## 20-揭秘Python协程

你好，我是景霄。

上一节课的最后，我们留下一个小小的悬念：生成器在 Python 2 中还扮演了一个重要角色，就是用来实现 Python 协程。

那么首先你要明白，什么是协程？

协程是实现并发编程的一种方式。一说并发，你肯定想到了多线程/多进程模型，没错，多线程/多进程，正是解决并发问题的经典模型之一。最初的互联网世界，多线程/多进程在服务器并发中，起到举足轻重的作用。

随着互联网的快速发展，你逐渐遇到了 C10K 瓶颈，也就是同时连接到服务器的客户达到了一万个。于是很多代码跑崩了，进程上下文切换占用了大量的资源，线程也顶不住如此巨大的压力，这时， NGINX 带着事件循环出来拯救世界了。

如果将多进程/多线程类比为起源于唐朝的藩镇割据，那么事件循环，就是宋朝加强的中央集权制。事件循环启动一个统一的调度器，让调度器来决定一个时刻去运行哪个任务，于是省却了多线程中启动线程、管理线程、同步锁等各种开销。同一时期的 NGINX，在高并发下能保持低资源低消耗高性能，相比 Apache 也支持更多的并发连接。

再到后来，出现了一个很有名的名词，叫做回调地狱（callback hell），手撸过 JavaScript 的朋友肯定知道我在说什么。我们大家惊喜地发现，这种工具完美地继承了事件循环的优越性，同时还能提供 async / await 语法糖，解决了执行性和可读性共存的难题。于是，协程逐渐被更多人发现并看好，也有越来越多的人尝试用 Node.js 做起了后端开发。（讲个笑话，JavaScript 是一门编程语言。）

回到我们的 Python。使用生成器，是 Python 2 开头的时代实现协程的老方法了，Python 3.7 提供了新的基于 asyncio 和 async / await 的方法。我们这节课，同样的，跟随时代，抛弃掉不容易理解、也不容易写的旧的基于生成器的方法，直接来讲新方法。

我们先从一个爬虫实例出发，用清晰的讲解思路，带你结合实战来搞懂这个不算特别容易理解的概念。之后，我们再由浅入深，直击协程的核心。

### 从一个爬虫说起

爬虫，就是互联网的蜘蛛，在搜索引擎诞生之时，与其一同来到世上。爬虫每秒钟都会爬取大量的网页，提取关键信息后存储在数据库中，以便日后分析。爬虫有非常简单的 Python 十行代码实现，也有 Google 那样的全球分布式爬虫的上百万行代码，分布在内部上万台服务器上，对全世界的信息进行嗅探。

话不多说，我们先看一个简单的爬虫例子：

```
import time

def crawl_page(url):
    print('crawling {}'.format(url))
    sleep_time = int(url.split('_')[-1])
    time.sleep(sleep_time)
```

```
    print('OK {}'.format(url))

def main(urls):
    for url in urls:
        crawl_page(url)

%time main(['url_1', 'url_2', 'url_3', 'url_4'])

########## 输出 ##########

crawling url_1
OK url_1
crawling url_2
OK url_2
crawling url_3
OK url_3
crawling url_4
OK url_4
Wall time: 10 s
```

（注意：本节的主要目的是协程的基础概念，因此我们简化爬虫的 scrawl_page 函数为休眠数秒，休眠时间取决于 url 最后的那个数字。）

这是一个很简单的爬虫，main() 函数执行时，调取 crawl_page() 函数进行网络通信，经过若干秒等待后收到结果，然后执行下一个。

看起来很简单，但你仔细一算，它也占用了不少时间，五个页面分别用了 1 秒到 4 秒的时间，加起来一共用了 10 秒。这显然效率低下，该怎么优化呢？

于是，一个很简单的思路出现了——我们这种爬取操作，完全可以并发化。我们就来看看使用协程怎么写。

```
import asyncio

async def crawl_page(url):
    print('crawling {}'.format(url))
    sleep_time = int(url.split('_')[-1])
    await asyncio.sleep(sleep_time)
    print('OK {}'.format(url))

async def main(urls):
    for url in urls:
        await crawl_page(url)

%time asyncio.run(main(['url_1', 'url_2', 'url_3', 'url_4']))

########## 输出 ##########

crawling url_1
OK url_1
crawling url_2
OK url_2
crawling url_3
OK url_3
crawling url_4
OK url_4
Wall time: 10 s
```

