第 11 章

测试驱动开发

测试驱动开发（TDD）是制造高质量软件的一种简单技术。它在 Python 社区被广泛地应用，在使用静态类型语言的社区中可能用得更多。这可能是由于开发人员认为编译程序可以完成大部分测试，它在生成二进制码的时候会进行许多检查。

因此，他们在开发阶段将停止执行测试。但是这往往会导致产生低质量的代码，以及花费很长时间的调试来使其正常工作。记住，大部分的缺陷与不良的语法无关，而与逻辑错误和可能导致重大破坏的微小误解有关。

本章将分成两个部分：

- 我不测试，提倡 TDD 并快速地描述如何借助标准库进行这种开发；
- 我测试，这是为实际进行测试并希望从中获得更多价值的开发人员准备的。

11.1 我 不 测 试

如果相信 TDD 的价值，请转到下一节。下一节中将关注于更高级的技术，以及在编写测试时如何简化工作。本节主要是为那些还没有使用这种方法的开发人员准备的，并且尝试倡导它的使用。

11.1.1 测试驱动开发原理

TDD 由编写覆盖所需功能的测试用例，然后编写该功能两部分工作组成。换句话说，将在代码存在之前编写测试用例。

例如，如果开发人员要编写一个计算一系列数字的平均值的函数，那么将首先编写几个

使用它的实例，并提供预期的结果，如下所示。

```
assert average(1, 2, 3) == 2
assert average(1, -3) == -1
```

这些例子可以由另一个人提供。现在将着手实现该函数，直到这两个示例能够正常工作，如下所示。

```
>>> def average(*numbers):
...     return sum(numbers) / len(numbers)
...
>>> assert average(1, 2, 3) == 2
>>> assert average(1, -3) == -1
```

缺陷或非预期的结果是该函数应该能够处理的新的测试用例，如下所示。

```
>>> assert average(0, 1) == 0.5
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
AssertionError
```

需要对代码做相应的修改，直到新的测试通过为止，如下所示。

```
>>> def average(*numbers):
...     # 确保可以使用浮点型
...     numbers = [float(number) for number in numbers]
...     return sum(numbers) / float(len(numbers))
...
>>> assert average(0, 1) == 0.5
```

更多的测试用例将改进代码，如下所示。

```
>>> try:
...     average()
... except TypeError:
...     # 希望空序列抛出一个类型错误
...     pass
...
Traceback (most recent call last):
   File "<stdin>", line 2, in <module>
   File "<stdin>", line 3, in average
ZeroDivisionError: integer division or modulo by zero
```

```
>>>
>>> def average(*numbers):
...     if numbers == ():
...         raise TypeError(('You need to provide at '
...                          'least one number'))
...     numbers = [float(number) for number in numbers]
...     return sum(numbers) / len(numbers)
...
>>> try:
...     average()
... except TypeError:
...     pass
...
```

到此，可以将所有的测试收集在 test 函数中，它将在每次修改代码时运行，如下所示。

```
>>> def test_average():
...     assert average(1, 2, 3) == 2
...     assert average(1, -3) == -1
...     assert average(0, 1) == 0.5
...     try:
...         average()
...     except TypeError:
...         pass
...
>>> test_average()
```

每次修改时，test_average 和 average 将一起修改，并且再次运行以确认所有测试用例仍然能够通过。其使用方法是将所有测试集中在当前包的 tests 文件夹中，每个模块在那里都可以有与之对应的测试模块。

这个方法提供以下好处：

- 避免软件退化；
- 增进代码质量；
- 提供最好的低层级文档；
- 更快地产生健壮的代码。

1. 避免软件退化

在开发生涯中，都要面对软件退化的问题。软件退化就是修改引入的新缺陷。退化的发

生是因为无法在某个点猜测代码库中的某个修改可能会导致的结果。修改一些代码可能破坏其他的功能，有时候会导致可怕的副作用，比如悄悄地对数据产生破坏。

为了避免退化，在每次修改时都应对软件的所有功能进行测试。

将代码库开放给多个开发人员将会放大这个问题，因为每个人都无法完全了解所有开发活动。虽然版本控制系统能够避免冲突，但是无法避免所有不必要的交互。

TDD 有助于减少软件退化。整个软件可以在每次修改之后自动测试。只要每个功能都有合适的测试集就能做到这一点。当真正实现 TDD 时，测试库和代码库将一起增长。

因为完整的测试可能持续相当长的时间，所以将其委托给 buidbot 是个好做法，它可以在后台执行这项工作（这在第 8 章中已经介绍过）。但是在本地重新运行测试应该由开发人员手工完成，至少在所关心的模块上。

