

## 本站简介

- 准备工作：刷题全家桶
  - 配套 Chrome 刷题插件
  - 配套 vscode 刷题插件
  - 配套 JetBrains 刷题插件
  - 算法可视化面板使用说明
  - 本站付费会员
- 极速入门：数据结构及排序
  - 本章导读
  - 基础知识
    - 力扣/LeetCode 解题须知
    - 时间空间复杂度入门
    - 使用可视化面板的 JS 基础
  - 手把手带你实现动态数组
    - 数组（顺序存储）基本原理
    - 动态数组代码实现
  - 手把手带你实现单/双链表
    - 链表（链式存储）基本原理
    - 链表代码实现
  - 手把手带你实现队列/栈
    - 队列/栈基本原理
    - 用链表实现队列/栈
    - 环形数组技巧
    - 用数组实现队列/栈
    - 双端队列（Deque）原理及实现
  - 哈希表的原理及实现
    - 哈希表核心原理
    - 用拉链法实现哈希表
    - 线性探查法的两个难点
    - 线性探查法的两种代码实现
    - 哈希集合的原理及代码实现
  - 哈希表结构的种种变换
    - 用链表加强哈希表（LinkedHashMap）
    - 用数组加强哈希表（ArrayHashMap）
- 二叉树结构及遍历
  - 二叉树基础及常见类型
  - 二叉树的递归/层序遍历
  - 多叉树的递归/层序遍历

- 二叉树结构的种种变换

- 二叉搜索树的应用及可视化
- 红黑树的完美平衡及可视化
- Trie/字典树/前缀树原理及可视化
- 二叉堆核心原理及可视化
- 二叉堆/优先级队列代码实现
- 线段树核心原理及可视化
- 正在更新 ing

- 图论数据结构及遍历

- 图结构基础及通用代码实现
- 图结构的 DFS/BFS 遍历
- Union Find 并查集原理
- 正在更新 ing

- 十大排序算法原理及可视化

- 排序算法的关键指标
- 选择排序所面临的问题
- 拥有稳定性：冒泡排序
- 运用逆向思维：插入排序
- 突破  $O(N^2)$ ：希尔排序
- 妙用二叉树前序位置：快速排序
- 妙用二叉树后序位置：归并排序
- 二叉堆结构的运用：堆排序
- 全新的排序原理：计数排序
- 博采众长：桶排序
- 基数排序 (Radix Sort)

- 正在更新 ing

- 第零章、核心刷题框架汇总

- 本章导读
- 学习数据结构和算法的框架思维
- 双指针技巧秒杀七道链表题目
- 双指针技巧秒杀七道数组题目
- 滑动窗口算法核心代码模板
- 二分搜索算法核心代码模板
- 动态规划解题套路框架
- 回溯算法解题套路框架
- BFS 算法解题套路框架
- 二叉树系列算法核心纲领
- 回溯算法秒杀所有排列/组合/子集问题
- 算法时空复杂度分析实用指南

- 第一章、经典数据结构算法

- 手把手刷链表算法

- 双指针技巧秒杀七道链表题目
- 【强化练习】链表双指针经典习题
- 单链表的花式反转方法汇总

- 如何判断回文链表
- 手把手刷数组算法
  - 双指针技巧秒杀七道数组题目
  - 二维数组的花式遍历技巧
  - 一个方法团灭 nSum 问题
  - 【强化练习】数组双指针经典习题
  - 小而美的算法技巧：前缀和数组
  - 【强化练习】前缀和技巧经典习题
  - 小而美的算法技巧：差分数组
  - 滑动窗口算法核心代码模板
  - 【强化练习】滑动窗口算法经典习题
  - 滑动窗口延伸：Rabin Karp 字符匹配算法
  - 二分搜索算法核心代码模板
  - 实际运用二分搜索时的思维框架
  - 【强化练习】二分搜索算法经典习题
  - 带权重的随机选择算法
  - 田忌赛马背后的算法决策
- 手把手刷二叉树算法
  - 二叉树系列算法核心纲领
  - 二叉树心法（思路篇）
  - 二叉树心法（构造篇）
  - 二叉树心法（后序篇）
  - 二叉树心法（序列化篇）
  - 二叉搜索树心法（特性篇）
  - 二叉搜索树心法（基操篇）
  - 二叉搜索树心法（构造篇）
  - 二叉搜索树心法（后序篇）
- 套模板解决 100 道二叉树习题
  - 【强化练习】用「遍历」思维解题 I
  - 【强化练习】用「遍历」思维解题 II
  - 【强化练习】用「遍历」思维解题 III
  - 【强化练习】用「分解问题」思维解题 I
  - 【强化练习】用「分解问题」思维解题 II
  - 【强化练习】同时运用两种思维解题
  - 【强化练习】利用后序位置解题 I
  - 【强化练习】利用后序位置解题 II
  - 【强化练习】利用后序位置解题 III
  - 【强化练习】运用层序遍历解题 I
  - 【强化练习】运用层序遍历解题 II
  - 【强化练习】二叉搜索树经典例题 I
  - 【强化练习】二叉搜索树经典例题 II
- 二叉树的拓展延伸
  - 拓展：最近公共祖先系列解题框架
  - 拓展：如何计算完全二叉树的节点数
  - 拓展：惰性展开多叉树
  - 拓展：归并排序详解及应用

- 拓展：快速排序详解及应用
- 拓展：用栈模拟递归迭代遍历二叉树
- 手把手设计数据结构
  - 队列实现栈以及栈实现队列
  - 【强化练习】栈的经典习题
  - 【强化练习】括号类问题汇总
  - 【强化练习】队列的经典习题
  - 单调栈算法模板解决三道例题
  - 【强化练习】单调栈的几种变体及经典习题
  - 单调队列结构解决滑动窗口问题
  - 【强化练习】单调队列的通用实现及经典习题
  - 算法就像搭乐高：手撸 LRU 算法
  - 算法就像搭乐高：手撸 LFU 算法
  - 常数时间删除/查找数组中的任意元素
  - 【强化练习】哈希表更多习题
  - 【强化练习】优先级队列经典习题
  - TreeMap/TreeSet 代码实现
  - Trie/字典树/前缀树代码实现
  - 【强化练习】Trie 树算法习题
  - 设计朋友圈时间线功能
  - 设计考场座位分配算法
  - 【强化练习】更多经典设计习题
  - 拓展：如何实现一个计算器
  - 拓展：两个二叉堆实现中位数算法
  - 拓展：数组去重问题（困难版）
- 手把手刷图算法
  - 环检测及拓扑排序算法
  - 众里寻他千百度：名流问题
  - 二分图判定算法
  - Union-Find 并查集算法
  - 【强化练习】并查集经典习题
  - Kruskal 最小生成树算法
  - Prim 最小生成树算法
  - Dijkstra 算法模板及应用
- 第二章、经典暴力搜索算法
  - DFS/回溯算法
    - 回溯算法解题套路框架
    - 回溯算法秒杀所有排列/组合/子集问题
    - 球盒模型：回溯算法穷举的两种视角
    - 解答回溯算法/DFS 算法的若干疑问
    - 一文秒杀所有岛屿题目
    - 回溯算法实践：数独和 N 皇后问题
    - 回溯算法实践：括号生成
    - 回溯算法实践：集合划分
    - 【强化练习】回溯算法经典习题 I
    - 【强化练习】回溯算法经典习题 II

- 【强化练习】回溯算法经典习题 III

- BFS 算法

- BFS 算法解题套路框架
    - 如何用 BFS 算法秒杀各种智力题
    - 【强化练习】BFS 经典习题 I
    - 【强化练习】BFS 经典习题 II
    - 正在更新 ing

- 第三章、经典动态规划算法

- 动态规划基本技巧

- 动态规划解题套路框架
    - 动态规划设计：最长递增子序列
    - base case 和备忘录的初始值怎么定？
    - 动态规划穷举的两种视角
    - 动态规划和回溯算法的思维转换
    - 对动态规划进行降维打击
    - 最优子结构原理和 dp 数组遍历方向

- 子序列类型问题

- 经典动态规划：编辑距离
    - 动态规划设计：最大子数组
    - 经典动态规划：最长公共子序列
    - 动态规划之子序列问题解题模板

- 背包类型问题

- 经典动态规划：0-1 背包问题
    - 经典动态规划：子集背包问题
    - 经典动态规划：完全背包问题
    - 背包问题的变体：目标和

- 用动态规划玩游戏

- 动态规划之最小路径和
    - 动态规划帮我通关了《魔塔》
    - 动态规划帮我通关了《辐射4》
    - 旅游省钱大法：加权最短路径
    - 经典动态规划：正则表达式
    - 经典动态规划：高楼扔鸡蛋
    - 经典动态规划：戳气球
    - 经典动态规划：博弈问题
    - 一个方法团灭 LeetCode 打家劫舍问题
    - 一个方法团灭 LeetCode 股票买卖问题

- 贪心类型问题

- 老司机加油算法
    - 贪心算法之区间调度问题
    - 扫描线技巧：安排会议室
    - 剪视频剪出一个贪心算法

- 如何运用贪心思想玩跳跃游戏

- 第四章、其他常见算法技巧

- 数学运算技巧

- 一行代码就能解决的算法题
  - 常用的位操作
  - 谈谈游戏中的随机算法
  - 讲两道常考的阶乘算法题
  - 如何高效寻找素数
  - 如何高效进行模幂运算
  - 如何同时寻找缺失和重复的元素
  - 几个反直觉的概率问题
  - 【强化练习】数学技巧相关习题

- 经典面试题

- 算法笔试「骗分」套路
  - 如何高效解决接雨水问题
  - 一文秒杀所有丑数系列问题
  - 分治算法详解：运算优先级
  - 一个方法解决三道区间问题
  - 谁能想到，斗地主也能玩出算法
  - 烧饼排序算法
  - 字符串乘法计算
  - 如何判定完美矩形

- 附录

- 习题汇总页面
  - [labuladong.online](#) 更新日志
  - 可视化面板更新日志
  - [Chrome](#) 刷题插件更新日志
  - [vscode](#) 刷题插件更新日志
  - [Jetbrain](#) 刷题插件更新日志
  - 网站/插件问题反馈

# 本站简介

## 本站效用

帮你练成框架化、模板化的思维方式解决算法题。如果满分 100 分，无论你的基础如何，本站都可以在最短的时间内帮你提升至 85 分的水平。

因为本站教的是模板化、框架化的思维模式，所以你练成的这个 85 分是可以隔绝运气，稳定复现的。换句话说，难度小于 85 的题目，你一定能做出来，难度大于 85 的题目，才依赖灵感和运气。

对比参考一下，假设你大学学计算机专业，上过学校开设的数据结构和算法必修课，工作中主要使用各种开发框架，从没刷过算法题，那么你的水平大概是 30~40 分。真的是这样，因为算法这个东西可以理解为一种独立的能力，和编程经验没有直接的关系，需要专门花时间练习。

所以不要觉得 85 分低，这个水平对于技术岗面试笔试绰绰够用了。当然，也不必畏惧算法，只要使用正确的方法专门花些时间刷题，算法水平提升起来也是很容易的。

## 阅读方法

对于初学者，按照本站目录顺序阅读，稳扎稳打即可。

对于有一定基础的读者（刷题量 > 300），第零章的内容必读，其他章节可以根据需求自行选择。

对于时间紧急临阵磨枪的读者（准备时间 < 30 天），建议面向习题进行训练，着重学习目录中标记为 **【强化练习】** 的章节，可以练出肌肉记忆，短期冲刺。

本站共讲解 500+ 道力扣算法题，主要有三种文章类型：

**① 数据结构基础教程**（约占本站全部内容的 10%）

主要集中在 **极速入门** 这一章。这部分内容讲解排序算法以及经典数据结构的核心原理和代码实现，一般不包含算法题，主要目的是让

**② 结合例题讲解经典算法框架**（约占本站全部内容的 50%）

一些比较经典的算法框架或算法题，我会用整篇文章结合具体的例题做比较细致的讲解，目的是让你理解原理。一般每篇文章包含 2~5 道例题，阅读时顺手就能做掉。

**③ 习题章节帮你熟练运用算法框架**（约占本站全部内容的 40%）

目录中标有 **【强化练习】** 的内容属于习题部分，一般紧跟在算法框架内容的后面。相比于例题，习题的重点在于举一反三和融会贯通，我会用大量习题手把手帮你训练框架思维，练出肌肉记忆，彻底掌握一类题型的解法。每篇强化练习包含 10 道左右的习题。

## 关于本站

认准本站的最新地址，持续更新：[labuladong.online](#)

本站同时支持手机、PC 阅读，手机端建议直接在微信内打开，方便一键登陆。

本站可以用框架思维手把手带你解决 LeetCode/力扣 500 道以上的题目，且训练出的算法思维不会退化，多年之后也能快速重拾解题能力。

点击本站右上角的放大镜图标可以搜索本站内容，支持直接搜索 LeetCode/力扣 的中英文题目、题号、链接。

本站在持续更新和优化，顶部有「更新日志」和「bug 反馈」入口。

由于我在不断更新、优化、修正算法教程，一些历史遗留的内容、网站都应该被废弃，因为现在有更简洁、优质的内容提供给你学习。

我在网络上其他平台发表的算法教程都已经过时，不再更新。

我曾经使用过的网站地址已经不再维护，包括：

[labuladong.gitee.io/algo/](http://labuladong.gitee.io/algo/)

[labuladong.github.io/algo/](http://labuladong.github.io/algo/)

[labuladong.gitbook.io/algo/](http://labuladong.gitbook.io/algo/)

曾经的 PDF 也已经过时，包括：《labuladong 的算法小抄》《labuladong 的刷题笔记》《labuladong 的算法秘籍》等。

应部分读者的要求，我整理了一份题单，汇总了本站讲解过的所有题目。

但是请注意，我个人觉得这个题单用处不大，也不建议你直接对着这个题单刷题。

因为题单里面的题目是随机排序的，没办法做到循序渐进地学习，也没办法对某个知识点针对性学习。

我会更建议你按照本站目录顺序学习，文中提到的题目，你顺手去做掉。等你刷完本站，这个题单里的所有题目，你就都做完了。

这个题单唯一的作用也许是证明我没吹牛，本站确实可以手把手带你解决 500 道以上的题目。安装 [Chrome 刷题插件](#) 后点开这个题单，可以看到所有题目都有我的题解/思路标记，即这些题目都是本站讲解过的。

## 力扣

## LeetCode

<https://leetcode.cn/problem-list/59jEaTgw/>    <https://leetcode.com/list/9zwo3ww5/>

## 关于作者

我是 [fucking-algorithm](#) 仓库的作者 labuladong，读者一般叫我「东哥」，首创框架思维解题，多次霸榜 GitHub Trending，目前已获得 125k star。

我致力于做最好的算法教程，专注小而精，不做大而全；力争让读者在最短的时间内，彻底拿下算法这个硬骨头。

[本站会员](#) 为本站唯一的付费项目，一顿饭钱，即可解锁本站所有内容和所有配套工具，享受最丝滑流畅的学习体验。

## 本站实用功能

### 1. 支持算法可视化动画

本站和所有配套插件的所有解法代码都配备了一套算法可视化面板，可以直观地查看算法执行过程，辅助理解算法逻辑。可视化面板放在每道题目的解法代码下方。

比如这个 [计算单链表环起点](#) 的算法，你可以多次点击 `if (fast == slow) break;` 这部分代码，即可看到快慢指针追逐相遇的过程；再多次点击 `while (slow != fast)` 这部分代码，即可看到快慢指针同步前进在环起点相遇：

## ► 🌟 代码可视化动画🌟

可视化面板支持对排序算法进行可视化，比如下面是 [插入排序](#) 的代码，你可以点击左上角的播放按钮，加快播放速度，查看排序过程：

## ► 😊 代码可视化动画😊

可视化面板对递归算法可视化有特殊的加强，比如斐波那契数的递归解法，你可以不断点击 `if (n < 2)` 这一行代码，即可看到递归树的生长过程：

## ► 🌟 代码可视化动画🌟

可视化面板可以大幅降低你对复杂算法的理解成本，具体的介绍请看 [可视化面板简介](#)，这里就不多展示了。

## 2. 支持阅读历史

在侧边栏中，未读过的文章会显示 标记，读过但未学完的文章会显示 标记，学完的文章会显示 标记，方便你了解自己的学习进度。

在文章内的引用链接，也会有同样的标记，方便你知道这篇文章是否已经学过。

## 3. 支持所有主流编程语言

本站和所有配套插件的所有题解代码都支持了 Java/C++/Python/Golang/JavaScript 等所有常用的编程语言，能尽可能照顾到更多读者的需求。

Java 代码都是我写的，其他语言的代码由 chatGPT 辅助翻译，但都经过了我的亲自验证和调试，确保正确性和一致性。

## 4. 代码图片注释

本站和所有配套插件中，对于较为复杂的代码块中会包含小灯泡图标，鼠标移至小灯泡图标上就会弹出图片辅助理解：

```
class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        // 上面的代码类似 hasCycle 函数
        if (fast == null || fast.next == null) {
            // fast 遇到空指针说明没有环
            return null;
        }

        // 重新指向头结点
        slow = head;
        // 快慢指针同步前进，相交点就是环起点
        while (slow != fast) {
            fast = fast.next;
            slow = slow.next;
        }
    }
}
```

```
        return slow;
    }
}
```

## 本站配套刷题插件

为了尽可能满足不同读者的需求，我开发维护了本站配套刷题插件，以让读者在喜欢的代码编辑器中刷题，方便摸鱼。

**在刷题插件中，所有本站讲解过的习题都有特殊加强，可以查看简短思路或详细题解，且支持可视化面板、图片注释等所有本站实用功能。**

刷题插件并不是必须安装的，但我会建议安装 Chrome 浏览器插件，因为你在阅读本站时，可能会经常跳转到 LeetCode/力扣 页面上刷题，Chrome 插件能够给你提供一些帮助。vscode/Jetbrain 插件可以根据自己的刷题需求选择安装。

各个刷题插件的安装使用方法详见 [Chrome 插件](#)、[vscode 插件](#)、[Jetbrain 插件](#)。

# 配套 Chrome 刷题插件

插件中的题解和思路都是本站的文章和习题讲解。换句话说，本站包含全部算法教程和习题，并没有什么内容是插件独有的。

所以刷题插件只是辅助大家学习本站的配套工具，并不是必须安装的，大家可以根据需求自行选择。

不过我会建议安装 Chrome 刷题插件，因为你在浏览器上学习本站时，可能会经常跳转到 力扣/LeetCode 的页面做题，这时候 Chrome 插件会提供一些帮助。

虽然我习惯叫这款插件为「Chrome 插件」，但实际上插件并不仅限于 Chrome 浏览器安装使用。像 Edge 浏览器、360 浏览器这些使用 Chromium 内核的浏览器都可以安装，具体见安装指南。

Chrome 刷题插件的主要功能是在力扣和 LeetCode 的页面上添加「题解」和「思路」按钮，方便跳转查看本站算法文章和解题思路：

labuladong 的 Chrome 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



## 功能展示

### 题目列表渲染

力扣 or LeetCode 的所有题目列表和题目详情页中会显示我的题解/思路按钮，支持所有常用编程语言：

The screenshot shows the 'LeetCode HOT 100' page. At the top, there's a 'HOT' badge and navigation links for '求职', '力扣经典', and '新手'. Below is a brief introduction: '精选 100 道力扣 (LeetCode) 上最热门的题目，适合初识算法与数据结构的新手和想要在短时间内高效提升的人，熟练掌握这 100 道题，你就已经具备了在代码世界通行的基本能力。' A progress bar indicates '完成度: 71/100'. The main area is a table listing 23 problems from the top 100, including their titles, difficulty levels (Simple, Medium, Hard), and completion percentages. Each row has a '题解' (Solution) button.

状态	题目	题解	通过率	难度	出现频率
✓	1. 两数之和	题解	52.9%	简单	高
✓	2. 两数相加	思路	42.3%	中等	高
✓	3. 无重复字符的最长子串	★ 题解	39.1%	中等	高
	4. 寻找两个正序数组的中位数	6410	41.6%	困难	高
✓	5. 最长回文子串	题解	37.4%	中等	高
✓	10. 正则表达式匹配	题解	31.0%	困难	高
✓	11. 盛最多水的容器	题解	60.7%	中等	高
	15. 三数之和	题解	36.6%	中等	高
✓	17. 电话号码的字母组合	思路	58.0%	中等	高
✓	19. 删除链表的倒数第 N 个结点	题解	45.1%	中等	高
✓	20. 有效的括号	题解	44.4%	简单	高
✓	21. 合并两个有序链表	题解	66.5%	简单	高
✓	22. 括号生成	★ 题解	77.6%	中等	高
✓	23. 合并K个升序链表	题解	57.5%	困难	高

点击「题解」按钮即可跳转到网站对应文章学习，点击「思路」按钮即可查看我精心撰写的简明解题思路和代码。

同时，我把网站里所有我讲过的题目整理出一份列表，安装插件后访问可以看到效果：

## 力扣版

## LeetCode 版

<https://leetcode.cn/problem-list/59jEaTgw/>    <https://leetcode.com/list/9zwo3ww5/>

## 思路/题解辅助刷题

题目详情页也会显示题解和思路按钮，可以直接复制带详细注释的代码，刷题非常方便：

The screenshot shows the details page for LeetCode problem 264.丑数 II. It includes tabs for '题目描述', '评论 (619)', '题解 (909)', '提交记录', and language selection ('Java'). The main content area shows the problem statement: '给你一个整数 n，请你判断该数是否为丑数。丑数就是只包含质因数 2, 3, 5 的正整数。' Below it is the '基本思路' section with a note: '这道题很精妙，你看它好像是道数学题，实际上它却是一个合并多个有序链表的问题，同时用到了筛选素数的思路。' It also includes sections for '示例 1:' and '示例 2:', both with Java code snippets. The '提示:' section contains hints and sample code for generating ugly numbers by multiplying 2, 3, and 5.

## 代码图片注释

对于比较复杂的解法，代码中会包含图片注释，方便理解代码逻辑。鼠标移动到小灯泡图表即可查看图片注释。

## 算法可视化面板

「思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：

三> 题库 < > ☰

题目描述 讨论 (1.7K) 题解 (3.8K) 提交记录

### 142. 环形链表 II

labuladong 题解 | 思路

中等 ✓ 2.1K ⚡

字节跳动 Google Facebook ...

给定一个链表的头节点 head，返回链表开始入环的第一个节点。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到环。评测系统内部使用整数 pos 来表示链表尾部连接到链表中的位置，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅表示不允许修改链表。

示例 1：

```
graph LR; 1((1)) --> 3((3)); 3 --> 5((5)); 5 --> 7((7)); 7 --> 2((2)); 2 --> 1;
```

输入: head = [3,2,0,-4], pos = 1  
输出: 返回索引为 1 的链表节点  
解释: 链表中有一个环, 其尾部连接到第二个节点。

示例 2:

控制台 ^

控制台

提交

bu 反馈 | 使用指南 | 多选配伍题

### ▼ 🎃 算法可视化 🎃

42 / 65: EXPRESSION

fast != null && fast.next != null = true

head = 1  
fast = 5  
slow = 7

head → 1 → 3 → 5 → 7 → 2 → 4 → head

detectCycle(head=1)

类似题目:

可视化面板的具体功能请参考 [算法可视化面板简介](#)。

## 安装方式

Chrome 刷题插件不止支持 Chrome 浏览器，像 Edge 浏览器、360 浏览器这些使用 Chromium 内核的浏览器都可以通过离线安装 crx 文件的方式安装。

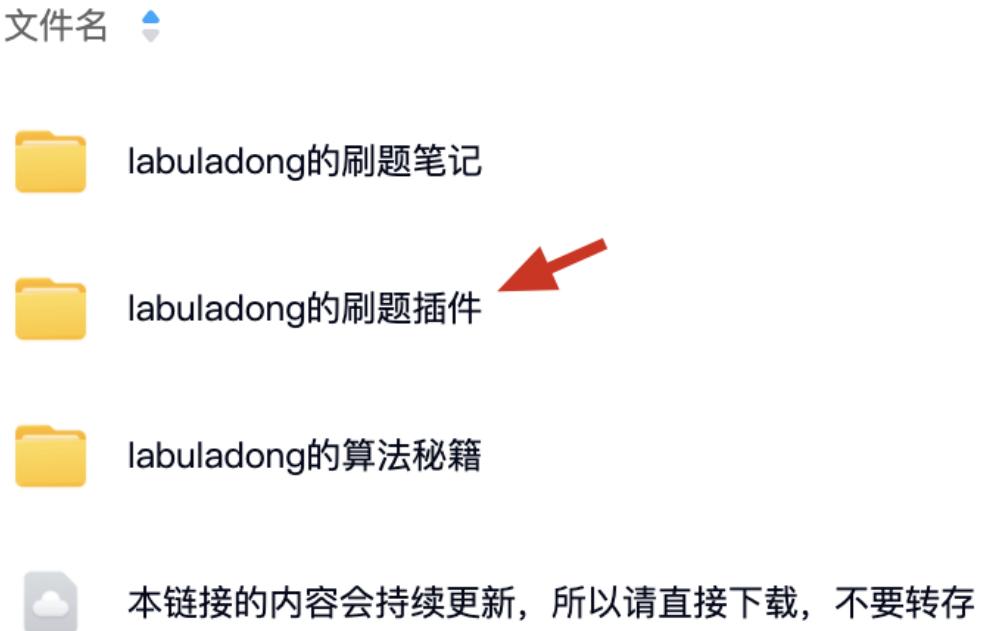
**Chrome 浏览器用户有条件的话建议直接在 Chrome 商店下载：**

<https://chrome.google.com/webstore/detail/leetcode-helper-by-labula/elafhogmnaappleckoedgipgmidneccg>

**Edge 浏览器用户**可以直接在 Edge 商店下载（国内也可以访问）：

<https://microsoftedge.microsoft.com/addons/detail/leetcode-helper-by-labula/mgfjpejofejdbnillfolnnjbiefpokln>

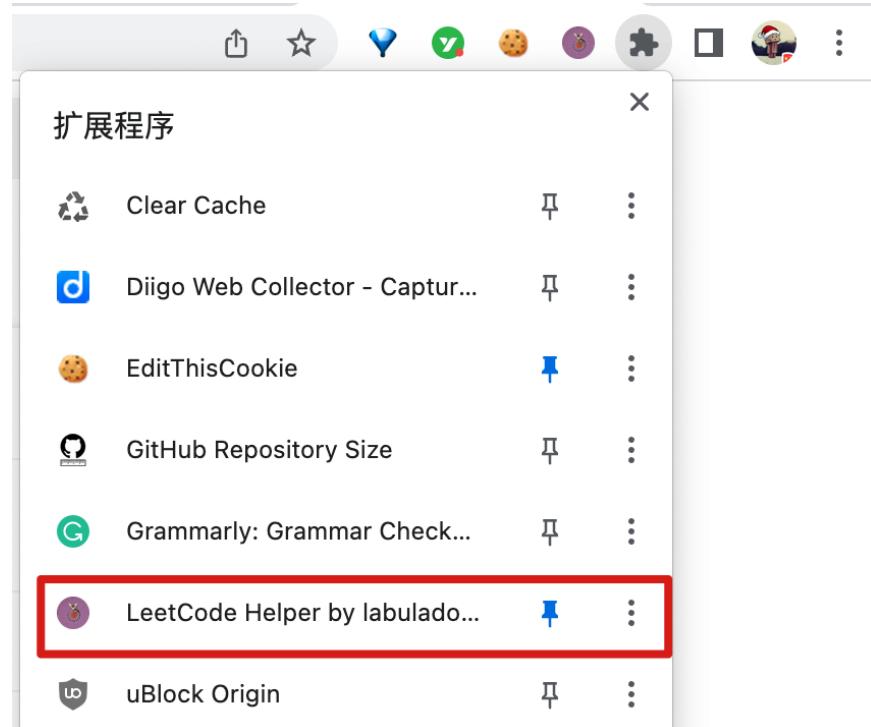
如果无法在线安装，也可以离线安装。在公众号后台回复关键词「插件」下载最新版本 crx 文件：



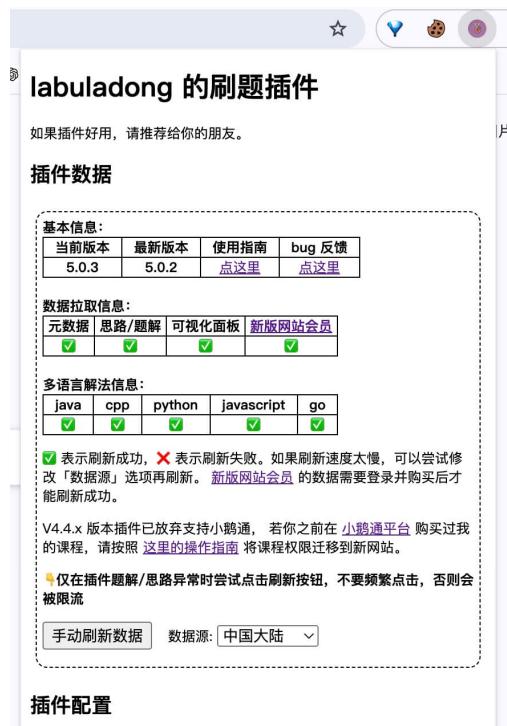
有了 crx 插件文件，手动安装的方法在网上一搜一大把，我就随便贴一个吧：

<https://cloud.tencent.com/developer/article/1894180>

安装成功后，可以在插件列表看到插件图标：



点击插件图标可以弹出插件弹窗，包含手动刷新数据的按钮和插件相关配置：



## 插件配置方法

点击插件图标打开插件弹窗，下滑即可看到配置选项：



列表渲染设置可以选择是否把题目列表中有题解/思路的题目行渲染成绿色。

在题目详情页，可以使用自定义快捷键快速打开/关闭思路弹窗和复制解法代码。

## 国际用户切换英文

插件的题解/思路/可视化面板的默认语言是中文，可以在配置页面可以切换为英文，方便国际用户使用。

## 更新方式

1、在 Chrome/Edge 商店安装的，会在新版本发布后自动更新。

2、通过 crx 文件离线安装的，我会在发布新版本之后更新百度网盘中的 crx 文件，所以你只需要重新在公众号后台回复关键词「插件」即可下载最新版本插件，再次安装即可。建议将插件固定在插件栏，有更新时会有 new 的提示字样。

任何时候，都推荐你使用最新版本的插件，因为我会不断优化插件的功能，修复 bug，所以如果你遇到了问题，也许更新到最新版本就能解决。

## 更新日志

详见 [Chrome 插件更新日志](#)。

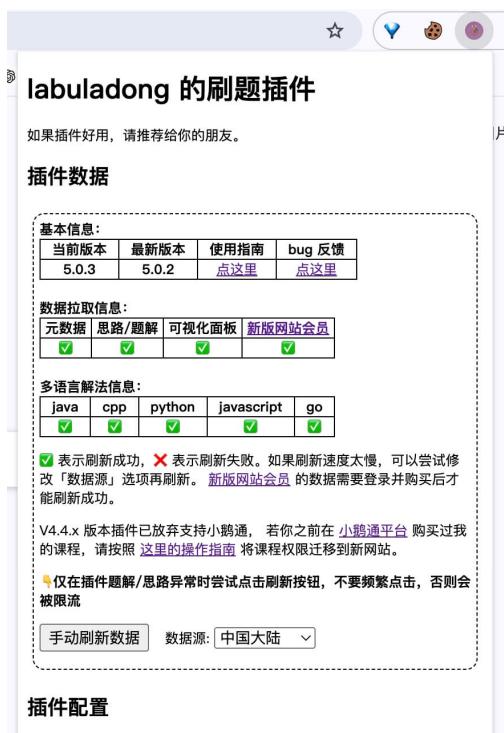
## 在插件中解锁本站习题讲解

本站习题章节中的所有题目也可以在插件中学习，不过需要你购买本站会员后手动刷新数据才能解锁对应习题的题解/思路/可视化等功能，具体操作方法参见 [会员购买页](#) 下方。

## 常见问题解决方法

### 安装插件后没有效果？

主要原因可能是网络问题，导致插件拉取初始数据时较慢。可以稍等一会儿，或者点击「手动刷新数据」按钮：



等待数据拉取完成后再尝试刷新网页，应该就能看到题解和思路按钮了。

### 无法成功刷新插件数据（显示 ✗ 标记）？

如果点击「手动刷新数据」也没有反应或全部显示 ✗ 标记，大概率是你的网络问题。「手动刷新数据」按钮右侧有一个下拉框，默认选项为 [自动选择](#)，即插件会根据网速自动选择 [中国大陆](#) 或 [非中国大陆](#) 拉取数据。

如果你的网络情况不稳定，可以尝试自己修改下拉框选项，选择 [中国大陆](#) 或 [非中国大陆](#)，然后再点击「手动刷新数据」按钮，直到出现 ✓ 标记为止。

## 切换深色/浅色模式后思路弹窗的样式异常？

正常来说，当力扣/LeetCode 进行深色/浅色模式切换后，只需刷新一下页面，插件也会适配深色/浅色模式。

如果刷新页面后插件样式依然异常，一般的原因是某些其他插件在网页插入的 CSS 样式和本插件的 CSS 样式冲突了。你需要排查是哪个其他插件引起的问题，然后暂时关闭该插件。

你可以复制这个地址到 Chrome 地址栏，即可打开本刷题插件的管理页面：

`chrome://extensions/?id=elafhogmnaappleckojedgipgmidneccg`

打开其中的「在无痕模式下启用」选项，然后开一个新的无痕窗口，访问出问题的力扣/LeetCode 页面，看看此时插件是否正常显示。

此时相当于关闭了其他所有插件，仅保留刷题插件。如果样式正常，就说明是其他插件插入的 CSS 引起的问题。

## 火狐浏览器可以安装吗？

不支持，因为火狐浏览器用的不是 chromium 内核。以前还有办法安装，但现在 chromium 内核接口升级了，所以除非火狐官方适配，否则无法安装 Chrome 插件。

不过，类似 QQ 浏览器、360 浏览器这些使用 chromium 内核的浏览器是可以安装的。

## macOS/Linux/Windows 都能安装吗？

是的，插件是寄生在浏览器里的，和操作系统无关。只要在你的操作系统安装 chromium 内核的浏览器，就能够正常安装使用插件。

## bug 反馈

可以在 GitHub 创建 Issue 反馈问题：

<https://github.com/labuladong/fucking-algorithm/issues/>

# 配套 vscode 刷题插件

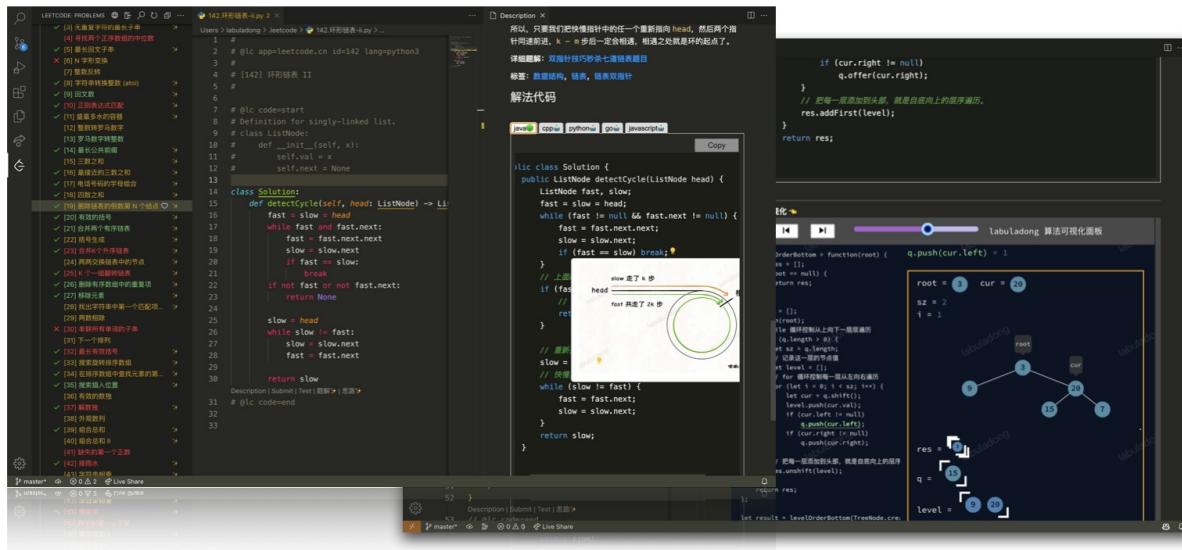
插件中的题解和思路都是本站的文章和习题讲解，换句话说，本站包含全部算法教程和习题，并没有什么内容是插件独有的。

只不过有些读者不喜欢在网页上刷题，认为在编辑器或 IDE 中刷题更方便编写和调试代码，于是我开发维护了各个平台的刷题插件，满足这部分读者的需求。

所以，刷题插件只是辅助大家学习本站的配套工具，并不是必须安装的，大家可以根据需求自行选择。

vscode 插件可以让大家在 vscode 编辑器中刷 LeetCode/力扣，同时查看我的思路讲解：

labuladong 的 vscode 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



## 使用方法

本插件是我基于以下两款开源 vscode 插件修改而来：

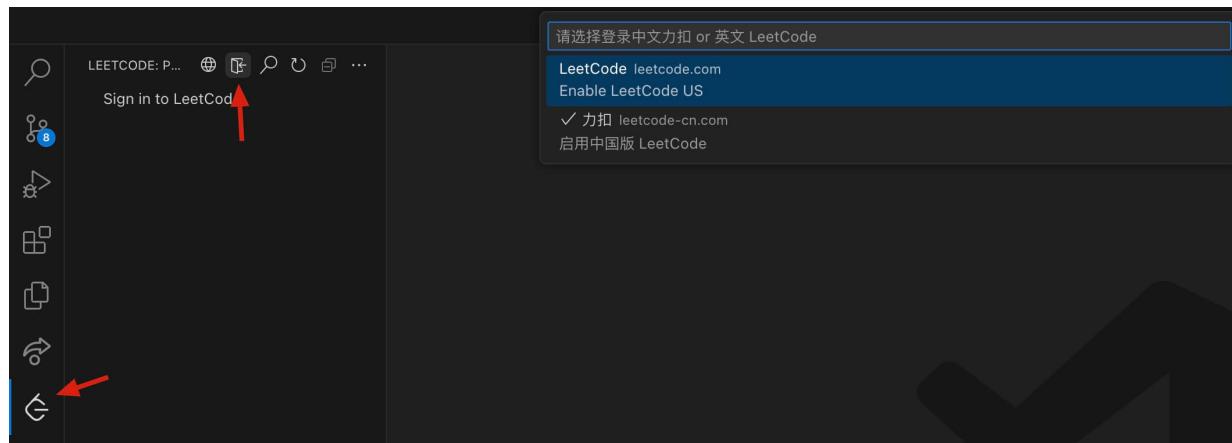
<https://github.com/LeetCode-OpenSource/vscode-leetcode>

<https://github.com/ccagml/leetcode-extension>

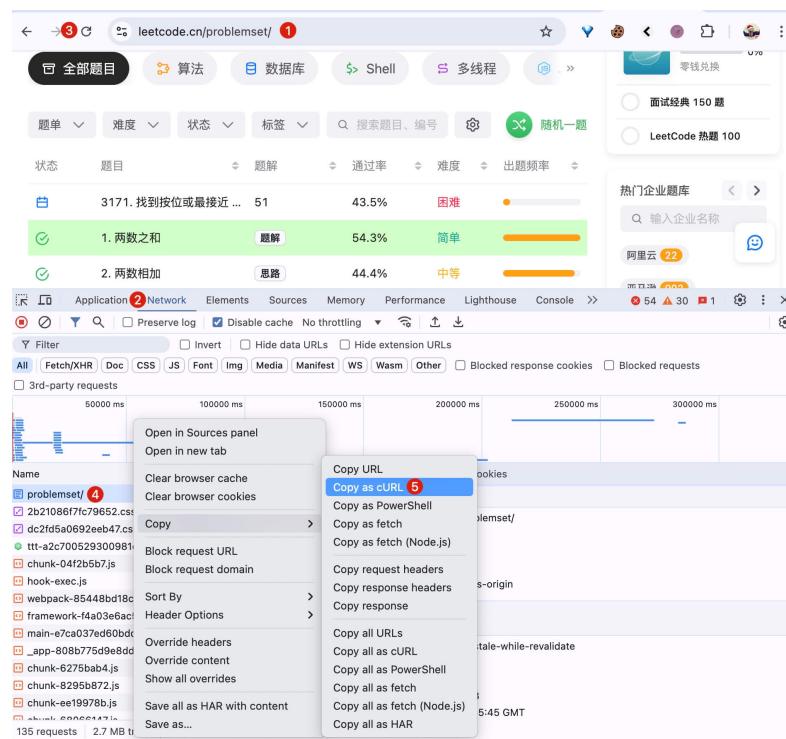
这两款插件各有各的优点和缺陷，我尽可能地取长补短，把它们的优势集合起来，并额外添加了一些好用的功能，下面简单介绍一下。

## 登录 力扣/LeetCode 账号

点击侧边栏的插件图标，点击登录图标，会引导你选择登录中文力扣或英文 LeetCode：



接下来按照提示输入力扣/LeetCode 的 cURL 命令，即可完成登录。



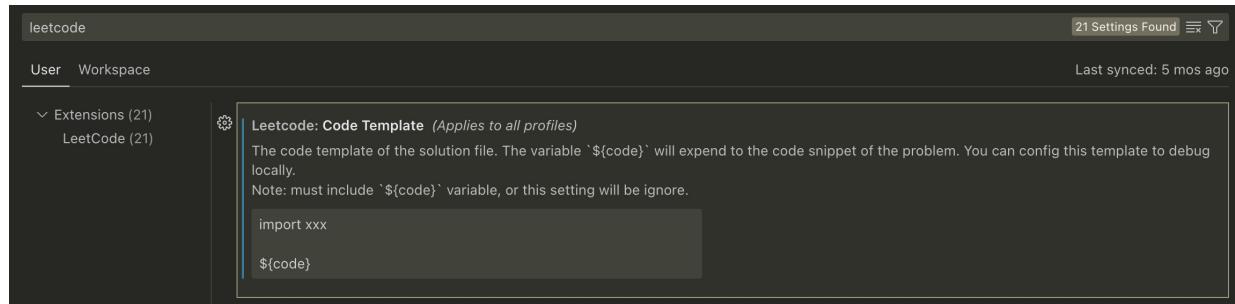
- 1 打开官网 <https://leetcode.cn/problemset/>（国际版为 <https://leetcode.com/problemset/>），并确保已经登录你的账号。
- 2 打开浏览器开发者工具（Chrome/Edge 按 F12），然后点击 Network 选项卡。
- 3 ~ 4 刷新页面，选中第一个网络请求，鼠标右键选择 Copy -> Copy as cURL。

若出现登录报错，大概率是浏览器给你的 cURL 命令有问题，可以尝试把复制出来的 cURL 命令粘贴到终端中执行，看看是否正常。

非主流的浏览器会有各种奇怪的问题，建议使用 Chrome/Edge 浏览器进行操作。

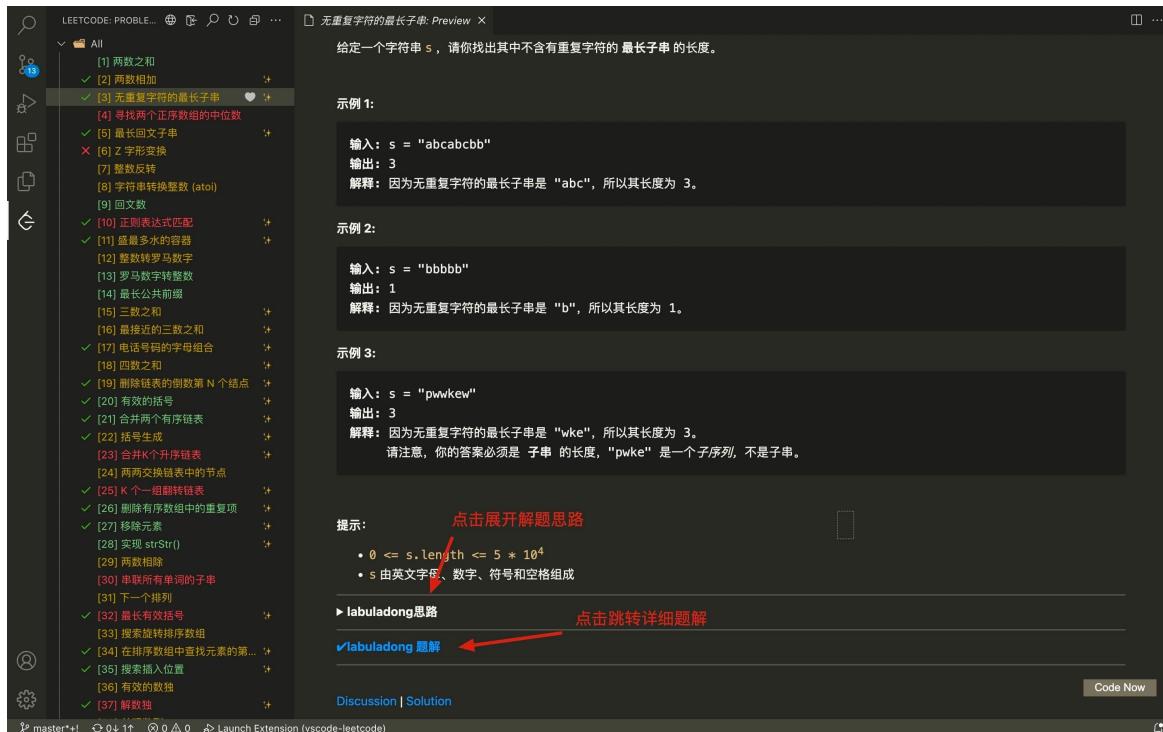
## 自定义代码模板

首先一个实用功能是可以配置代码模板，这样你就可以很方便地在本地编译和调试代码了：



## 题解/思路辅助刷题

另外一个重要的功能，是方便读者查看我的解题思路和详细题解。题目列表中带有 标记的都是我讲解过的题目，点开后可以查看我的题解或者思路：



你可以一边写代码一边查看我的思路讲解：

The screenshot shows a LeetCode problem titled "105.从前序与中序遍历序列构造二叉树" (Construct a binary tree from pre-order and in-order traversal). The code is written in Java and defines a `TreeNode` class and a `Solution` class with a `buildTree` method. To the right of the code is a diagram illustrating the mapping between pre-order and in-order traversal results. The pre-order traversal is shown as [1, 2, 5, 4, 6, 7, 3, 8, 9], where the root node is 1. The in-order traversal is shown as [5, 2, 6, 4, 7, 1, 8, 3, 9]. The diagram shows how the root node's value (1) is identified in the in-order sequence, and then the left and right subtrees are constructed based on the remaining elements.

我自己一直在坚持刷算法题并总结解题套路，所以插件中标 的题目将会越来越多，所有配套插件都会同步更新。

## 代码图片注释

「labuladong 思路」中的代码支持直接复制，且代码中的小灯泡图标会弹出图片辅助理解代码逻辑：

The screenshot shows a LeetCode problem titled "21.合并两个有序链表" (Merge Two Sorted Lists). The code is written in Java and defines a `ListNode` class and a `Solution` class with a `mergeTwoLists` method. To the right of the code is a diagram illustrating the merging of two sorted linked lists. It shows two lists, `l1` and `l2`, being merged into a new list. A dummy node `dummy` is used to simplify the head pointer. The `p` and `q` pointers are used to traverse the lists. Red arrows and boxes highlight specific parts of the code and the list nodes to aid in understanding.

## 算法可视化面板

「labuladong 思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：

The screenshot shows the LeetCode extension's visualization panel. On the left, there is a code editor with Java code for removing the nth node from the end of a singly-linked list. On the right, there is a visualization panel titled 'Description' which shows a diagram of a linked list with nodes labeled -1, 1, 2, 3, 4, 5. Two pointers, p1 and p2, are shown. p1 starts at node -1 and moves k steps forward. p2 starts at node -1 and moves n-k steps forward. A callout box highlights the function `removeNthFromEnd` with parameters `head = -1` and `n = 2`. Below the visualization, there is a list of similar problems.

```

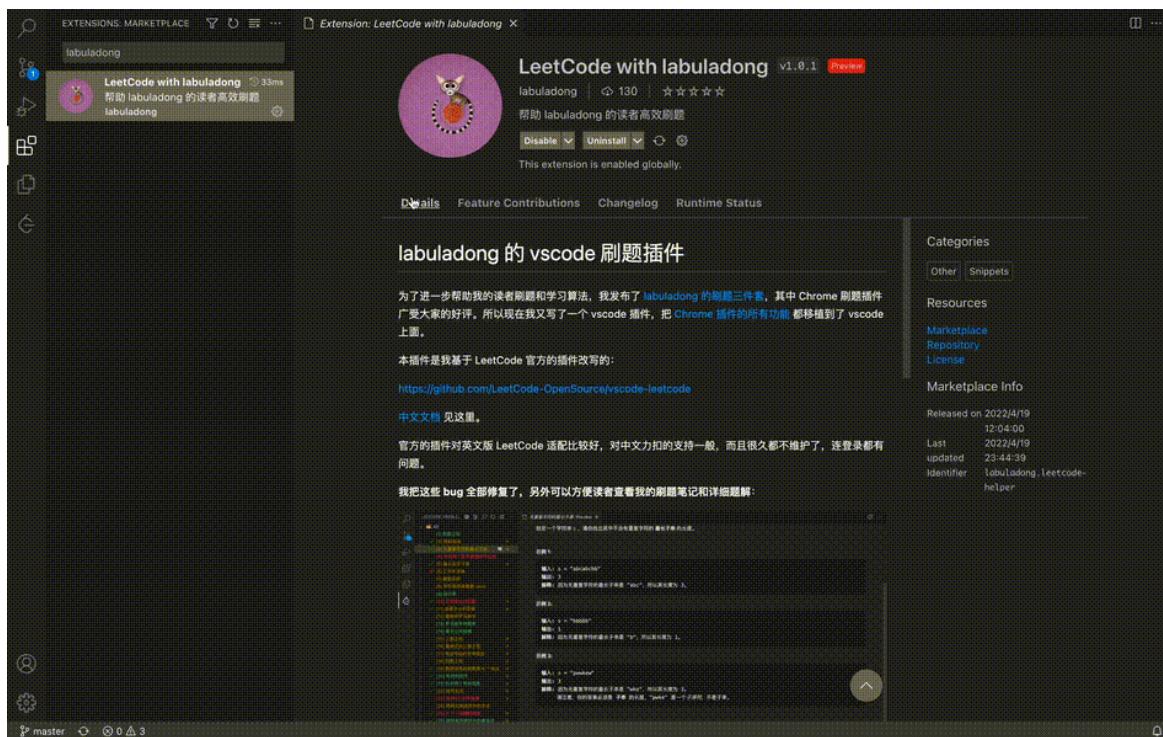
19.删除链表的倒数第-n个结点.java
Users > labuladong > .leetcode > 19.删除链表的倒数第-n个结点.java
15   * ListNode(int val, ListNode next) { this.val = val; this.next = next; }
16   *
17   */
18
19 class Solution {
20     // 主函数
21     public ListNode removeNthFromEnd(ListNode head, int n) {
22         // 虚拟头节点
23         ListNode dummy = new ListNode(-1);
24         dummy.next = head;
25         // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
26         ListNode x = findFromEnd(dummy, n + 1);
27         // 删掉倒数第 n 个节点
28         x.next = x.next.next;
29         return dummy.next;
30     }
31
32     // 返回链表的倒数第 k 个节点
33     ListNode findFromEnd(ListNode head, int k) {
34         ListNode p1 = head;
35         // p1 先走 k 步
36         for (int i = 0; i < k; i++) {
37             p1 = p1.next;
38         }
39         ListNode p2 = head;
40         // p1 和 p2 同时走 n - k 步
41         while (p1 != null) {
42             p2 = p2.next;
43             p1 = p1.next;
44         }
45         // p2 现在指向第 n - k 个节点
46         return p2;
47     }
48 }

Description | Submit | Test | 题解 | 思路 | 49 // @lc code=end
50
51

```

可视化面板的具体功能请参考 [算法可视化面板简介](#)。

用一个简短的 GIF 看下这个插件有多好用：



## 国际用户切换英文

插件的题解/思路/可视化面板的内容默认为中文。国际用户如果需要切换为英文，可以在 vscode 的设置页面中搜索配置关键词 `labuladongLanguage`，将中文改为 English 即可。

## 安装方法

微软插件商店在国内也可以正常访问，直接在 vscode 的插件商店中搜索关键词「labuladong」即可搜到插件，点击安装即可：



如果搜不到，可以从 vscode 插件商店的网页安装：

<https://marketplace.visualstudio.com/items?itemName=labuladong.leetcode-helper>

## 插件配置

在 vscode 的设置页面中搜索关键词 `labuladong`，可以看到插件相关的所有配置，可以根据自己的需求进行修改。

## 更新方式

默认情况下，vscode 会自动检测和更新插件。

## 更新日志

详见 [vscode 插件更新日志](#)。

## 在插件中解锁本站习题讲解

本站习题章节中的所有题目也可以在插件中学习，不过需要你购买本站会员后 **手动刷新数据** 才能解锁对应习题的题解/思路/可视化等功能，具体操作方法参见 [会员购买页](#) 下方。

## 常见问题解决方法

### 没有 ✨ 标记和题解/思路按钮？

如果没有看到题解/思路按钮，大概率是因为插件数据拉取失败了，你可以手动触发数据拉取，步骤如下：

- 1、按下快捷键 **F1**，会弹出 vscode 插件命令输入框。
- 2、在输入框输入 `labuladong`，就会查到一个 `Refresh labuladong.onlin data` 的命令。
- 3、点击这个命令，即可手动触发数据拉取。拉取完成后，题目列表中应该就会出现 ✨ 标记，对应题目就出现题解/思路按钮了。

### 设置代码文件的名字和路径？

插件支持按照不同的编程语言设置对应的代码文件的存储名称。

在 vscode 的设置页面中搜索配置关键词 `labuladong-leetcode filepath`，可以看到一个 `Edit in setting.json` 选项，点击后可以把你需要的配置写入 `settings.json` 文件中。

比如可以如下设置 python3 的代码文件的命名规范：

```
"labuladong-leetcode.filePath": {
    "python3": {
        "filename": "${id}.${cn_name}.${ext}"
    },
    // ...
}
```

可用的变量有：

题号	题目英文名	题目中文名	扩展名	当前时间日期	驼峰名称	下划线名称	短横线名称
\${id}	\${name}	\${cn_name}	\${ext}	\${yyyymmdd}	\${camelCaseName}	\${snake_case_name}	\${kebab-case-name}

## bug 反馈

可以在 GitHub 创建 Issue 反馈问题：

<https://github.com/labuladong/fucking-algorithm/issues/>

# 配套 JetBrains 刷题插件

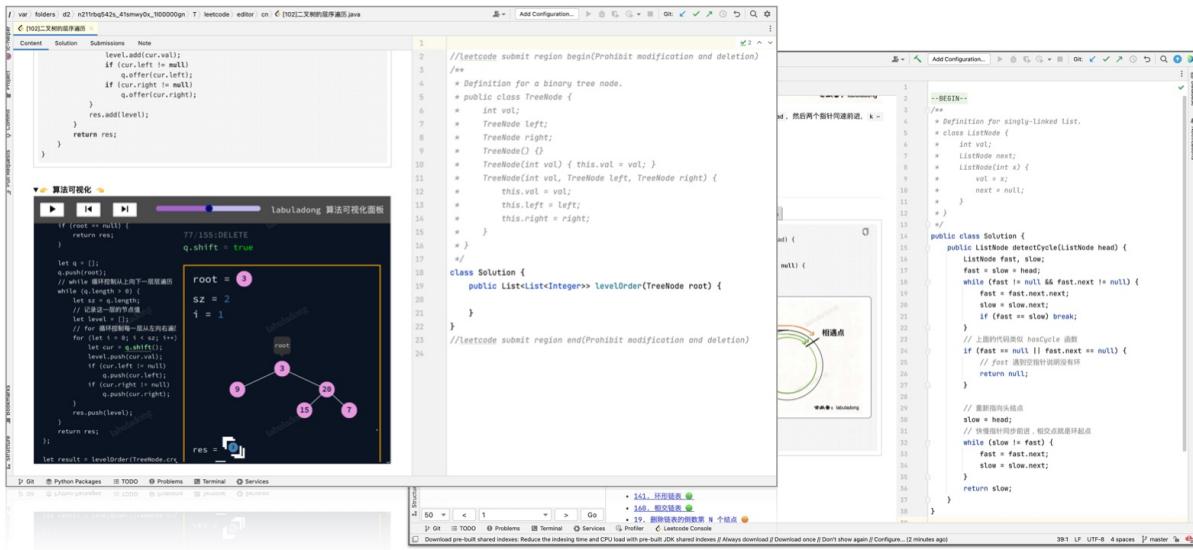
插件中的题解和思路都是本站的文章和习题讲解，换句话说，本站包含全部算法教程和习题，并没有什么内容是插件独有的。

只不过有些读者不喜欢在网页上刷题，认为在编辑器或 IDE 中刷题更方便编写和调试代码，所以我开发维护了各个平台的刷题插件满足这个需求，同时允许读者在插件中查看本站对题目的解法和思路。

综上，刷题插件只是辅助大家学习本站的配套工具，并不是必须安装的，大家可以根据需求自行选择。

JetBrains 刷题插件可以在 JetBrains 的所有 IDE（比如 IntelliJ, Pycharm 等等）中刷 LeetCode/力扣，同时可以查看本站习题章节所有题目的解题思路：

labuladong 的 JetBrains 刷题插件：添加「思路」、「题解」按钮，支持算法代码的可视化



## 使用指南

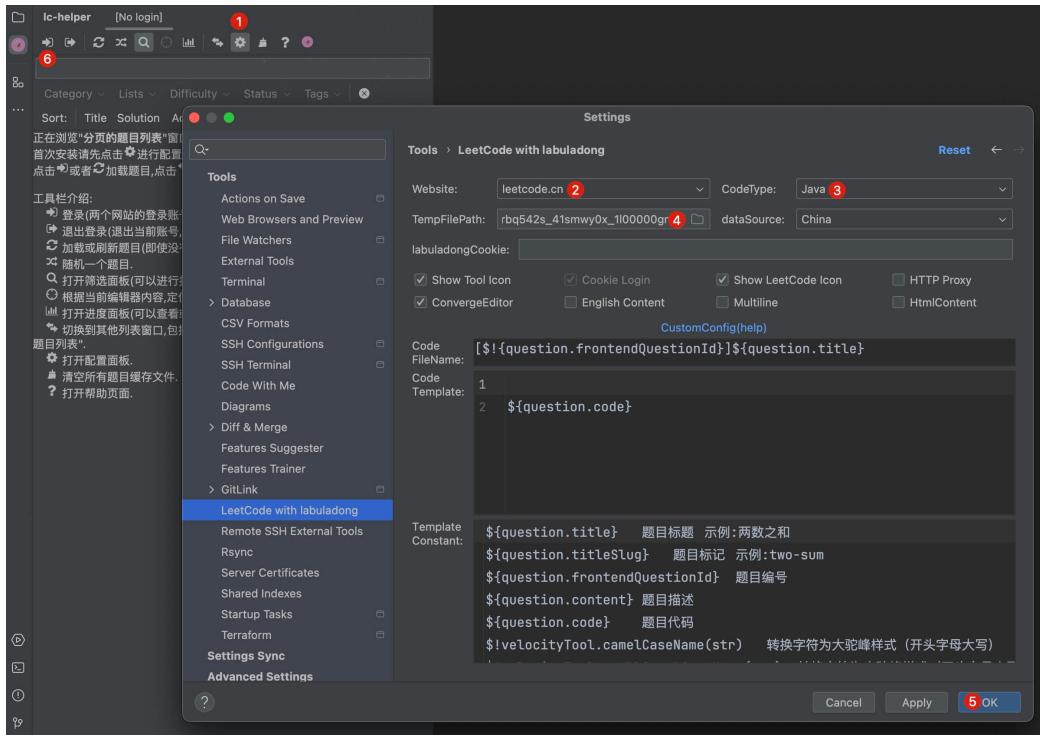
本插件基于开源插件 LeetCode Editor 开发（感谢开源作者 [@shuzijun](#)）：

<https://github.com/shuzijun/leetcode-editor/>

下面介绍一下本插件的基本的使用和我添加的辅助刷题功能。

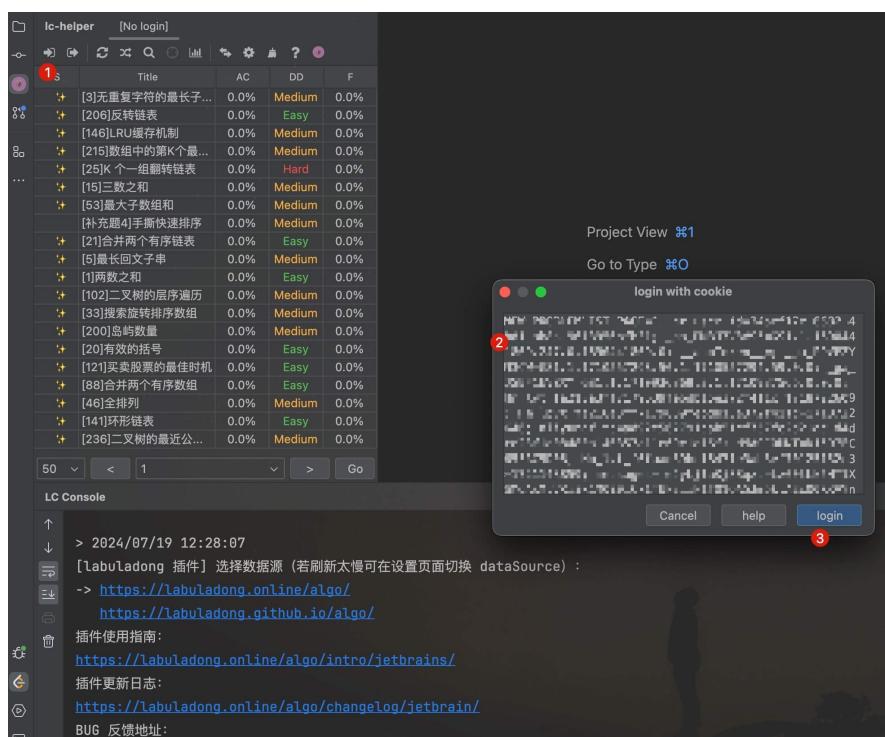
## 登录 力扣/LeetCode 账号

安装插件之后，侧边栏将出现插件图标，点击插件设置按钮，按下图进行设置：

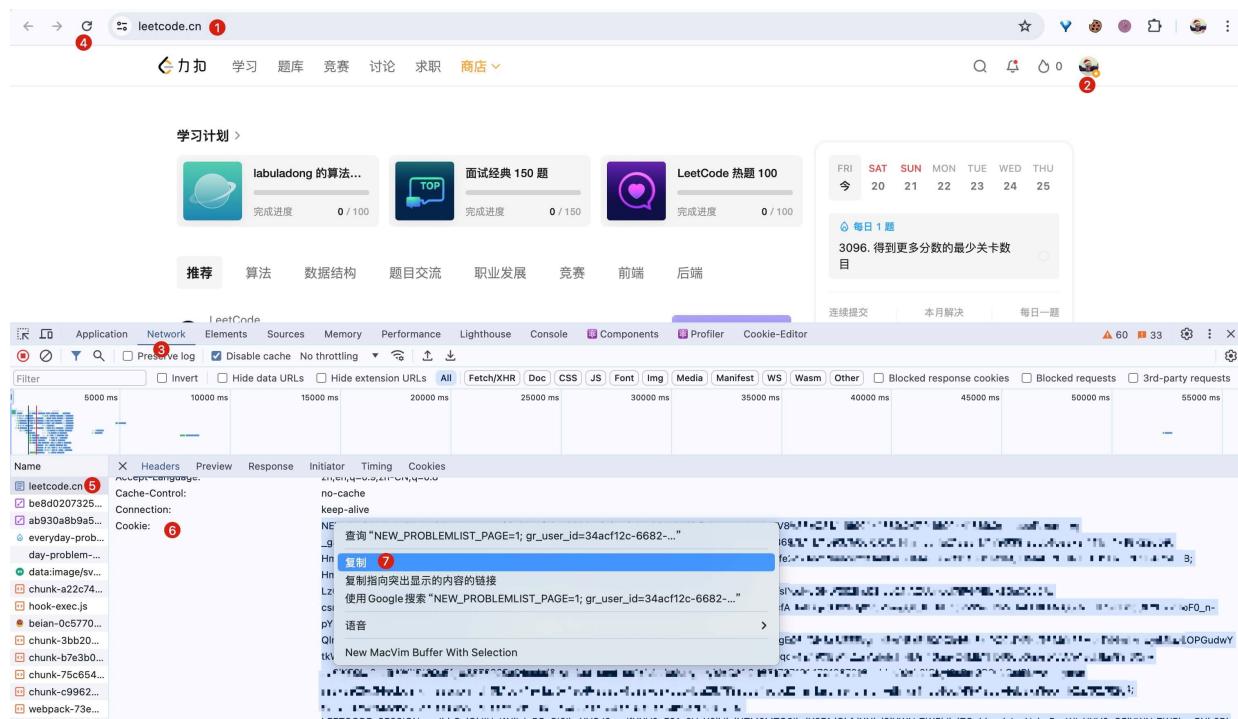


- 1 点击设置按钮，进入设置页面。
- 2 选择登录的站点，可以是英文版 LeetCode 或中文版力扣。
- 3 设置你希望使用的编程语言。
- 4 设置题目代码文件的存储路径。
- 5 设置完成后点击 OK 按钮。
- 6 点击登录按钮。

点击登录按钮之后，会弹出一个对话框让你输入 cookie。如果你在设置的登录英文版 LeetCode，则输入 leetcode.com 的网站 cookie，如果设置的是登录中文版力扣，则输入 leetcode.cn 的网站 cookie：



中文力扣 leetcode.cn 的 cookie 的获取方法如下，浏览器打开中文力扣官网 <https://leetcode.cn>，确保已经登录你的力扣账号，打开开发者工具（Chrome 浏览器可以按 F12 打开），复制 cookie 的值：



1 ~ 2 打开中文力扣的官网 <https://leetcode.cn> 并确保已经登录你的账号。

3 打开开发者工具（Chrome 按 F12），点击 Network 选项卡。

4 ~ 6 刷新页面，点击第一个请求，查看该请求的 Headers，其中有一个 Cookie 字段。

7 全选整个 cookie 字符串，右键复制。

获取英文 LeetCode 的 cookie 方法是类似的，只需打开英文版 LeetCode 的官网 <https://leetcode.com>，重复上面的操作即可，这里不再赘述。

把 cookie 粘贴到插件并点击登录，应该就可以成功在插件中登录你的力扣/LeetCode 账号，开始刷题了。

## 本地调试代码

着重强调一下自定义代码模板的功能，这个功能可以最大化发挥出 IDE 的优势：

- 可以自动给代码文件添加 `main` 函数。
- 可以自动修改文件名/类名为题目的英文名，方便组织和查找。
- 可以自动声明 LeetCode 中的内置的 `TreeNode`, `ListNode` 等数据结构。
- 利用 IDE 的自动补全和 debug 调试功能。

详细配置方法和案例项目 [见这里](#)。

除了 LeetCode Editor 的基本功能，我还添加了很多实用的功能，下面简单介绍一下。

## 题解/思路辅助刷题

登录成功后，有我的题解或思路的题目会标记 ✨：

STAT	Title	AC	DD	F
⭐	[15]三数之和	35.6%	Medium	9,13...
⭐	[16]最接近的三数之和	45.6%	Medium	8,32...
⭐	[17]电话号码的字母...	57.8%	Medium	6,984.5
⭐	[18]四数之和	39.3%	Medium	6,54...
⭐	[19]删除链表的倒数...	44.1%	Medium	7,39...
⭐	[20]有效的括号	44.6%	Easy	8,61...
⭐	[21]合并两个有序链表	66.7%	Easy	8,23...
⭐	[22]括号生成	77.5%	Medium	8,91...
⭐	[23]合并K个升序链表	57.0%	Hard	7,32...
⭐	[24]两两交换链表中...	70.9%	Medium	6,91...
⭐	[25]K个一组翻转链表	67.5%	Hard	8,13...

点开带 标记的题目详情页会显示「labuladong 题解」和「labuladong 思路」：

The screenshot shows a Java code editor within the IDE. The code is for merging two sorted linked lists. The 'labuladong' section contains two parts: '题解' (Solution) and '思路' (Thoughts). A red box highlights the '题解' section. Below it, there's a diagram illustrating the merge process using three pointers: p1, p2, and p.

```
1 //leetcode submit region begin(Prohibit modification and deletion)
2 /**
3  * Definition for singly-linked list.
4  * public class ListNode {
5  *     int val;
6  *     ListNode next;
7  *     ListNode() {}
8  *     ListNode(int val) { this.val = val; }
9  *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
10 }
11 */
12 class Solution {
13     public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
14         if (list1 == null) return list2;
15         if (list2 == null) return list1;
16         if (list1.val <= list2.val) {
17             list1.next = mergeTwoLists(list1.next, list2);
18             return list1;
19         } else {
20             list2.next = mergeTwoLists(list1, list2.next);
21             return list2;
22         }
23     }
24 }
25 //leetcode submit region end(Prohibit modification and deletion)
```

通知: JetBrains 刷题插件 bug 反馈 [点这里](#)。

⭐ labuladong 题解

▼ labuladong 思路

### 基本思路

经典算法题了，[双指针技巧](#) 用起来。

## 代码图片注释

「labuladong 思路」中的代码支持直接复制，且代码中的小灯泡图标会弹出图片辅助理解代码逻辑：

```

public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        // 上面的代码类似 hasCycle 函数
        if (fast == null || fast.next == null) {
            // fast 遇到空指针说明没有环
            return null;
        }

        // 重新指向头结点
        slow = head;
        // 快慢指针同步前进，相交点就是环起点
        while (slow != fast) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}

```

类似题目：

- 19. 删除链表的倒数第 N 个结点
- 21. 合并两个有序链表
- 23. 合并K个升序链表

## 算法可视化面板

「labuladong 思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：

```

int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    return 1 + Math.max(leftMax, rightMax);
}

class Solution {
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }

    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        maxDiameter = Math.max(maxDiameter, leftMax + rightMax);
        return 1 + Math.max(leftMax, rightMax);
    }

    maxDepth(root);
    return maxDiameter;
}

```

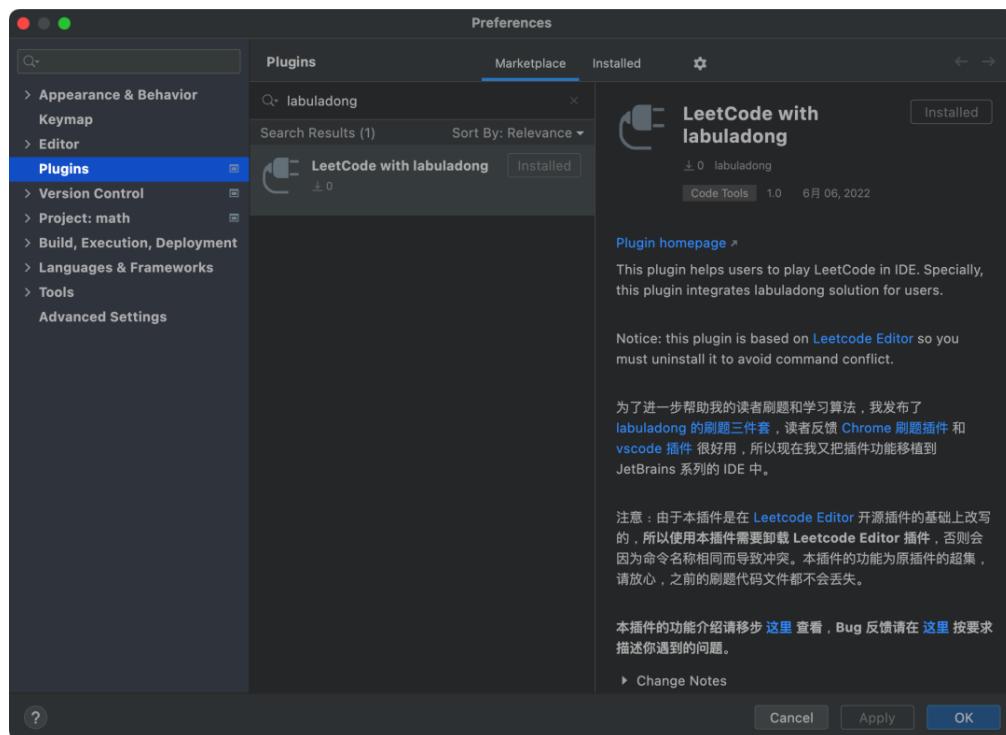
可视化面板的具体功能请参考 [算法可视化面板简介](#)。

## 国际用户切换英文

插件的题解/思路/可视化面板的内容默认为中文。国际用户如果需要切换为英文，可以在插件的设置页面中将 Language 选项改为 English 即可。

## 安装方式

我的插件全名为「**LeetCode with labuladong**」，在 JetBrains 系列 IDE 的插件商店中搜索关键词「labuladong」即可下载：



如果无法搜索到插件，可能是网络的问题。可以在 JetBrains 插件网页端按照指示下载安装：

<https://plugins.jetbrains.com/plugin/19317-leetcode-with-labuladong>

## 更新方式

JetBrains 家的 IDE 会自动检测更新，有更新时会有提示。建议及时更新最新版，以获得最流畅的体验。

## 更新日志

详见 [Jetbrain 插件更新日志](#)。

## 在插件中解锁本站习题讲解

本站习题章节中的所有题目也可以在插件中学习，不过需要你购买本站会员后手动刷新数据才能解锁对应习题的题解/思路/可视化等功能，具体操作方法参见 [会员购买页](#) 下方。

## 常见问题解决方法

### 如何本地调试代码

需要利用自定义代码模板的功能，请仔细参考上面使用指南的「本地调试代码」部分。

### 中文乱码？

对于比较新的 IDE 版本和操作系统来说，一般不会出现这个问题。如果出现了，可以参考 [这个帖子](#) 修改 IDE 的编码为 utf-8。

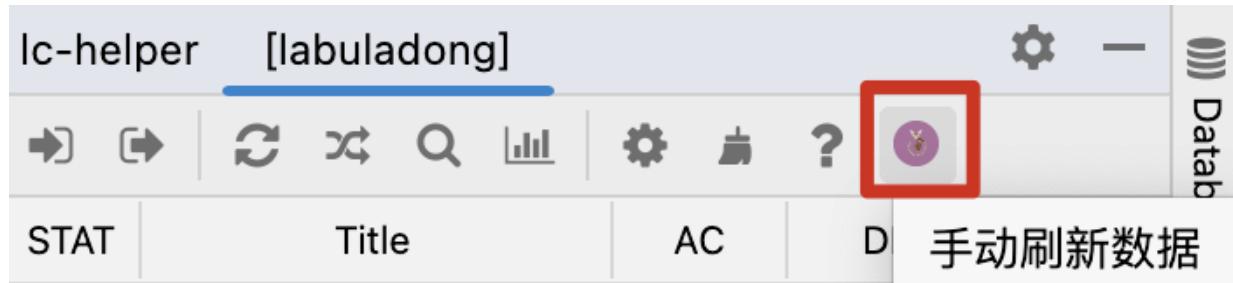
### 题目页面会突然变成空白？

你的 IDE 是否已经安装使用了很久？只有 IDE 重度使用者才会遇到这个 bug，原因很难排查，大概是 IDE 升级过程中的某些缓存/内部配置的问题导致的。一个最简单的解决办法是：去官网重新下载最新版 IDE，覆盖安装本地的 IDE（不用担心，

并不会覆盖已有的配置），这样以来所有问题都消失了。

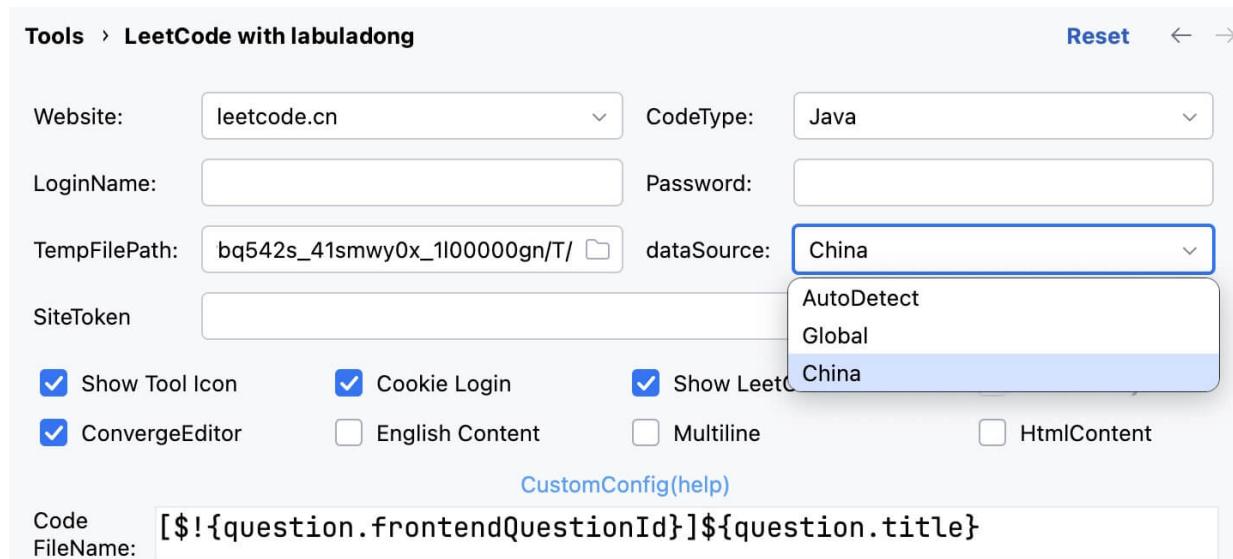
## 题目列表没有 ✨ 标记？

如果登录账号超过一分钟，题目列表还没有出现 ✨ 标记，可以点击工具栏最右侧的网站 logo 图标手动刷新 labuladong 的题解数据：



等待十几秒左右就会看到通知栏显示「手动刷新 labuladong 数据成功」，✨ 标记就会出现。

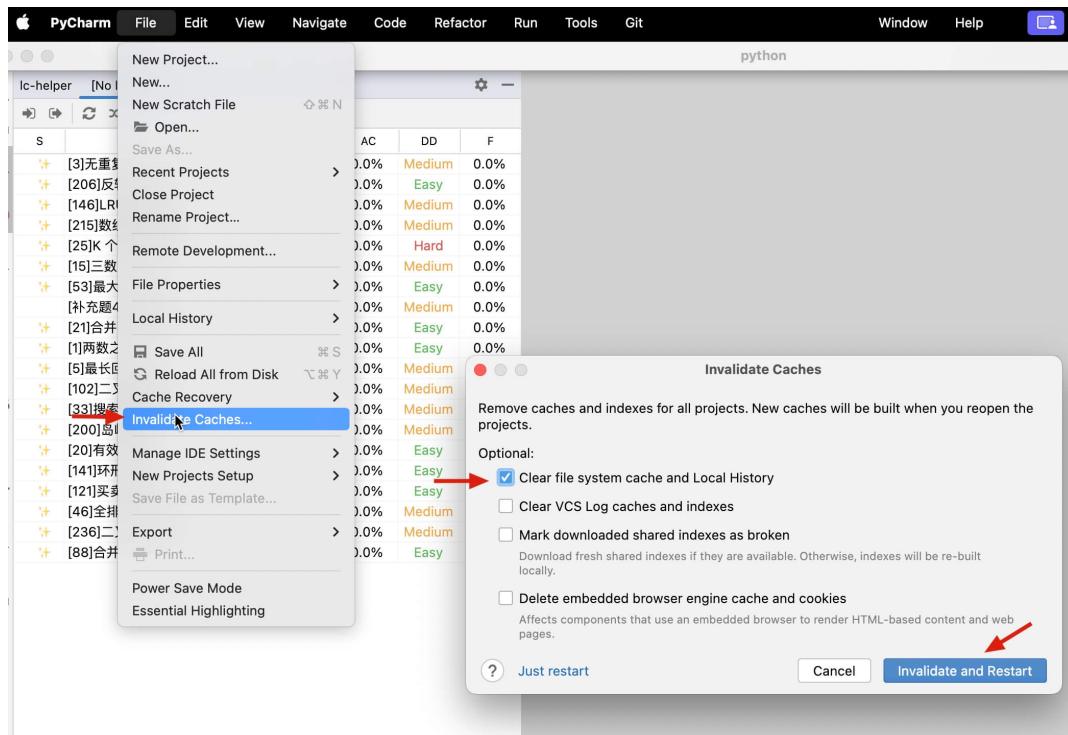
如果依然没有出现，大概率是网络问题。可以在设置中修改 `dataSource`（如果设置了数据源为 China，可能需要关闭网络代理）：



## 有 ✨ 标记但是没有思路和题解按钮？

如果显示数据拉取成功，但是题目详情页没有「labuladong 题解」和「labuladong 思路」按钮，是因为 IDE 的文件系统缓存没有刷新。

你可以尝试手动清除 IDE 的缓存并重启 IDE：



## 代码不会自动补全/纠错？

代码补全和纠错是 IDE 的基础功能，和插件无关。如果没有代码补全和纠错功能，一般是因为你的 IDE 没有配置好，或者是你的代码文件无法被 IDE 识别。

比如对于 Go 语言来说，需要在代码文件的第一行加上 `package main` 才能被 Goland IDE 识别，你可以使用前面介绍的「本地调试代码」功能来自动添加这一行。

类似的，再比如 Java 文件，需要在一个 Java 项目中才能被 IDE 识别。你可以修改插件配置，把 `TempFilePath` 设置为你的项目路径，并设置 `Code Template` 自动添加 package 名称，这样代码文件就会被保存到项目中，IDE 就能识别并给出代码补全了。

## 没有代码提交按钮？

旧版本插件可能出现这个问题，请确认是否使用的是最新版插件。正常来说，鼠标移动到代码编辑区域，右上角就会出现提交、测试等功能按钮。

同时，在代码文件中点击右键，也会出现提交、测试等功能按钮。

## bug 反馈

可以在 GitHub 创建 Issue 反馈问题：

<https://github.com/labuladong/fucking-algorithm/issues/>

# 算法可视化面板使用说明

每道题的可视化代码我都提前写好了，甚至我会在文章或者注释中告诉你应该如何操作可视化面板来观察算法的执行过程。

所以对于初学者和希望速成的读者，了解本文第一部分「基本用法」中的内容就够用了。

有些读者希望修改我的预设代码，或者对自己的一些奇思妙想进行可视化验证，那么就需要了解可视化面板的特殊能力，可以参考本文后半部分的内容。

可视化面板目前只支持 JavaScript 代码。如果你阅读可视化面板中的 JavaScript 代码有困难，我专门写了一个使用可视化面板的 [极简 JavaScript 教程](#)，帮助你 5 分钟上手。

可视化面板编辑器网页地址：

<https://labuladong.online/algo-visualize/>

另外，本站以及所有配套插件中嵌套的算法可视化面板也添加了「编辑」按钮，点击后即可直接修改并执行你的代码进行可视化。

## 基本用法

The screenshot shows the labuladong algorithm visualization interface. On the left is the code editor with the following JavaScript code:

```
// 这个函数生成所有长度为 n 的二进制数
var generateBinaryNumber = function(n) {
  let res = []
  let track = ""

  // @visualize status(track)
  var backtrack = function(i) {
    console.log(`进入 backtrack(${i})`);
    if (i == n) {
      res.push(track)
      console.log(`离开 backtrack(${i})`);
      return
    }
    // 跳转到上次执行 step:102
    // 跳转到下次执行 step:146
    for (let val of [0, 1]) {
      // @visualize choose("第 " + i + " 位选择 " + val)
      track = track + val
      backtrack(i + 1)
      // @visualize unchoose()
      track = track.substring(0, track.length - 1)
    }
    console.log(`离开 backtrack(${i})`);
  }
  backtrack(0)
  return res
}

let result = generateBinaryNumber(3)

Console output
进入 backtrack(0)
进入 backtrack(1)
进入 backtrack(2)
进入 backtrack(3)
离开 backtrack(3)
进入 backtrack(3)
离开 backtrack(3)
离开 backtrack(2)
进入 backtrack(2)
进入 backtrack(3)
离开 backtrack(3)
```

On the right is the visualization panel. It shows a backtracking tree for generating binary numbers of length 3. The root node is "backtrack = ('100') i = 3". It branches into two nodes: "0" and "1". The "0" branch leads to nodes "000", "001", and "010". The "1" branch leads to nodes "011" and "100". The "100" node has a "backtrack" button. The final result "res = [000, 001, 010]" is shown at the bottom.

左侧是代码面板，当前执行到的代码行会高亮显示，点击代码可以直接运行到指定位置；鼠标悬浮到代码上可以跳转到上次执行或下次执行的位置。

点击代码运行到指定位置是最常用的，这个功能就和你在 IDE 中 debug 代码的那个「运行到光标位置」功能一样。

一般我们不会从头开始一步步播放算法的执行，而是直接点击代码跳转到某个感兴趣的位置，然后再一步步执行，观察算法的执行过程。

顶部主要是一些控制按钮和滑动条，可以精细控制代码逐步执行，编辑按钮允许你修改运行我预设的代码。

右侧是可视化区域，用来显示变量、数据结构、堆栈信息、递归树等等。鼠标悬浮到数据结构上可以查看详细信息。

左下角是 Log Console，如果你的代码中使用了 `console.log`，输出内容会被打印在这里。`console.log` 函数被我加强了，可以自动根据递归深度给输出内容加上缩进，方便你观察递归过程。

右下角有一些悬浮按钮，提供复制代码、刷新面板、全屏显示、显示/隐藏 Log Console 等功能。

了解这些就足够你玩转可视化面板了，你可以在这两个面板上玩一玩点一点，尝试一下这些功能。

---

►  代码可视化动画 

---

►  代码可视化动画 

---

初学者了解上面的内容就够用了。下面的内容介绍可视化面板的特殊功能，供有需要的读者参考。

## 数据结构的可视化

面板可以可视化数组、链表、二叉树、哈希表、哈希集合等常见数据结构，下面具体介绍一下如何创建这些数据结构。

### 标准库数据结构

类似数组、哈希表等 JavaScript 内置的数据结构，就正常初始化和操作它们即可，比如：

---

►  代码可视化动画 

---

需要着重讲的是 力扣/LeetCode 中一些特殊的数据结构，比如单链表 `ListNode` 和二叉树 `TreeNode`，它们的基本定义和用法和力扣上一样，下面提供一些例子。

### 单链表

首先说一下单链表，单链表的每个节点有 `val`, `next` 属性，和力扣上的定义完全一致：

```
// 单链表节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function ListNode(val, next) {
    this.val = (val === undefined ? 0 : val)
    this.next = (next === undefined ? null : next)
}
```

你可以用 `ListNode.create` 方法快速创建一条单链表，这是一个简单的例子：

---

►  代码可视化动画 

---

### 双链表

双链表的每个节点有 `val`, `prev`, `next` 属性，构造函数和力扣完全一致：

```
// 双链表节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function DoubleListNode(val, prev, next) {
    this.val = (val === undefined ? 0 : val)
    this.prev = (prev === undefined ? null : prev)
    this.next = (next === undefined ? null : next)
}
```

你可以用 `DoubleListNode.create` 方法快速创建一条双链表，或者手动组装双链表。这是一些简单的例子：

▶ 代码可视化动画

## 二叉树

再说一下二叉树，二叉树每个节点有 `val`, `left`, `right` 属性，构造函数和力扣保持一致：

```
// 二叉树节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function TreeNode(val, left, right) {
    this.val = (val === undefined ? 0 : val)
    this.left = (left === undefined ? null : left)
    this.right = (right === undefined ? null : right)
}
```

你可以用 `TreeNode.create` 方法快速创建一棵二叉树。注意我们用列表来表示二叉树，表示方式和 [力扣题目中表示二叉树的方式](#) 相同，这里举两个例子：

▶ 代码可视化动画

每个二叉树节点有 `val`, `left`, `right` 属性，和力扣上的定义完全一致。

## 多叉树

多叉树的每个节点有 `val`, `children` 属性，构造函数和力扣完全一致：

```
// 多叉树节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function Node(val, children) {
    this.val = val;
    this.children = children;
}
```

你可以用 `Node.create` 来创建多叉树，大家主要注意一下多叉树的表示方法，我们还是用列表来表示多叉树，使用层序遍历的顺序，并用 `null` 来分割每组子节点。具体可以参考这道力扣题目 [429. N 叉树的层序遍历](#) 的示例。

▶ 代码可视化动画

每个多叉树节点有 `val`, `children` 属性，和力扣上的定义完全一致。

## 普通二叉搜索树

可以用 [BSTMap](#) 类创建普通的二叉搜索树存储键值对，底层原理见 [动手实现 TreeMap/TreeSet](#)。

```
type compareFn = (a: any, b: any) => number

const defaultCompare = (a: any, b: any) => {
    if (a === b) return 0
    return a < b ? -1 : 1
}

abstract class BSTNode {
    abstract key: any
    abstract value: any

    abstract left: null | BSTNode
    abstract right: null | BSTNode
}

abstract class BSTMap {
    static create(elems: any[] = [], compare: compareFn = defaultCompare): BSTMap {
    }

    abstract get(key: any): any

    abstract containsKey(key: any): boolean

    abstract put(key: any, value: any): void

    abstract remove(key: any): void

    abstract getMinKey(): any | null

    abstract getMaxKey(): any | null

    abstract keys(): any[]

    abstract _getRoot(): BSTNode | null
}
```

## 红黑二叉搜索树

可以用 [RBTreeMap](#) 类创建红黑二叉搜索树存储键值对，使用案例见 [红黑树基础](#)。

```
type compareFn = (a: any, b: any) => number

const defaultCompare = (a: any, b: any) => {
    if (a === b) return 0
    return a < b ? -1 : 1
}
```

```
// 红黑树节点
abstract class RBTreeNode {
    // 节点的颜色
    static RED: string = '#b10000'
    static BLACK: string = '#464546'

    abstract key: any
    abstract value: any

    abstract color: string

    abstract left: null | RBTreeNode
    abstract right: null | RBTreeNode

    static isRed(node: RBTreeNode): boolean {
        if (!node) return false
        return node.color === RBTreeNode.RED
    }
}

abstract class RBTreeMap {
    static create(elems: any[][] = [], compare: compareFn = defaultCompare): RBTreeMap {
    }

    abstract put(key: any, value: any): void
    abstract get(key: any): any
    abstract containsKey(key: any): boolean
    abstract getMinKey(): any | null
    abstract getMaxKey(): any | null
    abstract deleteMinKey(): void
    abstract deleteMaxKey(): void
    abstract delete(key: any): void
    abstract keys(): any[]
    abstract _getRoot(): RBTreeNode | null

    // 红黑树的一些关键方法
    static makeNode(key: any, value: any, color: string): RBTreeNode {
    }

    static makeRedNode(key: any, value: any): RBTreeNode {
    }

    static makeBlackNode(key: any, value: any): RBTreeNode {
    }

    // 左旋操作
```

```

static rotateLeft(node: RBTreeNode): RBTreeNode {
}

static moveRedLeft(node: RBTreeNode): RBTreeNode {
}

// 右旋操作
static rotateRight(node: RBTreeNode): RBTreeNode {
}

static moveRedRight(node: RBTreeNode): RBTreeNode {
}

// 颜色翻转
static flipColors(node: RBTreeNode): void {
}

// 平衡操作
static balance(node: RBTreeNode): RBTreeNode {
}
}

```

## 二叉堆（优先级队列）

我在[二叉堆基础及实现](#)中就结合可视化面板给大家讲解了二叉堆的基本概念、优先级队列的代码实现以及两个核心操作上浮(swim)和下沉(sink)的实现。同时，[堆排序算法详解](#)中也用到了二叉堆的一些方法完成排序。

```

type compareFn = (a: any, b: any) => number

const defaultCompare: compareFn = (a, b) => {
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
};

// 二叉堆节点
abstract class HeapNode {
    abstract val: any;
    abstract left: HeapNode;
    abstract right: HeapNode;
}

abstract class Heap {

    // 创建一个堆，可以传入初始元素或者比较函数
    static create(items: any[] = [], fn: compareFn = defaultCompare): Heap {
        throw new Error('Method not implemented.');
    }

    // 创建一个最大堆，可以传入初始元素
    static createMax(items?: any[]): Heap {
        throw new Error('Method not implemented.');
    }
}

```

```

// 创建一个最小堆，可以传入初始元素
static createMin(items?: any[]): Heap {
    throw new Error('Method not implemented.');
}

// 创建一个二叉堆节点
static makeNode(value: any): HeapNode {
    throw new Error('Method not implemented.');
}

// 向堆顶添加元素
abstract push(value: any): void;

// 弹出堆顶元素
abstract pop(): any;

// 查看堆顶元素
abstract peek(): any;

// 获取堆的元素个数
abstract size(): number;

// 显示堆底层的数组
abstract showArray(varName?: string): any[];

// 返回堆的根节点
// 主要用来讲解二叉堆原理
abstract _getRoot(): HeapNode | null;

// 将数组转换成完全二叉树，但不会自动应用堆的性质
// 主要用来讲解堆排序
static init(items: any[], fn: compareFn = defaultCompare) {
    throw new Error('Method not implemented.');
}

// 使节点上浮，维护二叉堆性质
// 主要用来讲解堆排序
static sink(heap: Heap, topIndex: number, size: number, compare: compareFn) {
    throw new Error('Method not implemented.');
}

// 使节点下沉，维护二叉堆性质
// 主要用来讲解堆排序
static swim(heap: Heap, index: number, compare: compareFn) {
    throw new Error('Method not implemented.');
}
}

```

可视化面板默认使用二叉树的逻辑结构展示二叉堆，调用 `showArray` 方法可以同时显示二叉堆的底层数组结构。下面是一些使用示例：

### ▶ 彩虹 代码可视化动画

## 线段树

`SegementTree` 类可以创建线段树，支持 `query`, `update` 等方法，底层原理见 [线段树基础介绍](#)。

目前内置了求和线段树、最大值线段树、最小值线段树，你可以直接调用 `SegementTree.create` 方法创建自定义线段树。

```
type mergeFn = (a: any, b: any) => any

const sumMerger = (a: any, b: any) => a + b
const maxMerger = (a: any, b: any) => Math.max(a, b)
const minMerger = (a: any, b: any) => Math.min(a, b)

// 线段树节点
abstract class SegmentNode {
    abstract val: any
    abstract l: number
    abstract r: number

    abstract left: SegmentNode | null
    abstract right: SegmentNode | null
}

// 线段树
abstract class SegmentTree {

    // 创建自定义线段树
    // merger 是一个函数，用来定义线段树的合并逻辑
    // mergerName 是合并规则的名称，便于在可视化面板中的线段树节点显示
    static create(items: any[], merger: mergeFn = sumMerger, mergerName: string = 'sum'): SegmentTree {
        }

    // 创建求和线段树，支持区间求和查询和单点更新
    static createSum(items: any[]): SegmentTree {
        return SegmentTree.create(items, sumMerger, 'sum')
    }

    // 创建最小值线段树，支持区间最小值查询和单点更新
    static createMin(items: any[]): SegmentTree {
        return SegmentTree.create(items, minMerger, 'min')
    }

    // 创建最大值线段树，支持区间最大值查询和单点更新
    static createMax(items: any[]): SegmentTree {
        return SegmentTree.create(items, maxMerger, 'max')
    }

    // 更新索引 index 的值为 value
    abstract update(index: number, value: any): void

    // 查询闭区间 [left, right] 的值
    abstract query(left: number, right: number): any

    // 显示底层数组
    abstract showArray(varName: string): any[]
}

// 获取线段树的根节点
```

```
    abstract _getRoot(): SegmentNode | null  
}
```

下面的可视化面板创建了一个求和线段树，展示了线段树的逻辑结构、底层数组和基本操作 `query`, `update`:

► 😊 代码可视化动画😊

## 并查集 (Union Find) 结构

`UF` 类可以创建并查集结构，用来解决图结构的动态连通性问题，底层原理见 [并查集核心原理](#)。

由于并查集结构有多种优化方式，比如路径压缩、按权重合并等，所以 `UF` 类提供了多种初始化方式：

- `UF.createNative(n: number)`: 创建一个未经优化的并查集结构，包含 `n` 个节点。
- `UF.createWeighted(n: number)`: 创建一个权重数组优化的并查集结构，包含 `n` 个节点。
- `UF.createPathCompression(n: number)`: 创建一个路径压缩优化的并查集结构，包含 `n` 个节点。

`UF` 对象主要有如下三个方法，用来解决动态连通性问题：

- `union(p: number, q: number)`: 合并两个节点 `p` 和 `q` 所在的连通分量。
- `connected(p: number, q: number)`: 判断节点 `p` 和 `q` 是否在同一个连通分量中。
- `count()`: 返回当前并查集中连通分量的数量。

并查集的逻辑结构是一个森林（若干棵多叉树），可视化面板为了方便展示，给这个森林创建了一个虚拟节点上，其中虚拟节点是透明的，每棵多叉树的根节点显示为红色，普通节点显示为绿色。这样就可以用一棵多叉树的形式展示森林结构，同时又能方便地看到每个连通分量的根节点。

并查集底层是用一个数组来实现森林的，调用 `showArray` 方法可以让可视化面板同时显示这个底层数组。

下面是使用权重数组优化的并查集的例子：

```
abstract class UF {  
  
    // 创建一个大小为 n 的普通并查集  
    static createNaive(n: number): UF {  
          
    }  
  
    // 创建一个大小为 n 的加权并查集  
    static createWeighted(n: number): UF {  
          
    }  
  
    // 创建一个大小为 n 的路径压缩并查集  
    static createPathCompression(n: number): UF {  
          
    }  
  
    // 创建一个大小为 n 的并查集，默认使用路径压缩  
    static create(n: number): UF {  
        return UF.createPathCompression(n)  
    }  
}
```

```
// 寻找 p 的根节点
find(p: number): number {
}

// 合并 p 和 q 所在的连通分量
union(p: number, q: number): void {
}

// 判断 p 和 q 是否在同一个连通分量
connected(p: number, q: number): boolean {
}

// 返回连通分量的数量
count(): number {
}

// 显示并查集的 parent 数组
showArray(varName: string): void {
}
}
```

---

▶  代码可视化动画 

## Trie 树/前缀树/字典树

`Trie.create()` 可以创建一棵 Trie 树，支持如下方法：

- `add(word: string)`: 向 Trie 树中添加一个字符串元素。
- `remove(word: string)`: 从 Trie 树中删除一个字符串元素。
- `containsKey(word: string)`: 判断 Trie 树中是否存在一个字符串元素。
- `shortestPrefixOf(word: string)`: 查找 Trie 树中指定字符串的最短前缀。
- `longestPrefixOf(word: string)`: 查找 Trie 树中指定字符串的最长前缀。
- `hasKeyWithPrefix(prefix: string)`: 判断 Trie 树中是否存在包含指定前缀的元素。
- `keysWithPrefix(prefix: string)`: 查找 Trie 树中所有包含指定前缀的元素。
- `keysWithPattern(pattern: string)`: 查找 Trie 树中所有匹配指定通配符模式的元素，通配符 `.` 可以匹配任意字符。
- `size()`: 返回 Trie 树中的字符串元素数量。

其中蓝色的节点是普通 Trie 节点，紫色的节点是包含单词的节点，鼠标移动到节点上可以显示该节点保存的单词。下面是一个 Trie 树的例子：

---

▶  代码可视化动画 

## console.log 增强

我在很久之前分享过一种在笔试中仅使用标准输出调试递归算法的技巧，即使用额外的缩进来区分递归深度。

所以我在可视化面板中对 `console.log` 函数（JavaScript 的打印输出函数）进行了增强，方便大家更直观地观察复杂算法的输出。

具体加强如下：

- 1、如果你在可视化面板中使用 `console.log`，输出内容会被打印在左下角的 Log Console 中，且会根据递归深度自动添加缩进。
- 2、点击控制台中的输出，会有一条虚线标记对齐所有相同缩进的输出，方便你了解哪些输出是同一递归深度的。

### ▶ 代码可视化动画

代码在开始递归和结束递归的时候进行输出，可以看到左下角的控制台输出了如下内容，相同层数的递归函数中的输出都有相同的缩进，方便区分：

```
enter traverse(1)
  enter traverse(2)
    enter traverse(3)
      enter traverse(4)
      leave traverse(4)
    leave traverse(3)
  leave traverse(2)
leave traverse(1)
```

你可以点击控制台中每行输出的第一个字符 `e` 或者 `l`，可以看到一条虚线标记对齐所有相同缩进的输出，更容易看出 `enter` 和 `leave` 的对应关系。

如果不只想用自动缩进的功能，可以把代码中的 `console.log` 改成 `console._log`，这样输出就不会自动缩进了。你可以自己修改代码试一试。

## 排序算法可视化

`@visualize shape nums rect` 注释可以把数组按照元素大小转化成类似直方图的形式，有助于直观地感受数组元素的大小关系，以便观察排序算法的执行过程。使用 `@visualize shape nums cycle` 可以将数组元素转换回默认的圆形显示。

下面以 [插入排序](#) 为例：

### ▶ 代码可视化动画

## 变量绑定为数组索引

默认情况下，当变量名作为索引值访问数组时，该变量会自动绑定到数组上，数组上会出现一个游标显示该变量所指向的元素。但有些时候一个变量可能并不会访问数组元素，但我们依然希望让这个变量显示在数组上。

比如上面的插入排序可视化中的 `sortedIndex` 变量，代码中始终没有出现类似 `nums[sortedIndex]` 这样的数组访问，但我们希望 `sortedIndex` 变量在右侧可视化面板中显示，因为它是一个用来标记已排序元素和未排序元素边界的变量。

这种情况下，我们可以使用 `@visualize bind nums[sortedIndex]` 注释，将 `sortedIndex` 变量绑定到 `nums` 数组上，这样右侧 `nums` 数组上就会始终出现一个游标显示 `sortedIndex` 所在的位置。

如果希望解绑，可以使用 `@visualize unbind nums[sortedIndex]` 注释。

## 颜色系统 {#color-system}

可以用 `@visualize color` 注释来设置节点/元素的颜色，颜色是一个以 # 开头的十六进制颜色代码（Hex Color Code），比如 `#8ec7dd` 就是浅蓝色。

颜色代码中可以出现 `?`，表示一个随机的十六进制数，以此生成随机颜色，比如 `#8e??dd`。

比如上面的插入排序可视化中，我使用 `@visualize color nums[sortedIndex] #8ec7dd` 设置变量 `sortedIndex` 的颜色。即 `sortedIndex` 索引指向的数组元素会变成浅蓝色，当 `sortedIndex` 变量移到新的索引时，旧索引的元素会恢复默认颜色。

如果你想对数组的某个索引染色，不希望随着变量的移动而移动，要用 `*` 标记声明对固定元素染色，不随着变量移动。

比如 `@visualize color *nums[0] #8ec7dd` 或 `@visualize color *nums[sortedIndex] #8ec7dd`，当 `sortedIndex = 2` 时，`nums[2]` 会变成浅蓝色，无论 `sortedIndex` 的值如何变化，`nums[2]` 都会保持浅蓝色。

对于节点对象也是类似的，可以对变量染色，比如 `@visualize color root #8ec7dd`；或者对指定节点染色，比如 `@visualize color *root.left #8ec7dd`。

如果想取消设置的颜色，可以使用 `unset` 作为颜色代码，比如 `@visualize color *root.left unset` 注释，即可恢复默认颜色。

通过 `@visualize color` 注释设置颜色后，颜色代码会被显示成一个颜色选择器，你也可以点击颜色选择器来修改颜色。

## 变量隐藏和作用域提升

使用 `@visualize global` 可以将变量的作用域提升到全局，这样，一些闭包变量就可以一直显示在右侧的可视化面板中。

使用 `@visualize hide` 可以隐藏变量，这样，你可以把一些无关的数据结构隐藏起来，避免右侧可视化面板过于拥挤。

用一个例子说明上面这两种注释的用法。比如说我想实现一个简单的 `Stack` 类，实现栈的 `push`, `pop`, `peek` 三个方法，我可以利用 JavaScript 函数的闭包这样来写：

---

### ▶ 代码可视化动画

---

但是拉到最后一步可以看到，`stack` 这个变量一直作为一个 `Object` 存在于右侧的可视化面板中，并没有什么实际意义，还占很大的地方。

其实我们更关心的是那个 `items` 变量，因为我们可以观察这个数组中元素变化的情况以了解 `push`, `pop` 方法是如何工作的。

然而由于 `items` 是在 `Stack` 函数内部声明的变量，所以当代码在函数体之外执行时，`items` 变量不在当前的作用域，所以右侧面板也不会把它可视化出来。

对于这种情况，我们可以使用 `@visualize global` 注释将 `items` 变量提升到全局作用域，然后用 `@visualize hide` 注释让 `stack` 变量在右侧面板中隐藏。

这样就可以达到我们想要的效果。请你点击 `stack.push(1)`；这行代码并往下执行，可以看到 `stack` 变量不会再出现，且 `items` 数组的更新过程会显示在右侧：

---

### ▶ 代码可视化动画

---

## 回溯/DFS 递归过程可视化

递归算法是很多读者头痛的，我之前写过一篇 [从树的角度理解一切递归算法](#)，从理论上抽象出了递归算法最基本的思维模型和编写技巧。

简单来说就是：把递归函数理解成递归树上的一个指针，回溯算法是在遍历一棵多叉树，并收集叶子节点的值；动态规划是在分解问题，用子问题的答案来推导原问题的答案。

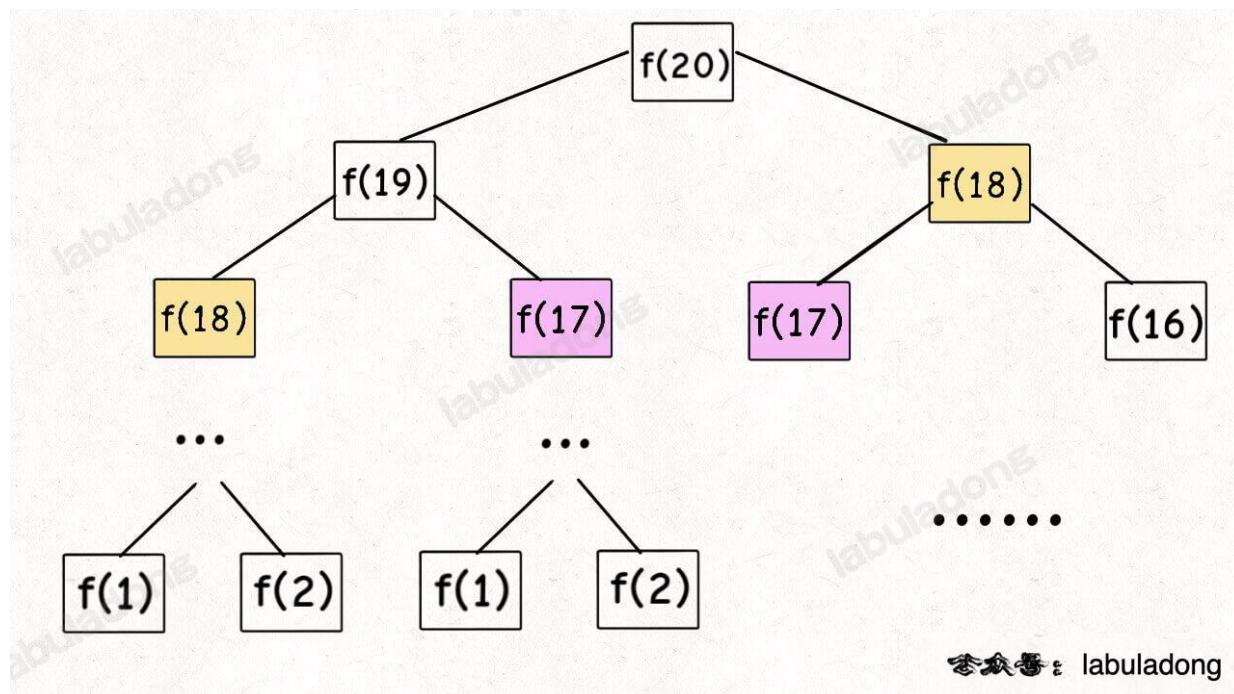
现在，我把文章中所阐述的思维模型融入了算法可视化面板，一定可以让你更直观地理解递归算法的执行过程。

对于递归算法，`@visualize status`, `@visualize choose` 和 `@visualize unchoose` 这几种注释可以帮到大忙，下面一一介绍。

## `@visualize status` 举例

一句话，`@visualize status` 注释可以放在需要可视化的递归函数的上方，用来控制递归树上节点的值。

看个最简单的例子，我在讲解 [动态规划算法核心框架](#) 时画出了斐波那契问题的递归树，上层规模较大的问题逐渐被分解：



如何描述算法运行的过程？递归函数 `fib` 就好像递归树上的一个指针，它不断把原问题分解成子问题，当问题分解到 base case（叶子节点）时，开始逐层返回子问题的答案，推导出原问题的答案。

结合 `@visualize status` 注释，可以很直观地看到这个过程。下面这个面板是斐波那契算法的递归树，正在试图计算 `fib(3)` 的值：

---

### ► 代码可视化动画

---

`@visualize status` 注释写在 `fib` 函数上边，具体含义是：

- 1、对这个 `fib` 函数开启递归树可视化功能，每次递归调用会被可视化为递归树上的一个节点，函数参数中的 `n` 的值会作为状态显示在节点上。
- 2、`fib` 函数被视为一个遍历这棵递归树的指针，处于堆栈路径的树枝会加粗显示。
- 3、如果函数有返回值，那么当函数结束，计算出某个节点返回值时，鼠标移动到这个节点上，会显示该返回值。

请你在步数栏输入 27 并敲回车，就可以跳转到第 27 步。把鼠标移动到节点 (2) 上，可以看到 `fib(2) = 1`，这说明 `fib(2)` 的值已经被计算出来了。

而处在递归路径上的 (5), (4), (3) 节点，它们的值还没被计算出来，你把鼠标移动上去也不会显示返回值。

请你尝试点击可视化面板的前进后退按钮，让算法向后运行几步，理解动态规划算法的递归树生长过程。

待到整棵递归树遍历完成之时，就是原问题 `fib(5)` 被计算出来之日，那时候整棵递归树上的所有节点都会显示返回值。

请你尝试拉动进度条到最后，看看这棵递归树的样子，并把鼠标移动到各个节点上，看看显示什么。

前面展示的斐波那契解法代码没有添加备忘录，是一个指数级时间复杂度的算法，现在我们来体验一下添加备忘录之后会对整棵递归树的生长产生什么影响。

👉 请你拉动进度条到算法的最后，看看递归树长什么样子。

#### ▶ 🔍 代码可视化动画🔍

👉 这是一个带备忘录优化的版本，请你拉动进度条到算法的最后，看看递归树长什么样子，和不带备忘录优化的递归树进行对比。

#### ▶ 😊 代码可视化动画😊

这样把整棵树可视化出来，你是不是就能很直观地理解动态规划通过备忘录消除重叠子问题的原理了？

## @visualize choose/unchoose 举例

一句话，`@visualize choose/unchoose` 注释分别放在递归函数调用之前和之后，控制递归树树枝上的值。

我编一个简单的回溯算法的题目吧，比如让你写一个 `generateBinaryNumber` 函数，生成长度为 `n` 的二进制数，比如 `n = 3` 时，生成 `000, 001, 010, 011, 100, 101, 110, 111` 这 8 个二进制数。

这道题的解法代码也不复杂，我来展示一下如何用 `@visualize choose/unchoose` 注释来可视化回溯过程的：

#### ▶ 😊 代码可视化动画😊

你可以把鼠标移动到递归树上的任意节点，就可以显示出从根节点到该节点的路径上的所有信息。

你结合这棵递归树去理解递归算法的代码，是不是就很直观了？

## BFS 过程可视化

正如 [二叉树思维（纲领篇）](#) 所说，BFS 算法是从二叉树的层序遍历扩展而来。

既然递归树都可以可视化出来，那么 BFS 过程也可以可视化出来。你可以使用 `@visualize bfs` 开启 BFS 算法的可视化，自动生成穷举树。节点上的值是队列中元素的值，用 `@visualize choose/unchoose` 注释可以控制树枝上的值。

下面我用一个简单的例子来展示 BFS 过程的可视化：

---

▶  代码可视化动画 

---

## 场景练习题

下面我会结合具体的算法题目，向你提问，要求你利用可视化面板高效分析题目和算法的执行过程。我会给出问题的答案，你可以先自己思考，然后再看我的解答。

这部分主要用到的是可视化面板的「执行到指定代码位置」的功能：你可以点击代码的某一部分，然后算法就会执行到那里，并将执行过程可视化。这个功能用得好，可以大幅提升你对算法的理解。

## 热身

首先，我带大家熟悉一下「执行到指定代码位置」的功能，在之前展示的单链表成环算法中，我其实已经带你用过了，我让你不断点击代码中的 `if (slow === fast)` 这一行，观看快慢指针的追逐过程：

---

▶  代码可视化动画 

---

为什么点击这部分就可以展示快慢指针的追逐呢？因为执行这部分代码时，`fast`, `slow` 指针刚刚移动完毕，所以点击这里，可以看到它们最完整的移动过程。

在下面的练习中，请你着重思考代码执行到某一部分时算法的状态，这样你就能更好地利用可视化面板，精确地控制算法的执行。

## 场景一：二叉树的前中后序遍历

---

▶  代码可视化动画 

---

上面这段代码是二叉树前中后序遍历，相信大家都不陌生。

请你点击播放按钮，查看这段代码的执行流程。播放时，上一步/下一步按钮将变成加速/减速按钮，用来调整播放速度。

你可以看到，二叉树将会被可视化出来，`root` 变量就好像一个指针，在二叉树上游走。我希望你能够使用「执行到指定代码位置」的功能，点击代码中的合适位置，精确控制 `root` 指针的移动顺序。

- 1、把面板复位，让 `root` 指针按照前序顺序访问整棵二叉树，即 `root` 指针按照 `1, 2, 4, 3, 5, 6` 的顺序遍历二叉树。
- 2、把面板复位，`root` 指针按照中序顺序访问整棵二叉树，即 `root` 指针按照 `2, 4, 1, 5, 3, 6` 的顺序遍历二叉树。
- 3、把面板复位，`root` 指针按照后序顺序访问整棵二叉树，即 `root` 指针按照 `4, 2, 5, 6, 3, 1` 的顺序遍历二叉树。

这几个问题不难：

- 1、不断点击 `preorderResult.push(root.val)` 这部分代码。
- 2、不断点击 `inorderResult.push(root.val)` 这部分代码。

3、不断点击 `postorderResult.push(root.val)` 这部分代码。

## 场景二：观察递归树的生长

👉 这是刚才讲到的不带备忘录的斐波那契算法，现在请结合你对递归树的理解，点击代码中的合适位置，使得每点击一次，递归树就向下生长一个节点，直到递归树完全长成。

### ▶ 🌟 代码可视化动画 🌟

你可以不断点击代码中 `if (n < 2)` 这一行，观察递归树的生长过程。

为什么点击这部分就可以展示递归树的生长过程呢？因为这部分代码是递归函数的 base case，每次递归必然都会执行这个判断逻辑，而递归树的每个节点其实就对应着每次递归，所以点击这个 if 语句就可以看到递归树上节点的生长过程。

这是一个非常常用的技巧，可以用来观察递归过程。

## 场景三：快速获得一个穷举结果

👉 这是前文展示的全排列算法的可视化面板，在场景二中你观察了斐波那契问题的递归树生长，现在请你点击代码中的合适位置，观察全排列问题的递归树生长。

### ▶ 🔎 代码可视化动画 🔎

这个很简单，不断点击 `if (track.length === nums.length)` 这部分代码即可观察递归树的生长。

将面板复位。

现在，我不想看递归树一个一个节点地生长了，没什么意思，我希望你点击代码中的合适位置，每点击一次，递归树就生长出一条「树枝」。

这是个回溯算法中很常见的场景哦，因为每条「树枝」的末端是叶子节点，而叶子节点一般就存储这回溯算法穷举出来的一个结果。比如你看这道题，每条树枝末端的叶子节点就是一个排列结果。

不断点击 `res.push([...track])` 这部分代码，每次点击，回溯树都将生长出一条「树枝」。

因为叶子节点其实就是递归树结束生长的节点，即 `backtrack` 函数的 base case。那么我们只要点击 base case 中的代码，就可以直接跳到递归树的叶子节点，也就相当于生长出了一条树枝。

## 场景四：理解自顶向下/自底向上两种思维模式

我在 [动态规划核心框架](#) 中讲过，自顶向下用 `memo` 备忘录的递归解法和自底向上用 `dp` 数组的迭代解法本质上是一样的，`dp` 数组中的值和 `memo` 备忘录中的值完全一样，两种解法可以互相转化。

下面我将同时给你自顶向下的递归解法和自底向上的迭代解法，请你完成如下题目：

- 1、在自顶向下递归解法的面板上，点击代码的合适位置，使得每次点击都在 `memo` 中更新一个元素。
- 2、在自底向上迭代解法的面板上，点击代码的合适位置，使得每次点击都在 `dp` 中更新一个元素。

3、对比两个解法中 `dp` 数组和 `memo` 数组的更新，理解自顶向下的递归解法和自底向上的迭代解法本质上是相同的。

▶  [代码可视化动画](#) 

▶  [代码可视化动画](#) 

1、在自顶向下递归解法的面板上，每次点击 `memo[n] = dp(memo, n - 1) + dp(memo, n - 2)` 这部分代码，都会在 `memo` 数组中更新一个元素。

2、在自底向上迭代解法的面板上，每次点击 `dp[i] = dp[i - 1] + dp[i - 2]` 这部分代码，都会在 `dp` 数组中更新一个元素。

3、结合可视化面板应该不难理解，不过你会发现 `memo[1]` 和 `dp[1]` 的值不一样，那是因为我们把 `dp[1]` 作为 base case 设置了初始值，而递归解法中的 base case 是直接 return 的，并不需要设置在 `memo` 中。

# 本站付费会员

以前我推出过《数据结构课程》《二叉树课程》，总是有读者搞不清楚课程和网站有什么区别，应该按照什么顺序学习。

为了避免读者有选择困难，我下线了旧版课程，对课程的内容进行全面翻新并整合到了[主站目录](#)。所以现在不再有课程的概念，只需购买 2024 新版会员这唯一一个付费项目，主站和插件全部功能都将解锁。

新用户无需再购买和学习旧课程，请购买 2024 新版网站会员后，直接在[主站目录](#)按顺序学习即可。

作为老读者的福利，拥有旧版课程/会员权限的用户可以继续学习旧版课程内容，且无需额外付费即可享有新版网站会员的所有权益，有效期按课程数量累加。如果你是 2024/2 之前购买的旧版课程，请按照[这里的步骤](#)把课程权限迁移到新网站。

比方说你在 2023/1/1 购买了《数据结构课程》一门课程，那么赠送你的新版网站会员到期时间是  $2023/1/1 + 1\text{年} = 2024/1/1$ 。

假设你在 2023/1/1 购买了《数据结构课程》，又在 2023/3/1 又购买了《二叉树课程》，那么赠送你的新版网站会员有效期是  $\min(2023/1/1, 2023/3/1) + 2\text{年} = 2025/1/1$ 。

本站同时支持中英版本，但是需要注意，[中文版本和英文版本是两个不同的付费项目，需要分别购买](#)。购买中文版不能解锁英文版，反之亦然。

英文版购买入口[在这里](#)。

## 购买会员（支持 Paypal）

购买 2024 新版网站会员，你将获得以下权益：

- 1、解锁[labuladong 的算法笔记网站](#)的所有文章学习权限，即目录中带有\*标记的教程。
- 2、解锁[labuladong 的算法笔记网站](#)目录中所有「强化练习」习题章节，即带有\*标记的章节。
- 3、解锁本站配套的[vscode 插件](#)、[Chrome 插件](#)、[JetBrains 插件](#)中所有的解题思路、代码和[可视化面板](#)的完整使用权限。
- 4、如果有需要，可以加入本站的微信讨论群。完成购买后，会显示加群的方法。

本站所有文章和习题章节中的每道题目我都会手把手教你如何思考，如何运用算法框架举一反三，能够迅速帮你练出肌肉记忆，彻底掌握算法。目前本站可以带你解决 500 道以上的算法题，我还在不断更新。

购买会员后，本站的会员专属内容会自动解锁，但是插件中的功能不会自动解锁，需要手动操作。

如果你需要使用配套刷题插件，往下翻有每个插件的详细操作方法，请注意查收。

新版网站会员限时优惠，并且会不断更新，所以会逐步涨价，越早入手越划算。购买后可解锁网站、习题、插件、可视化面板的所有内容和功能，有效期一年。只需一顿饭钱，够良心吧！

购买会员后，插件中的专属题解思路需要手动拉取数据才能解锁。下面分别介绍 Chrome 插件、vscode 插件和 JetBrains 插件的解锁方法，请根据你的使用情况进行选择。

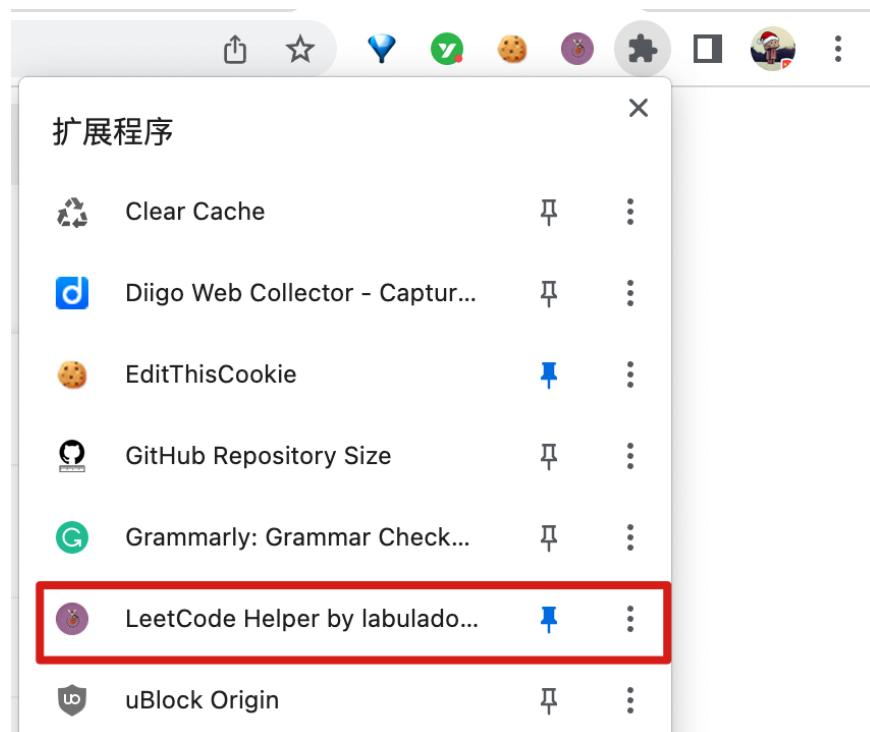
## 解锁 Chrome 插件

### 步骤一、安装并启动插件

按照 [Chrome 刷题插件安装使用指南](#) 的介绍安装插件，确保插件的基本使用没有问题。

### 步骤二、打开插件弹窗

在浏览器右上角的插件列表点击「LeetCode helper」插件的图标：



点击插件图标将会显示插件弹窗，弹窗中显示插件的数据拉取情况。未购买会员的情况下，「网站会员」这一栏会显示八叉 **×**。

### 步骤三、手动刷新数据

登录本站并购买会员后，点击插件弹窗中的「手动刷新数据」按钮并稍等几秒钟之后，网站会员的数据应该就会刷新成功，从八叉 **×** 变成对钩 **✓**：



## labuladong 的刷题插件

如果插件好用, 请推荐给你的朋友。

基本信息:			
当前版本	最新版本	使用指南	bug 反馈
4.4.3	4.4.2	<a href="#">点这里</a>	<a href="#">点这里</a>

数据拉取信息:				
元数据	思路/题解	可视化面板	all in one	网站会员
<input checked="" type="checkbox"/>				

多语言解法信息:				
java	cpp	python	javascript	go
<input checked="" type="checkbox"/>				

✓ 表示刷新成功, ✘ 表示刷新失败。如果刷新速度太慢, 可以尝试修改「数据源」选项再刷新。[all in one 网站会员](#) 的数据需要登录并购买后才能刷新成功。

V4.4.x 版本插件已放弃支持小鹏通, 若你之前在[小鹏通平台](#) 购买过我的课程, 请按照[这里的操作指南](#) 将课程权限迁移到新网站。

⚠️ 仅在插件题解/思路异常时尝试点击刷新按钮, 不要频繁点击, 否则会被限流

数据源:

## 解锁 vscode 插件

### 步骤一、安装并启动插件

按照 [vscode 刷题插件安装使用指南](#) 的介绍安装插件并登录你的力扣/LeetCode 账号, 确保插件的基本使用没有问题。

注意, 安装插件后左侧边栏会新增一个力扣图标, 你需要先点击这个图标让插件完成加载, 然后再进行下面的操作。否则会出现命令执行失败的问题。

### 步骤二、获取本站 cookie

我们需要借助本站的 cookie 辅助 vscode 插件拉取本站专属题解, 请确保你已经购买本站会员, 且已经登录本站。

访问 [labuladong.online](#) 并打开浏览器的开发者工具 (Chrome 浏览器快捷键 F12), 切换到 network, 然后刷新网页, 找到网络请求的 cookie, 点击右键进行复制:

The screenshot shows the Chrome DevTools Network tab with a list of requests. One request for 'home/' is selected. A context menu is open over the 'Cookie' field of this request, with the '复制' (Copy) option highlighted.

**本站简介**

本站的效用  
算法学习的四大坑  
坑一: 被误导, 白白浪费时间  
坑二: 胡子眉毛一把抓  
坑三: 喜欢抖机灵, 舍本逐末  
坑四: 对递归理解不透彻  
本站的实用功能  
1. 代码图片注释  
2. 支持算法可视化动画  
3. 支持阅读历史  
4. 内容互相引用, 融会贯通  
5. 支持所有主流编程语言

**本站简介**

labuladong 约 6534 字 大约 22 分钟

**本站的效用**

帮你练成框架化, 模板化的思维方式解决算法题。如果满分 100 分, 无论你的基础如何, 本站都可以在最短的时间内帮你提升至 85 分的水平。

**谁适合学习**

本站的教程老少皆宜, 适合所有算法功力小于 85 分的读者阅读。因为绝大多数人刷算法是为了面试找工作, 所以下面从校招和社招两类读者的角度讲讲本站为什么是性价比最高的选择。

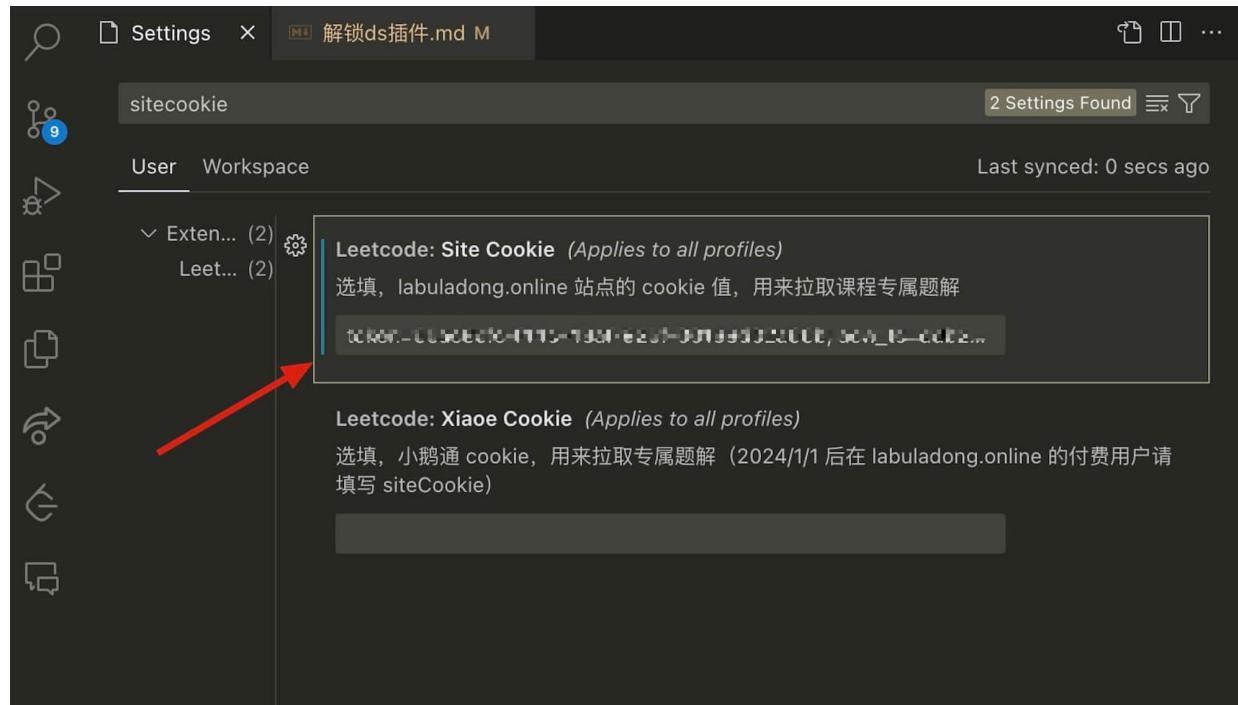
对于校招的应届毕业生, 你按照我所说的方法修炼, 从一开始就走对方向, 可以巨幅提高效率, 保你在大学期间玩也玩好, 学也学好, 俗称事业爱情双丰收 (手动滑稽)。

说正经的, 算法对校招非常重要的, 因为: 你还没有工作, 问不出什么正儿八经的项目经验, 问你八股文吧, 都是有标准答案的, 玩不出什么花活, 只有算法是唯一的变数, 可以有比较大的考察空间。所以大厂校招都喜欢考算法, 这东西很适合拿来自选校招生。

那么很多同学就保持了中学时代的题海战术, 天天刷题, 但遇到新题还是会, 为什么呢? 因为没有梳理出脉络呀, 你刷一道题就只是刷一道题, 难道你真要把题库里几千道题都做一遍背下来不成?

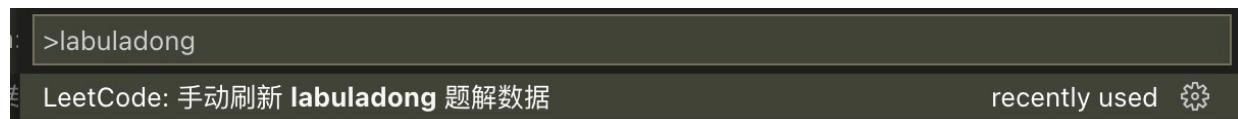
## 步骤三、在 vscode 填写 cookie

然后进入 vscode 的设置页面，搜索设置 `sitecookie`，把刚才复制的 cookie 字符串粘贴到输入框中：

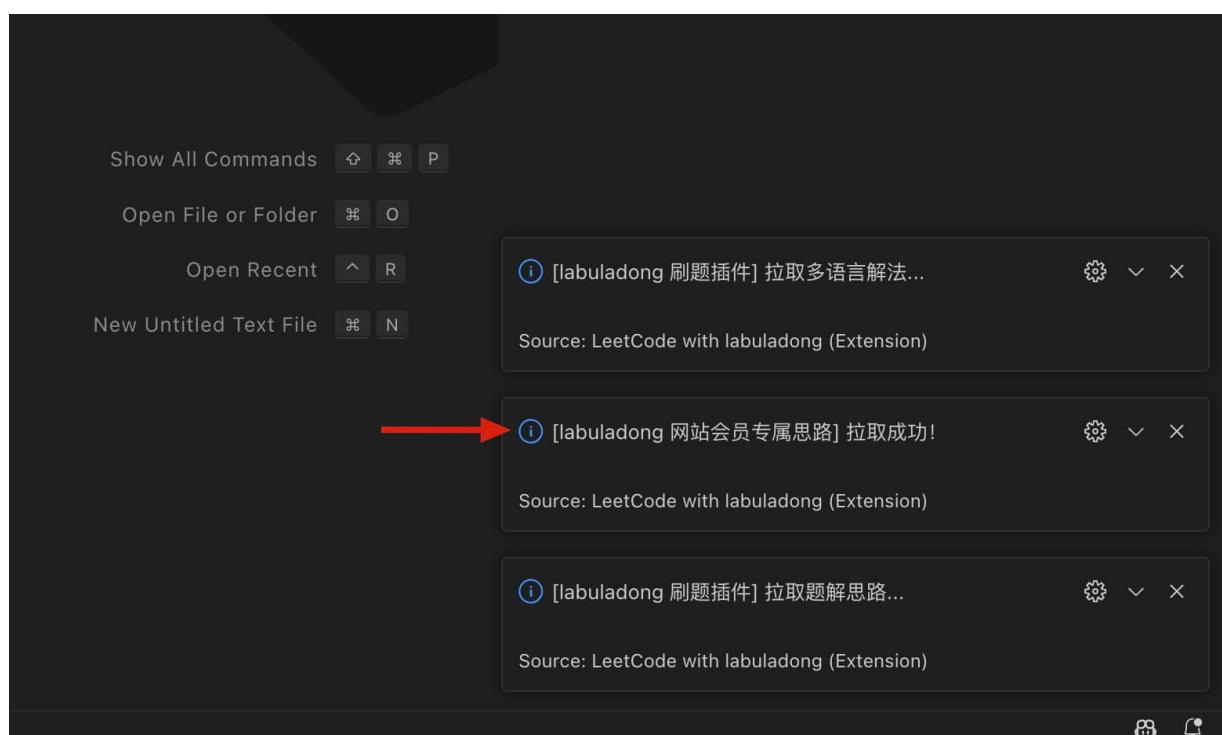


## 步骤四、手动刷新数据

在 vscode 中按下 **F1** 键，vscode 会弹出一个命令弹窗，输入关键词 `labuladong`，即可看到一个「手动刷新 labuladong 数据」的选项：



点击或回车执行刷新操作，稍等几秒钟之后应该会弹窗显示「网站会员专属思路拉取成功」：



接下来，你就可以在 vscode 中直接查看本站讲过的题目的专属题解了。

## 解锁 JetBrains 插件 {#unlock-jb-plugin}

### 步骤一、安装并启动插件

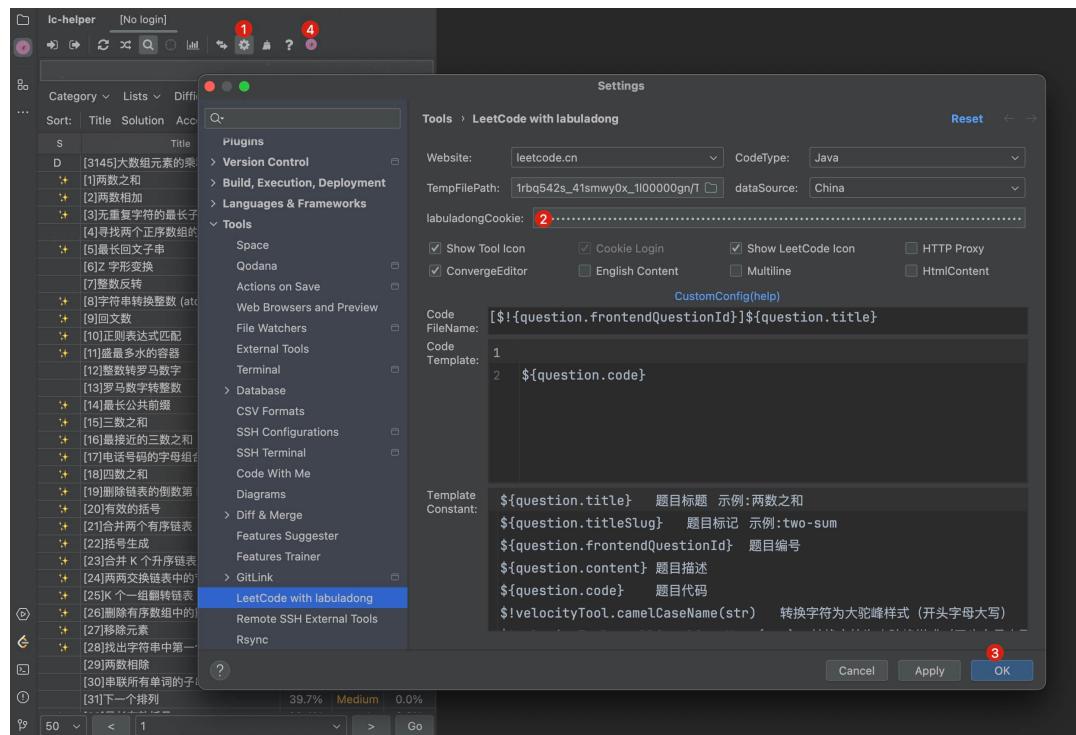
按照 [JetBrains 刷题插件安装使用指南](#) 的介绍安装插件并登录你的 力扣/LeetCode 账号，确保插件的基本使用没有问题。

### 步骤二、获取本站 cookie

和上面 vscode 插件的解锁方法的步骤二类似，请参考上面，这里就不赘述了。

### 步骤三、手动刷新数据

在浏览器中复制 cookie 后，按照下图所示的步骤操作：



① 点击设置图标，进入设置页面。

② 粘贴 cookie 到输入框中。

③ 点击 OK 按钮。

④ 点击 labuladong 网站图标，手动触发数据拉取。

插件会先拉取一些公开数据，最后会拉取本站会员专属题解数据。如果数据拉取很慢，可以尝试修改设置页面的「dataSource」选项，然后再次点击 logo 图标重新拉取数据。

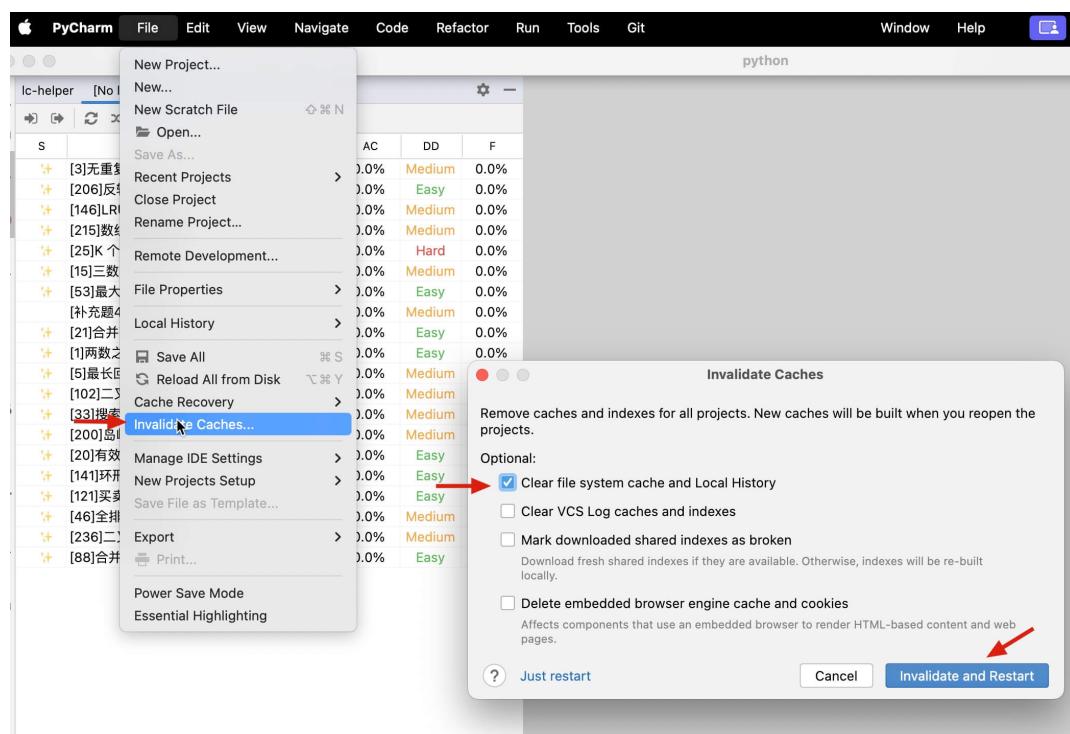
最后应该有类似下面的输出：



## 步骤四、清除 IDE 文件缓存

如果显示数据拉取成功, 应该就可以查看插件内所有的思路题解了。不过由于缓存的原因, 你之前已经打开的题目可能还是显示未解锁, 这是因为 IDE 的文件系统缓存没有刷新。

请你先关闭所有打开的题目文件, 然后在这里手动清除 IDE 的缓存并重启 IDE, 应该就正常了:



# 本章导读

---

## 本章适合谁

本章内容偏基础，适合还不太了解数据结构用法和实现原理的读者阅读。如果你已经对常见数据结构的实现原理了如指掌，可以跳过本章，直接阅读后面的章节开始刷题。

由于本章添加了很多高级的数据结构和排序算法，所以本章内容不再仅面向初学者。

对于希望系统掌握数据结构和算法的读者，都建议学习本章。

对于希望速成刷题能力应对笔试的读者，可以直接从第零章开始进入刷题环节。

## 本章导读

学习一个东西，最好的方法就是亲自动手实现它。本章不会讲解算法题，而是带读者了解所有常见的数据结构，并亲自动手实现它们。

了解了这些常见数据结构的底层原理，在后面的章节做算法题时，你才能准确利用每个数据结构的特点，并理解你写代码的时间复杂度。

本章的重点在于让读者理解每个数据结构的实现原理、优缺点和局限性，给出的 Java/C++/Golang/Python/JavaScript 代码实现只确保正确性和可读性。

至于编程语言层面的极致优化和最佳实践，不在本站的教学范围。如果你追求更深入地理解，可以参考对应编程语言的标准库。

在本章节中，会经常用到 [算法可视化面板](#) 对稍微复杂的数据结构操作进行可视化。可视化代码是用 JavaScript 写的，但是都比较简单，无论你是否了解 JavaScript 都应该很容易看懂。

# 力扣/LeetCode 解题须知

本站所有的例题、习题都选自 力扣/LeetCode 的公开免费题目，并给出题目的链接，你可以直接跳转到官网进行练习。

为了照顾初学者，下面快速介绍一下 力扣/LeetCode 上提交代码的注意事项。

## 核心代码模式

在 力扣/LeetCode 上刷题，就是给你一个函数框架，你需要实现函数逻辑，这种模式一般称之为**核心代码模式**。

比如力扣第一题两数之和，就是让你实现这样一个 `twoSum` 函数：

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        }
}
```

意思就是给你输入数组 `nums` 和目标值 `target`，你需要实现算法逻辑，最后计算返回一个数组。

你的解法代码提交到后台，它会给这个 `twoSum` 函数传入若干预定义的测试用例，对比你的返回值和正确答案，以此判断你的代码是否正确。

对于 Java/C++，有些标准库会默认导入，你不需要显式导入就可以直接用一些常用的标准库数据结构，比如 Java 的 `ArrayList`，C++ 的 `vector` 等。因为在网页上刷题时没有类似 IDE 的自动导入功能，所以这样还是比较省事的。

当然，如果你想要自动补全功能，可以考虑使用本站配套的 [Jetbrains IDE 插件](#) 或 [vscode 插件](#)。

从用户角度来说，核心代码模式应该是最方便的形式，因为你只需要专心写好算法逻辑就行了。目前大部分刷题平台和技术面试/笔试场景都是核心代码模式。但是以防万一，这里还是要介绍一下 ACM 模式。

## ACM 模式

简单说，ACM 模式就是更麻烦一些，题目给你输入的是特定格式的字符串，你需要自己解析这个字符串，然后把计算结果输出到标准输出。

你的代码提交到后台后，系统会把每个测试用例（字符串）用标准输入传给你的程序，然后对比你程序的标准输出和预期输出是否相同，以此判断你的代码是否正确。

比如牛客网就提供这种 ACM 模式，我随便截个图你看看，代码编辑框没有任何初始代码，你需要从头自己写：

The screenshot shows a programming contest interface. On the left, there's a detailed problem description with constraints (time limit: C/C++ 1s, memory limit: 64MB), a note about integer overflow, and a link to the original problem page. It also includes sections for input and output descriptions. On the right, there's a code editor window with a dark theme. The code editor shows a single digit '1' as the output. Below the code editor are buttons for 'Run Result', 'Self Test Input', a green 'Run' button with a self-test icon, and a 'Save and Submit' button.

对于 ACM 模式，一个技巧就是要对代码分层，把对字符串输入的处理逻辑和真正的算法逻辑分开，类似这样：

```
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // 主要负责接收处理输入的数据，构建输入的用例
        int[] nums;
        int target;
        int N = scanner.nextInt();
        ...
        // 调用算法逻辑进行求解
        System.out.println(twoSum(nums, target));
    }

    public static int[] twoSum(int[] nums, int target) {
        // 转化为核心代码模式，在这里写算法逻辑
    }
}
```

这样看起来就很清晰，输入处理的逻辑可以作为模板直接复制粘贴，最终转化成核心代码模式，就和你在力扣上刷题就没什么区别了。

我的建议是，平时学习时，用核心代码形式就行了，简单方便。到了面试/笔试前夕，花上一个小时就能熟悉 ACM 模式了。我记得牛客网有专门练习这种 ACM 模式的例题，你自己搜索一下即可。

## 提交代码可能出现的错误

如果提交的代码可以通过后台的所有测试用例，则称为提交成功，我们也常说 AC (Accepted)。

如果提交的代码不能通过所有测试用例，则称为提交失败，常见的失败原因有以下几种：

代码无法通过编译，这种错误一般是语法错误，比如拼写错误、缺少分号等。一般网页上手搓代码容易出现这种错误，使用本站配套 [Jetbrains IDE 插件](#) 或 [vscode 插件](#) 的话，编辑器会有基本的语法检查功能，一般不会出现这种错误。

代码可以编译通过，但是在运行时出现了数组越界、空指针异常等错误。这种错误一般是由于边界条件处理不当，你需要留意边界条件、特殊测试用例（也叫做 Corner case，比如输入为空等情况）。

代码可以编译通过，也可以运行，但是某些测试用返回的结果和正确答案不一致。这种错误一般是因为你的算法有问题，需要你根据出错的用例反思，甚至可能整个思路都错了，需要你推倒重来。

力扣预定义的测试用例中，越靠后的用例数据规模就越大。如果你的算法的时间复杂度太高，在运行这些大规模的测试用例时就容易超过系统规定的时间限制，导致超时错误。

想要解决这个问题，你需要检查以下几点：

- 1、是否有冗余计算，是否有更高效的解法思路来降低时间复杂度。
- 2、是否有编码错误，比如边界控制出错，错误地传值传引用导致无意义的数据拷贝等。

如果你卡在大规模的测试用例，一般就能说明你的算法结果是正确的，因为前面的小规模测试用例都通过了，只不过是时间复杂度太高，需要优化。

在笔试中，一般是按照通过的测试用例数量来计分的。所以如果你实在是想不出最优解法通过所有用例，提交一个暴力解，过几个用例捞点分，也是一种聪明的策略。

和超时错误类似，内存超限一般是算法的空间复杂度太高，在运行大数据规模的测试用例时，占用的内存超过了系统规定的内存限制。

想要解决这个问题，你需要检查以下几点：

- 1、是否有申请了多余的空间，是否有更高效的解法思路来降低空间复杂度。
- 2、是否在递归函数中错误地使用了传值参数导致无意义的数据拷贝。

## 力扣提交代码的注意事项

对于初次使用力扣刷题的读者，我这里再介绍几个初学者容易踩的坑。

## 不要用文件级别的全局变量

前面说了力扣这种核心代码模式，会给你的算法代码输入若干预定义的测试用例，然后比较你的返回值和正确答案，那么就有很重要的一点：

你的代码**不要用文件级别的全局变量**，否则不同的测试用例之间可能会相互影响。力扣官网对于这个问题也有说明：  
<https://support.leetcode.cn/hc/kb/article/1194344/>

有些读者说他提交的解法通过不了，但是把出错的测试用例单独拎出来运行就是对的，一提交就错了，都是因为这个原因。你单独拎出来的时候只运行了一个测试用例，但提交后多个测试用例之间的数据产生了影响，就可能出错。

我在本站的教程中，尤其是讲解递归算法的部分，会经常提到「全局变量」这个词，但我指的全局变量并不是指文件级别的全局变量，而是类级别的全局变量。

具体举例说明吧，就比如二叉树的前序遍历这道题，题目输入一个二叉树的根节点，让你返回二叉树的前序遍历结果，你可以这样写：

```
class Solution {  
    // 正确示例，类级别的全局变量  
    LinkedList<Integer> res = new LinkedList<>();  
  
    public List<Integer> preorderTraversal(TreeNode root) {  
        traverse(root);  
    }  
}
```

```
    return res;
}

void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 类内的其他函数都可以访问到 res
    res.add(root.val);
    traverse(root.left);
    traverse(root.right);
}
}
```

对于 Java/C++/Python 这种给了一个 **Solution** 类的情况，你就把需要共享访问的变量定义在类里面，对于 Go/JavaScript 这种没有类的情况，你可以定义高阶函数，运用闭包的特性，让内部函数也可以访问到共享变量。

或者，你可以把这个变量作为函数的参数传递，这样也是一种解决方案：

```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        // 正确示例，作为参数传递给其他函数
        LinkedList<Integer> res = new LinkedList<>();
        traverse(root, res);
        return res;
    }

    void traverse(TreeNode root, LinkedList<Integer> res) {
        if (root == null) {
            return;
        }
        // 访问修改 res 变量
        res.add(root.val);
        traverse(root.left, res);
        traverse(root.right, res);
    }
}
```

对于这种方案，你要注意编程语言的特性，函数参数到底应该传值还是传引用/指针，否则会出现效率问题，甚至出错。

## 提交时清除标准输出

在运行测试用例时，力扣会把你的算法的标准输出返回给你，也就是说你可以通过 `print` 打印一些数据来调试代码。这里注意，准备提交代码前一定要把所有的打印语句注释掉。

因为标准输出是 IO 操作，非常消耗时间。如果包含了打印语句，也许你的算法代码本身就是最优解，但提交后发现执行效率非常差，甚至因为超时而无法通过。

好了，其他没啥了，开始学习吧！

# 时间空间复杂度入门

考虑到这是第一章，我不会对时空复杂度做面面俱到的讲解，详细的 [算法时空复杂度分析实用指南](#) 安排在你学完几种常见算法的核心框架之后，那时候你的知识储备可以轻松理解时空复杂度分析的各种场景。

因为本章后面的内容会带你实现常见的排序算法和数据结构，我会分析它们的时间复杂度，所以这里还是要提前介绍一下时间/空间复杂度的概念，以及分析时间/空间复杂度的简化方法，避免初学者疑惑。

对于初学者，你只需要记住以下几点：

1、时空复杂度用 Big O 表示法表示（类似  $O(1)$ ,  $O(n^2)$ ,  $O(\log n)$  等）。它们都是估计值，不需要精确计算，且仅保留最高增长项。

比方说  $O(2n^2 + 3n + 1)$  等同于  $O(n^2)$ ,  $O(1000n + 1000)$  等同于  $O(n)$ 。

2、时间复杂度用来衡量一个算法的执行效率，空间复杂度用来衡量算法的内存消耗，它们都是越小越好。

比方说时间复杂度  $O(n)$  的算法比  $O(n^2)$  的算法执行效率高，空间复杂度  $O(1)$  的算法比  $O(n)$  的算法内存消耗小。

当然，一般我们要说明这个  $n$  代表什么，比如  $n$  代表输入的数组的长度。

3、如何估算？现在你可以简单理解：时间复杂度就看 `for` 循环的嵌套层数；空间复杂度就看算法申请了多少空间来存储数据。

我这里说按照 `for` 循环的嵌套层数来估算时间复杂度仅是简化的方法，其实是不准确的。正确的方法会在 [算法时空复杂度分析实用指南](#) 介绍，但是对于初学者学习本章内容来说，这种估算方法足够用了。

举几个例子来说比较直观。

**示例一，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ ：**

```
// 输入一个整数数组，返回所有元素的和
int getSum(int[] nums) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) {
        sum += nums[i];
    }
    return sum;
}
```

算法包含一个 `for` 循环遍历 `nums` 数组，所以时间复杂度是  $O(n)$ ，其中  $n$  代表 `nums` 数组的长度。

我们的算法只使用了一个 `sum` 变量，这个 `nums` 是题目给的输入，不算在我们算法的空间复杂度里面，所以空间复杂度是  $O(1)$ 。

**示例二，时间复杂度  $O(n^2)$ ，空间复杂度  $O(1)$ ：**

```
// 数组是否存在两个数，它们的和为 target?
boolean hasTargetSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
```

```
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) {
                return true;
            }
        }
    return false;
}
```

算法包含两个 for 循环嵌套，所以时间复杂度是  $O(n^2)$ ，其中  $n$  代表  $\text{nums}$  数组的长度。

我们的算法只使用了  $i$ ,  $j$  两个变量，这是常数级别的空间消耗，所以空间复杂度是  $O(1)$ 。

你也许会说，内层的 for 循环并没有遍历整个数组，且有可能提前 return，算法实际执行的次数应该是小于  $n^2$  的，时间复杂度还是  $O(n^2)$  吗？

是的，还是  $O(n^2)$ 。前面说了 Big O 表示法是估计值，不需要精确计算。具体到不同的输入，算法的实际执行次数确实会小于  $n^2$ ，但我们不需要关心。

我们只需要知道，看到嵌套 for 循环，时间复杂度就是  $O(n^2)$ 。

**示例三，时间复杂度  $O(n)$ ，空间复杂度  $O(n)$ ：**

```
void exampleFn(int n) {
    int[] nums = new int[n];
}
```

这个函数中创建了一个大小为  $n$  的数组，所以空间复杂度是  $O(n)$ ，但是要知道申请数组空间及初始化数组也需要时间，所以时间复杂度也是  $O(n)$ 。

**示例四，时间复杂度  $O(n)$ ，空间复杂度  $O(n)$ ：**

```
// 输入一个整数数组，返回一个新的数组，新数组的每个元素是原数组对应元素的平方
int[] squareArray(int[] nums) {
    int[] res = new int[nums.length];
    for (int i = 0; i < nums.length; i++) {
        res[i] = nums[i] * nums[i];
    }
    return res;
}
```

算法初始化  $\text{res}$  数组需要  $O(n)$  的时间复杂度，包含一个 for 循环，时间复杂度也是  $O(n)$ ，总的时间复杂度是还是  $O(n)$  其中  $n$  代表  $\text{nums}$  数组的长度。

我们声明了一个新的数组  $\text{res}$ ，这个数组的长度和  $\text{nums}$  数组一样，所以空间复杂度是  $O(n)$ 。

好了，初学者明白上面这些基本的时间、空间复杂度分析暂时就够用了，继续往下学习吧。

# 使用可视化面板的 JS 基础

每道题的可视化代码我都提前写好了，甚至我会在文章或者注释中告诉你应该如何操作可视化面板来观察算法的执行过程。所以就算你不了解 JavaScript，也能从每道题目配套的可视化面板中受益。

但是，有些读者想修改我的预设代码，或者对自己的一些奇思妙想进行可视化验证，那么就需要了解一些 JavaScript 的基本语法，本文就是为这些读者准备的。

因为我的 [算法可视化面板](#) 目前只支持 JavaScript 语言，所以我就写一个极简 JS 教程，带大家了解 JS 的基本用法，目的是让不了解 JS 的读者也能把算法可视化面板用起来。

现在 AI 工具这么发达，一般也不需要你从头写 JS，可以让 AI 工具帮你把其他语言的代码翻译成 JS，你只需要了解 JS 基础语法，能看懂代码就够了。所以只要你有任意其他熟悉的编程语言，花 5 分钟看看下面的内容，就能把可视化面板用起来了。

## JavaScript 基本语法

### 变量声明

首先说一下 JavaScript 的变量声明，它有三种声明变量的方式，分别是 `var`, `let`, `const`。

`const` 声明的是常量，一旦声明就不能再修改，这个没什么好说的，一般在工程代码中才有使用场景，在算法代码中用不用都无所谓。

```
const a = 1;
// 报错
a = 2;
```

`var` 和 `let` 都可以声明一般的变量，但它俩声明出来的变量可见性不一样，`var` 应该算是历史遗留问题，总之你可以认为在可视化面板中它俩的效果是一样的。

```
let str = "hello world";
str = "world"
// 输出 world
console.log(str);

if (true) {
    // 这里的 str 和外面的 str 是两个不同的变量
    let str = "hello";
    // 输出 hello
    console.log(str);
}
```

我的可视化代码中的顶层函数一般是用 `var` 声明的，这是因为 LeetCode 上给的 JS 函数签名都是用 `var` 声明的，所以我也沿用 `var` 了，不过你非要改成 `let` 也是没问题的。

### 函数声明

JavaScript 的函数声明也很简单，就是这样：

```
function add(a, b) {
    return a + b;
}
// 输出 3
console.log(add(1, 2));
```

这个函数的声明和其他编程语言的声明是一样的，不过 JavaScript 中可能还会见到下面的匿名函数声明方式，相当于是用变量接收了一个匿名函数，不过使用起来都是一样的：

```
// 把 let 换成 var 也一样的效果
let add2 = function(a, b) {
    return a + b;
}
// 输出 3
console.log(add2(1, 2));

// 用 ES6 的箭头函数声明
let add3 = (a, b) => {
    return a + b;
}
// 输出 3
console.log(add3(1, 2));
```

用 `function` 关键词声明的函数和用 `() => {}` 这种方式声明的箭头函数有一些关键差别，主要是函数体内使用 `this` 指针的行为不同，但是在算法题中基本不会用到这个特性，所以你可以认为这两种方式是一样的。

## 循环控制

JavaScript 的循环控制和其他编程语言也是一样的，常见的就是 `for` 和 `while`。

以遍历数组为例，先讲最常见的方式，用索引遍历：

```
// 遍历数组
let arr = [1, 2, 3, 4, 5];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
// 输出 1 2 3 4 5
```

然后是用 `for...of` 方式，可以直接遍历元素：

```
// 遍历数组
let arr = [1, 2, 3, 4, 5];
for (let item of arr) {
    console.log(item);
}
// 输出 1 2 3 4 5
```

注意 JavaScript 用的是 `of` 来遍历数组元素，不要用 `in`。`in` 有其他的作用，不过我们做算法题基本用不到，我就不啰嗦了。

我知道有些其他语言会用 `in` 遍历数组元素，所以特此提醒，不要搞错了。

`while` 循环和其他语言一样，举个简单例子：

```
let arr = [1, 2, 3, 4, 5];
let i = 0;
while (i < arr.length) {
    console.log(arr[i]);
    i++;
}
// 输出 1 2 3 4 5
```

## 条件判断

`if else` 的用法和其他语言完全一致，没啥可说的，举个简单的例子：

```
let a = 1;
if (a === 1) {
    console.log("a 等于 1");
} else {
    console.log("a 不等于 1");
}
```

这里也有一个小坑，就是 JavaScript 中的 `==` 和 `===` 的区别。`==` 是值相等就行，`===` 是值和类型都相等才行。

简单讲，你就记住其他语言中的 `==`, `!=` 在 JavaScript 中是 `==`, `!=`，不要在 JavaScript 中用 `==`, `!=` 就行了。

## JavaScript 基本数据结构

### 字符串

JavaScript 中的字符串和其他语言一样，没啥特别的，举个例子：

```
let str = "hello world";
// 输出 11
console.log(str.length);
// 输出 h
console.log(str[0]);
// 输出 true
console.log(str === "hello world");

// 字符串分割
let arr = str.split(" ");
// 输出 ["hello", "world"]
console.log(arr);
```

```
// 获取子串，多种方式都可以
// 输出 hello
console.log(str.substring(0, 5));
// 输出 hello
console.log(str.slice(0, 5));
// 输出 hello
console.log(str.substr(0, 5));

// 字符串拼接
let str2 = "world";
// 输出 hello world world
console.log(str + " " + str2);
```

## 数组

有以下几种方法创建数组：

```
let arr1 = [1, 2, 3, 4, 5];
let arr2 = new Array(1, 2, 3, 4, 5);

// 创建一个长度为 5 的数组，每个元素都是 undefined
let arr3 = new Array(5);

// 创建一个长度为 5 的数组，每个元素都是 0
let arr4 = new Array(5).fill(0);
```

数组的常见操作：

```
let arr = [1, 2, 3, 4, 5];

// 获取数组长度
// 输出 5
console.log(arr.length);

// 获取数组元素
// 输出 1
console.log(arr[0]);

// 修改数组元素
arr[0] = 100;

// 复制数组的所有元素到一个新数组
let arr2 = arr.slice();
// 这也是一种复制数组的方法
let arr3 = [...arr];

// 在数组末尾添加一个元素
arr.push(6);

// 删除数组尾部的元素
arr.pop();

// 在数组开头添加一个元素 888
```

```
// 不常用，因为算法中都尽量避免在数组的非末尾位置增删元素  
arr.unshift(888);  
  
// 删除数组开头的元素  
// 不常用，因为算法中都尽量避免在数组的非末尾位置增删元素  
arr.shift();
```

好了，了解上述基本操作完全足够你在算法题中使用数组了。

## 哈希表

通俗来讲，JavaScript 中的对象就可以理解为是哈希表，因为 JavaScript 对象就是若干键值对。不过 ES6 中引入了 **Map** 类型，所以我们就规范一些，用 **Map** 类型来创建哈希表。

**Map** 的基本操作：

```
let map = new Map();  
  
// 添加键值对  
map.set("a", 1);  
map.set("b", 2);  
map.set("c", 3);  
  
// 获取键值对  
// 输出 1  
console.log(map.get("a"));  
  
// 删除键值对  
map.delete("a");  
  
// 判断是否存在某个键  
// 输出 false  
console.log(map.has("a"));  
// 输出 true  
console.log(map.has("b"));  
  
// 遍历键值对  
for (let key of map.keys()) {  
    console.log(key, map.get(key));  
}  
// 输出 b 2 和 c 3
```

了解这些就差不多了，遇到不会处理的场景再查查文档就行了。

## 哈希集合

ES6 中引入的 **Set** 类型就是哈希集合，它用来存储不重复的元素，基本操作如下：

```
let set = new Set();  
  
// 添加元素  
set.add(1);  
set.add(2);  
set.add(3);
```

```
// 删除元素
set.delete(1);

// 判断是否存在某个元素
// 输出 false
console.log(set.has(1));
// 输出 true
console.log(set.has(2));

// 遍历元素
for (let item of set) {
    console.log(item);
}
// 输出 2 和 3
```

## 其他特殊数据结构

在算法题中，还会用到一些特殊的数据结构，比如链表、树、二叉堆等，这些数据结构不是 JavaScript 内置的，不过我的算法可视化面板中已经实现了这些特殊数据结构，具体用法可以参考我的 [算法可视化面板简介](#)。

# 数组（顺序存储）基本原理

我们在说「数组」的时候有多种不同的语境，因为不同的编程语言提供的数组类型和 API 是不一样的，所以开头先统一一下说辞，方便后面的讲解。

我认为暂且可以把「数组」分为两大类，一类是「静态数组」，一类是「动态数组」。

「静态数组」就是一块连续的内存空间，我们可以通过索引来访问这块内存空间中的元素，这才是数组的原始形态。

而「动态数组」是编程语言为了方便我们使用，在静态数组的基础上帮我们添加了一些常用的 API，比如 `push`, `insert`, `remove` 等等方法，这些 API 可以让我们更方便地操作数组元素，不用自己去写代码实现这些操作。

本章的内容就是带大家仅仅使用最原始的静态数组，自己实现一个动态数组，实现增删查改的常见 API。以后你在使用标准库提供的数据结构时，就知道它们的底层运行原理了。

有了动态数组，后面讲到的队列、栈、哈希表等复杂数据结构都会依赖它进行实现。

## 静态数组

静态数组在创建的时候就要确定数组的元素类型和元素数量。只有在 C++、Java、Golang 这类语言中才提供了创建静态数组的方式，类似 Python、JavaScript 这类语言并没有提供静态数组的定义方式。

静态数组的用法比较原始，实际软件开发中很少用到，写算法题也没必要用，我们一般直接用动态数组。但为了理解原理，在这里还是要讲解一下。

定义一个静态数组的方法如下：

```
// 定义一个大小为 10 的静态数组
int[] arr = new int[10];

// 使用索引赋值
arr[0] = 1;
arr[1] = 2;

// 使用索引取值
int a = arr[0];
```

就这，没有其他什么操作了。

拿 C++ 来举例吧，`int arr[10]` 这段代码到底做了什么事情呢？主要有这么几件事：

1、在内存中开辟了一段连续的内存空间，大小是 `10 * sizeof(int)` 字节。一个 int 在计算机内存中占 4 字节，也就是总共 40 字节。

2、定义了一个名为 `arr` 的数组指针，指向这段内存空间的首地址。

那么 `arr[1] = 2` 这段代码又做了什么事情呢？主要有这么几件事：

1、计算 `arr` 的首地址加上 `1 * sizeof(int)` 字节（4 字节）的偏移量，找到了内存空间中的第二个元素的首地址。

2、从这个地址开始的 4 个字节的内存空间中写入了整数 `2`。

我记得以前刚上大学的时候要学 C 语言基础，有些同学就绕不清楚什么指针的数组，数组的指针，绕来绕去的。其实只要明白了上面这个简单的流程，一切就很清楚了。

1、为什么数组的索引从 0 开始？就是方便取地址。`arr[0]` 就是 `arr` 的首地址，从这个地址往后的 4 个字节存储着第一个元素的值；`arr[1]` 就是 `arr` 的首地址加上 `1 * 4` 字节，也就是第二个元素的首地址，这个地址往后的 4 个字节存储着第二个元素的值。`arr[2], arr[3]` 以此类推。

2、因为数组的名字 `arr` 就指向整块内存的首地址，所以数组名 `arr` 就是一个指针。你直接取这个地址的值，就是第一个元素的值。也就是说，`*arr` 的值就是 `arr[0]`，即第一个元素的值。

3、如果不使用 `memset` 这种函数初始化数组的值，那么数组内的值是不确定的。因为 `int arr[10]` 这个语句只是请操作系统在内存中开辟了一块连续的内存空间，你也不知道这块空间是谁使用过的二手内存，你也不知道里面存了什么奇奇怪怪的东西。所以一般我们会用 `memset` 函数把这块内存空间的值初始化一下再使用。

当然，上面讲的这些内容都是针对 C/C++，因为大家学习计算机基础的时候都接触过。其他比如 Java Golang 这种语言，静态数组创建出来后会自动帮你把元素值都初始化为 0，所以不需要再显式进行初始化。

我梳理一下上面的因果逻辑，静态数组本质上就是一块连续的内存空间，`int arr[10]` 这个语句我们可以得知：

1、我们知道这块内存空间的首地址（数组名 `arr` 就指向这块内存空间的首地址）。

2、我们知道了每个元素的类型（比如 `int`），也就是知道了每个元素占用的内存空间大小（比如一个 `int` 占 4 字节，32 bit）。

3、这块内存空间是连续的，其大小为 `10 * sizeof(int)` 即 40 字节。

所以，我们获得了数组的超能力「随机访问」：只要给定任何一个数组索引，我可以在  $O(1)$  的时间内直接获取到对应元素的值。

因为我可以通过首地址和索引直接计算出目标元素的内存地址。计算机的内存寻址时间可以认为是  $O(1)$ ，所以数组的随机访问时间复杂度是  $O(1)$ 。

但是，一个人最大的优势往往也是他的最大劣势。数组连续内存的特性给了他随机访问的超能力，但它也因此吃了不少苦，下面介绍。

## 增删查改

数据结构的职责就是增删查改，再无其他。

那么刚刚介绍数组这种数据结构的底层原理，我们其实只介绍了「查」和「改」的部分，也就是通过索引修改和访问对应元素的值。那么「增删」这两个操作又是如何实现的呢？

### 增

要想给静态数组增加元素，这就有些复杂了，需要分情况讨论。

比方说，我有一个大小为 10 的数组，里面装了 4 个元素，现在想在末尾追加一个元素，怎么办？

比较简单，直接在对应的索引赋值就行了，这是大概的代码逻辑：

```
// 大小为 10 的数组已经装了 4 个元素
int[] arr = new int[10];
```

```
for (int i = 0; i < 4; i++) {
    arr[i] = i;
}

// 现在想在数组末尾追加一个元素 4
arr[4] = 4;

// 再在数组末尾追加一个元素 5
arr[5] = 5;

// 依此类推
// ...
```

可以看到，由于只是对索引赋值，所以在数组末尾追加元素的时间复杂度是  $O(1)$ 。

比方说，我有一个大小为 10 的数组 `arr`，前 4 个索引装了元素，现在想在第 3 个位置 (`arr[2]`) 插入一个新元素，怎么办？

这就要涉及「数据搬移」，给新元素腾出空位，然后再才能插入新元素。大概的代码逻辑是这样的：

```
// 大小为 10 的数组已经装了 4 个元素
int[] arr = new int[10];
for (int i = 0; i < 4; i++) {
    arr[i] = i;
}

// 在第 3 个位置插入元素 666
// 需要把第 3 个位置及之后的元素都往后移动一位
// 注意要倒着遍历数组中已有元素避免覆盖，不懂的话请看下方可视化面板
for (int i = 4; i > 2; i--) {
    arr[i] = arr[i - 1];
}

// 现在第 3 个位置空出来了，可以插入新元素
arr[2] = 666;
```

### ▶ 😊 代码可视化动画 😊

综上，在数组中间插入元素的时间复杂度是  $O(N)$ ，因为涉及到数据搬移，给新元素腾地方。

静态数组在创建时就要确定大小，比方说现在我创建了一个数组 `int arr[10]`（一块 40 字节的连续内存空间），然后在里面存了 10 个元素，这时候我想再插入一个元素，怎么办？无论是追加在尾部还是插入到中间，都没有位置留给新元素了。

有的读者可能说，这个简单呀，在这 40 字节后面再加上 4 个字节的连续内存空间，用来存储新的元素，不就行了吗？

不行的，连续内存必须一次性分配，分配完了之后就不能随意增减了。因为你这块连续内存后面的内存空间可能已经被其他程序占用了，不能说你想要就给你。

那怎么办呢？只能重新申请一块更大的内存空间，把原来的数据复制过去，再插入新的元素，这就是数组的「扩容」操作。

比方说，我重新创建一个更大的数组 `int arr[20]`，然后把原来的 10 个元素复制过去，这样就有空余位置插入新的元素了。

大概的逻辑是这样的：

```
// 大小为 10 的数组已经装满了
int[] arr = new int[10];
for (int i = 0; i < 10; i++) {
    arr[i] = i;
}

// 现在想在数组末尾追加一个元素 10
// 需要先扩容数组
int[] newArr = new int[20];
// 把原来的 10 个元素复制过去
for (int i = 0; i < 10; i++) {
    newArr[i] = arr[i];
}

// 旧数组的内存空间将由垃圾收集器处理
// ...

