单元测试

Unit testing made easy

裴孟齐

目录

测试的进化：version 1

```
1  # portfolio1.py
2
3  class Portfolio(object):
4      """A simple stock portfolio"""
5      def __init__(self):
6          # stocks is a list of lists:
7          #   [[name, shares, price], ...]
8          self.stocks = []
9
10     def buy(self, name, shares, price):
11         """Buy `name`: `shares` shares at `price`."""
12         self.stocks.append([name, shares, price])
13
14     def cost(self):
15         """What was the total cost of this portfolio?"""
16         amt = 0.0
17         for name, shares, price in self.stocks:
18             amt += shares * price
19         return amt
```

测试的进化：version 1

- 手工运行

First test: interactive

```
>>> p = Portfolio()
>>> p.cost()
0.0

>>> p.buy("IBM", 100, 176.48)
>>> p.cost()
17648.0

>>> p.buy("HPQ", 100, 36.15)
>>> p.cost()
21263.0
```

✓ Good: testing the code

✘ Bad: not repeatable

✘ Bad: labor intensive

✘ Bad: is it right?

测试的进化：version 2

- 脚本化

Second test: standalone

```
1  # porttest1.py
2  from portfolio1 import Portfolio
3
4  p = Portfolio()
5  print "Empty portfolio cost: %s" % p.cost()
6  p.buy("IBM", 100, 176.48)
7  print "With 100 IBM @ 176.48: %s" % p.cost()
8  p.buy("HPQ", 100, 36.15)
9  print "With 100 HPQ @ 36.15: %s" % p.cost()
```

```
1  $ python porttest1.py
2  Empty portfolio cost: 0.0
3  With 100 IBM @ 176.48: 17648.0
4  With 100 HPQ @ 36.15: 21263.0
```

✓ Good: testing the code

✓ Better: repeatable

✓ Better: low effort

✘ Bad: is it right?

测试的进化：version 3

- 显式地测量结果

Third test: expected results

```
4  p = Portfolio()
5  print "Empty portfolio cost: %s, should be 0.0" % p.cost()
6  p.buy("IBM", 100, 176.48)
7  print "With 100 IBM @ 176.48: %s, should be 17648.0" % p.cost()
8  p.buy("HPQ", 100, 36.15)
9  print "With 100 HPQ @ 36.15: %s, should be 21263.0" % p.cost()
```

```
1  $ python porttest2.py
2  Empty portfolio cost: 0.0, should be 0.0
3  With 100 IBM @ 176.48: 17648.0, should be 17648.0
4  With 100 HPQ @ 36.15: 21263.0, should be 21263.0
```

✓ **Good: repeatable with low effort**

✓ **Better: explicit expected results**

✘ **Bad: have to check the results yourself**

测试的进化：version 4

- 自动测量结果

Fourth test: check results automatically

```
4  p = Portfolio()
5  print "Empty portfolio cost: %s, should be 0.0" % p.cost()
6  assert p.cost() == 0.0
7  p.buy("IBM", 100, 176.48)
8  print "With 100 IBM @ 176.48: %s, should be 17648.0" % p.cost()
9  assert p.cost() == 17648.0
10 p.buy("HPQ", 100, 36.15)
11 print "With 100 HPQ @ 36.15: %s, should be 21263.0" % p.cost()
12 assert p.cost() == 21263.0
```

```
1  $ python porttest3.py
2  Empty portfolio cost: 0.0, should be 0.0
3  With 100 IBM @ 176.48: 17648.0, should be 17648.0
4  With 100 HPQ @ 36.15: 21263.0, should be 21263.0
```

✓ Good: repeatable with low effort

✓ Good: explicit expected results

✓ Good: results checked automatically

测试的进化：version 4

- 自动测量结果

Fourth test: what failure looks like

```
$ python porttest3_broken.py
Empty portfolio cost: 0.0, should be 0.0
With 100 IBM @ 176.48: 17648.0, should be 17600.0
Traceback (most recent call last):
  File "porttest3_broken.py", line 9, in <module>
    assert p.cost() == 17600.0
AssertionError
```

✓ Good: repeatable with low effort

✓ Good: expected results checked automatically

✓ OK: visible failure visible, but cluttered output

✘ Bad: failure stops tests

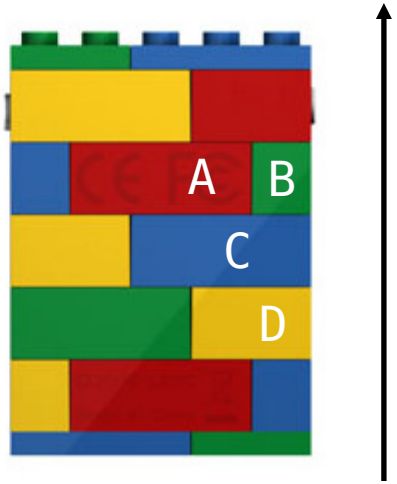
测试的进化：自动化测试

- 单元测试框架

```
1  # test_port1.py
2
3  import unittest
4  from portfolio1 import Portfolio
5
6  class PortfolioTest(unittest.TestCase):
7      def test_buy_one_stock(self):
8          p = Portfolio()
9          p.buy("IBM", 100, 176.48)
10         assert p.cost() == 17648.0
```

