



vlakir 24 мая 2022 в 22:59

Асинхронный python без головной боли (часть 1)

🕒 14 мин 👁 256K

Python*, Программирование*

Тutorial

1. Почему так сложно понять asyncio

Асинхронное программирование традиционно относят к темам для "продвинутых". Действительно, у новичков часто возникают сложности с практическим освоением асинхронности. В случае python на то есть весьма веские причины:

1. Асинхронность в python была стандартизирована сравнительно недавно. Библиотека `asyncio` появилась впервые в версии 3.5 (то есть в 2015 году), хотя возможность костыльно писать асинхронные приложения и даже фреймворки, конечно, была и раньше. Соответственно у Лутца она не описана, а, как всем известно, "чего у Лутца нет, того и знать не надо".
2. Рекомендуемый синтаксис асинхронных команд неоднократно менялся уже и после первого появления `asyncio`. В сети бродит огромное количество статей и роликов, использующих архаичный код различной степени давности, только сбивающий новичков с толку.

[Разблокировать темную тему](#)

ЧИТАЮТ СЕЙЧАС

[Запуск ракеты Ангара — неделю спустя](#)

👁 9.2K 💬 21 +21

[Я скучаю по механикам из старых игр](#)

👁 3.1K 💬 42 +42

[Представлен проект Windrecorder с открытым исходным кодом для записи и поиска всего, что происходило на экране в Windows](#)

👁 3.7K 💬 4 +4

[USDT приходит в TON, а доллары — в Telegram](#)

👁 12K 💬 45 +45



+135



992



56

пользовательских приложений. Там столько всего — глаза разбегаются. А между тем:
"Вам нужно знать всего около семи функций для использования `asyncio`" (с) Юрий Селиванов, автор PEP 492, в которой были добавлены инструкции `async` и `await`

На самом деле наша повседневная жизнь буквально наполнена асинхронностью.

Утром меня поднимает с кровати будильник в телефоне. Я когда-то давно поставил его на 8:30 и с тех пор он исправно выполняет свою работу. Чтобы понять когда вставать, мне не нужно таращиться на часы всю ночь напролет. Нет нужды и периодически на них посматривать (скажем, с интервалом в 5 минут). Да я вообще не думаю по ночам о времени, мой мозг занят более интересными задачами — просмотром снов, например. Асинхронная функция "подъем" находится в режиме ожидания. Как только произойдет событие "на часах 8:30", она сама даст о себе знать омерзительным Jingle Bells.

Иногда по выходным мы с собакой выезжаем на рыбалку. Добравшись до берега, я снаряжаю и забрасываю несколько донок с колокольчиками. И... Переключаюсь на другие задачи: разговариваю с собакой, любуюсь красотами природы, истребляю на себе комаров. Я не думаю о рыбе. Задачи "поймать рыбу удочкой N" находятся в режиме ожидания. Когда рыба будет готова к общению, одна из удочек сама даст о себе знать звонком колокольчика.

Будь я автором самого толстого в мире учебника по python, я бы рассказывал читателям про асинхронное программирование уже с первых страниц. Вот только написали "Hello, world!" и тут же приступили к созданию "Hello, asynchronous world!". А уже потом циклы, условия и все такое.

Но при написании этой статьи я все же облегчил себе задачу, предположив, что читатели уже знакомы с основами python и им не придется втолковывать что такое генераторы или менеджеры контекста. А если кто-то не знаком, тогда сейчас самое время ознакомиться.

Пара слов о терминологии

«Микрон» представил российский ПЛК на базе RISC-V на выставке ExpoElectronica 2024

 4.9K  36  +36

Любят ли на Хабре пет-проекты?

Опрос 

РАБОТА

Django разработчик
41 вакансия

Data Scientist
59 вакансий

Python разработчик
138 вакансий

Все вакансии

БЛИЖАЙШИЕ СОБЫТИЯ

В настоящем руководстве я стараюсь придерживаться не академических, а сленговых терминов, принятых в русскоязычных командах, в которых мне довелось работать. То есть "корутина", а не "сопрограмма", "футура", а не "фьючерс" и т. д. При всем при том, я еще не столь низко пал, чтобы, скажем, задачу именовать "таской". Если в вашем проекте приняты другие названия, прошу отнестись с пониманием и не устраивать терминологический холивар.

Внимание! Все примеры отлажены в консольном python 3.10. Вероятно в ближайших последующих версиях также работать будут. Однако обратной совместимости со старыми версиями не гарантирую. Если у вас что-то пошло не так, попробуйте, установить 3.10 и/или не пользоваться Jupyter.

2. Первое асинхронное приложение

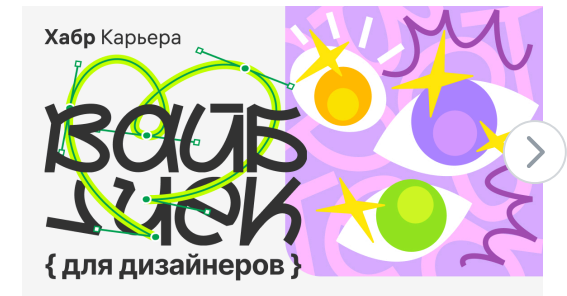
Предположим, у нас есть две функции в каждой из которых есть некая "быстрая" операция (например, арифметическое вычисление) и "медленная" операция ввода/вывода. Детали реализации медленной операции нам сейчас не важны. Будем моделировать ее функцией `time.sleep()`. Наша задача - выполнить обе задачи как можно быстрее.

Традиционное решение "в лоб":


Пример 2.1


```
import time

def fun1(x):
    print(x**2)
    time.sleep(3)
```



Дизайнеры, выбирайте себе компанию по вайбам на Хабр Карьере

 15 – 28 апреля

 Онлайн

[Подробнее в календаре](#)

```
print('fun1 завершена')

def fun2(x):
    print(x**0.5)
    time.sleep(3)
    print('fun2 завершена')

def main():
    fun1(4)
    fun2(4)

print(time.strftime('%X'))

main()

print(time.strftime('%X'))
```

Никаких сюрпризов - `fun2` честно ждет пока полностью отработает `fun1` (и быстрая ее часть, и медленная) и только потом начинает выполнять свою работу. Весь процесс занимает $3 + 3 = 6$ секунд. Строго говоря, чуть больше чем 6 за счет "быстрых" арифметических операций, но в выбранном масштабе разницу уловить невозможно.

Теперь попробуем сделать то же самое, но в асинхронном режиме. Пока просто запустите предложенный код, подробно мы его разберем чуть позже.

