数据科学与大数据技术

极速 Python：

高性能编码、计算与数据分析

[美] 蒂亚戈·罗德里格斯·安道(Tiago Rodrigues Antão) 著

沈 冲 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：01-2024-0792

Tiago Rodrigues Antão
Fast Python: High performance techniques for large datasets
EISBN: 9781617297939

图书在版编目(CIP)数据

极速 Python：高性能编码、计算与数据分析 / (美)蒂亚戈 • 罗德里格斯 • 安道 (Tiago Rodrigues Antão) 著；沈冲译. —北京：清华大学出版社，2024.3
(数据科学与大数据技术)
书名原文：Fast Python: High performance techniques for large datasets
ISBN 978-7-302-65629-6

I. ①极… II. ①蒂… ②沈… III. ①软件工具—程序设计 IV. ①TP311.561

中国国家版本馆 CIP 数据核字(2024)第 036286 号

责任编辑：王 军
装帧设计：孔祥峰
责任校对：马遥遥
责任印制：宋 林

出版发行：清华大学出版社
网 址：https://www.tup.com.cn，https://www.wqxuetang.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：北京同文印刷有限责任公司
经 销：全国新华书店
开 本：170mm×240mm 印 张：16.25 字 数：375 千字
版 次：2024 年 3 月第 1 版 印 次：2024 年 3 月第 1 次印刷
定 价：79.80 元

产品编号：097309-01

作 者 简 介

Tiago Rodrigues Antão 拥有信息学工程学士学位和生物信息学博士学位。他目前从事生物技术工作。Tiago 使用 Python 生态来处理科学计算和数据工程任务。大多数时候，他也使用底层编程语言(如 C 和 Rust)对算法的关键部分进行优化。目前，他在基于亚马逊 AWS 的云计算设备上进行开发，但他使用的基本是本地计算集群。

除了业内经历，他在科学计算方面有两段学术经历，包括在剑桥大学和牛津大学从事数据分析博士后研究工作。作为蒙大拿大学的研究员，他从零开始创建了用于分析生物数据的整套科学计算方法。

Tiago 是重要生物信息软件包 *Biopython*(用 Python 编写)的共同作者之一，也是 *Bioinformatics with Python Cookbook*(Packt 出版社，2022)一书的作者，该书已出版了第 3 版。他还在生物信息学领域发表了许多重要的科研论文。

致　谢

我要感谢编辑 Frances Lefkowitz 给予我的细致和耐心。还要感谢我的女儿和妻子，在我编写本书的几年时光里，无法陪伴在她们身边。感谢 Manning 出版社编辑团队齐心合力，促成本书出版。

向所有审稿人致以敬意，他们是：Abhilash Babu Jyotheendra Babu、Andrea Smith、Biswanath Chowdhury、Brian Griner、Brian S Cole、Dan Sheikh、Dana Robinson、Daniel Vasquez、David Paccoud、David Patschke、Grzegorz Mika、James Liu、Jens Christian B. Madsen、Jeremy Chen、Kalyan Reddy、Lorenzo De Leon、Manu Sareena、Nik Piepenbreier、Noah Flynn、Or Golan、Paulo Nuin、Pegah T. Afshar、Richard Vaughan、Ruud Gijsen、Shashank Kalanithi、Simeon Leyzerzon、Simone Sguazza、Sriram Macharla、Sruti Shiva kumar、Steve Love、Walter Alexander Mata López、William Jamir Silva 和 Xie Yikuan。感谢他们提出宝贵的建议，使本书质量更上一层楼。

关于封面插图

封面人物是“Bourgeoise de Passeau”或“Bourgeoise of Passeau”，即“帕绍的居民”，取自 Jacques Grasset de Saint-Sauveur 于 1797 年出版的一本图集。其中的每幅插图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识别其居所、职业、社会地位。Manning 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，以此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。

前　言

若干年前，我们团队正在使用的基于 Python 的数据管道突然崩溃，导致某个进程持续占用 CPU。该组件对公司业务至关重要，因此必须尽快解决该问题。我们核查了算法，始终没有发现问题。算法的实现步骤其实非常简单，但经过多名工程师的数小时排查，才发现问题在于程序在一个非常大的列表上进行搜索。在将列表转换为集合后，问题就迎刃而解了。最终，数据结构不仅变得更小，搜索时间也从数小时降低到毫秒级别。

这次故障对我触动很大：

- 虽然问题并不严重，但暴露出团队在开发过程中并不关注性能问题。如果经常使用代码分析器，我们就能在几分钟内发现问题，而不是耗费了好几个小时。
- 我们最终解决了问题，并且取得了双赢的结果，不仅程序查询时间更短，占用内存也更少。虽然在许多情况下，面对性能和成本需要做出取舍，但在某些情况下，兼顾两者不仅能获得满意的结果，还没有任何负面影响。
- 从更高的角度审视，结果也是双赢的。首先，查询速度更快非常有利于公司业务。其次，算法经过优化后，使用 CPU 的时间更短、耗能更低，也更加环保。
- 虽然单个案例意义有限，但我意识到许多程序员或许都在寻找类似的优化解决方案。

因此，我决定编写本书，以便其他程序员可以从中受益。我的目标是帮助经验丰富的 Python 程序员设计和实现更高效的解决方案，同时能够了解底层的权衡机制。我将采用全面且透彻的方式，通过探讨 Python 代码和重要的 Python 库，从算法角度来探究现代硬件架构及其影响，并分析 CPU 和存储性能。希望本书能够帮助读者在使用 Python 生态进行开发时，游刃有余地处理性能问题。

关 于 本 书

本书旨在帮助读者在 Python 生态系统中编写更高效的应用程序。更高效是指让代码使用的 CPU 周期更少、占用的存储空间更少、消耗的网络通信更少。

本书采用全面且透彻的方式来分析性能问题。书中不仅涉及纯 Python 代码的优化技术，还介绍了如何高效使用数据科学库，如 NumPy 和 pandas。由于 Python 在某些场景下性能不足，为了使代码运行速度更快，本书还讲解了 Cython。为了使内容尽量全面，本书还讨论了硬件对代码设计的影响，即分析现代计算机架构对算法性能的影响。另外，还探究了网络架构对效率的影响，以及 GPU 计算在快速数据分析领域中的运用。

