第一章 、常见运算

```
//取模运算:余数,可以用作循环
5%2 = 1
5/2 = 2
i++ //输出后再加
++i //加完再输出
```

第二章、常用数据结构与算法

一、数组、链表、跳表

- 原始数组

```
//1、 数据类型 [] 数组名=new 数据类型[ length];
int[] ary = new int[4];//初始化值为0
//2、初始化
int[] ary2 = {0, 1, 2};
//数组扩容,拷贝
int[] newInts = Arrays.copyOf(ary2, 5);//表面上对数组长度进行扩容，实际新开辟一个空间
System.out.println(Arrays.toString(newInts));//[0, 1, 0, 0, 0]
//数组复制
System.arraycopy(ary2, 1, ary, 1, 2);//把ary2中的内容从下标1开始复制到ary中，复制长度2
System.out.println(Arrays.toString(ary));//[0, 1, 2, 0]
```

- ArrayList

```
public boolean add(E e) {
  ensureCapacityInternal(size + 1);  // Increments modCount!!
  elementData[size++] = e;//学习写法：添加到size位置后，size++
  return true;
}
```

- 链表

```
//双向链表
class Node {
    public int key, val;
    public Node next, prev;
    public Node(int k, int v) {
        this.key = k;
        this.val = v;
    }
}
```

- 跳表
  - 升维：增加多级索引Olog(n)；随着增加删除索引索引可能需要重建，空间复杂度O(n)

1、解题模板

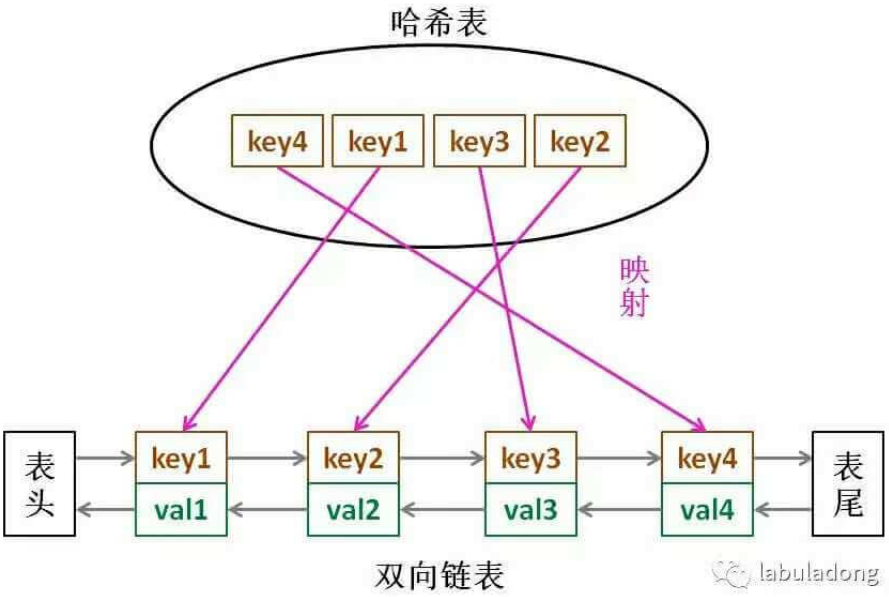
- 遍历

```
/* 基本的单链表节点 */
class ListNode {
    int val;
    ListNode next;
}

void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {
        // 迭代访问 p.val
    }
}

void traverse(ListNode head) {
    // 递归访问 head.val
    traverse(head.next);
}
```

2、力扣刷题

146. LRU 缓存机制

- **题解**：LRU 缓存算法的核心数据结构就是哈希链表，双向链表和哈希表的结合体。这个数据结构长这样：



- **答案**

```
class LRUCache {
    private HashMap<Integer, Node> map;
    private DoubleList cache;
    // 最大容量
    private int cap;

    public LRUCache(int capacity) {
        this.cap = capacity;
        map = new HashMap<>();
        cache = new DoubleList();
    }

    public int get(int key) {
        if (!map.containsKey(key)) return -1;
        int val = map.get(key).val;
        // 利用 put 方法把该数据提前
        put(key, val);
        return val;
    }

    public void put(int key, int val) {
        // 先把新节点 x 做出来
        Node x = new Node(key, val);
        if (map.containsKey(key)) {
            // 删除旧的节点，新的插到头部
            cache.remove(map.get(key));
            cache.addFirst(x);
            // 更新 map 中对应的数据
            map.put(key, x);
```

```
        } else {
            //满了则删除链表最后一个数据
            if (cap == cache.size()) {
                Node last = cache.removeLast();
                map.remove(last.key);
            }
            // 直接添加到头部
            cache.addFirst(x);
            map.put(key, x);
        }
    }

    static class DoubleList {
        //确保链表不为空，头结点为first.next
        private Node first = new Node(0, 0);
        private Node end = new Node(0, 0);
        private int size;

        public DoubleList() {
            first.next = end;
            end.prev = first;
            size = 0;
        }

        // 在链表头部添加节点 x，时间 O(1)
        public void addFirst(Node x) {
            Node temp = first.next;
            first.next = x;//第二个才是插入的头结点
            x.next = temp;
            temp.prev = x;
            x.prev = first;
            size++;
        }
        // 删除链表中的 x 节点（x 一定存在）
        // 由于是双链表且给的是目标 Node 节点，时间 O(1)
        public void remove(Node x) {
            x.next.prev = x.prev;
            x.prev.next = x.next;
            size--;
        }
        // 删除链表中最后一个节点，并返回该节点，时间 O(1)
        public Node removeLast() {
            Node last = end.prev;
            remove(last);
            return last;
        }

        // 返回链表长度，时间 O(1)
        public int size() {
```

```
            return size;
        }
    }

    static class Node {
        public int key, val;
        public Node next, prev;

        public Node(int k, int v) {
            this.key = k;
            this.val = v;
        }
    }
}
```

作者：labuladong 链接：https://leetcode-cn.com/problems/lru-cache/solution/lru-ce-lue-xiang-jie-he-shi-xian-by-labuladong/

142. 环形链表 II

给定一个链表，返回链表开始**入环的第一个节点**。 如果链表无环，则返回 `null`

- 解题思路
  - 这类链表题目一般都是使用双指针法解决的，例如寻找距离尾部第K个节点、寻找环入口、寻找公共尾部入口等。
  - slow再走a = 入口 = head走到入口 = a

