c13n #38

c13n

2025 年 11 月 19 日

第 I 部

深入理解并实现基本的基数排序（Radix Sort）算法

叶家炜

Oct 24, 2025

在计算机科学中，排序算法是基础且重要的主题。大多数常见的排序算法，如快速排序或归并排序，都基于元素之间的比较操作。这些比较排序算法的时间复杂度下限为 $O(n \log n)$，其中 n 是元素数量。然而，在某些场景下，例如对大量整数进行排序时，比较排序可能不是最高效的选择。基数排序作为一种非比较型整数排序算法，在特定条件下能够实现线性时间复杂度 $O(n \times k)$，其中 k 是数字的最大位数。本文将深入解析基数排序的核心思想、实现细节及其应用场景。

想象一下，您需要快速整理一叠按出生年月日记录的卡片。如果使用基于比较的方法，您可能需要反复对比不同卡片的日期，这个过程在数据量较大时会变得低效。基数排序通过一种独特的方式解决了这个问题：它不直接比较元素的大小，而是依据数字的每一位进行多轮排序。这种方法在处理整数数据时，尤其当数字的位数相对较少时，能够显著提升效率。基数排序的核心优势在于其非比较特性，这使得它能够突破比较排序的理论下限，在合适条件下实现近乎线性的性能。

1 第一部分：基数排序的核心思想

基数排序的核心思想是逐位排序。这类似于整理扑克牌时，先按花色分组，再在每个组内按点数排序。对于数字，基数排序会从最低位或最高位开始，依次对每一位进行排序。关键概念包括基数「Radix」和键「Key」。基数指的是每一位数字的取值范围，例如对于十进制数，基数为十，对应零到九。键则是排序所依据的特征，即数字的每一位。基数排序主要有两种方法：最低位优先「LSD」和最高位优先「MSD」。LSD 方法从数字的最低位（如个位）开始排序，逐步向高位推进，这种方法实现简单且广泛应用。MSD 方法则从最高位开始，采用分治策略递归处理，虽然可能在某些情况下提前结束，但实现较为复杂。本文将重点介绍 LSD 基数排序，因为它更易于理解和实现。

2 第二部分：LSD 基数排序的逐步拆解

为了清晰展示 LSD 基数排序的过程，我们以一个具体数组为例：[170, 45, 75, 90, 2, 802, 2, 66]。排序过程从最低位开始，逐位进行多轮排序。每一轮排序都必须保持稳定性，即相同键值的元素在排序后保持原有相对顺序。首先，我们找出数组中最大数字的位数，这里是三位（802）。然后，我们从个位开始排序：创建十个桶，对应数字零到九，将每个数字按其个位数放入对应桶中，例如 170 的个位是零，放入零号桶；45 的个位是五，放入五号桶。之后，按桶号顺序收集所有元素，得到新数组。接下来，对十位进行同样操作：基于上一轮结果，将数字按十位数放入桶中并收集。最后，对百位进行排序。经过这三轮后，数组变为有序状态。整个过程强调稳定性，例如在个位排序中，两个数字二（2 和 2）保持原顺序，确保最终结果的正确性。算法步骤可总结为：首先确定最大位数，然后从最低位到最高位循环，每轮执行分配和收集操作。分配阶段将元素按当前位数字放入对应桶，收集阶段按桶顺序取回元素。

3 第三部分：手把手实现基数排序（代码篇）

我们将使用 Python 语言实现基数排序，因为它语法清晰，易于理解。实现过程包括两个辅助函数和一个核心函数。首先，定义辅助函数 `get_digit(num, place)`，用于获取

数字在指定位置上的数字。例如，对于数字 170，get_digit(170, 0) 返回个位数字零，get_digit(170, 1) 返回十位数字七。该函数的实现基于数学公式：(abs(num) // (10 ** place)) % 10。这里，abs(num) 取绝对值以确保处理正数，// 表示整数除法，** 表示幂运算，% 取模得到当前位数字。另一个辅助函数 get_max_digit_count(nums) 用于计算数组中最大数字的位数。它先找到数组中的最大值，然后通过循环除以十来计数位数，例如对于 802，循环三次后得到三。

核心函数 radix_sort(nums) 负责执行排序。首先处理边界情况，如果数组为空或只有一个元素，直接返回。然后，调用 get_max_digit_count 获取最大位数。接下来，循环最大位数次，每次循环对应一位数字。在循环内，初始化十个空桶，用于存放数字。分配阶段遍历数组，使用 get_digit 获取当前位数字，将元素放入对应桶中。收集阶段清空原数组，然后按桶号顺序将桶中元素放回数组。最后，返回排序后的数组。代码测试时，使用示例数组 [170, 45, 75, 90, 2, 802, 2, 66]，打印每轮结果以验证正确性。例如，第一轮按个位排序后，数组可能变为 [170, 90, 2, 802, 2, 45, 75, 66]，其中个位数字有序。

```
def get_digit(num, place):
    return (abs(num) // (10 ** place)) % 10

def get_max_digit_count(nums):
    if not nums:
        return 0
    max_num = max(nums)
    count = 0
    while max_num != 0:
        count += 1
        max_num //= 10
    return count

def radix_sort(nums):
    if len(nums) <= 1:
        return nums
    max_digit_count = get_max_digit_count(nums)
    for k in range(max_digit_count):
        buckets = [[] for _ in range(10)]
        for num in nums:
            digit = get_digit(num, k)
            buckets[digit].append(num)
        nums = []
        for bucket in buckets:
            nums.extend(bucket)
    return nums
```

在代码解读中，get_digit 函数通过数学运算精确提取指定位的数字，确保处理各种整数。get_max_digit_count 函数使用循环计算位数，避免了对数运算可能带来的精度问题。核

心函数 radix_sort 通过嵌套循环实现多轮排序，外层循环控制位数，内层循环处理分配和收集。分配阶段利用列表推导创建桶，收集阶段使用 extend 方法高效合并元素。整个实现强调代码的可读性和效率，同时确保稳定性。

