开始写第一个django 应用  
第一部分  
通过这个教程，可以让你创建一个简单的django 应用  
该教程共包含两部分：  
1、一个面向用户的站点，用户可以查看调查列表，然后给它们投票  
2、一个管理站点，可以对调查列表进行增加，修改和删除  
在确认你已经安装好了django，你可以通过以下命令来输出你所安装的django的版本号：

python -c "import django; print(django.get\_version())"

如果django没有安装好，就会报错“No module named django”。  
本教程是基于Django 1.6/Python 2.x来编写的，如果你所安装的django版本不是1.6,可以在当前页的右边的搜索框下的版本下拉列表中选择自己所安装的版本号，或者升级你的django到1.6版本(即最新版本）。如果你使用的是python3.x,你可能需要对代码做一些修改，所以使用该版本的前提是你知道如何用python3.ｘ来实现本教程中用2.x实现的功能。

一、创建一个django项目  
使用命令cd到你想要保存项目的目录下，然后运行以下命令，生成一个django项目

django-admin.py startproject mysite

以上命令会在当前目录下生成一个叫mysite的目录，没有成功请参照：https://docs.djangoproject.com/en/1.6/faq/troubleshooting/#troubleshooting-django-admin-py

注意1：项目名称不要使用python的内建标识符或者Django的组建名称，这样会造成冲突;另外，处于安全考虑，不建议把django项目放在web服务器的根目录下(比如/var/www)。

看看mysite项目下面的文件：

[复制代码](javascript:void(0);)

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

[复制代码](javascript:void(0);)

注意2：如果你所创建的项目目录结构不是这样，可能是教程版本与你安装的版本不一致造成的  
这些文件分别是：  
mysite/ :项目名，你可以把它重命名成任意你喜欢的名字，参照注意1  
manage.py :一个命令行工具，用于与项目的交互。用法参照：https://docs.djangoproject.com/en/1.6/ref/django-admin/  
mysite/\_\_init\_\_.py:一个空文件，用于告诉python 解释器该文件所在目录是一个python包  
mysite/settings.py:用于项目的配置  
mysite/urls.py:用于url的地址映射以及管理URL的地址格式  
mysite/wsgi.py: 项目入口

二、现在启动服务器.在项目的根目录mysite下，运行以下命令：

python manage.py runserver

你会看到一下输出：

[复制代码](javascript:void(0);)

Validating models...

0 errors found

November 10, 2013 - 15:50:53

Django version 1.6, using settings 'mysite.settings'

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

Quit the server with CONTROL-C.

[复制代码](javascript:void(0);)

以上信息说明你已经成功启动了django 开发服务器，一个轻量级的纯python服务器。使用该服务器可以快速部署我们的web应用.  
注意3：当项目或者应用真正部署的时候不要使用该服务器

现在你在浏览器上访问http://127.0.0.1:8000/，就可以看到"Welcome to Django"的页面。  
现在你的应用成功创建了

注意4：可以使用以下命令修改服务器端口和iｐ地址：

python manage.py runserver 8080

python manage.py runserver 0.0.0.0:8000

这样你访问的 url就变成了：http://127.0.0.1:8080,http://0.0.0.0:8000  
不需要重启服务器，来更新代码

三、设置  
1、数据库设置  
现在编辑mysite/settings.py.有点类似于yii框架中main全局配置文件  
数据库默认使用Sqlite.如果你对数据库比较陌生，或者你只对django感兴趣。那么这个默认配置就是你最简单易用的选择。sqlite是自包含在python中的，不需要你额外需要安装和配置。  
如果你想使用另外的数据库，需要修改settings.py中的DATABASES'default成你想要使用的数据库名称。django支持的数据库，引擎有：'django.db.backends.sqlite3', 'django.db.backends.postgresql\_psycopg2', 'django.db.backends.mysql', or 'django.db.backends.oracle'. 等等  
如果你不是使用Sqlite作为你的数据库，那么还需要添加并设置USER/PASSWORD/HOST这些配置项，更多细节参照：https://docs.djangoproject.com/en/1.6/ref/settings/#std:setting-DATABASES

2、时区设置  
把TIME\_ZONE设置成你当前时区

3、INSTALLED\_APPS  
该配置项的值是一个元组，每个元素是在当前Django实例中激活的所有应用的名称。这些应用可以在被很多的项目使用，也可以打包然后在其他的项目中部署。  
默认情况下，django项目自动包含以下apps:  
django.contrib.admin---一个管理站点。这个将在第二部分的教程中用到  
django.contrib.auth---一个认证系统  
django.contrib.contenttypes---一个引入文档类型框架  
django.contrib.sessions---会话框架  
django.contrib.messages---消息框架  
django.contrib.staticfiles---管理静态文件的框架  
这些apps默认包含在新创键的项目中。以上这些应用会只用到数据库，所以需要在使用事前把需要用的数据表创建出来。创建命令如下：

python manage.py syncdb

syncdb命令是根据INSTALLED\_APPS的配置项创建它们需要的数据表.创建数据表的时候，会提示创建一个超级用户，用于以后访问数据时的认证。

4、创建模型  
现在项目已经创建并且设置好了，我们现在需要用它来做点其他的事情。  
django会自动生成每个应用的基础目录结构，这样就就可以专注与编程而不是创建应用的目录  
在项目的根目录下，使用以下目录创建应用：

python manage.py startapp polls

polls 应用的目录结构如下：

polls/

\_\_init\_\_.py

admin.py

models.py

tests.py

views.py

创建数据库应用的第一步是定义你的数据模型，实际上就是数据布局和一些额外的元数据  
模型是单一的，明确的数据源。它包含数据表中的所有字段和数据操作方法。django遵循DRY原则。我们的原则是一次定义，多次派生。

在本教程中，我们创建两个模型：Poll,Choice.Poll是包含的数据域有“ 题”和”问题发布日，Choice也有两个数据域：一个选项文本，以及对应的得票总数。每一个Choice与Poll联合使用  
两个模型定义如下，并保存文件为Polls/models：

[复制代码](javascript:void(0);)

from django.db import models

class Poll(models.Model):

　　question = models.CharField(max\_length=200)

　　pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

　　poll = models.ForeignKey(Poll)

　　choice\_text = models.CharField(max\_length=200)

　　votes = models.IntegerField(default=0)

[复制代码](javascript:void(0);)

以上代码表示，每个模型就是一个django.db.models.Model子类，每个模型类都会有定义一些变量，这些变量就是上面提到的数据域，这些数据域代表着数据库表中的字段。这些变量名就是对应的数据表字段名。每个变量的值是django中Field类的实例。

在创建Field实例中，可以第一个参数是可选的，通过指定第一个参数，可以使用可读性强的名称，在页面中表示一个数据域，比如上例中的pub\_date数据域，在页面中的显示名称为"date published".创建某些Field实例时，必须设置某些参数，比如CharField，需要传递一个max\_length参数。最后还需要注意，外键的定义ForeignKey,它告诉django每个Choice都有一个对应的Poll。django支持所有的数据映射关系类型：多对一，多对多，一对多，一对一。

5、激活模型  
django根据所定义的模型类：1、创建相应的数据库表 2、创建一个数据库访问API用于访问1中创建的数据库表。  
前提是安装所创建的polls应用。编辑mysite/settings.py，在INSTALLED\_APPS中添加polls即可安装应用  
然后使用以下命令来看看我们创建的数据库表：

python manage.py sql polls

你会看到类似于下面的输出：

[复制代码](javascript:void(0);)

BEGIN;

