

Python 中的异步编程：Asyncio

2017/07/16 • 基础知识 • 协程, 异步

分享到: 0

原文出处: MARIA YAKIMOVA 译文出处: 开源翻译

如果你已经决定要理解 Python 的异步部分, 欢迎来到我们的 “Asyncio How-to”。

注: 哪怕连 **异动范式** 的存在都不知道的情况下, 你也可以成功地使用 Python。但是, 如果你对底层运行模式感兴趣的话, asyncio 绝对值得查看。

异步是怎么一回事?

在传统的顺序编程中, 所有发送给解释器的指令会一条条被执行。此类代码的输出容易显现和预测。但是...

譬如说你有一个脚本向 3 个不同服务器请求数据。有时, 谁知什么原因, 发送给其中一个服务器的请求可能意外地执行了很长时间。想象一下从第二个服务器获取数据用了 10 秒钟。在你等待的时候, 整个脚本实际上什么也没干。如果你可以写一个脚本可以不去等待第二个请求而是仅仅跳过它, 然后开始执行第三个请求, 然后回到第二个请求, 执行之前离开的位置会怎么样呢。就是这样。你通过切换任务最小化了空转时间。尽管如此, **当你需要一个几乎没有 I/O 的简单脚本时, 你不想用异步代码。**

还有一件重要的事情要提, 所有代码在一个线程中运行。**所以如果你想让程序的一部分在后台执行同时干一些其他事情, 那是不可能的。**

准备开始

这是 asyncio 主概念最基本的定义:

是一个生成器, 但这个生成器只消费数据, 不生成数据

- **协程** — 消费数据的 **生成器**, **但是不生成数据**。Python 2.5 介绍了一种新的语法让发送数据到生成器成为可能。我推荐查阅 David Beazley “A Curious Course on Coroutines and Concurrency” 关于协程的详细介绍。
- **任务** — **协程调度器**。如果你观察下面的代码, 你会发现它只是让 event_loop **尽快** 调用它的 _step, 同时 **_step 只是调用协程的下一步。**

三个概念: 协程、任务、事件循环

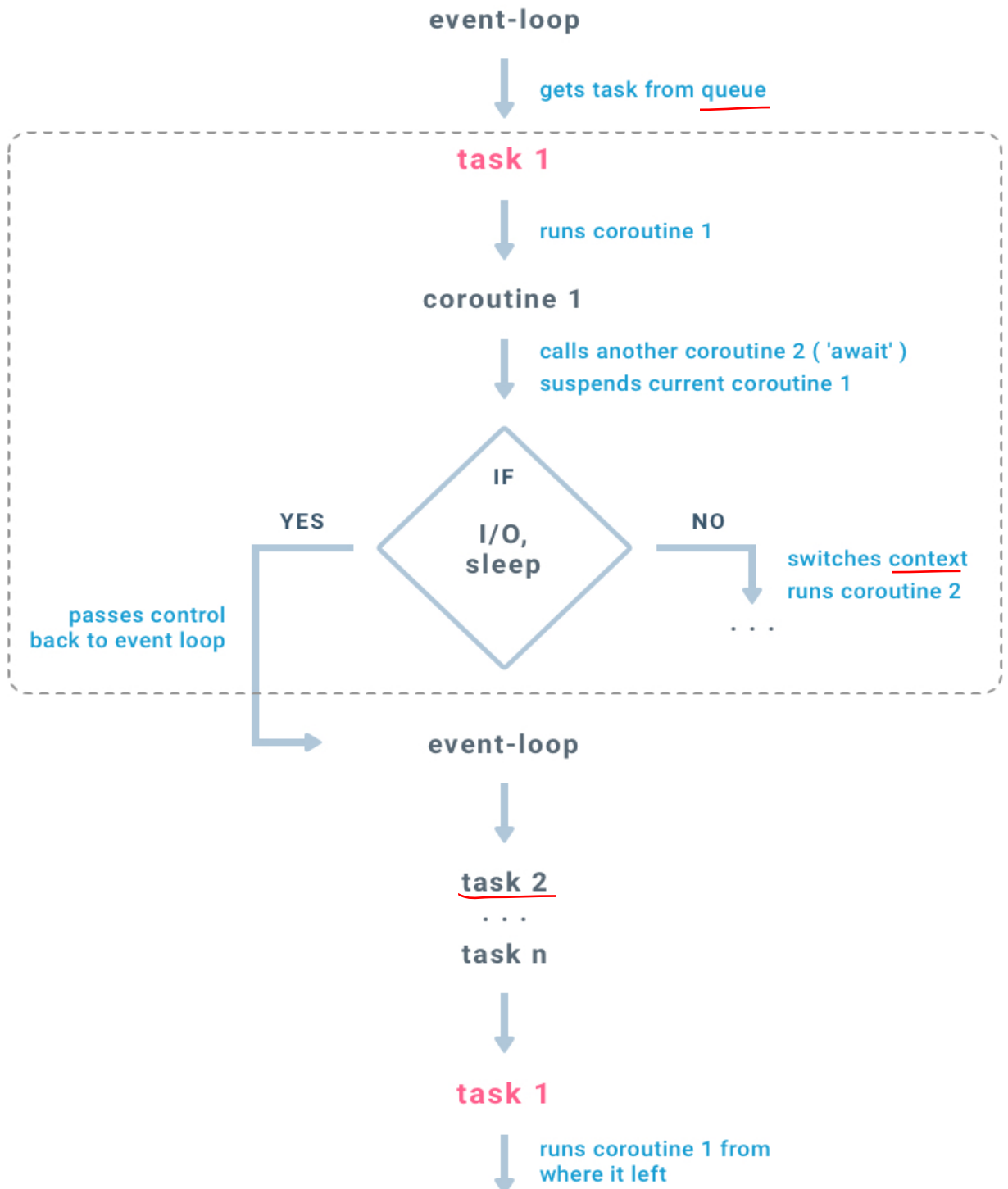
```
class Task(futures.Future):
    def __init__(self, coro, loop=None):
        super().__init__(loop=loop)
    ...
```

```
1 class Task(futures.Future):
2     def __init__(self, coro, loop=None):
3         super().__init__(loop=loop)
4         ...
5         self._loop.call_soon(self._step)
6
7     def _step(self):
8         ...
9         try:
10             ...
11             result = next(self._coro)
12         except StopIteration as exc:
13             self.set_result(exc.value)
14         except BaseException as exc:
15             self.set_exception(exc)
16             raise
17         else:
18             ...
19             self._loop.call_soon(self._step)
```