// 在新的大数组中追加新元素
newArr[10] = 10;
```

综上，数组的扩容操作会涉及到新数组的开辟和数据的复制，时间复杂度是  $O(N)$ 。

## 删

删除元素的操作和增加元素的操作类似，也需要分情况讨论。

比方说，我有一个大小为 10 的数组，里面装了 5 个元素，现在想删除末尾的元素，怎么办？

很简单，直接把末尾元素标记为一个特殊值代表已删除就行了，我们这里简单举例，就用 -1 作为特殊值代表已删除好了。后面带大家具体实现动态数组的时候，会有更完善的方法删除数组元素，这里只是为了说明删除数组尾部元素的本质就是进行一次随机访问，时间复杂度是  $O(1)$ 。

大概的代码逻辑是这样的：

```
// 大小为 10 的数组已经装了 5 个元素
int[] arr = new int[10];
for (int i = 0; i < 5; i++) {
    arr[i] = i;
}

// 删除末尾元素，暂时用 -1 代表元素已删除
arr[4] = -1;
```

比方说，我有一个大小为 10 的数组，里面装了 5 个元素，现在想删除第 2 个元素 (`arr[1]`)，怎么办？

这也要涉及「数据搬移」，把被删元素后面的元素都往前移动一位，保持数组元素的连续性。

大概的代码逻辑是这样的：

```
// 大小为 10 的数组已经装了 5 个元素
int[] arr = new int[10];
for (int i = 0; i < 5; i++) {
    arr[i] = i;
}

// 删除 arr[1]
// 需要把 arr[1] 之后的元素都往前移动一位
// 注意要正着遍历数组中已有元素避免覆盖，不懂的话请看下方可视化面板
for (int i = 1; i < 4; i++) {
    arr[i] = arr[i + 1];
}

// 最后一个元素置为 -1 代表已删除
arr[4] = -1;
```

### ▶ 代码可视化动画

综上，在数组中间删除元素的时间复杂度是  $O(N)$ ，因为涉及到数据搬移。

## 总结

综上，静态数组的增删查改操作的时间复杂度是：

- 增：
  - 在末尾追加元素： $O(1)$ 。
  - 在中间（非末尾）插入元素： $O(N)$ 。
- 删：
  - 删除末尾元素： $O(1)$ 。
  - 删除中间（非末尾）元素： $O(N)$ 。
- 查：给定指定索引，查询索引对应的元素的值，时间复杂度  $O(1)$ 。
- 改：给定指定索引，修改索引对应的元素的值，时间复杂度  $O(1)$ 。

有读者可能问，刚才不是还探讨过数组的扩容操作吗，扩容涉及到新数组空间的开辟和数据的复制，时间复杂度是  $O(N)$ ，这个复杂度为什么没有算到「增」的复杂度里面呢？

这个问题很好，但并不是每次增加元素的时候都会触发扩容，所以扩容的复杂度要用「均摊时间复杂度」来分析，这个概念我在 [时空复杂度分析方法](#) 中有详细的讲解，这里就不展开了。

还有个问题初学者要注意，我们说数组的查、改复杂度是  $O(1)$ ，这个仅仅适用于给定索引的情况。如果反过来，比方说给你一个值，让你去找这个值在数组中对应的索引，那你只能遍历整个数组去寻找对吧，这个复杂度就是  $O(N)$  了。

所以说要搞清楚原理，而不要去背概念。原理懂了，概念你自己都能推导出来的。

## 动态数组

刚才讲了静态数组的超能力和种种局限性，现在讲动态数组，动态数组是静态数组的强化版，也是我们在实际软件开发或者写算法题时最常用的数据结构之一。

首先，你不要以为动态数组可以解决静态数组在中间增删元素效率差的问题，不可能解决的。数组随机访问的超能力源于数组连续的内存空间，而连续的内存空间就不可避免地面对数据搬移和扩缩容的问题。

动态数组底层还是静态数组，只是自动帮我们进行数组空间的扩缩容，并把增删查改操作进行了封装，让我们使用起来更方便而已。

简单列举一下各个语言的动态数组使用方法：

```
// 创建动态数组
// 不用显式指定数组大小，它会根据实际存储的元素数量自动扩缩容
ArrayList<Integer> arr = new ArrayList<>();

for (int i = 0; i < 10; i++) {
    // 在末尾追加元素，时间复杂度 O(1)
    arr.add(i);
}

// 在中间插入元素，时间复杂度 O(N)
// 在索引 2 的位置插入元素 666
arr.add(2, 666);

// 在头部插入元素，时间复杂度 O(N)
arr.add(0, -1);

// 删除末尾元素，时间复杂度 O(1)
arr.remove(arr.size() - 1);

// 删除中间元素，时间复杂度 O(N)
// 删除索引 2 的元素
arr.remove(2);

// 根据索引查询元素，时间复杂度 O(1)
int a = arr.get(0);

// 根据索引修改元素，时间复杂度 O(1)
arr.set(0, 100);

// 根据元素值查找索引，时间复杂度 O(N)
int index = arr.indexOf(666);
```

在后面的章节，我会手把手带大家实现一个动态数组，让大家更加深入地理解动态数组的原理。

# 动态数组代码实现

阅读本文前，你需要先学习：

- 数组（顺序存储）基础

## 几个关键点

下面我会直接给出一个简单的动态数组代码实现，包含了基本的增删查改功能。这里先给出几个关键点，等会你看代码的时候可以着重注意一下。

### 关键点一、自动扩缩容

在上一章 [数组基础](#) 中只提到了数组添加元素时可能需要扩容，并没有提到缩容。

在实际使用动态数组时，缩容也是重要的优化手段。比方说一个动态数组开辟了能够存储 1000 个元素的连续内存空间，但是实际只存了 10 个元素，那就有 990 个空间是空闲的。为了避免资源浪费，我们其实可以适当缩小存储空间，这就是缩容。

我们这里就实现一个简单的扩缩容的策略：

- 当数组元素个数到底层静态数组的容量上限时，扩容为原来的 2 倍；
- 当数组元素个数缩减到底层静态数组的容量的 1/4 时，缩容为原来的 1/2。

### 关键点二、索引越界的检查

下面的代码实现中，有两个检查越界的方法，分别是 `checkElementIndex` 和 `checkPositionIndex`，你可以看到它们的区别仅仅在于 `index < size` 和 `index <= size`。

为什么 `checkPositionIndex` 可以允许 `index == size` 呢，因为这个 `checkPositionIndex` 是专门用来处理在数组中插入元素的情况。

比方说有这样一个 `nums` 数组，对于每个元素来说，合法的索引一定是 `index < size`：

```
nums = [5, 6, 7, 8]
index  0  1  2  3
```

但如果是要在数组中插入新元素，那么新元素可能的插入位置并不是元素的索引，而是索引之间的空隙：

```
nums = [ | 5 | 6 | 7 | 8 | ]
index   0   1   2   3   4
```

这些空隙都是合法的插入位置，所以说 `index == size` 也是合法的。这就是 `checkPositionIndex` 和 `checkElementIndex` 的区别。

### 关键点三、删除元素谨防内存泄漏

单从算法的角度，其实并不需要关心被删掉的元素应该如何处理，但是具体到代码实现，我们需要注意可能出现的内存泄漏。

在我给出的代码实现中，删除元素时，我都会把被删除的元素置为 `null`，以 Java 为例：

```
// 删
public E removeLast() {
    E deletedVal = data[size - 1];
    // 删除最后一个元素
    // 必须给最后一个元素置为 null，否则会内存泄漏
    data[size - 1] = null;
    size--;
    return deletedVal;
}
```

Java 的垃圾回收机制是基于 [图算法](#) 的可达性分析，如果一个对象再也无法被访问到，那么这个对象占用的内存才会被释放；否则，垃圾回收器会认为这个对象还在使用中，就不会释放这个对象占用的内存。

如果你不执行 `data[size - 1] = null` 这行代码，那么 `data[size - 1]` 这个引用就会一直存在，你可以通过 `data[size - 1]` 访问这个对象，所以这个对象被认为是可达的，它的内存就一直不会被释放，进而造成内存泄漏。

其他带垃圾回收功能的语言应该也是类似的，你可以具体了解一下你使用的编程语言的垃圾回收机制，这是写出无 bug 代码的基本要求。

## 其他细节优化

下面的代码当然不会是一个很完善的实现，会有不少可以进一步优化的点。比方说，我是用 `for` 循环复制数组数据的，实际上这种方式复制的效率比较差，大部分编程语言会提供更高效的数组复制方法，比如 Java 的 `System.arraycopy`。

不过它再怎么优化，本质上也是要搬移数据，时间复杂度都是  $O(n)$ 。本文的重点在于让你理解数组增删查改 API 的基本实现思路以及时间复杂度，如果对这些细节感兴趣，可以找到编程语言标准库的源码深入研究。

你可以借助力扣第 707 题「设计链表」来验证自己的实现是否正确。虽然这道题是关于链表的，但是它其实也不知道你底层到底是不是用链表实现的。咱主要是借用它的测试用例，来验证你的增删查改功能是否正确。

## 动态数组代码实现

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyArrayList<E> {
    // 真正存储数据的底层数组
    private E[] data;
    // 记录当前元素个数
    private int size;
    // 默认初始容量
    private static final int INIT_CAP = 1;

    public MyArrayList() {
        this(INIT_CAP);
    }

    public MyArrayList(int initCapacity) {
        data = (E[]) new Object[initCapacity];
```

```
        size = 0;
    }

    // 增
    public void addLast(E e) {
        int cap = data.length;
        // 看 data 数组容量够不够
        if (size == cap) {
            resize(2 * cap);
        }
        // 在尾部插入元素
        data[size] = e;
        size++;
    }

    public void add(int index, E e) {
        // 检查索引越界
        checkPositionIndex(index);

        int cap = data.length;
        // 看 data 数组容量够不够
        if (size == cap) {
            resize(2 * cap);
        }

        // 搬移数据 data[index..] -> data[index+1..]
        // 给新元素腾出位置
        for (int i = size - 1; i >= index; i--) {
            data[i + 1] = data[i];
        }

        // 插入新元素
        data[index] = e;

        size++;
    }

    public void addFirst(E e) {
        add(0, e);
    }

    // 删
    public E removeLast() {
        if (size == 0) {
            throw new NoSuchElementException();
        }
        int cap = data.length;
        // 可以缩容，节约空间
        if (size == cap / 4) {
            resize(cap / 2);
        }

        E deletedVal = data[size - 1];
        // 删除最后一个元素
        // 必须给最后一个元素置为 null，否则会内存泄漏
        data[size - 1] = null;
        size--;
    }

    return deletedVal;
}
```

```
}

public E remove(int index) {
    // 检查索引越界
    checkElementIndex(index);

    int cap = data.length;
    // 可以缩容，节约空间
    if (size == cap / 4) {
        resize(cap / 2);
    }

    E deletedVal = data[index];

    // 搬移数据 data[index+1..] -> data[index..]
    for (int i = index + 1; i < size; i++) {
        data[i - 1] = data[i];
    }

    data[size - 1] = null;
    size--;
}

return deletedVal;
}

public E removeFirst() {
    return remove(0);
}

// 查
public E get(int index) {
    // 检查索引越界
    checkElementIndex(index);

    return data[index];
}

// 改
public E set(int index, E element) {
    // 检查索引越界
    checkElementIndex(index);
    // 修改数据
    E oldVal = data[index];
    data[index] = element;
    return oldVal;
}

// 工具方法
public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}

// 将 data 的容量改为 newCap
private void resize(int newCap) {
    E[] temp = (E[]) new Object[newCap];
```

```
        for (int i = 0; i < size; i++) {
            temp[i] = data[i];
        }

        data = temp;
    }

    private boolean isElementIndex(int index) {
        return index >= 0 && index < size;
    }

    private boolean isPositionIndex(int index) {
        return index >= 0 && index <= size;
    }

    // 检查 index 索引位置是否可以存在元素
    private void checkElementIndex(int index) {
        if (!isElementIndex(index))
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
    }

    // 检查 index 索引位置是否可以添加元素
    private void checkPositionIndex(int index) {
        if (!isPositionIndex(index))
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
    }

    private void display() {
        System.out.println("size = " + size + " cap = " + data.length);
        System.out.println(Arrays.toString(data));
    }

    public static void main(String[] args) {
        // 初始容量设置为 3
        MyArrayList<Integer> arr = new MyArrayList<>(3);

        // 添加 5 个元素
        for (int i = 1; i <= 5; i++) {
            arr.addLast(i);
        }

        arr.remove(3);
        arr.add(1, 9);
        arr.addFirst(100);
        int val = arr.removeLast();

        for (int i = 0; i < arr.size(); i++) {
            System.out.println(arr.get(i));
        }
    }
}
```

# 链表（链式存储）基本原理

刷过力扣的读者肯定对单链表非常熟悉，力扣上的单链表节点定义如下：

```
class ListNode {  
    int val;  
    ListNode next;  
    ListNode(int x) { val = x; }  
}
```

这仅仅是一个最简单的**单链表节点**，方便力扣出算法题来考你。在实际的编程语言中，我们使用的链表节点会稍微复杂一点，类似这样：

```
class Node<E> {  
    E val;  
    Node<E> next;  
    Node<E> prev;  
  
    Node(Node<E> prev, E element, Node<E> next) {  
        this.val = element;  
        this.next = next;  
        this.prev = prev;  
    }  
}
```

主要区别有两个：

- 1、编程语言标准库一般都会提供泛型，即你可以指定 `val` 字段为任意类型，而力扣的单链表节点的 `val` 字段只有 `int` 类型。
- 2、编程语言标准库一般使用的都是双链表而非单链表。单链表节点只有一个 `next` 指针，指向下一个节点；而双链表节点有两个指针，`prev` 指向前一个节点，`next` 指向下一个节点。

有了 `prev` 前驱指针，链表支持双向遍历，但由于要多维护一个指针，增删查改时会稍微复杂一些，后面带大家实现双链表时会具体介绍。

## 为什么需要链表

前面介绍了[数组（顺序存储）的底层原理](#)，说白了就是一块连续的内存空间，有了这块内存空间的首地址，就能直接通过索引计算出任意位置的元素地址。

链表不一样，一条链表并不需要一整块连续的内存空间存储元素。链表的元素可以分散在内存空间的天涯海角，通过每个节点上的 `next`, `prev` 指针，将零散的内存块串联起来形成一个链式结构。

这样做的好处很明显，首先就是可以提高内存的利用效率，链表的节点不需要挨在一起，给点内存 `new` 出来一个节点就能用，操作系统会觉得这娃好养活。

另外一个好处，它的节点要用的时候就能接上，不用的时候拆掉就行了，从来不需要考虑扩缩容和数据搬移的问题，理论上讲，链表是没有容量限制的（除非把所有内存都占满，这不太可能）。

当然，不可能只有好处没有局限性。数组最大的优势是支持通过索引快速访问元素，而链表就不支持。

这个不难理解吧，因为元素并不是紧挨着的，所以如果你想要访问第 3 个链表元素，你就只能从头结点开始往顺着 `next` 指针往后找，直到找到第 3 个节点才行。

上面是对链表这种数据结构的基本介绍，接下来我们就结合代码实现单/双链表的几个基本操作。

## 单链表的基本操作

我先写一个工具函数，用于创建一条单链表，方便后面的讲解：

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

// 输入一个数组，转换为一条单链表
ListNode createLinkedList(int[] arr) {
    if (arr == null || arr.length == 0) {
        return null;
    }
    ListNode head = new ListNode(arr[0]);
    ListNode cur = head;
    for (int i = 1; i < arr.length; i++) {
        cur.next = new ListNode(arr[i]);
        cur = cur.next;
    }
    return head;
}
```

## 查/改

比方说，我想访问单链表的每一个节点，并打印其值，可以这样写：

```
// 创建一条单链表
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});

// 遍历单链表
for (ListNode p = head; p != null; p = p.next) {
    System.out.println(p.val);
}
```

类似的，如果是要通过索引访问或修改链表中的某个节点，也只能用 `for` 循环从头结点开始往后找，直到找到索引对应的节点，然后进行访问或修改。

## 增

直接看代码吧，很简单：

```
// 创建一条单链表
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});
```

```
// 在单链表头部插入一个新节点 0
ListNode newHead = new ListNode(0);
newHead.next = head;
head = newHead;
// 现在链表变成了 0 -> 1 -> 2 -> 3 -> 4 -> 5
```

## ▶ 🌟 代码可视化动画🌟

这个操作稍微复杂一点，因为我们要先从头结点开始遍历到链表的最后一个节点，然后才能在最后一个节点后面再插入新节点：

```
// 创建一条单链表
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});

// 在单链表尾部插入一个新节点 6
ListNode p = head;
// 先走到链表的最后一个节点
while (p.next != null) {
    p = p.next;
}
// 现在 p 就是链表的最后一个节点
// 在 p 后面插入新节点
p.next = new ListNode(6);

// 现在链表变成了 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

当然，如果我们持有对链表尾节点的引用，那么在尾部插入新节点的操作就会变得非常简单，不用每次从头去遍历了。这个优化会在后面具体实现双链表时介绍。

## ▶ 🎨 代码可视化动画🎨

这个操作稍微有点复杂，我们还是要先找到要插入位置的前驱节点，然后操作前驱节点把新节点插入进去：

```
// 创建一条单链表
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});

// 在第 3 个节点后面插入一个新节点 66
// 先要找到前驱节点，即第 3 个节点
ListNode p = head;
for (int i = 0; i < 2; i++) {
    p = p.next;
}
// 此时 p 指向第 3 个节点
// 组装新节点的后驱指针
ListNode newNode = new ListNode(66);
newNode.next = p.next;

// 插入新节点
```

```
p.next = newNode;  
// 现在链表变成了 1 -> 2 -> 3 -> 66 -> 4 -> 5
```

## ▶ 🎨 代码可视化动画

### 删

删除一个节点，首先要找到要被删除节点的前驱节点，然后把这个前驱节点的 `next` 指针指向被删除节点的下一个节点。这样就能把被删除节点从链表中摘除了。

```
// 创建一条单链表  
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});  
  
// 删除第 4 个节点，要操作前驱节点  
ListNode p = head;  
for (int i = 0; i < 2; i++) {  
    p = p.next;  
}  
  
// 此时 p 指向第 3 个节点，即要删除节点的前驱节点  
// 把第 4 个节点从链表中摘除  
p.next = p.next.next;  
  
// 现在链表变成了 1 -> 2 -> 3 -> 5
```

## ▶ 🌟 代码可视化动画

这个操作比较简单，找到倒数第二个节点，然后把它的 `next` 指针置为 `null` 就行了：

```
// 创建一条单链表  
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});  
  
// 删除尾节点  
ListNode p = head;  
// 找到倒数第二个节点  
while (p.next.next != null) {  
    p = p.next;  
}  
  
// 此时 p 指向倒数第二个节点  
// 把尾节点从链表中摘除  
p.next = null;  
  
// 现在链表变成了 1 -> 2 -> 3 -> 4
```

## ▶ 🎃 代码可视化动画

这个操作比较简单，直接把 `head` 移动到下一个节点就行了，直接看代码吧：

```
// 创建一条单链表
ListNode head = createLinkedList(new int[]{1, 2, 3, 4, 5});

// 删除头结点
head = head.next;

// 现在链表变成了 2 -> 3 -> 4 -> 5
```

不过可能有读者疑惑，之前那个旧的头结点 1 的 `next` 指针依然指向着节点 2，这样会不会造成内存泄漏？

不会的，这个节点 1 指向其他的节点是没关系的，只要保证没有其他引用指向这个节点 1，它就能被垃圾回收器回收掉。

当然，如果你非要显式把节点 1 的 `next` 指针置为 `null`，这是个很好的习惯，在其他场景中可能可以避免指针错乱的潜在问题。

在下面这个可视化面板中，我显式地把待删除节点的 `next` 指针置为 `null` 了：

#### ▶ 代码可视化动画

链表的增删查改操作确实比数组复杂。这是因为链表的节点不是紧挨着的，所以要增删一个节点，必须先找到它的前驱和后驱节点进行协同，然后才能通过指针操作把它插入或删除。

上面给出的代码还仅仅是最简单的例子，你会发现在头部、尾部、中间增删元素的代码都不一样。如果要实现一个真正可用的链表，你还要考虑到很多边界情况，比如链表可能为空、前后驱节点可能为空等，这些情况都得保证不出错。

而且，上面只是介绍了「单链表」，而我们下一章要实现的是「双链表」，双链表要同时维护前驱和后驱指针，指针操作会更复杂一些。

是不是已经不敢想了？不要怕，其实没你想的那么难，几个原因：

- 1、其实搞来搞去就那几个操作，等会儿带你动手实现链表 API 的时候，你亲自写一写，就会了。
- 2、复杂操作我都配了可视化面板，你可以结合面板中的代码和动画进行理解。
- 3、最重要的，我们会使用「虚拟头结点」技巧，把头、尾、中部的操作统一起来，同时还能避免处理头尾指针为空情况的边界情况。

虚拟节点技巧我在 [链表双指针技巧汇总](#) 中讲过，待会儿动手实现双链表的时候也会具体讲，这里就简单提一下。

## 双链表的基本操作

我先写一个工具函数，用于创建一条双链表，方便后面的讲解：

```
class DoublyListNode {
    int val;
    DoublyListNode next, prev;
    DoublyListNode(int x) { val = x; }
}
```

```
DoublyListNode createDoublyLinkedList(int[] arr) {
    if (arr == null || arr.length == 0) {
        return null;
    }
    DoublyListNode head = new DoublyListNode(arr[0]);
    DoublyListNode cur = head;
    // for 循环迭代创建双链表
    for (int i = 1; i < arr.length; i++) {
        DoublyListNode newNode = new DoublyListNode(arr[i]);
        cur.next = newNode;
        newNode.prev = cur;
        cur = cur.next;
    }
    return head;
}
```

## 查/改

对于双链表的遍历和查找，我们可以从头节点或尾节点开始，根据需要向前或向后遍历：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});
DoublyListNode tail = null;

// 从头节点向后遍历双链表
for (DoublyListNode p = head; p != null; p = p.next) {
    System.out.println(p.val);
    tail = p;
}

// 从尾节点向前遍历双链表
for (DoublyListNode p = tail; p != null; p = p.prev) {
    System.out.println(p.val);
}
```

访问或修改节点时，可以根据索引是靠近头部还是尾部，选择合适的方向遍历，这样可以一定程度上提高效率。

## 增

在双链表头部插入元素，需要调整新节点和原头节点的指针：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

// 在双链表头部插入新节点 0
DoublyListNode newHead = new DoublyListNode(0);
newHead.next = head;
head.prev = newHead;
head = newHead;
// 现在链表变成了 0 -> 1 -> 2 -> 3 -> 4 -> 5
```

▶ 🎨 代码可视化动画🎨

在双链表尾部插入元素时，如果我们持有尾节点的引用，这个操作会非常简单：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

DoublyListNode tail = head;
// 先走到链表的最后一个节点
while (tail.next != null) {
    tail = tail.next;
}

// 在双链表尾部插入新节点 6
DoublyListNode newNode = new DoublyListNode(6);
tail.next = newNode;
newNode.prev = tail;
// 更新尾节点引用
tail = newNode;

// 现在链表变成了 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

▶ 🎨 代码可视化动画🎨

在双链表的指定位置插入新元素需要调整前驱节点和后继节点的指针：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

// 在第 3 个节点后面插入新节点 66
// 找到第 3 个节点
DoublyListNode p = head;
for (int i = 0; i < 2; i++) {
    p = p.next;
}

// 组装新节点
DoublyListNode newNode = new DoublyListNode(66);
newNode.next = p.next;
newNode.prev = p;

// 插入新节点
p.next.prev = newNode;
p.next = newNode;

// 现在链表变成了 1 -> 2 -> 3 -> 66 -> 4 -> 5
```

▶ 🎨 代码可视化动画🎨

## 删

在双链表中删除节点时，需要调整前驱节点和后继节点的指针来摘除目标节点：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

// 删除第 4 个节点
// 先找到第 3 个节点
DoublyListNode p = head;
for (int i = 0; i < 2; i++) {
    p = p.next;
}

// 现在 p 指向第 3 个节点，我们把它后面那个节点摘除出去
DoublyListNode toDelete = p.next;

// 把 toDelete 从链表中摘除
p.next = toDelete.next;
toDelete.next.prev = p;

// 把 toDelete 的前后指针都置为 null 是个好习惯（可选）
toDelete.next = null;
toDelete.prev = null;

// 现在链表变成了 1 -> 2 -> 3 -> 5
```

### ▶ 😊 代码可视化动画😊

在双链表头部删除元素需要调整头节点的指针：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

// 删除头结点
DoublyListNode toDelete = head;
head = head.next;
head.prev = null;

// 清理已删除节点的指针
toDelete.next = null;

// 现在链表变成了 2 -> 3 -> 4 -> 5
```

### ▶ 😊 代码可视化动画😊

在单链表中，由于缺乏前驱指针，所以删除尾节点时需要遍历到倒数第二个节点，操作它的 `next` 指针，才能把尾节点摘除出去。

但在双链表中，由于每个节点都存储了前驱节点的指针，所以我们可以直接操作尾节点，把它自己从链表中摘除：

```
// 创建一条双链表
DoublyListNode head = createDoublyLinkedList(new int[]{1, 2, 3, 4, 5});

// 删除尾节点
DoublyListNode p = head;
// 找到尾结点
while (p.next != null) {
    p = p.next;
}

// 现在 p 指向尾节点
// 把尾节点从链表中摘除
p.prev.next = null;

// 把被删结点的指针都断开是个好习惯（可选）
p.prev = null;

// 现在链表变成了 1 -> 2 -> 3 -> 4
```

---

▶  [代码可视化动画](#) 

---

## 接下来

在下一篇文章中，我们分别用单链表和双链表实现一个拥有增删查改等基本操作的 `MyLinkedList`，并且会使用「虚拟头结点」技巧简化代码逻辑，避免处理头尾指针为空情况的边界情况。

# 链表代码实现

阅读本文前，你需要先学习：

- [链表（链式存储）基础](#)

## 几个关键点

下面我会分别用双链表和单链给出一个简单的 [MyLinkedList](#) 代码实现，包含了基本的增删查改功能。这里给出几个关键点，等会你看代码的时候可以着重注意一下。

### 关键点一、同时持有头尾节点的引用

在力扣做题时，一般题目给我们传入的就是单链表的头指针。但是在实际开发中，用的都是双链表，而双链表一般会同时持有头尾节点的引用。

因为在软件开发中，在容器尾部添加元素是个非常高频的操作，双链表持有尾部节点的引用，就可以在  $O(1)$  的时间复杂度内完成尾部添加元素的操作。

对于单链表来说，持有尾部节点的引用也有优化效果。比如你要在单链表尾部添加元素，如果没有尾部节点的引用，你就需要遍历整个链表找到尾部节点，时间复杂度是  $O(n)$ ；如果有尾部节点的引用，就可以在  $O(1)$  的时间复杂度内完成尾部添加元素的操作。

细心的读者可能会说，即便如此，如果删除一次单链表的尾结点，那么之前尾结点的引用就失效了，还是需要遍历一遍链表找到尾结点。

是的，但你再仔细想想，删除单链表尾结点的时候，是不是也得遍历到倒数第二个节点（尾结点的前驱），才能通过指针操作把尾结点删掉？那么这个时候，你不就可以顺便把尾结点的引用给更新了吗？

### 关键点二、虚拟头尾节点

在上一篇文章 [链表基础](#) 中我提到过「虚拟头尾节点」技巧，它的原理很简单，就是在创建双链表时就创建一个虚拟头节点和一个虚拟尾节点，无论双链表是否为空，这两个节点都存在。这样就不会出现空指针的问题，可以避免很多边界情况的处理。

举例来说，假设虚拟头尾节点分别是 [dummyHead](#) 和 [dummyTail](#)，那么一条空的双链表长这样：

```
dummyHead <-> dummyTail
```

如果你添加 [1, 2, 3](#) 几个元素，那么链表长这样：

```
dummyHead <-> 1 <-> 2 <-> 3 <-> dummyTail
```

你以前要把在头部插入元素、在尾部插入元素和在中间插入元素几种情况分开讨论，现在有了头尾虚拟节点，无论链表是否为空，都只需要考虑在中间插入元素的情况就可以了，这样代码会简洁很多。

当然，虚拟头结点会多占用一点内存空间，但是比起给你解决的麻烦，这点空间消耗是划算的。

对于单链表，虚拟头结点有一定的简化作用，但虚拟尾节点没有太大作用。

虚拟节点是你内部实现数据结构的技巧，对外是不可见的。比如按照索引获取元素的 `get(index)` 方法，都是从真实节点开始计算索引，而不是从虚拟节点开始计算。

### 关键点三、内存泄露？

在前文 [动态数组实现](#) 中，我提到了删除元素时，要注意内存泄露的问题。那么在链表中，删除元素会不会也有内存泄露的问题呢？

尤其是这样的写法，你觉得有没有问题：

```
// 假设单链表头结点 head = 1 -> 2 -> 3 -> 4 -> 5  
  
// 删除单链表头结点  
head = head.next;  
  
// 此时 head = 2 -> 3 -> 4 -> 5
```

细心的读者可能认为这样写会有内存泄露的问题，因为原来的那个头结点 1 的 `next` 指针没有断开，依然指向着节点 2。

但实际上这样写是 OK 的，因为 Java 的垃圾回收的判断机制是看这个对象是否被别人引用，而并不会 care 这个对象是否还引用着别人。

那个节点 1 的 `next` 指针确实还指向着节点 2，但是并没有别的指针引用节点 1 了，所以节点 1 最终会被垃圾回收器回收释放。所以说这个场景和数组中删除元素的场景是不一样的，你可以再仔细思考一下。

不过呢，删除节点时，最好还是把被删除节点的指针都置为 `null`，这是个好习惯，不会有什么代价，还可能避免一些潜在的问题。所以在下面的实现中，无论是否有必要，我都会把被删除节点上的指针置为 `null`。

你可以借助力扣第 707 题「设计链表」来验证自己的实现是否正确。

### 双链表代码实现

```
import java.util.NoSuchElementException;  
  
public class MyLinkedList<E> {  
    // 虚拟头尾节点  
    final private Node<E> head, tail;  
    private int size;  
  
    // 双链表节点  
    private static class Node<E> {  
        E val;  
        Node<E> next;  
        Node<E> prev;  
  
        Node(E val) {  
            this.val = val;  
        }  
    }  
  
    // 构造函数初始化虚拟头尾节点
```

```
public MyLinkedList() {
    this.head = new Node<>(null);
    this.tail = new Node<>(null);
    head.next = tail;
    tail.prev = head;
    this.size = 0;
}

// ***** 增 *****

public void addLast(E e) {
    Node<E> x = new Node<E>(e);
    Node<E> temp = tail.prev;
    // temp <-> tail
    temp.next = x;
    x.prev = temp;

    x.next = tail;
    tail.prev = x;
    // temp <-> x <-> tail
    size++;
}

public void addFirst(E e) {
    Node<E> x = new Node<E>(e);
    Node<E> temp = head.next;
    // head <-> temp
    temp.prev = x;
    x.next = temp;

    head.next = x;
    x.prev = head;
    // head <-> x <-> temp
    size++;
}

public void add(int index, E element) {
    checkPositionIndex(index);
    if (index == size) {
        addLast(element);
        return;
    }

    // 找到 index 对应的 Node
    Node<E> p = getNode(index);
    Node<E> temp = p.prev;
    // temp <-> p

    // 新要插入的 Node
    Node<E> x = new Node<E>(element);

    p.prev = x;
    temp.next = x;

    x.prev = temp;
    x.next = p;

    // temp <-> x <-> p
}
```

```
        size++;
    }

    // ***** 删 *****

    public E removeFirst() {
        if (size < 1) {
            throw new NoSuchElementException();
        }
        // 虚拟节点的存在是我们不用考虑空指针的问题
        Node<E> x = head.next;
        Node<E> temp = x.next;
        // head <-> x <-> temp
        head.next = temp;
        temp.prev = head;

        x.prev = null;
        x.next = null;
        // head <-> temp

        size--;
        return x.val;
    }

    public E removeLast() {
        if (size < 1) {
            throw new NoSuchElementException();
        }
        Node<E> x = tail.prev;
        Node<E> temp = tail.prev.prev;
        // temp <-> x <-> tail

        tail.prev = temp;
        temp.next = tail;

        x.prev = null;
        x.next = null;
        // temp <-> tail

        size--;
        return x.val;
    }

    public E remove(int index) {
        checkElementIndex(index);
        // 找到 index 对应的 Node
        Node<E> x = getNode(index);
        Node<E> prev = x.prev;
        Node<E> next = x.next;
        // prev <-> x <-> next
        prev.next = next;
        next.prev = prev;

        x.prev = x.next = null;
        // prev <-> next

        size--;
    }
}
```

```
        return x.val;
    }

// ***** 查 *****
public E get(int index) {
    checkElementIndex(index);
    // 找到 index 对应的 Node
    Node<E> p = getNode(index);

    return p.val;
}

public E getFirst() {
    if (size < 1) {
        throw new NoSuchElementException();
    }

    return head.next.val;
}

public E getLast() {
    if (size < 1) {
        throw new NoSuchElementException();
    }

    return tail.prev.val;
}

// ***** 改 *****
public E set(int index, E val) {
    checkElementIndex(index);
    // 找到 index 对应的 Node
    Node<E> p = getNode(index);

    E oldVal = p.val;
    p.val = val;

    return oldVal;
}

// ***** 其他工具函数 *****
public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}

private Node<E> getNode(int index) {
    checkElementIndex(index);
    Node<E> p = head.next;
    // TODO: 可以优化，通过 index 判断从 head 还是 tail 开始遍历
    for (int i = 0; i < index; i++) {
        p = p.next;
    }
}
```

```
        return p;
    }

    private boolean isElementIndex(int index) {
        return index >= 0 && index < size;
    }

    private boolean isPositionIndex(int index) {
        return index >= 0 && index <= size;
    }

    // 检查 index 索引位置是否可以存在元素
    private void checkElementIndex(int index) {
        if (!isElementIndex(index))
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
    }

    // 检查 index 索引位置是否可以添加元素
    private void checkPositionIndex(int index) {
        if (!isPositionIndex(index))
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
    }

    private void display() {
        System.out.println("size = " + size);
        for (Node<E> p = head.next; p != tail; p = p.next) {
            System.out.print(p.val + " <-> ");
        }
        System.out.println("null");
        System.out.println();
    }

    public static void main(String[] args) {
        MyLinkedList<Integer> list = new MyLinkedList<>();
        list.addLast(1);
        list.addLast(2);
        list.addLast(3);
        list.addFirst(0);
        list.add(2, 100);

        list.display();
        // size = 5
        // 0 <-> 1 <-> 100 <-> 2 -> 3 -> null
    }
}
```

## 单链表代码实现

```
import java.util.NoSuchElementException;

public class MyLinkedList2<E> {

    private static class Node<E> {
        E val;
```

```
Node<E> next;

Node(E val) {
    this.val = val;
    this.next = null;
}
}

private Node<E> head;
// 实际的尾部节点引用
private Node<E> tail;
private int size;

public MyLinkedList2() {
    this.head = new Node<E>(null);
    this.tail = head;
    this.size = 0;
}

public void addFirst(E e) {
    Node<E> newNode = new Node<E>(e);
    newNode.next = head.next;
    head.next = newNode;
    if (size == 0) {
        tail = newNode;
    }
    size++;
}

public void addLast(E e) {
    Node<E> newNode = new Node<E>(e);
    tail.next = newNode;
    tail = newNode;
    size++;
}

public void add(int index, E element) {
    checkPositionIndex(index);

    if (index == size) {
        addLast(element);
        return;
    }

    Node<E> prev = head;
    for (int i = 0; i < index; i++) {
        prev = prev.next;
    }
    Node<E> newNode = new Node<E>(element);
    newNode.next = prev.next;
    prev.next = newNode;
    size++;
}

public E removeFirst() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    Node<E> first = head.next;
```

```
head.next = first.next;
if (size == 1) {
    tail = head;
}
size--;
return first.val;
}

public E removeLast() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }

    Node<E> prev = head;
    while (prev.next != tail) {
        prev = prev.next;
    }
    E val = tail.val;
    prev.next = null;
    tail = prev;
    size--;
    return val;
}

public E remove(int index) {
    checkElementIndex(index);

    Node<E> prev = head;
    for (int i = 0; i < index; i++) {
        prev = prev.next;
    }

    Node<E> nodeToRemove = prev.next;
    prev.next = nodeToRemove.next;
    // 删除的是最后一个元素
    if (index == size - 1) {
        tail = prev;
    }
    size--;
    return nodeToRemove.val;
}

// ***** 查 *****
public E getFirst() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    return head.next.val;
}

public E getLast() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    return getNode(size - 1).val;
}

public E get(int index) {
```

```
    checkElementIndex(index);
    Node<E> p = getNode(index);
    return p.val;
}

// ***** 改 *****

public E set(int index, E element) {
    checkElementIndex(index);
    Node<E> p = getNode(index);

    E oldVal = p.val;
    p.val = element;

    return oldVal;
}

// ***** 其他工具函数 *****
public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}

private boolean isElementIndex(int index) {
    return index >= 0 && index < size;
}

private boolean isPositionIndex(int index) {
    return index >= 0 && index <= size;
}

// 检查 index 索引位置是否可以存在元素
private void checkElementIndex(int index) {
    if (!isElementIndex(index))
        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
}

// 检查 index 索引位置是否可以添加元素
private void checkPositionIndex(int index) {
    if (!isPositionIndex(index))
        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " +
size);
}

// 返回 index 对应的 Node
// 注意: 请保证传入的 index 是合法的
private Node<E> getNode(int index) {
    Node<E> p = head.next;
    for (int i = 0; i < index; i++) {
        p = p.next;
    }
    return p;
}

public static void main(String[] args) {
```

```
MyLinkedList2<Integer> list = new MyLinkedList2<>();
list.addFirst(1);
list.addFirst(2);
list.addLast(3);
list.addLast(4);
list.add(2, 5);

System.out.println(list.removeFirst()); // 2
System.out.println(list.removeLast()); // 4
System.out.println(list.remove(1)); // 5

System.out.println(list.getFirst()); // 1
System.out.println(list.getLast()); // 3
System.out.println(list.get(1)); // 3
}

}
```

# 队列/栈基本原理

阅读本文前，你需要先学习：

- 链表（链式存储）基础
- 数组（顺序存储）基础

计算机的两种存储方式，顺序存储（数组）和链式存储（链表）都讲完了，之后的所有数据结构都是基于这两种存储方式之上玩花活。

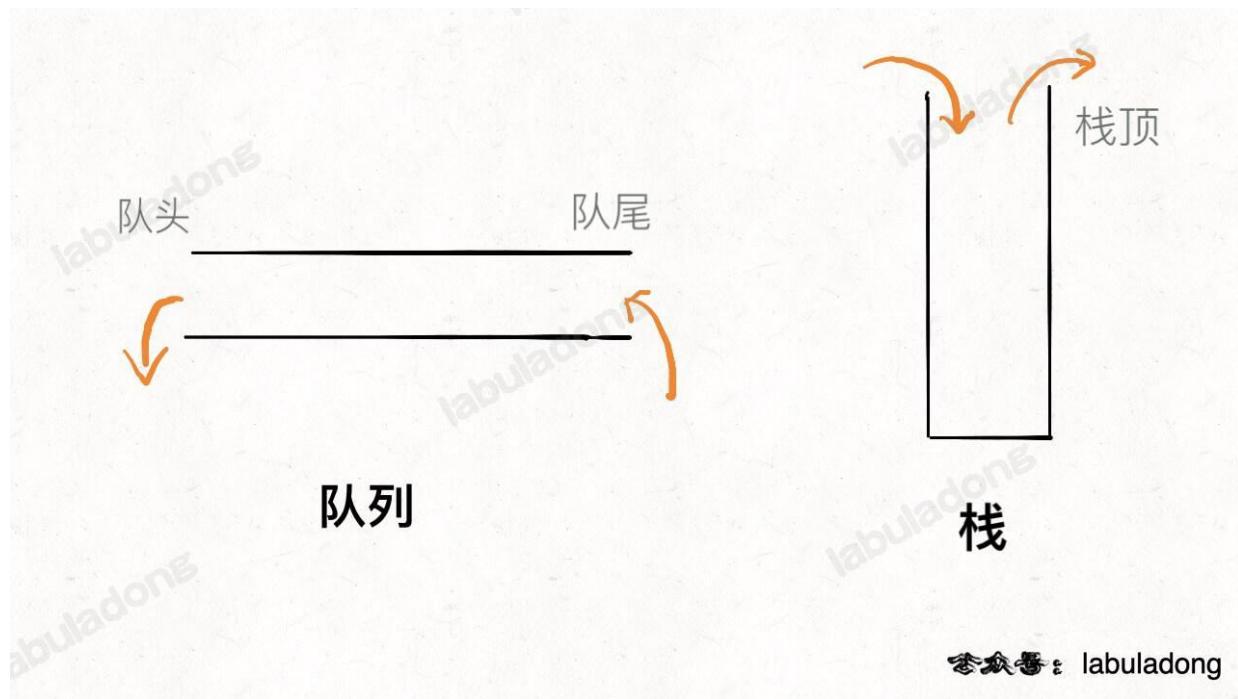
本文讲解队列和栈的基本原理，后面的文章会讲解如何用代码具体实现。

先说概念吧，其实队列和栈都是「操作受限」的数据结构。说它操作受限，主要是和基本的数组和链表相比，它们提供的 API 是不完整的。

比方说我们前面实现的数组和链表，增删查改的 API 都实现过了，你可以对任意一个索引元素进行增删查改，只要索引不越界，就随便你。

但是对于队列和栈，它们的操作是受限的：**队列只能在一端插入元素，另一端删除元素；栈只能在某一端插入和删除元素。**

形象地说，队列只允许在队尾插入元素，在队头删除元素，栈只允许在栈顶插入元素，从栈顶删除元素：



队列就像排队买票，先来的先买，后来的后买；栈就像一摞盘子，最先放的压在最下面，最后放的留在最上面，拿的时候也是最上面的先被拿走。所以我们常说，队列是一种「先进先出」的数据结构，栈是一种「先进后出」的数据结构，就是这个道理。

当然，这个图中把栈竖着画，队列横着画，只是为了更形象，但实际上它们底层都是数组和链表实现的，后面会讲到。

这两种数据结构的基本 API 如下：

```
// 队列的基本 API
class MyQueue<E> {
    // 向队尾插入元素，时间复杂度 O(1)
    void push(E e);

    // 从队头删除元素，时间复杂度 O(1)
    E pop();

    // 查看队头元素，时间复杂度 O(1)
    E peek();

    // 返回队列中的元素个数，时间复杂度 O(1)
    int size();
}

// 栈的基本 API
class MyStack<E> {
    // 向栈顶插入元素，时间复杂度 O(1)
    void push(E e);

    // 从栈顶删除元素，时间复杂度 O(1)
    E pop();

    // 查看栈顶元素，时间复杂度 O(1)
    E peek();

    // 返回栈中的元素个数，时间复杂度 O(1)
    int size();
}
```

不同编程语言中，队列和栈提供的方法名称可能不一样，但每个方法的效果肯定是一样的。

有些语言的标准库可能没有直接提供队列和栈，你可以自己用数组或者链表模拟出队列和栈的效果。下一章我就会先带你用链表实现队列和栈。

# 用链表实现队列/栈

阅读本文前，你需要先学习：

- 队列/栈基本原理

## 用链表实现栈

一些读者应该已经知道该怎么用链表作为底层数据结构实现队列和栈了。因为实在是太简单了，直接调用双链表的 API 就可以了。

注意我这里是直接用的 Java 标准库的 `LinkedList`，如果你用之前我们实现的 `MyLinkedList`，也是一样的。

```
import java.util.LinkedList;

// 用链表作为底层数据结构实现栈
public class MyLinkedStack<E> {
    private final LinkedList<E> list = new LinkedList<E>();

    // 向栈顶加入元素，时间复杂度 O(1)
    public void push(E e) {
        list.addLast(e);
    }

    // 从栈顶弹出元素，时间复杂度 O(1)
    public E pop() {
        return list.removeLast();
    }

    // 查看栈顶元素，时间复杂度 O(1)
    public E peek() {
        return list.getLast();
    }

    // 返回栈中的元素个数，时间复杂度 O(1)
    public int size() {
        return list.size();
    }

    public static void main(String[] args) {
        MyLinkedStack<Integer> stack = new MyLinkedStack<Integer>();
        stack.push(1);
        stack.push(2);
        stack.push(3);

        System.out.println(stack.peek()); // 3
        System.out.println(stack.pop()); // 3
        System.out.println(stack.peek()); // 2
    }
}
```

上面这段代码相当于是把双链表的尾部作为栈顶，在双链表尾部增删元素的时间复杂度都是  $O(1)$ ，符合要求。

当然，你也可以把双链表的头部作为栈顶，因为双链表头部增删元素的时间复杂度也是  $O(1)$ ，所以这样实现也是一样的。只要做几个修改 `addLast -> addFirst`, `removeLast -> removeFirst`, `getLast -> getFirst` 就行了。

## 用链表实现队列

同理，用链表实现队列也是一样的，也直接调用双链表的 API 就可以了：

```
import java.util.LinkedList;

// 用链表作为底层数据结构实现队列
public class MyLinkedQueue<E> {
    private final LinkedList<E> list = new LinkedList<>();

    // 向队尾插入元素，时间复杂度 O(1)
    public void push(E e) {
        list.addLast(e);
    }

    // 从队头删除元素，时间复杂度 O(1)
    public E pop() {
        return list.removeFirst();
    }

    // 查看队头元素，时间复杂度 O(1)
    public E peek() {
        return list.getFirst();
    }

    // 返回队列中的元素个数，时间复杂度 O(1)
    public int size() {
        return list.size();
    }

    public static void main(String[] args) {
        MyLinkedQueue<Integer> queue = new MyLinkedQueue<>();
        queue.push(1);
        queue.push(2);
        queue.push(3);

        System.out.println(queue.peek()); // 1
        System.out.println(queue.pop()); // 1
        System.out.println(queue.pop()); // 2
        System.out.println(queue.peek()); // 3
    }
}
```

上面这段代码相当于是把双链表的尾部作为队尾，把双链表的头部作为队头，在双链表的头尾增删元素的复杂度都是  $O(1)$ ，符合队列 API 的要求。

当然，你也可以反过来，把双链表的头部作为队尾，双链表的尾部作为队头。类似栈的实现，只要改一改 list 的调用方法就行了。

## 文末思考

双链表他比较牛，队列和栈的 API 考不倒它。但是你想一下，数组实现队列的时候，会不会有问题？

队列 API 要求一端增加元素，一端删除元素，而数组的头部无论是增加还是删除元素，时间复杂度都是  $O(n)$ 。这种情况下，有没有办法优化呢？

你可以思考一下，下一章我会告诉你答案。

# 环形数组技巧

阅读本文前，你需要先学习：

- [数组（顺序存储）基础](#)

环形数组技巧利用求模（余数）运算，将普通数组变成逻辑上的环形数组，可以让我们用  $O(1)$  的时间在数组头部增删元素。

## 环形数组原理

数组可能是环形的么？不可能。数组就是一块线性连续的内存空间，怎么可能有环的概念？

但是，我们可以在「逻辑上」把数组变成环形的，比如下面这段代码：

```
// 长度为 5 的数组
int[] arr = new int[]{1, 2, 3, 4, 5};
int i = 0;
// 模拟环形数组，这个循环永远不会结束
while (i < arr.length) {
    System.out.println(arr[i]);
    i = (i + 1) % arr.length;
}
```

这段代码的关键在于求模运算 `%`，也就是求余数。当 `i` 到达数组末尾元素时，`i + 1` 和 `arr.length` 取余数又会变成 0，即会回到数组头部，这样就在逻辑上形成了一个环形数组，永远遍历不完。

这就是环形数组技巧。这个技巧如何帮助我们在  $O(1)$  的时间在数组头部增删元素呢？

是这样，假设我们现在有一个长度为 6 的数组，现在其中只装了 3 个元素，如下（未装元素的位置用 `_` 标识）：

```
[1, 2, 3, _, _, _]
```

现在我们要在数组头部删除元素 `1`，那么我们可以把数组变成这样：

```
[_, 2, 3, _, _, _]
```

即，我们仅仅把元素 `1` 的位置标记为空，但并不做数据搬移。

此时，如果我们要在数组头部增加元素 `4` 和元素 `5`，我们可以把数组变成这样：

```
[4, 2, 3, _, _, 5]
```

你可以看到，当头部没有位置添加新元素时，它转了一圈，把新元素加到尾部了。

上面只是让大家对环形数组有一个直观地印象，环形数组的关键在于，它维护了两个指针 `start` 和 `end`，`start` 指向第一个有效元素的索引，`end` 指向最后一个有效元素的下一个位置索引。

这样，当我们在数组头部添加或删除元素时，只需要移动 `start` 索引，而在数组尾部添加或删除元素时，只需要移动 `end` 索引。

当 `start`, `end` 移动超出数组边界 (`< 0` 或 `>= arr.length`) 时，我们可以通过求模运算 `%` 让它们转一圈到数组头部或尾部继续工作，这样就实现了环形数组的效果。

## 动手环节

纸上得来终觉浅，绝知此事要躬行。

我在可视化面板实现了一个简单的环形数组，你可以点击下面代码中的 `arr.addLast` 或 `arr.addFirst`，注意观察 `start`, `end` 指针以及 `arr` 数组中元素的变化：

### ▶ 代码可视化动画

## 代码实现

在我的代码中，环形数组的区间被定义为左闭右开的，即 `[start, end)` 区间包含数组元素。所以其他的方法都是以左闭右开区间为基础实现的。

那么肯定就会有读者问，为啥要左闭右开，我就是想两端都开，或者两端都闭，不行么？

在 [滑动窗口算法核心框架](#) 中定义滑动窗口的边界时也会有类似的问题，这里我也解释一下。

理论上，你可以随机设计区间的开闭，但一般设计为左闭右开区间是最方便处理的。

因为这样初始化 `start = end = 0` 时区间 `[0, 0)` 中没有元素，但只要让 `end` 向右移动（扩大）一位，区间 `[0, 1)` 就包含一个元素 `0` 了。

如果你设置为两端都开的区间，那么让 `end` 向右移动一位后开区间 `(0, 1)` 仍然没有元素；如果你设置为两端都闭的区间，那么初始区间 `[0, 0]` 就已经包含了一个元素。这两种情况都会给边界处理带来不必要的麻烦，如果你非要使用的话，需要在代码中做一些特殊处理。

最后，我给出一个支持泛型的 Java 实现，你可以参考一下：

```
public class CycleArray<T> {
    private T[] arr;
    private int start;
    private int end;
    private int count;
    private int size;

    public CycleArray() {
        this(1);
    }

    @SuppressWarnings("unchecked")
    public CycleArray(int size) {
        this.size = size;
    }
```

```

// 因为 Java 不支持直接创建泛型数组，所以这里使用了类型转换
this.arr = (T[]) new Object[size];
// start 指向第一个有效元素的索引，闭区间
this.start = 0;
// 切记 end 是一个开区间，
// 即 end 指向最后一个有效元素的下一个位置索引
this.end = 0;
this.count = 0;
}

// 自动扩容辅助函数
@SuppressWarnings("unchecked")
private void resize(int newSize) {
    // 创建新的数组
    T[] newArr = (T[]) new Object[newSize];
    // 将旧数组的元素复制到新数组中
    for (int i = 0; i < count; i++) {
        newArr[i] = arr[(start + i) % size];
    }
    arr = newArr;
    // 重置 start 和 end 指针
    start = 0;
    end = count;
    size = newSize;
}

// 在数组头部添加元素，时间复杂度 O(1)
public void addFirst(T val) {
    // 当数组满时，扩容为原来的两倍
    if (isFull()) {
        resize(size * 2);
    }
    // 因为 start 是闭区间，所以先左移，再赋值
    start = (start - 1 + size) % size;
    arr[start] = val;
    count++;
}

// 删除数组头部元素，时间复杂度 O(1)
public void removeFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("Array is empty");
    }
    // 因为 start 是闭区间，所以先赋值，再右移
    arr[start] = null;
    start = (start + 1) % size;
    count--;
    // 如果数组元素数量减少到原大小的四分之一，则减小数组大小为一半
    if (count > 0 && count == size / 4) {
        resize(size / 2);
    }
}

// 在数组尾部添加元素，时间复杂度 O(1)
public void addLast(T val) {
    if (isFull()) {
        resize(size * 2);
    }
    // 因为 end 是开区间，所以是先赋值，再右移
}

```

```

        arr[end] = val;
        end = (end + 1) % size;
        count++;
    }

    // 删除数组尾部元素，时间复杂度 O(1)
    public void removeLast() {
        if (isEmpty()) {
            throw new IllegalStateException("Array is empty");
        }
        // 因为 end 是开区间，所以先左移，再赋值
        end = (end - 1 + size) % size;
        arr[end] = null;
        count--;
        // 缩容
        if (count > 0 && count == size / 4) {
            resize(size / 2);
        }
    }

    // 获取数组头部元素，时间复杂度 O(1)
    public T getFirst() {
        if (isEmpty()) {
            throw new IllegalStateException("Array is empty");
        }
        return arr[start];
    }

    // 获取数组尾部元素，时间复杂度 O(1)
    public T getLast() {
        if (isEmpty()) {
            throw new IllegalStateException("Array is empty");
        }
        // end 是开区间，指向的是下一个元素的位置，所以要减 1
        return arr[(end - 1 + size) % size];
    }

    public boolean isFull() {
        return count == size;
    }

    public int size() {
        return count;
    }

    public boolean isEmpty() {
        return count == 0;
    }
}

```

## 思考题

我们都说，在数组增删头部元素的时间复杂度是  $O(N)$ ，因为需要搬移元素。但是，如果我们使用环形数组，其实是可以实现在  $O(1)$  的时间复杂度内增删头部元素的。

当然，上面实现的这个环形数组只提供了 `addFirst`, `removeFirst`, `addLast`, `removeLast` 这几个方法，并没有提供 [我们之前实现的动态数组](#) 的某些方法，比如删除指定索引的元素，获取指定索引的元素，在指定索引插入元素等等。

但是你可以思考一下，难道环形数组实现不了这些方法么？环形数组实现这些方法，时间复杂度相比普通数组，有退化吗？

好像没有吧。

环形数组也可以删除指定索引的元素，也要做数据搬移，和普通数组一样，复杂度是  $O(N)$ ；

环形数组也可以获取指定索引的元素（随机访问），只不过不是直接访问对应索引，而是要通过 `start` 计算出真实索引，但计算和访问的时间复杂度依然是  $O(1)$ ；

环形数组也可以在指定索引插入元素，当然也要做数据搬移，和普通数组一样，复杂度是  $O(N)$ 。

你可以思考一下是不是这样。如果是这样，为什么编程语言的标准库中提供的动态数组容器底层并没有用环形数组技巧。

# 用数组实现队列/栈

阅读本文前，你需要先学习：

- 队列/栈基本原理

这篇文章带大家用数组作为底层数据结构实现队列和栈。

## 用数组实现栈

先用数组实现栈，这个不难，你把动态数组的尾部作为栈顶，然后调用动态数组的 API 就行了。因为数组尾部增删元素的时间复杂度都是  $O(1)$ ，符合栈的要求。

注意我这里用的是 Java 标准库的 `ArrayList`，如果你想用之前我们实现的 `MyArrayList`，也是一样的：

```
// 用数组作为底层数据结构实现栈
public class MyArrayStack<E> {
    private ArrayList<E> list = new ArrayList<E>();

    // 向栈顶加入元素，时间复杂度 O(1)
    public void push(E e) {
        list.add(e);
    }

    // 从栈顶弹出元素，时间复杂度 O(1)
    public E pop() {
        return list.remove(list.size() - 1);
    }

    // 查看栈顶元素，时间复杂度 O(1)
    public E peek() {
        return list.get(list.size() - 1);
    }

    // 返回栈中的元素个数，时间复杂度 O(1)
    public int size() {
        return list.size();
    }
}
```

按照我们之前实现 `MyArrayList` 的逻辑，是不行的。因为数组头部增删元素的时间复杂度都是  $O(n)$ ，不符合要求。

但是我们可以改用前文 [环形数组技巧](#) 中实现的 `CycleArray` 类，这个数据结构在头部增删元素的时间复杂度是  $O(1)$ ，符合栈的要求。

你直接调用 `CycleArray` 的 `addFirst` 和 `removeFirst` 方法实现栈的 API 就行，我这里就不写了。

## 用数组实现队列

有了前文 [环形数组](#) 中实现的 `CycleArray` 类，用数组作为底层数据结构实现队列就不难了吧。直接复用我们实现的 `CycleArray`，就可以实现标准队列了。当然，一些编程语言也有内置的环形数组实现，你也可以自行搜索使用：

```
public class MyArrayQueue<E> {
    private CycleArray<E> arr;

    public MyArrayQueue() {
        arr = new CycleArray<E>();
    }

    public void push(E t) {
        arr.addLast(t);
    }

    public E pop() {
        return arr.removeFirst();
    }

    public E peek() {
        return arr.getFirst();
    }

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

# 双端队列 (Deque) 原理及实现

阅读本文前，你需要先学习：

- 队列/栈基本原理
- 环形数组技巧

## 基本原理

如果你理解了前面讲解的内容，这个双端队列其实没啥可讲的了。所谓双端队列，主要是对比标准队列（FIFO 先进先出队列）多了一些操作罢了：

```
class MyDeque<E> {  
    // 从队头插入元素，时间复杂度 O(1)  
    void addFirst(E e);  
  
    // 从队尾插入元素，时间复杂度 O(1)  
    void addLast(E e);  
  
    // 从队头删除元素，时间复杂度 O(1)  
    E removeFirst();  
  
    // 从队尾删除元素，时间复杂度 O(1)  
    E removeLast();  
  
    // 查看队头元素，时间复杂度 O(1)  
    E peekFirst();  
  
    // 查看队尾元素，时间复杂度 O(1)  
    E peekLast();  
}
```

**标准队列** 只能在队尾插入元素，队头删除元素，而双端队列的队头和队尾都可以插入或删除元素。

普通队列就好比排队买票，先来的先买，后来的后买；而双端队列就好比一个过街天桥，两端都可以随意进出。当然，双端队列的元素就不再满足「先进先出」了，因为它比较灵活嘛。

在做算法题的场景中，双端队列用的不算很多。感觉只有 Python 用到的多一些，因为 Python 标准库没有提供内置的栈和队列，一般会用双端队列来模拟标准队列。

## 用链表实现双端队列

很简单吧，直接复用 [我们之前实现的 MyLinkedList](#) 或者标准库提供的 [LinkedList](#) 就行了。因为双链表本就支持 \$O(1)\$ 时间复杂度在链表的头尾增删元素：

```
import java.util.LinkedList;  
  
public class MyListDeque<E> {  
    private LinkedList<E> list = new LinkedList<>();  
  
    // 从队头插入元素，时间复杂度 O(1)
```

```

void addFirst(E e) {
    list.addFirst(e);
}

// 从队尾插入元素，时间复杂度 O(1)
void addLast(E e) {
    list.addLast(e);
}

// 从队头删除元素，时间复杂度 O(1)
E removeFirst() {
    return list.removeFirst();
}

// 从队尾删除元素，时间复杂度 O(1)
E removeLast() {
    return list.removeLast();
}

// 查看队头元素，时间复杂度 O(1)
E peekFirst() {
    return list.getFirst();
}

// 查看队尾元素，时间复杂度 O(1)
E peekLast() {
    return list.getLast();
}

public static void main(String[] args) {
    MyListDeque<Integer> deque = new MyListDeque<>();
    deque.addFirst(1);
    deque.addFirst(2);
    deque.addLast(3);
    deque.addLast(4);

    System.out.println(deque.removeFirst()); // 2
    System.out.println(deque.removeLast()); // 4
    System.out.println(deque.peekFirst()); // 1
    System.out.println(deque.peekLast()); // 3
}
}

```

## 用数组实现双端队列

也很简单吧，直接复用我们在[环形数组技巧](#)中实现的[CycleArray](#)提供的方法就行了。环形数组头尾增删元素的复杂度都是\$O(1)\$：

```

class MyArrayDeque<E> {
    private CycleArray<E> arr = new CycleArray<>();

    // 从队头插入元素，时间复杂度 O(1)
    void addFirst(E e) {
        arr.addFirst(e);
    }
}

```

```
// 从队尾插入元素，时间复杂度 O(1)
void addLast(E e) {
    arr.addLast(e);
}

// 从队头删除元素，时间复杂度 O(1)
E removeFirst() {
    return arr.removeFirst();
}

// 从队尾删除元素，时间复杂度 O(1)
E removeLast() {
    return arr.removeLast();
}

// 查看队头元素，时间复杂度 O(1)
E peekFirst() {
    return arr.getFirst();
}

// 查看队尾元素，时间复杂度 O(1)
E peekLast() {
    return arr.getLast();
}
```

# 哈希表核心原理

阅读本文前，你需要先学习：

- 数组（顺序存储）基础

首先，我需要先阐明一个初学者很容易犯的概念错误。

请问，哈希表和我们常说的 Map（键值映射）是不是同一个东西？不是。

这一点用 Java 来讲解就很清楚，Map 是一个 Java 接口，仅仅声明了若干个方法，并没有给出方法的具体实现：

```
interface Map<K, V> {  
    V get(K key);  
    void put(K key, V value);  
    V remove(K key);  
    // ...  
}
```

Map 接口本身只定义了键值映射的一系列操作，HashMap 这种数据结构根据自身特点实现了这些操作。还有其他数据结构也实现了这个接口，比如 TreeMap、LinkedHashMap 等等。

换句话说，你可以说 HashMap 的 get, put, remove 方法的复杂度都是  $O(1)$  的，但你不能说 Map 接口的复杂度都是  $O(1)$ 。因为如果换成其他的实现类，比如底层用二叉树结构实现的 TreeMap，这些方法的复杂度就变成  $O(\log N)$  了。

我为什么要强调这一点呢？主要是针对使用非 Java 语言的读者。

其他编程语言可能没有 Java 这么清晰的接口定义，所以很容易让读者把哈希表和 Map 键值对混为一谈，听到键值对操作，就认为其增删查改的复杂度一定是  $O(1)$ 。这是不对的，具体要看这个底层的数据结构是如何实现键值操作的。

那么这一章节我会带大家动手实现一个哈希表，探讨哈希表为什么能做到增删查改  $O(1)$  复杂度，以及解决哈希冲突的两种办法。

## 哈希表的基本原理

哈希表可以理解为一个加强版的数组。

数组可以通过索引在  $O(1)$  的时间复杂度内查找到对应元素，索引是一个非负整数。

哈希表是类似的，可以通过 key 在  $O(1)$  的时间复杂度内查找到这个 key 对应的 value。key 的类型可以是数字、字符串等多种类型。

怎么做的？特别简单，哈希表的底层实现就是一个数组（我们不妨称之为 table）。它把这个 key 通过一个哈希函数（我们不妨称之为 hash）转化成数组里面的索引，然后增删查改操作和数组基本相同：

```
// 哈希表伪码逻辑  
class MyHashMap {  
  
    private Object[] table;
```

```

// 增/改, 复杂度 O(1)
public void put(K key, V value) {
    int index = hash(key);
    table[index] = value;
}

// 查, 复杂度 O(1)
public V get(K key) {
    int index = hash(key);
    return table[index];
}

// 删, 复杂度 O(1)
public void remove(K key) {
    int index = hash(key);
    table[index] = null;
}

// 哈希函数, 把 key 转化成 table 中的合法索引
// 时间复杂度必须是 O(1), 才能保证上述方法的复杂度都是 O(1)
private int hash(K key) {
    // ...
}
}

```

具体实现上有不少细节需要处理，比如哈希函数的设计、哈希冲突的处理等等。但你只要明白了上面的核心原理，就已经成功了一半了，剩下的就是写代码了，这有何难呢？

下面我们来具体介绍一下上述增删查改过程中几个关键的概念和可能出现的问题。

## 几个关键概念及原理

**key** 是唯一的，**value** 可以重复

哈希表中，不可能出现两个相同的 **key**，而 **value** 是可以重复的。

明白了上面讲的原理应该很好理解，你直接类比数组就行了：

数组里面每个索引都是唯一的，不可能说你这个数组有两个索引 0。至于数组里面存什么元素，随便你，没人 care。

所以哈希表是一样的，**key** 的值不可能出现重复，而 **value** 的值可以随意。

## 哈希函数

哈希函数的作用是把任意长度的输入（key）转化成固定长度的输出（索引）。

你也看到了，增删查改的方法中都会用到哈希函数来计算索引，如果你设计的这个哈希函数复杂度是  $O(N)$ ，那么哈希表的增删查改性能就会退化成  $O(N)$ ，所以说这个函数的性能很关键。

这个函数还要保证的一点是，输入相同的 **key**，输出也必须要相同，这样才能保证哈希表的正确性。不能说现在你计算  $\text{hash("123")} = 5$ ，待会儿计算  $\text{hash("123")} = 6$ ，这样的话哈希表就废了。

那么哈希函数是如何把非整数类型的 **key** 转化成整数索引的？又是如何保证这个索引是合法的呢？

这个问题可以有很多种答案，不同的哈希函数设计会有不同的方法，我这里就结合 Java 语言说一个简单的办法。其他编程语言也是类似的，可以参考这个思路，查询相关的标准库文档。

任意 Java 对象都会有一个 `int hashCode()` 方法，在实现自定义的类时，如果不重写这个方法，那么它的默认返回值可以认为是该对象的内存地址。一个对象的内存地址显然是全局唯一的一个整数。

所以我们只要调用 `key` 的 `hashCode()` 方法就相当于把 `key` 转化成了一个整数，且这个整数是全局唯一的。

当然，这个方法也有一些问题，下面会讲解，但现在至少找到了一种把任意对象转化为整数的方法。

`hashCode` 方法返回的是 `int` 类型，首先一个问题就是，这个 `int` 值可能是负数，而数组的索引是非负整数。

那么你肯定想这样写代码，把这个值转化成非负数：

```
int h = key.hashCode();
if (h < 0) h = -h;
```

但这样有问题，`int` 类型可以表示的最小值是  $-2^{31}$ ，而最大值是  $2^{31} - 1$ 。所以如果 `h = -2^{31}`，那么 `-h = 2^{31}` 就会超出 `int` 类型的最大值，这叫做整型溢出，编译器会报错，甚至产生不可预知的结果。

为什么 `int` 的最小值是  $-2^{31}$ ，而最大值是  $2^{31} - 1$ ? 这涉及计算机补码编码的原理，简单说，`int` 就是 32 个二进制位，其中最高位（最左边那位）是符号位，符号位是 0 时表示正数，是 1 时表示负数。

现在的问题是，我想保证 `h` 非负，但又不能用负号直接取反。那么一个简单直接的办法是利用这种补码编码的原理，直接把最高位的符号位变成 0，就可以保证 `h` 是非负数了：

```
int h = key.hashCode();
// 位运算，把最高位的符号位去掉
// 另外，位运算的运行速度也会比一般的算术运算快
// 所以你看标准库的源码，能用位运算的地方它都会优先使用位运算
h = h & 0x7fffffff;
// 这个 0x7fffffff 的二进制表示是 0111 1111 ... 1111
// 即除了最高位（符号位）是 0，其他位都是 1
// 把 0x7fffffff 和其他 int 进行 & 运算之后，最高位（符号位）就会被清零，即保证了 h 是非
// 负数
```

关于补码编码的原理我这里就不详细展开了，有兴趣的话你可以自己搜索学习一下。

好的，上面解决了 `hashCode` 可能是负数的问题，但还有一个问题，就是这个 `hashCode` 一般都很大，我们需要把它映射成 `table` 数组的合法索引。

这个问题对你来说应该不难吧，我们之前在 [环形数组原理及实现](#) 里面用 `%` 求模运算来保证索引永远落在数组的合法范围内。所以这里也可以用 `%` 运算来保证索引的合法性，完整的 `hash` 函数实现如下：

```
int hash(K key) {
    int h = key.hashCode();
    // 保证非负数
    h = h & 0x7fffffff;
    // 映射到 table 数组的合法索引
```

```

        return h % table.length;
    }
}

```

当然，直接使用 `%` 也有问题，因为 `%` 这个求余数的运算比较消耗性能，一般在追求运行效率的标准库源码中会尽量避免使用 `%` 运算，而是使用位运算提升性能。

不过本章主要目的是带你理解实现一个简单的哈希表，就忽略这些细节优化了。有兴趣的话你可以去看一下 Java `HashMap` 的源码，看看它是如何实现这个 `hash` 函数的。

## 哈希冲突

上面给出了 `hash` 函数的实现，那么你肯定也会想到，如果两个不同的 `key` 通过哈希函数得到了相同的索引，怎么办呢？这种情况就叫做「哈希冲突」。

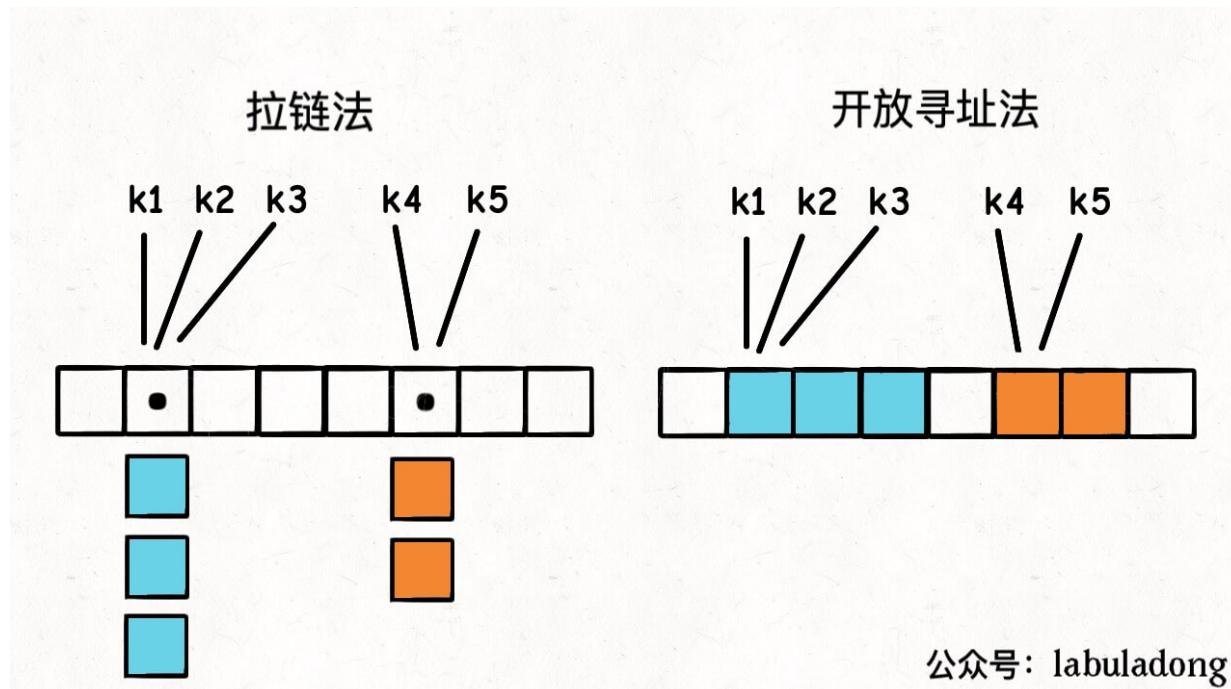
哈希冲突不可能避免，只能在算法层面妥善处理出现哈希冲突的情况。

哈希冲突是一定会出现的，因为这个 `hash` 函数相当于是把一个无穷大的空间映射到了一个有限的索引空间，所以必然会有不同的 `key` 映射到同一个索引上。

就好比三维物体映射到二维影子一样，这种有损压缩必然会出现信息丢失，有损信息本就无法和原信息一一对应。

出现哈希冲突的情况怎么解决？两种常见的解决方法，一种是 **拉链法**，另一种是 **线性探查法**（也经常被叫做**开放寻址法**）。

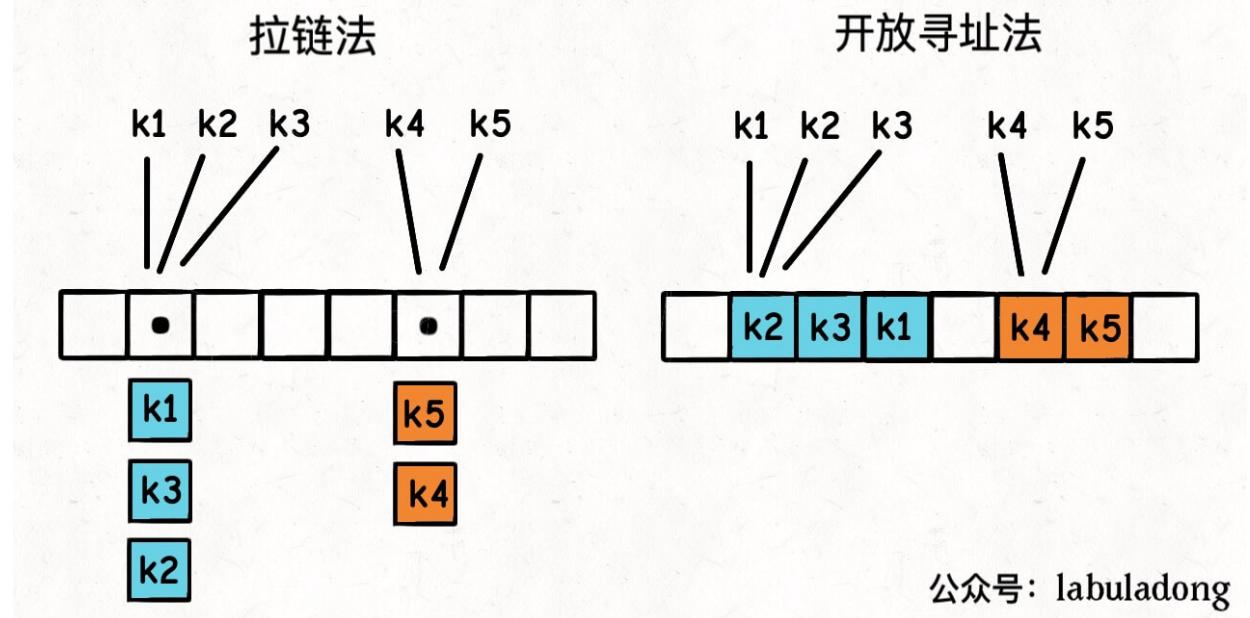
名字听起来高大上，说白了就是纵向延伸和横向延伸两种思路嘛：



拉链法相当于是哈希表的底层数组并不直接存储 `value` 类型，而是存储一个链表，当有多个不同的 `key` 映射到了同一个索引上，这些 `key -> value` 对儿就存储在这个链表中，这样就能解决哈希冲突的问题。

而线性探查法的思路是，一个 `key` 发现算出来的 `index` 值已经被别的 `key` 占了，那么它就去 `index + 1` 的位置看看，如果还是被占了，就继续往后找，直到找到一个空的位置为止。

比方说上图，`key` 的插入顺序是 `k2, k4, k5, k3, k1`，那么哈希表底层就会变成这样：



这里先讲一下原理，后面的章节我会手把手带大家分别实现这两种方法来解决哈希冲突。

## 扩容和负载因子

相信大家都听说过「负载因子」这个专业术语，现在你明白了哈希冲突的问题，就能理解负载因子的意义了。

拉链法和线性探查法虽然能解决哈希冲突的问题，但是它们会导致性能下降。

比如拉链法，你算出来 `index = hash(key)` 这个索引了，结果过去查出来的是个链表，你还得遍历一下这个链表，才能在里面找到你要的 `value`。这个过程的时间复杂度是  $O(K)$ ， $K$  是这个链表的长度。

线性探查法也是类似的，你算出来 `index = hash(key)` 这个索引了，你去这个索引位置查看，发现存储的不是要找的 `key`，但由于线性探查法解决哈希冲突的方式，你并不能确定这个 `key` 真的不存在，你必须顺着这个索引往后找，直到找到一个空的位置或者找到这个 `key` 为止，这个过程的时间复杂度也是  $O(K)$ ， $K$  为连续探查的次数。

所以说，如果频繁出现哈希冲突，那么  $K$  的值就会增大，这个哈希表的性能就会显著下降。这是我们需要注意避免的。

那么为什么会频繁出现哈希冲突呢？两个原因呗：

1、哈希函数设计的不好，导致 `key` 的哈希值分布不均匀，很多 `key` 映射到了同一个索引上。

2、哈希表里面已经装了太多的 `key-value` 对了，这种情况下即使哈希函数再完美，也没办法避免哈希冲突。

对于第一个问题没什么好说的，开发编程语言标准库的大佬们已经帮你设计好了哈希函数，你只要调用就行了。

对于第二个问题是我们可以控制的，即避免哈希表装太满，这就引出了「负载因子」的概念。

负载因子是一个哈希表装满的程度的度量。一般来说，负载因子越大，说明哈希表里面存储的 `key-value` 对越多，哈希冲突的概率就越大，哈希表的操作性能就越差。

负载因子的计算公式也很简单，就是 `size / table.length`。其中 `size` 是哈希表里面的 `key-value` 对的数量，`table.length` 是哈希表底层数组的容量。

你不难发现，用拉链法实现的哈希表，负载因子可以无限大，因为链表可以无限延伸；用线性探查法实现的哈希表，负载因子不会超过 1。

像 Java 的 HashMap，允许我们创建哈希表时自定义负载因子，不设置的话默认是 `0.75`，这个值是经验值，一般保持默认就行了。

当哈希表内元素达到负载因子时，哈希表会扩容。和之前讲解 [动态数组的实现](#) 是类似的，就是把哈希表底层 `table` 数组的容量扩大，把数据移到新的大数组中。`size` 不变，`table.length` 增加，负载因子就减小了。

## 为什么不能依赖哈希表的遍历顺序

你大概也听过一个编程常识，即哈希表中键的遍历顺序是无序的，不能依赖哈希表的遍历顺序来编写程序。这是为什么呢？

哈希表的遍历本质上就是遍历那个底层 `table` 数组：

```
// 遍历所有 key 的伪码逻辑

// 哈希表底层的 table 数组
KVNode[] table = new KVNode[1000];

// 获取哈希表中的所有键
// 我们不能依赖这个 keys 列表的顺序
List<KeyType> keys = new ArrayList<>();

for (int i = 0; i < table.length; i++) {
    KVNode node = table[i];
    if (node != null) {
        keys.add(node.key);
    }
}
```

你如果理解了前面讲的内容，应该已经能够理解这个问题了。

首先，由于 `hash` 函数要把你的 `key` 进行映射，所以 `key` 在底层 `table` 数组中的分布是随机的，不像数组/链表结构那样有个明确的元素顺序。

其次，刚才讲了哈希表达到负载因子时会怎样？会扩容对吧，也就是 `table.length` 会变化，且会搬移元素。

那么这个搬移数据的过程，是不是要用 `hash` 函数重新计算 `key` 的哈希值，然后放到新的 `table` 数组中？

而这个 `hash` 函数，它计算出的值依赖 `table.length`。也就是说，哈希表自动扩容后，同一个 `key` 的哈希值可能变化，即这个 `key-value` 对儿存储在 `table` 的索引也变了，所以遍历结果的顺序就和之前不一样了。

你观察到的现象就是，这次遍历的第一个键是 `key1`，但是增删几个元素再遍历，可能发现 `key1` 跑到最后去了。

所以说，这些东西没必要背的，原理搞明白了，你稍微推理下自己都能想通。

## 为什么不建议在 for 循环中增/删哈希表的 key

注意我这里说的是不建议，并不是一定不可以。因为不同的编程语言标准库对哈希表的实现不同，有些语言针对这种情况做了优化，所以到底行不行，要查阅文档。

我们这里仅从哈希表的原理上分析，在 for 循环中增/删哈希表的 `key`，是很容易出现问题的，原因和上面相同，还是扩容导致的哈希值变化。

遍历哈希表的 `key`，本质就是遍历哈希表底层的 `table` 数组，如果一边遍历一边增删元素，如果遍历到一半，插入/删除操作触发了扩容，整个 `table` 数组都变了，那么请问，接下来应该是什么行为？还有，在遍历过程中新插入/删除的元

素，是否应该被遍历到？

扩缩容导致 **key** 顺序变化是哈希表的特有行为，但即便排除这个因素，任何其他数据结构，也都不建议在遍历的过程中同时进行增删，否则很容易导致非预期的行为。

如果你非要这样做，请确保查阅了相关文档，明确这个操作的行为是什么，做到心里有数。

## **key** 必须是不可变的

只有那些不可变类型，才能作为哈希表的 **key**，这一点很重要。

所谓不可变类型，就是说这个对象一旦创建，它的值就不能再改变了。比如 Java 中的 **String**, **Integer** 等类型，一旦创建了这些对象，你就只能读取它的值，而不能再修改它的值了。

作为对比，Java 中的 **ArrayList**、**LinkedList** 这些对象，它们创建出来之后，可以往里面随意增删元素，所以它们是可变类型。

因此，你可以把 **String** 对象作为哈希表的 **key**，但不能把 **ArrayList** 对象作为哈希表的 **key**：

```
// 可以把不可变类型作为 key
Map<String, AnyOtherType> map1 = new HashMap<>();
Map<Integer, AnyOtherType> map2 = new HashMap<>();

// 不应该把可变类型作为 key
// 注意，这样写并不会产生语法错误，但是代码非常容易出 bug
Map<ArrayList<Integer>, AnyOtherType> map3 = new HashMap<>();
```

为啥不建议把可变类型作为 **key** 呢？就比如这个 **ArrayList** 吧，它的 **hashCode** 方法的实现逻辑如下：

```
public int hashCode() {
    for (int i = 0; i < elementData.length; i++) {
        h = 31 * h + elementData[i];
    }
}
```

第一个就是效率问题，每次计算 **hashCode** 都要遍历整个数组，复杂度是  $O(N)$ ，这样就会导致哈希表的增删查改操作的复杂度退化成  $O(N)$ 。

更严重的问题是，**ArrayList** 的 **hashCode** 是根据它里面的元素计算出来的，如果你往这个 **ArrayList** 里面增删元素，或者其中某个元素的 **hashCode** 值发生改变，那么这个 **ArrayList** 的 **hashCode** 返回值也会发生改变。

比方说，你现在用一个 **ArrayList** 类型的 **arr** 变量作为哈希表的 **key** 在哈希表中保存了对应的 **value**。但如果 **arr** 中的某个元素在程序的其他位置被修改了，那么 **arr** 的 **hashCode** 就会变化。此时你再用这个 **arr** 变量去哈希表中查询，发现找不到任何值了。

也就是说，你存入哈希表的 **key-value** 意外丢失了，这是非常非常严重的 **bug**，还会带来潜在的内存泄漏问题。

```
public class Test {
    public static void main(String[] args) {
        // 错误示例
        // 把可变类型作为 HashMap 的 key
        Map<ArrayList<Integer>, Integer> map = new HashMap<>();
```

```
ArrayList<Integer> arr = new ArrayList<>();
arr.add(1);
arr.add(2);

map.put(arr, 999);
System.out.println(map.containsKey(arr)); // true
System.out.println(map.get(arr)); // 999

arr.add(3);
// 出现严重 bug, 键值对丢失
System.out.println(map.containsKey(arr)); // false
System.out.println(map.get(arr)); // null

// 此时 map 底层的 table 中, arr 的键值对数据依然存在
// 但是由于 arr 的 hashCode 改变了, 此键值对无法被查找到
// 这也会导致内存泄漏, 因为这个 arr 变量被 map 引用着, 无法被垃圾回收
}

}
```

上面就是一个简单的错误示例。你也许会说，把元素 3 删掉，`arr -> 999` 这个键值对不就又出现了？或者，直接遍历哈希表底层的 `table` 数组，应该也可以看到这个键值对。

拜托🙏，你这是在写代码还是在写盗墓笔记呢？一会儿出现一会儿消失，你这个哈希表是幽灵附体了吗？

开个玩笑。实际上可变类型本身就是一种不确定性，在代码构成的屎山里，你怎么知道这个 `arr` 传递到哪里被修改了呢？

所以正确的做法是，使用不可变类型作为哈希表的 `key`，比方说用 `String` 类型作为 `key`。因为 Java 中的 `String` 对象一旦创建出来，它的值就不允许被改变，你就不会遇到上面的问题。

`String` 类型的 `hashCode` 方法也需要遍历所有字符，但是由于它的不可变性，这个值只要算出来一次，就可以缓存下来，不用每次都重新计算，所以 平均时间复杂度 依然是  $O(1)$ 。

我这里是用 Java 举的例子，其他语言也是类似的，你需要查询相关文档，了解标准库提供的哈希表是如何计算对象哈希值的，避免产生类似的问题。

## 总结

上面的说明应该已经吧哈希表的底层原理全部串起来了，最后模拟几个面试问题来总结一下本文的内容：

### 1、为什么我们常说，哈希表的增删查改效率都是 $O(1)$ ？

因为哈希表底层就是操作一个数组，其主要的时间复杂度来自于哈希函数计算索引和哈希冲突。只要保证哈希函数的复杂度在  $O(1)$ ，且合理解决哈希冲突的问题，那么增删查改的复杂度就都是  $O(1)$ 。

### 2、哈希表的遍历顺序为什么会变化？

因为哈希表在达到负载因子时会扩容，这个扩容过程会导致哈希表底层的数组容量变化，哈希函数计算出来的索引也会变化，所以哈希表的遍历顺序也会变化。

### 3、哈希表的增删查改效率一定是 $O(1)$ 吗？

不一定，正如前面分析的，只有哈希函数的复杂度是  $O(1)$ ，且合理解决哈希冲突的问题，才能保证增删查改的复杂度是  $O(1)$ 。

哈希冲突好解决，都是有标准答案的。关键是哈希函数的计算复杂度。如果使用了错误的 `key` 类型，比如前面用 `ArrayList` 作为 `key` 的例子，那么哈希表的复杂度就会退化成  $O(N)$ 。

### 4、为啥一定要用不可变类型作为哈希表的 `key`？

因为哈希表的主要操作都依赖于哈希函数计算出来的索引，如果 `key` 的哈希值会变化，会导致键值对意外丢失，产生严重的 bug。

要对自己使用的编程语言标准库中的源码有一定的了解，才能保证写出高效的代码。

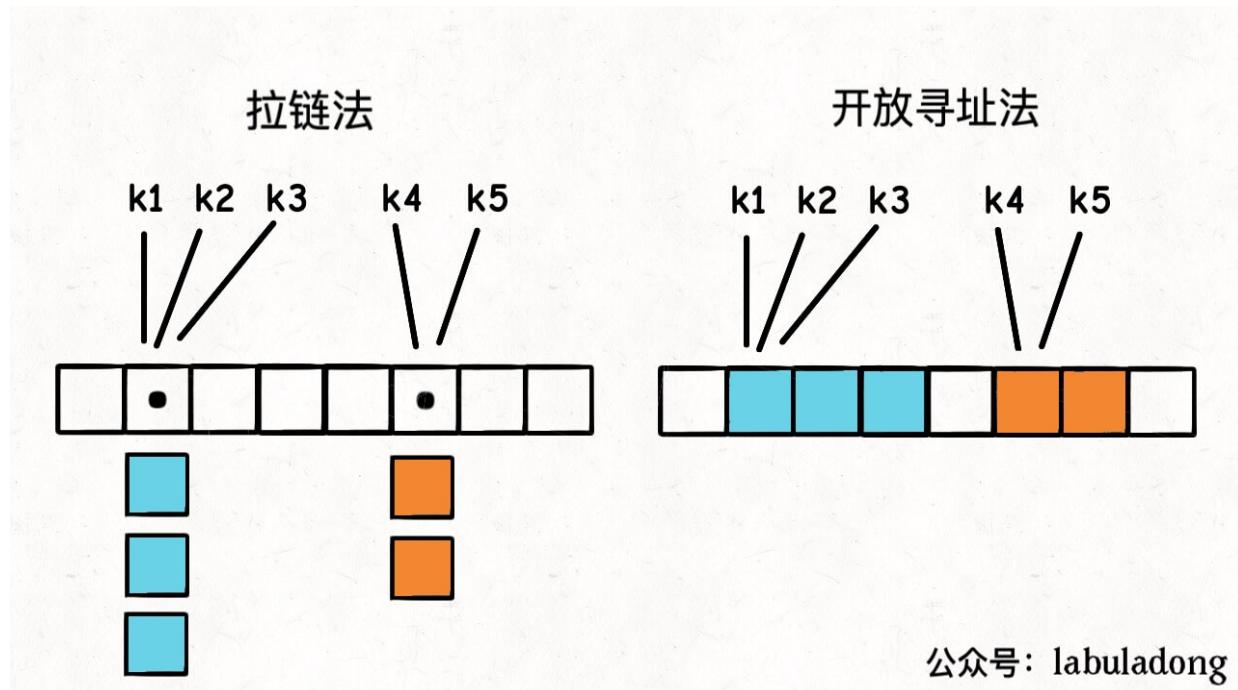
下面，就我将手把手带大家分别用拉链法和线性探查法来实现简单的哈希表，来加深对哈希表的理解。

# 用拉链法实现哈希表

阅读本文前，你需要先学习：

- 哈希表核心原理
- 链表（链式存储）基础

前文 [哈希表核心原理](#) 中我介绍了哈希表的核心原理和几个关键概念，其中提到了解决哈希冲突的方法主要有两种，分别是拉链法和开放寻址法（也常叫做线性探查法）：



本文就来具体介绍一下拉链法的实现原理和代码。

首先，我会结合 [可视化面板](#) 用拉链法实现一个简化版的哈希表，带大家直观地理解拉链法是如何实现增删查改的 API 并解决哈希冲突的，最后再给出一个比较完善的 Java 代码实现。

## 拉链法的简化版实现

[哈希表核心原理](#) 已经介绍过哈希函数和 `key` 的类型的关系，其中 `hash` 函数的作用是在  $O(1)$  的时间把 `key` 转化成数组的索引，而 `key` 可以是任意不可变的类型。

但是这里为了方便诸位理解，我先做如下简化：

- 1、我们实现的哈希表只支持 `key` 类型为 `int`, `value` 类型为 `int` 的情况，如果 `key` 不存在，就返回 `-1`。
- 2、我们实现的 `hash` 函数就是简单地取模，即 `hash(key) = key % table.length`。这样也方便模拟出哈希冲突的情况，比如当 `table.length = 10` 时，`hash(1)` 和 `hash(11)` 的值都是 1。
- 3、底层的 `table` 数组的大小在创建哈希表时就固定，不考虑负载因子和动态扩缩容的问题。

这些简化能够帮助我们聚焦增删查改的核心逻辑，并且可以借助 [可视化面板](#) 辅助大家学习理解。

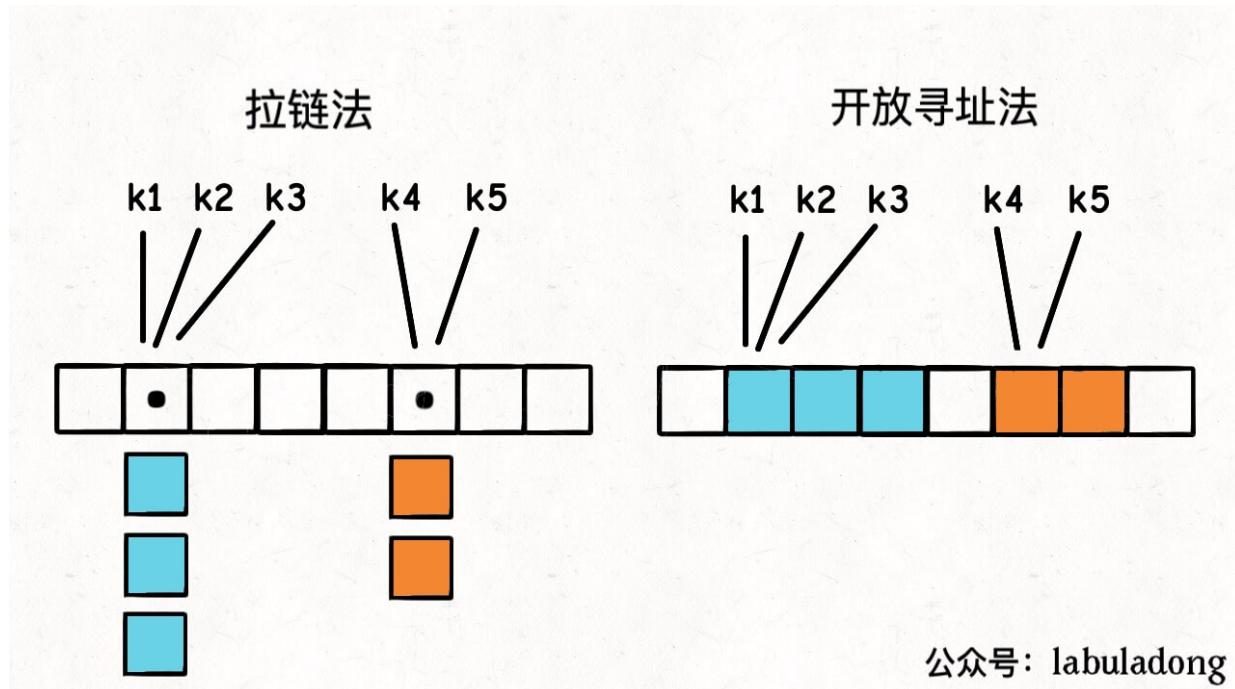
本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 线性探查法的两个难点

阅读本文前，你需要先学习：

- 哈希表核心原理

前文 [哈希表核心原理](#) 中我介绍了哈希表的核心原理和几个关键概念，其中提到了解决哈希冲突的方法主要有两种，分别是拉链法和线性探查法（也常叫做开放寻址法）：



由于线性探查法稍微复杂一些，本文先讲解实现线性探查法的几个难点，下篇文章再给出具体的代码实现。

## 简化场景

之前介绍的拉链法应该是比较简单的，无非就是 `table` 中每个元素都是一个链表，出现哈希冲突的话往链表里塞元素就行了。

而线性探查法会更复杂，主要有两个难点，涉及到多种数组操作技巧。在讲清楚这两个难点之前，我们先设定一个简化的场景：

假设我们的哈希表只支持 `key` 类型为 `int`, `value` 类型为 `int` 的情况，且 `table.length` 固定为 `10`, `hash` 函数的实现是 `hash(key) = key % 10`。因为这样比较容易模拟出哈希冲突，比如 `hash(1)` 和 `hash(11)` 的值都是 `1`。

线性探查法的大致逻辑如下：

```
// 线性探查法的基本逻辑，伪码实现

class MyLinearProbingHashMap {
    // 数组中每个元素都存储一个键值对
    private KVNode[] table = new KVNode[10];

    private int hash(int key) {
        return key % table.length;
```

```
}

public void put(int key, int value) {
    int index = hash(key);
    KVNode node = table[index];
    if (node == null) {
        table[index] = new KVNode(key, value);
    } else {
        // 线性探查法的逻辑
        // 向后探查, 直到找到 key 或者找到空位
        while (index < table.length && table[index] != null && table[index].key
!= key) {
            index++;
        }
        table[index] = new KVNode(key, value);
    }
}

public int get(int key) {
    int index = hash(key);
    // 向后探查, 直到找到 key 或者找到空位
    while (index < table.length && table[index] != null && table[index].key !=
key) {
        index++;
    }
    if (table[index] == null) {
        return -1;
    }
    return table[index].value;
}

public void remove(int key) {
    int index = hash(key);
    // 向后探查, 直到找到 key 或者找到空位
    while (index < table.length && table[index] != null && table[index].key !=
key) {
        index++;
    }
    // 删除 table[index]
    // ...
}
```

基于这个假设场景，我们来看看线性探查法的两个难点。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 线性探查法的两种代码实现

阅读本文前，你需要先学习：

- 线性探查法的两个难点

前文 [哈希表核心原理](#) 中我介绍了哈希表的核心原理和几个关键概念，[拉链法原理和实现](#) 中介绍了拉链法的实现，[线性探查法的两个难点](#) 介绍了线性探查法实现哈希表的难点所在，并给出了两种方法解决删除元素时的空洞问题，本文会同时给出这两种方法的参考代码实现。

本文会先结合可视化面板给出简化的实现，方便大家理解增删查改的过程，最后给完整实现。

简化实现中，具体简化的地方如下：

- 1、我们实现的哈希表只支持 `key` 类型为 `int`, `value` 类型为 `int` 的情况，如果 `key` 不存在，就返回 `-1`。
- 2、我们实现的 `hash` 函数就是简单地取模，即 `hash(key) = key % table.length`。这样也方便模拟出哈希冲突的情况，比如当 `table.length = 10` 时，`hash(1)` 和 `hash(11)` 的值都是 1。
- 3、底层的 `table` 数组的大小在创建哈希表时就固定，假设 `table` 数组不会被装满，不考虑负载因子和动态扩缩容的问题。

这些简化能够帮助我们聚焦增删查改的核心逻辑，并且可以借助 [可视化面板](#) 辅助大家学习理解。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 哈希集合的原理及代码实现

阅读本文前，你需要先学习：

- 哈希表核心原理

我讲解前面每种数据结构时，都会把原理和代码实现分到两篇文章里讲解，而这里讲哈希集合时，把原理和实现同时放在本文讲解，且本章节只有本文一篇文章，你有没有觉得奇怪？

哈哈，因为哈希集合没什么好讲的，它就是把前文讲的哈希表简单封装了一下：**哈希表的键，其实就是哈希集合。**

这么一句话就可以讲完了，不过我们还是稍微具体讲一下，照顾一下哈希集合的面子。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 用链表加强哈希表（LinkedHashMap）

阅读本文前，你需要先学习：

- 哈希表核心原理

前文 [哈希表原理](#) 从原理上分析了，不能依赖哈希表遍历 `key` 的顺序，即哈希表中的 `key` 是无序的。

但结合实际的编程经验，你可能会有些疑问。

比如熟悉 Python 的读者可能知道，Python 3.7 开始，标准库提供的哈希表 `dict` 就明确告诉你了，`dict` 的键的遍历顺序就是键的插入顺序。比如下面这段简单的代码：

```
d = dict()

d['a'] = 1
d['b'] = 2
d['c'] = 3
print(list(d.keys())) # ['a', 'b', 'c']

d['y'] = 4
print(list(d.keys())) # ['a', 'b', 'c', 'y']

d['d'] = 5
print(list(d.keys())) # ['a', 'b', 'c', 'y', 'd']
```

无论你插入多少键，`keys` 方法返回的所有键都是按照插入顺序排列，感觉就好像在向数组尾部追加元素一样。这怎么可能呢？

如果你熟悉 Golang，你会发现一个更神奇的现象。比如下面这段测试代码：

```
package main

import (
    "fmt"
)

func main() {
    // 初始化 map
    myMap := map[string]int{
        "1": 1,
        "2": 2,
        "3": 3,
        "4": 4,
        "5": 5,
    }

    // 定义遍历 map 的函数
    printMapKeys := func(m map[string]int) {
        for key := range m {
            fmt.Println(key, " ")
        }
    }
}
```

```
    fmt.Println()
}

// 多次遍历 map，观察键的顺序
printMapKeys(myMap)
printMapKeys(myMap)
printMapKeys(myMap)
printMapKeys(myMap)
}

// 我运行的结果如下：
// 1 2 3 4 5
// 5 1 2 3 4
// 2 3 4 5 1
// 1 2 3 4 5
```

也就是说，它每次遍历的顺序都是随机。但是按照前文 [哈希表原理](#) 所说，虽然哈希表的键是无序的，但是没有对哈希表做任何操作，遍历得到的结果应该不会变才对，Golang 的 map 每次遍历的顺序咋都不一样？这也太离谱了吧？

你可以先自己思考下原因，下面我给出答案。

先说 Golang 吧，每次遍历都乱序的原因就是，它故意的。

这个原因属实是让人有些哭笑不得，Golang 为了防止开发者依赖哈希表的遍历顺序，所以每次遍历都故意返回不同的顺序，可谓用心良苦。也可以从侧面看出，确实不少开发者没了解过哈希表的基本原理。

我们不妨进一步想想，它是怎么打乱顺序的呢？真是随机打乱吗？

其实不是，你仔细看看，它这个乱序是有规律的。有没有想起前面讲过的 [环形数组](#)？

	1	2	3	4	5
5		1	2	3	4
2	3	4	5		1
	1	2	3	4	5

看出来没有？如果不触发扩容的话，实际上它的遍历顺序应该也是固定的，只不过它不是每次都从底层 [table](#) 数组的头部开始，而是从一个随机的位置开始，然后利用环形数组技巧遍历整个 [table](#) 数组，这样就能保证多次遍历的顺序具有随机性。

再说 Python，它能让所有键按照插入顺序排列，是因为它把标准的哈希表和链表结合起来，组成了一种新的数据结构：[哈希链表](#)。

其他编程语言也有类似的实现，比如 Java 的 [LinkedHashMap](#)。这种数据结构兼具了哈希表 \$O(1)\$ 的增删查改效率，同时又可以像数组链表一样保持键的插入顺序。

它是怎么做的呢？下面我会具体讲解。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 用数组加强哈希表（ArrayHashMap）

阅读本文前，你需要先学习：

- 哈希表核心原理

上一章 [用链表加强哈希表](#) 我们利用 [双链表](#) 对哈希表进行了加强，实现了 [LinkedHashMap](#) 这种数据结构，让哈希表的键保持插入顺序。

链表能加强哈希表，数组作为链表的好兄弟，其实也能加强哈希表。

## 添加 [randomKey\(\)](#) API

现在我给你出个题，让你基于标准哈希表的 API 之上，再添加一个新的 [randomKey\(\)](#) API，可以在  $O(1)$  的时间复杂度返回一个随机键：

```
interface Map<K, V> {
    // 获取 key 对应的 value, 时间复杂度 O(1)
    V get(K key);

    // 添加/修改 key-value 对, 时间复杂度 O(1)
    void put(K key, V value);

    // 删除 key-value 对, 时间复杂度 O(1)
    void remove(K key);

    // 是否包含 key, 时间复杂度 O(1)
    boolean containsKey(K key);

    // 返回所有 key, 时间复杂度 O(N)
    List<K> keys();

    // 新增 API: 随机返回一个 key, 要求时间复杂度 O(1)
    K randomKey();
}
```

注意，我们一般说的随机，都是指均匀随机，即每个元素被选中的概率相等。比如你有  $n$  个元素，你的随机算法要保证每个元素被选中的概率都是  $1/n$ ，才叫均匀随机。

怎么样，你会不会做？不要小看这个简单的需求，实现方法其实是比较巧妙的。

通过前面的学习，你应该知道哈希表的本质就是一个 [table](#) 数组，现在让你随机返回一个哈希表的键，很容易就会联想到在数组中随机获取一个元素。

在标准数组，随机获取一个元素很简单，只要用随机数生成器生成一个  $[0, size]$  的随机索引，就相当于找了一个随机元素：

```
int randomElement(int[] arr) {
    Random r = new Random();
```

```
// 生成 [0, arr.length) 的随机索引
return arr[r.nextInt(arr.length)];
}
```

这个算法是正确的，它的复杂度是  $O(1)$ ，且每个元素被选中的概率都是  $1/n$ ， $n$  为  $arr$  数组的总元素个数。

但你注意，上面这个函数有个前提，就是数组中的元素是紧凑存储没有空洞的，比如  $arr = [1, 2, 3, 4]$ ，这样才能保证任意一个随机索引都对应一个有效的元素。

如果数组中有空洞就有问题了，比如  $arr = [1, 2, null, 4]$ ，其中  $arr[2] = null$  代表没有存储元素的空洞，那么如果你生成的随机数恰好是 2，请问你该怎么办？

也许你想说，可以向左或者向右线性查找，找到一个非空的元素返回，类似这样：

```
// 返回一个非空的随机元素（伪码）
int randomElement(int[] arr) {
    Random r = new Random();
    // 生成 [0, arr.length) 的随机索引
    int i = r.nextInt(arr.length);
    while (arr[i] == null) {
        // 随机生成的索引 i 恰巧是空洞
        // 借助环形数组技巧向右进行探查
        // 直到找到一个非空元素
        i = (i + 1) % arr.length;
    }
    return arr[i];
}
```

你这样是不行的，这个算法有两个问题：

1、有个循环，最坏时间复杂度上升到了  $O(N)$ ，不符合  $O(1)$  的要求。

2、这个算法不是均匀随机的，因为你的查找方向是固定的，空洞右侧的元素被选中的概率会更大。比如  $arr = [1, 2, null, 4]$ ，元素  $1, 2, 4$  被选中的概率分别是  $1/4, 1/4, 2/4$ 。

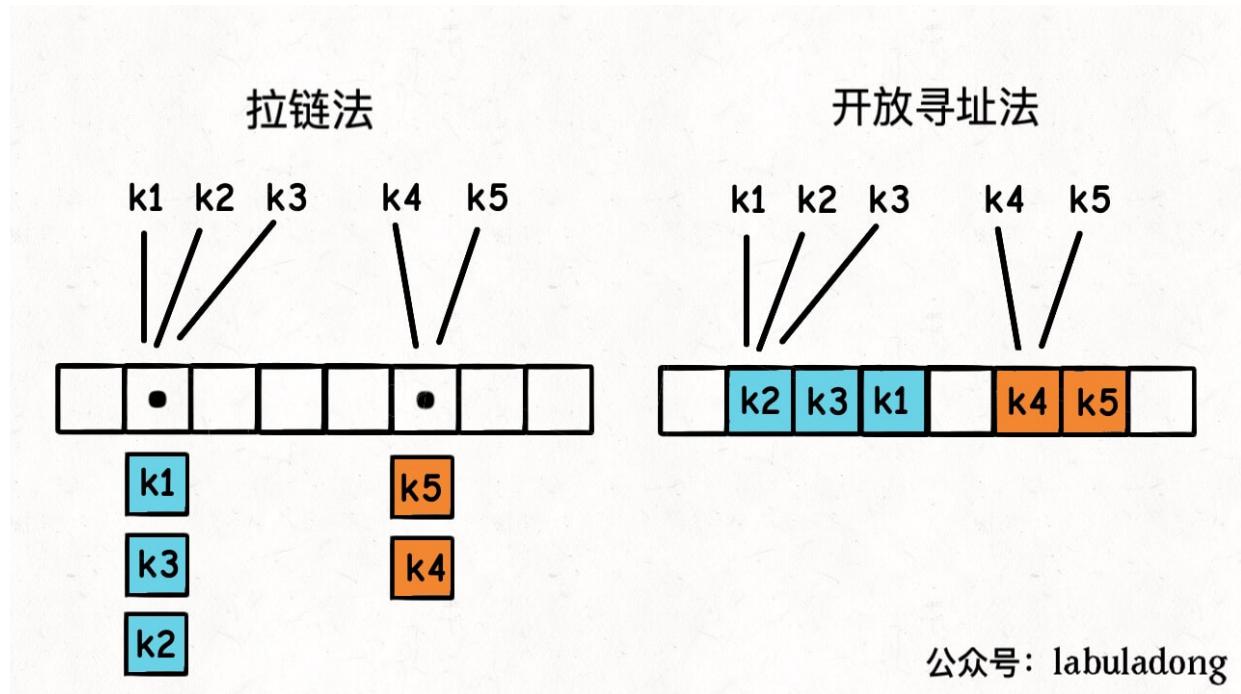
那也许还有个办法，一次运气不好，就多来随机几次，直到找到一个非空元素：

```
// 返回一个非空的随机元素（伪码）
int randomElement(int[] arr) {
    Random r = new Random();
    // 生成 [0, arr.length) 的随机索引
    int i = r.nextInt(arr.length);
    while (arr[i] == null) {
        // 随机生成的索引 i 恰巧是空洞
        // 重新生成一个随机索引
        i = r.nextInt(arr.length);
    }
    return arr[i];
}
```

现在这个算法是均匀随机的，但问题也非常明显，它的时间复杂度竟然依赖随机数！肯定不是  $O(1)$  的，不符合要求。

怎么样，从一个带有空洞的数组中随机返回一个元素是不是都把你难住了？

别忘了，我们现在的目标是从哈希表中随机返回一个键，哈希表底层的 `table` 数组不仅包含空洞，情况还会更复杂一些：



如果你的哈希表用开放寻址法解决哈希冲突，那还好，就是带空洞数组的场景。

如果你的哈希表用拉链法，那可麻烦了。数组里面的每个元素是一个链表，你光随机一个索引是不够的，还要随机链表中的一个节点。

而且注意概率，这个拉链法，就算你均匀随机到一个数组索引，又均匀随机该索引存储的链表节点，得到的这个键是均匀随机的么？

其实不是，上图中 `k1, k2, k3` 被随机到的概率是  $1/2 * 1/3 = 1/6$ ，而 `k4, k5` 被随机到的概率是  $1/2 * 1/2 = 1/4$ ，这不是均匀随机。

概率算法也是非常有意思的一类问题，无论算法题还是实际业务中都会用到一些经典的随机算法，我会在后文 [谈谈游戏中的随机算法](#) 和 [带权重的随机选择](#) 中详细讲解，这里暂时不需要掌握。

唯一的办法就是通过 `keys` 方法遍历整个 `table` 数组，把所有的键都存储到一个数组中，然后再随机返回一个键。但这样复杂度就是  $O(N)$  了，还是不符合要求。

是不是感觉已经走投无路了？所以说，还是要积累一些经典数据结构设计经验，如果面试笔试的时候遇到类似的问题，你现场想恐怕是很难的。下面我就来介绍一下如何用数组加强哈希表，轻松实现 `randomKey()` API。

本文为 [labuladong.online](http://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

# 二叉树基础及常见类型

阅读本文前，你需要先学习：

- 链表（链式存储）基础

我认为二叉树是最重要的基本数据结构，没有之一。

如果你是初学者，现在这个阶段我很难给你彻底解释清楚得出这个结论的原因，你需要认真学习本站后面的内容才能逐渐理解。我暂且总结两个点：

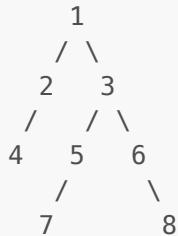
1、二叉树本身是比较简单的基础数据结构，但是很多复杂的数据结构都是基于二叉树的，比如红黑树（二叉搜索树）、[多叉树](#)、[二叉堆](#)、[图](#)、[字典树](#)、[并查集](#)、[线段树](#)等等。你把二叉树玩明白了，这些数据结构都不是问题；如果你不把二叉树搞明白，这些高级数据结构你也很难驾驭。

2、二叉树不单纯是一种数据结构，更代表着递归的思维方式。一切递归算法，比如[回溯算法](#)、[BFS 算法](#)、[动态规划](#)本质上也是把具体问题抽象成树结构，你只要抽象出来了，这些问题最终都回归二叉树的问题。同样看一段算法代码，在别人眼里是一串文本，每个字都认识，但连起来就不认识了；而在你眼里的代码就是一棵树，想咋改就咋改，咋改都能改对，实在是太简单了。

后面的数据结构章节包含大量关于二叉树的讲解和习题，你按照本站的目录顺序学习，我会带你把二叉树彻底搞懂，到时候你就明白我为什么这么重视二叉树了。

## 几种常见的二叉树

二叉树的主要难点在于做算法题，它本身其实没啥难的，就是这样一种树形结构嘛：



上面就是一棵普通的二叉树，几个术语你要了解一下：

1、每个节点下方直接相连的节点称为**子节点**，上方直接相连的节点称为**父节点**。比方说节点 3 的父节点是 1，左子节点是 5，右子节点是 6；节点 5 的父节点是 3，左子节点是 7，没有右子节点。

2、我们称最上方那个没有父节点的节点 1 为**根节点**，称最下层没有子节点的节点 4、7、8 为**叶子节点**。

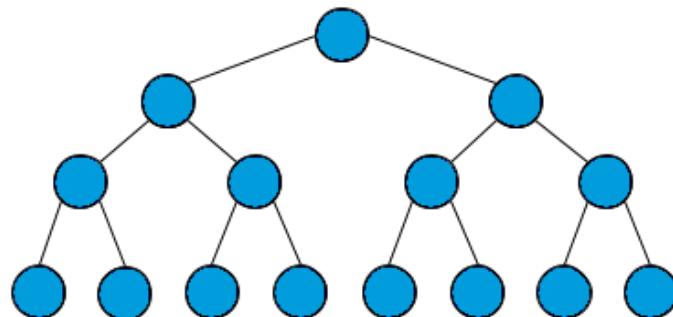
3、我们称从根节点到最下方叶子节点经过的节点个数为二叉树的最大深度/高度，上面这棵树的最大深度是 4，即从根节点 1 到叶子节点 7 或 8 的路径上的节点个数。

没啥别的可说的了，就是这么简单。

有一些稍微特殊一些的二叉树，有他们自己的名字，你要了解一下，后面做题时见到这些专业术语，你就知道题目在说啥了。

## 满二叉树

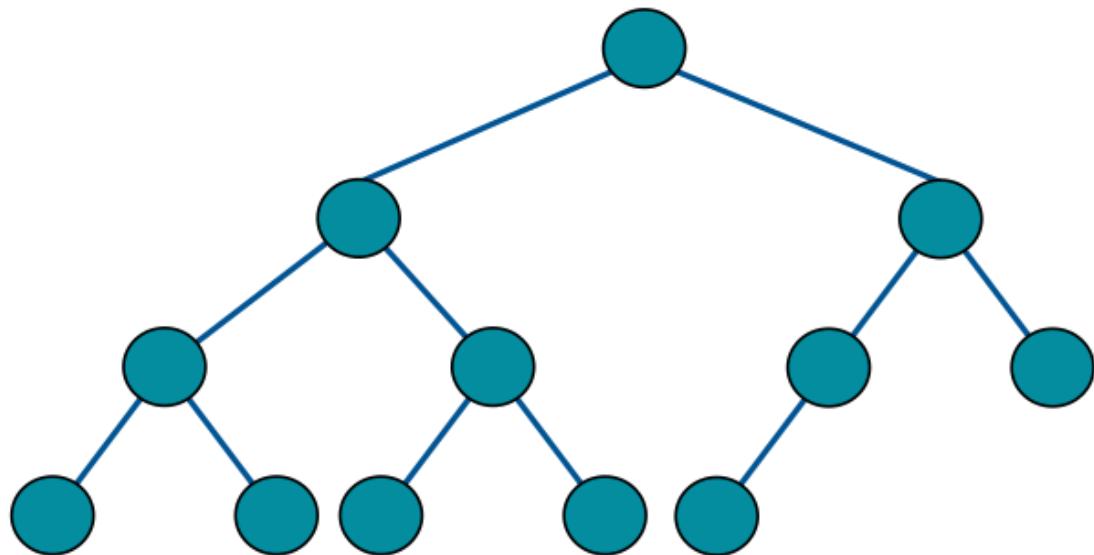
直接看图比较直观，满二叉树就是每一层节点都是满的，整棵树像一个正三角形：



满二叉树有个优势，就是它的节点个数很好算。假设深度为  $h$ ，那么总节点数就是  $2^h - 1$ ，等比数列求和嘛，我们应该都学过的。

## 完全二叉树

完全二叉树是指，二叉树的每一层的节点都紧凑靠左排列，且除了最后一层，其他每层都必须是满的：



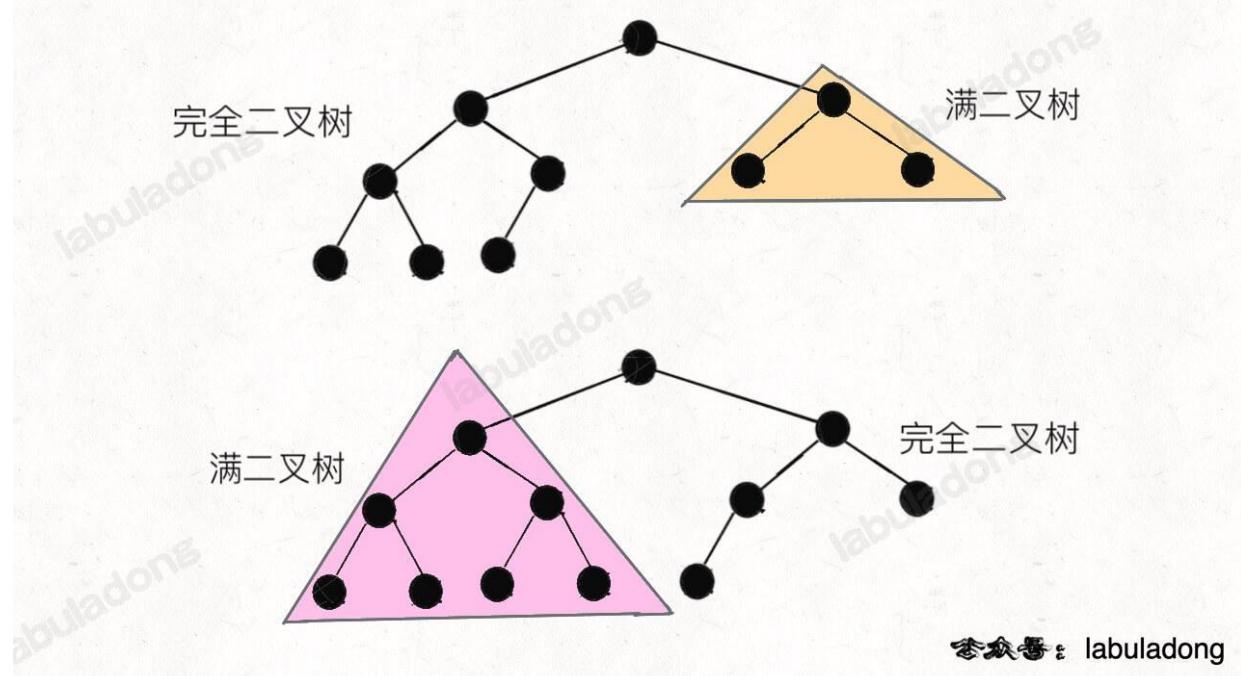
不难发现，满二叉树其实是一种特殊的完全二叉树。

完全二叉树的特点：由于它的节点紧凑排列，如果从左到右从上到下对它的每个节点编号，那么父子节点的索引存在明显的规律。

这个特点在讲到 [二叉堆原理及实现](#) 时会具体讲，做算法题时也会用到。

完全二叉树还有个比较难发觉的性质：完全二叉树的左右子树也是完全二叉树。

或者更准确地说应该是：完全二叉树的左右子树中，至少有一棵是满二叉树。



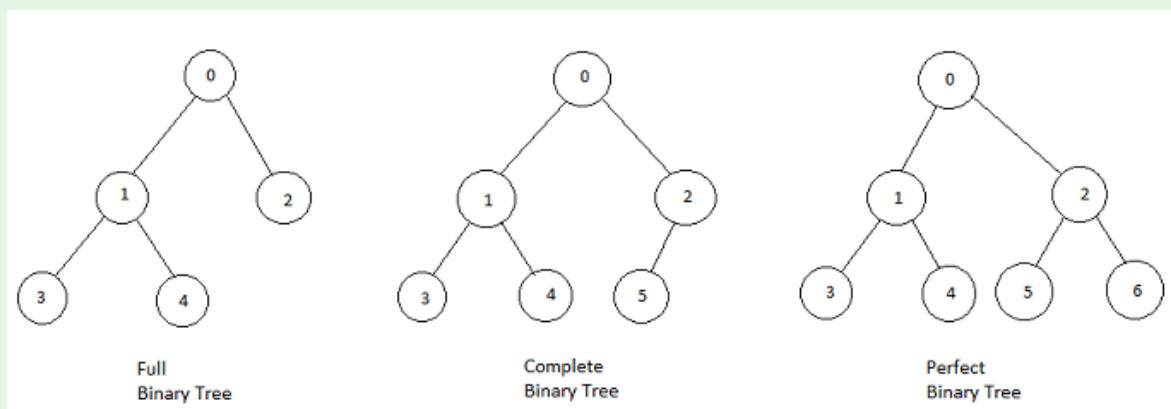
这个性质在做算法题的时候会用到，比如 [巧算完全二叉树的节点数](#)，这里就先提一下。

关于完全二叉树和满二叉树的定义，中文语境和英文语境似乎有点区别。

我们说的完全二叉树对应英文 Complete Binary Tree，这个没问题，说的是同一种树。

我们说的满二叉树，按理说应该翻译成 Full Binary Tree 对吧，但其实不是，满二叉树的定义对应英文的 Perfect Binary Tree。

而英文中的 Full Binary Tree 是指一棵二叉树的所有节点要么没有孩子节点，要么有两个孩子节点。



以上定义出自 wikipedia，这里就是顺便一提。其实名词叫什么都无所谓，你知道有这个区别，在看英文资料时留意一下就行了。

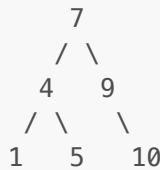
## 二叉搜索树

二叉搜索树 (Binary Search Tree, 简称 BST) 是一种很常见的二叉树，它的定义是：

对于树中的每个节点，其左子树的每个节点的值都要小于这个节点的值，右子树的每个节点的值都要大于这个节点的值。你可以简单记为「左小右大」。

我把「子树的每个节点」加粗了，这是初学者常犯的错误，不要只看子节点，而要看整棵子树的所有节点。

比方说，下面这棵树就是一棵 BST：



节点 7 的左子树所有节点的值都小于 7，右子树所有节点的值都大于 7；节点 4 的左子树所有节点的值都小于 4，右子树所有节点的值都大于 4，以此类推。

相反的，下面这棵树就不是 BST：



如果你只注意每个节点的左右子节点，似乎看不出问题。你应该看整棵子树，注意看节点 7 的左子树中有个节点 8，比 7 大，这就不符合 BST 的定义了。

**BST 是非常常用的数据结构。因为左小右大的特性，可以让我们在 BST 中快速找到某个节点，或者找到某个范围内的所有节点，这是 BST 的优势所在。**

比方说，对于一棵普通的二叉树，其中的节点大小没有任何规律可言，那么你要找到某个值为 x 的节点，只能从根节点开始遍历整棵树。

而对于 BST，你可以先对比根节点和 x 的大小关系，如果 x 比根节点大，那么根节点的整棵左子树就可以直接排除了，直接从右子树开始找，这样就可以快速定位到值为 x 的那个节点。

关于 BST，后面会有专门的章节详细讲解，并且配有大量的习题，这里先讲些基础概念就够你用了。

## 二叉树的实现方式

最常见的二叉树就是类似链表那样的链式存储结构，每个二叉树节点有指向左右子节点的指针，这种方式比较简单直观。

力扣/LeetCode 上给你输入的二叉树一般都是用这种方式构建的，二叉树节点类 `TreeNode` 一般长这样：

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { this.val = x; }
}
```

```
// 你可以这样构建一棵二叉树：
TreeNode root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
root.right.left = new TreeNode(5);
root.right.right = new TreeNode(6);
```

```
// 构建出来的二叉树是这样的：  
//      1  
//      / \\  
//     2   3  
//    /   / \  
//   4   5   6
```

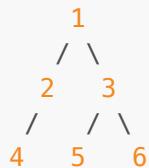
既然说上面是比较常见的实现方式，那言下之意就是还有其他实现方式，对吧？

是的，在[二叉堆原理及实现](#)和[并查集算法详解](#)中，我们会根据具体的需求场景选择用数组来存储二叉树。

在[可视化面板](#)可视化递归函数时，其实是根据函数堆栈生成的递归树，这也算是一种二叉树的实现方式。

另外，在一般的算法题中，我们可能会把实际问题抽象成二叉树结构，但我们并不需要真的用 [TreeNode](#) 创建一棵二叉树出来，而是直接用类似[哈希表](#)的结构来表示二叉树/多叉树。

比方说上面那棵二叉树：



我可以用一个哈希表，其中的键是父节点 id，值是子节点 id 的列表（每个节点的 id 是唯一的），那么一个键值对就是一个多叉树节点了，这棵多叉树就可以表示成这样：

```
// 1 -> [2, 3]  
// 2 -> [4]  
// 3 -> [5, 6]  
  
HashMap<Integer, List<Integer>> tree = new HashMap<>();  
tree.put(1, Arrays.asList(2, 3));  
tree.put(2, Collections.singletonList(4));  
tree.put(3, Arrays.asList(5, 6));
```

这样就可以模拟和操作二叉树/多叉树结构，后文讲到图论的时候你就会知道，它有一个新的名字叫做[邻接表](#)。

# 二叉树的递归/层序遍历

阅读本文前，你需要先学习：

- 二叉树的递归/层序遍历

了解了[二叉树基本概念和几种特殊的二叉树](#)，本文来讲解如何遍历和访问二叉树的节点。

## 递归遍历 (DFS)

二叉树的遍历框架：

```
// 基本的二叉树节点
class TreeNode {
    int val;
    TreeNode left, right;
}

// 二叉树的遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    traverse(root.left);
    traverse(root.right);
}
```

这段短小精干的代码为什么能遍历二叉树？又是以什么顺序遍历二叉树的？

对于 `traverse` 这样的递归遍历函数，你就可以把它理解成一个在二叉树结构上游走的指针，下面我用一个可视化就能直观地给你展现这个函数是如何遍历二叉树的。

点开这个可视化面板，右侧 `root` 指针的位置就是当前正在访问的节点，即 `traverse` 函数当前遍历到的位置，橙黄色的部分展示了 `traverse` 函数的堆栈，你可以点击左上角的播放按钮，观察 `root` 指针的位置变化：

### ▶ 🎃 代码可视化动画🎃

`traverse` 函数的遍历顺序就是一直往左子节点走，直到遇到空指针不能再走了，才尝试往右子节点走一步；然后再一直尝试往左子节点走，如此循环；如果左右子树都走完了，则返回上一层父节点。其实看代码也能看出来，先递归调用的 `root.left`，然后才递归调用的 `root.right` 嘛。

**这个递归遍历节点顺序是固定的，务必记住这个顺序，否则你肯定玩不转二叉树结构。**

那么有一些数据结构基础的读者肯定要问了：不对呀，只要上过大学的数据结构课程都知道，二叉树有前/中/后序三种遍历，会得到三种不同顺序的结果。为啥你这里说递归遍历节点的顺序是固定的呢？

这个问题很好，我来解答一下。

递归遍历的顺序，即 `traverse` 函数访问节点的顺序确实是固定的。正如上面那个可视化面板，`root` 指针在树上移动的顺序是固定的。

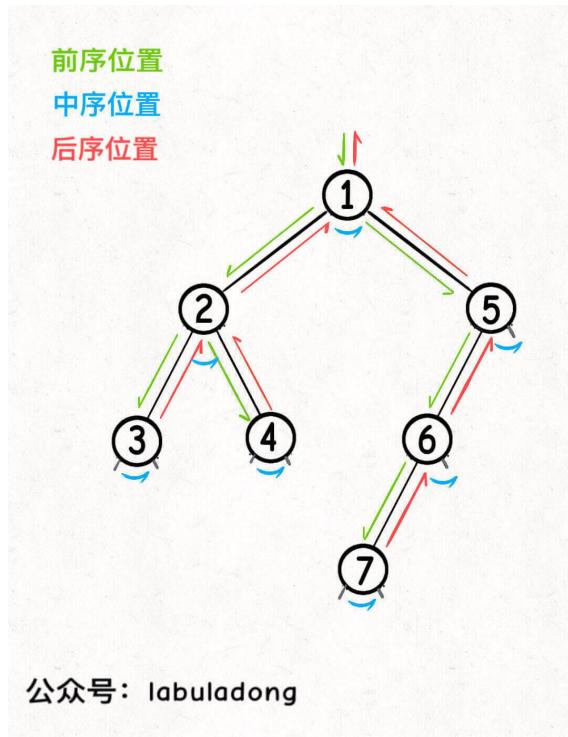
但是你在 `traverse` 函数中不同位置写代码，效果是可以不一样的。前中后序遍历的结果不同，原因是因为你把代码写在了不同位置，所以产生了不同的效果。

比方说，刚进入一个节点的时候，你还对它的子节点一无所知，而当你要离开一个节点的时候，它的所有子节点你都遍历过了。那么在这两种情况下写的代码，肯定是可以有不同的效果的。

你所谓的前中后序遍历，其实就是在上述二叉树遍历框架的不同位置写代码：

```
// 二叉树的遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    traverse(root.left);
    // 中序位置
    traverse(root.right);
    // 后序位置
}
```

前序位置的代码会在进入节点时执行；中序位置的代码会在左子树遍历完成后，遍历右子树之前执行；后序位置的代码会在左右子树遍历完成后执行：



你可以打开这个可视化面板，尝试以下操作：

- 1、一直点击 `preorderResult.push` 这行代码，观察右侧面板中 `root` 指针在树上移动的顺序和 `preorderResult` 的元素，这就是前序遍历顺序。
- 2、将可视化面板复位，尝试一直点击 `inorderResult.push` 这行代码并观察 `root` 指针的移动和 `inorderResult` 的元素，这就是中序遍历顺序。
- 3、将可视化面板复位，尝试一直点击 `postorderResult.push` 这行代码并观察 `root` 指针的移动和 `postorderResult` 的元素，这就是后序遍历顺序。

## ▶ 😊 代码可视化动画😊

二叉树的前中后序位置非常重要，我会在 [二叉树算法思想（纲领篇）](#) 和习题中深入探讨二叉树的前中后序遍历的区别和联系，以及如何运用到回溯算法、动态规划算法中，这里不展开。

那还有读者可能会问，为这个 `traverse` 函数要先递归遍历 `root.left` 后递归遍历 `root.right` 呢？能不能反过来？

当然可以，但一般所说的前中后序是默认先左后右的，所以这是个约定俗成的习惯。如果就你非要先右后左，那么你的前中后序遍历也和一般人理解的不一样，这不是徒增沟通成本？所以，除非做题的时候有特殊的需要，否则就按照先左后右的顺序写代码即可。

这里需要强调的是，[二叉搜索树（BST）](#) 的中序遍历结果是有序的，这是 BST 的一个重要性质。

可以看这个可视化面板，点击其中 `res.push(root.val);` 这一行代码，就可以看到中序遍历访问节点的顺序：

## ▶ 😊 代码可视化动画😊

在后续 [BST 相关习题集](#) 中，会有一些题目利用到这个特性。

## 层序遍历（BFS）

上面的递归遍历是依赖函数堆栈递归遍历二叉树的，如果把递归函数 `traverse` 看做一个指针，那么这个指针在二叉树上游走的顺序大概是从最左侧开始，一列一列走到最右侧。

你可以结合可视化面板，点击其中的 `if (root == null)` 这一行代码，看 `root` 变量在树上游走的顺序，就能直观地看到 `traverse` 函数访问节点的顺序：

## ▶ 😊 代码可视化动画😊

二叉树的层序遍历，顾名思义，就是一层一层地遍历二叉树。这个遍历方式需要借助队列来实现，而且根据不同的需求，主要有三种不同的写法，下面一一列举。

### 写法一

这是最简单的写法，代码如下：

```
void levelOrderTraverse(TreeNode root) {
    if (root == null) {
        return;
    }
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    while (!q.isEmpty()) {
        TreeNode cur = q.poll();
        // 访问 cur 节点
        System.out.println(cur.val);

        // 把 cur 的左右子节点加入队列
        if (cur.left != null) {
            q.offer(cur.left);
        }
        if (cur.right != null) {
```

```
        q.offer(cur.right);
    }
}
}
```

你可以打开这个可视化面板，点击其中的 `console.log(cur.val)`；这一行代码，观察 `cur` 变量在树上游走的顺序，就可以看到层序遍历是一层一层，从左到右的遍历二叉树节点：

▶  代码可视化动画

这种写法最大的优势就是简单。每次把队头元素拿出来，然后把它的左右子节点加入队列，就完事了。

但是这种写法的缺点是，无法知道当前节点在第几层。知道节点的层数是个常见的需求，比方说让你收集每一层的节点，或者计算二叉树的最小深度等等。

所以这种写法虽然简单，但用的不多，下面介绍的写法会更常见一些。

## 写法二

对上面的解法稍加改造，就得出了下面这种写法：

```
void levelOrderTraverse(TreeNode root) {
    if (root == null) {
        return;
    }
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    // 记录当前遍历到的层数（根节点视为第 1 层）
    int depth = 1;

    while (!q.isEmpty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 访问 cur 节点，同时知道它所在的层数
            System.out.println("depth = " + depth + ", val = " + cur.val);

            // 把 cur 的左右子节点加入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
        depth++;
    }
}
```

注意代码中的内层 for 循环：

```
int sz = q.size();
for (int i = 0; i < sz; i++) {
    ...
}
```

这个变量 `i` 记录的是节点 `cur` 是当前层的第几个，大部分算法题中都不会用到这个变量，所以你完全可以改用下面的写法：

```
int sz = q.size();
while (sz-- > 0) {
    ...
}
```

这个属于细节问题，按照自己的喜好来就行。

但是注意队列的长度 `sz` 一定要在循环开始前保存下来，因为在循环过程中队列的长度是会变化的，不能直接用 `q.size()` 作为循环条件。

你可以打开这个可视化面板，点击其中的 `console.log` 这一行代码，观察 `cur` 变量在树上游走的顺序，就可以看到还是一层一层，从左到右的遍历二叉树节点，但是这次还会输出节点所在的层数：

### ▶ 代码可视化动画

这种写法就可以记录下来每个节点所在的层数，可以解决诸如二叉树最小深度这样的问题，是我们最常用的层序遍历写法。

## 写法三

既然写法二是最常见的，为啥还有个写法三呢？因为要给后面的进阶内容做铺垫。

现在我们只是在探讨二叉树的层序遍历，但是二叉树的层序遍历可以衍生出 [多叉树的层序遍历](#)，[图的 BFS 遍历](#)，以及经典的 [BFS 暴力穷举算法框架](#)，所以这里要拓展延伸一下。

回顾写法二，我们每向下遍历一层，就给 `depth` 加 1，可以理解为每条树枝的权重是 1，二叉树中每个节点的深度，其实就是从根节点到这个节点的路径权重和，且同一层的所有节点，路径权重和都是相同的。

那么假设，如果每条树枝的权重和可以是任意值，现在让你层序遍历整棵树，打印每个节点的路径权重和，你会怎么做？

这样的话，同一层节点的路径权重和就不一定相同了，写法二这样只维护一个 `depth` 变量就无法满足需求了。

写法三就是为了解决这个问题，在写法一的基础上添加一个 `State` 类，让每个节点自己负责维护自己的路径权重和，代码如下：

```
class State {
    TreeNode node;
    int depth;

    State(TreeNode node, int depth) {
        this.node = node;
        this.depth = depth;
    }
}
```

```

}

void levelOrderTraverse(TreeNode root) {
    if (root == null) {
        return;
    }
    Queue<State> q = new LinkedList<>();
    // 根节点的路径权重和是 1
    q.offer(new State(root, 1));

    while (!q.isEmpty()) {
        State cur = q.poll();
        // 访问 cur 节点，同时知道它的路径权重和
        System.out.println("depth = " + cur.depth + ", val = " + cur.node.val);

        // 把 cur 的左右子节点加入队列
        if (cur.node.left != null) {
            q.offer(new State(cur.node.left, cur.depth + 1));
        }
        if (cur.node.right != null) {
            q.offer(new State(cur.node.right, cur.depth + 1));
        }
    }
}

```

你可以打开这个可视化面板，点击其中的 `console.log` 这一行代码，就可以看到还是一层一层，从左到右的遍历二叉树节点，还会输出节点所在的层数：

### ▶ 🎃 代码可视化动画🎃

这样每个节点都有了自己的 `depth` 变量，是最灵活的，可以满足所有 `BFS` 算法的需求。但是由于要额外定义一个 `State` 类比较麻烦，所以非必要的话，用写法二就够了。

其实你很快就会学到，这种边带有权重的场景属于图结构算法，在之后的 `BFS 算法习题集` 和 `dijkstra 算法` 中，才会用到这种写法。

## 其他遍历？

二叉树的遍历方式只有上面两种，也许有其他的写法，但都是表现形式上的差异，本质上不可能跳出上面两种遍历方式。

比方说，你可能看到用栈来迭代遍历二叉树的代码。但这本质还是递归遍历，只不过他手动维护栈模拟递归调用罢了。

再比如，你还可能看到递归地一层层遍历二叉树的代码。但这本质还是层序遍历，只不过他把层序遍历代码中的 `for` 循环用递归的形式展现了。

总之，不要被表象迷惑，二叉树的遍历方式就上面两种，结合后面的教程和习题，你把这两种遍历方式玩明白，一切暴力穷举算法都小菜一碟。

## 为什么 `BFS` 常用来寻找最短路径

用可视化面板结合一道例题，你立刻就能明白了。

来看力扣第 111 题「二叉树的最小深度」：

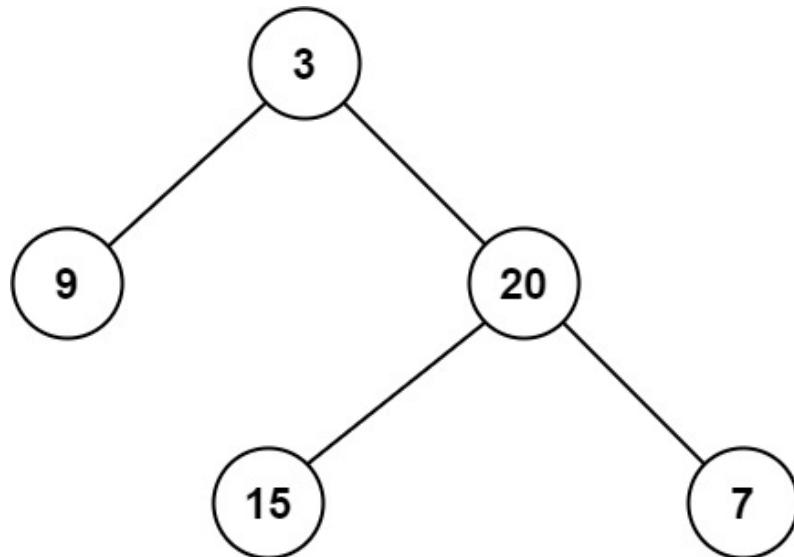
### ▼ 111. 二叉树的最小深度 [Leetcode](#) | [力扣](#)

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

示例 1：



```
输入: root = [3,9,20,null,null,15,7]
输出: 2
```

示例 2：

```
输入: root = [2,null,3,null,4,null,5,null,6]
输出: 5
```

提示：

- 树中节点数的范围在  $[0, 10^5]$  内
- $-1000 \leq \text{node.val} \leq 1000$

二叉树的最小深度即「根节点到最近的叶子节点的距离」，所以这道题本质上就是让你求最短距离。

DFS 递归遍历和 BFS 层序遍历都可以解决这道题，先看 DFS 递归遍历的解法：

```
class Solution {
    // 记录最小深度 (根节点到最近的叶子节点的距离)
    private int minDepth = Integer.MAX_VALUE;
    // 记录当前遍历到的节点深度
    private int currentDepth = 0;

    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 从根节点开始 DFS 遍历
        traverse(root);
    }

    private void traverse(TreeNode node) {
        if (node == null) {
            return;
        }
        currentDepth++;
        if (node.left == null && node.right == null) {
            minDepth = Math.min(minDepth, currentDepth);
        } else {
            traverse(node.left);
            traverse(node.right);
        }
        currentDepth--;
    }
}
```

```
        return minDepth;
    }

    private void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // 前序位置进入节点时增加当前深度
        currentDepth++;

        // 如果当前节点是叶子节点，更新最小深度
        if (root.left == null && root.right == null) {
            minDepth = Math.min(minDepth, currentDepth);
        }

        traverse(root.left);
        traverse(root.right);

        // 后序位置离开节点时减少当前深度
        currentDepth--;
    }
}
```

你可以点开下面这个可视化面板，点击其中 `if (root == null)` 这一行代码，即可看到递归函数 `traverse` 遍历二叉树节点的顺序。形象地说，它是从左到右，一条树枝一条树枝进行遍历的：

---

▶  [代码可视化动画](#) 

---

每当遍历到一条树枝的叶子节点，就会更新最小深度，**当遍历完整棵树后，就能算出整棵树的最小深度。**

你能不能在不遍历完整棵树的情况下，提前结束算法？不可以，因为你必须确切的知道每条树枝的深度（根节点到叶子节点的距离），才能找到最小的那个。

下面来看 BFS 层序遍历的解法。按照 BFS 从上到下逐层遍历二叉树的特点，当遍历到第一个叶子节点时，就能得到最小深度：

```
class Solution {
    public int minDepth(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // root 本身就是一层，depth 初始化为 1
        int depth = 1;

        while (!q.isEmpty()) {
            int sz = q.size();
            // 遍历当前层的节点
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                // 判断是否到达叶子结点
                if (cur.left == null && cur.right == null)
                    return depth;
                // 将下一层节点加入队列
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            depth++;
        }
    }
}
```

```
        q.offer(cur.left);
        if (cur.right != null)
            q.offer(cur.right);
    }
    // 这里增加步数
    depth++;
}
return depth;
}
```

你可以点开下面这个可视化面板，点击其中 `if (cur.left === null && cur.right === null)` 这一行代码，即可看到 BFS 一层一层遍历二叉树节点：

---

▶  代码可视化动画 

---

当它遍历到第二行的时候，就遇到第一个叶子节点了，这个叶子节点就是距离根节点最近的叶子节点，所以此时算法就结束了。

点击 `let result =` 这段代码，观察二叉树上节点的颜色，即可直观地看到 BFS 算法并没有遍历整棵树就找到了最小深度。

综上，你应该能理解为啥 BFS 算法经常用来寻找最短路径了：

由于 BFS 逐层遍历的逻辑，第一次遇到目标节点时，所经过的路径就是最短路径，算法可能并不需要遍历完所有节点就能提前结束。

DFS 遍历当然也可以用来寻找最短路径，但必须遍历完所有节点才能得到最短路径。

从时间复杂度的角度来看，两种算法在最坏情况下都会遍历所有节点，时间复杂度都是  $O(N)$ ，但在一般情况下，显然 BFS 算法的实际效率会更高。所以在寻找最短路径的问题中，BFS 算法是首选。

## 为什么 DFS 常用来寻找所有路径

在寻找所有路径的问题中，你会发现 DFS 算法用的比较多，BFS 算法似乎用的不多。

理论上两种遍历算法都是可以的，只不过 BFS 算法寻找所有路径的代码比较复杂，DFS 算法代码比较简洁。

你想啊，就以二叉树为例，如果要用 BFS 算法来寻找所有路径（根节点到每个叶子节点都是一条路径），队列里面就不能只放节点了，而需要使用第三种写法，新建一个 `State` 类，把当前节点以及到达当前节点的路径都存进去，这样才能正确维护每个节点的路径，最终计算出所有路径。

而使用 DFS 算法就简单了，它本就是一条树枝一条树枝从左往右遍历的，每条树枝就是一条路径，所以 DFS 算法天然适合寻找所有路径。

最后结合代码和可视化面板讲解一下，先看 DFS 算法的可视化面板，你可以多次点击 `if (root === null)` 这部分代码，观察 DFS 算法遍历所有节点并收集根节点到叶子节点的所有路径：

---

▶  代码可视化动画 

---

再看 BFS 算法的可视化面板，你可以多次点击 `if (node.left === null && node.right === null)` 这部分代码，即可观察 BFS 算法遍历所有节点并收集根节点到叶子节点的所有路径。鼠标移动到下方生成的 bfs 树节点上，可以看到每个 `State` 中存储的路径：

▶  代码可视化动画 

---

结合可视化面板中的 JavaScript 代码，可以明显感觉 BFS 算法代码要复杂。

综上，DFS 算法在寻找所有路径的问题中更常用，而 BFS 算法在寻找最短路径的问题中更常用。

# 多叉树的递归/层序遍历

阅读本文前，你需要先学习：

- [二叉树的递归/层序遍历](#)

多叉树结构就是 [二叉树结构](#) 的延伸。

多叉树的遍历就是 [二叉树遍历](#) 的延伸。

二叉树的节点长这样，每个节点有两个子节点：

```
class TreeNode {  
    int val;  
    TreeNode left;  
    TreeNode right;  
}
```

多叉树的节点长这样，每个节点有任意个子节点：

```
class Node {  
    int val;  
    List<Node> children;  
}
```

就这点区别，其他没了。

接下来我们来说下多叉树的遍历，和二叉树一样，也就递归遍历（DFS）和层序遍历（BFS）两种。

## 递归遍历（DFS）

对比二叉树的遍历框架看多叉树的遍历框架吧：

```
// 二叉树的遍历框架  
void traverse(TreeNode root) {  
    if (root == null) {  
        return;  
    }  
    // 前序位置  
    traverse(root.left);  
    // 中序位置  
    traverse(root.right);  
    // 后序位置  
}  
  
// N 叉树的遍历框架  
void traverse(Node root) {  
    if (root == null) {
```

```
        return;
    }
    // 前序位置
    for (Node child : root.children) {
        traverse(child);
    }
    // 后序位置
}
```

唯一的区别是，多叉树没有了中序位置，因为可能有多个节点嘛，所谓的中序位置也就没什么意义了。

可以看看下面这个可视化面板，点击其中 `preorderResult.push` 或 `postorderResult.push` 的代码部分，观察前序和后序遍历的执行顺序，其实和二叉树是一样的：

---

▶ 🎃 代码可视化动画🎃

---

## 层序遍历 (BFS)

多叉树的层序遍历和 [二叉树的层序遍历](#) 一样，都是用队列来实现，无非就是把二叉树的左右子节点换成了多叉树的所有子节点。所以多叉树的层序遍历也有三种写法，下面一一列举。

### 写法一

第一种层序遍历写法：

```
void levelOrderTraverse(Node root) {
    if (root == null) {
        return;
    }
    Queue<Node> q = new LinkedList<>();
    q.offer(root);
    while (!q.isEmpty()) {
        Node cur = q.poll();
        // 访问 cur 节点
        System.out.println(cur.val);

        // 把 cur 的所有子节点加入队列
        for (Node child : cur.children) {
            q.offer(child);
        }
    }
}
```

### 写法二

第二种能够记录节点深度的层序遍历写法：

```
void levelOrderTraverse(Node root) {
    if (root == null) {
        return;
    }
    Queue<Node> q = new LinkedList<>();
    q.offer(root);
```

```
// 记录当前遍历到的层数（根节点视为第 1 层）
int depth = 1;

while (!q.isEmpty()) {
    int sz = q.size();
    for (int i = 0; i < sz; i++) {
        Node cur = q.poll();
        // 访问 cur 节点，同时知道它所在的层数
        System.out.println("depth = " + depth + ", val = " + cur.val);

        for (Node child : cur.children) {
            q.offer(child);
        }
    }
    depth++;
}
```

## 写法三

第三种能够适配不同权重边的写法：

```
class State {
    Node node;
    int depth;

    public State(Node node, int depth) {
        this.node = node;
        this.depth = depth;
    }
}

void levelOrderTraverse(Node root) {
    if (root == null) {
        return;
    }
    Queue<State> q = new LinkedList<>();
    // 记录当前遍历到的层数（根节点视为第 1 层）
    q.offer(new State(root, 1));

    while (!q.isEmpty()) {
        State state = q.poll();
        Node cur = state.node;
        int depth = state.depth;
        // 访问 cur 节点，同时知道它所在的层数
        System.out.println("depth = " + depth + ", val = " + cur.val);

        for (Node child : cur.children) {
            q.offer(new State(child, depth + 1));
        }
    }
}
```

没啥好说的，有不明白的区对照前文 [二叉树遍历](#) 的层序遍历代码和可视化面板吧。

# 二叉搜索树的应用及可视化

阅读本文前，你需要先学习：

- 二叉树基础及常见类型
- 多叉树的递归/层序遍历

二叉搜索树是特殊的 [二叉树结构](#)，其主要的实际应用是 [TreeMap](#) 和 [TreeSet](#)。

前文 [几种常见的二叉树类型](#) 介绍二叉搜索树，接下来我会带你亲自实现一个类似 Java 标准库的 [TreeMap](#) 和 [TreeSet](#) 结构，帮助你知行合一。

不过呢，考虑到本文还处在数据结构基础的章节，本文仅讲解 [TreeMap/TreeSet](#) 的原理，动手实现 [TreeMap/TreeSet](#) 我放到了二叉树系列习题的后面。

因为和前面的哈希表、队列这些数据结构不同，树相关的数据结构需要比较强的递归思维，难度会上一个层级。如果你对递归的理解不够深入，现在给你讲的话不仅学习曲线有些陡峭，而且意义不大，就算你费了半天劲看懂了，遇到实际的题目还是不会，这很打击信心。

所以我建议循序渐进，后面二叉树的习题章节，用 100 多道实际的算法题手把手带你培养递归思维。刷完后你就可以秒杀所有二叉树相关的算法题了，再去看树相关的数据结构实现，就会感觉非常简单。甚至你都不用看我的代码，自己凭感觉就能实现 [TreeMap/TreeSet](#)。

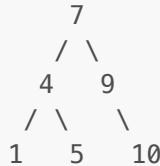
好了，废话不多说，让我们开始吧。

## 二叉搜索树的优势

前文 [几种常见的二叉树类型](#) 介绍过二叉搜索树（BST）的特点，即左小右大：

对于树中的每个节点，其左子树的每个节点的值都要小于这个节点的值，右子树的每个节点的值都要大于这个节点的值。

比方说下面这棵树就是一棵 BST：



这个左小右大的特性，可以让我们在 BST 中快速找到某个节点，或者找到某个范围内的所有节点，这是 BST 的优势所在。

你应该已经学过前文 [二叉树的遍历](#)，下面用标准的二叉树遍历函数结合可视化面板来对比展示一下 BST 和普通二叉树的操作差别。

你可以展开下面的两个面板，点击其中 `if (targetNode != null)` 这一行代码，直观感受一下两个搜索算法的效率差别：

▶ 🎃 代码可视化动画🎃

▶ 🎉 代码可视化动画🎉

这里展示的是查找目标元素的场景，可以看到，利用 BST 左小右大的特性，可以迅速定位到目标节点，理想的时间复杂度是树的高度  $O(\log N)$ ，而普通的二叉树遍历函数则需要  $O(N)$  的时间遍历所有节点。

至于其他增、删、改的操作，你首先查到目标节点，才能进行增删改的操作对吧？增删改的操作无非就是改一改指针，所以增删改的时间复杂度也是  $O(\log N)$ 。

## TreeMap/TreeSet 实现原理

你看 **TreeMap** 这个名字，应该就能看出来，它和前文介绍的 **哈希表 HashMap** 的结构是类似的，都是存储键值对的，**HashMap** 底层把键值对存储在一个 **table** 数组里面，而 **TreeMap** 底层把键值对存储在一棵二叉搜索树的节点里面。

至于 **TreeSet**，它和 **TreeMap** 的关系正如哈希表 **HashMap** 和哈希集合 **HashSet** 的关系一样，说白了就是 **TreeMap** 的简单封装，所以下面主要讲解 **TreeMap** 的实现原理。

力扣经典的 **TreeNode** 结构长这样：

```
class TreeNode {  
    int val;  
    TreeNode left;  
    TreeNode right;  
    TreeNode(int x) { this.val = x; }  
}
```

我们只要改一改这个经典的 **TreeNode** 结构，就可以用来实现 **TreeMap** 了：

```
// 大写 K 为键的类型，大写 V 为值的类型  
class TreeNode<K, V> {  
    K key;  
    V value;  
  
    TreeNode<K, V> left;  
    TreeNode<K, V> right;  
    TreeNode(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
}
```

我们将实现的 **TreeMap** 结构有如下 API：

```
// TreeMap 主要接口  
class MyTreeMap<K, V> {  
  
    // ***** Map 键值映射的基本方法 *****  
  
    // 增/改，复杂度 O(logN)  
    public void put(K key, V value) {}  
  
    // 查，复杂度 O(logN)  
    public V get(K key) {}
```

```
// 删, 复杂度 O(logN)
public void remove(K key) {}

// 是否包含键 key, 复杂度 O(logN)
public boolean containsKey(K key) {}

// 返回所有键的集合, 结果有序, 复杂度 O(N)
public List<K> keys() {}

// ***** TreeMap 提供的额外方法 *****
// 查找最小键, 复杂度 O(logN)
public K firstKey() {}

// 查找最大键, 复杂度 O(logN)
public K lastKey() {}

// 查找小于等于 key 的最大键, 复杂度 O(logN)
public K floorKey(K key) {}

// 查找大于等于 key 的最小键, 复杂度 O(logN)
public K ceilingKey(K key) {}

// 查找排名为 k 的键, 复杂度 O(logN)
public K selectKey(int k) {}

// 查找键 key 的排名, 复杂度 O(logN)
public int rank(K key) {}

// 区间查找, 复杂度 O(logN + M), M 为区间大小
public List<K> rangeKeys(K low, K high) {}

}
```

除了标准的增删查改方法 `get`, `put`, `remove`, `containsKey` 之外, `TreeMap` 还提供了很多额外方法, 主要和 `key` 的大小相关。怎么样, 是不是感觉很强大?

哈希表很实用, 但是它确实没办法很好地处理键之间的大小关系。前文 [用数组加强哈希表](#) 中实现的 `LinkedHashMap` 也只是做到按「插入顺序」排列哈希表中的键, 依然做不到按「大小顺序」排列。

本文为 labuladong.online 网站会员内容, 请 [点这里](#) 查看。

# 红黑树的完美平衡及可视化

阅读本文前，你需要先学习：

- 二叉树基础及常见类型
- 多叉树的递归/层序遍历
- 二叉搜索树的应用及可视化

红黑树是自平衡的二叉搜索树，它的树高在任何时候都能保持在  $O(\log N)$ （完美平衡），这样就能保证增删查改的时间复杂度都是  $O(\log N)$ 。

可视化面板支持创建红黑树：

▶ 🎨 代码可视化动画🎨

[二叉搜索树的应用及可视化](#) 讲了普通的二叉搜索树存储键值对实现 `TreeMap`/`TreeSet` 的思路。

二叉搜索树的操作效率取决于树高，树结构越平衡，树高就接近  $\log N$ ，增删查改的效率就比较高。而普通二叉搜索树最关键的问题是它不会自动对树进行平衡，特殊的情况下会退化成链表，增删查改的时间复杂度退化为  $O(N)$ 。

下面这个可视化面板就是一个例子，如果插入若干个有序的键值对，你就能发现每次新增的键都会被插入到最右侧，导致这棵二叉搜索树退化成了链表：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Trie/字典树/前缀树原理及可视化

阅读本文前，你需要先学习：

- [二叉树基础及常见类型](#)

Trie 树就是 [多叉树结构](#) 的延伸，是一种针对字符串进行特殊优化的数据结构。

Trie 树在处理字符串相关操作时有诸多优势，比如节省公共字符串前缀的内存空间、方便处理前缀操作、支持通配符匹配等。

下面这个可视化面板展示了 Trie 树的结构和主要 API，你可以逐行点击代码，观察 console 输出和右侧的 Trie 树结构变化：

▶ 🎃 代码可视化动画🎃

本文仅是 Trie 树（也叫做字典树、前缀树）的原理介绍，[动手实现 TrieMap/TrieSet](#) 我放到了二叉树系列习题章节后面的数据结构设计章节。理由和上篇 [TreeMap/TreeSet 原理](#) 相同，在基础知识章节我不准备讲解这种复杂结构的具体实现，初学者也没必要在这个阶段理解 Trie 树的代码实现。

但是我依然把 Trie 树的原理讲解放在这里，有两个目的：

- 1、让你直观地感受到二叉树结构的种种幻化，你也许能理解我的教程特别强调二叉树结构的原因了。
- 2、在开头让你知道有这么一种数据结构，了解它的 API 以及适用的场景。未来你遇到相关的问题，也许就能想到用 Trie 树来解决，最起码有个思路，大不了回来复制代码模板嘛。这种数据结构的实现都是固定的，笔试面试也不会让你从头手搓 Trie 树，复制粘贴直接拿去用就可以了。

好了，废话不多说，让我们开始吧。

对于 Trie 树的讲解，我会带你实现一个 [TrieMap](#) 和 [TrieSet](#)。先来梳理一下我们已经实现过的 Map/Set 类型：

- 标准的 [哈希表 HashMap](#)，底层借助一个哈希函数把键值对存在 [table](#) 数组中，有两种解决哈希冲突的方法。它的特点是快，即基本的增删查改操作时间复杂度都是  $O(1)$ 。哈希集合 [HashSet](#) 是 [HashMap](#) 的简单封装。
- [哈希链表 LinkedHashMap](#)，是 [双链表结构](#) 对标准哈希表的加强。它继承了哈希表的操作复杂度，并且可以让哈希表中的所有键保持「插入顺序」。[LinkedHashSet](#) 是 [LinkedHashMap](#) 的简单封装。
- [哈希数组 ArrayHashMap](#)，是 [数组结构](#) 对标准哈希表的加强。它继承了哈希表的操作复杂度，并且提供了一个额外的 [randomKey](#) 函数，可以在  $O(1)$  的时间返回一个随机键。[ArrayHashSet](#) 是 [ArrayHashMap](#) 的简单封装。
- [TreeMap 映射](#)，底层是一棵二叉搜索树，基本增删查改操作复杂度是  $O(\log N)$ ，它的特点是可以动态维护键值对的大小关系，有很多额外的 API 操作键值对。[TreeSet](#) 集合是 [TreeMap](#) 映射的简单封装。

[TrieSet](#) 也是 [TrieMap](#) 的简单封装，所以下面我们聚焦 [TrieMap](#) 的实现原理即可。

## Trie 树的主要应用场景

Trie 树是一种针对字符串有特殊优化的数据结构，这也许它又被叫做字典树的原因。Trie 树针对字符串的处理有若干优势，下面一一列举。

## 节约存储空间

用 `HashMap` 对比把，比如说这样存储几个键值对：

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 1);
map.put("app", 2);
map.put("appl", 3);
```

回想哈希表的实现原理，键值对会被存到 `table` 数组中，也就是说它真的创建 `"apple"`、`"app"`、`"appl"` 这三个字符串，占用了 12 个字符的内存空间。

但是注意，这三个字符串拥有共同的前缀，`"app"` 这个前缀被重复存储了三次，`"l"` 也被重复存储了两次。

如果换成 `TrieMap` 来存储：

```
// Trie 树的键类型固定为 String 类型，值类型可以是泛型
TrieMap<Integer> map = new TrieMap<>();
map.put("apple", 1);
map.put("app", 2);
map.put("appl", 3);
```

`Trie` 树底层并不会重复存储公共前缀，所以只需要 `"apple"` 这 5 个字符的内存空间来存储键。

这个例子数据量很小，你感觉重复存储几次没啥大不了，但如果键非常多、非常长，且存在大量公共前缀（现实中确实经常有这种情况，比如证件号），那么 `Trie` 树就能节约大量的内存空间。

## 方便处理前缀操作

举个例子就明白了：

```
// Trie 树的键类型固定为 String 类型，值类型可以是泛型
TrieMap<Integer> map = new TrieMap<>();
map.put("that", 1);
map.put("the", 2);
map.put("them", 3);
map.put("apple", 4);

// "the" 是 "themxyz" 的最短前缀
System.out.println(map.shortestPrefixOf("themxyz")); // "the"

// "them" 是 "themxyz" 的最长前缀
System.out.println(map.longestPrefixOf("themxyz")); // "them"

// "tha" 是 "that" 的前缀
System.out.println(map.hasKeyWithPrefix("tha")); // true

// 没有以 "thz" 为前缀的键
System.out.println(map.hasKeyWithPrefix("thz")); // false

// "that", "the", "them" 都是 "th" 的前缀
System.out.println(map.keysWithPrefix("th")); // ["that", "the", "them"]
```

除了 `keysWithPrefix` 方法的复杂度取决于返回结果的长度，其他前缀操作的复杂度都是  $O(L)$ ，其中  $L$  是前缀字符串长度。

你想想上面这几个操作，用 `HashMap` 或者 `TreeMap` 能做到吗？应该只能强行遍历所有键，然后一个个比较字符串前缀，复杂度非常高。

话说，这个 `keysWithPrefix` 方法，是不是很适合做自动补全功能呢？

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 二叉堆核心原理及可视化

阅读本文前，你需要先学习：

- 二叉树基础及常见类型
- 二叉树的递归/层序遍历

二叉堆是一种能够动态排序的数据结构，是 [二叉树结构](#) 的延伸。

二叉堆的主要操作就两个，`sink`（下沉）和 `swim`（上浮），用以维护二叉堆的性质。

二叉堆的主要应用有两个，首先是一种很有用的数据结构优先级队列（Priority Queue），第二是一种排序方法堆排序（Heap Sort）。

这个可视化面板直观地展示了二叉堆的基本操作，你可以点击跳转执行其中的代码，或自己修改代码玩一玩：

## ▶ 🎃 代码可视化动画🎃

下面我就结合可视化面板来展示二叉堆的原理，最后以优先级队列为例，展示二叉堆的代码实现。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 二叉堆/优先级队列代码实现

阅读本文前，你需要先学习：

- [二叉堆的原理](#)

前文 [二叉堆的原理](#) 介绍了二叉堆的基本性质、API 和常见应用。本文将结合 [可视化面板](#) 手把手带你实现一个优先级队列。

我们先实现一个简化版的优先级队列，用来帮你理解二叉堆的核心操作 `sink` 和 `swim`。最后我再用给出一个比较完整的代码实现。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 线段树核心原理及可视化

阅读本文前，你需要先学习：

- 二叉树基础及常见类型
- 二叉树的递归/层序遍历

线段树是 [二叉树结构](#) 的衍生，用于高效解决区间查询和动态修改问题，其中区间查询的时间复杂度为  $O(\log N)$ ，动态修改单个元素的时间复杂度为  $O(\log N)$ 。

可视化面板支持创建和使用线段树，下面的可视化面板展示了线段树的逻辑结构、底层数组和基本操作 `query`, `update`:

▶  [代码可视化动画](#) 

考虑到这是第一章，我并不准备深入讲解线段树的实现细节，而是带你了解线段树的基本原理以及应用场景，让你对线段树有一个直观的认识。具体的代码实现安排在二叉树系列习题章节后面的数据结构设计章节，这里没必要深入。

下面让我从 [选择排序](#) 中提到的一种优化思路开始，逐步引出线段树的原理，以及我们为什么需要线段树这种数据结构。

## 前情回顾

在前文 [选择排序](#) 中，我提出过一种使用 `suffixMin` 数组的优化尝试，即提前预算一个 `suffixMin` 数组，使得 `suffixMin[i] = min(nums[0..i])`，这样就可以在  $O(1)$  时间内查询 `nums[0..i]` 的最小值：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 图结构基础及通用代码实现

阅读本文前，你需要先学习：

- [多叉树的递归/层序遍历](#)

图结构就是 [多叉树结构](#) 的延伸。

在树结构中，只允许父节点指向子节点，不存在子节点指向父节点的情况，子节点之间也不会互相链接；而图中没有那么多限制，节点之间可以相互指向，形成复杂的网络结构。

其实我本来想把图这种数据结构也放到二叉树结构的延伸章节中。不过考虑到图结构特有的算法比较丰富，所以单独开一个章节来讲图结构及算法，也方便本站内容的更新。

因为图结构可以对更复杂的问题进行抽象，所以产生了更复杂的图论算法，比较经典的有 [二分图算法](#)、[拓扑排序](#)、[最短路径算法](#)、[最小生成树算法](#) 等，这些都会在后文介绍。

本文主要介绍图的基本概念，以及如何用代码实现图结构。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 图结构的 DFS/BFS 遍历

阅读本文前，你需要先学习：

- 图结构基础及通用代码实现
- 多叉树的递归/层序遍历

图的遍历就是 [多叉树遍历](#) 的延伸。主要的遍历方式还是深度优先搜索（DFS）和广度优先搜索（BFS）。唯一的区别是，树结构中不存在环，而图结构中可能存在环，所以我们需要标记遍历过的节点，避免遍历函数在环中死循环。

具体来说，遍历图的所有「节点」时，需要 `visited` 数组在前序位置标记节点；遍历图的所有「路径」时，需要 `onPath` 数组在前序位置标记节点，在后序位置撤销标记。

如果题目说这幅图不存在环，那么图的遍历就完全等同于多叉树的遍历。

下面我来结合实例，具体解释上述总结。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Union Find 并查集原理

阅读本文前，你需要先学习：

- 二叉树基础及常见类型
- 图结构基础及通用代码实现
- 图结构的 DFS/BFS 遍历

并查集（Union Find）结构是 [二叉树结构](#) 的衍生，用于高效解决无向图的连通性问题，可以在  $O(1)$  时间内合并两个连通分量，在  $O(1)$  时间内查询两个节点是否连通，在  $O(1)$  时间内查询连通分量的数量。

并查集算法有几种优化方法，可视化面板都做了支持。下面展示一个未经优化的并查集实现，最终多叉树几乎退化成单链表，导致算法效率降低。对于这个问题的优化思路和可视化展示，在下文中会详细介绍。

## ▶ 代码可视化动画

本文将介绍什么是图的动态连通性问题，以及为什么并查集（Union Find）算法能够高效解决动态连通性问题。

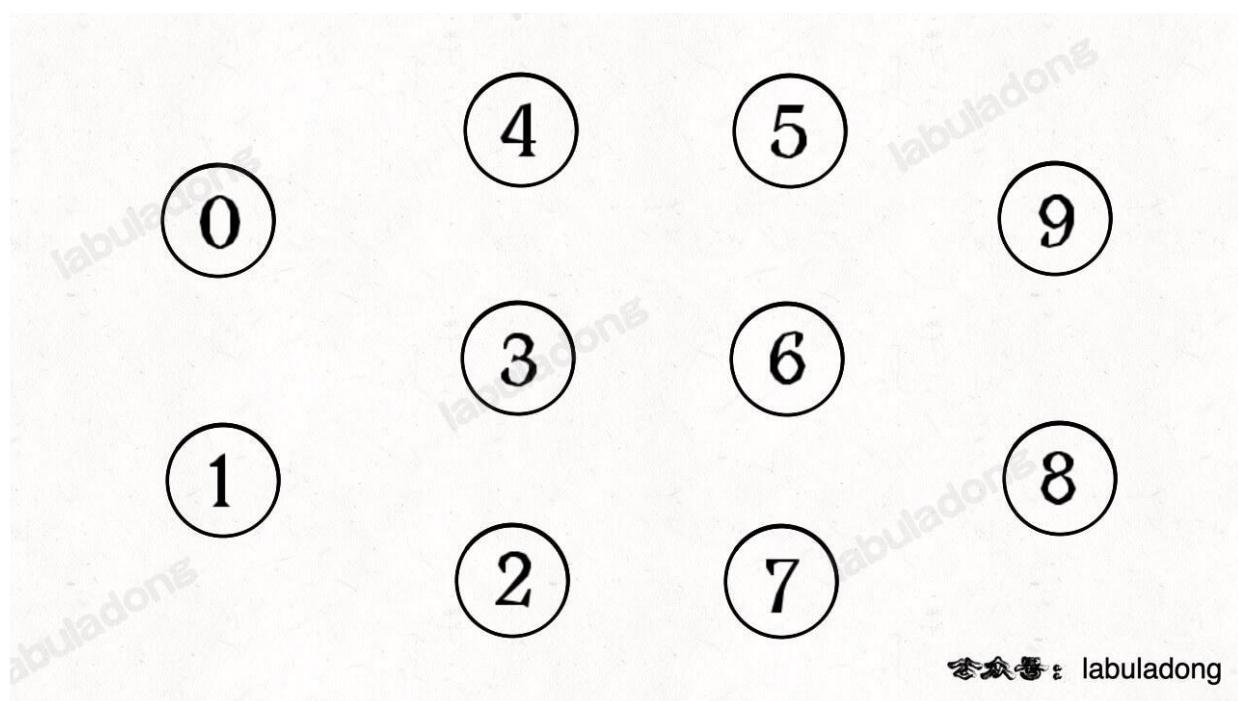
本文会结合 [可视化面板](#) 直观展示 Union Find 算法的核心原理，以及几种优化思路的效果。

考虑到这是基础章节，本文不涉及算法代码的实现细节。具体的代码实现和算法题的运用放在后面的 [Union Find 算法实现及应用](#) 和 [并查集经典习题](#) 章节中，建议初学者按照目录顺序循序渐进地学习。

## 动态连通性及术语

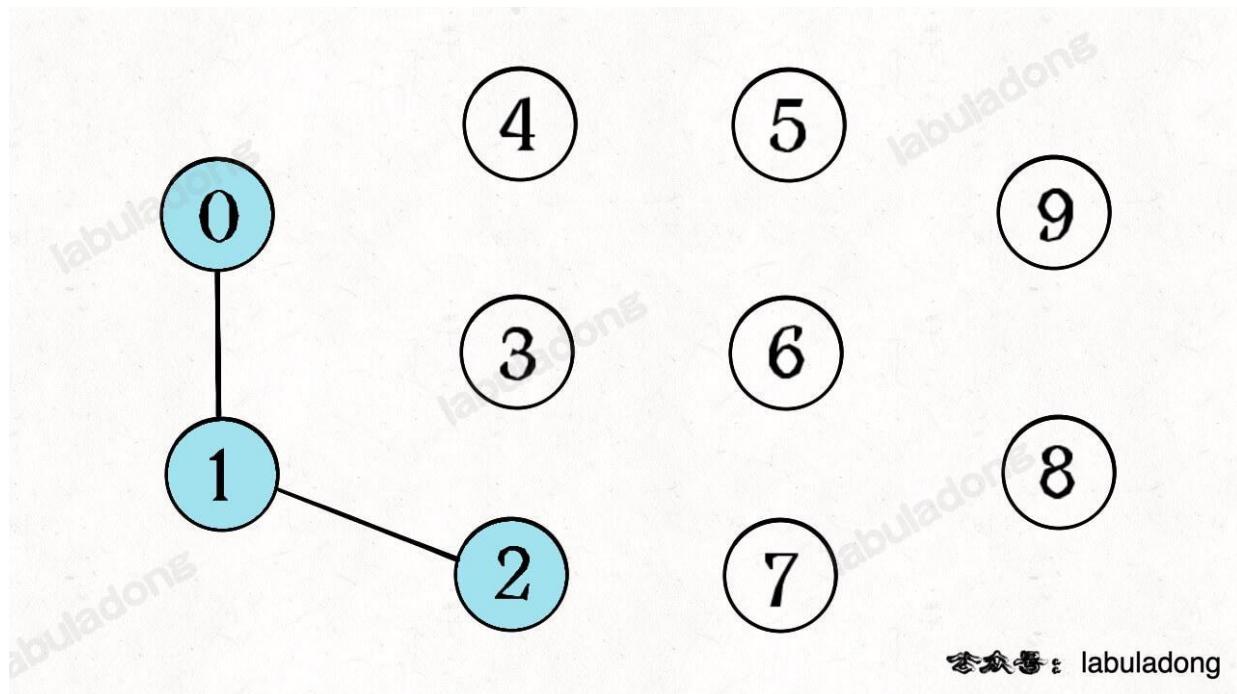
图论算法中专业术语比较多，我就用一个简单的例子来介绍几个专业术语。

比如下面这个例子，其中有 10 个节点，分别用 0~9 标记，虽然其中没有边，但它依然是一个图结构：



我们可以说这个图结构中，有 10 个「连通分量」，每个节点自身都是一个连通分量，因为它们自成一派，没有和其他节点相连。

现在将其中的一些节点进行「连接操作」：



© labuladong

此时，图结构中的节点  $0, 1, 2$  之间就有了连接关系，它们三个节点共同构成了一个连通分量，我们可以说这三个节点是「连通」的。

同时，这个图结构中的连通分量的数量从 10 减少到了 8，因为连接操作将多个连通分量合并成了一个。

- 1、自反性：节点  $p$  和  $p$  是连通的。
- 2、对称性：如果节点  $p$  和  $q$  连通，那么  $q$  和  $p$  也连通。
- 3、传递性：如果节点  $p$  和  $q$  连通， $q$  和  $r$  连通，那么  $p$  和  $r$  也连通。

判断这种「等价关系」非常实用，比如说编译器判断同一个内存对象的不同变量引用，比如社交网络中的朋友圈计算等等。

那么动态连通性问题就是说，给你输入一个图结构，然后进行若干次「连接操作」，同时可能会查询任意两个节点是否「连通」，或者查询当前图中有多少个「连通分量」。

我们的目标是设计一种数据结构，在尽可能小的时间复杂度下完成连接操作和查询操作。

## 为什么需要并查集算法

并查集（Union Find）结构提供如下 API：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 排序算法的关键指标

阅读本文前，你需要先学习：

- [数组基础](#)
- [时空复杂度入门](#)

一般刷题和面试/笔试的时候不会直接让你手撕排序算法，不过考虑到知识的完整性，我这里还是开一个章节，结合[可视化面板](#)讲解几种常见排序算法的原理、特点、时间复杂度和代码实现。

本文先介绍一下排序算法的几个关键指标，后面讲解到具体的排序算法时，都会根据这些指标来分析。

## 时空复杂度

首先一个指标肯定是时间复杂度和空间复杂度。

正如[时空复杂度入门](#)中所说，对于任意一个算法，其时间复杂度和空间复杂度都是越小越好的。

## 排序稳定性

稳定性是排序算法的一个重要性质，我们可以简单总结为：

对于序列中的相同元素，如果排序之后它们的相对位置没有发生改变，则称该排序算法为「稳定排序」，反之则为「不稳定排序」。

如果单单排序 int 数组，那么稳定性没有什么意义。但如果排序一些结构比较复杂的数据，那么稳定排序就会有一定的优势。

比如说现在你有若干订单数据，已经按照交易日期排好序了，现在你想对用户 ID 再进行排序，这样一来相同用户 ID 的订单就会聚集在一起，方便查看。稳定排序和不稳定排序的区别就体现在这里：

如果你用稳定排序算法，那么排序完成后，相同用户 ID 的订单依然会按照交易日期有序排列：

Date	UserID
2020-02-01	1001
2020-02-02	1001
2020-02-03	1001
2020-01-01	1002
2020-01-02	1002
2020-01-03	1002
...	

因为之前已经按照日期排好序了，对用户 ID 稳定排序之后，相同用户 ID 的订单的相对位置保持不变，所以在日期上依然是有序的。

如果你用不稳定排序算法，相同用户 ID 的订单相对位置可能变化，所以对于相同用户 ID 的订单，交易日期的有序性会丧失，相当于你之前对日期的排序白做了。

可以看到，稳定性是个很重要的性质，所以你在使用排序算法时要特别注意，避免出现预期之外的结果。

## 是否原地排序

原地排序就是指排序过程中不需要额外的辅助空间，只需要常数级别的额外空间，直接操作原数组进行排序。

注意，关键是是否需要额外的空间，而不是是否返回一个新的数组。具体来说就是类似这样的区别：

```
// 非原地排序
void sort(int[] nums) {
    // 排序过程中需要额外的辅助数组，消耗 O(N) 的空间
    int[] tmp = new int[nums.length];

    // 对 nums 进行排序
    for ...
}

// 原地排序
void sort(int[] nums) {
    // 直接操作 nums，不需要额外的辅助数组，消耗 O(1) 的空间
    for ...
}
```

不难想到，对于大数据量的排序，原地排序算法是比较有优势的。

排序算法的几个关键指标就是这些，后面我会介绍几种常见的排序算法，都会根据这些指标来分析它们的优劣。

# 选择排序所面临的问题

阅读本文前，你需要先学习：

- 排序算法的关键指标

选择排序是最简单朴素的排序算法，但是时间复杂度较高，且不是稳定排序。其他基础排序算法都是基于选择排序的优化。

你可以点开可视化面板，点击播放按钮，然后点击加速/减速按钮调节速度，即可直观感受选择排序的过程：

▶ 🎨 代码可视化动画

如果你是没接触过排序算法的初学者，那是最好的，不要急着看定义之类的东西；如果你之前了解过排序算法，现在请你忘记定义，忘记曾经背诵过的算法代码。

有了前面内容的铺垫，你已经有了一定的编程能力，能够解决一些基础的算法问题了。那么在这个前提下，我有一个学习方法分享，供你参考：

遇到一个新问题的时候，不要急着找人要一个标准答案，而应该启动自己的思考。被灌输一次标准答案，就错失一次机缘，少一分灵气。被灌得多了，人就傻了。

总有些读者，愁眉苦脸地找我诉苦，说算法题刷完了就忘怎么办啊。我还觉得这是好事呢，念念不忘的是执念，忘了才好，说明还没被塞满，这就是独立思考的机缘呀。

所以回到问题，让我们抓住这次机缘。现在就是给你输入一个数组，让你写个排序算法把所有元素从小到大排序，你说，怎么写？如果你从来没有思考过这个问题，可以停下几分钟想一想。

```
void sort(int[] nums) {
    // 你的代码，将 nums 中的元素从小到大排序
}
```

我第一次思考这个问题时，想到的最直接的方法是这样的：

先遍历一遍数组，找到数组中的最小值，然后把它和数组的第一个元素交换位置；接着再遍历一遍数组，找到第二小的元素，和数组的第二个元素交换位置；以此类推，直到整个数组有序。

这个算法有一个被大家熟知的名字，叫做「选择排序」，即每次都去遍历选择最小的元素。写成代码就是这样的：

@tab:active java

```
void sort(int[] nums) {
    int n = nums.length;
    // sortedIndex 是一个分割线
    // 索引 < sortedIndex 的元素都是已排序的
    // 索引 >= sortedIndex 的元素都是未排序的
    // 初始化为 0，表示整个数组都是未排序的
    int sortedIndex = 0;
    while (sortedIndex < n) {
```

```
// 找到未排序部分 [sortedIndex, n) 中的最小值
int minIndex = sortedIndex;
for (int i = sortedIndex + 1; i < n; i++) {
    if (nums[i] < nums[minIndex]) {
        minIndex = i;
    }
}
// 交换最小值和 sortedIndex 处的元素
int tmp = nums[sortedIndex];
nums[sortedIndex] = nums[minIndex];
nums[minIndex] = tmp;

// sortedIndex 后移一位
sortedIndex++;
}
```

@tab cpp

```
void sort(vector<int>& nums) {
    int n = nums.size();
    // sortedIndex 是一个分割线
    // 索引 < sortedIndex 的元素都是已排序的
    // 索引 >= sortedIndex 的元素都是未排序的
    // 初始化为 0, 表示整个数组都是未排序的
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 找到未排序部分 [sortedIndex, n) 中的最小值
        int minIndex = sortedIndex;
        for (int i = sortedIndex + 1; i < n; i++) {
            if (nums[i] < nums[minIndex]) {
                minIndex = i;
            }
        }
        // 交换最小值和 sortedIndex 处的元素
        int tmp = nums[sortedIndex];
        nums[sortedIndex] = nums[minIndex];
        nums[minIndex] = tmp;

        // sortedIndex 后移一位
        sortedIndex++;
    }
}
```

@tab python

```
def sort(nums: List[int]) -> None:
    n = len(nums)
    # sortedIndex 是一个分割线
    # 索引 < sortedIndex 的元素都是已排序的
    # 索引 >= sortedIndex 的元素都是未排序的
    # 初始化为 0, 表示整个数组都是未排序的
    sortedIndex = 0
    while sortedIndex < n:
        # 找到未排序部分 [sortedIndex, n) 中的最小值
```

```
minIndex = sortedIndex
for i in range(sortedIndex + 1, n):
    if nums[i] < nums[minIndex]:
        minIndex = i
# 交换最小值和 sortedIndex 处的元素
nums[sortedIndex], nums[minIndex] = nums[minIndex], nums[sortedIndex]

# sortedIndex 后移一位
sortedIndex += 1
```

@tab go

```
func sort(nums []int) {
    n := len(nums)
    // sortedIndex 是一个分割线
    // 索引 < sortedIndex 的元素都是已排序的
    // 索引 >= sortedIndex 的元素都是未排序的
    // 初始化为 0, 表示整个数组都是未排序的
    sortedIndex := 0
    for sortedIndex < n {
        // 找到未排序部分 [sortedIndex, n) 中的最小值
        minIndex := sortedIndex
        for i := sortedIndex + 1; i < n; i++ {
            if nums[i] < nums[minIndex] {
                minIndex = i
            }
        }
        // 交换最小值和 sortedIndex 处的元素
        nums[sortedIndex], nums[minIndex] = nums[minIndex], nums[sortedIndex]

        // sortedIndex 后移一位
        sortedIndex++
    }
}
```

@tab javascript

```
function sort(nums) {
    const n = nums.length;
    // sortedIndex 是一个分割线
    // 索引 < sortedIndex 的元素都是已排序的
    // 索引 >= sortedIndex 的元素都是未排序的
    // 初始化为 0, 表示整个数组都是未排序的
    let sortedIndex = 0;
    while (sortedIndex < n) {
        // 找到未排序部分 [sortedIndex, n) 中最小值的索引
        let minIndex = sortedIndex;
        for (let i = sortedIndex + 1; i < n; i++) {
            if (nums[i] < nums[minIndex]) {
                minIndex = i;
            }
        }
        // 交换最小值和 sortedIndex 处的元素
        [nums[sortedIndex], nums[minIndex]] = [nums[minIndex], nums[sortedIndex]];
    }
}
```

```
// sortedIndex 后移一位  
sortedIndex++;  
}  
}
```

上述算法的可视化过程如下：

### ▶ 🎃 代码可视化动画🎃

这个算法是正确的，稍加改动就可以作为力扣第 912 题「排序数组」的解法代码。

但这个算法无法通过 912 题的所有测试用例，最后会得到一个超时的错误，这说明算法的逻辑是正确的，只是时间复杂度较高，超出了题目的限制。

暂且不管如何通过 912 题，我们先来按照 [排序算法的几个关键指标](#) 来分析一下这个排序算法。

## 是否是原地排序

是的。因为算法并没有使用额外的数组空间进行辅助，只是用了几个变量，空间复杂度是  $O(1)$ 。

## 时空复杂度分析

这个 `sort` 函数中包含一个 `while` 循环嵌套一个 `for` 循环，相当于是这样：

```
for (int sortedIndex = 0; sortedIndex < n; sortedIndex++) {  
    for (int i = sortedIndex + 1; i < n; i++) {  
        // ...  
    }  
}
```

你看到了，这就是嵌套 `for` 循环，总的循环次数是  $(n - 1) + (n - 2) + (n - 3) + \dots + 1$ ，这是等差数列求和，结果近似是  $n^2 / 2$ ，所以这个排序算法的时间复杂度用 Big O 表示法就是  $O(n^2)$ ，其中  $n$  是待排序数组的元素个数。

而且你注意这个算法有个特点，即便整个数组已经是有序的，它还是会执行  $n^2 / 2$  次，即原始数据的有序度对算法的时间复杂度没有任何影响。

对于一般的算法时空复杂度分析，我们只需要从 Big O 表示法的角度来分析即可，即仅关心量级（最高次项）的大小，而不关心系数和低次项。

但是在分析不同排序算法的场景下，实际的执行次数，以及一些特殊情况（比如数组本身就有序的情况），还是有必要关注的。

因为有多种排序算法从 Big O 的视角来看都是  $O(n^2)$  复杂度，那么我们要根据他们的实际执行次数以及特殊情况下表现，来分析它们的优劣。

## 时间都去哪了？优化思路？

现在，请你观察这个算法的逻辑，仔细思考几分钟，时间复杂度是否还有优化的可能？

不要小看这里是基础章节，我讲的都是思维方法，未来你做任何题目，优化时间复杂度的思路和这里一模一样。

首先，如果代码没有写错，算法时间复杂度还是太高，那只有一种可能，就是存在冗余计算。

上述算法中出现冗余计算的地方比较容易看出来：

它首先遍历 `nums[0..]` 寻找最小值，然后遍历 `nums[1..]` 寻找最小值，然后遍历 `nums[2..]` 寻找最小值，以此类推。

那么请问，在遍历 `nums[0..]` 的时候，其实已经遍历过 `nums[1..]` 和 `nums[2..]` 的所有元素了，你为什么要再次遍历呢？

理论上，你应该可以在遍历 `nums[0..]` 的时候，顺便找到 `nums[1..]` 和 `nums[2..]` 的最小元素，对吧？如果能做到这一点，是不是就可以消掉内层的 for 循环，从而把时间复杂度降低一个数量级？

好，现在我们已经找到了冗余计算的症结所在，并且有了一个优化思路。那么这个思路是否可以实现呢？你是否能够在遍历 `nums[0..]` 的时候，顺便找到 `nums[1..]` 和 `nums[2..]` 的最小元素？

我将进行抽象，把这个优化场景转化成一个全新的问题：

给你一个数组 `nums`，请你计算一个新数组 `suffixMin` 数组，其中 `suffixMin[i]` 表示 `nums[i..]` 中的最小值。

如果正着思考，假设现在我知道了 `nums[0..]` 中的最小元素，我是否能够推导出 `nums[1..]` 中的最小元素呢？

答案是不可能。信息不足，我实在不知道如何根据 `min(nums[0..])` 推导出 `min(nums[1..])`，只能重新遍历一遍 `nums[1..]`。

但是，我自己都不相信，就是算个最小值，咋可能这么难搞呢？我的脑子被智子锁死了吗？？？

如果反过来思考，假设现在我知道了 `nums[1..]` 中的最小元素，我是否能够推导出 `nums[0..]` 中的最小元素呢？

答案是可以的，`min(nums[0..]) = min(nums[0], min(nums[1..]))`。

有了这个思路，这个 `suffixMin` 数组就能算出来了，关键是倒着计算：

```
int[] nums = new int[]{3, 1, 4, 2};  
// suffixMin[i] 表示 nums[i..] 中的最小值  
int[] suffixMin = new int[nums.length];  
  
// 从后往前计算 suffixMin  
suffixMin[nums.length - 1] = nums[nums.length - 1];  
for (int i = nums.length - 2; i >= 0; i--) {  
    suffixMin[i] = Math.min(nums[i], suffixMin[i + 1]);  
}  
  
// [1, 1, 2, 2]  
System.out.println(suffixMin);
```

好了，这个计算 `suffixMin` 数组的问题解决了，现在回到选择排序的优化，我现在只需要花  $O(n)$  的时间遍历一遍 `nums` 数组算出 `suffixMin` 数组，就可以在  $O(1)$  的时间内得到 `nums[1..]`, `nums[2..]`, ... 任意子数组的最小值。

按理说，现在我可以把选择排序的内层 for 循环消掉，时间复杂度优化成  $O(n)$  了，对吗？答案是不行。

请你思考几分钟，为什么不行，关键的问题在哪里？

有的读者可能会说，选择排序中需要知道最小元素的索引进行交换，而 `suffixMin` 数组里面只存储了最小元素的值，没有存储索引，所以无法优化选择排序。

但是，我完全可以建一个新数组 `minIndex`，在计算 `suffixMin` 数组的时候，同时在 `minIndex` 记录下最小元素对应的索引，所以这个问题不是关键。

**其实，问题的关键在于交换操作。**

`suffixMin` 数组正确工作有个前提，就是 `nums` 数组不可变。如果 `nums[i]` 的值发生改变，那么所有 `suffixMin[0..i]` 存储的最小值就失效了，需要重新计算一次才行。

这个应该不难理解吧，比方说你的 `suffixMin[3] = 6`，意思是 `nums[3..]` 中的最小值是 6。如果你修改了 `nums[5] = 2`，那么 `suffixMin[0..5]` 的值都应该变成 2，而不再是 6 了。

选择排序中的交换操作，会导致 `nums` 中的元素位置发生变化，进而导致 `suffixMin` 数组失效，这才是问题的本质。

综上，所有尝试都是错误的，选择排序无法进行任何优化。

那么我们花了那么多时间，尝试了种种方法，最后啥名堂也没弄出来，是不是很失败？

不，我认为这些才是有效的思考，是真正能够帮助读者掌握算法思维的。

比如上面预算算 `suffixMin` 的方法是一种经典的算法思维，后文 [前缀和技巧](#) 就会用到。`nums` 数组变化导致预算算数组 `suffixMin` 失效也是一类经典的算法问题，后文 [线段树基础](#) 会解决这个问题。

在本站的教程中，我会经常展现这种思考过程。在后面讲到的排序算法中，你也可以思考一下，它们的本质上和选择排序有什么区别？凭什么它们就能把时间复杂度降到  $O(n^2)$  以下？

## 排序的稳定性

请你按照 [排序算法的几个关键指标](#) 中对排序稳定性的定义，分析一下这个算法是不是稳定排序？

如果这个算法不是稳定排序，那么是什么操作导致了它失去了排序稳定性呢？是否可以优化这个算法，使它成为稳定排序？请思考几分钟，然后再看我的理解。

**选择排序算法不是稳定排序。**

按照稳定排序的定义，相同元素的相对位置不会发生变化才能称为稳定排序。你举个简单的例子就看出这个算法不稳定了：

```
[2', 2''', 2'', 1]
```

这个例子中有多个重复元素 2，我分别用 `2'`、`2'''`、`2''` 来区别这三个元素。如果这个排序算法是稳定的，那么排序后的结果应该保持三个 2 的相对顺序：

```
[1, 2', 2''', 2'']
```

但实际上，你在脑子里跑一下这个算法就能想到，它第一次寻找最小值时，肯定会把元素 `2'` 和 `1` 交换，这一下就会打乱 2 之间的相对顺序了：

```
[1, 2''', 2'', 2']
```

是交换操作，使得选择排序失去了排序的稳定性。

有没有什么办法可以优化这个算法，使它成为稳定排序？现在时间复杂度都到  $O(n^2)$  了，属于排序算法里面最差的那一档，好歹咱也支棱起来，把自己搞成一个稳定排序，行不行？

你可以自己想一想这个问题，在后面的排序算法，我们会尝试解决这个问题。

# 拥有稳定性：冒泡排序

阅读本文前，你需要先学习：

- [选择排序所面临的问题](#)
- [数组的增删查改操作](#)

冒泡算法是对[选择排序](#)的一种优化，通过交换`nums[sortedIndex]`右侧的逆序对完成排序，是一种稳定排序算法。

你可以点开可视化面板，点击播放按钮，然后点击加速/减速按钮调节速度，即可直观感受冒泡排序的过程：

▶  [代码可视化动画](#) 

前文讲解了[选择排序](#)这种最简单直接的排序算法，其中分析了选择排序的几个待优化的问题：

- 1、选择排序算法是个不稳定排序算法，因为每次都要交换最小元素和当前元素的位置，这样可能会改变相同元素的相对位置。
- 2、选择排序的时间复杂度和初始数据的有序度完全没有关系，即便输入的是一个已经有序的数组，选择排序的时间复杂度依然是 $O(n^2)$ 。
- 3、选择排序的时间复杂度是 $O(n^2)$ ，具体的操作次数大概是 $n^2/2$ 次，常规的优化思路无法降低时间复杂度。

那么本文就围绕着选择排序的种种缺陷，看看能不能想办法帮它解决一下。

## 重获排序稳定性

前文分析过选择排序失去稳定性的原因，即每次都要交换最小元素(`nums[minIndex]`)和当前元素(`nums[sortedIndex]`)，这样可能会改变相同元素的相对位置。

你仔细思考这个交换过程，其实它的目标是把`nums[minIndex]`放到到`nums[sortedIndex]`，至于`nums[sortedIndex]`这个位置的元素应该去哪里，它并不关心。之所以它用交换操作，只是因为交换操作最简单，不需要涉及数据搬移。

在交换过程中，把`nums[minIndex]`放到到`nums[sortedIndex]`的操作是不影响相同元素的相对顺序的：

```
[2, 2', 2'', 1, 1']  
  ^            ^  
[1, 2', 2'', _, 1']  
  ^            ^  
sortedIndex minIndex
```

真正破坏稳定性的，是让`nums[sortedIndex]`去`nums[minIndex]`的位置这一步：

```
[1, 2', 2'', 2, 1']  
  ^            ^
```

可以看到 `2`, `2'`, `2''` 这三个元素的相对顺序被打乱了。

所以优化的方向就在这里，你不要图省事儿直接把 `nums[sortedIndex]` 交换到 `nums[minIndex]`，而是模仿 在数组中部插入元素的操作，将 `nums[sortedIndex..minIndex]` 的元素整体向后移动一位，把 `nums[sortedIndex + 1]` 的位置空出来让 `nums[sortedIndex]` 这个元素去那里待着。

```
[2, 2', 2'', 1, 1']  
^ ^  
[1, 2', 2'', _, 1']  
^ _  
[1, _, 2', 2'', 1']  
^ ^  
[1, 2, 2', 2'', 1']  
^ ^  
sortedIndex minIndex
```

可以看到，这次 `2`, `2'`, `2''` 和 `1`, `1'` 的相对顺序都没有发生改变，选择排序就变成了稳定排序了。

具体代码如下，只需要把 [选择排序](#) 代码中交换元素的部分换一下即可：

```
// 对选择排序进行第一波优化，获得了稳定性  
void sort(int[] nums) {  
    int n = nums.length;  
    int sortedIndex = 0;  
    while (sortedIndex < n) {  
        // 在未排序部分中找到最小值 nums[minIndex]  
        int minIndex = sortedIndex;  
        for (int i = sortedIndex + 1; i < n; i++) {  
            if (nums[i] < nums[minIndex]) {  
                minIndex = i;  
            }  
        }  
  
        // 交换最小值和 sortedIndex 处的元素  
        // int tmp = nums[sortedIndex];  
        // nums[sortedIndex] = nums[minIndex];  
        // nums[minIndex] = tmp;  
  
        // 优化：将 nums[minIndex] 插入到 nums[sortedIndex] 的位置  
        // 将 nums[sortedIndex..minIndex] 的元素整体向后移动一位  
        int minValue = nums[minIndex];  
        // 数组搬移数据的操作  
        for (int i = minIndex; i > sortedIndex; i--) {  
            nums[i] = nums[i - 1];  
        }  
        nums[sortedIndex] = minValue;  
  
        sortedIndex++;  
    }  
}
```

你可以拿着这个算法去力扣第 912 题「排序数组」提交一下，虽然最后会超时无法通过，但是可以证明这个算法的正确性是没有问题的。

这个算法对比标准的选择排序，虽然拥有了稳定性，但是执行效率会下降，虽然从 Big O 表示法的角度来看，两层嵌套循环的时间复杂度还是  $O(n^2)$ ，但毕竟又加了一个 for 循环，实际执行次数肯定会大于标准选择排序的  $n^2/2$  次。

下面我们再来看看，能不能进一步优化，避免这个额外的 for 循环。

## 优化时间复杂度

仔细观察上面的算法代码，while 循环内部主要做了两件事：

- 1、第一个 for 循环寻找 `nums[sortedIndex..]` 中的最小值。
- 2、第二个 for 循环将这个最小值插入到 `nums[sortedIndex]` 的位置。

那么我们能否将这两个步骤合在一起呢？具体来说，你在寻找 `nums[sortedIndex..]` 中的最小值的时候能不能做些力所能及的事情，能不能做到找到最小值后，它就已经被放在正确的位置上，不需要再进行数据搬移了？

答案是可以的，看我操作：

```
// 对选择排序进行第二波优化，获得稳定性的同时避免额外的 for 循环
// 这个算法有另一个名字，叫做冒泡排序
void sort(int[] nums) {
    int n = nums.length;
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 寻找 nums[sortedIndex..] 中的最小值
        // 同时将这个最小值逐步移动到 nums[sortedIndex] 的位置
        for (int i = n - 1; i > sortedIndex; i--) {
            if (nums[i] < nums[i - 1]) {
                // swap(nums[i], nums[i - 1])
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
            }
        }
        sortedIndex++;
    }
}
```

---

### ▶ 🔍 代码可视化动画

这个优化就比较巧妙了，倒序遍历 `nums[sortedIndex..]`，如果发现逆序对儿，就交换顺序，这样最小值就会逐步移动到 `nums[sortedIndex]` 的位置。

而且由于我们只交换相邻的逆序对儿，不会去碰值相同的元素，所以这个算法是稳定排序。

这个算法的时间复杂度依然是  $O(n^2)$ ，实际执行次数和选择排序类似，也是一个等差数列求和，大约是  $n^2/2$  次。

这个算法的名字叫做**冒泡排序**，因为它的执行过程就像从数组尾部向头部冒出水泡，每次都会将最小值顶到正确的位置。

## 提前终止算法

上面说到选择排序的一个问题是，其时间复杂度和初始数据的有序度完全没有关系，即便输入的数组已经有序，选择排序依然会执行  $O(n^2)$  次操作。

在上面的一些列优化之后，就可以解决这个问题了，具体看代码：

```
// 进一步优化，数组有序时提前终止算法
void sort(int[] nums) {
    int n = nums.length;
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 加一个布尔变量，记录是否进行过交换操作
        boolean swapped = false;
        for (int i = n - 1; i > sortedIndex; i--) {
            if (nums[i] < nums[i - 1]) {
                // swap(nums[i], nums[i - 1])
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
                swapped = true;
            }
        }
        // 如果一次交换操作都没有进行，说明数组已经有序，可以提前终止算法
        if (!swapped) {
            break;
        }
        sortedIndex++;
    }
}
```

---

▶      代码可视化动画 

---

好了，以上就是针对选择排序的一系列优化，最终使它拥有了排序稳定性，并支持在数组有序时提前终止算法。唯一的遗憾是，时间复杂度依然是  $O(n^2)$ ，并没有降低。

下面我们继续探讨，看看还有什么方法能够改进选择排序。

# 运用逆向思维：插入排序

阅读本文前，你需要先学习：

- 选择排序所面临的问题
- 数组的增删查改操作
- 拥有稳定性：冒泡排序

插入排序是基于[选择排序](#)的一种优化，将 `nums[sortedIndex]` 插入到左侧的有序数组中。对于有序度较高的数组，插入排序的效率比较高。

你可以点开可视化面板，点击播放按钮，然后点击加速/减速按钮调节速度，即可直观感受插入排序的过程：

## ▶ 代码可视化动画

前文[选择排序所面临的问题](#)中分析了选择排序遇到的几个问题，然后逐步优化写出了[冒泡排序](#)，使得排序算法具有稳定性，且能够在输入数组的有序度较高时提前终止，提升效率。

回顾一下，冒泡排序的关键点在于对下面这段代码的优化：

```
// 对选择排序进行第一波优化，获得了稳定性
void sort(int[] nums) {
    int n = nums.length;
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 在未排序部分中找到最小值 nums[minIndex]
        int minIndex = sortedIndex;
        for (int i = sortedIndex + 1; i < n; i++) {
            if (nums[i] < nums[minIndex]) {
                minIndex = i;
            }
        }

        // 将 nums[minIndex] 插入到 nums[sortedIndex] 的位置
        // 将 nums[sortedIndex..minIndex] 的元素整体向后移动一位
        int minValue = nums[minIndex];
        // 数组搬移数据的操作
        for (int i = minIndex; i > sortedIndex; i--) {
            // swap(nums[i], nums[i - 1])
            nums[i] = nums[i - 1];
        }
        nums[sortedIndex] = minValue;

        sortedIndex++;
    }
}
```

## ▶ 代码可视化动画

为了避免 while 内存在两个 for 循环，我们使用了一种类似冒泡的方式逐步交换 `nums[sortedIndex..]` 中的逆序对，将最小值换到 `nums[sortedIndex]` 的位置。

好的，先停在这一步，让我们忘记冒泡排序的优化方法，你来思考一下，是否还有其他方法能够优化上述代码，把 while 循环中的两个 for 循环优化成一个 for 循环？

## 反向思维

上面的算法思路是：在 `nums[sortedIndex..]` 中找到最小值，然后将其插入到 `nums[sortedIndex]` 的位置。

那么我们能不能反过来想，在 `nums[0..sortedIndex-1]` 这个部分有序的数组中，找到 `nums[sortedIndex]` 应该插入的位置，然后进行插入呢？

当年我思考如何对插入排序进行优化时，是想到过这个思路的，因为我想利用数组的有序性呀：既然 `nums[0..sortedIndex-1]` 这部分是已经排好序的，那么我就可以用二分搜索来寻找 `nums[sortedIndex]` 应该插入的位置。

这样一来，上述代码中的内层第一个 for 循环，我可以给他优化成对数级别的复杂度。

但是仔细想想，用二分搜索好像是多此一举的。因为就算我用二分搜索找到了 `nums[sortedIndex]` 应该插入的位置，我还是需要搬移元素进行插入，那还不如一边遍历一遍交换元素的方法简单高效呢：

```
// 对选择排序进一步优化，想左侧有序数组中插入元素
// 这个算法有另一个名字，叫做插入排序
void sort(int[] nums) {
    int n = nums.length;
    // 维护 [0, sortedIndex) 是有序数组
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 将 nums[sortedIndex] 插入到有序数组 [0, sortedIndex) 中
        for (int i = sortedIndex; i > 0; i--) {
            if (nums[i] < nums[i - 1]) {
                // swap(nums[i], nums[i - 1])
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
            } else {
                break;
            }
        }
        sortedIndex++;
    }
}
```

### ▶ 🎬 代码可视化动画🎬

这个算法的名字叫做**插入排序**，它的执行过程就像是打扑克牌时，将新抓到的牌插入到手中已经排好序的牌中。

插入排序的空间复杂度是  $O(1)$ ，是原地排序算法。时间复杂度是  $O(n^2)$ ，具体的操作次数和选择排序类似，是一个等差数列求和，大约是  $n^2/2$  次。

插入排序是一种稳定排序，因为只有在 `nums[i] < nums[i - 1]` 的情况下才会交换元素，所以相同元素的相对位置不会发生改变。

## 初始有序度越高，效率越高

显然，插入排序的效率和输入数组的有序度有很大关系，可以举极端例子来理解：

如果输入数组已经有序，或者仅有个别元素逆序，那么插入排序的内层 for 循环几乎不需要执行元素交换，所以时间复杂度接近  $O(n)$ 。

如果输入的数组是完全逆序的，那么插入排序的效率就会很低，内层 for 循环每次都要对 `nums[0..sortedIndex-1]` 的所有元素进行交换，算法的总时间复杂度就接近  $O(n^2)$ 。

如果对比插入排序和冒泡排序，**插入排序的综合性能应该要高于冒泡排序。**

直观地说，插入排序的内层 for 循环，只需要对 `sortedIndex` 左侧 `nums[0..sortedIndex-1]` 这部分有序数组进行遍历和元素交换，大部分非极端情况下，可能不需要遍历完 `nums[0..sortedIndex-1]` 的所有元素；而冒泡排序的内层 for 循环，每次都需要遍历 `sortedIndex` 右侧 `nums[sortedIndex..]` 的所有元素。

所以冒泡排序的操作数大约是  $n^2/2$ ，而插入排序的操作数会小于  $n^2/2$ 。

你可以把插入排序的代码拿去力扣第 912 题「排序数组」提交，它最终依然会超时，但可以说明算法代码的逻辑是正确的。之后的文章我们继续探讨如何对排序算法进行优化。

# 突破 $O(N^2)$ : 希尔排序

阅读本文前，你需要先学习：

- 选择排序所面临的问题
- 运用逆向思维：插入排序

希尔排序是基于 [插入排序](#) 的简单改进，通过预处理增加数组的局部有序性，突破了插入排序的  $O(N^2)$  时间复杂度。

你可以点开可视化面板，点击播放按钮，然后点击加速/减速按钮调节速度，即可直观感受希尔排序的过程：

▶  [代码可视化动画](#) 

必须承认，希尔排序的思路很难想到，我是在《算法 4》第一次了解到这个算法，然后惊叹于这个算法的简单优化竟然能给插入排序带来如此大的提升。

首先我们要明确一个  **$h$  有序数组** 的概念。

## **$h$ 有序数组**

一个数组是  $h$  有序的，是指这个数组中任意间隔为  $h$ （或者说间隔元素的个数为  $h-1$ ）的元素都是有序的。

这个概念用文字不好描述清楚，直接看个例子吧。比方说  $h=3$  时，一个  $3$  有序数组是这样的：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 妙用二叉树前序位置：快速排序

阅读本文前，你需要先学习：

- [选择排序所面临的问题](#)
- [二叉树的遍历](#)

快速排序的核心思路需要结合 [二叉树的前序遍历](#) 来理解：在二叉树遍历的前序位置将一个元素排好位置，然后递归地将剩下的元素排好位置。

你可以点开这个可视化面板，点击全屏按钮，然后多次点击 `let p = ...` 这部分代码，即可直观地看到快排的递归过程和排序效果：

## ▶ 代码可视化动画

上来这一句总结是不是就把初学者听懵了？数组排序算法怎么扯到二叉树上了？

所以说，计算机思维和人类思维是不一样的。

正常人要排序数组，一般就是维护一个 `sortedIndex`，保持 `[0, sortedIndex]` 有序，逐步右移 `sortedIndex`，直到整个数组有序。这中间历经种种坎坷，逢山开路遇水搭桥，正如我们前面讲的 [选择排序](#)、[冒泡排序](#)、[插入排序](#)、[希尔排序](#)。

但是越是效率高的算法，离计算机思维越近，未经训练的人就越难理解。学过前面几种基础排序算法，现在你应该可以感觉到这一点了，容易理解和推导的排序算法复杂度全都是  $O(N^2)$ ，而突破  $O(N^2)$  的排序算法，都感觉不是人类能想出来的。

哪个人要是张嘴就说：排序数组简单啊，只要把一个元素排好序，然后把剩下元素排好序，就能把整个数组排好序了。那只能说这个人可能是三体人潜伏在地球的特务：）

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 妙用二叉树后序位置：归并排序

阅读本文前，你需要先学习：

- 二叉树的遍历
- 妙用二叉树前序位置：快速排序

归并排序的核心思路需要结合[二叉树的后序遍历](#)来理解：先利用递归把左右两半子数组排好序，然后在二叉树的后序位置合并两个有序数组。

你可以点开这个可视化面板，点击全屏按钮，然后多次点击`merge(arr, lo, mid, hi)`这一行代码，即可直观地看到归并排序的递归过程和排序效果：

▶  [代码可视化动画](#) 

考虑到这里是基础知识章节，我只会讲一下归并排序的整体思路，具体的代码实现和算法运用会安排在二叉树章节后面的[归并排序详解及运用](#)里，不建议初学者现在去看。

因为归并排序算法需要熟练掌握递归思维，且需要用到[双指针技巧](#)来合并两个有序数组，所以建议初学者按照本站目录顺序学习，到时候理解归并排序的代码会比较轻松。

## 归并排序核心思路

开头的这句总结虽然也比较抽象，但是有上一章[快速排序核心思路](#)的铺垫，你应该有点感觉。

我用快速排序的思路来对比一下，你就能直观感受到它俩的区别了：

前文快速排序的思路是，先把一个元素放到正确的位置（排好序），然后将这个元素左右两边剩下的元素利用递归分别排好序，最终整个数组就排好序了。代码框架如下：

本文为 [labuladong.online](#) 网站会员内容，请[点这里](#)查看。

# 二叉堆结构的运用：堆排序

阅读本文前，你需要先学习：

- [二叉堆基础](#)
- [二叉堆实现优先级队列](#)

堆排序是从 [二叉堆结构](#) 衍生出来的排序算法，复杂度为  $O(N \log N)$ 。堆排序主要分两步，第一步是在待排序数组上原地创建二叉堆（Heapify），然后进行原地排序（Sort）。

你可以打开下方可视化面板，点击跳转到 `let tree = ...` 这部分代码可以看到数组被抽象成完全二叉树；不断点击 `Heap.swim` 这部分代码，可以看到原地建堆的过程；点击 `Heap.sink` 这部分代码，可以看到原地排序的过程。

学习堆排序算法必须掌握二叉堆结构原理，否则可能完全无法理解排序过程。

## ▶ ★ 代码可视化动画★

前文 [二叉堆基础](#) 介绍过二叉堆结构，[二叉堆实现优先级队列](#) 利用二叉堆结构实现了一个 `SimpleMinPQ` 优先级队列，插入队列的元素会按照从小到大的顺序取出。

本文将介绍堆排序算法，它是基于二叉堆性质衍生出来的一种全新排序算法，非常优雅和高效。

首先，我要复述一下二叉堆实现优先级队列的几个关键原理，如果你有任何不理解的地方，务必回去复习前文，否则无法理解堆排序。

1、二叉堆（优先级队列）底层是用数组实现的，但是逻辑上是一棵完全二叉树，主要依靠 `swim`, `sink` 方法来维护堆的性质。

2、优先级队列可以分为小顶堆和大顶堆，小顶堆会将整个堆中最小的元素维护在堆顶，大顶堆会将整个堆中最大的元素维护在堆顶。

3、优先级队列插入元素时，首先把元素追加到二叉堆底部，然后调用 `swim` 方法把该元素上浮到合适的位置，时间复杂度是  $O(\log N)$ 。

4、优先级队列删除堆顶元素时，首先把堆底的最后一个元素交换到堆顶作为新的堆顶元素，然后调用 `sink` 方法把这个新的堆顶元素下沉到合适的位置，时间复杂度是  $O(\log N)$ 。

那么最简单的堆排序算法思路就是直接利用优先级队列，把所有元素塞到优先级队列里面，然后再取出来，不就完成排序了吗？

```
// 直接利用优先级队列对数组从小到大排序
void sort(int[] nums) {
    // 创建一个从小到大排序元素的小顶堆
    SimpleMinPQ pq = new SimpleMinPQ(nums.length);
    // 先把所有元素插入到优先级队列中
    for (int num : nums) {
        // push 操作会自动构建二叉堆，时间复杂度为 O(logN)
        pq.push(num);
    }
    // 再把所有元素取出来，就是从小到大排序的结果
}
```

```
for (int i = 0; i < nums.length; i++) {
    // pop 操作从堆顶弹出二叉堆堆中最小的元素，时间复杂度为 O(logN)
    nums[i] = pq.pop();
}
```

因为优先级队列的 `push`, `pop` 方法的复杂度都是  $O(\log N)$ , 所以整个排序的时间复杂度是  $O(N \log N)$ , 其中  $N$  是输入数组的长度。

这个思路可以得到正确的排序结果, 但空间复杂度是  $O(N)$ , 因为我们创建的这个优先级队列是一个额外的数据结构, 它的底层使用了一个数组来存储元素。

所以, 堆排序要解决的问题是, 不要使用额外的辅助空间, 直接在原数组上进行 `sink`, `swim` 操作, 在  $O(N \log N)$  的时间内完成排序。

1、原地建堆 (Heapify) : 直接把待排序数组原地变成一个二叉堆。

2、排序 (Sort) : 将元素不断地从堆中取出, 最终得到有序的结果。

你不妨自己思考几分钟, 对比一下优先级队列增删元素的过程, 其实利用 `swim`, `sink` 方法原地实现这两步并不难, 应该可以独立思考出来。

在具体讲解堆排序代码实现之前, 我先把二叉堆的 `swim`, `sink` 方法和配套的工具函数写出来, 因为后文我会带你逐步优化堆排序的代码, 就不重复实现这些函数了。

这些函数就是从 [二叉堆实现优先级队列](#) 中的优先级队列实现里抠出来的, 把数组作为函数参数传入, 其他的逻辑完全一样:

本文为 [labuladong.online](#) 网站会员内容, 请 [点这里](#) 查看。

# 全新的排序原理：计数排序

阅读本文前，你需要先学习：

- 排序算法的关键指标
- 选择排序所面临的问题

计数排序的原理比较简单：统计每种元素出现的次数，进而推算出每个元素在排序后数组中的索引位置，最终完成排序。

计数排序的时间和空间复杂度都是  $O(n + \max - \min)$ ，其中  $n$  是待排序数组长度， $\max - \min$  是待排序数组的元素范围。

