pytest 测试实战

[美] Brian Okken 著

陆阳 饶勇 译

赏味不足 审校

华中科技大学出版社

中国 • 武汉

前言
Preface

Python 语言越来越流行，不单在软件开发领域，它还深入数据分析、科学研究、测量等领域。随着 Python 在这些重要领域的发展，高效可靠地进行软件测试变得尤为重要。此外，越来越多的软件开发项目开始使用持续集成，而自动化测试正是其中的关键一环。由于软件迭代周期不断缩短，所以手工测试已经行不通。项目团队必须确定测试结果是可靠的，才能放心地发布产品。

于是 pytest 派上用场。

pytest 是什么？
What Is pytest?

pytest 是一款强大的 Python 测试工具，可以胜任各种类型或级别的软件测试工作，既适合开发团队、QA 团队、独立的测试小组使用，又适合练习测试驱动开发的个人，以及开源团队使用。实际上，越来越多的互联网项目开始放弃 unittest 和 nose，转而使用 pytest，比如 Mozilla 和 Dropbox。因为 pytest 会提供更丰富的功能，包括 assert 重写、第三方插件，以及其他测试工具无法比拟的 fixture 模型。

pytest 是一个软件测试框架。它是一款命令行工具，可以自动找到测试用

例执行，并且汇报测试结果。它有丰富的基础库，可以大幅提高用户编写测试用例的效率。它具备可扩展性，用户可以自己编写插件，或者安装第三方提供的插件。pytest 可以直接测试各类 Python 程序，也可以很容易地与其他工具集成到一起使用，比如持续集成、Web 端自动化测试等。

下面列举了一些 pytest 优于其他测试框架的地方。

- 简单的测试可以很简单地编写；
- 复杂的测试也可以很简单地编写；
- 测试的可读性强；
- 易于上手；
- 断言测试失败仅使用原生 `assert` 关键字，而不是 `self.assertEqual()`，或者 `self.assertLessThan()`；
- pytest 可以运行由 unittest 和 nose 编写的测试用例。

pytest 项目是由一个正在快速壮大的社区开发和维护的。它灵活、扩展性好，可以很容易地融入已有的开发测试流程。它不依赖于 Python 版本，Python 2（2.6 及更高版本）和 Python 3（3.3 及更高版本）都可以安装最新版本的 pytest。

通过实用示例学习
Learn pytest While Testing an Example Application

没有人愿意学习实际工作中永远不会出现的例子，我也一样。本书将围绕一个实用的测试项目展开学习，其中的测试技巧很容易运用到你的实际工作中。

Tasks 项目

The Tasks Project

我们的测试对象是一个叫 Tasks 的应用程序。Tasks 是一个小巧的任务进度追踪程序，它有一个命令行交互界面，设计结构与很多应用程序相似。我希望读者在编写针对 Tasks 项目测试用例的同时，能够轻松地掌握相关技巧，并运用到自己的测试项目中去。

虽然 Tasks 程序通过 CLI（command-line interface）交互，但其底层编码是通过调用 API 实现的，因此我们的测试工作主要面向的是 API。API 与数据库控制层交互，数据库控制层与文档数据库（MongoDB 或 TinyDB）交互。数据库的类型是在数据库初始化时配置好的。

在进一步讨论 API 之前，我们先了解一下 Tasks 所使用的命令行交互工具。以下是一段示例代码：

```
$ tasks add 'do something' --owner Brian
$ tasks add 'do something else'
$ tasks list
  ID    owner  done  summary
  --    -----  ----  -------
   1    Brian  False  do something
   2           False  do something else
$ tasks update 2 --owner Brian
$ tasks list
  ID    owner  done  summary
  --    -----  ----  -------
   1    Brian  False  do something
   2    Brian  False  do something else
$ tasks update 1 --done True
$ tasks list
  ID    owner  done  summary
  --    -----  ----  -------
   1    Brian   True  do something
   2    Brian  False  do something else
$ tasks delete 1
$ tasks list
  ID    owner  done   summary
  --    -----  ----   -------
   2    Brian   True  do something
$
```

这个任务管理程序并不复杂，但作为我们的示例已经够用。

测试方法
Test Strategy

虽然 pytest 可以用于单元测试、集成测试、系统测试（端到端测试）、功能测试，但 Tasks 项目将主要采用皮下测试（subcutaneous test）。以下是一些有关测试方法的基本定义。

单元测试 检查一小块代码（如一个函数，或一个类）的测试。第 1 章中的测试就是针对 Tasks 数据结构的单元测试。

集成测试 检查大段的代码（如多个类，或一个子系统）的测试。集成测试的规模介于单元测试与系统测试之间。

系统测试 检查整个系统的测试，通常要求测试环境尽可能接近最终用户的使用环境。

功能测试 检查单个系统功能的测试。就 Tasks 项目而言，针对增加、删除、更新一个任务条目的测试就属于功能测试。

皮下测试 不针对最终用户界面，而是针对用户界面以下的接口的测试。本书中的大部分测试都是针对 API 的，而不是针对 CLI 的，因此这些测试都属于皮下测试。

本书结构
How This Book Is Organized

第 1 章介绍 pytest 的安装，同时会介绍 Tasks 项目的数据结构部分（名为 `Task` 的 `namedtuple`），并用它作为测试示例。我们会学习如何指定测试文件运行，以及 pytest 常用的命令行命令，包括重新运行失败测试、遇到失败即停止

所有测试、控制堆栈跟踪、控制日志输出，等等。

第 2 章将使用 `pip` 在本地安装 Tasks 项目，学习在 Python 项目中如何组织测试目录，这样才能针对实际项目编写测试用例。这一章的所有示例都依赖外部程序，包括数据库写入。第 2 章的重点是测试函数，你将学习在 pytest 中高效使用断言语句。这一章还会讲解 marker 标记功能的用法，marker 标记可以将测试进行归类或分组，方便一起运行，也可以将某些测试标记为 skip（跳过不执行），marker 标记还可以告诉 pytest 我们知道某些测试是一定会失败的。如果希望运行指定的测试子集，除了使用 `marker`，还可以将测试代码组织成测试目录、测试模块、测试类，然后运行。

并非所有的测试代码都要放到测试函数中。第 3 章介绍如何将测试数据、启动逻辑、销毁逻辑放入 fixture（pytest 定义的一种测试脚手架）。设置系统（或子系统、系统单元）是软件测试的重要环节，第 3 章将介绍用一个简单的 fixture 完成这方面的工作（包括对数据库进行初始化，写入数据以备测试之用）。Fixture 模块的功能非常强大，你可以利用它简化测试代码，从而提高代码的可读性和可维护性。Fixture 像测试函数一样，也有参数。利用参数，你只需要编写一份代码，就可以针对 TinyDB 和 MongoDB（或其他 Tasks 项目支持的数据库）开展测试。

第 4 章介绍 pytest 内置的 fixture 以满足测试中常见的一些需求，包括生成和销毁临时目录、截取输出流（通过日志判定结果）、使用 monkey patch、检查是否发出警告，等等。

第 5 章讲解如何在 pytest 中添加命令行选项，如何改进打印输出，如何打包分发自己编写的插件，如何共享定制化的 pytest（包括 fixture）。这一章开发的插件可以改善 Tasks 项目测试失败时的输出呈现方式。你还将学习测试自己的测试插件(元测试)。读完这一章，想必你已经等不及编写自己的插件了。附录 C 收集了一些热门的社区插件，可供参考。

第 6 章讲解通过 pytest.ini 文件修改默认配置，自定义 pytest 的运行方式。pytest.ini 文件可以存放某些命令选项，从而减少你重复输入命令的次数；利用它还可以指定 pytest 忽略某些测试目录，或者指定 pytest 的最低版本，等等。使用 tox.ini 和 setup.cfg 文件也可以实现同样的功能。

第 7 章（最后一章）介绍 pytest 与其他工具的结合使用。我们将借助 tox 让 Tasks 项目在多个 Python 版本上运行；学习如何测试 Tasks 项目的 CLI 部分，而不必 mock 系统的其余部分；借助 coverage.py 检查 Tasks 项目代码块的测试覆盖情况；通过 Jenkins 发起测试并实时显示结果。最后，还会学习如何让 pytest 运行基于 unittest 的测试用例，以及把 pytest 的 fixture 共享给 unittest 的测试用例使用。

阅读基础
What You Need to Know

Python

你不必精通 Python，本书的示例语法很常见。

pip

本书使用 pip 安装 pytest 及其组件。获取新版本的 pip，请查阅附录 B。

命令行

我自己使用的是 Mac 笔记本和 bash，实际上，我只使用了 cd（进入指定目录）和 pytest 两条命令。Windows 系统和所有 UNIX 派生系统都有 cd 命令，所以本书的示例也应该适用于这些操作系统。

满足以上条件，就可以阅读本书了。不擅长编程的人一样可以学习使用 pytest 进行自动化测试。

示例代码与共享资源
Example Code and Online Resources

本书示例代码是使用 Python 3.6 和 pytest 3.2 编写的。pytest 3.2 支持 Python 2.6、Python 2.7、Python 3.3 及以后的版本。