看到这段代码，你应该发现了，在 Python 3.7 以上版本中，使用协程写异步程序非常简单。

首先来看 import asyncio，这个库包含了大部分我们实现协程所需的魔法工具。

async 修饰词声明异步函数，于是，这里的 crawl_page 和 main 都变成了异步函数。而调用异步函数，我们便可得到一个协程对象（coroutine object）。

举个例子，如果你 `print(crawl_page(''))`，便会输出`<coroutine object crawl_page at 0x000002BEDF141148>`，提示你这是一个 Python 的协程对象，而并不会真正执行这个函数。

再来说说协程的执行。执行协程有多种方法，这里我介绍一下常用的三种。

首先，我们可以通过 await 来调用。await 执行的效果，和 Python 正常执行是一样的，也就是说程序会阻塞在这里，进入被调用的协程函数，执行完毕返回后再继续，而这也是 await 的字面意思。代码中 `await asyncio.sleep(sleep_time)` 会在这里休息若干秒，`await crawl_page(url)` 则会执行 crawl_page() 函数。

其次，我们可以通过 asyncio.create_task() 来创建任务，这个我们下节课会详细讲一下，你先简单知道即可。

最后，我们需要 asyncio.run 来触发运行。asyncio.run 这个函数是 Python 3.7 之后才有的特性，可以让 Python 的协程接口变得非常简单，你不用去理会事件循环怎么定义和怎么使用的问题（我们会在下面讲）。一个非常好的编程规范是，asyncio.run(main()) 作为主程序的入口函数，在程序运行周期内，只调用一次 asyncio.run。

这样，你就大概看懂了协程是怎么用的吧。不妨试着跑一下代码，欸，怎么还是 10 秒？

10 秒就对了，还记得上面所说的，await 是同步调用，因此， crawl_page(url) 在当前的调用结束之前，是不会触发下一次调用的。于是，这个代码效果就和上面完全一样了，相当于我们用异步接口写了个同步代码。

现在又该怎么办呢？

其实很简单，也正是我接下来要讲的协程中的一个重要概念，任务（Task）。老规矩，先看代码。

```
import asyncio

async def crawl_page(url):
    print('crawling {}'.format(url))
    sleep_time = int(url.split('_')[-1])
    await asyncio.sleep(sleep_time)
    print('OK {}'.format(url))

async def main(urls):
    tasks = [asyncio.create_task(crawl_page(url)) for url in urls]
    for task in tasks:
```

```
        await task

%time asyncio.run(main(['url_1', 'url_2', 'url_3', 'url_4']))

########## 输出 ##########

crawling url_1
crawling url_2
crawling url_3
crawling url_4
OK url_1
OK url_2
OK url_3
OK url_4
Wall time: 3.99 s
```

你可以看到，我们有了协程对象后，便可以通过 asyncio.create_task 来创建任务。任务创建后很快就会被调度执行，这样，我们的代码也不会阻塞在任务这里。所以，我们要等所有任务都结束才行，用for task in tasks: await task 即可。

这次，你就看到效果了吧，结果显示，运行总时长等于运行时间最长的爬虫。

当然，你也可以想一想，这里用多线程应该怎么写？而如果需要爬取的页面有上万个又该怎么办呢？再对比下协程的写法，谁更清晰自是一目了然。

其实，对于执行 tasks，还有另一种做法：

```
import asyncio

async def crawl_page(url):
    print('crawling {}'.format(url))
    sleep_time = int(url.split('_')[-1])
    await asyncio.sleep(sleep_time)
    print('OK {}'.format(url))

async def main(urls):
    tasks = [asyncio.create_task(crawl_page(url)) for url in urls]
    await asyncio.gather(*tasks)

%time asyncio.run(main(['url_1', 'url_2', 'url_3', 'url_4']))

########## 输出 ##########

crawling url_1
crawling url_2
crawling url_3
crawling url_4
OK url_1
OK url_2
OK url_3
OK url_4
Wall time: 4.01 s
```

