单元测试和项目上线

【Django】Django项目结构与单元测试

学校的软工项目要开发一个网站，自然的想到用python+Django来做。由于之前没有用Django开发过大型的网站项目，所以遇到了一些问题。记录在此，便于以后查阅。

今天完成了项目结构的设计、部分的单元测试以及把代码使用策略模式重构。

项目结构

使用app完成功能

首先要明确网站基本的功能实现是要用各种各样的app来实现的，我觉得这样的优点有几点：

1. 功能划分明确，之后修改方便。
2. 因为有多个models，所以可以针对不同的功能设计数据库，也是为了功能逻辑之间的划分。
3. 多个test可以更有针对性的进行测试

具体的python代码：`django-admin startapp`。

app的具体位置

因为之前看过一本16堂课学会Django架站的书，上面的提到将app放到项目文件夹下，我觉得这样不够好，因为会导致app文件夹可能与templates或static文件夹搞混的情况，同样也不是很好看；所以我这次使用了一个apps文件夹存放所有的app。

QuerySet的具体使用

这里记录一下objects的使用，链接

Django单元测试

啊这个也踩了一些坑，不过挺浅的。。。Django的单元测试是基于`django.test.TestCase`这个类（继承unittest类）实现的。

具体步骤

首先编写测试类，测试类是继承TestCase类的，之后重载setUp方法，做一些测试的准备操作，我测试的是models中的数据表以及操作方法，因此import model中的各种类；然后就是编写测试方法了，我是一个一个个方法测试的，每个方法写几个test case，保证所有的分支都被覆盖。

写好了代码我发现了一个重要的问题，我不知道咋运行QAQ，求助度娘发现要用这个命令：

```
python manage.py test <module_name>

e.g.
python manage.py test apps.signin.tests
```

Django会自动地执行相应模块的所有tests。

那么做完unittest自然就想到要得到代码覆盖率，这里使用一个coverage的工具，要自己安装。

代码：`(cmd)pip install coverage`

coverage

coverage是一个检查单元测试覆盖率的工具，django的文档中也有简要的说明coverage的集成 文档地址

```
#测试并收集测试信息
coverage run --source='.' manage.py test --setting mandela.settings_test
#查看测试结果
coverage report -m
Name                                                                                 Stmts   Miss  Cover   Missing
------------------------------------------------------------------------------------------------------------
acquireportal/__init__.py                                                                0      0   100%
acquireportal/controler.py                                                              65     47    28%   22-56, 60-71, 76-79
acquireportal/migrations/0001_initial.py                                                 6      0   100%
acquireportal/migrations/0002_auto_20160622_1059.py                                      6      0   100%
acquireportal/migrations/0003_auto_20160622_1100.py                                      5      0   100%
....

------------------------------------------------------------------------------------------------------------
TOTAL                                                                                 8013   5858    27%
```

图片来自这篇博客

这样就得到了代码覆盖率。

Django单元测试

为什么需要自动化测试

- 对于大项目，对每个单元进行测试可以快速定位错误，确保项目质 量
- 在 python 中，利用 python 的测试化模块搭建一些测试不会比手 动测试一些数据更麻烦
  - 多写一些代码
  - （几乎）不和终端的输入输出打交道
  - 少和单步调试打交道
- 在 python 交互式终端的输入输出，可以马上成为一些测试样例
- 带有和 Java 类似的单元测试框架，可以提取出不同测试用例的相 同步骤
- 回归测试：一个测试用例可以在开发过程中反复测试，保证加入新 功能后不会妨碍原有功能的运转

测试覆盖率

- 定义：在运行测试时运行的代码占总代码量的比例
- 作用：找出没有被测试过的函数和代码行
- 自带 trace.py：python -m trace --help
  - --count 统计每行**被执行的次数**
  - --trace 程序每执行一行，就将这一行打印到标准输出
  - 统计被执行的函数名、调用关系、累积多次运行的总次数、**标出没有被运行的代码行**……

