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。

```python
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。

```
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。

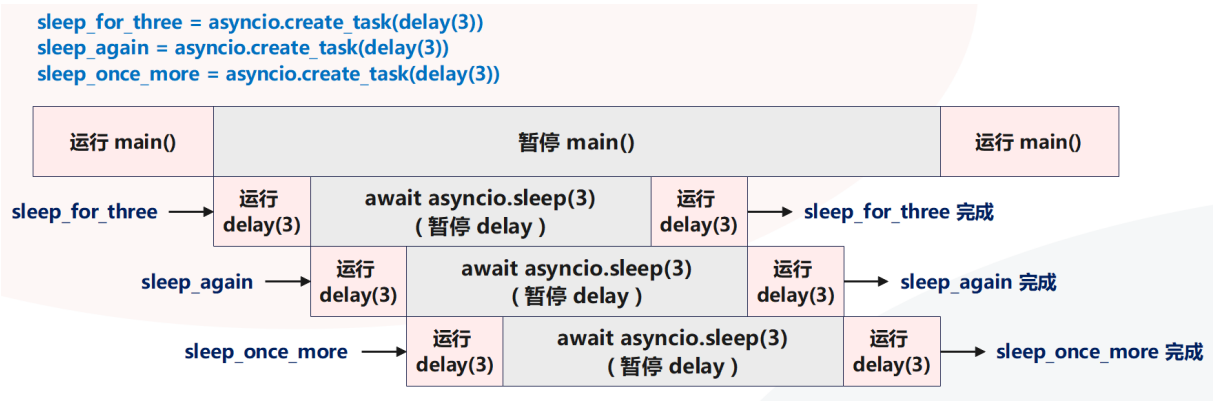
同时运行多个任务

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。

```
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
```

```python
async def main():
    long_task = asyncio.create_task(delay(10))
    seconds_elapsed = 0

    while not long_task.done():
        print("检测到任务尚未完成，一秒钟之后继续检测")
        await asyncio.sleep(1)
        seconds_elapsed += 1
        # 时间超过 5 秒，取消任务
        if seconds_elapsed == 5:
            long_task.cancel()

    try:
        # 等待 long_task 完成，显然执行到这里的时候，任务已经被取消
        # 不管是 await 一个已经取消的任务，还是 await 的时候任务被取消
        # 都会引发 asyncio.CancelledError
        await long_task
    except asyncio.CancelledError:
        print("任务被取消")

asyncio.run(main())
"""
检测到任务尚未完成，一秒钟之后继续检测
开始休眠 10 秒
检测到任务尚未完成，一秒钟之后继续检测
检测到任务尚未完成，一秒钟之后继续检测
检测到任务尚未完成，一秒钟之后继续检测
检测到任务尚未完成，一秒钟之后继续检测
检测到任务尚未完成，一秒钟之后继续检测
任务被取消
"""
```

在代码中我们创建了一个任务，它需要花费 10 秒的时间才能运行完成。然后创建一个 while 循环来检查该任务是否已完成，任务的 done 方法在任务完成时返回 True，否则返回 False。每一秒，我们检查任务是否已经完成，并记录到目前为止经历了多少秒。如果任务已经花费了 5 秒，就取消这个任务。然后来到 await long_task，将输出“任务被取消"，这表明捕获了一个 CancelledError。

关于取消任务需要注意的是，CancelledError 只能从 await 语句抛出。这意味着如果在任务在执行普通 Python 代码时被取消，那么该代码将一直运行，直到触发下一个 await 语句（如果存在），才能引发 CancelledError。

```python
import asyncio
```

```python
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():
    long_task = asyncio.create_task(delay(3))
    # 立刻取消
    long_task.cancel()
    # 但 CancelledError 只有在 await 取消的协程时才会触发
    # 所以下面的语句会正常执行
    print("我会正常执行")
    print("Hello World")
    print(list(range(10)))
    await asyncio.sleep(5)
    try:
        # 引发 CancelledError
        await long_task
    except asyncio.CancelledError:
        print("任务被取消")

asyncio.run(main())
"""
我会正常执行
Hello World
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
任务被取消
"""
```

