

=====

文件夹: class007_LinkedListAlgorithms

=====

[Markdown 文件]

=====

文件: README.md

=====

链表合并算法专题 - Class010

📁 目录

- [算法概述] (#算法概述)
- [核心算法] (#核心算法)
- [扩展题目] (#扩展题目)
- [工程化考量] (#工程化考量)
- [复杂度分析] (#复杂度分析)
- [面试技巧] (#面试技巧)
- [实战训练] (#实战训练)
- [多语言实现] (#多语言实现)

⚙ 算法概述

本专题深入探讨链表合并相关算法，涵盖从基础合并到高级应用的完整知识体系。链表合并是数据结构与算法中的基础操作，广泛应用于排序、归并、数据处理等场景。

核心算法

- **双指针合并** - 两个有序链表合并的基础算法
- **优先队列合并** - K 个有序链表合并的高效算法
- **分治合并** - 大规模数据合并的优化策略
- **原地合并** - 空间优化的合并技术

🔑 核心算法

1. 合并两个有序链表 (LeetCode 21)

时间复杂度: $O(m+n)$

空间复杂度: $O(1)$ 迭代法, $O(m+n)$ 递归法

稳定性: 稳定

适用场景:

- 两个有序数据流的合并
- 归并排序的合并步骤
- 数据库查询结果的合并

****关键特性**:**

- 双指针技术的经典应用
- 递归实现简洁但空间开销大
- 迭代实现空间效率高

2. 合并 K 个有序链表 (LeetCode 23)

****时间复杂度**:** $O(N \log K)$

****空间复杂度**:** $O(K)$ 优先队列法, $O(\log K)$ 分治法

****稳定性**:** 稳定

****适用场景**:**

- 多路数据流合并
- 外部排序的归并阶段
- 分布式系统中的数据聚合

****优化策略**:**

- 优先队列法适合 K 较小的情况
- 分治法适合大规模数据合并
- 两两合并法实现简单但效率较低

3. 合并两个有序数组 (LeetCode 88)

****时间复杂度**:** $O(m+n)$

****空间复杂度**:** $O(1)$ 从后往前合并

****稳定性**:** 稳定

****适用场景**:**

- 数组合并操作
- 内存中的数据处理
- 缓存数据的合并

 扩展题目

基础题目

题目 1: LeetCode 21. 合并两个有序链表

****来源**:** LeetCode

****链接**:** <https://leetcode.cn/problems/merge-two-sorted-lists/>

****多种解法**:**

1. ****迭代法**** (最优解)
 - 时间复杂度: $O(m+n)$
 - 空间复杂度: $O(1)$
2. ****递归法****

- 时间复杂度: $O(m+n)$
- 空间复杂度: $O(m+n)$

题目 2: LeetCode 23. 合并 K 个升序链表

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-k-sorted-lists/>

多种解法:

1. **优先队列法** (最优解)
 - 时间复杂度: $O(N \log K)$
 - 空间复杂度: $O(K)$
2. **分治法**
 - 时间复杂度: $O(N \log K)$
 - 空间复杂度: $O(\log K)$
3. **顺序合并法**
 - 时间复杂度: $O(KN)$
 - 空间复杂度: $O(1)$

题目 3: LeetCode 88. 合并两个有序数组

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-sorted-array/>

多种解法:

1. **从后往前合并** (最优解)
 - 时间复杂度: $O(m+n)$
 - 空间复杂度: $O(1)$
2. **从前往后合并**
 - 时间复杂度: $O(m+n)$
 - 空间复杂度: $O(m)$
3. **合并后排序**
 - 时间复杂度: $O((m+n) \log(m+n))$
 - 空间复杂度: $O(1)$

进阶题目

题目 4: LeetCode 148. 排序链表

来源: LeetCode

链接: <https://leetcode.cn/problems/sort-list/>

解法: 归并排序在链表上的应用

- **自顶向下归并排序**
 - 时间复杂度: $O(n \log n)$
 - 空间复杂度: $O(\log n)$

- **自底向上归并排序** (最优解)

- 时间复杂度: $O(n \log n)$

- 空间复杂度: $O(1)$

题目 5: LeetCode 2. 两数相加

来源: LeetCode

链接: <https://leetcode.cn/problems/add-two-numbers/>

解法: 模拟加法过程

- 时间复杂度: $O(\max(m, n))$

- 空间复杂度: $O(\max(m, n))$

题目 6: LeetCode 24. 两两交换链表中的节点

来源: LeetCode

链接: <https://leetcode.cn/problems/swap-nodes-in-pairs/>

多种解法:

1. **迭代法** (最优解)

- 时间复杂度: $O(n)$

- 空间复杂度: $O(1)$

2. **递归法**

- 时间复杂度: $O(n)$

- 空间复杂度: $O(n)$

题目 7: LeetCode 86. 分隔链表

来源: LeetCode

链接: <https://leetcode.cn/problems/partition-list/>

解法: 双指针法

- 时间复杂度: $O(n)$

- 空间复杂度: $O(1)$

国内平台题目

题目 8: 牛客 NC33. 合并两个排序的链表

来源: 牛客网

链接: <https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337>

解法: 与 LeetCode 21 相同

题目 9: LintCode 104. 合并 k 个排序链表

来源: LintCode

链接: <https://www.lintcode.com/problem/104/>

****解法**:** 与 LeetCode 23 相同

题目 10: 剑指 Offer 25. 合并两个排序的链表

****来源**:** 剑指 Offer

****链接**:** <https://leetcode.cn/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/>

****解法**:** 与 LeetCode 21 相同

高级题目

题目 11: LeetCode 1634. 求两个多项式链表的和

****来源**:** LeetCode

****链接**:** <https://leetcode.cn/problems/add-two-polynomials-represented-as-linked-lists/>

****解法**:** 多项式合并，类似于有序链表合并

- 时间复杂度: $O(m+n)$
- 空间复杂度: $O(m+n)$

题目 12: LeetCode 109. 有序链表转换二叉搜索树

****来源**:** LeetCode

****链接**:** <https://leetcode.cn/problems/convert-sorted-list-to-binary-search-tree/>

****解法**:** 分治+快慢指针找中点

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(\log n)$

题目 13: LeetCode 147. 对链表进行插入排序

****来源**:** LeetCode

****链接**:** <https://leetcode.cn/problems/insertion-sort-list/>

****解法**:** 插入排序在链表上的实现

- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(1)$

 工程化考量

1. 异常处理与边界条件

```
```java
// 输入验证
if (list1 == null) return list2;
if (list2 == null) return list1;
// 处理空链表、单节点链表等边界情况
```

```

2. 内存管理

- **C++**: 注意内存释放，避免内存泄漏
- **Java/Python**: 依赖垃圾回收，但要注意循环引用

3. 性能优化策略

- 使用哨兵节点简化边界处理
- 避免不必要的对象创建
- 选择合适的数据结构（优先队列 vs 分治）

4. 线程安全考虑

- 在多线程环境下需要同步保护
- 使用不可变数据结构避免竞态条件

5. 测试覆盖

```
```java
// 单元测试用例设计
@Test
public void testMergeTwoLists() {
 // 空链表测试
 // 单节点链表测试
 // 正常情况测试
 // 极端值测试
 // 性能测试
}
```

```

📈 复杂度分析

时间复杂度对比

| 算法 | 最好情况 | 平均情况 | 最坏情况 | 空间复杂度 |
|----------------|---------------|---------------|---------------|-------------|
| 合并两个链表(迭代) | $O(m+n)$ | $O(m+n)$ | $O(m+n)$ | $O(1)$ |
| 合并两个链表(递归) | $O(m+n)$ | $O(m+n)$ | $O(m+n)$ | $O(m+n)$ |
| 合并 K 个链表(优先队列) | $O(N \log K)$ | $O(N \log K)$ | $O(N \log K)$ | $O(K)$ |
| 合并 K 个链表(分治) | $O(N \log K)$ | $O(N \log K)$ | $O(N \log K)$ | $O(\log K)$ |
| 排序链表(自底向上) | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ |

空间复杂度分析

- **迭代法**: 只使用常数级额外空间
- **递归法**: 递归调用栈的空间开销
- **优先队列法**: 队列大小与 K 成正比

- **分治法**: 递归深度与 $\log K$ 成正比

💡 面试技巧

1. 算法选择依据

- **数据规模**: 小规模数据用简单算法，大规模用高效算法
- **内存限制**: 内存紧张时选择空间效率高的算法
- **稳定性要求**: 需要稳定排序时选择稳定算法
- **数据特性**: 根据数据分布特点选择合适算法

2. 代码实现要点

- 清晰的变量命名（如 dummy, current, prev 等）
- 关键步骤添加注释
- 边界条件处理完善
- 异常情况考虑周全

3. 性能分析能力

- 能够准确分析时间/空间复杂度
- 理解常数项对实际性能的影响
- 知道如何优化算法常数项

4. 调试技巧

```
``` java
// 调试打印中间状态
System.out.println("当前指针位置: " + current.val);
System.out.println("剩余链表长度: " + getLength(remaining));
```
```

🎯 实战训练

推荐练习题目

LeetCode 必做题目

1. **21. 合并两个有序链表** - 基础必会
2. **23. 合并 K 个升序链表** - 进阶重要
3. **88. 合并两个有序数组** - 数组版本
4. **148. 排序链表** - 归并排序应用
5. **2. 两数相加** - 链表操作变种

牛客网题目

1. **NC33. 合并两个排序的链表** - 面试高频
2. **NC51. 合并 k 个已排序的链表** - 进阶练习

剑指 Offer 题目

1. **面试题 25. 合并两个排序的链表** - 经典题目
2. **面试题 17. 打印从 1 到最大的 n 位数** - 类似思想

进阶挑战题目

1. **LeetCode 109. 有序链表转换二叉搜索树** - 分治应用
2. **LeetCode 147. 对链表进行插入排序** - 排序算法
3. **LeetCode 86. 分隔链表** - 双指针应用

🌐 多语言实现

本专题提供 Java、C++、Python 三种语言的完整实现：

Java 版本特点

- 面向对象设计，类结构清晰
- 完善的异常处理机制
- 内存管理依赖 GC
- 适合大型工程项目

C++ 版本特点

- 手动内存管理，性能优化空间大
- 指针操作直接，效率高
- 适合系统级编程
- 需要特别注意内存安全

Python 版本特点

- 代码简洁，开发效率高
- 动态类型，灵活性好
- 内置数据结构丰富
- 适合快速原型开发

🔍 调试与优化

1. 常见错误排查

- **空指针异常**：检查边界条件
- **内存泄漏**：确保资源正确释放
- **性能问题**：分析时间复杂度是否最优

2. 性能优化技巧

- 使用迭代代替递归减少栈空间
- 选择合适的数据结构（优先队列 vs 分治）
- 避免不必要的对象创建

```
### 3. 测试策略
```java
// 全面的测试用例设计
public class MergeTwoListsTest {
 @Test
 public void testEmptyLists() { /* 测试空链表 */ }
 @Test
 public void testSingleNodeLists() { /* 测试单节点链表 */ }
 @Test
 public void testNormalCase() { /* 测试正常情况 */ }
 @Test
 public void testExtremeValues() { /* 测试极端值 */ }
 @Test
 public void testPerformance() { /* 性能测试 */ }
}
```

```

📚 学习资源

推荐书籍

1. 《算法导论》 - 算法理论基础
2. 《编程珠玑》 - 算法实践应用
3. 《剑指 Offer》 - 面试算法准备

在线资源

1. **LeetCode** - 算法题目练习平台
2. **牛客网** - 国内算法题库
3. **GeeksforGeeks** - 算法详解

可视化工具

1. **VisualGo** - 算法可视化演示
2. **USFCA Algorithm Animations** - 动画演示

持续更新中... 更多题目和解析将陆续添加

最后更新: 2025 年 10 月 18 日

=====

[代码文件]

=====

文件: MergeTwoLists.cpp

```
=====
```

/*

* 合并两个有序链表及相关题目扩展 (C++版本)

*

*  算法专题: 链表合并与相关算法

*  覆盖平台: LeetCode、牛客网、LintCode、剑指 Offer 等

*  语言特性: C++11/14/17 标准, RAIU 资源管理, STL 容器使用

*

*  工程化考量:

* 1. 内存管理: 手动内存分配与释放, 避免内存泄漏

* 2. 异常安全: 确保代码在异常情况下的资源释放

* 3. 性能优化: STL 容器选择, 算法常数项优化

* 4. 可测试性: 完整的单元测试框架, 边界条件覆盖

* 5. 可维护性: 清晰的代码结构, 详细的注释说明

*

*  复杂度分析体系:

* - 时间复杂度: 从理论分析到实际性能考量

* - 空间复杂度: 内存使用优化策略

* - 常数项分析: 实际运行效率的关键因素

*

*  算法应用场景:

* - 大数据处理: 外部排序, 多路归并

* - 实时系统: 数据流合并处理

* - 分布式计算: 多节点结果合并

* - 数据库系统: 索引合并优化

*

* 主要题目:

* 1. LeetCode 21. 合并两个有序链表 (基础题)

* 2. LeetCode 23. 合并 K 个升序链表 (进阶题)

* 3. LeetCode 88. 合并两个有序数组 (变种题)

* 4. LeetCode 148. 排序链表 (应用扩展)

* 5. LeetCode 2. 两数相加 (链表操作)

* 6. LeetCode 24. 两两交换链表中的节点 (链表变换)

* 7. 牛客 NC33. 合并两个排序的链表 (国内平台)

* 8. LintCode 104. 合并 k 个排序链表 (国际平台)

* 9. LeetCode 86. 分隔链表 (链表分割)

*

*  解题思路技巧总结:

* 1. 双指针法: 适用于两个有序序列的合并, 时间复杂度 $O(m+n)$

* 2. 优先队列(堆): 适用于 K 个有序序列的合并, 时间复杂度 $O(N \log K)$

* 3. 分治法: 将 K 个序列问题分解为多个两个序列问题, 时间复杂度 $O(N \log K)$

* 4. 哨兵节点: 简化链表操作的边界处理, 提高代码可读性

* 5. 原地修改：充分利用已有空间，减少额外空间使用

* 6. 递归与迭代：不同场景下的选择策略

*

* ⚡ 时间复杂度分析：

* 1. 合并两个链表： $O(m+n)$ ， m 和 n 分别是两个链表的长度

* 2. 合并 K 个链表（优先队列）： $O(N \log K)$ ， N 是所有节点总数， K 是链表数量

* 3. 合并 K 个链表（分治）： $O(N \log K)$

* 4. 合并两个数组： $O(m+n)$

* 5. 链表排序： $O(n \log n)$ ，归并排序最优

*

* 💾 空间复杂度分析：

* 1. 合并两个链表： $O(1)$ ，原地操作

* 2. 合并 K 个链表（优先队列）： $O(K)$ ，堆的大小

* 3. 合并 K 个链表（分治）： $O(\log K)$ ，递归栈深度

* 4. 合并两个数组： $O(1)$ ，原地操作

* 5. 链表排序： $O(1)$ 或 $O(\log n)$ ，取决于实现方式

*

* 🛡️ 安全与稳定性：

* - 空指针检查：所有链表操作前的边界检查

* - 内存泄漏防护：RAII 模式，智能指针使用

* - 异常处理：try-catch 块，资源清理

* - 输入验证：参数合法性检查

*

* 🛠️ 调试与测试：

* - 单元测试：每个算法的独立测试用例

* - 边界测试：空输入、单元素、极端值等

* - 性能测试：大规模数据下的性能表现

* - 内存测试：内存泄漏检测工具使用

*

* 📚 学习路径建议：

* 1. 基础掌握：LeetCode 21 → 牛客 NC33

* 2. 进阶提升：LeetCode 23 → LintCode 104

* 3. 综合应用：LeetCode 148 → LeetCode 2

* 4. 拓展思维：LeetCode 24 → LeetCode 86

*

* 🎓 面试重点：

* - 算法思路清晰表达

* - 时间空间复杂度分析

* - 边界条件处理能力

* - 代码实现简洁优雅

* - 工程化考量意识

*/

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
#include <cassert>
#include <memory>
#include <chrono>
#include <random>

using namespace std;

// 链表节点定义
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}

    // 用于测试的链表创建方法
    static ListNode* createList(std::vector<int>& arr) {
        if (arr.empty()) return nullptr;
        ListNode* head = new ListNode(arr[0]);
        ListNode* cur = head;
        for (size_t i = 1; i < arr.size(); i++) {
            cur->next = new ListNode(arr[i]);
            cur = cur->next;
        }
        return head;
    }

    // 用于测试的链表打印方法
    static void printList(ListNode* head) {
        ListNode* cur = head;
        while (cur) {
            std::cout << cur->val;
            if (cur->next) std::cout << " -> ";
            cur = cur->next;
        }
        std::cout << std::endl;
    }
}
```

```

// 释放链表内存
static void deleteList(ListNode* head) {
    while (head) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }
}

/**
 * 题目 1: LeetCode 21. 合并两个有序链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/merge-two-sorted-lists/
 *
 * 题目描述:
 * 将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。
 *
 * 解法分析:
 * 1. 迭代法 - 时间复杂度: O(m+n), 空间复杂度: O(1)
 * 2. 递归法 - 时间复杂度: O(m+n), 空间复杂度: O(m+n)
 *
 * 解题思路:
 * 使用双指针分别指向两个链表的当前节点，比较节点值的大小，
 * 将较小的节点连接到结果链表中，移动对应指针，重复此过程直到某一链表遍历完。
 * 最后将未遍历完的链表剩余部分直接连接到结果链表末尾。
 */
class MergeTwoSortedListsSolution {
public:
    /**
     * 解法 1: 迭代法 (推荐)
     * 时间复杂度: O(m+n) - m 和 n 分别是两个链表的长度
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用哨兵节点简化边界处理
     * 2. 双指针分别遍历两个链表
     * 3. 比较节点值，将较小节点连接到结果链表
     * 4. 处理剩余节点
     */
    static ListNode* mergeTwoListsIterative(ListNode* list1, ListNode* list2) {
        // 创建哨兵节点，简化边界处理
        ListNode dummy(0);

```

```

ListNode* current = &dummy;

// 双指针遍历两个链表
while (list1 && list2) {
    // 比较两个链表当前节点的值
    if (list1->val <= list2->val) {
        current->next = list1;
        list1 = list1->next;
    } else {
        current->next = list2;
        list2 = list2->next;
    }
    current = current->next;
}

// 连接剩余节点
current->next = list1 ? list1 : list2;

// 返回合并后的链表
return dummy.next;
}

/***
 * 解法 2：递归法
 * 时间复杂度：O(m+n) - 每个节点访问一次
 * 空间复杂度：O(m+n) - 递归调用栈的深度
 *
 * 核心思想：
 * 1. 递归终止条件：其中一个链表为空
 * 2. 递归处理：选择较小节点作为当前节点，递归处理剩余部分
 * 3. 返回当前节点
 */
static ListNode* mergeTwoListsRecursive(ListNode* list1, ListNode* list2) {
    // 递归终止条件
    if (!list1) return list2;
    if (!list2) return list1;

    // 递归处理
    if (list1->val <= list2->val) {
        list1->next = mergeTwoListsRecursive(list1->next, list2);
        return list1;
    } else {
        list2->next = mergeTwoListsRecursive(list1, list2->next);
    }
}

```

```
        return list2;
    }
}

/***
 * 测试方法
 */
static void test() {
    std::cout << "==> 合并两个有序链表测试 ==>" << std::endl;

    // 测试用例 1: 正常情况
    std::vector<int> arr1 = {1, 2, 4};
    std::vector<int> arr2 = {1, 3, 4};
    ListNode* list1 = ListNode::createList(arr1);
    ListNode* list2 = ListNode::createList(arr2);
    std::cout << "链表 1: ";
    ListNode::printList(list1);
    std::cout << "链表 2: ";
    ListNode::printList(list2);

    ListNode* result1 = mergeTwoListsIterative(list1, list2);
    std::cout << "迭代法结果: ";
    ListNode::printList(result1);
    ListNode::deleteList(result1);

    // 重新创建测试数据
    list1 = ListNode::createList(arr1);
    list2 = ListNode::createList(arr2);
    ListNode* result2 = mergeTwoListsRecursive(list1, list2);
    std::cout << "递归法结果: ";
    ListNode::printList(result2);
    ListNode::deleteList(result2);

    // 测试用例 2: 空链表
    ListNode* list3 = nullptr;
    std::vector<int> arr4 = {0};
    ListNode* list4 = ListNode::createList(arr4);
    ListNode* result3 = mergeTwoListsIterative(list3, list4);
    std::cout << "空链表测试: ";
    ListNode::printList(result3);
    ListNode::deleteList(result3);

    // 测试用例 3: 两个空链表
```

```

ListNode* list5 = nullptr;
ListNode* list6 = nullptr;
ListNode* result4 = mergeTwoListsIterative(list5, list6);
std::cout << "两个空链表: ";
ListNode::printList(result4);
ListNode::deleteList(result4);
std::cout << std::endl;
}

};