Пример 2.2

```
import asyncio
import time

async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')

async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')

async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2

print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

Сюрприз! Мгновенно выполнились быстрые части обеих функций и затем через 3 секунды (3, а не 6!) одновременно появились оба текстовых сообщения. Полное ощущение, что функции выполнились параллельно (на самом деле нет).

А можно аналогичным образом добавить еще одну функцию-соню? Пожалуйста — хоть сто! Общее время выполнения программы будет по-прежнему определяться самой "тормознутой" из них. Добро пожаловать в асинхронный мир!

Что изменилось в коде?

1. Перед определениями функций появился префикс `async`. Он говорит интерпретатору, что функция должна выполняться асинхронно.
2. Вместо привычного `time.sleep` мы использовали `asyncio.sleep`. Это "неблокирующий sleep". В рамках функции ведет себя так же, как традиционный, но не останавливает интерпретатор в целом.
3. Перед вызовом асинхронных функций появился префикс `await`. Он говорит интерпретатору примерно следующее: *"я тут возможно немного потуплю, но ты меня не жди — пусть выполняется другой код, а когда у меня будет настроение продолжиться, я тебе маякну"*.
4. На базе функций мы при помощи `asyncio.create_task` создали задачи (что это такое разберем позже) и запустили все это при помощи `asyncio.run`

Как это работает:

- выполнялась быстрая часть функции `fun1`
- `fun1` сказала интерпретатору *"иди дальше, я посплю 3 секунды"*
- выполнялась быстрая часть функции `fun2`

- `fun2` сказала интерпретатору *"иди дальше, я посплю 3 секунды"*
- интерпретатору дальше делать нечего, поэтому он ждет пока ему маякнет первая проснувшаяся функция
- на доли миллисекунды раньше проснулась `fun1` (она ведь и уснула чуть раньше) и отпартовала нам об успешном завершении
- то же самое сделала функция `fun2`

Замените "поплю" на "пошлю запрос удаленному сервису и буду ждать ответа" и вы поймете как работает реальное асинхронное приложение.

Возможно в других руководствах вам встретится "старомодный" код типа:

Пример 2.3

```
import asyncio
import time

async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')

async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')
```

```
print(time.strftime('%X'))

loop = asyncio.get_event_loop()
task1 = loop.create_task(fun1(4))
task2 = loop.create_task(fun2(4))
loop.run_until_complete(asyncio.wait([task1, task2]))

print(time.strftime('%X'))
```

Результат тот же самый, но появилось упоминание о каком-то непонятном цикле событий (event loop) и вместо одной `asyncio.run` используются аж три функции: `asyncio.wait`, `asyncio.get_event_loop` и `asyncio.run_until_complete`. Кроме того, если вы используете python версии 3.10+, в консоль прилетает раздражающее предупреждение `DeprecationWarning: There is no current event loop`, что само по себе наводит на мысль, что мы делаем что-то слегка не так.

Давайте пока руководствоваться Дзен питона: *"Простое лучше, чем сложное"*, а цикл событий сам придет к нам... в свое время.

Пара слов о "медленных" операциях

Как правило, это все, что связано с вводом выводом: получение результата http-запроса, файловые операции, обращение к базе данных.

Однако следует четко понимать: для эффективного использования с `asyncio` любой медленный интерфейс должен поддерживать асинхронные функции. Иначе никакого выигрыша в производительности вы не получите. Попробуйте использовать в примере 2.2 `time.sleep` вместо `asyncio.sleep` и вы поймете о чем я.

Что касается http-запросов, то здесь есть великолепная библиотека `aiohttp`, честно реализующая асинхронный доступ к веб-серверу. С файловыми операциями сложнее. В Linux доступ к файловой системе по определению не асинхронный, поэтому, несмотря на наличие удобной библиотеки `aiofiles`, где-то в ее глубине всегда будет иметь место многопоточный "мостик" к низкоуровневым функциям ОС. С доступом к БД примерно то же самое. Вроде бы, последние версии `SQLAlchemy` поддерживают асинхронный доступ, но что-то мне подсказывает, что в основе там все тот же старый добрый `ThreadPool`. С другой стороны, в веб-приложениях львиная доля задержек относится именно к сетевому общению, так что "не вполне асинхронный" доступ к локальным ресурсам обычно не является бутылочным горлышком.

Внимательные читатели меня поправили в комментарях. В Linux, начиная с ядра 5.1, есть полноценный асинхронный интерфейс `io_uring` и это прекрасно. Кому интересны детали, рекомендую пройти [вот сюда](#).

3. Асинхронные функции и корутины

Теперь давайте немного разберемся с типами. Вернемся к "неасинхронному" примеру 2.1, слегка модифицировав его:

Пример 3.1

```
import time

def fun1(x):
    print(x**2)
    time.sleep(3)
    print('fun1 завершена')
```

```
def fun2(x):  
    print(x**0.5)  
    time.sleep(3)  
    print('fun2 завершена')
```

```
def main():  
    fun1(4)  
    fun2(4)
```

```
print(type(fun1))
```

```
print(type(fun1(4)))
```

Все вполне ожидаемо. Функция имеет тип `<class 'function'>`, а ее результат - `<class 'NoneType'>`

Теперь аналогичным образом исследуем "асинхронный" пример 2.2:

Пример 3.2

```
import asyncio  
import time
```

```
async def fun1(x):  
    print(x**2)
```

```
await asyncio.sleep(3)
print('fun1 завершена')
```

```
async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')
```

```
async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2
```

```
print(type(fun1))
```

```
print(type(fun1(4)))
```

Уже интереснее! Класс функции не изменился, но благодаря ключевому слову `async` она теперь возвращает не `<class 'NoneType'>`, а `<class 'coroutine'>`. Ничто превратилось в нечто! На сцену выходит новая сущность - **корутина**.

Что нам нужно знать о корутине? На начальном этапе немного. Помните как в python устроен генератор? Ну, это то, что функция начинает возвращать, если в нее добавить `yield` вместо `return`. Так вот, корутина — это разновидность генератора.

Корутина дает интерпретатору возможность возобновить базовую функцию, которая была приостановлена в месте размещения ключевого слова `await`.

И вот тут начинается терминологическая путаница, которая попила немало крови добрых разработчиков на собеседованиях. Сплошь и рядом корутиной называют **саму функцию**, содержащую `await`. Строго говоря, это неправильно. Корутина — это то, что **возвращает** функция с `await`. Чувствуете разницу между `f` и `f()`?

С генераторами, кстати, та же самая история. Генератором как-то повелось называть функцию, содержащую `yield`, хотя по правильному-то она "генераторная функция". А генератор — это именно тот объект, который генераторная функция возвращает.

Далее по тексту мы постараемся придерживаться правильной терминологии: асинхронная (или корутинная) функция — это `f`, а корутина — `f()`. Но если вы в разговоре назовете корутиной асинхронную функцию, беды большой не произойдет, вас поймут. *"Не важно, какого цвета кошка, лишь бы она ловила мышей"* (с) тов. Дэн Сяопин

4. Футуры и задачи

Продолжим исследовать нашу программу из примера 2.2. Помнится, на базе корутин мы там создали какие-то загадочные **задачи**:

Пример 4.1

