c13n #41

c13n

2025 年 11 月 19 日

第 I 部

Git 内部机制与核心操作

马浩琨

Nov 10, 2025

1 导言

大多数开发者在使用 Git 时，往往停留在 `git add`、`git commit` 和 `git push` 等高层命令层面，将 Git 视为一个神秘的「黑盒」。这种使用方式虽然高效，但在面对复杂冲突或状态异常时，却容易陷入困境。理解 Git 的内部机制不仅能帮助开发者精准排查问题，还能深化对 `reset`、`rebase` 和 `merge` 等操作的区别认知，从而建立正确的「Git 数据模型」心智模型。本文旨在通过解析 `.git` 目录结构，深入探讨 Git 的核心对象模型，并引导读者手动操作底层命令及编写简单脚本，模拟实现 `git init`、`git add` 和 `git commit` 等核心功能。本文面向有一定 Git 使用经验的中高级开发者，希望通过实践让读者真正「拥有」Git。

2 Git 的基石 —— 内容寻址文件系统

要理解 Git 的内部机制，首先需要探索 `.git` 目录的结构。执行 `tree .git` 命令后，可以看到一个典型仓库的骨架。其中，`objects` 目录是 Git 的数据存储核心，所有文件、目录和提交都存储于此；`refs` 目录则用于存储引用，包括分支和标签；`HEAD` 文件作为一个引用，指向当前所在分支；而 `index` 文件是暂存区的物理体现，以二进制形式记录文件状态。

Git 的核心在于内容寻址机制。这种机制不是通过文件名来访问数据，而是基于文件内容计算出一个唯一密钥，即 SHA-1 哈希值（未来可能过渡到 SHA-256）。具体来说，密钥的计算公式为 $Key = \text{SHA1}(\text{“blob”} + \text{文件内容长度} + \backslash 0 + \text{文件内容})$。例如，我们可以使用命令行工具手动计算一个字符串的 SHA-1 值。执行 `echo -e 'blob 16\0Hello Git World!' | openssl dgst -sha1` 或 `printf blob 16\0Hello Git World! | shasum`，输出结果便是该内容的唯一标识。内容寻址的优势在于确保数据的完整性 —— 任何微小改动都会导致密钥变化，防止数据被篡改；同时，它还支持去重，相同内容在对象库中仅存储一份。

3 Git 的核心对象模型

Git 的对象模型由三种核心类型构成：Blob、Tree 和 Commit。每种对象都承担着特定角色，并通过有向无环图（DAG）相互关联。

Blob 对象负责存储文件数据本身，但不包含任何文件名信息。我们可以通过底层命令 `git hash-object -w` 来创建并存储一个 Blob。例如，执行 `echo Hello, Git > hello.txt` 创建一个文件，然后运行 `git hash-object -w hello.txt`，该命令会输出一个 SHA-1 哈希值（如 8ab686eafeb1f44702738c8b0f24f2567c36da6d），同时将对象文件存入 `.git/objects` 目录。使用 `find .git/objects -type f` 可以查看新生成的文件，这验证了 Blob 的存储过程。

Tree 对象则代表一个目录结构，它存储文件名、文件模式（权限）以及指向对应 Blob 或其他 Tree 的引用。创建 Tree 对象需要先通过 `git hash-object -w` 生成多个 Blob，然后使用 `git update-index` 将这些 Blob 加入一个「假」的暂存区，最后通过 `git write-tree` 将当前索引状态写入一个 Tree 对象。这个过程模拟了 Git 如何组织文件系统目录。

Commit 对象用于存储提交的元数据，包括指向一个顶层 Tree 对象（代表项目快照）、父

Commit 对象（首次提交无父提交，合并提交有多个）、作者信息、提交时间戳和提交信息。我们可以基于已有的 Tree 对象，使用 `echo First commit | git commit-tree <tree-sha>` 来创建一个 Commit 对象。例如，如果 Tree 的 SHA-1 为 `abc123`，则命令会生成一个新的 Commit 哈希，这标志着一次提交的诞生。

这些对象之间的关系构成了 Git 版本历史的基础。Commit 指向 Tree，Tree 则包含多个 Blob 或子 Tree，形成一个有向无环图。这种结构确保了数据的高效存储和检索，是 Git 强大版本控制能力的核心。

4　实现核心操作 —— 从底层命令到脚本

通过底层命令模拟 Git 的核心操作，可以帮助我们更直观地理解其工作原理。首先，从 `git init` 开始。我们可以手动创建仓库骨架：建立 `.git` 目录及其子目录（如 `objects`、`refs/heads` 和 `refs/tags`），然后初始化 `HEAD` 文件，内容为 `ref: refs/heads/master`。这个过程本质上是构建 Git 仓库的基础环境。

接下来，模拟 `git add` 操作。该命令实际上执行两个步骤：将工作区文件内容创建为 Blob 对象并存入 `objects` 目录，同时更新索引文件（`.git/index`）以记录文件名、模式和 Blob 的 SHA-1。我们可以使用 `git hash-object -w` 创建 Blob，然后用 `git update-index` 更新索引。例如，执行 `git update-index --add --cacheinfo 100644 <blob-sha> filename.txt` 将文件加入索引，再通过 `git ls-files --stage` 查看索引内容，验证文件状态。

