【Python】並行処理

目次

マルチスレッド

■threadingモジュールによる

- ▶ ※ マルチスレッドを行うときにグローバルなデータを書き換えるのは避けよう。
- ▶ ※ マルチスレッドはI/Oバウンド問題の解決には有効だが、CPUバウンド問題には不適。
- ▶ ※ マルチスレッドを行う場合、プロセスは1個に限定するのがよい。
- ▶ threading モ
- ▶ スレを作成
- ▶ スレの活動を開始
- ▶ スレが終了するまで待つ

- ▶ ロックオブ
- ▶ ある処理を他スレッドと競合せずに行う
- ▶ ☆ threding モジュールでスロットリングを実装する

■multiprocessing.dummyモジュールによる

- ▶ multiprocessing.dummy モ
- ▶ プールを作成
- ▶ ※ あとは multioprocessing モジュールと同じ。

■concurrent.futuresモジュールによる

- ▶ concurrent.futures モ
- ▶ プールを作成
- ▶ ※ あとはマルチプロセスの場合と同じ。

【Python】並行処理

目次

- マルチスレッド
- マルチプロセス
- 非同期I/O

マルチスレッド

■threadingモジュールによる

- ▶ ※ マルチスレッドを行うときにグローバルなデータを書き換えるのは避けよう。
- ▶ ※ マルチスレッドはI/Oバウンド問題の解決には有効だが、CPUバウンド問題には不適。
- ▶ ※ マルチスレッドを行う場合、プロセスは1個に限定するのがよい。

- ▶ threading モ　　import threading
- ▶ スレを作成　　thread = Thread(target=*func*, args=(*arg1*, *arg2*, ...))
- ▶ スレの活動を開始　　thread.start() #
- ▶ スレが終了するまで待つ　　thread.join() #

- ▶ ロックオブ　　lock = threading.Lock()　　※ メインスレにて
- ▶ ある処理を他スレッドと競合せずに行う　　・lock.acquire()　処理　lock.release()　※ ワーカースレにて
・with lock:↲　処理　※ 〃
- ▶ ☆ threding モジュールでスロットリングを実装する

■multiprocessing.dummyモジュールによる

- ▶ multiprocessing.dummy モ　　import multiprocessing.dummy
- ▶ プールを作成　　pool = multiprocessing.dummy.Pool(*maxProcesses*)
- ▶ ※ あとは multioprocessing モジュールと同じ。

■concurrent.futuresモジュールによる

- ▶ concurrent.futures モ　　import concurrent.futures as cf　　※ `cf` は我流
- ▶ プールを作成　　executor = cf.ThreadPoolExecutor(max_workers=*n*)
- ▶ ※ あとはマルチプロセスの場合と同じ。

マルチプロセス

■multiprocessingモジュールによる

- ▶ multiprocessing モ
- ▶ プールを作成
- ▶ *func*の呼出しを
  なる早で予約
- ▶ **results**から完了した順に取得
- ▶ ※ 別モジュールでマルチプロセスを実装したのを読み込む場合、`if __name__ == '__main__':` を忘れずに。そしてできれば `mp.freeze_support()` も行う。
- ▶ ☆ 用例

■concurrent.futuresモジュールによる

- ▶ concurrent.futures モ
- ▶ プールを作成
- ▶ *func*の呼出しを
  なる早で予約
- ▶ フュの結果値
- ▶ フュのイテオブから完了
  した順に結果値を取得
- ▶ **results**から完了した順に取得
- ▶ プールの全リソースを開放

マルチプロセス

■multiprocessingモジュールによる

- ▶ multiprocessing モ　import multiprocessing as mp
- ▶ プールを作成　pool = mp.Pool(*maxProcesses*)
- ▶ *func*の呼出しを　results = pool.map(*func*, [(*arg1*$_1$, *arg2*$_1$, ...), (*arg1*$_2$, *arg2*$_2$, ...), ...])
  なる早で予約　や　results = pool.map(*func*, [*arg1*$_1$, *arg1*$_2$, ...])
- ▶ **results**から完了した順に取得　for result in results:↲　result を使った処理（任意）
- ▶ ※ 別モジュールでマルチプロセスを実装したのを読み込む場合、`if __name__ == '__main__':` を忘れずに。そしてできれば `mp.freeze_support()` も行う。
- ▶ ☆ 用例

■concurrent.futuresモジュールによる

- ▶ concurrent.futures モ　import concurrent.futures as cf　※ `cf` は我流
- ▶ プールを作成　executor = cf.ProcessPoolExecutor(max_workers=*n*)
- ▶ *func*の呼出しを　・future= executor.submit(*func*, *arg1*, *arg2*, ...)
  なる早で予約　・**results** = executor.map(*func*, [(*arg1*$_1$, *arg2*$_1$, ...), (*arg1*$_2$, *arg2*$_2$, ...), ...])
- ▶ フュの結果値　future.result()
- ▶ フュのイテオブから完了　for future in cf.as_completed([future0, future1, ...]):↲
  した順に結果値を取得　future.result() を使った処理（任意）
- ▶ **results**から完了した順に取得　for result in results:↲　result を使った処理（任意）
- ▶ プールの全リソースを開放　executor.shutdown() #

