

# 用 Python 3 的 `async / await` 做异步编程

2017/08/27 · 基础知识 · 异步

分享到: 0

原文出处: keakon 的涂鸦馆

前年我曾写过一篇《初探 Python 3 的异步 IO 编程》，当时只是初步接触了一下 `yield from` 语法和 `asyncio` 标准库。前些日子我在 V2EX 看到一篇《为什么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？》，激起了我再探 Python 3 异步编程的兴趣。然而看了很多文章和，才发现极少提到 `async` 和 `await` 实际意义的，绝大部分仅止步于对 `asyncio` 库的使用，真正有所帮助的只有《How the heck does `async/await` work in Python 3.5?》和《A tale of event loops》这两篇。

在接着写下去之前，我先列举一些 PEPs 以供参考：

- PEP 255 — Simple Generators
- PEP 342 — Coroutines via Enhanced Generators
- PEP 380 — Syntax for Delegating to a Subgenerator
- PEP 492 — Coroutines with `async` and `await` syntax
- PEP 525 — Asynchronous Generators

从这些 PEPs 中可以看出 Python 生成器 / 协程的发展历程：先是 PEP 255 引入了简单的生成器；接着 PEP 342 赋予了生成器 `send()` 方法，使其可以传递数据，协程也就有了实际意义；接下来，PEP 380 增加了 `yield from` 语法，简化了调用子生成器的语法；然后，PEP 492 将协程和生成器区分开，使得其更不易被用错；最后，PEP 525 提供了异步生成器，使得编写异步的数据产生器得到简化。

本文将简单介绍一下这些 PEPs，着重深入的则是 PEP 492。

首先提一下生成器（generator）。  


Generator function 是函数体里包含 `yield` 表达式的函数，它在调用时生成一个 `generator` 对象（以下将其命名为 `gen`）。第一次调用 `next(gen)` 或 `gen.send(None)` 时，将进入它的函数体：在执行到 `yield` 表达式时，向调用者返回数据；当函数返回时，抛出 `StopIteration` 异常。在该函数未执行完之

前，可再次调用 `next(gen)` 进入函数体，也可调用 `gen.send(value)` 向其传递参数，以供其使用（例如提供必要的的数据，或者控制其行为等）。

由于它主要的作用是产生一系列数据，所以一般使用 `for ... in gen` 的语法来遍历它，以简化 `next()` 的调用和手动捕捉 `StopIteration` 异常。  
即：

Python

```
while True:
    try:
        value = next(gen)
        process(value)
```

```
1 while True:
2     try:
3         value = next(gen)
4         process(value)
5     except StopIteration:
6         break
```

可以简化为：

Python

```
for value in gen:
    process(value)
```

```
1 for value in gen:
2     process(value)
```

值可以通过`gen.send(n)`或`next(gen)`来操作`gen`

由于生成器提供了再次进入一个函数体的机制，其实它已经可以当成协程来使用了。

写个很简单的例子：

Python

```
import select
import socket
```

```
1 import select
2 import socket
3
4
5 def coroutine():
6     sock = socket.socket()
```

```

7 sock.setblocking(0)
8 address = yield sock
9 try:
10     sock.connect(address)
11 except BlockingIOError:
12     pass
13 data = yield
14 size = yield sock.send(data)
15 yield sock.recv(size)
16
17
18 def main():
19     coro = coroutine()
20     sock = coro.send(None)
21     wait_list = (sock.fileno(),)
22     coro.send(('www.baidu.com', 80))
23     select.select((), wait_list, ())
24     coro.send(b'GET / HTTP/1.1\r\nHost: www.baidu.com\r\nConnection: Close\r\n\r\n')
25     select.select(wait_list, (), ())
26     print(coro.send(1024))

```

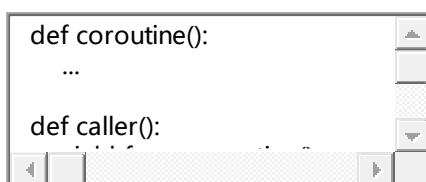
这里的 `coroutine` 函数用于处理连接和收发数据，而 `main` 函数则等待读写和传递参数。虽然看上去和同步的调用没啥区别，**但其实在 `main` 函数中可以同时执行多个 `coroutine`，以实现并发执行。**

