18 测试驱动开发（TDD）网络爬虫项目

更新时间：2019-06-17 10:56:38

> 学习这件事不在乎有没有人教你，最重要的是在于你自己有没有觉悟和恒心。
>
> —— 法布尔

爬虫属于一种非可视化的服务，如果每次验证代码的逻辑是否正确都得运行一次爬虫那这种方法是极为之低效，更糟糕的情况是像遇到豆瓣此类带有防爬机制的网站，可能在调试爬虫的时候IP就被封了！为了加快开发速度，提高爬虫代码的正确性，本节将详细地介绍如何用 TDD 的方法来设计爬虫，并为爬虫编写单元测试，在加速开发效率的同时保证爬虫的质量。

在上一节中，之所以我建议你不要急于运行已经写好的爬虫，是因为之前的内容之中有不少的部分我们是并不能完全确定他们能正确运行的，例如正则表达式、又例如各个输入处理器和蜘蛛的页面解释逻辑，在这么多个部分都没有把握完全正确的情况下运行，程序大数是会出一些状况的。

TDD 简介

> TDD是**测试驱动开发**（Test-Driven Development）的英文简称，是敏捷开发中的一项核心实践和技术，也是一种设计方法论。TDD的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码。TDD虽是敏捷方法的核心实践，但不只适用于XP（Extreme Programming），同样可以适用于其他开发方法和过程。—— 引用自百度

我接触过的大多数开发人员认为“测试”是*额外而不必要*的工作，只会*增加工作时间* 并 *拖延* 项目推进的日期；某些*优秀*的程序员则认为：“只要*思路清晰，技术过硬* 编写的代码出现Bug的机率是很少的”；某些具有“大型重构”经历的程序员则认为：“测试是程序中的一种负累，在结构性重构工作中大量的测试代码会因为程序结构的变化而丧失其存在的意义。”

我认为以上的这些观点都是错误的，测试不单单只是为了对软件质量作出保证的一种形式化的手段。恰恰相反，测试是一个被开发人员最容易忽视的**强大工具**，它还是项目开发的加速器。

为什么要测试？

开发人员不喜欢测试可能最根本原因是于**根本不知道该测些什么**，网上不少的文章要不就泛泛而谈测些什么加减乘除一类的小学生示例，要不就讳莫如深根本不知其用意。

什么是期待值？

据此，我们就以实例入手先搞懂到底就要测试什么？上一节我们写的自定义输入处理器和蜘蛛就是最大的入手点。

就以输入处理器 `Number` 为例，说实在话我自己作为代码的编写者我都不能确认以下的几点：

- 运行的时候会不会出错？
- 运行时 `Number` 能不能从 `"评分9.6"` 这样的子符串中将数值提取出来？

当然你会说我直接跑一次程序不就清楚了吗？是的，所以很多程序员都是写完代码后就以**主程序**为引导经过层层操作后才到达自己要测试的地方，虽然每次可能就花上10来秒，但如果你一天要跑主程序上百次之后那是多少秒？10000s 约为2个多小时！我们开发人员的生命就是浪费在等待主程序引导上的吗？

为什么不写一个直接引导我们想要确定正确的代码呢？

既然我的内心对 `Number` 的运行持有担忧，反过来想只要证明这些担忧是可以通过的那程序不就是没有问题了吗？所以，我对 `Number` 的**测试预期的结果**是：

- 运行 `Number` 时不能出错
- 运行 `Number` 时，要从 `"评分9.6"` 这样的字符串中格式化输出 `'9.6'`

测试预期又称为**“期望结果”**。

什么是断言？

> **断言**就是用代码来判断测试的输出结果是否完全符合**“期望值”**

我现在就撇开测试框架，随便写一个python脚本来测试一下 `Number` 是否可以用：

```
# number_test.py
processor = Number()
test_str = u'评分9.6'      # 测试输入值
expected = '9.6'           # 期望输出值
actual = processor(test_str) # 实际处理结果

if actual == expected:    # 判断实际输出是否符合期望
  print('Number测试通过')
else:
  print('Number测试失败')
```

这样是不是就可以测试出来了？只要在指命行运行这个脚本：

```
(venv) $ python number_test.py
(venv) $ Number测试通过
```

