

多线程的概念

```
1  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。
2  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
3  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
4
5  线程有就绪，阻塞，运行三种基本状态
6  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
7  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
8  运行：是指线程占有处理机，正在运行
9
10 1.计算密集型任务-多进程
11
12 2.计算密集型任务-多线程
13
14 #!/usr/bin/env python
15 # -*- coding:utf-8 -*-
16 #====#====#====#====
17 #Author:
18 #CreatDate:
19 #Version:
20 #====#====#====#====
21 import multiprocessing
22 import os,time
23
24 # 1.计算密集型任务-多进程
25 # def myfunc():
26 #     res=0
27 #     for i in range(10000000):
28 #         res*=i
29 #
30 # if __name__=='__main__':
31 #     mylist=[]
32 #     start=time.time()
33 #     for i in range(4):
34 #         p=multiprocessing.Process(target=myfunc)
35 #         p.start()
36 #         mylist.append(p)
37 #
38 #     for i in mylist:
39 #         i.join()
40 #
41 #     end=time.time()
42 #     print("使用进程需要:",end-start)#2.139927864074707
43
44
45 # 2.计算密集型任务-多线程
46 # import threading
```

```

47 # def myfunc():
48 #     res=0
49 #     for i in range(10000000):
50 #         res*=i
51 #
52 # if __name__=='__main__':
53 #     mylist=[]
54 #     start=time.time()
55 #     for i in range(4):
56 #         p=threading.Thread(target=myfunc)
57 #         p.start()
58 #         mylist.append(p)
59 #
60 #     for i in mylist:
61 #         i.join()
62 #
63 #     end=time.time()
64 #     print("使用线程需要:",end-start)#3.2804999351501465
65
66 #执行密集型计算任务时,多进程更快
67
68
69 3.计算IO型任务-多进程
70
71 4.计算IO型任务-多线程
72
73 # def myopen():
74 #     time.sleep(2)
75 #     print("---->",file=open('1.txt','w'))
76 #
77 # if __name__=='__main__':
78 #     mylist=[]
79 #     start=time.time()
80 #     for i in range(40):
81 #         p=multiprocessing.Process(target=myopen)
82 #         p.start()
83 #         mylist.append(p)
84 #
85 #     for i in mylist:
86 #         i.join()
87 #
88 #     end=time.time()
89 #     print("使用进程需要:",end-start)#7.486999988555908
90
91 # 4.计算IO型任务-多线程
92 import threading
93 def myopen():
94     time.sleep(2)
95     print("---->",file=open('1.txt','w'))
96
97 if __name__=='__main__':
98     mylist=[]
99     start=time.time()

```

```

100     for i in range(40):
101         p=threading.Thread(target=myopen)
102         p.start()
103         mylist.append(p)
104
105     for i in mylist:
106         i.join()
107
108     end=time.time()
109     print("使用线程需要:",end-start)#2.067413568496704
110
111     #总结:针对IO型任务,使用多线程更快

```

创建线程

```

1  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
2
3  第一种方式:
4  #!/usr/bin/env python
5  # -*- coding:utf-8 -*-
6  #====#====#====#====
7  #Author:
8  #CreatDate:
9  #Version:
10 #====#====#====#====
11 import time
12 import threading
13 def mytest(n):
14     print(f"线程名称为:{threading.current_thread().name} 开始执行")
15     time.sleep(3)
16     print(f"线程名称为:{threading.current_thread().name} 结束执行")
17
18 if __name__=='__main__':
19     t=threading.Thread(target=mytest,args=(1,))
20     t2=threading.Thread(target=mytest,args=(2,))
21     t3=threading.Thread(target=mytest,args=(3,))
22
23     t.start()
24     t2.start()
25     t3.start()
26
27     t.join()
28     t2.join()
29     t3.join()
30
31
32 第二种方式:
33
34 import time
35 import threading
36