目标读者

本书面向中高级读者。希望读者接触过目录中的大部分技术，并且最好亲手使用过其中一些。除了 IO 库和 GPU 计算章节，本书只提供了很少的介绍性内容，所以要求读者了解基础知识。如果你正在编写高性能代码，并面临高效处理海量数据的切实挑战，这本书可提供很多建议。

为了从本书吸收更多的知识，你应该具备多年的 Python 编程经验，熟悉 Python 控制流、列表、集合和字典，并具备一定 Python 标准库的使用经验，如 os、sys、pickle 和 multiprocessing。为了充分运用书中介绍的技术，你或多或少应操作过标准的数据分析库，即 NumPy 和 pandas，例如使用过 NumPy 中的数组和 pandas 中的数据帧。

如果你清楚如何通过 C 或 Rust 等语言接口来加速 Python 代码，或者了解 Cython 或 Numba 等优化工具，即使没有实际操作过，本书对你也会很有帮助。在 Python 中处理 IO 的经验对你也会有所帮助。鉴于 IO 库方面的学习资料较少，我们将从 Apache Parque 和 Zarr 库开始介绍。

读者应该熟练使用 Linux 终端(或 MacOS 终端)的基本命令。如果使用 Windows，请安装基于 UNIX 的终端，并熟悉命令行或 PowerShell。此外，你还需要在计算机上安装 Python。

在某些示例中，我会讲解云计算技巧，但访问云或了解云计算方面的知识不是阅读本书的必要条件。如果你对云计算方法感兴趣，建议学习执行云计算的基本操作，例如，创建实例和访问云服务存储。

虽然本书不要求读者接受过学术培训，但了解复杂度的基本概念是有益的。例如，理解与数据量呈线性关系的算法优于与数据量呈指数关系的算法。如果你想使用 GPU 进

行优化，阅读本书前无需了解相关内容。

本书内容：学习路线图

本书各章基本是独立的，读者可以翻阅任何感兴趣的章节。全书内容分为四部分。

第Ⅰ部分“基础知识”(第 1~4 章)，涵盖了入门知识。

- 第 1 章介绍了海量数据带来的问题，并解释了为什么必须关注计算和存储的效率。本章还介绍了全书的学习方法，并提供了如何根据需求学习本书的建议。
- 第 2 章讲解了对原生 Python 代码进行优化的方法。本章还讨论了 Python 数据结构的优化、代码分析、内存分配和惰性编程技术。
- 第 3 章讨论了 Python 中的并发和并行，以及如何最佳利用多进程和多线程(包括使用线程进行并行处理的限制)。本章还介绍了异步处理，它通常用于 Web 服务中的低负载、多并发请求场景。
- 第 4 章介绍了 NumPy，NumPy 是用于高效处理多维数组的库。NumPy 是当前所有数据处理技术的核心，是众多方法的基本库。本章分享了 NumPy 中的关键功能，如视图、广播和数组编程，以开发更高效的代码。

第Ⅱ部分“硬件”(第 5 章和第 6 章)，主要涉及如何发挥硬件和网络的最大效率。

- 第 5 章介绍了 Cython。Cython 基于 Python，可以生成非常高效的代码。Python 属于高级解释性语言，因此不适合用于优化硬件。其他几种语言，如 C 或 Rust，从设计之初就能在硬件层高效运行。Cython 属于底层语言，虽然它与 Python 很相似，但可以将 Cython 编译成 C 代码。要想写出高效的 Cython 代码，关键在于掌握代码基础和实现方法。在本章中，我们就将学习如何编写高效的 Cython 代码。
- 第 6 章讨论了硬件架构对高效 Python 代码设计的影响。鉴于计算机的设计模式，反直觉的编程方法可能比预期的更有效率。例如，在某些情况下，即使需要承担解压缩算法的计算开销，处理压缩数据也可能比处理未压缩数据的速度更快。本章还介绍了 CPU、内存、存储和网络对 Python 算法设计的影响。另外，本章讨论了 NumExpr 库，它能利用最新的硬件架构特性使 NumPy 代码更高效。

第Ⅲ部分“用于现代数据处理的应用和库”(第 7 章和第 8 章)，探讨了最新的用于数据处理的应用和库。

- 第 7 章讨论了如何高效使用 pandas，pandas 是 Python 中的数据帧库。我们将研究 pandas 相关的技术，以优化代码。和本书大多数章节不同，这一章的内容基于前面的章节。pandas 是在 NumPy 的基础上工作的，所以会借鉴第 4 章的内容，探索与 NumPy 相关的技术进而优化 pandas。我们还将探究如何用 NumExpr 和 Cython 优化 pandas。最后，介绍了 Arrow 库，除了可以提高处理 pandas 数据帧的性能，它还具有其他强大的功能。

- 第 8 章探讨了数据持久化的优化问题。我们讨论了能高效处理列型数据的库 Parquet，以及能处理大型磁盘数组的库 Zarr。本章还讨论了如何处理超过内存容量的数据集。

第Ⅳ部分“高级主题”(第 9 章和第 10 章)，涉及最后两个与众不同的方法，即 GPU 和 Dask 库。

- 第 9 章探讨了图形处理单元(Graphical Processing Unit，GPU)在处理大型数据集方面的使用方法。我们将看到，GPU 计算模型使用大量简单的处理单元，完全可以处理最新的数据科学问题。我们使用了两种不同的方法以利用 GPU 的优势。首先，将讨论现有 GPU 方面的库，这些库提供了与其他库类似的接口，例如 CuPy 是 NumPy 的 GPU 版本。另外，本章还介绍如何编写在 GPU 上运行的 Python 代码。
- 第 10 章讨论了 Dask。使用 Dask 库能编写出轻松扩展到计算集群上的并行代码，它提供了类似于 NumPy 和 pandas 的接口。计算集群既可以位于本地，又可以位于云端。

