循环链表

黄京

Jul 10, 2025

在数据结构领域，单链表是一种基础且广泛使用的线性结构。然而，单链表存在一个显著局限性：尾节点操作效率低下。例如，在单链表中插入或删除尾节点时，必须从头节点开始遍历整个链表，时间复杂度为 $O(n)$，其中 n 为节点数量。这种效率问题在需要频繁操作尾部的场景中尤为突出。循环链表的核心理念正是通过构建闭环结构来解决这一边界问题。其本质是将尾节点的指针指向头节点，形成一个无始无终的环。这种设计消除了单链表的“终点”概念，使得头尾操作变得高效。典型应用场景包括操作系统进程调度中的轮询算法、游戏开发中的角色循环队列，以及音频流处理中的数据缓冲区。在这些场景中，循环链表的环形特性天然支持连续遍历和高效拼接。

1　循环链表基础解析

循环链表的核心在于其闭环结构。在单向循环链表中，尾节点的 next 指针指向头节点；而双向循环链表则增加了 prev 指针，实现双向闭环。关键特性是空链表的表示方式：当链表为空时，头指针满足 head→next = head。这与单链表使用 NULL 表示空节点形成本质区别。遍历循环链表时，终止条件不再是 current != NULL，而是 current != head。这意味着遍历从任意节点开始，最终会返回起点。插入或删除头节点时，指针维护逻辑也不同于单链表。例如，删除头节点需修改尾节点的指针以维持闭环，否则会导致结构断裂。

2　循环链表的操作实现（附 C 代码）

实现循环链表的第一步是定义节点结构。以下代码展示了节点定义和初始化函数：

```c
typedef struct Node {
    int data; // 数据域，存储整数值
    struct Node* next; // 指针域，指向下一个节点
} Node;

Node* create_node(int data) {
    Node* new_node = (Node*)malloc(sizeof(Node)); // 动态分配内存
    new_node->data = data; // 设置数据值
    new_node->next = new_node; // 初始化自环，确保新节点指向自身
    return new_node; // 返回新节点指针
}
```

这段代码创建了一个新节点，并通过 new_node→next = new_node 实现自环初始化。这是循环链表的基础，确保单个节点也能形成闭环。

核心操作包括插入、删除和遍历。在空链表插入时，直接将头指针指向新节点：head = new_node;。头插法操作如下：

```
new_node->next = head->next; // 新节点指向原头节点的下一个节点
head->next = new_node; // 头节点指向新节点，完成插入
```

此操作在 $O(1)$ 时间内完成。尾插法则需定位尾节点：

```
Node* tail = head;
while (tail->next != head) { // 遍历至尾节点
    tail = tail->next;
}
tail->next = new_node; // 尾节点指向新节点
new_node->next = head; // 新节点指向头节点，维持闭环
```

尾插法的时间复杂度为 $O(n)$，但通过维护尾指针可优化至 $O(1)$。

删除操作需特别注意边界处理。删除头节点示例：

```
if (head->next == head) { // 单节点情况
    free(head);
    head = NULL;
} else {
    Node* prev_tail = head;
    while (prev_tail->next != head) { // 定位头节点的前驱（尾节点）
        prev_tail = prev_tail->next;
    }
    prev_tail->next = head->next; // 尾节点指向新头节点
    free(head); // 释放原头节点
    head = prev_tail->next; // 更新头指针
}
```

删除中间节点时，逻辑与单链表类似，但需额外维护闭环。

遍历循环链表使用 do-while 循环确保至少执行一次：

```
void print_list(Node* head) {
    if (!head) return; // 空链表直接返回
    Node* current = head;
    do {
        printf("%d␣", current->data); // 打印当前节点数据
        current = current->next; // 移至下一节点
    } while (current != head); // 终止条件：返回头节点
}
```

⚠ 关键陷阱：若误用 while (current != NULL) 会导致死循环，因为循环链表无 NULL 指针。
特殊边界处理包括单节点删除（直接释放内存并置空头指针）和约瑟夫环问题中的删除模式。后者涉及周期性删除节点，需精确控制遍历步长。

3 循环链表的优势与代价

循环链表的优势显著。头尾拼接操作在 $O(1)$ 时间内完成，优于单链表的 $O(n)$。例如，拼接两个循环链表只需修改尾节点指针。环形遍历无需边界判断，简化了迭代逻辑。在实现旋转缓冲区（如音频流）或轮询系统时，循环链表是天然选择。下表对比了关键操作的时间复杂度：

操作	单链表时间复杂度	循环链表时间复杂度
头插法	$O(1)$	$O(1)$
尾插法	$O(n)$	$O(n)$（可优化至 $O(1)$）
头尾拼接	$O(n)$	$O(1)$
遍历	$O(n)$	$O(n)$

然而，循环链表也存在缺陷。⚠ 内存泄漏风险较高：循环引用需手动释放所有节点，否则造成泄漏。⚠ 无限循环陷阱：遍历逻辑错误（如错误终止条件）易导致死循环。随机访问效率与单链表相同，均为 $O(n)$，不适合频繁随机查询的场景。

4 实战应用案例：约瑟夫问题求解

约瑟夫问题描述 N 人围圈报数，每数到第 K 人淘汰，求最后幸存者。循环链表提供优雅解法：

```
Node* josephus(int n, int k) {
    if (n < 1 || k < 1) return NULL; // 边界检查

    // 构建循环链表：创建 n 个节点并成环
    Node* head = create_node(1); // 头节点，数据为 1
    Node* prev = head; // 前驱指针
    for (int i = 2; i <= n; i++) {
        prev->next = create_node(i); // 添加新节点
        prev = prev->next; // 更新前驱
    }
    prev->next = head; // 尾节点指向头节点，闭环

    // 淘汰逻辑
    Node* current = head;
    while (current->next != current) { // 终止条件：只剩一个节点
        // 移动 k-1 步（跳过 k-1 个节点）
        for (int i = 1; i < k-1; i++) {
```

```
            current = current->next;
        }
        // 删除第 k 个节点
        Node* temp = current->next; // 临时保存待删除节点
        current->next = temp->next; // 跳过待删除节点
        free(temp); // 释放内存
        current = current->next; // 从下一节点继续
    }
    return current; // 返回幸存者节点
}
```

代码解读：首先生成包含 n 个节点的循环链表。淘汰阶段，每次移动 $k-1$ 步后删除第 k 个节点。循环终止时仅剩一个节点，即幸存者。时间复杂度为 $O(n \times k)$，空间复杂度 $O(n)$。

5 进阶讨论

双向循环链表扩展了单向版本，每个节点包含 prev 和 next 指针。插入操作需同时维护双向闭环：

```
new_node->next = current->next;
new_node->prev = current;
current->next->prev = new_node;
current->next = new_node;
```

⚠ 读者可尝试实现双向循环链表的删除操作，注意 prev 指针的更新。与数组实现的循环队列相比，循环链表在动态扩容上占优，但随机访问性能较差（数组为 $O(1)$，链表为 $O(n)$）。Linux 内核的 list.h 源码展示了工业级应用：通过宏定义实现高效通用的循环链表，支持进程调度和内存管理。

循环链表的适用场景可由决策树描述：若需高效头尾操作或连续遍历（如轮询系统），优先选择循环链表；若需随机访问，则考虑数组结构。关键学习收获是闭环思维在数据结构设计中的力量 —— 通过消除边界，提升操作效率。延伸学习建议包括跳表（优化查询效率）和循环双端队列（结合队列与链表优势）。掌握这些概念，可深化对环形数据流处理的理解。