Tasks 项目的源代码及测试用例可以从 pragprog.com 网站上本书的页面下载[1]。如果只是为了理解书中的测试，不必下载源代码，这些代码都通俗易懂。但如果你希望深入地理解 Tasks 项目，以便将来更好地运用学到的技巧，那就有必要下载源代码。我们在本书的页面上还提供了最新的勘误信息[2]。

我编程已有二十余年，从未遇到过像 pytest 这样优秀的测试工具。希望各位阅读本书后能够有所收获，也喜欢上 pytest。

[1] https://pragprog.com/titles/bopytest/source_code
[2] https://pragprog.com/titles/bopytest/errata

目录

Contents

第 1 章

pytest 入门
Getting Started with pytest

这是一个测试用例：

```
ch1/test_one.py
def test_passing():
    assert (1, 2, 3) == (1, 2, 3)
```

打开终端执行以下代码：

```
$ cd /path/to/code/ch1
$ pytest test_one.py
================test session starts==================
collected 1 items
test_one.py .
================1 passed in 0.01 seconds =============
```

test_one.py 后方的一个点号（.）表示：运行了一个测试用例，且测试通过。如果想查看详情，可以在 pytest 后面加上-v 或者--verbose 选项。

```
$ pytest -v test_one.py
===============test session starts ==================
collected 1 items
test_one.py::test_passing PASSED
================1 passed in 0.01 seconds =============
```

如果你使用的是彩色终端，那么 PASSED 和底部线条是绿色的。以下是一

个注定要失败的测试用例：

```
ch1/test_two.py
def test_failing():
    assert (1, 2, 3) == (3, 2, 1)
```

pytest 展示的失败信息非常清楚，这是它受欢迎的原因之一。

```
$ pytest test_two.py
==============test session starts =================
collected 1 items
test_two.py F
================ FAILURES ===================
_____________test_failing _____________
def test_failing():
>   assert (1, 2, 3) == (3, 2, 1)
E    assert (1, 2, 3) == (3, 2, 1)
E      At index 0 diff: 1 != 3
E      Use -v to get the full diff
test_two.py:2: AssertionError
============ 1 failed in 0.04 seconds ==============
```

pytest 有一块专门的区域展示 `test_failing` 的失败信息，它能准确地指出失败原因：index 0 is mismatch。重要的提示信息用红色字体显示（在彩色显示器上），以方便用户阅读。其中有一条提示指出，使用`-v` 可以得到更完整的前后对比信息，下面来试一试。

```
$ pytest -v test_two.py
===================== test session starts ======================
collected 1 items
test_two.py::test_failing FAILED
============================ FAILURES ============================
_________________________ test_failing _________________________
   def test_failing():
>      assert (1, 2, 3) == (3, 2, 1)
E      assert (1, 2, 3) == (3, 2, 1)
E        At index 0 diff: 1 != 3
E        Full diff:
E        - (1, 2, 3)
E        ?  ^     ^
E        + (3, 2, 1)
E        ?  ^     ^
test_two.py:2: AssertionError
=================== 1 failed in 0.04 seconds ===================
```

pytest 添加了几个脱字符（^），准确地指出了前后的区别。pytest 除了易读、易写、易运行、失败提示信息清晰，还有许多优点，如果你有耐心，请听我慢慢告诉你。在我心目中，pytest 是最优秀的测试框架。

本章讲解 pytest 的安装、pytest 的各种运行方式，以及最常用的命令行选项。后面几章的学习内容包括：如何最大限度地发挥 pytest 的优势来编写测试函数；如何编写 fixture（用于存放相同的启动逻辑和销毁逻辑）；如何使用 fixture 和 pytest 插件简化测试工作。

抱歉，我在这里使用了 `assert (1, 2, 3) == (3, 2, 1)`这样无聊的例子，实际软件测试中显然不会出现这样的例子。接下来，我会以一个名叫 Tasks 的软件项目为示例来编写测试。这个项目并不复杂，很容易理解，同时也比较接近真实的软件项目。

软件测试的一个目标是验证你的猜想——猜想软件的内部逻辑，包括第三方的模块、代码包，甚至 Python 内建的数据结构是如何运作的。Tasks 项目使用名为 Task 的数据结构，它是用 `namedtuple` 工厂函数生成的。`namedtuple` 是 Python 标准库的一部分。Task 用于在 UI 层和 API 层之间传递信息，我用它来演示 pytest 的运行，以及一些常用命令行选项的用法。

以下是一个 Task 结构：

```
from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
```

`namedtuple()`工厂方法在 Python 2.6 中就已经出现，但我发现许多 Python 开发者并不了解它。我觉得使用 Task 结构比使用`(1, 2, 3) == (1, 2, 3)`或者 `add(1, 2) == 3` 有趣得多。

在进一步讲解之前，我们先看看 pytest 的下载和安装方法。

1.1 资源获取
Getting pytest

pytest 的官方文档地址：https://docs.pytest.org。pytest 通过 PyPI（Python 官方包管理索引）分发托管：https://pypi.python.org/pypi/pytest。

就像其他在 PyPI 中托管的 Python 程序包一样，你可以在当前的虚拟环境中使用 pip 安装 pytest：

```
$ pip3 install -U virtualenv
$ python3 -m virtualenv venv
$ source venv/bin/activate
$ pip install pytest
```

如果你不太熟悉 vritualenv 或者 pip，请查阅附录 A。

Windows、Python 2、venv 的问题

virtualenv 和 pip 的例子应该可以在 Linux、macOS 等 POSIX（可移植操作系统标准接口）系统下正常运行，也支持多个 Python 版本，包括 2.7.9 以后的所有版本。

但是在 Windows 下，source/venv/bin/activate 这条命令无法使用，请使用 venv\Scripts\activate.bat：

```
C:\> pip3 install -U virtualenv
C:\> python3 -m virtualenv venv
C:\>venv\Scripts\activate.bat (venv)
C:\> pip install pytest
```

Python 3.6 以后的版本最好使用 venv 替代 virtualenv。Python 3.6 自带 venv，不必手动安装。但我听说在某些平台下，virtualenv 的表现更好。

1.2 运行 Pytest
Running pytest

```
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
    ...
```

如果你不提供任何参数，pytest 会在当前目录以及子目录下寻找测试文件，然后运行搜索到的测试代码。如果你提供一个或多个文件名、目录名，pytest 会逐个查找并运行所有测试。为了搜索到所有的测试代码，pytest 会递归遍历每个目录及其子目录。

举一个例子，我们新建一个 tasks 子目录，并且创建以下测试文件：

```
ch1/tasks/test_three.py
"""Test the Task data type."""
from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)

def test_defaults():
    """Using no parameters should invoke defaults."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2

def test_member_access():
    """Check .field functionality of namedtuple."""
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

你可以使用__new__.__defaults__创建默认的Task对象，不必指定所有属性。测试用例 test_defaults()中演示了默认值的校验。

测试用例 test_member_access()演示了如何利用属性名（而不是索引）来访问对象成员，这也是选用 namedtuple 的一个原因。

下面两个例子演示了_asdict()函数和_replace()函数的功能：

```
cha1/tasks/test_four.py
"""Test the Task data type."""

from collections import namedtuple

Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)

def test_asdict():
    """_asdict() should return a dictionary."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
                'owner': 'okken',
                'done': True,
                'id': 21}
    assert t_dict == expected

def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected
```

运行 pytest 时可以指定目录和文件。如果不指定，pytest 会搜索当前目录及其子目录中以test_开头或以_test结尾的测试函数。假设切换到ch1目录运行pytest（没有指定任何参数），那么pytest会运行ch1目录下的四个测试文件。

```
$ cd /path/to/code/ch1
$ pytest
====================== test session starts ======================
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..
============================ FAILURES ===========================
_____________________ test_failing ______________________
   def test_failing():
>      assert (1, 2, 3) == (3, 2, 1)
E      assert (1, 2, 3) == (3, 2, 1)
E        At index 0 diff: 1 != 3
E        Use -v to get the full diff
test_two.py:2: AssertionError
============== 1 failed, 5 passed in 0.08 seconds ==============
```

如果我们编写了新的测试，那么可以在 pytest 中指定需要测试的文件名或目录，或者预先切换到需要运行的目录：

```
$ pytest tasks/test_three.py tasks/test_four.py
====================== test session starts ======================
collected 4 items

tasks/test_three.py ..
tasks/test_four.py ..
=================== 4 passed in 0.02 seconds ====================
$ pytest tasks
====================== test session starts ======================
collected 4 items

tasks/test_four.py ..
tasks/test_three.py ..
=================== 4 passed in 0.03 seconds ====================
$ cd /path/to/code/ch1/tasks
$ pytest
====================== test session starts ======================
collected 4 items

test_four.py ..
test_three.py ..
=================== 4 passed in 0.02 seconds ====================
```

我们把 pytest 搜索测试文件和测试用例的过程称为测试搜索（test discovery）。只要你遵守 pytest 的命名规则，pytest 就能自动搜索所有待执行的测试用例。以下是几条主要的命名规则。

- 测试文件应当命名为 `test_<something>.py` 或者 `<something>_test.py`。
- 测试函数、测试类方法应当命名为 `test_<something>`。
- 测试类应当命名为 `Test<Something>`。

测试文件和测试函数最好以 `test_`开头，但如果先前编写的测试用例遵循的是其他命名规则，也可以修改默认的测试搜索规则，第 6 章会介绍这方面的技巧。

下面我们来逐句讲解 pytest 运行单个测试文件时的控制台输出信息：

