
Uma visão geral conceitual de `asyncio`

Release 3.13.7

Guido van Rossum and the Python development team

outubro 01, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Uma visão geral conceitual parte 1: o alto nível	2
1.1	Laço de Eventos	2
1.2	Funções assíncronas e corrotinas	2
1.3	Tarefas	3
1.4	<code>await</code>	4
2	Uma visão geral conceitual, parte 2: os detalhes	6
2.1	O funcionamento interno das corrotinas	6
2.2	Futuros	7
2.3	Um <code>asyncio.sleep</code> caseiro	7

Este artigo COMOFAZER tem como objetivo ajudá-lo a construir um modelo mental sólido de como `asyncio` funciona fundamentalmente, ajudando-lhe a compreender o como e o porquê por trás dos padrões recomendados.

Você pode estar curioso sobre alguns conceitos-chave de `asyncio`. Você conseguirá responder a essas perguntas com tranquilidade ao final deste artigo:

- O que acontece nos bastidores quando um objeto é aguardado?
- Como o `asyncio` diferencia uma tarefa que não precisa de tempo de CPU (como uma solicitação de rede ou leitura de arquivo) de uma tarefa que precisa (como calcular n -fatorial)?
- Como escrever uma variante assíncrona de uma operação, como uma suspensão assíncrona ou uma solicitação de banco de dados.

Ver também

- O [guia](#) que inspirou este artigo COMOFAZER, por Alexander Nordin.
- Esta [série de tutoriais detalhados do YouTube](#) sobre `asyncio` foi criada pelo membro da equipe principal do Python, Łukasz Langa.
- [500 linhas ou menos: um rastreador da Web com corrotinas `asyncio`](#) (em inglês) por A. Jesse Jiryu Davis e Guido van Rossum.

1 Uma visão geral conceitual parte 1: o alto nível

Na parte 1, abordaremos os principais blocos de construção de alto nível de `asyncio`: o laço de eventos, funções de corrotina, objetos de corrotina, tarefas e `await`.

1.1 Laço de Eventos

Tudo em `asyncio` acontece em relação ao laço de eventos. Ele é a estrela do show. É como um maestro de orquestra. Está nos bastidores, gerenciando recursos. Algum poder lhe é explicitamente concedido, mas grande parte de sua capacidade de realizar tarefas advém do respeito e da cooperação de seus operários.

Em termos mais técnicos, o laço de eventos contém uma coleção de tarefas a serem executadas. Algumas tarefas são adicionadas diretamente por você e outras indiretamente por `asyncio`. O laço de eventos pega uma tarefa do seu backlog de tarefas e a invoca (ou “dá a ela o controle”), semelhante a chamar uma função, e então essa tarefa é executada. Uma vez pausada ou concluída, ela retorna o controle para o laço de eventos. O laço de eventos então seleciona outra tarefa do seu pool e a invoca. Você pode *aproximadamente* pensar na coleção de tarefas como uma fila: as tarefas são adicionadas e processadas uma de cada vez, geralmente (mas nem sempre) em ordem. Esse processo se repete indefinidamente, com o laço de eventos em ciclos infinitos. Se não houver mais tarefas pendentes de execução, o laço de eventos é inteligente o suficiente para descansar e evitar o desperdício desnecessário de ciclos de CPU, e retornará quando houver mais trabalho a ser feito.

A execução eficaz depende do bom compartilhamento e da cooperação entre as tarefas; uma tarefa gananciosa pode monopolizar o controle e deixar as outras tarefas na miséria, tornando a abordagem geral do laço de eventos inútil.

```
import asyncio

# Isso cria um laço de eventos e percorre indefinidamente
# sua coleção de trabalhos
event_loop = asyncio.new_event_loop()
event_loop.run_forever()
```

1.2 Funções assíncronas e corrotinas

Esta é uma função básica e chata do Python:

```
def hello_printer():
    print(
        "Olá, sou uma impressora humilde e simples, embora tenha tudo "
        "que preciso na vida: -- \npapel novo e meu querido e amado "
        "parceiro no crime, o polvo."
    )
```

Chamar uma função regular invoca sua lógica ou corpo:

```
>>> hello_printer()
Olá, sou uma impressora humilde e simples, embora tenha tudo o que preciso na vida:
papel novo e meu querido e amado parceiro no crime, o polvo.
```

