并发编程图解

[俄] Kirill Bobrov(基里尔 • 波波洛夫) 著
林 润 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：01-2024-0884
Kirill Bobrov
Grokking Concurrency
EISBN: 9781633439771

图书在版编目 (CIP) 数据

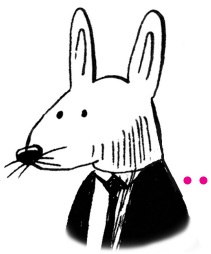
并发编程图解 / (俄罗斯) 基里尔·波波洛夫著 ; 林润译 .
北京 : 清华大学出版社, 2025. 5. -- ISBN 978-7-302-68983-6
Ⅰ. TP312.8-64
中国国家版本馆 CIP 数据核字第 202512XC40 号

责任编辑：王　军
封面设计：高娟妮
版式设计：恒复文化
责任校对：成凤进
责任印制：沈　露

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：大厂回族自治县彩虹印刷有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　印　　张：15.5　　字　　数：320千字
版　　次：2025年5月第1版　　印　　次：2025年5月第1次印刷
定　　价：98.00元

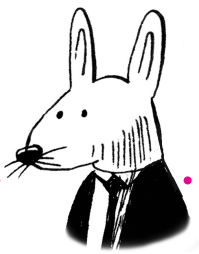
产品编号：101249-01

作者简介



基里尔•波波洛夫(Kirill Bobrov)是一名经验丰富且略带脾气的软件工程师。他擅长设计和开发高负载应用，对数据工程充满热情，目前专注于为全球公司实施前沿的数据工程方案。此外，基里尔也是热门插画技术博客(网址为https://luminousmen.com)背后的神秘高手。

致　谢



在深入探讨并发之前，我想先表达我的感激之情，没有这些杰出人士的支持，本书不可能出版。有人将写书比作跑马拉松，但我认为写书的体验更像一场狂野的过山车之旅，幸运的是，有这些伙伴的陪伴！

首先，我要向我的妻子叶卡捷琳娜•克里弗茨(Ekaterina Krivets)表达我最深的感激之情，她为本书贡献了所有令人惊叹的插图。

其次，我的团队也一直陪伴着我，并给予我支持。特别感谢基里尼娜•亚利雪娃(Kristina Ialysheva)、米哈伊尔•波尔托拉茨基(Mikhail Poltoratskii)、塔蒂娜•波罗丁娜(Tatiana Borodina)、安德烈•加维罗夫(Andrei Gavrilov)和亚历山大•贝尔尼茨基(Aleksandr Belnitskii)，他们始终在我身后，给予我信任和支持我。特别感谢维拉•克里弗茨(Vera Krivets)，她帮助我润色英文。

柏特•贝茨(Bert Bates)和布莱恩•哈纳菲(Brian Hanafee)的教导和理念深刻地影响了我对教学和讲解复杂概念的方式。感谢二位的宝贵建议和贡献。

我深深感谢Manning出版社的团队。迈克•斯蒂芬斯(Mike Stephens)引领了这场激动人心的冒险，我对他的感激无以言表，感谢他给予我此次机会。艾恩•豪(Ian Hough)耐心地逐章帮我修正书中的英文错误，感谢他帮助我编辑本书。阿瑟•朱巴列夫(Arthur Zubarev)，感谢他不厌其烦地审阅初稿中的错误并提供有价值的反馈。劳•科维(Lou Covey)，我想为我偶尔的失礼向他表示歉意，感谢他给予我的持续鼓励。马克•托马斯(Mark Thomas)，感谢他对本书进行技术审阅和代码检查。蒂凡尼•泰勒(Tiffany Taylor)的精细工作和专业知识显著提高了本书的清晰度和连贯性。凯蒂•坦纳(Katie Tennant)，没有他十分细致的审查和编辑，本书内容不可能得以完善并成功出版。

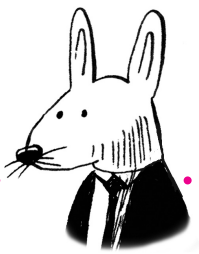
对于本书所有的审阅者，包括阿祖吉特•纳亚克(Abhijith Nayak)、阿姆拉•乌穆德鲁(Amrah Umudlu)、安德烈斯•萨科(Andres Sacco)、阿纳德•贝利(Arnaud Bailly)、巴

尔比尔•辛格(Balbir Singh)、比吉斯•科马兰(Bijith Komalan)、克里福德•萨伯(Clifford Thurber)、大卫•雅科博维奇(David Yakobovitch)、德米特里•沃罗比乌夫(Dmitry Vorobiov)、埃德度•梅伦德斯(Eddu Melendez)、埃尔内斯托•阿罗约(Ernesto Arroyo)、埃内斯托•波西(Ernesto Bossi)、艾山•坦德逊(Eshan Tandon)、埃兹拉•雪里德(Ezra Schroeder)、弗兰斯•欧林基(Frans Oilinki)、甘盛•斯瓦米纳汀(Ganesh Swaminathan)、格雷•古森斯(Glenn Goossens)、格雷戈里•瓦尔吉塞(Gregory Varghese)、伊玛克丽特•雷斯托•莫莎(Imaculate Resto Mosha)、詹姆斯•祖吉恩•刘(James Zhijun Liu)、吉里•捷西涅克拉(Jiří Činčura)、约翰顿•里维斯(Jonathan Reeves)、拉瓦尼亚•埃姆•克(Lavanya M K)、卢克•罗格(Luc Rogge)、马诺杰•雷德迪(Manoj Reddy)、马特•古柯斯基(Matt Gukowsky)、马特•惠尔克(Matt Welke)、米卡勒•达特雷(Mikael Dautrey)、诺兰•托(Nolan To)、奥利弗•科顿(Oliver Korten)、帕特里克•戈茨(Patrick Goetz)、帕特里克•雷金(Patrick Regan)、拉古纳斯•贾瓦哈尔(Ragunath Jawahar)、萨伊•赫格德(Sai Hegde)、赛尔吉奥•阿贝罗•罗德里格兹(Sergio Arbeo Rodríguez)、舍罗希卡•库拉蒂拉克(Shiroshica Kulatilake)、文卡塔•纳格恩达•巴布•亚纳马达拉(Venkata Nagendra Babu Yanamadala)、维塔利•拉尔什诺科夫(Vitaly Larchenkov)和威廉•詹米尔(William Jamir)，感谢他们为本书提供了宝贵的建议，使本书质量更上一层楼。

我还要特别感谢无私奉献的幕后英雄们。虽然他们的付出默默无闻，但其意义重大。他们是真正的明星！

最后，就像斯诺普•道格(Snoop Dogg)所言，我还要感谢我自己。如果没有我，就没有这本书。

前　言



畅想一个技术飞速发展的世界，其演进的速度远超狂奔的猎豹，人们对高效并发编程的需求达到了前所未有的程度。在这个世界中，软件工程师面临严峻的挑战，既要构建足以应对海量数据并能进行高速处理的系统，同时还要满足用户无尽的需求。这是一个并发既令人着迷又充满困惑的时代，而我们正生活在这个时代。

我曾经深受并发问题的困扰。后来，我偶然了解到“并发”和“异步”的概念，这无异于发现了一处隐藏的宝藏。如果能善加利用这份秘而不露的宝贵资源，就可以将普通代码变成算力惊人的程序。然而，这份宝藏非常复杂，涉及许多技术名词，比如并发、并行、线程、进程、多任务和协程等。为了揭开并发编程的神秘面纱，我渴望找到一位向导，帮助我将所有知识条理清晰地串联起来。但是，由于一直未能找到能够将不同编程语言理论与实践结合起来的教程，我决定亲自着手编写。《并发编程图解》一书就是这样诞生的。希望本书能够成为各位读者探索知识迷宫的指南针，解开谜题，照亮前行。

不同于普通技术图书，本书特意插入了许多读者感兴趣的故事和趣闻。相比于理论书籍，本书更像是一本风趣幽默的故事书，并配有多幅幽默的插图，读者不妨细数！在保持风趣幽默的同时，本书也不隐瞒对饺子和比萨的喜爱，学习并发编程本就是一件趣事！

本书将陪伴读者一同征服并发编程中的难题，解密异步编程的谜团。从并发基础知识到async和await的使用，本书将使用Python语言作为学习过程中的可靠伙伴。即使读者对Python不够熟悉也无需担心，本书涉及的概念和方法并不局限于具体实现。

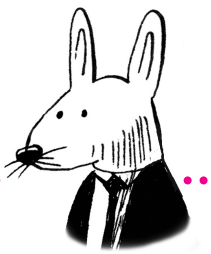
然而，读者可能会想：“为什么偏要选择Python呢？”这是因为Python在简单和强大之间实现了完美的平衡，能让开发者专注于并发的本质。此外，作为作者，我也毫不隐瞒对Python的喜爱。

无论你是希望加深对并发系统理解的资深开发者，还是对并发底层机制抱有好奇心的新人，本书都有适合你的内容。通过挖掘并发编程的秘密，读者将学习如何

构建可扩展、高效和有弹性的软件系统，以应对任何挑战。

亲爱的读者，准备开启一段独特的学习之旅吧！在这段旅程中，时空的界限将变得模糊，程序会以章鱼般的节奏“舞动”。是的，你没听错——章鱼。作为来自深海的可爱生物，其八条触须配合得天衣无缝，就像并发系统一样既复杂又迷人。我们的旅程马上开启！

关于本书



并发、异步和并行编程领域繁冗复杂，本书力求用清晰且幽默的方式讲解其中的基础知识和实践技巧。不同于学术研究论文和编程图书，本书重点介绍底层思想和原理，而不是具体的实现细节。本书使用了风趣易懂的语言，并利用图表而不是复杂的数学概念进行阐释。通过阅读本书，读者将了解并发编程框架，并能在实际开发场景中使用可扩展的解决方案。本书填补了市场空白，为想要掌握并发和异步的开发者提供了一条学习捷径，并提供了全面且易懂的指南。否则，开发者可能需要多年的开发经验才能真正理解并发编程的精髓。

目标读者

本书适合想要了解并发编程基础知识的读者。为了充分利用好本书，读者应具备计算机系统、编程语言和数据结构的基础知识，以及顺序编程方面的经验。读者不需要具备操作系统方面的知识，因为本书已提供所有必要的信息。虽然本书涉及网络概念，但不会对其进行详细讨论，因此假定读者具备网络基础知识。读者不需要对这些主题有深入了解，如果有需要，可在阅读本书的同时再进一步学习。

