宫水三叶的刷题日记

栈

Author ：宫水三叶
Date ：2021/10/07
QQ Group: 703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「栈」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「栈」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「栈」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **20. 有效的括号** ，难度为 **简单**。

Tag：「栈」、「有效括号」

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。

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
```

示例 4：

```
输入：s = "([)]"

输出：false
```

示例 5：

```
输入：s = "{[]}"

输出：true
```

提示：

- 1 <= s.length <= 10^4
- s 仅由括号 '()[]{}' 组成

栈 + 哈希表

这是道模拟题，同一类型的括号，一个右括号要对应一个左括号。

不难发现可以直接使用 `栈` 来解决：

代码：

```
class Solution {
    HashMap<Character, Character> map = new HashMap<Character, Character>(){{
        put(']', '[');
        put('}', '{');
        put(')', '(');
    }};
    public boolean isValid(String s) {
        Deque<Character> d = new ArrayDeque<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c == '(' || c == '{' || c == '[') {
                d.addLast(c);
            } else {
                if (!d.isEmpty() && d.peekLast() == map.get(c)) {
                    d.pollLast();
                } else {
                    return false;
                }
            }
        }
        return d.isEmpty();
    }
}
```

- 时间复杂度：对字符串 `s` 扫描一遍。复杂度为 $O(n)$
- 空间复杂度：使用的哈希表空间固定，不随着样本数量变大而变大。复杂度为 $O(1)$

注意：三叶使用了 `Deque` 双端队列来充当栈，而不是 `Stack`，这也是 JDK 推荐的做法。建议所有的 Java 同学都采用 `Deque` 作为栈。

不使用 `Stack` 的原因是 `Stack` 继承自 `Vector`，拥有了动态数组的所有公共 API，并不安全，而且 `Stack` 还犯了面向对象设计的错误：将组合关系当成了继承关系。

栈 + ASCII 差值

我们也可以利用 `"()"`、`"{}"` 和 `"[]"` 的左右部分在 ASCII 值上比较接近的事实。

`(` 和 `)` 分别对应 -7 和 -8；`[` 和 `]` 分别对应 43 和 45；`{` 和 `}` 分别对应 75 和 77。

也就是同类型的左右括号，相差不超过 2 ，同时不同类型的左右括号，相差大于 2。

利用此特性，我们可以节省一个哈希表：

代码：

```
class Solution {
    public boolean isValid(String s) {
        Deque<Integer> d = new ArrayDeque<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            int u = c - '0';
            if (c == '(' || c == '{' || c == '[') {
                d.addLast(u);
            } else {
                if (!d.isEmpty() && Math.abs(d.peekLast() - u) <= 2) {
                    d.pollLast();
                } else {
                    return false;
                }
            }
        }
        return d.isEmpty();
    }
}
```

- 时间复杂度：对字符串 `s` 扫描一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **32. 最长有效括号** ，难度为 **困难**。

Tag：「栈」、「括号问题」

给你一个只包含 `'('` 和 `')'` 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

```
输入：s = "(()"

输出：2

解释：最长有效括号子串是 "()"
```

示例 2：

```
输入：s = ")()())"

输出：4

解释：最长有效括号子串是 "()()"
```

示例 3：

```
输入：s = ""

输出：0
```

提示：

- 0 <= s.length <= 3 * 10^4
- s[i] 为 `'('` 或 `')'`

栈

从前往后扫描字符串 `s`。

使用 `i` 来记录当前遍历到的位置，使用 `j` 来记录最近的最长有效括号的开始位置的「前一个位置」。

只对 `'('` 进行入栈（入栈的是对应的下标），当遍历到 `')'` 的时候，由于栈中只有 `'('`，所以可以直接弹出一个 `'('` 与之匹配（如果有的话）。

再检查栈中是否还有 `'('`，如果有使用栈顶元素的下标来计算长度，否则使用 `j` 下标来计算

长度。

代码：

