[译] Python 3.5 协程究竟是个啥

Yusheng | Wed 09 March 2016

| Updated on Wed 09 March 2016

> * *原文链接 : How the heck does async/await work in Python 3.5?*
> * *原文作者 : Brett Cannon*
> * *译文出自 : 掘金翻译计划*
> * *译者 : @Yushneng*
> * *校对者: @L9m，@iThreeKing*

作者是 Python 语言的核心开发人员，作者在这篇文章里先是是从 Python 异步编程的发展历史一直介绍到 Python 3.5 中 `async` / `await` 新特性的提出，又从底层的实现的差异一直延伸到完整的代码实例，来说明旧的生成器作为协程的“权宜之计”与新语法的差别。真正做到了深入浅出地厘清了 Python 3.5 异步编程的各种概念及其原理，非常值得学

找出地座指于 Python 3.5 并发编程的各种概念及其原理，非常值得学习！协程和事件循环无论从理念还是实现上，都比线程好太多，看完这篇文章之后你一定会和作者有同样的感受：“我要在所有地方都用协程！”，所以，还在用 2.7 的，赶紧升级吧！

以下为译文全文，感谢@L9m，@iThreeKing的审校。

作为 Python 核心开发者之一，让我很想了解这门语言是如何运作的。我发现总有一些阴暗的角落我对其中错综复杂的细节不是很清楚，但是为了能够有助于 Python 的一些问题和其整体设计，我觉得我应该试着去理解 Python 的核心语法和内部运作机制。

但是直到最近我才理解 Python 3.5 中 `async` / `await` 的原理。我知道 Python 3.3 中的 `yield from` 和 Python 3.4 中的 `asyncio` 组合得来这一新语法。但较少处理网络相关的问题 - `asyncio` 并不仅限于此但确是重要用途 - 使我没太注意 `async` / `await` 。我知道：

```
yield from iterator
```

（本质上）相当于：

```
for x in iterator:
    yield x
```

我知道 `asyncio` 是事件循环框架可以进行异步编程，但是我只是知道这里面每个单词的意思而已，从没深入研究 `async` / `await` 语法组合背后的原理，我发现不理解 Python 中的异步编程已经对我造成了困扰。因此我决定花时间弄清楚这背后的原理究竟是什么。我从很多人那里得知他们也不了解异步编程的原理，因此我决定写这篇论文（是的，由于这篇文章花费时间之久以及篇幅之长，我的妻子已经将其定义为一篇论文）。

由于我想要正确地理解这些语法的原理，这篇文章涉及到一些关于 CPython 较为底层的技术细节。如果这些细节超出了你想了解的内容，或者你不能完全理解它们，都没关系，因为我为了避免这篇文章演变成一本书那么长，省略了一些 CPython 内部的细枝末节（比如说，如果你不知道 code object 有 flags，甚至不知道什么是 code object，这都没关系，也不用一定要从这篇文字中获得什么）。我试着在最后一小节中用更直接的方法做了总结，如果觉得文章对你来说细节太多，你完全可以跳过。

关于 Python 协程的历史课

根据维基百科给出的定义，“协程 是为非抢占式多任务产生子程序的计算机程序组件，协程允许不同入口点在不同位置暂停或开始执行程序”。从技术的角度来说，“协程就是你可以暂停执行的函数”。如果你把它理解成“就像生成器一样”，那么你就想对了。

退回到 Python 2.2，生成器第一次在PEP 255中提出（那时也把它成为迭代器，因为它实现了迭代器协议）。主要是受到Icon编程语言的启发，

生成器允许创建一个在计算下一个值时不会浪费内存空间的迭代器。例如你想要自己实现一个 `range()` 函数，你可以用立即计算的方式创建一

如你想要自己实现一个 `range()` 函数，你可以用立即计算的方式创建一个整数列表：

```
def eager_range(up_to):
    """Create a list of integers, from 0 to up_to, exclusive."""
    sequence = []
    index = 0
    while index < up_to:
        sequence.append(index)
        index += 1
    return sequence
```

