美吉生物 Majorbio

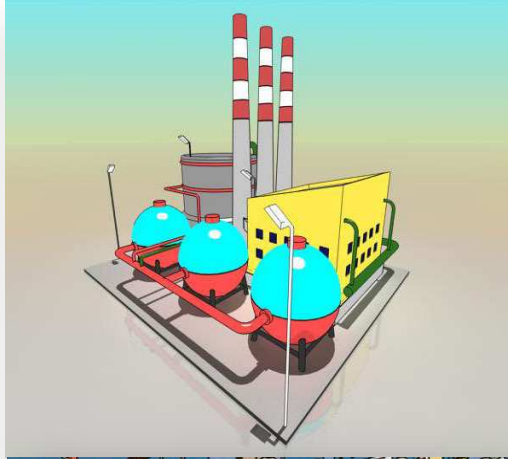
python多线程与多进程

——何胜 2016.4.25

美吉生物 Majorbio

计算机与工厂的类比

计算机(单核)	工厂
一个CPU	一个供电系统
多个进程	多个车间
一个CPU同一时刻仅能有一个进程在运行	假设 供电系统仅能供一个车间生产
一个进程有很多线程	一个车间有很多工人完成同一个任务
一个进程的多个线程是共享内存的	车间的东西工人们都是共享的
线程的防止多个线程同时修改一个共享内存,需要加锁,锁上时其他线程等待	车间有些房间很小,有的只能一个人使用,进去的人会上锁， 出来再开锁
某些内存区域可以共多个线程使用,可以有多个线程就需要上多少把锁， 超过就等待	有些房间虽然小， 但是可以容纳多个人,为了人数不超， 需要挂多个锁表明人数



进程是资源分配的最小单位

- 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；
- 系统进行资源分配和调度的一个独立单位；
- 进程本身不会运行，是线程的容器；
- 因创建而产生，因调度而运行，因等待资源或事件而被等待，因完成任务被撤消；

线程是CPU调度的最小单位

- 操作系统能够进行运算调度的最小单位；
- 一条线程指的是进程中一个单一顺序的控制流；
- 一个进程中可以并发多个线程，每条线程并行执行不同的任务；
- 同一进程中的多条线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等；

Python线程

优点

- 性能改善：不用复制和启动进程，减少性能开销。
- 简单易用：比进程需要考虑的方面少。
- 共享全局内存：线程均在同一个进程中运行，进程全局内存空间供所有线程使用。
- 可移植性：在不同平台上线程比进程更具可移植性。

缺陷

- 函数调用：线程只能是函数调用，不是启动新的程序，当然也可以在线程中使用*os.system*或者*subprocess*启动新程序。
- 线程同步和队列：共享的内存使用会造成使用冲突。
- 全局解释器锁*(GIL)*：*python2.7*中进程中的线程必须获取一个全局解释器锁才能运行，而锁只有一把。所有线程并不能真正在多核*CPU*上进行，在单个*CPU*性能不足的情况下，多线程并不会改善，反而会更慢。

*threading*模块

*threading*的基础方法：

- *activeCount:*正在运行的线程数量，记得主线程的存在，子线程*start*后才会存在
- *currentThread:*返回当前线程对象
- *enumerate:*返回当前正在运行的线程对象列表

threading模块

简单实例：

代码：

```
import time
import threading

def thread_func():
    for i in range(4):
        time.sleep(0.2)
        print('child thread:' + str(i))

thread_1 = threading.Thread(target=thread_func, name='my_thread_1')
thread_1.start()

for i in range(4):
    time.sleep(0.3)
    print('main  thread:' + str(i))

print('main   end!!!')
```

结果：

```
child thread:0
main  thread:0
child thread:1
child thread:2
main  thread:1
child thread:3
main  thread:2
main  thread:3
main   end!!!
[Finished in 1.332s]
```

threading模块

threading.Thread对象

初始化：threading.Thread(group=None, target=None, name=None, args=(), kwargs={})

- group:扩展保留，未使用
- target:可调用目标对象，一般为函数
- name:没有实际意义的线程名称，用于识别线程，但是多个线程可以取相同的名称，不提供默认为Thread-N(N为十进制数字)
- args:target对象的元组参数，如果只有一个参数用(参数,)，需要逗号，否则为字符串。
- kwargs:对象的字典参数，默认为空{}

threading.Timer对象：threading.Thread的子类，设定固定时间后运行

threading模块

threading.Thread对象

基本属性：

- daemon:是否是守护线程(守护线程从逻辑上依赖主线程的存在，如果是，主线程退出时会Kill它的守护线程)
- ident:线程标识符，没有start前为None，启动后为唯一标识符，在一个线程结束后，标识符可重用
- name:线程无实际意义名称

