

## 转 Python中使用多进程来实现并行处理的方法小结

2018年07月21日 18:04:05 云net 阅读数：2778

进程和线程是计算机软件领域里很重要的概念，进程和线程有区别，也有着密切的联系，先来辨析一下这两个概念。

## 1.定义

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源,只拥有一点儿微小的、专用的资源,如程序计数器、一组寄存器和栈,但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

## 2.关系

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。

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

### 3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，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。

4) 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，必须依存在进程中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看,多线程的意义在于一个应用程序中,有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用,来实现进程的管理以及资源分配。这就是进程和线程的重要区别。

#### 4.优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机移植。

这篇文章主要讲多进程在Python中的应用

Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动为子进程（称为子进程）复制了一份（称为父进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需调用getpid()就可以拿到父进程的ID。

python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程：

?

```
1 import os
2
3 print('Process (%s) start...' % os.getpid())
4 # Only works on Unix/Linux/Mac:
5 pid = os.fork()
6 if pid == 0:
7     print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
8 else:
9     print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
```

运行结果如下：

Process (876) start...

I (876) just created a child process (877).

I am child process (877) and my parent is 876.

由于Windows没有fork调用，上面的代码在Windows上无法运行。

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父子进程来处理新的http请求。

multiprocessing

如果你打算编写多进程的服务程序，Unix/linux无疑是正确的选择。由于Windows没有fork调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结

？

```
1 from multiprocessing import Process
2 import os
3
4 # 子进程要执行的代码
5 def run_proc(name):
6     print('Run child process %s (%s)...' % (name, os.getpid()))
7
8 if __name__ == '__main__':
9     print('Parent process %s.' % os.getpid())
10    p = Process(target=run_proc, args=('test',))
11    print('Child process will start.')
12    p.start()
13    p.join()
14    print('Child process end.')
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

？

```
1 from multiprocessing import Pool
2 import os, time, random
3
4 def long_time_task(name):
5     print('Run task %s (%s)...' % (name, os.getpid()))
6     start = time.time()
7     time.sleep(random.random() * 3)
8     end = time.time()
9     print('Task %s runs %0.2f seconds.' % (name, (end - start)))
10
11 if __name__ == '__main__':
12     print('Parent process %s.' % os.getpid())
13     p = Pool(4)
14     for i in range(5):
15         p.apply_async(long_time_task, args=(i,))
16     print('Waiting for all subprocesses done...')
17     p.close()
18     p.join()
19     print('All subprocesses done.')
```

执行结果如下：

Parent process 669.  
Waiting for all subprocesses done...  
Run task 0 (671)...  
Run task 1 (672)...  
Run task 2 (673)...  
Run task 3 (674)...  
Task 2 runs 0.14 seconds.  
Run task 4 (673)...  
Task 1 runs 0.27 seconds.

Task 3 runs 0.86 seconds.  
Task 0 runs 1.41 seconds.  
Task 4 runs 1.91 seconds.  
All subprocesses done.

代码解读：

对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，调用close()之后就不能再添加新的Process了。

请注意输出的结果，task 0，1，2，3是立刻执行的，而task 4要等待前面某个task完成后才执行，这是因为Pool的大小在我的电脑上是4，因此，最多行4个进程。这是Pool有意设计的限制，并不是操作系统的限制。如果改成：

?

1	p = Pool(5)
---	-------------

就可以同时跑5个进程。

由于Pool的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令nslookup www.python.org，这和命令行直接运行的效果是一样的：

?

1	import subprocess
2	
3	print('\$ nslookup www.python.org')
4	r = subprocess.call(['nslookup', 'www.python.org'])
5	print('Exit code:', r)

运行结果：

```
$ nslookup www.python.org
Server:      192.168.19.4
Address:     192.168.19.4#53
Non-authoritative answer:
www.python.org canonical name = python.map.fastly.net.
Name:   python.map.fastly.net
Address: 199.27.79.223
Exit code: 0
```

如果子进程还需要输入，则可以通过communicate()方法输入：

?

1	import subprocess
2	
3	print('\$ nslookup')
4	p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
5	output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
6	print(output.decode('utf-8'))
7	print('Exit code:', p.returncode)

上面的代码相当于在命令行执行命令nslookup，然后手动输入：

