Yuan先生

随笔 - 1 文章 - 141 评论 - 143

Py西游攻关之多线程(threading模块)

线程与进程

什么是线程(thread)?

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

A thread is an execution context, which is all the information a CPU needs to execute a stream of instructions.

Suppose you're reading a book, and you want to take a break right now, but you want to be able to come back and resume reading from the exact point where you stopped. One way to achieve that is by jotting down the page number, line number, and word number. So your execution context for reading a book is these 3 numbers.

If you have a roommate, and she's using the same technique, she can take the book while you're not using it, and resume reading from where she stopped. Then you can take it back, and resume it from where you were.

Threads work in the same way. A CPU is giving you the illusion that it's doing multiple computations at the same time. It does that by spending a bit of time on each computation. It can do that because it has an execution context for each computation. Just like you can share a book with your friend, many tasks can share a CPU.

On a more technical level, an execution context (therefore a thread) consists of the values of the CPU's registers.

Last: threads are different from processes. A thread is a context of execution, while a process is a bunch of resources associated with a computation. A process can have one or many threads.

Clarification: the resources associated with a process include memory pages (all the threads in a process have the same view of the memory), file descriptors (e.g., open sockets), and security credentials (e.g., the ID of the user who started the process).

什么是进程(process)?

An executing instance of a program is called a process.

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

进程与线程的区别?

1. Threads share the address space of the process that created it; processes have their own address space.
2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
4. New threads are easily created; new processes require duplication of the parent process.
5. Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.

6. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.

Python GIL(Global Interpreter Lock)

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

threading模块

一 线程的2种调用方式

直接调用

实例1:

```
import threading
import time

def sayhi(num): #定义每个线程要运行的函数

    print("running on number:%s" %num)

    time.sleep(3)

if __name__ == '__main__':

    t1 = threading.Thread(target=sayhi,args=(1,)) #生成一个线程实例
    t2 = threading.Thread(target=sayhi,args=(2,)) #生成另一个线程实例

    t1.start() #启动线程
    t2.start() #启动另一个线程

    print(t1.getName()) #获取线程名
    print(t2.getName())
```

继承式调用:

```
import threading
import time


class MyThread(threading.Thread):
    def __init__(self,num):
        threading.Thread.__init__(self)
        self.num = num

    def run(self):#定义每个线程要运行的函数

        print("running on number:%s" %self.num)

        time.sleep(3)

if __name__ == '__main__':

    t1 = MyThread(1)
    t2 = MyThread(2)
    t1.start()
    t2.start()
```

二 Join & Daemon

```
import threading
from time import ctime,sleep
import time

def music(func):
    for i in range(2):
        print ("Begin listening to %s. %s" %(func,ctime()))
        sleep(4)
        print("end listening %s"%ctime())

def move(func):
    for i in range(2):
        print ("Begin watching at the %s! %s" %(func,ctime()))
        sleep(5)
        print('end watching %s'%ctime())

threads = []
t1 = threading.Thread(target=music,args=('七里香',))
threads.append(t1)
t2 = threading.Thread(target=move,args=('阿甘正传',))
threads.append(t2)

if __name__ == '__main__':

    for t in threads:
        # t.setDaemon(True)
        t.start()
        # t.join()
    # t1.join()
    t2.join()########考虑这三种join位置下的结果?
    print ("all over %s" %ctime())
```

setDaemon(True):

将线程声明为守护线程，必须在start() 方法调用之前设置， 如果不设置为守护线程程序会被无限挂起。这个方法基本和join是相反的。当我们 在程序运行中，执行一个主线程，如果主线程又创建一个子线程，主线程和子线程就分兵两路，分别运行，那么当主线程完成想退出时，会检验子线程是否完成。如 果子线程未完成，则主线程会等待子线程完成后再退出。但是有时候我们需要的是 只要主线程完成了，不管子线程是否完成，都要和主线程一起退出，这时就可以 用setDaemon方法啦

join():

在子线程完成运行之前，这个子线程的父线程将一直被阻塞。

其它方法

```
thread 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
# 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
# isAlive(): 返回线程是否活动的。
# getName(): 返回线程名。
# setName(): 设置线程名。
```

