Python 多线程和多进程编程总结

简介

早已进入多核时代的计算机，怎能不用多线程和多进程进行加速。
我在使用python的过程中，用到过几次多线程和多进程加速，觉得
充分利用CPU节省时间是一种很有“延长生命”的感觉。现将网络上看到的python的
多线程和多进程编程常用的知识点汇总在这里。

线程与进程

线程与进程是操作系统里面的术语，简单来讲，每一个应用程序都有一个自己的进程。
操作系统会为这些进程分配一些执行资源，例如内存空间等。
在进程中，又可以创建一些线程，他们共享这些内存空间，并由操作系统调用，
以便并行计算。

32位系统受限于总线宽度，单个进程最多能够访问的地址空间
只有4G，利用物理地址扩展(PAE)
技术，可以让CPU访问超过4G内存。但是在单个进程还是只能访问4G空间，PAE的优势是可以让不同进程累计使用的内存超过4G。
在个人电脑上，还是建议使用64位系统，便于使用大内存
提升程序的运行性能。

多线程编程

线程的状态

创建线程之后，线程并不是始终保持一个状态。其状态大概如下：

- New 创建。
- Runnable 就绪。等待调度
- Running 运行。
- Blocked 阻塞。阻塞可能在 Wait Locked Sleeping
- Dead 消亡

线程的类型

线程有着不同的状态，也有不同的类型。大致可分为：

- 主线程
- 子线程
- 守护线程（后台线程）
- 前台线程

python的GIL

GIL即全局解释器锁，它使得python的多线程无法充分利用
多核的优势，但是对于I/O操作频繁的爬虫之类的程序，
利用多线程带来的优势还是很明显的。
如果要利用多核优势，还是用多进程吧。

创建线程

Python提供两个模块进行多线程的操作，分别是 `thread` 和 `threading` ，
前者是比较低级的模块，用于更底层的操作，一般应用级别的开发不常用。

第一种方法是创建 `threading.Thread` 的子类，重写 `run` 方法。

```
import time
import threading

class MyThread(threading.Thread):
    def run(self):
        for i in range(5):
            print 'thread {}, @number: {}'.format(self.name, i)
            time.sleep(1)

def main():
    print "Start main threading"
    # 创建三个线程
    threads = [MyThread() for i in range(3)]
    # 启动三个线程
    for t in threads:
        t.start()

    print "End Main threading"


if __name__ == '__main__':
    main()
```

输入如下：（不同的环境不一样）

```
Start main threading
thread Thread-1, @number: 0
thread Thread-2, @number: 0
thread Thread-3, @number: 0
End Main threading
thread Thread-1, @number: 1
thread Thread-3, @number: 1
thread Thread-2, @number: 1
```

```
thread Thread-3, @number: 2
thread Thread-1, @number: 2
 thread Thread-2, @number: 2
thread Thread-2, @number: 3
thread Thread-1, @number: 3
thread Thread-3, @number: 3
```

线程合并（join方法）

主线程结束后，子线程还在运行， join 方法使得主线程等到子线程结束
时才退出。

```
def main():
    print "Start main threading"

    threads = [MyThread() for i in range(3)]

    for t in threads:
        t.start()

    # 一次让新创建的线程执行 join
    for t in threads:
        t.join()

    print "End Main threading"
```

线程同步与互斥锁

为了避免线程不同步造成是数据不同步，可以对资源进行加锁。
也就是访问资源的线程需要获得锁，才能访问。
threading 模块正好提供了一个Lock功能

```
mutex = threading.Lock()
```

在线程中获取锁

```
mutex.acquire()
```

使用完后，释放锁

```
mutex.release()
```

可重入锁

为了支持在同一线程中多次请求同一资源，
python提供了可重入锁（RLock）。
RLock内部维护着一个 Lock 和一个 counter 变量，
counter 记录了acquire的次数，从而使得资源可以被多次require。
直到一个线程所有的acquire都被release，其他的线程才能获得资源。

创建RLock

```
mutex = threading.RLock()
```

线程内多次进入锁和释放锁

```
class MyThread(threading.Thread):

    def run(self):
        if mutex.acquire(1):
            print "thread {} get mutex".format(self.name)
            time.sleep(1)
            mutex.acquire()
```

```
        mutex.release()
        mutex.release()
```

条件变量

实用锁可以达到线程同步，前面的互斥锁就是这种机制。更复杂的环境，需要针对锁进行一些条件判断。Python提供了Condition对象。它除了具有acquire和release方法之外，还提供了wait和notify方法。线程首先acquire一个条件变量锁。如果条件不足，则该线程wait，如果满足就执行线程，甚至可以notify其他线程。其他处于wait状态的线程接到通知后会重新判断条件。

