Table of Contents *generated with* ***DocToc***

- 题目
- 思路
- 实现

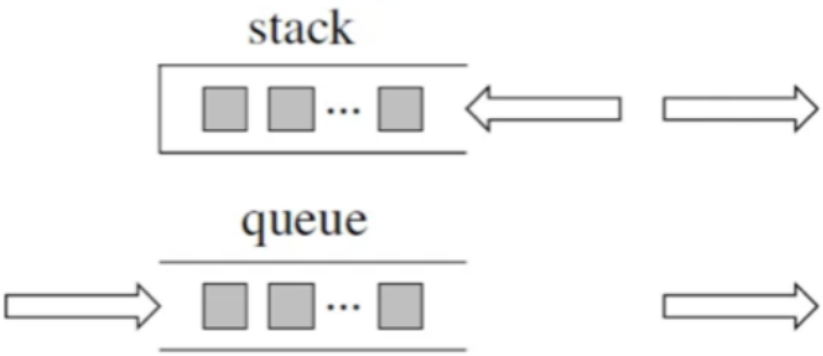
教程地址：**代码随想录 (programmercarl.com)**

算法学习之路 -- 栈和队列

栈和队列理论基础

1）基本原理：

1. 队列：先进先出FIFO
2. 栈：先进后出LIFO

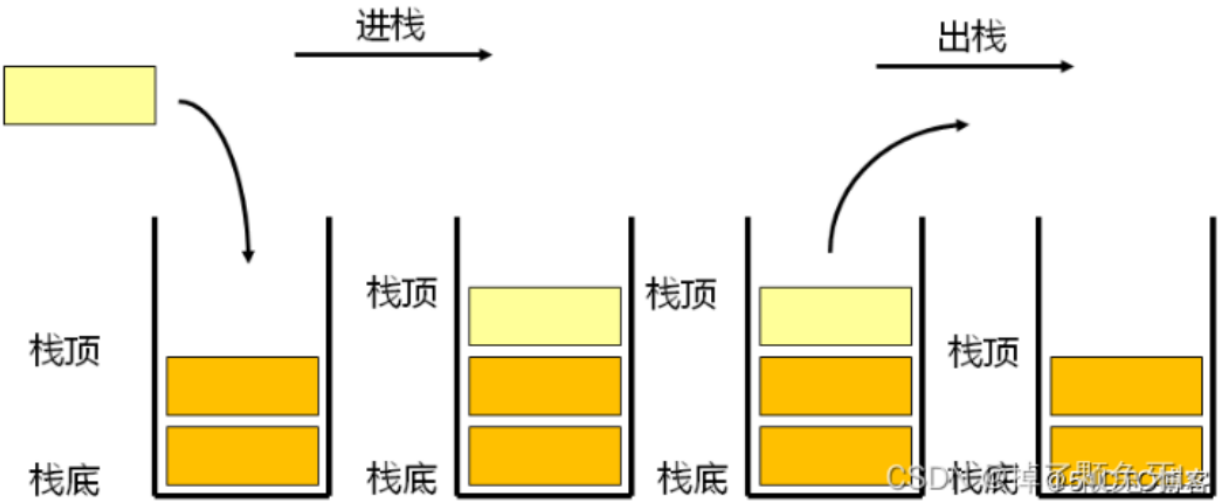


2）Java中的栈

1、概念：

栈是一种仅支持在表尾进行插入和删除操作的线性表，这一端被称为栈顶，另一端被称为栈底。元素入栈指的是 把新元素放到栈顶元素的上面 ，使之成为新的栈顶元素；元素出栈指的是从一个栈删除元素又称作出栈或退栈， 它是把栈顶元素删除掉 ，使其相邻的元素成为新的栈顶元素。栈中的元素遵守 后进先出 （LIFO)的原则.

– 后进先出（Last In First Out）



2、种类

Java中的栈底层实现有两种：

- 基于数组实现：**顺序栈（ArrayList）** => 查找速度快
- 基于链表实现：**链式栈（LinkedList）** => 插入和删除速度快

3、栈的实现

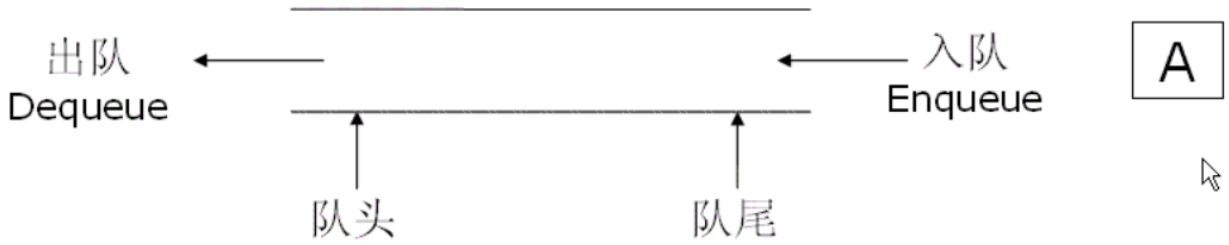
```
// 基于数组实现链表
public class Stack<E> {
    private E[] elementData; // 栈中的元素
    private int size; // 当前栈中元素个数

    public Stack() {
        elementData = (E[]) new Object[10]; // 默认长度为10
    }

    public Stack(int initCap) {
        elementData = (E[]) new Object[initCap]; // 初始长度
    }

    // 入栈
    public void push(E value) {
        // 扩容
        if(size == elementData.length) {
            int oldLength = elementData.length;
            int newLength = oldLength << 1;
            elementData = Arrays.copyOf(elementData, newLength);
        }
        // 在数组尾部添加元素
        elementData[size++] = value;
    }

    // 出栈,返回原来的栈顶元素
```

```
    public E pop () {
        if(getSize() == 0) {
            throw new NoSuchElementException("栈中没有元素！");
        }
        // 得到原来的栈顶元素位置
        E oldVaule = elementData[size - 1];
        size--;
        elementData[size] = null;
        return oldVaule;
    }

    // 查看栈顶元素
    public E peek() {
        if(getSize() == 0) {
            throw new NoSuchElementException("栈中没有元素！");
        }
        return elementData[size - 1];
    }

    // 获取当前栈的长度
    public int getSize() {
        return size;
    }

    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("[");
        for (int i = 0; i < size; i++) {
            stringBuilder.append(elementData[i]);
            if(i != size - 1) {
                stringBuilder.append(",");
            }
        }
        stringBuilder.append("]");
        return stringBuilder.toString();
    }
}
```

