

[Django 1.10文档中文版](#)

目录

第一章、Django1.10文档组成结构

- 1.1 获取帮助
- 1.2 文档的组织形式
- 1.3 第一步
- 1.4 模型层
- 1.5 视图层
- 1.6 模板层
- 1.7 表单
- 1.8 开发流程
- 1.9 admin站点
- 1.10 安全性
- 1.11 国际化和本地化
- 1.12 性能和优化
- 1.13 Python兼容性
- 1.14 地理框架
- 1.15 常用WEB应用工具
- 1.16 其它核心功能
- 1.17 Django开源项目

第二章、起步

2.1 Django速览

- 2.1.1 设计你的模型model
- 2.1.2 安装model
- 2.1.3 使用API
- 2.1.4 功能强大的动态admin后台管理界面
- 2.1.5 设计你的路由系统URLs
- 2.1.6 编写你的视图views
- 2.1.7 设计你的模板
- 2.1.8 总结

2.2 快速安装指南

- 2.2.1 安装Python
- 2.2.2 安装Django
- 2.2.3 安装验证

2.3 第一个Django app, Part 1:请求和响应

- 2.3.1 创建project
- 2.3.2 开发服务器development server
- 2.3.3 创建投票程序(polls app)
- 2.3.4 编写视图

2.4 第一个Django app, Part 2:模型和admin站点

- 2.4.1 数据库安装
- 2.4.2 创建模型models
- 2.4.3 激活模型
- 2.4.4 学会使用API
- 2.4.5 Django admin站点介绍

2.5 第一个Django app, Part 3:视图和模板

- 2.5.1 概览
- 2.5.2 编写更多的视图
- 2.5.3 编写能实际干点活的视图
- 2.5.4 404错误
- 2.5.5 使用模板系统
- 2.5.6 删除模板中硬编码的URLs
- 2.5.7 URL names的命名空间

2.6 第一个Django app, Part 4:表单和泛型视图

- 2.6.1 编写一个简单的form
- 2.6.2 使用泛型视图: 减少代码冗余

2.7 第一个Django app, Part 5:测试

- 2.7.1 自动化测试介绍
- 2.7.2 基本的测试策略
- 2.7.3 编写我们的第一个测试程序
- 2.7.4 测试一个视图
- 2.7.5 测试越多越好
- 2.7.6 进一步测试

2.8 第一个Django app, Part 6:静态文件

- 2.8.1 自定义app的外观
- 2.8.2 添加背景图片

2.9 第一个Django app, Part 7:自定义admin站点

- 2.9.1 自定义admin表单
- 2.9.2 添加关系对象
- 2.9.3 自定义admin change list
- 2.9.4 定制admin外观
- 2.9.5 定制admin首页
- 2.9.6 接下来学习什么?

2.10 高级教程: 如何编写可重用的apps

- 2.10.1 重用的概念
- 2.10.2 你的项目和可重用的app
- 2.10.3 安装一些必要工具
- 2.10.4 打包你的app
- 2.10.5 使用你自己的包
- 2.10.6 发布你的app
- 2.10.7 如何在virtualenv中安装python的包

2.11 接下来学什么

2.12 编写你的第一个Django补丁

3.2 模型和数据库Models and databases

3.2.2 查询操作making queries

3.3.8 会话sessions

第一章、Django1.10文档组成结构

1.1 获取帮助

获取帮助的方式:

- 参考第5.1节的FAQ，这里包含许多常见问题的回答
- 查找特定的信息？尝试一下智能检索、混合索引或者内容详细表
- 在10.2.1的Django用户邮件列表中查找信息或发布问题
- 在django的IRC频道提问或者查找IRC日志看看是否已经有类似问题
- 报告Django中的bug

1.2 文档的组织形式

Django有许多文档，一个高层次的概览有助于你快速找到想要的信息：

- 入门教程：手把手的教会你创建一个WEB应用。如果你是个Django或者WEB应用开发者新手，从这里开始学习是个不错的选择。也可以先看看下面的“第一步”。
- 主题向导：讨论一些比较关键的主题和概念，属于比较高的水平，提供一些有用的背景信息和解释。
- 参考指南：包含API的使用指导和一些Django内核的其它方面。主要描述Django是怎么工作的，如何使用它。学习这部分的前提是你需要对一些关键概念具有基本的理解和掌握。
- “怎么办”向导：主要讨论一些关键的问题和使用场景。比入门教程更深入，需要你对Django的工作机制有一定的了解。

1.3 第一步

如果你是Django或程序员新手，那么从这里开始吧！

- 从零开始：概览 | 安装
- 入门教程：Part 1：请求和响应 | Part 2：模型和admin站点 | Part 3：视图和模板 | Part 4：表单和泛型 | Part 5：测试 | Part 6：静态文件 | Part 7：自定义admin站点
- 高级教程：如何编写可重用的应用 | 编写你的第一个Django补丁

1.4 模型层

Django提供了一个抽象的模型层，用于组织和操纵你的WEB应用数据。

- 模型：模型介绍 | 字段类型 | Meta选项 | 模型类
- 查询结果集：执行查询 | 查询结果集方法参考 | 查询表达式
- 模型实例：实例方法 | 访问关系对象
- 迁移：迁移介绍 | 操作参考 | 计划编辑 | 编写迁移
- 高级：管理器 | 原生SQL | 事务 | 聚合 | 查找 | 自定义字段 | 多数据库 | 自定义查询 | 查询表达式 | 条件表达式 | 数据库函数
- 其它：支持的数据库 | 遗留数据库 | 提供初始化数据 | 优化数据库访问 | PostgreSQL特有

1.5 视图层

Django将封装响应用户请求逻辑和返回数据称为“视图”。在下面的列表中，你能找到所有和视图相关的内容：

- 基础：路由配置 | 视图函数 | 快捷方式 | 装饰器
- 参考：内置视图 | 请求/响应对象 | 模板响应对象
- 文件上传：概览 | 文件对象 | 储存API | 管理文件 | 自定义存储
- 基类视图：概览 | 内置显示视图 | 内置编辑视图 | 混合使用 | API参考 | 分类索引
- 高级：生成CSV | 生成PDF
- 中间件：概览 | 内置中间件类

1.6 模板层

模板层提供一种人性化的语法，用于渲染展示给用户的内容，主要内容包括下列：

- 基础：概览
- 对于设计师：语言概览 | 内置标签和过滤器 | 人性化
- 对于程序员：模板API | 自定义标签和过滤器

1.7 表单

Django 提供了一个内容丰富的框架可便利地创建表单及操作表单数据。

- 基础：概览 | 表单 API | 内置字段 | 内置小工具
- 高级：模型表单 | 表单外观 | 表单集 | 自定义验证

1.8 开发流程

学习不同的组件和工具，帮助你开发和测试Django应用。

- 设置：概览 | 全部设置列表
- 应用：概览
- 异常：概览
- Django-admin和manage.py：概览 | 添加自定义命令
- 测试：介绍 | 编写和运行测试 | 导入测试工具 | 高级主题
- 部署：概览 | WSGI服务器 | 部署静态文件 | 通过邮件跟踪代码错误

1.9 admin站点

在这里，你可以找到Django中最受欢迎的功能模块——admin站点的一切：

- admin站点
- admin动作
- admin文档生成器

1.10 安全性

开发 Web 应用时安全是最重要一个的主题，Django 提供了多重保护工具和机制：

- 安全概览
- 披露的Django安全问题
- 点击劫持的防护
- 跨站请求伪造防护
- 加密签名
- 安全中间件

1.11 国际化和本地化

Django 提供了一个强大的国际化和本地化框架，以协助您开发支持多国语言和世界各地区的应用：

- 概览 | 国际化 | 本地化 | 本地WEB UI 格式化和表单输入
- 时区

1.12 性能和优化

有许技术和工具可以帮助你的代码运行得更加高效、快速，占用更少的系统资源。

- 性能和优化概览

1.13 Python兼容性

Django 希望兼容多个不同特性和版本的 Python：

- Jython支持
- Python3 兼容性

1.14 地理框架

GeoDjango 想要做一个世界级的地理Web框架。 它的目标是尽可能轻松的构建GIS Web 应用和发挥空间数据的能力。

1.15 常用WEB应用工具

Django 为开发Web应用提供了多种常见的工具：

- 身份认证：概览 | 使用认证系统 | 密码管理 | 自定义认证 | API参考
- 缓存
- 日志
- 发送邮件
- 联合供稿(RSS/Atom)
- 分页
- 消息框架

- 序列化
- 会话
- 站点地图
- 静态文件管理
- 数据验证

1.16 其它核心功能

Django的其它核心功能包括：

- 按需处理内容
- 内容类型和泛型关系
- 简单页面
- 重定向
- 信号
- 系统检查框架
- 站点框架
- 在 Django 中使用 Unicode

1.17 Django开源项目

下面是Django项目本身的开发进程和如何做出贡献相关：

- 社区：如何参与 | 发布过程 | 团队组织 | 团队会议 | 主要人员 | 源码仓库 | 安全策略 | 邮件列表
- 设计理念：概述
- 文档：关于本文档
- 第三方发行：概述
- Django 的过去：API 稳定性 | 发行说明和升级说明 | 功能弃用时间轴

第二章、起步

2.1 Django速览

Django的开发背景是快节奏的新闻编辑室环境，因此它被设计成一个大而全的web框架，能够快速简单的完成任务。本节将快速介绍如何利用Django搭建一个数据库驱动的WEB应用。

它不会有太多的技术细节，只是让你理解Django是如何工作的。

2.1.1 设计你的模型model

Django提供了ORM，通过它，你能直接使用Python代码来描述你的数据库设计。下面是一个例子：

```
# mysite/news/models.py

from django.db import models

class Reporter(models.Model):
    full_name = models.CharField(max_length=70)
    def __str__(self): # __unicode__ on Python 2
        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, on_delete=models.CASCADE)
    def __str__(self): # __unicode__ on Python 2
        return self.headline
```

2.1.2 安装model

接下来，进入Django命令行工具，创建数据库表：

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

migrate命令查找所有可用的model，如果它还没有在数据库中存在，将根据model创建相应的表。注：也许你需要先执行\$ python manage.py makemigrations命令。

2.1.3 使用API

Django为你提供了大量的方便的数据库操作API，无需你编写额外的代码。下面是个例子：

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article
# No reporters are in the system yet.
>>> Reporter.objects.all()
<QuerySet []>
# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')
# Save the object into the database. You have to call save() explicitly.
>>> r.save()
# Now it has an ID.
>>> r.id
1
# Now the new reporter is in the database.
>>> Reporter.objects.all()
<QuerySet [<Reporter: John Smith>]>
# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'
# Django provides a rich database lookup 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.
# Create an article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
... content='Yeah.', reporter=r)
>>> a.save()
# Now the article is in the database.
>>> Article.objects.all()
<QuerySet [<Article: Django is cool>]>
# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'
# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
<QuerySet [<Article: Django is cool>]>
# The API follows relationships as far as you need, performing efficient
# JOINS for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith='John')
<QuerySet [<Article: Django is cool>]>
# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()
# Delete an object with delete().
>>> r.delete()
```

2.1.4 功能强大的动态admin后台管理界面

Django包含一个功能强大的admin后台管理模块，使用方便，要素齐全。有助于你快速开发。你只需要在下面两个文件中写几句短短代码：

mysite/news/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, on_delete=models.CASCADE)
```

mysite/news/admin.py

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

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

2.1.5 设计你的路由系统URLs

Django主张干净、优雅的路由设计，不建议在路由中出现类似.php或.asp之类的字眼。

路由都写在URLconf文件中，它建立起URL匹配模式和python函数之间的映射，起到了解耦的作用。下面是一个例子：

mysite/news/urls.py

```
from django.conf.urls import url
from . import views
urlpatterns = [
    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),
]
```

Django通过正则表达式，分析访问请求的url地址，匹配相应的views，调用对应的函数。

2.1.6 编写你的视图views

每一个视图都必须做下面两件事情之一：返回一个包含请求页面数据的HttpResponse对象或者弹出一个类似404页面的异常。

通常，视图通过参数获取数据，并利用它们渲染加载的模板。下面是一个例子：

mysite/news/views.py

```
from django.shortcuts import render
from .models import Article
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    context = {'year': year, 'article_list': a_list}
    return render(request, 'news/year_archive.html', context)
```

2.1.7 设计你的模板

Django有一个模板查找路径，在settings文件中，你可以指定路径列表，Django自动按顺序在列表中查找你调用的模板。一个模板看起来是下面这样的：

mysite/news/templates/news/year_archive.html

```
{% 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 %}
```

Django使用自己的模板渲染语法，Jinja2就是参考它设计出来的。双大括号包含起来的是变量，它将被具体的值替换。圆点不但可以用来查询属性，也可以用来调用字典键值，列表索引和调用函数。

Django具有模板继承、导入和加载的概念，分别使用extend、include和load语法。下面是一个基础模板大概的样子：

mysite/templates/base.html

```
{% load static %}
<html>
<head>
<title>{% block title %}{% endblock %}</title>
</head>
<body>

{% block content %}{% endblock %}
</body>
</html>
```

子模板继承母模板的内容，并加入自己独有的部分。通过更换母版，可以快速的修改整改站点的外观和样式。

2.1.8 总结

Django为你提供了大量的模块和组件，包括模板系统、模型系统、视图系统以及其他一些通用组件和专用组件。他们之间都是独立的，同时也是可选的，你完全可以使用自己的模板、模型、视图。但是，Django给你提供的是一个集成度高的高效率整体框架，如果你自己的水平不是很高，那建议还是使用Django提供的吧。

2.2 快速安装指南

在第三章的第一节有详细的安装指南，这里只是一个简单的安装向导，用于快速搭建环境进入下面的章节。

2.2.1 安装Python

Django与python版本的对应关系。

Django version	Python versions
1.8	2.7, 3.2 (until the end of 2016), 3.3, 3.4, 3.5
1.9, 1.10	2.7, 3.4, 3.5
1.11	2.7, 3.4, 3.5, 3.6
2.0	3.5+

请前往Python官网下载并安装python。另外，python和Django自带轻量级数据库SQLite3，因此，在学习阶段你无需安装并配置其他的数据库。

2.2.2 安装Django

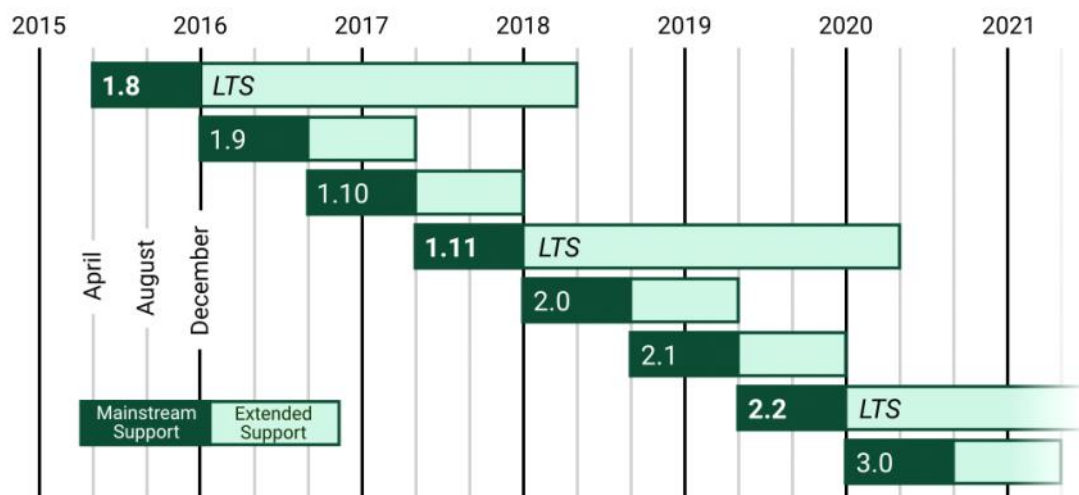
如果你是通过升级的方式安装Django，那么你需要先卸载旧的版本，具体查看3.1.4节。

你可以通过下面的3种方法安装Django：

- 安装你的操作系统提供的发行版本
- 安装官方版本（推荐）
- 安装开发版本

这里，请前往Django官网下载最新版本或通过`pip3 install django`进行安装。

下面是Django官方对版本的支持策略：



Release Series	Latest Release	End of mainstream support ¹	End of extended support ²
1.10	1.10.3	April 2017	December 2017
1.9	1.9.11	August 1, 2016	April 2017
1.8 LTS	1.8.16	December 1, 2015	Until at least April 2018
1.7	1.7.11	April 1, 2015	December 1, 2015
1.6	1.6.11	September 2, 2014	April 1, 2015
1.5	1.5.12	November 6, 2013	September 2, 2014
1.4 LTS	1.4.22	February 26, 2013	October 1, 2015
1.3	1.3.7	March 23, 2012	February 26, 2013

而这是未来发布版本的路线图：

Release Series	Release Date	End of mainstream support ¹	End of extended support ²
1.11 LTS ³	April 2017	December 2017	Until at least April 2020
2.0	December 2017	August 2018	April 2019
2.1	August 2018	April 2019	December 2019
2.2 LTS	April 2019	December 2019	Until at least April 2022
3.0	December 2019	August 2020	April 2021

2.2.3 安装验证

进入python环境，输入下列命令，注意版本号和你新安装的一致：

```
>>> import django
>>> print(django.get_version())
1.10
```

或者使用命令\$ python -m django --version查看版本号。

下面，我们将进入官方文档提供的编写第一个Django app教程！

2.3 第一个Django app, Part 1: 请求和响应

在这个例子中，我们将编写一个问卷调查网站，它包含下面两部分：

- 一个可以让人们进行投票和查看结果的公开站点
- 一个让你可以进行增删改查的后台admin管理界面

本教程使用Django 1.10 及Python 3.4以上版本！

2.3.1 创建project

进入你指定的某个目录，运行下面的命令：

```
$ django-admin startproject mysite
```

这将在目录下生成一个mysite目录，也就是你的这个Django项目的根目录。它包含了一系列自动生成的目录和文件，具备各自专有的用途。注意：在给项目命名时必须避开Django和Python的保留关键字，比如“django”，“test”等，否则会引起冲突和莫名的错误。对于mysite的放置位置，不建议放在传统的/var/www目录下，它会具有一定的数据暴露危险，因此Django建议你项目文件放在例如/home/mycode类似的位置。

一个新建立的项目结构大概如下：

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

详细解释：

- 外层的mysite/目录与Django无关，只是你项目的容器，可以任意命名。
- manage.py: 一个命令行工具，用于与Django进行不同方式的交互脚本，非常重要！
- 内层的mysite/目录是真正的项目文件包裹目录，它的名字是你引用内部文件的包名，例如：mysite.urls。
- mysite/init.py: 一个定义包的空文件。
- mysite/settings.py: 项目的主配置文件，非常重要！
- mysite/urls.py: 路由文件，所有的任务都是从这里开始分配，相当于Django驱动站点的内容表格，非常重要！
- mysite/wsgi.py: 一个基于WSGI的web服务器进入点，提供底层的网络通信功能，通常不用关心。