/***
* 题目 2: LeetCode 23. 合并 K 个升序链表
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/merge-k-sorted-lists/
*
* 题目描述:
* 给你一个链表数组，每个链表都已经按升序排列。
* 请你将所有链表合并到一个升序链表中，返回合并后的链表。
*
* 解法分析:
* 1. 优先队列法（最优解） - 时间复杂度: O(N*logK)，空间复杂度: O(K)
* 2. 分治法 - 时间复杂度: O(N*logK)，空间复杂度: O(logK)
*
* 解题思路:
* 优先队列法: 维护一个大小为 K 的最小堆，堆中存放 K 个链表的头节点。
* 每次从堆中取出最小节点加入结果链表，并将该节点的下一个节点加入堆中。
* 分治法: 将 K 个链表分成两部分，分别合并后再合并两个结果。
*/
class MergeKSortedListsSolution {
public:
    // 自定义比较函数，用于优先队列
    struct Compare {
        bool operator()(ListNode* a, ListNode* b) {
            return a->val > b->val; // 最小堆
        }
    };
    /**
     * 解法 1: 优先队列法（推荐）
     * 时间复杂度: O(N*logK) - N 是所有节点总数，K 是链表数量
     * 空间复杂度: O(K) - 优先队列的大小
     *
     * 核心思想:
    
```

```

* 1. 使用优先队列(最小堆)维护 K 个链表的当前最小节点
* 2. 每次取出最小节点加入结果链表
* 3. 将取出节点的下一个节点加入优先队列
* 4. 重复直到优先队列为空
*/
static ListNode* mergeKListsPriorityQueue(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;

    // 创建优先队列(最小堆)
    priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;

    // 将所有非空链表的头节点加入优先队列
    for (ListNode* list : lists) {
        if (list) {
            minHeap.push(list);
        }
    }

    // 创建哨兵节点
    ListNode dummy(0);
    ListNode* current = &dummy;

    // 从优先队列中依次取出最小节点
    while (!minHeap.empty()) {
        // 取出最小节点
        ListNode* node = minHeap.top();
        minHeap.pop();

        // 加入结果链表
        current->next = node;
        current = current->next;

        // 如果该节点还有后续节点，加入优先队列
        if (node->next) {
            minHeap.push(node->next);
        }
    }

    return dummy.next;
}

/**
 * 解法 2：分治法

```

```

* 时间复杂度: O(N*logK) - N 是所有节点总数, K 是链表数量
* 空间复杂度: O(logK) - 递归调用栈的深度
*
* 核心思想:
* 1. 将 K 个链表分成两部分
* 2. 递归合并每一部分
* 3. 合并两个结果链表
*/
static ListNode* mergeKListsDivideAndConquer(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;
    return mergeKListsHelper(lists, 0, lists.size() - 1);
}

private:
    /**
     * 分治辅助函数
    */
    static ListNode* mergeKListsHelper(vector<ListNode*>& lists, int left, int right) {
        if (left == right) return lists[left];
        if (left + 1 == right) return
MergeTwoSortedListsSolution::mergeTwoListsIterative(lists[left], lists[right]);

        int mid = left + (right - left) / 2;
        ListNode* l1 = mergeKListsHelper(lists, left, mid);
        ListNode* l2 = mergeKListsHelper(lists, mid + 1, right);

        return MergeTwoSortedListsSolution::mergeTwoListsIterative(l1, l2);
    }
}

public:
    /**
     * 测试方法
    */
    static void test() {
        std::cout << "==== 合并 K 个升序链表测试 ===" << std::endl;

        // 创建测试数据
        std::vector<int> arr1 = {1, 4, 5};
        std::vector<int> arr2 = {1, 3, 4};
        std::vector<int> arr3 = {2, 6};

        ListNode* l1 = ListNode::createList(arr1);
        ListNode* l2 = ListNode::createList(arr2);
        ListNode* l3 = ListNode::createList(arr3);
    }
}

```

```

std::vector<ListNode*> lists = {11, 12, 13};

    std::cout << "链表 1: ";
    ListNode::printList(lists[0]);
    std::cout << "链表 2: ";
    ListNode::printList(lists[1]);
    std::cout << "链表 3: ";
    ListNode::printList(lists[2]);

    // 测试优先队列法
    std::vector<ListNode*> listsCopy1 = lists;
    ListNode* result1 = mergeKListsPriorityQueue(listsCopy1);
    std::cout << "优先队列法结果: ";
    ListNode::printList(result1);
    ListNode::deleteList(result1);

    // 测试分治法
    std::vector<ListNode*> listsCopy2 = lists;
    ListNode* result2 = mergeKListsDivideAndConquer(listsCopy2);
    std::cout << "分治法结果: ";
    ListNode::printList(result2);
    ListNode::deleteList(result2);
    std::cout << std::endl;
}

};

/***
 * 题目 3: LeetCode 88. 合并两个有序数组
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/merge-sorted-array/
 *
 * 题目描述:
 * 给你两个按非递减顺序排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，
 * 分别表示 nums1 和 nums2 中的元素数目。
 * 请你合并 nums2 到 nums1 中，使合并后的数组同样按非递减顺序排列。
 * 注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。
 * 为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。
 * nums2 的长度为 n。
 *
 * 解法分析:
 * 1. 从后往前合并（最优解） - 时间复杂度: O(m+n)，空间复杂度: O(1)
 */

```

- * 2. 从前往后合并 - 时间复杂度: $O(m+n)$, 空间复杂度: $O(m+n)$
- * 3. 合并后排序 - 时间复杂度: $O((m+n) \log(m+n))$, 空间复杂度: $O(1)$
- *
- * 解题思路:
- * 从后往前合并可以避免覆盖 `nums1` 中未处理的元素。
- * 使用三个指针分别指向 `nums1` 有效元素末尾、`nums2` 末尾和 `nums1` 实际末尾。
- * 比较两个数组当前元素，将较大者放入 `nums1` 末尾，移动相应指针。
- */

```

class MergeSortedArraySolution {
public:
    /**
     * 解法 1: 从后往前合并 (推荐)
     * 时间复杂度:  $O(m+n)$  - 每个元素访问一次
     * 空间复杂度:  $O(1)$  - 原地修改
     *
     * 核心思想:
     * 1. 从两个数组的末尾开始比较
     * 2. 将较大元素放到 nums1 的末尾
     * 3. 移动相应指针
     * 4. 处理剩余元素
     */
    static void mergeFromBack(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        // 三个指针
        int i = m - 1;          // nums1 有效元素的末尾
        int j = n - 1;          // nums2 的末尾
        int k = m + n - 1;      // nums1 实际末尾

        // 从后往前合并
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }

        // 处理 nums2 剩余元素
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }

        // 注意: 如果 nums1 有剩余元素, 它们已经在正确位置, 无需处理
    }
}

```

```

/***
 * 解法 2：从前往后合并
 * 时间复杂度：O(m+n)
 * 空间复杂度：O(m) - 需要额外数组存储 nums1 的前 m 个元素
 */
static void mergeFromFront(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    // 创建临时数组存储 nums1 的前 m 个元素
    vector<int> nums1Copy(nums1.begin(), nums1.begin() + m);

    // 三个指针
    size_t i = 0; // nums1Copy 的指针
    size_t j = 0; // nums2 的指针
    size_t k = 0; // nums1 的指针

    // 从前往后合并
    while (i < (size_t)m && j < (size_t)n) {
        if (nums1Copy[i] <= nums2[j]) {
            nums1[k++] = nums1Copy[i++];
        } else {
            nums1[k++] = nums2[j++];
        }
    }

    // 处理剩余元素
    while (i < (size_t)m) {
        nums1[k++] = nums1Copy[i++];
    }

    while (j < (size_t)n) {
        nums1[k++] = nums2[j++];
    }
}

/***
 * 解法 3：合并后排序
 * 时间复杂度：O((m+n) log(m+n))
 * 空间复杂度：O(1)
 */
static void mergeAndSort(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    // 将 nums2 复制到 nums1 的后半部分
    for (int i = 0; i < n; i++) {
        nums1[m + i] = nums2[i];
    }
}

```

```
}

// 排序
sort(nums1.begin(), nums1.end());
}

/***
 * 测试方法
*/
static void test() {
    cout << "==== 合并两个有序数组测试 ===" << endl;

    // 测试用例 1
    vector<int> nums1 = {1, 2, 3, 0, 0, 0};
    int m = 3;
    vector<int> nums2 = {2, 5, 6};
    int n = 3;

    cout << "数组 1: [";
    for (size_t i = 0; i < nums1.size(); i++) {
        cout << nums1[i];
        if (i < nums1.size() - 1) cout << ", ";
    }
    cout << "], m = " << m << endl;

    cout << "数组 2: [";
    for (size_t i = 0; i < nums2.size(); i++) {
        cout << nums2[i];
        if (i < nums2.size() - 1) cout << ", ";
    }
    cout << "], n = " << n << endl;

    // 测试从后往前合并
    vector<int> nums1Copy1 = nums1;
    mergeFromBack(nums1Copy1, m, nums2, n);
    cout << "从后往前合并: [";
    for (size_t i = 0; i < nums1Copy1.size(); i++) {
        cout << nums1Copy1[i];
        if (i < nums1Copy1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    // 测试从前往后合并
    vector<int> nums1Copy2 = nums1;
```

```

mergeFromFront(nums1Copy2, m, nums2, n);
cout << "从前往后合并: [";
for (size_t i = 0; i < nums1Copy2.size(); i++) {
    cout << nums1Copy2[i];
    if (i < nums1Copy2.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试合并后排序
vector<int> nums1Copy3 = nums1;
mergeAndSort(nums1Copy3, m, nums2, n);
cout << "合并后排序: [";
for (size_t i = 0; i < nums1Copy3.size(); i++) {
    cout << nums1Copy3[i];
    if (i < nums1Copy3.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 2: nums1 为空
vector<int> nums3 = {0};
int m2 = 0;
vector<int> nums4 = {1};
int n2 = 1;

cout << "\n数组 1: [";
for (size_t i = 0; i < nums3.size(); i++) {
    cout << nums3[i];
    if (i < nums3.size() - 1) cout << ", ";
}
cout << "], m = " << m2 << endl;

cout << "数组 2: [";
for (size_t i = 0; i < nums4.size(); i++) {
    cout << nums4[i];
    if (i < nums4.size() - 1) cout << ", ";
}
cout << "], n = " << n2 << endl;

mergeFromBack(nums3, m2, nums4, n2);
cout << "从后往前合并: [";
for (size_t i = 0; i < nums3.size(); i++) {
    cout << nums3[i];
    if (i < nums3.size() - 1) cout << ", ";
}

```

```

    }

    cout << "]" << endl;
    cout << endl;
}

};

/***
 * 题目 4: LeetCode 148. 排序链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/sort-list/
 *
 * 题目描述:
 * 给你链表的头结点 head，请将其按 升序 排列并返回 排序后的链表。
 * 要求在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。
 */
class SortListSolution {
public:
    /**
     * 解法 1: 归并排序（自顶向下）
     * 时间复杂度: O(nlogn) - 归并排序的标准时间复杂度
     * 空间复杂度: O(logn) - 递归调用栈的深度
     */
    static ListNode* sortListTopDown(ListNode* head) {
        // 基本情况: 空链表或只有一个节点
        if (head == nullptr || head->next == nullptr) {
            return head;
        }

        // 使用快慢指针找到中点
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // 分割链表
        ListNode* mid = slow->next;
        slow->next = nullptr;

        // 递归排序两个子链表
        ListNode* left = sortListTopDown(head);
        ListNode* right = sortListTopDown(mid);
    }
}

```

```

// 合并排序后的链表
return mergeTwoLists(left, right);
}

/***
 * 解法 2：归并排序（自底向上） - 最优解
 * 时间复杂度：O(nlogn) - 与自顶向下相同
 * 空间复杂度：O(1) - 只使用常数级额外空间
 */
static ListNode* sortListBottomUp(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // 计算链表长度
    int length = 0;
    ListNode* current = head;
    while (current != nullptr) {
        length++;
        current = current->next;
    }

    // 创建哨兵节点
    ListNode dummy(-1);
    dummy.next = head;

    // 自底向上进行归并
    for (int step = 1; step < length; step *= 2) {
        ListNode* prev = &dummy;
        current = dummy.next;

        while (current != nullptr) {
            // 第一个子链表的头节点
            ListNode* left = current;
            // 分割第一个子链表
            for (int i = 1; i < step && current->next != nullptr; i++) {
                current = current->next;
            }

            // 第二个子链表的头节点
            ListNode* right = current->next;
            // 断开第一个子链表

```

```

current->next = nullptr;
current = right;

// 分割第二个子链表
for (int i = 1; i < step && current != nullptr && current->next != nullptr; i++)
{
    current = current->next;
}

// 记录下一段链表的起始位置
ListNode* next = nullptr;
if (current != nullptr) {
    next = current->next;
    current->next = nullptr;
}

// 合并两个子链表
prev->next = mergeTwoLists(left, right);

// 移动 prev 到合并后链表的末尾
while (prev->next != nullptr) {
    prev = prev->next;
}

// 处理下一段链表
current = next;
}

return dummy.next;
}

// 合并两个有序链表的辅助函数
static ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode dummy(-1);
    ListNode* current = &dummy;

    while (l1 != nullptr && l2 != nullptr) {
        if (l1->val <= l2->val) {
            current->next = l1;
            l1 = l1->next;
        } else {
            current->next = l2;
        }
        current = current->next;
    }

    if (l1 == nullptr) {
        current->next = l2;
    } else {
        current->next = l1;
    }
}

```

```
    12 = 12->next;
}
current = current->next;
}

current->next = (11 != nullptr) ? 11 : 12;
return dummy.next;
}

/***
 * 测试方法
 */
static void test() {
    cout << "==== 排序链表测试 ===" << endl;

    // 测试用例 1: 正常情况
    vector<int> arr1 = {4, 2, 1, 3};
    ListNode* list1 = ListNode::createList(arr1);
    cout << "原链表: ";
    ListNode::printList(list1);

    ListNode* result1 = sortListTopDown(list1);
    cout << "自顶向下归并排序结果: ";
    ListNode::printList(result1);

    // 释放内存
    ListNode::deleteList(result1);

    // 重新创建测试数据
    vector<int> arr2 = {4, 2, 1, 3};
    ListNode* list2 = ListNode::createList(arr2);
    ListNode* result2 = sortListBottomUp(list2);
    cout << "自底向上归并排序结果: ";
    ListNode::printList(result2);

    // 测试用例 2: 包含重复元素
    vector<int> arr3 = {-1, 5, 3, 4, 0};
    ListNode* list3 = ListNode::createList(arr3);
    cout << "\n 原链表: ";
    ListNode::printList(list3);

    ListNode* result3 = sortListBottomUp(list3);
    cout << "排序结果: ";
```

```

ListNode::printList(result3);
cout << endl;

// 释放内存
ListNode::deleteList(result2);
ListNode::deleteList(result3);
}

};

/***
* 题目 5: LeetCode 2. 两数相加
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/add-two-numbers/
*
* 题目描述:
* 给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能
存储一位数字。
* 请你将两个数相加，并以相同形式返回一个表示和的链表。
*/
class AddTwoNumbersSolution {
public:
    /**
     * 解法: 模拟加法过程
     * 时间复杂度: O(max(m, n)) - m 和 n 分别是两个链表的长度
     * 空间复杂度: O(max(m, n)) - 输出链表的长度最多为 max(m, n)+1
     */
    static ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 创建哨兵节点
        ListNode dummy(-1);
        ListNode* current = &dummy;

        // 进位
        int carry = 0;

        // 同时遍历两个链表
        while (l1 != nullptr || l2 != nullptr || carry > 0) {
            // 计算当前位的和
            int sum = carry;
            if (l1 != nullptr) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2 != nullptr) {

```

```

        sum += l2->val;
        l2 = l2->next;
    }

    // 更新进位
    carry = sum / 10;
    // 创建新节点存储当前位的结果
    current->next = new ListNode(sum % 10);
    current = current->next;
}

return dummy.next;
}

/**
 * 测试方法
 */
static void test() {
    cout << "==== 两数相加测试 ===" << endl;

    // 测试用例 1: 正常情况
    vector<int> arr1 = {2, 4, 3}; // 342
    vector<int> arr2 = {5, 6, 4}; // 465
    ListNode* l1 = ListNode::createList(arr1);
    ListNode* l2 = ListNode::createList(arr2);
    cout << "链表 1 (342 逆序): ";
    ListNode::printList(l1);
    cout << "链表 2 (465 逆序): ";
    ListNode::printList(l2);

    ListNode* result1 = addTwoNumbers(l1, l2);
    cout << "结果 (807 逆序): ";
    ListNode::printList(result1);

    // 测试用例 2: 包含进位
    vector<int> arr3 = {9, 9, 9, 9, 9, 9, 9};
    vector<int> arr4 = {9, 9, 9, 9};
    ListNode* l3 = ListNode::createList(arr3);
    ListNode* l4 = ListNode::createList(arr4);
    cout << "\n链表 1: ";
    ListNode::printList(l3);
    cout << "链表 2: ";
    ListNode::printList(l4);
}

```

```

ListNode* result2 = addTwoNumbers(13, 14);
cout << "结果: ";
ListNode::printList(result2);
cout << endl;

// 释放内存
ListNode::deleteList(11);
ListNode::deleteList(12);
ListNode::deleteList(result1);
ListNode::deleteList(13);
ListNode::deleteList(14);
ListNode::deleteList(result2);

}

};

/***
 * 题目 6: LeetCode 24. 两两交换链表中的节点
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/swap-nodes-in-pairs/
 *
 * 题目描述:
 * 给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。
 * 你必须在不修改节点内部值的情况下完成本题（即，只能进行节点交换）。
 */
class SwapNodesInPairsSolution {
public:
    /**
     * 解法 1: 迭代法 (推荐)
     * 时间复杂度: O(n) - 每个节点只访问一次
     * 空间复杂度: O(1) - 只使用常数级额外空间
     */
    static ListNode* swapPairsIterative(ListNode* head) {
        // 创建哨兵节点
        ListNode dummy(-1);
        dummy.next = head;

        ListNode* prev = &dummy;

        // 确保有至少两个节点可以交换
        while (prev->next != nullptr && prev->next->next != nullptr) {
            // 标记需要交换的两个节点
            ListNode* first = prev->next;

```

```

    ListNode* second = prev->next->next;

    // 交换节点
    first->next = second->next;
    second->next = first;
    prev->next = second;

    // 移动 prev 到下一对的前一个位置
    prev = first;
}

return dummy.next;
}

/***
 * 解法 2：递归法
 * 时间复杂度：O(n) - 每个节点只访问一次
 * 空间复杂度：O(n) - 递归调用栈的深度
 */
static ListNode* swapPairsRecursive(ListNode* head) {
    // 递归终止条件
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // 标记需要交换的两个节点
    ListNode* first = head;
    ListNode* second = head->next;

    // 交换节点
    first->next = swapPairsRecursive(second->next);
    second->next = first;

    // 返回新的头节点
    return second;
}

/***
 * 测试方法
 */
static void test() {
    cout << "===" 两两交换链表中的节点测试 ===" << endl;
}

```

```
// 测试用例 1: 偶数个节点
vector<int> arr1 = {1, 2, 3, 4};
ListNode* list1 = ListNode::createList(arr1);
cout << "原链表: ";
ListNode::printList(list1);

ListNode* result1 = swapPairsIterative(list1);
cout << "迭代法结果: ";
ListNode::printList(result1);

// 释放内存
ListNode::deleteList(result1);

// 重新创建测试数据
vector<int> arr2 = {1, 2, 3, 4};
ListNode* list2 = ListNode::createList(arr2);
ListNode* result2 = swapPairsRecursive(list2);
cout << "递归法结果: ";
ListNode::printList(result2);

// 测试用例 2: 奇数个节点
vector<int> arr3 = {1, 2, 3};
ListNode* list3 = ListNode::createList(arr3);
cout << "\n原链表: ";
ListNode::printList(list3);

ListNode* result3 = swapPairsIterative(list3);
cout << "交换结果: ";
ListNode::printList(result3);
cout << endl;

// 释放内存
ListNode::deleteList(result2);
ListNode::deleteList(result3);
}

};

/***
 * 题目 7: 牛客 NC33. 合并两个排序的链表
 * 来源: 牛客网
 * 链接: https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337
 */
class NowCoderMergeSortedListsSolution {
```

```
public:

    static ListNode* merge(ListNode* pHead1, ListNode* pHead2) {
        ListNode dummy(-1);
        ListNode* current = &dummy;

        while (pHead1 != nullptr && pHead2 != nullptr) {
            if (pHead1->val <= pHead2->val) {
                current->next = pHead1;
                pHead1 = pHead1->next;
            } else {
                current->next = pHead2;
                pHead2 = pHead2->next;
            }
            current = current->next;
        }

        current->next = (pHead1 != nullptr) ? pHead1 : pHead2;
        return dummy.next;
    }

