17.4. concurrent.futures – 並列タスク実行 (原文)

バージョン 3.2 で追加.

ソースコード: Lib/concurrent/futures/thread.py および Lib/concurrent/futures/process.py

---

concurrent.futures モジュールは、非同期に実行できる呼び出し可能オブジェクトの高水準のインタフェースを提供します。

非同期実行は ThreadPoolExecutor を用いてスレッドで実行することも、 ProcessPoolExecutor を用いて別々のプロセスで実行することもできます. どちらも Executor 抽象クラスで定義された同じインターフェースを実装します。

17.4.1. Executor オブジェクト (原文)

class concurrent.futures.**Executor** (原文)

非同期呼び出しを実行するためのメソッドを提供する抽象クラスです。このクラスを直接使ってはならず、具象サブクラスを介して使います。

submit(*fn*, **args*, ***kwargs*) (原文)

呼び出し可能オブジェクト *fn* を、 fn(*args **kwargs) として実行するようにスケジュールし、呼び出し可能オブジェクトの実行を表現する Future オブジェクトを返します。

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*) (原文)

Similar to map(func, *iterables) except:

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

The returned iterator raises a concurrent.futures.TimeoutError if __next__() is called and the result isn't available after *timeout* seconds from the original call to Executor.map(). *timeout* can be an int or a float. If *timeout* is not specified or None, there is no limit to the wait time.

If a *func* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using ProcessPoolExecutor, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With ThreadPoolExecutor, *chunksize* has no effect.

バージョン 3.5 で変更: *chunksize* 引数が追加されました。

shutdown(*wait=True*) (原文)

executor に対して、現在保留中のフューチャーが実行された後で、使用中のすべての資源を解放するように伝えます。シャットダウンにより後に Executor.submit() と Executor.map() を呼び出すと RuntimeError が送出されます。

wait が True の場合、すべての未完了のフューチャの実行が完了して Executor に関連付けられたリソースが解放されるまで、このメソッドは返りません。 *wait* が False の場合、このメソッドはすぐに返り、すべての未完了のフューチャの実行が完了したときに、 Executor に関連付けられたリソースが解放されます。 *wait* の値に関係なく、すべての未完了のフューチャの実行が完了するまで Python プログラム全体は終了しません。

with 文を使用することで、このメソッドを明示的に呼ばないようにできます。 with 文は Executor をシャットダウンします (*wait* を True にセットして Executor.shutdown() が呼ばれたかのように待ちます)。

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2. ThreadPoolExecutor (原文)

ThreadPoolExecutor はスレッドのプールを使用して非同期に呼び出しを行う、 Executor のサブクラスです。

Future に関連づけられた呼び出し可能オブジェクトが、別の Future の結果を待つ時にデッドロックすることがあります。例:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

以下でも同様です:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*) (原文)

最大で *max_workers* 個のスレッドを非同期実行に使う Executor のサブクラスです。

バージョン 3.5 で変更: *max_workers* が None か指定されない場合のデフォルト値はマシンのプロセッサの数に 5 を掛けたものになります。これは、 ThreadPoolExecutor は CPU の処理ではなく I/O をオーバーラップするのによく使用されるため、ProcessPoolExecutor のワーカーの数よりもこのワーカーの数を増やすべきであるという想定に基づいています。

バージョン 3.6 で追加: *thread_name_prefix* 引数が追加され、デバッグしやすくなるようにプールから作られたワーカスレッド threading.Thread の名前を管理できるようになりました。

17.4.2.1. ThreadPoolExecutor の例 (原文)

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.4.3. ProcessPoolExecutor (原文)

ProcessPoolExecutor はプロセスプールを使って非同期呼び出しを実施する Executor のサブクラスです。ProcessPoolExecutor は multiprocessing モジュールを利用します。このため Global Interpreter Lock を回避することができますが、pickle 化できるオブジェクトしか実行したり返したりすることができません。

`__main__` モジュールはワーカサブプロセスでインポート可能でなければなりません。 すなわち、 ProcessPoolExecutor は対話的インタープリタでは動きません。

ProcessPoolExecutor に渡された呼び出し可能オブジェクトから Executor や Future メソッドを呼ぶとデッドロックに陥ります。

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*) (原文)

Executor のサブクラスで、最大 *max_workers* のプールを使用して非同期な呼び出しを行います。 *max_workers* が `None` や与えられなかった場合、デフォルトでマシンのプロセッサの数になります。 *max_workers* が `0` 以下の場合 ValueError が送出されます。

バージョン 3.3 で変更: ワーカプロセスの1つが突然終了した場合、BrokenProcessPool エラーが送出されるようになりました。 以前は挙動は未定義でしたが、 executor や futures がフリーズしたりデッドロックを起こすことがしばしばでした。

17.4.3.1. ProcessPoolExecutor の例 (原文)

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.4.4. Future オブジェクト (原文)

Future クラスは呼び出し可能オブジェクトの非同期実行をカプセル化します。 Future のインスタンスは Executor.submit() によって生成されます。

class concurrent.futures.**Future** (原文)

呼び出し可能オブジェクトの非同期実行をカプセル化します。 Future インスタンスは Executor.submit() で生成され、テストを除いて直接生成すべきではありません。

cancel() (原文)

呼び出しのキャンセルを試みます。もし呼び出しが現在実行中でキャンセルすることができない場合、メソッドは `False` を返します。そうでない場合呼び出しはキャンセルされ、`True` を返します。

cancelled() (原文)

呼び出しが正常にキャンセルされた場合 `True` を返します。

running() (原文)

