**自动化测试是什么？**[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#what-are-automated-tests)

测试，是用来检查代码正确性的一些简单的程序。

测试在不同的层次中都存在。有些测试只关注某个很小的细节（某个模型的某个方法的返回值是否满足预期？），而另一些测试可能检查对某个软件的一系列操作（*某一用户输入序列是否造成了预期的结果？*）。其实这和我们在 [教程第 2 部分](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial02/)，里做的并没有什么不同，我们使用 [**shell**](https://docs.djangoproject.com/zh-hans/2.1/ref/django-admin/#django-admin-shell) 来测试某一方法的功能，或者运行某个应用并输入数据来检查它的行为。

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

#### 测试将节约你的时间[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#tests-will-save-you-time)

在某种程度上，能够「判断出代码是否正常工作」的测试，就称得上是个令人满意的了。在更复杂的应用程序中，组件之间可能会有数十个复杂的交互。

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

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

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

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

## 开始写我们的第一个测试[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#writing-our-first-test)

### 首先得有个 Bug[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#we-identify-a-bug)

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

Confirm the bug by using the [**shell**](https://docs.djangoproject.com/zh-hans/2.1/ref/django-admin/#django-admin-shell) to check the method on a question whose date lies in the future:

/ 

**$** python manage.py shell

**>>> import** **datetime**

**>>> from** **django.utils** **import** timezone

**>>> from** **polls.models** **import** Question

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

**>>>** future\_question = Question(pub\_date=timezone.now() + datetime.timedelta(days=30))

**>>>** *# was it published recently?*

**>>>** future\_question.was\_published\_recently()

True

因为将来发生的是肯定不是最近发生的，所以代码明显是错误的。

### 创建一个测试来暴露这个 bug[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#create-a-test-to-expose-the-bug)

我们刚刚在 [**shell**](https://docs.djangoproject.com/zh-hans/2.1/ref/django-admin/#django-admin-shell) 里做的测试也就是自动化测试应该做的工作。所以我们来把它改写成自动化的吧。

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

将下面的代码写入 **polls** 应用里的 **tests.py** 文件内：

polls/tests.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id1)

**import** **datetime**

**from** **django.test** **import** TestCase

**from** **django.utils** **import** timezone

**from** **.models** **import** Question

**class** **QuestionModelTests**(TestCase):

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

*"""*

*was\_published\_recently() returns False for questions whose pub\_date*

*is in the future.*

*"""*

time = timezone.now() + datetime.timedelta(days=30)

future\_question = Question(pub\_date=time)

self.assertIs(future\_question.was\_published\_recently(), False)

我们创建了一个 [**django.test.TestCase**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.TestCase) 的子类，并添加了一个方法，此方法创建一个 **pub\_date** 时未来某天的 **Question** 实例。然后检查它的 **was\_published\_recently()** 方法的返回值——它 应该是 False。

### 运行测试[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#running-tests)

在终端中，我们通过输入以下代码运行测试:

/ 

**$** python manage.py test polls

你将会看到运行结果:

Creating test database **for** alias 'default'...

System check identified no issues (0 silenced).

F

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

FAIL: test\_was\_published\_recently\_with\_future\_question (polls.tests.QuestionModelTests)

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

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'...

发生了什么呢？以下是自动化测试的运行过程：

* **python manage.py test polls** 将会寻找 **polls** 应用里的测试代码
* 它找到了 [**django.test.TestCase**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.TestCase) 的一个子类
* 它创建一个特殊的数据库供测试使用
* 它在类中寻找测试方法——以 **test** 开头的方法。
* 在 **test\_was\_published\_recently\_with\_future\_question** 方法中，它创建了一个 **pub\_date** 值为 30 天后的 **Question** 实例。
* 接着使用 **assertls()** 方法，发现 **was\_published\_recently()** 返回了 **True**，而我们期望它返回 **False**。

测试系统通知我们哪些测试样例失败了，和造成测试失败的代码所在的行号。

### 修复这个 bug[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#fixing-the-bug)

我们早已知道，当 **pub\_date** 为未来某天时， **Question.was\_published\_recently()** 应该返回 **False**。我们修改 **models.py** 里的方法，让它只在日期是过去式的时候才返回 **True**：

polls/models.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id2)

**def** was\_published\_recently(self):

now = timezone.now()

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

然后重新运行测试:

Creating test database **for** alias 'default'...

System check identified no issues (0 silenced).