```
$ cd /path/to/code/ch1/tasks
$ pytest test_three.py
================= test session starts ==================
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /path/to/code/ch1/tasks, inifile:
collected 2 items

test_three.py ..
================ 2 passed in 0.01 seconds ===============
```

从中我们可以获得很多信息：

===== test session starts ====

pytest 为每段测试会话（session）做了明确的分隔，一段会话就是 pytest 的一次调用，它可能包括多个目录下被执行的测试用例。后面还会介绍到，如果 pytest fixture 的作用范围是会话级别，那么会话的定义就显得尤为重要（参见第 3.6 节）。

platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0

我使用的是 Mac，所以显示的是 `platform darwin`，Windows 的表示方式略有不同。接着显示的是 Python、pytest 以及 pytest 包的版本。py 和 pluggy 都是 pytest 包，用于 pytest 的实现，它们均由 pytest 团队开发维护。

rootdir: /path/to/code/ch1/tasks, inifile:

`rootdir`（当前起始目录）是 pytest 搜索测试代码时最常使用的目录，`inifile` 用于列举配置文件（这里没有指定），文件名可能是 `pytest.ini`、`tox.ini` 或者 `setup.cfg`。关于配置文件的知识，请参考第 6 章。

collected 2 items

搜索范围内找到两个测试条目。

test_three.py ..

three.py 表示测试文件，每个文件的测试情况只占据一行，两个点号表示两个测试用例均已通过。点号仅仅表示通过，而 Failure（失败）、error（异常）、skip（跳过）、xfail（预期失败）、xpass（预期失败但通过）会被分别标记为 F、E、s、x、X。使用-v 或--verbose 可以看到更多细节。

== 2 passed in 0.01 seconds ==

表示测试通过的数量以及这段会话耗费的时间，如果存在未通过的测试用例，则会根据未通过的类型列举数量。

测试结果是测试人员了解测试始末的主要途径。在 pytest 中，测试函数可能返回多种结果，不只是通过或失败。

以下是可能出现的类型。

- PASSED(.)：测试通过。
- FAILED(F)：测试失败（也有可能是 XPASS 状态与 strict 选项冲突造成的失败，见后文）。
- SKIPPED(s)：测试未被执行。指定测试跳过执行，可以将测试标记为@pytest.mark.skip()，或者使用@pytest.mark.skipif()指定跳过测试的条件，请参考第 2.5 节。
- xfail(x)：预期测试失败，并且确实失败。使用@pytest.mark.xfail()指定你认为会失败的测试用例，请参考第 2.6 节。
- XPASS(X)：预期测试失败，但实际上运行通过，不符合预期。
- ERROR(E)：测试用例之外的代码触发了异常，可能由 fixture 引起，也可能由 hook 函数引起。

1.3 运行单个测试用例
Running Only One Test

学习测试最好从单个测试用例开始。你可以直接在指定文件后方添加::test_name，像下面这样：

```
$ cd /path/to/code/ch1
$ pytest -v tasks/test_four.py::test_asdict
=================== test session starts ===================
collected 3 items

tasks/test_four.py::test_asdict PASSED
================ 1 passed in 0.01 seconds =================
```

接下来看命令行选项。

1.4 使用命令行选项
Using Options

我们已经使用过-v 和--verbose 选项。pytest 还提供很多选项，本书只会用到一部分。你可以使用 pytest--help 查看全部选项。

下面列出常用的 pytest 命令选项，对 pytest 初学者而言，这些选项已经够用了。

```
$ pytest --help
  ... subset of the list ...
  -k EXPRESSION         only run tests/classes which match the given
                        substring expression.
                        Example: -k 'test_method or test_other' matches
                        all test functions and classes whose name
                        contains 'test_method' or 'test_other'.
  -m MARKEXPR           only run tests matching given mark expression.
                        example: -m 'mark1 and not mark2'.
  -x, --exitfirst       exit instantly on first error or failed test.
  --maxfail=num         exit after first num failures or errors.
  --capture=method      per-test capturing method: one of fd|sys|no.
  -s                    shortcut for --capture=no.
  --lf, --last-failed   rerun only the tests that failed last time
                        (or all if none failed)
```

```
  --ff, --failed-first  run all tests but run the last failures first.
  -v, --verbose        increase verbosity.
  -q, --quiet          decrease verbosity.
  -l, --showlocals     show locals in tracebacks (disabled by default).
  --tb=style           traceback print mode (auto/long/short/line/native/no).
  --durations=N        show N slowest setup/test durations (N=0 for all).
  --collect-only       only collect tests, don't execute them.
  --version            display pytest lib version and import information.
  -h, --help           show help message and configuration info
```

--collect-only 选项

--collect-only

使用--collect-only 选项可以展示在给定的配置下哪些测试用例会被运行。之所以首先介绍它，是因为该选项的输出内容可以作为一种参照。打开ch1 目录，使用此选项，你将看到本章到目前为止使用的全部测试用例。

```
$ cd /path/to/code/ch1
$ pytest --collect-only
=================== test session starts ===================
collected 6 items
 <Module 'test_one.py'>
  <Function 'test_passing'>
<Module 'test_two.py'>
  <Function 'test_failing'>
<Module 'tasks/test_four.py'>
  <Function 'test_asdict'>
  <Function 'test_replace'>
<Module 'tasks/test_three.py'>
  <Function 'test_defaults'>
  <Function 'test_member_access'>
============== no tests ran in 0.03 seconds ===============
```

--collect-only 选项可以让你非常方便地在测试运行之前，检查选中的测试用例是否符合预期。下面介绍-k 选项时，我们还会用到它。

-k 选项

-k EXPRESSION

-k 选项允许你使用表达式指定希望运行的测试用例。这个功能非常实用，如果某测试名是唯一的，或者多个测试名的前缀或后缀相同，那么可以使用表

达式来快速定位。假设希望选中 test_asdict()和 test_defaults()，那么可以使用--collect-only 验证筛选情况：

```
$ cd /path/to/code/ch1
$ pytest -k "asdict or defaults" --collect-only
=================== test session starts ===================
collected 6 items
<Module 'tasks/test_four.py'>
  <Function 'test_asdict'>
<Module 'tasks/test_three.py'>
  <Function 'test_defaults'>
=================== 4 tests deselected ====================
============== 4 deselected in 0.03 seconds ===============
```

看起来符合预期。现在把--collect-only 选项移除，让它们正常运行。

```
$ pytest -k "asdict or defaults"
=================== test session starts ===================
collected 6 items

tasks/test_four.py .
tasks/test_three.py .
=================== 4 tests deselected ====================
========= 2 passed, 4 deselected in 0.03 seconds ==========
```

很好，它们都通过了。不过运行结果也符合我们的预期吗？可以打开-v 或者--verbose 查看：

```
$ pytest -v -k "asdict or defaults"
=================== test session starts ===================
collected 6 items

tasks/test_four.py::test_asdict PASSED
tasks/test_three.py::test_defaults PASSED
=================== 4 tests deselected ====================
========= 2 passed, 4 deselected in 0.02 seconds ==========
```

很好，两个用例都通过验证了。

-m 选项

-m MARKEXPR

标记（marker）用于标记测试并分组，以便快速选中并运行。以

test_replace()和 test_member_access()为例，它们甚至都不在同一个文件里，如果你希望同时选中它们，那么可以预先做好标记。

使用什么标记名由你自己决定，假设你希望使用 run_these_please，则可以使用@pytest.mark.run_these_please 这样的装饰器（decorator）来做标记，像下面这样：

```
import pytest
...
@pytest.mark.run_these_please
def test_member_access():
...
```

给 test_replace()也做上同样的标记。有相同标记的测试（集合），可以一起运行。例如，这里我们使用 pytest –m run_these_please 命令就可以同时运行 test_replace()和 test_member_access()。

```
$ cd /path/to/code/ch1/tasks
$ pytest -v -m run_these_please
================== test session starts ==================
collected 4 items

test_four.py::test_replace PASSED
test_three.py::test_member_access PASSED
================== 2 tests deselected ==================
========= 2 passed, 2 deselected in 0.02 seconds =========
```

使用-m 选项可以用表达式指定多个标记名。使用-m "mark1 and mark2"可以同时选中带有这两个标记的所有测试用例。使用-m "mark1 and not mark2"则会选中带有 mark1 的测试用例，而过滤掉带有 mark2 的测试用例；使用-m "mark1 or mark2"则选中带有 mark1 或者 mark2 的所有测试用例。第 2.4 节还会详细介绍测试函数的标记。

-x 选项

-x, --exitfirst

正常情况下，pytest 会运行每一个搜索到的测试用例。如果某个测试函数

被断言失败，或者触发了外部异常，则该测试用例的运行就会到此为止，pytest 将其标记为失败后会继续运行下一个测试用例。通常，这就是我们期望的运行模式。但是在 debug 时，我们会希望遇到失败时立即停止整个会话，这时-x 选项就派上用场了。

让我们试着用**-x** 选项运行之前的 6 个测试用例：

