Pool

This lesson discusses the various APIs and their working for both thread and process pools.

Pool

The `Pool` object consists of a group of processes that can receive tasks for execution. The concept is very similar to a **thread pool**. Creating and tearing down threads is expensive so most programming language frameworks provide a notion of a pool of threads. Once a thread is done executing a task, it is returned back to the pool rather than being terminated. The process pool concept is similar where the processes are kept alive till there are tasks to be executed. Let's dive into an example to see how the `Pool` class works. Consider the code below:

Example of using Pool class

```
def init(main_id):
    print("pool process with id {0} received a task from mai
n process with id {1}".format(os.getpid(), main_id))


def square(x):
    return x * x


if __name__ == '__main__':
    main_process_id = os.getpid()

    pool = Pool(processes=1,
                initializer=init,
                initargs=(main_process_id,),
                maxtasksperchild=1)

    result = pool.apply(square, (3,))
    print(result)
```

The above program submits an integer to the pool to get the square of its value. If you run the below program you'll see the output showing PIDs of two different processes. One is the main process and the other is the pool process that actually does the computation.

```
from multiprocessing import Pool
import os


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".


def square(x):
    return x * x


if __name__ == '__main__':
    main_process_id = os.getpid()

    pool = Pool(processes=1,
                initializer=init,
                initargs=(main_process_id,),
                maxtasksperchild=1)

    result = pool.apply(square, (3,))
    print(result)
```

Let's understand the arguments we are passing to the `Process` class:

- **processes** is the number of worker processes to use. If processes is None then the number returned by `os.cpu_count()` is used, which is the number of CPUs on the system.
- **initializer** is an optional initialization method that is invoked before a process starts executing a task.
- **initargs** are the arguments passed to the initializer method.
- **maxtasksperchild** is the maximum number of tasks a pool process will execute before exiting. By default the pool process will live till the Pool exists.

Once a pool is defined, we invoke **`apply()`** method to submit a single task of squaring an integer. **`apply()`** is the simplest API that **`Pool`** provides. It executes a specified method in a single pool process. Furthermore it is a blocking call and the result is returned to the caller.

An asynchronous variant of the same method is availabe by the name of **`apply_async()`**. This non-blocking variant also takes in callbacks for success and failure. We rewrite the same program using the asynchronous version.

Using asynchronous apply

```
from multiprocessing import Pool
import os
import time


def init(main_id):
    print("pool process with id {0} received a task from mai
n process with id {1}".format(os.getpid(), main_id))


def square(x):
    return x * x


def on_success(result):
    print("result is " + str(result))


def on_error(err):
    print("error is " + str(err))


if __name__ == '__main__':
    main_process_id = os.getpid()

    pool = Pool(processes=1,
                initializer=init,
                initargs=(main_process_id,),
                maxtasksperchild=1)

    result = pool.apply_async(square, (9,), callback=on_succes
s, error_callback=on_error)

    # prevent main from exiting before the pool process comple
tes
    time.sleep(2)
```

```
from multiprocessing import Pool
import os
import time


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".


def square(x):
    return x * x


def on_success(result):
    print("result is " + str(result))


def on_error(err):
    print("error is " + str(err))


if __name__ == '__main__':
    main_process_id = os.getpid()

    pool = Pool(processes=1,
                initializer=init,
                initargs=(main_process_id,),
                maxtasksperchild=1)

    result = pool.apply_async(square, (9,), callback=on_success, error_callback=on_

    # prevent main from exiting before the pool process completes
    time.sleep(6)
```

If you pay attention at the output of the program, you'll notice that the initialization method is called twice since its print statement appears twice. The first time it gets printed is when we set-up the pool. If you comment out **line#30** you'll still see the initialization method being invoked because the pool creates worker processes. Next, one of the pool workers executes our task and then exits because we have set

maxtasksperchild to be exactly 1. This is why the two print statements show the worker processes having different pids. If you change the **maxtasksperchild** to 2 in the above code the second print statement would disappear.

Back

Next

Barrier, Semaphore, Condition Variable

... continued

Mark as Completed

Report an Issue

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