第122天：Flask 单元测试

原创 太阳雪 Python技术 2月7日

如果一个软件项目没有经过测试，就像做的菜里没加盐一样。Flask 作为一个 Web 软件项目，如何做单元测试呢，今天我们来了解下，基于 unittest 的 Flask 项目的单元测试。

什么是单元测试

单元测试是软件测试的一种类型。顾名思义，单元测试的对象是程序中的最小的单元，可以是一个函数，一个类，也可以是它们的组合。

相对于模块测试、集成测试以及系统测试等高级别的测试，单元测试一般由软件开发者而不是独立的测试工程师完成，且具有自动化测试的特质，因此单元测试也属于自动化测试。

在实际开发中，有一些测试建议：

- 测试单元应该关注于尽可能小的功能，要能证明它是正确的
- 每个测试单元必须是完全独立的，必须能单独运行
- 修改代码后，需要重新执行一次测试代码，以确保本次修改不会影响到其他部分
- 提交代码前，需要执行一次完整测试，以确保不会将不完整或者错误的代码提交，影响其他开发者
- 测试代码要和正常代码有明显的区分，测试代码文件应该是独立的

unittest 模块

Python 有很多单元测试框架，unittest、nose、pytest 等等，unittest 是 Python 内置的测试库，也是很多测试框架的基础，地位如同 Java 中的 JUnit，所以有时也被称作 PyUnit。

unittest 支持 `自动化测试`、可以在多个测试中 `共享设置测试环境和撤销测试环境代码`、`可以将分散的测试集中起来`，并且可以支持 `多种测试报告框架`，因此 unittest 有四种重要概念：

- test fixture 测试前后需要做些准备和清理工作，例如临时数据库连接、测试数据创建、测试用服务器创建，以及测试后的清理和销毁，test fixture 提供了 `setUp` 和 `tearDown` 接口来完成这些事情，并且可以被多个测试方法所共享
- test case 测试用例，是最小的测试单元，检测一个特定输入的响应结果，unittest 提供 `TestCase` 基类，以便开发者创建具体的测试用例类
- test suite 暂且翻译成测试套餐吧，是多个测试用例、测试套餐的组合，为了将一组相关的测试组织起来的工具
- test run 测试执行器是按照一定规则执行测试用例，记录并返回测试结果的组件

小试牛刀

unittest 不需要安装，直接导入，例如一个测试字符串方法的测试代码：

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

- 导入 unittest 模块
- 创建一个测试字符串方法的测试类，继承之 unittest 的 `TestCase`
- 编写测试方法，注意测试方法必须以 **test** 作为开头，这样才能被测试加载器识别，同时也是良好的编程习惯
- TestCase 提供了很多检验方法，例如 `assertEqual`、`assertTrue` 等等，用于对期望结果进行检测
- 最后，如果最为主代码被运行，调用 `unittest.main` 执行所有测试方法

运行代码:

```
python testBase.py
```

或者

```
python -m unittest testBase.py
```

结果如下：

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

```

```
OK
```

可以看到，执行了三个测试，没有发现异常情况，`.` 表示测试通过，数量表示执行了的测试方法个数

测试执行器

unittest.main 只给出了概要测试结果，如果需要更详细的报告，可以用 `测试执行器` 来运行测试代码

将 unittest.main() 换成:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

- 利用测试加载器（TestLoader）创建了一个测试套餐（TestSuite）
- 用测试执行器（TestRunner）执行测试代码
- TestTestRunner 是将结果作为文本格式输出
- 参数 verbosity=2 表示显示详细的测试报告

或者干脆为 unittest.main 提供参数 verbosity： `unittest.main(verbosity=2)`

运行结果如下：

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Flask 单元测试

Flask 作为一个 Web 项目，大多数代码需要在 Web 服务器环境下运行

- 所以需要为每个单元测试模拟一个 Web 环境
- 另外有些部分需要使用到数据库，所以还需要为这些测试准备一个数据库环境
- 最后有些业务处理代码，比如加工数据，数据运算等，可以进行独立测试，不需要 Web 环境

创建了一个简单项目，通过工厂方法创建 Flask 应用，有数据库的读写，下面逐步说明下测试脚本，测试代码文件 `testApp.py` 与项目代码在同一目录下

初始化环境