该书最后还包括两个附录。

- 附录 A 介绍如何安装所必需的软件，以使用书中的示例。
- 附录 B 介绍 Numba。Numba 是 Cython 的替代品，可以生成高效的底层代码。Cython 和 Numba 是生成底层代码的主要途径。为了解决真实场景中的问题，我推荐使用 Numba。但是，为什么正文中用了一整章来介绍 Cython，却将 Numba 放在附录中呢？这是因为本书的主要目标是为你打下坚实的基础，确保能在 Python 生态系统中编写出高效的代码，而 Cython 更适合深入了解内部的计算机制。

获取代码

本书包含许多源代码示例，既有行间代码，又有行内代码。为了区别于普通文本，这两种样式的源代码都采用等宽字体。有时，也会加粗代码，以强调代码发生的变动，例如，为已有代码添加新功能。

在许多示例中，不得不对初始源代码的样式进行调整。为了使页面版式更加美观，添加了换行符，并重新进行排版缩进。但在极少数情况下，如此调整后还要使用连行符(➥)。此外，如果文本中对代码进行了解释，通常会删除源代码中的注释。许多代码附带有注释，以突出重要概念。

本书示例的完整代码可从 GitHub 仓库(https://github.com/tiagoantao/python-performance)下载，或扫描本书封底的二维码进行下载。当发现错误或需要更新 Python 和库时，我会更新代码仓库。因此，请读者留意代码仓库的变动。代码仓库为每个章节列出了详细目录。

人们对代码风格各有偏好，我尽量调整了书中的代码，以使代码在纸质版的书中看起来更加美观。例如，我本来喜欢使用较长且具有描述性的变量名，但这种名称的印刷效果一般。因此，书中使用表达性的变量名，并遵循标准的 Python 惯例，如 PEP8，同时确保印刷效果的美观。本书对类型注释也采取了同样的做法，我原本想使用类型注释，但类型注释妨碍了代码的可读性。在极少数情况下，我使用算法来增强可读性，但算法

并不能应对所有极端情况，或使代码逻辑更加清晰。

在大多数情况下，本书中的代码适用于标准的 Python 解释器。在少数情况下，需要使用 IPython，特别是进行性能分析时。你也可以使用 Jupyter Notebook。

关于安装的细节可以参考附录 A。如果任何章节需要使用特殊的软件，将在适当的地方注明。

硬件和软件

读者可以使用任何操作系统运行本书中的代码。不过，大部分生产代码都部署在 Linux 上，因此 Linux 是首选系统。或者，也可以直接使用 MacOS X。如果使用的是 Windows，建议安装 Windows Subsystem for Linux(WSL)[1]。

除了操作系统，还可以选择 Docker。读者可以使用代码仓库中提供的 Docker 镜像，Docker 提供了容器化的 Linux 环境来运行代码。

建议你使用的计算机至少拥有 16 GB 的内存和 150 GB 的可用磁盘空间。第 9 章涉及与 GPU 相关的内容，需要使用 NVIDIA GPU，最低基于 Pascal 架构。过去五年中发布的大多数 GPU 都符合此要求。为了充分学习本书内容，有关环境准备的详细信息，请参见附录 A。

1 译者注，WSL 的安装方法可参考 https://learn.microsoft.com/en-us/windows/wsl/install#install-wsl-command。

目　录

第 I 部分

基础知识

第I部分讨论有关 Python 性能的基础方法。介绍 Python 原生库和基本数据结构，以及 Python 在没有外部库的情况下如何使用并行处理技术。本部分还专门用一整章探讨如何优化 NumPy。虽然 NumPy 是外部库，但它对数据处理至关重要，因此 NumPy 和 Python 原生方法一样都是基础。

第1章

对高效数据处理的迫切需求

本章内容

- 处理数据指数级增长所面临的挑战
- 比较传统和最新的计算架构
- Python 在数据分析中的作用和不足之处
- Python 高效计算方法

人们正以前所未有的速度和广度采集海量数据。但是，采集数据的过程具有很强的随意性和盲目性，不管如何处理、存储、访问、提炼数据，也不管数据的最终用途是什么，只是先将数据收集起来。在数据科学家分析数据之前，在设计师、政策制定者、开发者基于数据打造产品、提供服务、编写程序之前，软件工程师必须找到存储和处理大数据的方法。现在，大数据工程师比起以往任何时候，都更需要高效的方法来提高性能和优化存储。

在本书中，我分享了大量在自己工作中使用的提高性能和优化存储的策略。当发生问题时，简单粗暴地投入更多机器往往既不现实，也没有任何帮助。因此，本书中介绍的解决方案更多是靠理解和利用现有的东西：编码方法、硬件架构、系统架构、软件，当然还涉及 Python 语言、库和生态。

为了应对数据泛滥带来的问题，人们首选 Python 作为编程语言，并将 Python 作为使用其他编程语言的“胶水语言”。由于 Python 在数据科学和数据工程中使用广泛，进一步推动了 Python 的发展。根据 TIOBE 编程指数，Python 是目前最受欢迎的编程语言之一。在处理大数据方面，Python 有其独特的优势和局限性，但 Python 运行速度不够快，带来了一定的挑战。不过正如本书在后文展示的，有许多不同的方法和技巧，可以使 Python 更高效地处理大量数据。

在讨论解决方案之前，需要全面了解问题，这是第 1 章的主要目标。我们将更深入地讨论数据泛滥所带来的计算挑战，明确到底要处理什么问题。接下来，将讨论硬件、网络和云计算架构的作用，以了解为什么过去的解决方案(如提高 CPU 速度)无法解决当前的问题。然后，将讨论 Python 在处理大数据时面临的特殊挑战，包括 Python 的线程和 CPython 的全局解释器锁(Global Interpreter Lock，GIL)。在明确需要运用新方法才能使