基本方法：

- isAlive:bool返回，线程是否存在，start后为True，结束后为False
- isDaemon:bool返回，是否是守护线程
- join:等待，直至此线程终止，有参数为时间，即等待多久，时间到了，不管线程是否结束，都继续
- start:安排一个新线程运行run函数
- run:在新线程中运行的代码段，标准run调用target参数对象并运行
- setDaemon:参数为bool，True即设置线程为守护线程，设置守护线程必须在start之前

threading模块

守护线程实例：

代码：

```
import time
import threading

def thread_func():
    for i in range(4):
        time.sleep(0.5)
        print('child thread:' + str(i))

thread_1 = threading.Thread(target=thread_func)
thread_1.start()
print(thread_1.daemon)

for i in range(4):
    time.sleep(0.1)
    print('main  thread:' + str(i))

print('main   end!!!')
```

结果：

```
False
main  thread:0
main  thread:1
main  thread:2
main  thread:3
main   end!!!
child thread:0
child thread:1
child thread:2
child thread:3
[Finished in 2.152s]
```

threading模块

守护线程实例：

代码：

```
import time
import threading

def thread_func():
    for i in range(4):
        time.sleep(0.5)
        print('child thread:' + str(i))
thread_1 = threading.Thread(target=thread_func)
thread_1.setDaemon(True)
thread_1.start()
print(thread_1.daemon)

for i in range(4):
    time.sleep(0.1)
    print('main  thread:' + str(i))

print('main   end!!!')
```

结果：

```
True
main  thread:0
main  thread:1
main  thread:2
main  thread:3
main   end!!!
[Finished in 0.504s]
```

threading模块

主线程等待实例：

代码：

```
import time
import threading

def thread_func():
    for i in range(4):
        time.sleep(0.2)
        print('child thread:' + str(i))

thread_1 = threading.Thread(target=thread_func)
thread_1.setDaemon(True)
thread_1.start()
thread_1.join()
for i in range(4):
    time.sleep(0.1)
    print('main  thread:' + str(i))

print('main   end!!!')
```

结果：

```
child thread:0
child thread:1
child thread:2
child thread:3
main  thread:0
main  thread:1
main  thread:2
main  thread:3
main   end!!!
[Finished in 1.439s]
```

threading模块

同步访问共享对象混乱实例:(一个python进程只有一个sys.stdout)

代码:

```
import time
import threading

main_i = 0
def thread_func(a):
    time.sleep(0.1)
    global main_i
    main_i = main_i + 1
    print('child thread %s:%s' % (a, str(main_i)))
multi_thread = []
for i in range(20):
    multi_thread.append(threading.Thread(target=thread_func, args=(i, )))
for one_thread in multi_thread:
    one_thread.start()

time.sleep(1)
print('main count:' + str(main_i))
```

结果:

```
child thread 19:1child thread 18:2child thread 17:3child
thread 15:4child thread 14:5child thread 13:6


child thread 16:7


child thread 12:8child thread 11:9child thread 10:10child
thread 9:11child thread 8:12child thread 7:13child thread
5:14child thread 4:15child thread 3:16child thread 2:17child
thread 1:18child thread 0:19









child thread 6:14
main count:14
[Finished in 1.098s]
```

threading模块

互斥锁的使用实例：

代码：

```
import time
import threading
main_i = 0
mutex_lock = threading.Lock()   # 生成一个锁对象

def thread_func(a):
    time.sleep(0.1)
    mutex_lock.acquire()
    global main_i
    main_i = main_i + 1
    print('child thread %s:' % a + str(main_i))
    mutex_lock.release()
multi_thread = []
for i in range(20):
    multi_thread.append(threading.Thread(target=thread_func, args=(i, )))
for one_thread in multi_thread:
    one_thread.start()
time.sleep(1)
print('main count:' + str(main_i))
```

```
with mutex_lock:
    global main_i
    main_i = main_i + 1
    print('child thread %s:' % a + str(main_i))
```

