# Python asyncio 并发编程

[英] 马修·福勒(Matthew Fowler) 著

殷海英 译

清華大學出版社

北 京

北京市版权局著作权合同登记号 图字：01-2022-4390

Matthew Fowler
Python Concurrency with asyncio
EISBN: 978-1-61729-866-0


图书在版编目(CIP)数据

Python asyncio 并发编程/(英)马修 • 福勒(Matthew Fowler)著；殷海英译. —北京：清华大学出版社，2023.1

书名原文：Python Concurrency with asyncio
ISBN 978-7-302-62283-3

I. ①P… II. ①马… ②殷… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆 CIP 数据核字(2022)第 253162 号

责任编辑：王 军
装帧设计：孔祥峰
责任校对：成凤进
责任印制：曹婉颖

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-83470000 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经 销：全国新华书店
开 本：170mm×240mm 印 张：25.5 字 数：511 千字
版 次：2023 年 1 月第 1 版 印 次：2023 年 1 月第 1 次印刷
定 价：128.00 元

产品编号：096110-01

致我美丽的妻子 Kathy，感谢你一直陪伴在我身边。

# 译 者 序

在十多年前，我刚开始在当地大学中讲授 Python 入门课程。那时，在 36 个学时中，为大家介绍了 Python 的语法及基础知识，并使用课程中介绍的技术实现了很多有趣的数据分析示例。当时，我和同学们都是在旧式笔记本型计算机上完成这些示例，由于使用的模拟数据较少，即便使用的计算机硬件配置较低，似乎一切都能正常运行。随着后续为计算机专业和金融专业开设的数据科学相关课程中，我们所操作的数据规模不断膨胀，也不再使用笔记本型计算机完成工作。在云时代，我们与 AWS 以及其他公有云服务商合作，从那里得到了很多用于教学的计算资源。工作中经常需要创建带有 16 个计算核心、64GB 内存及 1TB SSD 存储的计算环境，我们发现，即便使用了多个 CPU 计算核心，但系统中往往只有一个计算核心在工作，而其他计算核心都处于空闲状态，这极大地浪费了计算资源。我和学生探索其中的原因，这是由于 Python 天生的“缺陷”造成的，我们尝试了很多可以让应用程序并行运行的解决方案。在 2012 年秋天，Python 发布了 3.3 版本，这个版本中引入了 asyncio 软件库，可通过它实现单线程的并发编程；在 Python 3.4 中，asyncio 已成为标准库的一部分。通过 asyncio，我们可在服务器(比如 Web 服务器)上实现大量的 I/O 并发操作，从而提升应用程序的效率。

在本书中，通过 14 章的内容，由浅入深地介绍如何通过 asyncio 实现并发编程，并使用一个贯穿全书的示例，介绍如何使用 asyncio 在服务器与客户端之间进行并发通信。看着这个示例由简单变得复杂，在掌握相关知识的同时，也给自己带来了不小的成就感。作为 Python 和数据科学的教学人员，我建议你在阅读本书时，认真学习书中的示例，并在自己的计算机上运行本书附带的程序，这将让你更好地理解本书所介绍的内容。别担心，本书使用的示例对计算机的要求并不高；我在完成相关练习时，使用的是 2015 年在拉斯维加斯的 Best buy 购买的 MacBook Pro，配置 i5 CPU、8GB 内存及 128GB SSD，使用这样的机器运行本书的示例代码毫无压力。

最后，我要感谢清华大学出版社的王军老师，感谢他对我的信任与支持，感谢他帮我出版多本有关高性能计算、数据科学、云计算的书籍。同时要感谢我的学生闫禹树，感谢他帮助我完成书稿的校对及代码的测试。

殷海英
加利福尼亚州埃尔赛贡多市

# 关 于 作 者

Matthew Fowler 拥有近 20 年的软件工程经验，曾任软件架构师、工程总监等多个职位。他起初为科学应用程序编写软件，然后转向全栈 Web 开发和分布式系统，最终领导多个开发人员和管理人员团队为拥有数千万用户的电子商务网站编写应用程序及构建系统。他与妻子 Kathy 住在马萨诸塞州的列克星敦。

# 致　谢

首先，我想感谢妻子 Kathy，当我不确定某件事是否有意义时，她总是在我身边提供帮助，在整个写作过程中她都非常支持我。也要感谢我的狗 Dug，它总是在我附近丢球，提醒我停止写作，去休息一下。

接下来，我要感谢编辑 Doug Rudder，以及技术审稿人 Robert Wenner。他们的反馈是非常宝贵的，帮助我按时完成这本书，并提高了本书的质量，确保我的代码和解释的正确性，并让一切通俗易懂。

致所有评论者：Alexey Vyskubov、Andy Miles、Charles M. Shelton、Chris Viner、Christopher Kottmyer、Clifford Thurber、Dan Sheikh、David Cabrero、Didier Garcia、Dimitrios Kouzis-Loukas、Eli Mayost、Gary Bake、Gonzalo Gabriel Jiménez Fuentes、Gregory A. Lussier、James Liu、Jeremy Chen、Kent R. Spillner、Lakshmi Narayanan Narasimhan、Leonardo Taccari、Matthias Busch、Pavel Filatov、Phillip Sorensen、Richard Vaughan、Sanjeev Kilarapu、Simeon Leyzerzon、Simon Tschöke、Simone Sguazza、Sumit K. Singh、Viron Dadala、William Jamir Silva 和 Zoheb Ainapore，他们的建议帮助这本书变得更好。

最后，我要感谢过去几年我遇到的多位老师、同事和导师。我从他们那里学到了很多，也成长了很多。我们在一起的经历，为我创作这本书，以及事业的成功提供了诸多帮助。没有你们，就不会有我今天的成就。感谢你们！

# 序

大概在 20 年前，我开始从事软件工程工作，编写了一个由 MATLAB、C++和 VB.NET 代码组成的系统，用于控制和分析来自质谱仪与其他实验室设备的数据。看到仅使用一行代码即可让程序按照预想的方式运行，那种兴奋感一直萦绕在我的脑海中，从那时起，我就知道软件工程将是我的未来职业。多年来，我逐渐转向 API 开发和分布式系统，主要专注于 Java 和 Scala，并在此过程中学习了大量的 Python 内容。