2.3.2 开发服务器development server

进入mysite目录，输入下面的命令：

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

你会看到下面的提示：

```
Performing system checks...
System check identified no issues (0 silenced).
You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.
September 07, 2016 - 15:50:53
Django version 1.10, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Django提供了一个用于开发的web服务器，使你无需配置一个类似Ngnix的线上服务器，就能让站点运行起来。但你也不要将开发服务器用于生产环境，它只是一个简易的测试服务器。

现在，在浏览器访问<http://127.0.0.1:8000/>，你将看到Django的欢迎页面，一切OK！

django开发服务器（以后省略）默认运行在内部的8000端口，如果你想指定，请在命令中显示给出，例如：

```
$ python manage.py runserver 0.0.0.0:8000
```

上面：Django将运行在8000端口，整个子网内都将可以访问，而不是本机。

注意：Django的开发服务器具有自动重载功能，当你的代码有修改，每隔一段时间服务器将自动更新。但是，有一些例如增加文件的动作，不会触发服务器重载，这时就需要你自己手动重启。

2.3.3 创建投票程序(polls app)

app与project的区别：

- 一个app实现某个功能，比如博客、公共档案数据库或者简单的投票系统；
- 一个project是配置文件和多个app的集合，他们组合成整个站点；
- 一个project可以包含多个app；
- 一个app可以属于多个project！

app的存放位置可以是任何地点，但是通常我们将它们都放在与manage.py同级目录下，这样方便导入文件。

进入mysite目录，确保与manage.py文件处于同一级，输入下述命令：

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

系统会自动生成 `polls` 目录，其结构如下：

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

2.3.4 编写视图

在 `polls/views.py` 文件中，输入下列代码：

```
from django.http import HttpResponse

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

为了调用该视图，我们还需要编写 `urlconf`。现在，在 `polls` 目录中新建一个文件，名字为 `urls.py`，在其中输入代码如下：

```
from django.conf.urls import url
from . import views

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

下一步是让项目的主 `urls` 文件指向我们建立的 `polls` 这个 `app` 独有的 `urls` 文件，你需要先导入 `include` 模块，打开 `mysite/urls.py` 文件，代码如下：

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

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

`include` 语法相当于二级路由策略，它将接收到的 `url` 地址去除了它前面的正则表达式，将剩下的字符串传递给下一级路由进行判断。在路由的章节，有更加详细的用法指导。

`include` 的背后是一种即插即用的思想。项目根路由不关心具体 `app` 的路由策略，只管往指定的二级路由转发，实现了解耦的特性。`app` 所属的二级路由可以根据自己的需要随意编写，不会和其它的 `app` 路由发生冲突。`app` 目录可以放置在任何位置，而不用修改路由。这是软件设计里很常见的一种模式。

建议：除了 `admin` 路由外，你应该尽量给每个 `app` 设计自己独立的二级路由。

好了，路由也搭建成功，下面我们启动服务器，然后在浏览器中访问地址 <http://localhost:8000/polls/>。一切正常的话，你将看到 “Hello, world. You’re at the polls index.”

`url()` 函数可以传递4个参数，其中2个是必须的：`regex` 和 `view`，以及2个可选的参数：`kwargs` 和 `name`。下面是具体的解释：

regex:

`regex` 是正则表达式的通用缩写，它是一种匹配字符串或 `url` 地址的语法。`Django` 拿着用户请求的 `url` 地址，在 `urls.py` 文件中对 `urlpatterns` 列表中的每一项条目从头开始进行逐一对比，一旦遇到匹配项，立即执行该条目映射的视图函数或二级路由，其后的条目将不再继续匹配。因此，`url` 路由的编写顺序至关重要！

需要注意的是，`regex` 不会去匹配 `GET` 或 `POST` 参数或域名，例如对于 `https://www.example.com/myapp/`，`regex` 只尝试匹配 `myapp/`。对于 `https://www.example.com/myapp/?page=3`，`regex` 也只尝试匹配 `myapp/`。

如果你想深入研究正则表达式，可以读一些相关的书籍或专论，但是在 `Django` 的实践中，你不需要多高深的正则表达式知识。

性能注释：正则表达式会进行预先编译当 `URLconf` 模块加载的时候，因此它的匹配搜索速度非常快，你通常感觉不到。

view:

当正则表达式匹配到某个条目时，自动将封装的 `HttpRequest` 对象作为第一个参数，正则表达式“捕获”到的值作为第二个参数，传递给该条目指定的视图。如果是简单捕获，那么捕获值将作为一个位置参数进行传递，如果是命名捕获，那么将作为关键字参数进行传递。

kwargs:

任意数量的关键字参数可以作为一个字典传递给目标视图。

name:

对你的URL进行命名，可以让你能够在Django的任意处，尤其是模板内显式地引用它。相当于给URL取了个全局变量名，你只需要修改这个全局变量的值，在整个Django中引用它的地方也将同样获得改变。这是极为古老、朴素和有用的设计思想，而且这种思想无处不在。

2.4 第一个Django app, Part 2:模型和admin站点

接着上一部分，本节将讲述如何安装数据库，编写第一个模型以及简要的介绍下Django自动生成的admin站点。

2.4.1 数据库安装

打开mysite/settings.py配置文件。Django默认使用内置的SQLite数据库。当然，如果你是在创建一个实际的项目，请使用类似MySQL的生产用数据库，避免以后面临数据库切换的头疼。

如果你想使用别的数据库，请先安装相应的数据库模块，并将settings文件中DATABASES 'default' 的键值进行相应的修改，用于连接你的数据库。其中：

ENGINE（引擎）：可以是' django.db.backends.sqlite3' 或者' django.db.backends.postgresql' , ' django.db.backends.mysql' , or ' django.db.backends.oracle' , 当然其它的也行。

NAME（名称）：数据库的名字。如果你使用的是默认的SQLite，那么数据库将作为一个文件将存放在你的本地机器内，NAME应该是这个文件的完整绝对路径，包括文件名。设置中的默认值os.path.join(BASE_DIR, 'db.sqlite3')，将把该文件储存在你的项目目录下。

如果你不是使用默认的SQLite数据库，那么一些诸如USER，PASSWORD和HOST的参数必须手动指定！更多细节参考后续的数据章节。

注意：

- 在使用非SQLite的数据库时，请务必首先在数据库提示符交互模式下创建数据库，你可以使用命令：“CREATE DATABASE database_name;”。
- 确保你在settings文件中提供的数据库用户具有创建数据库表的权限，因为在接下来的教程中，我们需要自动创建一个test数据库。
- 如果你使用的是SQLite，那么你无需做任何预先配置，直接使用就可以了。

在修改settings文件时，请顺便将TIME_ZONE设置为你所在的时区。

同时，请注意settings文件中顶部的INSTALLED_APPS设置项。它保存了所有的在当前项目中被激活的Django应用。你必须将你自定义的app注册在这里。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，INSTALLED_APPS中会自动包含下列条目，它们都是Django自动生成的：

- django.contrib.admin: admin站点
- django.contrib.auth: 身份认证系统
- django.contrib.contenttypes: 内容类型框架
- django.contrib.sessions: 会话框架
- django.contrib.messages: 消息框架
- django.contrib.staticfiles: 静态文件管理框架

上面的每个应用都至少需要使用一个数据库表，所以在它们之前我们需要在数据库中创建这些表。使用这个命令：\$ python manage.py migrate。

migrate命令将遍历INSTALLED_APPS设置中的所有项目，在数据库中创建对应的表，并打印出每一条动作信息。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。

提示：对于极简主义者，你完全可以在INSTALLED_APPS内注释掉任何或者全部的Django提供的通用应用。这样，migrate也不会再创建对应的数据表。

2.4.2 创建模型models

Django通过自定义python类的形式来定义具体的模型，每个模型代表数据库中的一张表，每个类的实例代表数据表中的一行数据，类中的每个变量代表数据表中的一列字段。Django通过ORM对数据库进行操作，奉行代码优先的理念，将python程序员和数据库管理员进行分分解耦。

在这个简单的投票应用中，我们将创建两个模型：Question和Choice。Question包含一个问题和一个发布日期。Choice包含两个字段：选择的文本和投票计数。每一条Choice都关联到一条Question。这些都是由python的类来体现，编写的全是python的代码，不接触任何sql语句。现在，编辑polls/models.py文件，具体代码如下：

polls/models.py

```
from django.db import models

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

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

上面的代码非常简单明了。每一个类都是`django.db.models.Model`的子类。每一个字段都是`Field`类的一个实例，例如用于保存字符串数据的`CharField`和用于保存时间类型的`DateTimeField`，它们告诉Django每一个字段保存的数据类型。

每一个 `Field` 实例的名字就是字段的名称（如：`question_text` 或者 `pub_date`）。在你的Python代码中会使用这个值，你的数据库也会将这个值作为表的列名。

你也可以在每个`Field`中使用一个可选的第一位置参数用于提供一个人类可读的字段名，让你的模型更友好，更易读，并且将被作为文档的一部分来增强代码的可读性。在本例中，仅定义了一个符合人类习惯的字段名`Question.pub_date`。对于模型中的其他字段，机器名称就已经足够我们认读了。

一些`Field`类必须提供某些特定的参数。例如`CharField`需要你指定`max_length`。这不仅是数据库结构的需要，同样也用于我们后面会谈到的数据验证功能。

有必填参数，当然就会有可选参数，比如在`votes`里我们将其默认值设为0。

最后请注意，我们使用`ForeignKey`定义了一个外键关系。它告诉Django，每一个`Choice`关联到一个对应的`Question`。Django支持通用的数据关系：一对一，多对一和多对多。

2.4.3 激活模型

上面的代码看着有点少，但却给予Django大量的信息，据此，Django会做下面两件事：

- 创建该app对应的数据库表结构
- 为`Question`和`Choice`对象创建基于python的数据库访问API

但是，首先，我们得先告诉项目，我们已经安装了投票应用。

Django思想：应用是“可插拔的”：你可以在多个项目使用一个应用，你也可以分发应用，它们不会被捆绑到一个给定的Django 项目中。

要将应用添加到项目中，需要在`INSTALLED_APPS`设置中增加指向该应用的配置文件的链接。对于本例的投票应用，它的配置文件是`polls/apps.py`，路径格式为`'polls.apps.PollsConfig'`。我们需要在`INSTALLED_APPS`中，将该路径添加进去。它看起来是这样的：

mysite/settings.py

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

现在Django已经知道你的投票应用的存在了，并把它加入了大家庭。我们再运行下一个命令：`$ python manage.py makemigrations polls`。你会看到类似下面的提示：

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

通过运行`migrations`命令，相当于告诉Django你对你的模型有改动，并且你想把这些改动保存为一个“迁移”。

`migrations`是Django保存模型修改记录的文件，它们是保存在磁盘上的文件。在例子中，它就是`polls/migrations/0001_initial.py`文件，你可以打开它看看，里面保存的都是可编辑的内容，方便你随时手动修改。

接下来有一个叫做`migrate`的命令将对数据库执行真正的迁移动作，下面我们就要介绍它。但是，在此之前，让我们先看看在`migration`的时候实际试行的SQL语句是什么。有一个叫做`sqlmigrate`的命令可以展示sql语句，例如：

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


你将会看到如下类似的文本（经过适当的格式调整，方便阅读）：

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
    ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;
COMMIT;
```

请注意：

- 实际的输出内容将取决于您使用的数据库会有所不同。上面的是PostgreSQL的输出。
- 表名是自动生成的，通过组合应用名 (`polls`) 和小写的模型名 – `question` 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"`。（你仍然可以重写此行为。）
- 外键关系由FOREIGN KEY显示声明。不要担心DEFERRABLE部分，它只是告诉PostgreSQL不要实施外键直到事务结束。
- 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 `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。
- 这些 sql 命令并比较在你的数据库中实际运行，它只是在屏幕上显示出来，以便让你了解 Django 真正执行的是什么。

如果你感兴趣，也可以运行`python manage.py check`命令，它将检查项目中所有没有进行迁移或者链接数据库的错误。

现在，我们可以运行`migrate`命令，在数据库中进行真正的表操作了。

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

`migrate`命令对所有还未实施的迁移记录进行操作，本质上就是将你对模型的修改体现到数据库中具体的表上面。Django通过一张叫做`django_migrations`的表，记录并跟踪已经实施的`migrate`动作，通过对比获得哪些`migrations`尚未提交。

`migrations`的功能非常强大，允许你随时修改你的模型，而不需要删除或者新建你的数据库或数据表，在不丢失数据的同时，实时动态更新数据库。我们将在后面的章节对此进行深入的阐述，但是现在，我们只需要记住修改模型时的操作分三步：

- 在`models.py`中修改模型
- 运行`python manage.py makemigrations`为改动创建迁移记录
- 运行`python manage.py migrate`，将迁移同步到数据库，落实修改动作。

之所以要将创建和实施迁移的动作分成两个命令两步走是因为你也许要通过版本控制系统（例如`github`，`svn`）提交你的项目代码，如果没有一个中间过程的保存文件（`migrations`），那么`github`如何知道以及记录、同步、实施你所进行过的模型修改动作呢？毕竟，`github`不和数据库直接打交道，也没法和你的本地数据库打交道。但是分开之后，你只需要将你的`migration`文件（例如上面的0001）上传到`github`，它就会知道一切。

2.4.4 学会使用API

下面，让我们进入python交互环境，学习使用Django提供的数据库访问API。要进入python的shell，请输入命令：

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

相比较直接输入“python”命令的方式进入python环境，调用manage.py参数能将DJANGO_SETTINGS_MODULE环境变量导入，它将自动按照mysite/settings.py中的设置，配置好你的python shell环境，这样，你就可以导入和调用任何你项目内的模块了。

或者你也可以这样，先进入一个纯净的python shell环境，然后启动Django，具体如下：

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

如果上述操作出现AttributeError异常，有可能是你正在使用一个和当前教程不匹配的Django版本。解决办法是学习较低版本的教程或更换更新版本的Django。

不管是哪种方式，你最终都是让python命令能够找到正确的模块地址，得到正确的导入。

当你进入shell后，尝试一下下面的API吧：

```
>>> from polls.models import Question, Choice # 导入我们写的模型类
# 现在系统内还没有questions
>>> Question.objects.all()
<QuerySet []>

# 创建一个新的question
# Django推荐使用timezone.now() 代替python内置的datetime.datetime.now()
from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# 你必须显式的调用save()方法，才能将对象保存到数据库内
>>> q.save()

# 默认情况，你会自动获得一个自增的名为id的主键
>>> q.id
1

# 通过python的属性调用方式，访问模型字段的值
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=UTC)

# 通过修改属性来修改字段的值，然后显式的调用save方法进行保存。
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() 用于查询数据库内的所有questions
>>> Question.objects.all()
<QuerySet [<Question: Question object>]>
```

这里等一下：上面的<Question: Question object>是一个不可读的内容展示，你无法从中获得任何直观的信息，为此我们需要一点小技巧，让Django在打印对象时显示一些我们指定的信息。返回polls/models.py文件，修改一下question和Choice这两个类，代码如下：

polls/models.py

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible # 当你想支持python2版本的时候才需要这个装饰器
class Question(models.Model):
    # ...
    def __str__(self): # 在python2版本中使用的是__unicode__
        return self.question_text

@python_2_unicode_compatible
class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

这个技巧不但对你打印对象时很有帮助，在你使用Django的admin站点时也同样有帮助。

请注意，这些都是普通的Python方法。下面我们自定义一个方法，作为示范：

polls/models.py

```
import datetime
```

```

from django.db import models
from django.utils import timezone

class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

```

请注意上面分别导入了两个关于时间的模块，一个是python内置的datetime一个是Django工具包提供的timezone。

保存修改后，我们重新启动一个新的python shell，再来看看其他的API:

```

>>> from polls.models import Question, Choice

# 先看看__str__()的效果，直观多了吧？
>>> Question.objects.all()
<QuerySet [ <Question: What's up?>]>

# Django提供了大量的关键字参数查询API
>>> Question.objects.filter(id=1)
<QuerySet [ <Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [ <Question: What's up?>]>

# 获取今年发布的问卷
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# 查询一个不存在的ID，会弹出异常
>>> Question.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Question matching query does not exist.

# Django为主键查询提供了一个缩写：pk。下面的语句和Question.objects.get(id=1)效果一样。
>>> Question.objects.get(pk=1)
<Question: What's up?>

# 看看我们自定义的方法用起来怎么样
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# 让我们试试主键查询
>>> q = Question.objects.get(pk=1)

# 显示所有与q对象有关系的choice集合，目前是空的，还没有任何关联对象。
>>> q.choice_set.all()
<QuerySet []>

# 创建3个choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice对象可通过API访问和他们关联的Question对象
>>> c.question
<Question: What's up?>

# 同样的，Question对象也可通过API访问关联的Choice对象
>>> q.choice_set.all()
<QuerySet [ <Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3

