

# asyncio并发

并发是指一次处理多件事。

并行是指一次做多件事。

二者不同，但是有联系。

一个关于结构，一个关于执行。

并发用于制定方案，用来解决可能（但未必）并行的问题。

——Rob Pike Go语言的创造者之一

真正的并行需要多个核心。现代的笔记本电脑有 4 个CPU核心，但是通常不经意间就有超过100个进程同时运行。因此，实际上大多数过程都是并发处理的，而不是并行处理。计算机始终运行着100多个进程，确保每个进程都有机会取得进展，不过，CPU本身同时做的事情不能超过四件。十年前使用的设备也能并发处理100个进程，不过都在同一个核心里。鉴于此，Rob Pike 才把那次演讲取名为“Concurrency Is Not Parallelism (It's Better)”[“并发不是并行(并发更好)”]

asyncio包，这个包使用事件循环驱动的协程实现并发。这是Python中最大也。是最具雄心壮志的库之一。

Python3.4把 Tulip添加到标准库中时，把它重命名为asyncio。这个包也兼容Python3.3。asyncio大量使用yield from表达式，因此与Python旧版不兼容。

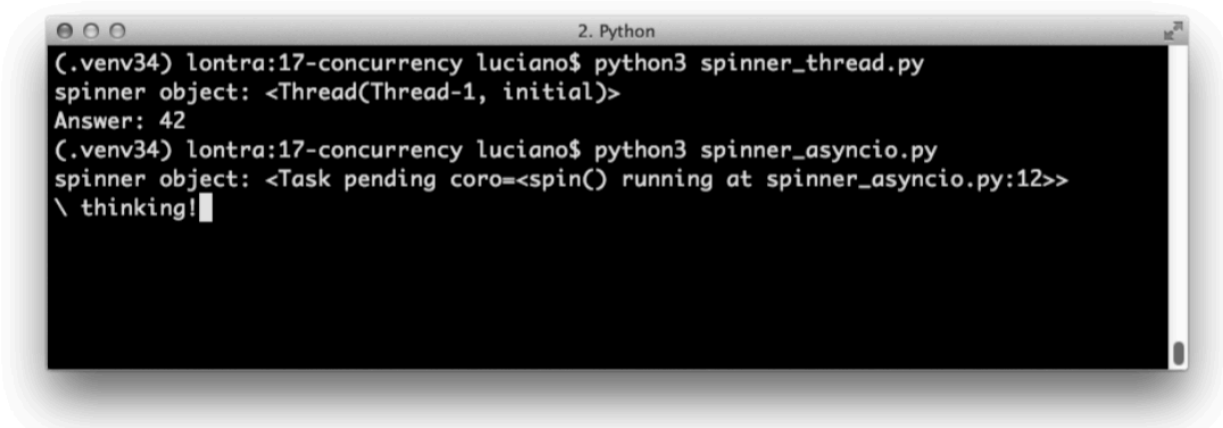
## 线程和协程对比

---

在长时间计算的过程中，在控制台中显示一个由 ASCII 字符 “|/-” 构成的动画旋转指针。

一个借由 threading 模块使用线程实现，一个借由 asyncio 包使用协程实现。我这么做是为了让你对比两种实现，理解如何不使用线程来实现并发行为。

---

A terminal window titled "2. Python" with a dark background and light text. It shows the execution of two Python scripts. The first script, spinner\_thread.py, outputs the spinner object as a Thread and the answer 42. The second script, spinner\_asyncio.py, outputs the spinner object as a Task pending coro and shows the text "\ thinking!" with a cursor.

```
(.venv34) lontra:17-concurrency luciano$ python3 spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.venv34) lontra:17-concurrency luciano$ python3 spinner_asyncio.py
spinner object: <Task pending coro=<spin() running at spinner_asyncio.py:12>>
\ thinking!█
```

spinner\_thread.py: 通过线程以动画形式显示文本式旋转指针

```
1 import threading
2 import itertools
3 import time
4 import sys
5
6 class Signal: # <1>
7     go = True
8
9
10 def spin(msg, done): # <2>
11     write, flush = sys.stdout.write, sys.stdout.flush
12     for char in itertools.cycle('|/-\\'): # <3>
13         status = char + ' ' + msg
14         write(status)
15         flush()
16         write('\x08' * len(status)) # <4>
17         if done.wait(.1): # <5>
18             break
19     write(' ' * len(status) + '\x08' * len(status)) # <6>
20
21
22 def slow_function(): # <7>
23     # pretend waiting a long time for I/O
24     time.sleep(3) # <8>
25     return 42
26
27
28 def supervisor(): # <9>
29     done = threading.Event()
30     spinner = threading.Thread(target=spin,
31                               args=('thinking!', done))
32     print('spinner object:', spinner) # <10>
33     spinner.start() # <11>
34     result = slow_function() # <12>
35     done.set() # <13>
36     spinner.join() # <14>
37     return result
38
39
40 def main():
41     result = supervisor() # <15>
42     print('Answer:', result)
43
44
45 if __name__ == '__main__':
46     main()
```