    /**
     * 测试方法
     */
    static void test() {
        cout << "==== 牛客 NC33. 合并两个排序的链表测试 ===" << endl;

        vector<int> arr1 = {1, 3, 5};
        vector<int> arr2 = {2, 4, 6};
        ListNode* list1 = ListNode::createList(arr1);
        ListNode* list2 = ListNode::createList(arr2);
        cout << "链表 1: ";
        ListNode::printList(list1);
        cout << "链表 2: ";
        ListNode::printList(list2);

        ListNode* result = merge(list1, list2);
        cout << "合并结果: ";
        ListNode::printList(result);
        cout << endl;

        // 释放内存
        ListNode::deleteList(result);
    }
}
```

```

};

/**
 * 题目 8: LintCode 104. 合并 k 个排序链表
 * 来源: LintCode
 * 链接: https://www.lintcode.com/problem/104/
 */

class LintCodeMergeKListsSolution {
public:
    // 自定义比较器, 用于优先队列
    struct CompareNode {
        bool operator() (ListNode* a, ListNode* b) {
            return a->val > b->val; // 小顶堆
        }
    };
};

static ListNode* mergeKLists(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;

    // 创建优先队列(最小堆)
    priority_queue<ListNode*, vector<ListNode*>, CompareNode> minHeap;

    // 将所有非空链表的头节点加入优先队列
    for (ListNode* list : lists) {
        if (list != nullptr) {
            minHeap.push(list);
        }
    }

    // 创建哨兵节点
    ListNode dummy(-1);
    ListNode* current = &dummy;

    // 从优先队列中依次取出最小节点
    while (!minHeap.empty()) {
        ListNode* node = minHeap.top();
        minHeap.pop();
        current->next = node;
        current = current->next;

        if (node->next != nullptr) {
            minHeap.push(node->next);
        }
    }
}

```

```
}

    return dummy.next;
}

/***
 * 测试方法
 */
static void test() {
    cout << "==== LintCode 104. 合并 k 个排序链表测试 ===" << endl;

    vector<int> arr1 = {2, 4};
    vector<int> arr2 = {1, 3, 5};
    vector<int> arr3 = {6, 7};
    ListNode* l1 = ListNode::createList(arr1);
    ListNode* l2 = ListNode::createList(arr2);
    ListNode* l3 = ListNode::createList(arr3);
    vector<ListNode*> lists = {l1, l2, l3};

    cout << "链表 1: ";
    ListNode::printList(lists[0]);
    cout << "链表 2: ";
    ListNode::printList(lists[1]);
    cout << "链表 3: ";
    ListNode::printList(lists[2]);

    ListNode* result = mergeKLists(lists);
    cout << "合并结果: ";
    ListNode::printList(result);
    cout << endl;

    // 释放内存
    ListNode::deleteList(result);
    // 注意: l1, l2, l3 已经被合并到 result 中, 不需要单独释放
}

};

/***
 * 题目 9: LeetCode 86. 分隔链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/partition-list/
 */
class PartitionListSolution {
```

```

public:
    static ListNode* partition(ListNode* head, int x) {
        ListNode lessHead(0);
        ListNode greaterHead(0);
        ListNode* less = &lessHead;
        ListNode* greater = &greaterHead;

        while (head != nullptr) {
            if (head->val < x) {
                less->next = head;
                less = less->next;
            } else {
                greater->next = head;
                greater = greater->next;
            }
            head = head->next;
        }

        greater->next = nullptr;
        less->next = greaterHead.next;

        return lessHead.next;
    }

    static void test() {
        cout << "==== LeetCode 86. 分隔链表测试 ===" << endl;
        vector<int> arr = {1, 4, 3, 2, 5, 2};
        ListNode* list = ListNode::createList(arr);
        cout << "原链表: ";
        ListNode::printList(list);
        ListNode* result = partition(list, 3);
        cout << "分隔后(x=3): ";
        ListNode::printList(result);
        cout << endl;
        ListNode::deleteList(result);
    }
};

/***
 * 题目 10: LeetCode 141. 环形链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/linked-list-cycle/
 */

```

* 题目描述:

* 给你一个链表的头节点 head , 判断链表中是否有环。

* 如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。

*

* 解法分析:

* 1. 快慢指针法 (Floyd 判圈算法) - 时间复杂度: $O(n)$, 空间复杂度: $O(1)$

*

* 解题思路:

* 使用两个指针，一个快指针和一个慢指针。快指针每次移动两步，慢指针每次移动一步。

* 如果链表中存在环，快指针最终会追上慢指针；如果不存在环，快指针会先到达链表末尾。

*/

```

class LinkedListCycleSolution {
public:
    /**
     * 解法: 快慢指针法 (Floyd 判圈算法)
     * 时间复杂度:  $O(n)$  - 最多遍历链表两次
     * 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 初始化快慢指针都指向头节点
     * 2. 快指针每次移动两步，慢指针每次移动一步
     * 3. 如果存在环，快指针会追上慢指针
     * 4. 如果不存在环，快指针会先到达链表末尾
     */
    static bool hasCycle(ListNode* head) {
        // 边界条件检查
        if (head == nullptr || head->next == nullptr) {
            return false;
        }

        // 初始化快慢指针
        ListNode* slow = head;
        ListNode* fast = head;

        // 移动指针
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;          // 慢指针每次移动一步
            fast = fast->next->next;   // 快指针每次移动两步

            // 如果快慢指针相遇，说明存在环
            if (slow == fast) {
                return true;
            }
        }
    }
}

```

```
}

// 如果快指针到达链表末尾，说明不存在环
return false;
}

/***
 * 测试方法
 */
static void test() {
    cout << "==== LeetCode 141. 环形链表测试 ===" << endl;

    // 测试用例 1: 无环链表
    cout << "测试用例 1: 无环链表" << endl;
    vector<int> arr1 = {1, 2, 3, 4};
    ListNode* list1 = ListNode::createList(arr1);
    cout << "链表: ";
    ListNode::printList(list1);
    cout << "是否有环: " << (hasCycle(list1) ? "true" : "false") << endl;
    ListNode::deleteList(list1);

    // 测试用例 2: 有环链表 (构造环)
    cout << "测试用例 2: 有环链表" << endl;
    vector<int> arr2 = {1, 2, 3, 4};
    ListNode* list2 = ListNode::createList(arr2);
    // 构造环: 将尾节点指向第二个节点
    ListNode* cur = list2;
    while (cur->next != nullptr) {
        cur = cur->next;
    }
    cur->next = list2->next; // 尾节点指向第二个节点
    cout << "链表: 1 -> 2 -> 3 -> 4 -> 2 (形成环)" << endl;
    cout << "是否有环: " << (hasCycle(list2) ? "true" : "false") << endl;
    // 注意: 有环的链表不能直接删除, 这里为了测试通过, 我们手动断开环
    cur->next = nullptr;
    ListNode::deleteList(list2);

    // 测试用例 3: 单节点无环
    cout << "测试用例 3: 单节点无环" << endl;
    ListNode* list3 = new ListNode(1);
    cout << "链表: 1" << endl;
    cout << "是否有环: " << (hasCycle(list3) ? "true" : "false") << endl;
    delete list3;
```

```

// 测试用例 4: 空链表
cout << "测试用例 4: 空链表" << endl;
ListNode* list4 = nullptr;
cout << "链表: null" << endl;
cout << "是否有环: " << (hasCycle(list4) ? "true" : "false") << endl;
cout << endl;
}

};

/***
 * 题目 11: LeetCode 142. 环形链表 II
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/linked-list-cycle-ii/
 *
 * 题目描述:
 * 给定一个链表的头节点 head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
 *
 * 解法分析:
 * 1. 快慢指针法 - 时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 解题思路:
 * 使用快慢指针找到环后，将快指针重新指向头节点，然后快慢指针都每次移动一步，
 * 当它们再次相遇时，相遇点就是环的入口节点。
 */

class LinkedListCycleIISolution {
public:
    /**
     * 解法: 快慢指针法
     * 时间复杂度: O(n) - 最多遍历链表三次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用快慢指针找到环
     * 2. 将快指针重新指向头节点
     * 3. 快慢指针都每次移动一步
     * 4. 再次相遇点就是环的入口
     */
    static ListNode* detectCycle(ListNode* head) {
        // 边界条件检查
        if (head == nullptr || head->next == nullptr) {
            return nullptr;
        }

```

```

// 第一阶段：使用快慢指针判断是否有环
ListNode* slow = head;
ListNode* fast = head;

while (fast != nullptr && fast->next != nullptr) {
    slow = slow->next;
    fast = fast->next->next;

    // 如果快慢指针相遇，说明存在环
    if (slow == fast) {
        break;
    }
}

// 如果没有环，返回 nullptr
if (fast == nullptr || fast->next == nullptr) {
    return nullptr;
}

// 第二阶段：找到环的入口
// 将快指针重新指向头节点
fast = head;
// 快慢指针都每次移动一步，直到相遇
while (slow != fast) {
    slow = slow->next;
    fast = fast->next;
}

// 相遇点就是环的入口
return slow;
}

/***
 * 测试方法
 */
static void test() {
    cout << "==== LeetCode 142. 环形链表 II 测试 ===" << endl;

    // 测试用例 1：无环链表
    cout << "测试用例 1：无环链表" << endl;
    vector<int> arr1 = {1, 2, 3, 4};
    ListNode* list1 = ListNode::createList(arr1);
}

```

```

cout << "链表: ";
ListNode::printList(list1);
ListNode* cycleStart1 = detectCycle(list1);
cout << "环的入口: " << (cycleStart1 != nullptr ? to_string(cycleStart1->val) : "null")
<< endl;
ListNode::deleteList(list1);

// 测试用例 2: 有环链表 (构造环)
cout << "测试用例 2: 有环链表" << endl;
vector<int> arr2 = {1, 2, 3, 4};
ListNode* list2 = ListNode::createList(arr2);
// 构造环: 将尾节点指向第二个节点
ListNode* cur = list2;
while (cur->next != nullptr) {
    cur = cur->next;
}
cur->next = list2->next; // 尾节点指向第二个节点
cout << "链表: 1 -> 2 -> 3 -> 4 -> 2 (形成环)" << endl;
ListNode* cycleStart2 = detectCycle(list2);
cout << "环的入口: " << (cycleStart2 != nullptr ? to_string(cycleStart2->val) : "null")
<< endl;
// 注意: 有环的链表不能直接删除, 这里为了测试通过, 我们手动断开环
cur->next = nullptr;
ListNode::deleteList(list2);

cout << endl;
}

};

/***
 * 题目 12: LeetCode 160. 相交链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/intersection-of-two-linked-lists/
 *
 * 题目描述:
 * 给你两个单链表的头节点 headA 和 headB , 请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点, 返回 null 。
 *
 * 解法分析:
 * 1. 双指针法 - 时间复杂度: O(m+n), 空间复杂度: O(1)
 *
 * 解题思路:
 * 使用两个指针分别遍历两个链表, 当一个指针到达链表末尾时, 将其指向另一个链表的头节点。

```

```

* 如果两个链表相交，两个指针会在相交节点相遇；如果不相交，两个指针会同时到达链表末尾。
*/
class IntersectionOfTwoLinkedListsSolution {
public:
    /**
     * 解法：双指针法
     * 时间复杂度：O(m+n) - 最多遍历两个链表各两次
     * 空间复杂度：O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想：
     * 1. 使用两个指针分别遍历两个链表
     * 2. 当指针到达链表末尾时，将其指向另一个链表的头节点
     * 3. 如果两个链表相交，两个指针会在相交节点相遇
     * 4. 如果不相交，两个指针会同时到达链表末尾
    */
    static ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {
        // 边界条件检查
        if (headA == nullptr || headB == nullptr) {
            return nullptr;
        }

        // 初始化两个指针
        ListNode* pointerA = headA;
        ListNode* pointerB = headB;

        // 当两个指针不相等时继续遍历
        while (pointerA != pointerB) {
            // 当指针到达链表末尾时，将其指向另一个链表的头节点
            pointerA = (pointerA == nullptr) ? headB : pointerA->next;
            pointerB = (pointerB == nullptr) ? headA : pointerB->next;
        }

        // 返回相交节点或 nullptr
        return pointerA;
    }

    /**
     * 测试方法
     */
    static void test() {
        cout << "===" LeetCode 160. 相交链表测试 ===" << endl;
        // 测试用例 1：相交链表
    }
}

```

```

cout << "测试用例 1: 相交链表" << endl;
vector<int> commonArr = {8, 4, 5};
vector<int> arrA = {4, 1};
vector<int> arrB = {5, 6, 1};

ListNode* common = ListNode::createList(commonArr);
ListNode* listA = ListNode::createList(arrA);
ListNode* listB = ListNode::createList(arrB);

// 构造相交链表
ListNode* curA = listA;
while (curA->next != nullptr) {
    curA = curA->next;
}
curA->next = common;

ListNode* curB = listB;
while (curB->next != nullptr) {
    curB = curB->next;
}
curB->next = common;

cout << "链表 A: 4 -> 1 -> 8 -> 4 -> 5" << endl;
cout << "链表 B: 5 -> 6 -> 1 -> 8 -> 4 -> 5" << endl;
ListNode* intersection1 = getIntersectionNode(listA, listB);
cout << "相交节点: " << (intersection1 != nullptr ? to_string(intersection1->val) :
"null") << endl;

// 注意: 相交链表需要特殊处理内存释放
curA->next = nullptr;
curB->next = nullptr;
ListNode::deleteList(listA);
ListNode::deleteList(listB);
ListNode::deleteList(common);

// 测试用例 2: 不相交链表
cout << "测试用例 2: 不相交链表" << endl;
vector<int> arrC = {1, 2, 3};
vector<int> arrD = {4, 5, 6};
ListNode* listC = ListNode::createList(arrC);
ListNode* listD = ListNode::createList(arrD);
cout << "链表 C: 1 -> 2 -> 3" << endl;
cout << "链表 D: 4 -> 5 -> 6" << endl;

```

```

        ListNode* intersection2 = getIntersectionNode(listC, listD);
        cout << "相交节点: " << (intersection2 != nullptr ? to_string(intersection2->val) :
    "null") << endl;
        ListNode::deleteList(listC);
        ListNode::deleteList(listD);

        cout << endl;
    }
};

/***
 * 题目 13: LeetCode 206. 反转链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/reverse-linked-list/
 *
 * 题目描述:
 * 给你单链表的头节点 head，请你反转链表，并返回反转后的链表。
 *
 * 解法分析:
 * 1. 迭代法 - 时间复杂度: O(n), 空间复杂度: O(1)
 * 2. 递归法 - 时间复杂度: O(n), 空间复杂度: O(n)
 *
 * 解题思路:
 * 迭代法: 使用三个指针分别指向前一个节点、当前节点和下一个节点，逐个反转节点的指向。
 * 递归法: 递归到链表末尾，然后在回溯过程中反转节点的指向。
 */
class ReverseLinkedListSolution {
public:
    /**
     * 解法 1: 迭代法 (推荐)
     * 时间复杂度: O(n) - 需要遍历链表一次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用三个指针: prev(前一个节点)、current(当前节点)、next(下一个节点)
     * 2. 逐个反转节点的指向
     * 3. 移动指针继续处理下一个节点
     */
    static ListNode* reverseListIterative(ListNode* head) {
        // 初始化指针
        ListNode* prev = nullptr;
        ListNode* current = head;

```

```

// 遍历链表
while (current != nullptr) {
    // 保存下一个节点
    ListNode* next = current->next;
    // 反转当前节点的指向
    current->next = prev;
    // 移动指针
    prev = current;
    current = next;
}

// 返回新的头节点
return prev;
}

/***
 * 解法 2：递归法
 * 时间复杂度：O(n) - 需要遍历链表一次
 * 空间复杂度：O(n) - 递归调用栈的深度
 *
 * 核心思想：
 * 1. 递归到链表末尾
 * 2. 在回溯过程中反转节点的指向
 */
static ListNode* reverseListRecursive(ListNode* head) {
    // 递归终止条件
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // 递归处理下一个节点
    ListNode* newHead = reverseListRecursive(head->next);
    // 反转当前节点和下一个节点的连接
    head->next->next = head;
    head->next = nullptr;

    // 返回新的头节点
    return newHead;
}

/***
 * 测试方法
*/

```

```
static void test() {
    cout << "==== LeetCode 206. 反转链表测试 ===" << endl;

    // 测试用例 1: 正常链表
    cout << "测试用例 1: 正常链表" << endl;
    vector<int> arr1 = {1, 2, 3, 4, 5};
    ListNode* list1 = ListNode::createList(arr1);
    cout << "原链表: ";
    ListNode::printList(list1);
    ListNode* reversed1 = reverseListIterative(list1);
    cout << "迭代法反转后: ";
    ListNode::printList(reversed1);
    ListNode::deleteList(reversed1);

    // 重新创建测试数据
    ListNode* list2 = ListNode::createList(arr1);
    ListNode* reversed2 = reverseListRecursive(list2);
    cout << "递归法反转后: ";
    ListNode::printList(reversed2);
    cout << endl;
    ListNode::deleteList(reversed2);

    // 测试用例 2: 单节点链表
    cout << "测试用例 2: 单节点链表" << endl;
    ListNode* list3 = new ListNode(1);
    cout << "原链表: ";
    ListNode::printList(list3);
    ListNode* reversed3 = reverseListIterative(list3);
    cout << "反转后: ";
    ListNode::printList(reversed3);
    delete reversed3;
    cout << endl;

    // 测试用例 3: 空链表
    cout << "测试用例 3: 空链表" << endl;
    ListNode* list4 = nullptr;
    cout << "原链表: ";
    ListNode::printList(list4);
    ListNode* reversed4 = reverseListIterative(list4);
    cout << "反转后: ";
    ListNode::printList(reversed4);
    cout << endl;
}
```

```

};

/**
 * 题目 14: LeetCode 234. 回文链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/palindrome-linked-list/
 *
 * 题目描述:
 * 给你一个单链表的头节点 head , 请你判断该链表是否为回文链表。如果是，返回 true ；否则，返回 false 。
 *
 * 解法分析:
 * 1. 快慢指针 + 反转链表 - 时间复杂度: O(n) , 空间复杂度: O(1)
 *
 * 解题思路:
 * 1. 使用快慢指针找到链表中点
 * 2. 反转后半部分链表
 * 3. 比较前半部分和反转后的后半部分
 * 4. 恢复链表结构(可选)
 */

class PalindromeLinkedListSolution {
public:
    /**
     * 解法: 快慢指针 + 反转链表
     * 时间复杂度: O(n) - 需要遍历链表多次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用快慢指针找到链表中点
     * 2. 反转后半部分链表
     * 3. 比较前半部分和反转后的后半部分
     */
    static bool isPalindrome(ListNode* head) {
        // 边界条件检查
        if (head == nullptr || head->next == nullptr) {
            return true;
        }

        // 第一步: 使用快慢指针找到链表中点
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast->next != nullptr && fast->next->next != nullptr) {

```

```

    slow = slow->next;
    fast = fast->next->next;
}

// 第二步：反转后半部分链表
ListNode* secondHalf = reverseList(slow->next);

// 第三步：比较前半部分和反转后的后半部分
ListNode* firstHalf = head;
ListNode* secondHalfCopy = secondHalf; // 保存用于恢复
bool isPalindrome = true;

while (secondHalf != nullptr) {
    if (firstHalf->val != secondHalf->val) {
        isPalindrome = false;
        break;
    }
    firstHalf = firstHalf->next;
    secondHalf = secondHalf->next;
}

// 第四步：恢复链表结构(可选)
slow->next = reverseList(secondHalfCopy);

return isPalindrome;
}

/**
 * 反转链表的辅助函数
 */
static ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;

    while (current != nullptr) {
        ListNode* next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    return prev;
}

```

```
/**  
 * 测试方法  
 */  
  
static void test() {  
    cout << "== LeetCode 234. 回文链表测试 ==" << endl;  
  
    // 测试用例 1: 回文链表  
    cout << "测试用例 1: 回文链表" << endl;  
    vector<int> arr1 = {1, 2, 2, 1};  
    ListNode* list1 = ListNode::createList(arr1);  
    cout << "链表: ";  
    ListNode::printList(list1);  
    cout << "是否为回文链表: " << (isPalindrome(list1) ? "true" : "false") << endl;  
    ListNode::deleteList(list1);  
  
    // 测试用例 2: 非回文链表  
    cout << "测试用例 2: 非回文链表" << endl;  
    vector<int> arr2 = {1, 2, 3, 4};  
    ListNode* list2 = ListNode::createList(arr2);  
    cout << "链表: ";  
    ListNode::printList(list2);  
    cout << "是否为回文链表: " << (isPalindrome(list2) ? "true" : "false") << endl;  
    ListNode::deleteList(list2);  
  
    // 测试用例 3: 单节点链表  
    cout << "测试用例 3: 单节点链表" << endl;  
    ListNode* list3 = new ListNode(1);  
    cout << "链表: ";  
    ListNode::printList(list3);  
    cout << "是否为回文链表: " << (isPalindrome(list3) ? "true" : "false") << endl;  
    delete list3;  
  
    cout << endl;  
}  
};  
  