```
$ cd /path/to/code/ch1
$ pytest -x
=================== test session starts ===================
collected 6 items

test_one.py .
test_two.py F
======================== FAILURES =========================
_____________________ test_failing ______________________
   def test_failing():
>      assert (1, 2, 3) == (3, 2, 1)
E      assert (1, 2, 3) == (3, 2, 1)
E        At index 0 diff: 1 != 3
E        Use -v to get the full diff
test_two.py:2: AssertionError
!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!
=========== 1 failed, 1 passed in 0.25 seconds ============
```

输出信息开头显示 pytest 收集到了 6 个测试条目，末尾显示有 1 个通过，有 1 个失败。Interrupted 提示我们测试中断了。

如果没有-x 选项，那么 6 个测试都会被执行，去掉-x 再运行一次，并且使用--tb=no 选项关闭错误信息回溯：

```
$ cd /path/to/code/ch1
$ pytest --tb=no
================== test session starts ==================
collected 6 items
test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..
=========== 1 failed, 5 passed in 0.09 seconds ============
```

可以看到，不使用-x 时，pytest 在 test_two.py 文件中遇到了测试失败，但并没有停止执行后面的测试用例。

--maxfail=num

--maxfail=num

-x 选项的特点是，一旦遇到测试失败，就会全局停止。假设你允许 pytest 失败几次后再停止，则可以使用--maxfail 选项，明确指定可以失败几次。

以目前仅仅存在一个失败测试的情况，很难展示这个特性，如果我们设置--maxfail=2，那么所有的测试都会被运行；如果设置--maxfail=1，就与-x 的作用相同。下面让我们来试试。

```
$ cd /path/to/code/ch1
$ pytest --maxfail=2 --tb=no
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..
=========== 1 failed, 5 passed in 0.08 seconds ============
$ pytest --maxfail=1 --tb=no
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F
!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!
=========== 1 failed, 1 passed in 0.19 seconds ============
```

这一次我们也使用--tb=no 关闭了错误堆栈回溯。

-s 与--capture=method

-s and --capture=method

-s 选项允许终端在测试运行时输出某些结果，包括任何符合标准的输出流信息。-s 等价于--capture=no。正常情况下，所有的测试输出都会被捕获。测试失败时，为了帮助你理解究竟发生了什么，pytest 会做出推断，并输出报告。-st 和--capture=no 选项关闭了输出捕获。我编写测试用例时，习惯添加几个 print()，以便观察某时刻测试执行到了哪个阶段。

使用-l/--showlocals 选项，在测试失败时会打印出局部变量名和它们的值，这样可以规避一些不必要的 print 语句。

信息捕获方法还有--capture=fd 和--capture=sys。使用--capture=sys 时，sys.stdout/stderr 将被输出至内存；使用--capture=fd 时，若文件描述符（file descriptor）为 1 或 2，则会被输出至临时文件中。

实际上，我自己很少使用 sys 和 fd，我用得最多的是-s。因为讲到-s，所以不得不提到--capture。

目前为止，我们的例子中还没有使用过打印语句，但是我建议读者自己尝试一下。

--lf（--last-failed）选项

-lf, --last-failed

当一个或多个测试失败时，我们常常希望能够定位到最后一个失败的测试用例重新运行，这时可以使用--lf 选项。

```
$ cd /path/to/code/ch1
$ pytest --lf
=================== test session starts ===================
run-last-failure: rerun last 1 failures
collected 6 items

test_two.py F
======================== FAILURES =========================
```

```
_____________________ test_failing ____________________
   def test_failing():
>      assert (1, 2, 3) == (3, 2, 1)
E      assert (1, 2, 3) == (3, 2, 1)
E        At index 0 diff: 1 != 3
E        Use -v to get the full diff
test_two.py:2: AssertionError
=================== 5 tests deselected ====================
========= 1 failed, 5 deselected in 0.08 seconds ==========
```

这里最好加上--tb选项，可以隐藏部分信息，或者指定其他的错误堆栈回溯模式，重新运行失败的测试用例。

--ff（--failed-first）选项

-ff, --failed-first

--ff（--failed-first）选项与--last-failed选项的作用基本相同，不同之处在于--ff会运行完剩余的测试用例。

```
$ cd /path/to/code/ch1
$ pytest --ff --tb=no
=================== test session starts ===================
run-last-failure: rerun last 1 failures first
collected 6 items

test_two.py F
test_one.py .
tasks/test_four.py ..
tasks/test_three.py ..
=========== 1 failed, 5 passed in 0.09 seconds ============
```

由于test_failing()是在test_two.py文件中，因此通常会在test_one.py之后运行，但在--ff选项作用下，test_failing()前一轮被认定为失败，会被首先执行。

-v（--verbose）选项

-v, --verbose

使用-v/--verbose选项，输出的信息会更详细。最明显的区别就是每个文件中的每个测试用例都占一行（先前是每个文件占一行），测试的名字和结果

都会显示出来，而不仅仅是一个点或字符。我们已经多次使用过-v，现在可以在--ff 中的例子添加该选项，与上文作对比：

```
$ cd /path/to/code/ch1
$ pytest -v --ff --tb=no
==================== test session starts ====================
run-last-failure: rerun last 1 failures first
collected 6 items

test_two.py::test_failing FAILED
test_one.py::test_passing PASSED
tasks/test_four.py::test_asdict PASSED
tasks/test_four.py::test_replace PASSED
tasks/test_three.py::test_defaults PASSED
tasks/test_three.py::test_member_access PASSED
=========== 1 failed, 5 passed in 0.07 seconds ============
```

在彩色显示器上可以看到 FAILED 标记为红色，PASSED 标记为绿色。

-q（--quiet）选项

-q, --quiet

该选项的作用与-v/--verbose 的相反，它会简化输出信息。我喜欢将-q 和--tb=line（仅打印异常的代码位置）搭配使用。

单独使用-q 的情况：

```
$ cd /path/to/code/ch1
$ pytest -q .F....
======================== FAILURES ========================
_____________________ test_failing ______________________
    def test_failing():
>       assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Full diff:
E         - (1, 2, 3)
E         ?  ^     ^
E         + (3, 2, 1)
E         ?  ^     ^

test_two.py:2: AssertionError
1 failed, 5 passed in 0.08 seconds
```

使用-q选项会简化输出信息，只保留最核心的内容。为了节省篇幅和突出重点，后文会频繁使用-q 和--tb=no。

-l（--showlocals）选项

-l, --showlocals

使用-l选项，失败测试用例由于被堆栈追踪，所以局部变量及其值都会显示出来。

目前为止，我们还没有包含局部变量的失败测试，因此我对test_replace()做了一些修改。

```
t_expected = Task('finish book', 'brian', True, 10)
```

改为：

```
t_expected = Task('finish book', 'brian', True, 11)
```

10 改成 11，由于 11 不是测试函数期望的值，所以会造成测试失败。这样就可以使用-l/--showlocals 选项了。

```
$ cd /path/to/code/ch1
$ pytest -l tasks
=================== test session starts ===================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..
======================== FAILURES =========================
_____________________ test_replace ______________________
    def test_replace():
        t_before = Task('finish book', 'brian', False)
        t_after = t_before._replace(id=10, done=True)
        t_expected = Task('finish book', 'brian', True, 11)
>       assert t_after == t_expected
E       AssertionError: assert Task(summary=...e=True, id=10) ==
Task(summary='...e=True, id=11)
E         At index 3 diff: 10 != 11
E         Use -v to get the full diff
t_after= Task(summary='finish book', owner='brian', done=True, id=10)
t_before= Task(summary='finish book', owner='brian', done=False,
id=None)
t_expected = Task(summary='finish book', owner='brian', done=True,
```

```
id=11)
tasks/test_four.py:20: AssertionError
=========== 1 failed, 3 passed in 0.08 seconds ============
```

assert 触发测试失败之后，代码片断下方显示的是本地变量 t_after、t_beforeh、t_expected 详细的值。

--tb=style 选项

--tb=style

--tb=style 选项决定捕捉到失败时输出信息的显示方式。某个测试用例失败后，pytest 会列举出失败信息，包括失败出现在哪一行、是什么失败、怎么失败的，此过程我们称之为“信息回溯”。大多数情况下，信息回溯是必要的，它对找到问题很有帮助，但有时也会对多余的信息感到厌烦，这时--tb=style 选项就有用武之地了。我推荐的 style 类型有 short、line、no。short 模式仅输出 assert 的一行以及系统判定内容（不显示上下文）；line 模式只使用一行输出显示所有的错误信息；no 模式则直接屏蔽全部回溯信息。

继续使用上文中对于 test_replace()的修改，利用它来看看各种失败回溯信息的显示方式。

使用--tb=no 屏蔽全部回溯信息。

```
$ cd /path/to/code/ch1
$ pytest --tb=no tasks
=================== test session starts ===================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..
=========== 1 failed, 3 passed in 0.04 seconds ============
```

使用--tb=line，它可以告诉我们错误的位置，有时这已经足够了（特别是在运行大量测试用例的情况下，可以用它发现失败的共性）。

```
$ pytest --tb=line tasks
=================== test session starts ===================
collected 4 items
```

```
tasks/test_four.py .F
tasks/test_three.py ..
======================== FAILURES ========================
/path/to/code/ch1/tasks/test_four.py:20:
AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
=========== 1 failed, 3 passed in 0.05 seconds ============
```

使用--tb=short，显示的回溯信息比前面两种模式的更详细。