本书内容

本书分三篇。第一篇“章鱼交响乐团：并发交响曲”，介绍了基本概念和编写并发程序的方法。通过分层的方式，从硬件层到应用层，第1~5章介绍了并发的基础知识。

第二篇“并发的章鱼触手：多任务、分解、同步”，讨论了如何利用抽象和流行的设计模式提高代码性能、可扩展性和弹性。在第6~9章，读者将学习如何避免构建并发系统时最常见的问题。

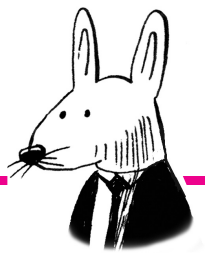
第三篇“异步章鱼：使用并发原理烹饪比萨”，基于前面章节介绍的并发知

识，将讨论如何从单台机器扩展到多台联网机器。经过扩展，事件可以异步进行，即一个事件的发生时间不同于另一事件。第10~12章将重点介绍异步概念，展示并发的另一维度。以前，异步只是用来处理并发或并行任务，但现在可以将异步和真正的并发操作结合起来，以提高系统性能。第13章通过逐步拆解一系列并发问题为本书画上句号，助力读者完全掌握并发编程的核心技能。

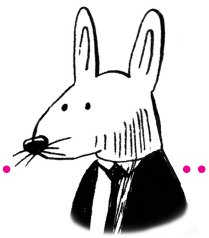
下载代码

本书中示例的完整源代码可以从GitHub(https://github.com/luminousmen/grokking_concurrency)下载，也可通过扫描本书封底的二维码下载。源代码的目的是作为程序实现的参考。为了便于学习，这些示例代码已经过一定优化，充当教学的工具，但不一定适用于生产环境。对于生产环境项目，建议使用成熟的库和框架，因为它们通常对性能进行了优化，经过了良好的测试且支持度更高。

目　录



第一篇
章鱼交响乐团：并发交响曲



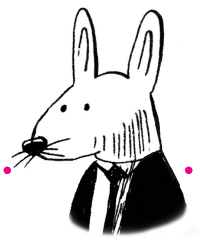
在安静的咖啡厅里，你正品尝着美味的咖啡，旁边突然传来一群程序员热烈的争论声。他们口中喊着并发计算、线程和进程间通信等专业词语，让人一头雾水。不只是你听不明白，其他人也都很困惑。

如果你曾去音乐厅听过现场音乐会，一定见过一群演奏家同时演奏不同乐器的场景，感受过音乐旋律的和谐动听。不同乐器的音色天衣无缝地融合在一起，演奏出令人惊叹的旋律。同时演奏乐器和并发很像。在并发中，多个进程或线程同时运行以实现共同的目标。

在本书的第1～5章，你将学习并发的基础知识，了解计算机的工作原理和不同类型的并发。我们将介绍顺序计算和并行计算，深入探讨硬件和软件实现并发的原理，还会分析各种进程间通信，这些通信方式使多个进程能够无缝协作。

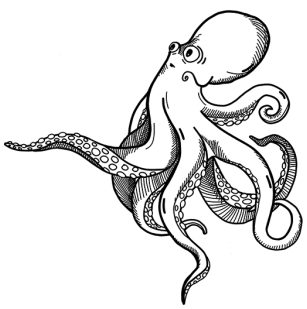
话不多说，端起一杯拿铁咖啡，加入程序员们的讨论吧。你一定会有所收获！

第1章 并发入门



本章内容：

- 为什么并发如此重要
- 如何衡量系统性能
- 并发的层级



利用片刻时间，仔细打量窗外的世界。世界万物是以线性、顺序的方式运动，还是以相互作用、彼此独立的复杂方式同时在运动呢？

尽管人们更善于顺序思考，比如按照待办事项列表逐一处理手头的工作——但现实是，世界的运行规律远比这复杂，其往往不按照顺序而是以并发的方式运行，相互关联的事件同时发生。从繁忙混乱的超市到配合天衣无缝的足球队，再到车水马龙的道路，并发无处不在。和自然界一样，计算机系统只有依靠并发，才能对复杂多变的现实世界进行建模、模拟和分析。

计算机通过并发可使系统同时处理多个任务。这些任务可能是一段程序、一台计算机或多台计算机构成的网络。如果不使用并发，应用则无法处理世界的复杂性。

当我们深入研究并发时，可能会面临诸多问题。首先要回答的问题是，为什么并发如此重要？

1.1 为什么并发如此重要

并发在软件工程中至关重要。高性能应用和对并发系统的需求，使并发编程成为软件工程师的必备技能。

并发编程并不是新概念，但在最近几年受到了越来越多的关注。随着现代计算机系统中内核和处理器数量的不断增加，掌握并发编程已成为编写软件的必要技能。公司正在寻找熟练掌握并发编程的开发者，因为并发通常是解决计算资源有限且性能需求高问题的唯一方法。

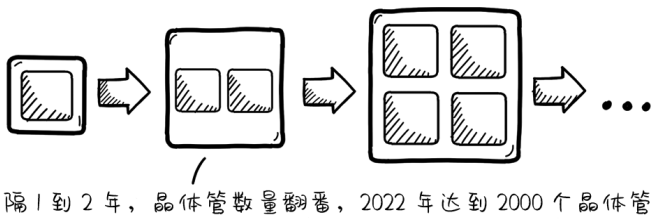
并发的最主要优势是能够提高系统性能，这也是人们研究并发的初衷。下面让我们一探究竟。

1.1.1 提升系统性能

当需要提高性能时，仅购买更高性能的计算机，为什么行不通呢？几十年前，人们确实是这么做的，但是最终发现这种做法并不可行。

摩尔定律

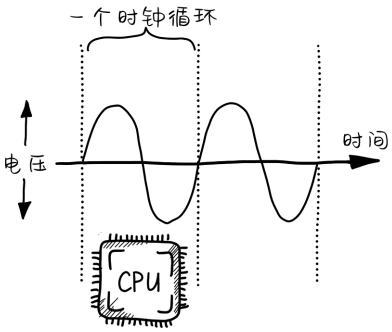
1965年，英特尔联合创始人戈登•摩尔(Gordon Moore)发现了一个规律。每隔两年左右就会出现新一代处理器，处理器中的晶体管数量大致会翻倍。摩尔得出结论，晶体管的数量及处理器时钟速度每隔24个月都会翻倍。这就是著名的摩尔定律。对于软件工程师而言，这意味着只需要等待两年，程序的运行速度就会翻倍。



然而，大约在2002年，情况发生了变化。正如著名的C++专家Herb Sutter所说，“免费午餐结束了”[1]。人们发现了处理器的物理尺寸和处理速度(处理器频率)之间的基本关系。运算执行时间取决于电路长度和光速。简而言之，只能在空间有限的情况下添加有限数量的晶体管，晶体管是计算机电路中的基本构建模块。由于温度的上升也起着重要作用，单纯增加处理器频率无法进一步提高性能。这导致了所谓的“多核危机”(multicore crisis)。

1 Herb Sutter 博客文章“免费午餐结束了”，网址是 http://www.gotw.ca/publications/concurrency-ddj.htm。

受限于物理因素，虽然单个处理器的时钟速度停止进展，但是对系统性能的要求并没有停止。芯片制造商转而以多核处理器的形式进行水平扩展。这迫使软件工程师、架构师和开发者适应具有多核处理器的架构。

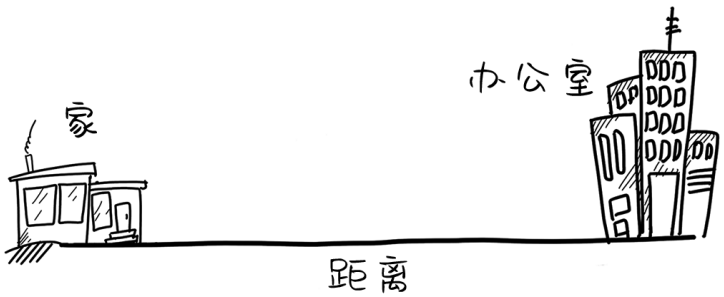


回顾这段历史，我们得出一个重要结论，并发的关键优势在于提高系统性能，这也是人们研究并发的初衷。其次，并发可以高效利用额外的计算资源。这引出了两个重要问题，如何评估性能，以及如何提高性能？

延迟与吞吐量

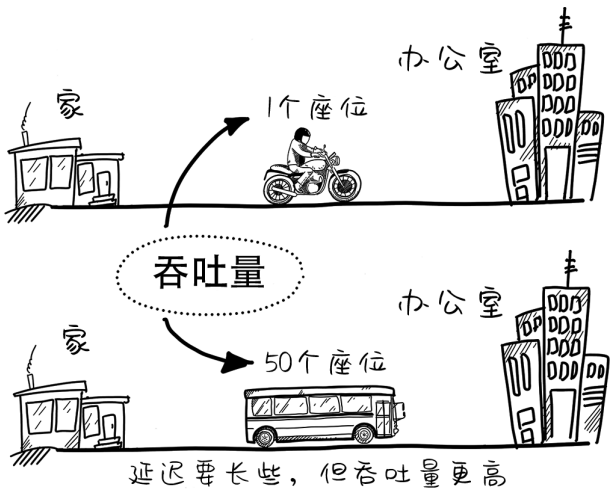
在计算机领域，有多种方式可以对性能进行量化，这取决于我们如何衡量计算机系统。方法之一是减少单个任务的执行时间，以提高任务完成量。

假设你平时骑摩托车往返家和办公室，单程需要一小时。你很在乎如何尽快到达办公室，因此使用该指标衡量系统性能。如果骑得快，则能够更快到达办公室。从计算系统的角度来看，这个时间称为延迟(latency)。延迟衡量的是单个任务从开始到结束所需的时间。



接下来，假设你在交通部门工作，职责是提高公交系统的运力。你不仅要关心如何让一个人更快地到达办公室，而且还要考虑增加单位时间内从家到办公地点的人数。这称为吞吐量(throughput)，即系统在一段时间内可以处理的任务数量。

理解延迟和吞吐量之间的区别非常重要。即使摩托车的速度比公交车快两倍，公交车的吞吐量也比摩托车高25倍(摩托车每小时将1个人运送到某个位置，而公交车每2小时将50个人运送到同样位置。按时间平均，即每小时运送25人！)。换句话说，更高的系统吞吐量不一定意味着较低的延迟。在优化性能时，对某一因素(如吞吐量)的改进可能会导致另一因素(如延迟)的恶化。