CREATE TABLE "polls\_poll" (

"id" integer NOT NULL PRIMARY KEY,

"question" varchar(200) NOT NULL,

"pub\_date" datetime NOT NULL

);

CREATE TABLE "polls\_choice" (

"id" integer NOT NULL PRIMARY KEY,

"poll\_id" integer NOT NULL REFERENCES "polls\_poll" ("id"),

"choice\_text" varchar(200) NOT NULL,

"votes" integer NOT NULL

);

COMMIT;

[复制代码](javascript:void(0);)

注意5：  
1、以上的命令输出依赖于你所使用的数据库类型，本例中为sqlite  
2、数据表名是由小写的ａｐｐ名和模型类名组成的  
3、主键ｉｄ是自动添加的，你可以在模型类定义的时候，定义自己的主键来覆盖它  
4、外键的字段名会自动的添加”\_外键名称“来表示本例中的外键名称为ｉｄ，你也可以定义覆盖它  
5、外键的实现使用的语句是 REFERENCES  
6、上面输出的信息或者操作并没有真正的执行

下面运行syncdb，在你的数据库中创建模型表

python manage.py syncdb

syncdb命令为INSTALLED\_APPS中所有的app创建它们需要的数据表，而且该命令只会创建那些需要的但是不存在的数据表，也就是所已经存在的数据表不会被重复创建。

6、使用数据模型ＡＰＩ  
使用以下命令进入django的shell 交互模式：

python manage.py shell

在shell模式下，你可以查看你所创建的模型以及存储的数据是否成功等  
详情见（https://docs.djangoproject.com/en/1.6/intro/tutorial01/#playing-with-the-api）

7、模型对象的可读性  
为每个模型对象添加\_\_str\_\_(\_\_unicode\_\_())，这样在打印显示一个模型对象的时候可读性更好

添加的代码为：

[复制代码](javascript:void(0);)

from django.db import models

class Poll(models.Model):

　　# ...

　　def \_\_unicode\_\_(self): # Python 3: def \_\_str\_\_(self):

　　　　return self.question

class Choice(models.Model):

　　# ...

　　def \_\_unicode\_\_(self): # Python 3: def \_\_str\_\_(self):

　　return self.choice\_text

现在添加一个自定义的方法：

import datetime

from django.utils import timezone

# ...

class Poll(models.Model):

　　# ...

　　def was\_published\_recently(self):

　　return self.pub\_date >= timezone.now() - datetime.timedelta(days=1)

[复制代码](javascript:void(0);)

本教程关注于教程一中自动生成的管理站点  
注意1：为你的客户或者员工创建一个管理站点来添加、删除、更改内容是很乏味的事情，这不需要太多的创造性。基于此，django完全自动化地为模型生成这个管理站点。django是从一个新闻类网站开发的环境中演化出来的，它的内容编辑以及内容发布的站点是明确分开的。网站管理源使用这个系统添加新闻，事件，体育得分等内容。然后这些内容在内容发布站点中显示。django解决了为管理者创建一个同一的交互界面来编辑内容。这个管理站点只是网站管理员有权限使用。

1、启动开发服务器  
django管理站点是默认激活的，启动服务器即可访问

python manage.py runserver

访问http://127.0.0.1:8000/admin/即可来到管理站点的登录界面  
关于汉化参照：https://docs.djangoproject.com/en/1.6/topics/i18n/translation/

2、登录的用户名密码为你在教程一种创建的数据库超级用户的帐号密码

3、在admin站点中注册polls app中定义的模型,打开admin.py,编辑成以下内容

from django.contrib import admin

from polls.models import Poll

admin.site.register(Poll)

贴士：在admin管理站点中，以app为单位来进行划分管理  
4、自定义admin显示格式  
把admin.py编辑成：

from django.contrib import admin

from polls.models import Poll

class PollAdmin(admin.ModelAdmin):

　　fields = ['pub\_date', 'question']  
admin.site.register(Poll, PollAdmin)

贴士：以上代码功能是在添加一个新的poll的界面，输入时间的field在输入问题的field上面，默认是输入问题的field在上面的  
或者，编辑成：

[复制代码](javascript:void(0);)

from django.contrib import admin

from polls.models import Poll

class PollAdmin(admin.ModelAdmin):

　　fieldsets = [

　　　　(None, {'fields': ['question']}),

　　　　('Date information', {'fields': ['pub\_date']}),

　　]

admin.site.register(Poll, PollAdmin)

[复制代码](javascript:void(0);)

贴士：同样是poll添加页面，把两个字段分别在两个不同的fieldset中显示  
再或者：

[复制代码](javascript:void(0);)

from django.contrib import admin

from polls.models import Poll

class PollAdmin(admin.ModelAdmin):

　　fieldsets = [

　　　　(None, {'fields': ['question']}),

　　　　('Date information', {'fields': ['pub\_date'], 'classes': ['collapse']}),

　　]

[复制代码](javascript:void(0);)

贴士：设置pub\_date fields默认是折叠状态的  
仔细比较以上的改变之间的区别  
贴士：可以在admin中自定义并指定模型的显示样式  
5、添加相关联的对象Choice  
编辑admin.py,添加以下代码

from django.contrib import admin

from polls.models import Choice

admin.site.register(Choice)

在Choice添加页面中，Poll域是一个下拉列表，包含了所有的poll。在django中外键都以下拉列表的形式显示，注意下拉列表方便的绿色十字，点击它可以弹出一个poll的添加页面，你在这个页面中添加的poll会同时在poll的存储数据库和下拉列表中动态更新。

但是，这样给每个poll一条一条的添加Choice太低效了。我们可以通过以下方式来提高效率  
在admin.py中，删除注册Choice那条语句，然后把文件内容改成以下：

[复制代码](javascript:void(0);)

from django.contrib import admin

from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):

model = Choice

extra = 3

class PollAdmin(admin.ModelAdmin):

fieldsets = [

(None, {'fields': ['question']}),

('Date information', {'fields': ['pub\_date'], 'classes': ['collapse']}),

]

inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)

[复制代码](javascript:void(0);)

贴士：通过上面的修改，我们可以在添加poll的页面中添加choice。默认生成3个choice的创建表单，但是你可以自如的增添

有一个小问题，那3个choice创建表单的显示方式有点占用页面显示的空间，django提供一种更好的表格形式，来显示这个通过内联的方式添加的对象。这时候需要你修改ChoiceInline类的声明成下面的形式：

class ChoiceInline(admin.TabularInline):

#...

默认情况下，每个poll对象的显示内容与“print poll对“内容一样，但是通过使用admin 选项list\_display可以把对象中想要显示出来的数据属性在poll的显示页面中显示出来，需要添加的代码如下：

class PollAdmin(admin.ModelAdmin):

　　# ...

　　list\_display = ('question', 'pub\_date')

它还可以包含通过调用对象中的方法，并显示结果，如下：

class PollAdmin(admin.ModelAdmin):

　　# ...

　　list\_display = ('question', 'pub\_date', 'was\_published\_recently')

你可以点击显示数据属性的列(注意只是数据属性的列，不包含was\_published\_recently)，可是实现以此列为准排序。另外默认的显示对象方法运行结果的列名就是方法名。  
不过你可以通过给该方法添加一些数据属性来改善这种情况，这样你也可以使用该列来进行排序了。代码如下：

[复制代码](javascript:void(0);)

class Poll(models.Model):

# ...

def was\_published\_recently(self):

return self.pub\_date >= timezone.now() - datetime.timedelta(days=1)

was\_published\_recently.admin\_order\_field = 'pub\_date'

was\_published\_recently.boolean = True

was\_published\_recently.short\_description = 'Published recently?'

