

Python 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的进程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠）-- 这就是线程的退让。

开始学习Python线程

Python中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用thread模块中的start_new_thread()函数来产生新线程。语法如下：

```
thread.start_new_thread ( function, args[, kwargs] )
```

参数说明：

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个tuple类型。
- kwargs - 可选参数。

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import thread
import time
# 为线程定义一个函数
def print_time( threadName, delay):
    count = 0
```

```
while count < 5:
    time.sleep(delay)
    count += 1
    print "%s: %s" % ( threadName, time.ctime(time.time()) )
# 创建两个线程
try:
    thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print "Error: unable to start thread"
while 1:
    pass
```

执行以上程序输出结果如下：

```
Thread-1: Thu Jan 22 15:42:17 2009
Thread-1: Thu Jan 22 15:42:19 2009
Thread-2: Thu Jan 22 15:42:19 2009
Thread-1: Thu Jan 22 15:42:21 2009
Thread-2: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:25 2009
Thread-2: Thu Jan 22 15:42:27 2009
Thread-2: Thu Jan 22 15:42:31 2009
Thread-2: Thu Jan 22 15:42:35 2009
```

线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用`thread.exit()`，他抛出`SystemExit exception`，达到退出线程的目的。

线程模块

Python通过两个标准库`thread`和`threading`提供对线程的支持。`thread`提供了低级别的、原始的线程以及一个简单的锁。

`threading` 模块提供的其他方法：

- `threading.currentThread()`: 返回当前的线程变量。
- `threading.enumerate()`: 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- `threading.activeCount()`: 返回正在运行的线程数量，与`len(threading.enumerate())`有相同的结果。

除了使用方法外，线程模块同样提供了`Thread`类来处理线程，`Thread`类提供了以下方法：

- `run()`: 用以表示线程活动的方法。
- `start()`: 启动线程活动。
- `join([time])`: 等待至线程中止。这阻塞调用线程直至线程的`join()` 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- `isAlive()`: 返回线程是否活动的。

- `getName()`: 返回线程名。
- `setName()`: 设置线程名。

使用Threading模块创建线程

使用Threading模块创建线程，直接从threading.Thread继承，然后重写__init__方法和run方法：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import threading
import time
exitFlag = 0
class myThread (threading.Thread): #继承父类threading.Thread
def __init__(self, threadID, name, counter):
threading.Thread.__init__(self)
self.threadID = threadID
self.name = name
self.counter = counter
def run(self): #把要执行的代码写到run函数里面 线程在创建后会直接运行run函数
print "Starting " + self.name
print_time(self.name, self.counter, 5)
print "Exiting " + self.name
def print_time(threadName, delay, counter):
while counter:
if exitFlag:
(threading.Thread).exit()
time.sleep(delay)
print "%s: %s" % (threadName, time.ctime(time.time()))
counter -= 1
# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# 开启线程
thread1.start()
thread2.start()
print "Exiting Main Thread"
```

以上程序执行结果如下；

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
```

```
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

线程同步

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，为了保证数据的正确性，需要对多个线程进行同步。

使用Thread对象的Lock和Rlock可以实现简单的线程同步，这两个对象都有acquire方法和release方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到acquire和release方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。

那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # 获得锁，成功获得锁定后返回True
        # 可选的timeout参数不填时将一直阻塞直到获得锁定
        # 否则超时后将返回False
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁
        threadLock.release()
    def print_time(threadName, delay, counter):
        while counter:
            time.sleep(delay)
            print "%s: %s" % (threadName, time.ctime(time.time()))
            counter -= 1
threadLock = threading.Lock()
```

```
threads = []
# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# 开启新线程
thread1.start()
thread2.start()
# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)
# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

线程优先级队列 (Queue)

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出）队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。

Queue模块中的常用方法：

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.get_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put_nowait(item) 相当Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import Queue
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
```