三 同步锁(Lock)

```
import time
import threading

def addNum():
```

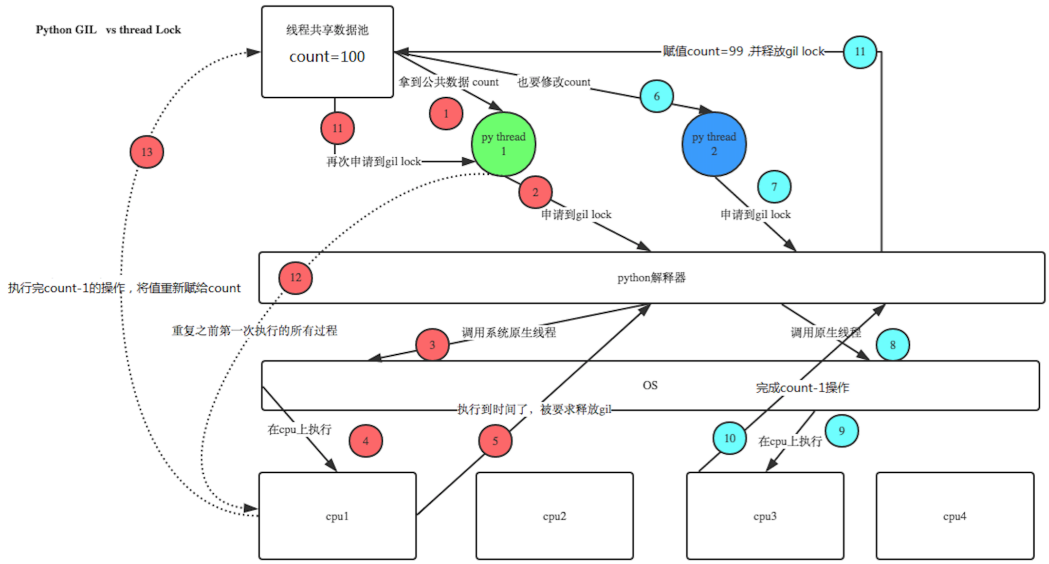
```
    global num #在每个线程中都获取这个全局变量
    # num-=1

    temp=num
    print('--get num:',num )
    #time.sleep(0.1)
    num =temp-1 #对此公共变量进行-1操作


num = 100  #设定一个共享变量
thread_list = []
for i in range(100):
    t = threading.Thread(target=addNum)
    t.start()
    thread_list.append(t)

for t in thread_list: #等待所有线程执行完毕
    t.join()

print('final num:', num )
```



注意：

1：why num-=1没问题呢？这是因为动作太快(完成这个动作在切换的时间内)

2： if sleep(1),现象会更明显，100个线程每一个一定都没有执行完就进行了切换，我们说过sleep就等效于IO阻塞，1s之内不会再切换回来，所以最后的结果一定是99.

多个线程都在同时操作同一个共享资源，所以造成了资源破坏，怎么办呢？

有同学会想用join呗，但join会把整个线程给停住，造成了串行，失去了多线程的意义，而我们只需要把计算(涉及到操作公共数据)的时候串行执行。

我们可以通过同步锁来解决这种问题

```
import time
import threading

def addNum():
    global num #在每个线程中都获取这个全局变量
    # num-=1
    lock.acquire()
    temp=num
    print('--get num:',num )
```

```
    #time.sleep(0.1)
    num =temp-1 #对此公共变量进行-1操作
    lock.release()

num = 100  #设定一个共享变量
thread_list = []
lock=threading.Lock()

for i in range(100):
    t = threading.Thread(target=addNum)
    t.start()
    thread_list.append(t)

for t in thread_list: #等待所有线程执行完毕
    t.join()

print('final num:', num )
```

问题解决，但

请问：同步锁与GIL的关系?

> Python的线程在GIL的控制之下，线程之间，对整个python解释器，对python提供的C API的访问都是互斥的，这可以看作是Python内核级的互斥机制。但是这种互斥是我们不能控制的，我们还需要另外一种可控的互斥机制————用户级互斥。内核级通过互斥保护了内核的共享资源，同样，用户级互斥保护了用户程序中的共享资源。
>
> GIL 的作用是：对于一个解释器，只能有一个thread在执行bytecode。所以每时每刻只有一条bytecode在被执行一个thread。GIL保证了bytecode 这层面上是thread safe的。
> 但是如果你有个操作比如 x += 1，这个操作需要多个bytecodes操作，在执行这个操作的多条bytecodes期间的时候可能中途就换thread了，这样就出现了data races的情况了。
>
> 那我的同步锁也是保证同一时刻只有一个线程被执行，是不是没有GIL也可以？是的；那要GIL有什么鸟用？你没治；