非同期I/O

■はじめに

- ▶ asyncio モ
- ▶ ※ asyncioは、イベントループが中核、コルーチンが基本単位、として考えるのがよい。フューチャーオブジェクトやタスクオブジェクトを基本単位として見るべきではない。
- ▶ ☆ 基本の形

■イベントループ

- ▶ ※ イベントループはasyncioで中核をなす存在。非同期プログミングの中心要素は協調的マルチタスクであるが、協調的マルチタスクが行われる場合、1プロセス、1スレッドのなかで

非同期I/O

■はじめに

- ▶ asyncio モ　import asyncio
- ▶ ※ asyncioは、イベントループが中核、コルーチンが基本単位、として考えるのがよい。フューチャーオブジェクトやタスクオブジェクトを基本単位として見るべきではない。
- ▶ ☆ 基本の形

■イベントループ

- ▶ ※ イベントループはasyncioで中核をなす存在。非同期プログミングの中心要素は協調的マルチタスクであるが、協調的マルチタスクが行われる場合、1プロセス、1スレッドのなかで

ある1つの関数が複数のタスクの実行を制御する。asyncioにおけるこの関数の実装にあたるのがイベントループである。

- (現在の) イベルを取得
- 半永久的にイベルを実行
- *awaitable*の完了まで〃
- あるプールでの*func*呼出しを予約
- イベルを停止
- イベルを閉じる
- *awaitable*の完了までイベルを実行して停止も行う

■コールバック

- ※ コールバックとは、将来のあるタイミングに実行されることを意図した関数。
- コルバ呼出しをなる早で予約
- 〃 を*delay*秒後 (以降) に予約
- 現在時刻を浮動小数点数で取得
- 予約したコルバの予約を取り消し

■Awaitableオブジェクト全般

- ※ Awaitableオブジェクトとは、`await` 式の中で使うことができるオブジェクト。
- ※ Awaitableオブジェクト：コルーチンオブジェクト、フューチャーオブジェクト、タスクオブジェクト
- ※ `await awaitable` によって、`awaitable` がラップしている（あるいはそのものである）コルーチンオブジェクトの実行を一時停止することができる。ただし、**何らかのコルーチン関数の定義内でしか使えない。**

■コルーチン関数、コルーチンオブジェクト

- ※ コルーチン関数、コルーチンオブジェクトは密接に関連した概念であり、2つまとめて「コルーチン」と呼ぶこともある。しかしこのように総称としてではなく、どちらかのことを指して単に「コルーチン」と呼ぶことがほとんどで、文脈に応じてその意味を判断する必要がある。
- コルチ関数を定義
- ※ コルーチンオブジェクトは、コルーチン関数を呼び出すと返ってくるオブジェクト。
- ※ コルーチンオブジェクトを実際に実行させるには、`await` を前につけるか（何らかのコルーチン関数の定義内である必要）、タスクオブジェクトでラップしてイベントループに渡す（ちなみに `loop.run_until_complete(coroObj)` では直接コルーチンオブジェクトを渡せているかに見えるが実は内部で暗黙的にタスクオブジェクト化されている）。

ある1つの関数が複数のタスクの実行を制御する。asyncioにおけるこの関数の実装にあたるのがイベントループである。

- (現在の) イベルを取得　loop = asyncio.get_event_loop()
- 半永久的にイベルを実行　loop.run_forever() #
- *awaitable*の完了まで〃　loop.run_until_complete(*awaitable*)　※: `awaitable` の結果値
- あるプールでの*func*呼出しを予約　loop.run_in_executor(executor, *func*, *arg1*, ...)　※: フュ
- イベルを停止　loop.stop() #　※その時点で予約済みの全コルバを実行してから停止
- イベルを閉じる　loop.close() #　※イベルは停止済みである必要あり
- *awaitable*の完了までイベルを実行して停止も行う　asyncio.run(*awaitable*)

■コールバック

- ※ コールバックとは、将来のあるタイミングに実行されることを意図した関数。
- コルバ呼出しをなる早で予約　loop.call_soon(*callback*, *arg1*, *arg2*, ...)　※: ハンドル
- 〃 を*delay*秒後 (以降) に予約　loop.call_later(*delay*, *callback*, *arg1*, ...)　※: 〃
- 現在時刻を浮動小数点数で取得　loop.time()
- 予約したコルバの予約を取り消し　コルバ予約時に返されたハンドル.cancel() #