```
import asyncio

async def fun1(x):
    print(x**2)
```

```
await asyncio.sleep(3)
print('fun1 завершена')
```

```
async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')
```

```
async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    print(type(task1))
    print(task1.__class__.__bases__)

    await task1
    await task2
```

```
asyncio.run(main())
```

Ага, значит задача (что бы это ни значило) имеет тип `<class '_asyncio.Task'>`. Привет, капитан Очевидность!

А кто ваша мама, ребята? А мама наша — ~~анархия~~ какая-то еще более загадочная **футура** (`<class '_asyncio.Future'>`).

В `asyncio` все шиворот-навыворот, поэтому сначала выясним что такое футура (которую мы видим впервые в жизни), а потом разберемся с ее дочкой задачей (с которой мы уже

имели честь познакомиться в предыдущем разделе).

Футура (если совсем упрощенно) — это оболочка для некой асинхронной сущности, позволяющая выполнять ее "как бы одновременно" с другими асинхронными сущностями, переключаясь от одной сущности к другой в точках, обозначенных ключевым словом `await`

Кроме того футура имеет внутреннюю переменную "результат", которая доступна через `.result()` и устанавливается через `.set_result(value)`. Пока ничего не надо делать с этим знанием, оно пригодится в дальнейшем.

У футуры на самом деле еще много чего есть внутри, но на данном этапе не будем слишком углубляться. Футуры в чистом виде используются в основном разработчиками фреймворков, нам же для разработки приложений приходится иметь дело с их дочками — **задачами**.

Задача — это частный случай футуры, предназначенный для оборачивания корутины.

Все трагически усложняется

Вернемся к примеру 2.2 и опишем его логику заново, используя теперь уже знакомые нам термины — корутины и задачи:

- корутину асинхронной функции `fun1` обернули задачей `task1`
- корутину асинхронной функции `fun2` обернули задачей `task2`
- в асинхронной функции `main` обозначили точку переключения к задаче `task1`
- в асинхронной функции `main` обозначили точку переключения к задаче `task2`
- корутину асинхронной функции `main` передали в функцию `asyncio.run`

Бр-р-р, ужас какой... Воистину: *"Во многих мудрости много печали; и кто умножает познания, умножает скорбь"* (Еккл. 1:18)

Все счастливо упрощается

А можно проще? Ведь понятие корутина нам необходимо, только чтобы отличать функцию от результата ее выполнения. Давайте попробуем временно забыть про них. Попробуем также перефразировать неуклюжие "точки переключения" и вот эти вот все "обернули-передали". Кроме того, поскольку `asyncio.run` — это единственная рекомендованная точка входа в приложение для python 3.8+, ее отдельное упоминание тоже совершенно излишне для понимания логики нашего приложения.

А теперь (барабанная дробь)... Мы вообще уберем из кода **все** упоминания об асинхронности. Я понимаю, что работать не будет, но все же давайте посмотрим что получится:

Пример 4.2 (не работающий)

```
def fun1(x):  
    print(x**2)  
  
    # запустили ожидание  
    sleep(3)  
  
    print('fun1 завершена')  
  
def fun2(x):  
    print(x**0.5)  
  
    # запустили ожидание
```

```
sleep(3)

print('fun2 завершена')

def main():
    # создали конкурентную задачу из функции fun1
    task1 = create_task(fun1(4))

    # создали конкурентную задачу из функции fun2
    task2 = create_task(fun2(4))

    # запустили задачу task1
    task1

    # запустили task2
    task2

main()
```

Кошунство, скажете вы? Нет, я всего лишь честно выполняю рекомендацию великого и ужасного Гвидо ван Россума:

"Прищурьтесь и притворитесь, что ключевых слов `async` и `await` нет"

Звучит почти как: *"Наденьте зеленые очки и притворитесь, что стекляшки — это изумруды"*

Итак, в "прищуренной вселенной Гвидо":

Задачи — это "ракеты-носители" для конкурентного запуска "боеголовок"-функций.

А если вообще без задач?

Как это? Ну вот так, ни в какие задачи ничего не заворачивать, а просто эвейтнуть в `main()` сами корутины. А что, имеем право!

Пробуем:

Пример 4.3 (неудачный)

```
import asyncio
import time

async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')

async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')

async def main():
    await fun1(4)
    await fun2(4)
```

```
print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

Грусть-печаль... Снова 6 секунд как в давнем примере 1.1, ни разу не асинхронном. Боеголовка без ракеты взлетать отказалась.

Вывод:

В `asyncio.run` нужно передавать асинхронную функцию с эвейтами на **задачи**, а не на корутины. Иначе не взлетит. То есть работать-то будет, но сугубо последовательно, без всякой конкурентности.

Пара слов о конкурентности

С точки зрения разработчика и (особенно) пользователя конкурентное выполнение в асинхронных и многопоточных приложениях выглядит почти как параллельное. На самом деле никакого параллельного выполнения чего бы то ни было в питоне нет и быть не может. Кто не верит — погулите аббревиатуру GIL. Именно поэтому мы используем осторожное выражение "**конкурентное выполнение задач**" вместо "параллельное".

Нет, конечно, если очень хочется настоящего параллелизма, можно запустить несколько интерпретаторов python одновременно (библиотека multiprocessing фактически так и делает). Но без крайней нужды лучше такими вещами не заниматься, ибо издержки чаще всего будут непропорционально велики по сравнению с профитом.

А что есть "крайняя нужда"? Это приложения-числодробилки. В них подавляющая часть времени выполнения расходуется на операции процессора и обращения к памяти. Никакого

ленивого ожидания ответа от медленной периферии, только жесткий математический хардкор. В этом случае вас, конечно, не спасет ни изящная асинхронность, ни неуклюжая мультипоточность. К счастью, такие негуманные приложения в практике веб-разработки встречаются нечасто.

5. Асинхронные менеджеры контекста и настоящее асинхронное приложение

Пришло время написать на `asyncio` не тупой перебор неблокирующих слипов, а что-то выполняющее действительно осмысленную работу. Но прежде чем приступить, разберемся с асинхронными менеджерами контекста.

Если вы умеете работать с обычными менеджерами контекста, то без труда освоите и асинхронные. Тут используется знакомая конструкция `with`, только с префиксом `async`, и те же самые контекстные методы, только с буквой `a` в начале.

