Python 多进程模式

马力　2020.03.09

主要内容

- multiprocessing
  - Process
  - Queue
  - Pipe
  - Lock
  - Pool
  - Manager

Python 中的进程与线程

- 课程之前编写的 Python 程序，都是执行单任务的进程，也就是只有一个线程
- 面对多任务，有三种实现方式：
  - 多进程模式；
    - multiprocessing
  - 多线程模式；
  - 多进程 + 多线程模式

多进程模式

- multiprocessing
  - Process
  - Queue
  - Pipe
  - Lock
  - Pool
  - Manager

多进程模式

- Process 参数

```python
class Process():
    name: str
    daemon: bool
    pid: Optional[int]
    exitcode: Optional[int]
    authkey: bytes
    sentinel: int
    # TODO: set type of group to None
    def __init__(self,
                 group: Any = ...,
                 target: Optional[Callable] = ...,
                 name: Optional[str] = ...,
                 args: Iterable[Any] = ...,
                 kwargs: Mapping[Any, Any] = ...,
                 *,
                 daemon: Optional[bool] = ...) -> None: ...
```

group	值始终为 None，为了今后在 multi-threading 功能扩展而预留的参数。目的是为了给线程加组标，加优先级用。
target	调用对象 ，即子进程要执行的任务
args	调用对象的位置参数
kwargs	调用对象的字典 ，kwargs = {"name" : "genji", "age" : 18}

多进程模式

- Process 属性

p.daemon	默认是 False， 如果设为 true， 代表 p 为后台运行的守护进程 ， 当 p 的父进程终止时 ，p 也随之终止，并且设定为 True 后 ，p 不能创建自己的新锦成 ， 必须在 p.start () 之前进行设置
p.name	进程的名字
p.pid	进程的 pid(进程号)
p.exitcode	进程在运行时为 None， 如果为 -N， 表示被信号 N 结束
p.authkey	进程的身份验证键 ， 默认是由 os.urandom() 随机生成的 32 字符的字符串 ． 这个键的用途是为涉及网络连接的底层进程间通信提供安全性 ， 这类连接只有在具有相同的身份验证键时才能成功

多进程模式

- Process 方法

p.start ()	启动进程，并调用子进程中的 p.run ()
p.run ()	进程启动时运行的方法 ， 正是他去调用 target 指定的函数
p.terminate ()	强制终止进程 p ， 不会进行任何清理操作，如果 p 创建了子进程，该子进程就成了僵尸进程 ， 使用该方法需要特别小心这种情况，如果 p 还保存了一个锁那么也将不会被释放 ， 进而导致死锁
p.is_alive ()	如果 p 仍然运行 ， 返回 True
p.join ([timeout])	主线程等待 p 终止（强调 ： 是主线程处于等待的状态 ， 而 p 是处于运行状态），timeout 是可选的超时时间 ， 需要强调的是 ，p.join 只能 join start 开启的进程 ， 而不能 join run 开启的进程

强调的是：运行完毕并非终止运行。

- 对主进程来说，运行完毕指的是主进程代码运行完毕
- 对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕

Test1.py

多进程模式

- Process

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print('Run child process %s (%s)...' % (name, os.getpid()))


print('Parent process %s.' % os.getpid())
p = Process(target=run_proc, args=('test',))
print('Child process will start.')
p.start()
p.join()
print('Child process end.')
```

```
Parent process 24148.
Child process will start.
Child process end.
```

- 不可在 jupyter notebook 中使用
- 在 windows 操作系统中由于没有 fork(linux 操作系统中创建进程的机制)，在创建进程的时候自动 import 启动它的这个文件，而在 import 的时候又执行了整个文件，因此如果将 process() 直接写在文件中就会无限递归创建子进程报错。必须把创建子进程的部分写在 if __name__ == '__main__':

```
PS C:\Data\pyrhonC4tmp> python .\test5.py
Parent process 10612.
Child process will start.
Run child process test (4976)...
Child process end.
```

Test2.py
Test3.py
Test4.py

多进程模式

- Queue
  - Queue 是一个近似 queue.Queue 的克隆

q.put (item)	将 item 放入队列中，如果当前队列已满，就会阻塞，直到有数据从管道中取出
q.put_nowait (item)	将 item 放入队列中，如果当前队列已满，不会阻塞，但是会报错
q.get ()	返回放入队列中的一项数据，取出的数据将是先放进去的数据，若当前队列为空，就会阻塞，直到放入数据进来
q.get_nowait ()	返回放入队列中的一项数据，同样是取先放进队列中的数据，若当前队列为空，不会阻塞，但是会报错

Test5.py

多进程模式

- Pipe
  - Pipe()返回一个由管道连接的连接对象，默认情况下是双工（双向）