再说一下 `yield from`。

如果一个生成器内部需要遍历另一个生成器，并将数据返回给调用者，你需要遍历它并处理所遇到的异常；而用了 `yield from` 后，则可以一行代码解决这些问题。具体例子就不列出了，PEP 380 里有详细的代码。

这对于协程而言也是一个利好，这使得它的调用也得到了简化：

Python



```

1 def coroutine():
2     ...
3
4 def caller():
5     yield from coroutine()

```

接下来就该轮到协程（`coroutine`）登场了。

从上文也可看出，调用 `yield from gen` 时，我无法判定我是遍历了一个生成器，还是调用了协程，这种混淆使得接口的设计者和使用者需要花费额外的工夫来约定和检查。

于是 Python 又先后添加了 `asyncio.coroutine` 和 `types.coroutine` 这两个装饰器来标注协程，这样就使得需要使用协程时，不至于误用了生成器。顺带一提，前者是 `asyncio` 库的实现，需要保持向下兼容，本文暂不讨论；后者则是 Python 3.5 的语言实现，实际上是给函数的 `__code__.co_flags` 设置 `CO_ITERABLE_COROUTINE` 标志。随后，`async def` 也被引入以用于定义协程，它则是设置 `CO_COROUTINE` 标志。

至此，协程和生成器就得以区分，其中以 `types.coroutine` 定义的协程称为基于生成器的协程（generator-based coroutine），而以 `async def` 定义的协程则称为原生协程（native coroutine）。

这两种协程之间的区别其实并不大，非要追究的话，主要有这些：

- 原生协程里不能有 `yield` 或 `yield from` 表达式。
- 原生协程被垃圾回收时，如果它从来没被使用过（即调用 `await coro` 或 `coro.send(None)`），会抛出 `RuntimeWarning`。
- 原生协程没有实现 `__iter__` 和 `__next__` 方法。
- 简单的生成器（非协程）不能 `yield from` 原生协程
- 对原生协程及其函数分别调用 `inspect.isgenerator()` 和 `inspect.isgeneratorfunction()` 将返回 `False`。

实际使用时，如果不考虑向下兼容，可以都用原生协程，除非这个协程里用到了 `yield` 或 `yield from` 表达式。

定义了协程函数以后，就可以调用它们了。

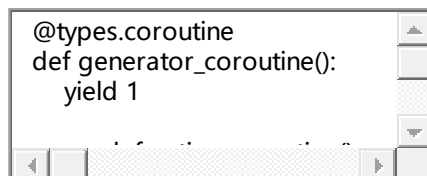
PEP 492 也引入了一个 `await` 表达式来调用协程，它的用法和 `yield from` 差不多，但是它只能在协程函数内部使用，且只能接 awaitable 的对象。

所谓 awaitable 的对象，就是其 `__await__` 方法返回一个迭代器的对象。原生协程和基于生成器的协程都是 `awaitable` 的对象。

另一种调用协程的方法则和生成器一样，调用其 `send` 方法，并自行迭代。这种方式主要用于在非协程函数里调用协程。

举例来说，调用的代码会类似这样：

Python

A screenshot of a Python code editor window. The code inside the editor is: 

```
@types.coroutine
def generator_coroutine():
    yield 1
```

```
1 @types.coroutine
2 def generator_coroutine():
3     yield 1
```

```

4
5 async def native_coroutine():
6     await generator_coroutine()
7
8 def main():
9     native_coroutine().send(None)

```

其中 `generator_coroutine` 函数里因为用到了 `yield` 表达式，所以只能定义成基于生成器的协程；`native_coroutine` 函数由于自身是协程，可以直接用 `await` 表达式调用其他协程；`main` 函数由于不是协程，因而需要用 `native_coroutine().send(None)` 这种方式来调用协程。