# API会自动进行连表操作，通过双下划线分割关系对象。连表操作可以无限多级，一层一层的连接。
# 下面是查询所有的Choices，它所对应的Question的发布日期是今年。（重用了上面的current_year结果）
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [ <Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# 使用delete方法删除对象
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()

```

获取更多关于模型关系的信息，请查看6.15.4章节。更多的数据库API和如何使用双下划线进行查询，请看3.2.2章节。

2.4.5 Django admin站点介绍

设计理念：为你的团队或客户编写用于增加、修改和删除内容的admin站点是一件非常乏味的工作并且没有多少创造性。因此，Django自动地为你通过模型构造了一个admin站点。这个站点只给站点管理员使用，并不对大众开放。

- 创建管理员用户

首先，通过下面的命令，创建一个可以登录admin站点的用户：

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

输入用户名：

Username: admin

输入邮箱地址：

Email address: admin@example.com

输入密码：

Password: *****

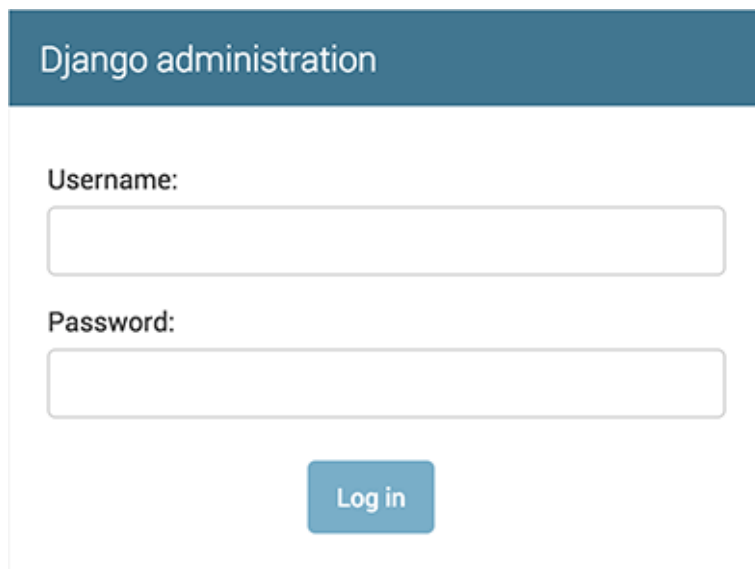
Password (again): *****

Superuser created successfully.

注意：Django1.10版本后，超级用户的密码强制要求具备一定的复杂性，不能再偷懒了。

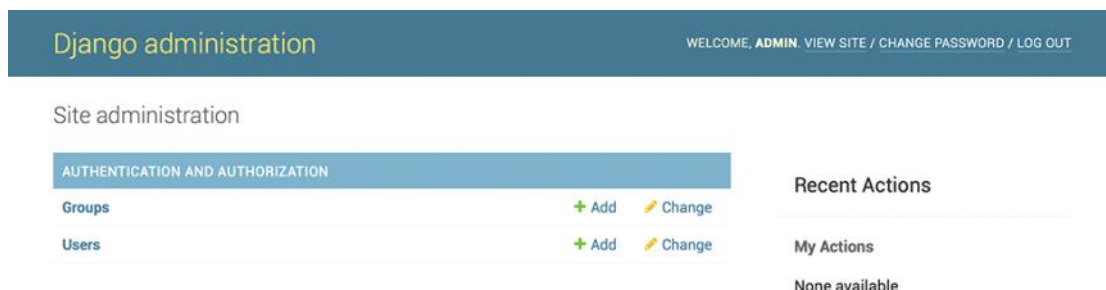
- 启动开发服务器

服务器启动后，在浏览器访问<http://127.0.0.1:8000/admin/>。你就能看到admin的登陆界面了：

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

- 进入admin站点

利用刚才建立的admin账户，登陆站点，你将看到如下的界面：

The image shows the Django administration dashboard after logging in. The header is dark blue with "Django administration" on the left and "WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)" on the right. Below the header, there's a section titled "Site administration". Under this section, there's a table with two rows: "Groups" and "Users". Each row has a green plus icon and the word "Add", and a yellow pencil icon and the word "Change". To the right of the table, there's a section titled "Recent Actions" with a sub-section "My Actions" and the text "None available".

当前已经有两个可编辑的内容：groups和users。它们是django.contrib.auth模块提供的身份认证框架。

- 在admin中注册你的投票应用

现在你还无法看到你的投票应用，必须先在admin中进行注册，告诉admin站点，请将poll的模型加入站点内，接受站点的管

理。

打开polls/admin.py文件，加入下面的内容：

polls/admin.py

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

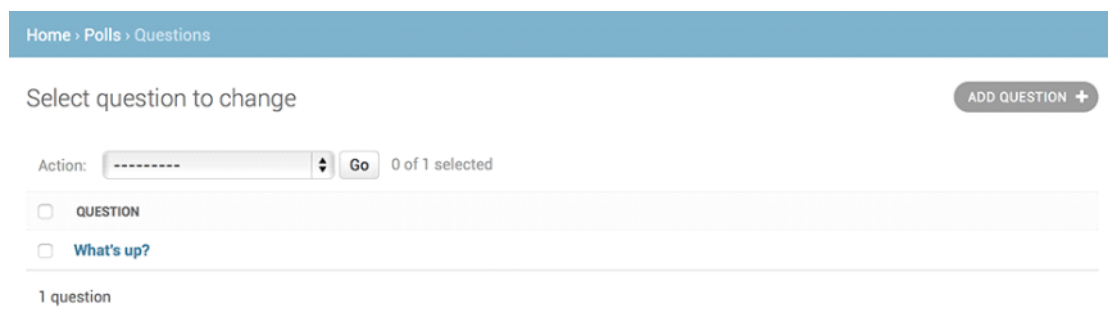
```
admin.site.register(Question)
```

- 浏览admin站点的功能

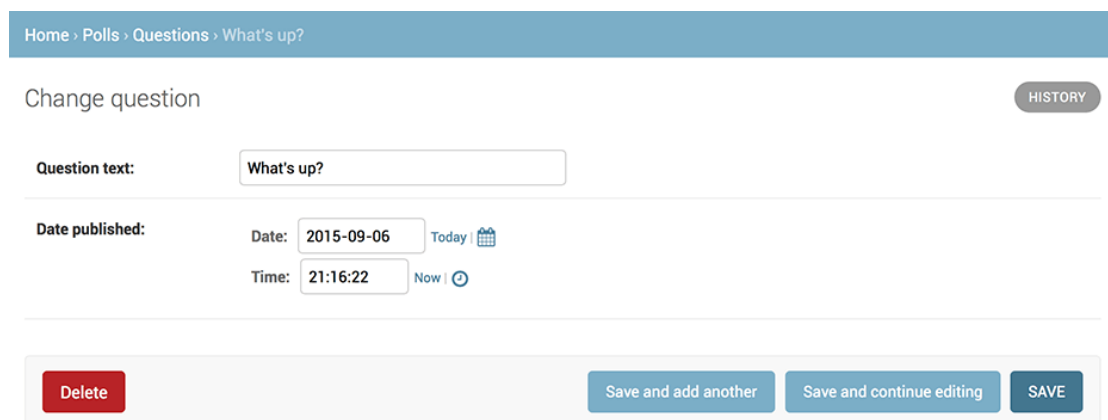
注册question模型后，刷新admin页面就能看到Question栏目了。



点击“Questions”，进入questions的修改列表页面。这个页面会显示所有的数据库内的questions对象，你可以在这里对它们进行修改。看到下面的“**What’s up?**”了么？它就是我们先前创建的一个question，并且通过__str__方法的帮助，显示了较为直观的信息，而不是一个冷冰冰的对象类型名称。



下面，点击**What’s up?**进入编辑界面：



这里需要注意的是：

- 表单是由Question模型自动生成的
- 不同的模型字段类型(DateTimeField, CharField)会表现为不同的HTML input框类型。
- 每一个DateTimeField都会获得一个JS缩写。日期的缩写是Today，并有一个日历弹出框；时间的缩写是Now，并有一个通用的时间输入列表框。

在页面的底部，则是一些可选项按钮：

- delete: 弹出一个删除确认页面
- save and add another: 保存当前修改，并加载一个新的空白的当前类型对象的表单。
- save and continue editing: 保存当前修改，并重新加载该对象的编辑页面。

- **save:** 保存修改，返回当前对象类型的列表页面。

如果“Date published”字段的值和你在前面教程创建它的时候不一致，可能是你没有正确的配置TIME_ZONE，在国内，通常是8个小时的时间差别。修改TIME_ZONE配置并重新加载页面，就能显示正确的时间了。

在页面的右上角，点击“History”按钮，你会看到你对当前对象的所有修改操作都在这里记录，包括修改时间和操作人，如下图所示：

Home > Polls > Questions > What's up? > History		
Change history: What's up?		
DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

2.5 第一个Django app, Part 3:视图和模板

本章承上启下，主要介绍Django的视图概念。

2.5.1 概览

一个视图就是一个网页“类型”，通常提供特定的功能或特定的模板。例如：在一个博客应用中，你可能会看到下列视图：

- 博客主页：显示最新发布的一些内容
- 条目详细页面：每个条目对应的永久页面
- 基于年的文章页面：显示指定年内的所有博客文章
- 基于月的文章页面：显示指定月内的所有博客文章
- 基于天的文章页面：显示指定日内的所有博客文章
- 发布评论：处理针对某篇博客发布的评论

在我们的投票应用中，我们将建立下面的视图：

- 问卷“index”页：显示最新的一些问卷
- 问卷“detail”页面：显示一个问卷的详细文本内容，没有调查结果但是有一个投票或调查表单。
- 问卷“results”页面：显示某个问卷的投票或调查结果。
- 投票动作页面：处理针对某个问卷的某个选项的投票动作。

在Django中，网页和其它的一些内容都是通过视图来分发的。视图表现为一个简单的Python函数（在基于类的视图中称为方法）。Django通过对比请求的URL地址来选择对应的视图。

在你平时的网页上，你可能经常会碰到类似“ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B”的url。庆幸的是Django支持使用更加简介的URL模式，而不需要编写上面那种复杂的url。

一个URL模式其实就是一个URL通用表达式，例如：/newsarchive///。为了使得URL模式映射到对应的视图，Django使用URLconfs来完成这一工作。本教程介绍基本的URLconfs使用方法，更多的内容，请参考6.23节。

2.5.2 编写更多的视图

下面，让我们打开polls/views.py文件，输入下列代码：

polls/views.py

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

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

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

然后，在polls/urls.py文件中加入下面的url模式，将其映射到我们上面新增的视图。

polls/urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

现在去浏览器中访问“/polls/34/”（注意：这里省略了域名。另外，使用了二级路由，url中都要添加polls部分，参考前面的章节），它将运行detail()方法，然后在页面中显示你在url里提供的ID。访问“/polls/34/results/”和“/polls/34/vote/”，将分别显示预定义的伪结果和投票页面。

上面访问的路由过程如下：当有人访问“/polls/34/”地址时，Django将首先加载mysite.urls模块，因为它是settings文件里设置的根URL配置文件。在该文件里，Django发现了urlpatterns变量，于是在其内按顺序的进行匹配。当它匹配上了^polls/，就脱去url中匹配的文本polls/，然后将剩下的文本“34/”，传递给“polls.urls”进行下一步的处理。在polls.urls，又匹配到了r'^(?P<question_id>[0-9]+)/\$'，最终结果就是调用该模式对应的detail()视图，也就是下面的函数：

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

函数中的question_id='34'参数，是由(?P[0-9]+)而来。在正则表达式中通过一个双圆括号，Django会捕获它匹配到的值并传递给对应的视图，作为视图的位置参数之一，而?P则表示我要给这个捕获的值指定一个特殊的变量名，在视图中可以通过question_id这个变量名随意的引用它，形成一个关键字参数，不用考虑参数的位置。至于[0-9]+则是一个很简单原生正则表达式，用于匹配一系列连续的数字，它匹配到的值也就是具体要传递的参数值。

所有的URL模式都是正则表达式，Django不限制你在url模式中的书写方式。但是，你真的没必要书写一个如下的较为愚蠢的包含".html"的模式，它显然是没必要，不够简练的：

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

你完全可以用下面的模式代替上面的：

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

2.5.3 编写能实际干点活的视图

前面我们说过，每个视图至少做两件事之一：返回一个包含请求页面的HttpResponse对象或者弹出一个类似Http404的异常。其它的则随你便，你爱干嘛干嘛。

你的视图可以从数据库读取记录，或者不读。你可以使用Django提供的模板系统或者第三方的Python模板系统，或者干脆啥也不用。你可以生成PDF文件、输出XML，创建ZIP压缩文件，任何你想做的事，使用任意你想用的Python库。

而Django想要的只有HttpResponse或者一个异常。

因为这样很简便，接下来让我们使用Django自己的数据库API，我们在上面的教程里介绍过的。下面是一个新的index()视图，它会根据发布日期显示最近的5个投票问卷，通过逗号分隔。

polls/views.py

```

from django.http import HttpResponseRedirect
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponseRedirect(output)

# 下面是那些没改动过的视图(detail, results, vote)

```

但是这里仍然有个问题：在视图中的页面时硬编码的。如果你想改变页面的显示，就必须修改这里的Python代码。因此，让我们来使用Django提供的模板系统，解耦视图和模板之间的联系。

首先，在polls目录下创建一个新的templates目录，Django会在它里面查找模板文件。

这里解释一下：Django项目的settings配置文件中定义了如何加载和渲染模板。默认的设置

是DjangoTemplates后端，并且APP_DIRS参数被设置为True。作为惯例，Django也会寻找每个在INSTALLED_APPS配置项里注册过的app本身目录下的templates子目录。

回到你刚才创建的templates目录中，再创建一个新的子目录名叫polls，进入该子目录，创建一个新的html文件index.html。换句话说，你的模板文件应该是polls/templates/polls/index.html。根据上面的解释，你现在可以在Django中直接使用polls/index.html引用该文件了。

模板命名空间：

你也许会想，为什么不把模板文件直接放在polls/templates目录下，而是费劲的再建个子目录polls呢？设想这么个情况，有另外一个app，它也有一个名叫index.html的文件，当Django在搜索模板时，有可能就找到它，然后退出搜索，这就命中了错误的目标，不是我们想要的结果。解决这个问题的最好办法就是在templates目录下再建立一个与app同名的子目录，将自己所属的模板都放到里面，从而达到独立命名空间的作用，不会再出现引用错误。

现在，将下列代码写入文件：

polls/templates/polls/index.html

```

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

```

同时，修改视图文件，让新的index.html文件生效：

polls/views.py

```

from django.http import HttpResponseRedirect
from django.template import loader
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponseRedirect(template.render(context, request))

```

上面的代码会加载polls? index.html文件，并传递给它一个参数，这个参数是一个字典，包含了模板变量名和python对象之间的映射关系。

在浏览器中通过访问“/polls/”，你可以看到一个列表，包含“What’s up”的问卷，以及连接到其对应详细内容页面的链接点。

快捷方式：render()

在实际运用中，加载模板、传递参数，返回HttpResponse对象是一整套再常用不过的操作了，为了节省力气，Django提供了一个快捷方式：render函数，一步到位！看如下代码：

polls/views.py

```

from django.shortcuts import render
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}

```

```
return render(request, 'polls/index.html', context)
```

`render()`函数的第一个位置参数是请求对象（就是`view`函数的第一个参数），第二个位置参数是模板，还可以有一个可选的第三参数---一个字典，包含需要传递给模板的数据。最后`render`函数返回一个经过字典数据渲染过的模板封装而成的`HttpResponse`对象。

2.5.4 404错误

现在让我们来编写返回具体问卷文本内容的视图：

`polls/views.py`

```
from django.http import Http404
from django.shortcuts import render
from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

这里有个新概念：如果请求的问卷ID不存在，那么会弹出一个`Http404`错误。

稍后我们会讨论你应该在`polls/detail.html`里面写点什么代码，但是现在你可以简单的先写这么个东西，用来展示上面的`404`错误：

`polls/templates/polls/detail.html`

```
{{ question }}
```

快捷方式：get_object_or_404()

就像`render`函数一样，Django同样为你提供了一个偷懒的方式，替代上面的多行代码，那就是`get_object_or_404()`方法，参考下面的代码：

`polls/views.py`

```
from django.shortcuts import get_object_or_404, render
from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

别说我没提醒你，和`render`一样，也需要在Django内置的快捷方式模块中导入`get_object_or_404()`！

`get_object_or_404()`函数将一个Django模型作为第一个位置参数，后面可以跟上任意个数的关键字参数（python函数参数的分类和语法一定要搞清楚了！这些关键字参数是传递给模型管理器的`get()`函数的，在后面会讲到。），如果对象不存在则弹出`Http404`错误。

理念：

为什么要费劲的使用一个`get_object_or_404()`快捷方式，而不是让系统自动的捕获`ObjectDoesNotExist`异常或者弹出模型API的`Http404`异常？仅仅只是为了少写点代码？

因为后两者会耦合模型层和视图层。Django的一个非常重要的设计目标是维持各层级之间的松耦合。更多的内容请参考3.3.5节。

同样，这里还有一个`get_list_or_404()`函数，和上面的`get_object_or_404()`类似，只不过是用来替代`filter()`函数，当查询列表为空时弹出`404`错误。（`filter`是模型API中用来过滤查询结果的函数，它的结果是一个列表集。而`get`则是查询一个结果的方法，和`filter`是一个和多个的区别！）

2.5.5 使用模板系统

回到`detail()`视图。将上下文变量`question`传递给对应的`html`模板，它看起来如下所示：

`polls/templates/polls/detail.html`

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

前面我们说过了，在模板系统中圆点“.”是万能的魔法师，你可以用它访问对象的属性。在例子`{{ question.question_text }}`中，Django首先会在`question`对象中尝试查找一个字典，如果失败，则尝试

在 `{% for %}` 循环中的方法调用——`poll.choice_set.all` 其实就是 Python 的代码 `poll.choice_set.all()`，它将返回一组可迭代的 `Choice` 对象，并用在 `{% for %}` 标签中。

2.5.6 删除模板中硬编码的URLs

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

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

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

```
# 添加新的单词'specifics'
url(r'^specifics/(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
```

2.5.7 URL names的命名空间

polls/urls.py

```
from django.conf.urls import url
from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

polls/templates/polls/index.html

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

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

注意引用方法是引号而不是圆点也不是斜杠！！！！！！！！！！！！！！！！！！！！

2.6 第一个Django app, Part 4:表单和泛型视图

2.6.1 编写一个简单的form

polls/templates/polls/detail.html

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

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

<form action="{% url 'polls:vote' question.id %}" method="post">
  {% csrf_token %}
```

```
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{% forloop.counter %}"
value="{{ choice.id }}" />
    <label for="choice{% forloop.counter %}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

简要说明：

- 上面的模板显示一系列单选按钮，按钮的值是选项的ID，按钮的名字是字符串"choice"。这意味着，当你选择了其中某个按钮，并提交表单，一个包含数据choice=#的POST请求将被发送到指定的url，#是被选择的选项的ID。这就是HTML表单的基本概念。
- 如果你有一定的前端开发基础，那么form标签的action属性和method属性你应该很清楚它们的含义，action表示你要发送的目的url，method表示提交数据的方式，一般分POST和GET，更多的解释就不是本教程干的事情了，你需要补课。
- forloop.counter是Django模板系统管理专门提供的一个变量，用来表示你当前循环的次数，一般用来给循环项目添加有序数标。
- 由于我们发送了一个POST请求，就必须考虑一个跨站请求伪造的问题，简称CSRF（具体含义请百度）。Django为你提供了一个简单的方法来避免这个困扰，那就是在form表单内添加一条{% csrf_token %}标签，标签名不可更改，固定格式，位置任意，只要是在form表单内。但是（译者注），这个方法对form表单的提交方式方便好使，但是如果是用ajax的方式提交数据，那么就很费劲了。个人认为不如直接在Django配置中关闭这个看似有作用，其实然并卵的CSRF得了。

现在，让我们创建一个处理提交过来的数据的视图。前面我们已经写了一个“占坑”的vote视图的url：

polls/urls.py

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

以及“占坑”的vote视图函数，我们把坑填起来：

polls/views.py

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.urls import reverse
from .models import Choice, Question
# ...

def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # 发生choice未找到异常时，重新返回表单页面，并给出提示信息
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # 成功处理数据后，自动跳转到结果页面，防止用户连续多次提交。
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

有些新的东西，我们要解释一下：

- request.POST是一个类似字典的对象，允许你通过键名访问提交的数据。本例中，request.POST['choice']返回被选择选项的ID，并且值的类型永远是string字符串，那怕它看起来像数字，记住了！！！！同样的，你也可以用类似的手段获取GET请求发送过来的数据，一个道理。
- request.POST['choice']有可能触发一个KeyError异常，如果你的POST数据里没有提供choice键值，在这种情况下，上面的代码会返回表单页面并给出错误提示。译者注：通常我们会给个默认值，防止这种异常的产生，例如：request.POST.get('choice', None)，一个None解决所有问题。
- 在选择计数器加一后，返回的是一个HttpResponseRedirect而不是先前我们常用的HttpResponse。HttpResponseRedirect需要一个参数：重定向的URL。这里有一个建议，当你成功处理POST数据后，应当保持一个良好的习惯，始终返回一个HttpResponseRedirect。这不仅仅是对Django而言，它是一个良好的WEB开发习惯。
- 我们在上面HttpResponseRedirect的构造器中使用了一个reverse()函数。它能帮助我们避免在视图函数中硬编码URL。它首先需要我们在URLconf中指定的name，然后是传递的数据。例如'/polls/3/results/'，其中的3是某个question.id的值。重定向后将进入'polls:results'对应的视图，并将question.id传递给它。白话来讲，就是把活扔给另外一个路由对应的视图去干。