条件变量可以看成不同的线程先后acquire获得锁，如果不满足条件，可以理解为被扔到一个（Lock或RLock）的waiting池。直达其他线程notify之后再重新判断条件。该模式常用于生成消费者模式：

```
queue = []

con = threading.Condition()

class Producer(threading.Thread):
    def run(self):
        while True:
            if con.acquire():
                if len(queue) > 100:
                    con.wait()
                else:
                    elem = random.randrange(100)
                    queue.append(elem)
                    print "Producer a elem {}, Now size is {}".format(elem, len(queue))
                    time.sleep(random.random())
                    con.notify()
                con.release()

class Consumer(threading.Thread):
    def run(self):
```

```
        while True:
            if con.acquire():
                if len(queue) < 0:
                    con.wait()
                else:
                    elem = queue.pop()
                    print "Consumer a elem {}. Now size is {}".format(elem, len(queue))
                    time.sleep(random.random())
                    con.notify()
                con.release()

def main():
    for i in range(3):
        Producer().start()

    for i in range(2):
        Consumer().start()
```

队列

带锁的队列 Queue 。
创建10个元素的队列

```
queue = Queue.Queue(10)
```

队列通过 put 加入元素，通过 get 方法获取元素。

线程通信

线程可以读取共享的内存，通过内存做一些数据处理。
这就是线程通信的一种，python还提供了更加高级的线程通信接口。
Event对象可以用来进行线程通信，调用event对象的wait方法，
线程则会阻塞等待，直到别的线程set之后，才会被唤醒。

```
class MyThread(threading.Thread):
    def __init__(self, event):
        super(MyThread, self).__init__()
        self.event = event

    def run(self):
        print "thread {} is ready ".format(self.name)
        self.event.wait()
        print "thread {} run".format(self.name)

signal = threading.Event()

def main():
    start = time.time()
    for i in range(3):
        t = MyThread(signal)
        t.start()
    time.sleep(3)
    print "after {}s".format(time.time() - start)
    signal.set()
```

后台线程

默认情况下，主线程退出之后，即使子线程没有join。那么主线程结束后，
子线程也依然会继续执行。如果希望主线程退出后，
其子线程也退出而不再执行，则需要设置子线程为后台线程。python提供了 setDeamon 方法。

进程

python中的多线程其实并不是真正的多线程，如果想要充分地使用多核CPU的资源，在python中大部分情况需要使用多进程。Python提供了非常好用的多进程包multiprocessing，只需要定义一个函数，Python会完成其他所有事情。借助这个包，可以轻松完成从单进程到并发执行的转换。multiprocessing支持子进程、通信和共享数据、执行不同形式的同步，提供了Process、Queue、Pipe、Lock等组件。

类Process

- 创建进程的类： Process([group [, target [, name [, args [, kwargs]]]]]) ，
  target表示调用对象，args表示调用对象的位置参数元组。
  kwargs表示调用对象的字典。name为别名。group实质上不使用。
- 方法：is_alive()、join([timeout])、run()、start()、terminate()。其中，Process以start()启动某个进程。
- 属性：authkey、daemon（要通过start()设置）、exitcode(进程在运行时为None、如果为–N，表示被信号N结束）、name、pid。其中daemon是父进程终止后自动终止，且自己不能产生新进程，必须在start()之前设置。

例：创建函数并将其作为单个进程

```
import multiprocessing
import time

def worker(interval):
    n = 5
    while n > 0:
        print("The time is {0}".format(time.ctime()))
        time.sleep(interval)
        n -= 1

if __name__ == "__main__":
    p = multiprocessing.Process(target = worker, args = (3,))
    p.start()
    print "p.pid:", p.pid
    print "p.name:", p.name
    print "p.is_alive:", p.is_alive()
```

结果

```
p.pid: 8736
p.name: Process-1
```

```
p.is_alive: True
The time is Tue Apr 21 20:55:12 2015
The time is Tue Apr 21 20:55:15 2015
The time is Tue Apr 21 20:55:18 2015
The time is Tue Apr 21 20:55:21 2015
The time is Tue Apr 21 20:55:24 2015
```