这里的代码也很好理解。唯一要注意的是，*tasks 解包列表，将列表变成了函数的参数；与之对应的是，** dict 将字典变成了函数的参数。

另外，asyncio.create_task，asyncio.run 这些函数都是 Python 3.7 以上的版本才提供的，自然，相比于旧接口它们也更容易理解和阅读。

### 解密协程运行时

说了这么多，现在，我们不妨来深入代码底层看看。有了前面的知识做基础，你应该很容易理解这两段代码。

```
import asyncio

async def worker_1():
    print('worker_1 start')
    await asyncio.sleep(1)
    print('worker_1 done')

async def worker_2():
    print('worker_2 start')
    await asyncio.sleep(2)
    print('worker_2 done')

async def main():
    print('before await')
    await worker_1()
    print('awaited worker_1')
    await worker_2()
    print('awaited worker_2')

%time asyncio.run(main())

########## 输出 ##########

before await
worker_1 start
worker_1 done
awaited worker_1
worker_2 start
worker_2 done
awaited worker_2
Wall time: 3 s
```

```
import asyncio

async def worker_1():
    print('worker_1 start')
    await asyncio.sleep(1)
    print('worker_1 done')

async def worker_2():
    print('worker_2 start')
    await asyncio.sleep(2)
    print('worker_2 done')
```

```
async def main():
    task1 = asyncio.create_task(worker_1())
    task2 = asyncio.create_task(worker_2())
    print('before await')
    await task1
    print('awaited worker_1')
    await task2
    print('awaited worker_2')

%time asyncio.run(main())

########## 输出 ##########

before await
worker_1 start
worker_2 start
worker_1 done
awaited worker_1
worker_2 done
awaited worker_2
Wall time: 2.01 s
```

不过，第二个代码，到底发生了什么呢？为了让你更详细了解到协程和线程的具体区别，这里我详细地分析了整个过程。步骤有点多，别着急，我们慢慢来看。

1. `asyncio.run(main())`，程序进入 main() 函数，事件循环开启；
2. task1 和 task2 任务被创建，并进入事件循环等待运行；运行到 print，输出 `'before await'`；
3. await task1 执行，用户选择从当前的主任务中切出，事件调度器开始调度 worker_1；
4. worker_1 开始运行，运行 print 输出`'worker_1 start'`，然后运行到 `await asyncio.sleep(1)`， 从当前任务切出，事件调度器开始调度 worker_2；
5. worker_2 开始运行，运行 print 输出 `'worker_2 start'`，然后运行 `await asyncio.sleep(2)` 从当前任务切出；
6. 以上所有事件的运行时间，都应该在 1ms 到 10ms 之间，甚至可能更短，事件调度器从这个时候开始暂停调度；
7. 一秒钟后，worker_1 的 sleep 完成，事件调度器将控制权重新传给 task_1，输出 `'worker_1 done'`，task_1 完成任务，从事件循环中退出；
8. await task1 完成，事件调度器将控制器传给主任务，输出 `'awaited worker_1'`，·然后在 await task2 处继续等待；
9. 两秒钟后，worker_2 的 sleep 完成，事件调度器将控制权重新传给 task_2，输出 `'worker_2 done'`，task_2 完成任务，从事件循环中退出；
10. 主任务输出 `'awaited worker_2'`，协程全任务结束，事件循环结束。

接下来，我们进阶一下。如果我们想给某些协程任务限定运行时间，一旦超时就取消，又该怎么做呢？再进一步，如果某些协程运行时出现错误，又该怎么处理呢？同样的，来看代码。

```
import asyncio

async def worker_1():
    await asyncio.sleep(1)
    return 1

async def worker_2():
```

```
    await asyncio.sleep(2)
    return 2 / 0

async def worker_3():
    await asyncio.sleep(3)
    return 3

async def main():
    task_1 = asyncio.create_task(worker_1())
    task_2 = asyncio.create_task(worker_2())
    task_3 = asyncio.create_task(worker_3())

    await asyncio.sleep(2)
    task_3.cancel()

    res = await asyncio.gather(task_1, task_2, task_3, return_exceptions=True)
    print(res)

%time asyncio.run(main())

########## 输出 ##########

[1, ZeroDivisionError('division by zero'), CancelledError()]
Wall time: 2 s
```

