編寫你的第一個 Django 應用，第 5 部分

這一篇從 教學第 4 部分 結尾的地方繼續講起。我們在前幾章成功的構建了一個在線投票應用，在這一部分裡我們將為它建立一些自動化測試。

從哪裡取得協助：

如果你在閱讀本教學的過程中有任何疑問，可以前往FAQ的:doc:Getting Help</faq/help> 的小節。

自動化測試簡介

自動化測試是什麼?

測試程式，是用來檢查你的程式能否正常執行的程式。

測試在不同的層次中都存在。有些測試只說明某個很小的細節（某個模型的某個方法的回傳值是否滿足預期？），而另一些測試可能檢查對某個軟體的一系列操作（*某一用戶輸入序列是否造成了預期的結果？*）。其實這和我們在 教學第 2 部分，裡做的並沒有什麼不同，我們使用 `shell` 來測試某一方法的功能，或者執行某個應用並輸入資料來檢查它的行為。

真正不同的地方在於，*自動化* 測試是由某個系統幫你自動完成的。當你建立好了一系列測試，每次修改應用程式後，就可以自動檢查出修改後的程式是否還像你曾經預期的那樣正常工作。你不需要花費大量時間來進行手動測試。

為什麼你需要寫測試

但是，為什麼需要測試呢？又為什麼是現在呢?

你可能覺得學 Python/Django 對你來說已經很滿足了，再學一些新東西的話看起來有點負擔過重並且沒什麼必要。畢竟，我們的投票應用看起來已經完美工作了。寫一些自動測試並不能讓它工作的更好。如果寫一個投票應用是你想用 Django 完成的唯一工作，那你確實沒必要學寫測試。但是如果你還想寫更複雜的專案，現在就是學習測試寫法的最好時機了。

測試將節約你的時間

在某種程度上，能夠「判斷出程式是否正常工作」的測試，就稱得上是個令人滿意的了。在更複雜的應用程式中，組件之間可能會有數十個複雜的交互。

對其中某一組件的改變，也有可能會造成意想不到的結果。判斷「程式是否正常工作」意味著你需要用大量的資料來完整的測試全部程式的功能，以確保你的小修改沒有對應用整體造成破壞 — 這太費時間了。

尤其是當你發現自動化測試能在幾秒鍾之內幫你完成這件事時，就更會覺得手動測試實在是太浪費時間了。當某人寫出錯誤的程式時，自動化測試還能協助你定位錯誤程式的位置。

有時候你會覺得，和富有創造性和生產力的業務程式比起來，編寫枯燥的測試程式實在是太無聊了，特別是當你知道你的程式完全沒有問題的時候。

然而，編寫測試還是要比花費幾個小時手動測試你的應用，或者為了找到某個小錯誤而胡亂翻看程式要有意義的多。

測試不僅能發現錯誤，而且能預防錯誤

「測試是開發的對立面」，這種思想是不對的。

如果沒有測試，整個應用的行為意圖會變得更加的不清晰。甚至當你在看自己寫的程式時也是這樣，有時候你需要仔細研讀一段程式才能搞清楚它有什麼用。

而測試的出現改變了這種情況。測試就好像是從內部仔細檢查你的程式，當有些地方出錯時，這些地方將會變得很顯眼 — *就算你自己沒有意識到那裡寫錯了*。

測試使你的程式更有吸引力

你也許遇到過這種情況：你編寫了一個絕贊的軟體，但是其他開發者看都不看它一眼，因為它缺少測試。沒有測試的程式不值得信任。 Django 最初開發者之一的 Jacob Kaplan-Moss 說過：“專案規劃時沒有包含測試是不科學的。”

其他的開發者希望在正式使用你的程式前先看你的程式的測試，這是你需要寫測試的另一個重要原因。

測試有利於團隊協作

前面的幾點都是從單人開發的角度來說的。複雜的應用可能由團隊維護。測試的存在保證了同事不會不小心破壞了了你的程式（也保證你不會不小心弄壞他們的）。如果你想作為一個 Django 程式員謀生的話，你必須擅長編寫測試!

