楔子

许多应用程序，尤其在当今的 Web 应用程序领域，严重依赖 IO 操作。这些类型的操作包括从 Intermet 下载网页的内容、通过网络与一组微服务进行通信，或者针对 MySOL、Postgres 等数据库同时运行多个查询。Web 请求或与微服务的通信可能需要数百毫秒，如果网络很慢，甚至可能需要几秒钟。数据库查询可能耗费大量时间，尤其是在该数据库处于高负载或查询很复杂的情况下，而且 Web 服务器可能需要同时处理数百或数千个请求。

一次发出许多这样的 IO 请求会导致严重的性能问题，如果像在顺序运行的应用程序中那样一个接一个地执行这些请求，将看到复合的性能影响。例如，如果正在编写一个需要下载 100 个网页或执行 100 个查询的应用程序，每个查询需要 1 秒的执行时间，那么应用程序将至少需要 100 秒才能运行完成。但是，如果利用并发性，同时启动下载和等待，理论上可在短短 1 秒内完成这些操作。

asyncio 最初是在 Python3.4 中引入的，作为在多线程和多进程之外，处理这些高度并发工作负载的另一种方式。对于使用 IO 操作的应用程序来说，适当地利用这个库可以极大地提高性能和资源利用率，并可同时启动许多长时间运行的任务。

在本节中，我们将学习并发的基础知识，以便更好地理解如何使用 Python 和 asyncio 库实现并发。以及了解 CPU 密集型工作和 IO 密集型工作之间的差异，从而了解哪种并发模型更适合某些特定需求。此外还有进程和线程的基本知识，以及 Python 中由全局解释器锁（GIL）引起的独特的并发性挑战。最后，我们将了解如何利用带有事件循环的非阻塞 IO 概念来只使用一个 Python 进程或线程实现并发，并且这也是 asyncio 的主要并发模型。

什么是 asyncio?

在同步应用程序中，代码按顺序运行，下一行代码在前一行代码完成后立即运行，并且一次只发生一件事。该模型适用于许多应用程序，但如果一行代码运行特别慢怎么办？这种情况下，这个运行很慢的代码行之后的其它所有代码都将被卡住，必须等待该行代码执行完成。这些潜在的运行较慢的代码行可能会阻止应用程序运行任何其他代码。许多人在操作图形界面应用程序时遇到过这种情况，我们在界面上四处单击，直到应用程序冻结卡死，出现一个微调器或一个无响应的用户界面。这是一个应用程序被阻止从而导致糟糕用户体验的示例。

尽管任何操作都可以阻塞应用程序，如果它花费的时间足够长，许多应用程序会阻塞等 IO。IO 是指计算机的输入和输出设备，例如键盘、硬盘驱动器，以及最常见的网卡，这些操作等待用户输入内容或从基于 Web 的 API 检索内容。在同步应用程序中（相对异步应用程序而言），我们将等待这些操作完成，然后才能

运行其它操作。这可能导致性能和响应性问题，因为我们只能在同一时间运行一个长时间的操作，并且该操作将阻止应用程序执行其它任何操作。

此问题的一种解决方案是引入并发性。简单来说，并发意味着允许同时处理多个任务。在并发 IO 的情况下，允许同时发出多个 Web 请求或允许多个客户端同时连接到 Web 服务器。

有几种方法可以在 Python 中实现这种并发性，Python 生态系统的最新成员之一是 asyncio 库。asyncio 是异步 IO 的缩写，它是一个 Python 库，允许使用异步编程模型运行代码。这让我们可以一次处理多个 IO 操作，同时仍然允许应用程序保持对外界的响应。

那什么是异步编程呢？这意味着一个特定的长时间运行的任务可以在后台运行，与主应用程序分开。系统可以自由地执行不依赖于该任务的其他工作，而不是阻止其他所有应用程序代码等待该长时间运行的任务完成。一旦长时间运行的任务完成会收到它已经完成的通知，以便对结果进行处理。

