多进程多线程

内存共享

通过Value，Array实现内存共享

```
from multiprocessing import Process
from multiprocessing import Value
from multiprocessing import Array
import sys

class Counter(Process):

  def __init__(self,id,name,counter,checkin):
    super().__init__()
    self._id=id
    self._name=name
    self._counter=counter
    self._checkin=checkin


  def run(self):
    with self._counter.get_lock():#large number of process will trigger error
      self._counter.value+=1
    print('{} update counter to {}'.format(self._name,self._counter.value))
    self._checkin[self._id]=1


if __name__=='__main__':
  num_processes=int(sys.argv[1])
  counter=Value('i',0,lock=True)
  checkin=Array('i',num_processes,lock=True)
  clist=[]
  for i in range(num_processes):
    c=Counter(i,'counter-{}'.format(i),counter,checkin)
    clist.append(c)
  for c in clist:
    c.start()
  for c in clist:
    c.join()
  print('child processes end...')
  print('in the main process, counter.value={}'.format(counter.value))
  for ck in checkin:
    print(ck,end=' ')
  print('')
```

通过Manager实现内存共享

```
from multiprocessing import Process
from multiprocessing import Manager, Lock
import time
import random


def register(d,name):
  if name in d:
    print('duplicated name found...register anyway...')
    d[name]+=1
  else:
    d[name]=1
  time.sleep(random.random())

def register_with_loc(d,name,lock):
  with lock:
    if name in d:
      print('duplicated name found...register anyway...')
      d[name]+=1
    else:
      d[name]=1
  time.sleep(random.random())


if __name__=='__main__':
  #Manager
  names=['Amy','Lily','Dirk','Lily', 'Denial','Amy','Amy','Amy']
  manager=Manager()
  dic=manager.dict()
  lock=Lock()
  #manager.list()
  students=[]
  for i in range(len(names)):
    #s=Process(target=register,args=(dic,names[i]))
    s=Process(target=register_with_loc,args=(dic,names[i],lock))
    students.append(s)
  for s in students:
    s.start()
  for s in students:
    s.join()
  print('all processes ended...')
  for k,v in dic.items():
    print("{}\t{}".format(k,v))
```

进程池

```
from multiprocessing import Process
from multiprocessing import Pool
from multiprocessing import current_process
import matplotlib.pyplot as plt
import os
import time

global_result=[]

def fib(max):
  n,a,b=0,0,1
  while n<max:
    a,b=b,a+b
    n+=1
  return b

def job(n):
  print('{} is working on {}...'.format(os.getpid(),n))
  time.sleep(2)
  return fib(n)

def add_result(res):#callback func
  global global_result
  print("called by {}, result is {}".format(current_process().pid,res))
  #也可以返回进程标识信息，用以识别结果（比如进程的参数）
  #可以用字典存储
  global_result.append(res)

def add_result_map(res):
  global global_result
  print("called by {}, result is {}".format(current_process().pid,res))
  for r in res:
    global_result.append(r)

if __name__=='__main__':
  p=Pool()#cpu determines
  ms=range(1,20)
  results=[]
  #同步调用
  #创建多个进程，但是只有一个执行，需要等到执行结束后再可以返结果
  #并不并行
  '''for m in ms:
    print('{} will be applied in main'.format(m))
    res=p.apply(job,args=(m,))#会等待执行结束后再执行下一个
    print(res)
    print('{} is applied in main'.format(m))
    results.append(res)
  p.close()#!!!
  print(results)
```

```
plt.figure()
plt.plot(ms,results)
plt.show()
plt.close()'''

#异步调用
#可以并行
'''for m in ms:
  res=p.apply_async(job,args=(m,))#注意这里res只是一个引用
  results.append(res)
  #如果马上打印，可能并没有结果
  print(res)
p.close()
p.join()
results2=[]
for res in results:
  results2.append(res.get())
print(results2)
plt.figure()
plt.plot(ms,results2)
plt.show()
plt.close()'''

#callback
'''for m in ms:
  p.apply_async(job,args=(m,),callback=add_result)
  #callback函数只有一个参数
  #callback函数是由主进程执行的
p.close()
p.join()
plt.figure()
plt.plot(ms,sorted(global_result))#顺序可能是乱的，这里可以排序解决，但其他问题不一定
plt.show()
plt.close()'''


#使用map
'''results3=p.map(job,ms)
print(type(results3))#list
p.close()
plt.figure()
plt.plot(ms,results3)
plt.show()
plt.close()'''

#使用map_async
'''p.map_async(job,ms,callback=add_result_map)
p.close()
p.join()
```

```
  plt.figure()
  print(len(ms))
  print(len(global_result))
  plt.plot(ms,global_result)
  plt.show()
  plt.close()'''
```

