


КОДИНГ

Async/await: асинхронные возможности в Python 3+

Николай enchantner Марков, 11.01.2017 15 мин на чтение  6

 15  44927



Содержание статьи

01. [Цикл передач на третьем канале](#)
02. [Сегодня в сопрограмме](#)
03. [Тетя Ася может все](#)
04. [Ближе к жизни](#)
05. [Сухой остаток](#)

Иногда у досточтимых джентльменов, обращающих внимание на разнообразие современных технологий асинхронности в Python, возникает вполне закономерный вопрос: «Что, черт возьми, со всем этим делать?» Тут вам и эвентлеты, и гринлеты, и корутины, и даже сам дьявол в ступе (Twisted). Поэтому собрались разработчики, почесали репу и решили: хватит терпеть четырнадцать конкурирующих стандартов, надо объединить их все в один! И как водится, в итоге стандартов стало пятнадцать... Ладно-ладно, шутка :). У событий, описанных в этой статье, конец будет более жизнеутверждающий.

Цикл передач на третьем канале

16 марта 2014 года произошло событие, которое привело к довольно бодрому холиварам, — вышел Python 3.4, а вместе с ним и своя внутренняя реализация event loop'a, которую окрестили **asyncio**. Идея у этой штуки была ровно такая, как я написал во введении: вместо того чтобы зависеть от внешних сишных реализаций отлова неблокирующих событий на сокетах (у gevent — libevent, у Tornado — IOloop и так далее), почему бы не встроить одну в сам язык?

Сказано — сделано. Теперь бывалые душители змей вместо того, чтобы в качестве ответа на набивший оскомину вопрос «Что такое

корутина?» нырять в генераторы и метод `.send()`, могли ткнуть в красивый декоратор `@asyncio.coroutine` и отправить вопрошающего читать документацию по нему.

Правда, сами разработчики отнеслись к новой спецификации довольно неоднозначно и с опаской. Хотя код и старался быть максимально совместимым по синтаксису со второй версией языка — проект **tulip**, который как раз был первой реализацией **PEP 3156** и лег в основу `asyncio`, был даже в каком-то виде бэкпортирован на устаревшую (да-да, я теперь ее буду называть только так) двойку.

Дело было еще и в том, что реализация, при всей ее красоте и приверженности дзену питона, получилась довольно неторопливая. Разогнанные **gevent** и **Tornado** все равно оказывались на многих задачах быстрее. Хотя, раз уж в народ в комьюнити настаивал на тюльпанах, в **Tornado** таки запилили экспериментальную поддержку `asyncio` вместо `IOLoop`, пусть она и была в разы медленнее. Но нашлось у новой реализации и преимущество — стабильность. Пусть соединения обрабатывались дольше, зато ответа в итоге дожидалась бОльшая доля клиентов, чем на многих других прославленных фреймворках. Да и ядро при этом, как ни странно, нагружалось чуть меньше.

Старт был дан, да и какой старт! Проекты на основе нового `event loop`'а начали возникать, как грибы после дождя, — обвязки для клиентов к базам данных, реализации различных протоколов, тысячи их! Появился даже сайт <http://asyncio.org/>, который собирал список всех этих проектов. Пусть даже этот сайт не открывался на момент написания статьи из-за ошибки DNS — можешь поверить на слово, там интересно. Надеюсь, он еще поднимется.

Но не все сразу заметили, что над новой версией Python завис великий и ужасный **PEP 492**...

Сегодня в сопрограмме

Так уж получилось, что довольно большое число людей изначально не до конца поняло смысл введения `asyncio` и считало его чем-то наподобие `gevent`, то есть сетевым или даже веб-фреймворком. Но суть у него была совсем другая — он открывал новые возможности асинхронного программирования в ядре языка.

Ты же помнишь в общих чертах, что такое генераторы и корутины (они же сопрограммы)? В контексте Python можно привести два определения генераторов, которые друг друга дополняют:

1. Генераторы — это объекты, предоставляющие интерфейс итератора, то есть запоминающие точку последнего останова, которые при каждом обращении к следующему элементу запускают какой-то ленивый код для его вычисления.
2. Генераторы — это функции, имеющие несколько точек входа и выхода, заданных с использованием оператора переключения контекста `yield`.

Корутины же всегда определялись как генераторы, которые, помимо того что вычисляли значения на каждом этапе, могли принимать на каждом обращении параметры, используемые для расчетов следующей итерации. По сути, это и есть вычислительные единицы в контексте того, что называют кооперативной многозадачностью, — можно сделать много таких легковесных корутин, которые будут очень быстро передавать друг другу управление.