```
class Solution {
    public int longestValidParentheses(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        Deque<Integer> d = new ArrayDeque<>();
        int ans = 0;
        for (int i = 0, j = -1; i < n; i++) {
            if (cs[i] == '(') {
                d.addLast(i);
            } else {
                if (!d.isEmpty()) {
                    d.pollLast();
                    int top = j;
                    if (!d.isEmpty()) top = d.peekLast();
                    ans = Math.max(ans, i - top);
                } else {
                    j = i;
                }
            }
        }
        return ans;
    }
}
```

- 时间复杂度：每个字符最多进栈和出栈一次。复杂度为 $O(n)$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **155. 最小栈** ，难度为 **简单**。

Tag：「栈」

设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) —— 将元素 x 推入栈中。
- pop() —— 删除栈顶的元素。
- top() —— 获取栈顶元素。
- getMin() —— 检索栈中的最小元素。

示例:

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]
```

输出：

```
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

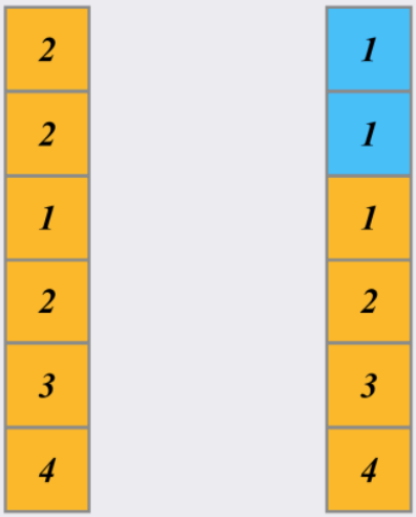
提示：

- pop、top 和 getMin 操作总是在 非空栈 上调用。

双栈解法

为了快速找到栈中最小的元素，我们可以使用一个辅助栈 help。

通过控制 help 的压栈逻辑来实现：*help 栈顶中始终存放着栈内元素的最小值。*



宫水三叶

1. ***push*** 操作*:* 所有的 ***push*** 在 ***data*** 栈中都直接执行；而在 ***help*** 栈中需要分情况讨论：
a. ***help*** 栈为空或者当前元素小于 ***help*** 的栈顶元素：直接将当前元素加入 ***help*** 栈中（黄色部分）；
b. 当前元素大于 ***help*** 栈顶元素：将 ***help*** 栈顶元素复制一份放到栈中（蓝色部分）

2. ***pop*** 和 ***top*** 操作：***data*** 栈和 ***help*** 栈的操作同步进行

代码：

```Java
class MinStack {
    Deque<Integer> data = new ArrayDeque<>();
    Deque<Integer> help = new ArrayDeque<>();

    public void push(int val) {
        data.addLast(val);
        if (help.isEmpty() || help.peekLast() >= val) {
            help.addLast(val);
        } else {
            help.addLast(help.peekLast());
        }
    }

    public void pop() {
        data.pollLast();
        help.pollLast();
    }

    public int top() {
        return data.peekLast();
    }

    public int getMin() {
        return help.peekLast();
    }
}
```

- 时间复杂度：所有的操作均为 $O(1)$
- 空间复杂度：$O(1)$

题目描述

这是 LeetCode 上的 **232. 用栈实现队列** ，难度为 **简单**。

Tag：「栈」、「队列」

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。

进阶：

- 你能否实现每个操作均摊时间复杂度为 O(1) 的队列？换句话说，执行 n 个操作的总时间复杂度为 O(n) ，即使其中一个操作可能花费较长时间。

示例：

```
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）

基本思路

无论「用栈实现队列」还是「用队列实现栈」，思路都是类似的。

都可以通过使用两个栈/队列来解决。

我们创建两个栈，分别为 `out` 和 `in`，用作处理「输出」和「输入」操作。

其实就是两个栈来回「倒腾」。

而对于「何时倒腾」决定了是 **O(n) 解法** 还是 **均摊 O(1) 解法**。

O(n) 解法

我们创建两个栈，分别为 `out` 和 `in`：

- **`in` 用作处理输入操作 `push()`，使用 `in` 时需确保 `out` 为空**
- **`out` 用作处理输出操作 `pop()` 和 `peek()`，使用 `out` 时需确保 `in` 为空**

```Java
class MyQueue {
    Deque<Integer> out, in;
    public MyQueue() {
        in = new ArrayDeque<>();
        out = new ArrayDeque<>();
    }

    public void push(int x) {
        while (!out.isEmpty()) in.addLast(out.pollLast());
        in.addLast(x);
    }

    public int pop() {
        while (!in.isEmpty()) out.addLast(in.pollLast());
        return out.pollLast();
    }

    public int peek() {
        while (!in.isEmpty()) out.addLast(in.pollLast());
        return out.peekLast();
    }

    public boolean empty() {
        return out.isEmpty() && in.isEmpty();
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

均摊 O(1) 解法

事实上，我们不需要在每次的「入栈」和「出栈」操作中都进行「倒腾」。

我们只需要保证，输入的元素总是跟在前面的输入元素的后面，而输出元素总是最早输入的那个元素即可。

可以通过调整「倒腾」的时机来确保满足上述要求，但又不需要发生在每一次操作中：

- **只有在「输出栈」为空的时候，才发生一次性的「倒腾」**

