c13n #37

c13n

2025 年 11 月 19 日

第 I 部

拓扑排序算法

李睿远

Oct 19, 2025

从依赖关系到线性序列，掌握有向无环图（DAG）的排序奥秘。拓扑排序是图论中一个基础而重要的算法，广泛应用于任务调度、依赖管理等场景。本文将系统性地介绍拓扑排序的理论基础、两种主流实现算法（Kahn 算法和基于 DFS 的算法），并通过 Python 代码示例进行详细解析，帮助读者深入理解其核心思想与应用。

在日常生活中，我们经常遇到各种依赖关系场景。例如，在大学课程选修中，学生必须完成《高等数学》后才能学习《数据结构》，而《数据结构》又是《算法分析》的先修课。这种依赖关系构成了一个典型的有向图结构。课程可以被视为「顶点」，先修关系则对应「有向边」，从先修课指向后续课。整个课程体系形成了一张有向图，而一个合理的学习计划就是这张图的一个拓扑序列。拓扑排序的目标正是将这种依赖关系转化为线性序列，确保所有前置条件得到满足。本文将深入探讨拓扑排序的原理与实现，帮助读者掌握这一关键算法。

1 前置知识：图论基础

在深入拓扑排序之前，我们需要了解一些图论基本概念。有向图是由顶点和带有方向的边组成的结构，其中每条边从一个顶点指向另一个顶点。顶点的「入度」是指向该顶点的边的数量，而「出度」是从该顶点指出的边的数量。以课程依赖为例，如果一门课程有多门先修课，其入度就较高；如果它被多门后续课程依赖，则出度较高。拓扑排序的核心前提是图必须为有向无环图（DAG），即图中不能存在任何环。如果图中存在环，例如 A 依赖 B，B 依赖 C，C 又依赖 A，依赖关系会形成死循环，无法得出满足所有依赖的线性序列。因此，拓扑排序仅适用于 DAG，环检测成为算法中的重要环节。

2 拓扑排序的核心思想与算法流程

拓扑排序的目标是生成一个顶点序列，使得对于图中任何一条有向边 $u \to v$，u 在序列中都出现在 v 之前。这确保了所有依赖关系得到遵守。接下来，我们将介绍两种主流算法：Kahn 算法和基于 DFS 的算法。

2.1 Kahn 算法（基于 BFS，贪心思想）

Kahn 算法采用广度优先搜索（BFS）的策略，其直觉是从没有任何依赖（即入度为 0）的顶点开始处理。这些顶点是天然的起点，算法逐步移除它们，并更新依赖关系。详细步骤如下：首先，初始化阶段计算每个顶点的入度，并将所有入度为 0 的顶点加入队列，同时准备一个结果列表。接着，循环处理队列中的顶点：取出一个顶点 u，加入结果列表，然后遍历 u 的所有邻接顶点 v，将 v 的入度减 1（相当于移除边 $u \to v$）。如果 v 的入度变为 0，则将 v 加入队列。最后，检查结果列表中的顶点数是否等于总顶点数；如果相等，排序成功；否则，说明图中存在环，无法完成拓扑排序。这种方法直观易理解，适合动态任务调度场景。

2.2 基于 DFS 的算法

基于深度优先搜索（DFS）的算法利用递归路径反映依赖关系。直觉是当一个顶点的所有后代都被访问完毕后，再将其加入序列，确保它出现在依赖项之后。详细步骤包括：初始化一个栈用于存储结果，并创建一个标记数组跟踪顶点状态（未访问、访问中、已访问）。然后，对每个未访问顶点执行 DFS：标记为访问中，递归访问邻接顶点。如果遇到访问中的顶点，

说明发现环，终止过程；否则，当所有邻接顶点处理完毕，标记为已访问并压入栈中。最后，将栈中元素弹出，得到拓扑序列。与 Kahn 算法相比，DFS 算法代码更简洁，但需注意环检测在递归中进行。

3 代码实现（以 Python 为例）

在代码实现中，我们使用邻接表表示图结构。邻接表以字典形式存储，键为顶点，值为该顶点指向的顶点列表。例如，图 `graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}` 表示顶点 A 指向 B 和 C，B 和 C 指向 D，D 无出边。这种表示法高效且易于遍历。

3.1 Kahn 算法实现

以下是 Kahn 算法的 Python 实现代码。我们使用 `collections.deque` 作为队列，以提高效率。

```
from collections import deque

def topological_sort_kahn(graph):
    # 初始化入度表：为每个顶点设置初始入度为 0
    in_degree = {node: 0 for node in graph}
    # 遍历图，计算每个顶点的实际入度
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    # 创建队列，将所有入度为 0 的顶点加入
    queue = deque([node for node in in_degree if in_degree[node] ==
        ↪ 0])
    topo_order = [] # 存储拓扑序列的结果列表

    # 循环处理队列中的顶点
    while queue:
        u = queue.popleft() # 取出一个入度为 0 的顶点
        topo_order.append(u) # 加入结果序列
        # 遍历 u 的所有邻接顶点 v
        for v in graph[u]:
            in_degree[v] -= 1 # 减少 v 的入度，相当于移除边 u->v
            if in_degree[v] == 0: # 如果 v 的入度变为 0，加入队列
                queue.append(v)

    # 检查是否有环：如果结果序列长度不等于顶点数，则存在环
    if len(topo_order) == len(graph):
```

```
        return topo_order
    else:
        return [] # 返回空列表表示有环，也可抛出异常

# 调用示例
graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}
print(topological_sort_kahn(graph)) # 输出可能为 ['A', 'B', 'C', 'D'] 或 ['A', 'C', 'B', 'D']
```

在这段代码中，首先初始化入度表，通过遍历图的边来填充每个顶点的入度值。然后，使用队列处理入度为 0 的顶点，逐步更新邻接顶点的入度。循环结束后，通过比较结果序列长度与顶点总数来判断是否存在环。该实现的时间复杂度为 $O(V+E)$，其中 V 是顶点数，E 是边数，因为每个顶点和边只被处理一次。

3.2 DFS 算法实现

基于 DFS 的拓扑排序算法代码如下。它使用递归和栈来管理顶点状态。

```
def topological_sort_dfs(graph):
    visited = {} # 标记字典：0 未访问，1 访问中，2 已访问
    stack = [] # 栈用于存储拓扑序列

    # 初始化所有顶点为未访问状态
    for node in graph:
        visited[node] = 0

    def dfs(node):
        if visited[node] == 1: # 如果遇到访问中的顶点，说明有环
            raise ValueError("图中存在环，无法进行拓扑排序！")
        if visited[node] == 2: # 已访问顶点，直接返回
            return
        visited[node] = 1 # 标记为访问中
        # 递归访问所有邻接顶点
        for neighbor in graph[node]:
            dfs(neighbor)
        visited[node] = 2 # 标记为已访问
        stack.append(node) # 顶点处理完毕后压入栈

    # 对每个未访问顶点执行 DFS
    for node in graph:
        if visited[node] == 0:
            dfs(node)

