2.8 Запуск отладчика PDB (Python Debugger) при падении тестов

`python` содержит встроенный отладчик PDB (Python Debugger). `pytest` позволяет запустить отладчик с помощью параметра командной строки:

```
pytest --pdb
```

Использование параметра позволяет запускать отладчик при каждом падении теста (или прерывании его с клавиатуры). Часто хочется сделать это для первого же упавшего теста, чтобы понять причину его падения:

```
pytest -x --pdb   # вызывает отладчик при первом падении и завершает тестовую сессию
pytest --pdb --maxfail=3  # вызывает отладчик для первых трех падений
```

Обратите внимание, что при любом падении информация об исключении сохраняется в `sys.last_value`, `sys.last_type` и `sys.last_traceback`. При интерактивном использовании это позволяет перейти к отладке после падения с помощью любого инструмента отладки. Можно также вручную получить доступ к информации об исключениях, например:

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

2.9 Запуск отладчика PDB (Python Debugger) в начале теста

`pytest` позволяет запустить отладчик сразу же при старте каждого теста. Для этого нужно передать следующий параметр:

```
pytest --trace
```

В этом случае отладчик будет вызываться при запуске каждого теста.

2.10 Установка точек останова

Чтобы установить точку останова, вызовите в коде `import pdb;pdb.set_trace()`, и `pytest` автоматически отключит перехват вывода для этого теста, при этом:

- на перехват вывода в других тестах это не повлияет;
- весь перехваченный ранее вывод будет обработан как есть;
- перехват вывода возобновится после завершения отладочной сессии (с помощью команды `continue`).

2.11 Использование встроенной функции breakpoint

`Python 3.7` содержит встроенную функцию `breakpoint()`. `pytest` поддерживает использование `breakpoint()` следующим образом:

- если вызывается `breakpoint()`, и при этом переменная `PYTHONBREAKPOINT` установлена в значение по умолчанию, `pytest` использует расширяемый отладчик PDB вместо системного;
- когда тестирование будет завершено, система снова будет использовать отладчик `Pdb` по умолчанию;
- если `pytest` вызывается с опцией `--pdb` то расширяемый отладчик PDB используется как для функции `breakpoint()`, так и для упавших тестов/необработанных исключений;
- для определения пользовательского класса отладчика можно использовать `--pdbcls`.

2.12 Профилирование продолжительности выполнения теста

Чтобы получить список 10 самых медленных тестов, выполните:

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

По умолчанию, `pytest` не покажет тесты со слишком маленькой (менее одной сотой секунды) длительностью выполнения, если в командной строке не будет передан параметр `-vv`.

2.13 Модуль `faulthandler`

Стандартный модуль faulthandler можно использовать для сброса трассировок `Python` при ошибке или по истечении времени ожидания.

При запуске `pytest` модуль автоматически подключается, если только в командной строке не используется опция `-p no:faulthandler`.

Кроме того, для сброса трассировок всех потоков в случае, когда тест длится более `X` секунд, можно использовать опцию faulthandler_timeout=X (для Windows неприменима).

Примечание: Эта функциональность была интегрирована из внешнего плагина pytest-faulthandler с двумя небольшими изменениями:

- чтобы ее отключить, используйте `-p no:faulthandler` вместо `--no-faulthandler`;
- опция командной строки `--faulthandler-timeout` превратилась в конфигурационную опцию faulthandler_timeout. Ее по-прежнему можно настроить из команндной строки, используя `-o faulthandler_timeout=X`.

2.14 Создание файлов формата JUnit

Чтобы создать результирующие файлы в формате, понятном Jenkins или другому серверу непрерывной интеграции, используйте вызов:

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

Команда создает xml-файл по указанному пути.

Чтобы задать имя корневого xml-элемента для набора тестов, можно настроить параметр `junit_suite_name` в конфигурационном файле:

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

Спецификация JUnit XML, по-видимому, указывает, что атрибут `"time"` должен сообщать об общем времени выполнения теста, включая выполнение setup- и teardown- методов (1, 2). Это поведение `pytest` по умолчанию. Чтобы вместо этого сообщать только о длительности вызовов, настройте параметр `junit_duration_report` следующим образом:

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

2.14.1 record_property

Чтобы записать дополнительную информацию для теста, используйте фикстуру `record_property`:

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

Такая запись добавит дополнительное свойство `example_key="1"` к сгенерированному тегу `testcase`:

```
<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>
```

Эту функциональность также можно использовать совместно с пользовательскими маркерами:

```
# content of conftest.py


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))
```

И в тесте:

```
# content of test_function.py
import pytest


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

В файле получим:

```
<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>
```

Предупреждение: Пожалуйста, обратите внимание, что использование этой возможности приведет к записи некорректного с точки зрения JUnitXML-схем последних версий файла и может вызывать проблемы при работе с некоторыми серверами непрерывной интеграции.

2.14.2 record_xml_attribute

Чтобы добавить дополнительный атрибут в элемент `testcase`, можно использовать фикстуру `record_xml_attribute`. Ее также можно использовать для переопределения существующих значений:

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

В отличие от `record_property`, дочерний элемент в данном случае не добавляется. Вместо этого в элемент `testcase` будет добавлен атрибут `assertions="REQ-1234"`, а значение атрибута `classname` по умолчанию будет заменено на `"classname=custom_classname"`:

```
<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>
```

Предупреждение: `record_xml_attribute` пока используется в режиме эксперимента, и в будущем может быть заменен чем-то более мощным и/или общим. Однако сама функциональность как таковая будет сохранена.

Использование `record_xml_attribute` поверх `record_xml_property` может быть полезным при парсинге xml-отчетов средствами непрерывной интеграции. Однако некоторые парсеры допускают не любые элементы и атрибуты. Многие инструменты (как в примере ниже), используют xsd-схему для валидации входящих xml. Поэтому убедитесь, что имена атрибутов, которые вы используете, являются допустимыми для вашего парсера.

Ниже представлена схема, которую использует `Jenkins` для валидации xml-отчетов:

```
<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>
```

Предупреждение: Пожалуйста, обратите внимание, что использование этой возможности приведет к записи некорректного с точки зрения JUnitXML-схем последних версий файла и может вызывать проблемы при работе с некоторыми серверами непрерывной интеграции.

Чтобы добавить свойства каждому тесту из набора, можно использовать фикстуру `record_testsuite_property` с параметром `scope="session"` (в этом случае она будет применяться ко всем тестам тестовой сессии).

```
import pytest


@pytest.fixture(scope="session", autouse=True)
def log_global_env_facts(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")


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

Этой фикстуре передаются имя (`name`) и значение (`value`) тэга `<property>`, который добавляется на уровне тестового набора для генерируемого xml-файла:

```
<testsuite errors="0" failures="0" name="pytest" skipped="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>
```

`name` должно быть строкой, а `value` будет преобразовано в строку и корректно экранировано.

В отличие от случаев использования *record_property* и *record_xml_attribute* созданный xml-файл будет совместим с последним стандартом `xunit`.

2.15 Создание файлов в формате resultlog

Для создания машиночитаемых логов в формате `plain-text` можно выполнить

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

и просмотреть содержимое по указанному пути `path`. Эти файлы также используются ресурсом PyPy-test для отображения результатов тестов после ревизий.

Предупреждение: Поскольку эта возможность редко используется, она запланирована к удалению в `pytest 6.0`.

Если вы пользуетесь ею, рассмотрите возможность использования нового плагина pytest-reportlog .

Подробнее см.: the deprecation docs.

2.16 Отправка отчетов на сервис pastebin

Создание ссылки для каждого упавшего теста:

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

Эта команда отправит информацию о прохождении теста на удаленный сервис регистрации и сгенерирует ссылку для каждого падения. Тесты можно отбирать как обычно, или, например, добавить `-x`, если вы хотите отправить данные по конкретному упавшему тесту.

Создание ссылки для лога тестовой сессии:

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

В настоящее время реализовна регистрация только в сервисе http://bpaste.net.

Изменено в вресии 5.2

Если по каким-то причинам не удалось создать ссылку, вместо падения всего тестового набора генерируется предупреждение.

2.17 Подключение плагинов

В командной строке можно явно подгрузить какой-либо внутренний или внешний плагин, используя опцию `-p`:

```
pytest -p mypluginmodule
```

Опция принимает параметр `name`, который может быть:

- Полным именем модуля, записанным через точку, например `myproject.plugins`. Имя должно быть импортируемым.
- «Входным» именем плагина, которое передается в `setuptools` при регистрации плагина. К примеру, чтобы подгрузить pytest-cov , нужно использовать:

  ```
  pytest -p pytest_cov
  ```

2.18 Отключение плагинов

Чтобы отключить загрузку определенных плагинов во время вызова, используйте опцию `-p` с префиксом `no:`.

Пример: чтобы отключить загрузку плагина `doctest`, который отвечает за выполнение тестов из строк «docstring», вызовите `pytest` следующим образом:

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

2.19 Вызов `pytest` из кода Python

`pytest` можно вызвать прямо в коде Python:

```
pytest.main()
```

Такой способ эквивалентен вызову «pytest» из командной строки. В этом случае вместо исключения `SystemExit` возвращается статус завершения. Можно также передавать параметры и опции:

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

Дополнительные плагины можно указать в `pytest.main`:

```
# content of myinvoke.py
import pytest


class MyPlugin:
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")


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

Выполнив этот код, увидим, что `MyPlugin` был загружен и применен:

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

================================== ERRORS ==================================
_______________________ ERROR at setup of test_error _______________________

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

test_example.py:6: AssertionError
================================= FAILURES =================================
_______________________________ test_fail ________________________________

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
```

Примечание: Вызов `pytest.main()` приводит к тому, что импортируются не только тесты, но и все модули, которые они используют. Из-за механизма кэширования импорта `Python` последующие вызовы `pytest.main()` из того же процесса не будут учитывать изменения в файлах, внесенные между вызовами. Поэтому не рекомендуется многократное использование `pytest.main()` в одном и том же процессе (например, при перезапуске тестов).