链表的陨落篇

链表篇内容主要介绍三个部分和相关的题目：基础技术，其中包括增删一个结点，反转整个链表，合并两个有序链表以及找链表的中点；介绍 Dummy Node 的重要性及应用；介绍 Two Pointers 方法的重要性及应用。

Part 1 – Basic Skills in Linked List

以下要介绍的链表基础技术非常重要，虽然可能不会直接出如此简单的题目，但几乎所有的链表题目都要使用这些基础技术做为Subroutine，娴熟的掌握链表基础技术可以让你将复杂题目拆解成由基础技术组成的Subroutine，让复杂题目也可以迎刃而解。

下面以4段代码来分别讲解这些基础技术：

1. Remove Duplicates from Sorted List
2. Reverse Linked List
3. Merge Two Sorted Lists
4. Find Middle Element in Linked List

1. Remove Duplicates from Sorted List，由于增删一个结点很类似，所以选择了一道删除的题目来代表此类操作。增删一个结点的关键是使用“差一法”，即获得要增加或者删除位置结点的前一个结点 pre，然后使用 pre.next = cur.next; 来将 cur 结点删除。

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

```
        return head;
    }
```

2. Reverse Linked List，反转链表也是链表题目中很常见的操作，所以我们一定要熟悉。在这里介绍两种反转链表的方法，分别用递归和迭代完成。

```
    // Solution 1 – Recursion
    public ListNode reverse(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode newHead = reverse(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }

    // Solution 2 – Iteration
    public ListNode reverse(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode pre = null;
        ListNode cur = head;
        while(cur != null){
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
```

3. Merge Two Sorted Lists，合并两个链表也是很常见的链表操作。这里用到了Dummy Node技术，而且这种以一个链表为模板，将另一个链表插入其中的思想也值得借鉴。

```
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        dummy.next = l1;
        ListNode pre = dummy;
```

```
        while(l1 != null && l2 != null){
            if(l1.val <= l2.val){
                l1 = l1.next;
            } else {
                ListNode next = l2.next;
                l2.next = pre.next;
                pre.next = l2;
                l2 = next;
            }
            pre = pre.next;
        }
        if(l2 != null){
            pre.next = l2;
        }
        return dummy.next;
}
```

4. Find Middle Element in Linked List，找链表中点的操作往往在要将链表一分为二的题目当中，例如使用 Merge Sort 来解 Sort List 这道题时就会使用到。同时这里还使用到了 Two Pointers 技术，也叫 walker & runner 技术，后面会具体说明。

```
    ListNode walker = head;
    ListNode runner = head.next;
    while(runner != null && runner.next != null){
        walker = walker.next;
        runner = runner.next.next;
    }
```

此时的 walker 一定指向中间结点。如果要拆分链表，walker.next 即会第二段链表的头。

Part 2 – Introduce Dummy Node in Linked List

Dummy Node技术在链表中是非常非常重要的，它可以避免复杂的分情况讨论，简化思路和代码，还可以提高正确率。切记，当题目要求可能会改变链表头部或者要创建一个新的链表的时候，请务必使用Dummy Node。下面同样选取例题进行讲解该技术，其中前4题是由于需要改变链表头而使用Dummy Node，而后2题是因为需要创建一个新的链表而使用Dummy Node，创建新链表时常和pre指针一同使用。

在 LeetCode 中使用 Dummy Node 技术的题目有：

1. Remove Duplicates from Sorted List II
2. Reverse Linked List II
3. Swap Nodes in Pairs
4. Reverse Nodes in k-Group
5. Partition List
6. Add Two Numbers

1. Remove Duplicates from Sorted List II，由于头部结点可能有重复所以也可能被删掉，此时就需要使用Dummy Node来避免讨论，题目比较简单，理清3种情况的应对方法即可。还有一个tricky的地方是，最后要单独处理一下cur.next == null而又有重复结点的终止。

```
public ListNode deleteDuplicates(ListNode head) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode pre = dummy;
    ListNode cur = head;
    while(cur != null && cur.next != null){
        if(cur.val != cur.next.val){
            if(pre.next == cur){
                pre = pre.next;
            } else {
                pre.next = cur.next;
            }
        }
        cur = cur.next;
    }
    if(cur.next == null && pre.next != cur){
        pre.next = null;
    }
    return dummy.next;
}
```