基礎測試策略

有好幾種不同的方法可以寫測試。

一些開發者遵循 "測試驅動" 的開發原則，他們在寫程式之前先寫測試。這種方法看起來有點反直覺，但事實上，這和大多數人日常的做法是相吻合的。我們會先描述一個問題，然後寫程式來解決它。「測試驅動」的開發方法只是將問題的描述抽象為了 Python 的測試樣例。

更普遍的情況是，一個剛接觸自動化測試的新手更傾向於先寫程式，然後再寫測試。雖然提前寫測試可能更好，但是晚點寫並不算太遲。

有時候很難決定從哪裡開始下手寫測試。如果你才寫了幾千行 Python 程式，選擇從哪裡開始寫測試確實不怎麼簡單。如果是這種情況，那麼在你下次修改程式（例如加新功能，或者修正 Bug）之前寫個測試是比較合理且有效的。

所以，我們現在就開始寫吧。

開始寫我們的第一個測試

首先得有個 Bug

幸運的是，我們的 **`polls`** 應用現在就有一個小 bug 需要被修正：我們的要求是如果 Question 是在一天之內發佈的，**`Question.was_published_recently()`** 方法將會回傳 **`True`**，然而現在這個方法在 **`Question`** 的 **`pub_date`** 欄位比當前時間還晚時也會回傳 True（這是個 Bug）。

用**`shell`**命令確認一下這個方法的日期的 bug：

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

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # 建立一個 pub_date 在未來 30 天的 Question 實例
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # 是最近發佈的嗎？?
>>> future_question.was_published_recently()
True
```

因為將來發生的是肯定不是最近發生的，所以程式明顯是錯誤的。

建立一個測試來暴露這個 bug

我們剛剛在 **shell** 裡做的測試也就是自動化測試應該做的工作。所以我們來把它改寫成自動化的吧。

按照慣例，Django 應用的測試應該寫在應用的 **tests.py** 文件裡。測試系統會自動的在所有以 **tests** 開頭的文件裡尋找並執行測試程式。

將下面的程式寫入 **polls** 應用裡的 **tests.py** 文件內：

polls/tests.py

```
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() 對於 questions 的 pub_date 是在未來
        會回傳 False。
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

我們建立了一個 **django.test.TestCase** 的子類別，並增加了一個方法，此方法建立一個 **pub_date** 時未來某天的 **Question** 實例。然後檢查它的 **was_published_recently()** 方法的回傳值 – 它 *應該* 是 False。

執行測試

在終端中，我們透過輸入以下程式執行測試:

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

你將會看到執行結果:

```
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'...
```

不一樣的錯誤?

若在此處你得到了一個 **NameError** 錯誤，你可能漏了 第二步 中將 **datetime** 和 **timezone** 匯入 **polls/model.py** 的步驟。複制這些語句，然後試著重新執行測試。

發生了什麼呢? 以下是自動化測試的執行過程:

- **python manage.py test polls** 將會尋找 **polls** 應用裡的測試程式
- 它找到了 **django.test.TestCase** 的一個子類別
- 它建立一個特殊的資料庫供測試使用
- 它在類別中尋找測試方法 – 以 **test** 開頭的方法。
- 在 **test_was_published_recently_with_future_question** 方法中，它建立了一個 **pub_date** 值為 30 天後的 **Question** 實例。
- 接著使用 **assertIs()** 方法，發現 **was_published_recently()** 回傳了 **True**，而我們期望它回傳 **False**。

測試系統通知我們哪些測試樣例失敗了，和造成測試失敗的程式所在的行號。

修正這個 bug

我們早已知道，當 **pub_date** 為未來某天時， **Question.was_published_recently()** 應該回傳 **False**。我們修改 **models.py** 裡的方法，讓它只在日期是過去式的時候才回傳 **True**:

polls/models.py

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

然後重新執行測試:

```
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'...
```

發現 bug 後，我們編寫了能夠暴露這個 bug 的自動化測試。在修正 bug 之後，我們的程式順利的透過了測試。

將來，我們的應用可能會出現其他的問題，但是我們可以肯定的是，一定不會再次出現這個 bug，因為只要執行一遍測試，就會立刻收到警告。我們可以認為應用的這一小部分程式永遠是安全的。