```Java
class MyQueue {
    Deque<Integer> out, in;
    public MyQueue() {
        in = new ArrayDeque<>();
        out = new ArrayDeque<>();
    }

    public void push(int x) {
        in.addLast(x);
    }

    public int pop() {
        if (out.isEmpty()) {
            while (!in.isEmpty()) out.addLast(in.pollLast());
        }
        return out.pollLast();
    }

    public int peek() {
        if (out.isEmpty()) {
            while (!in.isEmpty()) out.addLast(in.pollLast());
        }
        return out.peekLast();
    }

    public boolean empty() {
        return out.isEmpty() && in.isEmpty();
    }
}
```

- 时间复杂度：`pop()` 和 `peek()` 操作都是均摊 $O(1)$
- 空间复杂度：$O(n)$

关于「均摊复杂度」的说明

我们先用另外一个例子来理解「均摊复杂度」，大家都知道「哈希表」底层是通过数组实现的。

正常情况下，计算元素在哈希桶的位置，然后放入哈希桶，复杂度为 $O(1)$，假定是通过简单的“拉链法”搭配「头插法」方式来解决哈希冲突。

但当某次元素插入后，「哈希表」达到扩容阈值，则需要对底层所使用的数组进行扩容，这个复

杂度是 $O(n)$

显然「扩容」操作不会发生在每一次的元素插入中，因此扩容的 $O(n)$ 都会伴随着 `n` 次的 $O(1)$，也就是 $O(n)$ 的复杂度会被均摊到每一次插入当中，因此哈希表插入仍然是 $O(1)$ 的。

同理，我们的「倒腾」不是发生在每一次的「输出操作」中，而是集中发生在一次「输出栈为空」的时候，因此 `pop` 和 `peek` 都是均摊复杂度为 $O(1)$ 的操作。

由于本题的调用次数只有 100 次，所以铁定是一个人均 100% 的算法（0 ms）🐶🐶

我们需要对操作进行复杂度分析进行判断，而不是看时间来判断自己是不是均摊 O(1) 哦 ~

题目描述

这是 LeetCode 上的 **341. 扁平化嵌套列表迭代器** ，难度为 **中等**。

Tag：「DFS」、「队列」、「栈」

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

示例 1:

```
输入: [[1,1],2,[1,1]]
输出: [1,1,2,1,1]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,1,2,1,1]。
```

示例 2:

```
输入: [1,[4,[6]]]
输出: [1,4,6]
解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,4,6]。
```

DFS + 队列

由于所有的元素都是在初始化时提供的，因此一个朴素的做法是在初始化的时候进行处理。

由于存在嵌套，比较简单的做法是通过 DFS 进行处理，将元素都放至队列。

代码：

```
public class NestedIterator implements Iterator<Integer> {

    Deque<Integer> queue = new ArrayDeque<>();

    public NestedIterator(List<NestedInteger> nestedList) {
        dfs(nestedList);
    }

    @Override
    public Integer next() {
        return hasNext() ? queue.pollFirst() : -1;
    }

    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    void dfs(List<NestedInteger> list) {
        for (NestedInteger item : list) {
            if (item.isInteger()) {
                queue.addLast(item.getInteger());
            } else {
                dfs(item.getList());
            }
        }
    }
}
```

- 时间复杂度：构建迭代器的复杂度为 $O(n)$，调用 $next()$ 与 $hasNext()$ 的复杂度为 $O(1)$
- 空间复杂度：$O(n)$

递归 + 栈

另外一个做法是，我们不对所有的元素进行预处理。

而是先将所有的 `NestedInteger` 逆序放到栈中，当需要展开的时候才进行展开。

代码：