2. Reverse Linked List II，当m = 1时，链表头部也是需要参与反转的，此时Dummy Node又有出场的机会了。同时，注意下m，n与移动次数的关系即可。

```
public ListNode reverseBetween(ListNode head, int m, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode pre = dummy;
    int count = m - 1;
    while(count > 0){
        pre = pre.next;
        count--;
    }
    ListNode cur = pre.next;
    count = n - m;
    while(count > 0){
        ListNode next = cur.next.next;
        cur.next.next = pre.next;
        pre.next = cur.next;
        cur.next = next;
        count--;
    }
    return dummy.next;
}
```

3. Swap Nodes in Pairs，由于要改变相邻两个结点的位置，所以会改变头结点的位置，因此需要Dummy Node。这道题相对简单，不再赘述。

```
public ListNode swapPairs(ListNode head) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode pre = dummy;
    ListNode cur = head;
    while(cur != null && cur.next != null){
        ListNode next = cur.next.next;
        cur.next.next = pre.next;
        pre.next = cur.next;
        cur.next = next;
```

```
            pre = cur;
            cur = next;
        }
        return dummy.next;
    }
```

4. Reverse Nodes in k-Group，由于这道题也会改变链表的头结点，Dummy Node必不可少。使用一个count变量来记录反转段结点数量是否足够k个。注意reverse方法的写法。

```
    public ListNode reverseKGroup(ListNode head, int k) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        ListNode cur = head;
        int count = 0;
        while(cur != null){
            count++;
            ListNode next = cur.next;
            if(count == k){
                pre = reverse(pre, next);
                count = 0;
            }
            cur = next;
        }
        return dummy.next;
    }

    private ListNode reverse(ListNode pre, ListNode next){
        ListNode cur = pre.next;
        while(cur.next != next){
            ListNode tempNext = cur.next.next;
            cur.next.next = pre.next;
            pre.next = cur.next;
            cur.next = tempNext;
        }
        return cur;
    }
```

5. Partition List，这道题要求值小于x的结点必须出现在值大于等于x的结点左面。所以我们构造2个链表less和more来分别存储这两种结点，最后组合一下即可。注意，最后需要将morePre.next置为null，不然会出现错误。

```
public ListNode partition(ListNode head, int x) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode lessDummy = new ListNode(0);
    ListNode moreDummy = new ListNode(0);
    ListNode lessPre = lessDummy;
    ListNode morePre = moreDummy;
    while(head != null){
        if(head.val < x){
            lessPre.next = head;
            lessPre = lessPre.next;
        } else {
            morePre.next = head;
            morePre = morePre.next;
        }
        head = head.next;
    }
    lessPre.next = moreDummy.next;
    morePre.next = null;
    return lessDummy.next;
}
```

6. Add Two Numbers，这道题由于要构造一个新的链表来表示求和的结果，所以我们依然使用Dummy Node技术。题目比较简单，使用一个carry来保存进位即可。注意，最后如果carry不为0的话，说明还要新建一个结点来保存一个最高位。

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode pre = dummy;
    int carry = 0;
    while(l1 != null && l2 != null){
        int value = l1.val + l2.val + carry;
        ListNode newNode = new ListNode(value % 10);
        carry = value / 10;
        pre.next = newNode;
```

```
            pre = pre.next;
            l1 = l1.next;
            l2 = l2.next;
        }
        while(l1 != null){
            int value = l1.val + carry;
            ListNode newNode = new ListNode(value % 10);
            carry = value / 10;
            pre.next = newNode;
            pre = pre.next;
            l1 = l1.next;
        }
        while(l2 != null){
            int value = l2.val + carry;
            ListNode newNode = new ListNode(value % 10);
            carry = value / 10;
            pre.next = newNode;
            pre = pre.next;
            l2 = l2.next;
        }
        if(carry == 1){
            ListNode newNode = new ListNode(1);
            pre.next = newNode;
        }
        return dummy.next;
    }
```

Part 3 – Two Pointers

Two Pointers技术，也被称为Walker & Runner技术，是解决很多链表题目的利器。用法分为两种：第一种，walker每次移动一步而runner每次移动两步，一般用来解决Linked List Cycle这类找循环点的题目；第二种，walker和runner速度一样，一般用来解决Remove Nth Node From End of List这类要找倒数第几个结点或者要求walker和runner之间有固定距离的题目。下面同样选取例题进行讲解该技术，其中前3题是由于需要寻找循环点而使用Two Pointers技术，而后2题是因为需要找到特定点而使用Two Pointers技术。请仔细体会Two Pointers技术的用法。

在 LeetCode 中使用 Two Pointers 技术的题目有：

1. Linked List Cycle
2. Linked List Cycle II
3. Intersection of Two Linked Lists
4. Remove Nth Node From End of List
5. Rotate List

1. Linked List Cycle，最经典的walker & runner技术用法，walker每次移动1步，runner每次移动2步，如果有循环walker和runner终会在某个结点相遇。

