# python多线程详解

目录

* [python多线程详解](https://www.cnblogs.com/luyuze95/p/11289143.html#python多线程详解)
  + [一、线程介绍](https://www.cnblogs.com/luyuze95/p/11289143.html#一、线程介绍)
    - [什么是线程](https://www.cnblogs.com/luyuze95/p/11289143.html#什么是线程)
    - [为什么要使用多线程](https://www.cnblogs.com/luyuze95/p/11289143.html#为什么要使用多线程)
  + [二、线程实现](https://www.cnblogs.com/luyuze95/p/11289143.html#二、线程实现)
    - [threading模块](https://www.cnblogs.com/luyuze95/p/11289143.html#threading模块)
    - [自定义线程](https://www.cnblogs.com/luyuze95/p/11289143.html#自定义线程)
    - [守护线程](https://www.cnblogs.com/luyuze95/p/11289143.html#守护线程)
    - [主线程等待子线程结束](https://www.cnblogs.com/luyuze95/p/11289143.html#主线程等待子线程结束)
    - [多线程共享全局变量](https://www.cnblogs.com/luyuze95/p/11289143.html#多线程共享全局变量)
    - [互斥锁](https://www.cnblogs.com/luyuze95/p/11289143.html#互斥锁)
    - [递归锁](https://www.cnblogs.com/luyuze95/p/11289143.html#递归锁)
    - [信号量（BoundedSemaphore类)](https://www.cnblogs.com/luyuze95/p/11289143.html#信号量（boundedsemaphore类)
    - [事件（Event类）](https://www.cnblogs.com/luyuze95/p/11289143.html#事件（event类）)
  + [三、GIL（Global Interpreter Lock）全局解释器锁](https://www.cnblogs.com/luyuze95/p/11289143.html#三、gil（global-interpreter-lock）全局解释器锁)

## 一、线程介绍

**什么是线程**

**线程（Thread）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，它被包涵在进程之中，是进程中的实际运作单位。线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。

**为什么要使用多线程**

**线程在程序中是独立的、并发的执行流。**与分隔的进程相比，进程中线程之间的隔离程度要小，它们共享内存、文件句柄和其他进程应有的状态。

因为线程的划分尺度小于进程，使得多线程程序的并发性高。进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程比进程具有更高的性能，这是由于同一个进程中的线程都有共性多个线程共享同一个进程的虚拟空间。线程共享的环境包括进程代码段、进程的公有数据等，利用这些共享的数据，线程之间很容易实现通信。

操作系统在创建进程时，必须为该进程分配独立的内存空间，并分配大量的相关资源，但创建线程则简单得多。因此，使用多线程来实现并发比使用多进程的性能要高得多。

总结起来，使用**多线程编程具有如下几个优点**：

1. 进程之间不能共享内存，但线程之间共享内存非常容易。
2. 操作系统在创建进程时，需要为该进程重新分配系统资源，但创建线程的代价则小得多。因此，使用多线程来实现多任务并发执行比使用多进程的效率高。
3. Python 语言内置了多线程功能支持，而不是单纯地作为底层操作系统的调度方式，从而简化了 Python 的多线程编程。

## 二、线程实现

**threading模块 \*\***

普通创建方式

import threading

import time

def run(n):

print("task", n)

time.sleep(1)

print('2s')

time.sleep(1)

print('1s')

time.sleep(1)

print('0s')

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

t1 = threading.Thread(target=run, args=("t1",))

t2 = threading.Thread(target=run, args=("t2",))

t1.start()

t2.start()

----------------------------------

>>> task t1

>>> task t2

>>> 2s

>>> 2s

>>> 1s

>>> 1s

>>> 0s

>>> 0s

**自定义线程 \*\***

继承threading.Thread来自定义线程类，其本质是重构Thread类中的run方法

import threading

import time

class MyThread(threading.Thread):

def \_\_init\_\_(self, n):

super(MyThread, self).\_\_init\_\_() # 重构run函数必须要写

self.n = n

def run(self):

print("task", self.n)

time.sleep(1)

print('2s')

time.sleep(1)

print('1s')

time.sleep(1)

print('0s')

time.sleep(1)

if \_\_name\_\_ == "\_\_main\_\_":

t1 = MyThread("t1")

t2 = MyThread("t2")

t1.start()

t2.start()