试想想：

1. 如果你有10个类要测试甚至上百个呢？一个一个执行？还是说要另外写一个程序一次性来执行所有的测试文件？
2. 如果测试出错了能不能通过双击出错信息就跳到出错的代码位置呢？
3. 如果某些测试要先初始化一些变量，而其它的又不需要这些变量那该如何呢？
4. 对于更复杂的判断我们是不是要写一堆的`if…else`呢？这种输出性文字写得不累吗？
5. 如果我只想看是不是所有的测试都通过了是不是该一个一个地从这些输出上找'XXX测试失败'的文字才能知道谁通过了谁失败了呢？

测试工具简介 unittest

所有的单元测试框架就是为了解决我们上面提出的一系列问题而存在的。

`unittest` 是python内置的一套功能完备的单元测试框架。它可以很好的地集成在pyCharm中，利用pyCharm的调试器快速定位发现的缺陷与异常。对于python技术栈来说是个非常易学易用的好东东。它自带了以下这些开箱即用的功能：

- 测试运行器 `TestRunner` - 以命令行的方式自动发现与运行所有的测试
- 统一的测试报告 - 可以输出到终端、文件或者pyCharm的集成界面中。
- 一至的测试写法，我们将之称为测试单元 `TestCase`
- 通过测试集`TestSuite`可以对多个不同的测试单元进行分组
- 与pyCharm集成后可以双击自动转跳至发生错误的代码行上。(pyCharm的功能)

配置测试环境

首先，在项目的根本目录添加一个测试目录`tests`，然后要在该目录中增加一个`__init__.py`空文件，这样做的意思是告诉测试运行器`tests`是一个python 包(package)，只有放在包内的测试程序才可以被自动发现，否则你就得在运行的时候手工指定了。

在项目的根目录运行以下的指令：

```
(venv) $ mkdir tests
(venv) $ cd tests
(venv) $ touch __init__.py
```

或者直接在pyCharm中建一个python 包也行。

然后在`tests`目录下添加一个`test_processors.py`的类，在该文件中添加以下的代码:

```
# coding:utf-8
import unittest
from douban.processors import Number, Text, Date, Price


class ProcessorTestCase(unittest.TestCase):

    def test_number_processor(self):
        pass
```

`ProcessorTestCase`就是测试用例必须继承于`unittest.TestCase`，用来分组单元测试，同时提供代码断言库。

`test_number_processor`是我们准备运行单元测试的函数，我们可以将上文中的代码移植至此:

```
def test_number_processor(self):
  tests_text = "评分9.6"
  expected_text = '9.6'
  processor = Number()

  actual = processor([tests_text])
  self.assertEqual(actual[0], expected_text)
```

> 注:这里有一个小小的约定，单元测试函数必须以"testXXX"来命名才能被测试运行器自动发现并认为这是一个单元测试噢。

如果你想让测试更正规一些也可以用你的测试阅意图来命名这个测试函数，例如:

```
def test_number_must_be_format_number_string(self):
```

因为这个`Number`本来就很简单，而且只有一个输出方法，那就没有必要用这么啰嗦的命名了，约定是死的但人是活的，知识应该活学活用。

```
self.assertEqual(actual[0], expected_text)
```

这里就是代码断言，所有的代码断言都会以`assertXXX`开头来命名，当这个断言返回值就会直接通过，被测试执行器认为这个测试已成功，反之就会在测试报告中标记当前的断言判断出现不符而认为测试失败。

unittests提供的代码断言库还是非常地完善的，这里就不一一列举了你可以用pyDoc在终端上输出文档来看:

```
(venv) $ pydoc unittest.TestCase
```

或者直接参考官方的说明

运行测试:

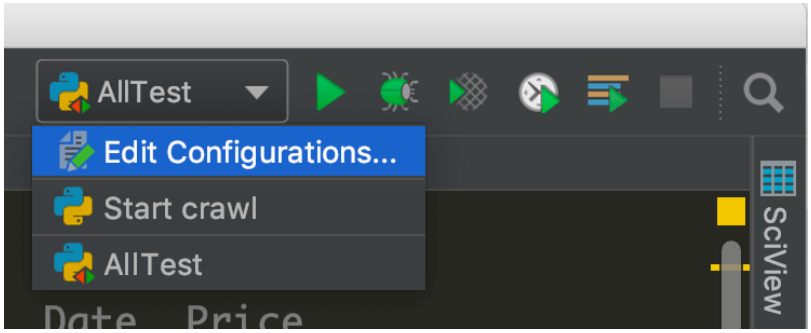
```
(venv)$ python -m unittest
```

然后你会看到以下的输出结果:

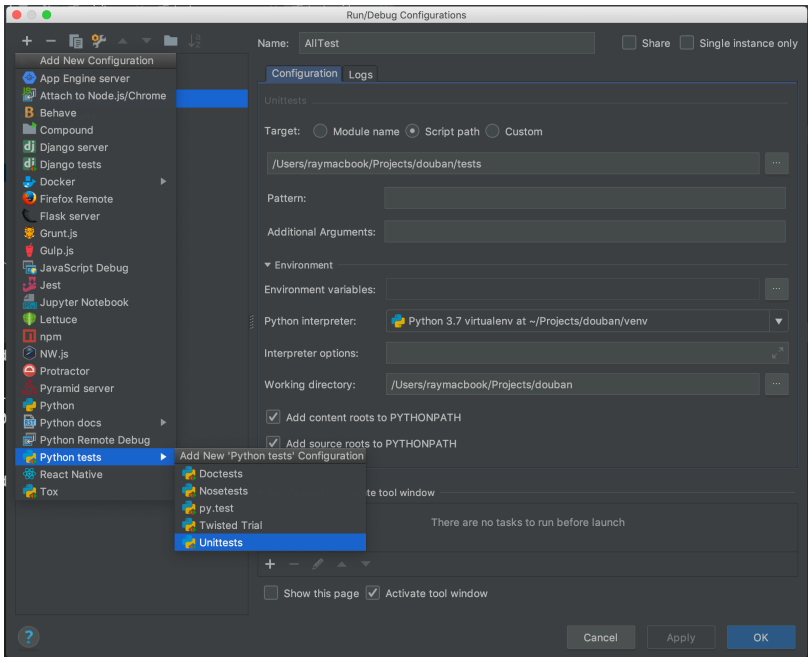
```
(venv) $ python -m unittest
.
----------------------------------------------------------
Ran 1 test in 0.000s

OK
```

集成pyCharm

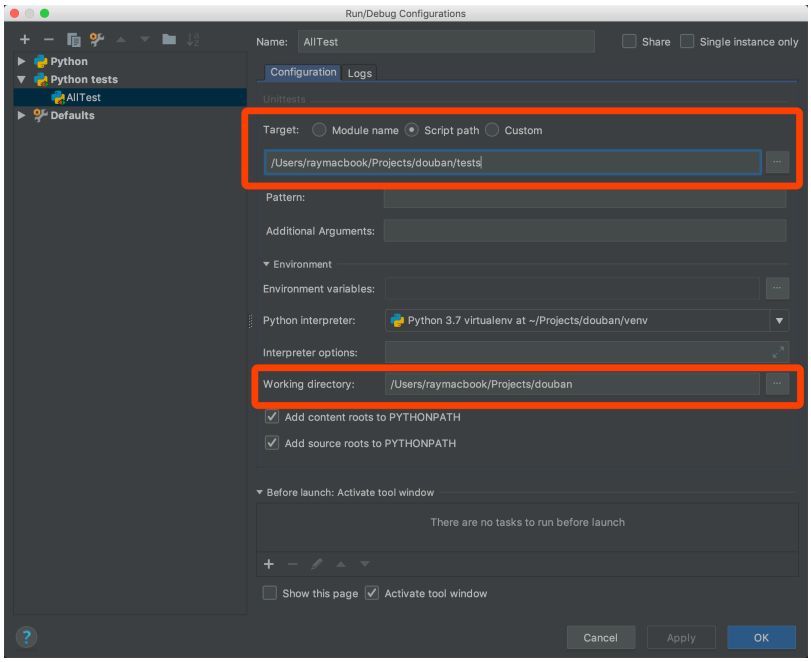
在pyCharm中的测试工具栏点击"Edit Configurations"，如下图示：



