

## 为什么要有线程

## 什么是线程

## 线程和进程之间的关系

## 线程的特点

## Python中的线程

- 进程是计算机中最小的资源分配单位
  - 进程对于操作系统来说还是有一定的负担
  - 创建一个进程 操作系统要分配的资源大致有：
    - 代码
    - 数据
    - 文件
1. 为什么要有线程？
    - a. 轻量级的概念
    - b. 他没有属于自己的资源
      - i. 一条线程只负责执行代码
      - ii. 没有自己独立的代码、变量、文件资源
  2. 什么是线程？
    - a. 线程是计算机中被cpu调度的最小单位
    - b. 我们的计算机当中的cpu都是执行的线程中的代码
  3. 进程和线程之间的关系
    - a. 每一个进程中都有至少一条线程在工作（就像每一辆车要有一个司机开）
  4. 线程的特点
    - a. 同一个进程中的所有线程的资源是共享的
    - b. 轻量级 没有自己的资源
    - c. 可以并发执行
  5. 线程和进程的区别
    - a. 占用的资源
    - b. 调度的效率

### c. 资源是否共享

问题：一个进程中的多个线程能够并行吗？（Python中不行）（java c++ c# 是可以的）  
见ppt

### 6. Python线程模块（threading）

```
1 from threading import Thread
2 from multiprocessing import Process
3 import time
4 import os
5
6 # def func(i):
7 #     print("我是线程{}".format(i))
8
9
10 # if __name__ == '__main__':
11 #     for i in range(10):
12 #         t = Thread(target=func, args=(i, ))
13 #         t.start()
14
15
16 # 为什么会轻量级？
17 # if __name__ == '__main__':
18 #     start1 = time.time()
19 #     t_lis = []
20 #     for i in range(100):
21 #         t = Thread(target=func, args=(i, ))
22 #         t.start()
23 #         t_lis.append(t)
24 #     for t in t_lis:
25 #         t.join()
26 #     tt = time.time() - start1
27 #
28 #     start2 = time.time()
29 #     p_lis = []
30 #     for i in range(100):
31 #         p = Process(target=func, args=(i,))
32 #         p.start()
33 #         p_lis.append(p)
```

```

34 #     for p in p_lis:
35 #         p.join()
36 #     pp = time.time() - start2
37 #
38 #     print('启动100个线程需要: %s' % tt)
39 #     print('启动100个进程需要: %s' % pp)
40
41 # 数据共享
42 num = 100
43
44 def func():
45     global num
46     num -= 1
47
48 t_lis = []
49 for i in range(100):
50     t = Thread(target=func)
51     t.start()
52     t_lis.append(t)
53 for t in t_lis:
54     t.join()
55
56 print(num)

```

## 7. 守护线程

```

1 import time
2 from threading import Thread
3
4
5 def func1():
6     while True:
7         time.sleep(0.5)
8         print(123)
9
10
11 def func2():
12     print('func2, start')
13     time.sleep(3)

```

```

14     print('func2 end')
15
16
17 t1 = Thread(target=func1)
18 t2 = Thread(target=func2)
19 # 设置守护线程
20 t1.setDaemon(True)
21 t1.start()
22 t2.start()
23 print('主线程的代码结束了')
24
25
26 # 结论:
27 # 守护线程是在主线程代码结束之后, 还等待了子线程结束才结束的
28 # 主线程结束, 就意味着主进程的结束
29 # 主线程等待所有的线程结束
30 # 主线程结束之后, 守护线程随着主进程的结束自然结束的

```

## 8. 线程的安全问题

### a. 写一个线程不安全的情况

```

1  from threading import Thread
2
3  num = 0
4
5
6  def func1():
7      global num
8      for i in range(1000000):
9          num -= 1
10
11 def func2():
12     global num
13     for i in range(1000000):
14         num += 1
15
16
17 if __name__ == '__main__':
18     t_lis = []

```

```

19     for i in range(10):
20         t1 = Thread(target=func1)
21         t2 = Thread(target=func2)
22         t1.start()
23         t2.start()
24         t_lis.append(t1)
25         t_lis.append(t2)
26     for t in t_lis:
27         t.join()
28
29     print("num =", num)

```

怎么解决上面的问题？（加上互斥锁）

```

1  from threading import Thread, Lock
2
3  num = 0
4
5
6  def func1(lock):
7      global num
8      for i in range(1000000):
9          lock.acquire()
10         num -= 1
11         lock.release()
12
13  def func2(lock):
14      global num
15      for i in range(1000000):
16          lock.acquire()
17          num += 1
18          lock.release()
19
20
21  if __name__ == '__main__':
22      lock = Lock()
23      t_lis = []
24      for i in range(10):
25          t1 = Thread(target=func1, args=(lock,))
26          t2 = Thread(target=func2, args=(lock,))