```

```
    return stack[::-1] # 反转栈，得到拓扑序列

# 调用示例
graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}
print(topological_sort_dfs(graph)) # 输出可能为 ['A', 'B', 'C', 'D'] 或 ['A', 'C', 'B', 'D']
```

这段代码通过 DFS 递归遍历图，使用 `visited` 字典跟踪顶点状态。在递归过程中，如果发现顶点处于访问中状态，则抛出环异常；否则，处理完所有邻接顶点后压入栈。最后，反转栈得到拓扑序列。该实现同样具有 $O(V+E)$ 的时间复杂度，但空间复杂度取决于递归深度。

4 算法分析与比较

拓扑排序的两种算法在时间复杂度和空间复杂度上相似，时间复杂度均为 $O(V+E)$，因为每个顶点和边只被访问一次。空间复杂度方面，Kahn 算法需要存储入度表和队列，通常为 $O(V)$；DFS 算法则需要递归栈空间，最坏情况下也为 $O(V)$。在选择算法时，Kahn 算法更直观，易于实现环检测，适合需要动态知道当前可执行任务的场景，如任务调度器。而 DFS 算法代码更简洁，如果需要求所有可能的拓扑序列，DFS 更容易通过回溯实现。实际应用中，应根据具体需求选择算法，例如在依赖管理系统中，Kahn 算法可能更实用。

5 实际应用场景

拓扑排序在多个领域有广泛应用。在编译过程中，它用于管理模块依赖关系，确保编译顺序正确；在任务调度与项目管理中，它帮助确定任务执行的先后顺序；软件包管理器如 apt 或 npm 利用拓扑排序解决库依赖的安装顺序；数据流系统中，它用于计算有依赖关系的单元格，例如 Excel 中的公式求值；此外，课程安排、工作流设计等场景也依赖拓扑排序来优化流程。这些应用凸显了拓扑排序在解决依赖问题中的重要性。

拓扑排序是针对有向无环图的关键算法，其核心在于将依赖关系转化为线性序列。本文介绍了 Kahn 算法和基于 DFS 的算法，两者均能高效实现拓扑排序，但各有适用场景。环检测是算法中不可忽视的环节，确保图的合理性。通过代码实现和详细解读，读者可以动手实践，进一步解决实际问题，如 LeetCode 上的课程表题目。掌握拓扑排序不仅提升算法能力，还为处理复杂依赖关系提供实用工具。

第 II 部

信号量（Semaphore）机制

杨岢瑞

Oct 20, 2025

在现代计算系统中，多进程或多线程并发执行已成为常态，但这种并发性带来了一个核心挑战：如何协调多个执行流对共享资源的访问。想象一个经典的生产者-消费者场景：有一个固定大小的数据缓冲区，生产者线程不断向缓冲区放入数据，而消费者线程则从中取走数据。当缓冲区已满时，生产者如果继续写入会导致数据覆盖；当缓冲区为空时，消费者如果尝试读取则会获取无效数据。这种问题不仅限于数据缓冲区，还延伸到打印机、文件句柄、数据库连接等任何共享资源。正是这些场景催生了对同步机制的需求，而信号量便是其中一种优雅而强大的解决方案。本文将带领读者从信号量的哲学思想出发，逐步深入其实现细节，最终通过代码实践掌握这一并发编程的基石工具。

6　信号量的核心思想与定义

信号量由荷兰计算机科学家艾兹格·迪杰斯特拉在 1960 年代提出，他是并发编程领域的先驱之一。信号量的核心隐喻是一个具备原子操作的计数器，这个简单的抽象却蕴含着解决复杂同步问题的巨大能量。信号量本质上是一个整型变量，配合两个不可分割的操作，用于控制对共享资源的访问。

信号量主要分为两种类型。二进制信号量，其值仅限于 0 或 1，常用于实现互斥锁，确保同一时刻只有一个线程能进入临界区。计数信号量则允许取值为非负整数，用于管理一组数量有限的资源，如连接池中的连接数或停车场空位。信号量的威力源于其两个基本操作：P 操作和 V 操作。P 操作源自荷兰语 Proberen，意为测试；V 操作源自 Verhogen，意为增加。

P 操作的逻辑是尝试获取资源：首先检查信号量值，如果大于零则原子性地减一并继续执行；否则线程等待直到条件满足。用伪代码表示如下：

```
function P(semaphore S):
    while S <= 0:
        ; // 忙等待或进入阻塞
    S = S - 1 // 原子地减少计数
```

这段代码中，while 循环实现了等待机制，当资源不足时线程会持续检查或进入睡眠状态。关键点在于检查信号量值和减少计数的操作必须是原子的，即不可被中断，否则可能导致竞态条件。

V 操作则用于释放资源：原子地将信号量值加一，如果有线程正在等待，则唤醒其中一个。伪代码如下：

```
function V(semaphore S):
    S = S + 1 // 原子地增加计数
    // 唤醒一个等待线程
```

这两个操作的原子性是信号量正确工作的基石。在单处理器系统中，原子性可以通过禁用中断实现；在多处理器环境中，则需要硬件提供的原子指令支持。

7 信号量的应用场景

信号量的应用广泛而深入，理解其典型使用场景有助于更好地掌握这一工具。首先考虑实现互斥锁的场景。通过初始化一个二进制信号量为 1，我们可以使用 P/V 操作保护临界区。线程在进入临界区前执行 P 操作，如果信号量值为 1 则减为 0 并进入；如果为 0 则等待。退出临界区时执行 V 操作，将值恢复为 1 并唤醒可能等待的线程。这种机制确保了同一时刻只有一个线程能执行临界区代码。

生产者-消费者问题是信号量的经典应用。假设有一个大小为 N 的缓冲区，我们需要三个信号量：empty 信号量初始化为 N，表示空位数量；full 信号量初始化为 0，表示已存放数据数量；mutex 信号量初始化为 1，用于保护对缓冲区的互斥访问。生产者线程的核心逻辑是：先执行 P(empty) 检查是否有空位，然后执行 P(mutex) 获取缓冲区访问权，放入数据后执行 V(mutex) 释放访问权，最后执行 V(full) 通知消费者有新数据。消费者线程则相反：先执行 P(full) 检查是否有数据，然后执行 P(mutex) 获取访问权，取走数据后执行 V(mutex) 释放访问权，最后执行 V(empty) 通知生产者有空位。这里的操作顺序至关重要，错误的顺序可能导致死锁。

