楔子

上一篇文章我们深入讨论了并发性，探讨了如何同时使用进程和线程实现并发，还探索了如何利用非阻塞 IO 和事件循环来实现只使用一个线程的并发性。本篇文章将介绍在 asyncio 中使用单线程并发模型编写程序的基础知识，使用本文中的技术，你将能执行长时间运行的操作，如 Web 请求、数据库查询和网络连接，并串联执行它们。

我们将了解更多关于协程构造，以及如何使用 async、await 语法来定义和运行协程。还将研究如何通过使用任务来并发运行协程，并通过创建可重用的计时器来检查并发节省的时间。最后，我们再来了解软件工程师使用 asyncio 时常犯的错误，以及如何使用调试模式来发现这些问题。

关于协程

可将协程想象成一个普通的 Python 函数，但它具有一个超能力：在遇到可能需要一段时间才能完成的操作时，能够暂停执行。当长时间运行的操作完成时，可唤醒暂停的协程，并执行该协程中的其他代码。当一个暂停的协程正在等待操作完成时，可运行其他代码，等待时其他代码的运行是应用程序并发的原因。还可同时运行多个耗时的操作，这能大大提高应用程序的性能。

要创建和暂停协程我们需要学习使用 Python 的 async 和 await 关键字，def 定义一个普通函数，调用之后直接执行；而 async def 会定义一个协程函数，调用之后得到协程。当有一个长时间运行的操作时，await 关键字可以让我们暂停协程。

使用 async 关键字创建协程

创建协程很简单，与创建普通 Python 函数没有太大区别。唯一的区别是，创建协程时不使用 def 关键字，而是使用 async def。async 关键字将函数标记为协程协程函数，而不是普通的 Python 函数。

```
async def coroutine():
    print("hello world")
```

这是一个简单的协程函数，不执行任何长时间的操作，它只是输出信息并返回。这意味着，将协程放在事件循环中时，它将立即执行，因为没有任何阻塞I/O，没有任何操作暂停执行。

```
async def coroutine_add_one(number):
    return number + 1

def add_one(number):
```

```
    return number + 1

function_result = add_one(1)
coroutine_result = coroutine_add_one(1)
print(function_result)
print(type(function_result))
"""
2
<class 'int'>
"""
print(coroutine_result)
print(type(coroutine_result))
"""
<coroutine object coroutine_add_one at 0×000002977045BAC0>
<class 'coroutine'>
"""
```

调用普通的 add_one 函数时，它会立即执行并返回我们期望的一个整数。但当调用 coroutine_add_one 时，并不会执行协程中的代码，而是得到一个协程对象。这一点很重要，因为当直接调用协程函数时，协程不会被执行。相反，它创建了一个可以稍后执行的协程对象，要执行协程，需要在事件循环中显式执行它。那么如何创建个事件循环并执行协程呢?

在 Python3.7 之前的版本中，如果不存在事件循环，必须创建一个事件循环。但 asyncio 库添加了几个抽象事件循环管理的函数，有一个方便的函数 asyncio.run，我们可以使用它来运行协程。

```
import asyncio

async def coroutine_add_one(number):
    return number + 1

coroutine_result = asyncio.run(coroutine_add_one(1))
print(coroutine_result)  # 2
```

正如我们期望的一样，我们已经正确地将协程放在事件循环中，并且已经执行了它。

asyncio.run 在这种情况下完成了一些重要的事情，首先创建了一个全新的事件循环。一旦成功创建，就会接受我们传递给它的任何协程，并运行它直到完成，然后返回结果。此函数还将对主协程完成后可能继续运行的内容进行清理，一切完成后，它会关闭并结束事件循环。

关于 asyncio.run 最重要的一点是，它旨在成为我们创建的 asyncio 应用程序的主要入口点。但我们也可以手动创建一个事件循环，然后运行协程，后面会说。

使用 await 关键字暂停执行

我们上面的例子中没有任何非阻塞代码，所以也不一定非要使用协程，定义成普通函数也是可以的。asyncio 的真正优势是能暂停执行，让事件循环在长时间运行的操作期间，运行其他任务。要暂停执行，可使用 await 关键字，await 关键字之后通常会调用协程（更具体地说是一个被称为 awaitable 的对象，它并不总是协程，我们将在后续学习中了解关于 awaitable 的更多内容）。