2．增进代码质量

在编写一个新的模块、类或函数时，开发人员关注于如何编写它以及如何生成最佳的代码。但是当他聚焦于算法时，可能会忘记了用户的看法：他的函数何时、如何使用？参数的使用是否简便和有逻辑性？API 的名称是否正确？

应用前面的章节中介绍的技巧，例如选择好的名称，就可以做到这一点。但是要使其具有效率，可用的方法只有编写测试用例。这是开发人员意识到所写的代码是否有逻辑性和易于使用的时机。往往，在模块、类或函数完成之后，马上就会发生第一次重构。

编写测试，也就是代码的一个具体使用场景，对于获取此类用户观点是有帮助的。因而，开发人员使用 TDD 往往能够生成更好的代码。

测试计算很复杂又带有副作用的函数是很困难的。注重测试的代码在编写时应该具有更清晰和模块化的结构。

3．提供最佳的开发人员文档

测试是开发人员学习软件工作方式的最佳途径。它们的代码主要针对使用场景。阅读它们，能够对代码的工作方式建立快速而且深入的观察。有时候，一个例子值一千句话。

始终与代码库一起更新的测试，将能够生成软件所能拥有的最好的开发人员文档。测试不会像文档一样静止不动，否则就会失效。

4．更快地产生健壮的代码

不进行测试将会导致需要花费很长的调试时间。在软件的一个部分中的缺陷可能会在“风马牛不相及”的另一个部分中发现。因此，不知道责任在谁，从而花费了过度的调试时间。测试失败是对付小缺陷的更好时机，因为可以获得更好的提示以了解问题的真正所在。而且测试往往比调试更加有趣，因为它是编码工作。

如果度量用于修复代码和编写代码的总时间，一般都会比 TDD 方法所需的时间更长。这在开始编写新代码时并不明显，因为建立测试环境和编写几个测试用例所需的时间看起来要比只编写代码的时间长得多。

但是有些测试环境确实难以建立。例如，当代码与一个 LDAP 或 SQL 服务器交互时，编写测试用例就不直接。这将在 11.2.2 小节中介绍。

11.1.2 哪一类测试

对于任何软件，都可以应用多种测试，其中最主要的是验收测试（或功能测试）和单元测试。

1. 验收测试

验收测试关注于功能，软件被看作一个黑盒子。它只是确定软件确实完成了预期的事情，使用和用户相同的介质，并控制输出。这些测试一般在开发周期之外编写，以验证应用程序符合要求。它们一般像是针对软件的一个检查列表。这些测试往往不是通过 TDD 完成的，而是由管理人员甚至客户来构建。在这种情况下，它们被称作用户验收测试。

同样，验受测试也应该使用 TDD 原则来完成。测试可以在功能编写之前提供。开发人员一般根据功能说明书来制作一些验收测试用例，并确保代码能够通过这些测试。

用于编写这些测试的工具取决于软件提供的用户界面。Python 开发人员最常用的工具如表 11.1 所示。

表 11.1 Python 开发人员常用工具

应用程序类型	工　具
应用程序	Selenium（用于带 JavaScript 的 Web UI）
Web 应用程序	zope.testbrowser（不测试 JS）
WSGI 应用程序	paste.test.fixture（不测试 JS）
Gnome 桌面应用程序	dogtail
Win32 桌面应用程序	pywinauto

若想获得包含更多功能测试工具的列表，可以访问 Grig Gheorghiu 维护的一个 wiki 页面 —— http://www.pycheesecake.org/wiki/ PythonTestingToolsTaxonomy。

2. 单元测试

单元测试是完全适合采用 TDD 方法的低层级测试。它们关注于单个模块（如一个单元）

并且为之提供测试，它不涉及其他任何模块。测试将模块与应用程序的其余部分隔离开。当需要外部依赖项（诸如数据库访问）时，将由仿真或模拟对象来代替。

3．Python 标准测试工具

Python 在标准程序库中提供了两个用来编写测试的模块：

- unittest（http://docs.python.org/lib/module-unittest.html）　最初是 Steve Purcell 编写的（以前叫 PyUnit）；
- doctest（http://docs.python.org/lib/module-doctest.html）　一个文字测试工具。