Пример 5.1

```
import asyncio

# имитация асинхронного соединения с некой периферией
async def get_conn(host, port):
    class Conn:
        async def put_data(self):
            print('Отправка данных...')
            await asyncio.sleep(2)
            print('Данные отправлены.')
```

```
async def get_data(self):
    print('Получение данных...')
    await asyncio.sleep(2)
    print('Данные получены.')

async def close(self):
    print('Завершение соединения...')
    await asyncio.sleep(2)
    print('Соединение завершено.')
```

```
print('Устанавливаем соединение...')
await asyncio.sleep(2)
print('Соединение установлено.')
return Conn()
```

```
class Connection:
```

```
# этот конструктор будет выполнен в заголовке with
```

```
def __init__(self, host, port):
```

```
    self.host = host
```

```
    self.port = port
```

```
# этот метод будет неявно выполнен при входе в with
```

```
async def __aenter__(self):
```

```
    self.conn = await get_conn(self.host, self.port)
```

```
    return self.conn
```

```
# этот метод будет неявно выполнен при выходе из with
```

```
async def __aexit__(self, exc_type, exc, tb):
```

```
    await self.conn.close()
```

```
async def main():
```

```
async with Connection('localhost', 9001) as conn:
    send_task = asyncio.create_task(conn.put_data())
    receive_task = asyncio.create_task(conn.get_data())

    # операции отправки и получения данных выполняем конкурентно
    await send_task
    await receive_task
```

```
asyncio.run(main())
```

Создавать свои асинхронные менеджеры контекста разработчику приложений приходится нечасто, а вот использовать готовые из асинхронных библиотек — постоянно. Поэтому нам полезно знать, что находится у них внутри.

Теперь, зная как работают асинхронные менеджеры контекста, можно написать ну очень полезное приложение, которое узнает погоду в разных городах при помощи библиотеки `aiohttp` и API-сервиса openweathermap.org:

Пример 5.2

```
import asyncio
import time
from aiohttp import ClientSession

async def get_weather(city):
    async with ClientSession() as session:
        url = f'http://api.openweathermap.org/data/2.5/weather'
        params = {'q': city, 'APPID': '2a4ff86f9aaa70041ec8e82db64abf56'}
```

```
    async with session.get(url=url, params=params) as response:
        weather_json = await response.json()
        print(f'{city}: {weather_json["weather"][0]["main"]}')

async def main(cities_):
    tasks = []
    for city in cities_:
        tasks.append(asyncio.create_task(get_weather(city)))

    for task in tasks:
        await task

cities = ['Moscow', 'St. Petersburg', 'Rostov-on-Don', 'Kaliningrad', 'Vladivostok',
         'Minsk', 'Beijing', 'Delhi', 'Istanbul', 'Tokyo', 'London', 'New York']

print(time.strftime('%X'))

asyncio.run(main(cities))

print(time.strftime('%X'))
```

"И говорит по радио товарищ Левитан: в Москве погода ясная, а в Лондоне — туман!" (с)
Е.Соев

Кстати, ключик к API дарю, пользуйтесь на здоровье.

Внимание! Если будет слишком много желающих потестить сервис с моим ключом, его могут временно заблокировать. В этом случае просто получите свой собственный, это

быстро и бесплатно.

Опрос 12-ти городов на моем канале 100Mb занимает доли секунды.

Обратите внимание, мы использовали два вложенных менеджера контекста: для сессии и для функции `get`. Так требует документация `aiohttp`, не будем с ней спорить.

Давайте попробуем реализовать тот же самый функционал, используя классическую синхронную библиотеку `requests` и сравним скорость:

Пример 5.3

```
import time
import requests

def get_weather(city):
    url = f'http://api.openweathermap.org/data/2.5/weather'
    params = {'q': city, 'APPID': '2a4ff86f9aaa70041ec8e82db64abf56'}

    weather_json = requests.get(url=url, params=params).json()
    print(f'{city}: {weather_json["weather"][0]["main"]}')

def main(cities_):
    for city in cities_:
        get_weather(city)

cities = ['Moscow', 'St. Petersburg', 'Rostov-on-Don', 'Kaliningrad', 'Vladivostok',
          'Minsk', 'Beijing', 'Delhi', 'Istanbul', 'Tokyo', 'London', 'New York']
```

```
print(time.strftime('%X'))

main(cities)

print(time.strftime('%X'))
```

Работает превосходно, но... В среднем занимает 2-3 секунды, то есть раз в 10 больше чем в асинхронном примере. Что и требовалось доказать.

А может ли асинхронная функция не просто что-то делать внутри себя (например, запрашивать и выводить в консоль погоду), но и **возвращать результат**? Ту же погоду, например, чтобы дальнейшей обработкой занималась функция верхнего уровня `main()` .

Нет ничего проще. Только в этом случае для группового запуска задач необходимо использовать уже не цикл с `await` , а функцию `asyncio.gather`

Давайте попробуем:

Пример 5.4

```
import asyncio
import time
from aiohttp import ClientSession

async def get_weather(city):
    async with ClientSession() as session:
        url = f'http://api.openweathermap.org/data/2.5/weather'
```



```

params = {'q': city, 'APPID': '2a4ff86f9aaa70041ec8e82db64abf56'}

async with session.get(url=url, params=params) as response:
    weather_json = await response.json()
    return f'{city}: {weather_json["weather"][0]["main"]}'

async def main(cities_):
    tasks = []
    for city in cities_:
        tasks.append(asyncio.create_task(get_weather(city)))

    results = await asyncio.gather(*tasks)

    for result in results:
        print(result)

cities = ['Moscow', 'St. Petersburg', 'Rostov-on-Don', 'Kaliningrad', 'Vladivostok',
          'Minsk', 'Beijing', 'Delhi', 'Istanbul', 'Tokyo', 'London', 'New York']

print(time.strftime('%X'))

asyncio.run(main(cities))

print(time.strftime('%X'))

```

Красиво получилось! Обратите внимание, мы использовали выражение со звездочкой `*tasks` для распаковки списка задач в аргументы функции `asyncio.gather`.

Пара слов о лишних сущностях

Кажется, я совершил невозможное. Настучал уже почти тысячу строк текста и ни разу не упомянул о **цикле событий**. Ну, почти ни разу. Один раз все-же упомянул: в примере 2.3 "как не надо делать". А между тем, в традиционных руководствах по `asyncio` этим самым циклом событий начинают душить несчастного читателя буквально с первой страницы. На самом деле цикл событий в наших программах присутствует, но он надежно скрыт от посторонних глаз высокоуровневыми конструкциями. До сих пор у нас не возникало в нем нужды, вот и я и не стал плодить лишних сущностей, руководствуясь принципом дорогого товарища Оккама.

Но рано или поздно жизнь заставит нас извлечь этот скелет из шкафа и рассмотреть его во всех подробностях.

Продолжение [здесь](#).

Теги: `async`, `await`, `asyncio`, асинхронность, асинхронное программирование, асинхронные задачи, асинхронные функции, асинхронный код

Хабы: Python, Программирование

Если эта публикация вас вдохновила и вы хотите поддержать автора — не стесняйтесь нажать на кнопку

Задонатить

Редакторский дайджест

Присылаем лучшие статьи раз в месяц



Электронная почта

**70****0**

Карма Рейтинг



Подписаться

Владимир Кириевский @vlakir

Python developer

Сайт Telegram

Комментарии 56

Reversaidx 25 мая 2022 в 00:13

Очень познавательно, спасибо!

+2 Ответить

Megakazbek 25 мая 2022 в 00:30

Лично у меня трудность возникает не в том, что это какой-то сложный механизм, а банально в том, что нужны специальные асинхронные функции, для большинства штук асинхронных библиотек нет, в итоге никакой пользы получить от этой фишки нельзя, а более практичным оказывается тупо multiprocessing.