----------------------------------

>>> task t1

>>> task t2

>>> 2s

>>> 2s

>>> 1s

>>> 1s

>>> 0s

>>> 0s

**守护线程 \*\*\***

**我们看下面这个例子，这里使用setDaemon(True) [ˈdiːmən]把所有的子线程都变成了主线程的守护线程，因此当主进程结束后，子线程也会随之结束。所以当主线程结束后，整个程序就退出了。**

import threading

import time

def run(n):

print("task", n)

time.sleep(1) #此时子线程停1s

print('3')

time.sleep(1)

print('2')

time.sleep(1)

print('1')

if \_\_name\_\_ == '\_\_main\_\_':

t = threading.Thread(target=run, args=("t1",))

t.setDaemon(True) #把子进程设置为守护线程，必须在start()之前设置

t.start()

print("end")

----------------------------------

>>> task t1

>>> end

**我们可以发现，设置守护线程之后，当主线程结束时，子线程也将立即结束，不再执行。**

**主线程等待子线程结束 \*\*\***

为了让守护线程执行结束之后，主线程再结束，我们可以使用join方法，让主线程等待子线程执行。

import threading

import time

def run(n):

print("task", n)

time.sleep(1) #此时子线程停1s

print('3')

time.sleep(1)

print('2')

time.sleep(1)

print('1')

if \_\_name\_\_ == '\_\_main\_\_':

t = threading.Thread(target=run, args=("t1",))

t.setDaemon(True) #把子进程设置为守护线程，必须在start()之前设置

t.start()

t.join() # 设置主线程等待子线程结束

print("end")

----------------------------------

>>> task t1

>>> 3

>>> 2

>>> 1

>>> end

**多线程共享全局变量 \*\*\***

线程是进程的执行单元，进程是系统分配资源的最小单位，所以在同一个进程中的多线程是共享资源的。

import threading

import time

g\_num = 100

def work1():

global g\_num

for i in range(3):

g\_num += 1

print("in work1 g\_num is : %d" % g\_num)

def work2():

global g\_num

print("in work2 g\_num is : %d" % g\_num)

if \_\_name\_\_ == '\_\_main\_\_':

t1 = threading.Thread(target=work1)

t1.start()

time.sleep(1)

t2 = threading.Thread(target=work2)

t2.start()

----------------------------------

>>> in work1 g\_num is : 103

>>> in work2 g\_num is : 103

**互斥锁 \*\*\***

由于线程之间是进行随机调度，并且每个线程可能只执行n条执行之后，当多个线程同时修改同一条数据时可能会出现脏数据，所以，出现了线程锁，即同一时刻允许一个线程执行操作。线程锁用于锁定资源，你可以定义多个锁, 像下面的代码, 当你需要独占某一资源时，任何一个锁都可以锁这个资源，就好比你用不同的锁都可以把相同的一个门锁住是一个道理。

由于线程之间是进行随机调度，如果有多个线程同时操作一个对象，如果没有很好地保护该对象，会造成程序结果的不可预期，我们也称此为“线程不安全”。

为了方式上面情况的发生，就出现了互斥锁(Lock)

from threading import Thread,Lock

import os,time

def work():

global n

lock.acquire()

temp=n

time.sleep(0.1)

n=temp-1

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

lock=Lock()

n=100

l=[]

for i in range(100):

p=Thread(target=work)

l.append(p)

p.start()

for p in l:

p.join()

**递归锁**

RLcok类的用法和Lock类一模一样，但它支持嵌套，在多个锁没有释放的时候一般会使用RLcok类。

import threading

import time

def Func(lock):

global gl\_num

lock.acquire()

gl\_num += 1

time.sleep(1)

print(gl\_num)