.

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

Ran 1 test **in** 0.001s

OK

Destroying test database **for** alias 'default'...

发现 bug 后，我们编写了能够暴露这个 bug 的自动化测试。在修复 bug 之后，我们的代码顺利的通过了测试。

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

### 更全面的测试[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#more-comprehensive-tests)

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

我们在上次写的类里再增加两个测试，来更全面的测试这个方法：

polls/tests.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id3)

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

*"""*

*was\_published\_recently() returns False for questions whose pub\_date*

*is older than 1 day.*

*"""*

time = timezone.now() - datetime.timedelta(days=1, seconds=1)

old\_question = Question(pub\_date=time)

self.assertIs(old\_question.was\_published\_recently(), False)

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

*"""*

*was\_published\_recently() returns True for questions whose pub\_date*

*is within the last day.*

*"""*

time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)

recent\_question = Question(pub\_date=time)

self.assertIs(recent\_question.was\_published\_recently(), True)

现在，我们有三个测试来确保 **Question.was\_published\_recently()** 方法对于过去，最近，和将来的三种情况都返回正确的值。

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

### 改善视图代码[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#improving-our-view)

现在的投票列表会显示将来的投票（ **pub\_date** 值是未来的某天)。我们来修复这个问题。

在 [教程的第 4 部分](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial04/) 里，我们介绍了基于 [**ListView**](https://docs.djangoproject.com/zh-hans/2.1/ref/class-based-views/generic-display/#django.views.generic.list.ListView) 的视图类：

polls/views.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id4)

**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]

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

polls/views.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id5)

**from** **django.utils** **import** timezone

然后我们把 **get\_queryset** 方法改写成下面这样：

polls/views.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id6)

**def** get\_queryset(self):

*"""*

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

*published in the future).*

*"""*

**return** Question.objects.filter(

pub\_date\_\_lte=timezone.now()

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

**Question.objects.filter(pub\_date\_\_lte=timezone.now())** returns a queryset containing **Question**s whose **pub\_date** is less than or equal to - that is, earlier than or equal to - **timezone.now**.

### 测试新视图[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#testing-our-new-view)

启动服务器、在浏览器中载入站点、创建一些发布时间在过去和将来的 **Questions** ，然后检验只有已经发布的 **Questions** 会展示出来，现在你可以对自己感到满意了。你不想每次修改可能与这相关的代码时都重复这样做 —— 所以让我们基于以上 [**shell**](https://docs.djangoproject.com/zh-hans/2.1/ref/django-admin/#django-admin-shell) 会话中的内容，再编写一个测试。

将下面的代码添加到 **polls/tests.py** ：

polls/tests.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id7)

**from** **django.urls** **import** reverse

然后我们写一个公用的快捷函数用于创建投票问题，再为视图创建一个测试类：

polls/tests.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id8)

**def** create\_question(question\_text, days):

*"""*

*Create a question with the given `question\_text` and published the*

*given number of `days` offset to now (negative for questions published*

*in the past, positive for questions that have yet to be published).*

*"""*

time = timezone.now() + datetime.timedelta(days=days)

**return** Question.objects.create(question\_text=question\_text, pub\_date=time)

**class** **QuestionIndexViewTests**(TestCase):

**def** test\_no\_questions(self):

*"""*

*If no questions exist, an appropriate message is displayed.*

*"""*

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

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

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

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

**def** test\_past\_question(self):

*"""*

*Questions with a pub\_date in the past are displayed on the*

*index page.*

*"""*

create\_question(question\_text="Past question.", days=-30)

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

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

**def** test\_future\_question(self):

*"""*

*Questions with a pub\_date in the future aren't displayed on*

*the index page.*

*"""*

create\_question(question\_text="Future question.", days=30)

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

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

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

**def** test\_future\_question\_and\_past\_question(self):

*"""*

*Even if both past and future questions exist, only past questions*

*are displayed.*

*"""*

create\_question(question\_text="Past question.", days=-30)

create\_question(question\_text="Future question.", days=30)

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

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

**def** test\_two\_past\_questions(self):

*"""*

*The questions index page may display multiple questions.*

*"""*

create\_question(question\_text="Past question 1.", days=-30)

create\_question(question\_text="Past question 2.", days=-5)

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

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question 2.>', '<Question: Past question 1.>']

)

