Multi-Threading の使い方

sugayu

2025 年 5 月 21 日

1　いつ使うのか

Python には GIL があるので、複数のスレッドを立てても実質的に一つのプロセス上でしか動かない。マルチスレッドが役に立つのは、GIL が開放されるとき、すなわちファイルアクセスやインターネットを介したデータのやりとりなどのブロッキング I/O が発生するときである。

2　マルチスレッドのやり方

簡単にマルチスレッドを実現するのであれば **ThreadPoolExecutor** が推奨される。他に **threading** などのパッケージもあるが、これらは低級な関数を提供するので、高度なスレッド操作が求められるとき以外は使うべきではない。

3　マルチスレッドとキューを同時に使いたいとき

ThreadPoolExecutor は内部で **queue.Queue** を使用しているので、**queue.Queue** を使用せず簡単に「スレッド数」と「タスク数」を別々に指定することができる。

しかしパイプラインのように、作業 1 が完了したあとにのみ作業 2 を開始できる、のような一連の作業群を並行に処理したい場合には **queue.Queue** が必要となる。ここでは、作業 1 の結果をキューに投げ、作業 2 用のスレッドはキューの中身があると動き出す、というものを想定する。

以下の例では作業 1, 2 として **sleep1**, **sleep2** という関数を用意し、Queue を使うことで二つの関数を連動させた。作業の回数は事前に **ntask** で指定されており、作業 1 も作業 2 も同じ回数だけ作業を行うと仮定した。

```python
from concurrent.futures import ThreadPoolExecutor
import threading
from queue import Queue
from time import sleep
from logging import getLogger
from numpy.random import default_rng

logger = getLogger(__name__)


##
def main() -> None:
    queue: Queue = Queue()

    rng = default_rng(222)
    ntasks = 10
    times = rng.uniform(1, 3, ntask)
```

```
    futures1, futures2 = [], []
    with ThreadPoolExecutor(3) as executor1, ThreadPoolExecutor(3) as executor2:
        for time in times:
            future = executor1.submit(sleep1, time, queue)
            futures1.append(future)

        for _ in times:
            future = executor2.submit(sleep2, queue)
            futures2.append(future)

        for future in futures1:
            logger.info(f'Result1: {future.result():.2f}')
        for future in futures2:
            logger.info(f'Result2: {future.result():.2f}')

    logger.info(f'Have all tasks been done?: {queue.empty()}')
    return


def sleep1(time: float, queue: Queue) -> float:
    '''mock I/O'''
    logger.info(f'Sleep1 in {threading.current_thread().name}: {time:.2f}')
    sleep(time)
    queue.put(time)
    return time


def sleep2(queue: Queue) -> float:
    '''mock I/O after sleep2'''
    time = queue.get()
    logger.info(f'Sleep2 in {threading.current_thread().name}: {time:.2f}')
    sleep(time)
    return time


if __name__ == '__main__':
    from sugayutils.log import mylogconfig

    mylogconfig()

    main()
```

結果から **sleep1** と **sleep2** が並行に動いていることが読み取れる。

```
[11:01:16,950] sleep1() 1.43: INFO - Sleep1 in ThreadPoolExecutor-0_0: 1.75
[11:01:16,950] sleep1() 1.43: INFO - Sleep1 in ThreadPoolExecutor-0_1: 2.41
[11:01:16,950] sleep1() 1.43: INFO - Sleep1 in ThreadPoolExecutor-0_2: 2.44
[11:01:18,707] sleep1() 1.43: INFO - Sleep1 in ThreadPoolExecutor-0_0: 2.73
[11:01:18,707] main() 1.33: INFO - Result1: 1.75
```

```
[11:01:18,707] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_0: 1.75
[11:01:19,361] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_1: 2.79
[11:01:19,361] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_1: 2.41
[11:01:19,361] main() l.33: INFO - Result1: 2.41
[11:01:19,399] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_2: 1.16
[11:01:19,399] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_2: 2.44
[11:01:19,399] main() l.33: INFO - Result1: 2.44
[11:01:20,562] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_2: 2.11
[11:01:20,562] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_0: 1.16
[11:01:21,444] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_0: 2.06
[11:01:21,444] main() l.33: INFO - Result1: 2.73
[11:01:21,723] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_0: 2.73
[11:01:22,155] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_1: 2.11
[11:01:22,156] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_1: 2.79
[11:01:22,156] main() l.33: INFO - Result1: 2.79
[11:01:22,156] main() l.33: INFO - Result1: 1.16
[11:01:22,676] sleep1() l.43: INFO - Sleep1 in ThreadPoolExecutor-0_2: 1.96
[11:01:22,676] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_2: 2.11
[11:01:22,676] main() l.33: INFO - Result1: 2.11
[11:01:23,511] main() l.33: INFO - Result1: 2.06
[11:01:24,271] main() l.33: INFO - Result1: 2.11
[11:01:24,458] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_0: 2.06
[11:01:24,637] main() l.33: INFO - Result1: 1.96
[11:01:24,638] main() l.35: INFO - Result2: 1.75
[11:01:24,638] main() l.35: INFO - Result2: 2.41
[11:01:24,638] main() l.35: INFO - Result2: 2.44
[11:01:24,638] main() l.35: INFO - Result2: 1.16
[11:01:24,638] main() l.35: INFO - Result2: 2.73
[11:01:24,789] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_2: 2.11
[11:01:24,950] sleep2() l.52: INFO - Sleep2 in ThreadPoolExecutor-1_1: 1.96
[11:01:24,950] main() l.35: INFO - Result2: 2.79
[11:01:24,950] main() l.35: INFO - Result2: 2.11
[11:01:26,528] main() l.35: INFO - Result2: 2.06
[11:01:26,903] main() l.35: INFO - Result2: 2.11
[11:01:26,911] main() l.35: INFO - Result2: 1.96
[11:01:26,911] main() l.37: INFO - Have all tasks been done?: True
```

作業 2 にとって事前に作業回数が分からない場合には工夫が必要になる。作業 1 が終了したら終了信号を `queue` に格納し、作業 2 は終了信号を受信したら作業を終了する。後続に作業 3 が存在した場合には、作業 2 の終了後に出力キューに終了信号を格納する。これらの作業をメソッドを通じて簡単にできるためのクラスを作っておくと便利である (例えば Slatkin ”Effective Python 2nd Ed.” 項目 55 の `ClosableQueue` が参考になる)。