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

gl\_num = 0

lock = threading.RLock()

for i in range(10):

t = threading.Thread(target=Func, args=(lock,))

t.start()

**信号量（BoundedSemaphore类)**

互斥锁同时只允许一个线程更改数据，而Semaphore是同时允许一定数量的线程更改数据 ，比如厕所有3个坑，那最多只允许3个人上厕所，后面的人只能等里面有人出来了才能再进去。

import threading

import time

def run(n, semaphore):

semaphore.acquire() #加锁

time.sleep(1)

print("run the thread:%s\n" % n)

semaphore.release() #释放

if \_\_name\_\_ == '\_\_main\_\_':

num = 0

semaphore = threading.BoundedSemaphore(5) # 最多允许5个线程同时运行

for i in range(22):

t = threading.Thread(target=run, args=("t-%s" % i, semaphore))

t.start()

while threading.active\_count() != 1:

pass # print threading.active\_count()

else:

print('-----all threads done-----')

**事件（Event类）**

python线程的事件用于主线程控制其他线程的执行，事件是一个简单的线程同步对象，其主要提供以下几个方法：

* clear 将flag设置为“False”
* set 将flag设置为“True”
* is\_set 判断是否设置了flag
* wait 会一直监听flag，如果没有检测到flag就一直处于阻塞状态

事件处理的机制：全局定义了一个“Flag”，当flag值为“False”，那么event.wait()就会阻塞，当flag值为“True”，那么event.wait()便不再阻塞。

#利用Event类模拟红绿灯

import threading

import time

event = threading.Event()

def lighter():

count = 0

event.set() #初始值为绿灯

while True:

if 5 < count <=10 :

event.clear() # 红灯，清除标志位

print("\33[41;1mred light is on...\033[0m")

elif count > 10:

event.set() # 绿灯，设置标志位

count = 0

else:

print("\33[42;1mgreen light is on...\033[0m")

time.sleep(1)

count += 1

def car(name):

while True:

if event.is\_set(): #判断是否设置了标志位

print("[%s] running..."%name)

time.sleep(1)

else:

print("[%s] sees red light,waiting..."%name)

event.wait()

print("[%s] green light is on,start going..."%name)

light = threading.Thread(target=lighter,)

light.start()

car = threading.Thread(target=car,args=("MINI",))

car.start()

## 三、GIL（Global Interpreter Lock）全局解释器锁

在非python环境中，单核情况下，同时只能有一个任务执行。多核时可以支持多个线程同时执行。但是在python中，无论有多少核，同时只能执行一个线程。究其原因，这就是由于GIL的存在导致的。

GIL的全称是Global Interpreter Lock(全局解释器锁)，来源是python设计之初的考虑，为了数据安全所做的决定。某个线程想要执行，必须先拿到GIL，我们可以把GIL看作是“通行证”，并且在一个python进程中，GIL只有一个。拿不到通行证的线程，就不允许进入CPU执行。GIL只在cpython中才有，因为cpython调用的是c语言的原生线程，所以他不能直接操作cpu，只能利用GIL保证同一时间只能有一个线程拿到数据。而在pypy和jpython中是没有GIL的。

Python多线程的工作过程：  
python在使用多线程的时候，调用的是c语言的原生线程。

* 拿到公共数据
* 申请gil
* python解释器调用os原生线程
* os操作cpu执行运算
* 当该线程执行时间到后，无论运算是否已经执行完，gil都被要求释放
* 进而由其他进程重复上面的过程
* 等其他进程执行完后，又会切换到之前的线程（从他记录的上下文继续执行），整个过程是每个线程执行自己的运算，当执行时间到就进行切换（context switch）。

**python针对不同类型的代码执行效率也是不同的：**

1、CPU密集型代码(各种循环处理、计算等等)，在这种情况下，由于计算工作多，ticks计数很快就会达到阈值，然后触发GIL的释放与再竞争（多个线程来回切换当然是需要消耗资源的），所以python下的多线程对CPU密集型代码并不友好。  
2、IO密集型代码(文件处理、网络爬虫等涉及文件读写的操作)，多线程能够有效提升效率(单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。所以python的多线程对IO密集型代码比较友好。

