Pool Executors

This lesson discusses the various APIs for thread and pool executors.

Pool Executors

In the previous sections, we studied how to create and manage threads and processes. However, managing these entities can be taxing on the developer so Python alleviates this burden by providing an interface which abstracts away the subtleties of starting and tearing down threads or processes. The **`concurrent.futures`** package provides the **`Executor`** interface which can be used to submit tasks to either threads or processes. The two subclasses are:

- ThreadPoolExecutor
- ProcessPoolExecutor

Tasks can be submitted synchronously or asynchronously to the pools for execution.

ThreadPoolExecutor

The **`ThreadPoolExecutor`** uses threads for executing submitted tasks. Let's look at a very simple example.

Using ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor
from threading import current_thread


def say_hi(item):

    print("\nhi " + str(item) + " executed in thread id " + cu
rrent_thread().name, flush=True)


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)
    lst = list()
    for i in range(1, 10):
        lst.append(executor.submit(say_hi, "guest" + str(i)))

    for future in lst:
        future.result()

    executor.shutdown()
```

We create a thread pool with a maximum of ten threads. Next, we run in a loop and submit tasks to be executed. The first argument to the `submit()` is a callable which gets invoked with the arguments that follow. If you examine the output you'll see that tasks are executed by threads with different names. The submit calls return what we call a **future**. The `Future` class represents the execution of the callable. Note that the invocation `future.result()` is blocking. Interestingly, if we change the code within the first for loop as follows, the execution becomes serial.

```
    for i in range(1, 10):
        future = executor.submit(say_hi, "guest" + str(i))
        future.result()
```

Generally this is not the pattern we use for submitting tasks. The idea is to be able to process multiple tasks in parallel and query the progress of the tasks using the future object. For instance, say your program is trying to download multiple mp3 files from the internet, then the download tasks can be carried out in parallel independent of each other. Or you are writing a script that copies several files from a source directory to a destination directory then again the task is amenable for parallel processing.

At the end of the program we invoke **`shutdown()`** method on the executor. By default, the executor will wait for the futures to complete and any associated resources to be freed. Any task submission after invoking shutdown will result in an exception. If the call is invoked with **wait**=**False** then the call becomes nonblocking. However, the entire Python program will not shut down until all the futures are done.

```
from concurrent.futures import ThreadPoolExecutor
from threading import current_thread


def say_hi(item):

    print("\nhi " + str(item) + " executed in thread id " + current_thread().name,


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)
    lst = list()
    for i in range(1, 10):
        lst.append(executor.submit(say_hi, "guest" + str(i)))

    for future in lst:
        future.result()

    executor.shutdown()
```

map

The **map()** returns an iterator over the results of applying a function to a list of values. Both the function and the values are passed-in as parameters to the **map()** call. The results are returned in-order of the input values. Consider the following snippet:

map() example

```
executor = ThreadPoolExecutor(max_workers=10)

it = executor.map(square, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                  chunksize=1, timeout=2)
```

In our example **square()** is the callable. Internally, the iterator's **__next()__** calls the **result()** method of the future, which is a blocking call. If the result isn't ready for one of the futures, the iteration will block. To mitigate this situation, we can specify a timeout value in the map call and if the result isn't ready after timeout number of seconds have elapsed, a **Timeout** exception is raised. We are able to iterate over the futures which have completed but the first future encountered whose computation hasn't completed will result in blocking the iteration.

In the example below, we simulate a timeout exception by sleeping for ten seconds for the fifth computation in the callable whilst the timeout is set to two seconds in the **map()** call. If you run the example you'll see the results of the first four futures printed before a timeout exception is raised.

Also note that if you don't specify the timeout argument in the map call, the iteration would simply block the fifth time until the fifth future completes.

```
from concurrent.futures import ThreadPoolExecutor
from threading import current_thread
import time


def square(item):
    if item == 5:
        time.sleep(10)
    return item * item


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)

    it = executor.map(square, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                      chunksize=1, timeout=2)

    for sq in it:
        print(sq)

    executor.shutdown()
```

In the above two examples, even though the main thread exits when it receives the timeout exception, however, the python program doesn't exit until the thread executing the fifth computation also exists. If the thread were a daemon thread then the program could exit without waiting for the thread to complete. Lastly, note that the argument **chunksize** will have no effect when used with a thread pool.

ProcessPoolExecutor

The process pool is very similar to a thread pool except that it is pool of processes that execute the task rather than threads. Let's rewrite one of the previous examples using a process pool instead of a thread pool.

```
from concurrent.futures import ProcessPoolExecutor
import multiprocessing
from multiprocessing import current_process
from threading import current_thread

import os



def say_hi(item):
    print(
        "\nhi " + str(item) + " executed in thread id " + curr
ent_thread().name + " in process id " + str(
            os.getpid()) + " with name " + current_process().n
ame,
        flush=True)



if __name__ == '__main__':
    print("Main process id " + str(os.getpid()))
    multiprocessing.set_start_method('spawn')
    executor = ProcessPoolExecutor(max_workers=10)
    lst = list()
    for i in range(1, 10):
        lst.append(executor.submit(say_hi, "guest" + str(i)))

    for future in lst:
        future.result()

    executor.shutdown()
```

```
from concurrent.futures import ProcessPoolExecutor
import multiprocessing
from multiprocessing import current_process
from threading import current_thread

import os


def say_hi(item):
    print(
        "\nhi " + str(item) + " executed in thread id " + current_thread().name + "
            os.getpid()) + " with name " + current_process().name,
        flush=True)


if __name__ == '__main__':
    print("Main process id " + str(os.getpid()))
    multiprocessing.set_start_method('spawn')
    executor = ProcessPoolExecutor(max_workers=10)
    lst = list()
    for i in range(1, 10):
        lst.append(executor.submit(say_hi, "guest" + str(i)))

    for future in lst:
        future.result()

    executor.shutdown()
```

Note that we can either have the processes spawned or forked by the process pool by setting the start method appropriately. If you change the start method to fork on **line#18** in the above example, the output would show that the processes were forked.

We can also use the **`map()`** API with the process pool. We redo the threadpool example using a processpool below.

```
from concurrent.futures import ProcessPoolExecutor
import os
import time


def square(item):
    print("Executed in process with id " + str(os.getpid()), flush=True)
    return item * item


if __name__ == '__main__':
    executor = ProcessPoolExecutor(max_workers=10)

    it = executor.map(square, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10), chunksize=1)

    for sq in it:
        print(sq)

    executor.shutdown()
```

The only major difference when using **`map()`** with threads vs processes is the effect of the **chunksize** argument. In the above example, we have set the chunksize to one which implies each square will be calculated by a different process. If we change the chunksize to five then we only require two processes to square the ten input values. Depending on the usecase it may happen that a chunksize set to a higher value results in faster execution as time is saved in creating and then tearing down more number of processes.

Back
Quiz 3

Next
Futures

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/pool-executors__concurrent-package__python-concurrency-for-senior-engineering-interviews)