（1）unittest

unittest 为 Python 提供的功能基本上和 JUnit 为 Java 所做的一切相同。它提供一个名为 TestCase 的基类，这个基类有一个用来验证调用输出的很大的方法集。

创建这个模块是为了编写单元测试，但是也可以用来编写验收测试，只要测试需要使用用户界面。有些测试框架提供了驱动工具的助手类，例如 unittest 之上的 Selenium。

要为一个模块编写一个简单的单元测试，首先从 TestCase 继承一个子类，然后再编写带有 test 前缀的方法。前面的示例，写出来的结果应该如下所示。

```
>>> import unittest
>>> class MyTests(unittest.TestCase):
...     def test_average(self):
...         self.assertEquals(average(1, 2, 3), 2)
...         self.assertEquals(average(1, -3), -1)
...         self.assertEquals(average(0, 1), 0.5)
...         self.assertRaises(TypeError, average)
...
>>> unittest.main()
.
----------------------------------------------------------
Ran 1 test in 0.000s
OK
```

main 函数扫描上下文并查找从 TestCase 继承的类。实例化这些类，然后运行所有以 test 开始的方法。

如果 average 函数在 utils.py 模块中，那么 test 类将被称为 UtilsTests，并且被写入 test_utils.py 文件中，如下所示。

```
import unittest
from utils import average
```

```
class UtilsTests(unittest.TestCase):
    def test_average(self):
        self.assertEquals(average(1, 2, 3), 2)
        self.assertEquals(average(1, -3), -1)
        self.assertEquals(average(0, 1), 0.5)
        self.assertRaises(TypeError, average)
    if __name__ == '__main__':
        unittest.main()
```

现在，每当 utils 模块发生改变时，test_utils 模块就将获得更多的测试。

为了正常工作，test_utils 模块需要有在上下文中可用的 utils 模块。这是因为两个文件在同一个文件夹中，或者测试运行程序将 utils 模块放在 Python 路径下。

Distutils develop 命令在这里很有帮助。

要在整个应用程序之上运行测试，那么将预先假定拥有一个脚本，这个脚本在所有测试模块之外构建一个测试活动（test campaign）。unittest 提供了一个 TestSuite 类，可以聚集多个测试并将它们作为一个测试活动来运行，只要它们都是 TestCase 或 TestSuite 的子类实例就行。

按照惯例，test 模块将提供一个 test_suite 函数，它将返回一个该模块在命令行下调用时用于__main__部分，或者为测试运行程序所用的 TestSuite 实例，如下所示。

```
import unittest
from utils import average
class MyTests(unittest.TestCase):
    def test_average(self):
        self.assertEquals(average(1, 2, 3), 2)
        self.assertEquals(average(1, -3), -1)
        self.assertEquals(average(0, 1), 0.5)
        self.assertRaises(TypeError, average)
class MyTests2(unittest.TestCase):
    def test_another_test(self):
        pass
def test_suite():
        """builds the test suite."""
        def suite(test_class):
            return unittest.makeSuite(test_class)
```

```
        suite = unittest.TestSuite()
        suite.addTests((suite(MyTests), suite(MyTests2)))
        return suite
if __name__ == '__main__':
        unittest.main(defaultTest='test_suite')
```

从 shell 程序上运行这个模块将打印出测试活动的输出，如下所示。

```
$ python test_utils.py
..
----------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```

一般来说，通过一个全局脚本就能运行所有的测试，它将在代码树上查找并运行测试。这被称作测试发现（test discovery），稍后将介绍它。

（2）doctest

doctest 是一个在交互式命令行会话中使用的模块，它用来从文本文件或 docstrings 中提取片段，并且回放它们，以检查实例中写入的输出与实际输出相同。

例如，下面这个文本文件（test.txt）可以作为测试来运行。

```
Check that the computer CPU is not getting too hot::
    >>> 1 + 1
    2
```

doctest 提供一些提取和运行测试的功能，如下所示。