+21 Ответить

Vindicar 25 мая 2022 в 01:24

Ну справедливости ради, из коробки есть механизмы, позволяющие легко завернуть поток/процесс в асинхронную задачу. Так что вполне можно спрятать один «неудобный» тип операций в мультипроцессинг, и пользоваться асинхронщиной для остального.

+9 Ответить

o  **satskov** 26 мая 2022 в 13:21 ^

для большинства штук асинхронных библиотек нет

Для какого большинства и каких штук?

Есть огромное кол-во библиотек на 99% кейсов с I/O с которыми можно столкнуться в повседневной работе (БД, http клиент/сервер, сокет)

Чего ещё не хватает web-разработчику для счастья?

♦ +2 Ответить 📖 ...

o  **linuxoid** 17 июн 2022 в 00:26 ^

А ещё иногда бывает не только веб... 🤖

♦ +1 Ответить 📖 ...

o  **DistortNeo** 25 мая 2022 в 02:26 ↗

В Linux доступ к файловой системе по определению не асинхронный

Уже нет. Недавно эта проблема была решена с помощью io_uring.

♦ +11 Ответить 📖 ...

o **funca** 25 мая 2022 в 10:37 ^

Там есть ещё древнющий aio (aka POSIX asynchronous I/O) - в питоне через caio / aiofile.

♦ +1 Ответить 📖 ...

o 404 **amarao** 25 мая 2022 в 09:39

В Linux доступ к файловой системе по определению не асинхронный,

В линуксе много разных методов. Вы говорите про posix, который Линукс (большей частью) поддерживает. Про io_uring выше комментариев.

♦ +3 Ответить

○  **shirvash** 25 мая 2022 в 10:04

Было бы здорово добавить после функций то, что выводилось в консоль. Особенно важно видеть время выполнения функций.

♦ +4 Ответить

○  **vlakir** 25 мая 2022 в 15:05

В учебном материале нецелесообразно, я считаю. При знакомстве с конкурентными вычислениями важно видеть процесс в динамике: что отработало быстро, что притормозилось, а что выполнилось как-бы одновременно. Если расставить повсюду временные метки и выводить в консоль тонны текста, то вся это мерцающая красота исчезнет) Ну а так да, в реальном приложении организуйте логирование и спокойно анализируйте бенчмарки постфактум.

♦ 0 Ответить

○  **shirvash** 26 мая 2022 в 17:02

При чтении статьи возникло ощущение радио, потому что после кода идет анализ и обсуждение результата, который я не видел. Поэтому пошел весь код сам запускать и смотреть что там выводилось, чтобы лучше понимать контекст.

А так, статья очень полезная. Спасибо большое, жду продолжение!

♦ +1 Ответить

o  **vlakir** 26 мая 2022 в 23:27 ^


Вот именно этого я и добивался) Не запуская код невозможно чему-то научиться. Удачи!

♦ 0 Ответить

o 😊 **Surgeon76** 25 мая 2022 в 10:42

Отличная статья! Прочёл с удовольствием!

♦ 0 Ответить

o  **danilovmy** 25 мая 2022 в 10:53

зануда_mode = on

Пример 2.3

используются три функции библиотеки `asyncio` вместо одной `run` в предыдущем примере.

в примере 2.2 используются `asyncio.create_task`, `asyncio.run`, `asyncio.sleep`. Вероятно, подразумевалось, что все убрано в функции и в главной функции вызывается только одна функция `asyncio`.

зануда_mode = off

вопрос по тексту, можно ли в Пример 5.2?

```
async def main(cities_):
    for city in cities_:
        await asyncio.create_task(get_weather(city))
```

и если да, то можно ли:

```
async def main(cities_):
    yield from (await asyncio.create_task(get_weather(city)) for city in cities)
```

но, поскольку, возвращаем "Task" зачем тогда async в main

```
def main(cities_):
    yield from (asyncio.create_task(get_weather(city)) for city in cities)
```

Или я что-то упустил?

0 Ответить

Healer 25 мая 2022 в 11:42

зануда_mode = on Код приведён в тексте, можно и попробовать самому. зануда_mode = off

Первое выполнится синхронно. Второе и третье не выполнится "yield from" not allowed in an async function, "AsyncGenerator[None, None]" is not iterable

0 Ответить

danilovmy 25 мая 2022 в 14:01

попробовал сам. То, что я хотел, не работает без gather

```
async def main(cities_):
    await asyncio.gather(*(asyncio.create_task(get_weather(city)) for city in
```

жаль, что gather не жрет нераспакованные генераторы.

YAKOROLEVAZAMKA 25 мая 2022 в 11:16

Статья, вероятно, хорошая (давно хотел погрузиться в асинхронность), но после запуска примера 2.2 я действительно получил сюрприз:

```
print(time.strftime('%X'))
```

```
asyncio.run(main())
```

```
print(time.strftime('%X'))
```

11:12:45

```
-----
RuntimeError                                Traceback (most recent call last)
C:\Users\SERGEY~1\SMY\AppData\Local\Temp\ipykernel_4008\1186977821.py in <module>
    21 print(time.strftime('%X'))
    22
--> 23 asyncio.run(main())
    24
    25 print(time.strftime('%X'))

C:\Anaconda3\lib\asyncio\runners.py in run(main, debug)
    31     """
    32     if events._get_running_loop() is not None:
--> 33         raise RuntimeError(
    34             "asyncio.run() cannot be called from a running event loop")
    35
```

```
RuntimeError: asyncio.run() cannot be called from a running event loop
```

А после запуска примера 2.3 было принято волевое решение дальше не читать (очень жаль):


```

async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')

async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')

print(time.strftime('%X'))

loop = asyncio.get_event_loop()
task1 = loop.create_task(fun1(4))
task2 = loop.create_task(fun2(4))
loop.run_until_complete(asyncio.wait([task1, task2]))

print(time.strftime('%X'))

```

11:12:34

```

-----
RuntimeError                                Traceback (most recent call last)
C:\Users\SERGEY~1\SMY\AppData\Local\Temp\ipykernel_4008\1092300447.py in <module>
     16 task1 = loop.create_task(fun1(4))
     17 task2 = loop.create_task(fun2(4))
--> 18 loop.run_until_complete(asyncio.wait([task1, task2]))

```

0 Ответить

vlakir 25 мая 2022 в 11:20

У вас какая версия python? Попробуйте 3.10


0 Ответить

 **YAKOROLEVAZAMKA** 25 мая 2022 в 11:29

Python 3.9.7

Но это на собственном ноуте, а в проде везде не выше 3.6, поэтому, к сожалению, статья про версию 3.10+ для меня мало актуальна) спасибо за ответ

0 Ответить

 **vlakir** 25 мая 2022 в 11:31

Ну, тут уж ничего не поделаешь. Обратная совместимость - болезненное место asyncio. Я рассчитываю, что статья проживет долго, поэтому использую конструкции, рекомендованные для самой новой доступной версии питона.