更全面的測試

我們已經搞定一小部分了，現在可以考慮全面的測試 **was_published_recently()** 這個方法以確定它的安全性，然後就可以把這個方法穩定下來了。事實上，在修正一個 bug 時不小心引入另一個 bug 會是非常令人尷尬的。

我們在上次寫的類別裡再增加兩個測試，來更全面的測試這個方法:

polls/tests.py

```
    def test_was_published_recently_with_old_question(self):
        """
        was_published_recently() 對於 questions 的 pub_date 是比 1 天還早(older)的
        會回傳 False。
        """
        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() 對於 questions 的 pub_date 是在 1 天之內的
        會回傳 True。
        """
        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)
```

現在，我們有三個測試來確保 **`Question.was_published_recently()`** 方法對於過去，最近，和將來的三種情況都回傳正確的值。

再次申明，儘管 **`polls`** 現在是個小型的應用，但是無論它以後變得到多麼複雜，無論他和其他程式如何交互，我們可以在一定程度上保證我們為之編寫測試的方法將按照預期的方式執行。

測試視圖

我們的投票應用是相當地沒有差別待遇的：它將會發佈任何的問題，包括那些 **`pub_date`** 欄位值是在未來的問題。我們應該改善這一點。當在設定 **`pub_date`** 為未來的某一天時，這應該被解釋為這個問題 Question 要等到所指定的時間點才發佈，而在此之前在視圖上是看不到的。

針對視圖的測試

修復上述 bug 錯誤後，我們首先撰寫了測試，然後再編寫了程式進行修復。事實上，這是一個「測試驅動」開發模式的實例，但我們按什麼順序進行工作並不重要。

在我們的第一個測試中，我們密切關注程式的內部行為。對於此次測試，我們希望檢查其行為，就像用戶使用瀏覽器開啟我們的應用所經歷的那樣。

在嘗試修復任何問題之前，讓我們看一下可供使用的工具。

Django 測試工具之 Client

Django 提供了一個 **`Client`** 測試能模擬用戶和程式在視圖層的互動。我們可以在 **`tests.py`** 甚至是 **`shell`** 中使用它。

我們將再次從 **`shell`** 開始，我們需要做一些在 **`tests.py`** 中不需要做的事情。第一步是在 **`shell`** 中設定測試環境:

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

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

`setup_test_environment()` 安裝了一個範本實現器，使我們可以檢查 response 的一些額外的屬性，例如 **`response.context`**，否則這些屬性將無法被使用。注意，這個方法並 *不會* 設定測試資料庫，所以接下來的程式將會在當前存在的資料庫上執行，輸出的內容可能由於資料

庫內容的不同而不同。如果你的 **settings.py** 中關於 **TIME_ZONE** 的設定不對，你可能無法取得到期望的結果。如果你之前忘了設定，在繼續之前檢查一下。

接下來，我們需要匯入測試 client 類別（稍後在 **tests.py** 中我們將會使用 **django.test.TestCase** 類別，該類別裡包含了自己的 client 實例，因此不需要這一步驟）：

```
>>> from django.test import Client
>>> # 建立用戶端實例以供我們使用
>>> client = Client()
```

準備好之後，我們可以請 client 為我們做一些工作:

```
>>> # 從 '/' 取得一個回應
>>> response = client.get('/')
Not Found: /
>>> # 我們應該從該位址獲得一個 404；如果您看到一個
>>> # "Invalid HTTP_HOST header" 錯誤和一個 400 的回應，您可能
>>> # 忽略了前面描述的 setup_test_environment() 呼叫。
>>> response.status_code
404
>>> # 另一方面，我們預期應該在 '/polls/' 找到些什麼
>>> # 我們將會使用 'reverse()' 而不是一個直接以程式碼撰寫的 URL
>>> 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&#x27;s up?</a></li>\n    \n    </ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>
```