最后，模拟 `git commit` 操作。这一过程涉及三个关键步骤：用当前索引创建 Tree 对象（`git write-tree`）、基于 Tree 和父 Commit 创建 Commit 对象（`git commit-tree`），以及更新分支引用。具体来说，先运行 `tree_sha=$(git write-tree)` 获取 Tree 哈希，然后执行 `commit_sha=$(echo My commit msg | git commit-tree $tree_sha)` 生成 Commit 哈希，最后通过 `echo $commit_sha > .git/refs/heads/master` 将分支指向新提交。此时，使用 `git log --oneline $commit_sha` 可以查看刚刚创建的提交历史，这标志着一个完整提交周期的实现。

除了核心对象，Git 还包含其他重要概念，如 Tag 对象和 Packfiles。Tag 对象是一种特殊类型，指向特定 Commit，用于提供永久性标记；Packfiles 则是 Git 的压缩机制，将多个松散对象打包以节省空间。这些机制进一步优化了 Git 的性能和可用性。

高层命令与底层命令之间存在紧密联系。例如，`git status` 通过比较 `HEAD`、`index` 和工作区三者的 Tree 差异来报告状态；`git branch` 本质上是在 `refs/heads` 下创建或删除文件；而 `git checkout` 则用指定 Commit 的 Tree 覆盖工作区并更新 `HEAD`。理解这些关系有助于在复杂场景中灵活运用 Git。

回顾全文，Git 的本质是一个「内容寻址文件系统」，其强大之处源于 Blob、Tree 和 Commit 对象构建的版本控制模型。鼓励读者在遇到问题时，多用 `git cat-file -p` 和 `git ls-tree` 等命令探查内部状态，以巩固理解。下一步，可以尝试用 Python 或 Go 等语言实现一个简单的 `my-git` 工具，这将进一步深化对 Git 原理的掌握。通过这种从理论到实践的探索，我们不仅能揭开 Git 的魔法外衣，还能在开发中游刃有余。

第 II 部

基数排序（Radix Sort）算法

李睿远

Nov 11, 2025

在计算机科学中，排序算法是基础且关键的主题。常见的比较排序算法如快速排序和归并排序，其时间复杂度通常为 O(n log n)，这是基于比较操作的理论下限。然而，是否存在一种方法能够突破「比较」这一范式，实现更优的性能呢？答案是肯定的，基数排序作为一种非比较型整数排序算法，通过逐位处理数字，可以在特定条件下达到线性时间复杂度。本文将深入解析基数排序的核心原理、实现方式及其应用，帮助你全面掌握这一高效算法。

5 基数排序初探

基数排序的核心思想在于将待排序元素视为由多个「位」组成的序列，而不是直接比较整体大小。具体来说，算法从最低位或最高位开始，依次对每一位进行稳定的排序操作。这个过程可以类比为整理扑克牌：如果我们先按点数分类，再在同点数内按花色排序，就类似于最低位优先方法；反之，如果先按花色分，再按点数排序，则类似于最高位优先方法。这种分步处理的方式使得基数排序能够避免直接比较元素，从而在整数或字符串排序中展现出独特优势。

基数排序的关键特性之一是其对稳定性的依赖。稳定性指的是在排序过程中，相等元素的相对顺序保持不变。基数排序的每一轮排序都必须使用稳定的次级算法，通常选择计数排序，因为如果次级排序不稳定，整个算法的正确性将无法保证。此外，基数排序主要适用于整数或可以分解为「位」的数据类型，如字符串。对于浮点数或其他复杂类型，则需要额外处理，例如通过转换或偏移来适应算法要求。

6 深入原理

最低位优先（LSD）方法是基数排序中最常见的实现方式。它从数字的最低位（如个位）开始，依次向高位进行稳定排序。例如，给定数组 `[170, 45, 75, 90, 2, 802, 24, 66]`，LSD 会先按个位排序，结果可能为 `[170, 90, 2, 802, 24, 45, 75, 66]`；接着按十位排序，得到 `[2, 802, 24, 45, 66, 170, 75, 90]`；最后按百位排序，完成整个排序过程。LSD 实现简单直观，但它是一种离线算法，需要预先知道最大数字的位数，以确定排序轮数。排序结束后，序列自然有序，无需额外合并步骤。

最高位优先（MSD）方法则从最高位开始排序，然后递归处理每个子桶。例如，对同一数组，MSD 会先按百位分桶，将数字分配到不同范围，再对每个桶内的数字递归排序低位。这种方法更接近分治策略，可能在某些情况下提前终止，例如当某个桶内只有一个元素时。MSD 在字符串字典序排序中尤为自然，因为它可以直接处理前缀。与 LSD 相比，MSD 在实现上可能更复杂，且性能受数据分布影响较大，但它能更早地排除无关比较。

7 代码实现

在实现基数排序时，我们通常选择计数排序作为稳定的次级排序算法。计数排序通过统计每个数字的出现次数，并利用前缀和确定元素位置，从而保证稳定性。以下以 LSD 方法为例，使用 Python 语言实现基数排序。代码将分步解释，确保每个细节清晰易懂。