四 线程死锁和递归锁

在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁，因为系统判断这部分资源都正在使用，所有这两个线程在无外力作用下将一直等待下去。下面是一个死锁的例子：

```
import threading,time

class myThread(threading.Thread):
    def doA(self):
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
        time.sleep(3)
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        lockB.release()
        lockA.release()

    def doB(self):
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        time.sleep(2)
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
        lockA.release()
        lockB.release()
    def run(self):
        self.doA()
        self.doB()
if __name__=="__main__":

    lockA=threading.Lock()
    lockB=threading.Lock()
    threads=[]
    for i in range(5):
        threads.append(myThread())
    for t in threads:
```

```
        t.start()
    for t in threads:
        t.join()#等待线程结束，后面再讲。
```

解决办法：使用递归锁，将

```
lockA=threading.Lock()
lockB=threading.Lock()<br>#--------------<br>lock=threading.RLock()
```

为了支持在同一线程中多次请求同一资源，python提供了“可重入锁”：threading.RLock。RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次acquire。直到一个线程所有的acquire都被release，其他的线程才能获得资源。

应用

```
import time

import threading

class Account:
    def __init__(self, _id, balance):
        self.id = _id
        self.balance = balance
        self.lock = threading.RLock()

    def withdraw(self, amount):

        with self.lock:
            self.balance -= amount

    def deposit(self, amount):
        with self.lock:
            self.balance += amount


    def drawcash(self, amount):#lock.acquire中嵌套lock.acquire的场景

        with self.lock:
            interest=0.05
            count=amount+amount*interest

            self.withdraw(count)


def transfer(_from, to, amount):

    #锁不可以加在这里 因为其他的其它线程执行的其它方法在不加锁的情况下数据同样是不安全的
     _from.withdraw(amount)

     to.deposit(amount)



alex = Account('alex',1000)
yuan = Account('yuan',1000)

t1=threading.Thread(target = transfer, args = (alex,yuan, 100))
t1.start()

t2=threading.Thread(target = transfer, args = (yuan,alex, 200))
t2.start()

t1.join()
t2.join()

print('>>>',alex.balance)
print('>>>',yuan.balance)
```

五 条件变量同步(Condition)

有一类线程需要满足条件之后才能够继续执行，Python提供了threading.Condition 对象用于条件变量线程的支持，它除了能提供RLock()或Lock()的方法外，还提供了 wait()、notify()、notifyAll()方法。

lock_con=threading.Condition([Lock/Rlock])： 锁是可选选项，不传人锁，对象自动创建一个RLock()。

```
wait()：条件不满足时调用，线程会释放锁并进入等待阻塞；
notify()：条件创造后调用，通知等待池激活一个线程；
notifyAll()：条件创造后调用，通知等待池激活所有线程。
```

实例

```
import threading,time
from random import randint
class Producer(threading.Thread):
    def run(self):
        global L
        while True:
            val=randint(0,100)
            print('生产者',self.name,":Append"+str(val),L)
            if lock_con.acquire():
                L.append(val)
                lock_con.notify()
                lock_con.release()
            time.sleep(3)
class Consumer(threading.Thread):
    def run(self):
        global L
        while True:
                lock_con.acquire()
                if len(L)==0:
                    lock_con.wait()
                print('消费者',self.name,":Delete"+str(L[0]),L)
                del L[0]
                lock_con.release()
                time.sleep(0.25)

if __name__=="__main__":

    L=[]
    lock_con=threading.Condition()
    threads=[]
    for i in range(5):
        threads.append(Producer())
    threads.append(Consumer())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

六 同步条件(Event)

条件同步和条件变量同步差不多意思，只是少了锁功能，因为条件同步设计于不访问共享资源的条件环境。event=threading.Event()：条件环境对象，初始值 为False；

```
event.isSet()：返回event的状态值；

event.wait()：如果 event.isSet()==False将阻塞线程；

event.set()： 设置event的状态值为True，所有阻塞池的线程激活进入就绪状态， 等待操作系统调度；

event.clear()：恢复event的状态值为False。
```

实例1：

```
import threading,time
class Boss(threading.Thread):
    def run(self):
        print("BOSS: 今晚大家都要加班到22:00。")
        event.isSet() or event.set()
        time.sleep(5)
        print("BOSS: <22:00>可以下班了。")
        event.isSet() or event.set()
class Worker(threading.Thread):
    def run(self):
        event.wait()
        print("Worker: 哎……命苦啊! ")
        time.sleep(0.25)
        event.clear()
        event.wait()
        print("Worker: OhYeah!")
if __name__=="__main__":
    event=threading.Event()
    threads=[]
    for i in range(5):
        threads.append(Worker())
    threads.append(Boss())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