3）Java中的队列

1、概念

队列是一种仅支持在表尾进行插入操作、在表头进行删除操作的线性表，插入端称为队尾，删除端称为队首，因整体类似排队的队伍而得名。它满足 先进先出的性质（FIFO），元素入队即将新元素加在队列的尾，元素出队即将队首元素取出，它后一个作为新的队首。



2、种类

- 基于数组实现： `ArrayQueue` **Java源码刨析之ArrayQueue_java_脚本之家 (jb51.net)**
- 基于链表实现： `LinkedQueue` **Java集合详解2：LinkedList和Queue_ITPUB博客**

3、队列的实现

```java
/**
 * 基于链表的队列
 */
public class LinkedQueue{
    private Node head;
    private Node tail;
    private int size;
    private class Node {
        private int data;
        private Node next;

        public Node(int data) {
            this.data = data;
        }
    }

    // 入队
    public void offer(int value) {
        Node node = new Node(value);
        if(head == null) {
            head = tail = node;
        } else {
            tail.next = node;
            tail = node;
        }
        size++;
    }

    // 出队(队首元素出队)
    public int poll() {
        if(size == 0) {
            throw new NoSuchElementException("对列为空！");
        } else {
            int oldValue = head.data;
            Node tempHead = head;
            head = head.next;
            tempHead.next = null;
```

```
            size--;
            return oldValue;
        }
    }

    // 查看队首元素
    public int peek() {
        if(size == 0) {
            throw new NoSuchElementException("对列为空！");
        }
        return head.data;
    }

    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("front[");
        Node node = head;
        while (node != null) {
            stringBuilder.append(node.data);
            if(node.next != null) {
                stringBuilder.append(",");
            }
            node = node.next;
        }
        stringBuilder.append("]tail");
        return stringBuilder.toString();
    }
}
```

LeetCode 典型例题

232、用栈实现队列

题目

题目链接：**232. 用栈实现队列 - 力扣（LeetCode）**

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（`push`、`pop`、`peek`、`empty`）：

实现 `MyQueue` 类：

- `void push(int x)` 将元素 x 推到队列的末尾
- `int pop()` 从队列的开头移除并返回元素
- `int peek()` 返回队列开头的元素
- `boolean empty()` 如果队列为空，返回 `true` ；否则，返回 `false`

说明：

- 你 **只能** 使用标准的栈操作 —— 也就是只有 `push to top`, `peek/pop from top`, `size`, 和 `is empty` 操作是合法的。

- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

示例 1：

```
输入：
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 1, 1, false]

解释：
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

提示：

- `1 ⩽ x ⩽ 9`
- 最多调用 `100` 次 `push`、`pop`、`peek` 和 `empty`
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 `pop` 或者 `peek` 操作

思路

使用栈来模拟队列的行为，仅仅使用一个栈，是一定不行的，所以需要两个栈，**一个输入栈，一个输出栈**，输入栈会把输入顺序颠倒，如果把输入栈的元素逐个弹出放到输出栈，再从输出栈弹出元素的时候，就可以负负得正，实现了先进先出。

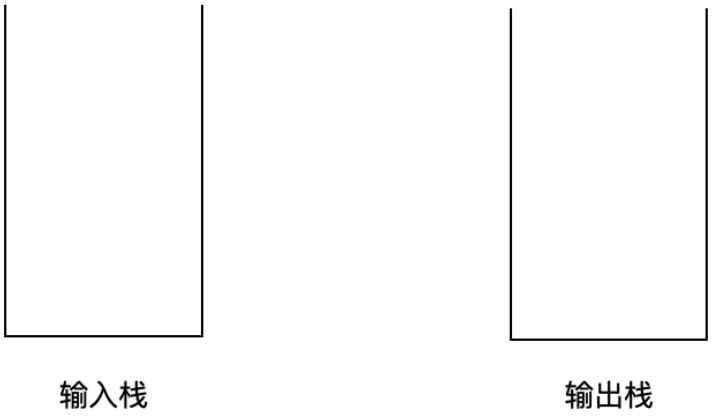
具体做法：

- 可以把一个栈当做「输入栈」，把另一个栈当做「输出栈」。
- 当 push() 新元素的时候，放到「输入栈」的栈顶，记此顺序为「输入序」。
- 当 pop() 元素的时候，是从「输出栈」弹出元素。如果「输出栈」为空，则把「输入栈」的元素逐个 `pop()` 并且 `push()` 到「输出栈」中，这一步会把「输入栈」的栈底元素放到了「输出栈」的栈顶。此时负负得正，从「输出栈」的 `pop()` 元素的顺序与「输入序」相同。

以题目的示例为例

```
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
```

制图：负雪明烛



初始时，两个栈都为空

在push数据的时候，只要数据放进输入栈就好，**但在pop的时候，操作就复杂一些，输出栈如果为空，就把进栈数据全部导入进来（注意是全部导入）**，再从出栈弹出数据，如果输出栈不为空，则直接从出栈弹出数据就可以了。

最后如何判断队列为空呢？**如果进栈和出栈都为空的话，说明模拟的队列为空了。**

实现

```
class MyQueue {

    private Stack<Integer> stackIn;
    private Stack<Integer> stackOut;


    public MyQueue() {
        stackIn = new Stack<>();
        stackOut = new Stack<>();
    }

    public void push(int x) {
        stackIn.push(x);
    }

    public int pop() {
        dumpStackIn();
        return stackOut.pop();
    }
```