这是选择排序的可视化面板，你可以点击 `sorted[count[index] - 1] = nums[i]` 这部分代码，即可看到有序数组形成的过程：

## ▶ 🎃 代码可视化动画🎃

比方说，输入一个 `nums` 数组，我统计出其中有 2 个元素 `1`，1 个元素 `3`，3 个元素 `6`，那么只要我在数组中依次填入 2 个 `1`，1 个 `3`，3 个 `6`，就能得到排序结果 `[1, 1, 3, 6, 6, 6]`。

我们做一道简单的题目就能明白了，来看力扣第 75 题「颜色分类」：

### ▼ 75. 颜色分类 [Leetcode](#) | [力扣](#)

给定一个包含红色、白色和蓝色、共  $n$  个元素的数组 `nums`，**原地** 对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 `0`、`1` 和 `2` 分别表示红色、白色和蓝色。

必须在不使用库内置的 `sort` 函数的情况下解决这个问题。

示例 1：

```
输入: nums = [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]
```

示例 2：

```
输入: nums = [2,0,1]
输出: [0,1,2]
```

提示：

- `n == nums.length`
- `1 <= n <= 300`
- `nums[i]` 为 `0`、`1` 或 `2`

## 进阶：

- 你能想出一个仅使用常数空间的一趟扫描算法吗？

这道题有多种思路，最优解法是用双指针技巧仅遍历一次数组完成排序，我会在[数组双指针技巧习题](#)中介绍。这里我们用计数排序的思路来解决这个问题，说白了就是让你对数组排序，且这个数组里只有0、1、2三种元素。

我们可以创建一个大小为3的count数组，`count[0]`, `count[1]`, `count[2]`分别表示数组中0、1、2出现的次数。然后我们按照`count`数组的统计结果，依次填充原数组即可。

@tab:active java

```
class Solution {
    public void sortColors(int[] nums) {
        // 统计 0, 1, 2 出现的次数
        int[] count = new int[3];
        for (int element : nums) {
            count[element]++;
        }

        // 按照 count 数组的统计结果，依次填充原数组
        int index = 0;
        for (int element = 0; element < 3; element++) {
            for (int i = 0; i < count[element]; i++) {
                nums[index] = element;
                index++;
            }
        }
    }
}
```

@tab cpp

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        // 统计 0, 1, 2 出现的次数
        vector<int> count(3, 0);
        for (int element : nums) {
            count[element]++;
        }

        // 按照 count 数组的统计结果，依次填充原数组
        int index = 0;
        for (int element = 0; element < 3; element++) {
            for (int i = 0; i < count[element]; i++) {
                nums[index] = element;
                index++;
            }
        }
    }
};
```

@tab python

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        # 统计 0, 1, 2 出现的次数
        count = [0] * 3
        for element in nums:
            count[element] += 1

        # 按照 count 数组的统计结果, 依次填充原数组
        index = 0
        for element in range(3):
            for _ in range(count[element]):
                nums[index] = element
                index += 1
```

@tab go

```
func sortColors(nums []int) {
    // 统计 0, 1, 2 出现的次数
    count := make([]int, 3)
    for _, element := range nums {
        count[element]++
    }

    // 按照 count 数组的统计结果, 依次填充原数组
    index := 0
    for element := 0; element < 3; element++ {
        for i := 0; i < count[element]; i++ {
            nums[index] = element
            index++
        }
    }
}
```

@tab javascript

```
var sortColors = function(nums) {
    // 统计 0, 1, 2 出现的次数
    const count = [0, 0, 0];
    for (const element of nums) {
        count[element]++;
    }

    // 按照 count 数组的统计结果, 依次填充原数组
    let index = 0;
    for (let element = 0; element < 3; element++) {
        for (let i = 0; i < count[element]; i++) {
            nums[index] = element;
            index++;
        }
    }
};
```

这就是一个简单的计数排序算法，不过这个题目给的场景比较简单，只有 0, 1, 2 三种元素，下面我们给出一个更通用的计数排序算法。

## 通用的计数排序

虽然计数排序的原理简单，但是在通用的计数排序代码中，还是有一些编码技巧的。

我们从提出问题开始。计数排序需要把数组中的元素作为 `count` 数组的索引才能计数，那么我们可以提出如下疑问：

- 1、是不是说只有当 `nums` 数组中的元素都是非负整数的时候才能用计数排序呢？包含负数时如何排序？对自定义的类型如何排序？
- 2、根据计数排序的原理，我们仅关心某一个元素出现了多少次，而并不关心相同元素的相对位置，那么看起来计数排序是一个不稳定排序，对吗？
- 3、因为计数排序需要将元素的值作为 `count` 数组的索引，那么如果数组中的最大元素的值很大时，会不会导致 `count` 数组太大，空间复杂度过高？

下面我们来一步一步思考这些问题，尝试给出解法。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 博采众长：桶排序

阅读本文前，你需要先学习：

- [计数排序](#)

桶排序算法的核心思想分三步：

- 1、将待排序数组中的元素使用映射函数分配到若干个「桶」中。
- 2、对每个桶中的元素进行排序。
- 3、最后将这些排好序的桶进行合并，得到排序结果。

打开下面的可视化面板，多次点击 `buckets[index].push(num)` 这行代码，即可看到元素分配到不同桶中的过程；多次点击 `insertSort(curBucket)` 这一行代码，即可看到对每个桶进行排序的过程；多次点击 `nums[index++] = num` 这行代码，即可看到合并有序桶的过程。

## ▶ 🎃 代码可视化动画🎃

桶排序算法可能并不常见，但我个人感觉它的思想非常有意思，因为你在它的算法思想中同时看到前面讲的 [归并排序](#) 和 [计数排序](#) 的影子，而且。

如果你按顺序学习了前面的所有算法，就会感慨这些算法之间千丝万缕的联系。看看这一代代计算机大佬，就为了「排序」这一个需求，真所谓八仙过海各显神通，精妙的想法层出不穷，我们作为后辈，何不好好品味一下呢？

## 桶排序的关键点

言归正传，桶排序的思路真的很简单，就是先把待排序数组中的元素分配到若干个桶中，对每个桶中的元素分别进行排序，最后再把这些桶中的元素按顺序合并起来。

这个思路是不是有点像 [归并排序](#)？都是把大的数组分成小的数组进行排序，最后再合并起来。不过桶排序更加灵活，三个核心步骤中每一步都可以变化：

- 1、如何将待排序元素分配到桶中？你需要决定桶的数量，并提供一个映射函数。
- 2、如何对每个桶中的元素进行排序？理论上可以使用任意排序算法，或者模拟 [归并排序](#) 的思路，对每个桶递归地运行桶排序。
- 3、如何将排好序的桶合并起来？后面的章节会讲 [合并多个有序链表/数组](#) 的通用算法，但那个算法会用到 [二叉堆结构](#)，且复杂度为  $O(n \log k)$ ，这里显然不适用：

如果我都用上二叉堆了，还搞什么桶排序，直接上 [堆排序](#) 不就完事了，是吧？所以这一步合并操作的时间复杂度不能超过  $O(n)$ ，要做到这一点，就要合理设计分配元素的映射函数。

关于这三个问题，我想首先探讨其中第二个问题。不知道你有没有想过，为什么要把待排序数组分成若干个桶，然后再对每个桶进行排序？这样排序，和直接对整个待排序数组排序相比，真的有区别吗？

答案是，如果暂时不考虑合并有序桶的算法复杂度，那么分开排序当然要比整体排序效率高。

以最简单的 [选择排序](#) 为例，如果我直接对大小为  $n$  的数组进行选择排序，那么时间复杂度是  $O(n^2)$ 。

假设我们将待排序数组分成  $k$  个桶，对于每个桶使用选择排序，那么总的时间复杂度是大于  $O(n^2)$ ，还是小于  $O(n^2)$ ？

这其实是一个简单的数学题，假设有个正整数  $n$ ，且它可以分解为  $n = n_1 + n_2 + \dots + n_k$ ，那么  $n^2$  和  $n_1^2 + n_2^2 + \dots + n_k^2$  哪个更大？

有多种思路可以得到答案，我们来看这种几何的思路：

把这个  $n^2$  想象成一个正方形的面积，而  $n_1^2 + n_2^2 + \dots + n_k^2$  是这个大正方形的一条边上的若干小正方形的面积之和，这样就能直观的理解了，显然这些小正方形的面积没有整个正方形的面积大，所以  $n^2 >= n_1^2 + n_2^2 + \dots + n_k^2$ 。

由此可知，**分开排序的时间复杂度总和是小于整体排序的**，这就是保证桶排序算法的数学基础。

基于正方形面积的这个抽象，我们还可以更进一步。当  $k$  无限大， $n_1, n_2, \dots, n_k$  无限小时会怎样？

正方形那条边上的小正方形面值之和越来越小，最终会和那条边融为一体，也就是说  $n_1^2 + n_2^2 + \dots + n_k^2$  的值会无限接近  $n$ 。

以此观之，如果桶排序将待排序元素分配到尽可能多的桶中（ $k$  尽可能大），即每个桶至多只有一个元素时，桶排序就转化成了**计数排序**，其复杂度也将降低到  $O(n)$ 。

即便不能做到每个桶只有一个元素，只要  $k > 1$ ，桶排序的时间复杂度也会小于  $O(n^2)$ ， $k$  越大，时间复杂度越接近  $O(n)$ 。

反过来，如果取最小值  $k = 1$ ，那桶排序就完全退化成了选择排序，时间复杂度是  $O(n^2)$ 。

当然，上述分析都没有考虑合并有序桶的时间复杂度，不过只要能在  $O(n)$  的时间内进行合并，那么桶排序的总时间复杂度依然是小于  $O(n^2)$  的。

下面我将来探讨如何把待排序元素分配到桶中，以及如何合并有序桶，最后给出桶排序的几种代码实现。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 基数排序 (Radix Sort)

阅读本文前，你需要先学习：

- 排序算法的关键指标
- 计数排序

基数排序是 [计数排序](#) 算法的扩展，它的主要思路是对待排序元素的每一位依次进行计数排序。由于计数排序是稳定的，所以对每一位完成计数排序后，所有元素就完成了排序。

下面这个可视化面板展示了基数排序的过程：点击 `let maxLen = 1` 这一行代码，可以看到算法将数组元素都转化为了非负数；多次点击 `countSort(nums, k)` 这一行代码，对每一位执行计数排序；最后点击 `nums[i] -= offset` 这一行代码，将数组元素转化回原来的值，完成排序：

▶  [代码可视化动画](#) 

首先解释一下基数排序 (Radix Sort) 这个名词。

基数 (Radix) 其实就是进制的意思，比如十进制数的基数就是 10，二进制数的基数就是 2。看这个名字就知道这个排序算法肯定和数字的进制有关，进而可以推断，这个算法不是通用排序算法，待排序数据必须是整数，或者能够通过某种规则转化成整数，才能使用基数排序。

我发现网上的很多资料会把基数排序和桶排序放在一起，认为基数排序是桶排序的应用。

但我不认同这种看法，我认为基数排序是计数排序的扩展，可以用来解决计数排序空间复杂度过高的问题，和桶排序关系不大。

现在你已经学习过 [计数排序](#) 和 [桶排序](#) 了，在我介绍了基数排序的原理后，你也可以自己思考一下，它是到底是计数排序的扩展还是桶排序的扩展。

## 基数排序的原理

基数排序的原理很简单，比方说输入的数组都是三位数 `nums = [329, 457, 839, 439, 720, 355, 350]`，我们先按照个位数排序，然后按照十位数排序，然后按照百位数排序，最终就完成了整个数组的排序。

**这里面的关键在于，对每一位的排序都必须是稳定排序，否则最终结果就不对了。**

用这个 `nums` 数组为例，演示一下基数排序的过程，我把每个数字竖着写，方便查看每一位的排序效果。

首先是初始状态：

```
329  
457  
839  
439  
720  
355  
350
```

按照个位数进行稳定排序，得到：

```
720  
350  
355  
457  
329  
839  
439  
^
```

再按照十位数进行稳定排序，得到：

```
720  
329  
839  
439  
350  
355  
457  
^
```

最后按照百位数进行稳定排序，得到：

```
329  
350  
355  
439  
457  
720  
839  
^
```

上面就是基数排序的过程，在给出解法代码之前，先解答一些关于基数排序的问题：

- 1、为什么对每一位必须使用稳定排序？
- 2、使用什么稳定排序比较好，为什么？
- 3、如果待排序数组中的数字不全是三位数，怎么办？有负数怎么办？
- 4、必须按照从个位数到高位数的顺序进行排序吗？是否可以反过来，从高位数到个位数进行排序？

## 为什么对每一位必须使用稳定排序

举个很简单的例子：

```
56  
57
```

个位数已经排好序了，现在要对十位数排序。

十位数都是 5，稳定排序可以保证这两个 5 的顺序不变，最终的结果就是正确的；而如果使用不稳定排序，这两个 5 的顺序就可能被打乱，最终的结果就不对了。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 本章导读

---

## 本章内容

前面的快速入门章节讲解了十种排序算法，并手把手带你实现了算法题中常见的数据结构，不过并没有真正意义上开始刷题。

**从本章开始，就要开始以刷题为主了，你将学习若干核心算法框架，然后通过大量的习题来运用巩固这些框架。**

本章内容很硬核，整个网站的所有文章、习题都是基于本章总结的这些算法框架展开的。

就连我自己也会在刷题过程中从本章获得新的灵感，从而不断更新和优化本章的内容。

所以我建议读者在未来的学习过程中时常来回顾本章的内容，相信你每次回顾都会有新的收获。

## 写给初学者

初学者阅读本章时，一定会感到吃力，这很正常。

不过我依然坚持把本章内容作为刷题的开篇章节，希望大家都看一看。因为只要你对本章内容留个印象，就一定可以在未来的刷题过程中受益，我有这个自信。

**初学者学习本章的内容，不太理解的地方跳过就行了，切忌 DFS 死磕。**

也就是说，不要因为文中有其他文章的引用链接，你就要去把那篇文章也看了。不需要，不需要，不需要。

链接是为了帮助那些在本站有一定阅读基础的读者串联知识点用的，你现在用不到。

不用担心，后面章节中的文章和习题全部都是围绕本章内容展开的，相当于把本章抽象总结出来的框架思维手把手带你运用 100 遍，很多问题你那时候才能懂，再回来一看就豁然开朗了。

## 写给有一定基础的读者

框架虽好，但要多动手练习，才能真正把算法思维内化于心，收发自如。

经常有读者跟我反馈：有了框架思维，见到大部分题目基本都有思路，但真要是机试写代码，就不容易写对。这种读者的情况大多是看完了本站的文章，但自己动手写题写得少。

**如果你也有这样的困惑，可以花些时间学习本站目录中标记为「强化练习」的文章，这是我最近新增的习题集，这些习题可以完全套用本章总结的代码框架完成，建议你亲自动手做一做。**

最后，祝你早日攻克算法这个难关！

# 学习数据结构和算法的框架思维



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

阅读本文前，你需要先学习：

- 数组基础
- 链表基础
- 哈希表原理
- 二叉树基础
- 二叉树的遍历

本文会把很多算法进行抽象和归纳，所以会包含大量其他文章链接。

第一次阅读本文的读者不要 DFS 学习本文，遇到没学过的算法或不理解的地方请跳过，只要对本文所总结的理论有些印象即可。在学习本站后面的算法技巧时，你自然可以逐渐理解本文的精髓所在，日后回来重读本文，会有更深的体会。

这几年在我自己不断刷题、思考和写教程的过程中，我对算法的理解也是在逐渐加深，所以今天再写一篇，融合了多篇旧文，把我多年的经验和思考浓缩成 7000 字，分享给大家。

本文主要有两部分，一是谈我对数据结构和算法本质的理解，二是概括各种常用的算法。全文没有什么硬核的代码，都是我的经验之谈，也许没有多么高大上，但肯定能帮你少走弯路，更透彻地理解和掌握算法。

种种数据结构，皆为数组（顺序存储）和链表（链式存储）的变换。

数据结构的关键点在于遍历和访问，即增删查改等基本操作。

种种算法，皆为穷举。

穷举的关键点在于无遗漏和无冗余。熟练掌握算法框架，可以做到无遗漏；充分利用信息，可以做到无冗余。

真正理解上面几句话，不仅本文 7000 字都不用看了，乃至本站的几十篇算法教程和 500 道习题都不用做了。

如果不理解，那么我就用下面的几千字，以及后面的几十篇文章和 500 道习题，来阐明上述的两句总结。大家在学习的时候，时刻品味这两句话，会大幅提高学习效率。

## 数据结构的存储方式

数据结构的存储方式只有两种：数组（顺序存储）和链表（链式存储）。

这句话怎么理解，不是还有哈希表、栈、队列、堆、树、图等等各种数据结构吗？

我们分析问题，一定要有递归的思想，自顶向下，从抽象到具体。你上来就列出这么多，那些都属于上层建筑，而数组和链表才是结构基础。因为那些多样化的数据结构，究其源头，都是在链表或者数组上的特殊操作，API 不同而已。

比如说 **队列、栈** 这两种数据结构既可以使用链表也可以使用数组实现。用数组实现，就要处理扩容缩容的问题；用链表实现，没有这个问题，但需要更多的内存空间存储节点指针。

**图结构** 的两种存储方式，邻接表就是链表，邻接矩阵就是二维数组。邻接矩阵判断连通性迅速，并可以进行矩阵运算解决一些问题，但是如果图比较稀疏的话很耗费空间。邻接表比较节省空间，但是很多操作的效率上肯定比不过邻接矩阵。

**哈希表** 就是通过散列函数把键映射到一个大数组里。而且对于解决散列冲突的方法，**拉链法** 需要链表特性，操作简单，但需要额外的空间存储指针；**线性探查法** 需要数组特性，以便连续寻址，不需要指针的存储空间，但操作稍微复杂些。

**树结构**，用数组实现就是「堆」，因为「堆」是一个完全二叉树，用数组存储不需要节点指针，操作也比较简单，经典应用有 **二叉堆**；用链表实现就是很常见的那种「树」，因为不一定是完全二叉树，所以不适合用数组存储。为此，在这种链表「树」结构之上，又衍生出各种巧妙的设计，比如 **二叉搜索树**、AVL 树、**红黑树**、**区间树**、B 树等等，以应对不同的问题。

综上，数据结构种类很多，甚至你也可以发明自己的数据结构，但是底层存储无非数组或者链表，二者的优缺点如下：

**\*\*数组\*\***由于是紧凑连续存储，可以随机访问，通过索引快速找到对应元素，而且相对节约存储空间。但正因为连续存储，内存空间必须一次性分配够，所以说数组如果要扩容，需要重新分配一块更大的空间，再把数据全部复制过去，时间复杂度  $O(N)$ ；而且你如果想在数组中间进行插入和删除，每次必须搬移后面的所有数据以保持连续，时间复杂度  $O(N)$ 。

**\*\*链表\*\***因为元素不连续，而是靠指针指向下一个元素的位置，所以不存在数组的扩容问题；如果知道某一元素的前驱和后驱，操作指针即可删除该元素或者插入新元素，时间复杂度  $O(1)$ 。但是正因为存储空间不连续，你无法根据一个索引算出对应元素的地址，所以不能随机访问；而且由于每个元素必须存储指向前后元素位置的指针，会消耗相对更多的储存空间。

## 数据结构的基本操作

对于任何数据结构，其基本操作无非遍历 + 访问，再具体一点就是：增删查改。

数据结构种类很多，但它们存在的目的都是在不同的应用场景，尽可能高效地增删查改，这就是数据结构的使命。

如何遍历 + 访问？我们仍然从最高层来看，各种数据结构的遍历 + 访问无非两种形式：线性的和非线性的。

线性就是 `for/while` 迭代为代表，非线性就是递归为代表。再具体一步，无非以下几种框架：

数组遍历框架，典型的线性迭代结构：

```
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        // 迭代访问 arr[i]
    }
}
```

链表遍历框架，兼具迭代和递归结构：

```
// 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);
}
```

二叉树遍历框架，典型的非线性递归遍历结构：

```
// 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);
}
```

你看二叉树的递归遍历方式和链表的递归遍历方式，相似不？再看看二叉树结构和单链表结构，相似不？如果再多几条叉，N 叉树你会不会遍历？

二叉树框架可以扩展为 N 叉树的遍历框架：

```
// 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);
}
```

N 叉树的遍历又可以扩展为图的遍历，因为图就是好几 N 叉棵树的结合体。你说图是可能出现环的？这个很好办，用个布尔数组 `visited` 做标记就行了，[图结构遍历](#) 中有具体讲解。

所谓框架，就是套路。不管增删查改，这些代码都是永远无法脱离的结构，你可以把这个结构作为大纲，根据具体问题在框架上添加代码就行了。

## 算法的本质

如果要让我一句话总结，我想说算法的本质就是「穷举」。

这么说肯定有人要反驳了，真的所有算法问题的本质都是穷举吗？没有例外吗？

例外肯定是有，比如[一行代码就能解决的算法题](#)，这些题目类似脑筋急转弯，都是通过观察，发现规律，然后找到最优解法，不过这类算法问题较少，不必特别纠结。再比如，密码学算法、机器学习算法，它们的本质确实不是穷举，而是数

学原理的编程实现，所以这类算法的本质是数学，不在我们所探讨的「数据结构和算法」的范畴之内。

顺便强调下，「算法工程师」做的这个「算法」，和「数据结构与算法」中的这个「算法」完全是两码事，免得一些初学读者误解。

对前者来说，重点在数学建模和调参经验，计算机真就只是拿来做计算的工具而已；而后的重点是计算机思维，需要你能够站在计算机的视角，抽象、化简实际问题，然后用合理的数据结构去解决问题。

所以，你千万别以为学好了数据结构和算法就能去做算法工程师，也不要以为只要不做算法工程师就不需要学习数据结构和算法。

坦白说，大部分开发岗位工作中都是基于现成的开发框架做事，不怎么会碰到底层数据结构和算法相关的问题，但另一个事实是，只要你想找技术相关的岗位，数据结构和算法的考察是绕不开的，因为这块知识点是公认的程序员基本功。

为了区分，不妨称算法工程师研究的算法为「数学算法」，称刷题面试的算法为「计算机算法」，我写的内容主要聚焦的是「计算机算法」。

这样解释应该很清楚了吧，我猜大部分人的目标是通过算法笔试，找一份开发岗位的工作，所以你真的不需要有多少数学基础，只要学会用计算机思维解决问题就够了。

其实计算机思维也没什么高端的，你想想计算机的特点是啥？不就是快嘛，你的脑回路一秒只能转一圈，人家 CPU 转几万圈无压力。所以计算机解决问题的方式大道至简，就是穷举。

我记得自己刚入门的时候，也觉得计算机算法是一个很高大上的东西，每见到一道题，就想着能不能推导出一个什么数学公式，啪的一下就能把答案算出来。

比如你和一个没学过计算机算法的人说你写了个计算排列组合的算法，他大概以为你发明了一个公式，可以直接算出所有排列组合。但实际上呢？没什么高大上的公式，我会在[回溯算法秒杀排列组合子集问题](#)讲解，其实就是把排列组合的所有可能抽象成一棵多叉树结构，然后你写代码去遍历这棵树，把所有的结果收集起来罢了。这有啥神奇的？

对计算机算法的误解也许是以前学数学留下的「后遗症」，数学题一般都是你仔细观察，找几何关系，列方程，然后算出答案。如果说你需要进行大规模穷举来寻找答案，那大概率是你的解题思路出问题了。

而计算机解决问题的思维恰恰相反：有没有什么数学公式就交给你们人类去推导吧，如果能找到一些巧妙的定理那最好，但如果找不到，那就穷举呗，反正只要复杂度允许，没有什么答案是穷举不出来的。理论上讲只要不断随机打乱一个数组，总有一天能得到有序的结果呢！当然，这绝不是一个好算法，因为鬼知道它要运行多久才有结果。

技术岗笔试面试考的那些算法题，求个最大值最小值什么的，你怎么求？把所有可行解穷举出来就能找到最值了呗，说白了不就这么点事儿么。

## 穷举的难点

但是，你千万不要觉得穷举这个事儿很简单，穷举有两个关键难点：**无遗漏、无冗余**。

遗漏，会直接导致答案出错，比如让你求最小值，你穷举时恰好把那个最小值漏掉了，这不就错了嘛。

冗余，会拖慢算法的运行速度，比如你的代码把完全相同的计算流程重复了十遍，那你的算法不就慢了十倍么，就有可能超过判题平台的时间限制。

**为什么会遗漏？**因为你对算法框架掌握不到位，不知道正确的穷举代码。

**为什么会冗余？**因为你没有充分利用信息。

所以，当你看到一道算法题，可以从这两个维度去思考：

**1、如何穷举？**即无遗漏地穷举所有可能解。

## 2、如何聪明地穷举？即避免所有冗余的计算，消耗尽可能少的资源求出答案。

不同类型的题目，难点是不同的，有的题目难在「如何穷举」，有的题目难在「如何聪明地穷举」。

什么算法的难点在「如何穷举」呢？一般是递归类问题，比方说回溯算法、动态规划系列算法。

先说回溯算法，就拿我们高中学过的排列组合问题举例，我们当时都可以找到规律在草稿纸上推导排列组合：根据第一位可能的选择，先固定第一位，然后看第二位有哪些可能的选择，然后固定第二位... 以此类推，但如果未经训练，你很难用代码来穷举所有排列组合，因为你很难把这个手动穷举的过程抽象成程序化的规律。

首先，你要把排列组合问题抽象成一棵树，其次你要精确地使用代码遍历这棵树的所有节点，不能漏不能多，才能写出正确的代码。在后面的章节中，我会先介绍[回溯算法核心框架](#)，然后在[回溯算法解决子集排列组合问题](#)一次性解决所有子集排列组合问题。

动态规划比回溯算法更难一点。它俩本质上都是穷举，但思考模式不同，回溯算法是「遍历」的思维，而动态规划是「分解问题」的思维。

我都不用举正儿八经的例子，就比方说，你看那棵树，回答我，树上有多少片叶子？

你如何穷举？顺着树枝去一片片数么？当然也可以的，但这是遍历的思维模式，胜似你手动推导排列组合的过程，属于回溯算法的范畴

**如果你具备分解问题的思维模式，你应该告诉我：树上只有一片叶子，和剩下的叶子。**

听到这个回答，就知道是个算法高手。

还有不开窍的小同学追问，那剩下的叶子有多少呢？答曰，只有一片，和剩下的叶子。不要再往下问了，只能说，谜底就在谜面上，到了那个时候，你自然知道剩多少了。

所以你知道为啥我说动态规划这类问题的难点在于「如何穷举」了吧？一个脑瓜正常的人，本来就不会用这种奇怪的思维方式来思考问题，但这种思维结合计算机就是杀手锏，所以你要练，练好了，随心所欲写算法，咋写都是对的。

我在[动态规划核心框架](#)阐述了动态规划系列问题的解题过程，无非就是先写出暴力穷举解法（状态转移方程），加个备忘录就成自顶向下的递归解法了，再改一改就成自底向上的递推迭代解法了，[动态规划的降维打击](#)里也讲过如何利用空间压缩技巧优化动态规划算法的空间复杂度。

其中加备忘录、空间压缩技巧都是「如何聪明地穷举」的范畴，套路固定，不是难点。你亲自去做动态规划的题目就会发现，自己根本想不出状态转移方程，即第一步的暴力解法都写不出来，所以说找状态转移方程（如何穷举）才是难点。

我专门写了[动态规划设计方法：数学归纳法](#)这篇文章，告诉你穷举的核心是数学归纳法，明确函数的定义，分解问题，然后利用这个定义递归求解子问题。

什么算法的难点在「如何聪明地穷举」呢？一些耳熟能详的非递归算法技巧，都可以归在这一类。

最简单的例子，比方说让你在有序数组中寻找一个元素，用一个for循环暴力穷举谁都会，但[二分搜索算法](#)就是更聪明的穷举方式，拥有更好的时间复杂度。

还有后文[Union Find 并查集算法详解](#)告诉你一种高效计算连通分量的技巧，理论上说，想判断图中的两个节点是否连通，我用DFS/BFS暴力搜索（穷举）肯定可以做到，但人家Union Find算法硬是用数组模拟树结构，给你把连通性相关的操作复杂度给干到\$O(1)\$了。

这就属于聪明地穷举，大佬们把这些技巧发明出来，你学过就会用，没学过恐怕很难想出这种思路。

再比如贪心算法技巧，前文[当老司机学会贪心算法](#)就告诉你，所谓贪心算法就是在题目中发现一些规律（专业点叫贪心选择性质），使得你不用完整穷举所有解就可以得出答案。

人家动态规划好歹是无冗余地穷举所有解，然后找一个最值，你贪心算法可好，都不用穷举所有解就可以找到答案，所以前文[贪心算法解决跳跃游戏](#)中贪心算法的效率比动态规划还高。当然，并不是所有问题都存在贪心选择性质让你投机取巧，所以全量穷举虽然朴实无华且枯燥，但真的是任何情况下都可以用的。

下面我概括性地列举一些常见的算法技巧，供大家学习参考。

## 数组/单链表系列算法

**单链表常考的技巧就是双指针，属于「如何聪明地穷举」这一类，[单链表双指针技巧汇总](#) 全给你总结好了，会者不难，难者不会。**

比如判断单链表是否成环，拍脑袋的暴力解是什么？就是用一个 `HashSet` 之类的数据结构来缓存走过的节点，遇到重复的就说明有环对吧。但我们用快慢指针可以避免使用额外的空间，这就是聪明地穷举嘛。

**数组常用的技巧有也是双指针相关的技巧，也都属于「如何聪明地穷举」这一类。[数组双指针技巧汇总](#) 全给你总结好了，会者不难，难者不会。**

**首先说二分搜索技巧**，可以归为两端向中心的双指针。如果让你在数组中搜索元素，一个 `for` 循环花  $O(N)$  时间穷举肯定能搞定对吧，但是二分搜索告诉你，如果数组是有序的，它只要  $O(\log N)$  的复杂度，这不就是一种更聪明的搜索方式么。

[二分搜索框架详解](#) 给你总结了二分搜索代码模板，保证不会出现搜索边界的问题。[二分搜索算法运用](#) 给你总结了二分搜索相关题目的共性以及如何将二分搜索思想运用到实际算法中。

**再说说滑动窗口算法技巧**，典型的快慢双指针。你用嵌套 `for` 循环花  $O(N^2)$  的时间肯定可以穷举出所有子数组，也就必然可以找到符合题目要求的子数组。但是滑动窗口算法表示，在某些场景下，它可以用一快一慢两个指针，只需  $O(N)$  的时间就可以找到答案，这就是更聪明地穷举方式。

[滑动窗口算法框架详解](#) 介绍了滑动窗口算法的适用场景以及通用代码模板，保你写出正确的代码。[滑动窗口习题](#) 中手把手带你运用滑动窗口框架解决各种问题。

**最后说说前缀和技巧 和 差分数组技巧。**

如果频繁地让你计算子数组的和，每次用 `for` 循环去遍历肯定没问题，但前缀和技巧预算一个 `preSum` 数组，就可以避免循环。

类似的，如果频繁地让你对子数组进行增减操作，也可以每次用 `for` 循环去操作，但差分数组技巧维护一个 `diff` 数组，也可以避免循环。

数组链表的技巧差不多就这些了，都比较固定，只要你都见过，运用出来的难度不算大，下面来说一说稍微有些难度的算法。

## 二叉树系列算法

老读者都知道，二叉树的重要性我之前说了无数次，因为二叉树模型几乎是所有高级算法的基础，尤其是那么多人说对递归的理解不到位，更应该好好刷二叉树相关题目。

在本站的二叉树章节，我会按照固定的公式和思维模式讲解了 150 道二叉树题目，可以手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

**二叉树心法（纲领篇）** 说过，二叉树题目的递归解法可以分两类思路，第一类是遍历一遍二叉树得出答案，第二类是通过分解问题计算出答案，这两类思路分别对应着[回溯算法核心框架](#) 和[动态规划核心框架](#)。

## 遍历的思维模式

## 什么叫通过遍历一遍二叉树得出答案？

就比如说计算二叉树最大深度这个问题让你实现 `maxDepth` 这个函数，你这样写代码完全没问题：

```
class Solution {

    // 记录最大深度
    int res = 0;
    // 记录当前遍历节点的深度
    int depth = 0;

    // 主函数
    int maxDepth(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历框架
    void traverse(TreeNode root) {
        if (root == null) {
            // 到达叶子节点
            res = Math.max(res, depth);
            return;
        }
        // 前序遍历位置
        depth++;
        traverse(root.left);
        traverse(root.right);
        // 后序遍历位置
        depth--;
    }
}
```

这个逻辑就是用 `traverse` 函数遍历了一遍二叉树的所有节点，维护 `depth` 变量，在叶子节点的时候更新最大深度。

你看这段代码，有没有觉得很熟悉？能不能和回溯算法的代码模板对应上？

不信你照着 [回溯算法核心框架](#) 中全排列问题的代码对比下，`backtrack` 函数就是 `traverse` 函数，换汤不换药，整体逻辑非常类似：

```
class Solution {
    // 记录所有全排列
    List<List<Integer>> res = new LinkedList<>();
    // 记录当前正在穷举的排列
    LinkedList<Integer> track = new LinkedList<>();

    // track 中的元素会被标记为 true，避免重复使用
    boolean[] used;

    // 主函数，输入一组不重复的数字，返回它们的全排列
    List<List<Integer>> permute(int[] nums) {
        used = new boolean[nums.length];

        backtrack(nums);
        return res;
    }
}
```

```
// 回溯算法核心框架，遍历回溯树，收集所有叶子节点上的全排列
void backtrack(int[] nums) {
    // 到达叶子节点，track 中的元素就是一个全排列
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            // nums[i] 已经在 track 中，跳过
            continue;
        }
        // 做选择
        track.add(nums[i]);
        used[i] = true;

        // 进入递归树的下一层
        backtrack(nums);

        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}
}
```

你看这代码虽然多，但本质不就是多叉树的遍历吗？所以说回溯算法本质就是遍历多叉树，你只要能把问题抽象成树结构，就一定能用回溯算法解决。

## 分解问题的思维模式

那什么叫通过分解问题计算答案？

同样是计算二叉树最大深度这个问题，你也可以写出下面这样的解法：

```
// 定义：输入根节点，返回这棵二叉树的最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 递归计算左右子树的最大深度
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 整棵树的最大深度就是左右子树的最大深度加一
    int res = Math.max(leftMax, rightMax) + 1;

    return res;
}
```

你看这段代码，有没有觉得很熟悉？有没有觉得有点动态规划解法代码的形式？

不信你看 [动态规划核心框架](#) 中凑零钱问题的暴力穷举解法：

```
// 定义：输入金额 amount，返回凑出 amount 的最少硬币个数
int coinChange(int[] coins, int amount) {
    // base case
    if (amount == 0) return 0;
    if (amount < 0) return -1;

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        // 递归计算凑出 amount - coin 的最少硬币个数
        int subProblem = coinChange(coins, amount - coin);
        if (subProblem == -1) continue;
        // 凑出 amount 的最少硬币个数
        res = Math.min(res, subProblem + 1);
    }

    return res == Integer.MAX_VALUE ? -1 : res;
}
```

这个暴力解法加个 `memo` 备忘录就是自顶向下的动态规划解法，你对照二叉树最大深度的解法代码，有没有发现很像？

## 思路拓展

如果你感受到最大深度这个问题两种解法的区别，那就趁热打铁，我问你，二叉树的前序遍历怎么写？

我相信大家都会对这个问题嗤之以鼻，毫不犹豫就可以写出下面这段代码：

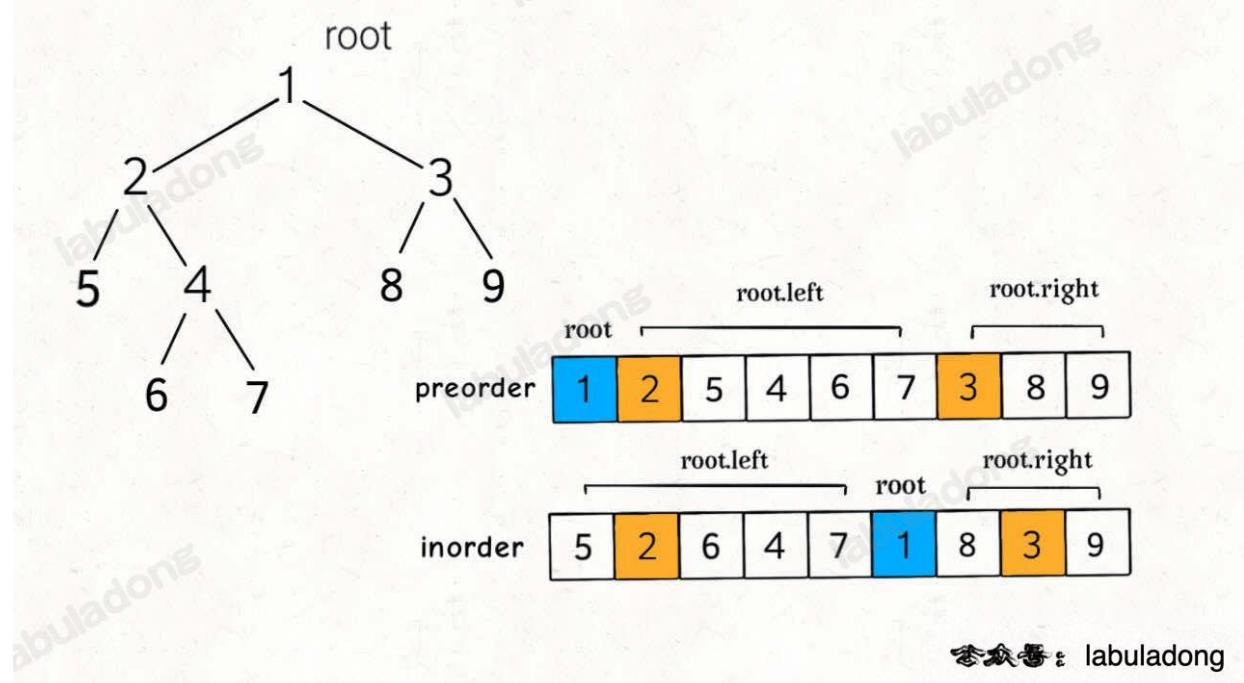
```
List<Integer> res = new LinkedList<>();

// 返回前序遍历结果
List<Integer> preorder(TreeNode root) {
    traverse(root);
    return res;
}

// 二叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序遍历位置
    res.add(root.val);
    traverse(root.left);
    traverse(root.right);
}
```

但是，你结合上面说到的两种不同的思维模式，二叉树的遍历是否也可以通过分解问题的思路解决呢？

可以观察一下二叉树前序遍历结果的特点：



你注意前序遍历的结果，根节点的值在第一位，后面接着左子树的前序遍历结果，最后接着右子树的前序遍历结果。

有没有体会出点什么来？其实完全可以重写前序遍历代码，用分解问题的形式写出来：

```
// 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
List<Integer> preorder(TreeNode root) {
    List<Integer> res = new LinkedList<>();
    if (root == null) {
        return res;
    }
    // 前序遍历的结果，root.val 在第一个
    res.add(root.val);
    // 后面接着左子树的前序遍历结果
    res.addAll(preorder(root.left));
    // 最后接着右子树的前序遍历结果
    res.addAll(preorder(root.right));
    return res;
}
```

你看，这就是用分解问题的思维模式写二叉树的前序遍历，如果写中序和后序遍历也是类似的。

## 层序遍历

除了动归、回溯（DFS）、分治，还有一个常用算法就是 BFS 了，[BFS 算法核心框架](#) 就是根据下面这段二叉树的层序遍历代码改装出来的：

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    int depth = 1;
    // 从上到下遍历二叉树的每一层
```

```

while (!q.isEmpty()) {
    int sz = q.size();
    // 从左到右遍历每一层的每个节点
    for (int i = 0; i < sz; i++) {
        TreeNode cur = q.poll();

        if (cur.left != null) {
            q.offer(cur.left);
        }
        if (cur.right != null) {
            q.offer(cur.right);
        }
    }
    depth++;
}
}

```

更进一步，图论相关的算法也是二叉树算法的延续。

比如 [图论基础](#), [环判断和拓扑排序](#) 和 [二分图判定算法](#) 就用到了 DFS 算法；再比如 [Dijkstra 算法模板](#)，就是改造版 BFS 算法加上一个类似 dp table 的数组。

好了，说的差不多了，上述这些算法的本质都是穷举二（多）叉树，有机会的话通过剪枝或者备忘录的方式减少冗余计算，提高效率，就这么点事儿。

## 最后总结

很多读者问我什么刷题方式是正确的，我认为正确的刷题方式应该是刷一道题能获得刷十道题的效果，不然力扣现在 2000 道题目，你都打算刷完么？

那么怎么做到呢？要有框架思维，学会提炼重点，寻找那个不变的东西。一个算法技巧可以包装出一万道题，如果你能一眼看穿它们的本质，那么一万道题等于一道，何必浪费时间去做呢？

**这就是框架的力量，能够保证你在快睡着的时候，依然能写出正确的程序；就算你啥都没学过，就这种思维方法，都能比别人高一个维度。**

授人以鱼不如授人以渔，算法真的没啥难的，只要有心，谁都可以学好。我希望你能在我这里培养出成体系的思维方法，享受支配算法的乐趣，而不是被算法支配。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">104. Maximum Depth of Binary Tree</a>	<a href="#">104. 二叉树的最大深度</a>	
<a href="#">112. Path Sum</a>	<a href="#">112. 路径总和</a>	
<a href="#">117. Populating Next Right Pointers in Each Node II</a>	<a href="#">117. 填充每个节点的下一个右侧节点指针 II</a>	
<a href="#">1214. Two Sum BSTs</a>	<a href="#">1214. 查找两棵二叉搜索树之和</a>	
<a href="#">294. Flip Game II</a>	<a href="#">294. 翻转游戏 II</a>	
<a href="#">341. Flatten Nested List Iterator</a>	<a href="#">341. 扁平化嵌套列表迭代器</a>	

LeetCode	力扣	难度
365. Water and Jug Problem	365. 水壶问题	
395. Longest Substring with At Least K Repeating Characters	395. 至少有 K 个重复字符的最长子串	
589. N-ary Tree Preorder Traversal	589. N 叉树的前序遍历	
590. N-ary Tree Postorder Traversal	590. N 叉树的后序遍历	
653. Two Sum IV - Input is a BST	653. 两数之和 IV - 输入二叉搜索树	
94. Binary Tree Inorder Traversal	94. 二叉树的中序遍历	
-	剑指 Offer 55 - I. 二叉树的深度	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 双指针技巧秒杀七道链表题目



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">142. Linked List Cycle II</a>	<a href="#">142. 环形链表 II</a>	🟡
<a href="#">23. Merge k Sorted Lists</a>	<a href="#">23. 合并K个升序链表</a>	🔴
-	<a href="#">剑指 Offer 22. 链表中倒数第k个节点</a>	🟢
<a href="#">21. Merge Two Sorted Lists</a>	<a href="#">21. 合并两个有序链表</a>	🟢
<a href="#">141. Linked List Cycle</a>	<a href="#">141. 环形链表</a>	🟢
<a href="#">19. Remove Nth Node From End of List</a>	<a href="#">19. 删除链表的倒数第 N 个结点</a>	🟡
<a href="#">86. Partition List</a>	<a href="#">86. 分隔链表</a>	🟡
<a href="#">876. Middle of the Linked List</a>	<a href="#">876. 链表的中间结点</a>	🟢
<a href="#">160. Intersection of Two Linked Lists</a>	<a href="#">160. 相交链表</a>	🟢

阅读本文前，你需要先学习：

- 链表基础
- 链表实现

tip：本文有视频版：[链表双指针技巧全面汇总](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文总结一下单链表的基本技巧，每个技巧都对应着至少一道算法题：

- 1、合并两个有序链表
- 2、链表的分解
- 3、合并  $k$  个有序链表
- 4、寻找单链表的倒数第  $k$  个节点
- 5、寻找单链表的中点
- 6、判断单链表是否包含环并找出环起点
- 7、判断两个单链表是否相交并找出交点

这些解法都用到了双指针技巧，所以说对于单链表相关的题目，双指针的运用是非常广泛的，下面我们就来一个一个看。

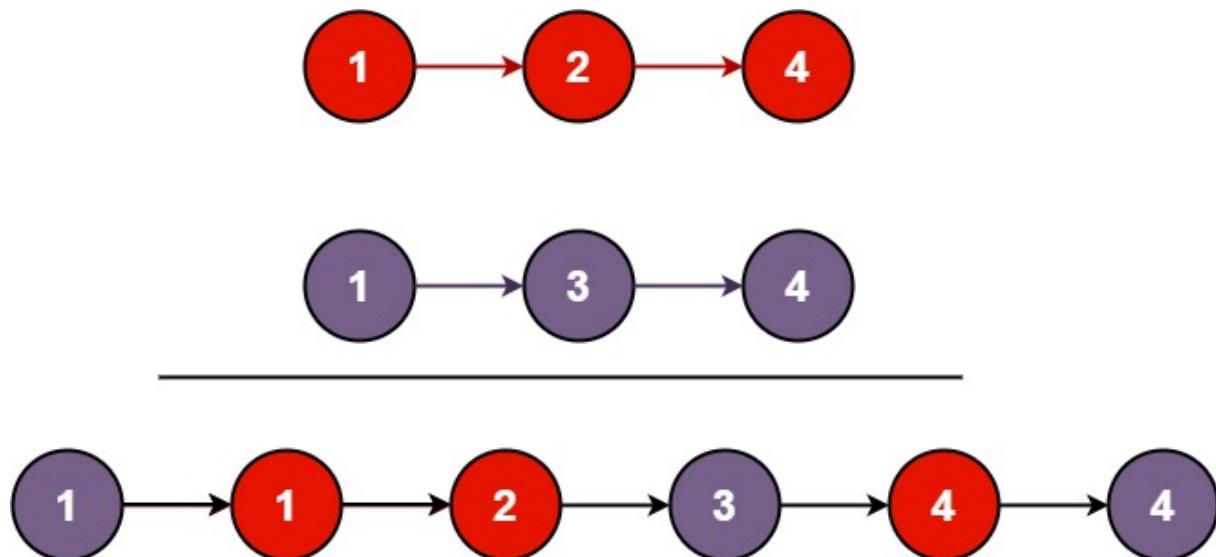
## 合并两个有序链表

这是最基本的链表技巧，力扣第 21 题「合并两个有序链表」就是这个问题，给你输入两个有序链表，请你把他俩合并成一个新的有序链表：

### ▼ 21. 合并两个有序链表 [Leetcode | 力扣](#)

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。

示例 1：



```
输入: l1 = [1,2,4], l2 = [1,3,4]
输出: [1,1,2,3,4,4]
```

示例 2：

```
输入: l1 = [], l2 = []
输出: []
```

示例 3：

```
输入: l1 = [], l2 = [0]
输出: [0]
```

提示：

- 两个链表的节点数目范围是  $[0, 50]$
- $-100 \leq \text{node.val} \leq 100$
- $\text{l1}$  和  $\text{l2}$  均按 非递减顺序 排列

```
// 函数签名如下
ListNode mergeTwoLists(ListNode l1, ListNode l2);
```

这题比较简单，我们直接看解法：

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1), p = dummy;
        ListNode p1 = l1, p2 = l2;

        while (p1 != null && p2 != null) {
            // 比较 p1 和 p2 两个指针
            // 将值较小的的节点接到 p 指针
            if (p1.val > p2.val) {
                p.next = p2;
                p2 = p2.next;
            } else {
                p.next = p1;
                p1 = p1.next;
            }
            // p 指针不断前进
            p = p.next;
        }

        if (p1 != null) {
            p.next = p1;
        }

        if (p2 != null) {
            p.next = p2;
        }

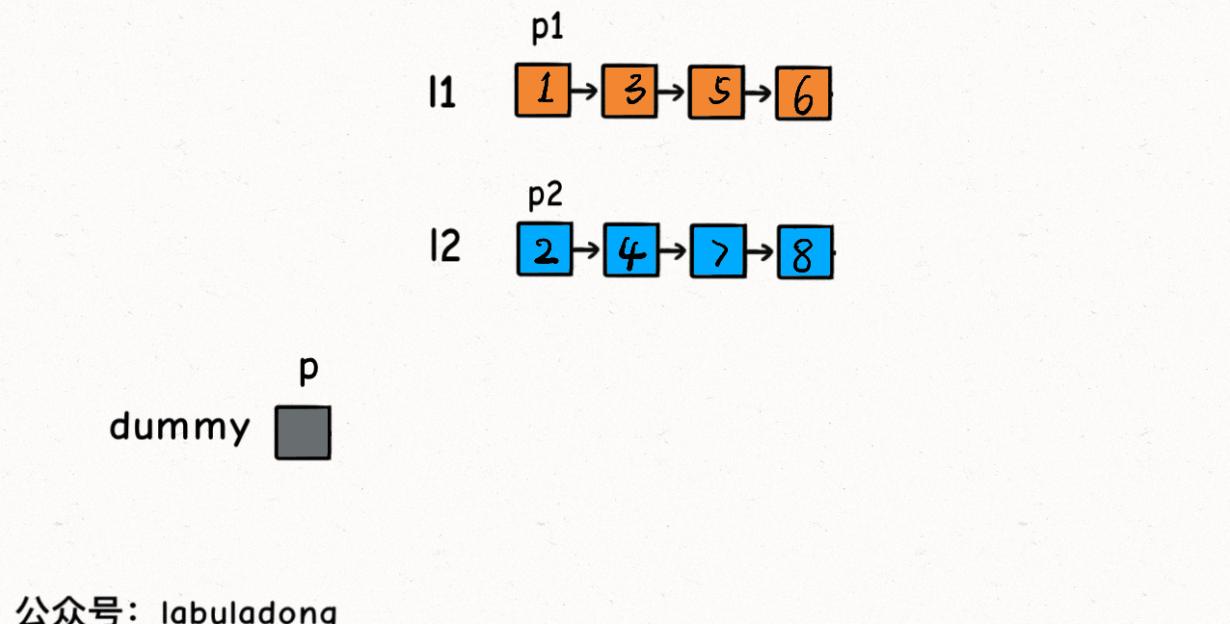
        return dummy.next;
    }
}
```

---

► 🎨 代码可视化动画🎨

---

我们的 while 循环每次比较 `p1` 和 `p2` 的大小，把较小的节点接到结果链表上，看如下 GIF：



公众号: labuladong

形象地理解，这个算法的逻辑类似于拉拉链，**l1**, **l2** 类似于拉链两侧的锯齿，指针 **p** 就好像拉链的拉索，将两个有序链表合并；或者说这个过程像蛋白酶合成蛋白质，**l1**, **l2** 就好比两条氨基酸，而指针 **p** 就好像蛋白酶，将氨基酸组合成蛋白质。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 **dummy** 节点。你可以试试，如果不使用 **dummy** 虚拟节点，代码会复杂一些，需要额外处理指针 **p** 为空的情况。而有了 **dummy** 节点这个占位符，可以避免处理空指针的情况，降低代码的复杂性。

经常有读者问我，什么时候需要用虚拟头结点？我这里总结下：当你需要创造一条新链表的时候，可以使用虚拟头结点简化边界情况的处理。

比如说，让你把两条有序链表合并成一条新的有序链表，是不是要创造一条新链表？再比你想把一条链表分解成两条链表，是不是也在创造新链表？这些情况都可以使用虚拟头结点简化边界情况的处理。

## 单链表的分解

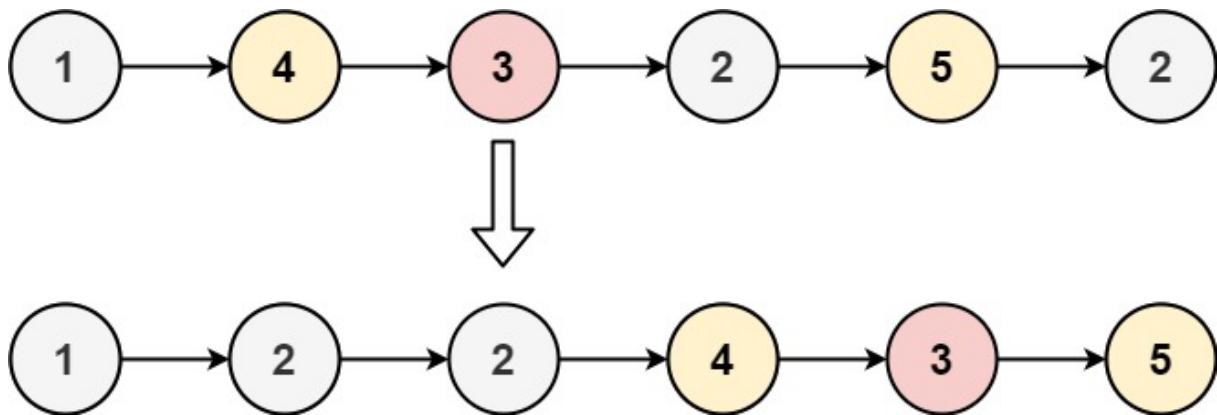
直接看下力扣第 86 题「分隔链表」：

### ▼ 86. 分隔链表 [Leetcode](#) | [力扣](#)

给你一个链表的头节点 **head** 和一个特定值 **x**，请你对链表进行分隔，使得所有 **小于 x** 的节点都出现在 **大于或等于 x** 的节点之前。

你应当 **保留** 两个分区中每个节点的初始相对位置。

示例 1：



```
输入: head = [1,4,3,2,5,2], x = 3  
输出: [1,2,2,4,3,5]
```

### 示例 2:

```
输入: head = [2,1], x = 2  
输出: [1,2]
```

### 提示:

- 链表中节点的数目在范围  $[0, 200]$  内
- $-100 \leq \text{node.val} \leq 100$
- $-200 \leq x \leq 200$

在合并两个有序链表时让你合二为一，而这里需要分解让你把原链表一分为二。具体来说，我们可以把原链表分成两个小链表，一个链表中的元素大小都小于  $x$ ，另一个链表中的元素都大于等于  $x$ ，最后再把这两条链表接到一起，就得到了题目想要的结果。

整体逻辑和合并有序链表非常相似，细节直接看代码吧，注意虚拟头结点的运用：

```
class Solution {  
    public ListNode partition(ListNode head, int x) {  
        // 存放小于 x 的链表的虚拟头结点  
        ListNode dummy1 = new ListNode(-1);  
        // 存放大于等于 x 的链表的虚拟头结点  
        ListNode dummy2 = new ListNode(-1);  
        // p1, p2 指针负责生成结果链表  
        ListNode p1 = dummy1, p2 = dummy2;  
        // p 负责遍历原链表，类似合并两个有序链表的逻辑  
        // 这里是将一个链表分解成两个链表  
        ListNode p = head;  
        while (p != null) {  
            if (p.val >= x) {  
                p2.next = p;  
                p2 = p2.next;  
            } else {  
                p1.next = p;  
                p1 = p1.next;  
            }  
        }  
        // 不能直接让 p 指针前进，
```

```
// p = p.next
// 断开原链表中的每个节点的 next 指针
ListNode temp = p.next;
p.next = null;
p = temp;
}
// 连接两个链表
p1.next = dummy2.next;

return dummy1.next;
}
```

我知道有很多读者会对这段代码有疑问：

```
// 不能直接让 p 指针前进,
// p = p.next
// 断开原链表中的每个节点的 next 指针
ListNode temp = p.next;
p.next = null;
p = temp;
```

不多废话，直接借助我们的可视化面板看一下就明白了。首先看下正确的写法：

---

►  代码可视化动画

---

如果你不断开原链表中的每个节点的 `next` 指针，那么就会出错，因为结果链表中会包含一个环：

---

►  代码可视化动画

---

总的来说，如果我们需要把原链表的节点接到新链表上，而不是 `new` 新节点来组成新链表的话，那么断开节点和原链表之间的链接可能是必要的。那其实我们可以养成一个好习惯，但凡遇到这种情况，就把原链表的节点断开，这样就不会出错了。

## 合并 k 个有序链表

看下力扣第 23 题「合并K个升序链表」：

▼ 23. 合并 K 个升序链表 [Leetcode](#) | [力扣](#)

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入: lists = [[1,4,5],[1,3,4],[2,6]]
输出: [1,1,2,3,4,4,5,6]
解释: 链表数组如下:
[
  1->4->5,
  1->3->4,
  2->6
```

]

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

## 示例 2:

输入: lists = []

输出: []

## 示例 3:

输入: lists = [[]]

输出: []

## 提示:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` 按升序排列
- `lists[i].length` 的总和不超过 `10^4`

```
// 函数签名如下  
ListNode mergeKLists(ListNode[] lists);
```

合并 `k` 个有序链表的逻辑类似合并两个有序链表，难点在于，如何快速得到 `k` 个节点中的最小节点，接到结果链表上？

这里我们就要用到优先级队列这种数据结构，把链表节点放入一个最小堆，就可以每次获得 `k` 个节点中的最小节点。关于优先级队列可以参考 [优先级队列（二叉堆）原理及实现](#)，本文不展开。

```
class Solution {  
    public ListNode mergeKLists(ListNode[] lists) {  
        if (lists.length == 0) return null;  
        // 虚拟头结点  
        ListNode dummy = new ListNode(-1);  
        ListNode p = dummy;  
        // 优先级队列，最小堆  
        PriorityQueue<ListNode> pq = new PriorityQueue<>(  
            lists.length, (a, b)->(a.val - b.val));  
        // 将 k 个链表的头结点加入最小堆  
        for (ListNode head : lists) {  
            if (head != null) {  
                pq.add(head);  
            }  
        }  
  
        while (!pq.isEmpty()) {  
            // 获取最小节点，接到结果链表中  
        }  
    }  
}
```

```

ListNode node = pq.poll();
p.next = node;
if (node.next != null) {
    pq.add(node.next);
}
// p 指针不断前进
p = p.next;
}
return dummy.next;
}
}

```

这个算法是面试常考题，它的时间复杂度是多少呢？

优先队列 `pq` 中的元素个数最多是 `k`，所以一次 `poll` 或者 `add` 方法的时间复杂度是  $O(\log k)$ ；所有的链表节点都会被加入和弹出 `pq`，所以算法整体的时间复杂度是  $O(N \log k)$ ，其中 `k` 是链表的条数，`N` 是这些链表的节点总数。

## 单链表的倒数第 `k` 个节点

从前往后寻找单链表的第 `k` 个节点很简单，一个 `for` 循环遍历过去就找到了，但是如何寻找从后往前数的第 `k` 个节点呢？

那你可能说，假设链表有 `n` 个节点，倒数第 `k` 个节点就是正数第 `n - k + 1` 个节点，不也是一个 `for` 循环的事儿吗？

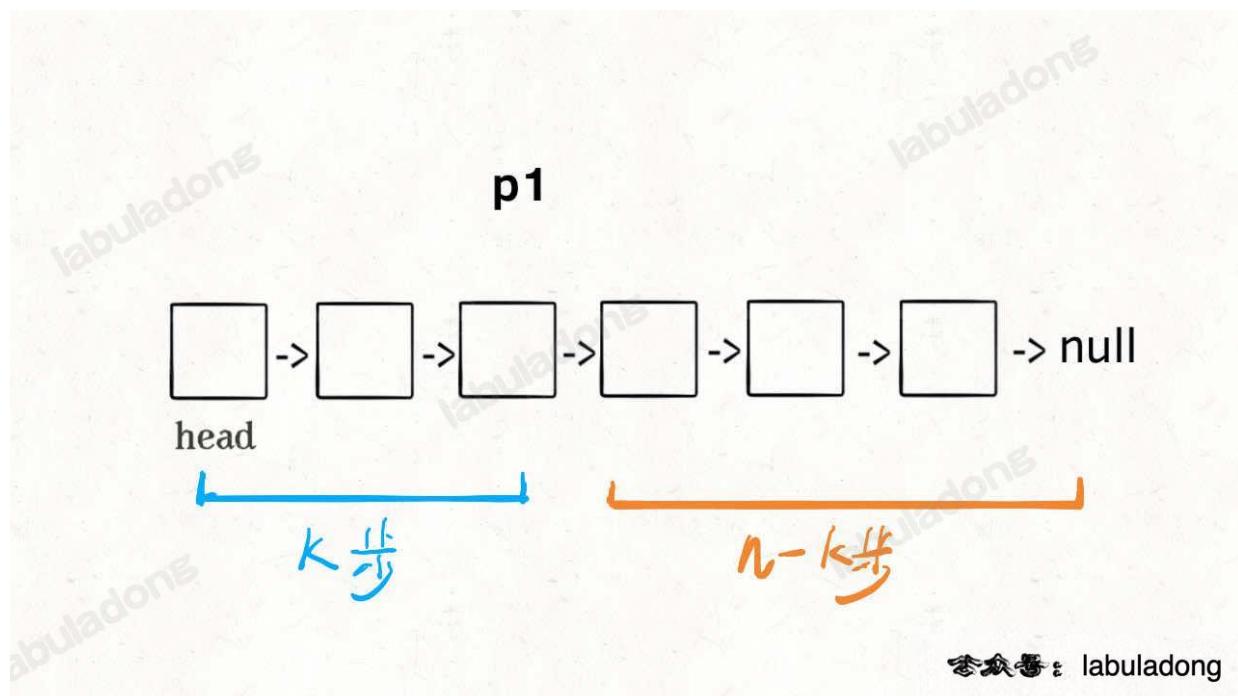
是的，但是算法题一般只给你一个 `ListNode` 头结点代表一条单链表，你不能直接得出这条链表的长度 `n`，而需要先遍历一遍链表算出 `n` 的值，然后再遍历链表计算第 `n - k + 1` 个节点。

也就是说，这个解法需要遍历两次链表才能得到倒数第 `k` 个节点。

那么，我们能不能只遍历一次链表，就算出倒数第 `k` 个节点？可以做到的，如果是面试问到这道题，面试官肯定也是希望你给出只需遍历一次链表的解法。

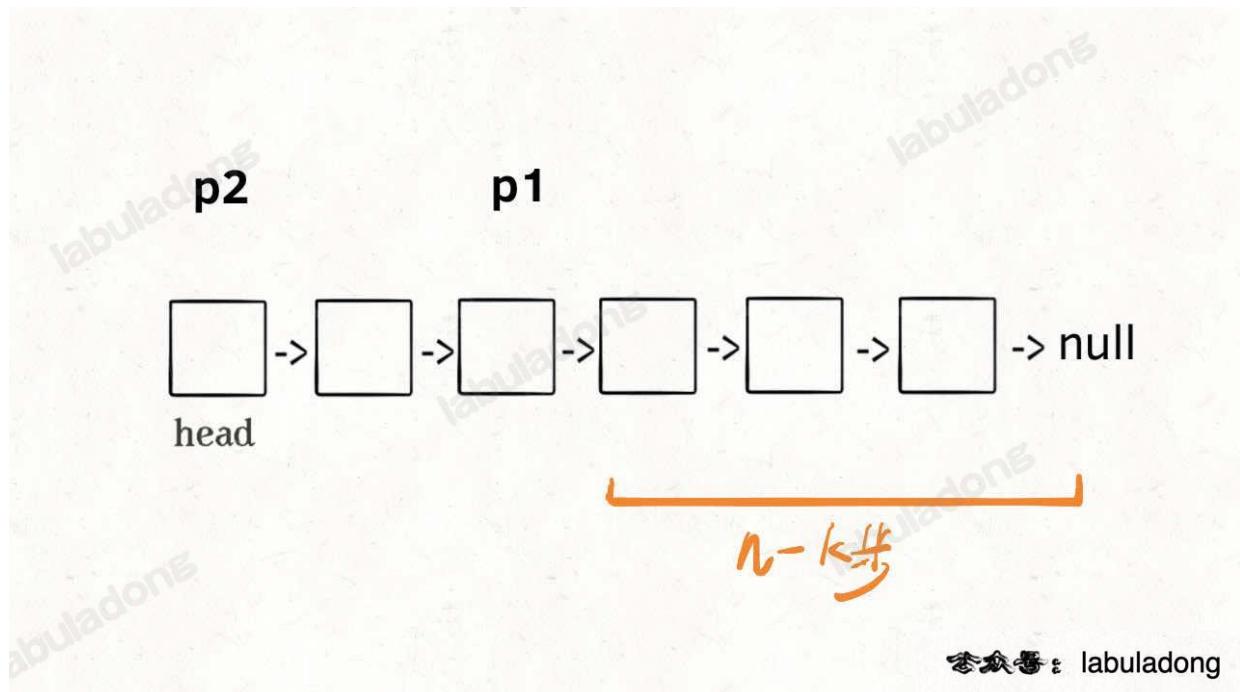
这个解法就比较巧妙了，假设 `k = 2`，思路如下：

首先，我们先让一个指针 `p1` 指向链表的头节点 `head`，然后走 `k` 步：

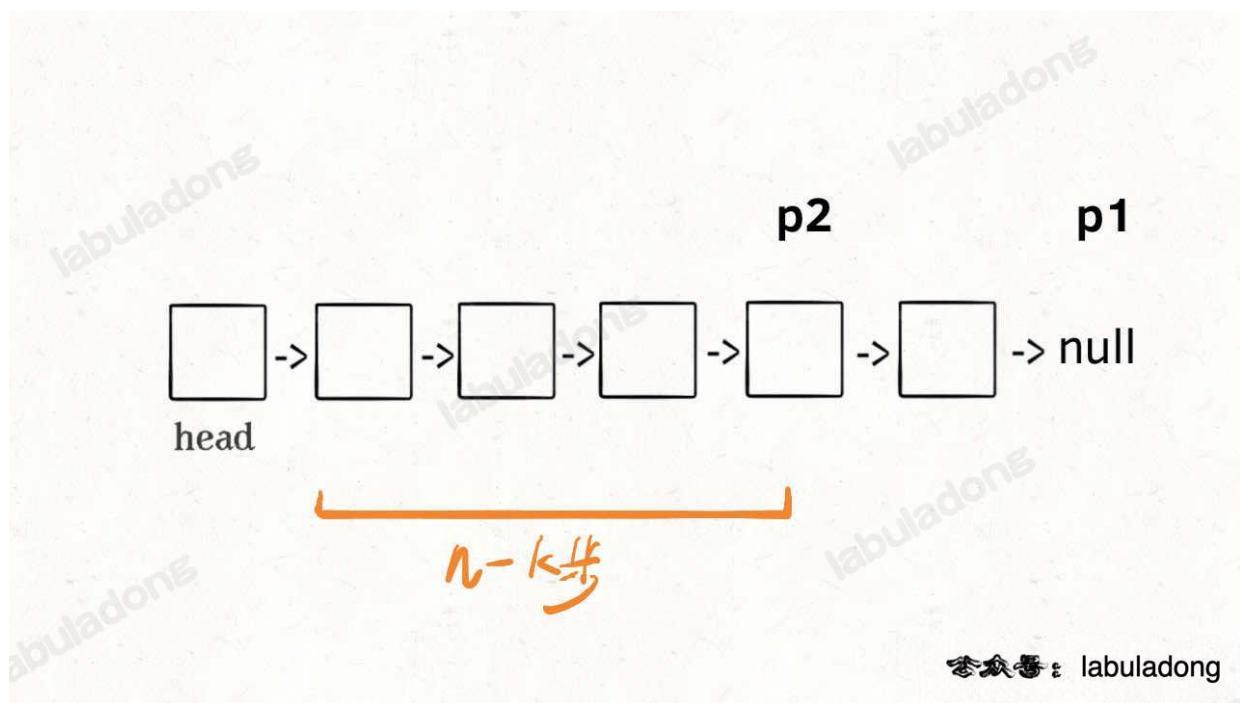


现在的 `p1`，只要再走 `n - k` 步，就能走到链表末尾的空指针了对吧？

趁这个时候，再用一个指针  $p2$  指向链表头节点  $head$ ：



接下来就很显然了，让  $p1$  和  $p2$  同时向前走， $p1$  走到链表末尾的空指针时前进了  $n - k$  步， $p2$  也从  $head$  开始前进了  $n - k$  步，停留在第  $n - k + 1$  个节点上，即恰好停在链表的倒数第  $k$  个节点上：



这样，只遍历了一次链表，就获得了倒数第  $k$  个节点  $p2$ 。

上述逻辑的代码如下：

```
// 返回链表的倒数第 k 个节点
ListNode findFromEnd(ListNode head, int k) {
    ListNode p1 = head;
    // p1 先走 k 步
    for (int i = 0; i < k; i++) {
        p1 = p1.next;
    }
}
```

```
ListNode p2 = head;
// p1 和 p2 同时走 n - k 步
while (p1 != null) {
    p2 = p2.next;
    p1 = p1.next;
}
// p2 现在指向第 n - k + 1 个节点, 即倒数第 k 个节点
return p2;
}
```

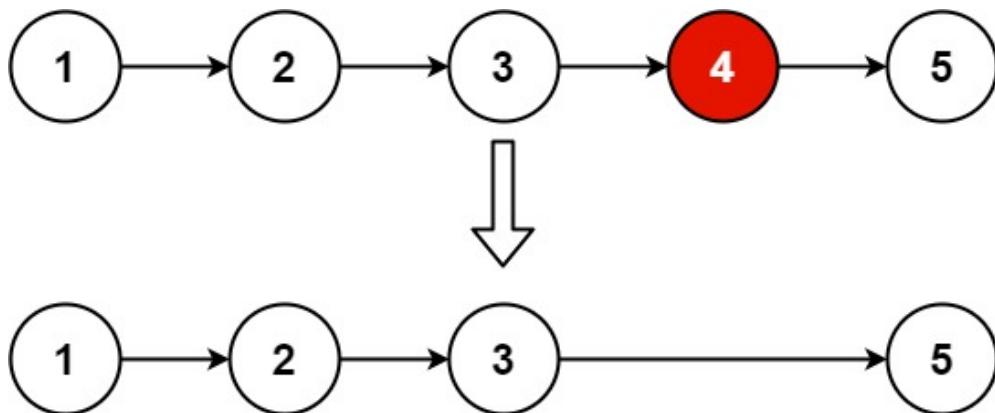
当然，如果用 big O 表示法来计算时间复杂度，无论遍历一次链表和遍历两次链表的时间复杂度都是  $O(N)$ ，但上述这个算法更有技巧性。

很多链表相关的算法题都会用到这个技巧，比如说力扣第 19 题「删除链表的倒数第 N 个结点」：

▼ 19. 删除链表的倒数第 N 个结点 [Leetcode](#) | [力扣](#)

给你一个链表，删除链表的倒数第  $n$  个结点，并且返回链表的头结点。

### 示例 1：



**输入:** head = [1,2,3,4,5], n = 2  
**输出:** [1,2,3,5]

## 示例 2：

**输入:** head = [1], n = 1  
**输出:** []

### 示例 3：

**输入:** head = [1, 2], n = 1  
**输出:** [1]

### 提示：

- 链表中结点的数目为  $sz$
  - $1 \leq sz \leq 30$
  - $0 \leq \text{Node.val} \leq 100$

- $1 \leq n \leq sz$

进阶：你能尝试使用一趟扫描实现吗？

我们直接看解法代码：

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
        ListNode x = findFromEnd(dummy, n + 1);
        // 删掉倒数第 n 个节点
        x.next = x.next.next;
        return dummy.next;
    }

    private ListNode findFromEnd(ListNode head, int k) {
        // 代码见上文
    }
}
```

### ▶ 🎨 代码可视化动画 🎨

这个逻辑就很简单了，要删除倒数第  $n$  个节点，就得获得倒数第  $n + 1$  个节点的引用，可以用我们实现的 `findFromEnd` 来操作。

不过注意我们又使用了虚拟头结点的技巧，也是为了防止出现空指针的情况，比如说链表总共有 5 个节点，题目就让你删除倒数第 5 个节点，也就是第一个节点，那按照算法逻辑，应该首先找到倒数第 6 个节点。但第一个节点前面已经没有节点了，这就会出错。

但有了我们虚拟节点 `dummy` 的存在，就避免了这个问题，能够对这种情况进行正确的删除。

## 单链表的中点

力扣第 876 题「链表的中间结点」就是这个题目，问题的关键也在于我们无法直接得到单链表的长度  $n$ ，常规方法也是先遍历链表计算  $n$ ，再遍历一次得到第  $n / 2$  个节点，也就是中间节点。

如果想一次遍历就得到中间节点，也需要耍点小聪明，使用「快慢指针」的技巧：

我们让两个指针 `slow` 和 `fast` 分别指向链表头结点 `head`。

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步，这样，当 `fast` 走到链表末尾时，`slow` 就指向了链表中点。

上述思路的代码实现如下：

```
class Solution {
    public ListNode middleNode(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

```
        fast = fast.next.next;
    }
    // 慢指针指向中点
    return slow;
}
}
```

## ▶ 🎨 代码可视化动画

需要注意的是，如果链表长度为偶数，也就是说中点有两个的时候，我们这个解法返回的节点是靠后的那个节点。

另外，这段代码稍加修改就可以直接用到判断链表成环的算法题上。

## 判断链表是否包含环

判断链表是否包含环属于经典问题了，解决方案也是用快慢指针：

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。

如果 `fast` 最终能正常走到链表末尾，说明链表中没有环；如果 `fast` 走着走着竟然和 `slow` 相遇了，那肯定是 `fast` 在链表中转圈了，说明链表中含有环。

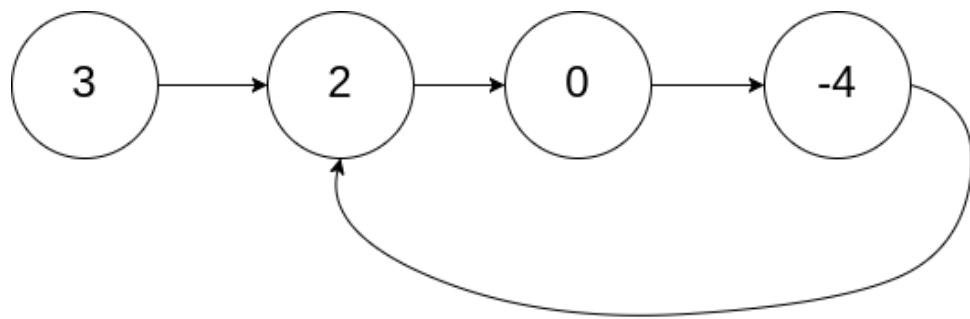
只需要把寻找链表中点的代码稍加修改就行了：

```
class Solution {
    public boolean hasCycle(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
            // 快慢指针相遇，说明含有环
            if (slow == fast) {
                return true;
            }
        }
        // 不包含环
        return false;
    }
}
```

## ▶ 🎃 代码可视化动画

当然，这个问题还有进阶版，也是力扣第 142 题「环形链表 II」：如果链表中含有环，如何计算这个环的起点？

为了避免读者迷惑，举个例子，环的起点是指下面这幅图中的节点 2：



这里先直接看一下寻找环起点的解法代码：

```
class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        // 上面的代码类似 hasCycle 函数
        if (fast == null || fast.next == null) {
            // fast 遇到空指针说明没有环
            return null;
        }

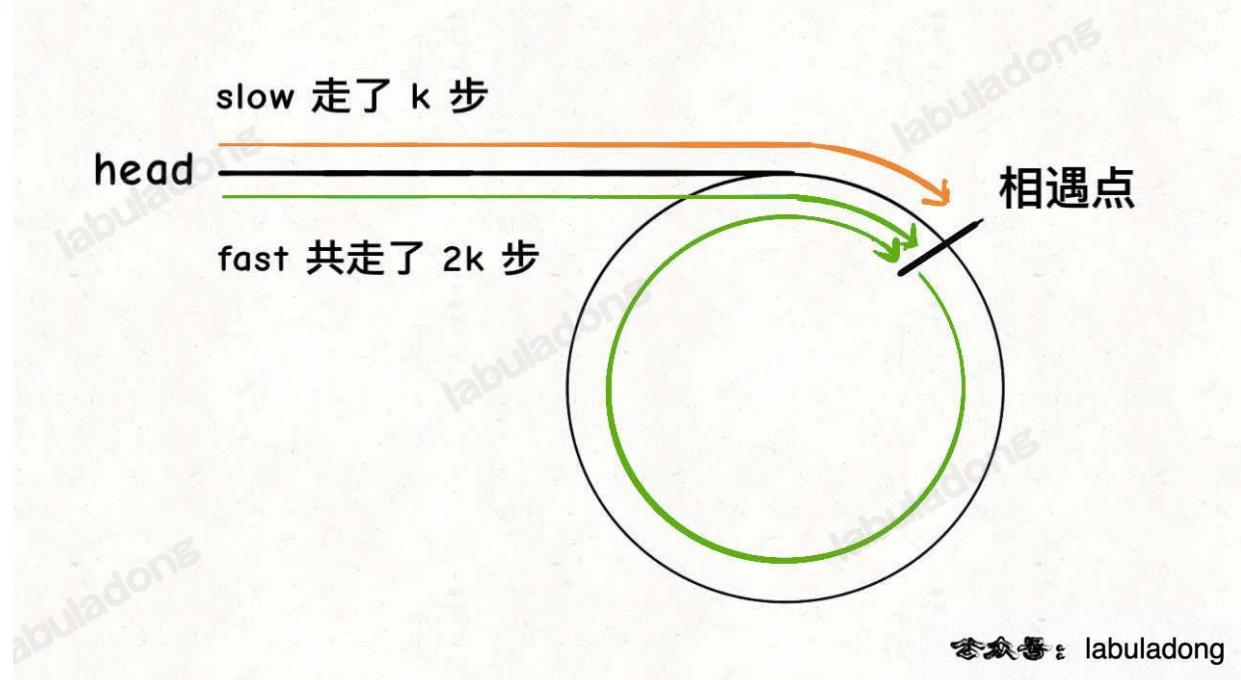
        // 重新指向头结点
        slow = head;
        // 快慢指针同步前进，相交点就是环起点
        while (slow != fast) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

#### ▶ 🎥 代码可视化动画 🎥

可以看到，当快慢指针相遇时，让其中一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。

为什么要这样呢？这里简单说一下其中的原理。

我们假设快慢指针相遇时，慢指针 `slow` 走了  $k$  步，那么快指针 `fast` 一定走了  $2k$  步：

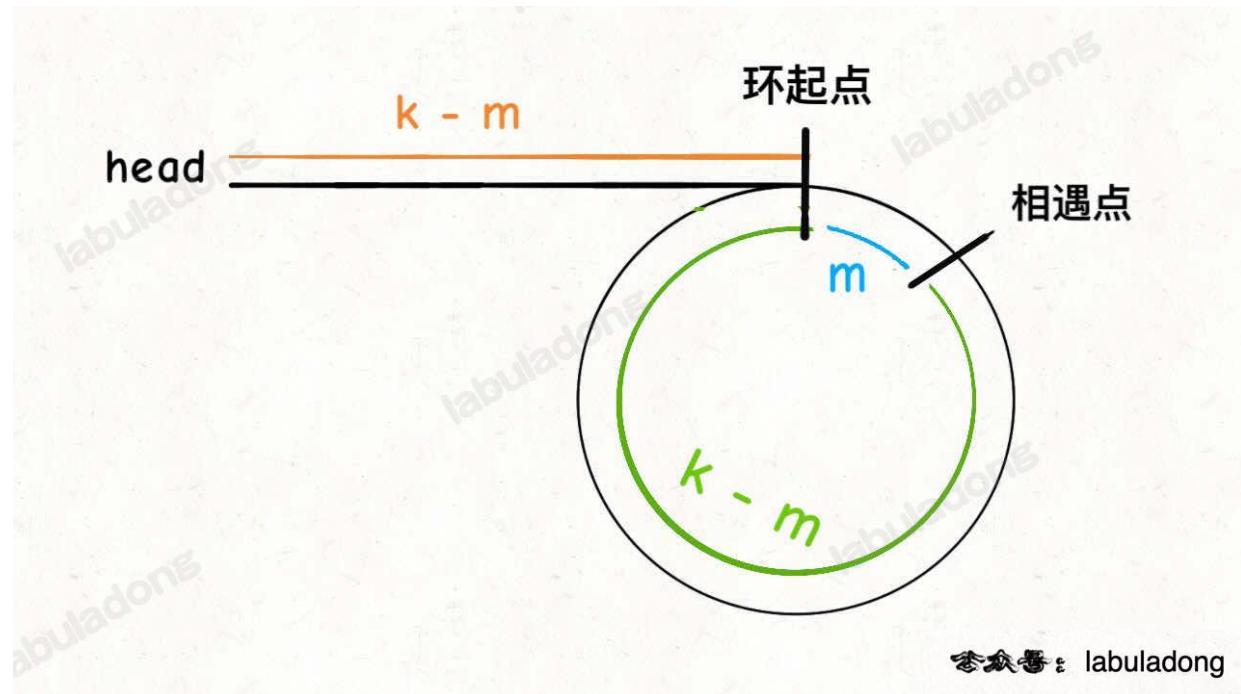


© labuladong

fast 一定比 slow 多走了  $k$  步，这多走的  $k$  步其实就是 fast 指针在环里转圈圈，所以  $k$  的值就是环长度的「整数倍」。

假设相遇点距环的起点的距离为  $m$ ，那么结合上图的 slow 指针，环的起点距头结点 head 的距离为  $k - m$ ，也就是说如果从 head 前进  $k - m$  步就能到达环起点。

巧的是，如果从相遇点继续前进  $k - m$  步，也恰好到达环起点。因为结合上图的 fast 指针，从相遇点开始走  $k$  步可以转回到相遇点，那走  $k - m$  步肯定就走到环起点了：



© labuladong

所以，只要我们把快慢指针中的任一个重新指向 head，然后两个指针同速前进， $k - m$  步后一定会相遇，相遇之处就是环的起点了。

## 两个链表是否相交

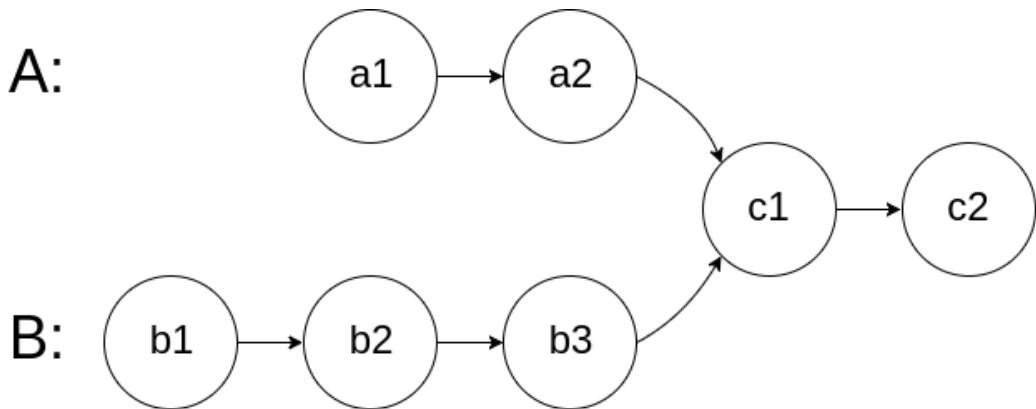
这个问题有意思，也是力扣第 160 题「相交链表」函数签名如下：

```
ListNode getIntersectionNode(ListNode headA, ListNode headB);
```

给你输入两个链表的头结点 `headA` 和 `headB`, 这两个链表可能存在相交。

如果相交, 你的算法应该返回相交的那个节点; 如果没相交, 则返回 `null`。

比如题目给我们举的例子, 如果输入的两个链表如下图:

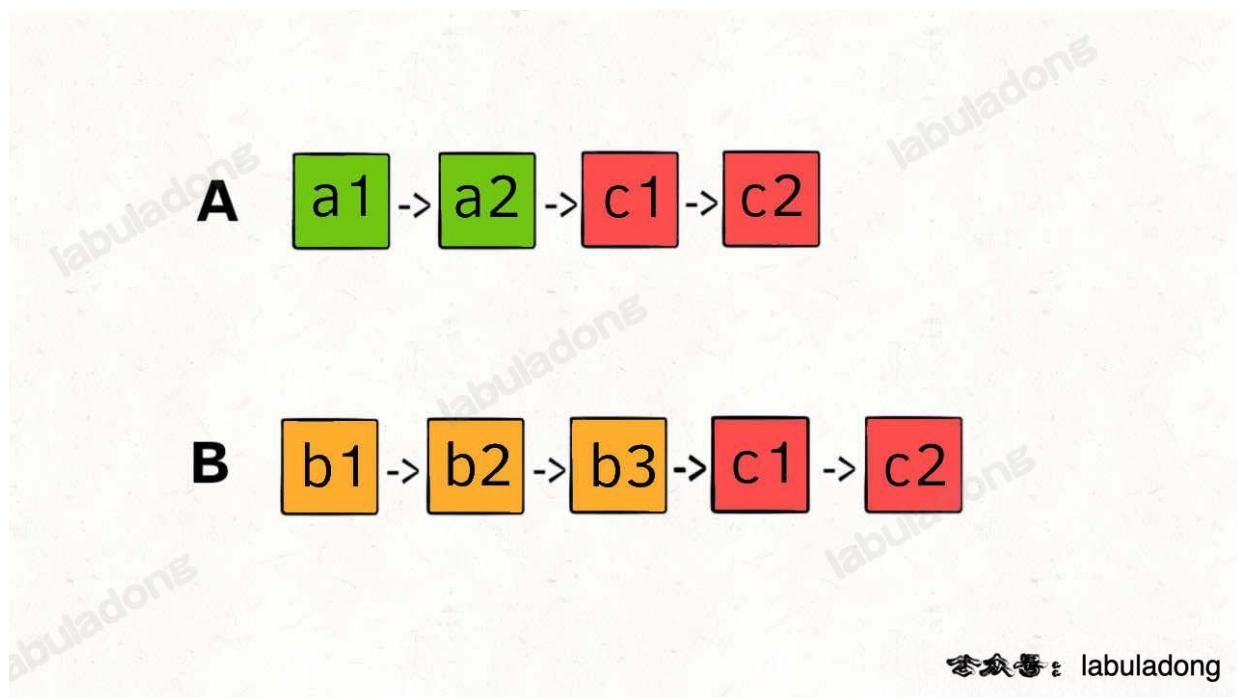


那么我们的算法应该返回 `c1` 这个节点。

这个题直接的想法可能是用 `HashSet` 记录一个链表的所有节点, 然后和另一条链表对比, 但这就需要额外的空间。

如果不额外的空间, 只使用两个指针, 你如何做呢?

难点在于, 由于两条链表的长度可能不同, 两条链表之间的节点无法对应:

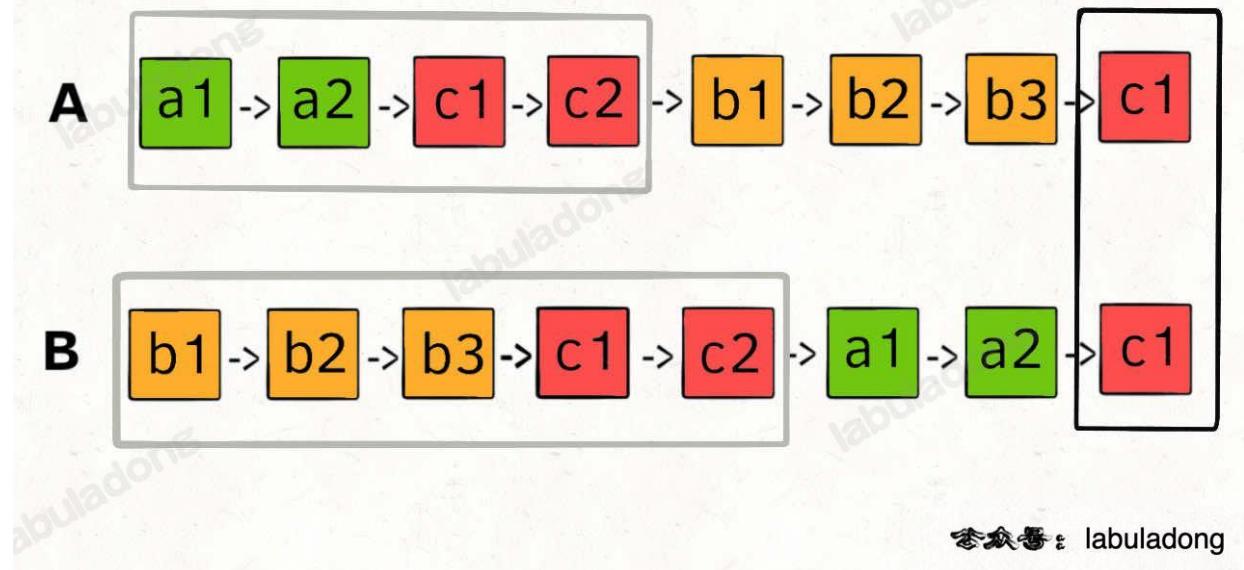


如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进, 并不能同时走到公共节点, 也就无法得到相交节点 `c1`。

解决这个问题的关键是, 通过某些方式, 让 `p1` 和 `p2` 能够同时到达相交节点 `c1`。

所以, 我们可以让 `p1` 遍历完链表 `A` 之后开始遍历链表 `B`, 让 `p2` 遍历完链表 `B` 之后开始遍历链表 `A`, 这样相当于「逻辑上」两条链表接在了一起。

如果这样进行拼接, 就可以让 `p1` 和 `p2` 同时进入公共部分, 也就是同时到达相交节点 `c1`:



那你可能会问，如果说两个链表没有相交点，是否能够正确的返回 null 呢？

这个逻辑可以覆盖这种情况的，相当于 c1 节点是 null 空指针嘛，可以正确返回 null。

按照这个思路，可以写出如下代码：

```
class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        // p1 指向 A 链表头结点, p2 指向 B 链表头结点
        ListNode p1 = headA, p2 = headB;
        while (p1 != p2) {
            // p1 走一步, 如果走到 A 链表末尾, 转到 B 链表
            if (p1 == null) {
                p1 = headB;
            } else {
                p1 = p1.next;
            }
            // p2 走一步, 如果走到 B 链表末尾, 转到 A 链表
            if (p2 == null) {
                p2 = headA;
            } else{
                p2 = p2.next;
            }
        }
        return p1;
    }
}
```

### ► ⭐ 代码可视化动画⭐

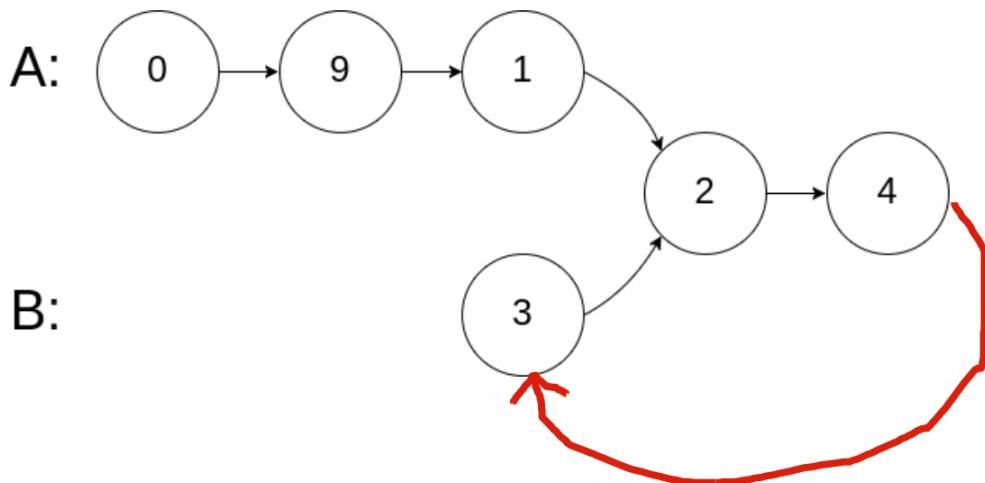
这样，这道题就解决了，空间复杂度为  $O(1)$ ，时间复杂度为  $O(N)$ 。

以上就是单链表的所有技巧，希望对你有启发。

2022/1/24 更新：

评论区有不少优秀读者对最后一题「寻找两条链表的交点」提出了一些其他思路，也补充到这里。

首先有读者提到，如果把两条链表首尾相连，那么「寻找两条链表的交点」的问题转换成了前面讲的「寻找环起点」的问题：



说实话我没有想到这种思路，不得不说这是一个很巧妙的转换！不过需要注意的是，这道题说不让你改变原始链表的结构，所以你把题目输入的链表转化成环形链表求解之后记得还要改回来，否则无法通过。

另外，还有读者提到，既然「寻找两条链表的交点」的核心在于让 `p1` 和 `p2` 两个指针能够同时到达相交节点 `c1`，那么可以通过预先计算两条链表的长度来做到这一点，具体代码如下：

```
class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = 0, lenB = 0;
        // 计算两条链表的长度
        for (ListNode p1 = headA; p1 != null; p1 = p1.next) {
            lenA++;
        }
        for (ListNode p2 = headB; p2 != null; p2 = p2.next) {
            lenB++;
        }
        // 让 p1 和 p2 到达尾部的距离相同
        ListNode p1 = headA, p2 = headB;
        if (lenA > lenB) {
            for (int i = 0; i < lenA - lenB; i++) {
                p1 = p1.next;
            }
        } else {
            for (int i = 0; i < lenB - lenA; i++) {
                p2 = p2.next;
            }
        }
        // 看两个指针是否会相同，p1 == p2 时有两种情况：
        // 1、要么是两条链表不相交，他俩同时走到尾部空指针
        // 2、要么是两条链表相交，他俩走到两条链表的相交点
        while (p1 != p2) {
    
```

```
    p1 = p1.next;
    p2 = p2.next;
}
return p1;
}
```

虽然代码多一些，但是时间复杂度是还是  $O(N)$ ，而且会更容易理解一些。

总之，我的解法代码并不一定就是最优或者最正确的，鼓励大家在评论区多多提出自己的疑问和思考，我也很高兴和大家探讨更多的解题思路~

到这里，链表相关的双指针技巧就全部讲完了，这些技巧的更多扩展延伸见 [更多链表双指针经典习题](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">109. Convert Sorted List to Binary Search Tree</a>	<a href="#">109. 有序链表转换二叉搜索树</a>	
<a href="#">1257. Smallest Common Region</a>	<a href="#">1257. 最小公共区域</a>	
<a href="#">1650. Lowest Common Ancestor of a Binary Tree III</a>	<a href="#">1650. 二叉树的最近公共祖先 III</a>	
<a href="#">1836. Remove Duplicates From an Unsorted Linked List</a>	<a href="#">1836. 从未排序的链表中移除重复元素</a>	
<a href="#">2. Add Two Numbers</a>	<a href="#">2. 两数相加</a>	
<a href="#">234. Palindrome Linked List</a>	<a href="#">234. 回文链表</a>	
<a href="#">264. Ugly Number II</a>	<a href="#">264. 丑数 II</a>	
<a href="#">313. Super Ugly Number</a>	<a href="#">313. 超级丑数</a>	
<a href="#">355. Design Twitter</a>	<a href="#">355. 设计推特</a>	
<a href="#">360. Sort Transformed Array</a>	<a href="#">360. 有序转化数组</a>	
<a href="#">373. Find K Pairs with Smallest Sums</a>	<a href="#">373. 查找和最小的 K 对数字</a>	
<a href="#">378. Kth Smallest Element in a Sorted Matrix</a>	<a href="#">378. 有序矩阵中第 K 小的元素</a>	
<a href="#">431. Encode N-ary Tree to Binary Tree</a>	<a href="#">431. 将 N 叉树编码为二叉树</a>	
<a href="#">88. Merge Sorted Array</a>	<a href="#">88. 合并两个有序数组</a>	
<a href="#">97. Interleaving String</a>	<a href="#">97. 交错字符串</a>	
<a href="#">977. Squares of a Sorted Array</a>	<a href="#">977. 有序数组的平方</a>	
-	<a href="#">剑指 Offer 18. 删除链表的节点</a>	
-	<a href="#">剑指 Offer 25. 合并两个排序的链表</a>	
-	<a href="#">剑指 Offer 49. 丑数</a>	
-	<a href="#">剑指 Offer 52. 两个链表的第一个公共节点</a>	
-	<a href="#">剑指 Offer II 021. 删除链表的倒数第 n 个结点</a>	

LeetCode	力扣	难度
-	剑指 Offer II 022. 链表中环的入口节点	🟡
-	剑指 Offer II 023. 两个链表的第一个重合节点	🟢
-	剑指 Offer II 027. 回文链表	🟢
-	剑指 Offer II 061. 和最小的 k 个数对	🟡
-	剑指 Offer II 078. 合并排序链表	🔴

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 双指针技巧秒杀七道数组题目



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">26. Remove Duplicates from Sorted Array</a>	26. 删除有序数组中的重复项	
-	剑指 Offer II 006. 排序数组中两个数字之和	
<a href="#">283. Move Zeroes</a>	283. 移动零	
<a href="#">167. Two Sum II - Input Array Is Sorted</a>	167. 两数之和 II - 输入有序数组	
<a href="#">344. Reverse String</a>	344. 反转字符串	
<a href="#">83. Remove Duplicates from Sorted List</a>	83. 删除排序链表中的重复元素	
-	剑指 Offer 57. 和为s的两个数字	
<a href="#">27. Remove Element</a>	27. 移除元素	
<a href="#">5. Longest Palindromic Substring</a>	5. 最长回文子串	

阅读本文前，你需要先学习：

- 数组基础
- 数组实现
- 单链表的六大解题套路

tip：本文有视频版：[数组双指针技巧汇总](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

在处理数组和链表相关问题时，双指针技巧是经常用到的，双指针技巧主要分为两类：**左右指针**和**快慢指针**。

所谓左右指针，就是两个指针相向而行或者相背而行；而所谓快慢指针，就是两个指针同向而行，一快一慢。

对于单链表来说，大部分技巧都属于快慢指针，[单链表的六大解题套路](#)都涵盖了，比如链表环判断，倒数第 K 个链表节点等问题，它们都是通过一个 **fast** 快指针和一个 **slow** 慢指针配合完成任务。

在数组中并没有真正意义上的指针，但我们可以把索引当做数组中的指针，这样也可以在数组中施展双指针技巧，**本文主要讲数组相关的双指针算法**。

## 一、快慢指针技巧

### 原地修改

数组问题中比较常见的快慢指针技巧，是让你原地修改数组。

比如说看下力扣第 26 题「删除有序数组中的重复项」，让你在有序数组去重：

▼ 26. 删除有序数组中的重复项 [Leetcode | 力扣](#)

给你一个 **非严格递增排列** 的数组 `nums`，请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次**，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致**。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 `k`，你需要做以下事情确保你的题解可以被通过：

- 更改数组 `nums`，使 `nums` 的前 `k` 个元素包含唯一元素，并按照它们最初在 `nums` 中出现的顺序排列。`nums` 的其余元素与 `nums` 的大小不重要。
- 返回 `k`。

**判题标准：**

系统会用下面的代码来测试你的题解：

```
int[] nums = [...]; // 输入数组
int[] expectedNums = [...]; // 长度正确的期望答案

int k = removeDuplicates(nums); // 调用

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有断言都通过，那么您的题解将被 **通过**。

**示例 1：**

**输入：** `nums = [1,1,2]`  
**输出：** `2, nums = [1,2,_]`

**解释：** 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度后面的元素。

**示例 2：**

**输入：** `nums = [0,0,1,1,1,2,2,3,3,4]`  
**输出：** `5, nums = [0,1,2,3,4]`

**解释：** 函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。不需要考虑数组中超出新长度后面的元素。

**提示：**

- `1 <= nums.length <= 3 * 104`
- `-104 <= nums[i] <= 104`
- `nums` 已按 **非严格递增** 排列

函数签名如下：

```
int removeDuplicates(int[] nums);
```

简单解释一下什么是原地修改：

如果不是原地修改的话，我们直接 new 一个 int[] 数组，把去重之后的元素放进这个新数组中，然后返回这个新数组即可。

但是现在题目让你原地删除，不允许 new 新数组，只能在原数组上操作，然后返回一个长度，这样就可以通过返回的长度和原始数组得到我们去重后的元素有哪些了。

由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难。但如果每找到一个重复元素就立即原地删除它，由于数组中删除元素涉及数据搬移，整个时间复杂度是会达到  $O(N^2)$ 。

高效解决这道题就要用到快慢指针技巧：

我们让慢指针 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赋值给 slow 并让 slow 前进一步。

这样，就保证了  $nums[0..slow]$  都是无重复的元素，当 fast 指针遍历完整个数组 nums 后， $nums[0..slow]$  就是整个数组去重之后的结果。

看代码：

```
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;
    }
}
```

算法执行的过程可以参考可视化面板：

---

▶  代码可视化动画

再简单扩展一下，看看力扣第 83 题「删除排序链表中的重复元素」，如果给你一个有序的单链表，如何去重呢？

其实和数组去重是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已，你对照着之前的代码来看：

```
class Solution {
    public 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;
}
```

算法执行的过程请看下面这个可视化面板：

▶ 🎃 代码可视化动画🎃

这里可能有读者会问，链表中那些重复的元素并没有被删掉，就让这些节点在链表上挂着，合适吗？

这就要探讨不同语言的特性了，像 Java/Python 这类带有垃圾回收的语言，可以帮我们自动找到并回收这些「悬空」的链表节点的内存，而像 C++ 这类语言没有自动垃圾回收的机制，确实需要我们编写代码时手动释放掉这些节点的内存。

不过话说回来，就算法思维的培养来说，我们只需要知道这种快慢指针技巧即可。

除了让你在有序数组/链表中去重，题目还可能让你对数组中的某些元素进行「原地删除」。

比如力扣第 27 题「移除元素」，看下题目：

▼ 27. 移除元素 [Leetcode](#) | [力扣](#)

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`，要通过此题，您需要执行以下操作：

- 更改 `nums` 数组，使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。
- 返回 `k`。

用户评测：

评测机将使用以下代码测试您的解决方案：

```
int[] nums = [...]; // 输入数组
int val = ...; // 要移除的值
int[] expectedNums = [...]; // 长度正确的预期答案。
                            // 它以不等于 val 的值排序。

int k = removeElement(nums, val); // 调用你的实现

assert k == expectedNums.length;
```

```
sort(nums, 0, k); // 排序 nums 的前 k 个元素
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有的断言都通过，你的解决方案将会 **通过**。

#### 示例 1：

```
输入: nums = [3,2,2,3], val = 3
输出: 2, nums = [2,2,_,_]
解释: 你的函数函数应该返回 k = 2，并且 nums 中的前两个元素均为 2。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

#### 示例 2：

```
输入: nums = [0,1,2,2,3,0,4,2], val = 2
输出: 5, nums = [0,1,4,0,3,_,_,_]
解释: 你的函数应该返回 k = 5，并且 nums 中的前五个元素为 0,0,1,3,4。
注意这五个元素可以任意顺序返回。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

#### 提示：

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 50$
- $0 \leq \text{val} \leq 100$

```
// 函数签名如下
int removeElement(int[] nums, int val);
```

题目要求我们把 `nums` 中所有值为 `val` 的元素原地删除，依然需要使用快慢指针技巧：

如果 `fast` 遇到值为 `val` 的元素，则直接跳过，否则就赋值给 `slow` 指针，并让 `slow` 前进一步。

这和前面说到的数组去重问题解法思路是完全一样的，直接看代码：

```
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;
    }
}
```

算法执行的过程可以参考可视化面板：

► 代码可视化动画

注意这里和有序数组去重的解法有一个细节差异，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`。

实现了这个 `removeElement` 函数，接下来看看力扣第 283 题「移动零」：

给你输入一个数组 `nums`，请你原地修改，将数组中的所有值为 0 的元素移到数组末尾，函数签名如下：

```
void moveZeroes(int[] nums);
```

比如说给你输入 `nums = [0,1,4,0,2]`，你的算法没有返回值，但是会把 `nums` 数组原地修改成 `[1,4,2,0,0]`。

结合之前说到的几个题目，你是否有已经有了答案呢？

稍微修改上一题中的 `removeElement` 函数就可以完成这道题，或者直接复用 `removeElement` 函数也可以。

题目让我们将所有 0 移到最后，其实就相当于移除 `nums` 中的所有 0，然后再把后面的元素都赋值为 0：

```
class Solution {
    public void moveZeroes(int[] nums) {
        // 去除 nums 中的所有 0，返回不含 0 的数组长度
        int p = removeElement(nums, 0);
        // 将 nums[p..] 的元素赋值为 0
        for (; p < nums.length; p++) {
            nums[p] = 0;
        }
    }

    public int removeElement(int[] nums, int val) {
        // 见上文代码实现
    }
}
```

► 代码可视化动画

到这里，原地修改数组的这些题目就已经差不多了。

## 滑动窗口

数组中另一大类快慢指针的题目就是「滑动窗口算法」。我在另一篇文章 [滑动窗口算法核心框架详解](#) 给出了滑动窗口的代码框架：

```
// 滑动窗口算法框架伪码
int left = 0, right = 0;

while (right < nums.size()) {
    // 增大窗口
```

```
window.addLast(nums[right]);
right++;

while (window needs shrink) {
    // 缩小窗口
    window.removeFirst(nums[left]);
    left++;
}
}
```

具体的题目本文就不重复了，这里只强调滑动窗口算法的快慢指针特性：

`left` 指针在后，`right` 指针在前，两个指针中间的部分就是「窗口」，算法通过扩大和缩小「窗口」来解决某些问题。

## 二、左右指针的常用算法

### 二分查找

我在另一篇文章 [二分查找框架详解](#) 中有详细探讨二分搜索代码的细节问题，这里只写最简单的二分算法，旨在突出它的双指针特性：

```
int binarySearch(int[] nums, int target) {
    // 一左一右两个指针相向而行
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid - 1;
    }
    return -1;
}
```

### n 数之和

看下力扣第 167 题「两数之和 II」：

#### ▼ 167. 两数之和 II - 输入有序数组 [Leetcode | 力扣](#)

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 `1 <= index1 < index2 <= numbers.length`。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

#### 示例 1:

输入: numbers = [2, 7, 11, 15], target = 9

输出: [1, 2]

解释: 2 与 7 之和等于目标数 9。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 2。返回 [1, 2]。

### 示例 2:

输入: numbers = [2, 3, 4], target = 6

输出: [1, 3]

解释: 2 与 4 之和等于目标数 6。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 3。返回 [1, 3]。

### 示例 3:

输入: numbers = [-1, 0], target = -1

输出: [1, 2]

解释: -1 与 0 之和等于目标数 -1。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 2。返回 [1, 2]。

### 提示:

- $2 \leq \text{numbers.length} \leq 3 * 10^4$
- $-1000 \leq \text{numbers}[i] \leq 1000$
- **numbers** 按 **非递减顺序** 排列
- $-1000 \leq \text{target} \leq 1000$
- **仅存在一个有效答案**

只要数组有序，就应该想到双指针技巧。这道题的解法有点类似二分查找，通过调节 **left** 和 **right** 就可以调整 **sum** 的大小：

```
class Solution {  
    public int[] twoSum(int[] numbers, int target) {  
        // 左一右两个指针相向而行  
        int left = 0, right = numbers.length - 1;  
        while (left < right) {  
            int sum = numbers[left] + numbers[right];  
            if (sum == target) {  
                // 题目要求的索引是从 1 开始的  
                return new int[]{left + 1, right + 1};  
            } else if (sum < target) {  
                // 让 sum 大一点  
                left++;  
            } else if (sum > target) {  
                // 让 sum 小一点  
                right--;  
            }  
        }  
        return new int[]{-1, -1};  
    }  
}
```

我在另一篇文章 [一个函数秒杀所有 nSum 问题](#) 中也运用类似的左右指针技巧给出了 [nSum](#) 问题的一种通用思路，本质上利用的也是双指针技巧。

## 反转数组

一般编程语言都会提供 [reverse](#) 函数，其实这个函数的原理非常简单，力扣第 344 题「反转字符串」就是类似的需求，让你反转一个 [char\[\]](#) 类型的字符数组，我们直接看代码吧：

```
void reverseString(char[] s) {
    // 一左一右两个指针相向而行
    int left = 0, right = s.length - 1;
    while (left < right) {
        // 交换 s[left] 和 s[right]
        char temp = s[left];
        s[left] = s[right];
        s[right] = temp;
        left++;
        right--;
    }
}
```

关于数组翻转的更多进阶问题，可以参见 [二维数组的花式遍历](#)。

## 回文串判断

回文串就是正着读和反着读都一样的字符串。比如说字符串 [aba](#) 和 [abba](#) 都是回文串，因为它们对称，反过来还是和本身一样；反之，字符串 [abac](#) 就不是回文串。

现在你应该能感觉到回文串问题和左右指针肯定有密切的联系，比如让你判断一个字符串是不是回文串，你可以写出下面这段代码：

```
boolean isPalindrome(String s) {
    // 一左一右两个指针相向而行
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

那接下来我提升一点难度，给你一个字符串，让你用双指针技巧从中找出最长的回文串，你会做吗？

这就是力扣第 5 题「最长回文子串」：

### ▼ 5. 最长回文子串 [Leetcode | 力扣](#)

给你一个字符串 [s](#)，找到 [s](#) 中最长的回文子串。

示例 1：

```
输入: s = "babad"
输出: "bab"
解释: "aba" 同样是符合题意的答案。
```

## 示例 2:

```
输入: s = "cbbd"
输出: "bb"
```

### 提示:

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母组成

函数签名如下:

```
String longestPalindrome(String s);
```

找回文串的难点在于，回文串的的长度可能是奇数也可能是偶数，解决该问题的核心是从中心向两端扩散的双指针技巧。

如果回文串的长度为奇数，则它有一个中心字符；如果回文串的长度为偶数，则可以认为它有两个中心字符。所以我们可以在实现这样一个函数：

```
// 在 s 中寻找以 s[l] 和 s[r] 为中心的最长回文串
String palindrome(String s, int l, int r) {
    // 防止索引越界
    while (l >= 0 && r < s.length()
        && s.charAt(l) == s.charAt(r)) {
        // 双指针，向两边展开
        l--; r++;
    }
    // 返回以 s[l] 和 s[r] 为中心的最长回文串
    return s.substring(l + 1, r);
}
```

这样，如果输入相同的 `l` 和 `r`，就相当于寻找长度为奇数的回文串，如果输入相邻的 `l` 和 `r`，则相当于寻找长度为偶数的回文串。

那么回到最长回文串的问题，解法的大致思路就是：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串
    找到以 s[i] 和 s[i+1] 为中心的回文串
    更新答案
```

翻译成代码，就可以解决最长回文子串这个问题：

```

String longestPalindrome(String s) {
    String res = "";
    for (int i = 0; i < s.length(); i++) {
        // 以 s[i] 为中心的最长回文子串
        String s1 = palindrome(s, i, i);
        // 以 s[i] 和 s[i+1] 为中心的最长回文子串
        String s2 = palindrome(s, i, i + 1);
        // res = longest(res, s1, s2)
        res = res.length() > s1.length() ? res : s1;
        res = res.length() > s2.length() ? res : s2;
    }
    return res;
}

```

### ▶ 😊 代码可视化动画😊

你应该能发现最长回文子串使用的左右指针和之前题目的左右指针有一些不同：之前的左右指针都是从两端向中间相向而行，而回文子串问题则是让左右指针从中心向两端扩展。不过这种情况也就回文串这类问题会遇到，所以我也把它归为左右指针了。

到这里，数组相关的双指针技巧就全部讲完了，这些技巧的更多扩展延伸见 [更多数组双指针经典高频题](#)。

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1. Two Sum</a>	1. 两数之和	
<a href="#">125. Valid Palindrome</a>	125. 验证回文串	
<a href="#">131. Palindrome Partitioning</a>	131. 分割回文串	
<a href="#">267. Palindrome Permutation II</a>	267. 回文排列 II	
<a href="#">281. Zigzag Iterator</a>	281. 锯齿迭代器	
<a href="#">42. Trapping Rain Water</a>	42. 接雨水	
<a href="#">543. Diameter of Binary Tree</a>	543. 二叉树的直径	
<a href="#">658. Find K Closest Elements</a>	658. 找到 K 个最接近的元素	
<a href="#">75. Sort Colors</a>	75. 颜色分类	
<a href="#">80. Remove Duplicates from Sorted Array II</a>	80. 删除有序数组中的重复项 II	
<a href="#">82. Remove Duplicates from Sorted List II</a>	82. 删除排序链表中的重复元素 II	
<a href="#">9. Palindrome Number</a>	9. 回文数	
-	剑指 Offer 21. 调整数组顺序使奇数位于偶数前面	
-	剑指 Offer 57. 和为s的两个数字	
-	剑指 Offer II 018. 有效的回文	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 滑动窗口算法核心代码模板



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">3. Longest Substring Without Repeating Characters</a>	<a href="#">3. 无重复字符的最长子串</a>	
<a href="#">567. Permutation in String</a>	<a href="#">567. 字符串的排列</a>	
<a href="#">76. Minimum Window Substring</a>	<a href="#">76. 最小覆盖子串</a>	
<a href="#">438. Find All Anagrams in a String</a>	<a href="#">438. 找到字符串中所有字母异位词</a>	

阅读本文前，你需要先学习：

- [数组基础](#)
- [数组实现](#)

关于双指针的快慢指针和左右指针的用法，可以参见前文 [双指针技巧汇总](#)，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。并总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

## 滑动窗口框架概览

滑动窗口算法技巧主要用来解决子数组问题，比如让你寻找符合某个条件的最长/最短子数组。

如果用暴力解的话，你需要嵌套 for 循环这样穷举所有子数组，时间复杂度是  $O(N^2)$ ：

```
for (int i = 0; i < nums.length; i++) {
    for (int j = i; j < nums.length; j++) {
        // nums[i, j] 是一个子数组
    }
}
```

滑动窗口算法技巧的思路也不难，就是维护一个窗口，不断滑动，然后更新答案，该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < nums.size()) {
    // 增大窗口
    window.addLast(nums[right]);
    right++;
}
```

```
while (window needs shrink) {
    // 缩小窗口
    window.removeFirst(nums[left]);
    left++;
}
}
```

基于滑动窗口算法框架写出的代码，时间复杂度是  $O(N)$ ，比嵌套 for 循环的暴力解法效率高。

肯定有读者要问了，你这个滑动窗口框架不也用了一个嵌套 while 循环？为啥复杂度是  $O(N)$  呢？

简单说，指针 `left`, `right` 不会回退（它们的值只增不减），所以字符串/数组中的每个元素都只会进入窗口一次，然后被移出窗口一次，不会说有某些元素多次进入和离开窗口，所以算法的时间复杂度就和字符串/数组的长度成正比。

反观嵌套 for 循环的暴力解法，那个 `j` 会回退，所以某些元素会进入和离开窗口多次，所以时间复杂度就是  $O(N^2)$  了。

我在 [算法时空复杂度分析实用指南](#) 有具体教大家如何从理论上估算时间空间复杂度，这里就不展开了。

这个问题本身就是错误的，**滑动窗口并不能穷举出所有子串**。要想穷举出所有子串，必须用那个嵌套 for 循环。

然而对于某些题目，并不需要穷举所有子串，就能找到题目想要的答案。滑动窗口就是这种场景下的一套算法模板，帮你对穷举过程进行剪枝优化，避免冗余计算。

所以在 [算法的本质](#) 中我把滑动窗口算法归为「如何聪明地穷举」一类。

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。即便你明白了这些细节，代码也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，保证不会出 bug。

因为本文的例题大多是子串相关的题目，字符串实际上就是数组，所以我就把输入设置成字符串了。你做题的时候根据具体题目自行变通即可：

```
// 滑动窗口算法伪码框架
void slidingWindow(String s) {
    // 用合适的数据结构记录窗口中的数据，根据具体场景变通
    // 比如说，我想记录窗口中元素出现的次数，就用 map
    // 如果我想记录窗口中的元素和，就可以只用一个 int
    Object window = ...

    int left = 0, right = 0;
    while (right < s.length()) {
        // c 是将移入窗口的字符
        char c = s[right];
        window.add(c)
        // 增大窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...
    }
}
```

```

// *** debug 输出的位置 ***
// 注意在最终的解法代码中不要 print
// 因为 IO 操作很耗时，可能导致超时
printf("window: [%d, %d]\n", left, right);
// *****

// 判断左侧窗口是否要收缩
while (left < right && window needs shrink) {
    // d 是将移出窗口的字符
    char d = s[left];
    window.remove(d)
    // 缩小窗口
    left++;
    // 进行窗口内数据的一系列更新
    ...
}
}
}

```

框架中两处 `...` 表示的更新窗口数据的地方，在具体的题目中，你需要做的就是往这里面填代码逻辑。而且，这两个 `...` 处的操作分别是扩大和缩小窗口的更新操作，等会你会发现它们操作是完全对称的。

说句题外话，有些读者评论我这个框架，说散列表速度慢，不如用数组代替散列表；还有些人喜欢把代码写得特别短小，说我这样代码太多余，速度不够快。我的意见是，算法主要看时间复杂度，你能确保自己的时间复杂度最优就行了。至于 LeetCode 的运行速度，那个有点玄学，只要不是慢的离谱就没啥问题，根本不值得你从编译层面优化，不要舍本逐末……

再说，我的算法教程重点在于算法思想，你先做到能把框架思维运用自如，然后随便你魔改代码好吧，保你怎么写都能写对。

言归正传，下面就直接上四道力扣原题来套这个框架，其中第一道题会详细说明其原理，后面四道就直接闭眼睛秒杀了。

## 一、最小覆盖子串

先来看看力扣第 76 题「最小覆盖子串」难度 Hard：

### ▼ 76. 最小覆盖子串 [Leetcode | 力扣](#)

给你一个字符串 `s`、一个字符串 `t`。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""`。

**注意：**

- 对于 `t` 中重复字符，我们寻找的子字符串中该字符数量必须不少于 `t` 中该字符数量。
- 如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

**示例 1：**

```

输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 %%%A%%%、%%%B%%% 和 %%%C%%%。

```

**示例 2：**

输入: s = "a", t = "a"  
输出: "a"  
解释: 整个字符串 s 是最小覆盖子串。

### 示例 3:

输入: s = "a", t = "aa"  
输出: ""  
解释: t 中两个字符 %%%a%%% 均应包含在 s 的子串中,  
因此没有符合条件的子字符串, 返回空字符串。

### 提示:

- m == s.length
- n == t.length
- 1 <= m, n <= 10<sup>5</sup>
- s 和 t 由英文字母组成

进阶: 你能设计一个在 O(m+n) 时间内解决此问题的算法吗?

就是说要在 S(source) 中找到包含 T(target) 中全部字母的一个子串, 且这个子串一定是所有可能子串中最短的。

如果我们使用暴力解法, 代码大概是这样的:

```
for (int i = 0; i < s.length(); i++)
    for (int j = i + 1; j < s.length(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

思路很直接, 但是显然, 这个算法的复杂度肯定大于 O(N<sup>2</sup>) 了, 不好。

滑动窗口算法的思路是这样:

1、我们在字符串 S 中使用双指针中的左右指针技巧, 初始化 left = right = 0, 把索引左闭右开区间 [left, right) 称为一个「窗口」。

理论上你可以设计两端都开或者两端都闭的区间, 但设计为左闭右开区间是最方便处理的。

因为这样初始化 left = right = 0 时区间 [0, 0) 中没有元素, 但只要让 right 向右移动 (扩大) 一位, 区间 [0, 1) 就包含一个元素 0 了。

如果你设置为两端都开的区间, 那么让 right 向右移动一位后区间 (0, 1) 仍然没有元素; 如果你设置为两端都闭的区间, 那么初始区间 [0, 0] 就包含了一个元素。这两种情况都会给边界处理带来不必要的麻烦。

2、我们先不断地增加 right 指针扩大窗口 [left, right), 直到窗口中的字符串符合要求 (包含了 T 中的所有字符)。

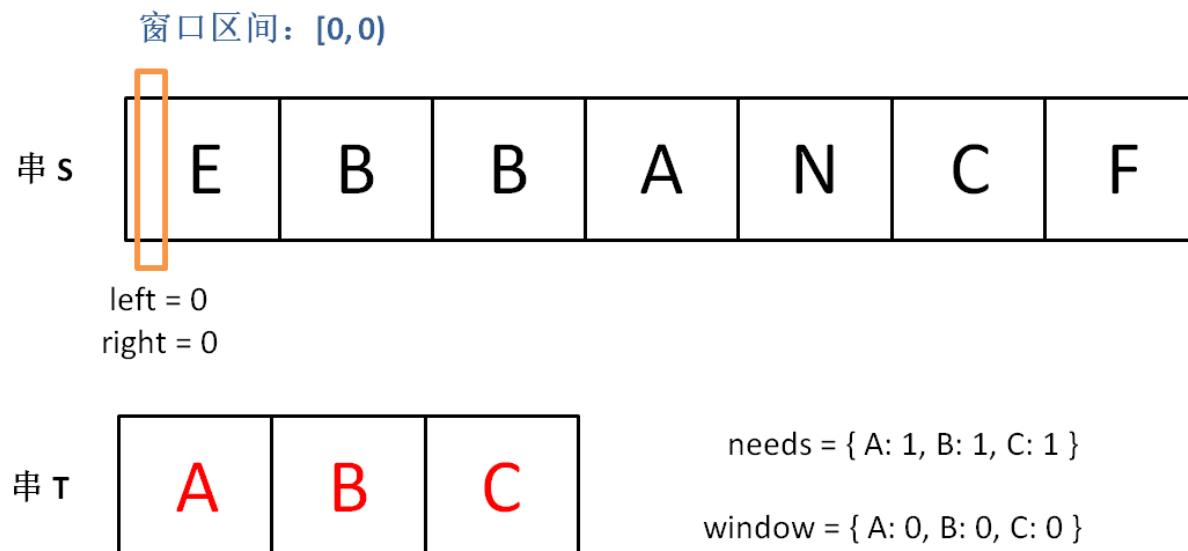
3、此时, 我们停止增加 right, 转而不断增加 left 指针缩小窗口 [left, right), 直到窗口中的字符串不再符合要求 (不包含 T 中的所有字符了)。同时, 每次增加 left, 我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 `right` 到达字符串 `S` 的尽头。

这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，就好像一条毛毛虫，一伸一缩，不断向右滑动，这就是「滑动窗口」这个名字的来历。

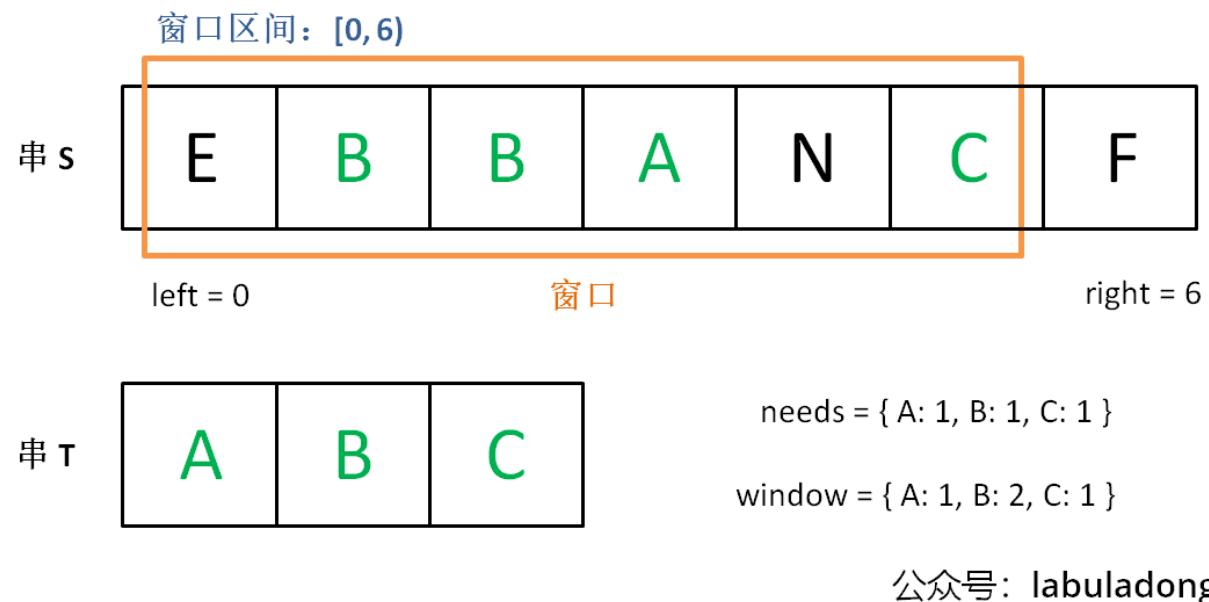
下面画图理解一下，`needs` 和 `window` 相当于计数器，分别记录 `T` 中字符出现次数和「窗口」中的相应字符的出现次数。

初始状态：



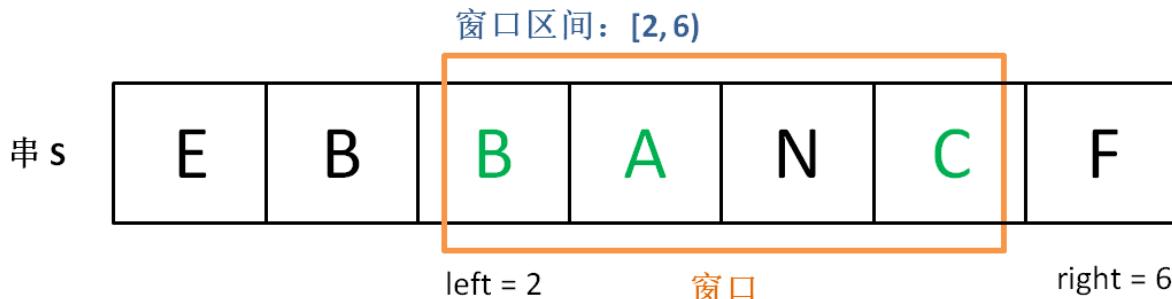
公众号：labuladong

增加 `right`，直到窗口  $[left, right]$  包含了 `T` 中所有字符：



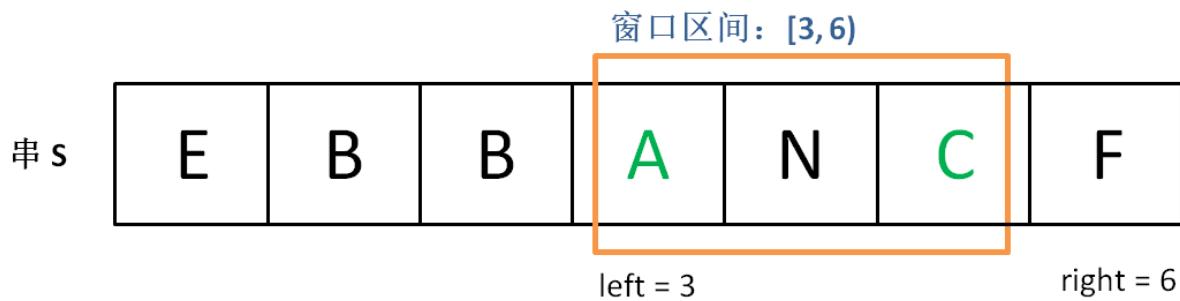
公众号：labuladong

现在开始增加 `left`，缩小窗口  $[left, right]$ ：



公众号: labuladong

直到窗口中的字符串不再符合要求, `left` 不再继续移动:



公众号: labuladong

之后重复上述过程, 先移动 `right`, 再移动 `left`…… 直到 `right` 指针到达字符串 `S` 的末端, 算法结束。

如果你能够理解上述过程, 恭喜, 你已经完全掌握了滑动窗口算法思想。现在我们来看看这个滑动窗口代码框架怎么用:

首先, 初始化 `window` 和 `need` 两个哈希表, 记录窗口中的字符和需要凑齐的字符:

```
// 记录 window 中的字符出现次数
HashMap<Character, Integer> window = new HashMap<>();
// 记录所需的字符出现次数
HashMap<Character, Integer> need = new HashMap<>();
for (int i = 0; i < t.length(); i++) {
    char c = t.charAt(i);
    need.put(c, need.getOrDefault(c, 0) + 1);
}
```

然后, 使用 `left` 和 `right` 变量初始化窗口的两端, 不要忘了, 区间 `[left, right)` 是左闭右开的, 所以初始情况下窗口没有包含任何元素:

```

int left = 0, right = 0;
int valid = 0;
while (right < s.length()) {
    // c 是将移入窗口的字符
    char c = s.charAt(right);
    // 右移窗口
    right++;
    // 进行窗口内数据的一系列更新
    ...
}

```

其中 `valid` 变量表示窗口中满足 `need` 条件的字符个数，如果 `valid` 和 `need.size()` 的大小相同，则说明窗口已满足条件，已经完全覆盖了串 `T`。

现在开始套模板，只需要思考以下几个问题：

- 1、什么时候应该移动 `right` 扩大窗口？窗口加入字符时，应该更新哪些数据？
- 2、什么时候窗口应该暂停扩大，开始移动 `left` 缩小窗口？从窗口移出字符时，应该更新哪些数据？
- 3、我们要的结果应该在扩大窗口时还是缩小窗口时进行更新？

如果一个字符进入窗口，应该增加 `window` 计数器；如果一个字符将移出窗口的时候，应该减少 `window` 计数器；当 `valid` 满足 `need` 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

下面是完整代码：

```

class Solution {
    public String minWindow(String s, String t) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        // 记录最小覆盖子串的起始索引及长度
        int start = 0, len = Integer.MAX_VALUE;
        while (right < s.length()) {
            // c 是将移入窗口的字符
            char c = s.charAt(right);
            // 扩大窗口
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).equals(need.get(c)))
                    valid++;
            }

            // 判断左侧窗口是否要收缩
            while (valid == need.size()) {
                // 在这里更新最小覆盖子串
                if (right - left < len) {
                    start = left;
                    len = right - left;
                }
                // 缩小窗口
                window.put(s.charAt(left), window.get(s.charAt(left)) - 1);
                if (window.get(s.charAt(left)).equals(need.get(s.charAt(left))))
                    valid--;
                left++;
            }
        }
        return start < len ? s.substring(start, start + len) : "";
    }
}

```

```
        len = right - left;
    }
    // d 是将移出窗口的字符
    char d = s.charAt(left);
    // 缩小窗口
    left++;
    // 进行窗口内数据的一系列更新
    if (need.containsKey(d)) {
        if (window.get(d).equals(need.get(d)))
            valid--;
        window.put(d, window.get(d) - 1);
    }
}
// 返回最小覆盖子串
return len == Integer.MAX_VALUE ? "" : s.substring(start, start + len);
}
```

## ▶ 🎃 代码可视化动画🎃

对 Java 包装类进行比较时要尤为小心，`Integer`, `String` 等类型应该用 `equals` 方法判定相等，而不能直接用等号 `==`，否则会出错。所以在缩小窗口更新数据的时候，不能直接写为 `window.get(d) == need.get(d)`，而要用 `window.get(d).equals(need.get(d))`，之后的题目代码同理。

上面的代码中，当我们发现某个字符在 `window` 的数量满足了 `need` 的需要，就要更新 `valid`，表示有一个字符已经满足要求。而且，你能发现，两次对窗口内数据的更新操作是完全对称的。

当 `valid == need.size()` 时，说明 T 中所有字符已经被覆盖，已经得到一个可行的覆盖子串，现在应该开始收缩窗口了，以便得到「最小覆盖子串」。

移动 `left` 收缩窗口时，窗口内的字符都是可行解，所以应该在收缩窗口的阶段进行最小覆盖子串的更新，以便从可行解中找到长度最短的最终结果。

至此，应该可以完全理解这套框架了，滑动窗口算法又不难，就是细节问题让人烦得很。以后遇到滑动窗口算法，你就按照这框架写代码，保准没有 bug，还省事儿。

下面就直接利用这套框架秒杀几道题吧，你基本上一眼就能看出思路了。

## 二、字符串排列

这是力扣第 567 题「字符串的排列」，难度中等：

### ▼ 567. 字符串的排列 [Leetcode | 力扣](#)

给你两个字符串 `s1` 和 `s2`，写一个函数来判断 `s2` 是否包含 `s1` 的排列。如果是，返回 `true`；否则，返回 `false`。

换句话说，`s1` 的排列之一是 `s2` 的子串。

示例 1：

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: true
解释: s2 包含 s1 的排列之一 ("ba").
```

## 示例 2：

```
输入: s1= "ab" s2 = "eidboaoo"
输出: false
```

### 提示：

- $1 \leq s1.length, s2.length \leq 10^4$
- $s1$  和  $s2$  仅包含小写字母

注意哦，输入的  $s1$  是可以包含重复字符的，所以这个题难度不小。

这种题目，是明显的滑动窗口算法，相当给你一个  $S$  和一个  $T$ ，请问你  $S$  中是否存在一个子串，包含  $T$  中所有字符且不包含其他字符？

首先，先复制粘贴之前的算法框架代码，然后明确刚才提出的几个问题，即可写出这道题的答案：

```
class Solution {
    // 判断 s 中是否存在 t 的排列
    public boolean checkInclusion(String t, String s) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        while (right < s.length()) {
            char c = s.charAt(right);
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).intValue() == need.get(c).intValue())
                    valid++;
            }

            // 判断左侧窗口是否要收缩
            while (right - left >= t.length()) {
                // 在这里判断是否找到了合法的子串
                if (valid == need.size())
                    return true;
                char d = s.charAt(left);
                left++;
                // 进行窗口内数据的一系列更新
                if (need.containsKey(d)) {
                    if (window.get(d).intValue() == need.get(d).intValue())
                        valid--;
                    window.put(d, window.get(d) - 1);
                }
            }
        }
        // 未找到符合条件的子串
        return false;
    }
}
```

```
    }  
}
```

## ▶ 🎃 代码可视化动画🎃

对于这道题的解法代码，基本上和最小覆盖子串一模一样，只需要改变几个地方：

- 1、本题移动 `left` 缩小窗口的时机是窗口大小大于 `t.length()` 时，因为排列嘛，显然长度应该是一样的。
- 2、当发现 `valid == need.size()` 时，就说明窗口中就是一个合法的排列，所以立即返回 `true`。

至于如何处理窗口的扩大和缩小，和最小覆盖子串完全相同。

由于这道题中 `[left, right]` 其实维护的是一个定长的窗口，窗口长度为 `t.length()`。因为定长窗口每次向前滑动时只会移出一个字符，所以完全可以把内层的 `while` 改成 `if`，效果是一样的。

## 三、找所有字母异位词

这是力扣第 438 题「找到字符串中所有字母异位词」，难度中等：

### ▼ 438. 找到字符串中所有字母异位词 [Leetcode](#) | 力扣

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1：

```
输入: s = "cbaebabacd", p = "abc"  
输出: [0,6]  
解释:  
起始索引等于 0 的子串是 "cba"，它是 "abc" 的异位词。  
起始索引等于 6 的子串是 "bac"，它是 "abc" 的异位词。
```

示例 2：

```
输入: s = "abab", p = "ab"  
输出: [0,1,2]  
解释:  
起始索引等于 0 的子串是 "ab"，它是 "ab" 的异位词。  
起始索引等于 1 的子串是 "ba"，它是 "ab" 的异位词。  
起始索引等于 2 的子串是 "ab"，它是 "ab" 的异位词。
```

提示：

- $1 \leq s.length, p.length \leq 3 * 10^4$
- `s` 和 `p` 仅包含小写字母

呵呵，这个所谓的字母异位词，不就是排列吗，搞个高端的说法就能糊弄人了吗？相当于，输入一个串 `S`，一个串 `T`，找到 `S` 中所有 `T` 的排列，返回它们的起始索引。

直接默写一下框架，明确刚才讲的 4 个问题，即可秒杀这道题：

```
class Solution {
    public List<Integer> findAnagrams(String s, String t) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        // 记录结果
        List<Integer> res = new ArrayList<>();
        while (right < s.length()) {
            char c = s.charAt(right);
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).equals(need.get(c))) {
                    valid++;
                }
            }
            // 判断左侧窗口是否要收缩
            while (right - left >= t.length()) {
                // 当窗口符合条件时，把起始索引加入 res
                if (valid == need.size())
                    res.add(left);
                char d = s.charAt(left);
                left++;
                // 进行窗口内数据的一系列更新
                if (need.containsKey(d)) {
                    if (window.get(d).equals(need.get(d))) {
                        valid--;
                    }
                    window.put(d, window.get(d) - 1);
                }
            }
        }
        return res;
    }
}
```

跟寻找字符串的排列一样，只是找到一个合法异位词（排列）之后将起始索引加入 `res` 即可。

▶ 🎃 代码可视化动画🎃

## 四、最长无重复子串

这是力扣第 3 题「无重复字符的最长子串」，难度中等：

▼ 3. 无重复字符的最长子串 [Leetcode | 力扣](#)

给定一个字符串 `s`，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: s = "bbbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: s = "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

提示:

- $0 \leq s.length \leq 5 * 10^4$
- $s$  由英文字母、数字、符号和空格组成

这个题终于有了点新意，不是一套框架就出答案，不过反而更简单了，稍微改一改框架就行了：

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> window = new HashMap<>();
        int left = 0, right = 0;
        // 记录结果
        int res = 0;
        while (right < s.length()) {
            char c = s.charAt(right);
            right++;
            // 进行窗口内数据的一系列更新
            window.put(c, window.getOrDefault(c, 0) + 1);
            // 判断左侧窗口是否要收缩
            while (window.get(c) > 1) {
                char d = s.charAt(left);
                left++;
                // 进行窗口内数据的一系列更新
                window.put(d, window.get(d) - 1);
            }
            // 在这里更新答案
            res = Math.max(res, right - left);
        }
        return res;
    }
}
```

## ▶ 代码可视化动画

这就是变简单了，连 `need` 和 `valid` 都不需要，而且更新窗口内数据也只需要简单的更新计数器 `window` 即可。

当 `window[c]` 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 `left` 缩小窗口了嘛。

唯一需要注意的是，在哪里更新结果 `res` 呢？我们要的是最长无重复子串，哪一个阶段可以保证窗口中的字符串是没有重复的呢？

这里和之前不一样，要在收缩窗口完成后更新 `res`，因为窗口收缩的 while 条件是存在重复元素，换句话说收缩完成后一定保证窗口中没有重复嘛。

好了，滑动窗口算法模板就讲到这里，希望大家能理解其中的思想，记住算法模板并融会贯通。回顾一下，遇到子数组/子串相关的问题，你只要能回答出来以下几个问题，就能运用滑动窗口算法：

- 1、什么时候应该扩大窗口？
- 2、什么时候应该缩小窗口？
- 3、什么时候应该更新答案？

我在 [滑动窗口经典习题](#) 中使用这套思维模式列举了更多经典的习题，旨在强化你对算法的理解和记忆，以后就再也不怕子串、子数组问题了。

## ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1004. Max Consecutive Ones III</a>	<a href="#">1004. 最大连续1的个数 III</a>	
<a href="#">1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit</a>	<a href="#">1438. 绝对差不超过限制的最长连续子数组</a>	
<a href="#">1658. Minimum Operations to Reduce X to Zero</a>	<a href="#">1658. 将 x 减到 0 的最小操作数</a>	
<a href="#">209. Minimum Size Subarray Sum</a>	<a href="#">209. 长度最小的子数组</a>	
<a href="#">219. Contains Duplicate II</a>	<a href="#">219. 存在重复元素 II</a>	
<a href="#">220. Contains Duplicate III</a>	<a href="#">220. 存在重复元素 III</a>	
<a href="#">340. Longest Substring with At Most K Distinct Characters</a>	<a href="#">340. 至多包含 K 个不同字符的最长子串</a>	
<a href="#">395. Longest Substring with At Least K Repeating Characters</a>	<a href="#">395. 至少有 K 个重复字符的最长子串</a>	
<a href="#">424. Longest Repeating Character Replacement</a>	<a href="#">424. 替换后的最长重复字符</a>	
<a href="#">560. Subarray Sum Equals K</a>	<a href="#">560. 和为 K 的子数组</a>	
<a href="#">713. Subarray Product Less Than K</a>	<a href="#">713. 乘积小于 K 的子数组</a>	
<a href="#">862. Shortest Subarray with Sum at Least K</a>	<a href="#">862. 和至少为 K 的最短子数组</a>	
<a href="#">剑指 Offer 48. 最长不含重复字符的子字符串</a>		

LeetCode	力扣	难度
-	剑指 Offer 57 - II. 和为s的连续正数序列	
-	剑指 Offer II 008. 和大于等于 target 的最短子数组	
-	剑指 Offer II 009. 乘积小于 K 的子数组	
-	剑指 Offer II 010. 和为 k 的子数组	
-	剑指 Offer II 014. 字符串中的变位词	
-	剑指 Offer II 015. 字符串中的所有变位词	
-	剑指 Offer II 016. 不含重复字符的最长子字符串	
-	剑指 Offer II 017. 含有所有字符的最短字符串	
-	剑指 Offer II 057. 值和下标之差都在给定的范围内	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

## 二分搜索算法核心代码模板



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
34. Find First and Last Position of Element in Sorted Array	34. 在排序数组中查找元素的第一个和最后一个位置	
-	剑指 Offer 53 - I. 在排序数组中查找数字 I	
704. Binary Search	704. 二分查找	

阅读本文前，你需要先学习：

- 数组基础
- 数组实现

tip：本文有视频版：[二分搜索核心框架套路](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文是旧文 [二分搜索详解](#) 的修订版，添加了对二分搜索算法更详细的分析。

先给大家讲个笑话乐呵一下：

有一天阿东到图书馆借了  $N$  本书，出图书馆的时候，警报响了，于是保安把阿东拦下，要检查一下哪本书没有登记出借。阿东正准备把每一本书在报警器下过一下，以找出引发警报的书，但是保安露出不屑的眼神：你连二分查找都不会吗？

于是保安把书分成两堆，让第一堆过一下报警器，报警器响，这说明引起报警的书包含在里面；于是再把这堆书分成两堆，把第一堆过一下报警器，报警器又响，继续分成两堆……

最终，检测了  $\log N$  次之后，保安成功的找到了那本引起警报的书，露出了得意和嘲讽的笑容。于是阿东背着剩下的书走了。

从此，图书馆丢了  $N - 1$  本书（手动狗头）。

二分查找并不简单，Knuth 大佬（发明 KMP 算法的那位）都说二分查找：**思路很简单，细节是魔鬼**。很多人喜欢拿整型溢出的 bug 说事儿，但是二分查找真正的坑根本就不是那个细节问题，而是在于到底要给  $mid$  加一还是减一，while 里到底用  $<=$  还是  $<$ 。

你要是没有正确理解这些细节，写二分肯定就是玄学编程，有没有 bug 只能靠菩萨保佑，谁写谁知道。

本文就来探究几个最常用的二分查找场景：寻找一个数、寻找左侧边界、寻找右侧边界。而且，我们就是要深入细节，比如不等号是否应该带等号， $mid$  是否应该加一等等。分析这些细节的差异以及出现这些差异的原因，保证你能灵活准确地

写出正确的二分查找算法。

另外再声明一下，对于二分搜索的每一个场景，本文还会探讨多种代码写法，目的是为了让你理解出现这些细微差异的本质原因，最起码你看到别人的代码时不会懵逼。实际上这些写法没有优劣之分，你喜欢哪种就用哪种好了。

## 零、二分查找框架

```
int binarySearch(int[] nums, int target) {  
    int left = 0, right = ...;  
  
    while(...) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] == target) {  
            ...  
        } else if (nums[mid] < target) {  
            left = ...  
        } else if (nums[mid] > target) {  
            right = ...  
        }  
    }  
    return ...;  
}
```

分析二分查找的一个技巧是：不要出现 `else`，而是把所有情况用 `else if` 写清楚，这样可以清楚地展现所有细节。本文都会使用 `else if`，旨在讲清楚，读者理解后可自行简化。

其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

另外提前说明一下，计算 `mid` 时需要防止溢出，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大，直接相加导致溢出的情况。

## 一、寻找一个数（基本的二分搜索）

这个场景是最简单的，可能也是大家最熟悉的，即搜索一个数，如果存在，返回其索引，否则返回 -1。

```
int binarySearch(int[] nums, int target) {  
    int left = 0;  
    // 注意  
    int right = nums.length - 1;  
  
    while(left <= right) {  
        int mid = left + (right - left) / 2;  
        if(nums[mid] == target)  
            return mid;  
        else if (nums[mid] < target)  
            // 注意  
            left = mid + 1;  
        else if (nums[mid] > target)  
            // 注意  
            right = mid - 1;  
    }  
    return -1;  
}
```

## ▶ 😊 代码可视化动画😊

这段代码可以解决力扣第 704 题「二分查找」，但我们深入探讨一下其中的细节。

### 为什么 while 循环的条件是 `<=` 而不是 `<`？

答：因为初始化 `right` 的赋值是 `nums.length - 1`，即最后一个元素的索引，而不是 `nums.length`。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 `[left, right]`，后者相当于左闭右开区间 `[left, right)`。因为索引大小为 `nums.length` 是越界的，所以我们把 `right` 这一边视为开区间。

我们这个算法中使用的是前者 `[left, right]` 两端都闭的区间。这个区间其实就是每次进行搜索的区间。

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```
if(nums[mid] == target)
    return mid;
```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？**搜索区间为空的时候应该终止**，意味着你没得找了，就等于没找到嘛。

`while(left <= right)` 的终止条件是 `left == right + 1`，写成区间的形式就是 `[right + 1, right]`，或者带个具体的数字进去 `[3, 2]`，可见**这时候区间为空**，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的，直接返回 -1 即可。

`while(left < right)` 的终止条件是 `left == right`，写成区间的形式就是 `[right, right]`，或者带个具体的数字进去 `[2, 2]`，**这时候区间非空**，还有一个数 2，但此时 while 循环终止了。也就是说区间 `[2, 2]` 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 `while(left < right)` 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
// ...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;
```

### 为什么是 `left = mid + 1, right = mid - 1`？

为什么 `left = mid + 1, right = mid - 1`？我看有的代码是 `right = mid` 或者 `left = mid`，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 `[left, right]`。那么当我们发现索引 `mid` 不是要找的 `target` 时，下一步应该去搜索哪里呢？

当然是去搜索区间 `[left, mid-1]` 或者区间 `[mid+1, right]` 对不对？因为 `mid` 已经搜索过，应该从搜索区间中去除。

### 此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

比如说给你有序数组 `nums = [1,2,2,2,3]`, `target` 为 2, 此算法返回的索引是 2, 没错。但是如果我想得到 `target` 的左侧边界，即索引 1，或者我想得到 `target` 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见，你也许会说，找到一个 `target`，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

## 二、寻找左侧边界的二分搜索

以下是最常见的代码形式，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {
    int left = 0;
    // 注意
    int right = nums.length;

    // 注意
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 注意
            right = mid;
        }
    }
    return left;
}
```

为什么 `while` 中是 `<` 而不是 `<=`？

答：用相同的方法分析，因为 `right = nums.length` 而不是 `nums.length - 1`。因此每次循环的「搜索区间」是 `[left, right)` 左闭右开。

`while(left < right)` 终止的条件是 `left == right`，此时搜索区间 `[left, left)` 为空，所以可以正确终止。

这里先要说一个搜索左右侧边界和上面这个算法的一个区别，也是很多读者问的：刚才的 `right` 不是 `nums.length - 1` 吗，为啥这里非要写成 `nums.length` 使得「搜索区间」变成左闭右开呢？

因为对于搜索左右侧边界的二分查找，这种写法比较普遍，我就拿这种写法举例，保证你以后遇到这类代码可以理解。你非要用两端都闭的写法反而更简单，我会在后面写相关的代码，把三种二分搜索都用一种两端都闭的写法统一起来，你耐心往后看就行了。

`target` 不存在时返回什么？

如果 `nums` 中不存在 `target` 这个值，计算出来的这个索引含义是什么？如果我想让它返回 -1，怎么做？

这是一个很好且很重要的问题，你把这个地方理解了，在二分搜索的实际应用场景中就不会懵逼。

直接说结论：如果 `target` 不存在，搜索左侧边界的二分搜索返回的索引是大于 `target` 的最小索引。

举个例子，`nums = [2,3,5,7]`, `target = 4`, `left_bound` 函数返回值是 2，因为元素 5 是大于 4 的最小元素。

有点绕晕了是吧？这个 `left_bound` 函数明明是搜索左边界，但是当 `target` 不存在的时候，却返回的是大于 `target` 的最小索引。这个结论不用死记，你要是拿不准，简单举个例子就能得到这个结论了。

所以跟你说二分搜索这个东西思路很简单，细节是魔鬼嘛，里面的坑太多了。要是真想考你，总有办法可以把你考到怀疑人生。

不是我故意把代码模板总结的这么复杂，而是二分搜索本身就很复杂，这些细节是不可能绕开的，如果你之前没有了解过这些细节，只能说明你之前学得不扎实。就算不用我总结的模板，你也必须搞清楚当 `target` 不存在时算法的行为，否则出了 bug 你都不知道咋改，真有这么严重。

话说回来，`left_bound` 的这个行为有一个好处。比方说现在让你写一个 `floor` 函数，就可以直接用 `left_bound` 函数来实现：

```
// 在一个有序数组中，找到「小于 target 的最大元素的索引」
// 比如说输入 nums = [1,2,2,2,3], target = 2, 函数返回 0，因为 1 是小于 2 的最大元素。
// 再比如输入 nums = [1,2,3,5,6], target = 4, 函数返回 2，因为 3 是小于 4 的最大元素。
int floor(int[] nums, int target) {
    // 当 target 不存在，比如输入 [4,6,8,10], target = 7
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 7 的最大元素
    // 当 target 存在，比如输入 [4,6,8,8,8,10], target = 8
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 8 的最大元素
    return left_bound(nums, target) - 1;
}
```

最后，我的建议是，如果你必须手写二分代码，那么你一定要了解清楚代码的种种行为，本文总结的框架就是在帮你理清这里面的细节。如果非必要，不要自己手写，尽肯能用编程语言提供的标准库函数，可以节约时间，而且标准库函数的行为在文档里都有明确的说明，不容易出错。

如果想让 `target` 不存在时返回 -1 其实很简单，在返回的时候额外判断一下 `nums[left]` 是否等于 `target` 就行了，如果不等于，就说明 `target` 不存在。需要注意的是，访问数组索引之前要保证索引不越界：

```
while (left < right) {
    // ...
}
// 如果索引越界，说明数组中无目标元素，返回 -1
if (left < 0 || left >= nums.length) {
    return -1;
}
// 提示：其实上面的 if 中 left < 0 这个判断可以省略，因为对于这个算法，left 不可能小于 0
// 你看这个算法执行的逻辑，left 初始化就是 0，且只可能一直往右走，那么只可能在右侧越界
// 不过我这里就同时判断了，因为在访问数组索引之前保证索引在左右两端都不越界是一个好习惯，没有坏处
// 另一个好处是让二分的模板更统一，降低你的记忆成本，因为等会儿寻找右边界的时候也有类似的出界判断

// 判断一下 nums[left] 是不是 target
return nums[left] == target ? left : -1;
```

为什么是 `left = mid + 1` 和 `right = mid`？

为什么 `left = mid + 1, right = mid`? 和之前的算法不一样?

答: 这个很好解释, 因为我们的「搜索区间」是 `[left, right)` 左闭右开, 所以当 `nums[mid]` 被检测之后, 下一步应该去 `mid` 的左侧或者右侧区间搜索, 即 `[left, mid)` 或 `[mid + 1, right)`。

为什么该算法能够搜索左侧边界?

答: 关键在于对于 `nums[mid] == target` 这种情况的处理:

```
if (nums[mid] == target)
    right = mid;
```

可见, 找到 `target` 时不要立即返回, 而是缩小「搜索区间」的上界 `right`, 在区间 `[left, mid)` 中继续搜索, 即不断向左收缩, 达到锁定左侧边界的目的。

为什么返回 `left` 而不是 `right`?

答: 都是一样的, 因为 `while` 终止的条件是 `left == right`。

能否统一成两端都闭的搜索区间?

能不能想办法把 `right` 变成 `nums.length - 1`, 也就是继续使用两边都闭的「搜索区间」? 这样就可以和第一种二分搜索在某种程度上统一起来了。

答: 当然可以, 只要你明白了「搜索区间」这个概念, 就能有效避免漏掉元素, 随便你怎么改都行。下面我们严格根据逻辑来修改:

因为你非要让搜索区间两端都闭, 所以 `right` 应该初始化为 `nums.length - 1`, `while` 的终止条件应该是 `left == right + 1`, 也就是其中应该用 `<=`:

```
int left_bound(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // if else ...
    }
}
```

因为搜索区间是两端都闭的, 且现在是搜索左侧边界, 所以 `left` 和 `right` 的更新逻辑如下:

```
if (nums[mid] < target) {
    // 搜索区间变为 [mid+1, right]
    left = mid + 1;
} else if (nums[mid] > target) {
    // 搜索区间变为 [left, mid-1]
    right = mid - 1;
} else if (nums[mid] == target) {
    // 收缩右侧边界
    right = mid - 1;
}
```

和刚才相同，如果想在找不到 `target` 的时候返回 `-1`，那么检查一下 `nums[left]` 和 `target` 是否相等即可：

```
// 此时 target 比所有数都大，返回 -1
if (left == nums.length) return -1;
// 判断一下 nums[left] 是不是 target
return nums[left] == target ? left : -1;
```

至此，整个算法就写完了，完整代码如下：

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 判断 target 是否存在于 nums 中
    // 如果越界，target 肯定不存在，返回 -1
    if (left < 0 || left >= nums.length) {
        return -1;
    }
    // 判断一下 nums[left] 是不是 target
    return nums[left] == target ? left : -1;
}
```

### ▶ 🔍 代码可视化动画

这样就和第一种二分搜索算法统一了，都是两端都闭的「搜索区间」，而且最后返回的也是 `left` 变量的值。只要把住二分搜索的逻辑，两种形式大家看自己喜欢哪种记哪种吧。

## 三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 注意
            left = mid + 1;
        } else if (nums[mid] < target) {
```

```
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid;
    }
}
// 注意
return left - 1;
}
```

## 为什么这个算法能够找到右侧边界？

答：类似地，关键点还是这里：

```
if (nums[mid] == target) {
    left = mid + 1;
}
```

当 `nums[mid] == target` 时，不要立即返回，而是增大「搜索区间」的左边界 `left`，使得区间不断向右靠拢，达到锁定右侧边界的目的。

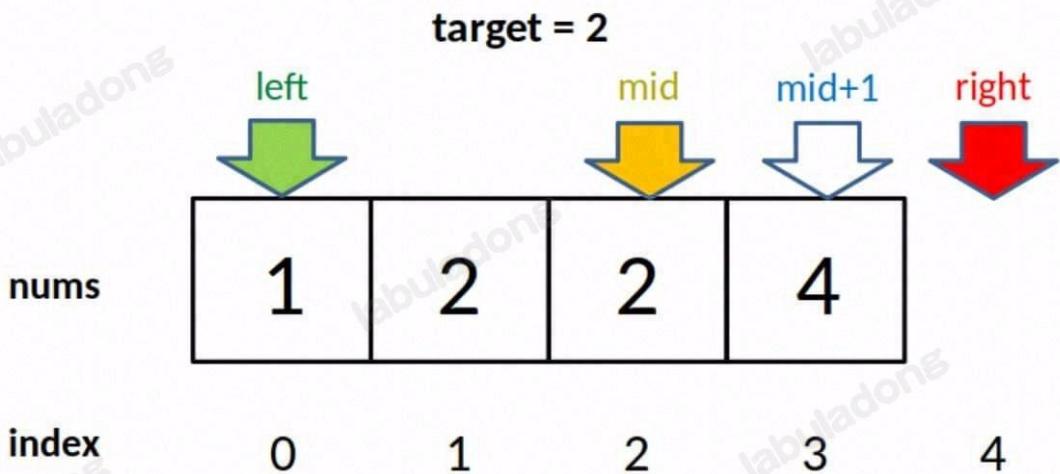
## 为什么返回 `left - 1`？

为什么最后返回 `left - 1` 而不像左侧边界的函数，返回 `left`？而且我觉得这里既然是搜索右侧边界，应该返回 `right` 才对。

答：首先，`while` 循环的终止条件是 `left == right`，所以 `left` 和 `right` 是一样的，你非要体现右侧的特点，返回 `right - 1` 好了。

至于为什么要减一，这是搜索右侧边界的一个特殊点，关键在锁定右边界时的这个条件判断：

```
// 增大 left，锁定右侧边界
if (nums[mid] == target) {
    left = mid + 1;
    // 这样想：mid = left - 1
}
```



公众号：labuladong

因为我们对 `left` 的更新必须是 `left = mid + 1`, 就是说 while 循环结束时, `nums[left]` 一定不等于 `target` 了, 而 `nums[left-1]` 可能是 `target`。

至于为什么 `left` 的更新必须是 `left = mid + 1`, 当然是为了把 `nums[mid]` 排除出搜索区间, 这里就不再赘述。

如果 `target` 不存在时返回什么?

如果 `nums` 中不存在 `target` 这个值, 计算出来的这个索引含义是什么? 如果我想让它返回 -1, 怎么做?

直接说结论, 和前面讲的 `left_bound` 相反: 如果 `target` 不存在, 搜索右侧边界的二分搜索返回的索引是小于 `target` 的最大索引。

这个结论不用死记, 你要是拿不准, 简单举个例子就能得到这个结论了。比如 `nums = [2,3,5,7], target = 4`, `right_bound` 函数返回值是 1, 因为元素 3 是小于 4 的最大元素。

与前面的建议相同, 考虑到二分搜索代码细节的复杂性, 如果非必要, 不要自己手写, 尽肯能用编程语言提供的标准库函数。

如果你想在 `target` 不存在时返回 -1, 很简单, 只要在最后判断一下 `nums[left-1]` 是不是 `target` 就行了, 类似之前的左侧边界搜索, 做一点额外的判断即可:

```

while (left < right) {
    // ...
}
// 判断 target 是否存在于 nums 中
// left - 1 索引越界的话 target 肯定不存在
if (left - 1 < 0 || left - 1 >= nums.length) {
    return -1;
}
// 判断一下 nums[left - 1] 是不是 target
return nums[left - 1] == target ? (left - 1) : -1;

```

4、是否也可以把这个算法的「搜索区间」也统一成两端都闭的形式呢？这样这三个写法就完全统一了，以后就可以闭着眼睛写出来了。

答：当然可以，类似搜索左侧边界的统一写法，其实只要改两个地方就行了：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩左侧边界即可
            left = mid + 1;
        }
    }
    // 最后改成返回 left - 1
    if (left - 1 < 0 || left - 1 >= nums.length) {
        return -1;
    }
    return nums[left - 1] == target ? (left - 1) : -1;
}
```

#### ► 🌟 代码可视化动画🌟

当然，由于 while 的结束条件为 `right == left - 1`，所以你把上述代码中的 `left - 1` 都改成 `right` 也没有问题，这样可能更有利于看出来这是在「搜索右侧边界」：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩左侧边界即可
            left = mid + 1;
        }
    }
    // 最后改成返回 right
    if (right < 0 || right >= nums.length) {
        return -1;
    }
    return nums[right] == target ? right : -1;
}
```

至此，搜索右侧边界的二分查找的两种写法也完成了，其实将「搜索区间」统一成两端都闭反而更容易记忆，你说是吧？

## 四、逻辑统一

有了搜索左右边界的二分搜索，你可以去解决力扣第 34 题「在排序数组中查找元素的第一个和最后一个位置」。接下来梳理一下这些细节差异的因果逻辑：

### 第一个，最基本的二分查找算法：

因为我们初始化 `right = nums.length - 1`  
所以决定了我们的「搜索区间」是 `[left, right]`  
所以决定了 `while (left <= right)`  
同时也决定了 `left = mid+1` 和 `right = mid-1`

因为我们只需找到一个 `target` 的索引即可  
所以当 `nums[mid] == target` 时可以立即返回

### 第二个，寻找左侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最左侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧右侧边界以锁定左侧边界

### 第三个，寻找右侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最右侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 `left = mid + 1`  
所以最后无论返回 `left` 还是 `right`，必须减一

对于寻找左右边界的二分搜索，比较常见的手法是使用左闭右开的「搜索区间」，我们还根据逻辑将「搜索区间」全都统一成了两端都闭，便于记忆，只要修改两处即可变化出三种写法：

```
int binary_search(int[] nums, int target) {  
    int left = 0, right = nums.length - 1;  
    while(left <= right) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] < target) {  
            left = mid + 1;  
        } else if (nums[mid] > target) {  
            right = mid - 1;  
        } else if (nums[mid] == target) {  
            // 直接返回
```

```

        return mid;
    }
}
// 直接返回
return -1;
}

int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定左侧边界
            right = mid - 1;
        }
    }
    // 判断 target 是否存在于 nums 中
    if (left < 0 || left >= nums.length) {
        return -1;
    }
    // 判断一下 nums[left] 是不是 target
    return nums[left] == target ? left : -1;
}

int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定右侧边界
            left = mid + 1;
        }
    }
    // 由于 while 的结束条件是 right == left - 1，且现在在求右边界
    // 所以用 right 替代 left - 1 更好记
    if (right < 0 || right >= nums.length) {
        return -1;
    }
    return nums[right] == target ? right : -1;
}

```

如果以上内容你都能理解，那么恭喜你，二分查找算法的细节不过如此。通过本文，你学会了：

- 1、分析二分查找代码时，不要出现 else，全部展开成 else if 方便理解。把逻辑写对之后再合并分支，提升运行效率。
- 2、注意「搜索区间」和 while 的终止条件，如果存在漏掉的元素，记得在最后检查。
- 3、如需定义左闭右开的「搜索区间」搜索左右边界，只要在 `nums[mid] == target` 时做修改即可，搜索右侧时需要减一。

4、如果将「搜索区间」全都统一成两端都闭，好记，只要稍改 `nums[mid] == target` 条件处的代码和返回的逻辑即可，推荐拿小本本记下，作为二分搜索模板。

最后我想说，以上二分搜索的框架属于「术」的范畴，如果上升到「道」的层面，二分思维的精髓就是：通过已知信息尽可能多地收缩（折半）搜索空间，从而增加穷举效率，快速找到目标。

理解本文能保证你写出正确的二分查找的代码，但实际题目中不会直接让你写二分代码，我会在 [二分查找的运用](#) 和 [二分查找的更多习题](#) 中进一步讲解如何把二分思维运用到更多算法题中。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1201. Ugly Number III</a>	<a href="#">1201. 丑数 III</a>	
<a href="#">1235. Maximum Profit in Job Scheduling</a>	<a href="#">1235. 规划兼职工作</a>	
<a href="#">162. Find Peak Element</a>	<a href="#">162. 寻找峰值</a>	
<a href="#">240. Search a 2D Matrix II</a>	<a href="#">240. 搜索二维矩阵 II</a>	
<a href="#">33. Search in Rotated Sorted Array</a>	<a href="#">33. 搜索旋转排序数组</a>	
<a href="#">35. Search Insert Position</a>	<a href="#">35. 搜索插入位置</a>	
<a href="#">658. Find K Closest Elements</a>	<a href="#">658. 找到 K 个最接近的元素</a>	
<a href="#">74. Search a 2D Matrix</a>	<a href="#">74. 搜索二维矩阵</a>	
<a href="#">792. Number of Matching Subsequences</a>	<a href="#">792. 匹配子序列的单词数</a>	
<a href="#">793. Preimage Size of Factorial Zeroes Function</a>	<a href="#">793. 阶乘函数后 K 个零</a>	
<a href="#">81. Search in Rotated Sorted Array II</a>	<a href="#">81. 搜索旋转排序数组 II</a>	
<a href="#">852. Peak Index in a Mountain Array</a>	<a href="#">852. 山脉数组的峰顶索引</a>	
-	<a href="#">剑指 Offer 04. 二维数组中的查找</a>	
-	<a href="#">剑指 Offer 53 - I. 在排序数组中查找数字 I</a>	
-	<a href="#">剑指 Offer 53 - II. 0~n-1中缺失的数字</a>	
-	<a href="#">剑指 Offer II 068. 查找插入位置</a>	
-	<a href="#">剑指 Offer II 069. 山峰数组的顶部</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 动态规划解题套路框架



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">509. Fibonacci Number</a>	<a href="#">509. 斐波那契数</a>	
<a href="#">322. Coin Change</a>	<a href="#">322. 零钱兑换</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的遍历框架](#)
- [多叉树结构及遍历框架](#)

tip：本文有视频版：[动态规划框架套路详解](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

这篇文章是我多年前一篇 200 多赞赏的 [动态规划详解](#) 的进阶版，我添加了更多干货内容，希望本文成为解决动态规划的一部「指导方针」。

动态规划问题（Dynamic Programming）应该是很多读者头疼的，不过这类问题也是最具有技巧性，最有意思的。本书使用了整整一个章节专门来写这个算法，动态规划的重要性也可见一斑。

本文解决几个问题：

动态规划是什么？解决动态规划问题有什么技巧？如何学习动态规划？

刷题刷多了就会发现，算法技巧就那几个套路，我们后续的动态规划系列章节，都在使用本文的解题框架思维，如果你心里有数，就会轻松很多。所以本文放在第一章，希望能够成为解决动态规划问题的一部指导方针，下面上干货。

首先，**动态规划问题的一般形式就是求最值**。动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求最长递增子序列呀，最小编辑距离呀等等。

既然是要求最值，核心问题是什么呢？**求解动态规划的核心问题是穷举**。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。

动态规划这么简单，就是穷举就完事了？我看到的动态规划问题都很难啊！

首先，虽然动态规划的核心思想就是穷举求最值，但是问题可以千变万化，穷举所有可行解其实并不是一件容易的事，需要你熟练掌握递归思维，只有列出**正确的「状态转移方程」**，才能正确地穷举。而且，你需要判断算法问题是否具备**「最优子结构」**，是否能够通过子问题的最值得到原问题的最值。另外，动态规划问题存在**「重叠子问题」**，如果暴力穷举的话效率会很低，所以需要你使用**「备忘录」**或者**「DP table」**来优化穷举过程，避免不必要的计算。

以上提到的重叠子问题、最优子结构、状态转移方程就是动态规划三要素。具体什么意思等会会举例详解，但是在实际的算法问题中，写出状态转移方程是最困难的，这也就是为什么很多朋友觉得动态规划问题困难的原因，我来提供我总结的一个思维框架，辅助你思考状态转移方程：

明确「状态」 -> 明确「选择」 -> 定义 dp 数组/函数的含义。

按上面的套路走，最后的解法代码就会是如下的框架：

```
# 自顶向下递归的动态规划
def dp(状态1, 状态2, ...):
    for 选择 in 所有可能的选择:
        # 此时的状态已经因为做了选择而改变
        result = 求最值(result, dp(状态1, 状态2, ...))
    return result

# 自底向上迭代的动态规划
# 初始化 base case
dp[0][0][...] = base case
# 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...)
```

下面通过斐波那契数列问题和凑零钱问题来详解动态规划的基本原理。前者主要是让你明白什么是重叠子问题（斐波那契数列没有求最值，所以严格来说不是动态规划问题），后者主要举集中于如何列出状态转移方程。

## 一、斐波那契数列

力扣第 509 题「斐波那契数」就是这个问题，请读者不要嫌弃这个例子简单，只有简单的例子才能让你把精力充分集中在算法背后的通用思想和技巧上，而不会被那些隐晦的细节问题搞的莫名其妙。想要困难的例子，接下来的动态规划系列里有的是。

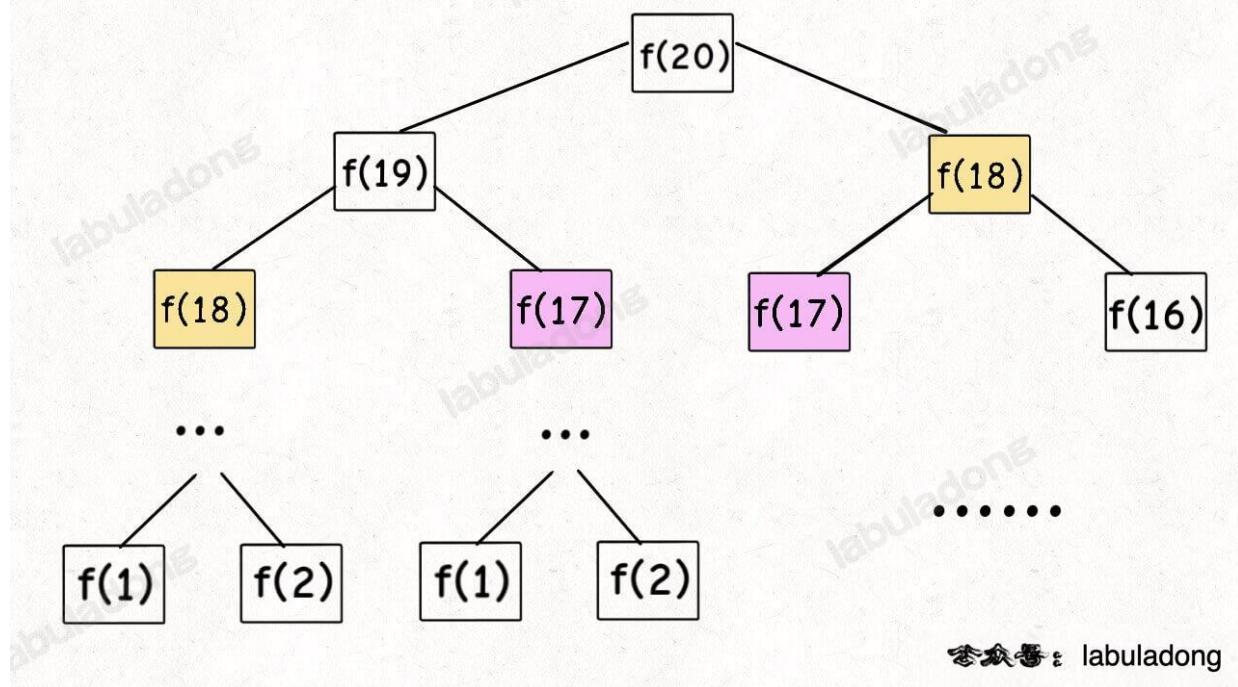
### 暴力递归

斐波那契数列的数学形式就是递归的，写成代码就是这样：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

### ▶ ⭐ 代码可视化动画⭐

这个不用多说了，学校老师讲递归的时候似乎都是拿这个举例。我们也知道这样写代码虽然简洁易懂，但是十分低效，低效在哪里？假设  $n = 20$ ，请画出递归树：



但凡遇到需要递归的问题，最好都画出递归树，这对你分析算法的复杂度，寻找算法低效的原因都有巨大帮助。

这个递归树怎么理解？就是说想要计算原问题  $f(20)$ ，我就得先计算出子问题  $f(19)$  和  $f(18)$ ，然后要计算  $f(19)$ ，我就得先算出子问题  $f(18)$  和  $f(17)$ ，以此类推。最后遇到  $f(1)$  或者  $f(2)$  的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

首先计算子问题个数，即递归树中节点的总数。显然二叉树节点总数为指数级别，所以子问题个数为  $O(2^n)$ 。

然后计算解决一个子问题的时间，在本算法中，没有循环，只有  $f(n - 1) + f(n - 2)$  一个加法操作，时间为  $O(1)$ 。

所以，这个算法的时间复杂度为二者相乘，即  $O(2^n)$ ，指数级别，爆炸。

观察递归树，很明显发现了算法低效的原因：存在大量重复计算，比如  $f(18)$  被计算了两次，而且你可以看到，以  $f(18)$  为根的这个递归树体量巨大，多算一遍，会耗费巨大的时间。更何况，还不止  $f(18)$  这一个节点被重复计算，所以这个算法及其低效。

这就是动态规划问题的第一个性质：重叠子问题。下面，我们想办法解决这个问题。

## 带备忘录的递归解法

明确了问题，其实就已经把问题解决了一半。既然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。

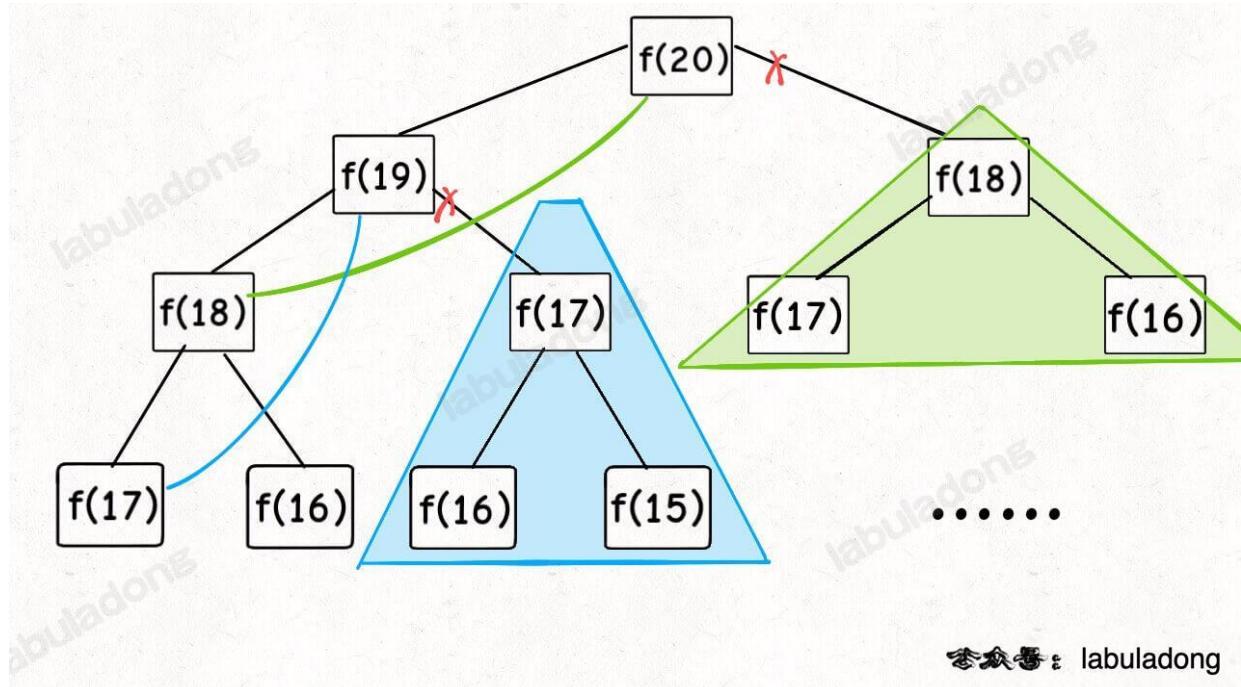
一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

```
int fib(int N) {
    // 备忘录全初始化为 0
    int[] memo = new int[N + 1];
    // 进行带备忘录的递归
    return dp(memo, N);
}
```

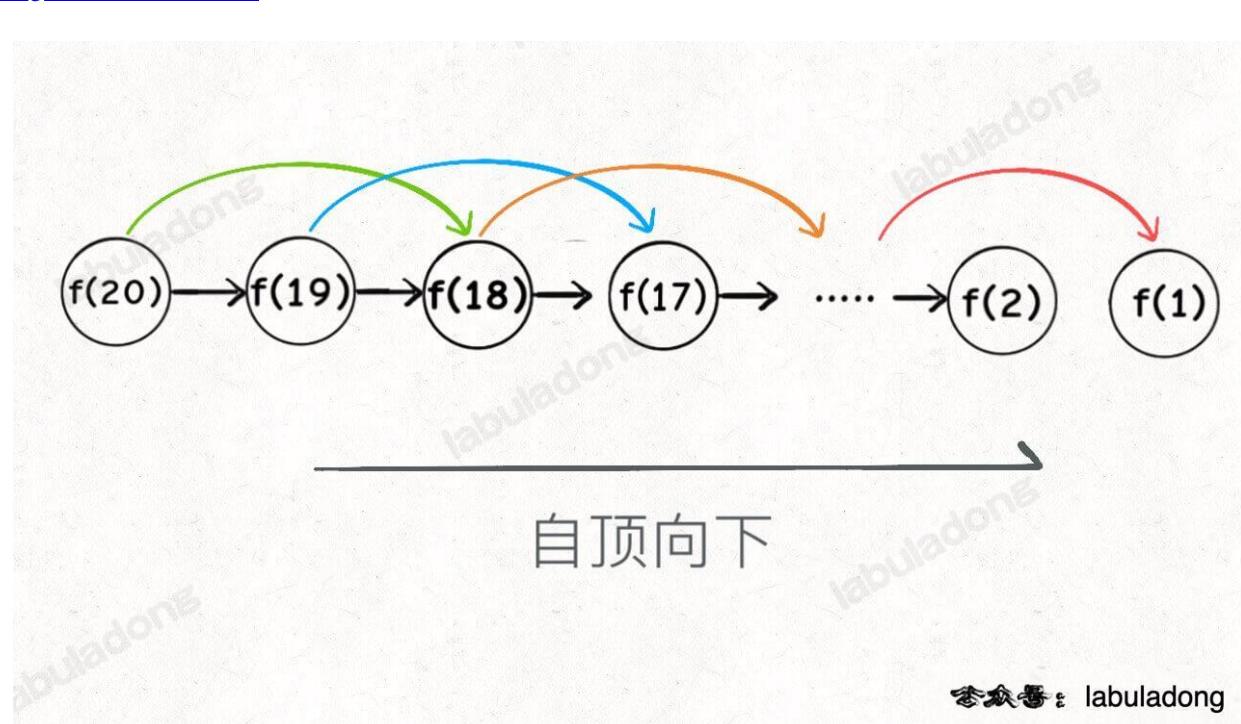
```
// 带着备忘录进行递归
int dp(int[] memo, int n) {
    // base case
    if (n == 0 || n == 1) return n;
    // 已经计算过，不用再计算了
    if (memo[n] != 0) return memo[n];
    memo[n] = dp(memo, n - 1) + dp(memo, n - 2);
    return memo[n];
}
```

▶ 🎃 代码可视化动画🎃

现在，画出递归树，你就知道「备忘录」到底做了什么。



实际上，带「备忘录」的递归算法，把一棵存在巨量冗余的递归树通过「剪枝」，改造成了一幅不存在冗余的递归图，极大减少了子问题（即递归图中节点）的个数。



递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

子问题个数，即图中节点的总数，由于本算法不存在冗余计算，子问题就是  $f(1), f(2), f(3) \dots f(20)$ ，数量和输入规模  $n = 20$  成正比，所以子问题个数为  $O(n)$ 。

解决一个子问题的时间，同上，没有什么循环，时间为  $O(1)$ 。

所以，本算法的时间复杂度是  $O(n)$ ，比起暴力算法，是降维打击。

至此，带备忘录的递归解法的效率已经和迭代的动态规划解法一样了。实际上，这种解法和常见的动态规划解法已经差不多了，只不过这种解法是「自顶向下」进行「递归」求解，我们更常见的动态规划代码是「自底向上」进行「递推」求解。

啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说  $f(20)$ ，向下逐渐分解规模，直到  $f(1)$  和  $f(2)$  这两个 base case，然后逐层返回答案，这就叫「自顶向下」。

啥叫「自底向上」？反过来，我们直接从最底下、最简单、问题规模最小、已知结果的  $f(1)$  和  $f(2)$  (base case) 开始往上推，直到推到我们想要的答案  $f(20)$ 。这就是「递推」的思路，这也是动态规划一般都脱离了递归，而是由循环迭代完成计算的原因。

## dp 数组的迭代（递推）解法

有了上一步「备忘录」的启发，我们可以把这个「备忘录」独立出来成为一张表，通常叫做 DP table，在这张表上完成「自底向上」的推算岂不美哉！

```
int fib(int N) {
    if (N == 0) return 0;
    int[] dp = new int[N + 1];
    // base case
    dp[0] = 0; dp[1] = 1;
    // 状态转移
    for (int i = 2; i <= N; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}
```

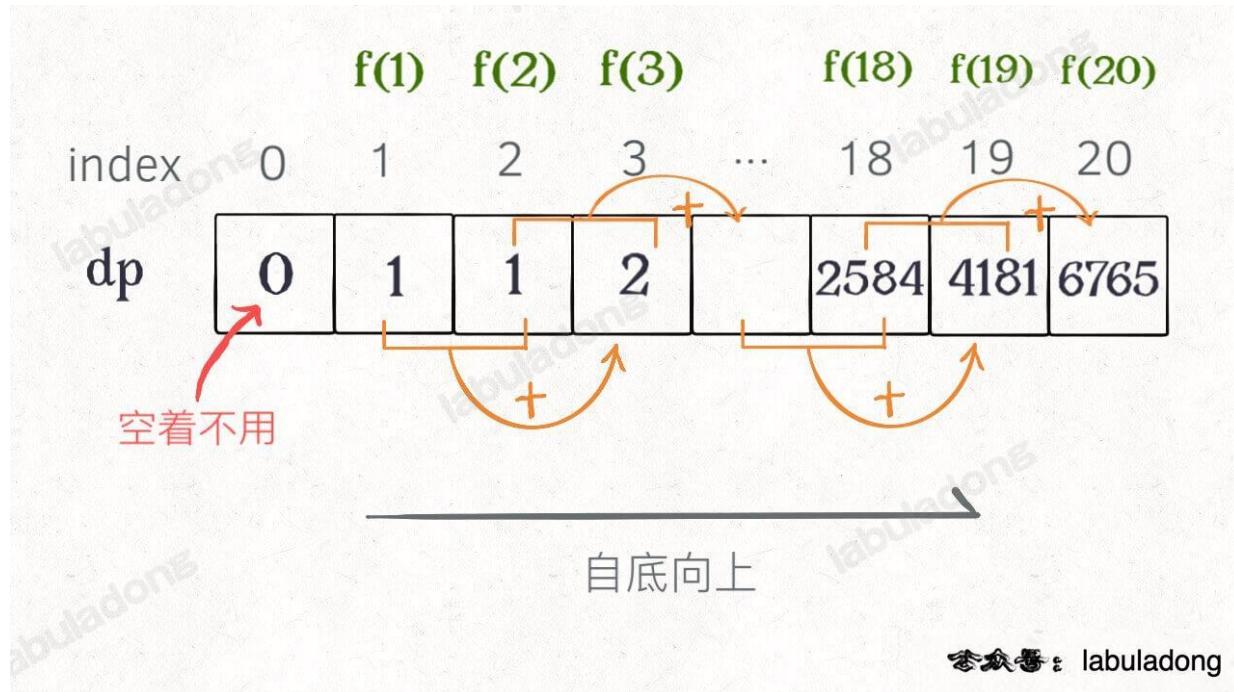
```

        return dp[N];
    }
}

```

### ▶ 🎃 代码可视化动画🎃

画个图就很好理解了，而且你发现这个 DP table 特别像之前那个「剪枝」后的结果，只是反过来算而已：



实际上，带备忘录的递归解法中的那个「备忘录」`memo` 数组，最终完成后就是这个解法中的 `dp` 数组，你对比一下可视化面板中两个算法执行的过程可以更直观地看出它俩的联系。

所以说自顶向下、自底向上两种解法本质其实是差不多的，大部分情况下，效率也基本相同。

## 拓展延伸

这里，引出「状态转移方程」这个名词，实际上就是描述问题结构的数学形式：

$$f(n) = \begin{cases} 1, & n = 1, 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

为啥叫「状态转移方程」？其实就是为了听起来高端。

$f(n)$  的函数参数会不断变化，所以你把参数  $n$  想做一个状态，这个状态  $n$  是由状态  $n - 1$  和状态  $n - 2$  转移（相加）而来，这就叫状态转移，仅此而已。

你会发现，上面的几种解法中的所有操作，例如 `return f(n - 1) + f(n - 2)`, `dp[i] = dp[i - 1] + dp[i - 2]`，以及对备忘录或 DP table 的初始化操作，都是围绕这个方程的不同表现形式。

可见列出「状态转移方程」的重要性，它是解决问题的核心，而且很容易发现，其实状态转移方程直接代表着暴力解法。

千万不要看不起暴力解，动态规划问题最困难的就是写出这个暴力解，即状态转移方程。

只要写出暴力解，优化方法无非是用备忘录或者 DP table，再无奥妙可言。

这个例子的最后，讲一个细节优化。

细心的读者会发现，根据斐波那契数列的状态转移方程，当前状态  $n$  只和之前的  $n-1$ ,  $n-2$  两个状态有关，其实并不需要那么长的一个 DP table 来存储所有的状态，只要想办法存储之前的两个状态就行了。

所以，可以进一步优化，把空间复杂度降为  $O(1)$ 。这也就是我们最常见的计算斐波那契数的算法：

```
int fib(int n) {
    if (n == 0 || n == 1) {
        // base case
        return n;
    }
    // 分别代表 dp[i - 1] 和 dp[i - 2]
    int dp_i_1 = 1, dp_i_2 = 0;
    for (int i = 2; i <= n; i++) {
        // dp[i] = dp[i - 1] + dp[i - 2];
        int dp_i = dp_i_1 + dp_i_2;
        // 滚动更新
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i_1;
}
```

### ▶ 🎃 代码可视化动画🎃

这一般是动态规划问题的最后一步优化，如果我们发现每次状态转移只需要 DP table 中的一部分，那么可以尝试缩小 DP table 的大小，只记录必要的数据，从而降低空间复杂度。

上述例子就相当于把 DP table 的大小从  $n$  缩小到 2，即把空间复杂度下降了一个量级。我会在后文 [对动态规划发动降维打击](#) 进一步讲解这个压缩空间复杂度的技巧，一般来说用来把一个二维的 DP table 压缩成一维，即把空间复杂度从  $O(n^2)$  压缩到  $O(n)$ 。

有人会问，动态规划的另一个重要特性「最优子结构」，怎么没有涉及？下面会涉及。斐波那契数列的例子严格来说不算动态规划，因为没有涉及求最值，以上旨在说明重叠子问题的消除方法，演示得到最优解法逐步求精的过程。下面，看第二个例子，凑零钱问题。

## 二、凑零钱问题

这是力扣第 322 题「零钱兑换」：

给你  $k$  种面值的硬币，面值分别为  $c_1, c_2 \dots c_k$ ，每种硬币的数量无限，再给一个总金额  $amount$ ，问你最少需要几枚硬币凑出这个金额，如果不可能凑出，算法返回 -1。算法的函数签名如下：

```
// coins 中是可选硬币面值，amount 是目标金额
int coinChange(int[] coins, int amount);
```

比如说  $k = 3$ ，面值分别为 1, 2, 5，总金额  $amount = 11$ 。那么最少需要 3 枚硬币凑出，即  $11 = 5 + 5 + 1$ 。

你认为计算机应该如何解决这个问题？显然，就是把所有可能的凑硬币方法都穷举出来，然后找找看最少需要多少枚硬币。

## 暴力递归

首先，这个问题是动态规划问题，因为它具有「最优子结构」的。要符合「最优子结构」，子问题间必须互相独立。啥叫相互独立？你肯定不想看数学证明，我用一个直观的例子来讲解。

比如说，假设你考试，每门科目的成绩都是互相独立的。你的原问题是考出最高的总成绩，那么你的子问题就是要把语文考到最高，数学考到最高……为了每门课考到最高，你要把每门课相应的选择题分数拿到最高，填空题分数拿到最高……当然，最终就是你每门课都是满分，这就是最高的总成绩。

得到了正确的结果：最高的总成绩就是总分。因为这个过程符合最优子结构，「每门科目考到最高」这些子问题是互相独立，互不干扰的。

但是，如果加一个条件：你的语文成绩和数学成绩会互相制约，不能同时达到满分，数学分数高，语文分数就会降低，反之亦然。

这样的话，显然你能考到的最高总成绩就达不到总分了，按刚才那个思路就会得到错误的结果。因为「每门科目考到最高」的子问题并不独立，语文数学成绩互相影响，无法同时最优，所以最优子结构被破坏。

回到凑零钱问题，为什么说它符合最优子结构呢？假设你有面值为 1, 2, 5 的硬币，你想求 `amount = 11` 时的最少硬币数（原问题），如果你知道凑出 `amount = 10, 9, 6` 的最少硬币数（子问题），你只需要把子问题的答案加一（再选一枚面值为 1, 2, 5 的硬币），求个最小值，就是原问题的答案。因为硬币的数量是没限制的，所以子问题之间没有相互制约，是互相独立的。

关于最优子结构的问题，后文 [动态规划答疑篇](#) 还会再举例探讨。

那么，既然知道了这是个动态规划问题，就要思考如何列出正确的状态转移方程？

1、确定「状态」，也就是原问题和子问题中会变化的变量。由于硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 `amount`。

2、确定「选择」，也就是导致「状态」产生变化的行为。目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额。所以说所有硬币的面值，就是你的「选择」。

3、明确 `dp` 函数/数组的定义。我们这里讲的是自顶向下的解法，所以会有一个递归的 `dp` 函数，一般来说函数的参数就是状态转移中会变化的量，也就是上面说到的「状态」；函数的返回值就是题目要求我们计算的量。就本题来说，状态只有一个，即「目标金额」，题目要求我们计算凑出目标金额所需的最少硬币数量。

所以我们可以这样定义 `dp` 函数：`dp(n)` 表示，输入一个目标金额 `n`，返回凑出目标金额 `n` 所需的最少硬币数量。

那么根据这个定义，我们的最终答案就是 `dp(amount)` 的返回值。

搞清楚上面这几个关键点，解法的伪码就可以写出来了：

```
// 伪码框架
int coinChange(int[] coins, int amount) {
    // 题目要求的最终结果是 dp(amount)
    return dp(coins, amount);
}

// 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
int dp(int[] coins, int n) {
    // 做选择，选择需要硬币最少的那个结果
    for (int coin : coins) {
        res = min(res, 1 + dp(coins, n - coin));
    }
}
```

```
    return res;
}
```

根据伪码，我们加上 base case 即可得到最终的答案。显然目标金额为 0 时，所需硬币数量为 0；当目标金额小于 0 时，无解，返回 -1：

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        // 题目要求的最终结果是 dp(amount)
        return dp(coins, amount);
    }

    // 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
    int dp(int[] coins, int amount) {
        // base case
        if (amount == 0) return 0;
        if (amount < 0) return -1;

        int res = Integer.MAX_VALUE;
        for (int coin : coins) {
            // 计算子问题的结果
            int subProblem = dp(coins, amount - coin);
            // 子问题无解则跳过
            if (subProblem == -1) continue;
            // 在子问题中选择最优解，然后加一
            res = Math.min(res, subProblem + 1);
        }

        return res == Integer.MAX_VALUE ? -1 : res;
    }
}
```

这里 `coinChange` 和 `dp` 函数的签名完全一样，所以理论上不需要额外写一个 `dp` 函数。但为了后文讲解方便，这里还是另写一个 `dp` 函数来实现主要逻辑。