■Awaitableオブジェクト全般

- ※ Awaitableオブジェクトとは、`await` 式の中で使うことができるオブジェクト。
- ※ Awaitableオブジェクト：コルーチンオブジェクト、フューチャーオブジェクト、タスクオブジェクト
- ※ `await awaitable` によって、`awaitable` がラップしている（あるいはそのものである）コルーチンオブジェクトの実行を一時停止することができる。ただし、**何らかのコルーチン関数の定義内でしか使えない。**

■コルーチン関数、コルーチンオブジェクト

- ※ コルーチン関数、コルーチンオブジェクトは密接に関連した概念であり、2つまとめて「コルーチン」と呼ぶこともある。しかしこのように総称としてではなく、どちらかのことを指して単に「コルーチン」と呼ぶことがほとんどで、文脈に応じてその意味を判断する必要がある。
- コルチ関数を定義　async def hoge(p1, ...):↲　· ·
- ※ コルーチンオブジェクトは、コルーチン関数を呼び出すと返ってくるオブジェクト。
- ※ コルーチンオブジェクトを実際に実行させるには、`await` を前につけるか（何らかのコルーチン関数の定義内である必要）、タスクオブジェクトでラップしてイベントループに渡す（ちなみに `loop.run_until_complete(coroObj)` では直接コルーチンオブジェクトを渡せているかに見えるが実は内部で暗黙的にタスクオブジェクト化されている）。

■フューチャーオブジェクト

- ※ フューチャー (`asyncio.Future`) オブジェクトは非同期処理の最終結果を表すオブジェクト。
- (普通の) フュの生成

- フュが完了しているか
- フュがキャンセル済みか

- フュに値を設定し完了させる
- フュに例外を設定し〃
- フュをキャンセル

- フュの結果値
- フュに設定された例外

- ※ フュに例外が設定されているにもかかわらず、例外を取得することなくイベントループを閉じてしまうと、`Future exception was never retrieved` と表示が出た。

- ※ フューチャーオブジェクトには、**フューチャーオブジェクトが完了したときに実行させたいコールバック**（ここでは完了コールバックと呼びたい）としてコールバックを複数設定できる。
- 完了コルバに追加
- 〃から削除

- イテオブ内のフュが全て完了したら完了になるコルチオブ

■タスクオブジェクト

- ※ タスクオブジェクトは、`asyncio.Future` の**サブクラス** `asyncio.Task` のインスタンス。コルーチンオブジェクトをラップしたオブジェクトで、コルーチンをイベントループに実行させるのにこれを介することになる。
- タスを作成
- タスの名前を設定・取得
- ※ **継承してはいる**ものの、フューチャーオブジェクトのメソッドのうち `.set_result()` と `.set_exception()` だけは使えない。ただ、それ以外のメソッドはすべて使える。

■フューチャーオブジェクト

- ※ フューチャー (`asyncio.Future`) オブジェクトは非同期処理の最終結果を表すオブジェクト。
- (普通の) フュの生成　fut = loop.create_future()

- フュが完了しているか　fut.done()
- フュがキャンセル済みか　fut.cancelled()

- フュに値を設定し完了させる　fut.set_result(値)
- フュに例外を設定し〃　fut.set_exception(例外)
- フュをキャンセル　fut.cancel() #

- フュの結果値　fut.result()　※: 結果値; エラー
- フュに設定された例外　fut.exception()　※: 例外か `None` ; エラー

- ※ フュに例外が設定されているにもかかわらず、例外を取得することなくイベントループを閉じてしまうと、`Future exception was never retrieved` と表示が出た。

- ※ フューチャーオブジェクトには、**フューチャーオブジェクトが完了したときに実行させたいコールバック**（ここでは完了コールバックと呼びたい）としてコールバックを複数設定できる。
- 完了コルバに追加　fut.add_done_callback(*callback*※) # ※ `fut` を引数に呼び出される
- 〃から削除　fut.remove_done_callback(*callback*) #

- イテオブ内のフュが全て完了したら完了になるコルチオブ
  - asyncio.wait(*iterable*※[1])　※[1] 要素はフュ (当然タスも可)
    ※(実行後): `(完了済みフュの集合, 未完了フュの集合)`
  - asyncio.gather(*iterable*※[1])　※(実行後): `[フュ0の結果値, …]`

■タスクオブジェクト

- ※ タスクオブジェクトは、`asyncio.Future` の**サブクラス** `asyncio.Task` のインスタンス。コルーチンオブジェクトをラップしたオブジェクトで、コルーチンをイベントループに実行させるのにこれを介することになる。
- タスを作成
  - task = loop.create_task(*coroObj*)
  - task = asyncio.create_task(coroObj)
  - task = asyncio.ensure_future(*coroObj*)
- タスの名前を設定・取得　task.set_name(*name*) #　・　task.get_name()
- ※ **継承してはいる**ものの、フューチャーオブジェクトのメソッドのうち `.set_result()` と `.set_exception()` だけは使えない。ただ、それ以外のメソッドはすべて使える。