```
20
21      public int peek() {
22          dumpStackIn();
23          return stackOut.peek();
24      }
25
26      public boolean empty() {
27          return stackIn.isEmpty() && stackOut.isEmpty();
28      }
29
30
31      private void dumpStackIn() {
32          if(!stackOut.isEmpty()) return;
33          while(!stackIn.isEmpty()) {
34              stackOut.push(stackIn.pop());
35          }
36      }
37  }
38
39  /**
40   * Your MyQueue object will be instantiated and called as such:
41   * MyQueue obj = new MyQueue();
42   * obj.push(x);
43   * int param_2 = obj.pop();
44   * int param_3 = obj.peek();
45   * boolean param_4 = obj.empty();
46   */
```

225、用队列实现栈

题目

题目链接：**225. 用队列实现栈 - 力扣（LeetCode）**

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（`push`、`top`、`pop` 和 `empty`）。

实现 `MyStack` 类：

- `void push(int x)` 将元素 x 压入栈顶。
- `int pop()` 移除并返回栈顶元素。
- `int top()` 返回栈顶元素。
- `boolean empty()` 如果栈是空的，返回 `true` ；否则，返回 `false` 。

注意：

- 你只能使用队列的基本操作 —— 也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。

- 你所使用的语言也许不支持队列。 你可以使用 list （列表）或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。

示例：

```
输入：
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 2, 2, false]

解释：
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False
```

提示：

- `1 ⩽ x ⩽ 9`
- 最多调用 `100` 次 `push`、 `pop`、 `top` 和 `empty`
- 每次调用 `pop` 和 `top` 都保证栈不为空

思路

队列模拟栈，其实一个队列就可以了

1）**两个队列实现**

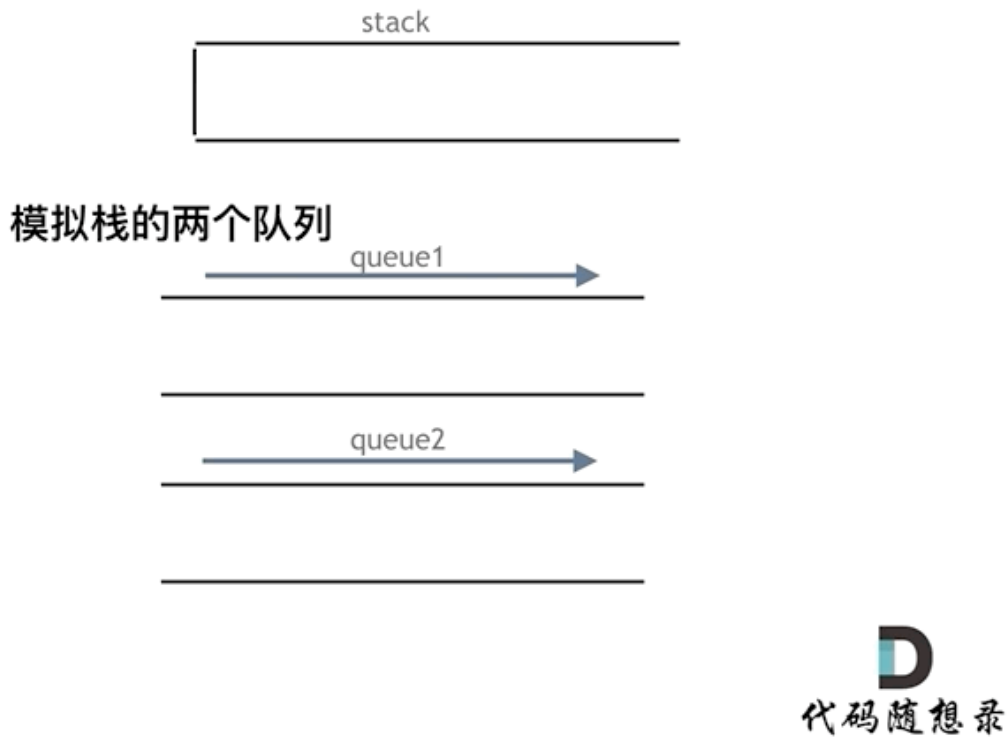
队列是先进先出的规则，把一个队列中的数据导入到另外一个队列中，数据的顺序并没有变，并没有编程先进后出的顺序 => 用队列实现栈和用栈实现队列思路是不一样的，这取决于两个数据结构的性质。

两个队列实现栈，和输入输出没有关系，而是使用另一个队列来备份数据

如下面动画所示，**用两个队列que1和que2实现队列的功能，que2其实完全就是一个备份的作用**，把que1最后面的元素以外的元素都备份到que2，然后弹出最后面的元素，再把其他元素从que2导回que1。

模拟的队列执行语句如下

```
queue.push(1);
queue.push(2);
queue.pop();   // 注意弹出的操作
queue.push(3);
queue.push(4);
queue.pop();  // 注意弹出的操作
queue.pop();
queue.pop();
queue.empty();
```



2）**一个队列实现**

一个队列在模拟栈弹出元素的时候只要将队列头部的元素（除了最后一个元素外） 重新添加到队列尾部，此时再去弹出元素就是栈的顺序了。

实现

1）两个队列实现

Queue

```
class MyStack {

    private Queue<Integer> queue1; // 和栈中保持一样元素的队列
```

```
    private Queue<Integer> queue2; // 辅助队列

    /** Initialize your data structure here. */
    public MyStack() {
        queue1 = new LinkedList<>();
        queue2 = new LinkedList<>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        queue2.offer(x); // 先放在辅助队列中
        while (!queue1.isEmpty()){
            queue2.offer(queue1.poll());
        }
        Queue<Integer> queueTemp;
        queueTemp = queue1;
        queue1 = queue2;
        queue2 = queueTemp; // 最后交换queue1和queue2，将元素都放到queue1中
    }

