宫水三叶的刷题日记

# 括号问题

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「括号问题」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「括号问题」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「括号问题」；
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 **22. 括号生成** ，难度为 **中等**。

Tag：「DFS」、「回溯算法」

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。

示例 1：

```
输入：n = 3

输出：["((()))","(()())","(())()","()(())","()()()"]
```

示例 2：

```
输入：n = 1

输出：["()"]
```

提示：

- 1 <= n <= 8

---

# DFS

既然题目是求所有的方案，那只能爆搜了，爆搜可以使用 `DFS` 来做。

从数据范围 `1 <= n <= 8` 来说，`DFS` 应该是稳稳的 AC。

这题的关键是我们要从题目中发掘一些性质：

1. 括号数为 `n`，那么一个合法的括号组合，应该包含 `n` 个左括号和 `n` 个右括号，组合总长度为 `2n`
2. 一对合法的括号，应该是先出现左括号，再出现右括号。那么意味着**任意一个右括号的左边，至少有一个左括号**

其中性质 2 是比较难想到的，我们可以用反证法来证明性质 2 总是成立：

假设某个右括号不满足「其左边至少有一个左括号」，即其左边没有左括号，那么这个右括号就找不到一个与之对应的左括号进行匹配。

这样的组合必然不是有效的括号组合。

使用我们「20. 有效的括号（简单）」的思路（栈）去验证的话，必然验证不通过。

掌握了这两个性质之后，我们可以设定一个初始值为 0 的得分值，令往组合添加一个 `(` 得分值 + 1，往组合添加一个 `)` 得分值 -1。

这样就有：

1. 一个合法的括号组合，最终得分必然为 0（左括号和右括号的数量相等，对应了性质 1）
2. 整个 `DFS` 过程中，得分值范围在 `[0, n]`（得分不可能超过 `n` 意味着不可能添加数量超过 `n` 的左括号，对应了性质 1；得分不可能为负数，意味着每一个右括号必然有一个左括号进行匹配，对应性质 2）

代码：

```
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> ans = new ArrayList<>();
        dfs(0, n * 2, 0, n, "", ans);
        return ans;
    }

    /**
    * i: 当前遍历到位置
    * n: 字符总长度
    * score: 当前得分，令 '(' 为 1， ')' 为 -1
    * max: 最大得分值
    * path: 当前的拼接结果
    * ans: 最终结果集
    */
    void dfs(int i, int n, int score, int max, String path, List<String> ans) {
        if (i == n) {
            if (score == 0) ans.add(path);
        } else {
            if (score + 1 <= max) dfs(i + 1, n, score + 1, max, path + "(", ans);
            if (score - 1 >= 0) dfs(i + 1, n, score - 1, max, path + ")", ans);
        }
    }
}
```

- 时间复杂度：放置的左括号数量为 `n`，右括号的个数总是小于等于左括号的个数，典型的卡特兰数问题。复杂度为 $O(C_{2n}^{n})$
- 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：$O(n)$

---

> **🍥 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 **301. 删除无效的括号** ，难度为 **困难**。

Tag：「括号问题」、「回溯算法」、「DFS」

给你一个由若干括号和字母组成的字符串 s ，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 任意顺序 返回。