В случае сетевого программирования именно это и позволяет нам быстро опрашивать события на сокете, обслуживая тысячи клиентов сразу. Ну или, в общем случае, мы можем **написать асинхронный драйвер** для любого I/O-устройства, будь то файловая система на `block device` или, скажем, воткнутая в USB Arduino.

Да, в ядре Python есть пара библиотек, которые изначально предназначались для похожих целей, — это `asyncore` и `asynchat`, но они были, по сути, экспериментальной оберткой над сетевыми сокетами, и код для них написан довольно давно. Если ты сейчас, в

начале 2017 года, читаешь эту статью — значит, настало время записать их в музейные экспонаты, потому что asyncio лучше.

Давай забудем на время про несвежий Python 2 и взглянем на реализацию простейшего асинхронного эхо-сервера в Python 3.4:

```
#!/usr/bin/env python
import asyncio

class EchoProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        self.transport = transport
        print('Connection from {}'.format(
            transport.get_extra_info('peername')
        ))

    def data_received(self, data):
        message = data.decode()
        print("Echoing back: {!r}".format(message))
        self.transport.write(data)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    server_coro = loop.create_server(EchoProtocol, '127.0.0.1', 7
    server = loop.run_until_complete(server_coro)
    loop.run_forever()
```

Нам ничто не мешает подключиться к этому серверу несколькими клиентами и отвечать всем сразу. Это можно проверить, например, с помощью netcat. При этом на сокете будет использоваться лучшая реализация поллинга событий из доступных в системе, в современном Linux это, разумеется, epoll.

Да, этот код асинхронный, но callback hell — тоже вещь довольно неприятная. Немного неудобно описывать асинхронные обработчики как гроздь висящих друг на друге колбэков, не находишь? Отсюда и проистекает тот самый классический вопрос: как же нам, кабанам,

писать асинхронный код, который не был бы похож на спагетти, а просто выглядел бы несложно и императивно? На этом месте передай привет в камеру ноутбука (если она у тебя не заклеена по совету [1]) тем, кто активно использует Twisted или, скажем, пишет на JavaScript, и поехали дальше.

А теперь давай возьмем Python 3.5 (давно пора) и напишем все на нем.

```
import asyncio

async def handle_tcp_echo(reader, writer):
    print('Connection from {}'.format(
        writer.get_extra_info('peername')
    ))
    while True:
        data = await reader.read(100)
        if data:
            message = data.decode()
            print("Echoing back: {!r}".format(message))
            writer.write(data)
            await writer.drain()
        else:
            print("Terminating connection")
            writer.close()
            break

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        asyncio.ensure_future(
            asyncio.start_server(handle_tcp_echo, '127.0.0.1', 7777,
                                loop=loop)
        )
    )
    loop.run_forever()
```

Красиво? Никаких классов, просто цикл, в котором мы принимаем

подключения и работаем с ними. Если этот код сейчас взорвал тебе мозг, то не волнуйся, мы рассмотрим основы этого подхода.

Для создания подобных серверов и вообще красивой асинхронной работы в Python Дэвид Бизли (обожаю этого парня) написал свою собственную библиотеку под названием **curio**. Крайне рекомендую ознакомиться, библиотека экспериментальная, но очень приятная. Например, код TCP-сервера на ней может выглядеть так:

```
from curio import run, spawn
from curio.socket import *

async def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    print('Server listening at', address)
    async with sock:
        while True:
            client, addr = await sock.accept()
            await spawn(echo_client(client, addr))

async def echo_client(client, addr):
    print('Connection from', addr)
    async with client:
        while True:
            data = await client.recv(100000)
            if not data:
                break
            await client.sendall(data)
    print('Connection closed')

if __name__ == '__main__':
    run(echo_server(('', 25000)))
```

Несложно заметить, что в случае асинхронного программирования подобным образом в питоне все будет крутиться (каламбур) вокруг того самого внутреннего IOLoop'a, который будет связывать события с их обработчиками. Одной из основных проблем, как я уже говорил, остается скорость — связка Python 2 + gevent, которая использует крайне быстрый libev, по производительности показывает гораздо лучшие результаты.

Но зачем держаться за прошлое? Во-первых, есть curio (см. врезку), а во-вторых, уже есть еще одна, гораздо более скоростная реализация event loop'a, написанная как подключаемый плагин для asyncio, — **uvloop**, основанный на адски быстром libuv.

Что, уже чувствуешь ураганный ветер из монитора?