**使用建议？**

python下想要充分利用多核CPU，就用多进程。因为每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行，在python中，多进程的执行效率优于多线程(仅仅针对多核CPU而言)。

**GIL在python中的版本差异：**

1、在python2.x里，GIL的释放逻辑是当前线程遇见IO操作或者ticks计数达到100时进行释放。（ticks可以看作是python自身的一个计数器，专门做用于GIL，每次释放后归零，这个计数可以通过sys.setcheckinterval 来调整）。而每次释放GIL锁，线程进行锁竞争、切换线程，会消耗资源。并且由于GIL锁存在，python里一个进程永远只能同时执行一个线程(拿到GIL的线程才能执行)，这就是为什么在多核CPU上，python的多线程效率并不高。  
2、在python3.x中，GIL不使用ticks计数，改为使用计时器（执行时间达到阈值后，当前线程释放GIL），这样对CPU密集型程序更加友好，但依然没有解决GIL导致的同一时间只能执行一个线程的问题，所以效率依然不尽如人意。

# python多线程详解（超详细）

笨小孩哈哈 2020-03-30 09:13:40 63234 收藏 929

分类专栏： python爬虫 文章标签： python python多线程 python线程详解

版权

python中的多线程是一个非常重要的知识点，今天为大家对多线程进行详细的说明，代码中的注释有多线程的知识点还有测试用的实例。

码字不易，阅读或复制完了，点个赞！

import threading

from threading import Lock,Thread

import time,os

'''

python多线程详解

## 什么是线程？

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，它被包涵在进程之中，是进程中的实际运作单位。

线程自己不拥有系统资源，只拥有一点儿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。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行

'''

'''

## 为什么要使用多线程？

线程在程序中是独立的、并发的执行流。与分隔的进程相比，进程中线程之间的隔离程度要小，它们共享内存、文件句柄

和其他进程应有的状态。

因为线程的划分尺度小于进程，使得多线程程序的并发性高。进程在执行过程之中拥有独立的内存单元，而多个线程共享

内存，从而极大的提升了程序的运行效率。

线程比进程具有更高的性能，这是由于同一个进程中的线程都有共性，多个线程共享一个进程的虚拟空间。线程的共享环境

包括进程代码段、进程的共有数据等，利用这些共享的数据，线程之间很容易实现通信。

操作系统在创建进程时，必须为改进程分配独立的内存空间，并分配大量的相关资源，但创建线程则简单得多。因此，使用多线程

来实现并发比使用多进程的性能高得要多。

'''

'''

总结起来，使用多线程编程具有如下几个优点：

进程之间不能共享内存，但线程之间共享内存非常容易。

操作系统在创建进程时，需要为该进程重新分配系统资源，但创建线程的代价则小得多。因此使用多线程来实现多任务并发执行比使用多进程的效率高

python语言内置了多线程功能支持，而不是单纯地作为底层操作系统的调度方式，从而简化了python的多线程编程。

'''

'''

## 普通创建方式

'''

# def run(n):

# print('task',n)

# time.sleep(1)

# print('2s')

# time.sleep(1)

# print('1s')

# time.sleep(1)

# print('0s')

# time.sleep(1)

#

# if \_\_name\_\_ == '\_\_main\_\_':

# t1 = threading.Thread(target=run,args=('t1',)) # target是要执行的函数名（不是函数），args是函数对应的参数，以元组的形式存在

# t2 = threading.Thread(target=run,args=('t2',))

# t1.start()

# t2.start()

'''

## 线程方式

自定义线程：继承threading.Thread来定义线程类，其本质是重构Thread类中的run方法

'''

