

小强多线程和协程入门练习

226 Python 如何创建线程

创建一个线程：

```
import threading
my_thread = threading.Thread()
```

[复制](#)

创建一个名称为 `my_thread` 的线程：

```
my_thread = threading.Thread(name='my_thread')
```

[复制](#)

创建线程，需要告诉这个线程，它能帮助我们做什么。

做什么，是通过参数 `target` 传入，参数类型为 `callable`。

```
In [49]: def print_i(i):
...:     print('打印i:%d'%(i,))

In [50]: my_thread = threading.Thread(target=print_i,args=(1,))
```

[复制](#)

`my_thread` 线程已全副武装，但是，我们得按下发射按钮，启动 `start()`，它才开始真正起飞。

```
In [52]: my_thread.start()
```

[复制](#)

打印结果如下，其中 `args` 指定函数 `print_i` 需要的参数 `i`，类型为元组。

```
打印i:1
```

[复制](#)

227 多线程：交替获得 CPU 时间片

为了更好解释，假定计算机是单核的，尽管对于 `cpython`，这个假定有些多余。

开辟 3 个线程，装载到 `threads` 中：

```
In [1]: import time
In [3]: import threading

In [14]: def print_time():
...:     for _ in range(5): # 在每个线程中打印 5 次
...:         time.sleep(0.1) # 模拟打印前的相关处理逻辑
...:         print('当前线程%s,打印结束时间为:%s' %(threading.current_thread().getName(),time.time()))

In [7]: threads = [threading.Thread(name='t%d'%(i,),target=print_time) for i in range(3)]
```

[复制](#)

启动 3 个线程：

```
In [8]: [t.start() for t in threads]
Out[8]: [None, None, None]
```

[复制](#)

打印结果，如下，

```
当前线程t0,打印结束时间为:1582761727.4976637
当前线程t1,打印结束时间为:1582761727.4976637
当前线程t2,打印结束时间为:1582761727.498664
当前线程t0,打印结束时间为:1582761727.597949
当前线程t1,打印结束时间为:1582761727.597949
当前线程t2,打印结束时间为:1582761727.599801
当前线程t1,打印结束时间为:1582761727.6984522
当前线程t0,打印结束时间为:1582761727.6984522
当前线程t2,打印结束时间为:1582761727.7001588
当前线程t1,打印结束时间为:1582761727.7988598
当前线程t0,打印结束时间为:1582761727.7996202
当前线程t2,打印结束时间为:1582761727.8006535
当前线程t1,打印结束时间为:1582761727.8994005
当前线程t0,打印结束时间为:1582761727.900454
当前线程t2,打印结束时间为:1582761727.9024456
```

[复制](#)

根据操作系统的调度算法，`t0`，`t1`，`t2` 三个线程，轮询获得 CPU 时间片。

228 抢夺全局变量出现的场景

全局变量，被当前进程中所有存活线程共享。这就意味着，抢夺全局变量的问题。

比如下面的例子，创建 10 个线程，它们都会竞争全局变量 `a`：

```
In [9]: import threading

In [10]: a = 0

In [11]: def add1():
...:     global a
...:     a += 1
...:     print('%s adds a to 1: %d'%(threading.current_thread().getName(),a))

In [12]: threads = [threading.Thread(name='t%d'%(i,),target=add1) for i in range(10)]

In [13]: [t.start() for t in threads]
```

[复制](#)

执行结果：

```
t0 adds a to 1: 1
t1 adds a to 1: 2
t2 adds a to 1: 3
```

[复制](#)[226 Python 如何创...](#)[227 多线程：交替...](#)[228 抢夺全局变量...](#)[229 多线程之副作...](#)[230 某些场景多线程...](#)[231 高效的协程及...](#)[232 数数Counter最...](#)[233 说说死锁、GIL...](#)

```
t3  adds a to 1: 4
t4  adds a to 1: 5
t5  adds a to 1: 6
t6  adds a to 1: 7
t7  adds a to 1: 8
t8  adds a to 1: 9
t9  adds a to 1: 10
```

