what is fantasy

细微之处有神明

搞定python多线程和多进程

1 概念梳理：

1.1 线程

1.1.1 什么是线程

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。一个线程是一个execution context（执行上下文），即一个cpu执行时所需要的一串指令。

1.1.2 线程的工作方式

假设你正在读一本书，没有读完，你想休息一下，但是你想在回来时恢复到当时读的具体进度。有一个方法就是记下页数、行数与字数这三个数值，这些数值就是execution context。如果你的室友在你休息的时候，使用相同的方法读这本书。你和她只需要这三个数字记下来就可以在交替的时间共同阅读这本书了。

线程的工作方式与此类似。CPU会给你一个在同一时间能够做多个运算的幻觉，实际上它在每个运算上只花了极少的时间，本质上CPU同一时刻只干了一件事。它能这样做就是因为它有每个运算的execution context。就像你能够和你朋友共享同一本书一样，多任务也能共享同一块CPU。

1.2 进程

一个程序的执行实例就是一个**进程**。每一个进程提供执行程序所需的所有资源。（进程本质上是资源的集合）

一个进程有一个虚拟的地址空间、可执行的代码、操作系统的接口、安全的上下文（记录启动该进程的用户和权限等等）、唯一的进程ID、环境变量、优先级类、最小和最大的工作空间（内存空间），还要有至少一个线程。

每一个进程启动时都会最先产生一个线程，即主线程。然后主线程会再创建其他的子线程。

> 与进程相关的资源包括:

- 内存页（**同一个进程中的所有线程共享同一个内存空间**）
- 文件描述符(e.g. open sockets)
- 安全凭证（e.g.启动该进程的用户ID）

1.3 进程与线程区别

1.同一个进程中的线程共享同一内存空间，但是进程之间是独立的。
2.同一个进程中的所有线程的数据是共享的（进程通讯），进程之间的数据是独立的。
3.对主线程的修改可能会影响其他线程的行为，但是父进程的修改（除了删除以外）不会影响其他子进程。
4.线程是一个上下文的执行指令，而进程则是与运算相关的一簇资源。
5.同一个进程的线程之间可以直接通信，但是进程之间的交流需要借助中间代理来实现。
6.创建新的线程很容易，但是创建新的进程需要对父进程做一次复制。
7.一个线程可以操作同一进程的其他线程，但是进程只能操作其子进程。
8.线程启动速度快，进程启动速度慢（但是两者运行速度没有可比性）。

2 多线程

2.1 线程常用方法

‹ 2019年4月 ›

日	一	二	三	四	五	六
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11

搜索

随笔分类(46)

html/css(2)
JavaScript(7)
LeetCode(1)
Linux(2)
Python(29)
python学习(1)
Web(2)
方法论(2)

随笔档案(48)

2017年8月 (1)
2017年3月 (1)
2017年2月 (7)
2017年1月 (1)
2016年12月 (12)
2016年11月 (5)
2016年10月 (21)

最新评论

1. Re:Python基本数据类型之int
谢谢分享
--龙灰灰
2. Re:python下ssh的简单实现
1
--梓沂
3. Re:搞定python多线程和多进程

方法	注释
start()	线程准备就绪，等待CPU调度
setName()	为线程设置名称
getName()	获取线程名称
setDaemon(True)	设置为守护线程
join()	逐个执行每个线程，执行完毕后继续往下执行
run()	线程被cpu调度后自动执行线程对象的run方法，如果想自定义线程类，直接重写run方法就行了

2.1.1 Thread类

1.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)

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
"""
```

2.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()
```

2.1.2 计算子线程执行的时间

3.3 进程锁（进程同步）这里，正确的执行结果是什么呢？测试过加不加锁都是随机顺序执行，那么加锁的意义是什么呢？问题可能比小白，希望能解答一下

--Gin阿金

4. Re:搞定python多线程和多进程
好文

--普莱斯丶

5. Re:搞定python多线程和多进程
请问下这是2.7还是3的？打算在自己的博客下添加博主的链接以便以后学习

--F·灬灬疯子

阅读排行榜

1. Python基本数据类型之set(59160)
2. 搞定python多线程和多进程(31359)
3. python下ssh的简单实现(22299)
4. Python基本数据类型之int(17900)
5. python奇技淫巧——max/min函数的用法(17617)

评论排行榜

1. 搞定python多线程和多进程(9)
2. Python基本数据类型之set(6)
3. css常用属性汇总(3)
4. Python的方法解析顺序(MRO)[转](3)
5. python奇技淫巧——max/min函数的用法(3)