# class MyThread(threading.Thread):

# def \_\_init\_\_(self,n):

# super(MyThread,self).\_\_init\_\_() #重构run函数必须写

# self.n = n

#

# def run(self):

# print('task',self.n)

# time.sleep(1)

# print('2s')

# time.sleep(1)

# print('1s')

# time.sleep(1)

# print('0s')

# time.sleep(1)

#

# if \_\_name\_\_ == '\_\_main\_\_':

# t1 = MyThread('t1')

# t2 = MyThread('t2')

# t1.start()

# t2.start()

'''

## 守护线程

下面这个例子，这里使用**setDaemon(True)**把所有的子线程都变成了主线程的守护线程，

因此当主线程结束后，子线程也会随之结束，所以当主线程结束后，整个程序就退出了。

所谓’线程守护’，就是主线程不管该线程的执行情况，只要是其他子线程结束且主线程执行完毕，主线程都会关闭。也就是说:主线程不等待该守护线程的执行完再去关闭。

'''

# def run(n):

# print('task',n)

# time.sleep(1)

# print('3s')

# time.sleep(1)

# print('2s')

# time.sleep(1)

# print('1s')

#

# if \_\_name\_\_ == '\_\_main\_\_':

# t=threading.Thread(target=run,args=('t1',))

# t.setDaemon(True)

# t.start()

# print('end')

'''

通过执行结果可以看出，设置守护线程之后，当主线程结束时，子线程也将立即结束，不再执行

'''

'''

## 主线程等待子线程结束

为了让守护线程执行结束之后，主线程再结束，我们可以使用join方法，让主线程等待子线程执行

'''

# def run(n):

# print('task',n)

# time.sleep(2)

# print('5s')

# time.sleep(2)

# print('3s')

# time.sleep(2)

# print('1s')

# if \_\_name\_\_ == '\_\_main\_\_':

# t=threading.Thread(target=run,args=('t1',))

# t.setDaemon(True) #把子线程设置为守护线程，必须在start()之前设置

# t.start()

# t.join() #设置主线程等待子线程结束

# print('end')

'''

## 多线程共享全局变量

**线程是进程的执行单元，进程是系统分配资源的最小执行单位，所以在同一个进程中的多线程是共享资源的**

'''

# g\_num = 100

# def work1():

# global g\_num

# for i in range(3):

# g\_num+=1

# print('in work1 g\_num is : %d' % g\_num)

#

# def work2():

# global g\_num

# print('in work2 g\_num is : %d' % g\_num)

#

# if \_\_name\_\_ == '\_\_main\_\_':

# t1 = threading.Thread(target=work1)

# t1.start()

# time.sleep(1)

# t2=threading.Thread(target=work2)

# t2.start()

'''

由于线程之间是进行随机调度，并且每个线程可能只执行n条执行之后，当多个线程同时修改同一条数据时可能会出现脏数据，

所以出现了线程锁，即同一时刻允许一个线程执行操作。线程锁用于锁定资源，可以定义多个锁，像下面的代码，当需要独占

某一个资源时，任何一个锁都可以锁定这个资源，就好比你用不同的锁都可以把这个相同的门锁住一样。

由于线程之间是进行随机调度的，如果有多个线程同时操作一个对象，如果没有很好地保护该对象，会造成程序结果的不可预期，

**我们因此也称为“线程不安全”。**

为了防止上面情况的发生，就出现了互斥锁（Lock）

'''

# def work():

# global n

# lock.acquire()

# temp = n

# time.sleep(0.1)

# n = temp-1

# lock.release()

#

#

# if \_\_name\_\_ == '\_\_main\_\_':

# lock = Lock()

# n = 100

# l = []

# for i in range(100):

# p = Thread(target=work)

# l.append(p)

# p.start()

# for p in l:

# p.join()

'''

## 递归锁

递归锁：RLcok类的用法和Lock类一模一样，但它支持嵌套，在多个锁没有释放的时候一般会使用RLock类

'''