    /** Removes the element on top of the stack and returns that element.
*/
    public int pop() {
        return queue1.poll(); // 因为queue1中的元素和栈中的保持一致，所以这
个和下面两个的操作只看queue1即可
    }

    /** Get the top element. */
    public int top() {
        return queue1.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return queue1.isEmpty();
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * boolean param_4 = obj.empty();
 */
```

Deque

```
class MyStack {

```

```java
    // Deque 接口继承了 Queue 接口
    // 所以 Queue 中的 add、poll、peek等效于 Deque 中的 addLast、pollFirst、peekFirst
    private Deque<Integer> que1; // 和栈中保持一样元素的队列
    private Deque<Integer> que2; // 辅助队列
    /** Initialize your data structure here. */
    public MyStack() {
        que1 = new ArrayDeque<>();
        que2 = new ArrayDeque<>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        que1.addLast(x);
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        int size = que1.size();
        size--;
        // 将 que1 导入 que2 ，但留下最后一个值
        while (size-- > 0) {
            que2.addLast(que1.peekFirst());
            que1.pollFirst();
        }

        int res = que1.pollFirst();
        // 将 que2 对象的引用赋给了 que1 ，此时 que1，que2 指向同一个队列
        que1 = que2;
        // 如果直接操作 que2，que1 也会受到影响，所以为 que2 分配一个新的空间
        que2 = new ArrayDeque<>();
        return res;
    }

    /** Get the top element. */
    public int top() {
        return que1.peekLast();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return que1.isEmpty();
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
```

```
 * boolean param_4 = obj.empty();
 */
```

2）一个队列实现

Queue

```
class MyStack {

    private Queue<Integer> queue;

    public MyStack() {
        queue = new LinkedList<>();
    }

    //每 offer 一个数（A）进来，都重新排列，把这个数（A）放到队列的队首
    public void push(int x) {
        queue.offer(x);
        int size = queue.size();
        //移动除了 A 的其它数
        while (size-- > 1)
            queue.offer(queue.poll());
    }

    public int pop() {
        return queue.poll();
    }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * boolean param_4 = obj.empty();
 */
```

Deque

```
class MyStack {
```

```
2
3       // Deque 接口继承了 Queue 接口
4       // 所以 Queue 中的 add、poll、peek等效于 Deque 中的 addLast、
    pollFirst、peekFirst
5       private Deque<Integer> que1; // 和栈中保持一样元素的队列
6       /** Initialize your data structure here. */
7       public MyStack() {
8           que1 = new ArrayDeque<>();
9       }
10
11      /** Push element x onto stack. */
12      public void push(int x) {
13          que1.addLast(x);
14      }
15
16      /** Removes the element on top of the stack and returns that element.
    */
17      public int pop() {
18          int size = que1.size();
19          size--;
20          // 将 que1 导入 que2 ，但留下最后一个值
21          while (size-- > 0) {
22              que1.addLast(que1.peekFirst());
23              que1.pollFirst();
24          }
25
26          int res = que1.pollFirst();
27          return res;
28      }
29
30      /** Get the top element. */
31      public int top() {
32          return que1.peekLast();
33      }
34
35      /** Returns whether the stack is empty. */
36      public boolean empty() {
37          return que1.isEmpty();
38      }
39  }
40
41  /**
42   * Your MyStack object will be instantiated and called as such:
43   * MyStack obj = new MyStack();
44   * obj.push(x);
45   * int param_2 = obj.pop();
46   * int param_3 = obj.top();
47   * boolean param_4 = obj.empty();
48   */
```

20、有效的括号

题目

题目链接：**20. 有效的括号 - 力扣（LeetCode）**

给定一个只包括 `'('`，`')'`，`'{'`，`'}'`，`'['`，`']'` 的字符串 `s` ，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1：

```
输入：s = "()"
输出：true
```

示例 2：

```
输入：s = "()[]{}"
输出：true
```

示例 3：

```
输入：s = "(]"
输出：false
```

提示：

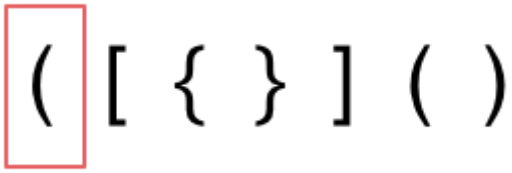
- `1 ≤ s.length ≤ 104`
- `s` 仅由括号 `'()[]{}'` 组成

思路

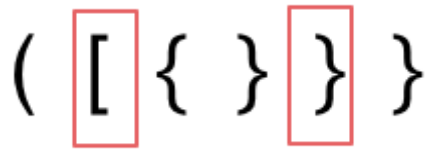
由于栈结构的特殊性，非常适合做对称匹配类的题目

首先分析字符串内的括号不匹配的情况

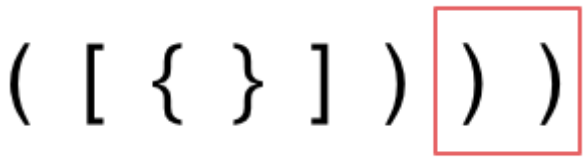
1. 情况一：字符串里的左方向的括号多余了，所以不匹配



2. 情况二：括号没有多余，但是括号的类型匹配不上



3. 情况三：字符串里的右方向的括号多余了，所以不匹配



技巧：在匹配左括号的时候，右括号先入栈，就只需要比较当前元素和栈顶相不相等就可以了，比左括号先入栈代码实现要简单的多了！

实现

```java
class Solution {
    public boolean isValid(String s) {
        Deque<Character> deque = new LinkedList<>();
        for(char ch : s.toCharArray()) {
            if(ch == '(') {
                deque.push(')');
            } else if (ch == '[') {
                deque.push(']');
            } else if (ch == '{') {
                deque.push('}');
            } else if (deque.isEmpty() || deque.peek() != ch) {
                return false;
            } else {
                // 如果是右括号，并且匹配栈顶元素，弹栈
                deque.pop();
            }
        }
        return deque.isEmpty();
    }
}
```

1047、删除字符串中的所有相邻重复项

题目

题目链接：**1047. 删除字符串中的所有相邻重复项 - 力扣（LeetCode）**

给出由小写字母组成的字符串 S ，**重复项删除操作**会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