然而这里存在的问题是，如果你想创建从0到1,000,000这样一个很大的序列，你不得不创建能容纳1,000,000个整数的列表。但是当加入了生成器之后，你可以不用创建完整的序列，你只需要能够每次保存一个整数的内存即可。

```
def lazy_range(up_to):
    """Generator to return the sequence of integers from 0 to u
    index = 0
    while index < up_to:
        yield index
        index += 1
```

让函数遇到 `yield` 表达式时暂停执行 - 虽然直到 Python 2.5 才成为声明语句 - 并且能够在后面重新执行，这对于减少内存使用、生成无限序列非常有用。

你有可能已经发现，生成器完全就是关于迭代器的。有一种更好的方式生成迭代器当然很好（尤其是当你可以给一个生成器对象添加

生成迭代器当然很好（完美是当你可以给一个生成器对象添加 `__iter__()` 方法时），但是人们知道，如果可以利用生成器“暂停”的部分，添加“将东西发送回生成器”的功能，那么 Python 突然就有了协程的概念（当然这里的协程仅限于 Python 中的概念；Python 中真实的协程在后面才会讨论）。将东西发送回暂停了的生成器这一特性通过 PEP 342添加到了 Python 2.5。与其它特性一起，PEP 342 为生成器引入了 `send()` 方法。这让我们不仅可以暂停生成器，而且能够传递值到生成器暂停的地方。还是以我们的 `range()` 为例，你可以让序列向前或向后跳过几个值：

```
def jumping_range(up_to):
    """Generator for the sequence of integers from 0 to up_to,

    Sending a value into the generator will shift the sequence
    """
    index = 0
    while index < up_to:
        jump = yield index
        if jump is None:
            jump = 1
        index += jump

if __name__ == '__main__':
    iterator = jumping_range(5)
    print(next(iterator))  # 0
    print(iterator.send(2))  # 2
    print(next(iterator))  # 3
    print(iterator.send(-1))  # 2
    for x in iterator:
        print(x)  # 3, 4
```

直到PEP 380 为 Python 3.3 添加了 `yield from` 之前，生成器都没有变动。严格来说，这一特性让你能够从迭代器（生成器刚好也是迭代器）

动。广告来说，这一特性让你能够从迭代器（生成器刚好也是迭代器）中返回任何值，从而可以干净利索的方式重构生成器。

```
def lazy_range(up_to):
    """Generator to return the sequence of integers from 0 to u
    index = 0
    def gratuitous_refactor():
        while index < up_to:
            yield index
            index += 1
    yield from gratuitous_refactor()
```

`yield from` 通过让重构变得简单，也让你能够将生成器串联起来，使返回值可以在调用栈中上下浮动，而不需对编码进行过多改动。

```
def bottom():
    # Returning the yield lets the value that goes up the call
    # down.
    return (yield 42)

def middle():
    return (yield from bottom())

def top():
    return (yield from middle())

# Get the generator.
gen = top()
value = next(gen)
print(value)  # Prints '42'.
try:
    value = gen.send(value * 2)
except StopIteration as exc:
```

```
except StopIteration as exc:
    value = exc.value
print(value)  # Prints '84'.
```

总结

Python 2.2 中的生成器让代码执行过程可以暂停。Python 2.5 中可以将值返回给暂停的生成器，这使得 Python 中协程的概念成为可能。加上 Python 3.3 中的 `yield from`，使得重构生成器与将它们串联起来都很简单。

什么是事件循环?

