asyncio 的概念概述

这篇 指南 旨在帮助您充分理解 `asyncio` 的基本运作原理，并使您理解推荐模式背后的原理和原因。

你可能会对某些关键的 `asyncio` 概念感到好奇。 读完本文后，你将能够轻松地回答这些问题：

- 当一个对象被等待时，幕后发生了什么？
- `asyncio` 如何区分不需要 CPU 时间的任务（如网络请求或文件读取）和与之相反的任务（如计算 n 的阶乘）？
- 如何编写一个操作的异步变体，例如异步的休眠或数据库请求。

参见:

- 启发这篇指南文章的 指南，作者是 Alexander Nordin。
- 这套深入讲解 `asyncio` 的 YouTube 教程系列，由 Python 核心团队成员 Łukasz Langa 制作。
- 500 Lines or Less: A Web Crawler With asyncio Coroutines，作者是 A. Jesse Jiryu Davis 和 Guido van Rossum。

概念概述第 1 部分：高层次

在第 1 部分中，我们将介绍主要的、高层级的 `asyncio` 构成部分：事件循环、协程函数、协程对象、任务和 `await`。

事件循环

`asyncio` 中的一切都与事件循环相关。它是演出的主角。它就像一名乐队指挥一样在幕后管理资源。它掌握着一些权力，但它完成工作的能力很大程度上来自于它的工蜂们的尊重与合作。

用更专业的术语来说，事件循环包含一组待运行的作业。 有些作业是由你直接添加的，有些则是由 `asyncio` 间接添加的。 事件循环会从其待处理事项中取出一个作业并唤起它（或称“给予其控制权”），类似于调用一个函数，然后该作业就会运行。 一旦它暂停或完成，它会将控制权返回给事件循环。 然后事件循环会从作业池中选择另一个作业并唤起它。 你可以 *粗略地* 将这组作业视为一个队列：作业被添加然后被逐个处理，通常（但不总是）按顺序进行。 此过程将无限地重复，事件循环也不停地循环下去。 如果没有待执行的作业，事件循环会足够智能地转入休息状态以避免浪费 CPU 周期，并在有更多工作需完成时恢复运行。

有效的执行依赖于作业的良好共享和合作；一个贪婪的作业可能会霸占控制权，让其他作业陷入饥饿，从而使整个事件循环机制变得毫无用处。

```
import asyncio

# 这会创建一个事件循环并无限循环地执行其作业集合。
```

```
event_loop = asyncio.new_event_loop()
event_loop.run_forever()
```

异步函数和协程

这是一个基本的、无趣的Python 函数：

```
def hello_printer():
    print(
        "Hi, I am a lowly, simple printer, though I have all I "
        "need in life -- \nfresh paper and my dearly beloved octopus "
        "partner in crime."
    )
```

调用一个普通函数会执行它的逻辑或函数体：

```
>>> hello_printer()
Hi, I am a lowly, simple printer, though I have all I need in life --
fresh paper and my dearly beloved octopus partner in crime.
```

与普通的 `def` 不同，async def 使它成为一个异步函数（或"协程函数"）。调用它会创建并返回一个协程 对象。

```
async def loudmouth_penguin(magic_number: int):
    print(
     "I am a super special talking penguin. Far cooler than that printer. "
     f"By the way, my lucky number is: {magic_number}."
    )
```

调用异步函数 `loudmouth_penguin` 不会执行打印语句； 相反，它会创建一个协程对象：

```
>>> loudmouth_penguin(magic_number=3)
<coroutine object loudmouth_penguin at 0x104ed2740>
```

"协程函数"和"协程对象"这两个术语经常被统称为协程。这可能会引起混淆！在本文中，协程特指协程 对象，或者更准确地说，是 `types.CoroutineType` 的实例 （原生协程 ）。请注意，协程也可以作为 `collections.abc.Coroutine` 的实例存在——这一点对于类型检查来说很重要。

协程代表函数体或逻辑。协程必须显式启动；再次强调，仅仅创建协程并不能启动它。值得注意的是，协程可以在函数体的不同位置暂停和恢复。这种暂停和恢复能力使得异步行为成为可能！

协程和协程函数是利用 生成器 和 生成器函数 构建的。回想一下，生成器函数是一个会 `yield` 的函数，就像这样：

```
def get_random_number():
    # 这是一个糟糕的随机数生成器！
    print("Hi")
    yield 1
    print("Hello")
    yield 7
    print("Howdy")
    yield 4
    ...
```

与协程函数类似，调用生成器函数并不会运行该函数，而是创建一个生成器对象：

```
>>> get_random_number()
<generator object get_random_number at 0x1048671c0>
```