当有人对某个问题投票后，vote()视图重定向到了问卷的结果显示页面。下面我们来写这个处理结果页面的视

图:

polls/views.py

```
from django.shortcuts import get_object_or_404, render

def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

同样，还需要写个模板。（译者注：路由、视图、模板、模型！你需要的套路....）

polls/templates/polls/results.html

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }}
    vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>
<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

现在你可以取浏览器中访问/polls/1/了，投票吧。你会看到一个结果页面，每投一次，它的内容就更新一次。如果你提交的时候没有选择项目，则会得到一个错误提示。

注释：（怎么这么多注释....）

在上面的vote视图中的代码存在一点小问题。如果有2个用户同时对某项进行提交时，很有可能发生同时对数据库进行读写的情况，它有可能导致数据的不协调，也就是所谓的“竞态”，如果你感兴趣，可以参考6.15节相关的通过使用F()查询来避免竞态的讨论和介绍。

2.6.2 使用泛型视图：减少代码冗余

上面的detail、index和results视图的代码非常相似，有点冗余，这是一个程序猿不能忍受的。他们都具有类似的业务逻辑，实现类似的功能：通过从URL传递过来的参数去数据库查询数据，加载一个模板，利用刚才的数据渲染模板，返回这个模板。由于这个过程是如此的常见，Django又很善解人意的帮你想办法偷懒了，它提供了一种快捷方式，名为“泛型视图”系统。

现在，让我们来试试看将原来的代码改为使用泛型视图的方式，整个过程分三步走：

- 改变URLconf
- 删除一些旧的无用的视图
- 采用基于泛型视图的新视图

注释：Django官方的“辩解”

为什么本教程的代码来回改动这么频繁？

答：通常在写一个Django的app时，我们一开始就要决定是使用泛型视图还是不用，而不是等到代码写到一半了才重构你的代码成泛型视图。但是本教程为了让你清晰的理解视图的内涵，“故意”走了一条比较2的路，因为我们的哲学是：在你使用计算器之前你得先知道基本的数学公式。

修改URLconf

打开polls/urls.py文件，将其修改成下面的样子：

```
from django.conf.urls import url
from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
    url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(), name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

请注意：在上面的第2,3条目中将原来的<question_id>修改成了<pk>。

修改视图

接下来，打开polls/views.py文件，删掉index、detail和results视图，替换成Django的泛型视图，如下所示：

polls/views.py

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect
```

```

from django.urls import reverse
from django.views import generic
from .models import Choice, Question

class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'
    def get_queryset(self):
        """返回最近发布的5个问卷."""
        return Question.objects.order_by('-pub_date')[:5]

class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'

class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'

def vote(request, question_id):
    ... # 这个视图未改变!!!

```

在这里，我们使用了两种泛型视图：**ListView**和**DetailView**（译者注：它们是作为父类被继承的）。这两者分别代表“显示一个对象的列表”和“显示特定类型对象的详细页面”的抽象概念。

- 每一种泛型视图都需要知道它要作用在哪个模型上，这通过`model`属性提供。
- **DetailView**泛型视图需要从URL捕获到的称为“pk”的主键值，因此我们在url文件中将2和3条目的`<question_id>`修改成了`<pk>`。

默认情况下，**DetailView**泛型视图使用一个称作`<app name>/<model name>_detail.html`的模板。在本例中，实际使用的是“`polls/question_detail.html`”。`template_name`属性就是用来指定这个模板名的，用于代替自动生成的默认模板名。（译者注：一定要仔细观察上面的代码，对号入座，注意细节。）同样的，在**results**列表视图中，为了指定`template_name`为“`polls/results.html`”，这样就确保了虽然**results**视图和**detail**视图同样继承了**DetailView**类，使用了同样的`model`：`Question`，但它们依然会显示不同的页面。（译者注：模板不同嘛！so easy!）

类似的，**ListView**泛型视图使用一个默认模板称为`<app name>/<model name>_list.html`。我们也使用`template_name`这个变量来告诉**ListView**使用我们已经存在的

“`polls/index.html`”模板，而不是使用它自己默认的那个。

在教程的前面部分，我们给模板提供了一个包含`question`和`latest_question_list`的上下文变量。而对于**DetailView**，`question`变量会被自动提供，因为我们使用了Django的模型（`Question`），Django会智能的选择合适的上下文变量。然而，对于**ListView**，自动生成的上下文变量是`question_list`。为了覆盖它，我们提供了`context_object_name`属性，指定说我们希望使用`latest_question_list`而不是`question_list`。

现在你可以运行开发服务器，然后试试基于泛型视图的应用程序了。

查看更多关于泛型视图的内容，请前往3.6节。

到这里，本节的内容结束了，你可以开始下一小节的学习。

2.7 第一个Django app, Part 5:测试

本章承上启下，主要介绍自动化测试相关的内容。

2.7.1 自动化测试介绍

什么是自动化测试

测试是一种例行工作用于检查你的代码的行为。

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？），一些测试则专注于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输入后，是否返回了期望的结果？）。这些其实和你早前在教程2中做的测试差不多，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是如何执行的。

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。一旦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那样运行，而无需执行耗时的手动测试。

为什么需要测试

那么，为什么要进行测试？而且为什么是现在？

你可能觉得自己的Python/Django能力已经足够，再去学习其他的东西也许不是那么的必要。毕竟，我们先前联系的投票应用已经表现得挺好了，将时间花在自动化测试上还不如用在改进我们的应用上。如果你学习Django就是为了创建这么一个简单的投票应用，那么进行自动化测试显然没有必要。但如果不是这样，那么现在是一个很好的学习机会。

测试可以节省你的时间

某种程度上，“检查并发现工作正常”似乎是种比较满意的测试结果。但在一些复杂的应用中，你会发现组件之间存在各种各样复杂的交互关系。

任何一个组件的改动，都有可能导致应用程序产生无法预料的结果。得出“似乎工作正常”的结果，可能意味着你需要使用二十种不同的测试数据来测试你的代码，而这仅仅是为了确保你没有搞砸某些事，很显然，这种方法效率低下。然而，自动化测试只需要数秒就可以完成以上的任务。如果出现了错误，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相比起有价值的、创造性的编程工作显得单调乏味、无趣，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生问题的原因，编

写测试程序的性价比还是很高的。

（译者：下面都是些测试重要性的论述，看标题就好了）

- 测试不仅仅可以发现问题，它们还能防止问题
- 测试使你的代码更受欢迎
- 测试有助于团队合作

2.7.2 基本的测试策略

编写测试程序有很多种方法。一些程序员遵循一种叫做“测试驱动开发”的规则，他们在编写代码前会先编好测试程序。看起来似乎有点反人类，但实际上这种方法与大多数人经常的做法很相似：先描述一个问题，然后编写代码来解决这个问题。测试驱动开发可以简单地用Python测试用例将问题格式化。

很多时候，刚接触测试的人会先编写一些代码后才编写测试程序。事实上，在之前就编写一些测试会好一点，但不管怎么说什么时候开始都不算晚。

有时候你很难决定从什么时候开始编写测试。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是不太容易的。这种情况下，在下次你对代码进行变更，添加一个新功能或者修复一个bug之时，编写你的第一个测试，效果会非常好。

下面，让我们马上来编写一个测试。

2.7.3 编写我们的第一个测试程序

发现BUG

很巧，在我们的投票应用中有一个小bug需要修改：在Question.was_published_recently()方法的返回值中，当Question在最近的一天发布的时候返回True（这是正确的），然而当Question在未来的日期内发布的时候也返回True（这是错误的）。

我们可以在admin后台创建一个发布日期在未来的Question，然后在shell中验证这个bug：

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # 创建一个发布日期在30天后的问卷
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # 测试一下返回值
>>> future_question.was_published_recently()
True
```

由于“将来”不等于“最近”，因此这显然是个bug。

创建一个测试来暴露这个bug

刚才我们是在shell中测试了这个bug，那如何通过自动化测试来发现这个bug呢？

通常，我们会把测试代码放在应用的tests.py文件中；测试系统将自动地从任何名字以test开头的文件中查找测试程序。

将下面的代码输入投票应用的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):
        """
        在将来发布的问卷应该返回False
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

我们在这里创建了一个django.test.TestCase的子类，它具有一个方法，该方法创建一个pub_date在未来的Question实例。最后我们检查was_published_recently()的输出，它应该是 False。

运行测试程序

在终端中，运行下面的命令，

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

你将看到结果如下：

```
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_future_question
self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False
-----
Ran 1 test in 0.001s
FAILED (failures=1)
Destroying test database for alias 'default'...
```

这其中都发生了些什么？：

- `python manage.py test polls`命令会查找所有投票应用中的测试程序
- 发现一个`django.test.TestCase`的子类
- 为测试创建一个专用的数据库
- 查找函数名以`test`开头的测试方法
- 在`test_was_published_recently_with_future_question`方法中，创建一个`Question`实例，该实例的`pub_data`字段的值是30天后的未来日期。
- 然后利用`assertIs()`方法，它发现`was_published_recently()`返回了`True`，而不是我们希望的`False`。

这个测试通知我们哪个测试失败了，错误出现在哪一行。

修复bug

我们已经知道了问题所在，现在可以去修复bug了。具体如下：

polls/models.py

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

再次运行测试程序：

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

更加全面的测试

我们可以使`was_published_recently()`方法更加可靠，事实上，在修复一个错误的同时又引入一个新的错误将是一件很令人尴尬的事。

下面，我们在同一个测试类中再额外添加两个其它的方法，来更加全面地进行测试：

polls/tests.py

```
def test_was_published_recently_with_old_question(self):
    """
    日期超过1天的将返回False。这里创建了一个30天前发布的实例。
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    最近一天内的将返回True。这里创建了一个1小时内发布的实例。
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertIs(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

返回正确的结果。

最后，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`Question.was_published_recently()`会按照预期的那样工作。

2.7.4 测试一个视图

这个投票应用没有辨别能力：它将会发布任何的`Question`，包括`pub_date`字段是未来的。我们应该改进这一点。让`pub_date`是将来时间的`Question`应该在将来发布，但是一直不可见，直到那个时间点才会变得可见。

在我们尝试修复任何事情之前，让我们先看一下可用的工具。

Django测试用客户端

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

先介绍使用`shell`的情况，这种方式下，需要做很多在`tests.py`中不必做的事。首先是设置测试环境：

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

`setup_test_environment()`会安装一个模板渲染器，它使我们可以检查一些额外的属性比如`response.context`，这些属性通常情况下是访问不到的。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`的不同而稍有不同。如果你当前`settings.py`中的`TIME_ZONE`不正确，那么你或许得不到预期的结果。在进行下一步之前，请确保时区设置正确。

下面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，不需要导入这个类）：

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

下面是具体的一些使用操作：

```
>>> # 从 '/' 获取响应
>>> response = client.get('/')
>>> # 这个地址应该返回的是404页面
>>> response.status_code
404
>>> # 另一方面我们希望在 '/polls/' 获取一些内容
>>> # 通过使用 'reverse()' 方法，而不是URL硬编码
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n <ul>\n \n <li><a href="/polls/1/">What&#39;s up?</a></li>\n \n </ul>\n\n'
>>> # 如果下面的操作没有正常执行，有可能是你前面忘了安装测试环境--setup_test_environment()
>>> response.context['latest_question_list']
<QuerySet [(<Question: What's up?>)]>
```

改进我们的视图

投票的列表会显示还没有发布的问卷（即`pub_date`在未来的问卷）。让我们来修复它。

在教程 4中，我们介绍了一个继承`ListView`的基类视图：

`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]
```

我们需要在`get_queryset()`方法中对比`timezone.now()`。首先导入`timezone`模块，然后修改

`get_queryset()`方法，如下：

`polls/views.py`

```
from django.utils import timezone

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]

```

`filter()`方法，确保了查询的结果是在当前时间之前，而不包含将来的日期。

测试新视图

对于没有测试概念的程序员，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，然后检验是否只有已经发布的Question才会展示出来，整个过程耗费大量的时间。对于有测试理念的程序员，不会每次修改与这相关的代码时都重复上述步骤，编写一测试程序是必然的。下面，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`：

首先导入`reverse`方法：

```
from django.urls import reverse
```

创建一个快捷函数来创建Question，同时创建一个新的测试类：

```

def create_question(question_text, days):
    """
    2个参数，一个是问卷的文本内容，另外一个是当前时间的偏移天数，负值表示发布日期在过去，正值表示发布日期在将来。
    """
    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):
        """
        如果问卷不存在，给出相应的提示。
        """
        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):
        """
        发布日期在过去的问卷将在index页面显示。
        """
        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):
        """
        发布日期在将来的问卷不会在index页面显示
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        即使同时存在过去和将来的问卷，也只有过去的问卷会被显示。
        """
        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):
        """

```

```

index页面可以同时显示多个问卷。
"""
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.>']
)

```

看一下具体的解释：

`create_question`是一个创建`Question`对象的函数。

`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，我们使用了`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`index`页面里不应该有任何`Question`。

诸如此类，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，检查系统的每一个状态变化，发布的是预期的结果。

测试 `DetailView` 视图

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`视图添加一个这样的约束：

`polls/views.py`

```

class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        return Question.objects.filter(pub_date__lte=timezone.now())

```

同样，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。

```

class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        访问发布时间在将来的detail页面将返回404.
        """
        future_question = create_question(question_text='Future question.', days=5)
        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        访问发布时间在过去的detail页面将返回详细问卷内容。
        """
        past_question = create_question(question_text='Past Question.', days=-5)
        url = reverse('polls:detail', args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)

```

更多的测试设计

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。这与我们上面的范例非常类似，实际上也有许多重复。

我们还可以在其它方面改进我们的应用，并随之不断地增加测试。例如，发布一个没有`Choices`的`Questions`就显得极不合理。所以，我们的视图应该检查这点并排除这些`Questions`。我们的测试会创建一个不带`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`并确保它会发布出来。

也许登陆的管理员用户应该被允许查看还没发布的`Questions`，但普通访问者则不行。最终要的是：无论添加什么代码来完成这个要求，都需要提供相应的测试代码，不管你是先编写测试程序然后让这些代码通过测试，还是先用代码解决其中的逻辑再编写测试程序来检验它。

从某种程度上来说，你一定会查看你的测试代码，然后想知道你的测试程序是否过于臃肿，我们接着看下面的内容：

2.7.5 测试越多越好

看起来我们的测试代码正在逐渐失去控制。以这样的速度，测试的代码量将很快超过我们的实际应用程序代码量，对比其它简洁优雅的代码，测试代码既重复又毫无美感。

没关系！随它去！大多数情况下，你可以完一个测试程序，然后忘了它。当你继续开发你的程序时，它将始终执行有效的测试功能。

有时，测试程序需要更新。假设我们让只有具有**Choices**的**Questions**才会发布，在这种情况下，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，使得它们保持最新，所以从某种程度上讲，测试可以自己测试自己。

在最坏的情况下，在你的开发过程中，你会发现许多测试变得多余。其实，这不是问题，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：

- 为每个模型或视图创建一个专属的**TestClass**
- 为你想测试的每一种情况建立一个单独的测试方法
- 为测试方法命名时最好从字面上能大概看出它们的功能

2.7.6 进一步测试

本教程只介绍了一些基本的测试。还有很多你可以做的工作，许多非常有用的工具可供你使用。

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，但你还可以使用一个“基于浏览器”的框架例如**Selenium**来测试你的**HTML**文件真实渲染的样子。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**JavaScript**的行为。它会启动一个浏览器，与你的网站进行交互，就像有一个人在操纵一样！**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，你可能为了实现持续集成，想在每次提交代码前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代码测试的覆盖率。这也有助于识别脆弱的甚至僵尸代码。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。覆盖率将帮助我们识别僵尸代码。查看3.9节《Testing in Django》来了解更多细节。

本节介绍了简单的测试方法。下一节我们将介绍静态文件。

2.8 第一个Django app, Part 6:静态文件

前面我们编写了一个经过测试的投票应用，现在让我们给它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，**WEB**应用一般需要提供一些其它的必要文件，比如图片文件、**JavaScript**脚本和**CSS**样式表等等，用来为用户呈现出一个完整的网页。在**Django**中，我们将这些文件称为“静态文件”。

对于小项目，这些都不是大问题，你可以将静态文件放在任何你的**web**服务器能够找到的地方。但是对于大型项目，尤其是那些包含多个**app**在内的项目，处理那些由**app**带来的多套不同的静态文件开始变得困难。

但这正是**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地方，并且这个地方在线上可以很容易维护。

2.8.1 自定义app的外观

首先在你的**polls**目录中创建一个**static**目录。**Django**将在那里查找静态文件，这与**Django**在**polls/templates/**中寻找对应的模板文件的方式是一致的。

Django的**STATICFILES_FINDERS**设置项中包含一个查找器列表，它们知道如何从各种源中找到静态文件。其中一个默认的查找器是**AppDirectoriesFinder**，它在每个**INSTALLED_APPS**下查找“**static**”子目录，例如我们刚创建的那个“**static**”目录。**admin**管理站点也为它的静态文件使用相同的目录结构。

在刚才的**static**中新建一个**polls**子目录，再在该子目录中创建一个**style.css**文件。换句话说，这个**css**样式文件应该是**polls/static/polls/style.css**。你可以通过书写**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，我们可以将静态文件直接放在**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匹配到的静态文件，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同名的另外一个子目录下（白话讲：多建一层与应用同名的子目录）。

译者：良好的目录结构是每个应用都应该创建自己的**urls**、**views**、**models**、**templates**和**static**，每个**templates**包含一个与应用同名的子目录，每个**static**也包含一个与应用同名的子目录。

将下面的代码写入样式文件：

polls/static/polls/style.css

```
li a {
    color: green;
}
```

接下来在模板文件的头部加入下面的代码：

polls/templates/polls/index.html

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

{% static %}模板标签会生成静态文件的绝对URL路径。
```

重新加载<http://localhost:8000/polls/>，你会看到Question的超链接变成了绿色（Django风格！），这意味着你的样式表被成功导入。

2.8.2 添加背景图片

下面，我们在polls/static/polls/目录下创建一个用于存放图片的images子目录，在这个子目录里放入background.gif文件。换句话说，这个文件的路径是polls/static/polls/images/background.gif。

修改你的css样式文件：

polls/static/polls/style.css

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

重新加载<http://localhost:8000/polls/>，你会在屏幕的右下方看到载入的背景图片。

警告：

显然，{% 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中路径相关的部分。

以上介绍的都是基础中的基础。更多的内容请查看4.15节《Managing static files》和6.5.12节《The staticfiles app》。4.16节《Deploying static files》讨论了更多关于如何在真实服务器上部署静态文件。

本节内容较少，下一节我们将介绍自定义Django的admin站点！

2.9 第一个Django app, Part 7: 自定义admin站点

本节我们主要介绍在第二部分简要提到过的Django自动生成的admin站点。

2.9.1 自定义admin表单

通过admin.site.register(Question)语句，我们在admin站点中注册了Question模型。Django会自动生成一个该模型的默认表单页面。如果你想自定义该页面的外观和工作方式，可以在注册对象的时候告诉Django你的选项。

下面是一个修改admin表单默认排序方式的例子：

首先修改admin.py的代码：

polls/admin.py

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

class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```

一般步骤是：创建一个模型管理类，将它作为第二个参数传递给admin.site.register()，随时随地修改模型的admin选项。

上面的修改，让“Publication date”字段显示在“Question”字段前面（默认是在后面）。如下图所示：

对于只有2个字段的情况，效果看起来还不是很明显，但是，如果，你有一打的字段，选择一种直观符合人类习惯的排序方式是一种重要的有用的细节处理。

同时，谈及包含大量字段的表单，你也许想将表单划分为一些字段集合。

polls/admin.py

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

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]
admin.site.register(Question, QuestionAdmin)
```