```
public boolean hasCycle(ListNode head) {
    if(head == null || head.next == null){
        return false;
    }
    ListNode walker = head;
    ListNode runner = head;
    while(runner != null && runner.next != null){
        walker = walker.next;
        runner = runner.next.next;
        if(walker == runner){
            return true;
        }
    }
    return false;
}
```

2. Linked List Cycle II，该题是上道题的升级版，在确认是否有循环的同时，如果有还要找到循环的起始点。由k = a + b + mc和2k = a + b + nc可得k = (n - m)c = a + b + mc可得a = (n – 2m)c – b。可知从链表头和相交点到循环点的距离是相同的。

```
public ListNode detectCycle(ListNode head) {
    if(head == null || head.next == null){
        return null;
    }
    ListNode walker = head;
    ListNode runner = head;
    while(runner != null && runner.next != null){
        walker = walker.next;
```

```
            runner = runner.next.next;
            if(walker == runner){
                break;
            }
        }
        if(walker != runner){
            return null;
        }
        walker = head;
        while(walker != runner){
            walker = walker.next;
            runner = runner.next;
        }
        return walker;
    }
```

3. Intersection of Two Linked Lists，这道题有3种解法，其中第三种使用了Two Pointers 技术，原理和上一道题一样，但并不推荐，因为要改变链表结构再改回来，推荐前2种解法。

```
    // Solution 1 - Same Length
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = getLength(headA);
        int lenB = getLength(headB);
        while(lenA > lenB){
            headA = headA.next;
            lenA--;
        }
        while(lenB > lenA){
            headB = headB.next;
            lenB--;
        }
        while(headA != headB){
            headA = headA.next;
            headB = headB.next;
        }
        return headA;
    }

    private int getLength(ListNode head){
        int count = 0;
```

```
        while(head != null){
            head = head.next;
            count++;
        }
        return count;
    }

    // Solution 2 - Change Head
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode curA = headA;
        ListNode curB = headB;
        while(curA != null && curB != null){
            curA = curA.next;
            curB = curB.next;
        }
        if(curA == null){
            curA = headB;
            while(curB != null){
                curA = curA.next;
                curB = curB.next;
            }
            curB = headA;
        } else {
            curB = headA;
            while(curA != null){
                curA = curA.next;
                curB = curB.next;
            }
            curA = headB;
        }
        while(curA != curB){
            curA = curA.next;
            curB = curB.next;
        }
        return curA;
}
```

```
// Solution 3 - LinkedList Cycle
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    if(headA == null || headB == null){
        return null;
    }
    ListNode tailB = headB;
    while(tailB.next != null){
        tailB = tailB.next;
    }
    tailB.next = headB;
    ListNode walker = headA;
    ListNode runner = headA;
    while(runner != null && runner.next != null){
        walker = walker.next;
        runner = runner.next.next;
        if(walker == runner){
            break;
        }
    }
    if(runner == null || runner.next == null || walker != runner){
        tailB.next = null;
        return null;
    }
    walker = headA;
    while(walker != runner){
        walker = walker.next;
        runner = runner.next;
    }
    tailB.next = null;
    return walker;
}
```

4. Remove Nth Node From End of List，该题属于最简单的第二种用法。由于要删除一个结点，所以要用“差一法”，即找到要删除结点的前一个结点。逆推出walker和runner间距离为2，所以先让runner跑2个结点，然后walker和runner同速后移，当runner走到最后一个结点，walker即为我们要找的pre结点。注意两个特殊情况，如果n大于链表长度，不需要删除直接返回head即可；如果count == n && runner == null时，说明要删除的是第一个结点，直接返回head.next即可。

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    if(head == null || n <= 0){
        return head;
    }
    ListNode walker = head;
    ListNode runner = head;
    int count = 0;
    while(runner!= null && count < n){
        runner = runner.next;
        count++;
    }
    if(count < n){
        return head;
    }
    if(runner == null){
        return head.next;
    }
    while(runner.next != null){
        walker = walker.next;
        runner = runner.next;
    }
    walker.next = walker.next.next;
    return head;
}
```

