Python 单线程、多线程、多进程、协程性能比较

背景

上次参加公司的一个评审，有一个脚本任务是python写的，然后运行时间过长，期间也是使用了多线程来跑数据的，其实在我们看来多线程也确实是来提升效率的。同事抛出来一个问题，说python其实是伪多线程，由于没人去实验过也没有详细的解说这方面，所以疑问也一直没有得到解答，假期正好我也用Python在测试Presto的运行情况，也使用到了多线程、多进程等测试样例，我们就一起来看一下吧这几种并发的性能和适用场景吧！

线程与进程、协程

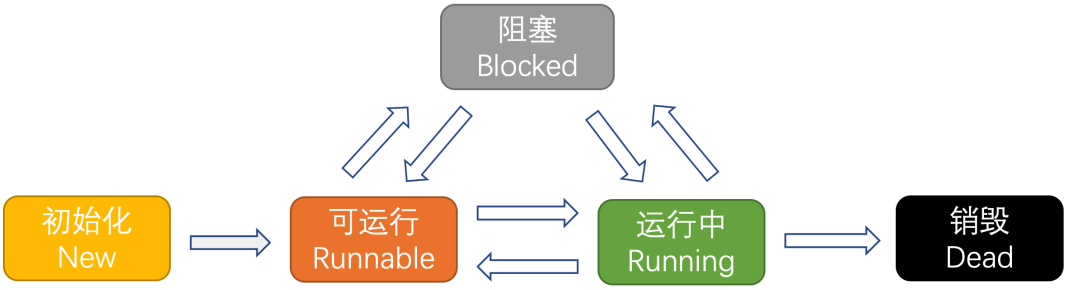
进程是什么

进程就是应用程序的启动实例。比如我们运行一个游戏，打开一个软件，就是开启了一个进程，进程拥有代码和打开的文件资源、数据资源、独立的内存空间；进程是资源分配的最小单位。

线程是什么

线程从属于进程，是程序的实际执行者。一个进程至少包含一个主线程，也可以有更多的子线程，线程拥有自己的栈空间；线程是 CPU 调度的最小单位。

线程具有五种状态：

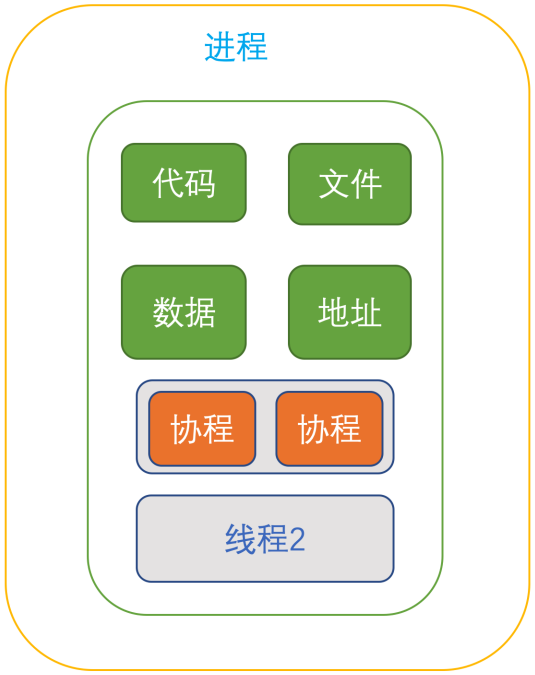


协程是什么

> A coroutine is a function that can suspend its execution (yield) until the given given YieldInstruction finishes.

官方解释就是协程是一个可以在给定的yield指令完成之前暂停执行的函数。通俗一点解释就是实际上协程是运行在线程上的一个代码块，它基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写程序来管理的轻量级线程叫做『用户空间线程』，具有对内核来说不可见的特性。

因为是自主开辟的异步任务，所以很多人也更喜欢叫它们纤程（Fiber），或者绿色线程（GreenThread）。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程；当出现IO阻塞的时候，由协程的调度器进行调度，通过将数据流立刻yield掉（主动让出），并且记录当前栈上的数据，阻塞完后立刻再通过线程恢复栈，并把阻塞的结果放到这个线程上去跑，这样看上去好像跟写同步代码没有任何差别，这整个流程可以称为coroutine，而跑在由coroutine负责调度的线程称为Fiber。比如Golang里的 go关键字其实就是负责开启一个Fiber，让func逻辑跑在上面。



Python的支持

Python 提供了 _thread（Python3 之前名为 thread ） 和 threading 两个线程模块。_thread 是低级、原始的模块，threading 是高级模块，对 _thread 进行了封装，增强了其功能与易用性，绝大多数时候，我们只需使用 threading 模块即可。