在 Python3.4 中，asyncio 首先引入了装饰器和生成器通过 yield from 来定义协程。协程是一种方法，当有一个可能长时间运行的任务时，它可以暂停，然后在任务完成时恢复。在 Python3.5 中，当关键字 async 和 await 被显式添加到语言中时，该语言实现了对协程和异步编程的顶级支持。这种语法在 C# 和 JavaScript 等其他编程语言中很常见，使异步代码看起来像是同步运行的。这样的异步代码易于阅读和理解，因为它看起来像大多数软件工程师熟悉的顺序流。而 asyncio 是一个库，使用称为单线程事件循环的并发模型以异步方式执行这些协程。

虽然 asyncio 的名字可能会让我们认为这个库只适用于 IO 操作，但它也可通过与多线程和多进程相互操作来处理其他类型的操作。通过这种互操作性，可使用线程和进程的 async 与 await 语法，使这些工作流更容易理解。这意味着这个库不仅适用于基于 IO 的并发性，还可以用于 CPU 密集型代码。为更好地理解 asyncio 可以帮助我们处理何种类型的工作负载，以及哪种并发模型最适合哪种并发类型，下面探索 IO 密集型和 CPU 密集型操作之间的差异。

什么是 IO 密集型和 CPU 密集型

将一个操作称为 IO 密集型或 CPU 密集型时，指的是阻止该操作更快地运行的限制因素。这意味着，如果提高操作所绑定的对象的性能，该操作将在更短时间内完成。

在 CPU 密集型操作的情况下，如果 CPU 更强大，它将更快地完成任务，例如将其时钟速度从 2GHz 提高到 3GHz。在 IO 密集型操作的情况下，如果 IO 设备能在更短的时间内处理更多数据，程序将变得更快。这可通过 ISP 增加网络带宽或升级到更快的网卡来实现。

CPU 密集型操作通常是 Python 中的计算和处理代码，例如计算 pi 的数值，或者应用业务逻辑循环遍历字典的内容。在 IO 密集型操作中，大部分时间将花在等待网络或其他 IO 设备上，例如向 Web 服务器发出请求或从机器的硬盘驱动器读取文件。

```
import httpx
res = httpx.get("http://www.baidu.com")  # IO 密集型（web 请求）

items = res.headers.items()
headers = [f"{key}: {val}" for key, val in items]  # CPU 密集型（响应处理）
formatted_headers = "\n".join(headers)  # CPU 密集型（字符串连接）

with open("headers.txt", "w", encoding="utf-8") as f:
    f.write(formatted_headers)  # IO 密集型（磁盘写入）
```

IO 密集型和 CPU 密集型操作通常并存，我们首先发出一个 IO 密集型请求来下载 http://www.baidu.com 的内容。一旦得到响应，将执行一个 CPU 密集型循环来格式化响应头，并将它们转换为一个由换行符分隔的字符串。然后打开一个文件，并将字符串写入该文件，这两种操作都是 IO 密集型操作。

异步 IO 允许在执行 IO 操作时暂停特定程序的执行，可在后台等待初始 IO 完成时运行其他代码。这允许同时执行许多 IO 操作，从而潜在地加快应用程序的运行速度。

了解并发、并行和多任务处理

为了深刻地理解并发如何帮助应用程序更好地运行，首先应学习并充分理解并发编程的术语。本节，我们将学习更多关于并发的含义，以及 asyncio 如何使用一个称为多任务的概念来实现它。并发性和并行性是两个概念，可以帮助我们理解安排和执行各种任务、方法与驱动动作的例程。

并发

当我们说两个任务并发时，是指它们在一个时间段内同时执行。例如一个面包师要烘焙两种不同的蛋糕，要烘焙这些蛋糕，需要预热烤箱。根据烤箱和烘焙温度的不同，预热可能需要几十分钟，但不需要等烤箱完成预热后才开始其他工作，比如把面粉、糖和鸡蛋混合在一起。在烤箱预热过程中，我们可以做其他工作，直到烤箱发出提示音，告诉我们它已经预热完成了。