首先，我们需要确定最大数字的位数，以决定排序轮数。这可以通过遍历数组并计算最大值来实现。例如，如果最大数字是 802，其位数为 3，则需进行三轮排序（个位、十位、百位）。

```
def radix_sort(arr):
    # 步骤 1: 寻找最大数，确定位数
    max_num = max(arr)
    exp = 1 # 从个位开始
    while max_num // exp > 0:
        # 使用计数排序对当前位进行排序
        n = len(arr)
        output = [0] * n
        count = [0] * 10 # 十进制数字范围 0-9

        # a. 计数：统计当前位上每个数字的出现次数
        for i in range(n):
            index = (arr[i] // exp) % 10
            count[index] += 1

        # b. 计算位置：将计数转换为前缀和，表示起始索引
        for i in range(1, 10):
            count[i] += count[i - 1]

        # c. 构建输出：从后向前遍历，保证稳定性
        i = n - 1
        while i >= 0:
            index = (arr[i] // exp) % 10
            output[count[index] - 1] = arr[i]
            count[index] -= 1
            i -= 1

        # d. 复制回原数组
        for i in range(n):
            arr[i] = output[i]

        exp *= 10 # 移动到下一位
```

在这段代码中，我们首先计算最大数字以确定循环次数。然后，在每一轮中，我们使用计数排序处理当前位。计数步骤统计每个数字（0-9）的出现频率；位置计算步骤将计数数组转换为前缀和，以确定每个数字在输出数组中的起始索引；构建输出步骤从原数组末尾开始遍历，确保相等元素的顺序不变；最后，将结果复制回原数组。关键点在于从后向前遍历，这维护了稳定性，因为计数排序中后出现的元素会被放置在输出数组的较后位置。每轮结束后，exp 乘以 10，以处理更高位。

实现基数排序时，常见的陷阱包括忽略稳定性或错误处理数字位。例如，如果构建输出时从前向后遍历，可能会破坏相对顺序。另外，确保 exp 正确递增，避免遗漏高位或重复处理。

8 算法分析

基数排序的时间复杂度为 O(d * (n + k))，其中 d 是最大数字的位数，n 是元素个数，k 是每位可能取值的范围（对于十进制，k=10）。当 d 为常数且 k 与 n 同阶时，时间复杂度可视为线性 O(n)。相比之下，比较排序算法如快速排序的平均时间复杂度为 O(n log n)，基数排序在特定条件下更具优势。例如，如果数字范围有限且位数较少，基数排序能显著提升性能。

空间复杂度主要来自计数排序的辅助数组，包括计数数组和输出数组，因此为 O(n + k)。这表示基数排序不是原地算法，需要额外内存空间。尽管这可能成为内存受限环境中的缺点，但其稳定性和高效性在许多应用中值得权衡。

基数排序的优缺点总结如下：优点包括在整数排序中可能达到线性时间、稳定性高；缺点则在于适用范围有限、需要额外空间，且当数字位数差异大时，性能可能不如某些自适应比较排序。因此，在选择算法时，需考虑数据特性和环境约束。

基数排序在实际应用中常用于处理固定位数的整数序列，例如身份证号、电话号码排序，或字符串字典序排列。在计算机图形学中，它也可用于某些像素处理算法。这些场景充分利用了基数排序的稳定性和高效性。

总结来说，基数排序通过「按位排序」和「依赖稳定性」的核心思想，实现了非比较排序的突破。其线性时间复杂度的优势在合适条件下显著，但需注意数据类型的限制。鼓励读者在涉及整数或字符串排序的任务中，尝试应用这一算法。

思考与拓展部分留给读者进一步探索：例如，如何对包含负数的数组进行基数排序？可以通过分离正负数组或使用偏移量处理；MSD 实现需调整代码结构为递归形式；对于非十进制数字，只需修改进制基数即可适应。这些拓展问题有助于深化对算法灵活性的理解。

第 III 部

异步编程（Async Programming）机制

黄梓淳

Nov 12, 2025

从「为什么需要异步」到「亲手实现一个微型事件循环」—— 本文将带你逐步揭开异步编程的神秘面纱。我们将从基础概念出发，深入探讨事件循环的核心机制，并最终通过动手实现一个微型异步框架来巩固理解。无论你是初学者还是有一定经验的开发者，这篇文章都将帮助你从本质层面掌握异步编程的精髓。

9　开篇明义 —— 我们为什么需要异步?