结果：

```
child thread 18:1
child thread 19:2
child thread 16:3
child thread 17:4
child thread 15:5
child thread 13:6
child thread 14:7
child thread 10:8
child thread 12:9
child thread 11:10
child thread 9:11
child thread 8:12
child thread 7:13
child thread 6:14
child thread 5:15
child thread 4:16
child thread 3:17
child thread 0:18
child thread 2:19
child thread 1:20
main count:20
[Finished in 1.12s]
```

threading模块

threading.RLock对象：与threading.Lock锁相似，不同之处在于，同一个线程中可以重复获取锁，不会产生死锁。

代码：

```
import threading
lock = threading.Lock()  # Lock对象
lock.acquire()
lock.acquire()  # 产生了死琐。
lock.release()
lock.release()
```

```
import threading
rLock = threading.RLock()  # RLock对象
rLock.acquire()
rLock.acquire()  #在同一线程内，程序不会堵塞。
rLock.release()
rLock.release()
```

threading模块

Quene.Quene队列对象:(队列数据结构特点——先进先出的类似列表对象)

初始化：*Queue.Queue(maxsize=0)*

- *maxsize* :队列的存放对象最大数量设定，0代表无限大

基本方法：

- *qsize*:当前队列中的对象数
- *empty*:队列是否为空
- *full*:队列是否满，队列满的状态会是*put*报错
- *put*:参数为，对象，*block*布尔值，*timeout*等待时间，*block*为真时，队列为满时，*put*会等待
- *get*:从队列中取出对象，先进先出，队列为空时报错。*block*参数为真时，当队列为空时等待，*timeout*参数设置等待时间

threading模块

使用队列同步化线程实例：

代码：

```
import time
import threading
import Queue

dataQuene = Queue.Queue(maxsize=0) # python 3.X: queue.Queue

def thread_func():
    data = dataQuene.get(block=True)
    print('thread_1:' + data)
thread_1 = threading.Thread(target=thread_func)
thread_1.start()

time.sleep(3)
print('main sleep end')
dataQuene.put('put from main')
```

结果：

```
main sleep end
thread_1:put from main
[Finished in 3.138s]
```

threading模块

threading.Event对象:(threading.Event方法返回)

代码：

```
import time
import threading
event = threading.Event()
print(event.is_set())
def thread_func():
    print('thread_1 is waitting')
    event.wait()
    print('thread_1 going')
    for i in range(4):
        time.sleep(0.5)
        print('thread_1 is running')
    print('thread_1 is waitting again')
    event.wait()
    print('thread_1 end')
thread_1 = threading.Thread(target=thread_func)
thread_1.start()
time.sleep(3)
print('let thread_1 go')
event.set()
event.clear()
time.sleep(3)
print('let thread_1 go secondly')
event.set()
```

结果：

```
False
thread_1 is waitting
let thread_1 go
thread_1 going
thread_1 is running
thread_1 is running
thread_1 is running
thread_1 is running
thread_1 is waitting again
let thread_1 go secondly
[Finished in 6.097s]
```

threading模块

threading.Condition对象:(threading.Condition方法返回,对象是复杂的锁形式，acquire()方法获得锁，wait()方法开始释放锁，wait()方法结束获得锁，release()释放锁，notify/notifyAll不操作锁)

代码：

```
import time
import threading

condition = threading.Condition()
mutex_lock = threading.Lock()
def thread_func():
    condition.acquire()
    print('thread acquire')
    condition.wait()
    print('wait over!!!')
    time.sleep(3)
    condition.release()
    with mutex_lock:
        print('thread end')
thread_1 = threading.Thread(target=thread_func)
thread_2 = threading.Thread(target=thread_func)
thread_1.start()
thread_2.start()
print('let other threads wait')
time.sleep(1)
condition.acquire()
# condition.notify(1)
print('notify!!!')
condition.notifyAll()
print('release main Lock')
condition.release()
```

结果：

```
thread acquire
thread acquire
let other threads wait
notify!!!
release main Lock
wait over!!!
thread end
wait over!!!
thread end
[Finished in 7.161s]
```

进程分支

fork(类unix系统)

守护进程：它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。

终端控制：每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制。

创建守护进程：创建子进程，父进程退出。子进程成为孤儿进程被1号进程收养。

代码：

```
import os
import time

def child():
    while True:
        time.sleep(1)
        print('child PID:%s' % os.getpid())
        print('chile PPID:%s' % os.getppid())

def parent():
    if os.fork() == 0:
        child()
    else:
        time.sleep(6)
        print('parent PID:%s' % (os.getpid()))
parent()
```

结果：

```
child PID:37040
chile PPID:37039
parent PID:37039
child PID:37040
chile PPID:37039
[sanger@inspur PythonTest]$ child PID:37040
chile PPID:1
child PID:37040
chile PPID:1
child PID:37040
chile PPID:1
child PID:37040
```

进程分支

exec组合：从fork分离出来的进程可以用exec方法启动新的程序(exec组合结果命令行参数的第一个是被忽略，anything代替)