```python
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()
```

```
PS C:\Data\pyrhonC4tmp> python .\test6.py
[42, None, 'hello']
```

Test6.py

多进程模式

- Lock

```python
from multiprocessing import Lock, Process
import time
import json
def search(n):
    dic = json.load(open("db.txt"))
    print("第%s个用户查到票剩余%s" % (n + 1, dic["count"]))

def get(n):
    dic = json.load(open('db.txt'))
    time.sleep(0.1)  # 模拟读数据延迟
    if dic["count"] > 0:
        dic["count"] -= 1
        time.sleep(0.2)  # 模拟写数据延迟
        json.dump(dic, open("db.txt", "w"))
        print("第%s个用户抢到票了" % (n + 1))

def work(n):
    search(n)
    get(n)

if __name__ == '__main__':
    for i in range(20):
        p = Process(target=work, args=(i,))
        p.start()
```

Test_tickets.py

多进程模式

- 进程池
  - 在实际处理问题的过程中，有时会有成千上万的任务需要被执行，我们不可能创建那么多进程去完成任务．首先创建进程需要时间，销毁进程同样需要时间．即便是真的创建好了这么多进程，操作系统也不允许他们同时执行的，这样反而影响了程序的效率．
  - 进程池 -- 定义一个池子，在里面放上固定数量的进程，有任务要处理的时候就会拿一个池中的进程来处理任务，等到处理完毕，进程并不关闭而是放回进程池中继续等待任务．如果需要有很多任务需要执行，池中的进程数不够，任务会就要等待进程执行完任务回到进程池，拿到空闲的进程才能继续执行．池中的进程数量是固定的，那么同一时间最多有固定数量的进程在运行．这样不会增加操作系统的调度难度，还节省了开闭进程的时间，也一定程度上能够实现并发效果。
- multiprocessing.Pool 模块

numprocess	要创建的进程数，如果省略，将默认使用 os.cpu_count () 的值
initializer	是每个工作进程启动时要执行的可调用对象，默认为 None
initargs	传给 initializer 的参数组

多进程模式

- multiprocessing.Pool

p.apply (func [,args [,kwargs]])	在一个池工作进程中执行 func(*args,**kwargs)，然后返回结果（同步调用）
p.apply_async(func [,args [,kwargs]])	在一个池工作进程中执行 func(*args,**kwargs)，然后返回结果（异步调用）
p.close()	关闭进程池，防止进一步操作．如果所有操作持续挂起，他们将在工作进程终止前完成
p.join()	等待所有工作进程退出．此方法只能在 close () 或 terminate () 之后调用

多进程模式

- multiprocessing.Pool

```
(base) C:\Users>python tester.py
Parent process 8440.
Waiting for all subprocesses done...
Run task 0 (14028)...
Run task 1 (1640)...
Run task 2 (6380)...
Run task 3 (14212)...
Task 0 runs 0.35 seconds.
Run task 4 (14028)...
Task 3 runs 1.41 seconds.
Task 2 runs 1.71 seconds.
Task 4 runs 1.53 seconds.
Task 1 runs 2.39 seconds.
All subprocesses done.
```

In [*]:

```python
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

```
Parent process 24148.
Waiting for all subprocesses done...
```

- apply_async 是异步的，就是说子进程执行的同时，主进程继续向下执行。所以“ Waiting for all subprocesses done...” 先打印出来， close 方法意味着不能再添加新的 Process 了。对 Pool 对象调用 join （）方法，会暂停主进程，等待所有的子进程执行完，所以“ All subprocesses done.” 最后打印。另， task

多进程模式

- Manager
  - Manager() 返回的管理器对象控制一个服务器进程，该进程保存 Python 对象并允许其他进程使用代理操作它们。

```python
from multiprocessing import Process, Manager


def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()


if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

Test8.py

参考资料

- https://docs.python.org/3/library/multiprocessing.html

Thanks