如果你想了解 `async` / `await`，那么理解什么是事件循环以及它是如何让异步编程变为可能就相当重要了。如果你曾做过 GUI 编程 - 包括网页前端工作 - 那么你已经和事件循环打过交道。但是由于异步编程的概念作为 Python 语言结构的一部分还是最近才有的事，你刚好不知道什么是事件循环也很正常。

回到维基百科，事件循环“是一种等待程序分配事件或消息的编程架构”。基本上来说事件循环就是，“当A发生时，执行B”。或许最简单的例子来解释这一概念就是用每个浏览器中都存在的JavaScript事件循环。当你点击了某个东西（“当A发生时”），这一点击动作会发送给JavaScript的事件循环，并检查是否存在注册过的 `onclick` 回调来处理这一点击（“执行B”）。只要有注册过的回调函数就会伴随点击动作的细节信息被执行。事件循环被认为是一种循环是因为它不停地收集事件并通过循环来发如何应对这些事件。

对 Python 来说，用来提供事件循环的 `asyncio` 被加入标准库中。`asyncio` 重点解决网络服务中的问题。事件循环在这里将来自套接

中。`asyncio` 重点解决网络服务中的问题，事件循环在这里将不再套接字（socket）的 I/O 已经准备好读和/或写作为“当A发生时”（通过 `selectors` 模块）。除了 GUI 和 I/O，事件循环也经常用于在别的线程或子进程中执行代码，并将事件循环作为调节机制（例如，合作式多任务）。如果你恰好理解 Python 的 GIL，事件循环对于需要释放 GIL 的地方很有用。

总结

事件循环提供一种循环机制，让你可以“在A发生时，执行B”。基本上来说事件循环就是监听当有什么发生时，同时事件循环也关心这件事并执行相应的代码。Python 3.4 以后通过标准库 `asyncio` 获得了事件循环的特性。

`async` 和 `await` 是如何运作的

Python 3.4 中的方式

在 Python 3.3 中出现的生成器与之后以 `asyncio` 的形式出现的事件循环之间，Python 3.4 通过并发编程的形式已经对异步编程有了足够的支持。_异步编程_简单来说就是代码执行的顺序在程序运行前是未知的（因此才称为异步而非同步）。_并发编程_是代码的执行不依赖于其他部分，即便是全都在同一个线程内执行（并发不是并行）。例如，下面 Python 3.4 的代码分别以异步和并发的函数调用实现按秒倒计时。

```
import asyncio

# Borrowed from http://curio.readthedocs.org/en/latest/tutoria

@asyncio.coroutine
def countdown(number, n):
```

```
def countdown(number, n):
    while n > 0:
        print('T-minus', n, '({})'.format(number))
        yield from asyncio.sleep(1)
        n -= 1

loop = asyncio.get_event_loop()
tasks = [
    asyncio.ensure_future(countdown("A", 2)),
    asyncio.ensure_future(countdown("B", 3))]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

Python 3.4 中，`asyncio.coroutine` 修饰器用来标记作为协程的函数，这里的协程是和 `asyncio` 及其事件循环一起使用的。这赋予了 Python 第一个对于协程的明确定义：实现了PEP 342添加到生成器中的这一方法的对象，并通过[`collections.abc.Coroutine` 这一抽象基类]表征的对象。这意味着突然之间所有实现了协程接口的生成器，即便它们并不是要以协程方式应用，都符合这一定义。为了修正这一点， `asyncio` 要求所有要用作协程的生成器必须由 `asyncio.coroutine` 修饰。

有了对协程明确的定义（能够匹配生成器所提供的API），你可以对任何 `asyncio.Future` 对象使用 `yield from` ，从而将其传递给事件循环，暂停协程的执行来等待某些事情的发生（ future 对象并不重要，只是 `asyncio` 细节的实现）。一旦 future 对象获取了事件循环，它会一直在那里监听，直到完成它需要做的一切。当 future 完成自己的任务之后，事件循环会察觉到，暂停并等待在那里的协程会通过 `send()` 方法获取future对象的返回值并开始继续执行。

以上面的代码为例。事件循环启动每一个 `countdown()` 协程，一直执行到遇见其中一个协程的 `yield from` 和 `asyncio.sleep()` 。这样会返回一个 `asyncio.Future` 对象并将其传递给事件循环，同时暂停这一协程的执行。事件循环会监控这一future对象，直到倒计时1秒钟之后（同时也会

行。事件循环会监控这个future对象，直到倒计时的1秒钟之后（同时也会检查其它正在监控的对象，比如像其它协程）。1秒钟的时间一到，事件循环会选择刚刚传递了future对象并暂停了的 `countdown()` 协程，将future对象的结果返回给协程，然后协程可以继续执行。这一过程会一直持续到所有的 `countdown()` 协程执行完毕，事件循环也被清空。稍后我会给你展示一个完整的例子，用来说明协程/事件循环之类的这些东西究竟是如何运作的，但是首先我想要解释一下 `async` 和 `await` 。

Python 3.5 从 `yield from` 到 `await`

在 Python 3.4 中，用于异步编程并被标记为协程的函数看起来是这样的：