```

```
37 class Maker(threading.Thread):
38     def __init__(self,n):
39         super().__init__()
40         self.n=n
41
42     def run(self):
43         print(f"线程名称为:{threading.current_thread().name} 开始执行")
44         time.sleep(3)
45         print(f"线程名称为:{threading.current_thread().name} 结束执行")
46
47
48 if __name__=='__main__':
49     t1=Maker(1)
50     t2 = Maker(2)
51     t3 = Maker(3)
52
53     t1.start()
54     t2.start()
55     t3.start()
56
57     t1.join()
58     t2.join()
59     t3.join()
60
61 第二种方式 ( run中调用业务函数 )
62 import time
63 import threading
64
65 def mytest(n):
66     print(f"线程名称为:{threading.current_thread().name} 开始执行")
67     time.sleep(3)
68     print(f"线程名称为:{threading.current_thread().name} 结束执行")
69
70 class Maker(threading.Thread):
71     def __init__(self,n):
72         super().__init__()
73         self.n=n
74
75     def run(self):
76         mytest(self.n)
77
78
79 if __name__=='__main__':
80     t1=Maker(1)
81     t2 = Maker(2)
82     t3 = Maker(3)
83
84     t1.start()
85     t2.start()
86     t3.start()
87
88     t1.join()
89     t2.join()
```

多线程同步-Lock

```
1  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
2
3  1.不加锁
4  #!/usr/bin/env python
5  # -*- coding:utf-8 -*-
6  #====#====#====#====
7  #Author:
8  #CreatDate:
9  #Version:
10 #====#====#====#====
11
12 import threading
13
14 num=0
15
16 def mytest(n):
17     global num
18     for i in range(100000):
19         num=num+n
20         num=num-n
21
22 if __name__=='__main__':
23     t1=threading.Thread(target=mytest,args=(6,))
24     t2=threading.Thread(target=mytest,args=(17,))
25     t3=threading.Thread(target=mytest,args=(11,))
26
27     t1.start()
28     t2.start()
29     t3.start()
30
31     t1.join()
32     t2.join()
33     t3.join()
34     #如果不加锁,每次运行后,全局变量的值都不同
35     print("num=",num)
36
37
38 2.加锁
39 #!/usr/bin/env python
40 # -*- coding:utf-8 -*-
41 #====#====#====#====
42 #Author:
43 #CreatDate:
44 #Version:
45 #====#====#====#====
46
47 import threading
48
```

```

49 num=0
50 #定义锁
51 lock=threading.Lock()
52 def mytest(n):
53     global num
54     #加锁
55     lock.acquire()
56     for i in range(100000):
57         num=num+n
58         num=num-n
59     #解锁
60     lock.release()
61
62 if __name__=='__main__':
63     t1=threading.Thread(target=mytest,args=(6,))
64     t2=threading.Thread(target=mytest,args=(17,))
65     t3=threading.Thread(target=mytest,args=(11,))
66
67     t1.start()
68     t2.start()
69     t3.start()
70
71     t1.join()
72     t2.join()
73     t3.join()
74     #如果不加锁,每次运行后,全局变量的值都不同
75     print("num=",num)
76
77
78 #Lock叫原始锁,Rlock叫重入锁
79 #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。
80 #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
81 #2.Rlock允许在同一线程中被多次acquire。而Lock却不允许这种情况
82 Lock原始锁的特点:
83 #!/usr/bin/env python
84 # -*- coding:utf-8 -*-
85 #====#====#====#====
86 #Author:
87 #CreateDate:
88 #Version:
89 #====#====#====#====
90 import threading
91
92 lock=threading.Lock()
93 def mytest02():
94     lock.acquire()
95
96 def mytest():
97     lock.acquire()
98     mytest02()
99
100
101 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()

