c13n #30

c13n

2025 年 11 月 19 日

第 I 部

基本的希尔排序（Shell Sort）算法

黄梓淳

Sep 08, 2

在计算机科学中，排序算法是基础且重要的主题。插入排序作为一种简单的排序方法，对于小规模或基本有序的数组表现出较高的效率，因为它通过逐个比较和移动元素来工作。然而，插入排序的局限性在于处理大规模乱序数组时，其时间复杂度为 $O(n^2)$，这主要是由于元素只能一位一位地移动，导致性能低下。为了解决这一问题，Donald Shell 在 1959 年提出了希尔排序，这是一种对插入排序的改进版本。希尔排序的核心思想是允许元素进行「长距离跳跃」，通过预处理使数组更快达到基本有序状态，从而显著提升排序效率。简而言之，希尔排序可以视为插入排序的「预排序」优化版。

1 希尔排序的核心思想与工作原理

希尔排序的工作原理基于一个关键概念：增量（Gap）。增量用于划分子序列，例如，当增量为 gap 时，数组会被分成多个子序列，每个子序列由间隔为 gap 的元素组成。具体来说，对于一个数组，当 gap=5 时，索引为 0、5、10 等的元素形成一个子序列，索引为 1、6、11 等的元素形成另一个子序列，以此类推。算法步骤包括：首先选择一个增量序列（如初始 gap 为 n/2），然后按当前增量将数组分割成子序列，并对每个子序列进行插入排序；之后缩小增量（如 gap/2），重复这个过程，直到增量为 1；最后，对整个数组进行一次插入排序，此时数组已基本有序，插入排序效率很高。

为了更直观地理解，让我们以一个具体数组为例进行演示。考虑数组 `[9，8，7，6，5，4，3，2，1]`，初始长度 n=9，因此初始 gap 为 4（n/2 取整）。首先，以 gap=4 划分子序列：子序列 1 包括索引 0、4、8 的元素（9, 5, 1），子序列 2 包括索引 1、5 的元素（8, 4），子序列 3 包括索引 2、6 的元素（7, 3），子序列 4 包括索引 3、7 的元素（6, 2）。对每个子序列进行插入排序后，数组变为 `[1，4，3，2，5，8，7，6，9]`。接下来，缩小 gap 到 2，重新划分子序列并排序，数组进一步优化。最后，gap=1 时进行最终插入排序，得到有序数组 `[1，2，3，4，5，6，7，8，9]`。这个过程展示了希尔排序如何通过大步距移动元素来加速排序。

2 关键问题：增量序列的选择

增量序列的选择对希尔排序的性能有显著影响。不同的序列会导致不同的时间复杂度。希尔原始序列使用简单的除法策略，如 gap = n/2, n/4, ..., 1，虽然易于理解和实现，但在最坏情况下时间复杂度仍为 $O(n^2)$，因此不是最优选择。为了提高性能，研究者提出了其他序列，例如 Hibbard 序列，其形式为 1, 3, 7, 15, ..., $2^k - 1$，这种序列可以将最坏情况时间复杂度优化到 $O(n^{3/2})$。另一个著名的是 Sedgewick 序列，它通过更复杂的计算提供更好的平均性能。在本文中，为了教学和实现简单，我们将使用希尔原始序列，即 gap 初始为 n/2，然后逐步缩小到 1。

3 代码实现

我们选择 Python 作为实现语言，因为其代码简洁易懂，适合教学目的。以下是希尔排序的 Python 实现，附有详细注释。

```
def shell_sort(arr):
    n = len(arr)
```

```
    # 初始增量 gap 设置为数组长度的一半
    gap = n // 2

    # 循环直到 gap 大于 0
    while gap > 0:
        # 从第 gap 个元素开始，对各个子序列进行插入排序
        for i in range(gap, n):
            temp = arr[i] # 当前需要插入的元素
            j = i
            # 在子序列中，进行插入排序：将比 temp 大的元素向后移动 gap 位
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            # 将 temp 插入到正确位置
            arr[j] = temp
        # 缩小增量
        gap //= 2
    return arr
```

现在，让我们逐行解析这段代码。函数 `shell_sort` 接受一个数组 `arr` 作为输入。首先，获取数组长度 `n`，并设置初始 `gap` 为 `n // 2`，即整数除法的一半。外层 `while` 循环控制增量的变化，只要 `gap` 大于 0 就继续执行。内层 `for` 循环从索引 `gap` 开始遍历数组，这是为了处理每个子序列的元素。变量 `temp` 存储当前需要插入的元素值，`j` 初始化为当前索引 `i`。内层 `while` 循环执行插入排序的核心操作：如果 `j` 大于等于 `gap` 且前一个子序列元素（索引 `j - gap`）大于 `temp`，则将较大的元素向后移动 `gap` 位，并递减 `j` 以继续比较。循环结束后，将 `temp` 插入到正确位置 `j`。完成内层循环后，缩小 `gap` 为原来的一半，重复过程直到 `gap` 为 0。最终返回排序后的数组。这段代码实现了希尔排序的基本逻辑，通过分组插入排序来优化性能。