我在 2015 年左右开始学习 Python，主要使用一个机器学习 pipeline，该 pipeline 收集传感器数据，并使用这些数据来预测传感器佩戴者的睡眠、运动等活动。当时，这个机器学习过程非常缓慢，以至于给客户造成困扰。我解决这个问题的方法之一就是利用并发性。当我深入研究 Python 并发编程的相关知识时，我发现这与我在 Java 世界中所熟悉的事物相比，有些事物很难探索和学习。为什么多线程不能像 Java 那样工作呢？使用多进程是否更有意义？新引入的 asyncio 会带来怎样的效果呢？什么是全局解释器锁，它为什么存在？关于 Python 中并发这一主题的书籍并不多，大多数知识分散在各种文档中，只有少数几个博客介绍相关内容。到今天，事情并没有太大的改变。虽然我们有更多的资源，但总体情况仍然是稀缺的、不连贯的，并且对于新手来说，并发技术表现得不是十分友好。

在 asyncio 仍处于起步阶段时，就已成为 Python 的一个重要模块。现在，除了多线程和多进程之外，单线程并发模型和协程是 Python 中并发性的核心组件。这意味着 Python 中的并发环境变得更大、更复杂，而对于那些想要学习它的人来说，仍然没有全面的学习资源可用。

我写这本书的动机是填补 Python 领域中并发主题(特别是异步和单线程并发)的空白。我想让所有开发人员都能更容易地接触到复杂且文档不足的单线程并发主题。我还想写一本书来增进开发人员对 Python 之外并发主题的一般理解。像 Node.js 这样的框架和像 Kotlin 这样的语言都有单线程并发模型与协程，所以从本书获得的知识也有利于在这些领域的学习。我希望本书能给开发人员的日常工作提供帮助——不仅在 Python 领域，而是在所有并发编程领域。

# 前　言

《Python asyncio 并发编程》旨在介绍如何在 Python 中利用并行技术提高应用程序的性能、吞吐量和响应能力。本书首先关注并行的核心主题，解释 asyncio 的单线程并发模型是如何工作的，以及协程和 async/await 语法的工作原理。然后介绍并发的实际应用，例如并行发出多个 Web 请求或数据库查询、管理线程和进程、构建 Web 应用程序，以及处理同步问题。

## 目标读者

本书适用于希望在现有或新的 Python 应用程序中，更好地理解和使用并发技术的中高级开发人员。本书的立足于在于通过通俗易懂的语言解释复杂的并发技术。你不需要拥有并发经验，当然，如果你拥有这方面的经验，可以更快地理解书中的内容。在本书中，我们将介绍很多知识(从基于 Web 的 API 到命令行应用程序)，帮助开发人员解决工作中遇到的许多问题。

## 内容路线图

本书一共 14 章，前几章介绍基础知识，后几章将介绍更多高级主题。

- **第 1 章**：专注于 Python 中的基本并发知识。将介绍什么是 CPU 密集型和 I/O 密集型工作负载，并介绍 asyncio 的单线程并发模型的工作原理。
- **第 2 章**：重点介绍 asyncio 协程的基础知识，以及如何使用 async/await 语法来构建使用并发的应用程序。
- **第 3 章**：重点介绍非阻塞套接字和选择器如何工作，以及如何使用 asyncio 构建 echo 服务器。
- **第 4 章**：重点介绍如何同时发出多个 Web 请求。通过本章的学习，我们将进一步了解关于并发运行协同程序的核心 asyncio API。

- **第 5 章**：重点介绍如何使用连接池同时进行多个数据库查询。还将介绍数据库上下文中的异步上下文管理器和异步生成器。
- **第 6 章**：专注于 multiprocessing 库，特别是如何将它与 asyncio 结合，来处理 CPU 密集型工作。将构建一个 map/reduce 应用程序来介绍它的工作方法。
- **第 7 章**：专注于多线程，特别是如何将它与 asyncio 结合来处理阻塞 I/O。这对于没有原生 asyncio 支持但仍然可以从并发中受益的库很有帮助。
- **第 8 章**：着重介绍网络流和协议。将使用网络流和协议来创建一个能同时处理多个用户聊天的服务器和客户端。
- **第 9 章**：主要介绍异步驱动的 Web 应用程序和 ASGI(异步服务器网关接口)。将探索一些 ASGI 框架，并讨论如何使用它们构建 Web API。还将探索 WebSocket。
- **第 10 章**：描述如何使用基于 asyncio 的 Web API 来构建微服务架构。
- **第 11 章**：重点讨论单线程并发同步问题，以及如何解决这些问题。将深入研究锁、信号量、事件和条件。
- **第 12 章**：重点关注异步队列。将使用异步队列来构建一个 Web 应用程序，该应用程序可以即时响应客户端请求，尽管需要在后台执行耗时的工作。
- **第 13 章**：专注于创建和管理子进程，展示如何读取和写入数据。
- **第 14 章**：专注于高级主题，例如强制事件循环迭代、上下文变量和创建自己的事件循环。这些信息对于 asyncio API 设计者和那些对 asyncio 事件循环的内部运作流程感兴趣的人十分有帮助。

对于读者来说，你应该仔细阅读前四章的内容，从而全面了解 asyncio 的工作原理、如何构建第一个真正的应用程序，以及如何使用核心 asyncio API 来并行运行协同程序(将在第 4 章中介绍)。此后，你可以根据自己的兴趣来阅读本书。

# 关于代码

本书包含许多代码示例，包括编号的代码清单。一些代码清单在同一章后面的清单中被重用，有些则在多个章节中被重用。对于跨多个章节重用的代码，将假定你已经创建了一个名为 util 的模块(你将在第 2 章中创建它)。对于每个单独的代码清单，假设你为该章创建了一个名为 chapter_{*chapter_number*}的模块，然后将代码放入格式为 listing_{*chapter_number*}_{*listing_number*}的 Python 文件中。例如，第 2 章中清单 2-2 的代码将位于一个名为 chapter_2 的模块中，该模块保存在名为 listing_2_2.py 的文件中。