另一个常见场景是控制并发线程数量。例如，一个数据库连接池最多允许 10 个连接，我们可以初始化一个计数信号量为 10。每个线程在获取连接前执行 P 操作，如果信号量值大于 0 则减一并继续；否则等待。释放连接后执行 V 操作，增加信号量值并唤醒等待线程。这种模式可以轻松扩展到任何资源池的管理。

8 动手实现：从零构建一个用户态信号量

理解了信号量的理论和应用后，我们将进入最富挑战性的部分：亲手实现一个用户态信号量。在用户态实现信号量的核心挑战在于如何保证 P/V 操作的原子性，因为我们无法直接使用硬件指令，但可以利用操作系统提供的原子操作或系统调用。

我们将探讨两种实现方案。第一种基于互斥锁和条件变量，这是最常用且易于理解的方式。我们定义信号量数据结构如下：

```
typedef struct my_semaphore {
   int value; // 信号量的计数值
   pthread_mutex_t mutex; // 保护 value 的互斥锁
   pthread_cond_t cond; // 用于线程等待和唤醒的条件变量
} my_sem_t;
```

这个结构体包含三个成员：value 存储信号量的当前值；mutex 是一个互斥锁，用于保护对 value 的并发访问；cond 是一个条件变量，用于实现线程的等待和唤醒机制。这种设计利用了 POSIX 线程库提供的基础设施，具有良好的可移植性。

接下来实现信号量的初始化函数：

```
int my_sem_init(my_sem_t *sem, int value) {
   sem->value = value;
   pthread_mutex_init(&sem->mutex, NULL);
   pthread_cond_init(&sem->cond, NULL);
```

```
    return 0;
}
```

这个函数接收一个信号量指针和初始值，初始化 value 字段，并设置互斥锁和条件变量。互斥锁用于确保对 value 的访问是互斥的，条件变量用于线程间的通信。

P 操作的实现如下：

```
void my_sem_wait(my_sem_t *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->value <= 0) {
        pthread_cond_wait(&sem->cond, &sem->mutex);
    }
    sem->value--;
    pthread_mutex_unlock(&sem->mutex);
}
```

这段代码首先获取互斥锁以确保原子性。然后使用 while 循环检查信号量值，如果值小于等于 0，则调用 pthread_cond_wait 使线程等待。这里必须使用 while 而不是 if，因为可能存在虚假唤醒 —— 线程可能在没有明确信号的情况下被唤醒，需要重新检查条件。当信号量值大于 0 时，线程减少 value 并释放互斥锁。条件变量等待时会自动释放互斥锁，允许其他线程操作信号量，被唤醒时会重新获取互斥锁。

V 操作的实现：

```
void my_sem_post(my_sem_t *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->value++;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->mutex);
}
```

这个函数首先获取互斥锁，增加信号量值，然后通过 pthread_cond_signal 唤醒一个等待的线程，最后释放互斥锁。如果有多个线程等待，唤醒哪一个取决于调度策略。

这种实现方案的优点是简单易懂，可移植性好，但涉及线程上下文切换，在高性能场景下可能不够高效。对于追求极致性能的应用，我们可以考虑第二种方案：基于原子操作和 Futex 的实现。

Futex 是 Linux 特有的快速用户态互斥锁机制，核心思想是在用户态进行竞态检查，仅在必要时陷入内核。我们利用 GCC 的原子操作内置函数实现信号量：

```
typedef struct futex_semaphore {
    int value;
} futex_sem_t;
```

P 操作实现：

```
void futex_sem_wait(futex_sem_t *sem) {
    int old_val;
```

```
    while (1) {
        old_val = __atomic_load_n(&sem->value, __ATOMIC_ACQUIRE);
        if (old_val > 0) {
            if (__atomic_compare_exchange_n(&sem->value, &old_val,
                ↪ old_val - 1,
                                false, __ATOMIC_ACQ_REL,
                                    ↪ __ATOMIC_ACQUIRE)) {
                break;
            }
        } else {
            syscall(SYS_futex, &sem->value, FUTEX_WAIT, old_val, NULL,
                ↪ NULL, 0);
        }
    }
}
```

这段代码使用 __atomic_load_n 原子加载当前值，如果值大于 0，则尝试通过 __atomic_compare_exchange_n 原子比较交换操作减少计数值。如果成功则退出循环，否则重试。如果值小于等于 0，则通过 futex 系统调用让线程睡眠。Futex 系统调用会检查值是否改变，如果改变则返回，避免不必要的睡眠。

V 操作实现：

```
void futex_sem_post(futex_sem_t *sem) {
    __atomic_fetch_add(&sem->value, 1, __ATOMIC_ACQ_REL);
    if (__atomic_load_n(&sem->value, __ATOMIC_ACQUIRE) <= 0) {
        syscall(SYS_futex, &sem->value, FUTEX_WAKE, 1, NULL, NULL, 0);
    }
}
```

这里使用 __atomic_fetch_add 原子增加信号量值，然后检查是否有线程在等待（值小于等于 0），如果有则通过 futex 系统调用唤醒一个线程。这种实现极大减少了内核态与用户态的切换，性能优异，是 Linux 内核和 glibc 中信号量的实现方式，但代价是复杂度和平台依赖性。

信号量作为并发编程的基石，通过简单的计数器模型和两个原子操作，优雅地解决了互斥与同步问题。我们从迪杰斯特拉的原始思想出发，探讨了信号量的类型和应用场景，最终深入实现细节，用两种不同层次的方案构建了用户态信号量。基于互斥锁和条件变量的实现简单易懂，适合大多数场景；基于原子操作和 Futex 的实现性能卓越，适合高性能需求。

然而，信号量并非完美无缺。其低级特性使得编程容易出错，错误的 P/V 操作顺序可能导致死锁；缺乏高级抽象使得代码难以维护。现代并发编程已经发展出更高级的工具，如 Java 的 java.util.concurrent 包、C++ 的 std::async、Go 的 channel 和 goroutine 等，这些工具在信号量基础上提供了更安全、更易用的抽象。