4 算法分析

希尔排序的时间复杂度分析较为复杂，因为它依赖于所选的增量序列。使用希尔原始序列时，最坏情况时间复杂度为 $O(n^2)$，这与插入排序相同，但平均情况通常优于插入排序。如果使用优化序列如 Hibbard 序列，最坏情况可改善到 $O(n^{3/2})$，最佳情况甚至可以达到 $O(n \log n)$。总体而言，平均时间复杂度大约在 $O(n^{13/10})$ 到 $O(n^2)$ 之间，具体取决于数据分布和序列选择。空间复杂度方面，希尔排序是原地排序算法，只需要常数级的额外空间（即 $O(1)$），因此非常高效 in terms of memory usage。然而，希尔排序是不稳定的排序算法，原因在于相同的元素可能在子序列排序过程中被打乱相对顺序，例如，如果两个相同值的元素位于不同的子序列，它们的顺序可能因移动而改变。

希尔排序的优点包括：它是插入排序的高效改进版，算法简单易于实现，对于中等大小的数组表现良好，且空间复杂度低（$O(1)$）。这些特点使其在资源受限的环境中有一定应用价值。缺点则在于时间复杂度依赖于增量序列，分析复杂，在最坏情况下性能可能不佳，而且

不如快速排序、堆排序或归并排序等高级算法快，因此通常被视为教学算法而非生产环境的首选。

总之，希尔排序的核心思想是通过分组插入排序和逐步缩小增量来优化排序过程。它适用于中等规模、非性能临界的数据排序，例如在嵌入式系统或内存受限的环境中，由于其简单的实现和低空间开销，可能被采用。然而，在大多数现代应用中，更高效的算法如快速排序或归并排序更受青睐。希尔排序更多地作为一种教学工具，帮助我们理解算法改进的思路。

5 互动与思考题

读者可以尝试修改代码，使用不同的增量序列（如 `gap // 2 + 1`）并观察性能差异，这有助于深入理解增量序列的影响。同时，思考一个问题：「希尔排序为什么是不稳定的？你能举出一个例子吗？」例如，考虑数组 `[3, 2, 2, 1]` 使用 gap=2 进行排序，可能打乱相同元素的顺序。对于扩展阅读，推荐了解其他排序算法如快速排序和归并排序，以 broaden your knowledge。

第 II 部

基本的链表（Linked List）数据结构

叶家炜

Sep 09,

在计算机科学中，数据结构是组织和存储数据的基础。数组作为一种常见的数据结构，提供连续的内存空间和随机访问能力，时间复杂度为 $O(1)$。然而，数组存在局限性：大小固定，插入和删除元素时需要移动后续所有元素，导致时间复杂度为 $O(n)$，这可能造成效率低下和内存浪费。例如，预分配固定大小可能导致内存不足或闲置。为了解决这些问题，链表应运而生。链表是一种动态数据结构，物理存储非连续，逻辑顺序通过指针链接实现，允许高效插入和删除操作。本文将带您深入理解链表的原理，并通过代码实现掌握其核心操作。

6 核心概念：链表的构成

链表的基本构建块是节点（Node）。每个节点包含两个部分：数据域（data）和指针域（next）。数据域存储实际值，指针域存储指向下一个节点的引用。头指针（Head Pointer）是访问链表的入口点，指向第一个节点。如果头指针丢失，整个链表将无法访问，因为它没有其他引用方式。节点结构可以简单表示为 `[ data | next ]`，其中 next 指向下一个节点或 null，表示链表结束。

7 链表 vs. 数组：一场经典的博弈

链表和数组在特性上各有优劣，选择取决于具体应用场景。数组分配静态连续内存，大小固定，支持随机访问，时间复杂度为 $O(1)$，但插入和删除元素效率低，为 $O(n)$，因为需要移动元素。链表分配动态非连续内存，大小可动态调整，访问元素需顺序进行，时间复杂度为 $O(n)$，但插入和删除在已知位置时效率高，为 $O(1)$。空间开销方面，数组无额外指针开销，而链表每个节点都有指针开销。因此，如果应用需要频繁随机访问，数组更合适；如果需要频繁插入删除，链表更优。

8 单链表的基本操作

单链表是最简单的链表形式，我们将使用 Python 实现基本操作，包括定义类、遍历、插入、删除和搜索。每个操作都配以算法思路、代码实现和详细解读，并分析时间复杂度。

8.1 定义节点类和链表类

首先，定义节点类。每个节点有数据域和指向下一个节点的指针。

```
class Node:
    def __init__(self, data):
        self.data = data # 初始化数据域，存储用户提供的数据
        self.next = None # 初始化指针域，默认指向 None，表示无后续节点
```

这段代码创建了一个 Node 类，构造函数接受 data 参数并初始化 next 为 None。这确保了每个节点可以独立存在，并准备被链接。

接下来，定义链表类，包含头指针。

```
class LinkedList:
    def __init__(self):
        self.head = None # 初始化头指针为 None，表示空链表
```