书中多处提到性能数字，例如程序完成的时间或每秒完成的 Web 请求。本书中的代码示例运行在配备 2.4GHz 8 核 Intel Core i9 处理器和 32GB 2667 MHz DDR4 RAM 的 2019 MacBook Pro 上，并进行了基准测试，使用千兆无线互联网连接。根据你运行的机器，这些数字会有所不同，加速或改进的因素也会有所不同。

本书中使用的代码可通过扫描封底二维码下载。

# 关于封面插图

封面上的人物是 Paysanne du Marquisat de Bade，即巴登侯爵府的农妇，取自 Jacques Grasset de Saint-Sauveur 于 1797 年出版的一本书。每张插画都是手工绘制和上色的。

在几个世纪前，很容易通过人们的穿着来确定他们住在哪里，确定他们的职业或社会地位。曼宁出版社用表现几个世纪前地区文化丰富多样性的插图作为书籍封面，反映计算机行业的创造性，并让这些珍贵的插图重新焕发生机。

# 目　录


第 1 章　asyncio 简介……1
1.1　什么是 asyncio……2
1.2　什么是 I/O 密集型和 CPU 密集型……3
1.3　了解并发、并行和多任务处理……4
1.3.1　并发……4
1.3.2　并行……5
1.3.3　并行与并发的区别……6
1.3.4　什么是多任务……6
1.3.5　协同多任务处理的优势……7
1.4　了解进程、线程、多线程和多处理……7
1.4.1　进程……7
1.4.2　线程……8
1.5　理解全局解释器锁……11
1.5.1　GIL 会释放吗……15
1.5.2　asyncio 和 GIL……17
1.6　单线程并发……17
1.7　事件循环的工作原理……20
1.8　本章小结……22
第 2 章　asyncio 基础……23
2.1　关于协程……23
2.1.1　使用 async 关键字创建协程……24
2.1.2　使用 await 关键字暂停执行……26
2.2　使用 sleep 引入长时间运行的协程……27
2.3　通过任务实现并行……30
2.3.1　创建任务……30
2.3.2　同时运行多个任务……31
2.4　取消任务和设置超时……34
2.4.1　取消任务……34
2.4.2　设置超时并使用 wait_for 执行取消……36
2.5　任务、协程、future 和 awaitable……38
2.5.1　关于 future……38
2.5.2　future、任务和协程之间的关系……40
2.6　使用装饰器测量协程执行时间……41
2.7　协程和任务的陷阱……43
2.7.1　运行 CPU 密集型代码……44
2.7.2　运行阻塞 API……46
2.8　手动创建和访问事件循环……47
2.8.1　手动创建事件循环……47
2.8.2　访问事件循环……48
2.9　使用调试模式……49
2.9.1　使用 asyncio.run……49
2.9.2　使用命令行参数……49
2.9.3　使用环境变量……50
2.10　本章小结……51

第 3 章 第一个 asyncio 应用程序……53
3.1 使用阻塞套接字……54
3.2 使用 telnet 连接到服务器……56
3.2.1 从套接字读取和写入数据……57
3.2.2 允许多个连接和阻塞的危险性……59
3.3 使用非阻塞套接字……61
3.4 使用选择器模块构建套接字事件循环……65
3.5 使用 asyncio 事件循环的回显服务器……68
3.5.1 套接字的事件循环协程……69
3.5.2 设计一个异步回显服务器……69
3.5.3 解决任务中的错误……72
3.6 正常关闭……74
3.6.1 监听信号……74
3.6.2 等待挂起的任务完成……76
3.7 本章小结……79
第 4 章 并发网络请求……81
4.1 aiohttp……82
4.2 异步上下文管理器……82
4.2.1 使用 aiohttp 发出 Web 请求……85
4.2.2 使用 aiohttp 设置超时……87
4.3 并发运行任务及重新访问……88
4.4 通过 gather 执行并发请求……91
4.5 在请求完成时立即处理……95
4.6 使用 wait 进行细粒度控制……99
4.6.1 等待所有任务完成……99
4.6.2 观察异常……102
4.6.3 当任务完成时处理结果……104
4.6.4 处理超时……107
4.6.5 为什么要将所有内容都包装在一个任务中……109
4.7 本章小结……110
第 5 章 非阻塞数据库驱动程序……111
5.1 关于 asyncpg……111
5.2 连接 Postgres 数据库……112
5.3 定义数据库模式……113
5.4 使用 asyncpg 执行查询……116
5.5 通过连接池实现并发查询……119
5.5.1 将随机 sku 插入 products 数据库……119
5.5.2 创建连接池从而同时运行查询……123
5.6 使用 asyncpg 管理事务……128
5.6.1 嵌套事务……130
5.6.2 手动管理事务……132
5.7 异步生成器和流式结果集……133
5.7.1 异步生成器介绍……134
5.7.2 使用带有流游标的异步生成器……135
5.8 本章小结……139
第 6 章 处理 CPU 密集型工作……141
6.1 介绍 multiprocessing 库……142
6.2 使用进程池……144
6.3 进程池执行器与 asyncio……146
6.3.1 进程池执行器……146
6.3.2 带有异步事件循环的进程池执行器……148
6.4 使用 asyncio 解决 MapReduce 的问题……149
6.4.1 简单的 MapReduce 示例……151
6.4.2 Google Books Ngram 数据集……153

