云--澈

Python之进程与线程

> PS:我们知道现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持"多任务"的操作系统。多任务的实现共有3种方式：多进程模式；多线程模式；多进程+多线程模式。Python既支持多进程又支持多线程，下面我们将会讨论如何编写这两种多任务程序。

参考原文

廖雪峰Python进程和线程

多进程

为了让Python程序实现**多进程**（multiprocessing），我们先来了解操作系统在这方面的相关知识。

fork

Unix/Linx操作系统提供了一个**fork（）**系统调用，它非常特殊，不同于普通的函数（调用一次，返回一次），fork（）**调用一次，返回两次**。这是因为操作系统自动把当前进程（父进程）**复制**了一份（子进程），然后分别在父进程和子进程中返回。

在子进程中永远返回0，而在父进程中返回子进程的ID。这样做是因为一个父进程可以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。

在Python中的os模块中封装了常见的系统调用，其中就包括了fork，可以在Python程序中轻松创建出子进程：

```
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))
```

结果：

```
Process (4423) start...
I (4423) just created a child process (4424).
I am child process (4424) and my parent is 4423.
```

> 注意:windows没有fork调用。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。

multiprocessing

既然Windows没有fork调用，那怎么在Windows上用Python编写多进程的程序？因为Python是跨平台的，其中的multiprocessing模块就是跨平台版本的多进程模块。

在multiprocessing模块中提供了一个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：

公告

昵称： 云--澈
园龄： 1年8个月
粉丝： 1
关注： 7
+加关注

< 2019年9

日	一	二	三
1	2	3	4
8	9	10	11
15	16	17	18
22	23	24	25
29	30	1	2
6	7	8	9

搜索

随笔分类

Docker(3)
Java Concurrency(2)
JavaSE(3)
Java容器(1)
Linux
PHP(2)
Python Web开发(1)

```
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.')

'''
Parent process 20280.
Child process will start.
Run child process test (5772)...
Child process end.
'''
```

> Tips:用multiprocessing创建子进程时，Process类代表一个进程，只需传入子进程需执行的函数和参数，用start方法启动子进程，join（）方法可以等待子进程结束后再往下运行（通常用于进程间的同步）。

Pool

上面的方法都是启动一个子进程，但是当我们要启动大量的子进程时，怎么办呢？可以用**进程池**的方式批量创建子进程：

按 Ctrl+C 复制代码

```
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 subprocess done...')
    p.close()
    p.join()
    print('All subprocesses done.')

'''
Parent process 896.
Waiting for all subprocess done...
Run task 0 (9728)...
Run task 1 (22216)...
Run task 2 (20572)...
Run task 3 (6844)...
Task 0 runs 0.36 seconds.
Run task 4 (9728)...
Task 4 runs 0.43 seconds.
Task 3 runs 0.94 seconds.
Task 1 runs 1.19 seconds.
Task 2 runs 2.72 seconds.
All subprocesses done.
'''
```

按 Ctrl+C 复制代码

注意apply_async是异步的，就是说子进程执行的同时，主进程继续向下执行。所以“Waiting for all subprocesses done...”先打印出来，close方法意味着不能再添加新的Process了。对Pool对象调用join（）方法，会暂停主进程，等待所有的子进程执行完，所以“All subprocesses done.”最后打印。

> Tips:task4最后执行，是因为Pool的默认大小是4（CPU的核数），所以最多执行4个进程。当然这是Pool有意设计的限制，并不是操作系统的限制，你也可以自己改变它的默认大小，就可以跑不止4个进程。

外部子进程

Python 爬虫(3)
Python基础(19)
Spring(4)
Spring Boot(7)
Spring MVC
Vue(10)
计算机网络知识(6)
面试题(9)
前端(1)
日常问题(1)
数据结构与算法(8)

随笔档案
2019年6月(12)
2019年4月(2)
2019年3月(8)
2019年2月(7)
2019年1月(8)
2018年12月(6)
2018年11月(2)
2018年8月(4)
2018年7月(1)
2018年6月(4)
2018年5月(10)
2018年4月(17)