实例2:

```
import threading,time
import random
def light():
    if not event.isSet():
        event.set() #wait就不阻塞 #绿灯状态
    count = 0
    while True:
        if count < 10:
            print('\033[42;1m--green light on---\033[0m')
        elif count <13:
            print('\033[43;1m--yellow light on---\033[0m')
        elif count <20:
            if event.isSet():
                event.clear()
            print('\033[41;1m--red light on---\033[0m')
        else:
            count = 0
            event.set() #打开绿灯
        time.sleep(1)
        count +=1
def car(n):
    while 1:
        time.sleep(random.randrange(10))
        if  event.isSet(): #绿灯
            print("car [%s] is running.." % n)
        else:
            print("car [%s] is waiting for the red light.." %n)
if __name__ == '__main__':
    event = threading.Event()
    Light = threading.Thread(target=light)
    Light.start()
    for i in range(3):
        t = threading.Thread(target=car,args=(i,))
        t.start()
```

七 信号量(Semaphore)

信号量用来控制线程并发数的，BoundedSemaphore或Semaphore管理一个内置的计数 器，每当调用acquire()时-1，调用release()时+1。

计数器不能小于0，当计数器为 0时，acquire()将阻塞线程至同步锁定状态，直到其他线程调用release()。(类似于停车位的概念)

BoundedSemaphore与Semaphore的唯一区别在于前者将在调用release()时检查计数 器的值是否超过了计数器的初始值，如果超过了将抛出一个异常。

实例:

```
import threading,time
class myThread(threading.Thread):
    def run(self):
        if semaphore.acquire():
            print(self.name)
            time.sleep(5)
            semaphore.release()
if __name__=="__main__":
    semaphore=threading.Semaphore(5)
    thrs=[]
    for i in range(100):
        thrs.append(myThread())
    for t in thrs:
        t.start()
```

八 多线程利器(queue)

queue is especially useful in threaded programming when information must be exchanged safely between multiple threads.

queue列队类的方法

创建一个“队列”对象
import Queue
q = Queue.Queue(maxsize = 10)
Queue.Queue类即是一个队列的同步实现。队列长度可为无限或者有限。可通过Queue的构造函数的可选参数maxsize来设定队列长度。如果maxsize小于1就表示队列长度无限。

将一个值放入队列中
q.put(10)
调用队列对象的put()方法在队尾插入一个项目。put()有两个参数，第一个item为必需的，为插入项目的值；第二个block为可选参数，默认为
1。如果队列当前为空且block为1，put()方法就使调用线程暂停,直到空出一个数据单元。如果block为0，put方法将引发Full异常。

将一个值从队列中取出
q.get()
调用队列对象的get()方法从队头删除并返回一个项目。可选参数为block，默认为True。如果队列为空且block为True，get()就使调用线程暂停，直至有项目可用。如果队列为空且block为False，队列将引发Empty异常。

Python Queue模块有三种队列及构造函数:
1、Python Queue模块的FIFO队列先进先出。 class queue.Queue(maxsize)
2、LIFO类似于堆，即先进后出。 class queue.LifoQueue(maxsize)
3、还有一种是优先级队列级别越低越先出来。 class queue.PriorityQueue(maxsize)

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

实例

实例1:

```
import threading,queue
from time import sleep
from random import randint
class Production(threading.Thread):
    def run(self):
        while True:
            r=randint(0,100)
            q.put(r)
            print("生产出来%s号包子"%r)
            sleep(1)
class Proces(threading.Thread):
    def run(self):
        while True:
            re=q.get()
            print("吃掉%s号包子"%re)
if __name__=="__main__":
    q=queue.Queue(10)
    threads=[Production(),Production(),Production(),Proces()]
    for t in threads:
        t.start()
```