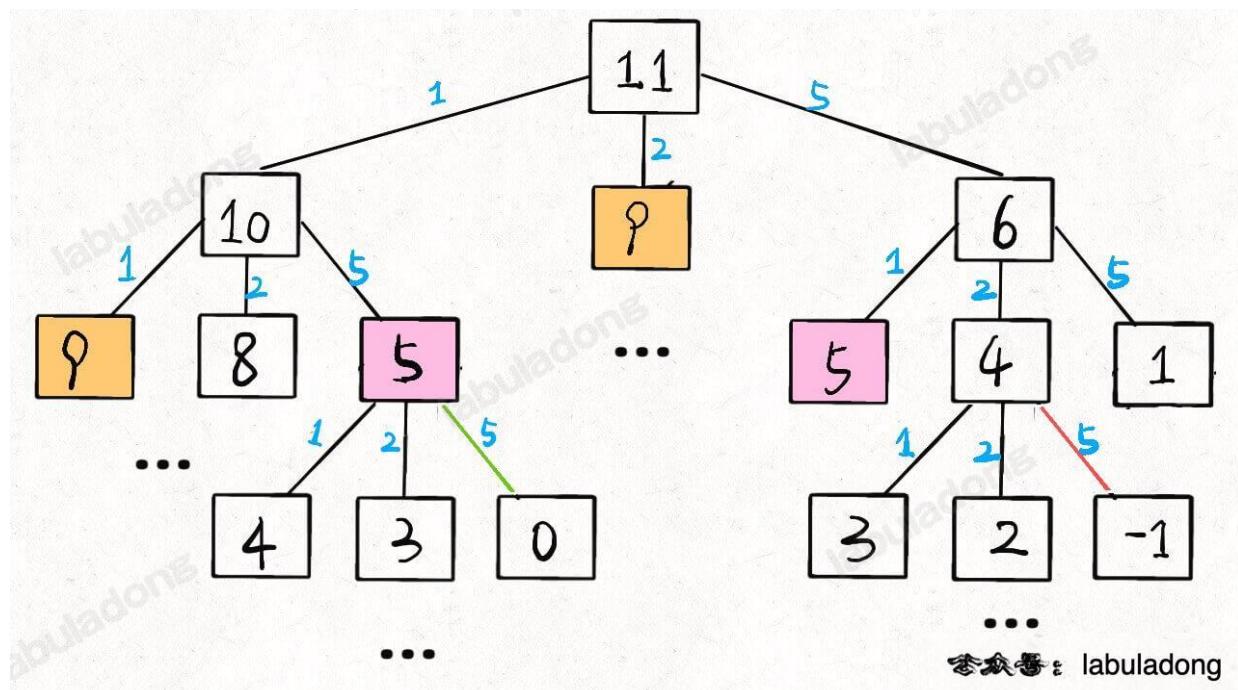
另外，我经常看到有读者留言问，子问题的结果为什么要加 1 (`subProblem + 1`)，而不是加硬币金额之类的。我这里统一提示一下，动态规划问题的关键是 `dp` 函数/数组的定义，你这个函数的返回值代表什么？你回过头去搞清楚这一点，然后就知道为什么要给子问题的返回值加 1 了。

### ▶ 🎨 代码可视化动画🎨

至此，状态转移方程其实已经完成了，以上算法已经是暴力解法了，以上代码的数学形式就是状态转移方程：

$$dp(n) = \begin{cases} 0, & n = 0 \\ -1, & n < 0 \\ \min\{dp(n - coin) + 1 \mid coin \in coins\}, & n > 0 \end{cases}$$

至此，这个问题其实就解决了，只不过需要消除一下重叠子问题，比如  $\text{amount} = 11$ ,  $\text{coins} = \{1, 2, 5\}$  时画出递归树看看：



递归算法的时间复杂度分析：子问题总数  $\times$  解决每个子问题所需的时间。

子问题总数为递归树的节点个数，但算法会进行剪枝，剪枝的时机和题目给定的具体硬币面额有关，所以可以想象，这棵树生长的并不规则，确切算出树上有多少节点是比较困难的。对于这种情况，我们一般的做法是按照最坏的情况估算一个时间复杂度的上界。

假设目标金额为  $n$ ，给定的硬币个数为  $k$ ，那么递归树最坏情况下高度为  $n$ （全用面额为 1 的硬币），然后再假设这是一棵满  $k$  叉树，则节点的总数在  $k^n$  这个数量级。

接下来看每个子问题的复杂度，由于每次递归包含一个 for 循环，复杂度为  $O(k)$ ，相乘得到总时间复杂度为  $O(k^n)$ ，指数级别。

## 带备忘录的递归

类似之前斐波那契数列的例子，只需要稍加修改，就可以通过备忘录消除子问题：

```
class Solution {
    int[] memo;

    public int coinChange(int[] coins, int amount) {
        memo = new int[amount + 1];
        // 备忘录初始化为一个不会被取到的特殊值，代表还未被计算
        Arrays.fill(memo, -666);

        return dp(coins, amount);
    }

    int dp(int[] coins, int amount) {
        if (amount == 0) return 0;
        if (amount < 0) return -1;
        // 查备忘录，防止重复计算
        if (memo[amount] != -666)
            return memo[amount];
        else
            memo[amount] = ... // 计算结果
    }
}
```

```

int res = Integer.MAX_VALUE;
for (int coin : coins) {
    // 计算子问题的结果
    int subProblem = dp(coins, amount - coin);
    // 子问题无解则跳过
    if (subProblem == -1) continue;
    // 在子问题中选择最优解，然后加一
    res = Math.min(res, subProblem + 1);
}
// 把计算结果存入备忘录
memo[amount] = (res == Integer.MAX_VALUE) ? -1 : res;
return memo[amount];
}
}

```

## ▶ 彩虹 代码可视化动画

不画图了，很显然「备忘录」大大减小了子问题数目，完全消除了子问题的冗余，所以子问题总数不会超过金额数  $n$ ，即子问题数目为  $O(n)$ 。处理一个子问题的时间不变，仍是  $O(k)$ ，所以总的时间复杂度是  $O(kn)$ 。

## dp 数组的迭代解法

当然，我们也可以自底向上使用 dp table 来消除重叠子问题，关于「状态」「选择」和 base case 与之前没有区别，dp 数组的定义和刚才 dp 函数类似，也是把「状态」，也就是目标金额作为变量。不过 dp 函数体现在函数参数，而 dp 数组体现在数组索引：

**dp 数组的定义：当目标金额为  $i$  时，至少需要  $dp[i]$  枚硬币凑出。**

根据我们文章开头给出的动态规划代码框架可以写出如下解法：

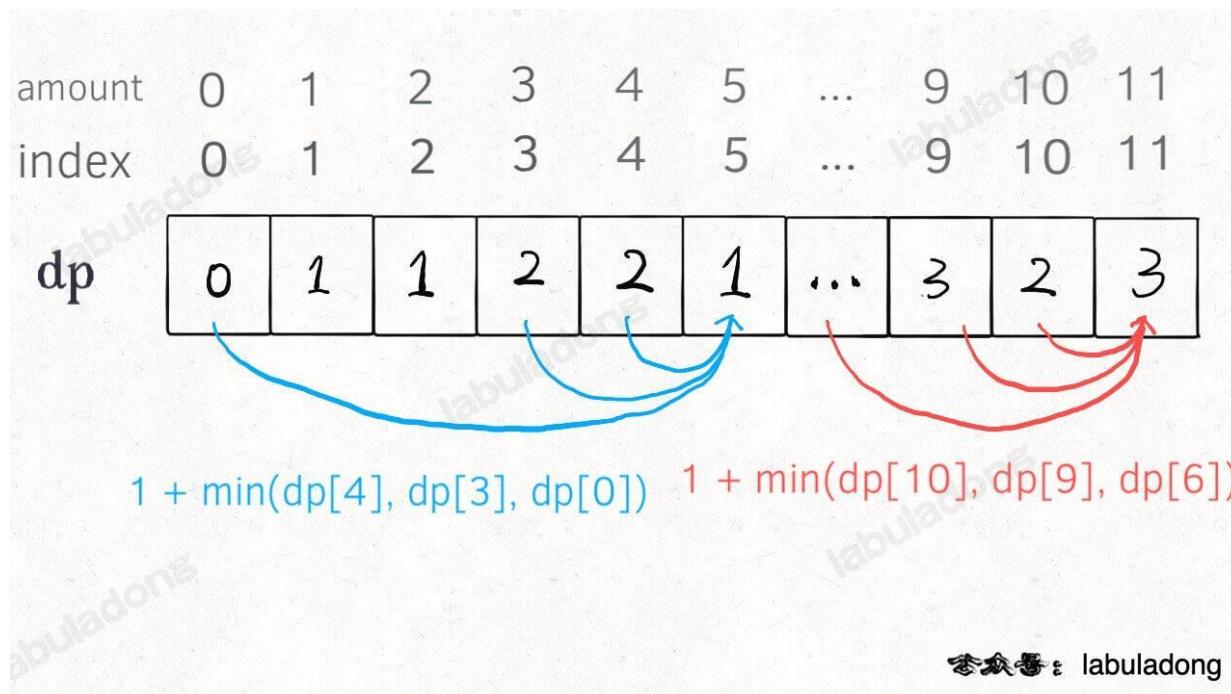
```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        // 数组大小为 amount + 1，初始值也为 amount + 1
        Arrays.fill(dp, amount + 1);

        // base case
        dp[0] = 0;
        // 外层 for 循环在遍历所有状态的所有取值
        for (int i = 0; i < dp.length; i++) {
            // 内层 for 循环在求所有选择的最小值
            for (int coin : coins) {
                // 子问题无解，跳过
                if (i - coin < 0) {
                    continue;
                }
                dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
            }
        }
        return (dp[amount] == amount + 1) ? -1 : dp[amount];
    }
}

```

为啥 `dp` 数组中的值都初始化为 `amount + 1` 呢，因为凑成 `amount` 金额的硬币数最多只可能等于 `amount`（全用 1 元面值的硬币），所以初始化为 `amount + 1` 就相当于初始化为正无穷，便于后续取最小值。为啥不直接初始化为 `int` 型的最大值 `Integer.MAX_VALUE` 呢？因为后面有 `dp[i - coin] + 1`，这就会导致整型溢出。



### 三、最后总结

第一个斐波那契数列的问题，解释了如何通过「备忘录」或者「dp table」的方法来优化递归树，并且明确了这两种方法本质上是一样的，只是自顶向下和自底向上的不同而已。

第二个凑零钱的问题，展示了如何流程化确定「状态转移方程」，只要通过状态转移方程写出暴力递归解，剩下的也就是优化递归树，消除重叠子问题而已。

如果你不太了解动态规划，还能看到这里，真得给你鼓掌，相信你已经掌握了这个算法的设计技巧。

计算机解决问题其实没有任何特殊的技巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考“如何穷举”，然后再追求“如何聪明地穷举”。

列出状态转移方程，就是在解决“如何穷举”的问题。之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不容易穷举完整。

备忘录、DP table 就是在追求“如何聪明地穷举”。用空间换时间的思路，是降低时间复杂度的不二法门，除此之外，试问，还能玩出啥花活？

之后我们会有一章专门讲解动态规划问题，如果有任何问题都可以随时回来重读本文，希望读者在阅读每个题目和解法时，多往「状态」和「选择」上靠，才能对这套框架产生自己的理解，运用自如。

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	111. 二叉树的最小深度	●
<a href="#">112. Path Sum</a>	112. 路径总和	●

LeetCode	力扣	难度
115. Distinct Subsequences	115. 不同的子序列	●
139. Word Break	139. 单词拆分	●
1696. Jump Game VI	1696. 跳跃游戏 VI	●
221. Maximal Square	221. 最大正方形	●
240. Search a 2D Matrix II	240. 搜索二维矩阵 II	●
256. Paint House	256. 粉刷房子	●
279. Perfect Squares	279. 完全平方数	●
343. Integer Break	343. 整数拆分	●
365. Water and Jug Problem	365. 水壶问题	●
542. 01 Matrix	542. 01 矩阵	●
576. Out of Boundary Paths	576. 出界的路径数	●
62. Unique Paths	62. 不同路径	●
63. Unique Paths II	63. 不同路径 II	●
70. Climbing Stairs	70. 爬楼梯	●
91. Decode Ways	91. 解码方法	●
-	剑指 Offer 04. 二维数组中的查找	●
-	剑指 Offer 10- I. 斐波那契数列	●
-	剑指 Offer 10- II. 青蛙跳台阶问题	●
-	剑指 Offer 14- I. 剪绳子	●
-	剑指 Offer 46. 把数字翻译成字符串	●
-	剑指 Offer II 091. 粉刷房子	●
-	剑指 Offer II 097. 子序列的数目	●
-	剑指 Offer II 098. 路径的数目	●
-	剑指 Offer II 103. 最少的硬币数目	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

# 回溯算法解题套路框架



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">46. Permutations</a>	<a href="#">46. 全排列</a>	★★★

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的遍历框架
- 多叉树结构及遍历框架

tip：本文有视频版：[回溯算法框架套路详解](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

这篇文章是很久之前的一篇[回溯算法详解](#)的进阶版。把框架给你讲清楚，你会发现回溯算法问题都是一个套路。

本文解决几个问题：

回溯算法是什么？解决回溯算法相关的问题有什么技巧？如何学习回溯算法？回溯算法代码是否有规律可循？

其实回溯算法和我们常说的 DFS 算法基本可以认为是同一种算法，它们的细微差异我在[关于 DFS 和回溯算法的若干问题](#)中有详细解释，本文聚焦回溯算法，不展开。

抽象地说，解决一个回溯问题，实际上就是遍历一棵决策树的过程，树的每个叶子节点存放着一个合法答案。你把整棵树遍历一遍，把叶子节点上的答案都收集起来，就能得到所有的合法答案。

站在回溯树的一个节点上，你只需要思考 3 个问题：

- 1、路径：也就是已经做出的选择。
- 2、选择列表：也就是你当前可以做的选择。
- 3、结束条件：也就是到达决策树底层，无法再做选择的条件。

如果你不理解这三个词语的解释，没关系，我们后面会用「全排列」这个经典的回溯算法问题来帮你理解这些词语是什么意思，现在你先留着印象。

代码方面，回溯算法的框架：

```
result = []
def backtrack(路径, 选择列表):
```

```

if 满足结束条件:
    result.add(路径)
    return

for 选择 in 选择列表:
    做选择
    backtrack(路径, 选择列表)
    撤销选择

```

其核心就是 **for 循环里面的递归**，在递归调用之前「做选择」，在递归调用之后「撤销选择」，特别简单。

什么叫做选择和撤销选择呢，这个框架的底层原理是什么呢？下面我们就通过「全排列」这个问题来解开之前的疑惑，详细探究一下其中的奥妙！

## 一、全排列问题

力扣第 46 题「全排列」就是给你输入一个数组 `nums`，让你返回这些数字的全排列。

我们这次讨论的全排列问题不包含重复的数字，包含重复数字的扩展场景我在后文 [回溯算法秒杀排列组合子集的九种题型](#) 中讲解。

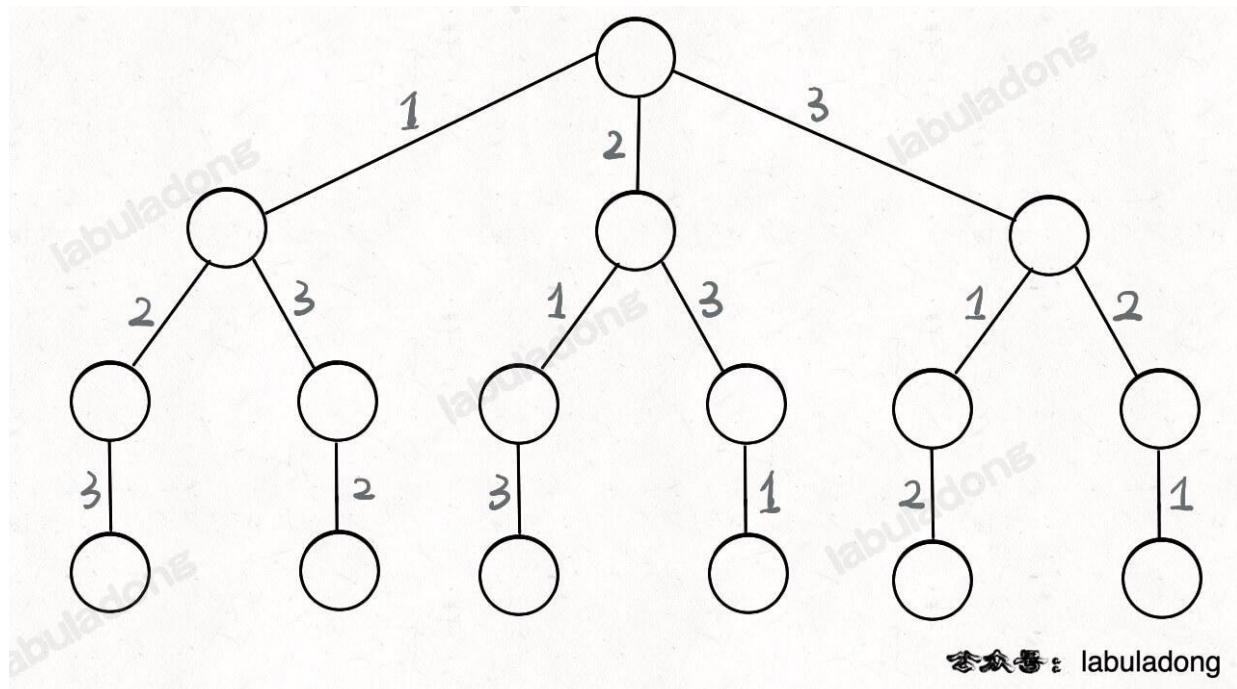
另外，有些读者之前看过的全排列算法代码可能是那种 `swap` 交换元素的写法，和我在本文介绍的代码不同。这是回溯算法两种穷举思路，我会在后文 [球盒模型：回溯算法穷举的两种视角](#) 讲明白。现在还不适合直接跟你讲那个解法，你照着我的思路学习即可。

我们在高中的时候就做过排列组合的数学题，我们也知道  $n$  个不重复的数，全排列共有  $n!$  个。那么我们当时是怎么穷举全排列的呢？

比方说给三个数 `[1, 2, 3]`，你肯定不会无规律地乱穷举，一般是这样：

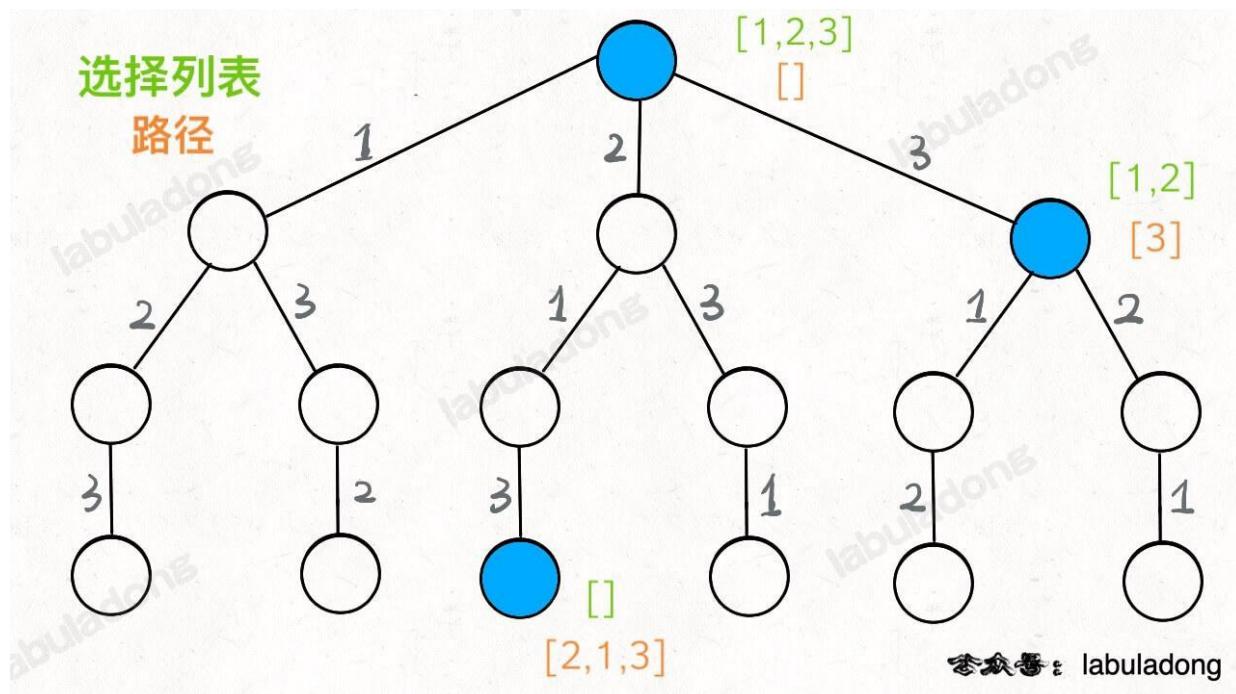
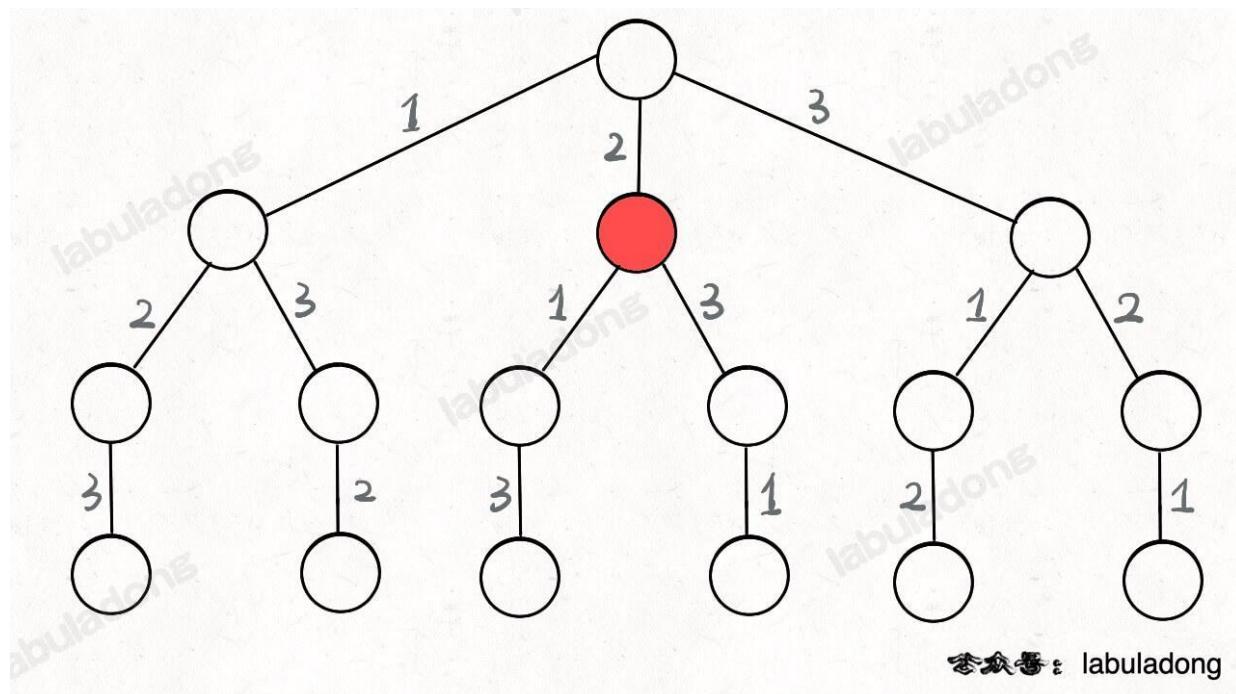
先固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；然后就只能变化第一位，变成 2，然后再穷举后两位……

其实这就是回溯算法，我们高中无师自通就会用，或者有的同学直接画出如下这棵回溯树：



只要从根遍历这棵树，记录路径上的数字，其实就是所有的全排列。我们不妨把这棵树称为回溯算法的「决策树」。

为啥说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



我们定义的 **backtrack** 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层叶子节点，其「路径」就是一个全排列。

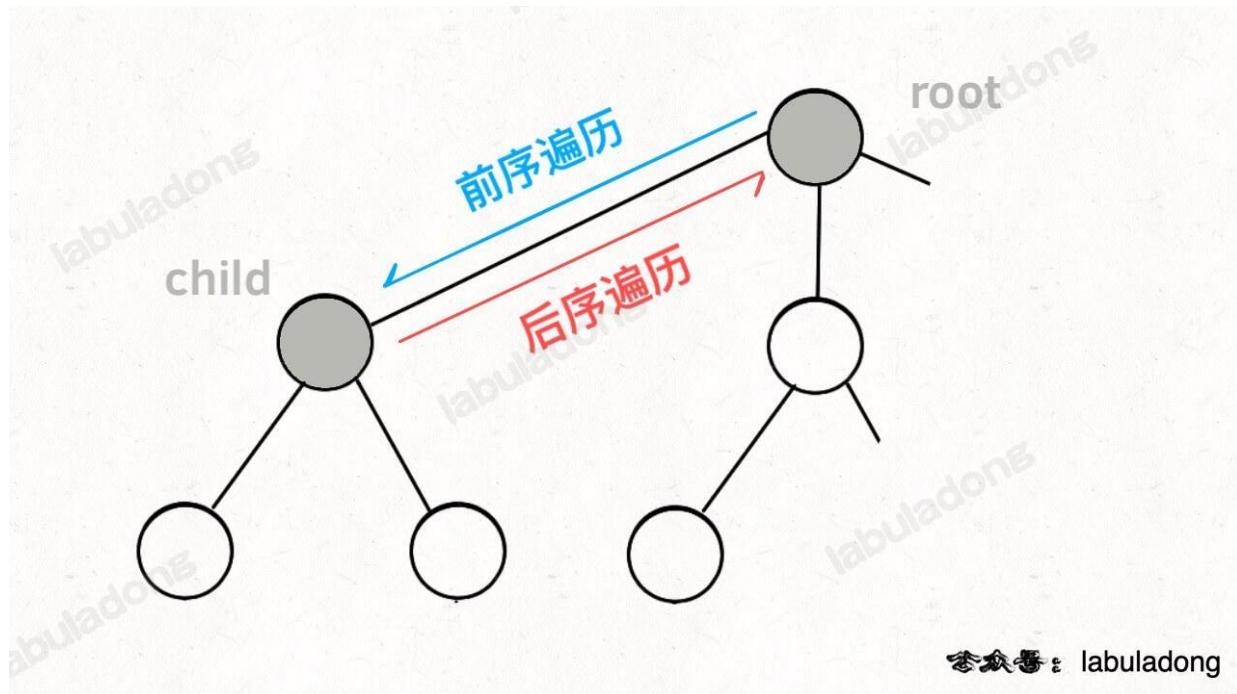
再进一步，如何遍历一棵树？这个应该不难吧。回忆一下之前 [学习数据结构的框架思维](#) 写过，各种搜索问题其实都是树的遍历问题，而多叉树的遍历框架就是这样：

```
void traverse(TreeNode root) {  
    for (TreeNode child : root.children) {  
        // 前序位置需要的操作  
        traverse(child);  
        // 后序位置需要的操作  
    }  
}
```

细心的读者肯定会疑问：多叉树 DFS 遍历框架的前序位置和后序位置应该在 for 循环外面，并不应该是在 for 循环里面呀？为什么在回溯算法中跑到 for 循环里面了？

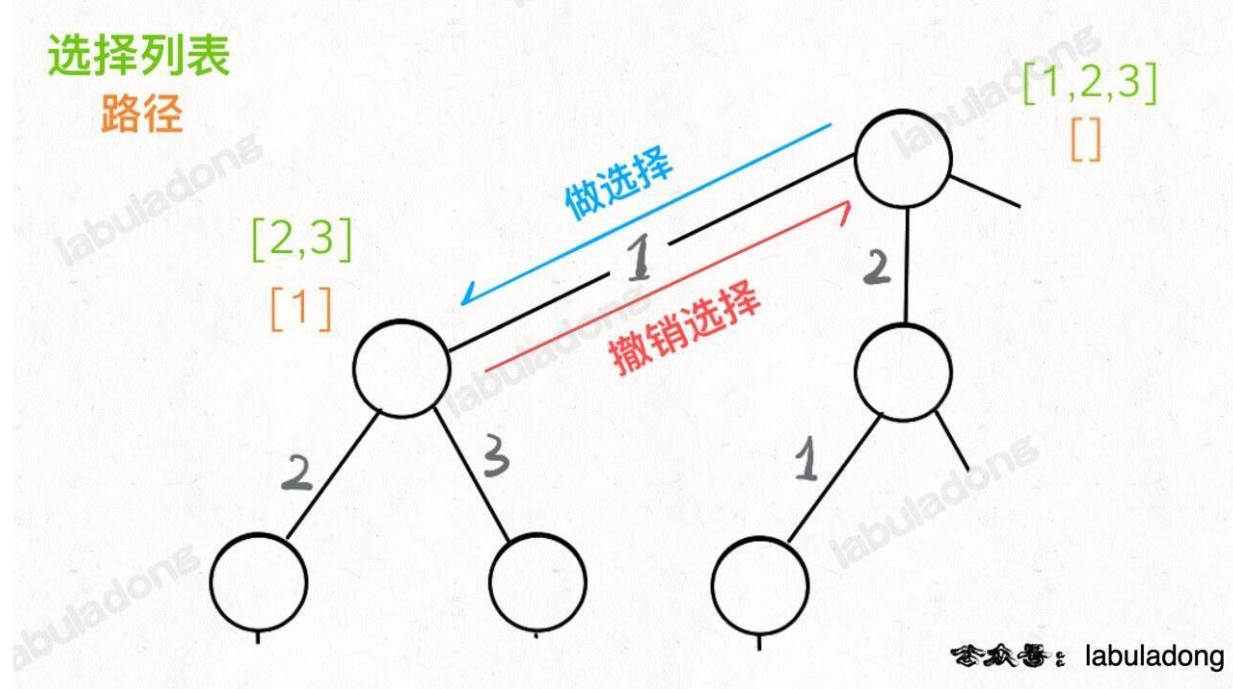
是的，DFS 算法的前序和后序位置应该在 for 循环外面，不过回溯算法和 DFS 算法略有不同，[解答回溯/DFS 算法的若干疑问](#) 会具体讲解，这里可以暂且忽略这个问题。

而所谓的前序遍历和后序遍历，他们只是两个很有用的时间点，我给你画张图你就明白了：



前序遍历的代码在进入某一个节点之前的那个时间点执行，后序遍历代码在离开某个节点之后的那个时间点执行。

回想我们刚才说的，「路径」和「选择」是每个节点的属性，函数在树上游走要正确处理节点的属性，那么就要在这两个特殊时间点搞点动作：



现在，你是否理解了回溯算法的这段核心框架？

```
for 选择 in 选择列表:
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

我们只要在递归之前做出选择，在递归之后撤销刚才的选择，就能正确得到每个节点的选择列表和路径。

下面，直接看全排列代码：

```
class Solution {
    List<List<Integer>> res = new LinkedList<>();

    // 主函数，输入一组不重复的数字，返回它们的全排列
    List<List<Integer>> permute(int[] nums) {
        // 记录「路径」
        LinkedList<Integer> track = new LinkedList<>();
        // 「路径」中的元素会被标记为 true，避免重复使用
        boolean[] used = new boolean[nums.length];

        backtrack(nums, track, used);
        return res;
    }

    // 路径：记录在 track 中
    // 选择列表：nums 中不存在于 track 的那些元素 (used[i] 为 false)
    // 结束条件：nums 中的元素全都在 track 中出现
    void backtrack(int[] nums, LinkedList<Integer> track, boolean[] used) {
        // 触发结束条件
        if (track.size() == nums.length) {
```

```

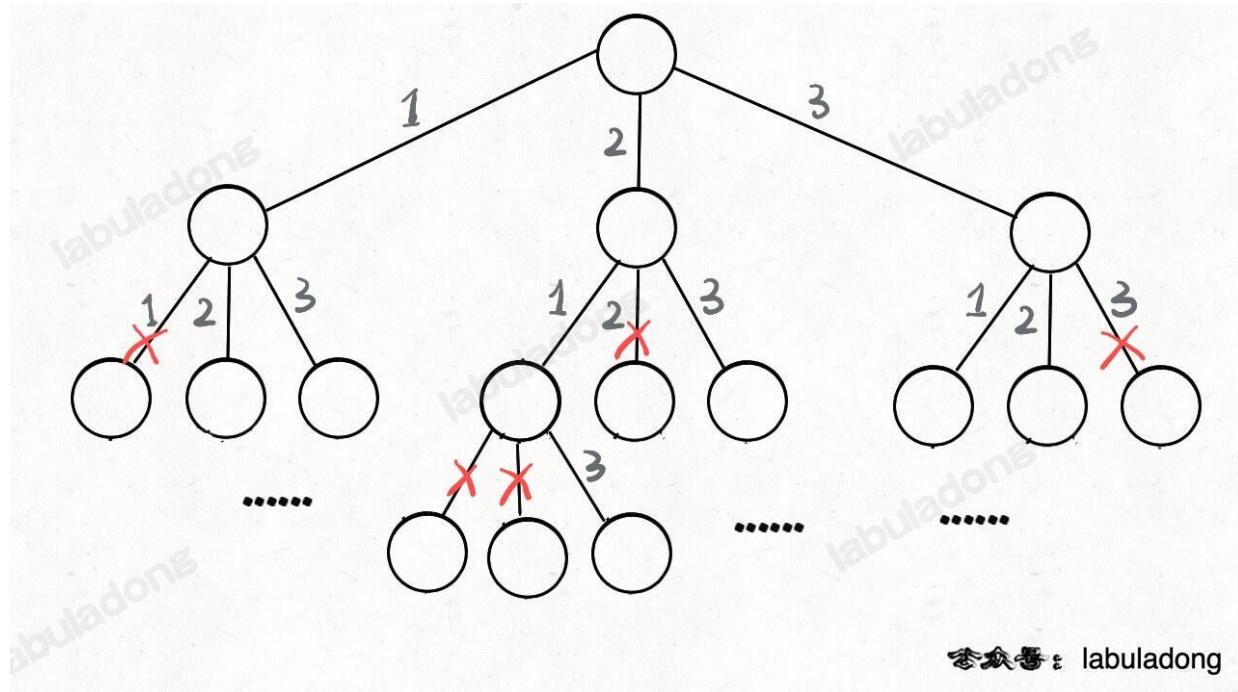
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            // nums[i] 已经在 track 中, 跳过
            continue;
        }
        // 做选择
        track.add(nums[i]);
        used[i] = true;
        // 进入下一层决策树
        backtrack(nums, track, used);
        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}

```

▶ 代码可视化动画

我们这里稍微做了些变通，没有显式记录「选择列表」，而是通过 `used` 数组排除已经存在 `track` 中的元素，从而推导出当前的选择列表：



至此，我们就通过全排列问题详解了回溯算法的底层原理。当然，这个算法解决全排列不是最高效的，你可能看到有的解法连 `used` 数组都不使用，通过交换元素达到目的。但是那种解法稍微难理解一些，我会在 [球盒模型：回溯算法两种穷举视角](#) 中介绍。

但是必须说明的是，不管怎么优化，都符合回溯框架，而且时间复杂度都不可能低于  $O(N!)$ ，因为穷举整棵决策树是无法避免的，你最后肯定要穷举出  $N!$  种全排列结果。

这也是回溯算法的一个特点，不像动态规划存在重叠子问题可以优化，回溯算法就是纯暴力穷举，复杂度一般都很高。

## 最后总结

回溯算法就是个多叉树的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作，算法框架如下：

```
def backtrack(...):
    for 选择 in 选择列表:
        做选择
        backtrack(...)
        撤销选择
```

写 **backtrack** 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

其实想想看，回溯算法和动态规划是不是有点像呢？我们在动态规划系列文章中多次强调，动态规划的三个需要明确的点就是「状态」「选择」和「base case」，是不是就对应着走过的「路径」，当前的「选择列表」和「结束条件」？

动态规划和回溯算法底层都把问题抽象成了树的结构，但这两种算法在思路上是完全不同的。在 [二叉树心法（纲领篇）](#) 你将看到动态规划和回溯算法更深层次的区别和联系。

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	<a href="#">111. 二叉树的最小深度</a>	
<a href="#">112. Path Sum</a>	<a href="#">112. 路径总和</a>	
<a href="#">113. Path Sum II</a>	<a href="#">113. 路径总和 II</a>	
<a href="#">131. Palindrome Partitioning</a>	<a href="#">131. 分割回文串</a>	
<a href="#">140. Word Break II</a>	<a href="#">140. 单词拆分 II</a>	
<a href="#">1593. Split a String Into the Max Number of Unique Substrings</a>	<a href="#">1593. 拆分字符串使唯一子字符串的数目最大</a>	
<a href="#">17. Letter Combinations of a Phone Number</a>	<a href="#">17. 电话号码的字母组合</a>	
<a href="#">22. Generate Parentheses</a>	<a href="#">22. 括号生成</a>	
<a href="#">301. Remove Invalid Parentheses</a>	<a href="#">301. 删除无效的括号</a>	
<a href="#">332. Reconstruct Itinerary</a>	<a href="#">332. 重新安排行程</a>	
<a href="#">39. Combination Sum</a>	<a href="#">39. 组合总和</a>	
<a href="#">51. N-Queens</a>	<a href="#">51. N 皇后</a>	
<a href="#">638. Shopping Offers</a>	<a href="#">638. 大礼包</a>	
<a href="#">698. Partition to K Equal Sum Subsets</a>	<a href="#">698. 划分为k个相等的子集</a>	
<a href="#">77. Combinations</a>	<a href="#">77. 组合</a>	
<a href="#">78. Subsets</a>	<a href="#">78. 子集</a>	
<a href="#">784. Letter Case Permutation</a>	<a href="#">784. 字母大小写全排列</a>	

LeetCode	力扣	难度
89. Gray Code	89. 格雷编码	
93. Restore IP Addresses	93. 复原 IP 地址	
-	剑指 Offer 34. 二叉树中和为某一值的路径	
-	剑指 Offer II 079. 所有子集	
-	剑指 Offer II 080. 含有 k 个元素的组合	
-	剑指 Offer II 081. 允许重复选择元素的组合	
-	剑指 Offer II 083. 没有重复元素集合的全排列	
-	剑指 Offer II 085. 生成匹配的括号	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

# BFS 算法解题套路框架



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	<a href="#">111. 二叉树的最小深度</a>	
<a href="#">752. Open the Lock</a>	<a href="#">752. 打开转盘锁</a>	

阅读本文前，你需要先学习：

- [多叉树基础及遍历](#)
- [图结构基础及通用实现](#)

后台有很多人问起 BFS 和 DFS 的框架，今天就来说说吧。

首先，你要说我没写过 BFS 框架，这话没错，今天写个框架你背住就完事儿了。但要是说没写过 DFS 框架，那你还真是说错了，**其实 DFS 算法就是回溯算法**，我们前文[回溯算法框架套路详解](#)就写过了，而且写得不是一般得好，建议好好复习，嘿嘿嘿~

BFS 的核心思想应该不难理解的，就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，我们写 BFS 算法都是用「队列」这种数据结构，每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：**BFS 找到的路径一定是最短的，但代价就是空间复杂度可能比 DFS 大很多**，至于为什么，我们后面介绍了框架就很容易看出来了。

本文就由浅入深写两道 BFS 的典型题目，分别是「二叉树的最小高度」和「打开密码锁的最少步数」，手把手教你怎么写 BFS 算法。

## 一、算法框架

要说框架的话，我们先举例一下 BFS 出现的常见场景好吧，**问题的本质就是让你在一幅「图」中找到从起点 **start** 到终点 **target** 的最近距离**，这个例子听起来很枯燥，但是**BFS 算法问题其实都是在干这个事儿**，把枯燥的本质搞清楚了，再去欣赏各种问题的包装才能胸有成竹嘛。

这个广义的描述可以有各种变体，比如走迷宫，有的格子是围墙不能走，从起点到终点的最短距离是多少？如果这个迷宫带「传送门」可以瞬间传送呢？

再比如说两个单词，要求你通过某些替换，把其中一个变成另一个，每次只能替换一个字符，最少要替换几次？

再比如说连连看游戏，两个方块消除的条件不仅仅是图案相同，还得保证两个方块之间的最短连线不能多于两个拐点。你玩连连看，点击两个坐标，游戏是如何判断它俩的最短连线有几个拐点的？

再比如……

净整些花里胡哨的，本质上看这些问题都没啥区别，就是一幅「图」，让你从一个起点，走到终点，问最短路径。这就是 BFS 的本质，框架搞清楚了直接默写就好。

记住下面这个框架就 OK 了：

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    // 核心数据结构
    Queue<Node> q;
    // 避免走回头路
    Set<Node> visited;

    // 将起点加入队列
    q.offer(start);
    visited.add(start);

    while (q not empty) {
        int sz = q.size();
        // 将当前队列中的所有节点向四周扩散
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            // 划重点：这里判断是否到达终点
            if (cur is target)
                return step;
            // 将 cur 的相邻节点加入队列
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
    }
    // 如果走到这里，说明在图中没有找到目标节点
}
```

队列 `q` 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

## 二、二叉树的最小高度

先来个简单的问题实践一下 BFS 框架吧，判断一棵二叉树的最小高度，这也是力扣第 111 题「二叉树的最小深度」：

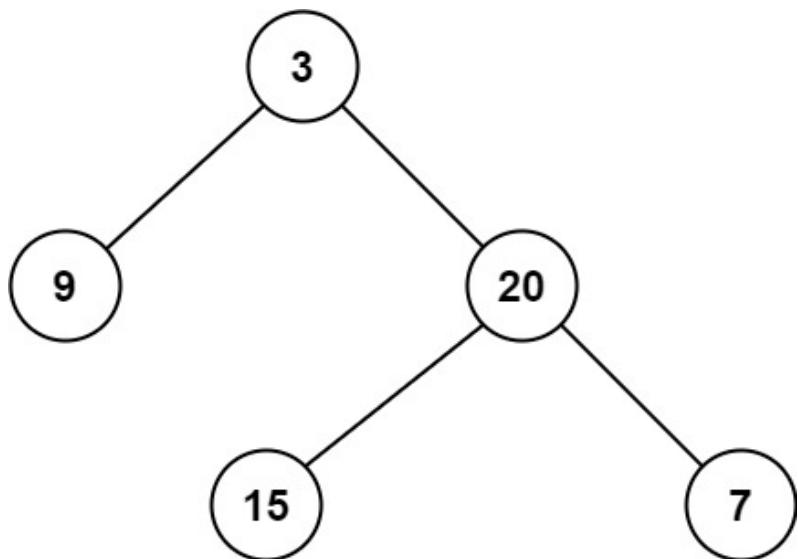
### ▼ 111. 二叉树的最小深度 [Leetcode | 力扣](#)

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

**说明：**叶子节点是指没有子节点的节点。

**示例 1：**



```
输入: root = [3,9,20,null,null,15,7]  
输出: 2
```

### 示例 2:

```
输入: root = [2,null,3,null,4,null,5,null,6]  
输出: 5
```

### 提示:

- 树中节点数的范围在  $[0, 10^5]$  内
- $-1000 \leq \text{node.val} \leq 1000$

怎么套到 BFS 的框架里呢？首先明确一下起点 `start` 和终点 `target` 是什么，怎么判断到达了终点？

显然起点就是 `root` 根节点，终点就是最靠近根节点的那个「叶子节点」嘛，叶子节点就是两个子节点都是 `null` 的节点：

```
if (cur.left == null && cur.right == null)  
    // 到达叶子节点
```

那么，按照我们上述的框架稍加改造来写解法即可：

```
class Solution {  
    public int minDepth(TreeNode root) {  
        if (root == null) return 0;  
        Queue<TreeNode> q = new LinkedList<>();  
        q.offer(root);  
        // root 本身就是一层, depth 初始化为 1  
        int depth = 1;  
  
        while (!q.isEmpty()) {  
            int sz = q.size();  
            // 将当前队列中的所有节点向四周扩散  
            for (int i = 0; i < sz; i++) {
```

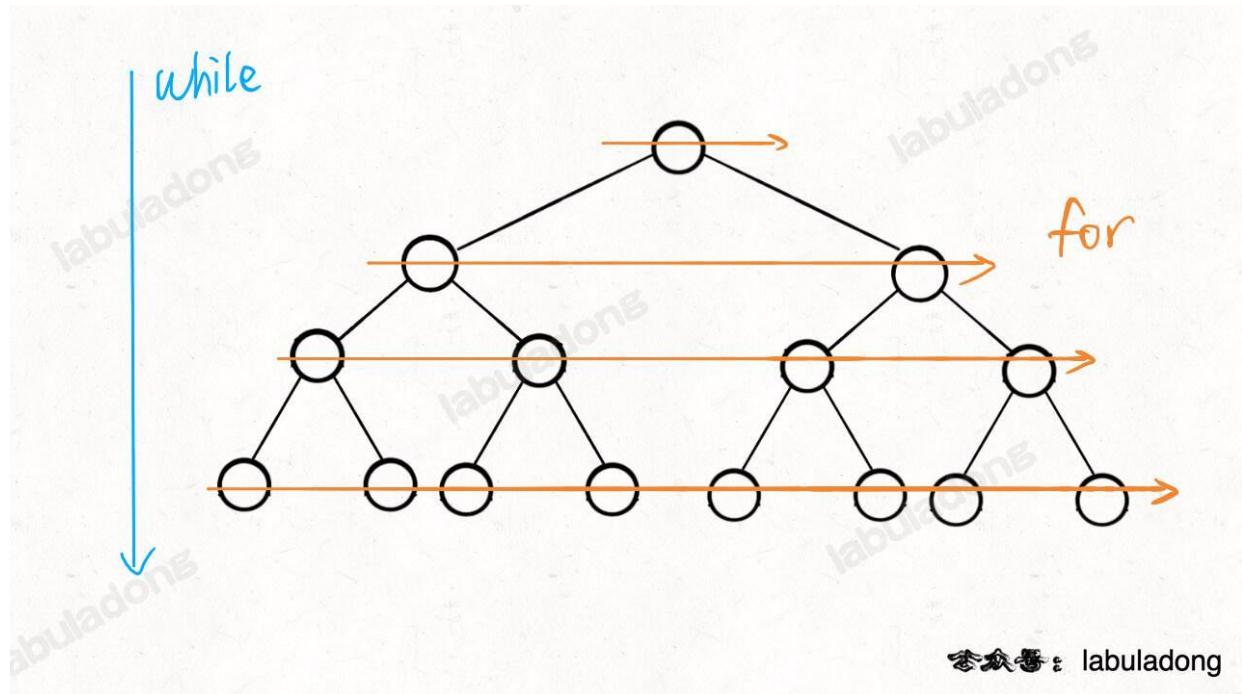
```

TreeNode cur = q.poll();
// 判断是否到达终点
if (cur.left == null && cur.right == null)
    return depth;
// 将 cur 的相邻节点加入队列
if (cur.left != null)
    q.offer(cur.left);
if (cur.right != null)
    q.offer(cur.right);
}
// 这里增加步数
depth++;
}
return depth;
}
}

```

### ▶ 🎥 代码可视化动画 🎥

这里注意这个 `while` 循环和 `for` 循环的配合，`while` 循环控制一层一层往下走，`for` 循环利用 `sz` 变量控制从左到右遍历每一层二叉树节点：



这一点很重要，这个形式在普通 BFS 问题中都很常见，但是在 Dijkstra 算法模板框架 中我们修改了这种代码模式，读完并理解本文后你可以去看看 BFS 算法是如何演变成 Dijkstra 算法在加权图中寻找最短路径的。

话说回来，二叉树本身是很简单的数据结构，我想上述代码你应该可以理解的，其实其他复杂问题都是这个框架的变形，再探讨复杂问题之前，我们解答两个问题：

#### 1、为什么 BFS 可以找到最短距离，DFS 不行吗？

首先，你看 BFS 的逻辑，`depth` 每增加一次，队列中的所有节点都向前迈一步，这保证了第一次到达终点的时候，走的步数是最少的。

DFS 不能找最短路径吗？其实也是可以的，但是时间复杂度相对高很多。你想啊，DFS 实际上是靠递归的堆栈记录走过的路径，你要找到最短路径，肯定得把二叉树中所有树权都探索完才能对比出最短的路径有多长对不对？而 BFS 借助队列做

到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离的。

形象点说，DFS 是线，BFS 是面；DFS 是单打独斗，BFS 是集体行动。这个应该比较容易理解吧。

## 2、既然 BFS 那么好，为啥 DFS 还要存在？

BFS 可以找到最短距离，但是空间复杂度高，而 DFS 的空间复杂度较低。

还是拿刚才我们处理二叉树问题的例子，假设给你的这个二叉树是满二叉树，节点数为  $N$ ，对于 DFS 算法来说，空间复杂度无非就是递归堆栈，最坏情况下顶多就是树的高度，也就是  $O(\log N)$ 。

但是你想想 BFS 算法，队列中每次都会储存着二叉树一层的节点，这样的话最坏情况下空间复杂度应该是树的最底层节点的数量，也就是  $N/2$ ，用 Big O 表示的话也就是  $O(N)$ 。

由此观之，BFS 还是有代价的，一般来说在找最短路径的时候使用 BFS，其他时候还是 DFS 使用得多一些（主要是递归代码好写）。

好了，现在你对 BFS 了解得足够多了，下面来一道难一点的题目，深化一下框架的理解吧。

## 三、解开密码锁的最少次数

这是力扣第 752 题「打开转盘锁」，比较有意思：

### ▼ 752. 打开转盘锁 [Leetcode | 力扣](#)

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有 10 个数字： $\text{0000000000}$ ,  $\text{0000100000}$ ,  $\text{0000200000}$ ,  
 $\text{0000300000}$ ,  $\text{0000400000}$ ,  $\text{0000500000}$ ,  $\text{0000600000}$ ,  $\text{0000700000}$ ,  $\text{0000800000}$ ,  
 $\text{0000900000}$ 。每个拨轮可以自由旋转：例如把  $\text{0000900000}$  变为  $\text{0000000000}$ ,  $\text{0000000000}$  变为  
 $\text{0000900000}$ 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为  $\text{0000000000}$ ，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回  $-1$ 。

### 示例 1:

```
输入: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出: 6
解释:
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的,
因为当拨动到 "0102" 时这个锁就会被锁定。
```

### 示例 2:

```
输入: deadends = ["8888"], target = "0009"
输出: 1
解释: 把最后一位反向旋转一次即可 "0000" -> "0009"。
```

### 示例 3:

**输入:** deadends = ["8887", "8889", "8878", "8898", "8788", "8988", "7888", "9888"], target = "8888"  
**输出:** -1  
**解释:** 无法旋转到目标数字且不被锁定。

提示:

- `1 <= deadends.length <= 500`
- `deadends[i].length == 4`
- `target.length == 4`
- `target` 不在 `deadends` 之中
- `target` 和 `deadends[i]` 仅由若干位数字组成

函数签名如下:

```
int openLock(String[] deadends, String target)
```

题目中描述的就是我们生活中常见的那种密码锁，如果没有任何约束，最少的拨动次数很好算，就像我们平时开密码锁那样直奔密码拨就行了。

但现在的难点就在于，不能出现 `deadends`，应该如何计算出最少的转动次数呢？

第一步，我们不管所有的限制条件，不管 `deadends` 和 `target` 的限制，就思考一个问题：如果让你设计一个算法，穷举所有可能的密码组合，你怎么做？

穷举呗，再简单一点，如果你只转一下锁，有几种可能？总共有 4 个位置，每个位置可以向上转，也可以向下转，也就是有 8 种可能对吧。

比如说从 "`0000`" 开始，转一次，可以穷举出 "`1000`", "`9000`", "`0100`", "`0900`..." 共 8 种密码。然后，再以这 8 种密码作为基础，对每个密码再转一下，穷举出所有可能...

仔细想想，这就可以抽象成一幅图，每个节点有 8 个相邻的节点，又让你求最短距离，这不就是典型的 BFS 嘛，框架就可以派上用场了，先写出一个「简陋」的 BFS 框架代码再说别的：

```
// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}

// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
```

```
// BFS 框架，打印出所有可能的密码
void BFS(String target) {
    Queue<String> q = new LinkedList<>();
    q.offer("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        // 将当前队列中的所有节点向周围扩散
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            // 判断是否到达终点
            System.out.println(cur);

            // 将一个节点的相邻节点加入队列
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                String down = minusOne(cur, j);
                q.offer(up);
                q.offer(down);
            }
        }
        // 在这里增加步数
    }
    return;
}
```

这段代码当然有很多问题，但是我们做算法题肯定不是一蹴而就的，而是从简陋到完美的。不要完美主义，咱要慢慢来，好不。

这段 BFS 代码已经能够穷举所有可能的密码组合了，但是显然不能完成题目，有如下问题需要解决：

- 1、会走回头路。比如说我们从 "0000" 拨到 "1000"，但是等从队列拿出 "1000" 时，还会拨出一个 "0000"，这样的话会产生死循环。
- 2、没有终止条件，按照题目要求，我们找到 `target` 就应该结束并返回拨动的次数。
- 3、没有对 `deadends` 的处理，按道理这些「死亡密码」是不能出现的，也就是说你遇到这些密码的时候需要跳过。

如果你能够看懂上面那段代码，真得给你鼓掌，只要按照 BFS 框架在对应的位置稍作修改即可修复这些问题：

```
class Solution {
    public int openLock(String[] deadends, String target) {
        // 记录需要跳过的死亡密码
        Set<String> deads = new HashSet<>();
        for (String s : deadends) deads.add(s);
        // 记录已经穷举过的密码，防止走回头路
        Set<String> visited = new HashSet<>();
        Queue<String> q = new LinkedList<>();
        // 从起点开始启动广度优先搜索
        int step = 0;
        q.offer("0000");
        visited.add("0000");

        while (!q.isEmpty()) {
            int sz = q.size();
```

```

    // 将当前队列中的所有节点向周围扩散
    for (int i = 0; i < sz; i++) {
        String cur = q.poll();

        // 判断是否到达终点
        if (deads.contains(cur))
            continue;
        if (cur.equals(target))
            return step;

        // 将一个节点的未遍历相邻节点加入队列
        for (int j = 0; j < 4; j++) {
            String up = plusOne(cur, j);
            if (!visited.contains(up)) {
                q.offer(up);
                visited.add(up);
            }
            String down = minusOne(cur, j);
            if (!visited.contains(down)) {
                q.offer(down);
                visited.add(down);
            }
        }
    }
    // 在这里增加步数
    step++;
}
// 如果穷举完都没找到目标密码，那就是找不到了
return -1;
}

// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}

// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
}

```

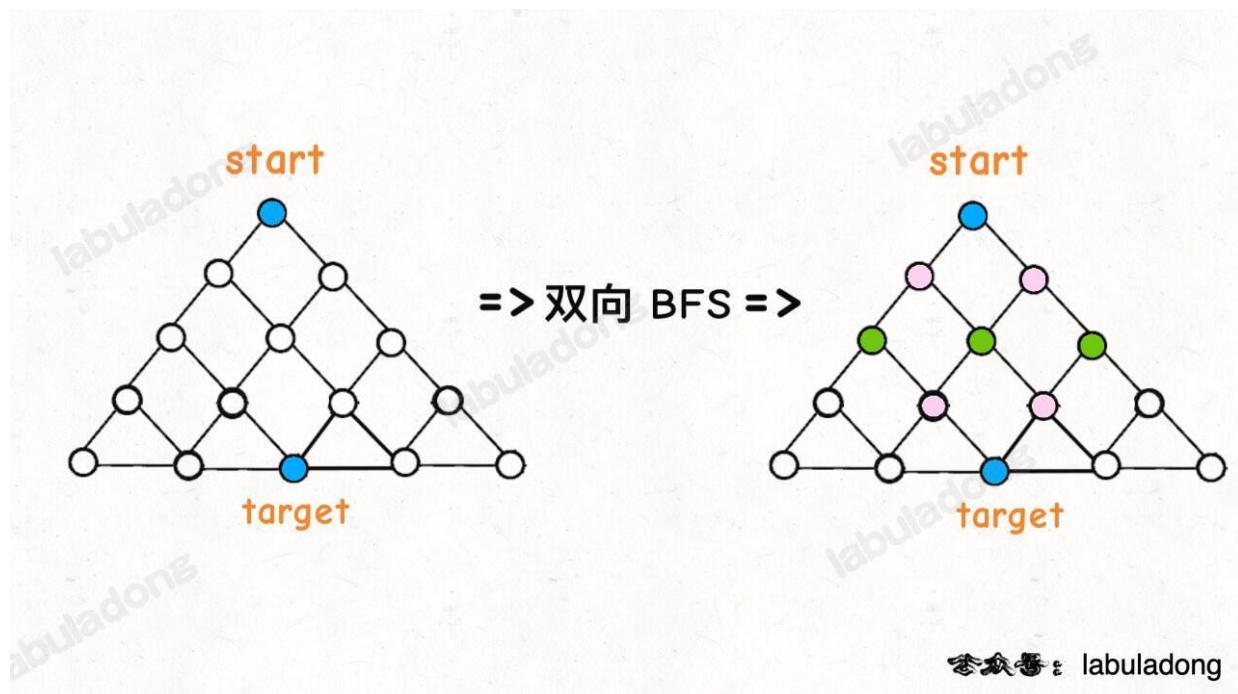
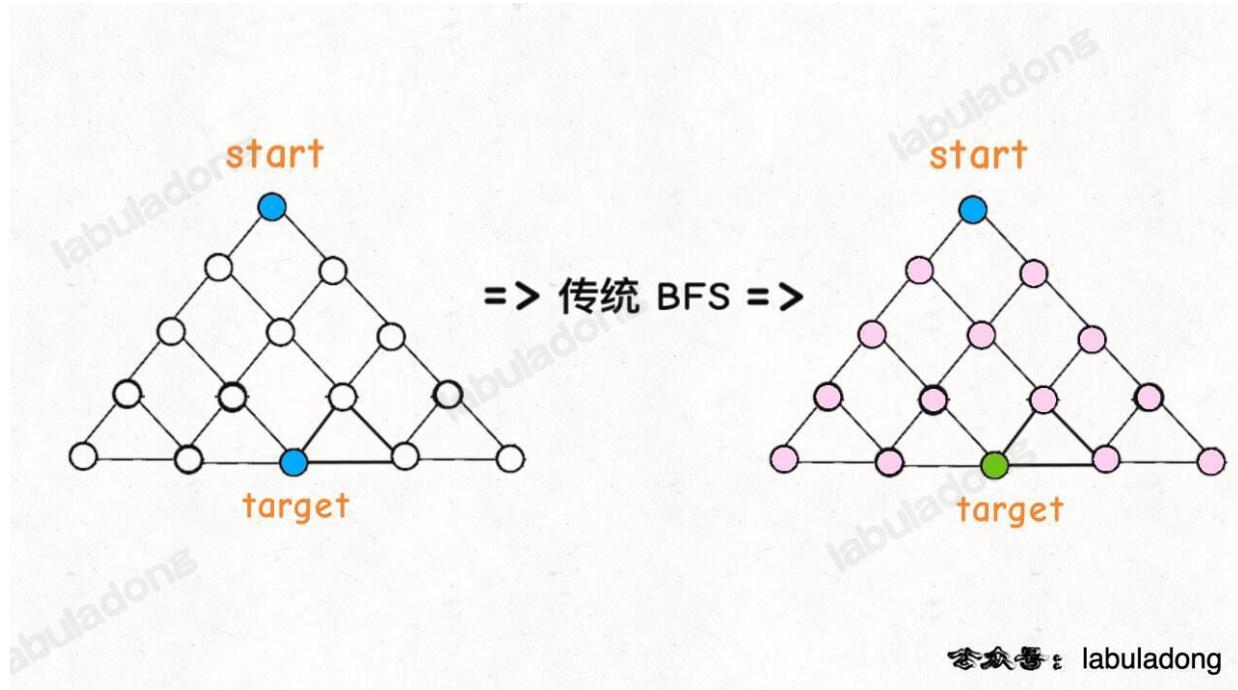
至此，我们就解决这道题目了。有一个比较小的优化：可以不需要 `dead` 这个哈希集合，可以直接将这些元素初始化到 `visited` 集合中，效果是一样的，可能更加优雅一些。

## 四、双向 BFS 优化

你以为到这里 BFS 算法就结束了？恰恰相反。BFS 算法还有一种稍微高级一点的优化思路：**双向 BFS**，可以进一步提高算法的效率。

篇幅所限，这里就提一下区别：传统的 BFS 框架就是从起点开始向四周扩散，遇到终点时停止；而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。

为什么这样能够提升效率呢？其实从 Big O 表示法分析算法复杂度的话，它俩的最坏复杂度都是  $O(N)$ ，但是实际上双向 BFS 确实会快一些，我给你画两张图看一眼就明白了：



图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到 target；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。从这个例子可以直观地感受到，双向 BFS 是要比传统 BFS 高效的。

不过，双向 BFS 也有局限，因为你必须知道终点在哪里。比如我们刚才讨论的二叉树最小高度的问题，你一开始根本就不知道终点在哪里，也就无法使用双向 BFS；但是第二个密码锁的问题，是可以使用双向 BFS 算法来提高效率的，代码稍加修改即可：

```

class Solution {
    public int openLock(String[] deadends, String target) {
        Set<String> deads = new HashSet<>();
        for (String s : deadends) deads.add(s);
        // 用集合不用队列，可以快速判断元素是否存在
        Set<String> q1 = new HashSet<>();
        Set<String> q2 = new HashSet<>();
        Set<String> visited = new HashSet<>();

        int step = 0;
        q1.add("0000");
        q2.add(target);

        while (!q1.isEmpty() && !q2.isEmpty()) {
            // 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果
            Set<String> temp = new HashSet<>();

            // 将 q1 中的所有节点向周围扩散
            for (String cur : q1) {
                // 判断是否到达终点
                if (deads.contains(cur))
                    continue;
                if (q2.contains(cur))
                    return step;

                visited.add(cur);

                // 将一个节点的未遍历相邻节点加入集合
                for (int j = 0; j < 4; j++) {
                    String up = plusOne(cur, j);
                    if (!visited.contains(up))
                        temp.add(up);
                    String down = minusOne(cur, j);
                    if (!visited.contains(down))
                        temp.add(down);
                }
            }
            // 在这里增加步数
            step++;
            // temp 相当于 q1
            // 这里交换 q1 q2，下一轮 while 就是扩散 q2
            q1 = q2;
            q2 = temp;
        }
        return -1;
    }

    // 将 s[j] 向上拨动一次
    String plusOne(String s, int j) {
        char[] ch = s.toCharArray();
        if (ch[j] == '9')
            ch[j] = '0';
        else
            ch[j] += 1;
        return new String(ch);
    }

    // 将 s[i] 向下拨动一次
}

```

```
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
```

双向 BFS 还是遵循 BFS 算法框架的，只是不再使用队列，而是使用 HashSet 方便快速判断两个集合是否有交集。

另外的一个技巧点就是 while 循环的最后交换 q1 和 q2 的内容，所以只要默认扩散 q1 就相当于轮流扩散 q1 和 q2。

其实双向 BFS 还有一个优化，就是在 while 循环开始时做一个判断：

```
// ...
while (!q1.isEmpty() && !q2.isEmpty()) {
    if (q1.size() > q2.size()) {
        // 交换 q1 和 q2
        temp = q1;
        q1 = q2;
        q2 = temp;
    }
    // ...
}
```

为什么这是一个优化呢？

因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，无论传统 BFS 还是双向 BFS，无论做不做优化，从 Big O 衡量标准来看，时间复杂度都是一样的，只能说双向 BFS 是一种 trick，算法运行的速度会相对快一点，掌握不掌握其实都无所谓。最关键的是把 BFS 通用框架记下来，反正所有 BFS 算法都可以用它套出解法。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1091. Shortest Path in Binary Matrix</a>	<a href="#">1091. 二进制矩阵中的最短路径</a>	
<a href="#">117. Populating Next Right Pointers in Each Node II</a>	<a href="#">117. 填充每个节点的下一个右侧节点指针 II</a>	
<a href="#">127. Word Ladder</a>	<a href="#">127. 单词接龙</a>	
<a href="#">1926. Nearest Exit from Entrance in Maze</a>	<a href="#">1926. 迷宫中离入口最近的出口</a>	
<a href="#">2850. Minimum Moves to Spread Stones Over Grid</a>	<a href="#">2850. 将石头分散到网格图的最少移动次数</a>	
<a href="#">286. Walls and Gates</a>	<a href="#">286. 墙与门</a>	
<a href="#">310. Minimum Height Trees</a>	<a href="#">310. 最小高度树</a>	

LeetCode	力扣	难度
329. Longest Increasing Path in a Matrix	329. 矩阵中的最长递增路径	
365. Water and Jug Problem	365. 水壶问题	
431. Encode N-ary Tree to Binary Tree	431. 将 N 叉树编码为二叉树	
433. Minimum Genetic Mutation	433. 最小基因变化	
490. The Maze	490. 迷宫	
505. The Maze II	505. 迷宫 II	
542. 01 Matrix	542. 01 矩阵	
547. Number of Provinces	547. 省份数量	
773. Sliding Puzzle	773. 滑动谜题	
863. All Nodes Distance K in Binary Tree	863. 二叉树中所有距离为 K 的结点	
994. Rotting Oranges	994. 腐烂的橘子	
-	剑指 Offer II 109. 开密码锁	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 二叉树系列算法核心纲领



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">144. Binary Tree Preorder Traversal</a>	<a href="#">144. 二叉树的前序遍历</a>	
<a href="#">543. Diameter of Binary Tree</a>	<a href="#">543. 二叉树的直径</a>	
<a href="#">104. Maximum Depth of Binary Tree</a>	<a href="#">104. 二叉树的最大深度</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

本文会把很多算法进行抽象和归纳，所以会包含大量其他文章链接。

第一次阅读本文的读者不要 DFS 学习本文，遇到没学过的算法或不理解的地方请跳过，只要对本文所总结的理论有些印象即可。在学习本站后面的算法技巧时，你自然可以逐渐理解本文的精髓所在，日后回来重读本文，会有更深的体会。

本站所有文章的脉络都是按照 [学习数据结构和算法的框架思维](#) 提出的框架来构建的，其中着重强调了二叉树题目的重要性，所以把本文放在第一章的必读系列中。

我刷了这么多年题，浓缩出二叉树算法的一个总纲放在这里，也许用词不是特别专业化，也没有什么教材会收录我的这些经验总结，但目前各个刷题平台的题库，没有一道二叉树题目能跳出本文划定的框架。如果你能发现一道题目和本文给出的框架不兼容，请留言告知我。

先在开头总结一下，二叉树解题的思维模式分两类：

1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现，这叫「遍历」的思维模式。

2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「分解问题」的思维模式。

无论使用哪种思维模式，你都需要思考：

如果单独抽出一个二叉树节点，它需要做什么事情？需要在什么时候（前/中/后序位置）做？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。

本文中会用题目来举例，但都是最最简单的题目，所以不用担心自己看不懂，我可以帮你从最简单的问题中提炼出所有二叉树题目的共性，并将二叉树中蕴含的思维进行升华，反手用到 [动态规划](#), [回溯算法](#), [分治算法](#), [图论算法](#) 中去，这也是我一直强调框架思维的原因。希望你在学习了上述高级算法后，也能回头再来看看本文，会对它们有更深刻的认识。

首先，我还是要不厌其烦地强调一下二叉树这种数据结构及相关算法的重要性。

## 二叉树的重要性

举个例子，比如两个经典排序算法 [快速排序](#) 和 [归并排序](#)，对于它俩，你有什么理解？

如果你告诉我，快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历，那么我就知道你是个算法高手了。

为什么快速排序和归并排序能和二叉树扯上关系？我们来简单分析一下他们的算法思想和代码框架：

快速排序的逻辑是，若要对 `nums[lo..hi]` 进行排序，我们先找一个分界点 `p`，通过交换元素使得 `nums[lo..p-1]` 都小于等于 `nums[p]`，且 `nums[p+1..hi]` 都大于 `nums[p]`，然后递归地去 `nums[lo..p-1]` 和 `nums[p+1..hi]` 中寻找新的分界点，最后整个数组就被排序了。

快速排序的代码框架如下：

```
void sort(int[] nums, int lo, int hi) {  
    // ***** 前序遍历位置 *****  
    // 通过交换元素构建分界点 p  
    int p = partition(nums, lo, hi);  
    // *****  
  
    sort(nums, lo, p - 1);  
    sort(nums, p + 1, hi);  
}
```

先构造分界点，然后去左右子数组构造分界点，你看这不就是一个二叉树的前序遍历吗？

再说说归并排序的逻辑，若要对 `nums[lo..hi]` 进行排序，我们先对 `nums[lo..mid]` 排序，再对 `nums[mid+1..hi]` 排序，最后把这两个有序的子数组合并，整个数组就排好序了。

归并排序的代码框架如下：

```
// 定义：排序 nums[lo..hi]  
void sort(int[] nums, int lo, int hi) {  
    int mid = (lo + hi) / 2;  
    // 排序 nums[lo..mid]  
    sort(nums, lo, mid);  
    // 排序 nums[mid+1..hi]  
    sort(nums, mid + 1, hi);  
  
    // ***** 后序位置 *****  
    // 合并 nums[lo..mid] 和 nums[mid+1..hi]  
    merge(nums, lo, mid, hi);  
    // *****  
}
```

先对左右子数组排序，然后合并（类似合并有序链表的逻辑），你看这是不是二叉树的后序遍历框架？另外，这不就是传说中的分治算法嘛，不过如此呀。

如果你一眼就识破这些排序算法的底细，还需要背这些经典算法吗？不需要。你可以手到擒来，从二叉树遍历框架就能扩展出算法了。

说了这么多，旨在说明，二叉树的算法思想的运用广泛，甚至可以说，只要涉及递归，都可以抽象成二叉树的问题。

接下来我们从二叉树的前中后序开始讲起，让你深刻理解这种数据结构的魅力。

## 深入理解前中后序

我先甩给你几个问题，请默默思考 30 秒：

- 1、你理解的二叉树的前中后序遍历是什么，仅仅是三个顺序不同的 List 吗？
- 2、请分析，后序遍历有什么特殊之处？
- 3、请分析，为什么多叉树没有中序遍历？

答不上来，说明你对前中后序的理解仅仅局限于教科书，不过没关系，我用类比的方式解释一下我眼中的前中后序遍历。

首先，回顾一下 [二叉树的 DFS/BFS 遍历](#) 中说到的二叉树递归遍历框架：

```
void traverse(TreeNode root) {  
    if (root == null) {  
        return;  
    }  
    // 前序位置  
    traverse(root.left);  
    // 中序位置  
    traverse(root.right);  
    // 后序位置  
}
```

先不管所谓前中后序，单看 `traverse` 函数，你说它在做什么事情？

其实它就是一个能够遍历二叉树所有节点的一个函数，和你遍历数组或者链表本质上没有区别：

```
// 迭代遍历数组  
void traverse(int[] arr) {  
    for (int i = 0; i < arr.length; i++) {  
  
    }  
}  
  
// 递归遍历数组  
void traverse(int[] arr, int i) {  
    if (i == arr.length) {  
        return;  
    }  
    // 前序位置  
    traverse(arr, i + 1);  
    // 后序位置  
}  
  
// 迭代遍历单链表  
void traverse(ListNode head) {  
    for (ListNode p = head; p != null; p = p.next) {
```

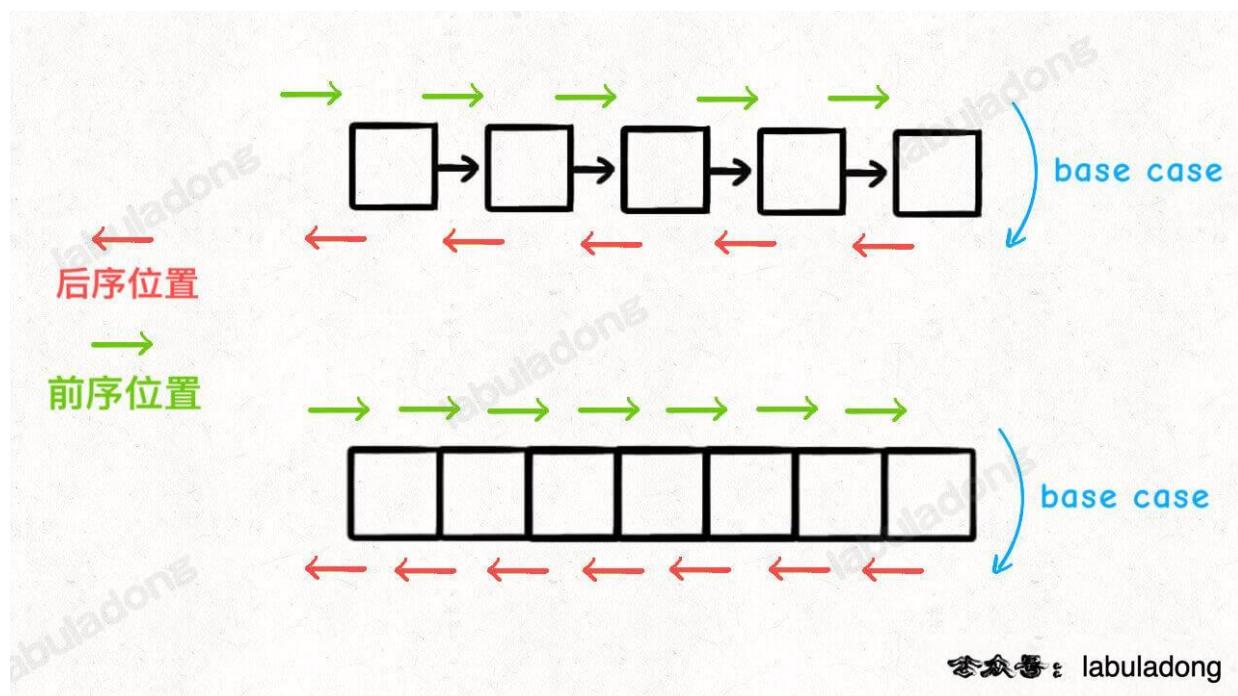
```
    }
}

// 递归遍历单链表
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    // 前序位置
    traverse(head.next);
    // 后序位置
}
```

单链表和数组的遍历可以是迭代的，也可以是递归的，二叉树这种结构无非就是二叉链表，它没办法简单改写成 for 循环的迭代形式，所以我们遍历二叉树一般都使用递归形式。

你也注意到了，只要是递归形式的遍历，都可以有前序位置和后序位置，分别在递归之前和递归之后。

所谓前序位置，就是刚进入一个节点（元素）的时候，后序位置就是即将离开一个节点（元素）的时候，那么进一步，你把代码写在不同位置，代码执行的时机也不同：



比如说，如果让你倒序打印一条单链表上所有节点的值，你怎么搞？

实现方式当然有很多，但如果你对递归的理解足够透彻，可以利用后序位置来操作：

```
// 递归遍历单链表，倒序打印链表元素
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    traverse(head.next);
    // 后序位置
    print(head.val);
}
```

结合上面那张图，你应该知道为什么这段代码能够倒序打印单链表了吧，本质上是利用递归的堆栈帮你实现了倒序遍历的效果。

那么说回二叉树也是一样的，只不过多了一个中序位置罢了。

教科书里只会问你前中后序遍历结果分别是什么，所以对于一个只上过大学数据结构课程的人来说，他大概以为二叉树的前中后序只不过对应三种顺序不同的 `List<Integer>` 列表。

但是我想说，**前中后序是遍历二叉树过程中处理每一个节点的三个特殊时间点**，绝不仅仅是三个顺序不同的 List：

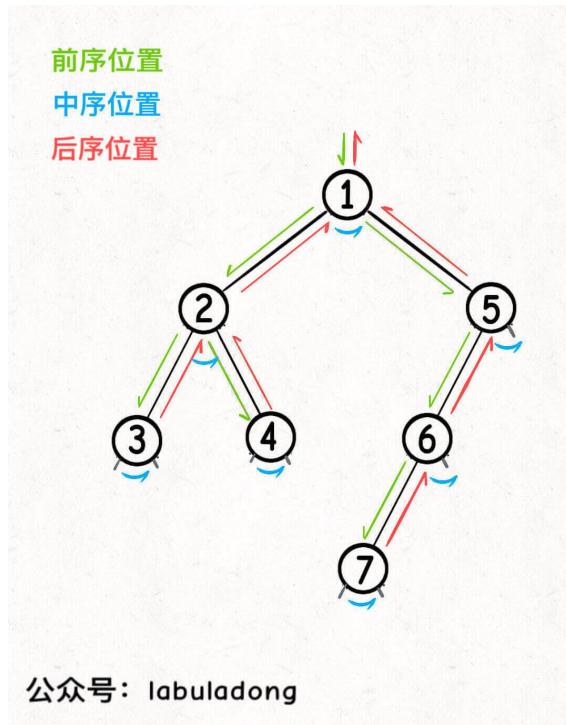
前序位置的代码在刚刚进入一个二叉树节点的时候执行；

后序位置的代码在将要离开一个二叉树节点的时候执行；

中序位置的代码在一个二叉树节点左子树都遍历完，即将开始遍历右子树的时候执行。

你注意本文的用词，我一直说前中后序「位置」，就是要和大家常说的前中后序「遍历」有所区别：你可以在前序位置写代码往一个 List 里面塞元素，那最后得到的就是前序遍历结果；但并不是说你就不可以写更复杂的代码做更复杂的事。

画成图，前中后序三个位置在二叉树上是这样：



你可以发现每个节点都有「唯一」属于自己的前中后序位置，所以我说前中后序遍历是遍历二叉树过程中处理每一个节点的三个特殊时间点。

这里你也可以理解为什么多叉树没有中序位置，因为二叉树的每个节点只会进行唯一一次左子树切换右子树，而多叉树节点可能有很多子节点，会多次切换子树去遍历，所以多叉树节点没有「唯一」的中序遍历位置。

说了这么多基础的，就是要帮你对二叉树建立正确的认识，然后你会发现：

**二叉树的所有问题，就是让你在前中后序位置注入巧妙的代码逻辑，去达到自己的目的，你只需要单独思考每一个节点应该做什么，其他的不用你管，抛给二叉树遍历框架，递归会在所有节点上做相同的操作。**

你也可以看到，[图论算法基础](#) 把二叉树的遍历框架扩展到了图，并以遍历为基础实现了图论的各种经典算法，不过这是后话，本文就不多说了。

## 两种解题思路

前文 [我的算法学习心得](#) 说过：

二叉树题目的递归解法可以分两类思路，第一类是遍历一遍二叉树得出答案，第二类是通过分解问题计算出答案，这两类思路分别对应着 [回溯算法核心框架](#) 和 [动态规划核心框架](#)。

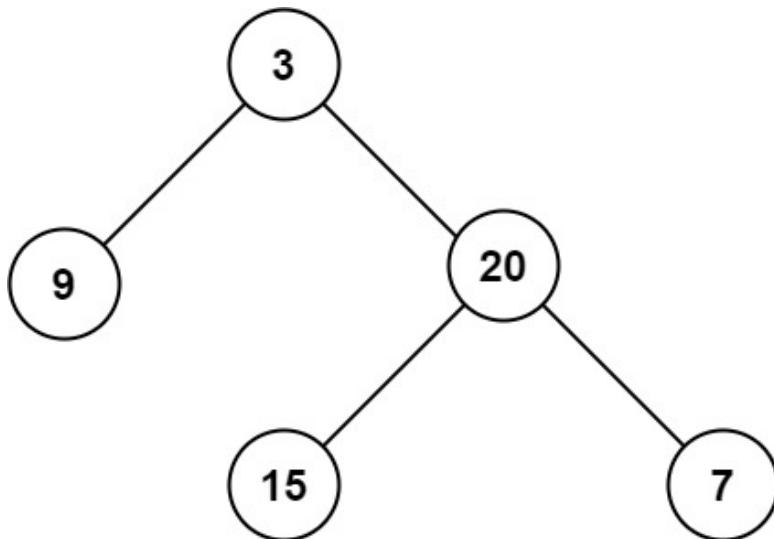
这里说一下我的函数命名习惯：二叉树中用遍历思路解题时函数签名一般是 `void traverse(...)`，没有返回值，靠更新外部变量来计算结果，而用分解问题思路解题时函数名根据该函数具体功能而定，而且一般会有返回值，返回值是子问题的计算结果。

与此对应的，你会发现我在 [回溯算法核心框架](#) 中给出的函数签名一般也是没有返回值的 `void backtrack(...)`，而在 [动态规划核心框架](#) 中给出的函数签名是带有返回值的 `dp` 函数。这也说明它俩和二叉树之间千丝万缕的联系。

虽然函数命名没有什么硬性的要求，但我还是建议你也遵循我的这种风格，这样更能突出函数的作用和解题的思维模式，便于你自己理解和运用。

当时我是用二叉树的最大深度这个问题来举例，重点在于把这两种思路和动态规划和回溯算法进行对比，而本文的重点在于分析这两种思路如何解决二叉树的题目。

力扣第 104 题「二叉树的最大深度」就是最大深度的题目，所谓最大深度就是根节点到「最远」叶子节点的最长路径上的节点数，比如输入这棵二叉树，算法应该返回 3：



你做这题的思路是什么？显然遍历一遍二叉树，用一个外部变量记录每个节点所在的深度，取最大值就可以得到最大深度，这就是遍历二叉树计算答案的思路。

解法代码如下：

```
class Solution {
    // 记录最大深度
    int res = 0;

    // 记录遍历到的节点的深度
    int depth = 0;

    public int maxDepth(TreeNode root) {
        traverse(root);
        return res;
    }

    void traverse(TreeNode node) {
        if (node == null) {
            return;
        }
        depth++;
        maxDepth(node.left);
        maxDepth(node.right);
        res = Math.max(res, depth);
        depth--;
    }
}
```

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    depth++;
    if (root.left == null && root.right == null) {
        // 到达叶子节点，更新最大深度
        res = Math.max(res, depth);
    }
    traverse(root.left);
    traverse(root.right);
    // 后序位置
    depth--;
}
}
```

---

▶ [代码可视化动画](#)

---

这个解法应该很好理解，但为什么需要在前序位置增加 `depth`，在后序位置减小 `depth`？

因为前面说了，前序位置是进入一个节点的时候，后序位置是离开一个节点的时候，`depth` 记录当前递归到的节点深度，你把 `traverse` 理解成在二叉树上游走的一个指针，所以当然要这样维护。

至于对 `res` 的更新，你放到前中后序位置都可以，只要保证在进入节点之后，离开节点之前（即 `depth` 自增之后，自减之前）就行了。

当然，你也很容易发现一棵二叉树的最大深度可以通过子树的最大深度推导出来，这就是分解问题计算答案的思路。

解法代码如下：

```
class Solution {
    // 定义：输入根节点，返回这棵二叉树的最大深度
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 利用定义，计算左右子树的最大深度
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 整棵树的最大深度等于左右子树的最大深度取最大值，
        // 然后再加上根节点自己
        int res = Math.max(leftMax, rightMax) + 1;

        return res;
    }
}
```

---

▶ [代码可视化动画](#)

---

只要明确递归函数的定义，这个解法也不难理解，但为什么主要的代码逻辑集中在后序位置？

因为这个思路正确的核心在于，你确实可以通过子树的最大深度推导出原树的深度，所以当然要首先利用递归函数的定义算出左右子树的最大深度，然后推出原树的最大深度，主要逻辑自然放在后序位置。

如果你理解了最大深度这个问题的两种思路，那么我们再回头看看最基本的二叉树前中后序遍历，就比如力扣第 144 题「二叉树的前序遍历」，让你计算前序遍历结果。

我们熟悉的解法就是用「遍历」的思路，我想应该没什么好说的：

```
class Solution {
    // 存放前序遍历结果
    List<Integer> res = new LinkedList<>();

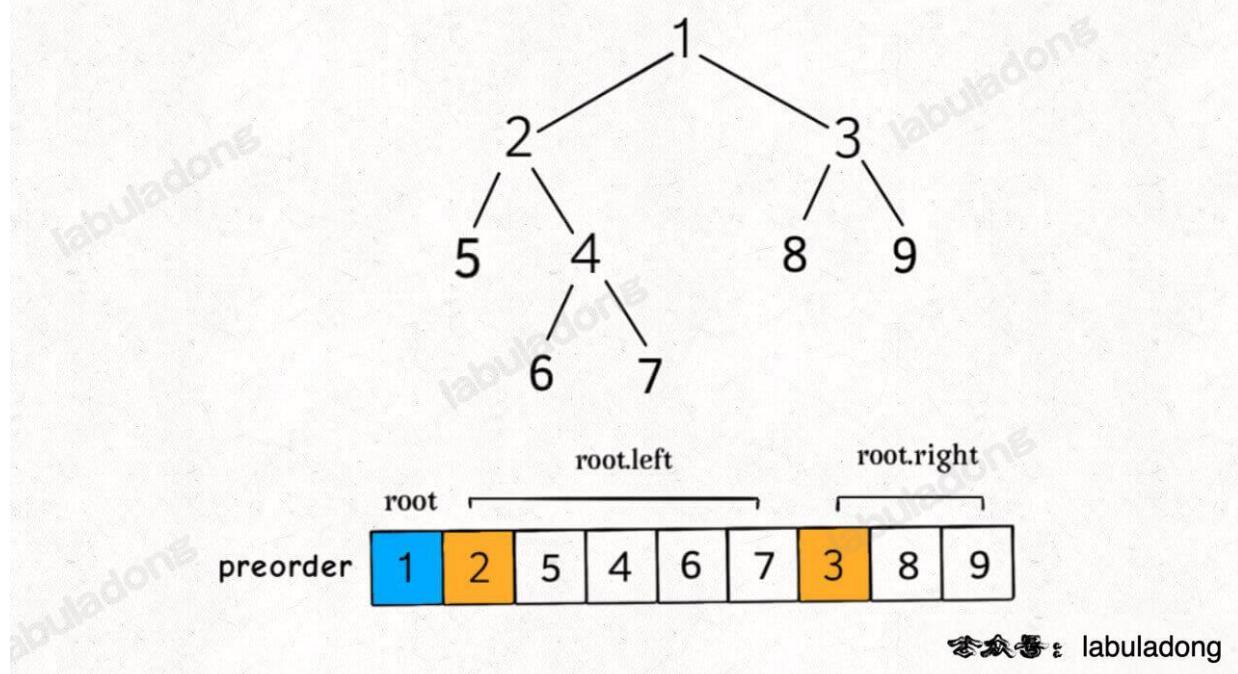
    // 返回前序遍历结果
    public List<Integer> preorderTraversal(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        // 前序位置
        res.add(root.val);
        traverse(root.left);
        traverse(root.right);
    }
}
```

但你是否能够用「分解问题」的思路，来计算前序遍历的结果？

换句话说，不要用像 `traverse` 这样的辅助函数和任何外部变量，单纯用题目给的 `preorderTraversal` 函数递归解题，你会不会？

我们知道前序遍历的特点是，根节点的值排在首位，接着是左子树的前序遍历结果，最后是右子树的前序遍历结果：



那这不就可以分解问题了么，一棵二叉树的前序遍历结果 = 根节点 + 左子树的前序遍历结果 + 右子树的前序遍历结果。

所以，你可以这样实现前序遍历算法：

```
class Solution {  
    // 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果  
    List<Integer> preorderTraversal(TreeNode root) {  
        List<Integer> res = new LinkedList<>();  
        if (root == null) {  
            return res;  
        }  
        // 前序遍历的结果，root.val 在第一个  
        res.add(root.val);  
        // 利用函数定义，后面接着左子树的前序遍历结果  
        res.addAll(preorderTraversal(root.left));  
        // 利用函数定义，最后接着右子树的前序遍历结果  
        res.addAll(preorderTraversal(root.right));  
        return res;  
    }  
}
```

中序和后序遍历也是类似的，只要把 `add(root.val)` 放到中序和后序对应的位置就行了。

这个解法短小精干，但为什么不常见呢？

一个原因是这个算法的复杂度不好把控，比较依赖语言特性。

Java 的话无论 ArrayList 还是 LinkedList，`addAll` 方法的复杂度都是  $O(N)$ ，所以总体的最坏时间复杂度会达到  $O(N^2)$ ，除非你自己实现一个复杂度为  $O(1)$  的 `addAll` 方法，底层用链表的话是可以做到的，因为多条链表只要简单的指针操作就能连接起来。

当然，最主要的原因还是因为教科书上从来没有这么教过……

上文举了两个简单的例子，但还有不少二叉树的题目是可以同时使用两种思路来思考和求解的，这就要靠你自己多去练习和思考，不要仅仅满足于一种熟悉的解法思路。

综上，遇到一道二叉树的题目时的通用思考过程是：

- 1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现。
- 2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值。
- 3、无论使用哪一种思维模式，你都要明白二叉树的每一个节点需要做什么，需要在什么时候（前中后序）做。

本站 [二叉树递归专项练习](#) 中列举了 100 多道二叉树习题，完全使用上述两种思维模式手把手带你练习，助你完全掌握递归思维，更容易理解高级的算法。

## 后序位置的特殊之处

说后序位置之前，先简单说下前序和中序。

前序位置本身其实没有什么特别的性质，之所以你发现好像很多题都是在前序位置写代码，实际上是因为我们习惯把那些对前中后序位置不敏感的代码写在前序位置罢了。

中序位置主要用在 BST 场景中，你完全可以把 BST 的中序遍历认为是遍历有序数组。

仔细观察，前中后序位置的代码，能力依次增强。

前序位置的代码只能从函数参数中获取父节点传递来的数据。

中序位置的代码不仅可以获取参数数据，还可以获取到左子树通过函数返回值传递回来的数据。

后序位置的代码最强，不仅可以获取参数数据，还可以同时获取到左右子树通过函数返回值传递回来的数据。

所以，某些情况下把代码移到后序位置效率最高；有些事情，只有后序位置的代码能做。

举些具体的例子来感受下它们的能力区别。现在给你一棵二叉树，我问你两个简单的问题：

- 1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
- 2、如何打印出每个节点的左右子树各有多少节点？

第一个问题可以这样写代码：

```
// 二叉树遍历函数
void traverse(TreeNode root, int level) {
    if (root == null) {
        return;
    }
    // 前序位置
    printf("Node %s at level %d", root.val, level);
    traverse(root.left, level + 1);
    traverse(root.right, level + 1);
}

// 这样调用
traverse(root, 1);
```

第二个问题可以这样写代码：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置
    printf("节点 %s 的左子树有 %d 个节点，右子树有 %d 个节点",
           root, leftCount, rightCount);

    return leftCount + rightCount + 1;
}
```

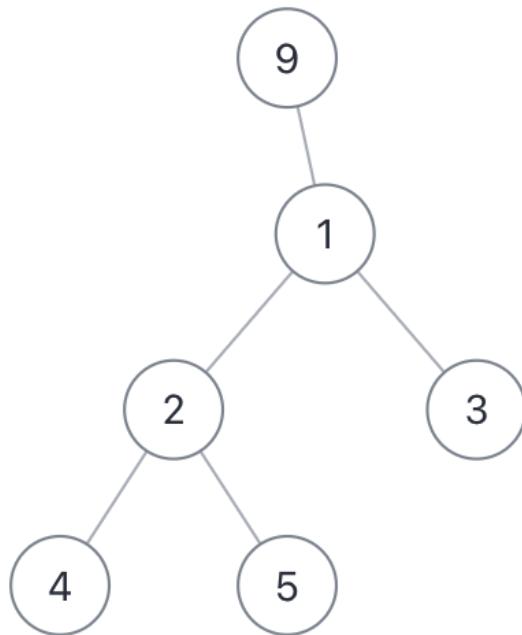
一个节点在第几层，你从根节点遍历过来的过程就能顺带记录，用递归函数的参数就能传递下去；而以一个节点为根的整棵树有多少个节点，你必须遍历完子树之后才能数清楚，然后通过递归函数的返回值拿到答案。

结合这两个简单的问题，你品味一下后序位置的特点，只有后序位置才能通过返回值获取子树的信息。

那么换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了。

接下来看下后序位置是如何在实际的题目中发挥作用的，简单聊下力扣第 543 题「二叉树的直径」，让你计算一棵二叉树的最长直径长度。

所谓二叉树的「直径」长度，就是任意两个结点之间的路径长度。最长「直径」并不一定要穿过根结点，比如下面这棵二叉树：



它的最长直径是 3，即 [4,2,1,3], [4,2,1,9] 或者 [5,2,1,3] 这几条「直径」的长度。

解决这题的关键在于，每一条二叉树的「直径」长度，就是一个节点的左右子树的最大深度之和。

现在让我求整棵树中的最长「直径」，那直截了当的思路就是遍历整棵树中的每个节点，然后通过每个节点的左右子树的最大深度算出每个节点的「直径」，最后把所有「直径」求个最大值即可。

最大深度的算法我们刚才实现过了，上述思路就可以写出以下代码：

```
class Solution {
    // 记录最大直径的长度
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        // 对每个节点计算直径，求最大直径
        traverse(root);
        return maxDiameter;
    }

    // 遍历二叉树
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        // 对每个节点计算直径
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        int myDiameter = leftMax + rightMax;
        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, myDiameter);

        traverse(root.left);
        traverse(root.right);
    }

    // 计算二叉树的最大深度
    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        return 1 + Math.max(leftMax, rightMax);
    }
}
```

这个解法是正确的，但是运行时间很长，原因也很明显，`traverse` 遍历每个节点的时候还会调用递归函数 `maxDepth`，而 `maxDepth` 是要遍历子树的所有节点的，所以最坏时间复杂度是  $O(N^2)$ 。

这就出现了刚才探讨的情况，前序位置无法获取子树信息，所以只能让每个节点调用 `maxDepth` 函数去算子树的深度。

那如何优化？我们应该把计算「直径」的逻辑放在后序位置，准确说应该是放在 `maxDepth` 的后序位置，因为 `maxDepth` 的后序位置是知道左右子树的最大深度的。

所以，稍微改一下代码逻辑即可得到更好的解法：

```
class Solution {
    // 记录最大直径的长度
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }
```

```
}

int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 后序位置，顺便计算最大直径
    int myDiameter = leftMax + rightMax;
    maxDiameter = Math.max(maxDiameter, myDiameter);

    return 1 + Math.max(leftMax, rightMax);
}
}
```

## ▶ 🎃 代码可视化动画🎃

这下时间复杂度只有 `maxDepth` 函数的  $O(N)$  了。

讲到这里，照应一下前文：遇到子树问题，首先想到的是给函数设置返回值，然后在后序位置做文章。

思考题：请你思考一下，运用后序遍历的题目使用的是「遍历」的思路还是「分解问题」的思路？

利用后序位置的题目，一般都使用「分解问题」的思路。因为当前节点接收并利用了子树返回的信息，这就意味着你把原问题分解成了当前节点 + 左右子树的子问题。

反过来，如果你写出了类似一开始的那种递归套递归的解法，大概率也需要反思是不是可以通过后序遍历优化了。

更多利用后序位置的习题参见 [手把手带你刷二叉树（后序篇）](#)、[手把手带你刷二叉搜索树（后序篇）](#) 和 [【强化练习】利用后序位置解题](#)。

## 以树的视角看动归/回溯/DFS算法的区别和联系

前文我说动态规划/回溯算法就是二叉树算法两种不同思路的表现形式，相信能看到这里的读者应该也认可了我这个观点。但有细心的读者经常提问：你的思考方法让我豁然开朗，但你好像一直没讲过 DFS 算法？

其实我在[一文秒杀所有岛屿题目](#)中就是用的 DFS 算法，但我确实没有单独用一篇文章讲 DFS 算法，因为 **DFS 算法和回溯算法非常类似，只是在细节上有所区别**。

这个细节上的差别是什么呢？其实就是「做选择」和「撤销选择」到底在 for 循环外面还是里面的区别，DFS 算法在外面，回溯算法在里面。

为什么有这个区别？还是要结合着二叉树理解。这一部分我就把回溯算法、DFS 算法、动态规划三种经典的算法思想，以及它们和二叉树算法的联系和区别，用一句话来说明：

动归/DFS/回溯算法都可以看做二叉树问题的扩展，只是它们的关注点不同：

- 动态规划算法属于分解问题（分治）的思路，它的关注点在整棵「子树」。
- 回溯算法属于遍历的思路，它的关注点在节点间的「树枝」。
- DFS 算法属于遍历的思路，它的关注点在单个「节点」。

怎么理解？我分别举三个例子你就懂了。

## 例子一：分解问题的思想体现

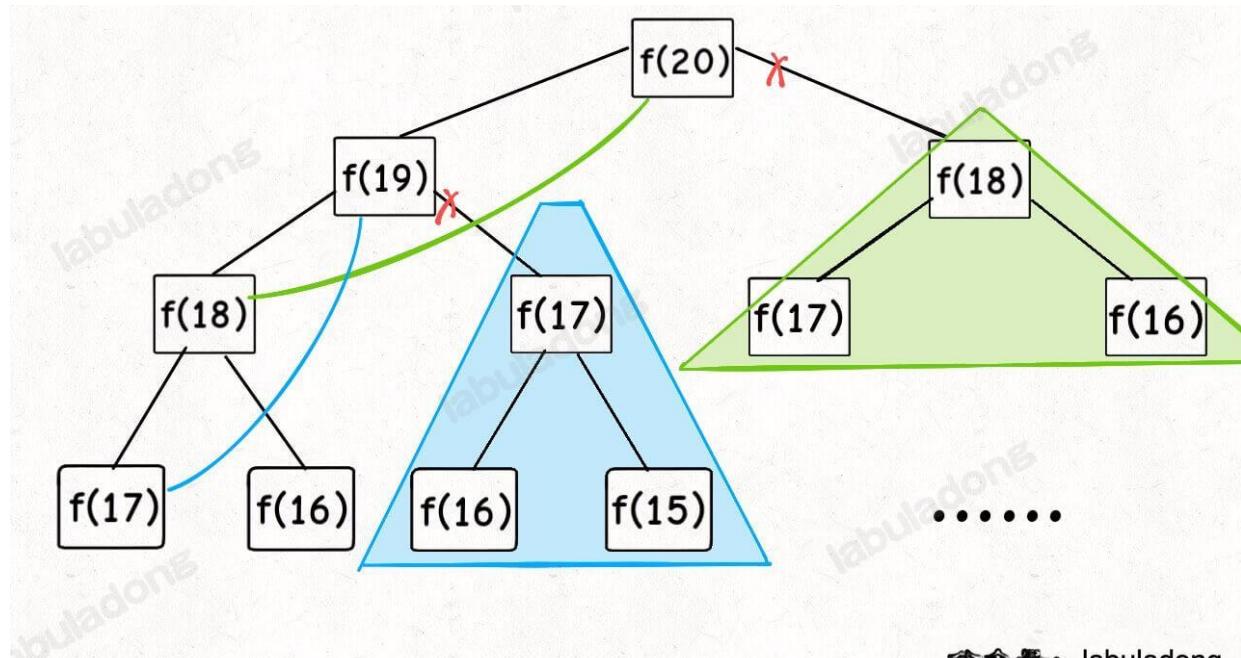
第一个例子，给你一棵二叉树，请你用分解问题的思路写一个 `count` 函数，计算这棵二叉树共有多少个节点。代码很简单，上文都写过了：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 当前节点关心的是两个子树的节点总数分别是多少
    // 因为子问题的结果可以推导出原问题的结果
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置，左右子树节点数加上自己就是整棵树的节点数
    return leftCount + rightCount + 1;
}
```

你看，这就是动态规划分解问题的思路，它的着眼点永远是结构相同的整个子问题，类比到二叉树上就是「子树」。

你再看看具体的动态规划问题，比如 [动态规划框架套路详解](#) 中举的斐波那契的例子，我们的关注点在一棵棵子树的返回值上：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```



## 例子二：回溯算法的思想体现

第二个例子，给你一棵二叉树，请你用遍历的思路写一个 `traverse` 函数，打印出遍历这棵二叉树的过程，你看下代码：

```
void traverse(TreeNode root) {
    if (root == null) return;
    printf("从节点 %s 进入节点 %s", root, root.left);
    traverse(root.left);
    printf("从节点 %s 回到节点 %s", root.left, root);

    printf("从节点 %s 进入节点 %s", root, root.right);
    traverse(root.right);
    printf("从节点 %s 回到节点 %s", root.right, root);
}
```

不难理解吧，好的，我们现在从二叉树进阶成多叉树，代码也是类似的：

```
// 多叉树节点
class Node {
    int val;
    Node[] children;
}

void traverse(Node root) {
    if (root == null) return;
    for (Node child : root.children) {
        printf("从节点 %s 进入节点 %s", root, child);
        traverse(child);
        printf("从节点 %s 回到节点 %s", child, root);
    }
}
```

这个多叉树的遍历框架就可以延伸出 [回溯算法框架套路详解](#) 中的回溯算法框架：

```
// 回溯算法框架
void backtrack(...) {
    // base case
    if (...) return;

    for (int i = 0; i < ...; i++) {
        // 做选择
        ...

        // 进入下一层决策树
        backtrack(...);

        // 撤销刚才做的选择
        ...
    }
}
```

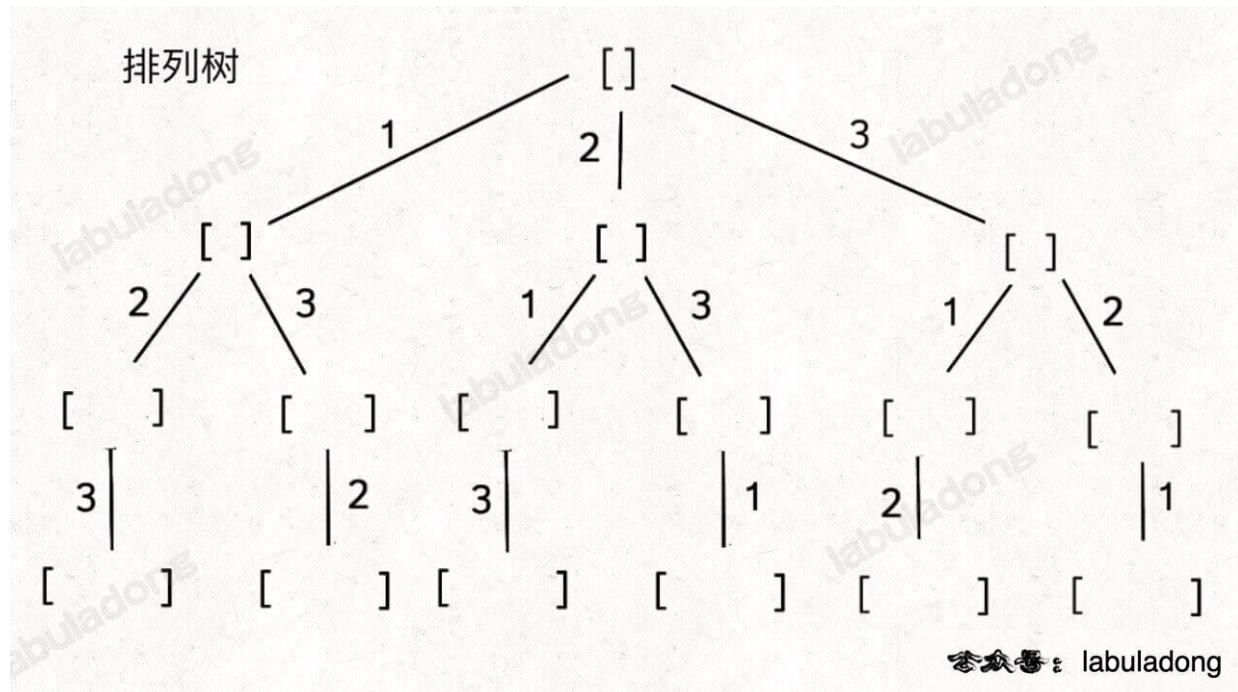
你看，这就是回溯算法遍历的思路，它的着眼点永远是在节点之间移动的过程，类比到二叉树上就是「树枝」。

你再看看具体的回溯算法问题，比如[回溯算法秒杀排列组合子集的九种题型](#)中讲到的全排列，我们的关注点在一条条树枝上：

```
// 回溯算法核心部分代码
void backtrack(int[] nums) {
    // 回溯算法框架
    for (int i = 0; i < nums.length; i++) {
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        // 进入下一层回溯树
        backtrack(nums);

        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}
```



### 例子三：DFS 的思想体现

第三个例子，我给你一棵二叉树，请你写一个 `traverse` 函数，把这棵二叉树上的每个节点的值都加一。很简单吧，代码如下：

```
void traverse(TreeNode root) {
    if (root == null) return;
    // 遍历过的每个节点的值加一
    root.val++;
    traverse(root.left);
    traverse(root.right);
}
```

你看，这就是 DFS 算法遍历的思路，它的着眼点永远是在单一的节点上，类比到二叉树上就是处理每个「节点」。

你再看看具体的 DFS 算法问题，比如 [一文秒杀所有岛屿题目](#) 中讲的前几道题，我们的关注点是 `grid` 数组的每个格子（节点），我们要对遍历过的格子进行一些处理，所以我说是用 DFS 算法解决这几道题的：

```
// DFS 算法核心逻辑
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 0) {
        return;
    }
    // 遍历过的每个格子标记为 0
    grid[i][j] = 0;
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

1	1	0	1	1
1	0	0	0	0
0	0	0	0	1
1	1	0	1	1

好，请你仔细品一下上面三个简单的例子，是不是像我说的：动态规划关注整棵「子树」，回溯算法关注节点间的「树枝」，DFS 算法关注单个「节点」。

有了这些铺垫，你就很容易理解为什么回溯算法和 DFS 算法代码中「做选择」和「撤销选择」的位置不同了，看下面两段代码：

```
// DFS 算法把「做选择」「撤销选择」的逻辑放在 for 循环外面
void dfs(Node root) {
    if (root == null) return;
    // 做选择
    print("我已经进入节点 %s 啦", root);
    for (Node child : root.children) {
        dfs(child);
    }
    // 撤销选择
```

```
    print("我将要离开节点 %s 啦", root);
}

// 回溯算法把「做选择」「撤销选择」的逻辑放在 for 循环里面
void backtrack(Node root) {
    if (root == null) return;
    for (Node child : root.children) {
        // 做选择
        print("我站在节点 %s 到节点 %s 的树枝上", root, child);
        backtrack(child);
        // 撤销选择
        print("我将要离开节点 %s 到节点 %s 的树枝上", child, root);
    }
}
```

看到了吧，你回溯算法必须把「做选择」和「撤销选择」的逻辑放在 for 循环里面，否则怎么拿到「树枝」的两个端点？

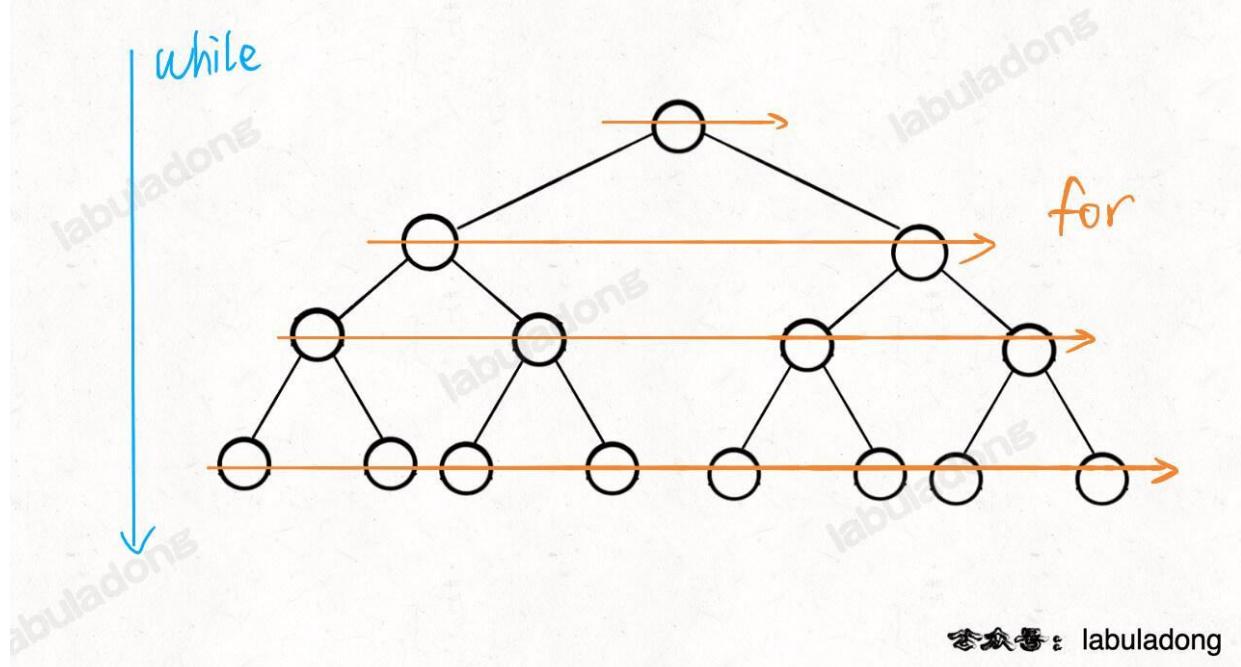
## 层序遍历

二叉树题型主要是用来培养递归思维的，而层序遍历属于迭代遍历，也比较简单，这里就过一下代码框架吧：

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
    }
}
```

这里面 while 循环和 for 循环分管从上到下和从左到右的遍历：



后文 [BFS 算法框架](#) 就是从二叉树的层序遍历扩展出来的，常用于求无权图的最短路径问题。

当然这个框架还可以灵活修改，题目不需要记录层数（步数）时可以去掉上述框架中的 for 循环，比如后文 [Dijkstra 算法](#) 中计算加权图的最短路径问题，详细探讨了 BFS 算法的扩展。

值得一提的是，有些很明显需要用层序遍历技巧的二叉树的题目，也可以用递归遍历的方式去解决，而且技巧性会更强，非常考察你对前中后序的把控。

好了，本文已经够长了，围绕前中后序位置算是把二叉树题目里的各种套路给讲透了，真正能运用出来多少，就需要你亲自刷题实践和思考了。

希望大家能探索尽可能多的解法，只要参透二叉树这种基本数据结构的原理，那么就很容易在学习其他高级算法的道路上找到抓手，打通回路，形成闭环（手动狗头）。

最后，[二叉树递归专项练习](#) 中会手把手带你运用本文所讲的技巧。

## 回答评论区的问题

关于层序遍历（以及其扩展出的 [BFS 算法框架](#)），我在最后多说几句。

如果你对二叉树足够熟悉，可以想到很多方式通过递归函数得到层序遍历结果，比如下面这种写法：

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();

    public List<List<Integer>> levelTraverse(TreeNode root) {
        if (root == null) {
            return res;
        }
        // root 视为第 0 层
        traverse(root, 0);
        return res;
    }

    void traverse(TreeNode root, int depth) {
        if (root == null) {
            return;
        }
        if (res.size() <= depth) {
            res.add(new ArrayList<Integer>());
        }
        res.get(depth).add(root.val);
        traverse(root.left, depth + 1);
        traverse(root.right, depth + 1);
    }
}
```

```
        return;
    }
    // 前序位置，看看是否已经存储 depth 层的节点了
    if (res.size() <= depth) {
        // 第一次进入 depth 层
        res.add(new LinkedList<>());
    }
    // 前序位置，在 depth 层添加 root 节点的值
    res.get(depth).add(root.val);
    traverse(root.left, depth + 1);
    traverse(root.right, depth + 1);
}
}
```

这种思路从结果上说确实可以得到层序遍历结果，但其本质还是二叉树的前序遍历，或者说 DFS 的思路，而不是层序遍历，或者说 BFS 的思路。因为这个解法是依赖前序遍历自顶向下、自左向右的顺序特点得到了正确的结果。

**抽象点说，这个解法更像是从左到右的「列序遍历」，而不是自顶向下的「层序遍历」。**所以对于计算最小距离的场景，这个解法完全等同于 DFS 算法，没有 BFS 算法的性能的优势。

还有优秀读者评论了这样一种递归进行层序遍历的思路：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();

    public List<List<Integer>> levelTraverse(TreeNode root) {
        if (root == null) {
            return res;
        }
        List<TreeNode> nodes = new LinkedList<>();
        nodes.add(root);
        traverse(nodes);
        return res;
    }

    void traverse(List<TreeNode> curLevelNodes) {
        // base case
        if (curLevelNodes.isEmpty()) {
            return;
        }
        // 前序位置，计算当前层的值和下一层的节点列表
        List<Integer> nodeValues = new LinkedList<>();
        List<TreeNode> nextLevelNodes = new LinkedList<>();
        for (TreeNode node : curLevelNodes) {
            nodeValues.add(node.val);
            if (node.left != null) {
                nextLevelNodes.add(node.left);
            }
            if (node.right != null) {
                nextLevelNodes.add(node.right);
            }
        }
        // 前序位置添加结果，可以得到自顶向下的层序遍历
        res.add(nodeValues);
        traverse(nextLevelNodes);
        // 后序位置添加结果，可以得到自底向上的层序遍历结果
    }
}
```

```

        // res.add(nodeValues);
    }
}

```

这个 `traverse` 函数很像递归遍历单链表的函数，其实就是把二叉树的每一层抽象理解成单链表的一个节点进行遍历。

相较上一个递归解法，这个递归解法是自顶向下的「层序遍历」，更接近 BFS 的奥义，可以作为 BFS 算法的递归实现扩展一下思维。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
100. Same Tree	100. 相同的树	
1008. Construct Binary Search Tree from Preorder Traversal	1008. 前序遍历构造二叉搜索树	
101. Symmetric Tree	101. 对称二叉树	
1022. Sum of Root To Leaf Binary Numbers	1022. 从根到叶的二进制数之和	
1026. Maximum Difference Between Node and Ancestor	1026. 节点与其祖先之间的最大差值	
108. Convert Sorted Array to Binary Search Tree	108. 将有序数组转换为二叉搜索树	
1080. Insufficient Nodes in Root to Leaf Paths	1080. 根到叶路径上的不足节点	
110. Balanced Binary Tree	110. 平衡二叉树	
111. Minimum Depth of Binary Tree	111. 二叉树的最小深度	
1110. Delete Nodes And Return Forest	1110. 删点成林	
1120. Maximum Average Subtree	1120. 子树的最大平均值	
113. Path Sum II	113. 路径总和 II	
114. Flatten Binary Tree to Linked List	114. 二叉树展开为链表	
116. Populating Next Right Pointers in Each Node	116. 填充每个节点的下一个右侧节点指针	
124. Binary Tree Maximum Path Sum	124. 二叉树中的最大路径和	
1245. Tree Diameter	1245. 树的直径	
1261. Find Elements in a Contaminated Binary Tree	1261. 在受污染的二叉树中查找元素	
129. Sum Root to Leaf Numbers	129. 求根节点到叶节点数字之和	
1315. Sum of Nodes with Even-Valued Grandparent	1315. 祖父节点值为偶数的节点和	
1325. Delete Leaves With a Given Value	1325. 删除给定值的叶子节点	
1339. Maximum Product of Splitted Binary Tree	1339. 分裂二叉树的最大乘积	
1367. Linked List in Binary Tree	1367. 二叉树中的链表	
1372. Longest ZigZag Path in a Binary Tree	1372. 二叉树中的最长交错路径	

LeetCode	力扣	难度
1373. Maximum Sum BST in Binary Tree	1373. 二叉搜索子树的最大键值和	
1376. Time Needed to Inform All Employees	1376. 通知所有员工所需的时间	
1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree	1379. 找出克隆二叉树中的相同节点	
1430. Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree	1430. 判断给定的序列是否是二叉树从根到叶的路径	
1443. Minimum Time to Collect All Apples in a Tree	1443. 收集树上所有苹果的最少时间	
1448. Count Good Nodes in Binary Tree	1448. 统计二叉树中好节点的数目	
145. Binary Tree Postorder Traversal	145. 二叉树的后序遍历	
1457. Pseudo-Palindromic Paths in a Binary Tree	1457. 二叉树中的伪回文路径	
1469. Find All The Lonely Nodes	1469. 寻找所有的独生节点	
1485. Clone Binary Tree With Random Pointer	1485. 克隆含随机指针的二叉树	
1490. Clone N-ary Tree	1490. 克隆 N 叉树	
1593. Split a String Into the Max Number of Unique Substrings	1593. 拆分字符串使唯一子字符串的数目最大	
1602. Find Nearest Right Node in Binary Tree	1602. 找到二叉树中最近的右侧节点	
1612. Check If Two Expression Trees are Equivalent	1612. 检查两棵二叉表达式树是否等价	
1740. Find Distance in a Binary Tree	1740. 找到二叉树中的距离	
2049. Count Nodes With the Highest Score	2049. 统计最高分的节点数目	
2096. Step-By-Step Directions From a Binary Tree Node to Another	2096. 从二叉树一个节点到另一个节点每一步的方向	
226. Invert Binary Tree	226. 翻转二叉树	
250. Count Univalue Subtrees	250. 统计同值子树	
254. Factor Combinations	254. 因子的组合	
257. Binary Tree Paths	257. 二叉树的所有路径	
267. Palindrome Permutation II	267. 回文排列 II	
270. Closest Binary Search Tree Value	270. 最接近的二叉搜索树值	
294. Flip Game II	294. 翻转游戏 II	
298. Binary Tree Longest Consecutive Sequence	298. 二叉树最长连续序列	
332. Reconstruct Itinerary	332. 重新安排行程	
333. Largest BST Subtree	333. 最大 BST 子树	
339. Nested List Weight Sum	339. 嵌套列表权重和	
366. Find Leaves of Binary Tree	366. 寻找二叉树的叶子节点	

LeetCode	力扣	难度
386. Lexicographical Numbers	386. 字典序排数	🟠
404. Sum of Left Leaves	404. 左叶子之和	🟢
426. Convert Binary Search Tree to Sorted Doubly Linked List 🔒	426. 将二叉搜索树转化为排序的双向链表 🔒	🟠
437. Path Sum III	437. 路径总和 III	🟠
501. Find Mode in Binary Search Tree	501. 二叉搜索树中的众数	🟢
508. Most Frequent Subtree Sum	508. 出现次数最多的子树元素和	🟠
513. Find Bottom Left Tree Value	513. 找树左下角的值	🟠
515. Find Largest Value in Each Tree Row	515. 在每个树行中找最大值	🟠
530. Minimum Absolute Difference in BST	530. 二叉搜索树的最小绝对差	🟢
538. Convert BST to Greater Tree	538. 把二叉搜索树转换为累加树	🟠
549. Binary Tree Longest Consecutive Sequence II 🔒	549. 二叉树中最长的连续序列 🔒	🟠
559. Maximum Depth of N-ary Tree	559. N 叉树的最大深度	🟢
563. Binary Tree Tilt	563. 二叉树的坡度	🟢
572. Subtree of Another Tree	572. 另一棵树的子树	🟢
582. Kill Process 🔒	582. 杀掉进程 🔒	🟠
606. Construct String from Binary Tree	606. 根据二叉树创建字符串	🟢
617. Merge Two Binary Trees	617. 合并二叉树	🟢
623. Add One Row to Tree	623. 在二叉树中增加一行	🟠
654. Maximum Binary Tree	654. 最大二叉树	🟠
663. Equal Tree Partition 🔒	663. 均匀树划分 🔒	🟠
666. Path Sum IV 🔒	666. 路径总和 IV 🔒	🟠
669. Trim a Binary Search Tree	669. 修剪二叉搜索树	🟠
671. Second Minimum Node In a Binary Tree	671. 二叉树中第二小的节点	🟢
687. Longest Univalue Path	687. 最长同值路径	🟠
776. Split BST 🔒	776. 拆分二叉搜索树 🔒	🟠
865. Smallest Subtree with all the Deepest Nodes	865. 具有所有最深节点的最小子树	🟠
894. All Possible Full Binary Trees	894. 所有可能的真二叉树	🟠
897. Increasing Order Search Tree	897. 递增顺序搜索树	🟢
938. Range Sum of BST	938. 二叉搜索树的范围和	🟢
951. Flip Equivalent Binary Trees	951. 翻转等价二叉树	🟠
965. Univalued Binary Tree	965. 单值二叉树	🟢

LeetCode	力扣	难度
968. Binary Tree Cameras	968. 监控二叉树	🔴
971. Flip Binary Tree To Match Preorder Traversal	971. 翻转二叉树以匹配先序遍历	🟡
979. Distribute Coins in Binary Tree	979. 在二叉树中分配硬币	🟡
987. Vertical Order Traversal of a Binary Tree	987. 二叉树的垂序遍历	🔴
988. Smallest String Starting From Leaf	988. 从叶结点开始的最小字符串	🟡
99. Recover Binary Search Tree	99. 恢复二叉搜索树	🟡
993. Cousins in Binary Tree	993. 二叉树的堂兄弟节点	🟢
998. Maximum Binary Tree II	998. 最大二叉树 II	🟡
-	剑指 Offer 06. 从尾到头打印链表	🟢
-	剑指 Offer 26. 树的子结构	🟡
-	剑指 Offer 27. 二叉树的镜像	🟢
-	剑指 Offer 28. 对称的二叉树	🟢
-	剑指 Offer 33. 二叉搜索树的后序遍历序列	🟡
-	剑指 Offer 34. 二叉树中和为某一值的路径	🟡
-	剑指 Offer 36. 二叉搜索树与双向链表	🟡
-	剑指 Offer 55 - I. 二叉树的深度	🟢
-	剑指 Offer 55 - II. 平衡二叉树	🟢
-	剑指 Offer II 044. 二叉树每层的最大值	🟡
-	剑指 Offer II 045. 二叉树最底层最左边的值	🟡
-	剑指 Offer II 049. 从根节点到叶节点的路径数字之和	🟡
-	剑指 Offer II 050. 向下的路径节点之和	🟡
-	剑指 Offer II 051. 节点之和最大的路径	🔴
-	剑指 Offer II 052. 展平二叉搜索树	🟢
-	剑指 Offer II 054. 所有大于等于节点的值之和	🟡

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 回溯算法秒杀所有排列/组合/子集问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">77. Combinations</a>	77. 组合	简单
<a href="#">90. Subsets II</a>	90. 子集 II	简单
-	剑指 Offer II 082. 含有重复元素集合的组合	简单
<a href="#">216. Combination Sum III</a>	216. 组合总和 III	中等
<a href="#">40. Combination Sum II</a>	40. 组合总和 II	中等
<a href="#">78. Subsets</a>	78. 子集	简单
<a href="#">47. Permutations II</a>	47. 全排列 II	中等
<a href="#">39. Combination Sum</a>	39. 组合总和	中等
<a href="#">46. Permutations</a>	46. 全排列	中等
-	剑指 Offer II 084. 含有重复元素集合的全排列	中等

阅读本文前，你需要先学习：

- [二叉树系列算法（纲领篇）](#)
- [回溯算法核心框架](#)

tip：本文有视频版：[回溯算法秒杀所有排列/组合/子集问题](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

虽然排列、组合、子集系列问题是高中就学过的，但如果想编写算法解决它们，还是非常考验计算机思维的，本文就讲讲编程解决这几个问题的核心思路，以后再有什么变体，你也能手到擒来，以不变应万变。

无论是排列、组合还是子集问题，简单说无非就是让你从序列 `nums` 中以给定规则取若干元素，主要有以下几种变体：

**形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，这也是最基本的形式。**

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该只有 `[7]`。

**形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次。**

以组合为例，如果输入 `nums = [2,5,2,1,2]`，和为 7 的组合应该有两种 `[2,2,2,1]` 和 `[5,2]`。

**形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次。**

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该有两种 `[2,2,3]` 和 `[7]`。

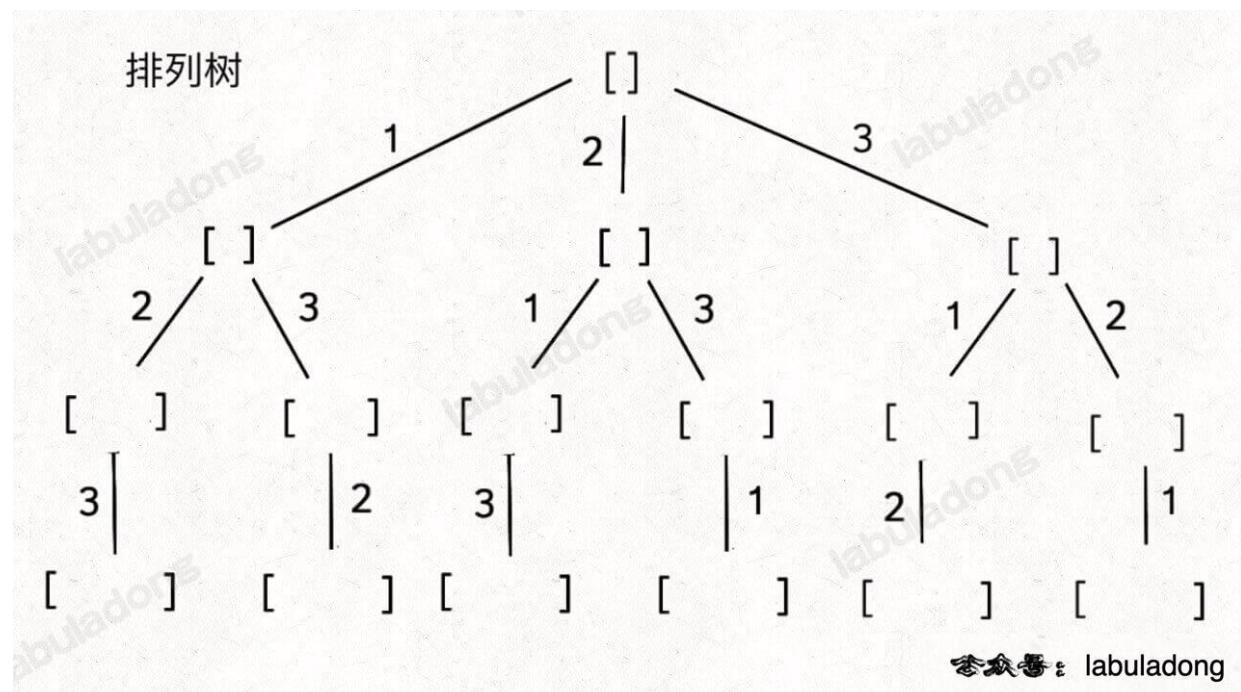
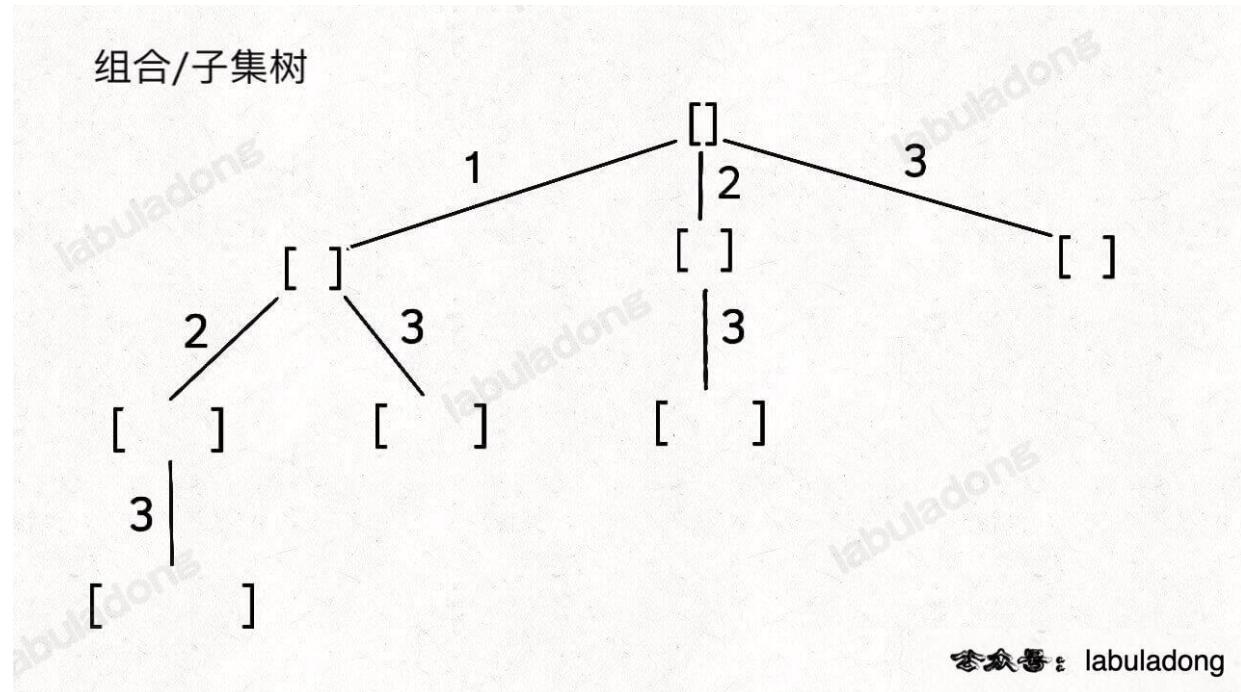
当然，也可以说有第四种形式，即元素可重可复选。但既然元素可复选，那又何必存在重复元素呢？元素去重之后就等同于形式三，所以这种情况不用考虑。

上面用组合问题举的例子，但排列、组合、子集问题都可以有这三种基本形式，所以共有 9 种变化。

除此之外，题目也可以再添加各种限制条件，比如让你求和为 `target` 且元素个数为 `k` 的组合，那这么一来又可以衍生出一堆变体，怪不得面试笔试中经常考到排列组合这种基本题型。

但无论形式怎么变化，其本质就是穷举所有解，而这些解呈现树形结构，所以合理使用回溯算法框架，稍改代码框架即可把这些问题一网打尽。

具体来说，你需要先阅读并理解前文 [回溯算法核心套路](#)，然后记住如下子集问题和排列问题的回溯树，就可以解决所有排列组合子集相关的问题：



为什么只要记住这两种树形结构就能解决所有相关问题呢？

首先，组合问题和子集问题其实是等价的，这个后面会讲；至于之前说的三种变化形式，无非是在这两棵树上剪掉或者增加一些树枝罢了。

那么，接下来我们就开始穷举，把排列/组合/子集问题的 9 种形式都过一遍，学学如何用回溯算法把它们一套带走。

另外，有些读者之前看过的排列/子集/组合的解法代码可能和我在本文介绍的代码不同。这是因为回溯算法有两种穷举视角，我会在后文 [球盒模型：回溯算法穷举的两种视角](#) 手把手给你讲清楚。现在还不适合直接跟你讲那些解法，你照着我的思路学习即可。

## 子集（元素无重不可复选）

力扣第 78 题「子集」就是这个问题：

题目给你输入一个无重复元素的数组 `nums`，其中每个元素最多使用一次，请你返回 `nums` 的所有子集。

函数签名如下：

```
List<List<Integer>> subsets(int[] nums)
```

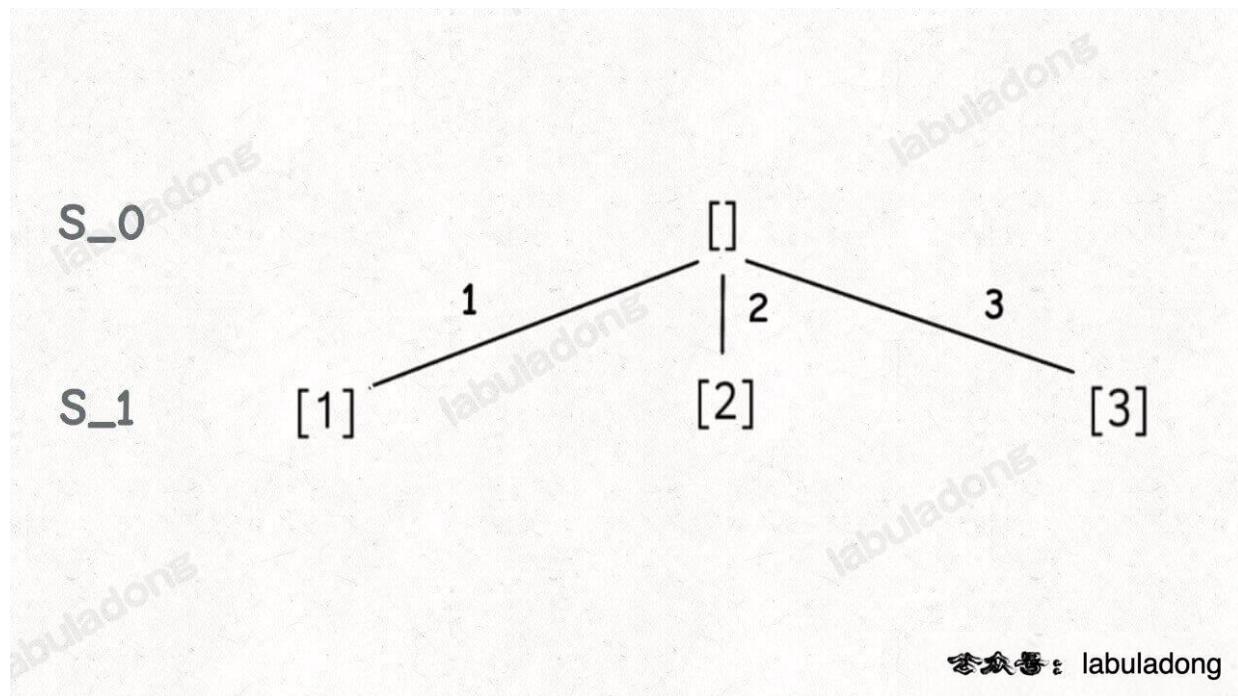
比如输入 `nums = [1,2,3]`，算法应该返回如下子集：

```
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

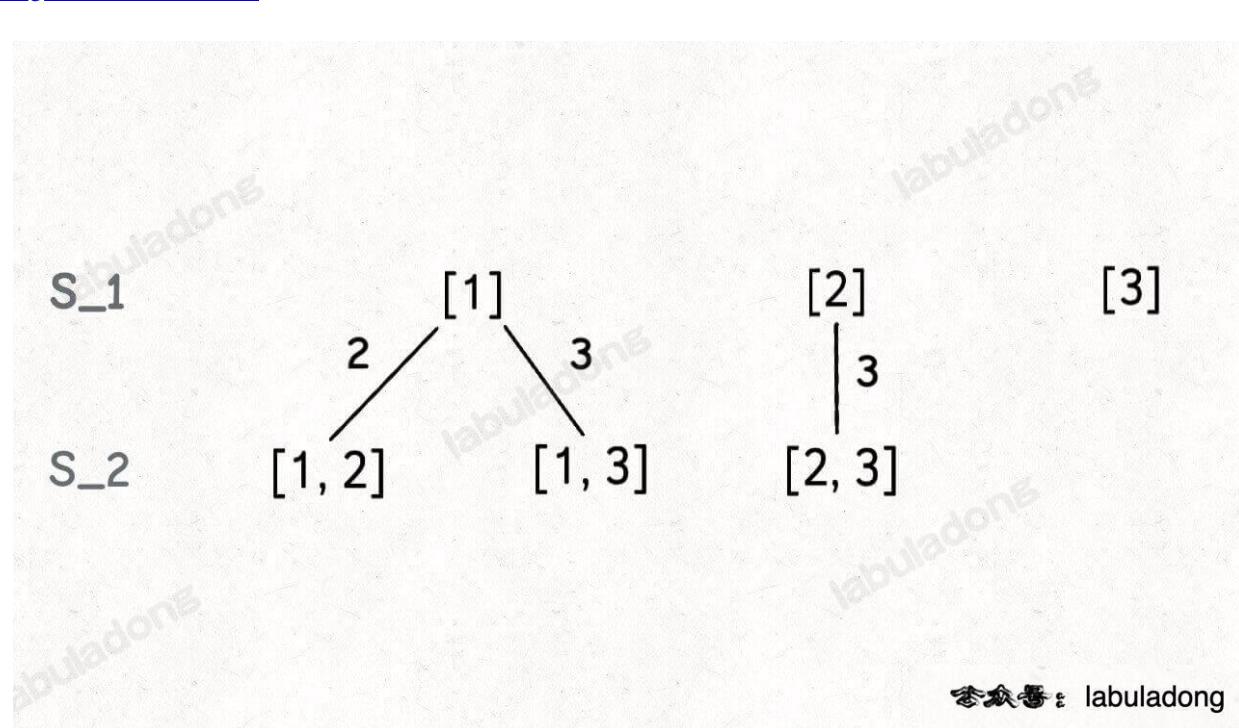
好，我们暂时不考虑如何用代码实现，先回忆一下我们的高中知识，如何手推所有子集？

首先，生成元素个数为 0 的子集，即空集 `[]`，为了方便表示，我称之为 `S_0`。

然后，在 `S_0` 的基础上生成元素个数为 1 的所有子集，我称为 `S_1`：



接下来，我们可以在 `S_1` 的基础上推导出 `S_2`，即元素个数为 2 的所有子集：



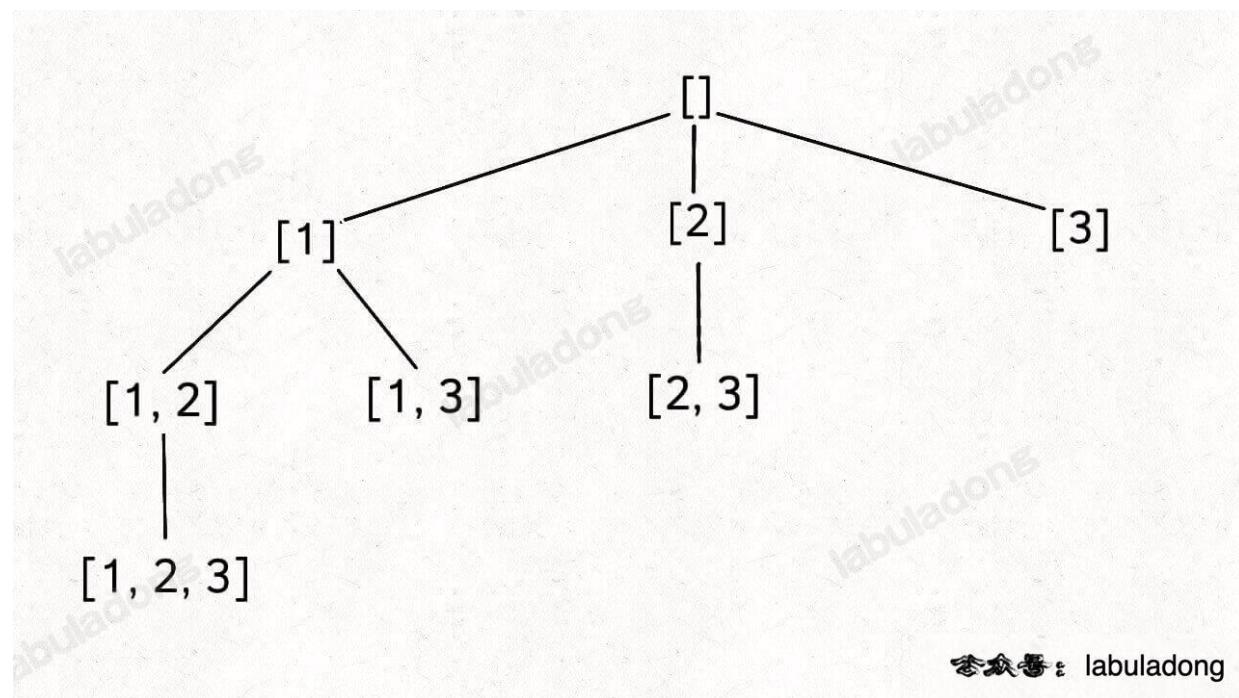
为什么集合 **[2]** 只需要添加 **3**，而不添加前面的 **1** 呢？

因为集合中的元素不用考虑顺序，**[1, 2, 3]** 中 **2** 后面只有 **3**，如果你添加了前面的 **1**，那么 **[2, 1]** 会和之前已经生成的子集 **[1, 2]** 重复。

换句话说，我们通过保证元素之间的相对顺序不变来防止出现重复的子集。

接着，我们可以通过 **S\_2** 推出 **S\_3**，实际上 **S\_3** 中只有一个集合 **[1, 2, 3]**，它是通过 **[1, 2]** 推出的。

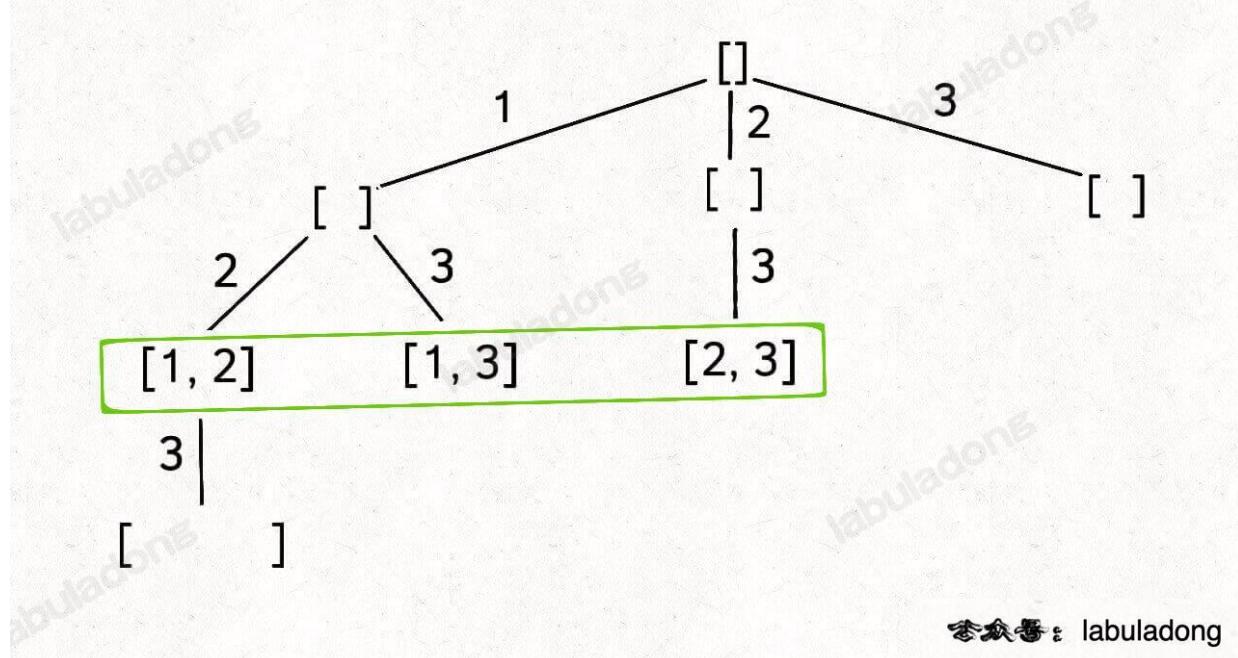
整个推导过程就是这样一棵树：



注意这棵树的特性：

如果把根节点作为第 **0** 层，将每个节点和根节点之间树枝上的元素作为该节点的值，那么第 **n** 层的所有节点就是大小为 **n** 的所有子集。

你比如大小为 **2** 的子集就是这一层节点的值：



© labuladong

注意，本文之后所说「节点的值」都是指节点和根节点之间树枝上的元素，且将根节点认为是第 0 层。

那么再进一步，如果想计算所有子集，那只要遍历这棵多叉树，把所有节点的值收集起来不就行了？

直接看代码：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();

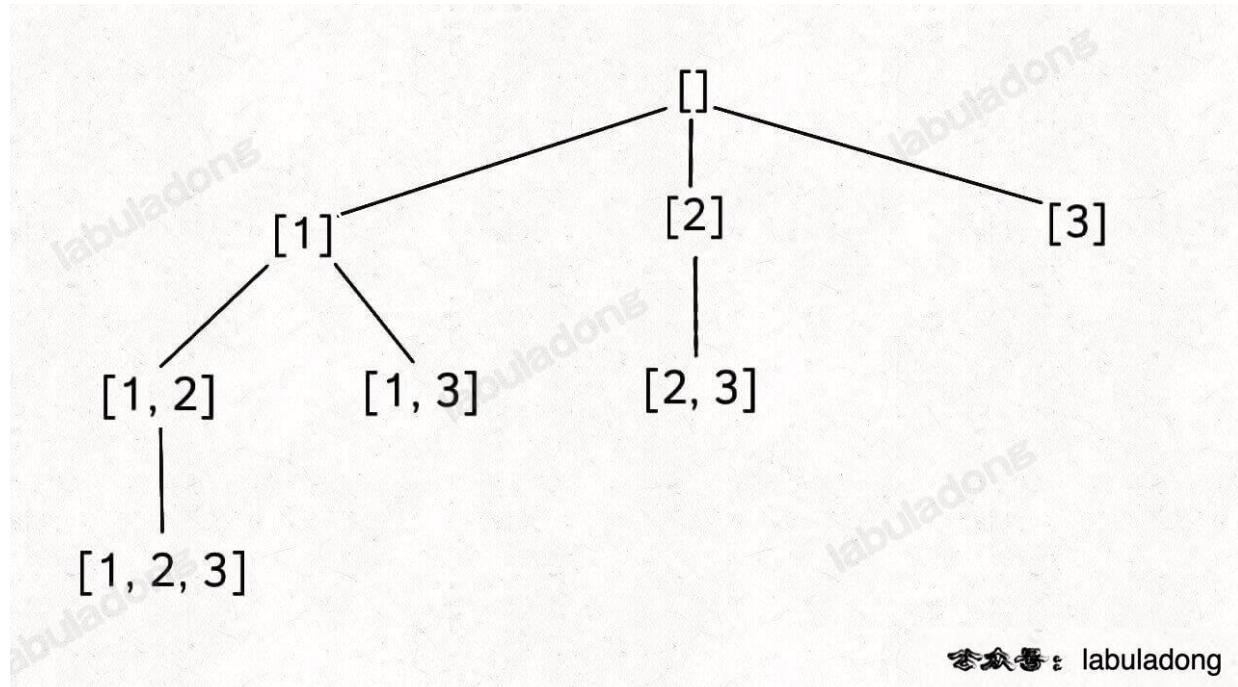
    // 主函数
    public List<List<Integer>> subsets(int[] nums) {
        backtrack(nums, 0);
        return res;
    }

    // 回溯算法核心函数，遍历子集问题的回溯树
    void backtrack(int[] nums, int start) {

        // 前序位置，每个节点的值都是一个子集
        res.add(new LinkedList<>(track));

        // 回溯算法标准框架
        for (int i = start; i < nums.length; i++) {
            // 做选择
            track.addLast(nums[i]);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(nums, i + 1);
            // 撤销选择
            track.removeLast();
        }
    }
}
```

看过前文 [回溯算法核心框架](#) 的读者应该很容易理解这段代码吧，我们使用 `start` 参数控制树枝的生长避免产生重复的子集，用 `track` 记录根节点到每个节点的路径的值，同时在前序位置把每个节点的路径值收集起来，完成回溯树的遍历就收集了所有子集：



© labuladong

最后，`backtrack` 函数开头看似没有 base case，会不会进入无限递归？

其实不会的，当 `start == nums.length` 时，叶子节点的值会被装入 `res`，但 for 循环不会执行，也就结束了递归。

---

▶ 🔍 代码可视化动画 🔍

---

## 组合（元素无重不可复选）

如果你能够成功的生成所有无重子集，那么你稍微改改代码就能生成所有无重组合了。

你比如说，让你在 `nums = [1, 2, 3]` 中拿 2 个元素形成所有的组合，你怎么做？

稍微想想就会发现，大小为 2 的所有组合，不就是所有大小为 2 的子集嘛。

所以我说组合和子集是一样的：大小为 `k` 的组合就是大小为 `k` 的子集。

比如力扣第 77 题「组合」：

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

函数签名如下：

```
List<List<Integer>> combine(int n, int k)
```

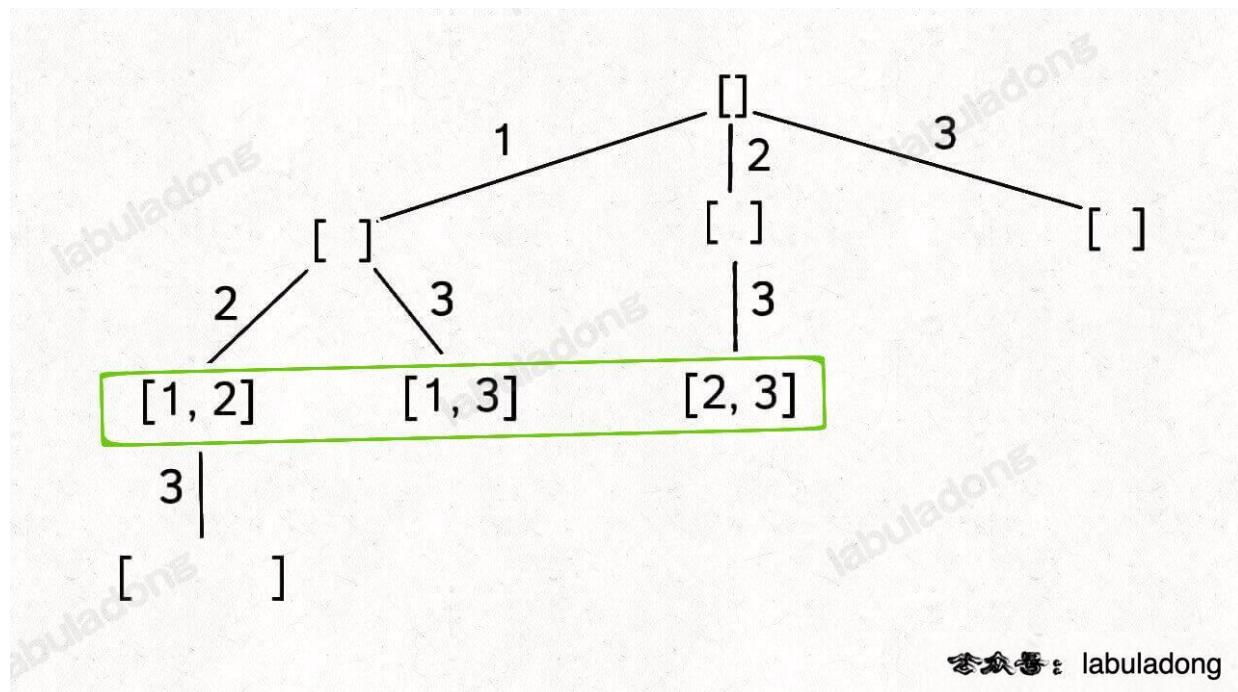
比如 `combine(3, 2)` 的返回值应该是：

```
[ [1, 2], [1, 3], [2, 3] ]
```

这是标准的组合问题，但我给你翻译一下就变成子集问题了：

给你输入一个数组 `nums = [1, 2, ..., n]` 和一个正整数 `k`，请你生成所有大小为 `k` 的子集。

还是以 `nums = [1, 2, 3]` 为例，刚才让你求所有子集，就是把所有节点的值都收集起来；现在你只需要把第 2 层（根节点视为第 0 层）的节点收集起来，就是大小为 2 的所有组合：



反映到代码上，只需要稍改 base case，控制算法仅仅收集第 `k` 层节点的值即可：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();

    // 主函数
    public List<List<Integer>> combine(int n, int k) {
        backtrack(1, n, k);
        return res;
    }

    void backtrack(int start, int n, int k) {
        // base case
        if (k == track.size()) {
            // 遍历到了第 k 层，收集当前节点的值
            res.add(new LinkedList<>(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = start; i <= n; i++) {
            // 选择
            track.addLast(i);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(i + 1, n, k);
            // 撤销选择
            track.removeLast();
        }
    }
}
```

```
        }
    }
}
```

这样，标准的组合问题也解决了。

---

▶ 🎃 代码可视化动画🎃

---

## 排列（元素无重不可复选）

排列问题在前文 [回溯算法核心框架](#) 讲过，这里就简单过一下。

力扣第 46 题「全排列」就是标准的排列问题：

给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。

函数签名如下：

```
List<List<Integer>> permute(int[] nums)
```

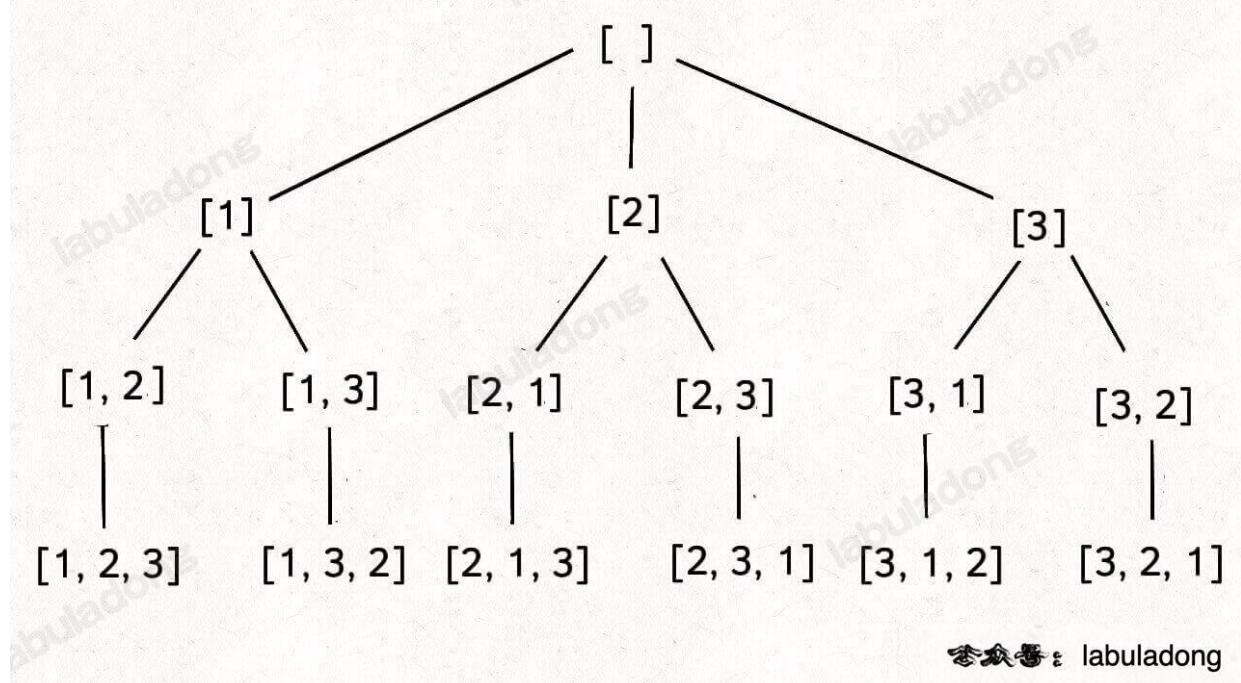
比如输入 `nums = [1,2,3]`，函数的返回值应该是：

```
[
  [1,2,3], [1,3,2],
  [2,1,3], [2,3,1],
  [3,1,2], [3,2,1]
]
```

刚才讲的组合/子集问题使用 `start` 变量保证元素 `nums[start]` 之后只会出现 `nums[start+1..]` 中的元素，通过固定元素的相对位置保证不出现重复的子集。

但排列问题本身就是让你穷举元素的位置，`nums[i]` 之后也可以出现 `nums[i]` 左边的元素，所以之前的那一套玩不转了，需要额外使用 `used` 数组来标记哪些元素还可以被选择。

标准全排列可以抽象成如下这棵多叉树：



✿✿✿✿ labuladong

我们用 `used` 数组标记已经在路径上的元素避免重复选择，然后收集所有叶子节点上的值，就是所有全排列的结果：

```

class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();
    // track 中的元素会被标记为 true
    boolean[] used;

    // 主函数，输入一组不重复的数字，返回它们的全排列
    public List<List<Integer>> permute(int[] nums) {
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }

    // 回溯算法核心函数
    void backtrack(int[] nums) {
        // base case, 到达叶子节点
        if (track.size() == nums.length) {
            // 收集叶子节点上的值
            res.add(new LinkedList(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = 0; i < nums.length; i++) {
            // 已经存在 track 中的元素，不能重复选择
            if (used[i]) {
                continue;
            }
            // 做选择
            used[i] = true;
            track.addLast(nums[i]);
            // 进入下一层回溯树
            backtrack(nums);
        }
    }
}

```

```
// 取消选择
track.removeLast();
used[i] = false;
}
}
}
```

## ▶ 代码可视化动画

这样，全排列问题就解决了。

但如果题目不让你算全排列，而是让你算元素个数为  $k$  的排列，怎么算？

也很简单，改下 `backtrack` 函数的 base case，仅收集第  $k$  层的节点值即可：

```
// 回溯算法核心函数
void backtrack(int[] nums, int k) {
    // base case, 到达第 k 层, 收集节点的值
    if (track.size() == k) {
        // 第 k 层节点的值就是大小为 k 的排列
        res.add(new LinkedList(track));
        return;
    }

    // 回溯算法标准框架
    for (int i = 0; i < nums.length; i++) {
        // ...
        backtrack(nums, k);
        // ...
    }
}
```

## 子集/组合（元素可重不可复选）

刚才讲的标准子集问题输入的 `nums` 是没有重复元素的，但如果存在重复元素，怎么处理呢？

力扣第 90 题「子集 II」就是这样一个问题：

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集。

函数签名如下：

```
List<List<Integer>> subsetsWithDup(int[] nums)
```

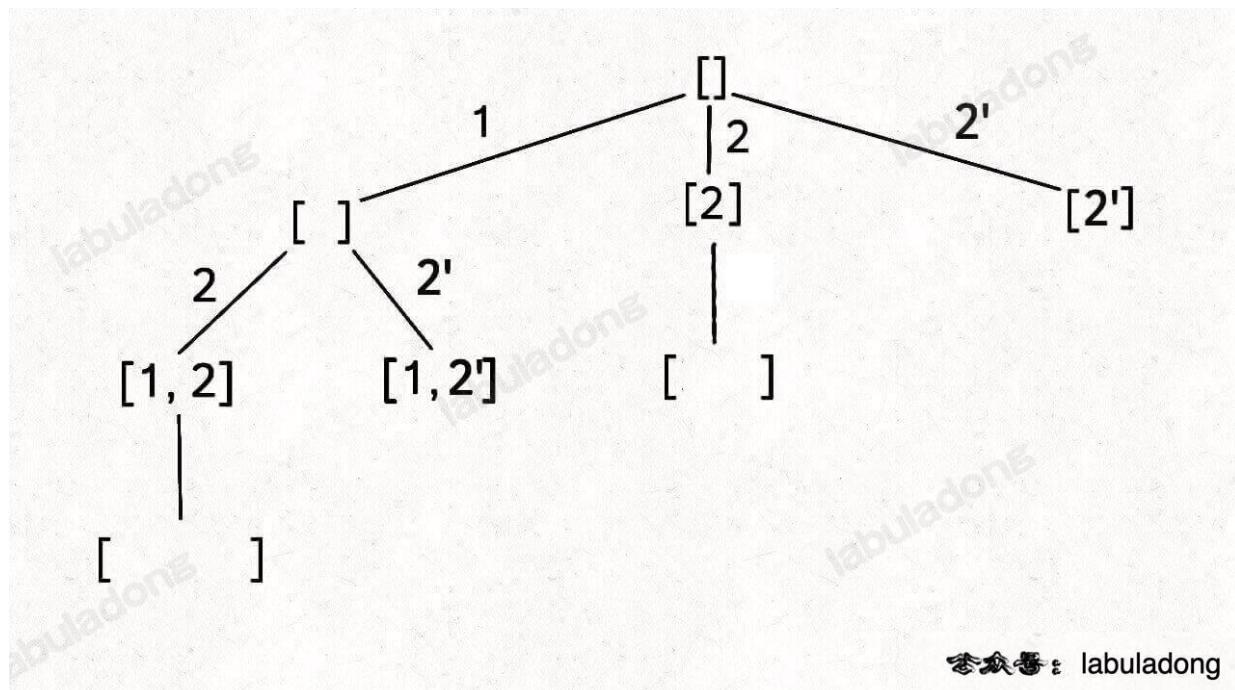
比如输入 `nums = [1,2,2]`，你应该输出：

```
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]
```

当然，按道理说「集合」不应该包含重复元素的，但既然题目这样问了，我们就忽略这个细节吧，仔细思考一下这道题怎么做才是正事。

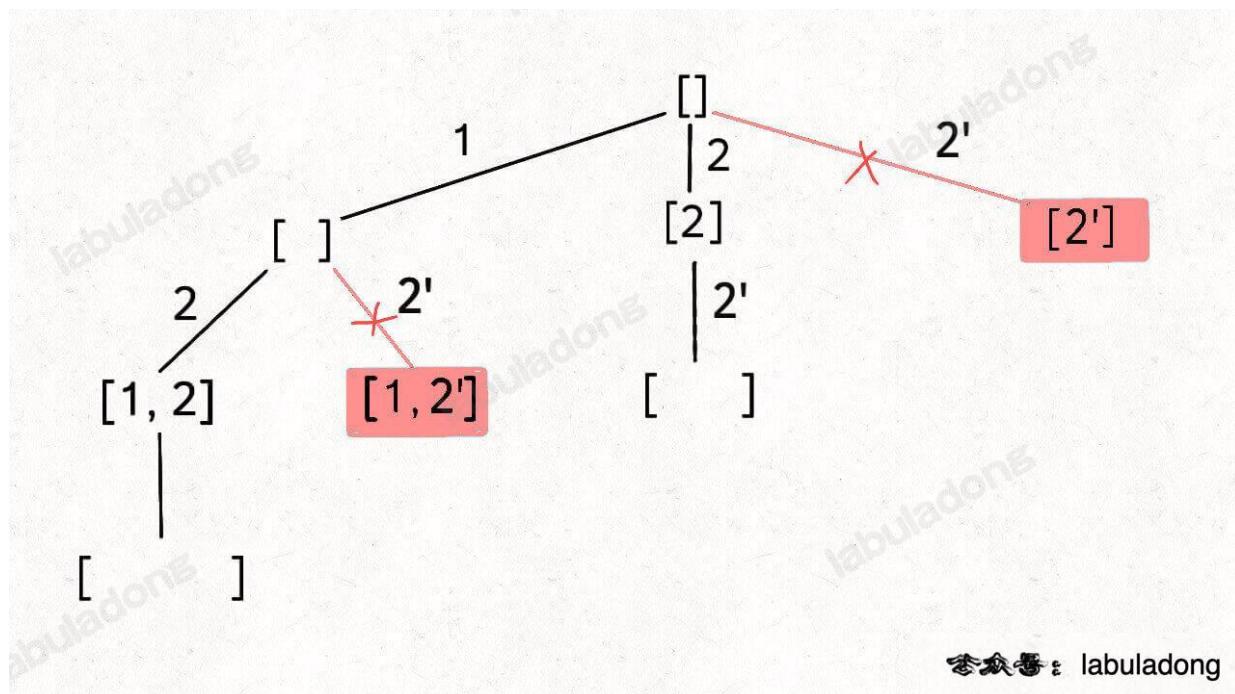
就以 `nums = [1, 2, 2]` 为例，为了区别两个 2 是不同元素，后面我们写作 `nums = [1, 2, 2']`。

按照之前的思路画出子集的树形结构，显然，两条值相同的相邻树枝会产生重复：



```
[  
  [],  
  [1], [2], [2'],  
  [1, 2], [1, 2'], [2, 2'],  
  [1, 2, 2']  
]
```

你可以看到，`[2]` 和 `[1, 2]` 这两个结果出现了重复，所以我们需要进行剪枝，如果一个节点有多条值相同的树枝相邻，则只遍历第一条，剩下的都剪掉，不要去遍历：



体现在代码上，需要先进行排序，让相同的元素靠在一起，如果发现 `nums[i] == nums[i-1]`，则跳过：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> track = new LinkedList<>();

    public List<List<Integer>> subsetsWithDup(int[] nums) {
        // 先排序，让相同的元素靠在一起
        Arrays.sort(nums);
        backtrack(nums, 0);
        return res;
    }

    void backtrack(int[] nums, int start) {
        // 前序位置，每个节点的值都是一个子集
        res.add(new LinkedList<>(track));

        for (int i = start; i < nums.length; i++) {
            // 剪枝逻辑，值相同的相邻树枝，只遍历第一条
            if (i > start && nums[i] == nums[i - 1]) {
                continue;
            }
            track.addLast(nums[i]);
            backtrack(nums, i + 1);
            track.removeLast();
        }
    }
}
```

## ▶ 彩虹 代码可视化动画

这段代码和之前标准的子集问题的代码几乎相同，就是添加了排序和剪枝的逻辑。

至于为什么要这样剪枝，结合前面的图应该也很容易理解，这样带重复元素的子集问题也解决了。

我们说了组合问题和子集问题是等价的，所以我们直接看一道组合的题目吧，这是力扣第 40 题「组合总和 II」：

给你输入 `candidates` 和一个目标和 `target`，从 `candidates` 中找出中所有和为 `target` 的组合。

`candidates` 可能存在重复元素，且其中的每个数字最多只能使用一次。

说这是一个组合问题，其实换个问法就变成子集问题了：请你计算 `candidates` 中所有和为 `target` 的子集。

所以这题怎么做呢？

对比子集问题的解法，只要额外用一个 `trackSum` 变量记录回溯路径上的元素和，然后将 base case 改一改即可解决这道题：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();
    // 记录 track 中的元素之和
    int trackSum = 0;
```

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {  
    if (candidates.length == 0) {  
        return res;  
    }  
    // 先排序，让相同的元素靠在一起  
    Arrays.sort(candidates);  
    backtrack(candidates, 0, target);  
    return res;  
}  
  
// 回溯算法主函数  
void backtrack(int[] nums, int start, int target) {  
    // base case, 达到目标和，找到符合条件的组合  
    if (trackSum == target) {  
        res.add(new LinkedList<>(track));  
        return;  
    }  
    // base case, 超过目标和，直接结束  
    if (trackSum > target) {  
        return;  
    }  
  
    // 回溯算法标准框架  
    for (int i = start; i < nums.length; i++) {  
        // 剪枝逻辑，值相同的树枝，只遍历第一条  
        if (i > start && nums[i] == nums[i - 1]) {  
            continue;  
        }  
        // 做选择  
        track.add(nums[i]);  
        trackSum += nums[i];  
        // 递归遍历下一层回溯树  
        backtrack(nums, i + 1, target);  
        // 撤销选择  
        track.removeLast();  
        trackSum -= nums[i];  
    }  
}
```

## ▶ ⭐ 代码可视化动画⭐

## 排列（元素可重不可复选）

排列问题的输入如果存在重复，比子集/组合问题稍微复杂一点，我们看看力扣第 47 题「全排列 II」：

给你输入一个可包含重复数字的序列 `nums`，请你写一个算法，返回所有可能的全排列，函数签名如下：

```
List<List<Integer>> permuteUnique(int[] nums)
```

比如输入 `nums = [1,2,2]`，函数返回：

```
[ [1,2,2],[2,1,2],[2,2,1] ]
```

先看解法代码：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> track = new LinkedList<>();
    boolean[] used;

    public List<List<Integer>> permuteUnique(int[] nums) {
        // 先排序，让相同的元素靠在一起
        Arrays.sort(nums);
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }

    void backtrack(int[] nums) {
        if (track.size() == nums.length) {
            res.add(new LinkedList(track));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (used[i]) {
                continue;
            }
            // 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
            if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
                continue;
            }
            track.add(nums[i]);
            used[i] = true;
            backtrack(nums);
            track.removeLast();
            used[i] = false;
        }
    }
}
```

### ▶ 代码可视化动画

你对比一下之前的标准全排列解法代码，这段解法代码只有两处不同：

- 1、对 `nums` 进行了排序。
- 2、添加了一句额外的剪枝逻辑。

类比输入包含重复元素的子集/组合问题，你大概应该理解这么做是为了防止出现重复结果。

但是注意排列问题的剪枝逻辑，和子集/组合问题的剪枝逻辑略有不同：新增了 `!used[i - 1]` 的逻辑判断。

这个地方理解起来就需要一些技巧性了，且听我慢慢到来。为了方便研究，依然把相同的元素用上标<sup>'</sup>以示区别。

假设输入为 `nums = [1,2,2']`，标准的全排列算法会得出如下答案：

```
[  
    [1,2,2'], [1,2',2],  
    [2,1,2'], [2,2',1],  
    [2',1,2], [2',2,1]  
]
```

显然，这个结果存在重复，比如 `[1,2,2']` 和 `[1,2',2]` 应该只被算作同一个排列，但被算作了两个不同的排列。

所以现在关键在于，如何设计剪枝逻辑，把这种重复去除掉？

答案是，保证相同元素在排列中的相对位置保持不变。

比如说 `nums = [1,2,2']` 这个例子，我保持排列中 `2` 一直在 `2'` 前面。

这样的话，你从上面 6 个排列中只能挑出 3 个排列符合这个条件：

```
[ [1,2,2'], [2,1,2'], [2,2',1] ]
```

这也就是正确答案。

进一步，如果 `nums = [1,2,2',2'']`，我只要保证重复元素 `2` 的相对位置固定，比如说 `2 -> 2' -> 2''`，也可以得到无重复的全排列结果。

仔细思考，应该很容易明白其中的原理：

标准全排列算法之所以出现重复，是因为把相同元素形成的排列序列视为不同的序列，但实际上它们应该是相同的；而如果固定相同元素形成的序列顺序，当然就避免了重复。

那么反映到代码上，你注意看这个剪枝逻辑：

```
// 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置  
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {  
    // 如果前面的相邻相等元素没有用过，则跳过  
    continue;  
}  
// 选择 nums[i]
```

当出现重复元素时，比如输入 `nums = [1,2,2',2'']`，`2'` 只有在 `2` 已经被使用的情况下才会被选择，同理，`2''` 只有在 `2'` 已经被使用的情况下才会被选择，这就保证了相同元素在排列中的相对位置保证固定。

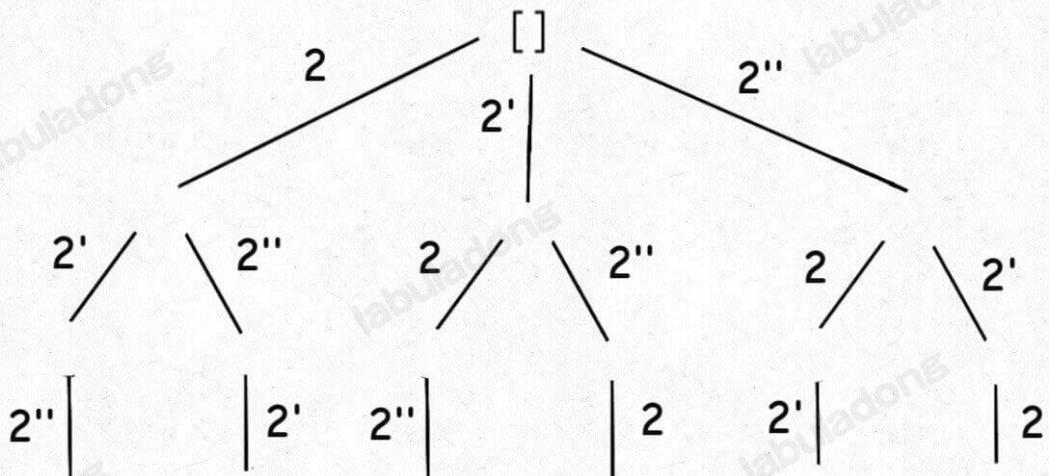
这里拓展一下，如果你把上述剪枝逻辑中的 `!used[i - 1]` 改成 `used[i - 1]`，其实也可以通过所有测试用例，但效率会有所下降，这是为什么呢？

之所以这样修改不会产生错误，是因为这种写法相当于维护了 `2'' -> 2' -> 2` 的相对顺序，最终也可以实现去重的效果。

但为什么这样写效率会下降呢？因为这个写法剪掉的树枝不够多。

比如输入 `nums = [2,2',2'']`，产生的回溯树如下：

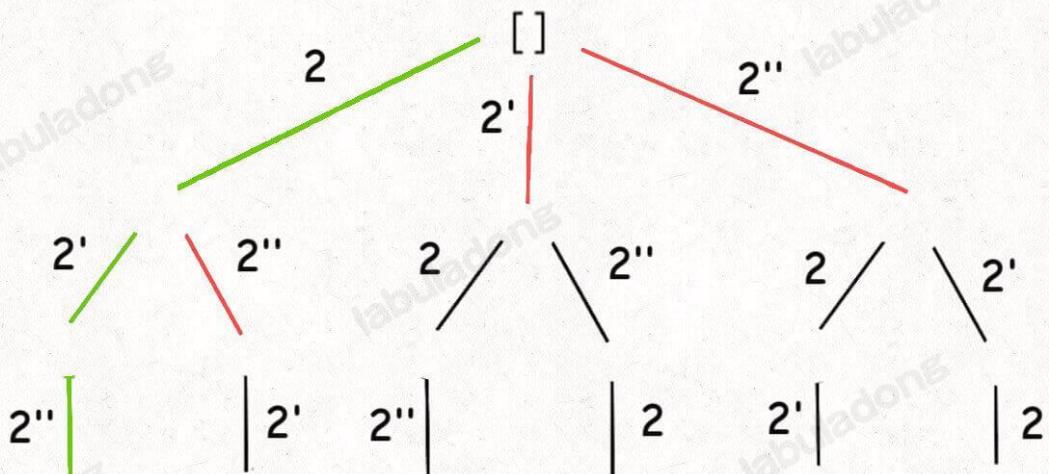
## 排列树



© labuladong

如果用绿色树枝代表 `backtrack` 函数遍历过的路径，红色树枝代表剪枝逻辑的触发，那么 `!used[i - 1]` 这种剪枝逻辑得到的回溯树长这样：

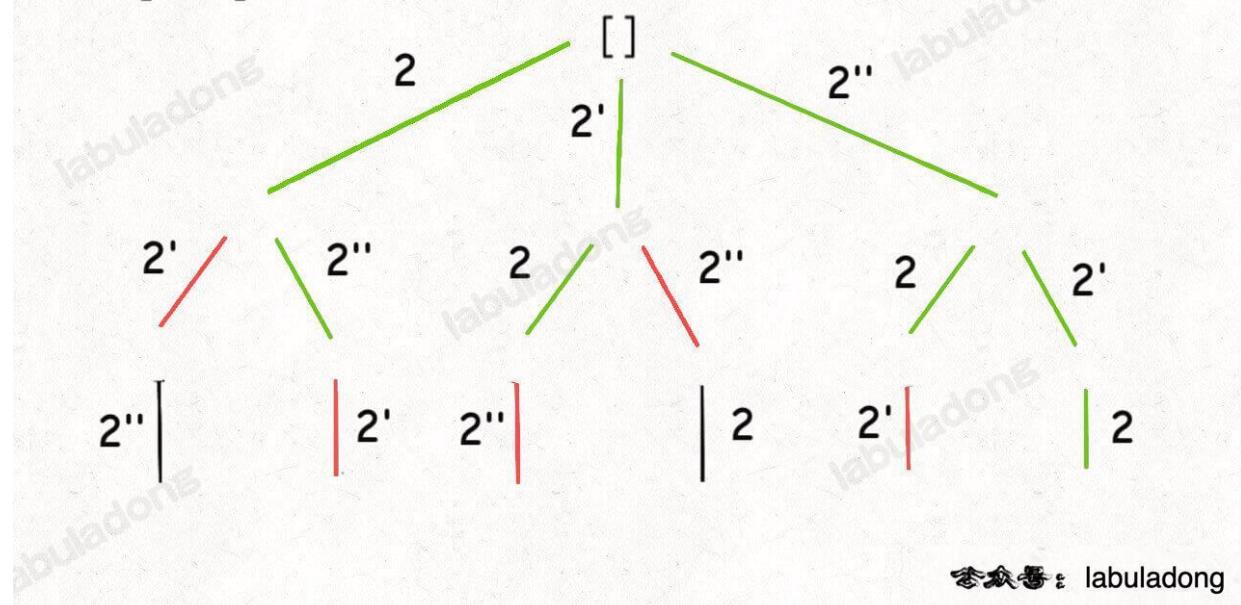
### `!used[i - 1]` 剪枝情况



© labuladong

而 `used[i - 1]` 这种剪枝逻辑得到的回溯树如下：

`used[i - 1]` 剪枝情况



可以看到，`!used[i - 1]` 这种剪枝逻辑剪得干净利落，而 `used[i - 1]` 这种剪枝逻辑虽然也能得到无重结果，但它剪掉的树枝较少，存在的无效计算较多，所以效率会差一些。

你可以使用可视化面板的「编辑」按钮自行修改代码验证一下，看看两种写法产生的回溯树有何差别：

▶ 代码可视化动画

当然，关于排列去重，也有读者提出别的剪枝思路：

```
void backtrack(int[] nums, LinkedList<Integer> track) {
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    // 记录之前树枝上元素的值
    // 题目说 -10 <= nums[i] <= 10, 所以初始化为特殊值
    int prevNum = -666;
    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            continue;
        }
        if (nums[i] == prevNum) {
            continue;
        }

        track.add(nums[i]);
        used[i] = true;
        // 记录这条树枝上的值
        prevNum = nums[i];

        backtrack(nums, track);

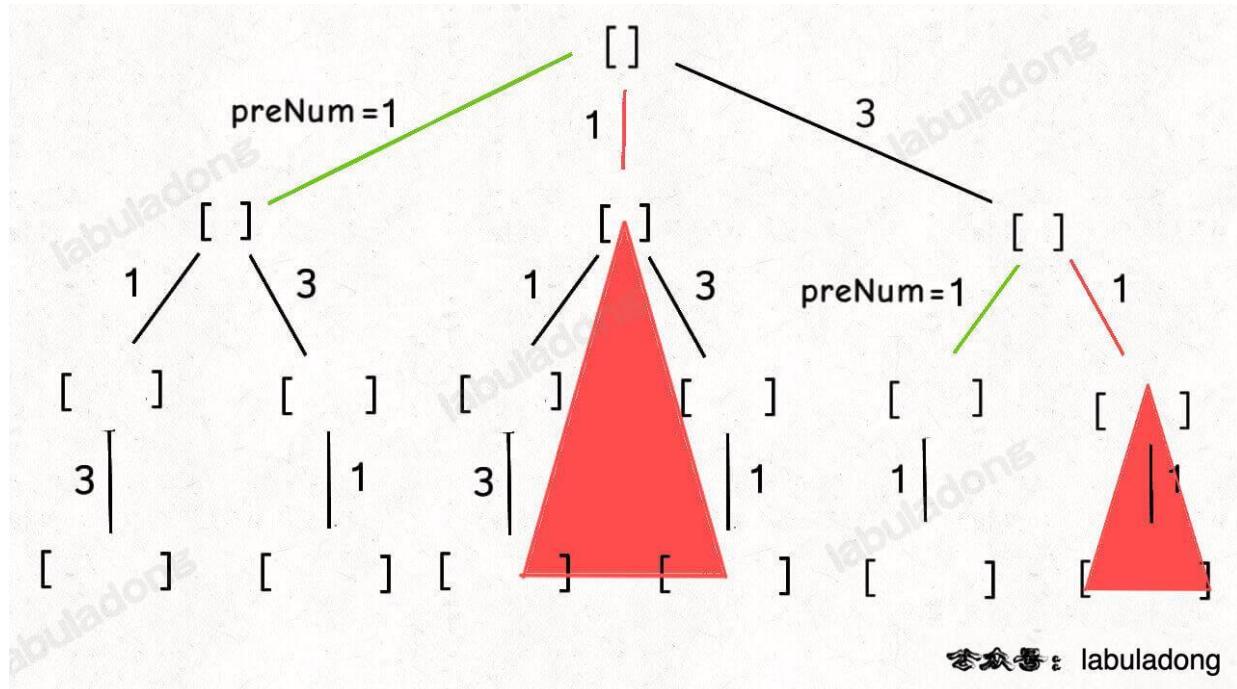
        track.removeLast();
        used[i] = false;
    }
}
```

```

    }
}

```

这个思路也是对的，设想一个节点出现了相同的树枝：



如果不作处理，这些相同树枝下面的子树也会长得一模一样，所以会出现重复的排列。

因为排序之后所有相等的元素都挨在一起，所以只要用 `prevNum` 记录前一条树枝的值，就可以避免遍历值相同的树枝，从而避免产生相同的子树，最终避免出现重复的排列。

好了，这样包含重复输入的排列问题也解决了。

## 子集/组合（元素无重可复选）

终于到了最后一种类型了：输入数组无重复元素，但每个元素可以被无限次使用。

直接看力扣第 39 题「组合总和」：

给你一个无重复元素的整数数组 `candidates` 和一个目标和 `target`，找出 `candidates` 中可以使数字和为目标数 `target` 的所有组合。`candidates` 中的每个数字可以无限制重复被选取。

函数签名如下：

```
List<List<Integer>> combinationSum(int[] candidates, int target)
```

比如输入 `candidates = [1,2,3]`, `target = 3`，算法应该返回：

```
[ [1,1,1], [1,2], [3] ]
```

这道题说是组合问题，实际上也是子集问题：`candidates` 的哪些子集的和为 `target`？

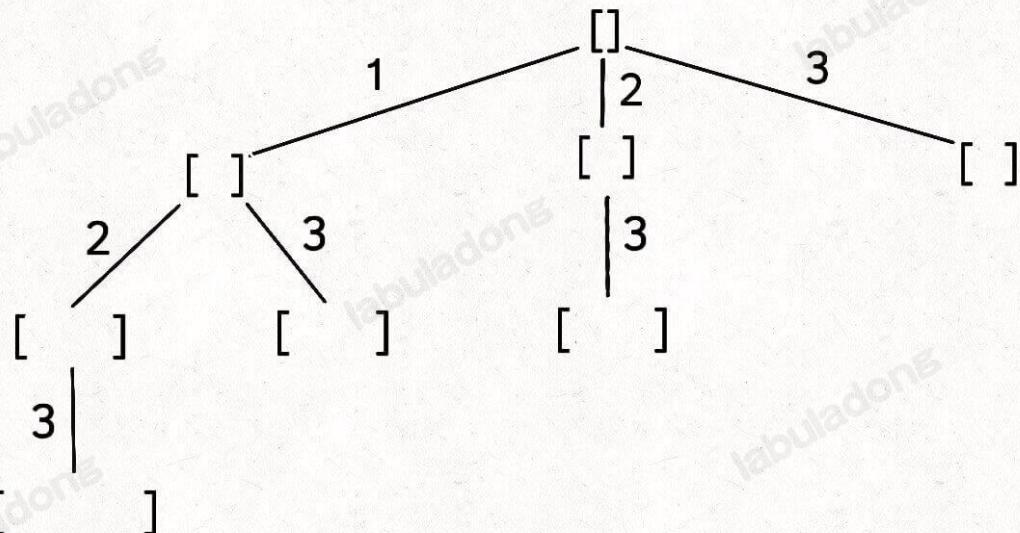
想解决这种类型的问题，也得回到回溯树上，我们不妨先思考思考，标准的子集/组合问题是如何保证不重复使用元素的？

答案在于 `backtrack` 递归时输入的参数 `start`:

```
// 无重组合的回溯算法框架
void backtrack(int[] nums, int start) {
    for (int i = start; i < nums.length; i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i + 1);
        // ...
    }
}
```

这个 `i` 从 `start` 开始，那么下一层回溯树就是从 `start + 1` 开始，从而保证 `nums[start]` 这个元素不会被重复使用：

组合/子集树

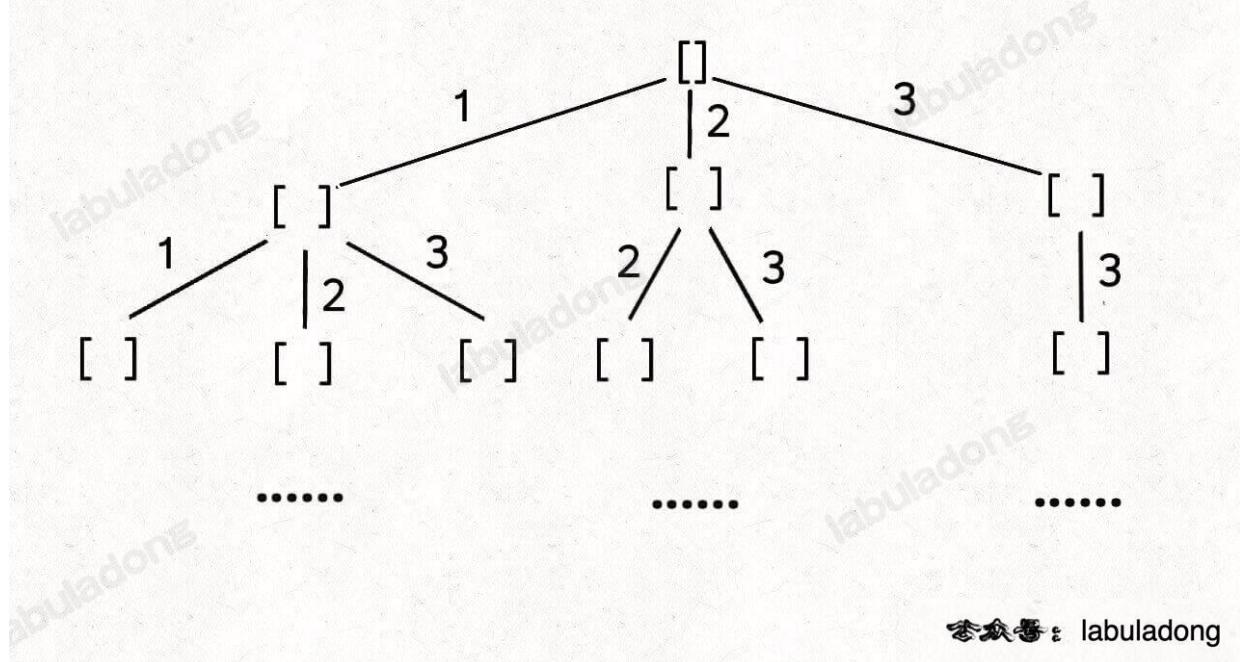


© labuladong

那么反过来，如果我想让每个元素被重复使用，我只要把 `i + 1` 改成 `i` 即可：

```
// 可重组合的回溯算法框架
void backtrack(int[] nums, int start) {
    for (int i = start; i < nums.length; i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i);
        // ...
    }
}
```

这相当于给之前的回溯树添加了一条树枝，在遍历这棵树的过程中，一个元素可以被无限次使用：



✿✿✿ labuladong

当然，这样这棵回溯树会永远生长下去，所以我们的递归函数需要设置合适的 base case 以结束算法，即路径和大于 `target` 时就没必要再遍历下去了。

这道题的解法代码如下：

```

class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();
    // 记录 track 中的路径和
    int trackSum = 0;

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        if (candidates.length == 0) {
            return res;
        }
        backtrack(candidates, 0, target);
        return res;
    }

    // 回溯算法主函数
    void backtrack(int[] nums, int start, int target) {
        // base case, 找到目标和, 记录结果
        if (trackSum == target) {
            res.add(new LinkedList<>(track));
            return;
        }
        // base case, 超过目标和, 停止向下遍历
        if (trackSum > target) {
            return;
        }
        // 回溯算法标准框架
        for (int i = start; i < nums.length; i++) {
            // 选择 nums[i]
            trackSum += nums[i];
            track.add(nums[i]);
        }
    }
}

```

```
// 递归遍历下一层回溯树
backtrack(nums, i, target);
// 同一元素可重复使用，注意参数
// 撤销选择 nums[i]
trackSum -= nums[i];
track.removeLast();
}
}
}
```

## ▶ 代码可视化动画

## 排列（元素无重可复选）

力扣上没有题目直接考察这个场景，我们不妨先想一下，`nums` 数组中的元素无重复且可复选的情况下，会有哪些排列？

比如输入 `nums = [1,2,3]`，那么这种条件下的全排列共有  $3^3 = 27$  种：

```
[  
[1,1,1],[1,1,2],[1,1,3],[1,2,1],[1,2,2],[1,2,3],[1,3,1],[1,3,2],[1,3,3],  
[2,1,1],[2,1,2],[2,1,3],[2,2,1],[2,2,2],[2,2,3],[2,3,1],[2,3,2],[2,3,3],  
[3,1,1],[3,1,2],[3,1,3],[3,2,1],[3,2,2],[3,2,3],[3,3,1],[3,3,2],[3,3,3]  
]
```

标准的全排列算法利用 `used` 数组进行剪枝，避免重复使用同一个元素。如果允许重复使用元素的话，直接放飞自我，去除所有 `used` 数组的剪枝逻辑就行了。

那这个问题就简单了，代码如下：

```
class Solution {  
  
    List<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> track = new LinkedList<>();  
  
    public List<List<Integer>> permuteRepeat(int[] nums) {
        backtrack(nums);
        return res;
    }  
  
    // 回溯算法核心函数
    void backtrack(int[] nums) {
        // base case, 到达叶子节点
        if (track.size() == nums.length) {
            // 收集叶子节点上的值
            res.add(new LinkedList(track));
            return;
        }  
  
        // 回溯算法标准框架
        for (int i = 0; i < nums.length; i++) {
            // 做选择
            track.add(nums[i]);
            // 进入下一层回溯树
        }
    }
}
```

```
        backtrack(nums);
        // 取消选择
        track.removeLast();
    }
}
}
```

至此，排列/组合/子集问题的九种变化就都讲完了。

## 最后总结

来回顾一下排列/组合/子集问题的三种形式在代码上的区别。

由于子集问题和组合问题本质上是一样的，无非就是 base case 有一些区别，所以把这两个问题放在一起看。

**形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，`backtrack` 核心代码如下：**

```
// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i + 1);
        // 撤销选择
        track.removeLast();
    }
}

// 排列问题回溯算法框架
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 撤销选择
        track.removeLast();
        used[i] = false;
    }
}
```

**形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次，其关键在于排序和剪枝，`backtrack` 核心代码如下：**

```
Arrays.sort(nums);
// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
```

```

// 回溯算法标准框架
for (int i = start; i < nums.length; i++) {
    // 剪枝逻辑，跳过值相同的相邻树枝
    if (i > start && nums[i] == nums[i - 1]) {
        continue;
    }
    // 做选择
    track.addLast(nums[i]);
    // 注意参数
    backtrack(nums, i + 1);
    // 撤销选择
    track.removeLast();
}

Arrays.sort(nums);
// 排列问题回溯算法框架
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 剪枝逻辑，固定相同的元素在排列中的相对位置
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 撤销选择
        track.removeLast();
        used[i] = false;
    }
}

```

形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次，只要删掉去重逻辑即可，`backtrack` 核心代码如下：

```

// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i);
        // 撤销选择
        track.removeLast();
    }
}

// 排列问题回溯算法框架
void backtrack(int[] nums) {

```

```
for (int i = 0; i < nums.length; i++) {  
    // 做选择  
    track.addLast(nums[i]);  
    backtrack(nums);  
    // 撤销选择  
    track.removeLast();  
}  
}
```

只要从树的角度思考，这些问题看似复杂多变，实则改改 base case 就能解决，这也是为什么我在 [学习算法和数据结构的框架思维](#) 和 [手把手刷二叉树（纲领篇）](#) 中强调树类型题目重要性的原因。