尽管如此，深入理解信号量这样的底层机制仍然至关重要。它不仅帮助我们诊断复杂的并发问题，还为编写高性能系统代码奠定基础。信号量的思想已经渗透到各种并发工具中，掌握

它相当于获得了理解现代并发编程的钥匙。在日益并发的计算世界中，这一古老而强大的工具依然闪耀着智慧的光芒。

第 III 部

Merkle 树数据结构

王思成

Oct 21, 2025

在当今数据驱动的世界中，如何高效地验证一个超大型文件（例如整个区块链账本或分布式数据库）是否被篡改，成为一个关键问题。逐个字节比对显然不现实，这不仅耗时巨大，还难以在分布式环境中协调。Merkle 树（又称哈希树）作为一种优雅的解决方案，通过树形结构和密码学哈希函数，将大量数据的「指纹」浓缩成一个单一的根哈希，从而实现高效、安全的完整性验证。这种数据结构在比特币、以太坊、Git 和 IPFS 等知名系统中扮演着基石角色，确保了数据的可信度和一致性。本文将深入解析 Merkle 树的原理，并带领读者从零开始，使用 Python 语言实现一个基本的 Merkle 树，涵盖从核心概念到代码实现的完整流程。

9 第一部分：Merkle 树的核心概念解析

9.1 1.1 什么是 Merkle 树?

Merkle 树可以类比为一个家族谱系或组织机构图，顶层代表整体，而底层代表个体成员。在技术上，它是一种树形数据结构，其中每个叶子节点对应数据块的哈希值，每个非叶子节点则是其子节点哈希值拼接后的哈希值。这种结构允许我们通过根节点（即 Merkle 根）来代表整个数据集的完整性，任何底层数据的改动都会通过哈希链反应到根节点，从而确保验证的高效性。例如，在比特币中，Merkle 树用于将所有交易哈希汇总成一个根哈希，嵌入区块头中，简化了轻量级客户端的验证过程。

9.2 1.2 核心构件：哈希函数

哈希函数是 Merkle 树的基石，它负责将任意长度的数据映射为固定长度的哈希值。一个理想的加密哈希函数（如 SHA-256）具备确定性（相同输入总是产生相同输出）、高效性（计算速度快）、抗碰撞性（难以找到两个不同输入产生相同输出）和雪崩效应（输入的微小变化导致输出的巨大变化）。这些特性确保了 Merkle 树的防篡改能力；例如，如果我们将哈希函数表示为 $H(x)$，其中 x 是输入数据，那么 Merkle 树的构建就依赖于递归应用 H 到子节点拼接的结果上。

9.3 1.3 Merkle 树的构建过程

构建一棵 Merkle 树的过程可以分为几个清晰的步骤。首先，将原始数据分割成若干个数据块，例如一个交易列表被分成多个独立单元。接着，对每个数据块进行哈希计算，生成叶子节点。然后，将相邻的两个叶子节点拼接后再次哈希，得到它们的父节点。这一过程递归进行，直到只剩下一个节点，即 Merkle 根。举例来说，如果有四个数据块 TX0、TX1、TX2 和 TX3，我们会先计算叶子节点 $H(TX0)$、$H(TX1)$、$H(TX2)$ 和 $H(TX3)$，然后拼接并哈希相邻叶子节点得到中间节点，例如 $H(H(TX0)||H(TX1))$，最终递归生成根节点。这种分层结构不仅压缩了数据表示，还支持高效的局部验证。

9.4 1.4 关键特性与优势

Merkle 树的核心优势在于其固定大小的根哈希和高效验证机制。无论底层数据量多大，Merkle 根始终是一个固定长度的哈希值（例如 SHA-256 生成 256 位哈希），这简化了存

储和传输。更重要的是，它支持高效验证单个数据块的完整性：通过提供从该数据块到根的路径（即 Merkle 路径），验证者只需计算路径上的哈希值，而无需处理整个数据集。防篡改特性则源于哈希函数的雪崩效应；任何数据块的微小变动都会导致根哈希的显著变化，从而立即暴露篡改行为。

10 第二部分：动手实现一个基本的 Merkle 树

10.1 2.1 环境与工具准备

为了从零实现 Merkle 树，我们选择 Python 作为编程语言，因为它简洁易读，适合教学和原型开发。必要的工具包括 Python 3.x 环境和标准库中的 hashlib 模块，该模块提供了 SHA-256 等加密哈希函数，无需额外安装。在代码中，我们将导入 hashlib 来执行哈希计算，确保实现的可靠性和一致性。

10.2 2.2 数据结构设计

我们首先定义 Merkle 树的基本数据结构。使用一个简单的 MerkleNode 类来表示树中的每个节点，该类包含 left、right 和 data 属性，其中 left 和 right 指向子节点，data 存储该节点的哈希值。然后，我们定义 MerkleTree 类，它管理整个树结构，核心属性包括 root（根节点）和 leaves（叶子节点列表）。这种设计允许我们灵活地构建和遍历树，同时保持代码的模块化。

```
import hashlib

class MerkleNode:
    def __init__(self, left=None, right=None, data=None):
        self.left = left
        self.right = right
        self.data = data

class MerkleTree:
    def __init__(self, data_list):
        self.leaves = []
        self.root = None
        self.build_tree(data_list)
```

这段代码定义了 MerkleNode 和 MerkleTree 类。MerkleNode 的构造函数初始化左右子节点和哈希数据，而 MerkleTree 的构造函数接受一个数据列表，并调用 build_tree 方法来构建整个树结构。通过将节点和数据分离，我们实现了清晰的责任划分，便于后续方法的实现。

10.3 2.3 核心方法实现

接下来，我们实现 Merkle 树的核心方法，包括哈希函数封装、叶子节点构建、树构建和根哈希获取。首先，_hash 方法负责处理字符串拼接和哈希计算，它使用 hashlib.sha256 生成哈希值，并返回十六进制字符串表示。

```
def _hash(self, data):
    if isinstance(data, str):
        data = data.encode('utf-8')
    return hashlib.sha256(data).hexdigest()
```

_hash 方法首先检查输入数据是否为字符串，如果是，则编码为字节，然后使用 SHA-256 计算哈希并返回十六进制形式。这确保了方法能处理各种输入类型，同时保持哈希的确定性和一致性。

然后，_build_leaves 方法将输入数据列表转换为叶子节点列表。它遍历每个数据项，计算其哈希值，并创建对应的 MerkleNode 实例。

```
def _build_leaves(self, data_list):
    self.leaves = [MerkleNode(data=self._hash(item)) for item in
        ↪ data_list]
```