# def func(lock):

# global gl\_num

# lock.acquire()

# gl\_num += 1

# time.sleep(1)

# print(gl\_num)

# lock.release()

#

#

# if \_\_name\_\_ == '\_\_main\_\_':

# gl\_num = 0

# lock = threading.RLock()

# for i in range(10):

# t = threading.Thread(target=func,args=(lock,))

# t.start()

'''

## 信号量（BoundedSemaphore类）

互斥锁同时只允许一个线程更改数据，而Semaphore是同时允许一定数量的线程更改数据，比如厕所有3个坑，

那最多只允许3个人上厕所，后面的人只能等里面有人出来了才能再进去

'''

# def run(n,semaphore):

# semaphore.acquire() #加锁

# time.sleep(3)

# print('run the thread:%s\n' % n)

# semaphore.release() #释放

#

#

# if \_\_name\_\_== '\_\_main\_\_':

# num=0

# semaphore = threading.BoundedSemaphore(5) #最多允许5个线程同时运行

# for i in range(22):

# t = threading.Thread(target=run,args=('t-%s' % i,semaphore))

# t.start()

# while threading.active\_count() !=1:

# pass

# else:

# print('----------all threads done-----------')

'''

python线程的事件用于主线程控制其他线程的执行，事件是一个简单的线程同步对象，其主要提供以下的几个方法：

clear将flag设置为 False

set将flag设置为 True

s\_set判断是否设置了flag

wait会一直监听flag，如果没有检测到flag就一直处于阻塞状态

事件处理的机制：全局定义了一个Flag，当Flag的值为False，那么event.wait()就会阻塞，当flag值为True，

那么event.wait()便不再阻塞

'''

event = threading.Event()

def lighter():

count = 0

event.set() #初始者为绿灯

while True:

if 5 < count <=10:

event.clear() #红灯，清除标志位

print("\33[41;lmred light is on...\033[0m]")

elif count > 10:

event.set() #绿灯，设置标志位

count = 0

else:

print('\33[42;lmgreen light is on...\033[0m')

time.sleep(1)

count += 1

def car(name):

while True:

if event.is\_set(): #判断是否设置了标志位

print('[%s] running.....'%name)

time.sleep(1)

else:

print('[%s] sees red light,waiting...'%name)

event.wait()

print('[%s] green light is on,start going...'%name)

# startTime = time.time()

light = threading.Thread(target=lighter,)

light.start()

car = threading.Thread(target=car,args=('MINT',))

car.start()

endTime = time.time()

# print('用时：',endTime-startTime)

'''

## GIL 全局解释器

在非python环境中，单核情况下，同时只能有一个任务执行。多核时可以支持多个线程同时执行。但是在python中，无论有多少个核

同时只能执行一个线程。究其原因，这就是由于GIL的存在导致的。

GIL的全程是全局解释器，来源是python设计之初的考虑，为了数据安全所做的决定。某个线程想要执行，必须先拿到GIL，我们可以把GIL看做是“通行证”，并且在一个python进程之中，GIL只有一个。拿不到线程的通行证，并且在一个python进程中，GIL只有一个，拿不到通行证的线程，就不允许进入CPU执行。GIL只在cpython中才有，因为cpython调用的是c语言的原生线程，所以他不能直接操作cpu，而只能利用GIL保证同一时间只能有一个线程拿到数据。而在pypy和jpython中是没有GIL的python在使用多线程的时候，调用的是c语言的原生过程。

'''

'''

## python针对不同类型的代码执行效率也是不同的

1、CPU密集型代码（各种循环处理、计算等），在这种情况下，由于计算工作多，ticks技术很快就会达到阀值，然后出发GIL的释放与再竞争（多个线程来回切换当然是需要消耗资源的），所以python下的多线程对CPU密集型代码并不友好。

2、IO密集型代码（文件处理、网络爬虫等设计文件读写操作），多线程能够有效提升效率（单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序的执行效率）。所以python的多线程对IO密集型代码比较友好。

'''

'''