/**  
 * 题目 15: LeetCode 19. 删除链表的倒数第 N 个结点  
 * 来源: LeetCode  
 * 链接: https://leetcode.cn/problems/remove-nth-node-from-end-of-list/  
 *  
 * 题目描述:  
 */
```

```
* 给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。
```

```
*
```

```
* 解法分析：
```

```
* 1. 快慢指针法 - 时间复杂度：O(n)，空间复杂度：O(1)
```

```
*
```

```
* 解题思路：
```

```
* 使用两个指针，快指针先移动 n+1 步，然后快慢指针同时移动，
```

```
* 当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点。
```

```
*/
```

```
class RemoveNthNodeFromEndOfListSolution {
```

```
public:
```

```
/**
```

```
* 解法：快慢指针法
```

```
* 时间复杂度：O(n) - 需要遍历链表一次
```

```
* 空间复杂度：O(1) - 只使用了常数级别的额外空间
```

```
*
```

```
* 核心思想：
```

```
* 1. 使用哨兵节点简化边界处理
```

```
* 2. 快指针先移动 n+1 步
```

```
* 3. 快慢指针同时移动
```

```
* 4. 当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点
```

```
*/
```

```
static ListNode* removeNthFromEnd(ListNode* head, int n) {
```

```
    // 创建哨兵节点，简化边界处理
```

```
    ListNode dummy(0);
```

```
    dummy.next = head;
```

```
    // 初始化快慢指针
```

```
    ListNode* fast = &dummy;
```

```
    ListNode* slow = &dummy;
```

```
    // 快指针先移动 n+1 步
```

```
    for (int i = 0; i <= n; i++) {
```

```
        fast = fast->next;
```

```
}
```

```
    // 快慢指针同时移动
```

```
    while (fast != nullptr) {
```

```
        fast = fast->next;
```

```
        slow = slow->next;
```

```
}
```

```
    // 删除倒数第 n 个节点
```

```
slow->next = slow->next->next;

// 返回头节点
return dummy.next;
}

/***
 * 测试方法
 */
static void test() {
    cout << "==== LeetCode 19. 删除链表的倒数第 N 个结点测试 ===" << endl;

    // 测试用例 1: 删除中间节点
    cout << "测试用例 1: 删除中间节点" << endl;
    vector<int> arr1 = {1, 2, 3, 4, 5};
    ListNode* list1 = ListNode::createList(arr1);
    cout << "原链表: ";
    ListNode::printList(list1);
    ListNode* result1 = removeNthFromEnd(list1, 2);
    cout << "删除倒数第 2 个节点后: ";
    ListNode::printList(result1);
    ListNode::deleteList(result1);

    // 测试用例 2: 删除头节点
    cout << "测试用例 2: 删除头节点" << endl;
    vector<int> arr2 = {1, 2, 3, 4, 5};
    ListNode* list2 = ListNode::createList(arr2);
    cout << "原链表: ";
    ListNode::printList(list2);
    ListNode* result2 = removeNthFromEnd(list2, 5);
    cout << "删除倒数第 5 个节点后: ";
    ListNode::printList(result2);
    ListNode::deleteList(result2);

    // 测试用例 3: 删除尾节点
    cout << "测试用例 3: 删除尾节点" << endl;
    vector<int> arr3 = {1, 2, 3, 4, 5};
    ListNode* list3 = ListNode::createList(arr3);
    cout << "原链表: ";
    ListNode::printList(list3);
    ListNode* result3 = removeNthFromEnd(list3, 1);
    cout << "删除倒数第 1 个节点后: ";
    ListNode::printList(result3);
```

```

ListNode::deleteList(result3);

cout << endl;
}

};

/***
 * 算法总结与技巧提升
 */
class AlgorithmSummary {
public:
    static void printSummary() {
        cout << "===== 链表合并算法总结 =====" << endl;
        cout << "1. 核心算法技巧:" << endl;
        cout << " - 双指针法: 适用于两个有序序列的合并, 时间复杂度 O(m+n)" << endl;
        cout << " - 优先队列法: 适用于 K 个有序序列的合并, 时间复杂度 O(N*logK)" << endl;
        cout << " - 分治法: 适用于 K 个序列的归并, 时间复杂度 O(N*logK)" << endl;
        cout << " - 哨兵节点: 简化链表操作的边界处理, 提高代码可读性" << endl;
        cout << " - 原地修改: 避免额外空间开销, 适用于数组合并等场景" << endl;
        cout << endl;
        cout << "2. 工程化考量:" << endl;
        cout << " - 异常处理: 处理空链表、单节点链表等边界情况" << endl;
        cout << " - 内存管理: 在 C++ 中需要注意释放链表内存, 避免内存泄漏" << endl;
        cout << " - 性能优化: 对于大规模数据, 优先队列的常数项优化很重要" << endl;
        cout << " - 线程安全: 在多线程环境下需要考虑同步问题" << endl;
        cout << endl;
        cout << "3. 调试技巧:" << endl;
        cout << " - 打印中间状态: 使用 cout 跟踪指针移动" << endl;
        cout << " - 边界测试: 测试空输入、单元素输入、极端值等情况" << endl;
        cout << " - 断言验证: 使用 assert 验证关键条件是否满足" << endl;
        cout << endl;
        cout << "4. 拓展应用:" << endl;
        cout << " - 归并排序: 链表排序的最佳选择之一" << endl;
        cout << " - 多路归并: 外部排序的基础算法" << endl;
        cout << " - 数据流处理: 实时合并多个有序数据流" << endl;
        cout << "=====\\n" << endl;
    }
};

/***
 * 综合测试函数
 */
void runAllTests() {

```

```

MergeTwoSortedListsSolution::test();
MergeKSortedListsSolution::test();
MergeSortedArraySolution::test();
SortListSolution::test();
AddTwoNumbersSolution::test();
SwapNodesInPairsSolution::test();
NowCoderMergeSortedListsSolution::test();
LintCodeMergeKListsSolution::test();
PartitionListSolution::test();

// 新增题目的测试
LinkedListCycleSolution::test();
LinkedListCycleIISolution::test();
IntersectionOfTwoLinkedListsSolution::test();
ReverseLinkedListSolution::test();
PalindromeLinkedListSolution::test();
RemoveNthNodeFromEndOfListSolution::test();

AlgorithmSummary::printSummary();
}

/***
 * 主函数 - 运行所有测试
 */
int main() {
    runAllTests();
    return 0;
}
=====
```

文件: MergeTwoLists.java

```

/*
 * 合并两个有序链表及相关题目扩展 (Java 版本)
 *
 * 算法专题: 链表合并与相关算法
 * 覆盖平台: LeetCode、牛客网、LintCode、剑指 Offer 等
 * 语言特性: Java 8+, 面向对象设计, 异常处理, 性能优化
 *
 * 工程化考量:
 * 1. 内存管理: 垃圾回收机制, 避免内存泄漏
 * 2. 异常安全: 完整的异常处理机制

```

- * 3. 性能优化：算法优化，数据结构选择
- * 4. 可测试性：单元测试框架，边界条件覆盖
- * 5. 可维护性：清晰的代码结构，详细的注释说明
- *
- * 复杂度分析体系：
 - * - 时间复杂度：从理论分析到实际性能考量
 - * - 空间复杂度：内存使用优化策略
 - * - 常数项分析：实际运行效率的关键因素
- *
- * 算法应用场景：
 - * - 大数据处理：外部排序，多路归并
 - * - 实时系统：数据流合并处理
 - * - 分布式计算：多节点结果合并
 - * - 数据库系统：索引合并优化
- *
- * 主要题目：
 - * 1. LeetCode 21. 合并两个有序链表（基础题）
 - * 2. LeetCode 23. 合并 K 个升序链表（进阶题）
 - * 3. LeetCode 88. 合并两个有序数组（变种题）
 - * 4. LeetCode 148. 排序链表（应用扩展）
 - * 5. LeetCode 2. 两数相加（链表操作）
 - * 6. LeetCode 24. 两两交换链表中的节点（链表变换）
 - * 7. 牛客 NC33. 合并两个排序的链表（国内平台）
 - * 8. LintCode 104. 合并 k 个排序链表（国际平台）
 - * 9. LeetCode 86. 分隔链表（链表分割）
- *
- * 解题思路技巧总结：
 - * 1. 双指针法：适用于两个有序序列的合并，时间复杂度 $O(m+n)$
 - * 2. 优先队列（堆）：适用于 K 个有序序列的合并，时间复杂度 $O(N \log K)$
 - * 3. 分治法：将 K 个序列问题分解为多个两个序列问题，时间复杂度 $O(N \log K)$
 - * 4. 哨兵节点：简化链表操作的边界处理，提高代码可读性
 - * 5. 原地修改：充分利用已有空间，减少额外空间使用
 - * 6. 递归与迭代：不同场景下的选择策略
- *
- * 时间复杂度分析：
 - * 1. 合并两个链表： $O(m+n)$ ，m 和 n 分别是两个链表的长度
 - * 2. 合并 K 个链表（优先队列）： $O(N \log K)$ ，N 是所有节点总数，K 是链表数量
 - * 3. 合并 K 个链表（分治）： $O(N \log K)$
 - * 4. 合并两个数组： $O(m+n)$
 - * 5. 链表排序： $O(n \log n)$ ，归并排序最优
- *
- * 空间复杂度分析：
 - * 1. 合并两个链表： $O(1)$ ，原地操作

- * 2. 合并 K 个链表(优先队列): $O(K)$, 堆的大小
- * 3. 合并 K 个链表(分治): $O(\log K)$, 递归栈深度
- * 4. 合并两个数组: $O(1)$, 原地操作
- * 5. 链表排序: $O(1)$ 或 $O(\log n)$, 取决于实现方式
- *
- * 安全与稳定性:
 - * - 空指针检查: 所有链表操作前的边界检查
 - * - 异常处理: try-catch 块, 参数验证
 - * - 输入验证: 参数合法性检查
- *
- * 调试与测试:
 - * - 单元测试: 每个算法的独立测试用例
 - * - 边界测试: 空输入、单元素、极端值等
 - * - 性能测试: 大规模数据下的性能表现
- *
- * 学习路径建议:
 - * 1. 基础掌握: LeetCode 21 -> 牛客 NC33
 - * 2. 进阶提升: LeetCode 23 -> LintCode 104
 - * 3. 综合应用: LeetCode 148 -> LeetCode 2
 - * 4. 拓展思维: LeetCode 24 -> LeetCode 86
- *
- * 面试重点:
 - * - 算法思路清晰表达
 - * - 时间空间复杂度分析
 - * - 边界条件处理能力
 - * - 代码实现简洁优雅
 - * - 工程化考量意识
- */

```
import java.util.*;  
  
/**  
 * 链表节点定义类  
 *  
 * 设计要点:

- - 使用构造函数简化节点创建
- - 提供静态工具方法便于测试
- - 包含输入验证和异常处理

  
 * 注意事项:

- - Java 垃圾回收机制自动管理内存
- - 注意避免循环引用
- - 考虑线程安全性

```

```
/*
class ListNode {
    int val;
    ListNode next;

    // 构造函数
    ListNode() {}

    ListNode(int val) { this.val = val; }

    ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }

    /**
     * 从数组创建链表（测试工具方法）
     *
     * 复杂度分析：
     * - 时间复杂度：O(n)，n 为数组长度
     * - 空间复杂度：O(n)，需要创建 n 个节点
     *
     * 使用场景：单元测试、算法演示
     *
     * @param arr 整数数组
     * @return 链表头节点
     * @throws IllegalArgumentException 如果输入数组为 null
     */
    public static ListNode createList(int[] arr) {
        // 输入验证
        if (arr == null) {
            throw new IllegalArgumentException("输入数组不能为 null");
        }
        if (arr.length == 0) return null;

        ListNode head = new ListNode(arr[0]);
        ListNode cur = head;
        for (int i = 1; i < arr.length; i++) {
            cur.next = new ListNode(arr[i]);
            cur = cur.next;
        }
        return head;
    }

    /**

```

```
* 打印链表内容（调试工具方法）
*
* 复杂度分析:
* - 时间复杂度: O(n), n 为链表长度
* - 空间复杂度: O(1), 只使用常数空间
*
* 使用场景: 调试、结果验证
*
* @param head 链表头节点
*/
public static void printList(ListNode head) {
    ListNode cur = head;
    while (cur != null) {
        System.out.print(cur.val);
        if (cur.next != null) {
            System.out.print(" -> ");
        }
        cur = cur.next;
    }
    System.out.println();
}

/***
* 获取链表长度（工具方法）
*
* 复杂度分析:
* - 时间复杂度: O(n), 需要遍历整个链表
* - 空间复杂度: O(1), 只使用常数空间
*
* @param head 链表头节点
* @return 链表长度
*/
public static int getLength(ListNode head) {
    int length = 0;
    ListNode cur = head;
    while (cur != null) {
        length++;
        cur = cur.next;
    }
    return length;
}

/***
```

```

* 验证链表是否有序 (测试工具方法)
*
* 复杂度分析:
* - 时间复杂度: O(n), 需要遍历整个链表
* - 空间复杂度: O(1), 只使用常数空间
*
* @param head 链表头节点
* @return 是否有序 (升序)
*/
public static boolean isSorted(ListNode head) {
    if (head == null || head.next == null) return true;

    ListNode cur = head;
    while (cur.next != null) {
        if (cur.val > cur.next.val) {
            return false;
        }
        cur = cur.next;
    }
    return true;
}

/**
* 题目 1: LeetCode 21. 合并两个有序链表
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/merge-two-sorted-lists/
*
* 题目描述:
* 将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。
*
* 解法分析:
* 1. 迭代法 - 时间复杂度: O(m+n), 空间复杂度: O(1)
* 2. 递归法 - 时间复杂度: O(m+n), 空间复杂度: O(m+n)
*
* 解题思路:
* 使用双指针分别指向两个链表的当前节点，比较节点值的大小，
* 将较小的节点连接到结果链表中，移动对应指针，重复此过程直到某一链表遍历完。
* 最后将未遍历完的链表剩余部分直接连接到结果链表末尾。
*/
class MergeTwoSortedListsSolution {
    /**
     * 解法 1: 迭代法 (推荐)

```

```

* 时间复杂度: O(m+n) - m 和 n 分别是两个链表的长度
* 空间复杂度: O(1) - 只使用了常数级别的额外空间
*
* 核心思想:
* 1. 使用哨兵节点简化边界处理
* 2. 双指针分别遍历两个链表
* 3. 比较节点值, 将较小节点连接到结果链表
* 4. 处理剩余节点
*/
public static ListNode mergeTwoListsIterative(ListNode list1, ListNode list2) {
    // 创建哨兵节点, 简化边界处理
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    // 双指针遍历两个链表
    while (list1 != null && list2 != null) {
        // 比较两个链表当前节点的值
        if (list1.val <= list2.val) {
            current.next = list1;
            list1 = list1.next;
        } else {
            current.next = list2;
            list2 = list2.next;
        }
        current = current.next;
    }

    // 连接剩余节点
    current.next = (list1 != null) ? list1 : list2;

    // 返回合并后的链表
    return dummy.next;
}

/**
* 解法 2: 递归法
* 时间复杂度: O(m+n) - 每个节点访问一次
* 空间复杂度: O(m+n) - 递归调用栈的深度
*
* 核心思想:
* 1. 递归终止条件: 其中一个链表为空
* 2. 递归处理: 选择较小节点作为当前节点, 递归处理剩余部分
* 3. 返回当前节点

```

```
/*
public static ListNode mergeTwoListsRecursive(ListNode list1, ListNode list2) {
    // 递归终止条件
    if (list1 == null) return list2;
    if (list2 == null) return list1;

    // 递归处理
    if (list1.val <= list2.val) {
        list1.next = mergeTwoListsRecursive(list1.next, list2);
        return list1;
    } else {
        list2.next = mergeTwoListsRecursive(list1, list2.next);
        return list2;
    }
}

/**
 * 测试方法
 * 测试用例覆盖:
 * - 正常情况
 * - 空链表测试
 * - 两个空链表
 * - 包含重复元素测试
 * - 极端值测试
 */
public static void test() {
    System.out.println("== LeetCode 21. 合并两个有序链表测试 ==");
    System.out.println("测试用例覆盖: 正常情况、边界条件、极端值");

    // 测试用例 1: 正常情况
    System.out.println("测试用例 1: 正常情况");
    ListNode list1 = ListNode.createList(new int[]{1, 2, 4});
    ListNode list2 = ListNode.createList(new int[]{1, 3, 4});
    System.out.print("链表 1: ");
    ListNode.printList(list1);
    System.out.print("链表 2: ");
    ListNode.printList(list2);

    ListNode result1 = mergeTwoListsIterative(list1, list2);
    System.out.print("迭代法结果: ");
    ListNode.printList(result1);

    // 重新创建测试数据
}
```

```
list1 = ListNode.createList(new int[]{1, 2, 4});
list2 = ListNode.createList(new int[]{1, 3, 4});
ListNode result2 = mergeTwoListsRecursive(list1, list2);
System.out.print("递归法结果: ");
ListNode.printList(result2);

// 测试用例 2: 空链表
System.out.println("测试用例 2: 空链表测试");
ListNode list3 = null;
ListNode list4 = ListNode.createList(new int[]{0});
ListNode result3 = mergeTwoListsIterative(list3, list4);
System.out.print("空链表测试: ");
ListNode.printList(result3);

// 测试用例 3: 两个空链表
System.out.println("测试用例 3: 两个空链表");
ListNode list5 = null;
ListNode list6 = null;
ListNode result4 = mergeTwoListsIterative(list5, list6);
System.out.print("两个空链表: ");
ListNode.printList(result4);

// 测试用例 4: 包含重复元素
System.out.println("测试用例 4: 包含重复元素");
ListNode list7 = ListNode.createList(new int[]{1, 1, 2, 3});
ListNode list8 = ListNode.createList(new int[]{1, 2, 2, 4});
ListNode result5 = mergeTwoListsIterative(list7, list8);
System.out.print("包含重复元素结果: ");
ListNode.printList(result5);

// 测试用例 5: 极端值测试
System.out.println("测试用例 5: 极端值测试");
ListNode list9 = ListNode.createList(new int[]{-10, -5, 0});
ListNode list10 = ListNode.createList(new int[]{-8, 100, 1000});
ListNode result6 = mergeTwoListsIterative(list9, list10);
System.out.print("极端值测试结果: ");
ListNode.printList(result6);

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}

/**
```

```
* 性能测试方法
*/
public static void performanceTest() {
    System.out.println("==> 性能测试 ==>");

    // 生成大规模测试数据
    int size = 10000;
    int[] arr1 = new int[size];
    int[] arr2 = new int[size];
    Random random = new Random();

    for (int i = 0; i < size; i++) {
        arr1[i] = random.nextInt(1000);
        arr2[i] = random.nextInt(1000);
    }
    Arrays.sort(arr1);
    Arrays.sort(arr2);

    ListNode list1 = ListNode.createList(arr1);
    ListNode list2 = ListNode.createList(arr2);

    // 测试迭代法性能
    long startTime = System.nanoTime();
    ListNode result1 = mergeTwoListsIterative(list1, list2);
    long endTime = System.nanoTime();
    System.out.printf("迭代法执行时间: %.3f ms\n", (endTime - startTime) / 1e6);

    // 重新生成测试数据
    list1 = ListNode.createList(arr1);
    list2 = ListNode.createList(arr2);

    // 测试递归法性能
    startTime = System.nanoTime();
    ListNode result2 = mergeTwoListsRecursive(list1, list2);
    endTime = System.nanoTime();
    System.out.printf("递归法执行时间: %.3f ms\n", (endTime - startTime) / 1e6);

    System.out.println("性能测试完成");
}

/**
 * 题目 2: LeetCode 23. 合并 K 个升序链表
```

```
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/merge-k-sorted-lists/
*
* 题目描述:
* 给你一个链表数组，每个链表都已经按升序排列。
* 请你将所有链表合并到一个升序链表中，返回合并后的链表。
*
* 解法分析:
* 1. 优先队列法（最优解） - 时间复杂度: O(N*logK)，空间复杂度: O(K)
* 2. 分治法 - 时间复杂度: O(N*logK)，空间复杂度: O(logK)
*
* 解题思路:
* 优先队列法: 维护一个大小为 K 的最小堆，堆中存放 K 个链表的头节点。
* 每次从堆中取出最小节点加入结果链表，并将该节点的下一个节点加入堆中。
* 分治法: 将 K 个链表分成两部分，分别合并后再合并两个结果。
*/
class MergeKSortedListsSolution {
    /**
     * 解法 1: 优先队列法 (推荐)
     * 时间复杂度: O(N*logK) - N 是所有节点总数, K 是链表数量
     * 空间复杂度: O(K) - 优先队列的大小
     *
     * 核心思想:
     * 1. 使用优先队列(最小堆)维护 K 个链表的当前最小节点
     * 2. 每次取出最小节点加入结果链表
     * 3. 将取出节点的下一个节点加入优先队列
     * 4. 重复直到优先队列为空
     */
    public static ListNode mergeKListsPriorityQueue(ListNode[] lists) {
        if (lists == null || lists.length == 0) return null;

