

FastAPI Aprender

Concorrência e async / await

Detalhes sobre a sintaxe `async def` para *funções de operação de rota* e alguns conceitos de código assíncrono, concorrência e paralelismo.

Com pressa?

TL;DR:

Se você estiver utilizando bibliotecas de terceiros que dizem para você chamar as funções com `await`, como:

```
results = await some_library()
```

Então, declare sua *função de operação de rota* com `async def` como:

```
@app.get('/')
async def read_results():
    results = await some_library()
    return results
```



Note

Você só pode usar `await` dentro de funções criadas com `async def`.

Se você está usando biblioteca de terceiros que se comunica com alguma coisa (um banco de dados, uma API, sistema de arquivos etc) e não tem suporte para utilizar `await` (esse é atualmente o caso para a maioria das bibliotecas de banco de dados), então declare suas *funções de operação de rota* normalmente, com apenas `def`, como:

```
@app.get('/')
def results():
    results = some_library()
    return results
```

Se sua aplicação (de alguma forma) não tem que se comunicar com nada mais e tem que esperar que o respondam, use `async def`.

Se você simplesmente não sabe, use apenas `def`.

Note: Você pode misturar `def` e `async def` nas suas *funções de operação de rota* tanto quanto necessário e definir cada função usando a melhor opção para você. FastAPI irá fazer a coisa certa com elas.

De qualquer forma, em ambos os casos acima, FastAPI irá trabalhar assincronamente e ser extremamente rápido.

Seguindo os passos acima, ele será capaz de fazer algumas otimizações de performance.





Detalhes Técnicos


Versões modernas de Python tem suporte para "**código assíncrono**" usando algo chamado "**corrotinas**", com sintaxe `async` e `await`.




Vamos ver aquela frase por partes na seção abaixo:



- **Código assíncrono**
- `async` e `await`
- **Corrotinas**

Código assíncrono

Código assíncrono apenas significa que a linguagem  tem um jeito de dizer para o computador / programa  que em certo ponto, ele  terá que esperar por *algo* para finalizar em outro lugar. Vamos dizer que esse *algo* seja chamado "arquivo lento" .

Então, durante esse tempo, o computador pode ir e fazer outro trabalho, enquanto o "arquivo lento"  termine.

Então o computador / programa  irá voltar toda hora que tiver uma chance porquê ele ainda está esperando o "arquivo lento", ou ele  nunca irá terminar todo o trabalho que tem até esse ponto. E ele  irá ver se alguma das tarefas que estava esperando já terminaram, fazendo o que quer que tinham que fazer.

Depois, ele  pega a primeira tarefa para finalizar (vamos dizer, nosso "arquivo lento" ) e continua o que ele tem que fazer com isso.

Esse "esperar por algo" normalmente se refere a operações I/O que são relativamente "lentas" (comparadas a velocidade do processador e da memória RAM), como esperar por:

- dados do cliente para serem enviados através da rede
- dados enviados pelo seu programa para serem recebidos pelo cliente através da rede
- conteúdo de um arquivo no disco para ser lido pelo sistema e entregue ao seu programa
- conteúdo que seu programa deu ao sistema para ser escrito no disco
- uma operação remota API
- uma operação no banco de dados para finalizar
- uma solicitação no banco de dados esperando o retorno do resultado
- etc.

Enquanto o tempo de execução é consumido mais pela espera das operações I/O, essas operações são chamadas de operações "limitadas por I/O".

Isso é chamado de "assíncrono" porque o computador / programa não tem que ser "sincronizado" com a tarefa lenta, esperando pelo exato momento que a tarefa finalize, enquanto não faz nada, para ser capaz de pegar o resultado da tarefa e dar continuidade ao trabalho.

Ao invés disso, sendo um sistema "assíncrono", uma vez finalizada, a tarefa pode esperar um pouco (alguns microssegundos) para que o computador / programa finalize o que quer que esteja fazendo, e então volte para pegar o resultado e continue trabalhando com ele.

Para "síncrono" (contrário de "assíncrono") também é utilizado o termo "sequencial", porque o computador / programa segue todos os passos, na sequência, antes de trocar para uma tarefa diferente, mesmo se alguns passos envolvam esperar.

Concorrência e hambúrgueres

Essa ideia de código **assíncrono** descrito acima é algo às vezes chamado de "**concorrência**". E é diferente de "**paralelismo**".

