循环链表（Circular Linked List）数据结构

黄梓淳

Sep 04, 2025

在日常生活中，我们经常遇到循环的场景，比如环形跑道上的跑步者、圆桌会议中的参与者轮流发言，这些场景都体现了循环的连续性。在计算机科学中，数据结构也需要模拟这种循环特性，这就是循环链表诞生的背景。首先，让我们回顾一下单链表。单链表是一种线性数据结构，每个节点包含数据和指向下一个节点的指针，但它的尾部节点指向 NULL，这意味着无法直接从尾部快速访问头部或其他节点，在某些操作中效率较低，例如在尾部插入或删除时可能需要遍历整个链表。

循环链表的核心思想是将链表的头尾相连，形成一个环状结构。这种设计解决了单链表的一些局限性，例如在约瑟夫问题、轮询调度算法或多人游戏循环中，循环链表可以提供更高效的解决方案。本文将带你深入理解循环链表的概念、特性、实现方式、关键操作以及应用场景，并通过代码示例帮助你掌握其实现。

1　初窥门径：什么是循环链表?

循环链表是一种特殊的链表结构，其中最后一个节点的指针域指向头节点（或第一个节点），从而形成一个闭环。与普通链表不同，循环链表没有真正的头尾之分，任何节点都可以作为遍历的起点。这使得从任意节点出发都能访问整个链表，增强了灵活性。

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，每个节点只包含一个指向下一个节点的指针；而双向循环链表中，每个节点还包含一个指向前一个节点的指针，允许双向遍历。本文将重点探讨单向循环链表的实现，并在后续部分简要介绍双向循环链表。

由于无法使用图片，我们可以用文字描述循环链表的可视化：想象一个节点序列，其中每个节点指向下一个，最后一个节点指向第一个节点，形成一个环形结构。例如，在单向循环链表中，节点 A 指向节点 B，节点 B 指向节点 C，节点 C 指回节点 A，如此循环。

2　庖丁解牛：实现单向循环链表（Singly Circular Linked List）

2.1　节点结构定义 (Node Class)

在实现单向循环链表之前，我们需要定义节点的结构。节点通常包含两个部分：数据域和指针域。数据域存储实际的数据，指针域指向下一个节点。以下是一个示例代码，使用 Java 和 Python 语言。

```
// Java 实现节点类
class Node {
   int data;
   Node next;
```

```
    Node(int data) {
        this.data = data;
        this.next = null; // 初始化时指针指向 null，后续在插入操作中形成环
    }
}
```

```
# Python 实现节点类
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

在这段代码中，我们定义了一个 Node 类，其中 data 字段存储整型数据（在 Python 中可以是任何类型），next 字段初始化为 None 或 null，表示暂时没有指向其他节点。在循环链表的插入操作中，我们会调整 next 指针以形成循环。这种设计确保了节点的灵活性，便于后续操作。

2.2 链表类框架 (LinkedList Class)

接下来，我们定义链表类来管理节点。对于单向循环链表，通常使用一个 tail 指针指向尾节点，而不是 head 指针。这是因为通过 tail.next 可以快速访问头节点，从而简化某些操作，例如在尾部插入节点的时间复杂度可以降低到 $O(1)$。以下是链表类的基本框架。

```
// Java 实现链表类
public class CircularLinkedList {
    private Node tail; // 尾节点指针
    // 构造函数初始化链表为空
    public CircularLinkedList() {
        this.tail = null;
    }
}
```

```
# Python 实现链表类
class CircularLinkedList:
    def __init__(self):
        self.tail = None # 尾节点指针
```

这里，我们只维护一个 tail 指针。当链表为空时，tail 为 None；当链表非空时，tail 指向尾节点，而 tail.next 指向头节点。这种设计优化了尾部操作，但需要注意在插入和删除时维护循环性。

2.3 核心操作详解

2.3.1 判断链表是否为空 (isEmpty)

判断链表是否为空是一个简单但重要的操作，它检查 `tail` 指针是否为 `null`。如果为空，表示链表没有节点；否则，链表至少有一个节点。代码实现如下。

```
// Java 实现 isEmpty 方法
public boolean isEmpty() {
    return tail == null;
}
```

