Home

一篇文章搞懂Python中的进程和线程

Nov 28 2016 • diggzhang
2246˚C

PYTHON 编程理论

进程 线程 廖老师的梗

Directory

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

使用 `top` 命令看有哪些进程在跑:

```
Processes: 371 total, 2 running, 15 stuck, 354 sleeping, 3142 threads  17:27:27
Load Avg: 2.57, 2.29, 2.07  CPU usage: 5.26% user, 4.79% sys, 89.94% idle
SharedLibs: 151M resident, 18M data, 17M linkedit.
MemRegions: 107884 total, 6773M resident, 131M private, 2803M shared.
PhysMem: 15G used (2386M wired), 988M unused.
VM: 1564G vsize, 527M framework vsize, 20146031(0) swapins, 21455983(0) swapouts
Networks: packets: 39381163/32G in, 38877596/18G out.
Disks: 4762039/190G read, 7146822/247G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP
96223- Microsoft Ex 0.1  15:30.25 356   2    2216  46M    0B     298M   96223
96200  Microsoft AU 0.0  00:11.45 5     0    169   5956K  0B     2480K  96200
91030  nwjs Helper  0.0  02:06.91 9     0    139   2920K  0B     15M    91027
91029  nwjs Helper  0.0  02:37:00 21    0    178   650M   10M    70M    91027
91028  nwjs Helper  0.0  19:01.66 5     0    76    34M    0B     9464K  91027
```

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像 Word 这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核 CPU 才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

- 一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。
- 还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。
- 当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

来啊，征服进程和线程，越过Python学习的一个大坎儿吧。

多进程 multiprocessing

这里有个前菜：

> 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，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），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` ，而子进程只需要调用 `getppid()` 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：

```
import os

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

运行结果如下：

```
Process (69673) start ...
I (69673) just created a child Process(69674)
I am child proces (69674) and my parent is 69673.
```

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` 服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

但是这个 `fork` 在windows操作系统是没有的。于是出现了处理 `fork` 的通用模块，以保证在不同操作系统间的调用。

`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束：

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

from multiprocessing import Process
import os

"""
    子进程要执行的代码
"""
def run_proc(name):
    print('Run child process %s (%s)' % (name, os.getpid()))

if __name__ == '__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=('test_code',))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
$ python forkbymutilprocessing.py
Parent process 70227.
Child process will start.
Run child process test_code (70228)
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。

进程池 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：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print("Run task %s (%s)..." % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print("Task %s run %0.f seconds." % (name, (end - start)))

if __name__ == "__main__":
    print("Parent process %s." % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print("Waiting for all subprocess done...")
    p.close()
    p.join()
    print("All subprocess done.")
```

执行结果：

```
➜ python3 testpool.py
Parent process 65899.
Waiting for all subprocess done...
Run task 0 (65900)...
Run task 1 (65901)...
```

```
Run task 2 (65902)...
Run task 3 (65903)...
Task 3 run 0 seconds.
Run task 4 (65903)...
Task 2 run 1 seconds.
Task 0 run 2 seconds.
Task 1 run 2 seconds.
Task 4 run 2 seconds.
All subprocess 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()` 之后就不能继续添加新的 `Process` 了。

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` 的默认大小设置成了 `4` (`p = Pool(4)`)，代表着最多同时执行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:

```
p = Pool(5)
```

就可以跑5个进程了。

由于 `Pool` 的默认大小是CPU的核数，如果你拥有8核CPU，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。当试图通过python做一些运维工作的时候， `subprocess` 简直是顶梁柱。

`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 <某个域名>` ，这和命令行直接运行的效果是一样的：

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

```
import subprocess

print("$ nslookup www.yangcongchufang.com")
r = subprocess.call(['nslookup', 'www.yangcongchufang.com'])
print("Exit code: ", r)
```

执行结果：

```
➜ python subcall.py
$ nslookup www.yangcongchufang.com
Server:         219.141.136.10
Address: 219.141.136.10#53

Non-authoritative answer:
Name:    www.yangcongchufang.com
Address: 103.245.222.133

('Exit code: ', 0)
```

如果子进程还需要输入，则可以通过`communicate()`方法输入：

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

import subprocess

print("$ nslookup")
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stde
output, err = p.communicate(b"set q=mx\nyangcongchufang.com\nexit\n")
print(output.decode("utf-8"))
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
yangcongchufang.com
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` 、 `Pipes` 等多种方式来交换数据。

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：

```
# _*_ coding:utf-8 _*_

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进程里的死循环，无法等待结束，只能强制终止
    pr.terminate()
```

实际执行效果：

```
$ python process_comm.py
Process to write: 94327
Put A to queue...
Process to read: 94328
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“模拟”出 fork 的效果，父进程所有 Python 对象必须通过 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 pickle 失败了。

进程小结

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。

我们前面提到了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 ，而不是模拟出来的线程。

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，我们只需要使用 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：

```
# _*_ coding:utf-8 _*_
import time, threading

def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)

# 执行效果:
$ python3 thread_test.py
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
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，它永远返回当前的线程的实例。主线程实例的名字叫 `MainThread` ，子线程的名字创建时指定，我们用 `LoopThread` 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，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修改一个变量，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：

```
# _*_ coding:utf-8 _*_
import time, threading

# 假定这是你银行的存款
balance = 0

def change_it(n):
    # 先存后取，结果应该是0
    global balance
    balance = balance + n
    balance = balance - n
    print balance

