Chapter 1 安装和启动

pytest作为一个测试框架，可以非常简单的建立易用性好，扩展性强的测试集。这些测试因为避免了大量的样板代码，所以可读性非常高。

你可以花费一点时间通过一个unittest或者略复杂的函数测试来验证你的应用程序或者库。

1.1 安装pytest

1.在你的python环境下运行下面的命令即可安装pytest

```
pip install -U pytest
```

2.检查你安装的pytest的版本信息是否正确：

```
$ pytest --version
This is pytest version 4.2.1, imported from /Users/david/Downloads/myPython/python3-
venv/lib/python3.7/site-packages/pytest.py
```

1.2 创建你的第一个测试

创建一个只有4行代码的简单函数：

```
# test_sample.py 的内容
def func(x):
    return x + 1
def test_answer():
    assert func(3) == 5
```

现在，你可以在test_sample.py的同级目录下直接执行pytest，结果如下：

```
$ pytest
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_sample.py F                                                      [100%]

================================= FAILURES =================================
_____________________________________ test_answer _____________________________________

    def test_answer():
>       assert func(3) == 5
```

```
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:4: AssertionError
========================== 1 failed in 0.07 seconds ==========================
```

这个测试的结果是失败的，因为func(3)的返回值不是5

1.3 运行多个测试

pytest会运行当前目录及子目录下所有以 test_*.py 和 *_test.py 命名的文件。文件匹配方式遵循 Standard test discovery rules

1.4 判断是否发生了指定的异常

使用raises可以判断代码是否抛出了异常：

```
# test_sysexit.py 的内容
import pytest
def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

使用"quiet"模式来执行这个测试：

```
$ pytest -q test_sysexit.py
.                                                                    [100%]
1 passed in 0.03 seconds
```

1.5 将多个测试用例放在一个class中

当你需要开发多个测试用例的时候，你可能需要将他们放在同一个class中，pytest可以很简单的创建包含多个测试用例的class：

```
# test_class.py的内容
class TestClass(object):
    def test_one(self):
        x = "this"
        assert 'h' in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, 'check')
```

pytest根据Conventions for Python test discovery查找所有的测试用例，所以可以找到所有以**test_**开头的测试函数。我们可以通过文件名来直接运行整个模块：

```
$ pytest -q test_class.py
.F                                                                   [100%]
================================= FAILURES =================================
_____________________________ TestClass.test_two _____________________________

self = <test_class.TestClass object at 0x10858fc18>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E    AssertionError: assert False
E     +  where False = hasattr('hello', 'check')

test_class.py:8: AssertionError
1 failed, 1 passed in 0.07 seconds
```

第一个测试用例passed，第二个测试用例failed。你可以很直观的观察到测试用例中进行判断的中间值，这可以帮助理解测试用例失败的原因。

1.6 为测试创建唯一的临时文件夹

pytest 提供 Builtin fixtures/function arguments来创建任意的资源，比如一个具有唯一的临时文件夹：

```
# test_tmpdir.py的内容
def test_needsfiles(tmpdir):
    print(tmpdir)
    assert 0
```

如果函数的签名中(函数签名包括函数的参数和返回值，以及参数的封送顺序等等)包含参数tmpdir，pytest就会在执行测试用例之前查找并调用特定的fixture创建所需资源。在本例中，pytest会创建一个unique-per-test-invocation临时文件夹：

```
$ pytest -q test_tmpdir.py
F
[100%]
================================= FAILURES =================================
_____________________________ test_needsfiles ______________________________

tmpdir = local('/private/var/folders/st/05jlyljx7nqb26zdr8nv17dw0000gn/T/pytest-of-
david/pytest-0/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print(tmpdir)
>       assert 0
E    assert 0
```

```
test_tmpdir.py:3: AssertionError
------------------------- Captured stdout call -------------------------
/private/var/folders/st/05jlyljx7nqb26zdr8nv17dw0000gn/T/pytest-of-david/pytest-
0/test_needsfiles0
1 failed in 0.07 seconds
```

关于tmpdir的更多信息请参考Temporary directories and files 通过下面的命令可以查看所有内置的pytest fixture：

```
pytest --fixtures
```

Chapter 2 用法

2.1 通过python -m pytest调用pytest

这是在2.0版本中新引入的功能。 你可以通过python的解释器，利用命令行来调用测试：

```
python -m pytest [...]
```

这种调用方式几乎等同于直接调用pytest [...],但需要注意的是这种通过python来调用的方式同时会将当前目录添加到sys.path

2.2 退出码

pytest有以下6种退出码：

Exit code 0: 找到所有测试用例并测试通过

Exit code 1: 找到测试用例并运行但是部分测试用例运行失败

Exit code 2: 用户中断了测试

Exit code 3: 执行过程中发生了内部错误

Exit code 4: pytest命令行使用错误

Exit code 5: 没有找到任何测试用例

2.3 版本信息，参数名，环境变量的帮助

```
pytest --version #显示pytest的import的路径
pytest --fixtures #显示内置的函数参数
pytest -h | --help #帮助信息
```

2.4 第一(N)次测试失败后停止

使用下面的参数可以让测试在第1(N)次测试失败后停止：

```
pytest -x  # 第一次测试失败后停止测试
pytest --maxfail=2 # 第2次测试失败后停止测试
```

2.5 指定/选择测试用例

Pytest在命令行中支持多种方式来运行和选择测试用例：

对模块中进行测试：

```
pytest test_mod.py
```

对文件夹中进行测试：

```
pytest testing/
```

通过关键字表达式来进行测试：

```
pytest -k "MyClass and not method"
```

这种方式会执行文件名，类名以及函数名与给定的字符串表达式相匹配的测试用例。 上面的用例会执行TestMyClass.test_something但是不会执行TestMyClass.test_method_simple

and not 在这里是表达式，未经过测试

通过节点id来进行测试

每个被选中的测试用例都会被分配一个唯一的nodeid，它由模块文件名和以下说明符组成:参数化的类名、函数名和参数，用::分隔。

可以通过下面的方式运行模块中的指定的测试用例：

```
pytest test_mod.py::test_func
```

也可以通过下面这种方式：

```
pytest test_mod.py::TestClass::test_method
```

通过标记符来进行测试

```
pytest -m slow
```

这种方式会运行所有通过装饰器 @pytest.mark.slow进行装饰的测试用例。

关于标记符请参考marks

通过包来运行

```
pytest --pyargs pkg.testing
```

这种方式会导入pkg.testing，并且基于该包所在为止来查找并运行测试用例。

2.6 修改python的traceback的打印

```
pytest --showlocals #在tracebacks中显示本地变量
pytest -l           #同上

pytest --tb=auto    #(默认显示方式) 该方式会显示tracebacks的第一条和最后一条的详细信息
                    #其余的信息会简略显示

pytest --tb=long    #详细显示所有的tracebacks
pytest --tb=short   #简略显示tracebacks
pytest --tb=line    #每个错误只显示一行
pytest --tb=native  #以python标准库模式输出
pytest --tb=no      #不输出tracebacks
```

另外还有比**–tb=long输出更详细的参数–full-trace**。在该参数下，KeyboardInterrupt(Ctrl+C)在中断traceback的输出的时候同时会打印栈信息。 这在测试耗时过长的时候非常有用，使用该组合可以帮助我们找到测试用例挂死在何处。

如果使用默认方式，测试被中断的时候不会有任何栈信息输出(因为KeyboardInterrupt被pytest捕获了)，使用该参数可以保证traceback能够显示出来。(这段翻译怪怪的)

2.7 详尽的测试报告

这是2.9的新功能。

参数**-r**可以用来在测试结束后展示一份“测试概要信息”，这使得在大型测试套中获取一份清楚的测试结果(失败，跳过等测试信息)十分简单。

示例：

```
#test_example.py的内容
import pytest

@pytest.fixture
def error_fixture():
```

```
    assert 0

def test_ok():
    print("ok")

def test_fail():
    assert 0

def test_error(error_fixture):
    pass

def test_skip():
    pytest.skip("skipping this test")

def test_xfail():
    pytest.xfail("xfailing this test")

@pytest.mark.xfail(reason="always xfail")
def test_xpass():
    pass
```

```
C:\>pytest -ra
========================== test session starts ==========================
platform win32 -- Python 3.7.2, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
rootdir: TMPDIR, inifile:
collected 6 items

test_example.py .FEsxX
[100%]

========================== ERRORS ==========================
_____________________________ ERROR at setup of test_error _____________________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E    assert 0

test_example.py:5: AssertionError
========================== FAILURES ==========================
_____________________________ test_fail _____________________________

    def test_fail():
>       assert 0
E    assert 0

test_example.py:11: AssertionError
========================== short test summary info ==========================
SKIPPED [1] C:\test_example.py:18: skipping this test
XFAIL test_example.py::test_xfail
  reason: xfailing this test
XPASS test_example.py::test_xpass always xfail
ERROR test_example.py::test_error
FAILED test_example.py::test_fail
```

```
========== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.09 seconds
==========
```

-r后可以追加一些参数，上面示例中的a表示"除了passes的所有信息"。

可以追加的参数列表如下：

f - failed

E - error

s - skipped

x - xfailed

X - xpassed

p - passed

P - passed with output

a - all except pP

可以同时追加多个参数，如果你想只看"failed"和"skipped"的测试结果，你可以执行：

```
C:\>pytest -rfs
========================== test session starts ==========================
platform win32 -- Python 3.7.2, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
rootdir: TMPDIR, inifile:
collected 6 items

test_example.py .FEsxX
[100%]

========================== ERRORS ==========================
_____________________________ ERROR at setup of test_error _____________________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E    assert 0

test_example.py:5: AssertionError
========================== FAILURES ==========================
_____________________________ test_fail _____________________________

    def test_fail():
>       assert 0
E    assert 0

test_example.py:11: AssertionError
========================== short test summary info ==========================
FAILED test_example.py::test_fail
SKIPPED [1] C:\test_example.py:18: skipping this test
```

```
========================== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error
in 0.13 seconds ==========================
```

p可以用来显示pass的测试用例，P会在测试报告中增加一段"PASSES"的信息来显示通过的测试用例：

```
C:\>pytest -rpP
========================== test session starts ==========================
platform win32 -- Python 3.7.2, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
rootdir: TMPDIR, inifile:
collected 6 items

test_example.py .FEsxX
[100%]

============================ ERRORS ============================
_____________________________ ERROR at setup of test_error _____________________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E    assert 0

test_example.py:5: AssertionError
============================ FAILURES ============================
_____________________________ test_fail _____________________________

    def test_fail():
>       assert 0
E    assert 0

test_example.py:11: AssertionError
========================== short test summary info ==========================
PASSED test_example.py::test_ok
============================ PASSES ============================
_____________________________ test_ok _____________________________
--------------------------- Captured stdout call ---------------------------
ok
========================== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error
in 0.08 seconds ==========================
```

2.8 测试失败时自动调用PDB

pytest允许通过命令行使能在测试失败时自动调用python的内置调试工具PDB：

```
pytest --pdb
```

这会在每次测试失败(或者发生KeyboardInterrupt)的时候都去调用PDB。通常我们可能只需要在第一次测试失败的时候来调用pdb：

```
pytest -x --pdb          #首次失败的时候调用pdb，然后结束测试进程
pytest --pdb --maxfail=3 #前三次失败的时候调用pdb
```

注意所有的异常信息都会保存在sys.last_value, sys.last_type和sys.last_traceback中。 在交互使用中，这允许使用任何调试工具进入后期调试。我们也可以手动的访问这些异常信息，如下：

```
(Pdb) import sys
(Pdb) sys.last_traceback.tb_lineno
1448
(Pdb) sys.last_value
AssertionError('assert 0')
(Pdb) sys.last_type
<class 'AssertionError'>
```

2.9 测试启动时调用PDB

pytest允许在测试启动时立即调用pdb：

```
pytest --trace
```

这会在每个测试开始时立即调用python的pdb

2.10 设置断点

在代码中使用python的原生接口**python import pdb; pdb.set_trace()**来设置断点，在pytest中会自动禁用该测试的输出捕获：

*其他测试的输出捕获不会受影响

*先前的测试中已经被捕获的输出会被正常处理

*同一测试中的后续输出不会被捕获，而被转发给sys.stdout。注意即使退出交互式pdb继续运行测试，这一设置依然生效

2.11 使用内置的断点函数

python3.7 引入了内置的断点函数 breakpoint()，pytest支持该函数：

*当breakpoint()被调用，并且PYTHONBREAKPOINT是默认值时，pytest将会使用内部自定义的PDB UI， 而不是系统默认的PDB

测试完成后，返回系统默认的PDB UI 当使用–pdb**参数时，breakpoint()和测试异常时都会使用内部自定义的PDB UI

```
-pdbcls可以用于指定自定义的调试类(这是什么鬼)
```

2.12 分析测试时间

显示最慢的10个测试步骤：

```
pytest --durations=10
```

默认情况下，如果测试时间很短(<0.01s)，这里不会显示执行时常，如果需要显示，在命令行中追加**-vv**参数

2.13 创建 JUnitXML 格式的文件

使用下面的方式可以创建Jekins或者其他的集成测试环境可读的结果文件：

```
pytest --junitxml=path
```

path是生成的XML文件

3.1引入的新特性：

可以在pytest的配置文件中设置junit_suite_name属性来配置测试结果XML中的root名称

```
[pytest]
junit_suite_name = my_suite
```

4.0引入的新特性：

Junit XML规范中"time"属性应该是总的测试执行的时间，包含setup和teardown。这也是pytest的默认行为。如果想要只显示测试的时间(call duration)，配置junit_duration_report:

```
[pytest]
junit_duration_report = call
```

2.13.1 record_property

省略

2.13.2 record_xml_attribute

3.4引入

使用record_xml_attribute可以在测试用例中增加额外的xml属性。该固件也可以用来复写已经存在的属性值：

```
def test_function(record_xml_attribute):
    record_xml_attribute("assertions", "REQ-1234")
    record_xml_attribute("classname", "custom_classname")
    print("hello world")
    assert True
```

与record_property不同，该固件不会新增子节点，而是在testcase的tag里面增加或者覆盖已有的属性：

```
<testcase classname="custom_classname" file="test_function.py" line="0"
name="test_function" time="0.003" assertions="REQ-1234">
    <system-out>
        hello world
    </system-out>
</testcase>
```

注意： record_xml_attribute尚处于实验阶段，将来可能会有所变化。(还有一大串注释，懒得翻译了)

2.13.3 LogXML:add_global_property

省略

2.14 创建resultlog格式文件

很少用，很快就要被移除了

2.15 将测试结果发送给在线的pastebin

Pastebin是一个用户存储纯文本的web应用。用户可以通过互联网分享文本片段，一般都是源代码。(没用过这个，不翻了 > <)

2.16 禁用插件

如果想要在运行时禁用特定的插件，使用 -p以及**no:**前缀符。

如： 禁止加载doctest插件：

```
pytest -p no:doctest
```

2.17 在python代码中运行pytest

2.0引入 你可以在python代码中直接调用pytest:

```
pytest.main()
```

这和在命令行中使用pytest是一样的， 这种方式不会抛出SystemExit异常，而会返回exitcode， 通过如下方式可以传入调用参数：

```
pytest.main(['-x', 'mytestdir'])
```

你可以在pytest.main中指定额外的插件：

```
# myinvoke.py的内容
import pytest
class MyPlugin(object):
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")

pytest.main(["-qq"], plugins=[MyPlugin()])
```

运行代码就可以发现MyPlugin被添加到hook里并被调用：

```
C:\>python myinvoke.py
.FEsxX
[100%]*** test run reporting finishing

========================= ERRORS =========================
_____________________ ERROR at setup of test_error ______________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E    assert 0

test_example.py:5: AssertionError
========================= FAILURES =========================
_____________________ test_fail ______________________

    def test_fail():
>       assert 0
E    assert 0

test_example.py:11: AssertionError
```

注意:调用pytest.main()会导入你的测试用例及其所引用的所有的模块。因为python存在模块导入的缓存机制，如果多次调用pytest.main()，后续的调用也不会再刷新这些导入的资源。因此，不建议再同一进程中多次调用pytest.main() (比如重新运行测试).

Chapter 3 在现有测试套中使用pytest

pytest可以与大多数现有的测试套一起使用，但他的测试行为与其他的测试工具(如nose或者python的默认的unittest)有差异. 在使用此部分之前，您需要安装pytest

3.1 与现有的测试套一起运行pytest

比如说你想要修改某处的已有的代码库，在将代码拉到你的开发环境后并且设置好python的环境后，你需要在你的工程的根目录下运行：

```
cd <repository>
pip install -e . #解决环境依赖可以通过"python setup.py develop"或"conda develop"（都没用过。。。囧）
```

这将在site-packages中设置一个symlink到你的代码，允许你在运行测试的时候编辑代码，就好像你已经安装了这个package一样。（这段看得我莫名其妙，想了一下，作者的本意应该是，如果想对某个类似github的库进行测试，但是测试代码跟库代码又不好放一起的话，就用这个命令行搞一下）

设置项目为开发模式，可以避免每次运行测试时都需要重新install。

当然，也可以考虑使用tox库。

Chapter 4 在测试用例中编写和上报断言

4.1 使用断言语句

pytest允许你在测试用例中使用标准的python断言，如下：

```
# test_assert1.py中的内容
def f():
    return 3

def test_function():
    assert f() == 4
```

本例中的函数期望返回一个固定的值。如果该断言失败了，你会看到该函数的返回值：

```
$ pytest test_assert1.py
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/david/Downloads/myPython/translate, inifile:
collected 1 item

test_assert1.py F                                                        [100%]

=================================== FAILURES ===================================
_____________________________ test_function _____________________________

    def test_function():
```

```
>       assert f() == 4
E       assert 3 == 4
E        +  where 3 = f()

test_assert1.py:5: AssertionError
========================== 1 failed in 0.07 seconds ==========================
```

pytest支持显示常见的子表达式的值，包括调用，属性，比较以及二元和一元运算符。(参看Demo of Python failure reports with purest 这允许你使用你习惯的python的在不丢失内省信息的情况下构造代码。（什么是内省信息？更详细的内部输出信息？） 如果你为断言指定了输出信息，那么不会输出任何内省信息，而是在traceback中直接输出指定的信息：

