Python协程

2017-02-20 | 编程之道 | 2488 ℃

真正有知识的人的成长过程，就像麦穗的成长过程：麦穗空的时候，麦子长得很快，麦穗骄傲地高高昂起，但是，麦穗成熟饱满时，它们开始谦虚，垂下麦芒。

——蒙田《蒙田随笔全集》

上篇论述了关于python多线程是否是鸡肋的问题，得到了一些网友的认可，当然也有一些不同意见，表示协程比多线程不知强多少，在协程面前多线程算是鸡肋。好吧，对此我也表示赞同，然而上篇我论述的观点不在于多线程与协程的比较，而是在于IO密集型程序中，多线程尚有用武之地。

对于协程，我表示其效率确非多线程能比，但本人对此了解并不深入，因此最近几日参考了一些资料，学习整理了一番，在此分享出来仅供大家参考，如有谬误请指正，多谢。

申明：*本文介绍的协程是入门级别，大神请绕道而行，谨防入坑。*

文章思路：本文将先介绍协程的概念，然后分别介绍Python2.x与3.x下协程的用法，最终将协程与多线程做比较并介绍异步爬虫模块。

协程

概念

协程，又称微线程，纤程，英文名Coroutine。协程的作用，是在执行函数A时，可以随时中断，去执行函数B，然后中断继续执行函数A（可以自由切换）。但这一过程并不是函数调用（没有调用语句），这一整个过程看似像多线程，然而协程只有一个线程执行。

优势

- 执行效率极高，因为子程序切换（函数）不是线程切换，由程序自身控制，没有切换线程的开销。所以与多线程相比，线程的数量越多，协程性能的优势越明显。

- 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，在控制共享资源时也不需要加锁，因此执行效率高很多。

说明：协程可以处理IO密集型程序的效率问题，但是处理CPU密集型不是它的长处，如要充分发挥CPU利用率可以结合多进程+协程。

以上只是协程的一些概念，可能听起来比较抽象，那么我结合代码讲一讲吧。这里主要介绍协程在Python的应用，Python2对协程的支持比较有限，生成器的yield实现了一部分但不完全，gevent模块倒是有比较好的实现；Python3.4以后引入了asyncio模块，可以很好的使用协程。

Python2.x协程

python2.x协程应用：

- yield
- gevent

python2.x中支持协程的模块不多，gevent算是比较常用的，这里就简单介绍一下gevent的用法。

Gevent

gevent是第三方库，通过greenlet实现协程，其基本思想：

当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO。

Install

pip install gevent

最新版貌似支持windows了，之前测试好像windows上运行不了.....

Usage

首先来看一个简单的爬虫例子：

```
#! -*- coding:utf-8 -*-

import gevent
from gevent import monkey;monkey.patch_all()
import urllib2

def get_body(i):
        print "start",i
        urllib2.urlopen("http://cn.bing.com")
        print "end",i

tasks=[gevent.spawn(get_body,i) for i in range(3)]
gevent.joinall(tasks)
```

运行结果：

```
start 0
start 1
start 2
end 2
end 0
end 1
```

说明：从结果上来看，执行get_body的顺序应该先是输出"start"，然后执行到urllib2时碰到IO堵塞，则会自动切换运行下一个程序（继续执行get_body输出start），直到urllib2返回结果，再执行end。也就是说，程序没有等待urllib2请求网站返回结果，而是直接先跳过了，等待执行完毕再回来获取返回值。值得一提的是，在此过程中，只有一个线程在执行，因此这与多线程的概念是不一样的。
换成多线程的代码看看:

```
import threading
import urllib2

def get_body(i):
        print "start",i
        urllib2.urlopen("http://cn.bing.com")
        print "end",i
for i in range(3):
        t=threading.Thread(target=get_body,args=(i,))
        t.start()
```

运行结果：

```
start 0
start 1
start 2
end 1
end 2
end 0
```

说明：从结果来看，多线程与协程的效果一样，都是达到了IO阻塞时切换的功能。不同的是，多线程切换的是线程（线程间切换），协程切换的是上下文（可以理解为执行的函数）。而切换线程的开销明显是要大于切换上下文的开销，因此当线程越多，协程的效率就越比多线程的高。（猜想多进程的切换开销应该是最大的）

Gevent使用说明

- monkey可以使一些阻塞的模块变得不阻塞，机制：遇到IO操作则自动切换，手动切换可以用gevent.sleep(0)（将爬虫代码换成这个，效果一样可以达到切换上下文）
- gevent.spawn 启动协程，参数为函数名称，参数名称