补充：

- 回调函数的参数只有一个，即结果

- 回调函数是由主进程调用的

- 回调函数应该迅速结束

- 回调的顺序跟子进程启动的顺序无关

- p.map()

• 并行，主进程会等待所有子进程结束

- p.map_async()

进一步抽象的结果

补充：马上调用Future的result()会阻塞

```
import concurrent.futures
from multiprocessing import current_process
import math

PRIMES = [
    1112272535095293,
    1112582705942171,
    1112272535095291,
    1115280095190773,
    1115797848077099,
    11099726899285419]

def is_prime(n):
    print(f"{current_process().pid}")
    if n % 2 == 0:
        return False
    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main_submit():
    results=[]
    with concurrent.futures.ProcessPoolExecutor() as executor:
```

```
        for number in PRIMES:
            n_future=executor.submit(is_prime,number)
            #print(n_future.result())#block
            results.append(n_future)
            #这里要注意，如果马上在主进程里获取结果，即n_future.result()，即主进程会阻塞，无法并行
            #因此建议先将n_future搁进list,等启动所有进程后再获取结果。
        for number, res in zip(PRIMES,results):
            print("%d is prime: %s" % (number,res.result()))

def main_map():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime_or_not in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime_or_not))

if __name__ == '__main__':
    main_submit()
    #main_map()
```

多线程

- 多线程编程
  - threading模块
  - multiprocessing模块和threading模块在使用层面十分相似
  - threading.currentThread()
    - 返回当前的线程实例
  - threading.enumerate()
    - 返回所有正在运行线程的list
  - threading.activeCount()
    - 返回正在运行的线程数量，与len(threading.enumerate())结果相同

创建多线程

```
from threading import Thread,currentThread
```

```
import time

def task(name):
    time.sleep(2)
    print('%s print name: %s' %(currentThread().name,name))

class Task(Thread):
    def __init__(self,name):
        super().__init__()
        self._name=name

    def run(self):
        time.sleep(2)
        print('%s print name: %s' % (currentThread().name,self._name))

if __name__ == '__main__':
    n=100
    var='test'
    t=Thread(target=task,args=('thread_task_func',))
    t.start()
    t.join()

    t=Task('thread_task_class')
    t.start()
    t.join()

    print('main')
```

线程同步

锁(threading.Lock, threading.RLock,可重入锁)

• 一旦线程获得重入锁，再次获取时将不阻塞

• 线程必须在每次获取后释放一次

• 区别：递归调用

– 信号量(threading.Semaphore)

– 事件(threading.Event)

条件

```
from threading import Thread, Condition,currentThread
import time
import random

con=Condition()
q=list()
```

```
def check_q():
  return len(q)

def consume():
  with con:
    con.wait_for(check_q)
    print("%s consume %s" %(currentThread().name,q.pop()))

def produce():
  with con:
    for i in range(2):
      q.append(random.random())
    con.notify_all()

clist=[]
for i in range(20):
  c=Thread(target=consume,name='c-'+str(i+1))
  clist.append(c)
plist=[]
for i in range(20):
  p=Thread(target=produce,name='p-'+str(i+1))
  plist.append(p)
for c in clist:
  c.start()
for p in plist:
  p.start()
```

定时器

```
from threading import Timer
from time import strftime, localtime

def get_time():
    print(strftime("%Y-%m-%d %H:%M:%S", localtime()))

print(strftime("%Y-%m-%d %H:%M:%S", localtime()))
while(1):
    t=Timer(2, get_time)
    t.start()
    t.join()
```

Barrier

• This class provides a simple synchronization

primitive for use by a fixed number of threads

that need to wait for each other.