```
self.q = q
def run(self):
    print "Starting " + self.name
    process_data(self.name, self.q)
    print "Exiting " + self.name
def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
            time.sleep(1)
threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1
# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1
# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
    queueLock.release()
# 等待队列清空
while not workQueue.empty():
    pass
# 通知线程是时候退出
exitFlag = 1
# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

以上程序执行结果：

```
Starting Thread-1
Starting Thread-2
Starting Thread-3
Thread-1 processing One
Thread-2 processing Two
Thread-3 processing Three
Thread-1 processing Four
Thread-2 processing Five
```

```
Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exiting Main Thread
```

[← Python isdecimal\(\)方法](#)[Python XML 解析 →](#)**1 篇笔记****写笔记**

Python 线程同步

以下代码可以直观展示加锁和不加锁时，对数据修改情况。

加锁时

```
# -*- encoding:UTF-8 -*-
# author : shoushixiong
# date   : 2018/11/22
import threading
import time
list = [0,0,0,0,0,0,0,0,0,0,0,0]
class myThread(threading.Thread):
    def __init__(self,threadId,name,counter):
        threading.Thread.__init__(self)
        self.threadId = threadId
        self.name = name
        self.counter = counter
    def run(self):
        print "开始线程:",self.name
        # 获得锁，成功获得锁定后返回 True
        # 可选的timeout参数不填时将一直阻塞直到获得锁定
        # 否则超时后将返回 False
        threadLock.acquire()
        print_time(self.name,self.counter,list.__len__())
        # 释放锁
        threadLock.release()
    def __del__(self):
        print self.name,"线程结束！"
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        list[counter-1] += 1
        print "[%s] %s 修改第 %d 个值, 修改后值为:%d" % (time.ctime(time.time()),threadName
,counter,list[counter-1])
        counter -= 1
threadLock = threading.Lock()
threads = []
```

```
# 创建新线程
thread1 = myThread(1,"Thread-1",1)
thread2 = myThread(2,"Thread-2",2)
# 开启新线程
thread1.start()
thread2.start()
# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)
# 等待所有线程完成
for t in threads:
    t.join()
print "主进程结束！"
```

输出结果为：

```
开始线程： Thread-1
开始线程： Thread-2
[Thu Nov 22 16:04:13 2018] Thread-1 修改第 12 个值，修改后值为:1
[Thu Nov 22 16:04:14 2018] Thread-1 修改第 11 个值，修改后值为:1
[Thu Nov 22 16:04:15 2018] Thread-1 修改第 10 个值，修改后值为:1
[Thu Nov 22 16:04:16 2018] Thread-1 修改第 9 个值，修改后值为:1
[Thu Nov 22 16:04:17 2018] Thread-1 修改第 8 个值，修改后值为:1
[Thu Nov 22 16:04:18 2018] Thread-1 修改第 7 个值，修改后值为:1
[Thu Nov 22 16:04:19 2018] Thread-1 修改第 6 个值，修改后值为:1
[Thu Nov 22 16:04:20 2018] Thread-1 修改第 5 个值，修改后值为:1
[Thu Nov 22 16:04:21 2018] Thread-1 修改第 4 个值，修改后值为:1
[Thu Nov 22 16:04:22 2018] Thread-1 修改第 3 个值，修改后值为:1
[Thu Nov 22 16:04:23 2018] Thread-1 修改第 2 个值，修改后值为:1
[Thu Nov 22 16:04:24 2018] Thread-1 修改第 1 个值，修改后值为:1
[Thu Nov 22 16:04:26 2018] Thread-2 修改第 12 个值，修改后值为:2
[Thu Nov 22 16:04:28 2018] Thread-2 修改第 11 个值，修改后值为:2
[Thu Nov 22 16:04:30 2018] Thread-2 修改第 10 个值，修改后值为:2
[Thu Nov 22 16:04:32 2018] Thread-2 修改第 9 个值，修改后值为:2
[Thu Nov 22 16:04:34 2018] Thread-2 修改第 8 个值，修改后值为:2
[Thu Nov 22 16:04:36 2018] Thread-2 修改第 7 个值，修改后值为:2
[Thu Nov 22 16:04:38 2018] Thread-2 修改第 6 个值，修改后值为:2
[Thu Nov 22 16:04:40 2018] Thread-2 修改第 5 个值，修改后值为:2
[Thu Nov 22 16:04:42 2018] Thread-2 修改第 4 个值，修改后值为:2
[Thu Nov 22 16:04:44 2018] Thread-2 修改第 3 个值，修改后值为:2
[Thu Nov 22 16:04:46 2018] Thread-2 修改第 2 个值，修改后值为:2
[Thu Nov 22 16:04:48 2018] Thread-2 修改第 1 个值，修改后值为:2
主进程结束！
Thread-1 线程结束！
Thread-2 线程结束！
```