```
输入："abbaca"
输出："ca"
解释：
例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。
```

提示：

1. `1 ≤ S.length ≤ 20000`
2. `S` 仅由小写英文字母组成。

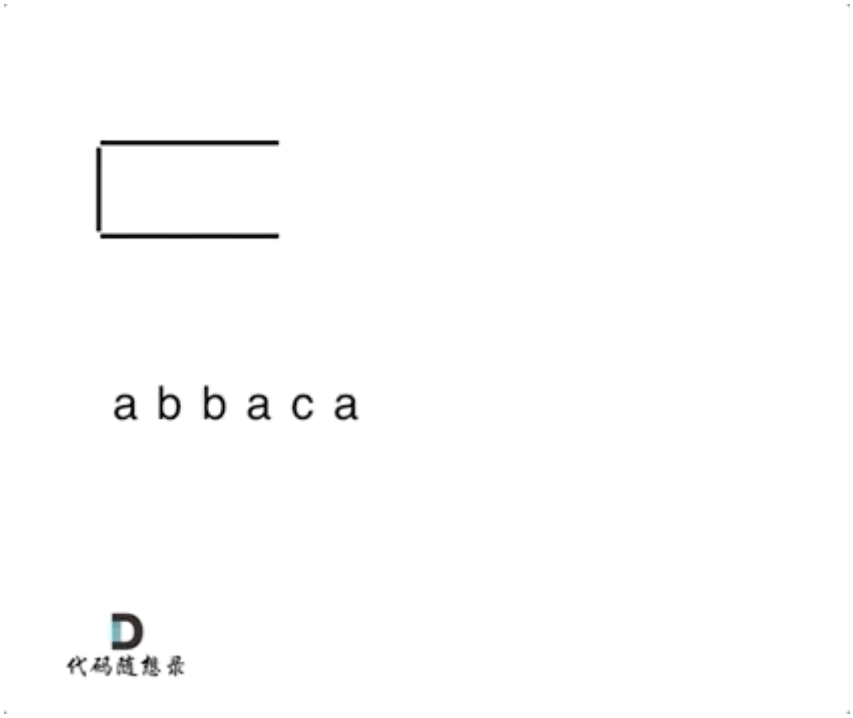
思路

本题也是用栈来解决的经典题目

栈中存放的元素分析：

在删除相邻重复项的时候，其实就是要知道当前遍历的这个元素，我们在前一位是不是遍历过一样数值的元素，那么如何记录前面遍历过的元素呢？

=> 所以就是用栈来存放，那么栈的目的，就是存放遍历过的元素，当遍历当前的这个元素的时候，去栈里看一下我们是不是遍历过相同数值的相邻元素。然后再去做对应的消除操作。



从栈中弹出剩余元素，此时是字符串ac，因为从栈里弹出的元素是倒序的，所以再对字符串进行反转一下，就得到了最终的结果。

实现

使用 `Deque` 作为栈

> `ArrayDeque` 会比 `LinkedList` 在除了删除元素这一点外会快一点
>
> 参考：**https://stackoverflow.com/questions/6163166/why-is-arraydeque-better-than-linkedlist**

```java
class Solution {
    public String removeDuplicates(String s) {
        ArrayDeque<Character> deque = new ArrayDeque<>();
        for(char ch : s.toCharArray()) {
```

```
            if(deque.isEmpty() || deque.peek() != ch) {
                deque.push(ch);
            } else {
                deque.pop();
            }
        }
        StringBuffer str = new StringBuffer();
        while(!deque.isEmpty()) {
            str.append(deque.pop());
        }
        return str.reverse().toString();
    }
}
```

使用字符串作为栈，省去了栈转字符串的操作

```
class Solution {
    public String removeDuplicates(String s) {
        StringBuilder res = new StringBuilder();
        // 记录res的长度
        int top = -1;
        for(char ch : s.toCharArray()) {
            if(top >= 0 && res.charAt(top) == ch) {
                // 当 top > 0,即栈中有字符时，当前字符如果和栈中字符相等，弹出栈顶字符，同时 top--
                res.deleteCharAt(top);
                top--;
            } else {
                // 否则，将该字符 入栈，同时top++
                res.append(ch);
                top++;
            }
        }
        return res.toString();
    }
}
```

双指针解法

```
class Solution {
    public String removeDuplicates(String s) {
        char[] ch = s.toCharArray();
        int fast = 0;
        int slow = 0;
        while(fast < s.length()){
            // 直接用fast指针覆盖slow指针的值
            ch[slow] = ch[fast];
            // 遇到前后相同值的，就跳过，即slow指针后退一步，下次循环就可以直接被覆盖掉了
            if(slow > 0 && ch[slow] == ch[slow - 1]){
```

```
                slow--;
            }else{
                slow++;
            }
            fast++;
        }
        return new String(ch,0,slow);
    }
}
```

150、逆波兰表达式求值

题目

题目链接：**150. 逆波兰表达式求值 - 力扣（LeetCode）**

给你一个字符串数组 `tokens` ，表示一个根据 **逆波兰表示法** 表示的算术表达式。

请你计算该表达式。返回一个表示表达式值的整数。

注意：

- 有效的算符为 `'+'` 、 `'-'` 、 `'*'` 和 `'/'` 。
- 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
- 两个整数之间的除法总是 **向零截断** 。
- 表达式中不含除零运算。
- 输入是一个根据逆波兰表示法表示的算术表达式。
- 答案及所有中间计算结果可以用 **32 位** 整数表示。

示例 1：

```
输入：tokens = ["2","1","+","3","*"]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9
```

示例 2：

