PYTHON*

Немного фактов о python asyncio

ИЗ ПЕСОЧНИЦЫ

Amelius0712 7 ноября 2016 в 16:25 30,4k

Всем привет! Хотелось бы поделиться опытом использования python asyncio. За полтора года использования в продакшене накопился некоторый опыт, общие приемы, облегчающие жизнь. Естественно, были и грабли, о которых также стоит упомянуть, ибо это поможет сэкономить кучу времени тем, кто только начинает использовать в своих приложениях asyncio. Кому интересно — прошу под кат.

Немного истории

Asyncio появился в Python версии 3.4, в 3.5 был добавлен более приятный глазу async/await синтаксис. Asyncio предоставляет из коробки Event loop, Future, Task, Coroutine, I/O multiplexing, Synchronization primitives. Это, конечно, не мало, но для полноценной разработки недостаточно. Для этого есть сторонние библиотеки. Отличная подборка есть вот тут. У себя в компании мы используем asyncio вместе с набором сторонних библиотек для написания микросервисов. По своей природе наши сервисы больше ориентированы на I/O нежели на CPU, так что для нас asyncio отлично подходит.

Собственно факты

Это не учебник по asyncio. Я не буду объяснять, почему асинхронный ввод/вывод это хорошо, или почему бы не использовать потоки. Не будет рассказов о корутинах, генераторах, event loop'ах и т.д. Также тут не будет никаких бенчмарков и сравнений с другими языками. Поехали!

Debug

Во-первых, PYTHONASYNCIODEBUG. Это переменная окружения, которая включает дебаг режим. Например, можно увидеть сообщения о том, что вы объявили функцию как корутину, но вызываете как обычную функцию(актуально для python3.4). Также необходимо настроить asyncio logger на уровень дебаг и еще разрешить вывод ResourseWarning. Можно увидеть много интересного: сообщения о том, что вы забыли закрыть транспорт или сам event loop(читай — забыли освободить ресурсы). Сравните запуск следующего кода с параметром интерпретатора -Wdefault и переменной окружения PYTHONASYNCIODEBUG=1 и без них (здесь и далее в примерах кода я буду опускать некоторые несущественные части такие как import или обработка исключений):

```
@asyncio.coroutine
def test():
    pass

loop = asyncio.get_event_loop()
test()
```

Правильное завершение

Кстати об освобождении ресурсов. Event loop надо уметь правильно остановить, дождавшись корректного заверешения всех тасок, закрытия соединений и т.д. И если с использованием run_until_complete() особых проблем нет, то с run_forever() все немного сложнее. Метод close() у event loop'а можно вызвать, только если он уже остановлен — т.е. после метода stop(). Лучше всего это сделать с помощью сигналов:

```
def handler(loop):
    loop.remove_signal_handler(signal.SIGTERM)
    loop.stop()

loop = asyncio.get_event_loop()
loop.add_signal_handler(signal.SIGTERM, handler, loop)

try:
    loop.run_forever()
finally:
    loop.close()
```

Далее в примерах кода я все же буду концентрироваться на самой сути, а не на правильном завершении программы.

Запуск блокирующего кода

Естественно, не для всего есть асинхронные библиотеки. Некоторый код так и остается блокирующим, и его надо как-то запускать, чтобы он не блокировал наш event loop. Для этого есть хороший метод run_in_executor(), который запускает то, что вы ему передали в одном из потоков встроенного пула, не блокируя основной поток с event loop'ом. Все бы хорошо, но с этим есть 2 проблемы. Во-первых, размер стандартного пула всего 5. Во-

вторых, в asyncio синхронный dns resolver, который запускается именно таким образом во встроенном пуле. Значит, за пул всего в 5 потоков будут конкурировать ваши синхронные операции, плюс все кому надо сделать getaddrinfo(). Выход — использовать свой пул. Всегда:

```
def blocking_function():
    time.sleep(42)

pool = ThreadPoolExecutor(max_workers=multiprocessing.cpu_count())
loop = asyncio.get_event_loop()
loop.run_in_executor(pool, blocking_function)
loop.close()
```