```
public class NestedIterator implements Iterator<Integer> {

    Deque<NestedInteger> stack = new ArrayDeque<>();

    public NestedIterator(List<NestedInteger> list) {
        for (int i = list.size() - 1; i >= 0; i--) {
            NestedInteger item = list.get(i);
            stack.addLast(item);
        }
    }

    @Override
    public Integer next() {
        return hasNext() ? stack.pollLast().getInteger() : -1;
    }

    @Override
    public boolean hasNext() {
        if (stack.isEmpty()) {
            return false;
        } else {
            NestedInteger item = stack.peekLast();
            if (item.isInteger()) {
                return true;
            } else {
                item = stack.pollLast();
                List<NestedInteger> list = item.getList();
                for (int i = list.size() - 1; i >= 0; i--) {
                    stack.addLast(list.get(i));
                }
                return hasNext();
            }
        }
    }
}
```

- 时间复杂度：构建迭代器的复杂度为 $O(n)$，$hasNext()$ 的复杂度为均摊 $O(1)$，

$next()$ 严格按照迭代器的访问顺序（ 先 $hasNext()$ 再 $next()$ ）的话为 $O(1)$，防御性编程生效的情况下为均摊 $O(1)$

- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **726. 原子的数量** ，难度为 **困难**。

Tag : 「模拟」、「数据结构运用」、「栈」、「哈希表」、「优先队列」

给定一个化学式 `formula`（作为字符串），返回每种原子的数量。

原子总是以一个大写字母开始，接着跟随0个或任意个小写字母，表示原子的名字。

如果数量大于 1，原子后会跟着数字表示原子的数量。如果数量等于 1 则不会跟数字。例如，`H2O` 和 `H2O2` 是可行的，但 `H1O2` 这个表达是不可行的。

两个化学式连在一起是新的化学式。例如 `H2O2He3Mg4` 也是化学式。

一个括号中的化学式和数字（可选择性添加）也是化学式。例如 `(H2O2)` 和 `(H2O2)3` 是化学式。

给定一个化学式，输出所有原子的数量。格式为：第一个（按字典序）原子的名子，跟着它的数量（如果数量大于 1），然后是第二个原子的名字（按字典序），跟着它的数量（如果数量大于 1），以此类推。

示例 1:

```
输入: formula = "H2O"

输出: "H2O"

解释: 原子的数量是 {'H': 2, 'O': 1}。
```

示例 2:

```
输入: formula = "Mg(OH)2"

输出: "H2MgO2"

解释: 原子的数量是 {'H': 2, 'Mg': 1, 'O': 2}。
```

示例 3:

```
输入: formula = "K4(ON(SO3)2)2"

输出: "K4N2O14S4"

解释: 原子的数量是 {'K': 4, 'N': 2, 'O': 14, 'S': 4}。
```

注意:

- 所有原子的第一个字母为大写，剩余字母都是小写。
- `formula` 的长度在[1, 1000]之间。
- `formula` 只包含字母、数字和圆括号，并且题目中给定的是合法的化学式。

数据结构 + 模拟

一道综合模拟题。

相比于（题解）227. 基本计算器 II 的表达式计算问题，本题设计模拟流程的难度要低很多，之所谓定位困难估计是使用到的数据结构较多一些。

为了方便，我们约定以下命名：

- 称一段完整的连续字母为「原子」
- 称一段完整的连续数字为「数值」
- 称 `(` 和 `)` 为「符号」

基本实现思路如下：

1. 在处理入参 `s` 的过程中，始终维护着一个哈希表 `map`，`map` 中 **实时维护** 着某个「原子」对应的实际「数值」（即存储格式为 `{H:2,S:1}`）；
   > 由于相同原子可以出在 `s` 的不同位置中，为了某个「数值」对「原子」的累

> 乘效果被重复应用，我们这里应用一个”小技巧“：为每个「原子」增加一个”编号后缀“。即实际存储时为 `{H_1:2, S_2:1, H_3:1}` 。

2. 根据当前处理到的字符分情况讨论：
    - 符号：直接入栈；
    - 原子：继续往后取，直到取得完整的原子名称，将完整原子名称入栈，同时在 `map` 中计数加 1；
    - 数值：继续往后取，直到取得完整的数值并解析，然后根据栈顶元素是否为 `)` 符号，决定该数值应用给哪些原子：
        - 如果栈顶元素不为 `)`，说明该数值只能应用给栈顶的原子
        - 如果栈顶元素是)，说明当前数值可以应用给「连续一段」的原子中
3. 对 `map` 的原子做 “合并” 操作：`{H_1:2, S_2:1, H_3:1}` => `{H:3, S:1}` ；
4. 使用「优先队列（堆）」实现字典序排序（也可以直接使用 `List`，然后通过 `Collections.sort` 进行排序），并构造答案。

代码：

