並行処理・並列処理と非同期処理

J23072　蝦名武尊

# はじめに

このドキュメントの目的は並列処理（マルチスレッド）と並行処理（マルチプロセス）やその動きについて具体例を用いてまとめることである。また、非同期処理のプロセスの動きについても触れていく。

# 並列処理と並行処理

並列処理とは複数の処理をコンピュータのプロセッサやコアなど複数の作業者を利用して同時に行う処理である。並列処理は大量の数値計算処理などCPUリソースを大量に消費するような処理（CPUバウンドな処理）を行う際に実装される。

また、単一の作業者が処理を実行し、その待ち時間中に別の処理を行う処理を並行処理と呼ぶ。並行処理はWebAPIの利用やディスクへの書き込みなど特に待ち時間が長い処理(I/Oバウンドな処理)を行う場合に向いている。

## マルチプロセスとマルチスレッド

並列処理や並行処理をコンピュータ上に実装するには「マルチプロセス」と「マルチスレッド」という方法が存在する。

マルチプロセスとは、複数のプロセスを同時に使用することである。プロセスとはコンピュータ上で動いている処理の単位であり、基本的に一つのプログラムに対して一つのプロセスが立ち上がる。このプロセスを複数起動させそれぞれに処理を行わせるという方法である。この方法では並列処理を実現することができる。

マルチスレッドとは、複数のスレッドを同時に使用することである。スレッドとはプロセスの中をさらに分割した処理の単位であり、一つのプログラムに対し必要に応じて複数のスレッドが立ち上がる。この方法では並行処理を実現することができる。

## 実装方法

　今回はPythonでの実装方法についてまとめていく。

　Pythonではマルチプロセスを実装するモジュール「multprocessing」とマルチスレッドを実装するモジュール「threading」が存在する。また、Python3.2以降は両方のモジュールをまとめたモジュール「concurrent.futures」が追加された。

### マルチスレッド

ここではマルチスレッドを実装した並行処理について具体例を混ぜて説明する。

from concurrent.futures import ThreadPoolExecutor

import time

def func\_1():

    for n in range(3):

        time.sleep(2)

        print(f'func\_1 - {n}')

def func\_2():

    for n in range(3):

        time.sleep(1)

        print(f'func\_2 - {n}')

def main():

    with ThreadPoolExecutor(max\_workers=2) as executor:

        executor.submit(func\_1)

        executor.submit(func\_2)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

上はマルチスレッドを実装したプログラムの具体例である。プログラムの内容はそれぞれ一定間隔で数値を表示するfunc\_1関数とfunc\_2関数を並行処理で実行するというものである。

このプログラムで特に重要な点を以下に示す。

from concurrent.futures import ThreadPoolExecutor

並行処理を実装するためのモジュール「ThreadPoolExecutor」をインポートし、マルチスレッドを実装するために必要なモジュールを導入する。

def main():

    with ThreadPoolExecutor(max\_workers=2) as executor:

executor.submit(func\_1)

        executor.submit(func\_2)

ThreadPoolExecutorでマルチスレッドの実装を行い、引数である”max\_workers”で並行処理で同時に動作する処理の最大数を指定する。その後、executor.submit()で並行処理で処理する関数を指定する。

func\_2 - 0

func\_1 - 0

func\_2 - 1

func\_2 - 2

func\_1 - 1

func\_1 - 2

　　　このプログラムを実行すると上のような結果が表示される。func\_1関数の停止時間が2秒、func\_2関数が1秒のため、func\_1関数がsleepしている待ち時間にfunc\_2関数が処理を行っている。

### マルチプロセス

次にマルチプロセスを実装した並行処理について具体例を混ぜて説明する。

from concurrent.futures import ProcessPoolExecutor

import time

def func\_1():

    for n in range(3):

        time.sleep(2)

        print(f'func\_1 - {n}')

def func\_2():

    for n in range(3):

        time.sleep(1)

        print(f'func\_2 - {n}')

def main():

    with ProcessPoolExecutor(max\_workers=2) as executor:

        executor.submit(func\_1)

        executor.submit(func\_2)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

上はマルチプロセスを実装したプログラムの具体例である。プログラムの内容はそれぞれ一定間隔で数値を表示するfunc\_1関数とfunc\_2関数を並行処理で実行するというものである。

このプログラムで特に重要な点を以下に示す。

from concurrent.futures import ProcessPoolExecutor

並列処理を実装するためのモジュール「ProcessPoolExecutor」をインポートし、マルチプロセスを実装するために必要なモジュールを導入する。

def main():

    with ProcessPoolExecutor(max\_workers=2) as executor:

        executor.submit(func\_1)

        executor.submit(func\_2)

ProcessPoolExecutorでマルチプロセスの実装を行い、引数である”max\_workers”で並列処理で同時に動作する処理の最大数を指定する。その後、executor.submit()で並列処理で処理する関数を指定する。

func\_2 - 0

func\_1 - 0

func\_2 - 1

func\_2 - 2

func\_1 - 1

func\_1 - 2

このプログラムを実行すると上のような結果が表示される。func\_1関数の停止時間が2秒、func\_2関数が1秒であるが並列処理のためそれぞれの待ち時間が互いの動作に影響をもたらすことはなく、独立して動作している。

# 非同期処理

　非同期処理とは、直前の処理の完了を待つことなく別の処理を走らせることである。一般的な処理方法である逐次処理とは異なり、処理を同時に走らせることができるため、処理の完了まで時間が掛かる場合、特にPythonではI/Oバウンドなど入出力に関わる処理で使用される。