例：创建函数并将其作为多个进程

```
import multiprocessing
import time

def worker_1(interval):
    print "worker_1"
    time.sleep(interval)
    print "end worker_1"

def worker_2(interval):
    print "worker_2"
    time.sleep(interval)
    print "end worker_2"

def worker_3(interval):
    print "worker_3"
    time.sleep(interval)
    print "end worker_3"

if __name__ == "__main__":
    p1 = multiprocessing.Process(target = worker_1, args = (2,))
    p2 = multiprocessing.Process(target = worker_2, args = (3,))
    p3 = multiprocessing.Process(target = worker_3, args = (4,))

    p1.start()
```

```
    p2.start()
    p3.start()

    print("The number of CPU is:" + str(multiprocessing.cpu_count()))
    for p in multiprocessing.active_children():
        print("child   p.name:" + p.name + "\tp.id" + str(p.pid))
    print "END!!!!!!!!!!!!!!!!!!"
```

结果

```
The number of CPU is:4
child   p.name:Process-3    p.id7992
child   p.name:Process-2    p.id4204
child   p.name:Process-1    p.id6380
END!!!!!!!!!!!!!!!!!!
worker_1
worker_3
worker_2
end worker_1
end worker_2
end worker_3
```

例：将进程定义为类

```
import multiprocessing
import time

class ClockProcess(multiprocessing.Process):
    def __init__(self, interval):
        multiprocessing.Process.__init__(self)
        self.interval = interval

    def run(self):
```

```
        n = 5
        while n > 0:
            print("the time is {0}".format(time.ctime()))
            time.sleep(self.interval)
            n -= 1

if __name__ == '__main__':
    p = ClockProcess(3)
    p.start()
```

注：进程p调用start()时，自动调用run()

结果

```
the time is Tue Apr 21 20:31:30 2015
the time is Tue Apr 21 20:31:33 2015
the time is Tue Apr 21 20:31:36 2015
the time is Tue Apr 21 20:31:39 2015
the time is Tue Apr 21 20:31:42 2015
```

例：daemon程序对比结果

不加daemon属性

```
import multiprocessing
import time

def worker(interval):
    print("work start:{0}".format(time.ctime()));
    time.sleep(interval)
    print("work end:{0}".format(time.ctime()));

if __name__ == "__main__":
```

```
    p = multiprocessing.Process(target = worker, args = (3,))
    p.start()
    print "end!"
```

结果

```
end!
work start:Tue Apr 21 21:29:10 2015
work end:Tue Apr 21 21:29:13 2015
```

加上daemon属性

```
import multiprocessing
import time

def worker(interval):
    print("work start:{0}".format(time.ctime()));
    time.sleep(interval)
    print("work end:{0}".format(time.ctime()));

if __name__ == "__main__":
    p = multiprocessing.Process(target = worker, args = (3,))
    p.daemon = True
    p.start()
    print "end!"
```

结果

```
end!
```

注：因子进程设置了daemon属性，主进程结束，它们就随着结束了。

设置daemon执行完结束的方法

```
import multiprocessing
import time

def worker(interval):
    print("work start:{0}".format(time.ctime()));
    time.sleep(interval)
    print("work end:{0}".format(time.ctime()));

if __name__ == "__main__":
    p = multiprocessing.Process(target = worker, args = (3,))
    p.daemon = True
    p.start()
    p.join()
    print "end!"
```

结果

```
work start:Tue Apr 21 22:16:32 2015
work end:Tue Apr 21 22:16:35 2015
end!
```

Lock

当多个进程需要访问共享资源的时候，Lock可以用来避免访问的冲突。

```
import multiprocessing
import sys

def worker_with(lock, f):
    with lock:
```

```
        fs = open(f, 'a+')
        n = 10
        while n > 1:
            fs.write("Lockd acquired via with\n")
            n -= 1
        fs.close()

def worker_no_with(lock, f):
    lock.acquire()
    try:
        fs = open(f, 'a+')
        n = 10
        while n > 1:
            fs.write("Lock acquired directly\n")
            n -= 1
        fs.close()
    finally:
        lock.release()

if __name__ == "__main__":
    lock = multiprocessing.Lock()
    f = "file.txt"
    w = multiprocessing.Process(target = worker_with, args=(lock, f))
    nw = multiprocessing.Process(target = worker_no_with, args=(lock, f))
    w.start()
    nw.start()
    print "end"
```

结果（输出文件）

```
Lockd acquired via with
Lockd acquired via with
Lockd acquired via with
```

```
Lockd acquired via with
Lockd acquired via with
Lockd acquired via with
Lockd acquired via with
Lockd acquired via with
Lockd acquired via with
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
Lock acquired directly
```

