

Lecture 5: 测试驱动开发

Xiao-Xin Li

Zhejiang University of Technology

Revision: 2023/04/27, 2023/05/03

测试与调试是程序设计者必备的基础技能，学习程序设计，我们首先应该从学习测试和调试开始。

关键字：测试驱动开发

1. 测试驱动开发

作为程序设计的初学者，我们写代码的顺序是怎样的？

- A. 先写产品代码，再写测试代码；
- B. 先写测试代码，再写产品代码；
- C. 边写产品代码，边写测试代码。

很多同学的做法应该是选 A：先写产品代码，再写测试代码。然而，这是严重错误的。

- 初学者可能会犯很多幼稚的错误，等辛辛苦苦把代码写完，再去测试，往往会发现程序中到处都是错误，自信心已经彻底崩溃了；
- 为了完成一个程序，我们往往会产生很多idea，我们的程序正是由这些idea构成的，对于每个idea，都需要边写代码，边测试；
- 程序的主要逻辑往往蕴含在测试代码中，而非产品代码中。

测试驱动（TDD）开发是个很重要的程序设计理念，是敏捷开发中的一项核心实践和技术。

测试代码一定要写在独立的 `test` 方法中吗？答案是否定的，经验丰富的开发人员往往是将测试代码蕴含在产品代码中。

2. 基于测试搭建产品代码的主要框架

在产品代码中，我们可以将测试代码放在程序的 `main` 方法中，也可以放在类的 `__call__()` 方法中。

```

1 if __name__ == '__main__':
2     """
3     整个程序的测试代码可以写在这里，主要是给出程序的输入和“预期输出”
4     为简单起见，可以省去“预期输出”，但不利于进行自动化测试。
5     """
6     player_scores = '\
7     李: 17 6 3 3    2 5 3 6 8\
8     楼: 6   9 5 10 14 3 8 4 6\
9     翁: 3   4 8 9    3 5 4 6 6'
10
11     AnalyScore(player_scores)()
12
13     print()
14     AnalyScore('scores.txt')()

```

```

1 class AnalyScore:
2     def __call__(self):
3         """
4         整个类的框架性代码可以在写类的 __call__ 方法中，
5         主要给出类的主要逻辑和模块，并用于驱动各模块的执行。
6         """
7         self.read_scores()
8         self.parse_scores()
9         self.print_scores()
10        self.report_winner()
11

```

2. 基于测试编写产品类中的主要方法

```

1 class AnalyScore:
2     def read_scores(all_scores):
3         """
4         1. 从字符串读取分数
5         2. 从文件读取分数
6         主要idea: 为了方便后续统一处理，要尽可能使得在两种情况下读取的分数的格式尽可能一致
7         主要问题: 如何区分输入的 all_scores 是字符串还是文件？
8         """
9         if not isinstance(all_scores, str):
10            return all_scores
11        self._read_score_from_file(all_scores)
12
13    def parse_scores(self, all_scores):

```

```

14         """
15         分离姓名和方法
16         主要问题：
17             (1) 如何快速分离？
18             (2) 如何存储分离后的结果（采用什么数据结构）？ 主要由后续如何使用该数据来决定。
19         """
20         self._split_by_token() # 实现该方法相对复杂，需要使用后面将会提到的“调试”技术
21
22         # 数据结构1: list
23         self.all_scores = ['name1', score_list1, 'name2', score_list2, ...]
24
25         # 数据结构2: dict
26         self.all_scores = {'name1': score_list1, 'name2': score_list2, ...}
27
28         # 数据结构2': dict
29         self.all_scores = dict(name1=score_list1, name2=score_list2, ...)
30
31     def print_scores_list(self):
32         """ self.all_scores 为 list """
33         all_scores = iter(self.all_scores)
34         for name, score in zip(all_scores, all_scores):
35             print(name, score)
36
37     def print_scores_dict(self):
38         """ self.all_scores 为 dict """
39         for name, score in all_scores.items():
40             print(name, score)
41
42     def __call__(self):
43         """
44         整个类的框架性代码可以在写这里，主要给出类的主要逻辑和模块，并驱动各模块的执行
45         """
46         self.read_scores()
47         self.parse_scores()
48         self.print_scores()
49         self.report_winner()
50

```

3. 重构(Refactor)主类 `AnalyScore`：用多种方法实现产品类

```

1 class AnalyScore:
2     def read_scores(all_scores):
3         if not isfile(all_scores):
4             return all_scores

```

```

5         self._read_score_from_file(all_scores)
6
7     def parse_scores(self, all_scores):
8         self._split_by_token()
9         pass
10
11    def print_scores(self):
12        pass
13
14    def __call__(self):
15        """
16        整个类的框架性代码可以在写这里，主要给出类的主要逻辑和模块，并驱动各模块的执行
17        """
18        self.read_scores()
19        self.parse_scores()
20        self.print_scores()
21        self.report_winner()
22

```

```

1 class AnalyScoreViaList(AnalyScore):
2     def parse_scores(self, all_scores):
3         super().parse_scores()
4         self.all_scores = ['name1', score_list1, 'name2', score_list2, ...]
5
6     def print_scores(self):
7         """ self.all_scores 为 list """
8         all_scores = iter(self.all_scores)
9         for name, score in zip(all_scores, all_scores):
10             print(name, score)