在计算机编程中，异步编程是一种处理输入输出（I/O）操作的高效方式。为了直观理解其必要性，我们可以通过一个生动的比喻来对比同步与异步的行为模式。想象一位厨师在厨房工作：同步模式就像一位「单线程」厨师，他必须严格按照顺序执行任务 —— 先烧水，等水烧开后才能切菜，然后等菜切好才能开始炒菜。在这个过程中，厨师在等待水烧开时完全处于闲置状态，资源利用率极低。而异步模式则像一位「有经验」的厨师，他会在烧水的同时去切菜，当水烧开的事件发生时，他再回来处理下面条的任务。这种模式显著提高了效率，使得资源在等待期间不被浪费。

从编程角度审视，我们面临的主要问题集中在两类任务上。I/O 密集型任务，例如网络请求、文件读写或数据库查询，其特点是大部分时间都花费在等待外部资源响应上，而 CPU 在此期间处于闲置状态。另一方面，CPU 密集型任务，如计算圆周率或视频编码，则真正消耗 CPU 计算资源。异步编程的核心目标正是解决 I/O 等待导致的资源浪费和性能瓶颈，使得单线程环境也能高效处理大量并发 I/O 操作。本文旨在引导读者理解异步编程的核心思想，剖析 `async/await` 语法糖背后的运行机制，并最终实现一个极简的、可运行的异步框架。

10　核心概念 —— 构建异步世界的基石

要深入异步编程，首先需要厘清几个关键概念。阻塞与非阻塞描述了调用发起后程序流程的行为：阻塞调用会导致程序停滞，直到操作完成；而非阻塞调用则允许程序立即继续执行，无需等待结果。同步与异步则关注任务完成的通知方式：同步模式下，程序主动等待任务完成；而异步模式下，程序通过回调或事件通知被动获知任务结果。

并发与并行是另一个容易混淆的概念。并发指的是多个任务在宏观上「同时」执行，通过快速切换实现，即使在单核 CPU 上也能实现；而并行则指真正的同时执行多个任务，通常需要多核 CPU 的支持。异步编程主要实现的是并发，而非必然的并行。例如，在单线程环境中，通过事件循环调度，多个 I/O 操作可以交替进行，从而在用户感知上实现「同时」处理。

11　异步的引擎 —— 事件循环（Event Loop）剖析

事件循环是异步编程的「大脑」和调度中心，它是一个不断循环的程序结构，负责协调和管理所有异步任务。事件循环的核心组件包括任务队列和事件触发器。任务队列（Task Queue 或 Callback Queue）用于存放准备就绪的回调函数或任务；事件触发器（Event Demultiplexer）则是操作系统提供的机制，如 `epoll`、`kqueue` 或 `IOCP`，用于监听多个 I/O 操作，并在某个操作完成时发出通知。

事件循环的工作流程遵循一个经典循环模式。首先，从任务队列中取出一个任务执行；执行过程中，如果遇到异步 I/O 调用，则将其交给事件触发器监听，而事件循环不会等待该操作完成，而是立即继续执行下一个任务；随后，检查事件触发器，看是否有已完成的 I/O 操

作，如果有，则将其对应的回调函数放入任务队列；最后，重复这一过程。只要所有任务都是纯计算或非阻塞 I/O，事件循环就能持续运转，不会被任何耗时操作阻塞。这种设计确保了系统的高响应性和资源利用率。

12 从回调地狱到现代语法 —— 异步编程的演进

异步编程的发展经历了多个阶段，每个阶段都旨在解决前一代的痛点。最初，回调函数（Callback）是异步编程的基础实现方式。例如，在 Node.js 中，`fs.readFile(file, callback)` 允许在文件读取完成后执行指定的回调函数。虽然回调函数简单直接，但它容易导致「回调地狱」（Callback Hell），即多层嵌套的回调使得代码难以阅读和维护。

为了改善代码结构，Promise 应运而生。Promise 将异步操作封装成一个对象，代表一个未来完成或失败的操作及其结果值。通过链式调用（如 `.then().catch()`），Promise 提供了更线性的代码流，有效缓解了回调地狱问题。Promise 本质上是一个状态机，包含 pending（等待中）、fulfilled（已完成）和 rejected（已拒绝）三种状态。

现代异步编程的终极方案是 Async/Await，它基于 Promise 的语法糖，让开发者能够以编写同步代码的方式处理异步操作。`async` 关键字用于声明一个异步函数，该函数总会返回一个 Promise；`await` 关键字则只能在 `async` 函数中使用，它会「暂停」函数的执行，等待后面的 Promise 解决，然后恢复执行并返回结果。这种语法大幅提升了代码的清晰度和直观性，错误处理也可以使用传统的 `try/catch` 块，使得异步代码更易于理解和维护。

13 动手实践 —— 实现一个微型异步框架

本章将带领读者亲手实现一个微型异步框架，使用 Python 的生成器（Generator）来模拟 `async/await` 的暂停和恢复机制。通过这个实践，我们将直观理解事件循环如何调度任务。

首先，我们定义设计目标：实现一个 `EventLoop` 类和一个 `Task` 类，其中 `Task` 类包装一个生成器来模拟协程，并能够处理模拟 I/O 操作（用 `time.sleep` 代替）。以下是分步代码实现和解读。

我们首先定义一个模拟异步 sleep 的函数 `async_sleep`。该函数返回一个生成器，通过 `yield` 发送一个信号，告知事件循环该任务需要休眠指定时间。

```
def async_sleep(delay):
    yield f“sleep_{delay}” # 通过 yield 发送休眠信号，事件循环根据此信号暂
        ↪ 停任务
```