但是注意：如果任务在取消的时候已经运行完毕了，那么 await 的时候就不会抛 CancelledError 了。

```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():
    long_task = asyncio.create_task(delay(3))
```

```
    await asyncio.sleep(5)
    # 显然执行到这里，任务已经结束了
    long_task.cancel()
    try:
        await long_task
        print("任务执行完毕")
    except asyncio.CancelledError:
        print("任务被取消")

asyncio.run(main())
"""
开始休眠 3 秒
休眠完成
任务执行完毕
"""
```

所以对一个已完成的任务调用 cancel 方法，没有任何影响。

设置超时并使用 wait_for 执行取消

每秒（或其他时间间隔）执行检查然后取消任务，并不是处理超时的最简单方法。理想情况下，我们应该有一个辅助函数，它允许指定超时并自动取消任务。

asyncio 通过名为 asyncio.wait_for 的函数提供此功能，该函数接收协程或任务对象，以及以秒为单位的超时时间。如果任务完成所需的时间超过了设定的超时时间，则会引发 TimeoutException，任务将自动取消。

为说明 wait_for 的工作原理，我们使用一个案例来说明：有一个任务需要 2 秒才能完成，但我们将它的超时时间设定为 1 秒。当得到一个 TimeoutError 异常时，我们将捕获异常，并检查任务是否被取消。

```
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():
    delay_task = asyncio.create_task(delay(2))
    try:
        result = await asyncio.wait_for(delay_task, 1)
        print("返回值:", result)
    except asyncio.TimeoutError:
```

```
        print("超时啦")
        # delay_task.cancelled() 用于判断任务是否被取消
        # 任务被取消：返回 True，没有被取消：返回 False
        print("任务是否被取消:", delay_task.cancelled())

asyncio.run(main())
"""
开始休眠 2 秒
超时啦
任务是否被取消: True
"""
```

应用程序运行 1 秒后，wait_for 语句将引发 TimeoutError，然后我们对其进行处理，并且 delay_task 被取消了。所以当一个任务超时的时候，会被自动取消。

所以通过 wait_for 语句就很方便，如果直接 await 一个任务，那么必须等到任务完成之后才能继续往下执行。如果任务一直完成不了，那么就会一直陷入阻塞。我们的目的是希望这个任务的执行时间是可控的，那么便可以使用 wait_for 并指定超时时间。注：使用 wait_for 必须要搭配 await，阻塞等待任务完成并拿到返回值、或者达到超时时间引发 TimeoutError 之后，程序才能往下执行。

因此“await 任务”和“await asyncio.wait_for(任务, timeout)”的效果是类似的，都是等待后面的任务完成并拿到它的返回值。但使用 wait_for 可以指定超时时间，在规定时间内如果没有完成，则抛出 TimeoutError，而不会一直陷入阻塞。

如果任务花费的时间比预期的长，在引发 TimeoutError 之后自动取消任务通常是个好主意。否则，可能有一个协程无限期地等待，占用永远不会释放的资源。但在某些情况下，我们可能希望保持协程运行。例如，我们可能想通知用户：某任务花费的时间比预期的要长，但即便超过了规定的超时时间，也不取消该任务。为此，可使用 asyncio.shield 函数包装任务，这个函数将防止传入的协程被取消，会给它一个屏蔽，将取消请求将忽略掉。

```
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():
    delay_task = asyncio.create_task(delay(2))
    try:
```

```
        # 通过 asyncio.shield 将 delay_task 保护起来
        result = await asyncio.wait_for(asyncio.shield(delay_task), 1)
        print("返回值:", result)
    except asyncio.TimeoutError:
        print("超时啦")
        # 如果超时依旧会引发 TimeoutError，但和之前不同的是
        # 此时任务不会被取消了，因为 asyncio.shield 会将取消请求忽略掉
        print("任务是否被取消:", delay_task.cancelled())
        # 从出现超时的地方，继续执行，并等待它完成
        result = await delay_task
        print("返回值:", result)

asyncio.run(main())
"""
开始休眠 2 秒
超时啦
任务是否被取消: False
休眠完成
返回值: 2
"""
```

