

並行処理と `async / await`

*path operation 関数*のための `async def` に関する詳細と非同期 (asynchronous) コード、並行処理 (Concurrency)、そして、並列処理 (Parallelism) の背景について。

急いでいますか？

TL;DR:

次のような、`await` を使用して呼び出すべきサードパーティライブラリを使用している場合:

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

以下の様に `async def` を使用して *path operation 関数* を宣言します。

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

!!! note "備考" `async def` を使用して作成された関数の内部でしか `await` は使用できません。

データベース、API、ファイルシステムなどと通信し、`await` の使用をサポートしていないサードパーティライブラリ (現在のほとんどのデータベースライブラリに当てはまります) を使用している場合、次の様に、単に `def` を使用して通常通り *path operation 関数* を宣言してください:

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

アプリケーションが (どういうわけか) 他の何とも通信せず、応答を待つ必要がない場合は、`async def` を使用して下さい。

よく分からない場合は、通常の `def` を使用して下さい。

備考: *path operation 関数*に必要なだけ `def` と `async def` を混在させ、それぞれに最適なオプションを使用して定義できます。それに応じてFastAPIは正しい処理を行います。

とにかく、上記のいずれの場合でもFastAPIは非同期で動作し、非常に高速です。

しかし、上記のステップに従うことで、パフォーマンスの最適化を行えます。

技術詳細

現代版のPythonは「非同期コード」を、「**コルーチン**」と称されるものを利用してサポートしています。これは `async` と `await` 構文を用います。

次のセクションで、フレーズ内のパーツを順に見ていきましょう:

- 非同期コード
- `async` と `await`
- コルーチン

非同期コード

非同期コードとは、言語🗨️がコード内のどこかで、コンピュータ/プログラム🖥️に *他の何か* がどこか別の箇所で終了するのを待つように伝える手段を持っていることを意味します。*他の何か* は「遅いファイル📁」と呼ばれているとしましょう。

したがって、コンピュータは「遅いファイル📁」が終了するまで、他の処理ができます。

コンピュータ/プログラム🖥️は再び待機する機会があるときや、その時点で行っていたすべての作業が完了するたびに戻ってきます。そして、必要な処理をしながら、コンピュータ/プログラム🖥️が待っていた処理のどれかが終わっているかどうか確認します。

次に、それ🖥️が最初のタスク (要するに、先程の「遅いファイル📁」) を終わらせて、そのタスクの結果を使う必要がある処理を続けます。

この「他の何かを待つ」とは、通常以下の様なものを待つような (プロセッサとRAMメモリの速度に比べて) 相対的に「遅い」I/O 操作を指します:

- ネットワーク経由でクライアントから送信されるデータ
- ネットワーク経由でクライアントが受信する、プログラムから送信されたデータ
- システムによって読み取られ、プログラムに渡されるディスク内のファイル内容
- プログラムがシステムに渡して、ディスクに書き込む内容
- リモートAPI操作
- データベース操作の完了
- データベースクエリが結果を返すこと
- など。

実行時間のほとんどがI/O 操作の待ち時間が占めるため、このような操作を「I/O バウンド」操作と言います。

コンピュータ/プログラムがこのような遅いタスクと「同期 (タスクの結果を取得して作業を続行するために、何もせずに、タスクが完了する瞬間を正確に待つ)」する必要がないため、「非同期」と呼ばれます。

その代わりに、「非同期」システムであることにより、いったん終了すると、タスクは、コンピュータ/プログラムが既に開始した処理がすべて完了するのをほんの少し (数マイクロ秒) 待って、結果を受け取りに戻ってきます。そして、処理を継続します。

「同期」の場合 (「非同期」とは異なり)、「シーケンシャル」という用語もよく使用されます。これは、コンピュータ/プログラムがすべてのステップを (待機が伴う場合でも別のタスクに切り替えることなく) 順番に実行するためです。

並行処理とハンバーガー

上記の非同期コードのアイデアは、「並行処理」と呼ばれることもあります。「並列処理」とは異なります。

並行処理と**並列処理**はどちらも「多かれ少なかれ同時に発生するさまざまなこと」に関連しています。

ただし、**並行処理**と**並列処理**の詳細はまったく異なります。

違いを確認するには、ハンバーガーに関する次の物語を想像してみてください:

並行ハンバーガー

ファストフード🍔を食べようと、好きな人😍とレジに並んでおり、レジ係👩🏻💻があなたの前にいる人達の注文を受けつけています。

それからあなたの番になり、好きな人😍と自分のために、2つの非常に豪華なハンバーガー🍔を注文します。

料金を支払います💵。

レジ係👩🏻💻はキッチンの男👨🏻🍳に向かって、あなたのハンバーガー🍔を準備しなければならないと伝えるために何か言いました(彼は現在、前のお客さんの商品を準備していますが)。

レジ係👩🏻💻はあなたに番号札を渡します。

待っている間、好きな人😍と一緒にテーブルを選んで座り、好きな人😍と長い間話をします(注文したハンバーガーは非常に豪華で、準備に少し時間がかかるので🌟🍔🌟)。

ハンバーガー🍔を待ちながら好きな人😍とテーブルに座っている間、あなたの好きな人がなんて素晴らしく、かわいくて頭がいいんだと🌟😍🌟惚れ惚れしながら時間を費やすことができます。

好きな人😍と話しながら待っている間、ときどき、カウンターに表示されている番号をチェックして、自分の番号かどうかを確認します。

その後、ついにあなたの番になりました。カウンターに行き、ハンバーガー🍔を手に入れてテーブルに戻ります。

あなたとあなたの好きな人😍はハンバーガー🍔を食べて、楽しい時間を過ごします🌟。

上記のストーリーで、あなたがコンピュータ/プログラム🖥だと想像してみてください。

列にいる間、あなたはアイドル状態です😴。何も「生産的」なことをせず、ただ自分の番を待っています。しかし、レジ係👩🏻💻は注文を受け取るだけなので(商品の準備をしているわけではない)、列は高速です。したがって、何も問題ありません。

それから、あなたの番になったら、実に「生産的な」作業を行います🤖、メニューを確認し、欲しいものを決め、好きな人😍の欲しいものを聞き、料金を支払い💵、現金またはカードを正しく渡したか確認し、正しく清算されたことを確認し、注文が正しく通っているかなどを確認します。

しかし、ハンバーガー🍔をまだできていないので、ハンバーガーの準備ができるまで待機⏸する必要があるため、レジ係👩🏻💻との作業は「一時停止⏸」になります。

しかし、カウンターから離れて、番号札を持ってテーブルに座っているときは、注意を好きな人😍に切り替えて🔗、その上で「仕事▶️🤖」を行なえます。その後、好きな人😍といちゃつくかのような、非常に「生産的な🤖」ことを再び行います。

次に、レジ係👩🏻💻は、「ハンバーガーの準備ができました🍔」と言って、カウンターのディスプレイに番号を表示しますが、表示番号があなたの番号に変わっても、すぐに狂ったように飛んで行くようなことはありません。あなたは自分の番号札を持って行って、他の人も自分の番号札があるので、あなたのハンバーガー🍔を盗む人がいないことは知っています。

なので、あなたは好きな人😍が話し終えるのを待って(現在の仕事▶️ / 処理中のタスクを終了します🤖)、優しく微笑んで、ハンバーガーを貰ってくるねと言います👍。

次に、カウンターへ、いまから完了する最初のタスク▶️へ向かい、ハンバーガー🍔を受け取り、感謝の意を表して、テーブルに持っていきます。これで、カウンターとのやり取りのステップ/タスクが完了しました👍。これにより、「ハンバーガーを食べる🔗▶️」という新しいタスクが作成されます。しかし、前の「ハンバーガーを取得する」というタスクは終了しました👍。

並列ハンバーガー

これらが「並行ハンバーガー」ではなく、「並列ハンバーガー」であるとしましょう。

あなたは好きな人👤と並列ファストフード🍔を買おうとしています。

列に並んでいますが、何人かの料理人兼、レジ係(8人としましょう)👤👤👤👤👤👤👤👤があなたの前にいる人達の注文を受けつけています。

8人のレジ係がそれぞれ自分で注文を受けるや否や、次の注文を受ける前にハンバーガーを準備するので、あなたの前の人達はカウンターを離れずに、ハンバーガー🍔ができるのを待っています🕒。