```
$ pytest --tb=short tasks
=================== test session starts ===================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..
======================== FAILURES ========================
_____________________ test_replace _____________________
tasks/test_four.py:20: in test_replace
    assert t_after == t_expected
E   AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
E     At index 3 diff: 10 != 11
E     Use -v to get the full diff
=========== 1 failed, 3 passed in 0.04 seconds ============
```

这些信息足够让你了解发生了什么。

除此之外，还有三种可选的模式。

--tb=long 输出最为详尽的回溯信息；--tb=auto 是默认值，如果有多个测试用例失败，仅打印第一个和最后一个用例的回溯信息（格式与 long 模式的一致）；--tb=native 只输出 Python 标准库的回溯信息，不显示额外信息。

--duration=N 选项

--durations=N

--duration=N 选项可以加快测试节奏。它不关心测试是如何运行的，只统计测试过程中哪几个阶段是最慢的（包括每个测试用例的 call、setup、teardown 过程）。它会显示最慢的 N 个阶段，耗时越长越靠前。如果使用

`--duration=0`，则会将所有阶段按耗时从长到短排序后显示。

本书的例子耗时都比较短，为了方便演示，我特意添加了一个休眠函数 `time.sleep(0.1)`，猜猜它在哪里。

```
$ cd /path/to/code/ch1
$ pytest --durations=3 tasks
================= test session starts =================
collected 4 items

tasks/test_four.py ..
tasks/test_three.py ..
============== slowest 3 test durations ===============
0.10s call     tasks/test_four.py::test_replace
0.00s setup    tasks/test_three.py::test_defaults
0.00s teardown tasks/test_three.py::test_member_access
============== 4 passed in 0.13 seconds ===============
```

最慢的阶段出现了，它的标签是 `call`，显然是它被休眠了 0.1 秒。紧随其后的两个阶段是 `setup` 和 `teardown`。每个测试用例大体上都包含三个阶段：call、setup、teardown。其中 setup 和 teardown 也称 fixture，你可以在 fixture 中添加代码，在测试之前让系统做好准备，在测试之后做必要的清理。第 3 章会详细讲解 fixture。

--version 选项

--version

使用`--version` 可以显示当前的 pytest 版本及安装目录：

```
$ pytest --version
This is pytest version 3.0.7, imported from
  /path/to/venv/lib/python3.5/site-packages/pytest.py
```

由于是在虚拟环境中安装的 pytest，所以会定位到对应的 `site-packages` 目录。

-h（--help）选项

-h, --help

即使你已经能够熟练使用 pytest，`-h` 选项依然非常有用，它不但能展示原生 pytest 的用法，还能展示新添加的插件的选项和用法。

使用`-h`选项可以获得：

- 基本用法：`pytest [options] [file_or_dir] [file_or_dir] [...]`
- 命令行选项及其用法，包括新添加的插件的选项及其用法。
- 可用于 ini 配置文件中的选项（第 6 章会详细介绍）。
- 影响 pytest 行为的环境变量（第 6 章会详细介绍）。
- 使用 `pytest --markers` 时的可用 marker 列表（第 2 章会详细介绍）。
- 使用 `pytest --fixtures` 时的可用 fixture 列表（第 3 章会详细介绍）。

帮助信息最后会显示一句话：

```
shown according to specified file_or_dir or current dir if not specified
```

这句话的意思为：显示结果取决于指定的文件或目录，未指定的则默认使用当前目录和文件。这句话非常重要，选项、marker、fixture 都会因为目录的变化而发生变化。这是因为 pytest 有可能在某个叫 `conftest.py` 的文件中搜索到 hook 函数（新增命令行选项）、新的 fixture 定义及新的 marker 定义。

Pytest 允许你在 `conftest.py` 和测试文件中自定义测试行为，其作用域仅仅是某个项目，甚至只是某个项目的测试子集。第 6 章会详细介绍 `conftest.py` 和 ini 配置文件。

1.5 练习
Exercises

1. 使用 `python -m virtualenv` 或 `python -m venv` 创建一个新的虚拟环境，专供本书的示例使用。我以前也拒绝这么做，但目前已经离不开它了。如果遇到困难，可以参见附录 A。

2. 尝试进入和退出虚拟环境。

   - `$ source venv/bin/activate`
   - `$ deactivate`

   Windows 环境：

   - `C:\Users\okken\sandbox>venv\scripts\activate.bat`
   - `C:\Users\okken\sandbox>deactivate`

3. 在虚拟环境中安装 pytest，如果遇困难，请参考附录 B。即使你已经安装过 pytest，也请你在虚拟环境中再安装一次。

4. 创建几个测试文件，可以借鉴前文的示例。使用 pytest 运行这几个测试文件。

5. 更改 `assert` 语句，不要使用 `assert something == something_else` 这样的语句，尝试使用类似下面这样的例子：

   - `assert 1 in [2, 3, 4]`
   - `assert a < b`
   - `assert 'fizz' not in 'fizzbuzz'`

1.6 预告
What's Next

本章介绍了 pytest 的获取及执行方式，但还没有介绍如何编写测试函数。第 2 章将讲解如何编写测试函数，如何使用不同的参数调用测试函数，如何将测试分发到类、模块和组件包。

第 2 章

编写测试函数
Writing Test Functions

在第 1 章中，我们学习了 pytest 是如何工作的，学习了如何指定测试目录、使用命令行选项。本章将讲解如何为 Python 程序包编写测试函数。即使你要测试的不是 Python 包，本章的大部分内容仍然适用。

在为 Tasks 项目编写测试之前，我会先介绍典型的可分发 Python 包的目录结构。然后演示如何在测试中使用 assert，如何处理可预期和不可预期的异常。

我会讲解如何借助类、模块、目录来组织测试，以便管理大量的测试。还会学习使用 marker 来标记希望同时运行的测试，使用内置 marker 跳过某些测试，为预期会失败的测试做标记。最后将介绍参数化测试，以便使用多组数据开展测试。

2.1 测试示例程序
Testing a Package

下面将以 Tasks 项目来演示如何为 Python 程序包编写测试。Tasks 是一个包含同名命令行工具的 Python 程序包。

附录 D 介绍了如何使用 PyPI 在团队内部或互联网上分发你的项目，在此不多赘述。下面来看看 Tasks 项目的目录结构，请思考各个文件在测试过程中所起到的作用。

以下是 Tasks 项目的文件结构：

```
tasks_proj/
├── CHANGELOG.rst
├── LICENSE
├── MANIFEST.in
├── README.rst
├── setup.py
├── src
│   └── tasks
│   ├── __init__.py
│   ├── api.py
│   ├── cli.py
│   ├── config.py
│   ├── tasksdb_pymongo.py
│   └── tasksdb_tinydb.py
└── tests
    ├── conftest.py
    ├── pytest.ini
    ├── func
    │   ├── __init__.py
    │   ├── test_add.py
    │   └── ...
    └── unit
        ├── __init__.py
        ├── test_task.py
        └── ...
```

这个目录结构展示了测试文件与整个项目的关系，其中有几个关键的文件值得我们注意：`conftest.py`、`pytest.ini`、`__init__.py`、`setup.py`，它们将在测试中发挥重要作用。

所有的测试都放在 `tests` 文件夹里，与包含程序源码的 `src` 文件夹分开。这并非是 pytest 的硬性要求，而是一个很好的习惯。

根目录文件 `CHANGELOG.rst`、`LICENSE`、`README.rst`、`MANIFEST.in`、

setup.py 将在附录 D 中详细介绍。setup.py 对于创建可分发的包很重要，本地其他项目可以将它作为依赖包导入。

功能测试和单元测试放在不同的目录下，这也不是硬性规定，但这样做可以让你更方便地分别运行两类测试。我习惯将这两类测试分开放，因为功能测试只会在改变系统功能时才有可能发生中断异常，而单元测试的中断在代码重构、业务逻辑实现期间都有可能发生。

项目目录中包含两类__init__.py 文件，一类出现在 src 目录下，一类出现在 tests 目录下。src/tasks/__init__.py 告诉 Python 解释器该目录是 Python 包。此外，执行到 import tasks 时，这个__init__.py 将作为该包主入口。该文件包含导入 api.py 模块的代码，因此 api.py 中的函数可以直接被 cli.py 和测试模块访问，比如可以直接使用 task.add()，而不必使用 task.api.add()。

test/func/__init__.py 和 test/unit/__init__.py 都是空文件，它们的作用是给 pytest 提供搜索路径，找到测试根目录以及 pytest.ini 文件。

pytest.ini 文件是可选的，它保存了 pytest 在该项目下的特定配置。项目中顶多包含一个配置文件，其中的指令可以调节 pytest 的工作行为，例如配置常用的命令行选项列表。第 6 章会详细介绍 pytest.ini。

conftest.py 文件同样是可选的，它是 pytest 的“本地插件库”，其中可以包含 hook 函数和 fixture。hook 函数可以将自定义逻辑引入 pytest，用于改善 pytest 的执行流程；fixture 则是一些用于测试前后执行配置及销毁逻辑的外壳函数，可以传递测试中用到的资源。第 3 章和第 4 章会详细介绍 fixtures。第 5 章会详细讲解 hook 函数。在包含多个子目录的测试中，hook 函数和 fixtures 的使用将被定义在 tests/conftests.py 内。同一个项目内可以包含多个 conftest.py 文件，例如 tests 目录下可以有一个 conftest.py 文件，在 tests 的每个子目录下也可以各有一个 conftest.py 文件。

如果你的测试目录没有按照规范来建立，可以在本书网站下载该项目的源码[1]，在后续工作中，可以按照类似结构存放测试文件。

本地安装 Tasks 项目程序包
Installing a Package Locally

测试文件 tests/test_task.py 中包含第 1.2 节用到的测试用例（原来存放在 test_three.py 和 test_four.py 文件里）。现在将这几个函数都归并到同一个文件里，并且使用了更有意义的名字。另外，还删除了原先文件中对 Task 数据结构的定义，它应该写在 api.py 里。

以下是 test_tasks.py：