取消和屏蔽会让人感到有些棘手，因为有几种值得注意的情况。下面来介绍一些基础但又很核心的知识，并随着讲解的案例越来越复杂，我们将更深入地探讨取消的工作原理。

相信你已经了解了协程和任务之间的关系，但任务和协程具体是如何关联的呢？我们来深度分析一下，并进一步了解 asyncio 的结构。

任务、协程、future 和 awaitable

关于 future

future 是一个 Python 对象，它包含一个你希望在未来某个时间点获得、但目前还不存在的值。通常，当创建 future 时，它没有任何值，因为它还不存在。在这种状态下，它被认为是不完整的、未解决的或根本没有完成的。然后一旦你得到一个结果，就可以设置 future 的值，这将完成 future。那时，我们可以认为它已经完成，并可从 future 中提取结果。

要了解 future 的基础知识，让我们尝试创建一个 future，设置它的值并提取该值。

```
import asyncio


# asyncio 里面有一个类 future，实例化之后即可得到 `future` 对象
# 然后 asyncio 里面还有一个类 Task，实例化之后即可得到 task 对象（也就是任务）
# 这个 Task 是 `future` 的子类，所以我们用的基本都是 task 对象，而不是 future 对象
# 但 future 这个类和 asyncio 的实现有着密不可分的关系，所以我们必须单独拿出来说一说
```

```
future = asyncio.Future()
print(future)  # <Future pending>
print(future.__class__)  # <class '_asyncio.Future'>
print(f"future 是否完成: {future.done()}")  # `future` 是否完成: False

# 设置一个值，通过 set_result
future.set_result("古明地觉")
print(f"future 是否完成: {future.done()}")  # future 是否完成: True
print(future)  # <Future finished result='古明地觉'>
print(f"future 的返回值: {future.result()}")  # `future` 的返回值: 古明地觉
```

可通过调用其类型对象 future 来创建 future，此时 future 上将没有结果集，因此调用其 done 方法将返回 False。此后用 set_result 方法设置 future 的值，这将把 future 标记为已完成。或者，如果想在 future 中设置一个异常，可调用 set_exception。