主要要看任务的类型，我们把任务分为I/O密集型和计算密集型，而多线程在切换中又分为I/O切换和时间切换。如果任务属于是I/O密集型，

若不采用多线程，我们在进行I/O操作时，势必要等待前面一个I/O任务完成后面的I/O任务才能进行，在这个等待的过程中，CPU处于等待

状态，这时如果采用多线程的话，刚好可以切换到进行另一个I/O任务。这样就刚好可以充分利用CPU避免CPU处于闲置状态，提高效率。但是

如果多线程任务都是计算型，CPU会一直在进行工作，直到一定的时间后采取多线程时间切换的方式进行切换线程，此时CPU一直处于工作状态，

此种情况下并不能提高性能，相反在切换多线程任务时，可能还会造成时间和资源的浪费，导致效能下降。这就是造成上面两种多线程结果不能的解释。

结论:I/O密集型任务，建议采取多线程，还可以采用多进程+协程的方式(例如:爬虫多采用多线程处理爬取的数据)；对于计算密集型任务，python此时就不适用了。

'''

————————————————

版权声明：本文为CSDN博主「笨小孩哈哈」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_40481076/article/details/101594705>

# python-实现多线程的三种方法

淋巴不想动 2019-02-09 16:47:31 3697 收藏 21

分类专栏： python 多线程

版权

## 1.\_thread.start\_new\_thread(了解)

import threading

import time

import \_thread

def job():

print("这是一个需要执行的任务。。。。。")

print("当前线程的个数:", threading.active\_count() )

print("当前线程的信息:", threading.current\_thread())

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

# 创建多线程时， 需要制定该线程执行的任务

\_thread.start\_new\_thread(job, ())

\_thread.start\_new\_thread(job, ())

job()

## 2.threading.Thread

import threading

import time

def job():

print("这是一个需要执行的任务。。。。。")

print("当前线程的个数:", threading.active\_count() )

time.sleep(1)

print("当前线程的信息:", threading.current\_thread())

if \_\_name\_\_ == '\_\_main\_\_':

# 创建多线程时， 需要制定该线程执行的任务.name线程名字 target目标函数名

t1 = threading.Thread(target=job,name='job1')

t2 = threading.Thread(target=job,name='job2')

t1.start()

t2.start()

print(threading.active\_count())

print("程序执行结束.....")

输出：

这是一个需要执行的任务。。。。。

当前线程的个数: 3

这是一个需要执行的任务。。。。。

3

程序执行结束.....

当前线程的个数: 3

当前线程的信息: <Thread(job1, started 140416648140544)>

当前线程的信息: <Thread(job2, started 140416639747840)>

出现的问题: 主线程执行结束， 但是子线程还在运行。

join()方法可以等待所有的子线程执行结束之后， 再执行主线程。

import threading

import time

def job():

print("这是一个需要执行的任务。。。。。")

print("当前线程的个数:", threading.active\_count() )

print("当前线程的信息:", threading.current\_thread())

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

# 创建多线程时， 需要制定该线程执行的任务.name线程名字 target目标函数名

t1 = threading.Thread(target=job,name='job1')

t2 = threading.Thread(target=job,name='job2')

t1.start()

t2.start()

print(threading.active\_count())

# 出现的问题: 主线程执行结束， 但是子线程还在运行。

# 等待所有的子线程执行结束之后， 再执行主线程

t1.join()

t2.join()

print("程序执行结束.....")

之前写过一个简单爬虫的实验，现在希望获取十个ip的城市和国家

-不使用多线程

import time

from urllib.request import urlopen

# 记录时间的装饰器

def timeit(f):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

res = f(\*args, \*\*kwargs)

end\_time = time.time()

print("%s函数运行时间:%.2f" % (f.\_\_name\_\_, end\_time - start\_time))

return res

return wrapper

def get\_addr(ip):

url = "http://ip-api.com/json/%s" % (ip)

urlObj = urlopen(url)

# 服务端返回的页面信息, 此处为字符串类型

pageContent = urlObj.read().decode('utf-8')

# 2. 处理Json数据

import json

# 解码: 将json数据格式解码为python可以识别的对象;

dict\_data = json.loads(pageContent)

print("""