这个例子其实也解释了 V2EX 里提到的那个问题，即为什么原生协程不能「真正的暂停执行并强制性返回给事件循环」。

假设事件循环在 `main` 函数里，原生协程是 `native_coroutine` 函数，那要怎样才能让它暂停并返回 `main` 函数呢？

很显然 `await generator_coroutine()` 是不行的，这会进入 `generator_coroutine` 的函数体，而不是回到 `main` 函数；如果 `yield` 一个值，又会遇到之前提到的一个限制，即原生协程里不能有 `yield` 表达式；最后仅剩 `return` 或 `raise` 这两种选择了，但它们虽然能回到 `main` 函数，却不是「暂停」，因为再也无法「继续」了。

指在 `native_coroutine` 中执行 `await generator_coroutine`，后面的 `yield` 也是指在 `native`

所以一般而言，如果要用 Python 3.5 来做异步编程的话，最外层的事件循环需要调用协程的 `send` 方法，里面大部分的异步方法都可以用原生协程来实现，但最底层的异步方法则需要用基于生成器的协程。

为了有个更直观的认识，再来举个例子，抓取 10 个百度搜索的页面：

Python

```

from selectors import DefaultSelector
import socket
from types import coroutine
from urllib.parse import urlparse

```

```

1 from selectors import DefaultSelector, EVENT_READ, EVENT_WRITE
2 import socket
3 from types import coroutine
4 from urllib.parse import urlparse
5
6
7 @coroutine
8 def until_readable(fileobj):
9     yield fileobj, EVENT_READ
10
11
12 @coroutine

```

```
13 def until_writable(fileobj):
14     yield fileobj, EVENT_WRITE
15
16
17 async def connect(sock, address):
18     try:
19         sock.connect(address)
20     except BlockingIOError:
21         await until_writable(sock)
22
23
24 async def recv(fileobj):
25     result = b''
26     while True:
27         try:
28             data = fileobj.recv(4096)
29             if not data:
30                 return result
31             result += data
32         except BlockingIOError:
33             await until_readable(fileobj)
34
35
36 async def send(fileobj, data):
37     while data:
38         try:
39             sent_bytes = fileobj.send(data)
40             data = data[sent_bytes:]
41         except BlockingIOError:
42             await until_writable(fileobj)
43
44
45 async def fetch_url(url):
46     parsed_url = urlparse(url)
47     if parsed_url.port is None:
48         port = 443 if parsed_url.scheme == 'https' else 80
49     else:
50         port = parsed_url.port
51
52     with socket.socket() as sock:
53         sock.setblocking(0)
54         await connect(sock, (parsed_url.hostname, port))
55         path = parsed_url.path if parsed_url.path else '/'
56         path_with_query = '{}?{}'.format(path, parsed_url.query) if parsed_url.query else path
```

```

57     await     send(sock,      'GET      {}      HTTP/1.1\r\nHost:      {}\r\nConnection:
58 Close\r\n\r\n'.format(path_with_query, parsed_url.netloc).encode())
59     content = await recv(sock)
60     print('{}: {}'.format(url, content))
61
62
63 def main():
64     urls = ['http://www.baidu.com/s?wd={}'.format(i) for i in range(10)]
65     tasks = [fetch_url(url) for url in urls] # 将任务定义成协程对象
66
67     with DefaultSelector() as selector:
68         while tasks or selector.get_map(): # 有要做的任务，或者有等待的 IO 事件
69             events = selector.select(0 if tasks else 1) # 如果有要做的任务，立刻获得当前已就绪
70 的 IO 事件，否则最多等待 1 秒
71             for key, event in events:
72                 task = key.data
73                 tasks.append(task) # IO 事件已就绪，可以执行新 task 了
74                 selector.unregister(key.fileobj) # 取消注册，避免重复执行
75
76             for task in tasks:
77                 try:
78                     fileobj, event = task.send(None) # 开始或继续执行 task
79                 except StopIteration:
80                     pass
81                 else:
82                     selector.register(fileobj, event, task) # task 还未执行完，需要等待 IO，将 task
83 注册为 key.data
84
85                 tasks.clear()