Python 提供了 multiprocessing 模块对多进程进行支持，它使用了与 threading 模块相似的 API 产生进程，除此之外，还增加了新的 API，用于支持跨多个输入值并行化函数的执行及跨进程分配输入数据，详细用法可以参考官方文档 https://docs.python.org/zh-cn/3/library/multiprocessing.html

Python3.5之前提供了yield、gevent类库来支持协程。Python3.5及以后提供了async/await来更好的实现协程

GLI

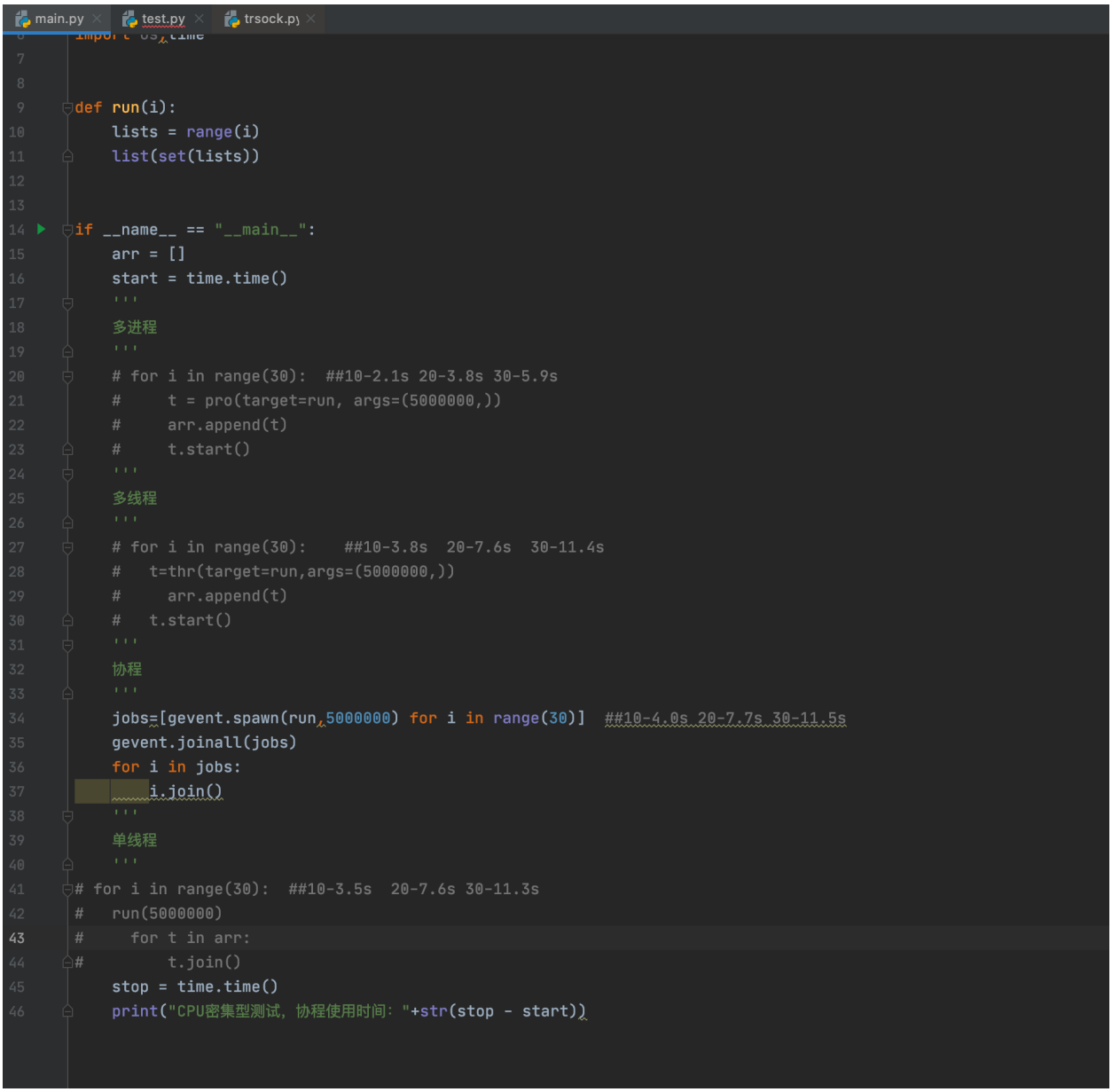
其实说到Python多线程，我们就离不开GLI,什么是GLI呢？GLI 全称 Global Interpreter Lock（全局解释器锁），是Python解释器CPython采用的一种控制机制，如果大家学过java那就知道我们使用并发的时候有关键词Synchronized类似，也是一种互斥锁。