- gevent.joinall 停止协程

Python3.x协程

为了测试Python3.x下的协程应用，我在virtualenv下安装了python3.6的环境。

python3.x协程应用：

- asynico + yield from（python3.4）
- asynico + await（python3.5）
- gevent

Python3.4以后引入了asyncio模块，可以很好的支持协程。

asynico

asyncio是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。asyncio的异步操作，需要在coroutine中通过yield from完成。

Usage

例子：（需在python3.4以后版本使用）

```
import asyncio

@asyncio.coroutine
def test(i):
        print("test_1",i)
        r=yield from asyncio.sleep(1)
        print("test_2",i)

loop=asyncio.get_event_loop()
tasks=[test(i) for i in range(5)]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

运行结果：

```
test_1 3
test_1 4
test_1 0
test_1 1
test_1 2
test_2 3
test_2 0
test_2 2
test_2 4
test_2 1
```

说明：从运行结果可以看到，跟gevent达到的效果一样，也是在遇到IO操作时进行切换（所以先输出test_1，等test_1输出完再输出test_2）。但此处我有一点不明，test_1的输出为什么不是按照顺序执行的呢？可以对比gevent的输出结果（希望大神能解答一下）。

asyncio说明

@asyncio.coroutine把一个generator标记为coroutine类型，然后，我们就把这个coroutine扔到EventLoop中执行。

test()会首先打印出test_1，然后，yield from语法可以让我们方便地调用另一个generator。由于asyncio.sleep()也是一个coroutine，所以线程不会等待asyncio.sleep()，而是直接中断并执行下一个消息循环。当asyncio.sleep()返回时，线程就可以从yield from拿到返回值（此处是None），然后接着执行下一行语句。

把asyncio.sleep(1)看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行EventLoop中其他可以执行的coroutine了，因此可以实现并发执行。

asynico/await

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法async和await，可以让coroutine的代码更简洁易读。

请注意，async和await是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

- 把@asyncio.coroutine替换为async；
- 把yield from替换为await。

Usage

例子（python3.5以后版本使用）：

```
1  import asyncio
2
3  async def test(i):
4          print("test_1",i)
5          await asyncio.sleep(1)
6          print("test_2",i)
7
8  loop=asyncio.get_event_loop()
9  tasks=[test(i) for i in range(5)]
10 loop.run_until_complete(asyncio.wait(tasks))
11 loop.close()
```

运行结果与之前一致。

说明：与前一节相比，这里只是把yield from换成了await，@asyncio.coroutine换成了async，其余不变。

gevent

同python2.x用法一样。

协程VS多线程

如果通过以上介绍，你已经明白多线程与协程的不同之处，那么我想测试也就没有必要了。因为当线程越来越多时，多线程主要的开销花费在线程切换上，而协程是在一个线程内切换的，因此开销小很多，这也许就是两者性能的根本差异之处吧。（个人观点）

异步爬虫

也许关心协程的朋友，大部分是用其写爬虫（因为协程能很好的解决IO阻塞问题），然而我发现常用的urllib、requests无法与asyncio结合使用，可能是因为爬虫模块本身是同步的（也可能是我没找到用法）。那么对于异步爬虫的需求，又该怎么使用协程呢？或者说怎么编写异步爬虫？
给出几个我所了解的方案：

- grequests （requests模块的异步化）
- 爬虫模块+gevent（比较推荐这个）
- aiohttp （这个貌似资料不多，目前我也不太会用）
- asyncio内置爬虫功能 （这个也比较难用）

协程池

作用：控制协程数量

```
from bs4 import BeautifulSoup

import requests

import gevent

from gevent import monkey, pool

monkey.patch_all()

jobs = []

links = []

p = pool.Pool(10)

urls = [

    'http://www.google.com',

    # ... another 100 urls

]

def get_links(url):

    r = requests.get(url)

    if r.status_code == 200:
```

```
30
31          soup = BeautifulSoup(r.text)
32
33          links + soup.find_all('a')
34
35
36  for url in urls:
37
38      jobs.append(p.spawn(get_links, url))
39
40  gevent.joinall(jobs)
```

本文没有太多的干货，都是一些自学时的笔记，分享给新手朋友，仅供参考

文章学习通道：

- Python多进程
- Python多线程

本文内容参考来源：廖雪峰python教程，推荐新手学习。

python 协程

‹ Python多线程鸡年不鸡肋 Shadowsocks折腾记 ›

正在载入来必力