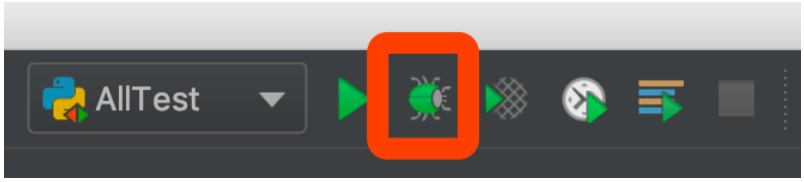
然后添加一个新的配置项目，选择 “Unittests”：



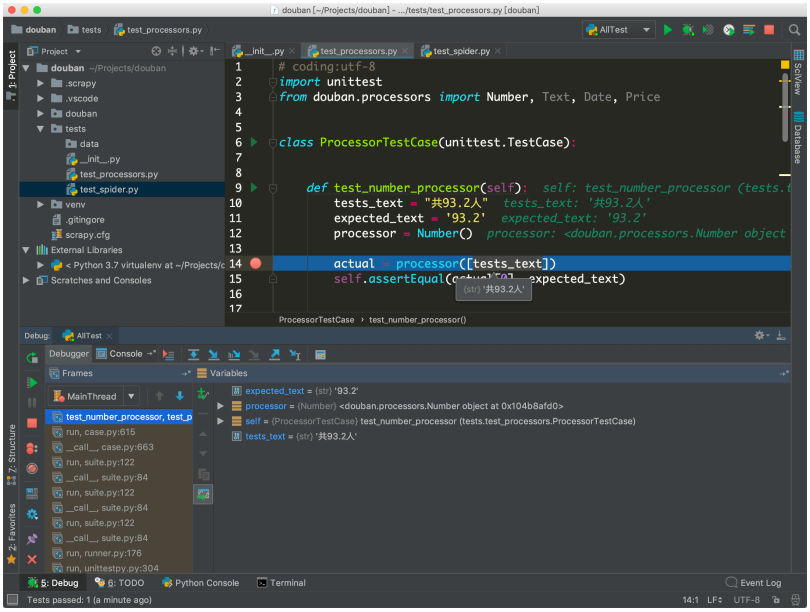
在出现的配置框内填写好测试目录与项目工作目录(项目根目录)：



配置完成后就可以点击测试工具栏中的"debug"按钮：



当程序运行到断点时就会自动打断，并跳至断点的代码行上，此时就可以从观察窗口看到代码中每个变量值与代码的执行逻辑是否正确，断点调试可是开发人员调试时的**大招**，如果连调试都不懂那就跟本算不上一个合格的开发人员了。只有通过调试器你才可以真正地了解程序背后的真相是什么！单步执行你的代码才能完全清楚你的代码是不是按照你的想法来执行的。



为自定义的 Processor 编写测试

既然已万事俱备，那么接下来我们就按照原来设计输入处理器的逻辑来一个一个地编写它们的单元测试,以下是`ProcessorTestCase`的完整代码:

```python
# coding:utf-8
import unittest
from douban.processors import Number, Text, Date, Price


class ProcessorTestCase(unittest.TestCase):

    def test_number_processor(self):
        tests_text = "共93.2人"
        expected_text = '93.2'
        processor = Number()

        actual = processor([tests_text])
        self.assertEqual(actual[0], expected_text)

    def test_text_processor(self):
        tests_text = "<div>This is a text with some <b>html</b> tags</div>"
        expected_text = "This is a text with some html tags"
        processor = Text()

        actual = processor([tests_text])
        self.assertEqual(actual[0], expected_text)

    def test_price_processor(self):
        tests_text = "¥24.2 元"
        expected_text = '24.2'
        processor = Price()

        actual = processor([tests_text])
        self.assertEqual(actual[0], expected_text)

    def test_date_processor(self):
        tests_text = "2015年2月3日"
        expected_text = '2015-02-03T00:00:00'
        processor = Date()

        actual = processor([tests_text])
        self.assertEqual(actual[0], expected_text)
```

小结

这节的重点是让你感性地对TDD建立一个最基本的概念，通过实际的动手实践加深对这些相关概念的理解。当建立了测试环境编写起测试代码并没有想象中那么难不是吗？

有测试与没有测试的项目最大的区别在于你已经不再是用眼睛来判断你的程序是否正确，而是用代码去判断每个你并没有多大信心或者是运行结果让你存疑的代码。通过调试器的单步运行，我们可以清楚地洞悉程序运行的每一步，调用的每个隐含在背后的代码，同时这也是一种在运动中学习代码了解框架的最佳做法。

17 数据提取过程中的类型化方法

19 测试BookSpider

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论