[复制代码](javascript:void(0);)

通过添加过滤侧边栏，可以提供一个简单的过滤功能，在PollAdmin中添加下面的代码：

list\_filter = ['pub\_date']

这个过滤器的过滤依据是pub\_date,由于pub\_date是DateTimeField类型，所以给出以下这个过滤选项：“Any date,” “Today,” “Past 7 days,” “This month,” “This ye

现在添加搜索框，同样是在PollAdmin中：

search\_fields = ['question']

你可以在列表中添加多个数据属性，查询方式就像数据库中的LIKE语句。change页面也支持分页机制，默认是每页最多显示100条记录。

显然的，admin站点最上面的"Django administration"字样只是起着占位符的作用，我们可以把他自定义成自己喜欢的文字。使用django的模板系统很容易做到。

在你的项目中创建自己的模板目录，模板目录可以在任何的django可以访问到的地方创建，但是建议把模板目录放在项目的根目录下便于使用。

编辑mysite/settings.py,添加TEMPLATE\_DIRS设置项  
TEMPLATE\_DIRS = [os.path.join(BASE\_DIR, 'templates')]  
TEMPLATE\_DIRS是一个可迭代的文件系统目录，也就是python中的搜索路径，通过它来获取加载对于位置的django模板

现在在根目录下创建templates目录，在该目录下添加一个admin目录，把django默认的admin模板复制到admin目录下，也就是django/contrib/admin/templates/admin/base\_site.html这个文件。  
你可以通过在python的命令行模式下输入一下命令来获取django的安装路径：

python

import django

django.\_\_file\_\_

现在打开刚刚复制过来的base\_site.html，找到"Django administrator"这些字，然后把它修改成你想要显示的字。然后刷新页面就可以看到变化了

这个模板文件包含很多文本像{% block branding %}and{{ title }}.{%和{{标签是django模板中的语言，当django渲染admin/base\_site.html这个模板时，这个模板语言就会解析成最终的HTML页面。在教程3中仔细的讲解django的模板语言。  
注意:django的所有模板语言都可以被覆盖。覆盖一个模板的方式，跟base\_site.html一样，从django系统默认的模板目录中复制到自己项目的模板目录下，然后修改。

管理站点默认显示在INSTALLED\_APPS声明并且在admin中注册中的应用，显示顺序是字母序。

管理站点的首页面对于的模板文件为admin/index.html(修改方式跟admin/base\_site.html一样),定制教程见教程3.

在教程3的主要内容是创建视图界面

django的设计思想：  
视图在django应用中是一个网页类型，通常它会提供特定的功能以及有一个指定的模板，比如说，在一个博客应用中，你可能需要使用以下类型的视图：  
1、博客主页——用于显示最新的文章  
2、博文显示页面——用于显示具体的博文  
3、按照年份存档的博文列表页面——用于显示指定年份的所有文章  
4、按照月份存放的博文列表页面——用于显示指定月份的所有文章  
5、按照某一天存档的博文列表页面——用于显示指定某一天的所有文章  
6、评论功能——用于处理给指定文章的发表评论

在我们创建poll应用中，会创建以下这4个视图：  
"index"——显示最新的几个polls  
"detail"——显示具体的某个poll，还带有一个投票的表单  
"results"——显示某个特定的poll的投票结果  
投票功能——处理给在一个特定的poll的某个choice投票的功能

在django中，web页面和其他的内容是通过试图来传递的。每个视图就表示一个简单的python 函数（或者说方法，对于基于类的视图）。根据请求的url来匹配选择一个视图（准确来讲，这里的url指的是请求url除去域名剩下的部分）

通过url获取视图的机制就是众所周知的"URLconfs",它使用URL模式来把匹配的url映射到某个视图上。  
本教程只是使用基本的URLconfs指令，更多细节参照：https://docs.djangoproject.com/en/1.6/ref/urlresolvers/#module-django.core.urlresolvers

1、创建第一个视图  
编辑polls/views.py，输入下面的代码：

from django.http import HttpResponse

def index(request):

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

上面的代码就是最简单的django视图了。为了让这个视图工作，需要把它映射到一个URL上，这样就需要一个URLconf  
在polls目录下创建一个URLconf，文件名叫urls.py.这时你的polls app目录应该是这样的:

[复制代码](javascript:void(0);)

polls/

\_\_init\_\_.py

admin.py

models.py

tests.py

urls.py

views.py

[复制代码](javascript:void(0);)

把polls/urls.py的文件内容如下:

[复制代码](javascript:void(0);)

from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',

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

)

[复制代码](javascript:void(0);)

下一步就是在主URLconf中添加polls/urls模块,打开mysite/urls.py,把import的内容和urlpatterns变成以下的样子:

[复制代码](javascript:void(0);)

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

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

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

)

[复制代码](javascript:void(0);)

现在你已经把index视图映射到polls这个url中了,通过访问http://localhost:8000/polls/就可以看到你在index视图定义的内容了  
这个url()函数需要传递4个参数,有两个是必须的regex(也就是正则表达式)和视图,另外两个是可选的kwargs和name.现在我们详细介绍以下这个函数的参数.  
url()参数 regex:  
"regex"是"regular expression"的缩写,正则表达式是字符串或者url(在这里)的匹配模式.django从第一个正则表达式开始遍历,知道找到和请求url匹配的正则表达式.注意,请求url的用于匹配的字符串不包括前面表示域名的部分以及后面get/post方法的参数部分.比如过,一个请求url为:http://www.example.com/myapp/,这个URLconf会匹配myapp/.在比如http://www.example.com/myapp/?page=3,URLconf也只会匹配myapp/.

更多关于正则表达式的内容参照http://en.wikipedia.org/wiki/Regular\_expression

url() 参数view:  
当django找到与url匹配的正则表达式,就会调用该正则表达式对应的视图函数,并且把HttpRequest对象作为第一个参数

url() 参数kwargs:  
可以向视图函数传递额外的参数,这个额外参数必须是一个字典类型的.  
url() 参数 name:往下看就知道怎么用了

创建其他的视图:

[复制代码](javascript:void(0);)

def detail(request, poll\_id):

　　return HttpResponse("You're looking at poll %s." % poll\_id)

def results(request, poll\_id):

　　return HttpResponse("You're looking at the results of poll %s." % poll\_id)

def vote(request, poll\_id):

　　return HttpResponse("You're voting on poll %s." % poll\_id)

[复制代码](javascript:void(0);)

修改polls/urls.py成以下内容:

[复制代码](javascript:void(0);)

from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',

# ex: /polls/

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

# ex: /polls/5/

　　url(r'^(?P<poll\_id>\d+)/$', views.detail, name='detail'),

# ex: /polls/5/results/

　　url(r'^(?P<poll\_id>\d+)/results/$', views.results, name='results'),

# ex: /polls/5/vote/

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

)

[复制代码](javascript:void(0);)

贴士:?P<poll\_id>是把视图的关键字参数名,它的参数值为\d+这个正则匹配的内容,在整个正则匹配结束后,会调用相应的视图函数,这些在匹配过程中生成的关键字参数也会传递进去.你可以看到上面定义的那三个视图都有一个叫"poll\_id"的参数.

访问"polls/34/",就会调用detail这个视图函数,显示id为34的poll,同样访问polls/34/results/和polls/34/vote分别会调用results和vote视图函数来显示在该函数中定义的内容.  
当请求/polls/34/这个url的时候,django会加载mysite.urls python模块,因为它通过ROOT\_URLCONF配置项直接标识了.这个模块把当前的url根mysite.urls文件中的urlpatterns中的正则表达式,来把这个请求url路由到其他的子URLconf中进一步处理.在mysite.urls每一个正则表达式中,对应着一个include回调函数,通过这种方式应用与请求url匹配的子URLconf.  
注意:回调函数是include时,正则表达式不能使用$,因为请求url没有与正则表达式匹配的那一部分需要保留下来进行进一步的处理.