如果你能够看到这里，真得给你鼓掌，相信你以后遇到各种乱七八糟的算法题，也能一眼看透它们的本质，以不变应万变。另外，考虑到篇幅，本文并没有对这些算法进行复杂度的分析，你可以使用我在 [算法时空复杂度分析实用指南](#) 讲到的复杂度分析方法尝试自己分析它们的复杂度。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1079. Letter Tile Possibilities</a>	<a href="#">1079. 活字印刷</a>	
<a href="#">131. Palindrome Partitioning</a>	<a href="#">131. 分割回文串</a>	
<a href="#">17. Letter Combinations of a Phone Number</a>	<a href="#">17. 电话号码的字母组合</a>	
<a href="#">254. Factor Combinations</a>	<a href="#">254. 因子的组合</a>	
<a href="#">267. Palindrome Permutation II</a>	<a href="#">267. 回文排列 II</a>	
<a href="#">368. Largest Divisible Subset</a>	<a href="#">368. 最大整除子集</a>	
<a href="#">491. Non-decreasing Subsequences</a>	<a href="#">491. 递增子序列</a>	
<a href="#">638. Shopping Offers</a>	<a href="#">638. 大礼包</a>	
<a href="#">967. Numbers With Same Consecutive Differences</a>	<a href="#">967. 连续差相同的数字</a>	
<a href="#">996. Number of Squareful Arrays</a>	<a href="#">996. 正方形数组的数目</a>	
-	<a href="#">剑指 Offer 38. 字符串的排列</a>	
-	<a href="#">剑指 Offer II 079. 所有子集</a>	
-	<a href="#">剑指 Offer II 080. 含有 k 个元素的组合</a>	
-	<a href="#">剑指 Offer II 081. 允许重复选择元素的组合</a>	
-	<a href="#">剑指 Offer II 083. 没有重复元素集合的全排列</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 算法时空复杂度分析实用指南



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

我以前的文章主要都是讲解算法的原理和解题的思维，对时间复杂度和空间复杂度的分析经常一笔带过，主要是基于以下两个原因：

- 1、对于偏小白的读者，我希望你集中精力理解算法原理。如果加入太多偏数学的内容，很容易把人劝退。
- 2、正确理解常用算法底层原理，是进行复杂度的分析的前提。尤其是递归相关的算法，只有你从树的角度进行思考和分析，才能正确分析其复杂度。

鉴于现在历史文章已经涵盖了所有常见算法的核心原理，所以我专门写一篇时空复杂度的分析指南，授人以鱼不如授人以渔，教给你一套通用的方法分析任何算法的时空复杂度。

本文会篇幅较长，会涵盖如下几点：

- 1、利用时间复杂度反推解题思路，减少试错时间。
- 2、时间都去哪儿了？哪些常见的编码失误会导致算法超时。
- 3、Big O 表示法的几个基本特点。
- 4、非递归算法中的时间复杂度分析。
- 5、数据结构 API 的效率衡量方法（摊还分析）。
- 6、递归算法的时间/空间复杂度的分析方法，这部分是重点，我会用动态规划和回溯算法举例。

在讲解复杂度的概念和具体计算方法之前，我先说一些实战中常见的技巧和容易踩的坑。

## 借助复杂度反推解题思路

不要以为复杂度分析是专门用来难为你的，它其实是来帮你的，它是来偷偷告诉你解题思路的。

**你应该在开始写代码之前就留意题目给的数据规模**，因为复杂度分析可以避免你在错误的思路上浪费时间，有时候它甚至可以直接告诉你这道题用什么算法。

为啥这样说呢，因为一般题目都会告诉我们测试用例的数据规模有多大，我们可以根据这个数据规模反推这道题能够允许的时间复杂度在什么范围，进一步反推我们应该要用什么算法。

举例来说吧，比如一个题目给你输入一个数组，其长度能够达到  $10^6$  这个量级，那么我们肯定可以知道这道题的时间复杂度大概要小于  $O(N^2)$ ，得优化成  $O(N \log N)$  或者  $O(N)$  才行。因为如果你写的算法是  $O(N^2)$  的，最大的复杂度会达到  $10^{12}$  这个量级，在大部分判题系统上都是跑不过去的。

为了把复杂度控制在  $O(N \log N)$  或者  $O(N)$ ，我们的选择范围就缩小了，可能符合条件的做法是：对数组进行排序处理、前缀和、双指针、一维 dp 等等，从这些思路切入就比较靠谱。像嵌套 for 循环、二维 dp、回溯算法这些思路，基本

可以直接排除掉了。

再举个更直接的例子，如果你发现题目给的数据规模很小，比如数组长度  $N$  不超过 20 这样的，那么我们可以断定这道题大概率要用暴力穷举算法。

因为判题平台肯定是尽可能扩大数据规模难为你，它一反常态给这么小的数据规模，肯定是因为最优解就是指数/阶乘级别的复杂度。你放心用 [回溯算法](#) 招呼它就行了，不用想别的算法了。

所以说啊，很多读者看题都不看那个数据规模，上来就闷声写代码，这是不对滴。你先把题目给的所有信息都考虑进去，再写代码，这样才能少走弯路。

## 由于编码失误导致复杂度异常

这部分主要总结一下大家（尤其是初学者）在写代码时容易犯的一些编码层面的错误。这些错误会产生预期之外的时间消耗，拖慢你的算法运行，甚至导致超时。

### 使用了标准输出

在写算法题时，我们可能会用到 `print/console.log` 函数来打印一些状态，以便调试算法。

但你准备提交代码时，记得把这些输出语句注释掉，因为标准输出属于 I/O 操作，会很大程度上拖慢你的算法代码运行。

### 错误地「传值」而非「传引用」

比如 C++ 里面，函数参数默认是传值的，比如你传入一个 `vector` 参数，那么这个数据结构会被完整地复制一份，导致额外的时间和空间消耗。尤其是当这个函数是个递归函数的时候，错误地传值几乎必然导致超时或超内存。

正确的做法是传引用，比如 `vector<int>&`，这样就不会产生额外的复制开销。使用其他语言的读者可以自查是否存在类似的问题，一定要搞清楚你的编程语言的参数传递机制。

### 接口对象的底层实现不明

这应该算是一个比较刁钻的问题，一般在 Java 这种面向对象的语言出现，出现地比较少，但也必须引起注意。

Java 中 `List` 是一个接口，它有很多实现类，比如 `ArrayList`、`LinkedList` 等等。如果你学了基础知识章节的 [手把手带你实现动态数组](#) 和 [手把手带你实现双链表](#)，就知道 `ArrayList` 和 `LinkedList` 的很多方法复杂度不同，比如 `ArrayList` 的 `get` 方法是  $O(1)$ ，而 `LinkedList` 的 `get` 方法是  $O(N)$ 。

那么如果函数参数给你传入一个 `List` 类型的参数，你敢不敢直接 `list.get(i)` 进行随机访问？不敢吧。

这种情况，一般需要自己创建一个 `ArrayList`，把传入的 `List` 的元素全部复制过去，再通过索引进行访问，这样比较保险。

主要就上面这几个吧，它们都是非算法思想层面的问题，大家留心注意一下应该没什么问题。下面开始正式讲解算法复杂度分析。

## Big O 表示法

$$O(g(n)) = \{ f(n) : \text{存在正常量 } c \text{ 和 } n_0, \text{ 使得对所有 } n \geq n_0, \text{ 有 } 0 \leq f(n) \leq c*g(n) \}$$

我们常用的这个符号  $O$ （大写字母  $o$ ，不是数字 0）其实代表一个函数的集合，比如  $O(n^2)$  代表着一个由  $g(n) = n^2$  派生出来的一个函数集合；我们说一个算法的时间复杂度为  $O(n^2)$ ，意思就是描述该算法的复杂度的函数属于这个函数集合之中。

理论上，你看明白这个抽象的数学定义，就可以解答你关于 Big O 表示法的一切疑问了。

但考虑到大部分人看到数学定义就头晕，我给你列举两个复杂度分析中会用到的特性，记住这两个就够用了。

### 1、只保留增长速率最快的项，其他的项可以省略。

首先，乘法和加法中的常数因子都可以忽略不计，比如下面的例子：

$$\begin{aligned}O(2N + 100) &= O(N) \\O(2^{N+1}) &= O(2 * 2^N) = O(2^N) \\O(M + 3N + 99) &= O(M + N)\end{aligned}$$

当然，不要见到常数就消，有的常数消不得，可能会用到我们中学学过的指数运算法则：

$$O(2^{2N}) = O(4^N)$$

除了常数因子，增长速率慢的项在增长速率快的项面前也可以忽略不计：

$$\begin{aligned}O(N^3 + 999 * N^2 + 999 * N) &= O(N^3) \\O((N + 1) * 2^N) &= O(N * 2^N + 2^N) = O(N * 2^N)\end{aligned}$$

以上列举的都是最简单常见的例子，这些例子都可以被 Big O 记号的定义正确解释。如果你遇到更复杂的复杂度场景，也可以根据定义来判断自己的复杂度表达式是否正确。

### 2、Big O 记号表示复杂度的「上界」。

换句话说，只要你给出的是一个上界，用 Big O 记号表示就都是正确的。

比如如下代码：

```
for (int i = 0; i < N; i++) {
    print("hello world");
}
```

如果说这是一个算法，那么显然它的时间复杂度是  $O(N)$ 。但如果你非要说它的时间复杂度是  $O(N^2)$ ，理论上讲是可以的，因为  $O$  记号表示一个上界嘛，这个算法的时间复杂度确实不会超过  $N^2$  这个上界呀，虽然这个上界不够「紧」，但符合定义，所以理论上说也没毛病。

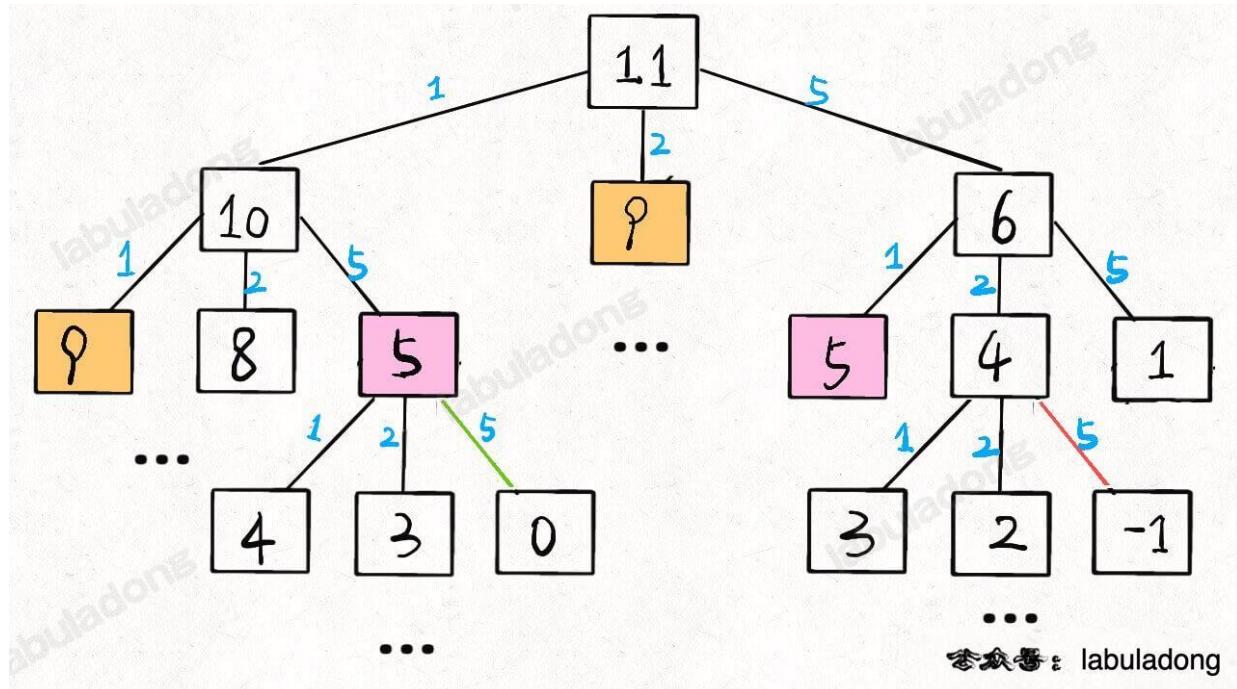
上述例子太简单，非要扩大它的时间复杂度上界显得没什么意义。但有些算法的复杂度会和算法的输入数据有关，没办法提前给出一个特别精确的时间复杂度，那么在这种情况下，用 Big O 记号扩大时间复杂度的上界就变得有意义了。

比如前文 [动态规划核心框架](#) 中讲到的凑零钱问题的暴力递归解法，核心代码框架如下：

```
// 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
int dp(int[] coins, int amount) {
    // base case
    if (amount == 0) return;
    // 状态转移
    for (int coin : coins) {
        dp(coins, amount - coin);
```

```
}
```

当 `amount = 11, coins = [1,2,5]` 时，算法的递归树就长这样：



后文会具体讲递归算法的时间复杂度计算方法，现在我们先求一下这棵递归树上的节点个数吧。

假设金额 `amount` 的值为  $N$ , `coins` 列表中元素个数为  $K$ , 那么这棵递归树就是一棵  $K$  叉树。但这棵树的生长和 `coins` 列表中的硬币面额有直接的关系，所以这棵树的形状会很不规则，导致我们很难精确地求出树上节点的总数。

对于这种情况，比较简单的处理方式就是按最坏情况做近似处理：

这棵树的高度有多高？不知道，那就按最坏情况来处理，假设全都是面额为 1 的硬币，这种情况下树高为  $N$ 。

这棵树的结构是什么样的？不知道，那就按最坏情况来处理，假设它是一棵满  $K$  叉树好了。

那么，这棵树上共有多少节点？都按最坏情况来处理，高度为  $N$  的一棵满  $K$  叉树，其节点总数为等比数列求和公式  $(K^{N-1})/(K-1)$ ，用 Big O 表示就是  $O(K^N)$ 。

当然，我们知道这棵树上的节点数其实没有这么多，但用  $O(K^N)$  表示一个粗略的上界是没问题的，也很容易看出来这是一个指数级的算法，需要想办法优化运行效率。

所以，有时候你自己估算出来的时间复杂度和别人估算的复杂度不同，并不一定代表谁算错了，可能你俩都是对的，只是是估算的精度不同。

理论上，我们当然希望得到一个比较准确比较「紧」的上界，但想得到准确的上界对数学的功力有一定要求，我们针对面试刷题的话不妨退而求其次，只要数量级（线性/指数级/对数级/平方级等）能对上就没问题。

在算法领域，除了用 Big O 表示渐进上界，还有渐进下界、渐进紧确界等边界的表示方法，有兴趣的读者可以自行搜索。不过从实用的角度看，以上对 Big O 记号表示法的讲解就够用了。

## 非递归算法分析

非递归算法的空间复杂度一般很容易计算，你看它有没有申请数组之类的存储空间就行了，所以我主要说下时间复杂度的分析。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 双指针技巧秒杀七道链表题目



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">142. Linked List Cycle II</a>	<a href="#">142. 环形链表 II</a>	🟡
<a href="#">23. Merge k Sorted Lists</a>	<a href="#">23. 合并K个升序链表</a>	🔴
-	<a href="#">剑指 Offer 22. 链表中倒数第k个节点</a>	🟢
<a href="#">21. Merge Two Sorted Lists</a>	<a href="#">21. 合并两个有序链表</a>	🟢
<a href="#">141. Linked List Cycle</a>	<a href="#">141. 环形链表</a>	🟢
<a href="#">19. Remove Nth Node From End of List</a>	<a href="#">19. 删除链表的倒数第 N 个结点</a>	🟡
<a href="#">86. Partition List</a>	<a href="#">86. 分隔链表</a>	🟡
<a href="#">876. Middle of the Linked List</a>	<a href="#">876. 链表的中间结点</a>	🟢
<a href="#">160. Intersection of Two Linked Lists</a>	<a href="#">160. 相交链表</a>	🟢

阅读本文前，你需要先学习：

- 链表基础
- 链表实现

tip：本文有视频版：[链表双指针技巧全面汇总](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文总结一下单链表的基本技巧，每个技巧都对应着至少一道算法题：

- 1、合并两个有序链表
- 2、链表的分解
- 3、合并  $k$  个有序链表
- 4、寻找单链表的倒数第  $k$  个节点
- 5、寻找单链表的中点
- 6、判断单链表是否包含环并找出环起点
- 7、判断两个单链表是否相交并找出交点

这些解法都用到了双指针技巧，所以说对于单链表相关的题目，双指针的运用是非常广泛的，下面我们就来一个一个看。

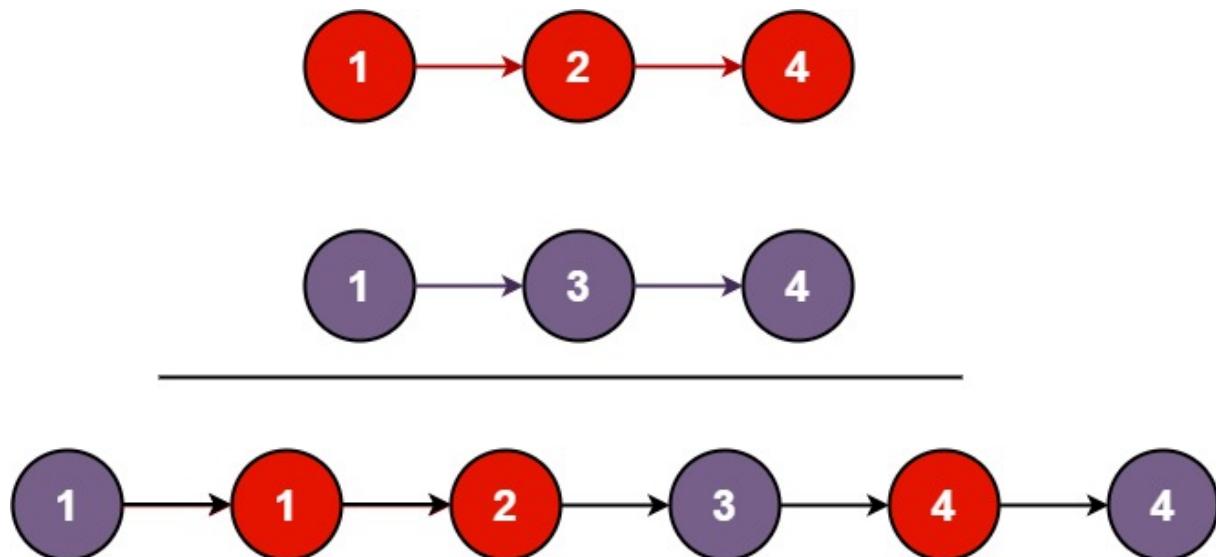
## 合并两个有序链表

这是最基本的链表技巧，力扣第 21 题「合并两个有序链表」就是这个问题，给你输入两个有序链表，请你把他俩合并成一个新的有序链表：

### ▼ 21. 合并两个有序链表 [Leetcode | 力扣](#)

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。

示例 1：



```
输入: l1 = [1,2,4], l2 = [1,3,4]
输出: [1,1,2,3,4,4]
```

示例 2：

```
输入: l1 = [], l2 = []
输出: []
```

示例 3：

```
输入: l1 = [], l2 = [0]
输出: [0]
```

提示：

- 两个链表的节点数目范围是  $[0, 50]$
- $-100 \leq \text{node.val} \leq 100$
- $\text{l1}$  和  $\text{l2}$  均按 非递减顺序 排列

```
// 函数签名如下
ListNode mergeTwoLists(ListNode l1, ListNode l2);
```

这题比较简单，我们直接看解法：

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1), p = dummy;
        ListNode p1 = l1, p2 = l2;

        while (p1 != null && p2 != null) {
            // 比较 p1 和 p2 两个指针
            // 将值较小的的节点接到 p 指针
            if (p1.val > p2.val) {
                p.next = p2;
                p2 = p2.next;
            } else {
                p.next = p1;
                p1 = p1.next;
            }
            // p 指针不断前进
            p = p.next;
        }

        if (p1 != null) {
            p.next = p1;
        }

        if (p2 != null) {
            p.next = p2;
        }

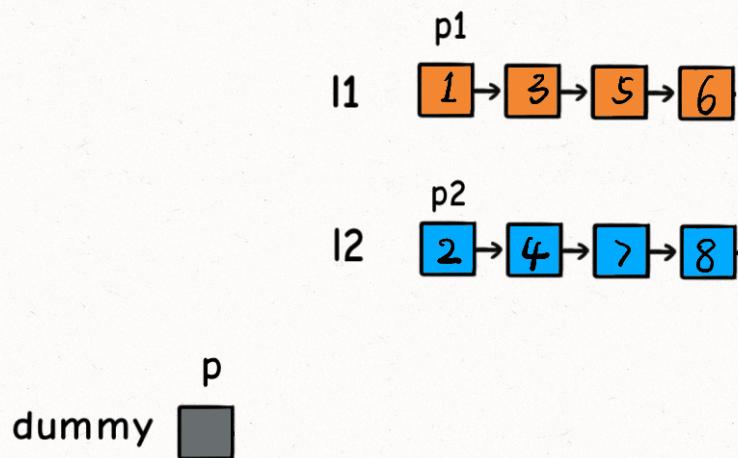
        return dummy.next;
    }
}
```

---

▶  代码可视化动画 

---

我们的 while 循环每次比较 `p1` 和 `p2` 的大小，把较小的节点接到结果链表上，看如下 GIF：



公众号: labuladong

形象地理解，这个算法的逻辑类似于拉拉链，**l1**, **l2** 类似于拉链两侧的锯齿，指针 **p** 就好像拉链的拉索，将两个有序链表合并；或者说这个过程像蛋白酶合成蛋白质，**l1**, **l2** 就好比两条氨基酸，而指针 **p** 就好像蛋白酶，将氨基酸组合成蛋白质。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 **dummy** 节点。你可以试试，如果不使用 **dummy** 虚拟节点，代码会复杂一些，需要额外处理指针 **p** 为空的情况。而有了 **dummy** 节点这个占位符，可以避免处理空指针的情况，降低代码的复杂性。

经常有读者问我，什么时候需要用虚拟头结点？我这里总结下：当你需要创造一条新链表的时候，可以使用虚拟头结点简化边界情况的处理。

比如说，让你把两条有序链表合并成一条新的有序链表，是不是要创造一条新链表？再比你想把一条链表分解成两条链表，是不是也在创造新链表？这些情况都可以使用虚拟头结点简化边界情况的处理。

## 单链表的分解

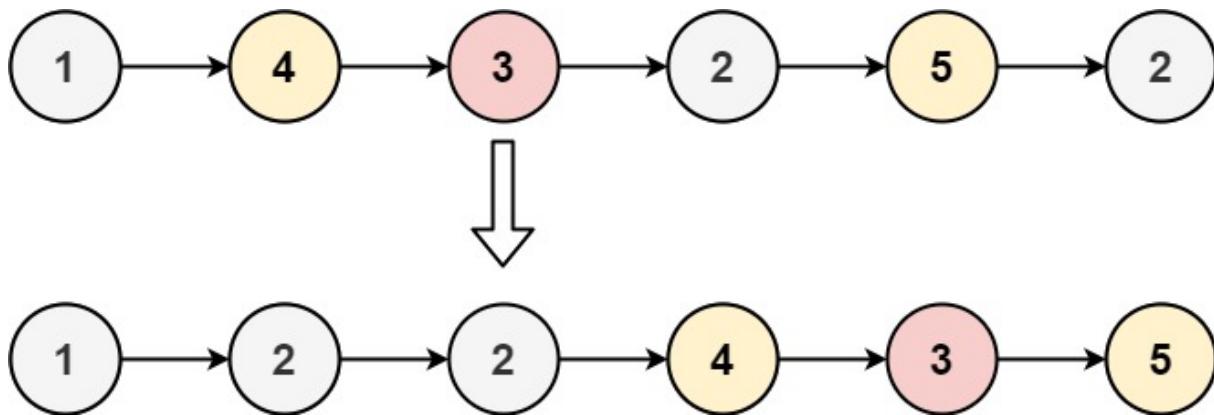
直接看下力扣第 86 题「分隔链表」：

### ▼ 86. 分隔链表 [Leetcode](#) | [力扣](#)

给你一个链表的头节点 **head** 和一个特定值 **x**，请你对链表进行分隔，使得所有 **小于 x** 的节点都出现在 **大于或等于 x** 的节点之前。

你应当 **保留** 两个分区中每个节点的初始相对位置。

示例 1：



输入: head = [1,4,3,2,5,2], x = 3  
输出: [1,2,2,4,3,5]

### 示例 2:

输入: head = [2,1], x = 2  
输出: [1,2]

### 提示:

- 链表中节点的数目在范围 [0, 200] 内
- 100 <= node.val <= "100"
- 200 <= x <= "200"

在合并两个有序链表时让你合二为一，而这里需要分解让你把原链表一分为二。具体来说，我们可以把原链表分成两个小链表，一个链表中的元素大小都小于  $x$ ，另一个链表中的元素都大于等于  $x$ ，最后再把这两条链表接到一起，就得到了题目想要的结果。

整体逻辑和合并有序链表非常相似，细节直接看代码吧，注意虚拟头结点的运用：

```
class Solution {  
    public ListNode partition(ListNode head, int x) {  
        // 存放小于 x 的链表的虚拟头结点  
        ListNode dummy1 = new ListNode(-1);  
        // 存放大于等于 x 的链表的虚拟头结点  
        ListNode dummy2 = new ListNode(-1);  
        // p1, p2 指针负责生成结果链表  
        ListNode p1 = dummy1, p2 = dummy2;  
        // p 负责遍历原链表，类似合并两个有序链表的逻辑  
        // 这里是将一个链表分解成两个链表  
        ListNode p = head;  
        while (p != null) {  
            if (p.val >= x) {  
                p2.next = p;  
                p2 = p2.next;  
            } else {  
                p1.next = p;  
                p1 = p1.next;  
            }  
        }  
        // 不能直接让 p 指针前进，
```

```
// p = p.next
// 断开原链表中的每个节点的 next 指针
ListNode temp = p.next;
p.next = null;
p = temp;
}
// 连接两个链表
p1.next = dummy2.next;

return dummy1.next;
}
```

我知道有很多读者会对这段代码有疑问：

```
// 不能直接让 p 指针前进,
// p = p.next
// 断开原链表中的每个节点的 next 指针
ListNode temp = p.next;
p.next = null;
p = temp;
```

不多废话，直接借助我们的可视化面板看一下就明白了。首先看下正确的写法：

---

► 🎃 代码可视化动画🎃

---

如果你不断开原链表中的每个节点的 `next` 指针，那么就会出错，因为结果链表中会包含一个环：

---

► 🎃 代码可视化动画🎃

---

总的来说，如果我们需要把原链表的节点接到新链表上，而不是 `new` 新节点来组成新链表的话，那么断开节点和原链表之间的链接可能是必要的。那其实我们可以养成一个好习惯，但凡遇到这种情况，就把原链表的节点断开，这样就不会出错了。

## 合并 k 个有序链表

看下力扣第 23 题「合并K个升序链表」：

▼ 23. 合并 K 个升序链表 [Leetcode](#) | [力扣](#)

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入: lists = [[1,4,5],[1,3,4],[2,6]]
输出: [1,1,2,3,4,4,5,6]
解释: 链表数组如下:
[
  1->4->5,
  1->3->4,
  2->6
```

```
]  
将它们合并到一个有序链表中得到。
```

```
1->1->2->3->4->4->5->6
```

### 示例 2:

```
输入: lists = []  
输出: []
```

### 示例 3:

```
输入: lists = [[]]  
输出: []
```

### 提示:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` 按升序排列
- `lists[i].length` 的总和不超过 `10^4`

```
// 函数签名如下  
ListNode mergeKLists(ListNode[] lists);
```

合并 `k` 个有序链表的逻辑类似合并两个有序链表，难点在于，如何快速得到 `k` 个节点中的最小节点，接到结果链表上？

这里我们就要用到优先级队列这种数据结构，把链表节点放入一个最小堆，就可以每次获得 `k` 个节点中的最小节点。关于优先级队列可以参考 [优先级队列（二叉堆）原理及实现](#)，本文不展开。

```
class Solution {  
    public ListNode mergeKLists(ListNode[] lists) {  
        if (lists.length == 0) return null;  
        // 虚拟头结点  
        ListNode dummy = new ListNode(-1);  
        ListNode p = dummy;  
        // 优先级队列，最小堆  
        PriorityQueue<ListNode> pq = new PriorityQueue<>(  
            lists.length, (a, b)->(a.val - b.val));  
        // 将 k 个链表的头结点加入最小堆  
        for (ListNode head : lists) {  
            if (head != null) {  
                pq.add(head);  
            }  
        }  
  
        while (!pq.isEmpty()) {  
            // 获取最小节点，接到结果链表中  
        }  
    }  
}
```

```

ListNode node = pq.poll();
p.next = node;
if (node.next != null) {
    pq.add(node.next);
}
// p 指针不断前进
p = p.next;
}
return dummy.next;
}
}

```

这个算法是面试常考题，它的时间复杂度是多少呢？

优先队列 `pq` 中的元素个数最多是 `k`，所以一次 `poll` 或者 `add` 方法的时间复杂度是  $O(\log k)$ ；所有的链表节点都会被加入和弹出 `pq`，所以算法整体的时间复杂度是  $O(N \log k)$ ，其中 `k` 是链表的条数，`N` 是这些链表的节点总数。

## 单链表的倒数第 `k` 个节点

从前往后寻找单链表的第 `k` 个节点很简单，一个 `for` 循环遍历过去就找到了，但是如何寻找从后往前数的第 `k` 个节点呢？

那你可能说，假设链表有 `n` 个节点，倒数第 `k` 个节点就是正数第 `n - k + 1` 个节点，不也是一个 `for` 循环的事儿吗？

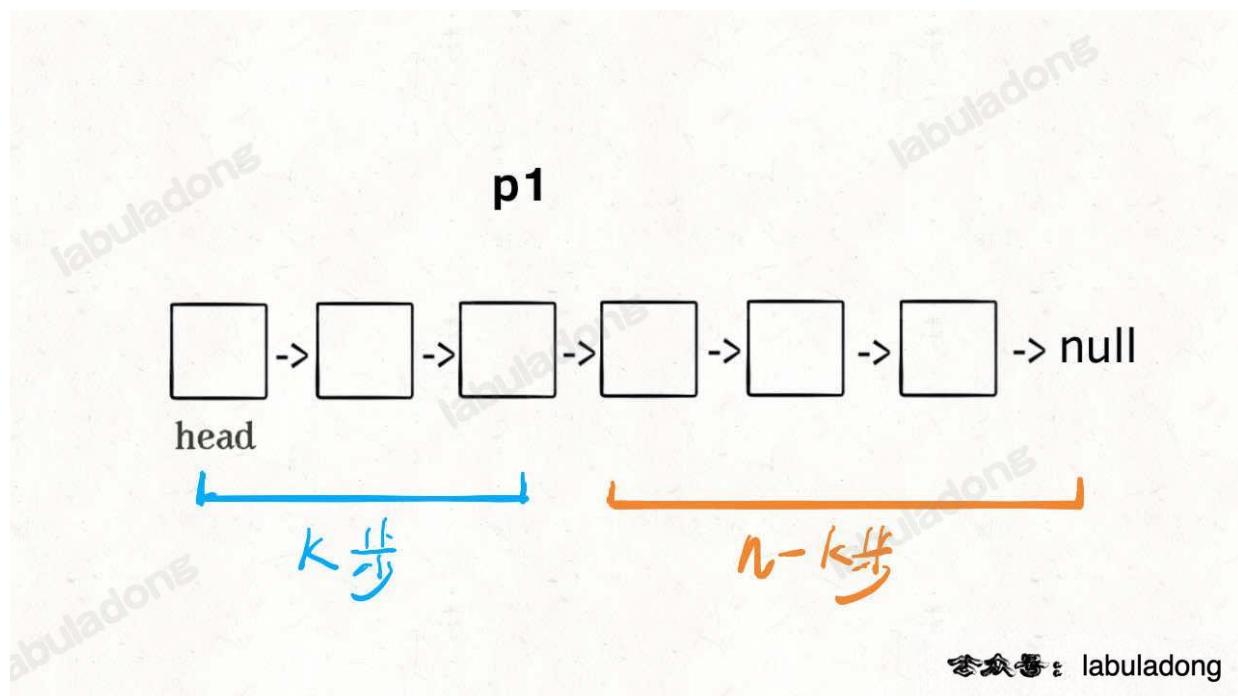
是的，但是算法题一般只给你一个 `ListNode` 头结点代表一条单链表，你不能直接得出这条链表的长度 `n`，而需要先遍历一遍链表算出 `n` 的值，然后再遍历链表计算第 `n - k + 1` 个节点。

也就是说，这个解法需要遍历两次链表才能得到倒数第 `k` 个节点。

那么，我们能不能只遍历一次链表，就算出倒数第 `k` 个节点？可以做到的，如果是面试问到这道题，面试官肯定也是希望你给出只需遍历一次链表的解法。

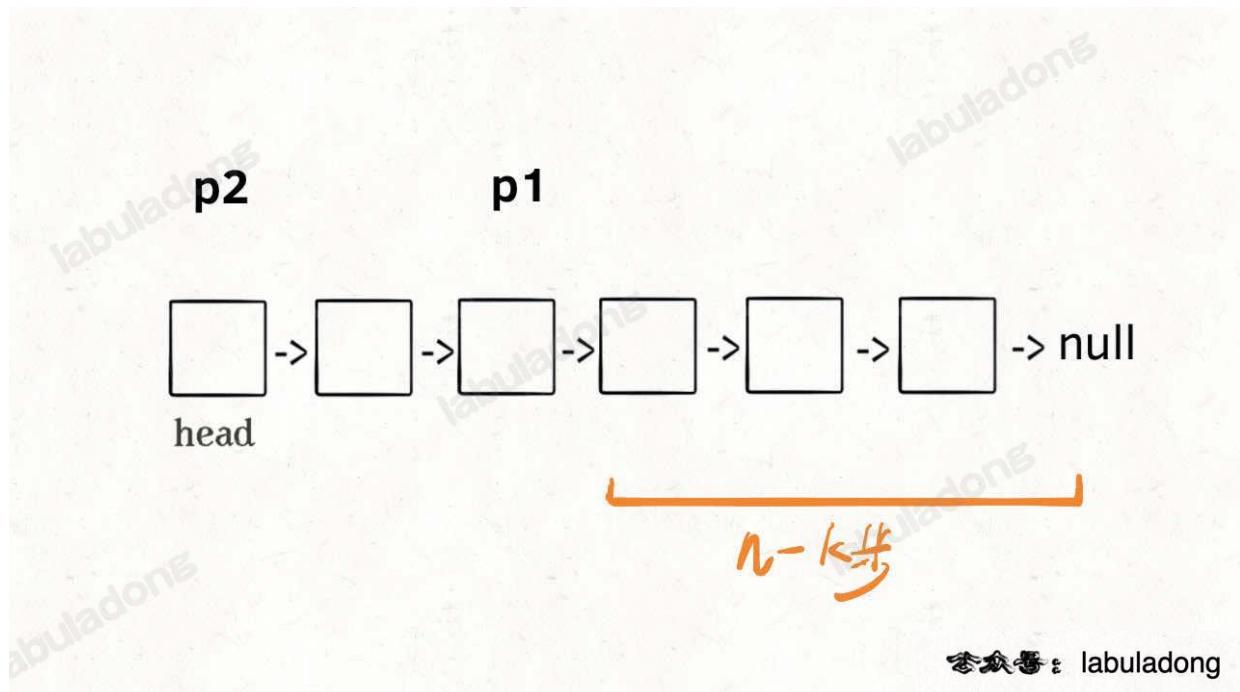
这个解法就比较巧妙了，假设 `k = 2`，思路如下：

首先，我们先让一个指针 `p1` 指向链表的头节点 `head`，然后走 `k` 步：

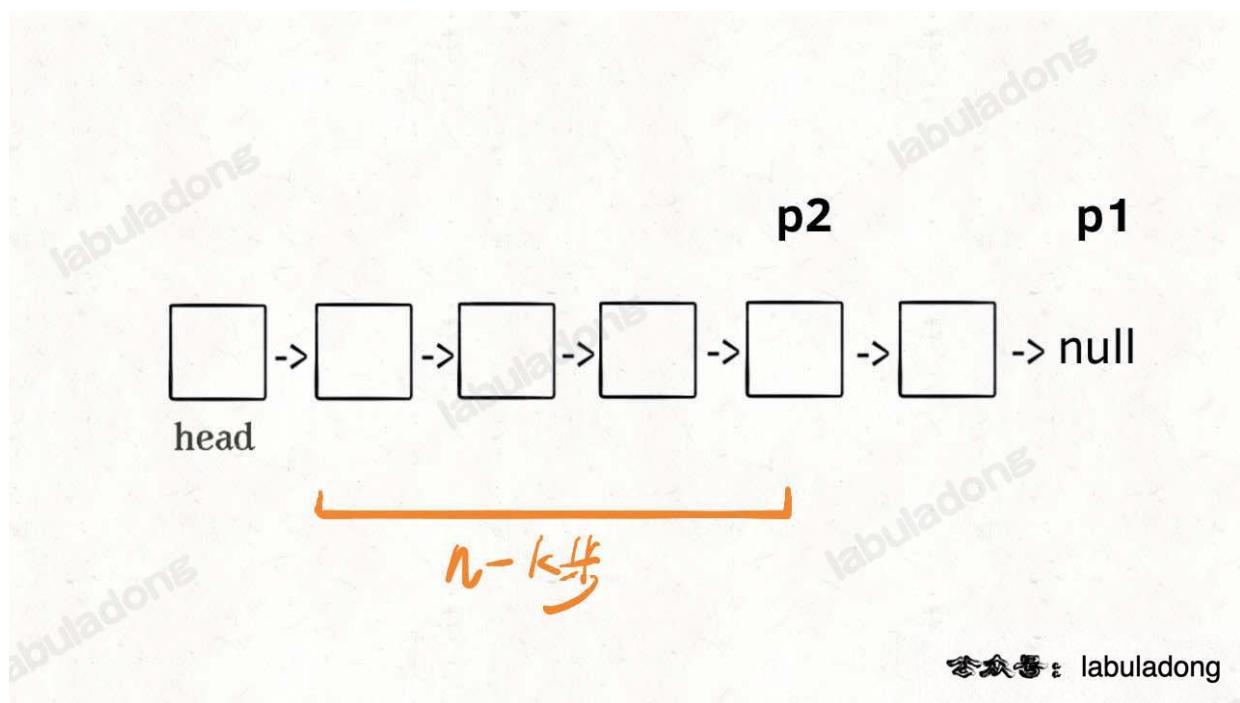


现在的 `p1`，只要再走 `n - k` 步，就能走到链表末尾的空指针了对吧？

趁这个时候，再用一个指针 `p2` 指向链表头节点 `head`：



接下来就很显然了，让 `p1` 和 `p2` 同时向前走，`p1` 走到链表末尾的空指针时前进了  $n - k$  步，`p2` 也从 `head` 开始前进了  $n - k$  步，停留在第  $n - k + 1$  个节点上，即恰好停在链表的倒数第  $k$  个节点上：



这样，只遍历了一次链表，就获得了倒数第  $k$  个节点 `p2`。

上述逻辑的代码如下：

```
// 返回链表的倒数第 k 个节点
ListNode findFromEnd(ListNode head, int k) {
    ListNode p1 = head;
    // p1 先走 k 步
    for (int i = 0; i < k; i++) {
        p1 = p1.next;
    }
}
```

```
ListNode p2 = head;
// p1 和 p2 同时走 n - k 步
while (p1 != null) {
    p2 = p2.next;
    p1 = p1.next;
}
// p2 现在指向第 n - k + 1 个节点, 即倒数第 k 个节点
return p2;
}
```

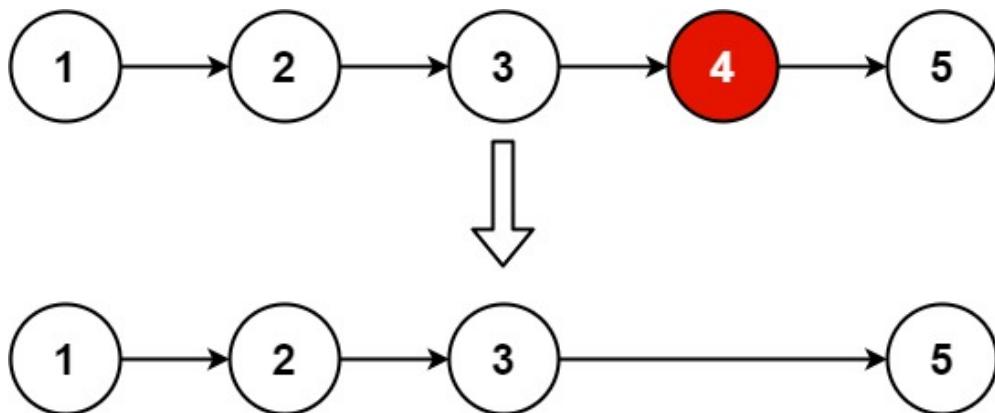
当然，如果用 big O 表示法来计算时间复杂度，无论遍历一次链表和遍历两次链表的时间复杂度都是  $O(N)$ ，但上述这个算法更有技巧性。

很多链表相关的算法题都会用到这个技巧，比如说力扣第 19 题「删除链表的倒数第 N 个结点」：

▼ 19. 删除链表的倒数第 N 个结点 [Leetcode](#) | [力扣](#)

给你一个链表，删除链表的倒数第  $n$  个结点，并且返回链表的头结点。

### 示例 1：



**输入:** head = [1,2,3,4,5], n = 2  
**输出:** [1,2,3,5]

### 示例 2：

**输入:** head = [1], n = 1  
**输出:** []

### 示例 3：

**输入:** head = [1, 2], n = 1  
**输出:** [1]

### 提示：

- 链表中结点的数目为  $sz$
  - $1 \leq sz \leq 30$
  - $0 \leq Node.val \leq 100$

- $1 \leq n \leq sz$

进阶：你能尝试使用一趟扫描实现吗？

我们直接看解法代码：

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
        ListNode x = findFromEnd(dummy, n + 1);
        // 删掉倒数第 n 个节点
        x.next = x.next.next;
        return dummy.next;
    }

    private ListNode findFromEnd(ListNode head, int k) {
        // 代码见上文
    }
}
```

### ▶ 🎃 代码可视化动画🎃

这个逻辑就很简单了，要删除倒数第  $n$  个节点，就得获得倒数第  $n + 1$  个节点的引用，可以用我们实现的 `findFromEnd` 来操作。

不过注意我们又使用了虚拟头结点的技巧，也是为了防止出现空指针的情况，比如说链表总共有 5 个节点，题目就让你删除倒数第 5 个节点，也就是第一个节点，那按照算法逻辑，应该首先找到倒数第 6 个节点。但第一个节点前面已经没有节点了，这就会出错。

但有了我们虚拟节点 `dummy` 的存在，就避免了这个问题，能够对这种情况进行正确的删除。

## 单链表的中点

力扣第 876 题「链表的中间结点」就是这个题目，问题的关键也在于我们无法直接得到单链表的长度  $n$ ，常规方法也是先遍历链表计算  $n$ ，再遍历一次得到第  $n / 2$  个节点，也就是中间节点。

如果想一次遍历就得到中间节点，也需要耍点小聪明，使用「快慢指针」的技巧：

我们让两个指针 `slow` 和 `fast` 分别指向链表头结点 `head`。

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步，这样，当 `fast` 走到链表末尾时，`slow` 就指向了链表中点。

上述思路的代码实现如下：

```
class Solution {
    public ListNode middleNode(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

```
        fast = fast.next.next;
    }
    // 慢指针指向中点
    return slow;
}
}
```

---

▶ 🌟 代码可视化动画🌟

---

需要注意的是，如果链表长度为偶数，也就是说中点有两个的时候，我们这个解法返回的节点是靠后的那个节点。

另外，这段代码稍加修改就可以直接用到判断链表成环的算法题上。

## 判断链表是否包含环

判断链表是否包含环属于经典问题了，解决方案也是用快慢指针：

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。

如果 `fast` 最终能正常走到链表末尾，说明链表中没有环；如果 `fast` 走着走着竟然和 `slow` 相遇了，那肯定是 `fast` 在链表中转圈了，说明链表中含有环。

只需要把寻找链表中点的代码稍加修改就行了：

```
class Solution {
    public boolean hasCycle(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
            // 快慢指针相遇，说明含有环
            if (slow == fast) {
                return true;
            }
        }
        // 不包含环
        return false;
    }
}
```

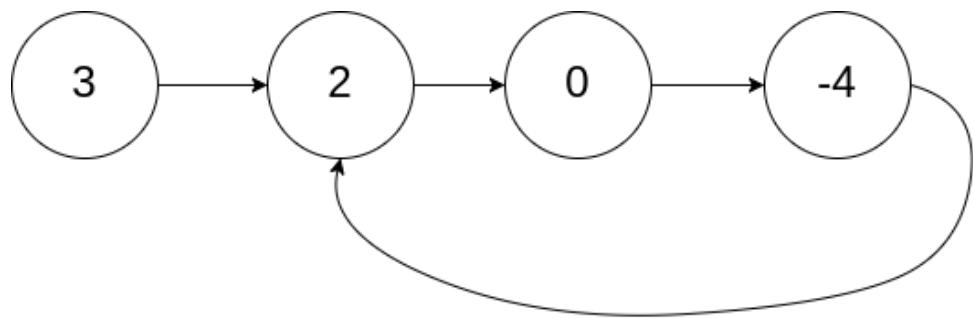
---

▶ 🎨 代码可视化动画🎨

---

当然，这个问题还有进阶版，也是力扣第 142 题「环形链表 II」：如果链表中含有环，如何计算这个环的起点？

为了避免读者迷惑，举个例子，环的起点是指下面这幅图中的节点 2：



这里先直接看一下寻找环起点的解法代码：

```
class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        // 上面的代码类似 hasCycle 函数
        if (fast == null || fast.next == null) {
            // fast 遇到空指针说明没有环
            return null;
        }

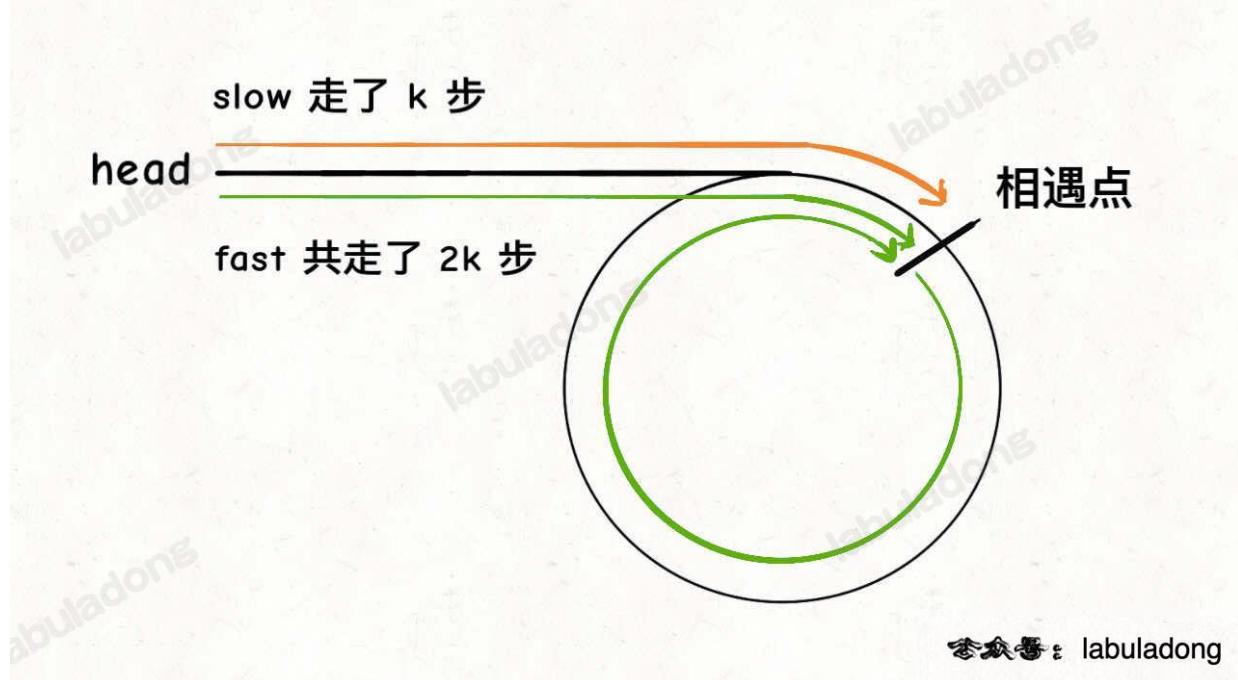
        // 重新指向头结点
        slow = head;
        // 快慢指针同步前进，相交点就是环起点
        while (slow != fast) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

### ▶ 🎨 代码可视化动画🎨

可以看到，当快慢指针相遇时，让其中一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。

为什么要这样呢？这里简单说一下其中的原理。

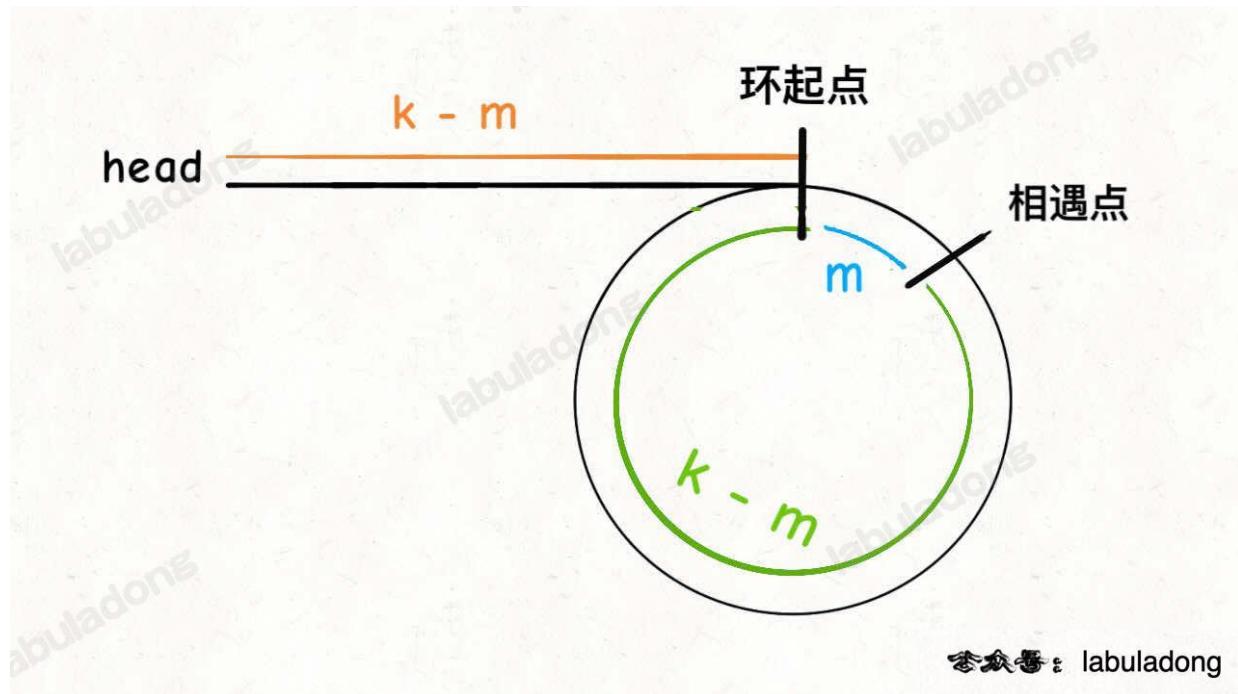
我们假设快慢指针相遇时，慢指针 `slow` 走了  $k$  步，那么快指针 `fast` 一定走了  $2k$  步：



fast 一定比 slow 多走了  $k$  步，这多走的  $k$  步其实就是 fast 指针在环里转圈圈，所以  $k$  的值就是环长度的「整数倍」。

假设相遇点距环的起点的距离为  $m$ ，那么结合上图的 slow 指针，环的起点距头结点 head 的距离为  $k - m$ ，也就是说如果从 head 前进  $k - m$  步就能到达环起点。

巧的是，如果从相遇点继续前进  $k - m$  步，也恰好到达环起点。因为结合上图的 fast 指针，从相遇点开始走  $k$  步可以转回到相遇点，那走  $k - m$  步肯定就走到环起点了：



所以，只要我们把快慢指针中的任一个重新指向 head，然后两个指针同速前进， $k - m$  步后一定会相遇，相遇之处就是环的起点了。

## 两个链表是否相交

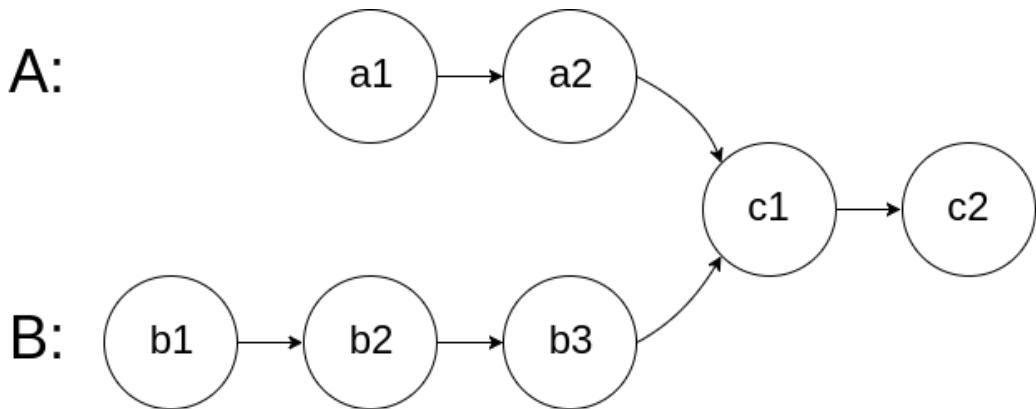
这个问题有意思，也是力扣第 160 题「相交链表」函数签名如下：

```
ListNode getIntersectionNode(ListNode headA, ListNode headB);
```

给你输入两个链表的头结点 `headA` 和 `headB`, 这两个链表可能存在相交。

如果相交, 你的算法应该返回相交的那个节点; 如果没相交, 则返回 `null`。

比如题目给我们举的例子, 如果输入的两个链表如下图:

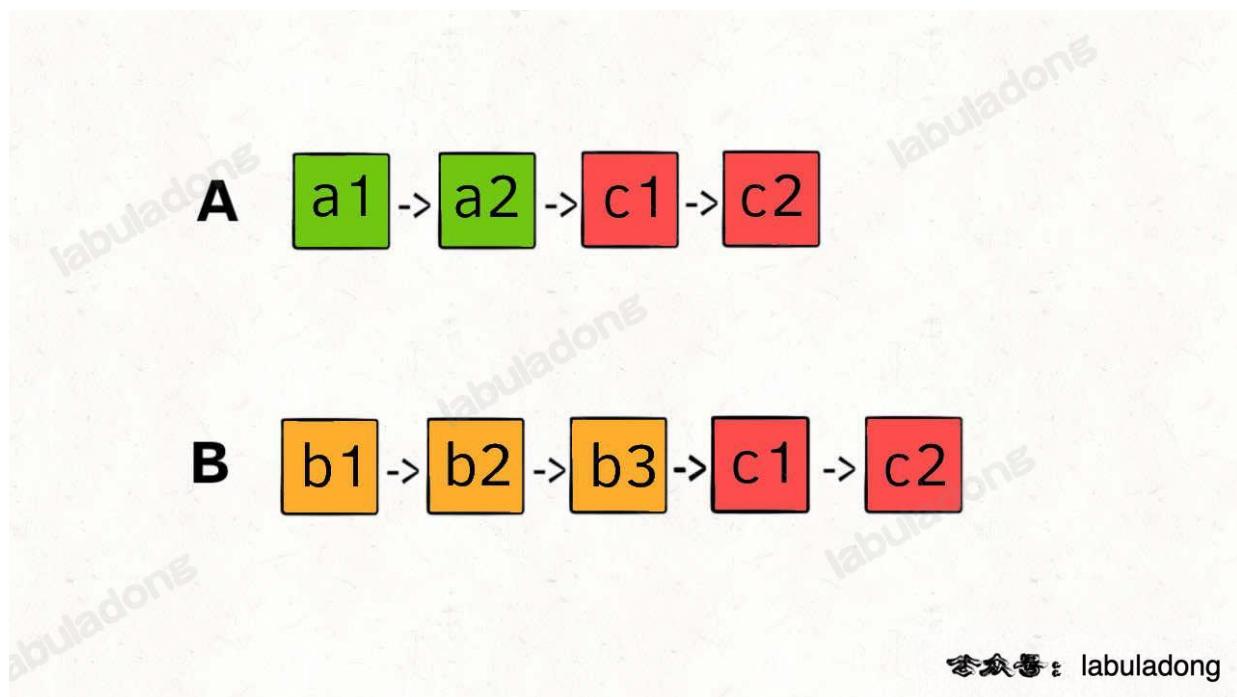


那么我们的算法应该返回 `c1` 这个节点。

这个题直接的想法可能是用 `HashSet` 记录一个链表的所有节点, 然后和另一条链表对比, 但这就需要额外的空间。

如果不额外的空间, 只使用两个指针, 你如何做呢?

难点在于, 由于两条链表的长度可能不同, 两条链表之间的节点无法对应:

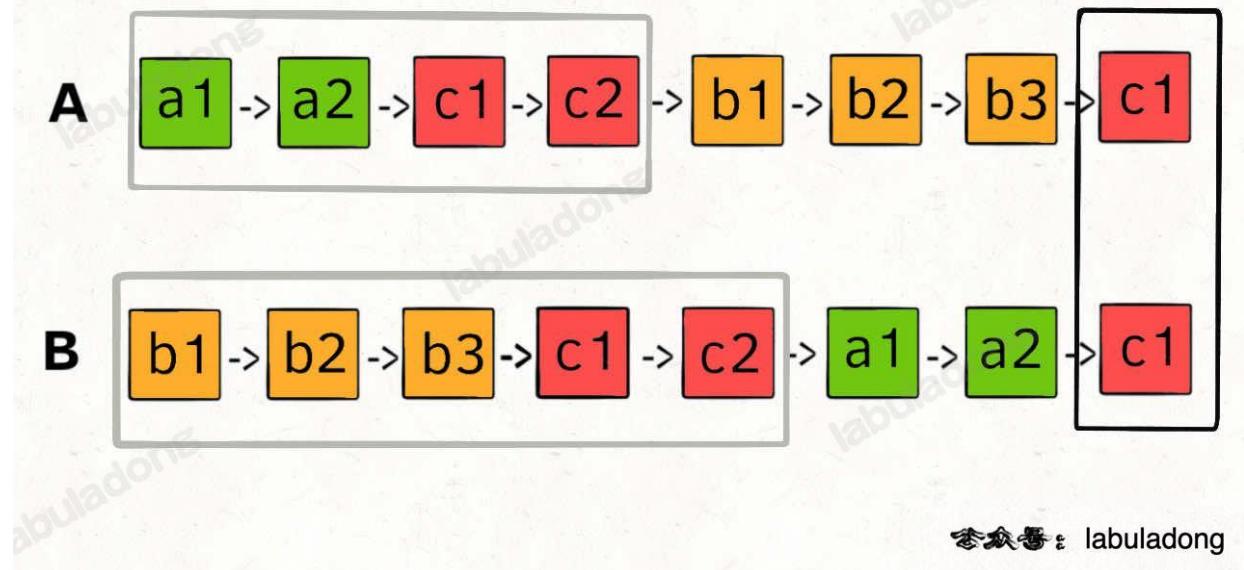


如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进, 并不能同时走到公共节点, 也就无法得到相交节点 `c1`。

解决这个问题的关键是, 通过某些方式, 让 `p1` 和 `p2` 能够同时到达相交节点 `c1`。

所以, 我们可以让 `p1` 遍历完链表 A 之后开始遍历链表 B, 让 `p2` 遍历完链表 B 之后开始遍历链表 A, 这样相当于「逻辑上」两条链表接在了一起。

如果这样进行拼接, 就可以让 `p1` 和 `p2` 同时进入公共部分, 也就是同时到达相交节点 `c1`:



✿✿✿✿ labuladong

那你可能会问，如果说两个链表没有相交点，是否能够正确的返回 null 呢？

这个逻辑可以覆盖这种情况的，相当于 c1 节点是 null 空指针嘛，可以正确返回 null。

按照这个思路，可以写出如下代码：

```
class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        // p1 指向 A 链表头结点, p2 指向 B 链表头结点
        ListNode p1 = headA, p2 = headB;
        while (p1 != p2) {
            // p1 走一步, 如果走到 A 链表末尾, 转到 B 链表
            if (p1 == null) {
                p1 = headB;
            } else {
                p1 = p1.next;
            }
            // p2 走一步, 如果走到 B 链表末尾, 转到 A 链表
            if (p2 == null) {
                p2 = headA;
            } else{
                p2 = p2.next;
            }
        }
        return p1;
    }
}
```

### ▶ 🎥 代码可视化动画

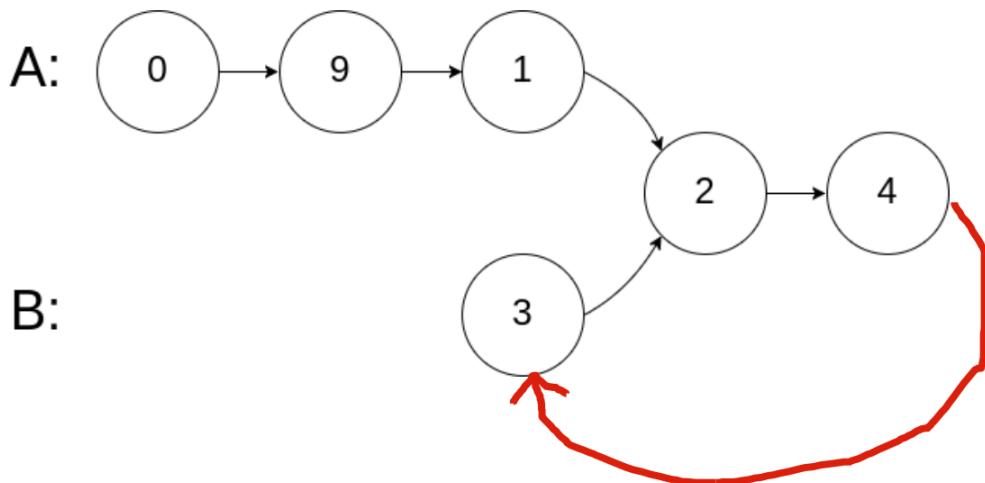
这样，这道题就解决了，空间复杂度为  $O(1)$ ，时间复杂度为  $O(N)$ 。

以上就是单链表的所有技巧，希望对你有启发。

2022/1/24 更新：

评论区有不少优秀读者对最后一题「寻找两条链表的交点」提出了一些其他思路，也补充到这里。

首先有读者提到，如果把两条链表首尾相连，那么「寻找两条链表的交点」的问题转换成了前面讲的「寻找环起点」的问题：



说实话我没有想到这种思路，不得不说这是一个很巧妙的转换！不过需要注意的是，这道题说不让你改变原始链表的结构，所以你把题目输入的链表转化成环形链表求解之后记得还要改回来，否则无法通过。

另外，还有读者提到，既然「寻找两条链表的交点」的核心在于让 `p1` 和 `p2` 两个指针能够同时到达相交节点 `c1`，那么可以通过预先计算两条链表的长度来做到这一点，具体代码如下：

```
class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = 0, lenB = 0;
        // 计算两条链表的长度
        for (ListNode p1 = headA; p1 != null; p1 = p1.next) {
            lenA++;
        }
        for (ListNode p2 = headB; p2 != null; p2 = p2.next) {
            lenB++;
        }
        // 让 p1 和 p2 到达尾部的距离相同
        ListNode p1 = headA, p2 = headB;
        if (lenA > lenB) {
            for (int i = 0; i < lenA - lenB; i++) {
                p1 = p1.next;
            }
        } else {
            for (int i = 0; i < lenB - lenA; i++) {
                p2 = p2.next;
            }
        }
        // 看两个指针是否会相同，p1 == p2 时有两种情况：
        // 1、要么是两条链表不相交，他俩同时走到尾部空指针
        // 2、要么是两条链表相交，他俩走到两条链表的相交点
        while (p1 != p2) {
    
```

```

        p1 = p1.next;
        p2 = p2.next;
    }
    return p1;
}

```

虽然代码多一些，但是时间复杂度是还是  $O(N)$ ，而且会更容易理解一些。

总之，我的解法代码并不一定就是最优或者最正确的，鼓励大家在评论区多多提出自己的疑问和思考，我也很高兴和大家探讨更多的解题思路~

到这里，链表相关的双指针技巧就全部讲完了，这些技巧的更多扩展延伸见 [更多链表双指针经典习题](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">109. Convert Sorted List to Binary Search Tree</a>	109. 有序链表转换二叉搜索树	
<a href="#">1257. Smallest Common Region</a>	1257. 最小公共区域	
<a href="#">1650. Lowest Common Ancestor of a Binary Tree III</a>	1650. 二叉树的最近公共祖先 III	
<a href="#">1836. Remove Duplicates From an Unsorted Linked List</a>	1836. 从未排序的链表中移除重复元素	
<a href="#">2. Add Two Numbers</a>	2. 两数相加	
<a href="#">234. Palindrome Linked List</a>	234. 回文链表	
<a href="#">264. Ugly Number II</a>	264. 丑数 II	
<a href="#">313. Super Ugly Number</a>	313. 超级丑数	
<a href="#">355. Design Twitter</a>	355. 设计推特	
<a href="#">360. Sort Transformed Array</a>	360. 有序转化数组	
<a href="#">373. Find K Pairs with Smallest Sums</a>	373. 查找和最小的 K 对数字	
<a href="#">378. Kth Smallest Element in a Sorted Matrix</a>	378. 有序矩阵中第 K 小的元素	
<a href="#">431. Encode N-ary Tree to Binary Tree</a>	431. 将 N 叉树编码为二叉树	
<a href="#">88. Merge Sorted Array</a>	88. 合并两个有序数组	
<a href="#">97. Interleaving String</a>	97. 交错字符串	
<a href="#">977. Squares of a Sorted Array</a>	977. 有序数组的平方	
-	剑指 Offer 18. 删除链表的节点	
-	剑指 Offer 25. 合并两个排序的链表	
-	剑指 Offer 49. 丑数	
-	剑指 Offer 52. 两个链表的第一个公共节点	
-	剑指 Offer II 021. 删除链表的倒数第 n 个结点	

LeetCode	力扣	难度
-	剑指 Offer II 022. 链表中环的入口节点	🟡
-	剑指 Offer II 023. 两个链表的第一个重合节点	🟢
-	剑指 Offer II 027. 回文链表	🟢
-	剑指 Offer II 061. 和最小的 k 个数对	🟡
-	剑指 Offer II 078. 合并排序链表	🔴

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

## 【强化练习】链表双指针经典习题

阅读本文前，你需要先学习：

- 链表基础
- 单链表的双指针技巧汇总

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 单链表的花式反转方法汇总



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">25. Reverse Nodes in k-Group</a>	<a href="#">25. K 个一组翻转链表</a>	
<a href="#">92. Reverse Linked List II</a>	<a href="#">92. 反转链表 II</a>	
<a href="#">206. Reverse Linked List</a>	<a href="#">206. 反转链表</a>	

-----  
反转单链表的迭代解法不是一个困难的事情，但是递归实现就有点难度了。如果再加一点难度，让你仅仅反转单链表中的一部分，你是否能够同时用迭代和递归实现呢？再进一步，如果让你  $k$  个一组反转链表，阁下又应如何应对？

本文就来由浅入深，一次性解决这些链表操作的问题。我会同时使用递归和迭代的方式，并结合可视化面板帮助你理解，以此强化你的递归思维以及操作链表指针的能力。

## 反转整个单链表

在 力扣/LeetCode 中，单链表的通用结构是这样的：

```
// 单链表节点的结构
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

单链表反转是一个比较基础的算法题，力扣第 206 题「反转链表」就是这个问题：

下面我们来尝试用多种方法解决这个问题。

## 迭代解法

这道题的常规做法就是迭代解法，通过操作几个指针，将链表中的每个节点的指针方向反转，没什么难点，主要是指针操作的细节问题。

这里直接给出代码，结合注释和可视化面板应该不难理解：

```
class Solution {
    // 反转以 head 为起点的单链表
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode pre = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = pre;
            pre = head;
            head = next;
        }
        return pre;
    }
}
```

```
        return head;
    }
    // 由于单链表的结构，至少要用三个指针才能完成迭代反转
    // cur 是当前遍历的节点，pre 是 cur 的前驱结点，nxt 是 cur 的后继结点
    ListNode pre, cur, nxt;
    pre = null; cur = head; nxt = head.next;
    while (cur != null) {
        // 逐个结点反转
        cur.next = pre;
        // 更新指针位置
        pre = cur;
        cur = nxt;
        if (nxt != null) {
            nxt = nxt.next;
        }
    }
    // 返回反转后的头结点
    return pre;
}
```

## ▶ 代码可视化动画

上面操作单链表的代码逻辑不复杂，而且也不止我这一种正确的写法。但是操作指针的时候，有一些很基本、很简单的小技巧，可以让你写代码的思路更清晰：

- 1、一旦出现类似 `nxt.next` 这种操作，就要条件反射地想到，先判断 `nxt` 是否为 `null`，否则容易出现空指针异常。
- 2、注意循环的终止条件。你要知道循环终止时，各个指针的位置，这样才能保返回正确的答案。如果你觉得有点复杂想不清楚，那就动手画一个最简单的场景跑一下算法，比如这道题就可以画一个只有两个节点的单链表 `1->2`，然后就能确定循环终止后各个指针的位置了。

## 递归解法

上面的迭代解法操作指针虽然有些繁琐，但是思路还是比较清晰的。如果现在让你用递归来反转单链表，你有啥想法没？

对于初学者来说可能很难想到，这很正常。如果你学习了后文的二叉树系列算法思维，回头再来看这道题，才有可能自己想出这个算法。

因为二叉树结构本身就是单链表的延伸，相当于二叉链表嘛，所以二叉树上的递归思维，套用到单链表上是一样的。

**递归反转单链表的关键在于，这个问题本身是存在子问题结构的。**

比方说，现在给你输入一个以 `1` 为头结点单链表 `1->2->3->4`，那么如果我忽略这个头结点 `1`，只拿出 `2->3->4` 这个子链表，它也是个单链表对吧？

那么你这个 `reverseList` 函数，只要输入一个单链表，就能给我反转对吧？那么你能不能用这个函数先来反转 `2->3->4` 这个子链表呢，然后再想办法把 `1` 接到反转后的 `4->3->2` 的最后面，是不是就完成了整个链表的反转？

```
reverseList(1->2->3->4) = reverseList(2->3->4) -> 1
```

这就是「分解问题」的思路，通过递归函数的定义，把原问题分解成若干规模更小、结构相同的子问题，最后通过子问题的答案组装原问题的解。

在后面的教程中会有专门的章节讲解和练习这种思维，这里不展开。

先来看看递归反转单链表的代码实现：

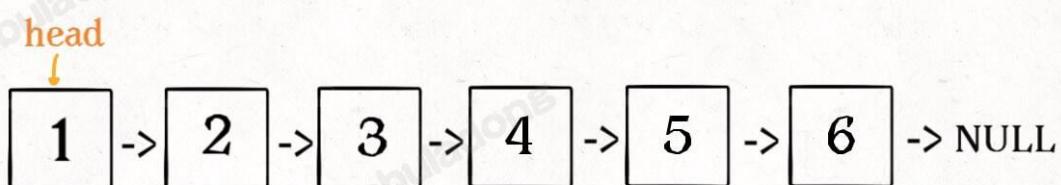
```
class Solution {
    // 定义：输入一个单链表头结点，将该链表反转，返回新的头结点
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode last = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return last;
    }
}
```

这个算法常常拿来显示递归的巧妙和优美，我们下面来详细解释一下这段代码，最后在给出可视化面板，你可以自己动手探究一下递归过程。

对于「分解问题」思路的递归算法，最重要的就是明确递归函数的定义。具体来说，我们的 `reverseList` 函数定义是这样的：

输入一个节点 `head`，将「以 `head` 为起点」的链表反转，并返回反转之后的头结点。

明白了函数的定义，再来看这个问题。比如说我们想反转这个链表：

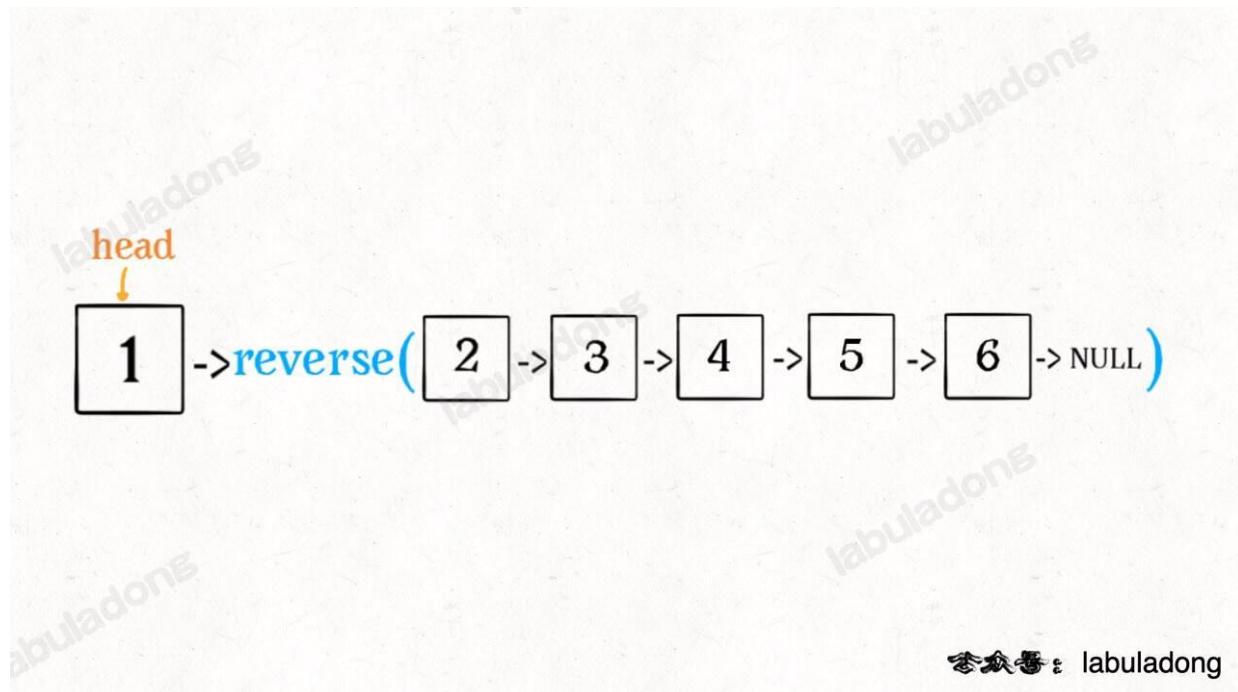


公众号：labuladong

那么输入 `reverseList(head)` 后，会在这里进行递归：

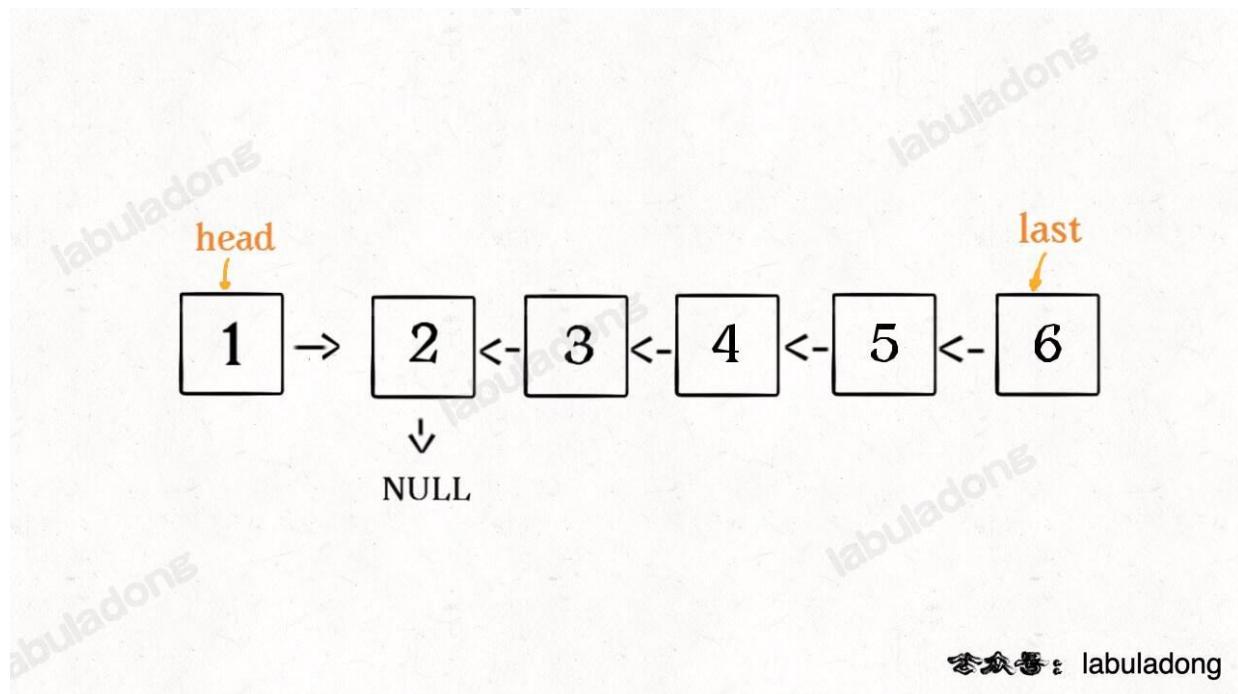
```
ListNode last = reverseList(head.next);
```

不要跳进递归（你的脑袋能压几个栈呀？），而是要根据刚才的函数定义，来弄清楚这段代码会产生什么结果：



© labuladong

这个 `reverseList(head.next)` 执行完成后，整个链表就成了这样：

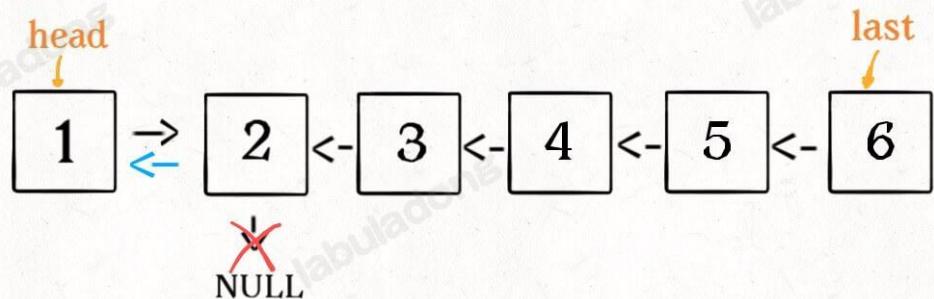


© labuladong

并且根据函数定义，`reverseList` 函数会返回反转之后的头结点，我们用变量 `last` 接收了。

现在再来看下面的代码：

```
head.next.next = head;
```



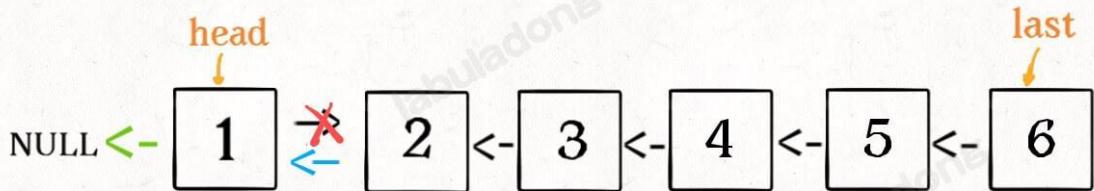
**head.next.next = head**

© labuladong

接下来：

```
head.next = null;  
return last;
```

**head.next = null**



© labuladong

神不神奇，这样整个链表就反转过来了！递归代码就是这么简洁优雅，不过其中有两个地方需要注意：

1、递归函数要有 base case，也就是这句：

```
if (head == null || head.next == null) {  
    return head;  
}
```

意思是如果链表为空或者只有一个节点的时候，反转结果就是它自己，直接返回即可。

2、当链表递归反转之后，新的头结点是 `last`，而之前的 `head` 变成了最后一个节点，别忘了链表的末尾要指向 `null`：

```
head.next = null;
```

这样，整个单链表就完成反转了，神不神奇？下面是递归反转链表的可视化过程：

▶ 🎃 代码可视化动画🎃

虽然可视化面板可以展示整个递归过程的所有细节，但我不建议初学者过于执着于细节。建议先依照上面图示讲解的思维方式理解递归，然后再通过可视化面板加深理解。

值得一提的是，递归操作链表并不高效。

递归解法和迭代解法相比，时间复杂度都是  $O(N)$ ，但是迭代解法的空间复杂度是  $O(1)$ ，而递归解法需要堆栈，空间复杂度是  $O(N)$ 。

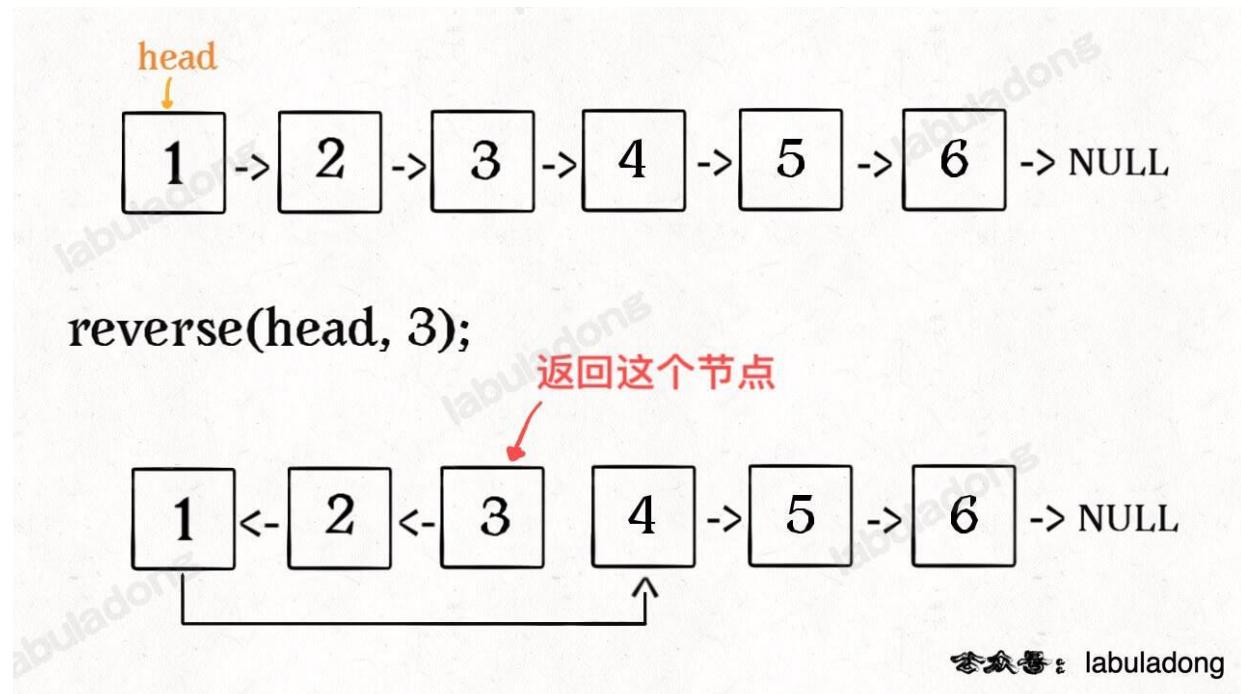
所以递归操作链表可以用来练习递归思维，但是考虑效率的话还是使用迭代算法更好。

## 反转链表前 N 个节点

这次我们实现一个这样的函数：

```
// 将链表的前 n 个节点反转 (n <= 链表长度)
ListNode reverseN(ListNode head, int n)
```

比如说对于下图链表，执行 `reverseN(head, 3)`：



本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 如何判断回文链表



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode

力扣

难度

[234. Palindrome Linked List](#)    [234. 回文链表](#)



阅读本文前，你需要先学习：

- 链表基础
- 链表双指针技巧
- 数组双指针技巧汇总

前文 [数组双指针技巧汇总](#) 讲解了回文串和回文序列相关的问题，先来简单回顾下。

寻找回文串的核心思想是从中心向两端扩展：

```
// 在 s 中寻找以 s[left] 和 s[right] 为中心的最长回文串
String palindrome(String s, int left, int right) {
    // 防止索引越界
    while (left >= 0 && right < s.length()
        && s.charAt(left) == s.charAt(right)) {
        // 双指针，向两边展开
        left--;
        right++;
    }
    // 返回以 s[left] 和 s[right] 为中心的最长回文串
    return s.substring(left + 1, right);
}
```

因为回文串长度可能为奇数也可能是偶数，长度为奇数时只存在一个中心点，而长度为偶数时存在两个中心点，所以上面这个函数需要传入 `l` 和 `r`。

而判断一个字符串是不是回文串就简单很多，不需要考虑奇偶情况，只需要[双指针技巧](#)，从两端向中间逼近即可：

```
boolean isPalindrome(String s) {
    // 一左一右两个指针相向而行
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

```
        }
        left++;
        right--;
    }
    return true;
}
```

以上代码很好理解吧，因为回文串是对称的，所以正着读和倒着读应该是一样的，这一特点是解决回文串问题的关键。

下面扩展这一最简单的情况，来解决：如何判断一个「单链表」是不是回文。

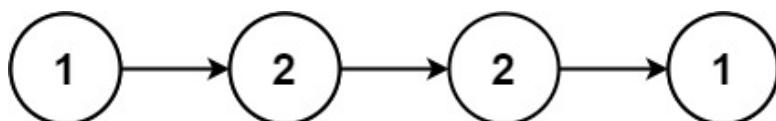
## 一、判断回文单链表

看下力扣第 234 题「回文链表」：

▼ 234. 回文链表 [Leetcode](#) | 力扣

给你一个单链表的头节点 `head`，请你判断该链表是否为回文链表。如果是，返回 `true`；否则，返回 `false`。

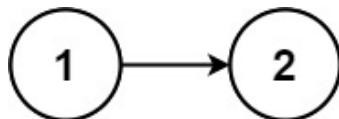
示例 1：



输入：`head = [1,2,2,1]`

输出：`true`

示例 2：



输入：`head = [1,2]`

输出：`false`

提示：

- 链表中节点数目在范围  $[1, 10^5]$  内
- $0 \leq \text{Node.val} \leq 9$

进阶：你能否用  $O(n)$  时间复杂度和  $O(1)$  空间复杂度解决此题？

函数签名如下：

```
boolean isPalindrome(ListNode head);
```

这道题的关键在于，单链表无法倒着遍历，无法使用双指针技巧。

那么最简单的办法就是，把原始链表反转存入一条新的链表，然后比较这两条链表是否相同。关于如何反转链表，可以参见前文 [递归翻转链表的一部分](#)。

我在 [学习数据结构的框架思维](#) 中说过，链表兼具递归结构，树结构不过是链表的衍生。那么，[链表其实也可以有前序遍历和后序遍历，借助二叉树后序遍历的思路，不需要显式反转原始链表也可以倒序遍历链表](#)：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    // 前序遍历代码
    traverse(root.left);
    // 中序遍历代码
    traverse(root.right);
    // 后序遍历代码
}

// 递归遍历单链表
void traverse(ListNode head) {
    // 前序遍历代码
    traverse(head.next);
    // 后序遍历代码
}
```

这个框架有什么指导意义呢？如果我想正序打印链表中的 `val` 值，可以在前序遍历位置写代码；反之，如果想倒序遍历链表，就可以在后序遍历位置操作：

```
// 倒序打印单链表中的元素值
void traverse(ListNode head) {
    if (head == null) return;
    traverse(head.next);
    // 后序遍历代码
    print(head.val);
}
```

说到这了，其实可以稍作修改，模仿双指针实现回文判断的功能：

```
class Solution {
    // 从左向右移动的指针
    ListNode left;
    // 从右向左移动的指针
    ListNode right;

    // 记录链表是否为回文
    boolean res = true;

    boolean isPalindrome(ListNode head) {
        left = head;
        traverse(head);
        return res;
    }

    void traverse(ListNode right) {
        if (right == null) {
            return;
        }
        if (left.val != right.val) {
            res = false;
        }
        left = left.next;
        traverse(right.next);
    }
}
```

```
}

// 利用递归，走到链表尾部
traverse(right.next);

// 后序遍历位置，此时的 right 指针指向链表右侧尾部
// 所以可以和 left 指针比较，判断是否是回文链表
if (left.val != right.val) {
    res = false;
}
left = left.next;
}
}
```

这么做的核心逻辑是什么呢？实际上就是把链表节点放入一个栈，然后再拿出来，这时候元素顺序就是反的，只不过我们利用的是递归函数的堆栈而已。如果不好理解，可以看下面这个可视化面板，你可以不断点击 `traverse(right.next);` 这一行代码，就能看出 `left`, `right` 的移动过程了：

#### ▶ 🎥 代码可视化动画 🎥

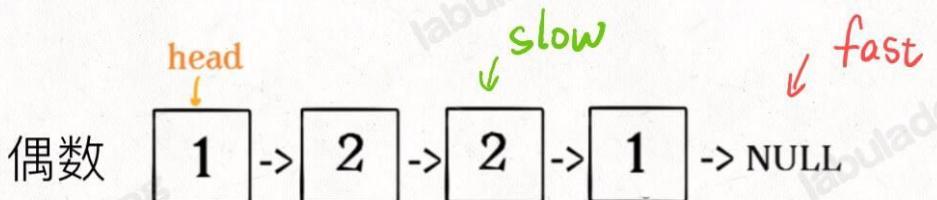
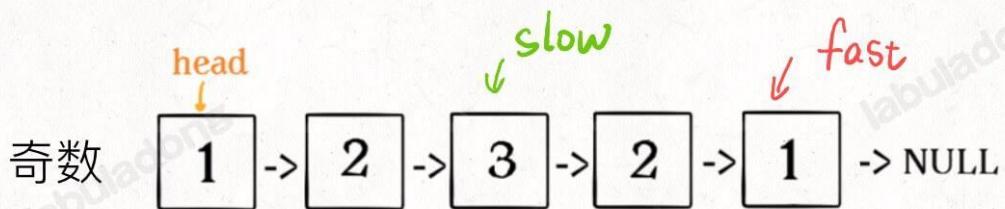
当然，无论造一条反转链表还是利用后序遍历，算法的时间和空间复杂度都是  $O(N)$ 。下面我们想想，能不能不用额外的空间，解决这个问题呢？

## 二、优化空间复杂度

更好的思路是这样的：

### 1、先通过 [链表双指针技巧](#) 中的快慢指针来找到链表的中点：

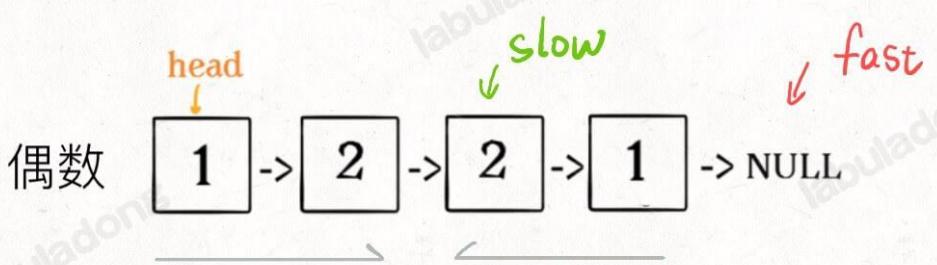
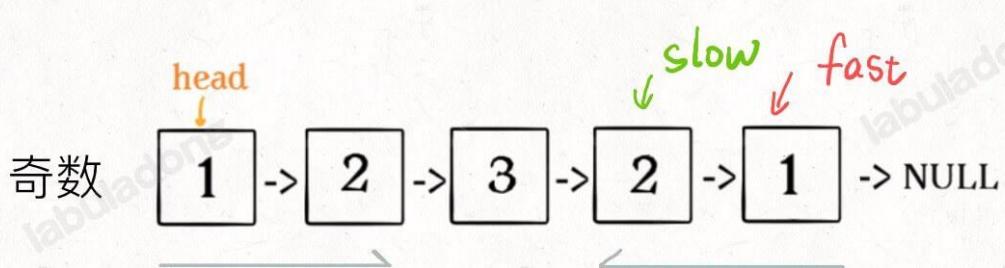
```
ListNode slow, fast;
slow = fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
// slow 指针现在指向链表中点
```



© labuladong

2、如果 fast 指针没有指向 null，说明链表长度为奇数， slow 还要再前进一步：

```
if (fast != null)
    slow = slow.next;
```



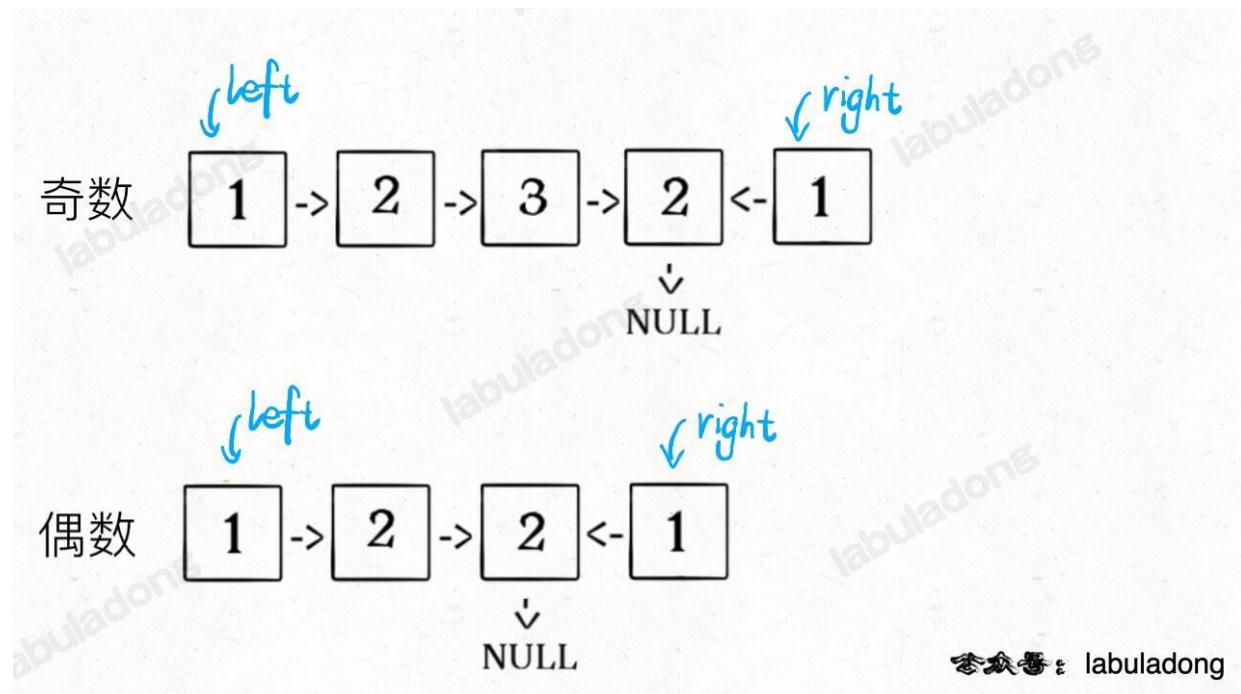
© labuladong

3、从 slow 开始反转后面的链表，现在就可以开始比较回文串了：

```
ListNode left = head;
ListNode right = reverse(slow);

while (right != null) {
    if (left.val != right.val)
        return false;
    left = left.next;
    right = right.next;
}
```

```
    left = left.next;
    right = right.next;
}
return true;
```



至此，把上面 3 段代码合在一起就高效地解决这个问题了，其中 `reverse` 函数可以参考 [翻转单链表](#)：

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode slow, fast;
        slow = fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        if (fast != null)
            slow = slow.next;

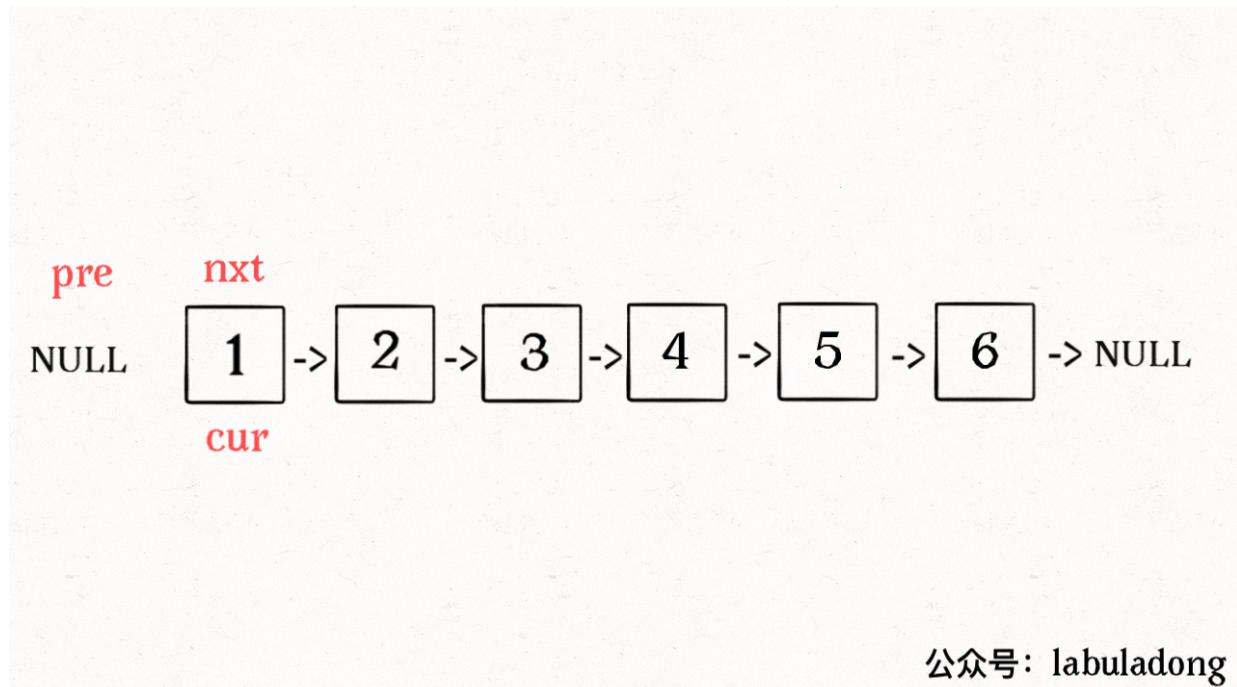
        ListNode left = head;
        ListNode right = reverse(slow);
        while (right != null) {
            if (left.val != right.val)
                return false;
            left = left.next;
            right = right.next;
        }

        return true;
    }

    ListNode reverse(ListNode head) {
        ListNode pre = null, cur = head;
        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
}
```

```
        pre = cur;
        cur = next;
    }
    return pre;
}
```

算法过程如下 GIF 所示：



---

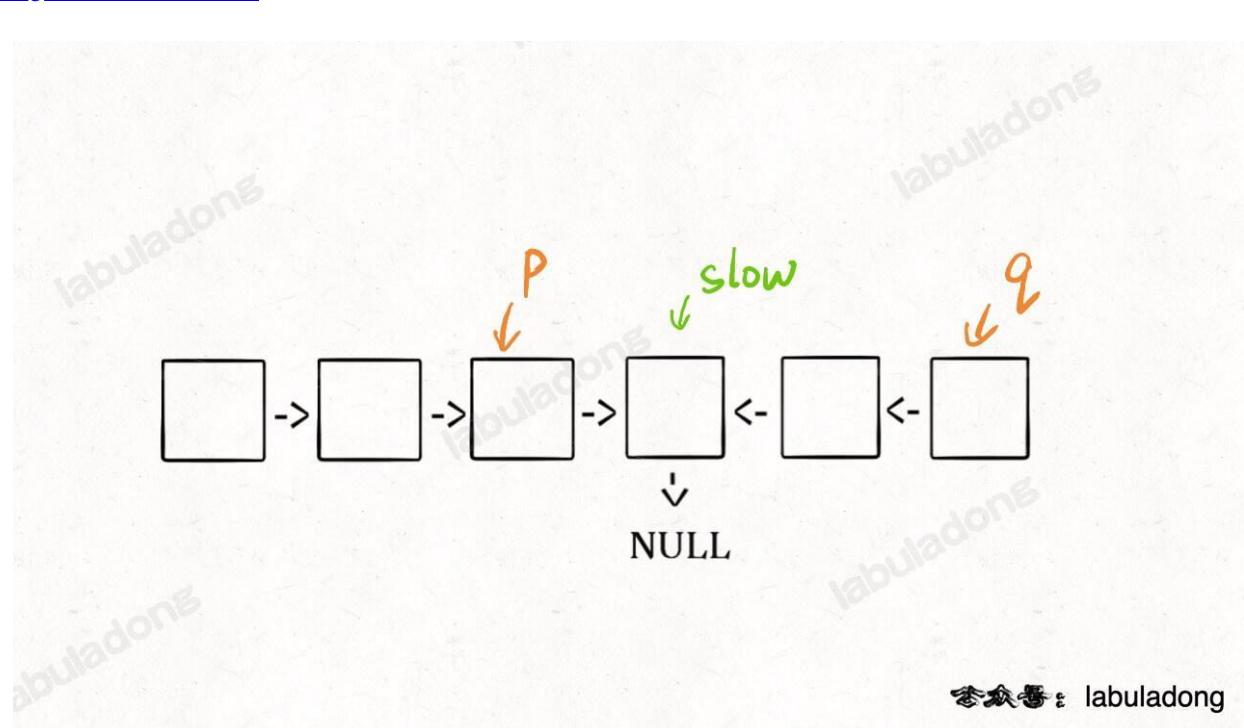
▶ 代码可视化动画

---

算法总体的时间复杂度  $O(N)$ ，空间复杂度  $O(1)$ ，已经是最优的了。

我知道肯定有读者会问：这种解法虽然高效，但破坏了输入链表的原始结构，能不能避免这个瑕疵呢？

其实这个问题很好解决，关键在于得到  $p$ ,  $q$  这两个指针位置：



© labuladong

这样，只要在函数 `return` 之前加一段代码即可恢复原先链表顺序：

```
p.next = reverse(q);
```

篇幅所限，我就不写了，读者可以自己尝试一下。

### 三、最后总结

首先，寻找回文串是从中间向两端扩展，判断回文串是从两端向中间收缩。对于单链表，无法直接倒序遍历，可以造一条新的反转链表，可以利用链表的后序遍历，也可以用栈结构倒序处理单链表。

具体到回文链表的判断问题，由于回文的特殊性，可以不完全反转链表，而是仅仅反转部分链表，将空间复杂度降到  $O(1)$ 。

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer II 027. 回文链表</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 双指针技巧秒杀七道数组题目



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">26. Remove Duplicates from Sorted Array</a>	26. 删除有序数组中的重复项	
-	剑指 Offer II 006. 排序数组中两个数字之和	
<a href="#">283. Move Zeroes</a>	283. 移动零	
<a href="#">167. Two Sum II - Input Array Is Sorted</a>	167. 两数之和 II - 输入有序数组	
<a href="#">344. Reverse String</a>	344. 反转字符串	
<a href="#">83. Remove Duplicates from Sorted List</a>	83. 删除排序链表中的重复元素	
-	剑指 Offer 57. 和为s的两个数字	
<a href="#">27. Remove Element</a>	27. 移除元素	
<a href="#">5. Longest Palindromic Substring</a>	5. 最长回文子串	

阅读本文前，你需要先学习：

- 数组基础
- 数组实现
- 单链表的六大解题套路

tip：本文有视频版：[数组双指针技巧汇总](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

在处理数组和链表相关问题时，双指针技巧是经常用到的，双指针技巧主要分为两类：**左右指针**和**快慢指针**。

所谓左右指针，就是两个指针相向而行或者相背而行；而所谓快慢指针，就是两个指针同向而行，一快一慢。

对于单链表来说，大部分技巧都属于快慢指针，[单链表的六大解题套路](#)都涵盖了，比如链表环判断，倒数第 K 个链表节点等问题，它们都是通过一个 **fast** 快指针和一个 **slow** 慢指针配合完成任务。

在数组中并没有真正意义上的指针，但我们可以把索引当做数组中的指针，这样也可以在数组中施展双指针技巧，**本文主要讲数组相关的双指针算法**。

## 一、快慢指针技巧

### 原地修改

数组问题中比较常见的快慢指针技巧，是让你原地修改数组。

比如说看下力扣第 26 题「删除有序数组中的重复项」，让你在有序数组去重：

▼ 26. 删除有序数组中的重复项 [Leetcode | 力扣](#)

给你一个 **非严格递增排列** 的数组 `nums`，请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次**，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致**。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 `k`，你需要做以下事情确保你的题解可以被通过：

- 更改数组 `nums`，使 `nums` 的前 `k` 个元素包含唯一元素，并按照它们最初在 `nums` 中出现的顺序排列。`nums` 的其余元素与 `nums` 的大小不重要。
- 返回 `k`。

**判题标准：**

系统会用下面的代码来测试你的题解：

```
int[] nums = [...]; // 输入数组
int[] expectedNums = [...]; // 长度正确的期望答案

int k = removeDuplicates(nums); // 调用

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有断言都通过，那么您的题解将被 **通过**。

**示例 1：**

**输入：** `nums = [1,1,2]`  
**输出：** `2, nums = [1,2,_]`

**解释：** 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度后面的元素。

**示例 2：**

**输入：** `nums = [0,0,1,1,1,2,2,3,3,4]`  
**输出：** `5, nums = [0,1,2,3,4]`

**解释：** 函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。不需要考虑数组中超出新长度后面的元素。

**提示：**

- `1 <= nums.length <= 3 * 104`
- `-104 <= nums[i] <= 104`
- `nums` 已按 **非严格递增** 排列

函数签名如下：

```
int removeDuplicates(int[] nums);
```

简单解释一下什么是原地修改：

如果不是原地修改的话，我们直接 new 一个 int[] 数组，把去重之后的元素放进这个新数组中，然后返回这个新数组即可。

但是现在题目让你原地删除，不允许 new 新数组，只能在原数组上操作，然后返回一个长度，这样就可以通过返回的长度和原始数组得到我们去重后的元素有哪些了。

由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难。但如果每找到一个重复元素就立即原地删除它，由于数组中删除元素涉及数据搬移，整个时间复杂度是会达到  $O(N^2)$ 。

高效解决这道题就要用到快慢指针技巧：

我们让慢指针 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赋值给 slow 并让 slow 前进一步。

这样，就保证了  $nums[0..slow]$  都是无重复的元素，当 fast 指针遍历完整个数组 nums 后， $nums[0..slow]$  就是整个数组去重之后的结果。

看代码：

```
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;
    }
}
```

算法执行的过程可以参考可视化面板：

---

▶ 🎨 代码可视化动画🎨

---

再简单扩展一下，看看力扣第 83 题「删除排序链表中的重复元素」，如果给你一个有序的单链表，如何去重呢？

其实和数组去重是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已，你对照着之前的代码来看：

```
class Solution {
    public 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;
}
```

算法执行的过程请看下面这个可视化面板：

▶  代码可视化动画 

这里可能有读者会问，链表中那些重复的元素并没有被删掉，就让这些节点在链表上挂着，合适吗？

这就要探讨不同语言的特性了，像 Java/Python 这类带有垃圾回收的语言，可以帮我们自动找到并回收这些「悬空」的链表节点的内存，而像 C++ 这类语言没有自动垃圾回收的机制，确实需要我们编写代码时手动释放掉这些节点的内存。

不过话说回来，就算法思维的培养来说，我们只需要知道这种快慢指针技巧即可。

除了让你在有序数组/链表中去重，题目还可能让你对数组中的某些元素进行「原地删除」。

比如力扣第 27 题「移除元素」，看下题目：

▼ 27. 移除元素 [Leetcode](#) | [力扣](#)

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`，要通过此题，您需要执行以下操作：

- 更改 `nums` 数组，使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。
- 返回 `k`。

用户评测：

评测机将使用以下代码测试您的解决方案：

```
int[] nums = [...]; // 输入数组
int val = ...; // 要移除的值
int[] expectedNums = [...]; // 长度正确的预期答案。
                            // 它以不等于 val 的值排序。

int k = removeElement(nums, val); // 调用你的实现

assert k == expectedNums.length;
```

```
sort(nums, 0, k); // 排序 nums 的前 k 个元素
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有的断言都通过，你的解决方案将会 **通过**。

### 示例 1：

```
输入: nums = [3,2,2,3], val = 3
输出: 2, nums = [2,2,_,_]
解释: 你的函数函数应该返回 k = 2，并且 nums 中的前两个元素均为 2。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

### 示例 2：

```
输入: nums = [0,1,2,2,3,0,4,2], val = 2
输出: 5, nums = [0,1,4,0,3,_,_,_]
解释: 你的函数应该返回 k = 5，并且 nums 中的前五个元素为 0,0,1,3,4。
注意这五个元素可以任意顺序返回。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

### 提示：

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 50$
- $0 \leq \text{val} \leq 100$

```
// 函数签名如下
int removeElement(int[] nums, int val);
```

题目要求我们把 `nums` 中所有值为 `val` 的元素原地删除，依然需要使用快慢指针技巧：

如果 `fast` 遇到值为 `val` 的元素，则直接跳过，否则就赋值给 `slow` 指针，并让 `slow` 前进一步。

这和前面说到的数组去重问题解法思路是完全一样的，直接看代码：

```
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;
    }
}
```

算法执行的过程可以参考可视化面板：

### ▶ 🎃 代码可视化动画🎃

注意这里和有序数组去重的解法有一个细节差异，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`。

实现了这个 `removeElement` 函数，接下来看看力扣第 283 题「移动零」：

给你输入一个数组 `nums`，请你原地修改，将数组中的所有值为 0 的元素移到数组末尾，函数签名如下：

```
void moveZeroes(int[] nums);
```

比如说给你输入 `nums = [0,1,4,0,2]`，你的算法没有返回值，但是会把 `nums` 数组原地修改成 `[1,4,2,0,0]`。

结合之前说到的几个题目，你是否有已经有了答案呢？

稍微修改上一题中的 `removeElement` 函数就可以完成这道题，或者直接复用 `removeElement` 函数也可以。

题目让我们将所有 0 移到最后，其实就相当于移除 `nums` 中的所有 0，然后再把后面的元素都赋值为 0：

```
class Solution {
    public void moveZeroes(int[] nums) {
        // 去除 nums 中的所有 0，返回不含 0 的数组长度
        int p = removeElement(nums, 0);
        // 将 nums[p..] 的元素赋值为 0
        for (; p < nums.length; p++) {
            nums[p] = 0;
        }
    }

    public int removeElement(int[] nums, int val) {
        // 见上文代码实现
    }
}
```

### ▶ 🔎 代码可视化动画🔍

到这里，原地修改数组的这些题目就已经差不多了。

## 滑动窗口

数组中另一大类快慢指针的题目就是「滑动窗口算法」。我在另一篇文章 [滑动窗口算法核心框架详解](#) 给出了滑动窗口的代码框架：

```
// 滑动窗口算法框架伪码
int left = 0, right = 0;

while (right < nums.size()) {
    // 增大窗口
```

```
window.addLast(nums[right]);
right++;

while (window needs shrink) {
    // 缩小窗口
    window.removeFirst(nums[left]);
    left++;
}
}
```

具体的题目本文就不重复了，这里只强调滑动窗口算法的快慢指针特性：

`left` 指针在后，`right` 指针在前，两个指针中间的部分就是「窗口」，算法通过扩大和缩小「窗口」来解决某些问题。

## 二、左右指针的常用算法

### 二分查找

我在另一篇文章 [二分查找框架详解](#) 中有详细探讨二分搜索代码的细节问题，这里只写最简单的二分算法，旨在突出它的双指针特性：

```
int binarySearch(int[] nums, int target) {
    // 一左一右两个指针相向而行
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid - 1;
    }
    return -1;
}
```

### n 数之和

看下力扣第 167 题「两数之和 II」：

#### ▼ 167. 两数之和 II - 输入有序数组 [Leetcode | 力扣](#)

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 `1 <= index1 < index2 <= numbers.length`。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

#### 示例 1:

输入: numbers = [2, 7, 11, 15], target = 9

输出: [1, 2]

解释: 2 与 7 之和等于目标数 9。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 2。返回 [1, 2]。

### 示例 2:

输入: numbers = [2, 3, 4], target = 6

输出: [1, 3]

解释: 2 与 4 之和等于目标数 6。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 3。返回 [1, 3]。

### 示例 3:

输入: numbers = [-1, 0], target = -1

输出: [1, 2]

解释: -1 与 0 之和等于目标数 -1。因此 index<sub>1</sub> = 1, index<sub>2</sub> = 2。返回 [1, 2]。

### 提示:

- $2 \leq \text{numbers.length} \leq 3 * 10^4$
- $-1000 \leq \text{numbers}[i] \leq 1000$
- **numbers** 按非递减顺序排列
- $-1000 \leq \text{target} \leq 1000$
- 仅存在一个有效答案

只要数组有序，就应该想到双指针技巧。这道题的解法有点类似二分查找，通过调节 **left** 和 **right** 就可以调整 **sum** 的大小：

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        // 左一右两个指针相向而行
        int left = 0, right = numbers.length - 1;
        while (left < right) {
            int sum = numbers[left] + numbers[right];
            if (sum == target) {
                // 题目要求的索引是从 1 开始的
                return new int[]{left + 1, right + 1};
            } else if (sum < target) {
                // 让 sum 大一点
                left++;
            } else if (sum > target) {
                // 让 sum 小一点
                right--;
            }
        }
        return new int[]{-1, -1};
    }
}
```

我在另一篇文章 [一个函数秒杀所有 nSum 问题](#) 中也运用类似的左右指针技巧给出了 [nSum](#) 问题的一种通用思路，本质上利用的也是双指针技巧。

## 反转数组

一般编程语言都会提供 [reverse](#) 函数，其实这个函数的原理非常简单，力扣第 344 题「反转字符串」就是类似的需求，让你反转一个 [char\[\]](#) 类型的字符数组，我们直接看代码吧：

```
void reverseString(char[] s) {
    // 一左一右两个指针相向而行
    int left = 0, right = s.length - 1;
    while (left < right) {
        // 交换 s[left] 和 s[right]
        char temp = s[left];
        s[left] = s[right];
        s[right] = temp;
        left++;
        right--;
    }
}
```

关于数组翻转的更多进阶问题，可以参见 [二维数组的花式遍历](#)。

## 回文串判断

回文串就是正着读和反着读都一样的字符串。比如说字符串 [aba](#) 和 [abba](#) 都是回文串，因为它们对称，反过来还是和本身一样；反之，字符串 [abac](#) 就不是回文串。

现在你应该能感觉到回文串问题和左右指针肯定有密切的联系，比如让你判断一个字符串是不是回文串，你可以写出下面这段代码：

```
boolean isPalindrome(String s) {
    // 一左一右两个指针相向而行
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

那接下来我提升一点难度，给你一个字符串，让你用双指针技巧从中找出最长的回文串，你会做吗？

这就是力扣第 5 题「最长回文子串」：

### ▼ 5. 最长回文子串 [Leetcode | 力扣](#)

给你一个字符串 [s](#)，找到 [s](#) 中最长的回文子串。

示例 1：

```
输入: s = "babad"
输出: "bab"
解释: "aba" 同样是符合题意的答案。
```

## 示例 2:

```
输入: s = "cbbd"
输出: "bb"
```

### 提示:

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母组成

函数签名如下:

```
String longestPalindrome(String s);
```

找回文串的难点在于，回文串的的长度可能是奇数也可能是偶数，解决该问题的核心是从中心向两端扩散的双指针技巧。

如果回文串的长度为奇数，则它有一个中心字符；如果回文串的长度为偶数，则可以认为它有两个中心字符。所以我们可以在实现这样一个函数：

```
// 在 s 中寻找以 s[l] 和 s[r] 为中心的最长回文串
String palindrome(String s, int l, int r) {
    // 防止索引越界
    while (l >= 0 && r < s.length()
        && s.charAt(l) == s.charAt(r)) {
        // 双指针，向两边展开
        l--; r++;
    }
    // 返回以 s[l] 和 s[r] 为中心的最长回文串
    return s.substring(l + 1, r);
}
```

这样，如果输入相同的 `l` 和 `r`，就相当于寻找长度为奇数的回文串，如果输入相邻的 `l` 和 `r`，则相当于寻找长度为偶数的回文串。

那么回到最长回文串的问题，解法的大致思路就是：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串
    找到以 s[i] 和 s[i+1] 为中心的回文串
    更新答案
```

翻译成代码，就可以解决最长回文子串这个问题：

```

String longestPalindrome(String s) {
    String res = "";
    for (int i = 0; i < s.length(); i++) {
        // 以 s[i] 为中心的最长回文子串
        String s1 = palindrome(s, i, i);
        // 以 s[i] 和 s[i+1] 为中心的最长回文子串
        String s2 = palindrome(s, i, i + 1);
        // res = longest(res, s1, s2)
        res = res.length() > s1.length() ? res : s1;
        res = res.length() > s2.length() ? res : s2;
    }
    return res;
}

```

## ▶ 🌟 代码可视化动画🌟

你应该能发现最长回文子串使用的左右指针和之前题目的左右指针有一些不同：之前的左右指针都是从两端向中间相向而行，而回文子串问题则是让左右指针从中心向两端扩展。不过这种情况也就回文串这类问题会遇到，所以我也把它归为左右指针了。

到这里，数组相关的双指针技巧就全部讲完了，这些技巧的更多扩展延伸见 [更多数组双指针经典高频题](#)。

## ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1. Two Sum</a>	1. 两数之和	
<a href="#">125. Valid Palindrome</a>	125. 验证回文串	
<a href="#">131. Palindrome Partitioning</a>	131. 分割回文串	
<a href="#">267. Palindrome Permutation II</a>	267. 回文排列 II	
<a href="#">281. Zigzag Iterator</a>	281. 锯齿迭代器	
<a href="#">42. Trapping Rain Water</a>	42. 接雨水	
<a href="#">543. Diameter of Binary Tree</a>	543. 二叉树的直径	
<a href="#">658. Find K Closest Elements</a>	658. 找到 K 个最接近的元素	
<a href="#">75. Sort Colors</a>	75. 颜色分类	
<a href="#">80. Remove Duplicates from Sorted Array II</a>	80. 删除有序数组中的重复项 II	
<a href="#">82. Remove Duplicates from Sorted List II</a>	82. 删除排序链表中的重复元素 II	
<a href="#">9. Palindrome Number</a>	9. 回文数	
-	剑指 Offer 21. 调整数组顺序使奇数位于偶数前面	
-	剑指 Offer 57. 和为s的两个数字	
-	剑指 Offer II 018. 有效的回文	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 二维数组的花式遍历技巧



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">48. Rotate Image</a>	<a href="#">48. 旋转图像</a>	
<a href="#">61. Rotate List</a>	<a href="#">61. 旋转链表</a>	
<a href="#">59. Spiral Matrix II</a>	<a href="#">59. 螺旋矩阵 II</a>	
<a href="#">151. Reverse Words in a String</a>	<a href="#">151. 反转字符串中的单词</a>	
<a href="#">54. Spiral Matrix</a>	<a href="#">54. 螺旋矩阵</a>	
-	<a href="#">剑指 Offer 29. 顺时针打印矩阵</a>	

阅读本文前，你需要先学习：

- [数组基础](#)
- [数组实现](#)

有些读者说，看了本站的很多文章，掌握了框架思维，可以解决大部分有套路框架可循的题目。

但是框架思维也不是万能的，有一些特定技巧呢，属于会者不难，难者不会的类型，只能通过多刷题进行总结和积累。

那么本文我分享一些巧妙的二维数组的花式操作，你只要有个印象，以后遇到类似题目就不会懵圈了。

## 顺/逆时针旋转矩阵

对二维数组进行旋转是常见的笔试题，力扣第 48 题「旋转图像」就是很经典的一道：

### ▼ 48. 旋转图像 [Leetcode](#) | [力扣](#)

给定一个  $n \times n$  的二维矩阵 `matrix` 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 **原地** 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

示例 1：

The diagram illustrates a 3x3 matrix rotation. On the left, a 3x3 grid contains the numbers 1, 2, 3 in the first row, 4, 5, 6 in the second, and 7, 8, 9 in the third. An arrow points to the right, where the same grid is shown rotated 90 degrees clockwise, with 7 at the top-left, 4 in the middle, 1 at the bottom-right, 8 in the middle, 5 in the middle, 2 at the bottom-right, 9 at the top-left, 6 in the middle, and 3 at the bottom-right.

1	2	3
4	5	6
7	8	9

7	4	1
8	5	2
9	6	3

```
输入: matrix = [[1,2,3],[4,5,6],[7,8,9]]  
输出: [[7,4,1],[8,5,2],[9,6,3]]
```

## 示例 2:

The diagram illustrates a 4x4 matrix rotation. On the left, a 4x4 grid contains the numbers 5, 1, 9, 11 in the first row, 2, 4, 8, 10 in the second, 13, 3, 6, 7 in the third, and 15, 14, 12, 16 in the fourth. An arrow points to the right, where the same grid is shown rotated 90 degrees clockwise, with 15 at the top-left, 13 in the middle, 2 at the bottom-right, 5 at the top-left, 14 in the middle, 3 at the bottom-right, 1 at the top-left, 12 at the top-left, 6 in the middle, 8 at the bottom-right, 9 at the top-left, 16 at the top-left, 7 in the middle, 10 at the bottom-right, and 11 at the top-left.

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

```
输入: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]  
输出: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
```

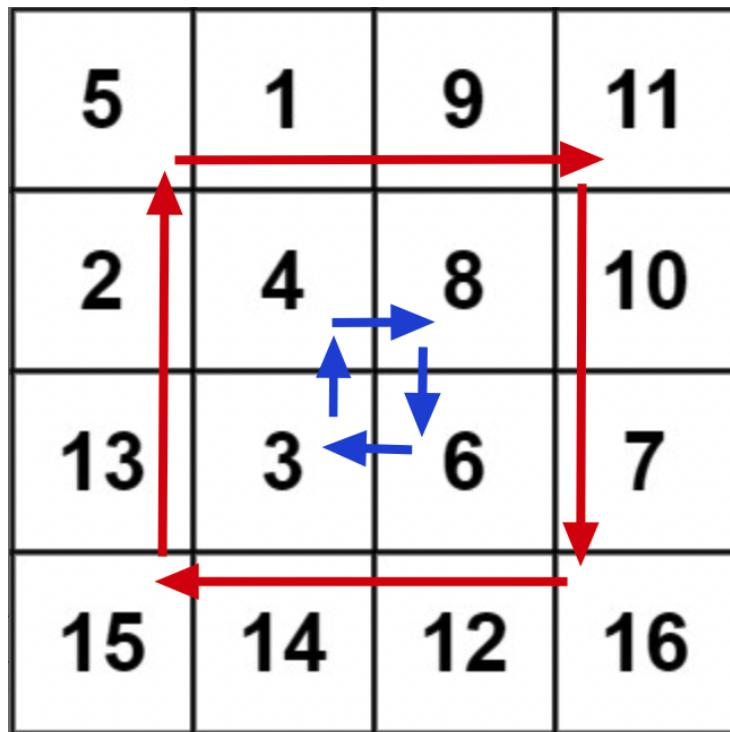
## 提示:

- `n == matrix.length == matrix[i].length`
- `1 <= n <= 20`
- `-1000 <= matrix[i][j] <= 1000`

题目很好理解，就是让你将一个二维矩阵顺时针旋转 90 度，难点在于要「原地」修改，函数签名如下：

```
void rotate(int[][] matrix)
```

如何「原地」旋转二维矩阵？稍想一下，感觉操作起来非常复杂，可能要设置巧妙的算法机制来「一圈一圈」旋转矩阵：



但实际上，这道题不能走寻常路，在讲巧妙解法之前，我们先看另一道谷歌曾经考过的算法题热热身：

给你一个包含若干单词和空格的字符串 `s`，请你写一个算法，原地反转所有单词的顺序。

比如说，给你输入这样一个字符串：

```
s = "hello world labuladong"
```

你的算法需要原地反转这个字符串中的单词顺序：

```
s = "labuladong world hello"
```

常规的方式是把 `s` 按空格 `split` 成若干单词，然后 `reverse` 这些单词的顺序，最后把这些单词 `join` 成句子。但这种方式使用了额外的空间，并不是「原地反转」单词。

正确的做法是，先将整个字符串 `s` 反转：

```
s = "gnodalubal dlrow olleh"
```

然后将每个单词分别反转：

```
s = "labuladong world hello"
```

这样，就实现了原地反转所有单词顺序的目的。力扣第 151 题「颠倒字符串中的单词」就是类似的问题，你可以顺便做一下。

上面这个小技巧还可以再包装包装，比如说你可以去看一下力扣第 61 题「旋转链表」：给你一个单链表，让你旋转链表，将链表每个节点向右移动 `k` 个位置。

比如说输入单链表  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ ,  $k = 2$ , 你的算法需要返回  $4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3$ , 即将链表每个节点向右移动 2 个位置。

这个题, 不要真傻乎乎地一个一个去移动链表节点, 我给你翻译翻译, 其实就是将链表的后  $k$  个节点移动到链表的头部嘛, 反应过来没有?

还没反应过来, 那再提示一下, 把后  $k$  个节点移动到链表的头部, 其实就是让你把链表的前  $n - k$  个节点和后  $k$  个节点原地翻转, 对不对?

这样, 是不是和前面说的原地翻转字符串中的单词是一样的道理呢? 你只需要先将整个链表反转, 然后将前  $n - k$  个节点和后  $k$  个节点分别反转, 就得到了结果。

当然, 这个题有一些小细节, 比如这个  $k$  可能大于链表的长度, 那么你需要先求出链表的长度  $n$ , 然后取模  $k = k \% n$ , 这样  $k$  就不会大于链表的长度, 且最后得到的结果也是正确的。

有时间的话自己去做一下这个题吧, 比较简单, 我这里就不贴代码了。

我讲上面这两道题的目的是什么呢?

旨在说明, 有时候咱们拍脑袋的常规思维, 在计算机看来可能并不是最优雅的; 但是计算机觉得最优雅的思维, 对咱们来说却并不那么直观。也许这就是算法的魅力所在吧。

本文为 [labuladong.online](#) 网站会员内容, 请 [点这里](#) 查看。

# 一个方法团灭 nSum 问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1. Two Sum</a>	1. 两数之和	●
<a href="#">18. 4Sum</a>	18. 四数之和	●
<a href="#">15. 3Sum</a>	15. 三数之和	●
<a href="#">167. Two Sum II - Input Array Is Sorted</a>	167. 两数之和 II - 输入有序数组	●

-----

阅读本文前，你需要先学习：

- [数组基础](#)
- [数组实现](#)

经常刷力扣的读者肯定知道鼎鼎有名的 `twoSum` 问题，不过除了 `twoSum` 问题，力扣上面还有 `3Sum`, `4Sum` 问题，以后如果想出个 `5Sum`, `6Sum` 也不是不可以。

总结来说，这类 `nSum` 问题就是给你输入一个数组 `nums` 和一个目标和 `target`，让你从 `nums` 选择 `n` 个数，使得这些数字之和为 `target`。

那么，对于这种问题有没有什么好办法用套路解决呢？本文就由浅入深，层层推进，用一个函数来解决所有 `nSum` 类型的问题。

提前说一下，对于本篇文章探讨的题目，使用 C++ 编写的代码更简洁易懂些，所以本文给出的都是 C++ 代码，你可以自行翻译成熟悉的语言。

## 一、twoSum 问题

我先来编一道 `twoSum` 题目：

如果假设输入一个数组 `nums` 和一个目标和 `target`，请你返回 `nums` 中能够凑出 `target` 的两个元素的值，比如输入 `nums = [1,3,5,6]`, `target = 9`，那么算法返回两个元素 `[3,6]`。可以假设只有且仅有一对儿元素可以凑出 `target`。

我们可以先对 `nums` 排序，然后利用前文 [双指针技巧](#) 写过的左右双指针技巧，从两端相向而行就行了：

```
int[] twoSum(int[] nums, int target) {
    // 先对数组排序
    Arrays.sort(nums);
```

```
// 左右指针
int lo = 0, hi = nums.length - 1;
while (lo < hi) {
    int sum = nums[lo] + nums[hi];
    // 根据 sum 和 target 的比较, 移动左右指针
    if (sum < target) {
        lo++;
    } else if (sum > target) {
        hi--;
    } else if (sum == target) {
        return new int[]{nums[lo], nums[hi]};
    }
}
return new int[]{};
```

这样就可以解决这个问题，力扣第 1 题「两数之和」和力扣第 167 题「两数之和 II - 输入有序数组」稍加修改就可以用类似的思路解决，我这里就不写了。

不过我要继续魔改题目，把这个题目变得更泛化，更困难一点：

**nums** 中可能有多对儿元素之和都等于 **target**，请你的算法返回所有和为 **target** 的元素对儿，其中不能出现重复。

函数签名如下：

```
List<List<Integer>> twoSumTarget(int[] nums, int target);
```

比如说输入为 **nums** = [1,3,1,2,2,3]，**target** = 4，那么算法返回的结果就是：[[1,3],[2,2]]（注意，我要求返回元素，而不是索引）。

对于修改后的问题，关键难点是现在可能有多个和为 **target** 的数对儿，还不能重复，比如上述例子中 [1,3] 和 [3,1] 就算重复，只能算一次。

首先，基本思路肯定还是排序加双指针：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】数组双指针经典习题

阅读本文前，你需要先学习：

- 数组基础
- 数组双指针技巧汇总

不要小看数组双指针技巧，它可以变出很多花样来。下面就来看看几个有趣的例题。

先看几道和 [数组双指针技巧汇总](#) 中例题类似的题目，巩固一下基础。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 小而美的算法技巧：前缀和数组



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">303. Range Sum Query - Immutable</a>	<a href="#">303. 区域和检索 - 数组不可变</a>	
<a href="#">304. Range Sum Query 2D - Immutable</a>	<a href="#">304. 二维区域和检索 - 矩阵不可变</a>	

阅读本文前，你需要先学习：

- [数组基础](#)
- [数组实现](#)

tip：本文有视频版：[前缀和/差分数组技巧精讲](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

前缀和技巧适用于快速、频繁地计算一个索引区间内的元素之和。

## 一维数组中的前缀和

先看一道例题，力扣第 303 题「区域和检索 - 数组不可变」，让你计算数组区间内元素的和，这是一道标准的前缀和问题：

### ▼ 303. 区域和检索 - 数组不可变 [Leetcode](#) | [力扣](#)

给定一个整数数组 `nums`，处理以下类型的多个查询：

1. 计算索引 `left` 和 `right`（包含 `left` 和 `right`）之间的 `nums` 元素的 **和**，其中 `left <= right`

实现 `NumArray` 类：

- `NumArray(int[] nums)` 使用数组 `nums` 初始化对象
- `int sumRange(int i, int j)` 返回数组 `nums` 中索引 `left` 和 `right` 之间的元素的 **总和**，包含 `left` 和 `right` 两点（也就是 `nums[left] + nums[left + 1] + ... + nums[right]`）

示例 1：

输入：

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

输出：

```
[null, 1, -1, -3]
```

**解释:**

```
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)
numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))
numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2 + (-1))
```

**提示:**

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^5 \leq \text{nums}[i] \leq 10^5$
- $0 \leq i \leq j < \text{nums.length}$
- 最多调用  $10^4$  次 `sumRange` 方法

```
// 题目要求你实现这样一个类
class NumArray {

    public NumArray(int[] nums) {}

    // 查询闭区间 [left, right] 的累加和
    public int sumRange(int left, int right) {}
}
```

`sumRange` 函数需要计算并返回一个索引区间之内的元素和，没学过前缀和的人可能写出如下代码：

```
class NumArray {

    private int[] nums;

    public NumArray(int[] nums) {
        this.nums = nums;
    }

    public int sumRange(int left, int right) {
        int res = 0;
        for (int i = left; i <= right; i++) {
            res += nums[i];
        }
        return res;
    }
}
```

这样，可以达到效果，但是效率很差，因为 `sumRange` 方法会被频繁调用，而它的最坏时间复杂度是  $O(N)$ ，其中  $N$  代表 `nums` 数组的长度。

这道题的最优解法是使用前缀和技巧，将 `sumRange` 函数的时间复杂度降为  $O(1)$ ，说白了就是不要在 `sumRange` 里面用 `for` 循环，咋整？

直接看代码实现：

```
class NumArray {
    // 前缀和数组
```

```

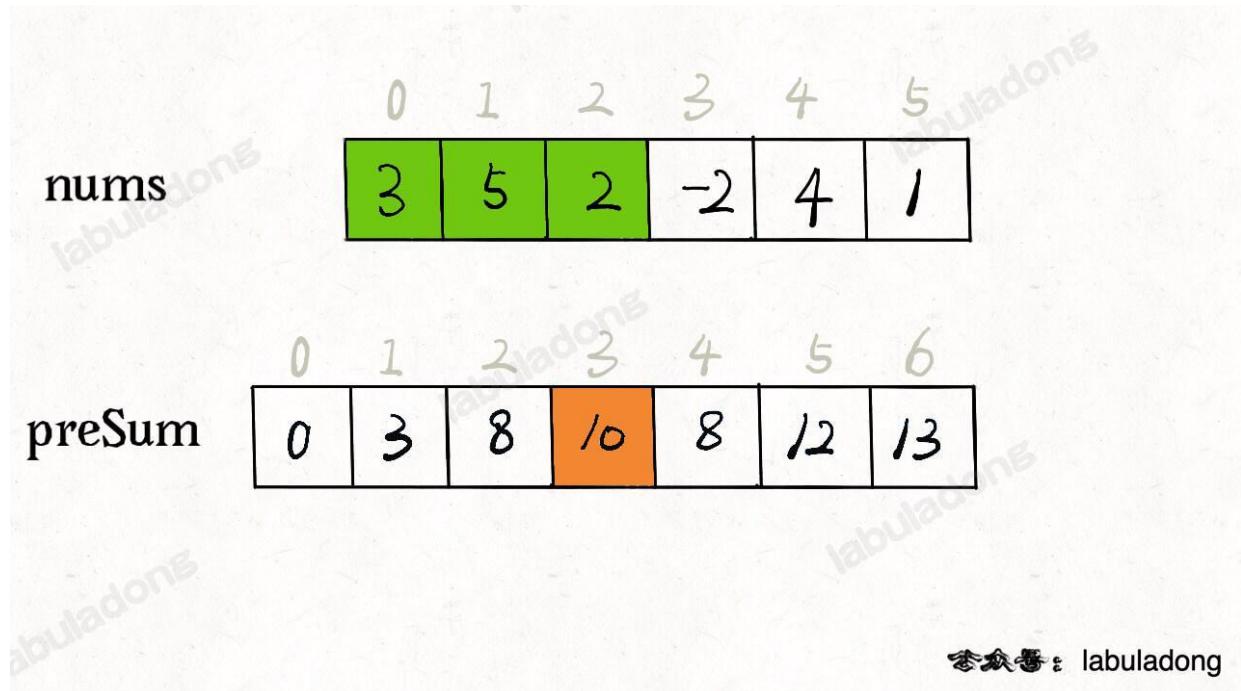
private int[] preSum;

// 输入一个数组，构造前缀和
public NumArray(int[] nums) {
    // preSum[0] = 0, 便于计算累加和
    preSum = new int[nums.length + 1];
    // 计算 nums 的累加和
    for (int i = 1; i < preSum.length; i++) {
        preSum[i] = preSum[i - 1] + nums[i - 1];
    }
}

// 查询闭区间 [left, right] 的累加和
public int sumRange(int left, int right) {
    return preSum[right + 1] - preSum[left];
}
}

```

核心思路是我们 new 一个新的数组 `preSum` 出来，`preSum[i]` 记录 `nums[0..i-1]` 的累加和，看图  $10 = 3 + 5 + 2$ ：



看这个 `preSum` 数组，如果我想求索引区间 `[1, 4]` 内的所有元素之和，就可以通过 `preSum[5] - preSum[1]` 得出。

这样，`sumRange` 函数仅仅需要做一次减法运算，避免了每次进行 `for` 循环调用，最坏时间复杂度为常数  $O(1)$ 。

这个技巧在生活中运用也挺广泛的，比方说，你们班上有若干同学，每个同学有一个期末考试的成绩（满分 100 分），那么请你实现一个 API，输入任意一个分数段，返回有多少同学的成绩在这个分数段内。

那么，你可以先通过计数排序的方式计算每个分数具体有多少个同学，然后利用前缀和技巧来实现分数段查询的 API：

```

// 存储着所有同学的分数
int[] scores;
// 试卷满分 100 分
int[] count = new int[100 + 1];
// 记录每个分数有几个同学
for (int score : scores)
    count[score]++;

```

```
// 构造前缀和
for (int i = 1; i < count.length; i++)
    count[i] = count[i] + count[i-1];
// 利用 count 这个前缀和数组进行分数段查询
```

接下来，我们看一看前缀和思路在二维数组中如何运用。

## 二维矩阵中的前缀和

这是力扣第 304 题「二维区域和检索 - 矩阵不可变」，其实和上一题类似，上一题是让你计算子数组的元素之和，这道题让你计算二维矩阵中子矩阵的元素之和：

### ▼ 304. 二维区域和检索 - 矩阵不可变 [Leetcode | 力扣](#)

给定一个二维矩阵 `matrix`，以下类型的多个请求：

- 计算其子矩形范围内元素的总和，该子矩阵的 **左上角** 为 (`row1, col1`)，**右下角** 为 (`row2, col2`)。

实现 `NumMatrix` 类：

- `NumMatrix(int[][] matrix)` 给定整数矩阵 `matrix` 进行初始化
- `int sumRegion(int row1, int col1, int row2, int col2)` 返回 **左上角** (`row1, col1`)、**右下角** (`row2, col2`) 所描述的子矩阵的元素 **总和**。

示例 1：

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

输入：

```
["NumMatrix","sumRegion","sumRegion","sumRegion"]
[[[[3,0,1,4,2],[5,6,3,2,1],[1,2,0,1,5],[4,1,0,1,7],[1,0,3,0,5]]],[2,1,4,3],
[1,1,2,2],[1,2,2,4]]
```

输出：

```
[null, 8, 11, 12]
```

解释：

```
NumMatrix numMatrix = new NumMatrix([[3,0,1,4,2],[5,6,3,2,1],[1,2,0,1,5],
[4,1,0,1,7],[1,0,3,0,5]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
```

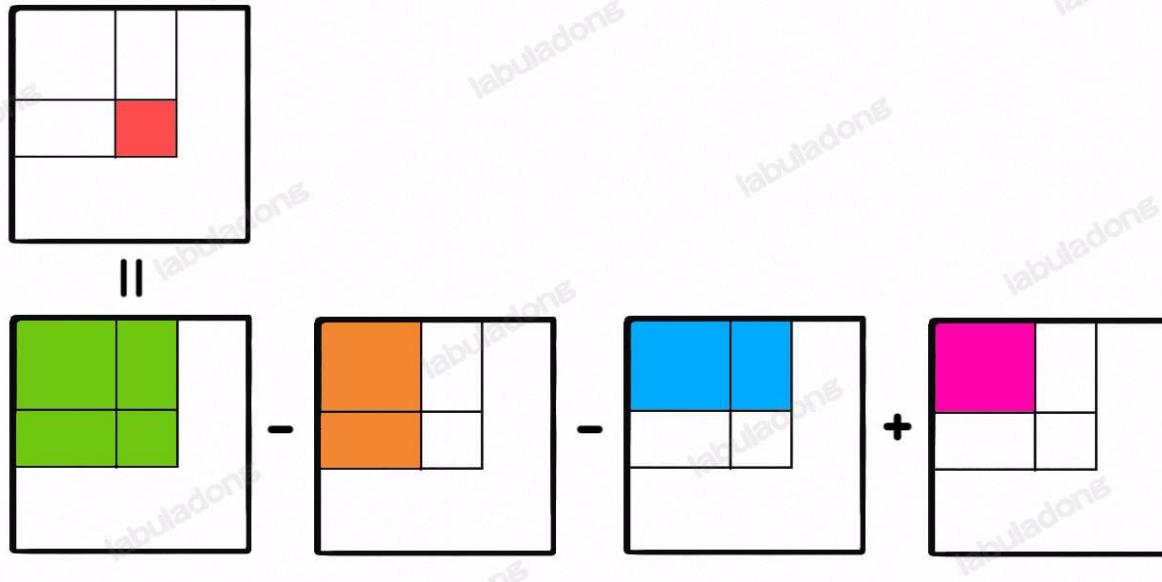
提示：

- `m == matrix.length`
- `n == matrix[i].length`

- $1 \leq m, n \leq 200$
- $-10^5 \leq \text{matrix}[i][j] \leq 10^5$
- $0 \leq \text{row}_1 \leq \text{row}_2 < m$
- $0 \leq \text{col}_1 \leq \text{col}_2 < n$
- 最多调用  $10^4$  次 `sumRegion` 方法

当然，你可以用一个嵌套 `for` 循环去遍历这个矩阵，但这样的话 `sumRegion` 函数的时间复杂度就高了，你算法的格局就低了。

注意任意子矩阵的元素和可以转化成它周边几个大矩阵的元素和的运算：



公众号： labuladong

而这四个大矩阵有一个共同的特点，就是左上角都是  $(0, 0)$  原点。

那么做这道题更好的思路和一维数组中的前缀和是非常类似的，我们可以维护一个二维 `preSum` 数组，专门记录以原点为顶点的矩阵的元素之和，就可以用几次加减运算算出任何一个子矩阵的元素和：

```
class NumMatrix {
    // 定义: preSum[i][j] 记录 matrix 中子矩阵 [0, 0, i-1, j-1] 的元素和
    private int[][] preSum;

    public NumMatrix(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        if (m == 0 || n == 0) return;
        // 构造前缀和矩阵
        preSum = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 计算每个矩阵 [0, 0, i, j] 的元素和
                preSum[i][j] = preSum[i-1][j] + preSum[i][j-1] + matrix[i - 1][j - 1] - preSum[i-1][j-1];
            }
        }
    }

    // 计算子矩阵 [x1, y1, x2, y2] 的元素和
    public int sumRegion(int x1, int y1, int x2, int y2) {
```

```
// 目标矩阵之和由四个相邻矩阵运算获得
return preSum[x2+1][y2+1] - preSum[x1][y2+1] - preSum[x2+1][y1] +
preSum[x1][y1];
}
}
```

这样，`sumRegion` 函数的时间复杂度也用前缀和技巧优化到了  $O(1)$ ，这是典型的「空间换时间」思路。

前缀和技巧就讲到这里，应该说这个算法技巧是会者不难难者不会，实际运用中还是要多培养自己的思维灵活性，做到一眼看出题目是一个前缀和问题。

除了本文举例的基本用法，前缀和数组经常和其他数据结构或算法技巧相结合，我会在 [前缀和技巧高频习题](#) 中举例讲解。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1314. Matrix Block Sum</a>	<a href="#">1314. 矩阵区域和</a>	
<a href="#">1352. Product of the Last K Numbers</a>	<a href="#">1352. 最后 K 个数的乘积</a>	
<a href="#">238. Product of Array Except Self</a>	<a href="#">238. 除自身以外数组的乘积</a>	
<a href="#">325. Maximum Size Subarray Sum Equals k</a>	<a href="#">325. 和等于 k 的最长子数组长度</a>	
<a href="#">327. Count of Range Sum</a>	<a href="#">327. 区间和的个数</a>	
<a href="#">437. Path Sum III</a>	<a href="#">437. 路径总和 III</a>	
<a href="#">523. Continuous Subarray Sum</a>	<a href="#">523. 连续的子数组和</a>	
<a href="#">525. Contiguous Array</a>	<a href="#">525. 连续数组</a>	
<a href="#">560. Subarray Sum Equals K</a>	<a href="#">560. 和为 K 的子数组</a>	
<a href="#">724. Find Pivot Index</a>	<a href="#">724. 寻找数组的中心下标</a>	
<a href="#">862. Shortest Subarray with Sum at Least K</a>	<a href="#">862. 和至少为 K 的最短子数组</a>	
<a href="#">918. Maximum Sum Circular Subarray</a>	<a href="#">918. 环形子数组的最大和</a>	
<a href="#">974. Subarray Sums Divisible by K</a>	<a href="#">974. 和可被 K 整除的子数组</a>	
-	<a href="#">剑指 Offer 57 - II. 和为s的连续正数序列</a>	
-	<a href="#">剑指 Offer II 010. 和为 k 的子数组</a>	
-	<a href="#">剑指 Offer II 011. 0 和 1 个数相同的子数组</a>	
-	<a href="#">剑指 Offer II 012. 左右两边子数组的和相等</a>	
-	<a href="#">剑指 Offer II 013. 二维子矩阵的和</a>	
-	<a href="#">剑指 Offer II 050. 向下的路径节点之和</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】前缀和技巧经典习题

阅读本文前，你需要先学习：

- 前缀和技巧
- 哈希表原理

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 小而美的算法技巧：差分数组



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

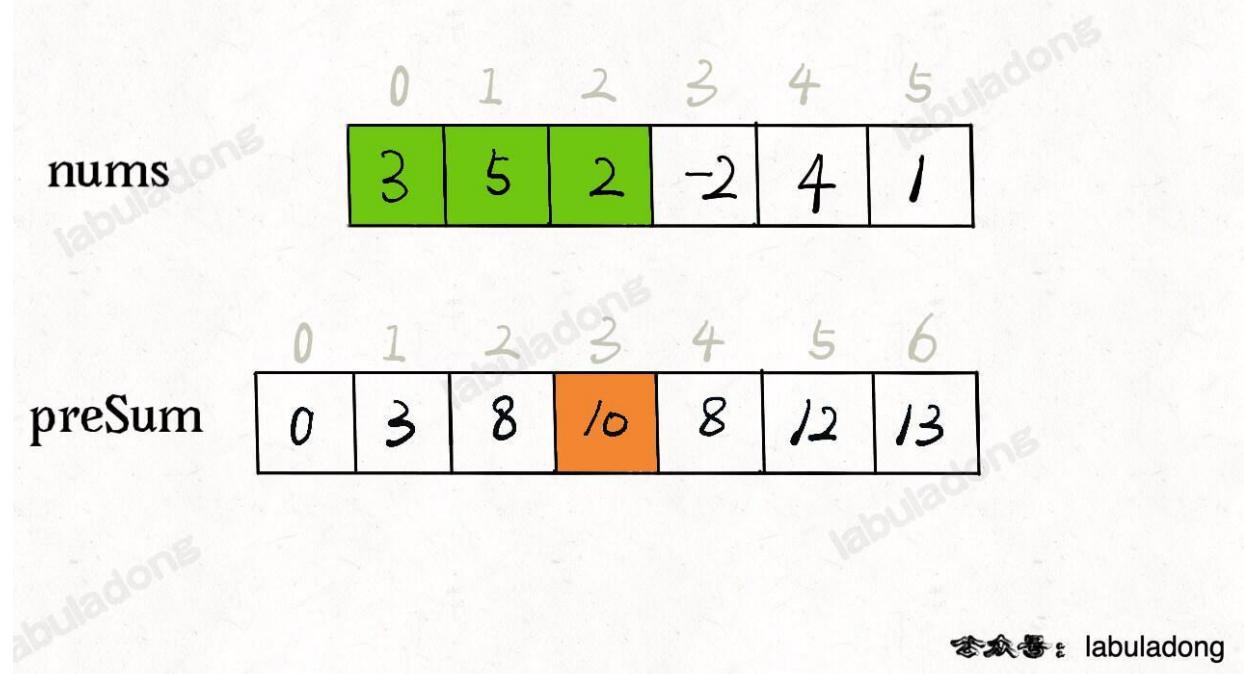
LeetCode	力扣	难度
<a href="#">370. Range Addition</a>	<a href="#">370. 区间加法</a>	简单
<a href="#">1109. Corporate Flight Bookings</a>	<a href="#">1109. 航班预订统计</a>	简单
<a href="#">1094. Car Pooling</a>	<a href="#">1094. 拼车</a>	简单

阅读本文前，你需要先学习：

- 数组基础
- 数组实现
- 前缀和技巧

前缀和技巧 主要适用的场景是原始数组不会被修改的情况下，频繁查询某个区间的累加和，核心代码就是下面这段：

```
class PrefixSum {  
    // 前缀和数组  
    private int[] preSum;  
  
    // 输入一个数组，构造前缀和  
    public PrefixSum(int[] nums) {  
        // preSum[0] = 0, 便于计算累加和  
        preSum = new int[nums.length + 1];  
        // 计算 nums 的累加和  
        for (int i = 1; i < preSum.length; i++) {  
            preSum[i] = preSum[i - 1] + nums[i - 1];  
        }  
    }  
  
    // 查询闭区间 [left, right] 的累加和  
    public int sumRange(int left, int right) {  
        return preSum[right + 1] - preSum[left];  
    }  
}
```



© labuladong

`preSum[i]` 就代表着 `nums[0..i-1]` 所有元素的累加和，如果我们想求区间 `nums[i..j]` 的累加和，只要计算 `preSum[j+1] - preSum[i]` 即可，而不需要遍历整个区间求和。

本文讲一个和前缀和思想非常类似的算法技巧「差分数组」，差分数组的主要适用场景是频繁对原始数组的某个区间的元素进行增减。

比如说，我给你输入一个数组 `nums`，然后又要求给区间 `nums[2..6]` 全部加 1，再给 `nums[3..9]` 全部减 3，再给 `nums[0..4]` 全部加 2，再给...

一通操作猛如虎，然后问你，最后 `nums` 数组的值是什么？

常规的思路很容易，你让我给区间 `nums[i..j]` 加上 `val`，那我就一个 `for` 循环给它们都加上呗，还能咋样？这种思路的时间复杂度是  $O(N)$ ，由于这个场景下对 `nums` 的修改非常频繁，所以效率会很低下。

这里就需要差分数组的技巧，类似前缀和技巧构造的 `preSum` 数组，我们先对 `nums` 数组构造一个 `diff` 差分数组，`diff[i]` 就是 `nums[i]` 和 `nums[i-1]` 之差：

```
int[] diff = new int[nums.length];
// 构造差分数组
diff[0] = nums[0];
for (int i = 1; i < nums.length; i++) {
    diff[i] = nums[i] - nums[i - 1];
}
```

**nums**

8	2	6	3	1
---	---	---	---	---

**diff**

8	-6	4	-3	-2
---	----	---	----	----

© labuladong

通过这个 **diff** 差分数组是可以反推出原始数组 **nums** 的，代码逻辑如下：

```
int[] res = new int[diff.length];
// 根据差分数组构造结果数组
res[0] = diff[0];
for (int i = 1; i < diff.length; i++) {
    res[i] = res[i - 1] + diff[i];
}
```

这样构造差分数组 **diff**，就可以快速进行区间增减的操作，如果你想对区间 **nums[i..j]** 的元素全部加 3，那么只需要让 **diff[i] += 3**，然后再让 **diff[j+1] -= 3** 即可：

**nums**

8	5	9	6	1
---	---	---	---	---

**diff**

8	-3	4	-3	-5
---	----	---	----	----

i

j

© labuladong

原理很简单，回想 **diff** 数组反推 **nums** 数组的过程，**diff[i] += 3** 意味着给 **nums[i..]** 所有的元素都加了 3，然后 **diff[j+1] -= 3** 又意味着对于 **nums[j+1..]** 所有元素再减 3，那综合起来，是不是就是对 **nums[i..j]** 中的所有元

素都加 3 了？

只要花费  $O(1)$  的时间修改 `diff` 数组，就相当于给 `nums` 的整个区间做了修改。多次修改 `diff`，然后通过 `diff` 数组反推，即可得到 `nums` 修改后的结果。

现在我们把差分数组抽象成一个类，包含 `increment` 方法和 `result` 方法：

```
// 差分数组工具类
class Difference {
    // 差分数组
    private int[] diff;

    // 输入一个初始数组，区间操作将在这个数组上进行
    public Difference(int[] nums) {
        assert nums.length > 0;
        diff = new int[nums.length];
        // 根据初始数组构造差分数组
        diff[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            diff[i] = nums[i] - nums[i - 1];
        }
    }

    // 给闭区间 [i, j] 增加 val (可以是负数)
    public void increment(int i, int j, int val) {
        diff[i] += val;
        if (j + 1 < diff.length) {
            diff[j + 1] -= val;
        }
    }

    // 返回结果数组
    public int[] result() {
        int[] res = new int[diff.length];
        // 根据差分数组构造结果数组
        res[0] = diff[0];
        for (int i = 1; i < diff.length; i++) {
            res[i] = res[i - 1] + diff[i];
        }
        return res;
    }
}
```

这里注意一下 `increment` 方法中的 `if` 语句：

```
void increment(int i, int j, int val) {
    diff[i] += val;
    if (j + 1 < diff.length) {
        diff[j + 1] -= val;
    }
}
```

当  $j+1 \geq diff.length$  时，说明是对 `nums[i]` 及以后的整个数组都进行修改，那么就不需要再给 `diff` 数组减 `val` 了。

## 算法实践

首先，力扣第 370 题「区间加法」就直接考察了差分数组技巧：

### ▼ 370. 区间加法 [Leetcode](#) | 力扣

假设你有一个长度为  $n$  的数组，初始情况下所有的数字均为 0，你将会被给出  $k$  个更新的操作。

其中，每个操作会被表示为一个三元组：[`startIndex`, `endIndex`, `inc`]，你需要将子数组  $A[\text{startIndex} \dots \text{endIndex}]$ （包括 `startIndex` 和 `endIndex`）增加 `inc`。

请你返回  $k$  次操作后的数组。

示例：

```
输入: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]  
输出: [-2,0,3,5,3]
```

解释：

初始状态：  
[0,0,0,0,0]

进行了操作 [1,3,2] 后的状态：  
[0,2,2,2,0]

进行了操作 [2,4,3] 后的状态：  
[0,2,5,5,3]

进行了操作 [0,2,-2] 后的状态：  
[-2,0,3,5,3]

那么我们直接复用刚才实现的 `Difference` 类就能把这道题解决掉：

```
class Solution {  
    public int[] getModifiedArray(int length, int[][] updates) {  
        // nums 初始化为全 0  
        int[] nums = new int[length];  
        // 构造差分解法  
        Difference df = new Difference(nums);  
  
        for (int[] update : updates) {  
            int i = update[0];  
            int j = update[1];  
            int val = update[2];  
            df.increment(i, j, val);  
        }  
  
        return df.result();  
    }  
}
```

当然，实际的算法题可能需要我们对题目进行联想和抽象，不会这么直接地让你看出来要用差分数组技巧，这里看一下力扣第 1109 题「航班预订统计」：

▼ 1109. 航班预订统计 [Leetcode | 力扣](#)

这里有  $n$  个航班，它们分别从 1 到  $n$  进行编号。

有一份航班预订表  $\text{bookings}$ ，表中第  $i$  条预订记录  $\text{bookings}[i] = [\text{first}_i, \text{last}_i, \text{seats}_i]$  意味着在从  $\text{first}_i$  到  $\text{last}_i$ （包含  $\text{first}_i$  和  $\text{last}_i$ ）的每个航班上预订了  $\text{seats}_i$  个座位。

请你返回一个长度为  $n$  的数组  $\text{answer}$ ，里面的元素是每个航班预定的座位总数。

示例 1：

```
输入: bookings = [[1,2,10], [2,3,20], [2,5,25]], n = 5
输出: [10,55,45,25,25]
解释:
航班编号      1   2   3   4   5
预订记录 1 :    10  10
预订记录 2 :        20  20
预订记录 3 :        25  25  25  25
总座位数:      10  55  45  25  25
因此, answer = [10,55,45,25,25]
```

示例 2：

```
输入: bookings = [[1,2,10], [2,2,15]], n = 2
输出: [10,25]
解释:
航班编号      1   2
预订记录 1 :    10  10
预订记录 2 :        15
总座位数:      10  25
因此, answer = [10,25]
```

提示：

- $1 \leq n \leq 2 * 10^4$
- $1 \leq \text{bookings.length} \leq 2 * 10^4$
- $\text{bookings}[i].length == 3$
- $1 \leq \text{first}_i \leq \text{last}_i \leq n$
- $1 \leq \text{seats}_i \leq 10^4$

函数签名如下：

```
int[] corpFlightBookings(int[][] bookings, int n)
```

这个题目就在那绕弯弯，其实它就是个差分数组的题，我给你翻译一下：

给你输入一个长度为  $n$  的数组  $\text{nums}$ ，其中所有元素都是 0。再给你输入一个  $\text{bookings}$ ，里面是若干三元组  $(i, j, k)$ ，每个三元组的含义就是要求你给  $\text{nums}$  数组的闭区间  $[i-1, j-1]$  中所有元素都加上  $k$ 。请你返回最后的  $\text{nums}$  数组

是多少？

因为题目说的 `n` 是从 1 开始计数的，而数组索引从 0 开始，所以对于输入的三元组 (`i`, `j`, `k`)，数组区间应该对应 `[i-1, j-1]`。

这么一看，不就是一道标准的差分数组题嘛？我们可以直接复用刚才写的类：

```
class Solution {
    public int[] corpFlightBookings(int[][] bookings, int n) {
        // nums 初始化为全 0
        int[] nums = new int[n];
        // 构造差分解法
        Difference df = new Difference(nums);

        for (int[] booking : bookings) {
            // 注意转成数组索引要减一哦
            int i = booking[0] - 1;
            int j = booking[1] - 1;
            int val = booking[2];
            // 对区间 nums[i..j] 增加 val
            df.increment(i, j, val);
        }
        // 返回最终的结果数组
        return df.result();
    }
}
```

这道题就解决了。

还有一道很类似的题目是力扣第 1094 题「拼车」，我简单描述下题目：

你是一个开公交车的司机，公交车的最大载客量为 `capacity`，沿途要经过若干车站，给你一份乘客行程表 `int[][] trips`，其中 `trips[i] = [num, start, end]` 代表着有 `num` 个旅客要从站点 `start` 上车，到站点 `end` 下车，请你计算是否能够一次把所有旅客运送完毕（不能超过最大载客量 `capacity`）。

函数签名如下：

```
boolean carPooling(int[][] trips, int capacity);
```

比如输入：

```
trips = [[2,1,5],[3,3,7]], capacity = 4
```

这就不能一次运完，因为 `trips[1]` 最多只能上 2 人，否则车就会超载。

相信你已经能够联想到差分数组技巧了：`trips[i]` 代表着一组区间操作，旅客的上车和下车就相当于数组的区间加减；只要结果数组中的元素都小于 `capacity`，就说明可以不超载运输所有旅客。

但问题是，差分数组的长度（车站的个数）应该是多少呢？题目没有直接给，但给出了数据取值范围：

```
0 <= trips[i][1] < trips[i][2] <= 1000
```

车站编号从 0 开始，最多到 1000，也就是最多有 1001 个车站，那么我们的差分数组长度可以直接设置为 1001，这样索引刚好能够涵盖所有车站的编号：

```
class Solution {
    public boolean carPooling(int[][] trips, int capacity) {
        // 最多有 1001 个车站
        int[] nums = new int[1001];

        // 构造差分解法
        Difference df = new Difference(nums);

        for (int[] trip : trips) {
            // 乘客数量
            int val = trip[0];

            // 第 trip[1] 站乘客上车
            int i = trip[1];

            // 第 trip[2] 站乘客已经下车,
            // 即乘客在车上的区间是 [trip[1], trip[2] - 1]
            int j = trip[2] - 1;

            // 进行区间操作
            df.increment(i, j, val);
        }

        int[] res = df.result();

        // 客车自始至终都不应该超载
        for (int i = 0; i < res.length; i++) {
            if (capacity < res[i]) {
                return false;
            }
        }
        return true;
    }
}
```

至此，这道题也解决了。

差分数组和前缀和数组都是比较常见且巧妙的算法技巧，分别适用不同的场景，而且是会者不难，难者不会。所以，关于差分数组的使用，你学会了吗？

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 滑动窗口算法核心代码模板



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">3. Longest Substring Without Repeating Characters</a>	<a href="#">3. 无重复字符的最长子串</a>	
<a href="#">567. Permutation in String</a>	<a href="#">567. 字符串的排列</a>	
<a href="#">76. Minimum Window Substring</a>	<a href="#">76. 最小覆盖子串</a>	
<a href="#">438. Find All Anagrams in a String</a>	<a href="#">438. 找到字符串中所有字母异位词</a>	

阅读本文前，你需要先学习：

- [数组基础](#)
- [数组实现](#)

关于双指针的快慢指针和左右指针的用法，可以参见前文 [双指针技巧汇总](#)，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。并总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

## 滑动窗口框架概览

滑动窗口算法技巧主要用来解决子数组问题，比如让你寻找符合某个条件的最长/最短子数组。

如果用暴力解的话，你需要嵌套 for 循环这样穷举所有子数组，时间复杂度是  $O(N^2)$ ：

```
for (int i = 0; i < nums.length; i++) {
    for (int j = i; j < nums.length; j++) {
        // nums[i, j] 是一个子数组
    }
}
```

滑动窗口算法技巧的思路也不难，就是维护一个窗口，不断滑动，然后更新答案，该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < nums.size()) {
    // 增大窗口
    window.addLast(nums[right]);
    right++;
}
```

```
while (window needs shrink) {
    // 缩小窗口
    window.removeFirst(nums[left]);
    left++;
}
}
```

基于滑动窗口算法框架写出的代码，时间复杂度是  $O(N)$ ，比嵌套 for 循环的暴力解法效率高。

肯定有读者要问了，你这个滑动窗口框架不也用了一个嵌套 while 循环？为啥复杂度是  $O(N)$  呢？

简单说，指针 `left`, `right` 不会回退（它们的值只增不减），所以字符串/数组中的每个元素都只会进入窗口一次，然后被移出窗口一次，不会说有某些元素多次进入和离开窗口，所以算法的时间复杂度就和字符串/数组的长度成正比。

反观嵌套 for 循环的暴力解法，那个 `j` 会回退，所以某些元素会进入和离开窗口多次，所以时间复杂度就是  $O(N^2)$  了。

我在 [算法时空复杂度分析实用指南](#) 有具体教大家如何从理论上估算时间空间复杂度，这里就不展开了。

这个问题本身就是错误的，滑动窗口并不能穷举出所有子串。要想穷举出所有子串，必须用那个嵌套 for 循环。

然而对于某些题目，并不需要穷举所有子串，就能找到题目想要的答案。滑动窗口就是这种场景下的一套算法模板，帮你对穷举过程进行剪枝优化，避免冗余计算。

所以在 [算法的本质](#) 中我把滑动窗口算法归为「如何聪明地穷举」一类。

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。即便你明白了这些细节，代码也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，保证不会出 bug。

因为本文的例题大多是子串相关的题目，字符串实际上就是数组，所以我就把输入设置成字符串了。你做题的时候根据具体题目自行变通即可：

```
// 滑动窗口算法伪码框架
void slidingWindow(String s) {
    // 用合适的数据结构记录窗口中的数据，根据具体场景变通
    // 比如说，我想记录窗口中元素出现的次数，就用 map
    // 如果我想记录窗口中的元素和，就可以只用一个 int
    Object window = ...

    int left = 0, right = 0;
    while (right < s.length()) {
        // c 是将移入窗口的字符
        char c = s[right];
        window.add(c)
        // 增大窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...
    }
}
```

```

// *** debug 输出的位置 ***
// 注意在最终的解法代码中不要 print
// 因为 IO 操作很耗时，可能导致超时
printf("window: [%d, %d]\n", left, right);
// *****

// 判断左侧窗口是否要收缩
while (left < right && window needs shrink) {
    // d 是将移出窗口的字符
    char d = s[left];
    window.remove(d)
    // 缩小窗口
    left++;
    // 进行窗口内数据的一系列更新
    ...
}
}
}

```

框架中两处 `...` 表示的更新窗口数据的地方，在具体的题目中，你需要做的就是往这里面填代码逻辑。而且，这两个 `...` 处的操作分别是扩大和缩小窗口的更新操作，等会你会发现它们操作是完全对称的。

说句题外话，有些读者评论我这个框架，说散列表速度慢，不如用数组代替散列表；还有些人喜欢把代码写得特别短小，说我这样代码太多余，速度不够快。我的意见是，算法主要看时间复杂度，你能确保自己的时间复杂度最优就行了。至于 LeetCode 的运行速度，那个有点玄学，只要不是慢的离谱就没啥问题，根本不值得你从编译层面优化，不要舍本逐末……

再说，我的算法教程重点在于算法思想，你先做到能把框架思维运用自如，然后随便你魔改代码好吧，保你怎么写都能写对。

言归正传，下面就直接上四道力扣原题来套这个框架，其中第一道题会详细说明其原理，后面四道就直接闭眼睛秒杀了。

## 一、最小覆盖子串

先来看看力扣第 76 题「最小覆盖子串」难度 Hard：

### ▼ 76. 最小覆盖子串 [Leetcode | 力扣](#)

给你一个字符串 `s`、一个字符串 `t`。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""`。

**注意：**

- 对于 `t` 中重复字符，我们寻找的子字符串中该字符数量必须不少于 `t` 中该字符数量。
- 如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

**示例 1：**

```

输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 %%%A%%%、 %%%B%%% 和 %%%C%%%。

```

**示例 2：**

输入: s = "a", t = "a"  
输出: "a"  
解释: 整个字符串 s 是最小覆盖子串。

### 示例 3:

输入: s = "a", t = "aa"  
输出: ""  
解释: t 中两个字符 %%%a%%% 均应包含在 s 的子串中,  
因此没有符合条件的子字符串, 返回空字符串。

### 提示:

- m == s.length
- n == t.length
- 1 <= m, n <= 10<sup>5</sup>
- s 和 t 由英文字母组成

进阶: 你能设计一个在 O(m+n) 时间内解决此问题的算法吗?

就是说要在 S(source) 中找到包含 T(target) 中全部字母的一个子串, 且这个子串一定是所有可能子串中最短的。

如果我们使用暴力解法, 代码大概是这样的:

```
for (int i = 0; i < s.length(); i++)
    for (int j = i + 1; j < s.length(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

思路很直接, 但是显然, 这个算法的复杂度肯定大于 O(N<sup>2</sup>) 了, 不好。

滑动窗口算法的思路是这样:

1、我们在字符串 S 中使用双指针中的左右指针技巧, 初始化 left = right = 0, 把索引左闭右开区间 [left, right) 称为一个「窗口」。

理论上你可以设计两端都开或者两端都闭的区间, 但设计为左闭右开区间是最方便处理的。

因为这样初始化 left = right = 0 时区间 [0, 0) 中没有元素, 但只要让 right 向右移动 (扩大) 一位, 区间 [0, 1) 就包含一个元素 0 了。

如果你设置为两端都开的区间, 那么让 right 向右移动一位后区间 (0, 1) 仍然没有元素; 如果你设置为两端都闭的区间, 那么初始区间 [0, 0] 就包含了一个元素。这两种情况都会给边界处理带来不必要的麻烦。

2、我们先不断地增加 right 指针扩大窗口 [left, right), 直到窗口中的字符串符合要求 (包含了 T 中的所有字符)。

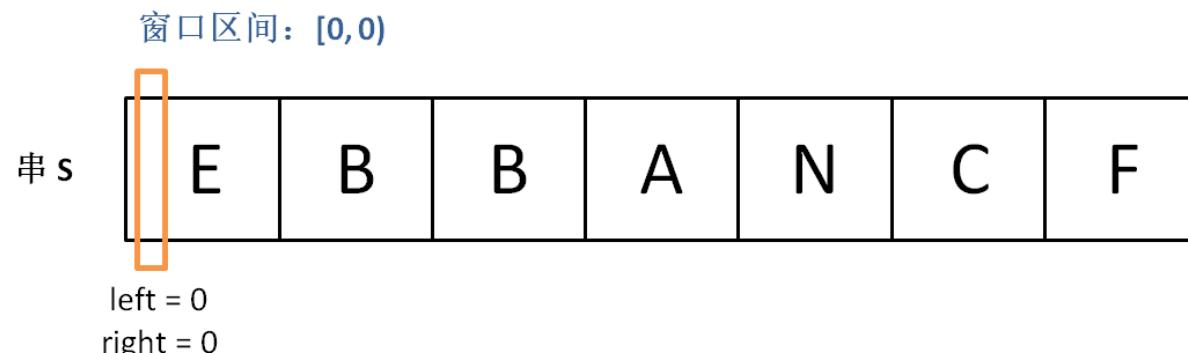
3、此时, 我们停止增加 right, 转而不断增加 left 指针缩小窗口 [left, right), 直到窗口中的字符串不再符合要求 (不包含 T 中的所有字符了)。同时, 每次增加 left, 我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 `right` 到达字符串 `S` 的尽头。

这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，就好像一条毛毛虫，一伸一缩，不断向右滑动，这就是「滑动窗口」这个名字的来历。

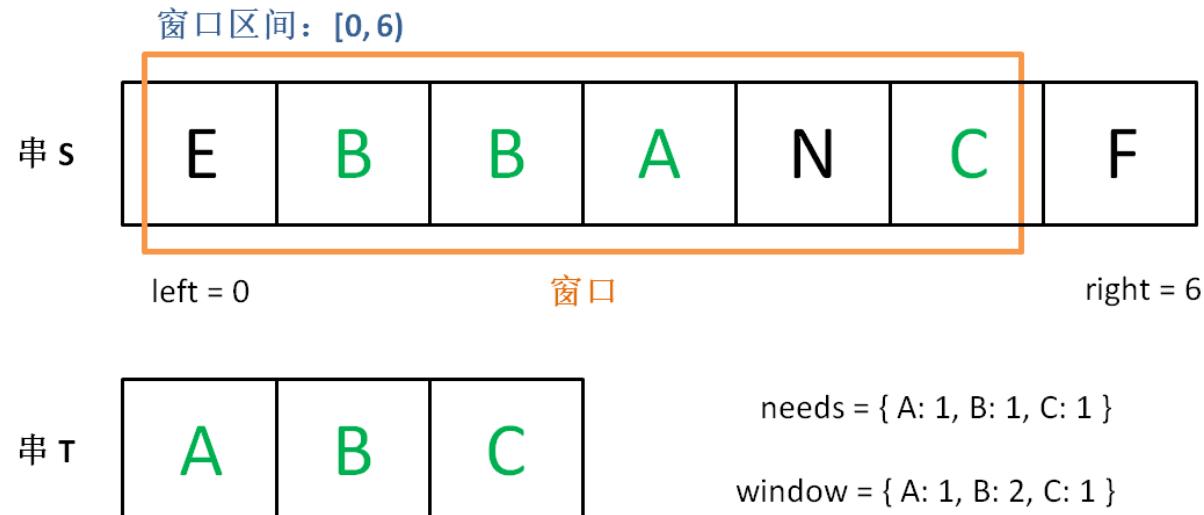
下面画图理解一下，`needs` 和 `window` 相当于计数器，分别记录 `T` 中字符出现次数和「窗口」中的相应字符的出现次数。

初始状态：



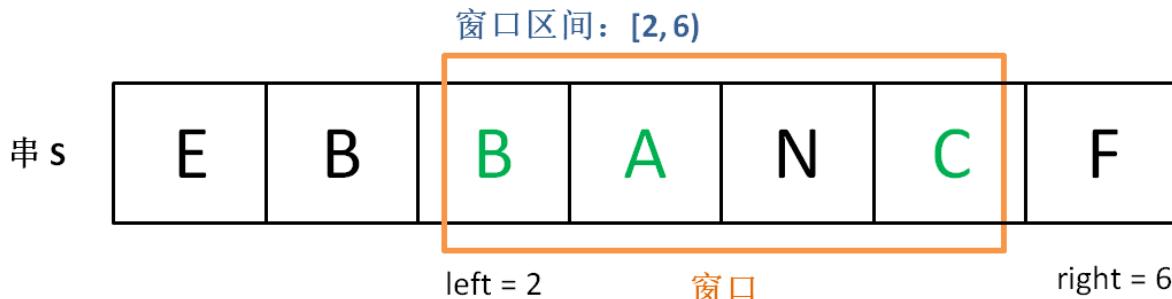
公众号：labuladong

增加 `right`，直到窗口  $[left, right]$  包含了 `T` 中所有字符：



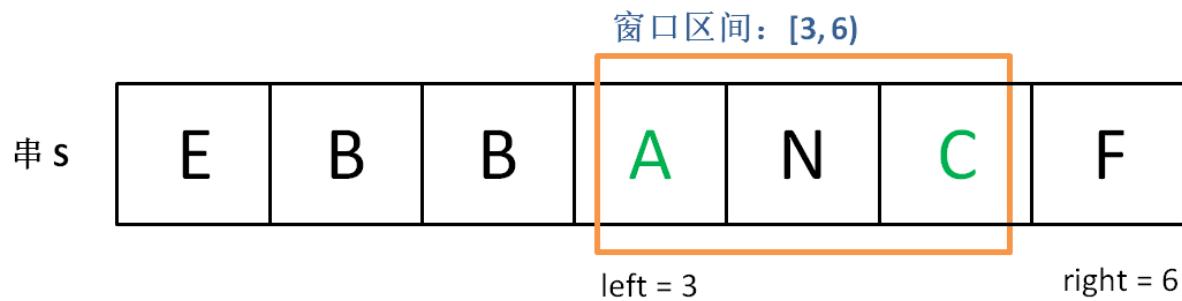
公众号：labuladong

现在开始增加 `left`，缩小窗口  $[left, right]$ ：



公众号: labuladong

直到窗口中的字符串不再符合要求, `left` 不再继续移动:



公众号: labuladong

之后重复上述过程, 先移动 `right`, 再移动 `left`…… 直到 `right` 指针到达字符串 S 的末端, 算法结束。

如果你能够理解上述过程, 恭喜, 你已经完全掌握了滑动窗口算法思想。现在我们来看看这个滑动窗口代码框架怎么用:

首先, 初始化 `window` 和 `need` 两个哈希表, 记录窗口中的字符和需要凑齐的字符:

```
// 记录 window 中的字符出现次数
HashMap<Character, Integer> window = new HashMap<>();
// 记录所需的字符出现次数
HashMap<Character, Integer> need = new HashMap<>();
for (int i = 0; i < t.length(); i++) {
    char c = t.charAt(i);
    need.put(c, need.getOrDefault(c, 0) + 1);
}
```

然后, 使用 `left` 和 `right` 变量初始化窗口的两端, 不要忘了, 区间 `[left, right)` 是左闭右开的, 所以初始情况下窗口没有包含任何元素:

```
int left = 0, right = 0;
int valid = 0;
while (right < s.length()) {
    // c 是将移入窗口的字符
    char c = s.charAt(right);
    // 右移窗口
    right++;
    // 进行窗口内数据的一系列更新
    ...
}
```

其中 `valid` 变量表示窗口中满足 `need` 条件的字符个数，如果 `valid` 和 `need.size()` 的大小相同，则说明窗口已满足条件，已经完全覆盖了串 `T`。

现在开始套模板，只需要思考以下几个问题：

- 1、什么时候应该移动 `right` 扩大窗口？窗口加入字符时，应该更新哪些数据？
- 2、什么时候窗口应该暂停扩大，开始移动 `left` 缩小窗口？从窗口移出字符时，应该更新哪些数据？
- 3、我们要的结果应该在扩大窗口时还是缩小窗口时进行更新？

如果一个字符进入窗口，应该增加 `window` 计数器；如果一个字符将移出窗口的时候，应该减少 `window` 计数器；当 `valid` 满足 `need` 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

下面是完整代码：

```
class Solution {
    public String minWindow(String s, String t) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        // 记录最小覆盖子串的起始索引及长度
        int start = 0, len = Integer.MAX_VALUE;
        while (right < s.length()) {
            // c 是将移入窗口的字符
            char c = s.charAt(right);
            // 扩大窗口
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).equals(need.get(c)))
                    valid++;
            }

            // 判断左侧窗口是否要收缩
            while (valid == need.size()) {
                // 在这里更新最小覆盖子串
                if (right - left < len) {
                    start = left;
                    len = right - left;
                }
                // 缩小窗口
                window.put(s.charAt(left), window.get(s.charAt(left)) - 1);
                if (window.get(s.charAt(left)).equals(need.get(s.charAt(left))))
                    valid--;
                left++;
            }
        }
        return start < len ? s.substring(start, start + len) : "";
    }
}
```

```
        len = right - left;
    }
    // d 是将移出窗口的字符
    char d = s.charAt(left);
    // 缩小窗口
    left++;
    // 进行窗口内数据的一系列更新
    if (need.containsKey(d)) {
        if (window.get(d).equals(need.get(d)))
            valid--;
        window.put(d, window.get(d) - 1);
    }
}
// 返回最小覆盖子串
return len == Integer.MAX_VALUE ? "" : s.substring(start, start + len);
}
```

## ▶ 🔍 代码可视化动画

对 Java 包装类进行比较时要尤为小心，`Integer`, `String` 等类型应该用 `equals` 方法判定相等，而不能直接用等号 `==`，否则会出错。所以在缩小窗口更新数据的时候，不能直接写为 `window.get(d) == need.get(d)`，而要用 `window.get(d).equals(need.get(d))`，之后的题目代码同理。

上面的代码中，当我们发现某个字符在 `window` 的数量满足了 `need` 的需要，就要更新 `valid`，表示有一个字符已经满足要求。而且，你能发现，两次对窗口内数据的更新操作是完全对称的。

当 `valid == need.size()` 时，说明 T 中所有字符已经被覆盖，已经得到一个可行的覆盖子串，现在应该开始收缩窗口了，以便得到「最小覆盖子串」。

移动 `left` 收缩窗口时，窗口内的字符都是可行解，所以应该在收缩窗口的阶段进行最小覆盖子串的更新，以便从可行解中找到长度最短的最终结果。

至此，应该可以完全理解这套框架了，滑动窗口算法又不难，就是细节问题让人烦得很。以后遇到滑动窗口算法，你就按照这框架写代码，保准没有 bug，还省事儿。

下面就直接利用这套框架秒杀几道题吧，你基本上一眼就能看出思路了。

## 二、字符串排列

这是力扣第 567 题「字符串的排列」，难度中等：

### ▼ 567. 字符串的排列 [Leetcode | 力扣](#)

给你两个字符串 `s1` 和 `s2`，写一个函数来判断 `s2` 是否包含 `s1` 的排列。如果是，返回 `true`；否则，返回 `false`。

换句话说，`s1` 的排列之一是 `s2` 的子串。

示例 1：

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: true
解释: s2 包含 s1 的排列之一 ("ba").
```

## 示例 2：

```
输入: s1= "ab" s2 = "eidboaoo"
输出: false
```

提示：

- $1 \leq s1.length, s2.length \leq 10^4$
- $s1$  和  $s2$  仅包含小写字母

注意哦，输入的  $s1$  是可以包含重复字符的，所以这个题难度不小。

这种题目，是明显的滑动窗口算法，相当给你一个  $S$  和一个  $T$ ，请问你  $S$  中是否存在一个子串，包含  $T$  中所有字符且不包含其他字符？

首先，先复制粘贴之前的算法框架代码，然后明确刚才提出的几个问题，即可写出这道题的答案：

```
class Solution {
    // 判断 s 中是否存在 t 的排列
    public boolean checkInclusion(String t, String s) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        while (right < s.length()) {
            char c = s.charAt(right);
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).intValue() == need.get(c).intValue())
                    valid++;
            }

            // 判断左侧窗口是否要收缩
            while (right - left >= t.length()) {
                // 在这里判断是否找到了合法的子串
                if (valid == need.size())
                    return true;
                char d = s.charAt(left);
                left++;
                // 进行窗口内数据的一系列更新
                if (need.containsKey(d)) {
                    if (window.get(d).intValue() == need.get(d).intValue())
                        valid--;
                    window.put(d, window.get(d) - 1);
                }
            }
        }
        // 未找到符合条件的子串
        return false;
    }
}
```

```
    }  
}
```

## ► 🌟 代码可视化动画🌟

对于这道题的解法代码，基本上和最小覆盖子串一模一样，只需要改变几个地方：

- 1、本题移动 `left` 缩小窗口的时机是窗口大小大于 `t.length()` 时，因为排列嘛，显然长度应该是一样的。
- 2、当发现 `valid == need.size()` 时，就说明窗口中就是一个合法的排列，所以立即返回 `true`。

至于如何处理窗口的扩大和缩小，和最小覆盖子串完全相同。

由于这道题中 `[left, right]` 其实维护的是一个定长的窗口，窗口长度为 `t.length()`。因为定长窗口每次向前滑动时只会移出一个字符，所以完全可以把内层的 `while` 改成 `if`，效果是一样的。

## 三、找所有字母异位词

这是力扣第 438 题「找到字符串中所有字母异位词」，难度中等：

### ▼ 438. 找到字符串中所有字母异位词 [Leetcode | 力扣](#)

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1：

```
输入: s = "cbaebabacd", p = "abc"  
输出: [0,6]  
解释:  
起始索引等于 0 的子串是 "cba"，它是 "abc" 的异位词。  
起始索引等于 6 的子串是 "bac"，它是 "abc" 的异位词。
```

示例 2：

```
输入: s = "abab", p = "ab"  
输出: [0,1,2]  
解释:  
起始索引等于 0 的子串是 "ab"，它是 "ab" 的异位词。  
起始索引等于 1 的子串是 "ba"，它是 "ab" 的异位词。  
起始索引等于 2 的子串是 "ab"，它是 "ab" 的异位词。
```

提示：

- $1 \leq s.length, p.length \leq 3 * 10^4$
- `s` 和 `p` 仅包含小写字母

呵呵，这个所谓的字母异位词，不就是排列吗，搞个高端的说法就能糊弄人了吗？相当于，输入一个串 `S`，一个串 `T`，找到 `S` 中所有 `T` 的排列，返回它们的起始索引。

直接默写一下框架，明确刚才讲的 4 个问题，即可秒杀这道题：

```
class Solution {
    public List<Integer> findAnagrams(String s, String t) {
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();
        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int valid = 0;
        // 记录结果
        List<Integer> res = new ArrayList<>();
        while (right < s.length()) {
            char c = s.charAt(right);
            right++;
            // 进行窗口内数据的一系列更新
            if (need.containsKey(c)) {
                window.put(c, window.getOrDefault(c, 0) + 1);
                if (window.get(c).equals(need.get(c))) {
                    valid++;
                }
            }
            // 判断左侧窗口是否要收缩
            while (right - left >= t.length()) {
                // 当窗口符合条件时，把起始索引加入 res
                if (valid == need.size())
                    res.add(left);
                char d = s.charAt(left);
                left++;
                // 进行窗口内数据的一系列更新
                if (need.containsKey(d)) {
                    if (window.get(d).equals(need.get(d))) {
                        valid--;
                    }
                    window.put(d, window.get(d) - 1);
                }
            }
        }
        return res;
    }
}
```

跟寻找字符串的排列一样，只是找到一个合法异位词（排列）之后将起始索引加入 `res` 即可。

▶  代码可视化动画

## 四、最长无重复子串

这是力扣第 3 题「无重复字符的最长子串」，难度中等：

▼ 3. 无重复字符的最长子串 [Leetcode | 力扣](#)

给定一个字符串 `s`，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

输入: s = "abcabcbb"  
输出: 3  
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: s = "bbbbbb"  
输出: 1  
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: s = "pwwkew"  
输出: 3  
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。  
请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

提示:

- $0 \leq s.length \leq 5 * 10^4$
- s 由英文字母、数字、符号和空格组成

这个题终于有了点新意，不是一套框架就出答案，不过反而更简单了，稍微改一改框架就行了：

```
class Solution {  
    public int lengthOfLongestSubstring(String s) {  
        Map<Character, Integer> window = new HashMap<>();  
        int left = 0, right = 0;  
        // 记录结果  
        int res = 0;  
        while (right < s.length()) {  
            char c = s.charAt(right);  
            right++;  
            // 进行窗口内数据的一系列更新  
            window.put(c, window.getOrDefault(c, 0) + 1);  
            // 判断左侧窗口是否要收缩  
            while (window.get(c) > 1) {  
                char d = s.charAt(left);  
                left++;  
                // 进行窗口内数据的一系列更新  
                window.put(d, window.get(d) - 1);  
            }  
            // 在这里更新答案  
            res = Math.max(res, right - left);  
        }  
        return res;  
    }  
}
```

▶  代码可视化动画 

这就是变简单了，连 `need` 和 `valid` 都不需要，而且更新窗口内数据也只需要简单的更新计数器 `window` 即可。

当 `window[c]` 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 `left` 缩小窗口了嘛。

唯一需要注意的是，在哪里更新结果 `res` 呢？我们要的是最长无重复子串，哪一个阶段可以保证窗口中的字符串是没有重复的呢？

这里和之前不一样，要在收缩窗口完成后更新 `res`，因为窗口收缩的 while 条件是存在重复元素，换句话说收缩完成后一定保证窗口中没有重复嘛。

好了，滑动窗口算法模板就讲到这里，希望大家能理解其中的思想，记住算法模板并融会贯通。回顾一下，遇到子数组/子串相关的问题，你只要能回答出来以下几个问题，就能运用滑动窗口算法：

- 1、什么时候应该扩大窗口？
- 2、什么时候应该缩小窗口？
- 3、什么时候应该更新答案？

我在 [滑动窗口经典习题](#) 中使用这套思维模式列举了更多经典的习题，旨在强化你对算法的理解和记忆，以后就再也不怕子串、子数组问题了。

▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1004. Max Consecutive Ones III</a>	1004. 最大连续1的个数 III	
<a href="#">1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit</a>	1438. 绝对差不超过限制的最长连续子数组	
<a href="#">1658. Minimum Operations to Reduce X to Zero</a>	1658. 将 x 减到 0 的最小操作数	
<a href="#">209. Minimum Size Subarray Sum</a>	209. 长度最小的子数组	
<a href="#">219. Contains Duplicate II</a>	219. 存在重复元素 II	
<a href="#">220. Contains Duplicate III</a>	220. 存在重复元素 III	
<a href="#">340. Longest Substring with At Most K Distinct Characters</a> 	340. 至多包含 K 个不同字符的最长子串	
<a href="#">395. Longest Substring with At Least K Repeating Characters</a>	395. 至少有 K 个重复字符的最长子串	
<a href="#">424. Longest Repeating Character Replacement</a>	424. 替换后的最长重复字符	
<a href="#">560. Subarray Sum Equals K</a>	560. 和为 K 的子数组	
<a href="#">713. Subarray Product Less Than K</a>	713. 乘积小于 K 的子数组	
<a href="#">862. Shortest Subarray with Sum at Least K</a>	862. 和至少为 K 的最短子数组	
-		
剑指 Offer 48. 最长不含重复字符的子字符串		

LeetCode	力扣	难度
-	剑指 Offer 57 - II. 和为s的连续正数序列	
-	剑指 Offer II 008. 和大于等于 target 的最短子数组	
-	剑指 Offer II 009. 乘积小于 K 的子数组	
-	剑指 Offer II 010. 和为 k 的子数组	
-	剑指 Offer II 014. 字符串中的变位词	
-	剑指 Offer II 015. 字符串中的所有变位词	
-	剑指 Offer II 016. 不含重复字符的最长子字符串	
-	剑指 Offer II 017. 含有所有字符的最短字符串	
-	剑指 Offer II 057. 值和下标之差都在给定的范围内	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

## 【强化练习】滑动窗口算法经典习题

阅读本文前，你需要先学习：

- [滑动窗口算法框架](#)

滑动窗口的应用非常广泛，但我们的框架可以套用所有需要滑动窗口算法的题目中，下面就来举例一些最经典的题目，我会反复强调 [滑动窗口算法框架](#) 中的思考方式，以强化你对这个算法的理解和记忆。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 滑动窗口延伸：Rabin Karp 字符匹配算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">28. Find the Index of the First Occurrence in a String</a>	28. 找出字符串中第一个匹配项的下标	
<a href="#">187. Repeated DNA Sequences</a>	187. 重复的DNA序列	

阅读本文前，你需要先学习：

- 哈希表原理及实现
- 滑动窗口算法框架

经常有读者留言，请我讲讲那些比较经典的算法，我觉得有这个必要，主要有以下原因：

1、经典算法之所以经典，一定是因为有独特新颖的设计思想，那当然要带大家学习一波。

2、我会尽量从最简单、最基本的设计切入，带你亲手推导出来这些经典算法的设计思想，自然流畅地写出最终解法。一方面消除大多数人对算法的恐惧，另一方面可以避免很多人对算法死记硬背的错误习惯。

我之前用状态机的思路讲解了 [KMP 算法](#)，说实话 KMP 算法确实不太好理解。不过今天我来讲一讲字符串匹配的另一种经典算法：[Rabin-Karp 算法](#)，这是一个很简单优雅的算法。

本文会由浅入深地讲明白这个算法的核心思路，先从最简单的字符串转数字讲起，然后研究一道力扣题目，到最后你就会发现 Rabin-Karp 算法使用的就是滑动窗口技巧，直接套前文讲的 [滑动窗口算法框架](#) 就出来了，根本不用死记硬背。

废话不多说了，直接上干货。

首先，我问你一个很基础的问题，给你输入一个字符串形式的正整数，如何把它转化成数字的形式？很简单，下面这段代码就可以做到：

```
String s = "8264";
int number = 0;
for (int i = 0; i < s.length(); i++) {
    // 将字符转化成数字
    number = 10 * number + (s.charAt(i) - '0');
    System.out.println(number);
}
// 打印输出:
// 8
// 82
```

```
// 826  
// 8264
```

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

## 二分搜索算法核心代码模板



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
34. Find First and Last Position of Element in Sorted Array	34. 在排序数组中查找元素的第一个和最后一个位置	
-	剑指 Offer 53 - I. 在排序数组中查找数字 I	
704. Binary Search	704. 二分查找	

阅读本文前，你需要先学习：

- 数组基础
- 数组实现

tip：本文有视频版：[二分搜索核心框架套路](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文是旧文 [二分搜索详解](#) 的修订版，添加了对二分搜索算法更详细的分析。

先给大家讲个笑话乐呵一下：

有一天阿东到图书馆借了  $N$  本书，出图书馆的时候，警报响了，于是保安把阿东拦下，要检查一下哪本书没有登记出借。阿东正准备把每一本书在报警器下过一下，以找出引发警报的书，但是保安露出不屑的眼神：你连二分查找都不会吗？

于是保安把书分成两堆，让第一堆过一下报警器，报警器响，这说明引起报警的书包含在里面；于是再把这堆书分成两堆，把第一堆过一下报警器，报警器又响，继续分成两堆……

最终，检测了  $\log N$  次之后，保安成功的找到了那本引起警报的书，露出了得意和嘲讽的笑容。于是阿东背着剩下的书走了。

从此，图书馆丢了  $N - 1$  本书（手动狗头）。

二分查找并不简单，Knuth 大佬（发明 KMP 算法的那位）都说二分查找：**思路很简单，细节是魔鬼**。很多人喜欢拿整型溢出的 bug 说事儿，但是二分查找真正的坑根本就不是那个细节问题，而是在于到底要给  $mid$  加一还是减一，while 里到底用  $<=$  还是  $<$ 。

你要是没有正确理解这些细节，写二分肯定就是玄学编程，有没有 bug 只能靠菩萨保佑，谁写谁知道。

本文就来探究几个最常用的二分查找场景：寻找一个数、寻找左侧边界、寻找右侧边界。而且，我们就是要深入细节，比如不等号是否应该带等号， $mid$  是否应该加一等等。分析这些细节的差异以及出现这些差异的原因，保证你能灵活准确地

写出正确的二分查找算法。

另外再声明一下，对于二分搜索的每一个场景，本文还会探讨多种代码写法，目的是为了让你理解出现这些细微差异的本质原因，最起码你看到别人的代码时不会懵逼。实际上这些写法没有优劣之分，你喜欢哪种就用哪种好了。

## 零、二分查找框架

```
int binarySearch(int[] nums, int target) {  
    int left = 0, right = ...;  
  
    while(...) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] == target) {  
            ...  
        } else if (nums[mid] < target) {  
            left = ...  
        } else if (nums[mid] > target) {  
            right = ...  
        }  
    }  
    return ...;  
}
```

分析二分查找的一个技巧是：不要出现 `else`，而是把所有情况用 `else if` 写清楚，这样可以清楚地展现所有细节。本文都会使用 `else if`，旨在讲清楚，读者理解后可自行简化。

其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

另外提前说明一下，计算 `mid` 时需要防止溢出，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大，直接相加导致溢出的情况。

## 一、寻找一个数（基本的二分搜索）

这个场景是最简单的，可能也是大家最熟悉的，即搜索一个数，如果存在，返回其索引，否则返回 -1。

```
int binarySearch(int[] nums, int target) {  
    int left = 0;  
    // 注意  
    int right = nums.length - 1;  
  
    while(left <= right) {  
        int mid = left + (right - left) / 2;  
        if(nums[mid] == target)  
            return mid;  
        else if (nums[mid] < target)  
            // 注意  
            left = mid + 1;  
        else if (nums[mid] > target)  
            // 注意  
            right = mid - 1;  
    }  
    return -1;  
}
```

## ▶ ⭐ 代码可视化动画⭐

这段代码可以解决力扣第 704 题「二分查找」，但我们深入探讨一下其中的细节。

### 为什么 while 循环的条件是 `<=` 而不是 `<`？

答：因为初始化 `right` 的赋值是 `nums.length - 1`，即最后一个元素的索引，而不是 `nums.length`。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 `[left, right]`，后者相当于左闭右开区间 `[left, right)`。因为索引大小为 `nums.length` 是越界的，所以我们把 `right` 这一边视为开区间。

我们这个算法中使用的是前者 `[left, right]` 两端都闭的区间。这个区间其实就是每次进行搜索的区间。

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```
if(nums[mid] == target)
    return mid;
```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？**搜索区间为空的时候应该终止**，意味着你没得找了，就等于没找到嘛。

`while(left <= right)` 的终止条件是 `left == right + 1`，写成区间的形式就是 `[right + 1, right]`，或者带个具体的数字进去 `[3, 2]`，可见**这时候区间为空**，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的，直接返回 -1 即可。

`while(left < right)` 的终止条件是 `left == right`，写成区间的形式就是 `[right, right]`，或者带个具体的数字进去 `[2, 2]`，**这时候区间非空**，还有一个数 2，但此时 while 循环终止了。也就是说区间 `[2, 2]` 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 `while(left < right)` 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
// ...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;
```

### 为什么是 `left = mid + 1, right = mid - 1`？

为什么 `left = mid + 1, right = mid - 1`？我看有的代码是 `right = mid` 或者 `left = mid`，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 `[left, right]`。那么当我们发现索引 `mid` 不是要找的 `target` 时，下一步应该去搜索哪里呢？

当然是去搜索区间 `[left, mid-1]` 或者区间 `[mid+1, right]` 对不对？因为 `mid` 已经搜索过，应该从搜索区间中去除。

### 此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

比如说给你有序数组 `nums = [1,2,2,2,3]`, `target` 为 2, 此算法返回的索引是 2, 没错。但是如果我想得到 `target` 的左侧边界，即索引 1，或者我想得到 `target` 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见，你也许会说，找到一个 `target`，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

## 二、寻找左侧边界的二分搜索

以下是最常见的代码形式，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {
    int left = 0;
    // 注意
    int right = nums.length;

    // 注意
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 注意
            right = mid;
        }
    }
    return left;
}
```

为什么 `while` 中是 `<` 而不是 `<=`？

答：用相同的方法分析，因为 `right = nums.length` 而不是 `nums.length - 1`。因此每次循环的「搜索区间」是 `[left, right)` 左闭右开。

`while(left < right)` 终止的条件是 `left == right`，此时搜索区间 `[left, left)` 为空，所以可以正确终止。

这里先要说一个搜索左右侧边界和上面这个算法的一个区别，也是很多读者问的：刚才的 `right` 不是 `nums.length - 1` 吗，为啥这里非要写成 `nums.length` 使得「搜索区间」变成左闭右开呢？

因为对于搜索左右侧边界的二分查找，这种写法比较普遍，我就拿这种写法举例，保证你以后遇到这类代码可以理解。你非要用两端都闭的写法反而更简单，我会在后面写相关的代码，把三种二分搜索都用一种两端都闭的写法统一起来，你耐心往后看就行了。

`target` 不存在时返回什么？

如果 `nums` 中不存在 `target` 这个值，计算出来的这个索引含义是什么？如果我想让它返回 -1，怎么做？

这是一个很好且很重要的问题，你把这个地方理解了，在二分搜索的实际应用场景中就不会懵逼。

直接说结论：如果 `target` 不存在，搜索左侧边界的二分搜索返回的索引是大于 `target` 的最小索引。

举个例子，`nums = [2,3,5,7]`, `target = 4`, `left_bound` 函数返回值是 2，因为元素 5 是大于 4 的最小元素。

有点绕晕了是吧？这个 `left_bound` 函数明明是搜索左边界，但是当 `target` 不存在的时候，却返回的是大于 `target` 的最小索引。这个结论不用死记，你要是拿不准，简单举个例子就能得到这个结论了。

所以跟你说二分搜索这个东西思路很简单，细节是魔鬼嘛，里面的坑太多了。要是真想考你，总有办法可以把你考到怀疑人生。

不是我故意把代码模板总结的这么复杂，而是二分搜索本身就很复杂，这些细节是不可能绕开的，如果你之前没有了解过这些细节，只能说明你之前学得不扎实。就算不用我总结的模板，你也必须搞清楚当 `target` 不存在时算法的行为，否则出了 bug 你都不知道咋改，真有这么严重。

话说回来，`left_bound` 的这个行为有一个好处。比方说现在让你写一个 `floor` 函数，就可以直接用 `left_bound` 函数来实现：

```
// 在一个有序数组中，找到「小于 target 的最大元素的索引」
// 比如说输入 nums = [1,2,2,2,3], target = 2, 函数返回 0，因为 1 是小于 2 的最大元素。
// 再比如输入 nums = [1,2,3,5,6], target = 4, 函数返回 2，因为 3 是小于 4 的最大元素。
int floor(int[] nums, int target) {
    // 当 target 不存在，比如输入 [4,6,8,10], target = 7
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 7 的最大元素
    // 当 target 存在，比如输入 [4,6,8,8,8,10], target = 8
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 8 的最大元素
    return left_bound(nums, target) - 1;
}
```

最后，我的建议是，如果你必须手写二分代码，那么你一定要了解清楚代码的种种行为，本文总结的框架就是在帮你理清这里面的细节。如果非必要，不要自己手写，尽肯能用编程语言提供的标准库函数，可以节约时间，而且标准库函数的行为在文档里都有明确的说明，不容易出错。

如果想让 `target` 不存在时返回 -1 其实很简单，在返回的时候额外判断一下 `nums[left]` 是否等于 `target` 就行了，如果不等于，就说明 `target` 不存在。需要注意的是，访问数组索引之前要保证索引不越界：

```
while (left < right) {
    // ...
}
// 如果索引越界，说明数组中无目标元素，返回 -1
if (left < 0 || left >= nums.length) {
    return -1;
}
// 提示：其实上面的 if 中 left < 0 这个判断可以省略，因为对于这个算法，left 不可能小于 0
// 你看这个算法执行的逻辑，left 初始化就是 0，且只可能一直往右走，那么只可能在右侧越界
// 不过我这里就同时判断了，因为在访问数组索引之前保证索引在左右两端都不越界是一个好习惯，没有坏处
// 另一个好处是让二分的模板更统一，降低你的记忆成本，因为等会儿寻找右边界的时候也有类似的出界判断

// 判断一下 nums[left] 是不是 target
return nums[left] == target ? left : -1;
```

## 为什么是 `left = mid + 1` 和 `right = mid`？

为什么 `left = mid + 1, right = mid`? 和之前的算法不一样?