- ❶这个类定义一个简单的可变对象；其中有个go属性，用于从外部控制线程。
- ❷这个函数会在单独的线程中运行。signal参数是前面定义的Signal类的实例。
- ❸这其实是个无限循环，因为itertools.cycle函数会从指定的序列中反复不断地生成元素。
- ❹这是显示文本式动画的诀窍所在：使用退格符（\x08）把光标移回来。
- ❺如果go属性的值不是True了，那就退出循环。
- ❻使用空格清除状态消息，把光标移回开头。
- ❼假设这是耗时的计算。
- ❽调用sleep函数会阻塞主线程，不过一定要这么做，以便释放 GIL，创建从属线程。
- ❾这个函数设置从属线程，显示线程对象，运行耗时的计算，最后杀死线程。
- ❿显示从属线程对象。输出类似于。
- ⓫启动从属线程。
- ⓬运行slow\_function函数，阻塞主线程。同时，从属线程以动画形式显示旋转指针。
- ⓭改变signal的状态；这会终止spin函数中的那个for循环。
- ⓮等待spinner线程结束。
- ⓯运行supervisor函数。

注意，Python没有提供终止线程的 API，这是有意为之的。若想关闭线程，必须给线程发送消息。这里，我使用的是signal.go属性：在主线程中把它设为False后，spinner 线程最终会注意到，然后干净地退出。

## asyncio.coroutine

下面来看如何使用@asyncio.coroutine装饰器替代线程，实现相同的行为。

asyncio包使用的“协程”是较严格的定义。适合asyncio API的协程在定义体中必须使用yield from，而不能使用yield。此外，适合asyncio的协程要由调用方驱动，并由调用方通过yield from调用；或者把协程传给asyncio包中的某个函数，例如asyncio.async(...)和本章要介绍的其他函数，从而驱动协程。最后，@asyncio.coroutine装饰器应该应用在协程上，如下述示例所示。

spinner\_asyncio.py：通过协程以动画形式显示文本式旋转指针

```

1 import asyncio
2 import itertools
3 import sys
4
5
6 @asyncio.coroutine # <1>
7 def spin(msg): # <2>
8     write, flush = sys.stdout.write, sys.stdout.flush
9     for char in itertools.cycle('|/-\\'):
10         status = char + ' ' + msg
11         write(status)
12         flush()
13         write('\x08' * len(status))
14         try:
15             yield from asyncio.sleep(.1) # <3>
16         except asyncio.CancelledError: # <4>
17             break
18     write(' ' * len(status) + '\x08' * len(status))
19
20
21 @asyncio.coroutine
22 def slow_function(): # <5>
23     # pretend waiting a long time for I/O
24     yield from asyncio.sleep(3) # <6>
25     return 42
26
27
28 @asyncio.coroutine
29 def supervisor(): # <7>
30     spinner = asyncio.async(spin('thinking!')) # <8>
31     print('spinner object:', spinner) # <9>
32     result = yield from slow_function() # <10>
33     spinner.cancel() # <11>
34     return result
35
36
37 def main():
38     loop = asyncio.get_event_loop() # <12>
39     result = loop.run_until_complete(supervisor()) # <13>
40     loop.close()
41     print('Answer:', result)
42
43
44 if __name__ == '__main__':
45     main()

```

除非想阻塞主线程，从而冻结事件循环或者整个应用，否则不要在asyncio协程中使用

`time.sleep(delay)`。如果协程需要在一段时间内什么也不做，应该使用`yield from asyncio.sleep(delay)`或者`await asyncio.sleep(delay)`。

❶ 打算交给 `asyncio` 处理的协程要使用 `@asyncio.coroutine` 装饰。这不是强制要求，但是强烈建议这么做。原因在本列表后面。

❷ 这里不需要示例 线程执行中 `spin` 函数中用来关闭线程的 `signal` 参数。

❸ 使用 `yield from asyncio.sleep(.1)` 代替 `time.sleep(.1)`，这样的休眠不会阻塞事件循环。

❹ 如果 `spin` 函数苏醒后抛出 `asyncio.CancelledError` 异常，其原因是发出了取消请求，因此退出循环。

❺ 现在，`slow_function`函数是协程，在用休眠假装进行I/O操作时，使用`yield from`继续执行事件循环。

❻ `yield from asyncio.sleep(3)` 表达式把控制权交给主循环，在休眠结束后恢复这个协程。

❼ 现在，`supervisor` 函数也是协程，因此可以使用 `yield from` 驱动 `slow_function` 函数。

❽ `asyncio.async(...)` 函数排定 `spin` 协程的运行时间，使用一个 `Task` 对象包装 `spin` 协程，并立即返回。

❾ 显示`Task`对象。输出类似于 `<Task pending coro=<spin() running at spinner_asyncio.py:12>>`。

❿ 驱动 `slow_function()` 函数。结束后，获取返回值。同时，事件循环继续运行，因为 `slow_function`函数最后使用`yield from asyncio.sleep(3)`表达式把控制权交回给了主循环。

11 `Task` 对象可以取消;取消后会在协程当前暂停的 `yield` 处抛出 `asyncio.CancelledError` 异常。协程可以捕获这个异常，也可以延迟取消，甚至拒绝取消。