Semaphore

Semaphore用来控制对共享资源的访问数量，例如池的最大连接数。

```
import multiprocessing
import time

def worker(s, i):
    s.acquire()
    print(multiprocessing.current_process().name + "acquire");
    time.sleep(i)
    print(multiprocessing.current_process().name + "release\n");
    s.release()

if __name__ == "__main__":
    s = multiprocessing.Semaphore(2)
    for i in range(5):
```

```
    p = multiprocessing.Process(target = worker, args=(s, i*2))
    p.start()
```

结果

```
Process-1acquire
Process-1release

Process-2acquire
Process-3acquire
Process-2release

Process-5acquire
Process-3release

Process-4acquire
Process-5release

Process-4release
```

Event

Event用来实现进程间同步通信。

```
import multiprocessing
import time

def wait_for_event(e):
    print("wait_for_event: starting")
    e.wait()
    print("wairt_for_event: e.is_set()->" + str(e.is_set()))
```

```
def wait_for_event_timeout(e, t):
    print("wait_for_event_timeout:starting")
    e.wait(t)
    print("wait_for_event_timeout:e.is_set->" + str(e.is_set()))

if __name__ == "__main__":
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name = "block",
            target = wait_for_event,
            args = (e,))

    w2 = multiprocessing.Process(name = "non-block",
            target = wait_for_event_timeout,
            args = (e, 2))
    w1.start()
    w2.start()

    time.sleep(3)

    e.set()
    print("main: event is set")
```

结果

```
wait_for_event: starting
wait_for_event_timeout:starting
wait_for_event_timeout:e.is_set->False
main: event is set
wairt_for_event: e.is_set()->True
```

Queue

Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。put方法用以插入数据到队列中，put方法还有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间。如果超时，会抛出Queue.Full异常。如果blocked为False，但该Queue已满，会立即抛出Queue.Full异常。

get方法可以从队列读取并且删除一个元素。同样，get方法有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，那么在等待时间内没有取到任何元素，会抛出Queue.Empty异常。如果blocked为False，有两种情况存在，如果Queue有一个值可用，则立即返回该值，否则，如果队列为空，则立即抛出Queue.Empty异常。Queue的一段示例代码：

```
import multiprocessing

def writer_proc(q):
    try:
        q.put(1, block = False)
    except:
        pass

def reader_proc(q):
    try:
        print q.get(block = False)
    except:
        pass

if __name__ == "__main__":
    q = multiprocessing.Queue()
    writer = multiprocessing.Process(target=writer_proc, args=(q,))
    writer.start()

    reader = multiprocessing.Process(target=reader_proc, args=(q,))
    reader.start()
```

```
    reader.join()
    writer.join()
```

结果

```
1
```

Pipe

Pipe方法返回(conn1, conn2)代表一个管道的两个端。Pipe方法有duplex参数，如果duplex参数为True(默认值)，那么这个管道是全双工模式，也就是说conn1和conn2均可收发。duplex为False，conn1只负责接受消息，conn2只负责发送消息。

send和recv方法分别是发送和接受消息的方法。例如，在全双工模式下，可以调用conn1.send发送消息，conn1.recv接收消息。如果没有消息可接收，recv方法会一直阻塞。如果管道已经被关闭，那么recv方法会抛出EOFError。

```
import multiprocessing
import time

def proc1(pipe):
    while True:
        for i in xrange(10000):
            print "send: %s" %(i)
            pipe.send(i)
            time.sleep(1)

def proc2(pipe):
    while True:
        print "proc2 rev:", pipe.recv()
        time.sleep(1)
```

```
def proc3(pipe):
    while True:
        print "PROC3 rev:", pipe.recv()
        time.sleep(1)

if __name__ == "__main__":
    pipe = multiprocessing.Pipe()
    p1 = multiprocessing.Process(target=proc1, args=(pipe[0],))
    p2 = multiprocessing.Process(target=proc2, args=(pipe[1],))
    #p3 = multiprocessing.Process(target=proc3, args=(pipe[1],))

    p1.start()
    p2.start()
    #p3.start()

    p1.join()
    p2.join()
    #p3.join()
```

结果

```
```

Pool

在利用Python进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用multiprocessing中的Process动态成生多个进程，十几个还好，但如果是上百个，上千个目标，手动的去限制进程数量却又太过繁琐，此时可以发挥进程池的功效。
Pool可以提供指定数量的进程，供用户调用，当有新的请求提交到pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来它。