答: 这个很好解释, 因为我们的「搜索区间」是 `[left, right)` 左闭右开, 所以当 `nums[mid]` 被检测之后, 下一步应该去 `mid` 的左侧或者右侧区间搜索, 即 `[left, mid)` 或 `[mid + 1, right)`。

为什么该算法能够搜索左侧边界?

答: 关键在于对于 `nums[mid] == target` 这种情况的处理:

```
if (nums[mid] == target)
    right = mid;
```

可见, 找到 `target` 时不要立即返回, 而是缩小「搜索区间」的上界 `right`, 在区间 `[left, mid)` 中继续搜索, 即不断向左收缩, 达到锁定左侧边界的目的。

为什么返回 `left` 而不是 `right`?

答: 都是一样的, 因为 `while` 终止的条件是 `left == right`。

能否统一成两端都闭的搜索区间?

能不能想办法把 `right` 变成 `nums.length - 1`, 也就是继续使用两边都闭的「搜索区间」? 这样就可以和第一种二分搜索在某种程度上统一起来了。

答: 当然可以, 只要你明白了「搜索区间」这个概念, 就能有效避免漏掉元素, 随便你怎么改都行。下面我们严格根据逻辑来修改:

因为你非要让搜索区间两端都闭, 所以 `right` 应该初始化为 `nums.length - 1`, `while` 的终止条件应该是 `left == right + 1`, 也就是其中应该用 `<=`:

```
int left_bound(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // if else ...
    }
}
```

因为搜索区间是两端都闭的, 且现在是搜索左侧边界, 所以 `left` 和 `right` 的更新逻辑如下:

```
if (nums[mid] < target) {
    // 搜索区间变为 [mid+1, right]
    left = mid + 1;
} else if (nums[mid] > target) {
    // 搜索区间变为 [left, mid-1]
    right = mid - 1;
} else if (nums[mid] == target) {
    // 收缩右侧边界
    right = mid - 1;
}
```

和刚才相同，如果想在找不到 `target` 的时候返回 `-1`，那么检查一下 `nums[left]` 和 `target` 是否相等即可：

```
// 此时 target 比所有数都大，返回 -1
if (left == nums.length) return -1;
// 判断一下 nums[left] 是不是 target
return nums[left] == target ? left : -1;
```

至此，整个算法就写完了，完整代码如下：

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 判断 target 是否存在于 nums 中
    // 如果越界，target 肯定不存在，返回 -1
    if (left < 0 || left >= nums.length) {
        return -1;
    }
    // 判断一下 nums[left] 是不是 target
    return nums[left] == target ? left : -1;
}
```

### ▶ 🔍 代码可视化动画

这样就和第一种二分搜索算法统一了，都是两端都闭的「搜索区间」，而且最后返回的也是 `left` 变量的值。只要把住二分搜索的逻辑，两种形式大家看自己喜欢哪种记哪种吧。

## 三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 注意
            left = mid + 1;
        } else if (nums[mid] < target) {
```

```
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid;
    }
}
// 注意
return left - 1;
}
```

## 为什么这个算法能够找到右侧边界？

答：类似地，关键点还是这里：

```
if (nums[mid] == target) {
    left = mid + 1;
}
```

当 `nums[mid] == target` 时，不要立即返回，而是增大「搜索区间」的左边界 `left`，使得区间不断向右靠拢，达到锁定右侧边界的目的。

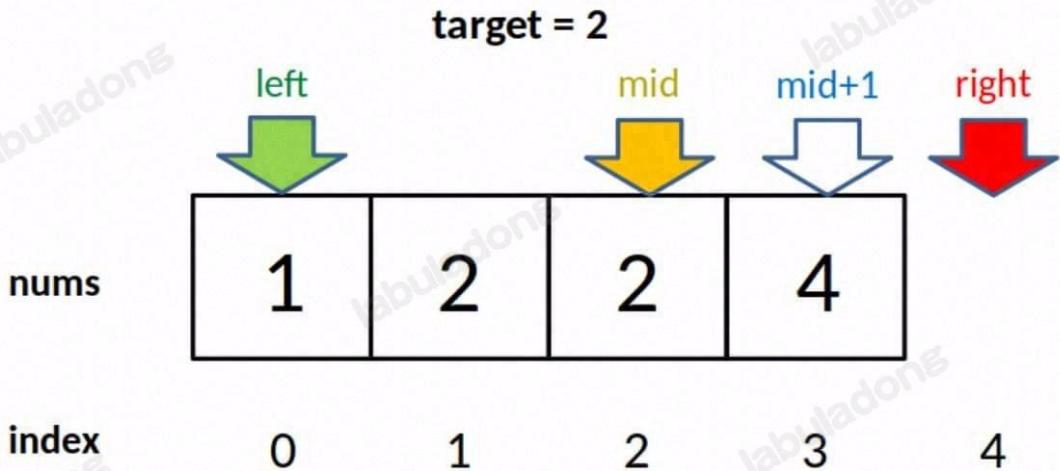
## 为什么返回 `left - 1`？

为什么最后返回 `left - 1` 而不像左侧边界的函数，返回 `left`？而且我觉得这里既然是搜索右侧边界，应该返回 `right` 才对。

答：首先，`while` 循环的终止条件是 `left == right`，所以 `left` 和 `right` 是一样的，你非要体现右侧的特点，返回 `right - 1` 好了。

至于为什么要减一，这是搜索右侧边界的一个特殊点，关键在锁定右边界时的这个条件判断：

```
// 增大 left，锁定右侧边界
if (nums[mid] == target) {
    left = mid + 1;
    // 这样想：mid = left - 1
}
```



公众号：labuladong

因为我们对 `left` 的更新必须是 `left = mid + 1`, 就是说 while 循环结束时, `nums[left]` 一定不等于 `target` 了, 而 `nums[left-1]` 可能是 `target`。

至于为什么 `left` 的更新必须是 `left = mid + 1`, 当然是为了把 `nums[mid]` 排除出搜索区间, 这里就不再赘述。

如果 `target` 不存在时返回什么?

如果 `nums` 中不存在 `target` 这个值, 计算出来的这个索引含义是什么? 如果我想让它返回 -1, 怎么做?

直接说结论, 和前面讲的 `left_bound` 相反: 如果 `target` 不存在, 搜索右侧边界的二分搜索返回的索引是小于 `target` 的最大索引。

这个结论不用死记, 你要是拿不准, 简单举个例子就能得到这个结论了。比如 `nums = [2,3,5,7], target = 4`, `right_bound` 函数返回值是 1, 因为元素 3 是小于 4 的最大元素。

与前面的建议相同, 考虑到二分搜索代码细节的复杂性, 如果非必要, 不要自己手写, 尽肯能用编程语言提供的标准库函数。

如果你想在 `target` 不存在时返回 -1, 很简单, 只要在最后判断一下 `nums[left-1]` 是不是 `target` 就行了, 类似之前的左侧边界搜索, 做一点额外的判断即可:

```

while (left < right) {
    // ...
}
// 判断 target 是否存在于 nums 中
// left - 1 索引越界的话 target 肯定不存在
if (left - 1 < 0 || left - 1 >= nums.length) {
    return -1;
}
// 判断一下 nums[left - 1] 是不是 target
return nums[left - 1] == target ? (left - 1) : -1;

```

4、是否也可以把这个算法的「搜索区间」也统一成两端都闭的形式呢？这样这三个写法就完全统一了，以后就可以闭着眼睛写出来了。

答：当然可以，类似搜索左侧边界的统一写法，其实只要改两个地方就行了：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩左侧边界即可
            left = mid + 1;
        }
    }
    // 最后改成返回 left - 1
    if (left - 1 < 0 || left - 1 >= nums.length) {
        return -1;
    }
    return nums[left - 1] == target ? (left - 1) : -1;
}
```

### ▶ 🎃 代码可视化动画🎃

当然，由于 while 的结束条件为 `right == left - 1`，所以你把上述代码中的 `left - 1` 都改成 `right` 也没有问题，这样可能更有利于看出来这是在「搜索右侧边界」：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩左侧边界即可
            left = mid + 1;
        }
    }
    // 最后改成返回 right
    if (right < 0 || right >= nums.length) {
        return -1;
    }
    return nums[right] == target ? right : -1;
}
```

至此，搜索右侧边界的二分查找的两种写法也完成了，其实将「搜索区间」统一成两端都闭反而更容易记忆，你说是吧？

## 四、逻辑统一

有了搜索左右边界的二分搜索，你可以去解决力扣第 34 题「在排序数组中查找元素的第一个和最后一个位置」。接下来梳理一下这些细节差异的因果逻辑：

### 第一个，最基本的二分查找算法：

因为我们初始化 `right = nums.length - 1`  
所以决定了我们的「搜索区间」是 `[left, right]`  
所以决定了 `while (left <= right)`  
同时也决定了 `left = mid+1` 和 `right = mid-1`

因为我们只需找到一个 `target` 的索引即可  
所以当 `nums[mid] == target` 时可以立即返回

### 第二个，寻找左侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最左侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧右侧边界以锁定左侧边界

### 第三个，寻找右侧边界的二分查找：

因为我们初始化 `right = nums.length`  
所以决定了我们的「搜索区间」是 `[left, right)`  
所以决定了 `while (left < right)`  
同时也决定了 `left = mid + 1` 和 `right = mid`

因为我们需找到 `target` 的最右侧索引  
所以当 `nums[mid] == target` 时不要立即返回  
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 `left = mid + 1`  
所以最后无论返回 `left` 还是 `right`，必须减一

对于寻找左右边界的二分搜索，比较常见的手法是使用左闭右开的「搜索区间」，我们还根据逻辑将「搜索区间」全都统一成了两端都闭，便于记忆，只要修改两处即可变化出三种写法：

```
int binary_search(int[] nums, int target) {  
    int left = 0, right = nums.length - 1;  
    while(left <= right) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] < target) {  
            left = mid + 1;  
        } else if (nums[mid] > target) {  
            right = mid - 1;  
        } else if (nums[mid] == target) {  
            // 直接返回
```

```

        return mid;
    }
}

// 直接返回
return -1;
}

int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定左侧边界
            right = mid - 1;
        }
    }
    // 判断 target 是否存在于 nums 中
    if (left < 0 || left >= nums.length) {
        return -1;
    }
    // 判断一下 nums[left] 是不是 target
    return nums[left] == target ? left : -1;
}

int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定右侧边界
            left = mid + 1;
        }
    }
    // 由于 while 的结束条件是 right == left - 1，且现在在求右边界
    // 所以用 right 替代 left - 1 更好记
    if (right < 0 || right >= nums.length) {
        return -1;
    }
    return nums[right] == target ? right : -1;
}

```

如果以上内容你都能理解，那么恭喜你，二分查找算法的细节不过如此。通过本文，你学会了：

- 1、分析二分查找代码时，不要出现 else，全部展开成 else if 方便理解。把逻辑写对之后再合并分支，提升运行效率。
- 2、注意「搜索区间」和 while 的终止条件，如果存在漏掉的元素，记得在最后检查。
- 3、如需定义左闭右开的「搜索区间」搜索左右边界，只要在 `nums[mid] == target` 时做修改即可，搜索右侧时需要减一。

4、如果将「搜索区间」全都统一成两端都闭，好记，只要稍改 `nums[mid] == target` 条件处的代码和返回的逻辑即可，推荐拿小本本记下，作为二分搜索模板。

最后我想说，以上二分搜索的框架属于「术」的范畴，如果上升到「道」的层面，二分思维的精髓就是：通过已知信息尽可能多地收缩（折半）搜索空间，从而增加穷举效率，快速找到目标。

理解本文能保证你写出正确的二分查找的代码，但实际题目中不会直接让你写二分代码，我会在 [二分查找的运用](#) 和 [二分查找的更多习题](#) 中进一步讲解如何把二分思维运用到更多算法题中。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1201. Ugly Number III</a>	<a href="#">1201. 丑数 III</a>	
<a href="#">1235. Maximum Profit in Job Scheduling</a>	<a href="#">1235. 规划兼职工作</a>	
<a href="#">162. Find Peak Element</a>	<a href="#">162. 寻找峰值</a>	
<a href="#">240. Search a 2D Matrix II</a>	<a href="#">240. 搜索二维矩阵 II</a>	
<a href="#">33. Search in Rotated Sorted Array</a>	<a href="#">33. 搜索旋转排序数组</a>	
<a href="#">35. Search Insert Position</a>	<a href="#">35. 搜索插入位置</a>	
<a href="#">658. Find K Closest Elements</a>	<a href="#">658. 找到 K 个最接近的元素</a>	
<a href="#">74. Search a 2D Matrix</a>	<a href="#">74. 搜索二维矩阵</a>	
<a href="#">792. Number of Matching Subsequences</a>	<a href="#">792. 匹配子序列的单词数</a>	
<a href="#">793. Preimage Size of Factorial Zeroes Function</a>	<a href="#">793. 阶乘函数后 K 个零</a>	
<a href="#">81. Search in Rotated Sorted Array II</a>	<a href="#">81. 搜索旋转排序数组 II</a>	
<a href="#">852. Peak Index in a Mountain Array</a>	<a href="#">852. 山脉数组的峰顶索引</a>	
-	<a href="#">剑指 Offer 04. 二维数组中的查找</a>	
-	<a href="#">剑指 Offer 53 - I. 在排序数组中查找数字 I</a>	
-	<a href="#">剑指 Offer 53 - II. 0~n-1中缺失的数字</a>	
-	<a href="#">剑指 Offer II 068. 查找插入位置</a>	
-	<a href="#">剑指 Offer II 069. 山峰数组的顶部</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 实际运用二分搜索时的思维框架



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1011. Capacity To Ship Packages Within D Days</a>	<a href="#">1011. 在 D 天内送达包裹的能力</a>	简单
<a href="#">875. Koko Eating Bananas</a>	<a href="#">875. 爱吃香蕉的珂珂</a>	简单
<a href="#">410. Split Array Largest Sum</a>	<a href="#">410. 分割数组的最大值</a>	困难

阅读本文前，你需要先学习：

- [二分查找框架详解](#)

在 [二分查找框架详解](#) 中我们详细研究了二分搜索的细节问题，探讨了「搜索一个元素」，「搜索左侧边界」，「搜索右侧边界」这三个情况，教你如何写出正确无 bug 的二分搜索算法。

但是前文总结的二分搜索代码框架仅仅局限于「在有序数组中搜索指定元素」这个基本场景，具体的算法问题没有这么直接，可能你都很难看出这个问题能够用到二分搜索。

所以本文就来总结一套二分搜索算法运用的框架套路，帮你在遇到二分搜索算法相关的实际问题时，能够有条理地思考分析，步步为营，写出答案。

## 原始的二分搜索代码

二分搜索的原型就是在「**有序数组**」中搜索一个元素 **target**，返回该元素对应的索引。

如果该元素不存在，那可以返回一个什么特殊值，这种细节问题只要微调算法实现就可实现。

还有一个重要的问题，如果「**有序数组**」中存在多个 **target** 元素，那么这些元素肯定挨在一起，这里就涉及到算法应该返回最左侧的那个 **target** 元素的索引还是最右侧的那个 **target** 元素的索引，也就是所谓的「**搜索左侧边界**」和「**搜索右侧边界**」，这个也可以通过微调算法的代码来实现。

我们前文 [二分搜索核心框架](#) 详细探讨了上述问题，对这块还不清楚的读者建议复习前文，已经搞清楚基本二分搜索算法的读者可以继续看下去。

在具体的算法问题中，常用到的是「**搜索左侧边界**」和「**搜索右侧边界**」这两种场景，很少有让你单独「**搜索一个元素**」。

因为算法题一般都让你求最值，比如让你求吃香蕉的「**最小速度**」，让你求轮船的「**最低运载能力**」，求最值的过程，必然是搜索一个边界的过过程，所以后面我们就详细分析一下这两种搜索边界的二分算法代码。

注意，本文我写的都是左闭右开的二分搜索写法，如果你习惯两端都闭的写法，可以自行改写代码。

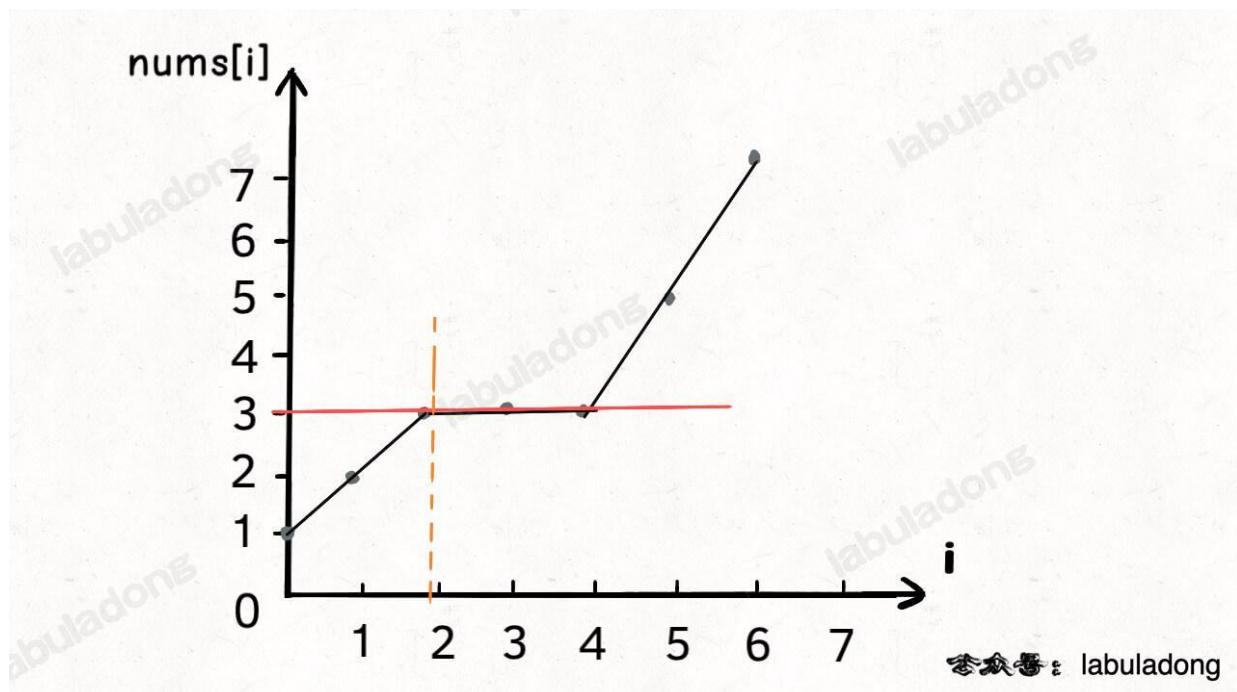
「搜索左侧边界」的二分搜索算法的具体代码实现如下：

```
// 搜索左侧边界
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 当找到 target 时，收缩右侧边界
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left;
}
```

假设输入的数组 `nums = [1,2,3,3,3,5,7]`，想搜索的元素 `target = 3`，那么算法就会返回索引 2。

如果画一个图，就是这样：

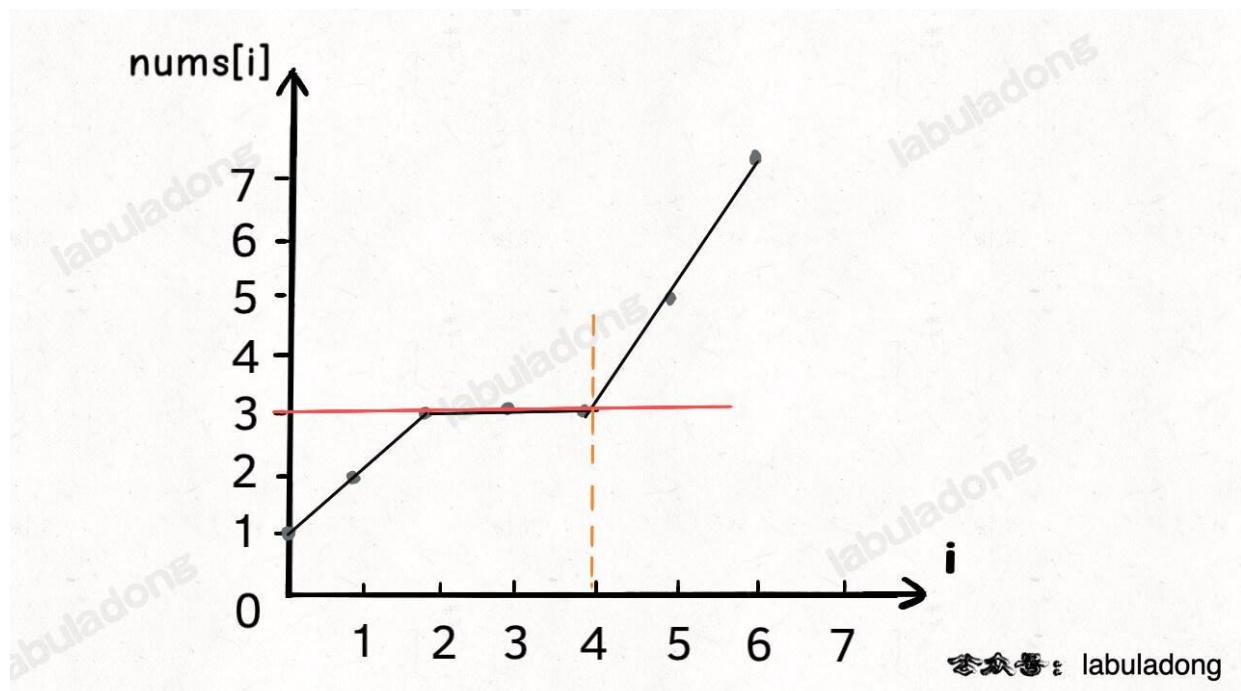


「搜索右侧边界」的二分搜索算法的具体代码实现如下：

```
// 搜索右侧边界
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;
```

```
while (left < right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] == target) {
        // 当找到 target 时, 收缩左侧边界
        left = mid + 1;
    } else if (nums[mid] < target) {
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid;
    }
}
return left - 1;
}
```

输入同上，那么算法就会返回索引 4，如果画一个图，就是这样：



好，上述内容都属于复习，我想读到这里的读者应该都能理解。记住上述的图像，所有能够抽象出上述图像的问题，都可以使用二分搜索解决。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】二分搜索算法经典习题

阅读本文前，你需要先学习：

- [二分搜索算法核心代码模板](#)
- [实际运用二分搜索时的思维框架](#)

二分搜索的精髓在于快速收缩搜索区间。本文带大家看一看二分搜索算法在数组中的经典运用场景。

### 二维数组中的二分搜索

前文 [二分搜索框架详解](#) 讲的二分搜索都是在一维数组中的，在二维矩阵中如何施展二分搜索呢？

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 带权重的随机选择算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
-	剑指 Offer II 071. 按权重生成随机数	●
<a href="#">528. Random Pick with Weight</a>	<a href="#">528. 按权重随机选择</a>	●

阅读本文前，你需要先学习：

- [前缀和算法技巧](#)
- [二分查找框架详解](#)

写这篇文章的原因是玩 LOL 手游。我有个朋友抱怨说打排位匹配的队友太菜了，我就说我打排位觉得队友都挺行的啊，好像不怎么坑？

朋友意味深长地说了句：一般隐藏分比较高的玩家，排位如果排不到实力相当的队友，就会排到一些菜狗。

嗯？我想了几秒钟感觉这小伙子不对劲，他意思是说我隐藏分低，还是说我就是那条菜狗？

我立马要求和他开黑打一把，证明我不是菜狗，他才是菜狗。开黑结果这里不便透露，大家猜猜吧。

打完之后我就来发文了，因为我对游戏的匹配机制有了一点思考。



我反野回来了

所谓「隐藏分」我不知道是不是真的，毕竟匹配机制是所有竞技类游戏的核心环节，想必非常复杂，不是简单几个指标就能搞定的。

但是如果把这个「隐藏分」机制简化，倒是一个值得思考的算法问题：系统如何以不同的随机概率进行匹配？

或者简单点说，如何带权重地做随机选择？

不要觉得这个很容易，如果给你一个长度为  $n$  的数组，让你从中等概率随机抽取一个元素，你肯定会展开，random 一个  $[0, n-1]$  的数字出来作为索引就行了，每个元素被随机选到的概率都是  $1/n$ 。

但假设每个元素都有不同的权重，权重地大小代表随机选到这个元素的概率大小，你如何写算法去随机获取元素呢？

力扣第 528 题「按权重随机选择」就是这样一个问题：

▼ 528. 按权重随机选择 [Leetcode](#) | 力扣

给你一个 **下标从 0 开始** 的正整数数组  $w$ ，其中  $w[i]$  代表第  $i$  个下标的权重。

请你实现一个函数 `pickIndex`，它可以 **随机地** 从范围  $[0, w.length - 1]$  内（含  $0$  和  $w.length - 1$ ）选出并返回一个下标。选取下标  $i$  的 **概率** 为  $w[i] / \text{sum}(w)$ 。

- 例如，对于  $w = [1, 3]$ ，挑选下标  $0$  的概率为  $1 / (1 + 3) = 0.25$ （即， $25\%$ ），而选取下标  $1$  的概率为  $3 / (1 + 3) = 0.75$ （即， $75\%$ ）。

示例 1：

```
输入:  
["Solution","pickIndex"]  
[[[1]],[]]  
输出:  
[null,0]  
解释:  
Solution solution = new Solution([1]);  
solution.pickIndex(); // 返回 0，因为数组中只有一个元素，所以唯一的选择是返回下标 0。
```

示例 2：

```
输入:  
["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]  
[[[1,3]],[],[],[],[],[]]  
输出:  
[null,1,1,1,1,0]  
解释:  
Solution solution = new Solution([1, 3]);  
solution.pickIndex(); // 返回 1，返回下标 1，返回该下标概率为 3/4 。  
solution.pickIndex(); // 返回 1  
solution.pickIndex(); // 返回 1  
solution.pickIndex(); // 返回 1  
solution.pickIndex(); // 返回 0，返回下标 0，返回该下标概率为 1/4 。
```

由于这是一个随机问题，允许多个答案，因此下列输出都可以被认为是正确的：

```
[null,1,1,1,1,0]  
[null,1,1,1,1,1]  
[null,1,1,1,0,0]  
[null,1,1,1,0,1]  
[null,1,0,1,0,0]  
.....  
诸若此类。
```

提示：

- $1 \leq w.length \leq 10^4$

- $1 \leq w[i] \leq 10^5$
- `pickIndex` 将被调用不超过  $10^4$  次

我们就来思考一下这个问题，解决按照权重随机选择元素的问题。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 田忌赛马背后的算法决策



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">870. Advantage Shuffle</a>	<a href="#">870. 优势洗牌</a>	困难

-----

阅读本文前，你需要先学习：

- 数组双指针技巧汇总
- 二叉堆原理
- 二叉堆实现优先级队列

田忌赛马的故事大家应该都听说过：

田忌和齐王赛马，两人的马分上中下三等，如果同等级的马对应着比赛，田忌赢不了齐王。但是田忌遇到了孙膑，孙膑就教他用自己的下等马对齐王的上等马，再用自己的上等马对齐王的中等马，最后用自己的中等马对齐王的下等马，结果三局两胜，田忌赢了。

当然，这段历史也挺有意思的，那个讽齐王纳谏，自恋的不行的邹忌和田忌是同一时期的人，他俩后来就杠上了。不过这是题外话，我们这里就打住。

以前学到田忌赛马课文的时，我就在想，如果不是三匹马比赛，而是一百匹马比赛，孙膑还能不能合理地安排比赛的顺序，赢得齐王呢？

当时没想出什么好的点子，只觉得这里面最核心问题是要尽可能让自己占便宜，让对方吃亏。总结来说就是，打得过就打，打不过就拿自己的垃圾和对方的精锐互换。

不过，我一直没具体把这个思路实现出来，直到最近刷到力扣第 870 题「优势洗牌」，一眼就发现这是田忌赛马问题的加强版：

给你输入两个长度相等的数组 `nums1` 和 `nums2`，请你重新组织 `nums1` 中元素的位置，使得 `nums1` 的「优势」最大化。

如果 `nums1[i] > nums2[i]`，就是说 `nums1` 在索引 `i` 上对 `nums2[i]` 有「优势」。优势最大化也就是说让你重新组织 `nums1`，尽可能多的让 `nums1[i] > nums2[i]`。

算法签名如下：

```
int[] advantageCount(int[] nums1, int[] nums2);
```

比如输入：

```
nums1 = [12, 24, 8, 32]  
nums2 = [13, 25, 32, 11]
```

你的算法应该返回 `[24, 32, 8, 12]`，因为这样排列 `nums1` 的话有三个元素都有「优势」。

这就像田忌赛马的情景，`nums1` 就是田忌的马，`nums2` 就是齐王的马，数组中的元素就是马的战斗力，你就是孙膑，展示你真正的技术吧。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 二叉树系列算法核心纲领



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">144. Binary Tree Preorder Traversal</a>	<a href="#">144. 二叉树的前序遍历</a>	
<a href="#">543. Diameter of Binary Tree</a>	<a href="#">543. 二叉树的直径</a>	
<a href="#">104. Maximum Depth of Binary Tree</a>	<a href="#">104. 二叉树的最大深度</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

本文会把很多算法进行抽象和归纳，所以会包含大量其他文章链接。

第一次阅读本文的读者不要 DFS 学习本文，遇到没学过的算法或不理解的地方请跳过，只要对本文所总结的理论有些印象即可。在学习本站后面的算法技巧时，你自然可以逐渐理解本文的精髓所在，日后回来重读本文，会有更深的体会。

本站所有文章的脉络都是按照 [学习数据结构和算法的框架思维](#) 提出的框架来构建的，其中着重强调了二叉树题目的重要性，所以把本文放在第一章的必读系列中。

我刷了这么多年题，浓缩出二叉树算法的一个总纲放在这里，也许用词不是特别专业化，也没有什么教材会收录我的这些经验总结，但目前各个刷题平台的题库，没有一道二叉树题目能跳出本文划定的框架。如果你能发现一道题目和本文给出的框架不兼容，请留言告知我。

先在开头总结一下，二叉树解题的思维模式分两类：

1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现，这叫「遍历」的思维模式。

2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「分解问题」的思维模式。

无论使用哪种思维模式，你都需要思考：

如果单独抽出一个二叉树节点，它需要做什么事情？需要在什么时候（前/中/后序位置）做？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。

本文中会用题目来举例，但都是最最简单的题目，所以不用担心自己看不懂，我可以帮你从最简单的问题中提炼出所有二叉树题目的共性，并将二叉树中蕴含的思维进行升华，反手用到 [动态规划](#), [回溯算法](#), [分治算法](#), [图论算法](#) 中去，这也是我一直强调框架思维的原因。希望你在学习了上述高级算法后，也能回头再来看看本文，会对它们有更深刻的认识。

首先，我还是要不厌其烦地强调一下二叉树这种数据结构及相关算法的重要性。

## 二叉树的重要性

举个例子，比如两个经典排序算法 [快速排序](#) 和 [归并排序](#)，对于它俩，你有什么理解？

如果你告诉我，快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历，那么我就知道你是个算法高手了。

为什么快速排序和归并排序能和二叉树扯上关系？我们来简单分析一下他们的算法思想和代码框架：

快速排序的逻辑是，若要对 `nums[lo..hi]` 进行排序，我们先找一个分界点 `p`，通过交换元素使得 `nums[lo..p-1]` 都小于等于 `nums[p]`，且 `nums[p+1..hi]` 都大于 `nums[p]`，然后递归地去 `nums[lo..p-1]` 和 `nums[p+1..hi]` 中寻找新的分界点，最后整个数组就被排序了。

快速排序的代码框架如下：

```
void sort(int[] nums, int lo, int hi) {  
    // ***** 前序遍历位置 *****  
    // 通过交换元素构建分界点 p  
    int p = partition(nums, lo, hi);  
    // *****  
  
    sort(nums, lo, p - 1);  
    sort(nums, p + 1, hi);  
}
```

先构造分界点，然后去左右子数组构造分界点，你看这不就是一个二叉树的前序遍历吗？

再说说归并排序的逻辑，若要对 `nums[lo..hi]` 进行排序，我们先对 `nums[lo..mid]` 排序，再对 `nums[mid+1..hi]` 排序，最后把这两个有序的子数组合并，整个数组就排好序了。

归并排序的代码框架如下：

```
// 定义：排序 nums[lo..hi]  
void sort(int[] nums, int lo, int hi) {  
    int mid = (lo + hi) / 2;  
    // 排序 nums[lo..mid]  
    sort(nums, lo, mid);  
    // 排序 nums[mid+1..hi]  
    sort(nums, mid + 1, hi);  
  
    // ***** 后序位置 *****  
    // 合并 nums[lo..mid] 和 nums[mid+1..hi]  
    merge(nums, lo, mid, hi);  
    // *****  
}
```

先对左右子数组排序，然后合并（类似合并有序链表的逻辑），你看这是不是二叉树的后序遍历框架？另外，这不就是传说中的分治算法嘛，不过如此呀。

如果你一眼就识破这些排序算法的底细，还需要背这些经典算法吗？不需要。你可以手到擒来，从二叉树遍历框架就能扩展出算法了。

说了这么多，旨在说明，二叉树的算法思想的运用广泛，甚至可以说，只要涉及递归，都可以抽象成二叉树的问题。

接下来我们从二叉树的前中后序开始讲起，让你深刻理解这种数据结构的魅力。

## 深入理解前中后序

我先甩给你几个问题，请默默思考 30 秒：

- 1、你理解的二叉树的前中后序遍历是什么，仅仅是三个顺序不同的 List 吗？
- 2、请分析，后序遍历有什么特殊之处？
- 3、请分析，为什么多叉树没有中序遍历？

答不上来，说明你对前中后序的理解仅仅局限于教科书，不过没关系，我用类比的方式解释一下我眼中的前中后序遍历。

首先，回顾一下 [二叉树的 DFS/BFS 遍历](#) 中说到的二叉树递归遍历框架：

```
void traverse(TreeNode root) {  
    if (root == null) {  
        return;  
    }  
    // 前序位置  
    traverse(root.left);  
    // 中序位置  
    traverse(root.right);  
    // 后序位置  
}
```

先不管所谓前中后序，单看 `traverse` 函数，你说它在做什么事情？

其实它就是一个能够遍历二叉树所有节点的一个函数，和你遍历数组或者链表本质上没有区别：

```
// 迭代遍历数组  
void traverse(int[] arr) {  
    for (int i = 0; i < arr.length; i++) {  
  
    }  
}  
  
// 递归遍历数组  
void traverse(int[] arr, int i) {  
    if (i == arr.length) {  
        return;  
    }  
    // 前序位置  
    traverse(arr, i + 1);  
    // 后序位置  
}  
  
// 迭代遍历单链表  
void traverse(ListNode head) {  
    for (ListNode p = head; p != null; p = p.next) {
```

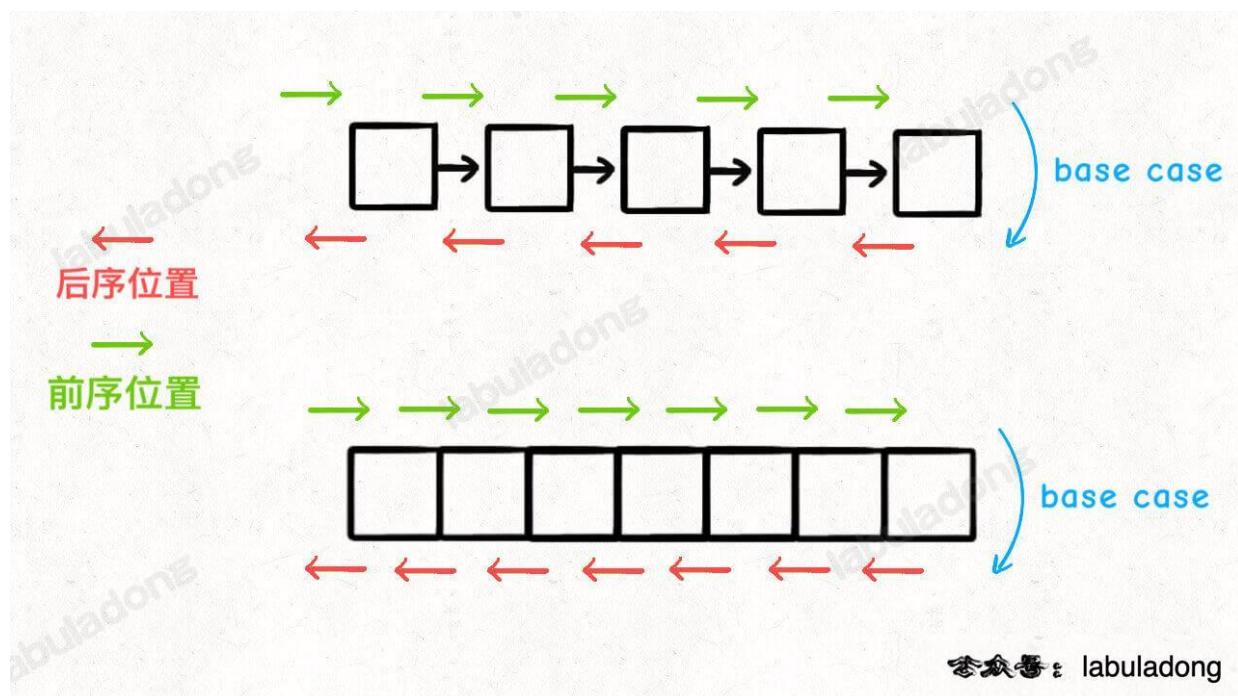
```
    }
}

// 递归遍历单链表
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    // 前序位置
    traverse(head.next);
    // 后序位置
}
```

单链表和数组的遍历可以是迭代的，也可以是递归的，二叉树这种结构无非就是二叉链表，它没办法简单改写成 for 循环的迭代形式，所以我们遍历二叉树一般都使用递归形式。

你也注意到了，只要是递归形式的遍历，都可以有前序位置和后序位置，分别在递归之前和递归之后。

所谓前序位置，就是刚进入一个节点（元素）的时候，后序位置就是即将离开一个节点（元素）的时候，那么进一步，你把代码写在不同位置，代码执行的时机也不同：



比如说，如果让你倒序打印一条单链表上所有节点的值，你怎么搞？

实现方式当然有很多，但如果你对递归的理解足够透彻，可以利用后序位置来操作：

```
// 递归遍历单链表，倒序打印链表元素
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    traverse(head.next);
    // 后序位置
    print(head.val);
}
```

结合上面那张图，你应该知道为什么这段代码能够倒序打印单链表了吧，本质上是利用递归的堆栈帮你实现了倒序遍历的效果。

那么说回二叉树也是一样的，只不过多了一个中序位置罢了。

教科书里只会问你前中后序遍历结果分别是什么，所以对于一个只上过大学数据结构课程的人来说，他大概以为二叉树的前中后序只不过对应三种顺序不同的 `List<Integer>` 列表。

但是我想说，**前中后序是遍历二叉树过程中处理每一个节点的三个特殊时间点**，绝不仅仅是三个顺序不同的 List：

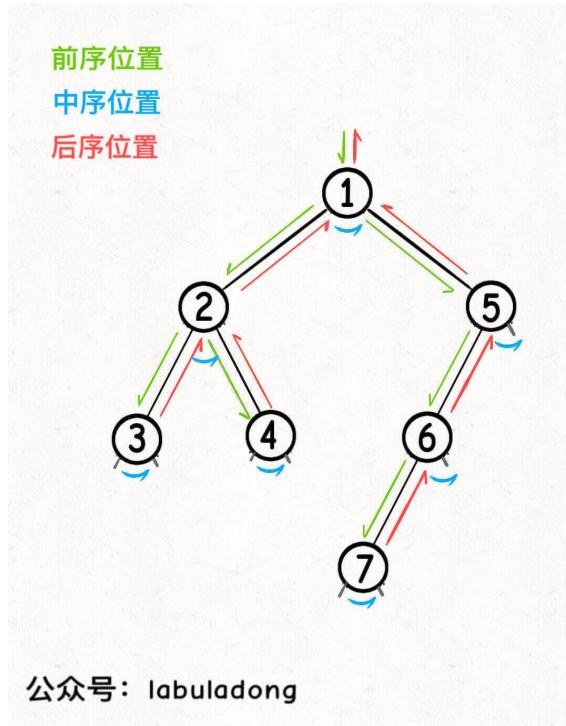
前序位置的代码在刚刚进入一个二叉树节点的时候执行；

后序位置的代码在将要离开一个二叉树节点的时候执行；

中序位置的代码在一个二叉树节点左子树都遍历完，即将开始遍历右子树的时候执行。

你注意本文的用词，我一直说前中后序「位置」，就是要和大家常说的前中后序「遍历」有所区别：你可以在前序位置写代码往一个 List 里面塞元素，那最后得到的就是前序遍历结果；但并不是说你就不可以写更复杂的代码做更复杂的事。

画成图，前中后序三个位置在二叉树上是这样：



你可以发现每个节点都有「唯一」属于自己的前中后序位置，所以我说前中后序遍历是遍历二叉树过程中处理每一个节点的三个特殊时间点。

这里你也可以理解为什么多叉树没有中序位置，因为二叉树的每个节点只会进行唯一一次左子树切换右子树，而多叉树节点可能有很多子节点，会多次切换子树去遍历，所以多叉树节点没有「唯一」的中序遍历位置。

说了这么多基础的，就是要帮你对二叉树建立正确的认识，然后你会发现：

**二叉树的所有问题，就是让你在前中后序位置注入巧妙的代码逻辑，去达到自己的目的，你只需要单独思考每一个节点应该做什么，其他的不用你管，抛给二叉树遍历框架，递归会在所有节点上做相同的操作。**

你也可以看到，[图论算法基础](#) 把二叉树的遍历框架扩展到了图，并以遍历为基础实现了图论的各种经典算法，不过这是后话，本文就不多说了。

## 两种解题思路

前文 [我的算法学习心得](#) 说过：

二叉树题目的递归解法可以分两类思路，第一类是遍历一遍二叉树得出答案，第二类是通过分解问题计算出答案，这两类思路分别对应着 [回溯算法核心框架](#) 和 [动态规划核心框架](#)。

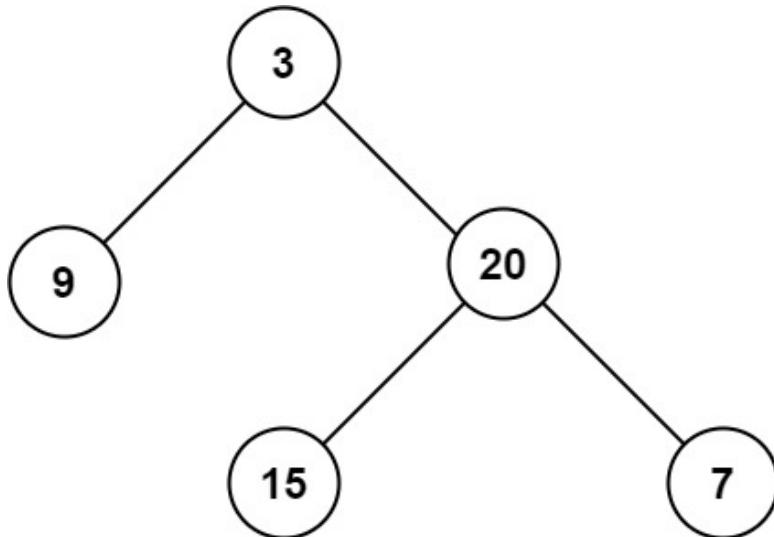
这里说一下我的函数命名习惯：二叉树中用遍历思路解题时函数签名一般是 `void traverse(...)`，没有返回值，靠更新外部变量来计算结果，而用分解问题思路解题时函数名根据该函数具体功能而定，而且一般会有返回值，返回值是子问题的计算结果。

与此对应的，你会发现我在 [回溯算法核心框架](#) 中给出的函数签名一般也是没有返回值的 `void backtrack(...)`，而在 [动态规划核心框架](#) 中给出的函数签名是带有返回值的 `dp` 函数。这也说明它俩和二叉树之间千丝万缕的联系。

虽然函数命名没有什么硬性的要求，但我还是建议你也遵循我的这种风格，这样更能突出函数的作用和解题的思维模式，便于你自己理解和运用。

当时我是用二叉树的最大深度这个问题来举例，重点在于把这两种思路和动态规划和回溯算法进行对比，而本文的重点在于分析这两种思路如何解决二叉树的题目。

力扣第 104 题「二叉树的最大深度」就是最大深度的题目，所谓最大深度就是根节点到「最远」叶子节点的最长路径上的节点数，比如输入这棵二叉树，算法应该返回 3：



你做这题的思路是什么？显然遍历一遍二叉树，用一个外部变量记录每个节点所在的深度，取最大值就可以得到最大深度，这就是遍历二叉树计算答案的思路。

解法代码如下：

```
class Solution {
    // 记录最大深度
    int res = 0;

    // 记录遍历到的节点的深度
    int depth = 0;

    public int maxDepth(TreeNode root) {
        traverse(root);
        return res;
    }

    void traverse(TreeNode node) {
        if (node == null) {
            return;
        }
        depth++;
        maxDepth(node.left);
        maxDepth(node.right);
        res = Math.max(res, depth);
        depth--;
    }
}
```

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    depth++;
    if (root.left == null && root.right == null) {
        // 到达叶子节点，更新最大深度
        res = Math.max(res, depth);
    }
    traverse(root.left);
    traverse(root.right);
    // 后序位置
    depth--;
}
}
```

---

▶ 🎨 代码可视化动画

这个解法应该很好理解，但为什么需要在前序位置增加 `depth`，在后序位置减小 `depth`？

因为前面说了，前序位置是进入一个节点的时候，后序位置是离开一个节点的时候，`depth` 记录当前递归到的节点深度，你把 `traverse` 理解成在二叉树上游走的一个指针，所以当然要这样维护。

至于对 `res` 的更新，你放到前中后序位置都可以，只要保证在进入节点之后，离开节点之前（即 `depth` 自增之后，自减之前）就行了。

当然，你也很容易发现一棵二叉树的最大深度可以通过子树的最大深度推导出来，这就是分解问题计算答案的思路。

解法代码如下：

```
class Solution {
    // 定义：输入根节点，返回这棵二叉树的最大深度
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 利用定义，计算左右子树的最大深度
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 整棵树的最大深度等于左右子树的最大深度取最大值，
        // 然后再加上根节点自己
        int res = Math.max(leftMax, rightMax) + 1;

        return res;
    }
}
```

---

▶ 😊 代码可视化动画

只要明确递归函数的定义，这个解法也不难理解，但为什么主要的代码逻辑集中在后序位置？

因为这个思路正确的核心在于，你确实可以通过子树的最大深度推导出原树的深度，所以当然要首先利用递归函数的定义算出左右子树的最大深度，然后推出原树的最大深度，主要逻辑自然放在后序位置。

如果你理解了最大深度这个问题的两种思路，那么我们再回头看看最基本的二叉树前中后序遍历，就比如力扣第 144 题「二叉树的前序遍历」，让你计算前序遍历结果。

我们熟悉的解法就是用「遍历」的思路，我想应该没什么好说的：

```
class Solution {
    // 存放前序遍历结果
    List<Integer> res = new LinkedList<>();

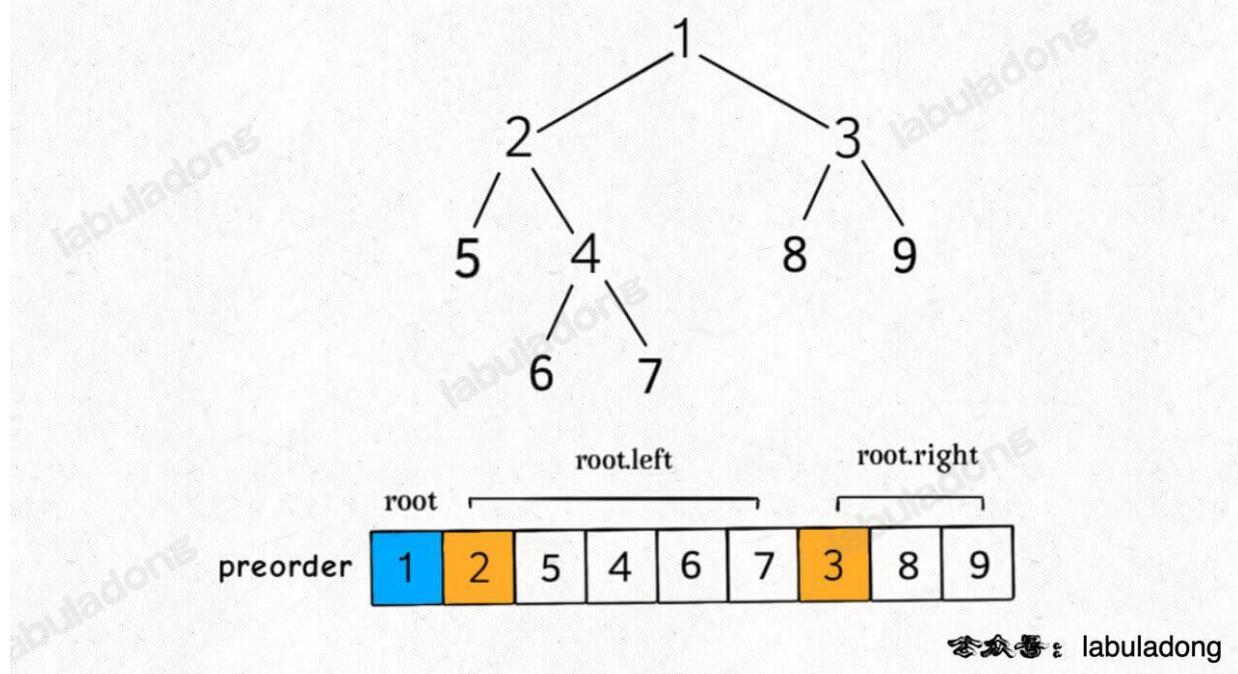
    // 返回前序遍历结果
    public List<Integer> preorderTraversal(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        // 前序位置
        res.add(root.val);
        traverse(root.left);
        traverse(root.right);
    }
}
```

但你是否能够用「分解问题」的思路，来计算前序遍历的结果？

换句话说，不要用像 `traverse` 这样的辅助函数和任何外部变量，单纯用题目给的 `preorderTraversal` 函数递归解题，你会不会？

我们知道前序遍历的特点是，根节点的值排在首位，接着是左子树的前序遍历结果，最后是右子树的前序遍历结果：



那这不就可以分解问题了么，一棵二叉树的前序遍历结果 = 根节点 + 左子树的前序遍历结果 + 右子树的前序遍历结果。

所以，你可以这样实现前序遍历算法：

```
class Solution {
    // 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
    List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new LinkedList<>();
        if (root == null) {
            return res;
        }
        // 前序遍历的结果，root.val 在第一个
        res.add(root.val);
        // 利用函数定义，后面接着左子树的前序遍历结果
        res.addAll(preorderTraversal(root.left));
        // 利用函数定义，最后接着右子树的前序遍历结果
        res.addAll(preorderTraversal(root.right));
        return res;
    }
}
```

中序和后序遍历也是类似的，只要把 `add(root.val)` 放到中序和后序对应的位置就行了。

这个解法短小精干，但为什么不常见呢？

一个原因是这个算法的复杂度不好把控，比较依赖语言特性。

Java 的话无论 ArrayList 还是 LinkedList，`addAll` 方法的复杂度都是  $O(N)$ ，所以总体的最坏时间复杂度会达到  $O(N^2)$ ，除非你自己实现一个复杂度为  $O(1)$  的 `addAll` 方法，底层用链表的话是可以做到的，因为多条链表只要简单的指针操作就能连接起来。

当然，最主要的原因还是因为教科书上从来没有这么教过……

上文举了两个简单的例子，但还有不少二叉树的题目是可以同时使用两种思路来思考和求解的，这就要靠你自己多去练习和思考，不要仅仅满足于一种熟悉的解法思路。

综上，遇到一道二叉树的题目时的通用思考过程是：

- 1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现。
- 2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值。
- 3、无论使用哪一种思维模式，你都要明白二叉树的每一个节点需要做什么，需要在什么时候（前中后序）做。

本站 [二叉树递归专项练习](#) 中列举了 100 多道二叉树习题，完全使用上述两种思维模式手把手带你练习，助你完全掌握递归思维，更容易理解高级的算法。

## 后序位置的特殊之处

说后序位置之前，先简单说下前序和中序。

前序位置本身其实没有什么特别的性质，之所以你发现好像很多题都是在前序位置写代码，实际上是因为我们习惯把那些对前中后序位置不敏感的代码写在前序位置罢了。

中序位置主要用在 BST 场景中，你完全可以把 BST 的中序遍历认为是遍历有序数组。

仔细观察，前中后序位置的代码，能力依次增强。

前序位置的代码只能从函数参数中获取父节点传递来的数据。

中序位置的代码不仅可以获取参数数据，还可以获取到左子树通过函数返回值传递回来的数据。

后序位置的代码最强，不仅可以获取参数数据，还可以同时获取到左右子树通过函数返回值传递回来的数据。

所以，某些情况下把代码移到后序位置效率最高；有些事情，只有后序位置的代码能做。

举些具体的例子来感受下它们的能力区别。现在给你一棵二叉树，我问你两个简单的问题：

- 1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
- 2、如何打印出每个节点的左右子树各有多少节点？

第一个问题可以这样写代码：

```
// 二叉树遍历函数
void traverse(TreeNode root, int level) {
    if (root == null) {
        return;
    }
    // 前序位置
    printf("Node %s at level %d", root.val, level);
    traverse(root.left, level + 1);
    traverse(root.right, level + 1);
}

// 这样调用
traverse(root, 1);
```

第二个问题可以这样写代码：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置
    printf("节点 %s 的左子树有 %d 个节点，右子树有 %d 个节点",
           root, leftCount, rightCount);

    return leftCount + rightCount + 1;
}
```

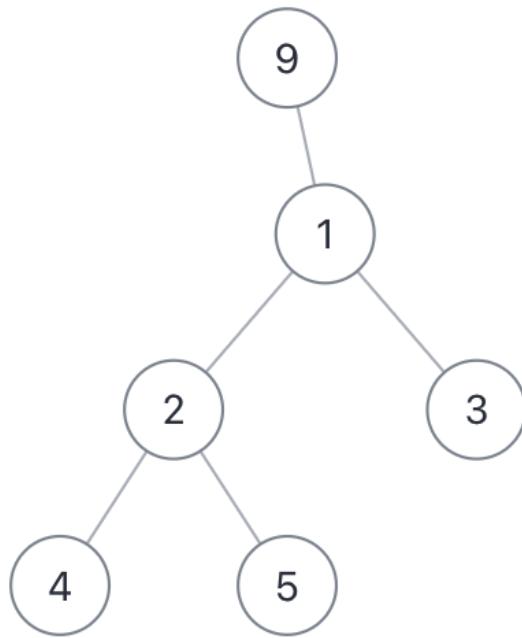
一个节点在第几层，你从根节点遍历过来的过程就能顺带记录，用递归函数的参数就能传递下去；而以一个节点为根的整棵树有多少个节点，你必须遍历完子树之后才能数清楚，然后通过递归函数的返回值拿到答案。

结合这两个简单的问题，你品味一下后序位置的特点，只有后序位置才能通过返回值获取子树的信息。

那么换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了。

接下来看下后序位置是如何在实际的题目中发挥作用的，简单聊下力扣第 543 题「二叉树的直径」，让你计算一棵二叉树的最长直径长度。

所谓二叉树的「直径」长度，就是任意两个结点之间的路径长度。最长「直径」并不一定要穿过根结点，比如下面这棵二叉树：



它的最长直径是 3，即 [4,2,1,3], [4,2,1,9] 或者 [5,2,1,3] 这几条「直径」的长度。

解决这题的关键在于，每一条二叉树的「直径」长度，就是一个节点的左右子树的最大深度之和。

现在让我求整棵树中的最长「直径」，那直截了当的思路就是遍历整棵树中的每个节点，然后通过每个节点的左右子树的最大深度算出每个节点的「直径」，最后把所有「直径」求个最大值即可。

最大深度的算法我们刚才实现过了，上述思路就可以写出以下代码：

```
class Solution {
    // 记录最大直径的长度
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        // 对每个节点计算直径，求最大直径
        traverse(root);
        return maxDiameter;
    }

    // 遍历二叉树
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        // 对每个节点计算直径
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        int myDiameter = leftMax + rightMax;
        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, myDiameter);

        traverse(root.left);
        traverse(root.right);
    }

    // 计算二叉树的最大深度
    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        return 1 + Math.max(leftMax, rightMax);
    }
}
```

这个解法是正确的，但是运行时间很长，原因也很明显，`traverse` 遍历每个节点的时候还会调用递归函数 `maxDepth`，而 `maxDepth` 是要遍历子树的所有节点的，所以最坏时间复杂度是  $O(N^2)$ 。

这就出现了刚才探讨的情况，前序位置无法获取子树信息，所以只能让每个节点调用 `maxDepth` 函数去算子树的深度。

那如何优化？我们应该把计算「直径」的逻辑放在后序位置，准确说应该是放在 `maxDepth` 的后序位置，因为 `maxDepth` 的后序位置是知道左右子树的最大深度的。

所以，稍微改一下代码逻辑即可得到更好的解法：

```
class Solution {
    // 记录最大直径的长度
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }
```

```
}

int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 后序位置，顺便计算最大直径
    int myDiameter = leftMax + rightMax;
    maxDiameter = Math.max(maxDiameter, myDiameter);

    return 1 + Math.max(leftMax, rightMax);
}
}
```

### ▶ 😊 代码可视化动画😊

这下时间复杂度只有 `maxDepth` 函数的  $O(N)$  了。

讲到这里，照应一下前文：遇到子树问题，首先想到的是给函数设置返回值，然后在后序位置做文章。

思考题：请你思考一下，运用后序遍历的题目使用的是「遍历」的思路还是「分解问题」的思路？

利用后序位置的题目，一般都使用「分解问题」的思路。因为当前节点接收并利用了子树返回的信息，这就意味着你把原问题分解成了当前节点 + 左右子树的子问题。

反过来，如果你写出了类似一开始的那种递归套递归的解法，大概率也需要反思是不是可以通过后序遍历优化了。

更多利用后序位置的习题参见 [手把手带你刷二叉树（后序篇）](#)、[手把手带你刷二叉搜索树（后序篇）](#) 和 [【强化练习】利用后序位置解题](#)。

## 以树的视角看动归/回溯/DFS算法的区别和联系

前文我说动态规划/回溯算法就是二叉树算法两种不同思路的表现形式，相信能看到这里的读者应该也认可了我这个观点。但有细心的读者经常提问：你的思考方法让我豁然开朗，但你好像一直没讲过 DFS 算法？

其实我在[一文秒杀所有岛屿题目](#)中就是用的 DFS 算法，但我确实没有单独用一篇文章讲 DFS 算法，因为 **DFS 算法和回溯算法非常类似，只是在细节上有所区别**。

这个细节上的差别是什么呢？其实就是「做选择」和「撤销选择」到底在 for 循环外面还是里面的区别，DFS 算法在外面，回溯算法在里面。

为什么有这个区别？还是要结合着二叉树理解。这一部分我就把回溯算法、DFS 算法、动态规划三种经典的算法思想，以及它们和二叉树算法的联系和区别，用一句话来说明：

动归/DFS/回溯算法都可以看做二叉树问题的扩展，只是它们的关注点不同：

- 动态规划算法属于分解问题（分治）的思路，它的关注点在整棵「子树」。
- 回溯算法属于遍历的思路，它的关注点在节点间的「树枝」。
- DFS 算法属于遍历的思路，它的关注点在单个「节点」。

怎么理解？我分别举三个例子你就懂了。

## 例子一：分解问题的思想体现

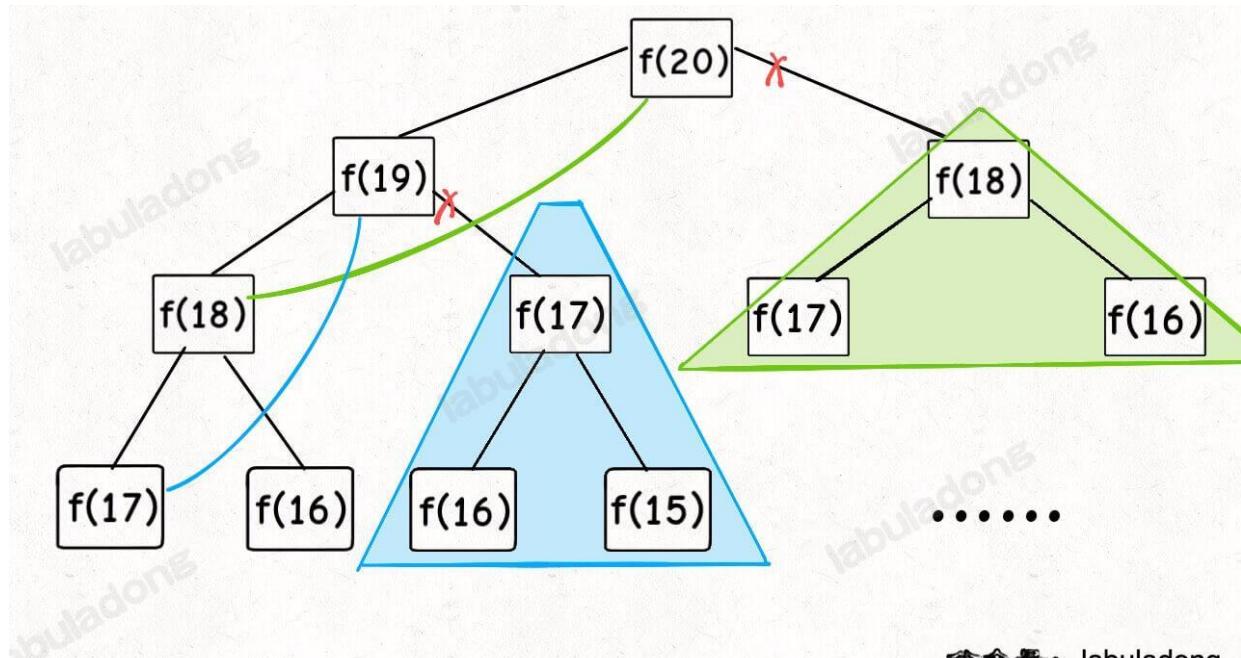
第一个例子，给你一棵二叉树，请你用分解问题的思路写一个 `count` 函数，计算这棵二叉树共有多少个节点。代码很简单，上文都写过了：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 当前节点关心的是两个子树的节点总数分别是多少
    // 因为子问题的结果可以推导出原问题的结果
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置，左右子树节点数加上自己就是整棵树的节点数
    return leftCount + rightCount + 1;
}
```

你看，这就是动态规划分解问题的思路，它的着眼点永远是结构相同的整个子问题，类比到二叉树上就是「子树」。

你再看看具体的动态规划问题，比如 [动态规划框架套路详解](#) 中举的斐波那契的例子，我们的关注点在一棵棵子树的返回值上：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```



## 例子二：回溯算法的思想体现

第二个例子，给你一棵二叉树，请你用遍历的思路写一个 `traverse` 函数，打印出遍历这棵二叉树的过程，你看下代码：

```
void traverse(TreeNode root) {
    if (root == null) return;
    printf("从节点 %s 进入节点 %s", root, root.left);
    traverse(root.left);
    printf("从节点 %s 回到节点 %s", root.left, root);

    printf("从节点 %s 进入节点 %s", root, root.right);
    traverse(root.right);
    printf("从节点 %s 回到节点 %s", root.right, root);
}
```

不难理解吧，好的，我们现在从二叉树进阶成多叉树，代码也是类似的：

```
// 多叉树节点
class Node {
    int val;
    Node[] children;
}

void traverse(Node root) {
    if (root == null) return;
    for (Node child : root.children) {
        printf("从节点 %s 进入节点 %s", root, child);
        traverse(child);
        printf("从节点 %s 回到节点 %s", child, root);
    }
}
```

这个多叉树的遍历框架就可以延伸出 [回溯算法框架套路详解](#) 中的回溯算法框架：

```
// 回溯算法框架
void backtrack(...) {
    // base case
    if (...) return;

    for (int i = 0; i < ...; i++) {
        // 做选择
        ...

        // 进入下一层决策树
        backtrack(...);

        // 撤销刚才做的选择
        ...
    }
}
```

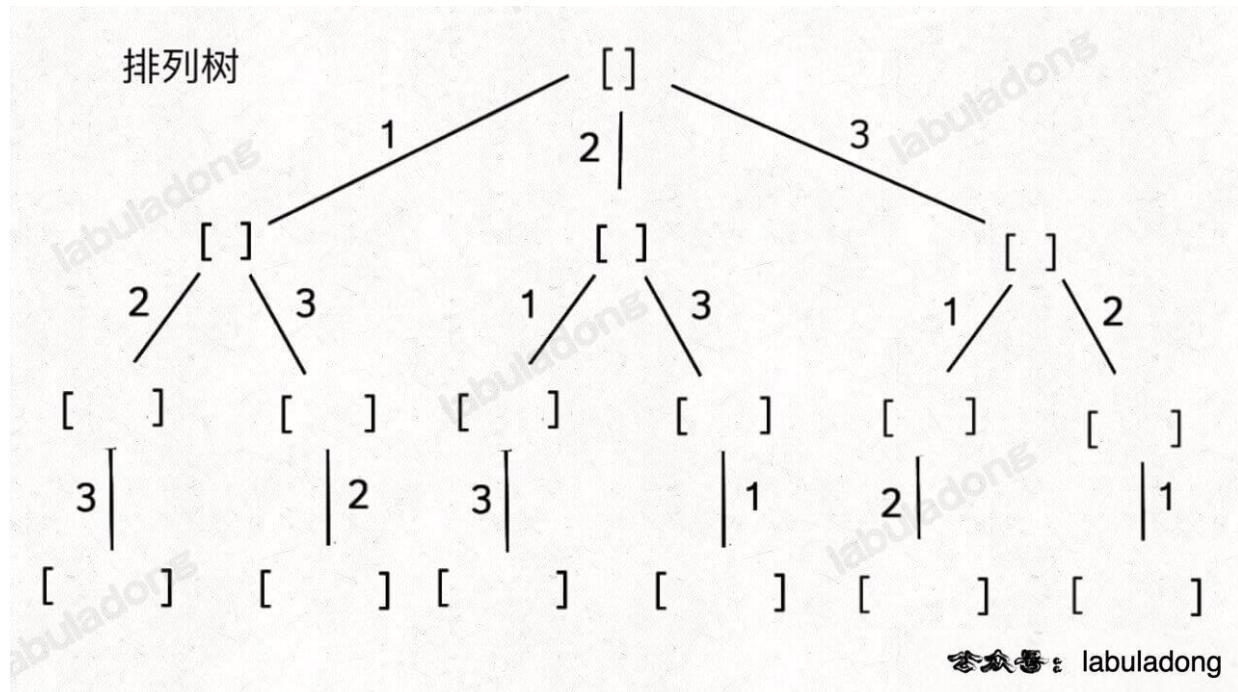
你看，这就是回溯算法遍历的思路，它的着眼点永远是在节点之间移动的过程，类比到二叉树上就是「树枝」。

你再看看具体的回溯算法问题，比如[回溯算法秒杀排列组合子集的九种题型](#)中讲到的全排列，我们的关注点在一条条树枝上：

```
// 回溯算法核心部分代码
void backtrack(int[] nums) {
    // 回溯算法框架
    for (int i = 0; i < nums.length; i++) {
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        // 进入下一层回溯树
        backtrack(nums);

        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}
```



### 例子三：DFS 的思想体现

第三个例子，我给你一棵二叉树，请你写一个 `traverse` 函数，把这棵二叉树上的每个节点的值都加一。很简单吧，代码如下：

```
void traverse(TreeNode root) {
    if (root == null) return;
    // 遍历过的每个节点的值加一
    root.val++;
    traverse(root.left);
    traverse(root.right);
}
```

你看，这就是 DFS 算法遍历的思路，它的着眼点永远是在单一的节点上，类比到二叉树上就是处理每个「节点」。

你再看看具体的 DFS 算法问题，比如 [一文秒杀所有岛屿题目](#) 中讲的前几道题，我们的关注点是 `grid` 数组的每个格子（节点），我们要对遍历过的格子进行一些处理，所以我说是用 DFS 算法解决这几道题的：

```
// DFS 算法核心逻辑
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 0) {
        return;
    }
    // 遍历过的每个格子标记为 0
    grid[i][j] = 0;
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

1	1	0	1	1
1	0	0	0	0
0	0	0	0	1
1	1	0	1	1

好，请你仔细品一下上面三个简单的例子，是不是像我说的：动态规划关注整棵「子树」，回溯算法关注节点间的「树枝」，DFS 算法关注单个「节点」。

有了这些铺垫，你就很容易理解为什么回溯算法和 DFS 算法代码中「做选择」和「撤销选择」的位置不同了，看下面两段代码：

```
// DFS 算法把「做选择」「撤销选择」的逻辑放在 for 循环外面
void dfs(Node root) {
    if (root == null) return;
    // 做选择
    print("我已经进入节点 %s 啦", root);
    for (Node child : root.children) {
        dfs(child);
    }
    // 撤销选择
```

```
    print("我将要离开节点 %s 啦", root);
}

// 回溯算法把「做选择」「撤销选择」的逻辑放在 for 循环里面
void backtrack(Node root) {
    if (root == null) return;
    for (Node child : root.children) {
        // 做选择
        print("我站在节点 %s 到节点 %s 的树枝上", root, child);
        backtrack(child);
        // 撤销选择
        print("我将要离开节点 %s 到节点 %s 的树枝上", child, root);
    }
}
```

看到了吧，你回溯算法必须把「做选择」和「撤销选择」的逻辑放在 for 循环里面，否则怎么拿到「树枝」的两个端点？

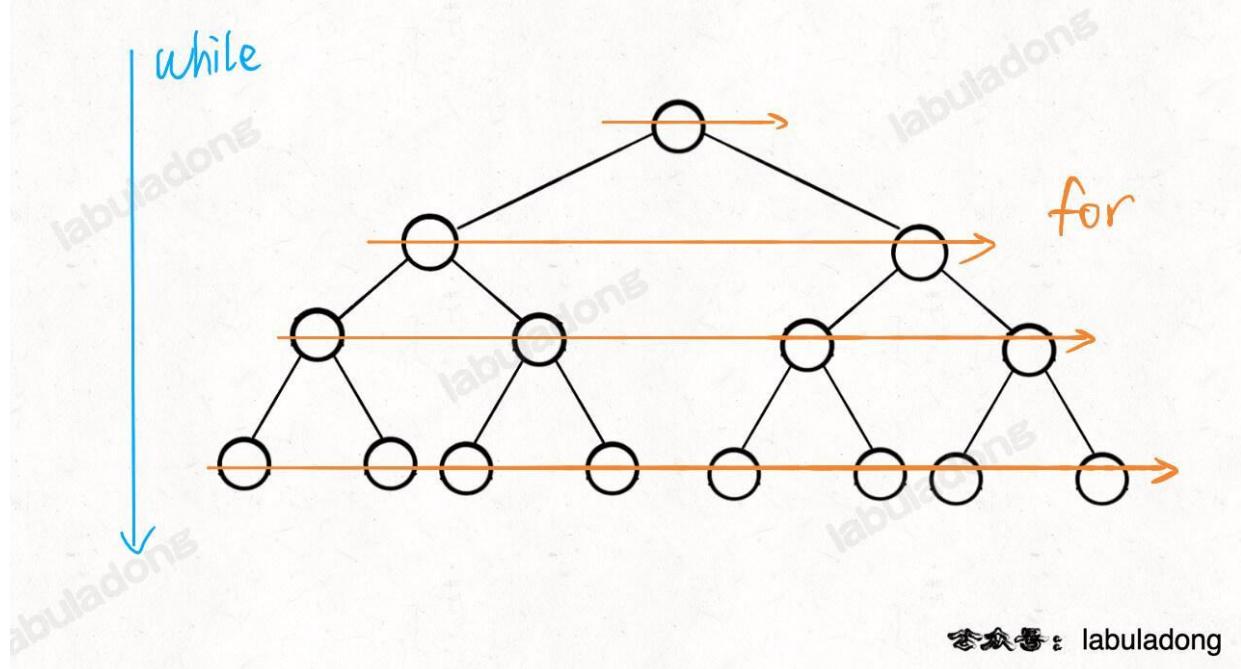
## 层序遍历

二叉树题型主要是用来培养递归思维的，而层序遍历属于迭代遍历，也比较简单，这里就过一下代码框架吧：

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
    }
}
```

这里面 while 循环和 for 循环分管从上到下和从左到右的遍历：



后文 [BFS 算法框架](#) 就是从二叉树的层序遍历扩展出来的，常用于求无权图的最短路径问题。

当然这个框架还可以灵活修改，题目不需要记录层数（步数）时可以去掉上述框架中的 for 循环，比如后文 [Dijkstra 算法](#) 中计算加权图的最短路径问题，详细探讨了 BFS 算法的扩展。

值得一提的是，有些很明显需要用层序遍历技巧的二叉树的题目，也可以用递归遍历的方式去解决，而且技巧性会更强，非常考察你对前中后序的把控。

好了，本文已经够长了，围绕前中后序位置算是把二叉树题目里的各种套路给讲透了，真正能运用出来多少，就需要你亲自刷题实践和思考了。

希望大家能探索尽可能多的解法，只要参透二叉树这种基本数据结构的原理，那么就很容易在学习其他高级算法的道路上找到抓手，打通回路，形成闭环（手动狗头）。

最后，[二叉树递归专项练习](#) 中会手把手带你运用本文所讲的技巧。

## 回答评论区的问题

关于层序遍历（以及其扩展出的 [BFS 算法框架](#)），我在最后多说几句。

如果你对二叉树足够熟悉，可以想到很多方式通过递归函数得到层序遍历结果，比如下面这种写法：

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();

    public List<List<Integer>> levelTraverse(TreeNode root) {
        if (root == null) {
            return res;
        }
        // root 视为第 0 层
        traverse(root, 0);
        return res;
    }

    void traverse(TreeNode root, int depth) {
        if (root == null) {
            return;
        }
        if (res.size() <= depth) {
            res.add(new ArrayList<Integer>());
        }
        res.get(depth).add(root.val);
        traverse(root.left, depth + 1);
        traverse(root.right, depth + 1);
    }
}
```

```
        return;
    }
    // 前序位置，看看是否已经存储 depth 层的节点了
    if (res.size() <= depth) {
        // 第一次进入 depth 层
        res.add(new LinkedList<>());
    }
    // 前序位置，在 depth 层添加 root 节点的值
    res.get(depth).add(root.val);
    traverse(root.left, depth + 1);
    traverse(root.right, depth + 1);
}
}
```

这种思路从结果上说确实可以得到层序遍历结果，但其本质还是二叉树的前序遍历，或者说 DFS 的思路，而不是层序遍历，或者说 BFS 的思路。因为这个解法是依赖前序遍历自顶向下、自左向右的顺序特点得到了正确的结果。

**抽象点说，这个解法更像是从左到右的「列序遍历」，而不是自顶向下的「层序遍历」。**所以对于计算最小距离的场景，这个解法完全等同于 DFS 算法，没有 BFS 算法的性能的优势。

还有优秀读者评论了这样一种递归进行层序遍历的思路：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();

    public List<List<Integer>> levelTraverse(TreeNode root) {
        if (root == null) {
            return res;
        }
        List<TreeNode> nodes = new LinkedList<>();
        nodes.add(root);
        traverse(nodes);
        return res;
    }

    void traverse(List<TreeNode> curLevelNodes) {
        // base case
        if (curLevelNodes.isEmpty()) {
            return;
        }
        // 前序位置，计算当前层的值和下一层的节点列表
        List<Integer> nodeValues = new LinkedList<>();
        List<TreeNode> nextLevelNodes = new LinkedList<>();
        for (TreeNode node : curLevelNodes) {
            nodeValues.add(node.val);
            if (node.left != null) {
                nextLevelNodes.add(node.left);
            }
            if (node.right != null) {
                nextLevelNodes.add(node.right);
            }
        }
        // 前序位置添加结果，可以得到自顶向下的层序遍历
        res.add(nodeValues);
        traverse(nextLevelNodes);
        // 后序位置添加结果，可以得到自底向上的层序遍历结果
    }
}
```

```

        // res.add(nodeValues);
    }
}

```

这个 `traverse` 函数很像递归遍历单链表的函数，其实就是把二叉树的每一层抽象理解成单链表的一个节点进行遍历。

相较上一个递归解法，这个递归解法是自顶向下的「层序遍历」，更接近 BFS 的奥义，可以作为 BFS 算法的递归实现扩展一下思维。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
100. Same Tree	100. 相同的树	
1008. Construct Binary Search Tree from Preorder Traversal	1008. 前序遍历构造二叉搜索树	
101. Symmetric Tree	101. 对称二叉树	
1022. Sum of Root To Leaf Binary Numbers	1022. 从根到叶的二进制数之和	
1026. Maximum Difference Between Node and Ancestor	1026. 节点与其祖先之间的最大差值	
108. Convert Sorted Array to Binary Search Tree	108. 将有序数组转换为二叉搜索树	
1080. Insufficient Nodes in Root to Leaf Paths	1080. 根到叶路径上的不足节点	
110. Balanced Binary Tree	110. 平衡二叉树	
111. Minimum Depth of Binary Tree	111. 二叉树的最小深度	
1110. Delete Nodes And Return Forest	1110. 删点成林	
1120. Maximum Average Subtree	1120. 子树的最大平均值	
113. Path Sum II	113. 路径总和 II	
114. Flatten Binary Tree to Linked List	114. 二叉树展开为链表	
116. Populating Next Right Pointers in Each Node	116. 填充每个节点的下一个右侧节点指针	
124. Binary Tree Maximum Path Sum	124. 二叉树中的最大路径和	
1245. Tree Diameter	1245. 树的直径	
1261. Find Elements in a Contaminated Binary Tree	1261. 在受污染的二叉树中查找元素	
129. Sum Root to Leaf Numbers	129. 求根节点到叶节点数字之和	
1315. Sum of Nodes with Even-Valued Grandparent	1315. 祖父节点值为偶数的节点和	
1325. Delete Leaves With a Given Value	1325. 删除给定值的叶子节点	
1339. Maximum Product of Splitted Binary Tree	1339. 分裂二叉树的最大乘积	
1367. Linked List in Binary Tree	1367. 二叉树中的链表	
1372. Longest ZigZag Path in a Binary Tree	1372. 二叉树中的最长交错路径	

LeetCode	力扣	难度
1373. Maximum Sum BST in Binary Tree	1373. 二叉搜索子树的最大键值和	
1376. Time Needed to Inform All Employees	1376. 通知所有员工所需的时间	
1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree	1379. 找出克隆二叉树中的相同节点	
1430. Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree	1430. 判断给定的序列是否是二叉树从根到叶的路径	
1443. Minimum Time to Collect All Apples in a Tree	1443. 收集树上所有苹果的最少时间	
1448. Count Good Nodes in Binary Tree	1448. 统计二叉树中好节点的数目	
145. Binary Tree Postorder Traversal	145. 二叉树的后序遍历	
1457. Pseudo-Palindromic Paths in a Binary Tree	1457. 二叉树中的伪回文路径	
1469. Find All The Lonely Nodes	1469. 寻找所有的独生节点	
1485. Clone Binary Tree With Random Pointer	1485. 克隆含随机指针的二叉树	
1490. Clone N-ary Tree	1490. 克隆 N 叉树	
1593. Split a String Into the Max Number of Unique Substrings	1593. 拆分字符串使唯一子字符串的数目最大	
1602. Find Nearest Right Node in Binary Tree	1602. 找到二叉树中最近的右侧节点	
1612. Check If Two Expression Trees are Equivalent	1612. 检查两棵二叉表达式树是否等价	
1740. Find Distance in a Binary Tree	1740. 找到二叉树中的距离	
2049. Count Nodes With the Highest Score	2049. 统计最高分的节点数目	
2096. Step-By-Step Directions From a Binary Tree Node to Another	2096. 从二叉树一个节点到另一个节点每一步的方向	
226. Invert Binary Tree	226. 翻转二叉树	
250. Count Univalue Subtrees	250. 统计同值子树	
254. Factor Combinations	254. 因子的组合	
257. Binary Tree Paths	257. 二叉树的所有路径	
267. Palindrome Permutation II	267. 回文排列 II	
270. Closest Binary Search Tree Value	270. 最接近的二叉搜索树值	
294. Flip Game II	294. 翻转游戏 II	
298. Binary Tree Longest Consecutive Sequence	298. 二叉树最长连续序列	
332. Reconstruct Itinerary	332. 重新安排行程	
333. Largest BST Subtree	333. 最大 BST 子树	
339. Nested List Weight Sum	339. 嵌套列表权重和	
366. Find Leaves of Binary Tree	366. 寻找二叉树的叶子节点	

LeetCode	力扣	难度
386. Lexicographical Numbers	386. 字典序排数	🟠
404. Sum of Left Leaves	404. 左叶子之和	🟢
426. Convert Binary Search Tree to Sorted Doubly Linked List 🔒	426. 将二叉搜索树转化为排序的双向链表 🔒	🟠
437. Path Sum III	437. 路径总和 III	🟠
501. Find Mode in Binary Search Tree	501. 二叉搜索树中的众数	🟢
508. Most Frequent Subtree Sum	508. 出现次数最多的子树元素和	🟠
513. Find Bottom Left Tree Value	513. 找树左下角的值	🟠
515. Find Largest Value in Each Tree Row	515. 在每个树行中找最大值	🟠
530. Minimum Absolute Difference in BST	530. 二叉搜索树的最小绝对差	🟢
538. Convert BST to Greater Tree	538. 把二叉搜索树转换为累加树	🟠
549. Binary Tree Longest Consecutive Sequence II 🔒	549. 二叉树中最长的连续序列 🔒	🟠
559. Maximum Depth of N-ary Tree	559. N 叉树的最大深度	🟢
563. Binary Tree Tilt	563. 二叉树的坡度	🟢
572. Subtree of Another Tree	572. 另一棵树的子树	🟢
582. Kill Process 🔒	582. 杀掉进程 🔒	🟠
606. Construct String from Binary Tree	606. 根据二叉树创建字符串	🟢
617. Merge Two Binary Trees	617. 合并二叉树	🟢
623. Add One Row to Tree	623. 在二叉树中增加一行	🟠
654. Maximum Binary Tree	654. 最大二叉树	🟠
663. Equal Tree Partition 🔒	663. 均匀树划分 🔒	🟠
666. Path Sum IV 🔒	666. 路径总和 IV 🔒	🟠
669. Trim a Binary Search Tree	669. 修剪二叉搜索树	🟠
671. Second Minimum Node In a Binary Tree	671. 二叉树中第二小的节点	🟢
687. Longest Univalue Path	687. 最长同值路径	🟠
776. Split BST 🔒	776. 拆分二叉搜索树 🔒	🟠
865. Smallest Subtree with all the Deepest Nodes	865. 具有所有最深节点的最小子树	🟠
894. All Possible Full Binary Trees	894. 所有可能的真二叉树	🟠
897. Increasing Order Search Tree	897. 递增顺序搜索树	🟢
938. Range Sum of BST	938. 二叉搜索树的范围和	🟢
951. Flip Equivalent Binary Trees	951. 翻转等价二叉树	🟠
965. Univalued Binary Tree	965. 单值二叉树	🟢

LeetCode	力扣	难度
968. Binary Tree Cameras	968. 监控二叉树	🔴
971. Flip Binary Tree To Match Preorder Traversal	971. 翻转二叉树以匹配先序遍历	🟡
979. Distribute Coins in Binary Tree	979. 在二叉树中分配硬币	🟡
987. Vertical Order Traversal of a Binary Tree	987. 二叉树的垂序遍历	🔴
988. Smallest String Starting From Leaf	988. 从叶结点开始的最小字符串	🟡
99. Recover Binary Search Tree	99. 恢复二叉搜索树	🟡
993. Cousins in Binary Tree	993. 二叉树的堂兄弟节点	🟢
998. Maximum Binary Tree II	998. 最大二叉树 II	🟡
-	剑指 Offer 06. 从尾到头打印链表	🟢
-	剑指 Offer 26. 树的子结构	🟡
-	剑指 Offer 27. 二叉树的镜像	🟢
-	剑指 Offer 28. 对称的二叉树	🟢
-	剑指 Offer 33. 二叉搜索树的后序遍历序列	🟡
-	剑指 Offer 34. 二叉树中和为某一值的路径	🟡
-	剑指 Offer 36. 二叉搜索树与双向链表	🟡
-	剑指 Offer 55 - I. 二叉树的深度	🟢
-	剑指 Offer 55 - II. 平衡二叉树	🟢
-	剑指 Offer II 044. 二叉树每层的最大值	🟡
-	剑指 Offer II 045. 二叉树最底层最左边的值	🟡
-	剑指 Offer II 049. 从根节点到叶节点的路径数字之和	🟡
-	剑指 Offer II 050. 向下的路径节点之和	🟡
-	剑指 Offer II 051. 节点之和最大的路径	🔴
-	剑指 Offer II 052. 展平二叉搜索树	🟢
-	剑指 Offer II 054. 所有大于等于节点的值之和	🟡

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二叉树心法（思路篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">226. Invert Binary Tree</a>	<a href="#">226. 翻转二叉树</a>	
<a href="#">116. Populating Next Right Pointers in Each Node</a>	<a href="#">116. 填充每个节点的下一个右侧节点指针</a>	
<a href="#">114. Flatten Binary Tree to Linked List</a>	<a href="#">114. 二叉树展开为链表</a>	

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历
- 二叉树心法（纲领篇）

tip：本文有视频版：[二叉树/递归的框架思维（纲领篇）](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文承接 [二叉树心法（纲领篇）](#)，先复述一下前文总结的二叉树解题总纲：

二叉树解题的思维模式分两类：

- 1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现，这叫「遍历」的思维模式。
- 2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「分解问题」的思维模式。

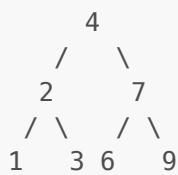
无论使用哪种思维模式，你都需要思考：

**如果单独抽出一个二叉树节点，它需要做什么事情？需要在什么时候（前/中/后序位置）做？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。**

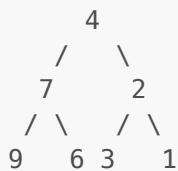
本文就以几道比较简单的题目为例，带你实践运用这几条总纲，理解「遍历」的思维和「分解问题」的思维有何区别和联系。

### 第一题、翻转二叉树

我们先从简单的题开始，看看力扣第 226 题「翻转二叉树」，输入一个二叉树根节点 `root`，让你把整棵树镜像翻转，比如输入的二叉树如下：



算法原地翻转二叉树，使得以 `root` 为根的树变成：



不难发现，只要把二叉树上的每一个节点的左右子节点进行交换，最后的结果就是完全翻转之后的二叉树。

那么现在开始在心中默念二叉树解题总纲：

### 1、这题能不能用「遍历」的思维模式解决？

可以，我写一个 `traverse` 函数遍历每个节点，让每个节点的左右子节点颠倒过来就行了。

单独抽出一个节点，需要让它做什么？让它把自己的左右子节点交换一下。

需要在什么时候做？好像前中后序位置都可以。

综上，可以写出如下解法代码：

```
class Solution {
    // 主函数
    public TreeNode invertTree(TreeNode root) {
        // 遍历二叉树，交换每个节点的子节点
        traverse(root);
        return root;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // *** 前序位置 ***
        // 每一个节点需要做的事就是交换它的左右子节点
        TreeNode tmp = root.left;
        root.left = root.right;
        root.right = tmp;

        // 遍历框架，去遍历左右子树的节点
        traverse(root.left);
        traverse(root.right);
    }
}
```

► 🎨 代码可视化动画🎃

你把前序位置的代码移到后序位置也可以，但是直接移到中序位置是不行的，需要稍作修改，这应该很容易看出来吧，我就不说了。

按理说，这道题已经解决了，不过为了对比，我们再继续思考下去。

## 2、这题能不能用「分解问题」的思维模式解决？

我们尝试给 `invertTree` 函数赋予一个定义：

```
// 定义：将以 root 为根的这棵二叉树翻转，返回翻转后的二叉树的根节点
TreeNode invertTree(TreeNode root);
```

然后思考，对于某一个二叉树节点 `x` 执行 `invertTree(x)`，你能利用这个递归函数的定义做点啥？

我可以先用 `invertTree(x.left)` 把 `x` 的左子树翻转，再用 `invertTree(x.right)` 把 `x` 的右子树翻转，最后把 `x` 的左右子树交换，这恰好完成了以 `x` 为根的整棵二叉树的翻转，即完成了 `invertTree(x)` 的定义。

直接写出解法代码：

```
class Solution {
    // 定义：将以 root 为根的这棵二叉树翻转，返回翻转后的二叉树的根节点
    public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }
        // 利用函数定义，先翻转左右子树
        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);

        // 然后交换左右子节点
        root.left = right;
        root.right = left;

        // 和定义逻辑自洽：以 root 为根的这棵二叉树已经被翻转，返回 root
        return root;
    }
}
```

► 🎃 代码可视化动画🎃

这种「分解问题」的思路，核心在于你要给递归函数一个合适的定义，然后用函数的定义来解释你的代码；如果你的逻辑成功自洽，那么说明你这个算法是正确的。

好了，这道题就分析到这，「遍历」和「分解问题」的思路都可以解决，看下一道题。

## 第二题、填充节点的右侧指针

这是力扣第 116 题「填充每个二叉树节点的右侧指针」，看下题目：

▼ 116. 填充每个节点的下一个右侧节点指针 [Leetcode | 力扣](#)

给定一个 完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 **NULL**。

初始状态下，所有 next 指针都被设置为 **NULL**。

#### 示例 1：

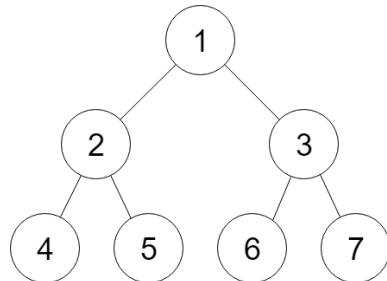


Figure A

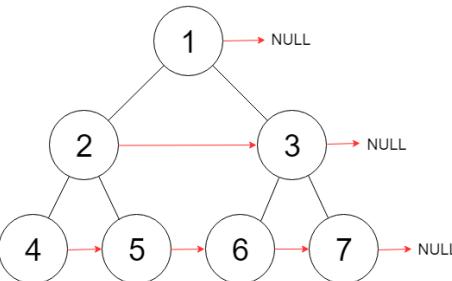


Figure B

输入：root = [1,2,3,4,5,6,7]  
输出：[1,#,2,3,#,4,5,6,7,#]

解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化的输出按层序遍历排列，同一层节点由 next 指针连接，% % % % # % % % % % 标志着每一层的结束。

#### 示例 2：

输入：root = []  
输出：[]

提示：

- 树中节点的数量在  $[0, 2^{12} - 1]$  范围内
- $-1000 \leq \text{node.val} \leq 1000$

进阶：

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

```
// 函数签名  
Node connect(Node root);
```

题目的意思就是把二叉树的每一层节点都用 **next** 指针连接起来：

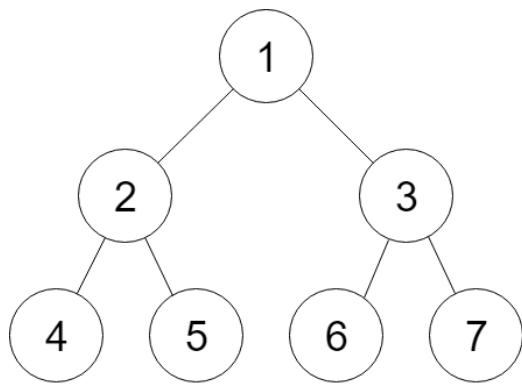


Figure A

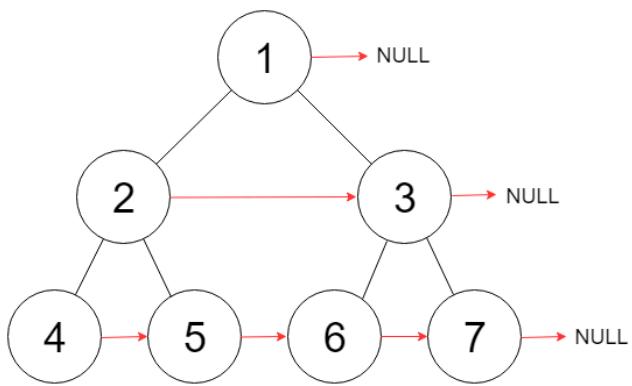


Figure B

而且题目说了，输入是一棵「完美二叉树」，形象地说整棵二叉树是一个正三角形，除了最右侧的节点 `next` 指针会指向 `null`，其他节点的右侧一定有相邻的节点。

这道题怎么做呢？来默念二叉树解题总纲：

### 1、这题能不能用「遍历」的思维模式解决？

很显然，一定可以。

每个节点要做的事也很简单，把自己的 `next` 指针指向右侧节点就行了。

也许你会模仿上一道题，直接写出如下代码：

```
// 二叉树遍历函数
void traverse(Node root) {
    if (root == null || root.left == null) {
        return;
    }
    // 把左子节点的 next 指针指向右子节点
    root.left.next = root.right;

    traverse(root.left);
    traverse(root.right);
}
```

但是，这段代码其实有很大问题，因为它只能把相同父节点的两个节点穿起来，再看看这张图：

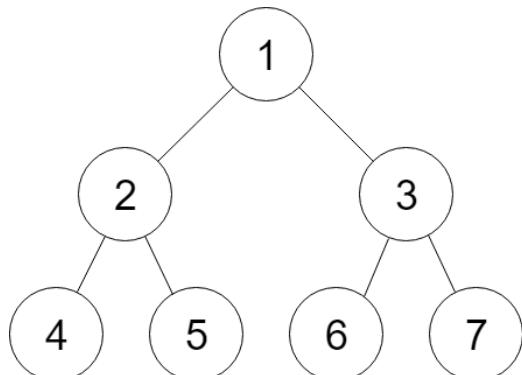


Figure A

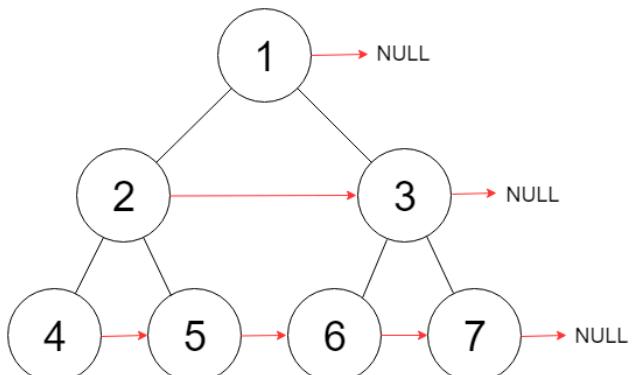
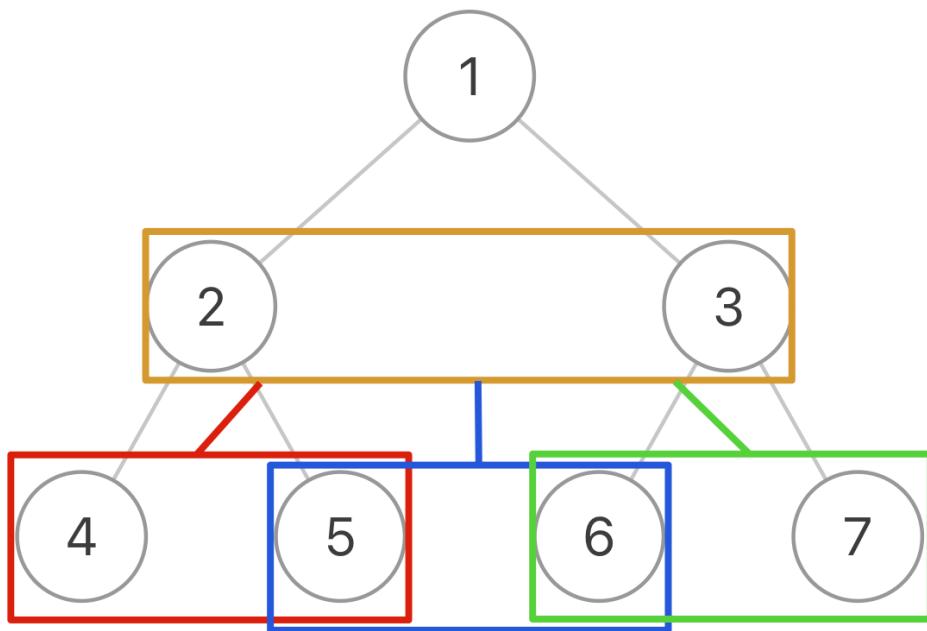


Figure B

节点 5 和节点 6 不属于同一个父节点，那么按照这段代码的逻辑，它俩就没办法被穿起来，这是不符合题意的，但是问题出在哪里？

传统的 `traverse` 函数是遍历二叉树的所有节点，但现在我们想遍历的其实是两个相邻节点之间的「空隙」。

所以我们可以把二叉树的基础上进行抽象，你把图中的每一个方框看做一个节点：



这样，一棵二叉树被抽象成了一棵三叉树，三叉树上的每个节点就是原先二叉树的两个相邻节点。

现在，我们只要实现一个 `traverse` 函数来遍历这棵三叉树，每个「三叉树节点」需要做的事就是把自己内部的两个二叉树节点穿起来：

```
class Solution {
    // 主函数
    public Node connect(Node root) {
        if (root == null) return null;
        // 遍历「三叉树」，连接相邻节点
        traverse(root.left, root.right);
        return root;
    }

    // 三叉树遍历框架
    void traverse(Node node1, Node node2) {
        if (node1 == null || node2 == null) {
            return;
        }
        // *** 前序位置 ***
        // 将传入的两个节点穿起来
        node1.next = node2;

        // 连接相同父节点的两个子节点
        traverse(node1.left, node1.right);
        traverse(node2.left, node2.right);
        // 连接跨越父节点的两个子节点
        traverse(node1.right, node2.left);
    }
}
```

这样，`traverse` 函数遍历整棵「三叉树」，将所有相邻节的二叉树节点都连接起来，也就避免了我们之前出现的问题，把这道题完美解决。

## 2、这题能不能用「分解问题」的思维模式解决？

嗯，好像没有什么特别好的思路，所以这道题无法使用「分解问题」的思维来解决。

## 第三题、将二叉树展开为链表

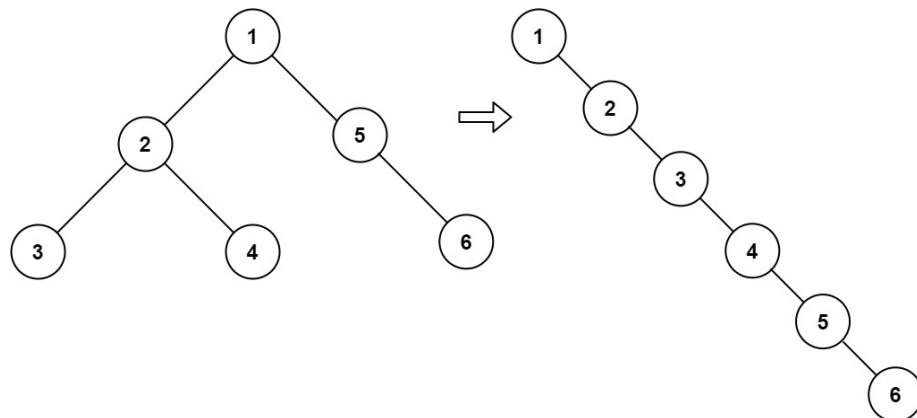
这是力扣第 114 题「将二叉树展开为链表」，看下题目：

### ▼ 114. 二叉树展开为链表 [Leetcode | 力扣](#)

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。
- 展开后的单链表应该与二叉树 [先序遍历](#) 顺序相同。

### 示例 1：



```
输入: root = [1,2,5,3,4,null,6]
输出: [1,null,2,null,3,null,4,null,5,null,6]
```

### 示例 2：

```
输入: root = []
输出: []
```

### 示例 3：

```
输入: root = [0]
输出: [0]
```

### 提示：

- 树中结点数在范围 `[0, 2000]` 内
- `-100 <= node.val <= 100`

进阶：你可以使用原地算法（`O(1)` 额外空间）展开这棵树吗？

```
// 函数签名如下
void flatten(TreeNode root);

for (int i = n - 2; i >= 0; i--)
    r_max[i] = Math.max(height[i], r_max[i + 1]);

for (int i = 1; i < n - 1; i++) {
    // hello world
    res += Math.min(l_max[i], r_max[i]) - height[i];
}
```

## 1、这题能不能用「遍历」的思维模式解决？

乍一看感觉是可以的：对整棵树进行前序遍历，一边遍历一边构造出一条「链表」就行了：

```
// 虚拟头节点, dummy.right 就是结果
TreeNode dummy = new TreeNode(-1);
// 用来构建链表的指针
TreeNode p = dummy;

void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    p.right = new TreeNode(root.val);
    p = p.right;

    traverse(root.left);
    traverse(root.right);
}
```

但是注意 `flatten` 函数的签名，返回类型为 `void`，也就是说题目希望我们在原地把二叉树拉平成链表。

这样一来，没办法通过简单的二叉树遍历来解决这道题了。

## 2、这题能不能用「分解问题」的思维模式解决？

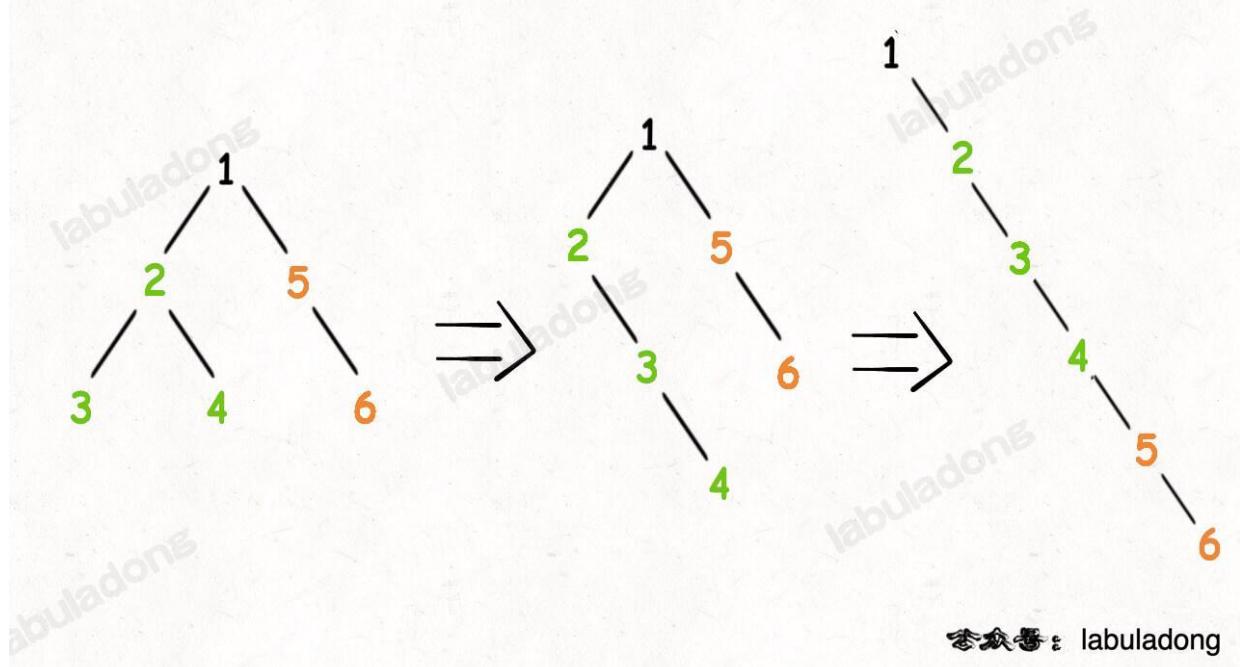
我们尝试给出 `flatten` 函数的定义：

```
// 定义：输入节点 root，然后 root 为根的二叉树就会被拉平为一条链表
void flatten(TreeNode root);
```

有了这个函数定义，如何按题目要求把一棵树拉平成一条链表？

对于一个节点 `x`，可以执行以下流程：

- 1、先利用 `flatten(x.left)` 和 `flatten(x.right)` 将 `x` 的左右子树拉平。
- 2、将 `x` 的右子树接到左子树下方，然后将整个左子树作为右子树。



这样，以  $x$  为根的整棵二叉树就被拉平了，恰好完成了  $\text{flatten}(x)$  的定义。

直接看代码实现：

```
class Solution {
    // 定义：将以 root 为根的树拉平为链表
    public void flatten(TreeNode root) {
        // base case
        if (root == null) return;

        // 利用定义，把左右子树拉平
        flatten(root.left);
        flatten(root.right);

        // *** 后序遍历位置 ***
        // 1、左右子树已经被拉平成一条链表
        TreeNode left = root.left;
        TreeNode right = root.right;

        // 2、将左子树作为右子树
        root.left = null;
        root.right = left;

        // 3、将原先的右子树接到当前右子树的末端
        TreeNode p = root;
        while (p.right != null) {
            p = p.right;
        }
        p.right = right;
    }
}
```

---

▶ 代码可视化动画

---

你看，这就是递归的魅力，你说  $\text{flatten}$  函数是怎么把左右子树拉平的？

不容易说清楚，但是只要知道 `flatten` 的定义如此并利用这个定义，让每一个节点做它该做的事情，然后 `flatten` 函数就会按照定义工作。

至此，这道题也解决了，我们前文 [k个一组翻转链表](#) 的递归思路和本题也有一些类似。

最后，首尾呼应，再次默写二叉树解题总纲。

二叉树解题的思维模式分两类：

1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现，这叫「遍历」的思维模式。

2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「分解问题」的思维模式。

无论使用哪种思维模式，你都需要思考：

如果单独抽出一个二叉树节点，它需要做什么事情？需要在什么时候（前/中/后序位置）做？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。

希望你能仔细体会，并运用到所有二叉树题目上。

本文就到这里，更多经典的二叉树习题以及递归思维的训练，请参见二叉树章节中的 [递归专项练习](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer 26. 树的子结构</a>	
-	<a href="#">剑指 Offer 27. 二叉树的镜像</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二叉树心法（构造篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">105. Construct Binary Tree from Preorder and Inorder Traversal</a>	<a href="#">105. 从前序与中序遍历序列构造二叉树</a>	
<a href="#">106. Construct Binary Tree from Inorder and Postorder Traversal</a>	<a href="#">106. 从中序与后序遍历序列构造二叉树</a>	
<a href="#">654. Maximum Binary Tree</a>	<a href="#">654. 最大二叉树</a>	
<a href="#">889. Construct Binary Tree from Preorder and Postorder Traversal</a>	<a href="#">889. 根据前序和后序遍历构造二叉树</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)
- [二叉树心法（纲领篇）](#)

本文是承接 [二叉树心法（纲领篇）](#) 的第二篇文章，先复述一下前文总结的二叉树解题总纲：

二叉树解题的思维模式分两类：

1、是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 `traverse` 函数配合外部变量来实现，这叫「遍历」的思维模式。

2、是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「分解问题」的思维模式。

无论使用哪种思维模式，你都需要思考：

如果单独抽出一个二叉树节点，它需要做什么事情？需要在什么时候（前/中/后序位置）做？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。

第一篇文章 [二叉树心法（思维篇）](#) 讲了「遍历」和「分解问题」两种思维方式，本文讲二叉树的构造类问题。

二叉树的构造问题一般都是使用「分解问题」的思路：构造整棵树 = 根节点 + 构造左子树 + 构造右子树。

接下来直接看题。

### 构造最大二叉树

先来道简单的，这是力扣第 654 题「最大二叉树」，题目如下：

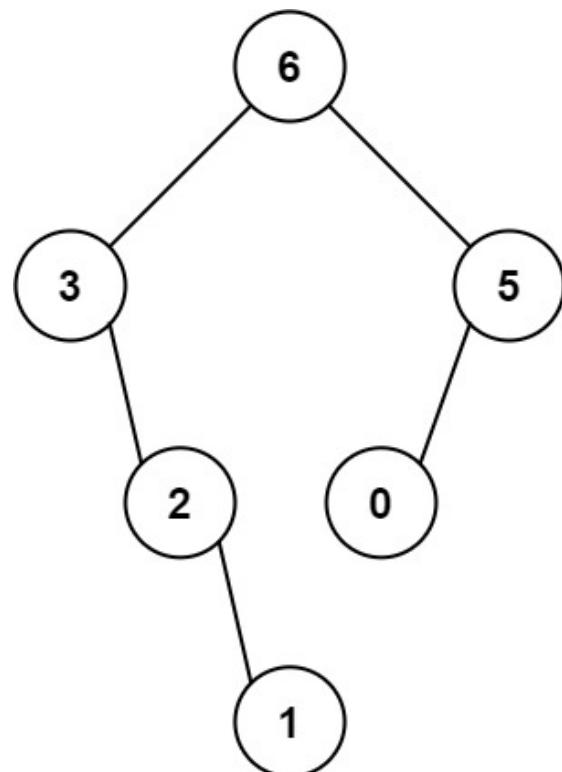
▼ 654. 最大二叉树 Leetcode | 力扣

给定一个不重复的整数数组 `nums`。**最大二叉树** 可以用下面的算法从 `nums` 递归地构建:

1. 创建一个根节点，其值为 `nums` 中的最大值。
2. 递归地在最大值 **左边** 的子数组前缀上 构建左子树。
3. 递归地在最大值 **右边** 的子数组后缀上 构建右子树。

返回 `nums` 构建的 **最大二叉树**。

示例 1:



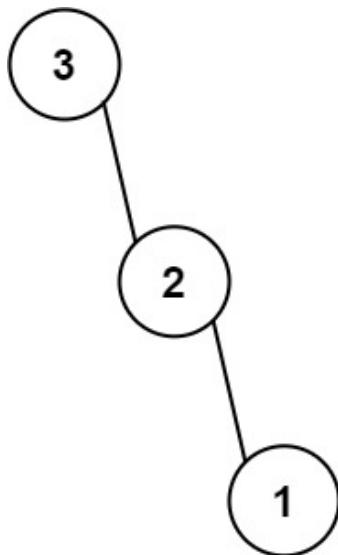
输入: `nums = [3,2,1,6,0,5]`

输出: `[6,3,5,null,2,0,null,null,1]`

解释: 递归调用如下所示:

- `[3,2,1,6,0,5]` 中的最大值是 6，左边部分是 `[3,2,1]`，右边部分是 `[0,5]`。
  - `[3,2,1]` 中的最大值是 3，左边部分是 `[]`，右边部分是 `[2,1]`。
    - 空数组，无子节点。
    - `[2,1]` 中的最大值是 2，左边部分是 `[]`，右边部分是 `[1]`。
      - 空数组，无子节点。
      - 只有一个元素，所以子节点是一个值为 1 的节点。
  - `[0,5]` 中的最大值是 5，左边部分是 `[0]`，右边部分是 `[]`。
    - 只有一个元素，所以子节点是一个值为 0 的节点。
    - 空数组，无子节点。

示例 2:



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

提示:

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 1000`
- `nums` 中的所有整数 互不相同

```
// 函数签名如下
TreeNode constructMaximumBinaryTree(int[] nums);
```

每个二叉树节点都可以认为是一棵子树的根节点，对于根节点，首先要做的当然是想办法把自己先构造出来，然后想办法构造自己的左右子树。

所以，我们要遍历数组把找到最大值 `maxVal`，从而把根节点 `root` 做出来，然后对 `maxVal` 左边的数组和右边的数组进行递归构建，作为 `root` 的左右子树。

按照题目给出的例子，输入的数组为 `[3,2,1,6,0,5]`，对于整棵树的根节点来说，其实在做这件事：

```
TreeNode constructMaximumBinaryTree([3,2,1,6,0,5]) {
    // 找到数组中的最大值
    TreeNode root = new TreeNode(6);
    // 递归调用构造左右子树
    root.left = constructMaximumBinaryTree([3,2,1]);
    root.right = constructMaximumBinaryTree([0,5]);
    return root;
}

// 当前 nums 中的最大值就是根节点，然后根据索引递归调用左右数组构造左右子树即可
// 再详细一点，就是如下伪码
TreeNode constructMaximumBinaryTree(int[] nums) {
    if (nums is empty) return null;
    // 找到数组中的最大值
    int maxVal = Integer.MIN_VALUE;
    int index = 0;
```

```

    for (int i = 0; i < nums.length; i++) {
        if (nums[i] > maxVal) {
            maxVal = nums[i];
            index = i;
        }
    }

    TreeNode root = new TreeNode(maxVal);
    // 递归调用构造左右子树
    root.left = constructMaximumBinaryTree(nums[0..index-1]);
    root.right = constructMaximumBinaryTree(nums[index+1..nums.length-1]);
    return root;
}

```

当前 `nums` 中的最大值就是根节点，然后根据索引递归调用左右数组构造左右子树即可。

明确了思路，我们可以重新写一个辅助函数 `build`，来控制 `nums` 的索引：

```

class Solution {

    public TreeNode constructMaximumBinaryTree(int[] nums) {
        return build(nums, 0, nums.length - 1);
    }

    // 定义：将 nums[lo..hi] 构造成符合条件的树，返回根节点
    TreeNode build(int[] nums, int lo, int hi) {
        // base case
        if (lo > hi) {
            return null;
        }

        // 找到数组中的最大值和对应的索引
        int index = -1, maxVal = Integer.MIN_VALUE;
        for (int i = lo; i <= hi; i++) {
            if (maxVal < nums[i]) {
                index = i;
                maxVal = nums[i];
            }
        }

        // 先构造出根节点
        TreeNode root = new TreeNode(maxVal);
        // 递归调用构造左右子树
        root.left = build(nums, lo, index - 1);
        root.right = build(nums, index + 1, hi);

        return root;
    }
}

```



至此，这道题就做完了，还是挺简单的对吧，下面看两道更困难一些的。

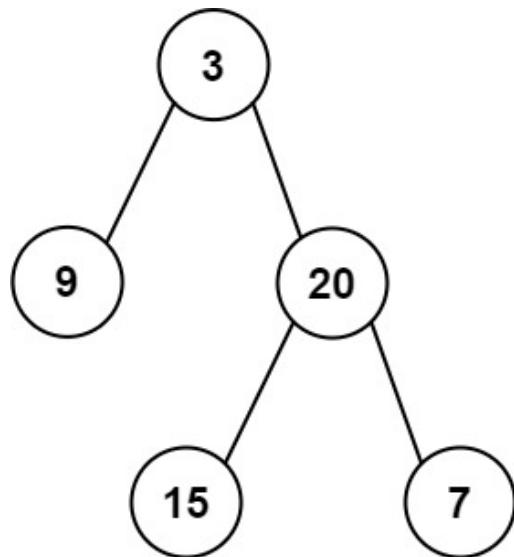
## 通过前序和中序遍历结果构造二叉树

力扣第 105 题「从前序和中序遍历序列构造二叉树」就是这道经典题目，面试笔试中常考：

### ▼ 105. 从前序与中序遍历序列构造二叉树 [Leetcode | 力扣](#)

给定两个整数数组 `preorder` 和 `inorder`，其中 `preorder` 是二叉树的先序遍历，`inorder` 是同一棵树的中序遍历，请构造二叉树并返回其根节点。

示例 1：



输入： `preorder = [3,9,20,15,7]`, `inorder = [9,3,15,20,7]`  
输出： `[3,9,20,null,null,15,7]`

示例 2：

输入： `preorder = [-1]`, `inorder = [-1]`  
输出： `[-1]`

提示：

- `1 <= preorder.length <= 3000`
- `inorder.length == preorder.length`
- `-3000 <= preorder[i], inorder[i] <= 3000`
- `preorder` 和 `inorder` 均无重复元素
- `inorder` 均出现在 `preorder`
- `preorder` 保证为二叉树的前序遍历序列
- `inorder` 保证为二叉树的中序遍历序列

```
// 函数签名如下  
TreeNode buildTree(int[] preorder, int[] inorder);
```

废话不多说，直接来想思路，首先思考，根节点应该做什么。

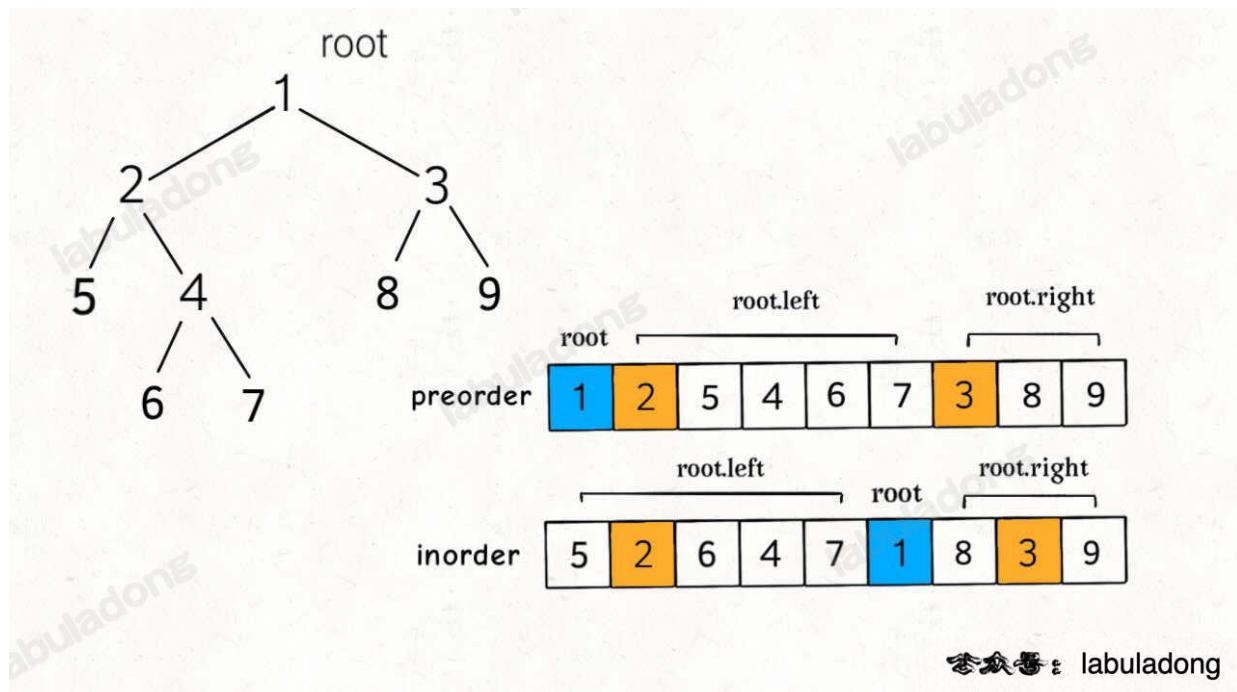
类似上一题，我们肯定要想办法确定根节点的值，把根节点做出来，然后递归构造左右子树即可。

我们先来回顾一下，前序遍历和中序遍历的结果有什么特点？

```
void traverse(TreeNode root) {
    // 前序遍历
    preorder.add(root.val);
    traverse(root.left);
    traverse(root.right);
}

void traverse(TreeNode root) {
    traverse(root.left);
    // 中序遍历
    inorder.add(root.val);
    traverse(root.right);
}
```

前文 [二叉树就那几个框架](#) 写过，这样的遍历顺序差异，导致了 `preorder` 和 `inorder` 数组中的元素分布有如下特点：



找到根节点是很简单的，前序遍历的第一个值 `preorder[0]` 就是根节点的值。

关键在于如何通过根节点的值，将 `preorder` 和 `inorder` 数组划分成两半，构造根节点的左右子树？

换句话说，对于以下代码中的 ? 部分应该填入什么：

```
TreeNode buildTree(int[] preorder, int[] inorder) {
    // 根据函数定义，用 preorder 和 inorder 构造二叉树
    return build(preorder, 0, preorder.length - 1,
                inorder, 0, inorder.length - 1);
}

// build 函数的定义：
// 若前序遍历数组为 preorder[preStart..preEnd]，
// 中序遍历数组为 inorder[inStart..inEnd]，
// 构造二叉树，返回该二叉树的根节点
TreeNode build(int[] preorder, int preStart, int preEnd,
```

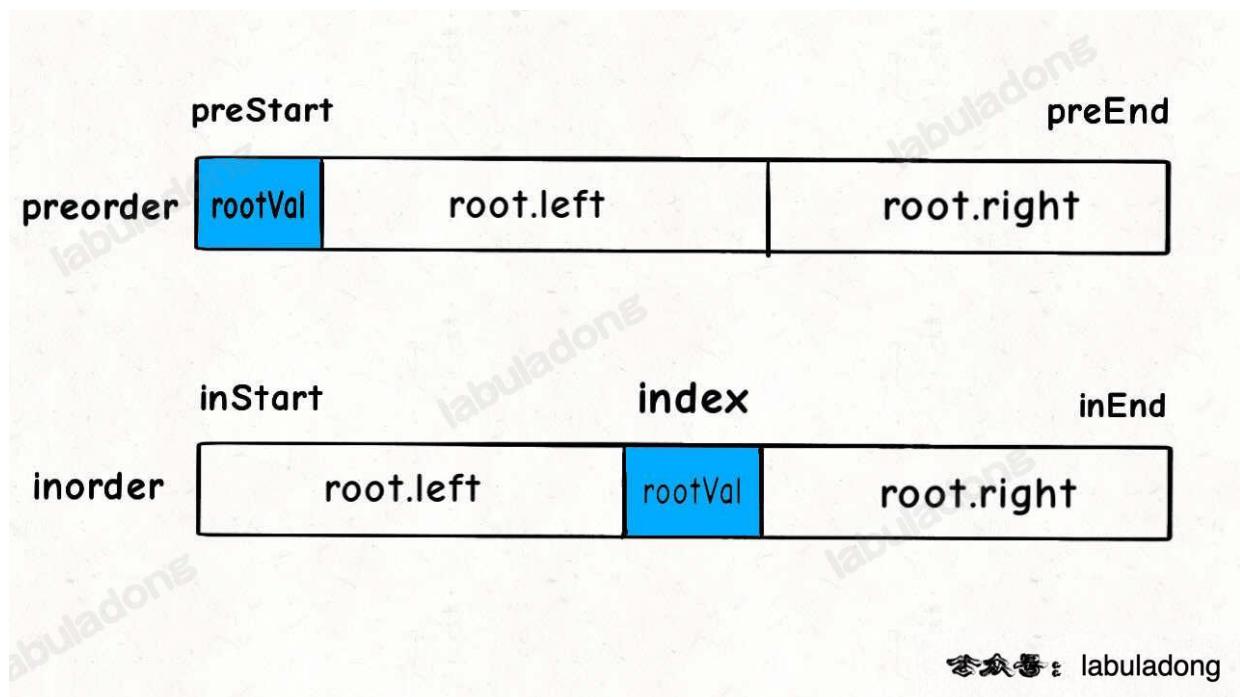
```

        int[] inorder, int inStart, int inEnd) {
    // root 节点对应的值就是前序遍历数组的第一个元素
    int rootVal = preorder[preStart];
    // rootVal 在中序遍历数组中的索引
    int index = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }

    TreeNode root = new TreeNode(rootVal);
    // 递归构造左右子树
    root.left = build(preorder, ?, ?,
                      inorder, ?, ?);
    root.right = build(preorder, ?, ?,
                       inorder, ?, ?);
    return root;
}

```

对于代码中的 `rootVal` 和 `index` 变量，就是下图这种情况：



另外，也有读者注意到，通过 `for` 循环遍历的方式去确定 `index` 效率不算高，可以进一步优化。

因为题目说二叉树节点的值不存在重复，所以可以使用一个 `HashMap` 存储元素到索引的映射，这样就可以直接通过 `HashMap` 查到 `rootVal` 对应的 `index`：

```

// 存储 inorder 中值到索引的映射
HashMap<Integer, Integer> valToIndex = new HashMap<>();

public TreeNode buildTree(int[] preorder, int[] inorder) {
    for (int i = 0; i < inorder.length; i++) {
        valToIndex.put(inorder[i], i);
    }
}

```

```
    return build(preorder, 0, preorder.length - 1,
                 inorder, 0, inorder.length - 1);
}

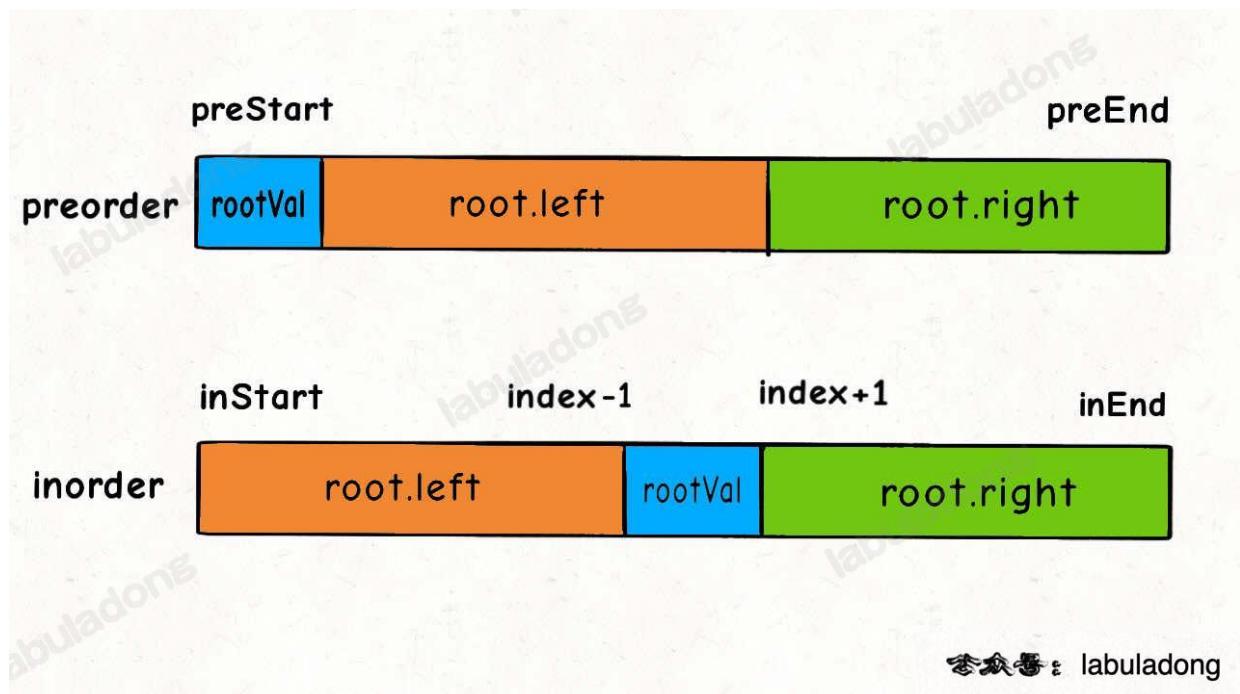
TreeNode build(int[] preorder, int preStart, int preEnd,
               int[] inorder, int inStart, int inEnd) {
    int rootVal = preorder[preStart];
    // 避免 for 循环寻找 rootVal
    int index = valToIndex.get(rootVal);
    // ...
}
```

现在我们来看图做填空题，下面这几个问号处应该填什么：

```
root.left = build(preorder, ?, ?,
                  inorder, ?, ?);

root.right = build(preorder, ?, ?,
                   inorder, ?, ?);
```

对于左右子树对应的 `inorder` 数组的起始索引和终止索引比较容易确定：

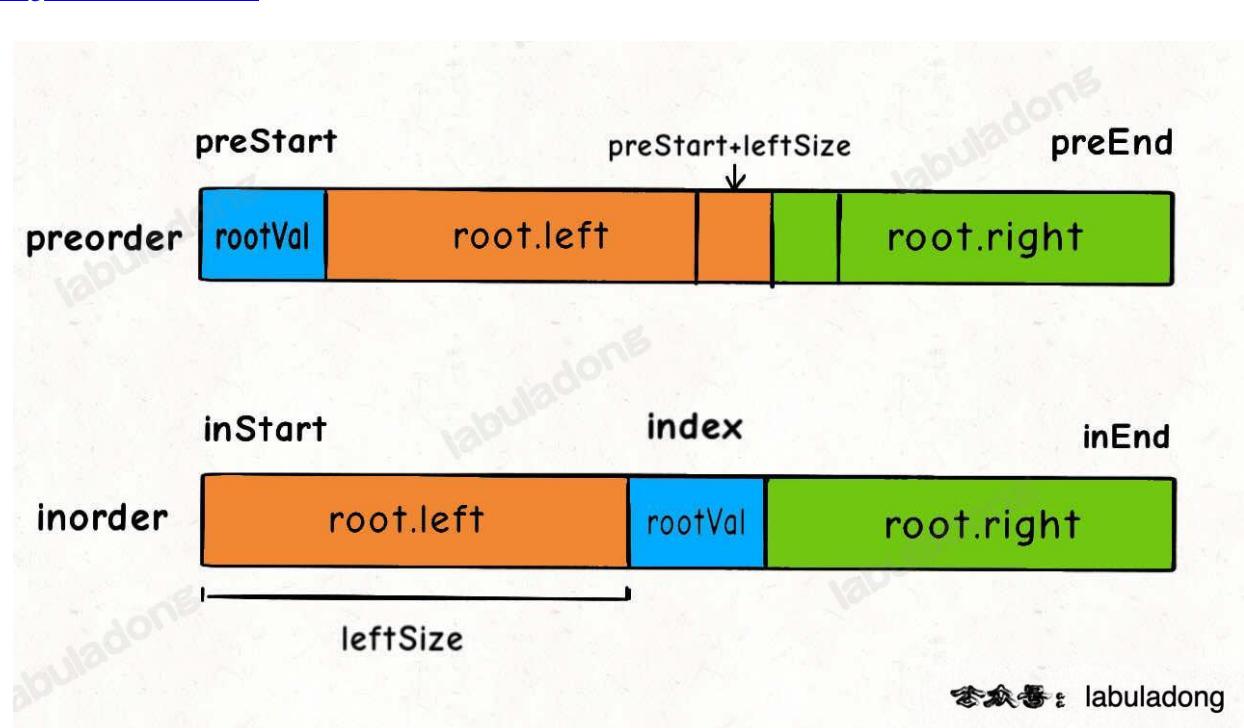


```
root.left = build(preorder, ?, ?,
                  inorder, inStart, index - 1);

root.right = build(preorder, ?, ?,
                   inorder, index + 1, inEnd);
```

对于 `preorder` 数组呢？如何确定左右数组对应的起始索引和终止索引？

这个可以通过左子树的节点数推导出来，假设左子树的节点数为 `leftSize`，那么 `preorder` 数组上的索引情况是这样的：



看着这个图就可以把 **preorder** 对应的索引写进去了：

```
int leftSize = index - inStart;

root.left = build(preorder, preStart + 1, preStart + leftSize,
                  inorder, inStart, index - 1);

root.right = build(preorder, preStart + leftSize + 1, preEnd,
                   inorder, index + 1, inEnd);
```

至此，整个算法思路就完成了，我们再补一补 base case 即可写出解法代码：

```
class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        for (int i = 0; i < inorder.length; i++) {
            valToIndex.put(inorder[i], i);
        }
        return build(preorder, 0, preorder.length - 1,
                    inorder, 0, inorder.length - 1);
    }

    // build 函数的定义：
    // 若前序遍历数组为 preorder[preStart..preEnd]，
    // 中序遍历数组为 inorder[inStart..inEnd]，
    // 构造二叉树，返回该二叉树的根节点
    TreeNode build(int[] preorder, int preStart, int preEnd,
                  int[] inorder, int inStart, int inEnd) {

        if (preStart > preEnd) {
            return null;
        }
```

```
// root 节点对应的值就是前序遍历数组的第一个元素
int rootVal = preorder[preStart];
// rootVal 在中序遍历数组中的索引
int index = valToIndex.get(rootVal);

int leftSize = index - inStart;

// 先构造出当前根节点
TreeNode root = new TreeNode(rootVal);
// 递归构造左右子树
root.left = build(preorder, preStart + 1, preStart + leftSize,
                   inorder, inStart, index - 1);

root.right = build(preorder, preStart + leftSize + 1, preEnd,
                   inorder, index + 1, inEnd);
return root;
}
}
```

我们的主函数只要调用 `build` 函数即可，你看着函数这么多参数，解法这么多代码，似乎比我们上面讲的那道题难很多，让人望而生畏，实际上呢，这些参数无非就是控制数组起止位置的，画个图就能解决了。

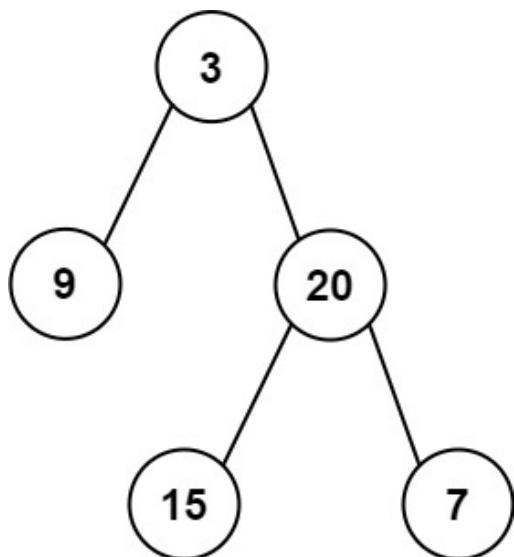
## 通过后序和中序遍历结果构造二叉树

类似上一题，这次我们利用后序和中序遍历的结果数组来还原二叉树，这是力扣第 106 题「从后序和中序遍历序列构造二叉树」：

### ▼ 106. 从中序与后序遍历序列构造二叉树 [Leetcode | 力扣](#)

给定两个整数数组 `inorder` 和 `postorder`，其中 `inorder` 是二叉树的中序遍历，`postorder` 是同一棵树的后序遍历，请你构造并返回这颗 二叉树。

示例 1：



输入： `inorder = [9, 3, 15, 20, 7]`, `postorder = [9, 15, 7, 20, 3]`  
输出： `[3, 9, 20, null, null, 15, 7]`

## 示例 2:

```
输入: inorder = [-1], postorder = [-1]
输出: [-1]
```

### 提示:

- $1 \leq \text{inorder.length} \leq 3000$
- $\text{postorder.length} == \text{inorder.length}$
- $-3000 \leq \text{inorder}[i], \text{postorder}[i] \leq 3000$
- $\text{inorder}$  和  $\text{postorder}$  都由不同的值组成
- $\text{postorder}$  中每一个值都在  $\text{inorder}$  中
- $\text{inorder}$  保证是树的中序遍历
- $\text{postorder}$  保证是树的后序遍历

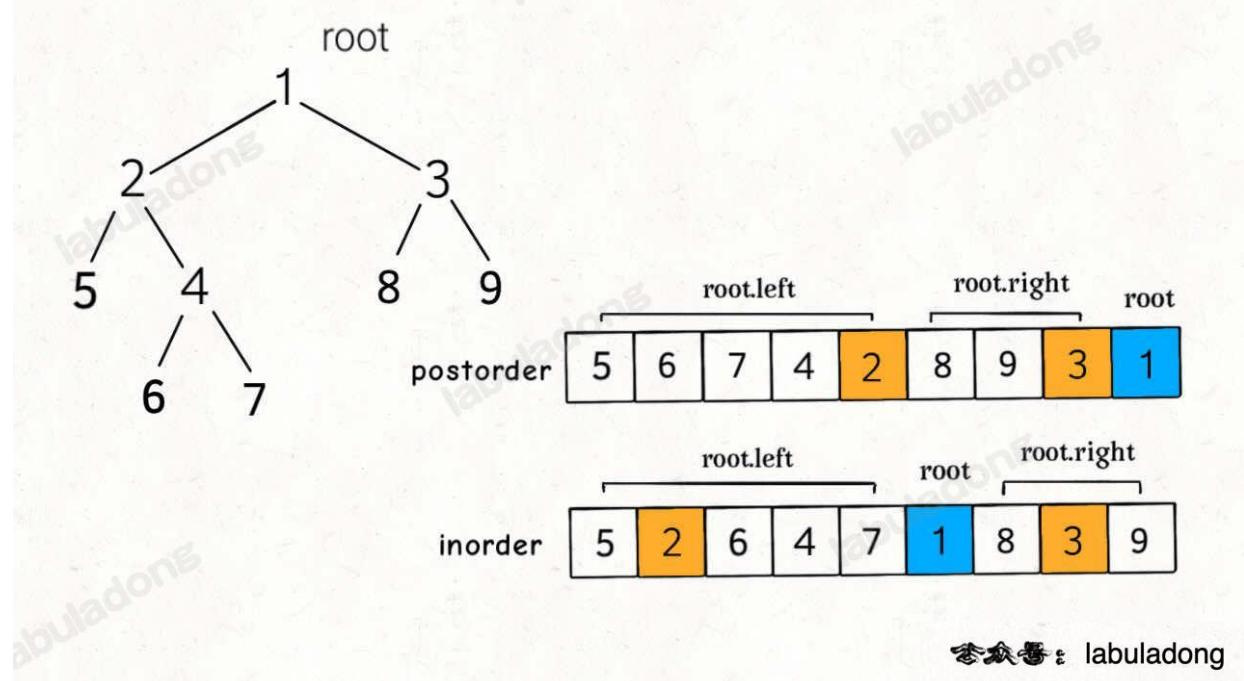
```
// 函数签名如下
TreeNode buildTree(int[] inorder, int[] postorder);
```

类似的，看下后序和中序遍历的特点：

```
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
    // 后序遍历
    postorder.add(root.val);
}

void traverse(TreeNode root) {
    traverse(root.left);
    // 中序遍历
    inorder.add(root.val);
    traverse(root.right);
}
```

这样的遍历顺序差异，导致了  $\text{postorder}$  和  $\text{inorder}$  数组中的元素分布有如下特点：



这道题和上一题的关键区别是，后序遍历和前序遍历相反，根节点对应的值为 `postorder` 的最后一个元素。

整体的算法框架和上一题非常类似，我们依然写一个辅助函数 `build`：

```

class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

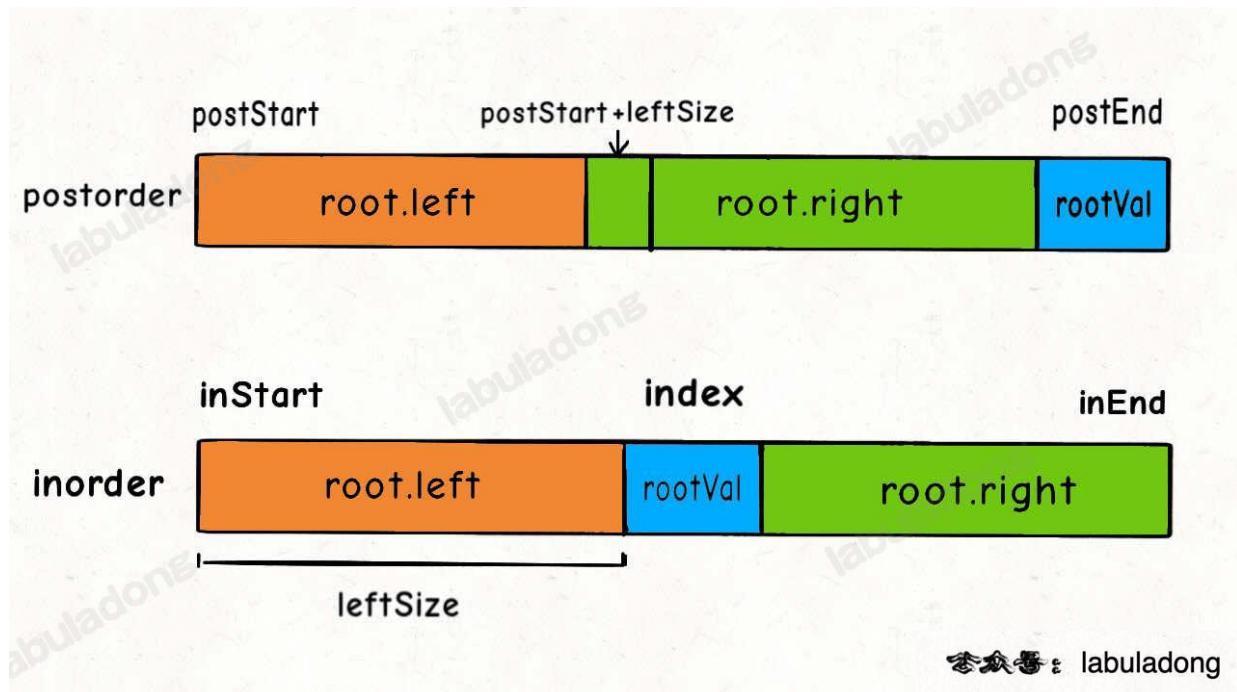
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        for (int i = 0; i < inorder.length; i++) {
            valToIndex.put(inorder[i], i);
        }
        return build(inorder, 0, inorder.length - 1,
                    postorder, 0, postorder.length - 1);
    }

    // build 函数的定义：
    // 后序遍历数组为 postorder[postStart..postEnd]，
    // 中序遍历数组为 inorder[inStart..inEnd]，
    // 构造二叉树，返回该二叉树的根节点
    TreeNode build(int[] inorder, int inStart, int inEnd,
                  int[] postorder, int postStart, int postEnd) {
        // root 节点对应的值就是后序遍历数组的最后一个元素
        int rootVal = postorder[postEnd];
        // rootVal 在中序遍历数组中的索引
        int index = valToIndex.get(rootVal);

        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        root.left = build(inorder, ?, ?,
                          postorder, ?, ?);
        root.right = build(inorder, ?, ?,
                           postorder, ?, ?);
        return root;
    }
}

```

现在 `postorder` 和 `inorder` 对应的状态如下：



我们可以按照上图将问号处的索引正确填入：

```
int leftSize = index - inStart;

root.left = build(inorder, inStart, index - 1,
                  postorder, postStart, postStart + leftSize - 1);

root.right = build(inorder, index + 1, inEnd,
                  postorder, postStart + leftSize, postEnd - 1);
```

综上，可以写出完整的解法代码：

```
class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        for (int i = 0; i < inorder.length; i++) {
            valToIndex.put(inorder[i], i);
        }
        return build(inorder, 0, inorder.length - 1,
                    postorder, 0, postorder.length - 1);
    }

    // build 函数的定义：
    // 后序遍历数组为 postorder[postStart..postEnd],
    // 中序遍历数组为 inorder[inStart..inEnd],
    // 构造二叉树，返回该二叉树的根节点
    TreeNode build(int[] inorder, int inStart, int inEnd,
                  int[] postorder, int postStart, int postEnd) {
```

```
if (inStart > inEnd) {
    return null;
}
// root 节点对应的值就是后序遍历数组的最后一个元素
int rootVal = postorder[postEnd];
// rootVal 在中序遍历数组中的索引
int index = valToIndex.get(rootVal);
// 左子树的节点个数
int leftSize = index - inStart;
TreeNode root = new TreeNode(rootVal);
// 递归构造左右子树
root.left = build(inorder, inStart, index - 1,
                   postorder, postStart, postStart + leftSize - 1);
root.right = build(inorder, index + 1, inEnd,
                    postorder, postStart + leftSize, postEnd - 1);
return root;
}
}
```

### ▶ 😊 代码可视化动画😊

有了前一题的铺垫，这道题很快就解决了，无非就是 `rootVal` 变成了最后一个元素，再改改递归函数的参数而已，只要明白二叉树的特性，也不难写出来。

## 通过后序和前序遍历结果构造二叉树

这是力扣第 889 题「根据前序和后序遍历构造二叉树」，给你输入二叉树的前序和后序遍历结果，让你还原二叉树的结构。

函数签名如下：

```
TreeNode constructFromPrePost(int[] preorder, int[] postorder);
```

这道题和前两道题有一个本质的区别：

**通过前序中序，或者后序中序遍历结果可以确定唯一一棵原始二叉树，但是通过前序后序遍历结果无法确定唯一的原始二叉树。**

题目也说了，如果有多种可能的还原结果，你可以返回任意一种。

为什么呢？我们说过，构建二叉树的套路很简单，先找到根节点，然后找到并递归构造左右子树即可。

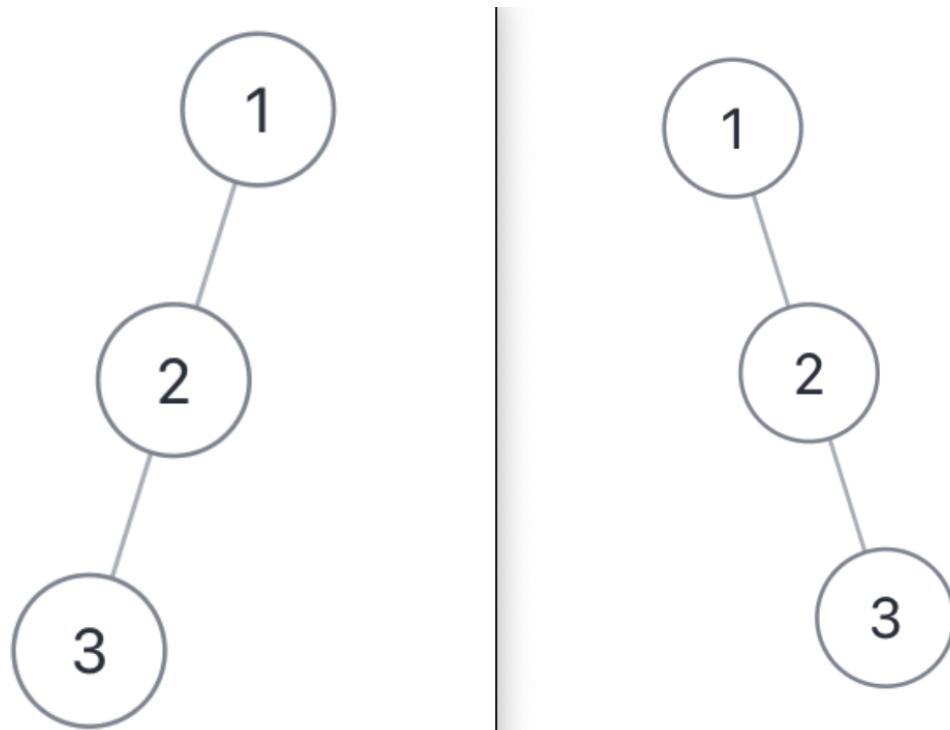
前两道题，可以通过前序或者后序遍历结果找到根节点，然后根据中序遍历结果确定左右子树（题目说了树中没有 `val` 相同的节点）。

这道题，你可以确定根节点，但是无法确切的知道左右子树有哪些节点。

举个例子，比如给你这个输入：

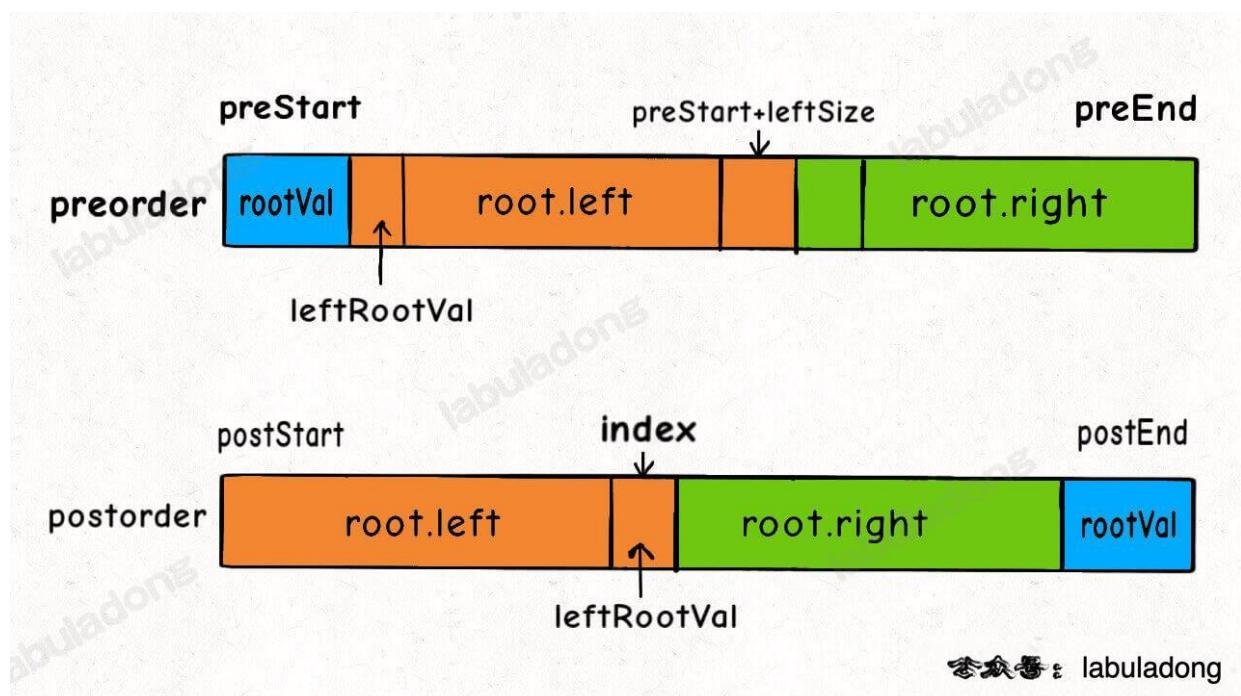
```
preorder = [1,2,3], postorder = [3,2,1]
```

下面这两棵树都是符合条件的，但显然它们的结构不同：



不过话说回来，用后序遍历和前序遍历结果还原二叉树，解法逻辑上和前两道题差别不大，也是通过控制左右子树的索引来构建：

- 1、首先把前序遍历结果的第一个元素或者后序遍历结果的最后一个元素确定为根节点的值。
- 2、然后把前序遍历结果的第二个元素作为左子树的根节点的值。
- 3、在后序遍历结果中寻找左子树根节点的值，从而确定了左子树的索引边界，进而确定右子树的索引边界，递归构造左右子树即可。



详情见代码。

```

class Solution {
    // 存储 postorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {
        for (int i = 0; i < postorder.length; i++) {
            valToIndex.put(postorder[i], i);
        }
        return build(preorder, 0, preorder.length - 1,
                    postorder, 0, postorder.length - 1);
    }

    // 定义：根据 preorder[preStart..preEnd] 和 postorder[postStart..postEnd]
    // 构建二叉树，并返回根节点。
    TreeNode build(int[] preorder, int preStart, int preEnd,
                  int[] postorder, int postStart, int postEnd) {
        if (preStart > preEnd) {
            return null;
        }
        if (preStart == preEnd) {
            return new TreeNode(preorder[preStart]);
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // root.left 的值是前序遍历第二个元素
        // 通过前序和后序遍历构造二叉树的关键在于通过左子树的根节点
        // 确定 preorder 和 postorder 中左右子树的元素区间
        int leftRootVal = preorder[preStart + 1];
        // leftRootVal 在后序遍历数组中的索引
        int index = valToIndex.get(leftRootVal);
        // 左子树的元素个数
        int leftSize = index - postStart + 1;

        // 先构造出当前根节点
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        // 根据左子树的根节点索引和元素个数推导左右子树的索引边界
        root.left = build(preorder, preStart + 1, preStart + leftSize,
                          postorder, postStart, index);
        root.right = build(preorder, preStart + leftSize + 1, preEnd,
                           postorder, index + 1, postEnd - 1);

        return root;
    }
}

```

▶  代码可视化动画

代码和前两道题非常类似，我们可以看着代码思考一下，为什么通过前序遍历和后序遍历结果还原的二叉树可能不唯一呢？

关键在这一句：

```
int leftRootVal = preorder[preStart + 1];
```

我们假设前序遍历的第二个元素是左子树的根节点，但实际上左子树有可能是空指针，那么这个元素就应该是右子树的根节点。由于这里无法确切进行判断，所以导致了最终答案的不唯一。

至此，通过前序和后序遍历结果还原二叉树的问题也解决了。

最后呼应下前文，二叉树的构造问题一般都是使用「分解问题」的思路：构造整棵树 = 根节点 + 构造左子树 + 构造右子树。先找出根节点，然后根据根节点的值找到左右子树的元素，进而递归构建出左右子树。

现在你是否明白其中的玄妙了呢？

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1008. Construct Binary Search Tree from Preorder Traversal</a>	<a href="#">1008. 前序遍历构造二叉搜索树</a>	
-	<a href="#">剑指 Offer 07. 重建二叉树</a>	
-	<a href="#">剑指 Offer 33. 二叉搜索树的后序遍历序列</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二叉树心法（后序篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">652. Find Duplicate Subtrees</a>	<a href="#">652. 寻找重复的子树</a>	困难

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历
- 二叉树心法（纲领篇）

本文是承接 [二叉树心法（纲领篇）](#) 的第四篇文章，主要讲二叉树后序位置的妙用，复述下前文关于后序遍历的描述：

前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码不仅可以获取参数数据，还可以获取到子树通过函数返回值传递回来的数据。

那么换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了。

多说无益，我们直接看题，这是力扣第 652 题「寻找重复的子树」：

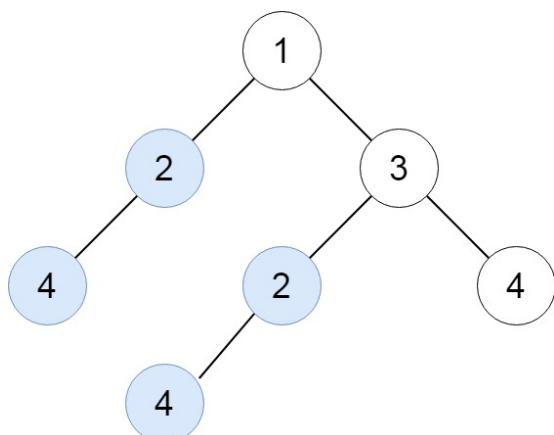
▼ [652. 寻找重复的子树](#) [Leetcode](#) | [力扣](#)

给你一棵二叉树的根节点 `root`，返回所有 **重复的子树**。

对于同一类的重复子树，你只需要返回其中任意 **一棵** 的根结点即可。

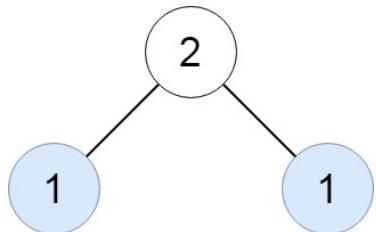
如果两棵树具有 **相同的结构** 和 **相同的结点值**，则认为二者是 **重复** 的。

**示例 1：**



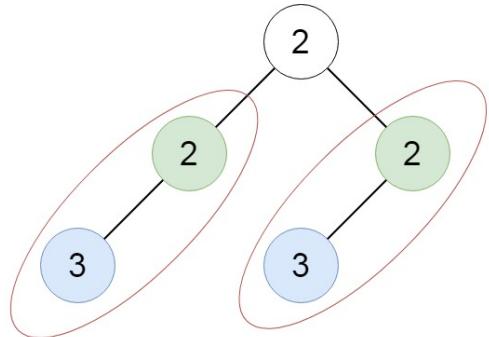
```
输入: root = [1,2,3,4,null,2,4,null,null,4]
输出: [[2,4],[4]]
```

示例 2:



```
输入: root = [2,1,1]
输出: [[1]]
```

示例 3:



```
输入: root = [2,2,2,3,null,3,null]
输出: [[2,3],[3]]
```

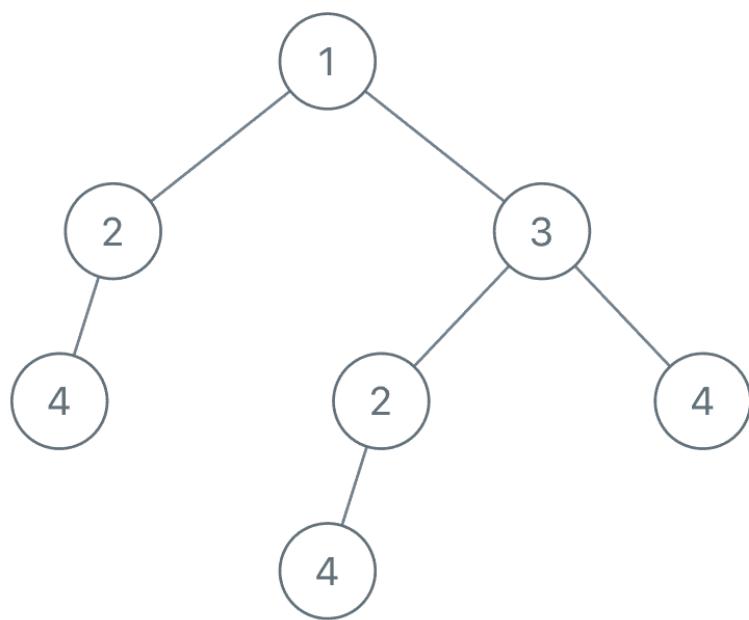
提示:

- 树中的结点数在 **[1, 5000]** 范围内。
- **-200 <= Node.val <= 200**

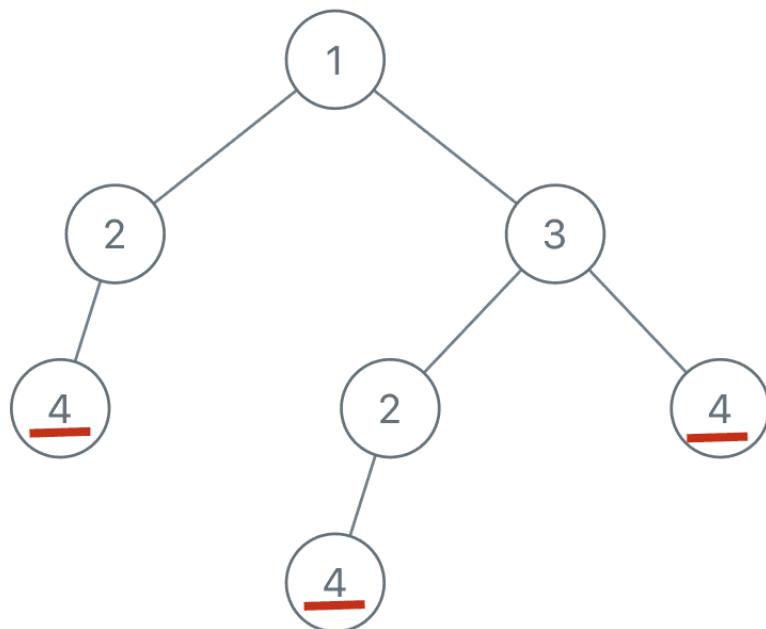
```
// 函数签名如下
List<TreeNode> findDuplicateSubtrees(TreeNode root);
```

我来简单解释下题目，输入是一棵二叉树的根节点 **root**，返回的是一个列表，里面装着若干个二叉树节点，这些节点对应的子树在原二叉树中是存在重复的。

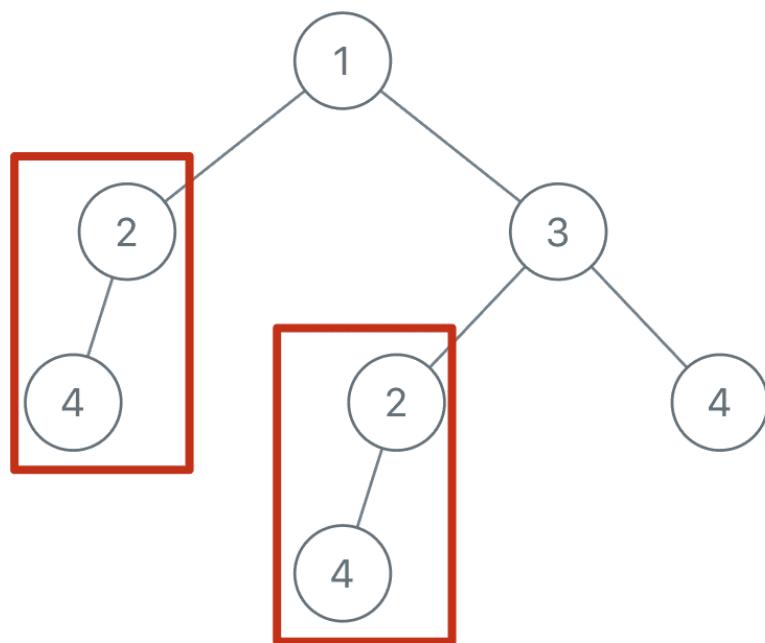
说起来比较绕，举例来说，比如输入如下的二叉树：



首先，节点 4 本身可以作为一棵子树，且二叉树中有多个节点 4：



类似的，还存在两棵以 2 为根的重复子树：



那么，我们返回的 `List` 中就应该有两个 `TreeNode`，值分别为 4 和 2（具体是哪个节点都无所谓）。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 二叉树心法（序列化篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">297. Serialize and Deserialize Binary Tree</a>	<a href="#">297. 二叉树的序列化与反序列化</a>	

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历
- 二叉树心法（纲领篇）

本文是承接 [二叉树心法（纲领篇）](#) 的第三篇文章，前文 [二叉树心法（构造篇）](#) 带你学习了二叉树构造技巧，本文加大难度，让你对二叉树同时进行「序列化」和「反序列化」。

要说序列化和反序列化，得先从 JSON 数据格式说起。

JSON 的运用非常广泛，比如我们经常将编程语言中的结构体序列化成 JSON 字符串，存入缓存或者通过网络发送给远端服务，消费者接受 JSON 字符串然后进行反序列化，就可以得到原始数据了。

这就是序列化和反序列化的目的，以某种特定格式组织数据，使得数据可以独立于编程语言。

那么假设现在有一棵用 Java 实现的二叉树，我想把它通过某些方式存储下来，然后用 C++ 读取这棵并还原这棵二叉树的结构，怎么办？这就需要对二叉树进行序列化和反序列化了。

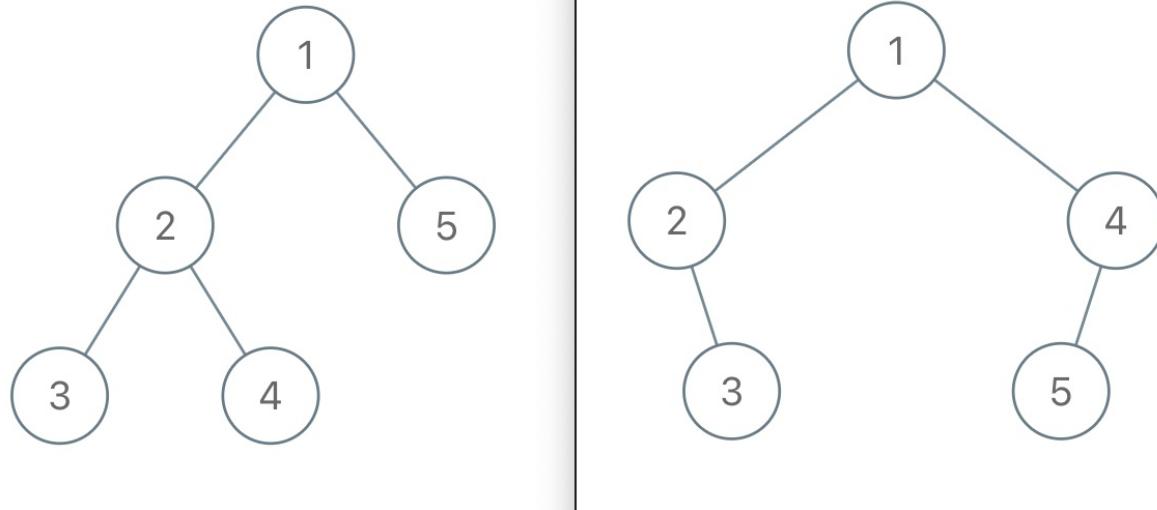
### 零、前/中/后序和二叉树的唯一性

谈具体的题目之前，我们先思考一个问题：**什么样的序列化的数据可以反序列化出唯一的一棵二叉树？**

比如说，如果给你一棵二叉树的前序遍历结果，你是否能够根据这个结果还原出这棵二叉树呢？

答案是也许可以，也许不可以，具体要看你给的前序遍历结果是否包含空指针的信息。如果包含了空指针，那么就可以唯一确定一棵二叉树，否则就不行。

举例来说，如果我给你这样一个不包含空指针的前序遍历结果 [1, 2, 3, 4, 5]，那么如下两棵二叉树都是满足这个前序遍历结果的：



所以给定不包含空指针信息的前序遍历结果，是不能还原出唯一的一棵二叉树的。

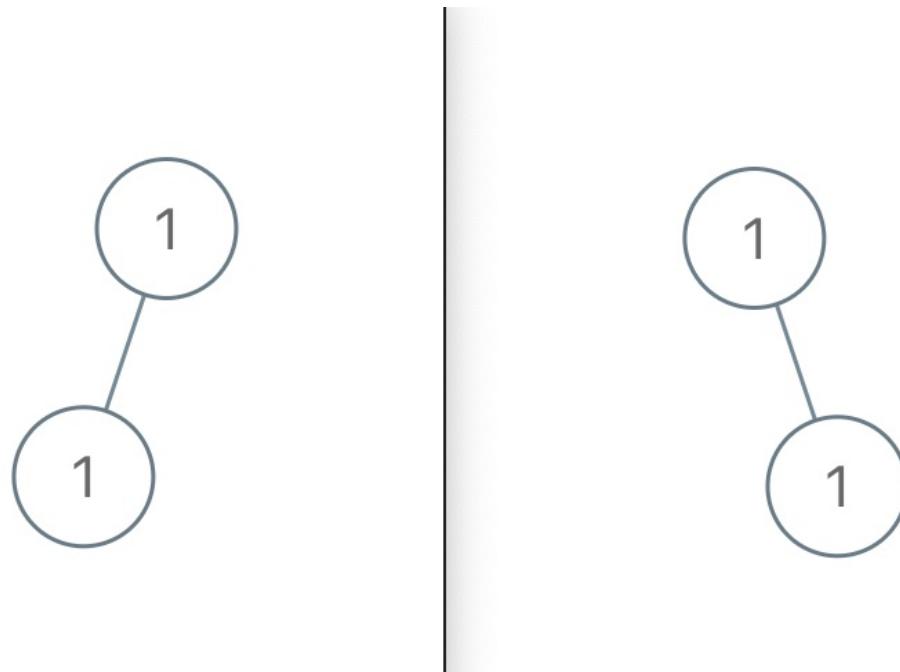
但如果我的前序遍历结果包含空指针的信息，那么就能还原出唯一的一棵二叉树了。比如说用 # 表示空指针，上图左侧的二叉树的前序遍历结果就是 [1, 2, 3, #, #, 4, #, #, 5, #, #]，上图右侧的二叉树的前序遍历结果就是 [1, 2, #, 3, #, #, 4, 5, #, #, #]，它俩就区分开了。

那么估计就有聪明的小伙伴说了：二叉树心法了。

首先要夸一下这种举一反三的思维，但很不幸，正确答案是，即便你包含了空指针的信息，也只有前序和后序的遍历结果才能唯一还原二叉树，中序遍历结果做不到。

本文后面会具体探讨这个问题，这里只简单说下原因：因为前序/后序遍历的结果中，可以确定根节点的位置，而中序遍历的结果中，根节点的位置是无法确定的。

更直观的，比如如下两棵二叉树显然拥有不同的结构，但它俩的中序遍历结果都是 [#, 1, #, 1, #]，无法区分：



说了这么多，总结下结论，当二叉树中节点的值不存在重复时：

1. 如果你的序列化结果中不包含空指针的信息，且你只给出一种遍历顺序，那么你无法还原出唯一的一棵二叉树。

2. 如果你的序列化结果中不包含空指针的信息，且你会给出两种遍历顺序，那么按照前文 [二叉树心法（构造篇）](#) 所说，分两种情况：

2.1. 如果你给出的是前序和中序，或者后序和中序，那么你可以还原出唯一的一棵二叉树。

2.2. 如果你给出前序和后序，那么你无法还原出唯一的一棵二叉树。

3. 如果你的序列化结果中包含空指针的信息，且你只给出一种遍历顺序，也要分两种情况：

3.1. 如果你给出的是前序或者后序，那么你可以还原出唯一的一棵二叉树。

3.2. 如果你给出的是中序，那么你无法还原出唯一的一棵二叉树。

我在开头提一下这些总结性的认识，可以理解性记忆，之后会遇到一些相关的题目，再回过头来看看这些总结，会有更深的理解，下面看具体的题目吧。

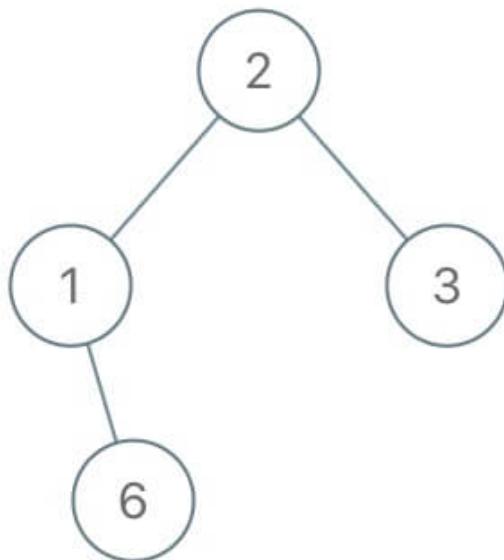
## 一、题目描述

力扣第 297 题「二叉树的序列化与反序列化」就是给你输入一棵二叉树的根节点 `root`，要求你实现如下一个类：

```
public class Codec {  
    // 把一棵二叉树序列化成字符串  
    public String serialize(TreeNode root) {}  
  
    // 把字符串反序列化成二叉树  
    public TreeNode deserialize(String data) {}  
}
```

我们可以用 `serialize` 方法将二叉树序列化成字符串，用 `deserialize` 方法将序列化的字符串反序列化成二叉树，至于以什么格式序列化和反序列化，这个完全由你决定。

比如说输入如下这样一颗二叉树：



`serialize` 方法也许会把它序列化成字符串 `2,1,#,6,#,#,3,#,#`, 其中 `#` 表示 `null` 指针, 那么把这个字符串再输入 `deserialize` 方法, 依然可以还原出这棵二叉树。

也就是说, 这两个方法会成对儿使用, 你只要保证他俩能够自洽就行了。

想象一下, 二叉树是一个二维平面内的结构, 而序列化出来的字符串是一个线性的一维结构。所谓的序列化不过就是把结构化的数据「打平」, 本质就是在考察二叉树的遍历方式。

二叉树的遍历方式有哪些? 递归遍历方式有前序遍历, 中序遍历, 后序遍历; 迭代方式一般是层级遍历。本文就把这些方式都尝试一遍, 来实现 `serialize` 方法和 `deserialize` 方法。

## 二、前序遍历解法

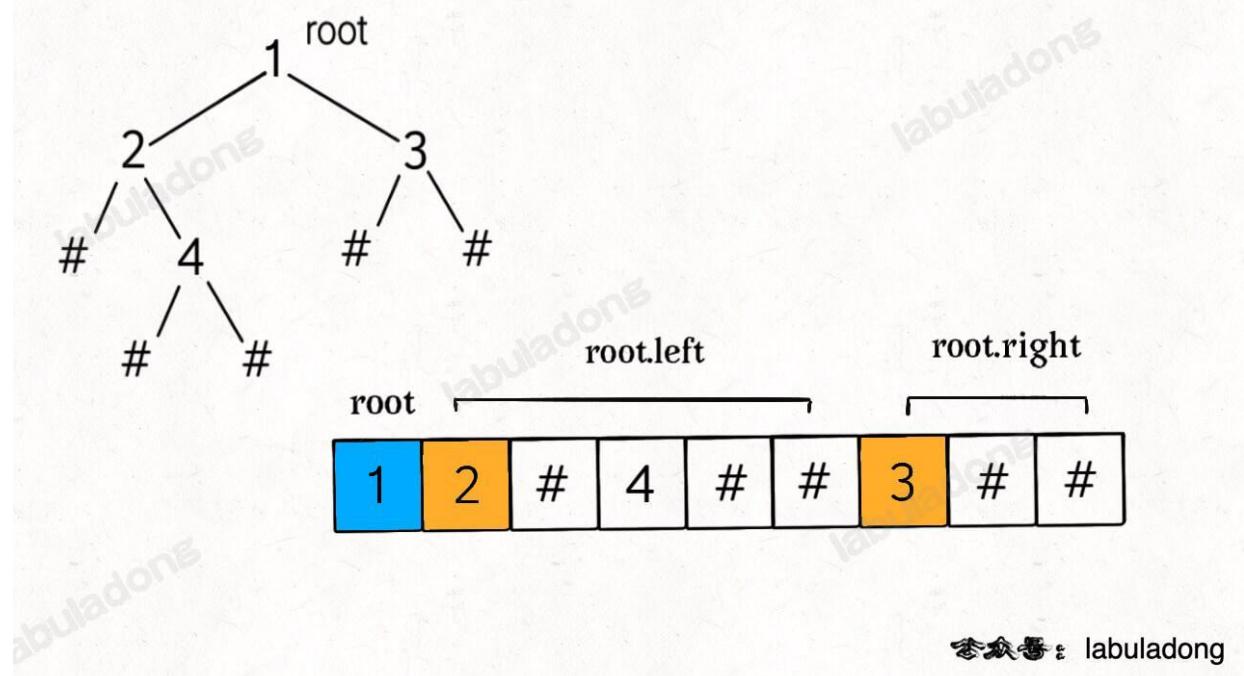
前文 [二叉树的遍历基础](#) 说过了二叉树的几种遍历方式, 前序遍历框架如下:

```
void traverse(TreeNode root) {  
    if (root == null) return;  
  
    // 前序位置的代码  
    traverse(root.left);  
    traverse(root.right);  
}
```

真的很简单, 在递归遍历两棵子树之前写的代码就是前序遍历代码, 那么请你看一看如下伪码:

```
LinkedList<Integer> res;  
void traverse(TreeNode root) {  
    if (root == null) {  
        // 暂且用数字 -1 代表空指针 null  
        res.addLast(-1);  
        return;  
    }  
  
    // ***** 前序位置 *****  
    res.addLast(root.val);  
    // *****  
  
    traverse(root.left);  
    traverse(root.right);  
}
```

调用 `traverse` 函数之后, 你是否可以立即想出这个 `res` 列表中元素的顺序是怎样的? 比如如下二叉树 (`#` 代表空指针 `null`) , 可以直观看出前序遍历做的事情:



© labuladong

那么 `res = [1,2,-1,4,-1,-1,3,-1,-1]`，这就是将二叉树「打平」到了一个列表中，其中 -1 代表 null。

那么，将二叉树打平到一个字符串中也是完全一样的：

```
// 代表分隔符的字符
String SEP = ",";

// 代表 null 空指针的字符
String NULL = "#";

// 用于拼接字符串
StringBuilder sb = new StringBuilder();

// 将二叉树打平为字符串
void traverse(TreeNode root, StringBuilder sb) {
    if (root == null) {
        sb.append(NULL).append(SEP);
        return;
    }

    // ***** 前序位置 *****
    sb.append(root.val).append(SEP);
    // *****

    traverse(root.left, sb);
    traverse(root.right, sb);
}
```

`StringBuilder` 可以用于高效拼接字符串，所以也可以认为是一个列表，用 , 作为分隔符，用 # 表示空指针 null，调用完 `traverse` 函数后，`sb` 中的字符串应该是 `1,2,#,4,#,#,3,#,#,`。

至此，我们已经可以写出序列化函数 `serialize` 的代码了：

```
class Codec {
    String SEP = ",";
```

```
String NULL = "#";

// 主函数，将二叉树序列化为字符串
public String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    _serialize(root, sb);
    return sb.toString();
}

// 辅助函数，将二叉树存入 StringBuilder
void _serialize(TreeNode root, StringBuilder sb) {
    if (root == null) {
        sb.append(NULL).append(SEP);
        return;
    }

    // ***** 前序位置 *****
    sb.append(root.val).append(SEP);
    // *****

    _serialize(root.left, sb);
    _serialize(root.right, sb);
}
}
```

现在，思考一下如何写 `deserialize` 函数，将字符串反过来构造二叉树。

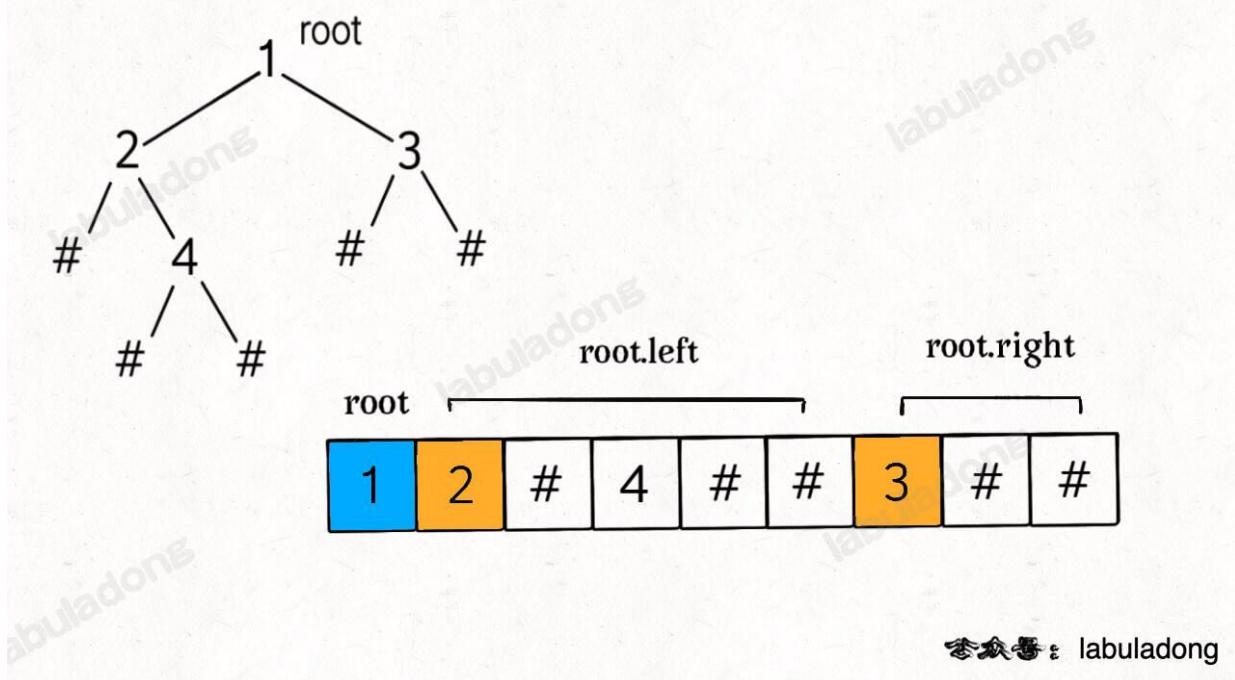
首先我们可以把字符串转化成列表：

```
String data = "1,2,#,4,#,#,3,#,#,";
String[] nodes = data.split(",");
```

这样，`nodes` 列表就是二叉树的前序遍历结果，问题转化为：如何通过二叉树的前序遍历结果还原一棵二叉树？

前文 [二叉树心法（构造篇）](#) 说过，至少要得到前、中、后序遍历中的两种互相配合才能还原二叉树。那是因为前文的遍历结果没有记录空指针的信息。这里的 `nodes` 列表包含了空指针的信息，所以只使用 `nodes` 列表就可以还原二叉树。

根据我们刚才的分析，`nodes` 列表就是一棵打平的二叉树：



© labuladong

那么，反序列化过程也是一样，先确定根节点 **root**，然后遵循前序遍历的规则，递归生成左右子树即可：

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

## 二叉搜索树心法（特性篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">538. Convert BST to Greater Tree</a>	538. 把二叉搜索树转换为累加树	
<a href="#">230. Kth Smallest Element in a BST</a>	230. 二叉搜索树中第K小的元素	
<a href="#">1038. Binary Search Tree to Greater Sum Tree</a>	1038. 从二叉搜索树到更大和树	

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历

前文手把手带你刷二叉树已经写了 [思维篇](#)，[构造篇](#)，[后序篇](#) 和 [序列化篇](#)。

今天开启二叉搜索树（Binary Search Tree，后文简写 BST）的系列文章，手把手带你刷 BST。

首先，BST 的特性大家应该都很熟悉了（详见基础知识章节的 [二叉树基础](#)）：

- 1、对于 BST 的每一个节点 `node`，左子树节点的值都比 `node` 的值要小，右子树节点的值都比 `node` 的值大。
- 2、对于 BST 的每一个节点 `node`，它的左侧子树和右侧子树都是 BST。

二叉搜索树并不算复杂，但我觉得它可以算是数据结构领域的半壁江山，直接基于 BST 的数据结构有 AVL 树，红黑树等等，拥有了自平衡性质，可以提供  $\log N$  级别的增删查改效率；还有 B+ 树，线段树等结构都是基于 BST 的思想来设计的。

从做算法题的角度来看 BST，除了它的定义，还有一个重要的性质：BST 的中序遍历结果是有序的（升序）。

也就是说，如果输入一棵 BST，以下代码可以将 BST 中每个节点的值升序打印出来：

```
void traverse(TreeNode root) {  
    if (root == null) return;  
    traverse(root.left);  
    // 中序遍历代码位置  
    print(root.val);  
    traverse(root.right);  
}
```

那么根据这个性质，我们来做两道算法题。

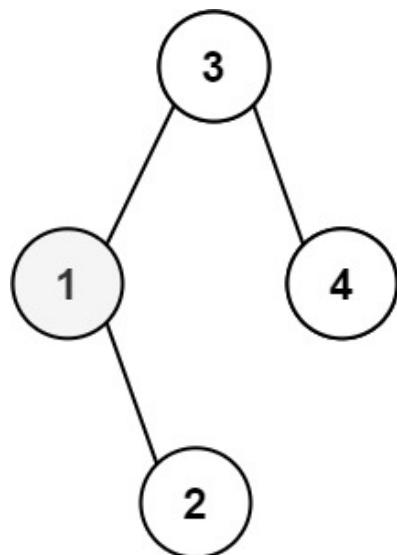
## 寻找第 K 小的元素

这是力扣第 230 题「二叉搜索树中第 K 小的元素」，看下题目：

### ▼ 230. 二叉搜索树中第K小的元素 [Leetcode](#) | 力扣

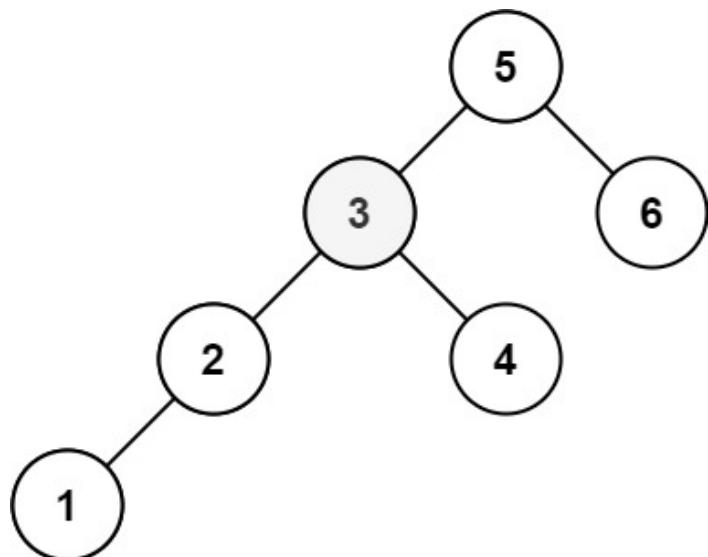
给定一个二叉搜索树的根节点 `root`，和一个整数 `k`，请你设计一个算法查找其中第 `k` 小的元素（从 1 开始计数）。

示例 1：



