Асинхронный python без головной боли (часть 1)

**14 мин**

**286K**

[Python\*](https://habr.com/ru/hubs/python/)[Программирование\*](https://habr.com/ru/hubs/programming/)

Туториал

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.
3. Официальная документация asyncio (разумеется, исчерпывающая и прекрасно организованная) рассчитана скорее на создателей фреймворков, чем на разработчиков пользовательских приложений. Там столько всего — глаза разбегаются. А между тем: *"Вам нужно знать всего около семи функций для использования asyncio" (c) 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

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)  
 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:

* выполнилась быстрая часть функции fun1
* fun1 сказала интерпретатору *"иди дальше, я посплю 3 секунды"*
* выполнилась 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 отрапортовала нам об успешном завершении
* то же самое сделала функция fun2

Замените "посплю" 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.runиспользуются 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 комментах. В Linux, начиная с ядра 5.1, есть полноценный асинхронный интерфейс io\_uring и это прекрасно. Кому интересны детали, рекомендую пройти**[**вот сюда**](https://habr.com/ru/post/589389/)**.**

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 многой 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()

Кощунство, 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](http://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!" (c) 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***](https://habr.com/ru/post/671798/)***.***

**Теги:**

* [async](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5basync%5d)
* [await](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bawait%5d)
* [asyncio](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5basyncio%5d)
* [асинхронность](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C%5d)
* [асинхронное программирование](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5+%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%5d)
* [асинхронные задачи](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5+%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%B8%5d)
* [асинхронные функции](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5+%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8%5d)
* [асинхронный код](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9+%D0%BA%D0%BE%D0%B4%5d)

**Хабы:**

* [Python](https://habr.com/ru/hubs/python/)
* [Программирование](https://habr.com/ru/hubs/programming/)

Асинхронный python без головной боли (часть 2)

**9 мин**

**57K**

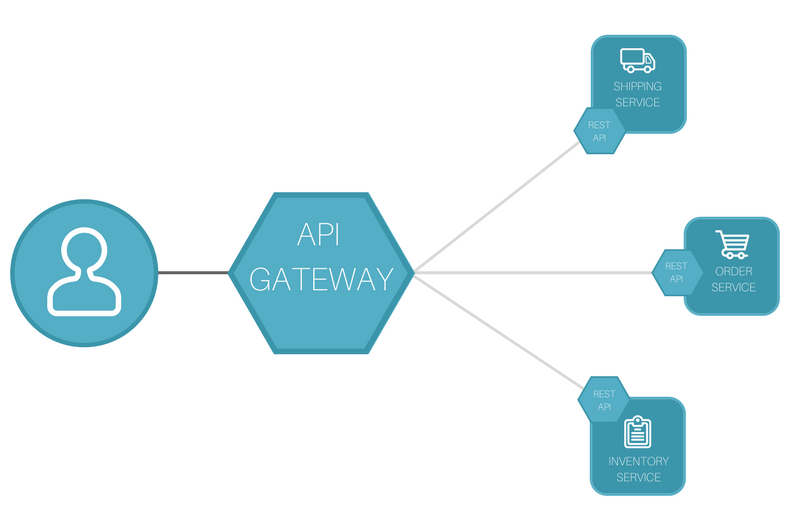
[Python\*](https://habr.com/ru/hubs/python/)[Программирование\*](https://habr.com/ru/hubs/programming/)

Туториал

***Это продолжение цикла статей про***asyncio***. Начало***[***здесь***](https://habr.com/ru/post/667630/)***.***

6. Веб-сервер aiohttp и другие жители асинхронного мира

Продолжаем готовить asyncio. Теперь мы уже знаем достаточно, чтобы написать модный асинхронный микросервис. Реализуем известный архитектурный паттерн "API-шлюз". Это довольно простая штука. По запросу на свой API-интерфейс приложение собирает данные из других API, обрабатывает и возвращает результат пользователю. При этом пользователь знает только одну точку входа, а все внутренние подробности (кто куда и зачем сходил) от него скрыты.



В предыдущей главе мы научились запрашивать погоду у сервиса [api.openweathermap.org](https://openweathermap.org/api). Давайте его слегка ~~украдем~~ импортозаместим. Вернее сказать, русифицируем. Пусть пользователь нашего сервиса передает название города на русском языке в параметрах GET-запроса и получает ответ в виде json опять-таки на великом и могучем. А откуда мы взяли информацию о погоде, пользователю знать не положено. Может у нас собственные метеостанции в каждой деревне стоят, поди проверь.