```
# This also works in Python 3.5.
@asyncio.coroutine
def py34_coro():
    yield from stuff()
```

Python 3.5 添加了 `types.coroutine` 修饰器，也可以像 `asyncio.coroutine` 一样将生成器标记为协程。你可以用 `async def` 来定义一个协程函数，虽然这个函数不能包含任何形式的 `yield` 语句；只有 `return` 和 `await` 可以从协程中返回值。

```
async def py35_coro():
    await stuff()
```

虽然 `async` 和 `types.coroutine` 的关键作用在于巩固了协程的定义，但是它将协程从一个简单的接口变成了一个实际的类型，也使得一个普通生成器和用作协程的生成器之间的差别变得更加明确

（ `inspect.iscoroutine()` 函数甚至明确规定必须使用 `async` 的方式定义才行）

义为什么。

你将发现不仅仅是 `async`，Python 3.5 还引入 `await` 表达式（只能用于 `async def` 中）。虽然 `await` 的使用和 `yield from` 很像，但 `await` 可以接受的对象却是不同的。`await` 当然可以接受协程，因为协程的概念是所有这一切的基础。但是当你使用 `await` 时，其接受的对象必须是*awaitable* 对象：必须是定义了 `__await__()` 方法且这一方法必须返回一个**不是**协程的迭代器。协程本身也被认为是 awaitable 对象（这也是 `collections.abc.Coroutine` 继承 `collections.abc.Awaitable` 的原因）。这一定义遵循 Python 将大部分语法结构在底层转化成方法调用的传统，就像 `a + b` 实际上是 `a.__add__(b)` 或者 `b.__radd__(a)` 。

`yield from` 和 `await` 在底层的差别是什么（也就是 `types.coroutine` 与 `async def` 的差别）？让我们看一下上面两则Python 3.5代码的例子所产生的字节码在本质上有何差异。`py34_coro()` 的字节码是：

```
>>> dis.dis(py34_coro)
  2           0 LOAD_GLOBAL              0 (stuff)
              3 CALL_FUNCTION            0 (0 positional, 0 key
              6 GET_YIELD_FROM_ITER
              7 LOAD_CONST               0 (None)
             10 YIELD_FROM
             11 POP_TOP
             12 LOAD_CONST               0 (None)
             15 RETURN_VALUE
```

`py35_coro()` 的字节码是：

```
>>> dis.dis(py35_coro)
  1           0 LOAD_GLOBAL              0 (stuff)