字段集合中每一个元组的第一个元素是该字段集合的标题。它让我们的页面看起来像下面的样子：

2.9.2 添加关系对象

好了，我们已经有了Question的admin页面，一个Question有多个Choices，但是我们还没有显示Choices的admin页面。有两个办法可以解决这个问题。第一个是像Question一样将Choice注册到admin站点，这很容易：

polls/admin.py

```
from django.contrib import admin
from .models import Choice, Question

# ...
admin.site.register(Choice)
```

现在访问admin页面，就可以看到Choice了，其“Add Choice”表单页面看起来如下图：

在这个表单中，Question字段是一个select选择框，包含了当前数据库中所有的Question实例。Django在admin站点中，自动地将所有的外键关系展示为一个select框。在我们的例子中，目前只有一个question对象存在。

请注意图中的绿色加号，它连接到Question模型。每一个包含外键关系的对象都会有这个绿色加号。点击它，会弹出一个新增Question的表单，类似Question自己的添加表单。填入相关信息点击保存后，Django自动将该Question保存在数据库，并作为当前Choice的关联外键对象。白话讲就是，新建一个Question并作为当前Choice的外键。

但是，实话说，这种创建方式的效率不怎么样。如果在创建Question对象的时候就可以直接添加一些Choice，那会更好。让我们来动手试试。

删除Choice模型对register()方法的调用。然后，编辑Question的注册代码如下：

polls/admin.py

```
from django.contrib import admin
from .models import Choice, Question

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

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

admin.site.register(Question, QuestionAdmin)
```

上面的代码告诉Django：Choice对象将在Question管理页面进行编辑，默认情况，请提供3个Choice对象的编辑区域。

加载“Add question”页面，应该看到如下图所示：

它的工作机制是：这里有3个插槽用于关联Choices，而且每当你重新返回一个已经存在的对象的“Change”页面，你又将获得3个新的额外的插槽可用。

在3个插槽的最后，还有一个“Add another Choice”链接。点击它，又可以获得一个新的插槽。如果你想删除新增的插槽，点击它右上方的X图标即可。但是，默认的三个插槽不可删除。下面是新增插槽的样子：

这里还有点小问题。上面页面中插槽纵队排列的方式需要占据大块的页面空间，查看起来很不方便。为此，

Django提供了一种扁平化的显示方式，你仅仅只需要修改一下ChoiceInline继承的类为admin.TabularInline替代先前的StackedInline：

polls/admin.py

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

刷新一下页面，你会看到类似表格的显示方式：

注意“DELETE”列，它可以删除那些已有的Choice和新建的Choice。

2.9.3 自定义admin change list

Question的admin页面我们已经修改得差不多了，下面让我们来微调一下“change list”页面，该页面显示了当前系统中所有的questions。

默认情况下，该页面看起来是这样的：

通常，Django只显示str()方法指定的内容。但是有时候，我们可能会想要同时显示一些别的内容。要实现这一目的，可以使用list_display属性，它是一个由字段组成的元组，其中的每一个字段都会按顺序显示在“change list”页面上，代码如下：

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):  
    # ...  
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

额外的，我们把was_published_recently()方法的结果也显示出来。现在，页面看起来会是下面的样子：

你可以点击每一列的标题，来根据这列的内容进行排序。但是，was_published_recently这一列除外，不支持这种根据函数输出结果进行排序的方式。同时请注意，was_published_recently这一列的列标题默认是方法的名字，内容则是输出的字符串表示形式。

可以通过给方法提供一些属性来改进输出的样式，就如下面所示：

polls/models.py

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

想要了解更多关于这些方法属性的信息，请参考6.5节《list_display》。

我们还可以对显示结果进行过滤，通过使用list_filter属性。在QuestionAdmin中添加下面的代码：

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

再次刷新change list页面，你会看到在页面右边多出了一个基于pub_date的过滤面板，如下图所示：

根据你选择的过滤条件的不同，Django会在面板中添加不容的过滤选项。由于pub_date是一个DateTimeField，因此，Django自动添加了这些选项：“Any date”，“Today”，“Past 7 days”，“This month”，“This year”。

顺理成章的，让我们添加一些搜索的能力：

```
search_fields = ['question_text']
```

这会在页面的顶部增加一个搜索框。当输入搜索关键字后，Django会在question_text字段内进行搜索。只要你愿意，你可以使用任意多个搜索字段，Django在后台使用的都是SQL查询语句的LIKE语法，但是，有限的搜索字段有助于后台的数据库查询效率。

也许你注意到了，页面还提供分页功能，默认每页显示100条。

2.9.4 定制admin外观

很明显，在每一个admin页面顶端都显示“Django administration”是很可笑的，它仅仅是个占位文本。利用Django的模板系统，很容易修改它。

定制你的项目模板

在`manage.py`文件同级下创建一个`templates`目录。打开你的设置文件`mysite/settings.py`，在`TEMPLATES`条目中添加一个`DIRS`选项：

`mysite/settings.py`

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

`DIRS`是一个文件系统目录的列表，是搜索路径。当加载Django模板时，会在`DIRS`中进行查找。

模板的组织方式：

就像静态文件一样，我们可以把所有的模板都放在一起，形成一个大大的模板文件夹，并且工作正常。但是我们不建议这样！我们建议每一个模板都应该存放在它所属应用的模板目录内（例如`polls/templates`）而不是整个项目的模板目录（`templates`），因为这样每个应用才可以被方便和正确的重用。请参考2.10节《如何重用apps》。

接下来，在刚才创建的`templates`中创建一个`admin`目录，将`admin/base_site.html`模板文件拷贝到该目录内。这个`html`文件来自Django源码，它位于`django/contrib/admin/templates`目录内。

Django的源代码在哪里？

如果你无法找到Django的源代码文件的存放位置，你可以使用下面的命令：

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

编辑该文件，用你喜欢的站点名字替换掉`{{ site_header|default:_(' Django administration') }}`（包括两个大括号一起），看起来像下面这样：

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

在这里，我们使用这个方法教会你如何重写模板。但是在实际的项目中，你可以使用`django.contrib.admin.AdminSite.site_header`属性（详见6.5节），方便的对这个页面`title`进行自定义。

请注意，所有Django默认的`admin`模板都可以被重写。类似刚才重写`base_site.html`模板的方法一样，从源代码目录将`html`文件拷贝至你自定义的目录内，然后修改文件。

定制你的应用模板

聪明的读者可能会问：但是`DIRS`默认是空的，Django是如何找到默认的`admin`模板呢？回答是，由于`APP_DIRS`被设置为`True`，Django将自动查找每一个应用包内的`templates/`子目录（不要忘了`django.contrib.admin`也是一个应用）。

我们的投票应用不太复杂，因此不需要自定义`admin`模板。但是如果它变得越来越复杂，因为某些功能而需要修改Django的标准`admin`模板，那么修改app的模板就比修改项目的模板更加明智。这样的话，你可以将投票应用加入到任何新的项目中，并且保证能够找到它所需要的自定义模板。

查看3.5节《[template loading documentation](#)》获取更多关于Django如何查找模板的信息。

2.9.5 定制admin首页

默认情况下，`admin`首页显示所有`INSTALLED_APPS`内并在`admin`应用中注册过的app，以字母顺序进行排序。

要定制`admin`首页，你需要重写`admin/index.html`模板，就像前面修改`base_site.html`模板的方法一样，从源代码目录拷贝到你指定的目录内。编辑该文件，你会看到文件内使用了一个`app_list`模板变量。该变量包含了所有已经安装的Django应用。你可以硬编码链接到指定对象的`admin`页面，使用任何你认为好的方法，用于替代这个`app_list`。

2.9.6 接下来学习什么？

至此，新手教程已经结束了。此时，你也许想看看2.11节的《下一步干什么》。

或者你对Python包机制很熟悉，对如何将投票应用转换成一个可重用的app感兴趣，请看2.10节《高级教程：

如何编写可重用的apps》。

2.10 高级教程：如何编写可重用的apps

2.10.1 重用的概念

The Python Package Index (PyPI)有大量的现成可用的Python库。 <https://www.djangopackages.com>

作为Django的app基地也有大量现成可用的apps。

包？App？

包是python重用代码的方式，以目录的形式体现，需要包含__init__.py文件，采用import的方式导入。app则是Django专用的包，包含一些通用的Django组件，例如models、tests、urls和views等子模块。

2.10.2 你的项目和可重用的app

通过前面的教程，你的项目结构如下：

```
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
```

它已经具备的project和app分离的条件。但是还需要一个打包的过程。

2.10.3 安装一些必要工具

使用setuptools和pip来打包我们的app。请先安装他们。

<https://pypi.python.org/pypi/setuptools>

<https://pypi.python.org/pypi/pip>

2.10.4 打包你的app

打包的意思是让你的app具有一种特殊的格式，使得它更容易被安装和使用。

1. 首先，在Django项目外面，为你的polls应用，准备一个父目录，取名django-polls；
为你的app选择一个合适的名字：
在取名前，去PYPI搜索一下是否有重名或冲突的包已经存在。建议给包名加上“django-”的前缀。名字中最后一个圆点的后面部分在INSTALLED_APPS中一定要独一无二，不能和任何Django的contrib packages中的重名，例如auth、admin、messages等等你。
2. 拷贝polls目录到该目录内；

3. 创建一个文件django-polls/README.rst，写入下面的内容：

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

```
-----
64 Chapter 2. Getting started
Django Documentation, Release 1.10.2a1
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.
```

1. 创建一个django-polls/LICENSE版权申明文件。大多数Django相关的app都基于BSD版权。
2. 接下来创建一个setup.py文件，用于编译和安装app。如何创建这个，请前往setuptools的官方文档获取详细的教程，本文不涉及。具体内容如下：

django-polls/setup.py

```
import os
from setuptools import find_packages, setup

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

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

setup(
    name='django-polls',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    license='BSD License', # example license
    description='A simple Django app to conduct Web-based polls.',
    long_description=README,
    url='https://www.example.com/',
    author='Your Name',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Framework :: Django :: X.Y', # replace "X.Y" as appropriate
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Replace these appropriately if you are stuck on Python 2.
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: 3.5',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

1. 默认只有python的模块和包会被打包进我们的app内。为了包含一些附加的文件，需要创建一个MANIFEST.in文件。为了将静态文件，模板等等非python语言编写的文件打包入内，我们需要在in文件内写入：

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

2. 该步骤可选，但是强烈推荐，将详细的说明文档一起打包。创建一个空的目录django-polls/docs，用于放置你的app相关文档。同时不要忘了，在django-polls/MANIFEST.in文件内写入一行recursive-include docs*。需要注意的是，如果docs目录是空的，那么它不会被打包进去。当然，许多apps通过在线的网站提供文档阅读。

3. 在你的django-polls目录内，运行python setup.py sdist命令。这将会创建一个dist目录，并编译成功你的新包，django-polls-0.1.tar.gz。

2.10.5 使用你自己的包

在安装包的时候，最好是以个人身份安装，而不是全系统范围的身份。这样可以有效减少给别的用户带去的影响或被别的用户影响。当然，最好的方式是在virtualenv环境下，类似隔离的沙盒环境。

1. 使用pip安装：`pip install --user django-polls/dist/django-polls-0.1.tar.gz`
2. 现在你可以在项目中使用这个包了
3. 卸载：`pip uninstall django-polls`

2.10.6 发布你的app

你可以：

- 通过邮件的形式发送给朋友
- 上传包到你的网站
- 推送到一个公开的仓库，例如PyPI，github等

<https://packaging.python.org/distributing/#uploading-your-project-to-pypi>是如何上传到PyPI的教程。

2.10.7 如何在virtualenv中安装python的包

前面，我们安装polls应用作为一个用户库，它有一些缺点：

- 修改用户库会影响到你系统上的其它Python软件
- 你无法同时运行此包的多个版本

解决这个问题最好的办法就是使用virtualenv。详见<https://virtualenv.pypa.io/en/stable/>

2.11 接下来学什么

本节主要介绍Django文档的划分，各部分的侧重点，如何找到自己感兴趣的内容。

由于此部分和文档最前面的目录导航重复较多，并且比较简单，就不翻译了。

2.11.1 在文档中查找

2.11.2 文档是如何组织的

2.11.3 文档是如何更新的

2.11.4 从哪里获取文档

2.11.5 不同版本之间的区别

2.12 编写你的第一个Django补丁

分类：[Django](#)

3.2.1 models模型

通常一个模型映射一张单独的数据表。

基本概念：

- 每个model都是django.db.models.Model的子类
- model的每个属性代表数据表的某一行
- Django将自动为你生成数据库访问API

3.2.1.1 快速展示：

下面的模型定义了一个“人”，它具有first_name和last_name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)
```

上面的代码，相当于下面的原生sql语句：

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

注意：

- 表名myapp_person由Django自动生成：项目名称+下划线+小写类名。你可以重写这部分功能的机制。
- Django自动创建id自增主键，当然，你也可以自己指定主键，
- 上面的sql语句基于PostgreSQL 语法，可能与你的实际情况有差别。

3.2.1.2 使用模型

创建了model之后，在使用它之前，你需要先在settings文件中的INSTALLED_APPS 处，注册models.py文件所在的APP。例如：

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

当你每次在INSTALLED_APPS处增加新的APP时，请务必执行命令python manage.py migrate。有可能要先make migrations。

3.2.1.3 Fields字段

model中最重要也是必须定义的部分。请不要使用clean、save、delete等model API内置的名字，防止命名冲突。

范例：

```
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, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Fields types 字段类型

fields类型的作用:

- 决定数据库中对应列的数据类型
- HTML中对应的form field类型, 例如- 在admin后台和自动生成的form表单中最小的数据验证需求
 - 3.22节《Making queries》
 - 6.15节《model field reference》《Field API reference》
 - 4.3节《Writing custom model fields.》

AutoField

一个自动增加的整数类型。通常你不需要使用它, Django自动帮你添加下面的字段:

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

BigAutoField

(1.10新增)

64位整数, 类似AutoField。1 to 9223372036854775807

BigIntegerField

64位整数, 类似IntegerField, -9223372036854775808 to 9223372036854775807。

在默认的form类型是textinput (用django的form功能自动生成的input标签)。

BinaryField

二进制数据类型。使用受限, 慎用。

BooleanField

布尔值类型。默认值是None。

默认的form类型是input checkbox。

如果要接收null值, 请使用NullBooleanField。

CharField

字符串类型。必须接收一个max_length参数。

默认的form类型是input text。

最常用的field!

CommaSeparatedIntegerField

逗号分隔的整数类型。必须接收一个max_length参数。

常用于表示较大的金额数目, 例如1,000,000元。

DateField

```
class DateField(auto_now=False, auto_now_add=False, **options)
```

日期类型。

一个Python中的datetime.date的实例。

在form中的默认类型是text。在admin后台, Django会帮你自动添加一个js的日历表和一个“Today”快捷方式, 以及附加的日期合法性验证。

参数: (所有参数互斥, 不能共存)

auto_now: 每当对象被保存时将字段设为当前日期, 常用于保存最后修改时间。

注意, 只有在使用save()方法时才更新, 其它操作不更新。

auto_now_add: 每当对象被创建时, 设为当前日期, 常用于保存创建日期。

注意, 它是不可修改的。

设置上面两个参数就相当于给field添加了editable=False and blank=True属性。如果想具有修改属性, 请用

default参数:

对于 DateField: default=date.today - from datetime.date.today()

对于 DateTimeField: default=timezone.now - from django.utils.timezone.now()

DateTimeField

class DateTimeField(auto_now=False, auto_now_add=False, **options)

日期时间类型。

Python的datetime.datetime的实例。与DateField相比就是多了小时、分和秒的显示，其它功能、参数、用法、默认值等等都一样。

DecimalField

class DecimalField(max_digits=None, decimal_places=None, **options)

固定精度的十进制小数。

相当于Python的Decimal实例，必须提供两个指定的参数！

参数:

max_digits: 最大的位数，必须大于或等于小数点位数

decimal_places: 小数点位数，精度。

范例：储存最大不超过999，带有2位小数位精度的数，定义如下：

models.DecimalField(..., max_digits=5, decimal_places=2)

5来自3+2！

当 localize=False，它在form中默认为NumberInput 类型。否则，是text类型。

DurationField

持续时间类型

存储一定期间的长度。类似python中的timedelta。在不同的数据库实现中有不同的表示方法。常用于进行时间之间的加减运算。但是小心了，这里有坑，PostgreSQL等数据库之间有兼容性问题！

EmailField

class EmailField(max_length=254, **options)

邮箱类型。

使用EmailValidator进行合法性验证。

FileField

class FileField(upload_to=None, max_length=100, **options)

上传文件类型

[filed类型表](#)

[自定义filed类型](#)

Field options选项

[model field reference](#)

字段选项

详细看6.15节《field option》

除了类似max_length是对CharFiled的必须参数外。还有一些是可选的常用参数：

- **null:** True时，Django在数据库用NULL保存空值。默认False。
- **blank:** true时，字段可以为空。默认false。和null不同的是，null是纯数据库层面的，而blank是验证相关的，它与表单验证是否允许输入框内为空有关，于数据库无关。所以要小心一个null为false，blank为true的字段接收到一个空值可能会出bug或异常。
- **choices:** 用于选择框，需要先提供一个二维的元组，第一个元素表示存在数据库内真实的值，第二个表示页面上显示的具体内容。例如：

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

要显示一个choices的值，可以使用get_FOO_display()方法，其中的FOO用字段名代替。

```
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)
```

使用方法：

```
>>> 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，可以是值或者一个回调对象。每次创建新对象时该回调都会被执行。注意：在某种原因不明的情况下将default设置为NONE，可能会引发IntegrityError: not null constraint failed，即非空约束失败异常，导致python manage.py migrate失败，此时可将None改为False或其它的值，只要不是None就行。
- **help_text:** 额外的帮助文本显示在表单部件上。