使用这样的方式,你就可以任意的定义到polls app的url路径,比如之前只是匹配"polls/",你也可以改成"polls/content/".

现在分析下,请求url polls/34/,这时的url映射是怎样工作的:  
首先,在mysite.urls中匹配了'^polls/'  
然后,django 把请求url中剩下为匹配的部分"34/"传递给polls.urls,为进一步匹配做准备,最后与"34/"匹配的正则表达式为r'^(?P<poll\_id>\d+)/$',调用相应的回调函数detail()视图,detail(request=<HttpRequest object>, poll\_id='34')  
poll\_id='34',来自r'^(?P<poll\_id>\d+),?P<poll\_id>标识匹配模式名(相当一个变量名),匹配模式"\d+"是用于匹配请求url正则表达式,把成功匹配到的字符串赋值给前面定义的模式,这里是poll\_id.使用小括号,把匹配到的结果作为参数传递到对应的回调函数中.

以下,使用html后缀的url不建议使用.如  
(r'^polls/latest\.html$', 'polls.views.index'),

让视图做点什么  
每个视图都自动完成这两个动作中的一个:返回一个HttpResponse对象,这个对象包含着请求页面的内容,如果请求的页面不存在就抛出一个类似Http404的异常.其他的功能就需要自己去实现了.

修改视图index(),让它实现最新创建的5个poll,poll之间用逗号隔开.代码如下:

[复制代码](javascript:void(0);)

from django.http import HttpResponse

from polls.models import Poll

def index(request):

　　latest\_poll\_list = Poll.objects.order\_by('-pub\_date')[:5]

　　output = ', '.join([p.question for p in latest\_poll\_list])

　　return HttpResponse(output)

[复制代码](javascript:void(0);)

以上的通过硬编码的方式来实现视图,会让每次修改页面布局的时候,修改python代码,这样很麻烦.我们可以通过django的模板系统把页面布局和代码逻辑分开,让模板来负责页面布局工作.而视图就负责代码逻辑的实现.

首先在polls目录下创建叫templates的目录.django会在这里查找用到的模板.django settings.py中的配置项TEMPLATE\_LOADERS是一个包含需要用到的templates的位置列表.一个默认的模板加载器django.template.loaders.app\_directories.Loader,它会在INSTALLED\_APPS中注册的app目录中查找templates子目录,而不需要我们在TEMPLATE\_DIRS中手动添加.

模板的组织  
可以把所有的模板放在一个大的templates目录中,就像在教程2中的admin 的templates目录.但是,最好的方式还是每个app自己维护一个templates目录,这样的代码组织更清晰.

在刚刚创建的polls/templates目录下创建一个叫做polls的目录,在这个目录下创建index.html文件.(polls/templates/polls/index.html.如果了解了app\_directories模板加载器的工作原理,就可以知道通过polls/index.html就可以引用刚刚定义的index.html模板文件了.

可以看到引用polls/index.html就是使用polls/templates/polls/index.html,也就是说根据polls/templates这段路径不能有效区别不同app间的模板,所以需要在app下的模板目录下创建一个polls目录来区分不同app中相同名字的模板,比如在admin app中也有index.html,那么引用名是admin/index.html.这就是把给不同的app模板,定义一个不同的命名空间.

编辑polls/index.html,添加以下代码:

[复制代码](javascript:void(0);)

{% if latest\_poll\_list %}

<ul>

　　{% for poll in latest\_poll\_list %}

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

　　{% endfor %}

</ul>

{% else %}

　　<p>No polls are available.</p>

{% endif %}

[复制代码](javascript:void(0);)

现在更新一个index()视图,代码如下:

[复制代码](javascript:void(0);)

from django.http import HttpResponse

from django.template import RequestContext, loader

from polls.models import Poll

def index(request):

　　latest\_poll\_list = Poll.objects.order\_by('-pub\_date')[:5]

　　template = loader.get\_template('polls/index.html')

　　context = RequestContext(request, {

　　　　'latest\_poll\_list': latest\_poll\_list,

　　})

　　return HttpResponse(template.render(context))

[复制代码](javascript:void(0);)

这些代码加载了polls/index.html模板,传递给他一个上下文,这个上下文是一个字典,它把python对象与模板中的模板变量名一一对应.比如这里的latest\_poll\_list对象对应着模板中的latest\_poll\_list变量.通过访问/polls/(这里略去域名部分http://127.....:8080/),即可看到我们定义的视图了.

上面使用的是常用的传递一个请求上下文,然后返回包含渲染过后的模板的响应.django还提供了更加简单的渲染方式.  
编辑index.html,代码如下:

[复制代码](javascript:void(0);)

from django.shortcuts import render

from polls.models import Poll

def index(request):

　　latest\_poll\_list = Poll.objects.all().order\_by('-pub\_date')[:5]

　　context = {'latest\_poll\_list': latest\_poll\_list}

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

[复制代码](javascript:void(0);)

提醒以下,一旦我们使用上面的方式渲染模板就不需要import loader和RequestContext了,但是还需要保存HttpResponse,因为detais其他三个视图还是使用传统的方式工作的.

抛出404错误  
现在编辑details视图,代码:

[复制代码](javascript:void(0);)

from django.http import Http404

from django.shortcuts import render

from polls.models import Poll

# ...

def detail(request, poll\_id):

　　try:

　　　　poll = Poll.objects.get(pk=poll\_id)

　　except Poll.DoesNotExist:

　　　　raise Http404

　　return render(request, 'polls/detail.html', {'poll': poll})

[复制代码](javascript:void(0);)

这里引入一个新的内容,如果访问的poll不存在就抛出一个Http404的异常.  
让detail模板工作的最快方式,编辑polls/detail.html,代码:

{{ poll }}

引入404异常更快捷的方式:get\_object\_or\_404(),代码:

[复制代码](javascript:void(0);)

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

from polls.models import Poll

# ...

def detail(request, poll\_id):

　　poll = get\_object\_or\_404(Poll, pk=poll\_id)

　　return render(request, 'polls/detail.html', {'poll': poll})

[复制代码](javascript:void(0);)

这个get\_object\_or\_404()函数第一个参数是django模型,还有任意数量的关键字参数,这些参数前面的实现方式中传递个get()的一样.

设计思想:  
为什么使用get\_object\_or\_404()而不是,ObjectDoesNotExist异常或者使用model API来抛出Http404异常.这样做是为了减少model层和view层的耦合度.django的设计目标之一是松耦合.更多关于松耦合,关注django.shortcuts模块.