最新评论
1. Re:Scrapy 组件的

上面的子进程的代码实现都是在主进程内部的，然而很多时候，子进程都是一个外部进程，我们需要控制子进程的输入和输出。

subprocess（可以在当前程序中执行其他程序或命令）模块可以让我们非常方便地启动一个外部子进程，然后**控制其输入和输出**：

```
import subprocess

print('$ nslookup www.python.org')
r = subprocess.call(['nslookup', 'www.python.org'])
print('Exit cod:', r)
'''
$ nslookup www.python.org
服务器:  ns.sc.cninfo.net
Address:  61.139.2.69

非权威应答:
名称:    dualstack.python.map.fastly.net
Addresses:  2a04:4e42:36::223
          151.101.72.223
Aliases:  www.python.org

Exit cod: 0
'''
```

上面的运行效果相当于在命令行直接输入“nslookup www.python.org”（相当于开了一个进程）。

如果子进程还需要通过手动输入一些参数，那么可以通过communicate()方法输入：

```
import subprocess

print('$ nslookup')
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
print(output.decode('gbk'))
print('Exit code:', p.returncode)

'''
$ nslookup
默认服务器:  ns.sc.cninfo.net
Address:  61.139.2.69

> > 服务器:  ns.sc.cninfo.net
Address:  61.139.2.69

python.org      MX preference = 50, mail exchanger = mail.python.org

mail.python.org internet address = 188.166.95.178
mail.python.org AAAA IPv6 address = 2a03:b0c0:2:d0::71:1
>
Exit code: 0
'''
```

上面的代码相当于在命令行直接输入nslookup,然后手动输入：

```
set q=mx
python.org
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（队列）、Pipes（管道）等多种方式来交换数据。下面我们就以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中读数据：

```
from multiprocessing import Process, Queue
import os, time, random
```

@ hello_415600 确实
你的评论，挺好的，爬虫
口，字段这些确实是经常
时候也遇到接口改变的情
么分析页面，提取字段就
↖(^ω^...

2. Re:Scrapy 组件的具

代码对应的字段，貌似有
修改一下 "id":"f3af29
9e99f7d9043bd4", "i
md5":"2b8bac2ff33c

3. Re:Scrapy 组件的具

代码对应的字段，貌似有
修改一下 "id":"f3af29
9e99f7d9043bd4", "i
md5":"2b8bac2ff33c

4. Re:Scrapy 组件的具

代码对应的字段，貌似有
修改一下 "id":"f3af29
9e99f7d9043bd4", "i
md5":"2b8bac2ff33c

5. Re:Scrapy 组件的具

是不是漏掉了。item['t

阅读排行榜

1. Python中使用SQLit
2. java创建对象的四种
3. Python之UDP编程(
4. 用Java写一个生产者
7)
5. JPA 与 JDBC 的区别
7)

评论排行榜

```
#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__':
    #父进程创建出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，写入队列
    pw.start()
    #启动子进程pr，读取队列
    pr.start()
    #等待pw进程结束
    pw.join()
    #pr进程死循环，无法等待，只能强行终止;
    pr.terminate()

'''
Process to write: 8768
Put A to queue...
Process to read: 19700
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
'''
```

在Linux/Unix下，multiprocessing模块分装了fork调用，而由于Windows没有fork调用，因此multiprocessing要想模拟出fork的效果。父进程中的所有Python对象都必须通过pickle序列化到子进程中，因此如果multiprocessing在Windows下调用失败了，要先考虑是不是pickle失败了。

多线程

前面已经说过了多个任务既可以用多进程来实现，又可以用多线程来实现。那么多线程与多进程相比，有什么优点呢？线程共享相同的内存空间，不同的线程可以读取内存中的同一变量(每个进程都有各自独立的空间)。线程带来的开销要比进程小。

由于线程是操作系统直接支持的**执行单元**，因此许多高级语言都内置了多线程的支持，Python也不例外，Python中的线程是真正的**Posix Thread**而不是模拟出来的线程。

要实现多线程，Python的标准库提供了两个模块：_thread和threading,前者是低级模块，后者是高级模块，后者分装了前者。绝大多数情况下，我们只需要使用threading这个高级的模块。

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（）执行：