代码：

```
import os
os.execlp("python", 'anything','MyProcess\child.py', '-a', 'b', 'c')
# os.execl("/usr/bin/python", 'anything','MyProcess\child.py', '-a', 'b', 'c')
# os.execvp('python', ('anything MyProcess\child.py a b', 'c'))
# os.execv('/usr/bin/python', ('anything MyProcess\child.py a b', 'c'))
print('NOT RUN')
```

拓展：

os.execle
os.execve
os.execlpe
os.execvpe

进程退出

sys,os,shell退出状态

sys.exit(n)：以n为退出状态码退出，实际抛出一个SystemExit异常，可以使用except捕捉，不结束进程。
os._exit(n)：直接结束进程，输出缓冲等将不会处理。

获取shell退出状态：
os.system() # 阻塞，直接返回退出状态码
os.popen() # 不阻塞，返回对象的close()方法阻塞，返回退出状态码
subprocess.check_call # 阻塞，检查退出状态码
subprocess.check_output # 阻塞，检查退出状态码，返回输出(windows上出错)
subprocess.Popen # 返回对象的wait()方法，阻塞，返回退出状态码。或者wait()后，调用对象的returncode属性
os.fork到os.wait的状态共享

进程退出

*os.wait(os.waitpid)*的状态共享

os.wait：等待所有子进程退出;*os.waitpid()*等待特定进程*id*的退出。
*os.wait*返回等待结束的进程*id*和退出状态，退出状态是一个*16*位二进制退出码(与系统进程退出有关)，退出码的低八位代表杀死该进程的信号编号，高八位代表退出状态。

代码：

```
import os
import time
def child():
    print('child pid:{}  exit_status:{}'.format(os.getpid(), 3))
    os._exit(3)

def parent():
    new_pid = os.fork()
    if new_pid == 0:
        child()
    else:
        pid, status = os.wait()
        print('process exit with \'exit\':' + str(os.WIFEXITED(status)))
        print(status >> 8)  # use low 8-bit
        print('parent got pid:{} status:{}'.format(pid, status))
if __name__ == '__main__':parent()
```

结果：

```
child pid:37067  exit_status:3
process exit with 'exit':True
3
parent got pid:37067 status:768
```

进程通信(*IPC*)

简单机制通信：

- 简单文件
- 命令行参数
- 程序退出状态代码
- *shell*环境变量
- 标准流重定向
- *os.popen*和*subprocess*流管道

其他通信工具：

- 套接字
- 匿名管道
- 具名管道
- 共享内存
- 信号

进程通信IPC

匿名管道

代码：

```
import os
import time
pipein, pipeout = os.pipe()
def child():
    line = os.read(pipeout, 2)   # 阻塞
    print('child get:{}'.format(line))

def parent():
    new_pid = os.fork()
    if new_pid == 0:
        child()
    else:
        os.write(pipein, 'msg'.encode())
if __name__ == '__main__':parent()
```

结果：

```
child get:ms
```

进程通信IPC

匿名管道

代码：

```
import os
import time
pipein, pipeout = os.pipe()
def child():
    line = os.read(pipein, 2)
    print('child get:{}'.format(line))
    os.write(pipeout, 'efg'.encode())


def parent():
    new_pid = os.fork()
    if new_pid == 0:
        child()
    else:
        os.write(pipeout, 'abc'.encode())
        time.sleep(0.5)
        line_c = os.read(pipein, 2)
        print('parent get:{}'.format(line_c))
if __name__ == '__main__':parent()
```

结果：

```
child get:ab
parent get:ce
```

进程通信IPC

套接字

server代码：

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.bind(('', 50008))
sock.listen(10)
while True:
    conn, addr = sock.accept()
    data = conn.recv(1024)
    print('server get:{}'.format(data))
    conn.send('server got:[{}]'.format(data).encode())
```

client代码：

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 50008))
sock.send('client info')
reply = sock.recv(1024)
sock.close()
print('client get:{}'.format(reply))
```

结果：

```
server get:client info
```

结果：

```
client get:server got:[client info]
```

进程通信IPC

信号:软件生成的事件，与跨进程的异常类似，但是信号是一种编号，特定的编号来识别，进行特定的行为。

代码：

```
import time, signal, os

def onSignal(a, b):
    print(a)
    print(b)
    print('Got signal at {}'.format(time.ctime(time.time())))
signal.signal(signal.SIGTERM, onSignal)
# signal.signal(15, onSignal)
print('pid:{}'.format(os.getpid()))
signal.pause()
```