Concorrência e **paralelismo** ambos são relacionados a "diferentes coisas acontecendo mais ou menos ao mesmo tempo".

Mas os detalhes entre *concorrência* e *paralelismo* são bem diferentes.

Para ver essa diferença, imagine a seguinte história sobre hambúrgueres:

Hambúrgueres concorrentes

Você vai com seu *crush* :heart_eyes: na lanchonete, fica na fila enquanto o caixa pega os pedidos das pessoas na sua frente.

Então chega a sua vez, você pede dois saborosos hambúrgueres para você e seu *crush* :heart_eyes:.

Você paga.

O caixa diz alguma coisa para o cara na cozinha para que ele tenha que preparar seus hambúrgueres (mesmo embora ele esteja preparando os lanches dos outros clientes).

O caixa te entrega seu número de chamada.

Enquanto você espera, você vai com seu *crush* :heart_eyes: e pega uma mesa, senta e conversa com seu *crush* :heart_eyes: por um bom tempo (como seus hambúrgueres são muito saborosos, leva um tempo para serem preparados).

Enquanto você está sentado na mesa com seu *crush* :heart_eyes:, esperando os hambúrgueres, você pode gastar o tempo admirando como lindo, maravilhoso e esperto é seu *crush* :heart_eyes:.

Enquanto espera e conversa com seu *crush* :heart_eyes:, de tempos em tempos, você verifica o número de chamada exibido no balcão para ver se já é sua vez.

Então a certo ponto, é finalmente sua vez. Você vai no balcão, pega seus hambúrgueres e volta para a mesa.

Você e seu *crush* :heart_eyes: comem os hambúrgueres e aproveitam o tempo.

Imagine que você seja o computador / programa nessa história.

Enquanto você está na fila, tranquilo, esperando por sua vez, não está fazendo nada "produtivo". Mas a fila é rápida porque o caixa só está pegando os pedidos, então está tudo bem.

Então, quando é sua vez, você faz o trabalho "produtivo" de verdade, você processa o menu, decide o que quer, pega a escolha de seu *crush* :heart_eyes:, paga, verifica se entregou o valor correto em dinheiro ou cartão de crédito, verifica se foi cobrado corretamente, verifica se seu pedido está correto etc.

Mas então, embora você ainda não tenha os hambúrgueres, seu trabalho no caixa está "pausado", porque você tem que esperar seus hambúrgueres estarem prontos.

Mas enquanto você se afasta do balcão e senta na mesa com o número da sua chamada, você pode trocar sua atenção para seu *crush* :heart_eyes:, e "trabalhar" nisso. Então você está novamente fazendo algo muito "produtivo", como flertar com seu *crush* :heart_eyes:.

Então o caixa diz que "seus hambúrgueres estão prontos" colocando seu número no balcão, mas você não corre que nem um maluco imediatamente quando o número exibido é o seu. Você sabe que ninguém irá roubar seus hambúrgueres porque você tem o número de chamada, e os outros tem os números deles.

Então você espera que seu *crush* :heart_eyes: termine a história que estava contando (terminar o trabalho atual / tarefa sendo processada), sorri gentilmente e diz que você está indo buscar os hambúrgueres.

Então você vai no balcão, para a tarefa inicial que agora está finalizada, pega os hambúrgueres, e leva para a mesa. Isso finaliza esse passo / tarefa da interação com o balcão. Agora é criada uma nova tarefa, "comer hambúrgueres", mas a tarefa anterior, "pegar os hambúrgueres" já está finalizada.

Hambúrgueres paralelos

Você vai com seu *crush* :heart_eyes: em uma lanchonete paralela.

Você fica na fila enquanto alguns (vamos dizer 8) caixas pegam os pedidos das pessoas na sua frente.

Todo mundo antes de você está esperando pelos hambúrgueres estarem prontos antes de deixar o caixa porque cada um dos 8 caixas vai e prepara o hambúrguer antes de pegar o próximo pedido.

Então é finalmente sua vez, e pede 2 hambúrgueres muito saborosos para você e seu *crush* :heart_eyes:.

Você paga.

O caixa vai para a cozinha.

Você espera, na frente do balcão, para que ninguém pegue seus hambúrgueres antes de você, já que não tem números de chamadas.

