多线程的概念

```
线程也叫轻量级进程，是操作系统能够进行运算调度的最小单位，它被包含在进程中，是进程中的实际运作单位。
线程自身不拥有系统资源，只拥有一些在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源
一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行

线程有就绪，阻塞，运行三种基本状态
就绪：是指线程具备运行的所有条件，逻辑上可以运行，等待处理机
阻塞：是指线程在等待一个事件（如某个信号量），逻辑上不能运行
运行：是指线程占有处理机，正在运行

1.计算密集型任务-多进程

2.计算密集型任务-多线程

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import multiprocessing
import os,time

# 1.计算密集型任务-多进程
# def myfunc():
#     res=0
#     for i in range(10000000):
#         res*=i
#
# if __name__=='__main__':
#     mylist=[]
#     start=time.time()
#     for i in range(4):
#         p=multiprocessing.Process(target=myfunc)
#         p.start()
#         mylist.append(p)
#
#     for i in mylist:
#         i.join()
#
#     end=time.time()
#     print("使用进程需要:",end-start)#2.139927864074707


# 2.计算密集型任务-多线程
# import threading
```

```
# def myfunc():
#     res=0
#     for i in range(10000000):
#         res*=i
#
# if __name__=='__main__':
#     mylist=[]
#     start=time.time()
#     for i in range(4):
#         p=threading.Thread(target=myfunc)
#         p.start()
#         mylist.append(p)
#
#     for i in mylist:
#         i.join()
#
#     end=time.time()
#     print("使用线程需要:",end-start)#3.2804999351501465

#执行密集型计算任务时,多进程更快


3.计算IO型任务-多进程

4.计算IO型任务-多线程

# def myopen():
#     time.sleep(2)
#     print("---->",file=open('1.txt','w'))
#
# if __name__=='__main__':
#     mylist=[]
#     start=time.time()
#     for i in range(40):
#         p=multiprocessing.Process(target=myopen)
#         p.start()
#         mylist.append(p)
#
#     for i in mylist:
#         i.join()
#
#     end=time.time()
#     print("使用进程需要:",end-start)#7.486999988555908

# 4.计算IO型任务-多线程
import threading
def myopen():
    time.sleep(2)
    print("---->",file=open('1.txt','w'))

if __name__=='__main__':
    mylist=[]
    start=time.time()
```

```
    for i in range(40):
        p=threading.Thread(target=myopen)
        p.start()
        mylist.append(p)

    for i in mylist:
        i.join()

    end=time.time()
    print("使用线程需要:",end-start)#2.067413568496704

#总结:针对IO型任务,使用多线程更快
```

创建线程

```
创建线程有2种方式，一种是实例化threading.Thread类，一种是继承threading.Thread，在子类中重写run和init方法

第一种方式:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import time
import threading
def mytest(n):
    print(f"线程名称为:{threading.current_thread().name} 开始执行")
    time.sleep(3)
    print(f"线程名称为:{threading.current_thread().name} 结束执行")

if __name__=='__main__':
    t=threading.Thread(target=mytest,args=(1,))
    t2=threading.Thread(target=mytest,args=(2,))
    t3=threading.Thread(target=mytest,args=(3,))

    t.start()
    t2.start()
    t3.start()

    t.join()
    t2.join()
    t3.join()


第二种方式：

import time
import threading

```

```
class Maker(threading.Thread):
    def __init__(self,n):
        super().__init__()
        self.n=n

    def run(self):
        print(f"线程名称为:{threading.current_thread().name} 开始执行")
        time.sleep(3)
        print(f"线程名称为:{threading.current_thread().name} 结束执行")


if __name__=='__main__':
    t1=Maker(1)
    t2 = Maker(2)
    t3 = Maker(3)

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()
```

第二种方式（run中调用业务函数）

```
import time
import threading

def mytest(n):
    print(f"线程名称为:{threading.current_thread().name} 开始执行")
    time.sleep(3)
    print(f"线程名称为:{threading.current_thread().name} 结束执行")

class Maker(threading.Thread):
    def __init__(self,n):
        super().__init__()
        self.n=n

    def run(self):
        mytest(self.n)


if __name__=='__main__':
    t1=Maker(1)
    t2 = Maker(2)
    t3 = Maker(3)

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
```

```
    t3.join()
```