示例 1:

```
输入: "()())()"
输出: ["()()()", "(())()"]
```

示例 2:

```
输入: "(a)())()"
输出: ["(a)()()", "(a())()"]
```

示例 3:

```
输入: ")("
输出: [""]
```

提示：

- 1 <= s.length <= 25
- s 由小写英文字母以及括号 '(' 和 ')' 组成
- s 中至多含 20 个括号

---

# DFS 回溯算法

由于题目要求我们将所有（最长）合法方案输出，因此不可能有别的优化，只能进行「爆搜」。

我们可以使用 DFS 实现回溯搜索。

基本思路：

我们知道所有的合法方案，必然有左括号的数量与右括号数量相等。

首先我们令左括号的得分为 1；右括号的得分为 -1。那么对于合法的方案而言，必定满足最终得分为 0。

同时我们可以预处理出「爆搜」过程的最大得分：`max = min(左括号的数量, 右括号的数量)`

**PS.「爆搜」过程的最大得分必然是：合法左括号先全部出现在左边，之后使用最多的合法右括号进行匹配。**

枚举过程中出现字符分三种情况：

- 普通字符：无须删除，直接添加
- 左括号：如果当前得分不超过 `max - 1` 时，我们可以选择添加该左括号，也能选择不添加
- 右括号：如果当前得分大于 0（说明有一个左括号可以与之匹配），我们可以选择添加该右括号，也能选择不添加

使用 Set 进行方案去重，`len` 记录「爆搜」过程中的最大子串，然后将所有结果集中长度为 `len` 的子串加入答案：

```java
class Solution {
    int len;
    public List<String> removeInvalidParentheses(String s) {
        char[] cs = s.toCharArray();
        int l = 0, r = 0;
        for (char c : cs) {
            if (c == '(') {
                l++;
            } else if (c == ')') {
                r++;
            }
        }
        int max = Math.min(l, r);
        Set<String> all = new HashSet<>();
        dfs(cs, 0, 0, max, "", all);
        List<String> ans = new ArrayList<>();
        for (String str : all) {
            if (str.length() == len) ans.add(str);
        }
        return ans;
    }
    /**
     * cs: 字符串 s 对应的字符数组
     * u: 当前决策到 cs 的哪一位
     * score: 当前决策方案的得分值（每往 cur 追加一个左括号进行 +1；每往 cur 追加一个右括号进行 -1）
     * max: 整个 dfs 过程的最大得分
     * cur: 当前决策方案
     * ans: 合法方案结果集
     */
    void dfs(char[] cs, int u, int score, int max, String cur, Set<String> ans) {
        if (u == cs.length) {
            if (score == 0 && cur.length() >= len) {
                len = Math.max(len, cur.length());
                ans.add(cur);
            }
            return;
        }
        if (cs[u] == '(') {
            if (score + 1 <= max) dfs(cs, u + 1, score + 1, max, cur + "(", ans);
            dfs(cs, u + 1, score, max, cur, ans);
        } else if (cs[u] == ')') {
            if (score > 0) dfs(cs, u + 1, score - 1, max, cur + ")", ans);
            dfs(cs, u + 1, score, max, cur, ans);
        } else {
            dfs(cs, u + 1, score, max, cur + String.valueOf(cs[u]), ans);
        }
```

```
    }
}
```

- 时间复杂度：不考虑 score 带来的剪枝效果，最坏情况下，每个位置都有两种选择。复杂度为 $O(n * 2^n)$
- 空间复杂度：最大合法方案数与字符串长度最多呈线性关系。复杂度为 $O(n)$

---

> **🍭 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 **678. 有效的括号字符串** ，难度为 **中等**。

Tag : 「有效括号问题」、「动态规划」、「模拟」

给定一个只包含三种字符的字符串：`(` ，`)` 和 `*`，写一个函数来检验这个字符串是否为有效字符串。

有效字符串具有如下规则：

- 任何左括号 `(` 必须有相应的右括号 `)`。
- 任何右括号 `)` 必须有相应的左括号 `(` 。
- 左括号 `(` 必须在对应的右括号之前 `)`。
- 可以被视为单个右括号 `)` ，或单个左括号 `(` ，或一个空字符串。
- 一个空字符串也被视为有效字符串。

示例 1:

```
输入: "()"

输出: True
```

示例 2:

```
输入: "(*)"

输出: True
```

示例 3:

```
输入: "(*))"

输出: True
```

注意:

- 字符串大小将在 [1，100] 范围内。

---

# 动态规划

**定义 $f[i][j]$ 为考虑前 $i$ 个字符（字符下标从 $1$ 开始），能否与 $j$ 个右括号形成合法括号序列。**

起始时只有 $f[0][0]$ 为 $true$，最终答案为 $f[n][0]$。

不失一般性的考虑 $f[i][j]$ 该如何转移：

- 当前字符为 `(`：如果 $f[i][j]$ 为 $true$，必然有 $f[i-1][j-1]$ 为 $true$，反之亦然。即有 $f[i][j] = f[i-1][j-1]$；
- 当前字符为 `)`：如果 $f[i][j]$ 为 $true$，必然有 $f[i-1][j+1]$ 为 $true$，反之亦然。即有 $f[i][j] = f[i-1][j+1]$；
- 当前字符为 `*`：根据 `*` 代指的符号不同，分为三种情况，只有有一种情况为 $true$ 即可。即有 $f[i][j] = f[i-1][j-1] \vee f[i-1][j+1] \vee f[i-1][j]$。

代码：

```Java
class Solution {
    public boolean checkValidString(String s) {
        int n = s.length();
        boolean[][] f = new boolean[n + 1][n + 1];
        f[0][0] = true;
        for (int i = 1; i <= n; i++) {
            char c = s.charAt(i - 1);
            for (int j = 0; j <= i; j++) {
                if (c == '(') {
                    if (j - 1 >= 0) f[i][j] = f[i - 1][j - 1];
                } else if (c == ')') {
                    if (j + 1 <= i) f[i][j] = f[i - 1][j + 1];
                } else {
                    f[i][j] = f[i - 1][j];
                    if (j - 1 >= 0) f[i][j] |= f[i - 1][j - 1];
                    if (j + 1 <= i) f[i][j] |= f[i - 1][j + 1];
                }
            }
        }
        return f[n][0];
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

---

## 模拟

通过解法一，我们进一步发现，对于某个 $f[i][x]$ 而言（即动规数组中的某一行），值为 $true$ 的必然为连续一段。

**即 由于存在可变化的 `*` 符号，因此考虑在考虑前 $i$ 个字符，其能与消耗的左括号的数量具有明确的「上界与下界」。且当前上界与下界的变化，仅取决于「当前为何种字符」，以及「处理上一个字符时上界与下界为多少」。**

但直接记录所能消耗的左括号上限和下限需要处理较多的边界问题。

我们可以使用与（题解）301. 删除无效的括号 类似的思路：

令左括号的得分为 $1$；右括号的得分为 $-1$。那么对于合法的方案而言，必定满足最终得分为 $0$

。

同时由于本题存在 `*`，因此我们需要记录得分的区间区间是多少，而不仅是一个具体的得分。

具体的，使用两个变量 `l` 和 `r` 分别表示「最低得分」和「最高得分」。

根据当前处理到的字符进行分情况讨论：

- 当前字符为 `(`：`l` 和 `r` 同时加一；
- 当前字符为 `)`：`l` 和 `r` 同时减一；
- 当前字符为 `*`：如果 `*` 代指成 `(` 的话，`l` 和 `r` 都进行加一；如果 `*` 代指成 `)` 的话，`l` 和 `r` 都进行减一；如果 `*` 不变的话，`l` 和 `r` 均不发生变化。因此总的 `l` 的变化为减一，总的 `r` 的变化为加一。

需要注意的是，在匹配过程中如果 `l` 为负数，需要重置为 $0$，因为如果当前序列本身为不合法括号序列的话，增加 `(` 必然还是不合法。同时，当出现 `l > r` 说明上界为负数，即右括号过多，必然为非合法方案，返回 $false$。

代码：

```
class Solution {
    public boolean checkValidString(String s) {
        int l = 0, r = 0;
        for (char c : s.toCharArray()) {
            if (c == '(') {
                l++; r++;
            } else if (c == ')') {
                l--; r--;
            } else {
                l--; r++;
            }
            l = Math.max(l, 0);
            if (l > r) return false;
        }
        return l == 0;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

---

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

💡**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「括号问题」获取下载链接。**

**觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：**