我们也不需要限制自己在完成第一个蛋糕之后才开始做第二个蛋糕。开始做蛋糕面糊，将其放进搅拌机，当第一团面糊搅拌完毕，就开始准备第二团面糊。在这个模型中，同时在不同的任务之间切换。这种任务之间的切换（烤箱加热时做其他事情，在两个不同的蛋糕之间切换）是并发行为。

并行

当我们说两个任务并行时，是指它们在同一个时间点同时执行。回到蛋糕烘焙示例，假设有第二个面包师的帮助。这种情况下，第一个面包师可以制作第 1 个蛋糕，而第二个面包师同时制作第 2 个蛋糕。两个人同时制作面糊属于并行运行，因为有两个不同的任务同时运行。

一句话总结并发和并行之间的区别：

- 并发：有多个任务在运行，但在特定的时间点，只有一个任务在运行；
- 并行：有多个任务在运行，但在特定的时间点，仍然有多个任务同时运行；

回到操作系统和应用程序，想象有两个应用程序在运行。在只有并发的系统中，可以在这些应用程序之间切换，先让一个应用程序运行一会儿，然后让另一个应用程序运行一会儿。如果切换的速度足够快，就会出现两件事同时发生的现象（或者说是假象）。但在一个并行的系统中，两个应用程序同时运行，系统同时全力运行两个任务，而不是在两个任务之间来回切换。

并发和并行的概念相似，并且经常容易混淆，一定要对它们之间的区别有清楚的认识。

并行与并发的区别

并发关注的是可以彼此独立发生的多个任务，可以在只有一个内核的 CPU 上实现并发，因为该操作将采用抢占式多任务的方式在任务之间切换。然而并行意味必须同时执行两个或多个任务，在只具有一个核心的机器上，这是不可能的。如果想成为可能，我们需要一个可同时运行多个任务的多核CPU。

虽然并行意味着并发，但并发并不总是意味着并行。在多核机器上运行的多线程应用程序既是并发的又是并行的。在此设置中，同时运行多个任务，并有多个内核独立执行与这些任务相关的代码。但是，通过多任务处理，可同时执行多个任务，而在给定时间只有一个任务在执行。

什么是多任务

多任务处理在当今世界无处不在，人们一边做早餐一边看电视，一边接电话一边等水烧开来泡茶。甚至在旅行途中，人们也可以多任务处理，例如在搭乘飞机时读自己喜欢的书。本节讨论两种主要的多任务处理：抢占式多任务处理和协同多任务处理。

抢占式多任务处理

在这个模型中，由操作系统决定如何通过一个称为时间片的过程，在当前正在执行的任务之间切换。当操作系统在任务之间切换时，我们称之为抢占。这种机制如何在后台工作取决于操作系统本身。它主要是通过使用多个线程或多个进程来实现的。

协同多任务处理

在这个模型中，不是依赖操作系统来决定何时在当前正在执行的任务之间切换，而是在应用程序中显式地编写代码，来让其他任务先运行。应用程序中的任务在它们协同的模型中运行，明确地说："先暂停我的任务一段时间，让其他任务先执行"。

协同多任务处理的优势

asyncio 使用协同多任务来实现并发性，当应用程序达到可以等待一段时间以返回结果的时间点时，在代码中显式地标记它，并让其它代码执行。一旦标记的任务完成，应用程序就"醒来"并继续执行该任务。这是一种并发形式，因为可同时启动多个任务，但重要的是，这不是并行模式，因为它们不会同时执行代码。

协同多任务处理优于抢占式多任务处理。首先，协同式多任务处理的资源密集度较低。当操作系统需要在线程或进程之间切换时，将涉及上下文切换。上下文切换是密集操作，因为操作系统只有保存有关正在运行的进程或线程的信息之后才能重新进行加载。