## 3.1. プロセスの動き

　　　ここでは非同期処理の実装に使用される”await”や”task”、”gather”、run\_in\_excutor”について使用例を混ぜながら解説を行う。

### 3.2.1 await

awaitとは非同期関数（asyncなど）の内部でのみ利用することができる構文である。awaitを呼び出すことでその時点で処理が一時停止し、指定したオブジェクトの完了を待つ。この間に他の処理を行ったり、処理の終了を待機したりと柔軟なプログラムを作成可能である。

以下、具体例である。

import asyncio

import time

async def main():

    print(f"main開始 {time.strftime('%X')}")

    await asyncio.sleep(1)

    await asyncio.sleep(2)

    print(f"main終了 {time.strftime('%X')}")

if \_\_name\_\_ == "\_\_main\_\_":

    asyncio.run(main())

プログラムの内容はsleepで3秒待ち、処理の開始と終了の時刻を表示するというものである。

async def main():

    print(f"main開始 {time.strftime('%X')}")

    await asyncio.sleep(1)

    await asyncio.sleep(2)

    print(f"main終了 {time.strftime('%X')}")

awaitを使用するにはasyncの宣言が必要である。また、awaitによって３秒の待機時間が発生している間は他の処理が停止している。処理が完了し次第、awaitの直後から処理を再開する。

### 3.2.2 task

　　　taskとはコルーチンを並行処理で動作させるために必要な構文である。Pythonではasyncio.create\_task()を利用して実装し、awaitで呼び出すことができる。

import asyncio

import time

async def main():

    print(f"main開始 {time.strftime('%X')}")

    task1 = asyncio.create\_task(asyncio.sleep(1))

    task2 = asyncio.create\_task(asyncio.sleep(2))

    await task1

    await task2

    print(f"main終了 {time.strftime('%X')}")

if \_\_name\_\_ == "\_\_main\_\_":

    asyncio.run(main())

　　　このプログラムは1秒スリープするtask1と2秒スリープするtask2をtaskにて並行処理するものである。

　　　asyncio.create\_task()にて指定された指示をそれぞれawait構文にて同時に実行しており、発生するスリーブ期間は2秒となっている。

### 3.2.3 gather

gatherとは複数のコルーチンやタスクを並行処理で動作させるために必要な構文である。また、処理したコルーチンやタスクの戻り値をまとめて受け取ることができる。Pythonではasyncio.gather()をを利用して実装する。

import asyncio

import time

async def function(sec):

    print(f"{sec}秒待つ")

    await asyncio.sleep(sec)

    return f"{sec}秒の待機に成功"

async def main():

    print(f"main開始 {time.strftime('%X')}")

    task1 = asyncio.create\_task(function(1))

    results = await asyncio.gather(function(2),task1)

    print(results)

    print(f"main終了 {time.strftime('%X')}")

if \_\_name\_\_ == "\_\_main\_\_":

    asyncio.run(main())

このプログラムはコルーチンfunctionとタスクtaskをgatherで並行処理し、同時に戻り値を受け取るものである。

asyncio.gather()で指定されたコルーチンやタスクは並行処理にて処理され、変数resultsに同時に格納されている。

### 3.2.4 run\_in\_executor

run\_in\_executorは通常awaitで使用できない通常の関数を非同期処理で処理するために使用される。run\_in\_executorを使用するにはasycio.get\_running\_loop()でその時点でスレッドで動いているイベントループを取得する。

import asyncio

import time

async def function(sec):

    print(f"{sec}秒待つ")

    loop = asyncio.get\_running\_loop()

    await loop.run\_in\_executor(None, time.sleep, sec)

    return f"{sec}秒の待機に成功"

async def main():

    print(f"main開始 {time.strftime('%X')}")

    results = await asyncio.gather(function(2),function(1))

    print(results)

    print(f"main終了 {time.strftime('%X')}")

if \_\_name\_\_ == "\_\_main\_\_":

    asyncio.run(main())

このプログラムではsleepをasyncioではなく通常の関数であるtimeで呼び出し、非同期処理を行うものである。

run\_in\_executorにおいて第一引数は使用するexecutorが指定され、今回与えられた”None”ではデフォルトで”ThreadPoolExecutor”が使用される。第二引数は使用する関数、第三引数は関数に渡す引数を与える。

# おわりに

　　　以上のようなモジュールや関数を使用することにより並列処理や並行処理、非同期処理を実装することができる。各手法はそれぞれ特性を持っており、並列処理はCPU のリソースを大量に使用するCPUバウンドな処理が、並行処理は特に待ち時間が長いI/Oバウンドな処理を行う際に選ばれるべきである。そして、複雑な待機時間を制御する場合は非同期処理が最適である。

# 参考文献

・Pythonプログラミング VTuber サプー.” **【Pythonプログラミング】並列処理の基本を解説！マルチスレッド・マルチプロセスをconcurrent futuresで実装！.**”Youtube, 2022/04/24, <https://youtu.be/et-cDFbVkQw?si=pp6YpLzrFSWPpB6t>

・Pythonプログラミング VTuber サプー.” **【非同期処理】Pythonの async / await 構文を使ってみよう！**

**.**”Youtube, 2023/08/19, <https://youtu.be/KnNWw5HX20w?si=3yDnZ9POY0sz5HL9>