```
# Python 实现 is_empty 方法
def is_empty(self):
    return self.tail is None
```

这段代码通过检查 `tail` 是否为空来返回布尔值。时间复杂度为 $O(1)$，因为它只涉及指针比较。在实际应用中，这个操作常用于前置检查，避免在空链表上执行无效操作。

2.3.2 在链表尾部插入节点 (insertAtEnd)

在尾部插入节点是循环链表的常见操作。我们需要处理两种场景：链表为空和非空。如果链表为空，新节点将自环（即 `next` 指向自身），并成为尾节点；如果链表非空，新节点插入到尾节点之后，并更新尾指针。以下是代码实现。

```
// Java 实现 insertAtEnd 方法
public void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (isEmpty()) {
        newNode.next = newNode; // 自环
        tail = newNode;
    } else {
        newNode.next = tail.next; // 新节点指向头节点
        tail.next = newNode; // 原尾节点指向新节点
        tail = newNode; // 更新尾指针
    }
}
```

```
# Python 实现 insert_at_end 方法
def insert_at_end(self, data):
    new_node = Node(data)
    if self.is_empty():
        new_node.next = new_node # 自环
```

```
        self.tail = new_node
    else:
        new_node.next = self.tail.next # 新节点指向头节点
        self.tail.next = new_node # 原尾节点指向新节点
        self.tail = new_node # 更新尾指针
```

在这段代码中，我们首先创建新节点。如果链表为空，新节点的 next 指向自身，形成自环，并设置 tail 为新节点。如果链表非空，新节点的 next 指向当前头节点（通过 tail.next 访问），然后原尾节点的 next 指向新节点，最后更新 tail 为新节点。这个过程确保了循环性，时间复杂度为 $O(1)$，因为它不需要遍历。

2.3.3 在链表头部插入节点 (insertAtFront)

头部插入操作类似尾部插入，但不需要更新尾指针（除非链表为空）。如果链表为空，操作与尾部插入相同；否则，新节点插入到头节点之前，并调整指针以维持循环。代码实现如下。

```
// Java 实现 insertAtFront 方法
public void insertAtFront(int data) {
    Node newNode = new Node(data);
    if (isEmpty()) {
        newNode.next = newNode;
        tail = newNode;
    } else {
        newNode.next = tail.next; // 新节点指向当前头节点
        tail.next = newNode; // 尾节点指向新节点，使其成为新头
    }
}
```

```
# Python 实现 insert_at_front 方法
def insert_at_front(self, data):
    new_node = Node(data)
    if self.is_empty():
        new_node.next = new_node
        self.tail = new_node
    else:
        new_node.next = self.tail.next # 新节点指向当前头节点
        self.tail.next = new_node # 尾节点指向新节点，使其成为新头
```

这里，如果链表为空，我们进行自环设置；否则，新节点的 next 指向当前头节点（tail.next），然后更新尾节点的 next 指向新节点，从而使新节点成为头节点。注意，尾指针 tail 没有改变，因为头节点变化不影响尾节点。时间复杂度为 $O(1)$。

2.3.4 删除头节点 (deleteFromFront)

删除头节点涉及调整指针以移除头节点，并维护循环性。场景包括链表为空、只有一个节点或多个节点。如果链表为空，直接返回；如果只有一个节点，将尾指针置空；否则，将尾节点的 next 指向头节点的下一个节点。代码实现如下。

```java
// Java 实现 deleteFromFront 方法
public void deleteFromFront() {
    if (isEmpty()) {
        System.out.println("链表为空，无法删除");
        return;
    }
    if (tail.next == tail) { // 只有一个节点
        tail = null;
    } else {
        tail.next = tail.next.next; // 跳过头节点
    }
}
```

```python
# Python 实现 delete_from_front 方法
def delete_from_front(self):
    if self.is_empty():
        print("链表为空，无法删除")
        return
    if self.tail.next == self.tail: # 只有一个节点
        self.tail = None
    else:
        self.tail.next = self.tail.next.next # 跳过头节点
```

在这段代码中，我们首先检查链表是否为空。如果只有一个节点，直接设置 tail 为 None 以清空链表；否则，通过 tail.next.next 跳过当前头节点，使尾节点直接指向新的头节点。时间复杂度为 $O(1)$，因为它只涉及指针调整。

2.3.5 删除尾节点 (deleteFromEnd)

删除尾节点是循环链表中的难点，因为单向链表无法直接访问前驱节点，需要遍历找到尾节点的前一个节点。场景包括链表为空、只有一个节点或多个节点。代码实现如下。

```java
// Java 实现 deleteFromEnd 方法
public void deleteFromEnd() {
    if (isEmpty()) {
        System.out.println("链表为空，无法删除");
        return;
```