接下来，我们实现 `Task` 类。该类封装一个生成器（代表协程），并提供 `run` 方法来驱动协程执行。当协程执行到 `yield` 语句时，任务会暂停，并根据 yield 的值处理相应操作（如安排定时器回调）。

```
class Task:
    def __init__(self, coro):
        self.coro = coro # 保存传入的生成器（协程）
        self.complete = False # 标记任务是否完成
```

```
    def run(self):
        try:
            # 驱动协程执行，next() 或 send() 返回 yield 的值
            signal = next(self.coro)
            # 这里假设 signal 是休眠信号，实际中可能处理多种信号
            return signal
        except StopIteration:
            self.complete = True # 生成器结束，任务完成
```

然后，我们实现核心的 EventLoop 类。该类维护就绪任务队列和休眠任务字典，并通过模拟时间推进来调度任务。

```
class EventLoop:
    def __init__(self):
        self.ready_tasks = [] # 就绪任务队列
        self.sleeping_tasks = {} # 休眠任务字典，键为唤醒时间，值为任务列表
        self.current_time = 0 # 模拟当前时间

    def add_task(self, task):
        self.ready_tasks.append(task) # 将新任务加入就绪队列

    def run(self):
        while self.ready_tasks or self.sleeping_tasks:
            # 处理所有就绪任务
            while self.ready_tasks:
                task = self.ready_tasks.pop(0)
                signal = task.run()
                if signal and signal.startswith( “sleep_” ):
                    # 解析休眠时间，并安排唤醒
                    delay = int(signal.split( “_” )[1])
                    wake_time = self.current_time + delay
                    if wake_time not in self.sleeping_tasks:
                        self.sleeping_tasks[wake_time] = []
                    self.sleeping_tasks[wake_time].append(task)
            # 推进时间并唤醒到期任务
            if self.sleeping_tasks:
                min_wake_time = min(self.sleeping_tasks.keys())
                self.current_time = min_wake_time
                woke_tasks = self.sleeping_tasks.pop(min_wake_time)
                self.ready_tasks.extend(woke_tasks)
```

最后，我们编写示例代码来演示框架的运行。定义示例任务函数，使用生成器语法模拟异步操作，并添加任务到事件循环中执行。

```
def example_task(name, seconds):
    print(f "{name} started at {loop.current_time} ")
    yield from async_sleep(seconds) # 模拟 await asyncio.sleep(seconds)
    print(f "{name} resumed at {loop.current_time} ")

# 创建事件循环实例并添加任务
loop = EventLoop()
loop.add_task(Task(example_task( "Task1" , 2)))
loop.add_task(Task(example_task( "Task2" , 1)))
loop.run()
```

运行上述代码，预期输出为：Task1 和 Task2 几乎同时开始，但 Task2 先恢复（因为休眠时间短），Task1 后恢复。这演示了并发执行的效果 —— 尽管是单线程，任务在等待期间不会阻塞事件循环。

通过本文的探讨，我们从异步编程的必要性出发，逐步理解了事件循环、Promise 和 Async/Await 等核心概念。最关键的是，我们通过动手实现一个简单的异步框架，揭示了 await 背后「暂停/恢复」的协程机制，这有助于读者在遇到复杂问题时从底层原理寻找解决方案。

在现实世界中，异步框架已广泛应用于各种编程环境。Python 的 asyncio 库基于类似我们实现的模型，但更复杂高效，使用真正的非阻塞 I/O；JavaScript 语言内置事件循环，成为前端和后端（如 Node.js）开发的基石；Go 语言则通过 Goroutine 和 Channel 提供了另一种优雅的并发模型。开发者在使用这些工具时，应注意避免在事件循环中执行真正的阻塞操作（如 time.sleep 或密集计算），否则会「卡死」整个循环。善用 async/await 语法，结合对底层机制的理解，将帮助你编写出清晰、高效的异步代码。

第 IV 部

基数排序（Radix Sort）算法

杨子凡

Nov 13, 2025

在数据处理领域，排序是一项基础而关键的操作。当我们面对海量数据时，例如为成千上万的手机号码排序，传统的比较排序算法如快速排序或归并排序虽然高效，但它们的时间复杂度存在一个理论下限 $O(n \log n)$。这自然引出一个问题：是否存在一种不通过比较就能实现排序的算法？答案是肯定的，基数排序正是这样一种非比较型的整数排序算法。它的核心思想是将整数按位数「切割」，逐位进行排序，在特定条件下时间复杂度甚至可以达到 $O(n)$。本文旨在带领读者彻底理解基数排序的原理，掌握其手动实现方法，并清晰认识其优缺点与适用场景。

14 核心思想与原理

基数排序的命名来源于「基数」这一概念。基数指的是进制的基数，例如在十进制中，基数为１０，这意味着我们需要１０个「桶」来分别存放数字０到９。关键码是排序所依据的属性，对于整数排序而言，关键码就是数字本身。稳定性是排序算法的一个重要特性，它表示如果两个元素在排序前相等，那么排序后它们的相对顺序保持不变。在基数排序中，稳定性至关重要，因为后序位的排序不能打乱前序位已经排好的顺序。