それからいよいよあなたの番になり、好きな人👤と自分のために、2つの非常に豪華なハンバーガー🍔を注文します。

料金を支払います💵。

レジ係はキッチンに行きます👤。

あなたはカウンターの前に立って待ちます🕒。番号札がないので誰もあなたよりも先にハンバーガー🍔を取らないようにします。

あなたと好きな人👤は忙しいので、誰もあなたの前に来させませんし、あなたのハンバーガーが到着したとき🕒に誰にも取ることを許しません。あなたは好きな人に注意を払えません😞。

これは「同期」作業であり、レジ係/料理人👤と「同期」します。レジ係/料理人👤がハンバーガー🍔を完成させてあなたに渡すまで待つ🕒必要があり、ちょうどその完成の瞬間にそこにいる必要があります。そうでなければ、他の誰かに取られるかもしれません。

その後、カウンターの前で長い時間待つ🕒から🕒、ついにレジ係/料理人👤がハンバーガー🍔を渡しに戻ってきます。

ハンバーガー🍔を取り、好きな人👤とテーブルに行きます。

ただ食べるだけ、それでおしまいです。🍔🍷。

ほとんどの時間、カウンターの前で待つのに費やされていたので🕒、あまり話したりいちゃつくことはありませんでした😞。

この並列ハンバーガーのシナリオでは、あなたは2つのプロセッサを備えたコンピュータ/プログラム🖥️(あなたとあなたの好きな人👤)であり、両方とも待機🕒していて、彼らは「カウンターで待機🕒」することに専念しています🎮。

ファストフード店には8つのプロセッサ(レジ係/料理人)👤👤👤👤👤👤👤👤があります。一方、並行ハンバーガー店には2人(レジ係と料理人)👤👤しかいなかったかもしれません。

しかし、それでも、最終的な体験は最高ではありません😞。

これは、ハンバーガー🍔の話と同等な話になります。

より「現実的な」例として、銀行を想像してみてください。

最近まで、ほとんどの銀行は複数の窓口👤👤👤👤に、行列🕒🕒🕒🕒🕒🕒🕒🕒🕒ができていました。

すべての窓口で、次々と、一人の客とすべての作業を行います👤🎮。

その上、長時間、列に並ばなければいけません🕒。そうしないと、順番が回ってきません。

銀行🏦での用事にあなたの好きな人👤を連れて行きたくはないでしょう。

ハンバーガーのまとめ

この「好きな人とのファストフードハンバーガー」のシナリオでは、待機🕒が多いため、並行システム🔄🔄🔄を使用の方がはるかに理にかなっています。

これは、ほとんどのWebアプリケーションに当てはまります。

多くのユーザーがいますが、サーバーは、あまり強くない回線でのリクエストの送信を待機🕒しています。

そして、レスポンスが返ってくるのをもう一度待機🕒します。

この「待機🕒」はマイクロ秒単位ですが、それでも、すべて合算すると、最終的にはかなり待機することになります。

これが、Web APIへの非同期🔄🔄🔄コードの利用が理にかなっている理由です。

ほとんどの既存の人気のあるPythonフレームワーク (FlaskやDjangoを含む) は、Pythonの新しい非同期機能ができる前に作成されました。したがって、それらをデプロイする方法は、並列実行と、新機能ほど強力ではない古い形式の非同期実行をサポートします。

しかし、WebSocketのサポートを追加するために、非同期Web Python (ASGI) の主な仕様はDjangoで開発されました。

そのような非同期性がNodeJSを人気にした理由です (NodeJSは並列ではありませんが)。そして、プログラミング言語としてのGoの強みでもあります。

そして、それはFastAPIで得られるパフォーマンスと同じレベルです。

また、並列処理と非同期処理を同時に実行できるため、テスト済みのほとんどのNodeJSフレームワークよりも高く、Goと同等のパフォーマンスが得られます。Goは、Cに近いコンパイル言語です ([Starletteに感謝します](#))。

並行は並列よりも優れていますか？

いや！それはこの話の教訓ではありません。

並行処理は並列処理とは異なります。多くの待機を伴う特定のシナリオに適しています。そのため、一般に、Webアプリケーション開発では並列処理よりもはるかに優れています。しかし、すべてに対してより良いというわけではありません。