```
    }
    if (tail.next == tail) { // 只有一个节点
        tail = null;
    } else {
        Node current = tail.next;
        while (current.next != tail) {
            current = current.next; // 遍历找到尾节点的前一个节点
        }
        current.next = tail.next; // 前一个节点指向头节点
        tail = current; // 更新尾指针
    }
}
```

```
# Python 实现 delete_from_end 方法
def delete_from_end(self):
    if self.is_empty():
        print("链表为空，无法删除")
        return
    if self.tail.next == self.tail: # 只有一个节点
        self.tail = None
    else:
        current = self.tail.next
        while current.next != self.tail:
            current = current.next # 遍历找到尾节点的前一个节点
        current.next = self.tail.next # 前一个节点指向头节点
        self.tail = current # 更新尾指针
```

这里，如果链表为空或只有一个节点，处理方式与删除头节点类似。对于多个节点，我们从头节点开始遍历，直到找到尾节点的前一个节点（即 `current.next == tail`），然后调整指针：将前一个节点的 `next` 指向头节点，并更新 `tail` 为前一个节点。时间复杂度为 $O(n)$，其中 n 是链表长度，因为需要遍历。

2.3.6 遍历链表 (display / traverse)

遍历循环链表时，终止条件不再是检查 `null`，而是检查是否回到起点。通常使用 `do-while` 循环来确保至少执行一次。代码实现如下。

```
// Java 实现 display 方法
public void display() {
    if (isEmpty()) {
        System.out.println("链表为空");
        return;
    }
```

```
    Node current = tail.next; // 从头节点开始
    do {
        System.out.print(current.data + " -> ");
        current = current.next;
    } while (current != tail.next); // 当再次回到头节点时停止
    System.out.println("( back to head )");
}
```

```
# Python 实现 display 方法
def display(self):
    if self.is_empty():
        print("链表为空")
        return
    current = self.tail.next # 从头节点开始
    while True:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.tail.next:
            break
    print("( back to head )")
```

在这段代码中，我们从头节点（tail.next）开始，逐个打印节点数据，直到再次遇到头节点为止。使用 do-while 结构（在 Python 中用 while True 和 break 模拟）确保至少打印一次。时间复杂度为 $O(n)$，因为它需要访问每个节点一次。

3 进阶探讨：双向循环链表简介

双向循环链表是循环链表的扩展，每个节点包含两个指针：next 指向后继节点，prev 指向前驱节点。这种结构允许双向遍历，并优化了一些操作。例如，删除尾节点的时间复杂度可以从 $O(n)$ 降低到 $O(1)$，因为可以直接通过 tail.prev 访问前驱节点，无需遍历。

然而，双向循环链表的实现更复杂，因为插入和删除操作需要维护两个指针（next 和 prev），代码量增加，但提供了更大的灵活性。在实际应用中，双向循环链表常用于需要频繁前后遍历的场景，如浏览器历史记录或高级数据结构的基础。

4 实际应用：循环链表用在哪里?

循环链表在计算机科学中有广泛的应用。在操作系统中，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 CPU 时间。在多媒体应用中，循环链表用于实现循环播放功能，如音乐播放器中的歌单循环。在游戏开发中，它可以模拟玩家回合制循环，例如棋类游戏中的轮流行动。此外，循环链表作为基础数据结构，常用于实现队列，其中入队和出队操作都可以在 $O(1)$ 时间内完成，如果维护了尾指针。

循环链表的优点包括从任意节点出发都能遍历整个链表，以及某些操作（如尾部插入）的高效性。但它也有缺

点，例如实现稍复杂，容易产生无限循环的 bug，需要谨慎处理边界条件。与单链表相比，循环链表在插入和删除操作上可能更高效（如果优化了指针），但遍历操作类似。总体而言，循环链表适用于需要循环访问的场景，而单链表更适用于线性数据处理。

5 练习与思考

为了巩固学习，建议尝试解决约瑟夫环问题，这是一个经典问题，可以用循环链表来模拟 elimination 过程。此外，思考如何检测一个链表是否是循环链表？快慢指针法是一种常见解决方案：使用两个指针，一个移动快，一个移动慢，如果它们相遇，则存在环。最后，挑战自己实现双向循环链表，以加深对指针操作的理解。

6 结束语

本文详细介绍了循环链表的概念、实现和应用。通过代码示例和解读，我们希望帮助你掌握这一数据结构。动手实现是学习的关键，鼓励你编写代码并尝试解决提出的问题。在下一篇文章中，我们可能会探讨双向循环链表的详细实现或其他高级话题。 Happy coding!