```
set q=mx
python.org
exit
```

进程间通信

Process之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的multiprocessing模块包装了底层的机制，提供了Queue、Pipe方式来交换数据。

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据：

?

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

31

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__ == '__main__':
    # 父进程创建Queue，并传给各个子进程：
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

👍

0

💬

🔖

📄

<

>

运行结果如下：

Process to write: 50563  
Put A to queue...  
Process to read: 50564  
Get A from queue.  
Put B to queue...  
Get B from queue.  
Put C to queue...  
Get C from queue.

在Unix/Linux下，multiprocessing模块封装了fork()调用，使我们不需要关注fork()的细节。由于Windows没有fork调用，因此，multiprocessing需要 “fork的效果”，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果multiprocessing在Windows下调用失败了，要先考虑是不是p了。

小结

在Unix/Linux下，可以使用fork()调用实现多进程。

要实现跨平台的多进程，可以使用multiprocessing模块。

进程间通信是通过Queue、Pipes等实现的。

多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而出来的线程。

Python的标准库提供了两个模块：\_thread 和 threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下，我们只用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

?

1	import time, threading		0
2			
3	# 新线程执行的代码:		
4	def loop():		
5	print('thread %s is running...' % threading.current_thread().name)		
6	n = 0		
7	while n < 5:		
8	n = n + 1		
9	print('thread %s >>> %s' % (threading.current_thread().name, n))		
10	time.sleep(1)		
11	print('thread %s ended.' % threading.current_thread().name)		
12			
13	print('thread %s is running...' % threading.current_thread().name)		
14	t = threading.Thread(target=loop, name='LoopThread')		
15	t.start()		
16	t.join()		
17	print('thread %s ended.' % threading.current_thread().name)		
18	thread MainThread is running...		
19	thread LoopThread is running...		
20	thread LoopThread >>> 1		
21	thread LoopThread >>> 2		
22	thread LoopThread >>> 3		
23	thread LoopThread >>> 4		
24	thread LoopThread >>> 5		
25	thread LoopThread ended.		
26	thread MainThread ended.		

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。主线程实例的名字叫MainThread，子线程的名字在创建时指定，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有意义，如果不起名字Python就自动给线程命名为Thread-1，Thread-2.....

Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

?

1	import time, threading
2	# 假定这是你的银行存款:
3	balance = 0
4	def change_it(n):
5	# 先存后取，结果应该为0:
6	global balance
7	balance = balance + n
8	balance = balance - n
9	def run_thread(n):
10	for i in range(100000):
11	change_it(n)
12	t1 = threading.Thread(target=run_thread, args=(5,))
13	t2 = threading.Thread(target=run_thread, args=(8,))
14	t1.start()
15	t2.start()
16	t1.join()
17	t2.join()
18	print(balance)

我们定义了一个共享变量balance，初始值为0，并且启动两个线程，先存后取，理论上结果应该为0，但是，由于线程的调度是由操作系统决定的，当t1、t2运行时，只要循环次数足够多，balance的结果就不一定是0了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

?

1	balance = balance + n
---	-----------------------

也分两步：

- 1. 计算balance + n，存入临时变量中；
- 2. 将临时变量的值赋给balance。

也就是可以看成：

？

1	x = balance + n
2	balance = x

数据错误的原因：是因为修改balance需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程同时修改一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改balance的时候，一定不能改。

如果我们要确保balance计算正确，就要给change\_it()上一把锁，当某个线程开始执行change\_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成冲突。创建一个锁就是通过threading.Lock()来实现：

？

1	balance = 0
2	lock = threading.Lock()
3	
4	def run_thread(n):
5	for i in range(100000):
6	# 先要获取锁:
7	lock.acquire()
8	try:
9	# 放心地改吧:
10	change_it(n)
11	finally:
12	# 改完了一定要释放锁:
13	lock.release()

当多个线程同时执行lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用try...finally来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以串行方式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部不能执行，也无法结束，只能靠操作系统强制终止。

多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核。如果把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

？

1	import threading, multiprocessing
2	
3	def loop():
4	x = 0
5	while True:
6	x = x ^ 1
7	
8	for i in range(multiprocessing.cpu_count()):
9	t = threading.Thread(target=loop)
10	t.start()

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有102%，也就是仅使用了一核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python字节码的解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能并行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非使用一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过非阻塞（比如NIO）的方式来实现，不过这样就失去了Python简单、易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程可以各自独立的GIL锁，互不影响。

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而数据的修改必须加锁。但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
?
1 import threading
2
3 # 创建全局ThreadLocal对象:
4 local_school = threading.local()
5
6 def process_student():
7     # 获取当前线程关联的student:
8     std = local_school.student
9     print('Hello, %s (in %s)' % (std, threading.current_thread().name))
10
11 def process_thread(name):
12     # 绑定ThreadLocal的student:
13     local_school.student = name
14     process_student()
15
16 t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
17 t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
18 t1.start()
19 t2.start()
20 t1.join()
21 t2.join()
```