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        //快慢指针
        //每次移动两步,有环则一定会在环的 某一个位置 超越慢指针
        ListNode fast = head, slow = head;
        while(true){
            if(fast == null || fast.next == null) return null;
            fast = fast.next.next;
            slow = slow.next;
            if(fast == slow) break;
        }
        // 假如fast == head --> 有环
        fast = head;
        while(fast != slow){
           fast = fast.next;
           slow = slow.next;
        }
        return fast;
    }
}
```

面试题 02.07. 链表相交

给定两个（单向）链表，判定它们是否相交并返回交点

- **解题思路**
  - 双指针(快慢指针，前后指针)

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
  //双指针:a和b以相同的速度走过相同的路程，如果最后一段路程相同则他们必然在共同路程的入口相遇;
  ListNode a = headA;
  ListNode b = headB;
  while(a != b){
    a = a == null ? headB : a.next;
    b = b == null ? headA : b.next;
  }
  return a;
}
```

206. 反转链表

- **解题思路**

  定义两个指针： pre（前） 和 cur(后)；每次让 cur 的 next 指向 per，实现一次局部反转，局部反转完成之后，pre 和 cur 同时往前移动一个位置，循环上述过程，直至 pre到达链表尾部

```
class Solution {
    public ListNode reverseList(ListNode head) {
        //双指针实现
        ListNode per = null;
        ListNode cur = head;
        ListNode temp = null;
        while(cur != null){
            temp = cur.next;
            //改变指向，局部反转
            cur.next = per;
            //指针向前移动
            per = cur;
            cur = temp;
        }
        return per;
    }
}
```

92. 反转链表 II

给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表

```
class Solution {
    public ListNode reverseBetween(ListNode head, int left, int right) {
        //头插法
        //定义一个头节点方便处理
        ListNode perHead = new ListNode(-1);
        perHead.next = head;

        //定义两个指针分别指向起始位置的前一个位置和起始位置
        ListNode per = perHead, cur = perHead.next;
        for(int i = 0; i < left - 1; i++){
            per = per.next;
            cur = cur.next;
        }

        //头插法1 234 5
        for(int i = left; i < right; i++){
            ListNode temp = cur.next;//待插入元素
            cur.next = cur.next.next;//移除待插入元素后指向下一位

            temp.next = per.next;//待插入元素的下一位指向头部元素的下一位
            per.next = temp;//待插入元素插入头部
        }
        return perHead.next;
    }
}
```

141. 环形链表

给定一个链表，判断链表中是否有环

- **解题思路**

  1、快慢指针，有环则一定追上；

  2、Set集合或者数组：存在则一定有环

```
public boolean hasCycle(ListNode head) {
  //快慢指针
  ListNode f = head, s = head;
  while(true){
    if(f == null || f.next == null) return false;
    f = f.next.next;
    s = s.next;
    if(f == s) return true;
  }
}
```

21. 合并两个有序链表

*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**
  - 迭代法：因为有序可以一次遍历两个链表，通过比较，改变指向，其中一个遍历完毕，把另一个指向最后即可

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        //迭代法
        ListNode head = new ListNode(-1);
        //维护一个指针，一直指向链表的最后一个位置
        ListNode per = head;
        while(l1 != null && l2 != null){
            if(l1.val >= l2.val){
                per.next = l2;
                l2 = l2.next;
            }else{
                per.next = l1;
                l1 = l1.next;
            }
            //指针后移一位
            per = per.next;
        }
        //链表末尾指向不为空的链表
        per.next = l1 == null ? l2 : l1;
        return head.next;
}
```

- 递归解法

```java
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        //递归解法，较小的节点指向其余待合并元素
        if(l1 == null) return l2;
        if(l2 == null) return l1;

        if(l1.val > l2.val){
           l2.next = mergeTwoLists(l1, l2.next);
           return l2;
        }else{
           l1.next =  mergeTwoLists(l1.next, l2);
           return l1;
        }
}
```

88. 合并两个有序数组

```java
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1;
        int j = n - 1;
        int k = m + n -1;
        //从后往前
        while(i >= 0 && j >= 0){
           nums1[k--] = nums1[i] > nums2[j] ? nums1[i--] : nums2[j--];
        }
        //把nums数组中从0到j的数据拷贝到m1中
        System.arraycopy(nums2, 0, nums1, 0, j + 1);
    }
}
```

66. 加一

给定一个由 整数 组成的 非空 数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

```java
class Solution {
    public int[] plusOne(int[] digits) {
        int len = digits.length;
        for (int i = len - 1; i >= 0; i--) {
            digits[i] = digits[i] == 9 ? 0 : digits[i] + 1;
            if(digits[i] != 0) {
                return digits;//无进位，结束返回
            }
        }
```

```
            digits = new int[len + 1]; //有多余进位，低位完全为0；
            digits[0] = 1;
            return digits;
        }
    }
```

27. 移除元素

- 快慢指针
    - `fast` 遇到需要去除的元素，则直接跳过，否则就**告诉 `slow` 指针（指向）**，并让 `slow` 前进一步
    - **注意这里和有序数组去重的解法有一个重要不同**，我们这里是先给 `nums[slow]` 赋值然后再给 `slow++`，这样可以保证 `nums[0..slow-1]` 是不包含值为 `val` 的元素的，最后的结果数组长度就是 `slow`