O `async def`, em oposição a um simples `def`, torna esta uma função assíncrona (ou “função de corrotina”). Chamá-la cria e retorna um objeto corrotina.

```
async def loudmouth_penguin(magic_number: int):
    print(
        "Eu sou um pinguim falante superespecial. Muito mais legal que aquela_
→ impressora. "
        f"Aliás, meu número da sorte é: {magic_number}."
    )
```

Chamar a função assíncrona, `loudmouth_penguin`, não executa a instrução de impressão; em vez disso, cria um objeto corrotina:

```
>>> loudmouth_penguin(magic_number=3)
<coroutine object loudmouth_penguin at 0x104ed2740>
```

Os termos “função de corrotina” e “objeto corrotina” são frequentemente confundidos com corrotina. Isso pode ser confuso! Neste artigo, corrotina se refere especificamente a um objeto corrotina, ou mais precisamente, a uma instância de `types.CoroutineType` (corrotina nativa). Observe que corrotinas também podem existir como instâncias de `collections.abc.Coroutine` — uma distinção importante para a verificação de tipos.

Uma corrotina representa o corpo ou a lógica da função. Uma corrotina precisa ser iniciada explicitamente; novamente, a mera criação da corrotina não a inicia. Notavelmente, a corrotina pode ser pausada e retomada em vários pontos do corpo da função. Essa capacidade de pausar e retomar é o que permite o comportamento assíncrono!

Corrotinas e funções de corrotina foram criadas aproveitando a funcionalidade de geradores e funções geradoras. Lembre-se: uma função geradora é uma função que executa `yield`, como esta:

```
def get_random_number():
    # Este seria um gerador de número aleatório ruim!
    print("Oi")
    yield 1
    print("Olá")
    yield 7
    print("E aí")
    yield 4
    ...
```

Semelhante a uma função de corrotina, chamar uma função geradora não a executa. Em vez disso, ela cria um objeto gerador:

```
>>> get_random_number()
<generator object get_random_number at 0x1048671c0>
```

Você pode prosseguir para o próximo `yield` de um gerador usando a função embutida `next()`. Em outras palavras, o gerador é executado e, em seguida, pausado. Por exemplo:

```
>>> generator = get_random_number()
>>> next(generator)
Oi
1
>>> next(generator)
Olá
7
```

1.3 Tarefas

Em termos gerais, tarefas são corrotinas (não funções de corrotina) vinculadas a um laço de eventos. Uma tarefa também mantém uma lista de funções de retorno de chamada cuja importância ficará clara em breve, quando discutirmos `await`. A maneira recomendada de criar tarefas é via `asyncio.create_task()`.

A criação de uma tarefa a agenda automaticamente para execução (adicionando um retorno de chamada para executá-la na lista de tarefas do laço de eventos, ou seja, coleção de tarefas).

Como há apenas um laço de eventos (em cada thread), `asyncio` se encarrega de associar a tarefa ao laço de eventos para você. Portanto, não há necessidade de especificar o laço de eventos.

```
coroutine = loudmouth_penguin(magic_number=5)
# Isso cria um objeto Task e agenda sua execução por meio do laço de eventos.
task = asyncio.create_task(coroutine)
```

Anteriormente, criamos manualmente o laço de eventos e o configuramos para ser executado indefinidamente. Na prática, é recomendado (e comum) usar `asyncio.run()`, que gerencia o laço de eventos e garante que a corrotina

fornecida termine antes de avançar. Por exemplo, muitos programas assíncronos seguem esta configuração:

```
import asyncio

async def main():
    # Faz todo tipo de coisas malucas, selvagens e assíncronas...
    ...

if __name__ == "__main__":
    asyncio.run(main())
    # O programa não alcançará a seguinte instrução de exibição
    # até que o main() da corrotina seja finalizado.
    print("main() da corrotina concluiu!")
```

É importante estar ciente de que a tarefa em si não é adicionada ao laço de eventos, apenas um retorno de chamada para a tarefa. Isso é importante se o objeto de tarefa que você criou for coletado como lixo antes de ser chamado pelo laço de eventos. Por exemplo, considere este programa:

```
1 async def hello():
2     print("hello!")
3
4 async def main():
5     asyncio.create_task(hello())
6     # Outras instruções assíncronas que são executadas por
7     # um tempo e cedem o controle ao laço de eventos...
8     ...
9
10 asyncio.run(main())
```