- 事件循环 — 把它想成 `asyncio` 的中心执行器。

现在我们看一下所有这些如何融为一体。正如我之前提到的，异步代码在一个线程中运行。

Thread



从上图可知：

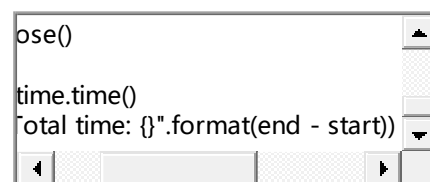
1. 消息循环是在线程中执行
2. 从队列中取得任务
3. 每个任务在协程中执行下一步动作
4. 如果在一个协程中调用另一个协程（`await <coroutine_name>`），会触发上下文切换，挂起当前协程，并保存现场环境（变量，状态），然后载入被调用协程
两种协程挂起
5. 如果协程的执行到阻塞部分（阻塞 I/O, Sleep），当前协程会挂起，并将控制权返回到线程的消息循环中，然后消息循环继续从队列中执行下一个任务。... 以此类推
6. 队列中的所有任务执行完毕后，消息循环返回第一个任务

异步和同步的代码对比

现在我们实际验证异步模式的切实有效，我会比较两段 python 脚本，这两个脚本除了 `sleep` 方法外，其余部分完全相同。在第一个脚本里，我会用标准的 `time.sleep` 方法，在第二个脚本里使用 `asyncio.sleep` 的异步方法。

这里使用 `Sleep` 是因为它是一个用来展示异步方法如何操作 I/O 的最简单办法。

使用同步 `sleep` 方法的代码：



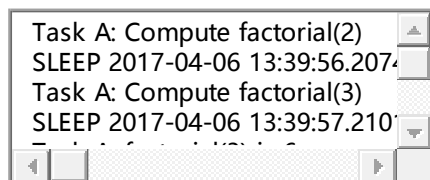
```
1 import asyncio
2 import time
3 from datetime import datetime
4
5
6 async def custom_sleep():
7     print('SLEEP', datetime.now())
8     time.sleep(1)
9
10 async def factorial(name, number):
11     f = 1
12     for i in range(2, number+1):
13         print('Task {}: Compute factorial({})'.format(name, i))
14         await custom_sleep()
15         f *= i
16     print('Task {}: factorial({}) is {}'.format(name, number, f))
```

```

17
18
19 start = time.time()
20 loop = asyncio.get_event_loop()
21
22 tasks = [
23     asyncio.ensure_future(factorial("A", 3)),
24     asyncio.ensure_future(factorial("B", 4)),
25 ]
26 loop.run_until_complete(asyncio.wait(tasks))
27 loop.close()
28
29 end = time.time()
30 print("Total time: {}".format(end - start))