```
import unittest

from app import create_app
from model import db

class TestAPP(unittest.TestCase):
    def setUp(self):
        self.app = create_app(config_name='testing')
        self.client = self.app.test_client()
        with self.app.app_context():
            db.create_all()

    def tearDown(self):
        with self.app.app_context():
            db.drop_all()
```

- 引入 unittest 模块
- 从 Flask 应用代码文件（`app.py`）中引入工厂方法 `create_app`
- 从模型代码文件（`model.py`）中引入数据库实例 `db`
- 创建测试类 `TestAPP`，继承自 `unittest.TestCase`
- 定义 `setUp` 方法，用工厂方法初始化 Flask 应用
- Flask 提供了 `测试应用` 的创建方法 `test_client`，返回测试应用实例
- 在应用实体环境下，初始化数据库
- 定义 `tearDown` 方法，在测试结束后销毁数据库中的结构和数据

简单测试

编写两个测试方法，分别对 Flask 应用的配置情况和首页进行测试：

```
def test_config(self):
        self.assertEqual(self.app.config['TESTING'], True)
        self.assertIsNotNone(self.app.config['SQLALCHEMY_DATABASE_URI'])

def test_index(self):
    ret = self.client.get('/')
    self.assertEqual(b'Hello world!', ret.data)
```

- 定义测试方法 test_config 用来测试 Flask app 的配置是否正常
- 因为测试方法时实体方法，所以从实体引用（self）中的 app 属性中，查看配置属性，注意测试应用 test_client 不能之间获取 Flask app 的配置
- 检测 `TESTING` 的值是否为 True，另外检查数据库连接是否存在

- 定义方法首页的方法 test_index，通过测试应用的 `get` 方法访问网站根目录
- 检测访问后的结果，在示例中，首页返回了字符串，确认下是否正确

此时运行测试代码可以得到如下

```
test_config (__main__.TestAPP) ... ok
test_index (__main__.TestAPP) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.066s
```

测试表单提交

在 Web 项目中，有很多需要交互的功能，例如表单提交，数据存储和查询，在 unittest 测试框架中，借助 Flask 的测试应用 test_client 可以轻松应对

示例项目中，有模拟用户注册和登录的功能，注册和登录都需要提交数据，并且只有在注册后，才能进行登录，所以将注册和登录编写成单独的功能：

```
def login(self, username):
    params = {'username': username}
    return self.client.post('/login', data=params, follow_redirects=True)

def register(self, username):
    params = {'username': username}
    return self.client.post('/register', data=params, follow_redirects=True)
```

- 定义登录方法 `login`，接受一个用户名的参数（这里忽略了密码等登录凭证）
- 利用测试应用 test_client 的 post 方法，访问登录地址，将提交的数据用 `词典` 数据结构通过 data 参数提交
- 定义注册方法 `register`，接受一个用户名的参数（同样忽略了密码等其他信息）
- 注册方法和登录类似，除了注册提交地址
- 注意到 post 的参数 `follow_redirects`，值为 `True` 的作用是支持浏览器跳转，即收到跳转状态码时会自动跳转，直到不是跳转状态码时才会返回
- 登录和注册方法可以处理更多的业务逻辑，最后将请求结果返回

有了注册和登录的协助，测试方法就更明晰：

```
def test_register(self):
    ret = self.register('bar')
    self.assertEqual(json.loads(ret.data)['success'], True)

def test_login(self):
    self.register('foo')
    ret = self.login('foo')
```

```
        return self.assertEqual(json.loads(ret.data)['username'], 'foo')

    def test_noRegisterLogin(self):
        ret = self.login('foo')
        return self.assertEqual(json.loads(ret.data)['success'], False)

    def test_login_get(self):
        ret = self.client.get('/login', follow_redirects=True)
        self.assertIn(b'Method Not Allowed', ret.data)
```

- 定义了 4 个测试方法，分别时单独的注册，注册后登录，未注册时的登录，和用 get 方法请求登录接口
- 每种方法都调用了 `login` 或者 `register` 方法，所以代码逻辑会更简洁
- 注册和登录接口，返回的时 JSON 格式数据，需要用 json.loads 将其转化为 `词典`
- `assertIn` 类似与 `indexOf` 方法，用来检测给定的字符串是否在结果中

运行上述的是测试，可以得到如下结果:

```
test_login (__main__.TestAPP) ... ok
test_login_get (__main__.TestAPP) ... ok
test_noRegisterLogin (__main__.TestAPP) ... ok
test_register (__main__.TestAPP) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.196s

OK
```

您可能已经发现，测试执行的结果和测试方法定义的顺序不一致

原因是测试加载器是按照测试名称字母顺序加载测试方法的，如果需要按照一定的顺序执行，需要用 `TestSuite` 设定执行顺序，如：

