

Pytest官方文档翻译

安装及入门

Python支持版本: Python 2.6,2.7,3.3,3.4,3.5,Jython,PyPy-2.3

支持的平台: Unix/Posix and Windows

PyPI包名: pytest

依赖项: py,colorama (Windows)

PDF文档: 下载最新本本文档

Pytest是一个使创建简单及可扩展性测试用例变得非常方便的框架。测试用例清晰、易读而无需大量的繁琐代码。只要几分钟你就可以对你的应用程序或者库展开一个小型的单元测试或者复杂的功能测试。

安装 Pytest

在命令行执行以下命令

```
pip install -U pytest
```

检查安装的Pytest版本

```
$ pytest --version
This is pytest version 3.x.y, imported from $PYTHON_PREFIX/lib/python3.6/site-packages/pytest.py
```

创建你的第一个测试用例

只需要4行代码即可创建一个简单的测试用例:

```
# test_sample.py 文件内容
def func(x):
    return x + 1

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

就是这么简单。现在你可以执行一下这个测试用例:

```
$ pytest
===== test session starts =====
```

```
platform linux -- Python 3.x.y,pytest-3.x.y,py-1.x.y,pluggy-0.x.y
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:5: AssertionError
===== 1 failed in 0.12 seconds =====
```

由于`func(3)`并不等于`5`,这次测试返回了一个失败的结果信息。

注意： 你可以使用`assert`语句来断言你测试用例的期望结果。Pytest的高级断言内省机制, 可以智能地展示断言表达式的中间结果, 来避免来源于JUnit的方法中的变量名重复问题。

执行多条测试用例

`pytest`命令会执行当前目录及子目录下所有`test_*.py`及`*_test.py`格式的文件。一般来说,用例需要遵循标准的测试发现规则。

断言抛出了指定异常

使用`raise`可以在相应代码的抛出的指定异常:

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

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

使用“静默”模式,执行这个测试用例如:

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

使用类组织多条测试用例

一旦你需要开发多条测试用例,你可能会想要使用类来组织它们。使用Pytest可以很轻松的创建包含多条用例的测试类:

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

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

Pytest可以发现所有遵循Python测试用例发现约定规则的用例,所以它能找到Test开头的测试类外以及类中所有以test_开头的函数及方法。测试类无需再继承任何对象。我们只需要简单地通过文件名来运行这个模块即可。

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

self = <test_class.TestClass object at 0xdeadbeef>

    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 0.12 seconds
```

第一条用例执行成功,第二天用例执行失败。你可以很容易地通过断言中变量的中间值来理解失败的原因。

函数测试中请求使用独立的临时目录

Pytest提供了内置fixtures方法参数,来使用任意资源,比如一个独立的临时目录:

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

在测试用例函数使用`tmpdir`作为参数,Pytest将在测试用例函数调用之前查找并调用`fixture`工厂方法来创建相应的资源。在测试运行之前,Pytest为每个测试用例创建一个独立的临时目录:

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

tmpdir = local('PYTEST_TMPDIR/test_needsfiles0')

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

test_tmpdir.py:3: AssertionError
----- Captured stdout call -----
PYTEST_TMPDIR/test_needsfiles0
1 failed in 0.12 seconds
```

有关`tmpdir`处理的更多信息,请参见: 临时目录和文件

进一步阅读

查看其他pytest文档资源,来帮助你建立自定义测试用例及独特的工作流:

- “使用`pytest -m pytest`来调用`pyest`” - 命令行调用示例
- “将`pytest`与原有测试套件一起使用” - 使用之前的测试用例
- “使用属性标记测试用例” - `pytest.mark`相关信息
- “`pytest fixtures`: 显式,模块化,可扩展” - 为你的测试提供函数基准
- “插件编写” - 管理和编写插件
- “优质集成实践” - 虚拟环境和测试分层

Pytest 使用及调用方法

使用`python -m pytest`调用pytest

2.0版本新增 你可以在命令行中通过Python编译器来调用Pytest执行测试:

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

通过`python`调用会将当前目录也添加到`sys.path`中,除此之外,这几乎等同于命令行直接调用`pytest [...]`。

可能出现的执行退出code

执行`pytest`可能会出现6中不同的退出code:

- 退出**code 0**: 收集并成功通过所有测试用例

- 退出**code 1**: 收集并运行了测试,部分测试用例执行失败
- 退出**code 2**: 测试执行被用户中断
- 退出**code 3**: 执行测试中发生内部错误
- 退出**code 4**: pytest命令行使用错误
- 退出**code 5**: 没有收集到测试用例

获取版本路径、命令行选项及环境变量相关帮助

```
pytest --version    # 显示pytest导入位置
pytest --fixtures    # 显示可用的内置方法参数
pytest -h --help    # 显示命令行及配置文件选项帮助信息
```

第1(N)次失败后停止测试

在第1(N)次用例失败后停止测试执行:

```
pytest -x           # 第1次失败后停止
pytest --maxfail=2  # 2次失败后停止
```

指定及选择测试用例

Pytest支持多种从命令行运行和选择测试用例的方法。运行模块内所有用例

```
pytest test_mod.py
```

运行目录内所有用例

```
pytest testing/
```

按关键字表达式运行用例

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

这将运行包含与指定表达式匹配的名称的测试用例,其中可以包括文件名、类名和函数名作为变量,并且支持Python运算符(and和or)操作。上面的示例将运行TestMyClass.test_something但不运行TestMyClass.test_method_simple。

按节点id运行测试 每次执行收集到的测试用例集合都会被分配一个唯一的nodeid,其中包含模块文件名,后跟说明符,如类名、函数名及参数,由::字符分隔。执行模块中某条指定的测试用例如:

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

另一个通过命令行挑选所执行测试用例的示例如:

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

通过标记(**Mark**)表达式运行测试

```
pytest -m slow
```

这将会执行所有带`@pytest.mark.slow`装饰器的用例。

有关更多信息,请参阅: [标记](#)

从包中运行测试

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

这将会导入`pkg.testing`并使用其文件系统位置来查找和运行测试。

修改Python原生追溯(traceback)信息

修改回追溯信息示例如:

```
pytest --showlocals # 在追溯信息中显示局部变量
pytest -l           # 显示局部变量 (简写)

pytest --tb=auto    # (默认) 第1和最后1条使用详细追溯信息,其他使用简短追溯信息

pytest --tb=long    # 详尽,信息丰富的追溯信息格式
pytest --tb=short   # 简短的追溯信息格式
pytest --tb=line    # 每个失败信息一行
pytest --tb=native  # Python标准库格式
pytest --tb=no      # 不使用追溯信息
```

详尽的测试结果摘要

2.9版本新增 `-r` 标志可用于在测试会话结束时显示测试结果摘要,从而可以在拥有大量用例的测试套件中轻松获得所有失败、跳过、标记失败(xfails)等测试结果的清晰描述。 例如:

```
$ pytest -ra
===== test session starts =====
```

```
platform linux -- Python 3.x.y,pytest-3.x.y,py-1.x.y,pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR,inifile:
collected 0 items

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

`-r`选项接受后面的多个字符,上面使用的`a`表示“除了执行通过(Pass)以外所有的结果”。以下是可以使用的可用字符的完整列表:

`-f` - 失败的用例 `-E` - 出错的用例 `-s` - 跳过的用例 `-x` - 标记失败的用例 `-X` - 标记成功的用例 `-p` - 成功用例 `-P` - 成功用例并输出信息 `-a` - 所有`pP`状态以外的用例

可以使用多个字符,例如,只查看失败和跳过的用例,你可以执行:

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

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

执行失败时进入PDB(Python调试器)

Python附带一个名为PDB的内置Python调试器。pytest允许通过命令行选项进入PDB提示符:

```
pytest --pdb
```

这将在每次失败(或KeyboardInterrupt)时调用Python调试器。一般,你可能只希望在第一次失败的测试中执行此操作以了解某种故障情况:

```
pytest -x --pdb # 在第一次用例失败时进入PDB
pytest --pdb --maxfail=3 # 在前3次失败是进入PDB
```

注意,在任何失败时,异常信息都存储在`sys.last_value`,`sys.last_type`和`sys.last_traceback`中。在交互模式中,这允许用户使用任何调试工具进行事后调试。也可以手动访问异常信息,例如:

```
>>> import sys
>>> sys.last_traceback.tb_lineno
42
>>> sys.last_value
AssertionError('assert result == "ok"',)
```

测试开始时进入PDB(Python调试器)

`pytest`允许用户通过命令行选项在每次测试开始时立即进入PDB提示符:

```
pytest --trace
```

这将在每次测试开始时调用Python调试器。

设置断点

要在代码中设置断点,需要在代码中使用Python原生`import pdb; pdb.set_trace()`进行调用,Pytest会自动禁用显示`print`输出,并捕获该用例输出结果:

- 其他测试中的输出捕获不受影响。
- 任何先前的测试输出已经被捕获并将被处理。
- 在同一测试中生成的任何后续输出都不会被捕获,而是直接发送到`sys.stdout`。注意:即使是退出交互式PDB跟踪会话并继续常规测试后发生的测试输出,这也适用。

使用内置断点方法

Python 3.7引入了内置`breakpoint()`函数。Pytest支持以下几种使用`breakpoint()`的方式:

- 当`PYTHONBREAKPOINT`设置为默认值,调用`breakpoint()`时,pytest将使用其内部PDB跟踪交互界面(PDB trace UI)而不是Python自带的`pdb`。
- 测试完成后,默认会重置为Python自带的PDB跟踪交互界面。
- 在pytest后使用`--pdb`参数,在失败的测试/未处理异常中,pytest内部PDB跟踪交互界面与`breakpoint()`同时使用。
- `--pdbcls`参数可指定要使用的调试器类。

分析测试用例执行时间

显示执行最慢的10条测试用例如:

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

默认情况下,Pytest不会显示<0.01s的测试时间,除非在命令上传递`-vv`。

创建JUnit XML格式的测试报告

要创建可由Jenkins或其他持续集成软件读取的XML测试报告,可以使用:

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

运行结束后,在指定路径`path`下创建一个XML报告文件 3.1版本新增可以通过修改配置中`junit_suite_name`字段的名称来更改XML报告中`root test suite`的名称。


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

record_property(添加新属性) 版本2.8新增 版本3.5更改: 在所有报告生成器(reporter)中用户属性 `record_xml_property`项已改为`record_property`,`record_xml_property`现已弃用。可以使用 `record_property`项来在XML报告中增加更多的日志信息:

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

在生成的`testcase`标签是会添加一个额外的属性`example_key="1"`:

```
<testcase classname="test_function" file="test_function.py" line="0"
name="test_function" time="0.0009">
  <properties>
    <property name="example_key" value="1" />
  </properties>
</testcase>
```

或者,你可以将此函数集成在自定义标记装饰器中:

```
# conftest.py文件内容

def pytest_collection_modifyitems(session,config,items):
    for item in items:
        for marker in item.iter_markers(name="test_id"):
            test_id = marker.args[0]
            item.user_properties.append(("test_id",test_id))
```

在你的测试用例中使用:

```
# test_function.py文件内容
import pytest

@pytest.mark.test_id(1501)
def test_function():
    assert True
```

这将导致:

```
<testcase classname="test_function" file="test_function.py" line="0"
name="test_function" time="0.0009">
  <properties>
    <property name="test_id" value="1501" />
  </properties>
</testcase>
```

警告： `record_property` 是一个实验性函数,将来可能会发生变化。另外,这将破坏一些XML结构验证,与某些持续集成软件一起使用时,可能会导致一些问题。

record_xml_attribute(修改xml节点属性) 3.4版本新增可以使用 `record_xml_attribute fixture` 向 `testcase` 标签中添加其他xml属性。也可以用来覆盖原有属性值:

```
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, 它不会在节点下添加子元素,而是在生成的 `testcase` 标签内添加一个属性 `assertions = "REQ-1234"`,并使用 `classname = custom_classname` 覆盖默认的 `classname` 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>
```

警告： `record_xml_attribute` 也是一个实验性函数,其界面可能会被更强大,更通用的未来版本所取代。但是,将保留函数本身。

通过使用 `record_xml_property` 可以为在使用持续集成工具解析xml报告时提供帮助。但是,一些解析器对允许的元素和属性非常严格。许多工具使用 `xsd` 模式(如下例所示)来验证传入的xml。确保使用解析器允许的属性名称。

以下是Jenkins用于验证xml报告的结构:

```
<xs:element name="testcase">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="skipped" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="error" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="failure" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="system-out" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="system-err" minOccurs="0" maxOccurs="unbounded"/>
```

```

    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="assertions" type="xs:string" use="optional"/>
    <xs:attribute name="time" type="xs:string" use="optional"/>
    <xs:attribute name="classname" type="xs:string" use="optional"/>
    <xs:attribute name="status" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

```

LogXML: add_global_property 3.0版本新增 如果要在`testsuite`级别添加属性节点,该节点可能包含与所有测试用例相关的属性,则可以使用`LogXML.add_global_properties`

```

import pytest

@pytest.fixture(scope="session")
def log_global_env_facts(f):

    if pytest.config.pluginmanager.hasplugin("junitxml"):
        my_junit = getattr(pytest.config, "_xml", None)

        my_junit.add_global_property("ARCH", "PPC")
        my_junit.add_global_property("STORAGE_TYPE", "CEPH")

    @pytest.mark.usefixtures(log_global_env_facts.__name__)
    def start_and_prepare_env():
        pass

    class TestMe(object):
        def test_foo(self):
            assert True

```

这会在生成的xml中的`testsuite`节点下的属性节中添加:

```

<testsuite errors="0" failures="0" name="pytest" skips="0" tests="1" time="0.006">
  <properties>
    <property name="ARCH" value="PPC"/>
    <property name="STORAGE_TYPE" value="CEPH"/>
  </properties>
  <testcase classname="test_me.TestMe" file="test_me.py" line="16" name="test_foo"
time="0.000243663787842"/>
</testsuite>

```

警告： 这依然是一个实验性的函数,其界面也可能被更强大,更通用的未来版本所取代,但也将保留该函数。

创建结果日志格式文件

3.0版本之后不推荐使用,计划在4.0版本中删除。

对于仍然需要类似函数的用户来说,可以使用提供测试数据流的`pytest-tap`插件。

如有任何疑虑,可以[建立一个问题(open an issue)](https://github.com/pytest-dev/pytest/issues/new)。

```
pytest --resultlog=path
```

执行后,在`path`路径中会创建一个纯文本结果日志文件,这些文件可以用于: 例如,在PyPy-test网页显示多个修订版的测试结果。

将测试报告发送到在线pastebin服务

为每条测试失败用例建立一个日志URL链接:

```
pytest --pastebin=failed
```

这会将测试运行信息提交到一个提供粘贴服务的远程服务器上,并为每条测试失败用例提供一个URL。你可以像平常一样查看搜集结果,或者使用`-x`参数,来只显示某个特定的测试失败结果。

为整个测试执行日志建立一个URL链接:

```
pytest --pastebin=all
```

目前只实现了粘贴到`http://bpaste.net`网站的服务。

禁用插件

可以通过`-p`选项与前缀`no:`一起使用,来在运行时禁用加载特定插件。

例如: 要禁用加载从文本文件执行doctest测试的`doctest`插件,可以通过以下方式运行Pytest:

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

在Python代码调用pytest

版本2.0新增 你可以在Python代码中直接调用pytest:

```
pytest.main()
```

这就和你从命令行调用“pytest”一样。但它不会引发`SystemExit`,而是返回`exitcode`。你可以传入选项和参数。

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

你可以为`pytest.main`指定其他插件:

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

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

运行它将显示已添加`MyPlugin`并调用其中的`hook`方法:

```
$ python myinvoke.py
.
*** test run reporting finishing [100%]
```

注意: 调用`pytest.main()`将会导入所有测试用例及其导入的其他模块。由于python导入系统的缓存机制,从同一进程后续调用`pytest.main()`不会反映调用之间对这些文件的更改。因此,不建议从同一进程(例如,为了新运行测试)多次调用`pytest.main()`。

原有TestSuite的执行方法

Pytest可以与大多数现有的测试套件(testsuite)一起使用,但是它的加载方式方式不像nose或Python的默认单元测试框架的测试运行器(test runner)。

在使用本节之前,你需要安装pytest。

使用pytest运行已存在的测试套件(test suite)

假设你想要在某个地方为现有仓库(respsitory)做贡献代码。 在使用某种版本控制软件拉取代码和设置完`virtualenv` (可选)后,你需要运行:

```
cd <仓库名>
pip install -e . # 环境所依赖的'python setup.py develop' 和 'conda develop'包
```

在你项目根目录中,这将为你的代码在`site-packages`中设置一个符号链接,来允许你无需安装自己的代码即可执行测试。

在开发模式下如此使用,可以避免每次要运行测试时重新安装,这比每次使用`sys.path`将测试指向本地代码更简单。

或者你可以考虑使用`[tox`。

译者注: 实际官方并没有写Pytest怎么执行TestSuite,执行方法可以参考个人的另一篇文章:

[<https://www.jianshu.com/p/6a05ccd3ca94>]

断言的编写和报告

使用assert语句进行断言

pytest允许你使用标准的Python`assert`断言语句来验证测试中的期望结果和实际结果。例如,你可以编写以下内容:

```
# test_assert1.py 文件内容
def f():
    return 3

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

来断言你的函数返回一个特定的值。如果此断言失败,你将看到函数调用的返回值:

```
$ pytest test_assert1.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, 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 0.12 seconds =====
```

Pytest支持显示常见的包括调用,属性,比较以及二元和一元运算符子表达式的值 (参考: [pytest执行Python测试失败报告示例](#))。 你可以在不使用繁琐的Python惯用构造样板代码的同时,不丢失断言失败的对比信息(内省信息)。

当然,你也可以像下面所示,指定断言失败的返回消息:

```
assert a % 2 == 0, "值为奇数,应为偶数"
```

这样将不会断言失败对比信息(内省信息),而只简单地在追溯信息中显示你指定的失败返回信息。有关断言内省的更多信息,请参阅[高级断言内省](#)。

异常断言

你可以像如下所示,使用`pytest.raises`作为上下文管理器来进行异常断言:

```
import pytest

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

如果需要访问实际的异常信息,你可以使用:

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

`excinfo`是一个`ExceptionInfo`实例,它是实际异常的装饰器。其主要属性有`.type`、`.value`及`.traceback`三种。版本3.0已修改在上下文管理器中,你可以使用参数`message`来指定自定义失败信息:

```
>>> with raises(ZeroDivisionError,message="Expecting ZeroDivisionError"):
...     pass
... Failed: Expecting ZeroDivisionError
```

如果你想编写适用于Python 2.4的测试代码,你还可以使用其他两种方法来测试预期的异常:

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

两者都可以对带任意参数的函数断言是否出现了期望的异常: `ExpectedException`。即使没有异常或出现了不同的异常,报告生成器也能输出一些有用的断言信息。

注意,也可以为`pytest.mark.xfail`指定一个“raises”参数,当引发异常时标记用例失败:

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

对于你在代码中故意设置的异常,使用`pytest.raises`断言更加好用,而将`@ pytest.mark.xfail`与`check`函数一起使用对于已知未修复或依赖中的bug会更好。

此外,上下文管理器接受`match`关键字参数来测试正则表达式匹配中的异常(如`unittest`中的`TestCase.assertRaisesRegex`方法):

```
import pytest

def myfunc():
    raise ValueError("Exception 123 raised")

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

`match`变量后的正则表达式与使用`re.search`函数来进行匹配一致。因此在上面的例子中,`match = '123'`不会引发异常。

警示断言

2.8版本新增 你可以使用`pytest.warns`检查代码是否引发了特定警告。

使用上下文对比

2.0版本新增 `Pytest`可以在断言的比较中提供丰富的上下文信息。例如:

```
# test_assert2.py 文件内容

def test_set_comparison():
    set1 = set("1308")
    set2 = set("8035")
    assert set1 == set2
```

当你运行这个模块后

```
$ pytest test_assert2.py
===== test session starts =====
platform linux -- Python 3.x.y,pytest-3.x.y,py-1.x.y,pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR,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:5: AssertionError
===== 1 failed in 0.12 seconds =====

```

对大量用例进行了特定对比:

- 长字符串断言: 显示上下文差异
- 长序列断言: 显示第一个失败的索引
- 字典断言: 显示不同的key-value对

有关更多示例,请参阅 报告样例。

自定义断言对比信息

可以通过实现hook方法`pytest_assertrepr_compare`来在断言结果中添加你自己的详细说明信息。

****pytest_assertrepr_compare(config,op,left,right)***- [源码] 返回失败断言表达式中的对比信息。

如果没有自定义对比信息,则返回None,否则返回一系列字符串。字符串将由换行符连接,但字符串中的任何换行符都将被转义。 请注意,除第一行外的所有行都将略微缩进,目的是将第一行作为摘要。 **参数:**

config(`pytest.config.Config*` - `pytest config` 对象 例如,在`conftest.py`文件中添加以下钩子(Hook)方法,可以为`Foo`对象提供了附加对比信息:

```

# conftest.py内容
from test_foocompare import Foo
def pytest_assertrepr_compare(op,left,right):
    if isinstance(left,Foo) and isinstance(right,Foo) and op == "==":
        return ['Foo实例对比:',
                '   值: %s != %s' % (left.val,right.val)]

```

现在,在测试模块使用

```

# test_foocompare.py内容
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
```

运行这个测试模块你可以看到`conftest.py`文件中定义的自定义输出:

```
$ pytest -q test_foocompare.py
F [100%]
===== FAILURES =====
_____ test_compare _____

    def test_compare():
        f1 = Foo(1)
        f2 = Foo(2)
>       assert f1 == f2
E       assert Foo实例对比:
E           值: 1 != 2

test_foocompare.py:11: AssertionError
1 failed in 0.12 seconds
```

高级断言内省

2.1版本新函数 报告有关失败断言的详细信息是通过在运行之前重写`assert`语句来实现的。重写的断言语句将内省信息放入断言失败消息中。**Pytest**只重写测试收集过程直接发现的测试模块中的`assert`断言,因此在支持模块(非测试模块)中的断言,不会被重写。

你可以在导入模块前通过调用`register_assert_rewrite`手动启用断言重写(比如可以在`conftest.py`这样使用)。

注意 **Pytest**通过使用导入hook方法写入新的`.pyc`文件来重写测试模块。通常这种结构比较清晰。但是,如果你混乱导入,导入的hook方法可能会受到干扰。如果是这种情况,你有两种选择: 通过将字符串 `PYTEST_DONT_REWRITE` 添加到其docstring来禁用特定模块的重写。使用 `--assert = plain` 禁用所有模块的重写。此外,如果无法写入新的`.pyc`文件(如在只读文件系统或zip文件中),重写将无提示失败。有关进一步的信息,请参阅: 本杰明彼得森写的[pytest的新断言改写的幕后故事]。

版本2.1新函数: 添加断言重写作为备用内省技术。 **版本2.1更改:** 引入`--assert`选项。弃用`--no-assert`和`--nomagic`。 **版本3.0版更改:** 删除`--no-assert`和`--nomagic`选项。 删除`--assert = reinterp``选项。

Pytest fixtures: 清晰 模块化 易扩展

2.0/2.3/2.4版本新函数 `text fixtures`的目的是为测试的重复执行提供一个可靠的固定基线。 `pytest fixture`比经典的 `xUnit setUp/tearDown`方法有着显著的改进:

- fixtures具有明确的名称,在测试用例/类/模块或整个项目中通过声明使用的fixtures名称来使用。
- fixtures以模块化方式实现,因为每个fixture名称都会触发调用fixture函数,该fixture函数本身可以使用其它的fixtures。

- 从简单的单元测试到复杂的函数测试,fixtures的管理允许根据配置和组件选项对fixtures和测试用例进行参数化,或者在测试用例/类/模块或整个测试会话范围内重复使用该fixture。

此外,pytest继续支持经典的xUnit风格的setup方法。 你可以根据需要混合使用两种样式,逐步从经典样式移动到新样式。 你也可以从现有的unittest.TestCase样式或基于nose的项目开始。

Fixtures作为函数参数使用

测试用例可以通过在其参数中使用fixtures名称来接收fixture对象。 每个fixture参数名称所对应的函数,可以通过使用`@pytest.fixture`注册成为一个fixture函数,来为测试用例提供一个fixture对象。 让我们看一个只包含一个fixture和一个使用它的测试用例的简单独立测试模块:

```
# ./test_smtpsimple.py内容
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 # for demo purposes
```

这里,test_ehlo需要smtp_connection来提供fixture对象。 pytest将发现并调用带`@pytest.fixture`装饰器的smtp_connection fixture函数。 运行测试如下所示:

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

在测试失败的回溯信息中,我们看到测试用例是使用`smtp_connection`参数调用的,即由`fixture`函数创建的`smtplib.SMTP()`实例。测试用例在我们故意的`assert 0`上失败。以下是`pytest`用这种方式调用测试用例使用的确切协议:

Fixtures: 依赖注入的主要例子

`Fixtures`允许测试用例能轻松引入预先定义好的初始化准备函数,而无需关心导入/设置/清理方法的细节。这是依赖注入的一个主要示例,其中`fixture`函数的函数扮演“注入器”的角色,测试用例来“消费”这些`fixture`对象。

`conftest.py`: 共享`fixture`函数

如果在测试中需要使用多个测试文件中的`fixture`函数,则可以将其移动到`conftest.py`文件中,所需的`fixture`对象会自动被`Pytest`发现,而不需要再每次导入。`fixture`函数的发现顺序从测试类开始,然后是测试模块,然后是`conftest.py`文件,最后是内置和第三方插件。

你还可以使用`conftest.py`文件来实现本地每个目录的插件。

共享测试数据

如果要使用数据文件中的测试数据,最好的方法是将这些数据加载到`fixture`函数中,以供测试用例注入使用。这利用到了`pytest`的自动缓存机制。

另一个好方法是在`tests`文件夹中添加数据文件。还有社区插件可用于帮助处理这方面的测试,例如: `pytest-datadir`和`pytest-datafiles`。

生效范围: 在测试类/测试模块/测试会话中共享`fixture`对象

由于`fixtures`对象需要连接形成依赖网,而通常创建时间比较长。扩展前面的示例,我们可以在`@pytest.fixture`调用中添加`scope = "module"`参数,以使每个测试模块只调用一次修饰的`smtp_connection` `fixture`函数(默认情况下,每个测试函数调用一次)。因此,测试模块中的多个测试用例将各自注入相同的`smtp_connection` `fixture`对象,从而节省时间。`scope`参数的可选值包括: `function`(函数),`class`(类),`module`(模块),`package`(包)及 `session`(会话)。

下一个示例将`fixture`函数放入单独的`conftest.py`文件中,以便来自目录中多个测试模块的测试可以访问`fixture`函数:

```
# conftest.py 文件内容
import pytest
import smtplib

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

`fixture`对象的名称依然是`smtp_connection`,你可以通过在任何测试用例或`fixture`函数(在`conftest.py`所在的目录中或下面)使用参数`smtp_connection`作为输入参数来访问其结果:

```
# test_module.py文件内容

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

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

我们故意插入失败的`assert 0`语句,以便检查发生了什么,运行测试并查看结果:

```
$ 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 = <smtpplib.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 = <smtpplib.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 =====
```

你会看到两个`assert 0`失败信息,更重要的是你还可以看到相同的(模块范围的)`smtp_connection`对象被传递到两个测试用例中,因为`pytest`在回溯信息中显示传入的参数值。因此,使用`smtp_connection`的两个测试用例运行速度与单个函数一样快,因为它们重用了相同的`fixture`对象。

如果你决定要使用`session`(会话,一次运行算一次会话)范围的`smtp_connection`对象,则只需如下声明:

```
@pytest.fixture(scope="session")
def smtp_connection():
    # the returned fixture value will be shared for
    # all tests needing it
    ...
```

最后,`class`(类)范围将为每个测试类调用一次`fixture`对象。

注意: `Pytest`一次只会缓存一个`fixture`实例。这意味着当使用参数化`fixture`时,`pytest`可能会在给定范围内多次调用`fixture`函数。

`package`(包)范围的`fixture`(实验性函数) **3.7版本新函数** 在`pytest 3.7`中,引入了包范围。当包的最后一次测试结束时,最终确定包范围的`fixture`函数。

警告: 此函数是实验性的,如果在获得更多使用后发现隐藏的角落情况或此函数的严重问题,可能会在将来的版本中删除。

谨慎使用此新函数,请务必报告你发现的任何问题。

高范围的`fixture`函数优先实例化

3.5版本新函数 在测试函数的`fixture`对象请求中,较高范围的`fixture`(例如`session`会话级)较低范围的`fixture`(例如`function`函数级或`class`类级)优先执行。相同范围的`fixture`对象的按引入的顺序及`fixtures`之间的依赖关系按顺序调用。

请考虑以下代码:

```
@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):  
    ...
```

`test_foo`中fixtures将按以下顺序执行:

1. `s1`: 是最高范围的fixture(会话级)
2. `m1`: 是第二高的fixture(模块级)
3. `tmpdir`: 是一个函数级的fixture,`f1`依赖它,因此它需要在`f1`前调用
4. `f1`: 是`test_foo`参数列表中第一个函数范围的fixture。
5. `f2`: 是`test_foo`参数列表中最后一个函数范围的fixture。

fixture结束/执行teardown代码

当fixture超出范围时,通过使用`yield`语句而不是`return`,pytest支持fixture执行特定的teardown代码。`yield`语句之后的所有代码都视为teardown代码:

```
# conftest.py 文件内容  
  
import smtplib  
import pytest  
  
@pytest.fixture(scope="module")  
def smtp_connection():  
    smtp_connection = smtplib.SMTP("smtp.gmail.com",587,timeout=5)  
    yield smtp_connection # provide the fixture value  
    print("teardown smtp")  
    smtp_connection.close()
```

无论测试的异常状态如何,`print`和`smtp.close()`语句将在模块中的最后一个测试完成执行时执行。

让我们执行一下(上文的`test_module.py`):

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

我们看到`smtp_connection`实例在两个测试完成执行后完成。 请注意,如果我们使用`scope = 'function'`修饰我们的fixture函数,那么每次单个测试都会进行fixture的setup和teardown。 在任何一种情况下,测试模块本身都不需要改变或了解fixture函数的这些细节。

请注意,我们还可以使用`with`语句无缝地使用`yield`语法:

```
# test_yield2.py 文件内容  
  
import smtplib
```

```
import pytest

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

测试结束后,smtp_connection连接将关闭,因为当with语句结束时,smtp_connection对象会自动关闭。

请注意,如果在设置代码期间(yield关键字之前)发生异常,则不会调用teardown代码(在yield之后)。执行teardown代码的另一种选择是利用请求上下文对象的addfinalizer方法来注册teardown函数。 以下是smtp_connection fixture函数更改为使用addfinalizer进行teardown:

```
# content of 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 # provide the fixture value
```

yield和addfinalizer方法在测试结束后调用它们的代码时的工作方式类似,但addfinalizer相比yield有两个主要区别:

1. 使用addfinalizer可以注册多个teardown函数。
2. 无论fixture中setup代码是否引发异常,都将始终调用teardown代码。即使其中一个资源无法创建/获取,也可以正确关闭fixture函数创建的所有资源:

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

在上面的示例中,如果“C28”因异常而失败,则“C1”和“C3”仍将正确关闭。当然,如果在注册finalize函数之前发生异常,那么它将被不会执行。

Fixtures中使用测试上下文的内省信息

Fixture函数可以接受request对象来内省“请求”测试函数,类或模块上下文。进一步扩展前一个smtp_connectionfixture例子,让我们从使用我们的fixture的测试模块中读取一个可选的服务器URL:

```
# content of 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()
```

我们使用该request.module属性可选地smtpserver从测试模块获取 属性。如果我们再次执行,那么没有太大变化:

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

2 failed in 0.12 seconds
```

让我们快速创建另一个测试模块,该模块实际上在其模块命名空间中设置服务器URL:

```
# content of test_anothersmtp.py

smtpserver = "mail.python.org" # will be read by smtp fixture

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

运行它:

```
$ 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！该`smtp_connectionFixture`方法函数从模块命名空间拿起我们的邮件服务器名称。

Fixtures工厂方法

“工厂作为Fixture方法”模式可以在单个测试中多次需要Fixture方法结果的情况下提供帮助。而不是直接返回数据,而是返回一个生成数据的函数。然后可以在测试中多次调用此函数。

Fixtures工厂方法可根据需要提供参数生成Fixture和方法:

```
@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")
```

如果工厂创建的数据需要管理,那么Fixture方法可以处理:

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

    created_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")
```

Fixtures参数化

可以对Fixture方法函数进行参数化,在这种情况下,它们将被多次调用,每次执行一组相关测试,即依赖于此Fixture方法的测试。测试函数通常不需要知道它们的重新运行。Fixture方法参数化有助于为可以以多种方式配置的组件编写详尽的函数测试。

扩展前面的示例,我们可以标记Fixture方法以创建两个smtp_connectionFixture方法实例,这将导致使用Fixture方法的所有测试运行两次。fixture函数通过特殊request对象访问每个参数:

```
# content of 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()
```

主要的变化是paramswith 的声明@pytest.fixture,一个值列表,每个值的Fixture方法函数将执行,并可以通过访问值request.param。没有测试函数代码需要更改。那么让我们再做一次:

```
$ 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 = <smtpplib.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\n8BITIME\nDSN\nSMTPUTF8\nCHUNKING'

test_module.py:5: AssertionError
----- Captured stdout setup -----
finalizing <smtpplib.SMTP object at 0xdeadbeef>
_____ test_noop[mail.python.org] _____

smtp_connection = <smtpplib.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 <smtpplib.SMTP object at 0xdeadbeef>
4 failed in 0.12 seconds

```

我们看到我们的两个测试函数分别针对不同的smtp_connection实例运行了两次。另请注意,对于mail.python.org 连接,第二个测试失败,test_ehlo因为预期的服务器字符串不同于发送到服务器字符串。

Pytest将建立一个字符串,它是用于在参数化Fixture方法,例如每个器材值测试ID test_ehlo[smtp.gmail.com]和test_ehlo[mail.python.org]在上述实施例。这些ID可用于-k选择要运行的特定案例,并且还可以在失败时识别特定案例。运行pytest --collect-only将显示生成的ID。

数字,字符串,布尔值和None将在测试ID中使用它们通常的字符串表示形式。对于其他对象,Pytest将根据参数名称生成一个字符串。可以使用ids关键字参数自定义测试ID中用于特定Fixture方法值的字符串:

```

# content of 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
```

上面显示了如何ids使用要使用的字符串列表或将使用fixture值调用的函数,然后必须返回要使用的字符串。在后一种情况下,如果函数返回,None则将使用pytest的自动生成的ID。

运行上述测试会导致使用以下测试ID:

```
$ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y,pytest-5.x.y,py-1.x.y,pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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 =====
```

使用参数化fixtures标记

pytest.param()可用于在参数化Fixture方法的值集中应用标记,其方式与@ pytest.mark.parametrize一样。

例如:

```
# content of 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
```

运行此测试将跳过data_set带值的调用2:

```
$ pytest test_fixture_marks.py -v
===== test session starts =====
platform linux -- Python 3.x.y,pytest-5.x.y,py-1.x.y,pluggy-0.x.y --
$PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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 =====
```

模块化: 在fixture函数中使用fixtures函数

你不仅可以在测试函数中使用Fixture方法,而且Fixture方法函数可以自己使用其他Fixture方法。这有助于你的Fixture方法的模块化设计,并允许在许多项目中重复使用特定于框架的Fixture方法。作为一个简单的例子,我们可以扩展前面的例子并实例化一个对象app,我们将已经定义的smtp_connection资源粘贴 到它中:

```
# content of 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
```

这里我们声明一个appfixture,它接收先前定义的 smtp_connectionfixture并App用它实例化一个对象。我们来吧:

```
$ pytest -v test_appsetup.py
===== test session starts =====
platform linux -- Python 3.x.y,pytest-5.x.y,py-1.x.y,pluggy-0.x.y --
$PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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 =====
```

由于参数化smtp_connection,测试将使用两个不同的App实例和相应的smtp服务器运行两次。没有必要为appFixture方法要意识到的smtp_connection 参数化,因为pytest将全面分析Fixture方法依赖关系图。

请注意,appFixture方法具有范围module并使用模块范围的smtp_connectionFixture方法。如果smtp_connection缓存在session范围上,该示例仍然可以工作: Fixture方法使用“更广泛”的范围Fixture方法,但不是相反的方式: 会话范围的Fixture方法不能以有意义的方式使用模块范围的Fixture方法。

使用fixture实例自动组织测试用例

pytest在测试运行期间最小化活动Fixture方法的数量。如果你有一个参数化Fixture方法,那么使用它的所有测试将首先用一个实例执行,然后在创建下一个Fixture方法实例之前调用终结器。除此之外,这还可以简化对创建和使用全局状态的应用程序的测试。

以下示例使用两个参数化Fixture方法,其中一个基于每个模块作用域,并且所有函数执行print调用以显示设置/拆卸流程:

```
# content of 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))
```

让我们以详细模式运行测试并查看打印输出：

```
$ pytest -v -s test_module.py
===== test session starts =====
platform linux -- Python 3.x.y,pytest-5.x.y,py-1.x.y,pluggy-0.x.y --
$PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... 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 0.12 seconds =====
```

你可以看到参数化模块范围的`modarg`资源导致测试执行的排序,从而导致尽可能少的“活动”资源。`mod1`参数化资源的终结器在`mod2`资源建立之前执行。

特别注意`test_0`是完全独立的并且首先完成。然后执行`mod1test_1 mod1`,然后执行`test_2`,然后执行`test_1,mod2`最后执行`test_2 mod2`。

该`otherarg`参数化资源(其函数范围)是之前设置和使用它的每一个测试后撕开了下来。

在类/模块/项目中使用fixtures

有时,测试函数不需要直接访问Fixture方法对象。例如,测试可能需要使用空目录作为当前工作目录,但不关心具体目录。以下是如何使用标准`tempfile`和`pytest fixture`来实现它。我们将`fixture`的创建分成`conftest.py`文件:

```
# content of conftest.py

import pytest
import tempfile
import os

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

并通过`usefixtures`标记在测试模块中声明它的使用方法:

```
# content of 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()) == []
```

由于`usefixtures`标记,`cleandir`每个测试用例的执行都需要Fixture方法,就像为每个测试用例指定一个“`cleandir`”函数参数一样。让我们运行它来验证我们的Fixture方法是否已激活且测试通过:

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

你可以像这样指定多个Fixture方法:

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

你可以使用标记机制的通用函数在测试模块级别指定Fixture方法使用情况:

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

请注意,必须调用指定的变量`pytestmark`,分配例如 `foomark`不会激活Fixture方法。

也可以将项目中所有测试所需的Fixture方法放入ini文件中:

```
content of pytest.ini
[pytest]
usefixtures = cleandir
```

警告 请注意,此标记对Fixture方法函数没有影响。例如,这将无法按预期工作:

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

目前,这不会产生任何错误或警告,但这应由 #3664 处理。

自动使用fixtures(xUnit 框架的setup固定方法)

有时,你可能希望自动调用fixture,而无需显式声明函数参数或使用`usefixtures`装饰器。作为一个实际的例子,假设我们有一个数据库fixture,它有一个开始/回滚/提交架构,我们希望通过事务和回滚自动包围每个测试用例。以下是这个想法的虚拟自包含实现:

```
# content of 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"]

```

类级别的`transactfixture`用`autouse = true`标记, 这意味着类中的所有测试用例都将使用此`fixture`而无需在测试函数签名中或使用类级`usefixtures`装饰器进行陈述。

如果我们运行它,我们得到两个通过测试:

```

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

```

以下是`autouseFixture`方法在其他范围内的工作原理:

- `autouse fixtures`服从`scope=`关键字参数: 如果`autouse fixture`具有`scope='session'`它,它将只运行一次,无论它在何处定义。`scope='class'`意味着它将每班运行一次,等等。
- 如果在测试模块中定义了`autouse fixture`,则其所有测试函数都会自动使用它。
- 如果在`conftest.py`文件中定义了`autouse fixture`,那么其目录下所有测试模块中的所有测试都将调用`fixture`。
- 最后,请谨慎使用: 如果你在插件中定义了`autouse fixture`,则会在安装插件的所有项目中为所有测试调用它。如果`Fixture`方法仅在任何情况下在某些设置(例如`ini`文件中)的情况下工作,则这可能是有用的。这样的全局`Fixture`方法应该总是快速确定它是否应该做任何工作并避免昂贵的进口或计算。

请注意,上述`transactFixture`方法很可能是你希望在项目中可用的`Fixture`方法,而不是通常处于活动状态。规范的方法是将`transact`定义放入`conftest.py`文件中,而不使用`autouse`:

```

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

```

然后让一个`TestClass`通过声明需要使用它:

```

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

```

此TestClass中的所有测试用例都将使用事务Fixture方法,而模块中的其他测试类或函数将不使用它,除非它们还添加transact引用。

不同级别的fixtures的覆盖(优先级)

相对于在较大范围的测试套件中的Test Fixtures方法,在较小范围子套件你可能需要重写和覆盖外层的Test Fixtures方法,从而保持测试代码的可读性和可维护性。

在文件夹级别(通过conftest文件)重写fixtures方法

假设用例目录结构为:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

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

  test_something.py
    # content of tests/test_something.py
    def test_username(username):
        assert username == 'username'

  subfolder/
    __init__.py

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

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

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

你可以看到,基础/上级fixtures方法可以通过子文件夹下的conftest.py中同名的fixtures方法覆盖,非常简单,只需要按照上面的例子使用即可。

在测试模块级别重写fixtures方法

假设用例文件结构如下:

```
tests/
  __init__.py

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

  test_something.py
    # content of 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
    # content of 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'
```

上面的例子中,用例模块(文件)中的`fixture`方法会覆盖文件夹`conftest.py`中同名的`fixtures`方法

在直接参数化方法中覆盖`fixtures`方法

假设用例文件结构为:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

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

    @pytest.fixture
    def other_username(username):
```

```

        return 'other-' + username

test_something.py
# content of 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'

```

在上面的示例中,username fixture方法的结果值被参数化值覆盖。 请注意,即使测试不直接使用(也未在函数原型中提及),也可以通过这种方式覆盖fixture的值。

使用非参数化**fixture**方法覆盖参数化**fixtures**方法,反之亦然

假设用例结构为:

```

tests/
__init__.py

conftest.py
# content of 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
# content of 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
# content of 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'
```

在上面的示例中,使用非参数化fixture方法覆盖参数化fixture方法,以及使用参数化fixture覆盖非参数化fixture以用于特定测试模块。这同样适用于文件夹级别的fixtures方法

使用Marks标记测试用例

通过使用`pytest.mark`你可以轻松地在测试用例上设置元数据。例如,一些常用的内置标记:

- `skip` - 始终跳过该测试用例
- `skipif` - 遇到特定情况跳过该测试用例
- `xfail` - 遇到特定情况,产生一个“期望失败”输出
- `parametrize` - 在同一个测试用例上运行多次调用(译者注: 参数化数据驱动)

创建自定义标记或将标记应用于整个测试类或模块很容易。文档中包含有关标记的示例,详情可参阅[使用自定义标记]。

注意: 标记只对测试用例有效,对fixtures方法无效。

在未知标记上引发异常: `-strict`

当使用`--strict`命令行参数时,未在`pytest.ini`文件中注册的任何标记都将引发异常。

标记可以通过以下方式注册:

```
[pytest]
markers =
    slow
    serial
```

这可用于防止用户意外输错标记名称。想要强制执行此操作的测试套件应将`--strict`添加到`addopts`:

```
[pytest]
addopts = --strict
markers =
    slow
    serial
```

标记改造和迭代

3.6版本新函数 `pytest` 的标记传统地实现是通过简单地在测试函数的 `__dict__` 中添加属性来进行标记。结果,标记意外的随着类的集成而传递。此外,使用 `@pytest.mark` 装饰器应用的标记和通过 `node.add_marker` 添加的标记存储的位置不同,用于检索它们的API也不一致。

这样,如果不深入了解测试代码内部结构,技术上几乎无法正确使用参数化数据,从而导致在高级的使用方法中出现细微且难以理解的bug。

根据标记声明/更改的方式,你都可以获得一个 `MarkerInfo` 对象,其中也可能会包含来自同级类的标记。当使用参数化标记,或 `node.add_marker` 时,会丢弃之前的使用装饰器声明的 `MarkDecorators` 标记。`MarkerInfo` 对象实际上是使用同一标记名的多个标记的合并视图,当然,`MarkerInfo` 也可以像单个标记一样使用。

最重要的是,即使标记是在类/模块上声明的,实际上,标记只能在函数中访问。原因是模块,类和函数/方法无法以相同的方式访问标记。

在 `pytest 3.6` 版本中引入了一个访问标记的新API,以解决初始设计中的问题,提供

`**_pytest.nodes.Node.iter_markers()` 方法以一致的方式迭代标记并重新进行内部处理,这很好地解决了初始设计的问题。

升级代码

不推荐使用原有的 `Node.get_marker(name)` 函数,因为它返回一个内部 `MarkerInfo` 对象,该对象包含应用于该节点的所有标记的合并名称和所有参数。

通常,有两种方案可以处理标记:

标记互相覆盖。顺序很重要,但你只需要将你的标记视为单独的标记即可。例如。对于测试用例中的 `log_level('debug')` 会覆盖模块级别的 `log_level('info')`。

在这种情况下,可以使用 `Node.get_closest_marker(name)`:

```
# 替换这个:
marker = item.get_marker("log_level")
if marker:
    level = marker.args[0]

# 通过这个:
marker = item.get_closest_marker("log_level")
if marker:
    level = marker.args[0]
```

在特定条件下使用标记。例如, `skipif(condition)` 标记,意味着你只想测试所有非 `condition` 条件的用例,顺序不重要。你可以将这个标记视为一个满足该条件的集合使用。

在这种情况下,迭代每个标记并单独处理它们的 `*args` 和 `**kwargs` 参数。

```
# 替换这个:
skipif = item.get_marker("skipif")
```



```
if skipif:
    for condition in skipif.args:
        # eval condition
        ...

# 通过这个:
for skipif in item.iter_markers("skipif"):
    condition = skipif.args[0]
    # eval condition
```

如果你不确定或遇到任何难题,你可以考虑提出一个待解决问题。

注意: 在未来的Pytest主要版本中,我们将引入基于类的标记,在这些标记处,标记将不再局限于Mark的实例。

Monkeypatching,对模块和环境进行Mock

有时,测试需要调用依赖于全局设置的函数,或调用无法轻松测试的代码(如网络访问)。 `monkeypatch fixture`可帮助你安全地设置/删除属性,字典项或环境变量,或修改`sys.path`以进行导入。 请参阅[monkeypatch博客文章](#),了解一些介绍材料并讨论其动机。

简单示例如: 猴子补丁方法

如果你想阻止`os.expanduser`返回某个目录,你可以在测试用例调用其之前,使用`monkeypatch.setattr()`方法改造这个函数:

```
# test_module.py 文件内容
import os.path
def getssh(): # 伪应用代码
    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'
```

这里在我们的测试用例中,使用猴子补丁改造了`os.path.expanduser`,然后再进行调用。 测试执行完成后对`os.path.expanduser`修改将被撤消。

Monkeypatching 返回对象: 构建mock类

全局补丁示例如:阻止"requests"库的远程操作

如果要阻止"requests"库在所有测试中执行http请求,你可以执行以下操作:

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

每个测试用例执行时都会自动使用该fixture,它将删除测试用例内置属性中的`request.session.Session.request`,以便在测试中任何使用requests库创建http请求的用例都将失败。

注意: 不建议使用猴子补丁改造Python内置函数,如`open`,`compile`等,因为它可能会破坏pytest的内部逻辑。如果必须要使用,你可以通过参数: `--tb = native`, `- tables = plain`和`--capture = no`来试试,不一定不会有问题。

注意: 改造`stdlib`函数和pytest依赖的某些第三方库本身可能会破坏pytest,因此在这些情况下,建议使用`MonkeyPatch.context()`来改造这些模块:

```
import functools

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

查看#3290号bug详情

Monkeypatching 环境变量

Monkeypatching字典

参考API

查阅MonkeyPatch类相关文档。

使用tmp目录和文件

tmp_path Fixture方法

3.9版本新函数 你可以使用`tmp_path` 在临时目录根目录中创建一个独立的临时目录以供测试调用。

`tmp_path`是一个`pathlib/pathlib2.Path`对象。以下是测试使用方法的示例如:

```
# test_tmp_path.py 文件内容
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

```

运行这个,我们可以看到,除了`assert 0`这一行,其他断言都正常测试通过:

```

$ 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
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 =====

```

tmp_path_factory Fixture方法

2.8版本新函数`tmpdir_factory`是一个session范围的fixture,可用于从任何其他测试用例及fixture中创建任意临时目录。

例如,假设你的测试套件需要使用程序动态生成在本地磁盘上的一个大图片,你可以整个测试session中只生成一次以节省时间,而不是为每个用例都在自己的`tmpdir`中计算并生成一次:

```

# conftest.py文件内容
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

# contents of test_image.py
def test_histogram(image_file):
    img = load_image(image_file)
    # 计算和测试histogram
```

有关详细信息,请参阅tmpdir_factory API。

tmpdir Fixture方法

tmpdir_factory Fixture方法

默认临时目录根目录

默认情况下,临时目录创建为系统临时目录的子目录。基本名称将是`pytest-数字`,其中数字将随着每次测试运行而递增。此外,第3个以后的临时目录会被删除。

你可以如下所示,修改默认的临时目录设置:

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

在本地计算机上分发测试时,pytest会为子进程配置临时目录根目录,以便所有临时数据都落在单个每个测试运行的临时目录根目录。

捕获stdout及stderr输出

默认 stdout/stderr/stdin 捕获行为

在测试执行期间,程序中的标准输出/标准错误输出都会被捕获到。如果测试或`setup`方法执行失败时,会在报错追溯信息中查看到程序中的标准输出及标准错误输出。(可以通过`--show-capture`命令行选项配置是否捕获程序中的标准输出/标准错误输出)。

此外,stdin被设置为“null”对象,测试运行过程中无法从中读取数据,因为在运行自动化测试时很少需要等待交互式输入。

捕获默认是通过拦截对低优先级文件描述符的写入来完成的。这允许捕获简单`print`语句的输出以及测试启动的子进程的输出。

设置捕获方法或禁用捕获

pytest可以通过两种方式捕获输出:

- 文件描述符(FD)级别捕获(默认): 将捕获进入操作系统文件描述符1和2的所有写入。
- `sys`级别捕获: 仅捕获Python文件`sys.stdout`和`sys.stderr`。不执行对文件描述符的写入捕获。

你可以在命令行中指定不同的参数来使用不同的捕获机制：

```
pytest -s          # 禁止捕获所有输出
pytest --capture=sys # 使用in-mem文件代替sys.stdout/stderr with
pytest --capture=fd  # 同时将filedescriptors 1和2指向临时文件
```

调试中使用print语句

默认捕获stdout / stderr输出的一个主要好处是可以使用print语句进行调试：

```
# test_module.py文件内容

def setup_function(function):
    print("setting up %s" % function)

def test_func1():
    assert True

def test_func2():
    assert False
```

运行此模块将只捕获失败用例相关的print信息,而不显示成功用例的print信息：

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

在测试用例中使用的捕获的输出

`capsys`,`capsysbinary`,`capfd`和`capfdbinary` fixture允许访问在测试执行期间创建的stdout / stderr输出。下面是一个测试函数示例,它执行一些与输出相关的检查：

```
def test_myoutput(capsys): # or use "capfd" for fd-level
    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()`调用时首先对输出流建立快照 - 并继续捕获输出,然后在该测试用例执行完成后,恢复原始输出流。而通过使用`capsys`可以避免在执行每个测试用例时都进行一次设置/重置输出流,并且还可以与`pytest`每次测试用例执行时捕获的输出信息进行交互。

如果要在`filedescriptor`级别捕获,可以使用`capfd fixture`,它提供完全相同的接口,但也允许捕获直接写入操作系统级输出流(FD1和FD2)的库或子进程的输出流中。

3.3版本新函数 `readouterr` 的返回值更改为具有两个属性`out`和`err`的`namedtuple`。

3.3版本新函数 如果测试中的代码写入了非文本数据,则可以使用`capsysbinary fixture`来捕获它,而后者会从`readouterr`方法返回字节。 `capfsysbinary fixture`目前仅在Python 3中可用。