多线程同步-Lock

```
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，这个时候就需要使用互斥锁

1.不加锁
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading

num=0

def mytest(n):
    global num
    for i in range(100000):
        num=num+n
        num=num-n

if __name__=='__main__':
    t1=threading.Thread(target=mytest,args=(6,))
    t2=threading.Thread(target=mytest,args=(17,))
    t3=threading.Thread(target=mytest,args=(11,))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()
    #如果不加锁,每次运行后,全局变量的值都不同
    print("num=",num)


2.加锁
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading

```

```
num=0
#定义锁
lock=threading.Lock()
def mytest(n):
    global num
    #加锁
    lock.acquire()
    for i in range(100000):
        num=num+n
        num=num-n
    #解锁
    lock.release()

if __name__=='__main__':
    t1=threading.Thread(target=mytest,args=(6,))
    t2=threading.Thread(target=mytest,args=(17,))
    t3=threading.Thread(target=mytest,args=(11,))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()
    #如果不加锁,每次运行后,全局变量的值都不同
    print("num=",num)


#Lock叫原始锁，Rlock叫重入锁
#1.Lock在锁定时不属于特定线程，也就是说，Lock可以在一个线程中上锁，在另一个线程中解锁。
#而对于RLock来说，只有当前线程才能释放本线程上的锁，即解铃还须系铃人
#2.RLock允许在同一线程中被多次acquire。而Lock却不允许这种情况
lock原始锁的特点:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import threading

lock=threading.Lock()
def mytest02():
    lock.acquire()

def mytest():
    lock.acquire()
    mytest02()


t=threading.Thread(target=mytest)
```

```
102
103 t.start()
104 t.join()
105 #不能在一个线程中连续加2次锁,会阻塞住
106
107 #!/usr/bin/env python
108 # -*- coding:utf-8 -*-
109 #====#====#====#====
110 #Author:
111 #CreatDate:
112 #Version:
113 #====#====#====#====
114 import threading
115 import time
116 lock=threading.Lock()
117 def mytest02():
118     time.sleep(5)
119     lock.release()
120
121 def mytest():
122     lock.acquire()
123     print(threading.current_thread().name)
124
125
126
127 t=threading.Thread(target=mytest)
128 t2=threading.Thread(target=mytest)
129 t3=threading.Thread(target=mytest02)#t3线程给t解锁,让t2可以去执行任务
130
131 t.start()
132 t2.start()
133 t3.start()
134 t.join()
135 t2.join()
136 t3.join()
137 #一个线程加锁,另一个线程可以给他解锁
138
139 3.重入锁
140 重入锁和原始锁差不多,只是他们有各种的特点
141
142 4.重入锁的特点
143 1.一个线程可以多次加锁
144 2.一个线程中的锁,只能本线程解锁
145 #!/usr/bin/env python
146 # -*- coding:utf-8 -*-
147 #====#====#====#====
148 #Author:
149 #CreatDate:
150 #Version:
151 #====#====#====#====
152 import threading
153 import time
154 lock=threading.RLock()
```

```
def mytest02():
    time.sleep(5)
    lock.release()

def mytest():
    lock.acquire()
    print(threading.current_thread().name)



t=threading.Thread(target=mytest)
t2=threading.Thread(target=mytest)
t3=threading.Thread(target=mytest02)#t3线程给t解锁,让t2可以去执行任务

t.start()
t2.start()
t3.start()
t.join()
t2.join()
t3.join()
#以上代码执行后报错,因为重入锁只能本线程解锁


5.重入锁的应用
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading

lock=threading.RLock()
def mytest02():
    lock.acquire()

def mytest():
    lock.acquire()
    mytest02()


t=threading.Thread(target=mytest)

t.start()
t.join()