基数排序有两种主要实现方式：最低位优先（LSD）和最高位优先（MSD）。LSD 从最低位（如个位）开始排序，依次向最高位进行，实现简单直观，是本文主要讲解的方式。MSD 则从最高位开始，然后递归地对每个桶内的数据进行下一位排序，更像一种分治策略，但实现稍显复杂。通过对比，LSD 因其易于理解和实现，常被用作入门教学的首选。

15 逐步拆解 LSD 基数排序过程

让我们通过一个具体示例来逐步拆解 LSD 基数排序的过程。假设我们有数组 `[170, 45, 75, 90, 2, 802, 2, 66]`。首先，我们需要找到数组中的最大数字，以确定最大位数。这里最大数字是８０２，有３位，因此我们需要进行３轮排序。

第一轮从最低位（个位）开始。我们创建１０个桶，对应数字０到９。然后遍历数组，根据每个数字的个位数字将其放入对应的桶中。例如，１７０的个位是０，放入桶０；４５的个位是５，放入桶５；以此类推。分配完成后，我们按桶号０到９的顺序依次收集元素放回数组。此时，数组按个位数字排序，结果为 `[170, 90, 2, 802, 2, 45, 75, 66]`。

第二轮处理十位数字。同样，创建１０个桶，根据十位数字分配元素。例如，１７０的十位是７，放入桶７；９０的十位是９，放入桶９。由于排序是稳定的，个位相同的数字在十位排序时会保持相对顺序。收集后，数组按十位和个位排序，结果为 `[2, 802, 2, 45, 66, 170, 75, 90]`。

第三轮处理百位数字，过程相同。完成后，整个数组有序，最终结果为 `[2, 2, 45, 66, 75, 90, 170, 802]`。这个示例清晰地展示了 LSD 基数排序的逐位排序过程，强调了稳定性在保持顺序中的作用。

16 动手实现基数排序

以下是用 Python 实现 LSD 基数排序的代码。我们将逐步解释关键部分，确保读者能够理解每一行代码的作用。

```
def radix_sort(arr):
```

```
    # 步骤一：寻找最大值与最大位数
    max_num = max(arr)
    max_digit = len(str(max_num)) # 通过转换为字符串获取位数

    # 步骤二：核心排序循环
    for digit in range(max_digit):
        # 创建 10 个桶
        buckets = [[] for _ in range(10)]

        # 分配过程：根据当前位数字将元素放入对应桶
        for num in arr:
            current_digit = (num // (10 ** digit)) % 10
            buckets[current_digit].append(num)

        # 收集过程：按顺序将桶中元素放回数组
        arr = []
        for bucket in buckets:
            arr.extend(bucket)

    return arr
```

现在，让我们详细解读这段代码。在步骤一中，我们使用 max(arr) 找到数组中的最大值 max_num，然后通过 len(str(max_num)) 计算其位数 max_digit。这里，将数字转换为字符串后取长度是一种简单直观的方法，用于确定排序的轮数。

在核心循环中，digit 从 0 到 max_digit-1 迭代，表示当前处理的位数（0 表示个位，1 表示十位，以此类推）。对于每一轮，我们使用列表推导式创建 10 个空桶 buckets。

分配过程中，对于每个数字 num，我们计算当前位的数字。表达式 (num // (10 ** digit)) % 10 是关键：当 digit=0（个位）时，10 ** 0 等于 1，num // 1 仍是 num，然后 % 10 得到个位数字；当 digit=1（十位）时，10 ** 1 等于 10，num // 10 去掉个位，然后 % 10 得到十位数字。这样，我们就能准确提取指定位的数字值。

收集过程时，我们初始化一个新数组 arr，然后按桶号 0 到 9 的顺序，使用 extend 方法将每个桶中的元素依次添加回数组。由于 extend 保持元素顺序，且桶是按数字顺序创建的，这确保了排序的稳定性。最终，函数返回排序后的数组。

17　深入分析与探讨

基数排序的性能分析是理解其优势的关键。设待排序元素个数为 n，最大位数为 k，基数为 r（在十进制中 $r = 10$）。每一轮分配需要遍历所有元素，时间复杂度为 $O(n)$，收集同样需要 $O(n)$ 时间。由于有 k 轮，总时间复杂度为 $O(k \times n)$。当 k 远小于 n 时，性能接近线性，表现出高效性。

空间复杂度方面，我们需要额外的 $O(n + r)$ 空间来存储桶和元素。这是一种典型的以空间换时间策略，在内存充足的情况下值得采用。

基数排序的优点包括时间复杂度可能达到 $O(n)$，且是稳定排序。缺点是非原地排序，需要额外空间，且适用范围有限，通常只适用于整数或可表示为整数的类型。如果最大位数 k 很大，效率会显著下降。

与其他排序算法相比，基数排序在特定场景下优势明显。例如，与快速排序相比，基数排序稳定且对数据分布不敏感；与计数排序相比，基数排序通过分治按位处理，避免了数据范围大时计数排序的空间消耗问题。

18 扩展与变种

标准 LSD 基数排序无法直接处理负数。一个常见的解决方案是将数组拆分为正数和负数两部分。对负数部分取绝对值进行排序，然后反转顺序（因为负数绝对值越大，实际值越小），再与正数部分合并。这样可以扩展算法的适用性。