了解进程、线程、多线程和多处理

为更好地了解 Python 中并发的工作原理，首先需要了解线程和进程如何工作的基础知识，然后研究如何将它们用于多线程和多进程以同时执行任务。让我们从进程和线程的定义开始学习。

进程

进程是具有其它应用程序无法访问的内存空间的应用程序运行状态，创建 Python 进程的一个例子是运行一个简单的“hello world”应用程序，或在命令行输入 Python 来启动 REPL（交互式环境）。

多个进程可以在一台机器上运行，如果有一台拥有多核 CPU 的机器，就可以同时执行多个进程。在只有一个核的 CPU 上，仍可通过时间片，同时运行多个应用程序。当操作系统使用时间片时，它会在一段时间后自动切换下一个进程并运行它。确定何时发生此切换的算法因操作系统而异。

线程

线程可以被认为是轻量级进程，此外线程是操作系统可以管理的最小结构，它们不像进程那样有自己的内存空间，相反，它们共享进程的内存。线程与创建它们的进程关联，一个进程总是至少有一个与之关联的线程，通常称为主线程。一个进程还可以创建其他线程，通常称为工作线程或后台线程，这些线程可与主线程同时执行其他工作。线程很像进程，可以在多核 CPU 上并行运行，操作系统也可通过时间片在它们之间切换。当运行一个普通的 Python 应用程序时，会创建一个进程以及一个负责执行具体代码的主线程。

进程是操作系统分配资源的最小单元，线程是操作系统用来调度 CPU 的最小单元。进程好比一个房子，而线程是房子里面干活的人，所以一个进程里面可以有多个线程，线程共享进程里面的资源。因此真正用来工作的是线程，进程只负责提供相应的内存空间和资源。

```
import os
import threading

print(f"进程启动，pid 为 {os.getpid()}")
print(f"该进程内部运行 {threading.active_count()} 个线程")
print(f"当前正在运行的线程是 {threading.current_thread().name}")
"""
进程启动，pid 为 16900
该进程内部运行 1 个线程
当前正在运行的线程是 MainThread
"""
```

进程还可创建新的线程，这些线程可通过所谓的多线程技术同时完成其他工作，并共享进程的内存。

```
import threading

def hello_from_threading():
    print(f"Hello {threading.current_thread().name} 线程")

hello_thread = threading.Thread(target=hello_from_threading)
hello_thread.start()

print(f"该进程内部运行 {threading.active_count()} 个线程")
print(f"当前正在运行的线程是 {threading.current_thread().name}")
hello_thread.join()
"""
Hello Thread-1 线程
该进程内部运行 2 个线程
当前正在运行的线程是 MainThread
"""
```

注意：当你执行这段代码时，你可能会看到来自两个线程将消息输出在了同一行，这是一个静态条件，后续讨论。

多线程应用程序是在许多编程语言中，实现并发的常用方法。然而在 Python 中利用线程的并发性存在一些挑战，因为会受到全局解释器锁的限制，导致多线程仅对 IO 密集型工作有用。

但多线程并不是实现并发的唯一方法，还可创建多个进程来同时工作，这称为多进程。在多进程中，父进程创建一个或多个由它管理的子进程，然后可将任务分配给子进程。

Python 也提供了多进程模块来处理这个问题，它的 API 类似于 threading 模块。

```
import multiprocessing
import os

def hello_from_process():
    print(f"当前子进程的 pid 为 {os.getpid()}")

# 在 Windows 上必须加上 if __name__ == '__main__'
# 否则多进程乎启动失败
if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()
    hello_process.join()
"""
当前子进程的 pid 为 12532
"""
```

多线程和多进程似乎是启用 Python 并发的最佳选择，然而这些并发模型的强大功能受到 Python 的全局解释器锁的限制。

理解全局解释器锁

全局解释器锁的缩写是 GIL，是 Python 社区中一个有争议的话题。简而言之，GIL 阻止一个 Python 进程在任何给定时间同时执行多个字节码指令。这意味着即使在多核机器上有多个线程，Python 进程一次也只能有一个线程运行 Python 代码。在拥有多核 CPU 的环境中，这对于希望利用多线程来提高应用程序性能的 Python开发人员来说是一个重大挑战。