```
输入：tokens = ["4","13","5","/","+"]
输出：6
解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6
```

示例 3：

```
输入：tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
输出：22
解释：该算式转化为常见的中缀算术表达式为：
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

提示：

- `1 ≤ tokens.length ≤ 104`
- `tokens[i]` 是一个算符（`"+"`、`"-"`、`"*"` 或 `"/"`），或是在范围 `[-200, 200]` 内的一个整数

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

- 平常使用的算式则是一种中缀表达式，如 `( 1 + 2 ) * ( 3 + 4 )` 。
- 该算式的逆波兰表达式写法为 `( ( 1 2 + ) ( 3 4 + ) * )` 。

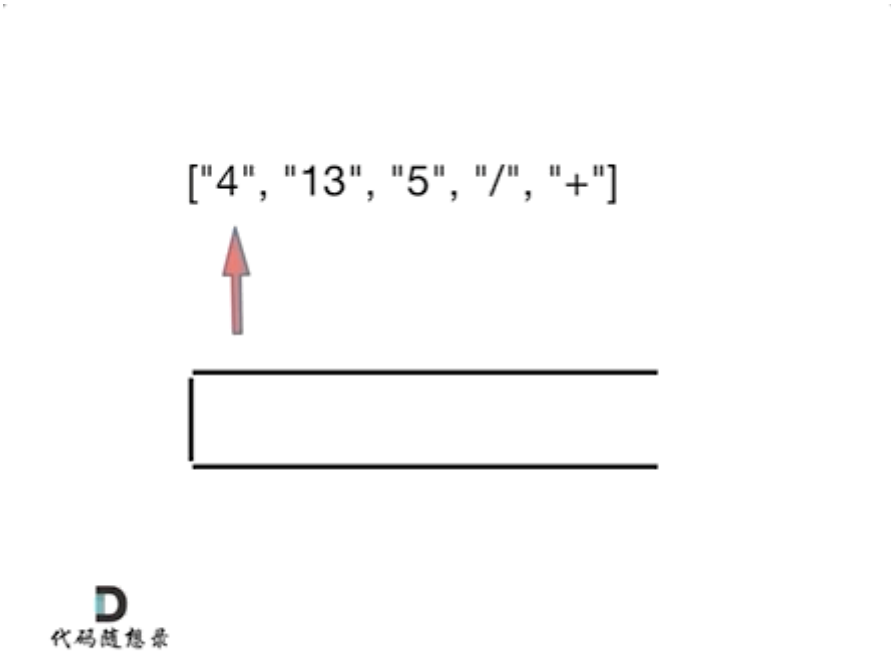
逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 `1 2 + 3 4 + *` 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中

思路

其实逆波兰表达式相当于二叉树中的后序遍历，可以把运算符作为中间结点，按照后序遍历的方式可以画出一颗二叉树

本题中的每一个子表达式要得出一个结果，然后拿这个结果再进行运算 => 就是一个相邻字符串消除的过程，和 LeetCode1047 非常像



实现

```java
class Solution {
    public int evalRPN(String[] tokens) {
        Deque<Integer> stack = new LinkedList<Integer>();
        for(String token : tokens) {
            if("+".equals(token)) {
                stack.push(stack.pop() + stack.pop());
            } else if ("-".equals(token)) {
                stack.push(- stack.pop() + stack.pop());
            } else if ("*".equals(token)) {
                stack.push(stack.pop() * stack.pop());
            } else if ("/".equals(token)) {
                int temp1 = stack.pop();
                int temp2 = stack.pop();
                stack.push(temp2 / temp1);
            } else {
                stack.push(Integer.valueOf(token));
            }
        }
        return stack.pop();
    }
}
```

239、滑动窗口的最大值

题目

题目链接：**239. 滑动窗口最大值 - 力扣（LeetCode）**

给你一个整数数组 `nums` ，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回 *滑动窗口中的最大值*。

示例 1：

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

示例 2：

```
输入：nums = [1], k = 1
输出：[1]
```

提示：

- $1 \leqslant nums.length \leqslant 105$
- $-104 \leqslant nums[i] \leqslant 104$
- $1 \leqslant k \leqslant nums.length$

思路

本题是单调队列的经典题目

难点：如何求一个区间里的最大值

1）方法一：暴力 ❌

暴力方法，遍历一遍的过程中每次从窗口中再找到最大的数值，这样很明显是O(n × k)的算法。

2）方法二：大顶堆（优先级队列） ❌

用一个大顶堆（优先级队列）来存放这个窗口里的k个数字，这样就可以知道最大的最大值是多少了， **但是问题是这个窗口是移动的，而大顶堆每次只能弹出最大值，我们无法移除其他数值，这样就造成大顶堆维护的不是滑动窗口里面的数值了。所以不能用大顶堆。**

此时我们需要一个队列，存放窗口里的元素，然后随着窗口的移动，队列也一进一出，每次移动之后，得到最大值

```
class MyQueue {
    public void pop(int value) {
    }
    public void push(int value) {
    }
    public int front() {
        return que.front();
    }
};
```

每次窗口移动的时候，调用 que.pop(滑动窗口中移除元素的数值)， que.push(滑动窗口添加元素的数值)，然后 que.front() 就返回我们要的最大值。=> 没有现成的数据结构，需要实现