注：sleep的时候是不会占用cpu的,在sleep的时候操作系统会把线程暂时挂起。

```
join()  #等此线程执行完后，再执行其他线程或主线程
threading.current_thread()      #输出当前线程

import threading
import time

def run(n):
    print("task", n,threading.current_thread())    #输出当前的线程
    time.sleep(1)
    print('3s')
    time.sleep(1)
    print('2s')
    time.sleep(1)
    print('1s')

strat_time = time.time()

t_obj = []   #定义列表用于存放子线程实例

for i in range(3):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()
    t_obj.append(t)

"""
由主线程生成的三个子线程
task t-0 <Thread(Thread-1, started 44828)>
task t-1 <Thread(Thread-2, started 42804)>
task t-2 <Thread(Thread-3, started 41384)>
"""

for tmp in t_obj:
    t.join()            #为每个子线程添加join之后，主线程就会等这些子线程执行完之后再执行。

print("cost:", time.time() - strat_time) #主线程

print(threading.current_thread())       #输出当前线程
"""
<_MainThread(MainThread, started 43740)>
"""
```

2.1.3 统计当前活跃的线程数

由于主线程比子线程快很多，当主线程执行active_count()时，其他子线程都还没执行完毕，因此利用主线程统计的活跃的线程数num = sub_num(子线程数量)+1(主线程本身)

```
import threading
import time

def run(n):
    print("task", n)
    time.sleep(1)       #此时子线程停1s

for i in range(3):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()

time.sleep(0.5)     #主线程停0.5秒
print(threading.active_count()) #输出当前活跃的线程数

"""
task t-0
task t-1
task t-2
4
"""
```

由于主线程比子线程慢很多，当主线程执行active_count()时，其他子线程都已经执行完毕，因此利用主线程统计的活跃的线程数num = 1(主线程本身)

```
import threading
import time


def run(n):
    print("task", n)
    time.sleep(0.5)       #此时子线程停0.5s
```

```
for i in range(3):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()

time.sleep(1)     #主线程停1秒
print(threading.active_count()) #输出活跃的线程数
"""
task t-0
task t-1
task t-2
1
"""
```

此外我们还能发现在python内部默认会等待最后一个进程执行完后再执行exit()，或者说python内部在此时有一个隐藏的join()。

2.2 守护进程

我们看下面这个例子，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')

for i in range(3):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.setDaemon(True)   #把子进程设置为守护线程，必须在start()之前设置
    t.start()

time.sleep(0.5)     #主线程停0.5秒
print(threading.active_count()) #输出活跃的线程数
"""
task t-0
task t-1
task t-2
4

Process finished with exit code 0
"""
```

2.3 GIL

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。

1. 拿到公共数据
2. 申请gil
3. python解释器调用os原生线程
4. os操作cpu执行运算
5. 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
6. 进而由其他进程重复上面的过程
7. 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。

2.4 线程锁

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”。

```
#实测：在python2.7、mac os下，运行以下代码可能会产生脏数据。但是在python3中就不一定会出现下面的问题。

import threading
import time

def run(n):
    global num
    num += 1

num = 0
t_obj = []

for i in range(20000):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()
    t_obj.append(t)

for t in t_obj:
    t.join()

print "num:", num
"""
产生脏数据后的运行结果：
num: 19999
"""
```

2.5 互斥锁（mutex）

为了方式上面情况的发生，就出现了互斥锁(Lock)

```
import threading
import time


def run(n):
    lock.acquire()  #获取锁
    global num
    num += 1
```

```
    lock.release()  #释放锁

lock = threading.Lock()     #实例化一个锁对象

num = 0
t_obj = []

for i in range(20000):
    t = threading.Thread(target=run, args=("t-%s" % i,))
    t.start()
    t_obj.append(t)

for t in t_obj:
    t.join()

print "num:", num
```

2.6 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使用RLcok类。

```
import threading
import time

gl_num = 0

lock = threading.RLock()

def Func():
    lock.acquire()
    global gl_num
    gl_num +=1
    time.sleep(1)
    print gl_num
    lock.release()

for i in range(10):
    t = threading.Thread(target=Func)
    t.start()
```

2.7 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.acquire()   #加锁
    time.sleep(1)
    print("run the thread:%s\n" % n)
    semaphore.release()     #释放


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,))
    t.start()

while threading.active_count() != 1:
    pass  # print threading.active_count()
else:
    print('-----all threads done-----')
```

2.8 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：

方法	注释
clear	将flag设置为“False”

方法	注释
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()
```

2.9 条件（Condition类）

使得线程等待，只有满足某条件时，才释放n个线程

2.10 定时器（Timer类）

定时器，指定n秒后执行某操作

```
from threading import Timer


def hello():
    print("hello, world")

t = Timer(1, hello)
t.start()  # after 1 seconds, "hello, world" will be printed
```

3 多进程

在linux中，每个进程都是由父进程提供的。每启动一个子进程就从父进程克隆一份数据，但是进程之间的数据本身是不能共享的。

```
from multiprocessing import Process
import time
def f(name):
    time.sleep(2)
    print('hello', name)
```

```
if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()

from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())  #获取父进程id
    print('process id:', os.getpid())   #获取自己的进程id
    print("\n\n")

def f(name):
    info('\033[31;1mfunction f\033[0m')
    print('hello', name)

if __name__ == '__main__':
    info('\033[32;1mmain process line\033[0m')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