Enquanto você e seu *crush* :heart_eyes: estão ocupados não permitindo que ninguém passe a frente e pegue seus hambúrgueres assim que estiverem prontos, você não pode dar atenção ao seu *crush* :heart_eyes:.

Isso é trabalho "síncrono", você está "sincronizado" com o caixa / cozinheiro. Você tem que esperar e estar lá no exato momento que o caixa / cozinheiro terminar os hambúrgueres e dá-los a você, ou então, outro alguém pode pegá-los.

Então seu caixa / cozinheiro finalmente volta com seus hambúrgueres, depois de um longo tempo esperando por eles em frente ao balcão.

Você pega seus hambúrgueres e vai para a mesa com seu *crush* :heart_eyes:.

Vocês comem os hambúrgueres, e o trabalho está terminado.

Não houve muita conversa ou flerte já que a maior parte do tempo foi gasto esperando os lanches na frente do balcão.

Nesse cenário dos hambúrgueres paralelos, você é um computador / programa com dois processadores (você e seu *crush* :heart_eyes:), ambos esperando e dedicando a atenção de estar "esperando no balcão" por um bom tempo.

A lanchonete paralela tem 8 processadores (caixas / cozinheiros). Enquanto a lanchonete dos hambúrgueres concorrentes tinham apenas 2 (um caixa e um cozinheiro).

Ainda assim, a última experiência não foi a melhor.

Essa poderia ser a história paralela equivalente aos hambúrgueres.

Para um exemplo "mais real", imagine um banco.

Até recentemente, a maioria dos bancos tinha muitos caixas e uma grande fila.

Todos os caixas fazendo todo o trabalho, um cliente após o outro.

E você tinha que esperar na fila por um longo tempo ou poderia perder a vez.

Você provavelmente não gostaria de levar seu *crush* :heart_eyes: com você para um rolezinho no banco.

Conclusão dos hambúrgueres

Nesse cenário dos "hambúrgueres com seu *crush* :heart_eyes:", como tem muita espera, faz mais sentido ter um sistema concorrente.

Esse é o caso da maioria das aplicações web.

Geralmente são muitos usuários, e seu servidor está esperando pelas suas conexões não tão boas para enviar as requisições.

E então esperando novamente pelas respostas voltarem.

Essa "espera" é medida em microssegundos, e ainda assim, somando tudo, é um monte de espera no final.

Por isso que faz muito mais sentido utilizar código assíncrono para APIs web.

A maioria dos frameworks Python existentes mais populares (incluindo Flask e Django) foram criados antes que os novos recursos assíncronos existissem em Python. Então, os meios que

eles podem ser colocados em produção para suportar execução paralela mais a forma antiga de execução assíncrona não são tão poderosos quanto as novas capacidades.

Mesmo embora a especificação principal para web assíncrono em Python (ASGI) foi desenvolvida no Django, para adicionar suporte para WebSockets.

Esse tipo de assincronicidade é o que fez NodeJS popular (embora NodeJS não seja paralelo) e que essa seja a força do Go como uma linguagem de programa.

E esse é o mesmo nível de performance que você tem com o **FastAPI**.

E como você pode ter paralelismo e sincronicidade ao mesmo tempo, você tem uma maior performance do que a maioria dos frameworks NodeJS testados e lado a lado com Go, que é uma linguagem compilada próxima ao C ([tudo graças ao Starlette](#)) [[↔](#)].

Concorrência é melhor que paralelismo?

Não! Essa não é a moral da história.

Concorrência é diferente de paralelismo. E é melhor em cenários **específicos** que envolvam um monte de espera. Por isso, geralmente é muito melhor do que paralelismo para desenvolvimento de aplicações web. Mas não para tudo.

Então, para equilibrar tudo, imagine a seguinte historinha:

▮ Você tem que limpar uma grande casa suja.

Sim, essa é toda a história.

Não há espera em lugar algum, apenas um monte de trabalho para ser feito, em múltiplos cômodos da casa.

Você poderia ter chamadas como no exemplo dos hambúrgueres, primeiro a sala de estar, então a cozinha, mas você não está esperando por nada, apenas limpar e limpar, as chamadas não afetariam em nada.

Levaria o mesmo tempo para finalizar com ou sem chamadas (concorrência) e você teria feito o mesmo tanto de trabalho.