Python 更加高效后，我会简要概述后文涉及的解决方案，这样读者在学习过程中就会变得轻松很多。

1.1 数据泛滥的严重性

你可能知道这两个计算定律，即摩尔定律(Moore's Law)和埃德霍尔姆定律(Edholm's Law)，后者指出了数据的指数式增长规律，前者则指出计算机处理数据的能力滞后于数据的增长。埃德霍尔姆定律表明通信数据速率每 18 个月翻一番，而摩尔定律预测处理器上可容纳的晶体管数量每两年翻一番。可以用埃德霍尔姆的数据传输率来表示采集的数据量，将摩尔的晶体管密度作为计算硬件速度和容量的指标。当对二者进行比较时，可以发现处理和存储数据的能力与采集数据的速度和数量之间存在 6 个月的滞后。因为用文字表达指数增长可能难以理解，所以我把这两个定律绘制在同一幅图中进行比较，如图 1.1 所示。

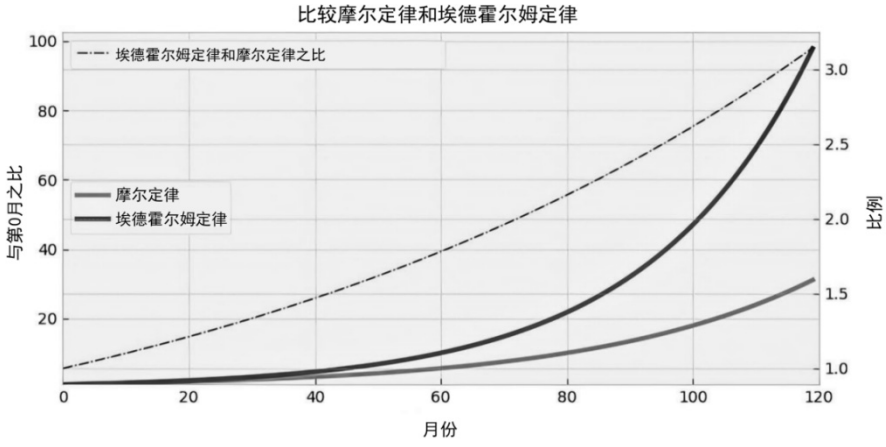

图 1.1 摩尔定律与埃德霍尔姆定律之比说明，硬件总是滞后于生成的数据量。并且，二者差距不断拉大

图 1.1 描述了待分析数据(埃德霍尔姆定律)与数据分析能力(摩尔定律)之间的不对称关系。其实，实际情况比图 1.1 描绘的更加严重。在第 6 章讨论现代 CPU 架构和摩尔定律时，我们将分析原因。本章先重点讨论数据增长，让我们看一个示例，网络流量是间接衡量数据的途径。如图 1.2 所示，近 30 年的网络流量增长曲线大致符合埃德霍尔姆定律。

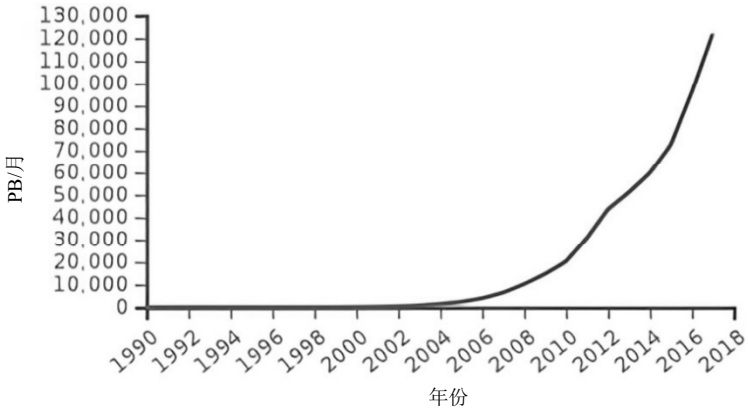

图 1.2　近 30 年的全球互联网流量增长情况，单位为每月百万亿字节(来源：https://en.wikipedia.org/ wiki/Internet_traffic)

此外，人类产生的数据中，有 90%是近两年产生的(参考《大数据及其意义》，http://mng.bz/v1ya)。不过，这些新数据的质量与其规模的相关性尚待研究。这里的问题是，人们需要处理生成的数据，但处理数据需要耗费大量资源。

软件工程师面临的问题不仅仅是数据规模，数据结构也在变化。有人预测，到 2025 年，大约 80%的数据可能是非结构化的(参考《发挥非结构化数据的力量》，http://mng.bz/BlP0)。简单来讲，从计算角度衡量，非结构化数据对数据处理的要求更高，我们将在后文详细讨论非结构化数据。

面对数据泛滥，我们又是如何应对如此海量的数据增长呢？事实证明，人们几乎无所作为。据《卫报》报道(http://mng.bz/Q8M4)，超过 99%的数据从未被分析过。阻碍人们挖掘数据价值的部分原因，是缺少高效的程序以对数据进行分析。

伴随数据增长和随之而来的数据处理需求，流传着这样一句口头禅，即“面对数据增长只需投入更多的服务器”。这句话是不对的，由于许多原因，投入更多的服务器往往不是可行或适宜的解决方案。相反，当需要提升现有系统的性能时，应该深入系统架构和实现，找到性能瓶颈并进行优化。我已经数不清有多少次在审查现有代码时，仅仅通过改正效率缺陷，就使性能提高了十倍之多。

最关键的是要理解，数据量大小和设备规模之间并不是线性关系。相比于机器，解决大数据问题需要对开发过程本身投入更多的时间和精力。这不仅适用于云环境，也适用于内部集群和单机实现。举一些示例，如下所示：