12 获取事件循环的引用。

13 驱动 `supervisor` 协程，让它运行完毕;这个协程的返回值是这次调用的返回值。

除非想阻塞主线程，从而冻结事件循环或整个应用，否则不要在 `asyncio` 协程中使用 `time.sleep(...)`。如果协程需要在一段时间内什么也不做，应该使用 `yield from asyncio.sleep(Delay)`。

使用 `@asyncio.coroutine` 装饰器不是强制要求，但是强烈建议这么做，因为这样能在一众普通的函数中把协程凸显出来，也有助于调试:如果还没从中产出值，协程就被垃圾回收了(意味着有操作未完成，因此有可能是个缺陷)，那就可以发出警告。这个装饰器不会预激协程。

注意，`spinner_thread.py` 和 `spinner_asyncio.py` 两个脚本的代码行数差不多。`supervisor` 函数是这两个示例的核心。下面详细对比二者。下面示例只列出了线程版和协程版的 `supervisor` 函数：

```

1 # 线程版 supervisor 函数
2 def supervisor(): # <7>
3     spinner = asyncio.async(spin('thinking!')) # <8>
4     print('spinner object:', spinner) # <9>
5     result = yield from slow_function() # <10>
6     spinner.cancel() # <11>
7     return result

```

```

1 # 协程版 supervisor 函数
2 @asyncio.coroutine
3     def supervisor():
4         spinner = asyncio.async(spin('thinking!'))
5         print('spinner object:', spinner)
6         result = yield from slow_function()
7         spinner.cancel()
8         return result

```

这两种 supervisor 实现之间的主要区别概述如下。

- `asyncio.Task` 对象差不多与 `threading.Thread` 对象等效。
- Task 对象用于驱动协程，Thread 对象用于调用可调用的对象。
- Task 对象不由自己动手实例化，而是通过把协程传给 `asyncio.async(...)` 函数或 `loop.create_task(...)` 方法获取。
- 获取的 Task 对象已经排定了运行时间(例如，由 `asyncio.async` 函数排定)；Thread 实例则必须调用 `start` 方法，明确告知让它运行。
- 在线程版 supervisor 函数中，`slow_function` 函数是普通的函数，直接由线程调用。在异步版 supervisor 函数中，`slow_function` 函数是协程，由 `yield from` 驱动。
- 没有 API 能从外部终止线程，因为线程随时可能被中断，导致系统处于无效状态。如果想终止任务，可以使用 `Task.cancel()` 实例方法，在协程内部抛出 `CancelledError` 异常。协程可以在暂停的 `yield` 处捕获这个异常，处理终止请求。
- supervisor 协程必须在 `main` 函数中由 `loop.rununtilcomplete` 方法执行。

上述比较应该能帮助你理解，与更熟悉的 `threading` 模型相比，`asyncio` 是如何编排并发作业的。

线程与协程之间的比较还有最后一点要说明:如果使用线程做过重要的编程，你就知道写出程序有多么困难，因为调度程序任何时候都能中断线程。必须记住保留锁，去保护程序中的重要部分，防止多步操作在执行的过程中中断，防止数据处于无效状态。

而协程默认会做好全方位保护，以防止中断。我们必须显式产出才能让程序的余下部分运行。对协程来说，无需保留锁，在多个线程之间同步操作，协程自身就会同步，因为在任意时刻只有一个协程运行。想交出控制权时，可以使用 `yield` 或 `yield from` 把控制权交还调度程序。这就是能够安全地取消协程的原因:按照定义，协程只能在暂停的 `yield` 处取消，因此可以处理 `CancelledError` 异常，执行清理操作。

# asyncio包

asyncio是Python3.4版本引入的标准库，至少是Python3.3(需要手动安装) 才可以使用。因为自 3.3 版本以来，

yield from 的引入使 Python 具备运行 asyncio 的基本条件。并发编程同时执行多条独立的逻辑流，即使它们是都工作在同个线程中的。每个协程都有独立的栈空间。

3.5版本中，新语法的引入使Python具备了原生协程，而不再是一种新的生成器类型。

asyncio包使用事件循环驱动的协程实现并发。

## Loop

事件循环是由asyncio中央执行装置。它提供多种设施，包括：

- 注册、执行、取消、延迟调用（超时）。
- 为各种通信创建客户机和服务器传输。
- 建立进程和相关的传输来与外部程序的通信。
- 将代价昂贵的函数调用委托给线程池。

事件循环的工作方式与用户设想存在一些偏差，理所当然的认知应是每个线程都可以有一个独立的 loop。但是在运行中，在主线程中才能通过 `asyncio.get_event_loop()` 创建一个新的 loop，而在其他线程时，使用 `get_event_loop()` 却会抛错。

正确的做法应该是 `asyncio.set_event_loop()` 进行当前线程与 loop 的显式绑定。由于 loop 的运作行为并不受 Python 代码的控制，所以无法稳定的将协程拓展到多线程中运行。

协程在工作时，并不了解是哪个 loop 在对其调度，即使调用 `asyncio.get_event_loop()` 也不一定能获取到真正运行的那个 loop。因此在各种库代码中，实例化对象时都必须显式的传递当前的 loop 以进行绑定。

```
1 | def get_event_loop():
2 |     """Return an asyncio event loop."""
```