```

多线程同步-信号量

```
```

互斥锁只能允许一个线程访问共享数据，信号量可以同时允许一定数量的线程访问共享数据

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading
import time
#创建信号量对象,用于控制线程的并发数
sem=threading.BoundedSemaphore(5)

def mytest(n):
    sem.acquire()
    time.sleep(3)
    print(f"{time.strftime('%H:%M:%S')}:{n} 在办理业务")
    sem.release()

mylist=[]
for i in range(12):
    t=threading.Thread(target=mytest,args=(i,))
    t.start()
    mylist.append(t)

for i in mylist:
    i.join()


```

多线程同步-条件对象-Condition

条件对象condition能让一个线程A停下来，等待其他线程B，线程B满足了某个条件后通知线程A继续运行。

具体步骤：
线程首先获取一个条件变量锁，如果条件不足，则该线程等待（wait）并释放条件变量锁；如果条件满足，就继续执行线程，执行完成后可以通知(notify)其他状态为wait的线程执行。其他处于wait状态的线程接到通知后会重新判断条件以确定是否继续执行

acquire: 请求锁
release：释放锁
wait: 线程挂起，等待被唤醒（notify或notifyAll），可以设置等待超时时间
notify：唤醒等待线程，里面可以指定唤醒几个等待线程，比如设置n=3，则表示随机唤醒等待的三个线程。
notify_all: 唤醒所有的等待线程。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
```

```
#Version:
#====#====#====#====

import threading
class Boy(threading.Thread):
    def __init__(self,cd,name):
        super().__init__()
        self.cd=cd
        self.name=name

    def run(self):
        #加锁,为后面的wait准备
        self.cd.acquire()
        print(self.name+":嫁给我吧!")
        #唤醒翠花
        self.cd.notify()
        #自己暂停,等待翠花回应
        self.cd.wait()
        print(self.name+"我单膝下跪,向最漂亮的翠花求婚,并送上砖戒")
        # 唤醒翠花
        self.cd.notify()
        self.cd.wait()
        print(self.name+"你的选择非常明智")
        self.cd.release()#释放锁

class Girl(threading.Thread):
    def __init__(self,cd,name):
        super().__init__()
        self.cd = cd
        self.name = name

    def run(self):
        #加锁
        self.cd.acquire()
        self.cd.wait()#等待二牛求婚
        print(self.name+"没有情调,太直男,不够浪漫,不答应")
        self.cd.notify()#唤醒二牛
        self.cd.wait()#等待二牛做浪漫的事情
        print(self.name+"好吧,答应你")
        self.cd.notify()  # 唤醒二牛
        self.cd.release()#释放锁

#创建条件对象
cd=threading.Condition()
boy=Boy(cd,"二牛")
girl=Girl(cd,"翠花")

#开启线程
girl.start()
boy.start()
```

```

```

多线程同步-事件-event

事件用于线程之间的通信，一个线程发出一个信号，其他一个或多个线程等待，调用Event对象的wait方法，线程会阻塞，
直到别的线程set后才会被唤醒

原理：
事件对象管理一个内部标志，通过set()方法将其设置为True，并使用clear()方法将其设置为False。wait()方法阻塞，直到标志为True。该标志初始为False。

Event内部默认有一个标志，初始值为Flase,A对象的wait()方法进入等待状态，B对象调用set()方法将内置的标志设置为True时，A对象就继续运行，clear()方法可以再将标志设置为Flase

is_set: 获取内部标志状态
set: 将内部标志设置为True。所有等待的线程都被唤醒
clear：将内部标志重置为False
wait：阻塞直到内部标志为true，可以设置等待超时时间。

注意：wait不会将内部标志修改为false，如果内部标志本来就为true，调用wait不会被阻塞。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading
import time
class Boy(threading.Thread):
    def __init__(self,cd,name):
        super().__init__()
        self.cd=cd
        self.name=name

    def run(self):
        print(self.name+":嫁给我吧!")
        #唤醒翠花
        self.cd.set()
        time.sleep(0.5)#暂停0.5秒,让翠花把flag设置为False
        #自己暂停,等待翠花回应
        self.cd.wait()
        self.cd.clear()
        print(self.name+"我单膝下跪,向最漂亮的翠花求婚,并送上砖戒")
        # 唤醒翠花
        self.cd.set()
        time.sleep(0.5)
```