链表类通过 head 属性管理整个链表。初始时链表为空，head 为 None。

8.2 遍历（Traversal）

遍历链表意味着从头指针开始，依次访问每个节点，直到遇到 null。算法思路是使用循环结构，从 head 出发，每次移动到 next 指针，直到 next 为 None。时间复杂度为 $O(n)$，因为需要访问每个节点一次。

```
def traverse(self):
    current = self.head # 从头指针开始，设置当前节点变量
    while current is not None: # 循环条件：当前节点不为空
        print(current.data) # 输出当前节点的数据，可根据需要处理数据
        current = current.next # 移动到下一个节点，更新当前节点引用
```

此代码通过 while 循环遍历链表。current 变量用于跟踪当前位置，循环继续直到 current 为 None。每一步输出数据，并更新 current 到 next。这确保了所有节点都被访问。

8.3 插入（Insertion）

插入操作分三种情况：在头部、尾部或指定节点后插入。每种情况有不同的时间复杂度。
在头部插入最高效，时间复杂度为 $O(1)$。只需创建新节点，将其 next 指向当前 head，然后更新 head。

```
def insert_at_head(self, data):
    new_node = Node(data) # 创建新节点实例
    new_node.next = self.head # 新节点的 next 指向当前头节点，链接到现有链
        ↪ 表
    self.head = new_node # 更新头指针指向新节点，使其成为新头
```

这段代码首先创建新节点，然后调整指针：新节点的 next 指向原 head，最后 head 更新为新节点。这确保了新节点插入到链表头部。
在尾部插入需遍历到最后一个节点，因此时间复杂度为 $O(n)$。找到尾节点（next 为 None），将其 next 指向新节点。

```
def insert_at_tail(self, data):
    new_node = Node(data) # 创建新节点
    if self.head is None: # 检查链表是否为空
        self.head = new_node # 如果为空，直接设置 head 为新节点
    else:
        current = self.head # 从头开始遍历
        while current.next is not None: # 循环直到找到最后一个节点（next 为
            ↪ None）
            current = current.next # 移动到下一个节点
        current.next = new_node # 将最后一个节点的 next 指向新节点，完成插入
```

代码首先处理空链表情况。如果不是空链表，则遍历到最后一个节点（current.next 为 None），然后设置其 next 为新节点。这确保了新节点添加到链表尾部。

在指定节点后插入，假设已知某个节点引用，时间复杂度为 $O(1)$。只需调整指针，无需遍历。

```
def insert_after(self, prev_node, data):
    if prev_node is None: # 检查前驱节点是否存在，避免错误
        print("Previous node cannot be None") # 输出错误信息
        return
    new_node = Node(data) # 创建新节点
    new_node.next = prev_node.next # 新节点的 next 指向 prev_node 的当前
        ↪ next
    prev_node.next = new_node # prev_node 的 next 更新为新节点，完成插入
```

这段代码首先验证 prev_node 不为 None。然后创建新节点，并调整指针：新节点的 next 指向 prev_node 的 next，prev_node 的 next 指向新节点。这实现了在指定节点后插入。

8.4 删除（Deletion）

删除操作也分三种情况：删除头节点、尾节点或指定值节点。每种情况有不同的复杂度。

删除头节点简单，时间复杂度为 $O(1)$。只需将 head 指向 head.next。

```
def delete_head(self):
    if self.head is None: # 检查链表是否为空
        return # 如果为空，直接返回，无操作
    self.head = self.head.next # 更新头指针指向下一个节点，原头节点被绕过
```

代码首先检查空链表。然后直接更新 head 为 head.next，这有效地删除了头节点，因为原头节点不再被引用。

删除尾节点需找到倒数第二个节点，因此时间复杂度为 $O(n)$。遍历到倒数第二个节点，将其 next 设为 None。

```
def delete_tail(self):
    if self.head is None: # 空链表检查
        return
    if self.head.next is None: # 检查是否只有一个节点
        self.head = None # 如果是，设置 head 为 None，链表变为空
        return
    current = self.head
    while current.next.next is not None: # 遍历直到倒数第二个节点
        current = current.next # 移动当前节点
    current.next = None # 设置倒数第二个节点的 next 为 None，删除尾节点
```

代码处理了空链表和单节点链表的情况。对于多节点链表，遍历到倒数第二个节点（current.next.next 为 None），然后设置其 next 为 None，从而删除尾节点。

删除指定值节点需找到该节点及其前驱节点，时间复杂度为 $O(n)$。遍历链表，比较数据，找到后调整指针。

```
def delete_value(self, key):
    current = self.head
    if current is not None and current.data == key: # 如果头节点匹配要删除的值
        self.head = current.next # 更新 head 指向下一个节点，删除头节点
        return
    prev = None # 用于跟踪前驱节点
    while current is not None and current.data != key: # 遍历寻找匹配节点
        prev = current # 保存当前节点为前驱
        current = current.next # 移动到下一个节点
    if current is None: # 如果未找到匹配节点
        return # 直接返回
    prev.next = current.next # 绕过当前节点，链接前驱节点的 next 到当前节点的 next
```