4 第四部分：深入分析与探讨

基数排序的时间复杂度为 $O(n \times k)$，其中 n 是元素数量，k 是最大数字的位数。这是因为外层循环执行 k 次，内层的分配和收集操作各为 $O(n)$。空间复杂度为 $O(n + r)$，其中 r 是基数（十进制下为十），因为需要额外存储 r 个桶和桶内的 n 个元素。基数排序是稳定的排序算法，这意味着在排序过程中，相同键值的元素保持原有顺序，这对于多关键字排序至关重要。例如，如果先按十位排序，再按个位排序，稳定性确保十位相同的元素在个位排序后仍保持正确顺序。

基数排序的优点包括：当 k 远小于 n 时，效率高于 $O(n \log n)$ 的比较排序；并且它是稳定的。缺点在于：它不是原地排序，需要额外空间；通常只适用于整数或可表示为整数的类型；如果数字范围很大（k 很大）而 n 较小，效率可能不如比较排序。与其他算法相比，基数排序在整数排序中具有独特优势。例如，与快速排序相比，基数排序在特定条件下性能更优，但快排是原地排序且通用性强。与计数排序相比，基数排序通过多轮排序解决了计数排序在数字范围大时空间消耗高的问题。

处理负数时，上述实现需要扩展。我们可以将数组分为正数和负数两部分。对负数部分取绝对值，进行基数排序后反转结果；对正数部分直接排序；最后合并两部分。这确保了负数能正确排序，同时保持算法稳定性。

基数排序通过逐位排序的方式，实现了对整数的高效排序。本文详细介绍了 LSD 基数排序的核心思想、实现步骤和复杂度分析，强调了稳定性和线性时间复杂度的前提条件。读者在理解后，可以尝试实现 MSD 基数排序或探索其他基数（如二进制）以优化性能。基数排序在现实中有广泛应用，例如早期卡片排序机和现代大数据处理。鼓励读者动手实现代码，并思考如何扩展至更复杂的数据类型。通过深入理解基数排序，我们不仅掌握了一种高效算法，更提升了解决实际问题的能力。

第 II 部

深入理解并实现基本的信号量（Semaphore）机制

杨岢瑞

Oct 25, 2025

想象一个只有五个停车位的停车场，多辆汽车代表多个线程同时试图进入。如果没有有效的管理机制，可能会导致超过五辆车挤入，引发混乱和冲突。这种场景在计算机科学中极为常见，多个进程或线程竞争有限资源时，无序访问会造成数据不一致或系统崩溃。为了解决这类问题，我们需要一种可靠的“看门人”机制来协调资源分配，这正是信号量诞生的背景。信号量作为一种经典的同步原语，不仅是操作系统和并发编程的核心工具，更是理解现代计算系统如何管理共享资源的关键。本文将带领读者从基础概念出发，逐步深入信号量的内部原理，并通过代码实现和经典案例，揭示其在解决同步与互斥问题中的强大能力。

5 信号量到底是什么?

信号量本质上是一个特殊的整型变量，它不仅仅存储一个数值，还包含一套完整的机制来管理这个值。其核心在于通过原子操作确保对资源的访问是可控的。信号量的定义包括三个基本要素：整数值表示当前可用资源的数量，等待队列用于在资源不足时存放被阻塞的线程或进程，以及两个原子操作 P 和 V。原子操作是信号量正确性的基石，因为它们保证了在检查和修改信号量值的过程中不会被其他操作中断。

P 操作通常称为 wait 或 acquire，其语义是尝试申请一个资源。伪代码流程如下：首先将信号量的值减一，然后检查新值是否小于零；如果小于零，说明资源不足，当前线程会被阻塞并加入等待队列。形象地说，这就像一位司机试图进入停车场：如果车位已满，他就必须等待。V 操作则称为 signal 或 release，用于释放资源。其流程是将信号量值加一，然后检查值是否小于等于零；如果是，说明有线程在等待，便从队列中唤醒一个线程。这相当于一辆车离开停车场，空出一个车位，并通知等待的司机进入。关键点在于，P 和 V 操作必须是原子的，这避免了竞态条件，同时等待队列的设计使得信号量不同于简单的忙等待机制，能够有效节省 CPU 资源。

6 信号量的两种类型与应用场景

信号量主要分为计数信号量和二进制信号量，它们在应用场景和功能上有所区别。计数信号量的值可以是任何非负整数，用于控制对多种资源的访问。例如，在数据库连接池中，如果最多允许十个连接，计数信号量初始化为十，每次连接申请时执行 P 操作，释放时执行 V 操作，确保连接数不超过上限。另一个经典例子是生产者-消费者问题中的缓冲区管理，信号量用于跟踪空位和满位的数量。

二进制信号量的值仅限于零或一，常用于实现互斥锁来保护临界区。通过将信号量初始值设为一，线程在进入临界区前执行 P 操作，离开后执行 V 操作。这样，第一个线程执行 P 操作后值变为零，顺利进入；第二个线程执行 P 操作后值变为负一，被阻塞；直到第一个线程执行 V 操作唤醒等待者。这种机制确保了同一时间只有一个线程访问共享资源，避免了数据竞争。总结来说，计数信号量适用于资源池管理，而二进制信号量更专注于互斥控制，两者在并发编程中各有其不可替代的作用。

7 实战：用代码实现一个简单的信号量

为了深入理解信号量的内部机制，我们尝试使用基本的同步原语如互斥锁和条件变量来实现一个简单的信号量。这里以 Python 为例，因为它语法清晰，易于演示。我们的目标是用

一个整型变量 count 记录信号量值，一个互斥锁 mutex 保护对 count 的并发访问，以及一个条件变量 condition 实现线程的等待和唤醒。

```
import threading

class SimpleSemaphore:
    def __init__(self, initial_value):
        self._count = initial_value
        self._mutex = threading.Lock()
        self._condition = threading.Condition(self._mutex)

    def P(self):
        with self._mutex:
            self._count -= 1
            while self._count < 0:
                self._condition.wait()

    def V(self):
        with self._mutex:
            self._count += 1
            if self._count <= 0:
                self._condition.notify()
```