Коварные Future

У Future есть одна очень интересная особенность: если в ней произойдет исключение — вы об этом ничего не узнаете, если только явно не спросите об этом у самой future. В документации есть хороший пример на эту тему. Вы увидите, что было исключение, только когда gc будет удалять объект future. Отсюда следует простое правило — всегда проверяете результат вашей future. Даже если по вашей задумке код внутри future должен просто крутиться в бесконечном цикле, и, казалось бы, негде проверять результат — все равно надо обработать исключения, например так:

```
async def handle_exception():
    try:
        await bug()
    except Exception:
        print('TADA!')
```

```
async def bug():
    raise Exception()

loop = asyncio.get_event_loop()
loop.create_task(handle_exception())
loop.run_forever()
loop.close()
```

await и __init__()

Невозможно. Магический метод __init__() не может содержать асинхронный код. Есть два пути. Или сделать у класса еще один метод, например, initialize(), который уже будет корутиной. Он будет содержать весь асинхронный код для инициализации, и его надо будет вызывать после создания объекта. Выглядит ужасно. Поэтому принято использовать функции-фабрики. Поясню на примере:

```
class Foo:
    def __init__(self, reader, writer, loop, *args, **kwargs):
        self._reader = reader
        self._writer = writer
        self._loop = loop

async def create_foo(loop):
    reader, writer = await asyncio.open_connection('127.0.0.1',
8888, loop=loop)
    return Foo(reader, writer, loop)

loop = asyncio.get_event_loop()
foo = loop.run_until_complete(create_foo(loop))
print(foo)
loop.close()
```

Wake up, Neo

Скажем, у вас есть таска, которая крутится в event loop'e и

периодически сбрасывает какой-нибудь буфер. Можно написать такой код:

```
async def flush_task():
    while True:
        # flushing...
        await asyncio.sleep(FLUSH_TIMEOUT)
```

Сделать create_task() — и все вроде бы хорошо, кроме одного: что делать, если по завершении вам необходимо принудительно сбросить содержимое буфера? Как заставить таску «проснуться»? Тут на помощь приходят примитивы синхронизации:

```
class Foo:

    def __init__(self, loop, *args, **kwargs):
        self._loop = loop
        self._waiter = asyncio.Event()
        self._flush_future =
self._loop.create_task(self.flush_task())

    async def flush_task(self):
        while True:
            try:
                await asyncio.wait_for(self._waiter.wait(),
timeout=FLUSH_TIMEOUT, loop=self._loop)
            except asyncio.TimeoutError:
                pass
            # flushing ...
            self._waiter.clear()

    def force_flush():
        self._waiter.set()

loop = asyncio.get_event_loop()
foo = Foo(loop)
loop.run_forever()
loop.close()
```

Тестирования

Тестировать асинхронный код можно и нужно. И делать это так же просто, как и в случае синхронного кода:

```
class TestCase(unittest.TestCase):

    def setUp(self):
        self.loop = asyncio.new_event_loop()
        asyncio.set_event_loop(None)

    def tearDown(self):
        self.loop.close()

    def test_001(self):
        async def func():
            self.assertEqual(42, 42)
        self.loop.run_until_complete(func())
```

Тесты отлично изолированы, т.к. в каждом новом тесте используется свой event loop. А можно пойти дальше и использовать pytest, где есть удобные декораторы.

Источники вдохновения

Прежде всего — личный опыт. Многое из перечисленного было осознано в результате «ловли граблей», а затем изучения документации и исходников asyncio. Также отличными примерами послужили исходники популярных библиотек, таких как aiohttp, aioredis, aiopg.

Спасибо всем, кто дочитал статью до конца. Удачи с asyncio!

Проголосовать:

+31

Поделиться:

Сохранить:

Комментарии (18)

Похожие публикации

Многопользовательский онлайн-шутер на WebGL и asyncio, часть вторая 5

Alex10 • 16 февраля 2016 в 11:06

Asyncio Tarantool Queue, вставай в очередь 14

shveenkov • 24 ноября 2015 в 15:00

Работа с ZeroMQ и PostgreSQL в asyncio 36

svetlov • 7 апреля 2014 в 01:04

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета) 69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16