还有类似功能的get\_list\_or\_404()函数,它是在使用filter()时,如果得到的list为空就抛出Http404异常

编辑detail.html,让它更好看一点,代码:

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

<ul>

　　{% for choice in poll.choice\_set.all %}

　　<li>{{ choice.choice\_text }}</li>

　　{% endfor %}

</ul>

模板使用点操作访问对象中的变量属性.以上面代码中的{{ poll.question }}为例,首先djangp会使用字典的键值访问来查找poll对象中键名为question的值,如果不存在,就会按照对象的数据属性访问方式来查找,在这个例子中,查找成功了得到了poll对象中的question属性的值,如果再失败,就会使用列表索引的方式查找了.更多关于模板的内容,参照:https://docs.djangoproject.com/en/1.6/topics/templates/

移除硬编码的url.  
编辑index.html,硬编码的代码如下:

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

硬编码存在一个问题就是,一旦修改访问index视图的请求url,就需要在这些模板中逐个修改这些硬编码的url.然而,如果你在URLconf中的url函数中定义了name属性,那么你在模板里可以使用name属性值替代之前的硬编码url.因为之前在定义detail视图的url为url(r'^(?P<poll\_id>\d+)/$', views.detail, name='detail'),所以这里吧/polls/编程 url 'detail',具体的代码:

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

注意:如果detail使用单引号出错,而去掉单引号就可以,那么你使用版本小于django1.5,可以通过在模板的前面加入代码,来解决:

{% load url from future %}

现在便利来了!如果你想要通过polls/specifics/1/来访问detail,那么你不需要修改模板,直接修改poll/urls.py,代码:

url(r'^specifics/(?P<poll\_id>\d+)/$', views.detail, name='detail'),

就可以了.

如果有多个app,django是怎么区分那些请求url是访问哪个app的视图.比如,polls app有一个detail视图,还有另外一个app 也有detail视图.django 通过使用{% url %}标签的方式,怎样区别不同的app访问.答案就是在跟URLconf中为不同的app定义一个命名空间.

注意:这个本人有点疑惑

贴实现代码:

[复制代码](javascript:void(0);)

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

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

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

)

[复制代码](javascript:void(0);)

相应的,需要修改index.html

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

改成:

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

写一个简单的表单  
更新detail.html,代码:

[复制代码](javascript:void(0);)

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

{% if error\_message %}<p><strong>{{ error\_message }}</strong></p>{% endif %}

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

{% csrf\_token %}

{% for choice in poll.choice\_set.all %}

<input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />

<label for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</label><br />

{% endfor %}

<input type="submit" value="Vote" />

</form>

[复制代码](javascript:void(0);)

解释一下上面的代码:  
1 上面代码为poll对象的每个choice对应一个单选按钮,每个单选按钮的value就是choice的id号,name为choice,也就是说,按下提交按钮,发送的POST数据是这样的choice=3.  
2 设置表单的action为{% url 'polls:vote' poll.id %},method为post.如果需要传递数据的话建议使用post方法.

3 forloop.counter 是for循环计数器

4 这里使用了POST表单,存在跨站点请求伪造的安全问题.值得庆幸的是,不需要你为此担心,django自带一个简单易用的拦截系统,那就是,所有的POST表单都应该使用{% csrf\_token %}模板标签.现在,创建一个django试图可以处理这个提交上去的数据.

还记得在polls/urls.py中,这条语句吗?

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

现在创建一个真正的投票功能试图vote(),编辑polls/views.py,代码:

[复制代码](javascript:void(0);)

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

from django.http import HttpResponseRedirect, HttpResponse

from django.core.urlresolvers import reverse

from polls.models import Choice, Poll

# ...

def vote(request, poll\_id):

　　p = get\_object\_or\_404(Poll, pk=poll\_id)

　　try:

　　　　selected\_choice = p.choice\_set.get(pk=request.POST['choice'])

　　except (KeyError, Choice.DoesNotExist):

　　# Redisplay the poll voting form.

　　return render(request, 'polls/detail.html', {

　　　　'poll': p,

　　　　'error\_message': "You didn't select a choice.",

　　})

　　else:

　　selected\_choice.votes += 1

　　selected\_choice.save()

　　# Always return an HttpResponseRedirect after successfully dealing

　　# with POST data. This prevents data from being posted twice if a

　　# user hits the Back button.

　　return HttpResponseRedirect(reverse('polls:results', args=(p.id,)))

[复制代码](javascript:void(0);)

以上代码包含一些内容在之前没有提到过的:  
1 request.POST是一个类似与字典的对象,可以通过键名访问到提交的数据,在这里request.POST['choice']返回被选择的choice的id.

注意:django同样也提供了获取get方法提交的数据的request.GET.

2 如果使用的键名不存在,会抛出KeyError的异常,上面的代码已经对这种异常进行处理了:重定向到提交数据的页面,同时显示错误信息.

3 增加相应choice的votes值之后,保存.并返回一个HttpResponseRedirect,而不是HttpResponse,这样会把显示页面重定向到第一个参数指定的url那里.

4 使用POST提交数据成功之后,都应该使用HttpResponseRedirect,这是web最佳实践,不仅仅适用于django.为什么?自己比较一下两者之间的不同吧

5 在HttpResponseRedirect中使用了reverse,这个函数避免的使用url硬编码,允许想模板中那样使用想polls:results这样的url表示方式.reverse返回的字符串类似这样的形式:  
'/polls/3/results/'  
这个3是p.id的值,这个请求url,会调用results视图来显示最终的页面.  
更多关于HttpRequest对象的内容,阅读:https://docs.djangoproject.com/en/1.6/ref/request-response/

现在编辑results视图,代码:

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

def results(request, poll\_id):

　　poll = get\_object\_or\_404(Poll, pk=poll\_id)

　　return render(request, 'polls/results.html', {'poll': poll})

以上代码几乎与detail视图,一模一样,稍后会解决这个冗余问题  
现在,编辑results.html模板,代码:

[复制代码](javascript:void(0);)

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

<ul>

{% for choice in poll.choice\_set.all %}

<li>{{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>

{% endfor %}

</ul>

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

[复制代码](javascript:void(0);)

现在访问/polls/1/,然后投票,你会看到每次投票results结果都要更新.如果你不选任何一个choice,然后提交,会看到错误信息.

使用通用模板,减少代码冗余  
上面创建的detail(),results(),index()都是那么的相似.这些视图都是根据请求url从数据库中获取数据,然后加载一个模板,然后渲染之后的模板.因此django提供了"generic views"(下面称"通用视图")系统来简化模板的开发.减少代码冗余.

通用视图对通用模式进行抽象,所以你甚至不需要编写一行代码也可以创建一个app.

现在在poll应用中使用通用视图系统,这个就可以把重复的代码去掉了.使用步骤:  
1 转换URLconf  
2 删除旧的,不必要的视图  
3 新的视图使用通用视图  
为什么要梳理代码:  
通常你在创建一个django app时,会分析通用视图是否能够可以解决你的问题.然后从一开始就使用它,而不是中途重构代码时才引入.本系列教程先是关注如何创建一个app,现在开始注重编程的核心概念.也就是说,你需要有基本的数学基础,才能使用一个计算器.

修改URLconf  
编辑polls/urls.py,代码:

[复制代码](javascript:void(0);)

from django.conf.urls import patterns, url

from polls import views

urlpatterns = patterns('',

　　url(r'^$', views.IndexView.as\_view(), name='index'),

　　url(r'^(?P<pk>\d+)/$', views.DetailView.as\_view(), name='detail'),

　　url(r'^(?P<pk>\d+)/results/$', views.ResultsView.as\_view(), name='results'),

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

)

[复制代码](javascript:void(0);)

修改视图  
接下来,移除旧的index,detail,results视图,使用django的通用视图代替.编辑polls/views.py,代码:

[复制代码](javascript:void(0);)

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

from django.http import HttpResponseRedirect

from django.core.urlresolvers import reverse

from django.views import generic

from polls.models import Choice, Poll

class IndexView(generic.ListView):

　　template\_name = 'polls/index.html'

　　context\_object\_name = 'latest\_poll\_list'

　　def get\_queryset(self):

　　"""Return the last five published polls."""

　　　　return Poll.objects.order\_by('-pub\_date')[:5]

class DetailView(generic.DetailView):

　　model = Poll

　　template\_name = 'polls/detail.html'

class ResultsView(generic.DetailView):

　　model = Poll

　　template\_name = 'polls/results.html'

def vote(request, poll\_id):

....