```
class Solution {
    public int removeElement(int[] nums, int val) {
        int fast = 0, slow = 0;
        while (fast < nums.length) {
            if (nums[fast] != val) {
                nums[slow] = nums[fast];
                slow++;
            }
            fast++;
        }
        return slow;
    }
}
```

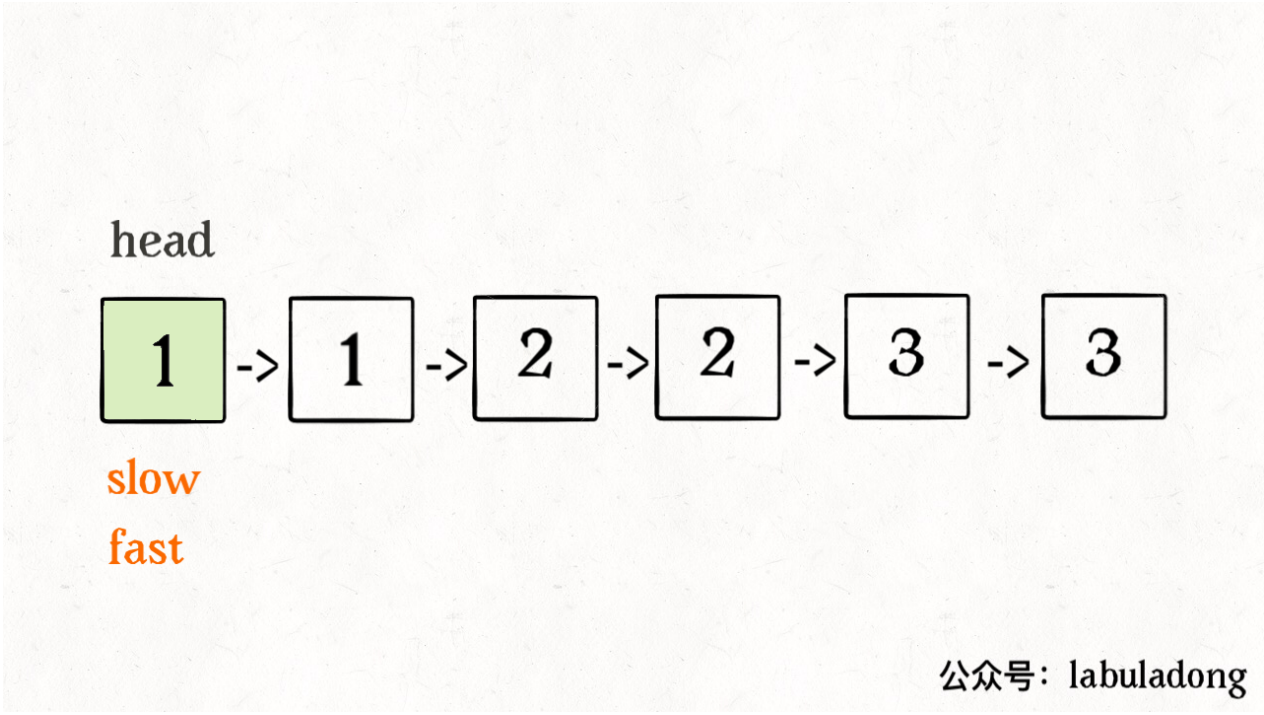
26. 删除有序数组中的重复项

- 使用双指针，因为数组有效，使用前后指针，当前后相同时前指针移动，后指针不动，如果不相同则把前指针移动一位并把元素复制到当前位置；
- 让慢指针 `slow` 走在后面，快指针 `fast` 走在前面探路，找到一个不重复的元素就告诉 `slow` 并让 `slow` 前进一步。这样当 `fast` 指针遍历完整个数组 `nums` 后，**`nums[0..slow]` 就是不重复元素**

```
class Solution {
    public int removeDuplicates(int[] nums) {
          if (nums.length == 0) return 0;
          int slow = 0, fast = 0;
          while (fast < nums.length) {
```

```
                if (nums[fast] != nums[slow]) {
                    slow++;
                    // 维护 nums[0..slow] 无重复
                    nums[slow] = nums[fast];
                }
                fast++;
            }
            // 数组长度为索引 + 1
            return slow + 1;
        }
}
```

83. 删除排序链表中的重复元素




```
class Solution {
  //快慢指针
  ListNode deleteDuplicates(ListNode head) {
      if (head == null) return null;
      ListNode slow = head, fast = head;
      while (fast != null) {
          if (fast.val != slow.val) {
              // nums[slow] = nums[fast];
              slow.next = fast;
              // slow++;
              slow = slow.next;
          }
          // fast++
          fast = fast.next;
      }
```

```
        // 断开与后面重复元素的连接
        slow.next = null;
        return head;
}
```

19. 删除链表的倒数第 N 个结点

- 解题思路：双指针构建出一个为n的步长，然后整体向后进行移动，**后指针下一位**为空时则前指针即为**倒数第n个数**

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        //增加个前指针,避免为空的问题
        ListNode per = new ListNode(0);
        per.next = head;
        //前后指针
        ListNode start = per,end = per;

        //end指针先移动n步，构建出n的距离，滑动窗口思路
        while(n != 0){
            end = end.next;
            n--;
        }
        //前后指针共同移动直到后指针为空
        while(end.next != null){
            end = end.next;
            start = start.next;
        }
        //此时的start即为倒数第n个节点；因为前面加了一个前节点per
        start.next = start.next.next;

        return per.next;
    }
}
```

203. 移除链表元素

给你一个链表的头节点 `head` 和一个整数 `val` ，请你删除链表中所有满足 `Node.val == val` 的节点，并返回 ***新的头节点*** *。*

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode per = new ListNode(-1);
```

```
        per.next = head;
        ListNode cur = per;

        while(cur.next != null){
            if(cur.next.val == val){
                cur.next = cur.next.next;
            }else{
                cur = cur.next;
            }
        }
        return per.next;
    }
}
```

189. 旋转数组

给定一个数组，将数组中的元素向右移动 `k` 个位置，其中 `k` 是非负数。

```
class Solution {
    public void rotate(int[] nums, int k) {
        //1、数组复制； 2、取模运算
        int len = nums.length;
        int[] res = new int[len];
        for(int i = 0; i < len; i++){
            res[(i + k)%len] = nums[i];//精髓
        }
        for(int i = 0; i < len; i++){
            nums[i] = res[i];
        }
    }
}
```

二、树、二叉树、二叉搜索树

1、 解题模板

- 遍历

```
/* 基本的二叉树节点 */
class TreeNode {
    int val;
    TreeNode left, right;
}
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}
```

- N叉树遍历

```
/* 基本的 N 叉树节点 */
class TreeNode {
    int val;
    TreeNode[] children;
}
void traverse(TreeNode root) {
    for (TreeNode child : root.children)
        traverse(child);
}
```

2、力扣刷题

94. 二叉树的中序遍历

*给定一个二叉树的根节点 `root` ，返回它的 **中序** 遍历。*

- 解题思路:递归求解

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
       List<Integer> res = new ArrayList();
       order(root, res);
       return res;
    }

    private void order(TreeNode root, List<Integer> res){
        //中序遍历 做根右
        if(null == root) return;
        order(root.left, res);
        res.add(root.val);
        order(root.right, res);
    }
}
```

590. N 叉树的后序遍历

*给定一个 N 叉树，返回其节点值的 **后序遍历** 。*

N 叉树 在输入中按层序遍历进行序列化表示，每组子节点由空值 `null` 分隔（请参见示例）。

- **解题思路**

```
class Solution {
    public List<Integer> postorder(Node root) {
       List<Integer> res = new ArrayList();
       inOrder(root, res);
       return res;
    }

    private void inOrder(Node node, List<Integer> res){
        if(null == node) return;
        //遍历孩子节点
        for(Node n : node.children){
            inOrder(n, res);
        }
        res.add(node.val);
    }
}
```