現在呼び出しが実行中でキャンセルできない場合 `True` を返します。

done() (原文)

呼び出しが正常にキャンセルされたか終了した場合 `True` を返します。

result(*timeout=None*) (原文)

呼び出しによって返された値を返します。呼び出しがまだ完了していない場合、このメソッドは *timeout* 秒の間待機します。呼び出しが *timeout* 秒間の間に完了しない場合、 concurrent.futures.TimeoutError が送出されます。 *timeout* にはintかfloatを指定できます。*timeout* が指定されていないか、 `None` である場合、待機時間に制限はありません。

future が完了する前にキャンセルされた場合 CancelledError が送出されます。

呼び出しが例外を送出した場合、このメソッドは同じ例外を送出します。

exception(*timeout=None*) (原文)

呼び出しによって送出された例外を返します。呼び出しがまだ完了していない場合、このメソッドは *timeout* 秒だけ待機します。呼び出しが *timeout* 秒の間に完了しない場合、 concurrent.futures.TimeoutError が送出されます。*timeout* にはintかfloatを指定できます。 *timeout* が指定されていないか、 None である場合、待機時間に制限はありません。

future が完了する前にキャンセルされた場合 CancelledError が送出されます。

呼び出しが例外を送出することなく完了した場合、None を返します。

add_done_callback(*fn*) (原文)

呼び出し可能な *fn* オブジェクトを future にアタッチします。futureがキャンセルされたか、実行を終了した際に、future をそのただ一つの引数として *fn* が呼び出されます。

追加された呼び出し可能オブジェクトは、追加された順番で呼びだされ、追加を行ったプロセスに属するスレッド中で呼び出されます。もし呼び出し可能オブジェクトが Exception のサブクラスを送出した場合、それはログに記録され無視されます。呼び出し可能オブジェクトが BaseException のサブクラスを送出した場合の動作は未定義です。

もしfutureがすでに完了しているか、キャンセル済みであれば、*fn* は即座に実行されます。

以下の Future メソッドは、ユニットテストでの使用と Executor を実装することを意図しています。

set_running_or_notify_cancel() (原文)

このメソッドは、Future に関連付けられたワークやユニットテストによるワークの実行前に、 Executor の実装によってのみ呼び出してください。

このメソッドが False を返す場合、 Future はキャンセルされています。つまり、 Future.cancel() が呼び出されて *True* が返っています。Future の完了を (as_completed() または wait() により) 待機するすべてのスレッドが起動します。

このメソッドが True を返す場合、 Future はキャンセルされて、実行状態に移行されています。つまり、Future.running() を呼び出すと *True* が返ります。

このメソッドは、一度だけ呼び出すことができ、Future.set_result() または Future.set_exception() がキャンセルされた後には呼び出すことができません。

set_result(*result*) (原文)

Future に関連付けられたワークの結果を *result* に設定します。

このメソッドは、 Executor の実装またはユニットテストによってのみ使用してください。

set_exception(*exception*) (原文)

Future に関連付けられたワークの結果を Exception *exception* に設定します。

このメソッドは、 Executor の実装またはユニットテストによってのみ使用してください。

17.4.5. モジュール関数 (原文)

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*) (原文)

fs によって与えられた (別の Executor インスタンスによって作成された可能性のある) 複数の Future インスタンスの完了を待機します。集合型を 2 要素含む名前付きのタプルを返します。1 つめの集合 done には、待機の完了前に完了したフューチャ (完了またはキャンセル済み) が含まれます。2 つめの集合 not_done には、未完了のフューチャが含まれます。

timeout で結果を返すまで待機する最大秒数を指定できます。*timeout* は整数か浮動小数点数をとります。*timeout* が指定されないか None の場合、無期限に待機します。

return_when でこの関数がいつ結果を返すか指定します。指定できる値は以下の 定数のどれか一つです:

定数	説明
FIRST_COMPLETED	いずれかのフューチャが終了したかキャンセルされたときに返します。
FIRST_EXCEPTION	いずれかのフューチャが例外の送出で終了した場合に返します。例外を送出したフューチャがない場合は、ALL_COMPLETED と等価になります。
ALL_COMPLETED	すべてのフューチャが終了したかキャンセルされたときに返します。

concurrent.futures.**as_completed**(*fs*, *timeout=None*) (原文)

フューチャの完了時 (完了またはキャンセル) に yield する *fs* によって与えられた (別の Executor インスタンスによって作成された可能性のある) 複数の Future インスタンスのイテレータを返します。 as_completed() が呼び出される前に完了したすべてのフューチ

ャは最初に yield されます。__next__() が呼び出され、その結果が as_completed() の元々の呼び出しから *timeout* 秒経った後も利用できない場合、返されるイタレータは concurrent.futures.TimeoutError を送出します。*timeout* は整数または浮動小数点数です。*timeout* が指定されないか None である場合、待ち時間に制限はありません。

参考:

PEP 3148 – futures – execute computations asynchronously
この機能を Python 標準ライブラリに含めることを述べた提案です。

17.4.6. 例外クラス (原文)

exception concurrent.futures.**CancelledError** (原文)

future がキャンセルされたときに送出されます。

exception concurrent.futures.**TimeoutError** (原文)

future の操作が与えられたタイムアウトを超過したときに送出されます。

exception concurrent.futures.process.**BrokenProcessPool** (原文)

RuntimeError から派生しています。 この例外クラスは ProcessPoolExecutor のワーカの1つが正常に終了されなかったとき (例えば外部から kill されたとき) に送出されます。

バージョン 3.3 で追加.