[复制代码](javascript:void(0);)

上面的代码使用了两个通用视图:ListView和DetailView,这两个视图的含义分别是显示对象列表和显示具体的对象.  
每个通用视图都需要知道它将对哪个模型进行操作.  
DetailView通用视图期望从URL中获取名为"pk"的主键值，因此我们将poll\_id变成pk。  
默认情况下，DetailView使用的模板命名规则为<app name>/<model name>\_detail.html.它对应这里的模板名为“polls/poll\_detail.html"。template\_name属性是用于告诉django使用指定的模板名，而不是自动生成的默认模板名。在这里也指定了results list view的模板名——确保它们按照自定义的方式渲染视图。  
同样的，ListView通用视图，使用默认的模板名规则<app name>/<model name>\_list.html.这里使用了template name属性指定了一个模板名“polls/index.html”  
在前面的教程中，模板已经提供了包含poll，latest\_poll\_list上下文变量的上下文。对于DetailView，poll变量是自动提供的-因为正在使用Poll django 模型，而Django能够为上下文变量定义合适的名称。然而，对于ListView，自动生成的上下文变量叫“poll\_list”。不过可以通过给context\_object\_name属性赋值为你想要使用的名字“latest\_poll\_list”,另外一种解决方案就是把模板中的latest\_poll\_list改成poll\_list,但是第一种方法更加简单和方便。  
运行服务器，使用经过通用视图重构过的poll app。  
通用视图的更多细节，阅读：https://docs.djangoproject.com/en/1.6/topics/class-based-views/

django 教程5

结束了教程4之后，我们已经建立了一个网络投票应用程序，现在为该应用创建自动化测试

引入自动化测试

什么是自动化测试？

测试是用于检验代码运行正确行的简单程序  
测试也分成多种级别。一些测试只用于检查某个细节（比如：某个模型方法是否返回期望得到的值），而还有一些用于测试完整的软件的运行（用户在网站上的输入序列是否会生成期望的结果）。这些测试方法跟教程1中的测试方法没什么两样，使用shell来检验方法的执行是否正确或者运行一个应用然后输入数据检查输出的结果是否与自己期望的输出一致。

那么自动化测试与之前的测试有那些不同呢？那就是，你只需要根据被测试的程序创建一次测试用例，即使之后你修改了代码，还是可以使用前面创建好的测试用力来测试你的代码。而不在需要进行浪费时间的手动测试。

为什么需要编写测试用例

为什么要编写测试用例，为什么是现在？  
你可能觉得仅仅学习python/django就够你受的了，居然还要学那些看似必不可少但有可能是无用武之地的东西。毕竟，我们的投票应用程序一直都能很好的工作。那么麻烦的创建为它创建自动化测试也不会让它变得更好。没错，如果自动化测试是完成投票应用的最后一步，那就没有创建自动化测试的必要了。但是，事实并非如此，现在才是创建自动化测试的最佳时机。

测试可以节省你的时间

某种程度来说，“检查程序是否正常工作”是一个不错的测试方案。在更加复杂的应用程序中，那里有几十个组件之间复杂的相互作用。

对组件有任何的改动都会对程序的执行产生意想不到的后果。“检查程序是否正常工作”意味着对应用程序的各个部分的功能分别通过20组不同的数据作为输入来进行测试，以上这些测试仅仅是用于保证你的改动没有错误，这不是测试的意义所在。

不可否认，自动化测试能够在几秒内帮你做到定位程序异常的位置。  
有时，测试就像一个苦差事把你从具有创造性的编程工作中脱离出来，让你陷入编写测试用例，测试代码准确性的乏味无趣的工作中。然而，编写测试用例来测试程序定位问题比手动测试和定位新引入问题的位置效率更高。

测试不仅仅是问题定位，还能防止错误出现

把测试作为软件开发中消极的一面，认为测试只是把存在的bug找出来，是错误的想法。  
没有测试，应用程序的目的或者意图会变得更加难以理解。即使是自己编写的代码，有时候也要花很长的时间才能明白那些代码究竟是做什么的。  
测试让这种状况得到改善。测试代码会在被测试代码出现错误是标记出出现错误的部分，即使连你自己都没有意识到哪里出错了。

测试让你的代码更有吸引力

也许你实现的那一部分代码很牛逼，但是你会发现其他的开发者会直接了当的拒绝阅读你的代码，以为它们缺少测试代码。没有测试代码，其他的开发者是不会信任你所创造出来的代码有多么多么的好。 Jacob Kaplan-Moss说过，“没有测试的代码从设计开始就已经失败了”（也就是说代码的测试从软件设计开始就要考虑的）

其他的开发者在真正使用你的代码之前会先测试你的代码，这是编写测试的另外一个原因。

测试可以帮助团队更好的合作

前面的观点是个人项目开发者的观点，但是如果是由团队来维护一个复杂项目，测试确保你的代码不会被同事破坏（当然你不能在他们毫不知情的情况下破坏他们的代码）。要想成为一个django程序员，你就必须善于编写测试用例。

基本的测试理论

编写测试的方法有很多  
有些程序员遵循“测试驱动开发”的开发原则，在写代码之前编写测试用例。咋一看，似乎有点本末倒置，但是事实上这种做法与很多人的开发方式是类似的：先描述问题，然后编写代码解决问题。测试驱动开发则是使用测试用例来简单描述问题。

很多时候，对编写测试的新手，他们会先编写一点代码，然后才去编写代码的测试用例。也许越早设计测试用例越好，但是只要开始编写用例就永远也不算晚。  
有时很难决定应该什么时候编写测试用例。但是如果你已经编写了成千上万行代码的时候，才选择去编写测试用例，这不是一件简单的事情。在这种情况下，写下第一个测试用例，那么在下一次对代码进行修改（不论是添加新的功能还是解bug）都会让你获益多多。  
那么，让我开始编写测试用例吧！

写下第一个测试用例

定位bug

幸运的是，投票应用程序中有小小的bug，测试用例就有了用武之地了：在Poll.was\_published\_recently()方法，不论投票是最近一天发布（True）的还是未来的某天发布（期望返回False）的都会返回True。  
你可以在管理站点，创建一个发布时间为未来某天的投票，会发现这条投票会出现在最近发布投票的列表中。  
你也可以使用shell看到这个bug：

[复制代码](javascript:void(0);)

>>> import datetime

>>> from django.utils import timezone

>>> from polls.models import Poll

>>> # create a Poll instance with pub\_date 30 days in the future

>>> future\_poll = Poll(pub\_date=timezone.now() + datetime.timedelta(days=30))

>>> # was it published recently?

>>> future\_poll.was\_published\_recently()

True

[复制代码](javascript:void(0);)

因为未来某天并不在最近的范畴里，这显然是bug。

创建测试用例把bug暴露出来

前面在shell中的bug暴露的过程就是我们编写测试用例的要做的事情，让我们把这个测试用例写出来。  
通常测试用例都会保存在应用的tests.py文件中，django的测试系统会自动的找到这个所有以test开头的文件里面的测试用例并执行。

把下面的代码，写到polls应用的tests.py文件中：

[复制代码](javascript:void(0);)

import datetime

from django.utils import timezone

from django.test import TestCase

from polls.models import Poll

class PollMethodTests(TestCase):

　　def test\_was\_published\_recently\_with\_future\_poll(self):

　　　　"""

　　　　was\_published\_recently() should return False for polls whose

　　　　pub\_date is in the future

　　　　"""

　　　　future\_poll = Poll(pub\_date=timezone.now() + datetime.timedelta(days=30))

　　　　self.assertEqual(future\_poll.was\_published\_recently(), False)

[复制代码](javascript:void(0);)

上面我们创建了一个django测试用例。定义了TestCase子类PollMethodTests，在它的test\_was\_published\_recently\_with\_future\_poll方法中创建一个poll实例，初始化pub\_date的值为未来的第30天，然后检查poll实例的was\_published\_recently()方法返回值是否等于False。

运行测试用例

在终端中，输入：

$ python manage.py test polls

你会看到类似以下的输出：

[复制代码](javascript:void(0);)

Creating test database for alias 'default'...

