Python3 SMTP发送邮件 Python3 XML 解析

Python3 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，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。每个独立的线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。

线程可以分为:

- **内核线程：**由操作系统内核创建和撤销。
- **用户线程：**不需要内核支持而在用户程序中实现的线程。

Python3 线程中常用的两个模块为：

- **_thread**
- **threading(推荐使用)**

thread 模块已被废弃。用户可以使用 threading 模块代替。所以，在 Python3 中不能再使用"thread" 模块。为了兼容性，Python3 将 thread 重命名为 "_thread"。

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：

```
#!/usr/bin/python3

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: 无法启动线程")

while 1:
   pass
```

执行以上程序输出结果如下：

```
Thread-1: Wed Apr  6 11:36:31 2016
Thread-1: Wed Apr  6 11:36:33 2016
Thread-2: Wed Apr  6 11:36:33 2016
Thread-1: Wed Apr  6 11:36:35 2016
Thread-1: Wed Apr  6 11:36:37 2016
Thread-2: Wed Apr  6 11:36:37 2016
Thread-1: Wed Apr  6 11:36:39 2016
Thread-2: Wed Apr  6 11:36:41 2016
Thread-2: Wed Apr  6 11:36:45 2016
Thread-2: Wed Apr  6 11:36:49 2016
```

执行以上程后可以按下 ctrl-c to 退出。

线程模块

Python3 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，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- 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

我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法：

```
#!/usr/bin/python3

import threading
import time

exitFlag = 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)
        print_time(self.name, self.counter, 5)
        print ("退出线程：" + self.name)

def print_time(threadName, delay, counter):
    while counter:
```

```
            if exitFlag:
                threadName.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)

    # 开启新线程
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    print ("退出主线程")
```

以上程序执行结果如下；

```
开始线程：Thread-1
开始线程：Thread-2
Thread-1: Wed Apr  6 11:46:46 2016
Thread-1: Wed Apr  6 11:46:47 2016
Thread-2: Wed Apr  6 11:46:47 2016
Thread-1: Wed Apr  6 11:46:48 2016
Thread-1: Wed Apr  6 11:46:49 2016
Thread-2: Wed Apr  6 11:46:49 2016
Thread-1: Wed Apr  6 11:46:50 2016
退出线程：Thread-1
Thread-2: Wed Apr  6 11:46:51 2016
Thread-2: Wed Apr  6 11:46:53 2016
Thread-2: Wed Apr  6 11:46:55 2016
退出线程：Thread-2
退出主线程
```

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：

```
#!/usr/bin/python3

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 ("开启线程：  " + self.name)
        # 获取锁，用于线程同步
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁，开启下一个线程
        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 ("退出主线程")
```

执行以上程序，输出结果为：

```
开启线程： Thread-1
开启线程： Thread-2
Thread-1: Wed Apr  6 11:52:57 2016
Thread-1: Wed Apr  6 11:52:58 2016
Thread-1: Wed Apr  6 11:52:59 2016
Thread-2: Wed Apr  6 11:53:01 2016
Thread-2: Wed Apr  6 11:53:03 2016
Thread-2: Wed Apr  6 11:53:05 2016
退出主线程
```

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:

```
#!/usr/bin/python3

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 ("开启线程: " + self.name)
        process_data(self.name, self.q)
        print ("退出线程: " + 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 ("退出主线程")
```

以上程序执行结果：

```
开启线程: Thread-1
开启线程: Thread-2
开启线程: Thread-3
Thread-3 processing One
Thread-1 processing Two
Thread-2 processing Three
Thread-3 processing Four
Thread-1 processing Five
退出线程: Thread-3
退出线程: Thread-2
退出线程: Thread-1
退出主线程
```

← Python3 SMTP发送邮件 Python3 XML 解析 →