%s

所在城市: %s

所在国家: %s

""" % (ip, dict\_data['city'], dict\_data['country']))

@timeit

def main():

ips = ['12.13.14.%s' % (i + 1) for i in range(10)]

for ip in ips:

get\_addr(ip)

if \_\_name\_\_ == '\_\_main\_\_':

main()

时间需要138.91秒。

-使用多线程

import threading

import time

from urllib.request import urlopen

def timeit(f):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

res = f(\*args, \*\*kwargs)

end\_time = time.time()

print("%s函数运行时间:%.2f" % (f.\_\_name\_\_, end\_time - start\_time))

return res

return wrapper

def get\_addr(ip):

url = "http://ip-api.com/json/%s" % (ip)

urlObj = urlopen(url)

# 服务端返回的页面信息, 此处为字符串类型

pageContent = urlObj.read().decode('utf-8')

# 2. 处理Json数据

import json

# 解码: 将json数据格式解码为python可以识别的对象;

dict\_data = json.loads(pageContent)

print("""

%s

所在城市: %s

所在国家: %s

""" % (ip, dict\_data['city'], dict\_data['country']))

@timeit

def main():

ips = ['12.13.14.%s' % (i + 1) for i in range(10)]

threads = []

for ip in ips:

# 实例化10个对象，target=目标函数名，args=目标函数参数(元组格式)

t = threading.Thread(target=get\_addr, args=(ip, ))

threads.append(t)

t.start()

# 等待所有子线程结束再运行主线程

[thread.join() for thread in threads]

if \_\_name\_\_ == '\_\_main\_\_':

main()

## 3.重写run方法

重写run方法， 实现多线程， 因为start方法执行时， 调用的是run方法;run方法里面编写的内容就是你要执行的任务；

import threading

import time

# 重写一个类，继承于threading.Thread

class MyThread(threading.Thread):

def \_\_init\_\_(self, jobName):

super(MyThread, self).\_\_init\_\_()

self.jobName = jobName

# 重写run方法， 实现多线程， 因为start方法执行时， 调用的是run方法;

# run方法里面编写的内容就是你要执行的任务；

def run(self):

print("这是一个需要执行的任务%s。。。。。" %(self.jobName))

print("当前线程的个数:", threading.active\_count() )

time.sleep(1)

print("当前线程的信息:", threading.current\_thread())

if \_\_name\_\_ == '\_\_main\_\_':

t1 = MyThread("name1")

t2 = MyThread("name2")

t1.start()

t2.start()

t1.join()

t2.join()

print("程序执行结束.....")

重写run方法实现刚才爬虫多线程案例

import threading

import time

from urllib.request import urlopen

def timeit(f):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

res = f(\*args, \*\*kwargs)

end\_time = time.time()

print("%s函数运行时间:%.2f" % (f.\_\_name\_\_, end\_time - start\_time))

return res

return wrapper

class MyThread(threading.Thread):

def \_\_init\_\_(self, ip):

super(MyThread, self).\_\_init\_\_()

self.ip = ip

def run(self):

url = "http://ip-api.com/json/%s" % (self.ip)

urlObj = urlopen(url)

# 服务端返回的页面信息, 此处为字符串类型

pageContent = urlObj.read().decode('utf-8')

# 2. 处理Json数据

import json

# 解码: 将json数据格式解码为python可以识别的对象;

dict\_data = json.loads(pageContent)

print("""

%s

所在城市: %s

所在国家: %s

""" % (self.ip, dict\_data['city'], dict\_data['country']))

@timeit

def main():

ips = ['12.13.14.%s' % (i + 1) for i in range(10)]

threads = []

for ip in ips:

# 实例化自己重写的类

t = MyThread(ip)

threads.append(t)

t.start()

[thread.join() for thread in threads]

if \_\_name\_\_ == '\_\_main\_\_':

main()

————————————————

版权声明：本文为CSDN博主「淋巴不想动」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/weixin\_43067754/article/details/86763905