Python

## Future

不用回调方法编写异步代码后，为了获取异步调用的结果，引入一个 Future 未来对象。Future 封装了与 loop 的交互行为，`add_done_callback` 方法向 `epoll` 注册回调函数，当 `result` 属性得到返回值后，会运行之前注册的回调函数，向上传递给 `coroutine`。

但是，每一个角色各有自己的职责，用 Future 向生成器 send result 以恢复工作状态并不合适，Future 对象本身的生存周期比较短，每一次注册回调、产生事件、触发回调过程后，其工作已经完成。



`run_until_complete()` 执行事件循环，直到Future完成。返回Future结果或者对应的异常。

```
1 def run_until_complete(self, future):
2     """Run the event loop until a Future is done.
3
4     Return the Future's result, or raise its exception.
5     """
6     raise NotImplementedError
```

Python

`run_forever()` 执行事件循环，直到调用`stop()`方法。

```
1 def run_forever(self):
2     """Run the event loop until stop() is called."""
3     raise NotImplementedError
```

Python

```
1 import asyncio
2
3 async def slow_operation(future):
4     await asyncio.sleep(1)
5     future.set_result('Future is done!')
6
7 def got_result(future):
8     print(future.result())
9     loop.stop()
10
11 loop = asyncio.get_event_loop()
12 future = asyncio.Future()
13 asyncio.ensure_future(slow_operation(future))
14 future.add_done_callback(got_result)
15 try:
16     loop.run_forever()
17 finally:
18     loop.close()
```

Python

## Task

Task，顾名思义，是维护生成器协程状态处理执行逻辑的任务，Task 内的`_step` 方法负责生成器协程与 EventLoop 交互过程的状态迁移:向协程 send 一个值，恢复其工作状态，协程运行到断点后，得到新的未来对象，再处理 future 与 loop 的回调注册过程。

### Task function

Note In the functions below, the optional loop argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is

## ensure\_future()

指定一个协程或future的执行，返回一个task对象

```
1 ensure_future(coro_or_future, *, loop=None)
2     Schedule the execution of a coroutine object:
3     wrap it in a future. Return a Task object.
4
5     If the argument is a Future, it is returned directly.
6
7     The function accepts any awaitable object.
```

Python

## wait()

`wait(futures, *, loop=None, timeout=None, return_when=ALL_COMPLETED)` 可以获取一个future的列表，同时返回一个将它们全包括在内的单独的协程(done, pending)。类似 `concurrent.futures.wait()`

所以我们可以这样写：

```
1
2 loop.run_until_complete(asyncio.wait([print_page('http://example.com/foo')
3                                           print_page('http://example.com/bar')])
```

Python

下载国旗的 `flags_asyncio.py` 脚本的完整代码清单。运作方式简述如下。

- (1) 首先，在 `downloadmany` 函数中获取一个事件循环，处理调用 `downloadone` 函数生成的几个协程对象。
- (2) `asyncio` 事件循环依次激活各个协程。
- (3) 客户代码中的协程(如 `get_flag`)使用 `yield from` 把职责委托给库里的协程(如 `aiohttp.request`)时，控制权交还事件循环，执行之前排定的协程。
- (4) 事件循环通过基于回调的低层 API，在阻塞的操作执行完毕后获得通知。
- (5) 获得通知后，主循环把结果发给暂停的协程。
- (6) 协程向前执行到下一个 `yield from` 表达式，例如 `get_flag` 函数中的 `yield from resp.read()`。事件循环再次得到控制权，重复第 4~6 步，直到事件循环终止。

```

1  import asyncio
2
3  import aiohttp # <1>
4
5  from flags import BASE_URL, save_flag, show, main # <2>
6
7
8  @asyncio.coroutine # <3>
9  def get_flag(cc):
10     url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
11     resp = yield from aiohttp.request('GET', url) # <4>
12     image = yield from resp.read() # <5>
13     return image
14
15
16  @asyncio.coroutine
17  def download_one(cc): # <6>
18     image = yield from get_flag(cc) # <7>
19     show(cc)
20     save_flag(image, cc.lower() + '.gif')
21     return cc
22
23
24  def download_many(cc_list):
25     loop = asyncio.get_event_loop() # <8>
26     to_do = [download_one(cc) for cc in sorted(cc_list)] # <9>
27     wait_coro = asyncio.wait(to_do) # <10>
28     res, _ = loop.run_until_complete(wait_coro) # <11>
29     loop.close() # <12>
30
31     return len(res)
32
33
34  if __name__ == '__main__':
35     main(download_many)
36

```

- ❶ 必须安装 aiohttp 包，它不在标准库中。
- ❷ 重用 flags 模块中的一些函数。
- ❸ 协程应该使用 @asyncio.coroutine 装饰。
- ❹ 阻塞的操作通过协程实现，客户代码通过 yield from 把职责委托给协程，以便异步运行协程。
- ❺ 读取响应内容是一项单独的异步操作。
- ❻ download\_one 函数也必须是协程，因为用到了 yield from。
- ❼ 与依序下载版 download\_one 函数唯一的区别是这一行中的 yield from；函数定义体中的其他代码与之前完全一样。
- ❽ 获取事件循环底层实现的引用。