```
>>> import doctest
>>> doctest.testfile('test.txt', verbose=True)
Trying:
   1 + 1
Expecting:
   2
ok
1 items passed all tests:
   1 tests in test.txt
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
*** DocTestRunner.merge: 'test.txt' in both testers; summing outcomes.
(0, 1)
```

使用 doctest 有许多好处：

- 包可以通过示例创建文档和测试；
- 文档示例总是最新的；
- 使用 doctest 中的示例来编写包，对具备前一小节中描述的用户视角有帮助。

但是，不要用 doctest 来使单元测试过期，它们应该只被用于在文档中提供人类可读的示例。换句话说，当测试关注低层级事务或者需要会使文档混乱的复杂测试装置时，不要使用 doctest。

有些 Python 框架（如 Zope）广泛地应用 doctest，有时候不熟悉代码的人会批评它们。一些 doctest 确实令人难以阅读和理解，因为示例破坏了技术写作的一个规则：它们不能在一个简单的命令提示符下被理解和运行，并且要求读者具有大量的知识。所以，本来是用来帮助新来者的文档，却因为代码示例而使其难以阅读。这些都是基于复杂的测试装置甚至特殊的测试 API，通过 TDD 建立的 doctest。

正如本章中关于文档的说明那样，当使用作为包文档的一部分的 doctest 时，应小心遵循技术写作的 7 条规则。

至此，应该对 TDD 所带来的好处建立了较好的总体认识。如果你仍然没有信心，可以在少数模块上进行尝试。使用 TDD 编写一个模块并度量构建它的时间，然后调试并重构它。你应该很快发现这是一种优秀的方法。

11.2 我 测 试

本节将描述几个开发人员在编写测试时会碰到的问题，以及针对它们的解决方法。此外，还将简略介绍一下社区中可用的测试运行程序和工具。

11.2.1 Unittest 的缺陷

unittest 模块在 Python 2.1 中就被引入了，并且已经被开发人员大量使用。但是，曾经因为 unittest 的弱点和局限性而招致失败的人开发了一些替代的测试框架，并且其在社区中已经兴起。

以下是常见的批评。

- 该框架笨重，因为：
  - 必须在 TestCase 子类中编写所有的测试；

- 必须在方法名称前加上 test；
- 被要求使用 TestCase 提供的断言方法；
- 必须为所要运行的测试活动建立测试套件。

- 框架难以扩展，因为它需要使用许多类的继承或诸如装饰器（decorator）之类的技巧。
- 测试装置有时难以组织，因为 setUp 和 tearDown 机制绑定在 TestCase 级别上，但是它们在每个测试时运行一次。换句话说，如果测试装置关注许多测试模块，那么它的创建和清理工作将十分不容易。
- 在 Python 软件之上不容易运行测试活动，必须编写额外的脚本来收集测试，聚集并运行它。

为了在编写测试时免受框架的僵化特性（这点和 Junit 很相似），需要一种更轻量级的方法。因为 Python 不要求工作在一个 100%的类环境中，所以提供一种不基于继承的更具 Python 风格的测试框架会更好。普通的方法是：

- 提供简单的将任何函数或类标记为测试的方法；
- 通过插件系统扩展框架；
- 为所有测试级别提供完整的测试装置环境，包括完整的活动、一组模块级和测试级的测试；
- 提供基于测试发现的测试运行程序，并提供大量的选项集。

Python 核心开发人员意识到 unittest 的弱点，并已经在 Python 3k 中完成了一些改进工作。

有人正在开发用于 Python 3k 的 unittest 替代品，其中一个项目是基于 test_harness 实现的（参见 http://oakwinter.com/code/）。

11.2.2　Unittest 替代品

有些第三方工具尝试通过提供 unittest 扩展的形式解决刚才提到的问题。

最常用的如下。

- nose　http://www.somethingaboutorange.com/mrl/projects/nose
- py.test　http://codespeak.net/py/dist/test.html

1．nose

nose 主要是一个具有强大发现功能的测试运行程序。它拥有大量的选项，允许在 Python 应用程序中运行所有类型的测试活动。

可以使用 easy_install 安装它，如下所示。

```
$ easy_install nose
Searching for nose
Reading http://pypi.python.org/simple/nose/
Reading http://somethingaboutorange.com/mrl/projects/nose/
...
Processing dependencies for nose
Finished processing dependencies for nose
```

（1）测试运行程序

安装完后，命令提示符下就多了一条 nosetests 命令。可以直接使用它来运行 11.1 节中介绍的测试，如下所示。