F

======================================================================

FAIL: test\_was\_published\_recently\_with\_future\_poll (polls.tests.PollMethodTests)

----------------------------------------------------------------------

Traceback (most recent call last):

File "/path/to/mysite/polls/tests.py", line 16, in test\_was\_published\_recently\_with\_future\_poll

self.assertEqual(future\_poll.was\_published\_recently(), False)

AssertionError: True != False

----------------------------------------------------------------------

Ran 1 test in 0.001s

FAILED (failures=1)

Destroying test database for alias 'default'...

[复制代码](javascript:void(0);)

这些输出到底是怎么来的呢？  
1、python manage.py test polls 这个命令会找到polls应用中的测试用例  
2、接着，找到django.test.TestCase的子类的定义，这里是PollMethodTests  
3、然后，创建一个专门用于测试的数据库文件  
4、这时，在测试用例子类中找到以test开头的方法（这里是：test\_was\_published\_recently\_with\_future\_poll），然后执行它  
5、最后，根据执行结果输出测试信息

上面的测试信息告诉我们那个测试用例运行失败了，还把导致失败的语句行号报告出来。

修复bug

知道了bug：如果pub\_date>的值是未来某天，Poll.was\_published\_recently()应该返回False。在models.py把bug修复，让它在pub\_date仅为过去时间时，才返回True。代码如下：

def was\_published\_recently(self):

　　now = timezone.now()

　　return now - datetime.timedelta(days=1) <= self.pub\_date < now

再运行一下测试代码：

[复制代码](javascript:void(0);)

Creating test database for alias 'default'...

.

----------------------------------------------------------------------

Ran 1 test in 0.001s

OK

Destroying test database for alias 'default'...

[复制代码](javascript:void(0);)

在确定了bug之后，编写测试用例暴露出bug，然后在代码中修复bug，最后代码测试通过。  
也许未来某一天我们的应用还会出现bug，但是不再会是这个bug。因为一旦出现同样的bug，测试用例会给出马上给出警告。这一部分一小段程序防止bug重复出现为程序的安全提供了永久的保障。

更全面的测试

还可以对was\_published\_recently()方法进行更多的测试。事实上，会出现修复一个bug后又引入新的bug的尴尬情况。因此为was\_published\_recently()方法再添加两个测试方法，来对它进行更加全面的测试。  
代码如下：

[复制代码](javascript:void(0);)

def test\_was\_published\_recently\_with\_old\_poll(self):

　　"""

　　was\_published\_recently() should return False for polls whose pub\_date

　　is older than 1 day

　　"""

　　old\_poll = Poll(pub\_date=timezone.now() - datetime.timedelta(days=30))

　　self.assertEqual(old\_poll.was\_published\_recently(), False)

def test\_was\_published\_recently\_with\_recent\_poll(self):

　　"""

　　was\_published\_recently() should return True for polls whose pub\_date

　　is within the last day

　　"""

　　recent\_poll = Poll(pub\_date=timezone.now() - datetime.timedelta(hours=1))

　　self.assertEqual(recent\_poll.was\_published\_recently(), True)

[复制代码](javascript:void(0);)

现在我们有三个测试方法来确认Poll.was\_published\_recently()方法对过去，最近，未来三种不同的投票发布时间有正确对应的正确的返回值。  
再强调一次，polls是一个简单的应用，但是无论未来它变得多么复杂或者它会和什么样的代码交互，我们都为它的每个方法编写了测试用例，确保它正确运行，得到期望的结果。

视图测试

这个polls应用有点不合理：它可以发布任何投票，包括发布时间是未来的某天的投票。这是个bug，应该修复。设置pub\_date为未来某天的投票理应在pub\_date那天之前是不可见状态的。

一个视图的测试

在修复前面的models中的bug的步骤是先编写测试代码，然后运行测试代码来定位bug，最后修复它。事实上，那是一个简单的测试驱动开发的例子，但是到底按照什么样的步骤来实施并不要紧。  
在我们第一个测试中，我们仅仅围绕着代码的内部逻辑在进行。在这次测试，我们要检查代码通过浏览器展现给用户的内容是否与我们的期望一致。  
在修复bug之前，我们先搞清楚自己掌握了那些工具。

django测试客户端

django提供一个测试客户端在视图级别上来模拟用户与代码的交互。可以在tests.py或者shell中使用它。  
我们又可以从shell开始，在这里我们需要先做两件在tests.py中不需要做的事情。第一件是设置测试环境：

>>> from django.test.utils import setup\_test\_environment

>>> setup\_test\_environment()

setup\_test\_environment()安装一个模板渲染器，允许我们来测试response额外的属性。否则上下文无法使用。注意这个方法还没有安装测试数据库，因此需要再一次运行前面已经创建了的测试数据库。输出信息会因为所创建的polls实例而不同。  
接下来，我们需要导入test 客户端类（后面tests.py文件中我们将使用django.test.TestCase类，它本身自带客户端类，因此不需要额外导入客户端）。

>>> from django.test.client import Client

>>> # create an instance of the client for our use

>>> client = Client()

做好准备后，我们可以让定义好的client为我们做点什么：

[复制代码](javascript:void(0);)

>>> # get a response from '/'

>>> response = client.get('/')

>>> # we should expect a 404 from that address

>>> response.status\_code

404

>>> # on the other hand we should expect to find something at '/polls/'

>>> # we'll use 'reverse()' rather than a harcoded URL

>>> from django.core.urlresolvers import reverse

>>> response = client.get(reverse('polls:index'))

>>> response.status\_code

200

>>> response.content

'\n\n\n <p>No polls are available.</p>\n\n'

>>> # note - you might get unexpected results if your ``TIME\_ZONE``

>>> # in ``settings.py`` is not correct. If you need to change it,

>>> # you will also need to restart your shell session

>>> from polls.models import Poll

>>> from django.utils import timezone

>>> # create a Poll and save it

>>> p = Poll(question="Who is your favorite Beatle?", pub\_date=timezone.now())

>>> p.save()

>>> # check the response once again

>>> response = client.get('/polls/')

>>> response.content

'\n\n\n <ul>\n \n <li><a href="/polls/1/">Who is your favorite Beatle?</a></li>\n \n </ul>\n\n'

>>> response.context['latest\_poll\_list']

[<Poll: Who is your favorite Beatle?>]

[复制代码](javascript:void(0);)

改善我们的视图

投票列表把还没有发布的投票也显示出来了。（比如，那些pub\_date是将来时间的投票）。让我们把它修复。  
在教程4中，我们使用了通用视图，IndexView类是基于ListView的：

[复制代码](javascript:void(0);)

class IndexView(generic.ListView):

　　template\_name = 'polls/index.html'

　　context\_object\_name = 'latest\_poll\_list'

　　def get\_queryset(self):

　　　　"""Return the last five published polls."""

　　　　return Poll.objects.order\_by('-pub\_date')[:5]

[复制代码](javascript:void(0);)

response.context\_data['latest\_poll\_list']提取出视图在上下文中设置的数据。  
修改get\_queryset方法，使它也可以与当前时间比较来检查时间。  
首先需要导入：

from django.utils import timezone

接着把get\_queryset方法改成如下样子：

def get\_queryset(self):

　　"""

　　Return the last five published polls (not including those set to be

　　published in the future).