在这段代码中，SimpleSemaphore 类初始化时设置初始值，并创建互斥锁和条件变量。P 方法首先获取互斥锁，确保原子性，然后将 _count 减一。如果减一后值小于零，线程会进入等待状态，此时条件变量会原子地释放锁并阻塞线程，直到被唤醒。这里使用 while 循环而非 if 语句是关键，因为它能防止虚假唤醒：线程可能在没有明确通知的情况下被唤醒，因此需要重新检查条件是否满足。V 方法同样在互斥锁保护下将 _count 加一，如果值小于等于零（表示有线程在等待），则通知一个等待的线程。这种实现忠实地还原了信号量的理论模型，其中 condition.wait() 对应阻塞操作，condition.notify() 对应唤醒操作，确保了线程安全和高效率。

8　经典案例：生产者-消费者问题

生产者-消费者问题是并发编程中的经典范例，它涉及多个生产者和消费者共享一个有限大小的缓冲区。生产者生成数据放入缓冲区，消费者从缓冲区取出数据，需要解决互斥和同步问题：互斥确保对缓冲区的访问不会同时进行，同步则要求缓冲区满时生产者等待，空时消费者等待。信号量提供了一种优雅的解决方案，使用三个信号量协作：一个二进制信号量 mutex 初始化为一来保护缓冲区，一个计数信号量 empty 初始化为 N 表示空缓冲区数量，另一个计数信号量 full 初始化为零表示满缓冲区数量。

生产者线程的伪代码如下：首先生产一个项目，然后执行 empty.P() 等待空位，接着执行 mutex.P() 进入临界区插入项目，离开临界区后执行 mutex.V()，最后执行 full.V() 通知多了一个满位。消费者线程则先执行 full.P() 等待满位，然后获取互斥锁取出项

目，释放锁后执行 empty.V() 通知多了一个空位，再消费项目。这种设计确保了在任何情况下都不会出现缓冲区溢出或下溢，同时避免了死锁。例如，如果缓冲区满，生产者会在 empty.P() 处阻塞，直到消费者释放空位；类似地，消费者在缓冲区空时会在 full.P() 处等待。通过信号量的精细协调，生产者和消费者能够高效、安全地协作，展示了信号量在解决复杂同步问题中的强大能力。

9 实际编程中的信号量

在实际编程中，各种语言和平台提供了内置的信号量实现，简化了开发过程。POSIX 标准中，<semaphore.h> 头文件定义了 sem_init、sem_wait、sem_post 和 sem_destroy 等函数，用于初始化和操作信号量。例如，sem_wait 对应 P 操作，sem_post 对应 V 操作，这些函数底层由操作系统保证原子性。在 Java 中，java.util.concurrent.Semaphore 类提供了丰富的 API，如 acquire 和 release 方法，支持公平性和超时设置，适用于高并发场景。Python 的 threading.Semaphore 类则与我们在实战中实现的 SimpleSemaphore 类似，但经过优化和测试，可以直接用于生产环境。这些实现虽然封装了底层细节，但核心原理与我们自定义的信号量一致，强调了原子操作和等待队列的重要性。开发者应根据具体需求选择合适的工具，例如在需要高性能时选择原生库，或在跨平台项目中使用语言标准库。

信号量作为并发编程的基石，通过计数器、等待队列和原子操作 P 与 V，提供了一种高效解决资源同步和互斥问题的方法。我们从停车场例子出发，理解了信号量的必要性；然后深入其定义，揭示了 P 和 V 操作的原子性关键；接着区分了计数信号量和二进制信号量的应用场景；并通过代码实现，展示了如何用互斥锁和条件变量构建信号量，强调了防止虚假唤醒的重要性。在生产者-消费者问题中，我们看到信号量如何协调多个线程，确保系统稳定运行。实际编程中，各种语言库提供了现成实现，但理解其原理有助于更好地应用和调试。信号量的思想深远影响了现代计算，例如在限流器和资源池中都能看到其影子。掌握信号量，不仅是学习并发编程的必经之路，更是提升系统设计能力的关键一步。

第 III 部

深入理解并实现基本的脚本语言解释器

杨子凡

Oct 26, 2025

10 从零开始，用几百行代码构建你的迷你「Python」或「JavaScript」

在编程世界中，我们经常使用像 Python 或 JavaScript 这样的脚本语言来执行动态任务。想象一下，当我们写下 `a = 10 + 2 * 3; print(a);` 这样一行简单的代码时，计算机是如何一步步理解并执行它的呢？这背后隐藏着解释器的神奇工作原理。本文将带你从零开始，深入探讨并亲手实现一个基本的脚本语言解释器。通过这个过程，你不仅能深化对编程语言内部机制的理解，还能掌握构建一个支持变量、算术运算、条件语句、循环和函数的动态类型语言的核心技能。我们称之为 MiniScript，并使用 Python 作为实现语言，因为其语法简洁，易于表达思想，但所涉及的原理是通用的。

造一个解释器可能听起来像在重复发明轮子，但这个过程却能极大地提升我们对编程语言本质的认识。以一个简单的代码片段为例：`a = 10 + 2 * 3; print(a);`。这行代码涉及赋值、算术运算和输出，计算机需要将其从文本转化为实际行动。通过实现解释器，我们可以直观地看到代码是如何被解析和执行的，这就像学习一门新语言时，理解其语法规则一样重要。解释器无处不在，从 Python 到 JavaScript，许多流行语言都依赖解释执行。更重要的是，一个基础的解释器并不复杂，其核心逻辑可以用几百行代码实现，这让它成为学习编译器设计和语言实现的绝佳起点。本文的目标是构建一个支持变量、算术运算、比较运算、条件语句、循环语句和函数的动态类型语言 MiniScript。最终，你将获得一个可运行的迷你解释器，并能通过动手实践加深对关键概念的理解。

11 背景知识：解释器与编译器的区别

在深入实现之前，我们需要先理解解释器和编译器的核心区别。编译器将源代码一次性翻译成另一种语言，通常是机器码，生成独立的可执行文件。这就像将一本中文书完整翻译成英文出版，整个过程是离线的。而解释器则直接读取源代码，边解析边执行，类似于同声传译，一边听一边现场翻译。对于 MiniScript，我们将采用解释器的方式，其工作流程可以概括为：源代码首先经过词法分析，被拆分成令牌流；然后通过语法分析构建抽象语法树；最后解释执行这棵树。这个流程确保了代码的逐步理解和执行，为后续实现奠定了基础。