5. Rotate List，这道题比较简单，但有一个比较tricky的地方就是所给的n可能大于链表长度，这时需要让n对链表长度进行取模。有两个办法，第一种是先计算链表长度，然后取模，再跑runner指针，这种的劣势是无论n是否大于链表长度都需要计算跑完整个链表一次；第二种是一边跑runner指针一边计算链表长度，如果n大于链表长度再取模重新跑runner即可。

```
public ListNode rotateRight(ListNode head, int n) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode walker = head;
    ListNode runner = head;
    int count = 0;
    while(runner.next != null && count < n){
        runner = runner.next;
```

```
            count++;
        }
        if(count < n){
            n %= count + 1;
            runner = head;
            count = 0;
            while(runner.next != null && count< n){
                runner = runner.next;
                count++;
            }
        }
        while(runner.next != null){
            walker = walker.next;
            runner = runner.next;
        }
        runner.next = head;
        ListNode newHead = walker.next;
        walker.next = null;
        return newHead;
    }
```

Part 4 – 综合题

现在基础技术，Dummy Node 和 Two Pointers 技术我们都介绍完毕，下面介绍几道综合题目来讲解下他们是如何在复杂题目中应用的。

在 LeetCode 中链表的综合题目有以下几道：

1. Reorder List
2. Merge k Sorted Lists
3. Copy List with Random
4. Insertion Sort List
5. Sort List

1. Reorder List，找链表中点 + 反转链表 + 增加结点。

```
    public void reorderList(ListNode head) {
        if(head == null || head.next == null){
            return;
```

```
        }
        ListNode walker = head;
        ListNode runner = head.next;
        while(runner != null && runner.next != null){
            walker = walker.next;
            runner = runner.next.next;
        }
        ListNode head2 = walker.next;
        walker.next = null;
        head2 = reverse(head2);
        while(head2 != null){
            ListNode next = head2.next;
            head2.next = head.next;
            head.next = head2;
            head = head2.next;
            head2 = next;
        }
    }

    private ListNode reverse(ListNode head){
        ListNode pre = null;
        ListNode cur = head;
        while(cur != null){
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }

    private ListNode reverse(ListNode head){
        if(head.next == null){
            return head;
        }
        ListNode newHead = reverse(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
}
```

2. Merge k Sorted Lists，第一种，分治 + 合并链表；第二种，Heap + Dummy Node.

```
// Solution 1 - Divide and Conquer
public ListNode mergeKLists(List<ListNode> lists) {
    if(lists == null || lists.size() == 0){
        return null;
    }
    return helper(lists, 0, lists.size() - 1);
}

private ListNode helper(List<ListNode> lists, int left, int right){
    if(left == right){
        return lists.get(left);
    }
    int mid = left + (right - left) / 2;
    ListNode head1 = helper(lists, left, mid);
    ListNode head2 = helper(lists, mid + 1, right);
    return merge(head1, head2);
}

private ListNode merge(ListNode l1, ListNode l2){
    ListNode dummy = new ListNode(0);
    dummy.next = l1;
    ListNode pre = dummy;
    while(l1 != null && l2 != null){
        if(l1.val <= l2.val){
            l1 = l1.next;
        } else {
            ListNode next = l2.next;
            l2.next = pre.next;
            pre.next = l2;
            l2 = next;
        }
        pre = pre.next;
    }
    if(l2 != null){
        pre.next = l2;
    }
    return dummy.next;
}
```

```
// Solution 2 - Heap
public ListNode mergeKLists(List<ListNode> lists) {
    if(lists == null || lists.size() == 0){
        return null;
    }
    PriorityQueue<ListNode> heap = new PriorityQueue<ListNode>(lists.size(),
new Comparator<ListNode>(){
        public int compare(ListNode n1, ListNode n2){
            return n1.val - n2.val;
        }
    });
    for(int i = 0; i < lists.size(); i++){
        ListNode node = lists.get(i);
        if(node != null){
            heap.offer(node);
        }
    }
    ListNode dummy = new ListNode(0);
    ListNode pre = dummy;
    while(!heap.isEmpty()){
        ListNode cur = heap.poll();
        pre.next = cur;
        pre = pre.next;
        if(cur.next != null){
            heap.offer(cur.next);
        }
    }
    return dummy.next;
}
```