```
ch2/tasks_proj/tests/unit/test_tasks.py
"""Test the Task data type."""
from tasks import Task

def test_asdict():
    """_asdict() should return a dictionary."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
                'owner': 'okken',
                'done': True, 'id': 21}
    assert t_dict == expected

def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected

def test_defaults():
    """Using no parameters should invoke defaults."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2
```

[1] https://pragprog.com/titles/bopytest/source_code

```
def test_member_access():
    """Check .field functionality of namedtuple."""
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

test_task.py 文件中包含以下 import 语句：

```
from tasks import Task
```

测试中如果希望使用 import tasks 或 from tasks import something，最好的方式是在本地使用 pip 安装 tasks 包。包的根目录包含一个 setup.py 文件，pip 可以直接使用。

你可以切换到 tasks_proj 根目录，运行 pip install .或 pip install -e . 来安装 tasks 包，也可以在上层目录使用 pip install -e task_proj 来安装 tasks 包。

```
$ cd /path/to/code
$ pip install ./tasks_proj/
$ pip install --no-cache-dir ./tasks_proj/
Processing ./tasks_proj
Collecting click (from tasks==0.1.0)
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
  ...
Collecting tinydb (from tasks==0.1.0)
  Downloading tinydb-3.4.0.tar.gz
Collecting six (from tasks==0.1.0)
  Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: click, tinydb, six, tasks
  Running setup.py install for tinydb ... done
  Running setup.py install for tasks ... done
Successfully installed click-6.7 six-1.10.0 tasks-0.1.0 tinydb-3.4.0
```

如果仅仅是测试 tasks 包，这个命令就足够了，但是，如果安装后希望修改源码重新安装，就需要使用-e 选项（editable）。

```
$ pip install -e ./tasks_proj/
Obtaining file:///path/to/code/tasks_proj Requirement already
satisfied: click in
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: tinydb in
```

```
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: six in
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Installing collected packages: tasks
  Found existing installation: tasks 0.1.0
    Uninstalling tasks-0.1.0:
      Successfully uninstalled tasks-0.1.0
  Running setup.py develop for tasks
Successfully installed tasks
```

现在来运行测试：

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest test_task.py
====================== test session starts ======================
collected 4 items

test_task.py ....
=================== 4 passed in 0.01 seconds ====================
```

`import` 语句执行成功，说明 `tasks` 包已正常安装。接下来的测试就可以使用 `import tasks` 了，下面来编写一些测试。

2.2 使用 assert 声明
Using assert Statements

用 pytest 编写测试时，若需要传递测试失败信息，则可以直接使用 Python 自带的 `assert` 关键字，这样做很方便，也是许多开发者选择 pytest 的原因。

如果使用其他测试框架，则可能会看到许多以 `assert` 开头的函数。下面列举了使用 `assert` 与各种以 `assert` 开头的函数的区别：

```
pytest               unittest
assert something     assertTrue(something)
assert a == b        assertEqual(a, b)
assert a <= b        assertLessEqual(a, b)
...                  ...
```

pytest 允许在 `assert` 关键字后面添加任何表达式（`assert <expression>`）。如果表达式的值通过 `bool` 转换后等于 `False`，则意味着测试失败。

pytest 有一个重要功能是可以重写 assert 关键字。pytest 会截断对原生 assert 的调用，替换为 pytest 定义的 assert，从而提供更多的失败信息和细节。在下面的例子中，可以看到重写 assert 关键字是多么必要：

ch2/tasks_proj/tests/unit/test_task_fail.py

```
"""Use the Task type to show test failures."""
from tasks import Task

def test_task_equality():
    """Different tasks should not be equal."""
    t1 = Task('sit there', 'brian')
    t2 = Task('do something', 'okken')
    assert t1 == t2

def test_dict_equality():
    """Different tasks compared as dicts should not be equal."""
    t1_dict = Task('make sandwich', 'okken')._asdict()
    t2_dict = Task('make sandwich', 'okkem')._asdict()
    assert t1_dict == t2_dict
```

以上都是 assert 失败的例子，可以看到回溯的信息很丰富。

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest test_task_fail.py
===================== test session starts =====================
collected 2 items

test_task_fail.py FF
========================== FAILURES ===========================
_____________________ test_task_equality ______________________
    def test_task_equality():
        t1 = Task('sit there', 'brian')
        t2 = Task('do something', 'okken')
>       assert t1 == t2
E       AssertionError: assert Task(summary=...alse, id=None) ==
Task(summary='...alse, id=None)
E         At index 0 diff: 'sit there' != 'do something'
E         Use -v to get the full diff
test_task_fail.py:6: AssertionError
_____________________ test_dict_equality ______________________
    def test_dict_equality():
        t1_dict = Task('make sandwich', 'okken')._asdict()
        t2_dict = Task('make sandwich', 'okkem')._asdict()
>       assert t1_dict == t2_dict
E       AssertionError: assert OrderedDict([...('id', None)]) ==
OrderedDict([(...('id', None)])
```

```
E        Omitting 3 identical items, use -v to show
E        Differing items:
E        {'owner': 'okken'} != {'owner': 'okkem'}
E        Use -v to get the full diff
test_task_fail.py:11: AssertionError
================== 2 failed in 0.06 seconds ==================
```

上面每个失败的测试用例在行首都用一个>号来标识。以 E 开头的行是 pytest 提供的额外判定信息，用于帮助我们了解异常的具体情况。

我有意将两列不匹配的参数放到测试用例 test_task_equality()中，但我们看到 pytest 只展示了第一列结果。按照错误信息中给出的建议，我们使用-v 选项再执行一遍：

```
$ pytest -v test_task_fail.py::test_task_equality
====================== test session starts ======================
collected 3 items

test_task_fail.py::test_task_equality FAILED
========================== FAILURES ==========================
_____________________ test_task_equality _____________________
    def test_task_equality():
        t1 = Task('sit there', 'brian')
        t2 = Task('do something', 'okken')
>       assert t1 == t2
E       AssertionError: assert Task(summary=...alse, id=None) ==
Task(summary='...alse, id=None)
E         At index 0 diff: 'sit there' != 'do something'
E         Full diff:
E         - Task(summary='sit there', owner='brian', done=False,
id=None)
E         ? ^^^ ^^^ ^^^^
E         + Task(summary='do something', owner='okken', done=False,
id=None)
E         ? +++^^^ ^^^ ^^^^
test_task_fail.py:6: AssertionError
================== 1 failed in 0.07 seconds ==================
```

这样就更一目了然了。pytest 不单指出了区别，还给出了区别详情。

这个例子中的 assert 仅用于判定是否相等。你可以在 pytest.org 上找到很多更复杂的 assert 例子。

2.3 预期异常
Expecting Exceptions

在 Tasks 项目的 API 中，有几个地方可能抛出异常。tasks/api.py 中有以下几个函数。

```
def add(task): # type: (Task) -> int
def get(task_id): # type: (int) -> Task
def list_tasks(owner=None): # type: (str|None) -> list of Task
def count(): # type: (None) -> int
def update(task_id, task): # type: (int, Task) -> None
def delete(task_id): # type: (int) -> None
def delete_all(): # type: () -> None
def unique_id(): # type: () -> int
def start_tasks_db(db_path, db_type): # type: (str, str) -> None
def stop_tasks_db(): # type: () -> None
```

cli.py 中的 CLI 代码与 api.py 中的 API 代码统一指定了发送给 API 函数的数据类型，假设检查到数据类型错误，异常很可能是由这些 API 函数抛出的。

为确保这些函数在发生类型错误时可以抛出异常，下面来做一下检验：在测试中使用错误类型的数据，引起 TypeError 异常。同时，我还使用了 with pytest.raises(<expected exception>)声明，像这样：

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
import pytest
import tasks

def test_add_raises():
    """add() should raise an exception with wrong type param."""
    with pytest.raises(TypeError):
        tasks.add(task='not a Task object')
```

测试用例 test_add_raises()中有 with pytest.raises(TypeError)声明，意味着无论 with 中的内容是什么，都至少会发生 TypeError 异常。如果测试通过，说明确实发生了我们预期的 TypeError 异常；如果抛出的是其他类型的异常，则与我们所预期的不一致，说明测试失败。

上面的测试中只检验了传参数据的“类型异常”，你也可以检验“值异常”。对于 start_tasks_db(db_path, db_type)来说，db_type 不单要求是字符串类型，还必须为'tiny'或'mongo'。为校验异常消息是否符合预期，可以通过增加 as excinfo 语句得到异常消息的值，再进行比对。

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
def test_start_tasks_db_raises():
    """Make sure unsupported db raises an exception."""
    with pytest.raises(ValueError) as excinfo:
        tasks.start_tasks_db('some/great/path', 'mysql')
    exception_msg = excinfo.value.args[0]
    assert exception_msg == "db_type must be a 'tiny' or 'mongo'"
```