　　"""

　　return Poll.objects.filter(pub\_date\_\_lte=timezone.now()).order\_by('-pub\_date')[:5]

Poll.objects.filter(pub\_date\_\_lte=timezone.now())返回一个queryset，它包含pub\_date小于或者等于timezone.now的Polls。

开始测试我们的视图

基于前面的shell会话，我们来创建自己的视图测试实例：  
在polls/tests.py中添加代码：

from django.core.urlresolvers import reverse

我们将使用工厂方法来创建poll实例和测试类。

[复制代码](javascript:void(0);)

def create\_poll(question, days):

　　"""

　　Creates a poll with the given `question` published the given number of

　　`days` offset to now (negative for polls published in the past,

　　positive for polls that have yet to be published).

　　"""

　　return Poll.objects.create(question=question,

　　pub\_date=timezone.now() + datetime.timedelta(days=days))

class PollViewTests(TestCase):

　　def test\_index\_view\_with\_no\_polls(self):

　　　　"""

　　　　If no polls exist, an appropriate message should be displayed.

　　　　"""

　　　　response = self.client.get(reverse('polls:index'))

　　　　self.assertEqual(response.status\_code, 200)

　　　　self.assertContains(response, "No polls are available.")

　　　　self.assertQuerysetEqual(response.context['latest\_poll\_list'], [])

　　def test\_index\_view\_with\_a\_past\_poll(self):

　　　　"""

　　　　Polls with a pub\_date in the past should be displayed on the index page.

　　　　"""

　　　　create\_poll(question="Past poll.", days=-30)

　　　　response = self.client.get(reverse('polls:index'))

　　　　self.assertQuerysetEqual(response.context['latest\_poll\_list'],['<Poll: Past poll.>'])

　　def test\_index\_view\_with\_a\_future\_poll(self):

　　　　"""

　　　　Polls with a pub\_date in the future should not be displayed on the

　　　　index page.

　　　　"""

　　　　create\_poll(question="Future poll.", days=30)

　　　　response = self.client.get(reverse('polls:index'))

　　　　self.assertContains(response, "No polls are available.", status\_code=200)

　　　　self.assertQuerysetEqual(response.context['latest\_poll\_list'], [])

　　def test\_index\_view\_with\_future\_poll\_and\_past\_poll(self):

　　　　"""

　　　　Even if both past and future polls exist, only past polls should be

　　　　displayed.

　　　　"""

　　　　create\_poll(question="Past poll.", days=-30)

　　　　create\_poll(question="Future poll.", days=30)

　　　　response = self.client.get(reverse('polls:index'))

　　　　self.assertQuerysetEqual(

　　　　response.context['latest\_poll\_list'],['<Poll: Past poll.>'])

　　def test\_index\_view\_with\_two\_past\_polls(self):

　　　　"""

　　　　The polls index page may display multiple polls.

　　　　"""

　　　　create\_poll(question="Past poll 1.", days=-30)

　　　　create\_poll(question="Past poll 2.", days=-5)

　　　　response = self.client.get(reverse('polls:index'))

　　　　self.assertQuerysetEqual(response.context['latest\_poll\_list'],['<Poll: Past poll 2.>', '<Poll: Past poll 1.>'])

[复制代码](javascript:void(0);)

让我们深入探讨上面的代码。第一个是poll的工厂方法，create\_poll,把创建poll实例的重复步骤抽离出来。test\_index\_view\_with\_no\_polls没有创建任何的poll实例，但是检查程序是否输出消息“No polls are available”和验证latest\_poll\_list是否为空。注意：django.test.TestCase类提供了一些额外的断言方法。在这里我们用到了assertContains()和assertQuerysetEqual()。  
在test\_index\_view\_with\_a\_past\_poll中，我们创建一个poll实例然后验证它是否会在poll列表中出现。  
在test\_index\_view\_with\_a\_future\_poll中，同样创建一个poll，pub\_date的时间为将来时。在每个测试方法中，数据库都会重置，以使前面方法创建poll不会出现的数据库里，这时候数据库应该是空的。  
等等。实际上，我们使用测试讲了一个故事，关于管理员输入了那些数据还有用户使用我们的网站看到了什么。对于检查系统每一个状态以及状态的变化的结果都是可见的。

测试DetailView

我们的代码工作正常。但是，即使future polls(pub\_date is future)，没有在index视图中显示出来，但是用户也可以通过正确的url访问到它们。所以需要在DetailView中对此加以约束。

[复制代码](javascript:void(0);)

class DetailView(generic.DetailView):

　　...

　　def get\_queryset(self):

　　　　"""

　　　　Excludes any polls that aren't published yet.

　　　　"""

　　return Poll.objects.filter(pub\_date\_\_lte=timezone.now())

[复制代码](javascript:void(0);)

当然，我们还要编写一些测试代码，来检查一个pub\_date is past的Poll能否正确显示，而pub\_date is future是否不能被访问到。

[复制代码](javascript:void(0);)

class PollIndexDetailTests(TestCase):

　　def test\_detail\_view\_with\_a\_future\_poll(self):

　　　　"""

　　　　The detail view of a poll with a pub\_date in the future should

　　　　return a 404 not found.

　　　　"""

　　　　future\_poll = create\_poll(question='Future poll.', days=5)

　　　　response = self.client.get(reverse('polls:detail', args=(future\_poll.id,)))

　　　　self.assertEqual(response.status\_code, 404)

　　def test\_detail\_view\_with\_a\_past\_poll(self):

　　　　"""

　　　　The detail view of a poll with a pub\_date in the past should display

　　　　the poll's question.

　　　　"""

　　　　past\_poll = create\_poll(question='Past Poll.', days=-5)

　　　　response = self.client.get(reverse('polls:detail', args=(past\_poll.id,)))

　　　　self.assertContains(response, past\_poll.question, status\_code=200)

[复制代码](javascript:void(0);)

更多测试的ideas

我们应该为ResultsView视图添加类似与get\_queryset方法，然后为该视图创建一个新的测试类。这些工作与前面的有大量的重复。poll应用还有很多地方可以提高的，比如那些没有Choice的投票就不要显示出来。也许，登录状态的管理员可以查看没有发布的Polls，而不是普通的访问者。还是那句话：无论添加任何功能都需要附带对应的测试代码，而编写测试和编写代码次序无关紧要。有一天你会发现，测试代码一直在不断的膨胀。

测试代码越多越好

看起来我们的测试代码的增长失去了控制。按照这样的速度，它们就会超过正常代码的数量了。而且相对简洁优雅的正常代码来说，里面有很多重复和不美观的代码。  
没关系，让它们增长。在大多数情况下，你只需要测试一次然后忘掉它。在你后续的开发中，它也能很好的工作。有时，测试也需要升级。比如前面提到的改善建议（不显示没有choice的poll），会使前面编好的测试代码会执行失败，单同时也告诉我们那些测试需要升级。测试代码可以自检。  
还有最坏的情况，有时候你会发现很多测试代码是冗余的，但是你还是可以继续开发，完全不受影响。而且冗余是一件好事。  
只要你对测试代码合理安排，它们就不会变得乱糟糟。原理原则：  
1、一个model/view对应一个独立的TestClass类  
2、一个test method 对应一种你要测试的情况  
3、使用测试功能来命名test method