**第五部分 进程和线程**

##### 多进程

###### 多进程

multiprocessing是python中开启多进程的一个模块。

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：

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',))

print('Child process will start.')

p.start()

p.join()

print('Child process end.')

执行结果如下：

Parent process 928.

Process will start.

Run child process test (929)...

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：

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 runs %0.2f 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 subprocesses done...')

p.close()

p.join()

print('All subprocesses done.')

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。

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的默认大小在我的电脑上是4，因此，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)

就可以同时跑5个进程。

由于Pool的默认大小是CPU的核数，如果拥有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：

import subprocess

print('$ nslookup www.python.org')

r = subprocess.call(['nslookup', 'www.python.org'])

print('Exit code:', r)

###### 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：

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()

multiprocessing进程间通信，需要将数据通过pickle序列化再传到子进程去，所以，如果multiprocessing在Windows下调用失败了，要先考虑是不是因为序列化的问题。

##### 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，只需要使用threading这个高级模块。

###### 创建线程

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：

import time, threading

# 新线程执行的代码:

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)

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：

import time, threading

# 假定这是你的银行存款:

balance = 0

def change\_it(n):

# 先存后取，结果应该为0:

global balance

balance = balance + n

balance = balance - n

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。

原因是因为高级语言的一条语句，会被翻译成若干条指令在CPU上执行，即使一个简单的计算：

balance = balance + n

也分两步：

计算balance + n，存入临时变量中；

将临时变量的值赋给balance。

也就是可以看成：

x = balance + n

balance = x

由于x是局部变量，两个线程各自都有自己的x，当t1和t2交替运行时，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()

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。

###### GIL

一般情况下，多核CPU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。

用Python写个死循环：

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()

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。

###### 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，因为每个线程处理不同的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():

# 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的代码有点别扭，更简单的方式是：使用ThreadLocal应运而生。

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()

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，但每个属性如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，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

##### 协程

在学习异步IO模型前，我们先来了解协程。

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

CPU中断时，多线程的执行就会切换，整体上看多线程是在交替进行。协程看上去类似多线程，因为协程在执行函数调用时也会发生中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

def A():

print('1')

print('2')

print('3')

def B():

print('x')

print('y')

print('z')

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

1

2

x

y

3

z

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但**协程的特点在于是一个线程执行**，那和多线程比，协程有何优势？

**最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。**

**第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。**

因为协程是一个线程执行，那怎么利用多核CPU呢？**最简单的方法是多进程+协程**，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过for循环来迭代，还可以不断调用next()函数获取由yield语句返回的下一个值。

但是Python的yield不但可以返回一个值，它还可以接收调用者发出的参数。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

def consumer():

r = ''

while True:

n = yield r

if not n:

return

print('[CONSUMER] Consuming %s...' % n)

r = '200 OK'

def produce(c):

c.send(None)

n = 0

while n < 5:

n = n + 1

print('[PRODUCER] Producing %s...' % n)

r = c.send(n)

print('[PRODUCER] Consumer return: %s' % r)

c.close()

c = consumer()

produce(c)

执行结果：

[PRODUCER] Producing 1...

[CONSUMER] Consuming 1...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 2...

[CONSUMER] Consuming 2...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 3...

[CONSUMER] Consuming 3...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 4...

[CONSUMER] Consuming 4...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 5...

[CONSUMER] Consuming 5...

[PRODUCER] Consumer return: 200 OK

注意到consumer函数是一个generator，把一个consumer传入produce后：

步骤1.首先调用c.send(None)启动生成器；

步骤2.然后，一旦生产了东西，通过c.send(n)切换到consumer执行；

步骤3.consumer通过yield拿到消息，处理，又通过yield把结果传回；

步骤4.produce拿到consumer处理的结果，继续生产下一条消息；

步骤5.produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。