```
assert a % 2 ==0, "value was odd, should be even"
```

更多断言内省信息请参考Advanced assertion introspection

4.2 异常的断言

你可以在上下文中使用pytest.raises来对异常产生断言：

```
import pytest

 def test_zero_division():
    with pytest.raise(ZeroDivisionError):
        1 / 0
```

如果你还需要访问异常的确切信息，你需要使用下面的方式：

```
def test_recursion_depth():
    with pytest.raises(RuntimeError) as excinfo:
        def f():
            f()
        f()
    assert 'Maximum recursion' in str(excinfo.value)
```

excinfo是ExceptionInfo的一个实例，里面包含了异常的详细信息，主要属性是.type, .value以及.traceback. 你可以通过match参数来使用正则表达式来匹配一个异常是否发生

```
import pytest
def myfunc():
    raise ValueError("Exception 123 raised")

def test_match():
    with pytest.raises(ValueError, match=r'.* 123 .*'):
        myfunc()
```

参数match与re.search行为是一致的，所以上面的match='123’ 能够正确的匹配到myfunc抛出的异常。

pytest.raises的另一种使用方法是传递一个函数给它，该函数使用给定的参数，并且最终判断该函数是否产生了期望的断言

```
pytest.raises(ExpectedException, func, *args, **kwargs)
```

测试报告会提供你一些帮助信息，比如是没有异常还是抛出了错误的异常

注意也可以在pytest.mark.xfail中使用raises来检查测试用例是否是因为抛出了异常而失败：

```
@pytest.mark.xfail(raises=IndexError)
def test_f():
    f()
```

使用pytest.raises更适合测试自己的代码故意引发的异常。

而使用@pytest.mark.xfail和检查函数更适合那些记录的未修复的bug(也就是那些应该发生的异常)或者依赖项中的异常。 （这两个场景在测试过程中是不一样的）

4.3 告警的断言

2.8 引入

你可以通过pytest.warns来判断是否产生了特定的告警

4.4 上下文比较

2.0 引入

pytest支持在进行比较时提供上下文的相关敏感信息

```
# test_assert2.py
def test_set_comparison():
    set1 = set("1308")
    set2 = set("8035")
    assert set1 == set2
```

运行如下：

```
$ pytest test_assert2.py
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/david/Downloads/myPython/translate, inifile:
collected 1 item
```

```
test_assert2.py F                                                    [100%]

================================= FAILURES =================================
_____________________________ test_set_comparison _____________________________

    def test_set_comparison():
        set1 = set("1308")
        set2 = set("8035")
>       assert set1 == set2
E       AssertionError: assert {'0', '1', '3', '8'} == {'0', '3', '5', '8'}
E         Extra items in the left set:
E         '1'
E         Extra items in the right set:
E         '5'
E         Use -v to get the full diff

test_assert2.py:4: AssertionError
========================== 1 failed in 0.07 seconds ==========================
```

4.5 为断言增加注释

你可以通过实现钩子函数pytest_assertrepr_compare来为断言增加自己的描述：

```
pytest_assertrepr_compare(config, op, left, right)
```

该钩子返回失败的断言中的注释。

如果没有自定义注释，返回None，否则返回一个字符串列表。列表中的字符串使用换行符来连接，同时字符串中的所有换行符都会被转义。

注意，除了第一行以外，其他所有行都会进行缩进，目的是将第一行作为摘要

```
Parameters config(_pytest.config.Config) - pytest config object
```

下面的示例在conftest.py中实现了该钩子函数，改变了Foo的对象的注释：

```
# conftest.py
from test_foocompare import Foo
def pytest_assertrepr_compare(op, left, right):
    if isinstance(left, Foo) and isinstance(right, Foo) and op == "==":
        return ['Comparing Foo instance:', ' vals: %s != %s' % (left.val, right.val)]
```

定义测试对象：

```
# test_foocompare.py
class Foo(object):
    def __init__(self, val):
        self.val = val
```

```
    def __eq__(self, other):
        return self.val == other.val

def test_compare():
    f1 = Foo(1)
    f2 = Foo(2)
    assert f1 == f2
```

运行这个测试，可以在测试结果中看到自定义的注释：

```
$ pytest -q test_foocompare.py
F [100%]
================================= FAILURES =================================
_______________________________ test_compare _______________________________
def test_compare():
f1 = Foo(1)
f2 = Foo(2)
> assert f1 == f2
E assert Comparing Foo instances:
E vals: 1 != 2
test_foocompare.py:11: AssertionError
1 failed in 0.12 seconds
```

4.6 高级断言内省

偷懒没翻。。。

Chapter 5 pytest fixture:直接，模块化，易扩展(总之就是Niublity)

2.0/2,3/2.4有更新

测试fixture的目的是提供一个测试的基线，在此基线基础上，可以更可靠的进行重复测试。Pytest的fixture相对于传统的xUnit的setup/teardown函数做了显著的改进:

- 测试fixture有明确的名称，通过在函数/模块/类或者整个项目中激活来使用
- 测试fixture是模块化的实现，使用fixture名即可触发特定的fixture，fixture可以在其他fixture中进行使用
- 测试fixture不仅可以进行简单的单元测试，也可以进行复杂的功能测试。可以根据配置和组件的选项进行参数化定制测试，或者跨函数/类/模块或者整个测试过程进行测试。

此外，pytest依然支持经典的xUnit的样式，你可以根据自己的喜好混合两种样式，甚至可以基于现有的unittest.TestCase或者nose的样式来开发

5.1 作为函数入参的fixture

测试函数可以通过接受一个已经命名的fixture对象来使用他们。对于每个参数名，如果fixture已经声明定义，会自动创建一个实例并传入该测试函数。fixture函数通过装饰器标志@pytest.fixture来注册。下面是一个简单的独立的测试模块，包含一个fixture及使用它的测试函数

```
# ./test_smtpsimple.py
import pytest

@pytest.fixture
def smtp_connection():
    import smtplib
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert 0 # 用于调试
```

这里，test_ehlo需要smtp_connection这个fixture的返回。pytest会在@pytest.fixture的fixture中查找并调用名为smtp_connection的fixture。运行这个测试结果如下：

```
$ pytest test_smtpsimple.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item
test_smtpsimple.py F [100%]
================================= FAILURES =================================
________________________________ test_ehlo _________________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_ehlo(smtp_connection):
response, msg = smtp_connection.ehlo()
assert response == 250
> assert 0 # for demo purposes
E assert 0
test_smtpsimple.py:11: AssertionError
========================== 1 failed in 0.12 seconds ==========================
```

测试的回显中可以看出测试函数调用了smtp_connection,这是由fixture函数创建的smtplib.SMTP()的一个实例。该函数在我们故意添加的assert 0处失败。以下是pytest的在调用该函数的时候的详细规则：

- pytest找到以test_作为前缀的测试用例test_ehlo。该测试函数有一个名为smtp_connection的入参。而在fixture函数中存在一个名为smtp_connection的fixture。
- smtp_connection()被调用来创建一个实例。
- test_ehlo(<smtp_connection实例>)被调用并在最后一行因为断言失败。注意，如果拼错了函数参数，或者使用了一个不可用的参数，你会看到一个包含可用函数参数列表的错误信息。

注意：你可以使用 pytest --fixtures test_simplefactory.py 来查看可用的fixture(如果想查看以_开头的fixture，请添加-v参数)

5.2 fixture：依赖注入的最佳实践

pytest的fixture允许测试函数轻松的接收和处理应用层对象的预处理，而不必关心import/setup/cleanup这些细节。 这是依赖注入的的一个极佳的示范，fixture函数是注入器，而测试函数是fixture的使用者

5.3 conftest.py: 共享fixture函数

实现测试用例的过程中，当你发现需要使用来自多个文件的fixture函数的时候，可以将这些fixture函数放到conftest.py中。

你不需要导入这些fixture函数，它会由pytest自动检索。

fixture函数的检索顺序是从测试类开始，然后测试的模块，然后就是conftest.py文件，最后是内置的插件和第三方插件。

你还可以使用conftest.py来为本地每个目录实现插件

5.4 共享测试数据

如果你要在测试中通过文件来使用测试数据，一个好的方式是通过fixture来加载这些数据后使用，这样就可以利用了pytest的自动缓存的机制。

另一个好的方式是将这些数据文件添加到测试文件夹中，这样可以用插件来管理这些测试数据，不如：pytest-datadir和pytest-datafiles

5.5 scope：在类/模块/整个测试中共享fixture实例

当fixture需要访问网络时，因为依赖于网络状况，通常是一个非常耗时的动作

扩展下上面的示例，我们可以将scope="module"参数添加到@pytest.fixture中，这样每个测试模块就只会调用一次smtp_connection的fixture函数(默认情况下时每个测试函数都调用一次)。因此，一个测试模块中的多个测试函数将使用同样的smtp_connection实例，从而节省了反复创建的时间。scope可能的值为：function, class, module, package 和 session

下面的示例将fixture函数放在独立的conftest.py中，这样可以在多个测试模块中访问使用该测试fixture：

```
# conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
```

```
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

fixture的名称依然为smtp_connection,你可以在任意的测试用例中通过该名称来调用该fixture(在conftest.py所在的目录及子目录下)

```
# test_module.py

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert b"smtp.gmail.com" in msg
    assert 0 # for debug

def test_noop(smtp_connection):
    response, msg = smtp_connection.noop()
    assert response == 250
    assert 0 # for debug
```

我们故意添加了assert 0的断言来查看测试用例的运行情况：

```
$ pytest test_module.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items
test_module.py FF [100%]
================================ FAILURES ================================
_____________________________ test_ehlo _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_ehlo(smtp_connection):
response, msg = smtp_connection.ehlo()
assert response == 250
assert b"smtp.gmail.com" in msg
> assert 0 # for demo purposes
E assert 0
test_module.py:6: AssertionError
_____________________________ test_noop _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_noop(smtp_connection):
response, msg = smtp_connection.noop()
assert response == 250
> assert 0 # for demo purposes
E assert 0
test_module.py:11: AssertionError
======================== 2 failed in 0.12 seconds ========================
```

可以看到这两个用例都失败了，并且你可以在traceback中看到smtp_connection被传进了这两个测试函数中。这两个函数复用了同一个smtp_connection实例

如果你需要一个session作用域的smtp_connection实例，你可以按照如下来定义：

```
@pytest.fixture(scope="session")
def smtp_connection():
    #该固件会在所有的用例中共享
```

scope定义为class的话会创建一个在每个class中调用一次的fixture

注意： Pytest对于每个fixture只会缓存一个实例，这意味着如果使用参数化的fixture，pytest可能会比定义的作用域更多次的调用fixture函数（因为需要创建不同参数的fixture）

5.5.1 package scope(实验阶段)

既然实验阶段，那就不翻了。。。囧~~~

5.6 scope越大，实例化越早

3.5 引入

当函数调用多个fixtures的时候，scope较大的(比如session)实例化早于scope较小的(比如function或者class)。同样scope的顺序则按照其在测试函数中定义的顺序及依赖关系来实例化。

考虑如下代码：

```
@pytest.fixture(scope="session")
def s1():
    pass

@pytest.fixture(scope="module")
def m1():
    pass

@pytest.fixture
def f1(tmpdir):
    pass

@pytest.fixture
def f2():
    pass

def test_foo(f1, m1, f2, s1):
    ...
```

该函数所请求的fixtures的实例化顺序如下：

- s1: 具有最大的scope(session)
- m1: 第二高的scope(module)
- tmpdir: f1需要使用该fixture，需要在f1之前实例化

- f1：在function级的scope的fixtures中，在test_foo中处于第一个
- f2：在function级的scope的fixtures中，在test_foo中处于最后一个

5.7 fixture的调用结束/执行清理代码

pytest支持在fixture退出作用域的时候执行相关的清理/结束代码。使用yield而不是return关键字的时候，yield后面的语句将会在fixture退出作用域的时候被调用来清理测试用例

```
# conftest.py

import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp_connection():
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp_connection
    print("teardown smtp")
    smtp_connection.close()
```

无论测试是否发生了异常，print及smtp.close()语句将在module的最后一个测试函数完成之后被执行

```
$ pytest -s -q --tb=no
FFteardown smtp
2 failed in 0.12 seconds
```

可以看到在两个测试函数完成后smtp_connection实例调用了相关的代码。注意如果我们定义scope为function级别(scope= ‘function’)，该部分代码会在每个测试函数结束后都会调用。测试函数本身并不需要关心fixture的实现的细节。

我们也可以在with语句中使用yield:

```
@pytest.fixture(scope="module")
def smtp_connection():
    with smtplib.SMTP("smtp.gmail.com", 587, timeout=5) as smtp_connection:
        yield smtp_connection
```

因为with语句结束,smtp_connection会在测试结束后被自动关闭. 注意如果在yield之前发生了异常，那么剩余的yield之后的代码则不会被执行。

另外一个处理teardown的代码的方式时使用addfinalizer函数来注册一个teardown的处理函数。 如下是一个例子：

```
# conftest.py
import smtplib
import pytest
```

```
@pytest.fixture(scope="module")
def smtp_connection(request):
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

    def fin():
        print("teardown smtp_connection")
        smtp_connection.close()
    request.addfinalizer(fin)
    return smtp_connection
```

yield和addfinalizer在测试结束之后的调用是基本类似的，addfinalizer主要有两点不同于yield：

- 可以注册多个完成函数
- 无论fixture的代码是否存在异常，addfinalizer注册的函数都会被调用，这样即使出现了异常，也可以正确的关闭那些在fixture中创建的资源

```
@pytest.fixture
def equipments(request):
    r = []
    for port in ('C1', 'C3', 'C28'):
        equip = connect(port)
        request.addfinalizer(equip.disconnect)
        r.append(equip)
    return r
```

该示例中，如果"C28"因为异常失败了，"C1"和"C3"也会被正确的关闭。当然，如果在addfinalizer调用注册前就发生了异常，这个注册的函数就不会被执行了

5.8 Fixtures可以获取测试对象的上下文

fixture函数可以通过接收一个request对象来获取"请求"的测试函数/类/模块的上下文。 使用前面的smtp_connection做为扩展示例，我们在测试模块中来使用我们的fixture读取一个可选的服务器URL：

```
# conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp_connection(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp_connection = smtplib.SMTP(server, 587, timeout=5)
    yield smtp_connection
    print("finalizing %s (%s)" % (smtp_connection, server))
    smtp_connection.close()
```

我们使用request.module属性从测试模块中选择获取smtpserver的属性。即使我们再执行一次，也不会有什么变化：

```
$ pytest -s -q --tb=no
FFfinalizing <smtplib.SMTP object at 0xdeadbeef> (smtp.gmail.com)
2 failed in 0.12 seconds
```

我们来快速的创建一个在其中设置了服务器的URL的测试模块：

```
# test_anothersmtp.py
smtpserver = "mail.python.org" #会被smtp fixture读取

def test_showhelo(smtp_connection):
    assert 0, smtp_connection.helo()
```

运行结果如下：

```
$ pytest -qq --tb=short test_anothersmtp.py
F [100%]
================================ FAILURES ================================
_____________________________ test_showhelo _____________________________
test_anothersmtp.py:5: in test_showhelo
assert 0, smtp_connection.helo()
E AssertionError: (250, b'mail.python.org')
E assert 0
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)
```

瞧，smtp_connection的实现函数使用了我们在模块中定义的新的server名

5.9 工厂化的fixtures

工厂化的fixture的模式对于一个fixture在单一的测试中需要被多次调用非常有用。fixture用一个生成数据的函数取代了原有的直接返回数据。该函数可以在测试中被多次调用。

如果需要，工厂也可以携带参数：

```
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {
            "name": name,
            "orders": []
        }
    return _make_customer_record

def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

如果需要管理工厂创建的数据，可以按照如下来处理fixture：

```
@pytest.fixture
def make_customer_record():
    create_records = []
    def _make_customer_record(name):
        record = models.Customer(name=name, orders=[])
        created_records.append(record)
        return record

    yield _make_customer_record

    for record in created_records:
        record.destroy()

def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

5.10 fixtures参数化

fixture函数可以进行参数化的调用，这种情况下，相关测试集会被多次调用，即依赖该fixture的测试的集合。测试函数通常无需关注这种重复测试

fixture的参数化有助于为那些可以以多种方式配置的组件编写详尽的功能测试

扩展之前的示例，我们标记fixture来创建两个smtp_connection的实例，这会使得所有的测试使用这两个不同的fixture运行两次：

```
# conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module", params=["smtp.gmail.com", "mail.python.org"])
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
        yield smtp_connection
        print("finalizing %s" % smtp_connection)
        smtp_connection.close()
```

相对于之前的代码，这里主要的改动就是为@pytest.fixture定义了一个params，params是一个可以通过request.params在fixture中进行访问的列表。无需修改其他的代码，让我们来运行它：

```
$ pytest -q test_module.py
FFFF [100%]
================================= FAILURES =================================
____________________________ test_ehlo[smtp.gmail.com] _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_ehlo(smtp_connection):
response, msg = smtp_connection.ehlo()
```

```
assert response == 250
assert b"smtp.gmail.com" in msg
> assert 0 # for demo purposes
E assert 0
test_module.py:6: AssertionError
_____________________________ test_noop[smtp.gmail.com] _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_noop(smtp_connection):
response, msg = smtp_connection.noop()
assert response == 250
> assert 0 # for demo purposes
E assert 0
test_module.py:11: AssertionError
_____________________________ test_ehlo[mail.python.org] _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_ehlo(smtp_connection):
response, msg = smtp_connection.ehlo()
assert response == 250
> assert b"smtp.gmail.com" in msg
E AssertionError: assert b'smtp.gmail.com' in b'mail.python.
‹→org\nPIPELINING\nSIZE 51200000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-
‹→MD5\nENHANCEDSTATUSCODES\n8BITMIME\nDSN\nSMTPUTF8\nCHUNKING'
test_module.py:5: AssertionError
---------------------------- Captured stdout setup -----------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
_____________________________ test_noop[mail.python.org] _____________________________
smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
def test_noop(smtp_connection):
response, msg = smtp_connection.noop()
assert response == 250
> assert 0 # for demo purposes
E assert 0
test_module.py:11: AssertionError
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
4 failed in 0.12 seconds
```