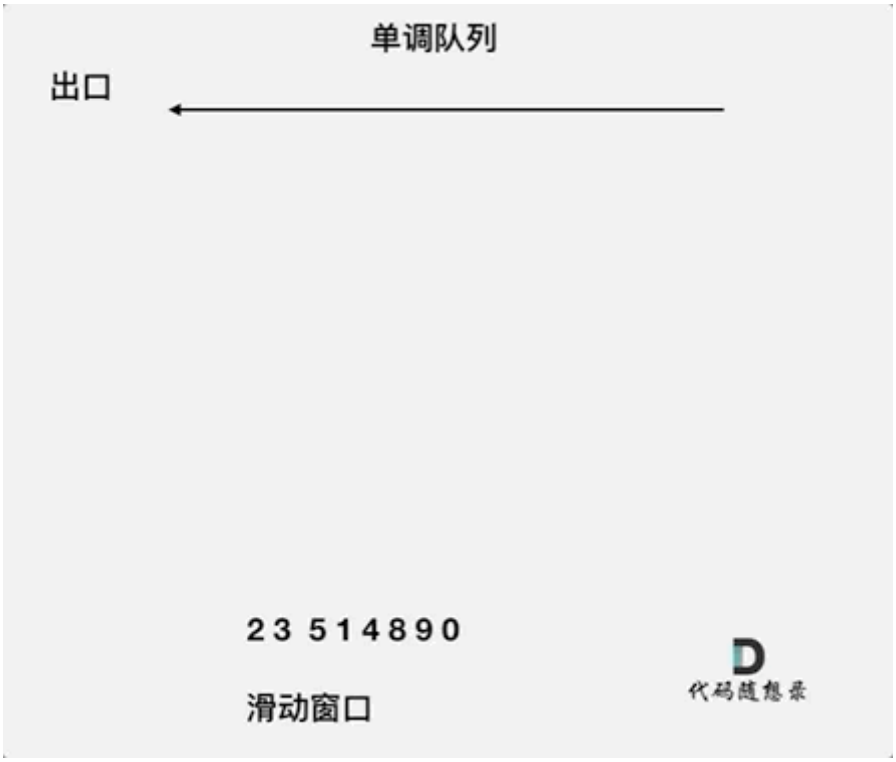
然后再分析一下，队列里的元素一定是要排序的，而且要最大值放在出队口，要不然怎么知道最大值呢。

但如果把窗口里的元素都放进队列里，窗口移动的时候，队列需要弹出元素。

那么问题来了，已经排序之后的队列 怎么能把窗口要移除的元素（这个元素可不一定是最大值）弹出呢。

其实队列没有必要维护窗口里的所有元素，只需要维护有可能成为窗口里最大值的元素就可以了，同时保证队列里的元素数值是由大到小的。

那么这个维护元素单调递减的队列就叫做**单调队列，即单调递减或单调递增的队列。**



对于窗口里的元素 {2, 3, 5, 1 ,4} ，单调队列里只维护 {5, 4} 就够了，保持单调队列里单调递减，此时队列出口元素就是窗口里最大元素。

单调队列里维护着 {5, 4} 怎么配合窗口进行滑动呢？

设计单调队列的时候，pop，和push操作要保持如下规则：

1. pop(value)：如果窗口移除的元素value等于单调队列的出口元素，那么队列弹出元素，否则不用任何操作
2. push(value)：如果push的元素value大于入口元素的数值，那么就将队列入口的元素弹出，直到push元素的数值小于等于队列入口元素的数值为止

保持如上规则，每次窗口移动的时候，只要问 que.front() 就可以返回当前窗口的最大值。

以输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3为例

实现

方法一：自定义队列

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums.length == 1) return nums;

        int len = nums.length - k + 1;
        // 存放结果元素的数组
        int[] res = new int[len];
        int num = 0;
        MyQueue myQueue = new MyQueue();
        // 先将前k个元素放入队列
        for(int i = 0; i < k; i++) {
```

```
            myQueue.add(nums[i]);
        }
        res[num++] = myQueue.peek();
        for(int i = k; i < nums.length; i++) {
            //滑动窗口移除最前面的元素，移除是判断该元素是否放入队列
            myQueue.poll(nums[i - k]);
            //滑动窗口加入最后面的元素
            myQueue.add(nums[i]);
            //记录对应的最大值
            res[num++] = myQueue.peek();
        }
        return res;
    }
    class MyQueue {
        Deque<Integer> deque = new LinkedList<>();
        //弹出元素时，比较当前要弹出的数值是否等于队列出口的数值，如果相等则弹出
        //同时判断队列当前是否为空
        void poll(int val) {
            if(!deque.isEmpty() && val == deque.peek()) {
                deque.poll();
            }
        }
        //添加元素时，如果要添加的元素大于入口处的元素，就将入口元素弹出
        //保证队列元素单调递减
        void add(int val) {
            while(!deque.isEmpty() && val > deque.getLast()) {
                deque.removeLast();
            }
            deque.add(val);
        }
        //队列队顶元素始终为最大值
        int peek() {
            return deque.peek();
        }
    }
}
```

方法二：利用双端队列手动实现单调队列

```
/**
 * 用一个单调队列来存储对应的下标，每当窗口滑动的时候，直接取队列的头部指针对应的值放入结果集即可
 * 单调队列类似 （tail -->） 3 --> 2 --> 1 --> 0 (--> head) (右边为头结点，元素存的是下标)
 */
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        ArrayDeque<Integer> deque = new ArrayDeque<>();
        int len = nums.length - k + 1;
        int[] res = new int[len];
```

```
        int index = 0;
        for(int i = 0; i < nums.length; i++) {
            // 根据题意，i为nums下标，是要在[i - k + 1, i] 中选到最大值，只需要保证两点
            // 1.队列头结点需要在[i - k + 1, i]范围内，不符合则要弹出
            while(!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }
            // 2.既然是单调，就要保证每次放进去的数字要比末尾的都大，否则也弹出
            while(!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }
            deque.offer(i);
            // 因为单调，当i增长到符合第一个k范围的时候，每滑动一步都将队列头节点放入结果就行了
            if(i >= k - 1) {
                res[index++] = nums[deque.peek()];
            }
        }
        return res;
    }
}
```

347、前K个高频元素

题目

题目链接：**347. 前 K 个高频元素 - 力扣（LeetCode）**

给你一个整数数组 `nums` 和一个整数 `k` ，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

提示:

- $1 \leqslant nums.length \leqslant 105$
- k 的取值范围是 $[1, 数组中不相同的元素的个数]$
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的

思路

本题涉及三个内容

- 统计元素出现频率
- 对频率排序
- 找出前K个高频元素

统计元素出现频率：