429. N 叉树的层序遍历

给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）

- 解题思路

```
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> res = new ArrayList();
        if(root == null) return res;
        inOrder(root, res, 0);
        return res;
    }

    private void inOrder(Node root,List<List<Integer>> res, int level){
        if(root == null) return;
        //遍历当前层
        if(res.size() < level + 1){
            res.add(new ArrayList()); //防止下标越界
        }
        res.get(level).add(root.val);
        for(Node n : root.children){
            inOrder(n, res, level + 1);//注意不能是level++
```

```
        }
    }
}
```

226. 翻转二叉树

- 解题思路

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        //递归终止条件
        if(null == root) return root;
        //当前层逻辑：位置交换
        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;
        //下探一层
        invertTree(root.left);
        invertTree(root.right);
        return root;
    }
}
```

98. 验证二叉搜索树

- 解题思路
    - 采用中序遍历，指针移动

```
class Solution {
    long per = Long.MIN_VALUE;//注意
    public boolean isValidBST(TreeNode root) {
        //根据中序遍历的特点（左，根，右）--》前一个节点小于后一个节点
        if(root == null) return true;
        //左节点
        if(!isValidBST(root.left)){
            return false;
        }
        //根节点处理,左要小于根
        if(per >= root.val){
            return false;
        }
        //记录前节点
        per = root.val;
        //右节点
        return isValidBST(root.right);
```

```
    }
}
```

104. 二叉树的最大深度

- 解题思路：DFS深度优先搜索
  - 节点为空时说明高度为 0，所以返回 0；节点不为空时则分别求左右子树的高度的最大值，同时加1表示当前节点的高度，返回该数值

```
class Solution {
    //深度优先搜索
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
```

110. 平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

- 解题思路

```
class Solution {
    public boolean isBalanced(TreeNode root) {
        //分别计算左右子树高度；
        return treeLevel(root) != -1;
    }

    private int treeLevel(TreeNode tree){
        //-1 不是平衡二叉树
        if(null == tree) return 0;

        int l = treeLevel(tree.left);
        if(l == -1) return -1;
        int r = treeLevel(tree.right);
        if(r == -1) return -1;

        return Math.abs(l - r) < 2 ?  Math.max(l,r) + 1 : -1;
    }

}
```

```
//用自底向上
```

```
public class BalancedBinaryTree {
    boolean res = true;

    public boolean isBalanced(TreeNode root) {
        helper(root);
        return res;
    }

    private int helper(TreeNode root) {
        if (root == null) return 0;
        int left = helper(root.left) + 1;
        int right = helper(root.right) + 1;
        if (Math.abs(right - left) > 1) res = false;
        return Math.max(left, right);
    }
}
```

三、栈、队列

1、解题模板

- 栈Stack：FILO
  - 实现类：Stack

```
public class Stack<E> extends Vector<E> {
    public E push(E item);//入栈
    public synchronized E pop();//出栈
    public synchronized E peek(); //获取栈顶元素
}
```

- 队列Queue：先进先出 FIFO
  - 实现类：LinkedList, ArrayDeque，LinkedBlockingQueue

```
public interface Queue<E> extends Collection<E> {//继承自collection
  boolean add(E e);//添加元素，添加到队头

  E remove();//移除元素，不存在则抛出异常
  E poll();//从队尾，移除元素，不存在则返回空

  E peek();//从队尾，检索但不移除元素，不存在则返回空
}
```

- 优先队列PriorityQueue

- 双端队列Deque
  - 实现类：LinkedList, ArrayDeque，LinkedBlockingDeque(阻塞队列)

```
public interface Deque<E> extends Queue<E> {
  void addFirst(E e);
  void addLast(E e);

  //以下四组会抛出异常
  E removeFirst();
  E removeLast();
  E getFirst();
  E getLast();

  //以下四组不会抛出异常
  E pollFirst();
  E pollLast();
  E peekFirst();
  E peekLast();
}
```

2、力扣刷题

四、哈希表、映射、集合

1、解题模板

2、力扣刷题

五、堆、二叉堆、图

1、解题模板

- Top K 问题有两种不同的解法，一种解法使用堆（优先队列），另一种解法使用类似快速排序的分治法
  - 堆，时间复杂度 O(nlog k)$O(n\log k)$

- 快排变形，（平均）时间复杂度 O(n)$O(n)$

2、力扣刷题

[剑指 Offer 40. 最小的k个数]

```
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
         //大顶堆实现,堆维护k个元素，插入元素小于堆顶元素则替换
         if(k == 0 || arr.length < k){
             return new int[0];
         }
        Queue<Integer> q = new PriorityQueue<>((v1,v2) -> v2 - v1);
        for(int i = 0; i < arr.length; i++){
            if(q.size() < k){
                q.offer(arr[i]);
            }else{
                //替换
                if(q.peek() > arr[i]){
                    q.poll();
                    q.add(arr[i]);
                }
            }
        }
        int[] res = new int[k];
        for (int i = 0; i < k; i++) {
            res[i] = q.poll();
        }
        return res;
    }
}
```

第三章 常用算法

一、递归

二、分治、回溯

1、解题模板

- 分治算法，可以认为是一种**算法思想**，通过将原问题分解成小规模的子问题，然后根据子问题的结果构造出原问题的答案
- 深度优先遍历、递归、栈，它们三者的关系，我个人以为它们背后统一的逻辑都是「**后进先出**」

```
//分治排序
void sort(int[] nums, int l, int r) {
    int mid = (l + r) / 2;
    /****** 分 ******/
    // 对数组的两部分分别排序
    sort(nums, l, mid);
    sort(nums, mid + 1, r);
    /****** 治 ******/
    // 合并两个排好序的子数组
    merge(nums, l, mid, r);
}
```

- 回溯算法就一种简单粗暴的算法技巧，说白了就是一个暴力穷举算法，归去来兮的感觉;
  - 回溯算法」强调了「深度优先遍历」思想的用途，用一个 不断变化 的变量，在尝试各种可能的过程中，搜索需要的结果。**强调了 回退 操作对于搜索的合理性**。而「深度优先遍历」强调一种遍历的思想，与之对应的遍历思想是「广度优先遍历
  - DFS 是一个劲的往某一个方向搜索，而回溯算法建立在 DFS 基础之上的，但不同的是在搜索过程中，达到结束条件后，恢复状态，回溯上一层，再次搜索。因此回溯算法与 DFS 的区别就是**有无状态重置**

2、力扣刷题

22. 括号生成

*数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。*

- 回溯求解

50. Pow(x, n)