- ⑨ 调用 `download_one` 函数获取各个国旗，然后构建一个生成器对象列表。
- ⑩ 虽然函数的名称是 `wait`，但它不是阻塞型函数。`wait` 是一个协程，等传给它的所有协程运行完毕后结束(这是 `wait` 函数的默认行为;参见这个示例后面的说明)。
- 11 执行事件循环，直到 `waitcoro` 运行结束;事件循环运行的过程中，这个脚本会在这里阻塞。我们忽略 `rununtil_complete` 方法返回的第二个元素。下文说明原因。
- 12 关闭事件循环

`asyncio.wait(...)` 协程的参数是一个由期物或协程构成的可迭代对象;`wait` 会分别把各个协程包装进一个 `Task` 对象。最终的结果是，`wait` 处理的所有对象都通过某种方式变成 `Future` 类的实例。`wait` 是协程函数，因此返回的是一个协程或生成器对象;`wait_coro` 变量中存储的正是这种对象。为了驱动协程，我们把协程传给 `loop.run_until_complete(...)` 方法。

`loop.run_until_complete` 方法的参数是一个期物或协程。如果是协程，`run_until_complete` 方法与 `wait` 函数一样，把协程包装进一个 `Task` 对象中。协程、期物和任务都能由 `yield from` 驱动，这正是 `rununtilcomplete` 方法对 `wait` 函数返回的 `waitcoro` 对象所做的工作。`waitcoro` 运行结束后返回一个元组，第一个元素是一系列结束的期物，第二个元素是一系列未结束的期物。但是 `wait` 函数有两个关键字参数，如果设定了可能会返回未结束的期物，这两个参数是 `timeout` 和 `return_when`。

## gather()

`gather` 的返回值为协程运行的结果。

```
1 | gather(*coros_or_futures, loop=None, return_exceptions=False)
2 |     Return a future aggregating results from the given coroutines
3 |     or futures.
```

```
1 | import asyncio
2 |
3 | async def factorial(name, number):
4 |     f = 1
5 |     for i in range(2, number+1):
6 |         print("Task %s: Compute factorial(%s)..." % (name, i))
7 |         await asyncio.sleep(1)
8 |         f *= i
9 |     print("Task %s: factorial(%s) = %s" % (name, number, f))
10 |
11 | loop = asyncio.get_event_loop()
12 | loop.run_until_complete(asyncio.gather(
13 |     factorial("A", 2),
14 |     factorial("B", 3),
15 |     factorial("C", 4),
16 | ))
17 | loop.close()
```

Python

## as\_completed()

`asyncio.as_completed(fs, *, loop=None, timeout=None)`, 通过它可以获取一个协程的列表, 同时返回一个按完成顺序生成协程的迭代器, 因此当你用它迭代时, 会尽快得到每个可用的结果。

```
1 | return an iterator whose values are coroutines.
2 |
3 | result = loop.run_until_complete(asyncio.as_completed(*tasks))
4 |
5 | for f in as_completed(fs):
6 |     result = yield from f # The 'yield from' may raise
7 |     # Use result code
8 |
9 |     # timeout 设置的一个比较特殊值的时候, 在所有future执行完前,
10 |    会引发timeout 超时, 然后yield from 也就会报错
11 |
12 |
```

Python

```
1 |
2 | import asyncio
3 | import collections
4 | import contextlib
5 |
6 | import aiohttp
7 | from aiohttp import web
8 | import tqdm
9 |
10 | from flags2_common import main, HTTPStatus, Result, save_flag
11 |
12 | # default set low to avoid errors from remote site, such as
13 | # 503 - Service Temporarily Unavailable
14 | DEFAULT_CONCUR_REQ = 5
15 | MAX_CONCUR_REQ = 1000
16 |
17 |
18 | class FetchError(Exception): # <1>
19 |     def __init__(self, country_code):
20 |         self.country_code = country_code
21 |
22 |
23 | @asyncio.coroutine
24 | def get_flag(base_url, cc): # <2>
25 |     url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
26 |     resp = yield from aiohttp.request('GET', url)
27 |     with contextlib.closing(resp):
28 |         if resp.status == 200:
```

```

29         image = yield from resp.read()
30         return image
31     elif resp.status == 404:
32         raise web.HTTPNotFound()
33     else:
34         raise aiohttp.HttpProcessingError(
35             code=resp.status, message=resp.reason,
36             headers=resp.headers)
37
38
39 @asyncio.coroutine
40 def download_one(cc, base_url, semaphore, verbose): # <3>
41     try:
42         with (yield from semaphore): # <4>
43             image = yield from get_flag(base_url, cc) # <5>
44     except web.HTTPNotFound: # <6>
45         status = HTTPStatus.not_found
46         msg = 'not found'
47     except Exception as exc:
48         raise FetchError(cc) from exc # <7>
49     else:
50         save_flag(image, cc.lower() + '.gif') # <8>
51         status = HTTPStatus.ok
52         msg = 'OK'
53
54     if verbose and msg:
55         print(cc, msg)
56
57     return Result(status, cc)