MSD 基数排序从最高位开始，递归排序，适用于某些场景，如字符串排序，但实现更复杂。对于字符串，可以按字符的 ASCII 码逐位排序，原理类似整数排序。

基数排序也可以应用于其他数据类型，如日期，只要能将它们转换为整数序列。例如，日期可以表示为年月日的数字组合，然后按位排序。

本文详细介绍了基数排序的核心思想、LSD 实现步骤、性能分析和扩展应用。基数排序作为一种非比较排序算法，在特定条件下展现出高效性，尤其适用于整数排序场景。通过手动实现代码，读者可以更深入地理解其工作原理。鼓励读者在实践中探索基数排序的更多应用，例如在大数据处理或数据库索引中。

19 附录与思考题

在附录中，我们提出几个思考题供读者进一步探索。首先，考虑如何使用队列数据结构来优化桶的实现，使得收集过程更自然。其次，尝试修改代码以处理包含负数的数组。最后，如果待排序数字的范围已知且较小，可以探索用计数排序替代每轮的桶排序，以优化性能。这些练习有助于加深对基数排序的理解和应用。

第 V 部

信号量（Semaphore）机制

杨子凡

Nov 15, 2025

掌握这个操作系统与并发编程的基石，亲手用代码实现它。

在并发编程中，多个线程或进程同时访问共享资源时，常常会导致不可预测的结果。考虑一个经典的生产者－消费者问题场景：假设有一个共享缓冲区，生产者线程负责向缓冲区添加数据，消费者线程负责从缓冲区取出数据。如果不加任何同步控制，生产者可能在消费者尚未处理完数据时就覆盖旧数据，或者消费者可能读取到无效或重复的数据。这种数据竞争和数据不一致问题，会严重破坏程序的正确性和可靠性。那么，我们如何协调多个线程或进程，确保它们安全、有序地访问有限的共享资源呢？

信号量正是为了解决这类并发问题而诞生的。它是由荷兰计算机科学家 Dijkstra 在二十世纪六十年代提出的伟大思想。简单来说，信号量是一个计数器，用于控制对共享资源的访问线程或进程数量。本文将带领您从理论到实践，深入理解信号量的核心机制，并亲手用代码实现一个基本的信号量。

20 信号量的核心概念剖析

信号量的本质是一个非负整数计数器，它通过两个不可分割的原子操作来管理资源访问。这两个操作通常被称为 wait（或 P、down、acquire）和 signal（或 V、up、release）。wait 操作的语义是尝试获取资源：如果计数器值大于零，则将其减一并继续执行；如果等于零，则当前线程或进程被阻塞，直到计数器值变为正数。signal 操作的语义是释放资源：将计数器值加一，并唤醒一个正在等待该信号量的线程或进程。这些操作的原子性确保了在并发环境下不会出现竞态条件。

信号量有两种基本类型：二进制信号量和计数信号量。二进制信号量的计数器值只能是零或一，它常用于实现互斥锁，保护临界区，保证同一时刻只有一个线程可以访问共享资源。计数信号量的计数器值可以是任意非负整数，它用于控制对一组完全相同资源的访问，例如在数据库连接池中限制并发连接数，或者实现流量控制。

为了更直观地理解信号量，我们可以借助现实世界的类比。想象一个停车场，它代表一个计数信号量：总车位数是信号量的初始值。每进入一辆车相当于执行 wait 操作，空闲车位数减一；每离开一辆车相当于执行 signal 操作，空闲车位数加一。当车位满时（计数器为零），新车必须等待，直到有车离开。另一个例子是厕所钥匙，它代表一个二进制信号量：只有一个钥匙。一个人拿着钥匙进去相当于 wait 操作，其他人必须等待；出来时归还钥匙相当于 signal 操作，下一个人才能进去。这些类比帮助我们形象化信号量的工作原理。

21 信号量的经典应用模式

在并发编程中，信号量常用于两种基本模式：互斥和同步。互斥模式通过一个初始值为一的二进制信号量实现，确保同一时刻只有一个线程可以进入临界区。线程在进入临界区前调用 wait 操作，如果信号量值为一，则减一并进入；如果为零，则阻塞。退出临界区时调用 signal 操作，将值加一，并唤醒一个等待线程。这种模式简单有效，但需要确保 wait 和 signal 操作成对出现，否则可能导致死锁或资源泄漏。

同步模式则更复杂，它用于协调线程间的执行顺序。一个经典例子是生产者－消费者问题：生产者线程生产数据并放入共享缓冲区，消费者线程从缓冲区取出数据。缓冲区大小有限，因此需要协调生产者和消费者的速度。这里使用三个信号量：empty_slots 表示空缓冲区数量，初始值为缓冲区大小；full_slots 表示已填充缓冲区数量，初始值为零；

mutex 是一个二进制信号量，初始值为一，用于保护对缓冲区的互斥访问。生产者首先等待 empty_slots，然后获取 mutex 添加数据，最后释放 mutex 并增加 full_slots。消费者则相反，等待 full_slots，获取 mutex 取出数据，释放 mutex 并增加 empty_slots。这种模式确保了生产者和消费者之间的有序协作，避免了数据竞争。