```

```

1 class AnalyScoreViaList2Dict(AnalyScoreViaList):
2     def parse_scores(self):
3         """ 将 self.all_scores 从 list 转换为 dict """
4         super().parse_scores()
5         self.all_scores = dict(zip(self.all_scores[::2], self.all_scores[1::2]))

```

```

1 class AnalyScoreViaDict(AnalyScore):
2     def parse_scores(self):
3         """ self.all_scores 为 dict """
4         self.all_scores = {'name1': score_list1, 'name2': score_list2, ...}
5

```

```

6     def print_scores(self):
7         """ self.all_scores 为 dict """
8         for name, score in all_scores.items():
9             print(name, score)

```

4. 基于断言重写 main 方法

```

1  if __name__ == '__main__':
2      player_scores = '\
3      李: 17 6 3 3   2 5 3 6 8\
4      楼: 6   9 5 10 14 3 8 4 6\
5      翁: 3   4 8 9   3 5 4 6 6'
6
7      expected_result = [
8          '李: [17, 6, 3, 3, 2, 5, 3, 6, 8] = 53',
9          '楼: [6, 9, 5, 10, 14, 3, 8, 4, 6] = 65',
10         '翁: [3, 4, 8, 9, 3, 5, 4, 6, 6] = 48',
11         '翁 won the game!'
12     ]
13
14     analy = AnalyScoreViaList(player_scores)
15     analy()
16     # 这里，一个问题：如何让产品代码在控制台原样输出，同时，还收集控制输出的结果呢？
17     assert analy.outputs == expected_result
18
19     analy = AnalyScoreViaList2Dict(player_scores)
20     analy()
21     assert analy.outputs == expected_result
22
23     analy = AnalyScoreViaDict(player_scores)
24     analy()
25     assert analy.outputs == expected_result

```

5. 基于 PyTest 构建单元测试代码

1. 编写单元测试代码：

- a. 以 `test` 开头，定义测试方法；
- b. 以 `Test` 开头，定义测试类，注意，测试类中不能使用 `__init__()` 方法，否则会报错：
empty suite；

2. 使用测试框架 `PyTest` :
 - a. 一个好的测试框架，应该具有统计和报告bug的能力；
 - b. 一个好的测试框架，应该能够被充分执行，而不会被中间产生的某个bug中断执行；
3. PyCharm可以和 `PyTest` 很好地结合：一旦装了 `PyTest` ，你会发现测试方法或类旁边出现了绿色按钮。
4. 回顾：3.2 Python 虚拟环境。
 - a. 如何在命令行下创建虚拟环境？
 - b. 为什么我们要自己管理虚拟环境，而不是让PyCharm来管理？
 - c. 如何在命令行下快速启动虚拟环境？
5. 安装 `PyTest` :

```
1 pip install pytest
```

6. 编写测试方法：

```
1 def test_analy():
2     player_scores = '\
3         李: 17 6 3 3   2 5 3 6 8\
4         楼: 6  9 5 10 14 3 8 4 6\
5         翁: 3  4 8 9   3 5 4 6 6'
6
7     expected_result = [
8         '李: [17, 6, 3, 3, 2, 5, 3, 6, 8] = 53',
9         '楼: [6, 9, 5, 10, 14, 3, 8, 4, 6] = 65',
10        '翁: [3, 4, 8, 9, 3, 5, 4, 6, 6] = 48',
11        '翁 won the game!'
12    ]
13
14    # 把所有的测试方法都放在一个篮子中，
15    # 一旦某个测试代码“断言”失败了，
16    # 后续所有测试代码都丧失了执行的机会。
17    analy = AnalyScoreViaList(player_scores)
18    analy()
19    assert analy.outputs == expected_result
20
21    analy = AnalyScoreViaList2Dict(player_scores)
22    analy()
23    assert analy.outputs == expected_result
24
```

```
25     analy = AnalyScoreViaDict(player_scores)
26     analy()
27     assert analy.outputs == expected_result
```

7. 编写测试类:

- a. 一个测试方法在执行时断言失败，不会影响其它测试方法的执行；
- b. Test Suites: 所有测试方法形成了测试套件（Test Suites）。

```
1  class TestAnaly:
2      # def __init__(self):
3      # 注意，测试类中不能使用__init__()方法，否则会报错: empty suite
4
5      player_scores = '\
6          李: 17 6 3 3   2 5 3 6 8\
7          楼: 6  9 5 10 14 3 8 4 6\
8          翁: 3  4 8 9   3 5 4 6 6'
9
10     expected_result = [
11         '李: [17, 6, 3, 3, 2, 5, 3, 6, 8] = 53',
12         '楼: [6, 9, 5, 10, 14, 3, 8, 4, 6] = 65',
13         '翁: [3, 4, 8, 9, 3, 5, 4, 6, 6] = 48',
14         '翁 won the game!'
15     ]
16
17     def test_1(self):
18         analy = AnalyScore(self.player_scores)
19         analy.call1()
20         assert analy.outputs == self.expected_result
21
22     def test_2(self):
23         analy = AnalyScore(self.player_scores)
24         analy.call2()
25         assert analy.outputs == self.expected_result
```