代码首先检查头节点是否匹配。如果不匹配，则遍历链表，使用 prev 变量记录前驱节点。找到匹配节点后，通过设置 prev.next 为 current.next 来删除当前节点。这确保了链表连续性。

8.5 搜索（Search）

搜索操作遍历链表，比较每个节点的数据与目标值，时间复杂度为 $O(n)$。

```
def search(self, key):
    current = self.head
    while current is not None: # 遍历整个链表
        if current.data == key: # 检查当前节点数据是否匹配
            return True # 找到匹配，返回 True
        current = current.next # 移动到下一个节点
    return False # 遍历结束未找到，返回 False
```

代码通过 while 循环遍历链表，逐个比较节点数据与 key。如果找到匹配，立即返回 True；否则，循环结束后返回 False。这实现了线性搜索。

9 常见的链表变体

除了单链表，还有其他变体如双向链表和循环链表。双向链表每个节点包含 prev、data 和 next 三部分，允许双向遍历，删除操作更高效，因为不需要寻找前驱节点，但空间开销更大。循环链表的尾节点 next 指向头节点，形成环状结构，适用于需要循环访问的场景，如操作系统中的轮询调度或约瑟夫问题。这些变体扩展了链表的应用范围，但增加了实现复杂度。

10 实战应用：链表在现实世界中的身影

链表在许多实际系统中扮演关键角色。它是高级数据结构的基础，例如栈和队列可以通过链表实现动态大小。在操作系统中，链表用于内存管理，如维护空闲内存块链表。文件系统目录结构常使用链表来组织文件和文件夹。浏览器历史记录功能常用双向链表实现前进和后退操作，因为双向遍历效率高。此外，LRU（最近最少使用）缓存淘汰算法结合哈希表和双向链表，实现 $O(1)$ 时间复杂度的存取操作，提升性能。这些应用展示了链表的实用性和灵活性。

链表作为一种动态数据结构，通过节点和指针实现非连续存储，支持高效插入和删除操作，但访问效率较低。本文详细介绍了单链表的原理和实现，包括遍历、插入、删除和搜索操作，并讨论了常见变体和实际应用。链表是学习更复杂数据结构如树和图的基石，鼓励读者动手实践，例如实现双向链表或解决实际问题如 LRU 缓存。未来，可以探索更多优化和变体，以应对不同场景的需求。

第 III 部

基本的循环链表（Circular Linked List）数据结构

马浩琨

Sep 10, 2025

在计算机科学中，数据结构的选择往往决定了算法的效率和应用的灵活性。想象一下，你在听音乐时使用循环播放功能，歌曲列表会无限重复；或者在操作系统中，CPU 使用轮询调度算法公平分配时间片给多个进程。这些场景都依赖于一种循环的逻辑，而普通的数据结构如单向链表，在到达尾部后无法直接返回头部，从而限制了其适用性。普通单向链表的尾节点指向 NULL，这表示链表的结束，但在循环场景中，我们需要一种能够无缝连接首尾的结构。这就是循环链表（Circular Linked List）登场的时候了。循环链表通过将尾节点指向头节点，形成一个闭环，解决了普通链表的局限性。本文将带领读者深入理解循环链表的核心概念，并通过代码实现其基本操作，旨在让读者真正掌握这一数据结构。

11 什么是循环链表？—— 核心概念剖析

循环链表是一种链式存储结构，其核心特征在于表中最后一个节点的指针域不再指向 NULL，而是指向头节点或第一个节点，从而形成一个环状结构。这种设计使得链表没有明确的开始或结束点，从任何节点出发都可以遍历整个链表。循环链表主要有两种类型：单向循环链表和双向循环链表。单向循环链表中，最后一个节点的 next 指针指向头节点；而双向循环链表则在此基础上，头节点的 prev 指针也指向最后一个节点，提供双向遍历的能力。本文将以单向循环链表为重点进行讲解，双向循环链表作为拓展内容稍后提及。

与普通单向链表相比，循环链表在结构和遍历方式上存在显著差异。普通单向链表的尾节点指向 NULL，结构是线性的，有明确的开始和结束；遍历时，终止条件是当前节点不为 NULL。而单向循环链表的尾节点指向头节点，结构是环形的，无明确的开始和结束；遍历时，终止条件是当前节点不等于头节点或当前节点的下一个节点不等于头节点，以避免无限循环。空链表的表示方式两者类似，通常用 head = NULL 表示，但在循环链表中，空链表也可以表示为头节点指向自身，不过这取决于具体实现。这些差异使得循环链表在处理循环性任务时更加高效和自然。

12 实现之旅：从零开始构建一个单向循环链表

首先，我们定义链表节点。节点结构与普通链表相同，包含数据域和指向下一个节点的指针。在 Python 中，我们可以定义一个 Node 类，其中 __init__ 方法初始化数据和 next 指针；在 Java 中，类似地定义一个 Node 类，使用构造函数设置数据和 next。代码示例如下：

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

```
class Node {
   int data;
   Node next;
   Node(int data) {
      this.data = data;
      this.next = null;
```

```
    }
}
```