实例2:

```
import time,random
import queue,threading
q = queue.Queue()
def Producer(name):
  count = 0
  while count <20:
    time.sleep(random.randrange(3))
    q.put(count)
    print('Producer %s has produced %s baozi..' %(name, count))
    count +=1
def Consumer(name):
  count = 0
  while count <20:
    time.sleep(random.randrange(4))
    if not q.empty():
        data = q.get()
        print(data)
        print('\033[32;1mConsumer %s has eat %s baozi...\033[0m' %(name, data))
    else:
        print("-----no baozi anymore----")
    count +=1
p1 = threading.Thread(target=Producer, args=('A',))
c1 = threading.Thread(target=Consumer, args=('B',))
p1.start()
c1.start()
```

实例3:

```
#实现一个线程不断生成一个随机数到一个队列中(考虑使用Queue这个模块)
# 实现一个线程从上面的队列里面不断的取出奇数
# 实现另外一个线程从上面的队列里面不断取出偶数

import random,threading,time
from queue import Queue
#Producer thread
class Producer(threading.Thread):
  def __init__(self, t_name, queue):
    threading.Thread.__init__(self,name=t_name)
    self.data=queue
  def run(self):
    for i in range(10):  #随机产生10个数字 , 可以修改为任意大小
      randomnum=random.randint(1,99)
```

```
      print ("%s: %s is producing %d to the queue!" % (time.ctime(), self.getName(), randomnum))
      self.data.put(randomnum) #将数据依次存入队列
      time.sleep(1)
    print ("%s: %s finished!" %(time.ctime(), self.getName()))

#Consumer thread
class Consumer_even(threading.Thread):
  def __init__(self,t_name,queue):
    threading.Thread.__init__(self,name=t_name)
    self.data=queue
  def run(self):
    while 1:
      try:
        val_even = self.data.get(1,5) #get(self, block=True, timeout=None) ,1就是阻塞等待,5是超时5秒
        if val_even%2==0:
          print ("%s: %s is consuming. %d in the queue is consumed!" % (time.ctime(),self.getName
          time.sleep(2)
        else:
          self.data.put(val_even)
          time.sleep(2)
      except:   #等待输入，超过5秒 就报异常
        print ("%s: %s finished!" %(time.ctime(),self.getName()))
        break
class Consumer_odd(threading.Thread):
  def __init__(self,t_name,queue):
    threading.Thread.__init__(self, name=t_name)
    self.data=queue
  def run(self):
    while 1:
      try:
        val_odd = self.data.get(1,5)
        if val_odd%2!=0:
          print ("%s: %s is consuming. %d in the queue is consumed!" % (time.ctime(), self.getNam
          time.sleep(2)
        else:
          self.data.put(val_odd)
          time.sleep(2)
      except:
        print ("%s: %s finished!" % (time.ctime(), self.getName()))
        break
#Main thread
def main():
  queue = Queue()
  producer = Producer('Pro.', queue)
  consumer_even = Consumer_even('Con_even.', queue)
  consumer_odd = Consumer_odd('Con_odd.',queue)
  producer.start()
  consumer_even.start()
  consumer_odd.start()
  producer.join()
  consumer_even.join()
  consumer_odd.join()
  print ('All threads terminate!')

if __name__ == '__main__':
  main()
```

注意：列表是线程不安全的

```
import threading,time

li=[1,2,3,4,5]

def pri():
    while li:
        a=li[-1]
        print(a)
        time.sleep(1)
        try:
            li.remove(a)
        except:
```

```
            print('----',a)

t1=threading.Thread(target=pri,args=())
t1.start()
t2=threading.Thread(target=pri,args=())
t2.start()
```

九 Python中的上下文管理器(contextlib模块)

上下文管理器的任务是：代码块执行前准备，代码块执行后收拾

1、如何使用上下文管理器：

如何打开一个文件，并写入"hello world"

```
filename="my.txt"
mode="w"
f=open(filename,mode)
f.write("hello world")
f.close()
```

当发生异常时（如磁盘写满），就没有机会执行第5行。当然，我们可以采用try-finally语句块进行包装：

```
writer=open(filename,mode)
try:
    writer.write("hello world")
finally:
    writer.close()
```