```

```

155 def mytest02():
156     time.sleep(5)
157     lock.release()
158
159 def mytest():
160     lock.acquire()
161     print(threading.current_thread().name)
162
163
164
165 t=threading.Thread(target=mytest)
166 t2=threading.Thread(target=mytest)
167 t3=threading.Thread(target=mytest02)#t3线程给t解锁,让t2可以去执行任务
168
169 t.start()
170 t2.start()
171 t3.start()
172 t.join()
173 t2.join()
174 t3.join()
175 #以上代码执行后报错,因为重入锁只能本线程解锁
176
177
178 5.重入锁的应用
179 #!/usr/bin/env python
180 # -*- coding:utf-8 -*-
181 #====#====#====#====
182 #Author:
183 #CreatDate:
184 #Version:
185 #====#====#====#====
186
187 import threading
188
189 lock=threading.RLock()
190 def mytest02():
191     lock.acquire()
192
193 def mytest():
194     lock.acquire()
195     mytest02()
196
197
198 t=threading.Thread(target=mytest)
199
200 t.start()
201 t.join()
202
203

```

多线程同步-信号量


```

1 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
2
3 #!/usr/bin/env python
4 # -*- coding:utf-8 -*-
5 #====#====#====#====
6 #Author:
7 #CreatDate:
8 #Version:
9 #====#====#====#====
10
11 import threading
12 import time
13 #创建信号量对象,用于控制线程的并发数
14 sem=threading.BoundedSemaphore(5)
15
16 def mytest(n):
17     sem.acquire()
18     time.sleep(3)
19     print(f"{time.strftime('%H:%M:%S')}:{n} 在办理业务")
20     sem.release()
21
22 mylist=[]
23 for i in range(12):
24     t=threading.Thread(target=mytest,args=(i,))
25     t.start()
26     mylist.append(t)
27
28 for i in mylist:
29     i.join()
30
31

```

多线程同步-条件对象-Condition

```

1 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。
2
3 具体步骤：
4 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
5
6 acquire: 请求锁
7 release: 释放锁
8 wait: 线程挂起，等待被唤醒（notify或notifyAll），可以设置等待超时时间
9 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。
10 notify_all: 唤醒所有的等待线程。
11
12 #!/usr/bin/env python
13 # -*- coding:utf-8 -*-
14 #====#====#====#====
15 #Author:
16 #CreatDate:

```

```
17 #Version:
18 #====#====#====#====
19
20 import threading
21 class Boy(threading.Thread):
22     def __init__(self,cd,name):
23         super().__init__()
24         self.cd=cd
25         self.name=name
26
27     def run(self):
28         #加锁,为后面的wait准备
29         self.cd.acquire()
30         print(self.name+":嫁给我吧!")
31         #唤醒翠花
32         self.cd.notify()
33         #自己暂停,等待翠花回应
34         self.cd.wait()
35         print(self.name+"我单膝下跪,向最漂亮的翠花求婚,并送上钻戒")
36         # 唤醒翠花
37         self.cd.notify()
38         self.cd.wait()
39         print(self.name+"你的选择非常明智")
40         self.cd.release()#释放锁
41
42 class Girl(threading.Thread):
43     def __init__(self,cd,name):
44         super().__init__()
45         self.cd = cd
46         self.name = name
47
48     def run(self):
49         #加锁
50         self.cd.acquire()
51         self.cd.wait()#等待二牛求婚
52         print(self.name+"没有情调,太直男,不够浪漫,不答应")
53         self.cd.notify()#唤醒二牛
54         self.cd.wait()#等待二牛做浪漫的事情
55         print(self.name+"好吧,答应你")
56         self.cd.notify() # 唤醒二牛
57         self.cd.release()#释放锁
58
59 #创建条件对象
60 cd=threading.Condition()
61 boy=Boy(cd,"二牛")
62 girl=Girl(cd,"翠花")
63
64 #开启线程
65 girl.start()
66 boy.start()
67
68
69
```