可以看到每个测试函数都是用不同的smtp_connection实例运行了两次。同时注意，mail.python.org这个连接在第二个测试中的test_ehlo因为一个不同的服务器字符串的检查而失败了

pytest会为每个fixture的参数值创建一个测试ID字符串，比如上面的例子中：test_ehlo[smtp.gmail.com]和test_ehlo[mail.python.org]。可以使用-k来通过这些ID选择特定的测试用例运行，也可以在失败的时候定位到具体的用例。运行pytest的时候带–collect-only可以显示这些生成的IDs

数字，字符串，布尔和None类型在测试ID中会保留他们自己的字符串的表示方式，其他的数据对象，pytest会创建一个基于参数名的字符串。可以通过ids关键字来自定义一个字符串来表示测试ID：

```
# test_ids.py
import pytest

@pytest.fixture(params=[0, 1], ids=["spam", "ham"])
def a(request):
```

```
    return request.param

def test_a(a):
    pass

def idfn(fixture_value):
    if fixture_value == 0:
        return "eggs"
    else:
        return None

@pytest.fixture(params=[0, 1], ids=idfn)
def b(request):
    return request.param

def test_b(b):
    pass
```

上面的示例展示了ids是如何使用一个定义的字符串列表的。后续的case展示了如果函数返回了None那么pytest会自动生成一个ID。 上面的示例结果如下：

```
$ pytest --collect-only
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 10 items
<Module test_anothersmtp.py>
<Function test_showhelo[smtp.gmail.com]>
<Function test_showhelo[mail.python.org]>
<Module test_ids.py>
<Function test_a[spam]>
<Function test_a[ham]>
<Function test_b[eggs]>
<Function test_b[1]>
<Module test_module.py>
<Function test_ehlo[smtp.gmail.com]>
<Function test_noop[smtp.gmail.com]>
<Function test_ehlo[mail.python.org]>
<Function test_noop[mail.python.org]>
======================== no tests ran in 0.12 seconds ========================
```

5.11 在参数化的fixture中使用marks

pytest.param()可以用来用来接收通过marks参数传入的标志，就像使用@pytest.mark.parametrize。 如下：

```
# test_fixture_marks.py
import pytest
@pytest.fixture(params=[0, 1, pytest.param(2, marks=pytest.mark.skip)])
def data_set(request):
   return request.param
```

```
def test_data(data_set):
   pass
```

运行该测试会跳过data_set中值为2的调用：

```
$ pytest test_fixture_marks.py -v
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 3 items
test_fixture_marks.py::test_data[0] PASSED [ 33%]
test_fixture_marks.py::test_data[1] PASSED [ 66%]
test_fixture_marks.py::test_data[2] SKIPPED [100%]
==================== 2 passed, 1 skipped in 0.12 seconds ====================
```

5.12 模块化： 通过fixture函数使用fixture

不仅测试函数可以使用fixture，fixture函数本身也可以使用其他的fixture。这可以使得fixture的设计更容易模块化，并可以在多个项目中复用fixture

扩展前面的例子作为一个简单的范例，我们在一个已经定义的smtp_connection中插入一个实例化的APP对象：

```
# test_appsetup.py

import pytest

class App(object):
    def __init__(self, smtp_connection):
        self.smtp_connection = smtp_connection

@pytest.fixture(scope="module")
def app(smtp_connection):
    return App(smtp_connection)

def test_smtp_connection_exists(app):
    assert app.smtp_connection
```

这里我们定义了一个名为app的fixture并且接收之前定义的smtp_connection的fixture，在其中实例化了一个App对象。运行结果如下：

```
$ pytest -v test_appsetup.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
```

```
collecting ... collected 2 items
test_appsetup.py::test_smtp_connection_exists[smtp.gmail.com] PASSED [ 50%]
test_appsetup.py::test_smtp_connection_exists[mail.python.org] PASSED [100%]
========================= 2 passed in 0.12 seconds =========================
```

因为对smtp_connection做了参数化，测试用例将会使用两个不同的App实例分别运行来连接各自的smtp服务器。App fixture无需关心smtp_connection的参数化，pytest会自动的分析其中的依赖关系

注意，app fixture声明了作用域是module，并使用了同样是module作用域的smtp_connection。如果smtp_connection是session的作用域，这个示例依然是有效的：fixture可以引用作用域更广泛的fixture，但是反过来不行，比如session作用域的fixture不能引用一个module作用域的fixture

5.13 fixture的自动分组

测试过程中，pytest会保持激活的fixture的数目是最少的 。 如果有一个参数化的fixture，那么所有使用它的测试用例会首先使用其一个实例来执行，直到它完成后才会去调用下一个实例。 这样做使得应用程序的测试中创建和使用全局状态更为简单(Why?)。

下面的示例使用了两个参数化的fixtures，其中一个是module作用域的，所有的函数多增加了打印用来展示函数的setup/teardown的过程:

```
# test_module.py
import pytest

@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print("  SETUP modarg %s" % param)
    yield param
    print("  TEARDOWN modarg %s" % param)

@pytest.fixture(scope="function", params=[1, 2])
def otherarg(request):
    param = request.param
    print("  SETUP otherarg %s" % param)
    yield param
    print("  TEARDOWN otherarg %s" % param)

def test_0(otherarg):
    print("  RUN test0 with otherarg %s" % otherarg)
def test_1(modarg):
    print("  RUN test1 with modarg %s" % modarg)
def test_2(otherarg, modarg):
    print("  RUN test2 with otherarg %s and modarg %s" % (otherarg, modarg))
```

运行结果如下:

```
$ pytest -v -s test_module.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y --
```

```
$PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:

collected 8 items

test_module.py::test_0[1]   SETUP otherarg 1
  RUN test0 with otherarg 1
PASSED  TEARDOWN otherarg 1

test_module.py::test_0[2]   SETUP otherarg 2
  RUN test0 with otherarg 2
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod1]   SETUP modarg mod1
  RUN test1 with modarg mod1
PASSED
test_module.py::test_2[mod1-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod1-2]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod1
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod2]   TEARDOWN modarg mod1
  SETUP modarg mod2
  RUN test1 with modarg mod2
PASSED
test_module.py::test_2[mod2-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod2-2]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod2
PASSED  TEARDOWN otherarg 2
  TEARDOWN modarg mod2


========================== 8 passed in 0.03 seconds ==========================
```

可以看到参数化的modarg使得测试的执行顺序保持这最少的激活fixtures的状态。mod1在mod2被创建前就已经被释放了。

特别需要注意的是，test_0首先执行完成，然后是test_1使用mod1执行，然后test_2使用mod1执行，然后是test_1使用mod2，最后是test_2使用mod2. function作用域的fixture：otherarg在每个测试函数前被创建，每次函数结束后释放。

注：有兴趣的话可以将function的otherarg改成module，测试函数的执行顺序会发生变化。

5.14 在classes/modules或者项目中使用fixtures

有时测试函数不需要直接使用fixture对象。比如测试用例可能需要操作一个空文件夹但是并不关心具体是哪个文件夹，这里就可以使用标准库的tmpfile来实现。我们将该fixture的实现放在conftest.py

中：

```
# conftest.py

    import pytest
    import tempfile
    import os

    @pytest.fixture()
    def cleandir():
        newpath = tempfile.mkdtemp()
        os.chdir(newpath)
```

通过usefixtures标记来使用它：

```
# test_setenv.py

import os
import pytest

@pytest.mark.usefixtures("cleandir")
class TestDirectoryInit(object):
    def test_cwd_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
        with open("myfile", "w") as f:
            f.write("hello")

    def test_cwd_again_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
```

因为使用了usefixtures，所以cleandir会在每个测试用例之前被调用，就好像为这些测试指定了"cleandir"入参一样。 如下是运行测试的结果：

```
$ pytest -q
.. [100%]
2 passed in 0.12 seconds
```

可以同时指定多个fixtures：

```
@pytest.mark.usefixtures("cleandir", "anotherfixture")
def test():
    ...
```

可以使用mark的通用特性来为测试module指定fixture：

```
pytestmark = pytest.mark.usefixtures("cleandir")
```

注意这里的变量只能命名为pytestmark，如果命名为其他变量(比如foomark)不会工作。

也可以将fixture写在ini文件中来在整个测试用例中使用fixture：

```
# pytest.ini
[pytest]
usefixtures = cleandir
```

警告 该标记不会对fixture函数生效，比如下面的代码不会按照你所想的调用my_other_fixture

```
@pytest.mark.usefixtures("my_other_fixture")
@pytest.fixture
def my_fixture_that_sadly_wont_use_my_other_fixture():
    ...
```

5.15 自动使用fixtures((嗑药一般的)飞一般的xUnit？)

有时可能希望在不用显式声明函数参数或者使用usefixtures装饰器的情况下自动调用相关的fixtures。作为一个实际生产中可能碰到的案例，假设我们有一个数据库的fixture，包含begin/rollback/commit等操作，并且我们希望在每个测试步骤前后分别调用数据库的事务处理和rollback操作。

下面是这个案例的一个实现：

```
# test_db_transact.py

import pytest

class DB(object):
    def __init__(self):
        self.intransaction = []
    def begin(self, name):
        self.intransaction.append(name)
    def rollback(self):
        self.intransaction.pop()

@pytest.fixture(scope="module")
def db():
    return DB()

class TestClass(object):
    @pytest.fixture(autouse=True)
    def transact(self, request, db):
        db.begin(request.function.__name__)
        yield
        db.rollback()

    def test_method1(self, db):
        assert db.intransaction == ["test_method1"]

    def test_method2(self, db):
        assert db.intransaction == ["test_method2"]
```

在class里定义的fixture transact标记了autouse为True，表示这个class里的所有测试函数无需任何其他声明就会自动的使用transact这个fixture。 运行这个测试，我们可以得到两个pass的测试结果：

```
$ pytest -q
.. [100%]
2 passed in 0.12 seconds
```

autouse的fixture遵循以下规则：

- autouse fixture遵守scope的定义，如果autouse fixture的scope为"session"，那么这个fixture无论定义在哪儿都只会运行一次，定义为"class"则表示在每个class中只会运行一次。
- 如果在module中定义了autouse，那么该module中的所有测试用例都会自动使用该fixture
- 如果在conftest.py中定义了autouse，那么该目录下的所有测试用例都会自动使用该fixture
- 最后，请谨慎使用该功能，如果你在插件中定义了一个autouse的fixture，那么所有使用了该插件的测试用例都会自动调用该fixture。这种方式在某些情况下是有用的，比如用ini文件配置fixture，这种全局的fixture应该快速有效的确定它应该完成哪些工作，避免代价高昂的导入和计算操作。

注意，上面的transact的fixture很可能只是你希望在项目中提供的一个fixture，而不是想要在每个测试用例中激活使用的。实现这一点的方式是将这个fixture移到conftest.py中并不要定义autouse:

```
# conftest.py
@pytest.fixture
def transact(request, db):
    db.begin()
    yield
    db.rollback()
```

在class中定义如何使用这个fixture：

```
@pytest.mark.usefixtures("transact")
class TestClass(object):
    def test_method1(self):
        ...
```

所有TestClass里的测试用例都会调用transact，而其他测试Class或测试函数则不会调用，除非他们也添加了transact这个fixture的引用。

5.16 重写fixtures

在大型的项目中，为了保持代码的可读性和可维护性，你可能需要重新在本地定义一个fixture来重写一个global或者root的fixture。

5.16.1 在文件夹(conftest)这一层重写

测试的文件结构如下：

```
tests/
    __init__.py
    conftest.py
        # tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'
    test_something.py
        # test/test_something.py
        def test_username(username):
            assert username == "username"

    subfolder/
        __init__.py

        conftest.py
            # tests/subfolder/conftest.py
            import pytest

            @pytest.fixture
            def username(username):
                return 'overridden-' + username

        test_something.py
            # tests/subfolder/test_something.py
            def test_username(username):
                assert username == 'overridden-username'
```

如上所示，fixture可以通过使用同样的函数名来进行重写。

5.16.2 在module这一层重写

文件结构如下：

```
tests/
    __init__.py

    conftest.py
        # tests/conftest.py
        @pytest.fixture
        def username():
            return 'username'
    test_something.py
        # tests/test_something.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-' + username
```

```
        def test_username(username):
            assert username == 'overridden-username'

    test_something_else.py
        # tests/test_something_else.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-else-' + username

        def test_username(username):
            assert username == 'overridden-else-username'
```

5.16.3 直接使用参数化的测试重写

文件结构如下：

```
tests/
    __init__.py

    conftest.py
        # tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

        @pytest.fixture
        def other_username(username):
            return 'other-' + username

    test_something.py
        # tests/test_something.py
        import pytest

        @pytest.mark.parametrize('username', ['directly-overridden-username'])
        def test_username(username):
            assert username == 'directly-overridden-username'

        @pytest.mark.parametrize('username', ['directly-overridden-username-other'])
        def test_username_other(other_username):
            assert other_username == 'other-directly-overridden-username-other'
```

上面的示例中，fixture的值被直接重写成了parametrize中定义的参数值。注意这种方式情况下，即使该测试用例不调用该fixture，这个fixture的值也是被重写过的（但是有什么用呢？）。

5.16.4 用参数化/非参数化的fixture重写非参数化/参数化的fixture

文件结构如下：

```
tests/
    __init__.py

    conftest.py
        # tests/conftest.py
        import pytest

        @pytest.fixture(params=['one', 'two', 'three'])
        def parametrized_username(request):
            return request.param

        @pytest.fixture
        def non_parametrized_username(request):
            return 'username'

    test_something.py
        # tests/test_something.py
        import pytest

        @pytest.fixture
        def parametrized_username():
            return 'overridden-username'

        @pytest.fixture(params=['one', 'two', 'three'])
        def non_parametrized_username(request):
            return request.param

        def test_username(parametrized_username):
            assert parametrized_username == 'overridden-username'

        def test_parametrized_username(non_parametrized_username):
            assert non_parametrized_username in ['one', 'two', 'three']

    test_something_else.py
        # tests/test_something_else.py

        def test_username(parametrized_username):
            assert parametrized_username in ['one', 'two', 'three']

        def test_username(non_parametrized_username):
            assert non_parametrized_username == 'username'
```

以上示例用参数化/非参数化的fixture重写了非参数化/参数化的fixture

Chapter 7 猴子补丁/模拟模块或环境的行为

有时我们需要修改函数的全局配置或者调用类似网络访问这些不容易测试的代码。monkeypatch可以用来安全的设置/删除一个属性，字典项或者环境变量，甚至可以改变import的路径sys.path. 参考monkeypatch blog post可以查看更多介绍信息及进展

7.1 MonkeyPatching函数

如果需要阻止os.expanduser返回特定值，你可以在调用该函数前使用monkeypatch.setattr()来修改该函数的返回

```
# test_module.py
import os.path
def getssh():
    return os.path.join(os.path.expanduser("~admin", '.ssh'))
def test_mytest(monkeypatch):
    def mockreturn(path):
        return '/abc'
    monkeypatch.setattr(os.path, 'expanduser', mockreturn)
    x = getssh()
    assert x == '/abc/.ssh'
```

这里对os.path.expanduser做了monkeypatch，然后调用的时候就会调用monkeypatch里定义的函数。当测试函数test_mytest结束，这个monkeypatch就会失效

7.2 禁止远程调用中"requests"

如果需要在所有测试用例中禁止http库中的"requests"，可以按照方案：

```
# conftest.py
import pytest
@pytest.fixture(autouse=True)
def no_requests(monkeypatch):
    monkeypatch.delattr("requests.sessions.Session.request")
```

该fixture被设置为了autouse，这样所有测试函数都会将request.sessions.Session.request删除而导致所有的http requests返回失败

注意，不建议对open/compile这些内置的函数做monkeypatch，这可能会导致ptest的内部异常。如果无法避免，使用–tb=native, --assert=plain以及–capture=no可能会有一些帮助，当然，不保证好使 _。

注意，对stdlib和一些第三方库使用monkeypatch可能会导致pytest的内部异常，因此推荐使用MonkeyPatch.context()来将作用域仅仅限制在你需要测试的代码块中

```
import functools
def test_partial(monkeypatch)
```

```
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
        assert functools.partial == 3
```

7.3 API

参考MonkeyPatch的类实现

Chapter 8 临时文件/文件夹

8.1 fixture: tmp_path

3.9更新

可以通过tmp_path来提供一个唯一的临时文件夹供函数调用。

tmp_path是pathlib/pathlib2.Path的一个对象，使用方法如下：

```
# test_tmp_path.py
import os
CONTENT = u"content"
def test_create_file(tmp_path):
    d = tmp_path / "sub"
    d.mkdir()
    p = d / "hello.txt"
    p.write_text(CONTENT)
    assert p.read_text() == CONTENT
    assert len(list(tmp_path.iterdir())) == 1
    assert 0
```

运行该测试可以发现除了最后用来观察结果的assert 0是失败的，其他的步骤都是pass的。

```
$ pytest test_tmp_path.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item
test_tmp_path.py F                                                  [100%]
================================ FAILURES ================================
_____________________________ test_create_file _____________________________
tmp_path = PosixPath('PYTEST_TMPDIR/test_create_file0')
    def test_create_file(tmp_path):
        d = tmp_path / "sub"
        d.mkdir()
        p = d / "hello.txt"
        p.write_text(CONTENT)
        assert p.read_text() == CONTENT
        assert len(list(tmp_path.iterdir())) == 1
```

```
>       assert 0
E       assert 0
test_tmp_path.py:13: AssertionError
========================= 1 failed in 0.12 seconds =========================
```

