慕沁

Python系列之 - 锁（GIL,Lock,Rlock,Event,信号量）

python 的解释器，有很多种，但市场占有率99.9%的都是基于c语言编写的CPython。在这个解释器里规定了GIL。

In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython’s memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)

意思：无论有多少个cpu，python在执行时会淡定的在同一时刻只允许一个线程运行。（一个进程可以开多个线程，但线程只能同时运行一个）

下面举例子：

```
def add():
    sum=0
    for i in range(10000000):
        sum+=i
    print("sum",sum)

def mul():
    sum2=1
    for i in range(1,100000):
        sum2*=i
    print("sum2",sum2)
import threading,time
start=time.time()

t1=threading.Thread(target=add)
t2=threading.Thread(target=mul)

l=[]
l.append(t1)
l.append(t2)
print(time.ctime())
add()
mul()

print("cost time %s"%(time.ctime()))
```

```
执行结果
Sat Apr 14 20:10:39 2018
sum 49999995000000
sum2 282422940796034787429342157802453551847
cost time Sat Apr 14 20:10:50 2018

sum2太大，截取了一部分
可已看出串行需要11秒，（-_-我还开着一些别的软件，需要占用大量内存）
```

```
然后我们用多线程分别执行两个程序
def add():
    sum=0
    for i in range(10000000):
        sum+=i
    print("sum",sum)

def mul():
```

接口测试(1)
芒果(3)
爬虫(14)
前端(11)
前后端分离(6)
权限菜单显示
权限管理(2)
软件(13)
设计模式(18)
首页(2)
数据分析(13)
数据库(16)
算法(4)
网路编程(14)
网络通信(2)
微服务(1)
微机原理(1)
小工具(10)
性能分析(6)
智能玩具(1)
自动化部署(12)
自动化运维(8)

```
    sum2=1
    for i in range(1,100000):
        sum2*=i
    print("sum2",sum2)
import threading,time
start=time.time()

t1=threading.Thread(target=add)
t2=threading.Thread(target=mul)

l=[]
l.append(t1)
l.append(t2)
print(time.ctime())
for t in l:
    t.start()
for t in l:
    t.join()
print("cost time %s"%(time.ctime()))
```

```
执行结果:
Sat Apr 14 20:14:51 2018
sum 4999999500000
sum2 28242294079603478742934215780245355
cost time Sat Apr 14 20:15:01 2018

可以看出执行速度加快了。
但是减少的时间不是很多。这是为什么呢?
```

首先我们需要知道任务类型分两种：CPU密集型、IO密集型

像上面的例子就是CPU密集型，需要大量的计算

而另一种就是需要频繁的进行输入输出（一遇到IO,就切换）

接着写一个IO密集型的例子：

```
import threading
import time
def music():
    print("begin to listen %s"%time.ctime())
    time.sleep(3)
    print("stop to listen %s" % time.ctime())

def game():

    print("begin to play game %s"%time.ctime())
    time.sleep(5)
    print("stop to play game %s" % time.ctime())

if __name__ == '__main__':
    t1=  threading.Thread(target=music)
    t2 = threading.Thread(target=game)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print("ending")
```

```
输出结果:
begin to listen Sat Apr 14 20:23:03 2018
begin to play game Sat Apr 14 20:23:03 2018
stop to listen Sat Apr 14 20:23:06 2018
stop to play game Sat Apr 14 20:23:08 2018
ending

Process finished with exit code 0
```

```
import threading
import time
def music():
    print("begin to listen %s"%time.ctime())
    time.sleep(3)
    print("stop to listen %s" % time.ctime())

def game():

    print("begin to play game %s"%time.ctime())
    time.sleep(5)
    print("stop to play game %s" % time.ctime())

if __name__ == '__main__':
    music()
    game()
    # t1=  threading.Thread(target=music)
    # t2 = threading.Thread(target=game)
    # t1.start()
    # t2.start()
    # t1.join()
    # t2.join()
    print("ending")
```

```
输出结果:
begin to listen Sat Apr 14 20:24:44 2018
stop to listen Sat Apr 14 20:24:47 2018
begin to play game Sat Apr 14 20:24:47 2018
stop to play game Sat Apr 14 20:24:52 2018
ending

Process finished with exit code 0
```

很明显 对于IO密集型多线程的优势非常明显.