初始化循环链表时，空链表 simply 用 head = None 或 head = null 表示。如果创建只有一个节点的循环链表，则该节点的 next 指针指向自身，形成自环。这确保了即使只有一个节点，链表也保持循环特性。

接下来，我们实现核心操作，包括遍历打印、在头部插入节点、在尾部插入节点和删除节点。每个操作都需要 careful 处理边界条件，如空链表或单节点链表，以避免错误或无限循环。

遍历打印操作的关键在于设置正确的终止条件。由于链表是循环的，我们不能依赖 NULL 来判断结束，而是需要检查是否回到了头节点。代码逻辑如下：从 head 开始，如果链表为空，直接返回；否则，打印当前节点数据，然后移动到下一个节点，直到再次遇到 head。这确保了遍历整个链表而不陷入无限循环。例如，在 Python 中，我们可以使用一个循环，条件是当前节点不为 None 且未回到起点，但更简单的方式是先检查空链表，然后从 head 开始遍历，直到 next 指针指向 head。

在头部插入节点时，步骤稍复杂。首先创建新节点。如果链表为空，新节点的 next 指向自身，并将 head 指向新节点。如果链表非空，需要找到尾节点（即 next 指向 head 的节点），然后将新节点的 next 指向当前 head，尾节点的 next 指向新节点，最后更新 head 为新节点。这个过程确保了新节点成为头节点，同时维护循环结构。图解上，可以想象为将新节点插入环的起点，并调整指针以闭合环。

在尾部插入节点类似，但效率较低，因为需要遍历到尾部。如果链表为空，等同于头部插入。否则，找到尾节点（其 next 指向 head），将尾节点的 next 指向新节点，新节点的 next 指向 head。这使新节点成为新的尾节点，保持循环。时间复杂度为 $O(n)$，因为需要遍历整个链表找到尾部，这与普通链表相同。

删除节点操作更具挑战性，尤其是处理边界情况。根据值删除节点时，首先检查链表是否为空。然后，找到要删除的节点 curr 及其前一个节点 prev。如果删除的是唯一节点，直接将 head 设为 null。如果删除的是头节点，需要找到尾节点，并更新尾节点的 next 指向新的头节点（即原头节点的下一个节点），然后更新 head。在一般情况，只需将 prev.next 设置为 curr.next，从而跳过 curr 节点。在非垃圾回收语言中，还需手动释放 curr 节点内存。这些步骤确保了删除后链表仍保持循环性。

13 循环链表的优势与应用场景

循环链表的优势主要体现在其环状结构上。从任何节点出发，都能遍历整个链表，这提高了灵活性和效率。例如，在实现队列时，循环链表可以仅用一个尾指针（tail）来管理，因为 tail.next 直接指向头节点，简化了入队和出队操作，无需维护额外的头指针。此外，循环链表天然适合模拟循环场景，如轮询调度或游戏回合制，减少了代码复杂性。

在应用场景方面，循环链表广泛应用于操作系统、多媒体和游戏开发。在操作系统中，CPU 时间片轮转调度算法使用循环链表来管理进程队列，确保每个进程公平获得执行时间。在多媒体应用中，循环播放列表利用循环链表来实现歌曲的无限循环。游戏开发中，玩家回合制系统可以通过循环链表轻松实现玩家顺序的循环。此外，循环链表作为基础数据结构，用于实现更高级的结构如循环队列和斐波那契堆，这些在算法优化中至关重要。

本文深入探讨了循环链表的核心概念、实现细节以及应用优势。循环链表通过首尾相连的设计，解决了普通链表在循环场景中的局限性，其关键在于指针操作和边界条件处理。读者在实现时，务必注意遍历的终止条件，避免无限循环，并仔细处理插入和删除操作中的特殊情况。

展望未来，双向循环链表提供了更大的灵活性，允许前后双向遍历，从而优化插入和删除操作。例如，在双向循环链表中，删除节点无需查找前驱节点，直接通过 prev 指针即可完成，时间复杂度可降至 $O(1)$ 在某些情况下。鼓励学有余力的读者探索双向循环链表的实现，以进一步扩展数据结构知识。

14 附录/练习

以下是完整的 Python 实现代码，包括节点定义和基本操作。读者可以运行此代码进行实验。

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

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def traverse(self):
        if self.head is None:
            print("链表为空")
            return
        current = self.head
        while True:
            print(current.data, end="␣->␣")
            current = current.next
            if current == self.head:
                break
        print("(头节点)")

    def insert_at_head(self, data):
        new_node = Node(data)
        if self.head is None:
            new_node.next = new_node
            self.head = new_node
        else:
            tail = self.head
```

```
        while tail.next != self.head:
            tail = tail.next
        new_node.next = self.head
        tail.next = new_node
        self.head = new_node

    def insert_at_tail(self, data):
        new_node = Node(data)
        if self.head is None:
            self.insert_at_head(data)
        else:
            tail = self.head
            while tail.next != self.head:
                tail = tail.next
            tail.next = new_node
            new_node.next = self.head

    def delete_node(self, key):
        if self.head is None:
            return
        current = self.head
        prev = None
        while True:
            if current.data == key:
                if current.next == self.head: # 如果只有一个节点或删除尾节点
                    if prev is None: # 删除头节点且是唯一节点
                        self.head = None
                    else:
                        prev.next = self.head
                else:
                    if prev is None: # 删除头节点但有多个节点
                        tail = self.head
                        while tail.next != self.head:
                            tail = tail.next
                        tail.next = current.next
                        self.head = current.next
                    else:
                        prev.next = current.next
                return
            prev = current
            current = current.next
            if current == self.head:
```