```Java
class Solution {
    class Node {
        String s; int v;
        Node (String _s, int _v) {
            s = _s; v = _v;
        }
    }
    public String countOfAtoms(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        Map<String, Integer> map = new HashMap<>();
        Deque<String> d = new ArrayDeque<>();
        int idx = 0;
        for (int i = 0; i < n; ) {
            char c = cs[i];
            if (c == '(' || c == ')') {
                d.addLast(String.valueOf(c));
                i++;
            } else {
                if (Character.isDigit(c)) {
                    // 获取完整的数字，并解析出对应的数值
                    int j = i;
                    while (j < n && Character.isDigit(cs[j])) j++;
                    String numStr = s.substring(i, j);
                    i = j;
                    int cnt = Integer.parseInt(String.valueOf(numStr));

                    // 如果栈顶元素是 )，说明当前数值可以应用给「连续一段」的原子中
                    if (!d.isEmpty() && d.peekLast().equals(")")) {
                        List<String> tmp = new ArrayList<>();

                        d.pollLast(); // pop )
                        while (!d.isEmpty() && !d.peekLast().equals("(")) {
                            String cur = d.pollLast();
                            map.put(cur, map.getOrDefault(cur, 1) * cnt);
                            tmp.add(cur);
                        }
                        d.pollLast(); // pop (

                        for (int k = tmp.size() - 1; k >= 0; k--) {
                            d.addLast(tmp.get(k));
                        }

                    // 如果栈顶元素不是 )，说明当前数值只能应用给栈顶的原子
                    } else {
                        String cur = d.pollLast();
```

```
                    map.put(cur, map.getOrDefault(cur, 1) * cnt);
                    d.addLast(cur);
                }
            } else {
                // 获取完整的原子名
                int j = i + 1;
                while (j < n && Character.isLowerCase(cs[j])) j++;
                String cur = s.substring(i, j) + "_" + String.valueOf(idx++);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
                i = j;
                d.addLast(cur);
            }
        }
    }

    // 将不同编号的相同原子进行合并
    Map<String, Node> mm = new HashMap<>();
    for (String key : map.keySet()) {
        String atom = key.split("_")[0];
        int cnt = map.get(key);
        Node node = null;
        if (mm.containsKey(atom)) {
            node = mm.get(atom);
        } else {
            node = new Node(atom, 0);
        }
        node.v += cnt;
        mm.put(atom, node);
    }

    // 使用优先队列（堆）对 Node 进行字典序排序，并构造答案
    PriorityQueue<Node> q = new PriorityQueue<Node>((a,b)->a.s.compareTo(b.s));
    for (String atom : mm.keySet()) q.add(mm.get(atom));

    StringBuilder sb = new StringBuilder();
    while (!q.isEmpty()) {
        Node poll = q.poll();
        sb.append(poll.s);
        if (poll.v > 1) sb.append(poll.v);
    }

    return sb.toString();
}
}
```

- 时间复杂度：最坏情况下，每次处理数值都需要从栈中取出元素进行应用，处理 `s`

的复杂度为 $O(n^2)$；最坏情况下，每个原子独立分布，合并的复杂度为 $O(n)$；将合并后的内容存入优先队列并取出构造答案的复杂度为 $O(n \log n)$；整体复杂度为 $O(n^2)$

- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1190. 反转每对括号间的子串** ，难度为 **中等**。

Tag：「双端队列」、「栈」

给出一个字符串 s（仅含有小写英文字母和括号）。

请你按照从括号内到外的顺序，逐层反转每对匹配括号中的字符串，并返回最终的结果。

注意，您的结果中 不应 包含任何括号。

示例 1：