12 第一步：词法分析 —— 从字符流到令牌流

词法分析是解释器的第一个阶段，目标是将源代码字符串拆分成一个个有意义的「单词」，即令牌。例如，代码 `a = 10 + 2;` 会被转换为令牌序列：`[Identifier('a'), Assign(), Number(10), Plus(), Number(2), Semicolon()]`。这个过程类似于人类阅读时识别单词和标点。实现词法分析器时，我们使用正则表达式来高效匹配字符序列。首先，定义令牌类型，如 `IDENTIFIER`、`NUMBER`、`PLUS` 等，这些类型代表了代码中的基本元素。然后，构建一个 `Lexer` 类，该类包含属性如 `input_text`（存储源代码字符串）和 `position`（跟踪当前处理位置），以及方法如 `next_token()`（用于获取下一个令牌）、`skip_whitespace()`（跳过空白字符）、`read_number()`（读取数字）和 `read_identifier()`（读取标识符）。这些方法协同工作，逐步扫描输入文本并生成令牌流。

以下是一个简单的 `Lexer` 类核心代码示例，我们使用 Python 实现。代码中，`Token` 类

用于表示令牌的类型和值，Lexer 类通过遍历字符来生成令牌。例如，next_token 方法会检查当前字符，如果是数字，则调用 read_number 来解析整数；如果是字母，则调用 read_identifier 来解析变量名。这个过程确保了代码的每个部分都被正确分类。

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

class Lexer:
    def __init__(self, input_text):
        self.input_text = input_text
        self.position = 0
        self.current_char = self.input_text[self.position] if self.
            ↪ input_text else None

    def advance(self):
        self.position += 1
        if self.position >= len(self.input_text):
            self.current_char = None
        else:
            self.current_char = self.input_text[self.position]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.
            ↪ isspace():
            self.advance()

    def read_number(self):
        result = ''
        while self.current_char is not None and self.current_char.
            ↪ isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def read_identifier(self):
        result = ''
        while self.current_char is not None and (self.current_char.
            ↪ isalnum() or self.current_char == '_'):
            result += self.current_char
            self.advance()
```

```
        return result

    def next_token(self):
        while self.current_char is not None:
            if self.current_char.isspace():
                self.skip_whitespace()
                continue
            if self.current_char.isdigit():
                return Token('NUMBER', self.read_number())
            if self.current_char.isalpha() or self.current_char == '_':
                identifier = self.read_identifier()
                if identifier in KEYWORDS: # KEYWORDS 是预定义的关键字字典，
                    ↪ 如 'if', 'while'
                    return Token(identifier.upper(), identifier)
                return Token('IDENTIFIER', identifier)
            if self.current_char == '=':
                self.advance()
                return Token('ASSIGN', '=')
            if self.current_char == '+':
                self.advance()
                return Token('PLUS', '+')
            # 类似处理其他操作符和符号
            self.advance()
        return Token('EOF', None)
```

在这段代码中，Lexer 类通过 advance 方法移动当前位置，并利用辅助函数如 read_number 和 read_identifier 来提取数字和标识符。next_token 方法是核心，它循环处理字符，跳过空白后，根据字符类型返回相应的令牌。例如，遇到数字时，它会累积所有连续数字字符并转换为整数；遇到字母时，它会读取整个标识符，并检查是否为关键字。这种设计确保了词法分析的高效性和准确性，为后续语法分析提供了干净的输入。

13 第二步：语法分析 —— 构建抽象语法树

语法分析是解释器的第二个阶段，目标是将令牌流转换为抽象语法树，这是一种树形数据结构，能体现代码的层次结构和语义。以表达式 a = 10 + 2 * 3 为例，其 AST 可能包含一个赋值节点，左子节点是变量 a，右子节点是一个二元操作节点，表示加法，其中左操作数是数字 10，右操作数是另一个二元操作节点，表示乘法（左操作数 2，右操作数 3）。这种结构清晰地反映了运算符的优先级和结合性，例如乘法在加法之前执行。AST 是后续解释执行的基础，因为它将线性代码转换为可遍历的树形形式。

为了定义 MiniScript 的语法，我们使用类似 BNF 的表示法来描述规则。例如，程序由多个语句组成，语句可以是表达式语句、if 语句或 while 语句；表达式可以进一步分解为赋值、相等比较、项、因子等。具体规则包括：程序是零个或多个语句的序列；语句包括表达

式语句（以分号结尾）、if 语句和 while 语句；表达式从赋值开始，赋值可以是标识符后跟等号和表达式，或者相等比较；相等比较又由比较操作构成，比较操作由项和关系运算符组成；项由因子和加减运算符组成；因子由一元操作和乘除运算符组成；一元操作可以是逻辑非或负号，后跟一元操作或基本元素；基本元素包括数字、字符串、标识符或括号内的表达式。这些规则确保了语法的递归和层次性。

实现语法分析器时，我们采用递归下降分析法，这是一种直观且适合手写的方法。Parser 类包含属性如 tokens（存储令牌列表）和 current（当前令牌索引），以及一系列方法对应每条语法规则，如 parse_program、parse_statement 和 parse_expression。核心技巧是使用前瞻令牌来决定解析路径，例如在解析表达式时，查看下一个令牌类型以选择正确的规则。

以下是一个 Parser 类的核心代码示例，展示如何解析表达式和赋值语句。代码中，parse_expression 方法从赋值开始处理，而 parse_assignment 检查是否为标识符后跟等号，否则递归处理相等比较。

```
class Parser:
   def __init__(self, tokens):
      self.tokens = tokens
      self.current = 0

   def peek(self):
      return self.tokens[self.current] if self.current < len(self.
          ↪ tokens) else None

   def advance(self):
      if self.current < len(self.tokens):
         self.current += 1

   def parse_program(self):
      statements = []
      while self.peek() and self.peek().type != 'EOF':
         statements.append(self.parse_statement())
      return Program(statements) # Program 是 AST 根节点类

   def parse_statement(self):
      token = self.peek()
      if token.type == 'IF':
         return self.parse_if_statement()
      elif token.type == 'WHILE':
         return self.parse_while_statement()
      else:
         expr = self.parse_expression()
         if self.peek() and self.peek().type == 'SEMICOLON':
```

