链滴

链表问题一些常用的套路与方法

作者：vcjmhg
原文链接：https://ld246.com/article/1602838561347
来源网站：链滴

概述

链表问题应该是数据结构中比较基础的一类问题，但同时也是在面试中常考的一类问题。但是围绕链问题的一些基本方法或者处理思想，也无外乎那几类，因此本文尝试对链表常用的一些方法或者套路行总结。

常用方法

1.头结点

增加头结点或者说哑巴节点这种方式，应该是我们在处理链表问题最常用的处理方式。简单来说引入结点有两个优点：

1. 由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作和在表的其他置上的操作一致，无需进行特殊处理。

2. 无论链表是否为空，其头指针是指向头结点的非空指针（空表中头结点的指针域为空），因此空表非控表的处理也就统一了。

总而言之，通过增加头结点，减少了在链表处理过程中对边界情况的判断，大大简化了程序的编写。

下边我们看一个例子：

leetcode 82删除排序链表中的重复元素 II：

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。

示例 1:

输入: 1->2->3->3->4->4->5

输出: 1->2->5

示例 2:

输入: 1->1->1->2->3

输出: 2->3

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list-ii

这个问题，可能解决问题的思路比较容易想：

由于链表的节点是有序的，因此我们可以在对链表进行遍历的过程中，可以比较当前遍历的节点(current)和其下一个节点（current.next）是否相等，如果相等则删除当前遍历的节点(current)，指针指向节点的下一个节点，继续进行该操作。

具体代码如下所示:

```
public static ListNode deleteDuplicates(ListNode head) {
  ListNode current = head;
  current = current.next;
  while (current != null) {
    while (current.next != null && current.val == current.next.val) {
      current.next = current.next.next;
    }
    current = current.next;
  }
  return head;
}
```

整个解法应该比较容易理解，但此时我们考虑如果此处不使用头节点该如何解决该问题?

如果取消了头结点，我们就需要考虑对链表第一个节点的处理，因为在该问题上，链表的第一个节点是有很大可能为重复节点，因此我们此处显然需要增加一个边界情况的判断，判断头结点是否为重复点。(head.val == head.next.val)。并且需要针对其为头结点的情况单独进行处理。

因此，此处我们可以简单总结一下头结点方法的使用场景：

只要是要处理的链表**第一个节点本身会发生变化的情况**都要考虑使用头结点，因为引入之后可能会极的减少对边界清理的处理。

2. 链表排序

链表排序，本身也可以是一个算法的题目，同时也是我们在解决链表问题时常用的中间手段。

下边我们看一个题目：

148. 排序链表:

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。

示例 1:

输入: 4->2->1->3

输出: 1->2->3->4

示例 2:

输入: -1->5->3->4->0

输出: -1->0->3->4->5

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/sort-list

该问题明显就是一个链表问题，但其难点可能在于**对空间复杂度和时间复杂度的要求比较严苛。这就致我们许多容易想到的方法都不能用，比如插入排序**、存思想等。因此此处我们必须从复杂度为O(nlogn)的排序算法中寻找到一个能用的，并且空间复杂度只常数级别的算法。

首先考虑，时间复杂度小于等于O(nlog(n))的算法有：

1. 折半插入排序
2. 希尔排序
3. 快速排序
4. 堆排序
5. 归并排序
6. 基数排序

同时我们考虑到链表本身性能比较差，因此如果排序过程涉及大量的随机访问，大概率该算法不能用比如快速排序、折半插入排序、希尔排序、堆排序（建堆的过程）。这些算法都不适用于链表。