Como não há referência ao objeto tarefa criado na linha 5, ele *pode* ser coletado como lixo antes que o laço de eventos o invoque. Instruções posteriores na corrotina `main()` transferem o controle de volta para o laço de eventos para que ele possa invocar outras tarefas. Quando o laço de eventos eventualmente tenta executar a tarefa, pode falhar e descobrir que o objeto `task` não existe! Isso também pode acontecer mesmo que uma corrotina mantenha uma referência a uma tarefa, mas seja concluída antes que ela termine. Quando a corrotina termina, as variáveis locais saem do escopo e podem estar sujeitas à coleta de lixo. Na prática, `asyncio` e o coletor de lixo do Python trabalham arduamente para garantir que esse tipo de coisa não aconteça. Mas isso não é motivo para ser imprudente!

1.4 await

`await` é uma palavra reservada do Python comumente usada de duas maneiras diferentes:

```
await task
await coroutine
```

De maneira crucial, o comportamento de `await` depende do tipo de objeto que está sendo aguardado.

Aguardar uma tarefa cederá o controle da tarefa ou corrotina atual para o laço de eventos. No processo de cessão de controle, algumas coisas importantes acontecem. Usaremos o seguinte exemplo de código para ilustrar:

```
async def plant_a_tree():
    dig_the_hole_task = asyncio.create_task(dig_the_hole())
    await dig_the_hole_task

    # Outras instruções associadas com plantar uma árvore.
    ...
```

Neste exemplo, imagine que o laço de eventos passou o controle para o início da corrotina `plant_a_tree()`. Como visto acima, a corrotina cria uma tarefa e a aguarda. A instrução `await dig_the_hole_task` adiciona um retorno de chamada (que retomará `plant_a_tree()`) à lista de retornos de chamada do objeto `dig_the_hole_task`. E então, a instrução cede o controle para o laço de eventos. Algum tempo depois, o laço de eventos passará o controle

para `dig_the_hole_task` e a tarefa concluirá o que for necessário. Assim que a tarefa for concluída, ela adicionará seus vários retornos de chamada ao laço de eventos, neste caso, uma chamada para retomar `plant_a_tree()`.

De modo geral, quando a tarefa aguardada termina (`dig_the_hole_task`), a tarefa original ou corrotina (`plant_a_tree()`) é adicionada novamente à lista de tarefas do laço de eventos para ser retomada.

Este é um modelo mental básico, porém confiável. Na prática, as transferências de controle são um pouco mais complexas, mas não muito. Na parte 2, abordaremos os detalhes que tornam isso possível.

Ao contrário de tarefas, aguardar uma corrotina não devolve o controle ao laço de eventos! Envolver uma corrotina em uma tarefa primeiro e depois aguardar isso cederia o controle. O comportamento de `await coroutine` é efetivamente o mesmo que invocar uma função Python síncrona comum. Considere este programa:

```
import asyncio

async def coro_a():
    print("Sou coro_a(). Oi!")

async def coro_b():
    print("Sou coro_b(). Espero que ninguém monopolize o laço de eventos...")

async def main():
    task_b = asyncio.create_task(coro_b())
    num_repeats = 3
    for _ in range(num_repeats):
        await coro_a()
    await task_b

asyncio.run(main())
```

A primeira instrução na corrotina `main()` cria `task_b` e a agenda para execução via laço de eventos. Em seguida, `coro_a()` é aguardado repetidamente. O controle nunca cede ao laço de eventos, e é por isso que vemos a saída de todas as três invocações de `coro_a()` antes da saída de `coro_b()`:

```
Sou coro_a(). Oi!
Sou coro_a(). Oi!
Sou coro_a(). Oi!
Sou coro_b(). Espero que ninguém monopolize o laço de eventos...
```

Se alterarmos `await coro_a()` para `await asyncio.create_task(coro_a())`, o comportamento muda. A corrotina `main()` cede o controle ao laço de eventos com essa instrução. O laço de eventos então prossegue com seu backlog de trabalho, chamando `task_b` e, em seguida, a tarefa que encerra `coro_a()` antes de retomar a corrotina `main()`.

```
Sou coro_b(). Espero que ninguém monopolize o laço de eventos...
Sou coro_a(). Oi!
Sou coro_a(). Oi!
Sou coro_a(). Oi!
```

Esse comportamento de `await coroutine` pode confundir muita gente! Este exemplo destaca como usar apenas `await coroutine` pode, involuntariamente, monopolizar o controle de outras tarefas e efetivamente paralisar o laço de eventos. `asyncio.run()` pode ajudar a detectar tais ocorrências por meio do sinalizador `debug=True`, que habilita o modo de depuração. Entre outras coisas, ele registrará quaisquer corrotinas que monopolizem a execução por 100 ms ou mais.