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
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` 交替执行时，只要循环次数足够多， `balance` 的结果就不一定是 `0` 了。

实际执行效果：

```
$ python thread_share_var.py
31
$ python thread_share_var.py
42
$ python thread_share_var.py
-6
$ python thread_share_var.py
0
$ python thread_share_var.py
7
$ python thread_share_var.py
13
```

究其原因，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，从而导致多个线程把同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，别的线程一定不能改。

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，不会造成修改的冲突。创建一个锁就是通过 `threading.Lock()` 来实现：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

加锁以后实现的实际效果：

```
$ python thread_share_var.py
0
$ python thread_share_var.py
0
$ python thread_share_var.py
0
```

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单线程模式执行，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，导致多个线程全部挂起，既不能执行，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核心。

要想把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：

```
# _*_ coding:utf-8 _*_
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

for i in range(multiprocessing.cpu_count()):
    t = threading.Thread(target=loop)
    t.start()
```

跑起来就知道什么叫，自己写的bug自己都收不了场了。生活只教会了我们谦逊和努力，然而bug却教会了我们如何抓狂起来。

上面的程序，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%，为什么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线程执行前，必须先获得 GIL 锁，然后，每执行100条字节码，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中只能交替执行，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，除非重写一个不带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C扩展来实现，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进程有各自独立的GIL锁，互不影响。

总结一下。

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。多线程的并发在Python中就是一个美丽的梦。

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，而全局变量的修改必须加锁。

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：

```
def process_student(name):
    std = Student(name)
    # std是局部变量，但是每个函数都要用到它，因此必须传进去
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
```

```
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理的不同 `Student` 对象，不能共享。

如果用一个全局 `dict` 存放所有的 `Student` 对象，然后以 `thread` 自身作为 `key` 获得线程对应的 `Student` 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把std放到全局变量global_dict中：
    global_dict[threading.current_thread()] = std
    do_task_1()
    do_task_2()

def do_task_1():
    # 不传入std,而是根据当前线程查找：
    std = global_dict[threading.current_thread()]
    ...

def do_task_2(arg):
    # 任何函数都可以查找出当前线程的std变量
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 `std` 对象在每层函数中的传递问题，但是，每个函数获取 `std` 的代码有点丑。

有没有更简单的方式？

`ThreadLocal` 应运而生，不用查找 `dict` ， `ThreadLocal` 帮你自动做这件事：

```
# _*_ coding:utf-8 _*_
import threading
```

```
# 创建全局ThreadLocal对象：
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定ThreadLocal的student
    local_school.student = name
    process_student()

t1 = threading.Thread(target=process_thread, args=('Alice',), name='Thread-A')
t2 = threading.Thread(target=process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()

# 执行结果
$ python thread_local.py
Hello, Alice (in Thread-A)
 Hello, Bob (in Thread-B)
$ python thread_local.py
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

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` 看成是全局变量，但每个属性例如 `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` ，还可以绑定其它变量，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，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这些资源。

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进程只负责分配任务，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也是有限的，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能直接造成整个进程崩溃，因为所有线程共享进程的内存。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，即将关闭”，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的问题，IIS的稳定性就不如Apache。为了缓解这个问题，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CPU执行多任务是一样的了，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（这叫准备新环境），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，把新任务的执行环境准备好（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，操作系统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也可以用多任务完成，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当等于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（因为IO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应用。

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，完全无法提升运行效率。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，我们才需要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任务，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，可以运行多个进程（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一个主要的趋势。

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上。

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，将任务分布到其他多个进程中，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的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：

```
# _*_ coding:utf-8 _*_
import random, time, queue
from multiprocessing.managers import BaseManager

# the queue of send tasks
task_queue = queue.Queue()
# the queue of recive tasks
```

```
result_queue = queue.Queue()

# cong BaseManager jicheng de QueueManager
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:
QueueManager.register('get_task_queue', callable=lambda: task_queue)
QueueManager.register('get_result_queue', callable=lambda: result_queue)

# 绑定端口5000, 设置验证码'abc':
manager = QueueManager(address=('', 5000), authkey=b'abc')

# 启动Queue
manager.start()

# 获得通过网络访问的Queue对象
task = manager.get_task_queue()
result = manager.get_result_queue()

# 放几个任务进去
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)

# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)

# Close
manager.shutdown()
print('master exit.')
```

请注意，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` 进行操作，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）：

```
# task_worker.py

import time, sys, queue
from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器，也就是运行task_master.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与task_master.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务,并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动`task_master.py`服务进程：

```
$ python3 task_master.py
Put task 3411...
Put task 1605...
Put task 1398...
```

```
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...
Put task 7866...
Try get results...
```

`task_master.py`进程发送完任务后，开始等待`result`队列的结果。现在启动`task_worker.py`进程：

```
$ python3 task_worker.py
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

`task_worker.py`进程结束，在`task_master.py`进程中会继续打印出结果：

```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n*n的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到 `task_worker.py` 中根本没有创建Queue的代码，所以，Queue对象存储在 `task_master.py` 进程中。

而Queue之所以能通过网络访问，就是通过 `QueueManager` 实现的。由于 `QueueManager` 管理的不止一个Queue，所以，要给每个Queue的网络调用接口起个名字，比如 `get_task_queue` 。

`authkey` 有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果 `task_worker.py` 的 `authkey` 和 `task_master.py` 的 `authkey` 不一致，肯定连接不上。

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

- T.B.C -