```
if __name__ == '__main__':
    suite = unittest.TestSuite()
    tests = [TestAPP('test_register'), TestAPP('test_login'), TestAPP('test_noRegisterLogin'), TestAP
    suite.addTests(tests)
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
```

- 创建 `TestSuite` 实例
- 将需要组织的测试方法放在数组中，用 `TestSuite` 的 `addTests` 方法添加到 `TestSuite` 实例中
- 用 `TestRuuner` 运行 `TestSuite` 实例

这样就会以设定的顺序执行测试方法了

代码覆盖率

测试中有个重要的概念就是代码覆盖率，如果存在没有被被覆盖的代码，就有可能编写的测试代码不够全面

`coverage` Python 的一个测试工具，不仅可以运行测试代码，还可以报告出代码覆盖率

安装

使用前，需要安装：

```
pip install coverage
```

执行测试

安装成功后，就可以在命令行中使用了，首先进入到测试代码的所在目录，

> 请注意 Python 包引用的查找位置，从不同的目录运行，可能会影响到目录下模块的引用，例如在同一目录下，引用模块，如果在上一级目录中运行代码，可能出现找不到模块的错误，此时只需要相对于运行目录，调整下代码中模块引用方式就好了，具体可参见Python Unit Testing – Structuring Your Project

执行如下命令:

```
coverage run testApp.py
```

结果如下：

```
test_config (__main__.TestAPP) ... ok
test_index (__main__.TestAPP) ... ok
test_login (__main__.TestAPP) ... ok
test_login_get (__main__.TestAPP) ... ok
test_noRegisterLogin (__main__.TestAPP) ... ok
test_register (__main__.TestAPP) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.226s
```

结果和之间运行测试代码类似，也就是说用 `coverage run` 命令可以代替 python 命令执行测试代码，例如

```
python -m unittest discover
```

```

```

将变为

```
coverage run -m unittest discover
```

覆盖率

coverage 更大的用处在于查看代码覆盖率，命令是 `coverage report`，例如:

```
coverage report testApp.py
```

结果如下：

```
Name         Stmts   Miss  Cover
--------------------------------
testApp.py      41      0   100%
```

- Name 指的是代码文件名
- Stmts 是执行的代码行数
- Miss 表示没有被执行的行数
- Cover 表示覆盖率，公式是(Stmts-Miss)/Stmts，即被执行代码所占比例，用百分比表示

如果要看到哪些行被忽略了，加上参数 `-m` 即可:

```
coverage report -m testApp.py
```

结果中会多一列 Missing，内容为执行的行号

> 代码覆盖率报告，是基于 `coverage run` 的运行结果的，所以没有测试的运行就无法得到覆盖率报告的

整体覆盖率报告

`coverage run` 在执行测试时，会记录所有被调用代码文件的执行情况，包括 Python 库中的代码，如果只想记录指定目录下的代码执行情况，需要用 `--source` 选项指定需要记录的目录，例如只记录当前目录下的执行情况：

```
coverage run --source . testApp.py
```

然后查看执行报告，例如：

```
Name         Stmts   Miss  Cover
--------------------------------
app.py          10      0   100%
config.py       17      1    94%
model.py        17      4    76%
route.py        19      1    95%
testApp.py      41      0   100%
--------------------------------
TOTAL          104      6    94%
```

如果执行时没有加上 --source 参数，也可以通过通配符文件名，指定要查看的代码文件：

```
coverage report *.py
```

结果同上

html 测试报告

如果项目中代码文件众多，在命令行中用文本方式显示测试报告就不太方便了，`coverage html` 可以将测试报告生成 html 文件，功能强大，显示效果更好：

```
coverage html -d testreport
```

参数 -d 用来指定测试报告存放的目录，如果不存在会创建

文件名是个连接，点击可以看到文件内容，并且将执行和未执行的代码标注的很清楚：

总结

今天介绍了 Flask 的单元测试，主要介绍了 Python 自带单元测试模块 unittest 的基本用法，以及 Flask 项目中单元测试的特点和方法，还介绍了 coverage 测试工具，以及代码覆盖率报告的用法。

最后需要强调的是：无论什么软件项目，单元测试是很有必要的，单元测试不仅可以确保项目的高质量交付，而且还为维护和查找问题节省了时间。

参考

https://medium.com/@neeti.jain/how-to-do-unit-testing-in-flask-and-find-code-coverage-fa5201399bc4

https://www.patricksoftwareblog.com/python-unit-testing-structuring-your-project/

https://coverage.readthedocs.io/en/coverage-5.0.3/index.html

https://testerhome.com/topics/11655

> 示例代码：https://github.com/JustDoPython/python-100-day/tree/master/day-122

系列文章

第121天：机器学习之决策树
从 0 学习 Python 0 - 120 大合集总结

PS： 公号内回复：Python，即可进入Python 新手学习交流群，一起**100天计划！**

-END-

Python 技术
关于 Python 都在这里