多处理可以同时运行多个字节码指令，因为每个 Python 进程都有自己的 GIL。

那为什么 GIL 会存在呢？答案在于 CPython 管理内存的方式。在 CPython 中，内存管理是通过引用计数实现的。每个对象都有一个引用计数（一个整数），用于跟踪有多少地方引用了该特定对象。当某处不再需要该引用的对象时，引用计数的值会减少，而当其他地方需要引用该对象时，它会增加。当引用计数为零时，说明没有人引用该对象，可将其从内存中删除。

而如果没有 GIL，那么引用计数机制就会出问题，具体细节可以参考我微信公众号上的这篇文章：为什么会有 GIL？如何释放 GIL 实现并行？ 。

asyncio 和 GIL

asyncio 利用 IO 操作释放 GIL 来提供并发性，即使只有一个线程也是如此。当使用 asyncio 时，创建了名为协程的对象，协程可被认为是在执行一个轻量级线程。就像我们可让多个线程同时运行，每个线程都有自己的并发 IO 操作一样，也可让多个协程一起运行。在等待 IO 密集型的协程完成时，仍可执行其他 Python 代码，从而提供并发性。

需要注意，asyncio 并没有绕过 GIL，而且仍然受制于 GIL。如果有一个CPU密集型任务，仍然需要使用多个进程来并发地执行它（这可以通过 asyncio 本身完成），否则将导致应用程序的性能问题。现在我们知道了仅使用一个线程就可以实现 IO 的并发性，下面深入了解如何使用非阻塞套接字实现并发性。

单线程并发

在上一节中，我们介绍了多线程作为一种实现 IO 操作并发性的机制。但是我们不需要多线程来实现这种并发性，可在一个进程和一个线程的范围内完成这一切。

我们利用了这样一个事实，即在系统级别，IO 操作可以并发完成。为更好地理解这一点，我们需要深入研究套接字是如何工作的，特别是非阻塞套接字是如何工作的。

什么是套接字

套接字（socket）是通过网络发送和接收数据的低级抽象，是在服务器之间传输数据的基础。套接字支持两种主要操作，发送字节和接收字节。我们将字节写入套接字，然后将其发送到一个远程地址，通常是某种类型的服务器。一旦发送了这些字节，就等待服务器将其响应写回套接字。一旦这些字节被发送回套接字，就可以读取结果。

套接字是一个低级概念，如果你把它们看作邮箱，就很容易理解了。你可以把信放在邮箱里，然后邮差拿起信，并递送到收件人的邮箱。收件人打开邮箱，打开信，然后根据内容的不同，收件人可能给你回信。在这个类比中，可将信里面的文字看作是要发送的数据或字节，将信件放入邮箱的操作看作是将字节写入套接字，将打开邮箱读取信件的操作看作是从套接字读取字节，将信件载体看作互联网上的传输机制（将数据路由到正确地址）。

在前面看到的从 baidu.com 获取内容的例子中，会打开一个连接到 baidu.com 服务器的套接字。然后编写一个请求来获取该套接字的内容，并等待服务器返回结果。

套接字在默认情况下是阻塞的，简单地说，这意味着等待服务器回复数据时，会暂停应用程序或阻塞它，直到获得数据并进行读取。因此，应用程序停止运行任何其他任务，直到从服务器获取数据、发生错误或出现超时。

在操作系统级别，不需要使用阻塞，套接字可在非阻塞模式下运行。在非阻塞模式下，当我们向套接字写入字节时，可以直接触发，而不必在意写入或读取操作，应用程序可以继续执行其他任务。之后，可让操作系统告知：我们收到了字节，并开始处理它。这使得应用程序可在等待字节返回的同时做任意数量的其他事情，不再阻塞和等待数据的返回，从而让程序响应更迅速，让操作系统通知何时有数据可供操作。