```
            self.advance()
        return ExpressionStatement(expr) # ExpressionStatement 是语句节点类

    def parse_expression(self):
        return self.parse_assignment()

    def parse_assignment(self):
        left = self.parse_equality()
        if self.peek() and self.peek().type == 'ASSIGN':
            self.advance()
            value = self.parse_assignment()
            return Assign(left, value) # Assign 是赋值节点类
        return left

    def parse_equality(self):
        left = self.parse_comparison()
        while self.peek() and self.peek().type in ('EQ', 'NEQ'): # EQ 和 NEQ 表示 == 和 !=
            operator = self.peek().type
            self.advance()
            right = self.parse_comparison()
            left = BinaryOp(left, operator, right) # BinaryOp 是二元操作节点类
        return left

    # 类似实现 parse_comparison、parse_term 等方法
```

在这段代码中，Parser 类通过 peek 方法查看当前令牌，而不消耗它，advance 方法移动到下一个令牌。parse_expression 从顶层开始解析，逐步下降到更具体的规则。例如，在 parse_assignment 中，如果遇到等号令牌，则构建赋值节点；否则，返回解析的表达式。这种方法确保了语法规则的正确应用，并构建出完整的 AST。通过递归下降，我们可以高效地处理嵌套结构，如表达式中的括号，从而为解释执行阶段提供结构化的输入。

14 第三步：解释执行 —— 让 AST「活」起来

解释执行是解释器的核心阶段，我们遍历 AST 并执行计算。为了清晰处理不同类型的节点，我们使用访问者模式，这允许为每种 AST 节点定义特定的执行逻辑。例如，数字节点直接返回值，而二元操作节点需要递归计算左右子树后再执行运算。同时，执行环境用于存储变量和函数定义，本质上是一个字典，支持作用域链，通过维护环境栈或让环境持有父环境引用来实现嵌套作用域。

Interpreter 类是解释执行的主体，包含属性如 environment（当前执行环境），以及一

系列 visit_XXX 方法，如 visit_BinaryOp、visit_Assign 和 visit_Variable。这些方法对应 AST 节点类型，确保每个节点都能被正确解释。例如，在访问字面量节点时，visit_Number 直接返回数字值；在访问二元操作节点时，visit_BinaryOp 先递归访问左子树和右子树获取值，然后根据操作符执行相应运算，如加法或比较。对于变量赋值，visit_Assign 将值写入环境；对于变量引用，visit_Variable 从环境中读取值。控制流节点如 if 语句和 while 循环，则通过计算条件表达式值来决定执行路径。

以下是一个 Interpreter 类的核心代码示例，展示如何实现基本节点的访问方法。代码中，我们假设环境是一个字典，支持设置和获取变量值。

```
class Environment:
    def __init__(self, parent=None):
        self.vars = {}
        self.parent = parent

    def get(self, name):
        if name in self.vars:
            return self.vars[name]
        elif self.parent:
            return self.parent.get(name)
        else:
            raise NameError(f"未定义的变量␣:␣{name}")

    def set(self, name, value):
        self.vars[name] = value

class Interpreter:
    def __init__(self):
        self.environment = Environment()

    def visit(self, node):
        method_name = f'visit_{type(node).__name__}'
        method = getattr(self, method_name, self.generic_visit)
        return method(node)

    def generic_visit(self, node):
        raise Exception(f 没有 visit_{type(node).__name__} 方法)

    def visit_Number(self, node):
        return node.value

    def visit_BinaryOp(self, node):
        left = self.visit(node.left)
```

```
        right = self.visit(node.right)
        if node.operator == 'PLUS':
            return left + right
        elif node.operator == 'MINUS':
            return left - right
        elif node.operator == 'MULTIPLY':
            return left * right
        elif node.operator == 'DIVIDE':
            return left / right
        elif node.operator == 'EQ':
            return left == right
        # 类似处理其他操作符
        else:
            raise Exception(f"未知操作符␣:␣{node.operator}")

    def visit_Assign(self, node):
        value = self.visit(node.value)
        self.environment.set(node.name, value)
        return value

    def visit_Variable(self, node):
        return self.environment.get(node.name)

    def visit_If(self, node):
        condition = self.visit(node.condition)
        if condition:
            return self.visit(node.then_branch)
        elif node.else_branch:
            return self.visit(node.else_branch)
        return None

    def visit_While(self, node):
        result = None
        while self.visit(node.condition):
            result = self.visit(node.body)
        return result
```

在这段代码中，Interpreter 类使用 visit 方法动态调用相应的节点访问方法。例如，visit_Number 直接返回节点的数值；visit_BinaryOp 先递归计算左右操作数的值，然后根据操作符执行运算，如加法或相等比较。对于赋值节点，visit_Assign 计算右侧表达式的值并存储到环境中；对于变量节点，visit_Variable 从环境中检索值。控制流节点如 If 和 While 通过条件判断来执行分支或循环体。这种设计确保了 AST 的逐步执行，并

处理了语句和表达式的区别：表达式产生值，而语句执行操作但不一定返回值。通过环境管理，我们实现了变量的动态存储和检索，为更复杂的特性如函数打下了基础。

15　进阶特性：实现函数

函数是编程语言中的核心抽象，允许我们将代码封装为可重用的单元。在 MiniScript 中，函数被视为一种可调用对象，包含参数列表和函数体（一个代码块 AST）。实现函数涉及两个主要步骤：定义和调用。在定义时，我们解析 function 关键字，创建一个 Function 对象，该对象存储参数、函数体和声明时的环境（用于实现闭包），并将其作为值存入当前环境。在调用时，我们计算实参的值，创建新的函数作用域（以函数定义时的环境为父环境），绑定形参和实参，然后在新环境中执行函数体。

以下是一个函数实现的核心代码示例。我们扩展 Interpreter 类来处理函数定义和调用，并引入 Function 类来表示函数对象。

```
class Function:
    def __init__(self, params, body, env):
        self.params = params
        self.body = body
        self.env = env # 定义时的环境，用于闭包

class Interpreter:
    # 之前的方法保持不变

    def visit_FunctionDef(self, node):
        func = Function(node.params, node.body, self.environment)
        self.environment.set(node.name, func)
        return func

    def visit_Call(self, node):
        func = self.visit(node.func)
        args = [self.visit(arg) for arg in node.args]
        if isinstance(func, Function):
            new_env = Environment(parent=func.env) # 创建新环境，父环境为定义时的环境
            for param, arg in zip(func.params, args):
                new_env.set(param, arg)
            old_env = self.environment
            self.environment = new_env
            try:
                result = self.visit(func.body)
            finally:
                self.environment = old_env
```