8.2 fixture: tmp_path_factory

3.9更新

tmp_path_factory是一个session作用域的fixture，可以在任意的fixture或者测试用例中来创建任何你想要的临时文件夹。

该fixture被用于取代tmpdir_factory，返回一个pathlib.Path的实例。

8.3 fixture: tmpdir

tmpdir可以为测试用例提供一个唯一的临时文件夹，这个文件夹位于base temporary directory (8.5). tmpdir是一个py.path.local对象，可以使用os.path的功能.示例如下：

```
# test_tmpdir.py
import os
def test_create_file(tmpdir):
    p = tmpdir.mkdir("sub").join("hello.txt")
    p.write("content")
    assert p.read() == "content"
    assert len(tmpdir.listdir()) == 1
    assert 0
```

运行该测试可以发现除了最后用来观察结果的assert 0是失败的，其他的步骤都是pass的。

```
$ pytest test_tmpdir.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item
test_tmpdir.py F                                                      [100%]
================================= FAILURES =================================
_____________________________ test_create_file _____________________________
tmpdir = local('PYTEST_TMPDIR/test_create_file0')
    def test_create_file(tmpdir):
        p = tmpdir.mkdir("sub").join("hello.txt")
        p.write("content")
        assert p.read() == "content"
        assert len(tmpdir.listdir()) == 1
>       assert 0
E       assert 0
test_tmpdir.py:7: AssertionError
========================= 1 failed in 0.12 seconds =========================
```

8.4 fixture: tmpdir_factory

2.8更新

tmp_factory是一个session作用域的fixture，可以在任意的fixture或者测试用例中来创建任何你想要的临时文件夹。

比如，假设你的测试套需要在磁盘上用程序生成一个很大的文件。你无需为每个测试用例都用tmpdir生成一遍，通过tmp_factory在整个session只生成一次即可：

```
# conftest.py
import pytest
@pytest.fixture(scope="session")
def image_file(tmpdir_factory):
    img = compute_expensive_image()
    fn = tmpdir_factory.mktemp("data").join("img.png")
    img.save(str(fn))
    return fn
# test_image.py
def test_histogram(image_file):
    img = load_image(image_file)
```

8.5 默认的临时文件夹

pytest创建的临时文件夹默认情况下是系统临时文件夹的一个子文件夹。一般会以pytest-开头，后面追加一串随测试运行增加的数字。同时，超过3个的临时目录会被删除。

可以通过下面的方式复写临时文件夹的配置：

```
pytest --basetemp=mydir
```

在本机进行分布式的测试时，pytest负责为各个子进程配置临时文件夹目录，保证每个进程的临时数据都落地在独立的临时文件夹中。

Chapter 9 捕获 stdout/stderr 输出

9.1 默认的stdout/stderr/stdin的捕获

测试过程中所有输出到stdout和stderr的信息都会被捕获，如果测试或者配置过程失败了，当前所有捕获的输出通常会和失败的traceback一起展示出来。（可以通过–show-capture命令行选项来进行配置）

另外，stdin被设置为null，所以不能从stdin进行读入，因为在运行自动化测试的时候，很少会要求有交互式的输入。 默认的捕获策略时截取写到底层文件描述符的数据，所以即使是子进程的测试输出也可以被捕获到！

9.2 设置捕获策略/禁止捕获

pytest执行捕获有两种方式：

- 默认的文件描述符级别的捕获：捕获所有写入操作系统文件描述符1和2的数据
- sys级别的捕获：仅捕获写入到python的文件系统sys.stdout和sys.stderr的数据。不会捕获写入到文件描述符的数据 你可以通过命令行的参数来选择捕获策略：

  pytest -s #禁止所有捕获 pytest --capture=sys #用内存文件取代sys.stdout/stderr pytest --capture=fd #将文件描述符指针1和2指向临时文件

9.3 利用print语句调试

pytest的默认的输出策略的好处是可以使用print语句来进行调试：

```
# test_module.py
def setup_function(function):
    print("setting up %s" % function)
def test_func1():
    assert Ture
def test_func2():
    assert False
```

运行该module，结果中精确的打印出来了失败函数，而另外一个通过的函数的输出则是隐藏的。

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items
test_module.py .F                                                    [100%]
================================= FAILURES =================================
_________________________________ test_func2 _________________________________
    def test_func2():
>       assert False
E       assert False
test_module.py:9: AssertionError
-------------------------- Captured stdout setup ---------------------------
setting up <function test_func2 at 0xdeadbeef>
==================== 1 failed, 1 passed in 0.12 seconds ====================
```

9.4 在测试函数中访问捕获的输出

capsys,capsysbinary, capfd和capfdbinary这些fixtures允许在测试运行期间访问stdout/stderr的标准输出。下面是一个示例：

```
def test_myoutput(capsys): # 也可以使用capfd
    print("hello")
    sys.stderr.write("world\n")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
    assert captured.err == "world\n"
    print("next")
    captured = capsys.readouterr()
    assert captured.out == "next\n"
```

readouterr()可以得到目前为止的输出快照，输出的捕获依然会继续。当测试完成后，输出stream会被重置。使用capsys的方式可以让你从必须关注如何配置/重置输出stream中解放出来。

capfd可以用在需要在文件描述符级别捕获输出的场景，调用的接口是相同的，但是允许从库或者子进程捕获那些直接写入操作系统的输出streams的输出(FD1和FD2). 还有一串更新历史。。。。lazy...

Chapter 10 捕获告警

3.1更新

从V3.1开始，pytest可以自动的捕获测试过程中的告警并在测试完成后显示出来

```
# test_show_warnings.py
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

def test_one()
    assert api_v1() == 1
```

运行结果如下：

```
$ pytest test_show_warnings.py
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_show_warnings.py .                                                  [100%]
============================= warnings summary ===============================
test_show_warnings.py::test_one
  $REGENDOC_TMPDIR/test_show_warnings.py:4: UserWarning: api v1, should use functions
from v2
    warnings.warn(UserWarning("api v1, should use functions from v2"))
```

```
-- Docs: https://docs.pytest.org/en/latest/warnings.html
=================== 1 passed, 1 warnings in 0.12 seconds ===================
```

-W参数可以用来控制是否显示告警，甚至可以控制是否将告警转换为Error：

```
$ pytest -q test_show_warnings.py -W error::UserWarning
F                                                                    [100%]
================================= FAILURES =================================
_____________________________________ test_one _____________________________________

    def test_one():
>       assert api_v1() == 1
test_show_warnings.py:8:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    def api_v1():
>       warnings.warn(UserWarning("api v1, should use functions from v2"))
E       UserWarning: api v1, should use functions from v2

test_show_warnings.py:4: UserWarning
1 failed in 0.12 seconds
```

在pytest.ini中使用filterwarnings也可以控制告警。比如，下面的配置会忽略所有的用户告警，但是会将其他的所有告警都转换成error：

```
[pytest]
filterwarnings =
    error
    ignore::UserWarning
```

当一个告警与多个配置匹配的话，那么放在最后的配置项生效。

-W参数和配置文件中的filterwarnings都是基于python原生的-W的选项和warnings.simplefilter，请参考python文档获取更多的示例和高级用法。

10.1 @pytest.mark.filterwarnings

3.2更新

可以使用@pytest.mark.filterwarnings来为一个特定的测试项添加告警筛选器，这可以让你更简单的控制哪些告警应该被捕获：

```
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

@pytest.mark.filterwarnings("ignore:api v1")
```

```
def test_one()
    assert api_v1() == 1
```

使用marker定义的筛选器的优先级高于命令行传递的参数以及在ini中的告警配置。

你可以将filterwarnings作为筛选器的一个mark，用于所有类的测试。或者通过设置pytestmark变量来将筛选器用于某个模块的所有测试：

```
# 将本模块中的所有告警转换成error
pytestmark = pytest.mark.filterwarnings("error")
```

10.2 禁止告警信息

虽然并不推荐，但是你依然可以使用–disable-warnings命令行参数去除测试输出中的所有的告警信息的显示。

10.3 禁止捕获告警

可以在pytest.ini中如下配置来完全禁止捕获告警：

```
[pytest]
addopts = -p no:warnings
```

或者在命令行中使用 -p no:warnings。这在你需要处理外部系统的告警时可能是会被用到的。（防止跟你自己的告警混淆？？）

10.4 废弃告警和将废弃告警

3.8引入，3.9更新

默认情况下pytest会显示DeprecationWarning以及PendingDeprecationWarning这两类来自用户的代码或者第三方库的告警。这可以帮助用户的代码保持整洁并且避免了当移除这些废弃内容时破坏了原有的代码。 有时你不想再显示那些无法控制的第三方的废弃告警，你可以在ini或者marks中使用告警筛选器来进行过滤，如下：

```
[pytest]
filterwarnings =
    ignore:.*U.*mode is deprecated:DeprecationWarning
```

这会忽略能够匹配所给的正则表达式的DeprecationWarning。

注意，如果告警是在python解释器层面定义的，使用环境变量PYTHONWARNINGS或者命令行参数-W，pytest则默认不会配置任何筛选器(没看懂。。。)

10.5 确保代码会触发废弃告警

你也可以通过调用一个全局的函数来检查是否确实有DeprecationWarning或者PendingDeprecationWarning：

```
import pytest

def test_global():
    pytest.deprecated_call(myfunction, 17)
```

默认情况下，DeprecationWarning和PendingDeprecationWarning不会被pytest.warns或者recwarn捕获，因为其内部有默认的Python告警筛选机制。如果你想要在你的代码中记录这些告警，使用warnings.simplefilter（‘always’）：

```
import warnings
import pytest

def test_deprecation(recwarn):
    warnings.simplefilter('always')
    warnings.warn("deprecated", DeprecationWarning)
    assert len(recwarn) == 1
    assert recwarn.pop(DeprecationWarning)
```

你也可用将其作为上下文管理器来使用：

```
def test_global():
    with pytest.deprecated_call():
        myobject.deprecated_method()
```

10.6 使用告警函数来检查告警

2.8引入

你可以使用pytest.warns来检查是否有特定的告警，用法类似于raises：

```
import warnings
import pytest

def test_warning():
    with pytest.warns(UserWarning):
        warnings.warn("my warning", UserWarning)
```

如果没有告警，那么测试就会失败。我们也可以使用match关键字来检查异常中是否包含特定的字符串：

```
>>> with warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)
>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)
>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
    ...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

也可以使用函数或者代码字符串来调用pytest.warns

```
pytest.warns(expected_warning, func, *args, **kwargs)
pytest.warns(expected_warning, "func(*args, **kwargs)")
```

该函数也会返回所有抛出的告警信息(warnings.WarningMessage对象)列表，你可以在其中检索你关心的额外信息：

```
with pytest.warns(RuntimeWarning) as record:
    warnings.warn("another warning", RuntimeWarning)

# 检查是否只有一个告警
assert len(record) == 1
# 检查告警信息是否匹配
assert record[0].message.args[0] == "another warning"
```

你也可以通过下面要讲的recwarn fixture来检查告警的详细信息。

注意：DeprecationWarning和PendingDeprecationWarning的处理并不一样，参考10.5

10.7 告警的记录

你可以通过pytest.warns或者fixture：recwarn来记录告警。

如果使用pytest.warns来记录告警并且不想产生任何断言，那么使用None作为参数传入：

```
with pytest.warns(None) as record:
    warnings.warn("user", UserWarning)
    warnings.warn("runtime", RuntimeWarning)

assert len(record) == 2
assert str(record[0].message == "user")
assert str(record[1].message == "runtime")
```

fixture: recwarn会将整个函数产生的告警都记录下来：

```
import warnings

def test_hello(recwarn):
    warnings.warn("hello", UserWarning)
    assert len(recwarn) == 1
    w = recwarn.pop(UserWarning)
    assert issubclass(w.category, UserWarning)
    assert str(w.message) == "hello"
    assert w.filename
    assert w.lineno
```

recwarn和pytest.warns返回的是同样的告警记录列表：一个WarningsRecorder的实例。你可以通过迭代器依次访问这个实例里的成员，使用len可以得到记录的告警的数量，也可以直接通过index来访问某一个具体的告警。

10.8 自定义失败信息

告警信息的结果处理使得在没有告警上报或其他需要的情况下，我们可以自定义一条测试失败的信息：

```
def test():
    with pytest.warns(Warning) as record:
        f()
        if not record:
            pytest.fail("Expected a warning")
```

当f()没有任何告警时，not record的值将是True,这时你可以通过调用pytest.fail来自定义一条错误信息。

10.9 pytest内部告警

3.8引入

pytest在某些情况下可能会产生自己的告警， 比如： 错误的用法或者废弃的特性的使用。

举个例子，如果一个类与python_classes匹配但是同时又定义了一个__init__构造函数，pytest会触发一个告警，因为这会导致class不能实例化（Why??）：

```
# test_pytest_warnings.py
class Test:
    def __init__(self):
        pass

    def test_foo(self):
        assert 1 == 1
```

```
$ pytest test_pytest_warnings.py -q
============================= warnings summary =============================
test_pytest_warnings.py:1
    $REGENDOC_TMPDIR/test_pytest_warnings.py:1: PytestWarning: cannot collect test class
'Test' because it has a __init__ constructor
        class Test:

-- Docs: https://docs.pytest.org/en/latest/warnings.html
1 warnings in 0.12 seconds
```

Chapter 11 这章不想搞，先不管了。。。

Chapter 12 Skip和xfail: 处理那些不会成功的测试用例

你可以对那些在某些特定平台上不能运行的测试用例或者你预期会失败的测试用例做一个标记，这样pytest在提供测试报告时可以做对应的处理以保持整个测试套的结果都是green的(一般都用绿色表示测试通过)

skip表示在满足某些情况下该测试用例是通过的，否则这个测试用例应该被跳过不执行。比较常见的例子是测试用例在windows平台下执行在非windows平台下不执行，或者比如数据库等外部资源不能访问时不执行某些测试用例。 xfail表示期望某个测试用例因为某些原因是失败的。一个常见的例子是一个新特性还没有实现或者bug还没有被修复。如果该测试用例已经被定义为pytest.mark.xfail但是又测试通过了，那么在最后的测试报告中会被标记为xpass。

pytest单独统计skip和xfail的测试用例，为了保持整洁，默认情况下测试报告中不会显示skipped/xfailed的测试用例的信息。你可以使用-r选项来查看相关的详细信息： pytest -rxXs # r:显示详细信息 x: xfailed, X: xpassed, s: skipped 你可以在pytest -h中查看-r的更多帮助。

12.1 跳过测试函数

2.9引入

最简单的方式就是使用skip装饰器:

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    ...
```

也可以在代码执行过程中直接调用pytest.skip(reason)来强制跳过:

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

在你不知道具体的skip的条件时，这种强制跳过的方法是很有用的。

还可以使用pytest.skip(reason, allow_module_level=True)来跳过整个module：

```
import sys
import pytest

if not sys.platform.startswith("win"):
    pytest.skip("skipping windows-only tests", allow_module_level=True)
```

12.1.1 skipif

2.0引入

你可以使用skipif来在某些条件下跳过测试。 下面是一个在检查python的版本是否高于3.6的示例：

```
import sys
@pytest.mark.skipif(sys.version_info < (3, 6), reason="require python3.6 or higher")
def test_function():
    ...
```

在查找用例的时候，如果判断skipif的条件是True，该用例会被跳过，如果使用-rs参数，详细的reason会在测试报告中体现

你可以在各个模块中共享skipif标记，比如有下面的模块定义：

```
# test_mymodule.py
import mymodule
minversion = pytest.mark.skipif(mymodule.__versioninfo__ < (1,1), reason="at least
mymodule-1.1 required")

@minversion
def test_function():
    ...
```

你可以在其他模块中import这个标记：

```
# test_myothermodule.py
from test_mymodule import minversion

@minversion
def test_anotherfunction():
    ...
```

在大型项目中，一般会将这些共享的标记放在同一个文件里供其他模块调用。

当然，你也可以使用条件字符串来代替布尔结果，但是这样因为一些兼容性的原因在模块间就不是很方便共享。

12.1.2 跳过模块或者class中的所有的测试

你可以在class上使用skipif标记：

```
@pytest.mark.skipif(sys.platform == "win32", reason="does not run on windows")
class TestPosixCalls(object):
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

如果满足条件，这个标记会跳过这个class下面的所有测试函数。

如果你要跳过模块中的所有测试，你需要使用全局的pytestmark：

```
# test_module.py
pytestmark = pytest.mark.skipif(...)
```

当一个测试函数有多个skipif装饰器时，任何一个装饰器满足条件时都会跳过该测试。

12.1.3 跳过文件或文件夹

有时你可能需要跳过一个完整的文件或者文件夹，比如依赖python特定版本的某些特性或者你不想pytest跑的测试代码。这种情况下你需要在collection阶段就将这些文件排除掉。 参考Chapter 27的Customizing test collection。

12.1.4 跳过因为import的依赖错误的用例

你可以在模块中或者测试用例中使用下面的方式：

```
docutils = pytest.importorskip("docutils")
```

如果docutils无法导入，这里就会跳过所有的测试，你可以基于某个库的版本号来进行判断：

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

这里的版本号是从模块的__version__属性里读出来的。

12.1.5 总结

无条件的跳过模块中的所有测试：

```
pytestmark = pytest.mark.skip("all tests still WIP")
```

有条件的跳过模块中的所有测试：

```
pytestmark = pytest.mark.skipif(sys.platform == "win32", "tests for linux only")
```

当import错误时，跳过模块中的所有测试：

```
pexpect = pytest.importorskip("pexpect")
```

12.2 XFail：标记测试用例是期望失败的

你可以使用xFail来标记你期望某个测试用例是失败的：

```
@pytest.mark.xfail
def test_function():
    ...
```

该测试用例会被正常执行，但是当它失败的时候不会有traceback。在测试报告中，该测试会被列举在“期望失败的用例”（XFAIL）或者“不应该通过的用例”（XPASS）里。