并发能够降低延迟。例如，可以将一个长期运行的任务分解为多个并行执行的小任务，从而降低整体执行时间。并发还可以通过同时处理多个任务来增加吞吐量。

此外，并发还能够隐藏延迟。当我们在等待电话、搭乘地铁上班或进行其他活动时，除了等待，也可以利用这段时间处理其他事情。例如，乘坐地铁时可以阅读电子邮件，这样就能同时完成多项任务，通过高效利用等待时间来隐藏延迟。隐藏延迟对于响应式系统至关重要，也适用于其他涉及等待的问题。

因此，并发可以通过以下三种主要方式提高系统性能：

- 降低延迟(即让一项工作更快完成)。
- 隐藏延迟(即让系统在高延迟操作期间完成其他任务)。
- 增加吞吐量(即让系统完成更多工作)。

了解了并发提高系统性能的途径后，我们再介绍并发的另一项应用。本章开头试图对周围复杂的世界进行建模，发现世界是并发运行的。接下来，我们将更具体地介绍并发如何解决大型或复杂的计算问题。

1.1.2 处理复杂和大型任务

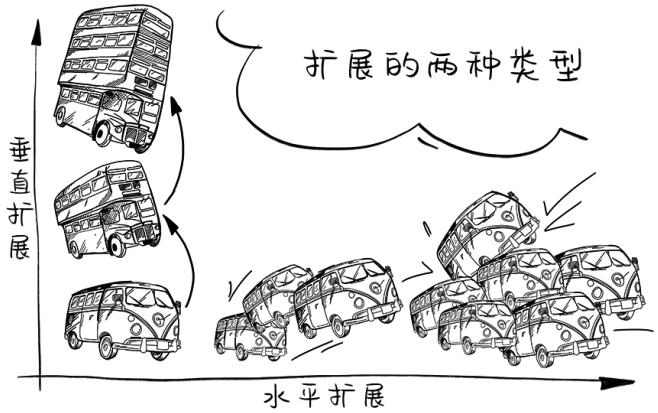
在开发与现实世界交互的系统时，软件工程师需要解决许多非常复杂的问题，而使用顺序系统来解决这些问题是不切实际的。这些复杂性可能来自问题的规模或系统中的某个高难度的组件。

可扩展性

问题的规模涉及系统的可扩展性，即系统通过增加资源提高性能的特性。增加系统可扩展性的方式主要分为两种类型，分别是垂直和水平。

垂直扩展(也称为向上扩展)是通过增加内存量来升级现有处理资源，或使用更强大的处理器替换旧处理器，从而提高系统性能。在这种情况下，可扩展性是有限的，因为提升单个处理器的速度存在难度，因此很容易达到性能上限。此外，升级到更强大的处理资源成本昂贵(例如，购买超级计算机)，用户要为顶级云实例或硬件支付更高的价格，但获得的性能提升却不总是与成本成正比。

减少特定任务的处理时间能提升一定性能，但最终还是需要对系统进行扩展。水平扩展(也称为横向扩展)通过在现有和新的处理资源之间分配负载来提高程序或系统性能。只要可以增加处理资源，我们就能持续提高系统性能。在这种情况下，可扩展性的瓶颈不会像垂直扩展那样立即出现。



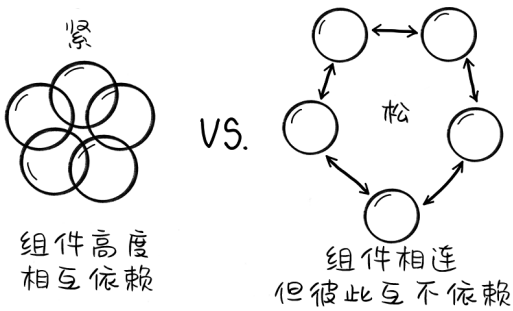
因此，软件行业通常倾向于采取水平扩展的方式。实时系统、海量数据、提高冗余度以提升可靠性、迁移到云端/SaaS环境以提升资源利率等诸如此类的需求，促使软件行业不得不进行水平扩展。

水平扩展涉及系统并发，单台计算机很难满足这些需求。而多台相互连接的机器，即计算集群(computing cluster)，可以在合理时间内完成数据处理任务。

解耦

大型问题的另一特点是其复杂性。然而，如果开发者不持续进行改进，则系统的复杂性不会降低。公司希望产品更强大且功能更完善，但这不可避免地会增加代码库、基础设施和运维工作的复杂性。开发者必须设计和实施不同的架构方法，以简化系统并将其分解为更简单的独立通信单元。

软件工程中的职责分离非常重要。作为基本的工程原则，分治(divide and conquer)可以创建松散耦合的系统。理论上，将相关代码(紧密耦合组件)分组，并分离不相关代码(松散耦合组件)，可以使应用程序更易于理解和测试，并减少错误数量。



从另一个角度看待并发，其本质上是一种解耦策略。对并发模块或单元进行功能分解，有助于单个组件专注于特定功能，更易于维护，同时降低整体系统的复杂性。软件工程师通过选择恰当的方式进行解耦，极大地提高了应用程序的性能、可扩展性、可靠性和内部结构。

并发在现代计算系统、操作系统(OS)和大型分布式集群中非常重要且应用广泛。并发有助于模拟真实世界，还能大幅提升系统的用户体验和开发效率，使开发者能够解决大型且复杂的问题。

探索并发编程将改变你对计算系统组成及运行的思考方式。本书将通过抽丝剥茧的方式讲解并发的各个层级。

1.2 并发的层级

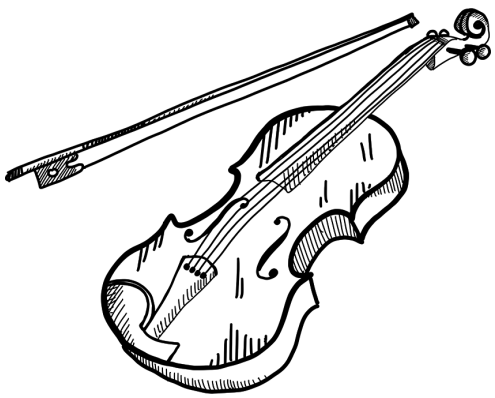
和大多数复杂的设计问题一样，并发包含多个层级。需要特别注意的是，在分层架构中，彼此对立或互斥的概念可能共存于不同的层级。例如，在顺序型机器上可以执行并发运算。

我倾向于将并发的层级架构想象成一个交响乐团，演奏曲目可能出自柴可夫斯基或其他音乐家。

- 顶层是概念层或设计层(应用层)。可以将其视为交响乐团中作曲家的作品；在计算机系统中，算法如同音乐符号，指导系统组件应执行的操作。

- 其次，是运行时中的多个任务(运行时系统层)。这如同音乐家们用不同的乐器合作演奏作品的不同部分。在指挥家的指挥下，音乐演奏从一个小组转移到另一个小组。类似于计算机系统中，不同进程各司其职，以实现整体目标。

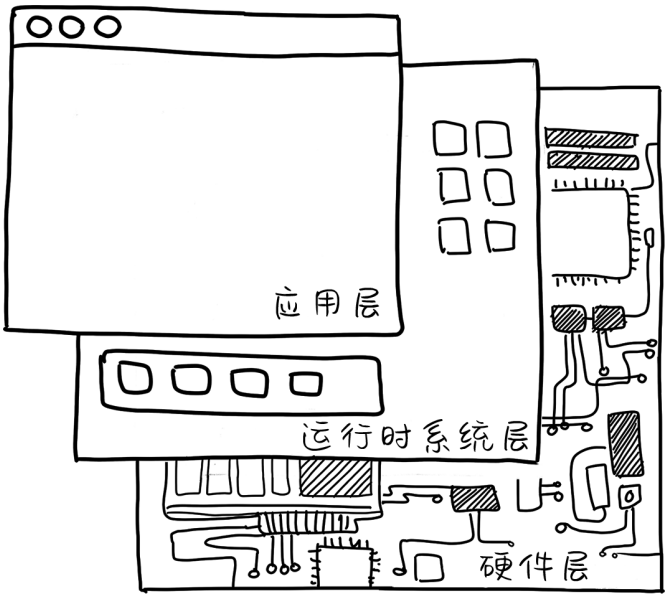
- 最后，是底层执行(硬件层)。我们仔细观察特定的乐器，如小提琴。每名小提琴手演奏的音符由一到四根在特定频率振荡的弦产生，其频率由弦的长度、直径、张力和密度决定。在计算机系统中，单个进程根据其专属指令执行任务。



每个层级描述的都是同一过程，但细节不同，有时还存在矛盾。

同样，在并发中也是如此。

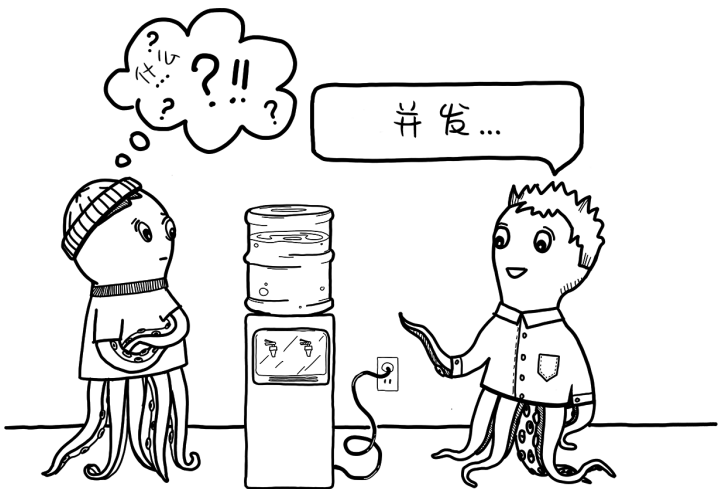
- 在硬件层，我们直接面对机器指令，这些指令由处理资源执行，并使用信号访问硬件外设。由于现代架构的复杂性不断提升，优化这些架构上的应用程序性能需要深入了解应用程序与硬件组件之间的交互。
- 在运行时系统层，系统调用、设备驱动程序和调度算法掩盖了与编程抽象相关的诸多不足，这些过程显著影响并发系统，因此需要深入了解。这一层通常由操作系统负责，将在第3章对其进行介绍。
- 在应用层，对更符合物理世界的运行方式进行抽象。软件工程师通过编写源代码，可以实现复杂的算法和业务逻辑。代码还可以利用编程语言特性修改执行流程，并表示通常只有软件工程师才能想到的抽象概念。