```
            return result
        else:
            raise Exception(f"{func}␣不是可调用对象")
```

在这段代码中，visit_FunctionDef 方法处理函数定义，它创建一个 Function 实例并存储到当前环境中。visit_Call 方法处理函数调用：首先计算函数表达式和参数值，然后检查函数是否为 Function 实例。如果是，它创建一个新环境，其父环境设置为函数定义时的环境，这实现了闭包 —— 函数可以访问定义时的变量。接着，将形参和实参绑定到新环境，临时切换解释器的环境来执行函数体，最后恢复原环境。这种设计确保了函数的作用域隔离和参数传递，同时通过闭包支持了高级特性如嵌套函数。例如，如果一个函数内部定义了另一个函数，内部函数可以访问外部函数的变量，因为这保存在定义时的环境中。

通过本文的旅程，我们从字符流开始，逐步实现了词法分析、语法分析和解释执行，构建了一个完整的 MiniScript 解释器。这个过程不仅让你理解了代码如何从文本转化为行动，还深化了对 AST、作用域和访问者模式等概念的认识。现在，你应该能够运行自己的迷你解释器，并扩展其功能。未来，你可以进一步优化性能，例如引入字节码和虚拟机；添加新特性如数组、字典和类；改进错误处理；或构建标准库。我鼓励你动手实践，分享代码，并参考资源如《编程语言实现模式》来继续探索。编程语言的实现是一个充满挑战和乐趣的领域，希望本文为你打开了这扇大门。

第 IV 部

深入理解并实现基本的深度优先搜索（DFS）算法

杨其臻

Oct 28, 2025

想象你身处一个迷宫，面前有多条岔路。你没有地图，只有一个目标：找到出口。这时，你有两种策略可以选择。策略一称为广度优先搜索，它要求你把所有岔路口都标记一下，然后一个个路口轮流去探索一步。策略二则截然不同，它让你选择一条路，一头扎到底，直到走到死胡同或者找到出口。如果走不通，就退回上一个岔路口，换另一条没走过的路继续深入。这种策略二正是我们今天要深入探讨的深度优先搜索（DFS）算法的核心思想。DFS 是一种用于遍历或搜索树或图数据结构的经典算法，它在计算机科学中有着广泛的应用。

16 DFS 的核心思想与算法流程

深度优先搜索（DFS）的官方定义是：一种用于遍历或搜索树或图数据结构的算法，它从根节点（或任意节点）开始，在回溯之前尽可能深地探索每一个分支。其核心思想可以概括为三个要素。第一是“不撞南墙不回头”，即沿着一条路径一直向下走，直到无法继续。第二是“回溯”，当无法继续前进时，退回一步（上一个节点），寻找其他未探索的路径。第三是递归与栈，这种“前进”与“回溯”的过程天然适合用递归或显式栈（Stack）来实现。
算法步骤的文字描述如下。首先，从起始节点开始，将其标记为“已访问”。接着，检查当前节点是否为目标节点。如果是，则搜索成功。如果否，则遍历当前节点的所有“未访问”的相邻节点。然后，对每一个相邻节点，递归地执行上述步骤。最后，如果所有相邻节点都已访问且未找到目标，则回溯到上一个节点。这个过程体现了 DFS 的深度优先特性，确保算法优先探索最深的路径。

17 图解 DFS：遍历一棵树

为了直观理解 DFS，我们以遍历一棵简单的二叉树为例。假设二叉树有根节点 A，左子节点 B，右子节点 C，B 有左子节点 D 和右子节点 E，C 有左子节点 F。DFS 的遍历从根节点 A 开始，首先访问 A，然后深入左子树，访问 B，接着继续深入左子树，访问 D。由于 D 没有子节点，算法回溯到 B，然后访问 B 的右子节点 E。回溯到 A 后，再深入右子树，访问 C，然后访问 C 的左子节点 F。整个过程访问顺序为 A、B、D、E、C、F，这体现了前序遍历模式。通过这种文字描述，读者可以感受到 DFS“一路到底”的访问模式，无需依赖图像。

18 DFS 的两种实现方式

18.1 递归实现（最直观）

递归实现利用系统的函数调用栈来隐式地实现回溯，代码简洁直观。以下是一个 Python 示例代码框架。

```
def dfs_recursive(node, target, visited=None):
    if visited is None:
        visited = set() # 用于记录已访问节点，避免重复访问（图结构尤其重要）

    if node is None or node in visited:
        return None

```

```
    # 1. 访问当前节点
    visited.add(node)
    print(f"Visiting node: {node.val}") # 执行访问操作
    if node.val == target:
        return node # 找到目标

    # 2. 递归地访问所有相邻节点（左子树、右子树）
    left_result = dfs_recursive(node.left, target, visited)
    if left_result is not None:
        return left_result

    right_result = dfs_recursive(node.right, target, visited)
    return right_result
```

代码解读：这个递归函数首先检查节点是否为空或已访问，如果是则返回。然后访问当前节点，标记为已访问，并检查是否为目标。如果不是，则递归调用左子树和右子树。递归过程自然地实现了“深入”：当调用左子树时，函数会一直向左深入，直到叶子节点；然后通过返回机制实现“回溯”，回到上一个节点继续探索右子树。使用 visited 集合避免了重复访问，这在图结构中至关重要。

18.2 迭代实现（使用显式栈）

迭代实现使用一个栈数据结构来模拟递归过程，手动管理待访问的节点。以下是一个 Python 示例代码框架。

```
def dfs_iterative(start_node, target):
    if not start_node:
        return None

    stack = [start_node] # 初始化栈，放入起始节点
    visited = set() # 记录已访问节点

    while stack: # 栈不为空则继续
        node = stack.pop() # 弹出栈顶元素（关键步骤！）

        if node in visited:
            continue

        # 访问当前节点
        visited.add(node)
        print(f"Visiting node: {node.val}")
        if node.val == target:
            return node # 找到目标
```