```
            break
```

给读者的挑战：尝试实现双向循环链表，并思考其与单向版本的性能差异。另外，探索仅使用一个指向尾节点的指针（tail）来实现循环链表，并实现基本操作；比较这种设计与使用 head 指针的优劣，例如在插入和删除操作中的效率变化。

第 IV 部

基本的栈（Stack）数据结构

李睿远

Sep 11, 2025

栈是计算机科学中最基础且无处不在的数据结构之一。想象一下日常生活中叠盘子的场景：我们总是将新盘子放在最上面，取用时也从最上面开始拿。这种后进先出的行为正是栈的核心理念。在计算机领域，栈的应用广泛而关键，例如函数调用栈管理着程序的执行流程，浏览器中的「后退」按钮依赖于栈来记录历史页面，甚至表达式求值和撤销操作都离不开栈的支持。本文将带领您从理论层面深入理解栈的概念，并通过代码实现两种常见的栈结构，最后探讨其经典应用场景。

15 栈的核心概念剖析

栈是一种线性数据结构，其操作遵循后进先出（LIFO, Last-In, First-Out）原则。这意味着最后一个被添加的元素将是第一个被移除的。栈的核心操作包括入栈（Push）和出栈（Pop）。入栈操作将一个新元素添加到栈顶，而出栈操作则移除并返回栈顶元素。此外，栈还支持一些辅助操作，例如查看栈顶元素（peek 或 top）、检查栈是否为空（isEmpty）、检查栈是否已满（isFull，仅适用于基于数组的实现）以及获取栈的大小（size）。从抽象数据类型（ADT）的角度来看，栈可以定义为支持这些操作的一个集合，其接口确保了数据访问的严格顺序性。

16 栈的实现方式（一）：基于数组

基于数组的实现是栈的一种常见方式，其思路是使用一个固定大小的数组来存储元素，并通过一个称为 top 的指针来跟踪栈顶的位置。初始化时，top 通常设置为 -1，表示栈为空。当执行入栈操作时，top 递增，并将新元素存储在数组的相应索引处；出栈操作则返回 top 指向的元素，并将 top 递减。这种实现的关键在于处理边界情况，例如当栈为空时尝试出栈会引发错误，而当栈满时尝试入栈会导致溢出。

以下是一个用 Python 实现的基于数组的栈示例代码：

```
class ArrayStack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = [None] * capacity
        self.top = -1

    def push(self, item):
        if self.is_full():
            raise Exception("Stack is full")
        self.top += 1
        self.stack[self.top] = item

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        item = self.stack[self.top]
        self.top -= 1
```

```
        return item

    def peek(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.stack[self.top]

    def is_empty(self):
        return self.top == -1

    def is_full(self):
        return self.top == self.capacity - 1

    def size(self):
        return self.top + 1
```

在这段代码中，我们定义了一个 ArrayStack 类，其初始化方法接受一个容量参数，并创建一个相应大小的数组。push 方法首先检查栈是否已满，如果未满，则递增 top 并将元素存入数组；pop 方法检查栈是否为空，然后返回栈顶元素并递减 top。peek 方法类似，但不修改栈。所有核心操作的时间复杂度均为 O(1)，因为它们只涉及简单的索引操作。空间复杂度为 O(n)，其中 n 是数组的容量。基于数组的栈实现简单高效，但缺点在于容量固定，可能发生栈溢出。

17　栈的实现方式（二）：基于链表

基于链表的栈实现提供了动态扩容的能力，其思路是使用单链表来存储元素，并将链表的头部作为栈顶。这样，入栈操作相当于在链表头部插入新节点，出栈操作则是移除头部节点。这种实现无需预先分配固定大小，因此更适合不确定数据量的场景。每个节点包含数据和指向下一个节点的指针，栈本身只需维护一个指向头部的指针。

以下是一个用 Python 实现的基于链表的栈示例代码：

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

class LinkedListStack:
    def __init__(self):
        self.head = None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.head
```

```
        self.head = new_node

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        item = self.head.data
        self.head = self.head.next
        return item

    def peek(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.head.data

    def is_empty(self):
        return self.head is None

    def size(self):
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next
        return count
```

在这段代码中，我们首先定义了一个 Node 类来表示链表节点，每个节点包含数据和一个指向下一个节点的指针。LinkedListStack 类的初始化方法将 head 指针设为 None，表示空栈。push 方法创建新节点并将其插入到链表头部；pop 方法检查栈是否为空，然后返回头部数据并更新 head 指针。peek 方法类似，但不修改链表。所有核心操作的时间复杂度均为 O(1)，因为链表头部的操作是常数时间的。空间复杂度为 O(n)，每个元素需要额外的指针空间。基于链表的栈优点在于动态容量，但缺点是需要更多内存用于指针，实现稍复杂。