```
$ nosetests -v
test_average (test_utils.MyTests) ... ok
test_another_test (test_utils.MyTests2) ... ok
builds the test suite. ... ok
----------------------------------------------------------------
Ran 3 tests in 0.010s
OK
```

nose 将递归浏览当前目录以发现测试，并创建一个自己的测试套件。这个简单的功能与 unittest 中启动测试所做的工作比起来已经是个优势。现在不需要为构建和运行测试活动准备样板代码了，唯一需要做的事就是编写测试类。

（2）编写测试

nose 向前发展了一步，它将运行所有名称符合正则表达式（(?:^|[b_.-])[Tt]est）的类和函数，所在的模块也匹配该表达式。一般而言，所有以 test 开头并且位于符合该模式的模块中的可调用部件也将作为测试执行。

例如，test_ok.py 将被 nose 识别并运行，如下所示。

```
$ more test_ok.py
def test_ok():
    print 'my test'
$ nosetests -v
test_ok.test_ok ... ok
----------------------------------------------------------------
Ran 1 test in 0.071s
OK
```

常规的 TestCase 类和 doctests 也会被执行。

最后，nose 提供和 TestCase 方法类似的断言函数。但是这些都将作为函数提供，使用 PEP8 命名约定，而不是 unittest 使用的 Java 约定（参见 http://code.google.com/p/python-nose/wiki/TestingTools）。

（3）编写测试装置

nose 支持 3 种级别的测试装置：

- 包级别　例如，setup 和 teardown 函数可以被添加到存储所有包的测试的文件夹的 __init__.py 模块中；
- 模块级　测试模块可以有自己的 setup 和 teardown 函数；
- 测试级别　可调用对象也可以提供使用 with_setup 装饰器的测试装置函数。

例如，要在模块和测试级别设置一个测试装置，可以使用以下代码。

```
def setup():
    # setup 代码，为整个模块启动
    ...
def teardown():
    # tear down 代码，为整个模块启动
    ...
def set_ok():
   # 只针对 test_ok 的 setup 代码
   ...
@with_setup(set_ok)
def test_ok():
   print 'my test'
```

（4）与 setuptools 和插件系统的集成

最后，nose 与 setuptools 实现了平滑集成，这样 test 命令就可以与之一起使用（python setup.py test）。这可以通过在 setup.py 中添加 test_suite 元数据来完成，示例如下。

```
setup(
    ...
    test_suite = 'nose.collector'
    ...)
```

nose 还为开发人员编写 nose 插件使用了 setuptools 入口点机制，这使他们可以覆盖或者修改从测试发现到测试输出的工具的所有特征。

在 http://nose-plugins.jottit.com 中可以看到一个插件列表。

（5）总结

nose 是一个完整的测试工具，修复了 unittest 存在的许多问题。不过它仍然使用了隐式的前缀名称，这为许多开发人员留下一个约束。但这个前缀是可以定制的，它仍然要求遵循一个约定。

这种在配置语句之上的约定挺不错的，比 unittest 所要求的样板代码要好得多。但是使用显式的装饰器，也可能是摆脱 test 前缀的一个好方法。

最后，插件方法使 nose 变得非常灵活，并允许开发人员定制该工具以符合他的要求。

可以在主目录添加一个.noserc 或 nose.cfg 文件，以指定 nosetests 启动时的默认选项。

一种好的方法是自动查找 doctests。

该文件的例子如下所示。

```
[nosetests]
with-doctest=1
doctest-extension=.txt
```

2．py.test

py.test 和 nose 非常相似，本小节将只介绍它的特点。这个工具被捆绑到包含其他工具的 py 包中。

nose 的灵感来自 py.test。

它也可以使用 easy_install 来安装，如下所示。

```
$ easy_install py
Searching for py
Best match: py 0.9.0
Finished processing dependencies for py
```

现在，命令提示符下将有一个新的命令 py.test，它可以和 nosetests 一样使用。这个工具和 nose 一样，使用一个模式匹配算法来捕捉需要运行的测试。该模式比 nose 使用得更严格，只捕捉：

- 在以 test 开头的文件中的以 Test 开头的类；
- 在以 test 开头的文件中的以 test 开头的函数。

请注意正确使用大小写：如果函数以大写的“T”开头，那么它将被当作一个类，从而被忽略；如果一个类以小写“t”开头，那么 py.test 将中断，因为它将尝试把其作为函数处理。

测试装置功能与 nose 类似，除了语义上有些不同之外。py.test 将从官方文档中查找每个测试模块中 3 个级别的测试装置。