        // 创建优先队列(最小堆)
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

        // 将所有非空链表的头节点加入优先队列
        for (ListNode list : lists) {
            if (list != null) {
                minHeap.offer(list);
            }
        }

        // 创建哨兵节点
        ListNode dummy = new ListNode(0);
```

```

ListNode current = dummy;

// 从优先队列中依次取出最小节点
while (!minHeap.isEmpty()) {
    // 取出最小节点
    ListNode node = minHeap.poll();

    // 加入结果链表
    current.next = node;
    current = current.next;

    // 如果该节点还有后续节点，加入优先队列
    if (node.next != null) {
        minHeap.offer(node.next);
    }
}

return dummy.next;
}

/**
 * 解法 2：分治法
 * 时间复杂度：O(N*logK) - N 是所有节点总数，K 是链表数量
 * 空间复杂度：O(logK) - 递归调用栈的深度
 *
 * 核心思想：
 * 1. 将 K 个链表分成两部分
 * 2. 递归合并每一部分
 * 3. 合并两个结果链表
 */
public static ListNode mergeKListsDivideAndConquer(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;
    return mergeKListsHelper(lists, 0, lists.length - 1);
}

private static ListNode mergeKListsHelper(ListNode[] lists, int left, int right) {
    if (left == right) return lists[left];
    if (left + 1 == right) {
        return MergeTwoSortedListsSolution.mergeTwoListsIterative(lists[left], lists[right]);
    }

    int mid = left + (right - left) / 2;
    ListNode l1 = mergeKListsHelper(lists, left, mid);

```

```
ListNode 12 = mergeKListsHelper(lists, mid + 1, right);

return MergeTwoSortedListsSolution.mergeTwoListsIterative(11, 12);
}

/**
 * 测试方法
 */
public static void test() {
    System.out.println("== LeetCode 23. 合并 K 个升序链表测试 ==");

    // 创建测试数据
    ListNode 11 = ListNode.createList(new int[]{1, 4, 5});
    ListNode 12 = ListNode.createList(new int[]{1, 3, 4});
    ListNode 13 = ListNode.createList(new int[]{2, 6});
    ListNode[] lists = {11, 12, 13};

    System.out.print("链表 1: ");
    ListNode.printList(lists[0]);
    System.out.print("链表 2: ");
    ListNode.printList(lists[1]);
    System.out.print("链表 3: ");
    ListNode.printList(lists[2]);

    // 测试优先队列法
    ListNode[] listsCopy1 = Arrays.copyOf(lists, lists.length);
    ListNode result1 = mergeKListsPriorityQueue(listsCopy1);
    System.out.print("优先队列法结果: ");
    ListNode.printList(result1);

    // 测试分治法
    ListNode[] listsCopy2 = Arrays.copyOf(lists, lists.length);
    ListNode result2 = mergeKListsDivideAndConquer(listsCopy2);
    System.out.print("分治法结果: ");
    ListNode.printList(result2);

    // 测试用例 2: 空链表数组
    System.out.println("测试用例 2: 空链表数组");
    ListNode[] emptyLists = new ListNode[0];
    ListNode result3 = mergeKListsPriorityQueue(emptyLists);
    System.out.print("空链表数组结果: ");
    ListNode.printList(result3);
```

```
// 测试用例 3: 包含空链表
System.out.println("测试用例 3: 包含空链表");
ListNode[] listsWithNull = {11, null, 13};
ListNode result4 = mergeKListsPriorityQueue(listsWithNull);
System.out.print("包含空链表结果: ");
ListNode.printList(result4);

// 测试用例 4: 单个链表
System.out.println("测试用例 4: 单个链表");
ListNode[] singleList = {11};
ListNode result5 = mergeKListsPriorityQueue(singleList);
System.out.print("单个链表结果: ");
ListNode.printList(result5);

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}
```

```
/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("== 性能测试 ==");

    // 生成大规模测试数据
    int k = 100; // 链表数量
    int n = 1000; // 每个链表的节点数
    ListNode[] lists = new ListNode[k];
    Random random = new Random();

    for (int i = 0; i < k; i++) {
        int[] arr = new int[n];
        for (int j = 0; j < n; j++) {
            arr[j] = random.nextInt(10000);
        }
        Arrays.sort(arr);
        lists[i] = ListNode.createList(arr);
    }
}
```

```
System.out.printf("测试数据规模: %d 个链表, 每个链表约%d 个节点\n", k, n);
```

```
// 测试优先队列法性能
ListNode[] listsCopy1 = Arrays.copyOf(lists, lists.length);
```

```

        long startTime = System.nanoTime();
        ListNode result1 = mergeKListsPriorityQueue(listsCopy1);
        long endTime = System.nanoTime();
        System.out.printf("优先队列法执行时间: %.3f ms\n", (endTime - startTime) / 1e6);

        // 测试分治法性能
        ListNode[] listsCopy2 = Arrays.copyOf(lists, lists.length);
        startTime = System.nanoTime();
        ListNode result2 = mergeKListsDivideAndConquer(listsCopy2);
        endTime = System.nanoTime();
        System.out.printf("分治法执行时间: %.3f ms\n", (endTime - startTime) / 1e6);

        System.out.println("性能测试完成");
    }

}

/***
 * 题目 10: LeetCode 141. 环形链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/linked-list-cycle/
 *
 * 题目描述:
 * 给你一个链表的头节点 head ，判断链表中是否有环。
 * 如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。
 *
 * 解法分析:
 * 1. 快慢指针法 (Floyd 判圈算法) - 时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 解题思路:
 * 使用两个指针，一个快指针和一个慢指针。快指针每次移动两步，慢指针每次移动一步。
 * 如果链表中存在环，快指针最终会追上慢指针；如果不存在环，快指针会先到达链表末尾。
 */
class LinkedListCycleSolution {
    /**
     * 解法: 快慢指针法 (Floyd 判圈算法)
     * 时间复杂度: O(n) - 最多遍历链表两次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 初始化快慢指针都指向头节点
     * 2. 快指针每次移动两步，慢指针每次移动一步
     * 3. 如果存在环，快指针会追上慢指针
     * 4. 如果不存在环，快指针会先到达链表末尾
    
```

```
/*
public static boolean hasCycle(ListNode head) {
    // 边界条件检查
    if (head == null || head.next == null) {
        return false;
    }

    // 初始化快慢指针
    ListNode slow = head;
    ListNode fast = head;

    // 移动指针
    while (fast != null && fast.next != null) {
        slow = slow.next;          // 慢指针每次移动一步
        fast = fast.next.next;    // 快指针每次移动两步

        // 如果快慢指针相遇，说明存在环
        if (slow == fast) {
            return true;
        }
    }

    // 如果快指针到达链表末尾，说明不存在环
    return false;
}

/***
 * 测试方法
 */
public static void test() {
    System.out.println("== LeetCode 141. 环形链表测试 ==");

    // 测试用例 1: 无环链表
    System.out.println("测试用例 1: 无环链表");
    ListNode list1 = ListNode.createList(new int[] {1, 2, 3, 4});
    System.out.print("链表: ");
    ListNode.printList(list1);
    System.out.println("是否有环: " + hasCycle(list1));

    // 测试用例 2: 有环链表 (构造环)
    System.out.println("测试用例 2: 有环链表");
    ListNode list2 = ListNode.createList(new int[] {1, 2, 3, 4});
    // 构造环: 将尾节点指向第二个节点
}
```

```

ListNode cur = list2;
while (cur.next != null) {
    cur = cur.next;
}
cur.next = list2.next; // 尾节点指向第二个节点
System.out.println("链表: 1 -> 2 -> 3 -> 4 -> 2 (形成环)");
System.out.println("是否有环: " + hasCycle(list2));

// 测试用例 3: 单节点无环
System.out.println("测试用例 3: 单节点无环");
ListNode list3 = new ListNode(1);
System.out.println("链表: 1");
System.out.println("是否有环: " + hasCycle(list3));

// 测试用例 4: 空链表
System.out.println("测试用例 4: 空链表");
ListNode list4 = null;
System.out.println("链表: null");
System.out.println("是否有环: " + hasCycle(list4));

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}

}

/***
 * 题目 11: LeetCode 142. 环形链表 II
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/linked-list-cycle-ii/
 *
 * 题目描述:
 * 给定一个链表的头节点 head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
 *
 * 解法分析:
 * 1. 快慢指针法 - 时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 解题思路:
 * 使用快慢指针找到环后，将快指针重新指向头节点，然后快慢指针都每次移动一步，
 * 当它们再次相遇时，相遇点就是环的入口节点。
 */
class LinkedListCycleIISolution {

    /**
     * 解法: 快慢指针法

```

```
* 时间复杂度: O(n) - 最多遍历链表三次
* 空间复杂度: O(1) - 只使用了常数级别的额外空间
*
* 核心思想:
* 1. 使用快慢指针找到环
* 2. 将快指针重新指向头节点
* 3. 快慢指针都每次移动一步
* 4. 再次相遇点就是环的入口
*/
public static ListNode detectCycle(ListNode head) {
    // 边界条件检查
    if (head == null || head.next == null) {
        return null;
    }

    // 第一阶段: 使用快慢指针判断是否有环
    ListNode slow = head;
    ListNode fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // 如果快慢指针相遇, 说明存在环
        if (slow == fast) {
            break;
        }
    }

    // 如果没有环, 返回 null
    if (fast == null || fast.next == null) {
        return null;
    }

    // 第二阶段: 找到环的入口
    // 将快指针重新指向头节点
    fast = head;
    // 快慢指针都每次移动一步, 直到相遇
    while (slow != fast) {
        slow = slow.next;
        fast = fast.next;
    }
}
```

```

    // 相遇点就是环的入口
    return slow;
}

/**
 * 测试方法
 */
public static void test() {
    System.out.println("==== LeetCode 142. 环形链表 II 测试 ===");

    // 测试用例 1: 无环链表
    System.out.println("测试用例 1: 无环链表");
    ListNode list1 = ListNode.createList(new int[] {1, 2, 3, 4});
    System.out.print("链表: ");
    ListNode.printList(list1);
    ListNode cycleStart1 = detectCycle(list1);
    System.out.println("环的入口: " + (cycleStart1 != null ? cycleStart1.val : "null"));

    // 测试用例 2: 有环链表 (构造环)
    System.out.println("测试用例 2: 有环链表");
    ListNode list2 = ListNode.createList(new int[] {1, 2, 3, 4});
    // 构造环: 将尾节点指向第二个节点
    ListNode cur = list2;
    while (cur.next != null) {
        cur = cur.next;
    }
    cur.next = list2.next; // 尾节点指向第二个节点
    System.out.println("链表: 1 -> 2 -> 3 -> 4 -> 2 (形成环)");
    ListNode cycleStart2 = detectCycle(list2);
    System.out.println("环的入口: " + (cycleStart2 != null ? cycleStart2.val : "null"));

    System.out.println("所有测试用例执行完成");
    System.out.println("=====");
}

}

/**
 * 题目 12: LeetCode 160. 相交链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/intersection-of-two-linked-lists/
 *
 * 题目描述:
 * 给你两个单链表的头节点 headA 和 headB，请你找出并返回两个单链表相交的起始节点。如果两个链表不

```

存在相交节点，返回 null 。

*

* 解法分析:

* 1. 双指针法 - 时间复杂度: $O(m+n)$, 空间复杂度: $O(1)$

*

* 解题思路:

* 使用两个指针分别遍历两个链表，当一个指针到达链表末尾时，将其指向另一个链表的头节点。

* 如果两个链表相交，两个指针会在相交节点相遇；如果不相交，两个指针会同时到达链表末尾。

*/

```
class IntersectionOfTwoLinkedListsSolution {
```

```
/**
```

* 解法: 双指针法

* 时间复杂度: $O(m+n)$ - 最多遍历两个链表各两次

* 空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

*

* 核心思想:

* 1. 使用两个指针分别遍历两个链表

* 2. 当指针到达链表末尾时，将其指向另一个链表的头节点

* 3. 如果两个链表相交，两个指针会在相交节点相遇

* 4. 如果不相交，两个指针会同时到达链表末尾

*/

```
public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
```

```
    // 边界条件检查
```

```
    if (headA == null || headB == null) {
```

```
        return null;
```

```
}
```

```
    // 初始化两个指针
```

```
    ListNode pointerA = headA;
```

```
    ListNode pointerB = headB;
```

```
    // 当两个指针不相等时继续遍历
```

```
    while (pointerA != pointerB) {
```

```
        // 当指针到达链表末尾时，将其指向另一个链表的头节点
```

```
        pointerA = (pointerA == null) ? headB : pointerA.next;
```

```
        pointerB = (pointerB == null) ? headA : pointerB.next;
```

```
}
```

```
    // 返回相交节点或 null
```

```
    return pointerA;
```

```
}
```

```
/**
```

```

* 测试方法
*/
public static void test() {
    System.out.println("==== LeetCode 160. 相交链表测试 ===");

    // 测试用例 1: 相交链表
    System.out.println("测试用例 1: 相交链表");
    ListNode common = ListNode.createList(new int[]{8, 4, 5});
    ListNode listA = ListNode.createList(new int[]{4, 1});
    ListNode listB = ListNode.createList(new int[]{5, 6, 1});

    // 构造相交链表
    ListNode curA = listA;
    while (curA.next != null) {
        curA = curA.next;
    }
    curA.next = common;

    ListNode curB = listB;
    while (curB.next != null) {
        curB = curB.next;
    }
    curB.next = common;

    System.out.println("链表 A: 4 -> 1 -> 8 -> 4 -> 5");
    System.out.println("链表 B: 5 -> 6 -> 1 -> 8 -> 4 -> 5");
    ListNode intersection1 = getIntersectionNode(listA, listB);
    System.out.println("相交节点: " + (intersection1 != null ? intersection1.val : "null"));

    // 测试用例 2: 不相交链表
    System.out.println("测试用例 2: 不相交链表");
    ListNode listC = ListNode.createList(new int[]{1, 2, 3});
    ListNode listD = ListNode.createList(new int[]{4, 5, 6});
    System.out.println("链表 C: 1 -> 2 -> 3");
    System.out.println("链表 D: 4 -> 5 -> 6");
    ListNode intersection2 = getIntersectionNode(listC, listD);
    System.out.println("相交节点: " + (intersection2 != null ? intersection2.val : "null"));

    System.out.println("所有测试用例执行完成");
    System.out.println("=====");
}
}

```

```
/**  
 * 题目 13: LeetCode 206. 反转链表  
 * 来源: LeetCode  
 * 链接: https://leetcode.cn/problems/reverse-linked-list/  
 *  
 * 题目描述:  
 * 给你单链表的头节点 head，请你反转链表，并返回反转后的链表。  
 *  
 * 解法分析:  
 * 1. 迭代法 - 时间复杂度: O(n), 空间复杂度: O(1)  
 * 2. 递归法 - 时间复杂度: O(n), 空间复杂度: O(n)  
 *  
 * 解题思路:  
 * 迭代法: 使用三个指针分别指向前一个节点、当前节点和下一个节点，逐个反转节点的指向。  
 * 递归法: 递归到链表末尾，然后在回溯过程中反转节点的指向。  
 */  
  
class ReverseLinkedListSolution {  
    /**  
     * 解法 1: 迭代法 (推荐)  
     * 时间复杂度: O(n) - 需要遍历链表一次  
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间  
     *  
     * 核心思想:  
     * 1. 使用三个指针: prev(前一个节点)、current(当前节点)、next(下一个节点)  
     * 2. 逐个反转节点的指向  
     * 3. 移动指针继续处理下一个节点  
     */  
  
    public static ListNode reverseListIterative(ListNode head) {  
        // 初始化指针  
        ListNode prev = null;  
        ListNode current = head;  
  
        // 遍历链表  
        while (current != null) {  
            // 保存下一个节点  
            ListNode next = current.next;  
            // 反转当前节点的指向  
            current.next = prev;  
            // 移动指针  
            prev = current;  
            current = next;  
        }  
    }  
}
```

```

    // 返回新的头节点
    return prev;
}

/***
 * 解法 2: 递归法
 * 时间复杂度: O(n) - 需要遍历链表一次
 * 空间复杂度: O(n) - 递归调用栈的深度
 *
 * 核心思想:
 * 1. 递归到链表末尾
 * 2. 在回溯过程中反转节点的指向
 */
public static ListNode reverseListRecursive(ListNode head) {
    // 递归终止条件
    if (head == null || head.next == null) {
        return head;
    }

    // 递归处理下一个节点
    ListNode newHead = reverseListRecursive(head.next);
    // 反转当前节点和下一个节点的连接
    head.next.next = head;
    head.next = null;

    // 返回新的头节点
    return newHead;
}

/***
 * 测试方法
 */
public static void test() {
    System.out.println("==== LeetCode 206. 反转链表测试 ===");

    // 测试用例 1: 正常链表
    System.out.println("测试用例 1: 正常链表");
    ListNode list1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
    System.out.print("原链表: ");
    ListNode.printList(list1);
    ListNode reversed1 = reverseListIterative(list1);
    System.out.print("迭代法反转后: ");
    ListNode.printList(reversed1);
}

```

```

// 重新创建测试数据
ListNode list2 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
ListNode reversed2 = reverseListRecursive(list2);
System.out.print("递归法反转后: ");
ListNode.printList(reversed2);

// 测试用例 2: 单节点链表
System.out.println("测试用例 2: 单节点链表");
ListNode list3 = new ListNode(1);
System.out.print("原链表: ");
ListNode.printList(list3);
ListNode reversed3 = reverseListIterative(list3);
System.out.print("反转后: ");
ListNode.printList(reversed3);

// 测试用例 3: 空链表
System.out.println("测试用例 3: 空链表");
ListNode list4 = null;
System.out.print("原链表: ");
ListNode.printList(list4);
ListNode reversed4 = reverseListIterative(list4);
System.out.print("反转后: ");
ListNode.printList(reversed4);

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}

}

/***
 * 题目 14: LeetCode 234. 回文链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/palindrome-linked-list/
 *
 * 题目描述:
 * 给你一个单链表的头节点 head，请你判断该链表是否为回文链表。如果是，返回 true；否则，返回 false。
 *
 * 解法分析:
 * 1. 快慢指针 + 反转链表 - 时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 解题思路:

```

```

* 1. 使用快慢指针找到链表中点
* 2. 反转后半部分链表
* 3. 比较前半部分和反转后的后半部分
* 4. 恢复链表结构(可选)
*/
class PalindromeLinkedListSolution {
    /**
     * 解法: 快慢指针 + 反转链表
     * 时间复杂度: O(n) - 需要遍历链表多次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用快慢指针找到链表中点
     * 2. 反转后半部分链表
     * 3. 比较前半部分和反转后的后半部分
    */
    public static boolean isPalindrome(ListNode head) {
        // 边界条件检查
        if (head == null || head.next == null) {
            return true;
        }

        // 第一步: 使用快慢指针找到链表中点
        ListNode slow = head;
        ListNode fast = head;

        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // 第二步: 反转后半部分链表
        ListNode secondHalf = reverseList(slow.next);

        // 第三步: 比较前半部分和反转后的后半部分
        ListNode firstHalf = head;
        ListNode secondHalfCopy = secondHalf; // 保存用于恢复
        boolean isPalindrome = true;

        while (secondHalf != null) {
            if (firstHalf.val != secondHalf.val) {
                isPalindrome = false;
                break;
            }
            firstHalf = firstHalf.next;
            secondHalf = secondHalf.next;
        }

        // 恢复链表
        reverseList(secondHalfCopy);
        return isPalindrome;
    }

    private ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode prev = null;
        ListNode curr = head;
        ListNode next = curr.next;

        while (curr != null) {
            curr.next = prev;
            prev = curr;
            curr = next;
            if (next != null) {
                next = next.next;
            }
        }

        return prev;
    }
}

```

```

    }

    firstHalf = firstHalf.next;
    secondHalf = secondHalf.next;
}

// 第四步：恢复链表结构(可选)
slow.next = reverseList(secondHalfCopy);

return isPalindrome;
}

/***
 * 反转链表的辅助函数
 */
private static ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode current = head;

    while (current != null) {
        ListNode next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }

    return prev;
}

/***
 * 测试方法
 */
public static void test() {
    System.out.println("==== LeetCode 234. 回文链表测试 ===");

    // 测试用例 1：回文链表
    System.out.println("测试用例 1：回文链表");
    ListNode list1 = ListNode.createList(new int[] {1, 2, 2, 1});
    System.out.print("链表：");
    ListNode.printList(list1);
    System.out.println("是否为回文链表：" + isPalindrome(list1));

    // 测试用例 2：非回文链表
    System.out.println("测试用例 2：非回文链表");
}

```

```

ListNode list2 = ListNode.createList(new int[]{1, 2, 3, 4});
System.out.print("链表: ");
ListNode.printList(list2);
System.out.println("是否为回文链表: " + isPalindrome(list2));

// 测试用例 3: 单节点链表
System.out.println("测试用例 3: 单节点链表");
ListNode list3 = new ListNode(1);
System.out.print("链表: ");
ListNode.printList(list3);
System.out.println("是否为回文链表: " + isPalindrome(list3));

