多线程编程

进程(process)和线程(Thread)

Linux是一个多任务操作系统。 这就意味着一次可以运行一个以上的程序。 每个占用一定时间运行的程序就叫 一个进程。 你运行的每一个命令会至少启动一个新进程，还有很多一直运行着的系统进程， 用以维持系统的正常运作。但是个人电脑甚至服务器的cpu内核是有限的，而进程上要远大于CPU个数，那怎么执行多任务呢？答案就是操作系统轮流让各个任务交替执行，任务轮流切换到前台，执行0.01秒让出切换其他的任务，就这样反复执行下去。本质上每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，使用者的感官像所有任务都在同时执行一样。

有些进程还不止同时干一件事，比如打开浏览器虽然是一个进程，它可以同时访问多个网页，能输入网址，填写表单，鼠标点击翻页等等。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。

如何提高任务执行效率?

1. 多进程模式。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
2. 多线程模式。启动一个进程，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
3. 多进程+多线程模式。启动多个进程，每个进程再启动多个线程

并发(Concurrency)和并行(Parallelism)

并发: 当有多个线程在操作时,如果系统只有一个CPU,则它根本不可能真正同时进行一个以上的线程，它只能把CPU运行时间划分成若干个时间段,再将时间 段分配给各个线程执行，在一个时间段的线程代码运行时，其它线程处于挂起状

并行: 当系统有一个以上CPU时, 则线程的操作有可能非并发。当一个CPU执行一个线程时，另一个CPU可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行

并发的关键是你有处理多个任务的能力，不一定要同时

并行的关键是你有同时处理多个任务的能力。并发和并行的关键就在于是否能同时处理

可以说“并行”概念是“并发”概念的一个子集，也就是说，你可以编写一个拥有多个线程或者进程的并发程序，但如果没有多核处理器来执行这个程序，那么就不能以并行方式来运行代码

-- 《并发的艺术》

全局解释锁 (Global Interpreter Lock - GIL)

GIL是计算机程序设计语言解释器用于同步线程的一种机制，它使得任何时刻仅有一个线程在执行。

CPython是用C语言实现的Python解释器。 作为官方实现，它是最广泛使用的Python解释器，在CPython里就用到了GIL，GIL也是经常被其他语言开发者吐槽Python语言的一个槽点。

CPython 内存管理不是线程安全的，因此需要 GIL 来保证多个原生线程不会并发执行 Python 字节码。所谓存在即合理，它在单线程的情况更快，并且在和 C 库结合时更方便，而且不用考虑线程安全问题，这也是早期 Python 最常见的应用场景和优势。

全局解释锁 (Global Interpreter Lock - GIL)

GIL是计算机程序设计语言解释器用于同步线程的一种机制，它使得任何时刻仅有一个线程在执行。

CPython是用C语言实现的Python解释器。 作为官方实现，它是最广泛使用的Python解释器，在CPython里就用到了GIL，GIL也是经常被其他语言开发者吐槽Python语言的一个槽点。

CPython 内存管理不是线程安全的，因此需要 GIL 来保证多个原生线程不会并发执行 Python 字节码。所谓存在即合理，它在单线程的情况更快，并且在和 C 库结合时更方便，而且不用考虑线程安全问题，这也是早期 Python 最常见的应用场景和优势。

threading简单例子

```python
import threading


def worker():
    print('Worker')


threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

```
❯ python threading_simple.py
Worker
Worker
Worker
Worker
Worker
```

```python
import threading


def worker(num):
    print(f'Worker: {num}')


threads = []
for i in range(5):
    t = threading.Thread(target=worker,
                         args=(i,))
    threads.append(t)
    t.start()
```

```
❯ python threading_simpleargs.py
Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

守护和非守护线程

```python
import threading
import time


def daemon():
    print('Daemon Starting')
    time.sleep(0.2)
    print('Daemon Exiting')


def non_daemon():
    print('NonDaemon Starting')
    print('NonDaemon Exiting')


d = threading.Thread(name='daemon', target=daemon,
                     daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