```
输入：s = "(abcd)"
输出："dcba"
```

示例 2：

```
输入：s = "(u(love)i)"
输出："iloveu"
```

示例 3：

```
输入：s = "(ed(et(oc))el)"
输出："leetcode"
```

示例 4：

```
输入：s = "a(bcdefghijkl(mno)p)q"
输出："apmnolkjihgfedcbq"
```

提示：

- 0 <= s.length <= 2000
- s 中只有小写英文字母和括号
- 我们确保所有括号都是成对出现的

基本分析

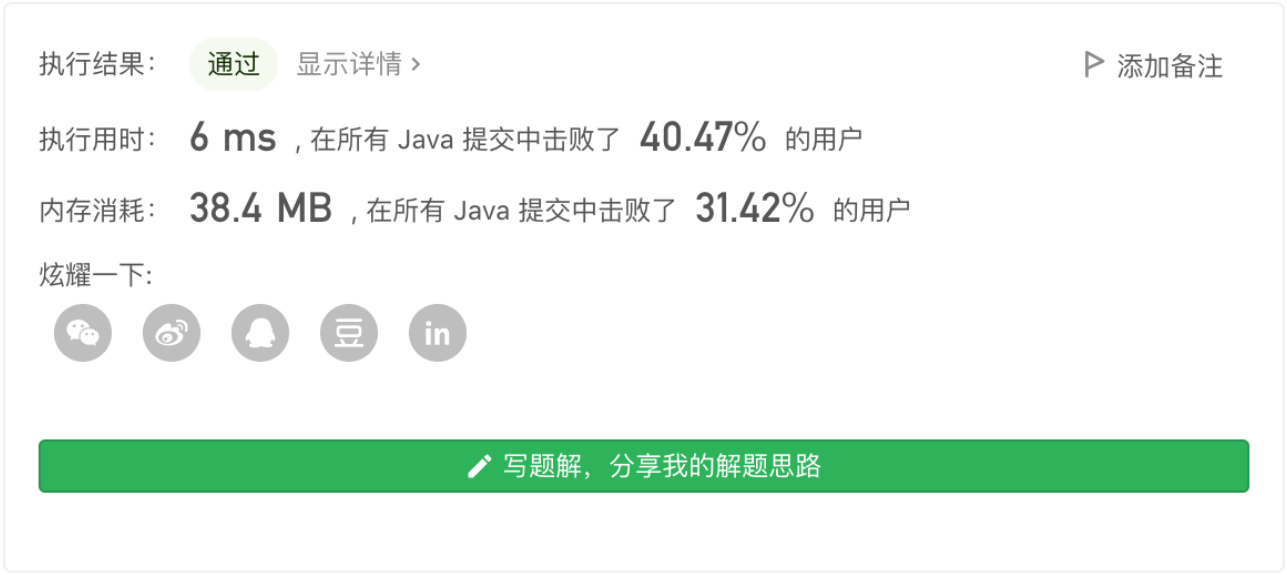
根据题意，我们可以设计如下处理流程：

- 从前往后遍历字符串，将不是 `)` 的字符串从「尾部」放入队列中
- 当遇到 `)` 时，从队列「尾部」取出字符串，直到遇到 `(` 为止，并对取出字符串进行翻转
- 将翻转完成后字符串重新从「尾部」放入队列
- 循环上述过程，直到原字符串全部出来完成
- 从队列「头部」开始取字符，得到最终的答案

可以发现，上述过程需要用到双端队列（或者栈，使用栈的话，需要在最后一步对取出字符串再进行一次翻转）。

在 Java 中，双端队列可以使用自带的 `ArrayDeque`，也可以直接使用数组进行模拟。

语言自带双端队列

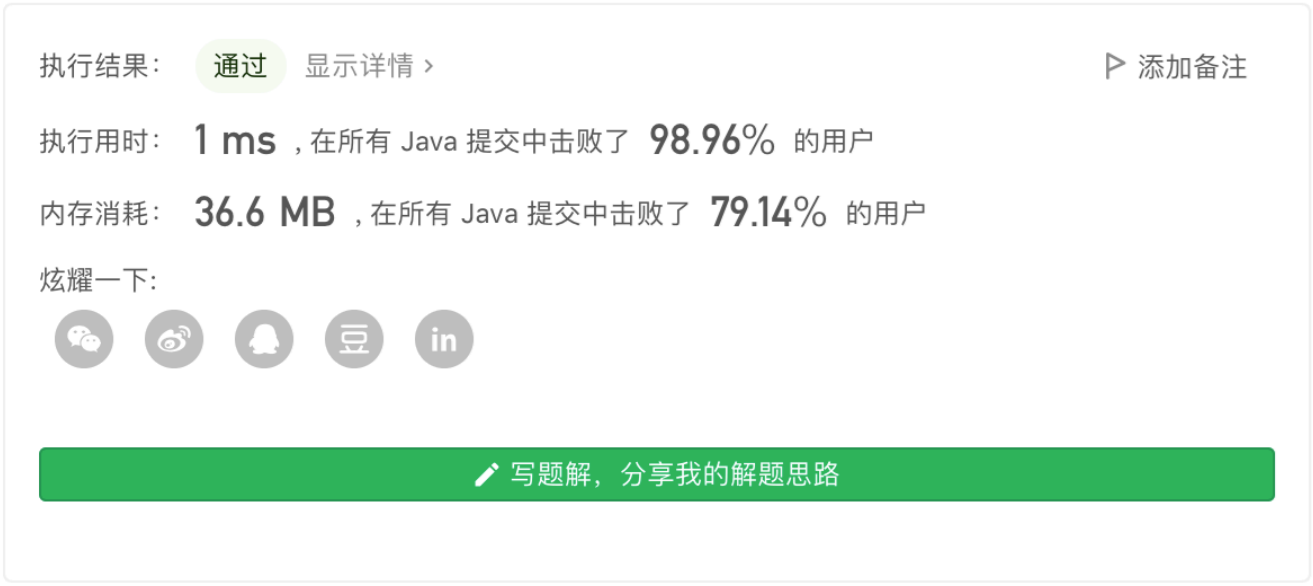


代码：