```
import time, threading

#新线程执行的代码
def loop():
    print('thread %s is runnging...' % 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)
```

1. Scrapy 组件的具体
2. JPQL 的基本使用(1
3. Python之UDP编程(

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

'''thread MainThread is running...
thread LoopTread is runnging...
thread LoopTread >>> 1
thread LoopTread >>> 2
thread LoopTread >>> 3
thread LoopTread >>> 4
thread LoopTread >>> 5
thread LoopTread ended.
Thread MainThread ended.
'''
```

由于任何进程都会默认启动一个线程，我们就把这个线程称为主线程，主线程又可以启动新的线程。上面的current_thread()函数返回当前线程的实例，主线程的名字就MainThread，子线程的名字是在创建时我们指定的。

使用多线程还是有风险的，因为在多线程所有变量被所有线程共享，此时可能会出现多个线程同时改变一个变量，导致出现错误。为了避免这个错误的出现，我们应该加锁lock。

Lock

我们先不使用lock，来看一个错误的实例：

```
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 #共享变量
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for n 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)
```

我们启动了连个线程，先存后取，理论上结果应该为0，但是线程对的调度也是由操作系统决定，所以，当t1 和 t2交替执行，循环次数够多，结果就不一定是0了。因为高级语言的一条语句在CPU执行时是若干条语句。

所以如果我们要保证balance的计算正确，就应该就上一把锁，使该变量同一时刻只能被一个线程操作。在这里我们就可以给change_it()加上一把锁：

```
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，然后继续执行下面的代码，其他线程就只能等待直到或取到锁为止。所以获取到锁的线程在用完后一定要释放锁，否则等待锁开启的线程，将永远等待，所以我们用try...finally来确保锁一定会被释放。

> Tips:锁的坏处就是阻止了多线程的**并发执行**，效率大大地下降了。当不同的线程持有不同的锁，并试图获取对方的锁时，可能会造成死锁。

小结：多线程编程，模型复杂，容易发生冲突，必须加锁以隔离，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，这就是**模拟**出来的**并发**（线程数量大于处理器数量）。

ThreadLocal

我们已经知道多线中变量是可以共享的，在多线程的环境下，每个线程都有自己的数据。那么每一个线程应该也可以拥有自己的**局部变量**，线程使用自己的局部变量比使用全局变量好，因为局部变量只能自己使用，不会影响其他的线程，而使用全局变量的话则必须加锁。

那么具体怎么在Python中使用线程的局部变量呢？那就是使用ThreadLocal，先来看一个例子：