_build_leaves 方法使用列表推导式高效地生成叶子节点，每个节点的 data 属性存储对应数据项的哈希值。这为树构建奠定了基础，确保所有叶子节点预先计算完成。

build_tree 方法是核心逻辑，它自底向上地构建整个 Merkle 树。该方法处理叶子节点列表，通过循环拼接相邻节点并计算父节点哈希，直到生成根节点。对于奇数个节点的情况，它会复制最后一个节点以保持平衡。

```
def build_tree(self, data_list):
    self._build_leaves(data_list)
    nodes = self.leaves[:]
    while len(nodes) > 1:
        new_level = []
        for i in range(0, len(nodes), 2):
            left = nodes[i]
            right = nodes[i+1] if i+1 < len(nodes) else nodes[i]
            combined = left.data + right.data
            parent = MerkleNode(left=left, right=right, data=self._hash(
                ↪ combined))
            new_level.append(parent)
        nodes = new_level
    self.root = nodes[0] if nodes else None
```

build_tree 方法首先调用 _build_leaves 初始化叶子节点，然后使用一个循环逐步构建上层节点。在每次迭代中，它处理节点列表中的每对节点，拼接它们的哈希值并计算父节点

哈希。如果节点数为奇数，则复制最后一个节点以确保配对。最终，当节点列表缩减为一个元素时，将其设置为根节点。这个过程体现了 Merkle 树的递归本质，同时通过迭代实现提高了效率。

get_root 方法简单返回根节点的哈希值，提供对整体数据完整性的访问点。

```
def get_root(self):
    return self.root.data if self.root else None
```

get_root 方法检查根节点是否存在，并返回其 data 属性，即 Merkle 根哈希。这允许用户快速获取树的整体指纹，用于验证或存储。

10.4 2.4 实现完整性验证：Merkle 证明

Merkle 证明是验证单个数据块完整性的关键机制，它涉及生成从该数据块到根的路径（Merkle 路径），并利用路径上的节点哈希进行验证。generate_proof 方法根据数据块索引生成其 Merkle 路径，路径包括从叶子节点到根过程中所需的所有兄弟节点哈希和方向信息。

```
def generate_proof(self, index):
    if index < 0 or index >= len(self.leaves):
        return None
    proof = []
    node = self.leaves[index]
    current_index = index
    nodes = self.leaves[:]
    while len(nodes) > 1:
        new_level = []
        for i in range(0, len(nodes), 2):
            left = nodes[i]
            right = nodes[i+1] if i+1 < len(nodes) else nodes[i]
            if current_index == i:
                proof.append(('right', right.data))
            elif current_index == i+1:
                proof.append(('left', left.data))
            parent = MerkleNode(left=left, right=right, data=self._hash(
                ↪ left.data + right.data))
            new_level.append(parent)
        current_index //= 2
        nodes = new_level
    return proof
```

generate_proof 方法首先检查索引的有效性，然后初始化一个空证明列表。它从指定索引的叶子节点开始，向上遍历树结构，在每一层记录兄弟节点的哈希和方向（左或右）。通过更新当前索引和节点列表，方法逐步构建路径，直到根节点。这确保了证明包含验证所需

的所有信息，而不暴露整个树结构。

verify_proof 静态方法则根据数据、Merkle 路径和根哈希，独立验证数据完整性。它从数据哈希开始，依次应用路径中的兄弟节点哈希，计算最终哈希并与根哈希比对。

```
@staticmethod
def verify_proof(data, proof, root_hash):
    current_hash = hashlib.sha256(data.encode('utf-8') if isinstance(
        ↪ data, str) else data).hexdigest()
    for direction, sibling_hash in proof:
        if direction == 'left':
            combined = sibling_hash + current_hash
        else:
            combined = current_hash + sibling_hash
        current_hash = hashlib.sha256(combined.encode('utf-8') if
            ↪ isinstance(combined, str) else combined).hexdigest()
    return current_hash == root_hash
```

verify_proof 方法首先计算输入数据的哈希，然后遍历证明路径，根据方向（左或右）拼接当前哈希与兄弟节点哈希，并递归计算新哈希。最终，它将结果与提供的根哈希比较，返回布尔值表示验证是否成功。这种方法体现了 Merkle 树的高效性，验证过程仅需对数级计算量，而非线性扫描整个数据集。

11 第三部分：深入探讨与进阶话题

11.1 3.1 可视化演示

为了直观展示 Merkle 树的构建过程，我们可以通过一个简单示例来打印树结构。假设输入数据为 [data1, data2, data3, data4]，我们首先计算每个数据块的哈希值作为叶子节点，然后递归构建中间节点和根节点。例如，叶子节点哈希可能为 H1、H2、H3 和 H4（其中 H 表示 SHA-256 哈希），中间节点为 H(H1||H2) 和 H(H3||H4)，最终根节点为 H(H(H1||H2) || H(H3||H4))。通过代码输出各级节点的哈希值，读者可以清晰看到数据如何从底层聚合到顶层，根哈希作为整体指纹的表示。

11.2 3.2 测试与边界情况处理

在实际应用中，测试 Merkle 树的边界情况至关重要，例如处理空数据、单个数据块或奇数个数据块。对于空数据，树构建应返回 None 或空根，避免运行时错误。单个数据块时，根节点直接等于该数据块的哈希。奇数个数据块则通过复制最后一个节点来平衡树结构，确保构建过程的正确性。此外，我们可以演示篡改场景：修改一个数据块后，根哈希会发生显著变化，且验证证明会失败，这突出了 Merkle 树的防篡改特性。通过系统测试，我们能够验证实现的健壮性和可靠性。

11.3 3.3 不同变体简介

Merkle 树有多种变体，适应不同应用场景。比特币中的 Merkle 树用于验证交易完整性，轻量级节点只需下载区块头和 Merkle 路径，即可确认交易是否包含在区块中，大大减少了带宽需求。以太坊则使用 Merkle Patricia 树，这是一种结合了 Merkle 树和 Patricia 树（一种压缩字典树）的高级数据结构，用于高效存储和验证键值对状态。这些变体扩展了基本 Merkle 树的功能，支持更复杂的查询和更新操作，体现了其在分布式系统中的灵活性和强大潜力。

回顾全文，Merkle 树的核心思想在于通过树形哈希结构，将大量数据的完整性验证问题简化为对单个根哈希的信任问题。其高效性体现在验证过程仅需对数级计算，而安全性则依赖于密码学哈希函数的抗篡改特性。从理论解析到代码实现，我们展示了如何构建一个基本 Merkle 树，并实现完整性验证，这为理解分布式系统数据一致性提供了坚实基础。