全局变量local\_school就是一个ThreadLocal对象，每个Thread对它都可以读写student属性，但互不影响。你可以把local\_school看成全局变量，但每个线程的local\_school.student都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可以用local\_school.student，还可以绑定其他变量，如local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等等，这样一个线程的所有调用到的处理函数都可以非常方便地访问资源。

一个ThreadLocal变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal解决了参数在一个线程中各个函数之间互相传递的问题。

进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。



多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数t的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可造成整个进程崩溃，因为所有线程共享内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作并将关闭”，其实往往是某个线程出了问题但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性问题，IIS的稳定性就不如Apache。为了解决这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核C多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web服务。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的c语言替换用Python这样运行速度极低的脚本语言，会提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多种这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以开多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是重要的趋势。

对应到Python语言，单进程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到子进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。





举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，通过managers模块把Queue通过网络暴露出去，就可以让其他机器的进程访问Queue了。

我们先看服务进程，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

？



1	import random, time, queue	 0
2	from multiprocessing.managers import BaseManager	
3		
4	# 发送任务的队列:	
5	task_queue = queue.Queue()	
6	# 接收结果的队列:	
7	result_queue = queue.Queue()	
8		
9	# 从BaseManager继承的QueueManager:	
10	class QueueManager(BaseManager):	
11	pass	<
12		
13	# 把两个Queue都注册到网络上, callable参数关联了Queue对象:	>
14	QueueManager.register('get_task_queue', callable=lambda: task_queue)	
15	QueueManager.register('get_result_queue', callable=lambda: result_queue)	
16	# 绑定端口5000, 设置验证码'abc':	
17	manager = QueueManager(address=('', 5000), authkey=b'abc')	
18	# 启动Queue:	
19	manager.start()	
20	# 获得通过网络访问的Queue对象:	
21	task = manager.get_task_queue()	
22	result = manager.get_result_queue()	
23	# 放几个任务进去:	
24	for i in range(10):	
25	n = random.randint(0, 10000)	
26	print('Put task %d...' % n)	
27	task.put(n)	
28	# 从result队列读取结果:	
29	print('Try get results...')	
30	for i in range(10):	
31	r = result.get(timeout=10)	
32	print('Result: %s' % r)	
33	# 关闭:	
34	manager.shutdown()	
35	print('master exit.')	

当我们在一台机器上写多进程程序时，创建的Queue可以直接拿来用，但是，在分布式多进程环境下，添加任务到Queue不可以直接对原始的task\_queue操作，那样就绕过了QueueManager的封装，必须通过manager.get\_task\_queue()获得的Queue接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

?

1	import time, sys, queue
2	from multiprocessing.managers import BaseManager
3	
4	# 创建类似的QueueManager:
5	class QueueManager(BaseManager):
6	pass
7	
8	# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
9	QueueManager.register('get_task_queue')
10	QueueManager.register('get_result_queue')
11	
12	# 连接到服务器，也就是运行task_master.py的机器:
13	server_addr = '127.0.0.1'
14	print('Connect to server %s...' % server_addr)
15	# 端口和验证码注意保持与task_master.py设置的完全一致:
16	m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
17	# 从网络连接:
18	m.connect()
19	# 获取Queue的对象:
20	task = m.get_task_queue()
21	result = m.get_result_queue()
22	# 从task队列取任务,并把结果写入result队列:
23	for i in range(10):
24	try:
25	n = task.get(timeout=1)
26	print('run task %d * %d...' % (n, n))
27	r = '%d * %d = %d' % (n, n, n*n)
28	time.sleep(1)
29	result.put(r)