```
def setup_module(module):
    """ setup up any state specific to the execution
        of the given module.
    """
def teardown_module(module):
    """ teardown any state that was previously setup
        with a setup_module method.
    """
def setup_class(cls):
    """ setup up any state specific to the execution
        of the given class (which usually contains tests).
    """
def teardown_class(cls):
    """ teardown any state that was previously setup
        with a call to setup_class.
    """
def setup_method(self, method):
    """ setup up any state tied to the execution of the given
        method in a class. setup_method is invoked for every
        test method of a class.
    """
def teardown_method(self, method):
    """ teardown any state that was previously setup
        with a setup_method call.
    """
```

每个函数将以当前模块、类或方法为参数。因此，测试装置将能够在上下文中工作，而不必查找它，这点和 nose 一样。但是 py.test 不像 nose 那样，提供了通过在包级别上添加 setup 和 teardown 函数来实现全局测试装置的方法。

py.test 独创的功能有：

- 禁用一些测试类的能力；
- 在多台电脑中分发测试的能力；
- 在该工具继续发现任务的同时立即开始测试。

（1）禁用测试类

该工具提供了一个简单的机制，可以在某些条件下禁用一部分测试。如果在测试类上找到 disabled 特性，那么它就将被选中。

例如，正如其文档中所说的那样，当一个测试是依赖于具体平台时，那么这个布尔值将被设置（来自官方文档的例子）如下。

```
class TestEgSomePosixStuff:
    disabled = sys.platform == 'win32'
    def test_xxx(self):
        ...
```

可以使用一个可调用对象来提供复杂的条件，如下所示。

```
def _disabled():
    # 在此为复杂的工作
    return 0
class Test_2:
    disabled = _disabled()
    def test_one(self):
        pass
```

不幸的是，这个特性不能是方法或属性，因为 py.test 只在类上调用 getattr(cls, 'disabled',0)。

（2）自动分发的测试

py.test 还有一个有趣的功能，它可以在多台电脑上分发测试。只要可以通过 SSH 访问的机器，py.test 都能通过发送将要执行的测试来驱动每台电脑。

但是，这个功能取决于网络。如果网络连接被破坏，那么从机将无法继续工作，因为它完全是由主机驱动的。

当一个项目具有很长的测试活动时，Buildbot 方法是首选。但是，当遇到一个消费许多资源来运行测试的应用程序时，py.test 分布模型可以用来实现测试的 ad hoc 分布。

（3）立即开始测试

py.test 在发现过程中可以采用一个迭代程序，也就是找到第一个测试时就可以直接启动。这加速了第一个测试输出，这在测试装置很慢的情况下是很好的。而且，第一个故障也将更快地发生。

（4）总结

py.test 和 nose 非常相似，因为不需要样板代码来集合测试。测试装置也可以分层配置，但是没有提供在包级别上直接启动一个配置的方法。不幸的是，它也没有提供和 nose 一样的插件系统。

最后，py.test 关注于更快地建立测试，在这个领域上，它也确实比其他工具更优越。

这个工具是一个更大框架的一部分，可以被独立分发以避免安装其他元素。

除非希望使用 py.test 的特殊功能，否则 nose 应该是首选。

11.2.3　仿真和模拟

在编写单元测试时，我们假设隔离了被测试的模块。测试向函数或方法提供一些数据并测试其输出。

这主要确保测试：

- 与应用程序的一个原子部件（可能是一个函数或类）有关；
- 提供确定的、可重复的结果。

有时候，程序中一个部件的隔离并不明显。例如，如果代码要发送邮件，那么它将调用 Python 中的 smtplib 模块，通过一个 telnet 连接与 SMTP 服务器协同工作。这不应该在测试运行时发生，理想的情况是，单元测试应该在任何没有外部依赖和副作用的电脑上运行。

由于 Python 的动态特性，使用 monkey patches 来修改测试装置的运行时代码是可能的，这可以仿真一个第三方代码库的行为。

1．建立一个仿真品

测试中的仿真行为可以通过发现测试代码与外部协作所需的最小交互集来创建。然后，手工返回输出，或者使用已经记录的真实数据。

这可以从一个空的类或函数开始，并将它用作一个替代品。然后启动测试，不断填充仿真品直到它正确表现。

以发送邮件的 mailer 模块中的一个函数 send 为例，如下所示。