```Java
class Solution {
    public String reverseParentheses(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        Deque<Character> d = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            char c = cs[i];
            if (c == ')') {
                StringBuilder path = new StringBuilder();
                while (!d.isEmpty()) {
                    if (d.peekLast() != '(') {
                        path.append(d.pollLast());
                    } else {
                        d.pollLast();
                        for (int j = 0; j < path.length(); j++) {
                            d.addLast(path.charAt(j));
                        }
                        break;
                    }
                }
            } else {
                d.addLast(c);
            }
        }
        StringBuilder sb = new StringBuilder();
        while (!d.isEmpty()) sb.append(d.pollFirst());
        return sb.toString();
    }
}
```

- 时间复杂度：每个 `(` 字符只会进出队列一次；`)` 字符串都不会进出队列，也只会被扫描一次；分析的重点在于普通字符，可以发现每个普通字符进出队列的次数取决于其右边的 `)` 的个数，最坏情况下每个字符右边全是右括号，因此复杂度可以当做 $O(n^2)$，但实际计算量必然取不满 n^2，将普通字符的重复弹出均摊到整个字符串处理过程，可以看作是每个字符串都被遍历常数次，复杂度为 $O(n)$
- 空间复杂度：$O(n)$

数组模拟双端队列



代码：

```Java
class Solution {
    char[] deque = new char[2009];
    int head = 0, tail = -1;
    char[] path = new char[2009];
    public String reverseParentheses(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        for (int i = 0; i < n; i++) {
            char c = cs[i];
            if (c == ')') {
                int idx = 0;
                while (tail >= head) {
                    if (deque[tail] == '(') {
                        tail--;
                        for (int j = 0; j < idx; j++) {
                            deque[++tail] = path[j];
                        }
                        break;
                    } else {
                        path[idx++] = deque[tail--];
                    }
                }
            } else {
                deque[++tail] = c;
            }
        }
        StringBuilder sb = new StringBuilder();
        while (tail >= head) sb.append(deque[head++]);
        return sb.toString();
    }
}
```

- 时间复杂度：每个 `(` 字符只会进出队列一次；`)` 字符串都不会进出队列，也只会被扫描一次；分析的重点在于普通字符，可以发现每个普通字符进出队列的次数取决于其右边的 `)` 的个数，最坏情况下每个字符右边全是右括号，因此复杂度可以当做 $O(n^2)$，但实际计算量必然取不满 n^2，将普通字符的重复弹出均摊到整个字符串处理过程，可以看作是每个字符串都被遍历常数次，复杂度为 $O(n)$
- 空间复杂度：$O(n)$

💡**更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。**

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「栈」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码