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。

1　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。

2　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。

3 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。

4 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。