6.4.3 使用 asyncio 进行映射和归约 …… 154
6.5 共享数据和锁 …… 159
6.5.1 共享数据和竞争条件 …… 160
6.5.2 使用锁进行同步 …… 163
6.5.3 与进程池共享数据 …… 166
6.6 多进程，多事件循环 …… 170
6.7 本章小结 …… 173
第 7 章 通过线程处理阻塞任务 …… 175
7.1 threading 模块 …… 176
7.2 通过 asyncio 使用线程 …… 180
7.2.1 request 库 …… 180
7.2.2 线程池执行器 …… 181
7.2.3 使用 asyncio 的线程池执行器 …… 183
7.2.4 默认执行器 …… 184
7.3 锁、共享数据和死锁 …… 186
7.3.1 可重入锁 …… 188
7.3.2 死锁 …… 190
7.4 单线程中的事件循环 …… 192
7.4.1 Tkinter …… 193
7.4.2 使用 asyncio 和线程构建响应式 UI …… 195
7.5 使用线程执行 CPU 密集型工作 …… 203
7.5.1 多线程与 hashlib …… 203
7.5.2 多线程与 NumPy …… 206
7.6 本章小结 …… 208
第 8 章 流 …… 211
8.1 流 …… 212
8.2 传输和协议 …… 212
8.3 流读取与流写入 …… 216
8.4 非阻塞命令行输入 …… 219
8.5 创建服务器 …… 230
8.6 创建聊天服务器和客户端 …… 232
8.7 本章小结 …… 239
第 9 章 Web 应用程序 …… 241
9.1 使用 aiohttp 创建 REST API … 242
9.1.1 什么是 REST …… 242
9.1.2 aiohttp 服务器基础知识 …… 243
9.1.3 连接到数据库并返回结果 …… 244
9.1.4 比较 aiohttp 和 Flask …… 251
9.2 异步服务器网关接口 …… 253
9.3 ASGI 与 Starlette …… 255
9.3.1 使用 Starlette 的 REST 端点 …… 255
9.3.2 WebSocket 与 Starlette …… 257
9.4 Django 异步视图 …… 261
9.4.1 在异步视图中运行阻塞工作 …… 267
9.4.2 在同步视图中使用异步代码 …… 268
9.5 本章小结 …… 269
第 10 章 微服务 …… 271
10.1 什么是微服务 …… 272
10.1.1 代码的复杂性 …… 272
10.1.2 可扩展性 …… 272
10.1.3 团队和堆栈独立性 …… 273
10.1.4 asyncio 如何提供帮助 … 273
10.2 backend-for-frontend 模式 … 273
10.3 实施产品列表 API …… 275
10.3.1 “用户收藏”服务 …… 275
10.3.2 实现基础服务 …… 276
10.3.3 实现 backend-for-frontend 服务 …… 281

10.3.4 重试失败的请求……287
10.3.5 断路器模式……290
10.4 本章小结……296
第 11 章 同步……297
11.1 了解单线程并发错误……298
11.2 锁……302
11.3 使用信号量限制并发性……306
11.4 使用事件来通知任务……312
11.5 条件……317
11.6 本章小结……322
第 12 章 异步队列……323
12.1 异步队列基本知识……324
12.1.1 Web 应用程序中的队列……331
12.1.2 网络爬虫队列……334
12.2 优先级队列……337
12.3 LIFO 队列……345
12.4 本章小结……347
第 13 章 管理子进程……349
13.1 创建子进程……349
13.1.1 控制标准输出……352
13.1.2 同时运行子进程……355
13.2 与子进程进行通信……359
13.3 本章小结……363
第 14 章 高级 asyncio……365
14.1 带有协程和函数的 API……366
14.2 上下文变量……368
14.3 强制事件循环迭代……370
14.4 使用不同的事件循环实现……371
14.5 创建自定义事件循环……373
14.5.1 协程和生成器……373
14.5.2 不建议使用基于生成器的协程……374
14.5.3 自定义可等待对象……376
14.5.4 使用带有 future 的套接字……379
14.5.5 任务的实现……381
14.5.6 实现事件循环……383
14.5.7 使用自定义事件循环实现服务器……386
14.6 本章小结……388

# 第1章

# asyncio 简介

本章内容：

- asyncio 及其优势
- 并发、并行、线程和进程
- 全局解释器锁及其对并发性的影响
- 非阻塞套接字如何仅用一个线程实现并发
- 基于事件循环并发的工作原理

许多应用程序，尤其在当今的 Web 应用程序领域，严重依赖 I/O 操作。这些类型的操作包括从 Internet 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 MySQL 或 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。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 I/O 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 Python 3.4 中引入的，作为在多线程和多处理之外，处理这些高度并发工作负载的另一种方式。对于使用 I/O 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。

在本章中，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。我们将了解 CPU 密集型工作和 I/O 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。还将了解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 I/O 概念来仅使用一个 Python 进程或线程实现并发。这是 asyncio 的主要并发模型。

## 1.1 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。但是，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他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待 I/O。I/O 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，然后才能运行其他操作。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他任何操作。

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 I/O 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 I/O 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 I/O 操作，同时仍然允许应用程序保持对外界的响应。

那么，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 Python 3.4 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 Python 3.5 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。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 I/O 操作，但它也可通过与多线程和多处理进行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 I/O 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 I/O 密集型和 CPU 密集型操作之间的差异。

## 1.2　什么是 I/O 密集型和 CPU 密集型

将一个操作称为 I/O 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 I/O 密集型操作的情况下，如果 I/O 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 I/O 密集型操作中，大部分时间将花在等待网络或其他 I/O 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。

**代码清单 1-1　I/O 密集型和 CPU 密集型操作**

```
import requests

response = requests.get('https:/ / www .example .com')

items = response.headers.items()

headers = [f'{key}: {header}' for key, header in items]
```