```
import smtplib
import email.Message
def send(sender, to, subject='None', body='None',
         server='localhost'):
```

```
    """sends a message."""
    message = email.Message.Message()
    message['To'] = to
    message['From'] = sender
    message['Subject'] = subject
    message.set_payload(body)
    server = smtplib.SMTP(server)
    try:
        res = server.sendmail(sender, to, message.as_string())
    finally:
        server.quit()
    return res
```

在本小节中将用 nose 来示范仿真和模拟。

对应的测试可能如下所示。

```
from mailer import send
from nose.tools import *
def test_send():
    res = send('tarek@ziade.org', 'tarek@ziade.org',
               'topic', 'body')
    assert_equals(res, {})
```

只要在本地主机上有一个 SMTP 服务器，这个测试将通过并且有效。否则，它将失败，示例如下。

```
$ nosetests -v
test_mailer.test_send ... ERROR
======================================================================
ERROR: test_mailer.test_send
----------------------------------------------------------------------
Traceback (most recent call last):
...
"...Versions/2.5/lib/python2.5/smtplib.py", line 310, in connect
    raise socket.error, msg
error: (61, 'Connection refused')
```

```
----------------------------------------------------------------------
Ran 1 test in 0.169s
```

可以添加一个补丁来仿真 SMTP 类，如下所示。

```
from mailer import send
from nose.tools import *
import smtplib
def patch_smtp():
    class FakeSMTP(object):
        pass
    smtplib._SMTP = smtplib.SMTP
    smtplib.SMTP = FakeSMTP
def unpatch_smtp():
    smtplib.SMTP = smtplib._SMTP
    delattr(smtplib, '_SMTP')
@with_setup(patch_smtp, unpatch_smtp)
def test_send():
    res = send('tarek@ziade.org', 'tarek@ziade.org',
               'topic', 'body')
    assert_equals(res, {})
```

然后，再次运行测试，如下所示。

```
$ nosetests -v
test_mailer.test_send ... ERROR
======================================================================
ERROR: test_mailer.test_send
----------------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: default __new__ takes no parameters
----------------------------------------------------------------------
Ran 1 test in 0.066s
FAILED (errors=1)
```

接着，完成 FakeSMTP 类直到测试通过。它将报告类所应该拥有的几个方法，如下所示。

```
class FakeSMTP(object):
    def __init__(self, *args, **kw):
        # 我们不关心
```

```
        pass
    def quit(self):
        pass
    def sendmail(self, *args, **kw):
        return {}
```

当然，仿真的类可以和新的测试一起演化，以提供更加复杂的行为，但是它应该尽可能保持简短。相同的原则也适用于更复杂的输出，记录它们并通过仿真的 API 供应。这常常使用诸如 LDAP 或 SQL 这样的第三方服务器来完成。

仿真有实际的局限性。如果决定仿真一个外部相关对象，可能会引入缺陷或者出现真正的服务器不会出现的不必要行为，反之亦然。

2．使用模拟

模拟对象是可以用于隔离测试过的代码的通用仿真对象（参见 http://en.wikipedia.org/wiki/Mock_object），它们能使输入输出的构建自动完成。在静态类型语言中，模拟对象有更大的用处，在 Python 这种动态语言中 monkey patch 比较困难。但是它们在 Python 中仍然有用，可以缩短模拟外部 API 的代码。

在 Python 中有许多可用的模拟库，最简单和最易用的是 Ian Bicking 的 minimock（参见 http://blog.ianbicking.org/minimock.html）。

它也很容易安装，如下所示。

```
$ easy_install minimock
Searching for minimock
Reading http://pypi.python.org/simple/minimock
...
Finished processing dependencies for minimock
```

这个库提供 3 个元素：

- mock　生成模拟对象并将其放入指定命名空间的函数；
- Mock　可以用来手工实例化一个模拟对象的模拟类；
- restore　删除 mock 所打补丁的函数。

在我们的示例中，使用 minimock 来修补 SMTP，要比手工仿真更为简单，如下所示。

```
from mailer import send
from nose.tools import *
import smtplib
from minimock import mock, restore, Mock
def patch_smtp():
```