```
        # !!! 注意：为了保持与递归相同的左子树优先顺序，需要反向压入相邻节点
        # 例如，先压入右子节点，再压入左子节点，这样左子节点会先被弹出
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

    return None # 未找到目标
```

代码解读：这个迭代版本使用一个栈来存储待访问节点。循环中，每次弹出栈顶节点（后进先出原则），访问它并检查是否为目标。然后，将相邻节点压入栈中，但为了模拟递归的左子树优先顺序，需要先压入右子节点，再压入左子节点。这样，左子节点会在栈顶先被弹出，确保深度优先。栈的 LIFO 特性保证了算法总是优先探索最近添加的节点，从而实现深度优先搜索。这种方法避免了递归可能导致的栈溢出问题，适用于深度较大的场景。

19 从树到图：DFS 的应用扩展与关键点

当将 DFS 应用到图结构时，图的特殊性在于可能存在环。如果不记录已访问节点，算法会在环中无限循环，导致性能问题或死循环。关键挑战在于如何避免重复访问。解决方案是使用一个 visited 集合（或数组）来记录所有已访问过的节点，这在之前的代码中已经体现。代码调整方面，只需将遍历“左右子节点”的逻辑替换为遍历图的“邻接表”或“邻接矩阵”，核心框架不变。例如，在图结构中，相邻节点可能通过列表或字典表示，DFS 会递归或迭代地访问每个邻接节点，同时维护 visited 集合以防止循环。

20 DFS 的经典应用场景

深度优先搜索在多个领域有经典应用。在路径查找中，它可以用于解决迷宫问题或判断图中两点是否连通。拓扑排序是另一个重要应用，用于有向无环图（DAG），解决任务调度、编译顺序等问题。DFS 还能检测图中是否有环：在遍历过程中，如果遇到一个“已访问”且不是父节点的节点，则说明存在环。此外，在无向图中，一次 DFS 可以遍历一个连通分量，用于计算图的连通性。更重要的是，DFS 是回溯算法的基石，解决 N 皇后、数独、组合求和等问题时，核心就是 DFS 加上剪枝策略，通过深度探索和回溯来寻找所有可能解。

21 DFS 的优缺点分析

深度优先搜索有其独特的优点和缺点。优点方面，实现简单，代码简洁，尤其是递归形式，易于理解和编写。对于深度很大但目标在深处的场景，DFS 可能比广度优先搜索（BFS）更快找到解，因为它优先探索深层路径。空间复杂度相对较低，主要取决于递归深度或栈的深度，在最坏情况下为 $O(h)$（树高）或 $O(V)$（图的节点数），其中 h 表示树的高度，V 表示节点数。缺点方面，DFS 不一定找到最短路径，这是 BFS 的优势，因为 DFS 可能先探索一条长路径而忽略更短的选项。如果搜索树深度无限，递归实现可能导致栈溢出，尤其是在编程语言中递归深度有限的情况下。在状态空间巨大且无解的情况下，DFS 可能会陷入“深

度陷阱”，性能不佳，因为它会一直深入直到回溯。

回顾 DFS 的核心，我们再次强调其“深度优先”和“回溯”的思想，这使它成为遍历树和图的高效工具。实现上，递归与迭代两种方式各有千秋，但都基于栈的原理。鼓励读者动手实践，例如在 LeetCode 等平台上尝试“二叉树的最大深度”或“路径总和”等题目，以加深对 DFS 的理解和应用。通过不断练习，读者可以掌握如何在不同场景下灵活运用 DFS。

22 附录与思考题

思考题一：如何修改 DFS 算法来记录并输出从起点到目标点的完整路径？这可以通过在递归或迭代过程中维护一个路径栈或列表来实现，每次访问节点时记录路径，回溯时移除节点。思考题二：在迭代实现的 DFS 中，如果希望访问顺序与递归实现完全一致，压栈顺序应该是怎样的？答案是先压入右子节点，再压入左子节点，以确保左子节点先被访问。相关阅读推荐广度优先搜索（BFS）算法，读者可以比较 DFS 与 BFS 的异同，进一步理解搜索策略的选择。

第 V 部

深入理解并实现基本的垃圾回收（Garbage Collection）机制

王思成

Oct 29, 2025

从手动管理到自动化 —— 亲手打造一个微型内存管理器

在编程领域，内存管理始终是一个核心挑战。以 C 和 C++ 为例，开发者必须手动管理堆内存，使用诸如 `malloc` 和 `free` 或 `new` 和 `delete` 等函数。然而，这种手动方式容易导致内存泄漏、悬空指针和双重释放等问题。随着软件系统复杂度的提升，这些错误的维护成本急剧增加，往往成为程序崩溃和安全漏洞的根源。垃圾回收机制应运而生，它自动追踪和释放不再使用的内存，从而提升开发效率、减少内存相关错误并增强程序健壮性。垃圾回收已成为许多现代高级语言如 Java、C#、Go、Python 和 JavaScript 的运行时核心。本文旨在引导读者从理论到实践，深入理解垃圾回收的核心原理，并最终使用 C 语言实现一个简单的标记-清除垃圾回收器，完成一次从手动管理到自动化的探索之旅。

23　垃圾回收的基础概念

垃圾回收的核心在于准确识别哪些内存是存活的，哪些是垃圾。首先，我们需要定义根对象。根对象是垃圾回收器遍历的起点，通常包括全局变量、栈上的局部变量和寄存器中的变量。所有能被根对象直接或间接访问到的对象都被视为存活对象，其余则被标记为垃圾。这个概念称为可达性，它通过引用链形成一幅存活对象图。例如，如果对象 A 引用对象 B，而对象 B 引用对象 C，且根对象能访问 A，那么 A、B、C 都是可达的，因此存活。垃圾回收的步骤可以分解为分配、追踪和回收。分配阶段处理程序的内存请求；追踪阶段从根对象开始遍历所有存活对象；回收阶段则释放垃圾对象占用的内存，以便后续复用。这一过程确保了内存资源的有效利用。

24　经典的垃圾回收算法

垃圾回收算法有多种实现方式，其中三种经典方法是引用计数法、标记-清除法和复制算法。引用计数法为每个对象维护一个引用计数器，记录指向它的指针数量。当计数器为零时，对象被立即回收。这种方法的优点是实时性高，垃圾一经产生即刻回收，但缺点是无法处理循环引用，且计数器更新开销大。标记-清除法分为两个阶段：标记阶段从根对象开始遍历对象图，为所有可达对象打上标记；清除阶段遍历整个堆，回收未标记的对象。它能正确处理循环引用，但可能导致内存碎片，并在执行时暂停整个程序。复制算法将堆分为两个空间，只在一个空间分配内存，GC 时将存活对象复制到另一个空间，然后交换角色。这解决了内存碎片问题，分配速度快，但内存利用率低，复制存活对象开销较大。总体而言，引用计数法适用于简单场景，标记-清除法平衡了复杂性和功能，而复制算法在特定条件下效率高。这些算法各有优劣，实际应用中常根据需求选择或组合使用。

25　动手实现：一个简单的标记-清除 GC 器

我们将使用 C 语言实现一个基本的标记-清除垃圾回收器，目标是为一个简单的对象系统提供 GC 支持。首先，设计核心数据结构。定义一个 `Object` 结构体，包含类型、标记位以及指向其他对象的指针。例如：