你可以看到，worker_1 正常运行，worker_2 运行中出现错误，worker_3 执行时间过长被我们 cancel 掉了，这些信息会全部体现在最终的返回结果 res 中。

不过要注意`return_exceptions=True`这行代码。如果不设置这个参数，错误就会完整地 throw 到我们这个执行层，从而需要 try except 来捕捉，这也就意味着其他还没被执行的任务会被全部取消掉。为了避免这个局面，我们将 return_exceptions 设置为 True 即可。

到这里，发现了没，线程能实现的，协程都能做到。那就让我们温习一下这些知识点，用协程来实现一个经典的生产者消费者模型吧。

```
import asyncio
import random

async def consumer(queue, id):
    while True:
        val = await queue.get()
        print('{} get a val: {}'.format(id, val))
        await asyncio.sleep(1)

async def producer(queue, id):
    for i in range(5):
        val = random.randint(1, 10)
        await queue.put(val)
        print('{} put a val: {}'.format(id, val))
        await asyncio.sleep(1)

async def main():
    queue = asyncio.Queue()

    consumer_1 = asyncio.create_task(consumer(queue, 'consumer_1'))
    consumer_2 = asyncio.create_task(consumer(queue, 'consumer_2'))
```

```
    producer_1 = asyncio.create_task(producer(queue, 'producer_1'))
    producer_2 = asyncio.create_task(producer(queue, 'producer_2'))

    await asyncio.sleep(10)
    consumer_1.cancel()
    consumer_2.cancel()

    await asyncio.gather(consumer_1, consumer_2, producer_1, producer_2, return_exceptions=True)

%time asyncio.run(main())

########## 输出 ##########

producer_1 put a val: 5
producer_2 put a val: 3
consumer_1 get a val: 5
consumer_2 get a val: 3
producer_1 put a val: 1
producer_2 put a val: 3
consumer_2 get a val: 1
consumer_1 get a val: 3
producer_1 put a val: 6
producer_2 put a val: 10
consumer_1 get a val: 6
consumer_2 get a val: 10
producer_1 put a val: 4
producer_2 put a val: 5
consumer_2 get a val: 4
consumer_1 get a val: 5
producer_1 put a val: 2
producer_2 put a val: 8
consumer_1 get a val: 2
consumer_2 get a val: 8
Wall time: 10 s
```

### 实战：豆瓣近日推荐电影爬虫

最后，进入今天的实战环节——实现一个完整的协程爬虫。

任务描述：https://movie.douban.com/cinema/later/beijing/ 这个页面描述了北京最近上映的电影，你能否通过 Python 得到这些电影的名称、上映时间和海报呢？这个页面的海报是缩小版的，我希望你能从具体的电影描述页面中抓取到海报。

听起来难度不是很大吧？我在下面给出了同步版本的代码和协程版本的代码，通过运行时间和代码写法的对比，希望你能对协程有更深的了解。（注意：为了突出重点、简化代码，这里我省略了异常处理。）