Doctest

- python 的 docstring 中有很多在交互式终端中运行的例子

- doctest 可以检查 docstring 中样例的正确性
- 同样的，可以把手动测试的终端输出放进 docstring 里，变成一个 测试用例
- 注意：一般小型的或者简单的功能可以使用，但是较大的项目或者较复杂的功能并不推荐使用 doctest

> 下面是一个样例

```
def factorial(n):
"""Return the factorial of n, an exact integer >= 0.

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30) 265252859812191058636308480000000
>>> factorial(-1)
Traceback (most recent call last):
    ...
 ValueError: n must be >= 0
"""

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Unittest

- 基于 JUnit 测试框架 (JAVA)
- 比 doctest 更加灵活, 功能更加强大
- 我们要提到的django单元测试就是基于unittest的，下面先简要介绍下如何使用unittest

如何使用Unittest

- 先构造 unittest.TestCase 的子类
- unittest.main() 用法
    - unittest.main() 函数将会实例化所有当前 py 文件下的 TestCase 的 子类并且执行以'test' 开头的函数
    - 若需要跨文件测试，则只需 import 含 testBase 子类的 py 文件再调 用 unittest.main() 即可
- startUp 与 tearDown 函数
    - startUp 将会在每个 test 函数执行前被调用
    - tearDown 将会在每个 test 函数执行后被调用
    - 通常用来初始化以及清除测试过程中的一些代码
- 常用的TestCase函数：
    - assert_(expr[, msg]) 与 failUnless(expr[, msg])

        - 会对不正确的结果报错
    - assertEqual(x, y[, msg]) 与 failUnlessEqual(x, y[, msg])
        - 值不等时报错并输出二者的值
    - assertAlmostEqual(x, y[, places[, msg]]) 与 failUnlessAlmostEqual(x, y[, places[, msg]])
        - 值不近似等时报错，用于浮点数的判等
    - assertRaises(exc, callable, ...) 与 failUnlessRaises(exc, callable, ...)
        - 该回调函数没有抛出 exc 的异常则报错

> 下面是一个样例

```
class ProductTestCase(unittest.TestCase):
    def testIntegers(self):
        x = 3
        y = 4
        p = my_math.product(x, y)
        self.failUnless(p == x*y, 'Integer multiplication failed')
if __name__ == '__main__': unittest.main()
************************Wrong************************
F
======================================================
FAIL: testIntegers (__main__.ProductTestCase)
------------------------------------------------------
Traceback (most recent call last):
                File "test_my_math.py", line 17, in testIntegers
self.failUnless(p == x*y, 'Integer multiplication failed')
AssertionError: Integer multiplication failed
------------------------------------------------------
Ran 1 tests in 0.001s
FAILED (failures=1)

************************Correct************************
.
------------------------------------------------------
Ran 1 tests in 0.001s
OK
```

- 当测试结果正确时，返回一个`.`而不正确时是返回一个`F`的

源码检查

- PyChecker
    - 检查 python 源码错误，例如传参错误，没有导入模块，同一作用域 中重定义函数、类方法等。
    - 使用：pychecker [options] file1.py file2.py ...
- PyLint
    - 支持大部分 PyChecker 功能
    - 更强大的功能，例如检查变量名是否符合规定，检查一行代码的长 度，一个声明过的接口是否被实现
    - 使用：pylint [options] module_or_package
- 结合 unittest 使用
    - 代码中嵌入

- PyChecker 与 PyLint 集成到 IDE

Django单元测试

> 讲了这么多python的单元测试，那如何进行Django的单元测试呢?

- 在Django的单元测试中，仍然推荐使用python的unittest模块，像我们刚刚在上面提到的使用方法，使用类名为django.test.TestCase，继承于python的unittest.TestCase。