你也可以在测试中强制的指定一个测试用例为XFAIL：

```
def test_function():
    if not vaild_config():
        pytest.xfail("failing configuration (but should work)")
```

这会无条件的使得test_function的结果是XFAIL。注意与marker的方式不一样的是，这里pytest.xfail后不会再有代码被执行，因为这里会立刻在内部抛出一个异常。

12.2.1 strict参数

2.9引入

XFAIL和XPASS的测试用例都不会导致整个测试套失败，除非指定了strict参数为True。

```
@pytest.mark.xfail(strict=True)
def test_function():
    ...
```

这个参数会让XPASS的结果在测试报告中变成失败。 也可以在ini中通过xfail_strict来指定strict参数：

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

12.2.2 reason参数

如同skipif一样，你可以标记你在某个特定平台上期望测试是失败的。

```
@pytest.mark.xfail(sys.version_info >= (3, 6), reason="Python3.6 api changes")
def test_function():
    ...
```

12.2.3 raises参数

如果你想更清楚的表达测试为什么失败，你可以用raises参数指定一个或者一组异常：

```
@pytest.mark.xfail(raises=RuntimeError)
def test_function():
    ...
```

如果该测试函数并不是因为raises中指定的异常导致的失败，那么该测试函数在测试结果中就会被标记为正常的失败，如果是raises中的异常导致的失败，那么就会被标记为预期的失败。

12.2.4 run 参数

如果一个测试应该被标记为xfail并且预期是失败的，但是你又暂时不想运行这个测试用例，那么你可以将run参数设置为False：

```
@pytest.mark.xfail(run=False)
def test_function():
    ...
```

这个特性在xfail的case发生问题（比如crash）了，后面准备进行处理的时候有用处。

12.2.5 忽略xfail

在命令行中指定下面的参数：

```
pytest --runxfail
```

可以强制让xfail和pytest.xfail失效。

12.2.6 范例

下面是一个xfail的用法的例子：

```
import pytest

xfail = pytest.mark.xfail

@xfail
def test_hello():
    assert 0

@xfail(run=False)
def test_hello2():
    assert 0

@xfail("hasattr(os, 'sep')")
def test_hello3():
    assert 0

@xfail(reason="bug 110")
def test_hello4():
    assert 0

@xfail('pytest.__version__[0] != "17"')
def test_hello5():
    assert 0

def test_hello6():
    pytest.xfail("reason")

@xfail(raise=IndexError)
def test_hello7():
    x = []
    x[1] = 1
```

带参数report-on-xfail来运行这个示例：

```
example $ pytest -rx xfail_demo.py
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/example, inifile:
collected 7 items
xfail_demo.py xxxxxxx [100%]
========================= short test summary info ==========================
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
   reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
   condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
   bug 110
XFAIL xfail_demo.py::test_hello5
   condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
   reason: reason
```

```
XFAIL xfail_demo.py::test_hello7

======================== 7 xfailed in 0.12 seconds =========================
```

12.3 参数化Skip/xfail

在参数化中可以将skip/xfail作为一个独立的实例当作marker使用：

```
import pytest

@pytest.mark.parametrize(
    ("n", "expected"),
    [
        (1, 2),
        pytest.param(1, 0, marks=pytest.mark.xfail),
        pytest.param(1, 3, marks=pytest.mark.xfail(reason="some bug")),
        (2, 3),
        (3, 4),
        (4, 5),
        pytest.param(10 ,11, marks=pytest.mark.skipif(sys.version_info >= (3, 0),
reason="py2k")),
    ],
)

def test_increment(n, expected):
    assert n + 1 == expected
```

带rx参数运行结果如下：

```
$ pytest -rx test_example.py
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/david/Downloads/myPython/python3-venv, inifile:
collected 7 items

testType.py .xx...s                                                      [100%]
============================ short test summary info ===========================
XFAIL testType.py::test_increment[1-0]
XFAIL testType.py::test_increment[1-3]
  some bug

================ 4 passed, 1 skipped, 2 xfailed in 0.07 seconds ================
```

Chapter 13 参数化

pytest有如下几种参数化的方式：

- pytest.fixture()可以对测试函数进行参数化
- @pytest.mark.parametrize允许对测试函数或者测试类定义多个参数和fixtures的集合

- pytest_generate_tests允许自定义参数化的扩展功能

13.1 @pytest.mark.parametrize: 参数化测试函数

2.2引入，2.4改进

内置的pytest.mark.parametrize装饰器可以用来对测试函数进行参数化处理。下面是一个典型的范例，检查特定的输入所期望的输出是否匹配：

```
# test_expectation.py
import pytest
@pytest.mark.parametrize("test_input, expected", [("3+5", 8), ("2+4", 6), ("6*9", 42),])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

装饰器@parametrize定义了三组不同的(test_input, expected)数据，test_eval则会使用这三组数据执行三次：

```
$ pytest
========================== test session starts ==============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_expectation.py ..F                                              [100%]

=================================== FAILURES ================================
_____________________________ test_eval[6*9-42] _____________________________
test_input = '6*9', expected = 42
    @pytest.mark.parametrize("test_input,expected", [
        ("3+5", 8),
        ("2+4", 6),
        ("6*9", 42),
    ])
    def test_eval(test_input, expected):
>       assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E       + where 54 = eval('6*9')

test_expectation.py:8: AssertionError
==================== 1 failed, 2 passed in 0.12 seconds =====================
```

该示例中，只有一组数据是失败的。通常情况下你可以在traceback中看到作为函数参数的input和output。

注意你也可以对模块或者class使用参数化的marker来让多个测试函数在不同的测试集下运行。

你也可以对参数集中的某个参数使用mark，比如下面使用了内置的mark.xfail：

```
# test_exception.py
import pytest
@pytest.mark.parametrize("test_input, expected", [("3+5", 8), ("2+4", 6), ("6*9", 42,
marks=pytest.mark.xfail),])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

运行结果如下：

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_expectation.py ..x                                          [100%]

=================== 2 passed, 1 xfailed in 0.12 seconds ===================
```

之前结果是失败的用例在这里已经被标记为xfailed了。

如果参数化的列表是一个空列表，比如参数是某个函数动态生成的，请参考empty_parameter_set_mark选项。

可以对一个函数使用多个parametrize的装饰器，这样多个装饰器的参数会组合进行调用：

```
import pytest
@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
    pass
```

这会穷举x和y的所有组合并进行调用。

13.2 pytest_generate_tests范例

有时你可能需要自定义或者动态的决定参数。你可以使用在collection期间运行的pytest_generate_tests的钩子。该钩子中你可以使用metafunc对象检查请求的上下文，重要的时，你可以调用metafunc.parametrize()来实现参数化。

如下是个从命令行获取参数化的参数的示例：

```
# test_strings.py
def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

```
# conftest.py
def pytest_addoption(parser):
    parser.addoption("--stringinput", action="append", default=[], help="list of
stringinputs to pass to test functions")

def pytest_generate_tests(metafunc):
    if 'stringinput' in metafunc.fixturenames:
        metafunc.parametrize("stringinput", metafunc.config.getoption('stringinput'))
```

如果传入两个string，case会运行两次并且pass：

```
$ pytest -q --stringinput="hello" --stringinput="world" test_strings.py
.. [100%]
2 passed in 0.12 seconds
```

传入一个会导致失败的字符：

```
$ pytest -q --stringinput="!" test_strings.py
F                                                                    [100%]
================================= FAILURES =================================
_____________________________ test_valid_string[!] _____________________________
stringinput = '!'

    def test_valid_string(stringinput):
>       assert stringinput.isalpha()
E       AssertionError: assert False
E       + where False = <built-in method isalpha of str object at 0xdeadbeef>()
E       + where <built-in method isalpha of str object at 0xdeadbeef> = '!'.
↪isalpha

test_strings.py:3: AssertionError
1 failed in 0.12 seconds
```

如果不指定，那么metafunc.parametrize()会使用一个空的参数列表：

```
$ pytest -q -rs test_strings.py
s                                                                    [100%]
========================= short test summary info ==========================
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function test_
↪valid_string at $REGENDOC_TMPDIR/test_strings.py:1
1 skipped in 0.12 seconds
```

addoption 增加了一个"–stringinput"选项，格式是追加到一个list中，该list会通过metafunc.config.getoption传递给metafunc.parametrize并生成一个名字为"stringinput"的参数，这个参数(fixture?)会提供给test_valid_string使用。

Chapter 14 缓存

2.8引入

14.1 用法

插件提供了两种命令行方式来运行上一次pytest测试失败的用例：

- –lf, --last-failed 仅重新运行失败的用例
- –ff, --failed-first 先运行上次失败的用例，然后再运行剩余的其他用例

–cache-clear参数用来在开始一轮新的测试前清理所有的缓存。（通常并不需要这么做）

其他插件可以通过访问config.cache对象来设置/获取在pytest调用过程中传递的json格式的值

注意：这个插件默认是enabled，但是也可以disable掉，参看 Deactivating / unregistering a plugin by name

---

14.2 重新运行失败的测试或者先运行失败的测试

我们先来创建一个有50次调用的用例，其中有两次测试是失败的：

```
# test_50.py
import pytest

@pytest.mark.parametrize("i", range(50))
def test_num(i):
    if i in (17, 25):
        pytest.fail("bad luck")
```

运行这个测试你会看到两个失败的用例：

```
$ pytest -q
.................F.......F........................ [100%]
================================= FAILURES =================================
_________________________________ test_num[17] _________________________________

i = 17
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
_________________________________ test_num[25] _________________________________

i = 25
```

```
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
2 failed, 48 passed in 0.12 seconds
```

如果你再使用–lf参数来运行：

```
$ pytest --lf
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items / 48 deselected / 2 selected
run-last-failure: rerun previous 2 failures

test_50.py FF                          [100%]

================================ FAILURES ================================
_________________________________ test_num[17] _________________________________
i = 17
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            > pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
_________________________________ test_num[25] _________________________________
i = 25
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            > pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
================= 2 failed, 48 deselected in 0.12 seconds ==================
```

你会发现只有两个失败的用例被重新运行了一次，其他的48个用例并没有被运行（未被选中）。

现在你再使用–ff选项来运行，所有测试用例都会被运行，而且之前失败的两个用例会首先运行：

```
$ pytest --ff
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items
run-last-failure: rerun previous 2 failures first
```

```
test_50.py FF.............................................. [100%]
================================= FAILURES =================================
_____________________________ test_num[17] _____________________________
i = 17
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            > pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
_____________________________ test_num[25] _____________________________
i = 25
    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
            > pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
=================== 2 failed, 48 passed in 0.12 seconds ====================
```

另外还有类似的参数–nf，–new-first：首先运行新增/修改的用例，然后再运行已有的用例。所有的用例都会按照修改时间来排序。

14.3 前一次没有运行失败的用例时

当上一次测试没有失败的用例，或者没有缓存数据时，pytest可以配置是否运行所有的用例或者不运行任何用例：

```
pytest --last-failed --last-failed-no-failures all #运行所有用例（默认）
pytest --last-failed --last-failed-no-failures none #不运行任何用例并退出
```

14.4 config.cache对象

插件或者conftest.py支持代码使用pytest config对象获取一个缓存值。如下是一个简单的例子：

```
# test_caching.py
import pytest
import time

def expensive_computation():
    print("running expensive computation...")

@pytest.fixture
def mydata(request):
    val = request.config.cache.get("example/value", None)
    if val is None:
        expensive_computation()
        val = 42
        request.config.cache.set("example/value", val)
```

```
    return val

def test_function(mydata):
    assert mydata == 23
```

当你第一次运行用例时：

```
$ pytest -q
F [100%]
================================= FAILURES =================================
_____________________________ test_function _____________________________
mydata = 42
    def test_function(mydata):
        > assert mydata == 23
E       assert 42 == 23
E       -42
E       +23

test_caching.py:17: AssertionError
------------------------- Captured stdout setup --------------------------
**running expensive computation...**
1 failed in 0.12 seconds
```

当你再次运行时，你会发现expensive_computation中的打印不会再出现了：

```
$ pytest -q
F [100%]
================================= FAILURES =================================
_____________________________ test_function _____________________________
mydata = 42

    def test_function(mydata):
        > assert mydata == 23
E       assert 42 == 23
E       -42
E       +23

test_caching.py:17: AssertionError
1 failed in 0.12 seconds
```

14.5 查看缓存内容

你可以通过命令行参数–cache-show来查看缓存内容：

```
$ pytest --cache-show
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
cachedir: $PYTHON_PREFIX/.pytest_cache
------------------------------ cache values ------------------------------
```

```
cache/lastfailed contains:
    {'test_50.py::test_num[17]': True,
    'test_50.py::test_num[25]': True,
    'test_assert1.py::test_function': True,
    'test_assert2.py::test_set_comparison': True,
    'test_caching.py::test_function': True,
    'test_foocompare.py::test_compare': True}
cache/nodeids contains:
    ['test_caching.py::test_function']
cache/stepwise contains:
    []
example/value contains:
42

======================== no tests ran in 0.12 seconds ========================
```

14.6 清除缓存

你可以使用–cache-clear来清理缓存：

```
pytest --cache-clear
```

在那些不希望每次测试之间会可能产生干扰的持续集成服务器上，推荐使用该选项，因为正确性比速度更为重要。

14.7 逐步调试

作为–lf-x的一种替代方法，特别是你预期到你的大部分测试用例都会失败的情况下，使用–sw， --stepwise 允许你每次运行的时候都修复一个用例。测试集将运行到第一个失败的用例，然后自行停止，下一次再调用时，测试集将从上次失败的用例继续运行，然后运行到下一次失败的测试用例再停止。你可以使用–stepwise-skip选项来跳过一个失败的用例，在下一个失败的用例处再停止，如果你一直搞不定当前的失败的用例且只想暂时忽略它，那么这个选项非常有用。

Chapter 15 对unittest.TestCase的支持

pytest支持运行基于python unittest套件的测试。这意味着可以使用pytest来执行当前已经存在的unittest的测试用例，并能够逐步的调整测试套来使用更多的pytest的特性。

想要执行unittest格式的测试用例，使用如下命令行：

```
pytest tests
```

pytest会自动的再test_.py或者_.test.py文件中去查找unittest.TestCase的子类和他们的测试方法。pytest支持几乎所有的unittest的功能： ！

- @unittest.skip装饰器

- setUp/tearDown
- setUpClass/tearDownClass
- setupModule/tearDownModule
- pytest还不支持如下功能：
- load_tests protocol
- subtests

15.1 好处

大多数测试用例无需修改任何代码即可使用pytest来执行并引入一些新的功能：

- 更多的tracebacks的信息
- stdout和stderr的捕获
- 使用-k和-m的测试选项（参考Test selection options）
- 在第一（N）次失败后停止运行
- -pdb的支持
- 可以通过pytest-xdist插件在多CPU上进行分布式测试
- 使用简单的assert语句取代了self.assert* 函数

15.2 unittes.TestCase子类中的pytest特性

下列pytest特性在unittest.TestCase的子类中生效：

- Marks: skip，skipif， xfail；
- Auto-use fixtures

不支持下面的pytest的特性，因为设计哲学的不同，应该以后也不会支持：

- Fixtures
- Parametrization
- Custom hooks

第三方插件可能支持也可能不支持，取决于插件和对应的测试套

15.3 在unittest.TestCase的子类中使用marks来引用pytest fixtures

使用pytest允许你使用fixture来运行unittest.TestCase风格的测试。

下面是一个示例：

conftest.py

下面定义了一个fixture，可以通过名字来引用该fixture

```
import pytest

@pytest.fixture(scope="class")
def db_class(request):
    class DummyDB(object):
        pass

    request.cls.db = DummyDB()
```

这个fixture会在class里创建一个DummyDB的实例并赋值给类中的db属性。fixture通过传入一个特殊的可以访问调用该fixture的测试用例上下文（比如测试用例的cls属性）的request对象来实现。该架构可以使得fixture与测试用例解耦，并允许通过使用fixture名称这种最小引用来使用fixture。

下面开始在unittest.TestCase中使用该fixture：

```
# test_unittest_db.py

import unittest
import pytest

@pytest.mark.usefixtures("db_class")
class MyTest(unittest.TestCase):
    def test_method1(self):
        assert hasattr(self, "db")
        assert 0, self.db   # 测试目的

    def test_method2(self):
        assert 0, self.db   # 测试目的
```

类装饰器@pytest.mark.usefixtures(“db_class”)保证了pytest的fixture函数db_class在每个类中只会被调用一次。由于故意放置的导致失败的断言语句，我们可以在traceback中看到self.db的值：

```
$ pytest test_unittest_db.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_unittest_db.py FF                                                [100%]

================================= FAILURES =================================
_____________________________ MyTest.test_method1 _____________________________

self = <test_unittest_db.MyTest testMethod=test_method1>
```

```
    def test_method1(self):
        assert hasattr(self, "db")
>       assert 0, self.db # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef>
E       assert 0

test_unittest_db.py:9: AssertionError
_____________________________ MyTest.test_method2 ______________________________

self = <test_unittest_db.MyTest testMethod=test_method2>

    def test_method2(self):
>       assert 0, self.db # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef>
E       assert 0

test_unittest_db.py:12: AssertionError
========================== 2 failed in 0.12 seconds ==========================
```

通过默认的traceback可以看到这两个测试函数共享了一个self.db的实例，与我们定义fixture为class的预期行为一致。

15.4 自动调用的fixtures以及访问其他fixtures

通常最好是在你需要使用的fixtures的测试用例中显式的声明使用哪些fixtures，但是有时你可能希望在某些特定的上下文中自动调用一些fixtures。

毕竟传统的unittest的风格中允许这种隐式的调用，你可能已经习惯这种风格了。

你可以使用@pytest.fixture(autouse=True)来标记fiture，并且在你需要使用的地方对其进行定义。

我们来看一个例子，一个名为initdir的fixture，该fixture可以使所有TestCase的类在一个临时文件夹中执行，并在执行前初始化samplefile.ini。initdir使用pytest内置的fixture：tmpdir来为每个测试用例创建临时文件夹。

```python
# test_unittest_cleandir.py
import pytest
import unittest

class MyTest(unittest.TestCase):

    @pytest.fixture(autouse=True)
    def initdir(self, tmpdir):
        tmpdir.chdir()
        tmdir.join("samplefile.ini").write("# testdata")

    def test_method(self):
        with open("samplefile.ini") as f:
            s = f.read()

        assert "testdata" in s
```