同步锁Lock

多个线程都在同时操作同一个共享资源，所以造成了资源破坏，怎么办呢？(join会造成串行，失去所线程的意义)

```
import time
import threading
def addNum():
    global num #在每个线程中都获取这个全局变量
    temp=num
    time.sleep(0.0001)
    num =temp-1 #对此公共变量进行-1操作
num = 100  #设定一个共享变量
thread_list = []
for i in range(100):
    t = threading.Thread(target=addNum)
    t.start()
    thread_list.append(t)
for t in thread_list: #等待所有线程执行完毕
    t.join()
print('final num:', num )
结果:
final num: 87

Process finished with exit code 0
```

线程之间竞争资源，谁抢到谁执行

```
我们可以通过同步锁来解决这种问题

R=threading.Lock()

####
def sub():
    global num
    R.acquire()
    temp=num-1
    time.sleep(0.1)
    num=temp
    R.release()
# 即运行到此就变成了串行，本语言无力改变
```

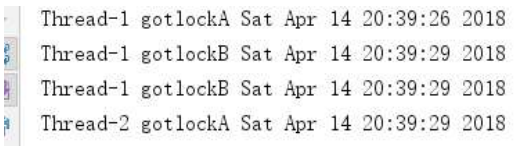
线程死锁和递归锁RLock

```
import threading,time
class myThread(threading.Thread):
    def doA(self):
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
        time.sleep(3)
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        lockB.release()
        lockA.release()

    def doB(self):
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        time.sleep(2)
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
        lockA.release()
        lockB.release()

    def run(self):
        self.doA()
        self.doB()
if __name__=="__main__":

    lockA=threading.Lock()
    lockB=threading.Lock()
    threads=[]
    for i in range(5):
        threads.append(myThread())
    for t in threads:
        t.start()
```



```
Thread-1 gotlockA Sat Apr 14 20:39:26 2018
Thread-1 gotlockB Sat Apr 14 20:39:29 2018
Thread-1 gotlockB Sat Apr 14 20:39:29 2018
Thread-2 gotlockA Sat Apr 14 20:39:29 2018
```

结果就卡到这里了,?,Thread-1申请lockB|Thread-2申请lockB，但是两者都申请不到于是产生了死锁

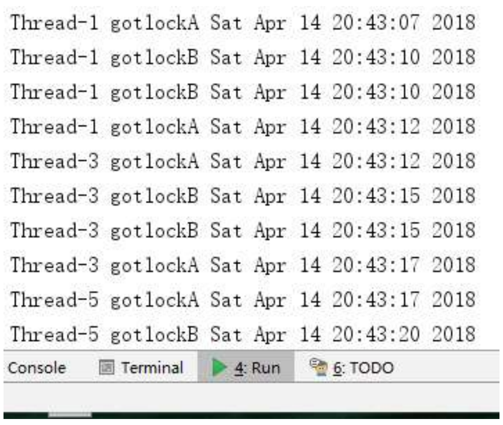
于是------当某个线程申请到一个锁，其余线程不能再申请。于是有了递归锁（其实就是内部维护了一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次acquire。直到一个线程所有的acquire都被release，其他的线程才能获得资源。）

```
import threading,time
class myThread(threading.Thread):
    def doA(self):
        R_lock.acquire()
        print(self.name,"gotlockA",time.ctime())
        time.sleep(3)
        R_lock.acquire()
        print(self.name,"gotlockB",time.ctime())
        R_lock.release()
        R_lock.release()

    def doB(self):
        R_lock.acquire()
        print(self.name,"gotlockB",time.ctime())
        time.sleep(2)
        R_lock.acquire()
        print(self.name,"gotlockA",time.ctime())
        R_lock.release()
        R_lock.release()

    def run(self):
        self.doA()
        self.doB()
if __name__=="__main__":
    R_lock = threading.RLock()
    threads=[]
    for i in range(5):
        threads.append(myThread())
    for t in threads:
        t.start()
```