I/O 密集型 Web 请求

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') as file:
    file.write(formatted_headers)  ← I/O 密集型
                                     磁盘写入
```

I/O 密集型和 CPU 密集型操作通常并存。我们首先发出一个 I/O 密集型请求来下载 https://www.example.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的标头，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 I/O 密集型操作。

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

## 1.3 了解并发、并行和多任务处理

为了更好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。

### 1.3.1 并发

当我们说两个任务并发时，是指这些任务同时发生。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。

### 1.3.2　并行

虽然并发意味着多个任务同时进行，但并不意味着它们并行运行。当我们说某事并行运行时，意思是不仅有两个或多个任务同时发生，而且它们在同时执行。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一个蛋糕，而第二个面包师同时制作第二个蛋糕。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(如图 1-1 所示)。

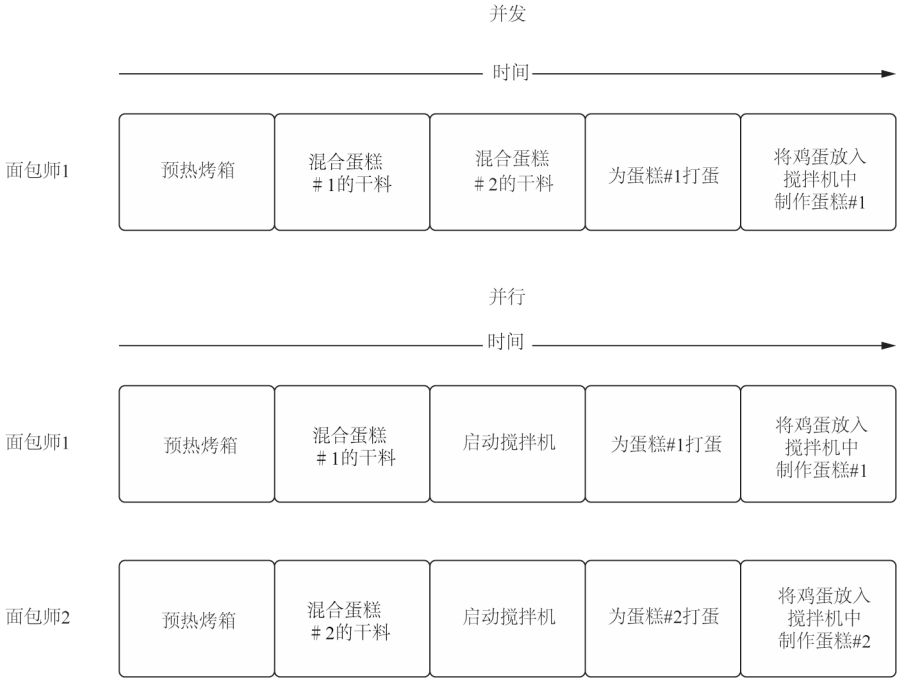


图 1-1　使用并发，有多个任务同时发生，但在特定时间点，只有一个任务在运行。使用并行，有多个任务同时发生，并且在特定时间点有多个任务同时运行

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)。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(如图 1-2 所示)，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。

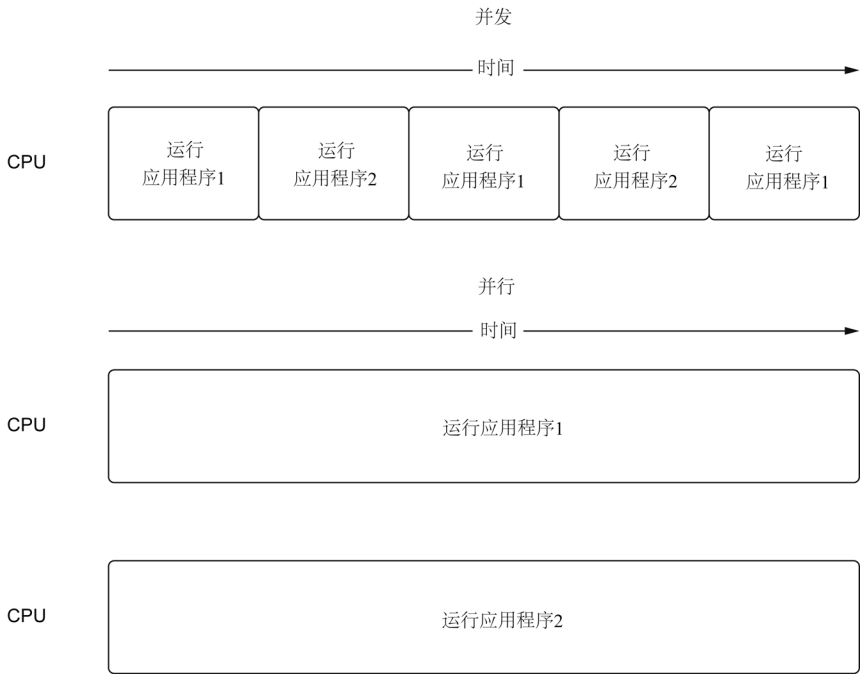


图 1-2 并发系统中，可在两个应用程序之间切换。并行系统中，可同时全力运行两个应用程序

### 1.3.3 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(在下一节中定义)在任务之间切换。然而，并行性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。为使这成为可能，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。

### 1.3.4 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，可让其他任务先去运行。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。”

### 1.3.5　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。这允许其他代码在等待结果返回后台时运行。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。当操作系统需要在运行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才能重新进行加载。

## 1.4　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处理以同时执行任务。让我们从进程和线程的定义开始学习。

### 1.4.1　进程

进程是具有其他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(读取 eval 输出循环)。

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。在只有一个内核的 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。

### 1.4.2 线程

线程可以被认为是轻量级进程。此外，它们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，它们共享创建进程的内存。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运行 Python 应用程序的主线程。

**代码清单 1-2 简单 Python 应用程序中的进程和线程**

```
import os
import threading

print(f'Python process running with process id: {os.getpid()}')
total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')
```

图 1-3 中勾勒出代码清单 1-2 的进程。我们创建一个简单的应用程序来展示主线程的基础知识。首先获取进程 ID(进程的唯一标识符)并输出它，以证明确实有一个专用进程正在运行。然后，获取正在运行的线程的活动计数以及当前线程的名称，以表明正在运行一个线程——主线程。虽然每次运行此代码时进程 ID 都是不同的，但运行代码清单 1-2 将给出如下输出：

```
Python process running with process id: 98230
Python currently running 1 thread(s)
The current thread is MainThread
```

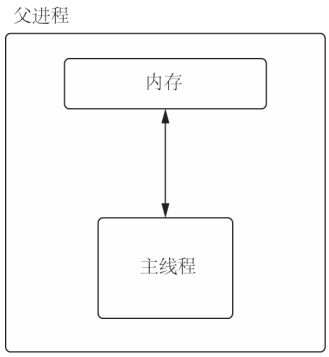


图 1-3　带有一个主线程(从内存中读取数据)的进程

进程还可创建共享主进程内存的其他线程。这些线程可通过所谓的多线程技术同时完成其他工作。

**代码清单 1-3　创建多线程 Python 应用程序**

```
import threading

def hello_from_thread():
    print(f'Hello from thread {threading.current_thread()}!')