```
❯ python threading_daemon.py
Daemon Starting
NonDaemon Starting
NonDaemon Exiting
```

```python
import threading
import time


def daemon():
    print('Daemon Starting')
    time.sleep(0.2)
    print('Daemon Exiting')


def non_daemon():
    print('NonDaemon Starting')
    print('NonDaemon Exiting')


d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)
t = threading.Thread(name='non-daemon',
                     target=non_daemon)

d.start()
t.start()
d.join()
t.join()
```

```
❯ python threading_daemon_join.py
Daemon Starting
NonDaemon Starting
NonDaemon Exiting
Daemon Exiting
```

同步机制

Semaphore(信号量)

```python
import time
from random import random
from threading import Thread, Semaphore

sema = Semaphore(3)


def foo(tid):
    with sema:
        print('{} acquire sema'.format(tid))
        wt = random() * 2
        time.sleep(wt)
    print('{} release sema'.format(tid))


threads = []

for i in range(5):
    t = Thread(target=foo, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

```
❯ python threading_semaphore.py
0 acquire sema
1 acquire sema
2 acquire sema
0 release sema
3 acquire sema
3 release sema
4 acquire sema
2 release sema
1 release sema
4 release sema
```

Lock(锁)

```python
import time
from threading import Thread

value = 0


def getlock():
    global value
    new = value + 1
    time.sleep(0.001)  # 使用sleep让线程有机会切换
    value = new


threads = []

for i in range(100):
    t = Thread(target=getlock)
    t.start()
    threads.append(t)

for t in threads:
    t.join()

print(value)
```

不加锁 ⬅️

```
❯ python threading_nolock.py
10
```

```python
import time
from threading import Thread, Lock

value = 0
lock = Lock()


def getlock():
    global value
    with lock:
        new = value + 1
        time.sleep(0.001)
        value = new

threads = []

for i in range(100):
    t = Thread(target=getlock)
    t.start()
    threads.append(t)

for t in threads:
    t.join()

print(value)
```

加锁 ⬅️

```
❯ python threading_lock.py
100
```

RLock(可重入锁)

```python
import threading

lock = threading.Lock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))


lock = threading.RLock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

```
❯ python threading_rlock.py
First try : True
Second try: False
First try : True
Second try: True
```

Condition(条件)

```python
import time
import threading

def consumer(cond):
    t = threading.currentThread()
    with cond:
        cond.wait()  # 等待满足的条件
        print(f'{t.name}: Resource is available to consumer')


def producer(cond):
    t = threading.currentThread()
    with cond:
        print(f'{t.name}: Making resource available')
        cond.notifyAll()  # 唤醒消费者


condition = threading.Condition()

c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
p = threading.Thread(name='p', target=producer, args=(condition,))

c1.start()
time.sleep(1)
c2.start()
time.sleep(1)
p.start()
```

```
❯ python threading_condition.py
p: Making resource available
c1: Resource is available to consumer
c2: Resource is available to consumer
```

Event(事件)

```python
import time
import threading
from random import randint


def consumer(event, l):
    t = threading.currentThread()
    while 1:
        event_is_set = event.wait(2)
        if event_is_set:
            try:
                integer = l.pop()
                print(f'{integer} popped from list by {t.name}')
                event.clear()  # 重置事件状态
            except IndexError:  # 为了让刚启动时容错
                pass


def producer(event, l):
    t = threading.currentThread()
    while 1:
        integer = randint(10, 100)
        l.append(integer)
        print(f'{integer} appended to list by {t.name}')
        event.set()       # 设置事件
        time.sleep(1)


event = threading.Event()
l = []

threads = []

for name in ('consumer1', 'consumer2'):
    t = threading.Thread(name=name, target=consumer, args=(event, l))
    t.start()
    threads.append(t)

p = threading.Thread(name='producer1', target=producer, args=(event, l))
p.start()
threads.append(p)

for t in threads:
    t.join()
```

```
❯ python threading_event.py
65 appended to list by producer1
65 popped from list by consumer2
29 appended to list by producer1
29 popped from list by consumer1
32 appended to list by producer1
32 popped from list by consumer2
```

Queue(队列)