改善視圖程式

現在的投票清單表會顯示尚未發佈的投票（即**pub_date** 的值是未來的某天)。我們來解決這個問題。

在 教學的第 4 部分 裡，我們介紹了基於 **ListView** 的視圖類別：

polls/views.py

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

    def get_queryset(self):
        """回傳最近發佈的五個問題。"""
        return Question.objects.order_by('-pub_date')[:5]
```

我們需要修改 **get_queryset()** 方法並對其進行變更，讓它同時可以透過比較**timezone.now()** 來檢查日期。首先我們需要新增一行 import 語句：

polls/views.py

```
from django.utils import timezone
```

然後我們把 **get_queryset** 方法改寫成下面這樣：

polls/views.py

```
def get_queryset(self):
    """
    回傳最後五個已發佈的問題（不包括計劃在
    未來發佈的問題）。
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) 傳回了那些在所有的 **Question** 中的 **pub_date** 的值比'現在時間'還小於或等於的一個查詢集(queryset) — 意即早於或等於 **timezone.now**。

測試新視圖

啟動伺服器、在瀏覽器中載入網站、建立一些發佈時間在過去和將來的 **Questions**，然後檢驗只有已經發佈的 **Questions** 會顯示出來，現在你可以對自己感到滿意了。*你不想每次修改可能與這相關的程式時都重復這樣做* — 所以讓我們基於以上 **shell** 會話中的內容，再編寫一個測試。

將下面的程式增加到 **polls/tests.py**：

polls/tests.py

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

然後我們寫一個公用的快捷函數用於建立投票問題，再為視圖建立一個測試類別：

polls/tests.py

```
def create_question(question_text, days):
    """
    使用指定的 `question_text` 建立一個問題，並發佈
    從現在起所指定偏移的 `天數` （對於過去發佈的問題為負數
    ，對於尚未發佈的問題為正數）。
    """
    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):
        """
        如果沒有問題，則會顯示相關的訊息。
        """
        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):
        """
        過去帶有 pub_date 的問題將顯示在索引
        的頁面上。
        """
        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):
        """
        帶有在未來 pub_date 的問題沒有顯示在
        索引的頁面上。
        """
        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):
        """
        即使存在過去和將來的問題，也只有過去的問題
        會顯示在頁面上。
        """
        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):
        """
        問題索引頁面可能會顯示多個問題。
        """
        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.>']
        )
```

讓我們更詳細地看下以上這些內容。

首先是一個快捷函數 **`create_question`**，它封裝了建立投票的流程，減少了重復程式。

`test_no_questions` 方法裡沒有建立任何投票，它檢查回傳的網頁上有沒有 "No polls are available." 這段消息和 **`latest_question_list`** 是否為空。注意到 **`django.test.TestCase`** 類別提供了一些額外的 assertion 方法，在這個例子中，我們使用了 **`assertContains()`** 和 **`assertQuerysetEqual()`**。

在 **`test_past_question`** 方法中，我們建立了一個投票並檢查它是否出現在欄表中。

在 **`test_future_question`** 中，我們建立 **`pub_date`** 在未來某天的投票。資料庫會在每次呼叫測試方法前被重置，所以第一個投票已經沒了，所以主頁中應該沒有任何投票。

剩下的那些也都差不多。實際上，測試就是假裝一些管理員的輸入，然後透過用戶端的表現是否符合預期來判斷新加入的改變是否破壞了原有的系統狀態。

測試 `DetailView`

我們的工作似乎已經很完美了？不，還有一個問題：就算在發佈日期時未來的那些投票不會在目錄頁 *index* 裡出現，但是如果用戶知道或者猜到正確的 URL ，還是可以開啟到它們。所以我們得在 **`DetailView`** 裡增加一些約束：

polls/views.py

```
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        排除所有尚未發佈的問題。
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

當然，我們將增加一些測試來檢驗 **pub_date** 在過去的 **Question** 可以顯示出來，而 **pub_date** 在未來的不可以：

polls/tests.py