必须在调用 set_result（设置结果）之后才能调用 result（获取结果），并且 set_result 只能调用一次，但 result 可以调用多次。

然后我们来看一下 future 的源码，这里先只展示和当前介绍的内容相关的部分。

```
class Future:
    # future 实例有以下三个属性非常重要
    # _state: 运行状态，有三种，分别是 PENDING（正在运行）、CANCELLED（已取消）、FINISHED
（已完成）
    # _result: future 完成之后的设置的结果
    # _exception: future 报错时设置的异常

    def cancel(self):
        # cancel 方法，负责取消一个 future
        # 并且该方法有返回值，取消成功返回 True，取消失败返回 False
        self.__log_traceback = False
        # 检测状态是否为 PENDING，不是 PENDING，说明 future 已经运行完毕或取消了
        # 那么返回 False 表示取消失败，但对于 future 而言则无影响
        if self._state != _PENDING:
            return False
        # 如果状态是 PENDING，那么将其改为 CANCELLED
        self._state = _CANCELLED
        self.__schedule_callbacks()
        return True

    def cancelled(self):
        # 判断 future 是否被取消，那么检测它的状态是否为 CANCELLED 即可
        return self._state == _CANCELLED
```

```
    def done(self):
        # 判断 future 是否已经完成，那么检测它的状态是否不是 PENDING 即可
        # 注意：CANCELLED 和 FINISHED 都表示已完成
        return self._state != _PENDING

    def result(self):
        # 调用 result 方法相当于获取 future 设置的结果
        # 但如果它的状态为 CANCELLED，表示取消了，那么抛出 CancelledError
        # 正如同你 await 一个已取消的任务一样，因为 await 会阻塞任务并拿到它的执行结果
        # 如果任务已取消，同样抛出 CancelledError
        # 所以 future 和 task 是很相似的，因为 Task 本身就是 future 的子类
        # 至于 future 和 Task 具体的区别，我们一会儿再说
        if self._state == _CANCELLED:
            raise exceptions.CancelledError
        # 如果状态不是 FINISHED（说明还没有设置结果），那么抛出 asyncio.InvalidStateError
异常
        # 所以我们不能在 set_result 之前调用 result
        if self._state != _FINISHED:
            raise exceptions.InvalidStateError('Result is not ready.')
        self.__log_traceback = False
        # 走到这里说明状态为 FINISHED
        # 但不管是正常执行、还是出现异常，都会将状态标记为 FINISHED
        # 如果是出现异常，那么调用 result 会将异常抛出来
        if self._exception is not None:
            raise self._exception
        # 否则返回设置的结果
        return self._result

    def exception(self):
        # 无论是正常执行结束，还是出现异常，future 的状态都是已完成
        # 如果是正常执行结束，那么 self._result 就是结果，self._exception 为 None
        # 如果是出现异常，那么 self._result 为 None，self._exception 就是异常对象本身

        # 因此调用 result 和 exception 都要求 future 的状态为 FINISHED
        # 如果为 CANCELLED，那么同样抛出 CancelledError
        if self._state == _CANCELLED:
            raise exceptions.CancelledError
        # 如果为 PENDING，那么抛出 asyncio.InvalidStateError 异常
        if self._state != _FINISHED:
            raise exceptions.InvalidStateError('Exception is not set.')
        self.__log_traceback = False
        # 返回异常本身
        # 因此如果你不确定 future 到底是正常执行结束，还是抛了异常
```

```
            # 那么可以先调用 future.exception()，如果为 None，说明正常执行，再通过
future.result() 获取结果
        # 如果 future.exception() 不为 None，那么拿到的就是异常本身
        return self._exception

    def set_result(self, result):
        # 当 future 正常执行结束时，会通过 set_result 设置结果
        # 显然在设置结果的时候，future 的状态应该为 PENDING
        if self._state != _PENDING:
            raise exceptions.InvalidStateError(f'{self._state}: {self!r}')
        # 然后设置 self._result，当程序调用 future.result() 时会返回 self._result
        self._result = result
        # 并将状态标记为 FINISHED，表示一个任务从 PENDING 变成了 FINISHED
        # 所以我们不能对一个已完成的 future 再次调用 set_result
        # 因为第二次调用 set_result 的时候，状态已经不是 PENDING 了
        self._state = _FINISHED
        self.__schedule_callbacks()

    def set_exception(self, exception):
        # 和 set_result 类似，都表示任务从 PENDING 变成 FINISHED
        if self._state != _PENDING:
            raise exceptions.InvalidStateError(f'{self._state}: {self!r}')
        # 但 exception 必须是异常，且不能是 StopIteration 异常
        if isinstance(exception, type):
            exception = exception()
        if type(exception) is StopIteration:
            raise TypeError("StopIteration interacts badly with generators "
                            "and cannot be raised into a Future")
        # 将 self._exception 设置为 exception
        # 调用 future.exception() 的时候，会返回 self._exception
        self._exception = exception
        # 将状态标记为已完成
        self._state = _FINISHED
        self.__schedule_callbacks()
        self.__log_traceback = True
```

整个过程应该很好理解，我们通过一段代码再演示一下：

```
import asyncio

future = asyncio.Future()
# future 是否已完成
print(future.done())  # False
print(future._state != "PENDING")  # False
```

```
print(future._state)  # PENDING

# 获取结果
try:
    future.result()
except asyncio.InvalidStateError:
    print("future 尚未完成，不能获取结果")
    """
    future 尚未完成，不能获取结果
    """
# 但是我们可以通过 future._result 去获取（不推荐）
# 显然拿到的是 None
print(future._result)  # None
print(future._exception)  # None

future.set_result("我是返回值")
print(future.done())  # True
print(future._state)  # FINISHED
print(future.result() == future._result == "我是返回值")  # True
```

非常简单，但是我们在设置结果或设置异常的时候，应该通过 set_result() 和 set_exception()，不要通过类似 future._result = "…" 的方式。同理获取返回值或异常时，也要用 future.result() 和 future.exception()，不要直接用 future._result 或 future._exception，因为这背后还涉及状态的维护。

然后 future 也可以用在 await 表达式中，如果对一个 future 执行 await 操作，那么会处于阻塞，直到 future 有一个可供使用的值。这和 await 一个任务是类似的，当任务里面的协程 return 之后会解除阻塞，并拿到返回值。而 await future，那么当 future 有了值，await 同样会拿到它，并解除阻塞。

