Django unittest测试框架特性简述

我们知道 Django 单元测试的实现是基于 Python unittest 测试框架，unittest 作为功能完善的单元测试框架，它的相关特性也可以应用在 Django 项目中，在本节我们将介绍 unittest 框架的相关特性是如何在 Django 中进行应用的。

1. unittest框架核心概念

unittest 单元测试框架受到了 JUnit（JUnit 是一个 Java 语言的单元测试框架）的启发，它与其他语言的测试框架有着相似的风格。unittest 包含了4 个核心概念，给大家总结如下：

- test fixture：它代表的是初始化和清理测试环境，它最常见的使用场景，比如数据库连接的创建与销毁。
- test case：它代表的是 unittest.TestCase 类实例，一个完整的测试单元，通过运行这个测试单元实现最终的测试验证。
- test suite：它代表的是 test case 的集合，同时 test suite 之间可以进行嵌套，从而达到多个测试任务一起执行的目的。
- test runner：它代表的是运行测试用例，然后给用户最终的测试结果。

1) 初始化和清理测试环境

初始化和清理测试环境是使用 unittest 框架进行测试第一步也是最重要的一步，这个过程中涉及到了两个方法即 setUp 和 tearDown，它们分别负责初始化工作和环境的清理工作，它们在每次执行测试用例的前后执行，即 setUp 在执行测试用例前执行处理化工作，tearDown 在测试用例执行完成后做一些清理的工作。

我们可以把需要初始化的工作放在 setUp 中执行，比如《Django项目编写单元测试用例》中 ExampleTest 类的 test_modle 与 test_view 方法创建一个 pub1 实例，我们可以把它当成一种初始化工作，我们对 ExampleTest 类稍微进行改动，代码如下所示：

```
1. from django.test import TestCase
2. from index.models import Book, PubName
3. class ExampleTest(TestCase):
4.     def setUp(self):
5.         print('我负责测试环境初始化')
6.         self.pub1=PubName.objects.create(pubname="编程帮出版")
7.
8.     def test_model(self):
```

```
        print('执行test_model测试')
        book=Book.objects.create(title='Servlet',price='35.00', retail_price='35.00',pub=self.pub1)
        self.assertTrue(book is not None)
        self.assertNotEqual(Book.objects.count(),8)
        self.assertEqual(Book.objects.count(),1)

    def test_view(self):
        print('执行test_view测试')
        book=Book.objects.create(title='Jsp',price='25.00', retail_price='25.00',pub=self.pub1)
        response=self.client.get('/index/update_book/%d/' % book.id)#视图访问获取response
        self.assertEqual(response.status_code,200)
    def tearDown(self):
        print('我负责清理测试环境')
```

然后我们执行测试用例，最后在 CMD 命令行工具中得到如下结果：

```
C:¥Users¥Administrator¥Book¥BookStore>python manage.py test index.tests.ExampleTest
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
我负责测试环境初始化
执行test_model测试
增加了新的书籍
我负责清理测试环境
.我负责测试环境初始化
执行test_view测试
增加了新的书籍
我负责清理测试环境
.
----------------------------------------------------------------------
Ran 2 tests in 0.035s
OK
Destroying test database for alias 'default'...
```

可以看出在每个测试用例执行的前后都执行了 setUp 和 tearDown 方法。但是还有一个问题值得我们思考，如果某些测试用例需要跳过执行，或者直接跳过整个测试类，我们又应该如何做呢，下面我们继续探究。

2.装饰器实现跳过测试与预期失败

unittest 框架还有一个重要特性就是可以跳过某些测试用例或者已经预期失败的用例。跳过测试用例很好理解，就是不执行某些测试方法，甚至可以直接跳过测试类，那么预期失败怎么解释呢，其实也很好理解，就是已经预测到因为某些原因导致的测试不通过，但是在执行测试的时候不希望该条测试用例仍然标记为失败，那么在这个时候，我们就可以想办法跳过。

1) 跳过测试装饰器

跳过测试的功能可以使用装饰器实现，这类装饰器有以下三个：

- unittest.skip(reason)：无条件跳过，其中 reason 用来表示跳过测试的原因；
- unittest.skipIf(condition,reason)：当条件（condition）成立的时候，跳过测试；
- unittestUnless(condition,reason)：与 skipIf 相反，当条件（condition）不成立的时候，跳过测试。

它的应用如下所示：

```
class SkipTest(unittest.TestCase):
    @unittest.skip('不用测试A用例')
    def test_A(self):
        print('测试A')
    @unittest.skipIf(True,'跳过B')
    def test_B(self):
        print('测试B')
    @unittest.skipUnless(False,'跳过C')
    def test_C(self):
        print('测试C')
```

执行测试命令发，得到如下输出：

```
C:¥Users¥Administrator¥Book¥BookStore>python manage.py test -v 3 index.test.SkipTest
Skipping setup of unused database(s): default.
System check identified no issues (0 silenced).
test_A (index.test.SkipTest) ... skipped '不用测试A用例'
test_B (index.test.SkipTest) ... skipped '跳过B'
test_C (index.test.SkipTest) ... skipped '跳过C'
----------------------------------------------------------------------
Ran 3 tests in 0.001s
OK (skipped=3)
```

如果想直接跳过测试类，可以直接在类名上方使用装饰器即可，这里就不再赘述。

2) 跳过预期失败装饰器

使用unittest.expectedFailure处理预期失败的用例，使用方法和上面的跳过装饰器一样，不过这里有一点需要注意，不管是标注了该装饰器的方法可以通过测试，还是标注了该装饰器的类中有通过测试的方法，它们都会被认为是测试失败即 FAILED，它提供了两个参数，如下所示：

FAILED (expected failures=1, unexpected successes=1)

含义可想而知， expected failures=1 表示使用改装饰器的方法确实测试不通过；unexpected successes=1 表示该方法中某些断言可以测试通过，但并不代表所有断言都通过测试。当然 unittest 不止这些特性，如果小伙伴们感兴趣可以继续参阅官方文档《unittest单元测试框架》。