不过，在参考我给出的代码之前，你是不是可以自己先动手写一下、跑一下呢？

```
import requests
from bs4 import BeautifulSoup

def main():
    url = "https://movie.douban.com/cinema/later/beijing/"
    init_page = requests.get(url).content
    init_soup = BeautifulSoup(init_page, 'lxml')
```

```
    all_movies = init_soup.find('div', id="showing-soon")
    for each_movie in all_movies.find_all('div', class_="item"):
        all_a_tag = each_movie.find_all('a')
        all_li_tag = each_movie.find_all('li')

        movie_name = all_a_tag[1].text
        url_to_fetch = all_a_tag[1]['href']
        movie_date = all_li_tag[0].text

        response_item = requests.get(url_to_fetch).content
        soup_item = BeautifulSoup(response_item, 'lxml')
        img_tag = soup_item.find('img')

        print('{} {} {}'.format(movie_name, movie_date, img_tag['src']))

%time main()

########## 输出 ##########

阿拉丁 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2553992741.jpg
龙珠超：布罗利 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2557371503.jpg
五月天人生无限公司 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2554324453.jpg
... ...
直播攻略 06月04日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2555957974.jpg
Wall time: 56.6 s
```

```
import asyncio
import aiohttp

from bs4 import BeautifulSoup

async def fetch_content(url):
    async with aiohttp.ClientSession(
        headers=header, connector=aiohttp.TCPConnector(ssl=False)
    ) as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    url = "https://movie.douban.com/cinema/later/beijing/"
    init_page = await fetch_content(url)
    init_soup = BeautifulSoup(init_page, 'lxml')

    movie_names, urls_to_fetch, movie_dates = [], [], []

    all_movies = init_soup.find('div', id="showing-soon")
    for each_movie in all_movies.find_all('div', class_="item"):
        all_a_tag = each_movie.find_all('a')
        all_li_tag = each_movie.find_all('li')

        movie_names.append(all_a_tag[1].text)
        urls_to_fetch.append(all_a_tag[1]['href'])
        movie_dates.append(all_li_tag[0].text)

    tasks = [fetch_content(url) for url in urls_to_fetch]
    pages = await asyncio.gather(*tasks)

    for movie_name, movie_date, page in zip(movie_names, movie_dates, pages):
        soup_item = BeautifulSoup(page, 'lxml')
        img_tag = soup_item.find('img')
```

```
        print('{} {} {}'.format(movie_name, movie_date, img_tag['src']))

%time asyncio.run(main())

########## 输出 ##########

阿拉丁 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2553992741.jpg
龙珠超：布罗利 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2557371503.jpg
五月天人生无限公司 05月24日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2554324453.jpg
... ...
直播攻略 06月04日 https://img3.doubanio.com/view/photo/s_ratio_poster/public/p2555957974.jpg
Wall time: 4.98 s
```

## 总结

到这里，今天的主要内容就讲完了。今天我用了较长的篇幅，从一个简单的爬虫开始，到一个真正的爬虫结束，在中间穿插讲解了 Python 协程最新的基本概念和用法。这里带你简单复习一下。

- 协程和多线程的区别，主要在于两点，一是协程为单线程；二是协程由用户决定，在哪些地方交出控制权，切换到下一个任务。
- 协程的写法更加简洁清晰，把async / await 语法和 create_task 结合来用，对于中小级别的并发需求已经毫无压力。
- 写协程程序的时候，你的脑海中要有清晰的事件循环概念，知道程序在什么时候需要暂停、等待 I/O，什么时候需要一并执行到底。

最后的最后，请一定不要轻易炫技。多线程模型也一定有其优点，一个真正牛逼的程序员，应该懂得，在什么时候用什么模型能达到工程上的最优，而不是自觉某个技术非常牛逼，所有项目创造条件也要上。技术是工程，而工程则是时间、资源、人力等纷繁复杂的事情的折衷。

## 思考题

最后给你留一个思考题。协程怎么实现回调函数呢？欢迎留言和我讨论，也欢迎你把这篇文章分享给你的同事朋友，我们一起交流，一起进步。


**精选留言：**

- KaitoShy 2019-06-25 06:16:24
  思考题，实现回调，不知道是否正确：

```
import asyncio

async def sleep():
print('func sleep is start')
await asyncio.sleep(1)
print('func sleep is over')
return 'stop'

async def continue_sleep():
print('i wanna to sleep again')
await asyncio.sleep(2)

def callback_test(future):
print('Callback: {}'.format(future.result()))

async def main():
task_1 = asyncio.create_task(sleep())
task_2 = asyncio.create_task(continue_sleep())

done, pending = await asyncio.wait({task_1, task_2})
if task_1 in done:
task_1.add_done_callback(callback_test)

asyncio.run(main())
==========返回==========
func sleep is start
i wanna to sleep again
func sleep is over
```

Callback: stop

- 汪zZ 2019-06-24 22:37:38
RuntimeError: asyncio.run() cannot be called from a running event loop
一直报错，是因为我python是3.7.0吗？

- ©HJ 2019-06-24 22:10:08
%time 在windows里会报错

- 长青 2019-06-24 20:33:32
老。。。师。。。。以后能不能把下面大篇幅的讲解直接写到代码里面 这样看起来对我们小白来说会更明了。。。。。多谢！！！