什么是单元测试

- 针对模块的测试
- 应用的最小可测试部件



为什么要单元测试

- 单元测试的几个好处
  - 保证程序的正确性
  - 从使用者角度思考问题、设计方案
  - 提供可以运行的文档
  - 强化代码各模块的独立性
  - 方便未来的代码重构
  - 测试比调试更容易
  - 消除恐惧

如何编写单元测试

好的测试

- 自动化
- 快速
- 靠谱
- 富信息
- 专注

如何编写单元测试

- Java
  - Junit
  - TestNG
  - Spring Test
  - Mockito

编写单元测试：JUnit

- 注解
  - @Test
  - @SetUp
  - @TearDown
  - @BeforeClass
  - @RunWith
- 类
  - Org.junit.Assert
  - Org.junit.TestCase
  - Org.junit.TestResult
  - org.junit.TestSuite
    一组 TestCase

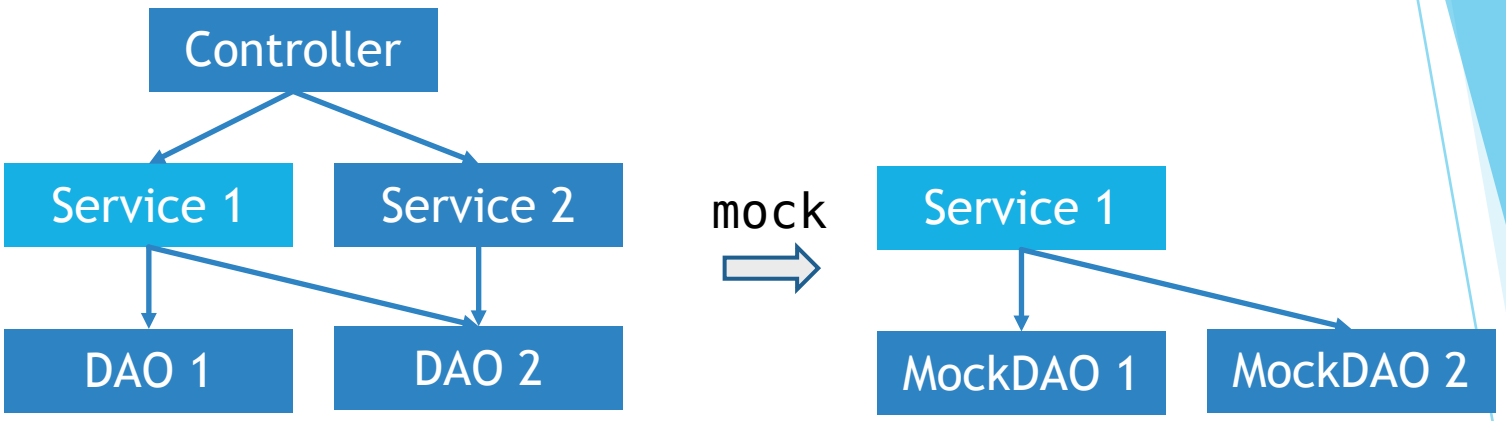
编写单元测试：TestNG

- @DataProvider
  参数化测试用例
  参数化对象：测试函数的参数
- Junit 的参数化方案：Parameterized
  参数化对象：整个 TestCase 的参数

编写单元测试：Spring Test

- @RunWith(SpringJUnit4ClassRunner.class)
  启动 Spring 对测试类的支持
- @ContextConfiguration
  指定 Spring 配置文件或者配置类的位置
- @Transactional
  启用自动的事务管理
- @Autowired
  注入 bean

编写单元测试：Mockito



- List<String> list = **mock**(List.class);
- **when**(list.get(0)).**thenReturn**("helloworld");
- **verify**(list).get(0);
- when(list.get(**anyInt()**)).thenReturn("hello","world");

TDD（Test Driven Development）

- 先写测试，后写实现
- 红->绿->红->重构

Pros	Cons
• 在开发前先进行好设计，拥有全面的视角 • 留下了良好的文档 • 开发过程更有自信	• 容易遗漏边界测试 • 一些应用本身比较难构造单元测试 • 只为了满足测试开发，忽视了实际需求