```
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):
    #绑定当前线程关联的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()
'''
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
'''
```

全局变量local_school就是一个ThreadLocal对象，每个线程对她都可以读写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。

ThreaLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求用户信息身份等。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。

> Tip: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

前面我们已经介绍了**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，在多任务环境下，通常是**一个**Master，**多个**Worker。

如果我们用**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。

其中**多进程模式**最大的**优点**就是**稳定性高**，这是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crash了，所有的进程就crash了，但是概率很低毕竟Master进程只负责分配任务），著名的Apache最早采用的就是多进程模式。但是**多进程的缺点**就

是创建进程的**代价大**，在**Unix/Linux**系统下，用fork调用还行，但是在**Windows**下创建进程的**开销巨大**。另外，操作系统能同时运行的进程也是有限的，在CPU和内存的限制下，如果有几千个进程同时运行，那么操作系统连调度都会成问题。

而**多线程模式**通常比多进程模式**快一点**，但也快不到哪去。而且，多线程模式**致命的缺点**就是因为任何一个线程**crash**了都可能造成整个进程crash，因为所有线程**共享进程**的内存。

> Tips： 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的模式。由于多线程存在稳定性问题，IIS的稳定性就不如Apache。但是现在为了平衡，IIS和Apache现在又有了多进程+多线程的混合模式。

什么时候采用多任务呢?

我们需要考虑任务的类型，我们可以把任务分为**计算密集型**和**IO密集型**。

顾名思义，**计算密集型任务**的特点就是要进行**大量的计算**，**消耗大量的CPU资源**，如计算圆周率，对视频进行高清解码等，全靠CPU的运算能力。这种计算密集型任务最好**不要用多任务**完成，因为这样会切换很多次才能执行完，切换任务花费的时间就很长了，就会导致CPU的效率低下。

> Tips:由于计算密集型任务主要消耗CPU资源，因此代码运行效率就非常重要了。因为Python这样的脚本语言运行效率很低，所以对于计算密集型任务，最好用C语言编写。

再来说**IO密集型任务**，涉及到**网络**、**磁盘的IO**任务都是IO密集型任务，特点是**CPU消耗很少**，任务的大部分时间都在等待IO操作的完成。对于IO密集型任务，任务越多，CPU效率越高（但还是有一个限度）。

> Tips:常见的大部分任务都是IO密集型任务，比如Web应用，对于**IO密集型任务**，最适合的语言就是开发效率最高（代码量最少）的语言，所以**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，单进程单线程模式会导致别的任务无法执行，因此我们才需要多进程或多线程的模型来支持多任务并发。**异步文件IO方式中，线程发送一个IO请求到内核，然后继续处理其他的事情，内核完成IO请求后，将会通知线程IO操作完成了。**

如果充分利用操作系统提供的异步IO支持，就可以利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**。使用异步IO编程模型来实现多任务是一个主要的趋势。

在Python中，**单进程单线程**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**程序了。

分布式进程

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上上的多个CPU中。

我们已经知道Python的multiprocessing模块支持**多进程**，其中的managers**子模块**还支持把**多进程分布**到多台机器上。

> 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**服务进程**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?

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：

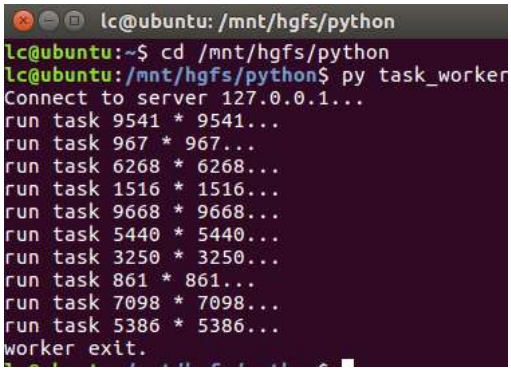
```
import random, time, queue
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = queue.Queue()
# 接收结果的队列:
result_queue = queue.Queue()

# 从BaseManager继承的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='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) #暂停10秒等待分布式进程处理结果并返回
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
print('master exit.')
```

接着在另一台机器上启动**任务进程**也可以是本机:

```
# 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='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.')
```

```
lc@ubuntu:/mnt/hgfs/python$ py task_master.py
Put task 9541...
Put task 967...
Put task 6268...
Put task 1516...
Put task 9668...
Put task 5440...
Put task 3250...
Put task 861...
Put task 7098...
Put task 5386...
Try get results...
Result: 9541 * 9541 = 91030681
Result: 967 * 967 = 935089
Result: 6268 * 6268 = 39287824
Result: 1516 * 1516 = 2298256
Result: 9668 * 9668 = 93470224
Result: 5440 * 5440 = 29593600
Result: 3250 * 3250 = 10562500
Result: 861 * 861 = 741321
Result: 7098 * 7098 = 50381604
Result: 5386 * 5386 = 29008996
master exit.
```



注意：先启动master进程，完成两个队列的网上注册，接着发出请求队列task，等待result队列的结果；此时启动worker进程对task队列进行操作，然后写入到result队列中；master得到响应结果，打印出result。

> Tips:Python的**分布式进程**的接口**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。

不积跬步，无以至千里；不积小流，无以成江海

分类： Python基础

云--澈
关注 - 7
粉丝 - 1
+加关注

« 上一篇： 物理层
» 下一篇： 数据链路层

posted @ 2018-04-27 17:13 云--澈 阅读(140) 评论(0) 编辑 收藏

相关博文：

- Day-12: 进程和线程
- Python进程和线程
- Python笔记_第四篇_高阶编程_进程、线程、协程_3.进程vs线程