```
        self.cd.wait()
        print(self.name+"你的选择非常明智")


class Girl(threading.Thread):
    def __init__(self,cd,name):
        super().__init__()
        self.cd = cd
        self.name = name

    def run(self):
        self.cd.wait()#等待二牛求婚
        self.cd.clear()#让flag为False,让二牛的wait有效果
        print(self.name+"没有情调,太直男,不够浪漫,不答应")
        self.cd.set()#唤醒二牛
        time.sleep(0.5)#让二牛把flag设置为False
        self.cd.wait()#等待二牛做浪漫的事情
        self.cd.clear()
        print(self.name+"好吧,答应你")
        self.cd.set()  # 唤醒二牛

#创建时间对象
cd=threading.Event()
boy=Boy(cd,"二牛")
girl=Girl(cd,"翠花")

#开启线程
girl.start()
boy.start()
```

Event和condition最大的区别在于，condition调用wait的时候肯定会被阻塞，直到另外一个线程调用notify或notifyall将其唤醒，但是event不会调用wait不见得被阻塞，只有当内部标志为false的时候，event调用wait才会被阻塞。Event就好比十字路口的交通信号灯，绿灯的时候所有车辆必须通行（也就是没法阻塞，你不走后面的车主会揍你的），红灯的时候所有车辆都得等待。而condition就好比沉睡的公主，她睡着后（wait）必须有人将她唤醒（notify/notifyall），否则会一直沉睡（阻塞）。

多线程优先级队列-queue

```
先进先出（queue）
后进先出（LifoQueue）
优先队列(PriorityQueue)
```

put 方法用以插入数据到队列中，put 方法还有两个可选参数:blocked 和 timeout。如果blocked为True(默认值)，并且 timeout 为正值，则该方法会阻塞timeout 指定的时间，直到该队列有剩余的空间。如果超时，则会抛出 Queue.Full 异常。如果blocked 为 False，但该 Queue 已满，则会立即抛出 Queue.Full 异常。

get 方法可以从队列读取并删除一个元素。同样，get方法有两个可选参数:blocked 和 timeout。如果 blocked 为 True 默认值)，并且 timeout 为正值，在等待时间内没有取到任何元素，则会抛出 Queue.Empty 天异常。如果 blocked 为False，那么将会有两种情况存在: Queue 有一个值可用，立即返回该值，否则队列为空，立即抛出 Queue.Empty异常。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import threading
import time
import queue

#创建队列队象
q=queue.Queue(maxsize=500)

def mytest():
    for i in range(500):
        q.put("书本-"+str(i))

    while True:
        q.put("书本")
        time.sleep(1)

def mytest02():
    while True:
        msg=q.get()
        print(msg)
        time.sleep(1)


t1=threading.Thread(target=mytest)
t2=threading.Thread(target=mytest02)

t1.start()
t2.start()
t1.join()
t2.join()
```

线程池-pool

在面向对象编程中，创建和销毁对象是很费时间的的，因为创建一个对象要获取内存资源或其他更多资源。虚拟机也将试图跟踪每一个对象，以便更能够在对象销毁后进行垃圾回收。同样的道理，多任务情况下每次都会生成一个新线程，执行任务后资源再被回收就显得非常低效，因此线程池就是解决这个问题的办法。类似的例子还有连接池、进程池等。

将任务添加到线程池中，线程池会自动指定一个空空闲的线程去执行任务，当超过线程池的最大线程数时，任务需要等待有新的空闲线程后才会被执行。

下面的代码是比较一个耗时2秒的任务,顺序执行和用线程池中的线程来执行,哪个效率高

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
from multiprocessing.dummy import Pool as ThreadPool

import time

def mytest(n):
    print("n=",n)
    time.sleep(2)

#主线程,调用5次mytest
start=time.time()
for i in range(5):
    mytest(i)
end=time.time()
print("顺序执行的时间为:",end-start)#10.002500057220459

start2=time.time()
p=ThreadPool(processes=5)
res=p.map(mytest,range(5))
p.close()
p.join()
end2=time.time()
print("线程池的时间为:",end2-start2)#2.109921932220459
```