每个线程执行一次，`a` 的值被加 1，最后 `a` 变为 10，结果看起来一切正常。

运行上面代码十几遍，一切也都正常。

所以，能下结论：这段代码是线程安全的吗？

编写多线程程序，只要有读取和修改全局变量的情况，如果不采取措施，就一定不是线程安全的。

尽管，有时某些情况的资源竞争，暴露出问题的概率极低。

如果某个线程修改全局变量 `a` 后，其他线程获取的，还是未修改前的值，问题就会暴露。

但是，`a = a + 1` 这种修改操作，花费的时间太短，短到我们无法想象。

线程间轮询执行时，都能获取到最新的、修改后的值。

所以，暴露问题的概率就变得很低。

不过，现实中使用多线程，目的也不会仅仅就是为了跑一个 `a = a+1` 这种操作。

更大可能，线程中执行任务，会耗费一定时间。

所以，怎样编写线程安全的代码，变得非常重要。

229 多线程之副作用

数据写入数据库操作，一般会耗费可以感知的时间。

为模拟数据写入库动作，简化起见，等效地，延长修改变量 `a` 的时间，问题很快就会还原出来。

```
In [16]: import threading

In [17]: import time

In [18]: a = 0

In [19]: def add1():
...:     global a
...:     tmp = a + 1
...:     time.sleep(0.2) # 延时0.2秒, 模拟写入所需时间
...:     a = tmp
...:     print('%s  adds a to 1: %d'%(threading.current_thread().getName(),a))

In [20]: threads = [threading.Thread(name='t%d'%(i,),target=add1) for i in range(10)]

In [21]: [t.start() for t in threads]
Out[21]: [None, None, None, None, None, None, None, None, None, None]
```

运行代码，仅仅一次，问题就很快完全暴露，结果如下：

```
t2  adds a to 1: 1
t1  adds a to 1: 1
t0  adds a to 1: 1
t6  adds a to 1: 1
t3  adds a to 1: 1
t8  adds a to 1: 1
t5  adds a to 1: 1
t9  adds a to 1: 1
t7  adds a to 1: 1
t4  adds a to 1: 1
```

看到，10 个线程全部运行后，`a` 的值只相当于一个线程执行的结果。为什么？

修改 `a` 前，有 0.2 秒的休眠时间。

某个线程被延时后，CPU 立即分配计算资源给其他线程。

直到所有线程被分配到计算资源，已经运行完 `a = a + 1` 后，根据结果反映出，0.2 秒的休眠时间还没耗尽，这样每个线程获取到的 `a` 值都是 0，所以才出现上面的结果。

以上最核心的三行代码：

```
tmp = a + 1
time.sleep(0.2) # 延时0.2秒, 模拟写入所需时间
a = tmp
```

230 某些场景多线程加锁后变为鸡肋

Python 提供的锁机制，是解决上面问题的方法之一。

某段代码只能单线程执行时，加上锁，其他线程等待，直到被释放后，其他线程再争锁，竞争到锁的线程执行代码，再释放锁，重复此过程，直到所有线程都走一遍竞争到锁和释放锁的过程。

```
In [22]: import threading
In [23]: import time
```

创建一把锁 `locka`：

```
In [24]: locka = threading.Lock()
```

通过 `locka.acquire()` 获得锁，通过 `locka.release()` 释放锁。

获得锁和释放锁之间的代码，只能单线程执行。

```
In [25]: a = 0

In [26]: def add1():
...:     global a
...:     try:
...:         locka.acquire() # 获得锁
...:         tmp = a + 1
...:         time.sleep(0.2) # 延时0.2秒, 模拟写入所需时间
...:         a = tmp
...:     finally:
...:         locka.release() # 释放锁
...:     print('%s adds a to 1: %d'%(threading.current_thread().getName(),a))
```

创建和开始线程：

```
In [27]: threads = [threading.Thread(name='t%d'%(i),target=add1) for i in range(10)]

In [28]: [t.start() for t in threads]
Out[28]: [None, None, None, None, None, None, None, None, None, None]
```

执行结果，如下：

```
t0 adds a to 1: 1
t1 adds a to 1: 2
t2 adds a to 1: 3
t3 adds a to 1: 4
t4 adds a to 1: 5
t5 adds a to 1: 6
t6 adds a to 1: 7
t7 adds a to 1: 8
t8 adds a to 1: 9
t9 adds a to 1: 10
```

打印结果一切正常。

但是，再仔细想想，这已经是单线程顺序执行。

就本案例而言，已经失去多线程的价值。

并且，还带来了因为线程创建开销，浪费时间的副作用。除此之外，还有一个很大风险。

程序中只有一把锁，通过 `try...finally` 还能确保不发生死锁。但是，当程序中启用多把锁，很容易发生死锁。

考虑使用场合，避免死锁，是多线程开发，需要格外注意的一些问题。

231 高效的协程及案例

在同一个线程中，如果发生以下事情：

A 函数执行时被中断，传递一些数据给 B 函数；

B 函数拿到这些数据后开始执行，执行一段时间后，发送一些数据到 A 函数；

就这样交替执行.....