Mas nesse caso, se você trouxesse os 8 ex-caixas / cozinheiros / agora-faxineiros, e cada um deles (mais você) pudessem dividir a casa para limpá-la, vocês fariam toda a limpeza em **paralelo**, com a ajuda extra, e terminariam muito mais cedo.

Nesse cenário, cada um dos faxineiros (incluindo você) poderia ser um processador, fazendo a sua parte do trabalho.

E a maior parte do tempo de execução é tomada por trabalho (ao invés de ficar esperando), e o trabalho em um computador é feito pela CPU, que podem gerar problemas que são chamados de "limite de CPU".

Exemplos comuns de limite de CPU são coisas que exigem processamento matemático complexo.

Por exemplo:

- **Processamento de áudio ou imagem**
- **Visão do Computador:** uma imagem é composta por milhões de pixels, cada pixel tem 3 valores (cores, processamento que normalmente exige alguma computação em todos esses pixels ao mesmo tempo)
- **Machine Learning:** Normalmente exige muita multiplicação de matrizes e vetores. Pense numa grande folha de papel com números e multiplicando todos eles juntos e ao mesmo tempo.
- **Deep Learning:** Esse é um subcampo do Machine Learning, então o mesmo se aplica. A diferença é que não há apenas uma grande folha de papel com números para multiplicar, mas um grande conjunto de folhas de papel, e em muitos casos, você utiliza um processador especial para construir e/ou usar modelos.

Concorrência + Paralelismo: Web + Machine learning

Com **FastAPI** você pode levar a vantagem da concorrência que é muito comum para desenvolvimento web (o mesmo atrativo de NodeJS).

Mas você também pode explorar os benefícios do paralelismo e multiprocessamento (tendo múltiplos processadores rodando em paralelo) para trabalhos pesados que geram **limite de CPU** como aqueles em sistemas de Machine Learning.

Isso, mais o simples fato que Python é a principal linguagem para **Data Science**, Machine Learning e especialmente Deep Learning, faz do FastAPI uma ótima escolha para APIs web e aplicações com Data Science / Machine Learning (entre muitas outras).

Para ver como alcançar esse paralelismo em produção veja a seção sobre [Deployment ↗](#).

async e await

Versões modernas do Python tem um modo muito intuitivo para definir código assíncrono. Isso faz parecer normal o código "sequencial" e fazer o "esperar" para você nos momentos certos.

Quando tem uma operação que exigirá espera antes de dar os resultados e tem suporte para esses recursos Python, você pode escrever assim:

```
burgers = await get_burgers(2)
```

A chave aqui é o `await`. Ele diz ao Python que ele tem que esperar por `get_burgers(2)` para finalizar suas coisas antes de armazenar os resultados em `burgers`. Com isso, o Python saberá que ele pode ir e fazer outras coisas nesse meio tempo (como receber outra requisição).

Para o `await` funcionar, tem que estar dentro de uma função que suporte essa assincronicidade. Para fazer isso, apenas declare a função com `async def`:

```
async def get_burgers(number: int):  
    # Fazer alguma coisa assíncrona para criar os hambúrgueres  
    return burgers
```

...ao invés de `def`:

```
# Isso não é assíncrono  
def get_sequential_burgers(number: int):  
    # Faz alguma coisa sequencial para criar os hambúrgueres  
    return burgers
```

Com `async def`, o Python sabe que, dentro dessa função, tem que estar ciente das expressões `await`, e que isso pode "pausar" a execução dessa função, e poderá fazer outra coisa antes de voltar.

Quando você quiser chamar uma função `async def`, você tem que "esperar". Então, isso não funcionará:

```
# Isso não irá funcionar, porque get_burgers foi definido com: async def  
burgers = get_burgers(2)
```

Então, se você está usando uma biblioteca que diz que você pode chamá-la com `await`, você precisa criar as *funções de operação de rota* com `async def`, como em:

```
@app.get('/burgers')  
async def read_burgers():  
    burgers = await get_burgers(2)  
    return burgers
```

Mais detalhes técnicos

Você deve ter observado que `await` pode ser usado somente dentro de funções definidas com `async def`.

Mas ao mesmo tempo, funções definidas com `async def` tem que ser aguardadas. Então, funções com `async def` podem ser chamadas somente dentro de funções definidas com `async def` também.