```
typedef struct Object {
    int type; // 对象类型，例如用 0 表示整数，1 表示配对
    int marked; // 标记位，用于 GC 标记阶段
```

```
    struct Object* left; // 左子对象，模拟引用关系
    struct Object* right; // 右子对象，模拟引用关系
    struct Object* next; // 下一个对象，用于维护全局链表
} Object;
```

在这个结构中，type 字段标识对象类型，marked 用于标记阶段记录对象是否存活，left 和 right 模拟对象间的引用关系，next 用于将所有对象链接成一个链表，便于遍历。接下来，我们模拟一个虚拟机来管理 GC 状态。定义一个 VM 结构体：

```
typedef struct {
    Object* stack[STACK_SIZE]; // 栈数组，模拟根集合
    int stack_size; // 当前栈大小
    Object* first_object; // 指向第一个对象的指针，用于遍历所有对象
    size_t num_objects; // 已分配对象数量
    size_t max_objects; // GC 触发阈值，当 num_objects 达到此值时触发 GC
} VM;
```

stack 数组存储根对象，例如全局变量或局部变量；stack_size 记录栈中元素数量；first_object 指向对象链表的头部；num_objects 和 max_objects 用于控制 GC 触发条件。现在，实现核心函数。首先是分配函数 gc_alloc：

```
Object* gc_alloc(VM* vm, int type) {
    if (vm->num_objects >= vm->max_objects) {
        gc(vm); // 如果达到 GC 阈值，则触发垃圾回收
    }
    Object* obj = malloc(sizeof(Object));
    obj->type = type;
    obj->marked = 0;
    obj->left = NULL;
    obj->right = NULL;
    // 将新对象添加到链表头部
    obj->next = vm->first_object;
    vm->first_object = obj;
    vm->num_objects++;
    return obj;
}
```

这个函数在分配新对象前检查是否达到 GC 阈值，如果是则调用 GC 函数。它使用 malloc 分配内存，初始化对象字段，并将对象添加到全局链表中，同时更新对象计数。标记函数 mark 采用递归方式遍历对象图：

```
void mark(Object* object) {
    if (object == NULL || object->marked) return;
    object->marked = 1;
    mark(object->left); // 递归标记左子对象
```

```
    mark(object->right); // 递归标记右子对象
}
```

它检查对象是否为空或已标记，否则设置标记位并递归标记其子对象，确保所有可达对象都被标记。mark_all 函数从虚拟机的根集合开始标记：

```
void mark_all(VM* vm) {
    for (int i = 0; i < vm->stack_size; i++) {
        mark(vm->stack[i]);
    }
}
```

这个函数遍历栈中的所有根对象，对每个根对象调用 mark 函数，从而覆盖整个存活对象图。清除函数 sweep 负责回收未标记对象：

```
void sweep(VM* vm) {
    Object** obj = &vm->first_object;
    while (*obj) {
        if (!(*obj)->marked) {
            Object* unreached = *obj;
            *obj = unreached->next;
            free(unreached);
            vm->num_objects--;
        } else {
            (*obj)->marked = 0; // 重置标记位以备下次 GC
            obj = &(*obj)->next;
        }
    }
}
```

它遍历对象链表，如果对象未标记，则释放其内存并更新链表；否则重置标记位。GC 主函数 gc 组合这些步骤：

```
void gc(VM* vm) {
    mark_all(vm);
    sweep(vm);
}
```

这个函数简单地调用 mark_all 和 sweep，完成整个垃圾回收过程。为了测试，我们可以编写代码创建对象、构建引用关系（包括循环引用），然后强制触发 GC 并验证回收效果。例如，创建两个对象相互引用，然后移除根引用，触发 GC 后检查它们是否被正确回收。

26　现代 GC 的进阶与优化

在实际应用中，垃圾回收器远比我们实现的简单版本复杂。分代收集是一种常见优化，基于对象生命周期观察：绝大多数对象都是「朝生夕死」的。因此，堆被划分为年轻代和老年

代。新对象在年轻代创建，经历多次 GC 后仍存活的对象会晋升到老年代。年轻代使用复制算法进行频繁 GC，而老年代使用标记-清除或其他算法，减少 GC 频率。这提升了整体效率，例如在年轻代中，复制算法的快速分配和碎片避免特性得到充分利用。增量式与并发式 GC 则旨在缩短 Stop-The-World 暂停时间。增量式 GC 将 GC 工作分解为多个小步骤，与用户程序交替执行；并发式 GC 在后台线程运行，与用户程序线程同时执行，但需处理复杂的并发问题，如读写屏障以确保数据一致性。这些优化使得垃圾回收在现代系统中更加高效和透明。

垃圾回收自动化了内存管理，显著提升了软件可靠性和开发效率。我们实现的标记-清除 GC 器简单有效，能处理循环引用，但存在内存碎片和执行暂停等局限性。读者可以在此基础上扩展，例如实现复制算法以优化碎片问题，或添加多线程支持以探索并发 GC。在实际语言运行时中，垃圾回收是一个极其复杂的子系统，通常是多种算法的结合，需要根据应用场景精心调优。通过本次实践，我们希望读者不仅理解了垃圾回收的核心原理，还能激发进一步探索的兴趣，亲手打造更高效的内存管理器。