CPython使用GLI来实现并发控制，即同一时刻只能有一条线程在执行python字节码。Python官网下载获得的就是CPython解释器，也是目前最流程的一种解释器，那么我们还有其他解释器PyPy、Psyco、Jython （也称 JPython）、IronPython 等解释器，其中 Jython 与 IronPython 分别采用 Java 与 C# 语言实现，就没有采用 GIL 机制；而 GIL 也不是 Python 特性，Python 可以完全独立于 GIL 运行

从上面的解释我们大概也可以看得出来，如果我们使用了CPython的解释器，那么多线程的使用必定会受到GLI的影响，但是它真的影响巨大吗？其实也分情况的！

简单的性能测试

CPU密集型



测试的结果

- 并发10次：【多进程】2.1s 【多线程】3.8s 【协程】4.0s 【单线程】3.5s
- 并发20次：【多进程】3.8s 【多线程】7.6s 【协程】7.7s 【单线程】7.6s
- 并发30次：【多进程】4.1s 【多线程】13.4s 【协程】13.4s 【单线程】11.3s

```
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，协程使用时间： 13.6315469742
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，协程使用时间： 13.6613349915
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，协程使用时间： 13.5704200268
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，协程使用时间： 13.4697310925
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#rz -y
Sent - main.py
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多线程使用时间： 13.5034310818
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多线程使用时间： 13.7035608292
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多线程使用时间： 13.5693078041
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多线程使用时间： 13.4217779636
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#rz -y
Sent - main.py
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多进程使用时间： 4.159979105
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多进程使用时间： 4.11174583435
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多进程使用时间： 4.24153900146
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
CPU密集型测试，多进程使用时间： 4.10614395142
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#
```

可以看到，在CPU密集型的测试下，多进程效果明显比其他的好，多线程、协程与单线程效果差不多。这是因为只有多进程完全使用了CPU的计算能力。在代码运行时，我们也能够看到，只有多进程可以将CPU使用率占满

IO密集型

```python
def urllib2_(url):
    f = open('tmp.txt', 'w')
    urllib2.urlopen(url, timeout=10).read()
def gevent_(urls):
    jobs=[gevent.spawn(urllib2_,url) for url in urls]
    gevent.joinall(jobs,timeout=10)
    for i in jobs:
        i.join()
def thread_(urls):
    a=[]
    for url in urls:
        t=threading.Thread(target=urllib2_,args=(url,))
        a.append(t)
    for i in a:
        i.start()
    for i in a:
        i.join()
def process_(urls):
    a=[]
    for url in urls:
        t=Process(target=urllib2_,args=(url,))
        a.append(t)
    for i in a:
        i.start()
    for i in a:
        i.join()
if __name__=="__main__":
    urls=["https://www.bing.com/"]*1000
    t1=time.time()
    gevent_(urls)
    t2=time.time()
    print 'gevent-time:%s' % str(t2-t1)
    thread_(urls)
    t4=time.time()
    print 'thread-time:%s' % str(t4-t2)
    arr = []
    start = time.time()
    for url in urls:
        p = Process(target=urllib2_,args=(url,))
        arr.append(p)
        p.start()
    for p in arr:
        p.join()
    stop = time.time()
    print 'process-time:%s' % str(stop-start)
```

我查阅了了很多资料，做了很多实验，但是得出的结果却和他们的有点不一样

1000次：

```
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:28.2582111359
thread-time:27.3806519508
thread-time:28.5730090141
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#rz -y
Sent - main.py
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:27.4546639919
thread-time:27.7567119598
process-time:28.3343498707
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#
```

500次：

```
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:14.215077877
thread-time:13.707280159
process-time:9.16258001328
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:13.5862908363
thread-time:14.4480202198
process-time:8.41807603836
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:13.7776482105
thread-time:13.8097388744
process-time:8.33735394478
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#
```

100次：

```
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:2.95804095268
thread-time:3.02543497086
process-time:1.92124795914
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:3.10279989243
thread-time:2.88265299797
process-time:1.56478881836
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#python main.py
gevent-time:3.82060217857
thread-time:3.14862990379
process-time:1.90560102463
[root@docker-200527203654414340 /data/o2/o2script/py/tmp]#
```

最终得出的结果是IO密集型的情况在进程数不多的情况下效率确实是进程比较高一点，但是进程数一旦到达峰值，效率就不如多线程和协程了！至于多线程和协程，其实效率都差不多！