这种执行调用模式，被称为[协程](#)。

可以看到，协程是在同一线程中函数间的切换，而不是线程间的切换，因此执行效率更优，Python 的异步操作正是基于高效的协程机制。

下面通过一个例子，加深对协程的理解。

```
def A():
    a_list = ['1', '2', '3']
    for to_b in a_list:
        from_b = yield to_b
        print('receive %s from B' % (from_b,))
        print('do some complex process for A during 200ms ')

def B(a):
    from_a = a.send(None)
    print('response %s from A ' % (from_a,))
    print('B is analysising data from A')
    b_list = ['x', 'y', 'z']
    try:
        for to_a in b_list:
            from_a = a.send(to_a)
            print('response %s from A ' % (from_a,))
            print('B is analysising data from A')
    except StopIteration:
        print('---from a done---')
    finally:
        a.close()
```

调用：

```
a = A()
B(a)
```

分析执行过程：

- 1 `a.send(None)` 激活 A 函数，并执行到 `yield to_b`，把变量 `to_b` 传递给 B 函数，A 函数中断；
- 2 `from_a` 就是 上步 A 函数返回的 `to_b` 值，然后执行分析这个值；
- 3 当执行到 `a.send(to_a)` 时，B 函数将加工后的 `to_a` 值发送给 A 函数；
- 4 `from_b` 变量接收来自 B 函数的发送，然后使用此值做分析 200 ms 后，又将 `to_b` 传递给 B 函数，A函数中断；
- 5 重复 2、3、4
- 6 直到 `from_a` 获取不到响应值，函数触发 `StopIteration` 异常，程序执行结束。

执行结果：

```
response 1 from A
B is analysising data from A
receive x from B
do some complex process for A during 200ms
response 2 from A
B is analysising data from A
receive y from B
do some complex process for A during 200ms
response 3 from A
B is analysising data from A
receive z from B
do some complex process for A during 200ms
---from A done---
```

通过上述看到，协程是在同一个线程中，不同函数间交替的、协作的执行完成任务。

232 数数Counter最好用

Counter 正如名字那样，它的主要功能就是计数。

我们在分析数据时，基本都会涉及计数，真的家常便饭。

习惯使用 list 的朋友，往往会这样统计：

```
sku_purchase = [3, 8, 3, 10, 3, 3, 1, 3, 7, 6, 1, 2, 7, 0, 7, 9, 1, 5, 1, 0]

d = {}
for i in sku_purchase:
    if d.get(i) is None:
        d[i] = 1
    else:
        d[i] += 1

d_most = dict(sorted(d.items(), key=lambda item: item[1], reverse=True))
print(d_most)

# 最受欢迎的商品(键为商品编号), 排序结果:
# {3: 5, 1: 4, 7: 3, 0: 2, 8: 1, 10: 1, 6: 1, 2: 1, 9: 1, 5: 1}
```

但是，如果使用 Counter，能写出更加简化的代码。

首先，导入 Counter 类：

```
In [35]: from collections import Counter
```

然后，使用一行代码搞定：

```
In [42]: Counter(sku_purchase).most_common()
Out[42]:
[(3, 5), (1, 4), (7, 3), (0, 2), (8, 1), (10, 1), (6, 1), (2, 1), (9, 1), (5, 1)]
```

仅仅一行代码，便输出统计结果。并且，输出按照购买次数的由大到小排序好的列表，比如，编号为3 的商品最受欢迎，一共购买了 5 次。

除此之外，使用 Counter 能快速统计，一句话中单词出现次数，一个单词中字符出现次数。

如下所示，

```
In [41]: Counter('i love python so much').most_common()
Out[41]:
[(' ', 4),
 ('o', 3),
 ('h', 2),
 ('i', 1),
 ('l', 1),
 ('v', 1),
 ('e', 1),
 ('p', 1),
 ('y', 1),
 ('t', 1),
 ('n', 1),
 ('s', 1),
 ('m', 1),
 ('u', 1),
 ('c', 1)]
```

233 说说死锁、GIL 锁、协程

多个子线程在系统资源竞争时，都在等待对方解除占用状态。

比如，线程 A 等待着线程 B 释放锁 b，同时，线程 B 等待着线程 A 释放锁 a。

在这种局面下，线程 A 和线程 B 都相互等待着，无法执行下去，这就是死锁。


为了避免死锁发生，cython 使用 GIL 锁，确保同一时刻只有一个线程在执行，所以其实是伪多线程。

所以，python 里常常使用协程技术来代替多线程。

多进程、多线程的切换是由系统决定，而协程由我们自己决定。

协程无需使用锁，也就不会发生死锁。同时，利用协程的协作特点，高效的完成了原编程模型只能通过多个线程才能完成的任务。

还没有评论





存