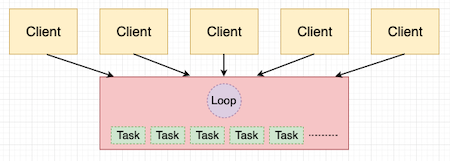
Мы уже освоили http-клиента библиотеки aiohttp, с помощью которого можно обращаться к внешним API. Оказывается, в этой же библиотеке есть и все необходимое для создания полноценного http-сервера. Для начала напишем просто зеркальный прокси:

Пример 6.1

**import** asyncio  
**import** json  
**from** aiohttp **import** ClientSession, web  
  
  
**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()  
 **try**:  
 **return** weather\_json["weather"][0]["main"]  
 **except** KeyError:  
 **return** 'Нет данных'  
  
  
**async** **def** handle(request):  
 city = request.rel\_url.query['city']  
 weather = **await** get\_weather(city)  
 result = {'city': city, 'weather': weather}  
  
 **return** web.Response(text=json.dumps(result, ensure\_ascii=**False**))  
  
  
**async** **def** main():  
 app = web.Application()  
 app.add\_routes([web.get('/weather', handle)])  
 runner = web.AppRunner(app)  
 **await** runner.setup()  
 site = web.TCPSite(runner, 'localhost', 8080)  
 **await** site.start()  
  
 **while** **True**:  
 **await** asyncio.sleep(3600)  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(main())

В асинхронной функции get\_weather ничего нового нет, мы ее лишь слегка причесали, чтобы запрос погоды для несуществующего города не приводил к трагическим последствиям для всего нашего приложения. За обработку запроса отвечает функция handle ("ручка" на сленге бэкендеров). Из запроса извлекается параметр city и передается в get\_weather. Далее формируется результирующий ответ в виде json. Адрес нашего сервиса и тип запроса задается вapp.add\_routes.

Стоп! А где же задачи? Не переживайте. Когда мы имеем дело с асинхронными веб-фреймворками (а aiohttp — это именно фреймворк, хоть и супер-минималистический), вся работа по созданию и запуску задач asyncio происходит у фреймворка "под капотом". Нам, как разработчикам, теперь нет нужды беспокоится об этих низменных деталях. Приложение мирно спит в бесконечном неблокирующем цикле, пока не придет запрос GET на определенный URL. Как только это произойдет, отработает логика в ручке. И снова баю-бай до следующего запроса. Но если первый запрос еще не успел обработаться, как поступил следующий, фреймворк отработает его в отдельной задаче, не дожидаясь (по возможности) окончания обработки первого. В этом сама суть асинхронности.



Заходим браузером на адрес: localhost:8080/weather?city=Sochi и получаем симпатичный json:

{"city": "Sochi", "weather": "Clouds"}

Кстати, если вы всерьез решили заняться бэкенд-разработкой, одним браузером вам никак не обойтись. Потребуется инструмент, позволяющий залезать вглубь HTTP. Стандарт де-факто здесь [Postman](https://www.postman.com/), но в природе существуют и альтернативные решения.

Скелет нашего приложения готов, теперь начинаем наращивать на нем мышцы. Воспользуемся бесплатным API переводчика [libretranslate.de](https://libretranslate.de/):

Пример 6.2

**import** asyncio  
**import** json  
**from** aiohttp **import** ClientSession, web  
  
  
**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()  
 **try**:  
 **return** weather\_json["weather"][0]["main"]  
 **except** KeyError:  
 **return** 'Нет данных'  
  
  
**async** **def** get\_translation(text, source, target):  
 **async** **with** ClientSession() **as** session:  
 url = 'https://libretranslate.de/translate'  
  
 data = {'q': text, 'source': source, 'target': target, 'format': 'text'}  
  
 **async** **with** session.post(url, json=data) **as** response:  
 translate\_json = **await** response.json()  
  
 **try**:  
 **return** translate\_json['translatedText']  
 **except** KeyError:  
 **return** text  
  
  
**async** **def** handle(request):  
 city\_ru = request.rel\_url.query['city']  
 city\_en = **await** get\_translation(city\_ru, 'ru', 'en')  
  
 weather\_en = **await** get\_weather(city\_en)  
 weather\_ru = **await** get\_translation(weather\_en, 'en', 'ru')  
  
 result = {'city': city\_ru, 'weather': weather\_ru}  
  
 **return** web.Response(text=json.dumps(result, ensure\_ascii=**False**))  
  
  
**async** **def** main():  
 app = web.Application()  
 app.add\_routes([web.get('/weather', handle)])  
 runner = web.AppRunner(app)  
 **await** runner.setup()  
 site = web.TCPSite(runner, 'localhost', 8080)  
 **await** site.start()  
  