```
    mock('smtplib.SMTP',
         returns=Mock('smtp_connection',
                      sendmail=Mock('sendmail',
                          returns={})
                      )
         )
def unpatch_smtp():
    restore()
@with_setup(patch_smtp, unpatch_smtp)
def test_send():
    res = send('tarek@ziade.org',
               'tarek@ziade.org', 'topic', 'body')
    assert_equals(res, {})
```

returns 特性允许定义调用返回的元素。当使用模拟对象时，每当一个特性被代码调用，它将立刻为该特性创建一个新的模拟对象，所以不会抛出异常。这是先前编写的 quit 方法的情况。

如果一个方法必须返回一个特定值，那么模拟对象可以为那个方法手工实例化，在其 returns 参数中返回该值。这个特殊的模拟对象可以作为关键字参数传递。在 sendmail 中就做到了这一点。

现在，再次运行测试，如下所示。

```
$ nosetests -v -s
test_mailer.test_send ... Called smtplib.SMTP('localhost')
Called sendmail(
    'tarek@ziade.org',
    'tarek@ziade.org',
    'To: tarek@ziade.org\nFrom: tarek@ziade.org\nSubject: topic\n\nbody')
Called smtp_connection.quit()
ok
----------------------------------------------------------------
Ran 1 test in 0.122s
OK
```

注意，被调用的元素由 minimock 打印输出。这使之成为 doctest 的一个良好候选者。

11.2.4　文档驱动开发

doctest 是 Python 与其他语言相比的真正优势所在。文本使用的代码实例也可以作为测试运行这一事实改变了 TDD 的实施方式。例如，文档的一部分可以在开发周期中通过 doctests

来完成。这个方法也确保了所提供的示例是最新的并且确实可以工作。

通过 doctest 而不是常规的单元测试来构建软件，被称为文档驱动开发（DDD）。开发人员以普通的英语来说明代码的作用，同时实现它。

编写一个故事

在 DDD 中，编写 doctest 是由构造一个关于代码块如何工作和如何使用的故事来完成的。原则用普通的英语描述，然后在文本中放置几个代码使用场景。一个好的方法是从编写代码如何工作的文本开始，然后添加一些代码实例。这是第 6 章中编写模块的方式。

回过头看看 atomisator.parser 包的 doctest，第一个版本的文本如下所示。

```
================
atomisator.parser
================
The parser knows how to return a feed content with
a function available as a top-level function.
This function takes the feed url and returns an iterator
on its content. A second parameter can specify how
many entries have to be returned before the iterator is
exhausted. If not given, it is fixed to 10
```

接着完成该实例，由此构建的代码如下。

```
================
atomisator.parser
================
The parser knows how to return a feed content, with
the 'parse' function, available as a top-level function::
   >>> from atomisator.parser import parse
This function takes the feed url and returns an iterator
on its content. A second parameter can specify how
many entries have to be returned before the iterator is
exhausted. If not given, it is fixed to 10::

   >>> res = parse('http://example.com/feed.xml')
   >>> res
   <generator ...>
```

稍后，doctest 将可能继续演化，以考虑新的元素或必需的更改。这个 doctest 对于希望使

用该包的开发人员也是一个良好的文档，应该记得用这个方法来修改。

在文档中编写测试的常见缺点是使其变成一个无法理解的文本块。如果发生这种情况，它就不应该被作为文档的一部分。

也就是说，一些专门通过 doctest 进行工作的开发人员常常将他们的 doctest 分成两类：一种是易于理解和使用的，它们可以作为包文档的一部分；另一种是不可理解的，只能用于构建和测试软件。

许多开发人员认为后一种情况中 doctest 应该被放弃，而赞同使用常规的单元测试。另一些人甚至将 doctest 专用于缺陷修复。

所以，在 doctest 和常规测试之间的平衡是品味问题，取决于团队，只要发布的 doctest 是易读的就行。

在一个项目中使用 DDD 时，应注意可读性并决定哪些 doctest 符合成为发行文档一部分的判断条件。

11.3 小　结

本章提倡使用 TDD，并提供了更多关于以下方面的信息：

- unittest 的缺点；
- 第三方工具——nose 和 py.test；
- 如何构建仿真和模拟；
- 文档驱动开发。

下一章将关注于优化程序的方法。