- 随风の 2019-06-24 18:12:16
关于%time运行不了的 大家可以试试
%%time
main(['url_1', 'url_2', 'url_3', 'url_4'])
相当于两个time.time()计算执行时间

- 程序员人生 2019-06-24 16:44:57
关于思考题，这样写不知道对不对？
import asyncio

async def computer(a,b,func):
res = func(a,b)
print("res:{}".format(res))
return res

def max(a,b):
print("max")
return [a,b][a < b]

def min(a,b):
print("min")
return [a,b][a > b]

async def main():
task1 = asyncio.create_task(computer(1, 2, max))
task2 = asyncio.create_task(computer(4, 3, min))

await asyncio.gather(task1,task2,return_exceptions=True)

asyncio.run(main())

执行结果：
max
res:2
min
res:3

- 佛系学python 2019-06-24 16:42:44
  老师，你那代码我一个都运行不出来。。%time那里会报错

- 佛系学python 2019-06-24 16:29:24
  老师，课程的代码是基于py3的吗？

- enjoylearning 2019-06-24 13:04:06
  并发一直想用future executor 来着，不知道这两个有什么区别没

- csn 2019-06-24 12:39:11
  为什么在spyder下运行协程报错，
  提示“ File "C:\ProgramData\Anaconda3\lib\asyncio\runners.py", line 34, in run
  "asyncio.run() cannot be called from a running event loop")

  RuntimeError: asyncio.run() cannot be called from a running event loop”

  在pycharm下执行正常

- catshitfive 2019-06-24 11:47:18
  用adaconda升级了到python version 3.7.3,发现新的api asyncio.run()在调用的时候报错：asyncio.run() cannot be called from a running event loop,在网上看了下解决方法(不清楚为什么这么做):pip thirdparty模块:nest-asyncio,然后调用其模块内函数apply就可以正常使用了

- tt 2019-06-24 11:41:14
  学习笔记:异步和阻塞。

  阻塞主要是同步编程中的概念:执行一个系统调用，如果暂时没有返回结果，这个调用就不会返回，那这个系统调用后面的应用代码也不会执行，整个应用被“阻塞”了。

  同步编程也可以有非阻塞的方式，在系统调用没有完成时直接返回一个错误码。

  异步调用是系统调用完成后返回应用一个消息，应用响应消息，获取结果。

  上面说的系统调用 对应Python中的异步函数——一个需要执行较长时间的任务。响应消息应该对应回调函数，但我觉得await异步函数返回本身就相当于系统告知应用系统调用完成了，后面的代码起码可以完成部分回调函数做的事情。

  阻塞和异步必须配合才能完成异步编程。

  1、await异步函数会造成阻塞;
  2、await一个task不会造成阻塞，但是task对应的异步函数中必定几乎await一个异步函数，这个异步函数会阻塞这个task的执行，这样其余task才能被事件调度器的调度从而获取执行机会。

- 阿西吧 2019-06-24 10:17:58
  这个代码是不是只有在linux环境才能运行成功

- 阿西吧 2019-06-24 10:17:42
  %time 是什么语法糖，直接输出时间，要怎么写？

- 阿西吧 2019-06-24 10:13:32
  协程的第一个例子，一运行就报错：
  RuntimeError: asyncio.run() cannot be called from a running event loop

  win7 64位 python3.7.3

- cotter 2019-06-24 09:43:57
  受教了，第一次听说这个高级功能！
  我在工作中遇到一个需要并发的问题，用python在后台并发执行shell ,并发数量用时间范围控制，要不停的改时间分多次串行，方法比较笨拙。协程可以简化我的代码。
  老师，并发很多事件应该也是需要消耗很多资源，协程改如何控制并发数量？

- 李 2019-06-24 09:30:21
  代码中我敲的time那一块一直报invalid syntax？
  %time asyncio.run

- zx钟 zz 2019-06-24 09:07:03
  官方文档看了好久 还是一脸懵逼。

- Hoo-Ah 2019-06-24 09:02:21
  使用asyncio获取事件循环，将执行的函数使用loop创建一个任务。add_done_callback将回掉函数传进去。

- 敏杰 2019-06-24 08:21:37
  创建loop事件循环asyncio.get_event_loop