当我们进行复杂的操作时，try-finally语句就会变得丑陋，采用with语句重写：

```
with open(filename,mode) as writer:
    writer.write("hello world")
```

as指代了从open()函数返回的内容，并把它赋给了新值。with完成了try-finally的任务。

2、自定义上下文管理器

with语句的作用类似于try-finally，提供一种上下文机制。要应用with语句的类，其内部必须提供两个内置函数__enter__和__exit__。前者在主体代码执行前执行，后者在主体代码执行后执行。as后面的变量，是在__enter__函数中返回的。

```
class echo():
    def output(self):
        print "hello world"
    def __enter__(self):
        print "enter"
        return self  #可以返回任何希望返回的东西
    def __exit__(self,exception_type,value,trackback):
        print "exit"
        if exception_type==ValueError:
            return True
        else:
            return Flase

>>>with echo as e:
    e.output()

输出:
enter
hello world
exit
```

完备的__exit__函数如下：

```
def __exit__(self,exc_type,exc_value,exc_tb)
```

其中，exc_type:异常类型；exc_value:异常值；exc_tb:异常追踪信息

当__exit__返回True时，异常不传播

3、contextlib模块

contextlib模块的作用是提供更易用的上下文管理器，它是通过Generator实现的。contextlib中的contextmanager作为装饰器来提供一种针对函数级别的上下文管理机制，常用框架如下：

```
from contextlib import contextmanager
@contextmanager
def make_context():
    print 'enter'
    try:
        yield "ok"
    except RuntimeError,err:
        print 'error',err
    finally:
        print 'exit'

>>>with make_context() as value:
    print value

输出为:
    enter
    ok
    exit
```

其中，yield写入try-finally中是为了保证异常安全（能处理异常）as后的变量的值是由yield返回。yield前面的语句可看作代码块执行前操作，yield之后的操作可以看作在__exit__函数中的操作。

以线程锁为例：

```
@contextlib.contextmanager
def loudLock():
    print 'Locking'
    lock.acquire()
    yield
    print 'Releasing'
    lock.release()

with loudLock():
    print 'Lock is locked: %s' % lock.locked()
    print 'Doing something that needs locking'

#Output:
#Locking
#Lock is locked: True
#Doing something that needs locking
#Releasing
```

4、contextlib.nested:减少嵌套

对于：

```
with open(filename,mode) as reader:
    with open(filename1,mode1) as writer:
        writer.write(reader.read())
```

可以通过contextlib.nested进行简化：

```
with contextlib.nested(open(filename,mode),open(filename1,mode1)) as (reader,writer):
    writer.write(reader.read())
```

在python 2.7及以后，被一种新的语法取代：

```
with open(filename,mode) as reader,open(filename1,mode1) as writer:
    writer.write(reader.read())
```

5、contextlib.closing()

file类直接支持上下文管理器API，但有些表示打开句柄的对象并不支持，如urllib.urlopen()返回的对象。还有些遗留类，使用close()方法而不支持上下文管理器API。为了确保关闭句柄，需要使用closing()为它创建一个上下文管理器（调用类的close方法）。

```
import contextlib
class myclass():
    def __init__(self):
        print '__init__'
    def close(self):
        print 'close()'

with contextlib.closing(myclass()):
    print 'ok'
```

输出:

```
__init__
ok
close()
```

自定义线程池

简单版本:

```
import queue
import threading
import time

class ThreadPool(object):

    def __init__(self, max_num=20):
        self.queue = queue.Queue(max_num)
        for i in range(max_num):
            self.queue.put(threading.Thread)

    def get_thread(self):
        return self.queue.get()

    def add_thread(self):
        self.queue.put(threading.Thread)


'''
pool = ThreadPool(10)

def func(arg, p):
    print(arg)
    time.sleep(1)
    p.add_thread()


for i in range(30):
    Pool = pool.get_thread()
    t = Pool(target=func, args=(i, pool))
    t.start()
'''
```

复杂版本:

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import queue
import threading
import contextlib
import time

StopEvent = object()
```

```
class ThreadPool(object):

    def __init__(self, max_num, max_task_num = None):
        if max_task_num:
            self.q = queue.Queue(max_task_num)
        else:
            self.q = queue.Queue()
        self.max_num = max_num
        self.cancel = False
        self.terminal = False
        self.generate_list = []
        self.free_list = []

    def run(self, func, args, callback=None):
        """
        线程池执行一个任务
        :param func: 任务函数
        :param args: 任务函数所需参数
        :param callback: 任务执行失败或成功后执行的回调函数，回调函数有两个参数1、任务函数执行状态；2、任务函数
        :return: 如果线程池已经终止，则返回True否则None
        """
        if self.cancel:
            return
        if len(self.free_list) == 0 and len(self.generate_list) < self.max_num:
            self.generate_thread()
        w = (func, args, callback,)#主线程
        self.q.put(w)#主线程