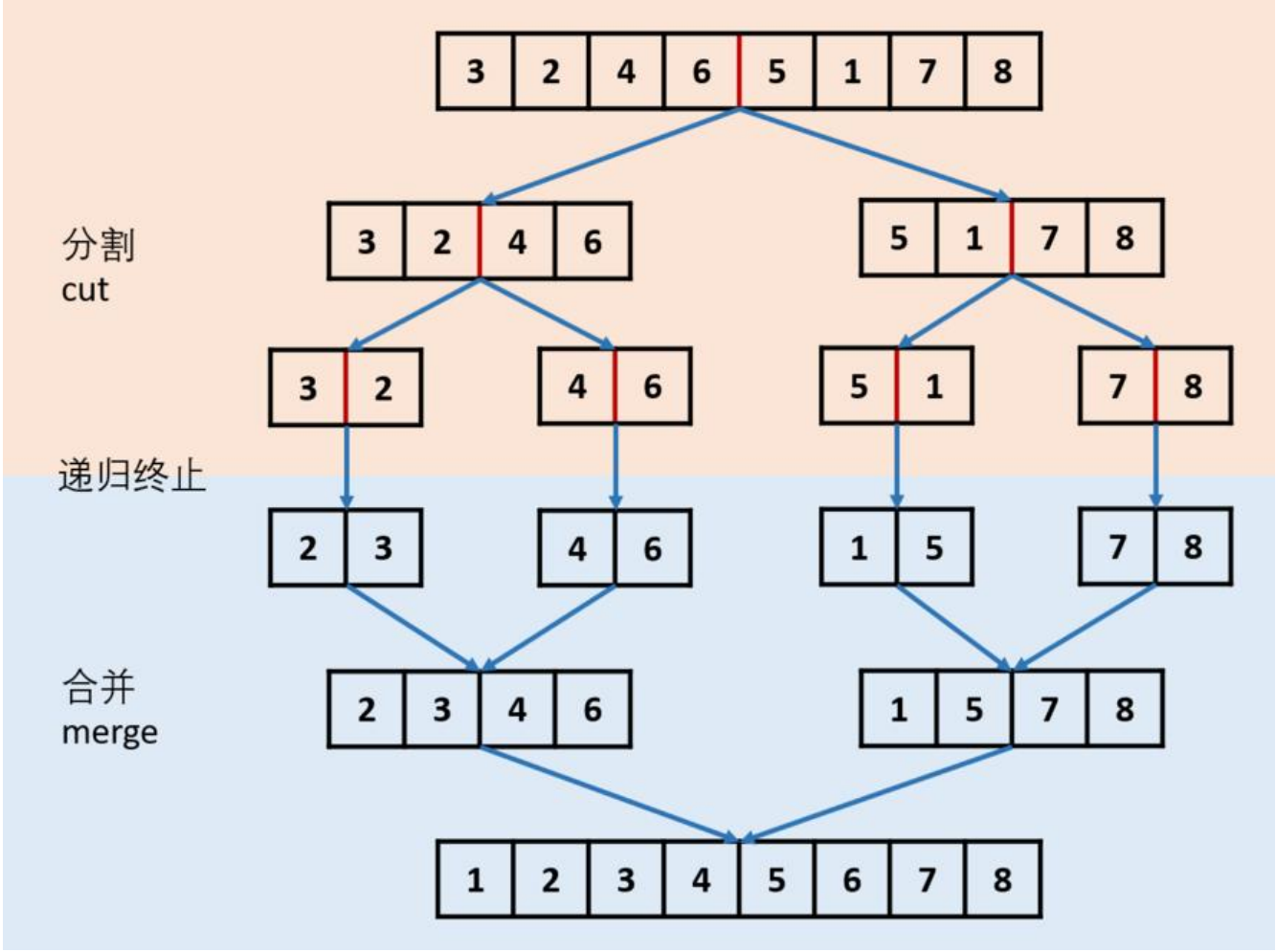
基数排序空间复杂度比较大一般是O（r），r是排序队列的个数。因此也不实用该题目。

最后我们只能考虑使用归并排序。

整个过程可以分成如下步骤：

1. **拆分：**找到链表中间节点，获取左半链表和右半链表
2. **排序：**分别对左侧链表和右侧链表进行归并排序
3. **合并：**将排序后的左侧链表和右侧链表进行合并

其过程可以简单用下图来表示：



代码实现如下：

```
// 使用归并算法进行链表排序
  public static ListNode mergesort(ListNode head) {
    // 如果链表只有一个节点直接返回
    if (head == null || head.next == null) {
      return head;
    }
    // 找到链表的中间节点
    ListNode middle = findMiddle(head);
    // 获取后半段的链表节点,同时与前半段节点断开
    ListNode tail = middle.next;
    middle.next = null;

    // 对左侧进行排序
    ListNode left = mergesort(head);
    // 对右侧进行排序
    ListNode right = mergesort(tail);
    // 合并两条链,注意是将left和right进行合并
    ListNode result = merge(left, right);
    return result;
  }
  // 将两个有序链表进行合并
  private static ListNode merge(ListNode left, ListNode right) {
    ListNode headNode = new ListNode(0);
```

原文链接：链表问题一些常用的套路与方法

```
    ListNode tail = headNode;
    headNode.next = left;

    while (left != null && right != null) {
      if (left.val < right.val) {
        tail.next = left;
        left = left.next;
      } else {
        tail.next = right;
        right = right.next;
      }
      tail = tail.next;
    }

    // 将非空的节点直接链接到temp后边
    if (left != null) {
      tail.next = left;
    }
    if (right != null) {
      tail.next = right;
    }
    return headNode.next;
  }

  // 寻找链表的中间节点，可以使用快慢指针
  private static ListNode findMiddle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head.next;
    while (fast != null && fast.next != null) {
      // 快指针一次走两步
      fast = fast.next.next;
      // 慢指针一次走一步
      slow = slow.next;
    }
    return slow;
  }
```

3. 链表插入与删除

链表的插入与删除操作应该是解决链表问题最常用的基础手段。但关于链表的插入和删除还是有若干问的，比如对链表进行插入，就可以简单分成**头插法和尾插法**。链表删除也是有若干边界情况要考虑但关于这两个操作是比较基础的，此处不进行详述，我们此处就看几个例题来回顾一下。

83. 删除排序链表中的重复元素

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

输入: 1->1->2

输出: 1->2

示例 2:

输入: 1->1->2->3->3

输出: 1->2->3

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list

这个问题比较容易，此处直接给出一种思路，通过双指针来进行解决，一个指针指向当前遍历节点，一个指针指向当前遍历节点的上一个节点，两个节点值一直，则删除当前遍历节点，以此类推。

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode current = head;
    while (current != null && current.next != null) {
        if (current.next.val == current.val) {
            current.next = current.next.next;
        } else {
            current = current.next;
        }
    }
    return head;
}
```

4. 翻转链表

链表翻转应该是我们解决某些特殊问题的时候，比较有效的突破口，尤其是那些对链表按照指定规则行重排序的问题，在无计可施的时候，通过翻转有时候可以有效的找到突破口。

143.重排链表

给定一个单链表 L：L0→L1→…→Ln-1→Ln ，

将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1:

给定链表 1->2->3->4, 重新排列为 1->4->2->3.

示例 2:

给定链表 1->2->3->4->5, 重新排列为 1->5->2->4->3.

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/reorder-list

通过链表翻转，我们可以迅速想到一种**思路**：

找到中点断开，翻转后面部分，然后合并前后两个链表

具体代码实现如下：

```java
/** 按照要求重新对链表进行排序: 给定链表 1->2->3->4, 重新排列为 1->4->2->3. */
 public static void reorderList(ListNode head) {
  // 边界情况进行处理，如果只有一个节点或者节点为空，直接返回
  if (head == null || head.next == null) {
   return;
  }
  // 如果只有一个节点直接反掌结果
  if (head == null || head.next == null) {
   return;
  }
  ListNode headNode = new ListNode(0);
  headNode.next = head;
  // 获取中间链表节点
  ListNode middle = getMiddle(head);
  ListNode tail = middle.next;
  middle.next = null;
  // 对后一半的链表进行翻转
  tail = reverse(tail);
  ListNode temp = headNode;
  // 将两段链表交替连接
  while (head != null && tail != null) {
   temp.next = head;
   head = head.next;
   temp = temp.next;

   temp.next = tail;
   tail = tail.next;
   temp = temp.next;
  }
  // 非空节点连接到链表末尾
  if (head != null) {
   temp.next = head;
  }
  if (tail != null) {
   temp.next = tail;
  }
 }
 // 对链表进行翻转
 private static ListNode reverse(ListNode head) {
  ListNode pre = null;
  ListNode temp;
  while (head != null) {
   temp = head.next;
   // 此处应该通过head的next来实现翻转
   head.next = pre;
   pre = head;
   head = temp;
  }
  return pre;
 }

 // 获取链表的中间节点
 private static ListNode getMiddle(ListNode head) {
  ListNode slow = head;
```

```
    ListNode fast = head.next;
    while (fast != null && fast.next != null) {
      fast = fast.next.next;
      slow = slow.next;
    }
    return slow;
  }
```

当然解决该问题的方法不知一种，比如我们也可以用存储的思路的来解决，将链表转成ArrayList来进解决。具体代码如下：

```
  /**
   *  使用存储解决该问题
   *  基本思路：将链表转换乘ArraryList类型然后从后向前来进行遍历
   * @param head
   */
  public static void reorderListByStorage(ListNode head) {
    //边界情况处理
    if(head == null || head.next == null){
      return;
    }
    //将链表转换乘ArrayList进行操作
    ArrayList<ListNode> list = new ArrayList<ListNode>();
    while (head != null){
      list.add(head);
      head = head.next;
    }
    //通过双指针连接乘新的链表
    int i=0,j=list.size()-1;
    while (i<j){
      list.get(i).next = list.get(j);
      i++;
      //边界情况处理 i == j
      if(i == j){
        break;
      }
      list.get(j).next=list.get(i);
      j--;
    }
    //将最后一个节点的next置空
    list.get(i).next = null;
  }
```

5. 快慢指针

快慢指针或者说双指针，毫无疑问是解决链表问题最常用的操作，比如在寻找链表中间节点的时候就常用。

```
private static ListNode getMiddle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head.next;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
```

```
    }
    return slow;
}
```

而且应用起来也比较灵活，比如可以用来判断链表是否有环：

141. 环形链表

给定一个链表，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。 为了表示给定链中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1 则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true 。 否则，返回 false 。

来源：力扣（LeetCode）

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

思路：快慢指针，快慢指针相同则有环，证明：如果有环每走一步快慢指针距离会减 1

代码如下：

```
/**
     * 使用快慢指针来解决链表是否有环的判断
     * @param head
     * @return
     */
    public static boolean hasCycle(ListNode head){
        //边界情况处理
        if (head ==null || head.next ==null){
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next;
        boolean hasCycle = false;
        while (fast != null && fast.next != null){
            //如果两个指针重逢则证明一定有环
            if (fast == slow){
                hasCycle = true;
                break;
            }
            //slow指针每次走一步，fast指针每次走两步
            slow = slow.next;
            fast = fast.next.next;
        }
        return hasCycle;
    }
```

总结

本文主要总结了解决链表问题时，常用的一些套路，比如增加头结点、对链表进行排序、节点插入与

除、链表翻转以及快慢指针，希望能给读者以帮助。