- **primary_key:** 主键。设置为True时，当前字段变为主键，并关闭Django自动生成的id主键功能。另外，主键字段不可修改，如果你赋个新值则会创建个新记录。

```
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: true时，在整个表内该字段的数据不可重复。

自动主键字段

默认情况下，Django给你提供了自动的主键：

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

字段详细名称

除了外键、多对多和一对一字段外，所有的字段都可以有一个可选的第一位置参数：**verbose name**。如果没给这个参数，Django会利用字段的属性名自动创建它，并将下划线转换为空格。

下面这个例子verbose name是"person's first name"：

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

下面这个例子verbose name是"first name"：

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

对于外键、多对多和一对一字段，由于第一个参数需要用来指定模型类，因此必须用关键字参数**verbose_name**来指定详细名。如下：

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

```

        on_delete=models.CASCADE,
        verbose_name="related place",
    )

```

关系relationships

1. **多对一**：也就是外键，使用`django.db.models.ForeignKey`。需要第一位置参数 为相关联的模型类。例如：

```

from django.db import models
class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...

```

外键关系字段放在多的一方，比如上面的多种汽车出自同一厂商。

也可以创建递归关系（自己多对一关联自己，使用`self`作为指向的模型名），或者关联到尚未创建的模型。

建议将外键字段名设置为关联类的小写名称。非强制。

更多`ForeignKey`相关，请查看6.15节《the model field reference》

1. **多对多**：`ManyToManyField`。需要一个位置参数，指向关联的模型。例如：

```

from django.db import models

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

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

```

同对一对一样，也可以建立递归的自己关联自己的多对多，或关联到一个尚未定义的模型。

建议将字段名取成关联模型的小写复数，例如`toppings`。

多对多字段放在关联双方的任何一方都可以，但是只能在一方，不能同时。但通常会考虑现实的逻辑，将其放在更符合基本情况的一方。

更多`ManyToManyField`相关，请查看6.15节《the model field reference》

1. **多对多的额外字段**

一般情况，普通的多对多已经够用，无需自己创建第三张关系表。但是某些情况可能更复杂一点，比如有音乐家表和音乐家分组表，这是个多对多的关系，但是如果你想保存某个音乐家加入某个音乐家分组的时间呢？

`Django`提供了一个`through`参数，用于指定中间模型，你可以将类似加入时间等其他字段放在这个中间模型内。例子如下：

```

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, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)

```

对于中间模型，有一些限制：

- 中间模型只能包含一个指向源模型的外键关系，上面例子中，也就是在`Membership`中只能有`person`和`group`外键关系各一个，不能多。否则，你必须显式的通过`ManyToManyField.through_fields`指定关联的对象。参考下面的例子：

```

from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):

```



```

name = models.CharField(max_length=128)
members = models.ManyToManyField(
    Person,
    through='Membership',
    through_fields=('group', 'person'),
)

class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    invite_reason = models.CharField(max_length=64)

```

- 对于一个通过中间模型关联自身的多对多，两个同样的外键关系是允许的，但此时他们被看做不同的两边。但是，如果多于2个，一样要如同上面显式的通过`ManyToManyField.through_fields`指定关联的对象。
- 当你通过中间模型创建一个关联自身的多对多，你必须显式的指出`symmetrical=False`。这样，Django才会关闭默认的对等特性。具体参考6.15

下面是一些使用例子：

```

>>> 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()
<QuerySet [Person: Ringo Starr]>
>>> ringo.group_set.all()
<QuerySet [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()
<QuerySet [Person: Ringo Starr, Person: Paul McCartney]>

```

与普通的多对多不一样，使用中间模型的多对多不能使用`add()`、`create()`、`remove()`、和`set()`方法来创建、删除关系，看下面：

```

>>> # 无效
>>> beatles.members.add(john)
>>> # 无效
>>> beatles.members.create(name="George Harrison")
>>> # 无效
>>> beatles.members.set([john, paul, ringo, george])

```

为什么？因为上面的方法无法提供加入时间、邀请原因等中间模型需要的字段内容。唯一的办法只能是通过创建中间模型的实例来创建这种类型的多对多关联。

但是`clear()`方法是有效的，它能清空所有的多对多关系。

```

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

```

一旦你通过创建中间模型实例的方法建立了多对多的关联，你立刻就可以像普通的多对多那样进行查询操作：

```

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

```

可以使用中间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))
<QuerySet [Person: Ringo Starr]>

```

可以像普通模型一样使用中间模型：

```
>>> 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.'
>>> 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.'
```

1. 一对一关系: OneToOneField, 同样需要关联模型作为第一位置参数。

举例, 你设计一个“场所”的模型, 包括地址、电话等等。然后, 你又想创建一个基于“场所”的餐馆模型, 你不需要重复上面的场所的字段, 只需要在餐馆模型中建立一个OneToOneField关联到“场所”模型。(事实上, 通常会使用继承的方法, 它隐含了一个一对一关系)。

同样, 一对一也可以递归关联自己, 或关联未定义模型。

一对一还有一个可选的parent_link参数。

其它模块的Models

直接在文件顶部导入其它模块内的模型, 然后正常使用!

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

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        ZipCode,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

field name的命名限制

- 不能是Python的保留关键字
- 不能包含2个及以上的下划线。因为2个下划线在Django查询语法中有特殊作用。

但是SQL的保留字, 如join、where和select是可以用的。

自定义字段类型

参考4.3《Writing custom model fields》

3.2.1.4 meta 选项

方法: 在模型内部创建class Meta

```
from django.db import models

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

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

metadata:指的是“任何非字段相关的内容”, 例如排序依据 (ordering), 表名 (db_table), 或者人类可读的单数、复数名 ((verbose_name and verbose_name_plural))。所有的都是可选, 非必须的。完整的参考表见6.15.6《model option reference》

3.2.1.5 模型属性

objects: 对于一个模型, 最重要的属性是Manager (管理器)。它是模型用来查询、操作数据的接口。如果没有自定义Manager, 那么它的默认名字就是“objects”。它只能通过类名进行访问, 不能通过类的实例访问。

3.2.1.6 模型方法

下面是在模型中自定义方法的实例:

```
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) # 将方法转换为属性

```

下面是2个常用的经常被定义的方法：

str():python3版本，在打印模型时，指定显示的内容。

get_absolute_url(): Django用它来获取对象的URL，每一个包含URL的对象都必须定义这个方法。参考6.15.

重写内置的模型方法

例如，你想在**save()**方法执行前后先干点什么：

```

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()
        # 调用内置的save () 方法
        super(Blog, self).save(*args, **kwargs)
        do_something_else()

```

或者阻止某些人无法进行**save()**。

```

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":
            # 叫这个名字的博客无法保存
            return
        else:
            # 其它的正常保存
            super(Blog, self).save(*args, **kwargs)

```

注意三点：

1. 一定要调用父类方法**super(Blog, self).save(*args, **kwargs)**保证正常的工作
2. 一定要用***args,**kwargs**的传参方式，保证无论如何，参数都被正确的传递给父类方法。
3. 在进行批量处理时，自定义方法可能不管用。

3.2.1.7 模型的继承

所有的模型类必须继承**django.db.models.Model**。

Django有三种继承的方式：

- 抽象类：被用来继承的类，**Abstract base classes**，将共同的数据抽离出来，供子类继承重用，它不会创建表
- 多表类：**Multi-table inheritance**，每一个模型都有自己的数据库表。
- 代理模型：**Proxy models**

继承抽象基类：

值需要在**Meta**类里添加**abstract=True**，就可以将一个模型转换为抽象基类。Django不会为这种类创建实际的

数据库表，他们也没有管理器，不能被实例化也无法直接保存，它们就是用来被继承的。例如：

```
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name, age, home_group三个字段。基类和子类不能有一样的字段名。

Meta的继承：

如果子类没有声明自己的Meta类，那么它将继承抽象基类的Meta类。下面的例子则扩展了基类的Meta：

```
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'
```

抽象基类也可以继承别的抽象基类，但不要忘记在Meta类里添加abstract=True。这个选项才是决定一个类是普通的类还是抽象基类的根本。

区别related_name和related_query_name，当你在抽象基类中使用时应该包含'%(app_label)s' 和'%(class)s'：

- '%(class)s' 用字段所属子类的小写名替换
- '%(app_label)s' 用子类所属app的小写名替换

例如，对于common/models.py模块：

```
from django.db import models

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

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

对于另外一个模块rare/models.py：

```
from common.models import Base

class ChildB(Base):
    pass
```

对于上面的继承关系：

- common.ChildA.m2m字段的reverse name（反向关系名）应该是common_childa_related；reverse query name(反向查询名)应该是common_childas。
- common.ChildB.m2m字段的reverse name（反向关系名）应该是common_childb_related；reverse query name(反向查询名)应该是common_childbs。
- rare.ChildB.m2m字段的reverse name（反向关系名）应该是rare_childb_related；reverse query name(反向查询名)应该是rare_childbs。

具体时候什么名字，取决你如何通过'%(class)s' 'and' %(app_label)s构造名称字符串。但是，如果你

忘了这个技术细节，那么在使用时会弹出异常。

multi-table inheritance多表继承

这种继承方式，父类和子类都有自己的数据库表，内部隐含了一个一对一的关系。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)
```

Restaurant类将包含Place类的字段，并且各有各的数据库表和字段，比如：

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

如果一个Place同时也是一个Restaurant，你可以使用小写的子类名，在父类中访问它，例如：

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

但是，如果这个Place是个纯粹的Place，并不是一个Restaurant，那么上面的调用方式会报错Restaurant.DoesNotExist。

Meta和多表继承

在多表继承的情况下，由于父类和子类都在数据库内有物理存在的表，父类的Meta类会对子类造成不好的影响，因此，Django在这种情况下关闭了子类继承父类的Meta功能。这一点和抽象基类的继承方式有所不同。

但是，还有2个meta元素特殊一点，那就是ordering和get_latest_by，这两个参数是会被继承的。因此，如果在多表继承中，你不想让你的子类继承父类的上面两种参数，就必须在子类中显示的指出或重写。如下：

```
class ChildModel(ParentModel):
    # ...

class Meta:
    # 移除父类对子类的排序影响
    ordering = []
```

继承和反向关联

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。如果你与该父类的另一个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就会在你运行或验证(validation)时抛出异常。

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)
```

这会产生下面的错误：

```
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'.
```

解决方法是：向customers字段中添加related_name: models.ManyToManyField(Place, related_name='provider')。

指定链接父类的字段

之前提到，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。

Proxy models代理模型

使用多表继承时，父类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。比如：

更改默认的`manager`，或是添加一个新方法。

代理模型可以实现这一点：为原始模型创建一个代理。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像使用原始`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`，而不会对原始`model`产生影响。

声明代理模型和声明普通模型没有什么不同。设置`Meta`类中`proxy`的值为`True`，就完成了对代理模型的声明。

举例，假设你想给`Person`模型添加一个方法。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`类将操作和`Person`类一样的数据库表。并且，任何新的`Person`实例都可以通过`MyPerson`类进行访问，反之亦然。

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

下面是通过代理实现排序，但原父类不排序的方法：

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

现在，普通的`Person`查询是无序的，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：它会使用依赖于原生`Person`的代码，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。并且不能继承自多个非抽象基类，这是因为一个代理模型不能连接不同的数据表。
- 代理模型可以同时继承任意多个抽象基类，前提是这些抽象基类没有定义任何模型字段。
- 代理模型可以同时继承多个别的代理模型，前提是这些代理模型继承同一个非抽象基类。（早期`Django`版本不支持这一条）

代理模型管理器

如不指定，则继承父类的管理器。如果你自己定义了管理器，那它就会成为默认管理器，但是父类的管理器依然有效。如下例子：

```
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，并继承时把他放在主基类的后面：

```
# 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

```

代理继承与非托管模型之间的差异

代理继承看上去和使用Meta类中的managed属性的非托管模型非常相似。

在创建非托管模型时要谨慎设置Meta.db_table，这是因为创建的非托管模型映射某个已存在的模型，并且有自己的方法。如果你要保证这两个模型同步并对程序进行改动，那么就会变得繁冗而脆弱。

一般规则是：

- 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，那就使用Meta.managed=False。通常情况下，对模型数据库创建视图和表格不需要由Django控制时，就使用这个选项。
- 如果你想对模型做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，代理模型相当于完全复制了原始模型的存储结构。

多重继承

注意，多重继承和多表继承是两码事，两个概念。

Django的模型体系支持多重继承，就像Python一样。同时，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，这意味着如果多个父类都含有Meta类，只有第一个父类的会被使用，剩下的会忽略掉。

通常，你不需要使用多重继承。最常用的情况是“混入”（mix-in）：为每一个继承了mix-in的类添加一个特别额外的字段或方法。

但是，尽量让你的继承关系简单和直接，避免不必要的混乱和复杂。

请注意，继承同时含有相同id主键域的类将抛出异常。为了解决这个问题，你可以在基类模型中显式的使用AutoField字段。如下例子所示：

```

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

```

或者使用一个共同的祖先来持有AutoField字段，如下所示：

```

class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
    ...

class BookReview(Book, Article):
    pass

```

重写字段

在python中，子类可以重写所有的父类属性。但在Django中却不一定。如果一个非抽象模型基类有一个叫做author的字段，那么在它的子类中，你不能创建任何也叫做author的模型字段或属性（这个限制对抽象模型无效）。这些字段有可能被另外的字段或值重写，或通过设置filed_name=None而被移除。

警告：模型管理器继承自抽象基类。重写一个被继承管理器引用的继承字段可能导致小bug。参考3.2《Custom managers and model inheritance》

注意：一些字段会在模型上定义一些额外的属性，例如ForeignKey会定义一个额外的属性（将"_id"附加在字段名上）。就像related_name和related_query_name一样。这些额外的属性不可被重写，除非定义他们的字段被改变或移除了。

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。

3.2.1.8 在包中组织模型

manage.py startapp命令的执行会创建一个app的文件结构，并包含一个models.py文件。但是，如果你有很多模型，那么将它们分隔放在不同的文件中会是个好主意。

想这么做，只需要创建一个模型包。首先，移除models.py文件，再建立一个myapp/models/目录，在目录里创建个__init__.py文件，最后创建存放模型的文件。你必须在__init__.py中导入那些模型models。

例如，如果你有一个organic.py和synthetic.py文件在models目录里，那么，init.py文件里应该这么写代码：

myapp/models/init.py

```
from .organic import Person
from .synthetic import Robot
```

在文件中显式的导入每一个模型比使用from .models import * 这种一股脑的导入方式更好，这样不会导致命名空间的混乱，让代码更可读，更利于代码分析工具进行检查。

更多请查看6.15节的Models，这里包含了所有的API和细节，很长很长的文档.....

3.2.2 查询操作

6.15章节包含所有模型相关的API解释。

后面的内容基于如下的一个博客应用模型：

```
from django.db import models

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

```

tagline = models.TextField()

def __str__(self):
    return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

def __str__(self):
    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):
    return self.headline

```

3.2.2.1 创建对象

如果模型存在mysite/blog/models.py文件中，那么创建对象的方式如下：

```

>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()

```

在后台，这会运行一条SQL的INSERT语句。如果你不显示的调用save()方法，Django不会立刻将该操作反映到数据库中。save()方法没有返回值，它可以接受一些额外的参数。

如果想要一行代码完成上面的操作，请使用creat()方法，它可以省略save的步骤：

```

b = Blog.objects.create(name='Beatles Blog', tagline='All the latest Beatles news.')

```

3.2.2.2 保存对象

使用save()方法，保存对数据库内已有对象的修改。例如：如果已经存在b5对象在数据库内：

```

>>> b5.name = 'New name'
>>> b5.save()

```

在后台，这会运行一条SQL的UPDATE语句。如果你不显示的调用save()方法，Django不会立刻将该操作反映到数据库中。

保存外键和多对多字段

保存一个外键字段和保存普通字段没什么区别，只是要注意值的类型要正确。下面的例子，有一个Entry的实例entry和一个Blog的实例cheese_blog，然后把cheese_blog作为值赋给了entry的blog属性，最后调用save方法进行保存。

```

>>> 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()

```

多对多字段的保存稍微有点区别，需要调用一个add()方法，而不是直接给属性赋值，但它不需要调用save方法。如下例所示：

```

>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)

```

在一行语句内，同时添加多个对象到多对多的字段，如下所示：

```

>>> 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)

```

如果你指定或添加了错误类型的对象，Django会抛出异常。

3.2.2.3 检索对象

想要从数据库内检索对象，你需要基于模型类，通过管理器（Manager）构造一个查询结果集（QuerySet）。

每个QuerySet代表一些数据库对象的集合。它可以包含零个、一个或多个过滤器（filters）。Filters缩小查询结果的范围。在SQL语法中，一个QuerySet相当于一个SELECT语句，而filter则相当于WHERE或者LIMIT一类的子句。

通过模型的Manager获得QuerySet，每个模型至少具有一个Manager，默认情况下，它被称作“objects”，可以通过模型类直接调用它，但不能通过模型类的实例调用它，以此实现“表级别”操作和“记录级别”操作的强制分离。如下所示：

```
>>> 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."
```

检索所有对象

使用all()方法，可以获取某张表的所有记录。

```
>>> all_entries = Entry.objects.all()
```

过滤器检索对象

有2个方法可以用来过滤QuerySet的结果，分别是：

filter(kwargs): 返回一个根据指定参数查询出来的QuerySet

exclude(kwargs): 返回除了根据指定参数查询出来结果的QuerySet

其中，**kwargs参数的格式必须是Django设置的一些字段格式。

例如：

```
Entry.objects.filter(pub_date__year=2006)
```

它等同于：

```
Entry.objects.all().filter(pub_date__year=2006)
```

链式过滤

filter和exclude的结果依然是个QuerySet，因此它可以继续被filter和exclude，这就形成了链式过滤：

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 30)
... )
```

（译者：这里需要注意的是，当在进行跨关系的链式过滤时，结果可能和你想象的不一样，参考下面的跨多值关系查询）

被过滤的QuerySets都是唯一的

每一次过滤，你都会获得一个全新的QuerySet，它和之前的QuerySet没有任何关系，可以完全独立的被保存，使用和重用。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())
```

例子中的q2和q3虽然由q1得来，是q1的子集，但是都是独立自主存在的。同样q1也不会受到q2和q3的影响。

QuerySets都是懒惰的

一个创建QuerySets的动作不会立刻导致任何的数据库行为。你可以堆栈所有的filter动作一整天，Django不会运行任何实际的数据库查询动作，直到QuerySets被提交(evaluated)。

简而言之就是，只有碰到某些特定的操作，Django才会将所有的操作体现到数据库内，否则它们只是保存在内存和Django的层面中。这是一种提高数据库查询效率，减少操作次数的优化设计。看下面的例子：

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

上面的例子，看起来执行了3次数据库访问，实际上只是在`print`语句时执行了1次访问。通常情况，`QuerySets`的检索不会立刻执行实际的数据库查询操作，直到出现类似`print`的请求，也就是所谓的`evaluated`。具有有哪些操作会触发`evaluated`，请查看6.15.8章节。

检索单一对象

`filter`方法始终返回的是`QuerySets`，那怕只有一个对象符合过滤条件，返回的也是包含一个对象的`QuerySets`。

如果你确定你的检索只会获得一个对象，那么你可以使用`Manager`的`get()`方法来直接返回这个对象。