11.4 4.2 广阔的应用场景

Merkle 树在多个领域发挥着关键作用。在区块链中，它支持比特币的 SPV（简单支付验证）节点，使轻量级客户端能够高效验证交易。在版本控制系统如 Git 中，Merkle 树用于管理代码提交的完整性，确保历史记录不可篡改。分布式文件系统如 IPFS 利用它来验证文件块在点对点网络中的正确性。此外，数据库系统使用 Merkle 树进行数据同步和一致性检查，这些应用凸显了其作为现代数据基础设施核心组件的价值。

11.5 4.3 下一步学习方向

为了进一步深入学习，读者可以阅读比特币白皮书或以太坊黄皮书，深入了解 Merkle 树在真实系统中的应用细节。尝试实现更复杂的变体，如 Merkle Patricia 树，可以提升对高级数据结构的理解。此外，探索在自定义分布式项目中的应用，例如构建一个简单的区块链或文件存储系统，将理论知识与实践相结合，深化对数据完整性和分布式信任机制的认识。

本文的完整代码已托管在 GitHub 仓库中，读者可以访问并运行示例，进一步实验和扩展。如果您在实现过程中遇到问题或有创新想法，欢迎在评论区分享和讨论，共同探索 Merkle 树的无限可能。

第 IV 部

哈希表数据结构

黄梓淳

Oct 22, 2025

12 导言

在日常生活中，我们经常遇到需要快速查找信息的场景，例如使用字典查询单词释义、翻阅电话簿查找联系人号码，或者通过图书馆索引定位书籍位置。这些场景都基于一个共同的概念——「键-值」对，其中每个键都唯一对应一个值。如果使用简单的数组来存储这些键值对，查找特定元素时可能需要遍历整个数组，导致时间复杂度高达 $O(n)$，当数据量巨大时，这种线性搜索的效率将变得极低。哈希表（Hash Table）作为一种高效的数据结构，能够在平均 $O(1)$ 时间复杂度下完成插入、删除和查找操作，极大地提升了数据处理的性能。本文将深入解析哈希表的核心工作原理，并逐步指导读者使用 Python 语言实现一个基本的哈希表，帮助大家从理论过渡到实践。

13 第一部分：哈希表的核心思想与工作原理

13.1 1.1 什么是哈希表?

哈希表是一种基于哈希函数的数据结构，它通过将键（Key）映射到数组中的特定位置来存储和访问对应的值（Value）。这种设计巧妙地利用了「空间换时间」的策略，将查找操作的平均时间复杂度从 $O(n)$ 降低到 $O(1)$。具体来说，哈希表的核心在于使用一个数组作为底层存储，并通过哈希函数将任意键转换为固定范围内的整数索引，从而直接定位到数组中的位置进行数据操作。

13.2 1.2 哈希函数

哈希函数在哈希表中扮演着关键角色，它的主要作用是将任意大小的输入（即键）转换为一个固定范围的整数值，这个值作为数组的索引。一个理想的哈希函数应具备三个重要特性：确定性、高效性和均匀分布性。确定性确保相同的键总是产生相同的哈希值；高效性要求计算过程快速，不影响整体性能；均匀分布性则保证键的哈希值尽可能均匀地分布在数组空间中，从而最小化冲突的发生。例如，对于字符串键，常用的哈希函数可能基于字符的 ASCII 值进行加权求和，再对数组容量取模。

13.3 1.3 哈希冲突

哈希冲突是指两个或多个不同的键经过哈希函数计算后，得到相同的数组索引。根据鸽巢原理，由于哈希函数的输出范围有限，而输入键的数量可能无限，冲突是不可避免的。解决哈希冲突的方法有多种，其中链地址法和开放定址法是最常见的两种。本文将重点介绍链地址法的实现，因为它简单易懂且能有效处理冲突。

14 第二部分：解决哈希冲突的关键技术

14.1 2.1 链地址法

链地址法的核心思想是将哈希表的每个数组位置（称为「桶」）视为一个链表的头节点。当多个键映射到同一个索引时，它们会被存储在同一个链表中。插入操作时，先计算键的哈希

值找到对应桶，然后遍历链表：如果键已存在，则更新其值；否则，在链表末尾添加新节点。查找操作类似，通过哈希值定位桶后遍历链表比较键。删除操作则需要遍历链表并移除对应节点。链地址法的优点是实现简单，能有效处理冲突；缺点是需要额外空间存储指针，且如果链表过长，性能可能退化为 $O(n)$。

14.2 2.2 开放定址法

开放定址法是另一种解决冲突的方法，它要求所有元素都存储在数组本身中。当发生冲突时，系统会按照预定义的探测序列（如线性探测、二次探测或双重哈希）在数组中寻找下一个空闲位置。例如，线性探测会依次检查索引 $i+1$、$i+2$ 等，直到找到空位。与链地址法相比，开放定址法节省了指针空间，但可能面临聚集问题，影响性能。本文主要聚焦链地址法的实现，但了解开放定址法有助于拓宽对哈希表设计的认识。

15 第三部分：动手实现一个哈希表（以链地址法为例）

15.1 3.1 设计数据结构

在实现哈希表时，我们首先需要定义其底层数据结构。一个基本的哈希表类（例如 MyHashMap）应包含以下属性：size 表示当前键值对的数量；capacity 指定底层数组的容量；load_factor 为负载因子（默认 0.75），用于触发扩容；buckets 是一个数组，每个元素是一个链表的头节点，用于存储键值对。负载因子的计算公式为 $\frac{size}{capacity}$，当超过阈值时，哈希表会自动扩容以维持性能。

15.2 3.2 定义键值对节点类

为了在链表中存储键值对，我们需要定义一个节点类（例如 Node）。这个类包含三个属性：key 存储键，value 存储值，next 指向下一个节点的指针。在 Python 中，我们可以用一个简单的类来实现。

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

这段代码定义了一个 Node 类，初始化方法 __init__ 接收 key 和 value 参数，并将 next 指针设为 None，表示链表末尾。每个节点都封装了一个键值对，并通过 next 指针连接成链表，从而支持链地址法的冲突解决。

15.3 3.3 实现核心方法

接下来，我们实现哈希表的核心方法，包括初始化、哈希函数、插入、查找、删除和扩容。首先，定义 MyHashMap 类的构造函数 __init__。它初始化哈希表的容量、负载因子和桶数组。默认初始容量设为 16，每个桶初始化为空链表。

```
class MyHashMap:
    def __init__(self, initial_capacity=16, load_factor=0.75):
        self.capacity = initial_capacity
        self.load_factor = load_factor
        self.size = 0
        self.buckets = [None] * self.capacity
```

这段代码中，buckets 被初始化为一个长度为 capacity 的数组，每个元素为 None，表示空链表。size 记录当前元素数量，load_factor 用于后续扩容判断。