• Each of the threads tries to pass the barrier by calling the wait() method and will block until all of the threads have made their wait() calls. At this point, the threads are released simultaneously.

```
from threading import Thread, Barrier,currentThread
import time
import random

b=Barrier(5, timeout=5)

def server():
  b.wait()
  print('一桌凑齐，开麻...')

def client():
  time.sleep(random.randint(1,4))
  print('%s 入桌' % currentThread().name)
  b.wait()

Thread(target=server).start()

for i in range(4):
  c=Thread(target=client,name="client-"+str((i+1)))
  c.start()
```

线程局部变量

– 通过threding.local()创建

– 线程在使用该类变量时会创建独立的拷贝

– 看似全局变量（可以被各个线程调用），但各线程拥有各自己独立的副本

– 避免数据的同步

```
from threading import Thread
import threading
import random
import time

class Runner(Thread):
  def __init__(self,name,timer,stime):
    super().__init__()
    self._name=name
```

```
    self._timer=timer
    self._stime=stime

  def run(self):
    self._timer.start=self._stime
    time.sleep(random.randint(1,3))
    print('in {}, timer starts since {}'.format(self._name,self._timer.start))


if __name__=='__main__':
  timer=threading.local()
  timer.start=0
  tlist=[]
  for i in range(10):
    r=Runner('thread-{}'.format(i),timer,random.randint(1,100))
    tlist.append(r)
  for r in tlist:
    r.start()
  for r in tlist:
    r.join()
  print('child threads end')
  print('in main thread, timer.start={}'.format(timer.start))
```

队列

– queue.Queue

– queue.LifoQueue

– queue.PriorityQueue (元素是元组，第一个元素为优先级)

```
from threading import Thread,currentThread
import queue
import random
import time

def worker():
    while True:
        item = q.get()
        if item is None:
            break
        time.sleep(random.randint(1,2))
        print("%s get %s" %(currentThread().name,item))
        q.task_done()

num_worker_threads=10

q = queue.Queue()
```

```
threads = []
for i in range(10):
    t = Thread(target=worker)
    threads.append(t)
    t.start()


for item in "abcdefghijklmnopqrstuvwxyz0123456789":
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()
```

线程池

```
import concurrent.futures
import urllib.request
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

URLS = ['http://www.buaa.edu.cn',
        'https://www.163.com',
        'https://www.baidu.com',
        'https://www.qq.com',
        'https://www.cnbeta.com',
        'https://www.bilibili.com',
        'https://www.mi.com']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    fs = [executor.submit(load_url, url, 60) for url in URLS]
    for url, future in zip(URLS, concurrent.futures.as_completed(fs)):
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
```

```
        else:
            #print(data)
            print('%r page is %d KB' % (url, (len(data)//8)//1024))
```

```
import concurrent.futures
import random
import tqdm
import time
import sys

def task(x):
  time.sleep(x*0.1)
  return x

def download_many(count):
  with concurrent.futures.ThreadPoolExecutor(max_workers=count) as executor:
    todomap=[]
    for x in range(100):
      future=executor.submit(task,x)
      todomap.append(future)
    dointer=concurrent.futures.as_completed(todomap)
    dointer=tqdm.tqdm(dointer,total=len(todomap))
    for future in dointer:
      res=future.result()

if __name__=='__main__':
  download_many(int(sys.argv[1]))
```

进程与线程比较

```
from multiprocessing import Process
from threading import Thread
import os,time,random

def dense_cal():
    res=0
    for i in range(100000000):
        res*=i

def dense_io():
    time.sleep(2)#simulate the io delay

def diff_pt(P=True,pn=4,target=dense_cal):
    tlist=[]
    start=time.time()
    for i in range(pn):
        if P:
```

```
            p=Process(target=target)
        else:
            p=Thread(target=target)
        tlist.append(p)
        p.start()
    for p in tlist:
        p.join()
    stop=time.time()
    if P:
        name='multi-process'
    else:
        name='multi-thread'
    print('%s run time is %s' %(name,stop-start))

if __name__=='__main__':
    diff_pt(P=True)
    diff_pt(P=False)

    diff_pt(P=True,pn=100,target=dense_io)
    diff_pt(P=False,pn=100,target=dense_io)
```