```
>>> one_entry = Entry.objects.get(pk=1)
```

在`get`方法中你可以使用任何`filter`方法的查询参数，一模一样。

注意：使用`get()`方法和使用`filter()`方法然后通过`[0]`的方式分片，有着不同的地方。看似两者都是获取单一对象。但是，如果在查询时没有匹配到对象，那么`get()`方法将抛出`DoesNotExist`异常。这个异常是模型类的一个属性，在上面的例子中，如果不存在主键为1的`Entry`对象，那么Django将抛出`Entry.DoesNotExist`异常。

类似地，在使用`get()`方法查询时，如果结果超过1个，则会抛出`MultipleObjectsReturned`异常，这个异常也是模型类的一个属性。

其它QuerySet方法

查看6.15.8获取更多不同的`QuerySet`方法相关信息

QuerySet使用限制

使用类似Python对列表进行切片的方法可以对`QuerySet`进行范围取值。它相当于SQL语句中的`LIMIT`和`OFFSET`子句。参考下面的例子：

```
>>> Entry.objects.all()[:5]      # 返回前5个对象
>>> Entry.objects.all()[5:10]    # 返回第6个到第10个对象
```

注意：不支持负索引！例如 `Entry.objects.all()[-1]`是不允许的

通常情况，切片操作会返回一个新的`QuerySet`，并且不会被立刻执行。但是有一个例外，那就是指定步长的时候，查询操作会立刻在数据库内执行，如下：

```
>>> Entry.objects.all()[0:10:2]
```

若要获取单一的对象而不是一个列表（例如，`SELECT foo FROM bar LIMIT 1`），可以简单地使用索引而不是切片。例如，下面的语句返回数据库中根据标题排序后的第一条`Entry`：

```
>>> Entry.objects.order_by('headline')[0]
```

它相当于：

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

注意：如果没有匹配到对象，那么第一种方法会抛出`IndexError`异常，而第二种方式会抛出`DoesNotExist`异常。

也就是说在使用`get`和切片的时候，要注意查询结果的元素个数。

字段查询

字段查询其实就是`filter()`、`exclude()`和`get()`方法的关键字参数。

其基本格式是：`field__lookuptype=value`，注意其中是双下划线。

例如：

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
# 相当于：
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

其中的`field`必须是模型中定义的`field`之一。但是有一个例外，那就是`ForeignKey`字段，你可以为其添加一个“`_id`”后缀（单下划线）。这种情况下键值是外键模型的主键原生值。例如：

```
>>> Entry.objects.filter(blog_id=4)
```

如果你传递了一个非法的键值，查询函数会抛出`TypeError`异常。

Django的数据库API支持20多种查询类型，完整的参考手册在6.15节，下面介绍一些常用的：

exact:

默认类型。如果你不提供查询类型，或者关键字参数不包含一个双下划线，那么查询类型就是这个默认的`exact`。

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
# 相当于
# SELECT ... WHERE headline = 'Cat bites dog';
# 下面两个相当
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14)        # __exact is implied
```

icontains:

不区分大小写。

```
>>> Blog.objects.get(name__icontains="beatles blog")
# 匹配"Beatles Blog", "beatles blog",甚至"BeAtLEs bLoG".
```

contains:

表示包含的意思！大小写敏感！

```
Entry.objects.get(headline__contains='Lennon')
# 相当于
# SELECT ... WHERE headline LIKE '%Lennon%';
# 匹配'Today Lennon honored', 但不匹配'today lennon honored'
```

icontains:

`contains`的大小写不敏感模式。

startswith和endswith

以什么开头和以什么结尾。大小写敏感！

`istartswith`和`iendswith`是不区分大小写的模式。

跨越关系查询

Django提供了强大并且直观的方式解决跨越关联的查询，它在后台自动执行包含`JOIN`的SQL语句。要跨越某个关联，只需使用关联的模型字段名称，并使用双下划线分隔，直至你想要的字段（可以链式跨越，无限跨度）。例如：

```
# 返回所有Blog的name为'Beatles Blog'的Entry对象
# 一定要注意，返回的是Entry对象，而不是Blog对象。
# objects前面用的是哪个class，返回的就是哪个class的对象。
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

反之亦然，如果要引用一个反向关联，只需要使用模型的小写名！

```
# 获取所有的Blog对象，前提是它所关联的Entry的headline包含'Lennon'
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

如果你在多级关联中进行过滤而且其中某个中间模型没有满足过滤条件的值，Django将把它当做一个空的（所有的值都为`NULL`）但是合法的对象，不会抛出任何异常或错误。例如，在下面的过滤器中：

```
Blog.objects.filter(entry__authors__name='Lennon')
```

如果`Entry`中没有关联任何的`author`，那么它将当作其没有`name`，而不会因为缺少`author`引发一个错误。通常，这是比较符合逻辑的处理方式。唯一可能让你困惑的是当你使用`isnull`的时候：

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

这将返回`Blog`对象，它关联的`entry`对象的`author`字段的`name`字段为空，以及`Entry`对象的`author`字段为空。如果你不需要后者，你可以这样写：

```
Blog.objects.filter(entry__authors__isnull=False,entry__authors__name__isnull=True)
```

跨越多值关系查询

最基本的`filter`和`exclude`的关键字参数只有一个，这种情况很好理解。但是当关键字参数有多个，且是跨越外键或者多对多的情况下，那么就比较复杂，让人迷惑了。我们看下面的例子：


```
Blog.objects.filter(entry__headline__contains='Lennon', entry__pub_date__year=2008)
```

这是一个跨外键的，两个过滤参数的查询。此时我们理解两个参数之间属于-与“and”的关系，也就是说，过滤出来的Blog对象对应的entry对象必须同时满足上面两个条件。这点很好理解。但是，看下面的用法：

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__pub_date__year=2008)
```

把两个参数拆开，放在两个filter调用里面，按照我们前面说过的链式过滤，这个结果应该和上面的例子一样。可实际上，它不一样，Django在这种情况下，将两个filter之间的关系设计为-或“or”，这真是让人头疼。

多对多关系下的多值查询和外键foreignkey的情况一样。

但是，更头疼的来了，exclude的策略设计的又和filter不一样！

```
Blog.objects.exclude(entry__headline__contains='Lennon', entry__pub_date__year=2008,)
```

这会排除headline中包含“Lennon”的Entry和在2008年发布的Entry，中间是一个-和“or”的关系！

那么要排除同时满足上面两个条件的对象，该怎么办呢？看下面：

```
Blog.objects.exclude(
    entry=Entry.objects.filter(
        headline__contains='Lennon',
        pub_date__year=2008,
    ),
)
```

（译者：有没有很坑爹的感觉？所以，建议在碰到跨关系的多值查询时，尽量使用Q查询）

使用F表达式引用模型的字段

到目前为止的例子中，我们都是将模型字段与常量进行比较。但是，如果你想将模型的一个字段与同一个模型的另外一个字段进行比较该怎么办？

使用Django提供的F表达式！

例如，为了查找comments数目多于pingbacks数目的Entry，可以构造一个F()对象来引用pingback数目，并在查询中使用该F()对象：

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django支持对F()对象进行加、减、乘、除、取模以及幂运算等算术操作。两个操作数可以是常数和和其它F()对象。例如查找comments数目比pingbacks两倍还要多的Entry，我们可以这么写：

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```

为了查询rating比pingback和comment数目总和要小的Entry，我们可以这么写：

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

你还可以在F()中使用双下划线来进行跨表查询。例如，查询author的名字与blog名字相同的Entry：

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

对于date和date/time字段，还可以加或减去一个timedelta对象。下面的例子将返回发布时间超过3天后被修改的所有Entry：

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

F()对象还支持.bitand()和.bitor()两种位操作，例如：

```
>>> F('somefield').bitand(16)
```

主键的快捷查询方式：pk

pk就是primary key的缩写。通常情况下，一个模型的主键为“id”，所以下面三个语句的效果一样：

```
>>> 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
```

可以联合其他类型的参数：

```
# 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)
```

可以跨表操作：

```
>>> Entry.objects.filter(blog__id__exact=3)
>>> Entry.objects.filter(blog__id=3)
>>> Entry.objects.filter(blog__pk=3)
```

（译者：当主键不是id的时候，请注意了！）

自动转义百分符号和下划线

在原生SQL语句中%符号有特殊的作用。Django帮你自动的转义了百分符号和下划线，你可以和普通字符一样使用它们，如下所示：

```
>>> Entry.objects.filter(headline__contains='%')
# 它和下面的一样
# SELECT ... WHERE headline LIKE '%\%%';
```

缓存和查询集合

每个QuerySet都包含一个缓存，用于减少对数据库的实际操作。

对于新创建的QuerySet，它的缓存是空的。当QuerySet第一次被提交后，数据库执行实际的查询操作，Django会把查询的结果保存在QuerySet的缓存内，随后的对于该QuerySet的提交将重用这个缓存的数据。

要想高效的利用查询结果，降低数据库负载，你必须善于利用缓存。看下面的例子，这会造成2次实际的数据库操作，加倍数据库的负载，同时由于时间差的问题，可能在两次操作之间数据被删除或修改或添加，导致脏数据的问题：

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

为了避免上面的问题，好的使用方式如下，这只产生一次实际的查询操作，并且保持了数据的一致性：

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # 提交查询
>>> print([p.pub_date for p in queryset]) # 重用查询缓存
```

何时不会被缓存

有一些操作不会缓存QuerySet，例如切片和索引。这就导致这些操作没有缓存可用，每次都会执行实际的数据库查询操作。例如：

```
>>> queryset = Entry.objects.all()
>>> print queryset[5] # 查询数据库
>>> print queryset[5] # 再次查询数据库
```

但是，如果已经遍历过整个QuerySet，那么就相当于缓存过，后续的操作则会使用缓存，例如：

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # 查询数据库
>>> print queryset[5] # 使用缓存
>>> print queryset[5] # 使用缓存
```

下面的这些操作都将遍历QuerySet并建立缓存：

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

注意：简单的打印QuerySet并不会建立缓存，因为__repr__()调用只返回全部查询集的一个切片。

3.2.2.4 使用Q对象进行复杂查询

普通filter函数里的条件都是“and”逻辑，如果你想实现“or”逻辑怎么办？用Q查询！

Q来自django.db.models.Q，用于封装关键字参数的集合，可以作为关键字参数用于filter、exclude和get等函数。

例如：

```
from django.db.models import Q
Q(question__startswith='What')
```

可以使用“&”或者“|”或“~”来组合Q对象，分别表示与或非逻辑。它将返回一个新的Q对象。

```
Q(question__startswith='Who')|Q(question__startswith='What')
# 这相当于：
```

```
# WHERE question LIKE 'Who%' OR question LIKE 'What%'
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

你也可以这么使用，默认情况下，以逗号分隔的都表示AND关系：

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
# 它相当于
# SELECT * from polls WHERE question LIKE 'Who%'
# AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

当关键字参数和Q对象组合使用时，Q对象必须放在前面，如下例子：

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who',
)
```

如果关键字参数放在Q对象的前面，则会报错。

（译者：关于Q对象还有更高级的使用方式）

3.2.2.5 比较对象

要比较两个模型实例，只需要使用python提供的双等号比较符就可以了。在后台，其实比较的是两个实例的主键的值。

下面两种方法是等同的：

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

如果模型的主键不叫做“id”也没关系，后台总是会使用正确的主键名字进行比较，例如，如果一个模型的主键的名字是“name”，那么下面是相等的：

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

3.2.2.6 删除对象

删除对象使用的是delete()方法。该方法将返回被删除对象的总数量和一个字典，字典包含了每种被删除对象的类型和该类型的数量。如下所示：

```
>>> e.delete()
(1, {'weblog.Entry': 1})
```

可以批量删除。每个QuerySet都有一个delete()方法，它能删除该QuerySet的所有成员。例如：

```
>>> Entry.objects.filter(pub_date__year=2005).delete()
(5, {'webapp.Entry': 5})
```

需要注意的是，有可能不是每一个对象的delete方法都被执行。如果你改写了delete方法，为了确保对象被删除，你必须手动迭代QuerySet进行逐一删除操作。

当Django 删除一个对象时，它默认使用SQL的ON DELETE CASCADE约束，也就是说，任何有外键指向要删除对象的对象将一起被删除。例如：

```
b = Blog.objects.get(pk=1)
# 下面的动作将删除该条Blog和所有的它关联的Entry对象
b.delete()
```

这种级联的行为可以通过的ForeignKey的on_delete参数自定义。

注意，delete()是唯一没有在管理器上暴露出来的方法。这是刻意设计的一个安全机制，用来防止你意外地请求类似Entry.objects.delete()的动作，而不慎删除了所有的条目。如果你确实想删除所有的对象，你必须明确地请求一个完全的查询集，像下面这样：

```
Entry.objects.all().delete()
```

3.2.2.7 复制模型实例

虽然没有内置的方法用于复制模型的实例，但还是很容易创建一个新的实例并将原实例的所有字段都拷贝过来。最简单的方法是将原实例的pk设置为None，这会创建一个新的实例copy。示例如下：

```
blog = Blog(name='My blog', tagline='Bloggging is easy')
blog.save() # blog.pk == 1
#
blog.pk = None
```

```
blog.save() # blog.pk == 2
```

但是在使用继承的时候，情况会变得复杂，如果有下面一个Blog的子类：

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)
    #
    django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')
    django_blog.save() # django_blog.pk == 3
```

基于继承的工作机制，你必须同时将pk和id设为None：

```
django_blog.pk = None
django_blog.id = None
django_blog.save() # django_blog.pk == 4
```

上面的方法都没有涉及关联对象。如果你想复制关系，你必须手动多写一点代码，像下面这样：

```
entry = Entry.objects.all()[0] # 一些原始的entry对象
old_authors = entry.authors.all()
entry.pk = None
entry.save()
entry.authors = old_authors # 保存新的多对多关系
```

3.2.2.8 批量更新对象

使用update()方法可以批量为QuerySet中所有的对象进行更新操作。

```
# 更新所有2007年发布的entry的headline
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

你只可以对普通字段和ForeignKey字段使用这个方法。若要更新一个普通字段，只需提供一个新的常数值。若要更新ForeignKey字段，需设置新值为你想指向的新模型实例。例如：

```
>>> b = Blog.objects.get(pk=1)
# 修改所有的Entry，让他们都属于b
>>> Entry.objects.all().update(blog=b)
```

update方法会被立刻执行，并返回操作匹配到的行的数目（有可能不等于要更新的行的数量，因为有些行可能已经有这个新值了）。唯一的约束是：只能访问一张数据库表。你可以根据关系字段进行过滤，但你只能更新模型主表的字段。例如：

```
>>> b = Blog.objects.get(pk=1)
# Update all the headlines belonging to this Blog.
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

要注意的是update()方法会直接转换成一个SQL语句，并立刻批量执行。它不会运行模型的save()方法，或者产生pre_save或post_save信号（调用save()方法产生）或者服从auto_now字段选项。如果你想保存QuerySet中的每个条目并确保每个实例的save()方法都被调用，你不需要使用任何特殊的函数来处理。只需要迭代它们并调用save()方法：

```
for item in my_queryset:
    item.save()
```

update方法可以配合F表达式。这对于批量更新同一模型中某个字段特别有用。例如增加Blog中每个Entry的pingback个数：

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

然而，与filter和exclude子句中的F()对象不同，在update中你不可以使用F()对象进行跨表操作，你只可以引用正在更新的模型的字段。如果你尝试使用F()对象引入另外一张表的字段，将抛出FieldError异常：

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

3.2.2.9 关联的对象

利用本节一开始的模型，一个Entry对象e可以通过blog属性e.blog获取关联的Blog对象。反过来，Blog对象b可以通过entry_set属性b.entry_set.all()访问与它关联的所有Entry对象。

一对多（外键）

正向查询

直接通过圆点加属性，访问外键对象：

```
>>> e = Entry.objects.get(id=2)
```

```
>>> e.blog # 返回关联的Blog对象
```

要注意的是，对外键的修改，必须调用**save**方法进行保存，例如：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

如果一个外键字段设置有**null=True**属性，那么可以通过给该字段赋值为**None**的方法移除外键值：

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

在第一次对一个外键关系进行正向访问的时候，关系对象会被缓存。随后对同样外键关系对象的访问会使用这个缓存，例如：

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog) # 访问数据库，获取实际数据
>>> print(e.blog) # 不会访问数据库，直接使用缓存的版本
```

请注意**QuerySet**的**select_related()**方法会递归地预填充所有的一对多关系到缓存中。例如：

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog) # 不会访问数据库，直接使用缓存
>>> print(e.blog) # 不会访问数据库，直接使用缓存
```

反向查询

如果一个模型有**ForeignKey**，那么该**ForeignKey**所指向的外键模型的实例可以通过一个管理器进行反向查询，返回源模型的所有实例。默认情况下，这个管理器的名字为**FOO_set**，其中**FOO**是源模型的小写名称。该管理器返回的查询集可以用前面提到的方式进行过滤和操作。

```
>>> 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()
```

你可以在**ForeignKey**字段的定义中，通过设置**related_name**来重写**FOO_set**的名字。举例说明，如果你修改**Entry**模型**blog = ForeignKey(Blog, on_delete=models.CASCADE,**

related_name='entries')，那么上面的例子会变成下面的样子：

```
>>> 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()
```

使用自定义的反向管理器

默认情况下，用于反向关联的**RelatedManager**是该模型默认管理器的子类。如果你想为一个查询指定一个不同的管理器，你可以使用下面的语法：

```
from django.db import models

class Entry(models.Model):
    #...
    objects = models.Manager() # 默认管理器
    entries = EntryManager() # 自定义管理器
```

```
b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

当然，指定的自定义反向管理器也可以调用它的自定义方法：

```
b.entry_set(manager='entries').is_published()
```

处理关联对象的其它方法

除了在前面“检索对象”一节中定义的**QuerySet**方法之外，**ForeignKey**管理器还有其它方法用于处理关联的对象集合。下面是每个方法的概括，完整的细节可以在关联**6.15.4**中找到。

add(obj1, obj2, ...)：添加指定的模型对象到关联的对象集中。

create(kwargs)**：创建一个新的对象，将它保存并放在关联的对象集中。返回新创建的对象。

remove(obj1, obj2, ...)：从关联的对象集中删除指定的模型对象。

`clear()`: 清空关联的对象集。

`set(objs)`: 重置关联的对象集。

若要一次性给关联的对象集赋值, 使用`set()`方法, 并给它赋值一个可迭代的对象集合或者一个主键值的列表。
例如:

```
b = Blog.objects.get(id=1)
b.entry_set.set([e1, e2])
```

在这个例子中, `e1`和`e2`可以是完整的`Entry`实例, 也可以是整数的主键值。

如果`clear()`方法可用, 那么在将可迭代对象中的成员添加到集合中之前, 将从`entry_set`中删除所有已经存在的对象。如果`clear()`方法不可用, 那么将直接添加可迭代对象中的成员而不会删除所有已存在的对象。

这节中的每个反向操作都将立即在数据库内执行。所有的增加、创建和删除操作也将立刻自动地保存到数据库内。

多对多

多对多关系的两端都会自动获得访问另一端的API。这些API的工作方式与前面提到的“反向”一对多关系的用法一样。

唯一的区别在于属性的名称: 定义`ManyToManyField`的模型使用该字段的属性名称, 而“反向”模型使用源模型的小写名称加上`'_set'` (和一对多关系一样)。