```

- ❶ 这个自定义的异常用于包装其他 HTTP 或网络异常，并获取 country\_code，以便报告 错误。
- ❷ get\_flag 协程有三种返回结果:返回下载得到的图像;HTTP 响应码为 404 时，抛出 web.HTTPNotFound 异常;返回其他 HTTP 状态码时，抛出 aiohttp.HttpProcessingError 异常。
- ❸ semaphore参数是asyncio.Semaphore 类(<https://docs.python.org/3/library/asyncio-sync.html#asyncio.Semaphore>)的实例。Semaphore 类是同步装置，用于限制并发请求数量。
- ❹ 在 yield from 表达式中把 semaphore 当成上下文管理器使用，防止阻塞整个系统:如果 semaphore 计数器的值是所允许的最大值，只有这个协程会阻塞。
- ❺ 退出这个 with 语句后，semaphore 计数器的值会递减，解除阻塞可能在等待同一个 semaphore 对象的其他协程实例。
- ❻ 如果没找到国旗，相应地设置 Result 的状态。
- ❼ 其他异常当作 FetchError 抛出，传入国家代码，并使用“PEP 3134 —Exception Chaining and Embedded Tracebacks”(https://www.python.org/dev/peps/pep-3134/)引入的 raise X from Y 句法链接原来的异常。
- ❽ 这个函数的作用是把国旗文件保存到硬盘中。

可以看出，与依序下载版相比，示例中的 get\_flag 和 download\_one 函数改动幅度很大，因为现在这

两个函数是协程，要使用 yield from 做异步调用。

```
1  @asyncio.coroutine
2  def downloader_coro(cc_list, base_url, verbose, concur_req): # <1>
3      counter = collections.Counter()
4      semaphore = asyncio.Semaphore(concur_req) # <2>
5      to_do = [download_one(cc, base_url, semaphore, verbose)
6                for cc in sorted(cc_list)] # <3>
7
8      to_do_iter = asyncio.as_completed(to_do) # <4>
9      if not verbose:
10         to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list)) # <5>
11     for future in to_do_iter: # <6>
12         try:
13             res = yield from future # <7>
14         except FetchError as exc: # <8>
15             country_code = exc.country_code # <9>
16             try:
17                 error_msg = exc.__cause__.args[0] # <10>
18             except IndexError:
19                 error_msg = exc.__cause__.__class__.__name__ # <11>
20             if verbose and error_msg:
21                 msg = '*** Error for {}: {}'.format(country_code, error_msg)
22                 print(msg)
23             status = HTTPStatus.error
24         else:
25             status = res.status
26
27         counter[status] += 1 # <12>
28
29     return counter # <13>
30
31
32 def download_many(cc_list, base_url, verbose, concur_req):
33     loop = asyncio.get_event_loop()
34     coro = downloader_coro(cc_list, base_url, verbose, concur_req)
35     counts = loop.run_until_complete(coro) # <14>
36     loop.close() # <15>
37
38     return counts
39
40
41 if __name__ == '__main__':
42     main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

- ❶ 这个协程的参数与 download\_many 函数一样，但是不能直接调用，因为它是协程函数，而不是像 download\_many 那样的普通函数。

- ❷ 创建一个 `asyncio.Semaphore` 实例，最多允许激活 `concur_req` 个使用这个计数器的协程。
- ❸ 多次调用 `download_one` 协程，创建一个协程对象列表。
- ❹ 获取一个迭代器，这个迭代器会在期物运行结束后返回期物。
- ❺ 把迭代器传给 `tqdm` 函数，显示进度。
- ❻ 迭代运行结束的期物;这个循环与上边示例中 `download_many` 函数里的那个十分相似;不同的部分主要是异常处理，因为两个 HTTP 库(`requests` 和 `aiohttp`)之间有差异。
- ❼ 获取 `asyncio.Future` 对象的结果，最简单的方法是使用 `yield from`，而不是调用 `future.result()` 方法。
- ❽ `download_one` 函数抛出的各个异常都包装在 `FetchError` 对象里，并且链接原来的异常。
- ❾ 从 `FetchError` 异常中获取错误发生时的国家代码。
- ❿ 尝试从原来的异常(`__cause__`)中获取错误消息。如果在原来的异常中找不到错误消息，使用所链接异常的类名作为错误消息。记录结果。与其他脚本一样，返回计数器。`download_many` 函数只是实例化 `downloader_coro` 协程，然后通过 `run_until_complete` 方法把它传给事件循环。所有工作做完后，关闭事件循环，返回 `counts`。

在 `asyncio` 包的 API 中使用 `yield from` 时，可以看出：

- 我们编写的协程链条始终通过把最外层委派生成器传给 `asyncio` 包 API 中的某个函数(如 `loop.run_until_complete(...)`)驱动。也就是说，使用 `asyncio` 包时，我们编写的代码不通过调用 `next(...)` 函数或 `.send(...)` 方法驱动协程——这一点由 `asyncio` 包实现的事件循环去做。
- 我们编写的协程链条最终通过 `yield from` 把职责委托给 `asyncio` 包中的某个协程函数 或 协程方法(例如示例 18-2 中的 `yield from asyncio.sleep(...)`)，或者其他库中实现高层协议的协程(`get_flag` 协程里的 `resp = yield from aiohttp.request('GET', url)`)。

## asyncio.Future vs concurrent.futures.Future

`asyncio.Future` 类与 `concurrent.futures.Future` 类的接口基本一致，不过实现方式不同，不可以互换。“PEP 3156—Asynchronous IO Support Rebooted: the ‘asyncio’ Module”(<https://www.python.org/dev/peps/pep-3156/>)对这个不幸状况是这样说的：

未来可能会统一 `asyncio.Future` 和 `concurrent.futures.Future` 类实现的期物(例如，为后者添加兼容 `yield from` 的 `__iter__` 方法)。

from (<https://docs.python.org/3.6/library/asyncio-task.html#future>)