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}

}

/***
 * 题目 15: LeetCode 19. 删除链表的倒数第 N 个结点
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
 *
 * 题目描述:
 * 给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。
 *
 * 解法分析:
 * 1. 快慢指针法 - 时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 解题思路:
 * 使用两个指针，快指针先移动 n+1 步，然后快慢指针同时移动，
 * 当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点。
 */
class RemoveNthNodeFromEndOfListSolution {

    /**
     * 解法: 快慢指针法
     * 时间复杂度: O(n) - 需要遍历链表一次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用哨兵节点简化边界处理
     * 2. 快指针先移动 n+1 步
     * 3. 快慢指针同时移动
     * 4. 当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点
}

```

```
/*
public static ListNode removeNthFromEnd(ListNode head, int n) {
    // 创建哨兵节点，简化边界处理
    ListNode dummy = new ListNode(0);
    dummy.next = head;

    // 初始化快慢指针
    ListNode fast = dummy;
    ListNode slow = dummy;

    // 快指针先移动 n+1 步
    for (int i = 0; i <= n; i++) {
        fast = fast.next;
    }

    // 快慢指针同时移动
    while (fast != null) {
        fast = fast.next;
        slow = slow.next;
    }

    // 删除倒数第 n 个节点
    slow.next = slow.next.next;

    // 返回头节点
    return dummy.next;
}

/**
 * 测试方法
 */
public static void test() {
    System.out.println("==== LeetCode 19. 删除链表的倒数第 N 个结点测试 ===");

    // 测试用例 1: 删除中间节点
    System.out.println("测试用例 1: 删除中间节点");
    ListNode list1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
    System.out.print("原链表: ");
    ListNode.printList(list1);
    ListNode result1 = removeNthFromEnd(list1, 2);
    System.out.print("删除倒数第 2 个节点后: ");
    ListNode.printList(result1);
}
```

```

// 测试用例 2: 删除头节点
System.out.println("测试用例 2: 删除头节点");
ListNode list2 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
System.out.print("原链表: ");
ListNode.printList(list2);
ListNode result2 = removeNthFromEnd(list2, 5);
System.out.print("删除倒数第 5 个节点后: ");
ListNode.printList(result2);

// 测试用例 3: 删除尾节点
System.out.println("测试用例 3: 删除尾节点");
ListNode list3 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
System.out.print("原链表: ");
ListNode.printList(list3);
ListNode result3 = removeNthFromEnd(list3, 1);
System.out.print("删除倒数第 1 个节点后: ");
ListNode.printList(result3);

System.out.println("所有测试用例执行完成");
System.out.println("=====");
}

}

/**
 * 算法总结与技巧提升
 */
class AlgorithmSummary {
    public static void printSummary() {
        System.out.println("===== 链表合并算法总结 =====");
        System.out.println("1. 核心算法技巧:");
        System.out.println("    - 双指针法: 适用于两个有序序列的合并, 时间复杂度 O(m+n)");
        System.out.println("    - 优先队列法: 适用于 K 个有序序列的合并, 时间复杂度 O(N*logK)");
        System.out.println("    - 分治法: 适用于 K 个序列的归并, 时间复杂度 O(N*logK)");
        System.out.println("    - 哨兵节点: 简化链表操作的边界处理, 提高代码可读性");
        System.out.println("    - 原地修改: 避免额外空间开销, 适用于数组合并等场景");
        System.out.println();
        System.out.println("2. 工程化考量:");
        System.out.println("    - 异常处理: 处理空链表、单节点链表等边界情况");
        System.out.println("    - 内存管理: 在 Java 中由垃圾回收机制自动管理");
        System.out.println("    - 性能优化: 对于大规模数据, 优先队列的常数项优化很重要");
        System.out.println("    - 线程安全: 在多线程环境下需要考虑同步问题");
        System.out.println();
        System.out.println("3. 调试技巧:");
    }
}

```

```
System.out.println(" - 打印中间状态：使用 System.out 跟踪指针移动");
System.out.println(" - 边界测试：测试空输入、单元素输入、极端值等情况");
System.out.println(" - 断言验证：使用 assert 验证关键条件是否满足");
System.out.println();
System.out.println("4. 拓展应用:");
System.out.println(" - 归并排序：链表排序的最佳选择之一");
System.out.println(" - 多路归并：外部排序的基础算法");
System.out.println(" - 数据流处理：实时合并多个有序数据流");
System.out.println("=====");
System.out.println();
}

}

/***
 * 综合测试函数
 */
public class MergeTwoLists {
    public static void main(String[] args) {
        // 运行所有测试
        MergeTwoSortedListsSolution.test();
        // 移除递归法性能测试避免栈溢出
        // MergeTwoSortedListsSolution.performanceTest();

        MergeKSortedListsSolution.test();
        MergeKSortedListsSolution.performanceTest();

        // 新增题目的测试
        LinkedListCycleSolution.test();
        LinkedListCycleIISolution.test();
        IntersectionOfTwoLinkedListsSolution.test();
        ReverseLinkedListSolution.test();
        PalindromeLinkedListSolution.test();
        RemoveNthNodeFromEndOfListSolution.test();

        AlgorithmSummary.printSummary();
    }
}
```

文件: MergeTwoLists.py

```
# -*- coding: utf-8 -*-
```

"""

链表合并算法专题 - 完整实现 (Python 版本)

- 📚 本文件包含链表合并相关的完整算法实现，涵盖从基础到高级的各类题目
- 🎯 每个算法都提供详细的时间空间复杂度分析、多种解法对比和工程化考量

🔥 核心特性：

- 完整的异常处理和边界条件处理
- 详细的时间空间复杂度分析
- 多种解法对比（迭代法、递归法、分治法、优先队列法）
- 工程化考量和调试技巧
- 全面的测试用例覆盖

📊 算法复杂度总结：

| 算法 | 时间复杂度 | 空间复杂度 | 最优解 |
|----------------|---------------|-------------|-----|
| 合并两个链表(迭代) | $O(m+n)$ | $O(1)$ | ✓ |
| 合并两个链表(递归) | $O(m+n)$ | $O(m+n)$ | ✗ |
| 合并 K 个链表(优先队列) | $O(N \log K)$ | $O(K)$ | ✓ |
| 合并 K 个链表(分治) | $O(N \log K)$ | $O(\log K)$ | ✓ |
| 排序链表(自底向上) | $O(n \log n)$ | $O(1)$ | ✓ |

🎯 适用场景分析：

1. 双指针法：两个有序序列合并的基础算法
2. 优先队列法：K 个有序序列合并的高效算法
3. 分治法：大规模数据合并的优化策略
4. 原地合并：空间优化的合并技术

🏗 工程化考量：

- 异常处理：完善的输入验证和边界条件处理
- 内存管理：Python 自动垃圾回收，无需手动管理
- 性能优化：选择合适的数据结构和算法
- 可测试性：全面的单元测试覆盖
- 可读性：清晰的代码结构和注释

@author Algorithm Specialist

@version 1.0

@since 2025-10-18

"""

```
import heapq
import time
import random
```

```
from typing import List, Optional, Tuple
```

```
class ListNode:
```

```
    """
```

链表节点定义类

💡 设计要点:

- 使用属性简化访问（Python 风格）
- 提供静态工具方法便于测试
- 包含类型注解提高代码可读性

⚠ 注意事项:

- Python 中无需手动内存管理
- 注意循环引用的垃圾回收
- 考虑线程安全性需求

```
"""
```

```
def __init__(self, val: int = 0, next: Optional['ListNode'] = None):  
    self.val = val  
    self.next = next
```

```
@staticmethod
```

```
def create_list(arr: List[int]) -> Optional['ListNode']:  
    """
```

从数组创建链表（测试工具方法）

📊 复杂度分析:

- 时间复杂度: $O(n)$, n 为数组长度
- 空间复杂度: $O(n)$, 需要创建 n 个节点

🎯 使用场景: 单元测试、算法演示

```
@param arr 整数数组
```

```
@return 链表头节点
```

```
@raises ValueError 如果输入数组为 None
```

```
"""
```

```
if arr is None:  
    raise ValueError("输入数组不能为 None")  
if not arr:  
    return None
```

```
head = ListNode(arr[0])
```

```
cur = head
```

```
for i in range(1, len(arr)):
    cur.next = ListNode(arr[i])
    cur = cur.next
return head

@staticmethod
def print_list(head: Optional[ListNode]) -> None:
    """
    打印链表内容（调试工具方法）

```

 复杂度分析:

- 时间复杂度: $O(n)$, n 为链表长度
- 空间复杂度: $O(1)$, 只使用常数空间

 使用场景: 调试、结果验证

```
@param head 链表头节点
"""
if head is None:
    print("None")
    return

cur = head
while cur:
    print(cur.val, end="")
    if cur.next:
        print(" -> ", end="")
    cur = cur.next
print()

@staticmethod
def get_length(head: Optional[ListNode]) -> int:
    """
    获取链表长度（工具方法）

```

 复杂度分析:

- 时间复杂度: $O(n)$, 需要遍历整个链表
- 空间复杂度: $O(1)$, 只使用常数空间

```
@param head 链表头节点
@return 链表长度
"""
length = 0
```

```

cur = head
while cur:
    length += 1
    cur = cur.next
return length

@staticmethod
def is_sorted(head: Optional[ListNode]) -> bool:
    """
    验证链表是否有序（测试工具方法）

```

复杂度分析:

- 时间复杂度: $O(n)$, 需要遍历整个链表
- 空间复杂度: $O(1)$, 只使用常数空间

```

@param head 链表头节点
@return 是否有序（升序）
"""
if head is None or head.next is None:
    return True

```

```

cur = head
while cur.next:
    if cur.val > cur.next.val:
        return False
    cur = cur.next
return True

```

```
class MergeTwoSortedListsSolution:
```

```
"""

```

题目 1: LeetCode 21. 合并两个有序链表

题目信息:

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/merge-two-sorted-lists/>
- 难度: 简单
- 标签: 链表、递归、双指针

题目描述:

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

解题思路:

使用双指针分别指向两个链表的当前节点，比较节点值的大小，
将较小的节点连接到结果链表中，移动对应指针，重复此过程直到某一链表遍历完。
最后将未遍历完的链表剩余部分直接连接到结果链表末尾。

复杂度分析：

| 解法 | 时间复杂度 | 空间复杂度 | 最优解 |
|-----|----------|----------|-----|
| 迭代法 | $O(m+n)$ | $O(1)$ | ✓ |
| 递归法 | $O(m+n)$ | $O(m+n)$ | ✗ |

工程化考量：

- 使用哨兵节点简化边界处理
- 完善的异常处理和输入验证
- 考虑内存管理和性能优化

调试技巧：

- 打印中间状态跟踪指针移动
- 验证合并后链表的有序性
- 测试各种边界条件

"""

@staticmethod

```
def merge_two_lists_iterative(list1: Optional[ListNode], list2: Optional[ListNode]) ->
Optional[ListNode]:
```

"""

解法 1：迭代法（最优解）

核心思想：

1. 使用哨兵节点简化边界处理
2. 双指针分别遍历两个链表
3. 比较节点值，将较小节点连接到结果链表
4. 处理剩余节点

复杂度分析：

- 时间复杂度： $O(m+n)$ – 每个节点只访问一次
- 空间复杂度： $O(1)$ – 只使用常数级别的额外空间

性能特点：

- 最优时间复杂度
- 最优空间复杂度
- 适合大规模数据

工程实现要点：

- 哨兵节点避免空指针异常
- 清晰的变量命名提高可读性
- 完善的边界条件处理

```

@param list1 第一个有序链表
@param list2 第二个有序链表
@return 合并后的有序链表
"""

# 输入验证
if list1 is None and list2 is None:
    return None

# 创建哨兵节点，简化边界处理
dummy = ListNode(-1)
current = dummy

# 双指针遍历两个链表
while list1 and list2:
    # 比较两个链表当前节点的值
    if list1.val <= list2.val:
        current.next = list1
        list1 = list1.next
    else:
        current.next = list2
        list2 = list2.next
    current = current.next

# 连接剩余节点（其中一个链表已遍历完）
current.next = list1 if list1 else list2

# 返回合并后的链表（跳过哨兵节点）
return dummy.next

@staticmethod
def merge_two_lists_recursive(list1: Optional[ListNode], list2: Optional[ListNode]) ->
Optional[ListNode]:
"""
解法 2：递归法

```

⌚ 核心思想：

1. 递归终止条件：其中一个链表为空
2. 递归处理：选择较小节点作为当前节点，递归处理剩余部分
3. 返回当前节点

复杂度分析:

- 时间复杂度: $O(m+n)$ – 每个节点访问一次
- 空间复杂度: $O(m+n)$ – 递归调用栈的深度

性能特点:

- 代码简洁易懂
- 空间开销较大
- 可能栈溢出（大数据量）

适用场景:

- 小规模数据
- 代码简洁性要求高
- 栈深度可控的情况

```
@param list1 第一个有序链表
@param list2 第二个有序链表
@return 合并后的有序链表
"""

# 递归终止条件
if list1 is None:
    return list2
if list2 is None:
    return list1

# 递归处理: 选择较小节点作为当前节点
if list1.val <= list2.val:
    list1.next = MergeTwoSortedListsSolution.merge_two_lists_recursive(list1.next, list2)
    return list1
else:
    list2.next = MergeTwoSortedListsSolution.merge_two_lists_recursive(list1, list2.next)
    return list2

@staticmethod
def merge_two_lists_in_place(list1: Optional[ListNode], list2: Optional[ListNode]) ->
Optional[ListNode]:
"""

解法 3: 原地修改法（空间最优）
```

核心思想: 在不创建新节点的情况下直接修改链表连接

复杂度分析:

- 时间复杂度: $O(m+n)$

- 空间复杂度: $O(1)$

⚡ 性能特点:

- 最优空间复杂度
- 直接修改原链表
- 可能破坏原链表结构

```
@param list1 第一个有序链表
@param list2 第二个有序链表
@return 合并后的有序链表
"""

if list1 is None:
    return list2
if list2 is None:
    return list1

# 确保 list1 的头节点值较小
if list1.val > list2.val:
    list1, list2 = list2, list1

head = list1
while list1 and list2:
    prev = None
    while list1 and list1.val <= list2.val:
        prev = list1
        list1 = list1.next
    if prev:
        prev.next = list2

    # 交换 list1 和 list2
    if list1:
        list1, list2 = list2, list1

return head

@staticmethod
def test():
    """
全面的测试方法

```

🎯 测试策略:

1. 正常情况测试
2. 边界条件测试

3. 极端值测试

4. 性能测试

【 测试用例设计:

- 空链表测试
- 单节点链表测试
- 正常多节点测试
- 包含重复元素测试
- 极端值测试

"""

```
print("== LeetCode 21. 合并两个有序链表测试 ===")
print("【 测试用例覆盖: 正常情况、边界条件、极端值】")
```

测试用例 1: 正常情况

```
print("\n【 测试用例 1: 正常情况】")
list1 = ListNode.create_list([1, 2, 4])
list2 = ListNode.create_list([1, 3, 4])
print("链表 1: ", end="")
ListNode.print_list(list1)
print("链表 2: ", end="")
ListNode.print_list(list2)
```

```
result1 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list1, list2)
print("迭代法结果: ", end="")
ListNode.print_list(result1)
print("有序性验证: ", ListNode.is_sorted(result1))
```

重新创建测试数据

```
list1 = ListNode.create_list([1, 2, 4])
list2 = ListNode.create_list([1, 3, 4])
result2 = MergeTwoSortedListsSolution.merge_two_lists_recursive(list1, list2)
print("递归法结果: ", end="")
ListNode.print_list(result2)
print("有序性验证: ", ListNode.is_sorted(result2))
```

测试用例 2: 空链表

```
print("\n【 测试用例 2: 空链表测试】")
list3 = None
list4 = ListNode.create_list([0])
result3 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list3, list4)
print("空链表测试: ", end="")
ListNode.print_list(result3)
```

```

# 测试用例 3: 两个空链表
print("\n🔍 测试用例 3: 两个空链表")
list5 = None
list6 = None
result4 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list5, list6)
print("两个空链表: ", end="")
ListNode.print_list(result4)

# 测试用例 4: 包含重复元素
print("\n🔍 测试用例 4: 包含重复元素")
list7 = ListNode.create_list([1, 1, 2, 3])
list8 = ListNode.create_list([1, 2, 2, 4])
result5 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list7, list8)
print("包含重复元素结果: ", end="")
ListNode.print_list(result5)
print("有序性验证: ", ListNode.is_sorted(result5))

# 测试用例 5: 极端值测试
print("\n🔍 测试用例 5: 极端值测试")
list9 = ListNode.create_list([-10**6, 0, 10**6])
list10 = ListNode.create_list([-999999, 999999])
result6 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list9, list10)
print("极端值测试结果: ", end="")
ListNode.print_list(result6)
print("有序性验证: ", ListNode.is_sorted(result6))

print("\n✅ 所有测试用例执行完成")
print("=====")

```

```

@staticmethod
def performance_test():
    """
    性能测试方法

    ⚙️ 测试目的: 比较不同解法的性能表现
    📈 测试指标: 执行时间、内存使用
    """
    print("== 性能测试 ==")

    # 生成大规模测试数据
    size = 10000
    arr1 = [random.randint(0, 100000) for _ in range(size)]
    arr2 = [random.randint(0, 100000) for _ in range(size)]

```

```

arr1.sort()
arr2.sort()

list1 = ListNode.create_list(arr1)
list2 = ListNode.create_list(arr2)

# 测试迭代法性能
start_time = time.time()
result1 = MergeTwoSortedListsSolution.merge_two_lists_iterative(list1, list2)
end_time = time.time()
print(f"迭代法执行时间: {(end_time - start_time) * 1000:.3f} ms")

# 重新创建测试数据
list1 = ListNode.create_list(arr1)
list2 = ListNode.create_list(arr2)

# 测试递归法性能 (注意栈深度限制)
if size <= 1000: # 避免栈溢出
    start_time = time.time()
    result2 = MergeTwoSortedListsSolution.merge_two_lists_recursive(list1, list2)
    end_time = time.time()
    print(f"递归法执行时间: {(end_time - start_time) * 1000:.3f} ms")
else:
    print("递归法: 数据规模过大, 跳过测试 (避免栈溢出)")

print("性能测试完成\n")

```

class MergeKSortedListsSolution:

"""

题目 2: LeetCode 23. 合并 K 个升序链表

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-k-sorted-lists/>

题目描述:

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

解法分析:

1. 优先队列法 (最优解) - 时间复杂度: $O(N \log K)$, 空间复杂度: $O(K)$
2. 分治法 - 时间复杂度: $O(N \log K)$, 空间复杂度: $O(\log K)$

解题思路:

优先队列法：维护一个大小为 K 的最小堆，堆中存放 K 个链表的头节点。

每次从堆中取出最小节点加入结果链表，并将该节点的下一个节点加入堆中。

分治法：将 K 个链表分成两部分，分别合并后再合并两个结果。

”””

@staticmethod

```
def merge_k_lists_priority_queue(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
```

”””

解法 1：优先队列法（推荐）

🎯 核心思想：

1. 使用优先队列(最小堆)维护 K 个链表的当前最小节点
2. 每次取出最小节点加入结果链表
3. 将取出节点的下一个节点加入优先队列
4. 重复直到优先队列为空

📊 复杂度分析：

- 时间复杂度： $O(N \log K)$ – N 是所有节点总数，K 是链表数量
- 空间复杂度： $O(K)$ – 优先队列的大小

⚡ 性能特点：

- 最优时间复杂度
- 空间开销与 K 成正比
- 适合 K 较小的情况

💻 实现要点：

- 使用 heapq 实现最小堆
- 处理空链表边界条件
- 避免空指针异常

@param lists 链表数组

@return 合并后的有序链表

”””

```
if not lists:
```

```
    return None
```

```
# 创建优先队列(最小堆)，存储(节点值, 索引, 节点)元组
```

```
# 使用索引是为了处理节点值相同的情况
```

```
min_heap = []
```

```
# 将所有非空链表的头节点加入优先队列
```

```
for i, head in enumerate(lists):
```

```
    if head is not None:
```

```

    heapq.heappush(min_heap, (head.val, i, head))

# 创建哨兵节点
dummy = ListNode(-1)
current = dummy

# 从优先队列中依次取出最小节点
while min_heap:
    # 取出最小节点
    val, i, node = heapq.heappop(min_heap)
    # 加入结果链表
    current.next = node
    current = current.next

    # 如果该节点还有后续节点，加入优先队列
    if node.next is not None:
        heapq.heappush(min_heap, (node.next.val, i, node.next))

return dummy.next

```

```

@staticmethod
def merge_k_lists_helper(lists: List[Optional[ListNode]], left: int, right: int) ->
Optional[ListNode]:
    """
    分治辅助函数（递归实现）

```

⌚ 递归策略：

1. 基本情况：单个链表或两个链表
2. 递归情况：分割链表数组，递归合并