在后台，这是由几个不同的事件通知系统执行的，具体取决于我们运行的操作系统。asyncio 已经足够抽象，可在不同的通知系统之间切换，这取决于操作系统具体支持哪一个。以下是特定操作系统使用的事件通知系统：

- kqueue：FreeBSD 和 macOS
- epoll：Linux
- IOCP（IO完成端口）：Windows

这些系统会跟踪非阻塞套接字，并在准备好让我们做某事时通知我们。这个通知系统是 asyncio 实现并发的基础，在 asyncio 的并发模型中，只有一个线程在特定时间执行 Python。遇到一个 IO 操作时，将它交给操作系统的事件通知系统来跟踪它，一旦完成这个切换，Python 线程就可以继续自由地运行其他代码，或者为操作系统添加更多的非阻塞套接字来跟踪。IO 操作完成时，唤醒正在等待结果的任务，然后继续运行该 IO 操作之后出现的其他 Python代码。下图通过几个单独的操作来可视化这个流程，每个操作都依赖于一个套接字。

发出的非阻塞 IO 请求会立即返回，并告诉 OS 监视套接字中的数据，这允许 execute_other_code() 可以立刻执行，而不是等待 IO 请求完成。等到 IO 请求真正完成时，会收到通知，然后应用程序再去处理即可。

但是，如何跟踪哪些是正在等待 IO 的任务，哪些是作为常规 Python 代码运行的任务呢？答案在于一个称为事件循环的构造。

事件循环的工作原理

事件循环是每个 asyncio 应用程序的核心，事件循环是许多系统中相当常见的设计模式，并且已经存在了相当长的一段时间。如果你曾在浏览器中使用 JavaScript 发送异步 Web 请求，那么你已经在事件循环上创建了一个任务。Windows GUI 应用程序在幕后使用所谓的消息循环作为处理键盘输入等事件的主要机制，同时允许 UI 进行绘制。

最基本的事件循环非常简单，我们创建一个包含事件或消息列表的队列，然后启动循环，在消息进入队列时一次处理一条消息。在 Python 中，一个基本的事件循环可能看起来像下面这样：

```
from collections import deque
```

```
messages = deque()

while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

在 asyncio 中，事件循环保留任务队列而不是消息。任务是协程的包装器，协程可以在遇到 IO 密集型操作时暂停执行，并让事件循环运行其他不等待 IO 操作完成的任务。

创建一个事件循环时，会创建一个空的任务队列，然后将任务添加到要运行的队列中。事件循环的每次迭代都会检查需要运行的任务，并一次运行一个，直到任务遇到 IO 操作。然后任务将被暂停，指示操作系统监视相应的套接字以完成 IO，并寻找下一个要运行的任务。在事件循环的每次迭代中，会检查是否有 IO 操作已完成的任务，如果有则唤醒，并让它们继续运行。

主线程将任务提交给事件循环，此后事件循环可以运行它们。

为说明这一点，假设我们有三个任务，每个任务都发出一个异步 Web 请求。想象一下，这些任务有一些代码负责完成一些准备工作（是 CPU 密集型的），然后它们发出 Web 请求，然后又是一些 CPU 密集型的后处理代码。现在，同时将这些任务提交给事件循环。在伪代码中，可以这样写。