```
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.
```

与外键字段中一样, 在多对多的字段中也可以指定`related_name`名。

(译者注: 在一个模型中, 如果存在多个外键或多对多的关系, 必须给他们分别加上不同的`related_name`, 用于反向查询)

一对一

一对一非常类似多对一关系, 可以简单的通过模型的属性访问关联的模型。

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()
#
ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

不同之处在于反向查询的时候。一对一关系中的关联模型同样具有一个管理器对象, 但是该管理器表示一个单一的对象而不是对象的集合:

```
e = Entry.objects.get(id=2)
e.entrydetail # 返回关联的EntryDetail对象
```

如果没有对象赋值给这个关系, `Django`将抛出一个`DoesNotExist`异常。

可以给反向关联进行赋值, 方法和正向的关联一样:

```
e.entrydetail = ed
```

反向关联是如何实现的?

一些ORM框架需要你在关系的两端都进行定义。`Django`的开发者认为这违反了DRY (Don't Repeat Yourself)原则, 所以在`Django`中你只需要在一端进行定义。

那么这是怎么实现的呢? 因为在关联的模型类没有被加载之前, 一个模型类根本不知道有哪些类和它关联。

答案在app registry! 在`Django`启动的时候, 它会导入所有`INSTALLED_APPS`中的应用和每个应用中的模型模块。每创建一个新的模型时, `Django`会自动添加反向的关系到所有关联的模型。如果关联的模型还没有导入, `Django`将保存关联的记录并在关联的模型导入时添加这些关系。

由于这个原因, 将模型所在的应用都定义在`INSTALLED_APPS`的应用列表中就显得特别重要。否则, 反向关联将不能正确工作。

通过关联对象进行查询

涉及关联对象的查询与正常值的字段查询遵循同样的规则。当你指定查询需要匹配的值时，你可以使用一个对象实例或者对象的主键值。

例如，如果你有一个`id=5`的`Blog`对象`b`，下面的三个查询将是完全一样的：

```
Entry.objects.filter(blog=b) # 使用对象实例
Entry.objects.filter(blog=b.id) # 使用实例的id
Entry.objects.filter(blog=5) # 直接使用id
```

3.2.2.10 调用原生SQL语句

如果你发现需要编写的Django查询语句太复杂，你可以回归到手工编写SQL语句。Django对于编写原生的SQL查询有许多选项，参见3.2.6节执行原生的SQL查询。

最后，需要注意的是Django的数据库层只是一个数据库接口。你可以利用其它的工具、编程语言或数据库框架来访问数据库，Django没有强制指定你非要使用它的某个功能或模块。

[Django 1.10 中文文档](#)-----3.3.8 会话sessions

django支持匿名会话。它将数据存放在服务器端，并抽象cookies的发送和接收过程。cookie包含一个会话ID而不是数据本身（除非你使用的是基于后端的cookie）。

3.3.8.1 启用会话

Django通过一个中间件来实现会话功能。要启用会话就要先启用该中间件。

编辑MIDDLEWARE设置，确保存在`django.contrib.sessions.middleware.SessionMiddleware`这一行。默认情况在新建的项目中它是存在的。

如果你不想使用会话功能，那么在`settings`文件中，将`SessionMiddleware`从MIDDLEWARE中删除，将`django.contrib.sessions`从`INSTALLED_APPS`中删除就OK了。

3.3.8.2 配置会话引擎

默认情况下，django将会话数据保存在数据库内（通过使用`django.contrib.sessions.models.Session`模型）。当然，你也可以将数据保存在文件系统或缓存内。

1. 基于数据库的会话

先在`INSTALLED_APPS`设置中，确保`django.contrib.sessions`的存在，然后运行`manage.py migrate`命令在数据库内创建sessions表。

2. 基于缓存的会话

从性能角度考虑，也许你想使用基于缓存的会话。

首先，你得先配置好你的缓存，请参考3.11节查看详细。

警告：

本地缓存不是多进程安全的，因此对于生产环境不是一个好的选择。

如果你定义有多个缓存，**django**将使用默认的那个。如果你想用其它的，请将**SESSION_CACHE_ALIAS**参数设置为那个缓存的名字。

配置好缓存后，你可以选择两种保存数据的方法：

- 一是将**SESSION_ENGINE**设置为"**django.contrib.sessions.backends.cache**"，简单的对会话进行保存。但是这种方法不是很可靠，因为当缓存数据存满时将清除部分数据，或者遇到缓存服务器重启。
- 为了数据安全保障，你可以将**SESSION_ENGINE**设置为"**django.contrib.sessions.backends.cached_db**"。这种方式在每次缓存的时候会同时将数据在数据库内写一份。当缓存不可用时，会话会从数据库内读取数据。

两种方法都很迅速，但是第一种简单的缓存更快一些，因为它忽略了数据的持久性。如果你使用缓存+数据库的方式，你同样需要按上面小节所述对数据库进行配置。

3. 基于文件的会话

将**SESSION_ENGINE**设置为"**django.contrib.sessions.backends.file**"。同时，你必须查看

SESSION_FILE_PATH配置（默认根据**tempfile.gettempdir()**生产，就像/tmp目录），确保你的文件存储目录，以及Web服务器对该目录具有读写权限。

4. 基于cookie的会话

将**SESSION_ENGINE**设置为"**django.contrib.sessions.backends.signed_cookies**"。Django将使用加密签名工具和安全密钥设置保存会话的数据。

注意：

建议将**SESSION_COOKIE_HTTPONLY**设置为**True**，阻止javascript对会话数据的访问。

3.3.8.3 在视图中使用会话

当会话中间件启用后，传递给视图**request**参数的**HttpRequest**对象将包含一个**session**属性，就像一个字典对象一样。

你可以在视图的任何地方读写**request.session**属性，或者多次编辑使用它。

```
class backends.base.SessionBase
# 这是所有会话对象的基类，包含标准的字典方法：
__getitem__(key)
    Example: fav_color = request.session['fav_color']
__setitem__(key, value)
    Example: request.session['fav_color'] = 'blue'
__delitem__(key)
    Example: del request.session['fav_color']. 如果不存在会抛出异常
__contains__(key)
    Example: 'fav_color' in request.session
get(key, default=None)
    Example: fav_color = request.session.get('fav_color', 'red')
pop(key, default=__not_given)
    Example: fav_color = request.session.pop('fav_color', 'blue')
keys()
items()
setdefault()
clear()

# 它还有下面的方法：
flush()
    # 删除当前的会话数据和会话cookie。经常用在用户退出后，删除会话。

set_test_cookie()
    # 设置一个测试cookie，用于探测用户浏览器是否支持cookies。由于cookie的工作机制，你只有在下次用户请求的时候才可以测试。
test_cookie_worked()
    # 返回True或者False，取决于用户的浏览器是否接受测试cookie。你必须在之前先调用set_test_cookie()方法。
delete_test_cookie()
    # 删除测试cookie。
set_expiry(value)
    # 设置cookie的有效期。可以传递不同类型的参数值：
    • 如果值是一个整数，session将在对应的秒数后失效。例如request.session.set_expiry(300) 将在300秒后失效。
```

- 如果值是一个datetime或者timedelta对象，会话将在指定的日期失效
 - 如果为0，在用户关闭浏览器后失效
 - 如果为None，则将使用全局会话失效策略
- 失效时间从上一次会话被修改的时刻开始计时。

```
get_expiry_age()
    # 返回多少秒后失效的秒数。对于没有自定义失效时间的会话，这等同于SESSION_COOKIE_AGE.
    # 这个方法接受2个可选的关键字参数
```

- modification: 会话的最后修改时间（datetime对象）。默认是当前时间。
- expiry: 会话失效信息，可以是datetime对象，也可以是int或None

```
get_expiry_date()
    # 和上面的方法类似，只是返回的是日期
```

```
get_expire_at_browser_close()
    # 返回True或False，根据用户会话是否是浏览器关闭后就结束。
```

```
clear_expired()
    # 删除已经失效的会话数据。
```

```
cycle_key()
    # 创建一个新的会话密钥用于保持当前的会话数据。django.contrib.auth.login() 会调用这个方法。
```

1. 会话序列化

Django默认使用JSON序列化会话数据。你可以在SESSION_SERIALIZER设置中自定义序列化格式，甚至写入警告说明。但是我们强烈建议你还是使用JSON，尤其是以cookie的方式进行会话时。

举个例子，这里有一个使用pickle序列化会话数据的攻击场景。如果你使用的是已签名的Cookie会话并且SECRET_KEY被攻击者知道了（通过其它手段），攻击者就可以在会话中插入一个字符串，在pickle反序列化时，可以在服务器上执行危险的代码。在因特网上这个攻击技术很简单并很容易使用。尽管Cookie会话会对数据进行签名以防止篡改，但是SECRET_KEY的泄漏却使得一切前功尽弃。

绑定的序列化方法

```
class serializers.JSONSerializer
```

对 django.core.signing中JSON序列化方法的一个包装。只可以序列化基本的数据类型。另外，JSON只支持以字符串作为键值，使用其它的类型会导致异常

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following serialization & deserialization
>>> # of session data
>>> request.session[0] # KeyError
>>> request.session['0']
'bar'
```

同样，无法被JSON编码的，例如非UTF8格式的字节'\xd9'一样是无法被保存的，它会导致UnicodeDecodeError异常。

```
class serializers.PickleSerializer
```

支持任意类型的python对象，但是就像前面说的，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方法

你的序列化类必须分别实现dumps(self, obj)和loads(self, data)方法，用来实现序列化和反序列化会话数据字典。

2. 会话对象使用建议

- 使用普通的python字符串作为request.session字典的键值。这不是一条硬性规则而是为方便起见。
- 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。
- 不要用新对象覆盖request.session，不要访问或设置它的属性。像一个python字典一样的使用它。

3. 范例

这个简单的视图设置一个has_commented变量为True在用户发表评论后。它不允许用户重复发表评论。

```
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!')
```

下面是一个简单的用户登录视图：

```
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.")
```

下面则是一个退出登录的视图，与上面的相关：

```
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更严谨，以防止意外的数据泄露，它会调用request.session的flush()方法。我们使用这个例子只是演示如何利用会话对象来工作，而不是一个完整的logout()实现。

3.3.8.4 设置测试cookie

为了方便，Django 提供一个简单的方法来测试用户的浏览器是否接受Cookie。只需在一个视图中调用request.session的set_test_cookie()方法，并在随后的视图中调用test_cookie_worked()获取测试结果（True或False）。注意，不能在同一个视图中调用这两个方法。

造成这种分割调用的原因是cookie的工作机制。当你设置一个cookie时，你无法立刻得到结果，知道浏览器发送下一个请求。

在测试后，记得使用delete_test_cookie()方法清除测试数据。

下面是一个典型的范例：

```
from django.http import HttpResponse
from django.shortcuts import render

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(request, 'foo/login_form.html')
```

3.8.3.5 在视图外使用session

注意：

在下面的例子中，我们直接从django.contrib.sessions.backends.db中导入了SessionStore对象。在你的实际代码中，你应该采用下面的导入方法，根据SESSION_ENGINE的设置进行导入，如下所示：

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
```

在视图外有一个API可以操作会话数据：

```
>>> 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.create()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceed'
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceed')
>>> s['last_login']
1376587691
```

SessionStore.create()用于创建一个新的会话。save()方法用于保存一个已经存在的会话。create方法会调用save方法并循环直到生成一个未使用的session_key。直接调用save方法也可以创建一个新的会话，但在生成session_key的时候有可能和已经存在的发生冲突。

如果你使用的是`django.contrib.sessions.backends.db`模式，那么每一个会话其实就是一个普通的Django模型，你可以使用普通的Django数据库API访问它。会话模型的定义在`django/contrib/sessions/models.py`文件里。例如：

```
>>> 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()`方法才能获得会话字典，因为字典是采用编码格式保存的。如下：

```
>>> s.session_data
'KGRwMQpTJl9hdXR0X3VzZXJfaWQnCnAyCkxkCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

3.3.8.6 何时保存会话

默认情况下，只有当会话字典的任何值被指定或删除的时候，Django才会将会话内容保存到会话数据库内。

```
# 会话被修改
request.session['foo'] = 'bar'
# 会话被修改
del request.session['foo']
# 会话被修改
request.session['foo'] = {}
# 会话没有被修改，只是修改了request.session['foo']
request.session['foo']['bar'] = 'baz'
```

要理解上面最后一种情况有点费劲。我们可以通过设置会话对象的`modified`属性值，显式地告诉会话对象它已经被修改过：`request.session.modified = True`

要改变上面的默认行为，将`SESSION_SAVE_EVERY_REQUEST`设置为`True`，那么每一次单独的请求过来，Django都会保存会话到数据库。

注意，会话的Cookie只有在在一个会话被创建或修改后才会再次发送。如果`SESSION_SAVE_EVERY_REQUEST`为`True`，每个请求都会发送cookie。

类似地，会话Cookie的失效部分在每次发送会话Cookie时都会更新。

如果响应的状态码为500，则会话不会被保存。

3.3.8.7 浏览器生存期间会话 VS 持久会话

默认情况下，`SESSION_EXPIRE_AT_BROWSER_CLOSE`设置为`False`，也就是说cookie保存在用户的浏览器内，直到失效日期，这样用户就不必每次打开浏览器后都要再登录一次。

相反的`SESSION_EXPIRE_AT_BROWSER_CLOSE`设置为`True`，则意味着浏览器一关闭，cookie就失效，每次重新打开浏览器，你就得重新登录。

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，前面我们已经描述过了。

注意：有些浏览器（比如Chrome）具有在关闭后重新打开浏览器，会话依然保持的功能。这会与Django的`SESSION_EXPIRE_AT_BROWSER_CLOSE`设置发生冲突。请一定要小心。

3.3.8.8 清除已保存的会话

随着用户的访问，会话数据会越来越庞大。如果你使用的是数据库保存模式，那么`django_session`表的内容会逐渐增长。如果你使用的是文件模式，那么你的临时目录内的文件数量会不断增加。

造成这个问题的原因是，如果用户手动退出登录，Django将自动删除会话数据，但是如果用户不退出登录，那么对应的会话数据不会被删除。

Django没有提供自动清除失效会话的机制。因此，你必须自己完成这项工作。Django提供了一个命令`clearsessions`用于清除会话数据，建议你基于这个命令设置一个周期性的自动清除机制。

不同的是，使用缓存模式的会话不需要你清理数据，因为缓存系统自己有清理过期数据的机制。使用cookie模式的会话也不需要，因为数据都存在用户的浏览器内，不用你帮忙。

3.3.8.9 设置

这里有一些Django的设置，用于帮助你控制会话的行为：

- `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
- SESSION_SERIALIZER

3.3.8.10 会话安全

一个站点下的子域名能够在为整个域名的客户设置Cookie。如果子域名被不受信任的用户控制，那么可能发生会话安全问题。

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。如果该攻击者控制了bad.example.com域名，那么他就可以使用这个域名来发送他的会话密钥给你，因为子域名允许在*.example.com上设置Cookie。当你访问good.example.com时，你有可能以攻击者的身份登录，然后无意中泄露了你的个人敏感信息（例如信用卡信息）到攻击者的账号中。攻击者自然就获得了这些信息。

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"，这可能导致来自该站点的会话Cookie被发送到bad.example.com。

3.3.8.11 技术细节

- 会话字典接收任意的json序列化值，或者任何可通过pickle序列化的python对象
- 会话数据被保存在一张名为django_session的表内
- Django 只发送它需要的Cookie。如果你没有设置任何会话数据，它不会发送任何Cookie

SessionStore对象

在会话内部，Django使用一个与会话引擎对应的会话保存对象。根据管理，这个会话保存对象命名为SessionStore，位于SESSION_ENGINE设置指定的模块内。

所有Django支持的SessionStore类都继承SessionBase类，并实现了下面的数据操作方法：

- exists()
- create()
- save()
- delete()
- load()
- clear_expired()

为了创建一个自定义会话引擎或修改一个现成的引擎，你也许需要创建一个新的类，它继承SessionBase类或任何其他已经存在的SessionStore类。

3.3.8.12 扩展基于数据库的会话引擎

Django 1.9版本以后才有的功能。

要创建一个自定义的基于数据库的会话引擎，需要继承AbstractBaseSession类或者SessionStore类。

AbstractBaseSession和BaseSessionManager可以从django.contrib.sessions.base_session内导入，因此不一定非要在INSTALLED_APPS中包含django.contrib.sessions。

```
class base_session.AbstractBaseSession # 抽象会话基类
    session_key
        主键。最多40个字符。目前是一个32为随机数字或字母组合字符串。
    session_data
        一个包含了编码过的或序列化过的会话字典的字符串
    expire_date
        失效日期
    get_session_store_class()
        这是一个类方法。返回一个会话保存类。
    get_decoded()
        返回解码后的会话数据。通过会话保存类进行解码。
```

你也可以自定义模型管理器，通过编写一个BaseSessionManager的子类。

```
class base_session.BaseSessionManager
    encode(session_dict)
```


通过会话保存类，将会话字典序列化或编码成一个字符串
`save(session_key, session_dict, expire_date)`
根据一个提供的session密钥保存会话数据，或者删除一个空的会话

通过重写下面这些SessionStore类的方法和属性，可以进行自定义：

```
class backends.db.SessionStore
    实现基于数据库的会话保存
    get_model_class()
        这是一个类方法。如果有需要，重写这个方法并返回一个自定义的会话模型。
    create_model_instance(data)
        返回一个会话模型对象的新实例，它代表当前会话的状态。重写这个方法，你将获得在它被保存之前，修改
        会话模型数据的能力。

class backends.cached_db.SessionStore
    实现基于缓存和数据库的会话保存
    cache_key_prefix
        在会话密钥前添加一个前缀，用于构造缓存键值字符串。
```

范例

下面的例子展示一个自定义的基于数据库的会话引擎，包括一个额外的数据列用于储存用户的ID。

```
from django.contrib.sessions.backends.db import SessionStore as DBStore
from django.contrib.sessions.base_session import AbstractBaseSession
from django.db import models

class CustomSession(AbstractBaseSession):
    account_id = models.IntegerField(null=True, db_index=True)

    @classmethod
    def get_session_store_class(cls):
        return SessionStore

class SessionStore(DBStore):
    @classmethod
    def get_model_class(cls):
        return CustomSession

    def create_model_instance(self, data):
        obj = super(SessionStore, self).create_model_instance(data)
        try:
            account_id = int(data.get('_auth_user_id'))
        except (ValueError, TypeError):
            account_id = None
        obj.account_id = account_id
        return obj
```

如果你是通过从Django内置的cached_db会话保存迁移到自定义的cached_db，你应该重写缓存键值的前缀，以防止命名空间的冲突，如下所示：

```
class SessionStore(CachedDBStore):
    cache_key_prefix = 'mysessions.custom_cached_db_backend'
    # ...
```

3.3.8.13 URLs中的会话IDs

Django的会话框架完全地、唯一地基于Cookie。它不像PHP一样，把会话的ID放在URL中。它不仅使得URL变得丑陋，还使得你的网站易于受到通过"Referer"头部进行窃取会话ID的攻击。