因为initdir被标记为了autouse，所以该class下的所有的测试用例都会自动的调用initdir，就像定义了 @pytest.mark.usefixtures(“initdir”)一样。

运行结果如下：

```
$ pytest -q test_unittest_cleandir.py
.                                                           [100%]
1 passed in 0.12 seconds
```

可以看到测试结果是pass的，因为initdir这个fixture在执行test_method之前已经被调用了。

注意：unittest.TestCase不能直接使用fixture作为入参，因为这可能导致unittest.TestCase运行的某些异常。使用上面示例中的usefixture和autouse的方式可以将pytest的fixture使用到已有的unittest用例中，然后逐步的将unittest.TestCase的用例转化成pytest中的断言风格的测试用例。

Chapter 16 运行nose的测试用例（没用过nose。。跳过）

Chapter 17 经典的xunit-style

本章描述了一种实现fixtures经典且流行的方式。 注意在unittest和nose中的不同

17.1 模块级的setup/teardown

如果你在某个module中有多个测试用例/测试类，你可以选择实现下面的fixtures函数，这些函数只会在测试该模块中的所有函数中只被调用一次：

```
def setup_module(module):
    '''在模块开始执行前，任何特定的需要在该模块中执行的内容'''

def teardown_module(module):
    '''在模块退出执行前，任何特定的需要在该模块中执行的内容'''
```

在pytest3.0中，module这个参数是可选的。

17.2 class级的setup/teardown

同样的下面的方法在进入class和退出class的测试用例时会被调用一次：

```
@classmethod
def setup_class(cls):
    '''在class开始执行前，任何特定的需要在该class中执行的内容'''

@classmethod
def teardown_class(cls):
    '''在class退出执行前，任何特定的需要在该class中执行的内容'''
```

17.3 函数级别的setup/teardown

```
def setup_method(self, method):
    '''在method开始执行前，任何特定的需要在该method中执行的内容'''

def teardown_method(self, method):
    '''在method退出执行前，任何特定的需要在该method中执行的内容'''
```

在pytest3.0中，method这个参数是可选的。

如果你想直接在module里定义，可以使用下面的方式：

```
def setup_function(function):
    '''在function开始执行前，任何特定的需要在该function中执行的内容'''

def teardown_function(function):
    '''在function退出执行前，任何特定的需要在该function中执行的内容'''
```

在pytest3.0中，function这个参数是可选的。

注意事项：

- setup/teardown可以在测试进程中被多次调用
- 如果setup失败或者被跳过了，那么teardown也不会被调用
- pytest4.2之前，xunit-style没有严格的执行fixtures的scope的调用顺序，比如setup-method可能会在一个session作用域的自动使用的fixture前就被调用 现在这种scope的问题已经被解决。

Chapter 18 安装和使用插件

本章讨论如何安装和使用第三方插件。 可以使用pip很容易的安装第三方插件：

```
pip install pytest-NAME
pip uninstall pytest-NAME
```

安装插件后，pytest可以自动集成，不需要再次激活。 下面是一些流行的pytest插件（懒得翻译了。。。囧）：

- pytest-django: write tests for django apps, using pytest integration.
- pytest-twisted: write tests for twisted apps, starting a reactor and processing deferreds from test functions.
- pytest-cov: coverage reporting, compatible with distributed testing
- pytest-xdist: to distribute tests to CPUs and remote hosts, to run in boxed mode which allows to survive segmentation faults, to run in looponfailing mode, automatically re-running failing tests on file changes.
- pytest-instafail: to report failures while the test run is happening.
- pytest-bdd and pytest-konira to write tests using behaviour-driven testing.
- pytest-timeout: to timeout tests based on function marks or global definitions.
- pytest-pep8: a --pep8 option to enable PEP8 compliance checking.
- pytest-flakes: check source code with pyflakes.
- oejskit: a plugin to run javascript unittests in live browsers. 在http://plugincompat.herokuapp.com/中可以查看不同的插件对于不同的python及pytest的版本的兼容性。 还可以在https://pypi.org/search/?q=pytest- 查找更多的插件。

18.1 在module或者conftest文件中请求/加载插件

可以在module或者conftest中使用下面的方式加载插件，当module或者conftest被加载时，该插件也会被加载：

```
pytest_plugins = ("myapp.testsupport.myplugin",)
```

注意： 不推荐在非根目录的conftest.py中调用pytest_plugins。具体原因见下一章。

注意： pytest_plugins是保留字，不应该在自定义插件中使用该保留字。

18.2 查看被激活的插件

使用 pytest --trace-config 查看当前环境哪些插件被激活了。

18.3 通过插件名去激活

你可以通过下面的方式禁用某些插件：

```
pytest -p no:NAME
```

这表示任何尝试激活/加载NAME的插件都不会生效。 如果你想无条件的在项目中禁用某个插件，你可以在pytest.ini中添加如下选项：

```
[pytest]
addopts = -p no:NAME
```

如果仅想在某些特定的环境下禁用（比如CI服务器），你可以配置PYTEST_ADDOPTS这个环境变量为-p no：NAME

Chapter 19 自定义插件

一个插件包含一个或者多个钩子函数。下一章详细描述了如何自定义一个钩子函数。pytest针对configuration/collection/running/reporting实现了如下插件：

- 内置插件： 从pytest的内部目录_pytest加载
- 外部插件： 通过setuptools安装的外部插件
- conftest.py插件： 在测试目录下可以被自动加载的插件 理论上，每个钩子函数都可以定义N个python函数。这里的N指的是按照规范定义的钩子函数（也就是钩子函数可以用同一个函数名称定义多个）。 所有的规范的测试函数都以pytest_作为前缀，很容易通过函数名识别出来。

19.1 插件的加载顺序

pytest加载插件的顺序如下：

- 加载所有内置插件
- 加载所有通过setup tools安装的外部插件
- 加载通过命令行参数-p name指定的插件
- 加载conftest.py，如果没有指定测试目录，以当前目录作为测试目录，如果指定了测试目录，加载所有conftest.py以及test*/conftest.py
- 加载conftest.py中由变量pyetst_plugins指定的插件

19.2 conftest.py： 本地文件夹指定的插件

conftest.py插件包含了为本目录实现的钩子函数。测试用例只会加载那些为本目录定义的conftest.py里所定义的钩子。 如下pytest_runtest_setup钩子只有a目录下的测试函数会调用，其他目录的函数则不会使用该钩子

```
a/conftest.py:
    def pytest_runtest_setup(item):
        print("setting up", item)

a/test_sub.py:
    def test_sub():
        pass

test_flat.py:
    def test_flat():
        pass
```

你可以按照如下方式来运行：

```
pytest test_flat.py --capture=no #不会显示"setting up"
pytest a/test_sub.py --capture=no #会显示"setting up"
```

19.3 自定义插件

你可以参考下面的例子来实现自定义插件：

- 一个自定义的collection插件： 参考chapter 27.7
- pytest的内置插件
- pytest的外部插件

这些插件都实现了钩子函数，或者一些fixtrues来扩展功能。

19.4 让其他用户可以安装你的插件

如果你想要你的插件可以分享给其他人安装，你需要在setup.py中定义一个所谓的切入点以便pytest找到你的插件。切入点时setuptools提供的一个功能。 在下面的例子中，为pytest定义了pytest11这个切入点，这样你就可以在你的项目中使用该插件了：

```
# setup.py
from setuptools import setup

setup(
    name="myproject",
    packages=["myproject"],
    entry_points={"pytest11":["name_of_plugin = myproject.pluginmodule"]},
    classifiers=["Frameword :: Pytest"]
)
```

如果使用了该方式安装，pytest会安装myproject.pluginmodule插件。

注意： 确保包含了"Frameword :: Pytest"这个分类器，这可以让用户更方便的找到你的插件。

19.5 重写断言

烦躁

19.6 在module或者conftest文件中加载插件

你可以在module或者conftest中加载插件：

```
pytest_plugins = ["name1", "name2"]
```

当module或者conftest被加载时，该插件也会被加载(跟18.1有啥区别？)。所有module都可以被作为一个插件来进行加载，包括内部module：

```
pytest_plugins = "myapp.testsupport.myplugin"
```

pytest_plugins是递归调用的，如果上面的例子中的myapp.testsupport.myplugin也定义了pytest_plugins， 其中所定义的上下文也会被加载为插件。

注意： 不推荐在非根目录的conftest.py中调用pytest_plugins。 conftest.py虽然是为每个目录定义的文件，但是一旦插件被导入，那么该插件会在所有的目录中生效。为了避免混淆，在非根目录的conftest.py定义pytest_plugins是不推荐的，并且会抛出一条告警信息。

19.7 通过插件名访问

如果你要在插件中使用另外一个插件的代码，你可以通过下面的方式来引用：

```
plugin = config.pluginmanager.get_plugin("name_of_plugin")
```

使用–trace-config可以查看已经存在的插件。

19.8 测试插件

pytest通过一个名为pytester的插件来测试插件，这个插件默认是disable的，你需要在使用前enable。 你可以在conftest.py中添加下面的代码来enable该插件

```
# conftest.py

pytest_plugins = ["pytester"]
```

你也可以通过传入命令行参数-p pytester来enable。

我们来通过下面的例子来展示你可以通过这个插件做哪些事。 假设我们实现了一个插件，包含一个名为hello的fixture，该fixture会返回一个“Hello World！”的字符串：

```
# -*- coding: utf-8 -*-

import pytest

def pytest_addoption(parser):
```

```
    group = parser.getgroup("helloworld")
    group.addoption(
        "--name",
        action="store",
        dest="name",
        default="World",
        help='Default "name" for hello().',
    )

@pytest.fixture
def hello(request):
    name = request.config.getoption("name")

    def _hello(name=None):
        if not name:
            name = request.config.getoption("name")

        return _hello
```

现在我们就可以使用testdir提供的API来创建一个临时的conftest.py和测试文件，这些临时文件可以用来运行测试用例并返回结果，可以用来检查测试的输出是否符合预期

```
def test_hello(testdir):
    """确保插件是正常工作的"""

    # 创建临时文件conftest.py
    testdir.makeconftest(
        """
        import pytest

        @pytest.fixture(params=["Brianna", "Andreas", "Floris",])
        def name(request):
            return request.param
        """
    )

    # 创建临时的测试文件
    testdir.makepyfile(
        """
        def test_hello_default(hello):
            assert hello() == "Hello World!"

        def test_hello_name(hello, name):
            assert hello(name) == "Hello {0}!".format(name)
        """
    )

    result = testdir.runpytest()
    result.assert_outcomes(pass=4)
```

Chapter 20 编写钩子函数

20.1 钩子函数的验证和执行

pytest的钩子函数定义在插件中。我们来看一个典型的钩子函数pytest_collection_modifyitems(session, config, items)，该钩子函数会在pytest完成测试对象的collection之后调用。

当我们在插件中定义pytest_collection_modifyitems时，pytest会在注册阶段检查参数名是否符合要求，如果不匹配，则会退出。

下面是一个示例：

```
def pytest_collection_modifyitems(config, items):
    #在collection完成后调用，可以在这里修改“items”
```

这里，pytest使用了config（pytest配置项）以及items（待测试对象）作为入参，没有使用session。这种动态的参数“删减”使得pytest是可以做到“将来兼容”。我们可以引入新的钩子参数，而不破坏现有的钩子实现的原型，这也是pytest插件可以做到长期兼容的原因之一。

注意钩子函数不允许抛出异常，如果在钩子函数中抛出异常，会使得pytest的运行中断。

20.2 firstresult：在第一个非None结果的用例终止

pytest的钩子函数都有一个列表，其中是该钩子函数的所有的非None的结果（这句感觉怪怪的，感觉意思是钩子都存在一个列表里，然后pytest会顺着这个列表调用同名的钩子函数。。。）。

一些钩子定义了first result=true选项，这样钩子在调用时只会执行到注册的N个钩子函数中的第一个返回非None结果的钩子，然后用该非None的结果作为整个钩子调用的结果返回。这种情况下，剩下的钩子函数不会再被调用。

20.3 hookwrapper：在钩子的上下文执行

2.7引入

pytest插件允许对钩子进行包装。钩子的包装函数是一个生成器并且只会yield一次。当pytest调用钩子的时候首先会调用钩子的包装函数并且将传给钩子的参数也传递给该包装函数。

在yield处，pytest会执行钩子函数并且将结果返回给yield。

下面是包装函数的一个示例：

```
import pytest

@pytest.hook.impl(hookwrapper=True)
def pytest_pyfunc_call(pyfuncitem):
    do_something_before_next_hook_executes()

    outcome = yield
```

```
    res = outcome.get_result()
    post_process_result(res)
    outcome.force_result(new_res)
```

注意，钩子包装器本身并不返回结果，它们只是围绕钩子做一些跟踪处理。如果底层钩子的结果是一个可变的对象，那么包装器可以修改该结果，但最好避免这样做。

20.4 钩子函数的排序/调用示例

任意的钩子函数都可能不止一个实现，通常我们可以看到钩子函数被执行了N次，这里的N次就是注册的同名的N个钩子函数。 通过调整这N个钩子函数的位置，可以影响钩子函数的调用过程：

```
# Plugin 1
@pytest.hookimpl(tryfirst=True)
def pytest_collection_modifyitems(items):
    # 会尽早执行
    ...

# Plugin 2
@pytest.hookimpl(trylast=True)
def pytest_collection_modifyitems(items):
    # 会尽可能晚的执行
    ...

# Plugin 3
@pytest.hookimpl(hookwrapper=True)
def pytest_collection_modifyitems(items):
    # 会在tryfirst之前执行
    outcome = yield
    # 会在所有非hookwrappers之后执行
```

下面是执行顺序：

- 调用Plugin 3的pytest_collection_modifyitems直到yield
- 调用Plugin 1的pytest_collection_modifyitems
- 调用Plugin 2的pytest_collection_modifyitems
- 调用Plugin 3的pytest_collection_modifyitems的yield之后的代码

20.5 定义新的钩子

插件和conftest.py可能会定义一些新的钩子来改变一些pytest的行为或者与其他的插件进行交互：
pytest_addhooks(pluginmanager) 在插件注册阶段来添加一个新的钩子函数：
pluginmanager.add_hookspecs(module_or_class, prefix)

```
注意这个钩子与hookwrapper=True不兼容
```

定义的新的钩子一般是一个空函数，仅包含一些文档注释来描述这个钩子应该什么时候被调用，期望返回值是什么

20.6 。。。

Chapter 21 日志

3.3 引入， 3.4更新

pytest默认捕获WARNING或以上级别的log并在测试结束后输出： 使用pytest不带参数运行：

```
pytest
```

显示失败的用例：

```
----------------------- Captured stdlog call ----------------------
test_reporting.py 26 WARNING text going to logger
----------------------- Captured stdout call ----------------------
text going to stdout
----------------------- Captured stderr call ----------------------
text going to stderr
==================== 2 failed in 0.02 seconds ====================
```

如果要格式化日志和时间，使用下面的格式化参数来指定：

```
pytest --log-format="%(asctime)s %(levelname)s %(message)s" --log-date-format="%Y-%m-%d
%H:%M:%S"
```

显示失败的测试用例的格式如下：

```
2010-04-10 14:48:44 WARNING text going to logger
----------------------- Captured stdout call ----------------------
text going to stdout
----------------------- Captured stderr call ----------------------
text going to stderr
==================== 2 failed in 0.02 seconds ====================
```

也可以在pytest.ini文件中指定：

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

甚至可以禁止掉失败用例的所有输出（stdout，stderr以及logs）：

```
pytest --show-capture=no
```

21.1 fixture: caplog

通过caplog可以改变测试用例内部的log的级别：

```
def test_foo(caplog):
    caplog.set_level(logging.INFO)
    pass
```

默认是设置的root logger，但是也可以通过参数指定：

```
def test_foo(caplog):
    caplog.set_level(logging.CRITICAL, logger='root.baz')
    pass
```

log的level会在测试结束后自动恢复。

也可以在with代码块中通过上下文管理器临时改变log的level

```
def test_bar(caplog):
    with caplog.at_level(logging.INFO):
        pass
```

同样，可以通过参数指定logger：

```
def test_bar(caplog):
    with caplog.at_level(logging.CRITICAL, logger='root.baz'):
        pass
```

最后，在测试运行期间发送给logger的所有日志都可以在fixture使用，包括logging.logrecord实例和最终日志文本。当您希望对消息的内容进行断言时，此选项非常有用：

```
def test_baz(caplog):
    func_under_test()
    for record in caplog.records:
        assert record.levelname != 'CRITICAL'
    assert 'wally' not in caplog.text
```

其他的属性可以参考logging.Logrecord类。

你可以使用record_tuples来保证消息使用给定的紧急级别进行记录的：

```
def test_foo(caplog):
    logging.getLogger().info('boo %s', 'arg')

    assert caplog.record_tuples == [('root', logging.INFO, 'boo arg')]
```

使用caplog.clear()来重置捕获的日志：

```
def test_something_with_clearing_records(caplog):
    some_method_that_creates_log_records()
    caplog.clear()
    your_test_method()
    assert ['Foo'] == [rec.message for rec in caplog.records]
```

caplog.records只包含当前阶段的信息，所以在setup阶段这里只会有setup的信息，call和teardown也一样。

可以通过caplog.get_records(when)来获取测试过程中其他阶段的log。下面这个例子，在teardown中检查了setup和call阶段的log来确保在使用某些fixtures的时候没有任何告警

```
@pytest.fixture
def window(caplog):
    window = create_window()
    yield window
    for when in ("setup", "call"):
        messages = [
            x.message for x in caplog.get_records(when) if x.level == logging.WARNING
        ]
        if messages:
            pytest.fail("warning messages encountered during testing:
{}".format(messages))
```