```
class Solution {
 public double myPow(double x, int n) {
        long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        return fastPow(x, N);
    }

 public double fastPow(double x, long n) {
        //1、分治
        if (n == 0) return 1.0;
        double subRes = fastPow(x, n >> 1);
```

```
        return n % 2 == 0 ? subRes * subRes : subRes * subRes * x;
    }
}
```

78. 子集

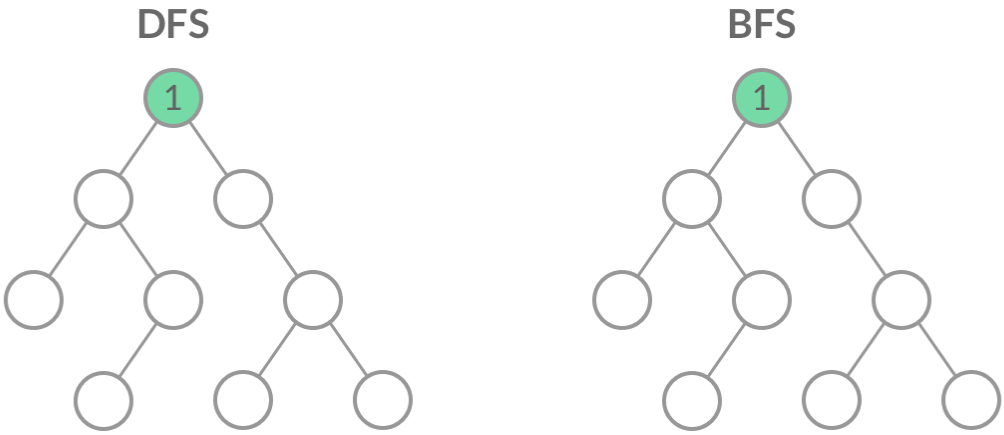
```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> res = new ArrayList();
        if(nums == null) return res;
        dfs(res, nums, new ArrayList(), 0);
        return res;
    }

    private void dfs(List<List<Integer>> res, int[] nums, List<Integer> list,
int index){
        //递归终止条件
        if(index == nums.length){
            res.add(new ArrayList(list));
            return;
        }
        //1 12 13 123
        dfs(res, nums, list, index + 1);
        list.add(nums[index]);//出栈

        // 2
        dfs(res, nums, list, index + 1);

        //reverse the current state
        list.remove(list.size() - 1);
    }
}
```

三、深度优先和广度优先搜索

1、解题模板

- DFS 遍历使用**递归**：
    - 递归的方式隐含地使用了系统的 **栈**，我们不需要自己维护一个数据结构

```
void dfs(TreeNode root) {
    if (root == null) {
        return;
    }
    dfs(root.left);
    dfs(root.right);
}
```



- BFS 遍历使用**队列**数据结构：
    - BFS 的使用场景总结：层序遍历、最短路径问题

```
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll(); // Java 的 pop 写作 poll()
        if (node.left != null) {
            queue.add(node.left);
        }
        if (node.right != null) {
            queue.add(node.right);
        }
    }
}

// 二叉树的层序遍历
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
      //在每一层遍历开始前，先记录队列中的结点数量 nn（这一层的结点数量），然后一口气处理完这一层的结点
        int n = queue.size();
        for (int i = 0; i < n; i++) {
            // 变量 i 无实际意义，只是为了循环 n 次
```

```
            TreeNode node = queue.poll();
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
    }
}
```

2、力扣刷题

102. 二叉树的层序遍历

- 递归实现

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList();
        return inOrder(root, 0, res);
    }

    private List<List<Integer>> inOrder(TreeNode root, int level,
List<List<Integer>> res){
        if (null == root) return res;

        if(res.size() <= level){
            res.add(new ArrayList());
        }

        res.get(level).add(root.val);
        inOrder(root.left, level + 1, res);
        inOrder(root.right, level +  1, res);
        return res;
    }
}
```

- BFS

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();

    Queue<TreeNode> queue = new ArrayDeque<>();
    if (root != null) {
        queue.add(root);
    }
    while (!queue.isEmpty()) {
        int n = queue.size();//记录一层的节点数量，然后一次性遍历完成
```

```
            List<Integer> level = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            res.add(level);
        }

        return res;
    }
```

作者：nettee 链接：https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/bfs-de-shi-yong-chang-jing-zong-jie-ceng-xu-bian-l/

四、贪心算法

1、解题模板

2、力扣刷题

455. 分发饼干

```
class Solution {
    public int findContentChildren(int[] g, int[] s) {
        //贪心算法，排序，小饼干满足小胃口
        Arrays.sort(g);
        Arrays.sort(s);
        int i = 0;
        int j = 0;
        while(i < g.length && j < s.length){
            if(g[i] <= s[j]){
                i++;
            };
            j++;
        }
        return i;
    }
```

```
}
```

五、二分查找

六、动态规划

1、解题模板

- 动态规划问题的一般形式就是**穷举求最值**
- 态规划问题一定会**具备「最优子结构」**

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 求最值(选择1, 选择2...)
```

2、力扣刷题

剑指 Offer 42. 连续子数组的最大和

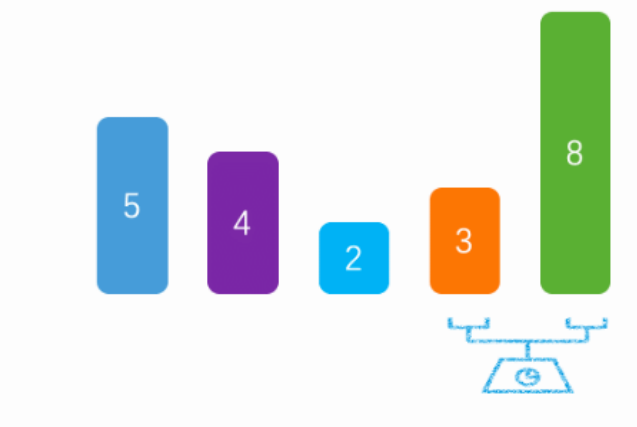
- 解题思路:
    - 设动态规划列表 dp ， dp[i]代表以元素 nums[i]为结尾的连续子数组最大和

```
class Solution {
    public int maxSubArray(int[] nums) {
        //动态规划
        int res = nums[0];
        for(int i = 1; i < nums.length; i++){
            nums[i] += Math.max(nums[i - 1], 0);
            res = Math.max(nums[i], res);
        }
        return res;
    }
}
```