结果:



```
Thread-1 gotlockA Sat Apr 14 20:43:07 2018
Thread-1 gotlockB Sat Apr 14 20:43:10 2018
Thread-1 gotlockB Sat Apr 14 20:43:10 2018
Thread-1 gotlockA Sat Apr 14 20:43:12 2018
Thread-3 gotlockA Sat Apr 14 20:43:12 2018
Thread-3 gotlockB Sat Apr 14 20:43:15 2018
Thread-3 gotlockB Sat Apr 14 20:43:15 2018
Thread-3 gotlockA Sat Apr 14 20:43:17 2018
Thread-5 gotlockA Sat Apr 14 20:43:17 2018
Thread-5 gotlockB Sat Apr 14 20:43:20 2018
```

同步条件(Event)

```
An event is a simple synchronization object;the event represents an internal flag,
and threads can wait for the flag to be set, or set or clear the flag themselves.
```

事件是一个简单的同步对象;事件表示一个内部标志,
线程可以等待设置标志，或设置或清除标志本身。

```
event = threading.Event()
```

#客户机线程可以等待设置标志。

```
# a client thread can wait for the flag to be set
event.wait()
```

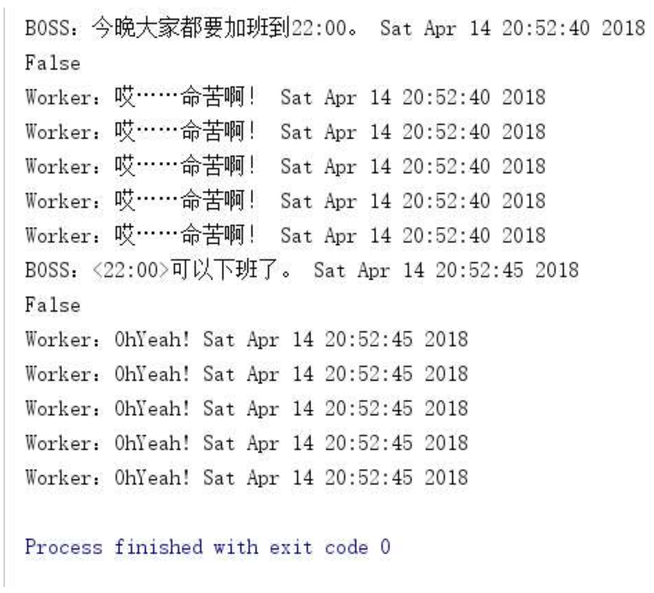
一个服务器线程可以设置或重置它。

```
# a server thread can set or reset it
event.set()
event.clear()
```

如果设置了标志，等待方法不会执行任何操作。
如果标记被清除，等待将阻塞直到它再次被设置。
任何数量的线程都可以等待相同的事件。

```
If the flag is set, the wait method doesn't do anything.
If the flag is cleared, wait will block until it becomes set again.
Any number of threads may wait for the same event.


import threading,time
class Boss(threading.Thread):
    def run(self):
        print("BOSS: 今晚大家都要加班到22:00。")
        print(event.isSet())
        event.set()
        time.sleep(5)
        print("BOSS: <22:00>可以下班了。")
        print(event.isSet())
        event.set()
class Worker(threading.Thread):
    def run(self):
        event.wait()
        print("Worker: 哎……命苦啊! ")
        time.sleep(1)
        event.clear()
        event.wait()
        print("Worker: OhYeah!")
if __name__=="__main__":
    event=threading.Event()
    threads=[]
    for i in range(5):
        threads.append(Worker())
    threads.append(Boss())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```