私有方法 _hash 负责计算键的哈希值。它使用 Python 内置的 hash() 函数，并对容量取模以确保索引在有效范围内。同时，处理负哈希值的情况。

```
    def _hash(self, key):
        if key is None:
            raise ValueError("Key cannot be None")
        return hash(key) % self.capacity
```

这里，hash(key) 生成一个整数哈希值，通过取模操作 % self.capacity 将其映射到 0 到 capacity-1 的范围内。如果键为 None，则抛出异常，避免无效输入。

put 方法用于插入或更新键值对。它先计算哈希索引，然后遍历对应链表：如果键存在，则更新值；否则，在链表末尾添加新节点。完成后，检查负载因子，必要时触发扩容。

```
    def put(self, key, value):
        index = self._hash(key)
        if self.buckets[index] is None:
            self.buckets[index] = Node(key, value)
        else:
            current = self.buckets[index]
            while current:
                if current.key == key:
                    current.value = value
                    return
                if current.next is None:
                    break
                current = current.next
            current.next = Node(key, value)
        self.size += 1
        if self.size / self.capacity > self.load_factor:
            self._resize()
```

在 put 方法中，如果桶为空，则直接创建新节点作为头节点；否则，遍历链表查找键。如果找到，更新值；如果未找到，在末尾添加新节点。最后，增加 size 并检查是否需要扩容，这有助于保持哈希表的高效性。

get 方法根据键查找值。它计算索引后遍历链表，返回匹配的值或 None。

```
    def get(self, key):
        index = self._hash(key)
        current = self.buckets[index]
        while current:
            if current.key == key:
                return current.value
            current = current.next
        return None
```

这段代码通过哈希索引定位桶，然后线性搜索链表。如果找到键，返回对应值；否则返回None，表示键不存在。

remove 方法删除指定键的节点。它使用双指针技巧（prev 和 curr）遍历链表，便于删除操作。

```
    def remove(self, key):
        index = self._hash(key)
        current = self.buckets[index]
        prev = None
        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.buckets[index] = current.next
                self.size -= 1
                return
            prev = current
            current = current.next
```

在 remove 方法中，prev 指针记录前一个节点，curr 指向当前节点。如果找到键，则调整指针跳过该节点，实现删除。这确保了链表结构的正确性。

最后，_resize 方法负责动态扩容。当负载因子超标时，它创建一个新数组（容量翻倍），并将所有元素重新哈希到新数组中。

```
    def _resize(self):
        old_buckets = self.buckets
        self.capacity *= 2
        self.buckets = [None] * self.capacity
        self.size = 0
        for bucket in old_buckets:
            current = bucket
            while current:
                self.put(current.key, current.value)
```

```
            current = current.next
```

在 _resize 方法中，旧桶数组被保存，新数组容量翻倍。然后遍历旧数组中的每个节点，使用 put 方法重新插入到新数组中。这个过程称为「重哈希」，它通过减少链表长度来提升性能。

16 第四部分：复杂度分析与优化探讨

16.1 4.1 时间复杂度分析

在理想情况下，哈希表的插入、删除和查找操作的平均时间复杂度为 $O(1)$，这依赖于哈希函数的均匀分布性。然而，最坏情况下（例如所有键都冲突），性能可能退化为 $O(n)$，因为需要遍历长链表。因此，设计高质量的哈希函数至关重要。

16.2 4.2 空间复杂度

哈希表的空间复杂度为 $O(n+m)$，其中 n 是元素数量，m 是数组容量。这包括了存储元素和桶数组的空间。通过调整负载因子，可以在时间和空间之间取得平衡。

16.3 4.3 优化方向

为了进一步提升哈希表性能，可以考虑以下优化策略：选择合适的初始容量和负载因子，以减少扩容频率；设计更均匀的哈希函数，例如使用多项式滚动哈希；优化动态扩容策略，例如逐步扩容；以及在链表过长时转换为更高效的数据结构（如红黑树），正如 JDK 8 中的 HashMap 实现。

本文详细介绍了哈希表的核心思想，包括哈希函数的作用和冲突解决机制，并以链地址法为例实现了基本的哈希表。通过代码实现和复杂度分析，我们看到了哈希表在平均 $O(1)$ 时间复杂度下的高效性。哈希表在实际应用中广泛存在，例如 Python 的 dict、Java 的 HashMap，以及数据库索引和缓存系统。对于进阶学习，建议读者阅读标准库源码，了解并发哈希表（如 ConcurrentHashMap）和其他哈希算法，以深化对数据结构的理解。

第 V 部

CRDT（无冲突复制数据类型）数据结构

马浩琨

Oct 23, 2025

在当今分布式系统日益普及的背景下，数据一致性成为开发者必须面对的挑战。尤其是在多用户协作场景中，如在线文档编辑，如何确保所有副本最终一致，而不依赖复杂的冲突解决机制？CRDT（Conflict-free Replicated Data Types）作为一种优雅的解决方案，通过数学原理和巧妙的设计，实现了强最终一致性。本文将带你从理论到实践，深入解析 CRDT 的核心思想，并亲手实现两个基础类型：G-Counter 和 PN-Counter。

想象一个多人在线协作文档的场景，比如 Notion 或 Google Docs。当两个用户同时在文档的同一位置插入不同内容时，系统如何保证所有用户的视图最终一致？传统方法如悲观锁会破坏用户体验，无法支持离线工作；乐观锁则将冲突解决的负担抛给用户，导致操作中断；中央仲裁器则存在单点故障风险。CRDT 通过设计可交换、关联和幂等的操作，使得无论操作顺序如何，所有副本都能自动收敛到相同状态，无需人工干预。这不仅提升了系统的可用性，还降低了开发复杂度。

17 CRDT 的核心思想：数学的优雅

CRDT 的核心在于其数学基础，特别是半格理论。首先，我们需要区分状态复制和操作复制。状态复制直接同步完整数据状态，简单但数据量大；操作复制同步导致状态变化的操作，高效但需处理顺序问题。CRDT 属于操作复制范畴，但通过设计消除了对顺序的依赖。

半格是一种数学结构，定义在偏序集上，能够计算任意两个元素的上确界。在 CRDT 中，状态被组织成半格，合并函数即为计算上确界的操作。例如，求一组数字的最大值，无论比较顺序如何，结果总是相同。这保证了收敛性。CRDT 分为基于状态和基于操作两大流派。基于状态的 CRDT 直接同步状态，合并函数计算上确界，可靠但不依赖可靠传输，缺点是状态可能膨胀；基于操作的 CRDT 同步操作，要求操作满足交换律、结合律和幂等律，传输数据量小，但需可靠广播。