```
class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        帶有未來 pub_date 的問題的詳細視圖
        會回傳一個 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):
        """
        帶有過去 pub_date 的問題的詳細視圖
        顯示該問題的本文。
        """
        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)
```

更多的測試思路

我們應該給 **ResultsView** 也增加一個類似的 **get_queryset** 方法，並且為它建立測試。這和我們之前幹的差不多，事實上，基本就是重復一遍。

我們還可以從各個方面改進投票應用，但是測試會一直伴隨我們。比方說，在目錄頁上顯示一個沒有選項 **Choices** 的投票問題就沒什麼意義。我們可以檢查並排除這樣的投票題。測試可以建立一個沒有選項的投票，然後檢查它是否被顯示在目錄上。當然也要建立一個有選項的投票，然後確認它確實被顯示了。

恩，也許你想讓管理員能在目錄上看見未被發佈的那些投票，但是普通用戶看不到。不管怎麼說，如果你想要增加一個新功能，那麼同時一定要為它編寫測試。不過你是先寫程式還是先寫測試那就隨你了。

在未來的某個時刻，你一定會去查看測試程式，然後開始懷疑：「這麼多的測試不會使程式越來越複雜嗎？」。別著急，我們馬上就會談到這一點。

當需要測試的時候，測試用例越多越好

貌似我們的測試多的快要失去控制了。按照這樣發展下去，測試程式就要變得比應用的實際程式還要多了。而且測試程式大多都是重復且不優雅的，特別是在和業務程式比起來的時候，這種感覺更加明顯。

但是這沒關聯！ 就讓測試程式繼續肆意增長吧。大部分情況下，你寫完一個測試之後就可以忘掉它了。在你繼續開發的過程中，它會一直默默無聞地為你做貢獻的。

但有時測試也需要更新。想象一下如果我們修改了視圖，只顯示有選項的那些投票，那麼只前寫的很多測試就都會失敗。*但這也明確地告訴了我們哪些測試需要被更新*，所以測試也會測試自己。

最壞的情況是，當你繼續開發的時候，發現之前的一些測試現在看來是多餘的。但是這也不是什麼問題，多做些測試也 *不錯*。

如果你對測試有個整體規劃，那麼它們就幾乎不會變得混亂。下面有幾條好的建議：

- 對於每個模型和視圖都建立單獨的 **`TestClass`**
- 每個測試方法只測試一個功能
- 給每個測試方法起個能描述其功能的名字

深入程式測試

在本教學中，我們僅僅是了解了測試的基礎知識。你能做的還有很多，而且世界上有很多有用的工具來幫你完成這些有意義的事。

舉個例子，在上述的測試中，我們已經從程式邏輯和視圖回應的角度檢查了應用的輸出，現在你可以從一個更加 "in-browser" 的角度來檢查最終實現出的 HTML 是否符合預期，使用 Selenium 可以很輕松的完成這件事。這個工具不僅可以測試 Django 框架裡的程式，還可以檢查其他部分，例如說你的 JavaScript。它假裝成是一個正在和你網站進行交互的瀏覽器，就好像有個真人在開啟網站一樣！Django 它提供了 **`LiveServerTestCase`** 來和 Selenium 這樣的工具進行交互。

如果你在開發一個很複雜的應用的話，你也許想在每次提交程式時自動執行測試，也就是我們所說的持續集成 continuous integration，這樣就能實現質量控制的自動化，起碼是部分自動化。

一個找出程式中未被測試部分的方法是檢查程式覆蓋率。它有助於找出程式中的薄弱部分和無用部分。如果你無法測試一段程式，通常說明這段程式需要被重構或者刪除。想知道程式覆蓋率和無用程式的詳細資訊，查看文件 Integration with coverage.py 取得詳細資訊。

文件 Django 中的測試 裡有關於測試的更多資訊。

接下來要做什麼?

如果你想深入了解測試，就去看 Django 中的測試 。

當你已經比較熟悉測試 Django 視圖的方法後，就可以繼續閱讀 教學第 6 部分，學習靜態文件管理的相關知識。

‹ 編寫你的第一個 Django 應用，第 4 部分

編寫你的第一個 Django 應用，第 6 部分 ›