让我们更详细地看下以上这些内容。

首先是一个快捷函数 **create\_question**，它封装了创建投票的流程，减少了重复代码。

**test\_no\_questions** 方法里没有创建任何投票，它检查返回的网页上有没有 "No polls are available." 这段消息和 **latest\_question\_list** 是否为空。注意到 [**django.test.TestCase**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.TestCase) 类提供了一些额外的 assertion 方法，在这个例子中，我们使用了 [**assertContains()**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.SimpleTestCase.assertContains) 和 [**assertQuerysetEqual()**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.TransactionTestCase.assertQuerysetEqual) 。

在 **test\_past\_question** 方法中，我们创建了一个投票并检查它是否出现在列表中。

在 **test\_future\_question** 中，我们创建 **pub\_date** 在未来某天的投票。数据库会在每次调用测试方法前被重置，所以第一个投票已经没了，所以主页中应该没有任何投票。

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

### 测试 DetailView[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#testing-the-detailview)

我们的工作似乎已经很完美了？不，还有一个问题：就算在发布日期时未来的那些投票不会在目录页 index 里出现，但是如果用户知道或者猜到正确的 URL ，还是可以访问到它们。所以我们得在 **DetailView** 里增加一些约束：

polls/views.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id9)

**class** **DetailView**(generic.DetailView):

...

**def** get\_queryset(self):

*"""*

*Excludes any questions that aren't published yet.*

*"""*

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

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

polls/tests.py[**¶**](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#id10)

**class** **QuestionDetailViewTests**(TestCase):

**def** test\_future\_question(self):

*"""*

*The detail view of a question with a pub\_date in the future*

*returns a 404 not found.*

*"""*

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\_past\_question(self):

*"""*

*The detail view of a question with a pub\_date in the past*

*displays the question's text.*

*"""*

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)

### 更多的测试思路[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#ideas-for-more-tests)

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

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

恩，也许你想让管理员能在目录上看见未被发布的那些投票，但是普通用户看不到。不管怎么说，如果你想要增加一个新功能，那么同时一定要为它编写测试。不过你是先写代码还是先写测试那就随你了。

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

## 当需要测试的时候，测试用例越多越好[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#when-testing-more-is-better)

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

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

但有时测试也需要更新。想象一下如果我们修改了视图，只显示有选项的那些投票，那么只前写的很多测试就都会失败。但这也明确地告诉了我们哪些测试需要被更新，所以测试也会测试自己。

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

如果你对测试有个整体规划，那么它们就几乎不会变得混乱。下面有几条好的建议：

* 对于每个模型和视图都建立单独的 **TestClass**
* 每个测试方法只测试一个功能
* 给每个测试方法起个能描述其功能的名字

## 深入代码测试[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#further-testing)

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

举个例子，在上述的测试中，我们已经从代码逻辑和视图响应的角度检查了应用的输出，现在你可以从一个更加 "in-browser" 的角度来检查最终渲染出的 HTML 是否符合预期，使用 Selenium 可以很轻松的完成这件事。这个工具不仅可以测试 Django 框架里的代码，还可以检查其他部分，比如说你的 JavaScript。它假装成是一个正在和你站点进行交互的浏览器，就好像有个真人在访问网站一样！Django 它提供了[**LiveServerTestCase**](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/tools/#django.test.LiveServerTestCase) 来和 Selenium 这样的工具进行交互。

如果你在开发一个很复杂的应用的话，你也许想在每次提交代码时自动运行测试，也就是我们所说的持续集成 [continuous integration](https://en.wikipedia.org/wiki/Continuous_integration) ，这样就能实现质量控制的自动化，起码是部分自动化。

一个找出代码中未被测试部分的方法是检查代码覆盖率。它有助于找出代码中的薄弱部分和无用部分。如果你无法测试一段代码，通常说明这段代码需要被重构或者删除。想知道代码覆盖率和无用代码的详细信息，查看文档 [Integration with coverage.py](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/advanced/#topics-testing-code-coverage) 获取详细信息。

文档 [Django 中的测试](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/) 里有关于测试的更多信息。

## 接下来要做什么？[¶](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial05/#what-s-next)

如果你想深入了解测试，就去看 [Django 中的测试](https://docs.djangoproject.com/zh-hans/2.1/topics/testing/) 。

当你已经比较熟悉测试 Django 视图的方法后，就可以继续阅读 [教程第 6 部分](https://docs.djangoproject.com/zh-hans/2.1/intro/tutorial06/) ，学习静态文件管理的相关知识。