```

脚本输出：



```

Task A: Compute factorial(2)
SLEEP 2017-04-06 13:39:56.207479
Task A: Compute factorial(3)
SLEEP 2017-04-06 13:39:57.210128

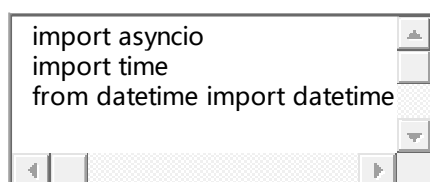
```

```

1 Task A: Compute factorial(2)
2 SLEEP 2017-04-06 13:39:56.207479
3 Task A: Compute factorial(3)
4 SLEEP 2017-04-06 13:39:57.210128
5 Task A: factorial(3) is 6
6
7 Task B: Compute factorial(2)
8 SLEEP 2017-04-06 13:39:58.210778
9 Task B: Compute factorial(3)
10 SLEEP 2017-04-06 13:39:59.212510
11 Task B: Compute factorial(4)
12 SLEEP 2017-04-06 13:40:00.217308
13 Task B: factorial(4) is 24
14
15 Total time: 5.016386032104492

```

使用异步 Sleep 的代码：



```

import asyncio
import time
from datetime import datetime

```

```

1 import asyncio

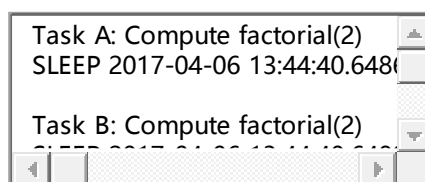
```

```

2 import time
3 from datetime import datetime
4
5
6 async def custom_sleep():
7     print('SLEEP {}'.format(datetime.now()))
8     await asyncio.sleep(1)
9
10 async def factorial(name, number):
11     f = 1
12     for i in range(2, number+1):
13         print('Task {}: Compute factorial({})'.format(name, i))
14         await custom_sleep()
15         f *= i
16     print('Task {}: factorial({}) is {}'.format(name, number, f))
17
18
19 start = time.time()
20 loop = asyncio.get_event_loop()
21
22 tasks = [
23     asyncio.ensure_future(factorial("A", 3)),
24     asyncio.ensure_future(factorial("B", 4)),
25 ]
26 loop.run_until_complete(asyncio.wait(tasks))
27 loop.close()
28
29 end = time.time()
30 print("Total time: {}".format(end - start))

```

脚本输出：



```

1 Task A: Compute factorial(2)
2 SLEEP 2017-04-06 13:44:40.648665
3
4 Task B: Compute factorial(2)
5 SLEEP 2017-04-06 13:44:40.648859
6
7 Task A: Compute factorial(3)
8 SLEEP 2017-04-06 13:44:41.649564

```

9
10 Task B: Compute factorial(3)
11 SLEEP 2017-04-06 13:44:41.649943
12
13 Task A: factorial(3) is 6
14
15 Task B: Compute factorial(4)
16 SLEEP 2017-04-06 13:44:42.651755
17
18 Task B: factorial(4) is 24
19
20 Total time: 3.008226156234741

从输出可以看到，异步模式的代码执行速度快了大概两秒。当使用异步模式的时候（每次调用 `await asyncio.sleep(1)` ），进程控制权会返回到主程序的消息循环里，并开始运行队列的其他任务（任务 A 或者任务 B）。

当使用标准的 `sleep` 方法时，当前线程会挂起等待。什么也不会做。实际上，标准的 `sleep` 过程中，当前线程也会返回一个 python 的解释器，可以操作现有的其他线程，但这是另一个话题了。

推荐使用异步模式编程的几个理由

很多公司的产品都广泛的使用了异步模式，如 Facebook 旗下著名的 React Native 和 RocksDB 。像 Twitter 每天可以承载 50 亿的用户访问，靠的也是异步模式编程。所以说，通过代码重构，或者改变模式方法，就能让系统工作的更快，为什么不去试一下呢？

1 赞 5 收藏 [评论](#)

用 Python 3 的 `async / await` 做异步编程

- Gevent 拾遗
- Gevent 调度流程解析
- Python Enhanced Generator — Coroutine
- Greenlet 详解