请牢记这些广泛使用的层级，在学习并发编程的过程中，我们会不断提到它们。

1.3 本书内容

众所周知，掌握并发编程是一项挑战。并发的复杂性源于即使是有经验的开发者往往也难以用文字清晰地阐释其原理，口头表达也同样难。本书将通过简单易懂的方式详细讲解并发的相关知识。



本书不涉及并发编程的所有细节，而是致力于帮助读者入门并掌握核心学习内容。本书探讨了并发编程中涉及的问题，并介绍创建并发和可扩展应用的最佳途径。

初级和中级程序员将通过本书掌握编写并发系统的基本知识。为了充分利用本书，读者应该具备一定的编程经验，但不需要是专家。本书通过具体示例解释了关键概念，并使用Python编程语言演示操作。

本书分为三篇，分别涵盖了不同水平的并发。第一篇讨论并发编程的基本概念和术语，覆盖从硬件层到应用层的知识。

第二篇专注于设计并发程序和流行的并发模式。本篇还介绍了如何在构建并发系统时避免常见的并发问题。

第三篇将从单台机器扩展到通过网络连接的多台机器，并在集群上运行应用程序。本篇探讨了任务之间异步通信的重要性。此外，还分步介绍了如何编写并发应用程序。

完成本书的学习后，读者将掌握有关并发编程和最新的异步、并发编程方法。本书通过由浅入深的教学方式，从底层硬件操作开始学习，直到掌握更高层次的应用程序设计，并将理论转化为实践。

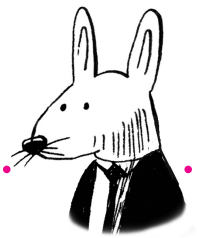
本书中的所有代码都是使用Python 3.9编程语言编写的，并已在macOS和Linux操作系统上进行了测试。本书的讲解不依赖任何特定的编程语言，只是引用了Linux内核子系统。所有示例的源代码都可以通过GitHub仓库(https://github.com/luminousmen/grokking_concurrency)下载，也可以通过扫描本书封底的二维码下载。

1.4 本章小结

- 并发系统能够同时处理多个任务。
- 在现实世界中，任何给定时间都会同时发生许多事件。模拟现实世界的过程需要借助并发编程。
- 通过降低或隐藏延迟，并发极大地提高了系统的吞吐量和性能，同时更高效地利用现有资源。

- 本书通篇使用了可扩展性和解耦这两个概念。
  - 可扩展性可以是垂直的或水平的。垂直扩展通过升级现有处理能力来提高程序和系统的性能。水平扩展通过在现有和新的处理资源之间分配负载来提高性能。行业内的转型趋势是采用水平扩展的方式进行架构升级，而并发编程是水平扩展的先决条件。
  - 复杂的问题可以分解为简单的组件，并将其相互链接。在某种程度上，并发也是一种解耦策略，可以协助解决大型和复杂的问题。
- 在探索未知世界时，地图是不可或缺的工具。在本书中，我们使用并发层级作为导航，即应用层、运行时系统层和硬件层。

第2章 串行执行和并行执行



本章内容：

- 程序在运行过程中涉及的术语
- 在并发的最底层，不同物理任务的执行方法
- 创建第一个并行程序
- 并行计算方法的极限

数千年来[1](虽然并没有这么久，但确实已经有很长时间了)，开发者一直在使用最简单的计算模型——顺序模型来编写程序。串行执行方法是顺序编程的核心，也是我们介绍并发编程的起点。本章将介绍底层执行层中的不同执行方法。

1 译者注，现代意义的编程开发是从 20 世纪开始的，其实只有数年的时间，这是作者的幽默写法。

2.1 程序概念

并发编程面临的首要问题，实际上也是计算机科学领域中的常见问题，即人们命名事物的能力非常有限。我们有时会使用相同的词汇描述多个不同的概念，或者使用不同的词汇描述同一事物，甚至在不同的语境下使用不同的词汇描述不同的事物。有时，人们甚至会创造新词汇。

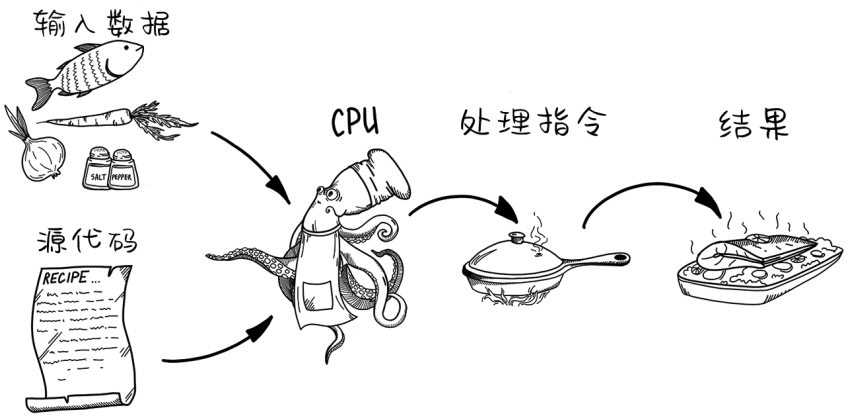
注意

CAPTCHA是一个缩写词，其全称是“用于区分计算机和人的完全自动公共图灵测试”(Completely Automated Public Turing test to tell Computers and Humans Apart)。

因此，在研究计算的执行之前，了解被执行的内容以及本书中使用的术语将会非常有用。通常而言，程序是一系列计算机系统执行或运行的指令。

程序必须先编写才能执行。程序是通过使用任一编程语言编写源代码来完成的。源代码就像是烹饪书中的菜谱，它包含一系列步骤，指导厨师利用原材料制作出一顿美味的菜肴。烹饪涉及多个要素，包括菜谱、厨师以及原材料。

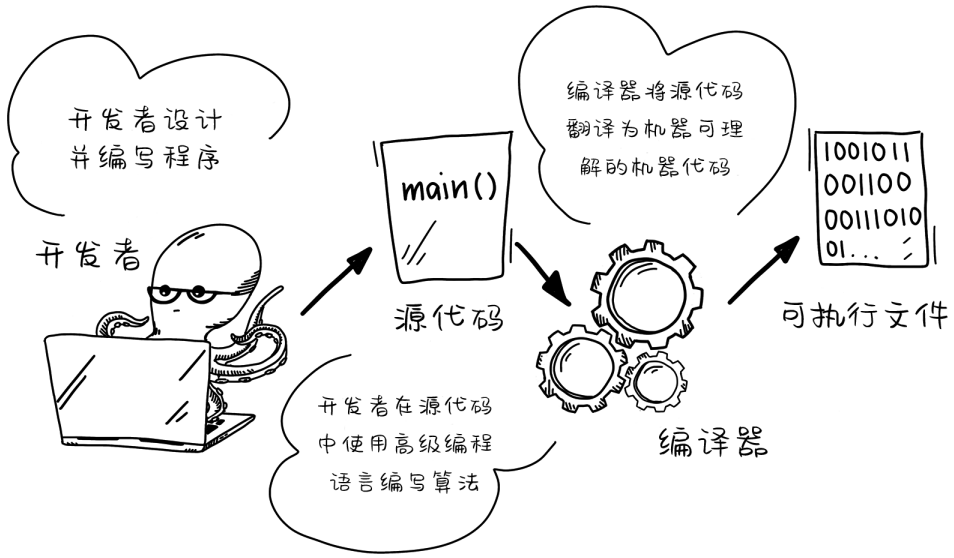
执行程序与遵循菜谱类似。我们有程序的源代码(菜谱)、厨师(处理器，即CPU)和原材料(程序的输入数据)。



处理器本身无法单独完成任何一项有意义的任务，它无法对事物进行排序或搜索具有特定特征的对象，只能执行有限数量的简单任务。所有的“智慧”能力都是由程序决定的。无论处理能力有多强，如果不指明方向，开发者也无法完成任何任务。将一项任务转换为处理器可执行的步骤是开发者的工作，就像烹饪书的作者编写菜谱。

开发者通常使用编程语言描述想要完成的任务。然而，CPU无法直接理解用常规编程语言编写的源代码。首先，必须将源代码翻译成机器代码，这是CPU能够理

解的语言。翻译是由专门的程序(编译器)完成的。编译器生成的文件，通常称为可执行文件(executable)，其中包含CPU可以理解和执行的机器级指令。



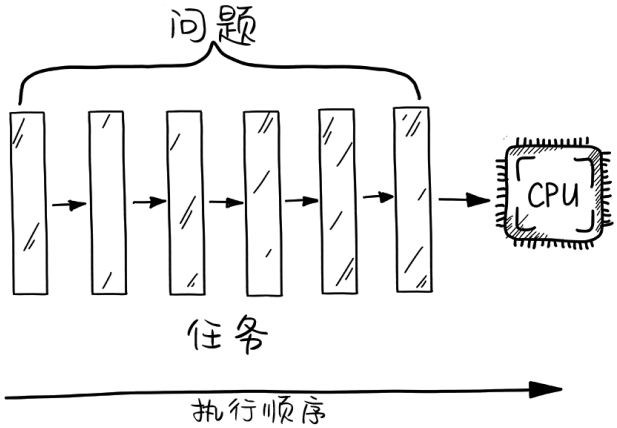
当CPU执行机器代码时，它可以采用几种不同的方法。处理多条指令的最基本方法是串行执行(serial execution)，这是顺序计算的核心。下一节将研究这个问题。

2.2 串行执行

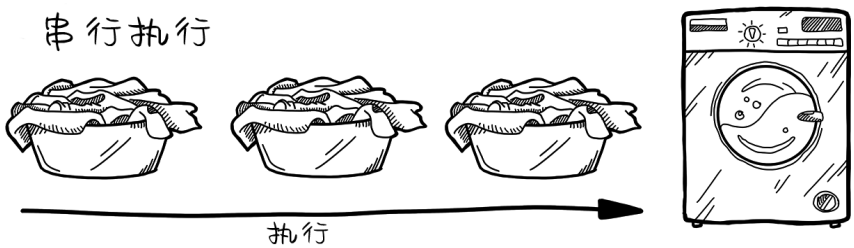
如前所述，程序是一系列指令的列表，通常这些指令的列表顺序很重要。回到菜谱的例子，假设你按照自己最喜欢的菜谱开始烹饪，但却以错误的顺序执行菜谱的步骤。例如，在将鸡蛋与面粉混合之前煎了鸡蛋，这样的结果不会让人满意。对于许多任务而言，步骤的顺序至关重要。