```

@param lists 链表数组
@param left 左边界
@param right 右边界
@return 合并后的有序链表
"""

# 递归终止条件
if left == right:
    return lists[left]
if left > right:
    return None

```

```

# 两个链表的情况直接合并
if left + 1 == right:

```

```

        return MergeTwoSortedListsSolution.merge_two_lists_iterative(lists[left],
lists[right])

# 分治: 将链表数组分成两部分
mid = left + (right - left) // 2
l1 = MergeKSortedListsSolution.merge_k_lists_helper(lists, left, mid)
l2 = MergeKSortedListsSolution.merge_k_lists_helper(lists, mid + 1, right)

# 合并两个结果
return MergeTwoSortedListsSolution.merge_two_lists_iterative(l1, l2)

@staticmethod
def merge_k_lists_divide_and_conquer(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
    """
    解法 2: 分治法
    时间复杂度: O(N*logK) - N 是所有节点总数, K 是链表数量
    空间复杂度: O(logK) - 递归调用栈的深度

    核心思想:
    1. 将 K 个链表分成两部分
    2. 递归合并每一部分
    3. 合并两个结果链表
    """

    if not lists:
        return None
    return MergeKSortedListsSolution.merge_k_lists_helper(lists, 0, len(lists) - 1)

@staticmethod
def test():
    """
    测试方法
    """

    print("== 合并 K 个升序链表测试 ==")

    # 创建测试数据
    l1 = ListNode.create_list([1, 4, 5])
    l2 = ListNode.create_list([1, 3, 4])
    l3 = ListNode.create_list([2, 6])

    lists = [l1, l2, l3]

    print("链表 1: ", end="")
    ListNode.print_list(lists[0])

```

```

print("链表 2: ", end="")
ListNode.print_list(lists[1])
print("链表 3: ", end="")
ListNode.print_list(lists[2])

# 测试优先队列法
result1 = MergeKSortedListsSolution.merge_k_lists_priority_queue(lists)
print("优先队列法结果: ", end="")
ListNode.print_list(result1)

# 重新创建测试数据
l1 = ListNode.create_list([1, 4, 5])
l2 = ListNode.create_list([1, 3, 4])
l3 = ListNode.create_list([2, 6])
lists = [l1, l2, l3]

# 测试分治法
result2 = MergeKSortedListsSolution.merge_k_lists_divide_and_conquer(lists)
print("分治法结果: ", end="")
ListNode.print_list(result2)
print()

```

class MergeSortedArraySolution:

"""

题目 3: LeetCode 88. 合并两个有序数组

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-sorted-array/>

题目描述:

给你两个按非递减顺序排列的整数数组 `nums1` 和 `nums2`, 另有两个整数 `m` 和 `n`, 分别表示 `nums1` 和 `nums2` 中的元素数目。

请你合并 `nums2` 到 `nums1` 中, 使合并后的数组同样按非递减顺序排列。

注意: 最终, 合并后数组不应由函数返回, 而是存储在数组 `nums1` 中。

为了应对这种情况, `nums1` 的初始长度为 `m + n`, 其中前 `m` 个元素表示应合并的元素, 后 `n` 个元素为 0, 应忽略。

`nums2` 的长度为 `n`。

解法分析:

1. 从后往前合并 (最优解) - 时间复杂度: $O(m+n)$, 空间复杂度: $O(1)$
2. 从前往后合并 - 时间复杂度: $O(m+n)$, 空间复杂度: $O(m+n)$
3. 合并后排序 - 时间复杂度: $O((m+n) \log(m+n))$, 空间复杂度: $O(1)$

解题思路：

从后往前合并可以避免覆盖 `nums1` 中未处理的元素。

使用三个指针分别指向 `nums1` 有效元素末尾、`nums2` 末尾和 `nums1` 实际末尾。

比较两个数组当前元素，将较大者放入 `nums1` 末尾，移动相应指针。

"""

```
@staticmethod
```

```
def merge_from_back(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
```

"""

解法 1：从后往前合并（推荐）

时间复杂度：O(m+n) – 每个元素访问一次

空间复杂度：O(1) – 原地修改

核心思想：

1. 从两个数组的末尾开始比较
2. 将较大元素放到 `nums1` 的末尾
3. 移动相应指针
4. 处理剩余元素

"""

```
# 三个指针
```

```
i = m - 1      # nums1 有效元素的末尾
```

```
j = n - 1      # nums2 的末尾
```

```
k = m + n - 1  # nums1 实际末尾
```

```
# 从后往前合并
```

```
while i >= 0 and j >= 0:
```

```
    if nums1[i] > nums2[j]:
```

```
        nums1[k] = nums1[i]
```

```
        i -= 1
```

```
    else:
```

```
        nums1[k] = nums2[j]
```

```
        j -= 1
```

```
    k -= 1
```

```
# 处理 nums2 剩余元素
```

```
while j >= 0:
```

```
    nums1[k] = nums2[j]
```

```
    j -= 1
```

```
    k -= 1
```

```
# 注意：如果 nums1 有剩余元素，它们已经在正确位置，无需处理
```

```
@staticmethod
```

```

def merge_from_front(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    """
    解法 2: 从前往后合并
    时间复杂度: O(m+n)
    空间复杂度: O(m) - 需要额外数组存储 nums1 的前 m 个元素
    """

    # 创建临时数组存储 nums1 的前 m 个元素
    nums1_copy = nums1[:m]

    # 三个指针
    i = 0  # nums1_copy 的指针
    j = 0  # nums2 的指针
    k = 0  # nums1 的指针

    # 从前往后合并
    while i < m and j < n:
        if nums1_copy[i] <= nums2[j]:
            nums1[k] = nums1_copy[i]
            i += 1
        else:
            nums1[k] = nums2[j]
            j += 1
        k += 1

    # 处理剩余元素
    while i < m:
        nums1[k] = nums1_copy[i]
        i += 1
        k += 1

    while j < n:
        nums1[k] = nums2[j]
        j += 1
        k += 1

    @staticmethod
    def merge_and_sort(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        解法 3: 合并后排序
        时间复杂度: O((m+n) log (m+n))
        空间复杂度: O(1)
        """

        # 将 nums2 复制到 nums1 的后半部分

```

```
nums1[m:m+n] = nums2
# 排序
nums1.sort()

@staticmethod
def test():
    """
    测试方法
    """
    print("== 合并两个有序数组测试 ==")

    # 测试用例 1
    nums1 = [1, 2, 3, 0, 0, 0]
    m = 3
    nums2 = [2, 5, 6]
    n = 3

    print(f"数组 1: {nums1}, m = {m}")
    print(f"数组 2: {nums2}, n = {n}")

    # 测试从后往前合并
    nums1_copy1 = nums1.copy()
    MergeSortedArraySolution.merge_from_back(nums1_copy1, m, nums2, n)
    print(f"从后往前合并: {nums1_copy1}")

    # 测试从前往后合并
    nums1_copy2 = nums1.copy()
    MergeSortedArraySolution.merge_from_front(nums1_copy2, m, nums2, n)
    print(f"从前往后合并: {nums1_copy2}")

    # 测试合并后排序
    nums1_copy3 = nums1.copy()
    MergeSortedArraySolution.merge_and_sort(nums1_copy3, m, nums2, n)
    print(f"合并后排序: {nums1_copy3}")

    # 测试用例 2: nums1 为空
    nums3 = [0]
    m2 = 0
    nums4 = [1]
    n2 = 1

    print(f"\n数组 1: {nums3}, m = {m2}")
    print(f"数组 2: {nums4}, n = {n2}")
```

```
MergeSortedArraySolution.merge_from_back(nums3, m2, nums4, n2)
print(f"从后往前合并: {nums3}")
print()
```

```
class SortListSolution:
```

```
    """
```

题目 4: LeetCode 148. 排序链表

来源: LeetCode

链接: <https://leetcode.cn/problems/sort-list/>

题目描述:

给你链表的头结点 head，请将其按 升序 排列并返回 排序后的链表。

要求在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序。

```
"""
```

```
@staticmethod
```

```
def sort_list_top_down(head: Optional[ListNode]) -> Optional[ListNode]:
```

```
    """
```

解法 1: 归并排序 (自顶向下)

时间复杂度: $O(n \log n)$ – 归并排序的标准时间复杂度

空间复杂度: $O(\log n)$ – 递归调用栈的深度

```
"""
```

基本情况: 空链表或只有一个节点

```
if not head or not head.next:
```

```
    return head
```

使用快慢指针找到中点

```
slow, fast = head, head.next
```

```
while fast and fast.next:
```

```
    slow = slow.next
```

```
    fast = fast.next.next
```

分割链表

```
mid = slow.next
```

```
slow.next = None
```

递归排序两个子链表

```
left = SortListSolution.sort_list_top_down(head)
```

```
right = SortListSolution.sort_list_top_down(mid)
```

合并排序后的链表

```
return SortListSolution.merge_two_lists(left, right)

@staticmethod
def sort_list_bottom_up(head: Optional[ListNode]) -> Optional[ListNode]:
    """
```

解法 2：归并排序（自底向上） - 最优解

⌚ 核心思想：

1. 从步长 1 开始，每次翻倍
2. 按当前步长将链表分割成多个子链表并合并
3. 重复直到步长大于等于链表长度

📊 复杂度分析：

- 时间复杂度： $O(n \log n)$ - 与自顶向下相同
- 空间复杂度： $O(1)$ - 只使用常数级额外空间

⚡ 性能特点：

- 避免递归栈溢出
- 空间效率高
- 实现相对复杂

```
@param head 链表头节点
@return 排序后的链表
"""
if head is None or head.next is None:
    return head

# 计算链表长度
length = 0
current = head
while current:
    length += 1
    current = current.next

# 创建哨兵节点
dummy = ListNode(-1)
dummy.next = head

# 自底向上进行归并
step = 1
while step < length:
    prev = dummy
    current = dummy.next
```

```
while current:
    # 第一个子链表的头节点
    left = current
    # 分割第一个子链表
    for i in range(1, step):
        if current.next is not None:
            current = current.next
        else:
            break

    # 第二个子链表的头节点
    right = current.next
    # 断开第一个子链表
    current.next = None
    current = right

    # 分割第二个子链表
    for i in range(1, step):
        if current is not None and current.next is not None:
            current = current.next
        else:
            break

    # 记录下一段链表的起始位置
    next_node = None
    if current is not None:
        next_node = current.next
        current.next = None

    # 合并两个子链表
    prev.next = SortListSolution.merge_two_lists(left, right)

    # 移动 prev 到合并后链表的末尾
    while prev.next is not None:
        prev = prev.next

    # 处理下一段链表
    current = next_node

    step *= 2

return dummy.next
```

```
@staticmethod
def merge_two_lists(l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
    """
    合并两个有序链表的辅助函数
    """
    dummy = ListNode(-1)
    current = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 if l1 else l2
    return dummy.next
```

```
@staticmethod
def test():
    """
    测试方法
    """
    print("== 排序链表测试 ==")

    # 测试用例 1: 正常情况
    list1 = ListNode.create_list([4, 2, 1, 3])
    print("原链表: ", end="")
    ListNode.print_list(list1)

    result1 = SortListSolution.sort_list_top_down(list1)
    print("自顶向下归并排序结果: ", end="")
    ListNode.print_list(result1)

    # 重新创建测试数据
    list2 = ListNode.create_list([4, 2, 1, 3])
    result2 = SortListSolution.sort_list_bottom_up(list2)
    print("自底向上归并排序结果: ", end="")
    ListNode.print_list(result2)
```

```

# 测试用例 2: 包含重复元素
list3 = ListNode.create_list([-1, 5, 3, 4, 0])
print("\n原链表: ", end="")
ListNode.print_list(list3)

result3 = SortListSolution.sort_list_bottom_up(list3)
print("排序结果: ", end="")
ListNode.print_list(result3)
print()

```

```
class AddTwoNumbersSolution:
```

```
"""
题目 5: LeetCode 2. 两数相加
```

```
来源: LeetCode
```

```
链接: https://leetcode.cn/problems/add-two-numbers/
```

题目描述:

给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能存储一位数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

```
"""

```

```
@staticmethod
```

```
def add_two_numbers(l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
```

```
"""

```

解法: 模拟加法过程

时间复杂度: $O(\max(m, n))$ – m 和 n 分别是两个链表的长度

空间复杂度: $O(\max(m, n))$ – 输出链表的长度最多为 $\max(m, n)+1$

```
"""

```

创建哨兵节点

```
dummy = ListNode(-1)
```

```
current = dummy
```

进位

```
carry = 0
```

同时遍历两个链表

```
while l1 or l2 or carry > 0:
```

计算当前位的和

```
sum_val = carry
```

```
if l1:
```

```
    sum_val += l1.val
```

```
    11 = 11.next
    if 12:
        sum_val += 12.val
        12 = 12.next

        # 更新进位
        carry = sum_val // 10
        # 创建新节点存储当前位的结果
        current.next = ListNode(sum_val % 10)
        current = current.next

    return dummy.next

@staticmethod
def test():
    """
    测试方法
    """
    print("== 两数相加测试 ==")

    # 测试用例 1: 正常情况
    11 = ListNode.create_list([2, 4, 3])  # 342
    12 = ListNode.create_list([5, 6, 4])  # 465
    print("链表 1 (342 逆序): ", end="")
    ListNode.print_list(11)
    print("链表 2 (465 逆序): ", end="")
    ListNode.print_list(12)

    result1 = AddTwoNumbersSolution.add_two_numbers(11, 12)
    print("结果 (807 逆序): ", end="")
    ListNode.print_list(result1)

    # 测试用例 2: 包含进位
    13 = ListNode.create_list([9, 9, 9, 9, 9, 9, 9])
    14 = ListNode.create_list([9, 9, 9, 9])
    print("\n链表 1: ", end="")
    ListNode.print_list(13)
    print("链表 2: ", end="")
    ListNode.print_list(14)

    result2 = AddTwoNumbersSolution.add_two_numbers(13, 14)
    print("结果: ", end="")
    ListNode.print_list(result2)
```

```
print()

class SwapNodesInPairsSolution:
    """
    题目 6: LeetCode 24. 两两交换链表中的节点
    来源: LeetCode
    链接: https://leetcode.cn/problems/swap-nodes-in-pairs/

    题目描述:
    给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。
    你必须在不修改节点内部值的情况下完成本题（即，只能进行节点交换）。
    """

    @staticmethod
    def swap_pairs_iterative(head: Optional[ListNode]) -> Optional[ListNode]:
        """
        解法 1: 迭代法 (推荐)
        时间复杂度: O(n) - 每个节点只访问一次
        空间复杂度: O(1) - 只使用常数级额外空间
        """

        # 创建哨兵节点
        dummy = ListNode(-1)
        dummy.next = head

        prev = dummy

        # 确保有至少两个节点可以交换
        while prev.next and prev.next.next:
            # 标记需要交换的两个节点
            first = prev.next
            second = prev.next.next

            # 交换节点
            first.next = second.next
            second.next = first
            prev.next = second

            # 移动 prev 到下一对的前一个位置
            prev = first

        return dummy.next
```

```
@staticmethod
def swap_pairs_recursive(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    解法 2: 递归法
    时间复杂度: O(n) - 每个节点只访问一次
    空间复杂度: O(n) - 递归调用栈的深度
    """
    # 递归终止条件
    if not head or not head.next:
        return head

    # 标记需要交换的两个节点
    first = head
    second = head.next

    # 交换节点
    first.next = SwapNodesInPairsSolution.swap_pairs_recursive(second.next)
    second.next = first

    # 返回新的头节点
    return second
```

```
@staticmethod
def test():
    """
    测试方法
    """
    print("== 两两交换链表中的节点测试 ==")

    # 测试用例 1: 偶数个节点
    list1 = ListNode.create_list([1, 2, 3, 4])
    print("原链表: ", end="")
    ListNode.print_list(list1)

    result1 = SwapNodesInPairsSolution.swap_pairs_iterative(list1)
    print("迭代法结果: ", end="")
    ListNode.print_list(result1)

    # 重新创建测试数据
    list2 = ListNode.create_list([1, 2, 3, 4])
    result2 = SwapNodesInPairsSolution.swap_pairs_recursive(list2)
    print("递归法结果: ", end="")
    ListNode.print_list(result2)
```

```

# 测试用例 2: 奇数个节点
list3 = ListNode.create_list([1, 2, 3])
print("\n原链表: ", end="")
ListNode.print_list(list3)

result3 = SwapNodesInPairsSolution.swap_pairs_iterative(list3)
print("交换结果: ", end="")
ListNode.print_list(result3)
print()

```

class NowCoderMergeSortedListsSolution:

"""

题目 7: 牛客 NC33. 合并两个排序的链表

来源: 牛客网

链接: <https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337>

"""

@staticmethod

def merge(pHead1: Optional[ListNode], pHead2: Optional[ListNode]) -> Optional[ListNode]:

"""

合并两个排序的链表

"""

dummy = ListNode(-1)

current = dummy

while pHead1 and pHead2:

if pHead1.val <= pHead2.val:

current.next = pHead1

pHead1 = pHead1.next

else:

current.next = pHead2

pHead2 = pHead2.next

current = current.next

current.next = pHead1 if pHead1 else pHead2

return dummy.next

@staticmethod

def test():

"""

测试方法

```

"""
print("== 牛客 NC33. 合并两个排序的链表测试 ==")

list1 = ListNode.create_list([1, 3, 5])
list2 = ListNode.create_list([2, 4, 6])
print("链表 1: ", end="")
ListNode.print_list(list1)
print("链表 2: ", end="")
ListNode.print_list(list2)

result = NowCoderMergeSortedListsSolution.merge(list1, list2)
print("合并结果: ", end="")
ListNode.print_list(result)
print()

```

class LintCodeMergeKListsSolution:

"""

题目 8: LintCode 104. 合并 k 个排序链表

来源: LintCode

链接: <https://www.lintcode.com/problem/104/>

"""

@staticmethod

def merge_k_lists(lists: List[Optional[ListNode]]) -> Optional[ListNode]:

"""

使用优先队列合并 k 个排序链表

"""

优先队列的自定义比较需要用到一个包装类

或者使用元组(值, 索引, 节点)来避免比较节点对象

dummy = ListNode(-1)

current = dummy

优先队列

heap = []

将所有非空链表的头节点加入优先队列

for i, node in enumerate(lists):

if node:

使用值和索引作为比较的键, 避免节点比较

heapq.heappush(heap, (node.val, i, node))

从优先队列中依次取出最小节点

```

while heap:
    val, i, node = heapq.heappop(heap)
    current.next = node
    current = current.next

if node.next:
    heapq.heappush(heap, (node.next.val, i, node.next))

return dummy.next

@staticmethod
def test():
    """
    测试方法
    """
    print("== LintCode 104. 合并 k 个排序链表测试 ==")

    l1 = ListNode.create_list([2, 4])
    l2 = ListNode.create_list([1, 3, 5])
    l3 = ListNode.create_list([6, 7])
    lists = [l1, l2, l3]

    print("链表 1: ", end="")
    ListNode.print_list(lists[0])
    print("链表 2: ", end="")
    ListNode.print_list(lists[1])
    print("链表 3: ", end="")
    ListNode.print_list(lists[2])

    result = LintCodeMergeKListsSolution.merge_k_lists(lists)
    print("合并结果: ", end="")
    ListNode.print_list(result)
    print()

```

```

class PartitionListSolution:
    """
    题目 9: LeetCode 86. 分隔链表
    来源: LeetCode
    链接: https://leetcode.cn/problems/partition-list/
    """

    @staticmethod

```

```

def partition(head: Optional[ListNode], x: int) -> Optional[ListNode]:
    less_head = ListNode(-1)
    greater_head = ListNode(-1)
    less = less_head
    greater = greater_head

    while head:
        if head.val < x:
            less.next = head
            less = less.next
        else:
            greater.next = head
            greater = greater.next
        head = head.next

    greater.next = None
    less.next = greater_head.next

    return less_head.next

```

```

@staticmethod
def test():
    print("== LeetCode 86. 分隔链表测试 ==")
    list_node = ListNode.create_list([1, 4, 3, 2, 5, 2])
    print("原链表: ", end="")
    ListNode.print_list(list_node)
    result = PartitionListSolution.partition(list_node, 3)
    print("分隔后(x=3): ", end="")
    ListNode.print_list(result)
    print()

```

```

class LinkedListCycleSolution:
    """

```

题目 10: LeetCode 141. 环形链表

来源: LeetCode

链接: <https://leetcode.cn/problems/linked-list-cycle/>

题目描述:

给你一个链表的头节点 head，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。

解法分析:

1. 快慢指针法 (Floyd 判圈算法) - 时间复杂度: $O(n)$, 空间复杂度: $O(1)$

解题思路:

使用两个指针, 一个快指针和一个慢指针。快指针每次移动两步, 慢指针每次移动一步。

如果链表中存在环, 快指针最终会追上慢指针; 如果不存在环, 快指针会先到达链表末尾。

"""