```
输入: root = [3,1,4,null,2], k = 1  
输出: 1
```

示例 2：



```
输入: root = [5,3,6,2,4,null,null,1], k = 3  
输出: 3
```

提示：

- 树中的节点数为 `n`。

- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

进阶：如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第  $k$  小的值，你将如何优化算法？

这个需求很常见吧，一个直接的思路就是升序排序，然后找第  $k$  个元素呗。BST 的中序遍历其实就是升序排序的结果，找第  $k$  个元素肯定不是什么难事。

按照这个思路，可以直接写出代码：

```
class Solution {  
    int kthSmallest(TreeNode root, int k) {  
        // 利用 BST 的中序遍历特性  
        traverse(root, k);  
        return res;  
    }  
  
    // 记录结果  
    int res = 0;  
    // 记录当前元素的排名  
    int rank = 0;  
    void traverse(TreeNode root, int k) {  
        if (root == null) {  
            return;  
        }  
        traverse(root.left, k);  
  
        // 中序代码位置  
        rank++;  
        if (k == rank) {  
            // 找到第 k 小的元素  
            res = root.val;  
            return;  
        }  
  
        traverse(root.right, k);  
    }  
}
```

### ▶ 🎃 代码可视化动画🎃

这道题就做完了，不过呢，还是要多说几句，因为这个解法并不是最高效的解法，而是仅仅适用于这道题。

我们前文 [高效计算数据流的中位数](#) 中就提过今天的这个问题：

如果让你实现一个在二叉搜索树中通过排名计算对应元素的方法 `select(int k)`，你会怎么设计？

如果按照我们刚才说的方法，利用「BST 中序遍历就是升序排序结果」这个性质，每次寻找第  $k$  小的元素都要中序遍历一次，最坏的时间复杂度是  $O(N)$ ， $N$  是 BST 的节点个数。

要知道 BST 性质是非常牛逼的，像红黑树这种改良的自平衡 BST，增删查改都是  $O(\log N)$  的复杂度，让你算一个第  $k$  小元素，时间复杂度竟然要  $O(N)$ ，有点低效了。

所以说，计算第  $k$  小元素，最好的算法肯定也是对数级别的复杂度，不过这个依赖于 BST 节点记录的信息有多少。

我们想一下 BST 的操作为什么这么高效？就拿搜索某一个元素来说，BST 能够在对数时间找到该元素的根本原因还是在 BST 的定义里，左子树小右子树大嘛，所以每个节点都可以通过对比自身的值判断去左子树还是右子树搜索目标值，从而避免了全树遍历，达到对数级复杂度。

那么回到这个问题，想找到第  $k$  小的元素，或者说找到排名为  $k$  的元素，如果想达到对数级复杂度，关键也在于每个节点得知道他自己排第几。

比如说你让我查找排名为  $k$  的元素，当前节点知道自己排名第  $m$ ，那么我可以比较  $m$  和  $k$  的大小：

- 1、如果  $m == k$ ，显然就是找到了第  $k$  个元素，返回当前节点就行了。
- 2、如果  $k < m$ ，那说明排名第  $k$  的元素在左子树，所以可以去左子树搜索第  $k$  个元素。
- 3、如果  $k > m$ ，那说明排名第  $k$  的元素在右子树，所以可以去右子树搜索第  $k - m - 1$  个元素。

这样就可以将时间复杂度降到  $O(\log N)$  了。

那么，如何让每一个节点知道自己的排名呢？

这就是我们之前说的，需要在二叉树节点中维护额外信息。每个节点需要记录，以自己为根的这棵二叉树有多少个节点。

也就是说，我们 `TreeNode` 中的字段应该如下：

```
class TreeNode {  
    int val;  
    // 以该节点为根的树的节点总数  
    int size;  
    TreeNode left;  
    TreeNode right;  
}
```

有了 `size` 字段，外加 BST 节点左小右大的性质，对于每个节点 `node` 就可以通过 `node.left` 推导出 `node` 的排名，从而做到我们刚才说到的对数级算法。

当然，`size` 字段需要在增删元素的时候需要被正确维护，力扣提供的 `TreeNode` 是没有 `size` 这个字段的，所以我们这道题就只能利用 BST 中序遍历的特性实现了，但是我们上面说到的优化思路是 BST 的常见操作，还是有必要理解的。

## BST 转化累加树

力扣第 538 题和 1038 题都是这道题，完全一样，你可以把它们一块做掉。看下题目：

### ▼ 538. 把二叉搜索树转换为累加树 [Leetcode | 力扣](#)

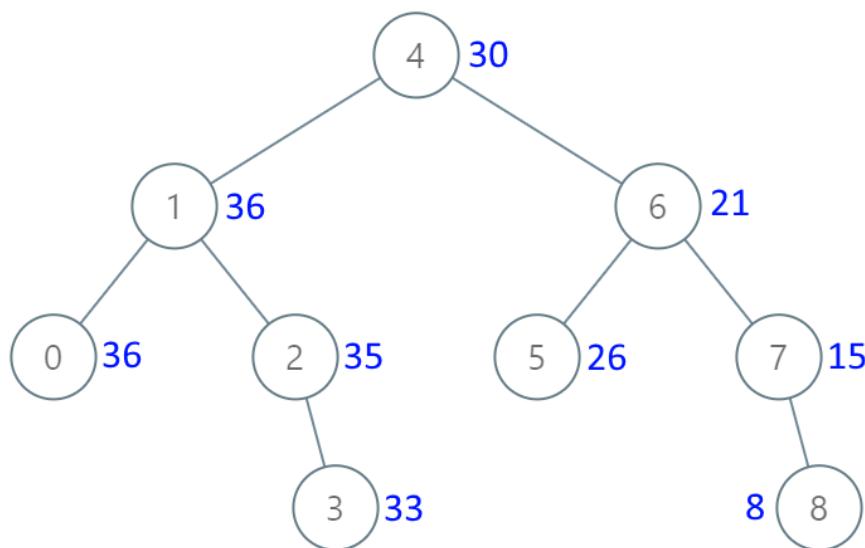
给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下，二叉搜索树满足下列约束条件：

- 节点的左子树仅包含键 小于 节点键的节点。
- 节点的右子树仅包含键 大于 节点键的节点。
- 左右子树也必须是二叉搜索树。

注意：本题和 1038: <https://leetcode-cn.com/problems/binary-search-tree-to-greater-sum-tree/> 相同

### 示例 1：



```
输入: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

#### 示例 2:

```
输入: root = [0,null,1]
输出: [1,null,1]
```

#### 示例 3:

```
输入: root = [1,0,2]
输出: [3,3,2]
```

#### 示例 4:

```
输入: root = [3,2,4,1]
输出: [7,9,4,10]
```

#### 提示:

- 树中的节点数介于 0 和  $10^4$  之间。
- 每个节点的值介于  $-10^4$  和  $10^4$  之间。
- 树中的所有值 互不相同。
- 给定的树为二叉搜索树。

题目应该不难理解，比如图中的节点 5，转化成累加树的话，比 5 大的节点有 6, 7, 8，加上 5 本身，所以累加树上这个节点的值应该是  $5+6+7+8=26$ 。

我们需要把 BST 转化成累加树，函数签名如下：

```
TreeNode convertBST(TreeNode root)
```

按照二叉树的通用思路，需要思考每个节点应该做什么，但是这道题上很难想到什么思路。

BST 的每个节点左小右大，这似乎是一个有用的信息，既然累加和是计算大于等于当前值的所有元素之和，那么每个节点都去计算右子树的和，不就行了吗？

这是不行的。对于一个节点来说，确实右子树都是比它大的元素，但问题是它的父节点也可能是比它大的元素呀？这个没法确定的，我们又没有触达父节点的指针，所以二叉树的通用思路在这里用不了。

此路不通，我们不妨换一个思路，还是利用 BST 的中序遍历特性。

刚才我们说了 BST 的中序遍历代码可以升序打印节点的值，那如果我想降序打印节点的值怎么办？

很简单，只要把递归顺序改一下，先遍历右子树，后遍历左子树就行了：

```
void traverse(TreeNode root) {  
    if (root == null) return;  
    // 先递归遍历右子树  
    traverse(root.right);  
    // 中序遍历代码位置  
    print(root.val);  
    // 后递归遍历左子树  
    traverse(root.left);  
}
```

这段代码可以降序打印 BST 节点的值，如果维护一个外部累加变量 `sum`，然后把 `sum` 赋值给 BST 中的每一个节点，不就将 BST 转化成累加树了吗？

看下代码就明白了：

```
class Solution {  
    TreeNode convertBST(TreeNode root) {  
        traverse(root);  
        return root;  
    }  
  
    // 记录累加和  
    int sum = 0;  
    void traverse(TreeNode root) {  
        if (root == null) {  
            return;  
        }  
        traverse(root.right);  
        // 维护累加和  
        sum += root.val;  
        // 将 BST 转化成累加树  
        root.val = sum;  
        traverse(root.left);  
    }  
}
```

▶ 🎨 代码可视化动画🎨

这道题就解决了，核心还是 BST 的中序遍历特性，只不过我们修改了递归顺序，降序遍历 BST 的元素值，从而契合题目累加树的要求。

简单总结下吧，BST 相关的问题，要么利用 BST 左小右大的特性提升算法效率，要么利用中序遍历的特性满足题目的要求，也就这么些事儿吧。

本文就到这里，更多经典的二叉树习题以及递归思维的训练，请参见二叉树章节中的 [习题部分](#)

▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer II 054. 所有大于等于节点的值之和</a>	困难

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二叉搜索树心法（基操篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">98. Validate Binary Search Tree</a>	98. 验证二叉搜索树	
<a href="#">700. Search in a Binary Search Tree</a>	700. 二叉搜索树中的搜索	
<a href="#">450. Delete Node in a BST</a>	450. 删除二叉搜索树中的节点	
<a href="#">701. Insert into a Binary Search Tree</a>	701. 二叉搜索树中的插入操作	

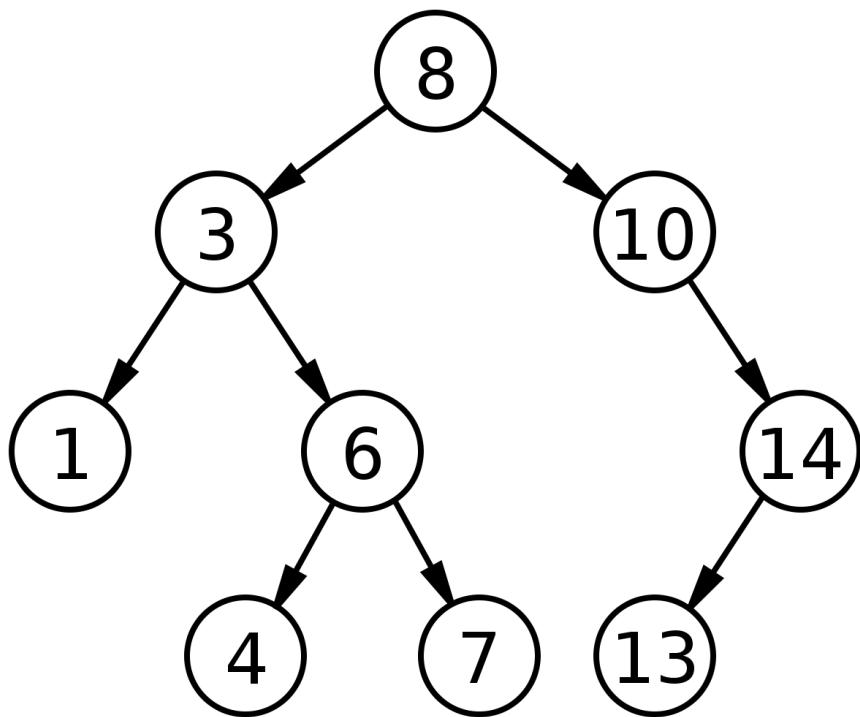
-----

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

我们前文 [二叉搜索树心法（特性篇）](#) 介绍了 BST 的基本特性，还利用二叉搜索树「中序遍历有序」的特性来解决了几道题目，本文来实现 BST 的基础操作：判断 BST 的合法性、增、删、查。其中「删」和「判断合法性」略微复杂。

BST 的基础操作主要依赖「左小右大」的特性，可以在二叉树中做类似二分搜索的操作，寻找一个元素的效率很高。比如下面这就是一棵合法的二叉树：



对于 BST 相关的问题，你可能会经常看到类似下面这样的代码逻辑：

```
void BST(TreeNode root, int target) {  
    if (root.val == target)  
        // 找到目标，做点什么  
    if (root.val < target)  
        BST(root.right, target);  
    if (root.val > target)  
        BST(root.left, target);  
}
```

这个代码框架其实和二叉树的遍历框架差不多，无非就是利用了 BST 左小右大的特性而已。接下来看下 BST 这种结构的基础操作是如何实现的。

## 一、判断 BST 的合法性

力扣第 98 题「验证二叉搜索树」就是让你判断输入的 BST 是否合法：

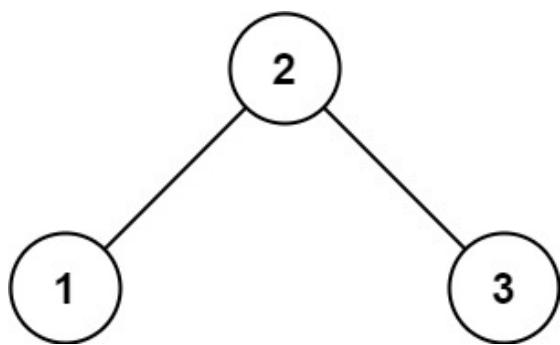
### ▼ 98. 验证二叉搜索树 [Leetcode](#) | 力扣

给你一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。

**有效** 二叉搜索树定义如下：

- 节点的左子树只包含 小于 当前节点的数。
- 节点的右子树只包含 大于 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

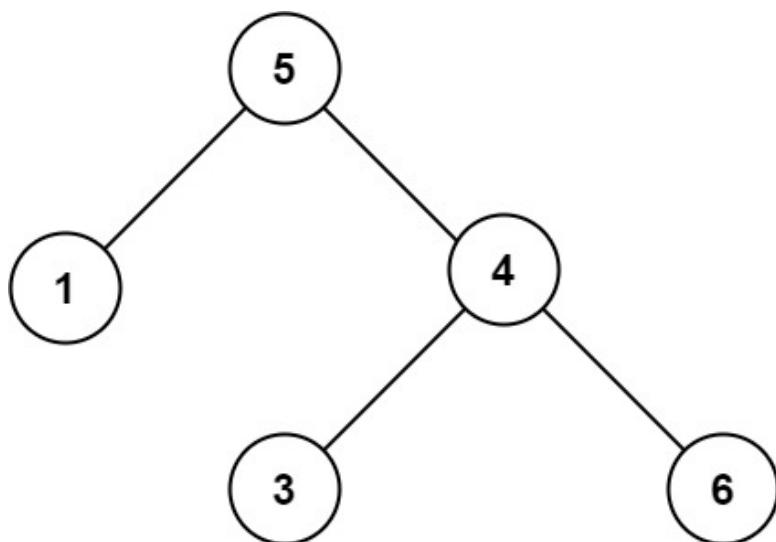
**示例 1：**



输入: root = [2,1,3]

输出: true

示例 2:



输入: root = [5,1,4,null,null,3,6]

输出: false

解释: 根节点的值是 5 , 但是右子节点的值是 4 。

提示:

- 树中节点数目范围在  $[1, 10^4]$  内
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

注意, 这里是有坑的哦。按照 BST 左小右大的特性, 每个节点想要判断自己是否是合法的 BST 节点, 要做的事不就是比较自己和左右孩子吗? 感觉应该这样写代码:

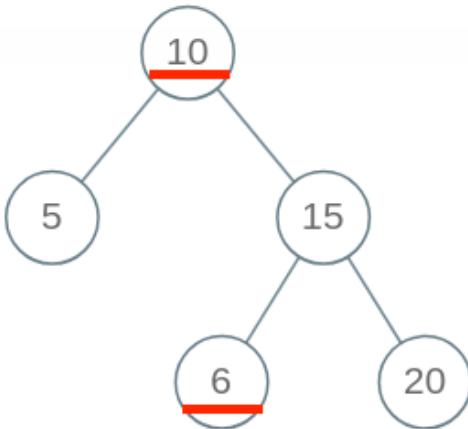
```
boolean isValidBST(TreeNode root) {  
    if (root == null) return true;  
    // root 的左边应该更小  
    if (root.left != null && root.left.val >= root.val)  
        return false;  
    // root 的右边应该更大  
    if (root.right != null && root.right.val <= root.val)  
        return false;  
}
```

```

        return isValidBST(root.left)
        && isValidBST(root.right);
    }
}

```

但是这个算法出现了错误，BST 的每个节点应该要小于右边子树的**所有节点**，下面这个二叉树显然不是 BST，因为节点 10 的右子树中有一个节点 6，但是我们的算法会把它判定为合法 BST：



错误的原因在于，对于每一个节点 `root`，代码值检查了它的左右孩子节点是否符合左小右大的原则；但是根据 BST 的定义，`root` 的整个左子树都要小于 `root.val`，整个右子树都要大于 `root.val`。

问题是，对于某一个节点 `root`，他只能管得了自己的左右子节点，怎么把 `root` 的约束传递给左右子树呢？请看正确的代码：

```

class Solution {
    public boolean isValidBST(TreeNode root) {
        return _isValidBST(root, null, null);
    }

    // 定义：该函数返回 root 为根的子树的所有节点是否满足 max.val > root.val > min.val
    public boolean _isValidBST(TreeNode root, TreeNode min, TreeNode max) {
        // base case
        if (root == null) return true;
        // 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
        if (min != null && root.val <= min.val) return false;
        if (max != null && root.val >= max.val) return false;
        // 根据定义，限定左子树的最大值是 root.val，右子树的最小值是 root.val
        return _isValidBST(root.left, min, root)
            && _isValidBST(root.right, root, max);
    }
}

```

## ▶ 🎥 代码可视化动画

我们通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉树算法的一个小技巧吧。

## 在 BST 中搜索元素

力扣第 700 题「二叉搜索树中的搜索」就是让你在 BST 中搜索值为 `target` 的节点，函数签名如下：

```
TreeNode searchBST(TreeNode root, int target);
```

如果是在一棵普通的二叉树中寻找，可以这样写代码：

```
TreeNode searchBST(TreeNode root, int target) {  
    if (root == null) return null;  
    if (root.val == target) return root;  
    // 当前节点没找到就递归地去左右子树寻找  
    TreeNode left = searchBST(root.left, target);  
    TreeNode right = searchBST(root.right, target);  
  
    return left != null ? left : right;  
}
```

这样写完全正确，但这段代码相当于穷举了所有节点，适用于所有二叉树。那么应该如何充分利用 BST 的特殊性，把「左小右大」的特性用上？

很简单，其实不需要递归地搜索两边，类似二分查找思想，根据 `target` 和 `root.val` 的大小比较，就能排除一边。我们把上面的思路稍稍改动：

```
TreeNode searchBST(TreeNode root, int target) {  
    if (root == null) {  
        return null;  
    }  
    // 去左子树搜索  
    if (root.val > target) {  
        return searchBST(root.left, target);  
    }  
    // 去右子树搜索  
    if (root.val < target) {  
        return searchBST(root.right, target);  
    }  
    // 当前节点就是目标值  
    return root;  
}
```

▶ 🎃 代码可视化动画🎃

## 在 BST 中插入一个数

对数据结构的操作无非遍历 + 访问，遍历就是「找」，访问就是「改」。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

因为 BST 一般不会存在值重复的节点，所以我们一般不会在 BST 中插入已存在的值。下面的代码都默认不会向 BST 中插入已存在的值。

上一个问题，我们总结了 BST 中的遍历框架，就是「找」的问题。直接套框架，加上「改」的操作即可。

一旦涉及「改」，就类似二叉树的构造问题，函数要返回 **TreeNode** 类型，并且要对递归调用的返回值进行接收。

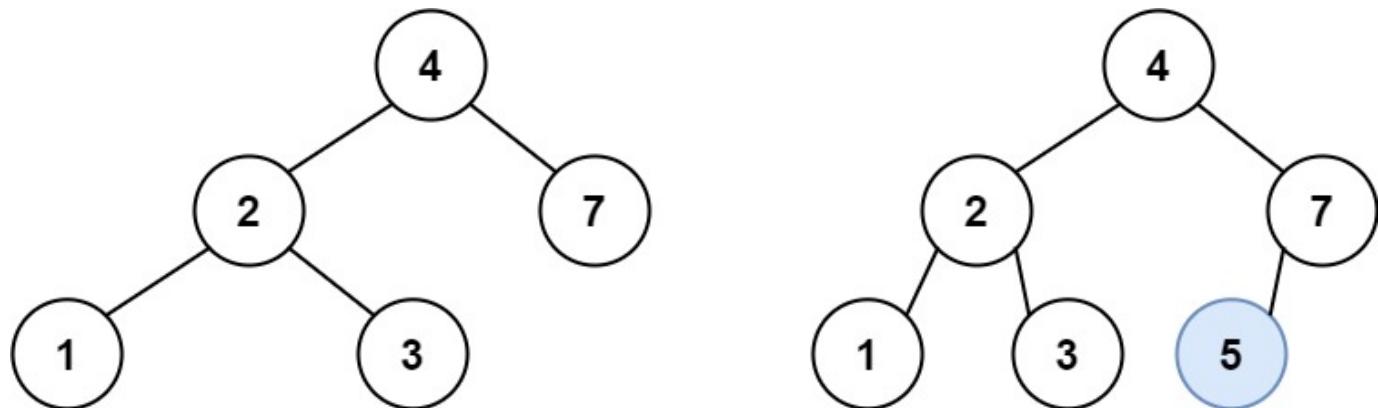
力扣第 701 题「二叉搜索树中的插入操作」就是这个问题：

▼ 701. 二叉搜索树中的插入操作 [Leetcode](#) | [力扣](#)

给定二叉搜索树（BST）的根节点 **root** 和要插入树中的值 **value**，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。输入数据 **保证**，新值和原始二叉搜索树中的任意节点值都不同。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回 **任意有效的结果**。

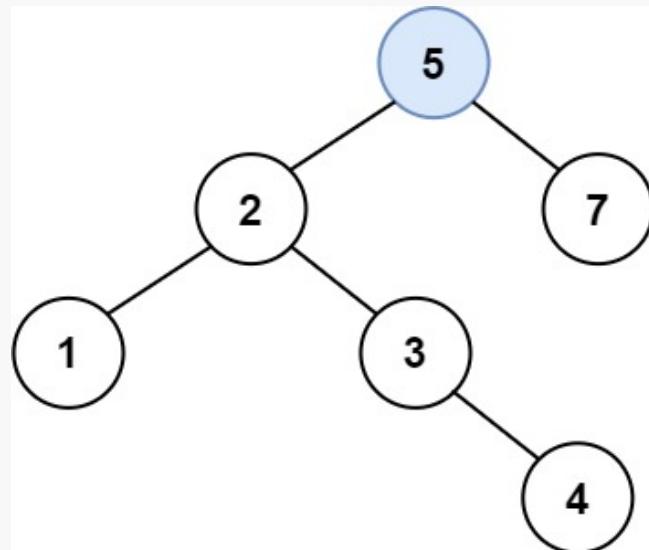
示例 1：



输入: `root = [4,2,7,1,3], val = 5`

输出: `[4,2,7,1,3,5]`

解释: 另一个满足题目要求可以通过的树是:



示例 2：

输入: `root = [40,20,60,10,30,50,70], val = 25`

输出: `[40,20,60,10,30,50,70,null,null,25]`

示例 3：

```
输入: root = [4,2,7,1,3,null,null,null,null,null], val = 5
输出: [4,2,7,1,3,5]
```

提示:

- 树中的节点数将在  $[0, 10^4]$  的范围内。
- $-10^8 \leq \text{Node.val} \leq 10^8$
- 所有值 `Node.val` 是 **独一无二** 的。
- $-10^8 \leq \text{val} \leq 10^8$
- 保证 `val` 在原始BST中不存在。

直接看解法代码吧，可以结合注释和可视化面板的来理解：

```
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) {
            // 找到空位置插入新节点
            return new TreeNode(val);
        }

        // 去右子树找插入位置
        if (root.val < val) {
            root.right = insertIntoBST(root.right, val);
        }
        // 去左子树找插入位置
        if (root.val > val) {
            root.left = insertIntoBST(root.left, val);
        }
        // 返回 root, 上层递归会接收返回值作为子节点
        return root;
    }
}
```

---

►  代码可视化动画 

### 三、在 BST 中删除一个数

力扣第 450 题「删除二叉搜索树中的节点」就是让你在 BST 中删除一个值为 `key` 的节点：

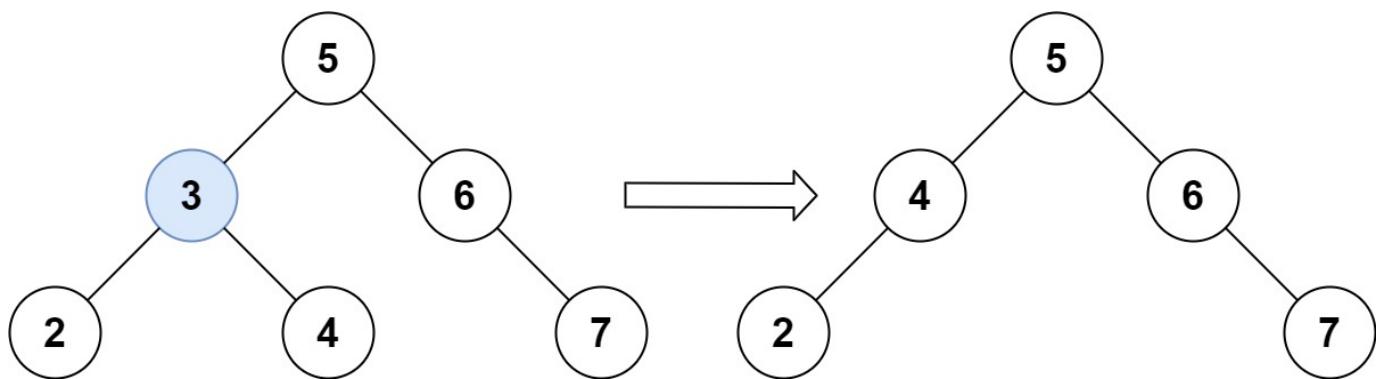
▼ 450. 删除二叉搜索树中的节点 [Leetcode | 力扣](#)

给定一个二叉搜索树的根节点 `root` 和一个值 `key`，删除二叉搜索树中的 `key` 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1. 首先找到需要删除的节点；
2. 如果找到了，删除它。

示例 1:



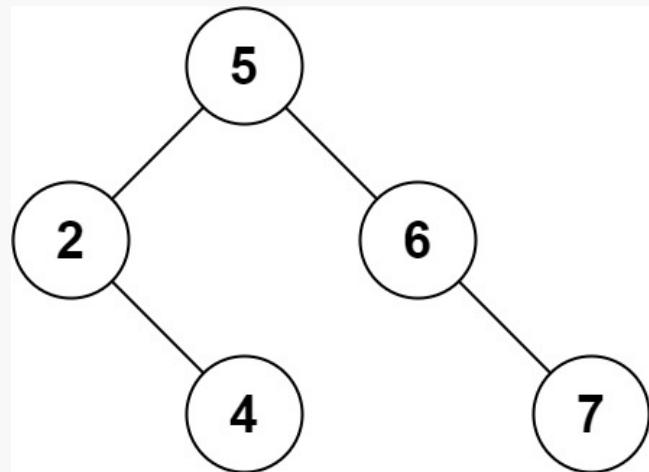
输入: root = [5,3,6,2,4,null,7], key = 3

输出: [5,4,6,2,null,null,7]

解释: 给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7]，如下图所示。

另一个正确答案是 [5,2,6,null,4,null,7]。



### 示例 2:

输入: root = [5,3,6,2,4,null,7], key = 0

输出: [5,3,6,2,4,null,7]

解释: 二叉树不包含值为 0 的节点

### 示例 3:

输入: root = [], key = 0

输出: []

### 提示:

- 节点数的范围  $[0, 10^4]$ .
- $-10^5 \leq \text{Node.val} \leq 10^5$
- 节点值唯一
- `root` 是合法的二叉搜索树
- $-10^5 \leq \text{key} \leq 10^5$

进阶：要求算法时间复杂度为  $O(h)$ ,  $h$  为树的高度。

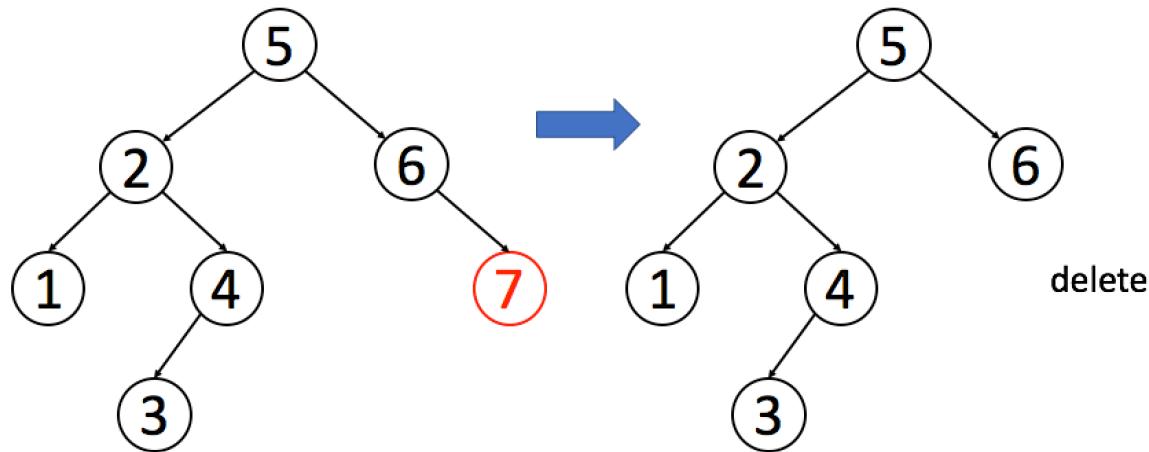
这个问题稍微复杂，跟插入操作类似，先「找」再「改」，先把框架写出来再说：

```
TreeNode deleteNode(TreeNode root, int key) {  
    if (root.val == key) {  
        // 找到啦，进行删除  
    } else if (root.val > key) {  
        // 去左子树找  
        root.left = deleteNode(root.left, key);  
    } else if (root.val < key) {  
        // 去右子树找  
        root.right = deleteNode(root.right, key);  
    }  
    return root;  
}
```

找到目标节点了，比方说是节点 A，如何删除这个节点，这是难点。因为删除节点的同时不能破坏 BST 的性质。有三种情况，用图片来说明。

情况 1：A 恰好是末端节点，两个子节点都为空，那么它可以当场去世了。

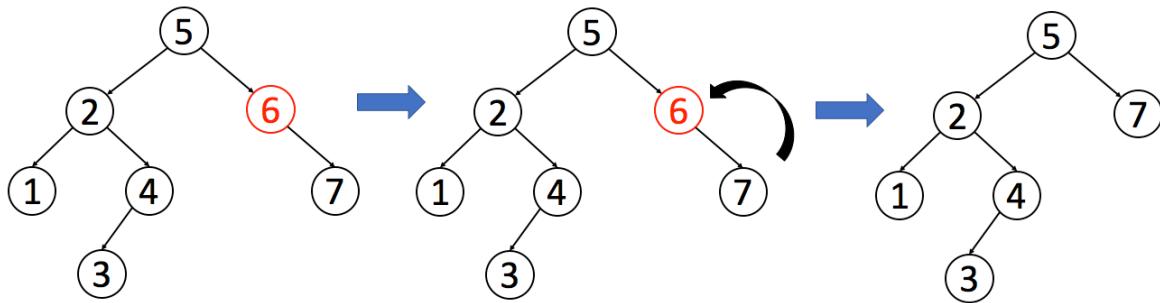
### Case 1: No Child



```
if (root.left == null && root.right == null)  
    return null;
```

情况 2：A 只有一个非空子节点，那么它要让这个孩子接替自己的位置。

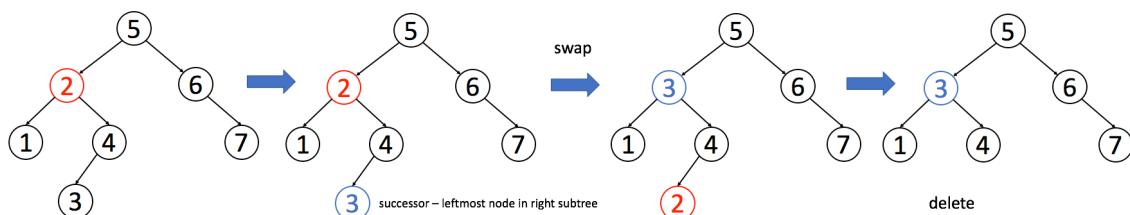
### Case 2: One Child



```
// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;
```

**情况 3:** A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式讲解。

### Case 3: Two Children



```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 转而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

三种情况分析完毕，填入框架，简化一下代码：

```
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return null;
        if (root.val == key) {
            // 这两个 if 把情况 1 和 2 都正确处理了
            if (root.left == null) return root.right;
            if (root.right == null) return root.left;
            // 处理情况 3
            // 获得右子树最小的节点
            TreeNode minNode = getMin(root.right);
            // 删除右子树最小的节点
            root.right = deleteNode(root.right, minNode.val);
        }
    }
}
```

```
// 用右子树最小的节点替换 root 节点
minNode.left = root.left;
minNode.right = root.right;
root = minNode;
} else if (root.val > key) {
    root.left = deleteNode(root.left, key);
} else if (root.val < key) {
    root.right = deleteNode(root.right, key);
}
return root;
}

TreeNode getMin(TreeNode node) {
    // BST 最左边的就是最小的
    while (node.left != null) node = node.left;
    return node;
}
}
```

## ▶ 代码可视化动画

这样，删除操作就完成了。注意一下，上述代码在处理情况 3 时通过一系列略微复杂的链表操作交换 `root` 和 `minNode` 两个节点：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
minNode.left = root.left;
minNode.right = root.right;
root = minNode;
```

有的读者可能会疑惑，替换 `root` 节点为什么这么麻烦，直接改 `val` 字段不就行了？看起来还更简洁易懂：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
root.val = minNode.val;
```

但对于这道算法题来说是可以的，但这样操作并不完美，我们一般不会通过修改节点内部的值来交换节点。因为在实际应用中，BST 节点内部的数据域是用户自定义的，可以非常复杂，而 BST 作为数据结构（一个工具人），其操作应该和内部存储的数据域解耦，所以我们更倾向于使用指针操作来交换节点，根本没必要关心内部数据。

最后简单总结一下吧，通过这篇文章，我们总结出了如下几个技巧：

1、如果当前节点会对下面的子节点有整体影响，可以通过辅助函数增长参数列表，借助参数传递信息。

- 2、掌握 BST 的增删查改方法。
- 3、递归修改数据结构时，需要对递归调用的返回值进行接收，并返回修改后的节点。

本文就到这里，更多经典的二叉树习题以及递归思维的训练，请参见二叉树章节中的 [递归专项练习](#)

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer 33. 二叉搜索树的后序遍历序列</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二叉搜索树心法（构造篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">96. Unique Binary Search Trees</a>	<a href="#">96. 不同的二叉搜索树</a>	简单
<a href="#">95. Unique Binary Search Trees II</a>	<a href="#">95. 不同的二叉搜索树 II</a>	简单

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

之前写了两篇手把手刷 BST 算法题的文章，[第一篇](#) 讲了中序遍历对 BST 的重要意义，[第二篇](#) 写了 BST 的基本操作。

本文就来写手把手刷 BST 系列的第三篇，循序渐进地讲两道题，如何计算所有有效 BST。

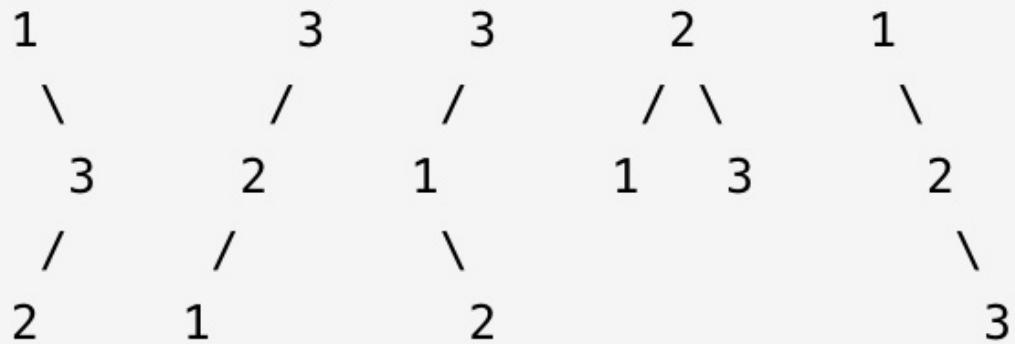
第一道题是力扣第 96 题「不同的二叉搜索树」，给你输入一个正整数  $n$ ，请你计算，存储  $\{1, 2, 3, \dots, n\}$  这些值共有多少种不同的 BST 结构。

函数签名如下：

```
int numTrees(int n);
```

比如说输入  $n = 3$ ，算法返回 5，因为共有如下 5 种不同的 BST 结构存储  $\{1, 2, 3\}$ ：

$n = 3$  时有如下 5 种不同的 BST 结果：



这就是一个正宗的穷举问题，那么什么方式能够正确地穷举有效 BST 的数量呢？

我们前文说过，不要小看「穷举」，这是一件看起来简单但是比较有技术含量的事情，问题的关键就是不能数漏，也不能数多，你咋整？

之前 [手把手刷二叉树第一期](#) 说过，二叉树算法的关键就在于明确根节点需要做什么，其实 BST 作为一种特殊的二叉树，核心思路也是一样的。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 二叉搜索树心法（后序篇）



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1373. Maximum Sum BST in Binary Tree</a>	<a href="#">1373. 二叉搜索子树的最大键值和</a>	

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历
- 二叉树心法（纲领篇）

本文是承接 [二叉树心法（纲领篇）](#) 的第五篇文章，主要讲二叉树后序位置的妙用，复述下前文关于后序遍历的描述：

前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码不仅可以获取参数数据，还可以获取到子树通过函数返回值传递回来的数据。

那么换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了。

其实二叉树的题目真的不难，无非就是前中后序遍历框架来回倒嘛，只要你把一个节点该做的事情安排好，剩下的抛给递归框架即可。

但是对于有的题目，不同的遍历顺序时间复杂度不同。尤其是这个后序位置的代码，有时候可以大幅提升算法效率。

我们再看看后序遍历的代码框架：

```
void traverse(TreeNode root) {  
    traverse(root.left);  
    traverse(root.right);  
    // 后序代码的位置  
    // 在这里处理当前节点  
}
```

看这个代码框架，你说什么情况下需要在后序位置写代码呢？

如果当前节点要做的事情需要通过左右子树的计算结果推导出来，就要用到后序遍历。

下面就讲一个经典的算法问题，可以直观地体会到后序位置的妙用。这是力扣第 1373 题「二叉搜索子树的最大键值和」，函数签名如下：

```
int maxSumBST(TreeNode root);
```

本文为 [labuladong.online](#) 网站会员内容, 请 [点这里](#) 查看。

## 【强化练习】用「遍历」思维解题 I

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（思路篇）
- 二叉树心法（纲领篇）

一般来说，如果让你在二叉树的「树枝」上做文章，那么用遍历的思维模式解题是比较自然的想法。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】用「遍历」思维解题 II

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（思路篇）
- 二叉树心法（纲领篇）

如果让你在二叉树中的某些节点上做文章，一般来说也可以直接用遍历的思维模式。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】用「遍历」思维解题 III

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（思路篇）
- 二叉树心法（纲领篇）

二叉树的题目还会夹杂着一些其他算法技巧一起考你，不过都离不开二叉树的遍历框架，看几道例题，体会一下。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】用「分解问题」思维解题 I

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（构造篇）
- 二叉树心法（纲领篇）

最常见的，二叉树的构造问题一般都会用到分解问题的思维模式。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】用「分解问题」思维解题 II

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（构造篇）
- 二叉树心法（纲领篇）

### 技巧一

类似于判断镜像二叉树、翻转二叉树的问题，一般也可以用分解问题的思路，无非就是把整棵树的问题（原问题）分解成子树之间的问题（子问题）。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】同时运用两种思维解题

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（纲领篇）

有的题目可以同时用「遍历」和「分解问题」两种思路来解，你可以利用这些题目训练自己的思维。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】利用后序位置解题 I

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（后序篇）
- 二叉树心法（纲领篇）

有些题目，你按照拍脑袋的方式去做，可能发现需要在递归代码中调用其他递归函数计算字数的信息。一般来说，出现这种情况时你可以考虑用后序遍历的思维方式来优化算法，利用后序遍历传递子树的信息，避免过高的时间复杂度。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】利用后序位置解题 II

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（后序篇）
- 二叉树心法（纲领篇）

像求和、求高度这种基本的二叉树函数很容易写，有时候只要在它们的后序位置添加一点代码，就能得到我们想要的答案。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】利用后序位置解题 III

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（后序篇）
- 二叉树心法（纲领篇）

写在后序位置的代码是最潇洒的，上通父节点（可以通过函数参数获取父节点信息），下通子树（可以通过递归返回值收集子树信息），有少部分难度比较大的题目会同时用到这两个特性。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】运用层序遍历解题 I

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（纲领篇）

二叉树大部分题目都可以用递归的算法解决，但少部分题目用递归比较麻烦的话，我们可以考虑使用层序遍历的方式解决。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】运用层序遍历解题 II

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉树心法（纲领篇）

有些二叉树的特殊题型需要用层序遍历的方式来做，比如下面列举的几道题。这类问题你就当开阔眼界有个印象就行了，不会有太多变体。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】二叉搜索树经典例题 I

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉搜索树心法（特性篇）
- 二叉树心法（纲领篇）

下面列出的 BST 题目主要考察 BST 左小右大、中序遍历有序的特点。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】二叉搜索树经典例题 II

阅读本文前，你需要先学习：

- 二叉树的遍历基础
- 二叉搜索树心法（基操篇）
- 二叉树心法（纲领篇）

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：最近公共祖先系列解题框架



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">236. Lowest Common Ancestor of a Binary Tree</a>	236. 二叉树的最近公共祖先	
<a href="#">235. Lowest Common Ancestor of a Binary Search Tree</a>	235. 二叉搜索树的最近公共祖先	
<a href="#">1676. Lowest Common Ancestor of a Binary Tree IV</a>	1676. 二叉树的最近公共祖先 IV	
<a href="#">1650. Lowest Common Ancestor of a Binary Tree III</a>	1650. 二叉树的最近公共祖先 III	
<a href="#">1644. Lowest Common Ancestor of a Binary Tree II</a>	1644. 二叉树的最近公共祖先 II	

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的 DFS/BFS 遍历
- 二叉树心法（纲领篇）

如果说笔试的时候经常遇到各种动归回溯这类稍有难度的题目，那么面试会倾向于一些比较经典的问题，难度不算大，而且也比较实用。

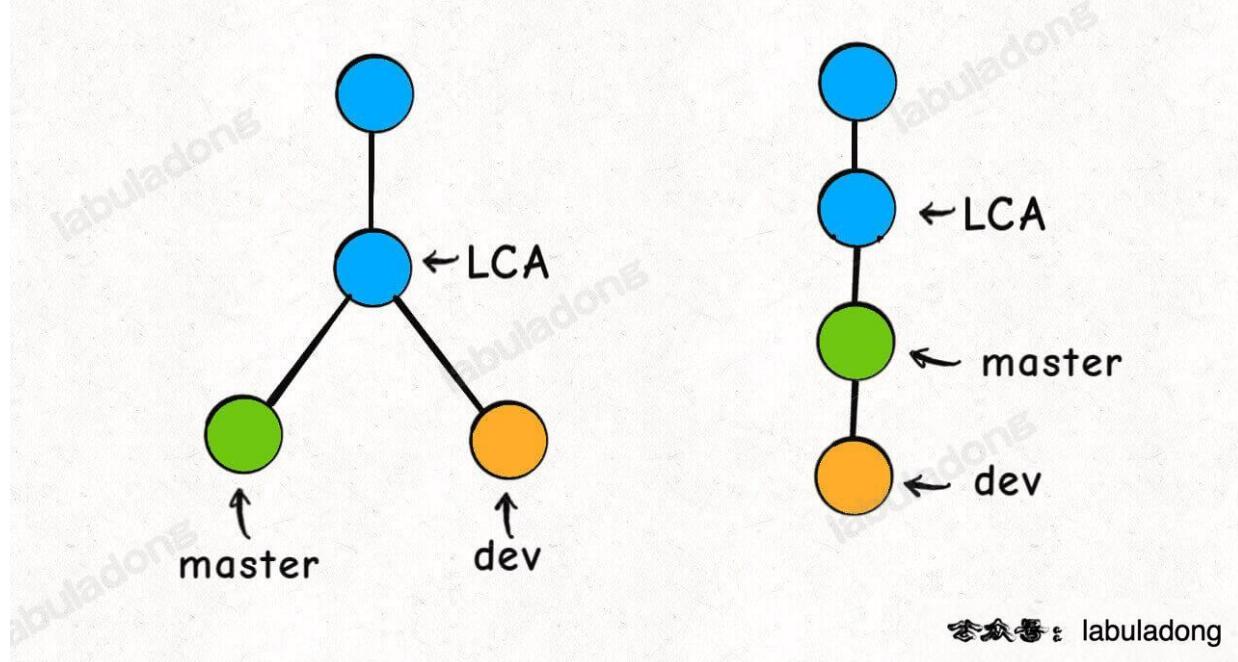
本文就用 Git 引出一个经典的算法问题：最近公共祖先（Lowest Common Ancestor，简称 LCA）。

`git pull` 这个命令我们经常会用，它默认是使用 `merge` 方式将远端别人的修改拉到本地；如果带上参数 `git pull -r`，就会使用 `rebase` 的方式将远端修改拉到本地。

这二者最直观的区别就是：`merge` 方式合并的分支会看到很多「分叉」，而 `rebase` 方式合并的分支就是一条直线。但无论哪种方式，如果存在冲突，Git 都会检测出来并让你手动解决冲突。

那么问题来了，Git 是如何检测两条分支是否存在冲突的呢？

以 `rebase` 命令为例，比如下图的情况，我站在 `dev` 分支执行 `git rebase master`，然后 `dev` 就会接到 `master` 分支之上：



© labuladong

这个过程中，Git 是这么做的：

首先，找到这两条分支的最近公共祖先 **LCA**，然后从 **master** 节点开始，重演 **LCA** 到 **dev** 几个 **commit** 的修改，如果这些修改和 **LCA** 到 **master** 的 **commit** 有冲突，就会提示你手动解决冲突，最后的结果就是把 **dev** 的分支完全接到 **master** 上面。

那么，Git 是如何找到两条不同分支的最近公共祖先的呢？这就是一个经典的算法问题了，下面我来由浅入深讲一讲。

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

## 拓展：如何计算完全二叉树的节点数



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">222. Count Complete Tree Nodes</a>	<a href="#">222. 完全二叉树的节点个数</a>	简单

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

如果让你数一下一棵普通二叉树有多少个节点，这很简单，只要在二叉树的遍历框架上加一点代码就行了。

但是，力扣第 222 题「完全二叉树的节点个数」给你一棵完全二叉树，让你计算它的节点个数，你会不会？算法的时间复杂度是多少？

这个算法的时间复杂度应该是  $O(\log N * \log N)$ ，如果你心中的算法没有达到这么高效，那么本文就是给你写的。

关于「完全二叉树」和「满二叉树」等名词的定义，可以参考基础知识章节的 [二叉树基础](#)。

### 一、思路分析

现在回归正题，如何求一棵完全二叉树的节点个数呢？

```
// 输入一棵完全二叉树，返回节点总数
int countNodes(TreeNode root);
```

如果是一个普通二叉树，显然只要向下面这样遍历一边即可，时间复杂度  $O(N)$ ：

```
public int countNodes(TreeNode root) {
    if (root == null) return 0;
    return 1 + countNodes(root.left) + countNodes(root.right);
}
```

那如果是一棵满二叉树，节点总数就和树的高度呈指数关系：

```
public int countNodes(TreeNode root) {
    int h = 0;
```

```
// 计算树的高度
while (root != null) {
    root = root.left;
    h++;
}
// 节点总数就是  $2^h - 1$ 
return (int)Math.pow(2, h) - 1;
}
```

完全二叉树比普通二叉树特殊，但又没有满二叉树那么特殊，计算它的节点总数，可以说是普通二叉树和完全二叉树的结合版，先看代码：

```
class Solution {
    public int countNodes(TreeNode root) {
        TreeNode l = root, r = root;
        // 沿最左侧和最右侧分别计算高度
        int hl = 0, hr = 0;
        while (l != null) {
            l = l.left;
            hl++;
        }
        while (r != null) {
            r = r.right;
            hr++;
        }
        // 如果两侧计算的高度相同，则是一棵满二叉树
        if (hl == hr) {
            return (int)Math.pow(2, hl) - 1;
        }
        // 如果两侧的高度不同，则按照普通二叉树的逻辑计算
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
}
```

## ▶ 🔍 代码可视化动画🔍

结合刚才针对满二叉树和普通二叉树的算法，上面这段代码应该不难理解，就是一个结合版，但是其中降低时间复杂度的技巧是非常微妙的。

## 二、复杂度分析

开头说了，这个算法的时间复杂度是  $O(\log N * \log N)$ ，这是怎么算出来的呢？

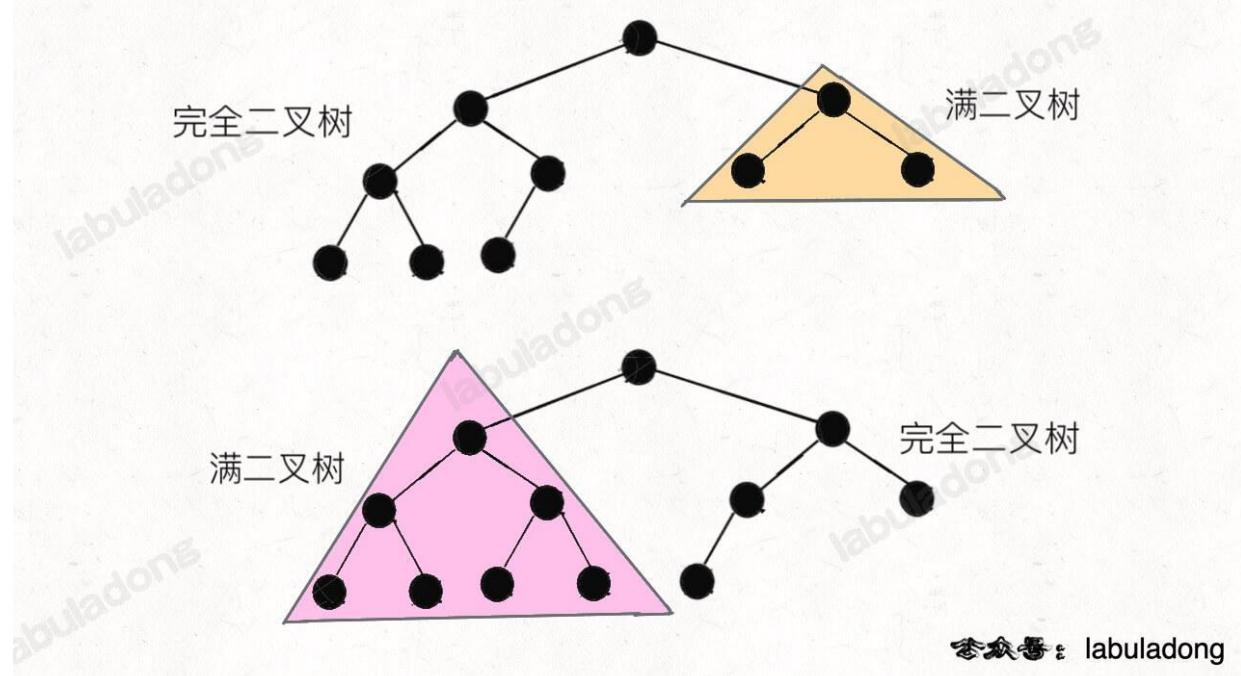
直觉感觉好像最坏情况下是  $O(N * \log N)$  吧，因为之前的 while 需要  $\log N$  的时间，最后要  $O(N)$  的时间向左右子树递归：

```
return 1 + countNodes(root.left) + countNodes(root.right);
```

关键点在于，这两个递归只有一个会真的递归下去，另一个一定会触发  $hl == hr$  而立即返回，不会递归下去。

为什么呢？原因如下：

一棵完全二叉树的两棵子树，至少有一棵是满二叉树：



看图就明显了吧，由于完全二叉树的性质，其子树一定有一棵是满的，所以一定会触发 `hl == hr`，只消耗  $O(\log N)$  的复杂度而不会继续递归。

综上，算法的递归深度就是树的高度  $O(\log N)$ ，每次递归所花费的时间就是 while 循环，需要  $O(\log N)$ ，所以总体的时间复杂度是  $O(\log N * \log N)$ 。

所以说，「完全二叉树」这个概念还是有它存在的原因的，不仅适用于数组实现二叉堆，而且连计算节点总数这种看起来简单的操作都有高效的算法实现。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 拓展：惰性展开多叉树



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">341. Flatten Nested List Iterator</a>	341. 扁平化嵌套列表迭代器	简单

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的遍历框架
- 多叉树结构及遍历框架

今天来讲一道非常有启发性的设计题目，为什么说它有启发性，我们后面再说。

### 一、题目描述

这是力扣第 341 题「扁平化嵌套列表迭代器」，我来描述一下题目：

首先，现在有一种数据结构 `NestedInteger`，这个结构中存的数据可能是一个 `Integer` 整数，也可能是一个 `NestedInteger` 列表。注意，这个列表里面装着的是 `NestedInteger`，也就是说这个列表中的每一个元素可能是个整数，可能又是个列表，这样无限递归嵌套下去……

`NestedInteger` 有如下 API：

```
public class NestedInteger {  
    // 如果其中存的是一个整数，则返回 true，否则返回 false  
    public boolean isInteger();  
  
    // 如果其中存的是一个整数，则返回这个整数，否则返回 null  
    public Integer getInteger();  
  
    // 如果其中存的是一个列表，则返回这个列表，否则返回 null  
    public List<NestedInteger> getList();  
}
```

我们的算法会被输入一个 `NestedInteger` 列表，我们需要做的就是写一个迭代器类，将这个带有嵌套结构 `NestedInteger` 的列表「拍平」：

```
public class NestedIterator implements Iterator<Integer> {
    // 构造器输入一个 NestedInteger 列表
    public NestedIterator(List<NestedInteger> nestedList) {}

    // 返回下一个整数
    public Integer next() {}

    // 是否还有下一个元素?
    public boolean hasNext() {}
}
```

我们写的这个类会被这样调用，先调用 `hasNext` 方法，后调用 `next` 方法：

```
NestedIterator i = new NestedIterator(nestedList);
while (i.hasNext())
    print(i.next());
```

题目给的几个示例如下：

示例 1：

输入：nestedList = [[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：nestedList = [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]。

比如示例 1，输入的列表里有三个 `NestedInteger`，两个列表型的 `NestedInteger` 和一个整型的 `NestedInteger`。

学过设计模式的朋友应该知道，迭代器也是设计模式的一种，目的就是为调用者屏蔽底层数据结构的细节，简单地通过 `hasNext` 和 `next` 方法有序地进行遍历。

为什么说这个题目很有启发性呢？因为我最近在用一款类似印象笔记的软件，叫做 Notion（挺有名的）。这个软件的一个亮点就是「万物皆 block」，比如说标题、页面、表格都是 block。有的 block 甚至可以无限嵌套，这就打破了传统笔记本「文件夹」->「笔记本」->「笔记」的三层结构。

回想这个算法问题，`NestedInteger` 结构实际上也是一种支持无限嵌套的结构，而且可以同时表示整数和列表两种不同类型，我想 Notion 的核心数据结构 block 估计也是这样的一种设计思路。

那么话说回来，对于这个算法问题，我们怎么解决呢？`NestedInteger` 结构可以无限嵌套，怎么把这个结构「打平」，为迭代器的调用者屏蔽底层细节，得到扁平化的输出呢？

## 二、解题思路

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：归并排序详解及应用



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">315. Count of Smaller Numbers After Self</a>	<a href="#">315. 计算右侧小于当前元素的个数</a>	
<a href="#">493. Reverse Pairs</a>	<a href="#">493. 翻转对</a>	
<a href="#">912. Sort an Array</a>	<a href="#">912. 排序数组</a>	
<a href="#">327. Count of Range Sum</a>	<a href="#">327. 区间和的个数</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的遍历框架](#)
- [多叉树结构及遍历框架](#)
- [二叉树系列算法（纲领篇）](#)

一直都有很多读者说，想让我用框架思维讲一讲基本的排序算法，我觉得确实得讲讲，毕竟学习任何东西都讲求一个融会贯通，只有对其本质进行比较深刻的理解，才能运用自如。

本文就先讲归并排序，给一套代码模板，然后讲讲它在算法问题中的应用。阅读本文前我希望你读过前文 [手把手刷二叉树（纲领篇）](#)。

我在讲二叉树的时候，提了一嘴归并排序，说归并排序就是二叉树的后序遍历，当时就有很多读者留言说醍醐灌顶。

知道为什么很多读者遇到递归相关的算法就觉得烧脑吗？因为还处在「看山是山，看水是水」的阶段。

就说归并排序吧，如果给你看代码，让你脑补一下归并排序的过程，你脑子里会出现什么场景？

这是一个数组排序算法，所以你脑补一个数组的 GIF，在那一个个交换元素？如果是这样的话，那格局就低了。

但如果你脑海中浮现出的是一棵二叉树，甚至浮现出二叉树后序遍历的场景，那格局就高了，大概率掌握了我经常强调的 [框架思维](#)，用这种抽象能力学习算法就省劲多了。

那么，归并排序明明就是一个数组算法，和二叉树有什么关系？接下来我就具体讲讲。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 拓展：快速排序详解及应用



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">215. Kth Largest Element in an Array</a>	<a href="#">215. 数组中的第K个最大元素</a>	简单
<a href="#">912. Sort an Array</a>	<a href="#">912. 排序数组</a>	简单

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的遍历框架](#)
- [多叉树结构及遍历框架](#)
- [二叉树系列算法（纲领篇）](#)

前文 [归并排序算法详解](#) 通过二叉树的视角描述了归并排序的算法原理以及应用，很多读者大呼精妙，那我就趁热打铁，今天继续用二叉树的视角讲一讲快速排序算法的原理以及运用。

## 快速排序算法思路

首先我们看一下快速排序的代码框架：

```
void sort(int[] nums, int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    // 对 nums[lo..hi] 进行切分
    // 使得 nums[lo..p-1] <= nums[p] < nums[p+1..hi]
    int p = partition(nums, lo, hi);
    // 去左右子数组进行切分
    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}
```

其实你对比之后可以发现，快速排序就是一个二叉树的前序遍历：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
```

```
    return;
}
// ***** 前序位置 *****
print(root.val);
// *****
traverse(root.left);
traverse(root.right);
}
```

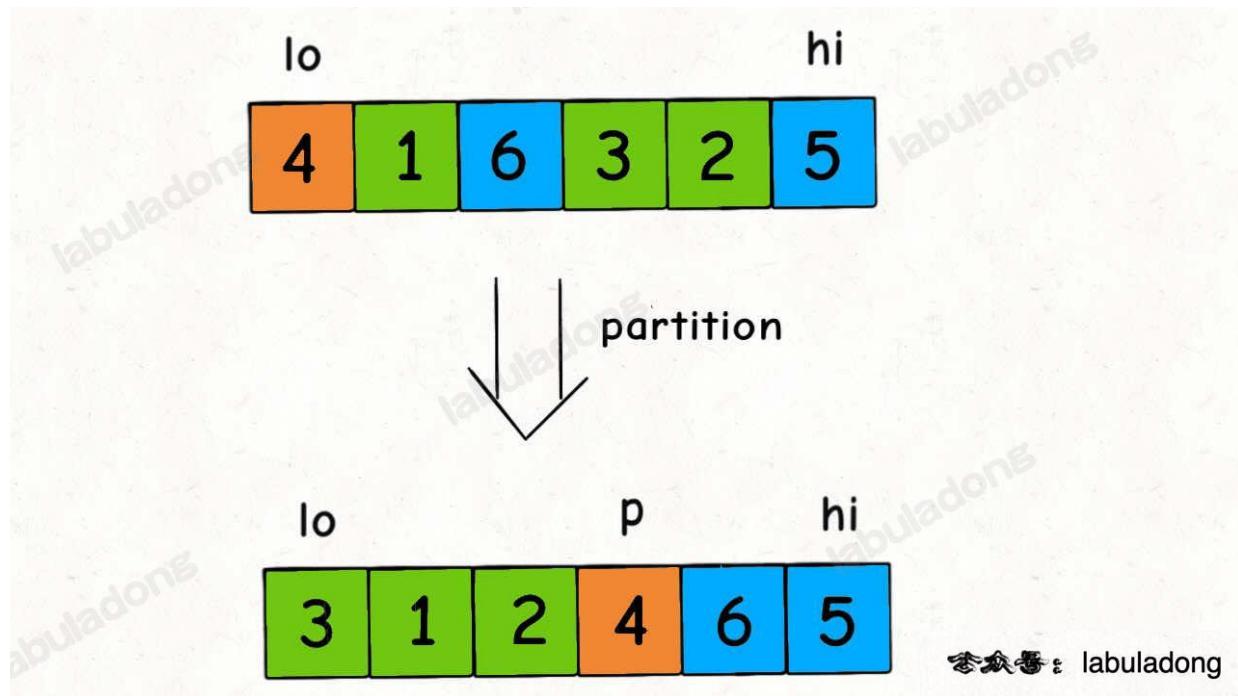
另外，前文 [归并排序详解](#) 用一句话总结了归并排序：先把左半边数组排好序，再把右半边数组排好序，然后把两半数组合并。

同时我提了一个问题，让你一句话总结快速排序，这里说一下我的答案：

**快速排序是先将一个元素排好序，然后再将剩下的元素排好序。**

为什么这么说呢，且听我慢慢道来。

快速排序的核心无疑是 `partition` 函数，`partition` 函数的作用是在 `nums[lo..hi]` 中寻找一个切分点 `p`，通过交换元素使得 `nums[lo..p-1]` 都小于等于 `nums[p]`，且 `nums[p+1..hi]` 都大于 `nums[p]`：



一个元素左边的元素都比它小，右边的元素都比它大，啥意思？不就是它自己已经被放到正确的位置上了吗？

所以 `partition` 函数干的事情，其实就是把 `nums[p]` 这个元素排好序了。

一个元素被排好序了，然后呢？你再把剩下的元素排好序不就得了。

剩下的元素有哪些？左边一坨，右边一坨，去吧，对子数组进行递归，用 `partition` 函数把剩下的元素也排好序。

从二叉树的视角，我们可以把子数组 `nums[lo..hi]` 理解成二叉树节点上的值，`sort` 函数理解成二叉树的遍历函数。

参照二叉树的前序遍历顺序，快速排序的运行过程如下 GIF：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：用栈模拟递归遍历二叉树



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的 DFS/BFS 遍历](#)

我们一般用 [递归遍历或者层序遍历的方式](#) 处理二叉树就完全够了。

本文介绍的利用栈迭代遍历二叉树的方法，本质上还是用栈手动模拟递归过程，在本站其他习题中都用不到，面试时也几乎没有面试官非要难为你写这种代码。

所以本文内容仅作为思维拓展，不要求必须掌握。如果不感兴趣，可以放心地跳过本文的内容。

[二叉树的 DFS/BFS 遍历](#) 介绍过二叉树的递归遍历和层序遍历方法，这两种方法是最简单实用的。

有些读者在后台问我如何将前中后序的递归框架改写成迭代形式。我以前背过一些迭代实现二叉树前中后序遍历的代码模板，比较短小，容易记，但通用性较差。

通用性较差的意思是说，模板只是针对「用迭代的方式返回二叉树前/中/后序的遍历结果」这个问题，函数签名类似这样，返回一个 [TreeNode](#) 列表：

```
List<TreeNode> traverse(TreeNode root);
```

如果给一些稍微复杂的二叉树问题，比如 [最近公共祖先](#)，[二叉搜索子树的最大键值和](#)，想把这些递归解法改成迭代，就无能为力了。

而我想要的是一个万能的模板，可以把一切二叉树递归算法都改成迭代。

换句话说，类似二叉树的递归框架：

```
void traverse(TreeNode root) {  
    if (root == null) return;  
    // 前序遍历代码位置  
    traverse(root.left);  
    // 中序遍历代码位置  
    traverse(root.right);  
    // 后序遍历代码位置  
}
```

迭代框架也应该有前中后序代码的位置：

```
void traverse(TreeNode root) {
    while (...) {
        if (...) {
            // 前序遍历代码位置
        }
        if (...) {
            // 中序遍历代码位置
        }
        if (...) {
            // 后序遍历代码位置
        }
    }
}
```

我如果想把递归改成迭代，直接把递归解法中前中后序对应位置的代码复制粘贴到迭代框架里，就可以直接运行，得到正确的结果。

理论上，所有递归算法都可以利用栈改成迭代的形式，因为计算机本质上就是借助栈来迭代地执行递归函数的。

所以本文就来利用「栈」模拟函数递归的过程，总结一套二叉树通用迭代遍历框架。

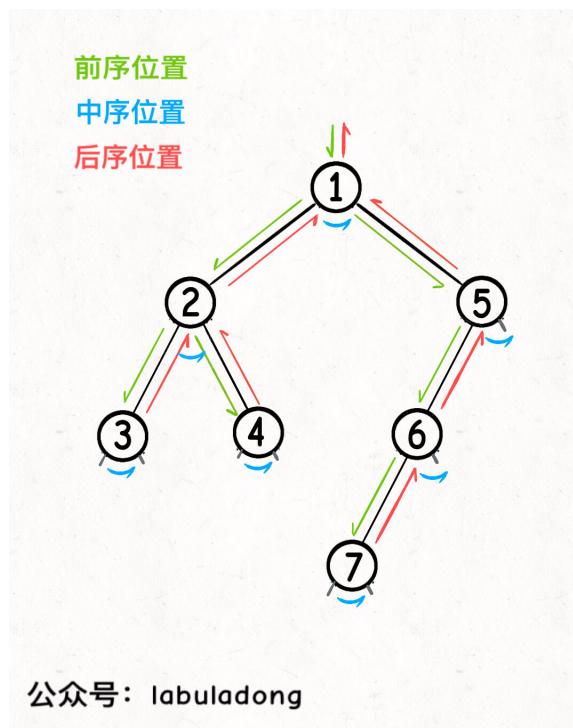
## 递归框架改为迭代

按照 [二叉树心法（纲领篇）](#)，二叉树的递归框架中，前中后序遍历位置就是几个特殊的时间点：

前序遍历位置的代码，会在刚遍历到当前节点 `root`，遍历 `root` 的左右子树之前执行；

中序遍历位置的代码，会在在遍历完当前节点 `root` 的左子树，即将开始遍历 `root` 的右子树的时候执行；

后序遍历位置的代码，会在遍历完以当前节点 `root` 为根的整棵子树之后执行。



如果从递归代码上来看，上述结论是很容易理解的：

```
void traverse(TreeNode root) {
    if (root == null) return;
    // 前序遍历代码位置
    traverse(root.left);
    // 中序遍历代码位置
    traverse(root.right);
    // 后序遍历代码位置
}
```

不过，如果我们想将递归算法改为迭代算法，就不能从框架上理解算法的逻辑，而要深入细节，思考计算机是如何进行递归的。

假设计算机运行函数 A，就会把 A 放到调用栈里面，如果 A 又调用了函数 B，则把 B 压在 A 上面，如果 B 又调用了 C，那就再把 C 压到 B 上面……

当 C 执行结束后，C 出栈，返回值传给 B，B 执行完后出栈，返回值传给 A，最后等 A 执行完，返回结果并出栈，此时调用栈为空，整个函数调用链结束。

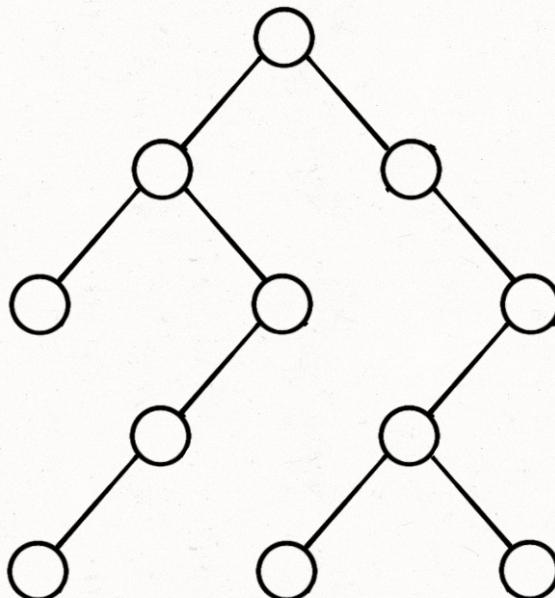
我们递归遍历二叉树的函数也是一样的，当函数被调用时，被压入调用栈，当函数结束时，从调用栈中弹出。

那么我们可以写出下面这段代码模拟递归调用的过程：

```
// 模拟系统的函数调用栈
Stack<TreeNode> stk = new Stack<>();

void traverse(TreeNode root) {
    if (root == null) return;
    // 函数开始时压入调用栈
    stk.push(root);
    traverse(root.left);
    traverse(root.right);
    // 函数结束时离开调用栈
    stk.pop();
}
```

如果在前序遍历的位置入栈，后序遍历的位置出栈，`stk` 中的节点变化情况就反映了 `traverse` 函数的递归过程（GIF 中绿色节点就是被压入栈中的节点，灰色节点就是弹出栈的节点）：



公众号：labuladong

简单说就是这样一个流程：

- 1、拿到一个节点，就一路向左遍历（因为 `traverse(root.left)` 排在前面），把路上的节点都压到栈里。
- 2、往左走到头之后就开始退栈，看看栈顶节点的右指针，非空的话就重复第 1 步。

写成迭代代码就是这样：

```
class Solution {  
    private Stack<TreeNode> stk = new Stack<>();  
  
    public List<Integer> traverse(TreeNode root) {  
        pushLeftBranch(root);  
  
        while (!stk.isEmpty()) {  
            TreeNode p = stk.pop();  
            pushLeftBranch(p.right);  
        }  
    }  
  
    // 左侧树枝一撸到底，都放入栈中  
    private void pushLeftBranch(TreeNode p) {  
        while (p != null) {  
            stk.push(p);  
            p = p.left;  
        }  
    }  
}
```

上述代码虽然已经可以模拟出递归函数的运行过程，不过还没有找到递归代码中的前中后序代码位置，所以需要进一步修改。

## 迭代代码框架

想在迭代代码中体现前中后序遍历，关键点在哪里？

当我从栈中拿出一个节点 **p**, 我应该想办法搞清楚这个节点 **p** 左右子树的遍历情况。

如果 **p** 的左右子树都没有被遍历, 那么现在对 **p** 进行操作就属于前序遍历代码。

如果 **p** 的左子树被遍历过了, 而右子树没有被遍历过, 那么现在对 **p** 进行操作就属于中序遍历代码。

如果 **p** 的左右子树都被遍历过了, 那么现在对 **p** 进行操作就属于后序遍历代码。

上述逻辑写成伪码如下:

```
class Solution {
    private Stack<TreeNode> stk = new Stack<>();

    public List<Integer> traverse(TreeNode root) {
        pushLeftBranch(root);

        while (!stk.isEmpty()) {
            TreeNode p = stk.peek();

            if (p 的左子树被遍历完了) {
                // *****
                // * 中序遍历代码位置 *
                // *****
                pushLeftBranch(p.right);
                // 去遍历 p 的右子树
            }

            if (p 的右子树被遍历完了) {
                // *****
                // * 后序遍历代码位置 *
                // *****
                stk.pop();
                // 以 p 为根的树遍历完了, 出栈
            }
        }
    }

    private void pushLeftBranch(TreeNode p) {
        while (p != null) {
            // *****
            // * 前序遍历代码位置 *
            // *****
            stk.push(p);
            p = p.left;
        }
    }
}
```

有刚才的铺垫, 这段代码应该是不难理解的, 关键是如何判断 **p** 的左右子树到底被遍历过没有呢?

其实很简单, 我们只需要维护一个 **visited** 指针, 指向「上一次遍历完成的根节点」, 就可以判断 **p** 的左右子树遍历情况了

下面是迭代遍历二叉树的完整代码框架:

```

class Solution {
    // 模拟函数调用栈
    private Stack<TreeNode> stk = new Stack<>();

    // 左侧树枝一撸到底
    private void pushLeftBranch(TreeNode p) {
        while (p != null) {
            // *****
            // * 前序遍历代码位置 *
            // *****
            stk.push(p);
            p = p.left;
        }
    }

    public List<Integer> traverse(TreeNode root) {
        // 指向上一次遍历完的子树根节点
        TreeNode visited = new TreeNode(-1);
        // 开始遍历整棵树
        pushLeftBranch(root);

        while (!stk.isEmpty()) {
            TreeNode p = stk.peek();

            // p 的左子树被遍历完了，且右子树没有被遍历过
            if ((p.left == null || p.left == visited)
                && p.right != visited) {
                // *****
                // * 中序遍历代码位置 *
                // *****
                // 去遍历 p 的右子树
                pushLeftBranch(p.right);
            }
            // p 的右子树被遍历完了
            if (p.right == null || p.right == visited) {
                // *****
                // * 后序遍历代码位置 *
                // *****
                // 以 p 为根的子树被遍历完了，出栈
                // visited 指针指向 p
                visited = stk.pop();
            }
        }
    }
}

```

代码中最有技巧性的是这个 `visited` 指针，它记录最近一次遍历完的子树根节点（最近一次 `pop` 出栈的节点），我们可以根据对比 `p` 的左右指针和 `visited` 是否相同来判断节点 `p` 的左右子树是否被遍历过，进而分离出前中后序的代码位置。

`visited` 指针初始化指向一个新 `new` 出来的二叉树节点，相当于一个特殊值，目的是避免和输入二叉树中的节点重叠。

只需把递归算法中的前中后序位置的代码复制粘贴到上述框架的对应位置，就可以把任意递归的二叉树算法改写成迭代形式了。

比如，让你返回二叉树后序遍历的结果，你就可以这样写：

```
class Solution {
    private Stack<TreeNode> stk = new Stack<>();

    public List<Integer> postorderTraversal(TreeNode root) {
        // 记录后序遍历的结果
        List<Integer> postorder = new ArrayList<>();
        TreeNode visited = new TreeNode(-1);

        pushLeftBranch(root);
        while (!stk.isEmpty()) {
            TreeNode p = stk.peek();

            if ((p.left == null || p.left == visited)
                && p.right != visited) {
                pushLeftBranch(p.right);
            }

            if (p.right == null || p.right == visited) {
                // 后序遍历代码位置
                postorder.add(p.val);
                visited = stk.pop();
            }
        }

        return postorder;
    }

    private void pushLeftBranch(TreeNode p) {
        while (p != null) {
            stk.push(p);
            p = p.left;
        }
    }
}
```

当然，任何一个二叉树的算法，如果你想把递归改成迭代，都可以套用这个框架，只要把递归的前中后序位置的代码对应过来就行了。

迭代解法到这里就搞定了，不过我还是想强调，除了 BFS 层级遍历之外，二叉树的题目还是用递归的方式来做，因为递归是最符合二叉树结构特点的。

说到底，这个迭代解法就是在用栈模拟递归调用，所以对照着递归解法，应该不难理解和记忆。

本文就到这里，更多经典的二叉树习题以及递归思维的训练，请参见二叉树章节中的 [递归专项练习](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
----------	----	----

LeetCode	力扣	难度
<a href="#">173. Binary Search Tree Iterator</a>	<a href="#">173. 二叉搜索树迭代器</a>	
-	<a href="#">剑指 Offer II 055. 二叉搜索树迭代器</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 队列实现栈以及栈实现队列



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

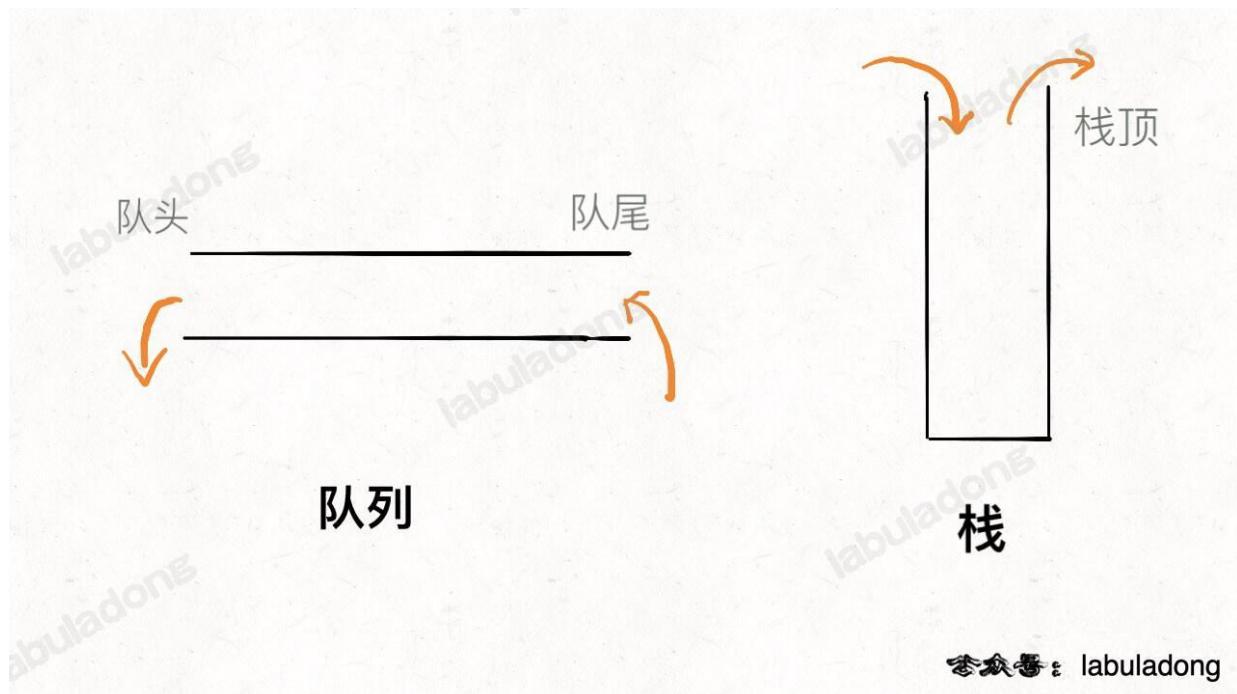
读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">225. Implement Stack using Queues</a>	225. 用队列实现栈	简单
<a href="#">232. Implement Queue using Stacks</a>	232. 用栈实现队列	简单
-	剑指 Offer 09. 用两个栈实现队列	简单

阅读本文前，你需要先学习：

- 数组基础
- 链表基础
- 队列基础

队列是一种先进先出的数据结构，栈是一种先进后出的数据结构，形象一点就是这样：



这两种数据结构底层其实都是数组或者链表实现的，只是 API 限定了它们的特性，具体实现可以参见基础知识章节的 [队列/栈的原理及实现](#)。

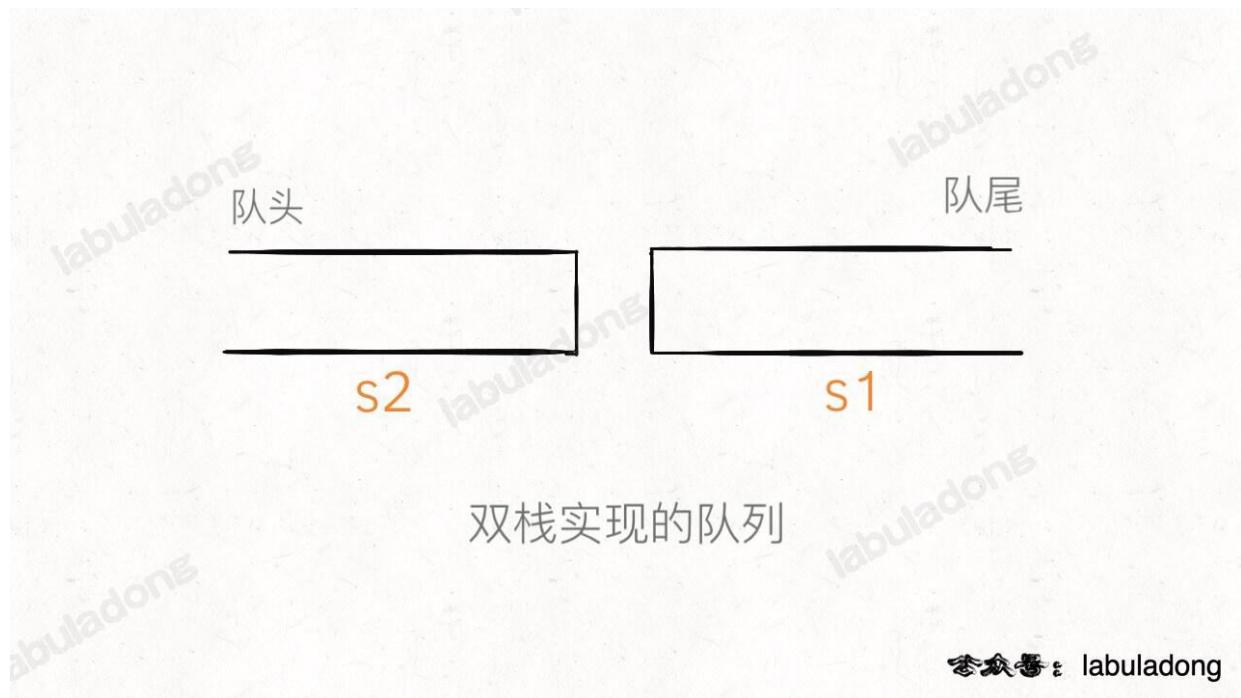
今天来看看如何使用「栈」的特性来实现一个「队列」，如何用「队列」实现一个「栈」。

## 一、用栈实现队列

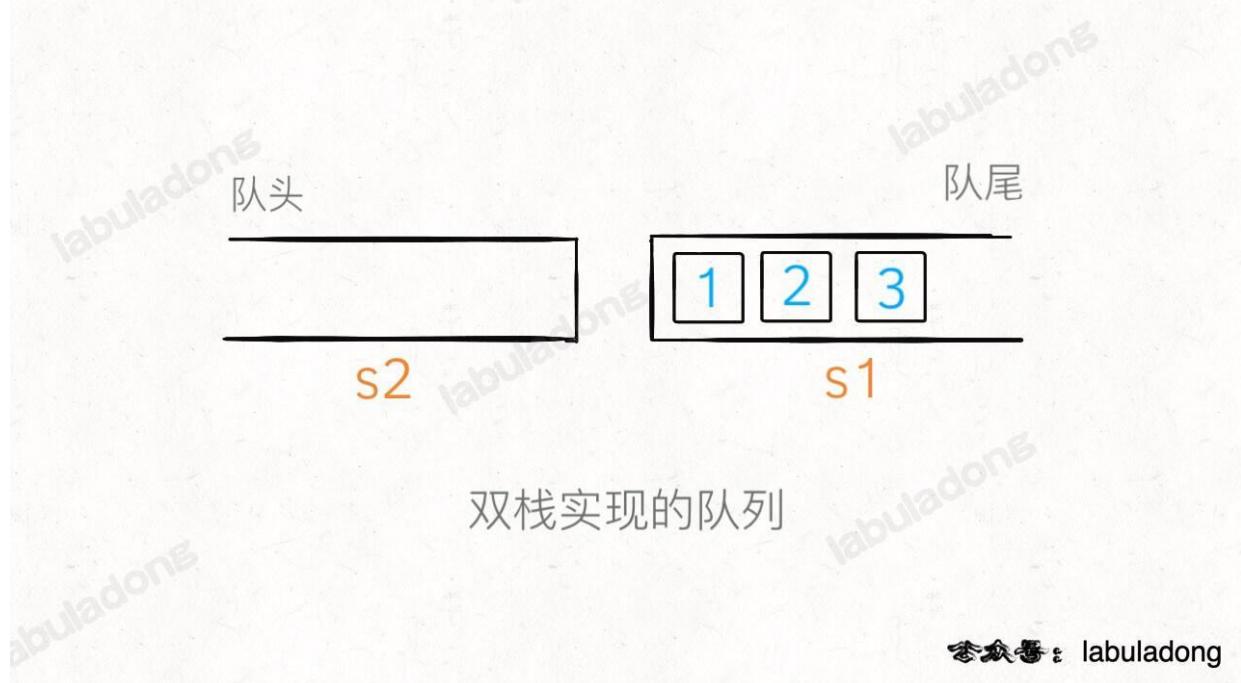
力扣第 232 题「用栈实现队列」让我们实现的 API 如下：

```
class MyQueue {  
  
    // 添加元素到队尾  
    public void push(int x);  
  
    // 删除队头的元素并返回  
    public int pop();  
  
    // 返回队头元素  
    public int peek();  
  
    // 判断队列是否为空  
    public boolean empty();  
}
```

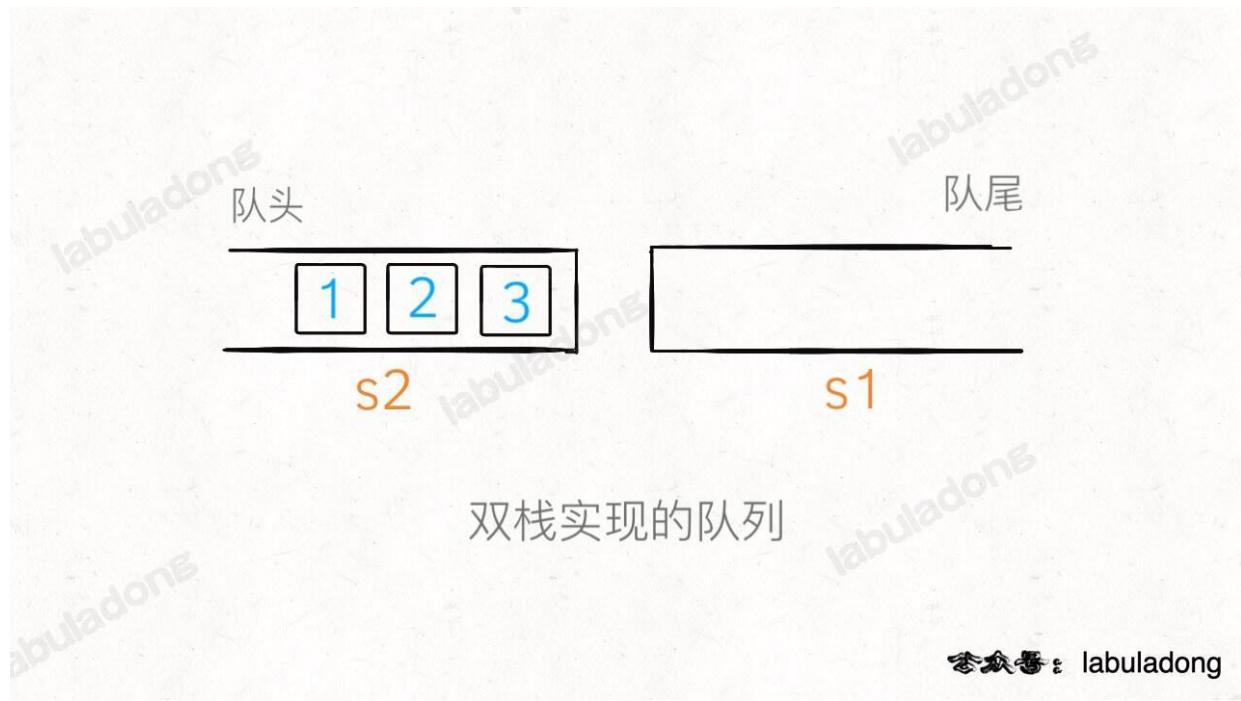
我们使用两个栈  $s_1$ ,  $s_2$  就能实现一个队列的功能（这样放置栈可能更容易理解）：



当调用  $\text{push}$  让元素入队时，只要把元素压入  $s_1$  即可，比如说  $\text{push}$  进 3 个元素分别是 1,2,3，那么底层结构就是这样：



那么如果这时候使用 `peek` 查看队头的元素怎么办呢？按道理队头元素应该是 1，但是在 `s1` 中 1 被压在栈底，现在就要轮到 `s2` 起到一个中转的作用了：当 `s2` 为空时，可以把 `s1` 的所有元素取出再添加进 `s2`，这时候 `s2` 中元素就是先进先出顺序了：



当 `s2` 中存在元素时，直接调用操作 `s2` 的 `pop` 方法，弹出的就是最先插入的元素，即实现了队列的 `pop` 操作。

完整代码如下：

```
class MyQueue {
    private Stack<Integer> s1, s2;

    public MyQueue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
}
```

```

// 添加元素到队尾
public void push(int x) {
    s1.push(x);
}

// 返回队头元素
public int peek() {
    if (s2.isEmpty())
        // 把 s1 元素压入 s2
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }
    return s2.peek();
}

// 删除队头元素并返回
public int pop() {
    // 先调用 peek 保证 s2 非空
    peek();
    return s2.pop();
}

// 判断队列是否为空
// 两个栈都为空才说明队列为空
public boolean empty() {
    return s1.isEmpty() && s2.isEmpty();
}
}

```

至此，就用栈结构实现了一个队列，核心思想是利用两个栈互相配合。

值得一提的是，这几个操作的时间复杂度是多少呢？

有点意思是 `peek` 操作，调用它时可能触发 `while` 循环，这样的话时间复杂度是  $O(N)$ ，但是大部分情况下 `while` 循环不会被触发，时间复杂度是  $O(1)$ 。由于 `pop` 操作调用了 `peek`，它的时间复杂度和 `peek` 相同。

像这种情况，可以说它们的**最坏时间复杂度**是  $O(N)$ ，因为包含 `while` 循环，**可能需要从 s1 往 s2 搬移元素**。

但是它们的**均摊时间复杂度**是  $O(1)$ ，这个要这么理解：对于一个元素，最多只可能被搬运一次，也就是说 `peek` 操作平均到每个元素的时间复杂度是  $O(1)$ 。

关于时间复杂度的分析方法，详见 [时空复杂度实用分析方法](#)。

## 二、用队列实现栈

如果说双栈实现队列比较巧妙，那么用队列实现栈就比较简单粗暴了，只需要一个队列作为底层数据结构就能实现了。

力扣第 225 题「用队列实现栈」让我们实现如下 API：

```

class MyStack {

    // 添加元素到栈顶
    public void push(int x);

    // 删除栈顶的元素并返回
    public int pop();
}

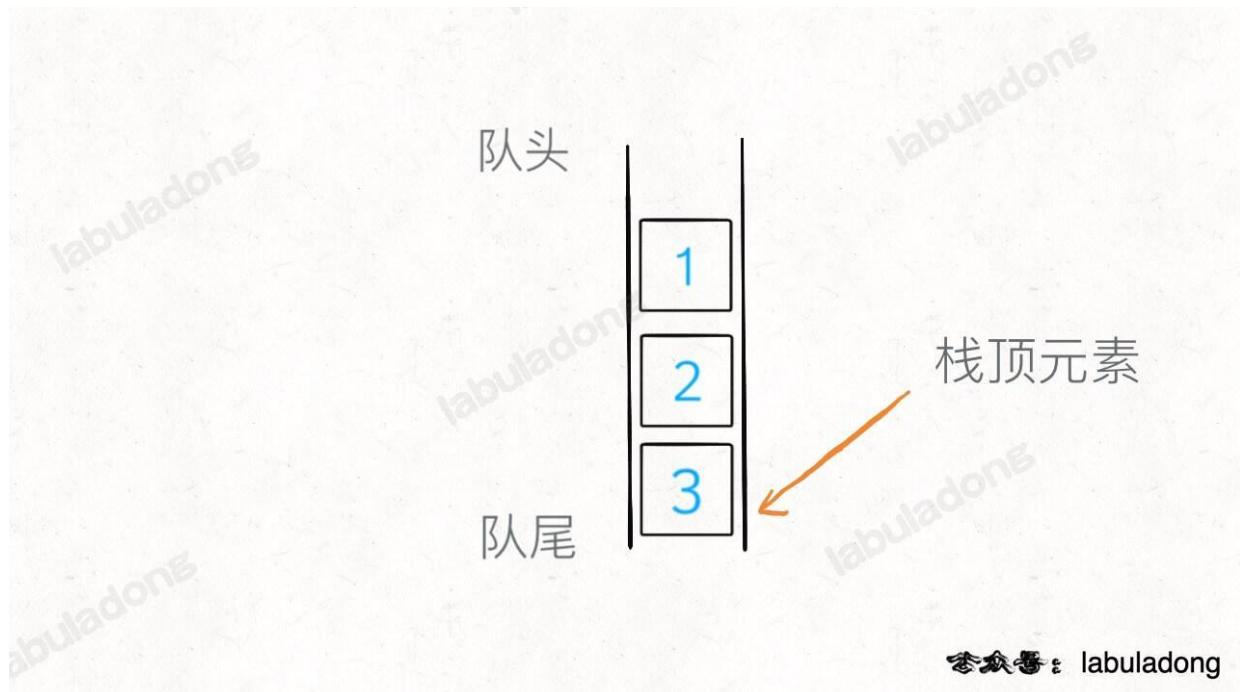
```

```
// 返回栈顶元素  
public int top();  
  
// 判断栈是否为空  
public boolean empty();  
}
```

先说 `push` API，直接将元素加入队列，同时记录队尾元素，因为队尾元素相当于栈顶元素，如果要 `top` 查看栈顶元素的话可以直接返回：

```
class MyStack {  
    Queue<Integer> q = new LinkedList<>();  
    int top_elem = 0;  
  
    // 添加元素到栈顶  
    public void push(int x) {  
        // x 是队列的队尾，是栈的栈顶  
        q.offer(x);  
        top_elem = x;  
    }  
  
    // 返回栈顶元素  
    public int top() {  
        return top_elem;  
    }  
  
    public boolean empty() {  
        return q.isEmpty();  
    }  
}
```

我们的底层数据结构是先进先出的队列，每次 `pop` 只能从队头取元素；但是栈是后进先出，也就是说 `pop` API 要从队尾取元素：



解决方法简单粗暴，把队列前面的都取出来再加入队尾，让之前的队尾元素排到队头，这样就可以取出了：



```
class MyStack {  
    // 为了节约篇幅，省略上文给出的代码部分...  
  
    // 删除栈顶的元素并返回  
    public int pop() {  
        int size = q.size();  
        while (size > 1) {  
            q.offer(q.poll());  
            size--;  
        }  
        // 之前的队尾元素已经到了队头  
        return q.poll();  
    }  
}
```

这样实现还有一点小问题就是，原来的队尾元素被推到队头并删除了，但是 `top_elem` 变量没有更新，我们还需要一点小修改：

```
class MyStack {  
    // 为了节约篇幅，省略上文给出的代码部分...  
  
    // 删除栈顶的元素并返回  
    public int pop() {  
        int size = q.size();  
        // 留下队尾 2 个元素  
        while (size > 2) {  
            q.offer(q.poll());  
            size--;  
        }  
        // 记录新的队尾元素  
        top_elem = q.peek();  
        q.offer(q.poll());  
    }  
}
```

```
// 删除之前的队尾元素
return q.poll();
}
}
```

这样就实现完了，完整的代码如下：

```
class MyStack {
    Queue<Integer> q = new LinkedList<>();
    int top_elem = 0;

    // 添加元素到栈顶
    public void push(int x) {
        q.offer(x);
        top_elem = x;
    }

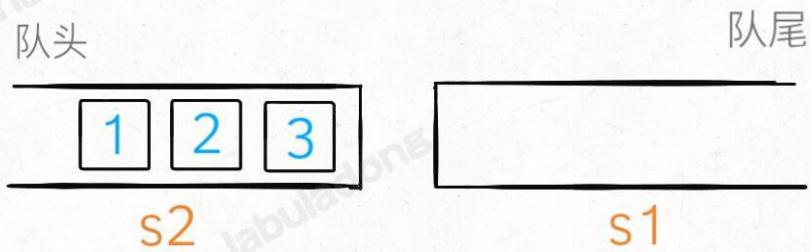
    // 删除栈顶的元素并返回
    public int pop() {
        int size = q.size();
        while (size > 2) {
            q.offer(q.poll());
            size--;
        }
        top_elem = q.peek();
        q.offer(q.poll());
        return q.poll();
    }

    // 返回栈顶元素
    public int top() {
        return top_elem;
    }

    // 判断栈是否为空
    public boolean empty() {
        return q.isEmpty();
    }
}
```

很明显，用队列实现栈的话，`pop` 操作时间复杂度是  $O(N)$ ，其他操作都是  $O(1)$ 。

个人认为，用队列实现栈是没啥亮点的问题，但是用双栈实现队列是值得学习的。



双栈实现的队列

© labuladong

从栈 `s1` 搬运元素到 `s2` 之后，元素在 `s2` 中就变成了队列的先进先出顺序，这个特性有点类似「负负得正」，确实不太容易想到。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer 09. 用两个栈实现队列</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】栈的经典习题

阅读本文前，你需要先学习：

- 队列/栈的原理

### 考察先进后出性质

对于栈这种数据结构的考察，主要考察先进后出特点的运用，比如表达式运算、括号合法性检测等问题，下面列出几个使用栈的经典场景。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】括号类问题汇总

阅读本文前，你需要先学习：

- 队列/栈的原理

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】队列的经典习题

阅读本文前，你需要先学习：

- 队列/栈的原理

### 考察先进先出性质

队列常见考点主要是元素「先进先出」的顺序特性，比如维护队列内的元素在「时序上」的某些性质，下面是几道例题，队列充当「滑动窗口」的作用。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 单调栈算法模板解决三道例题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">739. Daily Temperatures</a>	<a href="#">739. 每日温度</a>	●
<a href="#">503. Next Greater Element II</a>	<a href="#">503. 下一个更大元素 II</a>	●
<a href="#">496. Next Greater Element I</a>	<a href="#">496. 下一个更大元素 I</a>	●
-	<a href="#">剑指 Offer II 038. 每日温度</a>	●

阅读本文前，你需要先学习：

- [数组基础](#)
- [链表基础](#)
- [队列/栈基础](#)

栈（stack）是很简单的一种数据结构，先进后出的逻辑顺序，符合某些问题的特点，比如说函数调用栈。单调栈实际上就是栈，只是利用了一些巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持有序（单调递增或单调递减）。

听起来有点像堆（heap）？不是的，单调栈用途不太广泛，只处理一类典型的问题，比如「下一个更大元素」，「上一个更小元素」等。本文用讲解单调队列的算法模版解决「下一个更大元素」相关问题，并且探讨处理「循环数组」的策略。至于其他的变体和经典例题，我会在下一篇文章 [单调栈变体和经典习题](#) 讲解。

## 单调栈模板

现在给你出这么一道题：输入一个数组 `nums`，请你返回一个等长的结果数组，结果数组中对应索引存储着下一个更大元素，如果没有更大的元素，就存 -1。函数签名如下：

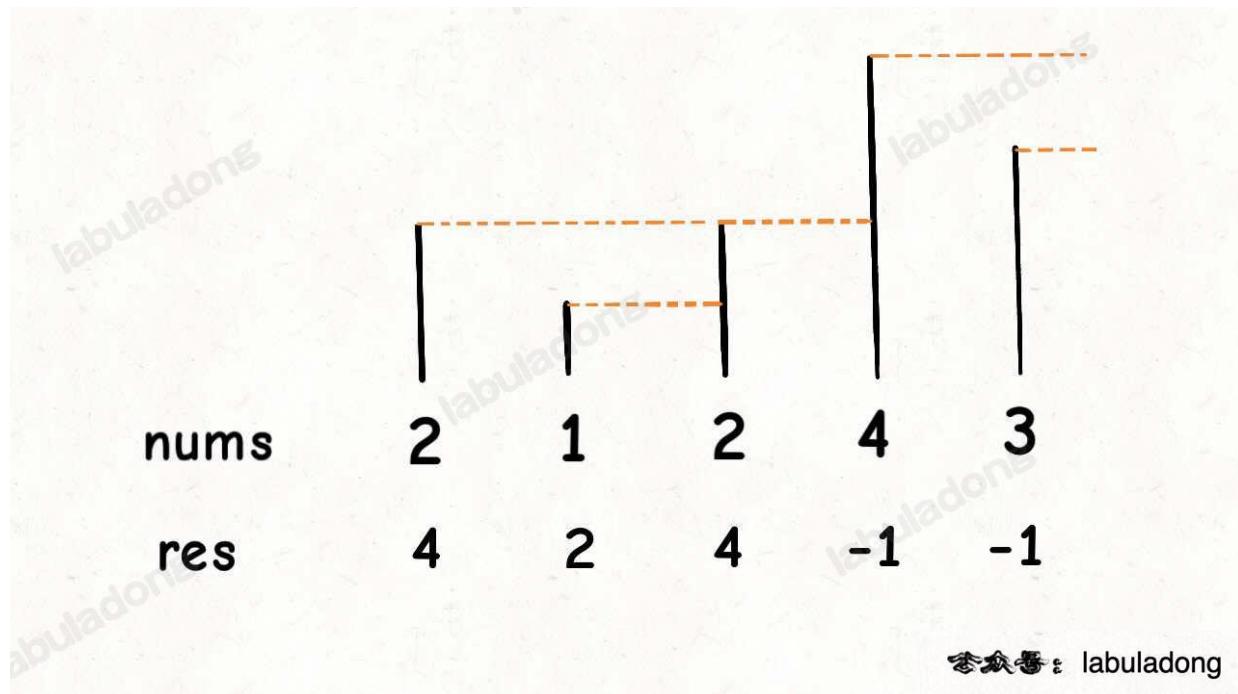
```
int[] calculateGreaterElement(int[] nums);
```

比如说，输入一个数组 `nums = [2, 1, 2, 4, 3]`，你返回数组 `[4, 2, 4, -1, -1]`。因为第一个 2 后面比 2 大的数是 4；1 后面比 1 大的数是 2；第二个 2 后面比 2 大的数是 4；4 后面没有比 4 大的数，填 -1；3 后面没有比 3 大的数，填 -1。

这道题的暴力解法很好想到，就是对每个元素后面都进行扫描，找到第一个更大的元素就行了。但是暴力解法的时间复杂度是  $O(n^2)$ 。

这个问题可以这样抽象思考：把数组的元素想象成并列站立的人，元素大小想象成人的身高。这些人面对你站成一列，如何求元素「2」的下一个更大元素呢？很简单，如果能够看到元素「2」，那么他后面可见的第一个人就是「2」的下一个更

大元素，因为比「2」小的元素身高不够，都被「2」挡住了，第一个露出来的就是答案。



这个情景很好理解吧？带着这个抽象的情景，先来看下代码。

```
int[] calculateGreaterElement(int[] nums) {
    int n = nums.length;
    // 存放答案的数组
    int[] res = new int[n];
    Stack<Integer> s = new Stack<>();
    // 倒着往栈里放
    for (int i = n - 1; i >= 0; i--) {
        // 判定个子高矮
        while (!s.isEmpty() && s.peek() <= nums[i]) {
            // 矮个起开，反正也被挡着了。。。
            s.pop();
        }
        // nums[i] 身后的更大元素
        res[i] = s.isEmpty() ? -1 : s.peek();
        s.push(nums[i]);
    }
    return res;
}
```

这就是单调队列解决问题的模板。for 循环要从后往前扫描元素，因为我们借助的是栈的结构，倒着入栈，其实是正着出栈。while 循环是把两个「个子高」元素之间的元素排除，因为他们的存在没有意义，前面挡着个「更高」的元素，所以他们不可能被作为后续进来的元素的下一个更大元素了。

这个算法的时间复杂度不是那么直观，如果你看到 for 循环嵌套 while 循环，可能认为这个算法的复杂度也是  $O(n^2)$ ，但是实际上这个算法的复杂度只有  $O(n)$ 。

分析它的时间复杂度，要从整体来看：总共有  $n$  个元素，每个元素都被 `push` 入栈了一次，而最多会被 `pop` 一次，没有任何冗余操作。所以总的计算规模是和元素规模  $n$  成正比的，也就是  $O(n)$  的复杂度。

## 问题变形

单调栈的代码实现比较简单，下面来看一些具体题目。

## 496. 下一个更大元素 I

首先来一个简单的变形，力扣第 496 题「下一个更大元素 I」：

▼ 496. 下一个更大元素 I [Leetcode](#) | 力扣

`nums1` 中数字 `x` 的 **下一个更大元素** 是指 `x` 在 `nums2` 中对应位置 **右侧** 的 **第一个** 比 `x` 大的元素。

给你两个 **没有重复元素** 的数组 `nums1` 和 `nums2`，下标从 **0** 开始计数，其中 `nums1` 是 `nums2` 的子集。

对于每个 `0 <= i < nums1.length`，找出满足 `nums1[i] == nums2[j]` 的下标 `j`，并且在 `nums2` 确定 `nums2[j]` 的 **下一个更大元素**。如果不存在下一个更大元素，那么本次查询的答案是 `-1`。

返回一个长度为 `nums1.length` 的数组 `ans` 作为答案，满足 `ans[i]` 是如上所述的 **下一个更大元素**。

示例 1：

```
输入: nums1 = [4,1,2], nums2 = [1,3,4,2].  
输出: [-1,3,-1]  
解释: nums1 中每个值的下一个更大元素如下所述:  
- 4 , 用加粗斜体标识, nums2 = [1,3,4,2]。不存在下一个更大元素, 所以答案是 -1 。  
- 1 , 用加粗斜体标识, nums2 = [1,3,4,2]。下一个更大元素是 3 。  
- 2 , 用加粗斜体标识, nums2 = [1,3,4,2]。不存在下一个更大元素, 所以答案是 -1 。
```

示例 2：

```
输入: nums1 = [2,4], nums2 = [1,2,3,4].  
输出: [3,-1]  
解释: nums1 中每个值的下一个更大元素如下所述:  
- 2 , 用加粗斜体标识, nums2 = [1,2,3,4]。下一个更大元素是 3 。  
- 4 , 用加粗斜体标识, nums2 = [1,2,3,4]。不存在下一个更大元素, 所以答案是 -1 。
```

提示：

- `1 <= nums1.length <= nums2.length <= 1000`
- `0 <= nums1[i], nums2[i] <= 104`
- `nums1` 和 `nums2` 中所有整数 **互不相同**
- `nums1` 中的所有整数同样出现在 `nums2` 中

进阶：你可以设计一个时间复杂度为 `O(nums1.length + nums2.length)` 的解决方案吗？

这道题给你输入两个数组 `nums1` 和 `nums2`，让你求 `nums1` 中的元素在 `nums2` 中的下一个更大元素，函数签名如下：

```
int[] nextGreaterElement(int[] nums1, int[] nums2)
```

其实和把我们刚才的代码改一改就可以解决这道题了，因为题目说 `nums1` 是 `nums2` 的子集，那么我们先把 `nums2` 中每个元素的下一个更大元素算出来存到一个映射里，然后再让 `nums1` 中的元素去查表即可：

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        // 记录 nums2 中每个元素的下一个更大元素
        int[] greater = calculateGreaterElement(nums2);
        // 转化成映射: 元素 x -> x 的下一个最大元素
        HashMap<Integer, Integer> greaterMap = new HashMap<>();
        for (int i = 0; i < nums2.length; i++) {
            greaterMap.put(nums2[i], greater[i]);
        }
        // nums1 是 nums2 的子集, 所以根据 greaterMap 可以得到结果
        int[] res = new int[nums1.length];
        for (int i = 0; i < nums1.length; i++) {
            res[i] = greaterMap.get(nums1[i]);
        }
        return res;
    }

    int[] calculateGreaterElement(int[] nums) {
        // 见上文
    }
}
```



## 739. 每日温度

再看看力扣第 739 题「每日温度」：

给你一个数组 `temperatures`, 这个数组存放的是近几天的天气气温, 你返回一个等长的数组, 计算: 对于每一天, 你还要至少等多少天才能等到一个更暖和的气温; 如果等不到那一天, 填 0。函数签名如下:

```
int[] dailyTemperatures(int[] temperatures);
```

比如说给你输入 `temperatures = [73, 74, 75, 71, 69, 76]`, 你返回 `[1, 1, 3, 2, 1, 0]`。因为第一天 73 华氏度, 第二天 74 华氏度, 比 73 大, 所以对于第一天, 只要等一天就能等到一个更暖和的气温, 后面的同理。

这个问题本质上也是找下一个更大元素, 只不过现在不是问你下一个更大元素的值是多少, 而是问你当前元素距离下一个更大元素的索引距离而已。

相同的思路, 直接调用单调栈的算法模板, 稍作改动就可以, 直接上代码吧:

```
class Solution {
    public int[] dailyTemperatures(int[] temperatures) {
        int n = temperatures.length;
        int[] res = new int[n];
        // 这里放元素索引, 而不是元素
        Stack<Integer> s = new Stack<>();
        // 单调栈模板
        for (int i = n - 1; i >= 0; i--) {
            while (!s.isEmpty() && temperatures[s.peek()] <= temperatures[i]) {
                s.pop();
            }
        }
    }
}
```

```
// 得到索引间距
res[i] = s.isEmpty() ? 0 : (s.peek() - i);
// 将索引入栈，而不是元素
s.push(i);
}
return res;
}
```

单调栈讲解完毕，下面开始另一个重点：如何处理「循环数组」。

## 如何处理环形数组

同样是求下一个更大元素，现在假设给你的数组是个环形的，如何处理？力扣第 503 题「下一个更大元素 II」就是这个问题：输入一个「环形数组」，请你计算其中每个元素的下一个更大元素。

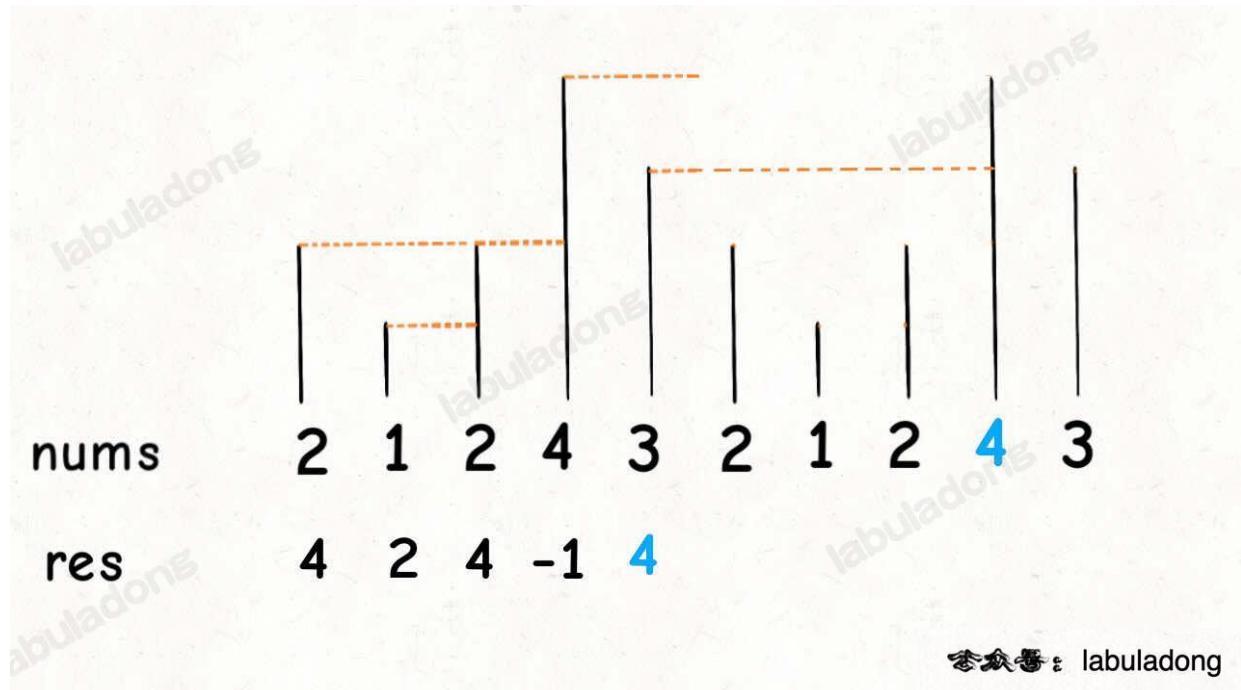
比如输入 `[2,1,2,4,3]`，你应该返回 `[4,2,4,-1,4]`，因为拥有了环形属性，最后一个元素 3 绕了一圈后找到了比自己大的元素 4。

如果你看过基础知识章节的 [环形数组技巧](#) 应该比较熟悉，我们一般是通过 % 运算符求模（余数），来模拟环形特效：

```
int[] arr = {1,2,3,4,5};
int n = arr.length, index = 0;
while (true) {
    // 在环形数组中转圈
    print(arr[index % n]);
    index++;
}
```

这个问题肯定还是要用单调栈的解题模板，但难点在于，比如输入是 `[2,1,2,4,3]`，对于最后一个元素 3，如何找到元素 4 作为下一个更大元素。

对于这种需求，常用套路就是将数组长度翻倍：



这样，元素 3 就可以找到元素 4 作为下一个更大元素了，而且其他的元素都可以被正确地计算。

有了思路，最简单的实现方式当然可以把这个双倍长度的数组构造出来，然后套用算法模板。但是，我们可以不用构造新数组，而是利用循环数组的技巧来模拟数组长度翻倍的效果。直接看代码吧：

```
class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] res = new int[n];
        Stack<Integer> s = new Stack<>();
        // 数组长度加倍模拟环形数组
        for (int i = 2 * n - 1; i >= 0; i--) {
            // 索引 i 要求模，其他的和模板一样
            while (!s.isEmpty() && s.peek() <= nums[i % n]) {
                s.pop();
            }
            res[i % n] = s.isEmpty() ? -1 : s.peek();
            s.push(nums[i % n]);
        }
        return res;
    }
}
```

## ▶ 代码可视化动画

这样，就可以巧妙解决环形数组的问题，时间复杂度  $O(N)$ 。

最后提出一些问题吧，本文提供的单调栈模板是 `nextGreaterElement` 函数，可以计算每个元素的下一个更大元素，但如果题目让你计算上一个更大元素，或者计算上一个更大或相等的元素，应该如何修改对应的模板呢？而且在实际应用中，题目不会直接让你计算下一个（上一个）更大（小）的元素，你如何把问题转化成单调栈相关的问题呢？

我会在 [单调栈的几种变体及习题](#) 对比单调栈的几种其他形式，并在给出单调栈的经典例题。更多数据结构设计类题目参见[数据结构设计经典习题](#)。

## ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1019. Next Greater Node In Linked List</a>	<a href="#">1019. 链表中的下一个更大节点</a>	
<a href="#">1944. Number of Visible People in a Queue</a>	<a href="#">1944. 队列中可以看到的人数</a>	
<a href="#">402. Remove K Digits</a>	<a href="#">402. 移掉 K 位数字</a>	
<a href="#">42. Trapping Rain Water</a>	<a href="#">42. 接雨水</a>	
<a href="#">901. Online Stock Span</a>	<a href="#">901. 股票价格跨度</a>	
<a href="#">918. Maximum Sum Circular Subarray</a>	<a href="#">918. 环形子数组的最大和</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】单调栈的几种变体及经典习题

阅读本文前，你需要先学习：

- 队列/栈的原理
- 单调栈模板

本文给出单调栈模板的更多变形，后面讲的一些经典例题可以抽象成下面的这些标准场景，你到时候可以直接复制粘贴代码去解决。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 单调队列结构解决滑动窗口问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">面试题59 - II. 队列的最大值 LCOF</a>	<a href="#">面试题59 - II. 队列的最大值</a>	🟡
<a href="#">239. Sliding Window Maximum</a>	<a href="#">239. 滑动窗口最大值</a>	🔴

阅读本文前，你需要先学习：

- 数组基础
- 链表基础
- 队列/栈基础

前文用 [单调栈解决三道算法问题](#) 介绍了单调栈这种特殊数据结构，本文写一个类似的数据结构「单调队列」。

也许这种数据结构的名字你没听过，其实没啥难的，就是一个「队列」，只是使用了一点巧妙的方法，使得队列中的元素全都是单调递增（或递减）的。

为啥要发明「单调队列」这种结构呢，主要是为了解决下面这个场景：

给你一个数组 `window`，已知其最值为 `A`，如果给 `window` 中添加一个数 `B`，那么比较一下 `A` 和 `B` 就可以立即算出新的最值；但如果要从 `window` 数组中减少一个数，就不能直接得到最值了，因为如果减少的这个数恰好是 `A`，就需要遍历 `window` 中的所有元素重新寻找新的最值。

这个场景很常见，但不用单调队列似乎也可以，比如优先级队列也是一种特殊的队列，专门用来动态寻找最值的，我创建一个大（小）顶堆，不就可以很快拿到最大（小）值了吗？

如果单纯地维护最值的话，优先级队列很专业，队头元素就是最值。但优先级队列无法满足标准队列结构「先进先出」的时间顺序，因为优先级队列底层利用二叉堆对元素进行动态排序，元素的出队顺序是元素的大小顺序，和入队的先后顺序完全没有关系。

所以，现在需要一种新的队列结构，既能够维护队列元素「先进先出」的时间顺序，又能够正确维护队列中所有元素的最值，这就是「单调队列」结构。

「单调队列」这个数据结构主要用来辅助解决滑动窗口相关的问题，前文 [滑动窗口核心框架](#) 把滑动窗口算法作为双指针技巧的一部分进行了讲解，但有些稍微复杂的滑动窗口问题不能只靠两个指针来解决，需要上更先进的数据结构。

比方说，你注意看前文 [滑动窗口核心框架](#) 讲的几道题目，每当窗口扩大 (`right++`) 和窗口缩小 (`left++`) 时，你单凭移出和移入窗口的元素即可决定是否更新答案。

但本文开头说的那个判断一个窗口中最值的例子，你无法单凭移出窗口的那个元素更新窗口的最值，除非重新遍历所有元素，但这样的话时间复杂度就上来了，这是我们不希望看到的。

我们来看看力扣第 239 题「滑动窗口最大值」，就是一道标准的滑动窗口问题：

给你输入一个数组 `nums` 和一个正整数 `k`，有一个大小为 `k` 的窗口在 `nums` 上从左至右滑动，请你输出每次窗口中 `k` 个元素的最大值。

函数签名如下：

```
int[] maxSlidingWindow(int[] nums, int k);
```

比如说力扣给出的一个示例：

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

接下来，我们就借助单调队列结构，用  $O(1)$  时间算出每个滑动窗口中的最大值，使得整个算法在线性时间完成。

## 一、搭建解题框架

在介绍「单调队列」这种数据结构的 API 之前，先来对比一下 [普通的队列](#) 的标准 API 和单调队列实现的 API：

```
// 普通队列的 API
class Queue {
    // enqueue 操作，在队尾加入元素 n
    void push(int n);
    // dequeue 操作，删除队头元素
    void pop();
}

// 单调队列的 API
class MonotonicQueue {
    // 在队尾添加元素 n
    void push(int n);
    // 返回当前队列中的最大值
    int max();
    // 队头元素如果是 n，删除它
    void pop(int n);
}
```

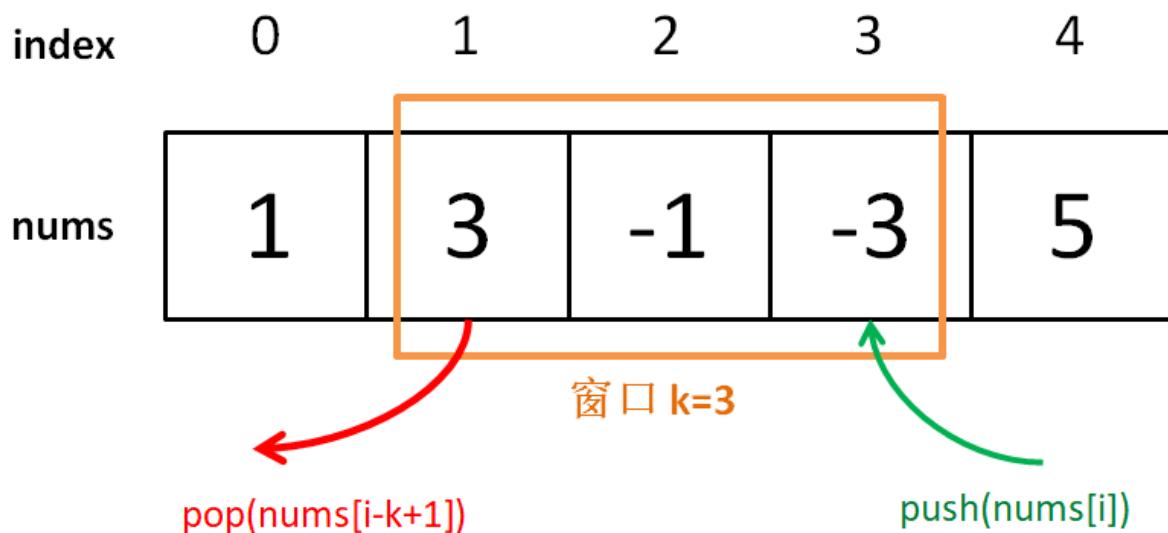
当然，单调队列这几个 API 的实现方法肯定跟一般的 `Queue` 不一样，不过我们暂且不管，而且认为这几个操作的时间复杂度都是  $O(1)$ ，先把这道「滑动窗口」问题的解答框架搭出来：

```

int[] maxSlidingWindow(int[] nums, int k) {
    MonotonicQueue window = new MonotonicQueue();
    List<Integer> res = new ArrayList<>();

    for (int i = 0; i < nums.length; i++) {
        if (i < k - 1) {
            // 先把窗口的前 k - 1 填满
            window.push(nums[i]);
        } else {
            // 窗口开始向前滑动
            // 移入新元素
            window.push(nums[i]);
            // 将当前窗口中的最大元素记入结果
            res.add(window.max());
            // 移出最后的元素
            window.pop(nums[i - k + 1]);
        }
    }
    // 将 List 类型转化成 int[] 数组作为返回值
    int[] arr = new int[res.size()];
    for (int i = 0; i < res.size(); i++) {
        arr[i] = res.get(i);
    }
    return arr;
}

```



这个思路很简单吧，下面我们开始重头戏，单调队列的实现。

## 二、实现单调队列数据结构

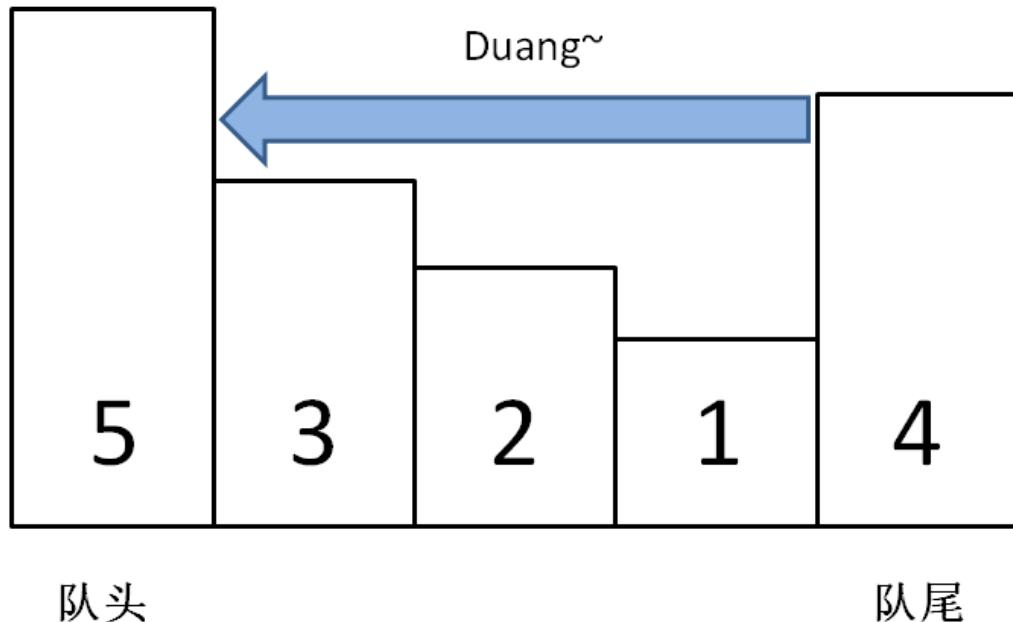
观察滑动窗口的过程就能发现，实现「单调队列」必须使用一种数据结构支持在头部和尾部进行插入和删除，很明显 [双链表](#) 是满足这个条件的。

「单调队列」的核心思路和「单调栈」类似，[push](#) 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉：

```
class MonotonicQueue {
    // 双链表，支持快速在头部和尾部增删元素
    // 维护其中的元素自尾部到头部单调递增
    private LinkedList<Integer> maxq = new LinkedList<>();

    // 在尾部添加一个元素 n，维护 maxq 的单调性质
    public void push(int n) {
        // 将前面小于自己的元素都删除
        while (!maxq.isEmpty() && maxq.getLast() < n) {
            maxq.pollLast();
        }
        maxq.addLast(n);
    }
}
```

你可以想象，加入数字的大小代表人的体重，体重大大的会把前面体重不足的压扁，直到遇到更大的量级才停住。



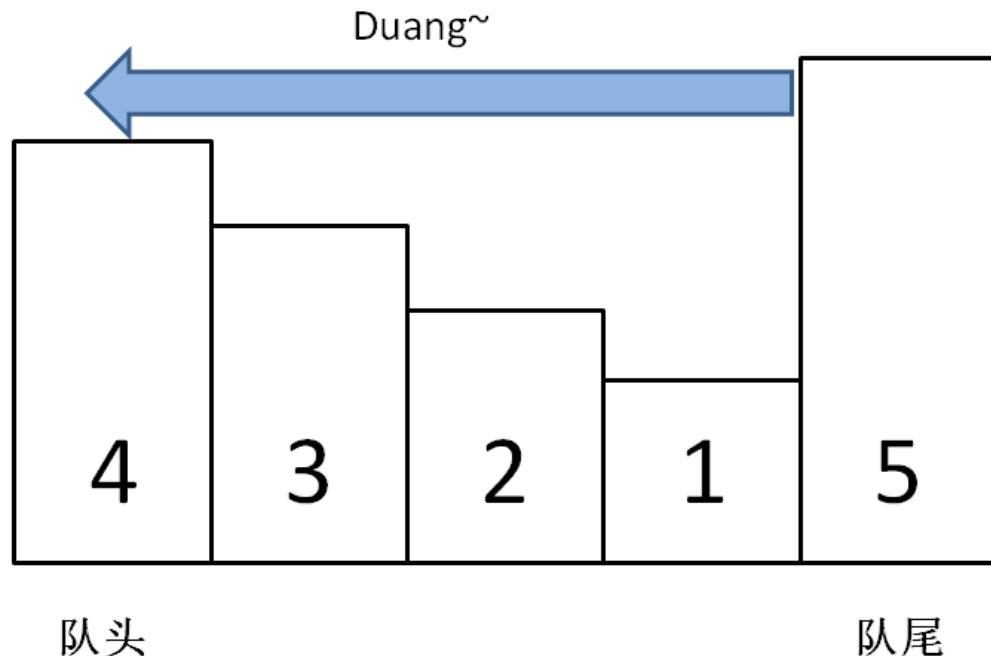
如果每个元素被加入时都这样操作，最终单调队列中的元素大小就会保持一个**单调递减**的顺序，因此我们的 `max` 方法就很好写了，只要把队头元素返回即可；`pop` 方法也是操作队头，如果队头元素是待删除元素 `n`，那么就删除它：

```
class MonotonicQueue {
    // 为了节约篇幅，省略上文给出的代码部分...

    public int max() {
        // 队头的元素肯定是最大的
        return maxq.getFirst();
    }

    public void pop(int n) {
        if (n == maxq.getFirst()) {
            maxq.pollFirst();
        }
    }
}
```

`pop` 方法之所以要判断 `n == maxq.getFirst()`, 是因为我们想删除的队头元素 `n` 可能已经在 `push` 的过程中被「压扁」了, 可能已经不存在了, 这种情况就不用删除了:



至此，单调队列设计完毕，看下完整的解题代码：

```
// 单调队列的实现
class MonotonicQueue {
    LinkedList<Integer> maxq = new LinkedList<>();
    public void push(int n) {
        // 将小于 n 的元素全部删除
        while (!maxq.isEmpty() && maxq.getLast() < n) {
            maxq.pollLast();
        }
        // 然后将 n 加入尾部
        maxq.addLast(n);
    }

    public int max() {
        return maxq.getFirst();
    }

    public void pop(int n) {
        if (n == maxq.getFirst()) {
            maxq.pollFirst();
        }
    }
}

class Solution {
    int[] maxSlidingWindow(int[] nums, int k) {
        MonotonicQueue window = new MonotonicQueue();
        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (i >= k - 1) {
                res.add(window.max());
            }
            if (i >= k - 1) {
                window.pop(nums[i - k + 1]);
            }
            window.push(nums[i]);
        }
        return res.toArray(new int[res.size()]);
    }
}
```

```

    if (i < k - 1) {
        // 先填满窗口的前 k - 1
        window.push(nums[i]);
    } else {
        // 窗口向前滑动，加入新数字
        window.push(nums[i]);
        // 记录当前窗口的最大值
        res.add(window.max());
        // 移出旧数字
        window.pop(nums[i - k + 1]);
    }
}
// 需要转成 int[] 数组再返回
int[] arr = new int[res.size()];
for (int i = 0; i < res.size(); i++) {
    arr[i] = res.get(i);
}
return arr;
}
}

```

有一点细节问题不要忽略，在实现 `MonotonicQueue` 时，我们使用了 Java 的 `LinkedList`，因为链表结构支持在头部和尾部快速增删元素；而在解法代码中的 `res` 则使用的 `ArrayList` 结构，因为后续会按照索引取元素，所以数组结构更合适。其他语言的实现也要注意这些细节。

关于单调队列 API 的时间复杂度，读者可能有疑惑：`push` 操作中含有 `while` 循环，最坏情况下的时间复杂度应该  $O(N)$  呀，再加上一层 `for` 循环，本算法的时间复杂度应该是  $O(N^2)$  才对吧？

这里就用到了 [算法时空复杂度分析指南](#) 中讲到的摊还分析：

单独看 `push` 操作，最坏时间复杂度确实是  $O(N)$ ，但是平均时间复杂度是  $O(1)$ 。我们一般用平均复杂度而不是最坏时间复杂度来衡量 API 接口，所以这个算法整体的时间复杂度是  $O(N)$ ，而不是  $O(N^2)$ 。

也可以这样从整体上分析：整个算法做的事情就是把 `nums` 中的每个元素加入和移出 `window` 至多一次，不可能把同一个元素多次移入移出 `window`，所以整体的时间复杂度是  $O(N)$ 。

空间复杂度很容易分析，就是窗口的大小  $O(k)$ 。

## 拓展延伸

最后，我提出几个问题请大家思考：

1、本文给出的 `MonotonicQueue` 类只实现了 `max` 方法，你是否能够再额外添加一个 `min` 方法，在  $O(1)$  的时间返回队列中所有元素的最小值？

2、本文给出的 `MonotonicQueue` 类的 `pop` 方法还需要接收一个参数，这不那么优雅，而且有悖于标准队列的 API，请你修复这个缺陷。

3、请你实现 `MonotonicQueue` 类的 `size` 方法，返回单调队列中元素的个数（注意，由于每次 `push` 方法都可能从底层的 `q` 列表中删除元素，所以 `q` 中的元素个数并不是单调队列的元素个数）。

也就是说，你是否能够实现单调队列的通用实现：

```

// 单调队列的通用实现，可以高效维护最大值和最小值
class MonotonicQueue<E extends Comparable<E>> {

```

```
// 标准队列 API, 向队尾加入元素  
public void push(E elem);  
  
// 标准队列 API, 从队头弹出元素, 符合先进先出的顺序  
public E pop();  
  
// 标准队列 API, 返回队列中的元素个数  
public int size();  
  
// 单调队列特有 API, O(1) 时间计算队列中元素的最大值  
public E max();  
  
// 单调队列特有 API, O(1) 时间计算队列中元素的最小值  
public E min();  
}
```

我将在 [单调队列通用实现及应用](#) 中给出单调队列的通用实现和经典习题。更多数据结构设计类题目参见 [数据结构设计经典习题](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1425. Constrained Subsequence Sum</a>	<a href="#">1425. 带限制的子序列和</a>	
<a href="#">1696. Jump Game VI</a>	<a href="#">1696. 跳跃游戏 VI</a>	
<a href="#">862. Shortest Subarray with Sum at Least K</a>	<a href="#">862. 和至少为 K 的最短子数组</a>	
<a href="#">918. Maximum Sum Circular Subarray</a>	<a href="#">918. 环形子数组的最大和</a>	
-	<a href="#">剑指 Offer 59 - I. 滑动窗口的最大值</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】单调队列的通用实现及经典习题

阅读本文前，你需要先学习：

- 队列/栈的原理
- 单调队列解决滑动窗口问题

### 通用实现

我先提供一个单调队列结构的通用实现，这里涉及 Java 的泛型，`E` 就代表任意类型，本站开头的 [Java 语言基础](#) 有讲过，这里就不细讲了。

`E extends Comparable<E>` 的意思是这个类型 `E` 需要实现 `Comparable` 接口，即类型 `E` 是可比较的，比如 `Integer`, `String` 这种实现了 `compareTo` 方法的类型。原因也很好理解，因为你要求队列中元素的最值嘛，所以元素当然得是有大小之分（可比较）的。

我们原先的简陋实现包含了 `max` 方法的实现，其原理是在底层维护了一个队列 `maxq`，维护这个队列中从尾部到头部的元素单调递增。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 算法就像搭乐高：手撸 LRU 算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">146. LRU Cache</a>	<a href="#">146. LRU 缓存</a>	★★

-----

阅读本文前，你需要先学习：

- 链表基础
- 链表实现
- 哈希表基础
- 哈希表实现

LRU 算法就是一种缓存淘汰策略，原理不难，但是面试中写出没有 bug 的算法比较有技巧，需要对数据结构进行层层抽象和拆解，本文就带你写一手漂亮的代码。

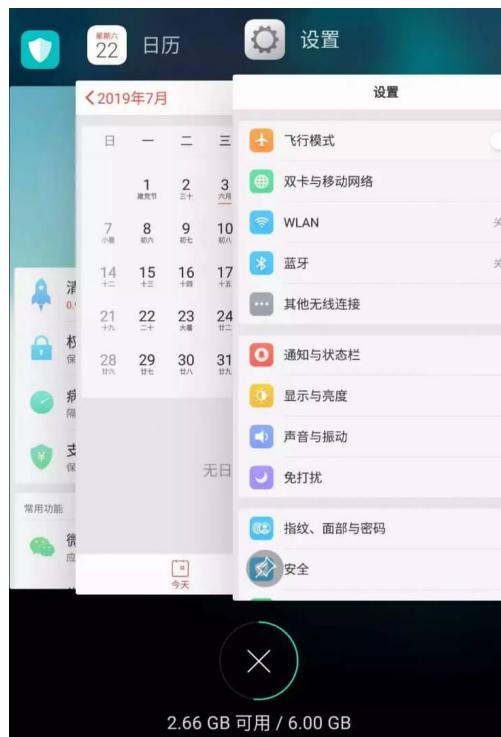
计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉哪些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。

举个简单的例子，安卓手机都可以把软件放到后台运行，比如我先后打开了「设置」「手机管家」「日历」，那么现在它们在后台排列的顺序是这样的：

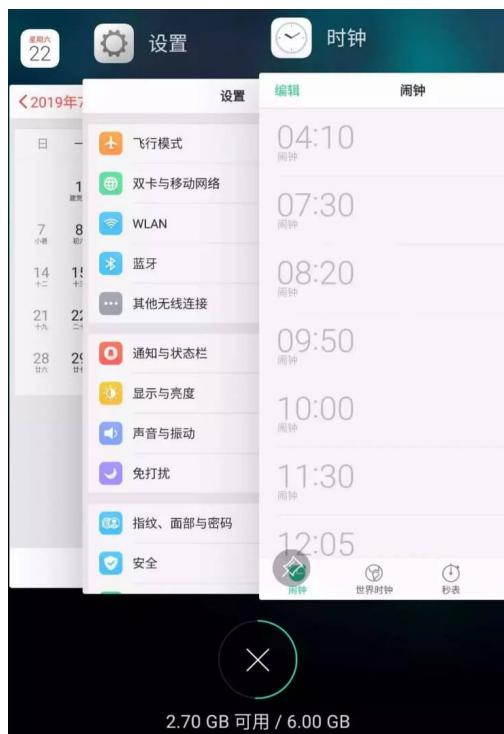


但是这时候如果我访问了一下「设置」界面，那么「设置」就会被提前到第一个，变成这样：



假设我的手机只允许我同时开 3 个应用程序，现在已经满了。那么如果我新开了一个应用「时钟」，就必须关闭一个应用为「时钟」腾出一个位置，关那个呢？

按照 LRU 的策略，就关最底下的「手机管家」，因为那是最久未使用的，然后把新开的应用放到最上面：



现在你应该理解 LRU (Least Recently Used) 策略了。当然还有其他缓存淘汰策略，比如不要按访问的时序来淘汰，而是按访问频率 (LFU 策略) 来淘汰等等，各有应用场景。本文讲解 LRU 算法策略，我会在 [LFU 算法详解](#) 中讲解 LFU 算法。

## 一、LRU 算法描述

力扣第 146 题「LRU缓存机制」就是让你设计数据结构：

首先要接收一个 `capacity` 参数作为缓存的最大容量，然后实现两个 API，一个是 `put(key, val)` 方法存入键值对，另一个是 `get(key)` 方法获取 `key` 对应的 `val`，如果 `key` 不存在则返回 -1。

注意哦，`get` 和 `put` 方法必须都是  $O(1)$  的时间复杂度，我们举个具体例子来看看 LRU 算法怎么工作。

```
// 缓存容量为 2
LRUCache cache = new LRUCache(2);
// 你可以把 cache 理解成一个队列
// 假设左边是队头，右边是队尾
// 最近使用的排在队头，久未使用的排在队尾
// 圆括号表示键值对 (key, val)

cache.put(1, 1);
// cache = [(1, 1)]

cache.put(2, 2);
// cache = [(2, 2), (1, 1)]

// 返回 1
cache.get(1);
// cache = [(1, 1), (2, 2)]
// 解释：因为最近访问了键 1，所以提前至队头
// 返回键 1 对应的值 1

cache.put(3, 3);
// cache = [(3, 3), (1, 1)]
// 解释：缓存容量已满，需要删除内容空出位置
// 优先删除久未使用的数据，也就是队尾的数据
```

```
// 然后把新的数据插入队头

// 返回 -1 (未找到)
cache.get(2);
// cache = [(3, 3), (1, 1)]
// 解释: cache 中不存在键为 2 的数据

cache.put(1, 4);
// cache = [(1, 4), (3, 3)]
// 解释: 键 1 已存在, 把原始值 1 覆盖为 4
// 不要忘了也要将键值对提前到队头
```

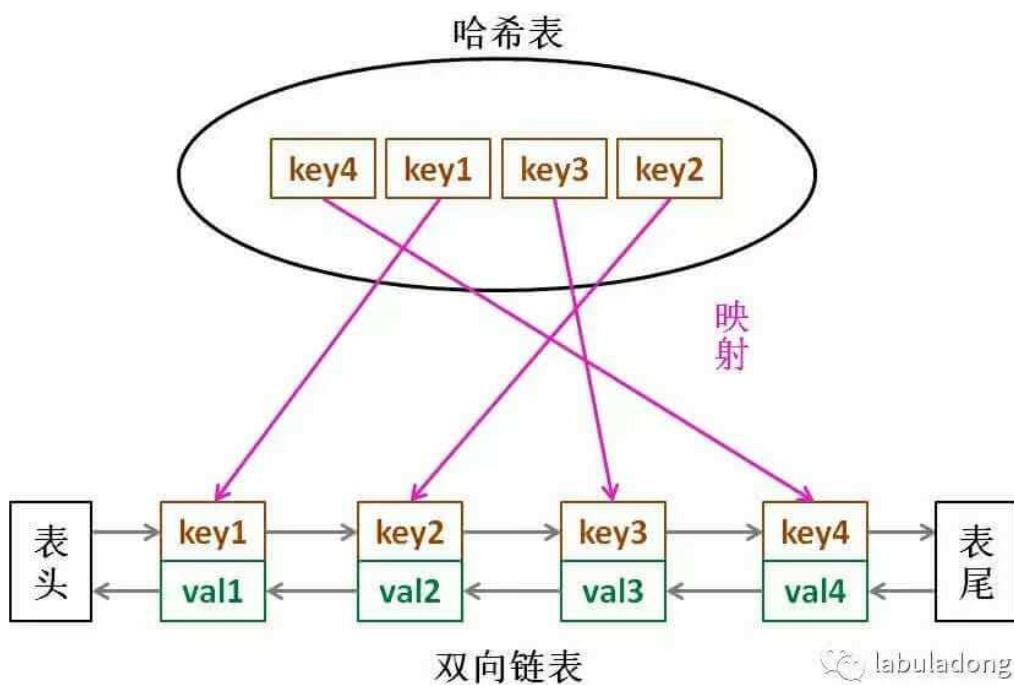
## 二、LRU 算法设计

分析上面的操作过程，要让 `put` 和 `get` 方法的时间复杂度为  $O(1)$ ，我们可以总结出 `cache` 这个数据结构必要的条件：

- 1、显然 `cache` 中的元素必须有时序，以区分最近使用的和久未使用的数据，当容量满了之后要删除最久未使用的那个元素腾位置。
- 2、我们要在 `cache` 中快速找某个 `key` 是否已存在并得到对应的 `val`；
- 3、每次访问 `cache` 中的某个 `key`，需要将这个元素变为最近使用的，也就是说 `cache` 要支持在任意位置快速插入和删除元素。

那么，什么数据结构同时符合上述条件呢？哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：哈希链表 `LinkedHashMap`。

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：



借助这个结构，我们来逐一分析上面的 3 个条件：

- 1、如果我们每次默认从链表尾部添加元素，那么显然越靠尾部的元素就是最近使用的，越靠头部的元素就是最久未使用的。
- 2、对于某一个 `key`，我们可以通过哈希表快速定位到链表中的节点，从而取得对应 `val`。

3、链表显然是支持在任意位置快速插入和删除的，改改指针就行。只不过传统的链表无法按照索引快速访问某一个位置的元素，而这里借助哈希表，可以通过 **key** 快速映射到任意一个链表节点，然后进行插入和删除。

也许读者会问，为什么要是双向链表，单链表行不行？另外，既然哈希表中已经存了 **key**，为什么链表中还要存 **key** 和 **val** 呢，只存 **val** 不就行了？

想的时候都是问题，只有做的时候才有答案。这样设计的原因，必须等我们亲自实现 LRU 算法之后才能理解，所以我们开始看代码吧～

### 三、代码实现

很多编程语言都有内置的哈希链表或者类似 LRU 功能的库函数，但是为了帮大家理解算法的细节，我们先自己造轮子实现一遍 LRU 算法，然后再使用 Java 内置的 **LinkedHashMap** 来实现一遍。

首先，我们把 **双链表** 的节点类写出来，为了简化，**key** 和 **val** 都认为是 int 类型：

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

然后依靠我们的 **Node** 类型构建一个双链表，实现几个 LRU 算法必须的 API：

```
class DoubleList {  
    // 头尾虚节点  
    private Node head, tail;  
    // 链表元素数  
    private int size;  
  
    public DoubleList() {  
        // 初始化双向链表的数据  
        head = new Node(0, 0);  
        tail = new Node(0, 0);  
        head.next = tail;  
        tail.prev = head;  
        size = 0;  
    }  
  
    // 在链表尾部添加节点 x, 时间 O(1)  
    public void addLast(Node x) {  
        x.prev = tail.prev;  
        x.next = tail;  
        tail.prev.next = x;  
        tail.prev = x;  
        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.prev.next = x.next;  
    }  
}
```

```

        x.next.prev = x.prev;
        size--;
    }

    // 删除链表中第一个节点，并返回该节点，时间 O(1)
    public Node removeFirst() {
        if (head.next == tail)
            return null;
        Node first = head.next;
        remove(first);
        return first;
    }

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

}

```

如果对链表的操作不熟悉，可以看前文 [手把手带你实现双链表](#)。

到这里就能回答刚才「为什么必须要用双向链表」的问题了，因为我们需要删除操作。删除一个节点不光要得到该节点本身的指针，也需要操作其前驱节点的指针，而双向链表才能支持直接查找前驱，保证操作的时间复杂度 O(1)。

注意我们实现的双链表 API 只能从尾部插入，也就是说靠尾部的数据是最近使用的，靠头部的数据是最久未使用的。

有了双向链表的实现，我们只需要在 LRU 算法中把它和哈希表结合起来即可，先搭出代码框架：

```

class LRUCache {
    // key -> Node(key, val)
    private HashMap<Integer, Node> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    private DoubleList cache;
    // 最大容量
    private int cap;

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

```

先不慌去实现 LRU 算法的 `get` 和 `put` 方法。由于我们要同时维护一个双链表 `cache` 和一个哈希表 `map`，很容易漏掉一些操作，比如说删除某个 `key` 时，在 `cache` 中删除了对应的 `Node`，但是却忘记在 `map` 中删除 `key`。

解决这种问题的有效方法是：在这两种数据结构之上提供一层抽象 API。

说的有点玄幻，实际上很简单，就是尽量让 LRU 的主方法 `get` 和 `put` 避免直接操作 `map` 和 `cache` 的细节。我们可以先实现下面几个函数：

```

class LRUCache {
    // 为了节约篇幅，省略上文给出的代码部分...
}

```

```

// 将某个 key 提升为最近使用的
private void makeRecently(int key) {
    Node x = map.get(key);
    // 先从链表中删除这个节点
    cache.remove(x);
    // 重新插到队尾
    cache.addLast(x);
}

// 添加最近使用的元素
private void addRecently(int key, int val) {
    Node x = new Node(key, val);
    // 链表尾部就是最近使用的元素
    cache.addLast(x);
    // 别忘了在 map 中添加 key 的映射
    map.put(key, x);
}

// 删除某一个 key
private void deleteKey(int key) {
    Node x = map.get(key);
    // 从链表中删除
    cache.remove(x);
    // 从 map 中删除
    map.remove(key);
}

// 删除最久未使用的元素
private void removeLeastRecently() {
    // 链表头部的第一个元素就是最久未使用的
    Node deletedNode = cache.removeFirst();
    // 同时别忘了从 map 中删除它的 key
    int deletedKey = deletedNode.key;
    map.remove(deletedKey);
}
}

```

这里就能回答之前的问答题「为什么要在链表中同时存储 key 和 val，而不是只存储 val」，注意 `removeLeastRecently` 函数中，我们需要用 `deletedNode` 得到 `deletedKey`。

也就是说，当缓存容量已满，我们不仅仅要删除最后一个 `Node` 节点，还要把 `map` 中映射到该节点的 `key` 同时删除，而这个 `key` 只能由 `Node` 得到。如果 `Node` 结构中只存储 `val`，那么我们就无法得知 `key` 是什么，就无法删除 `map` 中的键，造成错误。

上述方法就是简单的操作封装，调用这些函数可以避免直接操作 `cache` 链表和 `map` 哈希表，下面我先来实现 LRU 算法的 `get` 方法：

```

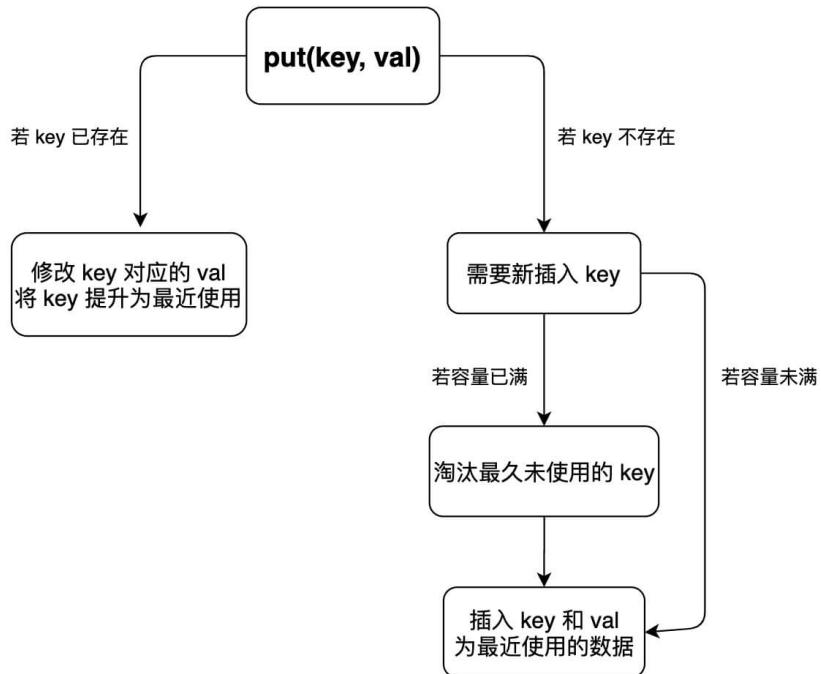
class LRUCache {
    // 为了节约篇幅，省略上文给出的代码部分...

    public int get(int key) {
        if (!map.containsKey(key)) {
            return -1;
        }
        // 将该数据提升为最近使用的
    }
}

```

```
        makeRecently(key);
        return map.get(key).val;
    }
}
```

**put** 方法稍微复杂一些，我们先来画个图搞清楚它的逻辑：



这样我们可以轻松写出 **put** 方法的代码：

```
class LRUCache {
    // 为了节约篇幅，省略上文给出的代码部分...

    public void put(int key, int val) {
        if (map.containsKey(key)) {
            // 删除旧的数据
            deleteKey(key);
            // 新插入的数据为最近使用的数据
            addRecently(key, val);
            return;
        }

        if (cap == cache.size()) {
            // 删除最久未使用的元素
            removeLeastRecently();
        }
        // 添加为最近使用的元素
        addRecently(key, val);
    }
}
```

至此，你应该已经完全掌握 LRU 算法的原理和实现了。看下完整的实现：

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

// 双向链表
class DoubleList {
    // 头尾虚节点
    private Node head, tail;
    // 链表元素数
    private int size;

    public DoubleList() {
        // 初始化双向链表的数据
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
        size = 0;
    }

    // 在链表尾部添加节点 x, 时间 O(1)
    public void addLast(Node x) {
        x.prev = tail.prev;
        x.next = tail;
        tail.prev.next = x;
        tail.prev = x;
        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.prev.next = x.next;
        x.next.prev = x.prev;
        size--;
    }

    // 删除链表中第一个节点, 并返回该节点, 时间 O(1)
    public Node removeFirst() {
        if (head.next == tail)
            return null;
        Node first = head.next;
        remove(first);
        return first;
    }

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

```

class LRUCache {
    // key -> Node(key, val)
    private HashMap<Integer, Node> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    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;
        }
        // 将该数据提升为最近使用的
        makeRecently(key);
        return map.get(key).val;
    }

    public void put(int key, int val) {
        if (map.containsKey(key)) {
            // 删除旧的数据
            deleteKey(key);
            // 新插入的数据为最近使用的数据
            addRecently(key, val);
            return;
        }

        if (cap == cache.size()) {
            // 删除最久未使用的元素
            removeLeastRecently();
        }
        // 添加为最近使用的元素
        addRecently(key, val);
    }

    private void makeRecently(int key) {
        Node x = map.get(key);
        // 先从链表中删除这个节点
        cache.remove(x);
        // 重新插到队尾
        cache.addLast(x);
    }

    private void addRecently(int key, int val) {
        Node x = new Node(key, val);
        // 链表尾部就是最近使用的元素
        cache.addLast(x);
        // 别忘了在 map 中添加 key 的映射
        map.put(key, x);
    }

    private void deleteKey(int key) {
        Node x = map.get(key);
        // 从链表中删除
    }
}

```

```
        cache.remove(x);
        // 从 map 中删除
        map.remove(key);
    }

    private void removeLeastRecently() {
        // 链表头部的第一个元素就是最久未使用的
        Node deletedNode = cache.removeFirst();
        // 同时别忘了从 map 中删除它的 key
        int deletedKey = deletedNode.key;
        map.remove(deletedKey);
    }
}
```

我们最后用 Java 的内置类型 `LinkedHashMap` 来实现 LRU 算法，逻辑和之前完全一致：

```
class LRUCache {
    int cap;
    LinkedHashMap<Integer, Integer> cache = new LinkedHashMap<>();
    public LRUCache(int capacity) {
        this.cap = capacity;
    }

    public int get(int key) {
        if (!cache.containsKey(key)) {
            return -1;
        }
        // 将 key 变为最近使用
        makeRecently(key);
        return cache.get(key);
    }

    public void put(int key, int val) {
        if (cache.containsKey(key)) {
            // 修改 key 的值
            cache.put(key, val);
            // 将 key 变为最近使用
            makeRecently(key);
            return;
        }

        if (cache.size() >= this.cap) {
            // 链表头部就是最久未使用的 key
            int oldestKey = cache.keySet().iterator().next();
            cache.remove(oldestKey);
        }
        // 将新的 key 添加链表尾部
        cache.put(key, val);
    }

    private void makeRecently(int key) {
        int val = cache.get(key);
        // 删除 key, 重新插入到队尾
        cache.remove(key);
        cache.put(key, val);
    }
}
```

对于其他语言，可以自行搜索一下是否有 [LinkedHashMap](#) 这种数据结构。一般面试时也不用从头手写双链表，直接使用内置的数据结构实现 LRU 算法即可。

至此，LRU 算法就没有什么神秘的了。更多数据结构设计相关的题目参见 [数据结构设计经典习题](#)。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer II 031. 最近最少使用缓存</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 算法就像搭乐高：手撸 LFU 算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
460. LFU Cache	460. LFU 缓存	●

-----

阅读本文前，你需要先学习：

- 哈希表基础
- 哈希表实现

上篇文章 [带你手写 LRU 算法](#) 写了 LRU 缓存淘汰算法的实现方法，本文来写另一个著名的缓存淘汰算法：LFU 算法。

LRU 算法的淘汰策略是 Least Recently Used，也就是每次淘汰那些最久没被使用的数据；而 LFU 算法的淘汰策略是 Least Frequently Used，也就是每次淘汰那些使用次数最少的数据。

LRU 算法的核心数据结构是使用哈希链表 [LinkedHashMap](#)，首先借助链表的有序性使得链表元素维持插入顺序，同时借助哈希映射的快速访问能力使得我们可以在  $O(1)$  时间访问链表的任意元素。

从实现难度上来说，LFU 算法的难度大于 LRU 算法，因为 LRU 算法相当于把数据按照时间排序，这个需求借助链表很自然就能实现，你一直从链表头部加入元素的话，越靠近头部的元素就是新的数据，越靠近尾部的元素就是旧的数据，我们进行缓存淘汰的时候只要简单地将尾部的元素淘汰掉就行了。

而 LFU 算法相当于是把数据按照访问频次进行排序，这个需求恐怕没有那么简单，而且还有一种情况，如果多个数据拥有相同的访问频次，我们就得删除最早插入的那个数据。也就是说 LFU 算法是淘汰访问频次最低的数据，如果访问频次最低的数据有多条，需要淘汰最旧的数据。

所以说 LFU 算法是要复杂很多的，而且经常出现在面试中，因为 LFU 缓存淘汰算法在工程实践中经常使用，也有可能是因为 LRU 算法太简单了。不过话说回来，这种著名的算法的套路都是固定的，关键是由于逻辑较复杂，不容易写出漂亮且没有 bug 的代码。

那么本文我就带你拆解 LFU 算法，自顶向下，逐步求精，就是解决复杂问题的不二法门。

## 一、算法描述

要求你写一个类，接受一个 `capacity` 参数，实现 `get` 和 `put` 方法：

```
class LFUCache {  
    // 构造容量为 capacity 的缓存  
    public LFUCache(int capacity) {}  
    // 在缓存中查询 key
```

```
public int get(int key) {}  
// 将 key 和 val 存入缓存  
public void put(int key, int val) {}  
}
```

`get(key)` 方法会去缓存中查询键 `key`, 如果 `key` 存在, 则返回 `key` 对应的 `val`, 否则返回 -1。

`put(key, value)` 方法插入或修改缓存。如果 `key` 已存在, 则将它对应的值改为 `val`; 如果 `key` 不存在, 则插入键值对 `(key, val)`。

当缓存达到容量 `capacity` 时, 则应该在插入新的键值对之前, 删除使用频次 (后文用 `freq` 表示) 最低的键值对。如果 `freq` 最低的键值对有多个, 则删除其中最旧的那个。

```
// 构造一个容量为 2 的 LFU 缓存  
LFUCache cache = new LFUCache(2);  
  
// 插入两对 (key, val), 对应的 freq 为 1  
cache.put(1, 10);  
cache.put(2, 20);  
  
// 查询 key 为 1 对应的 val  
// 返回 10, 同时键 1 对应的 freq 变为 2  
cache.get(1);  
  
// 容量已满, 淘汰 freq 最小的键 2  
// 插入键值对 (3, 30), 对应的 freq 为 1  
cache.put(3, 30);  
  
// 键 2 已经被淘汰删除, 返回 -1  
cache.get(2);
```

## 二、思路分析

一定先从最简单的开始, 根据 LFU 算法的逻辑, 我们先列举出算法执行过程中的几个显而易见的事实:

- 1、调用 `get(key)` 方法时, 要返回该 `key` 对应的 `val`。
- 2、只要用 `get` 或者 `put` 方法访问一次某个 `key`, 该 `key` 的 `freq` 就要加一。
- 3、如果在容量满了的时候进行插入, 则需要将 `freq` 最小的 `key` 删除, 如果最小的 `freq` 对应多个 `key`, 则删除其中最旧的那个。

好的, 我们希望能够在  $O(1)$  的时间内解决这些需求, 可以使用基本数据结构来逐个击破:

- 1、使用一个 `HashMap` 存储 `key` 到 `val` 的映射, 就可以快速计算 `get(key)`。

```
HashMap<Integer, Integer> keyToVal;
```

- 2、使用一个 `HashMap` 存储 `key` 到 `freq` 的映射, 就可以快速操作 `key` 对应的 `freq`。

```
HashMap<Integer, Integer> keyToFreq;
```

3、这个需求应该是 LFU 算法的核心，所以我们分开说：

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 常数时间删除/查找数组中的任意元素



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">710. Random Pick with Blacklist</a>	<a href="#">710. 黑名单中的随机数</a>	
<a href="#">380. Insert Delete GetRandom O(1)</a>	<a href="#">380. O(1) 时间插入、删除和获取随机元素</a>	

阅读本文前，你需要先学习：

- 数组基础
- 数组实现
- 哈希表基础
- 哈希表实现
- 用数组加强哈希表

本文讲两道比较有技巧性的数据结构设计题，都是和随机读取元素相关的，我在前文 [谈谈游戏中的随机算法](#) 也写过类似的问题。

这些问题的一个技巧点在于，如何结合哈希表和数组，使得数组的删除操作时间复杂度也变成  $O(1)$ ？下面来一道道看。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】哈希表更多习题

阅读本文前，你需要先学习：

- 哈希表原理
- 拉链法实现哈希表

哈希表在算法题中是常考的数据结构，主要出现在数据结构的设计题，或者配合数组、字符串等结构进行考察（比如之前的[前缀和习题](#)），这里列出一些常见的哈希表考题。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】优先级队列经典习题

阅读本文前，你需要先学习：

- [二叉堆基础](#)
- [二叉堆实现优先级队列](#)

二叉堆的主要应用是优先级队列，而优先级队列的特色是**动态排序**，插入的元素可以自动维护正确的顺序。当然，[二叉搜索树](#)也可以做到动态排序，但优先级队列提供的接口更简单，实现也更简单。

一般来说，用到优先级队列的题目主要分两类，一类是把多个有序序列合并成一个，另一类是在多个有序序列中寻找第  $k$  个最大元素这类题，我们分别来看。

### 类型一，合并有序序列

先来看第一类，类似于合并有序链表这样的题目。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# TreeMap/TreeSet 代码实现

阅读本文前，你需要先学习：

- [TreeMap/TreeSet 原理](#)

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Trie/字典树/前缀树代码实现

阅读本文前，你需要先学习：

- 二叉树遍历
- 多叉树遍历
- Trie 树原理
- 100 道二叉树习题强化练习

关于 Trie 树的原理，请参见基础知识章节的 [Trie 树原理](#)。本文将直接给出 Trie 树的代码实现，并用 Trie 树解决几道实际的算法题。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】Trie 树算法习题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">208. Implement Trie (Prefix Tree)</a>	<a href="#">208. 实现 Trie (前缀树)</a>	简单
<a href="#">677. Map Sum Pairs</a>	<a href="#">677. 键值映射</a>	简单
-	<a href="#">剑指 Offer II 063. 替换单词</a>	简单
-	<a href="#">剑指 Offer II 066. 单词之和</a>	简单
<a href="#">1804. Implement Trie II (Prefix Tree) </a>	<a href="#">1804. 实现 Trie (前缀树) II </a>	简单
<a href="#">648. Replace Words</a>	<a href="#">648. 单词替换</a>	简单
<a href="#">211. Design Add and Search Words Data Structure</a>	<a href="#">211. 添加与搜索单词 - 数据结构设计</a>	简单
-	<a href="#">剑指 Offer II 062. 实现前缀树</a>	简单

阅读本文前，你需要先学习：

- Trie 树原理
- TrieMap/TrieSet 代码实现

有了 TrieMap 和 TrieSet，力扣上所有前缀树相关的题目都可以直接套用了，下面我举几个题目实践一下。

首先，前文 [TrieMap/TrieSet 代码实现](#) 给出的 TrieMap/TrieSet 执行效率在具体的题目里面肯定是有优化空间的。

比如力扣前缀树相关题目的输入都被限制在小写英文字母 a-z，所以 TrieNode 其实不用维护一个大小为 256 的 children 数组，大小设置为 26 就够了，可以减小时间和空间上的复杂度。

另外，之前给出的 Java/cpp 代码带有泛型，在做算法题的时候其实不需要，去掉泛型也可以获得一定的效率提升。

### 208. 实现 Trie (前缀树)

先看下力扣第 208 题「实现前缀树」：

#### ▼ 208. 实现 Trie (前缀树) [Leetcode](#) | [力扣](#)

Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true`（即，在检索之前已经插入）；否则，返回 `false`。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix`，返回 `true`；否则，返回 `false`。

示例：

输入

```
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
```

输出

```
[null, null, true, false, true, null, true]
```

解释

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");    // 返回 True
trie.search("app");     // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app");     // 返回 True
```

提示：

- `1 <= word.length, prefix.length <= 2000`
- `word` 和 `prefix` 仅由小写英文字母组成
- `insert`、`search` 和 `startsWith` 调用次数 总计 不超过  $3 * 10^4$  次

题目让我们实现的几个函数其实就是 `TrieSet` 的部分 API，所以直接封装一个 `TrieSet` 就能解决这道题了：

```
class Trie {
    // 直接封装 TrieSet
    TrieSet set = new TrieSet();

    // 插入一个元素
    public void insert(String word) {
        set.add(word);
    }

    // 判断元素是否在集合中
    public boolean search(String word) {
        return set.contains(word);
    }

    // 判断集合中是否有前缀为 prefix 的元素
    public boolean startsWith(String prefix) {
        return set.hasKeyWithPrefix(prefix);
    }
}
```

```
// 见上文
class TrieSet {}
```

## 648. 单词替换

接下来看下力扣第 648 题「单词替换」：

### ▼ 648. 单词替换 [Leetcode | 力扣](#)

在英语中，我们有一个叫做 **词根**(root) 的概念，可以词根 **后面** 添加其他一些词组成另一个较长的单词——我们称这个词为**衍生词** (**derivative**)。例如，词根 **help**，跟随着 **继承词** "**ful**"，可以形成新的单词 "**helpful**"。

现在，给定一个由许多 **词根** 组成的词典 **dictionary** 和一个用空格分隔单词形成的句子 **sentence**。你需要将句子中的所有 **衍生词** 用 **词根** 替换掉。如果 **衍生词** 有许多可以形成它的 **词根**，则用 **最短的词根** 替换它。

你需要输出替换之后的句子。

#### 示例 1:

```
输入: dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by the
battery"
输出: "the cat was rat by the bat"
```

#### 示例 2:

```
输入: dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"
输出: "a a b c"
```

#### 提示:

- $1 \leq \text{dictionary.length} \leq 1000$
- $1 \leq \text{dictionary[i].length} \leq 100$
- **dictionary[i]** 仅由小写字母组成。
- $1 \leq \text{sentence.length} \leq 10^6$
- **sentence** 仅由小写字母和空格组成。
- **sentence** 中单词的总量在范围  $[1, 1000]$  内。
- **sentence** 中每个单词的长度在范围  $[1, 1000]$  内。
- **sentence** 中单词之间由一个空格隔开。
- **sentence** 没有前导或尾随空格。

现在你学过 Trie 树结构，应该可以看出来这题就在考察最短前缀问题。

所以可以把输入的词根列表 **dict** 存入 **TrieSet**，然后直接复用我们实现的 **shortestPrefixOf** 函数就行了：

```
String replaceWords(List<String> dict, String sentence) {
    // 先将词根都存入 TrieSet
    TrieSet set = new TrieSet();
    for (String key : dict) {
        set.add(key);
    }
    StringBuilder sb = new StringBuilder();
```

```
String[] words = sentence.split(" ");
// 处理句子中的单词
for (int i = 0; i < words.length; i++) {
    // 在 Trie 树中搜索最短词根（最短前缀）
    String prefix = set.shortestPrefixOf(words[i]);
    if (!prefix.isEmpty()) {
        // 如果搜索到了，改写为词根
        sb.append(prefix);
    } else {
        // 否则，原样放回
        sb.append(words[i]);
    }

    if (i != words.length - 1) {
        // 添加单词之间的空格
        sb.append(' ');
    }
}

return sb.toString();
}

// 见上文
class TrieSet {}

// 见上文
class TrieMap {}
```

## 211. 添加与搜索单词 - 数据结构设计

继续看力扣第 211 题「添加与搜索单词 - 数据结构设计」：

▼ 211. 添加与搜索单词 - 数据结构设计 [Leetcode | 力扣](#)

请你设计一个数据结构，支持 **添加新单词** 和 **查找字符串是否与任何先前添加的字符串匹配**。

实现词典类 **WordDictionary**：

- **WordDictionary()** 初始化词典对象
- **void addWord(word)** 将 **word** 添加到数据结构中，之后可以对它进行匹配
- **bool search(word)** 如果数据结构中存在字符串与 **word** 匹配，则返回 **true**；否则，返回 **false**。**word** 中可能包含一些 **%%%%%**，每个 **.** 都可以表示任何一个字母。

示例：

输入：

```
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
```

输出：

```
[null,null,null,null,false,true,true,true]
```

解释：

```
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
```

```
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // 返回 False
wordDictionary.search("bad"); // 返回 True
wordDictionary.search(".ad"); // 返回 True
wordDictionary.search("b.."); // 返回 True
```

提示：

- $1 \leq word.length \leq 25$
- `addWord` 中的 `word` 由小写英文字母组成
- `search` 中的 `word` 由 %%%%%%%%%%%%% 或小写英文字母组成
- 最多调用  $10^4$  次 `addWord` 和 `search`

这道题的考点就在于这个 `search` 函数进行通配符匹配，其实是我们给 `TrieSet` 实现的 `hasKeyWithPattern` 方法，直接套就行了：

```
class WordDictionary {
    TrieSet set = new TrieSet();

    // 在 TrieSet 中添加元素
    public void addWord(String word) {
        set.add(word);
    }

    // 通配符匹配元素
    public boolean search(String word) {
        return set.hasKeyWithPattern(word);
    }
}

// 见上文
class TrieSet {}

// 见上文
class TrieMap {}
```

上面列举的这几道题用的都是 `TrieSet`，下面来看看 `TrieMap` 的题目。

## 1804. 实现一个 Trie (前缀树 II)

先看力扣第 1804 题「实现前缀树 II」：

### ▼ 1804. 实现 Trie (前缀树) II | Leetcode | 力扣

前缀树 (`trie`，发音为 "try") 是一个树状的数据结构，用于高效地存储和检索一系列字符串的前缀。前缀树有许多应用，如自动补全和拼写检查。

实现前缀树 `Trie` 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 将字符串 `word` 插入前缀树中。
- `int countWordsEqualTo(String word)` 返回前缀树中字符串 `word` 的实例个数。
- `int countWordsStartingWith(String prefix)` 返回前缀树中以 `prefix` 为前缀的字符串个数。
- `void erase(String word)` 从前缀树中移除字符串 `word`。

## 示例 1：

## 输入

```
["Trie", "insert", "insert", "countWordsEqualTo", "countWordsStartingWith",
"erase", "countWordsEqualTo", "countWordsStartingWith", "erase",
"countWordsStartingWith"]
[[], ["apple"], ["apple"], ["apple"], ["app"], ["apple"], ["apple"], ["app"],
["apple"], ["app"]]
```

## 输出

```
[null, null, null, 2, 2, null, 1, 1, null, 0]
```

## 解释

```
Trie trie = new Trie();
trie.insert("apple");           // 插入 "apple"。
trie.insert("apple");           // 插入另一个 "apple"。
trie.countWordsEqualTo("apple"); // 有两个 "apple" 实例，所以返回 2。
trie.countWordsStartingWith("app"); // "app" 是 "apple" 的前缀，所以返回 2。
trie.erase("apple");           // 移除一个 "apple"。
trie.countWordsEqualTo("apple"); // 现在只有一个 "apple" 实例，所以返回 1。
trie.countWordsStartingWith("app"); // 返回 1
trie.erase("apple");           // 移除 "apple"。现在前缀树是空的。
trie.countWordsStartingWith("app"); // 返回 0
```

## 提示：

- $1 \leq \text{word.length}, \text{prefix.length} \leq 2000$
- `word` 和 `prefix` 只包含小写英文字母。
- `insert`、`countWordsEqualTo`、`countWordsStartingWith` 和 `erase` 总共调用最多  $3 * 10^4$  次。
- 保证每次调用 `erase` 时，字符串 `word` 总是存在于前缀树中。

这题就可以用到 `TrieMap`，每个插入的 `word` 就是键，插入的次数就是对应的值，然后复用 `TrieMap` 的 API 就能实现题目要求的这些函数：

```
class Trie {
    // 封装我们实现的 TrieMap
    TrieMap<Integer> map = new TrieMap<>();

    // 插入 word 并记录插入次数
    public void insert(String word) {
        if (!map.containsKey(word)) {
            map.put(word, 1);
        } else {
            map.put(word, map.get(word) + 1);
        }
    }

    // 查询 word 插入的次数
    public int countWordsEqualTo(String word) {
        if (!map.containsKey(word)) {
            return 0;
        }
        return map.get(word);
    }

    // 累加前缀为 prefix 的键的插入次数总和
}
```

```
public int countWordsStartingWith(String prefix) {
    int res = 0;
    for (String key : map.keysWithPrefix(prefix)) {
        res += map.get(key);
    }
    return res;
}

// word 的插入次数减一
public void erase(String word) {
    int freq = map.get(word);
    if (freq - 1 == 0) {
        map.remove(word);
    } else {
        map.put(word, freq - 1);
    }
}

// 见上文
class TrieMap {}
```

## 677. 键值映射

反正都是直接套模板，也没什么意思，再看最后一道题目吧，这是力扣第 677 题「键值映射」：

### ▼ 677. 键值映射 [Leetcode](#) | 力扣

设计一个 map，满足以下几点：

- 字符串表示键，整数表示值
- 返回具有前缀等于给定字符串的键的值的总和

实现一个 MapSum 类：

- `MapSum()` 初始化 MapSum 对象
- `void insert(String key, int val)` 插入 `key-val` 键值对，字符串表示键 `key`，整数表示值 `val`。如果键 `key` 已经存在，那么原来的键值对 `key-value` 将被替代成新的键值对。
- `int sum(string prefix)` 返回所有以该前缀 `prefix` 开头的键 `key` 的值的总和。

示例 1：

```
输入：  
["MapSum", "insert", "sum", "insert", "sum"]  
[], ["apple", 3], ["ap"], ["app", 2], ["ap"]]
```

```
输出：  
[null, null, 3, null, 5]
```

解释：

```
MapSum mapSum = new MapSum();  
mapSum.insert("apple", 3);  
mapSum.sum("ap");           // 返回 3 (apple = 3)  
mapSum.insert("app", 2);  
mapSum.sum("ap");           // 返回 5 (apple + app = 3 + 2 = 5)
```

提示：

- $1 \leq \text{key.length}, \text{prefix.length} \leq 50$
- **key** 和 **prefix** 仅由小写英文字母组成
- $1 \leq \text{val} \leq 1000$
- 最多调用 50 次 **insert** 和 **sum**

这道题还是标准的 **TrieMap** 的应用，直接看代码吧：

```
class MapSum {  
    // 封装我们实现的 TrieMap  
    TrieMap<Integer> map = new TrieMap<>();  
  
    // 插入键值对  
    public void insert(String key, int val) {  
        map.put(key, val);  
    }  
  
    // 累加所有前缀为 prefix 的键的值  
    public int sum(String prefix) {  
        List<String> keys = map.keysWithPrefix(prefix);  
        int res = 0;  
        for (String key : keys) {  
            res += map.get(key);  
        }  
        return res;  
    }  
}  
  
// 见上文  
class TrieMap {}
```

Trie 树这种数据结构的实现原理和题目实践就讲完了，如果你能够看到这里，真得给你鼓掌。纸上得来终觉浅，绝知此事要躬行，我建议最好亲手实现一遍上面的代码，去把题目刷一遍，才能对 Trie 树有更深入的理解。

► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">14. Longest Common Prefix</a>	<a href="#">14. 最长公共前缀</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 设计朋友圈时间线功能



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">355. Design Twitter</a>	<a href="#">355. 设计推特</a>	困难

阅读本文前，你需要先学习：

- 链表基础
- 哈希表基础
- 二叉堆基础

力扣第 355 「设计推特」不仅题目本身很有意思，而且把合并多个有序链表的算法和面向对象设计（OO design）结合起来了，很有实际意义，本文就带大家来看看这道题。

至于 Twitter 的什么功能跟算法有关系，等我们描述一下题目要求就知道了。

## 一、题目及应用场景简介

Twitter 和微博功能差不多，我们主要实现这样几个 API：

```
class Twitter {  
  
    // user 发表一条 tweet 动态  
    public void postTweet(int userId, int tweetId) {}  
  
    // 返回该 user 关注的人（包括他自己）最近的动态 id  
    // 最多 10 条，而且这些动态必须按从新到旧的时间线顺序排列  
    public List<Integer> getNewsFeed(int userId) {}  
  
    // follower 关注 followee，如果 Id 不存在则新建  
    public void follow(int followerId, int followeeId) {}  
  
    // follower 取关 followee，如果 Id 不存在则什么都不做  
    public void unfollow(int followerId, int followeeId) {}  
}
```

举个具体的例子，方便大家理解 API 的具体用法：

```
Twitter twitter = new Twitter();

twitter.postTweet(1, 5);
// 用户 1 发送了一条新推文 5

twitter.getNewsFeed(1);
// return [5], 因为自己是关注自己的

twitter.follow(1, 2);
// 用户 1 关注了用户 2

twitter.postTweet(2, 6);
// 用户2发送了一个新推文 (id = 6)

twitter.getNewsFeed(1);
// return [6, 5]
// 解释：用户 1 关注了自己和用户 2，所以返回他们的最近推文
// 而且 6 必须在 5 之前，因为 6 是最近发送的

twitter.unfollow(1, 2);
// 用户 1 取消关注了用户 2

twitter.getNewsFeed(1);
// return [5]
```

这个场景在我们的现实生活中非常常见。拿朋友圈举例，比如我刚加到女神的微信，然后我去刷新一下我的朋友圈动态，那么女神的动态就会出现在我的动态列表，而且会和其他动态按时间排好序。只不过 Twitter 是单向关注，微信好友相当于双向关注。除非，被屏蔽...