现在可以更细致地观察异常。as 后面的变量是 ExceptionInfo 类型，它会被赋予异常消息的值。

在这个例子中，我们希望确保第一个（也是唯一的）参数引起的异常消息能与某个字符串匹配。

2.4 测试函数的标记
Marking Test Functions

pytest 提供了标记机制，允许你使用 marker 对测试函数做标记。一个测试函数可以有多个 marker，一个 marker 也可以用来标记多个测试函数。

讲解 marker 的作用需要看实际的例子。比如我们选一部分测试作为冒烟测试，以了解当前系统中是否存在大的缺陷。通常冒烟测试不会包含全套测试，只选择可以快速出结果的测试子集，让开发者对系统健康状况有一个大致的了解。

为了把选定的测试加入冒烟测试，可以对它们添加@pytest.mark.smoke 装饰器。先用 test_api_exceptions.py 中的两个测试作为例子（注意：smoke 和 get 标记是我定义的，并非 pytest 内置的）。

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
@pytest.mark.smoke
def test_list_raises():
    """list() should raise an exception with wrong type param."""
    with pytest.raises(TypeError):
        tasks.list_tasks(owner=123)

@pytest.mark.get
@pytest.mark.smoke
def test_get_raises():
    """get() should raise an exception with wrong type param."""
    with pytest.raises(TypeError):
        tasks.get(task_id='123')
```

现在只需要在命令中指定-m marker_name，就可以运行它们。

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v -m 'smoke' test_api_exceptions.py
====================== test session starts ======================
collected 7 items

test_api_exceptions.py::test_list_raises PASSED
test_api_exceptions.py::test_get_raises PASSED
====================== 5 tests deselected =======================
============ 2 passed, 5 deselected in 0.03 seconds ============
$ pytest -v -m 'get' test_api_exceptions.py
====================== test session starts ======================
collected 7 items

test_api_exceptions.py::test_get_raises PASSED
====================== 6 tests deselected =======================
============ 1 passed, 6 deselected in 0.01 seconds ============
```

别忘了-v 是--verbose 的简写，它用于展示具体运行的测试的名字。指定-m 'smoke'，会运行标记为@pytest.mark.smoke 的两个测试；而指定-m 'get'，会运行标记为@pytest.mark.get 的那个测试。这样做显得清晰明了。

-m 后面也可以使用表达式，可以在标记之间添加 and、or、not 关键字。

```
$ pytest -v -m 'smoke and get' test_api_exceptions.py
====================== test session starts ======================
collected 7 items
test_api_exceptions.py::test_get_raises PASSED
====================== 6 tests deselected =======================
============ 1 passed, 6 deselected in 0.03 seconds ============
```

上面的命令只会运行既有 smoke 标记，又有 get 标记的测试。还可以加上 not。

```
$ pytest -v -m 'smoke and not get' test_api_exceptions.py
====================== test session starts ======================
collected 7 items
test_api_exceptions.py::test_list_raises PASSED
====================== 6 tests deselected =======================
============ 1 passed, 6 deselected in 0.03 seconds ============
```

'smoke and not get'的作用是筛选出有 smoke 标记，但没有 get 标记的测试。

完善冒烟测试

Filling Out the Smoke Test

上面的例子还不能算是合理的冒烟测试，因为两个测试并未涉及数据库改动（显然这是有必要的）。

现在来编写两个测试，它们都包含增加 task 对象的逻辑，然后把其中一个放到冒烟测试里。

ch2/tasks_proj/tests/func/test_add.py

```
import pytest
import tasks
from tasks import Task

def test_add_returns_valid_id():
    """tasks.add(<valid task>) should return an integer."""
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    # THEN returned task_id is of type int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)

@pytest.mark.smoke
def test_added_task_has_id_set():
    """Make sure the task_id field is set by tasks.add()."""
    # GIVEN an initialized tasks db
    # AND a new task is added
```

```
    new_task = Task('sit in chair', owner='me', done=True)
    task_id = tasks.add(new_task)

    # WHEN task is retrieved
    task_from_db = tasks.get(task_id)

    # THEN task_id matches id field
    assert task_from_db.id == task_id
```

两个测试中都包含一句注释：GIVEN an initialized tasks db，但实际上并未涉及数据库初始化。现在这里可以定义一个 fixture，用于测试前后控制数据库的连接。

ch2/tasks_proj/tests/func/test_add.py

```
@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    """Connect to db before testing, disconnect after."""
    # Setup : start db
    tasks.start_tasks_db(str(tmpdir), 'tiny')

    yield # this is where the testing happens

    # Teardown : stop db
    tasks.stop_tasks_db()
```

这个例子中使用的 tmpdir 是 pytest 内置的 fixture。第 4 章将详细介绍内置 fixture，第 3 章将讲解如何编写及使用自定义的 fixture，以及这里用到的 autouse 参数。

本例中的 autouse 表示当前文件中的所有测试都将使用该 fixture。yield 之前的代码将在测试运行前执行，而 yield 之后的代码会在测试运行后执行。若有必要，yield 也可以返回数据给测试。后续章节会详细介绍 fixture，这里由于需要初始化数据库，才提前使用它。（pytest 也支持经典的 nose 和 unittest 风格的 setup 和 teardown 函数。感兴趣的读者可以查阅附录 F。）

我们暂时把对 fixture 的讨论放到一边。请切换到项目根目录，运行我们的冒烟测试。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v -m 'smoke'
====================== test session starts ======================
collected 56 items

tests/func/test_add.py::test_added_task_has_id_set PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
====================== 53 tests deselected ======================
=========== 3 passed, 53 deselected in 0.11 seconds ============
```

可以看到，带有相同标记的测试即使存放在不同的文件下，也会被一起执行。

2.5 跳过测试 Skipping Tests

第 2.4 节使用的标记是自定义的。pytest 自身内置了一些标记：`skip`、`skipif`、`xfail`。本节介绍 `skip` 和 `skipif`，下一节介绍 `xfail`。

`skip` 和 `skipif` 允许你跳过不希望运行的测试。比方，我们不确定 `tasks.unique_id()`是否会按照预期的方式工作：它每次调用后返回的是不同的数值吗？它会返回一个数据库中尚不存在的值吗？

为此，下面来编写一个测试（注意上一节的 `initialized_tasks_db` fixture 仍然有效）。

ch2/tasks_proj/tests/func/test_unique_id_1.py

```
import pytest
import tasks
def test_unique_id():
    """Calling unique_id() twice should return different numbers."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

然后执行这个测试。

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest test_unique_id_1.py
===================== test session starts ======================
collected 1 item s

test_unique_id_1.py F
============================ FAILURES ============================
_____________________ test_unique_id _____________________
    def test_unique_id():
        """Calling unique_id() twice should return different numbers."""
        id_1 = tasks.unique_id()
        id_2 = tasks.unique_id()
>       assert id_1 != id_2
E       assert 1 != 1
test_unique_id_1.py:12: AssertionError
=================== 1 failed in 0.06 seconds ===================
```

很遗憾，我们的测试方法有问题。查看 API 文档后，发现其中有一句话：`"""Return an integer that does not exist in the db."""`（会返回一个数据库中不存在的整型数，但是没有说会保证每次返回值都不同。）

我们可以直接修改测试，不过让我们试试把它标记为skip。

ch2/tasks_proj/tests/func/test_unique_id_2.py

```
@pytest.mark.skip(reason='misunderstood the API')
def test_unique_id_1():
    """Calling unique_id() twice should return different numbers."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2

def test_unique_id_2():
    """unique_id() should return an unused id."""
    ids = []
    ids.append(tasks.add(Task('one')))
    ids.append(tasks.add(Task('two')))
    ids.append(tasks.add(Task('three')))
    # grab a unique id
    uid = tasks.unique_id()
    # make sure it isn't in the list of existing ids
    assert uid not in ids
```

要跳过某个测试，只需要简单地在测试函数上方添加`@pytest.mark.skip()`装饰器即可。

再运行一次。

```
$ pytest -v test_unique_id_2.py
==================== test session starts ============================
collected 2 items

test_unique_id_2.py::test_unique_id_1 SKIPPED
test_unique_id_2.py::test_unique_id_2 PASSED
==============1 passed, 1 skipped in 0.02 seconds ==================
```

实际上，可以给要跳过的测试添加理由和条件，比如希望它只在包版本低于 0.2.0 时才生效，这时应当使用 `skipif` 来替代 `skip`：

```
ch2/tasks_proj/tests/func/test_unique_id_3.py
@pytest.mark.skipif(tasks.__version__ <'0.2.0',
                    reason='not supported until version 0.2.0')

def test_unique_id_1():
    """Calling unique_id() twice should return different numbers."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

