嗑一嗑数组与链表面试题

1、数组

面试题列表：

1：字节跳动，滴滴打车最近面试题：283. 移动零

2：华为，腾讯最近面试题：11. 盛最多水的容器

3：美团点评，快手最近面试题：88. 合并两个有序数组

4：facebook，谷歌最近面试题：66. 加一

5：字节跳动，美团点评最近面试题：9. 回文数

6：阿里，华为，腾讯最近面试题：1. 两数之和

7：腾讯，爱奇艺，拼多多最近面试题：15. 三数之和

8：华为，字节跳动最近面试题：18. 四数之和

9：顺丰，小米最近面试题，LCP 18. 早餐组合

10：腾讯，美团点评最近面试题：26. 删除排序数组中的重复项

11：微软，字节跳动最近面试题：189. 旋转数组

数组操作的核心思想：**数组下标变换，双指针（快慢指针）**，

（1）283. 移动零-C

1：**双指针思想（快慢指针）**，一维数组的下标变换操作思想

```
class Solution {
    // 双指针移动 时间复杂度O(n) 空间复杂度O(1)
    public void moveZeroes(int[] nums) {
        if (nums == null || nums.length < 2) {
            return;
        }
        //从前到后非零元素的指针
        int p1 = 0;
        for (int i =0; i < nums.length; i++) {
            if (nums[i] != 0) {
                nums[p1] = nums[i];
                p1++;
            }
        }
        while (p1 < nums.length) {
            nums[p1] = 0;
            p1++;
        }
    }
}
```

注意查看精选题解，还有一次循环解决问题的解法

（2）11. 盛最多水的容器-C

1：暴力破解，朴素解法

```
class Solution {
    //暴力破解法 枚举出所有坐标的组合,求出每个组合
    public int maxArea(int[] height) {
        //定义容纳水的最大值
        int max = 0 ;
        for (int i=0; i<height.length;i++) {
            for (int j=i+1;j<height.length;j++) {
                int area = (j-i) * Math.min(height[i], height[j]);
                max = Math.max(max, area);
            }
        }
        return max;
    }
}
```

2：双指针夹逼（左右指针）

```
class Solution {
    public int maxArea(int[] height) {
        //定义最大水的值
        int res = 0;
        //定义双指针
        int i=0;
        int j= height.length -1;
        while (i!=j) {
            int rest = Math.min(height[i],height[j]) * (j - i);
            if ( height[i] < height[j] ) {
                i++;
            }else {
                j--;
            }
            res = res > rest ? res:rest;
        }
        return res;
    }
}
```

（3）88. 合并两个有序数组-P

双指针

1： 时间复杂度O(m+n) 空间复杂度O(m)

```
class Solution {
    //双指针 时间复杂度O(m+n) 空间复杂度O(m)
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        //先将nums1中的m个数拷贝出来
        int[] nums1_cp = new int[m];
        System.arraycopy(nums1, 0, nums1_cp, 0, m);

        //循环nums1和nums2,两个指针分别依次后移并且比较大小
        int i=0;//nums1_cp下标
        int j=0;//nums2下标
        int p=0;//nums1下标
        while (i < m && j < n) {
            nums1[p++] = nums1_cp[i] > nums2[j] ? nums2[j++]:nums1_cp[i++];
        }

        //看哪个数组中还有剩余
        if (i<m) {
             // nums1_cp中还有剩余，把剩余的直接拷贝到nums1中
             System.arraycopy(nums1_cp, i, nums1, p, m-i);
        }
        if (j<n) {
            // nums2中还有剩余，把剩余的拷贝到nums1中去
            System.arraycopy(nums2, j, nums1, p, n-j);
        }
    }

}
```

2： 时间复杂度O(m+n) 空间复杂度O(1)

```
class Solution {
    //双指针法 空间复杂度O(1)
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        //定义三个指针
        int i = m -1;
        int j = n - 1;
        int p = m+n-1;

        while (i>=0 && j>=0) {
            nums1[p--] = nums1[i] > nums2[j] ? nums1[i--]:nums2[j--];
        }

        //只有nums2有剩余才需要处理
        if (j>=0) {
            System.arraycopy(nums2, 0, nums1, 0, j+1);
        }
    }
}
```