```
1  Map<Integer,Integer> occurrences = new HashMap<>();
```

对频率排序：**优先队列**

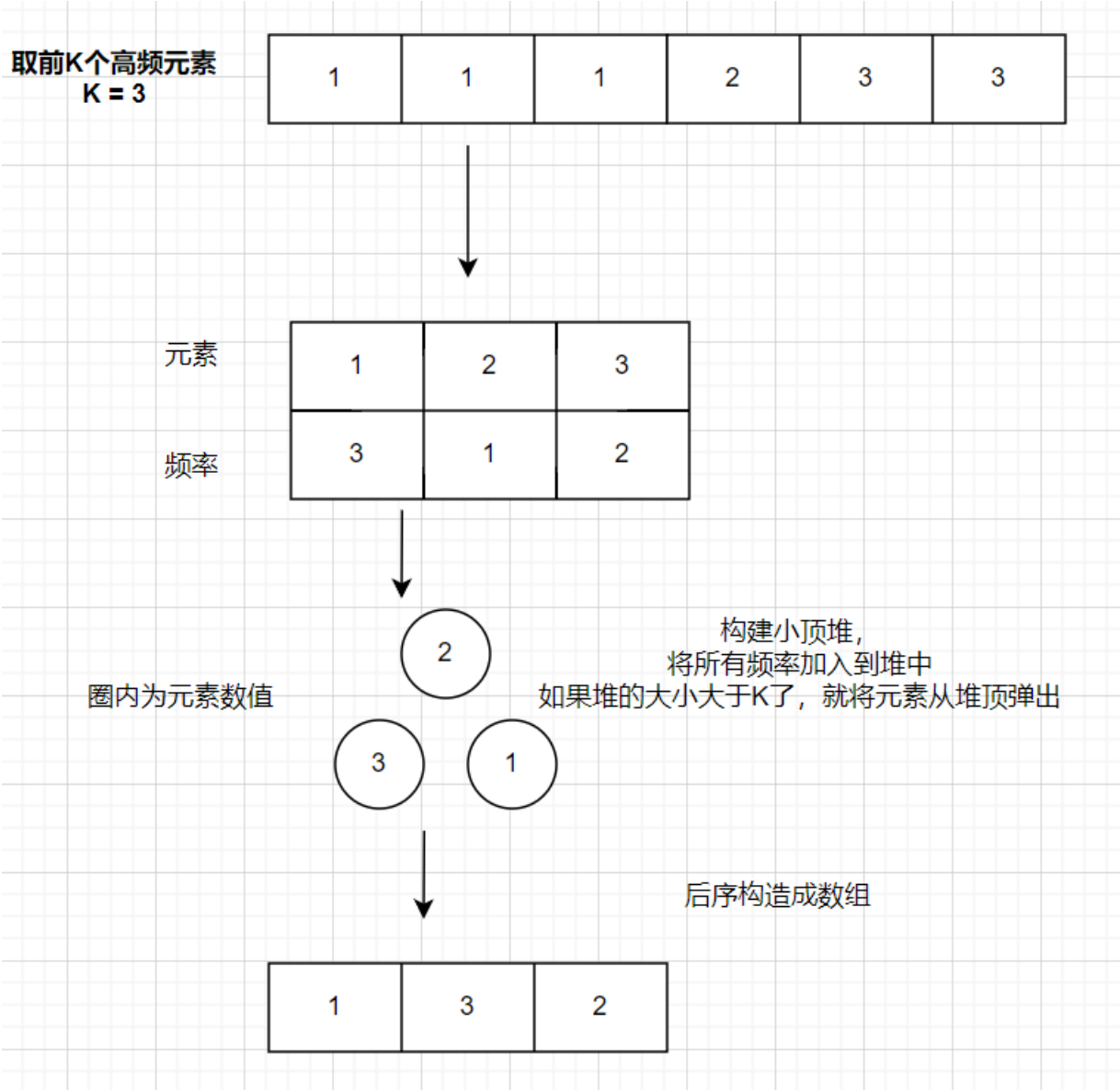
```
1  PriorityQueue<int[]> queue = new PriorityQueue<>((m,n) -> m[1] - n[1]);
```

优先队列：其实**就是一个披着队列外衣的堆**，因为优先队列对外接口只是从队头取元素，从队尾添加元素，再无其他取元素的方式，看起来就是一个队列。而且优先队列内部元素是自动依照元素的权值排列

> 堆：
>
> **堆是一棵完全二叉树，树中每个结点的值都不小于（或不大于）其左右孩子的值。** 如果父亲结点是大于等于左右孩子就是大顶堆，小于等于左右孩子就是小顶堆。

此处需要使用**小顶堆：因为要统计最大前k个元素，只有小顶堆每次将最小的元素弹出，最后小顶堆里积累的才是前k个最大元素。**

使用大顶堆的劣势：在每次移动更新大顶堆的时候，每次弹出都把最大的元素弹出去了，那么怎么保留下来前K个高频元素呢?



实现

Java代码

```java
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> occurrences = new HashMap<>();
        for(int num : nums) {
            occurrences.put(num, occurrences.getOrDefault(num, 0) + 1);
        }
        PriorityQueue<int[]> queue = new PriorityQueue<>((m, n) -> m[1] - n[1]);
        for(Map.Entry<Integer,Integer> entry : occurrences.entrySet()) {
            int num = entry.getKey();
            int cnt = entry.getValue();
            if(queue.size() == k) {
                if(queue.peek()[1] < cnt) {
                    queue.poll();
                    queue.offer(new int[] {num, cnt});
                }
            } else {
```

```
                queue.offer(new int[] {num, cnt});
            }
        }
        int[] ans = new int[k];
        for(int i = 0; i < k; i++) {
            ans[i] = queue.poll()[0];
        }
        return ans;
    }
}
```