22　动手实现：构建我们自己的信号量

现在，让我们亲手实现一个简单的信号量。我们将使用 Python 的 threading 模块，因为它提供了清晰的线程模型和同步原语。我们的目标是构建一个 MySemaphore 类，包含计数器、等待队列，并保证 wait 和 signal 操作的原子性。

设计思路如下：我们需要一个整数 value 作为计数器，一个等待队列 queue 用于存放被阻塞的线程，以及一个机制来保证操作的原子性。在 Python 中，我们可以使用 threading.Condition，它内部基于 Lock，并提供了等待和通知功能，适合实现信号量。Condition 确保了在修改共享状态时的互斥访问，并支持线程的阻塞和唤醒。

以下是 MySemaphore 类的代码实现：

```
import threading

class MySemaphore:
    def __init__(self, value=1):
        self.value = value
        self.condition = threading.Condition()

    def wait(self):
        with self.condition:
            while self.value == 0:
                self.condition.wait()
            self.value -= 1

    def signal(self):
        with self.condition:
            self.value += 1
            self.condition.notify()
```

在构造函数中，我们初始化 value 为给定值（默认为一），并创建一个 Condition 对象。wait 方法使用 with self.condition 语句获取锁，确保原子性。如果 value 大于零，则直接减一并返回；如果 value 等于零，则调用 condition.wait() 阻塞当前线程。这里使用 while 循环而非 if 语句，是为了处理伪唤醒问题：线程可能被意外唤醒，因此需要重新检查条件。signal 方法同样在锁保护下执行，它将 value 加一，并调用 condition.notify() 唤醒一个等待线程。

关键点在于，wait 方法在阻塞线程前会释放锁，这是为了避免死锁。如果不释放锁，其他线程无法执行 signal 操作来改变条件。被唤醒的线程会重新获取锁，并再次检查 value，确保在唤醒后条件仍然满足。这种实现虽然简化，但捕捉了信号量的核心行为。

为了测试我们的实现，我们可以编写一个简单的生产者－消费者程序：

```
import threading
import time

buffer = []
buffer_size = 5
empty = MySemaphore(buffer_size)
full = MySemaphore(0)
mutex = MySemaphore(1)

def producer():
    for i in range(10):
        empty.wait()
        mutex.wait()
        buffer.append(i)
        print(f"Produced␣{i}")
        mutex.signal()
        full.signal()
        time.sleep(0.1)

def consumer():
    for i in range(10):
        full.wait()
        mutex.wait()
        item = buffer.pop(0)
        print(f"Consumed␣{item}")
        mutex.signal()
        empty.signal()
        time.sleep(0.1)

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)
t1.start()
t2.start()
t1.join()
t2.join()
```

在这个测试程序中，我们定义了一个共享缓冲区 buffer，以及三个 MySemaphore 对象：empty 初始化为缓冲区大小，full 初始化为零，mutex 初始化为一来保护缓冲区。生产者线程循环生产数据，首先等待 empty 信号量（表示有空位），然后获取 mutex 锁，添加数据到缓冲区，释放 mutex，并增加 full 信号量。消费者线程类似，等待 full 信号量（表示有数据），获取 mutex，取出数据，释放 mutex，并增加 empty 信号量。通过添加延时，

我们可以观察线程间的协调行为。运行这个程序，应该能看到生产者和消费者交替执行，没有数据竞争或缓冲区溢出。

23 进阶话题与现实中的考量

尽管信号量是强大的并发工具，但它也有局限性。首先，信号量容易出错：wait 和 signal 必须成对出现，且顺序错误可能导致死锁。例如，如果一个线程在持有信号量时发生异常，可能无法释放资源。其次，信号量可能引发优先级反转问题：高优先级线程可能被低优先级线程持有的信号量所阻塞，影响系统实时性。此外，在现代并发编程中，有许多更安全、抽象的替代品，如互斥锁（Mutex）、条件变量（Condition Variable）、通道（Channel，如在 Go 语言中）或 asyncio.Semaphore（在 Python 中）。这些高级原语通常封装了信号量的复杂性，提供更直观的接口。

然而，信号量在现代编程中仍有广泛应用。例如，在限流（Rate Limiting）场景中，信号量可以控制单位时间内的请求数量；在数据库连接池中，信号量管理并发连接数；在操作系统内核中，信号量用于进程同步和资源管理。理解信号量的原理，有助于我们更好地使用这些高级工具，并在需要时实现自定义的同步机制。

信号量是并发编程领域的基石之一，它通过一个计数器及其原子操作来解决资源访问的协调问题。我们回顾了信号量的核心概念：它是一个非负整数计数器，支持 wait 和 signal 操作，有两种基本类型 —— 二进制信号量用于互斥，计数信号量用于同步。通过亲手实现一个简单的 MySemaphore 类，我们不仅理解了其内部机制，还体会了原子性、阻塞和唤醒等关键概念。尽管信号量有局限性，但它在许多场景下仍是有效的工具。掌握信号量，是迈向构建健壮、高效并发系统的重要一步。希望本文能帮助您在并发编程的旅程中更进一步。