你可以通过内置函数 next() 执行生成器到下一个 yield。换句话说，生成器运行，然后暂停。例如：

```
>>> generator = get_random_number()
>>> next(generator)
Hi
1
>>> next(generator)
Hello
7
```

任务

粗略地说，任务 是绑定到事件循环的协程（而非协程函数）。任务还维护一个回调函数列表，这些回调函数的重要性在稍后讨论 await 时会更加清晰。推荐使用 asyncio.create_task() 创建任务。

创建任务会自动安排它的执行（通过在事件循环的待办事项列表（即作业集合）中添加回调函数来运行它）。

由于（每个线程中）只有一个事件循环，asyncio 会帮你把任务与事件循环关联起来。因此，你无需指定事件循环。

```
coroutine = loudmouth_penguin(magic_number=5)
# 这将创建一个 Task 对象并通过事件循环安排其执行。
task = asyncio.create_task(coroutine)
```

之前，我们手动创建了事件循环并将其设置为永久运行。实际上，推荐（且常见）的做法是使用 asyncio.run()，它负责管理事件循环并确保提供的协程在继续执行之前结束。例如，许多异步程序都遵循以下设置：

```
import asyncio

async def main():
    # 执行各种稀奇古怪、天马行空的异步操作……
    ...

if __name__ == "__main__":
    asyncio.run(main())
    # 直到协程 main() 结束，程序才会到达下面的打印语句。
    print("coroutine main() is done!")
```

需要注意的是，任务本身不会被添加到事件循环中，只有任务的回调函数才会被添加到事件循环中。如果你创建的任务对象在被事件循环调用之前就被垃圾回收了，这就会产生问题。例如，考虑这个程序：

```
1 async def hello():
2     print("hello!")
3
4 async def main():
5     asyncio.create_task(hello())
6     # 其他异步指令运行一段时间并将控制权交还给事件循环......
7     ...
8
9 asyncio.run(main())
```

由于没有对第 5 行创建的任务对象的引用，它 *可能* 在事件循环调用它之前就被垃圾回收了。协程 `main()` 中的后续指令将控制权交还给事件循环，以便它可以调用其他作业。当事件循环最终尝试运行该任务时，它可能会失败并发现任务对象不存在！即使协程持有对某个任务的引用，但如果协程在该任务结束之前就完成了，也可能发生这种情况。当协程退出时，局部变量超出范围，可能被垃圾回收。实际上，`asyncio` 和 Python 的垃圾回收器会非常努力地确保此类事情不会发生。但这并不是鲁莽行事的理由！

await

`await` 是一个 Python 关键字，通常以两种不同的方式使用：

```
await task
await coroutine
```

从关键方面来说，`await` 的行为取决于所等待对象的类型。

等待任务会将控制权从当前任务或协程交还给事件循环。在交还控制权的过程中，会发生一些重要的事情。我们将使用以下代码示例来说明：

```
async def plant_a_tree():
    dig_the_hole_task = asyncio.create_task(dig_the_hole())
    await dig_the_hole_task

    # 与植树相关的其他指令。
    ...
```

在这个例子中，假设事件循环已经将控制权交给了协程 `plant_a_tree()` 的开始部分。如上所示，协程创建了一个任务，然后对其执行了 await。`await dig_the_hole_task` 这条指令会将一个回调函数（用于恢复 `plant_a_tree()` 的执行）添加到 `dig_the_hole_task` 对象的回调函数列表中。随后，这条指令将控制权交还给事件循环。过一段时间后，事件循环会将控制权传递给 `dig_the_hole_task`，该任务会完成它需要做的工作。一旦任务结束，它会将它的各种回调函数添加到事件循环中，在这里是恢复 `plant_a_tree()` 的执行。

一般来说，当等待的任务完成时（`dig_the_hole_task`），原先的任务或协程（`plant_a_tree()`）将被添加回事件循环的待办列表以便恢复运行。

这是一个基础但可靠的思维模型。实际操作中，控制权交接会稍微复杂一些，但不会复杂太多。在第 2 部分中，我们将逐步讲解实现这一目标的细节。

与任务不同，等待协程并不会将控制权交还给事件循环！ 先将协程包装到任务中，然后再等待，会导致控制权交还。`await coroutine` 的行为实际上与调用常规的同步 Python 函数相同。考虑以下程序：

```
import asyncio

async def coro_a():
   print("I am coro_a(). Hi!")

async def coro_b():
   print("I am coro_b(). I sure hope no one hogs the event loop...")

async def main():
   task_b = asyncio.create_task(coro_b())
   num_repeats = 3
   for _ in range(num_repeats):
      await coro_a()
   await task_b

asyncio.run(main())
```