为理解这一点，让我们考虑一个返回 future 的 Web 请求的示例。发出一个返回 future 的请求应该立即完成，但由于请求需要一些时间，所以 future 还处于 PENDING 状态。然后一旦请求完成，结果将被设置，那么 future 会变成 FINISHED 状态，我们就可以访问它了，这个概念类似于 JavaScript 中的 Promise。而在 Java 中，这些被称为 completable future。

```
import asyncio

async def set_future_value(future):
    await asyncio.sleep(1)
    future.set_result("Hello World")

def make_request():
```

```
    future = asyncio.Future()
    # 创建一个任务来异步设置 future 的值
    asyncio.create_task(set_future_value(future))
    return future

async def main():
    # 注意这里的 `make_request`，它是一个普通的函数，如果在外部直接调用肯定是会报错的
    # 因为没有事件循环，在执行 set_future_value 时会报错
    # 但如果在协程里面调用是没问题的，因为协程运行时，事件循环已经启动了
    # 此时在 make_request 里面，会启动一个任务
    future = make_request()
    print(f"future 是否完成: {future.done()}")
    # 阻塞等待，直到 future 有值，什么时候有值呢?
    # 显然是当协程 set_future_value 里面执行完 future.set_result 的时候
    value = await future  # 暂停 main()，直到 future 的值被设置完成
    print(f"future 是否完成: {future.done()}")
    print(value)

asyncio.run(main())
"""
future 是否完成: False
future 是否完成: True
Hello World
"""
```

在代码中我们定义了一个函数 make request，该函数里面创建了一个 future 和一个任务，该任务将在 1 秒后异步设置 future 的结果。然后在主函数中调用 make_request，当调用它时，将立即得到一个没有结果的 future。然后 await future 会让主协程陷入等待，并将执行权交出去。一旦当 future 有值了，那么再恢复 main() 协程，拿到返回值进行处理。

但在 asyncio 中，你应该很少主动创建 future。更多时候，你将遇到一些返回 future 的异步 API，并可能需要使用基于回调的代码。举个例子：

```
import asyncio

def callback(future):
    print(f"future 已完成，值为 {future.result()}")

async def main():
    future = asyncio.Future()
    # 绑定一个回调，当 `future` 有值时会自动触发回调的执行
    future.add_done_callback(callback)
```

```
    future.set_result("666")

asyncio.run(main())
"""
future 已完成，值为 666
"""
```

asycio API 的实现很大程度上依赖于 future，因此最好对它们的工作原理有基本的了解，后续我们还会深入介绍。

future、任务和协程之间的关系

future 和任务之间有很密切的关系，事实上任务直接继承自 future，准确来说是 Task 继承自 Future。future 可以被认为代表了我们暂时不会拥有的值，而一个任务可以被认为是一个协程和一个 future 的组合。创建一个任务时，我们正在创建一个空的 future，并运行协程。然后当协程运行得到结果或出现异常时，我们将设置 future 的结果或异常。