KILL：

```
kill   48481
```

结果：

```
pid:48481
15
<frame object at 0x7fc4e5a0e050>
Got signal at Fri Apr 22 17:08:12 2016
```

multiprocessing模块

Process对象：

代码：

```
import os
from multiprocessing import Process

print(str(os.getpid()) +':hello world!')

def foo():
    print(str(os.getpid()) +':hi!')

if __name__ == '__main__':   # windows下必要
    p = Process(target=foo)
    p.start()
```

Windows结果：

```
232:hello world!
6584:hello world!
6584:hi!
```

Linux结果：

```
12709:hello world!
12721:hi!
```

multiprocessing模块

Queue对象：由于类Unix系统和Windows系统的不同

错误代码：说明windows与linux的不同

```
from multiprocessing import Process, Queue
import time, os
q = Queue()
def f():
    print(str(os.getpid()) + ':' + str(q))
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    p = Process(target=f)
    p.start()
    print(str(os.getpid()) + ':' + str(q))
    print q.get()
    p.join()
```

Windows结果：

```
2968:<multiprocessing.queues.Queue object at 0x000000000291E048>
6396:<multiprocessing.queues.Queue object at 0x000000000293E1D0>
# 死锁
```

Linux结果：

```
65452:<multiprocessing.queues.Queue object at 0x7f06637285d0>
65453:<multiprocessing.queues.Queue object at 0x7f06637285d0>
[42, None, 'hello']
```

multiprocessing模块

Queue对象：由于类Unix系统和Windows系统的不同

正确代码：说明windows与linux的不同

```
from multiprocessing import Process, Queue
import time, os
que = Queue()
def f(q):
    print(str(os.getpid()) + ':' +str(q))
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    p = Process(target=f, args=(que, ))
    p.start()
    print(str(os.getpid()) + ' parent:' + str(que))
    p.join()
    print que.get()
```

Windows结果：

```
5004 parent:<multiprocessing.queues.Queue object at 0x00000000029BD080>
6664:<multiprocessing.queues.Queue object at 0x000000000284DFD0>
[42, None, 'hello']
```

Linux结果：

```
23771 parent:<multiprocessing.queues.Queue object at 0x7fca7d26e610>
23782:<multiprocessing.queues.Queue object at 0x7fca7d26e610>
[42, None, 'hello']
```

multiprocessing模块

Pipe管道：

代码：

```
from multiprocessing import Process, Pipe

def f(conn):
    print(conn)
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn, ))
    p.start()
    print('parent:' + str(child_conn))
    print('parent:' + str(parent_conn))
    print parent_conn.recv()
    p.join()
```

Windows结果：

```
parent:<read-write PipeConnection, handle 652>
parent:<read-write PipeConnection, handle 360>
[42, None, 'hello']
<read-write PipeConnection, handle 32>
```

Linux结果：

```
parent:<read-write Connection, handle 7>
parent:<read-write Connection, handle 6>
<read-write Connection, handle 7>
[42, None, 'hello']
```

multiprocessing模块

multiprocessing与threading类似的线/进 程同步化方式：
Lock
Event
Condition
Semaphore
需要注意的是，不共享内存的多进程中各个对象必须作为参数传入子进程，尤其在Windows系统中

multiprocessing模块

内存共享：Value和Array都是线/进 程安全的，实际这两者都是ctypes类型，有更多的ctypes类型供使用。

代码：

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))
    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print num.value
    print arr[:]
```

结果：

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

multiprocessing模块

服务进程-Manager:单独用于多进程对象共享的进程，支持对象list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Queue, Value, Array

代码：

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()
if __name__ == '__main__':
    manager = Manager()
    d = manager.dict()
    l = manager.list(range(10))
    p = Process(target=f, args=(d, l))
    p.start()
    p.join()
    print d
    print l
```

结果：

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[Finished in 0.979s]
```

multiprocessing模块

进程池-Pool：

代码：

```
from multiprocessing import Pool
import time

def f(x):
    time.sleep(1)
    return x**2

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3, 4, 5, 6]))
```

结果：

```
[1, 4, 9, 16, 25, 36]
[Finished in 2.331s]
```

进程池-Pool对象的其他方法：

apply:运行一个进程，阻塞

apply_async:运行一个进程，不阻塞，返回一个multiprocessing.pool.AsyncResult对象，有get，ready，wait，successful方法

map_async:运行多个进程，不阻塞，返回一个multiprocessing.pool.AsyncResult对象，有get，ready，wait，successful方法

imap:返回一个可迭代对象，有next方法，递进计算，阻塞