```

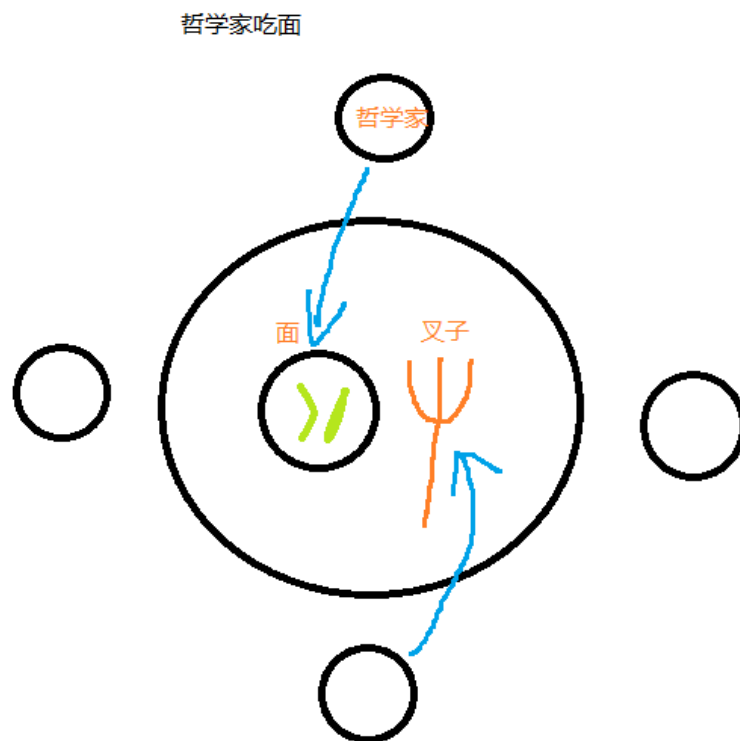
```

27     t1.start()
28     t2.start()
29     t_lis.append(t1)
30     t_lis.append(t2)
31     for t in t_lis:
32         t.join()
33
34     print("num =", num)

```

递归锁

举个例子



写一个死锁程序

```

1  import time
2  from threading import Thread, Lock
3
4  noodle_lock = Lock() # 面锁
5  fork_lock = Lock() # 叉子锁
6
7  def eat1(name):
8      noodle_lock.acquire()
9      print('%s拿到面条了' % name)
10     fork_lock.acquire()

```

```

11     print('%s拿到叉子了' % name)
12
13     print('%s吃面' % name)
14     time.sleep(3)
15     fork_lock.release()
16     print('%s放下叉子' % name)
17     noodle_lock.release()
18     print('%s放下面' % name)
19
20 def eat2(name):
21     fork_lock.acquire()
22     print('%s拿到叉子了' % name)
23     noodle_lock.acquire()
24     print('%s拿到面条了' % name)
25     print('%s吃面' % name)
26     time.sleep(3)
27     noodle_lock.release()
28     print('%s放下面' % name)
29     fork_lock.release()
30     print('%s放下叉子' % name)
31
32
33 if __name__ == '__main__':
34     name_list1 = ["尹精赛", "郑俊杰"]
35     name_list2 = ["阿莫西林", "小明哥"]
36     for name in name_list1:
37         Thread(target=eat1, args=(name, )).start()
38     for name in name_list2:
39         Thread(target=eat2, args=(name, )).start()

```

## 解决死锁现象（使用递归锁）

```

1 import time
2 from threading import Thread, Lock, RLock
3
4 # noodle_lock = Lock() # 面锁
5 # fork_lock = Lock() # 叉子锁
6
7
8 fork_lock = noodle_lock = RLock()

```

```
9
10
11 def eat1(name):
12     noodle_lock.acquire()
13     print('%s拿到面条了' % name)
14     fork_lock.acquire()
15     print('%s拿到叉子了' % name)
16     print('%s吃面' % name)
17     time.sleep(3)
18     fork_lock.release()
19     print('%s放下叉子' % name)
20     noodle_lock.release()
21     print('%s放下面' % name)
22
23 def eat2(name):
24     fork_lock.acquire()
25     print('%s拿到叉子了' % name)
26     noodle_lock.acquire()
27     print('%s拿到面条了' % name)
28     print('%s吃面' % name)
29     time.sleep(3)
30     noodle_lock.release()
31     print('%s放下面' % name)
32     fork_lock.release()
33     print('%s放下叉子' % name)
34
35
36 if __name__ == '__main__':
37     name_list1 = ["尹精赛", "郑俊杰"]
38     name_list2 = ["阿莫西林", "小明哥"]
39     for name in name_list1:
40         Thread(target=eat1, args=(name, )).start()
41     for name in name_list2:
42         Thread(target=eat2, args=(name, )).start()
43
44
45 # 递归锁可以解决互斥锁的死锁问题
46 # 互斥锁
47     # 两把锁
```



```
48      # 多个线程抢
49 # 递归锁
50      # 一把锁
51      # 多个线程抢
52 # 递归锁好不好
53 # 递归锁并不是一个好的解决方案
54 # 死锁现象的发生不是互斥锁的原因
55 # 而是程序员的逻辑有问题
```