18 实战：实现一个 G-Counter（增长计数器）

G-Counter 适用于分布式计数器场景，每个副本只能增加自身计数，但需获取全局总和。其数据结构不使用单一整数，而是采用字典，键为副本 ID，值为该副本的计数值。例如，状态可能表示为 `{ replica_A: 5, replica_B: 3, replica_C: 7 }`。

操作设计包括 increment 和 value 方法。increment 方法只增加当前副本自身的计数。以下是用 JavaScript 伪代码实现的示例：

```
function increment() {
    this.state[this.replicaId] += 1;
}
```

在这段代码中，`this.state` 是存储计数的字典，`this.replicaId` 是当前副本的标识符。increment 操作通过简单递增当前副本的计数值，确保每个副本只修改自身部分，从而避免冲突。这种设计的关键在于，操作是局部的，不会影响其他副本。

value 方法用于计算所有副本值的总和：

```
function value() {
    return Object.values(this.state).reduce((sum, count) => sum +
        ↪ count, 0);
```

```
}
```

这里，Object.values(this.state) 获取所有计数值，reduce 方法对它们求和，初始值为 0。即使状态为空，也能正确返回结果。这个方法体现了全局视图的生成，而不需要同步所有操作。

合并策略是 CRDT 的精华，它基于半格思想。合并函数 merge 接收另一个状态，并对每个副本的计数取最大值：

```
function merge(otherState) {
   const mergedState = {};
   const allReplicaIds = new Set([...Object.keys(this.state), ...
       ↪ Object.keys(otherState)]);
   for (let id of allReplicaIds) {
      mergedState[id] = Math.max(this.state[id] || 0, otherState[id]
          ↪ || 0);
   }
   this.state = mergedState;
}
```

在这个函数中，首先创建新对象 mergedState，然后获取所有涉及的副本 ID（包括当前状态和另一个状态的键）。对于每个副本 ID，使用 Math.max 取当前状态和另一个状态中该副本计数的最大值；如果某个副本不存在，则用 0 作为默认值。最后，更新当前状态为合并后的状态。为什么这种方法有效？因为 Math.max 操作满足交换律、结合律和幂等律，这正是半格上确界操作的体现。交换律确保顺序无关，结合律允许任意分组合并，幂等律保证重复合并不改变结果。

为了演示，假设副本 A 和 B 初始状态为空。A 执行两次 increment，状态变为 { A: 2 }；B 执行一次 increment，状态变为 { B: 1 }。合并时，对每个副本取最大值，结果状态为 { A: 2, B: 1 }，value 返回 3。即使合并顺序相反，结果相同，证明了收敛性。

19 进阶：实现一个 PN-Counter（增减计数器）

G-Counter 只能增加计数，无法处理减少操作，这在现实场景如取消赞或减少库存中不够用。PN-Counter 通过分解为两个 G-Counter 来解决这一问题：P 记录增加操作，N 记录减少操作。最终值等于 P 的值减去 N 的值。例如，状态可能表示为 { P: { A: 3, B: 2 }, N: { A: 1, B: 0 } }。

操作设计包括 increment、decrement 和 value 方法。increment 调用 P 的 increment，decrement 调用 N 的 increment，value 计算差值。以下是伪代码示例：

```
function increment() {
   this.P.increment();
}

function decrement() {
   this.N.increment();
```

```
}

function value() {
  return this.P.value() - this.N.value();
}
```

在这里，increment 和 decrement 分别委托给 P 和 N 的 increment 方法，确保增减操作独立跟踪。value 方法简单计算差值，避免了直接操作全局状态。
合并策略是分别合并 P 和 N 两个 G-Counter：

```
function merge(other) {
  this.P.merge(other.P);
  this.N.merge(other.N);
}
```

由于 P 和 N 的合并都是收敛的（基于 G-Counter 的合并函数），它们的差值也会收敛。例如，假设副本 A 增加两次后减少一次，状态为 P: { A: 2 }, N: { A: 1 }；副本 B 增加一次，状态为 P: { B: 1 }, N: { B: 0 }。合并后，P 变为 { A: 2, B: 1 }，N 变为 { A: 1, B: 0 }，value 返回 (2+1) - (1+0) = 2。这种设计的正确性源于增减操作的独立性：合并时取最大值，确保不丢失任何操作信息。

20 超越计数器：CRDT 的广阔世界

除了计数器，CRDT 还应用于更复杂的数据结构。OR-Set（观察移除集合）解决了朴素集合中「添加」和「移除」操作不满足交换律的问题。在 OR-Set 中，每个元素关联唯一标记（如 UUID）；添加时生成新标记，移除时记录「墓碑」集合；查询时，元素存在当且仅当其标记在添加集且不在移除集。这确保了无论操作顺序如何，最终集合状态一致。其他 CRDT 类型包括寄存器（如 LWW-Register，基于时间戳实现最后写入获胜）、地图（由其他 CRDT 组成的复合结构）和文本序列（如 RGA 或 Logoot，用于协同编辑）。这些领域更复杂，但原理类似，通过设计可交换操作确保收敛。

21 CRDT 的优缺点与最佳实践

CRDT 的优势在于提供强最终一致性，保证无冲突收敛；支持低延迟操作，本地立即生效，无需等待服务器；以及高可用性，允许网络分区和离线工作。然而，CRDT 也面临挑战，如状态膨胀（由于墓碑或向量元数据增长，需要压缩机制）、语义限制（并非所有数据结构都能轻松设计成 CRDT）、复杂性（理解和使用比传统结构更复杂）和安全性问题（恶意副本可能污染状态，需额外机制如纯操作 CRDT）。适用场景包括协作文档编辑、分布式计数器和排行榜、购物车、分布式配置和标志管理，以及物联网设备状态同步。在这些场景中，CRDT 能有效提升系统鲁棒性和用户体验。
总之，CRDT 通过数学的优雅解决了分布式数据同步的核心挑战。从 G-Counter 到 PN-Counter 的实现，我们看到了「设计可交换操作」和「利用半格合并」的核心方法论。鼓励读者动手实现这些基础 CRDT，并尝试更复杂的类型。进一步学习资源包括论文《A comprehensive study of Convergent and Commutative Replicated Data

Types》、开源库如 automerge、yjs、delta-crdt，以及网站 crdt.tech。通过实践，你将更深入理解分布式系统的魔法。

22　附录：代码仓库链接

本文涉及的 G-Counter 和 PN-Counter 完整实现代码已发布在 GitHub 仓库中，欢迎访问和贡献。