              3 CALL_FUNCTION            0 (0 positional, 0 key
              6 GET_AWAITABLE
```

```
 0 GET_AWAITABLE
 7 LOAD_CONST               0 (None)
10 YIELD_FROM
11 POP_TOP
12 LOAD_CONST               0 (None)
15 RETURN_VALUE
```

忽略由于 `py34_coro()` 的 `asyncio.coroutine` 修饰器所带来的行号的差别，两者之间唯一可见的差异是 `GET_YIELD_FROM_ITER` 操作码 对比 `GET_AWAITABLE` 操作码。两个函数都被标记为协程，因此在这里没有差别。 `GET_YIELD_FROM_ITER` 只是检查参数是生成器还是协程，否则将对其参数调用 `iter()` 方法（只有用在协程内部的时候 `yield from` 所对应的操作码才可以接受协程对象，在这个例子里要感谢 `types.coroutine` 修饰符将这个生成器在C语言层面标记为 `CO_ITERABLE_COROUTINE` ）。

但是 `GET_AWAITABLE` 的做法不同，其字节码像 `GET_YIELD_FROM_ITER` 一样接受协程，但是不接受没有被标记为协程的生成器。就像前面讨论过的一样，除了协程以外，这一字节码还可以接受_awaitable_对象。这使得 `yield from` 和 `await` 表达式都接受协程但分别接受一般的生成器和awaitable对象。

你可能会想，为什么基于 `async` 的协程和基于生成器的协程会在对应的暂停表达式上面有所不同？主要原因是出于最优化Python性能的考虑，确保你不会将刚好有同样API的不同对象混为一谈。由于生成器默认实现协程的API，因此很有可能在你希望用协程的时候错用了一个生成器。而由于并不是所有的生成器都可以用在基于协程的控制流中，你需要避免错误地使用生成器。但是由于 Python 并不是静态编译的，它最好也只能在用基于生成器定义的协程时提供运行时检查。这意味着当用 `types.coroutine` 时，Python 的编译器将无法判断这个生成器是用作协程还是仅仅是普通的生成器（记住，仅仅因为 `types.coroutine` 这一语法

的字面意思，并不意味着在此之前没有人做过 `types = spam` 的操作），因此编译器只能基于当前的情况生成有着不同限制的操作码

因此编译器只能基于当前的情况生成有着不同限制的操作码。

关于基于生成器的协程和 `async` 定义的协程之间的差异，我想说明的关键点是只有基于生成器的协程可以真正的暂停执行并强制性返回给事件循环。你可能不了解这些重要的细节，因为通常你调用的像是 `asyncio.sleep()` function 这种事件循环相关的函数，由于事件循环实现他们自己的API，而这些函数会处理这些小的细节。对于我们绝大多数人来说，我们只会跟事件循环打交道，而不需要处理这些细节，因此可以只用 `async` 定义的协程。但是如果你和我一样好奇为什么不能仅通过 `async` 定义的协程来实现 `asyncio.sleep()` ，那么这里的解释应该可以让你顿悟。

总结

让我们用简单的话来总结一下。用 `async def` 可以定义得到_协程_。定义协程的另一种方式是通过 `types.coroutine` 修饰器 -- 从技术实现的角度来说就是添加了 `CO_ITERABLE_COROUTINE` 标记 -- 或者是 `collections.abc.Coroutine` 的子类。你只能通过基于生成器的定义来实现协程的暂停。

_awaitable 对象_要么是一个协程要么是一个定义了 `__await__()` 方法的对象 -- 也就是 `collections.abc.Awaitable` -- 且 `__await__()` 必须返回一个不是协程的迭代器。 `await` 表达式基本上与 `yield from` 相同但只能接受awaitable对象（普通迭代器不行）。 `async` 定义的函数要么包含 `return` 语句 -- 包括所有Python函数缺省的 `return None` -- 和/或者 `await` 表达式（ `yield` 表达式不行）。 `async` 函数的限制确保你不会将基于生成器的协程与普通的生成器混合使用，因为对这两种生成器的期望是非常不同的。

将 `async` / `await` 看做异步编程的 API

我想要重点指出的地方实际上在我看David Beazley's Python Brasil 2015 keynote之前还没有深入思考过。在他的演讲中，David 指出 `async` / `await` 实际上是异步编程的 API（他在 Twitter 上向我重申过）。David 的意思是人们不应该将 `async` / `await` 等同于 `asyncio`，而应该将 `asyncio` 看作是一个利用 `async` / `await` API 进行异步编程的框架。

David 将 `async` / `await` 看作是异步编程的API创建了 `curio` 项目来实现他自己的事件循环。这帮助我弄清楚 `async` / `await` 是 Python 创建异步编程的原料，同时又不会将你束缚在特定的事件循环中也无需与底层的细节打交道（不像其他编程语言将事件循环直接整合到语言中）。这允许像 `curio` 一样的项目不仅可以在较低层面上拥有不同的操作方式（例如 `asyncio` 利用 future 对象作为与事件循环交流的 API，而 `curio` 用的是元组），同时也可以集中解决不同的问题，实现不同的性能特性（例如 `asyncio` 拥有一整套框架来实现运输层和协议层，从而使其变得可扩展，而 `curio` 只是简单地让用户来考虑这些但同时也让它运行地更快）。

考虑到 Python 异步编程的（短暂）历史，可以理解人们会误认为 `async` / `await` == `asyncio`。我是说 `asyncio` 帮助我们可以在 Python 3.4 中实现异步编程，同时也是 Python 3.5 中引入 `async` / `await` 的推动因素。但是 `async` / `await` 的设计意图就是为了让其足够灵活从而**不需要**依赖 `asyncio` 或者仅仅是为了适应这一框架而扭曲关键的设计决策。换句话说，`async` / `await` 延续了 Python 设计尽可能灵活的传统同时又非常易于使用（实现）。

一个例子

到这里你的大脑可能已经灌满了新的术语和概念，导致你想要从整体上把握所有这些东西是如何让你可以实现异步编程的稍微有些困难。为了帮助你让这一切更加具体化，这里有一个完整的（伪造的）异步编程的例子，将代码与事件循环及其相关的函数一一对应起来。这个例子用包

例子，将代码与事件循环及其相关的函数一一对应起来。这个例子里包含的几个协程，代表着火箭发射的倒计时，并且看起来是同时开始的。这是通过并发实现的异步编程；3个不同的协程将分别独立运行，并且都在同一个线程内完成。