18 两种实现方式的对比与选择

在选择栈的实现方式时，需要根据应用场景权衡利弊。数组实现具有固定容量，性能稳定且无内存分配开销，但可能发生栈溢出；链表实现则支持动态扩容，无需担心栈满，但每个元素需要额外指针空间，且操作可能有微小内存分配开销。总体而言，如果能预估数据量上限且追求极致性能，数组实现是更好的选择；如果需要处理不确定大小的数据，链表实现更灵活。在实际开发中，还应考虑语言特性和库支持，例如在 Python 中，列表本身就可以模拟栈，但理解底层实现有助于优化和调试。

19　栈的经典应用场景实战

栈在计算机科学中有许多经典应用，其中之一是括号匹配检查。这个问题要求检查一个字符串中的括号（如 (), [], {}）是否正确匹配和闭合。算法思路是遍历字符串，遇到左括号时入栈，遇到右括号时与栈顶左括号匹配，如果匹配则出栈，否则返回错误。最终，栈应为空表示所有括号匹配。以下是核心代码片段：

```
def is_balanced(expression):
    stack = []
    mapping = {')': '(', ']': '[', '}': '{'}
    for char in expression:
        if char in mapping.values():
            stack.append(char)
        elif char in mapping.keys():
            if not stack or stack.pop() != mapping[char]:
                return False
    return not stack
```

这段代码使用一个列表模拟栈，遍历表达式时处理括号。时间复杂度为 O(n)，其中 n 是表达式长度。

另一个应用是表达式求值， specifically 逆波兰表达式（后缀表达式）。例如，表达式 [2, 1, +, 3, *] 等价于 (2 + 1) * 3。算法思路是遍历表达式，遇到数字时入栈，遇到运算符时弹出两个操作数进行计算，并将结果入栈。最终栈顶即为结果。以下是核心代码片段：

```
def eval_rpn(tokens):
    stack = []
    for token in tokens:
        if token in "+-*/":
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(int(a / b))
        else:
            stack.append(int(token))
    return stack.pop()
```

这段代码处理数字和运算符，利用栈进行中间结果存储。时间复杂度为 O(n)。

栈还广泛应用于函数调用栈（Call Stack），这是编程语言中管理函数调用、局部变量和返回地址的核心机制。每当函数被调用时，其信息被压入栈；函数返回时，信息被弹出。这确保了程序的正确执行流程，是栈最基础的应用之一。

通过本文，我们深入探讨了栈的核心特性、两种实现方式及其经典应用。栈作为一种后进先出的数据结构，在计算机科学中扮演着不可或缺的角色。基于数组的实现简单高效，适合固定容量场景；基于链表的实现动态灵活，适合不确定数据量的情况。经典应用如括号匹配和表达式求值展示了栈的实际价值。作为进阶思考，读者可以尝试用栈来实现队列，或者设计一个支持 O(1) 时间获取最小元素的栈（Min Stack）。此外，栈内存与堆内存的区别以及深度优先搜索（DFS）算法中栈的应用也是值得延伸阅读的主题。

20 互动环节

欢迎读者在评论区分享您对栈的理解或实现代码。例如，您可以尝试用栈来反转一个字符串：遍历字符串并将每个字符入栈，然后出栈即可得到反转结果。这是一个简单的练习，有助于巩固栈的操作。如果您有任何问题或想法，请随时留言讨论！

第 V 部

浏览器中 JavaScript 定时器的节流机制解析

叶家炜

Sep 12, 2025

在开发 Web 应用时，许多开发者会遇到一个令人困惑的现象：使用 setInterval 实现的倒计时计时器，当用户切换到其他浏览器标签页一段时间后返回，发现计时器的时间似乎“变慢”或出现了“跳秒”。这并不是代码中的 Bug，而是浏览器的一种主动优化行为，称为定时器节流（Timer Throttling）。本文旨在帮助读者理解这一机制的原理，学会如何诊断相关问题，并掌握在必要时绕过节流或适应节流的最佳实践。

21 浏览器为何要“节流”定时器?

浏览器对定时器进行节流主要是出于性能优化和资源节约的考虑。频繁的定时器回调会阻止 CPU 进入空闲状态，从而增加功耗，尤其是在移动设备上，这会显著影响电池寿命。此外，浏览器需要确保用户正在交互的前台页面获得最充足的系统资源，以保持流畅的用户体验。因此，被隐藏或最小化的页面被视为低优先级，其任务应被降级处理，以避免不必要的资源消耗。这种优化行为有助于整体系统效率的提升，符合现代 Web 应用对性能的严格要求。

22 节流机制的核心原理与表现

定时器节流主要影响 setTimeout 和 setInterval，当页面处于后台标签页或最小化状态时触发。不同浏览器有不同的节流策略。在 Chromium 内核的浏览器（如 Chrome、Edge）中，延迟时间被限制为至少 1 秒（1000 毫秒），如果定时器设置了嵌套，延迟时间至少为 4 秒（4000 毫秒）。例如，一个设置了 setInterval(fn, 100) 的页面在后台时，fn 最多每秒执行一次。Firefox 的行为类似，后台标签页中的超时延迟至少为 1 秒。Safari 则采取更激进的策略，延迟可能延长到几分钟。需要注意的是，前台页面不受影响，requestAnimationFrame 由于与屏幕刷新率挂钩，在页面不可见时不会执行，因此也不被节流。Web Worker 运行在独立线程中，通常不受主页面节流策略的影响，这为解决方案提供了关键途径。

23 如何检测与调试定时器节流?