```python
import time
import threading
from random import random
from queue import Queue

q = Queue()


def double(n):
    return n * 2


def producer():
    while 1:
        wt = random()
        time.sleep(wt)
        q.put((double, wt))


def consumer():
    while 1:
        task, arg = q.get()
        print arg, task(arg)
        q.task_done()


for target in(producer, consumer):
    t = threading.Thread(target=target)
    t.start()
```

1. put: 向队列中添加一个项。
2. get: 从队列中删除并返回一个项。
3. task_done: 当某一项任务完成时调用。
4. join: 阻塞直到所有的项目都被处理完。

```
❯ python threading_queue.py
0.6460134753838902 1.2920269507677804
0.3753490007022894 0.7506980014045788
0.802132153297252 1.604264306594504
...
```

PriorityQueue(优先级队列)

```python
import time
import threading
from random import randint
from queue import PriorityQueue


q = PriorityQueue()


def double(n):
    return n * 2


def producer():
    count = 0
    while 1:
        if count > 5:
            break
        pri = randint(0, 100)
        print(f'put :{pri}')
        q.put((pri, double, pri))  # (priority, func, args)
        count += 1


def consumer():
    while 1:
        if q.empty():
            break
        pri, task, arg = q.get()
        print(f'[PRI:{pri}] {arg} * 2 = {task(arg)}')
        q.task_done()
        time.sleep(0.1)


t = threading.Thread(target=producer)
t.start()
time.sleep(1)
t = threading.Thread(target=consumer)
t.start()
```

```
❯ python priority_queue.py
put :29
put :28
put :50
put :22
put :59
put :84
[PRI:22] 22 * 2 = 44
[PRI:28] 28 * 2 = 56
[PRI:29] 29 * 2 = 58
[PRI:50] 50 * 2 = 100
[PRI:59] 59 * 2 = 118
[PRI:84] 84 * 2 = 168
```

线程池

```python
import time
import threading
from random import random
from queue import Queue


def double(n):
    return n * 2


class Worker(threading.Thread):
    def __init__(self, queue):
        super(Worker, self).__init__()
        self._q = queue
        self.daemon = True
        self.start()
    def run(self):
        while 1:
            f, args, kwargs = self._q.get()
            try:
                print(f'USE: {self.name}')
                print(f(*args, **kwargs))
            except Exception as e:
                print(e)
            self._q.task_done()


class ThreadPool:
    def __init__(self, num_t=5):
        self._q = Queue(num_t)
        # Create Worker Thread
        for _ in range(num_t):
            Worker(self._q)
    def add_task(self, f, *args, **kwargs):
        self._q.put((f, args, kwargs))
    def wait_complete(self):
        self._q.join()


pool = ThreadPool()
for _ in range(8):
    wt = random()
    pool.add_task(double, wt)
    time.sleep(wt)
pool.wait_complete()
```

```
In : from multiprocessing.pool import ThreadPool
In : pool = ThreadPool(5)
In : pool.map(lambda x: x**2, range(5))
Out: [0, 1, 4, 9, 16]
```

```
❯ python threading_pool.py
USE: Thread-1
1.2242336274294512
USE: Thread-2
0.33551030923669845
USE: Thread-3
1.983203032740676
USE: Thread-4
0.17497357459817908
USE: Thread-5
0.800482952348968
USE: Thread-1
0.5387715034373937
USE: Thread-2
1.4306829877074754
USE: Thread-3
0.3191099710464387
```

延伸阅读

1. https://blog.golang.org/concurrency-is-not-parallelism
2. https://wiki.python.org/moin/GlobalInterpreterLock
3. https://pymotw.com/3/threading/index.html