七、排序算法

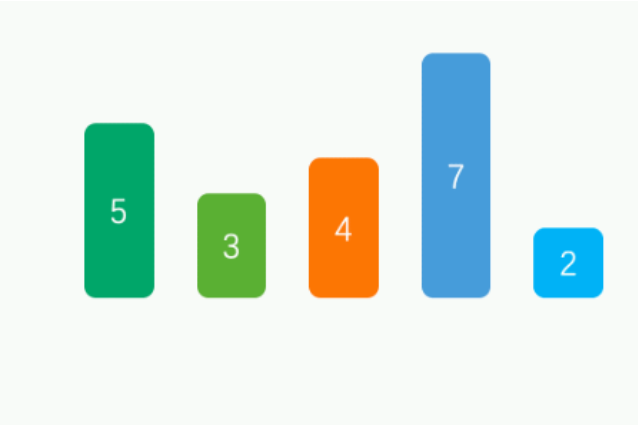
1、解题模板

- **冒泡排序**
  - 每次遍历把大的放到最后



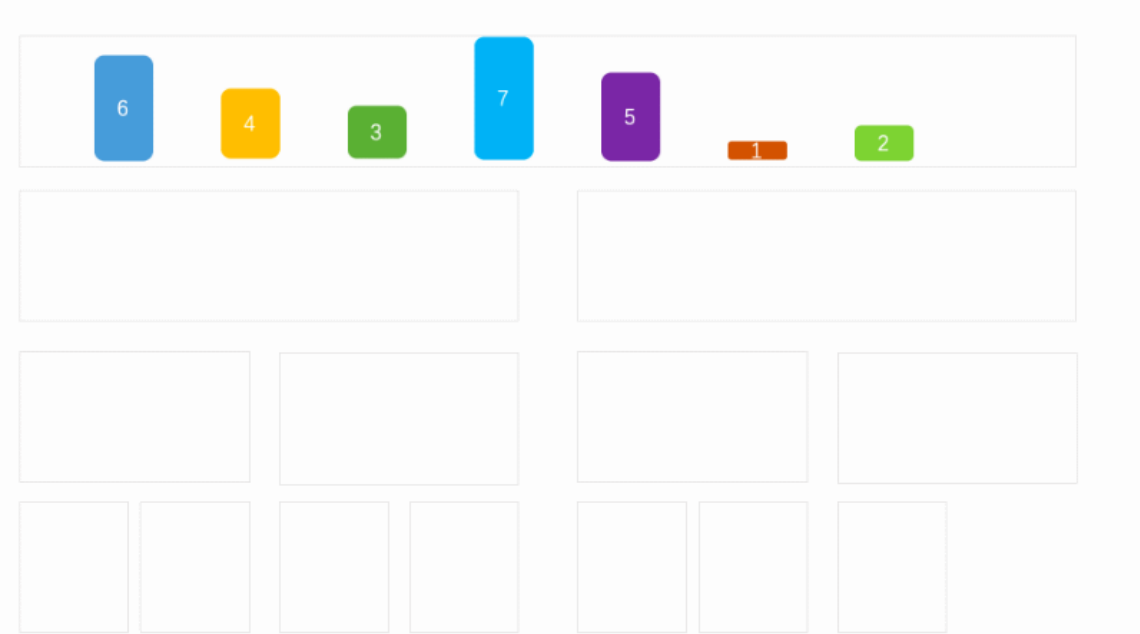

```
private static void mpsort(int[] a) {
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length - i - 1; j++) {
                //内层循环，升序（如果前一个值比后一个值大，则交换）
                if (a[j] > a[j + 1]) {
                    swap(a, j, j + 1);
                }
            }
        }
}
private static void swap(int[] a, int j, int i) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

- 插入排序
  - 第二层从后往前比较，小的往前插入，**大则说明插入完毕退出循环**



```
public static void main(String[] args) {
        int arr[] = {7, 5, 3, 2, 4};
        //插入排序
        for (int i = 1; i < arr.length; i++) {
            //外层循环，从第二个开始比较
            for (int j = i; j > 0; j--) {
                //内存循环，与前面排好序的数据比较，如果后面的数据小于前面的则交换
                if (arr[j] < arr[j - 1]) {
                    int temp = arr[j - 1];
                    arr[j - 1] = arr[j];
                    arr[j] = temp;
                } else {
                    //如果不小于，说明插入完毕，退出内层循环
                    break;
                }
            }
        }
    }
```

- 归并排序
  - 对半拆成最小单位，然后将两半数据合并成一个有序的列表




```
/**
   * 归并排序
   * 分治思想
   */
private static void mergeSort(int[] arr) {
  int[] temp = new int[arr.length];
  subSort(arr, temp, 0,arr.length -1);
```

```
}

private static void subSort(int[] arr, int[] temp, int left, int right) {
  if (left < right){
    int mid = (left + right) >> 1;
    //左边排序
    subSort(arr, temp, left, mid);
    //右边排序
    subSort(arr, temp, mid + 1, right);
    //合并
    merge(arr, temp, left, mid, right);
  }

}

/**
  * 合并两个有序数组
  */
private static void merge(int[] arr, int[] temp, int left, int mid, int right)
{
  //左序列指针
  int i = left;
  //右序列指针
  int j = mid + 1;
  int k = 0;

  while (i <= mid && j <= right){
    temp[k++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
  }
  //把左边填入,只会有一个越界
  while (i <= mid){
    temp[k++] = arr[i++];
  }
  //把右边填入
  while (j <= right){
    temp[k++] = arr[j++];
  }
  k = 0;
  //将temp中的元素全部拷贝到原数组中
  while(left <= right){
    arr[left++] = temp[k++];
  }
}
```

2、力扣刷题

八、字符串算法

第五章、其他相关知识

一、多线程

1、常用思路

- Volatile 可见性，自旋锁等待；（类似于等待通知）
- 锁资源:synchronized + wait/notify; lock + condition + await/singal
- CountDownLatch/CyclicBarrier(循环场景)/Semahphore
- 阻塞队列BlockingQueue: put/take

2、力扣刷题

1114. 按序打印

我们使用线程等待的方式实现执行屏障，使用释放线程等待的方式实现屏障消除

- 使用synchronized + wait/notify 实现
    - 锁升级：偏向锁 -> 轻量级锁（自旋锁）->重量级锁 （MarkWord锁标志位）
        - 偏向锁：只有一个线程，无资源竞争，打上线程标签（可重入，偏向于这一个线程）
        - 轻量级锁：大于2个线程产生资源竞争，CAS机制，自旋等待，用户态层面，消耗CPU；（自适应自旋，自旋次数大于10或者竞争线程大于多少，JVM优化选择）
        - 重量级锁：需要向操作系统内核申请，开销大，Lock机制，等待，但不浪费CPU