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

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')

hello_thread.join()
```

图 1-4 中描绘出代码清单 1-3 的进程和线程。创建一个方法来输出当前线程的名称，然后创建一个线程来运行该方法。再调用线程的 start 方法开始运行它。最后调用 join 方法。join 会让程序暂停，直到启动的线程完成。如果运行前面的代码，将看到如下输出：

```
Hello from thread <Thread(Thread-1, started 123145541312512)>!
Python is currently running 2 thread(s)
```

```
The current thread is MainThread
```

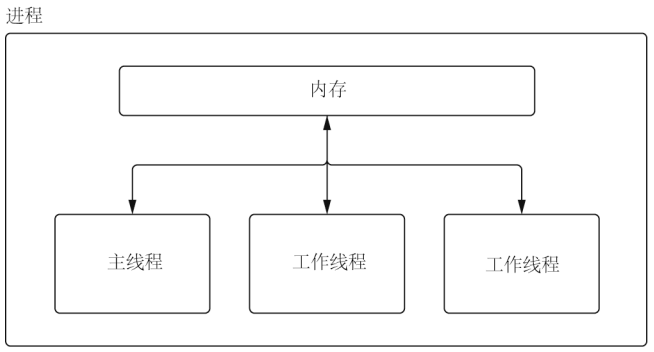


图 1-4 具有两个工作线程和一个主线程的多线程程序，所有线程共享进程的内存

请注意，在运行此命令时，你可能会看到来自线程的 hello，而 Python 当前正在运行 2 个线程消息在同一行输出。这是一个竞态条件，我们将在下一节以及第 6 章和第 7 章中对此进行深入探讨。

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。多线程仅对 I/O 密集型工作有用，因为会受到全局解释器锁的限制，这将在 1.5 节中讨论。

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。这称为多处理。在多处理中，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 提供了多处理模块来处理这个问题。它的 API 类似于 threading 模块的 API。首先创建一个带有 target 函数的进程。然后调用 start 方法来执行它，最后调用 join 方法等待它完成运行。

**代码清单 1-4 创建多个进程**

```
import multiprocessing
import os

def hello_from_process():
    print(f'Hello from child process {os.getpid()}!')

if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()
    print(f'Hello from parent process {os.getpid()}')
```

```
hello_process.join()
```

图 1-5 中描绘出代码清单 1-4 的进程和线程。我们创建一个输出其进程 ID 的子进程，还输出父进程 ID 以证明正在运行不同的进程。对于 CPU 密集型工作，多处理通常是最佳选择。

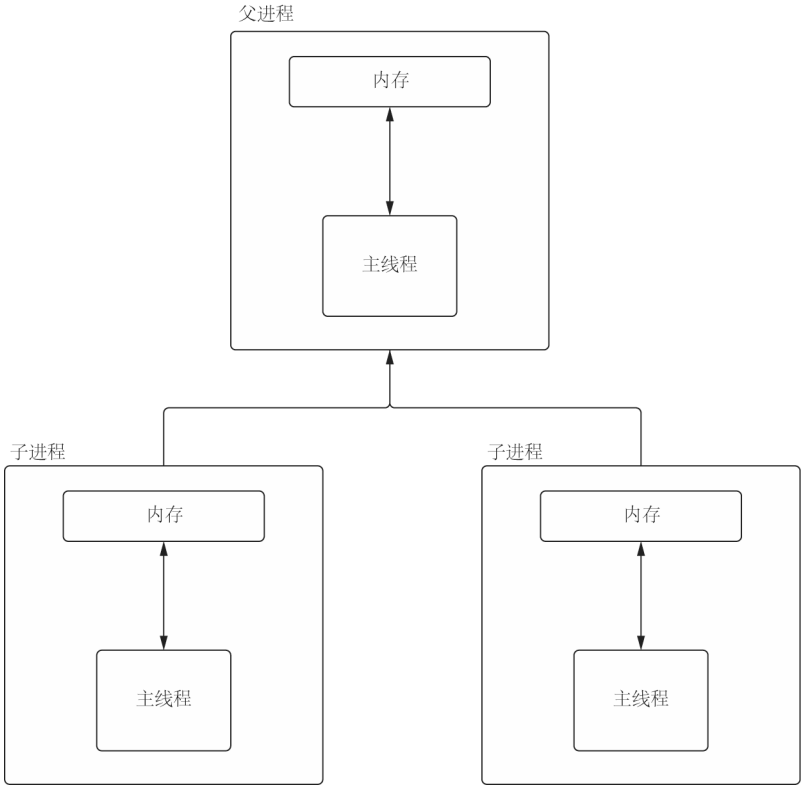


图 1-5　带有一个父进程和两个子进程的多处理应用程序

多线程和多处理似乎是启用 Python 并发的最佳选择。然而，这些并发模型的强大功能受到 Python 的一个实现细节——全局解释器锁的限制。

## 1.5　理解全局解释器锁

全局解释器锁的缩写是 GIL，读作 gill，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执行多个 Python 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。

**注意**

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。

那么，为什么 GIL 会存在呢？答案在于 CPython 管理内存的方式。在 CPython 中，内存主要由称为“引用计数”的进程管理。引用计数通过跟踪当前谁需要访问特定的 Python 对象(如整数、字典或列表)来工作。引用计数的值是一个整数，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，而当其他人需要该对象时，它会增加。当引用计数为零时，没有人引用该对象，可将其从内存中删除。

**什么是 CPython？**

CPython 是 Python 的参考实现。通过参考实现(语言的标准实现)，Python 被用作语言正确行为的参考。Python 还有其他实现，例如用于在 Java 虚拟机上运行的 Jython 和为.NET 框架设计的 IronPython。

与线程的冲突源于 CPython 中的实现不是线程安全的。当我们说 CPython 不是线程安全时，意指如果两个或多个线程修改一个共享变量，该变量可能以意外状态结束。这种意外状态取决于线程访问变量的顺序，通常称为竞态条件。当两个线程需要同时引用一个 Python 对象时，可能出现竞态条件。

如图 1-6 所示，如果两个线程同时增加引用计数，可能遇到一个线程导致引用计数为零的情况，而对象仍在被另一个线程使用。当尝试读取可能被删除的内存数据时，可能导致应用程序崩溃。

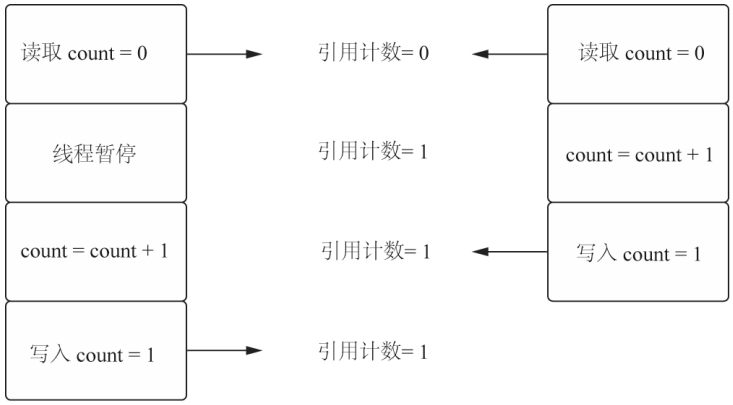


图 1-6　竞态条件示例，两个线程试图同时增加引用计数，得到的不是期望值 2，而是 1

为了演示 GIL 对多线程编程的影响，让我们来看看计算斐波那契数列中第 n 个数

的 CPU 密集型任务。下面将使用一个相当慢的算法来演示一个耗时的操作。更好的解决方案应该是利用内存或数学算法来提高性能。

**代码清单 1-5　生成斐波那契数列并计算时间**