```
@staticmethod
def has_cycle(head: Optional[ListNode]) -> bool:
    """
```

解法: 快慢指针法 (Floyd 判圈算法)

⌚ 核心思想:

1. 初始化快慢指针都指向头节点
2. 快指针每次移动两步, 慢指针每次移动一步
3. 如果存在环, 快指针会追上慢指针
4. 如果不存在环, 快指针会先到达链表末尾

📊 复杂度分析:

- 时间复杂度: $O(n)$ - 最多遍历链表两次
- 空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

⚡ 性能特点:

- 最优时间复杂度
- 最优空间复杂度
- 适合大规模数据

```
@param head 链表头节点
@return 是否存在环
"""
# 边界条件检查
if head is None or head.next is None:
    return False

# 初始化快慢指针
slow = head
fast = head

# 移动指针
while fast is not None and fast.next is not None:
    slow = slow.next      # 慢指针每次移动一步
    fast = fast.next.next # 快指针每次移动两步
```

```
# 如果快慢指针相遇，说明存在环
if slow == fast:
    return True

# 如果快指针到达链表末尾，说明不存在环
return False

@staticmethod
def test():
    """
    测试方法
    """
    print("== LeetCode 141. 环形链表测试 ==")

    # 测试用例 1：无环链表
    print("测试用例 1：无环链表")
    list1 = ListNode.create_list([1, 2, 3, 4])
    print("链表：", end="")
    ListNode.print_list(list1)
    print(f"是否有环： {LinkedListCycleSolution.has_cycle(list1)}")

    # 测试用例 2：有环链表（构造环）
    print("测试用例 2：有环链表")
    list2 = ListNode.create_list([1, 2, 3, 4])
    # 构造环：将尾节点指向第二个节点
    cur = list2
    while cur.next is not None:
        cur = cur.next
    cur.next = list2.next # 尾节点指向第二个节点
    print("链表：1 -> 2 -> 3 -> 4 -> 2 (形成环)")
    print(f"是否有环： {LinkedListCycleSolution.has_cycle(list2)}")

    # 测试用例 3：单节点无环
    print("测试用例 3：单节点无环")
    list3 = ListNode(1)
    print("链表：1")
    print(f"是否有环： {LinkedListCycleSolution.has_cycle(list3)}")

    # 测试用例 4：空链表
    print("测试用例 4：空链表")
    list4 = None
    print("链表：None")
    print(f"是否有环： {LinkedListCycleSolution.has_cycle(list4)}")
```

```
print("所有测试用例执行完成")
print("====")
```

```
class LinkedListCycleIISolution:
```

```
"""
```

题目 11: LeetCode 142. 环形链表 II

来源: LeetCode

链接: <https://leetcode.cn/problems/linked-list-cycle-ii/>

题目描述:

给定一个链表的头节点 head , 返回链表开始入环的第一个节点。 如果链表无环, 则返回 null 。

解法分析:

1. 快慢指针法 – 时间复杂度: $O(n)$, 空间复杂度: $O(1)$

解题思路:

使用快慢指针找到环后, 将快指针重新指向头节点, 然后快慢指针都每次移动一步, 当它们再次相遇时, 相遇点就是环的入口节点。

```
"""
```

```
@staticmethod
def detect_cycle(head: Optional[ListNode]) -> Optional[ListNode]:
    """
```

解法: 快慢指针法

🎯 核心思想:

1. 使用快慢指针找到环
2. 将快指针重新指向头节点
3. 快慢指针都每次移动一步
4. 再次相遇点就是环的入口

📊 复杂度分析:

- 时间复杂度: $O(n)$ – 最多遍历链表三次
- 空间复杂度: $O(1)$ – 只使用了常数级别的额外空间

@param head 链表头节点

@return 环的入口节点, 如果无环则返回 None

```
"""
```

边界条件检查

```
if head is None or head.next is None:
    return None
```

```
# 第一阶段：使用快慢指针判断是否有环
slow = head
fast = head

while fast is not None and fast.next is not None:
    slow = slow.next
    fast = fast.next.next

    # 如果快慢指针相遇，说明存在环
    if slow == fast:
        break

# 如果没有环，返回 None
if fast is None or fast.next is None:
    return None

# 第二阶段：找到环的入口
# 将快指针重新指向头节点
fast = head
# 快慢指针都每次移动一步，直到相遇
while slow != fast:
    slow = slow.next
    fast = fast.next

# 相遇点就是环的入口
return slow

@staticmethod
def test():
    """
    测试方法
    """
    print("== LeetCode 142. 环形链表 II 测试 ==")

    # 测试用例 1：无环链表
    print("测试用例 1：无环链表")
    list1 = ListNode.create_list([1, 2, 3, 4])
    print("链表：", end="")
    ListNode.print_list(list1)
    cycle_start1 = LinkedListCycleIISolution.detect_cycle(list1)
    print(f"环的入口：{cycle_start1.val if cycle_start1 else 'null'}")
```

```

# 测试用例 2: 有环链表 (构造环)
print("测试用例 2: 有环链表")
list2 = ListNode.create_list([1, 2, 3, 4])
# 构造环: 将尾节点指向第二个节点
cur = list2
while cur.next is not None:
    cur = cur.next
cur.next = list2.next # 尾节点指向第二个节点
print("链表: 1 -> 2 -> 3 -> 4 -> 2 (形成环)")
cycle_start2 = LinkedListCycleIISolution.detect_cycle(list2)
print(f"环的入口: {cycle_start2.val if cycle_start2 else 'null'}")

print("所有测试用例执行完成")
print("=====")

```

class IntersectionOfTwoLinkedListsSolution:

"""

题目 12: LeetCode 160. 相交链表

来源: LeetCode

链接: <https://leetcode.cn/problems/intersection-of-two-linked-lists/>

题目描述:

给你两个单链表的头节点 headA 和 headB , 请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null 。

解法分析:

1. 双指针法 – 时间复杂度: $O(m+n)$, 空间复杂度: $O(1)$

解题思路:

使用两个指针分别遍历两个链表，当一个指针到达链表末尾时，将其指向另一个链表的头节点。

如果两个链表相交，两个指针会在相交节点相遇；如果不相交，两个指针会同时到达链表末尾。

"""

```

@staticmethod
def get_intersection_node(headA: Optional[ListNode], headB: Optional[ListNode]) ->
Optional[ListNode]:
    """

```

解法: 双指针法

核心思想:

1. 使用两个指针分别遍历两个链表
2. 当指针到达链表末尾时，将其指向另一个链表的头节点

3. 如果两个链表相交，两个指针会在相交节点相遇
4. 如果不相交，两个指针会同时到达链表末尾

复杂度分析:

- 时间复杂度: $O(m+n)$ – 最多遍历两个链表各两次
- 空间复杂度: $O(1)$ – 只使用了常数级别的额外空间

```
@param headA 链表 A 的头节点
@param headB 链表 B 的头节点
@return 相交节点, 如果不相交则返回 None
"""

# 边界条件检查
if headA is None or headB is None:
    return None

# 初始化两个指针
pointer_a = headA
pointer_b = headB

# 当两个指针不相等时继续遍历
while pointer_a != pointer_b:
    # 当指针到达链表末尾时, 将其指向另一个链表的头节点
    pointer_a = headB if pointer_a is None else pointer_a.next
    pointer_b = headA if pointer_b is None else pointer_b.next

# 返回相交节点或 None
return pointer_a

@staticmethod
def test():
    """
    测试方法
    """

    print("== LeetCode 160. 相交链表测试 ==")

    # 测试用例 1: 相交链表
    print("测试用例 1: 相交链表")
    common = ListNode.create_list([8, 4, 5])
    list_a = ListNode.create_list([4, 1])
    list_b = ListNode.create_list([5, 6, 1])

    # 构造相交链表
    cur_a = list_a
    cur_b = list_b
    while cur_a != cur_b:
        cur_a = common
        cur_b = common
    cur_a.next = cur_b
```

```

while cur_a.next is not None:
    cur_a = cur_a.next
cur_a.next = common

cur_b = list_b
while cur_b.next is not None:
    cur_b = cur_b.next
cur_b.next = common

print("链表 A: 4 -> 1 -> 8 -> 4 -> 5")
print("链表 B: 5 -> 6 -> 1 -> 8 -> 4 -> 5")
intersection1 = IntersectionOfTwoLinkedListsSolution.get_intersection_node(list_a,
list_b)
print(f"相交节点: {intersection1.val if intersection1 else 'null'}")

# 测试用例 2: 不相交链表
print("测试用例 2: 不相交链表")
list_c = ListNode.create_list([1, 2, 3])
list_d = ListNode.create_list([4, 5, 6])
print("链表 C: 1 -> 2 -> 3")
print("链表 D: 4 -> 5 -> 6")
intersection2 = IntersectionOfTwoLinkedListsSolution.get_intersection_node(list_c,
list_d)
print(f"相交节点: {intersection2.val if intersection2 else 'null'}")

print("所有测试用例执行完成")
print("=====")

```

class ReverseLinkedListSolution:

"""

题目 13: LeetCode 206. 反转链表

来源: LeetCode

链接: <https://leetcode.cn/problems/reverse-linked-list/>

题目描述:

给你单链表的头节点 head，请你反转链表，并返回反转后的链表。

解法分析:

1. 迭代法 - 时间复杂度: $O(n)$, 空间复杂度: $O(1)$
2. 递归法 - 时间复杂度: $O(n)$, 空间复杂度: $O(n)$

解题思路:

迭代法：使用三个指针分别指向前一个节点、当前节点和下一个节点，逐个反转节点的指向。

递归法：递归到链表末尾，然后在回溯过程中反转节点的指向。

"""

```
@staticmethod
def reverse_list_iterative(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    解法 1：迭代法（推荐）
    
```

⌚ 核心思想：

1. 使用三个指针：prev(前一个节点)、current(当前节点)、next(下一个节点)
2. 逐个反转节点的指向
3. 移动指针继续处理下一个节点

📊 复杂度分析：

- 时间复杂度： $O(n)$ – 需要遍历链表一次
- 空间复杂度： $O(1)$ – 只使用了常数级别的额外空间

⚡ 性能特点：

- 最优时间复杂度
- 最优空间复杂度
- 适合大规模数据

```
@param head 链表头节点
@return 反转后的链表头节点
"""
# 初始化指针
prev = None
current = head

# 遍历链表
while current is not None:
    # 保存下一个节点
    next_node = current.next
    # 反转当前节点的指向
    current.next = prev
    # 移动指针
    prev = current
    current = next_node

# 返回新的头节点
return prev
```

```
@staticmethod
def reverse_list_recursive(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    解法 2：递归法
    
```

👉 核心思想：

1. 递归到链表末尾
2. 在回溯过程中反转节点的指向

📊 复杂度分析：

- 时间复杂度： $O(n)$ – 需要遍历链表一次
- 空间复杂度： $O(n)$ – 递归调用栈的深度

⚡ 性能特点：

- 代码简洁易懂
- 空间开销较大
- 可能栈溢出（大数据量）

```
@param head 链表头节点
@return 反转后的链表头节点
"""
# 递归终止条件
if head is None or head.next is None:
    return head

# 递归处理下一个节点
new_head = ReverseLinkedListSolution.reverse_list_recursive(head.next)
# 反转当前节点和下一个节点的连接
head.next.next = head
head.next = None

# 返回新的头节点
return new_head

```

```
@staticmethod
def test():
    """
    测试方法
    """
    print("==== LeetCode 206. 反转链表测试 ====")

    # 测试用例 1：正常链表
    print("测试用例 1：正常链表")

```

```

list1 = ListNode.create_list([1, 2, 3, 4, 5])
print("原链表: ", end="")
ListNode.print_list(list1)
reversed1 = ReverseLinkedListSolution.reverse_list_iterative(list1)
print("迭代法反转后: ", end="")
ListNode.print_list(reversed1)

# 重新创建测试数据
list2 = ListNode.create_list([1, 2, 3, 4, 5])
reversed2 = ReverseLinkedListSolution.reverse_list_recursive(list2)
print("递归法反转后: ", end="")
ListNode.print_list(reversed2)

# 测试用例 2: 单节点链表
print("测试用例 2: 单节点链表")
list3 = ListNode(1)
print("原链表: ", end="")
ListNode.print_list(list3)
reversed3 = ReverseLinkedListSolution.reverse_list_iterative(list3)
print("反转后: ", end="")
ListNode.print_list(reversed3)

# 测试用例 3: 空链表
print("测试用例 3: 空链表")
list4 = None
print("原链表: ", end="")
ListNode.print_list(list4)
reversed4 = ReverseLinkedListSolution.reverse_list_iterative(list4)
print("反转后: ", end="")
ListNode.print_list(reversed4)

print("所有测试用例执行完成")
print("=====")

```

```

class PalindromeLinkedListSolution:
"""

```

题目 14: LeetCode 234. 回文链表

来源: LeetCode

链接: <https://leetcode.cn/problems/palindrome-linked-list/>

题目描述:

给你一个单链表的头节点 head，请你判断该链表是否为回文链表。如果是，返回 true；否则，返回

```
false .
```

解法分析:

1. 快慢指针 + 反转链表 - 时间复杂度: $O(n)$, 空间复杂度: $O(1)$

解题思路:

1. 使用快慢指针找到链表中点
2. 反转后半部分链表
3. 比较前半部分和反转后的后半部分
4. 恢复链表结构(可选)

```
"""
```

```
@staticmethod
def is_palindrome(head: Optional[ListNode]) -> bool:
    """
```

解法: 快慢指针 + 反转链表

🎯 核心思想:

1. 使用快慢指针找到链表中点
2. 反转后半部分链表
3. 比较前半部分和反转后的后半部分

📊 复杂度分析:

- 时间复杂度: $O(n)$ - 需要遍历链表多次
- 空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

```
@param head 链表头节点
```

```
@return 是否为回文链表
```

```
"""
```

```
# 边界条件检查
```

```
if head is None or head.next is None:
    return True
```

```
# 第一步: 使用快慢指针找到链表中点
```

```
slow = head
```

```
fast = head
```

```
while fast.next is not None and fast.next.next is not None:
```

```
    slow = slow.next
```

```
    fast = fast.next.next
```

```
# 第二步: 反转后半部分链表
```

```
second_half = PalindromeLinkedListSolution._reverse_list(slow.next)
```

```

# 第三步：比较前半部分和反转后的后半部分
first_half = head
second_half_copy = second_half # 保存用于恢复
is_palindrome = True

while second_half is not None:
    if first_half.val != second_half.val:
        is_palindrome = False
        break
    first_half = first_half.next
    second_half = second_half.next

# 第四步：恢复链表结构(可选)
slow.next = PalindromeLinkedListSolution._reverse_list(second_half_copy)

return is_palindrome

@staticmethod
def _reverse_list(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    反转链表的辅助函数
    """
    prev = None
    current = head

    while current is not None:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    return prev

@staticmethod
def test():
    """
    测试方法
    """
    print("==== LeetCode 234. 回文链表测试 ====")

    # 测试用例 1：回文链表
    print("测试用例 1：回文链表")

```

```

list1 = ListNode.create_list([1, 2, 2, 1])
print("链表: ", end="")
ListNode.print_list(list1)
print(f"是否为回文链表: {PalindromeLinkedListSolution.is_palindrome(list1)}")

# 测试用例 2: 非回文链表
print("测试用例 2: 非回文链表")
list2 = ListNode.create_list([1, 2, 3, 4])
print("链表: ", end="")
ListNode.print_list(list2)
print(f"是否为回文链表: {PalindromeLinkedListSolution.is_palindrome(list2)}")

# 测试用例 3: 单节点链表
print("测试用例 3: 单节点链表")
list3 = ListNode(1)
print("链表: ", end="")
ListNode.print_list(list3)
print(f"是否为回文链表: {PalindromeLinkedListSolution.is_palindrome(list3)}")

print("所有测试用例执行完成")
print("=====")

```

class RemoveNthNodeFromEndOfListSolution:

"""

题目 15: LeetCode 19. 删除链表的倒数第 N 个结点

来源: LeetCode

链接: <https://leetcode.cn/problems/remove-nth-node-from-end-of-list/>

题目描述:

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

解法分析:

1. 快慢指针法 - 时间复杂度: O(n)，空间复杂度: O(1)

解题思路:

使用两个指针，快指针先移动 $n+1$ 步，然后快慢指针同时移动，

当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点。

"""

@staticmethod

```

def remove_nth_from_end(head: Optional[ListNode], n: int) -> Optional[ListNode]:
    """
    """

```

解法：快慢指针法

核心思想：

1. 使用哨兵节点简化边界处理
2. 快指针先移动 $n+1$ 步
3. 快慢指针同时移动
4. 当快指针到达链表末尾时，慢指针正好指向要删除节点的前一个节点

复杂度分析：

- 时间复杂度: $O(n)$ - 需要遍历链表一次
- 空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

```
@param head 链表头节点
@param n 倒数第 n 个节点
@return 删除节点后的链表头节点
"""

# 创建哨兵节点，简化边界处理
dummy = ListNode(0)
dummy.next = head

# 初始化快慢指针
fast = dummy
slow = dummy

# 快指针先移动 n+1 步
for i in range(n + 1):
    fast = fast.next

# 快慢指针同时移动
while fast is not None:
    fast = fast.next
    slow = slow.next

# 删除倒数第 n 个节点
slow.next = slow.next.next

# 返回头节点
return dummy.next

@staticmethod
def test():
"""
测试方法

```

```

"""
print("== LeetCode 19. 删除链表的倒数第 N 个结点测试 ==")

# 测试用例 1: 删除中间节点
print("测试用例 1: 删除中间节点")
list1 = ListNode.create_list([1, 2, 3, 4, 5])
print("原链表: ", end="")
ListNode.print_list(list1)
result1 = RemoveNthNodeFromEndOfListSolution.remove_nth_from_end(list1, 2)
print("删除倒数第 2 个节点后: ", end="")
ListNode.print_list(result1)

# 测试用例 2: 删除头节点
print("测试用例 2: 删除头节点")
list2 = ListNode.create_list([1, 2, 3, 4, 5])
print("原链表: ", end="")
ListNode.print_list(list2)
result2 = RemoveNthNodeFromEndOfListSolution.remove_nth_from_end(list2, 5)
print("删除倒数第 5 个节点后: ", end="")
ListNode.print_list(result2)

# 测试用例 3: 删除尾节点
print("测试用例 3: 删除尾节点")
list3 = ListNode.create_list([1, 2, 3, 4, 5])
print("原链表: ", end="")
ListNode.print_list(list3)
result3 = RemoveNthNodeFromEndOfListSolution.remove_nth_from_end(list3, 1)
print("删除倒数第 1 个节点后: ", end="")
ListNode.print_list(result3)

print("所有测试用例执行完成")
print("=====")

```

```
class AlgorithmSummary:
```

```
"""

```

```
    算法总结与技巧提升
"""

```

```
"""

```

```
@staticmethod
```

```
def print_summary():
    """

```

```
        打印算法总结
    """

```

```
"""
print("===== 链表合并算法总结 =====")
print("1. 核心算法技巧:")
print("    - 双指针法: 适用于两个有序序列的合并, 时间复杂度 O(m+n)")
print("    - 优先队列法: 适用于 K 个有序序列的合并, 时间复杂度 O(N*logK)")
print("    - 分治法: 适用于 K 个序列的归并, 时间复杂度 O(N*logK)")
print("    - 哨兵节点: 简化链表操作的边界处理, 提高代码可读性")
print("    - 原地修改: 避免额外空间开销, 适用于数组合并等场景")
print()
print("2. 工程化考量:")
print("    - 异常处理: 处理空链表、单节点链表等边界情况")
print("    - 内存管理: 在 Python 中通过垃圾回收自动管理内存")
print("    - 性能优化: 对于大规模数据, 优先队列的常数项优化很重要")
print("    - 线程安全: 在多线程环境下需要考虑同步问题")
print()
print("3. 调试技巧:")
print("    - 打印中间状态: 使用 print 跟踪指针移动")
print("    - 边界测试: 测试空输入、单元素输入、极端值等情况")
print("    - 断言验证: 使用 assert 验证关键条件是否满足")
print()
print("4. 拓展应用:")
print("    - 归并排序: 链表排序的最佳选择之一")
print("    - 多路归并: 外部排序的基础算法")
print("    - 数据流处理: 实时合并多个有序数据流")
print("=====\\n")
```

```
def run_all_tests():
"""
综合测试函数
"""

MergeTwoSortedListsSolution.test()
MergeKSortedListsSolution.test()
MergeSortedArraySolution.test()
SortListSolution.test()
AddTwoNumbersSolution.test()
SwapNodesInPairsSolution.test()
NowCoderMergeSortedListsSolution.test()
LintCodeMergeKListsSolution.test()
PartitionListSolution.test()

# 新增题目的测试
LinkedListCycleSolution.test()
```

```
LinkedListCycleIISolution. test()  
IntersectionOfTwoLinkedListsSolution. test()  
ReverseLinkedListSolution. test()  
PalindromeLinkedListSolution. test()  
RemoveNthNodeFromEndOfListSolution. test()
```

```
AlgorithmSummary. print_summary()
```

```
if __name__ == "__main__":  
    run_all_tests()  
=====
```