编程也是如此。当解决编程问题时，我们首先将问题分解成一系列子任务，然后依次或串行地执行这些子任务。基于任务的编程使我们能够以与机器无关的方式讨论计算，并为构造模块化程序提供框架。

任务可以视为一项工作。如果是关于CPU执行，则我们可以将任务称为指令(instruction)。任务也可以是一系列操作的序列，它们共同构成了对现实世界模型的抽象，如写入文件、旋转图像或在屏幕上打印消息。任务可以包含单个操作或多个操作(我们将在后面的章节讨论更多关于这方面的内容)，但它是逻辑上独立的工作单元。我们使用任务(task)这一术语作为执行单元的通用抽象。



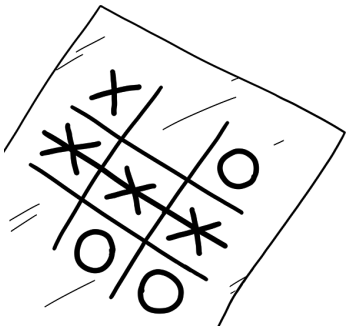
任务的串行执行类似于一个链条，第一个任务完成后紧接着是第二个任务，第二个任务完成后是第三个任务，以此类推，任务之间没有重叠的时间段。假设今天是洗衣日，你有一堆衣服要洗涤。不过，与许多家庭一样，你只有一台洗衣机，而且曾经将最喜欢的白色T恤和彩色T恤一起洗过。真是悲剧！



铭记教训后，你先把白色衣物放进洗衣机里洗涤，然后是黑色衣物，接着是被子，最后是毛巾。任何人完成洗衣的最短时间是由洗衣机的速度和洗衣量决定的。即使有很多衣服要洗涤，我们仍然必须按顺序依次进行。每次执行都会阻塞整个处理资源，如在白衣服洗涤中途就开始洗黑衣服，这种做法是不可取的。

2.3 顺序计算

要描述动态或与时间相关的现象，可以使用“顺序”这一术语。这是程序或系统的一种概念属性，更多体现的是程序或系统的设计及其在源代码中的编写方式，而不是实际执行过程。

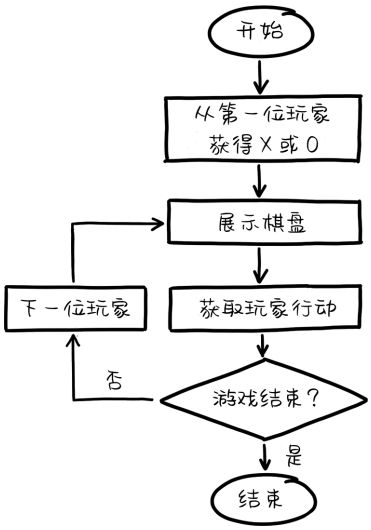


假设要实现一个三子棋游戏。游戏规则很简单，有两名玩家，其中一名玩家选择O标记，另一名玩家选择X标记。玩家轮流在棋盘上画出X或O。如果一名玩家在棋盘的横向、纵向或对角线上连续放置三个标记，则该名玩家获胜。如果棋盘已满且没有玩家获胜，则游戏以平局结束。

你能编写出这样的游戏吗？

讨论游戏的逻辑。玩家轮流通过输入行号和列号告诉程序他们想在哪里画标记。当一名玩家行动后，程序检查该名玩家是否取胜或者是否平局，然后切换到另一名玩家的回合。游戏按照这种方式持续进行，直到有一名玩家获胜或者平局。如果玩家获胜，程序会显示获胜玩家的信息，然后用户按任意键退出程序。

下图展示了三子棋游戏的逻辑示意图。



程序使用顺序步骤处理此问题。每个步骤都依赖于前一步骤的结果。因此，每个步骤都会阻塞后续步骤的执行。我们只能使用顺序编程方法来实现此类程序。

如示意图所示，程序的计算模型是由游戏规则即算法决定的。任务之间存在清晰的依赖关系，无法以任何方式进行分解。我们无法预测玩家尚未做出的行动，也不能让玩家连续行动两次，这违反游戏规则，属于作弊。

注意

实际上，依赖于前一步完成的任务并不多见。因此，对于日常工作中面临的绝大多数编程问题，开发者使用并发编程相对容易。我们将在后续章节中讨论这个问题。

需要注意的是，后续步骤必须在前一步骤完成后才能进行。你能想到哪些任务需要串行执行吗?

与顺序编程相对的是并发编程。并发编程的思想是任务可以被拆分为独立的计算单元，并可以按任意串行执行，同时生成相同的结果。

顺序计算的优势与劣势

顺序计算既有优势也有劣势。

简单(优势)

任何程序都可以使用顺序计算这种范式来编写。这种范式逻辑清晰且预测性强，因此最为常见。当进行编程开发时，往往会首先考虑顺序范式。例如，先做饭，然后吃饭，最后洗碗是合理的任务序列；而先吃饭，然后洗碗，最后做饭则毫无逻辑。

顺序计算是一种简单明了的方法，具有明确且逐步的指示，指导开发者在何时执行特定的操作。顺序计算无需检查依赖步骤是否已完成，只有前一个操作执行完成后，下一个操作才会开始执行。

扩展性(劣势)

可扩展性是系统处理日益增长的任务量的能力，或者是提升系统处理任务能力的潜力。如果系统通过添加处理资源来提升性能，则该系统具有可扩展性。在顺序计算中，扩展系统的唯一方法是提升系统资源(如CPU、内存等)的性能。这种方式属于垂直扩展，受限于市场上可用的CPU性能。

开销(劣势)

在顺序计算中，程序执行的不同步骤之间无需进行通信或同步。但是，可用资源的低效利用会导致间接开销，即使程序层面没有问题，开发者也可能无法挖掘系统中所有可用资源，从而导致效率降低和不必要的成本。即使系统只有一个单核处理器，其利用率可能仍然较低。第6章将深入研究其原因。

2.4 并行执行

如果你熟悉园艺，则可能知道种植番茄通常需要四个月的时间。那么考虑一个问题，一年内是否只能种植三株番茄呢？

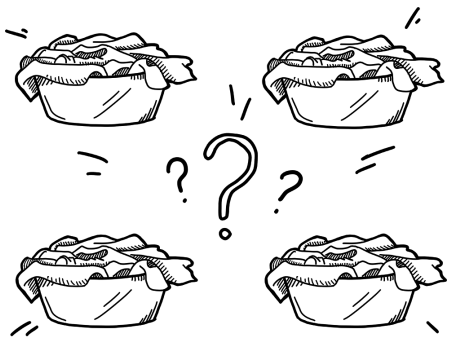
显然，答案是错的，因为可以同时种植多株番茄。

在串行执行中，只能在同一时刻执行一条指令。大多数人首先学习的是顺序编程，并且大多数程序使用顺序编程编写，即从主函数的main函数开始执行，按顺序依次执行任务/函数、调用/操作。

当否定同时只能做一件工作的假设时，我们便开启了并行计算的大门，就像同时种植多株番茄一样。不过，并行执行的程序通常更难编写。我们来看一个简单的类比。

如何快速洗衣服

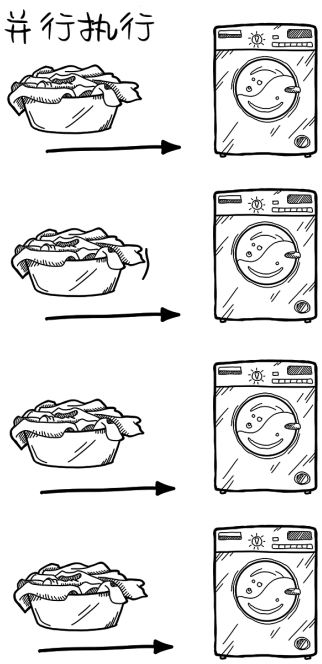
恭喜！你刚刚赢得了彩票大奖，奖品是免费的夏威夷机票。但在飞机起飞前，你面临一个挑战，必须在几个小时内洗好四篮衣服。然而，无论洗衣机效率多高，都无法同时洗多篮衣服，并且你也不想将衣服混合起来洗。



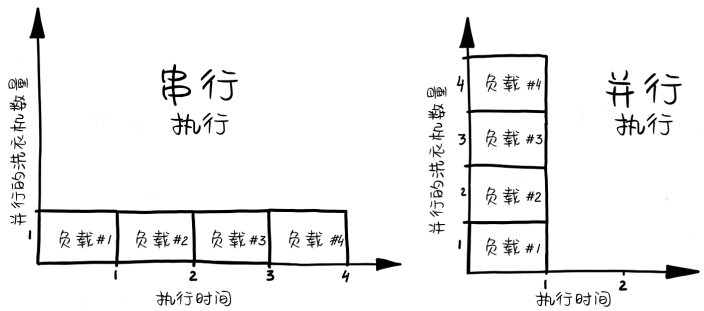
编程和洗衣服相似，顺序程序运行所需的时间受限于处理器的速度和处理器执行指令序列的速度。由于每篮衣服独立于其他衣服，因此如果拥有多台洗衣机，就可以通过并行处理更快地完成任务。

因此，你决定去最近的洗衣店。洗衣店里有多台洗衣机，因此可以同时使用四台独立的洗衣机分别洗涤四篮衣服。这种情况下，可以说所有洗衣机都是在并行工作，即同一时间清洗多篮衣物。因此，吞吐量提高为原先的四倍。

还记得在第1章中讨论的水平扩展吗？这里使用的就是这种方法。



并行执行意味着任务执行是在物理上同时进行。并行执行是相对于串行执行而言的。可以通过并行执行支持的任务数量来衡量并行性。因为使用了四台洗衣机，所以并行性等于4。



现在，我们了解了什么是并行执行，接下来讨论实现并行执行需要什么条件。

2.5 并行计算的要求

在深入研究并行执行前，我们首先讨论实现并行所需的条件，即任务独立和硬件支持。

2.5.1 任务独立

在顺序计算中，所有操作都是通过增加CPU时钟速度来进行加速的。这是降低延迟最简单的解决方案。这种方案不需要任何特殊的程序设计，所需要的只是更强大的处理器。并行计算则主要通过将问题分解为可并发且彼此独立执行的任务，以降低延迟。

注意

大型程序通常由众多小程序组成。例如，网络服务器处理来自网络浏览器的请求，并用HTML页面进行响应。服务器像小程序一样处理每个请求，最理想的情况是服务器支持同时处理多个网络请求。