使用 await 关键字将导致它后面的协程运行，这与直接调用协程不同，因为直接调用只会产生一个协程对象。await 表达式也会暂停它所在的协程，直到等待的协程完成并返回结果。等待的协程完成时，将访问它返回的结果，并唤醒 await 所在的协程。

```
import asyncio

async def add_one(number):
    return number + 1

async def main():
    # main() 协程将暂停执行，直到 add_one(1) 运行完毕
    one_plus_one = await add_one(1)
    # main() 协程将暂停执行，直到 add_one(2) 运行完毕
    two_plus_one = await add_one(2)

    print(one_plus_one)
    print(two_plus_one)

asyncio.run(main())
"""
2
3
"""
```

在上面的代码中，我们两次暂停执行。首先等待对 add_one(1) 的调用，一旦得到结果，主函数将取消暂停并将 add_one(1) 的返回值分配给变量 one_plus_one。然后对 add_one(2) 执行相同的操作，并输出结果。我们来应用程序的执行流程可视化一样，如下图所示，图中的每个块代表一行或多行代码在任何给定时刻发生的事情。

使用 sleep 引入长时间运行的协程

之前的例子没有使用任何运行时间较长的操作，主要用来帮助我们学习协程的基本语法。为充分了解协程的优势，并展示如何同时运行多个事件，需要引入一些长时间运行的操作。我们不会立即进行 Web API 或数据库查询，这对于它们将花费多少时间是不确定的，我们会通过指定想要等待的时间来模拟长时间运行的操作。而实现这一点，可以通过 asyncio.sleep 函数。

使用 asyncio.sleep 让协程休眠给定的秒数。这将在预定的时间内暂停协程，模拟对数据库或 Web API 进行长时间运行的调用情况。

由于 asyncio.sleep 本身是一个协程，所以必须将它与 await 关键字一起使用，如果单独调用它，会得到一个协程对象。既然 asyncio.sleep 是一个协程，这意味着当协程等待它时，其他代码也能够运行。

```
import asyncio

async def hello_world():
    # 暂停 hello_world 协程一秒钟
    await asyncio.sleep(1)
    return "hello world"

async def main():
    # 暂停 main 协程，直到 hello_world 协程运行完毕
    message = await hello_world()
    print(message)

asyncio.run(main())
"""
hello world
"""
```

运行这个应用程序时，程序将等待 1 秒钟，然后输出打印信息。由于 hello_world 是一个协程，使用 asyncio.sleep 将其暂停 1 秒，因此现在有 1 秒的时间可以同时运行其他代码。

```python
import asyncio

async def delay(seconds):
    print(f"开始休眠 {seconds} 秒")
    await asyncio.sleep(seconds)
    print(f"休眠完成")
    return seconds

async def add_one(number):
    return number + 1

async def hello_world():
    await delay(1)
    return "hello world"

async def main():
    # 暂停 main()，直到 add_one(1) 返回
    one_plus_one = await add_one(1)
    # 暂停 main()，直到 hello_world() 返回
    message = await hello_world()
    print(one_plus_one)
    print(message)

asyncio.run(main())
"""
开始休眠 1 秒
休眠完成
2
hello world
"""
```

在 main 协程里面分别通过 await 驱动 add_one(1) 和 hello_world() 两个协程执行，然后打印它们的返回值，但是在打印 one_plus_one 之前需要等待一秒，因为在 hello_world() 协程里面 sleep 了一秒。但我们真正想要的结果是，在 await sleep 的时候，立刻执行其它的代码，比如立刻打印 one_plus_one，但实际情况却没有。

这是为什么呢？答案是在 await 暂停当前的协程之后、以及 await 表达式给我们一个值之前不会执行该协程中的其他任何代码。因为 hello_world_message 函数需要 1 秒后才能给出一个值，所以主协程将暂停 1 秒。这种情况下，代码表现得好像它是串行的。

事实上从源代码本身也能够理解，因为代码是一行一行写的，所以自然也要一行一行执行。而 await 后面跟一个协程之后，会驱动协程执行，并等到驱动的协程运行完毕之后才往下执行。因此这个逻辑就决定了，await 是串行的，一个 await 执行完毕之后才能执行下一个 await。如果我们想摆脱这种顺序模型，同时运行 add_one 和 hello_world，那么需要引入一个被称为“任务”的概念。