3.0版本新函数 要暂时禁用测试中的捕获,`capsys`和`capfd`都有一个`disabled()`方法,可以用作上下文管理器,禁用`with`块内的捕获:

```
def test_disabling_capturing(capsys):
    print("输出被捕获到了")
    with capsys.disabled():
        print("输出未捕获到,直接使用sys.stdout标准输出")
    print("这个输出也被捕获到了")
```

捕获警告信息

3.1版中的新函数。

从版本开始3.1,pytest现在会在测试执行期间自动捕获警告并在会话结束时显示它们:

```
# content of 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pytest现在产生这个输出:

```
$ 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
collected 1 item

test_show_warnings.py . [100%]

===== warnings summary =====
test_show_warnings.py::test_one
  $REGENDOC_TMPDIR/test_show_warnings.py:5: 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`可以传递该标志以控制将显示哪些警告,甚至将其转换为错误:

```
$ 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:10:
-----

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:5: UserWarning
1 failed in 0.12 seconds
```

可以`pytest.ini`使用`filterwarnings`ini选项在文件中设置相同的选项。例如,以下配置将忽略所有用户警告,但会将所有其他警告转换为错误。

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

当警告与列表中的多个选项匹配时,将执行最后一个匹配选项的操作。

这两个`-W`命令行选项和`filterwarnings`INI选项是基于Python的`[-W`选项,所以请参考这些部分Python文档的其他例子和高级的使用方法英寸

@pytest.mark.filterwarnings

版本3.2中的新函数。

你可以使用`@pytest.mark.filterwarnings`向特定测试项添加警告过滤器,以便更好地控制应在测试,类甚至模块级别捕获哪些警告:

```
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
```

使用标记应用的过滤器优先于在命令行上传递或由`filterwarnings`ini选项配置的过滤器。

你可以通过使用`filterwarnings`标记作为类装饰器或通过设置`pytestmark`变量将模块中的所有测试应用于类的所有测试:

```
# turns all warnings into errors for this module
pytestmark = pytest.mark.filterwarnings("error")
```

积分转到[Florian Schulze](#)获取[pytest-warnings](#)插件中的参考实现。

禁用警告摘要

虽然不推荐,但你可以使用`--disable-warnings`命令行选项从测试运行输出中完全禁止警告摘要。

完全禁用警告捕获

此插件默认启用,但可以在你的`pytest.ini`文件中完全禁用:

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

或者传入命令行。如果测试套件使用外部系统处理警告,这可能很有用。`-pno:warnings`

弃用警告和待命记录警告

版本3.8中的新函数。

在3.9版中更改。

默认情况下,pytest将显示`DeprecationWarning`和`PendingDeprecationWarning`从用户代码和第三方库警告,建议[PEP-0565]。这有助于用户保持代码现代化,并在有效删除已弃用的警告时避免破坏。

有时隐藏在你无法控制的代码(例如第三方库)中发生的某些特定弃用警告很有用,在这种情况下,你可以使用警告过滤器选项(ini或标记)来忽略这些警告。

例如:

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

这将忽略`DeprecationWarning`消息开头与正则表达式匹配的所有类型的警告。".*U.*modeisdeprecated"

注意

如果在解释器级别配置警告,使用[PYTHONWARNINGS环境变量或-W命令行选项,pytest将默认不配置任何过滤器。

此外,pytest不遵循PEP-0506重置所有警告过滤器的建议,因为它可能会破坏通过调用自行配置警告过滤器的测试套件`warnings.simplefilter(请参阅问题[# 2430以获取该示例)。

确保代码触发弃用警告

你还可以调用全局帮助程序来检查某个函数调用是否触发a`DeprecationWarning`或`PendingDeprecationWarning`:

```
import pytest

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

默认情况下,`DeprecationWarning`并`PendingDeprecationWarning`不会被使用时,捕捉`pytest.warns`或`recwarn`因为默认的Python警告过滤器隐藏起来。如果你希望在自己的代码中记录它们,请使用以下命令`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)
```

你还可以将其用作上下文管理器:

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

用警告函数断言警告

版本2.8中的新函数。

你可以检查代码是否引发了特定警告`pytest.warns`,其工作方式类似于[引发:

```
import warnings
import pytest

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

如果未提出相关警告,测试将失败。`match`断言异常与文本或正则表达式匹配的关键字参数:

```
>>> 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+/pre>):
...     warnings.warn("value must be 42",UserWarning)

>>> with warns(UserWarning,match=r'must be \d+/pre>):
...     warnings.warn("this is not here",UserWarning)
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

你还可以调用`pytest.warns`函数或代码字符串:

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

该函数还返回所有引发警告(作为`warnings.WarningMessage`对象)的列表,你可以查询其他信息:

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

# check that only one warning was raised
assert len(record) == 1
```

```
# check that the message matches
assert record[0].message.args[0] == "another warning"
```

或者,你可以使用`recwarnFixture`方法详细检查凸起的警告(见下文)。

注意

`DeprecationWarning`并且`PendingDeprecationWarning`区别对待;请参阅[\[确保代码触发弃用警告\]](#)。

录制警告

你可以使用Fixture方法`pytest.warns`或使用`recwarnFixture`方法记录凸起的警告。

要在`pytest.warns`不声明任何有关警告的情况下进行记录,请传递`None`为预期的警告类型:

```
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"
```

该`recwarnFixture`方法将记录整个函数的警告:

```
import warnings

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

双方`recwarn`并`pytest.warns`返回相同的接口,用于记录警告: 一个`WarningsRecorder`实例。要查看记录的警告,你可以迭代此实例,调用`len`它以获取已记录警告的数量,或将其编入索引以获取特定记录的警告。

完整的API :`WarningsRecorder`.

自定义失败消息

记录警告提供了在未发出警告或满足其他条件时生成自定义测试失败消息的机会。

```
def test():
    with pytest.warns(Warning) as record:
```

```
f()
if not record:
    pytest.fail("Expected a warning!")
```

如果在呼叫时没有发出警告`f`,那么将评估为。然后,你可以使用自定义错误消息进行调用。

```
notrecord`True`pytest.fail
```

内部pytest警告

版本3.8中的新函数。

pytest可能会在某些情况下生成自己的警告,例如使用不当或不推荐使用的函数。

例如,如果遇到匹配`python_classes`但也定义`__init__`构造函数的类,pytest将发出警告,因为这会阻止实例化类:

```
# content of 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
```

可以使用用于过滤其他类型警告的相同内置机制来过滤这些警告。

请阅读我们的[向后兼容性政策,了解我们如何继续弃用并最终删除函数。

pytest使用以下警告类型,它们是公共API的一部分:

`class PytestWarning` 基类: `UserWarning` pytest发出的所有警告的基类。

`class PytestDeprecationWarning` 基类: `pytest.PytestWarning` 将来版本中将删除的函数的警告类。

`class RemovedInPytest4Warning` 基类: `pytest.PytestDeprecationWarning` 计划在pytest 4.0中删除的函数的警告类。

`class PytestExperimentalApiWarning` 基类: `pytest.PytestWarning` 警告类别用于表示pytest中的实验。谨慎使用,因为API可能会在未来版本中更改甚至完全删除\

模块及测试文件中集成doctest测试

编码

使用doctest选项

默认情况下,Pytest按照python `doctest`模块标准`test*.txt`模式进行匹配。你也可以通过使用以下命令更改匹配模式:

```
pytest --doctest-glob='*.rst'
```

在命令行上。从版本开始2.9, `--doctest-glob`可以在命令行中多次使用。

3.1版中的新增函数: 你可以使用`doctest_encoding`ini选项指定将用于这些doctest文件的编码:

```
# content of pytest.ini
[pytest]
doctest_encoding = latin1
```

默认编码为UTF-8。

你还可以在所有python模块(包括常规python测试模块)中从docstrings触发doctests的运行:

```
pytest --doctest-modules
```

你可以将这些更改永久保存在项目中,方法是将它们放入`pytest.ini`文件中,如下所示:

```
# content of pytest.ini
[pytest]
addopts = --doctest-modules
```

如果你有一个这样的文本文件:

```
# content of example.rst

hello this is a doctest
>>> x = 3
>>> x
3
```

和另一个这样的:

```
# content of mymodule.py
def something():
    """ a doctest in a docstring
    >>> something()
    42
    """
    return 42
```

那么你可以在pytest没有命令行选项的情况下调用:

```
$ 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: pytest.ini
collected 1 item

mymodule.py . [100%]

===== 1 passed in 0.12 seconds =====
```

可以使用getfixture帮助器使用Fixture方法:

```
# content of example.rst
>>> tmp = getfixture('tmpdir')
>>> ...
>>>
```

此外,在执行文本doctest文件时,支持[使用类,模块或项目中的Fixture。

标准doctest模块提供了一些设置标志来配置doctest测试的严格性。在pytest中,你可以使用配置文件启用这些标志。要使pytest忽略尾随空格并忽略冗长的异常堆栈跟踪,你只需编写:

```
[pytest]
doctest_optionflags= NORMALIZE_WHITESPACE IGNORE_EXCEPTION_DETAIL
```

pytest还引入了新的选项,允许doctests在Python 2和Python 3中运行不变:

- **ALLOW_UNICODE**: 启用时,u前缀将从预期doctest输出中的unicode字符串中删除。
- **ALLOW_BYTES**: 启用时,b前缀将从预期doctest输出中的字节字符串中删除。

与任何其他选项标志一样,可以pytest.ini使用doctest_optionflagsini选项启用这些标志:

```
[pytest]
doctest_optionflags = ALLOW_UNICODE ALLOW_BYTES
```

或者,可以通过doc测试本身中的内联注释启用它:

```
# content of example.rst
>>> get_unicode_greeting() # doctest: +ALLOW_UNICODE
'Hello'
```

默认情况下,pytest仅报告给定doctest的第一次失败。如果你想在即使遇到故障时继续测试,请执行以下操作:

```
pytest --doctest-modules --doctest-continue-on-failure
```

3.0版中的新函数。

该`doctest_namespace` Fixture方法可用于注入到项目中,你的文档测试运行的命名空间。它旨在用于你自己的Fixture方法中,以提供将它们与上下文一起使用的测试。

`doctest_namespace`是一个标准dict对象,你可以将要放置在doctest命名空间中的对象放入其中:

```
# content of conftest.py
import numpy
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace['np'] = numpy
```

然后可以直接在你的doctests中使用它:

```
# content of numpy.py
def arange():
    """
    >>> a = np.arange(10)
    >>> len(a)
    10
    """
    pass
```

请注意,与正常情况一样`conftest.py`,在`conftest`所在的目录树中发现了`fixture`。意味着如果将doctest与源代码放在一起,则相关的`conftest.py`需要位于同一目录下。在同级目录树中不会发现Fixtures!

输出格式

3.0版中的新函数。

你可以通过使用选项标准文档测试模块格式的一个更改失败你的文档测试diff的输出格式(见

`doctest.REPORT_UDIFF`):

```
pytest --doctest-modules --doctest-report none
pytest --doctest-modules --doctest-report udiff
pytest --doctest-modules --doctest-report cdiff
pytest --doctest-modules --doctest-report ndiff
pytest --doctest-modules --doctest-report only_first_failure
```

pytest-specific 特性

skip及xfail: 处理不能成功的测试用例

你可以标记无法在某些平台上运行的测试用例或你希望失败的测试用例,以便pytest可以相应地处理它们并提供测试会话的摘要,同时保持测试套件为*通过*状态。

一个**Skip**意味着你希望如果某些条件得到满足你的测试才执行,否则pytest应该完全跳过运行该用例。常见示例是在非Windows平台上跳过仅限Windows的测试用例,或者跳过依赖于当前不可用的外部资源的测试用例(例如数据库)。

一个**xfail**意味着已知测试失败并标记原因。一个常见的例子是对尚未实现的函数的测试,或者尚未修复的错误。当测试通过时尽管预计会失败(标记为`pytest.mark.xfail`),但它在*并将在测试摘要及报告中显示为**xpass**。

pytest分别计算并列出skip和xfail测试。默认情况下不显示有关skip/ xfailed测试的详细信息,以避免输出混乱。你可以使用该`-r`选项查看与测试进度中显示的“短”字母对应的详细信息:

```
pytest -rxXs # show extra info on xfailed,xpassed,and skipped tests
```

`-r`可以通过运行找到有关该选项的更多详细信息。`pytest-h`

(请参阅[如何更改命令行选项默认值])

跳过测试用例

版本2.9中的新函数。

跳过测试用例的最简单方法是使用**skip**装饰器标记它,可以传递一个可选的**reason**:

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

或者,也可以通过调用`pytest.skip(reason)`函数在测试执行或设置期间强制跳过:


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

当在导入时间内无法评估跳过条件时,命令性方法很有用。

也可以在模块级别跳过整个模块: `pytest.skip(reason,allow_module_level=True)`

```
import sys
import pytest

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

参考: `pytest.mark.skip`

skipif

2.0版中的新函数。

如果你希望有条件地跳过某些内容,则可以使用`skipif`。下面是一个标记在Python3.6之前的解释器上运行时要跳过的测试函数的示例如:

```
import sys

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

如果条件`True`在收集期间评估为,则将跳过测试函数,使用时会在摘要中显示指定的原因-`rs`。

你可以`skipif`在模块之间共享标记。考虑这个测试模块:

```
# content of test_mymodule.py
import mymodule

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

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

你可以导入`mark`标记并在另一个测试模块中重复使用它:

```
# test_myothermodule.py
from test_mymodule import minversion

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

对于较大的测试套件,通常最好有一个文件来定义标记,然后在整个测试套件中一致地应用这些标记。

或者,你可以使用[条件字符串而不是布尔值,但它们不能在模块之间轻松共享,因此主要出于向后兼容性原因支持它们。

参考: `pytest.mark.skipif`

跳过类或模块的所有测试用例

你可以`skipif`在类上使用标记(与任何其他标记一样):

```
@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"
```

如果条件是`True`,则此标记将为该类的每个测试用例生成跳过结果。

如果要跳过模块的所有测试用例,可以`pytestmark`在全局级别使用该名称:

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

如果将多个`skipif`装饰器应用于测试用例,则如果任何跳过条件为真,则将跳过该装饰器。

跳过文件或目录

有时你可能需要跳过整个文件或目录,例如,如果测试依赖于Python版本特定的函数或包含你不希望运行pytest的代码。在这种情况下,你必须从集合中排除文件和目录。有关更多信息,请参阅[自定义测试集合]。

跳过缺少的导入依赖关系

你可以在模块级别或测试或测试`setup`方法中使用以下帮助程序:

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

如果`docutils`无法在此处导入,则会导致测试跳过结果。你还可以根据库的版本号跳过:

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

将从指定模块的`__version__`属性中读取版本。

摘要

以下是如何在不同情况下跳过模块中的测试用例的快速指南:

1. 无条件地跳过模块中的所有测试用例如:

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

2. 根据某些条件跳过模块中的所有测试用例如:

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

3. 如果缺少某些导入,则跳过模块中的所有测试用例如:

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

XFail: 将测试函数标记为预期失败

你可以使用`xfail`标记指示你希望测试失败:

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

将运行此测试,但在失败时不会报告回溯。相反,终端报告会将其列在“预期失败”(XFAIL)或“意外传递”(XPASS)部分中。

或者,你也可以XFAIL在测试或设置函数中强制标记测试:

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

这将无条件地制作`test_function`XFAIL`。请注意,`pytest.xfail`调用后不会执行其他代码,与标记不同。那是因为它是通过引发已知异常在内部实现的。

参考: `pytest.mark.xfail`

`strict`参数

版本2.9中的新函数。

双方`XFAIL`并`XPASS`除非不失败的测试套件,`strict`只有关键字的参数作为传递`True`:

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

这将导致`XPASS`(“意外通过”)此测试的结果导致测试套件失败。

你可以`strict`使用`xfail_strict`ini选项更改参数的默认值:

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

`reason`参数

与`skipif`一样,你也可以在特定平台上标记你对失败的期望:

```
@pytest.mark.skipif(3,6),reason="python3.6 api changes")
def test_function():
    ...
```

`raises`参数

如果你想更具体地说明测试失败的原因,可以在`raises`参数中指定单个异常或异常元组。

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

然后,如果测试失败并且没有提到的例外,那么测试将被报告为常规失败`raises`。

`run`参数

如果测试应标记为`xfail`并且如此报告但不应该执行,请使用以下`run`参数`False`:

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

这对于崩溃解释器的xfailing测试特别有用,应该稍后进行调查。

忽略xfail

通过在命令行上指定:

```
pytest --runxfail
```

你可以强制运行并报告xfail标记的测试,就像它根本没有标记一样。这也导致pytest.xfail没有效果。

示例

这是一个简单的测试文件,有几个使用方法:

```
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(raises=IndexError)
def test_hello7():
```

```
x = []
x[1] = 1
```

使用report-on-xfail选项运行它会提供以下输出:

```
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
collected 7 items

xfail_demo.py xxxxxxxx [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 =====
```

跳过/ xfail与参数多态

使用参数化时,可以将skip和xfail等标记应用于各个测试实例如:

```
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
```

Fixture方法及测试用例的参数化

Pytest在多个级别启用测试参数化:

- `pytest.fixture()` 允许一个[参数化Fixture方法。
- `@pytest.mark.parametrize` 允许在测试函数或类中定义多组参数和Fixture。
- `pytest_generate_tests` 允许用户定义自定义参数化方案或扩展。

`@pytest.mark.parametrize`: 参数化测试函数

2.2版中的新函数。

版本2.4中的更改: 一些改进。

内置的`[pytest.mark.parametrize]`装饰器支持测试函数的参数的参数化。以下是测试函数的典型示例,该函数实现检查某个输入是否导致预期输出:

```
# content of 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
```

这里,`@parametrize`装饰器定义了三个不同的`(test_input,expected)`元组,以便`test_eval`函数依次使用它们运行三次:

```
$ 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
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:6: AssertionError
===== 1 failed, 2 passed in 0.12 seconds =====
```

注意

默认情况下,pytest会转义unicode字符串中用于参数化的任何非ascii字符,因为它有几个缺点。但是,如果你想在参数化中使用unicode字符串并在终端中按原样(非转义)查看它们,请在以下位置使用此选项`pytest.ini`:

```
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = True
```

但请记住,这可能会导致不必要的副作用甚至是错误,具体取决于所使用的操作系统和当前安装的插件,因此使用它需要你自担风险。

如本例所示,只有一对输入/输出值无法通过简单的测试用例。和通常的测试函数参数一样,你可以在`traceback`中看到`input`和`output`值。

请注意,你还可以在类或模块上使用参数化标记(请参阅[使用属性标记测试函数]),这将使用参数集调用多个函数。

也可以在参数化中标记单个测试实例,例如使用内置`mark.xfail`:

```
# content of test_expectation.py
import pytest

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

我们运行这个:

```
$ 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
collected 3 items

test_expectation.py ..x [100%]

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


之前导致失败的一个参数集现在显示为“`xfailed`(预期失败)”测试。

如果提供的值`parametrize`导致空列表 - 例如,如果它们是由某个函数动态生成的 - 则`pytest`的行为由该`empty_parameter_set_mark`选项定义。

要获得多个参数化参数的所有组合,你可以堆叠`parametrize`装饰器:

```
import pytest

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

这将运行与设定参数的测试`x=0/y=2,x=1/y=2,x=0/y=3`,并`x=1/y=3`在装饰的秩序排气参数。

基本的`pytest_generate_tests`例子

有时你可能希望实现自己的参数化方案或实现一些动力来确定`Fixture`的参数或范围。为此,你可以使用`pytest_generate_tests`在收集测试函数时调用的钩子。通过传入的`metafunc`对象,你可以检查请求的测试上下文,最重要的是,你可以调用`metafunc.parametrize()`以引起参数化。

例如,假设我们想要运行一个测试,我们想通过一个新的`pytest`命令行选项设置字符串输入。让我们首先编写一个接受`stringinput` fixture函数参数的简单测试:

```
# content of test_strings.py

def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

现在我们添加一个`conftest.py`包含命令行选项和测试函数参数化的文件:

```
# content of 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"))
```

如果我们现在传递两个stringinput值,我们的测试将运行两次:

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

让我们运行一个stringinput导致测试失败:

```
$ pytest -q --stringinput="!" test_strings.py
F
===== 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:4: AssertionError
1 failed in 0.12 seconds
```

正如所料,我们的测试用例失败

如果你没有指定stringinput,它将被跳过,因为`metafunc.parametrize()`将使用空参数列表调用:

```
$ pytest -q -rs test_strings.py
s
===== short test summary info =====
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function
test_valid_string at $REGENDOC_TMPDIR/test_strings.py:2
1 skipped in 0.12 seconds
```

请注意,`metafunc.parametrize`使用不同的参数集多次调用时,这些集合中的所有参数名称都不能重复,否则将引发错误。

更多示例

有关更多示例,你可能需要查看[更多参数化示例](#)。

缓存:使用跨执行状态

版本2.8中的新函数。

使用方法

该插件提供了两个命令行选项,用于重新运行上次pytest调用的失败:

- `--lf,--last-failed`- 只重新运行故障。
- `--ff,--failed-first`- 先运行故障然后再运行其余的测试。

对于清理(通常不需要),`--cache-clear`选项允许在测试运行之前删除所有跨会话缓存内容。

其他插件可以访问`config.cache`对象以在调用之间设置/获取json可编码值pytest。

注意

此插件默认启用,但如果需要可以禁用: 请参阅[按名称取消激活/取消注册插件(此插件的内部名称为`cacheprovider`)]。

首先只重新运行故障或故障

首先,让我们创建50个测试调用,其中只有2个失败:

```
# content of test_50.py
import pytest

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

如果你是第一次运行它,你会看到两个失败:

```
$ 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

```

如果你然后运行它`--lf`:

```

$ 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
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 =====

```

你只运行了上次运行中的两个失败测试,而尚未运行48个测试(“取消选择”)。

现在,如果使用该`--ff`选项运行,将运行所有测试,但首先执行先前的失败(从一系列FF和点中可以看出):

```

$ 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
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 =====

```

新的`--nf`,`--new-first`选项: 首先运行新的测试,然后是其余的测试,在这两种情况下,测试也按文件修改时间排序,最新的文件首先出现。

上次运行中没有测试失败时的行为

如果在上次运行中没有测试失败,或者没有`lastfailed`找到缓存数据,`pytest`则可以使用该`--last-failed-no-failures`选项配置运行所有测试或不运行测试,该选项采用以下值之一:

```

pytest --last-failed --last-failed-no-failures all    # run all tests (default
behavior)
pytest --last-failed --last-failed-no-failures none  # run no tests and exit

```

新的`config.cache`对象

插件或`conftest.py`支持代码可以使用`pytestconfig`对象获取缓存值。这是一个实现[pytest fixture]的基本示例插件[: 显式,模块化,可伸缩,它在pytest调用中重用以前创建的状态:

```
# content of 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
```

如果你是第一次运行此命令,则可以看到`print`语句:

```
$ pytest -q
F [100%]
===== FAILURES =====
_____ test_function _____

mydata = 42

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

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

如果再次运行它,将从缓存中检索该值,并且不会打印任何内容:

```
$ pytest -q
F [100%]
===== FAILURES =====
_____ test_function _____

mydata = 42
```

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

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

有关更多详细信息,请参阅[config.cache]。

检查缓存内容

你始终可以使用`--cache-show`命令行选项查看缓存的内容:

```
$ 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
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 =====
```

清除缓存内容

你可以通过添加如下`--cache-clear`选项来指示pytest清除所有缓存文件和值:

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

对于Continuous Integration服务器的调用,建议使用此选项,其中隔离和正确性比速度更重要。

逐步修复失败用例

作为替代方案,尤其是对于你希望测试套件的大部分都会失败的情况,允许你一次修复一个。测试套件将运行直到第一次失败然后停止。在下次调用时,测试将从上次失败测试继续,然后运行直到下一次失败测试。你可以使用该选项忽略一个失败的测试,并在第二个失败的测试中停止测试执行。如果你遇到失败的测试而只是想稍后忽略它,这将非常有用。--lf-x`--sw`--stepwise`--stepwise-skip

unittest.TestCase支持

Pytest支持unittest开箱即用的基于Python的测试。它旨在利用现有unittest的测试套件将pytest用作测试运行器,并允许逐步调整测试套件以充分利用pytest的函数。

要使用运行现有unittest样式的测试套件pytest,请键入:

```
pytest tests
```

pytest将自动收集unittest.TestCase子类及其test方法test_*.py或*_test.py文件。

几乎所有unittest函数都受支持:

- @unittest.skip风格装饰;
- setUp/tearDown;
- setUpClass/tearDownClass;
- setUpModule/tearDownModule;

到目前为止,pytest不支持以下函数:

- load_tests协议;
- 分测验;

开箱即用的好处

通过使用pytest运行测试套件,你可以使用多种函数,在大多数情况下无需修改现有代码:

- 获得[更多信息性的追溯;
- stdout和stderr捕获;
- ;
- 在第一次(或N次)故障后停止;
- -pdb);
- 使用[pytest-xdist插件将测试分发给多个CPU;
- 使用[普通的assert-statements对此非常有帮助);

unittest.TestCase子类中的pytest特性

以下pytest函数适用于unittest.TestCase子类:

- 标记: skip,[skipif,[xfail;
- 自动使用Fixture方法;

下面pytest函数不工作,也许永远也因不同的设计理念:

- Fixture方法);

- 参数化;
- 定制挂钩;

第三方插件可能运行也可能不运行,具体取决于插件和测试套件。

unittest.TestCase 使用标记将pytestFixture方法混合到子类中

运行`unittestpytest`允许你使用其[Fixture方法机制进行`unittest.TestCase`样式测试。假设你至少浏览了`pytest fixture`函数,让我们跳转到一个集成`pytestdb_class`fixture,设置类缓存数据库对象,然后从`unittest`样式测试中引用它的示例如:

```
# content of conftest.py

# we define a fixture function below and it will be "used" by
# referencing its name from tests

import pytest

@pytest.fixture(scope="class")
def db_class(request):
    class DummyDB(object):
        pass
    # set a class attribute on the invoking test context
    request.cls.db = DummyDB()
```

这定义了一个fixture函数`db_class`- 如果使用的话 - 为每个测试类调用一次,并将`class-level db`属性设置为一个`DummyDB`实例。fixture函数通过接收一个特殊`request`对象来实现这一点,该对象允许访问[请求测试上下文,例如`cls`属性,表示使用该fixture的类。该架构将Fixture方法写入与实际测试代码分离,并允许通过最小参考(Fixture方法名称)重新使用Fixture方法。那么让`unittest.TestCase`我们使用fixture定义编写一个实际的类:

```
# content of 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 # fail for demo purposes

    def test_method2(self):
        assert 0, self.db # fail for demo purposes
```

