译者：Django 文档协作翻译小组，原文：Signals。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。

系统检查框架

New in Django 1.7.

系统检查框架是为了验证Django项目的一系列静态检查。它可以检测到普遍的问题，并且提供如何修复的提示。这个框架可以被扩展，所以你可以轻易地添加你自己的检查。

检查可以由 `check` 命令显式触发。检查会在大多数命令之前隐式触发，包括 `runserver` 和 `migrate` 。由于性能因素，检查不作为在部署中使用的WSGI栈的一部分运行。如果你需要在你的部署服务器上运行系统检查，显式使用 `check` 来触发它们。

严重的错误会完全阻止Django命令(像 `runserver`)的运行。少数问题会通过控制台来报告。如果你检查了警告的原因，并且愿意无视它，你可以使用你项目设置文件中的 `SILENCED_SYSTEM_CHECKS` 设置，来隐藏特定的警告。

系统检查参考中列出了所有Django可执行的所有检查。

编写你自己的检查

这个框架十分灵活，允许你编写函数，执行任何其他类型的所需检查。下面是一个桩（stub）检查函数的例子：

```
from django.core.checks import register

@register()
def example_check(app_configs, **kwargs):
    errors = []
    # ... your check logic here
    return errors
```

检查函数必须接受 `app_configs` 参数；这个参数是要被检查的应用列表。如果是None，检查会运行在项目中所有安装的应用上。 `**kwargs` 参数用于进一步的扩展。

消息

这个函数必须返回消息的列表。如果检查的结果中没有发现问题，检查函数必须返回一个空列表。

class `CheckMessage` (*level*, *msg*, *hint*, *obj=None*, *id=None*)

由检查方法产生的警告和错误必须是 `CheckMessage` 的示例。 `CheckMessage` 的实例封装了一个可报告的错误或者警告。它同时也提供了可应用到消息的上下文或者提示，以及一个用于过滤的唯一的标识符。

它的概念非常类似于消息框架或者 日志框架中的消息。消息使用表明其严重性的 `level` 来标记。

构造器的参数是：

`level`

The severity of the message. Use one of the predefined values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. If the level is greater or equal to `ERROR`, then Django will prevent management commands from executing. Messages with level lower than `ERROR` (i.e. warnings) are reported to the console, but can be silenced.

`msg`

A short (less than 80 characters) string describing the problem. The string should *not* contain newlines.

`hint`

A single-line string providing a hint for fixing the problem. If no hint can be provided, or the hint is self-evident from the error message, the hint can be omitted, or a value of `None` can be used.

`obj`

Optional. An object providing context for the message (for example, the model where the problem was discovered). The object should be a model, field, or manager or any other object that defines `__str__` method (on Python 2 you need to define `__unicode__` method). The method is used while reporting all messages and its result precedes the message.

`id`

Optional string. A unique identifier for the issue. Identifiers should follow the pattern `applabel.X001`, where `X` is one of the letters `CEWID`, indicating the message severity (`C` for criticals, `E` for errors and so). The number can be allocated by the application, but should be unique within that application.

也有一些快捷方式，使得创建通用级别的消息变得简单。当使用这些方法时你可以忽略 `level` 参数，因为它由类名称暗示。

class `Debug`(*msg*, *hint*, *obj=None*, *id=None*)

class `Info`(*msg*, *hint*, *obj=None*, *id=None*)

class `Warning`(*msg*, *hint*, *obj=None*, *id=None*)

class `Error`(*msg*, *hint*, *obj=None*, *id=None*)

class `Critical`(*msg*, *hint*, *obj=None*, *id=None*)

消息是可比较的。你可以轻易地编写测试：

```
from django.core.checks import Error
errors = checked_object.check()
expected_errors = [
    Error(
        'an error',
        hint=None,
        obj=checked_object,
        id='myapp.E001',
    )
]
self.assertEqual(errors, expected_errors)
```

注册和标记检查

最后，你的检查函数必须使用系统检查登记处来显式注册。

`register`(**tags)(function*)

你可以向 `register` 传递任意数量的标签来标记你的检查。Tagging checks is useful since it allows you to run only a certain group of checks. For example, to register a compatibility check, you would make the following call:

```
from django.core.checks import register, Tags

@register(Tags.compatibility)
def my_check(app_configs, **kwargs):
    # ... perform compatibility checks and collect errors
    return errors
```

New in Django 1.8.

你可以注册“部署的检查”，它们只和产品配置文件相关，像这样：

```
@register(Tags.security, deploy=True)
def my_check(app_configs, **kwargs):
    ...
```

这些检查只在 `--deploy` 选项传递给 `check` 命令的情况下运行。

你也可以通过向 `register` 传递一个可调用对象（通常是个函数）作为第一个函数，将 `register` 作为函数使用，而不是一个装饰器。

下面的代码和上面等价：

```
def my_check(app_configs, **kwargs):
    ...
register(my_check, Tags.security, deploy=True)
```

Changed in Django 1.8:

添加了将注册用作函数的功能。

字段、模型和管理器检查

在一些情况下，你并不需要注册检查函数 -- 你可以直接使用现有的注册。

字段、方法和模型管理器都实现了 `check()` 方法，它已经使用检查框架注册。如果你想要添加额外的检查，你可以扩展基类中的实现，进行任何你需要的额外检查，并且将任何消息附加到基类生成的消息中。强烈推荐你将每个检查分配到单独的方法中。

考虑一个例子，其中你要实现一个叫做 `RangedIntegerField` 的自定义字段。这个字段向 `IntegerField` 的构造器中添加 `min` 和 `max` 参数。你可能想添加一个检查，来确保用户提供了小于等于最大值的最小值。下面的代码段展示了如何实现这个检查：

```
from django.core import checks
from django.db import models

class RangedIntegerField(models.IntegerField):
    def __init__(self, min=None, max=None, **kwargs):
        super(RangedIntegerField, self).__init__(**kwargs)
        self.min = min
        self.max = max

    def check(self, **kwargs):
        # Call the superclass
        errors = super(RangedIntegerField, self).check(**kwargs)

        # Do some custom checks and add messages to `errors`:
        errors.extend(self._check_min_max_values(**kwargs))

        # Return all errors and warnings
        return errors

    def _check_min_max_values(self, **kwargs):
        if (self.min is not None and
                self.max is not None and
                self.min > self.max):
            return [
                checks.Error(
                    'min greater than max.',
                    hint='Decrease min or increase max.',
                    obj=self,
                    id='myapp.E001',
                )
            ]
        # When no error, return an empty list
        return []
```

如果你想要向模型管理器添加检查，应该在你的 `Manager` 的子类上执行同样的方法。

如果你想要向模型类添加检查，方法也大致相同：唯一的不同是检查是类方法，并不是实例方法：

```
class MyModel(models.Model):
    @classmethod
    def check(cls, **kwargs):
        errors = super(MyModel, cls).check(**kwargs)
        # ... your own checks ...
        return errors
```

> 译者：Django 文档协作翻译小组，原文：System check framework。
>
> 本文以 CC BY-NC-SA 3.0 协议发布，转载请保留作者署名和文章出处。
>
> Django 文档协作翻译小组人手紧缺，有兴趣的朋友可以加入我们，完全公益性质。交流群：467338606。