```
class Foo {
    private boolean firstFinish = false;
    private boolean secondFinish = false;
    Object lock = new Object();

    public Foo() {
    }

    public void first(Runnable printFirst) throws InterruptedException {
        synchronized(lock){
                printFirst.run();
                firstFinish = true;
                //通知所有在等待lock的线程
                lock.notifyAll();
```

```
        }
    }

    public void second(Runnable printSecond) throws InterruptedException {
           synchronized(lock){
               //当1还未执行结束，则自旋等待,防止出现中途跳出
               while(!firstFinish){
                   lock.wait();
               }
             while()
                printSecond.run();
                secondFinish = true;
                lock.notifyAll();
            }
    }

    public void third(Runnable printThird) throws InterruptedException {
           synchronized(lock){
               //当2还未执行结束则自旋等待
               while(!secondFinish){
                   lock.wait();
               }
               printThird.run();
            }
    }
}
```

- volatile实现
    - 保证可见性，通知其他线程重新从内存中加载缓存数据
    - 禁止指令重排：字节码层面加标识，jvm层面内存屏障，操作系统Lock

```
class Foo {
    //保证可见性及禁止指令重排序
    volatile int count=1;
    public Foo() {
    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        count++;
    }

    public void second(Runnable printSecond) throws InterruptedException {
      //自旋，volatile更新数据后会通知到其他线程获取最新的值
      while (count!=2);
        printSecond.run();
        count++;
    }
```

```
    public void third(Runnable printThird) throws InterruptedException {
      //自旋，volatile更新数据后会通知到其他线程获取最新的值
      while (count!=3);
        printThird.run();
    }
}
```

- Lock+Condition+await()/signall()实现

```
class Foo {
    int num;
    Lock lock;
    //精确的通知和唤醒线程
    Condition condition1, condition2, condition3;

    public Foo() {
        num = 1;
        lock = new ReentrantLock();
        condition1 = lock.newCondition();
        condition2 = lock.newCondition();
        condition3 = lock.newCondition();
    }

    public void first(Runnable printFirst) throws InterruptedException {
        lock.lock();
        try {
            //自旋等待
            while (num != 1) {
                condition1.await();
            }
            printFirst.run();
            num = 2;
            condition2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void second(Runnable printSecond) throws InterruptedException {
        lock.lock();
        try {
            while (num != 2) {
                condition2.await();
            }
            printSecond.run();
            num = 3;
```

```
            condition3.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void third(Runnable printThird) throws InterruptedException {
        lock.lock();
        try {
            while (num != 3) {
                condition3.await();
            }
            printThird.run();
            num = 1;
            condition1.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

- CountDownLatch(减计数器)实现
    - 一个线程等待一组线程执行完再执行，等待线程调用await()方法如果计数器未到达0则一直等待，其余线程执行完后调用countDown()方法会把计数器减一；所有需要等待的线程执行完则计数器为0，等待的线程开始执行；

```
lass Foo {
    CountDownLatch a;
    CountDownLatch b;

    public Foo() {
        //等待一个线程执行完
        a = new CountDownLatch(1);
        b = new CountDownLatch(1);
    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        a.countDown();//执行完-1
    }

    public void second(Runnable printSecond) throws InterruptedException {
        a.await();
        printSecond.run();
```

```
        b.countDown();
    }

    public void third(Runnable printThird) throws InterruptedException {
        b.await();
        printThird.run();
    }
}
```

- Semaphore(信号量)
  - Semaphore与CountDownLatch相似，不同的地方在于Semaphore的值被获取到后是可以释放的，并不像CountDownLatch那样一直减到底
  - 获得Semaphore的线程处理完它的逻辑之后，你就可以调用它的Release()函数将它的**计数器重新加1**，这样其它被阻塞的线程就可以得到调用了

```
class Foo {
    private Semaphore sa;
    private Semaphore sb;
    public Foo() {
        sa = new Semaphore(0);//等待first执行完后再+许可
        sb = new Semaphore(0);
    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        sa.release();//给second加许可，释放一个sa的信号量
    }

    public void second(Runnable printSecond) throws InterruptedException {
        sa.acquire();
        printSecond.run();
        sb.release();//给third加许可
    }

    public void third(Runnable printThird) throws InterruptedException {
        sb.acquire();
        printThird.run();
    }
}
```

- 阻塞队列

```
class Foo {
    BlockingQueue<String> blockingQueue12, blockingQueue23;

    public Foo() {
        //同步队列,没有容量，进去一个元素，必须等待取出来以后，才能再往里面放一个元素
        blockingQueue12 = new SynchronousQueue<>();
```

```
        blockingQueue23 = new SynchronousQueue<>();
    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        blockingQueue12.put("stop");
    }

    public void second(Runnable printSecond) throws InterruptedException {
        blockingQueue12.take();
        printSecond.run();
        blockingQueue23.put("stop");
    }

    public void third(Runnable printThird) throws InterruptedException {
        blockingQueue23.take();
        printThird.run();
    }
}
```

1115. 交替打印FooBar

- Semaphore 在该场景下有点类似红绿灯交替变换的情境，因此信号量成了首选思路：

```
class FooBar {
    private int n;
    public FooBar(int n) {
        this.n = n;
    }
    Semaphore foo = new Semaphore(1);
    Semaphore bar = new Semaphore(0);

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.acquire();
            printFoo.run();
            bar.release();
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.acquire();
            printBar.run();
            foo.release();
        }
    }
```

```
}
```