不加锁时

同样是上面实例的代码，注释以下两行代码：


```
threadLock.acquire()  
threadLock.release()
```

输出结果为：

```
开始线程： Thread-1  
开始线程： Thread-2  
[Thu Nov 22 16:09:20 2018] Thread-1 修改第 12 个值，修改后值为:1  
[Thu Nov 22 16:09:21 2018] Thread-2 修改第 12 个值，修改后值为:2  
[Thu Nov 22 16:09:21 2018] Thread-1 修改第 11 个值，修改后值为:1  
[Thu Nov 22 16:09:22 2018] Thread-1 修改第 10 个值，修改后值为:1  
[Thu Nov 22 16:09:23 2018] Thread-1 修改第 9 个值，修改后值为:1  
[Thu Nov 22 16:09:23 2018] Thread-2 修改第 11 个值，修改后值为:2  
[Thu Nov 22 16:09:24 2018] Thread-1 修改第 8 个值，修改后值为:1  
[Thu Nov 22 16:09:25 2018] Thread-2 修改第 10 个值，修改后值为:2  
[Thu Nov 22 16:09:25 2018] Thread-1 修改第 7 个值，修改后值为:1  
[Thu Nov 22 16:09:26 2018] Thread-1 修改第 6 个值，修改后值为:1  
[Thu Nov 22 16:09:27 2018] Thread-2 修改第 9 个值，修改后值为:2  
[Thu Nov 22 16:09:27 2018] Thread-1 修改第 5 个值，修改后值为:1  
[Thu Nov 22 16:09:28 2018] Thread-1 修改第 4 个值，修改后值为:1  
[Thu Nov 22 16:09:29 2018] Thread-2 修改第 8 个值，修改后值为:2  
[Thu Nov 22 16:09:29 2018] Thread-1 修改第 3 个值，修改后值为:1  
[Thu Nov 22 16:09:30 2018] Thread-1 修改第 2 个值，修改后值为:1  
[Thu Nov 22 16:09:31 2018] Thread-2 修改第 7 个值，修改后值为:2  
[Thu Nov 22 16:09:31 2018] Thread-1 修改第 1 个值，修改后值为:1  
[Thu Nov 22 16:09:33 2018] Thread-2 修改第 6 个值，修改后值为:2  
[Thu Nov 22 16:09:35 2018] Thread-2 修改第 5 个值，修改后值为:2  
[Thu Nov 22 16:09:37 2018] Thread-2 修改第 4 个值，修改后值为:2  
[Thu Nov 22 16:09:39 2018] Thread-2 修改第 3 个值，修改后值为:2  
[Thu Nov 22 16:09:41 2018] Thread-2 修改第 2 个值，修改后值为:2  
[Thu Nov 22 16:09:43 2018] Thread-2 修改第 1 个值，修改后值为:2  
主进程结束！  
Thread-1 线程结束！  
Thread-2 线程结束！
```

zhoushixiong 4个月前 [11-22]