```
import datetime
import heapq
import types
import time

class Task:

    """Represent how long a coroutine should before starting ag

    Comparison operators are implemented for use by heapq. Two-
    tuples unfortunately don't work because when the datetime.d
    instances are equal, comparison falls to the coroutine and
    implement comparison methods, triggering an exception.

    Think of this as being like asyncio.Task/curio.Task.
    """

    def __init__(self, wait_until, coro):
        self.coro = coro
        self.waiting_until = wait_until

    def __eq__(self, other):
        return self.waiting_until == other.waiting_until

    def __lt__(self, other):
        return self.waiting_until < other.waiting_until

class SleepingLoop:

    """An event loop focused on delaying execution of coroutine
```

```
        An event loop focused on delaying execution of coroutine

    Think of this as being like asyncio.BaseEventLoop/curio.Ker
    """

    def __init__(self, *coros):
        self._new = coros
        self._waiting = []

    def run_until_complete(self):
        # Start all the coroutines.
        for coro in self._new:
            wait_for = coro.send(None)
            heapq.heappush(self._waiting, Task(wait_for, coro))
        # Keep running until there is no more work to do.
        while self._waiting:
            now = datetime.datetime.now()
            # Get the coroutine with the soonest resumption ti
            task = heapq.heappop(self._waiting)
            if now < task.waiting_until:
                # We're ahead of schedule; wait until it's time
                delta = task.waiting_until - now
                time.sleep(delta.total_seconds())
                now = datetime.datetime.now()
            try:
                # It's time to resume the coroutine.
                wait_until = task.coro.send(now)
                heapq.heappush(self._waiting, Task(wait_until,
            except StopIteration:
                # The coroutine is done.
                pass

@types.coroutine

def sleep(seconds):
    """Pause a coroutine for the specified number of seconds
```

```
    Pause a coroutine for the specified number of seconds.

    Think of this as being like asyncio.sleep()/curio.sleep().
    """
    now = datetime.datetime.now()
    wait_until = now + datetime.timedelta(seconds=seconds)
    # Make all coroutines on the call stack pause; the need to
    # necessitates this be generator-based and not an async-ba
    actual = yield wait_until
    # Resume the execution stack, sending back how long we act
    return actual - now

async def countdown(label, length, *, delay=0):
    """Countdown a launch for `length` seconds, waiting `delay`

    This is what a user would typically write.
    """
    print(label, 'waiting', delay, 'seconds before starting cou
    delta = await sleep(delay)
    print(label, 'starting after waiting', delta)
    while length:
        print(label, 'T-minus', length)
        waited = await sleep(1)
        length -= 1
    print(label, 'lift-off!')

def main():
    """Start the event loop, counting down 3 separate launches.