```

main()

其他的函数都没什么好说的，主要解释下 `until_readable`、`until_writable` 和 `main` 函数。

其实 `until_readable` 和 `until_writable` 函数都是 `yield` 一个 `(fileobj, event)` 元组，用于告知事件循环，这个 `fileobj` 的 `event` 事件需要被监听。

而在 `main` 函数中，事件循环遍历并执行 `tasks` 里包含的协程。这些协程在等待 IO 时返回事件循环，由事件循环注册事件及其对应的协程。到下一个事件循环时，取出所有就绪的事件，继续执行其对应的协程，就完成了整个的异步执行过程。

如果关注到 `fetch_url` 函数，就会发现业务逻辑用到的代码其实挺简单，只是 `await` 异步函数而已。这虽然简化了大部分的开发工作，但其实也限制了它的表达能力，因为在一个协程内，不能同时 `await` 多个异步函数——它实际上是顺序执行的，只是不同协程之间可以异步执行而已。

考虑一个 HTTP/2 的客户端，它和服务端之间的连接是多路复用的，也就是可以在一个连接里同时发出和接收多份数据，而这些数据的传输是乱序的。如果一份 JavaScript 资源已经下载完毕，没必要再等其他的图片资源下载完毕才能执行。要做到这点，就需要协程有并发执行多个子协程，共同完成任务的能力。这在使用多线程或回调函数时是很容易做到的，但使用 `await` 就显得捉襟见肘了。倒也不是不能做，只是需要拿之前的代码改下，`yield` 一些子协程，并在事件循环中判断一下类型就行了。

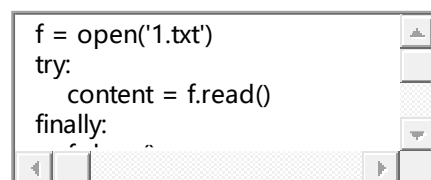
虽然仅用上述提到的东西，已经能做异步编程了，但我还是得补充 2 个漏掉的语法知识：

### 1. `async with`

先考虑普通的 `with` 语句，它的主要作用是在进入和退出一个区域时，做一些初始化和清理工作。

例如：

Python

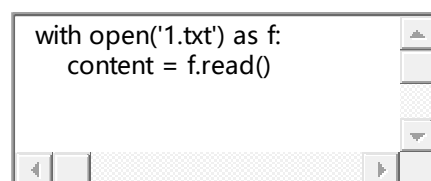


```
f = open('1.txt')
try:
    content = f.read()
finally:
```

```
1 f = open('1.txt')
2 try:
3     content = f.read()
4 finally:
5     f.close()
```

就可以改写为：

Python



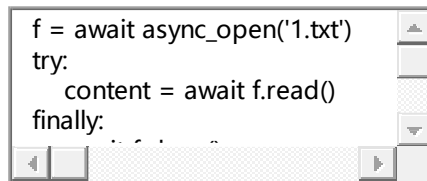
```
with open('1.txt') as f:
    content = f.read()
```

```
1 with open('1.txt') as f:
2     content = f.read()
```

这里要求 `open` 函数返回的 `f` 对象带有 `__enter__` 和 `__exit__` 方法。其中，`__enter__` 方法只需要返回一个文件对象就行了，`__exit__` 则需要调用这个文件对象的 `close` 方法。类似的，假设这个 `open` 函数和 `close` 方法变成了异步的，你的代码可能是这样的：

Python



A screenshot of a Python code editor window. The code inside is:

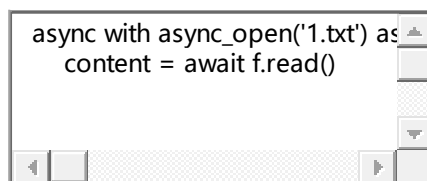
```
f = await async_open('1.txt')
try:
    content = await f.read()
finally:
    await f.close()
```

