parallel - homework & notes

徐红柚 19377031

2022-09-28

目录

1 Python 线程学习

Python3 线程中常用的两个模块为：- _thread （thread 已被废弃，为了兼容性 thread 重命名为 _thread）- threading（推荐）

Python 中使用线程有两种方式：函数或者用类来包装线程对象。

1.1 _thread

函数式：调用 _thread 模块中的 start_new_thread() 函数来产生新线程。语法如下:

```
_thread.start_new_thread ( function, args[, kwargs] )
```

- function 线程函数
- args 传递给线程函数的参数, 他必须是个 tuple 类型
- kwargs 可选参数

1.2 EXAMPLE 1 _thread

```
import _thread
import time

# 为线程定义一个函数
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print ("%s: %s" % ( threadName, time.ctime(time.time()) ))

# 创建两个线程
try:
   _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print ("Error: 无法启动线程")

while 1:
   pass
```

1.3 threading

_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- `threading.currentThread()`: 返回当前的线程变量。
- `threading.enumerate()`: 返回一个包含正在运行的线程的 list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- `threading.activeCount()`: 返回正在运行的线程数量，与 len(threading.enumerate()) 有相同的结果。

1.3.1 Thread 类

- `run()`: 用以表示线程活动的方法。
- `start()`: 启动线程活动。
- `join([time])`: 等待至线程中止。这阻塞调用线程直至线程的 join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- `isAlive()`: 返回线程是否活动的。
- `getName()`: 返回线程名。
- `setName()`: 设置线程名。

1.4 EXAMPLE 2 threading

我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法：

```
import threading
import time

exitFlag = 0
```

```
class myThread (threading.Thread):
    def __init__(self, threadID, name, delay):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.delay = delay
    def run(self):
        print (" 开始线程: " + self.name)
        print_time(self.name, self.delay, 5)
        print (" 退出线程: " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print (" 退出主线程")
```

1.5 线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是 0，线程"set" 从后向前把所有元素改成 1，而线程"print" 负责从前往后读取列表并打印。

那么，可能线程"set" 开始改的时候，线程"print" 便来打印列表了，输出就成了一半 0 一半 1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set" 要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print" 获得锁定了，那么就让线程"set" 暂停，也就是同步阻塞；等到线程"print" 访问完毕，释放锁以后，再让线程"set" 继续。

经过这样的处理，打印列表时要么全部输出 0，要么全部输出 1，不会再出现一半 0 一半 1 的尴尬场面。

1.6 EXAMPLE 3 线程同步

```
import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, delay):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
```

```
        self.delay = delay
    def run(self):
        print (" 开启线程： " + self.name)
        # 获取锁，用于线程同步
        threadLock.acquire()
        print_time(self.name, self.delay, 3)
        # 释放锁，开启下一个线程
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
```

```
print (" 退出主线程")
```

1.7 线程优先级队列（Queue）

Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括 FIFO（先入先出) 队列 Queue，LIFO（后入先出）队列 LifoQueue，和优先级队列 PriorityQueue。

这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

Queue 模块中的常用方法:

- `Queue.qsize()` 返回队列的大小
- `Queue.empty()` 如果队列为空，返回 True, 反之 False
- `Queue.full()` 如果队列满了，返回 True, 反之 False
- `Queue.full` 与 maxsize 大小对应
- `Queue.get([block[, timeout]])` 获取队列，timeout 等待时间
- `Queue.get_nowait()` 相当 Queue.get(False)
- `Queue.put(item)` 写入队列，timeout 等待时间
- `Queue.put_nowait(item)` 相当 Queue.put(item, False)
- `Queue.task_done()` 在完成一项工作之后，`Queue.task_done()` 函数向任务已经完成的队列发送一个信号 `Queue.join()` 实际上意味着等到队列为空，再执行别的操作

1.8 EXAMPLE 4 Queue

```
import queue
import threading
import time

exitFlag = 0
```

```
class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print (" 开启线程: " + self.name)
        process_data(self.name, self.q)
        print (" 退出线程: " + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
```

```
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print (" 退出主线程")
```

2 LAB - Homework

- Think about a slow piece of code you have ever written for either your past assignments or projects.
- Parallelize it in Python.

```
from threading import Thread,currentThread
import time

def task(name):
    time.sleep(2)
```

```
    print('%s print name: %s' %(currentThread().name,name))

class Task(Thread):
    def __init__(self,name):
        super().__init__()
        self._name=name

    def run(self):
        time.sleep(2)
        print('%s print name: %s' % (currentThread().name,self._name))

if __name__ == '__main__':
    n=100
    var='test'
    t=Thread(target=task,args=('thread_task_func',))
    t.start()
    t.join()

    t=Task('thread_task_class')
    t.start()
    t.join()

    print('main')
```

Result:

```
Thread-19 print name: thread_task_func
thread_task_class print name: thread_task_class
main
```