Тетя Ася может все

Итак, что же мы имеем? Мы имеем асинхронные функции, они же корутины. Вот такие:

```
In [1]: async def hello(name):  
...:     return "Hello, {}".format(name)
```

Если мы просто так возьмем и вызовем эту функцию, ничего не произойдет, потому что нам вернется ленивая корутина. Но мы же помним из статей о генераторах, что нам нужно сделать, чтобы ее запустить? Правильно — передать ей контекст через оператор yield. Формально этого yield'a у нее нет, но мы можем послать в нее значение для того, чтобы «промотать» корутину до следующего переключения контекста:

```
In [2]: h = hello("Vasya")  
In [3]: h.send(None)  
StopIteration      Traceback (most recent call last)  
<ipython-input-3-919e6cc93359> in <module>()
```



```
----> 1 h.send(None)

StopIteration: Hello, Vasya!
```

Что-то знакомое, да? Генератор исчерпался и выкинул `StopIteration`. Можно, конечно, написать обработчик исключения и дергать корутины через него, но это будет выглядеть крайне странно. Но! Мы можем очень легко вызвать эту корутину из другой корутины!

```
In [4]: async def call_vasya():
...:     print(await hello("Vasya"))
```

Да, мы ее просто «подождем», как маму из той самой песни. Таким образом мы можем выстроить целый разветвленный граф из корутин, которые «ожидают» друг друга и передают управление туда и обратно. Если ты сейчас вскочил с кресла и воскликнул: «Да это же кооперативная многозадачность!» — молодец, к этому все и шло.

Кстати, если все равно назло маме вызвать функцию без `await` внутри корутины, то нам не просто вернется coroutine object, но еще и в консоль упадет большой warning и напоминание coroutine 'blablabla' was never awaited. Ее никто не дождался, поэтому она обиделась и не стала исполняться. Но такие сообщения очень помогают в отладке.

А еще — нельзя просто так взять и вызвать `await` в интерактивном REPL'е, потому что он не является корутиной сам по себе:

```
In [5]: await call_vasya()
File "<ipython-input-5-7736397d2048>", line 1
  await call_vasya()
          ^
SyntaxError: invalid syntax
```

В остальных случаях `await` можно писать где угодно внутри корутины,

за исключением списковых включений (они же list comprehensions, и это обещают добавить в ближайших релизах) и лямбд (потому что они сами не корутины). А async можно использовать, например, для методов в классе (за исключением «приватных» **методов**, которые могут дергаться самим Python'ом, понятия не имеющим, что у вас там корутина).

Давай напишем, как нам теперь реально запустить всю эту катавасию:

```
In [6]: import asyncio
In [7]: loop = asyncio.get_event_loop()
In [8]: loop.run_until_complete(call_vasya())
Out[8]: 'Hello, Vasya!'
```

Все довольно просто: мы достаем event loop и заставляем корутину запуститься в нем. Много кода, скажешь? Не особо на самом деле, особенно с учетом того, какие преимущества это нам дает.

Ближе к жизни

Я мог бы рассказать еще про такие штуки, как async for и async with:

- первое — это просто итерация по объекту с ожиданием корутины на каждом шаге, объект должен иметь интерфейсные методы **aiter** и **anext**;
- второе — управление контекстом через вызовы корутин — необходимые методы, соответственно, **aenter** и **aexit**.

Но лучше почитай про всякую глубинную магию по ссылкам во врезке, а сейчас давай обратимся к более практическому примеру.



WWW

Еще пара трюков

Отличная презентация по экосистеме и возможностям

Шикарный пост на тему подхода в целом, а не конкретно про Python

Есть банальная, казалось бы, задача, которая практически нереализуема во втором питоне, — запустить подпроцесс и асинхронно читать его вывод по мере поступления, как, собственно, и должен работать PIPE.

