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。