```
class asyncio.Future(*, loop=None)
```

This class is almost compatible with `concurrent.futures.Future`.

Differences:

- `result()` 和 `exception()` 不接收 `timeout` 参数并在期物未完成时抛出异常。



- Callbacks registered with `add_done_callback()` are always called via the event loop's `call_soon()`.
- This class is not compatible with the `wait()` and `as_completed()` functions in the `concurrent.futures` package.

## future实例创建

`concurrent.futures.Future` 只是调度执行某物的结果；在 `asyncio` 包中，`BaseEventLoop.create_task(...)` 方法接收一个协程，排定它的运行时间，然后返回一个 `asyncio.Task` 实例——也是 `asyncio.Future` 类的实例，因为 `Task` 是 `Future` 的子类，用于包装协程。这与调用 `Executor.submit(...)` 方法创建 `concurrent.futures.Future` 实例是一个道理。

## future对象的方法

与 `concurrent.futures.Future` 类似，`asyncio.Future` 类也提供了 `.done()`、`.add_done_callback(...)` 和 `.result()` 等方法。

### result() 方法差别很大

- `concurrent.futures.Future.result()` 方法可以指定 `timeout` 参数，设置超时时间
- `asyncio.Future` 类的 `.result()` 方法没有参数，因此不能指定超时时间。此外，如果调用 `.result()` 方法时协程还没运行完毕，那么 `.result()` 方法不会阻塞去等待结果，而是抛出 `asyncio.InvalidStateError` 异常。然而，获取 `asyncio.Future` 对象的结果通常使用 `yield from`，从中产出结果。使用 `yield from` 处理协程，等待协程运行完毕这一步无需我们关心，而且不会阻塞事件循环，因为在 `asyncio` 包中，`yield from` 的作用是把控制权还给事件循环。

注意，使用 `yield from` 处理协程与使用 `add_done_callback` 方法处理协程的作用一样：延迟的操作结束后，事件循环不会触发回调对象，而是设置协程的返回值；而 `yield from` 表达式则在暂停的协程中生成返回值，恢复执行协程。

总之，因为 `asyncio.Future` 类的目的是与 `yield from` 一起使用，所以通常不需要使用以下方法。

- 无需调用 `my_future.add_done_callback(...)`，因为可以直接把想在协程运行结束后执行的操作放在协程中 `yield from my_future` 表达式的后面。这是协程的一大优势：协程是可以暂停和恢复的函数。
- 无需调用 `my_future.result()`，因为 `yield from` 从协程中产出的值就是结果（例如，`result = yield from my_future`）。

当然，有时也需要使用 `.done()`、`.add_done_callback(...)` 和 `.result()` 方法。但是一般情况下，`asyncio.Future` 对象由 `yield from` 驱动，而不是靠调用这些方法驱动。

## async与await

Python 3.5 中引入了这两个关键字用以取代 `asyncio.coroutine` 与 `yield from`，从语义上定义了原生协程

关键字，避免了使用者对生成器协程与生成器的混淆。这个阶段(3.0-3.4)使用 Python 的人不多，因此历史包袱不重，可以进行一些较大的革新。

`await` 的行为类似 `yield from`，但是它们异步等待的对象并不一致，`yield from` 等待的是一个生成器对象，而`await`接收的是定义了`_await_`方法的 `awaitable` 对象。

在 Python 中，协程也是 `awaitable` 对象。

- `async def`函数必定是协程，即使里面不含有`await`语句。
- 如果在`async`函数里面使用`yield`或`yield from`语句，会引发`SyntaxError`异常。
- 在CPython内部，引入两个新的代码对象标识（code object flags）：

(1) `CO_COROUTINE`表示这是原生协程。（由新语法定义）

(2) `CO_ITERABLECOROUTINE`表示这是用生成器实现的协程，但是和原生协程兼容。（用装饰器`types.coroutine()`装饰过的生成器协程）

- 调用一个普通生成器，返回一个生成器对象（generator object）；相应的，调用一个协程返回一个协程对象（coroutine object）。
- 协程不再抛出`StopIteration`异常，因为抛出的`StopIteration`异常会被包装（wrap）成一个`RuntimeError`异常。（在Python 3.5，对于普通生成器要想这样需要进行`future import`，见PEP 479）。
- 如果一个协程从未`await`等待就被垃圾收集器销毁了，会引发一个`RuntimeWarning`异常（见“调试特性”）

`asyncio`的编程模型就是一个消息循环。我们从`asyncio`模块中直接获取一个`EventLoop`的引用，然后把需要执行的协程扔到`EventLoop`中执行，就实现了异步IO 我们需要一个事件循环。

我们可以通过`asyncio.get_event_loop()`得到一个标准的事件循环，之后使用它的`run_until_complete()`方法来运行协同程序。所以，为了使之之前的协同程序运行，我们只需要做下面的步骤：