```
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) is {fib(number)}')

def fibs_no_threading():
    print_fib(40)
    print_fib(41)

start = time.time()

fibs_no_threading()

end = time.time()

print(f'Completed in {end - start:.4f} seconds.')
```

这个实现使用递归，总体上是一个较慢的算法，需要指数级 $O(2^N)$时间才能完成。如果需要输出两个斐波那契数列，那么同步调用它们，并为结果计时是很容易的，就像在前面的代码清单中所做的那样。

根据运行的 CPU 的速度，将看到不同的时序，但运行代码清单 1-5 中的代码将产生如下的输出：

```
fib(40) is 63245986
fib(41) is 102334155
Completed in 65.1516 seconds.
```

这是一个相当长的计算，但对 print_fib 的函数调用是相互独立的。这意味着它们可放在多个线程中，理论上 CPU 的多个内核可同时运行，从而加快应用程序的运行速度。

**代码清单 1-6 多线程的斐波那契数列**

```
import threading
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

def fibs_with_threads():
    fortieth_thread = threading.Thread(target=print_fib, args=(40,))
    forty_first_thread = threading.Thread(target=print_fib, args=(41,))

    fortieth_thread.start()
    forty_first_thread.start()

    fortieth_thread.join()
    forty_first_thread.join()

start_threads = time.time()

fibs_with_threads()

end_threads = time.time()

print(f'Threads took {end_threads - start_threads:.4f} seconds.')
```

在代码清单 1-6 中，我们创建了两个线程，一个用于计算 fib(40)，一个用于计算 fib(41)，并通过在每个线程上调用 start()来同时启动它们。然后调用 join()，这将导致主程序等待线程完成。鉴于同时开始计算 fib(40)和 fib(41)，并同时执行它们，你会认

为可看到合理的加速。但即使在多核 CPU 的机器上，看到的结果依旧如下所示，没有加速。

```
fib(40) is 63245986
fib(41) is 102334155
Threads took 66.1059 seconds.
```

线程版应用程序花费了几乎相同的时间。事实上，它甚至慢了一点！这几乎完全是由 GIL 以及创建和管理线程所产生的开销造成的。虽然线程确实是并发运行的，但由于锁的缘故，一次只允许其中一个线程运行 Python 代码。这使另一个线程处于等待状态，直到第一个线程执行完成，完全否定了多个线程的价值。

### 1.5.1　GIL 会释放吗

基于前面的例子，你可能想知道 Python 中的线程是否会发生并发，因为 GIL 会阻止并发运行两行 Python 代码。然而，GIL 并不是永远保持不变的，所以还可以通过多线程提升性能。

当 I/O 操作发生时，全局解释器锁被释放。这让我们可在涉及 I/O 时，使用多个线程来执行并发工作，但不用于 CPU 密集型的 Python 代码本身(某些情况下，有一些值得注意的例外情况会为 CPU 密集型工作释放 GIL，我们将在后续章节中讨论这些问题)。为说明这一点，让我们观察一个读取网页状态代码的例子。

**代码清单 1-7　同步读取状态码**

```
import time
import requests

def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)

sync_start = time.time()

read_example()
read_example()

sync_end = time.time()
```

```
    print(f'Running synchronously took {sync_end - sync_start:.4f}
seconds.')
```

在代码清单 1-7 中，我们检索 example.com 的内容并输出状态代码两次。根据网络连接速度和我们的位置，当运行这段代码时，将看到如下的输出：

```
200
200
Running synchronously took 0.2306 seconds.
```

现在已经有了一个同步版本的基线，可编写一个多线程版本进行比较。在多线程版本中，为尝试同时运行它们，将为对 example.com 的每个请求创建一个线程。

**代码清单 1-8　通过多线程读取状态码**

```
import time
import threading
import requests

def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)

thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()

thread_1.start()
thread_2.start()

print('All threads running!')