在`@pytest.mark.usefixtures("db_class")`类的装饰可确保`pytest`固定函数`db_class`被调用每一次班。由于故意失败的断言语句,我们可以看看`self.db`回溯中的值:

```

$ 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
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 =====

```

这个默认的pytest回溯显示两个测试用例共享同一个`self.db`实例,这是我们在编写上面的类范围的`fixture`函数时的意图。

使用`autouseFixture`方法和访问其他`Fixture`方法

虽然通常更好地明确声明对给定测试需要使用的`Fixture`方法,但有时你可能想要在给定的上下文中自动使用`Fixture`方法。毕竟,传统的`unittest-setup`风格要求使用这种隐含的`Fixture`方法编写,而且很有可能,你习惯它或者喜欢它。

你可以使用标记`Fixture`方法函数`@pytest.fixture(autouse=True)`并在要使用它的上下文中定义`Fixture`方法函数。让我们看一个`initdir``Fixture`方法,它使一个`TestCase`类的所有测试用例都在一个预先初始化的临时目录中执行`samplefile.ini`。我们的`initdir``fixture`本身使用`pytest builtin[tmpdirfixture]`来委托创建一个`per-test`临时目录:

```

# content of test_unittest_cleandir.py
import pytest
import unittest

```

```
class MyTest(unittest.TestCase):

    @pytest.fixture(autouse=True)
    def initdir(self, tmpdir):
        tmpdir.chdir() # change to pytest-provided temporary directory
        tmpdir.join("samplefile.ini").write("# testdata")

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

由于该`autouse`标志,`initdir`fixture函数将用于定义它的类的所有方法。这是
`@pytest.mark.usefixtures("initdir")`在类中使用标记的快捷方式,如上例所示。

运行此测试模块.....:

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

...给我们一个通过测试,因为`initdir`Fixture方法函数在之前执行`test_method`。

注意

`unittest.TestCase`方法不能直接接收`fixture`参数作为实现可能会导致运行通用`unittest.TestCase`测试套件的能力。

以上`usefixtures`和`autouse`示例应该有助于将`pytest`Fixture方法混合到`unittest`套件中。

你也可以逐步从子类化转移`unittest.TestCase`到普通断言,然后开始逐步从完整的`pytest`函数集中受益。

注意

从`unittest.TestCase`子类运行测试`--pdb`将禁用针对发生异常的情况的`tearDown`和`cleanup`方法。这允许对所有在其`tearDown`机器中具有重要逻辑的应用程序进行适当的事后调试。但是,支持此函数会产生以下副作用:如果人们覆盖`unittest.TestCase`__call__``或者`run`需要以`debug`相同的方式覆盖(对于标准`unittest`也是如此)。

注意

由于两个框架之间的架构差异,在`unittest`测试`call`阶段而不是在`pytest`标准`setup`和`teardown`阶段中执行基于测试的设置和拆卸。在某些情况下,这一点非常重要,特别是在推理错误时。例如,如果`unittest`基于`a`的套件在设置期间出现错误,`pytest`则在其`setup`阶段期间将报告没有错误,并且将在此期间引发错误`call`。

运行Nose用例

`Pytest`基本支持运行`Nose`框架格式的测试用例。

使用方法

后安装pytest类型:

```
python setup.py develop # make sure tests can import our package
pytest # instead of 'nosetests'
```

你应该能够运行你的nose样式测试并利用pytest的函数。

支持的nose风格

- 在模块/类/方法级别进行设置和拆卸
- SkipTest异常和标记
- 设置/拆卸装饰器
- `yield`基于测试及其设置(从pytest 3.0开始被认为已弃用)
- `__test__` 模块/类/函数的属性
- nose工具的一般使用方法

不支持的习语/已知问题

- unittest-style仅在类上被识别,而在普通类上不被识别。在普通类上也支持这些方法,但pytest故意不支持。由于nose和pytest都已经支持它,因此像nose一样复制unittest-API似乎没什么用。但是,如果你认为pytest应该支持普通类的单元测试拼写,请发帖[到这个[问题](#)。
`setUp,tearDown,setUpClass,tearDownClass`unittest.TestCase`nose`setup_class,teardown_class,setup_method,teardown_method[`
- 通过扩展sys.path / import语义,nose导入具有相同导入路径(例如tests.test_mod)但不同文件系统路径(例如tests/test_mode.py和other/tests/test_mode.py)的测试模块。pytest不这样做,但在[# 268中有讨论增加一些支持。请注意,[nose2选择避免此sys.path / import hackery。

如果将conftest.py文件放在项目的根目录中(由pytest确定),pytest将对该目录下面的代码运行测试“nose style”,方法是将其添加到你sys.path的安装代码而不是运行。

如果你运行设置项目而不是或任何包管理器等效项,你可能会发现自己想要这样做。建议在此模式下使用像tox这样的虚拟环境进行开发。`pythonsetup.pyinstall`pythonsetup.pydevelop`

- 没有正确收集和执行nose式doctests,doctest fixtures也不起作用。
- 没有nose配置被识别。
- `yield`基于方法的方法不能setup正确支持,因为该setup方法总是在同一个类实例中调用。目前没有计划修复此问题,因为yield-test在pytest 3.0中已弃用,pytest.mark.parametrize建议使用。

经典xUnit风格的setup/teardown

本节介绍了如何在每个模块/类/函数的基础上实现Fixture(setup和teardown测试状态)的经典而流行的方法。

注意

虽然这些`setup/teardown`方法对于来自`unittest`或`nose`的人来说简单且熟悉,但`background`你也可以考虑使用`pytest`更强大的[Fixture]机制利用依赖注入的概念,允许更模块化和更可扩展的方法来管理测试状态,特别是对于大型项目和函数测试。你可以在同一文件中混合两种Fixture机制,但`unittest.TestCase`子类的测试用例不能接收Fixture参数。

模块级别setup/teardown

如果在单个模块中有多个测试函数和测试类,则可以选择实现以下`fixture`方法,这些方法通常会针对所有函数调用一次:

```
def setup_module(module):
    """ setup any state specific to the execution of the given module."""

def teardown_module(module):
    """ teardown any state that was previously setup with a setup_module
    method.
    """
```

从`pytest-3.0`开始,`module`参数是可选的。

类级别setup/teardown

类似地,在调用类的所有测试用例之前和之后,在类级别调用以下方法:

```
@classmethod
def setup_class(cls):
    """ setup any state specific to the execution of the given class (which
    usually contains tests).
    """

@classmethod
def teardown_class(cls):
    """ teardown any state that was previously setup with a call to
    setup_class.
    """
```

方法和函数级别setup/teardown

同样,围绕每个方法调用调用以下方法:

```
def setup_method(self, method):
    """ setup any state tied to the execution of the given method in a
    class. setup_method is invoked for every test method of a class.
    """

def teardown_method(self, method):
    """ teardown any state that was previously setup with a setup_method
```

```
call.  
"""
```

从pytest-3.0开始,method参数是可选的。

如果你希望直接在模块级别定义测试函数,还可以使用以下函数来实现fixture:

```
def setup_function(function):  
    """ setup any state tied to the execution of the given function.  
    Invoked for every test function in the module.  
    """  
  
def teardown_function(function):  
    """ teardown any state that was previously setup with a setup_function  
    call.  
    """
```

从pytest-3.0开始,function参数是可选的。

备注:

- 每个测试过程可以多次调用setup / teardown对。
- 如果存在相应的setup函数并且跳过了失败/,则不会调用teardown函数。
- 在pytest-4.2之前,xunit样式的函数不遵守fixture的范围规则,因此例如setup_method可以在会话范围的autouse fixture之前调用a。

现在,xunit风格的函数与Fixture机制集成在一起,并遵守调用中涉及的Fixture方法的适当范围规则。

安装和使用插件

本节讨论如何安装和使用第三方插件。有关编写自己的插件的信息,请参阅[编写插件]。

安装第三方插件可以通过以下方式轻松完成pip:

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

如果安装了插件,则pytest自动查找并集成它,无需激活它。

这是一些流行插件的小注释列表:

- pytest-django应用程序编写测试。
- pytest-twisted应用程序编写测试,启动反应堆并处理测试函数的延迟。
- pytest-cov: 覆盖率报告,与分布式测试兼容
- pytest-xdist: 将测试分发到CPU和远程主机,以盒装模式运行,允许分段故障,在looononfailing模式下运行,自动重新运行文件更改的失败测试。

- `pytest-instafail`: 在测试运行期间报告失败。
- `pytest-bdd`使用行为驱动测试编写测试。
- `pytest-timeout`: 根据函数标记或全局定义进行超时测试。
- `pytest-pep8`: `--pep8`启用PEP8合规性检查的选项。
- `pytest-flakes`: 用`pyflakes`检查源代码。
- `oejskit`: 在实时浏览器中运行javascript `unittests`的插件。

要查看具有针对不同`pytest`和Python版本的最新测试状态的所有插件的完整列表,请访问[plugincompat](#)。

你还可以通过[\[pytest-pypi.org\]](#)搜索发现更多插件。

在测试模块或`conftest`文件中要求/加载插件

你可以在测试模块或`conftest`文件中要求插件,如下所示:

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

加载测试模块或`conftest`插件时,也会加载指定的插件。

注意

`pytest_plugins`不建议使用非根`conftest.py`文件中使用变量的插件。请参阅“编写插件”部分中的[完整说明]。

注意

该名称`pytest_plugins`是保留的,不应用作自定义插件模块的名称。

找出哪些插件是可用的

如果要查找环境中哪些插件处于可用状态,可以键入:

```
pytest --trace-config
```

并将获得一个扩展的测试标题,显示激活的插件及其名称。它还会在加载时打印本地插件aka[`conftest.py`文件]。

按名称取消/取消注册插件

你可以阻止插件加载或取消注册:

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

这意味着任何后续尝试激活/加载命名插件都不起作用。

如果要无条件禁用项目插件,可以将此选项添加到`pytest.ini`文件中:

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

或者,仅在某些环境中禁用它(例如在CI服务器中),可以将`PYTEST_ADDOPTS`环境变量设置为`-pno:name`

请参阅[\[查找有关如何获取插件名称的活动插件\]](#)。

插件编写

很容易为你自己的项目实现[\[本地conftest插件\]](#)或可以在许多项目中使用的可[\[安装的插件\]](#),包括第三方项目。如果你只想使用但不能编写插件,请参阅[\[安装和使用插件\]](#)。

插件包含一个或多个钩子(hooks)方法函数。[\[编写钩子\(hooks\)方法\]](#)解释了如何自己编写钩子(hooks)方法函数的基础知识和细节。`pytest`通过调用以下插件的[\[指定挂钩来实现配置,收集,运行和报告的所有方面\]](#):

- 内置插件: 从[pytest的内部_pytest](#)目录加载。
- 外部插件
- `conftest.py` plugins: 在测试目录中自动发现的模块

原则上,每个钩子(hooks)方法调用都是一个`1:N`Python函数调用,其中`N`是给定规范的已注册实现函数的数量。所有规范和实现都遵循[pytest_](#)前缀命名约定,使其易于区分和查找。

Pytest启动时的插件发现顺序

`pytest`通过以下方式在工具启动时加载插件模块:

- 通过加载所有内置插件
- 通过加载通过[\[setuptools入口点注册\]](#)的所有插件。
- 通过预扫描选项的命令行并在实际命令行解析之前加载指定的插件。`-pname`
- 通过[conftest.py](#)命令行调用推断加载所有文件:
 - 如果未指定测试路径,则使用当前`dir`作为测试路径
 - 如果存在,则加载[conftest.py](#)并[test*/conftest.py](#)相对于第一个测试路径的目录部分。

请注意,`pytestconftest.py`在工具启动时没有在更深的嵌套子目录中找到文件。将[conftest.py](#)文件保存在顶级测试或项目根目录中通常是个好主意。

- 通过递归加载文件中[pytest_plugins](#)变量指定的所有插件[conftest.py](#)

conftest.py: 本地每目录插件

本地[conftest.py](#)插件包含特定于目录的钩子(hooks)方法实现。`Hook Session`和测试运行活动将调用[conftest.py](#)靠近文件系统根目录的文件中定义的所有挂钩。实现[pytest_runtest_setup](#)钩子(hooks)方法的示例,以便在[a](#)子目录中调用而不是为其他目录调用:


```
a/conftest.py:
def pytest_runtest_setup(item):
    # called for running each test in 'a' directory
    print("setting up",item)

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

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

以下是运行它的方法:

```
pytest test_flat.py --capture=no # will not show "setting up"
pytest a/test_sub.py --capture=no # will show "setting up"
```

注意

如果你的`conftest.py`文件不在python包目录中(即包含一个`__init__.py`),那么“import conftest”可能不明确,因为`conftest.py`你`PYTHONPATH`或者也可能有其他文件`sys.path`。因此,项目要么放在`conftest.py`包范围内,要么永远不从`conftest.py`文件中导入任何内容,这是一种很好的做法。

另请参见: `pytest import mechanisms`和`sys.path / PYTHONPATH`。

编写自己的插件

如果你想编写插件,可以从中复制许多现实示例如:

- 自定义集合示例插件: 在Yaml文件中指定测试的基本示例
- 内置插件,提供pytest自己的函数
- 许多外部插件提供额外的函数

所有这些插件都实现了[钩子(hooks)方法以扩展和添加函数。

注意

请务必查看优秀[的`cookiecutter-pytest-plugin`。

该模板提供了一个很好的起点,包括一个工作插件,使用tox运行的测试,一个全面的README文件以及一个预先配置的入口点。

另外考虑[将你的插件贡献给`pytest-dev`一旦它拥有一些非自己的快乐用户。

使你的插件可以被他人安装

如果你想让你的插件在外部可用,你可以为你的发行版定义一个所谓的入口点,以便`pytest`找到你的插件模块。入口点是[`setuptools`。pytest查找`pytest11`入口点以发现其插件,因此你可以通过在`setuptools-involution`中定

义插件来使插件可用:

```
# sample ./setup.py file
from setuptools import setup

setup(
    name="myproject",
    packages=["myproject"],
    # the following makes a plugin available to pytest
    entry_points={"pytest11": ["name_of_plugin = myproject.pluginmodule"]},
    # custom PyPI classifier for pytest plugins
    classifiers=["Framework :: Pytest"],
)
```

如果以这种方式安装包,pytest将myproject.pluginmodule作为可以定义[挂钩的插件加载。

注意

确保包含在[PyPI分类器

断言重写

其中一个主要特性pytest是使用普通的断言语句以及断言失败时表达式的详细内省。这是由“断言重写”提供的,它在编译为字节码之前修改了解析的AST。这是通过一个完成的PEP 302导入挂钩,在pytest启动时及早安装,并在导入模块时执行此重写。但是,由于我们不想测试不同的字节码,因此你将在生产中运行此挂钩仅重写测试模块本身以及作为插件一部分的任何模块。任何其他导入的模块都不会被重写,并且会发生正常的断言行为。

如果你在其他模块中有断言助手,你需要启用断言重写,你需要pytest在导入之前明确要求重写这个模块。

注册一个或多个要在导入时重写的模块名称。

此函数将确保此模块或程序包内的所有模块将重写其assert语句。因此,你应确保在实际导入模块之前调用此方法,如果你是使用包的插件,则通常在__init__.py中调用。

抛出: **TypeError**- 如果给定的模块名称不是字符串。

当你编写使用包创建的pytest插件时,这一点尤为重要。导入挂钩仅将入口点confest.py中列出的文件和任何模块pytest11视为插件。作为示例,请考虑以下包:

```
pytest_foo/__init__.py
pytest_foo/plugin.py
pytest_foo/helper.py
```

使用以下典型setup.py提取物:

```
setup(...,entry_points={"pytest11": ["foo = pytest_foo.plugin"]},...)
```

在这种情况下,只会`pytest_foo/plugin.py`被重写。如果辅助模块还包含需要重写的断言语句,则需要导入之前将其标记为这样。通过将其标记为在`__init__.py`模块内部进行重写,这是最简单的,当导入包中的模块时,将始终首先导入该模块。这种方式`plugin.py`仍然可以`helper.py`正常导入。然后,内容`pytest_foo/__init__.py`将需要如下所示:

```
import pytest

pytest.register_assert_rewrite("pytest_foo.helper")
```

在测试模块或`conftest`文件中要求/加载插件

你可以在测试模块或这样的`conftest.py`文件中要求插件:

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

加载测试模块或`conftest`插件时,也会加载指定的插件。任何模块都可以作为插件祝福,包括内部应用程序模块:

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

`pytest_plugins`变量是递归处理的,所以请注意,在上面的示例中,如果`myapp.testsupport.myplugin`也声明`pytest_plugins`,变量的内容也将作为插件加载,依此类推。

注意

`pytest_plugins`不建议使用非根`conftest.py`文件中使用变量的插件。

这很重要,因为`conftest.py`文件实现了每个目录的钩子(hooks)方法实现,但是一旦导入了插件,它就会影响整个目录树。为了避免混淆,不推荐`pytest_plugins`在任何`conftest.py`不在测试根目录中的文件中进行定义,并将发出警告。

这种机制使得在应用程序甚至外部应用程序中共享Fixture方法变得容易,而无需使用`setuptools`入口点技术创建外部插件。

导入的插件`pytest_plugins`也会自动标记为断言重写(请参阅参考资料

`pytest.register_assert_rewrite()`在导入模块之前自行调用,也可以安排代码以延迟导入,直到注册插件为止。

按名称访问另一个插件

如果一个插件想要与另一个插件的代码协作,它可以通过插件管理器获得一个引用,如下所示:

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

如果要查看现有插件的名称,请使用该`--trace-config`选项。

注册为通用标记

测试插件

pytest附带一个名为的插件`pytester`,可帮助你为插件代码编写测试。默认情况下,该插件处于禁用状态,因此你必须先启用它,然后才能使用它。

你可以通过`conftest.py`将以下行添加到测试目录中的文件来执行此操作:

```
# content of conftest.py

pytest_plugins = ["pytester"]
```

或者,你可以使用命令行选项调用pytest。 `-ppytester`

这将允许你使用`testdir` fixture来测试你的插件代码。

让我们用一个例子演示你可以用插件做什么。想象一下,我们开发了一个插件,它提供了一个`hello`产生函数的`fixture`,我们可以用一个可选参数调用这个函数。如果我们不提供值或者我们提供字符串值,它将返回字符串值。`HelloWorld!`Hello{value}!`

```
# -*- 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 {name}!".format(name=name)

    return _hello
```

现在,`testdir` fixture提供了一个方便的API来创建临时`conftest.py`文件和测试文件。它还允许我们运行测试并返回一个结果对象,通过它我们可以断言测试的结果。

```
def test_hello(testdir):
    """Make sure that our plugin works."""

    # create a temporary conftest.py file
    testdir.makeconftest(
        """
import pytest

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

    # create a temporary pytest test file
    testdir.makepyfile(
        """
def test_hello_default(hello):
    assert hello() == "Hello World!"

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

    # run all tests with pytest
    result = testdir.runpytest()

    # check that all 4 tests passed
    result.assert_outcomes(passed=4)
```

另外,可以在运行pytest之前复制示例文件夹的示例

```
# content of pytest.ini
[pytest]
pytester_example_dir = .
```

```
# content of test_example.py

def test_plugin(testdir):
    testdir.copy_example("test_example.py")
    testdir.runpytest("-k", "test_example")
```

```
def test_example():
    pass
```

```
$ 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: pytest.ini
collected 2 items

test_example.py .. [100%]

===== warnings summary =====
test_example.py::test_plugin
  $REGENDOC_TMPDIR/test_example.py:4: PytestExperimentalApiWarning:
    testdir.copy_example is an experimental api that may change over time
    testdir.copy_example("test_example.py")

-- Docs: https://docs.pytest.org/en/latest/warnings.html
===== 2 passed,1 warnings in 0.12 seconds =====
```

有关`runpytest()`返回的结果对象及其提供的方法的更多信息,请查看[RunResult](#)文档。

编写钩子(hooks)方法函数

钩子(hooks)方法函数验证和执行

pytest为任何给定的钩子(hooks)方法规范调用已注册插件的钩子(hooks)方法函数。让我们看一下钩子(hooks)方法的典型钩子(hooks)方法函数,pytest在收集完所有测试项目后调用。

`pytest_collection_modifyitems(session,config,items)`

当我们`pytest_collection_modifyitems`在插件中实现一个函数时,pytest将在注册期间验证你是否使用了与规范匹配的参数名称,如果没有则拯救。

让我们看一下可能的实现:

```
def pytest_collection_modifyitems(config,items):
    # called after collection is completed
    # you can modify the ``items`` list
    ...
```

这里,pytest将传入`config`(pytest配置对象)和`items`(收集的测试项列表),但不会传入`session`参数,因为我们没有在函数签名中列出它。这种动态的“修剪”参数允许pytest“未来兼容”:我们可以引入新的钩子(hooks)方法命名参数而不破坏现有钩子(hooks)方法实现的签名。这是pytest插件的一般长期兼容性的原因之一。

请注意,除了`pytest_runtest_*`不允许引发异常之外的钩子(hooks)方法函数。这样做会打破pytest运行。

firstresult: 首先停止非无结果

大多数对`pytest`钩子(hooks)方法的调用都会产生一个**结果列表,**其中包含被调用钩子(hooks)方法函数的所有非None结果。

一些钩子(hooks)方法规范使用该`firstresult=True`选项,以便钩子(hooks)方法调用仅执行,直到N个注册函数中的第一个返回非None结果,然后将其作为整个钩子(hooks)方法调用的结果。在这种情况下,不会调用其余的钩子(hooks)方法函数。

hookwrapper: 在其他钩子(hooks)方法周围执行

版本2.7中的新函数。

`pytest`插件可以实现钩子(hooks)方法包装器,它包装其他钩子(hooks)方法实现的执行。钩子(hooks)方法包装器是一个生成器函数,它只产生一次。当`pytest`调用钩子(hooks)方法时,它首先执行钩子(hooks)方法包装器并传递与常规钩子(hooks)方法相同的参数。

在钩子(hooks)方法包装器的屈服点,`pytest`将执行下一个钩子(hooks)方法实现,并以`Result`封装结果或异常信息的实例的形式将其结果返回到屈服点。因此,屈服点本身通常不会引发异常(除非存在错误)。

以下是钩子(hooks)方法包装器的示例定义:

```
import pytest

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

    outcome = yield
    # outcome.excinfo may be None or a (cls,val,tb) tuple

    res = outcome.get_result() # will raise if outcome was exception

    post_process_result(res)

    outcome.force_result(new_res) # to override the return value to the plugin
system
```

请注意,钩子(hooks)方法包装器本身不返回结果,它们只是围绕实际的钩子(hooks)方法实现执行跟踪或其他副作用。如果底层钩子(hooks)方法的结果是一个可变对象,它们可能会修改该结果,但最好避免它。

有关更多信息,请参阅[插件文档](#)。

钩子(hooks)方法函数排序/调用示例

对于任何给定的钩子(hooks)方法规范,可能存在多个实现,因此我们通常将`hook`执行视为`1:N`函数调用,其中N是已注册函数的数量。有一些方法可以影响钩子(hooks)方法实现是在其他人之前还是之后,即在N-sized函数列表中的位置:

```

# Plugin 1
@pytest.hookimpl(tryfirst=True)
def pytest_collection_modifyitems(items):
    # will execute as early as possible
    ...

# Plugin 2
@pytest.hookimpl(trylast=True)
def pytest_collection_modifyitems(items):
    # will execute as late as possible
    ...

# Plugin 3
@pytest.hookimpl(hookwrapper=True)
def pytest_collection_modifyitems(items):
    # will execute even before the tryfirst one above!
    outcome = yield
    # will execute after all non-hookwrappers executed

```

这是执行的顺序:

1. Plugin3的`pytest_collection_modifyitems`被调用直到屈服点,因为它是一个钩子(hooks)方法包装器。
2. 调用Plugin1的`pytest_collection_modifyitems`是因为它标有`tryfirst=True`。
3. 调用Plugin2的`pytest_collection_modifyitems`因为它被标记`trylast=True`(但即使没有这个标记,它也会在Plugin1之后出现)。
4. 插件3的`pytest_collection_modifyitems`然后在屈服点之后执行代码。`yield`接收一个`Result`实例,该实例封装了调用非包装器的结果。包装不得修改结果。

这是可能的使用`tryfirst`,并`trylast`结合还`hookwrapper=True`处于这种情况下,它会影响彼此之间`hookwrappers`的排序。

声明新钩子(hooks)方法

插件和`conftest.py`文件可以声明新钩子(hooks)方法,然后可以由其他插件实现,以便改变行为或与新插件交互:

在插件注册时调用,允许通过调用添加新的挂钩。

`pluginmanager.add_hookspecs(module_or_class, prefix)` 参数:

pluginmanager(`pytest.config.PytestPluginManager`) - pytest插件管理器

注意: 这个钩子(hooks)方法与之不相容`hookwrapper=True`。

钩子(hooks)方法通常被声明为do-nothing函数,它们只包含描述何时调用钩子(hooks)方法以及期望返回值的文档。

有关示例,请参阅[xdist]中。

可选择使用第三方插件的钩子(hooks)方法

由于标准的[验证机制,方法可能有点棘手: 如果你依赖未安装的插件,验证将失败并且错误消息对你的用户没有多大意义。

一种方法是将钩子(hooks)方法实现推迟到新的插件,而不是直接在插件模块中声明钩子(hooks)方法函数,例如:

```
# contents of myplugin.py

class DeferPlugin(object):
    """Simple plugin to defer pytest-xdist hook functions."""

    def pytest_testnodown(self, node, error):
        """standard xdist hook function.
        """

    def pytest_configure(config):
        if config.pluginmanager.hasplugin("xdist"):
            config.pluginmanager.register(DeferPlugin())
```

这具有额外的好处,允许你根据安装的插件有条件地安装挂钩。

记录日志

caplog fixture方法

实时日志

版本记录

Pytest3.4中不兼容的更改

API参考

- 功能
 - `pytest.approx`
 - `pytest.fail`
 - `pytest.skip`
 - `pytest.importorskip`
 - `pytest.xfail`
 - `pytest.exit`
 - `pytest.main`
 - `pytest.param`
 - `pytest.raises`
 - `pytest.deprecated_call`
 - `pytest.register_assert_rewrite`
 - `pytest.warns`
 - `pytest.freeze_includes`
- 分数
 - `pytest.mark.filterwarnings`