```
1 | loop = asyncio.get_event_loop()
2 | loop.run_until_complete(print_page('http://example.com'))
```

```

1  import asyncio
2  import socket
3
4  import aiohttp # <1>
5
6  from flags import BASE_URL, save_flag, show, main # <2>
7
8
9  async def get_flag(client, cc): # <3>
10     url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
11     async with client.get(url) as resp: # <4>
12         assert resp.status == 200
13         return await resp.read() # <5>
14
15
16  async def download_one(client, cc): # <6>
17     image = await get_flag(client, cc) # <7>
18     show(cc)
19     save_flag(image, cc.lower() + '.gif')
20     return cc
21
22
23  async def download_many(loop, cc_list):
24     tcpconnector = aiohttp.TCPConnector(family=socket.AF_INET)
25     async with aiohttp.ClientSession(connector=tcpconnector) as client:
26         # async with aiohttp.ClientSession(loop=loop) as client: # <8>
27         to_do = [download_one(client, cc) for cc in sorted(cc_list)] # <
28         res = await asyncio.gather(*to_do)
29         return len(res) # <10>
30
31
32  def start(cc_list):
33     loop = asyncio.get_event_loop() # <11>
34     res = loop.run_until_complete(download_many(loop, cc_list)) # <12>
35     loop.close() # <13>
36     return res
37
38
39  if __name__ == '__main__':
40     main(start)

```

## async + await

新的await表达式用于获得协程执行的结果：

```

1 | async def read_data(db):
2 |     data = await db.fetch('SELECT ...')
3 |     ...

```

以CPython内部，await使用了yield from的实现，但加入了一个额外步骤——验证它的参数类型。await只接受awaitable对象。

如果在async def函数之外使用await语句，会引发SyntaxError异常。这和def函数之外使用yield语句一样。

如果await右边不是一个awaitable对象，会引发TypeError异常。

```

1 | import asyncio
2 | import datetime
3 |
4 | async def display_date(loop):
5 |     end_time = loop.time() + 5.0
6 |     while True:
7 |         print(datetime.datetime.now())
8 |         if (loop.time() + 1.0) >= end_time:
9 |             break
10 |        await asyncio.sleep(1)
11 |
12 | loop = asyncio.get_event_loop()
13 | # Blocking call which returns when the display_date() coroutine is done
14 | loop.run_until_complete(display_date(loop))
15 | loop.close()

```

## 协程嵌套

```

1 | import asyncio
2 |
3 | async def compute(x, y):
4 |     print("Compute %s + %s ..." % (x, y))
5 |     await asyncio.sleep(1.0)
6 |     return x + y
7 |
8 | async def print_sum(x, y):
9 |     result = await compute(x, y)
10 |    print("%s + %s = %s" % (x, y, result))
11 |
12 | loop = asyncio.get_event_loop()
13 | loop.run_until_complete(print_sum(1, 2))
14 | loop.close()

```

## 异步上下文管理器和“async with”

异步上下文管理器（asynchronous context manager），可以在它的enter和exit方法里挂起、调用异步代码。

为此，我们设计了一套方案，添加了两个新的魔术方法：`_aenter_`和`_aexit_`，它们必须返回一个awaitable。

```
1 class AsyncContextManager:
2     async def __aenter__(self):
3         await log('entering context')
4
5     async def __aexit__(self, exc_type, exc, tb):
6         await log('exiting context')
```

Python

## 数据库异步事务

```
1 async def commit(session, data):
2     ...
3
4     async with session.transaction():
5         ...
6         await session.update(data)
7     ...
```

Python

## async for

一个异步可迭代对象（asynchronous iterable）能够在迭代过程中调用异步代码，而异步迭代器就是能够在next方法中调用异步代码。为了支持异步迭代：

- 1、一个对象必须实现`aiter`方法，该方法返回一个异步迭代器（asynchronous iterator）对象。
- 2、一个异步迭代器对象必须实现`anext`方法，该方法返回一个awaitable类型的值。
- 3、为了停止迭代，`anext`必须抛出一个`StopAsyncIteration`异常。

## 调试

新手在使用协程时可能忘记使用yield from语句，比如：

```
1 @asyncio.coroutine
2 def useful():
3     asyncio.sleep(1) # 前面忘写yield from, 所以程序在这里不会挂起1秒
```

Python

在asyncio里，对于此类错误，有一个特定的调试方法。装饰器 `@coroutine` 用一个特定的对象包装

(wrap) 所有函数，这个对象有一个析构函数 (destructor) 用于记录警告信息。无论何时，一旦被装饰过的生成器被垃圾回收，会生成一个详细的记录信息（具体哪个函数、回收时的stack trace等等）。包装对象提供一个\_\_repr\_\_方法用来输出关于生成器的详细信息。

如何启用这些调试工具，由于这些调试工具在生产模式里什么也不做，比如 @coroutine必须是在系统变量PYTHONASYNCIODEBUG出现时才具有调试功能。这时可以给asyncio程序进行如下设置：EventLoop.set\_debug(true)，这时使用另一套调试工具，对 @coroutine的行为没有影响。