なので、バランスをとるために、次の物語を想像して下さい：

あなたは大きくて汚れた家を掃除する必要があります。

はい、以上です。

待機🕒せず、家の中の複数の場所でたくさんの仕事をするだけです。

あなたはハンバーガーの例のように、最初はリビングルーム、次にキッチンのように順番にやっていくことができますが、何かを待機🕒しているわけではなく、ただひたすらに掃除をするだけで、順番は何にも影響しません。

順番の有無に関係なく (並行に) 同じ時間がかかり、同じ量の作業が行われることになるでしょう。

しかし、この場合、8人の元レジ係/料理人/現役清掃員🧹🧹🧹🧹🧹🧹🧹を手配できて、それぞれ (さらにあなたも) が家の別々の場所を掃除できれば、追加の助けを借りて、すべての作業を並列に行い、はるかに早く終了できるでしょう。

このシナリオでは、清掃員 (あなたを含む) のそれぞれがプロセッサとなり、それぞれの役割を果たします。

また、実行時間のほとんどは (待機ではなく) 実際の作業に費やされ、コンピュータでの作業はCPUによって行われます。これらの問題は「CPUバウンド」と言います。

CPUバウンド操作の一般的な例は、複雑な数学処理が必要なものです。

例えば:

- **オーディオ** や **画像処理**。
- **コンピュータビジョン**: 画像は数百万のピクセルで構成され、各ピクセルには3つの値/色があり、通常、これらのピクセルで何かを同時に計算する必要がある処理。
- **機械学習**: 通常、多くの「行列」と「ベクトル」の乗算が必要です。巨大なスプレッドシートに数字を入れて、それを同時に全部掛け合わせることを考えてみてください。
- **ディープラーニング**: これは機械学習のサブフィールドであるため、同じことが当てはまります。乗算する数字がある単一のスプレッドシートではなく、それらの膨大な集合で、多くの場合、それらのモデルを構築および/または使用するために特別なプロセッサを使用します。

並行処理 + 並列処理: Web + 機械学習

FastAPIを使用すると、Web開発で非常に一般的な並行処理 (NodeJSの主な魅力と同じもの) を利用できます。

ただし、機械学習システムのような **CPUバウンド** ワークロードに対して、**並列処理**とマルチプロセッシング (複数のプロセスが並列で実行される) の利点を活用することもできます。

さらに、Pythonが**データサイエンス**、機械学習、特にディープラーニングの主要言語であるという単純な事実により、FastAPIはデータサイエンス/機械学習のWeb APIおよびアプリケーション (他の多くのアプリケーションとの) に非常によく適合しています。



本番環境でこの並列処理を実現する方法については、[デプロイ](#)に関するセクションを参照してください。

async と await

現代的なバージョンのPythonには、非同期コードを定義する非常に直感的な方法があります。これにより、通常の「シーケンシャル」コードのように見え、適切なタイミングで「待機」します。

結果を返す前に待機する必要があり、これらの新しいPython機能をサポートする操作がある場合は、次のようにコーディングできます。

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

カギは `await` です。結果を `burgers` に保存する前に、`get_burgers(2)` の処理①の完了を待つ②必要があります。これをPythonに伝えます。これでPythonは、その間に (別のリクエストを受信するなど) 何か他のことができる   ことを知ります。

`await` が機能するためには、非同期処理をサポートする関数内にある必要があります。これは、`async def` で関数を宣言するだけでよいです:

```
async def get_burgers(number: int):
    # ハンバーガーを作成するために非同期処理を実行
    return burgers
```

... `def` のかわりに:

```
# 非同期ではない
def get_sequential_burgers(number: int):
    # ハンバーガーを作成するためにシーケンシャルな処理を実行
    return burgers
```

`async def` を使用すると、Pythonにその関数内で `await` 式(その関数の実行を「一時停止」し、結果が戻るまで他の何かを実行する)を認識しなければならないと伝えることができます。

`async def` 関数を呼び出すときは、「`await`」しなければなりません。したがって、これは機能しません:

```
# get_burgersはasync defで定義されているので動作しない
burgers = get_burgers(2)
```

したがって、`await` で呼び出すことができるライブラリを使用している場合は、次のように `async def` を使用して、それを使用する *path operation* 関数を作成する必要があります:

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

より発展的な技術詳細

`await` は `async def` で定義された関数内でのみ使用できることがわかったかと思います。

しかし同時に、`async def` で定義された関数は「`await`される」必要があります。なので、`async def` を持つ関数は、`async def` で定義された関数内でのみ呼び出せます。