21.2 实时log

设置log_cli为true，pytest会将log直接在控制台输出。

使用–log-cli-level可以设置在控制台输出的log的level，该设置可以用level的名称或者是代表level的数字。

另外，也可以通过–log-cli-format和–log-cli-data-format指定格式，如果没有指定，则使用的是默认值–log-format和–log-date-format，这些只对控制台生效（废话，带cli）

所有的cli的配置项都可以在ini中配置，配置项名为：

- log_cli_level
- log_cli_format
- log_cli_date_format

如果想要将所有log记录到某个文件，使用–log-file=/path/to/log/flie这种方式。

也可以使用–log-file-level指定写入到log文件的log的level。 另外，也可以通过–log-file-format和–log-file-data-format指定格式，如果没有指定，则使用的是默认值–log-format和–log-date-format（跟控制台是一样的用法）

所有的log文件的配置项都可以在ini中配置，配置项名为：

- log_file
- log_file_level
- log_file_format
- log_file_date_format

21.3

21.4

Chapter 22 API参考

22. 函数

22.1.1 pytest.approx

approx(expected, rel=None, abs=None, nan_ok=False) 判断两个数字(或两组数字)是否在误差允许范围内相等。 由于浮点数的精度问题，有的时候我们认为相等的数字其实并不相等：

```
>>> 0.1 + 0.2 == 0.3
False
```

这种情形在编写测试用例的时候经常碰到，我们需要判断某些浮点值是否符合预期。一个解决方法是判断浮点值是否在误差范围内：

```
>>> abs((0.1 + 0.2) - 0.3) < 1e-6
True
```

然而，这种方式既繁琐又不易理解。此外，并不推荐上面这种绝对比较的方式，因为没有一种对所有情况都适用的标准误差。1e-6对于1左右的数字来说是一个很好的误差，但是对于大数来说太小了，对于更小的数来说又太大了。最好的方式是将误差作为期望值的一部分，但是这样的方式更难正确并简洁的用代码来表达。

approx类使用了尽量简洁的语法来进行浮点数比较：

```
>>> from pytest import approx
>>> 0.1 + 0.2 == approx(0.3)
True
```

同样可以用于数据集：

```
>>> (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))
True
```

也可以用于字典：

```
>>> {'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})
True
```

numpy格式的数组：

```
>>> import numpy as np
>>> np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))
True
```

numpy的标量（没研究过标量）：

```
>>> import numpy as np
>>> np.array([0.1, 0.2]) + np.array([0.2, 0.1]) == approx(0.3)
True
```

可以通过参数来设置approx的相对或者绝对误差：

```
>>> 1.0001 == approx(1)
False
>>> 1.0001 == approx(1, rel=1e-3)
True
>>> 1.0001 == approx(1, abs=1e-3)
True
```

如果指定了abs，那么比较时不会考虑相对误差，换句话说，即使误差在默认的相对误差1e-6的范围内，如果超过了abs定义的误差，这两个数的比较结果也是不想等的。但是如果abs和rel都定义了，那么只要满足其中任何一个，都会被认为时相等的：

```
>>> 1 + 1e-8 == approx(1)
True
>>> 1 + 1e-8 == approx(1, abs=1e-12)
False
>>> 1 + 1e-8 == approx(1, rel=1e-6, abs=1e-12)
True
```

如果你准备使用approx，那么你可能想知道这个方法与其他的浮点数比较的通用方法有什么不同，当然其他的比较流行的比较方式也都是基于rel和abs误差的，但是确实时有区别的：

- math.isclose(a, b, rel_tol=1e-9, abs_tol=0.0)
- numpy.isclose(a, b, rtol=1e-5, atol=1e-8)
- unittest.TestCase.assertAlmostEqual(a, b)

- a == pytest.approx(b, rel=1e-6, abs=1e-12)

注意，3.2 更新： 为了保持一致性，如果使用>, >=, < 以及<=会引发一个TypeError

22.1.2 pytest.fail

fail(msg=”, pytrace=True) 使用给定的消息显式地使正在执行的测试失败。 参数：

- msg(str) - 显示给用户的失败消息
- pytrace(bool) - 如果设置为false，那么使用msg作为完整的失败信息而不显示python的traceback

22.1.3 pytest.skip

skip(msg[, allow_module_level=False]) 使用给定的消息跳过正在执行的测试。 该函数只能在测试阶段（setup/call/teardown）或者collection阶段通过使能allow_module_level调用。 参数：

- allow_module_level(bool) - 允许该函数在module level调用，跳过该module剩余的测试。默认为False 最好使用pytest.mark.skipif

22.1.4 pytest.importskip

importorskip(modname, minversion=None, reason=None) 如果module不能被import，那么跳过该测试 参数：

- modename(str) - 待import的module的名字
- minversion(str) - 如果定义了该参数，那么定义在__version__中的版本号必须不小于该值，否则也会跳过测试
- reason(str) - 如果定义了该参数，那么这个字符串会在module不能import的时候显示

22.1.5 pytest.xfail

xfail(reason=”) 使用给定的原因强制当前的测试或者setup失败。 该函数只能在测试阶段调用(setup/call/teardown) 最好使用pytest.mark.xfail

22.1.6 pytest.exit

exit(msg, returncode=None) 退出测试进程 参数：

- msg(str) - 退出时显示的信息
- returncode(int) - 退出pytest的时候的返回码

22.1.7 pytest.main

main(args=None, plugins=None) 在进程中执行测试结束后，返回一个退出码 参数：

- args - 命令行参数列表
- plugins - 初始化阶段自动注册的插件对象列表

22.1.8 pytest.param

param(*values[, id][, marks]) 为pytest.mark.parametrize或parametrized fixtures指定参数：

```
@pytest.mark.parametrize("test_input, expected", [
    ("3+5", 8),
    pytest.param("6*9", 42, marks=pytest.mark.xfail)
])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

参数：

- values - 按照顺序定义的参数集
- marks - 为参数集指定的marker
- id(str) - 为该参数集指定一个id

22.1.9 pytest.raises

with raises(expected_exception: Exception[, match][, message]) as excinfo 判断代码或者函数是否调用/抛出了指定的异常。 参数：

- match - 如果指定了该参数，检查抛出的异常是否与该文本匹配
- message - (4.1以后不再推荐使用) 如果指定了该参数，在没有匹配到异常时显示此消息

使用pytest.raises作为上下文管理器可以捕获指定的异常：

```
>>> with raises(ZeroDivisionError):
...     1/0
```

如果上面的代码没有抛出ZeroDivisionError异常，这里的检查结果就是失败的。 也可以使用match参数来检查异常是否与指定文本相匹配

```
>>> with raises(ValueError, match='must be 0 or None'):
...     raise ValueError("value must be 0 or None")
>>> with raises(ValueError, match=r'must be \d+$'):
...     raise ValueError("value must be 42")
```

上下文管理器会生成一个ExceptionInfo对象，其中包含了捕获的异常的详细信息：

```
>>> with raises(ValueError) as exc_info:
...     raise ValueError("value must be 42")
```

```
>>> assert exc_info.type is ValueError
>>> assert exc_info.value.args[0] == "value must be 42"
```

4.1以后不再推荐： 使用message参数自定义失败信息。 因为用户经常用错。。。。

还有一堆。。。感觉平时用不着。不写了

22.1.10 pytest.deprecated_call

with deprecated_call() 检查是否触发了DeprecationWarning或者 PendingDeprecationWarning：

```
>>> import warnings
>>> def api_call_v2():
...     warnings.warn('use v3 of this api', DeprecationWarning)
...     return 200
>>> with deprecated_call():
...     assert api_call_v2() == 200
```

22.1.11 pytest.register_assert_rewrite

register_assert_rewrite(*names) 注册一个或者多个module名，这些module的assert会在import的时候被重写 该函数会使得这些package内的module的assert语句被重写，所以需要保证该函数在module被import之前调用，通常放在__init__.py中 如果给定的module名不是string类型会抛出TypeError

22.1.12 pytest.warns

with warns(expected_warning: Exception[, match]) 判断是否出现了指定的告警。

参数expected_warning可以是一个或者一系列warning类。

该函数也可以作为上下文管理器，或者使用pytest.raises的其他用法：

```
>>> with warns(RuntimeWarning):
...     warnings.warn("my warning", RuntimeWarning)


上下文管理器中，你可以时match来匹配特定的warning

>>> with warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)
>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)
>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

22.1.13 pytest.freeze_includes

freeze_includes() 返回cx_freeze提供给pytest的module名称列表。。。(这玩意有啥用？)

22.2 Marks

22.2.1 pytest.mark.filterwarnings

为测试函数添加告警filter pytest.mark.filterwarnings(filter) 参数：

- filter(str) - 告警字符串，由元组(action, message, category, module, lineno)组成，通过“：”来分割。可选字段可以省略。module名不是正则转义的。

比如：

```
@pytest.mark.warnings("ignore:.*usage will be deprecated.*:DeprecationWarning")
def test_foo():
    ...
```

22.2.2 pytest.mark.parametrize

Metafunc.parametrize(argnames, argvalues, indirect=False, ids=None, scope=None) 根据给定的参数列表里的参数添加新的函数调用。参数化是在collection阶段执行的。如果有比较耗费资源的设置，请使用间接设置的方法而不是在测试用例里直接设置。

参数：

- argnames - 由逗号分隔的代表参数名的字符串，或者一个参数字符串的列表/元组
- argvalues - argvalues的数量决定了测试函数会被调用多少次。 如果只有一个参数，那么argvalues是一个list，如果有N个参数，argvalues是一个N元tuple，tuple里的每个值代表一个参数。
- indirect - 由参数名或者布尔值组成的列表。 这个列表是argnames的一个子集。如果这里配置了argnames，那么对应的列表中的argname会作为request.param传递给相关的fixture函数，因此使用这种方式可以在setup阶段执行开销较大的设置。
- ids - 字符串列表或者一个函数。 如果是字符串，则这里的id作为设置的测试id一一对应于argvalues。如果没有指定任何参数作为测试id，那么自动生成测试id。 如果是函数，那么这个函数接受一个入参(单个的argvalue)并返回一个字符串或者None，如果返回的是None，那么自动生成测试id。 如果没有提供id，将根据argvalues自动生成。
- scope - 如果指定了该参数，则表示参数范围。scope用于按照参数对测试进行分组，同时也会复写fixture的范围，允许在测试上下文/配置中动态设置scope

22.2.3 pytest.mark.skip

无条件的跳过一个测试函数 pytest.mark.skip(*, reason=None) 参数：

- reason(str) - 跳过测试函数的原因

22.2.4 pytest.mark.skipif pytest.mark.skipif(condition, *, reason=None) 参数：

- condition(bool or str) - True/False 跳过函数的判断条件
- reason(str) - 跳过测试函数的原因

22.2.5 pytest.mark.usefixtures

对该函数使用指定的fixtures 注意该函数对fixture函数不生效 pytest.mark.usefixtures(*names) 参数：

- args - 待使用的fixture的字符串名称

22.2.6 pytest.mark.xfail

指定一个函数是预期失败的。 pytest.mark.xfail(condition=None, *, reason=None, raises=None, run=True, strict=False) 参数：

- condition(bool or str) - True/False 函数失败的判断条件
- reason(str) - 测试函数失败的原因
- raises (Exception) – 期望该函数抛出的异常。其他未指定的异常会导致该测试用失败
- run(bool) - 是否期望该函数被执行。如果设置为False，该函数不会被执行并且测试结果是xfail。
- strict(bool) - balabalbala。。。。。

22.2.7 自定义marks

可以使用pytest.mark来动态的创建marks：

```
@pytest.mark.timeout(10, "slow", method="thread")
def test_function():
    ...
```

上面的代码会创建一个Mark的对象去收集待测试的item，这个对象可以通过fixtures或者钩子使用Node.iter_markers来访问。 mark对象具有以下属性：

```
mark.args = (10, "slow")
mark.kwargs = {"method": "thread"}
```

22.3 Fixtures

Fixtures可以在函数或者其他fixtures中定义参数名来调用： 测试函数调用fixture的示例：

```
def test_output(capsys):
    print("hello")
    out, err = capsys.readouterr()
    assert out == "hello\n"
```

fixture调用fixture的示例：

```
@pytest.fixture
def db_session(tmpdir):
    fn = tmpdir / "db.file"
    return connect(str(fn))
```

22.3.1 @pytest.fixture

@fixture(scope=’ function’ , params=None, autouse=False, ids=None, name=None) 标记下面的函数是fixture的装饰器。

该装饰器可以用来定义一个带参数或者不带参数的fixture函数。

fixture的函数名可以在定义之后用于在运行测试前调用：测试modules或者classes可以使用pytest.mark.usefixtures(fixturename)。

测试函数可以直接使用fixture的函数名作为入参来调用fixture用以注入fixture函数的返回值。

fixture函数可以使用return或者yield来返回需要的值。如果使用yield，那么yield后面的代码将会在teardown部分调用，并且函数中只能有一个yield。 参数：

- scope - fixture的作用域：function(默认)/class/module/package/session， 注意package目前是实验阶段。
- params - 可选的参数列表。 每个参数都会触发一次调用。
- autouse - 如果设置为True，那么所有作用域内的测试函数都会自动调用该fixture。如果是False，那么需要显式的调用fixture
- ids - 对应参数的id，如果没有设置id，那么会根据参数自动生成id
- name - 重命名这个fixture

22.3.2 config.cache

config.cache对象允许其他插件和fixture在测试运行过程中进行存储和取值。 在fixture中使用该特性需要在fixture中引入pytestconfig，然后通过ptestconfig.cache来获取。

cache插件使用了json标准模块中的dumps/loads。

Cache.get(key, default) 根据给定的key查找并返回cache中的值。如果没有缓存值或者无法读取，返回默认的指定值。 参数：

- key - 通常以插件或者应用名开头。
- default - 当未命中缓存或者无效缓存值时返回的默认值，必须提供。

Cache.set(key, value) 根据给定的值设置cache。 参数：

- key - 通常以插件或者应用名开头。
- value - 必须是python的基本类型的组合，包含字典列表等一些嵌套类型。
-

Cache.makedir(name)

返回name指定的文件夹对象。如果文件夹不存在，会自动创建该文件夹。 参数：

- name- 必须是一个不包含/分隔符的字符串。确保名称包含你的插件或者应用程序的识别符，避免与其他cache用户发生冲突。

22.3.3 capsys

capsys() 使能对sys.stdout和sys.stderr的捕获，保证被捕获的输出可以通过capsys.readouterr()来访问，capsys.readouterr()会返回(out, err)格式的元组，其中out和err是text对象。

返回值是CaptureFixture的一个实例，示例如下：

```
def test_output(capsys):
    print("hello")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
```

class CaptureFixture capsys(), capsysbinary(), capfd() and capfdbinary() fixtures 会返回该类的实例.

readouterr() 读取并返回目前捕获的输出，重置内部buffer。 返回一个以out和err为属性的元组。

with disabled() 在with块代码中临时禁止捕获输出。

22.3.4 capsysbinary

capsysbinary() 使能对sys.stdout和sys.stderr的捕获，保证被捕获的输出可以通过capsysbinary.readouterr()来访问，capsysbinary.readouterr()会返回(out, err)格式的元组，其中out和err是bytes对象。

返回值是CaptureFixture的一个实例，示例如下：

```
def test_output(capsysbinary):
    print("hello")
    captured = capsysbinary.readouterr()
    assert captured.out == b"hello\n"
```

22.3.5 capfd

capfd() 使能对文件描述符1和2的捕获，保证被捕获的输出可以通过capfd.readouterr()来访问，capfd.readouterr()会返回(out, err)格式的元组，其中out和err是text对象。

返回值是CaptureFixture的一个实例，示例如下：

```
def test_system_echo(capfd):
    os.system('echo "hello"')
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
```

22.3.6 capfdbinary

capfdbinary() 使能对文件描述符1和2的捕获，保证被捕获的输出可以通过capfdbinary.readouterr()来访问，capfdbinary.readouterr()会返回(out, err)格式的元组，其中out和err是bytes对象。

返回值是CaptureFixture的一个实例，示例如下：

```
def test_system_echo(capfdbinary):
    os.system('echo "hello"')
    captured = capfdbinary.readouterr()
    assert captured.out == "hello\n"
```

22.3.7 doctest_namespace

doctest_namespace() 该fixture会返回一个被注入到doctests的命名空间的dict。 该fixture通常与autouse一起使用：

```
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace["np"] = numpy
```

22.3.8 request

request用于提供请求进行测试的函数的信息的fixture。

class FixtureRequest 来自测试函数或者fixture函数的fixture请求。

提供了访问请求上下文的能力，如果fixture是间接参数化的，提供可选项param

```
*fixturename = None*
    正在执行此请求的fixture名
*scope = None*
    作用域， 为function/class/module/session之一
```

```
*fixturenames*
    在该请求中所有激活的fixtures名
*node*
    基本collection的节点(取决于当前的request作用域)
*config*
    该request相关的pytest config对象
*function*
    测试函数对象，如果该request是function作用域的话
*cls*
    。。。。。太多了。懒得烦了。
```

22.3.9 pytestconfig

pytestconfig() 作用域为session，返回_pytest.config.Config对象。

```
def test_foo(pytestconfig):
    if pytestconfig.getoption("verbose"):
        ...
```

22.3.10 record_property

record_property() 为测试添加一个额外的属性。 用户属性是测试报告的一部分，并且可以由报告者来进行配置，比如JUnit XML。该固件的入参格式为(name, value)， 其中value会自动使用xml编码。

```
def test_function(record_property):
    record_property("example_key", 1)
```