- 你的解决方案只需要一台计算机，但突然间你需要更多机器。增加机器意味着必须管理大量机器，在机器之间分配计算任务，并确保数据正确分区。你可能还需要一台文件系统服务器。维护服务器集群或维护云的成本，要比维护单台计算机高得多。
- 你的解决方案在内存中运行良好，但后来数据量增加，超过了内存大小。为了处理存储在磁盘中的新增数据，通常需要重构大量的代码。同时，代码本身的复杂

度也会增加。例如，如果主数据库位于磁盘之上，你可能需要创建缓存策略。或者可能需要在多个进程中进行并发读取，或者更糟糕的是要进行并发写入。

- 你使用的是 SQL 数据库，突然达到了服务器的最大吞吐能力。如果这只是读取容量的问题，可能只需要创建若干读取副本就可以了。但如果是写入问题，该怎么做呢？你可以采用分片[1]，或者转而使用其他性能更优的 NoSQL 数据库。
- 如果你使用的是供应商提供的云计算平台，你可能发现无限扩展性能更多是营销噱头，技术上难以实现。在许多场景下，如果你遇到性能瓶颈，唯一可行的解决方案是改进正在使用的方法，但优化过程需要投入大量的精力、财力和人力。

希望这些示例能够说明，仅靠“增加机器”不能带来性能的增长，而是需要做出多方面的努力，以应对系统增加的复杂度。即使是在单台计算机上使用并行计算这样相对“简单”的方法，也会带来并行处理的所有问题(竞争条件、死锁等)。使用更有效的解决方案，能对复杂性、可靠性和成本产生巨大的影响。

最后，事实证明，即使能做到线性扩展计算资源(实际上做不到)，也会产生伦理和生态问题：据相关预测，与“数据海啸”有关的能源消耗占全球电力生产的 20%(参考《数据海啸》，http://mng.bz/X5GE)，而且在更新硬件、处理旧设备的同时，也存在处理废弃垃圾的问题。

好消息是，在处理大数据时提高计算效率有助于降低计算开销、降低解决方案架构的复杂性、减少存储需求、缩短开发周期，以及节省能源。有时，更高效的解决方案也可能是价格最低廉的。例如，使用恰当的数据结构，只需少量开发，就能减少计算时间。

另一方面，我们要探究的许多解决方案都存在开发成本，并且会增加一定的复杂度。当查看数据并预测数据增长时，你必须做出判断以进行合理的优化，这是因为没有万金油或一刀切的解决方案。也就是说，可能只有一条规则可以全面适用：适用于奈飞、谷歌、亚马逊、苹果或脸书的解决方案，可能对你没有帮助，除非你是这些公司中的一员。

大多数人接触的数据量远远低于科技巨头公司的数据量。尽管你接触的数据量很大，也很复杂，但比起大公司的数据量，它可能仍然要低几个数量级。有些人认为适用于大公司的方法同样适用于小公司或个人，在我看来，这种看法是错误的。通常，不那么复杂的解决方案往往更适合大众。

世界正在快速发展，数据和算法的规模、复杂度在极速增长，我们需要更复杂的技术，以高效和低成本的方式进行计算和存储。只是在必要的时候，才应该添加硬件、扩展计算资源。当你搭建架构、实施解决方案时，虽然用到的技术可能不同，但仍可以使用同样的思维方式关注效率问题。

1.2 现代计算架构和高性能计算

创建更高效的解决方案是非常具体的任务。首先，我们要划定问题的范围，也就是弄清要解决的实际问题是什么。同样重要的是应确定计算架构，我们将在该计算架构中

1 分片是指对数据进行分区，使不同数据位于不同的服务器。

运行解决方案。计算架构对设计出最佳的优化方法至关重要，所以必须对其加以考量。本节将了解影响解决方案设计和实施的主要架构问题。

1.2.1　计算机内部的变化

计算机内部发生了翻天覆地的变化。首先，对于最新的 CPU，其处理能力的提高主要归功于并行单元的数量，过去主要依靠 CPU 内核的运算速度。计算机还可以配备图形处理单元(Graphics Processing Unit，GPU)，这些单元最初只是用于图形处理，但现在也可用于通用计算。事实上，许多人工智能算法的高效实现都是依靠 GPU 完成的。不过，至少从用户角度来看，GPU 的架构与 CPU 完全不同：GPU 由成千上万的计算单元组成，并在所有单元中进行相同的“简单”计算。二者的内存模型也完全不同。这些差异意味着，对 GPU 进行编程需要采用与 CPU 编程完全不同的方法。

为了理解如何使用 GPU 进行数据处理，需要了解 GPU 的设计初衷和架构思想。正如其名，GPU 最初是用来进行图形处理的。游戏其实是对计算要求最高的应用。游戏和常见的图形应用持续不断地更新屏幕上的数百万个像素点。为解决这一问题，GPU 硬件架构设计有许多小型处理核心。单个 GPU 很容易拥有数千个内核，而 CPU 通常具有不到 10 个内核。GPU 内核简单得多，并且大多数内核上运行着相同的代码。因此，GPU 非常适合运行大量相似的任务，如更新像素。

基于 GPU 的强大处理能力，出现了图形处理单元通用计算(General-Purpose computing on Graphics Processing Units，GPGPU)，人们尝试将其用于其他任务。由于 GPU 架构的组织方式，GPU 特别适用于大规模并行任务。许多现代人工智能算法，如基于神经网络的算法，往往是大规模并行的。因此，GPU 和并行算法非常契合。

不过，CPU 和 GPU 之间的区别不仅在于内核数量和复杂度。GPU 显存是与主存分离的，具有很强算力的 GPU 尤其如此。因此，在主存和 GPU 内存之间存在传输数据的问题。因此，在使用 GPU 时，必须要考虑内核数量和显存问题。

我们将在第 9 章详细阐释，用 Python 对 GPU 进行编程比对 CPU 编程难得多，也更加烦琐。尽管如此，利用 GPU 进行 Python 编程，可以大大提高代码性能。