The editor has a standard interface with a scrollbar on the right and navigation buttons at the bottom.

```
1 f = await async_open('1.txt')
2 try:
3     content = await f.read()
4 finally:
5     await f.close()
```

你就可以用 `async with` 来改写它：

Python

A screenshot of a Python code editor window. The code inside is:

```
async with async_open('1.txt') as f:
    content = await f.read()
```

```
1 async with async_open('1.txt') as f:
2     content = await f.read()
```

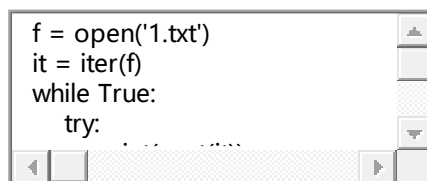
相应的，`async_open` 函数返回的 `f` 对象需要实现 `__aenter__` 和 `__aexit__` 这 2 个异步方法。

## 2. `async for`

这里也先考虑普通的 `for` 语句，它的主要作用是遍历一个迭代器。

例如：

Python

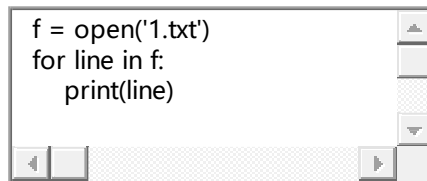
A screenshot of a Python code editor window. The code inside is:

```
f = open('1.txt')
it = iter(f)
while True:
    try:
        line = next(it)
        print(line)
```

```
1 f = open('1.txt')
2 it = iter(f)
3 while True:
4     try:
5         print(next(it))
6     except StopIteration:
7         break
```

可以改写为：

Python

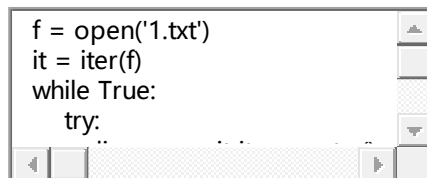


```
f = open('1.txt')
for line in f:
    print(line)
```

```
1 f = open('1.txt')
2 for line in f:
3     print(line)
```

这里要求 `open` 函数返回的 `f` 对象返回一个迭代器对象，即实现了 `__iter__` 方法，这个方法要返回一个实现了 `__next__` 方法的对象。而 `__next__` 方法在每次调用时，都返回下一行的文件内容，直到文件结束时抛出 `StopIteration` 异常。类似的，假如 `__next__` 方法变成了异步的，你的代码可能是这样的：

Python

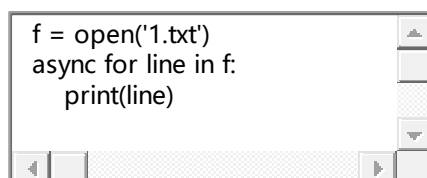


```
f = open('1.txt')
it = iter(f)
while True:
    try:
```

```
1 f = open('1.txt')
2 it = iter(f)
3 while True:
4     try:
5         line = await it.__anext__()
6         print(line)
7     except StopAsyncIteration:
8         break
```

你可以用 `async with` 来改写它：

Python



```
f = open('1.txt')
async for line in f:
    print(line)
```

```
1 f = open('1.txt')
2 async for line in f:
3     print(line)
```

相应的，所需实现的方法分别变成了 `__aiter__` 和 `__anext__`。其中，后者是异步方法。顺带一提，PEP 525 引入的异步生成器（asynchronous generator）就实现了这两个方法。在异步方法中使用 `yield` 表达式，会将其变成异步生成器函数（Python 3.6 以后可用，3.5 之前是语法错误）。值得注

意的是，异步生成器没有实现 `__await__` 方法，因此它不是协程，也不能被 `await`。

1 赞 8 收藏

[评论](#)

## 相关文章

- Python 中的异步编程：Asyncio
- Python 的异步 IO：Asyncio 简介（一）
- 深入理解 python3.4 中 Asyncio 库与 Node.js 的异步 IO 机制
- Python 并发编程之协程/异步 IO
- 异步任务神器 Celery 简明笔记