- `pytest.mark.parametrize`
 - `pytest.mark.skip`
 - `pytest.mark.skipif`
 - `pytest.mark.usefixtures`
 - `pytest.mark.xfail`
 - 自定义标记
- 赛程
 - `@ pytest.fixture`
 - `config.cache`的
 - `capsys`
 - `capsysbinary`
 - `capfd`
 - `capfdbinary`
 - `doctest_namespace`
 - 请求
 - `pytestconfig`
 - `record_property`
 - `caplog`
 - 猴补丁
 - `TESTDIR`
 - `recwarn`
 - `tmp_path`
 - `tmp_path_factory`
 - `TMPDIR`
 - `tmpdir_factory`
- 钩
 - 引导钩子
 - 初始化挂钩
 - 测试运行挂钩
 - 收藏钩
 - 报告挂钩
 - 调试/交互挂钩
- 对象
 - `CallInfo`
 - 类
 - 集电极
 - 配置
 - `ExceptionInfo`
 - `FixtureDef`
 - `FSCollector`
 - 功能
 - 项目
 - `MarkDecorator`
 - `MarkGenerator`
 - 标记
 - `Metafunc`

- 模
- 节点
- 分析器
- 插件管理
- PytestPluginManager
- 会议
- 测试报告
- _结果
- 特殊变量
 - collect_ignore
 - collect_ignore_glob
 - pytest_plugins
 - pytest_mark
 - PYTEST_DONT_REWRITE(模块文档字符串)
- 环境变量
 - PYTEST_ADDOPTS
 - PYTEST_DEBUG
 - PYTEST_PLUGINS
 - PYTEST_DISABLE_PLUGIN_AUTOLOAD
 - PYTEST_CURRENT_TEST
- 配置选项

Functions

pytest.approx

断言两个数字(或两组数字)在某个容差范围内彼此相等。

由于浮点运算的复杂性,我们直觉期望相等的数字并不总是如此:

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

编写测试时通常会遇到此问题,例如,确保浮点值是您期望的值。处理此问题的一种方法是断言两个浮点数等于某个适当的容差范围内:

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

但是,这样的比较写作起来既乏味又难以理解。此外,通常不鼓励像上面这样的绝对比较,因为没有适合所有情况的容忍度。`1e-6`对周围的数字有好处¹,但对于非常大的数字而言太小而对于非常小的数字而言太大。最好将公差表示为预期值的一小部分,但这样的相对比较更难以正确和简洁地书写。

该`approx`类采用语法是尽可能直观执行浮点比较:

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

相同的语法也适用于数字序列:

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

字典值:

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

numpy 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
```

对于numpy标量的数组:

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

默认情况下, `approx` 将 $1e-6$ 其预期值的相对容差 (即百万分之一) 内的数字视为相等。如果预期值是这样的 `0.0`, 那么这种处理会产生令人惊讶的结果, 因为除了 `0.0` 本身之外什么都不是 `0.0`。为了不那么令人惊讶地处理这种情况, `approx` 还要考虑在 $1e-12$ 其预期值的绝对容差内的数字是相等的。Infinity 和 NaN 是特殊情况。无论相对容差如何, 无穷大只被视为与自身相等。默认情况下, NaN 不被视为等于任何内容, 但您可以通过将 `nan_ok` 参数设置为 `True` 来使其等于自身。(这是为了便于比较使用 NaN 的数组表示“无数据”。)

通过将参数传递给 `approx` 构造函数, 可以更改相对容差和绝对容差:

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

如果您指定`abs`但不指定`rel`,则比较将不会考虑相对容差。换句话说,`1e-6`如果超过指定的绝对容差,则默认相对容差范围内的两个数字仍将被视为不相等。如果同时指定两个 `abs`和`rel`,如果满足任一公差,则数字将被视为相等:

```
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
```

如果您正在考虑使用`approx`,那么您可能想知道它与比较浮点数的其他好方法的比较。所有这些算法都基于相对和绝对容差,并且应该在大多数情况下达成一致,但它们确实存在有意义的差异:

- `math.isclose(a, b, rel_tol=1e-9, abs_tol=0.0)`: `True`如果相对误差满足WRT无论是`a`或`b`,或者如果绝对容差得到满足。因为相对容差是`a`和两者一起计算的`b`,所以该测试是对称的(即既不是`a`也不 `b`是“参考值”)。如果要进行比较,则必须指定绝对容差,`0.0`因为默认情况下没有容差。仅在`python> = 3.5`中可用。[更多信息...]
- `numpy.isclose(a, b, rtol=1e-5, atol=1e-8)`: 如果和之间的差值小于相对容差`wrt` 和绝对容差之`a`和`b`则为真`b`。由于相对容差仅计算为`wrt b`,因此该测试是不对称的,您可以将其`b`视为参考值。支持比较序列由`numpy.allclose`。提供。[更多信息...]
- `unittest.TestCase.assertAlmostEqual(a, b)`: 如果`a`并且`b` 在绝对容差范围内,则为真`1e-7`。不考虑相对容差,并且不能改变绝对容差,因此该功能不适用于非常大或非常小的数字。此外,它只在子类中可用,`unittest.TestCase`并且它很难看,因为它不遵循PEP8。[更多信息...]
- `a == pytest.approx(b, rel=1e-6, abs=1e-12)`: 如果满足相对容差`b`或者满足绝对容差,则为真。由于相对容差仅计算为`wrt b`,因此该测试是不对称的,您可以将其`b`视为参考值。在明确指定绝对公差而非相对公差的特殊情况下,仅考虑绝对公差。

警告

在3.2版中更改。

为了避免不一致的行为,`TypeError`是提高了`>`,`>=`,`<`和`<=`比较。以下示例说明了问题:

```
assert 0.1 + 1e-10 # calls approx(0.1).__gt__(0.1 + 1e-10)
assert 0.1 + 1e-10 > approx(0.1) # calls approx(0.1).__lt__(0.1 + 1e-10)
```

在第二个例子中,人们期望 被调用。但相反,用于比较。这是因为丰富比较的调用层次结构遵循固定行为。[更多信息...]

pytest.fail

参考: `Skip`和`xfail`: 处理无法成功的测试

`fail(msg="", pytrace=True)`: 使用给定消息显式地设置用例为失败状态。

参数:

- **msg(str)** - 显示用户失败原因的消息。
- **pytrace(bool)** - 如果为false,则msg表示完整的失败信息,并且不报告任何python回溯。

pytest.skip

skip(msg[, allow_module_level=False]): 使用给定消息跳过测试用例。

应仅在测试(设置,调用或拆除)期间或使用`allow_module_level`标志在收集期间调用此函数。此函数也可以在doctests中调用。

参数:

- **allow_module_level(bool)** - 允许在模块级别调用此函数,跳过模块的其余部分。默认为False。

注意: 最好在可能的情况下使用`pytest.mark.skipif`标记来声明在某些条件下跳过的测试,例如不匹配的平台或依赖项。同样,使用`#doctest: + SKIP`指令(请参阅doctest.SKIP)可以静态地跳过doctest。

pytest.importorskip

importorskip(modname, minversion=None, reason=None): 导入并返回请求的模块`modname`,或者如果无法导入模块,则跳过当前测试。

参数:

- **modname(str)** - 要导入的模块的名称
- **minversion(str)** - 如果给定,导入的模块`__version__`属性必须至少为此最小版本,否则仍会跳过测试。
- **reason(str)** - 如果给定,则无法导入模块时,此原因显示为消息。

pytest.xfail

xfail(reason=""): 由于给定的原因,强制标记失败测试用例或测试准备函数。

只能在测试函数, setup函数或teardown函数中使用此函数。

注意:

最好在可能的情况下使用`pytest.mark.xfail`标记,在某些条件(如已知错误或缺少的功能)下声明测试是否为xfailed。

pytest.exit

exit(msg, returncode=None): 退出测试过程。

参数:

- **msg(str)** - 退出时显示的消息。
- **returncode(int)** - 返回退出pytest时使用的代码。

pytest.main

main(args=None, plugins=None): 执行进程内测试运行后返回退出代码。

参数:

- **args** - 命令行参数列表。
- **plugins** - 初始化期间要自动注册的插件对象列表。

pytest.param

`param(*values[, id][, marks])`: 在`[pytest.mark.parametrize]`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** - 按顺序的参数集值的变量args。
- 标记 - 要应用于此参数集的单个标记或标记列表。
- **id(str)** - 属于此参数集的id。

pytest.raises

参考: 关于预期异常的断言。

`with raises(expected_exception: Exception[, match][, message]) as excinfo`: 断言代码块/函数调用会引发 `expected_exception` 或引发失败异常。

参数:

- **match** - 如果指定,则断言异常与text或regex匹配
- 消息 - ****(自4.1弃用)****如果指定,提供了一种定制的失败消息,如果异常没有升高

使用`pytest.raises`的上下文管理器,这将捕获特定类型的异常:

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

如果代码块没有引发预期的异常(`ZeroDivisionError`在上面的示例中),或者根本没有异常,则检查将失败。

您还可以使用keyword参数`match`来断言异常与text或regex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+/pre>):
...     raise ValueError("value must be 42")
```

上下文管理器生成一个`ExceptionInfo`对象,可用于检查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"
```

自4.1版本后不推荐使用: 在上下文管理器表单中,您可以使用keyword参数 `message` 指定在`pytest.raises`检查失败时将显示的自定义失败消息。这已被弃用,因为它被认为是容易出错的,因为用户通常意味着使用它`match`。

注意

当`pytest.raises`用作上下文管理器时,值得注意的是,应用正常的上下文管理器规则,并且引发的异常必须是上下文管理器范围内的最后一行。之后,在上下文管理器范围内的代码行将不会被执行。例如:

```
value = 15
>>> with raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...     assert exc_info.type is ValueError # this will not execute
```

相反,必须采取以下方法(注意范围的差异):

```
with raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...
>>> assert exc_info.type is ValueError
```

使用 `pytest.mark.parametrize`

使用`pytest.mark.parametrize`时,可以对测试进行参数化,使得某些运行引发异常,而其他运行则不会。

有关示例,请参阅[参数化条件提升]。

遗产形式

可以通过传递一个名为`lambda`来指定一个可调用的:

```
raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ...>
```

或者你可以用参数指定一个任意的`callable`:


```
def f(x): return 1/x
...
>>> raises(ZeroDivisionError,f,0)
<ExceptionInfo ...>
>>> raises(ZeroDivisionError,f,x=0)
<ExceptionInfo ...>
```

上面的表单完全受支持,但不建议使用新代码,因为上下文管理器表单被认为更具可读性且不易出错。

注意: 与Python中捕获的异常对象类似,显式清除对返回`ExceptionInfo`对象的本地引用可以帮助Python解释器加速其垃圾回收。

清除这些引用会中断引用循环(`ExceptionInfo` -> 捕获异常 -> 帧堆栈引发异常 -> 当前帧堆栈 -> 局部变量 -> `ExceptionInfo`),这会使Python保留从该循环引用的所有对象(包括当前帧中的所有局部变量)活着,直到下一个循环垃圾收集运行。有关`try`更多信息,请参阅官方Python 语句文档。

pytest.deprecated_call

参考: 确保代码触发弃用警告。

with `deprecated_call()`: 上下文管理器,可用于确保代码块触发`DeprecationWarning`或`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
```

`deprecated_call`也可以通过传递函数来使用,`*args`并且`*kwargs`在这种情况下,它将确保调用产生上面的警告类型之一。`func(*args, **kwargs)`

pytest.register_assert_rewrite

参考: 断言重写。

`register_assert_rewrite(*names)`: 注册一个或多个要在导入时重写的模块名称。

此函数将确保此模块或程序包内的所有模块将重写其`assert`语句。因此,您应确保在实际导入模块之前调用此方法,如果您是使用包的插件,则通常在`__init__.py`中调用。

抛出: **TypeError** - 如果给定的模块名称不是字符串。

pytest.warns

参考: 使用警告功能发出警告

`with warns(expected_warning: Exception[, match]):` 断言代码会引发一类特定的警告。

具体来说,参数`expected_warning`可以是警告类或警告类序列,并且`with`块内部必须发出该类或类的警告。

该助手生成一个`warnings.WarningMessage`对象列表,每个警告引发一个对象。

此函数可用作上下文管理器,`pytest.raises`也可以使用任何其他方法:

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

在上下文管理器表单中,您可以使用keyword参数`match`来断言异常与`text`或`regex`匹配:

```
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+/pre>):
...     warnings.warn("value must be 42",UserWarning)

>>> with warns(UserWarning,match=r'must be \d+/pre>):
...     warnings.warn("this is not here",UserWarning)
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

pytest.freeze_includes

参考: 冻结pytest。

`freeze_includes()`: 返回pytest使用的模块名称列表,应由`cx_freeze`包含。

标记(Marks)

可以使用标记应用元数据来测试函数(但不是Fixture方法),然后可以通过Fixture方法或插件访问。

pytest.mark.filterwarnings

参考: @ `pytest.mark.filterwarnings`。 为标记的测试项添加警告过滤器。

`pytest.mark.filterwarnings(filter)`

参数: **filter(str)** - 一个警告规范字符串,由Python文档的“警告过滤器”部分中指定的元组(操作,消息,类别,模块,行号)的内容组成,以“:”分隔。 可以省略可选字段。 传递用于过滤的模块名称不是正则表达式转义。 例如:

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

pytest.mark.parametrize

参考: 参数化Fixture方法和测试函数。

Metafunc.`parametrize(argnames,argvalues,indirect = False,ids = None,scope = None)

使用给定argnames的argvalues列表向基础测试函数添加新调用。在收集阶段执行参数化。如果你需要设置昂贵的资源,请参阅设置间接,以便在测试设置时进行。

参数:

- **argnames**- 以逗号分隔的字符串,表示一个或多个参数名称,或参数字符串的列表/元组。
- **argvalues**- argvalues列表确定使用不同参数值调用测试的频率。如果只指定了一个argname,则argvalues是值列表。如果指定了N个argnames,则argvalues必须是N元组的列表,其中每个tuple-element为其各自的argname指定一个值。
- **indirect**- argnames或boolean的列表。参数列表名称(argnames的子集)。如果为True,则列表包含argnames中的所有名称。对应于此列表中的argname的每个argvalue将作为request.param传递到其各自的argname fixture函数,以便它可以在测试的设置阶段而不是在收集时执行更昂贵的设置。
- **ids**- 字符串ID列表或可调用的列表。如果字符串,则每个字符串对应于argvalues,以便它们是测试ID的一部分。如果将None作为特定测试的id给出,则将使用该参数的自动生成的id。如果是可调用的,它应该采用一个参数(单个argvalue)并返回一个字符串或返回None。如果为None,将使用该参数的自动生成的id。如果没有提供id,它们将自动从argvalues生成。
- **scope**- 如果指定,则表示参数的范围。范围用于按参数实例对测试进行分组。它还将覆盖任何fixture函数定义的范围,允许使用测试上下文或配置设置动态范围。

pytest.mark.skip

参考: 跳过测试函数。

pytest.mark.`skip(**,reason = None): 无条件地跳过测试函数。

参数:

- **reason(str)** - 跳过测试函数的原因。

pytest.mark.skipif

参考: 跳过测试函数。 如果条件是,则跳过测试函数True。

`pytest.mark.`skipif(条件,**,原因=无)[

参数:

- **condition(bool)**。
- **reason(str)** - 跳过测试函数的原因。

pytest.mark.usefixtures

参考: 使用类,模块或项目中的Fixture方法。 将测试函数标记为使用给定的Fixture方法名称。

警告 应用于**Fixture**方法函数时,该标记无效。

`pytest.mark.usefixtures(*名称)`

参数:

- **args**- 要使用的fixture的名称,作为字符串

pytest.mark.xfail

参考: XFail: 将测试函数标记为预期失败。 标记测试函数按 *预期失败*。

`pytest.mark.xfail(condition = None, **reason = None, raises = None, run = True, strict = False)`

参数:

- **condition**(*bool*)。
- **reason**(*str*) - 测试函数标记为xfail的原因。
- **raises**(*异常*) - 期望由测试函数引发的异常子类;其他例外将无法通过测试。
- **run**(*bool*) - 如果实际应该执行测试函数。如果`False`,该函数将始终为xfail并且不会被执行(如果函数是segfaulting则很有用)。
- **strict**(*布尔*) -
 - 如果`False`(默认值),该函数将在终端输出中显示,就xfailed好像它失败一样,就像xpass`它通过一样。在这两种情况下,这都不会导致测试套件整体失败。这对于标记稍后要解决的片状测试(随机失败的测试)特别有用。
 - 如果`True`,该函数将在终端输出中显示为xfailed失败,但如果它意外通过则将使测试套件失败。这对于标记始终失败的函数特别有用,并且应该有明确的指示它们是否意外地开始通过(例如,库的新版本修复了已知错误)。

自定义标记

标记是使用工厂对象动态创建的,pytest.mark并作为装饰器应用。 例如:

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

将创建并附加一个Mark。该mark对象将具有以下属性:

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

Fixtures方法

参考: pytest fixtures: 显式,模块化,可扩展。 测试函数或其他Fixtures通过将它们声明为参数名称来RequestFixtures。 需要Fixtures的测试示例如:

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

需要另一个Fixtures的Fixtures示例如:

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

有关更多详细信息,请参阅完整的[Fixture方法文档]。

@ pytest.fixture

@fixture(scope='function', params=None, autouse=False, ids=None, name=None): 装饰器标记Fixtures工厂方法。

可以使用该装饰器(带或不带参数)来定义Fixtures方法。稍后可以引用fixture函数的名称,以便在运行测试之前调用它: 测试模块或类可以使用`pytest.mark.usefixtures(fixturename)`标记。

测试函数可以直接使用Fixtures名称作为输入参数,在这种情况下,将注入从Fixtures函数返回的Fixtures实例。

Fixtures可以使用`return`或`yield`语句为测试函数提供它们的值。无论测试结果如何,`yield`在`yield`语句作为拆卸代码执行后使用代码块时,必须只生成一次。

参数:

- **scope**- 指此Fixtures共享范围的一个`"function"`(默认),
,,`"class"`或。`"module"``"package"``"session"``"package"`在这个时候被认为是实验性的。
- **params**- 一个可选的参数列表,它将导致多次调用fixture函数和使用它的所有测试。
- **autouse**- 如果为True,则为所有可以看到它的测试激活fixture命令。如果为False(默认值),则需要显式引用来激活Fixtures。
- **ids**- 每个对应于参数的字符串ID列表,以便它们是测试ID的一部分。如果没有提供id,它们将从params自动生成。
- **name**- Fixture方法的名称。这默认为装饰函数的名称。如果Fixtures在定义它的同一模块中使用,Fixtures的方法名称将被RequestFixtures的方法arg遮蔽;解决此问题的一种方法是命名装饰函数
`fixture_<fixturename>`,然后使用`@pytest.fixture(name='<fixturename>')`。

config.cache的

参考: 缓存: 使用跨testrun状态。该`config.cache`对象允许其他插件和Fixtures在测试运行中存储和检索值。要从Fixture方法Request访问它`pytestconfig`到你的Fixture方法并得到它`pytestconfig.cache`。在引擎盖下,缓存插件使用stdlib模块的简单`dumps/loadsAPIjson`。

Cache.get(key, default): 返回给定键的缓存值。如果尚未缓存任何值或无法读取值,则返回指定的默认值。

参数:

- **key**- 必须是/**分隔值**。通常,名字是你的插件或你的应用程序的名称。
- **default**- 必须在缓存未命中或缓存值无效的情况下提供。

Cache.set(key, value): 保存给定密钥的值。

参数:

- **key**- 必须是/**分隔值**。通常,名字是你的插件或你的应用程序的名称。
- **value**- 必须是基本python类型的任意组合,包括嵌套类型,例如字典列表。

Cache.mkdir(name): 返回具有给定名称的目录路径对象。如果该目录尚不存在,则将创建该目录。你可以使用它来管理文件,例如跨测试会话存储/检索数据库转储。

参数:

- **name**- 必须是不包含/**分隔符**的字符串。确保该名称包含你的插件或应用程序标识符,以防止与其他缓存用户冲突。

capsys

参考: 捕获stdout / stderr输出。

capsys(): 启用对**sys.stdout**和的写入文本捕获**sys.stderr**。

捕获的输出通过**capsys.readouterr()**方法调用提供,方法调用返回一个**namedtuple**。并且将对象。(out,err)**outerr`text`返回的实例CaptureFixture`**。

例如:

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

class CaptureFixture: **capsys()**, **capsysbinary()**, **capfd()** 及 **capfdbinary()** Fixtures函数所返回的对象。

readouterr(): 到目前为止,读取并返回捕获的输出,重置内部缓冲区。

返回: 捕获内容作为带有**out**和**err**字符串属性的**namedtuple**

with disabled(): 暂时禁用“with”块内的捕获。

capsysbinary

参考: 捕获stdout / stderr输出。

capsysbinary(): 启用字节捕获写入**sys.stdout**和**sys.stderr**。

捕获的输出通过`capsysbinary.readouterr()`方法调用提供,方法调用返回一个namedtuple。并且将对象。(out,err)outerr`bytes`返回的实例CaptureFixture`。例如:

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

capfd

参考: 捕获stdout / stderr输出。

capfd(): 允许写的文字捕捉到文件描述符1和2。捕获的输出通过`capfd.readouterr()`方法调用提供,方法调用返回一个namedtuple。并且将对象。(out,err)outerr`text`返回的实例CaptureFixture`。例如:

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

capfdbinary

参考: 捕获stdout / stderr输出。

capfdbinary(): 启用字节捕获文件描述符1和2。

捕获的输出通过`capfd.readouterr()`方法调用提供,方法调用返回一个namedtuple。并且将对象。(out,err)outerr`byte`

返回的实例CaptureFixture。例如:

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

doctest_namespace

参考: 模块和测试文件的Doctest集成。

doctest_namespace(): 返回一个dict将被注入doctests命名空间的Fixture。

通常这个Fixtures与另一个autouseFixtures一起使用:

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

有关更多详细信息: 'doctest_namespace' Fixtures。

Request

参考: 根据命令行选项将不同的值传递给测试函数。该 `request` Fixtures 是一个特殊的 Fixtures 提供 Request 测试用例的信息。

`class FixtureRequest`: 来着测试函数或 Fixtures 方法的请求。Request 对象提供对 Request 测试上下文的访问, 并且具有可选 `param` 属性, 以防 Fixtures 间接参数化。

`fixturename = None`: 正在执行此 Request 的 Fixtures

`scope = None`: 范围字符串, "方法", "类", "模块", "会话" 之一

`fixturenames`: 此 Request 中所有活动 Fixture 方法的名称

`node`: 底层集合节点(取决于当前 Request 范围)

`config`: 与此 Request 关联的 pytest 配置对象。

`function`: 如果 Request 具有按方法范围, 则测试函数对象。

`cls`: 收集测试函数的 class(可以是 None)。

`instance`: 收集测试函数的实例(可以是 None)。

`module`: 收集测试函数的 python 模块对象。

`fspath`: 收集此测试的测试模块的文件系统路径。

`keywords`: 底层节点的关键字/标记字典。

`session`: Pytest 会话对象。

`addfinalizer(finalizer)`: 在 Request 测试上下文完成执行的最后一次测试之后添加要调用的终结器/拆卸函数。

`applymarker(marker)`: 将标记应用于单个测试函数调用。如果你不希望在所有函数调用中都有关键字/标记, 则此方法很有用。

参数:

- **marker**-`_pytest.mark.MarkDecorator`。

`raiseerror(msg)`: 使用给定的消息引发 `FixtureLookupError`。

`getfixturevalue`: 动态运行命名 Fixtures 方法。

建议尽可能通过函数参数声明 Fixture 方法。但是, 如果你只能在测试设置时决定是否使用其他 Fixtures, 则可以使用此方法在 Fixtures 或测试用例体内检索它。

`getfuncargvalue(argname)`: 不推荐使用,请使用`getfixturevalue`。

pytestconfig

`pytestconfig()`: 返回`_pytest.config.Config`对象的会话范围的fixture。例如:

```
def test_foo(pytestconfig):
    if pytestconfig.getoption("verbose") > 0:
        ...
```

record_property

参考: `record_property`。

`record_property()`: 为调用测试添加额外的属性。用户属性成为测试报告的一部分,可供配置的报告者使用,如JUnit XML。该fixture可以调用,其值自动进行xml编码。(name,value) 例如:

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

record_testsuite_property

参考: `record_testsuite_property`

`record_testsuite_property()`: 将新的标记记录为根的子级。这适用于编写有关整个测试套件的全局信息,并且与xunit2 JUnit系列兼容。

这是一个会话范围的Fixture方法,用(name,value)调用。 例:

```
def test_foo(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")
```

name必须是一个字符串,value将被转换为字符串并正确地进行xml-escaped。

caplog

参考: 日志。

`caplog()`: 访问和控制日志捕获。 通过以下属性/方法可以获取捕获的日志:

```
string containing formatted log output
* caplog.records          -> list of logging.LogRecord instances
* caplog.record_tuples    -> list of (logger_name,level,message) tuples
* caplog.clear()          -> clear captured records and formatted log output string
```

这将返回一个`_pytest.logging.LogCaptureFixture`实例。

`class LogCaptureFixture(item)`: 提供对日志捕获的访问和控制。

handler

返回类型: `LogCaptureHandler`

`get_records(when)`: 获取其中一个可能的测试阶段的日志记录。

参数:

- **when(str)** - 从哪个测试阶段获取记录。有效值为: "setup","call"和"teardown"。

返回类型: 列表 [`logging.LogRecord`] 返回: 在给定阶段捕获的记录列表

版本3.4中的新方法。

`text`: 返回日志文本。

`records`: 返回日志记录列表。

`record_tuples`: 返回用于断言比较的日志记录的精简版本的列表。元组的格式是:
(`logger_name`,`log_level`,`message`)`

`messages`: 返回格式插值日志消息的列表。

与包含格式字符串和插值参数的“记录”不同,此列表中的日志消息都是内插的。与包含处理程序输出的“text”不同,此列表中的日志消息与级别,时间戳等一致,使得精确比较更可靠。

请注意,不包括回溯或堆栈信息(来自`logging.exception()`或记录函数的`exc_info`或`stack_info`参数),因为这是由处理程序中的格式化程序添加的。

版本3.7中的新方法。

`clear()`: 重置日志记录列表和捕获的日志文本。

`set_level(level, logger=None)`: 设置捕获日志的级别。在测试结束时,该级别将恢复到之前的值。

参数:

- **level(int)** - 记录器到级别。
- **logger(str)** - 更新级别的记录器。如果未给出,则更新根记录器级别。

*版本3.4中更改: *此方法更改的记录器级别将在测试结束时恢复为其初始值。

`with at_level(level, logger=None)`: 上下文管理器,用于设置捕获日志的级别。在'with'语句结束后,级别将恢复为其原始值。

参数:

- **level(int)** - 记录器到级别。
- **logger(str)** - 更新级别的记录器。如果未给出,则更新根记录器级别。

Monkeypatching

参考: Monkeypatching / mocking 模块和环境。

返回的`monkeypatch` fixture提供了这些辅助方法来修改对象,字典或`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)
```