开发者可以使用浏览器开发者工具来检测定时器节流。在 Performance 面板中，录制性能时间线并观察 Timer Fired 事件的间隔，在后台阶段会看到间隔明显变大。另一种简单的方法是在 Console 面板中，在定时器回调中打印 Date.now() 时间戳，然后切换到其他标签页再返回，观察输出间隔的变化。例如，以下代码可以帮助调试：

```
setInterval(() => {
  console.log('Timestamp:', Date.now());
}, 100);
```

这段代码每隔 100 毫秒打印当前时间戳。当页面切换到后台时，输出间隔会变为至少 1 秒，从而直观地展示节流效果。此外，Page Lifecycle API 提供了 document.visibilityState 属性和 visibilitychange 事件，可以用来感知页面是否可见。例如：

```
document.addEventListener('visibilitychange', () => {
  if (document.visibilityState === 'visible') {
```

```
    console.log('Page is visible');
  } else {
    console.log('Page is hidden');
  }
});
```

这段代码监听页面可见性变化事件，当页面隐藏或显示时输出相应信息，帮助开发者判断定时器是否被节流。

24　应对策略：我们需要并如何绕过节流?

在考虑绕过节流之前，首先需要问自己是否真的有必要。大多数情况下，浏览器的节流行为是合理且有益的。如果需要精确计时，首选方案是使用 Web Worker。Web Worker 运行在独立线程中，不受主线程节流影响。以下是一个示例代码，展示如何创建 Worker 并在其中运行 setInterval：

```
// 主线程代码
const worker = new Worker('worker.js');
worker.postMessage('start');
worker.onmessage = (e) => {
  if (e.data === 'tick') {
    // 处理计时事件
    console.log('Tick received');
  }
};

// worker.js
self.addEventListener('message', (e) => {
  if (e.data === 'start') {
    setInterval(() => {
      self.postMessage('tick');
    }, 100);
  }
});
```

在这个代码中，主线程创建一个 Worker 并发送消息启动定时器。Worker 中的 setInterval 会以精确的间隔执行，即使主页面在后台。Worker 通过 postMessage 与主线程通信，传递计时事件，从而避免了节流影响。

另一种方案是基于 visibilitychange 事件的补偿策略，适用于倒计时等场景。当页面重新可见时，计算隐藏时长并调整计时器。示例代码：

```
let startTime = Date.now();
let elapsedTime = 0;
let timer;
```

```

document.addEventListener('visibilitychange', () => {
  if (document.visibilityState === 'hidden') {
    clearInterval(timer);
    elapsedTime = Date.now() - startTime;
  } else {
    startTime = Date.now() - elapsedTime;
    timer = setInterval(updateTimer, 1000);
  }
});

function updateTimer() {
  const currentTime = Date.now() - startTime;
  console.log('Elapsed time:', currentTime);
}
```

这段代码监听页面可见性变化事件。当页面隐藏时，清除定时器并记录已过时间；当页面可见时，重新设置定时器并补偿丢失的时间，确保计时准确性。

还可以使用 requestAnimationFrame 模拟间隔，但页面隐藏时不会执行，因此不是绕过而是适应。例如：

```
let lastTime = 0;
const interval = 100; // 目标间隔毫秒

function loop(timestamp) {
  const delta = timestamp - lastTime;
  if (delta >= interval) {
    // 执行业务逻辑
    console.log('Executing at interval');
    lastTime = timestamp;
  }
  requestAnimationFrame(loop);
}

requestAnimationFrame(loop);
```

这里，requestAnimationFrame 用于在页面可见时循环，通过计算时间差 Δt 来控制执行频率，其中 Δt 表示当前帧与上一帧的时间差。当 $\Delta t \geq 100$ 毫秒时，执行回调，否则继续循环。这种方式适用于动画等场景，但后台时自动暂停。

特定场景下，播放音频或视频可能避免节流，但这并非可靠方案，仅作为了解。

尊重浏览器的节流策略，默认情况下不要盲目绕过节流。区分不同场景：需要精确计时的后台任务使用 Web Worker；状态同步类任务使用 visibilitychange 事件补偿；页面动画使用 requestAnimationFrame。避免在后台执行不必要的密集任务，以保护用户设备的

性能和电池寿命。总之，定时器节流是浏览器以用户体验为中心的优化，开发者应理解并优雅地适应它，或在必要时使用高级 API 如 Web Worker。通过合理的设计，可以在满足功能需求的同时，保持应用的高效和友好。