22.3.11 caplog

caplog() 访问和控制log的捕获。 可以通过下面的方法来访问捕获的log：

```
* caplog.text                 -> string格式的日志输出
* caplog.records              -> logging.LogRecord实例列表
* caplog.record_tuples        -> (logger_name, level, message)列表
* caplog.clear()              -> 清除捕获的log并且格式化log字符串
```

该fixture会返回_pytest.logging.LogCaptureFixture实例。

22.3.12 monkeypatch

monkeypatch()

monkeypatch可以用来在实现中修改对象/字典/os.environ：

```
monkeypatch.setattr(obj, name, value, raising=True)
monkeypatch.delattr(obj, name, raising=True)
monkeypatch.setitem(mapping, name, value)
monkeypatch.delitem(obj, name, raising=True)
monkeypatch.setenv(name, value, prepend=False)
monkeypatch.delenv(name, raising=True)
monkeypatch.syspath_prepend(path)
monkeypatch.chdir(path)
```

所有的改动都会在测试函数或者fixture结束后取消。参数raising绝对了是否在set/deletion操作没有具体对象的时候是否抛出KeyError或者AttributeError。

该fixture会返回MonkeyPatch实例。

22.3.13 testdir

testdir提供了对黑盒测试非常有用的Testdir实例，这是测试插件的理想方案。 要使用它，请将其包含在最顶层的conftest.py文件中：

```
pytest_plugins = 'pytester'
```

22.3.14 recwarn

recwarn() 返回一个包含了测试函数触发的所有告警记录的WarningsRecorder实例。 参考python的告警文档： http://docs.python.org/library/warnings.html

22.3.15 tmpdir

tmpdir() 返回一个对每个测试用例来说都是唯一的临时文件夹，该临时文件夹是系统的临时文件夹的一个子文件夹。 返回对象是py.path.local。

22.3.16 tmpdir_factory

tmpdir_factory实例包含两个方法：

- TempdirFactory.mktemp(basename, numbered=True) 在系统的临时文件夹下面创建一个子文件夹并返回这个子文件夹。如果numbered是True，会通过添加一个比现有的文件夹使用的前缀数字更大的数字作为文件夹名的前缀。
- TempdirFactory.getbasetemp() 对_tmppath_factory.getbasetemp的后向兼容

22.4 Hooks

22.4.1 Bootstrapping hooks (百度翻译为步步为营，尼玛什么鬼)

Bootstrapping hooks 在插件注册的一开始就被调用(内部和setuptools插件)。
pytest_load_initial_conftests(early_config, parser, args) 在命令行选项解析之前，实现加载初始化conftest文件。

注意该钩子不会被conftest.py本身调用，只对setuptools插件生效

参数：

- early_config (_pytest.config.Config) – pytest配置对象
- args (list[str]) – 命令行传递的参数
- parser (_pytest.config.Parser) – 添加命令行选项

pytest_cmdline_preparse(config, args) 不推荐使用。在选项解析前修改命令行参数。 不推荐，将来会被移除，使用pytest_load_initial_conftest()代替。

注意该钩子不会被conftest.py本身调用，只对setuptools插件生效

参数：

- config (_pytest.config.Config) – pytest配置对象
- args (list[str]) – 命令行传递的参数

```
pytest_cmdline_parse(pluginmanager, args)
```

返回初始化好的配置对象，解析特定的参数。

注意该钩子不会被conftest.py本身调用，只对setuptools插件生效

参数：

- pluginmanager (_pytest.config.PytestPluginManager) – pytest的插件管理器
- args (list[str]) – 命令行传递的参数

```
pytest_cmdline_main(config)
```

该函数是用来执行命令行的主函数。默认的实现是调用配置的钩子并运行主函数runtest_mainloop：

注意该钩子不会被conftest.py本身调用，只对setuptools插件生效

参数：

- config (_pytest.config.Config) – pytest配置对象

22.4.2 初始化钩子

初始化钩子在插件及conftest.py中调用。

pytest_addoption(parser) 注册argparse-style选项以及ini-style配置值，在测试运行开始调用一次。

注意，因为pytest的插件查找机制，该函数只应该在测试的root文件夹下的插件及conftest.py中实现

参数：

- parser (_pytest.config.Parser) – 添加命令行选项使用pytest.addoption(...)，添加ini-file值使用parser.addini(...)

之后可以通过config对象来访问：

- config.getoption(name) 来获取命令行选项的值
- config.getini(name) 来获取从ini-style文件中读取的值

config对象在许多内部对象中通过.config属性来传递，也可以使用pytestconfig这个fixture来获取config对象。

注意该钩子与hookwrapper=True不兼容

pytest_addhooks(pluginmanager) 在插件注册时调用，通过pluginmanager.add_hookspecs(module_or_class, prefix)来添加新的钩子。 参数：

pluginmanager (_pytest.config.PytestPluginManager) – pytest的插件管理器

注意该钩子与hookwrapper=True不兼容

pytest_configure(config) 允许插件和conftest文件执行初始化配置。 命令行选项解析完成后，该钩子会被每一个插件和最初的conftest文件(root目录下的conftest?)的调用。 这之后，该钩子会被其他import的conftest文件调用。

==注意该钩子与hookwrapper=True不兼容

参数：

- config (_pytest.config.Config) – pytest配置对象

pytest_unconfigure(config) 在测试进程退出前调用。

参数：

- config (_pytest.config.Config) – pytest配置对象

pytest_sessionstart(session) 该钩子在Session对象创建之后，collection和运行测试之前调用。 参数：

- session(_pytest.main.Session)- pytest session 对象

pytest_sessionfinish(session, exitstatus) 在所有测试完成后，在向系统返回退出状态前调用。 参数：

- session(_pytest.main.Session)- pytest session 对象 exitstatus(int) - pytest返回给系统的状态

22.4.3 测试运行时的钩子

所有运行时钩子都会接收一个pytest.Item对象。

pytest_runtestloop(session) 执行main runtest loop的时候调用(在collection完成之后)。

参数：

- session(_pytest.main.Session)- pytest session 对象

pytest_runtest_protocol(item, nextitem) 为指定的测试项实现runtest_setup/call/teardown协议，包含捕获异常以及调用报告钩子。

参数：

- item - 执行runtest协议的测试项
- nextitem - 待调度的下一个测试项。该参数会被传递给pytest_runtest_teardown(). 返回值：
- boolean 如果没有更多的钩子实现需要调用的话返回True

pytest_runtest_logstart(nodeid, location) 发出开始运行单个测试项的信号。

该钩子在pytest_runtest_setup(), pytest_runtest_call() 和 pytest_runtest_teardown()之前调用。

参数：

- nodeid(str) - 测试项的完整id
- location - 一个三元组(filename，linenum，testnum)

pytest_runtest_logfinish(nodeid, location) 发出单个测试项运行结束的信号。

该钩子在pytest_runtest_setup(), pytest_runtest_call() 和 pytest_runtest_teardown()之后调用。
参数：

- nodeid(str) - 测试项的完整id
- location - 一个三元组(filename，linenum，testnum)

pytest_runtest_setup(item) 在pytest_runtest_call(item)之前调用

pytest_runtest_call(item) 执行测试用例

pytest_runtest_teardown(item, nextitem) 在pytest_runtest_call(item)之后调用

pytest_runtest_makereport(item, call) 根据给定的pytest.Item和_pytest.runner.CallInfo返回一个_pytest.runner.TestReport对象。

可以在_pytest.runner中查看这些钩子的默认实现来加深理解。

22.4.4 Collection钩子

pytest_collection(session) 为整个测试执行collection规则。

参数：

- session(_pytest.main.Session)- pytest session 对象

pytest_ignore_collect(path, config) 返回True时会禁止在path目录下执行collection。在调用其他钩子前，会优先对所有文件及文件夹调用该钩子。

参数：

- path(str) - 待分析的目录
- config(_pytest.config.Config) - pytest config对象

pytest_collect_directory(path, parent) 在遍历一个文件夹做collection之前调用 参数：

- path(str) - 待分析的目录

pytest_collect_file(path, parent) 根据path返回collection节点或者None。任何新的节点都必须有指定的parent。

参数：

- path(str) - 待collect的目录

pytest_pycollect_makeitem(collector, name, obj) 返回自定义的item/collector。 说白了，就是定义哪些文件可以作为测试对象。

pytest_generate_tests(metafunc) 生成对测试函数的(多个)参数化调用。

pytest_make_parametrize_id(config, val, argname) 根据val返回一个用于@pytest.mark.parametrize的字符串表达式。如果钩子无法解析val，返回None。

参数：

- config(_pytest.config.Config) - pytest config对象
- val - 参数化的值
- argname (str) – pytest自动生成的参数名

collection完成后，可以使用下面的钩子修改项目顺序、删除或修改测试项目：

pytest_collection_modifyitems(session, config, items)

参数：

- session(_pytest.main.Session)- pytest session 对象

- config(_pytest.config.Config) - pytest config对象
- items (List[_pytest.nodes.Item]) - 测试对象列表

22.4.5 Reporting钩子

pytest_collectstart(collector) collector开始做collection的时候调用。

pytest_itemcollected(item) 找到了一个测试项

pytest_collectreport(report) collector完成collection的时候调用。

pytest_deselected(items) 通过关键字取消测试项时调用

pytest_report_header(config, startdir) 返回一个作为最终的终端输出的报告的header信息的字符串。

参数：

- config(_pytest.config.Config) - pytest config对象
- startdir - 作为起始目录的py.path对象

注意，因为pytest的插件查找机制，该函数只应该在测试的root文件夹下的插件及conftest.py中实现

pytest_report_collectionfinish(config, startdir, items)

3.2引入

返回一个字符串或者字符串列表，用于在collection成功完成后显示。

这里的字符串会在标准的"collected X items"之后显示。

参数：

- config(_pytest.config.Config) - pytest config对象
- startdir - 作为起始目录的py.path对象
- items - pytest待执行的测试项列表，这个列表不应该有任何改动
- pytest_report_teststatus(report, config)

返回测试报告的分类/短字母/细节(没用过。。。。)。

参数：

- config(_pytest.config.Config) - pytest config对象

pytest_terminal_summary(terminalreporter, exitstatus, config) 向测试概要报告中增加一个分段。

参数：

- terminalreporter (_pytest.terminal.TerminalReporter) – 内部使用的终端测试报告对象

- exitstatus (int) – 返回给操作系统的返回码
- config(_pytest.config.Config) - pytest config对象

pytest_fixture_setup(fixturedef, request)

执行fixture的setup

返回值为fixture函数的返回值。

如果fixture函数的返回值是None，那么该钩子函数的其他实现会被继续调用

pytest_fixture_post_finalizer(fixturedef, request) 在fixture的teardown之后调用，但是在清理cache之前，所以该钩子中fixturedef.cached_result的结果依然可以访问。

pytest_warning_captured(warning_message, when, item) pytest告警插件捕获到告警时调用。

参数：

- warning_message (warnings.WarningMessage) – 捕获的告警。
- when (str) – 告警捕获的时间，可能值为：
- “config”：pytest配置/初始化阶段
- “collect”：测试collection阶段
- “runtest”：测试运行阶段
- item (pytest.Item|None) – DEPRECATED强烈不推荐！！！

pytest_runtest_logreport(report) 处理测试的setup/call/teardown各个阶段的测试报告时调用。

pytest_assertrepr_compare(config, op, left, right) 为某些类型自定义断言表达式 返回失败的断言表达式的说明。 如果没有自定义，返回None，否则返回字符串列表。

- config(_pytest.config.Config) - pytest config对象

22.4.6 Debugging/Interaction钩子

pytest_internalerror(excrepr, excinfo) 内部发生错误时调用

pytest_keyboard_interrupt(excinfo) 键盘发生中断时调用

pytest_exception_interact(node, call, report) 在出现可能以交互方式处理的异常时调用。 该钩子只有在发生非内部异常(比如skip.Exception就是内部异常)时调用

pytest_enter_pdb(config, pdb) 调用pdb.set_trace()时调用，可以用在插件中，用于在进入python调试器进入交互模式前做一些特殊处理。

参数：

- config(_pytest.config.Config) - pytest config对象
- pdb(pdb.Pdb) - Pdb实例

22.5 对象Objects

没啥好翻译的。。。。当手册查吧

22.6 特殊变量

测试module中，一些全局变量会被特殊处理：

22.6.1 pytest_plugins

在测试modules或者conftest.py中用来注册全局的新增的插件。可以是一个字符串或者字符串序列：

```
pytest_plugins = "myapp.testsupport.myplugin"
pytest_plugins = ("myapp.testsupport.tools", "myapp.testsupport.regression")
```

22.6.2 pytest_mark

在测试modules用来新增全局的适用于所有函数及方法的marks。可以是一个mark或者marks序列：

```
import pytest
pytestmark = pytest.mark.webtest
```

```
import pytest
pytestmark = (pytest.mark.integration, pytest.mark.slow)
```

22.6.3 PYTEST_DONT_REWRITE(module docstring)

PYTEST_DONT_REWRITE可以在module的docstring中用于禁止断言重写

22.7 环境变量

环境变量可以改变pytest的行为。

22.7.1 PYTEST_ADDOPTS

用于定义展示给用户的命令行。

22.7.2 PYTEST_DEBUG

当设置该变量时，pytest会打印tracing以及调试信息。

22.7.3 PYTEST_PLUGINS

使用逗号分割的列表，表示需要加载的插件：

```
export PYTEST_PLUGINS=mymodule.plugin,xdist
```

22.7.4 PYTEST_DISABLE_PLUGIN_AUTOLOAD

当设置该变量了，禁止插件通过setuptools自动加载插件，仅允许加载显式声明的插件。

22.7.5 PYTEST_CURRENT_TEST

用户无需关心该变量，该变量是由pytest在内部用于设置当前测试的名称，以便其他进程可以检查。

22.8 配置项

内置的配置项可以位于pytest.ini/tox.ini/setup.cfg。所有配置项必须位于[pytest]段落中(setup.cfg中是在[tool:pytest])

注意：不推荐使用setup.cfg，可能会有问题

定义在文件中的配置项可以被命令行参数 -o/–override覆盖。格式为 name=value。示例如下：

```
pytest -o console_output_style=classic -o cache_dir=/tmp/mycache
```

addopts 添加命令行参数。 如下在ini中配置：

```
# pytest.ini
[pytest]
addopts = --maxfail=2 -rf # 发生两次失败时退出测试并上报错误信息
```

此时执行 pytest test_hello.py实际上是执行的下面的命令：

```
pytest --maxfail=2 -rf test_hello.py
```

cache_dir 3.2引入

设置缓存插件的目录。默认位于根目录下的.pytest_cache。

confcutdir ....。。。》》》

console_output_style

3.3引入 设置控制台输出格式：

- classic: 经典pytest格式
- progress: 类似pytest格式，带一个进度指示器
- count: 通过显示完成的测试数来取代原有的百分比显示。

默认是progress格式。 按照下面的方式可以重设该格式：

```
# pytest.ini
[pytest]
console_output_style = classic
```

doctest_encoding 3.1引入 用于用docstring解码文本文件的默认编码

doctest_optionflags 标准doctest模块里的一个或者多个doctest的标志。

empty_parameter_set_mark 3.4引入 允许在参数化中选择空参数集的操作：

- skip - 空参数集时跳过测试(默认)
- xfail - 空参数集时设置测试为xfail(run=False)
- fail_at_collect - 空参数集时抛出异常

pytest.ini

[pytest] empty_parameter_set_mark = xfail

注意新版本中有计划将默认值改为xfail，因为xfail看起来问题比较少。

---

filterwarnings 3.1引入

设置哪些告警会被显示。默认所有告警会在测试结束后显示。

```
# pytest.ini
[pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
```

改配置是告诉pytest忽略所有deprecation告警，并将其他告警转换为error。

junit_family junit_suite_name JUNIT。没用过

log_cli_date_format 3.3引入

格式化控制台输出的log的时间戳格式：

```
[pytest]
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

log_cli_format 3.3引入 格式化控制台输出的log的格式：

```
[pytest]
log_cli_format = %(asctime)s %(levelname)s %(message)s
```

log_cli_level 3.3引入 设置控制台输出的的log的最小的级别。可以是整数值或者level的名称

```
[pytest]
log_cli_level = INFO
```

log_date_format 3.3引入 格式化log的时间戳格式：

```
[pytest]
log_date_format = %Y-%m-%d %H:%M:%S
```

log_file 3.3引入 设置log的输出文件：

```
[pytest]
log_file = logs/pytest-logs.txt
```

log_file_date_format 3.3引入 格式化输出到log文件中的log的时间戳格式：

```
[pytest]
log_file_date_format = %Y-%m-%d %H:%M:%S
```

log_file_format 3.3引入 log文件中的log的格式：

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
```

log_file_level 3.3引入 设置log文件中的log的最小的级别。可以是整数值或者level的名称

```
[pytest]
log_file_level = INFO
```

logformat 3.3引入 log的格式：

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
```

log_level 3.3引入 log的最小的级别。可以是整数值或者level的名称

```
[pytest]
log_level = INFO
```

log_print 3.3引入 如果设置为False，会禁止显示失败的测试的log

```
[pytest]
log_print = False
```

markers 表示测试中可用的markers：

```
[pytest]
markers =
    slow
    serial
```

minversion 指定pytest不能小于某个版本号：

```
# pytest.ini
[pytest]
minversion = 3.0 # will fail if we run with pytest-2.8
```

norecursedirs ...

python_classes ...

python_files ...

python_functions ...

testpaths 2.8引入 指定从根目录下的哪个目录来执行，用于加快寻找测试用例和避免跑一些不想跑的用例。

```
[pytest]
testpaths = testing doc
```

pytest只会执行根目录下的testing和doc目录下的用例。

usefixtures 对所有测试函数生效的fixtures列表。定义在配置项中的fixture与在所有的函数上显式的标记@pytest.mark.usefixtures的效果是一样的：

```
[pytest]
usefixtures =
    clean_db
```

xfail_strict 如果设置为True，原来那些被标记为@pytest.mark.xfail的实际上是通过的用例都会被标记为失败。

```
[pytest]
xfail_strict = True
```