（4）66. 加一-C

数学处理

```
class Solution {
    public int[] plusOne(int[] digits) {
        //从数组尾部开始循环
        int carry = 1; //向下成后面还有一位，向前进了一位
        for (int i=digits.length-1; i>=0;i--) {
            int temp = digits[i];
            digits[i] = (temp + carry) % 10;
            carry = (temp + carry ) / 10;
            if (carry ==0) {
                break;
            }
        }
        if (carry > 0) {
            int[] result = new int[digits.length+1];
            result[0] = 1;
            return result;
        }
        return digits;
    }
}
```

（5）9. 回文数-P

双指针，数学处理

```
class Solution {
    //双指针夹逼
    public boolean isPalindrome(int x) {
        //x是负数或者x的末尾为0肯定不是回文数，但要排除0本身
        if (x < 0 || (x %10 ==0 && x!=0)) {
            return false;
        }
        //将x的后半段翻转，构造一个新的数和x前半段比较
        int revertNum = 0;

        while (x > revertNum) { //x的位数在缩小，revertNum的位数在增大
            revertNum =  revertNum *10 + x % 10;
            x = x / 10;
        }

        return x == revertNum || x == revertNum/10;
    }
}
```

（6）1. 两数之和-C

缓存思想

1：暴力朴素解法

```
class Solution {
    //暴力破解 两层for循环 时间复杂度O(n^2)
    public int[] twoSum(int[] nums, int target) {
        for (int i=0;i<nums.length;i++) {
            for (int j=i+1;j<nums.length;j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i,j};
                }
            }
        }
        return new int[]{};
    }
}
```

2：缓存

```
class Solution {
    //缓存 时间复杂度O(n)
    public int[] twoSum(int[] nums, int target) {
        //构造缓存
        Map<Integer,Integer> cache = new HashMap();
        for (int i=0;i<nums.length;i++) {
            if (cache.containsKey(target - nums[i])) {
                return new int[]{cache.get(target - nums[i]),i};
            }
            cache.put(nums[i], i);
        }
        return new int[]{};
    }
}
```

（7）15. 三数之和-C

1：暴力求解，难点在于去重，以下是未AC的代码

```
class Solution {
    //暴力解法，三层遍历
    public List<List<Integer>> threeSum(int[] nums) {
        if (nums==null || nums.length <3) {
            return Collections.emptyList();
        }
        Set<List<Integer>> result = new LinkedHashSet();
        //第一层循环
        for (int i = 0;i<nums.length-2;i++) {
            for (int j = i+1;j<nums.length-1;j++) {
                for (int k=j+1;k<nums.length;k++) {
                    if (nums[i] + nums[j] +nums[k] ==0) {
                        List list = new ArrayList();
                        list.add(nums[i]);
                        list.add(nums[j]);
                        list.add(nums[k]);
                        list.sort(Comparator.naturalOrder());
                        result.add(list);
                    }
                }
            }
        }
        return new ArrayList(result);
    }
}
```

2：分解成**n-2**个两数之和，使用缓存

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        //过滤掉明显不符合条件的
        if (nums==null || nums.length <3) {
            return Collections.emptyList();
        }
        // n-2种情况的twosum,目的是为了twosum时能使用缓存
        Set<List<Integer>> set = new LinkedHashSet();
        for (int i=0;i<nums.length-2;i++) {
            int target = -nums[i];

            Map<Integer,Integer> cache = new HashMap();
            for (int j=i+1;j<nums.length;j++) {
                if (cache.containsKey(target-nums[j])) {
                    List<Integer> list = new ArrayList();
                    list.add(-target);
                    list.add(nums[j]);
                    list.add(target-nums[j]);