虽然没有取得 GPU 那样显著的进步，CPU 编程方式也发生了巨大的变化。不同于 GPU，CPU 编程中的大部分变化都可以通过 Python 实现。与过去相比，制造商通过不同的方法来提升 CPU 性能。基于物理学定律，CPU 制造商的解决方案是创建更多的并行处理，而不是将精力放在提高单核的速度上。有时，摩尔定律被表述为 CPU 速度每 24 个月翻一番，但准确的说法是晶体管密度每两年翻一番。十多年前，CPU 速度和晶体管密度之间就不再是线性关系了，CPU 速度增长不多，进入了平台期。但是，数据与算法复杂度持续增长，越来越难以进行处理。CPU 制造商最先提出了解决方案，即允许 CPU 进行更多的并行计算，为每台计算机配备更多的 CPU、每个 CPU 有更多的内核、在内核中展开多线程。处理器不再是纯粹的顺序计算，而是进行更多的并发执行。并发执行需要对计算机编程方式进行范式转变。以前，更换 CPU 后，程序运行速度能得到提高。但是现在，程序速度能否提高取决于开发者是否意识到底层架构向并行编程范式转变。

对于最新的 CPU，其编程方式有很多变化，第 6 章将对其进行深入分析，其中一些

变化是非常反直觉的，必须引起注意。例如，虽然近年来 CPU 的速度增长不多，但 CPU 仍然比 RAM 快几个数量级。如果 CPU 中不存在缓存，那么大部分时间 CPU 都会闲置，这是因为 CPU 大部分时间都在等待 RAM。这意味着，有时使用压缩数据(包括解压缩的开销)比使用原始数据要快。这是因为如果将压缩数据放在 CPU 缓存中，则原本闲置的等待访问 RAM 的 CPU 周期就可以用来解压数据，而原本用于解压的 CPU 周期则可以进行计算。压缩文件系统也存在类似的现象，压缩文件系统有时会比原始文件系统更快。在 Python 中，可以直接应用这个特性。例如，通过简单地修改 NumPy 数组内部表征的布尔标识，你可以利用缓存定位大大提升 NumPy 的处理速度。表 1.1 列出了不同类型内存的访问时间和大小，包括 CPU 缓存、RAM、本地硬盘和远程存储。这里不要计较列出的数字是否精确，而是要关注存储大小和访问时间的数量级差异。

表 1.1 台式机不同存储层级的大小和访问时间

类型	存储大小	访问时间
CPU		
L1 缓存	256 KB	2 ns
L2 缓存	1 MB	5 ns
L3 缓存	6 MB	30 ns
RAM		
DIMM	8 GB	100 ns
二级存储		
SSD	256 GB	50 μs
HDD	2 TB	5 ms
三级存储		
NAS(网络接入服务器)	100 TB	取决于网络
云设备	1 PB	取决于供应商

表 1.1 包括了三级存储，三级存储位于计算机之外、网络之上。网络情况也发生了很大的变化，下一节将进行讨论。

1.2.2 网络的变化

在高性能计算环境中，网络既是增加存储的方式，又是增加计算能力的途径。虽然我们偏向使用单台计算机来解决问题，但有时使用计算集群是不可避免的。对拥有多台计算机的云端或本地架构进行优化，是实现高性能计算的必由之路。

使用多台计算机和外部存储引发了一类全新的与分布式计算相关的问题，即网络拓扑结构、跨机器共享数据、管理跨网络运行的进程。有很多示例，例如，在需要高性能和低延迟的服务中使用 REST API 的成本是什么？如何解决远程文件系统的弊端并优化性能？

为了优化网络栈，必须理解网络栈的各个层级，如图 1.3 所示。网络之外是代码和 Python 库，它们对下面的网络层进行选择。在网络栈的顶端，通常用 HTTPS 进行数据传输、用 JSON 作为数据格式。

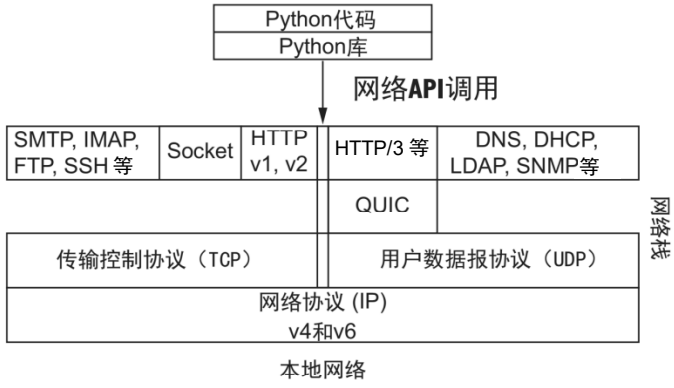

图 1.3　通过网络栈进行 API 调用。了解可用于网络通信的替代方案，可以极大地提高基于网络的应用程序的速度

虽然这对许多应用来说是完全合理的方案，但在对网络速度和延迟要求很高的场景下，还有性能更高的方案。例如，二进制数据格式可能比 JSON 更高效。另外，可以用 TCP 套接字替换 HTTP。也有更激进的方案，例如替换整个 TCP 传输层。大多数网络应用协议都使用 TCP，但有几个例外，如 DNS 和 DHCP，它们都是基于 UDP 的。TCP 协议的可靠性很高，不过要以性能作为代价。有时候不需要如此高的可靠性，则网络开销较小的 UDP 是更高效的选择。

在传输协议之下，是网络协议(IP)和物理计算资源。当设计解决方案时，物理计算资源非常重要。例如，相比于不可靠的网络，如果本地网络非常可靠，则可能丢失数据的 UDP 是更好的选择。

1.2.3　云计算

过去，大部分数据处理都是在单台计算机或本地计算机集群上进行的。目前，基于云的计算资源变成了主流，云中的所有服务器都是“虚拟”的，云由外部公司维护。有时，利用所谓的无服务器(serverless)计算，我们甚至不用直接与服务器打交道。