O design intencionalmente troca alguma clareza conceitual em torno do uso de `await` por melhor desempenho. Cada vez que uma tarefa é aguardada, o controle precisa ser passado por toda a pilha de chamadas até o laço de eventos. Isso pode parecer insignificante, mas em um programa grande com muitos `await` e uma pilha de chamadas extensa, essa sobrecarga pode representar um significativo prejuízo ao desempenho.

2 Uma visão geral conceitual, parte 2: os detalhes

A parte 2 detalha os mecanismos que `asyncio` usa para gerenciar o fluxo de controle. É aqui que a mágica acontece. Você sairá desta seção sabendo o que `await` faz nos bastidores e como criar seus próprios operadores assíncronos.

2.1 O funcionamento interno das corrotinas

`asyncio` utiliza quatro componentes para passar o controle.

`coroutine.send(arg)` é o método usado para iniciar ou retomar uma corrotina. Se a corrotina foi pausada e agora está sendo retomada, o argumento `arg` será enviado como valor de retorno da instrução `yield` que a pausou originalmente. Se a corrotina estiver sendo usada pela primeira vez (em vez de ser retomada), `arg` deve ser `None`.

```
1 class Rock:
2     def __await__(self):
3         value_sent_in = yield 7
4         print(f"Rock.__await__ resumindo com o valor: {value_sent_in}.")
5         return value_sent_in
6
7 async def main():
8     print("Iniciando main() da corrotina.")
9     rock = Rock()
10    print("Aguardando rock...")
11    value_from_rock = await rock
12    print(f"Corrotina recebeu valor: {value_from_rock} de rock.")
13    return 23
14
15 coroutine = main()
16 intermediate_result = coroutine.send(None)
17 print(f"Corrotina pausou e retornou o valor intermediário: {intermediate_result}.")
18
19 print(f"Resumindo corrotina e enviando o valor: 42.")
20 try:
21     coroutine.send(42)
22 except StopIteration as e:
23     returned_value = e.value
24 print(f"O main() da corrotina finalizou e forneceu o value: {returned_value}.")
```

`yield`, como de costume, pausa a execução e retorna o controle ao chamador. No exemplo acima, `yield`, na linha 3, é chamado por `... = await rock` na linha 11. Em termos mais gerais, `await` chama o método `__await__()` do objeto fornecido. `await` também faz algo muito especial: ele propaga (ou “repassa”) quaisquer `yields` que recebe na cadeia de chamadas. Neste caso, voltamos a `... = coroutine.send(None)` na linha 16.

A corrotina é retomada por meio da chamada `coroutine.send(42)` na linha 21. A corrotina continua de onde foi executada (ou pausada) com `yield` na linha 3 e executa as instruções restantes em seu corpo. Quando uma corrotina termina, ela levanta uma exceção `StopIteration` com o valor de retorno anexado ao atributo `value`.

Esse trecho de código produz esta saída:

```
Beginning coroutine main().
Awaiting rock...
Coroutine paused and returned intermediate value: 7.
Resuming coroutine and sending in value: 42.
Rock.__await__ resumining with value: 42.
Coroutine received value: 42 from rock.
Coroutine main() finished and provided value: 23.
```

It's worth pausing for a moment here and making sure you followed the various ways that control flow and values were passed. A lot of important ideas were covered and it's worth ensuring your understanding is firm.

The only way to yield (or effectively cede control) from a coroutine is to `await` an object that yields in its `__await__` method. That might sound odd to you. You might be thinking:

1. What about a `yield` directly within the coroutine function? The coroutine function becomes an async generator function, a different beast entirely.
2. What about a `yield` from within the coroutine function to a (plain) generator? That causes the error: `SyntaxError: yield from not allowed in a coroutine`. This was intentionally designed for the sake of simplicity – mandating only one way of using coroutines. Initially `yield` was barred as well, but was re-accepted to allow for async generators. Despite that, `yield from` and `await` effectively do the same thing.