                    list.sort(Comparator.naturalOrder());
                    set.add(list);
                }else {
                    cache.put(nums[j], j);
                }
            }
        }
        return new ArrayList(set);
    }
}
```

3：数组排序，双指针夹逼（左右指针）（适合已排好序数据），剪枝

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        //特殊情况直接返回
        if (nums == null || nums.length < 3) {
            return Collections.EMPTY_LIST;
        }
        //对数组排序,想利用已排好序的数组特性,a+b+c=0,不可能都是正数，也不可能都是负数，如果排好序后nums[i] >0,后面的就不用再看了肯定不会出现a+b+c=0的情况
        Arrays.sort(nums);

        //利用双指针
        List<List<Integer>> result = new ArrayList();
        for (int i=0;i<nums.length-2;i++) {
            if (nums[i] > 0) {
                break;
            }
            if (i>0 && nums[i] == nums[i-1]) {
                continue;
            }
            int a = nums[i];
            //定义双指针
            int j = i+1;//左指针
            int k = nums.length-1; //右指针
            while (j<k) {
                int b  = nums[j];
                int c =  nums[k];
                if ((a+b+c) == 0) {
                    List<Integer> list = new ArrayList();
                    list.add(a);
                    list.add(b);
                    list.add(c);
                    result.add(list);
                    //去重左右指针重复的元素
                    while (j<k && nums[j] == nums[j+1]) {
                        j++;
                    }
                    while (j<k && nums[k] == nums[k-1]) {
                        k--;
                    }

                    j++;
                    k--;
                }else if ((a+b+c) < 0){ // 左指针右移
                    j++;
                }else { //右指针左移
                    k--;
                }
            }
        }
        return result;
    }
}
```

（8）18.四数之和-P

参照三数之和，双指针，剪枝

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        //特殊条件判断
        if (nums == null || nums.length < 4) {
            return Collections.EMPTY_LIST;
        }
        //排序数组
        Arrays.sort(nums);

        //遍历
        List<List<Integer>> result = new ArrayList();

        for (int i=0;i<nums.length - 3;i++) {
            //前后重复则跳过
            if (i>0 && nums[i] == nums[i-1]) {
                continue;
            }
            //判断当前循环的最小值是否大于target，如果是则直接退出
            if (nums[i] + nums[i+1] + nums[i+2] +nums[i+3] > target) {
                break;
            }
            //判断当前循环的最大值是否小于target，如果是则直接退出
            if (nums[nums.length-1] + nums[nums.length-2]+nums[nums.length-3]+nums[nums.length-
4] < target) {
                break;
            }

            for (int j = i+1;j<nums.length-2;j++) {
                //跳过前后重复值
                if (j> i+1 && nums[j] == nums[j-1]) {
                    continue;
                }
                //在这层循环中找最小值和最大值分别和target比较
                int min = nums[i] + nums[j] + nums[j+1] +nums[j+2];
                int max = nums[i] + nums[nums.length-1] + nums[nums.length-2] +
nums[nums.length-3];
                if (min > target || max < target) {
                    break;
                }

                //定义双指针
                int a = nums[i];
                int b = nums[j];
                int m = j+1;
                int n = nums.length -1;
                while (m < n) {
                    int c = nums[m];
                    int d = nums[n];
                    if (a+b+c+d ==target) {
                        List<Integer> list = new ArrayList();
                        list.add(a);
                        list.add(b);

                        list.add(c);
```

```
                        list.add(d);
                        result.add(list);
                        //双指针去重
                        while (m<n && nums[m] == nums[m+1]) {
                            m++;
                        }
                        while (m<n && nums[n] == nums[n-1]) {
                            n--;
                        }
                        m++;
                        n--;
                    }else if (a+b+c+d < target) {
                        m++;
                    }else {
                        n--;
                    }
                }
            }
        }

        return result;
    }
}
```

（9）LCP 18. 早餐组合-C

1，暴力破解，简单，但未AC，超时

```
class Solution {
    //暴力破解 未AC 时间超时
    public int breakfastNumber(int[] staple, int[] drinks, int x) {
        //定义结果集数量
        int count = 0;
        for (int i=0;i<staple.length;i++) {

            for (int j=0;j<drinks.length;j++) {

                if (staple[i] + drinks[j] <=x) {
                    count++;
                }
            }
        }
        return count % 1000000007;
    }
}
```

2，双指针夹逼（左右指针），排序数组，剪枝

```java
class Solution {
    //时间复杂度O（m+n），空间复杂度O(1)
    public int breakfastNumber(int[] staple, int[] drinks, int x) {
        //定义结果集数量
        int count = 0;
        //对数组排好序，能利用已排序数组的一些特性
        Arrays.sort(staple);
        Arrays.sort(drinks);