3. Copy List with Random，第一种，Dummy Node + HashMap；第二种，复制+拆分。

```
// Solution 1 - HashMap -> O(n) Space Complexity
public RandomListNode copyRandomList(RandomListNode head) {
    RandomListNode dummy = new RandomListNode(0);
    RandomListNode pre = dummy;
    RandomListNode cur = head;
    HashMap<RandomListNode, RandomListNode> map = new
HashMap<RandomListNode, RandomListNode>();
    while(cur != null){
```

```
            RandomListNode newNode = new RandomListNode(cur.label);
            map.put(cur, newNode);
            pre.next = newNode;
            pre = pre.next;
            cur = cur.next;
        }
        cur = head;
        while(cur != null){
            map.get(cur).random = map.get(cur.random);
            cur = cur.next;
        }
        return dummy.next;
    }
    // Solution 2 - Original List -> O(1) Space Complexity
    public RandomListNode copyRandomList(RandomListNode head) {
        RandomListNode cur = head;
        while(cur != null){
            RandomListNode newNode = new RandomListNode(cur.label);
            newNode.next = cur.next;
            cur.next = newNode;
            cur = newNode.next;
        }
        cur = head;
        while(cur != null){
            if(cur.random != null){
                cur.next.random = cur.random.next;
            }
            cur = cur.next.next;
        }
        cur = head;
        RandomListNode dummy = new RandomListNode(0);
        RandomListNode pre = dummy;
        while(cur != null){
            pre.next = cur.next;
            pre = pre.next;
            cur.next = pre.next;
            cur = cur.next;
        }
        return dummy.next;
    }
```

4. Insertion Sort List，使用了Dummy Node技术，原理和Insertion Sort一模一样。

```
public ListNode insertionSortList(ListNode head) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode dummy = new ListNode(0);
    ListNode pre = dummy;
    ListNode cur = head;
    while(cur != null){
        pre = dummy;
        while(pre.next != null && cur.val > pre.next.val){
            pre = pre.next;
        }
        ListNode next = cur.next;
        cur.next = pre.next;
        pre.next = cur;
        cur = next;
    }
    return dummy.next;
}
```

5. Sort List，这道题同样提供两种做法：第一种，Merge Sort - 找链表中点 + 合并链表；第二种，Quick Sort – Dummy Node。

```
// Solution 1 - Merge Sort
public ListNode sortList(ListNode head) {
    if(head == null || head.next == null){
        return head;
    }
    ListNode walker = head;
    ListNode runner = head.next;
    while(runner != null && runner.next != null){
        walker = walker.next;
        runner = runner.next.next;
    }
    ListNode head1 = head;
    ListNode head2 = walker.next;
    walker.next = null;
    head1 = sortList(head1);
    head2 = sortList(head2);
```

```
        return merge(head1, head2);
    }

    private ListNode merge(ListNode l1, ListNode l2){
        ListNode dummy = new ListNode(0);
        dummy.next = l1;
        ListNode pre = dummy;
        while(l1 != null && l2 != null){
            if(l1.val <= l2.val){
                l1 = l1.next;
            } else {
                ListNode next = l2.next;
                l2.next = pre.next;
                pre.next = l2;
                l2 = next;
            }
            pre = pre.next;
        }
        if(l2 != null){
            pre.next = l2;
        }
        return dummy.next;
    }

    // Solution 2 - Quick Sort
    public ListNode sortList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode smallDummy = new ListNode(0);
        ListNode equalDummy = new ListNode(0);
        ListNode largeDummy = new ListNode(0);
        ListNode smallPre = smallDummy;
        ListNode equalPre = equalDummy;
        ListNode largePre = largeDummy;
        ListNode cur = head;
        while(cur != null){
            if(cur.val < head.val){
                smallPre.next = cur;
                smallPre = smallPre.next;
```

```
            } else if(cur.val > head.val){
                largePre.next = cur;
                largePre = largePre.next;
            } else {
                equalPre.next = cur;
                equalPre = equalPre.next;
            }
            cur = cur.next;
        }
        smallPre.next = null;
        equalPre.next = null;
        largePre.next = null;
        ListNode small = sortList(smallDummy.next);
        ListNode large = sortList(largeDummy.next);
        if(small == null){
            equalPre.next = large;
            return equalDummy.next;
        } else {
            cur = small;
            while(cur.next != null){
                cur = cur.next;
            }
            cur.next = equalDummy.next;
            equalPre.next = large;
            return small;
        }
    }
```