云计算不仅仅是增加更多的计算机或网络存储，云计算对如何处理存储和计算资源有一套专门的插件，这些插件会影响云计算的性能。此外，虚拟计算机会给 CPU 优化带来麻烦。例如，在裸机中，你可以设计一个考虑到缓存位置问题的解决方案，但在虚拟机中，无法知道缓存是否被另一个同时执行的虚拟机抢占了。如何在云计算环境中保持算法的高效性是个问题。另外，云计算的成本模式完全不同。使用云计算的时间就是金钱，因此高效的解决方案变得更加重要。

云中的许多计算和存储解决方案也是专有的，具有特殊的 API 和功能。使用这些专有的解决方案也会对性能产生影响，应该加以考虑。因此，虽然与传统集群有关的大多数问题也适用于云，但有时会碰到云计算特有的问题。现在我们对架构作用的局限性有了初步了解，知道了架构会对性能造成影响。接下来，探讨 Python 在高性能计算中的优势和劣势。

1.3 Python 的局限性

现代数据处理应用中广泛使用了 Python。与任何语言一样，Python 有优点，也有缺点。使用 Python 有很好的理由，但这里更关心如何应对 Python 在高性能数据处理方面的限制。

毫不掩饰地说，Python 在高性能计算方面的表现非常差。如果性能和并行计算是仅有的考虑因素，就不会有人使用 Python。但是，Python 具有丰富的用于数据分析的库，并且有完善的文档，还有非常好的社区支持。这些理由是使用 Python 的原因。

程序圈中流传着这样一句话：“没有慢语言，只有慢实现。”我不同意这个观点。要求开发者使用 Python 这样的动态高级语言(或者 JavaScript)，在速度方面与 C、C++、Rust 或 Go 这样的底层语言竞争是不公平的。

像动态类型和垃圾回收这样的功能，会对性能造成负面影响。但这样还好，因为在很多情况下，程序员的时间比计算时间更有价值。不过，过多的声明和动态语法会导致计算变慢，并造成内存开销变大。鱼与熊掌，不可兼得。

话虽如此，这不是 Python 语言实现性能不佳的借口。我们可以使用 CPython 重构代码，CPython 是性能最佳的 Python 实现。为了进行简单的分析，可以运行一个矩阵乘法函数并对其计时。然后，用另一个 Python 实现(如 PyPy)来运行它。接着，将代码转换为 JavaScript 并再次计时。使用 JavaScript 是比较公平的，因为 JavaScript 也是动态语言，使用 C 语言进行比较就不公平。

不过，在这个示例中，CPython 的表现并不是那么好。Python 是一门天生较慢的语言，而它的顶级实现似乎并没有把速度作为主要考量。好消息是，现在能克服大部分这样的问题。人们创建了许多应用程序和库，可以解决大多数性能问题。你仍然可以用 Python 写代码，Python 在占用内存不大的情况下表现得非常好。你只需要在写代码的同时，留意 Python 的缺陷。

注意

对于本书的大部分内容，当涉及 Python 时，指的是 CPython 实现。如果不是 CPython 实现，我会明确指出。

鉴于 Python 在性能方面的限制，有时仅仅优化 Python 代码是不够的。在某些情况下，我们会用底层语言重构部分代码，或者注释现有代码，使用代码转换工具重构为底层语言。需要重构的代码通常不多，所以 Python 仍然是最佳选择。当进行最后阶段的优化时，可能 90%以上仍然是 Python 代码。NumPy、scikit-learn 和 SciPy 就是如此，这些库中对

计算要求很高的部分通常用 C 或 Fortran 实现。

全局解释器锁

在关于 Python 性能的讨论中，全局解释器锁(Global Interpreter Lock，GIL)是绕不开的话题。GIL 究竟是什么呢？虽然 Python 有线程的概念，但 CPython 的 GIL 只允许同一时间点执行一个线程。即使在多核处理器上，也只能在一个时间点执行一个线程。

Python 的其他实现，如 Jython 和 IronPython，没有 GIL，因此可以使用多核处理器中的所有内核。但是 CPython 仍然是主流实现，所有的主要库都是为 CPython 开发的。此外，Jython 和 IronPython 分别依赖于 JVM 和.NET。因此，鉴于 CPython 拥有丰富的库，它最终成了默认的 Python 实现。我们将在本书中简要讨论其他实现，特别是 PyPy，但在示例中仍然使用 CPython。

为了理解如何绕过 GIL，有必要理解并发和并行之间的区别。并发是指一定数量的任务可以在时间上重叠，尽管它们可能不是在同一时间运行。例如，它们可以交错运行。并行是指任务在同一时间执行。所以，在 Python 中可以实现并发，但不能实现并行。这种说法对吗？

没有并行的并发仍然是相当有用的。这方面最好的示例是 JavaScript 和 Node.JS，Node.JS 广泛用于 Web 服务器的后端。在许多服务器端的网络任务中，大部分时间实际上是在等待 IO。这是线程自愿放弃控制的好时机，这样其他线程就可以继续进行计算。最新的 Python 中有类似的异步方法，我们将对其进行讨论。

下面回到主要问题，GIL 是否带来了严重的性能问题？在大多数情况下，答案其实是否定的。有两个主要原因：

- 大部分高性能代码，尤其是内部循环代码，可能要用底层语言编写，正如之前所讨论的。
- Python 为底层语言提供了释放 GIL 的机制。

这意味着，当进入用底层语言重构的部分代码时，你可以指示 Python 继续与其他 Python 并行线程工作，并行线程是用底层语言实现的。并且，只在安全的情况下释放 GIL。例如，如果你不写入对象，其他线程就可能使用该对象。