thread_1.join()
thread_2.join()

thread_end = time.time()

print(f'Running with threads took {thread_end - thread_start:.4f}
seconds.')
```

执行代码清单 1-8 时，会看到如下的输出，这同样取决于网络连接条件和地理位置：

```
All threads running!
200
200
Running with threads took 0.0977 seconds.
```

这大约比不使用多线程的原始版本快两倍，因为几乎在同一时间运行了两个请求！当然，根据你的网络连接情况和机器配置，你将看到不同的结果，但数字的变化比例应该与上面的示例相似。

那么，如何为 I/O 释放 GIL 而不是为 CPU 密集型操作释放 GIL 呢？答案在于在后台进行的系统调用。对于 I/O 而言，低级系统调用在 Python 运行时之外。这允许 GIL 被释放，因为它不直接与 Python 对象交互。这种情况下，GIL 只有在接收到的数据被转换回 Python 对象时才会被重新获取。然后，在操作系统级别，I/O 操作并发执行。这个模型提供了并发，但没有提供并行。在其他语言(如 Java 或 C++)中，可在多核机器上获得真正的并行，因为它们没有 GIL，可同时执行。然而，在 Python 中，由于 GIL 的存在，我们能做的就是并行化 I/O 操作，而且在给定时间内只有一段 Python 代码在执行。

### 1.5.2　asyncio 和 GIL

asyncio 利用 I/O 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 I/O 操作一样，也可让多个协程一起运行。在等待 I/O 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 I/O 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。

## 1.6　单线程并发

在上一节中，我们介绍了多线程作为一种实现 I/O 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，I/O 操作可以并发完成。为更好地理解这一点，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，然后信差拿起信，并递送到收件人的邮箱。收件人打开邮箱，打开信。根据内容的不同，收件人可能给你回信。在这个类比中，可将字母看作想要发送的数据或字节，将信件放入邮箱的操作看作将字节写入套接字，将打开邮箱读取信件的操作看作从套接字读取字节，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 example.com 获取内容的情况下，打开一个连接到 example.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：在本例中，是 Web 页面的 HTML。图 1-7 中显示了进出服务器的字节流。

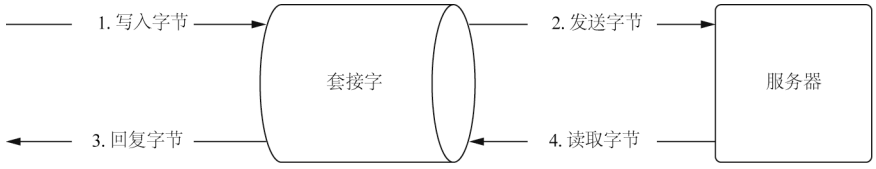


图 1-7　向套接字写入字节，并从套接字读取字节

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，会停止应用程序或阻塞它，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，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(I/O 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 I/O 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 线程就可以自由地继续运行其他 Python 代码，或者为操作系统添加更多的非阻塞套接字来跟踪。I/O 操作完成时，“唤醒”正在等待结果的任务，然后继续运行该 I/O 操作之后出现的其他 Python 代码。可在图 1-8 中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。

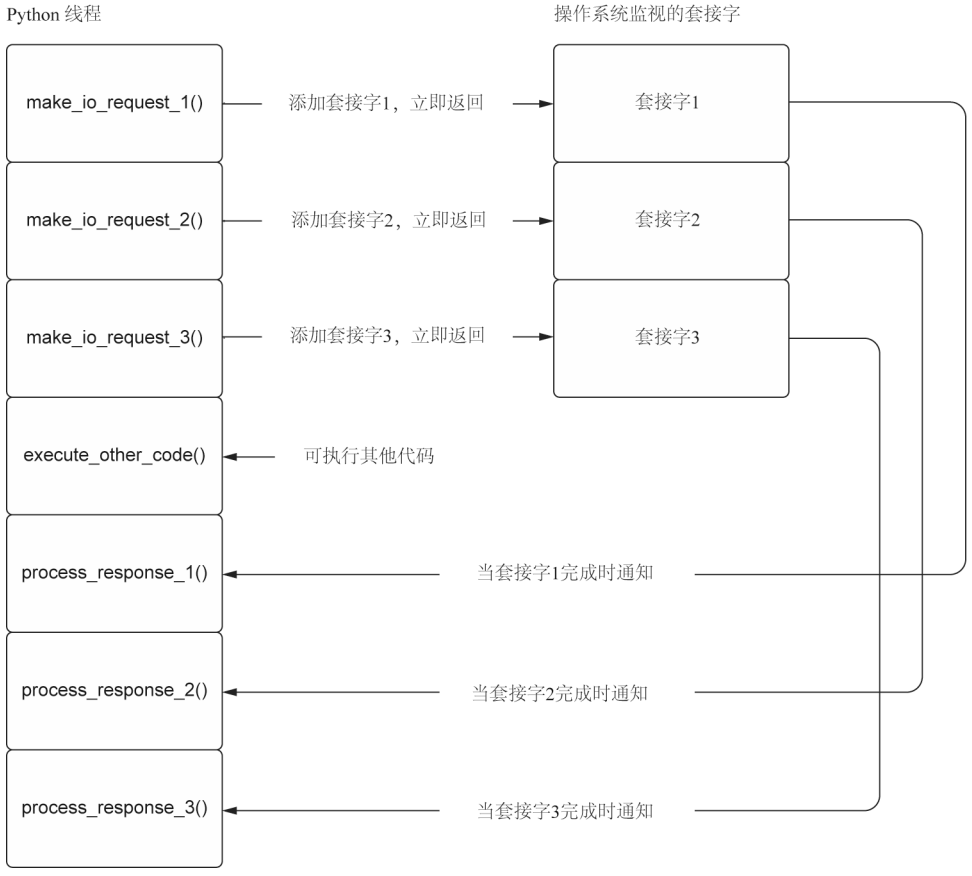


图 1-8　发出的非阻塞 I/O 请求会立即返回，并告诉 O/S 监视套接字中的数据。这允许 execute_other_code() 立即运行，而不是等待 I/O 请求完成。稍后可在 I/O 完成时收到通知并处理响应