Учиться вообще лучше на актуальном, мне кажется. Когда набьешь руку, разобраться в старых конструкциях уже не составляет проблемы.

0 Ответить

 **worldmind** 25 мая 2022 в 12:30

руenv вам в руки

+2 Ответить

 **ShashkovS** 25 мая 2022 в 11:30

asyncio.run появилась в python 3.7 (см. docs.python.org/3/library/asyncio-task.html#asyncio.run)
До этого нужно было писать

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Другая возможная проблема — это использование IPython ≥ 7.0 или соответствующего Jupyter. Там уже запущен свой event loop от ipython'a и нельзя создать новый. Соответственно нужно вызывать первую функцию сразу с await'ом

```
await main()
```

А чтобы получить текущий event loop —

```
loop = asyncio.get_running_loop()
```

♦ +2 Ответить

◉  **YAKOROLEVAZAMKA** 25 мая 2022 в 17:26 ^

Спасибо, в первом примере помог `await main()`, во втором -

```
import nest_asyncio
nest_asyncio.apply()
```

@EvilsInterrupt

К сожалению да, Windows :(`set_event_loop_policy` не помогло

♦ 0 Ответить

◉  **EvilsInterrupt** 25 мая 2022 в 17:03 ^

Это не вина автора вы используете Windows , а на эту тему есть общеизвестный баг .

Попробуйте применить:


```
asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
```

0 Ответить

o  **maksim_R** 25 мая 2022 в 22:28 ^

Anaconda (Spyder) сама по себе является асинхронной программой на Питоне. Поэтому внутри неё не вызвать ещё более асинхронный код. О чем и говорит ошибка: cannot be called from running event loop.

0 Ответить

o  **TheCellMan** 26 мая 2022 в 04:57 ^

А вы случайно не в jupyter notebook его запускали?

0 Ответить

o  **YAKOROLEVAZAMKA** 26 мая 2022 в 16:18 ^

конечно в юпитере)

если бы не юпитер, я бы может никогда и программировать не стал)

в общем в моём комментарии выше я нашел как убрать ошибки (и даже добиться правильной работы кода), посему статью буду читать дальше)

-1 Ответить

o  **ShashkovS** 25 мая 2022 в 11:30

Перенесено в habr.com/ru/post/667630/#comment_24376224

0 Ответить

aelih 25 мая 2022 в 11:50

Автор, вот встретились бы лично - пожал бы руку!!! Только недавно с этим асинх боролся и тут такой подарок! Лучи добра тебе!:)

0 Ответить

stgunholy 25 мая 2022 в 15:16

Суперская статья! Действительно самое простое объяснение из тех что я видел

+1 Ответить

avshkol 25 мая 2022 в 15:44

Не понял логики разработчиков библиотеки: зачем нужно каждую функцию отправлять в `asyncio.create_task`, ведь функция и так объявлена как `async` - увидели, что запускаем `async` - функцию, и выполняйте `create_task` и всё, что нужно, но под капотом...?

0 Ответить

vlakir 25 мая 2022 в 15:53

```
async def my_breakfast():  
    print('поставил варить яйцо')  
  
    await asyncio.sleep(300)  
  
    print('съел яйцо')  
  
#
```

```
async def your_breakfast():  
  
    print('поставил варить яйцо')  
  
    await asyncio.sleep(300)  
  
    print('съел яйцо')
```

Если бы создатели asyncio согласились бы с вашим предложением, мы бы с вами позавтракали очень быстро, но сырыми яйцами, к сожалению...

0 Ответить

avshkol 25 мая 2022 в 16:00

Увидев, что мы запускаем асинх- функции fun1 и fun2, они бы сами обернули их в:


```
task1 = asyncio.create_task(fun1(4))  
task2 = asyncio.create_task(fun2(4))  
  
await task1  
await task2
```

0 Ответить

vlakir 25 мая 2022 в 16:09

То есть, если мы делаем `await f()`, значит обозначаем точку переключения, а если просто `f()`, то неявно оборачиваем корутину в задачу? Ну, в принципе такой синтаксический сахар наверно имеет право на жизнь. Становитесь контрибьютором питона и попробуйте реализовать)

0 Ответить

o **funca** 25 мая 2022 в 20:28  

В JavaScript `async / await` сделаны жадными как `Promise`. При вызове `async` функции автоматически создается задача и отправляется в очередь на исполнение в `event loop`. `await`, в свою очередь, просто ждёт результат.

В питоне асинхронщину задизайнили иначе - лениво.

Вызов `async` функции возвращает объект - корутину, - которая ни чего не делает.

`asyncio.run()` создаёт `event loop`, запускает (корневую) корутину и блокирует поток до получения результата.

`await` запускает корутину изнутри другой корутины в текущем `event loop` и ждёт результат.

Для запуска корутины без ожидания (как это делает `Promise`) используется `asyncio.create_task(coro)`. Либо `asyncio.gather(*aws)`, если надо запустить сразу несколько. Нужно только следить, чтобы ссылка на возвращаемое значение сохранялась до конца вычисления, иначе его пожрет GC и все оборвется на самом интересном месте (промис бы отработал до конца не смотря ни на что).

В JS только один `event loop`, поэтому было вполне разумно закопать его внутрь `promise / async / await` как деталь реализации, упростив работу прикладному программисту. В питоне отзеркалили более ранний вариант корутин на генераторах, дали возможность использовать разные `event loop` и выставили все кишки наружу.

 +6 Ответить  ...


o **funca** 25 мая 2022 в 21:31  

Feature это адаптер для вычислений, через который с ними взаимодействует `event loop`. По завершении вычисления в объекте `Feature` сохраняется его результат (или брошенное на произвол судьбы исключение). В отличие от джаваскриптовых `Promise`, которые выражают отложенное значение, фичи это процессы вычисления (т.е. их можно принудительно обрывать методом `.cancel()`).

Task это адаптер для корутин, представляющий их в виде Feature для отправки в event loop. Промежуточные значения, генерируемые корутиной, отправляются в event loop для дальнейшего вычисления. Результат последнего сохраняется в качестве результата всей задачи.

Coroutine - можно рассматривать как функцию с многократными выходами и входами. Или по-простому, генератор последовательности значений, где в точках возврата промежуточных значений используется слово await (раньше было yield, и настоящие генераторы asyncio тоже до сих пор поддерживает). Разница в том, что через yield из генератора можно вернуть любое значение, а через await из async функции только awaitable (т.е feature, coro или task) - защита event loop от дурака.

◆ +1 Ответить

○  **svrcom** 25 мая 2022 в 16:41

Столько комментариев, а про twisted даже никто и не вспомнил. Хотя он с в сравнении с asyncio как мерседес с запорожцем. По поводу введения в асинхронное программирование есть отличная статья <https://glyph.twistedmatrix.com/2014/02/unyielding.html> . Async/await + asyncio хорош тем, что человек ранее писавший только синхронный код легко его может начать писать асинхронный. Но вот дальше будут проблемы с тем, что большинство не понимает в каком контексте этот асинхронный код запускается и что сломается если такой обработчик заблокируется и/или в какой момент передается/возвращается управление в event_loop.