        //定义双指针
        int p1 = 0;
        int p2 = drinks.length-1;
        while (p1 < staple.length && p2 >=0) {
            if (staple[p1] + drinks[p2] <=x) {
                //巧妙的点在这，因为drinks是已排好序的,那既然p2位置的元素能满足条件,那在它之前的都应该满足
                count = (count + p2+1) % 1000000007; //为了防止数太大

                p1++;
            }else {
                p2--;
            }

        }
        return count;
    }
}
```

（10）26. 删除排序数组中的重复项-P

双指针（快慢指针），剪枝

```java
class Solution {
    public int removeDuplicates(int[] nums) {
        //特殊情况判断
        if (nums==null) {
            return 0;
        }
        if (nums.length ==1) {
            return 1;
        }
        //定义不重复元素的个数
        int count = 0;
        //定义不重复元素指针
        int p = 0;
        for (int i=0;i<nums.length;i++) {

            if (i > 0 && nums[i] == nums[i-1]) {
                continue;
            }
            nums[p++] = nums[i];
            count++;
        }
        return count;
    }
}
```

> 类似283、移动零

（11）189. 旋转数组-P

1：暴力解法

```java
class Solution {
    //暴力朴素解法 数组每次向后移动一位，总共操作k次
    public void rotate(int[] nums, int k) {
        //特殊判断
        if (nums==null || nums.length ==1) {
            return;
        }
        for (int i=0;i<k;i++) {
            //从最后一位开始，依次和前一位交换
            for (int j=nums.length-1;j>0;j--) {
                swap(nums, j, j-1);
            }
        }
    }

    private void swap(int[] nums,int i,int j){
        int temp = nums[j];
        nums[j] = nums[i];
        nums[i] = temp;
    }

}
```

2：数学思想+双指针

```
class Solution {
    public void rotate(int[] nums, int k) {
        //规范k的取值
        k = k % nums.length;
        //使用数学思想，基于一个事实：当我们旋转数组 k 次， k%n 个尾部元素会被移动到头部，剩下的元素会被向后移动。在这个方法中，我们首先将所有元素反转。然后反转前 k 个元素，再反转后面 n-kn-k 个元素，就能得到想要的结果
        reverse(nums, 0, nums.length-1);
        reverse(nums, 0, k-1);
        reverse(nums, k, nums.length-1);
    }

    //翻转[start,end]内的数组元素
    private void reverse(int[] nums,int start,int end){
        while (start < end) {
            int temp = nums[end];
            nums[end] = nums[start];
            nums[start] = temp;

            start++;
            end--;
        }
    }
}
```

2、链表

面试题列表：

1：阿里，腾讯，百度，美团点评最近面试题，141. 环形链表

2：字节跳动最近面试题，142. 环形链表 II

3：华为，阿里，腾讯，快手最近面试题，2. 两数相加

4：华为，快手，百度最近面试题，21. 合并两个有序链表

5：字节跳动，谷歌最近面试题，24. 两两交换链表中的节点

6：阿里，猿辅导，百度最近面试题，25. K 个一组翻转链表

（1）141. 环形链表-C

1，双指针（快慢指针）

```java
public class Solution {
    public boolean hasCycle(ListNode head) {
        //特殊判断
        if (head == null || head.next == null) {
            return false;
        }
        ListNode fast = head;
        ListNode slow = head;
        //两个指针分别向后走
        while (true) {
            fast = fast.next.next;
            slow = slow.next;

            if (fast == null || fast.next == null) {
                return false;
            }
            if (fast == slow) {
                break;
            }
        }

        return true;
    }
}
```

2，缓存思想

```java
public class Solution {
    public boolean hasCycle(ListNode head) {
        //利用缓存思想
        Set set = new HashSet();
        while (head != null) {
            if (set.contains(head)) {
                return true;
            }
            set.add(head);
            head = head.next;
        }
        return false;
    }
}
```

（2）142. 环形链表 II-C

跟环形链表一样，用缓存法能解决

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        //缓存法，每次走过一个结点就缓存起来，并且每次遍历时先判断缓存中是否已有该节点了，第一次缓存命中的结点也就是入环结点
        Set cache = new HashSet();
        while (head !=null) {
            if (cache.contains(head)) {
                return head;
            }
            cache.add(head);
            head = head.next;
        }
        return null;
    }
}
```