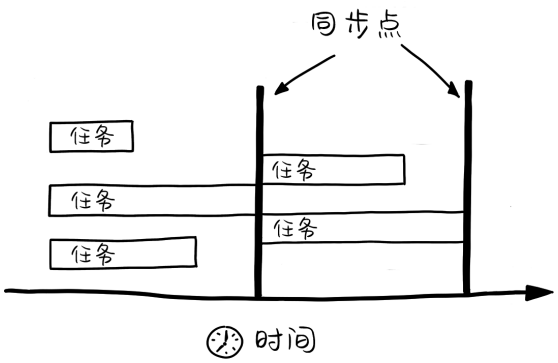
运用并行计算时，需要对具体问题进行具体分析。如果要将并行计算应用于某个任务，该任务必须可以分解为一组独立任务，以便每个处理资源可以同时处理算法的不同部分。这里的独立是指只要结果相同，处理资源可以按任意顺序和任意位置处理任务。如果任务不符合该分解要求，则无法并行处理任务。

判断程序是否可以并行执行的关键在于分析哪些任务可以分解，以及哪些任务可以独立执行。第7章将详细讨论如何进行任务分解。

注意

并行执行和串行执行之间的逻辑关系是单向的，也就是说支持并行执行的程序总能转换为顺序程序，但顺序程序不一定能转换为并行程序。

任务并不总是独立的，因为并非每个程序或算法都可以从头到尾被分解为独立的任务。有些任务是独立的，有些则不是。如果某些任务依赖于先前执行的任务，则这些任务不被视为独立任务。开发者必须对依赖程序的不同片段进行同步，以获取正确的结果。同步意味着等待依赖项执行的任务会被阻塞。在三子棋示例中，玩家的落子就阻塞了程序的执行。通过同步协调相互依赖的并行计算，会严重限制程序的并行性。与简单的顺序程序相比，阻塞和同步对编写并行程序造成了严峻的挑战(第8章将进行详细讨论)。



但是，投入更多的精力来编写并行程序是完全值得的。如果程序无误，并行执行可以提高程序的整体吞吐量。将大型任务分解为小任务后，能更快地完成，或在给定的时间内完成更多任务。

那些需要很少的同步或不需要同步的任务有时称为“尴尬的并行任务”。这些任务可以很轻易地分解为独立任务，然后并行执行。这类任务通常属于科学计算。例如，可以将子集分配给多个处理资源，以分配寻找素数的任务。

注意

处理“尴尬的并行任务”并不让人难为情！相反，因为编程相对简单，所以处理此类并行程序相对放松。近年来，“尴尬的并行”一词被赋予了其他含义。尴尬的并行算法往往具有少量的进程间通信，这是获得良好性能的关键，因此“尴尬的并行”通常是指具有低通信需求的算法。第5章会对此进行简要介绍。

因此，并行程度更多取决于问题本身，而不是试图解决问题的开发者。

2.5.2 硬件支持

并行计算需要硬件支持，并且需要硬件具备多个处理资源。如果处理资源少于

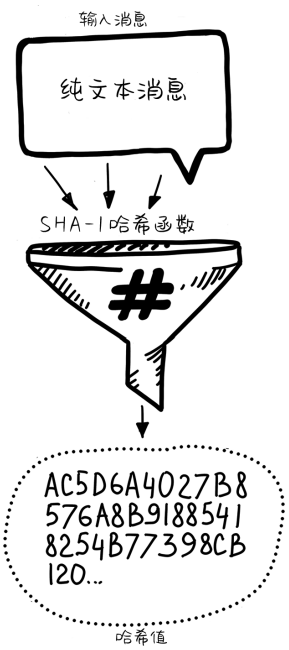
两个，则无法实现真正的并行。下一章将讨论硬件以及硬件如何支持多个并发操作。满足所有并行计算的条件后，接下来我们将探讨并行计算的实质。

2.6 并行计算

并行计算利用分解技术将大型或复杂问题分解成小任务，然后利用运行时系统的并行执行高效地处理这些小任务。以下示例将演示并行化的强大功能。

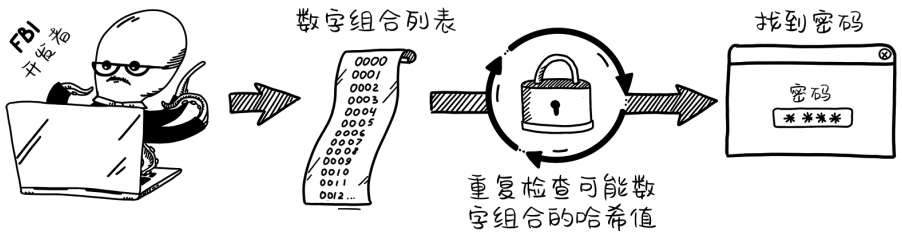
假设你在FBI的IT部门工作，在下一个任务中，你必须编写程序以破解密码(特定长度的数字组合)，并访问一个可以摧毁整个世界的系统。

寻找正确密码的常用方法是反复猜测密码(也称为暴力破解法)，计算其混淆形式(加密哈希)，并将得到的加密哈希与系统上存储的哈希值进行比较。假设你已经有了密码的加密哈希。



接下来，该如何实现程序呢？

暴力破解被视为一种解决密码问题的通用方法，需要列出所有可能的组合，进行遍历，并检查每个特定的解是否能解决问题。因此，你需要遍历所有可能的数字组合列表，并检查每个加密哈希是否与系统哈希值相匹配。



经过几个昼夜的努力，你终于想出了如何检查加密哈希且遍历所有可能的数字组合，并使用自己喜欢的编程语言编写程序。该算法会生成数字组合，并检查加密哈希。如果匹配，则打印出找到的密码，程序随之结束；如果不匹配，则转到下一个组合并重复这个循环。

最简单的方法是使用顺序计算一次性处理所有可能的密码组合。CPU每次处理一个任务，然后再获取下一个任务并按顺序执行，直到完成所有任务。使用串行执行解决问题的步骤如前一幅图所示，代码如下所示：

```
# Chapter 2/password_cracking_sequential.py
import time
import math
import hashlib
import typing as T

def get_combinations(*, length: int, min_number: int = 0) -> T.List[str]:
    combinations = []
    max_number = int(math.pow(10, length) - 1)
    for i in range(min_number, max_number + 1):
        str_num = str(i)
        zeros = "0" * (length - len(str_num))
        combinations.append("".join((zeros, str_num)))
    return combinations


def get_crypto_hash(password: str) -> str:
    return hashlib.sha256(password.encode()).hexdigest()


def check_password(expected_crypto_hash: str,
                   possible_password: str) -> bool:
    actual_crypto_hash = get_crypto_hash(possible_password)
    return expected_crypto_hash == actual_crypto_hash
```

在给定范围内，生成固定位数的密码组合列表

比较可能的密码的加密哈希和存储于系统的密码的加密哈希

```
def crack_password(crypto_hash: str, length: int) -> None:
    print("Processing number combinations sequentially")
    start_time = time.perf_counter()
    combinations = get_combinations(length=length)
    for combination in combinations:
        if check_password(crypto_hash, combination):
            print(f"PASSWORD CRACKED: {combination}")
            break
```

依次生成并测试所有固定位数的可能密码，一旦匹配成功，则立即停止

```
    process_time = time.perf_counter() - start_time
    print(f"PROCESS TIME: {process_time}")

if __name__ == "__main__":
    crypto_hash = \
      "e24df920078c3dd4e7e8d2442f00e5c9ab2a231bb3918d65cc50906e49ecaef4"
    length = 8
    crack_password(crypto_hash, length)
```

输出结果如下所示：

```
Processing number combinations sequentially
PASSWORD CRACKED: 87654321
PROCESS TIME: 64.60886170799999
```

问题迎刃而解，你充满自信地将程序交给执行任务的特工。008特工向你点头致意，随即饮尽了一杯伏特加马提尼。

我们都知道特工从不信任任何人。还不到一个小时，008特工便闯入你的办公室，告诉你程序运行太慢。根据他们的计算，在超级设备上处理所有可能的密码组合需要一小时！“我只有几分钟时间，这栋大楼就要片瓦无存了！”008特工害怕地说，又饮尽了一杯伏特加马提尼。他离开房间时扔下一道命令：“给程序提速！”

如何才能让程序跑得更快呢？

最明显的方法是提高CPU的性能。通过提高超级设备的时钟频率，可以在相同时间内处理更多的密码。不过，这种方法存在局限性，因为CPU速度有物理上限。而且，FBI拥有的处理器已经是最快的了，很难再提高CPU性能。这是顺序计算的最大缺点，即使计算机系统拥有多个处理资源，也很难进行扩展。

另一种加快程序执行速度的方法是将程序分解为独立的部分，并将子任务分配给多个处理资源以便同时处理。拥有的处理资源越多，任务越小，处理速度就越快。这是并行计算的核心思想，我们将在第8章和第12章深入研究。

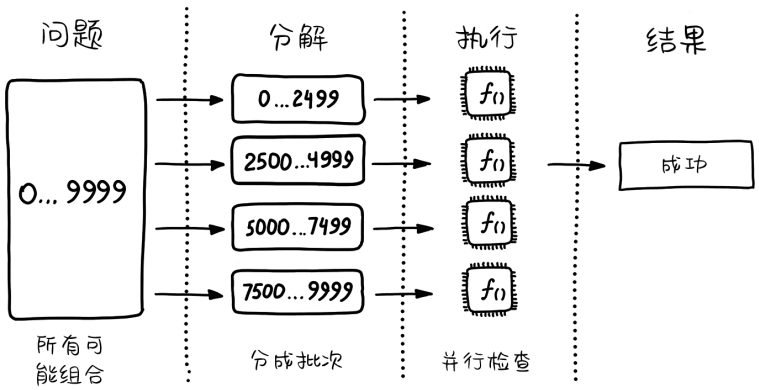
那么，使用并行执行可以解决密码问题吗？超级设备使用了一颗高级CPU，具有许多内核。因此，已经满足了硬件条件，可以并行执行任务。

然后，是否可以将问题分解为独立的子任务呢？将检查单个密码组合作为不依赖于其他任务的子任务，检查某个特定密码前不需要检查之前所有的密码。处理密码的顺序不重要，关键在于找到正确的密码，因此可以完全独立执行，不存在依赖关系。

由于存在硬件支持和子任务独立性，所以已经满足了并行计算的所有要求。接下来，我们着手设计最终解决方案。

对于此类问题的第一步是将问题分解为单独的任务。正如我们所观察到的，检查单个密码可以作为独立的子任务，这些子任务可以并行执行。因为彼此不存在依赖关系，所以无需同步。因此，这是一个相对简单的并行问题。