```
BOSS：今晚大家都要加班到22:00。 Sat Apr 14 20:52:40 2018
False
Worker：哎……命苦啊！ Sat Apr 14 20:52:40 2018
Worker：哎……命苦啊！ Sat Apr 14 20:52:40 2018
Worker：哎……命苦啊！ Sat Apr 14 20:52:40 2018
Worker：哎……命苦啊！ Sat Apr 14 20:52:40 2018
Worker：哎……命苦啊！ Sat Apr 14 20:52:40 2018
BOSS：<22:00>可以下班了。 Sat Apr 14 20:52:45 2018
False
Worker：OhYeah! Sat Apr 14 20:52:45 2018
Worker：OhYeah! Sat Apr 14 20:52:45 2018
Worker：OhYeah! Sat Apr 14 20:52:45 2018
Worker：OhYeah! Sat Apr 14 20:52:45 2018
Worker：OhYeah! Sat Apr 14 20:52:45 2018

Process finished with exit code 0
```

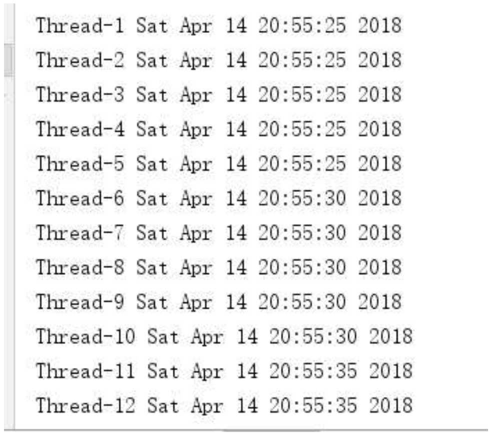
信号量

信号量用来控制线程并发数的，BoundedSemaphore或Semaphore管理一个内置的计数 器，每当调用acquire()时-1，调用release()时+1。

计数器不能小于0，当计数器为 0时，acquire()将阻塞线程至同步锁定状态，直到其他线程调用release()。(类似于停车位的概念)

BoundedSemaphore与Semaphore的唯一区别在于前者将在调用release()时检查计数 器的值是否超过了计数器的初始值，如果超过了将抛出一个异常。

```
import threading,time
class myThread(threading.Thread):
    def run(self):
        if semaphore.acquire():
            print(self.name)
            time.sleep(5)
            semaphore.release()
if __name__=="__main__":
    semaphore=threading.Semaphore(5)
    thrs=[]
    for i in range(100):
        thrs.append(myThread())
    for t in thrs:
        t.start()
```



```
Thread-1 Sat Apr 14 20:55:25 2018
Thread-2 Sat Apr 14 20:55:25 2018
Thread-3 Sat Apr 14 20:55:25 2018
Thread-4 Sat Apr 14 20:55:25 2018
Thread-5 Sat Apr 14 20:55:25 2018
Thread-6 Sat Apr 14 20:55:30 2018
Thread-7 Sat Apr 14 20:55:30 2018
Thread-8 Sat Apr 14 20:55:30 2018
Thread-9 Sat Apr 14 20:55:30 2018
Thread-10 Sat Apr 14 20:55:30 2018
Thread-11 Sat Apr 14 20:55:35 2018
Thread-12 Sat Apr 14 20:55:35 2018
```

分类: 网路编程

好文要顶 关注我 收藏该文

慕沁
关注 - 36
粉丝 - 20
+加关注

1 0

posted @ 2018-04-14 20:56 慕沁 阅读(2505) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

(评论功能已被禁用)

相关博文:
- Python线程同步锁,信号量
- 线程、GIL、Lock锁、RLock锁、Semaphore锁、同步条件event
- Day12- Python基础12 线程、GIL、Lock锁、RLock锁、Semaphore锁、同步条件event
- python3GIL锁/互斥锁Lock和递归锁Rlock
- python线程/线程锁/信号量