では、このニワトリと卵の問題について、最初の `async` 関数をどのように呼び出すのでしょうか？

FastAPIを使用している場合、その「最初の」関数が *path operation* 関数であり、FastAPIが正しく実行する方法を知っているので、心配する必要はありません。

しかし、FastAPI以外で `async` / `await` を使用したい場合は、[公式Pythonドキュメントを参照して下さい](#)。

非同期コードの他の形式

`async` と `await` を使用するスタイルは、この言語では比較的新しいものです。

非同期コードの操作がはるかに簡単になります。

等価な(またはほとんど同一の)構文が、最近のバージョンのJavaScript (ブラウザおよびNodeJS) にも最近組み込まれました。

しかし、その前は、非同期コードの処理はかなり複雑で難解でした。

以前のバージョンのPythonでは、スレッドや[Gevent](#)が利用できました。しかし、コードは理解、デバッグ、そして、考察がはるかに複雑です。

以前のバージョンのNodeJS / ブラウザJavaScriptでは、「コールバック」を使用していました。これは、[コールバック地獄](#)につながります。

コルーチン

コルーチンは、`async def` 関数によって返されるものを指す非常に洒落た用語です。これは、開始できて、いつか終了する関数のようなものであるが、内部に `await` があるときは内部的に一時停止🛑されることもあるものだとしてPythonは認識しています。

`async` と `await` を用いた非同期コードを使用するすべての機能は、「コルーチン」を使用するものとして何度もまとめられています。Goの主要機能である「ゴルーチン」に相当します。

まとめ

上述したフレーズを見てみましょう：

現代版のPythonは「**非同期コード**」を、「**コルーチン**」と称されるものを利用してサポートしています。これは `async` と `await` 構文を用います。

今では、この意味がより理解できるはずです。🌟

(Starletteを介して) FastAPIに力を与えて、印象的なパフォーマンスを実現しているものはこれがすべてです。

非常に発展的な技術的詳細

!!! warning "注意" 恐らくスキップしても良いでしょう。

この部分は**FastAPI**の仕組みに関する非常に技術的な詳細です。

かなりの技術知識（コルーチン、スレッド、ブロッキングなど）があり、FastAPIが`async def`と通常の`def`をどのように処理するか知りたい場合は、先に進んでください。

Path operation 関数

`path operation` 関数を `async def` の代わりに通常の `def` で宣言すると、(サーバーをブロックするので) 直接呼び出す代わりに外部スレッドプール(`await`される)で実行されます。

上記の方法と違った方法の別の非同期フレームワークから来ており、小さなパフォーマンス向上(約100ナノ秒)のために通常の `def` を使用して些細な演算のみ行う `path operation` 関数を定義するのに慣れている場合は、FastAPIではまったく逆の効果になることに注意してください。このような場合、`path operation` 関数がブロッキング🛑を実行しないのであれば、`async def` の使用をお勧めします。

それでも、どちらの状況でも、FastAPIが過去のフレームワークよりも(またはそれに匹敵するほど) [高速になる](#) {internal-link target=_blank}可能性があります。

依存関係

依存関係についても同様です。依存関係が `async def` ではなく標準の `def` 関数である場合、外部スレッドプールで実行されます。

サブ依存関係

(関数定義のパラメーターとして) 相互に必要な複数の依存関係とサブ依存関係を設定できます。一部は `async def` で作成され、他の一部は通常の `def` で作成されます。それでも動作し、通常の `def` で作成されたものは、「`await`される」代わりに(スレッドプールから)外部スレッドで呼び出されます。

その他のユーティリティ関数

あなたが直接呼び出すユーティリティ関数は通常の `def` または `async def` で作成でき、FastAPIは呼び出す方法に影響を与えません。

これは、FastAPIが呼び出す関数と対照的です: *path operation 関数* と依存関係。

ユーティリティ関数が `def` を使用した通常関数である場合、スレッドプールではなく直接 (コードで記述したとおりに) 呼び出されます。関数が `async def` を使用して作成されている場合は、呼び出す際に `await` する必要があります。

繰り返しになりますが、これらは非常に技術的な詳細であり、検索して辿り着いた場合は役立つでしょう。

それ以外の場合は、上記のセクションのガイドラインで問題ないはずです: [急いでいますか？](#)。