Ну и в отличие от twisted asyncio это голый event-loop без асинхронной реализации библиотек, протоколов и тд.

◆ +1 Ответить

○ **funca** 25 мая 2022 в 22:22 ^

asyncio писали с оглядкой на twisted, чтобы затащить минимальную реализацию низкоуровневых интерфейсов для подобных фреймворков в стандартную библиотеку. Кодить прикладной уровень это вроде как задача для разработчиков сторонних пакетов.

0 Ответить

EvilsInterrupt 25 мая 2022 в 17:11

@vlakir читаю ваши слова:

Сплошь и рядом корутиной называют саму функцию, содержащую `await`. Строго говоря, это неправильно.

Возможно я что-то неверно понимаю, но офиц. документация в разделе `awaitables` говорит что

- a *coroutine function*: an `async def` function;

А вот возвращаемый результат именуют:

- a *coroutine object*: an object returned by calling a *coroutine function*.

Да, они помечают это словами "In this documentation " , но мне кажется, что назвать корутиной саму функцию не такая уж и неправильная затея.

0 Ответить

Buchachalo 25 мая 2022 в 17:15

Ну так автор и написал что с пациентом нечего не случится если обозвать и функцию и ответ корутиной.

0 Ответить

alekssamos 25 мая 2022 в 21:35

Итак, Пример 2.2.

Можете объяснить, зачем тогда придумали функцию `asyncio gather`?

Типа когда через обычный вызов,

то сначала выполняется одна строка кода,

пока она не завершится, вторая не выполнится,

а начнёт только после завершения первой?

А при `gather` эти две строки выполнятся сразу за вдвое меньший промежуток времени (если и у одной, и у второй есть `sleep(3)` и перейдёт дальше)?

0 Ответить

svpcorn 26 мая 2022 в 12:38

Вот поэтому для реальных вещей используется `twisted`, так как он не скрывает асинхронность за магией `async/await` и не обещает пользователю, что писать асинхронный код ничуть не сложнее, чем синхронный.

0 Ответить

Vindicar 31 мая 2022 в 09:09

Попробую объяснить это так. Если вы делаете вызов

```
await fun1(42)
await fun2(42)
```

То получается такая последовательность действий:

- Создать объект корутины `fun1()`
- Запланировать исполнение объекта корутины `fun1()` в рабочем цикле
- Вернуть управление в рабочий цикл до завершения объекта корутины `fun1()`
- Создать объект корутины `fun2()`
- Запланировать исполнение объекта корутины `fun2()` в рабочем цикле
- Вернуть управление в рабочий цикл до завершения объекта корутины `fun2()`

Т.е. пока не закончится выполнение `fun1()`, выполнение `fun2()` даже не начнётся – они выполняются строго последовательно. Если же использовать `gather()`, последовательность действий будет иная:

- Создать объект корутины `fun1()`
- Создать объект корутины `fun2()`
- Запланировать исполнение объекта корутины `fun1()` в рабочем цикле
- Запланировать исполнение объекта корутины `fun2()` в рабочем цикле
- Вернуть управление в рабочий цикл до завершения обоих объектов

Т.е. выполнение `fun2()` начинается, как только `fun1()` вернёт управление в цикл, в нашем случае с помощью `sleep()`. После этого `fun1()` и `fun2()` исполняются конкурентно.

А пример с `create_task()` по сути делает то же самое, что и `gather()` - т.е. я бы сказал что `gather()` это просто удобная обёртка.

0 Ответить

Denis_Sk 25 мая 2022 в 22:28

Статья просто супер. С нетерпением жду продолжения.

0 Ответить

un1t 28 мая 2022 в 11:39

```
async with aiohttp.ClientSession() as session:
    async with session.get(url) as response:
        weather_json = await response.json()
```

Я пробовал писать небольшое асинхронное приложение. Был неприятно удивлен количеством уровней вложенности. Чтобы просто HTTP запрос сделать уже 2 уровня вложенности получается, запрос к базе тоже оборачивается в `"async with async_session()"`.

Получается неудобно и некрасиво. В JS с этим гораздо приятнее работать

```
const response = await fetch('/movies');
const movies = await response.json();
```

0 Ответить

DistortNeo 28 мая 2022 в 14:07

Вам ничто не мешает написать нужные обёртки. А Python просто даёт больше гибкости.

0 Ответить

healfy 30 мая 2022 в 16:43

В реальных приложениях обычно сессию создают 1 раз, а потом уже используют не для вызовов, плюс вложенность из-за контекстных менеджеров, при желании можно писать строго, но вопрос зачем

0 Ответить

mari_an_shum 31 мая 2022 в 09:37

Хорошая статья для теоретического введения в проблематику. К сожалению, после ее прочтения начинающий не сможет самостоятельно использовать описанные конструкции.

0 Ответить

orlovdl 31 мая 2022 в 12:33

`aiofiles` - построен полностью на тредпуле, это не мешает ему быть конкурентным.

`aiofile` - использует `caio` под капотом, который для linux использует системные вызовы ядра для асинхронной работы с файлами, или тредпул на си, для мака тредпул на си или реализацию на

python тредах для windows.

По поводу баз данных, тоже не правда, `sqlalchemy>1.4` тоже не использует никаких тредов если только драйвер не использует их (`aiosqlite` построен на потоках, иначе никак)

На самом деле никакого параллельного выполнения чего бы то ни было в питоне нет и быть не может. Кто не верит — погулите аббревиатуру GIL.

Это тоже довольно популярное заблуждение, тот-же упомянутый выше `saio` отпускает GIL когда читает из фалов, ему его держать не нужно, просто отдал в ядро системный вызов и потом как на `eventfd` триггернется `epoll` приходи за результатом.

GIL не является проблемой в python, то что вы не можете его выключить для python кода, это правда, если вам нужна, например, математика многоточечная, берите `numpy/numba/cython`, там или GIL уже отключен при `cru-bound` вызовах, или можно его отключить (`"with nogil:"` в `Cython`).

Множество стандартных функций интерпретатора отпускает GIL, та-же работа с файлами, `os.pread` например, да и вообще поищите в исходниках интерпретатора `Py_BEGIN_ALLOW_THREADS` все что в теле под этим макросом, выполняется конкурентно безо всякого GIL, и да, `sqlite` в их числе.

◆ +1 Ответить

○  **art1z** 2 июн 2022 в 09:19

После 20 лет в JavaScript так приятно видеть что наконец-то Питон его догнал (шутка).

Микротаски и `Promise.race`, `Promise.allSettled` на очереди.

А на деле, большое спасибо за статью, с футурами чуть яснее стало после вашей аналогии, ну и `event_loop` без внимания таки оставить не удастся.