3.1 进程间通信

由于进程之间数据是不共享的，所以不会出现多线程GIL带来的问题。多进程之间的通信通过Queue()或Pipe()来实现

3.1.1 Queue()

使用方法跟threading里的queue差不多

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

3.1.2 Pipe()

Pipe的本质是进程之间的数据传递，而不是数据共享，这和socket有点像。pipe()返回两个连接对象分别表示管道的两端，每端都有send()和recv()方法。如果两个进程试图在同一时间的同一端进行读取和写入那么，这可能会损坏管道中的数据。

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

3.2 Manager

通过Manager可实现进程间数据的共享。Manager()返回的manager对象会通过一个服务进程，来使其他进程通过代理的方式操作python对象。manager对象支持 `list` , `dict` , `Namespace` , `Lock` , `RLock` , `Semaphore` , `BoundedSemaphore` , `Condition` , `Event` , `Barrier` , `Queue` , `Value` , `Array` .

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.append(1)
    print(l)
```

```
if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()

        l = manager.list(range(5))
        p_list = []
        for i in range(10):
            p = Process(target=f, args=(d, l))
            p.start()
            p_list.append(p)
        for res in p_list:
            res.join()

        print(d)
        print(l)
```

3.3 进程锁（进程同步）

数据输出的时候保证不同进程的输出内容在同一块屏幕正常显示，防止数据乱序的情况。
Without using the lock output from the different processes is liable to get all mixed up.

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

3.4 进程池

由于进程启动的开销比较大，使用多进程的时候会导致大量内存空间被消耗。为了防止这种情况发生可以使用进程池，（由于启动线程的开销比较小，所以不需要线程池这种概念，多线程只会频繁得切换cpu导致系统变慢，并不会占用过多的内存空间）

进程池中常用方法：

`apply()` 同步执行（串行）

`apply_async()` 异步执行（并行）

`terminate()` 立刻关闭进程池

`join()` 主进程等待所有子进程执行完毕。必须在close或terminate()之后。

`close()` 等待所有进程结束后，才关闭进程池。

```
from  multiprocessing import Process,Pool
import time

def Foo(i):
    time.sleep(2)
    return i+100

def Bar(arg):
    print('-->exec done:',arg)

pool = Pool(5)  #允许进程池同时放入5个进程

for i in range(10):
    pool.apply_async(func=Foo, args=(i,),callback=Bar)  #func子进程执行完后，才会执行callback，否则call
    #pool.apply(func=Foo, args=(i,))

print('end')
pool.close()
pool.join() #主进程等待所有子进程执行完毕。必须在close()或terminate()之后。
```

进程池内部维护一个进程序列，当使用时，去进程池中获取一个进程，如果进程池序列中没有可供使用的进程，那么程序就会等待，直到进程池中有可用进程为止。在上面的程序中产生了10个进程，但是只能有5同时被放入进程池，剩下的都被暂时挂起，并不占用内存空间，等前面的五个进程执行完后，再执行剩下5个进程。

4 补充：协程

线程和进程的操作是由程序触发系统接口，最后的执行者是系统，它本质上是操作系统提供的功能。而协程的操作则是程序员指定的，在python中通过yield，人为的实现并发处理。

协程存在的意义：对于多线程应用，CPU通过切片的方式来切换线程间的执行，线程切换时需要耗时。协程，则只使用一个线程，分解一个线程成为多个“微线程”，在一个线程中规定某个代码块的执行顺序。

协程的适用场景：当程序中存在大量不需要CPU的操作时（IO）。
常用第三方模块gevent和greenlet。（本质上，gevent是对greenlet的高级封装，因此一般用它就行，这是一个相当高效的模块。）

4.1 greenlet

```
from greenlet import greenlet

def test1():
    print(12)
    gr2.switch()
    print(34)
    gr2.switch()

def test2():
    print(56)
    gr1.switch()
    print(78)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

实际上，greenlet就是通过switch方法在不同的任务之间进行切换。

4.2 gevent

```
from gevent import monkey; monkey.patch_all()
import gevent
import requests

def f(url):
    print('GET: %s' % url)
    resp = requests.get(url)
    data = resp.text
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://github.com/'),
])
```

通过joinall将任务f和它的参数进行统一调度，实现单线程中的协程。代码封装层次很高，实际使用只需要了解它的几个主要方法即可。

posted @ 2017-02-24 22:30 morra 阅读(31360) 评论(9) 编辑 收藏

评论列表

#1楼 2017-06-02 19:49 **kingsea0_0**

博主我想把这篇转到自己的博客可以吗，主要想以后自己看的时候方便。会注明出处

支持(0) 反对(0)

#2楼[楼主] 2017-06-14 19:32 **morra**

@ kingsea0_0
欢迎转载 :)

支持(0) 反对(0)

#3楼 2017-07-17 01:07 **defined**