 **while** **True**:  
 **await** asyncio.sleep(3600)  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(main())

Теперь в ручке дважды вызывается асинхронная функция get\_translation (обратите внимание, на этот раз мы передаем параметры внешнему сервису в виде json через тело запроса POST) и, вуаля:

localhost:8080/weather?city=Сочи

{"city": "Сочи", "weather": "Облака"}

*Над всей Испанией безоблачное небо, а в деревне Гадюкино опять идут дожди...*

Но что это за микросервис без логгера? Однако использовать в насквозь асинхронном приложении привычную синхронную (а значит блокирующую) библиотеку logging — это не наш путь. Воспользуемся правильной асинхронной библиотекой логгирования aiologger:

Пример 6.3

**import** asyncio  
**import** json  
**from** aiohttp **import** ClientSession, web  
**from** aiologger.loggers.json **import** JsonLogger  
  
  
logger = JsonLogger.with\_default\_handlers(  
 level='DEBUG',  
 serializer\_kwargs={'ensure\_ascii': **False**},  
 )  
  
  
**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()  
 **try**:  
 **return** weather\_json["weather"][0]["main"]  
 **except** KeyError:  
 **return** 'Нет данных'  
  
  
**async** **def** get\_translation(text, source, target):  
 **await** logger.info(f'Поступил запрос на на перевод слова: {text}')  
  
 **async** **with** ClientSession() **as** session:  
 url = 'https://libretranslate.de/translate'  
  
 data = {'q': text, 'source': source, 'target': target, 'format': 'text'}  
  
 **async** **with** session.post(url, json=data) **as** response:  
 translate\_json = **await** response.json()  
  
 **try**:  
 **return** translate\_json['translatedText']  
 **except** KeyError:  
 logger.error(f'Невозможно получить перевод для слова: {text}')  
 **return** text  
  
  
**async** **def** handle(request):  
 city\_ru = request.rel\_url.query['city']  
  
 **await** logger.info(f'Поступил запрос на город: {city\_ru}')  
  
 city\_en = **await** get\_translation(city\_ru, 'ru', 'en')  
 weather\_en = **await** get\_weather(city\_en)  
 weather\_ru = **await** get\_translation(weather\_en, 'en', 'ru')  
  
 result = {'city': city\_ru, 'weather': weather\_ru}  
  
 **return** web.Response(text=json.dumps(result, ensure\_ascii=**False**))  
  
  
**async** **def** main():  
 app = web.Application()  
 app.add\_routes([web.get('/weather', handle)])  
 runner = web.AppRunner(app)  
 **await** runner.setup()  
 site = web.TCPSite(runner, 'localhost', 8080)  
 **await** site.start()  
  
 **while** **True**:  
 **await** asyncio.sleep(3600)  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(main())

Обратите внимание, асинхронный логгер ни на миллисекунду не задержит работу нашего приложения в целом. Он будет использовать паузы, пока мы ожидаем запроса от пользователя или ответа от внешнего сервиса. Поэтому, если мы будем наблюдать за его работой под серьезной нагрузкой, может обнаружиться, что он заметно отстает от реального времени. Но в конечном итоге все что должно будет выведено в лог и ни один символ не потеряется.