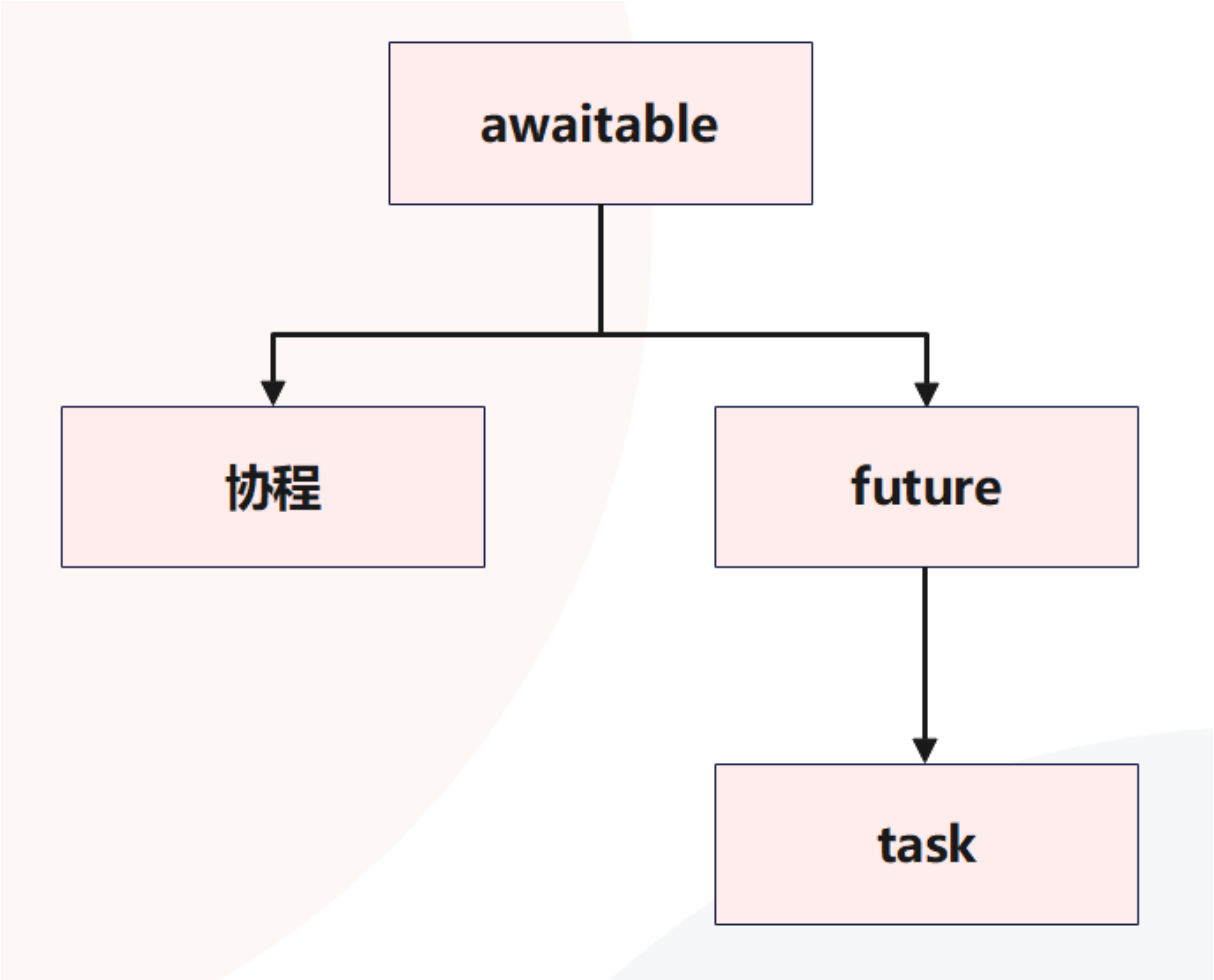
所以“await 任务”什么时候结束，显然是当协程执行完毕并将返回值设置在 future 里面的时候。如果直接 await future，那么需要我们手动调用 future.set_result；如果 await 任务，那么当协程执行完毕时会自动调用 future.set_result（执行出错则自动调用 future.set_exception），因为任务是基于协程包装得到的，它等价于一个协程加上一个 future。

但不管 await 后面跟的是任务还是 future，本质上都是等到 future 里面有值之后，通过 future.result() 拿到里面的值。

所以当 await 任务的时候，如果任务执行出错了，那么会怎么样呢？首先出错了，那么任务里面的 future 会调用 set_exception 设置异常。而前面在看 future 源码的时候，我们知道：如果没有出现异常，那么调用 result 返回结果，调用 exception 会返回 None；如果出现异常，那么调用 exception 会返回异常，调用 result 会将异常抛出来。而 await 任务，本质上就是在调用内部 future 的 result 方法，显然如果任务执行出错，那么会将出错时产生的异常抛出来。

再来看看协程，任务、future、协程，三者都可以跟在 await 关键字后面，那么它们有没有什么共同之处呢？

很简单，它们之间的共同点是 awaitable 抽象基类，这个类定义了一个抽象的魔法函数 __await__，任何实现了 __await__ 方法的对象都可以在 await 表达式中使用。协程直接继承自 awaitable，future 也是如此，而任务则是对 future 进行了扩展。



我们将可在 await 表达式中使用的对象称为 awaitable 对象，你会经常在 asyncio 文档中看到 awaitable 的术语，因为许多 API 方法并不关心你是否传入协程、任务或 future。

现在我们了解了任务、协程和 future 的基础知识，那如何评估它们的性能呢？到目前为止，我们只推测了它们运行所需的时间。为了使代码更严谨，让我们添加一些功能来测量执行时间。