协程 `main()` 中的第一条语句创建了 `task_b` 并调度它通过事件循环运行。 然后，将重复地等待 `coro_a()`。 控制权从未被交还给事件循环，这就是为什么在 `coro_b()` 的输出之前我们会看到所有三次唤起 `coro_a()` 的输出。invocations before

```
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_b(). I sure hope no one hogs the event loop...
```

如果我们将 `await coro_a()` 改为 `await asyncio.create_task(coro_a())`，行为就会发生变化。协程 `main()` 会通过该语句将控制权交还给事件循环。然后，事件循环会继续处理其积压的工作，先调用 `task_b`，然后调用包装 `coro_a()` 的任务，最后恢复协程 `main()`。

```
I am coro_b(). I sure hope no one hogs the event loop...
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
```

这种 `await coroutine` 的行为可能会困扰很多人！ 这个例子强调了仅使用 `await coroutine` 可能会无意中霸占其他任务的控制权并在实际上阻滞事件循环。 `asyncio.run()` 可以通过 `debug=True` 旗标来检测这种情况，它将会启用 调试模式。 此外，它还会记录任何独占执行时间 100 毫秒以上的协程。

该设计有意牺牲了 `await` 用法的某些概念明晰度以提升性能。 每当有任务被等待时，控制权都需要沿着调用栈一路向上传递到事件循环。 这听起来可能微不足道，但在一个具有大量 `await` 语句和深度调用栈的大型程序中，这种开销可能会累积到明显拖累性能。

概念概述第 2 部分：核心细节与运作机制

第 2 部分将详细介绍 asyncio 用于管理控制流的机制。这正是魔法发生的地方。读完本节后，您将了解 `await` 在幕后做了什么，以及如何创建您自己的异步运算符。

协程的内部工作原理

asyncio 利用四个组件来传递控制权。

coroutine.send(arg) 是用于启动或恢复协程的方法。 如果协程已暂停并正在被恢复，则参数 `arg` 将作为原先暂停它的 `yield` 语句的返回值被发送。 如果协程是首次被使用（而不是被恢复），则 `arg` 必须为 `None`。

```
 1 class Rock:
 2     def __await__(self):
 3         value_sent_in = yield 7
 4         print(f"Rock.__await__ resuming with value: {value_sent_in}.")
 5         return value_sent_in
 6
 7 async def main():
 8     print("Beginning coroutine main().")
 9     rock = Rock()
10     print("Awaiting rock...")
11     value_from_rock = await rock
12     print(f"Coroutine received value: {value_from_rock} from rock.")
13     return 23
14
15 coroutine = main()
16 intermediate_result = coroutine.send(None)
17 print(f"Coroutine paused and returned intermediate value: {intermediate_result}
18
19 print(f"Resuming coroutine and sending in value: 42.")
20 try:
21     coroutine.send(42)
22 except StopIteration as e:
23     returned_value = e.value
24 print(f"Coroutine main() finished and provided value: {returned_value}.")
```

yield 像往常一样暂停执行并将控制权返回给调用者。 在上面的例子中，第 3 行的 `yield` 被第 11 行的 `... = await rock` 调用。 更宽泛地说，`await` 会调用给定对象的 __await__() 方法。 `await` 还会做一件非常特别的事情：它会将接收到的任何 `yield` 沿着调用链向上传播（或称“传递”）。 在本例中，这将回到第 16 行的 `... = coroutine.send(None)`。

协程通过第 21 行的 `coroutine.send(42)` 调用恢复。协程从第 3 行 `yield` (或暂停) 的位置继续执行，并执行其主体中的剩余语句。协程完成后，它会引发一个 StopIteration 异常，并将返回值附加在 value 属性中。

该代码片段产生以下输出：

```
Beginning coroutine main().
Awaiting rock...
Coroutine paused and returned intermediate value: 7.
Resuming coroutine and sending in value: 42.
Rock.__await__ resuming with value: 42.
```

```
Coroutine received value: 42 from rock.
Coroutine main() finished and provided value: 23.
```

这里值得暂停一下，确保您已经理解了控制流和值传递的各种方式。我们涵盖了很多重要的概念，确保您理解得足够牢固。

从协程中“yield”（或有效地放弃控制权）的唯一方法是 `await` 一个在其 `__await__` 方法中 `yield` 的对象。这听起来可能有点奇怪。你可能会想：

1. What about a `yield` directly within the coroutine function? The coroutine function becomes an async generator function, a different beast entirely.

2. What about a yield from within the coroutine function to a (plain) generator? That causes the error: `SyntaxError: yield from not allowed in a coroutine.` This was intentionally designed for the sake of simplicity -- mandating only one way of using coroutines. Initially `yield` was barred as well, but was re-accepted to allow for async generators. Despite that, `yield from` and `await` effectively do the same thing.