多线程同步-事件-event

```
1  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，
2  直到别的线程set后才会被唤醒
3
4  原理：
5  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。
6
7  Event内部默认有一个标志，初始值为False,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False
8
9  is_set：获取内部标志状态
10 set：将内部标志设置为True。所有等待的线程都被唤醒
11 clear：将内部标志重置为False
12 wait：阻塞直到内部标志为true，可以设置等待超时时间。
13
14 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。
15 #!/usr/bin/env python
16 # -*- coding:utf-8 -*-
17 #====#====#====#====
18 #Author:
19 #CreatDate:
20 #Version:
21 #====#====#====#====
22
23 import threading
24 import time
25 class Boy(threading.Thread):
26     def __init__(self,cd,name):
27         super().__init__()
28         self.cd=cd
29         self.name=name
30
31     def run(self):
32         print(self.name+":嫁给我吧!")
33         #唤醒翠花
34         self.cd.set()
35         time.sleep(0.5)#暂停0.5秒,让翠花把flag设置为False
36         #自己暂停,等待翠花回应
37         self.cd.wait()
38         self.cd.clear()
39         print(self.name+"我单膝下跪,向最漂亮的翠花求婚,并送上钻戒")
40         # 唤醒翠花
41         self.cd.set()
42         time.sleep(0.5)
```

```

43         self.cd.wait()
44         print(self.name+"你的选择非常明智")
45
46
47     class Girl(threading.Thread):
48         def __init__(self,cd,name):
49             super().__init__()
50             self.cd = cd
51             self.name = name
52
53         def run(self):
54             self.cd.wait()#等待二牛求婚
55             self.cd.clear()#让flag为False,让二牛的wait有效果
56             print(self.name+"没有情调,太直男,不够浪漫,不答应")
57             self.cd.set()#唤醒二牛
58             time.sleep(0.5)#让二牛把flag设置为False
59             self.cd.wait()#等待二牛做浪漫的事情
60             self.cd.clear()
61             print(self.name+"好吧,答应你")
62             self.cd.set() # 唤醒二牛
63
64     #创建时间对象
65     cd=threading.Event()
66     boy=Boy(cd,"二牛")
67     girl=Girl(cd,"翠花")
68
69     #开启线程
70     girl.start()
71     boy.start()
72
73
74
75
76
77
78     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）。
79

```

多线程优先级队列-queue

- 1 先进先出（queue）
- 2 后进先出（LifoQueue）
- 3 优先队列（PriorityQueue）
- 4

```
5 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。
6
7 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。
8
9 #!/usr/bin/env python
10 # -*- coding:utf-8 -*-
11 #====#====#====#====
12 #Author:
13 #CreatDate:
14 #Version:
15 #====#====#====#====
16
17 import threading
18 import time
19 import queue
20
21 #创建队列对象
22 q=queue.Queue(maxsize=500)
23
24 def mytest():
25     for i in range(500):
26         q.put("书本-"+str(i))
27
28     while True:
29         q.put("书本")
30         time.sleep(1)
31
32 def mytest02():
33     while True:
34         msg=q.get()
35         print(msg)
36         time.sleep(1)
37
38
39 t1=threading.Thread(target=mytest)
40 t2=threading.Thread(target=mytest02)
41
42 t1.start()
43 t2.start()
44 t1.join()
45 t2.join()
46
47
48
```

线程池-pool

1 在面向对象编程中，创建和销毁对象是很费时间的，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。

2 将任务添加到线程池中，线程池会自动指定一个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。

3

4 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

5

6 `#!/usr/bin/env python`

7 `# -*- coding:utf-8 -*-`

8 `#====#====#====#====`

9 `#Author:`

10 `#CreateDate:`

11 `#Version:`

12 `#====#====#====#====`

13 `from multiprocessing.dummy import Pool as ThreadPool`

14

15 `import time`

16

17 `def mytest(n):`

18 `print("n=",n)`

19 `time.sleep(2)`

20

21 `#主线程,调用5次mytest`

22 `start=time.time()`

23 `for i in range(5):`

24 `mytest(i)`

25 `end=time.time()`

26 `print("顺序执行的时间为:",end-start)#10.002500057220459`

27

28 `start2=time.time()`

29 `p=ThreadPool(processes=5)`

30 `res=p.map(mytest,range(5))`

31 `p.close()`

32 `p.join()`

33 `end2=time.time()`

34 `print("线程池的时间为:",end2-start2)#2.109921932220459`

35

36

37

38