在Request测试用例或Fixtures完成后,所有修改都将被撤消。`raising`如果设置/删除操作没有目标,则该参数确定是否将引发`KeyError`或`AttributeError`。这将返回一个`MonkeyPatch`实例。

`class MonkeyPatch`: `monkeypatch` Fixtures返回的对象保留了`setattr` / `item` / `env` / `syspath`更改的记录。

`with context()`: 上下文管理器返回一个新`MonkeyPatch`对象,该对象撤消`with`在退出时在块内完成的任何修补:

```
import functools
def test_partial(monkeypatch):
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
```

在测试结束之前需要撤消一些补丁的情况下很有用,例如`stdlib`模拟可能会在模拟时破坏`pytest`本身的函数(例如,参见[#3290])。

`setattr(target, name, value=, raising=True)`: 在目标上设置属性值,记住旧值。如果该属性不存在,则默认引发`AttributeError`。

为方便起见,你可以指定一个字符串`target`,将其解释为虚线导入路径,最后一部分是属性名称。示例如:将设置模块的方法。`monkeypatch.setattr("os.getcwd", lambda: "/")`getcwd` os`

该`raising`值确定如果属性尚不存在,`setattr`是否应该失败(默认为`True`,这意味着它将引发)。

`delattr(target, name=, raising=True)`: 删除属性`name`的`target`,默认情况下提高`AttributeError`的它的属性,以前不存在的。

如果未`name`指定且`target`为字符串,则将其解释为虚线导入路径,最后一部分为属性名称。

如果`raising`设置为`False`,则缺少属性时不会引发异常。

`setitem(dic, name, value)`: 将字典条目设置`name`为值。

`delitem(dic, name, raising=True)`: `name`从`dict`删除。如果不存在则引发`KeyError`。

如果`raising`设置为`False`,则如果缺少密钥,则不会引发异常。

`setenv(name, value, prepend=None)`: 将环境变量设置`name`为`value`。如果`prepend`是字符,请读取当前环境变量值并`value`在`prepend`字符旁边添加前缀。

`delenv(name, raising=True)`: `name`从环境中删除。如果不存在则引发`KeyError`。

如果`raising`设置为`False`,则在缺少环境变量时不会引发异常。

`syspath_prepend(path)`: 前置`path`到`sys.path`导入位置列表。

`chdir(path)`: 将当前工作目录更改为指定的路径。`Path`可以是字符串或`py.path.local`对象。

`undo()`: 撤消以前的更改。此调用使用撤消堆栈。除非你在撤消调用后执行更多`monkeypatching`,否则再次调用它无效。

通常不需要调用`undo()`,因为它在拆除期间会自动调用。

请注意,在单个测试函数调用中使用相同的`monkeypatchfixture`。如果测试函数本身和其中一个测试`Fixtures`同时使用`monkeypatch`,则调用`undo()`将撤消两个函数中所做的所有更改。

testdir

此`fixture`提供了一个`Testdir`对测试文件的黑盒测试有用的实例,使其成为测试插件的理想选择。

要使用它,请包含在最顶层的`conftest.py`文件中:

```
pytest_plugins = 'pytester'
```

`class Testdir`: 临时测试目录,带有测试/运行`pytest`本身的工具。这是基于`tmpdir``Fixtures`,但提供了许多方法,有助于测试`pytest`本身。除非`chdir()`使用,否则所有方法都将`tmpdir`用作其当前工作目录。属性:

`Tmpdir`: `py.path.local`临时目录的实例。 `Plugins`: 与`parseconfig()`。最初这是一个空列表,但插件可以添加到列表中。要添加到列表中的项目类型取决于使用它们的方法,因此请参阅它们以获取详细信息。

`CLOSE_STDIN`: ``builtins.object``的别名。

exception `TimeoutExpired`

`finalize()`: 清理全局状态工件。

一些方法修改全局解释器状态,这会尝试清除它。它不会删除临时目录,因此可以在测试运行完成后查看它。

`make_hook_recorder(pluginmanager)`: `HookRecorder`为`PluginManager`创建一个新的。

`chdir()`: `cd`进入临时目录。

这在实例化时自动完成。

`makefile(ext, *args, **kwargs)`: 在`testdir`中创建新文件。

参数:

- **ext(str)** - 文件应使用的扩展名,包括点,例如`.py`。

- **args(list*)** - 所有args将被视为字符串并使用换行符连接。结果将作为内容写入文件。该文件的名称将基于Request此Fixtures的测试用例。
- **kwargs**- 每个关键字都是文件的名称,而它的值将被写为文件的内容。

例子:

```
testdir.makefile(".txt", "line1", "line2")
testdir.makefile(".ini", pytest="[pytest]\naddopts=-rs\n")
```

makeconftest(source): 使用'source'作为内容写一个conftest.py文件。

makeini(source): 使用'source'作为内容写一个tox.ini文件。

getinifg(source): 从tox.ini配置文件返回pytest部分。

makepyfile(*args, **kwargs): makefile()的扩展名为.py的快捷方式。

maketxtfile(*args, **kwargs): makefile()的扩展名为.txt的快捷方式。

syspathinsert(path=None): 将目录添加到sys.path,默认为tmpdir。

当该对象在每次测试结束时死亡时,会自动撤消。

mkdir(name): 创建一个新的(子)目录。

mkpydir(name): 创建一个新的python包。

这会创建一个带有空__init__.py文件的(子)目录,因此它被识别为python包。

class Session(config)

exception Failed: 在测试运行失败时发出停止信号。

exception Interrupted: 发出中断的测试运行信号。

for ... in collect(): 返回此集合节点的子项(项和收集器)列表。

getnode(config, arg): 返回文件的集合节点。

参数:

- **config**-_pytest.config.Config创建配置
- **arg**-py.path.local文件的一个实例

getpathnode(path): 返回文件的集合节点。

这就好比getnode()创建(配置)pytest配置实例。

参数:

- **path**-py.path.local文件的一个实例

genitems(colitems): 从集合节点生成所有测试项。 这将递归到集合节点并返回其中包含的所有测试项的列表。

`runitem(source)`: 运行“test_func”项。

调用测试实例(包含测试用例的类)必须提供一个`.getrunner()`方法,该方法应该返回一个可以为单个项运行测试协议的运行器,例如`_pytest.runner.runtestprotocol()`。

`inline_runsource(source, *cmdlineargs)`: 使用中运行测试模块`pytest.main()`。

此运行将“source”写入临时文件并`pytest.main()`在其上运行,返回`HookRecorder`结果的实例。

参数:

- **source**- 测试模块的源代码
- **cmdlineargs**- 要使用的任何额外命令行参数

返回: `HookRecorder`结果的实例

`inline_genitems(*args)`: `pytest.main(['--collectonly'])`在进程中运行。

运行`pytest.main()`函数在测试过程本身内运行所有`pytestinline_run()`,但返回收集项和`HookRecorder`实例的元组。

`inline_run(*args, plugins=(), no_reraise_ctrlc=False)`: `pytest.main()`在进程中运行,返回`HookRecorder`。

运行该`pytest.main()`函数以在测试过程本身内运行所有`pytest`。这意味着它可以返回一个`HookRecorder`实例,该实例从该运行中提供比通过匹配`stdout` / `stderr`可以完成的更详细的结果`runpytest()`。

参数:

- **args**- 要传递给的命令行参数`pytest.main()`
- 插件- (仅限关键字)`pytest.main()`实例应使用的额外插件实例

返回: 一个`HookRecorder`实例

`runpytest_inprocess(*args, **kwargs)`: 返回运行`pytest in-process`的结果,提供与`self.runpytest()`提供的类似的接口。

`runpytest(*args, **kwargs)`: 运行`pytest`内联或子进程,具体取决于命令行选项“-runpytest”并返回`aRunResult`。

`parseconfig(*args)`: 从给定的命令行`args`返回一个新的`pytest Config`实例。

这将调用`_pytest.config`中的`pytest`引导代码来创建一个新的`_pytest.core.PluginManager`并调用`pytest_cmdline_parse`挂钩来创建一个新`_pytest.config.Config`实例。如果`plugins`已经填充,则应该是要使用`PluginManager`注册的插件模块。

`parseconfigure(*args)`: 返回新的`pytest`配置的`Config`实例。

这会返回一个新`_pytest.config.Config`,但也会调用`pytest_configure`钩子方法。

`getitem(source, funcname='test_func')`: 返回测试项目以获得测试用例。

这会将源写入`python`文件并在生成的模块上运行`pytest`的集合,返回所`Request`的函数名称的测试项。

参数:

- **source**- 模块源

- **funcname**- 要为其返回测试项的测试函数的名称

`getitems(source)`: 返回从模块收集的所有测试项目。

这会将源写入python文件并在生成的模块上运行pytest的集合,返回其中包含的所有测试项。

`getmodulecol(source, configargs=(), withinit=False)`: 返回模块集合节点`source`。

这将写入`source`文件`makepyfile()`,然后在其上运行pytest集合,返回测试模块的集合节点。

参数:

- **source**- 要收集的模块的源代码
- **configargs**- 要传递给的任何额外参数`parseconfigure()`
- **withinit**- 是否也将`__init__.py`文件写入同一目录以确保它是一个包

`collect_by_name(modcol, name)`: 从模块集合返回name的集合节点。

这将在模块集合节点中搜索与给定名称匹配的集合节点。

参数:

- **modcol**- 模块集合节点;看到`getmodulecol()`
- **name**- 要返回的节点的名称

`popen(cmdargs, stdout=-1, stderr=-1, stdin=<class 'object'>, **kw)`: 调用`subprocess.Popen`。

这会调用`subprocess.Popen`,确保当前工作目录在PYTHONPATH中。 你可能想要使用`run()`。

`run(*cmdargs, timeout=None, stdin=<class 'object'>)`: 运行带参数的命令。

使用`subprocess.Popen`运行进程保存`stdout`和`stderr`。

参数:

- **args**- 传递给`subprocess.Popen()`的参数序列
- **timeout**- 超时和提升之后的秒数`Testdir.TimeoutExpired`

返回一个`RunResult`。

`runpython(script)`: 使用`sys.executable`作为解释器运行python脚本。 返回一个`RunResult`。

`runpython_c(command)`: 运行`python -c"command"`,返回一个`RunResult`。

`runpytest_subprocess(*args, timeout=None)`: 运行pytest作为具有给定参数的子进程。

添加到`plugins`列表中的所有插件都将使用`-p`命令行选项添加。另外`--basetemp`用于将任何临时文件和目录放在带有“`runpytest-`”前缀的编号目录中,以避免与临时文件和目录的正常编号pytest位置冲突。

参数:

- **args**- 传递给pytest子进程的参数序列
- **timeout**- 超时和提升之后的秒数`Testdir.TimeoutExpired`

返回一个`RunResult`。

`spawn_pytest(string, expect_timeout=10.0)`: 使用`pexpect`运行`pytest`。

这确保使用正确的`pytest`并设置临时目录位置。

返回`pexpect`子对象。

`spawn(cmd, expect_timeout=10.0)`: 使用`pexpect`运行命令。

返回`pexpect`子对象。

`class RunResult`: 运行命令的结果。

属性:

`Ret`: 返回值 `Outlines`: 从`stdout`捕获的行列表 `Errlines`: 从`stderr`捕获的行列表 `Stdout`: `LineMatcher`方法 `Stderr`: `LineMatcher` `stderr` `Duration`: 持续时间(秒)

`parseoutcomes()`: 从解析测试过程产生的终端输出返回`outcometrings -> num`的字典。

`assert_outcomes(passed=0, skipped=0, failed=0, error=0, xpassed=0, xfailed=0)`: 断言在测试运行的文本输出中,指定的结果与相应的数字一起出现(0表示未发生)。

`class LineMatcher`: 灵活的文本匹配。

这是一个测试大文本(例如命令输出)的便利类。

构造函数采用一系列行而没有它们的尾随换行符,即`text.splitlines()`。

`str()`: 返回整个原始文本。

`fnmatch_lines_random(lines2)`: 使用任何顺序在输出中存在检查行。

使用检查行`fnmatch.fnmatch`。参数是必须以任何顺序出现在输出中的行列表。

`re_match_lines_random(lines2)`: 输出中的检查行使用`re.match`任何顺序。

参数是必须以任何顺序出现在输出中的行列表。

`get_lines_after(fnline)`: 返回文本中给定行后面的所有行。 给定的行可以包含`glob`通配符。

`fnmatch_lines(lines2)`: 使用搜索匹配的行搜索捕获的文本`fnmatch.fnmatch`。

参数是必须匹配并且可以使用`glob`通配符的行列表。如果它们不匹配,则调用`pytest.fail()`。匹配和不匹配也打印在标准输出上。

`re_match_lines(lines2)`: 使用搜索匹配的行搜索捕获的文本`re.match`。

参数是必须匹配的行列表`re.match`。如果它们不匹配,则调用`pytest.fail()`。

匹配和不匹配也打印在标准输出上。

recwarn

参考: 警告断言

返回`WarningsRecorder`记录测试函数发出的所有警告的实例。有关警告类别的信息,请参阅:警告断言。

`class WarningsRecorder`: 用于记录引发警告的上下文管理器。改编自`warnings.catch_warnings`。

`list`: 记录的警告列表。

`pop(cls=<class 'Warning'>)`: 弹出第一个录制的警告,如果不存在则引发异常。

`clear()`: 清除录制的警告列表。

每个记录的警告都是一个例子`warnings.WarningMessage`。


注意: `RecordedWarning`在pytest 3.1中从普通类改为命名元组

注意: `DeprecationWarning`并且`PendingDeprecationWarning`区别对待;请参阅[确保代码触发弃用警告]。

tmp_path

参考: 临时目录和文件

`tmp_path()`: 返回临时目录路径对象,该对象对于每个测试函数调用是唯一的,创建为基本临时目录的子目录。返回的对象是一个`pathlib.Path`对象。

注意: 在python .6中,这是一个`pathlib2.Path`

tmp_path_factory

参考: `tmp_path_factory`Fixtures `tmp_path_factory`实例有以下方法:

`TempPathFactory.mktemp(basename, numbered=True)`: 制作工厂管理的临时目录

`TempPathFactory.getbasetemp()`: 返回基本临时目录。

tmpdir

参考: 临时目录和文件

`tmpdir()`: 返回临时目录路径对象,该对象对于每个测试函数调用是唯一的,创建为基本临时目录的子目录。返回的对象是`[py.path.local]`路径对象。

tmpdir_factory

参考: `'tmpdir_factory'`Fixtures `tmpdir_factory`实例有以下方法:

创建基本临时目录的子目录并将其返回。如果`numbered`,通过添加大于任何现有的数字前缀来确保该目录是唯一的。

向后compat装饰器 `_tmppath_factory.getbasetemp`

Hooks

参考: 编写插件。

引用可由`[conftest.py]`文件实现的所有Hook方法。

引导时的Hook方法

引导时的Hook方法要求尽早注册插件(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初始`conftest`文件的加载。

注意: 不会为`conftest.py`文件调用此Hook方法,仅适用于`setuptools`插件。

参数:

- **early_config**(`_pytest.config.Config`) - `pytest`配置对象
- **args**(`list*`) - 在命令行上传递的参数列表
- **解析器**(`_pytest.config.Parser`) - 添加命令行选项

`pytest_cmdline_preparse(config,args)`: (不推荐)在选项解析之前修改命令行参数。此钩子被认为已弃用,将在未来的`pytest`版本中删除。考虑`pytest_load_initial_conftests()`改用。

注意: 不会为`conftest.py`文件调用此Hook方法,仅适用于`setuptools`插件。

参数:

- **config**(`_pytest.config.Config`) - `pytest`配置对象
- **args**(`list*`) - 在命令行上传递的参数列表

`pytest_cmdline_parse(pluginmanager,args)`: 返回初始化的配置对象,解析指定的`args`。在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

注意: `plugins`当使用`[pytest.main]`为传递给`arg`的插件类调用此挂接。

参数:

- **pluginmanager**(`_pytest.config.PytestPluginManager`) - `pytest`插件管理器
- **args**(`list*`) - 在命令行上传递的参数列表

`pytest_cmdline_main(config)`: 要求执行主命令行动作。默认实现将调用`configure hooks`和`runtest_mainloop`。

注意: 不会为`conftest.py`文件调用此Hook方法,仅适用于`setuptools`插件。在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

参数:

- **config**(`_pytest.config.Config`) - `pytest`配置对象

初始化时的Hook方法

初始化时的Hook方法调用插件和`conftest.py`文件。

`pytest_addoption(解析器)`: 注册`argparse`-style选项和`ini`-style配置值,在测试运行开始时调用一次。

注意: `conftest.py`由于`pytest`[在启动期间发现插件的方式,此函数应仅在位于测试根目录的插件或文件中实现。

参数:

- 解析器(`_pytest.config.Parser`) - 要添加命令行选项,请调用`parser.addoption(...)`。添加ini文件值调用`parser.addini(...)`。

以后可以`config`分别通过对象访问选项:

- `config.getoption(name)`检索命令行选项的值。
- `config.getini(name)`检索从ini样式文件中读取的值。

配置对象通过`.config`属性在许多内部对象上传递,或者可以作为`pytestconfig` Fixture方法检索。

注意: 这个Hook方法与`hookwrapper=True`冲突。

`pytest_addhooks(pluginmanager)`: 在插件注册时调用,允许通过调用添加新的Hook方法。

`pluginmanager.add_hookspecs(module_or_class, prefix)`

参数:

- `pluginmanager(_pytest.config.PytestPluginManager)` - pytest插件管理器

注意: 这个钩子与之不相容`hookwrapper=True`。

`pytest_configure(config)`: 允许插件和`conftest`文件执行初始配置。

在解析了命令行选项后,为每个插件和初始`conftest`文件调用此Hook方法。之后,在导入钩子时会调用其他`conftest`文件。

注意: 这个钩子与之不相容`hookwrapper=True`。

参数:

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

`pytest_unconfigure(config)`: 在退出测试过程之前调用。

参数:

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

`pytest_sessionstart(session)`: 在`Session`创建对象之后以及执行收集和进入运行测试循环之前调用。

参数:

- `session(_pytest.main.Session)` - pytest会话对象

`pytest_sessionfinish(session, exitstatus)`: 在整个测试运行完成之后调用,在将退出状态返回到系统之前。

参数:

- `session(_pytest.main.Session)` - pytest会话对象
- `exitstatus(int)` - pytest将返回系统的状态

`pytest_plugin_registered(plugin, manager)`: 一个新的pytest插件已注册。

参数:

- 插件- 插件模块或实例
- **manager**(`_pytest.config.PytestPluginManager`) - pytest插件管理器

注意: 这个钩子与之不相容`hookwrapper=True`。

测试运行时的Hook方法

所有与`runtest`相关的钩子都会收到一个`pytest.Item`对象。

`pytest_runtestloop`(会话): 要求执行主运行测试循环(收集完成后)。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

参数:

- **session**(`_pytest.main.Session`) - pytest会话对象

`pytest_runtest_protocol(item,nextitem)`: 为给定的测试项实现`runtest_setup` / `call` / `teardown`协议,包括捕获异常和调用报告Hook方法。

参数:

- **item**- 为其执行运行测试协议的测试项目。
- **nextitem**- 预定下一个测试项目(如果这是我朋友的结束,则为无)。这个论点被传递给了`pytest_runtest_teardown()`。

返回布尔值: 如果不应调用其他钩子实现,则为`True`。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

`pytest_runtest_logstart(nodeid,location)`: 发出运行单个测试项目的信号。

在调用`pytest_runtest_setup()`Hook方法之前会调用此Hook方法。

参数:

- **nodeid**(*str*) - 项目的完整ID
- **location**- 三倍(filename,linenum,testname)`

`pytest_runtest_logfinish(nodeid,location)`: 发出运行单个测试项目的完整信号。

调用`pytest_runtest_setup()`Hook方法之后, 调用此Hook方法。

参数:

- **nodeid**(*str*) - 项目的完整ID
- **location**- (filename,linenum,testname)`三者的组合

`pytest_runtest_setup(item)`: 以前称作`pytest_runtest_call(item)`。

`pytest_runtest_call(item)`: 调用以执行测试项。

`pytest_runtest_teardown(item,nextitem)`: 在`pytest_runtest_call`之后调用。

参数:

- **nextitem**- 计划下一个测试项目(如果没有安排其他测试项目,则为None)。这个参数可以用来执行精确的拆卸,即调用足够的终结器,以便nextitem只需要调用setup-functions。

`pytest_runtest_makereport(item,call)`: 返回`_pytest.runner.TestReport`。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

为了更深入地理解,你可以查看这些Hook方法的默认实现,`_pytest.runner`也可能与其`_pytest.pdb`进行交互
`_pytest.capture`以及其输入/输出捕获,以便在发生测试失败时立即进入交互式调试。

在`_pytest.terminal`具体报告使用报告Hook方法,打印有关测试运行的信息。

`pytest_pyfunc_call(pyfuncitem)`: 调用底层测试函数。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

收集用例时的Hook方法

`pytest`调用以下Hook方法来收集测试文件及目录:

`pytest_collection(session)`: 执行给定会话的收集协议。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止。

参数:

- **session**(`_pytest.main.Session`) - `pytest`会话对象

`pytest_ignore_collect(path,config)`: 返回True以防止考虑此收集路径。在调用更具体的钩子之前,会查询所有文件和目录的钩子。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

参数:

- **path**(`str`) - 要分析的路径
- **config**(`_pytest.config.Config`) - `pytest`配置对象

`pytest_collect_directory(path,parent)`: 在遍历目录以获取集合文件之前调用。

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

参数:

- **path**(`str`) - 要分析的路径

`pytest_collect_file(path,parent)`: 返回集合给定路径的节点或无。任何新节点都需要将指定`parent`的父节点作为父节点。

参数:

- **path**(`str`) - 要收集的路径

`pytest_pycollect_makemodule(路径, 父母)`: 返回给定路径的Module收集器或None。将为每个匹配的测试模块路径调用此Hook方法。如果要为不匹配的文件创建测试模块作为测试模块,则需要使用`pytest_collect_file`Hook方法。

在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

要影响Python模块中的对象集合,可以使用以下Hook方法:

`pytest_pycollect_makeitem(收藏家, 名字, obj)`: 返回模块中python对象的自定义项/收集器,或者无。

在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

`pytest_generate_tests(metafunc)`: 生成(多个)参数化调用到测试函数。

`pytest_make_parametrize_id(config, val, argname)`: 返回`val@ pytest.mark.parametrize`调用将使用的给定的用户友好字符串表示形式。如果钩子不知道,则返回None`val`。`argname`如果需要,参数名称可用。

在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

参数:

- **config**(`_pytest.config.Config`) - pytest配置对象
- **val**- 参数化值
- **argname**(`str`) - pytest生成的自动参数名称

收集完成后,你可以修改项目的顺序,删除或以其他方式修改测试项目:

`pytest_collection_modifyitems(会话, 配置, 项目)`: 在执行收集后调用,可以就地过滤或重新排序项目。

参数:

- **session**(`_pytest.main.Session`) - pytest会话对象
- **config**(`_pytest.config.Config`) - pytest配置对象
- **items**(`List***_pytest.nodes.Item**`*) - 项目对象列表

`pytest_collection_finish(会话)`: 在执行和修改集合后调用。

参数:

- **session**(`_pytest.main.Session`) - pytest会话对象

生成测试结果时的Hook方法

与会话报告相关的Hook方法:

`pytest_collectstart(collector)`: Collector开始收集。

`pytest_make_collect_report(collector)`: 执行`collector.collect()`并返回CollectReport。 在第一个非空结果处停止,请参见: `firstresult`: 在第一个非空结果处停止

`pytest_itemcollected(item)`: 我们刚收集了一个测试项目。

`pytest_collectreport(report)`: Collector完成收集。

`pytest_deselected(items)`: 要求通过关键字取消选择的测试项目。

`pytest_report_header(config, startdir)`: 返回一个字符串或字符串列表,以显示为终端报告的标题信息。

参数:

- **config**(`pytest.config.Config`) - pytest配置对象
- **startdir**- 带起始目录的`py.path`对象

注意: `conftest.py`由于pytest[在启动期间发现插件的方式,此函数应仅在位于测试根目录的插件或文件中实现。

`pytest_report_collectionfinish(config, startdir, items)`: 返回集合成功完成后要显示的字符串或字符串列表。此字符串将显示在标准的“收集的X项目”消息之后。

版本3.2中的新函数。

参数:

- **config**(`pytest.config.Config`) - pytest配置对象
- **startdir**- 带起始目录的`py.path`对象
- **items**- 将要执行的pytest项列表;此列表不应修改。

`pytest_report_teststatus(报告,配置)`: 返回结果类别,简短和冗长的报告单词。

参数:

- **config**(`pytest.config.Config`) - pytest配置对象

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

`pytest_terminal_summary(terminalreporter, exitstatus, config)`: 在终端摘要报告中添加一个部分。

参数:

- **terminalreporter**(`pytest.terminal.TerminalReporter`) - 内部终端报告对象
- **exitstatus**(`int`) - 将报告回操作系统的退出状态
- **config**(`pytest.config.Config`) - pytest配置对象

新的4.2版: 该`config`参数。

`pytest_fixture_setup(fixturedef, request)`: 执行Fixture方法设置执行。

返回: 调用fixture函数的返回值

在第一个非空结果处停止,请参见: **firstresult**: 在第一个非空结果处停止

注意: 如果fixture函数返回None,则将根据`firstresult`选项。

`pytest_fixture_post_finalizer(fixturedef, request)`: 在Fixture方法拆卸后调用,但在清除缓存之前, `fixturedef.cached_result`仍然可以访问Fixture方法结果缓存。