В последнее время я сильно разочаровался во встроенном модуле subprocess, но к нам на помощь спешит асинхронная реализация, на данный момент часть asyncio. И там это делается просто и красиво.

```
#!/usr/bin/env python

import asyncio
import shlex

async def _read_stream(stream, cb):
    """
    Асинхронно читаем из потока
    """
    while True:
        line = await stream.readline()
        if line:
            cb(line)
        else:
            break

async def _stream_subprocess(cmd, stdout_cb, stderr_cb):
    """
    Создаем процесс и делаем два экземпляра корутин,
    читающих из его stdout и stderr
    """
    process = await asyncio.create_subprocess_exec(
```

```
        *cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE
    )
    await asyncio.wait([
        _read_stream(process.stdout, stdout_cb),
        _read_stream(process.stderr, stderr_cb)
    ])
    return await process.wait()

def execute(cmd, stdout_cb, stderr_cb):
    """
    Оборачиваем все в event loop
    """
    loop = asyncio.get_event_loop()
    rc = loop.run_until_complete(
        _stream_subprocess(
            cmd,
            stdout_cb,
            stderr_cb,
        )
    )
    loop.close()
    return rc

if __name__ == '__main__':
    """
    Запускаем команду, передавая ей асинхронные обработчики
    для stdout и stderr
    """
    cmd = (
        "bash -c \"echo stdout && sleep 1 && \"
        "echo stderr 1>&2 && sleep 1 && echo done\""
    )
    print(execute(
        shlex.split(cmd),
        lambda x: print("STDOUT: %s" % x),
        lambda x: print("STDERR: %s" % x),
    ))
```



WWW

Пример нагло стырен [отсюда](#) с небольшими модификациями, потому что он простой и красивый.

Да, это много кода. Но он должен казаться гораздо более понятным после объяснений выше, да и вообще он довольно легко читается. Я, честно сказать, искренне надеюсь, что именно возможности наподобие описанных позволят наконец большому количеству народа распробовать Python 3.5 и перейти на него окончательно.

Сухой остаток

Зачем все это нужно? Затем, что слишком много программ рано или поздно упрутся в блокировки — когда мы читаем из сокета, когда мы ждем вывод от процесса, когда мы ждем сигнал от устройства и т. д. и т. п. Обычно такие вещи делаются, например, бесконечным циклом — мы будем стучаться, пока не появятся новые данные нам для обработки, а потом условие выполнится и запустится какой-то код.

Так вот, зачем так делать, если мы можем попросить систему саму отправить нам из kernel space (и опять все дружно скажем «Ave eroll!») сообщение о том, что у нас есть новые данные? Не тратя вычислительные ресурсы на ненужный код.

Я думаю, всем любителям питона стоит исследовать этот новый мир, который нам стал доступен совсем недавно и теперь активно развивается. Нам больше не надо патчить модуль `socket` через `gevent` и терпеть адские баги. У нас уже есть готовые асинхронные

библиотеки для работы с базами данных (например, [aiopg](#)), протоколами ([aiozmq](#)), сторонними сервисами через API ([aiobotocore](#)) и написания скоростных серверов ([aiohttp](#)).

Мало ссылок? Ладно, вот еще одна: реализация протокола HTTP2, которую можно гонять хоть на потоках, хоть на корутинах, — очень интересный проект [hyper-h2](#).

Так чего ты еще тут сидишь? Иди пиши код! Удачи!



Николай enchantner Марков

Теги: [Python](#) [Выбор редактора](#) [исходный код](#) [синхронизация](#) [Статьи](#)

6 комментариев



Nestor Smirnoff

11.01.2017 at 18:49

Спасибо, интересный обзор asyncio. Еще есть библиотека для работы с PostgreSQL — [asynprg](#). На практике в 3-4 раза быстрее [psycopg2](#)

[Ответить](#)



afiskon

12.01.2017 at 17:09

Потрясающая статья. Большое спасибо вам за нее!

[Ответить](#)



heavis

13.01.2017 at 09:05

Простите мне пошлость, но «бывалые душителы змей» звучит очень двусмысленно:)

если серьезно:

— спасибо за статью, здорово написано!

— из примера с python 3.4 не понял, что такое callback hell. Возможно, пример слишком простой. Можно подробнее для нуба?

— если есть, дайте пожалуйста ссылку на один самый крутой tutorial по асинхронному python (чтобы python3.5+, разрабатывалось что-то полезное и всё было подробно разжевано. если на русском — будет великолепно, но это не обязательно).

[Ответить](#)



baragoz

13.01.2017 at 10:14

Это ж Хакер, я думаю, что про душителй змей это специально 😊

[Ответить](#)



Dmitry Tsatsarin

20.01.2017 at 23:48

ребят, офтопик, я тут новичок, а тут всегда так мало народа по сравнению с хабром?

[Ответить](#)



baragoz

22.01.2017 at 22:11

Ну это ж типа журнал, а там типа блоги, user generated content и все такое :). Хотя и тут бы не помешало

[Ответить](#)



br0ke

02.06.2018 at 18:26

Спасибо! Комментарий отправлен на проверку модератору.

Полинтернета прочел — нифига не понял, а тут понятно написано. Спасибо!

[Ответить](#)