```
def make_request():
    cpu_bound_setup()
    io_bound_web_request()
    cpu_bound_postprocess()

task_one = make_request()
task_two = make_request()
task_three = make_request()
```

三个任务都以 CPU 密集型工作开始，并且使用单线程，因此只有第一个任务开始执行代码，其他两个任务则在等待。一旦任务1 中的 CPU 密集型设置工作完成，则会遇到一个 IO 密集型操作，并会暂停自己："我正在等待 IO，由于 IO 是操作系统负责的，并且不耗费 CPU，所以我要将执行权交出去，此时其他任何等待运行的任务都可以运行。"

当任务遇见 IO 阻塞的时候，会将执行权交给事件循环，事件循环再去找其他可以运行任务。

一旦发生这种情况，任务2 就可以开始执行了，任务2 同样会先执行其 CPU 密集型代码然后暂停，等待 IO。此时任务1 和任务2 在同时等待它们的网络请求完成，由于任务1 和任务2 都暂停等待 IO，于是开始运行任务3。任务3 同样也会暂停以等待其 IO 完成，此时三个任务都处于 IO 等待状态。

然后某一时刻任务1 的 Web 请求完成了，那么操作系统的事件通知系统会提醒我们此 IO 已完成（IO 操作不耗费 CPU，只要 IO 操作发起，剩下的交给操作系统。这时候线程可以去执行其它任务，当 IO 完成之后操作系统会通知我们），可以在任务2 和任务3 都在等待其IO完成时，继续执行任务1。

如果 CPU 密集代码的耗时忽略不计，IO 密集代码需要 2 秒钟，那么这三个任务执行完毕也只需要 2 秒钟，因为三个 web 请求是同时发出的。如果是同步代码，那么在返回响应之前，CPU 做不了其它事情，必须等到响应返回之后，才能发送下一个请求，那么整个过程就需要 6 秒钟的时间。

当任务一执行 cpu_bound_setup 的时候，任务二和任务三只能处于等待状态。当任务一进入 IO 的时候，任务二也开始执行 cpu_bound_setup，此时任务一和任务三需要处于等待状态。但对于任务一来说，由于它当前正处于 IO，CPU 给它也没法执行，不过也正因为它处于 IO（要 CPU 也没用），才将 CPU 交出去。

当任务二执行完 cpu_bound_setup 进入 IO 的时候，任务三开始执行 cpu_bound_setup，执行完之后进入 IO。

可以看到整个过程 CPU 没有处于空闲状态，在任务阻塞的时候立刻切换到其它任务上执行。然后当任务一、任务二、任务三都处于 IO 阻塞时，由于已经没有准备就绪的任务了，那么此时 CPU 就只能处于空闲了。接下来在某一时刻，任务一的 IO 结束了，操作系统会通知我们，然后执行任务一的 cpu_bound_postprocess。

所以每个任务等待 IO 的重叠是 asyncio 真正节省时间的地方。

小结

1. CPU 密集型的工作主要使用计算机处理器，而 IO 密集型的工作主要使用网络或其他输入/输出设备。asyncio 主要帮助我们使 IO 密集型工作并发执行，但也提供了使 CPU 密集型工作并发进行的 API。
2. 进程操作系统资源分配的基本单元，线程是操作系统调度操作 CPU 的基本单元。
3. 在使用非阻塞套接字时，可指示操作系统告诉我们何时有数据传入，而不是在等待数据传入时停止应用程序。这是允许 asyncio 仅使用单个线程实现并发的部分原因。
4. asyncio 应用程序的核心，事件循环会一直循环下去，寻找 CPU 密集型任务，同时暂停正在等待 IO 的任务。注意：这里的暂停，不是说任务就完全处于静止状态了，而是指不需要在该任务上浪费 CPU 了。因为 IO 操作一旦发起，就

由操作系统接管，程序可以去执行其它任务了，但 IO 操作的过程是不会中断的，因为它不耗费 CPU。而当 IO 操作完成后，会收到操作系统通知，然后可以再切换回该任务，继续执行该任务剩余的 CPU 密集代码。这样就保证了 CPU 的利用率，其实说白了核心就一句话：在任务陷入 IO 阻塞的时候，不要傻傻地等待，而是去执行其他的任务。

比如你同时和三个女孩聊天，当给某个女孩发出消息之后，不要一直处于等待状态，而是继续给别的女孩发消息。当女孩回你消息之后，你再切换回来，继续发。

因此操作系统提供的非阻塞 IO + IO 多路复用，是事件循环的核心。