Então, sobre o ovo e a galinha, como você chama a primeira função async?

Se você estiver trabalhando com **FastAPI** não terá que se preocupar com isso, porque essa "primeira" função será a sua *função de operação de rota*, e o FastAPI saberá como fazer a coisa certa.

Mas se você quiser usar `async` / `await` sem FastAPI, [verifique a documentação oficial Python \[↔\]](#).

Outras formas de código assíncrono

Esse estilo de usar `async` e `await` é relativamente novo na linguagem.

Mas ele faz o trabalho com código assíncrono muito mais fácil.

Essa mesma sintaxe (ou quase a mesma) foi também incluída recentemente em versões modernas do JavaScript (no navegador e NodeJS).

Mas antes disso, controlar código assíncrono era bem mais complexo e difícil.

Nas versões anteriores do Python, você poderia utilizar threads ou [Gevent \[↔\]](#). Mas o código é um pouco mais complexo de entender, debugar, e pensar sobre.

Nas versões anteriores do NodeJS / Navegador JavaScript, você poderia utilizar "callbacks". O que leva ao [inferno do callback \[↔\]](#).

Corrotinas

Corrotina é apenas um jeito bonitinho para a coisa que é retornada de uma função `async def`. O Python sabe que é uma função que pode começar e terminar em algum ponto, mas que pode ser pausada internamente também, sempre que tiver um `await` dentro dela.

Mas toda essa funcionalidade de código assíncrono com `async` e `await` é muitas vezes resumida como "corrotina". É comparável ao principal recurso chave do Go, a "Gorotina".

Conclusão

Vamos ver a mesma frase com o conteúdo cima:

Versões modernas do Python tem suporte para "**código assíncrono**" usando algo chamado "**corrotinas**", com sintaxe `async` e `await`.

Isso pode fazer mais sentido agora.

Tudo isso é o que deixa o FastAPI poderoso (através do Starlette) e que o faz ter uma performance impressionante.

Detalhes muito técnicos



Warning

Você pode provavelmente pular isso.

Esses são detalhes muito técnicos de como **FastAPI** funciona por baixo do capô.

Se você tem algum conhecimento técnico (corrotinas, threads, blocking etc) e está curioso sobre como o FastAPI controla o `async def` vs normal `def`, vá em frente.

Funções de operação de rota

Quando você declara uma *função de operação de rota* com `def` normal ao invés de `async def`, ela é rodada em uma threadpool externa que então é aguardada, ao invés de ser chamada diretamente (ela poderia bloquear o servidor).

Se você está chegando de outro framework assíncrono que não faz o trabalho descrito acima e você está acostumado a definir triviais *funções de operação de rota* com simples `def` para ter um mínimo ganho de performance (cerca de 100 nanosegundos), por favor observe que no **FastAPI** o efeito pode ser bem o oposto. Nesses casos, é melhor usar `async def` a menos que suas *funções de operação de rota* utilizem código que performem bloqueamento IO.

Ainda, em ambas as situações, as chances são que o **FastAPI** será [ainda mais rápido](#) → do que (ou ao menos comparável a) seus frameworks antecessores.

Dependências

O mesmo se aplica para as dependências. Se uma dependência tem as funções com padrão `def` ao invés de `async def`, ela é rodada no threadpool externo.

Sub-dependências

Você pode ter múltiplas dependências e sub-dependências exigindo uma a outra (como parâmetros de definições de funções), algumas delas podem ser criadas com `async def` e

algumas com `def` normal. Isso ainda poderia funcionar, e aquelas criadas com `def` podem ser chamadas em uma thread externa ao invés de serem "aguardadas".

Outras funções de utilidade

Qualquer outra função de utilidade que você chame diretamente pode ser criada com `def` normal ou `async def` e o FastAPI não irá afetar o modo como você a chama.

Isso está em contraste às funções que o FastAPI chama para você: *funções de operação de rota* e dependências.

Se sua função de utilidade é uma função normal com `def`, ela será chamada diretamente (como você a escreve no código), não em uma threadpool, se a função é criada com `async def` então você deve esperar por essa função quando você chamá-la no seu código.

Novamente, esses são detalhes muito técnicos que provavelmente possam ser úteis caso você esteja procurando por eles.

Caso contrário, você deve ficar bem com as dicas da seção acima: [Com pressa?](#).