    This is what a user would typically write.
    """
    loop = SleepingLoop(countdown('A', 5), countdown('B', 3, de
                        countdown('C', 4, delay=1))

    start = datetime.datetime.now()
    loop.run_until_complete()
```

```
    loop.run_until_complete()
    print('Total elapsed time is', datetime.datetime.now() - s

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

就像我说的，这是伪造出来的，但是如果你用 Python 3.5 去运行，你会发现这三个协程在同一个线程内独立运行，并且总的运行时间大约是5秒钟。你可以将 `Task` ， `SleepingLoop` 和 `sleep()` 看作是事件循环的提供者，就像 `asyncio` 和 `curio` 所提供给你的一样。对于一般的用户来说，只有 `countdown()` 和 `main()` 函数中的代码才是重要的。正如你所见， `async` 和 `await` 或者是这整个异步编程的过程并没什么黑科技；只不过是 Python 提供给你帮助你更简单地实现这类事情的API。

我对未来的希望和梦想

现在我理解了 Python 中的异步编程是如何运作的了，我想要一直用它！这是如此绝妙的概念，比你之前用过的线程好太多了。但是问题在于 Python 3.5 还太新了， `async` / `await` 也太新了。这意味着还没有太多库支持这样的异步编程。例如，为了实现 HTTP 请求你要么不得不自己徒手构建，要么用像是 `aiohttp` 之类的框架 将 HTTP 添加在另外一个事件循环的顶端，或者寄希望于更多像 `hyper` 库一样的项目不停涌现，可以提供对于 HTTP 之类的抽象，可以让你随便用任何 I/O 库 来实现你的需求（虽然可惜的是 `hyper` 目前只支持 HTTP/2）。

对于我个人来说，我希望更多像 `hyper` 一样的项目可以脱颖而出，这样我们就可以在从 I/O中读取与解读二进制数据之间做出明确区分。这样的抽象非常重要，因为Python多数 I/O 库中处理 I/O 和处理数据是紧紧耦合在一起的。Python 的标准库 `http` 就有这样的问题，它不提供HTTP解析而只有一个连接对象为你处理所有的 I/O。而如果你寄希望于 `requests` 可以支持异步编程，那你的希望已经破灭了，因为 `requests` 的同步 I/O 已经烙进它的设计中了。Python 在网络堆栈上很

`requests` 的同步 I/O 已经超过它的设计平台。Python 在网络堆栈上很多层都缺少抽象定义，异步编程能力的改进使得 Python 社区有机会对此作出修复。我们可以很方便地让异步代码像同步一样执行，这样一些填补异步编程空白的工具可以安全地运行在两种环境中。

我希望 Python 可以让 `async` 协程支持 `yield` 。或者需要用一个新的关键词来实现（可能像 `anticipate` 之类？），因为不能仅靠 `async` 就实现事件循环让我很困扰。幸运的是，我不是唯一一个这么想的人，而且PEP 492的作者也和我意见一致，我觉得还是有机会可以移除掉这点小瑕疵。

结论

基本上 `async` 和 `await` 产生神奇的生成器，我们称之为协程，同时需要一些额外的支持例如 awaitable 对象以及将普通生成器转化为协程。所有这些加到一起来支持并发，这样才使得 Python 更好地支持异步编程。相比类似功能的线程，这是一个更妙也更简单的方法。我写了一个完整的异步编程例子，算上注释只用了不到100行 Python 代码 -- 但仍然非常灵活与快速（curio FAQ 指出它比 `twisted` 要快 30-40%，但是要比 `gevent` 慢 10-15%，而且全部都是有纯粹的 Python 实现的；记住 Python 2 + Twisted 内存消耗更少同时比Go更容易调试，想象一下这些能帮你实现什么吧！）。我非常高兴这些能够在 Python 3 中成为现实，我也非常期待 Python 社区可以接纳并将其推广到各种库和框架中区，可以使我们都能够受益于 Python 异步编程带来的好处！

Python 译文