```
class TestDefault(TestCase):

    def setUp(self):
        # 设置配置
        settings.IGNORE_WECHAT_SIGNATURE = True

        # user1 => not bind
        # user2 => bind
        User.objects.create(open_id='abc')
        User.objects.create(open_id='a', student_id='2016013265')

        # textMsgs => 用户一般可能输入(成功)
        self.textMsgs = ['balabala', 'gg', '抢火车票']

    # 是否返回帮助
    def is_default(self, content):
        pattern = '对不起，没有找到您需要的信息:('
        return content.find(pattern) != -1

    def test_text(self):
        users = User.objects.all()

        for user in users:
            for textMsg in self.textMsgs:
                fromUser = user.open_id
                curTime = str(getTimeStamp(datetime.datetime.now()))
                msgId = str(random.randint(0, 99999)) + curTime
                data = getTextXml(fromUser, curTime, textMsg, msgId)

                response = self.client.post(
                    path='/wechat/',
                    content_type='application/xml',
                    data=data
                )

                content = str(response.content.decode('utf-8'))
                self.assertEqual(self.is_default(content), True)
```

- 执行目录下所有的测试(所有的test*.py文件):

> python manage.py test

运行测试的时候，测试程序会在所有以test开头的文件中查找所有的test cases(inittest.TestCase的子类),自动建立测试集然后运行测试。（注意是所有app的test文件，包括子目录中的)

- 执行项目的所有的test测试:

> python manage.py test

运行结果：

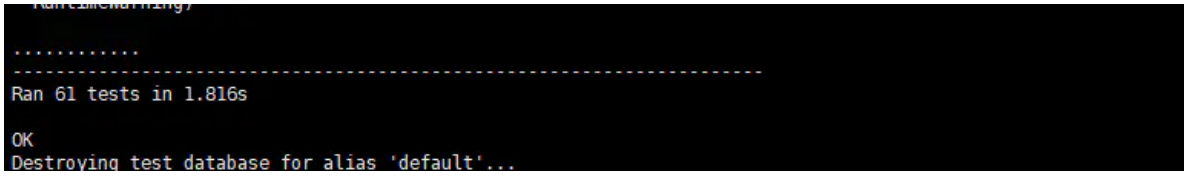

image.png

- 执行项目某个下tests包里的测试：

> python manage.py xxx.tests

- 单独执行某个test case：

> python manage.py xxx.TestDefault

- 单独执行某个测试方法：

> python manage.py xxx.TestDefault.testxxx/py

- 通配测试文件名：

> python manage.py test–pattern=”tests_*.py”

- 启用warnings提醒：

> python -Wall manage.py test

数据库

> 测试是需要数据库的，django会为测试单独生成数据库。不管你的测试是否通过,当你所有的测试都执行过后,这个测试数据库就会被销毁。

- 默认情况下,测试数据库的名字是test_DATABASE_NAME,DATABASE_NAME是你在settings.py里配置的数据库名；
  如果你需要给测试数据库一个其他的名字,在settings.py中指定TEST_DATABASE_NAME的值
- 使用sqlite3时，数据库是在内存中创建的。
  除了数据库是单独创建的以外,测试工具会使用相同的数据库配置--DATABASE_ENGINE, DATABASE_USER, DATABASE_HOST等等.
  创建测试数据库的用户DATABASE_USER(settings中)指定,所以你需要确认 DATABASE_USER有足够的权限去创建数据库。

> 为了保证所有的测试都从干净的数据库开始，执行顺序如下：

- 1.所有的TestCase子类首先运行。
- 2.所有其他的单元测试(unittest.TestCase,SimpleTestCase,TransactionTestCase)。
- 3.其它的测试(例如doctests等)

> 加速测试

- 可以将PASSWORD_HASHERS设置为更快的算法：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.MD5PasswordHasher',
)
```

travis CI与django的单元测试

- 在.travis.yml文件中加入（修改）：

```
script:
    - python manage.py test
```