双指针法，数学证明

```
public class Solution {
    public ListNode detectCycle(ListNode head) {

        //定义快慢指针
        ListNode fast = head;
        ListNode slow = head;
        //让快慢指针第一次相遇
        while (true) {
            if (fast == null || fast.next == null) {
                return null;
            }
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) {
                break;
            }
        }
        //第一次相遇后让fast重新从头节点开始走，fast和slow每次走相同的步数，直到他们相遇，相遇的结点就是环的入口
        fast = head;
        while (fast != slow) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

（3）2. 两数相加-C

双指针（快慢指针），数学加法，事先构造链表头节点，返回其**next**（哨兵）

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        //特殊判断
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        //构建进位
        int carry = 0;
        //构建返回的链表，先把头节点构建出来，这样后续节点构建的时候代码统一，如果不先构建一个头节点的话，后面代码首先需要new一个头节点，然后依次构建next指针，如果事先构造一个头节点的话，后续代码都只需要构建next指针即可，当然这个头节点的值跟最终返回的结果没关系，最终在返回时返回头节点的next节点即可
        ListNode l3 = new ListNode(-1);
        ListNode temp = l3;//记录一下哨兵

        //l1和l2从头节点开始相加构建返回的新节点，l1和l2有可能长度不一样，不够的地方我们用0代替即可
        while (l1 != null || l2 != null) { //保证能把最长的链表走完
            int a = l1 == null ? 0:l1.val;
            int b = l2 == null ? 0:l2.val;
            int c = (a+b+carry) % 10;
            carry = (a+b+carry) /10;

            //构建l3的结点
            l3.next = new ListNode(c);

            //三个链表依次向后移动
            l3 = l3.next;
            if (l1 != null) {
                l1 = l1.next;
            }
            if (l2 != null) {
                l2 = l2.next;
            }

        }

        //如果最后还有进位比如[5] [5] => [0,1]
        if (carry > 0) {
            l3.next = new ListNode(carry);
        }

        return temp.next;
    }
}
```

（4）21. 合并两个有序链表-P

双指针（快慢指针），事先构造链表头节点，返回其**next**（哨兵）

```
public ListNode mergeTwoLists2(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode l3 = new ListNode(-1);
        ListNode result = l3;
        while (l1!=null && l2!=null) {
            if (l1.val < l2.val) {
                l3.next = l1;
                l1 = l1.next;
            }else {
                l3.next = l2;
                l2 = l2.next;
            }
            l3 = l3.next;
        }
        l3.next = l1 == null ? l2:l1;
        return result.next;
    }
```

（5）24. 两两交换链表中的节点-C

哨兵

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        //特殊判断
        if (head == null || head.next == null) {
            return head;
        }

        ListNode pre = new ListNode(-1);
        pre.next = head;
        ListNode temp = head.next;
        while ( head != null && head.next != null) {
                ListNode next =  head.next;
                ListNode next_next = next.next;

                next.next = head;
                head.next = next_next;
                pre.next = next;

                pre = head;
                head = next_next;
        }

        return temp;
    }
}
```

（6）25. K 个一组翻转链表-P

```java
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        //构造pre节点
        ListNode dump = new ListNode(-1);
        dump.next = head;

        //定义双指针
        ListNode pre = dump;
        ListNode end = dump;

        //由end指针去找寻翻转的尾部节点,然后翻转中间这一部分，后续的节点同理
        while (end.next !=null) {
            //end走k步
            for (int i=0;i<k&&end!=null;i++) {
                end = end.next;
            }
            //不够k个翻转了,直接跳出循环
            if (end == null) {
                break;
            }
            //够k个，则开始翻转[pre.next,end]这部分
            ListNode start = pre.next;
            ListNode next = end.next;
            end.next = null;//断开与后面的连接，翻转的时候可以采用翻转单链表的方法来进行
            pre.next = reverse(start);

            start.next = next;//翻转完成后再接上后面的部分

            pre = start;
            end = pre;

        }
        return dump.next;
    }

    //反转单链表
    private ListNode reverse(ListNode head){
        ListNode pre = null;
        ListNode curr = head;
        while (curr!=null) {
            ListNode next = curr.next;
            curr.next = pre;

            pre = curr;
            curr = next;
        }
        return pre;
    }

}
```