`skipif()`的判定条件可以是任何 Python 表达式，这里比对的是包版本。

无论是 `skip` 还是 `skipif`，我都写上了跳过的理由（尽管 `skip` 并不要求这样做）。我习惯在使用 `skip`、`skipif`、`xfail` 时都写清楚理由。

以下是修改后的测试输出情况。

```
$ pytest test_unique_id_3.py
==================== test session starts ============================
collected 2 items

test_unique_id_3.py s.
============ 1 passed, 1 skipped in 0.02 seconds ==================
```

`s.`表明有一个测试被跳过，有一个测试通过。使用`-v`选项可以看到具体是哪一个测试被跳过，哪一个测试通过了。

```
$ pytest -v test_unique_id_3.py
==================== test session starts ============================
collected 2 items
```

```
test_unique_id_3.py::test_unique_id_1 SKIPPED
test_unique_id_3.py::test_unique_id_2 PASSED
============= 1 passed, 1 skipped in 0.03 seconds ===================
```

但我们仍然看不到跳过的原因，这时可以使用-rs。

```
$ pytest -rs test_unique_id_3.py
======================== test session starts ========================
collected 2 items

test_unique_id_3.py s.
====================== short test summary info ======================
SKIP [1] func/test_unique_id_3.py:5: not supported until version 0.2.0
================ 1 passed, 1 skipped in 0.03 seconds ================
```

-r chars 选项有帮助文档。

```
$ pytest --help
...
    -r chars

    show extra test summary info as specified by chars
    (f)ailed, (E)error, (s)skipped, (x)failed, (X)passed,
    (p)passed, (P)passed with output, (a)all except pP.
...
```

该选项不但可以帮助用户了解某些测试被跳过的原因，还可以用于查看其他测试结果。

2.6 标记预期会失败的测试
Marking Tests as Expecting to Fail

使用 skip 和 skipif 标记，测试会直接跳过，而不会被执行。使用 xfail 标记，则告诉 pytest 运行此测试，但我们预期它会失败。接下来，我会使用 xfail 修改 unique_id()测试：

```
ch2/tasks_proj/tests/func/test_unique_id_4.py
@pytest.mark.xfail(tasks.__version__ <'0.2.0',
                   reason='not supported until version 0.2.0')
def test_unique_id_1():
```

```
    """Calling unique_id() twice should return different numbers."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2

@pytest.mark.xfail()
def test_unique_id_is_a_duck():
    """Demonstrate xfail."""
    uid = tasks.unique_id()
    assert uid == 'a duck'

@pytest.mark.xfail()
def test_unique_id_not_a_duck():
    """Demonstrate xpass."""
    uid = tasks.unique_id()
    assert uid != 'a duck'
```

第一个测试与上一节的一样，只不过我用 `xfail` 替换了 `skip`。后面两个测试都使用了 `xfail` 标记，区别在于一个是`==`，一个是`!=`，所以结果必然是一个通过，一个失败。

下面执行该测试文件。

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest test_unique_id_4.py
======================== test session starts ========================
collected 4 items

test_unique_id_4.py xxX.
========== 1 passed, 2 xfailed, 1 xpassed in 0.07 seconds ===========
```

`x` 代表 `XFAIL`，意味着“expected to fail”（预期失败，实际上也失败了）。大写的 `X` 代表 `XPASS`，意味着“expected to fail but passed”（预期失败，但实际运行并没有失败）。

使用`--verbose` 选项，将逐条输出更多信息。

```
$ pytest -v test_unique_id_4.py
======================== test session starts ========================
collected 4 items

test_unique_id_4.py::test_unique_id_1 xfail
test_unique_id_4.py::test_unique_id_is_a_duck xfail
test_unique_id_4.py::test_unique_id_not_a_duck XPASS
```

```
test_unique_id_4.py::test_unique_id_2 PASSED
========== 1 passed, 2 xfailed, 1 xpassed in 0.08 seconds ===========
```

对于标记为 xfail，但实际运行结果是 XPASS 的测试，可以在 pytest 配置中强制指定结果为 FAIL（像下面这样修改 pytest.ini 文件即可）。

```
[pytest]
xfail_strict=true
```

第 6 章将详细介绍 pytest.ini 文件。

2.7 运行测试子集
Running a Subset of Tests

前面已经讨论了如何给测试做标记，以及如何根据标记运行测试。运行测试子集有很多种方式，不但可以选择运行某个目录、文件、类中的测试，还可以选择运行某一个测试用例（可能在文件中，也可能在类中）。本节将介绍测试类，并通过表达式来完成测试函数名的字符串匹配。

单个目录
A Single Directory

运行单个目录下的所有测试，以目录作为 pytest 的参数即可。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest tests/func --tb=no
====================== test session starts ======================
collected 50 items

tests/func/test_add.py ..
tests/func/test_add_variety.py ............................
tests/func/test_api_exceptions.py .......
tests/func/test_unique_id_1.py F
tests/func/test_unique_id_2.py s.
tests/func/test_unique_id_3.py s.
tests/func/test_unique_id_4.py xxX.
 1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 0.26 seconds
```

加上-v选项后再运行一次。通过观察其中的命名输出方式，学习如何指定

目录、类、测试。

```
$ pytest -v tests/func --tb=no
====================== test session starts ======================
collected 50 items

tests/func/test_add.py::test_add_returns_valid_id PASSED
tests/func/test_add.py::test_added_task_has_id_set PASSED
...
tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
...
tests/func/test_unique_id_1.py::test_unique_id FAILED
tests/func/test_unique_id_2.py::test_unique_id_1 SKIPPED
tests/func/test_unique_id_2.py::test_unique_id_2 PASSED
...
tests/func/test_unique_id_4.py::test_unique_id_1 xfail
tests/func/test_unique_id_4.py::test_unique_id_is_a_duck xfail
tests/func/test_unique_id_4.py::test_unique_id_not_a_duck XPASS
tests/func/test_unique_id_4.py::test_unique_id_2 PASSED

 1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 0.30 seconds
```

下面几个例子会用到上面的命名方式。

单个测试文件/模块
A Single Test File/Module

运行单个文件里的全部测试，以路径名加文件名作为 pytest 参数即可。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest tests/func/test_add.py
====================== test session starts ======================
collected 2 items

tests/func/test_add.py ..
================== 2 passed in 0.05 seconds ======================
```

这样的例子前面已经出现许多次了。

单个测试函数
A Single Test Function

运行单个测试函数，只需要在文件名后面添加::符号和函数名。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_add.py::test_add_returns_valid_id
==================== test session starts ==========================
collected 3 items

tests/func/test_add.py::test_add_returns_valid_id PASSED
================= 1 passed in 0.02 seconds ========================
```

使用-v 可以显示执行的是哪个函数。

单个测试类
A Single Test Class

测试类用于将某些相似的测试函数组合在一起，举个例子：

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
class TestUpdate():
    """Test expected exceptions with tasks.update()."""

    def test_bad_id(self):
        """A non-int id should raise an excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id={'dict instead': 1},
                         task=tasks.Task())

    def test_bad_task(self):
        """A non-Task task should raise an excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id=1, task='not a task')
```

由于这两个测试都是在测试 update()函数，把它们放到一个类中是合理的。要运行该类，可以在文件名后面加上::符号和类名（与运行单个测试函数类似）。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate
=================== test session starts ===========================
collected 7 items
```

```
tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
tests/func/test_api_exceptions.py::TestUpdate::test_bad_task PASSED
================== 2 passed in 0.03 seconds ==========================
```

单个测试类中的测试方法
A Single Test Method of a Test Class

如果不希望运行测试类中的所有测试，只想指定运行其中一个，一样可以在文件名后面添加::符号和方法名。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate::test_bad_id
===================== test session starts ======================
collected 1 item

tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
=================== 1 passed in 0.03 seconds ===================
```

通过列举详情了解句法

注意：运行测试子集时，不必记忆指定目录、文件、函数、类、类方法的句法，其格式与 `pytest -v` 的输出是一致的。

用测试名划分测试集合
A Set of Tests Based on Test Name

-k选项允许用一个表达式指定需要运行的测试，该表达式可以匹配测试名（或其子串）。表达式中可以包含 and、or、not。

下面来运行所有名字中包含_raises 的测试。

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v -k _raises
===================== test session starts ======================
collected 56 items
```

```
tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_delete_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED
====================== 51 tests deselected ======================
=========== 5 passed, 51 deselected in 0.07 seconds ============
```

如果要跳过 test_delete_raises()的执行，则可以使用 and 和 not。

```
$ pytest -v -k "_raises and not delete"
====================== test session starts ======================
collected 56 items

tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED
====================== 52 tests deselected ======================
=========== 4 passed, 52 deselected in 0.06 seconds ============
```

上面学习了如何指定测试文件、目录、类、函数，如何利用-k选项选择部分测试。下面介绍如何将一个测试函数用作多个测试用例，即以多组测试数据运行。

2.8 参数化测试
Parametrized Testing

向函数传值并检验输出结果是软件测试的常用手段，但是对大部分功能测试而言，仅仅使用一组数据是无法充分测试函数功能的。参数化测试允许传递多组数据，一旦发现测试失败，pytest 会及时报告。

为了介绍参数化测试需要解决的问题，我举一个关于 add()的例子。

ch2/tasks_proj/tests/func/test_add_variety.py

```
import pytest
import tasks
from tasks import Task
```