Future

Future 是一个用来表示计算状态和结果的对象。该术语指的是尚未发生的事情，而 Future 对象则是一种用来关注这些事情的方式。

Future 对象有几个重要的属性。 其一是它的状态，可以是“待处理”、“已取消”或“已完成”。 其二是它的结果，当状态转换为已完成时它就会被设定。 与协程不同，Future 并不代表要执行的实际计算；相反，它代表该计算的状态和结果，有点像一个状态灯（红色、黄色或绿色）或指示器。

为了获得这些功能，`asyncio.Task` 继承了 `asyncio.Future` 类。上一节提到任务存储了一个回调函数列表，这并不完全准确。实际上，实现这些逻辑的是 `Future` 类，而 `Task` 继承了它。

Future 也可以被直接使用（无需通过任务）。任务会在协程完成后将自身标记为已完成。而 Future 的功能更加多样，由你来指定它何时标记为已完成。因此，Future 是一个灵活的接口，您可以自定义等待和恢复的条件。

自制 asyncio.sleep

我们将通过一个例子来说明如何利用 Future 来创建自己的异步睡眠变体（`async_sleep`），模仿了 `asyncio.sleep()`。

这个代码段在为事件循环注册了一些任务然后等待由 `asyncio.create_task` 创建的任务，它包装在 `async_sleep(3)` 协程中。 我们希望该任务在三秒之后才结束，但不会阻止其他任务的运行。

```python
async def other_work():
    print("I like work. Work work.")

async def main():
    # 向事件循环添加一些其他任务，这样在异步休眠时就可以做一些事情。
    work_tasks = [
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work()),
```

```
        asyncio.create_task(other_work())
    ]
    print(
        "Beginning asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    await asyncio.create_task(async_sleep(3))
    print(
        "Done asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    # asyncio.gather 有效地等待集合中的每个任务。
    await asyncio.gather(*work_tasks)
```

下面，我们使用 Future 来自定义控制何时将任务标记为已完成。如果 future.set_result() <asyncio.Future.set_result>`（负责将该 Future 标记为已完成的方法）从未被调用，那么该任务将永远不会结束。我们还借助了另一个任务（稍后会看到），它将监视已过去的时间，并相应地调用 ``future.set_result()`()。

```
async def async_sleep(seconds: float):
    future = asyncio.Future()
    time_to_wake = time.time() + seconds
    # 将监视任务添加到事件循环。
    watcher_task = asyncio.create_task(_sleep_watcher(future, time_to_wake))
    # 阻塞直到 future 被标记为已完成。
    await future
```

下面，我们将使用一个相当简单的 YieldToEventLoop() 对象从其 __await__ 方法中 yield，将控制权交还给事件循环。 这实际上与调用 asyncio.sleep(0) 相同，但这种方式更为明晰，更不用说在展示如何实现 asyncio.sleep 时直接使用它有点作弊！

与往常一样，事件循环会轮番处理其任务，给予它们控制权并在它们暂停或完成时收回控制权。 运行 _sleep_watcher(...) 协程的 watcher_task 将在事件循环的每个完整周期中被唤起一次。 在每次恢复时，它将检查时间，如果经过的时间不够，则会再次暂停并将控制权交还给事件循环。 一旦经过了足够的时间，_sleep_watcher(...) 会将该 Future 标记为已完成并通过退出无限的 while 循环来结束执行。 鉴于这个辅助任务在事件循环的每个周期中只会被唤起一次，因此你应该注意到这个异步休眠将 *至少* 休眠三秒，而不是恰好三秒。 请注意 asyncio.sleep 也同样如此。

```
class YieldToEventLoop:
    def __await__(self):
        yield

async def _sleep_watcher(future, time_to_wake):
    while True:
        if time.time() >= time_to_wake:
            # 这标记 future 为已完成。
            future.set_result(None)
            break
        else:
            await YieldToEventLoop()
```

以下是程序的完整输出：

```
$ python custom-async-sleep.py
Beginning asynchronous sleep at time: 14:52:22.
I like work. Work work.
I like work. Work work.
I like work. Work work.
Done asynchronous sleep at time: 14:52:25.
```

你可能会觉得这种异步睡眠的实现过于复杂。确实如此。这个例子旨在通过一个简单的示例来展示 Future 的多功能性，以便可以模仿更复杂的需求。作为参考，你可以不使用 Future 来实现它，如下所示：

```
async def simpler_async_sleep(seconds):
    time_to_wake = time.time() + seconds
    while True:
        if time.time() >= time_to_wake:
            return
        else:
            await YieldToEventLoop()
```

目前就说这些了。 希望你已准备好更自信地深入探索异步编程或是查看 文档其余部分 中的进阶主题。