通过任务实现并行

前面我们看到，直接调用协程时，并没有把它放在事件循环中运行，相反会得到一个协程对象。如果想运行，要么通过 asyncio.run，要么在一个协程里面通过 await 关键字进行驱动（在 A 协程里面 await B 协程，如果 A 协程运行了，那么 B 协程也会被驱动）。虽然通过这些工具，可编写异步代码，但不能同时运行任何东西，要想同时运行协程，需要将它包装成任务。

任务是协程的包装器，它安排协程尽快在事件循环上运行，并提供一系列的方法来获取协程的运行状态和返回值。这种调度和执行以非阻塞方式发生，这意味着一旦创建一个任务，那么任务就会立刻运行。并且由于是非阻塞的，我们可以同时运行多个任务，举个例子。

创建任务

创建任务是通过 asyncio.create_task 函数来实现的，当调用这个函数时，需要给它传递一个协程，然后返回一个任务对象。一旦有了一个任务对象，就可以把它放在一个 await 表达式中，它完成后就会提取返回值。

```python
import asyncio

async def delay(seconds):
    print(f"开始休眠 {seconds} 秒")
    await asyncio.sleep(seconds)
    print(f"休眠完成")
    return seconds

async def main():
    # 将 delay(3) 包装成任务，注：包装完之后直接就丢到事件循环里面运行了
    # 因此这里会立即返回，而返回值是一个 asyncio.Task 对象
```

```
    sleep_for_three = asyncio.create_task(delay(3))
    print("sleep_for_three:", sleep_for_three.__class__)
    # 至于协程究竟有没有运行完毕，我们可以通过 Task 对象来查看
    # 当协程运行完毕或者报错，都看做是运行完毕了，那么调用 Task 对象的 done 方法会返
回 True
    # 否则返回 False，由于代码是立即执行，还没有到 3 秒钟，因此打印结果为 False
    print("协程(任务)是否执行完毕:", sleep_for_three.done())
    # 这里则保证必须等到 Task 对象里面的协程运行完毕后，才能往下执行
    result = await sleep_for_three
    print("协程(任务)是否执行完毕:", sleep_for_three.done())
    print("返回值:", result)

asyncio.run(main())
"""
sleep_for_three: <class '_asyncio.Task'>
协程(任务)是否执行完毕: False
开始休眠 3 秒
休眠完成
协程(任务)是否执行完毕: True
返回值: 3
"""
```

如果我们直接 `await delay(3)`，那么在打印之前需要至少等待 3 秒，但通过将它包装成任务，会立即扔到事件循环里面运行。此时主程序可以直接往下执行，至于协程到底什么时候执行完毕、有没有执行完毕，则通过 `Task` 对象（任务）来查看。当然你也可以 `await` 一个 `Task` 对象，保证里面的协程运行完毕后才能往下执行。

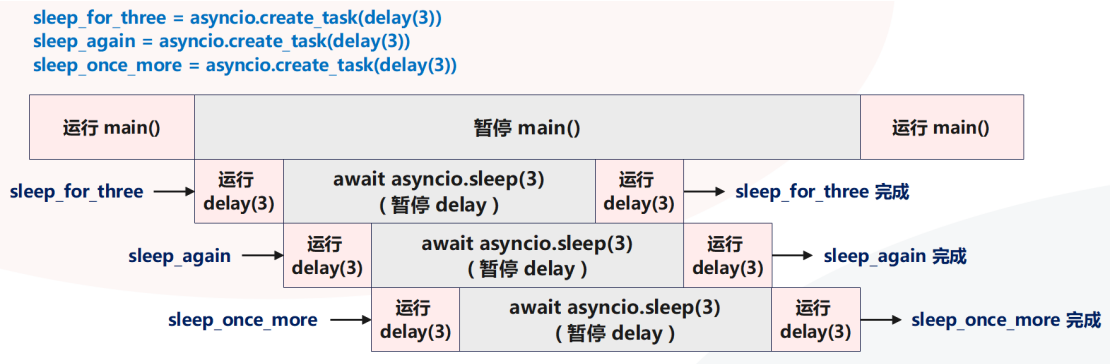
同时运行多个任务

鉴于任务是立即创建并计划尽快运行，这允许同时运行许多长时间的任务。

```
import asyncio

async def delay(seconds):
    print(f"开始休眠 {seconds} 秒")
    await asyncio.sleep(seconds)
    print(f"休眠完成")
    return seconds

async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))
    sleep_once_more = asyncio.create_task(delay(3))
```

```
    await sleep_for_three
    await sleep_again
    await sleep_once_more

asyncio.run(main())
"""
开始休眠 3 秒
开始休眠 3 秒
开始休眠 3 秒
休眠完成
休眠完成
休眠完成
"""
```