如下图所示，解决方案被分成了若干步骤。第一步是为每个处理资源创建要检查的密码范围(数据块)。然后，将数据块分配给可用的处理资源。从而为每个处理资源分配了一组密码范围。下一步是实际执行。



代码如下所示：

```
ChunkRange = T.Tuple[int, int]

def get_chunks(num_ranges: int,
               length: int) -> T.Iterator[ChunkRange]:
    max_number = int(math.pow(10, length) - 1)
    chunk_starts = [int(max_number / num_ranges * i)
                    for i in range(num_ranges)]
    chunk_ends = [start_point - 1
                  for start_point in
                  chunk_starts[1:]] + [max_number]
    return zip(chunk_starts, chunk_ends)


def crack_password_parallel(crypto_hash: str, length: int) -> None:
    num_cores = cpu_count()
    chunks = get_chunks(num_cores, length)

    # 并行运行
    # for chunk_start, chunk_end in chunks:
    #     crack_chunk(crypto_hash, length, chunk_start, chunk_end)}
```

将大型整数范围分解为小数据块，每个数据块包含大致相同数量的密码，以使用多核处理器并行处理

获取处理器数量

使用独立进程并发处理各分块的伪代码

我们添加了一个新函数，即crack_password_parallel，它在多个内核上并行执行crack_password函数。尽管在不同编程语言中，crack_password函数的形式可能不同，但其作用是一样的，即创建一组并行单元，将密码范围分配给并行单元以实现并行执行。这通常需要使用伪代码(一种用模拟实际代码的格式化语言编写、人类可读的程序逻辑表示)，我们将在第4章和第5章中进一步讨论具体代码。

注意

即使使用伪代码，所提供的示例也极具实战价值。例如，MATLAB语言具有parfor结构体，使得并行执行for循环变得非常简单。Python有joblib包，使用Parallel类可以轻松实现并行。R语言具有相同功能的Parallel库。标准Scala库具有并行集合，可简化并行编程，用户不必了解底层并行细节。

由于使用了并行计算，008特工再次在紧要关头拯救了世界。然而，我们大多数人没有FBI的资源。并行执行存在限制和成本，我们将其应用到问题处理之前需要认真考虑这些因素。下一节将讨论这些问题。

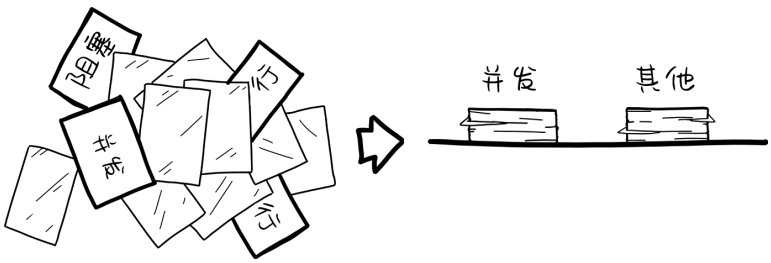
2.7 阿姆达尔定律

一位母亲需要九个月才能生下孩子。然而，即使九位母亲通力合作，可能在一个月内生下孩子吗？

人们可能会认为，通过无限增加处理器的数量，可以加速系统的运行。但遗憾的是，这种做法并不可行。Gene Amdahl发现的阿姆达尔定律清晰地展示了这一点。

到目前为止，我们分析了并行算法的执行。尽管并行算法可能包含串行执行的部分，但是大体可分为完全并行的部分和完全顺序的多个部分。顺序部分可能只是尚未并行化的步骤，或者如之前看到的那样不支持并行。

假设你有一大堆上面写着定义的索引卡片。你想找到有关并发的卡片，并将它们放到单独的堆中。但是，卡片杂乱无章地堆放在一起。幸好，你身边有两位朋友，可以将卡片分成两堆，每个人负责一堆，并告诉他们要寻找的目标。然后每个人可以在自己的卡堆中进行翻找。一旦有人找到了并发卡片，就可以宣布其发现，并将其放入一个单独的堆中。

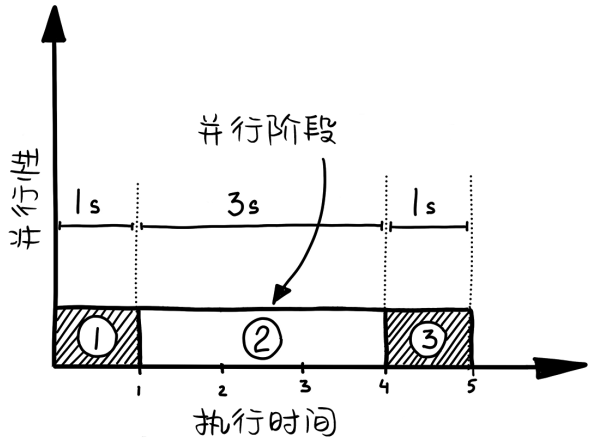


算法可以分为如下三个步骤。

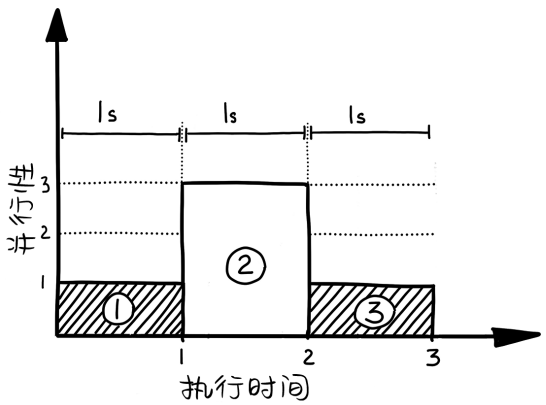
(1) 将大堆分成小堆，每个人负责一堆(串行)。

(2) 每个人在各自的堆中翻找“并发”卡片(并行)。

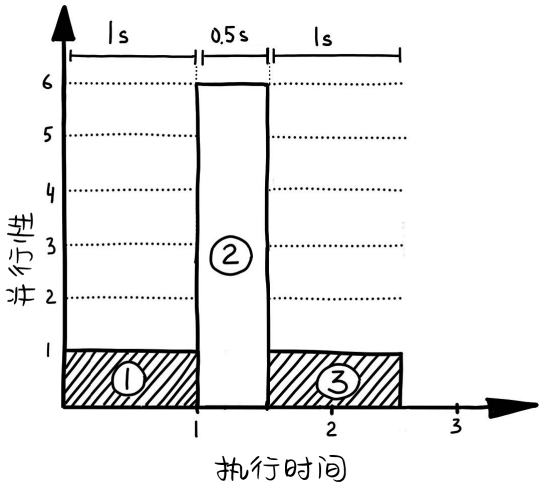
(3) 将并发卡片放到单独的堆中(串行)。



该算法的第(1)和第(3)步各需要1s，第(2)步需要3s。因此，如果由你自己单独完成，执行整个算法需要5s。第(1)和第(3)步在算法上属于串行，无法将其分解成独立的任务并使用并行执行。但是可以很容易地在第(2)步中使用并行执行，方法是将卡片分成任意数量的小堆，并让足够多的朋友分别执行这一步。通过借助两位朋友的帮助，可将该部分的执行时间缩短至1s。如此，整个程序现在仅需3s，提速40%。加速比例是通过拥有一定处理资源的并行执行时间与拥有单一处理资源的最优串行执行时间的比值来计算的。

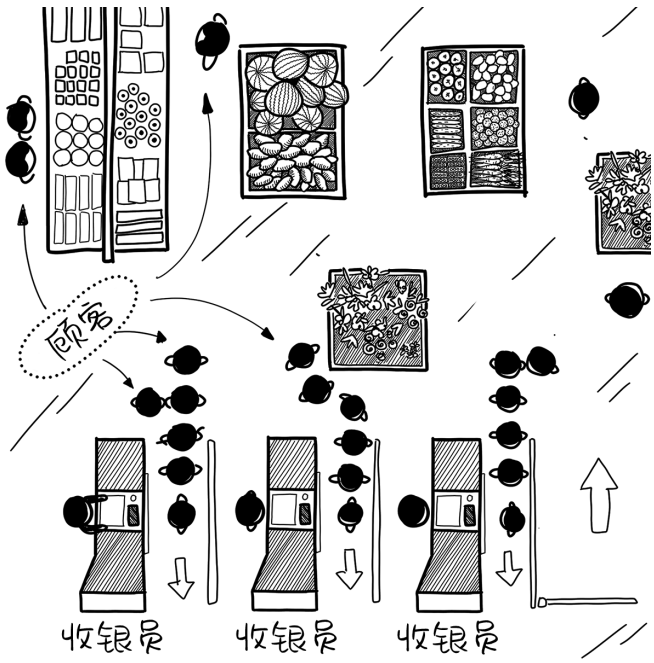


如果继续增加朋友的数量会发生什么呢？例如，再加入三位朋友，一共有六个人。执行程序的第二步只需半秒。则整个算法的执行时间可缩短至2.5s，加速比例为50%。



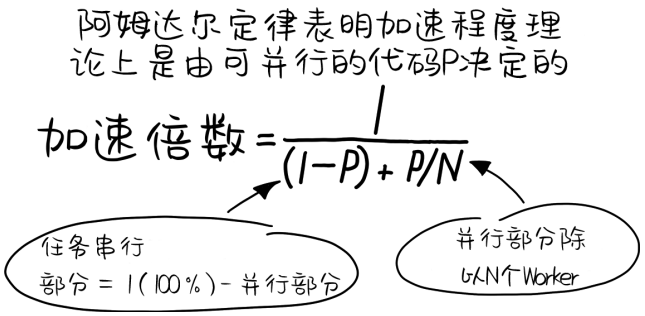
按照同样的逻辑，你甚至可以邀请城市中所有人，以立即完成算法的并行部分(但理论上，存在通信成本开销，这将在第6章中讨论)。最终，你仍然有至少2 s的延迟，也就是算法的串行部分。

并行程序的速度取决于其最慢的顺序部分。这一现象在你去购物中心时可以观察到。数百个人可以同时购物，很少相互干扰。但到了付款时，人们会排队等候，因为收银员的数量比准备离开的顾客少。

