单元测试内部培训

张龙

前言

课程目标

1. 了解单元测试的基本概念
   - 单元测试
   - 理念

2. 掌握单元测试的思想和方法
   - Python单元测试框架
   - Mock

3. 能够写出规范的单元测试

概念

1. 单元&单元测试

程序单元是应用的最小可测试部件。

在过程化编程中，一个单元就是**单个程序、函数、过程**等；

对于面向对象编程，最小单元就是**方法**，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

——维基百科

概念

2. 单元测试

单元测试是用代码测代码

一个“单元测试”是一段**自动化的代码**，这段代码调用被测试的工作单元，之后对这个单元的单个最终结果的某些假设**进行检验**。单元测试几乎都是用单元测试框架编写的。单元测试容易编写，能快速运行。单元测试**可靠、可读，并且可以维护**。只要产品代码不发生变化，单元测试的结果是稳定的。

单元测试的划分不是越小越好，对用户接口的可见度更高，测试会更容易维护

单元测试需要自动化进行，可靠、可读、可维护是单元测试中需要考虑的问题

概念

3. 单元测试 VS 集成测试

	单元测试	集成测试
测试对象	• 方法、函数、类	• 功能、模块
测试方法	• 白盒、灰盒	• 灰盒、黑盒
测试数据	• 构造的测试数据，不依赖其他的测试单元或其他系统	• 真实的数据，和其他的功能、模块有一定的依赖关系
测试执行	• 自动化	• 手工+自动化
测试时间	• 短	• 长

概念

4. W.W.W

When

1. 具体实现代码之前
2. 与具体实现代码同步进行
3. 事后单元测试
4. 对没有单元测试的既有代码进行维护和改造

What

1. 逻辑复杂的
2. 容易出错的
3. 不易理解的
4. 后期需求变更可能性相对比较大的

Who

1. 单元测试应当由具体实现代码的开发者进行
2. 维护和改造者

概念

其它

1. 不要为了单元测试而单元测试
2. 切勿过于追求覆盖率
3. 长期来看，可以提高代码质量、减少维护成本、降低重构难度；短期来看，需要投入更多的人力成本
4. 像艺术作品一样，创作过程和成品是没有标准答案的

单元测试框架

例1. widget.py

```
# 将要被测试的类
class Widget:
    def __init__(self, size = (40, 40)):
        self._size = size
    def getSize(self):
        return self._size
    def resize(self, width, height):
        if width  0  or height < 0:
            raise ValueError, "illegal size"
        self._size = (width, height)
    def dispose(self):
        pass
```

采用手工方式进行单元测试的Python程序员很可能会写出类似例2的测试代码来，

例2. manual.py

```
from widget import Widget
# 执行测试的类
class TestWidget:
    def testSize(self):
        expectedSize = (40, 40);
        widget = Widget()
        if widget.getSize() == expectedSize:
            print "test [Widget]: getSize works perfected!"
        else:
            print "test [Widget]: getSize doesn't work!"
# 测试
if __name__ == '__main__':
    myTest = TestWidget()
    myTest.testSize()
```

单元测试框架

问题

1. 程序员完全可能写出十种不同的测试程序来
2. 需要编写大量的辅助代码才能进行单元测试

解决方案

Python单元测试框架

- The Python unit testing framework，简称为PyUnit
- 标准库的一部分（Python 2.1和更高版本）

单元测试框架

示例

```
    例3. auto.py
from widget import Widget
import unittest
# 执行测试的类
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget()
    def tearDown(self):
        self.widget = None
    def testSize(self):
        self.assertEqual(self.widget.getSize(), (40, 40))
# 构造测试集
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testSize"))
    return suite
# 测试
if __name__ == "__main__":
    unittest.main(defaultTest = 'suite')
```

单元测试框架

轻松测试

1. 导入 import unittest
2. 继承 class Test(unittest.TestCase)
3. 可选项 setUp/tearDown/setUpClass/tearDownClass
4. 自定义类的方法方法名须以test字符作为前缀，考虑各种边界条件
5. 说出你的期待 self.assertXXX

单元测试框架

不常用的几个概念

1. 测试套件
2. 嵌套测试套件

单元测试框架

引出的问题

1. 文件命名规范
   - test_${module_name}.py
2. 如何组织代码
   - 在模块下面建tests/目录，再将单元测试放在这个目录下面

   一来这样避免了测试代码离被测试代码太远的弊端，
   二来通过文件夹组织了起来，也不会对项目的结构造成干扰

   - 所有的测试放在一个tests目录里，内部结构跟源码保持一致

进阶-mock

- 什么是Mock
  - **Mock**者，**模仿、仿造**也
  - Mock 是python的一个**单元测试库**
  - 允许你用mock的对象**代替**你的代码的真正实现部分
  - 允许你**动态设置**对象的属性和方法的返回值
  - 允许你对所测试代码部分的方法调用，属性和变量值进行**断言**
  - 终极目的是**解耦**，即只对当前逻辑负责，而不管嵌套逻辑

进阶-mock

- 轻松测试：
  1. 替换，用Mock对象替换真实对象、方法
  2. 设置属性和返回值，甚至是异常
  3. 执行测试函数
  4. 断言你的期望

  演示

番外篇-代码的坏味道

- 重复代码
- 过长函数
- 过大的类
- 过长的参数列
- 发散式变化（一个类因不同的变化而修改）
- 霰弹式修改（一个变化引起好多类的修改）
- 依恋情结（函数从对象中取出太多的值用来计算）
- Switch惊悚现身（switch和过多的if else）
- 过多的注释（注释被当做除臭剂）

——选自《重构-改善既有代码设计》

参考资料

- http://sebug.net/paper/books/dive-into-python3/unit-testing.html （单元测试实例）
- http://pypi.python.org/pypi/mock#downloads（下载地址）
- http://www.voidspace.org.uk/python/mock/0.8/（Mocking and Testing Library）
- http://docs.python.org/dev/library/unittest.mock（mock object library）

Q&A