例：使用进程池

```
#coding: utf-8
import multiprocessing
import time

def func(msg):
    print "msg:", msg
    time.sleep(3)
    print "end"

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes = 3)
    for i in xrange(4):
        msg = "hello %d" %(i)
        pool.apply_async(func, (msg, ))   #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去

    print "Mark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~~"
    pool.close()
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束
    print "Sub-process(es) done."
```

一次执行结果

```
mMsg: hark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~~ello 0

msg: hello 1
msg: hello 2
end
msg: hello 3
end
end
```

```
end
Sub-process(es) done.
```

函数解释：

apply_async(func[, args[, kwds[, callback]]]) 它是非阻塞，apply(func[, args[, kwds]])是阻塞的（理解区别，看例1例2结果区别）
close()　关闭pool，使其不在接受新的任务。
terminate()　结束工作进程，不在处理未完成的任务。
join()　主进程阻塞，等待子进程的退出， join方法要在close或terminate之后使用。

执行说明：创建一个进程池pool，并设定进程的数量为3，xrange(4)会相继产生四个对象[0, 1, 2, 4]，四个对象被提交到pool中，因pool指定进程数为3，所以0、1、2会直接送到进程中执行，当其中一个执行完事后才空出一个进程处理对象3，所以会出现输出“msg: hello 3”出现在”end”后。因为为非阻塞，主函数会自己执行自个的，不搭理进程的执行，所以运行完for循环后直接输出“mMsg: hark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~~”，主程序在pool.join（）处等待各个进程的结束。

例：使用进程池（阻塞）

```
#coding: utf-8
import multiprocessing
import time

def func(msg):
    print "msg:", msg
    time.sleep(3)
    print "end"

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes = 3)
    for i in xrange(4):
        msg = "hello %d" %(i)
        pool.apply(func, (msg, ))   #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进
```

```
去

    print "Mark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~"
    pool.close()
    pool.join()  #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束
    print "Sub-process(es) done."
```

一次执行的结果

```
msg: hello 0
end
msg: hello 1
end
msg: hello 2
end
msg: hello 3
end
Mark~ Mark~ Mark~~~~~~~~~~~~~~~~~~~~~
Sub-process(es) done.
```

例：使用进程池，并关注结果

```
msg: hello 0
msg: hello 1
msg: hello 2
end
end
end
::: donehello 0
::: donehello 1
::: donehello 2
Sub-process(es) done.
```

例：使用多个进程池

```
#coding: utf-8
import multiprocessing
import os, time, random

def Lee():
    print "\nRun task Lee-%s" %(os.getpid()) #os.getpid()获取当前的进程的ID
    start = time.time()
    time.sleep(random.random() * 10) #random.random()随机生成0-1之间的小数
    end = time.time()
    print 'Task Lee, runs %0.2f seconds.' %(end - start)

def Marlon():
    print "\nRun task Marlon-%s" %(os.getpid())
    start = time.time()
    time.sleep(random.random() * 40)
    end=time.time()
    print 'Task Marlon runs %0.2f seconds.' %(end - start)

def Allen():
    print "\nRun task Allen-%s" %(os.getpid())
    start = time.time()
    time.sleep(random.random() * 30)
    end = time.time()
    print 'Task Allen runs %0.2f seconds.' %(end - start)

def Frank():
    print "\nRun task Frank-%s" %(os.getpid())
    start = time.time()
    time.sleep(random.random() * 20)
    end = time.time()
```

```
    print 'Task Frank runs %0.2f seconds.' %(end - start)

if __name__=='__main__':
    function_list= [Lee, Marlon, Allen, Frank]
    print "parent process %s" %(os.getpid())

    pool=multiprocessing.Pool(4)
    for func in function_list:
        pool.apply_async(func)    #Pool执行函数，apply执行函数,当有一个进程执行完毕后，会添加一个新的进程到pool中

    print 'Waiting for all subprocesses done...'
    pool.close()
    pool.join()    #调用join之前，一定要先调用close() 函数，否则会出错, close()执行后不会有新的进程加入到pool,join函数等待素有子进程结束
    print 'All subprocesses done.'
```

一次执行结果

```
parent process 7704

Waiting for all subprocesses done...
Run task Lee-6948

Run task Marlon-2896

Run task Allen-7304

Run task Frank-3052
Task Lee, runs 1.59 seconds.
Task Marlon runs 8.48 seconds.
Task Frank runs 15.68 seconds.
```

```
Task Allen runs 18.08 seconds.
All subprocesses done.
```

资料来源

1. http://www.cnblogs.com/kaituorensheng/p/4445418.html
2. http://python.jobbole.com/85177/

created in 2016-05-31 14:44