在上面的代码中启动了三个任务，每个任务需要 3 秒才能完成。但由于对 `create_task` 的每次调用都会立即返回，因此会立即到达 `await sleep_for_three` 语句，并且三个任务都丢到了事件循环，开启执行。由于 `asyncio.sleep` 属于 IO，因此会进行切换，所以三个任务是并发执行的，这也意味着整个程序会在 3 秒钟左右完成，而不是 9 秒钟。



随着我们添加更多任务，性能提升效果会更明显，比如启动了 10 个这样的任务，仍然只需要大约 3 秒，从而使速度提高 10 倍，再来看个例子：

```
import asyncio

async def delay(seconds):
    print(f"开始休眠 {seconds} 秒")
    await asyncio.sleep(seconds)
    print(f"休眠完成")
    return seconds

async def hello_from_second():
    for i in range(10):
```

```
        await asyncio.sleep(1)
        print("你好，我每秒钟负责打印一次")

async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))

    await hello_from_second()

asyncio.run(main())
"""
开始休眠 3 秒
开始休眠 3 秒
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
休眠完成
休眠完成
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
你好，我每秒钟负责打印一次
"""
```

一旦协程被包装成任务，那么运行就开始了（被丢到事件循环当中），而主程序依旧可以往下执行。然后执行 `await hello_from_second()`，此时程序会阻塞在这里，不管 `await` 后面跟的是协程对象还是基于协程封装的 `Task` 对象（任务），它都要求 `await` 后面的对象运行完毕并返回一个值之后，才能继续往下执行。

最终结果就如打印的那样，但需要注意的是：我们不能这样写。

```
async def main():
    await hello_from_second()

    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))
```

如果是这种方式的话，那么必须等到 `hello_from_second()` 运行完毕后，下面的两个任务才能执行，因为 `await` 是阻塞的。

同理下面的编写方式也不行：

```
async def main():
    sleep_for_three = await asyncio.create_task(delay(3))
    sleep_again = await asyncio.create_task(delay(3))

    await hello_from_second()
```

还是那句话，协程被包装成 Task 对象的时候就已经开始运行了，你可以让主程序继续往下执行，也可以使用 await 让主程序等它执行完毕，就像这段代码一样。但很明显，此时就相当于串行了，无法达到并发的效果。

最佳实践：在实际工作中，不要直接 await 一个协程，而是将协程包装成任务来让它运行。当你的代码逻辑依赖某个任务的执行结果时，再对该任务执行 await，拿到它的返回值。

取消任务和设置超时

网络连接可能不可靠，用户的连接可能因为网速变慢而中断，或者网络服务器崩溃导致现有的请求无法处理。因此对于发出的请求，需要特别小心，不要无限期地等待。如果无限期等待一个不会出现的结果，可能导致应用程序挂起，从而导致精糕的用户体验。

在之前的示例中，如果任务一直持续下去，我们将被困在等待 await 语句完成而没有反馈的情况，也没有办法阻止这样的事情发生。因此 asyncio 提供了一个机制，允许我们手动取消任务，或者超时之后自动取消。

取消任务

取消任务很简单，每个任务对象都有一个名为 cancel 的方法，可以在想要停止任务时调用它。取消一个任务将导致该任务在执行 await 时引发 CancelledError，然后再根据需要处理它。

为说明这一点，假设启动了一个长时间运行的任务，但我们不希望它运行的时间超过 5 秒。如果任务没有在 5 秒内完成，就可以停止该任务，并向用户报告：该任务花费了太长时间，我们正在停止它。我们还希望每秒钟都输出一个状态更新，为用户提供最新信息，这样就可以让用户了解任务的运行状态。