`pytest_warning_captured(warning_message, when, item)`: 处理内部pytest warnings插件捕获的警告。

参数:

- **warning_message**(`warnings.WarningMessage`) - 捕获的警告。这与生成的对象相同 `warnings.catch_warnings()`, 并且包含与参数相同的属性 `warnings.showwarning()`。
- **when**(`str`) - 指示何时捕获警告。可能的值:
 - **"config"**: 在pytest配置/初始化阶段。
 - **"collect"**: 在测试收集期间。
 - **"runtest"**: 在测试执行期间。
- **item**(`pytest.Item` `None`) - **DEPRECATED**: 此参数与以后版本不兼容 `pytest-xdist`, 并且将始终 `None` 在以后的版本中接收。正在执行的项目, 如果 `when` 是 **"runtest"**, 否则 `None`。

这是报告测试执行的中心Hook方法:

`pytest_runtest_logreport(report)`: 处理与执行测试的相应阶段有关的测试设置/调用/拆除报告。

断言相关Hook方法:

`pytest_assertrepr_compare(config, op, left, right)`: 返回失败的断言表达式中的比较解释。

如果没有自定义说明, 则返回 `None`, 否则返回字符串列表。该字符串将通过新行被加入, 但任何换行符 `\n` 在字符串将被转义。请注意, 除第一行外的所有行都将略微缩进, 目的是将第一行作为摘要。

参数:

- **config**(`_pytest.config.Config`) - pytest配置对象

`pytest_assertion_pass(item, lineno, orig, expl)`: 每当断言通过时, Hook就会调用。

在传递断言后使用此Hook方法进行一些处理。原始断言信息在 `orig` 字符串中可用, 而pytest内省断言信息在 `expl` 字符串中可用。

必须通过 `enable_assertion_pass_hook` ini-file 选项显式启用此Hook方法:

```
[pytest]
enable_assertion_pass_hook=true
```

启用此选项时, 需要清除项目目录和解释器库中的 `.pyc` 文件, 因为断言需要重写。

参数:

- **item**(`_pytest.nodes.Item`) - 当前测试的pytest项目对象
- **lineno**(`int`) - 断言语句的行号
- **orig**(`string`) - 带有原始断言的字符串
- **expl**(`string`) - 带有断言解释的字符串

注意: 这个Hook方法是实验性的, 因此它的参数甚至钩子本身可能会在未来的任何pytest版本中被更改/删除而不会发出警告。

如果您发现此Hook有用, 请分享您的反馈, 以解决问题。

调试/交互Hook方法

很少有Hook方法可用于特殊报告或与异常交互:

`pytest_internalerror(excrepr, excinfo)`: 内部出错时调用。

`pytest_keyboard_interrupt(excinfo)`: 键盘中断时调用。

`pytest_exception_interact(节点, 呼叫, 报告)`: 在引发异常时调用, 可以交互式处理。

只有在引发的异常不是内部异常, 如`skip.Exception`时才会调用此Hook方法。

`pytest_enter_pdb(config, pdb)`: 调用`pdb.set_trace()`时, 插件可以使用插件在python调试器进入交互模式之前采取特殊操作。

参数:

- **config**(`_pytest.config.Config`) - pytest配置对象
- **pdb**(`pdb.Pdb`) - Pdb实例

Objects

Objects的完整使用可参考: `Fixturs`及`Hooks`方法。

CallInfo

`class CallInfo`: 结果/异常信息是一个函数调用。

Class

`class Class`: 基类: `_pytest.python.PyCollector`

收集器的测试用例。

`collect()`: 返回此集合节点的子项(项和收集器)列表。

Collector

`class Collector`: 基类 `_pytest.nodes.Node`

收集器实例通过`collect()`创建子项, 从而迭代地构建树。

exception `CollectError`: 基类 `Exception`

收集期间出错, 包含自定义消息。

`collect()`: 返回此集合节点的子项(项和收集器)列表。

`repr_failure(excinfo)`: 表示集合失败。

Config

`class Config`: 访问配置值, `pluginmanager`和插件钩子。 `option=None`: 访问命令行选项作为属性。(已弃用), 请 `getoption()` 改用

`pluginmanager=None`: 一个`pluginmanager`实例

`add_cleanup(func)`: 添加一个在配置对象不使用时调用的函数(通常与`pytest_unconfigure`一致)。

`classmethod fromdictargs(option_dict,args)`: 构造函数可用于子进程。

`addinvalue_line(name,line)`: 在ini-file选项中添加一行。该选项必须已声明但可能尚未设置,在这种情况下,该行将成为其值中的第一行。

`getini(name)`: 从ini文件返回配置值。如果未通过先前`parser.addini`调用(通常来自插件)注册指定的名称,则会引发`ValueError`。

`getoption(name, default=, skip=False)`: 返回命令行选项值。

参数:

- **name**- 选项的名称。你也可以指定文字`--OPT`选项而不是“dest”选项名称。
- **default**- 如果不存在该名称的选项,则为默认值。
- **skip**- 如果选项不存在或具有None值,则为true raise `pytest.skip`。

`getvalue(name,path = None)`: (不推荐使用,请使用`getoption()`)

`getvalueorskip(name,path = None)`: (不推荐使用,使用`getoption(skip = True)`)

ExceptionInfo

`class ExceptionInfo(excinfo,striptext = "",traceback = None)`: 包装`sys.exc_info()`对象并提供导航回溯的帮助。

`classmethod from_current(exprinfo = None)` 返回与当前回溯匹配的`ExceptionInfo`

警告 实验性API

参数: **exprinfo**- 一个文本字符串,帮助确定我们是否应该`AssertionError`从输出中剥离,默认为异常消息/`__str__()`

`classmethod for_later()` 返回一个未填充的`ExceptionInfo`

type: 异常类

value: 异常值

tb: 异常原始追溯

typename: 异常的类型名称

traceback: 追溯信息

`exonly(tryshort = False)`: 将异常作为字符串返回

当'tryshort'解析为True,异常是`_pytest._code._AssertionError`时,只返回异常表示的实际异常部分(因此'AssertionError: '从头开始删除)

`errisinstance(exc)`: 如果异常是exc的实例,则返回True

`getrepr(showlocals = False, style = 'long', abspath = False, tbfilter = True, funcargs = False, truncate_locals = True, chain = True)`: 返回str()表示此异常信息。

参数:

- **showlocals**(bool) - 显示每个回溯条目的本地人。忽略如果`style=="native"`。
- **style**(str) - long short no native traceback style
- **abspath**(bool) - 如果路径应更改为绝对路径或保持不变。
- **tbfilter**(bool) - 隐藏包含局部变量的条目`__tracebackhide__==True`。忽略如果`style=="native"`。
- **funcargs**(bool) - 为每个回溯条目显示Fixture方法(用于传统目的的"funcargs")。
- **truncate_locals**(bool) - 使用`showlocals==True`,确保本地可以安全地表示为字符串。
- **chain**(bool) - 如果显示Python 3中的链接异常。

*在版本3.9中更改: *添加了`chain`参数。

`match(regexp)`: 将正则表达式'`regexp`'与异常的字符串表示形式匹配。如果匹配则返回True(这样就可以写'`assert excinfo.match()`'). 如果不匹配则引发AssertionError。

FixtureDef

`class FixtureDef`: 基类object

工厂定义的容器。

FSCollector

`class FSCollector`: 基类`_pytest.nodes.Collector`

Function

`class Function`: 基类

`_pytest.python.FunctionMixin`, `_pytest.nodes.Item`, `_pytest.compat.FuncargnamesCompatAttr` 函数项负责设置和执行Python测试函数。 `originalname=None`: 原始函数名称, 没有任何装饰(例如参数`"[...]"`化为函数名称添加后缀)。 3.0版中的新函数。

`function`: 底层python'函数'对象

`runtest()`: 执行基础测试函数。

`setup()`: 执行此测试函数的设置。

Item

`class Item`: 基类`_pytest.nodes.Node` 一个基本的测试调用项。 请注意, 对于单个函数, 可能存在多个测试调用项。

`user_properties=None`: `user`属性是一个元组列表(名称, 值), 用于保存此测试的用户定义属性。

`add_report_section(when, key, content)`: 添加一个新的报表部分, 类似于内部完成添加stdout和stderr捕获的输出:

```
item.add_report_section("call", "stdout", "report section contents")
```

参数:

- **when**(STR) - 一个可能捕获的状态,"setup","call","teardown"。
- **key**(str) - 部分名称,可以随意定制。Pytest使用"stdout"和"stderr"内部。
- **content**(str) - 作为字符串的完整内容。

MarkDecorator

`class MarkDecorator(mark)`: 测试函数和测试类的装饰器。应用时,它将创建`MarkInfo`可以[通过钩子作为项目关键字检索的对象。MarkDecorator实例通常是这样创建的:

```
mark1 = pytest.mark.NAME # simple MarkDecorator
mark2 = pytest.mark.NAME(name1=value) # parametrized MarkDecorator
```

然后可以作为装饰器应用于测试函数:

```
@mark2
def test_function():
    pass
```

调用MarkDecorator实例时,它会执行以下操作:

1. 如果使用单个类作为其唯一的位置参数调用而没有其他关键字参数,则它会将自身附加到类,以便自动应用于该类中的所有测试用例。
2. 如果使用单个函数作为唯一的位置参数调用而没有其他关键字参数,则会将MarkInfo对象附加到函数,其中包含已存储在MarkDecorator内部的所有参数。
3. 当在任何其他情况下调用时,它执行“假构造”调用,即它返回一个新的MarkDecorator实例,其原始MarkDecorator的内容使用传递给此调用的参数进行更新。

注意: 上述规则阻止MarkDecorator对象仅存储单个函数或类引用作为其位置参数,而不包含其他关键字或位置参数。

name: mark.name的别名

args: mark.args的别名

kwargs: mark.kwargs的别名

with_args(** args*,*** kwargs*): 返回添加了额外参数的MarkDecorator

与调用不同,即使唯一参数是可调用/类,也可以使用它

返回: MarkDecorator

MarkGenerator

`class MarkGenerator`: `MarkDecorator`对象的工厂 - 作为`pytest.mark`单例实例公开。例如:

```
import pytest
@pytest.mark.slowtest
def test_function():
    pass
```

将在`MarkInfo`对象上设置一个“最慢”的`test_function`对象。

Mark

`class Mark (name: str,args,kwargs)`

`name=None`: 商标名称

`args=None`: 标记装饰器的位置参数

`kwargs=None`: 标记装饰器的关键字参数

`combined_with(other)`

参数:

- **other(mark)** - 与之结合的商标: 返回类型: `Mark`标记

通过附加`args`和合并映射来组合

Metafunc

`class Metafunc(definition,fixtureinfo,config,cls = None,module = None)`: `Metafunc`对象被传递给`pytest_generate_tests`钩子。它们有助于检查测试函数并根据测试配置或在定义测试函数的类或模块中指定的值生成测试。

`config=None`: 访问`_pytest.config.Config`测试Session的对象

`module=None`: 定义测试函数的模块对象。

`function=None`: 底层python测试函数

`fixturenames=None`: 测试函数所需的Fixture方法名称集

`cls=None`: 在或中定义测试函数的类对象`None`。

`parametrize(argnames,argvalues,indirect = False,ids = None,scope = None)`: 使用给定`argnames`的`argvalues`列表向基础测试函数添加新调用。在收集阶段执行参数化。如果你需要设置昂贵的资源,请参阅设置间接,以便在测试设置时进行。

参数:

- **argnames**- 以逗号分隔的字符串,表示一个或多个参数名称,或参数字符串的列表/元组。

- **argvalues-argvalues**列表确定使用不同参数值调用测试的频率。如果只指定了一个argname,则argvalues是值列表。如果指定了N个argnames,则argvalues必须是N元组的列表,其中每个tuple-element为其各自的argname指定一个值。
- **indirect**- argnames或boolean的列表。参数列表名称(argnames的子集)。如果为True,则列表包含argnames中的所有名称。对应于此列表中的argname的每个argvalue将作为request.param传递到其各自的argname fixture函数,以便它可以在测试的设置阶段而不是在收集时执行更昂贵的设置。
- **ids**- 字符串ID列表或可调用的列表。如果字符串,则每个字符串对应于argvalues,以便它们是测试ID的一部分。如果将None作为特定测试的id给出,则将使用该参数的自动生成的id。如果是可调用的,它应该采用一个参数(单个argvalue)并返回一个字符串或返回None。如果为None,将使用该参数的自动生成的id。如果没有提供id,它们将自动从argvalues生成。
- **范围**- 如果指定,则表示参数的范围。范围用于按参数实例对测试进行分组。它还将覆盖任何fixture函数定义的范围,允许使用测试上下文或配置设置动态范围。

Module

`class Module`: 基类 `_pytest.nodes.File`, `_pytest.python.PyCollector` 测试类和函数的收集器。

`collect()`: 返回此集合节点的子项(项和收集器)列表。

Node

`class Node`: Collector的基类和测试集合树的Item。收集器子类有子节点,Items是终端节点。

`name=None`: 父节点范围内的唯一名称

`parent=None`: 父收集器节点。

`config=None`: pytest配置对象

`session=None`: 此节点所属的Session

`fspath=None`: 收集此节点的文件系统路径(可以是None)

`keywords=None`: 从所有范围收集的关键字/标记

`own_markers=None`: 属于此节点的标记对象

`extra_keyword_matches=None`: 允许添加额外的关键字以用于匹配

`ihook`: 用于调用pytest钩子的fspath敏感钩子代理

`warn(warning)`: 发出此项目的警告。

除非明确禁止,否则将在测试Session后显示警告

参数:

- **警告(警告)** - 要发出的警告实例。必须是PytestWarning的子类。

抛出: **ValueError**- 如果warning实例不是PytestWarning的子类。

使用方法示例如:

```
node.warn(PytestWarning("some message"))
```

nodeid: a :: - 表示其集合树地址的分隔字符串。

listchain(): 从收集树的根开始返回所有父收集器的列表。

add_marker(marker, append = True): 动态地将标记对象添加到节点。

参数:

- **marker**(str或pytest.mark.*对象) -append=True是否附加标记,如果False插入位置0。

iter_markers(name=None)

参数:

- **name**- 如果给定,则按name属性过滤结果 迭代节点的所有标记

for ... in iter_markers_with_node(name = None)

参数:

- **name**- 如果给定,则按name属性过滤结果 迭代节点的所有标记返回元组序列(节点,标记)

get_closest_marker(name,default=None): 返回与名称匹配的标记,从最近(例如函数)到更远级别(例如模块级别)。

参数:

- **default**- 找不到标记的回退返回值
- **name**- 要过滤的名称

listextrakeywords(): 在self和任何父母中返回一组所有额外的关键字。

addfinalizer(fin): 在最终确定此节点时注册要调用的函数。

只有在此节点在设置链中处于活动状态时才能调用此方法,例如在self.setup()期间。

getparent(cls): 获取下一个父节点(包括ownelf),它是给定类的一个实例

Parser

class Parser: 解析命令行参数和ini文件值。

变量: **extra_info**- 泛型参数的字典 ->在处理命令行参数时出错的情况下显示的值。

getgroup(name,description =",after = None): 获取(或创建)命名选项组。

- **name**: 选项组的名称。
- **description**: -help输出的长描述。
- **after**: 其他组的名称,用于排序-help输出。 返回的组对象具有**adoption**与签名相同的方法,parser.adoption但将在输出中的相应组中显示。 pytest.--help

`addoption(** opts,** attrs*)`: 注册命令行选项。

- `opts`: 选项名称,可以是短期或长期期权。
- `attrs`: 相同的属性,其中`add_option()`所述的函数`argparse`库接受。

在命令行解析选项在`pytest`配置对象上可用之后`config.option.NAME`,`NAME`通常通过传递`dest`属性来设置,例如。`addoption("--long",dest="NAME",...)`

`parse_known_args(args,namespace = None)`: 此时解析并返回具有已知参数的命名空间对象。

`parse_known_and_unknown_args(args,namespace = None)`: 解析并返回具有已知参数的名称空间对象,此时其余参数未知。

`addini(name,help,type = None,default = None)`: 注册ini文件选项。

名称: ini-variable的名称

类型: 该变量的类型,可以是`pathlist`,`args`,`linelist`或`bool`。

默认: 默认值,如果不存在ini-file选项但是被查询。 可以通过调用来检索ini变量的值`config.getini(name)`。

PluginManager

`class PluginManager` Core `Pluginmanager`类,用于管理插件对象的注册和1: N钩子调用。 你可以通过致电注册新的挂钩`add_hookspecs(module_or_class)`。 你可以通过调用注册插件对象(包含挂钩)`register(plugin)`。`Pluginmanager`初始化为在注册的插件对象的dict名称中搜索的前缀。 出于调试目的,你可以调用`enable_tracing()`哪个随后将调试信息发送到跟踪帮助程序。

`register(plugin,name=None)`: 如果名称被阻止注册,请注册插件并返回其规范名称或无。 如果插件已注册,则引发`ValueError`。

`unregister(plugin = None,name = None)`: 从内部数据结构取消注册插件对象及其所有包含的钩子实现。

`set_blocked(name)`: 阻止给定名称的注册,如果已经注册,则注销。

`is_blocked(name)`: 如果名称被注册该名称的插件,则返回`True`。

`add_hookspecs(module_or_class)`: 添加给定`module_or_class`中定义的新钩子规范。 如果相应地修饰了函数,则会识别它们。

`get_plugins()`: 返回已注册的插件集。

`is_registered(plugin)`: 如果插件已经注册,则返回`True`。

`get_canonical_name(plugin)`: 返回插件对象的规范名称。 请注意,插件可以使用由`register(plugin,name)`的调用者指定的其他名称注册。 要获取已注册插件的名称,请使用`get_name(plugin)`。

`get_plugin(name)`: 返回给定名称的插件或`None`。

`has_plugin(name)`: 如果注册了具有给定名称的插件,则返回`True`。

`get_name(plugin)`: 注册插件的返回名称,如果未注册,则返回`None`。

`check_pending()`: 验证所有未针对钩子规范验证的钩子是可选的,否则引发`PluginValidationError`

`load_setuptools_entrypoints(group, name=None)`: 从查询指定的setuptools加载模块group。

参数:

- **group(str)** - 加载插件的入口点组
- **name(str)** - 如果给定,只加载给定的插件name。

返回类型: int

返回: 通过此调用返回加载的插件数。

`list_plugin_distinfo()`: 返回所有setuptools注册插件的distinguishedfo / plugin元组列表。

`list_name_plugin()`: 返回名称/插件对的列表。

`get_hookcallers(plugin)`: 获取指定插件的所有钩子调用者。

`add_hookcall_monitoring(before, after)`: 在所有挂钩的跟踪函数之前/之后添加并返回一个撤销函数,当被调用时,将删除添加的跟踪器。

`before(hook_name, hook_impls, kwargs)`将在所有挂钩调用之前调用并接收一个hookcaller实例,一个HookImpl实例列表和一个钩子调用的关键字参数。

`after(outcome, hook_name, hook_impls, kwargs)`接收相同的参数, `before`但也接收一个`_Result`表示整个钩子调用的结果的对象。

`enable_tracing()`: 启用对钩子调用的跟踪并返回撤销函数。

`subset_hook_caller(name, remove_plugins)`: 为命名方法返回一个新的_HookCaller实例,该方法管理除remove_plugins之外的所有已注册插件的调用。

PytestPluginManager

`class PytestPluginManager`: 基类pluggy.manager.PluginManager

覆盖pluggy.PluginManager以添加特定于pytest的函数:

- 从命令行加载插件,加载插件中的PYTEST_PLUGINSenv变量和pytest_plugins全局变量;
- `conftest.py`在启动期间装载; `addhooks(module_or_class)`

自2.8版以来已弃用。请`pluggy.PluginManager.add_hookspecs`改用。

`parse_hookimpl_opts(plugin, name)`

`parse_hookspec_opts(module_or_class, name)`

`register(plugin, name=None)`: 如果名称被阻止注册,请注册插件并返回其规范名称或无。如果插件已注册,则引发ValueError。

`getplugin(name)`

`hasplugin(name)` 如果注册了具有给定名称的插件,则返回True。

`pytest_configure(config)`

`consider_preparse(args)`

`consider_pluginarg(arg)`

`consider_conftest(conftestmodule)`

`consider_env()`

`consider_module(mod)`

`import_plugin(modname, consideration_entry_points = False)`: 用插件导入插件`modname`。如果`consider_entry_points`为`True`,则还会将入口点名称视为查找插件。

Session

`class Session`: 基类`_pytest.nodes.FSCollector` *exception* `Interrupted`: 基类`KeyboardInterrupt` 发出中断的测试运行信号。

exception `Failed`: 基类`Exception` 在测试运行失败时发出停止信号。

`for ... in collect()`: 返回此集合节点的子项(项和收集器)列表。

TestReport

`class TestReport`: 基本测试报告对象(如果失败,也用于设置和拆除调用)。

`nodeid=None`: 规范化的集合节点ID

`location=None`: `a(filesystempath,lineno,domaininfo)`元组指示测试项的实际位置 - 它可能与收集的测试项不同,例如,如果方法是从不同的模块继承的。

`keywords=None`: `name -> value`字典,包含与测试调用关联的所有关键字和标记。

`outcome=None`: 测试结果,总是“通过”,“失败”,“跳过”之一。

`longrepr=None`: 无或失败表示。

`when=None`: 'setup','call','teardown'之一表示测试阶段。

`user_properties=None`: `user`属性是一个元组列表(名称,值),用于保存用户定义的测试属性

`sections=None`: 需要编组的额外信息对的列表。通过`pytest`用于添加从捕获的文本和,但是可以通过其它的插件可用于任意信息添加到报表。`(str,str)``stdout``stderr`

`duration=None`: 只运行测试所花费的时间

classmethod `from_item_and_call(item,call)`使用标准项和调用信息创建和填充`TestReport`的工厂方法。

`caplog`: 如果启用了日志捕获,则返回捕获的日志行 版本3.5中的新函数。

`capstderr`: 如果启用了捕获,则从`stderr`返回捕获的文本 3.0版中的新函数。

`capstdout`: 如果启用了捕获,则从`stdout`返回捕获的文本 3.0版中的新函数。

`count_towards_summary`: 如果此报告应计入测试Session结束时显示的总计,则返回True: “1通过,1失败等”。 *试验性功能*

注意: 此函数被认为是实验性的,因此请注意即使在修补程序版本中它也会发生变化。

`head_line`: 返回此报告的longrepr输出显示的头行,更常见的是在故障期间的回溯表示期间: *试验性功能*

```
_____ Test.foo _____
```

在上面的示例中,head_line是“Test.foo”。

注意: 此函数被认为是实验性的,因此请注意即使在修补程序版本中它也会发生变化。

`longreprtext` 只读属性,返回完整的字符串表示形式longrepr。 *3.0版中的新函数。*

_Result

`class _Result(result, excinfo)` result: 获取此挂钩调用的结果(DEPRECATED支持`get_result()`)。

`force_result(result)`: 强制结果result。

如果挂钩被标记为`firstresult`单个值,则应设置否则设置(修改)结果列表。调用期间发现的任何异常都将被删除。

`get_result()`: 获取此挂钩调用的结果。

如果钩子被标记为`firstresult`只有一个值,则返回结果列表。

特殊变量

pytest在测试模块中定义时以特殊方式处理一些全局变量。

collect_ignore

参考: 自定义测试集合

可以在`conftest.py`文件中声明以排除测试目录或模块。需要成为`list[str]`。

```
collect_ignore = ["setup.py"]
```

collect_ignore_glob

参考: 自定义测试集合

可以在`conftest.py`文件中声明,以使用Unix shell样式的通配符排除测试目录或模块。需要在`list[str]`哪里str可以包含glob模式。

```
collect_ignore_glob = ["*_ignore.py"]
```

pytest_plugins

参考: 在测试模块或conftest文件中要求/加载插件

可以在*测试模块*和*conftest.py*文件中在全局级别声明以注册其他插件。可以是一个或。`str``Sequence[str]`

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

```
pytest_plugins = ("myapp.testsupport.tools", "myapp.testsupport.regression")
```

pytest_mark

参考: 标记整个类或模块

可以在*测试模块*的全局级别声明,以将一个或多个[标记

```
import pytest

pytestmark = pytest.mark.webtest
```

```
import pytest

pytestmark = [pytest.mark.integration, pytest.mark.slow]
```

PYTEST_DONT_REWRITE(模块文档字符串)

`PYTEST_DONT_REWRITE`可以将文本添加到任何模块*docstring*以禁用该模块的[断言重写。

环境变量

可用于更改pytest行为的环境变量。

PYTEST_ADDOPTS

它包含一个命令行(由py: mod: shlex模块解析),该命令行将添加到用户给出的命令行之前,有关详细信息,请参阅[如何更改命令行选项默认值。

PYTEST_DEBUG

设置后,pytest将打印跟踪和调试信息。

PYTEST_PLUGINS

包含应作为插件加载的以逗号分隔的模块列表:

```
export PYTEST_PLUGINS=mymodule.plugin,xdist
```

PYTEST_DISABLE_PLUGIN_AUTOLOAD

设置后,通过`setuptools`入口点禁用插件自动加载。只会加载明确指定的插件。

PYTEST_CURRENT_TEST

这并不是由用户设置,而是由`pytest`在内部设置当前测试的名称,以便其他进程可以检查它,有关详细信息,请参阅 `[PYTEST_CURRENT_TEST]`环境变量。

配置选项

这里是一个可以在被写入内置的配置选项的列表`pytest.ini`,`tox.ini`或`setup.cfg`通常位于版本库的根文件。所有选项必须在一个`[pytest]`部分下(`[tool:pytest]`对于`setup.cfg`文件)。警告 的使用`setup.cfg`是不推荐,除非非常简单的用例。`.cfg`文件使用不同的解析器`pytest.ini`,`tox.ini`这可能导致难以追踪问题。如果可能,建议使用后面的文件来保存`pytest`配置。配置文件选项可以通过使用在命令行中覆盖,`-o/--override`也可以多次传递。预期的格式是`name=value`。例如:

```
pytest -o console_output_style=classic -o cache_dir=/tmp/mycache
```

addopts

将指定`OPTS`的命令行参数添加到命令行参数集中,就像它们已由用户指定一样。示例如:如果你有此`ini`文件内容:

```
# content of pytest.ini
[pytest]
addopts = --maxfail=2 -rf # exit after 2 failures,report fail info
```