另外，多进程(即同时运行多个进程)不受 GIL 的影响，GIL 只影响线程。所以即使在纯 Python 中，仍然有充足的方法部署并行解决方案。

因此，从理论上讲，GIL 与性能密切相关。但在实践中，很容易处理 GIL 带来的问题。我们将在第 3 章深入探讨 GIL。

1.4　解决方案小结

本书探讨了使 Python 代码获得高性能的方法。因为代码涉及具体场景，所以必须从数据、算法、计算架构等更宽广的视角出发，才能设计出高效的代码。虽然做不到在一本书中探讨架构和算法的方方面面，但我会尽量帮助你理解 CPU、GPU、存储、网络协

议和云架构，以及其他系统方面的考量(如图 1.4 所示)，这样你就能为提高 Python 代码性能做出合理的判断。通过阅读本书，无论是单机、支持 GPU 的计算机、集群，还是云环境，你都能评估其计算架构的优点和缺点，并实施解决方案以充分发挥性能。

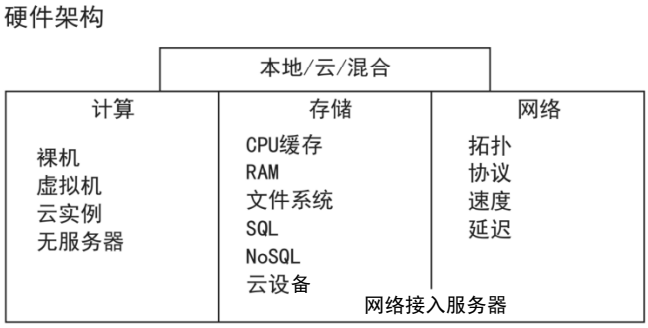

图 1.4 在选择高性能编码方案时，必须考虑底层硬件架构

本书的目标是向读者介绍一系列解决方案，并展示每种解决方案的最佳应用方式，以便你能为特定资源、目标和问题选择并实施最高效的解决方案。我们将花大量时间通过实例进行探究，这样就可以看到这些方法的效果，包括正面和负面的。这里没有规定要应用所有的方法，也没有规定要按特定顺序应用这些方法。每种方法在性能和效率方面都能带来或大或小的收益，但也需要做出取舍。如果你了解系统中可支配的资源，以及改进该系统的可用策略，就可以选择性地使用时间和资源。为了帮助你理解这些方法，表 1.2 列出了书中展示的技术，并对系统开发过程中涉及的组件或领域进行了总结。

表 1.2 本书各章的目标

领域	应用	章节
充分利用 Python 解释器	Python 解释器	第 2 章，发挥内置功能的最高性能
了解 Python 的内置功能，发挥计算机的最佳计算能力	Python 解释器	第 3 章，并发、并行和异步
发挥数据科学基础库的最佳性能	Python 库	第 4 章，高性能 NumPy
当 Python 性能不足时，发掘底层语言的性能	Python 库	第 5 章，使用 Cython 重构核心代码
了解硬件对计算性能的影响	硬件	第 6 章，内存层级、存储和网络
挖掘表格型数据	Python 库	第 7 章，高性能 pandas 和 Apache Arrow
使用最新 Python 持久化库提高存储效率	Python 库	第 8 章，大数据存储
理解 GPU 编程的重要性，并使用 Python 进行 GPU 编程	硬件	第 9 章，使用 GPU 进行数据分析
使用多台计算机处理应用程序	Python 库和硬件	第 10 章，使用 Dask 分析大数据

表 1.2 列出了许多内容，为了避免造成混乱，所以强调一下重点领域的实际应用。读完本书后，你将能够查看原生 Python 代码，并理解内置数据结构和算法的性能影响。你将能够发现并以更合适的解决方案替换低效代码。例如，对于在恒定不变的列表上进行搜索，则将列表替换为集合，或者使用非对象数组替换对象列表以提高速度。针对性能不佳的算法，你还能对代码进行分析，找到导致出现性能问题的部分，并用最佳方法优化这些代码片段。

如前所述，本书使用的是流行的 Python 数据处理和分析库(如 pandas 和 NumPy)，目标是改进其使用方式。在计算方面，这是一个很宽泛的话题，所以我们不会讨论非常高级的库。例如，不会讨论对 TensorFlow 的优化，但会提及使底层算法更高效的方法。

关于数据存储和转换，你将查看数据源，并了解数据格式对高效处理和存储的影响。然后，你将能够对数据进行转换，使所有需要的信息仍然得以保留，但对数据的访问效率将大大提高。最后，你还会学习 Dask，它是基于 Python 的框架，可用于开发并行计算方案，从单台机器扩展到非常大的计算集群或云计算设备。

本书不是实战手册，而是一本介绍优化的思维方式，以及通过什么途径提高性能的书籍。因此，所讨论的方法在大多数情况下应该经得起硬件、软件、网络、系统，甚至是数据本身的变化。虽然不是每种技术都能提高性能，甚至在每种情况下都能派上用场，但完整阅读本书是学习方法、打开思路、制定解决方案的最佳途径。当你接触到各种问题时，可以把本书作为参考，挑选想要使用的方法。

注意

*在继续学习本书之前需要设置软件，一定要查看附录 A 中关于设置环境的详细方法，以便运行每个示例中的代码。*代码仓库是 https://github.com/tiagoantao/python-performance。

1.5　本章小结

- 因为数据量越来越大，所以若想挖掘数据中的价值，必须提高处理数据的效率。
- 算法复杂度的增加造成了额外的计算开销，必须使用适宜的方法以减轻对计算的影响。
- 不同计算架构存在很大差异。现在的网络也包含云计算。计算机内部有强大的 GPU，其计算范式与 CPU 有很大区别。我们需要利用不同的硬件设备。
- Python 是一门杰出的数据分析语言，拥有丰富的数据处理库和框架。但是，Python 在性能方面存在严重问题。我们需要规避这些问题，用复杂的算法处理大量数据。
- 虽然某些要处理的问题可能很棘手，但大多数问题是可以解决的。本书旨在向读者介绍大量可供选择的解决方案，并介绍每种解决方案在特定场景下的最佳使用方法。这样当你碰到具体问题时，就能选择并实施最有效的方法。