2.2 Futuros

A future is an object meant to represent a computation's status and result. The term is a nod to the idea of something still to come or not yet happened, and the object is a way to keep an eye on that something.

A future has a few important attributes. One is its state which can be either “pending”, “cancelled” or “done”. Another is its result, which is set when the state transitions to done. Unlike a coroutine, a future does not represent the actual computation to be done; instead, it represents the status and result of that computation, kind of like a status light (red, yellow or green) or indicator.

`asyncio.Task` subclasses `asyncio.Future` in order to gain these various capabilities. The prior section said tasks store a list of callbacks, which wasn't entirely accurate. It's actually the `Future` class that implements this logic, which `Task` inherits.

Futures may also be used directly (not via tasks). Tasks mark themselves as done when their coroutine is complete. Futures are much more versatile and will be marked as done when you say so. In this way, they're the flexible interface for you to make your own conditions for waiting and resuming.

2.3 Um `asyncio.sleep` caseiro

We'll go through an example of how you could leverage a future to create your own variant of asynchronous sleep (`async_sleep`) which mimics `asyncio.sleep()`.

This snippet registers a few tasks with the event loop and then awaits a coroutine wrapped in a task: `async_sleep(3)`. We want that task to finish only after three seconds have elapsed, but without preventing other tasks from running.

```
async def other_work():
    print("I like work. Work work.")

async def main():
    # Add a few other tasks to the event loop, so there's something
    # to do while asynchronously sleeping.
    work_tasks = [
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work())
    ]
    print(
        "Beginning asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime('%H:%M:%S') }."
    )
    await asyncio.create_task(async_sleep(3))
    print(
        "Done asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime('%H:%M:%S') }."
    )
    # asyncio.gather effectively awaits each task in the collection.
    await asyncio.gather(*work_tasks)
```

Below, we use a future to enable custom control over when that task will be marked as done. If `future.set_result()` (the method responsible for marking that future as done) is never called, then this task will never finish. We've also enlisted the help of another task, which we'll see in a moment, that will monitor how much time has elapsed and, accordingly, call `future.set_result()`.

```
async def async_sleep(seconds: float):
    future = asyncio.Future()
    time_to_wake = time.time() + seconds
    # Add the watcher-task to the event loop.
    watcher_task = asyncio.create_task(_sleep_watcher(future, time_to_wake))
    # Block until the future is marked as done.
    await future
```

Below, we'll use a rather bare object, `YieldToEventLoop()`, to yield from `__await__` in order to cede control to the event loop. This is effectively the same as calling `asyncio.sleep(0)`, but this approach offers more clarity, not to mention it's somewhat cheating to use `asyncio.sleep` when showcasing how to implement it!

As usual, the event loop cycles through its tasks, giving them control and receiving control back when they pause or finish. The `watcher_task`, which runs the coroutine `_sleep_watcher(...)`, will be invoked once per full cycle of the event loop. On each resumption, it'll check the time and if not enough has elapsed, then it'll pause once again and hand control back to the event loop. Eventually, enough time will have elapsed, and `_sleep_watcher(...)` will mark the future as done, and then itself finish too by breaking out of the infinite `while` loop. Given this helper task is only invoked once per cycle of the event loop, you'd be correct to note that this asynchronous sleep will sleep *at least* three seconds, rather than exactly three seconds. Note this is also of true of `asyncio.sleep`.

```
class YieldToEventLoop:
    def __await__(self):
        yield

async def _sleep_watcher(future, time_to_wake):
    while True:
        if time.time() >= time_to_wake:
            # This marks the future as done.
            future.set_result(None)
            break
        else:
            await YieldToEventLoop()
```

Aqui está a saída completa do programa:

```
$ python custom-async-sleep.py
Beginning asynchronous sleep at time: 14:52:22.
I like work. Work work.
I like work. Work work.
I like work. Work work.
Done asynchronous sleep at time: 14:52:25.
```

You might feel this implementation of asynchronous sleep was unnecessarily convoluted. And, well, it was. The example was meant to showcase the versatility of futures with a simple example that could be mimicked for more complex needs. For reference, you could implement it without futures, like so:

```
async def simpler_async_sleep(seconds):
    time_to_wake = time.time() + seconds
    while True:
        if time.time() >= time_to_wake:
            return
        else:
            await YieldToEventLoop()
```


But, that's all for now. Hopefully you're ready to more confidently dive into some async programming or check out advanced topics in the rest of the documentation.