- Lock（公平锁） 公平锁也是实现交替执行一个不错的选择：

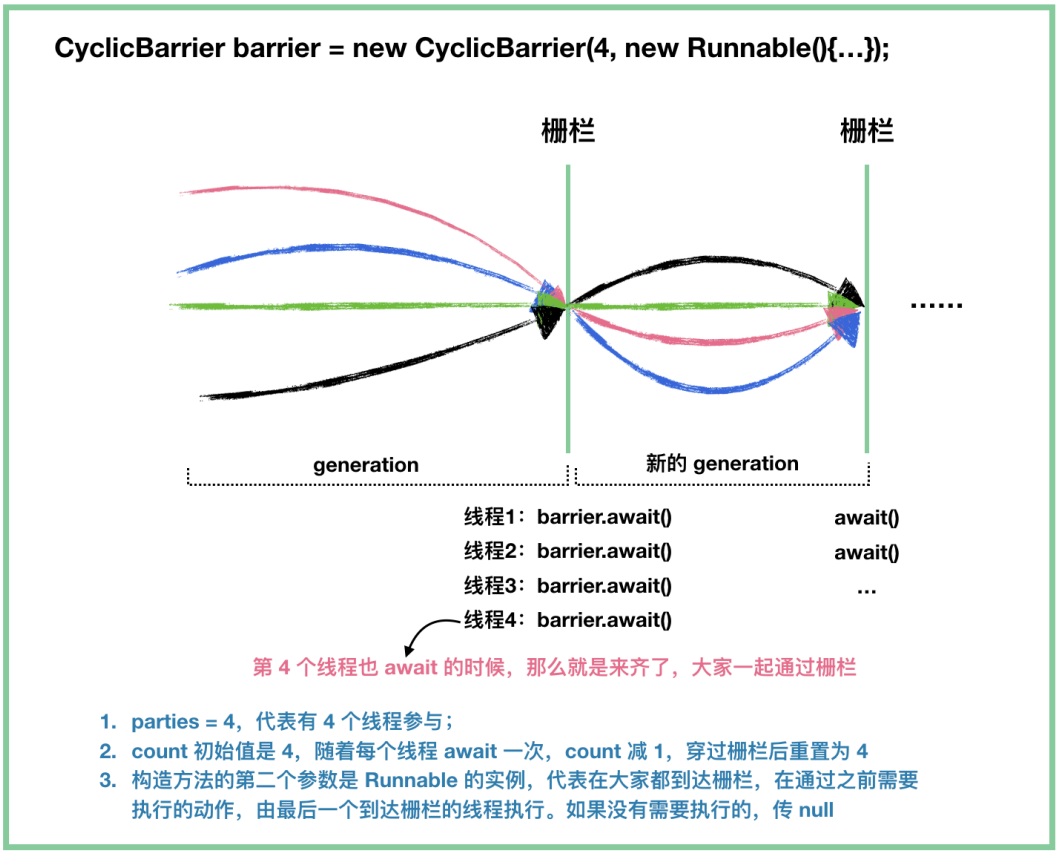
```
class FooBar {
    private int n;
    public FooBar(int n) {
        this.n = n;
    }

    Lock lock = new ReentrantLock(true);
    volatile boolean permitFoo = true;

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; ) {
            lock.lock();
            try {
              if(permitFoo) {
                  printFoo.run();
                    i++;
                    permitFoo = false;
              }
            }finally {
              lock.unlock();
            }
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; ) {
            lock.lock();
            try {
              if(!permitFoo) {
                  printBar.run();
                  i++;
                  permitFoo = true;
              }
            }finally {
              lock.unlock();
            }
        }
    }
 }
```

- 无锁，volatile 以上的公平锁方案完全可以改造成无锁方案：

```
class FooBar {
    private int n;
```

```
    public FooBar(int n) {
        this.n = n;
    }

    volatile boolean permitFoo = true;

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; ) {
            if (permitFoo) {
                printFoo.run();
                i++;
                permitFoo = false;//下一次一定是要等待其他线程完成修改
            }
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; ) {
            if (!permitFoo) {
                printBar.run();
                i++;
                permitFoo = true;
            }
        }
    }
}
```

- CyclicBarrier
  - CyclicBarrier 可以有不止一个栅栏，因为它的栅栏（Barrier）可以重复使用（Cyclic）



- 在CyclicBarrier类的内部有一个计数器，每个线程在到达屏障点的时候都会**调用await方法将自己阻塞**，此时计数器会减1，当计数器减为0的时候所有因调用await方法而被阻塞的线程将被唤醒。这就是实现**一组线程相互等待的原理**
- 在场景一中提过，CyclicBarrier更适合用在循环场景中，那么我们来试一下：

```
class FooBar {
    private int n;

    public FooBar(int n) {
        this.n = n;
    }

    CyclicBarrier cb = new CyclicBarrier(2);
    volatile boolean fin = true;

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            while (!fin) ;//自旋等待，必须要加锁否则下一次任然可能是自己抢占了时间片
            printFoo.run();
            fin = false;
            try {
                cb.await();//阻塞自己，等待其他线程到达屏障点-->到达后进入下一次循环；
            } catch (BrokenBarrierException e) {
            }
        }
    }
```

```
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            try {
                cb.await();//阻塞自己，等待其他线程到达屏障点-->到达后执行打印逻辑
            } catch (BrokenBarrierException e) {
            }
            printBar.run();
            fin = true;//类似于唤醒在自旋的线程
        }
    }
}
```

- 阻塞队列BlockingQueue

```
public class FooBar {
    private int n;
    private BlockingQueue<Integer> bar = new LinkedBlockingQueue<>(1);
    private BlockingQueue<Integer> foo = new LinkedBlockingQueue<>(1);
    public FooBar(int n) {
        this.n = n;
    }
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.put(i);//在take前都只能阻塞
            printFoo.run();
            bar.put(i);
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.take();
            printBar.run();
            foo.take();//执行完释放，类似于通知
        }
    }
}
```

1188. 设计有限阻塞队列

- synchronized + wait/notify + 链表

```
class BoundedBlockingQueue {
    //通过链表实现，可以从头结点添加，尾结点删除
    private LinkedList<Integer> list;
```

```
    private int capacity;
    private volatile int size;

    Object lock = new Object();

    public BoundedBlockingQueue(int capacity) {
        this.list = new LinkedList();
        this.capacity = capacity;
        this.size = 0;
    }

    public void enqueue(int element) throws InterruptedException {
        synchronized(lock){
            while(size + 1 > capacity) lock.wait();//自旋等待
            size++;
            list.addFirst(element);
            lock.notify();
        }
    }

    public int dequeue() throws InterruptedException {
        synchronized(lock){
           while(size <= 0) lock.wait();//自旋等待
           int res = list.removeLast();
           size--;
           lock.notify();
           return res;
        }
    }

    public int size() {
        return size;
    }
}
```

二、布隆过滤器

三、LRU缓存