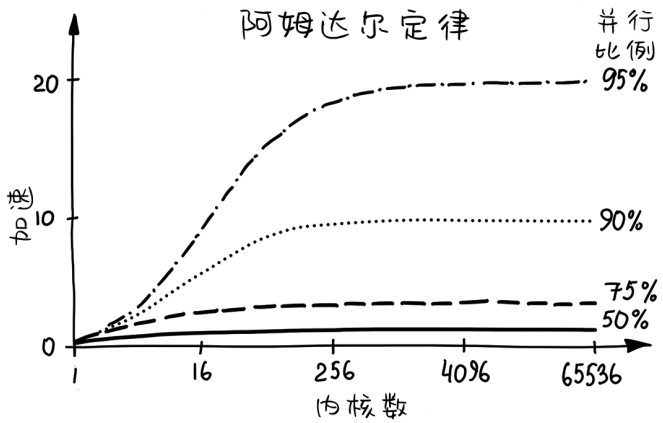


编程也是如此。由于无法加速程序的顺序部分，增加资源数量不会影响顺序部分的执行。这是理解阿姆达尔定律的关键。程序使用并行计算的极限速度受到程序顺序部分的限制。该定律描述了在假设使用并行计算的情况下，向系统添加资源可以期望获得的最大加速。例如，阿姆达尔定律预测，如果一段程序的三分之二是顺序的，则无论有多少处理器，加速倍数永远不会超过1.5倍。

更正式地，阿姆达尔定律可使用以下公式表示。



公式看起来很简单，我们填入值进行分析。例如，如果程序的33%是顺序的，那么添加100万个处理器最多可提供三倍的加速。我们无法对程序的三分之一的顺序部分进行加速，因此即使其余部分的程序能够瞬间运行完毕，性能收益也不会超过300%。虽然添加若干处理器通常可以提供显著的加速，但随着处理器数量的增加，优势很快就会消失。在不考虑算法或通信开销的情况下，对于不同比例的可并行代码，处理器数量与加速倍数之间的关系如下图所示。



我们也可以反向思考这个数学问题。例如，如果有2500个处理器，为了达到100倍加速，程序必须以何种程度进行并行化呢？将值代入阿姆达尔定律，得到$100 \leqslant 1/(S + (1 - S)/2500)$。通过计算$S$，可以得到程序中的串行部分要少于1%。

总之，阿姆达尔定律说明了只有在程序高度并行化时，使用多个处理器进行并行计算才真正有用。即使可以编写并行程序，也不一定总是值得执行，因为并行化带来的成本和开销有时会超过其收益。阿姆达尔定律是评估并行化程序收益的实用工具，可帮助判断是否值得进行并行化。

2.8 古斯塔夫森定律

并行化确实为程序性能关键环节实现了真正的加速，但无法加速程序的所有部分，除非程序属于并行问题。对于其他任务，性能收益则存在优化极限。

但是，我们也可以从不同的角度思考阿姆达尔定律。如果我们将可并行化部分所做的工作量加倍，从原先的3个任务变为6个，示例程序仍然只运行5s。因此，同时完成的任务数是6，程序仍然在5s内运行，总共完成8个任务。相比于两个处理器，任务完成量是原先的1.6倍。如果再添加几个处理器，每个处理器都完成相同的工作量，则可以在5s内完成11个任务。这样一来，完成的任务量增长为原先的2.6倍。

根据阿姆达尔定律，假定在工作量保持不变的前提下，加速率反映了并行程序执行时间的减少程度。然而，加速率也可以看作在恒定时间段内执行任务的增加量(吞吐量)。古斯塔夫森定律正是源自这一假设。

古斯塔夫森定律为并行限制提供了更乐观的视角。如果我们不断增加工作量，则顺序部分的影响将越来越小，拥有的处理器数量越多，则加速率也会成比例地增加。

因此，当有人引用阿姆达尔定律来说明并行无法解决问题时，你可以转而运用古斯塔夫森定律提出解决方法。这就是超级计算机和分布式系统在并行方面取得成功的关键，因为我们可以不断增加数据的规模。

现在，我们已经了解了并行计算，接下来将探讨并行与并发的关系。

2.9 并发与并行

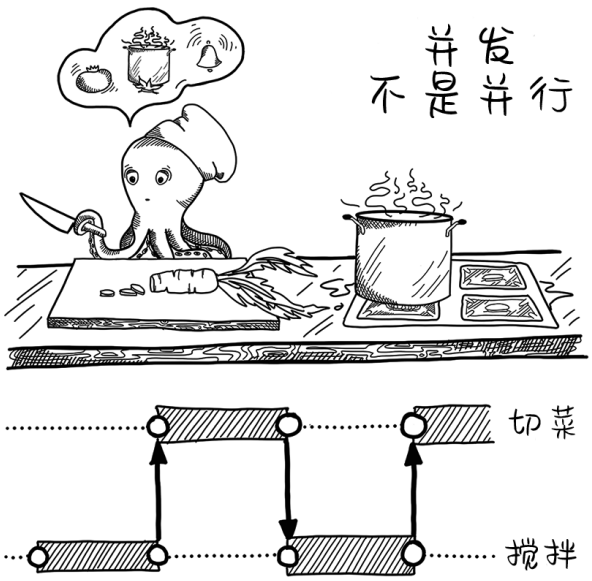
“并行”与“并发”这两个词的字面含义十分相似，非常容易造成混淆，甚至在计算机论文中也可能被误用。区分并行编程和并发编程至关重要，因为它们基于不同的概念层面，追求不同的目标。

并发是指多个任务在重叠的时间段内启动、运行和完成，而不是按照特定顺序。并行是指多个任务在具有多个计算资源的硬件上同时运行，如多核处理器。二者并不相同。

设想一名厨师在切沙拉期间要不时地搅拌炖锅里的汤这一场景。厨师必须停下

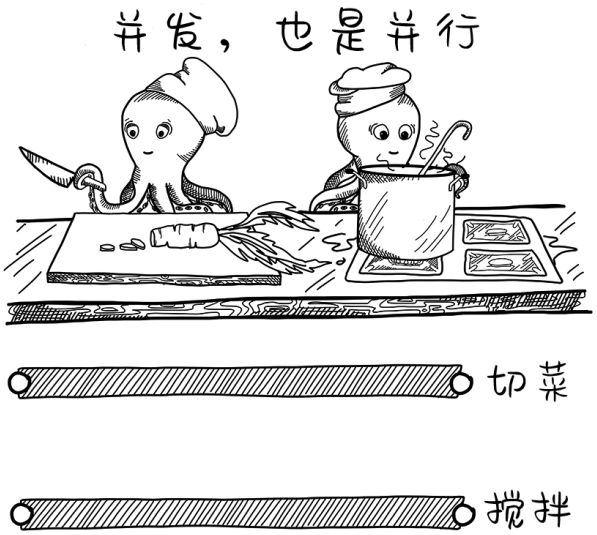
切菜的动作，检查炖锅，再继续切菜，并重复这一过程，直到所有工作完成。

正如你所看到的，这里只有一个处理资源，即厨师，而并发主要与流程有关。如果不使用并发，厨师必须等到炖锅里的汤准备好后才能开始切菜。



并行性是一种实现属性，特指在运行时多个任务同时物理执行，需要具有多个计算资源的硬件。并行位于硬件层面。

回到厨房的比喻，假如现在有两名厨师，一名负责搅拌，一名负责切沙拉。我们通过增加另一个处理资源，即另一名厨师，从而分配工作。并行是并发的子类，在同时处理多项任务之前，我们必须首先能够管理多项任务。



并发和并行关系的本质在于，并发在不影响结果的条件下能转换为并行，但并发本身并不意味着并行。此外，并行不一定是并发。通常情况下，优化器可以使用管道处理、宽向量操作、单指令流多数据流(Single Instruction Multiple Data，SIMD)操作和分治等技术，将没有语义并发的程序分解为并行组件。本书将在后续章节介绍其中相关方法。

正如Unix和Go编程先驱Rob Pike指出的，“并发是同时处理很多事情，并行是同时做很多事情”。[1]程序的并发依赖于编程语言及其编程方式，而并行则依赖于实际执行环境。在单核CPU中，我们能够获得并发，但不能获得并行。但是，两者都超越了传统的顺序模型，顺序模型同时只能做一件事。

为了更好地区分并发和并行，可以考虑以下几点。

- 程序可以是并发的，但不是并行的。程序在给定时间段内处理多个任务(即使同一时刻没有在执行两个任务，也视为同时处理多个任务，参见第6章)。
- 程序可以是并行的，但不是并发的，这意味着程序可以同时处理单个任务的多个子任务。
- 程序可以既不是并行的，也不是并发的，这意味着程序仅能按顺序处理单个任务，而任务永远不会被分解为子任务。
- 程序可以既是并行的，又是并发的，这意味着程序可以在同一时刻并发处理多个任务或单个任务的子任务(并行执行它们)。

假设有一段将值插入哈希表的程序。从将插入操作分配到多个内核的角度来看，这是并行的。但从协调对哈希表访问的角度来看，这是并发的。如果仍然不理解后者，下一章将详细解释这一概念。

并发涵盖了各种主题，包括进程之间的相互作用、共享和竞争资源(如内存、文件和I/O访问)、多个进程之间的同步以及在进程之间分配处理器时间。这些问题不仅出现在多处理器和分布式处理环境中，也出现在单处理器系统中。在下一章中，我们将从了解程序运行的环境开始，包括计算机硬件和运行时系统。

2.10 本章小结

- 问题一旦转换为程序，就能被分解为一系列任务。最简单的方式是按顺序执行。
- 任务可以是逻辑上独立的工作单元。

1 Rob Pike 在 Heroku 的 Waza 会议上发表了题为“并发不是并行”的演讲，网址是 https://go.dev/blog/waza-talk。

- 顺序计算意味着程序中列出的每项任务都依赖于所有前置任务的执行，即按代码中列出的顺序执行。
- 串行执行指的是依次在一个处理单元上执行一组有序的指令。当每个任务的输入需要前一个任务的输出时，串行执行是必要的。
- 并行执行指的是同时执行多个计算。当任务可以独立执行时，可以使用并行执行。
- 并行计算使用多个处理单元同时解决问题。这通常导致对程序进行大幅度的重新设计，即分解问题、创建或调整算法、在程序中添加同步点等。
- 并发描述涉及同时处理多个任务的工作方式。并行依赖于实际的运行时环境，并且需要多个处理资源和分解算法中的任务独立性。程序的并发依赖于编程语言及其编程方式，而并行则依赖于实际执行环境。
- 阿姆达尔定律是一个实用的工具，用于评估并行执行程序的收益，以判断这样做是否具有实际意义。
- 古斯塔夫森定律描述了如何在阿姆达尔定律限制下，利用计算机系统完成更多工作。