Python Multiprocessing Programming Tips

Su Jianlin

February 19, 2017

1 The Process

In Python, if you want to perform multi-process computing, it is generally implemented through `multiprocessing`. The most commonly used feature is the process pool within `multiprocessing`. For example:

```
from multiprocessing import Pool
import time

def f(x):
    time.sleep(1)
    print x+1
    return x+1

a = range(10)
pool = Pool(4)
b = pool.map(f, a)
pool.close()
pool.join()

print b
```

Writing it this way is concise, clear, and indeed convenient. Interestingly, you only need to change `multiprocessing` to `multiprocessing.dummy` to switch the program from multi-processing to multi-threading.

2 Objects

Python is an object-oriented programming language, and we often encapsulate programs into classes. However, within a class, the above method does not work well. For example:

```
from multiprocessing import Pool
import time

class test:
    def __init__(self):
        self.a = range(10)
    def run(self):
        def f(x):
            time.sleep(1)
            print x+1
            return x+1
        pool = Pool(4)
        self.b = pool.map(f, self.a)
        pool.close()
        pool.join()
```

```
16
17 t = test()
18 t.run()
19 print t.b
```

This code, which looks quite natural, throws an error when run:

```
cPickle.PicklingError: Can't pickle <type 'function'>: attribute lookup
__builtin__.function failed
```

However, if you replace `multiprocessing` with `multiprocessing.dummy`, no error occurs. Simply put, this is because variables cannot be shared between multiple processes, whereas multiple threads reside within the same process and naturally do not face this issue.

3 Imitation

To research multi-process programming within objects, I made several attempts. Later, I realized that many modules in `gensim` support parallelism, so I decided to imitate them. Sure enough, I found ldamulticore.py. After repeatedly comparing and studying it with online materials, I summarized a relatively concise, convenient, and universal way of writing it.

Like most multi-process programming, to communicate between processes, a `Queue` object needs to be established. The difference is that general online tutorials use the `Process` function from `multiprocessing` combined with loop statements to start multiple processes, while using `Pool` usually fails (unless you use `multiprocessing.Manager.Queue`, refer to this article). However, `gensim` uses a trick with `Pool` that allows starting multiple processes directly through `Pool`. The work of experts is indeed different. The reference code is as follows:

```
1  from multiprocessing import Pool, Queue
2  import time
3
4  class test:
5      def __init__(self):
6          self.a = range(10)
7      def run(self):
8          in_queue, out_queue = Queue(), Queue()
9          for i in self.a:
10             in_queue.put(i)
11         def f(in_queue, out_queue):
12             while not in_queue.empty():
13                 time.sleep(1)
14                 out_queue.put(in_queue.get()+1)
15         pool = Pool(4, f, (in_queue, out_queue))
16         self.b = []
17         while len(self.b) < len(self.a):
18             if not out_queue.empty():
19                 t = out_queue.get()
20                 print t
21                 self.b.append(t)
22         pool.terminate()
23
24 t = test()
25 t.run()
26 print t.b
```

In summary, the approach is to establish two `Queues`: one responsible for task queuing and the other for retrieving results. What is quite magical is that `Pool` actually has second and

third parameters! For specific details, please see the official documentation. These are the initialization functions for the `Pool`, which also run in parallel automatically.

Note that after running the line `pool = Pool(4, f, (in_queue, out_queue))`, the multi-process starts, but it does not wait for the processes to finish; instead, it immediately executes the subsequent statements. At this point, you could use `pool.close()` and `pool.join()` as before to let the processes complete before continuing. However, the solution used here is to directly execute the result-retrieval statements and determine whether the processes have finished through that process. Once finished, the process pool is closed via `pool.terminate()`. This style of writing is basically universal.