Ну а теперь вишенка на торте — асинхронный доступ к базе данных. Предположим, в процессе работы нашего приложения нам надо что-то писать в БД, ну, например, сохранять поступившие запросы (ничего более умного мне как-то в голову не пришло). Самая простая БД в мире, как известно, — это SQLite. И для нее, к счастью, есть асинхронный драйвер aiosqlite. Пробуем:

Пример 6.4

**import** json  
**import** aiosqlite  
**import** asyncio  
**from** aiohttp **import** ClientSession, web  
**from** aiologger.loggers.json **import** JsonLogger  
**from** datetime **import** datetime  
  
  
logger = JsonLogger.with\_default\_handlers(  
 level='DEBUG',  
 serializer\_kwargs={'ensure\_ascii': **False**},  
 )  
  
  
**async** **def** create\_table():  
 **async** **with** aiosqlite.connect('weather.db') **as** db:  
 **await** db.execute('CREATE TABLE IF NOT EXISTS requests '  
 '(date text, city text, weather text)')  
 **await** db.commit()  
  
  
**async** **def** save\_to\_db(city, weather):  
 **async** **with** aiosqlite.connect('weather.db') **as** db:  
 **await** db.execute('INSERT INTO requests VALUES (?, ?, ?)',  
 (datetime.now(), city, weather))  
 **await** db.commit()  
  
  
**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()  
 **try**:  
 **return** weather\_json["weather"][0]["main"]  
 **except** KeyError:  
 **return** 'Нет данных'  
  
  
**async** **def** get\_translation(text, source, target):  
 **await** logger.info(f'Поступил запрос на на перевод слова: {text}')  
  
 **async** **with** ClientSession() **as** session:  
 url = 'https://libretranslate.de/translate'  
  
 data = {'q': text, 'source': source, 'target': target, 'format': 'text'}  
  
 **async** **with** session.post(url, json=data) **as** response:  
 translate\_json = **await** response.json()  
  
 **try**:  
 **return** translate\_json['translatedText']  
 **except** KeyError:  
 logger.error(f'Невозможно получить перевод для слова: {text}')  
 **return** text  
  
  
**async** **def** handle(request):  
 city\_ru = request.rel\_url.query['city']  
  
 **await** logger.info(f'Поступил запрос на город: {city\_ru}')  
  
 city\_en = **await** get\_translation(city\_ru, 'ru', 'en')  
 weather\_en = **await** get\_weather(city\_en)  
 weather\_ru = **await** get\_translation(weather\_en, 'en', 'ru')  
  
 result = {'city': city\_ru, 'weather': weather\_ru}  
  
 **await** save\_to\_db(city\_ru, weather\_ru)  
  
 **return** web.Response(text=json.dumps(result, ensure\_ascii=**False**))  
  
  
**async** **def** main():  
 **await** create\_table()  
 app = web.Application()  
 app.add\_routes([web.get('/weather', handle)])  
 runner = web.AppRunner(app)  
 **await** runner.setup()  
 site = web.TCPSite(runner, 'localhost', 8080)  
 **await** site.start()  
  
 **while** **True**:  
 **await** asyncio.sleep(3600)  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 asyncio.run(main())

Для продвинутых

Можете заглянуть внутрь созданного на лету файла weather.db (только используйте не текстовый просмотрщик, а какую-нибудь утилиту для работы с БД, например, [DBeaver](https://dbeaver.io/)). Для каждого запроса создается соответствующая запись в таблице requests. И снова никаких блокировок, мы ведь живем в асинхронном мире.

В заключение этого раздела хочу вас поздравить. Теперь вы имеете в руках все необходимое для создания собственных асинхронных веб-приложений. Неважно какой фреймворк вы будете использовать: FastAPI, Tornado, Falcon или какой-нибудь еще. Принцип останется тот же самый как в старом добром aiohttp: создаем ручку и в ней нанизываем "шашлык" из вызовов асинхронных функций. Главное, за чем необходимо следить — это чтобы в эти функции не затесался какой-нибудь блокирующий зловред из скучной пыльной синхронной вселенной.



На этом временно прощаемся.

***Продолжение***[***здесь***](https://habr.com/ru/articles/774582/)***.***

**Теги:**

* [async](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5basync%5d)
* [await](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5bawait%5d)
* [asyncio](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5basyncio%5d)
* [асинхронность](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C%5d)
* [асинхронное программирование](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5+%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%5d)
* [асинхронные задачи](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5+%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%B8%5d)
* [асинхронные функции](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5+%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8%5d)
* [асинхронный код](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5b%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9+%D0%BA%D0%BE%D0%B4%5d)
* [aiohttp](https://habr.com/ru/search/?target_type=posts&order=relevance&q=%5baiohttp%5d)

**Хабы:**

* [Python](https://habr.com/ru/hubs/python/)
* [Программирование](https://habr.com/ru/hubs/programming/)