Вчера только бодался в `pytest` с ошибками.

got Future <Future pending cb=[Protocol._on_waiter_completed()]> attached to a different loop

◆ 0 Ответить

○  **chewarer** 2 июн 2022 в 17:17

Пытаюсь переписать со "старомодного" способа на `asyncio.run()`. Но не нравится что получается даже более громоздко.

Вместо:

```
loop = get_or_create_eventloop()
validation_status = loop.run_until_complete(asyncio.gather(
    is_access_token_valid(access_token),
    is_id_token_valid(id_token),
))
```

Получается:

```
async def main():
    return await asyncio.gather(
        asyncio.create_task(is_access_token_valid(access_token)),
        asyncio.create_task(is_id_token_valid(id_token)),
    )

validation_status = asyncio.run(main())
```

Хотелось бы написать так, но это не работает:

```
validation_status = asyncio.run(
    asyncio.gather(
        asyncio.create_task(is_access_token_valid(access_token)),
        asyncio.create_task(is_id_token_valid(id_token)),
    )
)
```

○  **KorsarD** 29 авг 2022 в 21:48

Благодаря этой статье, получил главный ответ на свой вопрос. От того что я написал асинхронную функцию, она все равно будет выполняться синхронно. Асинхронность проявляется лишь тогда, когда таких функций несколько, когда создается конкуренция нескольких функций. Раньше мне казалось, что после `await` должно начаться какая то магия, она есть, но для не для самой функции, а для другой

◆ 0 Ответить  ...

○  **Gim6626** 20 сен 2022 в 15:34 

А разве при использовании `gather` нужно создание задач? Вот такой код работает примерно за 3 секунды и без `create_task` :

```
import asyncio
import datetime

async def fun1():
    print(1)
    await asyncio.sleep(3)

async def fun2():
    print(2)
    await asyncio.sleep(3)

async def main():
    await asyncio.gather(fun1(), fun2())
```

```
begin = datetime.datetime.now()
asyncio.run(main())
end = datetime.datetime.now()
print(end - begin)
```

0 Ответить

o  **nikolaysalionov** 27 мар 2023 в 10:07

Спасибо, все просто и хорошо разложили.

У меня вопрос по правильному подходу к обработке ошибок, например если в последнем примере с получением погоды для списка городов. Как обработать ошибку 5xx и продолжить цикл?

0 Ответить

o  **zolandv** 19 ноя 2023 в 17:43

Хороший подход, понравилась Парадигма избегать раньше времени погружения в поиск Цикла. Отвлекает от Сути. Bravo!

0 Ответить

o  **o_O_Tync** 16 дек 2023 в 17:53

Екклесиаст! 👍

0 Ответить

Зарегистрируйтесь на Хабре, чтобы оставить комментарий

Публикации

ЛУЧШИЕ ЗА СУТКИ

ПОХОЖИЕ



atomlib 19 часов назад

48 лет вместе с Zilog Z80

Простой 10 мин 11K

+95 32 25 +25



Shkaff 9 часов назад

На гребне гравитационной волны: космический детектор LISA

Средний 12 мин 1.8K

+38 12 11 +11



Bright_Translate 13 часов назад

Поиск по коду — это сложно

Простой 5 мин 3K

Обзор

Перевод

 +31  18  1 +1



neskuchan 10 часов назад

Найти работу в IT: миссия 2024

 Простой  7 мин  3.1K

Обзор

 +26  17  6 +6



Grigory_Otrepyev 9 часов назад

Запуск ракеты Ангара — неделю спустя

 Сложный  6 мин  9.1K

Мнение

 +23  9  20 +20



kksen 15 часов назад

Как запустить IT-подкаст: прошли этот путь и расскажем обо всех подводных камнях

 7 мин  2K

Кейс

 +22  16  2 +2



dalerank 4 часа назад

Я скучаю по механикам из старых игр

🕒 13 мин 👁 3.1K

💎 +21 📖 11 💬 42 +42



MarkNest 11 часов назад

Стала доступнее веб-страничка, которая строит спектр отражения и пропускания света слоистой средой

🕒 2 мин 👁 1.3K

💎 +15 📖 13 💬 2 +2



ibessonov 6 часов назад

Миллер, Рабин, вектор

🔒 Сложный 🕒 16 мин 👁 977

💎 +13 📖 15 💬 6 +6



GlobalSign_admin 5 часов назад

Фальшивые криптокошельки в официальном каталоге Ubuntu — индикатор более серьёзной угрозы

🕒 3 мин 👁 1.6K

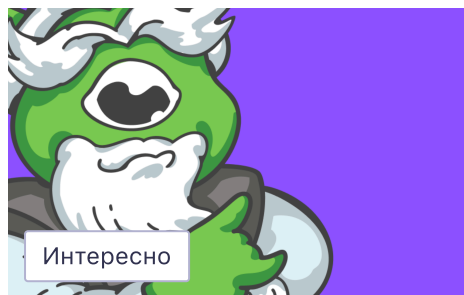
💎 +12 📖 6 💬 6 +6

Любят ли на Хабре пет-проекты?

Опрос 

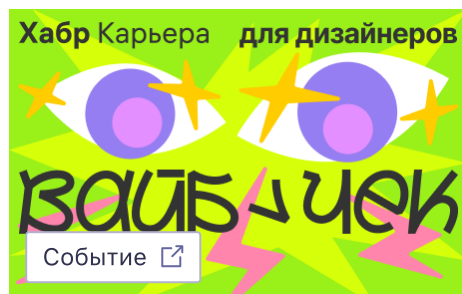
Показать еще

МИНУТОЧКУ ВНИМАНИЯ



Интересно

Глупым вопросам и ошибкам —
быть! IT-менторство на ХК



Событие 

Дизайнеры, узнайте какая
компания подходит вам по
вайбам



Интересно

IT-события, которые ты ищешь,
тоже ищут тебя

ЗАКАЗЫ

Разработать бота для Reddit

60000 руб./за проект · 1 отклик · 58 просмотров

Разработать telegram-бота с помощью web up

60000 руб./за проект · 6 откликов · 74 просмотра

Парсер выгрузки поиска с сайта закупки гов

5000 руб./за проект · 15 откликов · 64 просмотра

Разработка модуля для маркетплейса на cs- cart multi-vendor

40000 руб./за проект · 5 откликов · 36 просмотров

Телеграмм бот - игра тапалка (нужно понимание в PVP боях)

150000 руб./за проект · 13 откликов · 92 просмотра

Больше заказов на Хабр Фрилансе

Ваш аккаунт

- Войти
- Регистрация

Разделы

- Статьи
- Новости
- Хабы
- Компании
- Авторы
- Песочница

Информация

- Устройство сайта
- Для авторов
- Для компаний
- Документы
- Соглашение
- Конфиденциальность

Услуги

- Корпоративный блог
- Медийная реклама
- Нативные проекты
- Образовательные программы
- Стартапам