    def generate_thread(self):
        """
        创建一个线程
        """
        t = threading.Thread(target=self.call)
        t.start()

    def call(self):
        """
        循环去获取任务函数并执行任务函数
        """
        current_thread = threading.currentThread()
        self.generate_list.append(current_thread)

        event = self.q.get()#if q为空，则阻塞住，一直等到有任务进来并把它取出来
        while event != StopEvent:

            func, arguments, callback = event
            try:
                result = func(*arguments)
                success = True
            except Exception as e:
                success = False
                result = None

            if callback is not None:
                try:
                    callback(success, result)
                except Exception as e:
                    pass

            with self.worker_state(self.free_list, current_thread):
                if self.terminal:
                    event = StopEvent
                else:
                    event = self.q.get()#key: 该线程在这里继续等待新的任务，任务来了，继续执行
                                        #暂时将该线程对象放到free_list中。
        else:

            self.generate_list.remove(current_thread)

    def close(self):
        """
        执行完所有的任务后，所有线程停止
        """
        self.cancel = True
        full_size = len(self.generate_list)
        while full_size:
            self.q.put(StopEvent)
```

```
            full_size -= 1

    def terminate(self):
        """
        无论是否还有任务，终止线程
        """
        self.terminal = True

        while self.generate_list:
            self.q.put(StopEvent)

        self.q.queue.clear()

    @contextlib.contextmanager
    def worker_state(self, free_list, worker_thread):
        """
        用于记录线程中正在等待的线程数
        """
        free_list.append(worker_thread)#新的任务来的时候判断
                                # if len(self.free_list) == 0 and len(self.generate_list) < self
                                # 任务得创建新的线程来处理；如果len(self.free_list) ！= 0：由阻塞着的存
        try:
            yield
        finally:
            free_list.remove(worker_thread)

# How to use


pool = ThreadPool(5)

def callback(status, result):
    # status, execute action status
    # result, execute action return value
    pass


def action(i):
    time.sleep(1)
    print(i)

for i in range(30):
    ret = pool.run(action, (i,), callback)

time.sleep(2)
print(len(pool.generate_list), len(pool.free_list))
print(len(pool.generate_list), len(pool.free_list))

# pool.close()
# pool.terminate()
```

延伸：

```
import contextlib
import socket
@contextlib.contextmanager
def context_socket(host,port):
    sk=socket.socket()
    sk.bind((host,port))
    sk.listen(5)
    try:
        yield sk
    finally:sk.close()

with context_socket('127.0.0.1',8888) as socket:
    print(socket)
```

Yuan先生
关注 - 1
粉丝 - 3948
我在关注他 取消关注

8 0

posted @ 2016-08-03 17:54 Yuan先生 阅读(8798) 评论(4) 编辑 收藏

Post Comment

#1楼 2017-08-16 10:41 | Jasonmer_ND 回复 引用

老师，为什么您的主页看不到你的文章了。

支持(0) 反对(0)

#2楼 2017-09-27 20:03 | Jasonmer_ND 回复 引用

老师你有个地方写错了。。
将一个值放入队列中
q.put(10)
调用队列对象的put()方法在队尾插入一个项目。put()有两个参数，第一个item为必需的，为插入项目的值；第二个block为可选参数，默认为
1。如果队列当前为空且block为1，put()方法就使调用线程暂停,直到空出一个数据单元。如果block为0，put方法将引发Full异常。

应该是队列当前满了且block为1，，才暂停线程
队列满了且block为0，引发Full异常

支持(0) 反对(0)

#3楼 2017-11-13 08:59 | jiew4ever 回复 引用

五 条件变量同步(Condition) 示例中使用print('ok') 对wait()后被激活进行测试,测试结果中控制台上是能多次看到 ok 的

支持(0) 反对(1)

#4楼 2017-11-27 17:33 | 西木野挖掘姬 回复 引用

五 条件变量同步(Condition) 我把consumer线程放在了列表第一个 就能显示ok了 在wait前面再加上一句print('xxx') 测试会发现无论如何都会第一个显示 xxx 这是不是意味着 threads列表里顺序后面的线程抢不过前面的？ 在原程序中 之所以consumer之前有5个包子 是因为consumer在最后 一直都抢不过把？老师可以试下改变列表里线程的顺序

支持(0) 反对(0)