发行实际意味着: `pytesttest_hello.py`

```
pytest --maxfail=2 -rf test_hello.py
```

默认是不添加选项。

cache_dir

设置存储缓存插件内容的目录。默认目录是`.pytest_cache`在`[rootdir]`中。

confcutdir

设置向上搜索`conftest.py`文件的目录。默认情况下,pytest将停止`conftest.py`从项目的`pytest.ini/tox.ini/`向上搜索文件(`setup.cfg`如果有),或者直到文件系统根目录。

console_output_style

运行测试时设置控制台输出样式:

- **classic**: 经典的pytest输出。
- **progress**: 喜欢经典的pytest输出,但带有进度指示器。
- **count**: 像进度一样,但随着测试完成次数而不是百分比显示进度。默认值为**progress**,但**classic**如果你愿意,或者新模式导致意外问题,你可以回退到:

```
# content of pytest.ini
[pytest]
console_output_style = classic
```

doctest_encoding

用于解码带有文档字符串的文本文件的默认编码。[看看pytest如何处理doctests。]

doctest_optionflags

标准**doctest**模块中的一个或多个doctest标志名称。[看看pytest如何处理doctests。]

empty_parameter_set_mark

允许在参数化中为空参数选择操作

- **skip**使用空参数跳过测试(默认)
- **xfail**使用空参数标记测试为xfail(`run = False`)
- **fail_at_collect**如果**parametrize**收集空参数集,则引发异常

```
# content of pytest.ini
[pytest]
empty_parameter_set_mark = xfail
```

注意: 计划**xfail**在将来的版本中更改此选项的默认值,因为这被认为不易出错,有关详细信息,请参阅 #3155。

filterwarnings

设置应为匹配的警告采取的过滤器和操作的列表。默认情况下,测试会话期间发出的所有警告都将在测试会话结束时显示在摘要中。

```
# content of pytest.ini
[pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
```

这告诉pytest忽略弃用警告并将所有其他警告变为错误。有关更多信息,请参阅[警告捕获](#)。

junit_family

版本4.2中的新函数。配置生成的JUnit XML文件的格式。可能的选择是:

- **xunit1(或legacy)**: 生成旧样式输出,与xunit 1.0格式兼容。这是默认值。
- **xunit2**: 生成[xunit 2.0样式输出,哪个应该与最新的Jenkins版本更兼容。

```
[pytest]
junit_family = xunit2
```

junit_suite_name

要设置根测试套件xml项的名称,可以junit_suite_name在配置文件中配置该选项:

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

log_cli_date_format

设置一个time.strftime()兼容的字符串,该字符串将在格式化实时日志记录的日期时使用。

```
[pytest]
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

有关更多信息,请参阅[实时日志](#)。

log_cli_format

设置logging用于格式化实时日志记录消息的兼容字符串。

```
[pytest]
log_cli_format = %(asctime)s %(levelname)s %(message)s
```

有关更多信息,请参阅[\[实时日志\]](#)。

log_cli_level

设置应为实时日志记录捕获的最小日志消息级别。可以使用整数值或级别的名称。

```
[pytest]
log_cli_level = INFO
```

有关更多信息,请参阅[\[实时日志\]](#)。

log_date_format

设置`time.strftime()`与日志记录捕获格式化日期时将使用的兼容字符串。

```
[pytest]
log_date_format = %Y-%m-%d %H:%M:%S
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_file

`pytest.ini`除了活动的其他日志记录工具之外,还应设置相对于应写入日志消息的文件的文件名。

```
[pytest]
log_file = logs/pytest-logs.txt
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_file_date_format

设置`time.strftime()`在格式化日志文件的日期时将使用的兼容字符串。

```
[pytest]
log_file_date_format = %Y-%m-%d %H:%M:%S
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_file_format

设置一个`logging`兼容的字符串,用于格式化重定向到日志文件的日志消息。

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_file_level

设置应为日志记录文件捕获的最小日志消息级别。可以使用整数值或级别的名称。

```
[pytest]
log_file_level = INFO
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_format

设置`logging`用于格式化捕获的日志消息的兼容字符串。

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_level

设置应记录捕获的最小日志消息级别。可以使用整数值或级别的名称。

```
[pytest]
log_level = INFO
```

有关更多信息,请参阅[\[日志记录\]](#)。

log_print

如果设置为`False`,将禁用显示失败测试的捕获日志消息。

```
[pytest]
log_print = False
```

有关更多信息,请参阅[\[日志记录\]](#)。

markers

使用`--strict`命令行参数时,只允许使用已知的标记(由代码核心`pytest`或某些插件定义)。你可以在此设置中列出其他标记,以将其添加到白名单。 你可以列出每行一个标记名称,从选项名称缩进。

```
[pytest]
markers =
    slow
    serial
```

minversion

指定运行测试所需的最小`pytest`版本。

```
# content of pytest.ini
[pytest]
minversion = 3.0 # will fail if we run with pytest-2.8
```

norecursedirs

设置目录`basename`模式以避免在递归测试发现时使用。各个(`fnmatch`样式)模式应用于目录的基本名称,以决定是否递归到目录。模式匹配字符:

```
-    matches everything
?    matches any single character
[seq] matches any character in seq
[!seq] matches any char not in seq
```

默认模式是。设置替换默认值。以下是如何避免某些目录的示例

如: `.*, 'build', 'dist', 'CVS', '_darcs', '{arch}', '*.egg', 'venv'``norecursedirs`

```
[pytest]
norecursedirs = .svn _build tmp*
```

这将告诉`pytest`我们不要查看典型的`subversion`或`sphinx-build`目录或任何`tmp`前缀目录。此外,`pytest`将尝试通过激活脚本的存在智能地识别和忽略`virtualenv`。除非`--collect-in-virtualenv`给出,否则在测试收集期间不会考虑任何被视为虚拟环境根目录的目录。另请注意,`norecursedirs`优先于`--collect-in-virtualenv`;例如,如果你打算在`virtualenv`中使用匹配的基本目录运行测试,则除了使用该标志外, `.*`还必须覆盖。

`norecursedirs``--collect-in-virtualenv`

python_classes

一个或多个名称前缀或glob样式模式,用于确定考虑用于测试集合的类。通过在模式之间添加空格来搜索多个glob模式。默认情况下,pytest会将任何以前缀Test为前缀的类视为测试集合。以下是如何从以下结尾的类中收集测试的示例Suite:

```
[pytest]
python_classes = *Suite
```

请注意,unittest.TestCase无论此选项如何,始终都会收集派生类,因为unittest自己的集合框架用于收集这些测试。

python_files

一个或多个Glob样式的文件模式,用于确定哪些python文件被视为测试模块。通过在模式之间添加空格来搜索多个glob模式:

```
[pytest]
python_files = test_*.py check_*.py example_*.py
```

或者每行一个:

```
[pytest]
python_files =
test_*.py
check_*.py
example_*.py
```

默认情况下,匹配的文件test_*.py和*_test.py将被视为测试模块。

python_functions

一个或多个名称前缀或glob-patterns,用于确定哪些测试函数和方法被视为测试。通过在模式之间添加空格来搜索多个glob模式。默认情况下,pytest会将任何前缀test为函数的函数视为测试。以下是如何收集以下结尾的测试函数和方法的示例_test:

```
[pytest]
python_functions = *_test
```

请注意,这对生成在派生类上的方法没有影响,因为自己的集合框架用于收集这些测试。

unittest.TestCase` unittest 有关更多详细示例,请参阅[更改命名约定]。

testpaths

当从`[rootdir]`目录执行`pytest`时,如果在命令行中没有给出特定的目录,文件或测试ID,则设置应搜索测试的目录列表。当所有项目测试都在一个已知位置以加速测试收集并避免意外接收不需要的测试时非常有用。

```
[pytest]
testpaths = testing doc
```

这告诉`pytest`只在从根目录执行时查找`testing`和`doc`目录中的测试。

usefixtures

将应用于所有测试函数的Fixture方法列表;这在语义上与将`@pytest.mark.usefixtures`标记应用于所有测试函数相同。

```
[pytest]
usefixtures =
    clean_db
```

xfail_strict

如果设置为`True`,则标记为`@pytest.mark.xfail`实际成功的测试将默认为测试套件失败。有关更多信息,请参阅`[strict]`参数。

```
[pytest]
xfail_strict = True
```

优质集成实践

使用pip安装包

对于开发,我们建议你使用`[venv]`来安装应用程序和任何依赖项,以及`pytest`包本身。这可确保你的代码和依赖项与系统Python安装隔离。

接下来,使用`setup.py`使用以下最低内容将文件放在包的根目录中:

```
from setuptools import setup, find_packages

setup(name="PACKAGENAME", packages=find_packages())
```

`PACKAGENAME`包裹的名称在哪里。然后,你可以通过从同一目录运行,以“可编辑”模式安装程序包:

```
pip install -e .
```

它允许你更改源代码(测试和应用程序)并随意重新运行测试。这与运行类似,或者使用符号链接将你的包安装到开发代码中。`pythonsetup.pydevelop`condadevelop`

Python测试发现的约定

`Pytest`实现以下标准测试发现:

- 如果未指定参数,则从`testpaths`(如果已配置)或当前目录开始收集。或者,命令行参数可以用于目录,文件名或节点ID的任意组合。
- 递归到目录,除非它们匹配`norecursedirs`。
- 在这些目录,搜索`test_*.py`或`*_test.py`文件,由他们进口[的测试包名。
- 从这些文件中收集测试项目:
 - `test`在类之外的前缀测试函数或方法
 - `test`前缀测试`Test`类中的前缀测试函数或方法(没有`__init__`方法)

有关如何自定义测试发现的示例[更改标准(Python)测试发现。

在Python模块中,`pytest`还使用标准的[`unittest.TestCase`子类化技术发现测试。

选择测试布局结构/导入规则

`pytest`支持两种常见的测试布局:

在应用程序代码外测试

如果你有许多函数测试,或者出于其他原因希望将测试与实际应用程序代码分开(通常是个好主意),那么将测试放入实际应用程序代码之外的额外目录可能会很有用:

```
setup.py
mypkg/
  __init__.py
  app.py
  view.py
tests/
  test_app.py
  test_view.py
  ...
```

这有以下好处:

- 执行后,你的测试可以针对已安装的版本运行。`pipinstall.`
- 执行后,你可以使用可编辑安装对本地副本运行测试。`pipinstall--editable.`
- 如果你没有`setup.py`文件并且依赖于默认情况下Python将当前目录放入`sys.path`以导入你的包,则可以执行直接对本地副本执行测试,而不使用。`python-mpytest`pip`

注意: 有关调用和调用之间差异的更多信息,请参阅[`pytest`导入机制和`sys.path` / `PYTHONPATH`。

`pytest`python-mpytest`

请注意,使用此方案时,你的测试文件必须具有唯一的名称,因为`pytest`将它们作为顶级模块导入,因为没有包来从中获取完整的包名称。换句话说,在上面的示例中的试验文件将被导入为`test_app`和`test_view`通过加入顶层模

块`tests/`到`sys.path`。

如果需要具有相同名称的测试模块,可以将`__init__.py`文件添加到`tests`文件夹和子文件夹,并将其更改为包:

```
setup.py
mypkg/
...
tests/
  __init__.py
  foo/
    __init__.py
    test_view.py
  bar/
    __init__.py
    test_view.py
```

现在Pytest将加载模块`tests.foo.test_view`并`tests.bar.test_view`允许你使用相同名称的模块。但是现在这引入了一个微妙的问题: 为了从`tests`目录中加载测试模块,pytest将存储库的根目录`sys.path`添加到,这增加了现在`mypkg`也可导入的副作用。如果你使用像`tox`这样的工具在虚拟环境中测试程序包,则会出现问题,因为你要测试程序包的已安装版本,而不是存储库中的本地代码。

在这种情况下,强烈建议使用`src`应用程序根包位于根目录的子目录中的布局:

```
setup.py
src/
  mypkg/
    __init__.py
    app.py
    view.py
  tests/
    __init__.py
    foo/
      __init__.py
      test_view.py
    bar/
      __init__.py
      test_view.py
```

这种布局可以防止许多常见的陷阱,并且有很多好处,这在[lonelcristianmaries的有更好的解释。

测试作为应用程序代码的一部分

如果测试和应用程序模块之间存在直接关系并希望将它们与应用程序一起分发,则将测试目录内联到应用程序包中非常有用:

```
setup.py
mypkg/
  __init__.py
```

```
app.py
view.py
test/
    __init__.py
    test_app.py
    test_view.py
    ...
```

在此方案中,使用以下`--pyargs`选项可以轻松运行测试:

```
pytest --pyargs mypkg
```

`pytest`将发现`mypkg`安装位置并从那里收集测试。

请注意,此布局也与`src`上一节中提到的布局一起使用。

注意: 你可以为你的应用程序使用Python3命名空间包(PEP420),但`pytest`仍将根据文件的存在执行[测试包名称发现`__init__.py`。如果你使用上面两个推荐的文件系统布局中的一个,但是`__init__.py`从你的目录中删除它们,那么它应该适用于Python3.3及更高版本。但是,从“内联测试”开始,你将需要使用绝对导入来获取应用程序代码。

注意: 如果`pytest`在递归到文件系统时找到“a / b / test_module.py”测试文件,它将确定导入名称,如下所示:

- 确定`basedir`: 这是第一个“向上”(朝向根)目录,不包含`__init__.py`。如果如两者a并b包含一个`__init__.py`文件,然后父目录a将成为`basedir`。
- 执行以使测试模块可以在完全限定的导入名称下导入。`sys.path.insert(0,basedir)`
- `importa.b.test_module`其中路径是通过将路径分隔符/转换为“.”字符来确定的。这意味着你必须遵循将目录和文件名直接映射到导入名称的约定。

这种有点进化的导入技术的原因在于,在较大的项目中,多个测试模块可能相互导入,因此导出规范的导入名称有助于避免出现意外情况,例如测试模块导入两次。

tox

一旦完成了你的工作并希望确保你的实际软件包通过所有测试,你可能需要查看[tox](#),[virtualenv](#)测试自动化工具及其[pytest支持。`tox`帮助你使用预定义的依赖项设置`virtualenv`环境,然后使用选项执行预配置的测试命令。它将针对已安装的软件包运行测试,而不是针对源代码检查,从而有助于检测包装故障。

与`setuptools`集成 你可以使用[`pytest-runner`插件将测试运行集成到基于`setuptools`的项目中。

将此添加到`setup.py`文件:

```
from setuptools import setup

setup(
    # ...,
    setup_requires=["pytest-runner",...],
```

```
tests_require=["pytest",...],
# ...,
)
```

并在`setup.cfg`文件中创建一个别名:

```
[aliases]
test=pytest
```

如果你现在输入:

```
python setup.py test
```

这将使用执行你的测试`pytest-runner`。因为这是一个独立版本,`pytest`无需事先安装,无论如何都需要调用`test`命令。你还可以使用其他参数传递给`pytest`,例如测试目录或其他选项`--addopts`。

你还可以`setup.cfg`通过将文件放入以下`[tool:pytest]`部分来指定文件中的其他`pytest-ini`选项:

```
[tool:pytest]
addopts = --verbose
python_files = testing/*/*.py
```

手动整合

如果由于某种原因你不想/不能使用`pytest-runner`,你可以编写自己的`setuptools`测试命令来调用`pytest`。

```
import sys

from setuptools.command.test import test as TestCommand

class PyTest(TestCommand):
    user_options = [("pytest-args=", "a", "Arguments to pass to pytest")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = ""

    def run_tests(self):
        import shlex

        # import here, cause outside the eggs aren't loaded
        import pytest

        errno = pytest.main(shlex.split(self.pytest_args))
        sys.exit(errno)
```



```
setup(  
    # ...,  
    tests_require=["pytest"],  
    cmdclass={"pytest": PyTest},  
)
```

现在,如果你运行:

```
python setup.py test
```

这将`pytest`在需要时下载,然后按照你的预期运行测试。你可以使用`--pytest-args`或`-a`命令行选项传递单个参数字符串。例如:

```
python setup.py test -a "--durations=5"
```

相当于运行 `pytest--durations=5`

片状测试是表现出间歇性或偶发性失败的测试,似乎具有非确定性行为。有时它会通过,有时会失败,而且不清楚为什么。本页讨论了可以提供帮助的`pytest`函数以及识别,修复或减轻它们的其他一般策略。

片状测试

为什么片状测试是个问题

当使用连续集成(CI)服务器时,片状测试尤其麻烦,因此在合并新代码更改之前必须通过所有测试。如果测试结果不是一个可靠的信号 - 测试失败意味着代码更改破坏了测试 - 开发人员可能会对测试结果产生不信任,这可能导致忽略真正的失败。它也是浪费时间的一个来源,因为开发人员必须重新运行测试套件并调查虚假故障。

潜在的根本原因

系统状态

从广义上讲,一个片状测试表明测试依赖于一些未被适当控制的系统状态 - 测试环境没有充分隔离。更高级别的测试更有可能是因为他们依赖更多的状态。

当测试套件并行运行时(例如使用`pytest-xdist`),有时会出现片状测试。这可以表明测试依赖于测试排序。

- 也许不同的测试是在自身之后无法清理并留下导致片状测试失败的数据。
- 片状测试依赖于先前测试的数据,该测试不会自行清理,并且并行运行以前的测试并不总是存在
- 修改全局状态的测试通常不能并行运行。

过于严格的断言

过于严格的断言可能会导致浮点比较以及时序问题。`[pytest.approx`在这里很有用。

Pytest特性

Xfail严格模式

`pytest.mark.xfailwithstrict=False`可用于标记测试,以便其失败不会导致整个构建中断。这可以被视为手动隔离,永久使用是相当危险的。

PYTEST_CURRENT_TEST

`PYTEST_CURRENT_TEST`环境变量可用于确定“哪个测试卡住了”。

插件

重新运行任何失败的测试可以通过给予他们额外的机会来减轻片状测试的负面影响,这样整体构建就不会失败。几个pytest插件支持这个:

- 片状测试
- `pytest-flakefinder`
- `pytest-rerunfailures`
- `pytest-replay`: 这个插件有助于重现CI运行期间观察到的局部崩溃或片状测试。

故意随机化测试的插件可以帮助公开测试状态问题:

- `pytest随机顺序`
- `pytest随机`

其他一般策略

拆分测试套件

将单个测试套件拆分为两个是常见的,例如单元与集成,并且仅将单元测试套件用作CI门。这也有助于保持构建时间的可管理性,因为高级别测试往往更慢。但是,这意味着打破构建的代码可能会合并,因此需要额外的警惕来监视集成测试结果。

失败的视频/截图

对于UI测试,这些对于了解测试失败时UI的状态非常重要。`pytest-splinter`可以与`pytest-bdd`这样的插件一起使用,并且可以在测试失败时保存屏幕截图,这有助于隔离原因。

删除或重写测试

如果其他测试涵盖了该函数,则可能会删除该测试。如果没有,也许它可以在较低的水平重写,这将消除片状或使其来源更明显。

隔离

Mark Lapierre在2018年的一篇文章中讨论了[隔离测试的优缺点]。

在失败时重新运行的CI工具

Azure管道(Azure云CI / CD工具,以前称为Visual Studio Team Services或VSTS)具有[识别片状测试和重新运行失败测试的函数]。

研究

这是一个有限的列表,请提交问题或拉取请求以扩展它!

- Gao,Zebao,Yalan Liang,Myra B. Cohen,Atif M. Memon和Zhen Wang。“使系统用户交互式测试可重复:何时以及我们应该控制什么?”在*软件工程(ICSE),2015 IEEE / ACM第37届IEEE国际会议上*,第一卷。1,pp.55-65。IEEE,2015年[PDF](#)
- Palomba,Fabio和Andy Zaidman。“测试气味的重构是否会导致固定片状测试?”在*软件维护和演进(ICSME),2017 IEEE国际会议上*,第1-12页。IEEE,2017.[Google\[Drive中的PDF](#)
- Bell,Jonathan,Owolabi Legunsen,Michael Hilton,Lamyaa Eloussi,Tifany Yung和Darko Marinov。“DeFlaker: 自动检测片状测试。”在*2018年国际软件工程会议论文集*中。2018.[\[PDF](#)

资源

- 在
- 破碎的建筑: 在持续集成测试中建立信任测试在2017年奥斯汀SeleniumConf上讲话(视频)
- 测试和代码播客: 片状测试以及如何处理它们由Brian Okken和Anthony Shaw,2018年
- 微软:
 - 我们如何通过测试VSTS来实现
 - 博客和谈话(视频)
- 谷歌:
 - 谷歌的片状测试和我们如何减轻他们的影响,约翰米科,2016年
 - 谷歌的片状测试来自哪里? 作者: Jeff Listfield,2017

以下是pytest可能需要更改[sys.path](#)以导入测试模块或[conftest.py](#)文件的方案列表。

Pytest导入机制和sys.path/PYTHONPATH

包中的测试模块及conftest.py文件

考虑这个文件和目录布局:

```
root/  
|- foo/  
  - __init__.py  
  - conftest.py  
  - bar/  
    - __init__.py  
    - tests/  
      - __init__.py  
      - test_foo.py
```

执行时:

```
pytest root/
```

pytest会发现`foo/bar/tests/test_foo.py`并意识到它是一个包的一部分,因为`__init__.py`在同一个文件夹中有一个文件。然后它将向上搜索,直到它找到仍包含`__init__.py`文件的最后一个文件夹,以便找到包根(在本例中`foo/`)。要加载模块,它将插入`root/`到前面`sys.path`(如果不存在),以便`test_foo.py`作为*模块*加载`foo.bar.tests.test_foo``。

相同的逻辑适用于该`conftest.py`文件: 它将作为`foo.conftest`模块导入。

当测试存在于包中以避免出现问题并允许测试模块具有重复的名称时,保留完整的包名称非常重要。在[Python测试发现的约定中也详细讨论了这一点。

独立测试模块及`conftest.py`文件

考虑这个文件和目录布局:

```
root/  
|- foo/  
   - conftest.py  
   - bar/  
      - tests/  
         - test_foo.py
```

执行时:

```
pytest root/
```

pytest会发现`foo/bar/tests/test_foo.py`并意识到它不是包的一部分,因为`__init__.py`同一个文件夹中没有文件。然后它将添加`root/foo/bar/tests`到`sys.path`以`test_foo.py`作为模块导入`test_foo`。`conftest.py`通过添加`root/foo`以`sys.path`将其导入为文件,对文件执行相同操作`conftest`。

因此,此布局不能包含具有相同名称的测试模块,因为它们都将导入全局导入命名空间。

在[Python测试发现的约定中也详细讨论了这一点。

调用通过`python -m pytest`调用pytest

运行pytest而不是产生几乎相同的行为,除了前一个调用将添加当前目录。另请参阅[通过`python -m pytest`调用pytest。`python-mpytest[...]\`pytest[...]\`sys.path`

以下是pytest可能需要更改`sys.path`以导入测试模块或`conftest.py`文件的方案列表。

运行pytest而不是产生几乎相同的行为,除了前一个调用将添加当前目录。另请参阅[通过`python -m pytest`调用pytest。`python-mpytest[...]\`pytest[...]\`sys.path`

示例和自定义技巧

这是一个(不断增长的)示例列表。如果你需要更多示例或有疑问,请联系我们。另请参阅包含许多示例代码段的综合文档。此外,stackoverflow.com上的pytest专栏通常会有示例解答。

基本示例参考:

- 安装和入门: 基础入门示例
- 断言及断言语句: 基础断言示例
- Pytest Fixture: 显式,模块化,扩展: 基本fixture/setup示例
- 参数化Fixture和测试用例:基本测试用例的参数化
- unittest.TestCase支持: 基本unittest集成示例
- 运行Nose用例: 基本Nosetests集成示例

以下示例针对你可能遇到的各种用例。

- Pytest失败用例报告示例
- 基本使用方式及示例
 - 根据命令行选项将不同的值传递给测试函数
 - 动态添加命令行选项
 - 根据命令行选项控制跳过测试
 - 编写完善的集成断言助手
 - 检测是否在pytest运行中运行
 - 添加信息以测试报告标题
 - 分析测试持续时间
 - 增量测试 - 测试步骤
 - 包/目录级固定Fixture(设置)
 - 后处理测试报告/失败
 - 在Fixture方法中提供测试结果信息
 - PYTEST_CURRENT_TEST环境变量
 - 冻结pytest
- 参数化测试
 - 根据命令行生成参数组合
 - 测试ID的不同选项
 - 快速移植“testscenarios”
 - 推迟参数化资源的设置
 - 在特定参数上应用间接
 - 通过每类配置参数化测试用例
 - 具有多个Fixture方法的间接参数化
 - 可选实现/导入的间接参数化
 - 设置单个参数化测试的标记或测试ID
 - 参数化条件提升
- 使用自定义标记
 - 标记测试函数并为运行选择它们
 - 根据节点ID选择测试
 - 使用基于其名称来选择测试-k expr
 - 注册标记
 - 标记整个类或模块
 - 使用参数化时标记单个测试
 - 用于控制测试运行的自定义标记和命令行选项
 - 将可调用的标记传递给自定义标记
 - 阅读从多个地方设置的标记

- 使用pytest标记平台特定测试
 - 根据测试名称自动添加标记
- 一个会话Fixture方法,可以查看所有收集的测试
- 更改标准(Python)测试发现
 - 在测试收集期间忽略路径
 - 在测试收集期间取消选择测试
 - 保持从命令行指定的重复路径
 - 更改目录递归
 - 更改命名约定
 - 将cmdline参数解释为Python包
 - 找出收集的内容
 - 自定义测试集合
- 使用非Python脚本测试用例
 - 在Yaml文件中指定测试的基本示例

Bash自动补全设置

在Linux/Mac bash shell环境下,可以使用argcomplete对pytest命令进行自动补全。首先要安装和启用argcomplete。

使用以下命令安装argcomplete:

```
sudo pip install 'argcomplete>=0.5.7'
```

全局激活argcomplete命令补全,对所有支持的Python包生效,可以执行:

```
sudo activate-global-python-argcomplete
```

仅对于pytest永久启用命令补全,可以执行:

```
register-python-argcomplete pytest >> ~/.bashrc
```

仅对pytest一次性启用命令补全,可以执行:

```
eval "$(register-python-argcomplete pytest)"
```