这几个 API 中大部分都很好实现，最核心的功能难点应该是 `getNewsFeed`，因为返回的结果必须在时间上有序，但问题是用户的关注是动态变化的，怎么办？

这里就涉及到算法了：如果我们把每个用户各自的推文存储在链表里，每个链表节点存储文章 `id` 和一个时间戳 `time`（记录发帖时间以便比较），而且这个链表是按 `time` 有序的，那么如果某个用户关注了 `k` 个用户，我们就可以用合并 `k` 个有序链表的算法合并出有序的推文列表，正确地 `getNewsFeed` 了！

具体的算法等会讲解。不过，就算我们掌握了算法，应该如何编程表示用户 `User` 和推文动态 `Tweet` 才能把算法流畅地用出来呢？这就涉及简单的面向对象设计了，下面我们来由浅入深，一步一步进行设计。

## 二、面向对象设计

根据刚才的分析，我们需要一个 `User` 类，储存 `user` 信息，还需要一个 `Tweet` 类，储存推文信息，并且要作为链表的节点。所以我们先搭建一下整体的框架：

```
class Twitter {
    private static int timestamp = 0;
    private static class Tweet {}
    private static class User {}

    // 还有那几个 API 方法
    public void postTweet(int userId, int tweetId) {}
    public List<Integer> getNewsFeed(int userId) {}
    public void follow(int followerId, int followeeId) {}
    public void unfollow(int followerId, int followeeId) {}

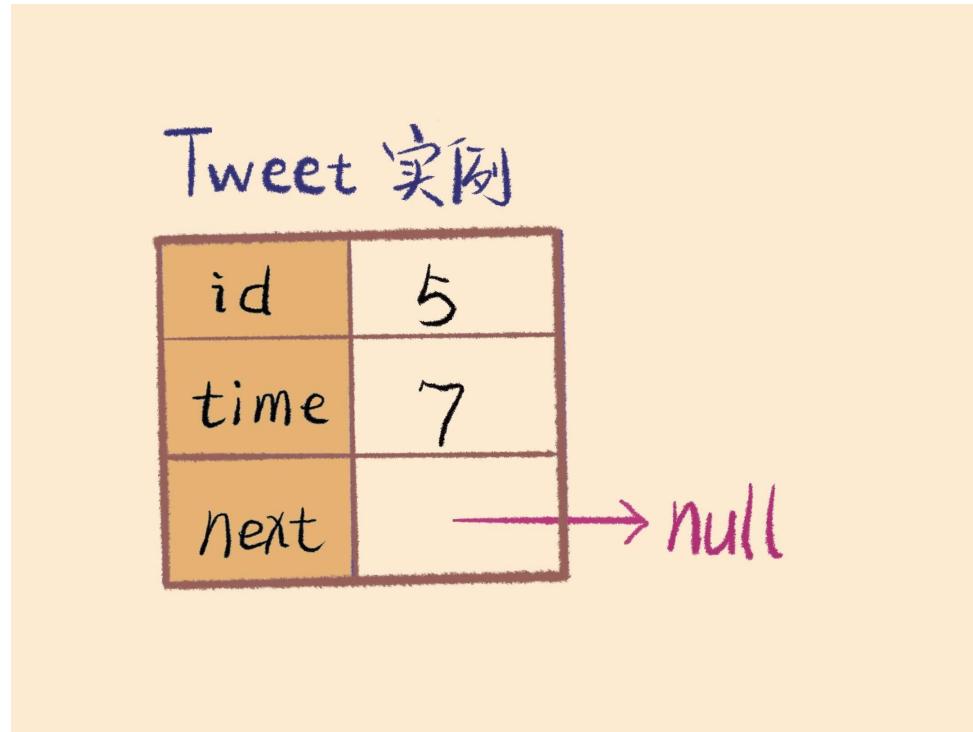
}
```

之所以要把 `Tweet` 和 `User` 类放到 `Twitter` 类里面，是因为 `Tweet` 类必须要用到一个全局时间戳 `timestamp`，而 `User` 类又需要用到 `Tweet` 类记录用户发送的推文，所以它们都作为内部类。不过为了清晰和简洁，下文会把每个内部类和 API 方法单独拿出来实现。

## Tweet 类的实现

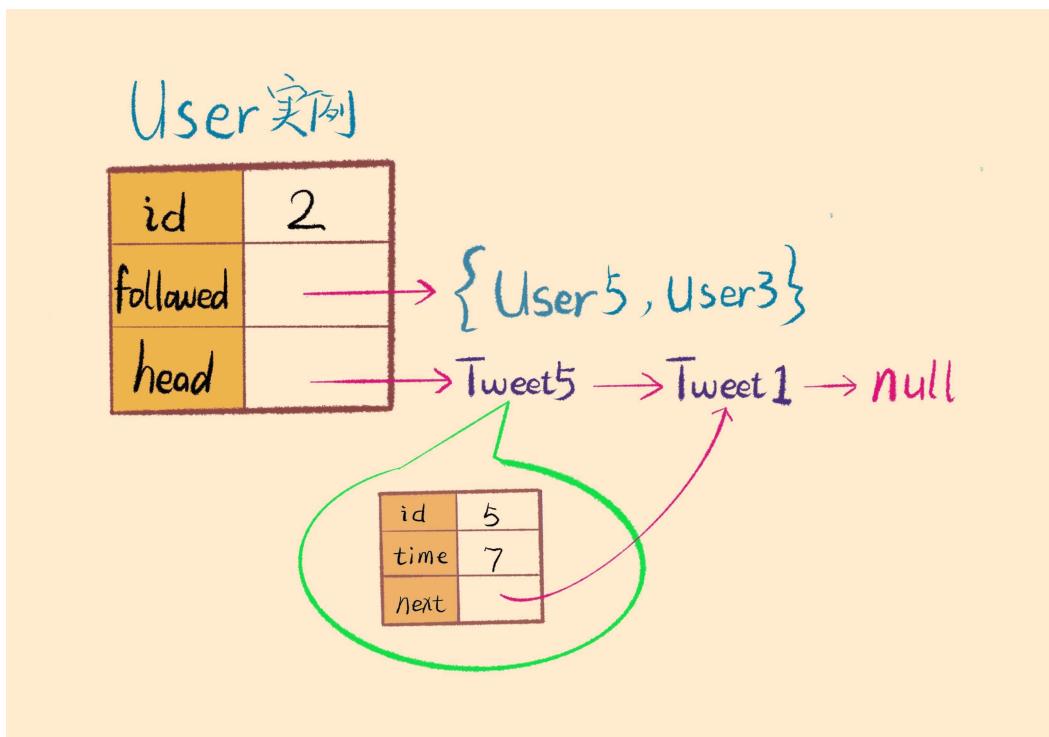
根据前面的分析，`Tweet` 类很容易实现：每个 `Tweet` 实例需要记录自己的 `tweetId` 和发表时间 `time`，而且作为链表节点，要有一个指向下一个节点的 `next` 指针。

```
class Tweet {  
    private int id;  
    private int time;  
    private Tweet next;  
  
    // 需要传入推文内容 (id) 和发文时间  
    public Tweet(int id, int time) {  
        this.id = id;  
        this.time = time;  
        this.next = null;  
    }  
}
```



## User 类的实现

我们根据实际场景想一想，一个用户需要存储的信息有 `userId`，关注列表，以及该用户发过的推文列表。其中关注列表应该用集合（Hash Set）这种数据结构来存，因为不能重复，而且需要快速查找；推文列表应该由链表这种数据结构储存，以便于进行有序合并的操作。画个图理解一下：



除此之外，根据面向对象的设计原则，「关注」、「取关」和「发文」应该是 User 的行为，况且关注列表和推文列表也存储在 User 类中，所以我们也应该给 User 添加 follow, unfollow 和 post 这几个方法：

```
// static int timestamp = 0
class User {
    private int id;
    public Set<Integer> followed;
    // 用户发表的推文链表头结点
    public Tweet head;

    public User(int userId) {
        followed = new HashSet<>();
        this.id = userId;
        this.head = null;
        // 关注一下自己
        follow(id);
    }

    public void follow(int userId) {
        followed.add(userId);
    }

    public void unfollow(int userId) {
        // 不可以取关自己
        if (userId != this.id)
            followed.remove(userId);
    }

    public void post(int tweetId) {
        Tweet twt = new Tweet(tweetId, timestamp);
        timestamp++;
        // 将新建的推文插入链表头
        // 越靠前的推文 time 值越大
        twt.next = head;
        head = twt;
    }
}
```

```
    }
}
```

## 几个 API 方法的实现

```
class Twitter {
    private static int timestamp = 0;
    private static class Tweet {...}
    private static class User {...}

    // 我们需要一个映射将 userId 和 User 对象对应起来
    private HashMap<Integer, User> userMap = new HashMap<>();

    // user 发表一条 tweet 动态
    public void postTweet(int userId, int tweetId) {
        // 若 userId 不存在, 则新建
        if (!userMap.containsKey(userId))
            userMap.put(userId, new User(userId));
        User u = userMap.get(userId);
        u.post(tweetId);
    }

    // follower 关注 followee
    public void follow(int followerId, int followeeId) {
        // 若 follower 不存在, 则新建
        if (!userMap.containsKey(followerId)){
            User u = new User(followerId);
            userMap.put(followerId, u);
        }
        // 若 followee 不存在, 则新建
        if (!userMap.containsKey(followeeId)){
            User u = new User(followeeId);
            userMap.put(followeeId, u);
        }
        userMap.get(followerId).follow(followeeId);
    }

    // follower 取关 followee, 如果 Id 不存在则什么都不做
    public void unfollow(int followerId, int followeeId) {
        if (userMap.containsKey(followerId)) {
            User flwer = userMap.get(followerId);
            flwer.unfollow(followeeId);
        }
    }

    // 返回该 user 关注的人 (包括他自己) 最近的动态 id
    // 最多 10 条, 而且这些动态必须按从新到旧的时间线顺序排列
    public List<Integer> getNewsFeed(int userId) {
        // 需要理解算法, 见下文
    }
}
```

## 三、算法设计

实现合并 k 个有序链表的算法需要用到优先级队列（Priority Queue），这种数据结构是二叉堆最重要的应用。你可以理解为它可以对插入的元素自动排序，乱序的元素插入其中就被放到了正确的位置，可以按照从小到大（或从大到小）有序地取出元素。具体可以看这篇 [二叉堆实现优先级队列](#)。

```
PriorityQueue pq
# 乱序插入
for i in {2,4,1,9,6}:
    pq.add(i)
while pq not empty:
    # 每次取出第一个（最小）元素
    print(pq.pop())

# 输出有序: 1,2,4,6,9
```

借助这种牛逼的数据结构支持，我们就很容易实现这个核心功能了。注意我们把优先级队列设为按 time 属性从大到小降序排列，因为 time 越大意味着时间越近，应该排在前面：

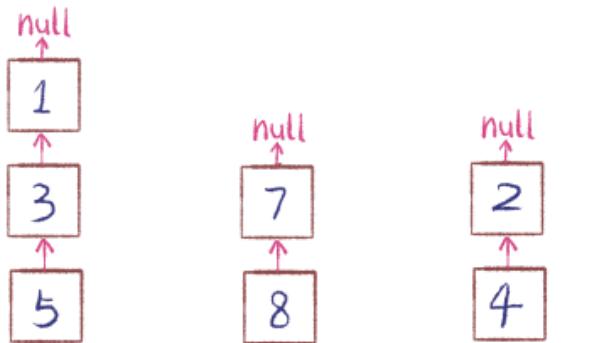
```
class Twitter {
    // 为了节约篇幅，省略上文给出的代码部分...

    public List<Integer> getNewsFeed(int userId) {
        List<Integer> res = new ArrayList<>();
        if (!userMap.containsKey(userId)) return res;
        // 关注列表的用户 Id
        Set<Integer> users = userMap.get(userId).followed;
        // 自动通过 time 属性从大到小排序，容量为 users 的大小
        PriorityQueue<Tweet> pq =
            new PriorityQueue<>(users.size(), (a, b)->(b.time - a.time));

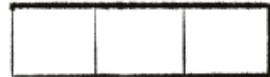
        // 先将所有链表头节点插入优先级队列
        for (int id : users) {
            Tweet twt = userMap.get(id).head;
            if (twt == null) continue;
            pq.add(twt);
        }

        while (!pq.isEmpty()) {
            // 最多返回 10 条就够了
            if (res.size() == 10) break;
            // 弹出 time 值最大的（最近发表的）
            Tweet twt = pq.poll();
            res.add(twt.id);
            // 将下一篇 Tweet 插入进行排序
            if (twt.next != null)
                pq.add(twt.next);
        }
        return res;
    }
}
```

这个过程是这样的，下面是我制作的一个 GIF 图描述合并链表的过程。假设有三个 Tweet 链表按 time 属性降序排列，我们把他们降序合并添加到 res 中。注意图中链表节点中的数字是 time 属性，不是 id 属性：



优先级队列：



res :

至此，这道一个极其简化的 Twitter 时间线功能就设计完毕了，更多数据结构设计相关的题目参见 [数据结构设计经典习题](#)。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 设计考场座位分配算法



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">855. Exam Room</a>	<a href="#">855. 考场就座</a>	困难

本文讲一讲力扣第 855 题「考场就座」，有趣且具有一定技巧性。这种题目并不像动态规划这类算法拼智商，而是看你对常用数据结构的理解和写代码的水平。

先来描述一下题目：假设有一个考场，考场有一排共  $N$  个座位，索引分别是  $[0..N-1]$ ，考生会陆续进入考场考试，并且可能在任何时候离开考场。

你作为考官，要安排考生们的座位，满足：**每当一个学生进入时，你需要最大化他和最近其他人的距离；如果有多个这样的座位，安排到他到索引最小的那个座位。**这很符合实际情况对吧，

也就是请你实现下面这样一个类：

```
class ExamRoom {
    // 构造函数，传入座位总数 N
    public ExamRoom(int N);
    // 来了一名考生，返回你给他分配的座位
    public int seat();
    // 坐在 p 位置的考生离开了
    // 可以认为 p 位置一定坐有考生
    public void leave(int p);
}
```

比方说考场有 5 个座位，分别是  $[0..4]$ ：

第一名考生进入时（调用 `seat()`），坐在任何位置都行，但是要给他安排索引最小的位置，也就是返回位置 0。

第二名学生进入时（再调用 `seat()`），要和旁边的人距离最远，也就是返回位置 4。

第三名学生进入时，要和旁边的人距离最远，应该做到中间，也就是座位 2。

如果再进一名学生，他可以坐在座位 1 或者 3，取较小的索引 1。

以此类推。

刚才所说的情况，没有调用 `leave` 函数，不过读者肯定能够发现规律：

如果将每两个相邻的考生看做线段的两端点，新安排考生就是找最长的线段，然后让该考生在中间把这个线段「二分」，中点就是给他分配的座位。`leave(p)` 其实就是去除端点  $p$ ，使得相邻两个线段合并为一个。

核心思路很简单对吧，所以这个问题实际上在考察你对数据结构的理解。对于上述这个逻辑，你用什么数据结构来实现呢？

## 一、思路分析

根据上述思路，首先需要把坐在教室的学生抽象成线段，我们可以简单的用一个大小为 2 的数组表示。

另外，思路需要我们找到「最长」的线段，还需要去除线段，增加线段。

但凡遇到在动态过程中取最值的要求，肯定要使用有序数据结构，我们常用的数据结构就是二叉堆和平衡二叉搜索树了。

二叉堆实现的优先级队列取最值的时间复杂度是  $O(\log N)$ ，但是只能删除最大值。平衡二叉树也可以取最值，也可以修改、删除任意一个值，而且时间复杂度都是  $O(\log N)$ 。

综上，二叉堆不能满足 `leave` 操作，应该使用平衡二叉树。所以这里我们会用到 Java 的一种数据结构 `TreeSet`，这是一种有序数据结构，底层由红黑树维护有序性。

这里顺便提一下，一说到集合（Set）或者映射（Map），有的读者可能就想当然的认为是哈希集合（HashSet）或者哈希表（HashMap），这样理解是有点问题的。

因为哈希集合/映射底层是由哈希函数和数组实现的，特性是遍历无固定顺序，但是操作效率高，时间复杂度为  $O(1)$ 。

而集合/映射还可以依赖其他底层数据结构，常见的就是红黑树（一种平衡二叉搜索树），特性是自动维护其中元素的顺序，操作效率是  $O(\log N)$ 。这种一般称为「有序集合/映射」。

我们使用的 `TreeSet` 就是一个有序集合，目的就是为了保持线段长度的有序性，快速查找最大线段，快速删除和插入。

## 二、简化问题

首先，如果有多个可选座位，需要选择索引最小的座位对吧？我们先简化一下问题，暂时不管这个要求，实现上述思路。

这个问题还用到一个常用的编程技巧，就是使用一个「虚拟线段」让算法正确启动，这就和链表相关的算法需要「虚拟头结点」一个道理。

```
class ExamRoom {
    // 将端点 p 映射到以 p 为左端点的线段
    private Map<Integer, int[]> startMap;
    // 将端点 p 映射到以 p 为右端点的线段
    private Map<Integer, int[]> endMap;
    // 根据线段长度从小到大存放所有线段
    private TreeSet<int[]> pq;
    private int N;

    public ExamRoom(int N) {
        this.N = N;
        startMap = new HashMap<>();
        endMap = new HashMap<>();
        pq = new TreeSet<>((a, b) -> {
            // 算出两个线段的长度
            int distA = distance(a);
            int distB = distance(b);
            // 长度更长的更大，排后面
            return distA - distB;
        });
        // 在有序集合中先放一个虚拟线段
        addInterval(new int[] {-1, N});
    }
}
```

```

/* 去除一个线段 */
/* remove a segment */
private void removeInterval(int[] intv) {
    pq.remove(intv);
    startMap.remove(intv[0]);
    endMap.remove(intv[1]);
}

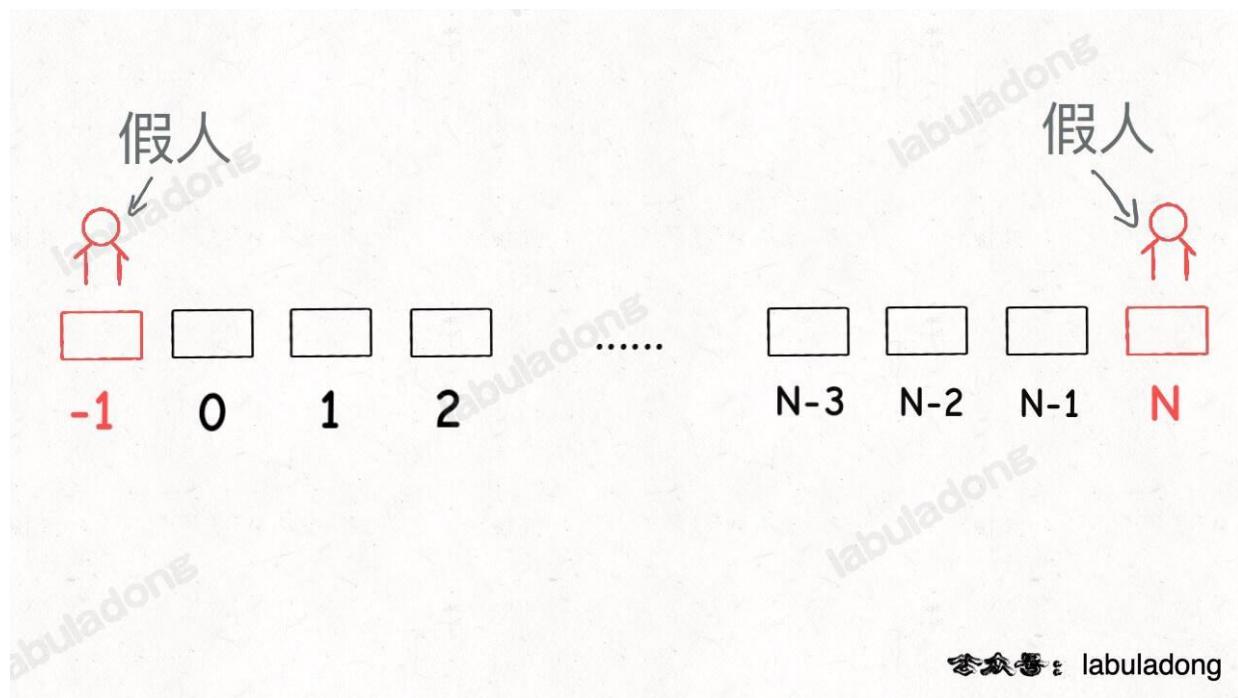
/* 增加一个线段 */
/* add a segment */
private void addInterval(int[] intv) {
    pq.add(intv);
    startMap.put(intv[0], intv);
    endMap.put(intv[1], intv);
}

/* 计算一个线段的长度 */
/* calculate the length of a segment */
private int distance(int[] intv) {
    return intv[1] - intv[0] - 1;
}

// ...
}

```

「虚拟线段」其实就是为了将所有座位表示为一个线段：



有了上述铺垫，主要 API `seat` 和 `leave` 就可以写了：

```

class ExamRoom {
    // ...

    public int seat() {
        // 从有序集合拿出最长的线段
        int[] longest = pq.last();

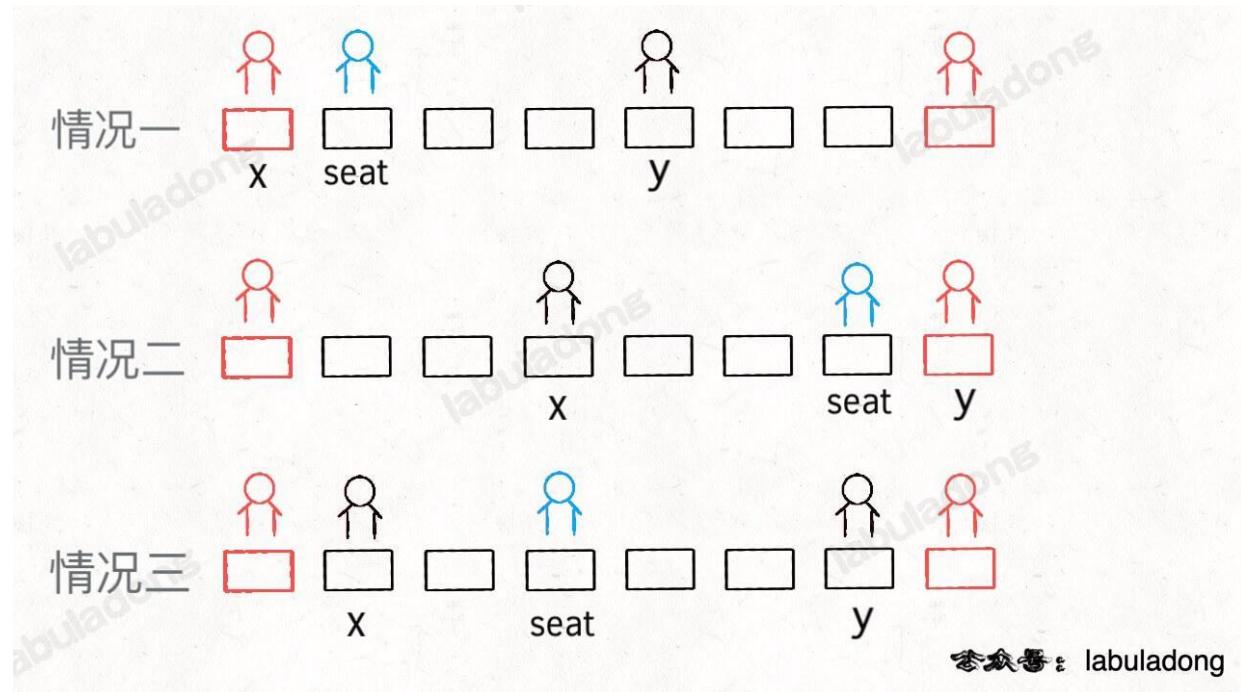
```

```

int x = longest[0];
int y = longest[1];
int seat;
if (x == -1) { // 情况一
    seat = 0;
} else if (y == N) { // 情况二
    seat = N - 1;
} else { // 情况三
    seat = (y - x) / 2 + x;
}
// 将最长的线段分成两段
int[] left = new int[] {x, seat};
int[] right = new int[] {seat, y};
removeInterval(longest);
addInterval(left);
addInterval(right);
return seat;
}

public void leave(int p) {
    // 将 p 左右的线段找出来
    int[] right = startMap.get(p);
    int[] left = endMap.get(p);
    // 合并两个线段成为一个线段
    int[] merged = new int[] {left[0], right[1]};
    removeInterval(left);
    removeInterval(right);
    addInterval(merged);
}
}

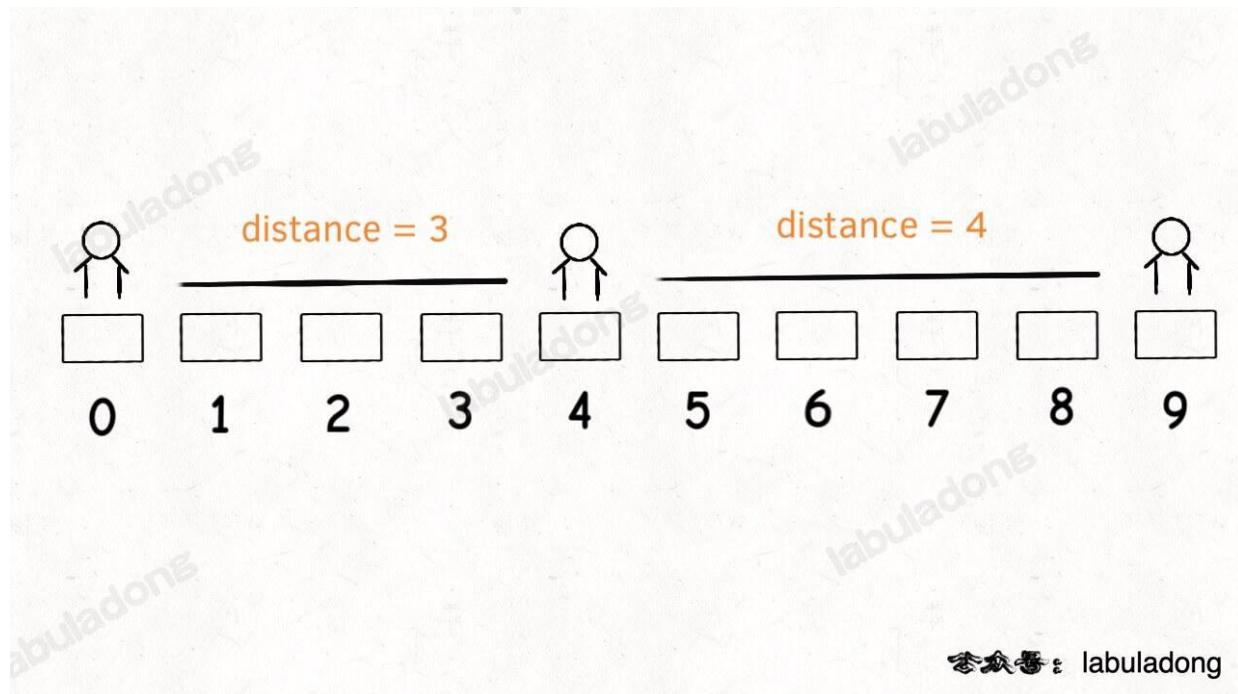
```



至此，算法就基本实现了，代码虽多，但思路很简单：找最长的线段，从中间分隔成两段，中点就是 `seat()` 的返回值；找 `p` 的左右线段，合并成一个线段，这就是 `leave(p)` 的逻辑。

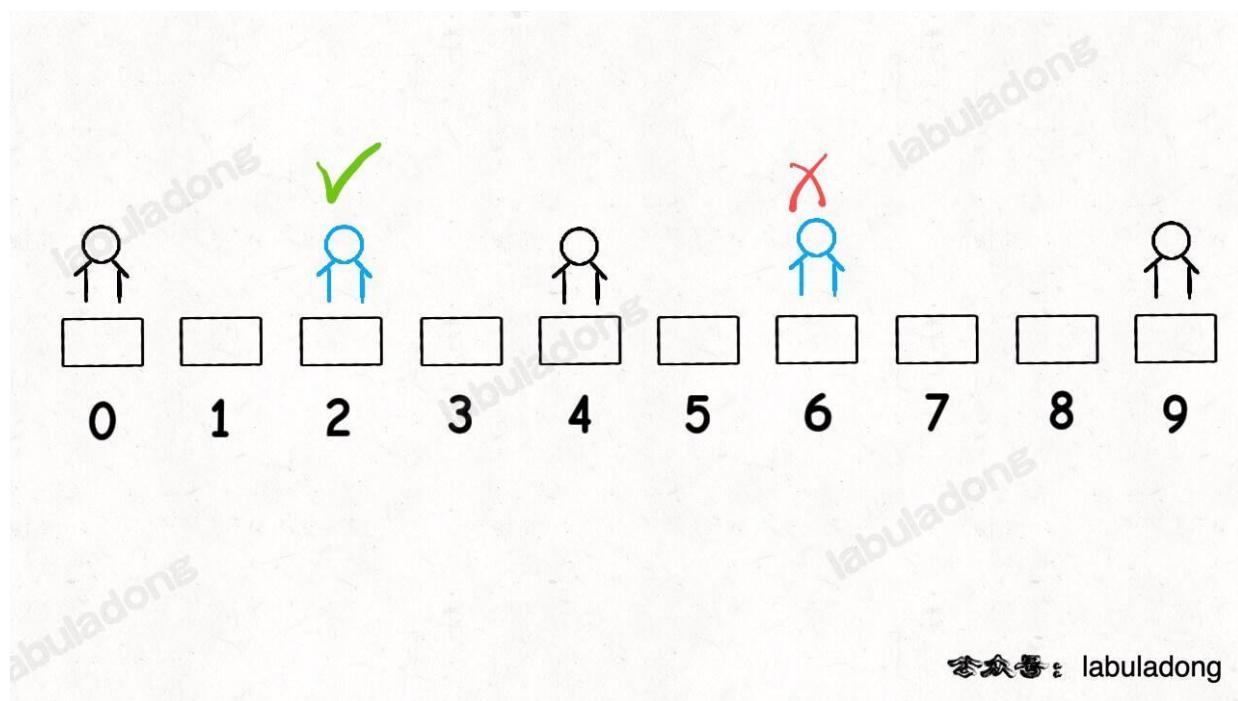
### 三、进阶问题

但是，题目要求多个选择时选择索引最小的那个座位，我们刚才忽略了这个问题。比如下面这种情况会出错：



参考书：labuladong

现在有序集合里有线段  $[0, 4]$  和  $[4, 9]$ ，那么最长线段 `longest` 就是后者，按照 `seat` 的逻辑，就会分割  $[4, 9]$ ，也就是返回座位 6。但正确答案应该是座位 2，因为 2 和 6 都满足最大化相邻考生距离的条件，二者应该取较小的。



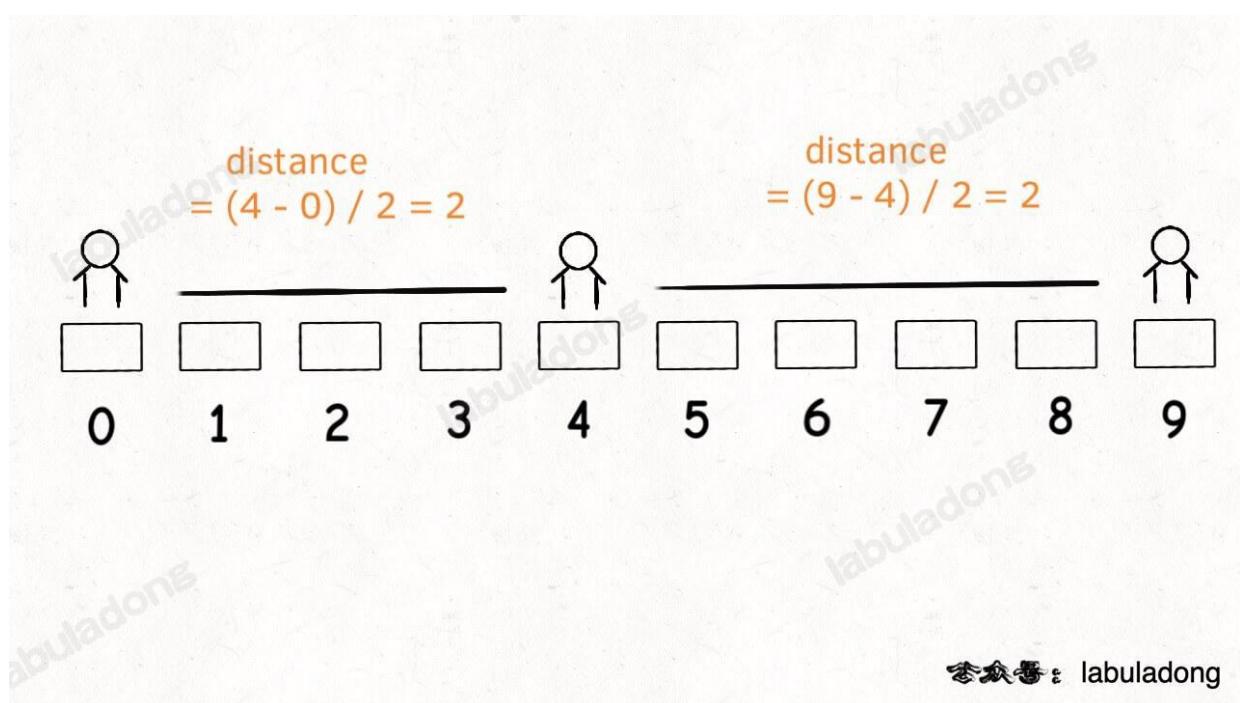
参考书：labuladong

遇到题目的这种要求，解决方式就是修改有序数据结构的排序方式。具体到这个问题，就是修改 `TreeMap` 的比较函数逻辑：

```
    pq = new TreeSet<>((a, b) -> {
        int distA = distance(a);
        int distB = distance(b);
        // 如果长度相同，就比较索引
        if (distA == distB)
            return b[0] - a[0];
        return distA - distB;
   });
```

除此之外，还要改变 `distance` 函数，不能简单地让它计算一个线段两个端点间的长度，而是让它计算该线段中点和端点之间的长度。

```
class ExamRoom {
    // ...
    private int distance(int[] intv) {
        int x = intv[0];
        int y = intv[1];
        if (x == -1) return y;
        if (y == N) return N - 1 - x;
        // 中点和端点之间的长度
        return (y - x) / 2;
    }
}
```



这样，`[0, 4]` 和 `[4, 9]` 的 `distance` 值就相等了，算法会比较二者的索引，取较小的线段进行分割。

这道算法题目算是完全解决了，完整代码如下：

```
class ExamRoom {
    // 将端点 p 映射到以 p 为左端点的线段
    private Map<Integer, int []> startMap;
    // 将端点 p 映射到以 p 为右端点的线段
    private Map<Integer, int []> endMap;
    // 根据线段长度从小到大存放所有线段
    private TreeSet<int []> pq;
    private int N;

    public ExamRoom(int N) {
        this.N = N;
    }
}
```

```

startMap = new HashMap<>();
endMap = new HashMap<>();
pq = new TreeSet<>((a, b) -> {
    int distA = distance(a);
    int distB = distance(b);
    // 如果长度相同，就比较索引
    if (distA == distB)
        return b[0] - a[0];
    return distA - distB;
});
// 在有序集合中先放一个虚拟线段
addInterval(new int[]{ -1, N });
}

public int seat() {
    // 从有序集合拿出最长的线段
    int[] longest = pq.last();
    int x = longest[0];
    int y = longest[1];
    int seat;
    // 情况一
    if (x == -1) {
        seat = 0;
    // 情况二
    } else if (y == N) {
        seat = N - 1;
    // 情况三
    } else {
        seat = (y - x) / 2 + x;
    }
    // 将最长的线段分成两段
    int[] left = new int[]{x, seat};
    int[] right = new int[]{seat, y};
    removeInterval(longest);
    addInterval(left);
    addInterval(right);
    return seat;
}

public void leave(int p) {
    // 将 p 左右的线段找出来
    int[] right = startMap.get(p);
    int[] left = endMap.get(p);
    // 合并两个线段成为一个线段
    int[] merged = new int[]{left[0], right[1]};
    removeInterval(left);
    removeInterval(right);
    addInterval(merged);
}

// 增加一个线段
private void addInterval(int[] intv) {
    pq.add(intv);
    startMap.put(intv[0], intv);
    endMap.put(intv[1], intv);
}

// 去除一个线段
private void removeInterval(int[] intv) {
}

```

```
    pq.remove(intv);
    startMap.remove(intv[0]);
    endMap.remove(intv[1]);
}

// 计算一个线段的长度
private int distance(int[] intv) {
    int x = intv[0];
    int y = intv[1];
    if (x == -1) return y;
    if (y == N) return N - 1 - x;
    // 中点和端点之间的长度
    return (y - x) / 2;
}
}
```

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】更多经典设计习题

阅读本文前，你需要先学习：

- [用链表加强哈希表](#)
- [用数组加强哈希表](#)

设计类题目都是让你把基本数据结构进行组合，去解决某些具体场景中的问题。我们先来看几道比较简单但比较有意思的题目吧。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：如何实现一个计算器



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">227. Basic Calculator II</a>	<a href="#">227. 基本计算器 II</a>	🟡
<a href="#">772. Basic Calculator III</a> 🔒	<a href="#">772. 基本计算器 III</a> 🔒	🔴
<a href="#">224. Basic Calculator</a>	<a href="#">224. 基本计算器</a>	🔴

阅读本文前，你需要先学习：

- 队列/栈的原理

我们最终要实现的计算器功能如下：

- 1、输入一个字符串，可以包含 `+` `-` `*` `/`、数字、括号以及空格，你的算法返回运算结果。
- 2、要符合运算法则，括号的优先级最高，先乘除后加减。
- 3、除号是整数除法，无论正负都向 0 取整 ( $5/2=2$ ,  $-5/2=-2$ )。
- 4、可以假定输入的算式一定合法，且计算过程不会出现整型溢出，不会出现除数为 0 的意外情况。

比如输入如下字符串，算法会返回 9：

$$\begin{aligned} & 3 * (2 - 6 / (3 - 7)) \\ & = 3 * (2 - 6 / (-4)) \\ & = 3 * (2 - (-1)) \\ & = 9 \end{aligned}$$

可以看到，这就已经非常接近我们实际生活中使用的计算器了，虽然我们以前肯定都用过计算器，但是如果简单思考一下其算法实现，就会大惊失色：

- 1、按照常理处理括号，要先计算最内层的括号，然后向外慢慢化简。这个过程我们手算都容易出错，何况写成算法呢！
- 2、要做到先乘除，后加减，这一点教会小朋友还不算难，但教给计算机恐怕有点困难。
- 3、要处理空格。我们为了美观，习惯性在数字和运算符之间打个空格，但是计算之中得想办法忽略这些空格。

我记得很多大学数据结构的教材上，在讲栈这种数据结构的时候，应该都会用计算器举例，但是有一说一，讲的真的垃圾，不知道多少未来的计算机科学家就被这种简单的数据结构劝退了。

那么本文就来聊聊怎么实现上述一个功能完备的计算器功能，**关键在于层层拆解问题，化整为零，逐个击破，几条简单的算法规则就可以处理极其复杂的运算，相信这种思维方式能帮大家解决各种复杂问题。**

下面就来拆解，从最简单的一个问题开始。

## 一、字符串转整数

是的，就是这么一个简单的问题，首先告诉我，怎么把一个字符串形式的正整数，转化成 int 型？

```
String s = "458";  
  
int n = 0;  
for (int i = 0; i < s.length(); i++) {  
    char c = s.charAt(i);  
    n = 10 * n + (c - '0');  
}  
// n 现在就等于 458
```

这个还是很简单的吧，老套路了。但是即便这么简单，依然有坑：`(c - '0')` 的这个括号不能省略，否则可能造成整型溢出。

因为变量 `c` 是一个 ASCII 码，如果不加括号就会先加后减，想象一下 `s` 如果接近 INT\_MAX，就会溢出。所以用括号保证先减后加才行。

## 二、处理加减法

现在进一步，如果输入的这个算式只包含加减法，而且不存在空格，你怎么计算结果？我们拿字符串算式 `1-12+3` 为例，来说一个很简单的思路：

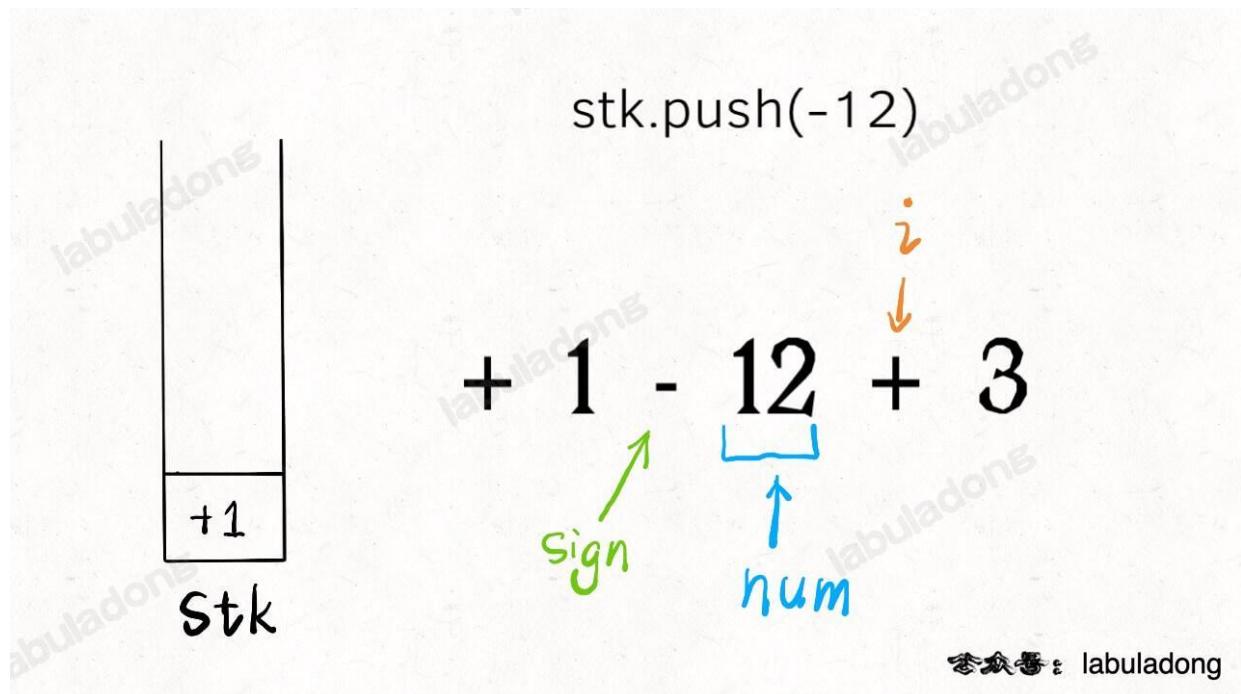
- 1、先给第一个数字加一个默认符号 `+`，变成 `+1-12+3`。
- 2、把一个运算符和数字组合成一对儿，也就是三对儿 `+1`, `-12`, `+3`，把它们转化成数字，然后放到一个栈中。
- 3、将栈中所有的数字求和，就是原算式的结果。

我们直接看代码，结合一张图就看明白了：

```
int calculate(String s) {  
    Stack<Integer> stk = new Stack<>();  
    // 记录算式中的数字  
    int num = 0;  
    // 记录 num 前的符号，初始化为 +  
    char sign = '+';  
    for (int i = 0; i < s.length(); i++) {  
        char c = s.charAt(i);  
        // 如果是数字，连续读取到 num  
        if (Character.isDigit(c)) {  
            num = 10 * num + (c - '0');  
        }  
        // 如果不是数字，就是遇到了下一个符号，或者是算式的末尾  
        // 那么之前的数字和符号就要存进栈中  
        if (c == '+' || c == '-' || i == s.length() - 1) {  
            switch (sign) {  
                case '+':  
                    stk.push(num); break;
```

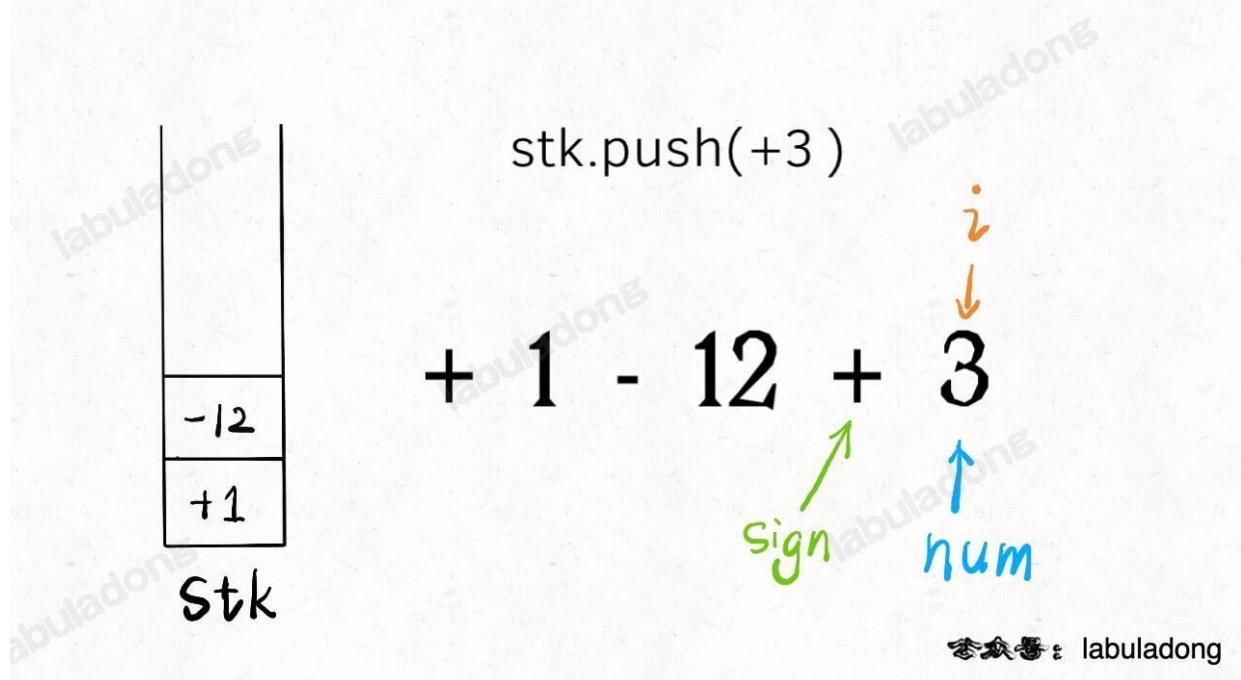
```
        case '-':
            stk.push(-num); break;
    }
    // 更新符号为当前符号，数字清零
    sign = c;
    num = 0;
}
}
// 将栈中所有结果求和就是答案
int res = 0;
while (!stk.isEmpty()) {
    res += stk.pop();
}
return res;
}
```

我估计就是中间带 `switch` 语句的部分有点不好理解吧，`i` 就是从左到右扫描，`sign` 和 `num` 跟在它身后。当 `s[i]` 遇到一个运算符时，情况是这样的：



所以说，此时要根据 `sign` 的 case 不同选择 `nums` 的正负号，存入栈中，然后更新 `sign` 并清零 `nums` 记录下一对儿符合和数字的组合。

另外注意，不只是遇到新的符号会触发入栈，当 `i` 走到了算式的尽头 (`i == s.size() - 1`)，也应该将前面的数字入栈，方便后续计算最终结果。



© labuladong

至此，仅处理紧凑加减法字符串的算法就完成了，请确保理解以上内容，后续的内容就基于这个框架修改完善事儿了。

### 三、处理乘除法

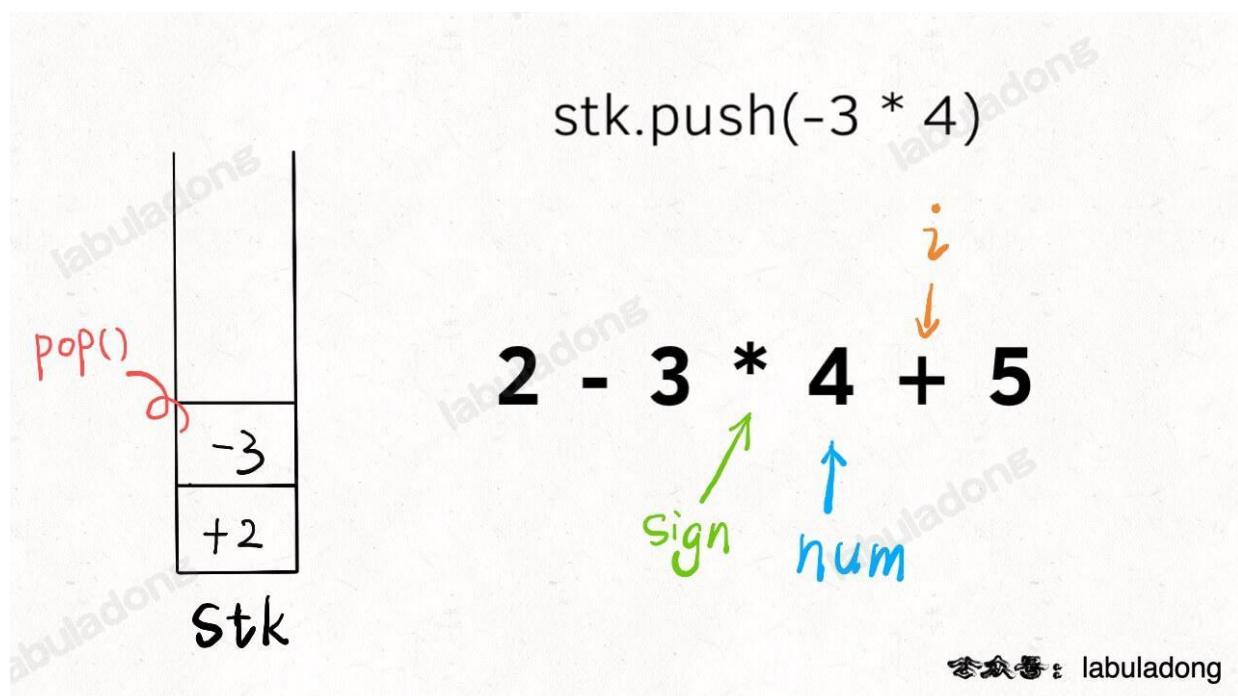
其实思路跟仅处理加减法没啥区别，拿字符串 `2-3*4+5` 举例，核心思路依然是把字符串分解成符号和数字的组合。

比如上述例子就可以分解为 `+2`, `-3`, `*4`, `+5` 几对儿，我们刚才不是没有处理乘除号吗，很简单，其他部分都不用变，在 `switch` 部分加上对应的 case 就行了：

```
int calculate(String s) {
    Stack<Integer> stk = new Stack<>();
    // 记录算式中的数字
    int num = 0;
    // 记录 num 前的符号，初始化为 +
    char sign = '+';
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (Character.isDigit(c)) {
            num = 10 * num + (c - '0');
        }

        if (c == '+' || c == '-' || c == '*' || c == '/' || i == s.length() - 1) {
            int pre;
            switch (sign) {
                case '+':
                    stk.push(num); break;
                case '-':
                    stk.push(-num); break;
                    // 只要拿出前一个数字做对应运算即可
                case '*':
                    pre = stk.pop();
                    stk.push(pre * num);
                    break;
                case '/':
                    pre = stk.pop();
```

```
        stk.push(pre / num);
        break;
    }
    // 更新符号为当前符号，数字清零
    sign = c;
    num = 0;
}
}
// 将栈中所有结果求和就是答案
int res = 0;
while (!stk.isEmpty()) {
    res += stk.pop();
}
return res;
}
```



乘除法优先于加减法体现在，乘除法可以和栈顶的数结合，而加减法只能把自己放入栈。

现在我们思考一下如何处理字符串中可能出现的空格字符。其实按照目前的代码，我们根本不用特殊处理空格字符，你注意 if 条件，当字符 `c` 是空格时，不会对它做任何处理，直接跳过了。

好了，我们现在的算法已经可以按照正确的法则计算加减乘除，并且自动忽略空格符，剩下的就是如何让算法正确识别括号了。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：两个二叉堆实现中位数算法



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">295. Find Median from Data Stream</a>	<a href="#">295. 数据流的中位数</a>	困难

阅读本文前，你需要先学习：

- [二叉堆原理](#)
- [二叉堆实现优先级队列](#)

如果输入一个数组，让你求中位数，这个好办，排个序，如果数组长度是奇数，最中间的一个元素就是中位数，如果数组长度是偶数，最中间两个元素的平均数作为中位数。

如果数据规模非常巨大，排序不太现实，那么也可以使用概率算法，随机抽取一部分数据，排序，求中位数，作为所有数据的中位数。

本文说的中位数算法比较困难，也比较精妙，是力扣第 295 题「数据流的中位数」：

### ▼ 295. 数据流的中位数 [Leetcode](#) | [力扣](#)

**中位数**是有序整数列表中的中间值。如果列表的大小是偶数，则没有中间值，中位数是两个中间值的平均值。

- 例如 `arr = [2,3,4]` 的中位数是 3。
- 例如 `arr = [2,3]` 的中位数是  $(2 + 3) / 2 = 2.5$ 。

实现 MedianFinder 类：

- `MedianFinder()` 初始化 `MedianFinder` 对象。
- `void addNum(int num)` 将数据流中的整数 `num` 添加到数据结构中。
- `double findMedian()` 返回到目前为止所有元素的中位数。与实际答案相差  $10^{-5}$  以内的答案将被接受。

### 示例 1：

**输入**

```
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
```

**输出**

```
[null, null, null, 1.5, null, 2.0]
```

### 解释

```
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);      // arr = [1]
medianFinder.addNum(2);      // arr = [1, 2]
medianFinder.findMedian();   // 返回 1.5 ((1 + 2) / 2)
medianFinder.addNum(3);      // arr[1, 2, 3]
medianFinder.findMedian();   // return 2.0
```

### 提示:

- $-10^5 \leq \text{num} \leq 10^5$
- 在调用 `findMedian` 之前，数据结构中至少有一个元素
- 最多  $5 * 10^4$  次调用 `addNum` 和 `findMedian`

```
// 题目让你设计这样一个类
class MedianFinder {

    // 添加一个数字
    public void addNum(int num) {}

    // 计算当前添加的所有数字的中位数
    public double findMedian() {}
}
```

其实，所有关于「流」的算法都比较难，比如我在前文 [谈谈游戏中的随机算法](#) 写过如何从数据流中等概率随机抽取一个元素，如果说你没有接触过这个问题的话，还是很难想到解法的。

这道题要求在数据流中计算平均数，我们先想一想常规思路。

## 尝试分析

一个直接的解法可以用一个数组记录所有 `addNum` 添加进来的数字，通过插入排序的逻辑保证数组中的元素有序，当调用 `findMedian` 方法时，可以通过数组索引直接计算中位数。

但是用数组作为底层容器的问题也很明显，`addNum` 搜索插入位置的时候可以用二分搜索算法，但是插入操作需要搬移数据，所以最坏时间复杂度为  $O(N)$ 。

那换链表？链表插入元素很快，但是查找插入位置的时候只能线性遍历，最坏时间复杂度还是  $O(N)$ ，而且 `findMedian` 方法也需要遍历寻找中间索引，最坏时间复杂度也是  $O(N)$ 。

那么就用平衡二叉树呗，增删查改复杂度都是  $O(\log N)$ ，这样总行了吧？

比如用 Java 提供的 `TreeSet` 容器，底层是红黑树，`addNum` 直接插入，`findMedian` 可以通过当前元素的个数推出计算中位数的元素的排名。

很遗憾，依然不行，这里有两个问题：

第一，`TreeSet` 是一种 `Set`，其中不存在重复元素的元素，但是我们的数据流可能输入重复数据的，而且计算中位数也是需要算上重复元素的。

第二，`TreeSet` 并没有实现一个通过排名快速计算元素的 API。假设我想找到 `TreeSet` 中第 5 大的元素，并没有一个现成可用的方法实现这个需求。

如果让你实现一个在二叉搜索树中通过排名计算对应元素的方法 `select(int index)`，你会怎么设计？你可以思考一下，我会把答案写在留言区置顶。

除了平衡二叉树，还有没有什么常用的数据结构是动态有序的？优先级队列（二叉堆）行不行？

好像也不太行，因为优先级队列是一种受限的数据结构，只能从堆顶添加/删除元素，我们的 `addNum` 方法可以从堆顶插入元素，但是 `findMedian` 函数需要从数据中间取，这个功能优先级队列是没办法提供的。

可以看到，求个中位数还是挺难的，我们使尽浑身解数都没有一个高效地思路，下面直接来看解法吧，比较巧妙。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 拓展：数组去重问题（困难版）



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1081. Smallest Subsequence of Distinct Characters</a>	<a href="#">1081. 不同字符的最小子序列</a>	
<a href="#">316. Remove Duplicate Letters</a>	<a href="#">316. 去除重复字母</a>	

-----

阅读本文前，你需要先学习：

- 队列/栈的原理
- 单调栈原理及实现

关于去重算法，应该没什么难度，往哈希集合里面塞不就行了么？

最多给你加点限制，问你怎么给有序数组原地去重，这个我们前文 [双指针技巧秒杀七道数组题目](#) 讲过。

本文讲的问题应该是去重相关算法中难度最大的了，把这个问题搞懂，就再也不用怕数组去重问题了。

这是力扣第 316 题「去除重复字母」，题目如下：

### ▼ 316. 去除重复字母 Leetcode | 力扣

给你一个字符串 `s`，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 **返回结果的字典序最小**（要求不能打乱其他字符的相对位置）。

示例 1：

```
输入: s = "bcabc"
输出: "abc"
```

示例 2：

```
输入: s = "cbacdcbc"
输出: "acdb"
```

提示：

- $1 \leq s.length \leq 10^4$

- $s$  由小写英文字母组成

注意：该题与 1081 <https://leetcode-cn.com/problems/smallest-subsequence-of-distinct-characters> 相同

这道题和第 1081 题「不同字符的最小子序列」的解法是完全相同的，你可以把这道题的解法代码直接粘过去把 1081 题也干掉。

题目要求总结出来有三点：

要求一、要去重。

要求二、去重字符串中的字符顺序不能打乱  $s$  中字符出现的相对顺序。

要求三、在所有符合上一条要求的去重字符串中，字典序最小的作为最终结果。

上述三条要求中，要求三可能有点难理解，举个例子。

比如说输入字符串  $s = "babc"$ ，去重且符合相对位置的字符串有两个，分别是 " $bac$ " 和 " $abc$ "，但是我们的算法得返回 " $abc$ "，因为它的字典序更小。

按理说，如果我们想要有序的结果，那就得对原字符串排序对吧，但是排序后就不能保证符合  $s$  中字符出现顺序了，这似乎是矛盾的。

其实这里会借鉴前文 [单调栈解题框架](#) 中讲到的「单调栈」的思路，没看过也无妨，等会你就明白了。

我们先暂时忽略要求三，用「栈」来实现一下要求一和要求二，至于为什么用栈来实现，后面你就知道了：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 环检测及拓扑排序算法



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">210. Course Schedule II</a>	<a href="#">210. 课程表 II</a>	困难
<a href="#">207. Course Schedule</a>	<a href="#">207. 课程表</a>	困难

阅读本文前，你需要先学习：

- 图结构基础及通用实现
- 图结构的 DFS/BFS 遍历

tip：本文有视频版：[拓扑排序详解及应用](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

图这种数据结构有一些比较特殊的算法，比如二分图判断，有环图无环图的判断，拓扑排序，以及最经典的最小生成树，单源最短路径问题，更难的就是类似网络流这样的问题。

不过目前来看，像网络流这种问题，你又不是打竞赛的，没时间的话就没必要学了；像 [最小生成树](#) 和 [最短路径问题](#)，虽然从刷题的角度遇到的不多，但它们属于经典算法，学有余力可以掌握一下；像 [二分图判定](#)、[拓扑排序](#)这一类，本质上就是图的遍历，属于比较基本的算法，应该熟练地掌握。

那么本文就结合具体的算法题，来说两个图论算法：有向图的环检测、拓扑排序算法。

这两个算法既可以用 DFS 思路解决，也可以用 BFS 思路解决，相对而言 BFS 解法从代码实现上看更简洁一些，但 DFS 解法有助于你进一步理解递归遍历数据结构的奥义，所以本文中我先讲 DFS 遍历的思路，再讲 BFS 遍历的思路。

## 环检测算法（DFS 版本）

先来看看力扣第 207 题「课程表」：

### ▼ 207. 课程表 Leetcode | 力扣

你这个学期必须选修 `numCourses` 门课程，记为 `0` 到 `numCourses - 1`。

在选修某些课程之前需要一些先修课程。先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]`，表示如果要学习课程 `ai` 则 必须 先学习课程 `bi`。

- 例如，先修课程对 `[0, 1]` 表示：想要学习课程 `0`，你需要先完成课程 `1`。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true`；否则，返回 `false`。

### 示例 1:

输入: numCourses = 2, prerequisites = [[1,0]]

输出: true

解释: 总共有 2 门课程。学习课程 1 之前, 你需要完成课程 0 。这是可能的。

### 示例 2:

输入: numCourses = 2, prerequisites = [[1,0],[0,1]]

输出: false

解释: 总共有 2 门课程。学习课程 1 之前, 你需要先完成课程 0 ; 并且学习课程 0 之前, 你还应先完成课程 1 。这是不可能的。

### 提示:

- $1 \leq \text{numCourses} \leq 2000$
- $0 \leq \text{prerequisites.length} \leq 5000$
- $\text{prerequisites}[i].length == 2$
- $0 \leq a_i, b_i < \text{numCourses}$
- $\text{prerequisites}[i]$  中的所有课程对互不相同

```
// 函数签名如下
boolean canFinish(int numCourses, int[][] prerequisites);
```

题目应该不难理解, 什么时候无法修完所有课程? 当存在循环依赖的时候。

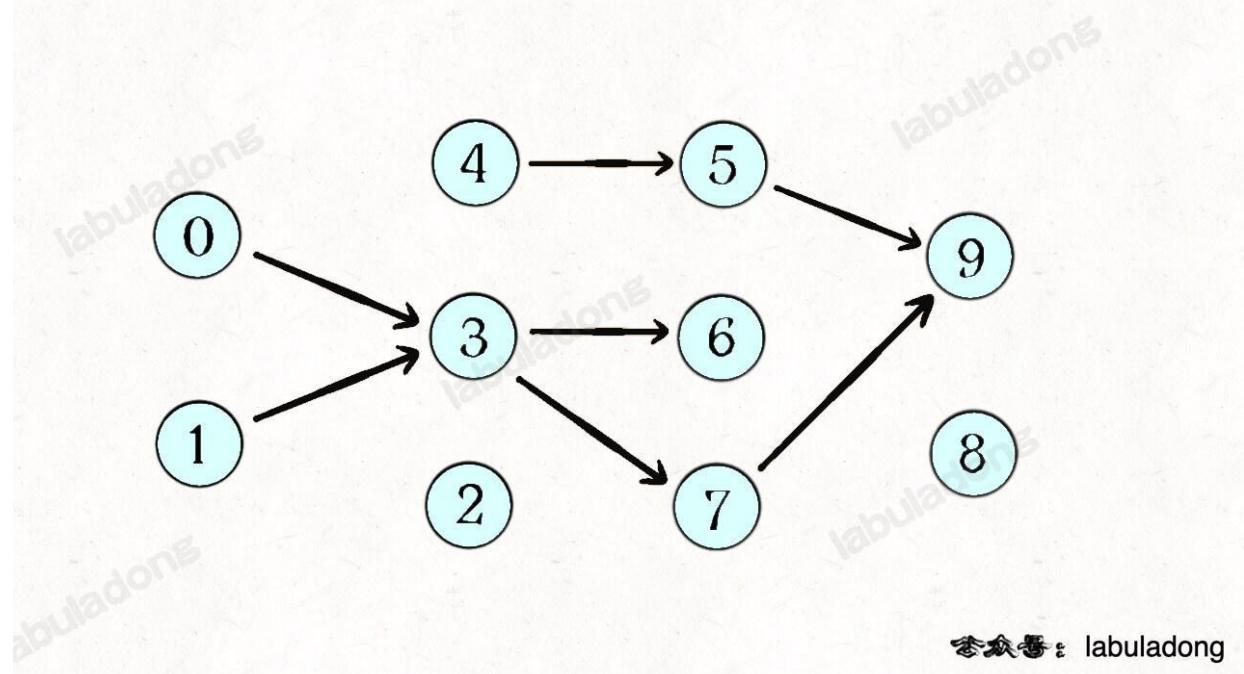
其实这种场景在现实生活中也十分常见, 比如我们写代码 import 包也是一个例子, 必须合理设计代码目录结构, 否则会出现循环依赖, 编译器会报错, 所以编译器实际上也使用了类似算法来判断你的代码是否能够成功编译。

看到依赖问题, 首先想到的就是把问题转化成「有向图」这种数据结构, 只要图中存在环, 那就说明存在循环依赖。

具体来说, 我们首先可以把课程看成「有向图」中的节点, 节点编号分别是  $0, 1, \dots, \text{numCourses}-1$ , 把课程之间的依赖关系看做节点之间的有向边。

比如说必须修完课程 1 才能去修课程 3, 那么就有一条有向边从节点 1 指向 3。

所以我们可以根据题目输入的 `prerequisites` 数组生成一幅类似这样的图:



如果发现这幅有向图中存在环，那就说明课程之间存在循环依赖，肯定没办法全部上完；反之，如果没有环，那么肯定能上完全部课程。

好，那么想解决这个问题，首先我们要把题目的输入转化成一幅有向图，然后再判断图中是否存在环。

如何转换成图呢？我们前文 [图结构的存储](#) 写过图的两种存储形式，邻接矩阵和邻接表。

这里我就用邻接表形式存储图吧，首先可以写一个建图函数：

```
List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
    // 图中共有 numCourses 个节点
    List<Integer>[] graph = new LinkedList[numCourses];
    for (int i = 0; i < numCourses; i++) {
        graph[i] = new LinkedList<>();
    }
    for (int[] edge : prerequisites) {
        int from = edge[1], to = edge[0];
        // 添加一条从 from 指向 to 的有向边
        // 边的方向是「被依赖」关系，即修完课程 from 才能修课程 to
        graph[from].add(to);
    }
    return graph;
}
```

图建出来了，怎么判断图中有没有环呢？

很简单，无非就是想考你如何遍历图中的所有路径嘛，如果我能遍历所有路径，那么路径是否成环不就容易算出来了吗？

[图的 DFS/BFS 遍历基础](#) 写了如何用 DFS 算法遍历图的所有路径，如果忘记了请去复习，下面要用到。

我这里先直接套用遍历所有路径的 DFS 代码模板，用一个 `hasCycle` 变量记录是否存在环，当重复遍历到 `onPath` 中的节点时，就说明遇到了环，设置 `hasCycle = true`。

基于这个思路，先看第一版代码（会超时）：

```

class Solution {
    // 记录递归堆栈中的节点
    boolean[] onPath;
    // 记录图中是否有环
    boolean hasCycle = false;

    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);

        onPath = new boolean[numCourses];

        for (int i = 0; i < numCourses; i++) {
            // 遍历图中的所有节点
            traverse(graph, i);
        }
        // 只要没有循环依赖可以完成所有课程
        return !hasCycle;
    }

    // 图遍历函数，遍历所有路径
    void traverse(List<Integer>[] graph, int s) {
        if (hasCycle) {
            // 如果已经找到了环，也不用再遍历了
            return;
        }

        if (onPath[s]) {
            // s 已经在递归路径上，说明成环了
            hasCycle = true;
            return;
        }

        // 前序代码位置
        onPath[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序代码位置
        onPath[s] = false;
    }

    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 代码见前文
    }
}

```

注意图中并不是所有节点都相连，所以要用一个 for 循环将所有节点都作为起点调用一次 DFS 搜索算法。

其实这个解法已经是正确的了，因为遍历了所有路径，一定可以判定是否成环。但是这个解法无法通过所有测试用例，会超时。那么原因肯定也能猜出来，**有冗余计算呗**。

哪里有冗余计算呢？我举个例子你就明白了。

假设现在你以节点 2 为起点遍历所有可达的路径，最终发现没有环。

假设另一个节点 5 有一条指向 2 的边，你在以 5 为起点遍历所有可达的路径时，肯定还会走到 2，那么请问，此时你是否还需要继续遍历 2 的所有可达路径呢？

答案是不需要了，因为第一次你没找到环，那么这次也不可能找到环。想明白这里面的冗余计算没有？你如果觉得有反例，可以自己画一下，实际上是没有反例的。

那么对症下药就行了：如果我们发现一个节点之前被遍历过，就可以直接跳过，不用再重复遍历了。

优化后的代码如下：

```
class Solution {
    // 记录一次递归堆栈中的节点
    boolean[] onPath;
    // 记录节点是否被遍历过
    boolean[] visited;
    // 记录图中是否有环
    boolean hasCycle = false;

    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);

        onPath = new boolean[numCourses];
        visited = new boolean[numCourses];

        for (int i = 0; i < numCourses; i++) {
            // 遍历图中的所有节点
            traverse(graph, i);
        }
        // 只要没有循环依赖可以完成所有课程
        return !hasCycle;
    }

    // 图遍历函数，遍历所有路径
    void traverse(List<Integer>[] graph, int s) {
        if (hasCycle) {
            // 如果已经找到了环，也不用再遍历了
            return;
        }

        if (onPath[s]) {
            // s 已经在递归路径上，说明成环了
            hasCycle = true;
            return;
        }

        if (visited[s]) {
            // 不用再重复遍历已遍历过的节点
            return;
        }

        if (hasCycle) {
            // 如果已经找到了环，也不用再遍历了
            return;
        }

        // 前序代码位置
        visited[s] = true;
        onPath[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
    }
}
```

```
// 后序代码位置  
onPath[s] = false;  
}  
  
List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {  
    // 代码见前文  
}  
}
```

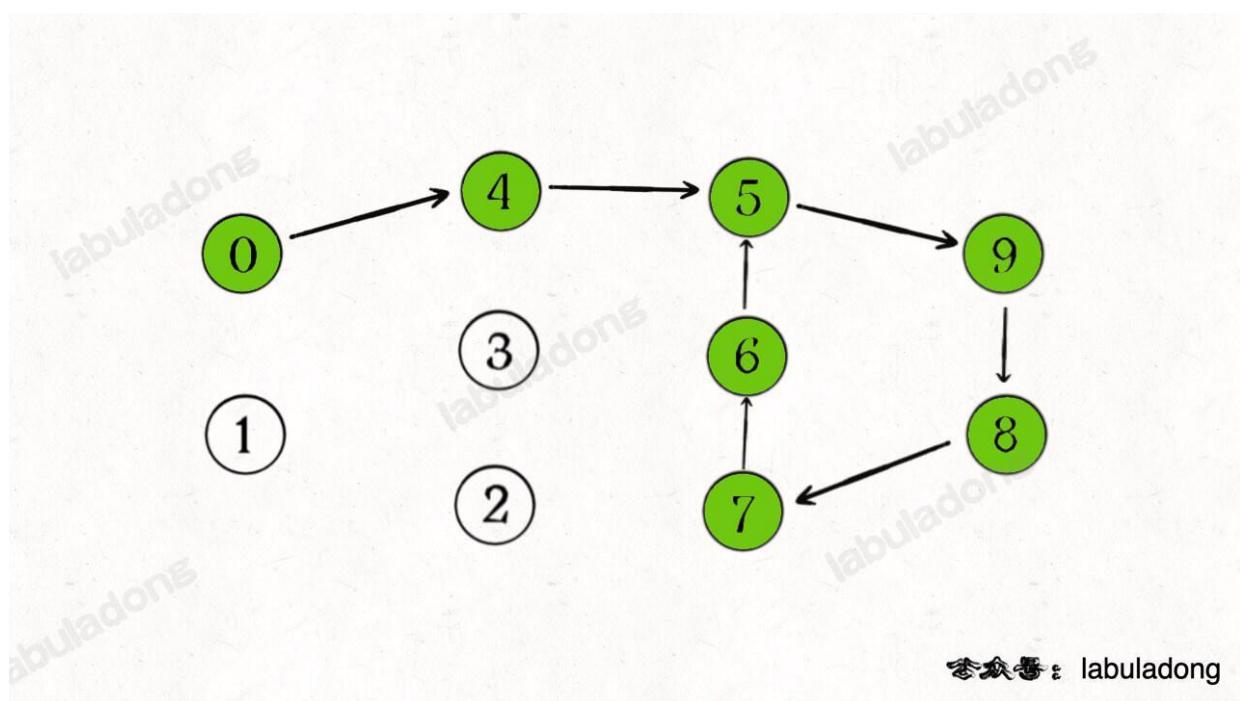
## ▶ 代码可视化动画

这道题就解决了，核心就是判断一幅有向图中是否存在环。

不过如果出题人继续提问，让你不仅要判断是否存在环，还要返回这个环具体有哪些节点，怎么办？

你可能说，`onPath` 里面为 `true` 的索引，不就是组成环的节点编号吗？

不是的，假设从节点 `0` 开始遍历，下图中绿色的节点是递归的路径，它们在 `onPath` 中的值都是 `true`，但显然成环的节点只是其中的一部分：



这个问题大家可以先思考一下，办法肯定有很多啦，我只给出一个常用的解法。

最简单直接的解法是，在 `boolean[] onPath` 数组的基础上，我们再使用一个 `Stack<Integer> path` 栈，把遍历过程中经过的节点顺序也保存下来。

比如按照上图绿色的遍历顺序，`path` 从栈底到栈顶的元素就是 `[0, 4, 5, 9, 8, 7, 6]`。此时又一次遇到了节点 `5`，那么就可以知道 `[5, 9, 8, 7, 6]` 这部分是环了。

接下来，我们来再讲一个经典的图算法：拓扑排序。

## 拓扑排序算法（DFS 版本）

看下力扣第 210 题「课程表 II」：

▼ 210. 课程表 II Leetcode | 力扣

现在你总共有 `numCourses` 门课需要选，记为 `0` 到 `numCourses - 1`。给你一个数组 `prerequisites`，其中 `prerequisites[i] = [ai, bi]`，表示在选修课程 `ai` 前 必须 先选修 `bi`。

- 例如，想要学习课程 `0`，你需要先完成课程 `1`，我们用一个匹配来表示：`[0,1]`。

返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序，你只要返回 任意一种 就可以了。如果不可能完成所有课程，返回 一个空数组。

示例 1：

```
输入: numCourses = 2, prerequisites = [[1,0]]  
输出: [0,1]  
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1]。
```

示例 2：

```
输入: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]  
输出: [0,2,1,3]  
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。  
因此，一个正确的课程顺序是 [0,1,2,3]。另一个正确的排序是 [0,2,1,3]。
```

示例 3：

```
输入: numCourses = 1, prerequisites = []  
输出: [0]
```

提示：

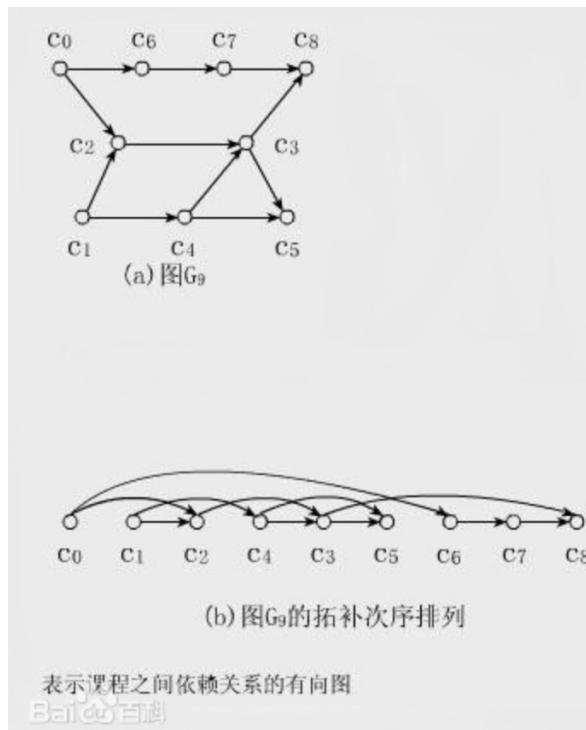
- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= numCourses * (numCourses - 1)`
- `prerequisites[i].length == 2`
- `0 <= ai, bi < numCourses`
- `ai != bi`
- 所有 `[ai, bi]` 互不相同

这道题就是上道题的进阶版，不是仅仅让你判断是否可以完成所有课程，而是进一步让你返回一个合理的上课顺序，保证开始修每个课程时，前置的课程都已经修完。

函数签名如下：

```
int[] findOrder(int numCourses, int[][] prerequisites);
```

这里我先说一下拓扑排序 (Topological Sorting) 这个名词，网上搜出来的定义很数学，这里干脆用百度百科的一幅图来让你直观地感受下：



直观地说就是，让你把一幅图「拉平」，而且这个「拉平」的图里面，所有箭头方向都是一致的，比如上图所有箭头都是朝右的。

很显然，如果一幅有向图中存在环，是无法进行拓扑排序的，因为肯定做不到所有箭头方向一致；反过来，如果一幅图是「有向无环图」，那么一定可以进行拓扑排序。

但是我们这道题和拓扑排序有什么关系呢？

其实也不难看出来，如果把课程抽象成节点，课程之间的依赖关系抽象成有向边，那么这幅图的拓扑排序结果就是上课顺序。

首先，我们先判断一下题目输入的课程依赖是否成环，成环的话是无法进行拓扑排序的，所以我们可以复用上一道题的主函数：

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    if (!canFinish(numCourses, prerequisites)) {
        // 不可能完成所有课程
        return new int[]{};
    }
    // ...
}
```

那么关键问题来了，如何进行拓扑排序？是不是又要秀什么高大上的技巧了？

**其实特别简单，把图结构后序遍历的结果进行反转，就是拓扑排序的结果。**

有的读者提到，他在网上看到的拓扑排序算法就是后序遍历结果，不用对后序遍历结果进行反转，这是为什么呢？

你确实可以看到这样的解法，原因是建图的时候对边的定义和我不同。我建的图中箭头方向是「被依赖」关系，比如节点 1 指向 2，含义是节点 1 被节点 2 依赖，即做完 1 才能去做 2，因为这样更符合我们的直觉。

如果你反过来，把有向边定义为「依赖」关系，那么整幅图中边全部反转，就可以不对后序遍历结果反转。具体来说，就是把我的解法代码中 `graph[from].add(to);` 改成 `graph[to].add(from);` 就可以不反转了。

直接看解法代码吧，在上一题环检测的代码基础上添加了记录后序遍历结果的逻辑：

```

class Solution {
    // 记录后序遍历结果
    List<Integer> postorder = new ArrayList<>();
    // 记录是否存在环
    boolean hasCycle = false;
    boolean[] visited, onPath;

    // 主函数
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);
        visited = new boolean[numCourses];
        onPath = new boolean[numCourses];
        // 遍历图
        for (int i = 0; i < numCourses; i++) {
            traverse(graph, i);
        }
        // 有环图无法进行拓扑排序
        if (hasCycle) {
            return new int[]{};
        }
        // 逆后序遍历结果即为拓扑排序结果
        Collections.reverse(postorder);
        int[] res = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            res[i] = postorder.get(i);
        }
        return res;
    }

    // 图遍历函数
    void traverse(List<Integer>[] graph, int s) {
        if (onPath[s]) {
            // 发现环
            hasCycle = true;
        }
        if (visited[s] || hasCycle) {
            return;
        }
        // 前序遍历位置
        onPath[s] = true;
        visited[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序遍历位置
        postorder.add(s);
        onPath[s] = false;
    }

    // 建图函数
    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 代码见前文
    }
}

```

▶ 彩虹 代码可视化动画

代码虽然看起来多，但是逻辑应该是很清楚的，只要图中无环，那么我们就调用 `traverse` 函数对图进行 DFS 遍历，记录后序遍历结果，最后把后序遍历结果反转，作为最终的答案。

那么为什么后序遍历的反转结果就是拓扑排序呢？

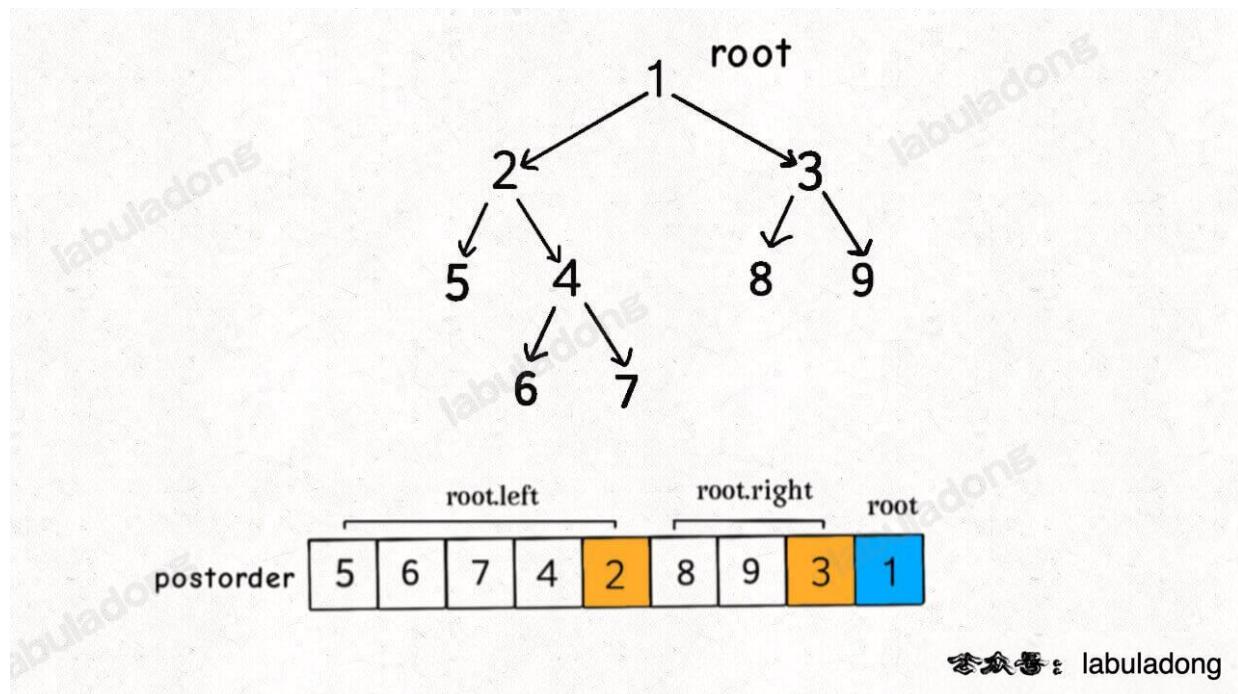
我这里也避免数学证明，用一个直观的例子来解释，我们就说二叉树，这是我们说过很多次的二叉树遍历框架：

```
void traverse(TreeNode root) {
    // 前序遍历代码位置
    traverse(root.left)
    // 中序遍历代码位置
    traverse(root.right)
    // 后序遍历代码位置
}
```

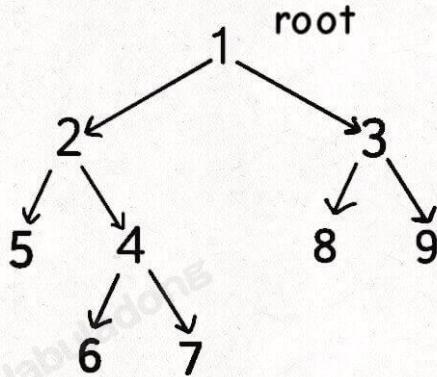
二叉树的后序遍历是什么时候？遍历完左右子树之后才会执行后序遍历位置的代码。换句话说，当左右子树的节点都被装到结果列表里面了，根节点才会被装进去。

后序遍历的这一特点很重要，之所以拓扑排序的基础是后序遍历，是因为一个任务必须等到它依赖的所有任务都完成之后才能开始执行。

你把二叉树理解成一幅有向图，边的方向是由父节点指向子节点，那么就是下图这样：



对于标准的后序遍历结果，根节点出现在最后，只要把遍历结果反过来，就是拓扑排序结果：



reversePostorder



© labuladong

我知道有读者会问，后序遍历结果反转，和前序遍历结果有什么关系？

对于二叉树来说你看起来好像有关系，实际上二者没有任何关系。你千万不要认为后序遍历反转的结果等同于前序遍历结果。

它俩的关键区别在 [二叉树思想（纲领篇）](#) 已经讲过了，后序位置的代码是等到左右子树都遍历完才执行的，只有它才能体现出「依赖」关系，其他遍历顺序都做不到。

## 环检测算法 (BFS 版本)

刚才讲了用 DFS 算法利用 `onPath` 数组判断是否存在环；也讲了用 DFS 算法利用逆后序遍历进行拓扑排序。

其实 BFS 算法借助 `indegree` 数组记录每个节点的「入度」，也可以实现这两个算法。不熟悉 BFS 算法的读者可阅读后文 [BFS 算法核心框架](#)。

所谓「出度」和「入度」是「有向图」中的概念，很直观：如果一个节点 `x` 有 `a` 条边指向别的节点，同时被 `b` 条边所指，则称节点 `x` 的出度为 `a`，入度为 `b`。

先说环检测算法，直接看 BFS 的解法代码：

```

class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // 建图，有向边代表「被依赖」关系
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);
        // 构建入度数组
        int[] indegree = new int[numCourses];
        for (int[] edge : prerequisites) {
            int from = edge[1], to = edge[0];
            // 节点 to 的入度加一
            indegree[to]++;
        }

        // 根据入度初始化队列中的节点
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                q.add(i);
            }
        }
    }
}
  
```

```
// 节点 i 没有入度，即没有依赖的节点
// 可以作为拓扑排序的起点，加入队列
q.offer(i);
}

// 记录遍历的节点个数
int count = 0;
// 开始执行 BFS 循环
while (!q.isEmpty()) {
    // 弹出节点 cur，并将它指向的节点的入度减一
    int cur = q.poll();
    count++;
    for (int next : graph[cur]) {
        indegree[next]--;
        if (indegree[next] == 0) {
            // 如果入度变为 0，说明 next 依赖的节点都已被遍历
            q.offer(next);
        }
    }
}

// 如果所有节点都被遍历过，说明不成环
return count == numCourses;
}

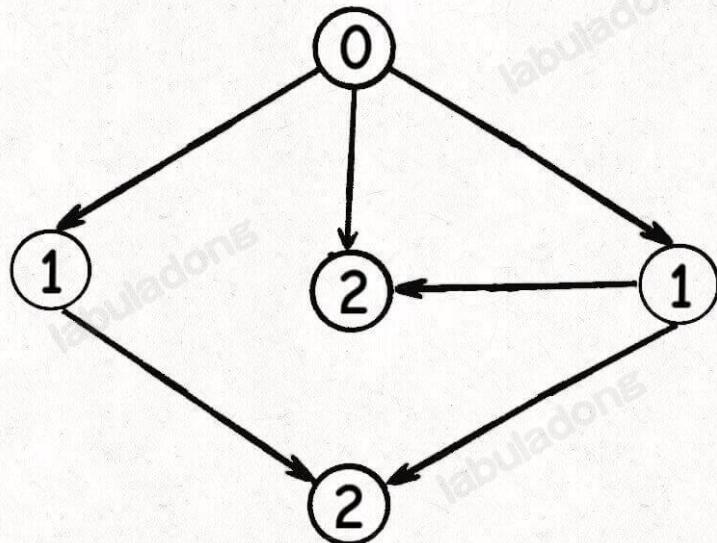
// 建图函数
List<Integer>[] buildGraph(int n, int[][] edges) {
    // 见前文
}
```

我先总结下这段 BFS 算法的思路：

- 1、构建邻接表，和之前一样，边的方向表示「被依赖」关系。
- 2、构建一个 `indegree` 数组记录每个节点的入度，即 `indegree[i]` 记录节点 `i` 的入度。
- 3、对 BFS 队列进行初始化，将入度为 0 的节点首先装入队列。
- 4、开始执行 BFS 循环，不断弹出队列中的节点，减少相邻节点的入度，并将入度变为 0 的节点加入队列。
- 5、如果最终所有节点都被遍历过（`count` 等于节点数），则说明不存在环，反之则说明存在环。

我画个图你就容易理解了，比如下面这幅图，节点中的数字代表该节点的入度：

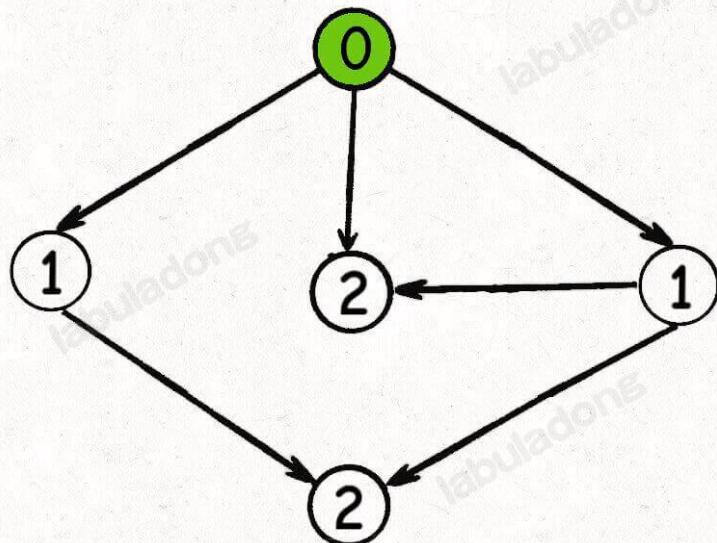
- 未入队
- 队列中
- 已离队



© labuladong

队列进行初始化后，入度为 0 的节点首先被加入队列：

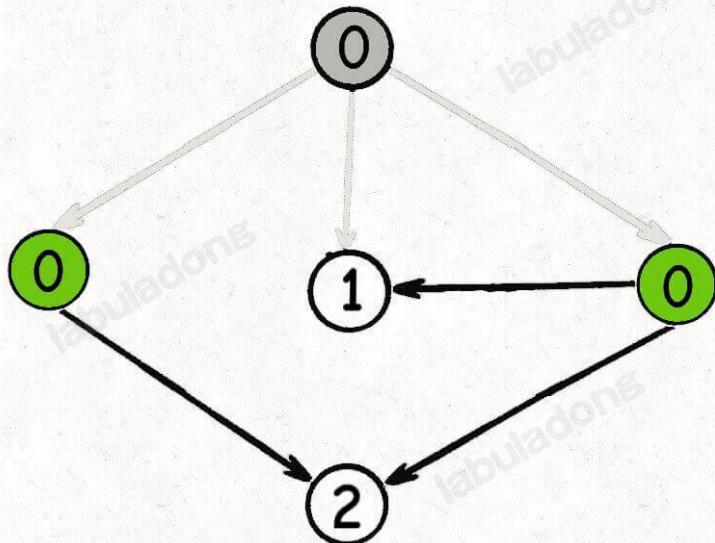
- 未入队
- 队列中
- 已离队



© labuladong

开始执行 BFS 循环，从队列中弹出一个节点，减少相邻节点的入度，同时将新产生的入度为 0 的节点加入队列：

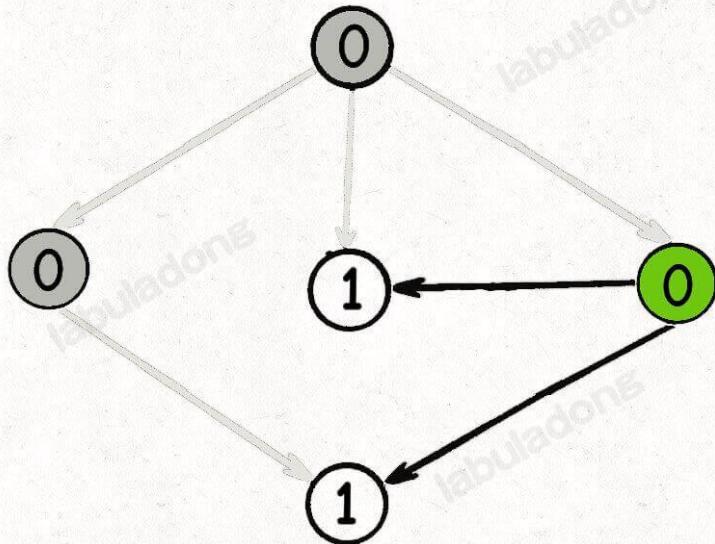
- 未入队
- 队列中
- 已离队



© labuladong

继续从队列弹出节点，并减少相邻节点的入度，这一次没有新产生的入度为 0 的节点：

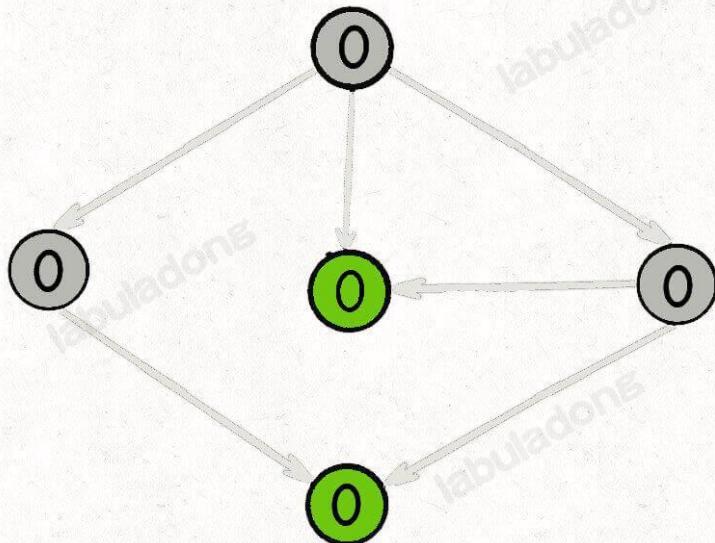
- 未入队
- 队列中
- 已离队



© labuladong

继续从队列弹出节点，并减少相邻节点的入度，同时将新产生的入度为 0 的节点加入队列：

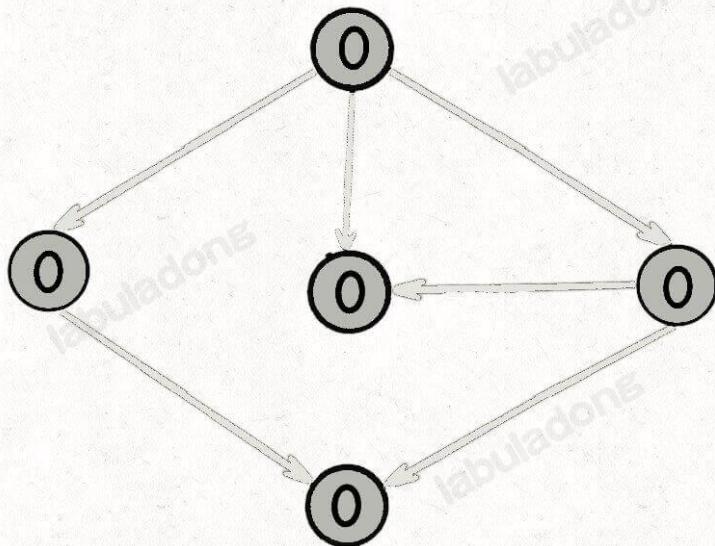
- 未入队
- 队列中
- 已离队



公众号：labuladong

继续弹出节点，直到队列为空：

- 未入队
- 队列中
- 已离队

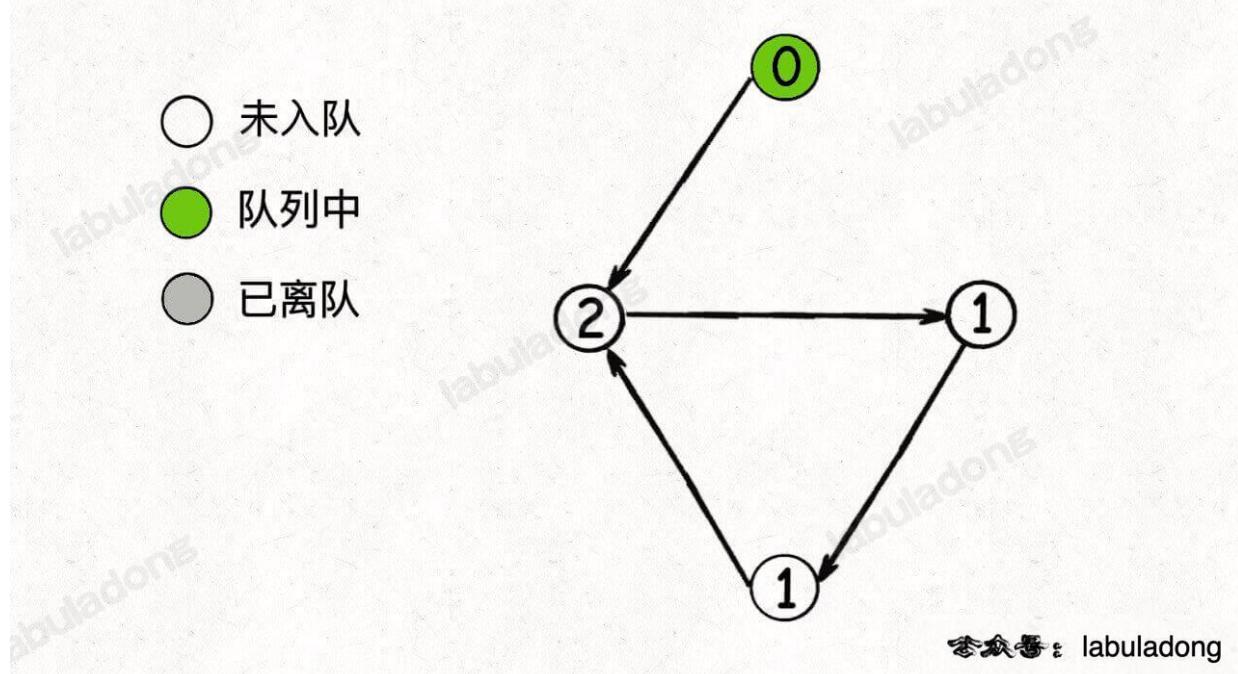


公众号：labuladong

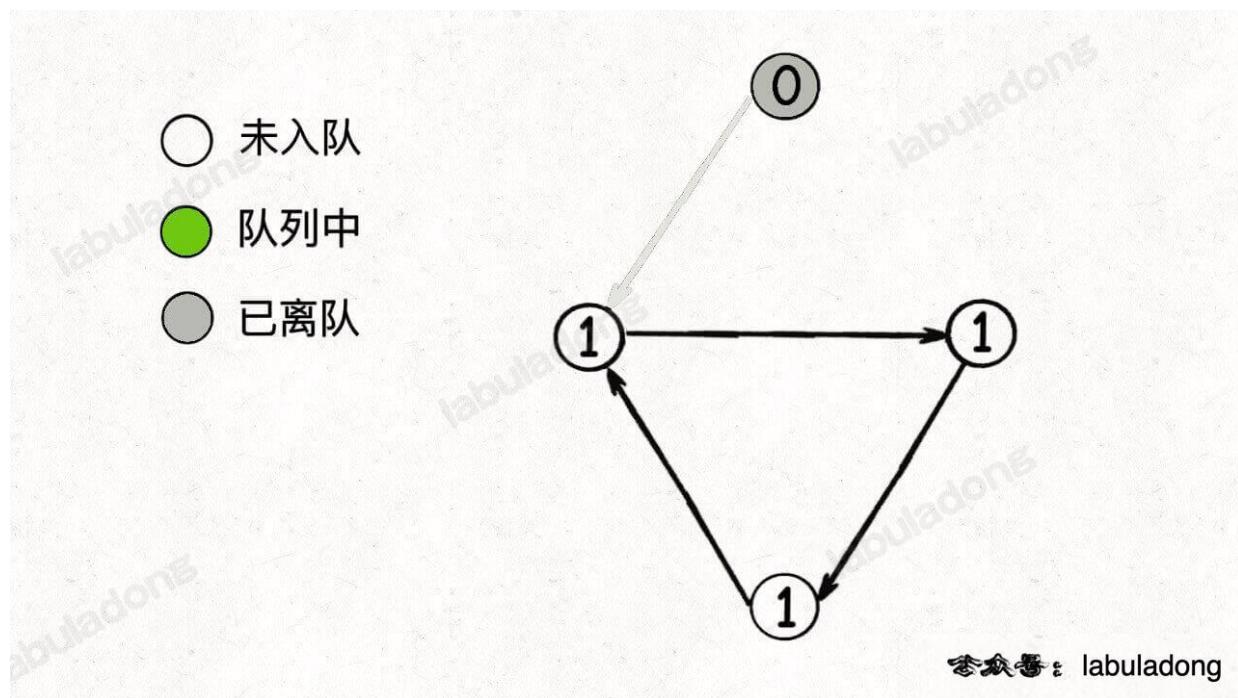
这时候，所有节点都被遍历过一遍，也就说明图中不存在环。

反过来说，如果按照上述逻辑执行 BFS 算法，存在节点没有被遍历，则说明成环。

比如下面这种情况，队列中最初只有一个入度为 0 的节点：



当弹出这个节点并减小相邻节点的入度之后队列为空，但并没有产生新的入度为 0 的节点加入队列，所以 BFS 算法终止：

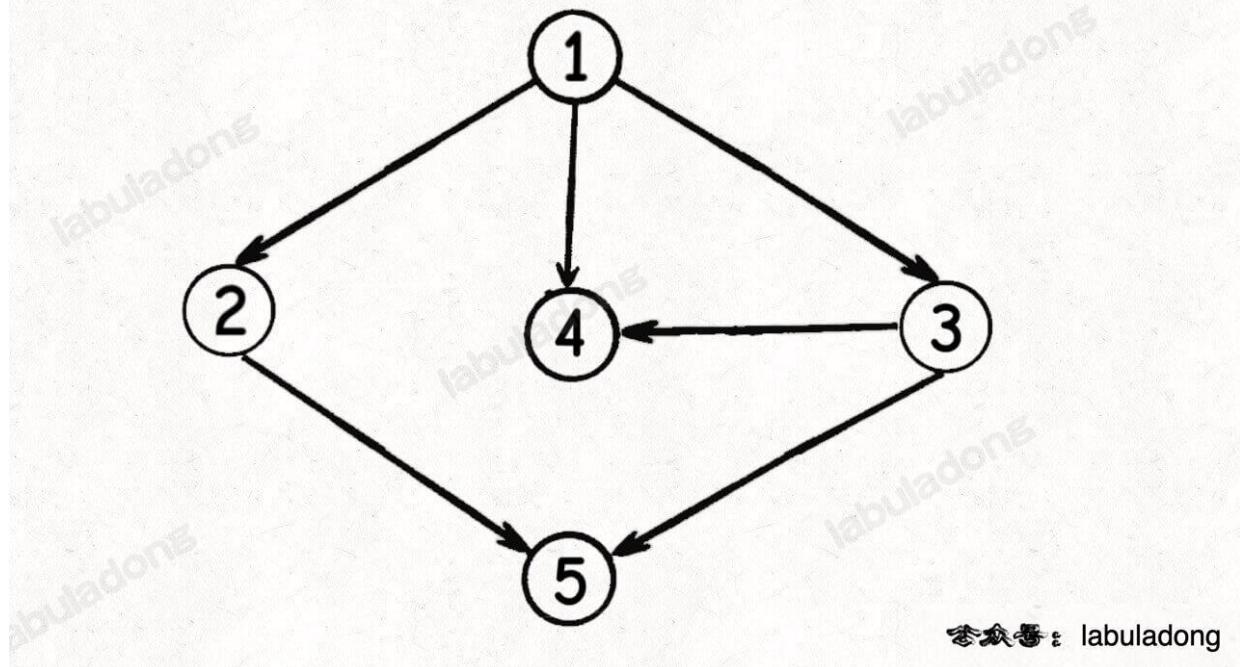


你看到了，如果存在节点没有被遍历，那么说明图中存在环，现在回头去看 BFS 的代码，你应该就很容易理解其中的逻辑了。

## 拓扑排序算法（BFS 版本）

如果你能看懂 BFS 版本的环检测算法，那么就很容易得到 BFS 版本的拓扑排序算法，因为节点的遍历顺序就是拓扑排序的结果。

比如刚才举的第一个例子，下图每个节点中的值即入队的顺序：



© labuladong

显然，这个顺序就是一个可行的拓扑排序结果。

所以，我们稍微修改一下 BFS 版本的环检测算法，记录节点的遍历顺序即可得到拓扑排序的结果：

```
class Solution {

    public int[] findOrder(int numCourses, int[][] prerequisites) {
        // 建图，和环检测算法相同
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);
        // 计算入度，和环检测算法相同
        int[] indegree = new int[numCourses];
        for (int[] edge : prerequisites) {
            int from = edge[1], to = edge[0];
            indegree[to]++;
        }

        // 根据入度初始化队列中的节点，和环检测算法相同
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                q.offer(i);
            }
        }

        // 记录拓扑排序结果
        int[] res = new int[numCourses];
        // 记录遍历节点的顺序（索引）
        int count = 0;
        // 开始执行 BFS 算法
        while (!q.isEmpty()) {
            int cur = q.poll();
            // 弹出节点的顺序即为拓扑排序结果
            res[count] = cur;
            count++;
            for (int next : graph[cur]) {
                indegree[next]--;
                if (indegree[next] == 0) {
```

```
        q.offer(next);
    }
}

if (count != numCourses) {
    // 存在环，拓扑排序不存在
    return new int[] {};
}

return res;
}

// 建图函数
List<Integer>[] buildGraph(int n, int[][] edges) {
    // 见前文
}
}
```

按道理，[图的遍历](#)都需要 `visited` 数组防止走回头路，这里的 BFS 算法其实是通过 `indegree` 数组实现的 `visited` 数组的作用，只有入度为 0 的节点才能入队，从而保证不会出现死循环。

好了，到这里环检测算法、拓扑排序算法的 BFS 实现也讲完了，继续留一个思考题：

对于 BFS 的环检测算法，如果问你形成环的节点具体是哪些，你应该如何实现呢？

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">310. Minimum Height Trees</a>	<a href="#">310. 最小高度树</a>	
-	<a href="#">剑指 Offer II 113. 课程顺序</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 众里寻他千百度：名流问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode

力扣

难度

[277. Find the Celebrity](#) 🔒

[277. 搜索名人](#) 🔒



阅读本文前，你需要先学习：

- 图结构基础及通用实现

今天来讨论经典的「名流问题」：

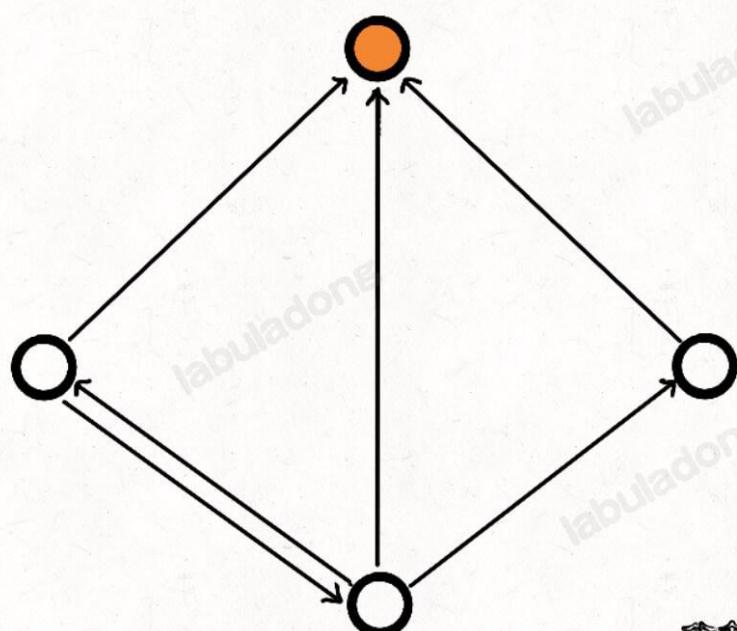
给你  $n$  个人的社交关系（你知道任意两个人之间是否认识），然后请你找出这些人中的「名人」。

所谓「名人」有两个条件：

- 1、所有其他人都认识「名人」。
- 2、「名人」不认识任何其他人。

这是一个图相关的算法问题，社交关系嘛，本质上就可以抽象成一幅图。

如果把每个人看做图中的节点，「认识」这种关系看做是节点之间的有向边，那么名人就是这幅图中一个特殊的节点：



李公孙：labuladong

这个节点没有一条指向其他节点的有向边；且其他所有节点都有一条指向这个节点的有向边。

或者说的专业一点，名人节点的出度为 0，入度为  $n - 1$ 。

那么，这  $n$  个人的社交关系是如何表示的呢？

前文 [图论算法基础](#) 说过，图有两种存储形式，一种是邻接表，一种是邻接矩阵，邻接表的主要优势是节约存储空间；邻接矩阵的主要优势是可以迅速判断两个节点是否相邻。

对于名人问题，显然会经常需要判断两个人之间是否认识，也就是两个节点是否相邻，所以我们可以用邻接矩阵来表示人和人之间的社交关系。

那么，把名流问题描述成算法的形式就是这样的：

给你输入一个大小为  $n \times n$  的二维数组（邻接矩阵）`graph` 表示一幅有  $n$  个节点的图，每个人都是图中的一个节点，编号为 0 到  $n - 1$ 。

如果 `graph[i][j] == 1` 代表第  $i$  个人认识第  $j$  个人，如果 `graph[i][j] == 0` 代表第  $i$  个人不认识第  $j$  个人。

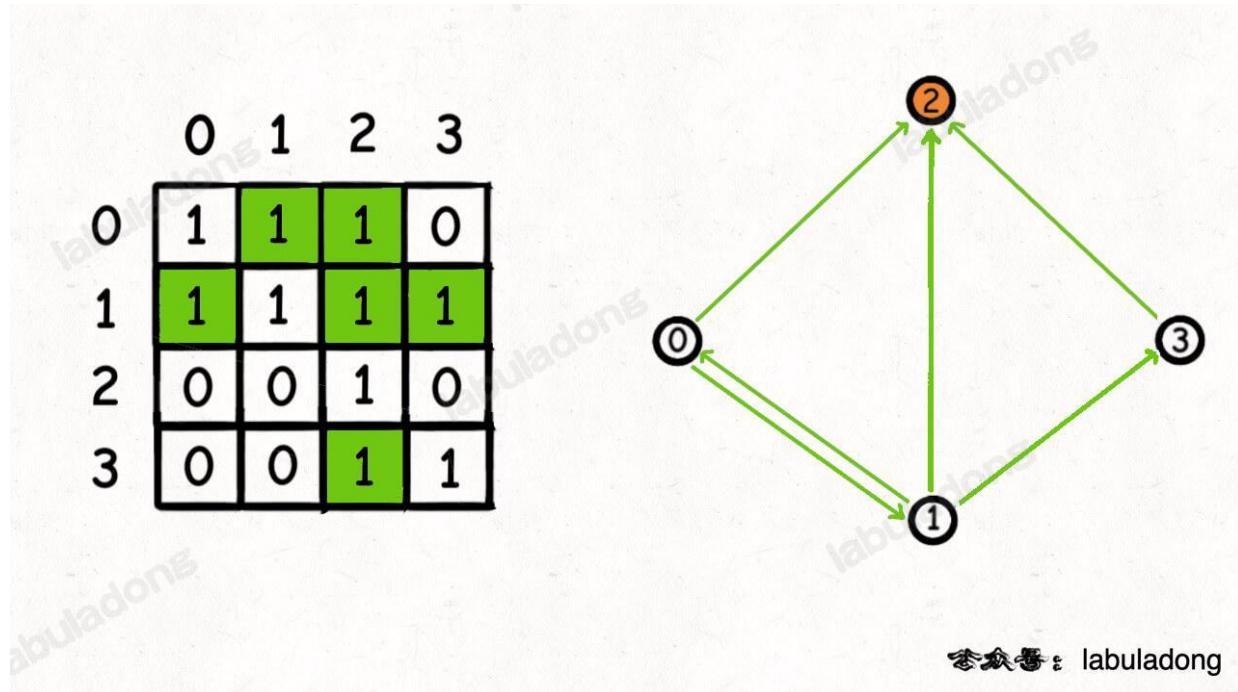
有了这幅图表示人与人之间的关系，请你计算，这  $n$  个人中，是否存在「名人」？

如果存在，算法返回这个名人的编号，如果不存在，算法返回 -1。

函数签名如下：

```
int findCelebrity(int[][] graph);
```

比如输入的邻接矩阵长这样：



那么算法应该返回 2。

力扣第 277 题「搜寻名人」就是这个经典问题，不过并不是直接把邻接矩阵传给你，而是只告诉你总人数  $n$ ，同时提供一个 API `knows` 来查询人和人之间的社交关系：

```
// 可以直接调用，能够返回 i 是否认识 j
boolean knows(int i, int j);

// 请你实现：返回「名人」的编号
int findCelebrity(int n) {
    // todo
}
```

很明显，`knows` API 本质上还是在访问邻接矩阵。为了简单起见，我们后面就按力扣的题目形式来探讨一下这个经典问题。

## 暴力解法

我们拍拍脑袋就能写出一个简单粗暴的算法：

```
class Solution extends Relation {
    public int findCelebrity(int n) {
        for (int cand = 0; cand < n; cand++) {
            int other;
            for (other = 0; other < n; other++) {
                if (cand == other) continue;
                // 保证其他人都认识 cand，且 cand 不认识任何其他人
                // 否则 cand 就不可能是名人
                if (knows(cand, other) || !knows(other, cand)) {
                    break;
                }
            }
            if (other == n) {
                // 找到名人
                return cand;
            }
        }
        // 没有一个人符合名人特性
        return -1;
    }
}
```

`cand` 是候选人（candidate）的缩写，我们的暴力算法就是从头开始穷举，把每个人都视为候选人，判断是否符合「名人」的条件。

刚才也说了，`knows` 函数底层就是在访问一个二维的邻接矩阵，一次调用的时间复杂度是  $O(1)$ ，所以这个暴力解法整体的最坏时间复杂度是  $O(N^2)$ 。

那么，是否有其他高明的办法来优化时间复杂度呢？其实是有优化空间的，你想想，我们现在最耗时的地方在哪里？

对于每一个候选人 `cand`，我们都要用一个内层 `for` 循环去判断这个 `cand` 到底符不符合「名人」的条件。

这个内层 `for` 循环看起来就蠢，虽然判断一个人「是名人」必须用一个 `for` 循环，但判断一个人「不是名人」就不用这么麻烦了。

因为「名人」的定义保证了「名人」的唯一性，所以我们可以利用排除法，先排除那些显然不是「名人」的人，从而避免 `for` 循环的嵌套，降低时间复杂度。

## 优化解法

我再重复一遍所谓「名人」的定义：

- 1、所有其他人都认识名人。
- 2、名人不认识任何其他人。

这个定义就很有意思，它保证了人群中最多有一个名人。

这很好理解，如果有两个人同时是名人，那么这两条定义就自相矛盾了。

换句话说，只要观察任意两个候选人的关系，我一定能确定其中的一个人不是名人，把他排除。

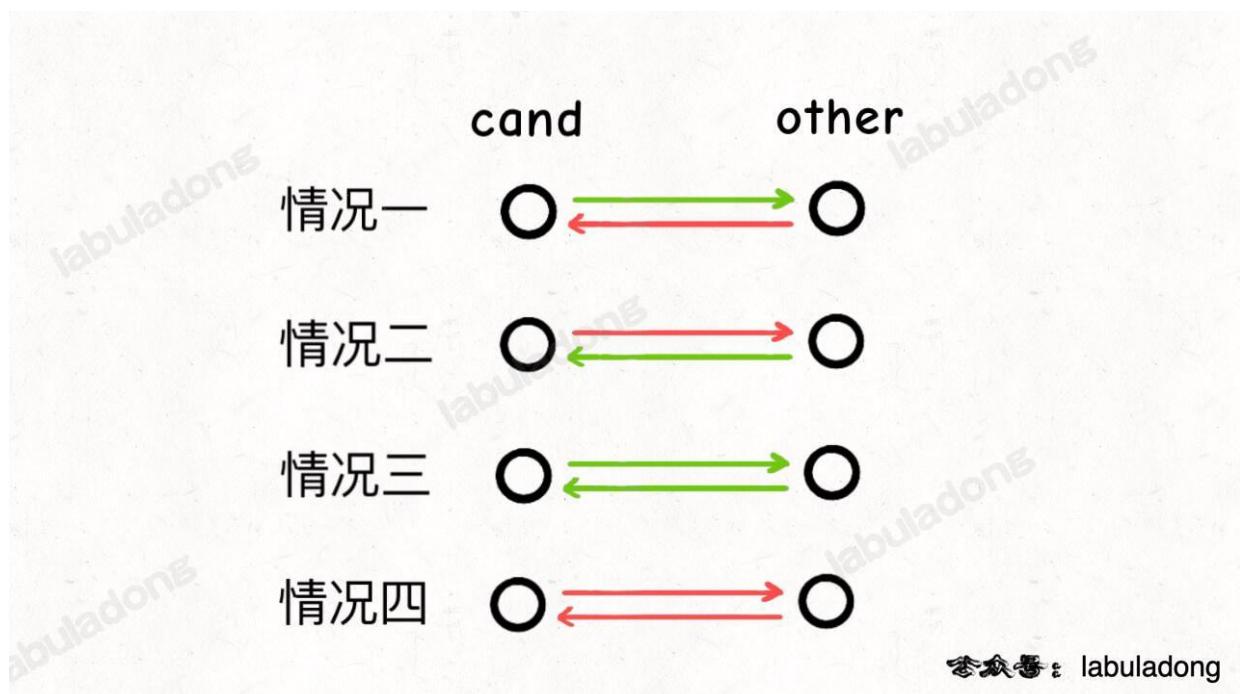
至于另一个候选人是不是名人，只看两个人的关系肯定是不能确定的，但这不重要，重要的是排除掉一个必然不是名人的候选人，缩小了包围圈。

这是优化的核心，也是比较难理解的，所以我们先来说说为什么观察任意两个候选人的关系，就能排除掉一个。

你想想，两个人之间的关系可能是什么样的？

无非就是四种：你认识我不认识你，我认识你你不认识我，咱俩互相认识，咱俩互相不认识。

如果把人比作节点，红色的有向边表示不认识，绿色的有向边表示认识，那么两个人的关系无非是如下四种情况：



不妨认为这两个人的编号分别是 **cand** 和 **other**，然后我们逐一分析每种情况，看看怎么排除掉一个人。

对于情况一，**cand** 认识 **other**，所以 **cand** 肯定不是名人，排除。因为名人不可能认识别人。

对于情况二，**other** 认识 **cand**，所以 **other** 肯定不是名人，排除。

对于情况三，他俩互相认识，肯定都不是名人，可以随便排除一个。

对于情况四，他俩互不认识，肯定都不是名人，可以随便排除一个。因为名人应该被所有其他人认识。

综上，只要观察任意两个之间的关系，就至少能确定一个人不是名人，上述情况判断可以用如下代码表示：

```
if (knows(cand, other) || !knows(other, cand)) {  
    // cand 不可能是名人  
} else {
```

```
// other 不可能是名人
}
```

如果能够理解这一个特点，那么写出优化解法就简单了。

我们可以不断从候选人中选两个出来，然后排除掉一个，直到最后只剩下一个候选人，这时候再使用一个 `for` 循环判断这个候选人是否是货真价实的「名人」。

这个思路的完整代码如下：

```
class Solution extends Relation {
    public int findCelebrity(int n) {
        if (n == 1) return 0;
        // 将所有候选人装进队列
        LinkedList<Integer> q = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            q.addLast(i);
        }
        // 一直排除，直到只剩下一个候选人停止循环
        while (q.size() >= 2) {
            // 每次取出两个候选人，排除一个
            int cand = q.removeFirst();
            int other = q.removeFirst();
            if (knows(cand, other) || !knows(other, cand)) {
                // cand 不可能是名人，排除，让 other 归队
                q.addFirst(other);
            } else {
                // other 不可能是名人，排除，让 cand 归队
                q.addFirst(cand);
            }
        }
        // 现在排除得只剩一个候选人，判断他是否真的是名人
        int cand = q.removeFirst();
        for (int other = 0; other < n; other++) {
            if (other == cand) {
                continue;
            }
            // 保证其他人都认识 cand，且 cand 不认识任何其他人
            if (!knows(other, cand) || knows(cand, other)) {
                return -1;
            }
        }
        // cand 是名人
        return cand;
    }
}
```

这个算法避免了嵌套 `for` 循环，时间复杂度降为  $O(N)$  了，不过引入了一个队列来存储候选人集合，使用了  $O(N)$  的空间复杂度。

`LinkedList` 的作用只是充当一个容器把候选人装起来，每次找出两个进行比较和淘汰，但至于具体找出哪两个，都是无所谓的，也就是说候选人归队的顺序无所谓，我们用的是 `addFirst` 只是方便后续的优化，你完全可以用 `addLast`，结果都是一样的。

是否可以进一步优化，把空间复杂度也优化掉？

## 最终解法

如果你能够理解上面的优化解法，其实可以不需要额外的空间解决这个问题，代码如下：

```
class Solution extends Relation {
    public int findCelebrity(int n) {
        int cand = 0;
        for (int other = 1; other < n; other++) {
            if (!knows(other, cand) || knows(cand, other)) {
                // cand 不可能是名人，排除
                // 假设 other 是名人
                cand = other;
            } else {
                // other 不可能是名人，排除
                // 什么都不用做，继续假设 cand 是名人
            }
        }

        // 现在的 cand 是排除的最后结果，但不能保证一定是名人
        for (int other = 0; other < n; other++) {
            if (cand == other) continue;
            // 需要保证其他人都认识 cand，且 cand 不认识任何其他人
            if (!knows(other, cand) || knows(cand, other)) {
                return -1;
            }
        }

        return cand;
    }
}
```

我们之前的解法用到了 `LinkedList` 充当一个队列，用于存储候选人集合，而这个优化解法利用 `other` 和 `cand` 的交替变化，模拟了我们之前操作队列的过程，避免了使用额外的存储空间。

现在，解决名人问题的解法时间复杂度为  $O(N)$ ，空间复杂度为  $O(1)$ ，已经是最优解法了。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 二分图判定算法



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
-	剑指 Offer II 106. 二分图	简单
<a href="#">785. Is Graph Bipartite?</a>	<a href="#">785. 判断二分图</a>	简单
<a href="#">886. Possible Bipartition</a>	<a href="#">886. 可能的二分法</a>	简单

阅读本文前，你需要先学习：

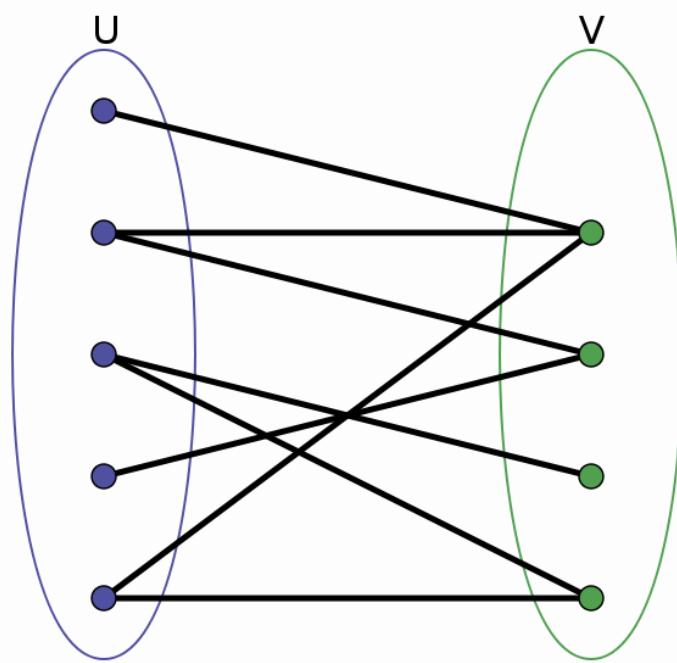
- 图结构基础及通用实现
- 图结构的 DFS/BFS 遍历

tip：本文有视频版：[二分图判定算法及应用](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

今天来讲一个经典图论算法：二分图判定。

### 二分图简介

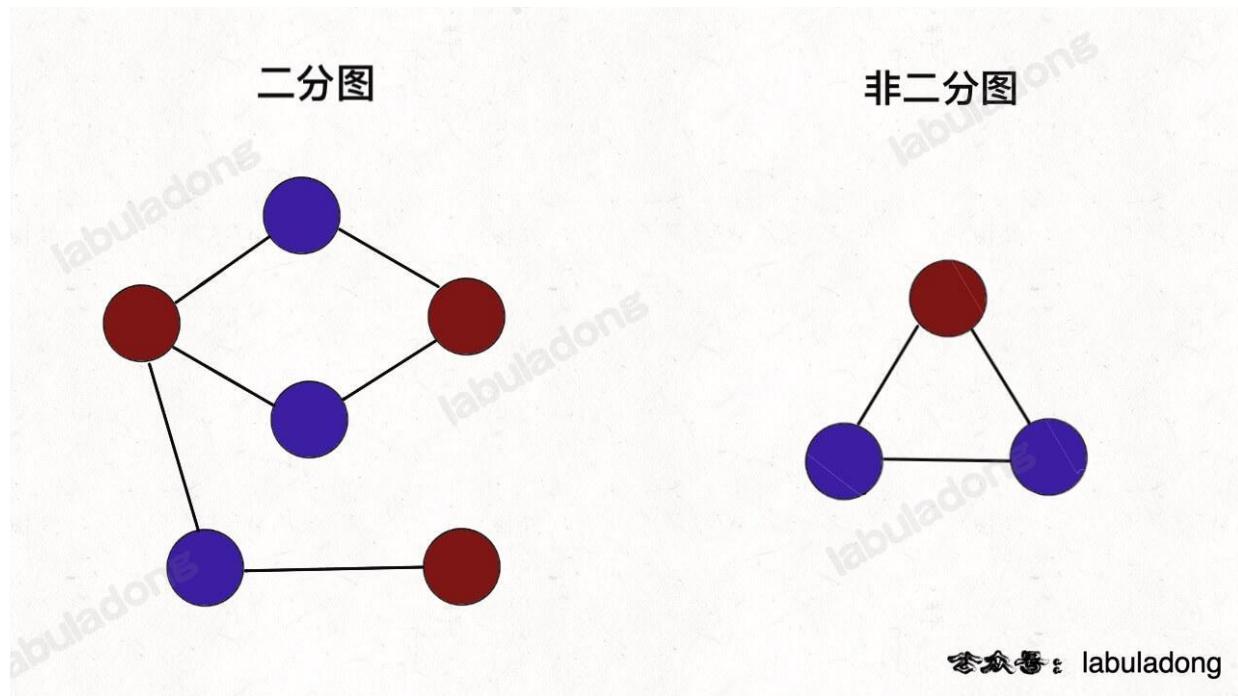
二分图的顶点集可分割为两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，且两个子集内的顶点不相邻。



其实图论里面很多术语的定义都比较拗口，不容易理解。我们甭看这个死板的定义了，来玩个游戏吧：

给你一幅「图」，请你用两种颜色将图中的所有顶点着色，且使得任意一条边的两个端点的颜色都不相同，你能做到吗？

这就是图的「双色问题」，其实这个问题就等同于二分图的判定问题，如果你能够成功地将图染色，那么这幅图就是一幅二分图，反之则不是：



在具体讲解二分图判定算法之前，我们先来说说计算机大佬们闲着无聊解决双色问题的目的是什么。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Union-Find 并查集算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">130. Surrounded Regions</a>	<a href="#">130. 被围绕的区域</a>	
<a href="#">990. Satisfiability of Equality Equations</a>	<a href="#">990. 等式方程的可满足性</a>	
<a href="#">323. Number of Connected Components in an Undirected Graph</a>	<a href="#">323. 无向图中连通分量的数目</a>	
<a href="#">684. Redundant Connection</a>	<a href="#">684. 冗余连接</a>	

阅读本文前，你需要先学习：

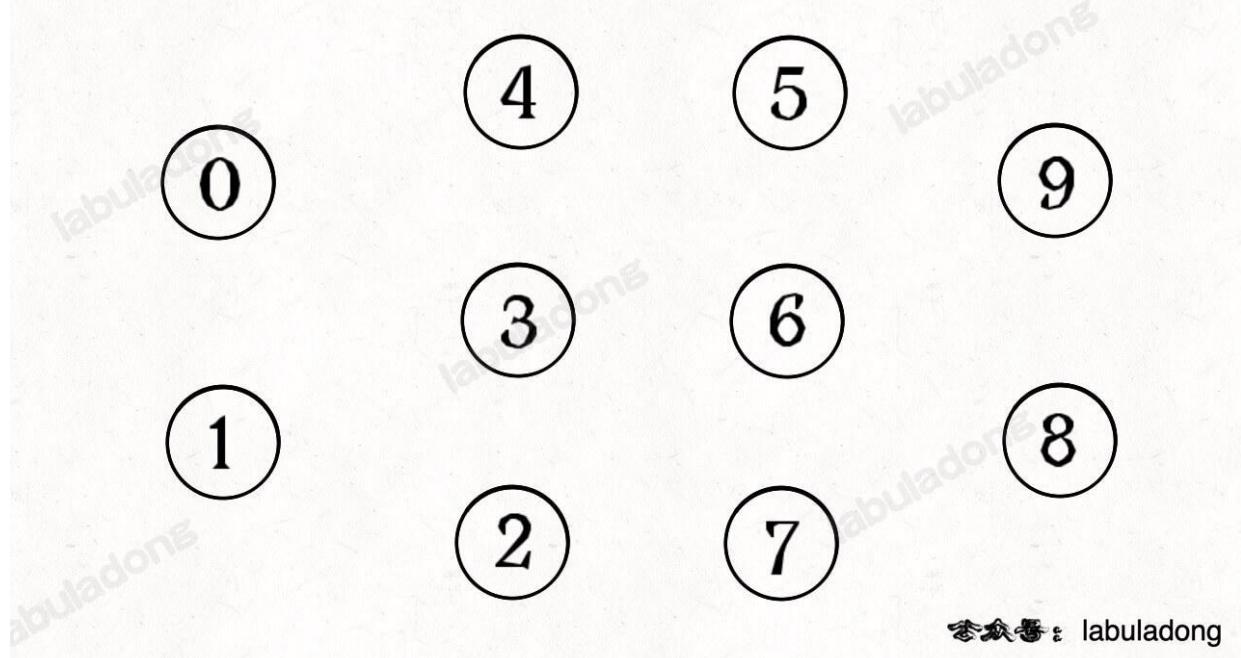
- [多叉树基础及遍历](#)
- [图结构基础及通用实现](#)

并查集（Union-Find）算法是一个专门针对「动态连通性」的算法，我之前写过两次，因为这个算法的考察频率高，而且它也是最小生成树算法的前置知识，所以我整合了本文，争取一篇文章把这个算法讲明白。

首先，从什么是图的动态连通性开始讲。

## 一、动态连通性

简单说，动态连通性其实可以抽象成给一幅图连线。比如下面这幅图，总共有 10 个节点，他们互不相连，分别用 0~9 标记：



© labuladong

现在我们的 Union-Find 算法主要需要实现这两个 API:

```
class UF {  
    // 将 p 和 q 连接  
    public void union(int p, int q);  
    // 判断 p 和 q 是否连通  
    public boolean connected(int p, int q);  
    // 返回图中有多少个连通分量  
    public int count();  
}
```

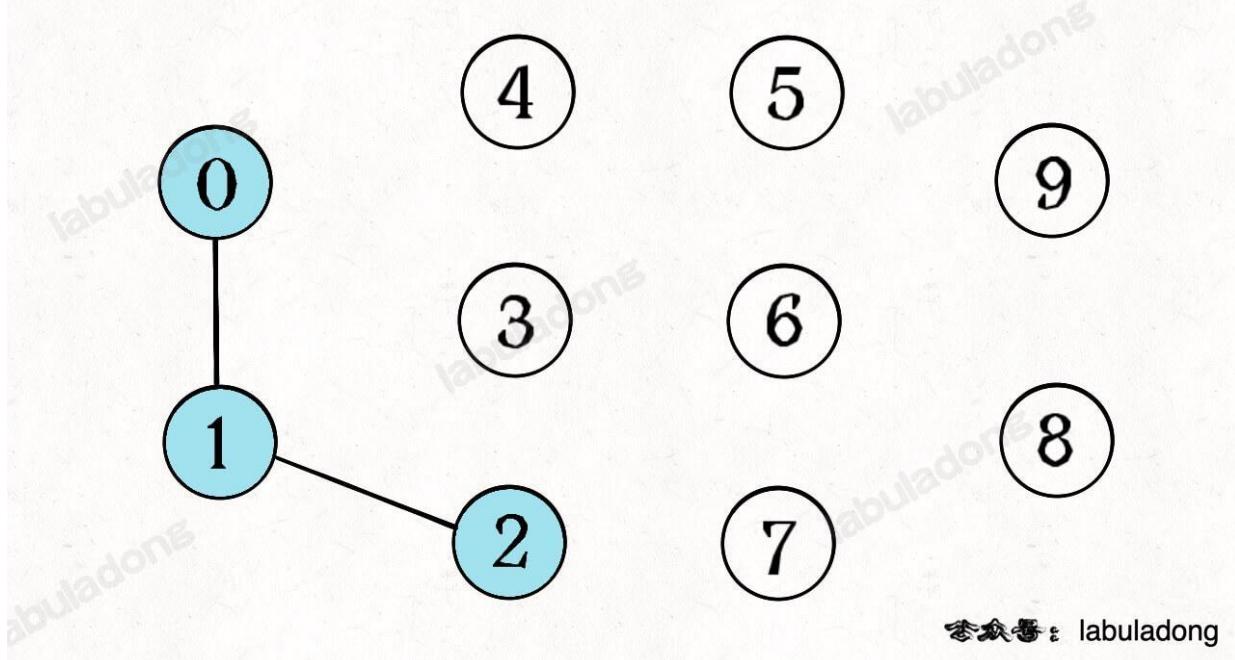
这里所说的「连通」是一种等价关系，也就是说具有如下三个性质：

- 1、自反性：节点  $p$  和  $p$  是连通的。
- 2、对称性：如果节点  $p$  和  $q$  连通，那么  $q$  和  $p$  也连通。
- 3、传递性：如果节点  $p$  和  $q$  连通， $q$  和  $r$  连通，那么  $p$  和  $r$  也连通。

比如说之前那幅图，0~9 任意两个不同的点都不连通，调用 `connected` 都会返回 false，连通分量为 10 个。

如果现在调用 `union(0, 1)`，那么 0 和 1 被连通，连通分量降为 9 个。

再调用 `union(1, 2)`，这时 0,1,2 都被连通，调用 `connected(0, 2)` 也会返回 true，连通分量变为 8 个。



© labuladong

判断这种「等价关系」非常实用，比如说编译器判断同一个变量的不同引用，比如社交网络中的朋友圈计算等等。

这样，你应该大概明白什么是动态连通性了，Union-Find 算法的关键就在于 `union` 和 `connected` 函数的效率。那么用什么模型来表示这幅图的连通状态呢？用什么数据结构来实现代码呢？

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】并查集经典习题

阅读本文前，你需要先学习：

- 并查集（Union Find）算法详解

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Kruskal 最小生成树算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1584. Min Cost to Connect All Points</a>	<a href="#">1584. 连接所有点的最小费用</a>	
<a href="#">1135. Connecting Cities With Minimum Cost</a>	<a href="#">1135. 最低成本联通所有城市</a>	
<a href="#">261. Graph Valid Tree</a>	<a href="#">261. 以图判树</a>	

-----

阅读本文前，你需要先学习：

- 图结构基础及通用实现
- 图结构的 DFS/BFS 遍历
- Union-Find 并查集算法

图论中知名度比较高的算法应该就是 Dijkstra 最短路径算法，环检测和拓扑排序，二分图判定算法 以及今天要讲的最小生成树（Minimum Spanning Tree）算法了。

最小生成树算法主要有 Prim 算法（普里姆算法）和 Kruskal 算法（克鲁斯卡尔算法）两种，这两种算法虽然都运用了贪心思想，但从实现上来说差异还是蛮大的。

本文先来讲比较简单易懂的 Kruskal 算法，然后在下一篇文章 [Prim 算法模板](#) 中聊 Prim 算法。

Kruskal 算法其实很容易理解和记忆，其关键是要熟悉并查集算法，如果不熟悉，建议先看下前文 [Union-Find 并查集算法](#)。

接下来，我们从最小生成树的定义说起。

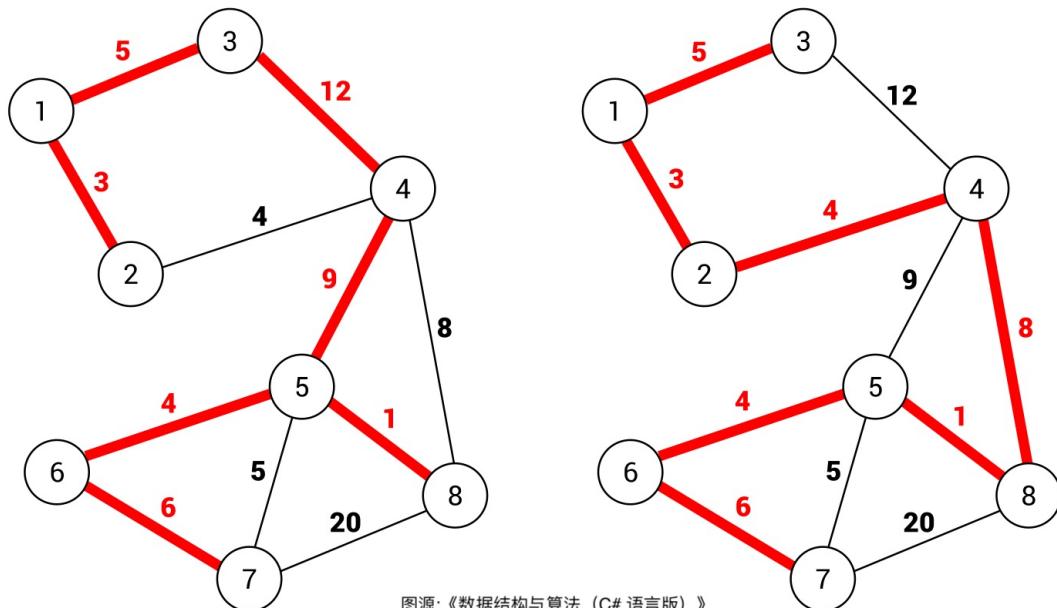
## 什么是最小生成树

先说「树」和「图」的根本区别：树不会包含环，图可以包含环。

如果一幅图没有环，完全可以拉伸成一棵树的模样。说的专业一点，树就是「无环连通图」。

那么什么是图的「生成树」呢，其实按字面意思也好理解，就是在图中找一棵包含图中的所有节点的树。专业点说，生成树是含有图中所有顶点的「无环连通子图」。

容易想到，一幅图可以有很多不同的生成树，比如下面这幅图，红色的边就组成了两棵不同的生成树：



图源:《数据结构与算法 (C# 语言版)》

对于加权图，每条边都有权重，所以每棵生成树都有一个权重和。比如上图，右侧生成树的权重和显然比左侧生成树的权重和要小。

那么最小生成树很好理解了，所有可能的生成树中，权重和最小的那棵生成树就叫「最小生成树」。

一般来说，我们都是在无向加权图中计算最小生成树的，所以使用最小生成树算法的现实场景中，图的边权重一般代表成本、距离这样的标量。

在讲 Kruskal 算法之前，需要回顾一下 Union-Find 并查集算法。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# Prim 最小生成树算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1584. Min Cost to Connect All Points</a>	1584. 连接所有点的最小费用	
<a href="#">1135. Connecting Cities With Minimum Cost</a>	1135. 最低成本联通所有城市	

阅读本文前，你需要先学习：

- 图结构基础及通用实现
- 图结构的 DFS/BFS 遍历
- Kruskal 最小生成树算法

本文要介绍的 Prim 算法和前文介绍的 Kruskal 算法都是经典的最小生成树算法，阅读本文之前，希望你读过前文 [Kruskal 最小生成树算法](#)，了解最小生成树的基本定义以及 Kruskal 算法的基本原理，这样就能很容易理解 Prim 算法逻辑了。

## 对比 Kruskal 算法

图论的最小生成树问题，就是让你从图中找若干边形成一个边的集合  $mst$ ，这些边有以下特性：

- 1、这些边组成的是一棵树（树和图的区别在于不能包含环）。
- 2、这些边形成的树要包含所有节点。
- 3、这些边的权重之和要尽可能小。

那么 Kruskal 算法是使用什么逻辑满足上述条件，计算最小生成树的呢？

首先，Kruskal 算法用到了贪心思想，来满足权重之和尽可能小的问题：

先对所有边按照权重从小到大排序，从权重最小的边开始，选择合适的边加入  $mst$  集合，这样挑出来的边组成的树就是权重和最小的。

在挑选边的时候是有讲究的，如果一条边的两个节点已经是连通的，则这条边会使  $mst$  集合中出现环；如果最后的连通分量总数大于 1，则说明形成的是多棵树（森林）而不是一棵最小生成树。

所以，Kruskal 算法用到了 [Union-Find 并查集算法](#)，来保证挑选出来的这些边组成的一定是一棵「树」，而不会包含环或者形成一片「森林」。

那么，本文的主角 Prim 算法是使用什么逻辑来计算最小生成树的呢？

首先，Prim 算法也使用贪心思想来让生成树的权重尽可能小，也就是「切分定理」，这个后文会详细解释。

其次，Prim 算法使用 **BFS 算法思想** 和 **visited** 布尔数组避免成环，来保证选出来的边最终形成的一定是一棵树。

Prim 算法不需要事先对所有边排序，而是利用优先级队列动态实现排序的效果，所以我觉得 Prim 算法类似于 Kruskal 的动态过程。

下面介绍一下 Prim 算法的核心原理：切分定理。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# Dijkstra 算法模板及应用



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">743. Network Delay Time</a>	743. 网络延迟时间	
<a href="#">1514. Path with Maximum Probability</a>	1514. 概率最大的路径	
<a href="#">1631. Path With Minimum Effort</a>	1631. 最小体力消耗路径	

阅读本文前，你需要先学习：

- 图结构基础及通用实现
- 图结构的 DFS/BFS 遍历

其实，很多算法的底层原理异常简单，无非就是一步一步延伸，变得看起来好像特别复杂，特别牛逼。

但如果你看过历史文章，应该可以对算法形成自己的理解，就会发现很多算法都是换汤不换药，毫无新意，非常枯燥。

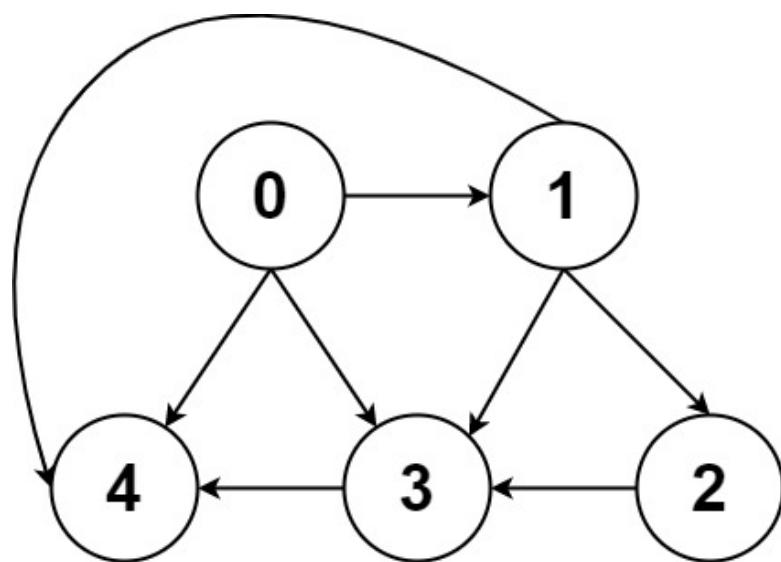
比如，[二叉树心法（总纲）](#) 中说二叉树非常重要，你把这个结构掌握了，就会发现[动态规划](#)，[分治算法](#)，[回溯（DFS）算法](#)，[BFS 算法框架](#)，[Union-Find 并查集算法](#)，[二叉堆实现优先级队列](#) 就是把二叉树翻来覆去的运用。

那么本文又要告诉你，Dijkstra 算法（一般音译成迪杰斯特拉算法）无非就是一个 BFS 算法的加强版，它们都是从二叉树的层序遍历衍生出来的。

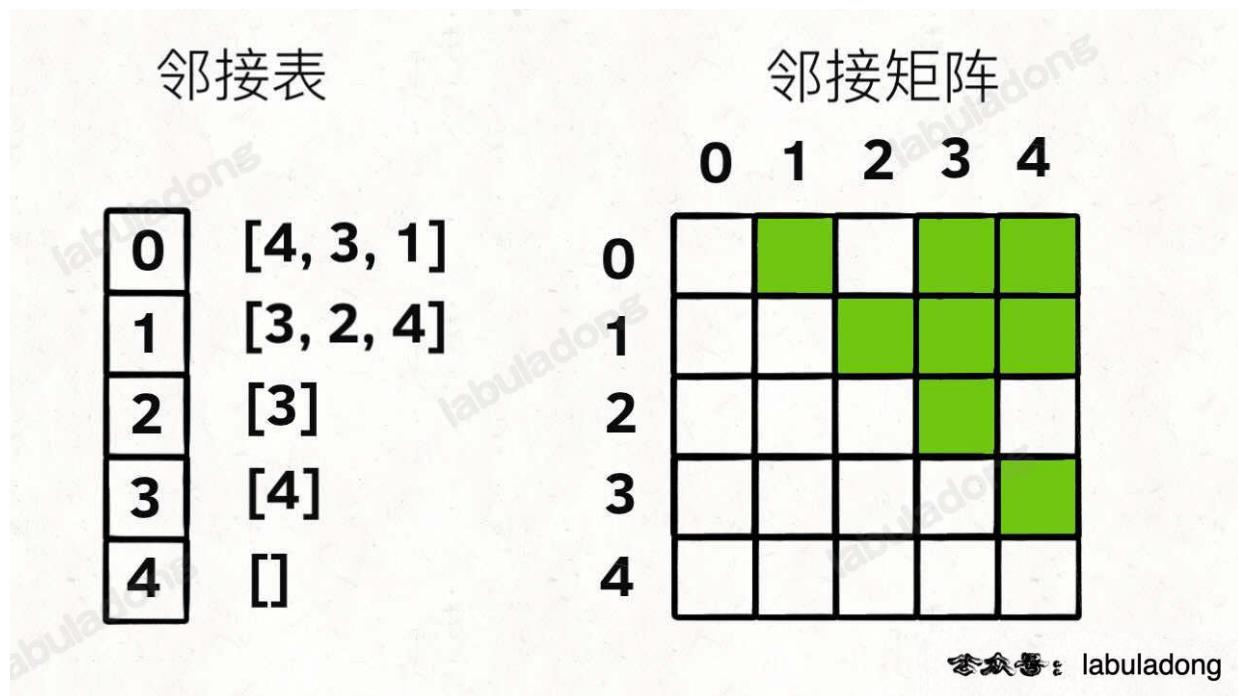
下面我们由浅入深，从二叉树的层序遍历聊到 Dijkstra 算法，给出 Dijkstra 算法的代码框架，顺手秒杀几道运用 Dijkstra 算法的题目。

## 图的抽象

前文[图论第一期：遍历基础](#)说过「图」这种数据结构的基本实现，图中的节点一般就抽象成一个数字（索引），图的具体实现一般是「邻接矩阵」或者「邻接表」。



比如上图这幅图用邻接表和邻接矩阵的存储方式如下：



前文 [图论第二期：拓扑排序](#) 告诉你，我们用邻接表的场景更多，结合上图，一幅图可以用如下 Java 代码表示：

```
// graph[s] 存储节点 s 指向的节点（出度）
List<Integer>[] graph;
```

如果你想把一个问题抽象成「图」的问题，那么首先要实现一个 API `adj`：

```
// 输入节点 s 返回 s 的相邻节点
List<Integer> adj(int s);
```

类似多叉树节点中的 `children` 字段记录当前节点的所有子节点，`adj(s)` 就是计算一个节点 `s` 的相邻节点。

比如上面说的用邻接表表示「图」的方式，`adj` 函数就可以这样表示：

```
List<Integer>[] graph;

// 输入节点 s, 返回 s 的相邻节点
List<Integer> adj(int s) {
    return graph[s];
}
```

当然，对于「加权图」，我们需要知道两个节点之间的边权重是多少，所以还可以抽象出一个 `weight` 方法：

```
// 返回节点 from 到节点 to 之间的边的权重
int weight(int from, int to);
```

这个 `weight` 方法可以根据实际情况而定，因为不同的算法题，题目给的「权重」含义可能不一样，我们存储权重的方式也不一样。

有了上述基础知识，就可以搞定 Dijkstra 算法了，下面我给你从二叉树的层序遍历开始推演出 Dijkstra 算法的实现。

## 二叉树层级遍历和 BFS 算法

我们之前说过二叉树的层级遍历框架：

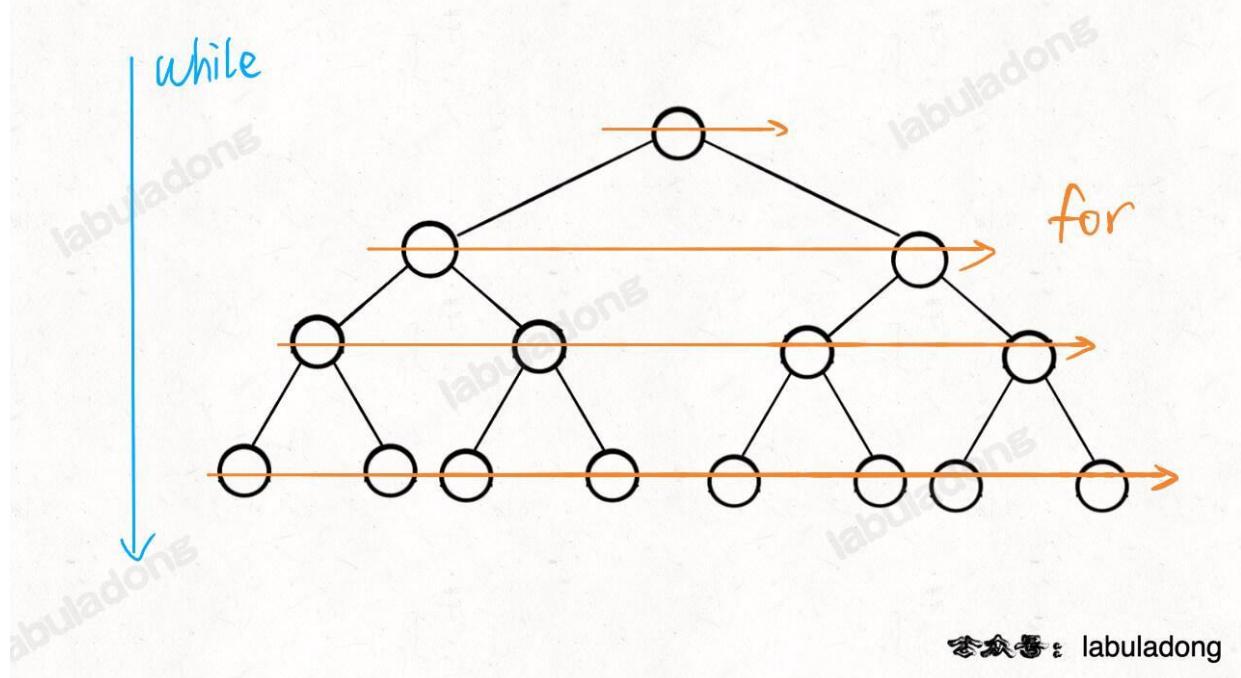
```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    int depth = 1;
    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            printf("节点 %s 在第 %s 层", cur, depth);

            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
        depth++;
    }
}
```

我们先来思考一个问题，注意二叉树的层级遍历 `while` 循环里面还套了个 `for` 循环，为什么要这样？

`while` 循环和 `for` 循环的配合正是这个遍历框架设计的巧妙之处：



© labuladong

**while** 循环控制一层一层往下走，**for** 循环利用 **sz** 变量控制从左到右遍历每一层二叉树节点。

注意我们代码框架中的 **depth** 变量，其实就记录了当前遍历到的层数。换句话说，每当我们遍历到一个节点 **cur**，都知道这个节点属于第几层。

算法题经常会问二叉树的最大深度呀，最小深度呀，层序遍历结果呀，等等问题，所以记录下来这个深度 **depth** 是有必要 的。

基于二叉树的遍历框架，我们又可以扩展出多叉树的层序遍历框架：

```
// 输入一棵多叉树的根节点，层序遍历这棵多叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    int depth = 1;
    // 从上到下遍历多叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            printf("节点 %s 在第 %s 层", cur, depth);

            // 将下一层节点放入队列
            for (TreeNode child : cur.children) {
                q.offer(child);
            }
        }
        depth++;
    }
}
```

基于多叉树的遍历框架，我们又可以扩展出 BFS（广度优先搜索）的算法框架：

```

// 输入起点，进行 BFS 搜索
int BFS(Node start) {
    // 核心数据结构
    Queue<Node> q;
    // 避免走回头路
    Set<Node> visited;

    // 将起点加入队列
    q.offer(start);
    visited.add(start);

    // 记录搜索的步数
    int step = 0;
    while (q not empty) {
        int sz = q.size();
        // 将当前队列中的所有节点向四周扩散一步
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            printf("从 %s 到 %s 的最短距离是 %s", start, cur, step);

            // 将 cur 的相邻节点加入队列
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
        step++;
    }
}

```

如果对 BFS 算法不熟悉，可以看后文 [BFS 算法框架](#)，这里只是为了让你做个对比，所谓 BFS 算法，就是把算法问题抽象成一幅「无权图」，然后继续玩二叉树层级遍历那一套罢了。

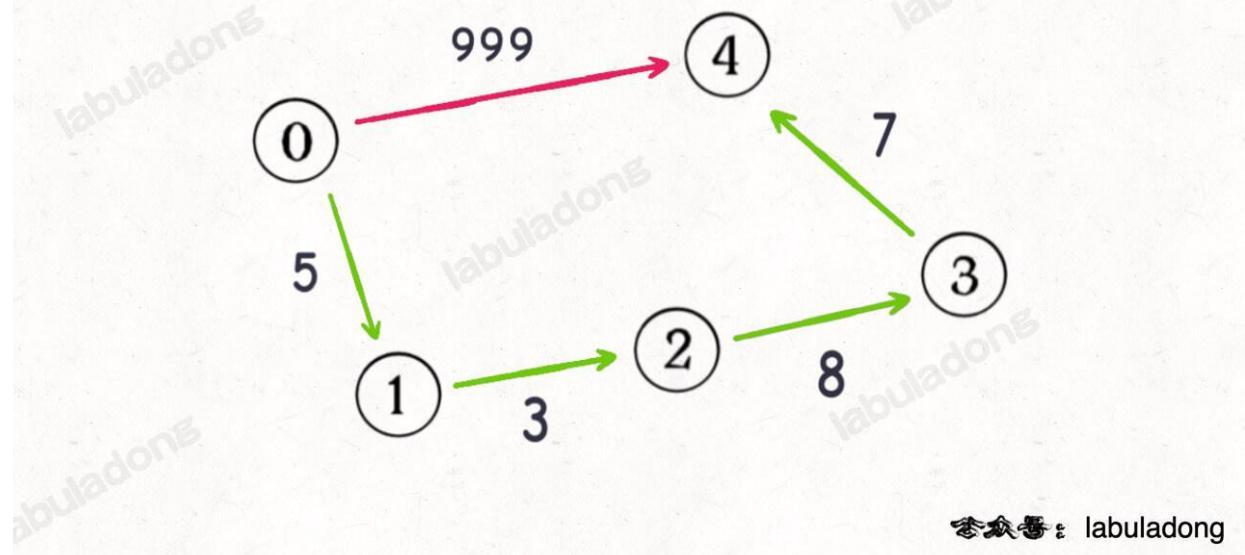
注意，我们的 BFS 算法框架也是 `while` 循环嵌套 `for` 循环的形式，也用了一个 `step` 变量记录 `for` 循环执行的次数，而非就是多用了一个 `visited` 集合记录走过的节点，防止走回头路罢了。

为什么这样呢？

所谓「无权图」，与其说每条「边」没有权重，不如说每条「边」的权重都是 1，从起点 `start` 到任意一个节点之间的路径权重就是它们之间「边」的条数，那可不就是 `step` 变量记录的值么？

再加上 BFS 算法利用 `for` 循环一层一层向外扩散的逻辑和 `visited` 集合防止走回头路的逻辑，当你每次从队列中拿出节点 `cur` 的时候，从 `start` 到 `cur` 的最短权重就是 `step` 记录的步数。

但是，到了「加权图」的场景，事情就没有这么简单了，因为你不能默认每条边的「权重」都是 1 了，这个权重可以是任意正数（Dijkstra 算法要求不能存在负权重边），比如下图的例子：



如果沿用 BFS 算法中的 `step` 变量记录「步数」，显然红色路径一步就可以走到终点，但是这一步的权重很大；正确的最小权重路径应该是绿色的路径，虽然需要走很多步，但是路径权重依然很小。

其实 Dijkstra 和 BFS 算法差不多，不过在讲解 Dijkstra 算法框架之前，我们首先需要对之前的框架进行如下改造：

想办法去掉 `while` 循环里面的 `for` 循环。

有了刚才的铺垫，这个不难理解，刚才说 `for` 循环是干什么用的来着？

是为了让二叉树一层一层往下遍历，让 BFS 算法一步一步向外扩散，因为这个层数 `depth`，或者这个步数 `step`，在之前的场景中有用。

但现在我们想解决「加权图」中的最短路径问题，「步数」已经没有参考意义了，「路径的权重之和」才有意义，所以这个 `for` 循环可以被去掉。

怎么去掉？就拿二叉树的层级遍历来说，其实你可以直接去掉 `for` 循环相关的代码：

```
// 输入一棵二叉树的根节点，遍历这棵二叉树所有节点
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 遍历二叉树的每一个节点
    while (!q.isEmpty()) {
        TreeNode cur = q.poll();
        printf("我不知道节点 %s 在第几层", cur);

        // 将子节点放入队列
        if (cur.left != null) {
            q.offer(cur.left);
        }
        if (cur.right != null) {
            q.offer(cur.right);
        }
    }
}
```

但问题是，没有 `for` 循环，你也没办法维护 `depth` 变量了。

如果你想同时维护 `depth` 变量，让每个节点 `cur` 知道自己在第几层，可以想其他办法，比如新建一个 `State` 类，记录每个节点所在的层数：

```
class State {
    // 记录 node 节点的深度
    int depth;
    TreeNode node;

    State(TreeNode node, int depth) {
        this.depth = depth;
        this.node = node;
    }
}

// 输入一棵二叉树的根节点，遍历这棵二叉树所有节点
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<State> q = new LinkedList<>();
    q.offer(new State(root, 1));

    // 遍历二叉树的每一个节点
    while (!q.isEmpty()) {
        State cur = q.poll();
        TreeNode cur_node = cur.node;
        int cur_depth = cur.depth;
        printf("节点 %s 在第 %s 层", cur_node, cur_depth);

        // 将子节点放入队列
        if (cur_node.left != null) {
            q.offer(new State(cur_node.left, cur_depth + 1));
        }
        if (cur_node.right != null) {
            q.offer(new State(cur_node.right, cur_depth + 1));
        }
    }
}
```

这样，我们就可以不使用 `for` 循环也确切地知道每个二叉树节点的深度了。

如果你能够理解上面这段代码，我们就可以来看 Dijkstra 算法的代码框架了。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 回溯算法解题套路框架



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">46. Permutations</a>	<a href="#">46. 全排列</a>	●

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的遍历框架
- 多叉树结构及遍历框架

tip：本文有视频版：[回溯算法框架套路详解](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

这篇文章是很久之前的一篇[回溯算法详解](#)的进阶版。把框架给你讲清楚，你会发现回溯算法问题都是一个套路。

本文解决几个问题：

回溯算法是什么？解决回溯算法相关的问题有什么技巧？如何学习回溯算法？回溯算法代码是否有规律可循？

其实回溯算法和我们常说的 DFS 算法基本可以认为是同一种算法，它们的细微差异我在[关于 DFS 和回溯算法的若干问题](#)中有详细解释，本文聚焦回溯算法，不展开。

抽象地说，解决一个回溯问题，实际上就是遍历一棵决策树的过程，树的每个叶子节点存放着一个合法答案。你把整棵树遍历一遍，把叶子节点上的答案都收集起来，就能得到所有的合法答案。

站在回溯树的一个节点上，你只需要思考 3 个问题：

- 1、路径：也就是已经做出的选择。
- 2、选择列表：也就是你当前可以做的选择。
- 3、结束条件：也就是到达决策树底层，无法再做选择的条件。

如果你不理解这三个词语的解释，没关系，我们后面会用「全排列」这个经典的回溯算法问题来帮你理解这些词语是什么意思，现在你先留着印象。

代码方面，回溯算法的框架：

```
result = []
def backtrack(路径, 选择列表):
```

```

if 满足结束条件:
    result.add(路径)
    return

for 选择 in 选择列表:
    做选择
    backtrack(路径, 选择列表)
    撤销选择

```

其核心就是 **for 循环里面的递归**，在递归调用之前「做选择」，在递归调用之后「撤销选择」，特别简单。

什么叫做选择和撤销选择呢，这个框架的底层原理是什么呢？下面我们就通过「全排列」这个问题来解开之前的疑惑，详细探究一下其中的奥妙！

## 一、全排列问题

力扣第 46 题「全排列」就是给你输入一个数组 `nums`，让你返回这些数字的全排列。

我们这次讨论的全排列问题不包含重复的数字，包含重复数字的扩展场景我在后文 [回溯算法秒杀排列组合子集的九种题型](#) 中讲解。

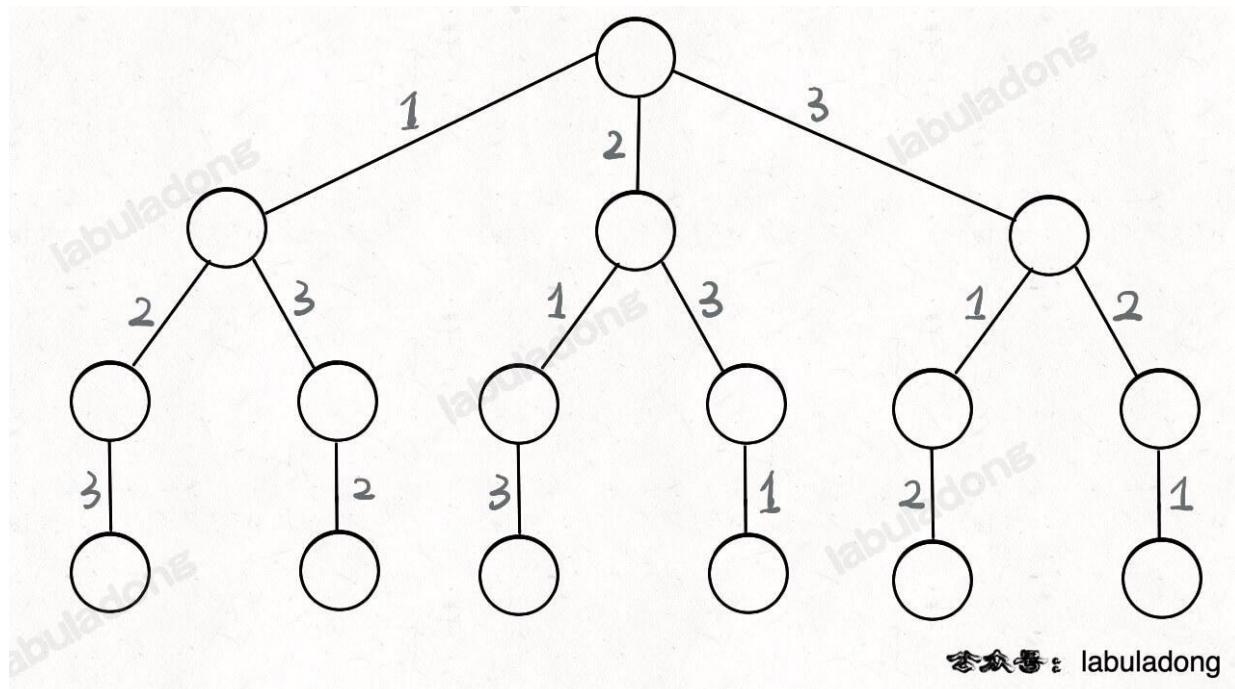
另外，有些读者之前看过的全排列算法代码可能是那种 `swap` 交换元素的写法，和我在本文介绍的代码不同。这是回溯算法两种穷举思路，我会在后文 [球盒模型：回溯算法穷举的两种视角](#) 讲明白。现在还不适合直接跟你讲那个解法，你照着我的思路学习即可。

我们在高中的时候就做过排列组合的数学题，我们也知道  $n$  个不重复的数，全排列共有  $n!$  个。那么我们当时是怎么穷举全排列的呢？

比方说给三个数 `[1, 2, 3]`，你肯定不会无规律地乱穷举，一般是这样：

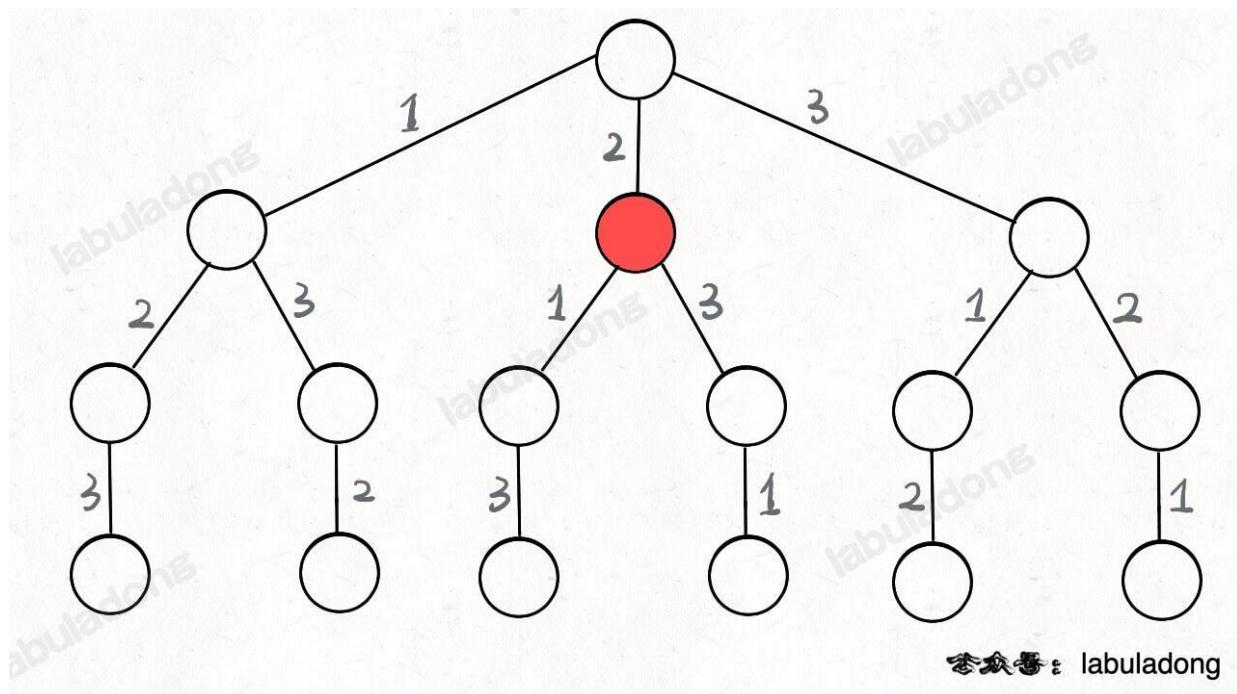
先固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；然后就只能变化第一位，变成 2，然后再穷举后两位……

其实这就是回溯算法，我们高中无师自通就会用，或者有的同学直接画出如下这棵回溯树：



只要从根遍历这棵树，记录路径上的数字，其实就是所有的全排列。我们不妨把这棵树称为回溯算法的「决策树」。

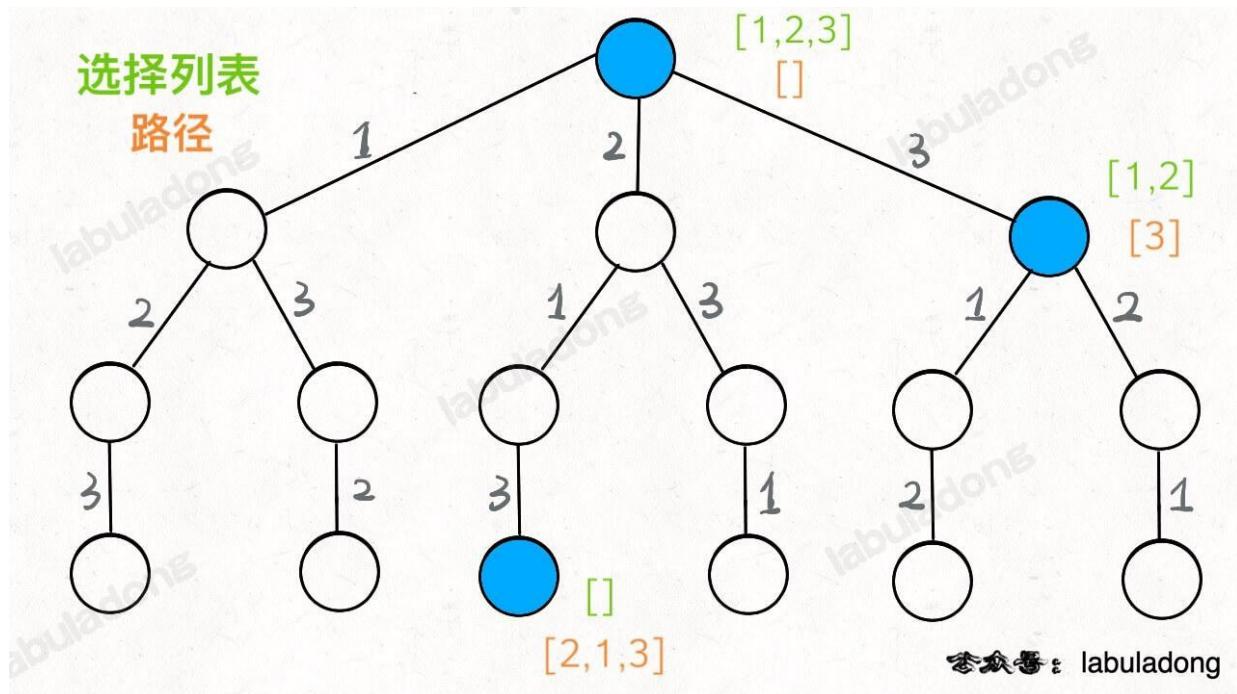
为啥说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



你现在就在做决策，可以选择 1 那条树枝，也可以选择 3 那条树枝。为啥只能在 1 和 3 之中选择呢？因为 2 这个树枝在你身后，这个选择你之前做过了，而全排列是不允许重复使用数字的。

现在可以解答开头的几个名词：[2] 就是「路径」，记录你已经做过的选择；[1, 3] 就是「选择列表」，表示你当前可以做出的选择；「结束条件」就是遍历到树的底层叶子节点，这里也就是选择列表为空的时候。

如果明白了这几个名词，可以把「路径」和「选择」列表作为决策树上每个节点的属性，比如下图列出了几个蓝色节点的属性：



我们定义的 `backtrack` 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层叶子节点，其「路径」就是一个全排列。

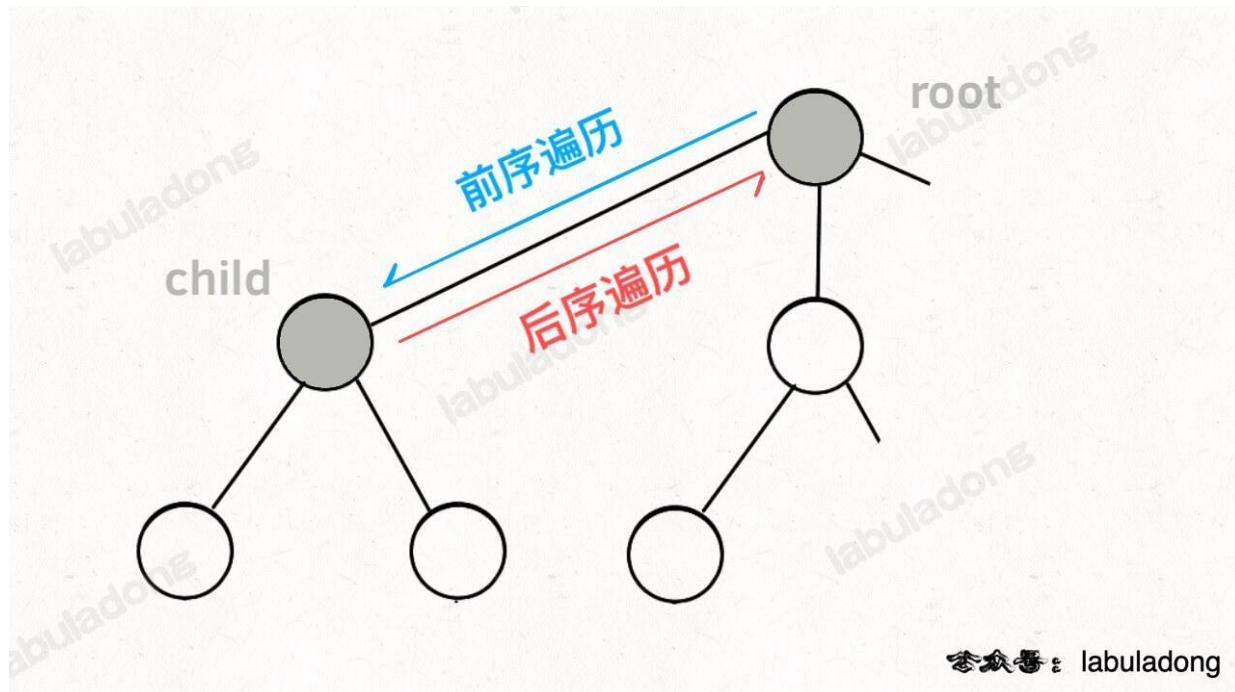
再进一步，如何遍历一棵树？这个应该不难吧。回忆一下之前 [学习数据结构的框架思维](#) 写过，各种搜索问题其实都是树的遍历问题，而多叉树的遍历框架就是这样：

```
void traverse(TreeNode root) {  
    for (TreeNode child : root.children) {  
        // 前序位置需要的操作  
        traverse(child);  
        // 后序位置需要的操作  
    }  
}
```

细心的读者肯定会疑问：多叉树 DFS 遍历框架的前序位置和后序位置应该在 for 循环外面，并不应该是在 for 循环里面呀？为什么在回溯算法中跑到 for 循环里面了？

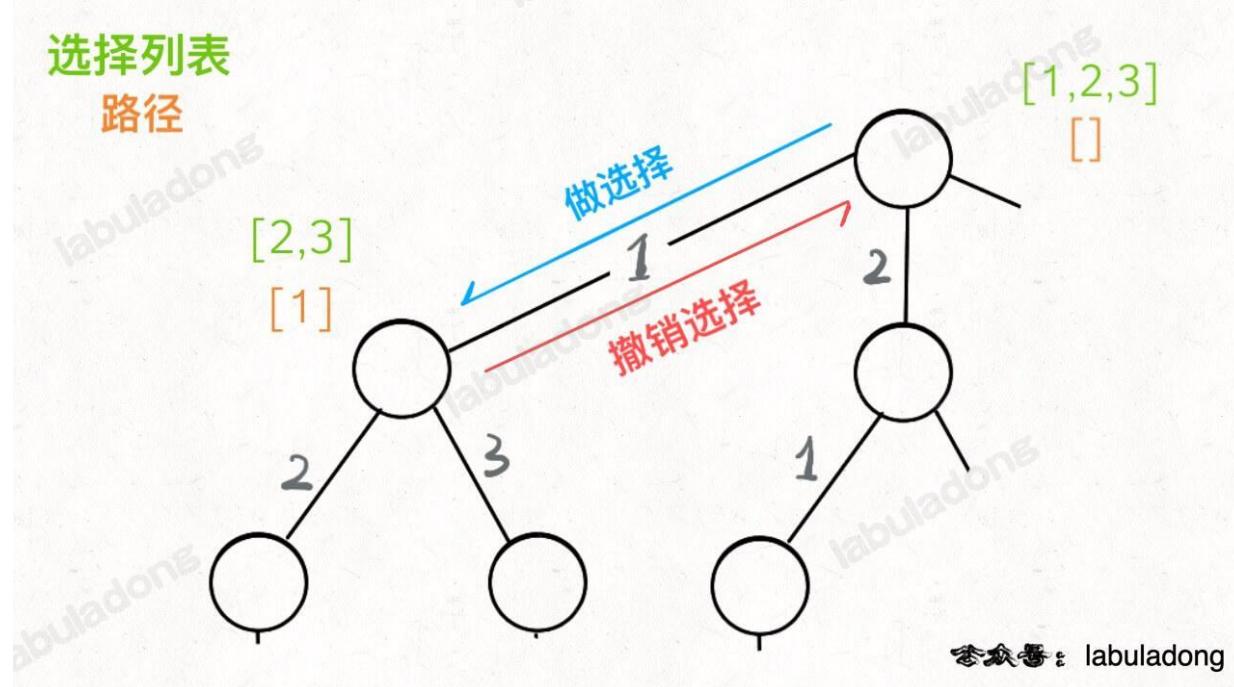
是的，DFS 算法的前序和后序位置应该在 for 循环外面，不过回溯算法和 DFS 算法略有不同，[解答回溯/DFS 算法的若干疑问](#) 会具体讲解，这里可以暂且忽略这个问题。

而所谓的前序遍历和后序遍历，他们只是两个很有用的时间点，我给你画张图你就明白了：



前序遍历的代码在进入某一个节点之前的那个时间点执行，后序遍历代码在离开某个节点之后的那个时间点执行。

回想我们刚才说的，「路径」和「选择」是每个节点的属性，函数在树上游走要正确处理节点的属性，那么就要在这两个特殊时间点搞点动作：



现在，你是否理解了回溯算法的这段核心框架？

```
for 选择 in 选择列表:
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

我们只要在递归之前做出选择，在递归之后撤销刚才的选择，就能正确得到每个节点的选择列表和路径。

下面，直接看全排列代码：

```
class Solution {
    List<List<Integer>> res = new LinkedList<>();

    // 主函数，输入一组不重复的数字，返回它们的全排列
    List<List<Integer>> permute(int[] nums) {
        // 记录「路径」
        LinkedList<Integer> track = new LinkedList<>();
        // 「路径」中的元素会被标记为 true，避免重复使用
        boolean[] used = new boolean[nums.length];

        backtrack(nums, track, used);
        return res;
    }

    // 路径：记录在 track 中
    // 选择列表：nums 中不存在于 track 的那些元素 (used[i] 为 false)
    // 结束条件：nums 中的元素全都在 track 中出现
    void backtrack(int[] nums, LinkedList<Integer> track, boolean[] used) {
        // 触发结束条件
        if (track.size() == nums.length) {
```

```

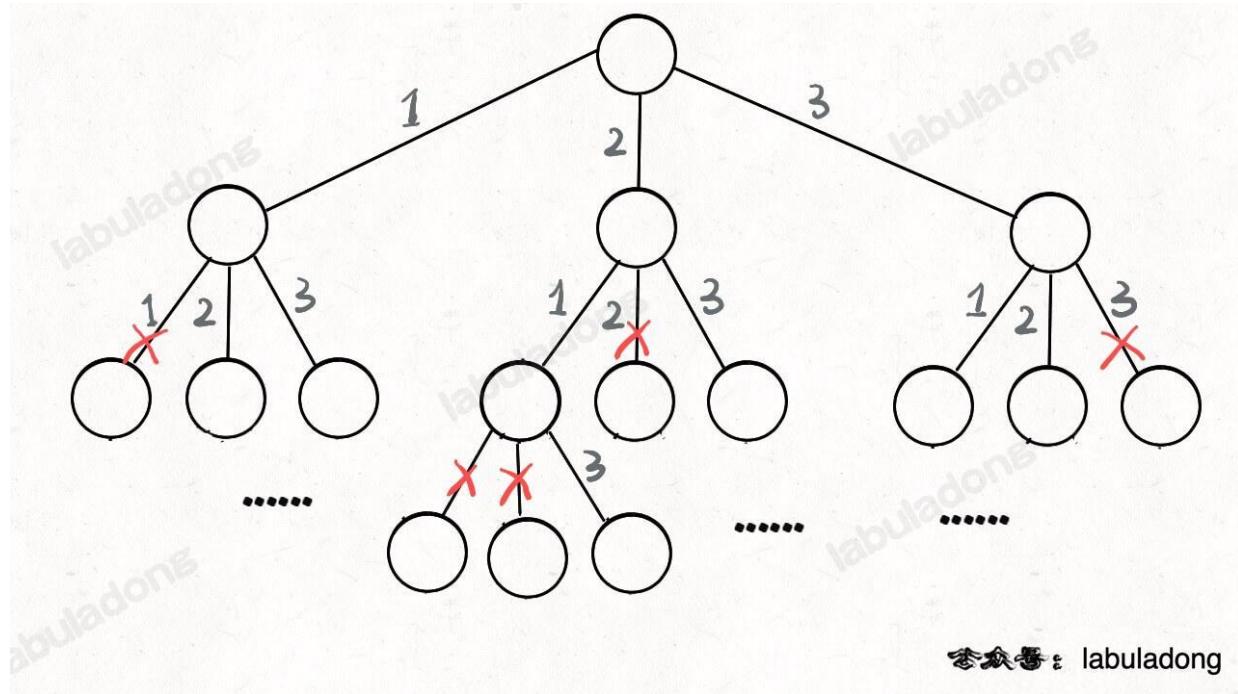
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            // nums[i] 已经在 track 中, 跳过
            continue;
        }
        // 做选择
        track.add(nums[i]);
        used[i] = true;
        // 进入下一层决策树
        backtrack(nums, track, used);
        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}

```

▶  代码可视化动画

我们这里稍微做了些变通，没有显式记录「选择列表」，而是通过 `used` 数组排除已经存在 `track` 中的元素，从而推导出当前的选择列表：



至此，我们就通过全排列问题详解了回溯算法的底层原理。当然，这个算法解决全排列不是最高效的，你可能看到有的解法连 `used` 数组都不使用，通过交换元素达到目的。但是那种解法稍微难理解一些，我会在 [球盒模型：回溯算法两种穷举视角](#) 中介绍。

但是必须说明的是，不管怎么优化，都符合回溯框架，而且时间复杂度都不可能低于  $O(N!)$ ，因为穷举整棵决策树是无法避免的，你最后肯定要穷举出  $N!$  种全排列结果。

这也是回溯算法的一个特点，不像动态规划存在重叠子问题可以优化，回溯算法就是纯暴力穷举，复杂度一般都很高。

## 最后总结

回溯算法就是个多叉树的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作，算法框架如下：

```
def backtrack(...):
    for 选择 in 选择列表:
        做选择
        backtrack(...)
        撤销选择
```

写 `backtrack` 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

其实想想看，回溯算法和动态规划是不是有点像呢？我们在动态规划系列文章中多次强调，动态规划的三个需要明确的点就是「状态」「选择」和「base case」，是不是就对应着走过的「路径」，当前的「选择列表」和「结束条件」？

动态规划和回溯算法底层都把问题抽象成了树的结构，但这两种算法在思路上是完全不同的。在 [二叉树心法（纲领篇）](#) 你将看到动态规划和回溯算法更深层次的区别和联系。

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	<a href="#">111. 二叉树的最小深度</a>	
<a href="#">112. Path Sum</a>	<a href="#">112. 路径总和</a>	
<a href="#">113. Path Sum II</a>	<a href="#">113. 路径总和 II</a>	
<a href="#">131. Palindrome Partitioning</a>	<a href="#">131. 分割回文串</a>	
<a href="#">140. Word Break II</a>	<a href="#">140. 单词拆分 II</a>	
<a href="#">1593. Split a String Into the Max Number of Unique Substrings</a>	<a href="#">1593. 拆分字符串使唯一子字符串的数目最大</a>	
<a href="#">17. Letter Combinations of a Phone Number</a>	<a href="#">17. 电话号码的字母组合</a>	
<a href="#">22. Generate Parentheses</a>	<a href="#">22. 括号生成</a>	
<a href="#">301. Remove Invalid Parentheses</a>	<a href="#">301. 删除无效的括号</a>	
<a href="#">332. Reconstruct Itinerary</a>	<a href="#">332. 重新安排行程</a>	
<a href="#">39. Combination Sum</a>	<a href="#">39. 组合总和</a>	
<a href="#">51. N-Queens</a>	<a href="#">51. N 皇后</a>	
<a href="#">638. Shopping Offers</a>	<a href="#">638. 大礼包</a>	
<a href="#">698. Partition to K Equal Sum Subsets</a>	<a href="#">698. 划分为k个相等的子集</a>	
<a href="#">77. Combinations</a>	<a href="#">77. 组合</a>	
<a href="#">78. Subsets</a>	<a href="#">78. 子集</a>	
<a href="#">784. Letter Case Permutation</a>	<a href="#">784. 字母大小写全排列</a>	

LeetCode	力扣	难度
89. Gray Code	89. 格雷编码	
93. Restore IP Addresses	93. 复原 IP 地址	
-	剑指 Offer 34. 二叉树中和为某一值的路径	
-	剑指 Offer II 079. 所有子集	
-	剑指 Offer II 080. 含有 k 个元素的组合	
-	剑指 Offer II 081. 允许重复选择元素的组合	
-	剑指 Offer II 083. 没有重复元素集合的全排列	
-	剑指 Offer II 085. 生成匹配的括号	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

# 回溯算法秒杀所有排列/组合/子集问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">77. Combinations</a>	77. 组合	简单
<a href="#">90. Subsets II</a>	90. 子集 II	简单
-	剑指 Offer II 082. 含有重复元素集合的组合	简单
<a href="#">216. Combination Sum III</a>	216. 组合总和 III	中等
<a href="#">40. Combination Sum II</a>	40. 组合总和 II	中等
<a href="#">78. Subsets</a>	78. 子集	简单
<a href="#">47. Permutations II</a>	47. 全排列 II	中等
<a href="#">39. Combination Sum</a>	39. 组合总和	中等
<a href="#">46. Permutations</a>	46. 全排列	中等
-	剑指 Offer II 084. 含有重复元素集合的全排列	中等

阅读本文前，你需要先学习：

- [二叉树系列算法（纲领篇）](#)
- [回溯算法核心框架](#)

tip：本文有视频版：[回溯算法秒杀所有排列/组合/子集问题](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

虽然排列、组合、子集系列问题是高中就学过的，但如果想编写算法解决它们，还是非常考验计算机思维的，本文就讲讲编程解决这几个问题的核心思路，以后再有什么变体，你也能手到擒来，以不变应万变。

无论是排列、组合还是子集问题，简单说无非就是让你从序列 `nums` 中以给定规则取若干元素，主要有以下几种变体：

**形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，这也是最基本的形式。**

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该只有 `[7]`。

**形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次。**

以组合为例，如果输入 `nums = [2,5,2,1,2]`，和为 7 的组合应该有两种 `[2,2,2,1]` 和 `[5,2]`。

**形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次。**

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该有两种 `[2,2,3]` 和 `[7]`。

当然，也可以说有第四种形式，即元素可重可复选。但既然元素可复选，那又何必存在重复元素呢？元素去重之后就等同于形式三，所以这种情况不用考虑。

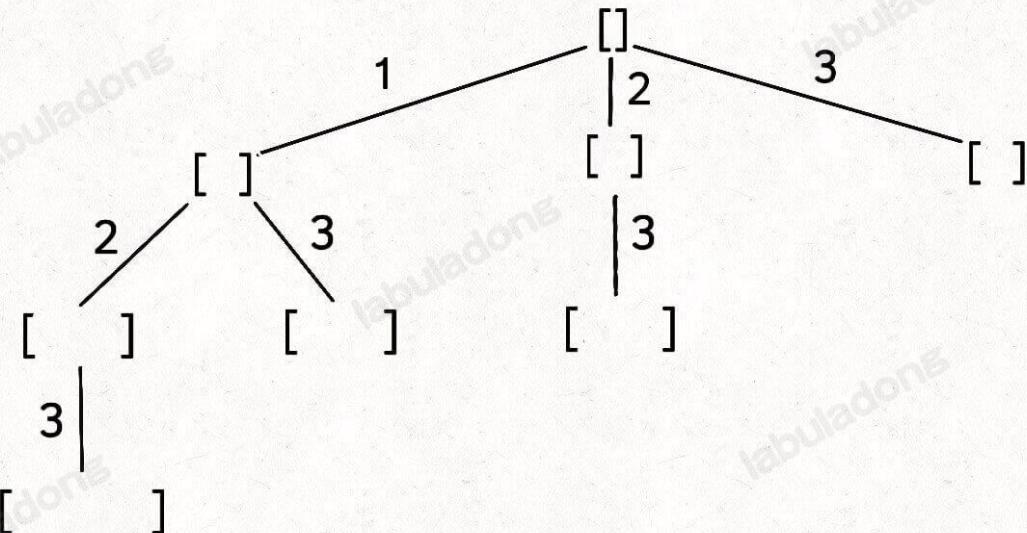
上面用组合问题举的例子，但排列、组合、子集问题都可以有这三种基本形式，所以共有 9 种变化。

除此之外，题目也可以再添加各种限制条件，比如让你求和为 `target` 且元素个数为 `k` 的组合，那这么一来又可以衍生出一堆变体，怪不得面试笔试中经常考到排列组合这种基本题型。

但无论形式怎么变化，其本质就是穷举所有解，而这些解呈现树形结构，所以合理使用回溯算法框架，稍改代码框架即可把这些问题一网打尽。

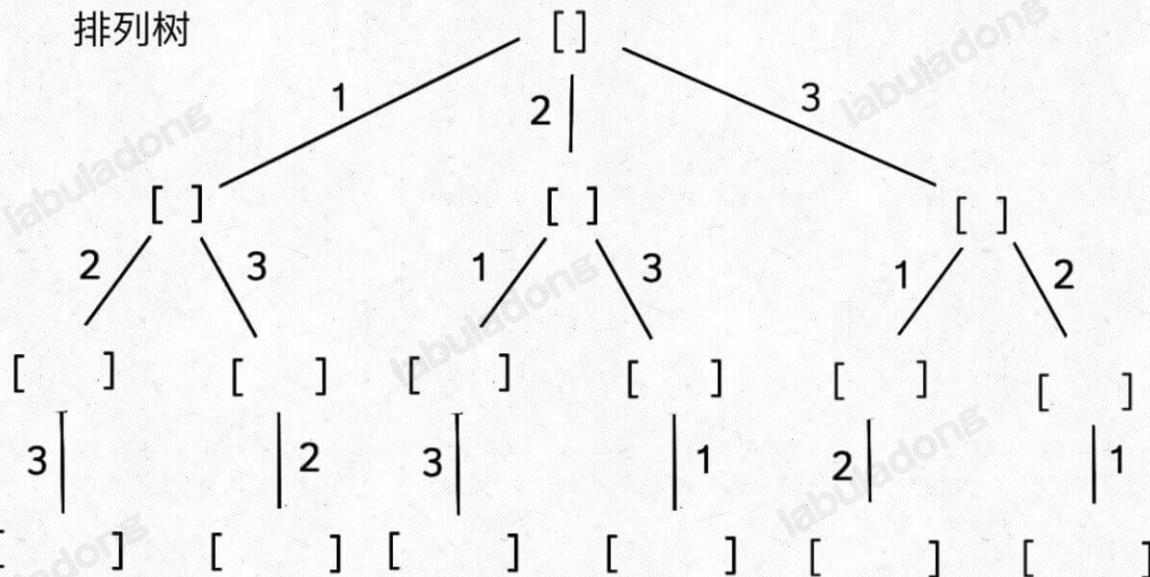
具体来说，你需要先阅读并理解前文 [回溯算法核心套路](#)，然后记住如下子集问题和排列问题的回溯树，就可以解决所有排列组合子集相关的问题：

组合/子集树



© labuladong

排列树



© labuladong

为什么只要记住这两种树形结构就能解决所有相关问题呢？

首先，组合问题和子集问题其实是等价的，这个后面会讲；至于之前说的三种变化形式，无非是在这两棵树上剪掉或者增加一些树枝罢了。

那么，接下来我们就开始穷举，把排列/组合/子集问题的 9 种形式都过一遍，学学如何用回溯算法把它们一套带走。

另外，有些读者之前看过的排列/子集/组合的解法代码可能和我在本文介绍的代码不同。这是因为回溯算法有两种穷举视角，我会在后文 [球盒模型：回溯算法穷举的两种视角](#) 手把手给你讲清楚。现在还不适合直接跟你讲那些解法，你照着我的思路学习即可。

## 子集（元素无重不可复选）

力扣第 78 题「子集」就是这个问题：

题目给你输入一个无重复元素的数组 `nums`，其中每个元素最多使用一次，请你返回 `nums` 的所有子集。

函数签名如下：

```
List<List<Integer>> subsets(int[] nums)
```

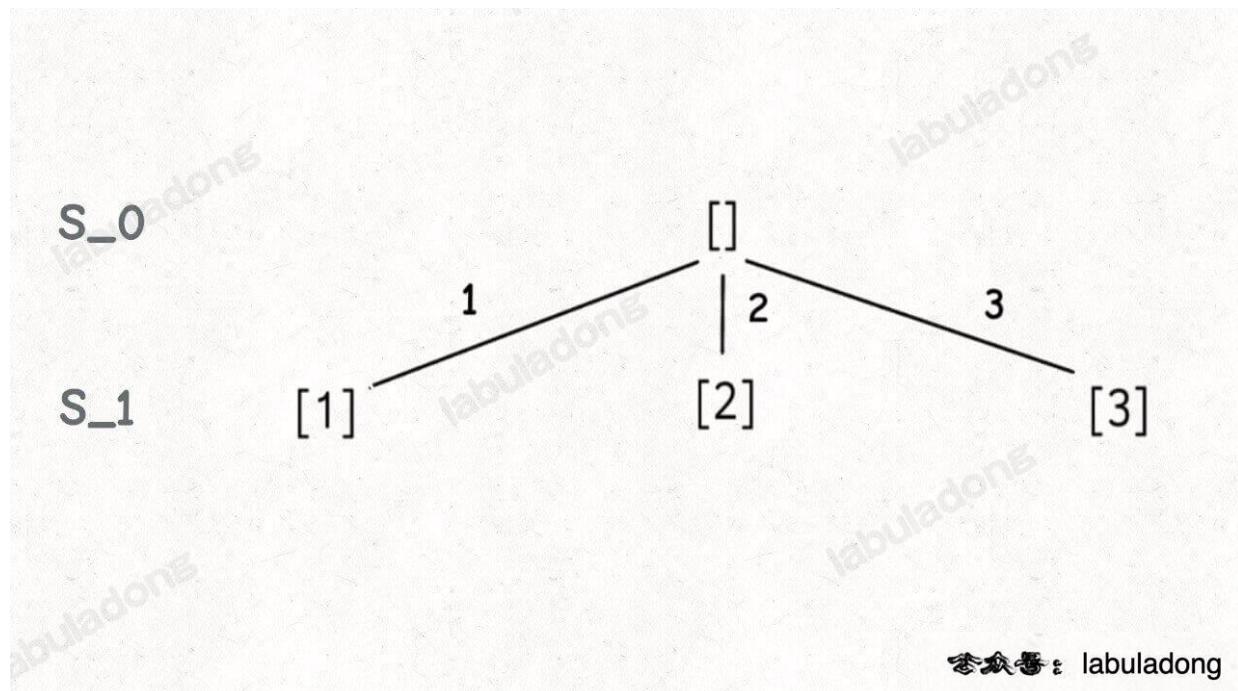
比如输入 `nums = [1,2,3]`，算法应该返回如下子集：

```
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

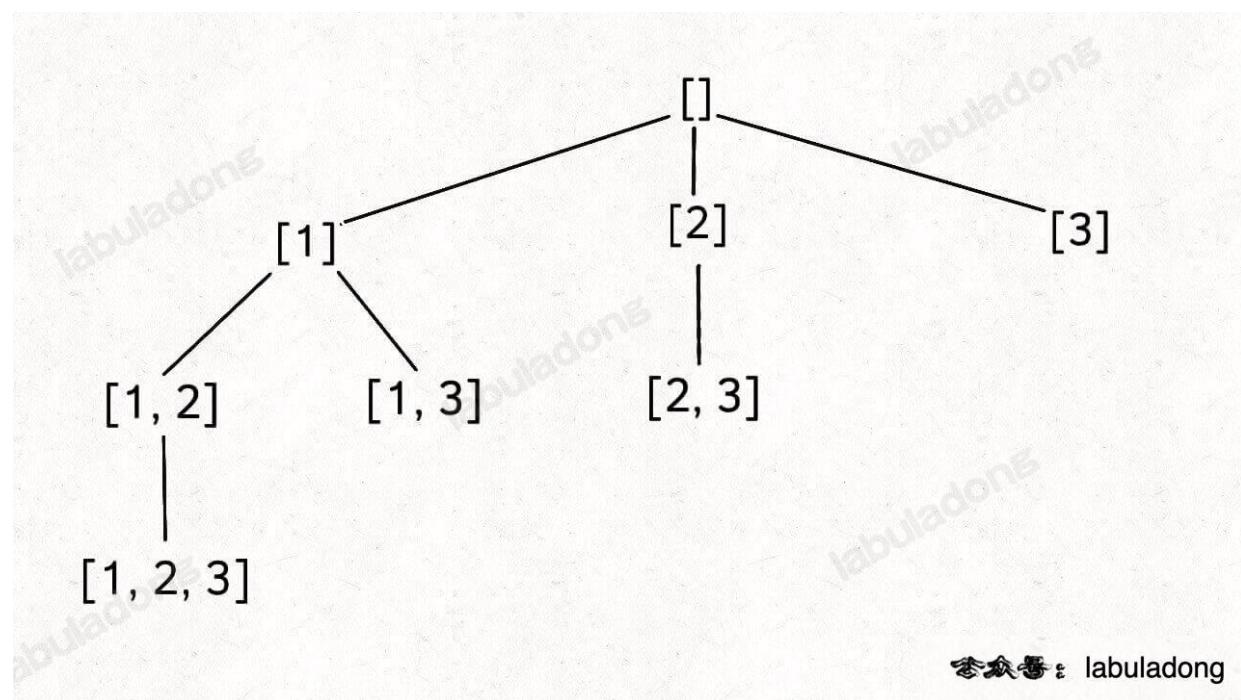
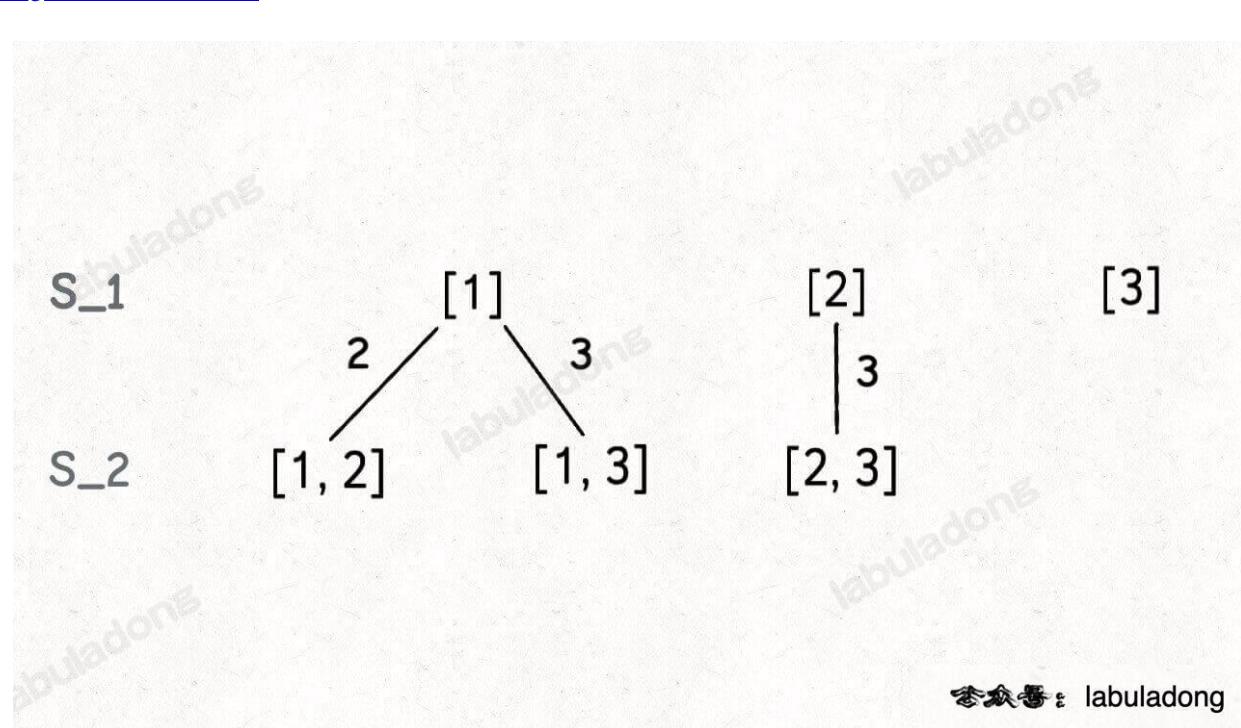
好，我们暂时不考虑如何用代码实现，先回忆一下我们的高中知识，如何手推所有子集？

首先，生成元素个数为 0 的子集，即空集 `[]`，为了方便表示，我称之为 `S_0`。

然后，在 `S_0` 的基础上生成元素个数为 1 的所有子集，我称为 `S_1`：

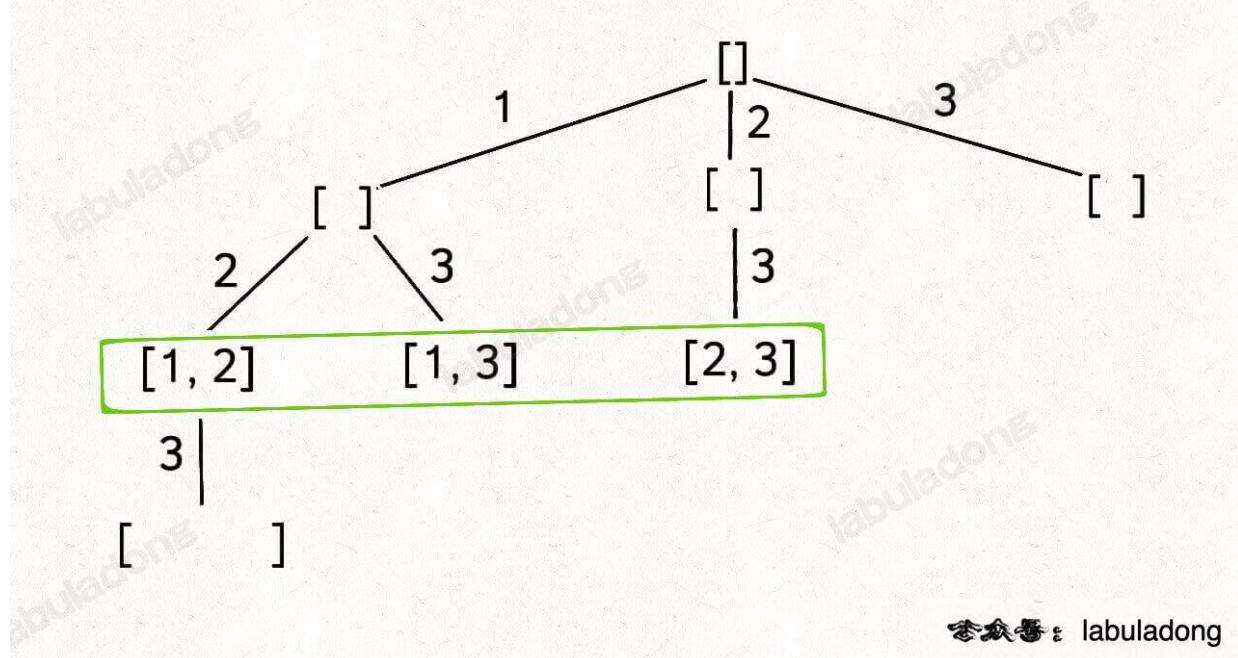


接下来，我们可以在 `S_1` 的基础上推导出 `S_2`，即元素个数为 2 的所有子集：



如果把根节点作为第  $0$  层, 将每个节点和根节点之间树枝上的元素作为该节点的值, 那么第  $n$  层的所有节点就是大小为  $n$  的所有子集。

你比如大小为  $2$  的子集就是这一层节点的值:



© labuladong

注意，本文之后所说「节点的值」都是指节点和根节点之间树枝上的元素，且将根节点认为是第 0 层。

那么再进一步，如果想计算所有子集，那只要遍历这棵多叉树，把所有节点的值收集起来不就行了？

直接看代码：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();

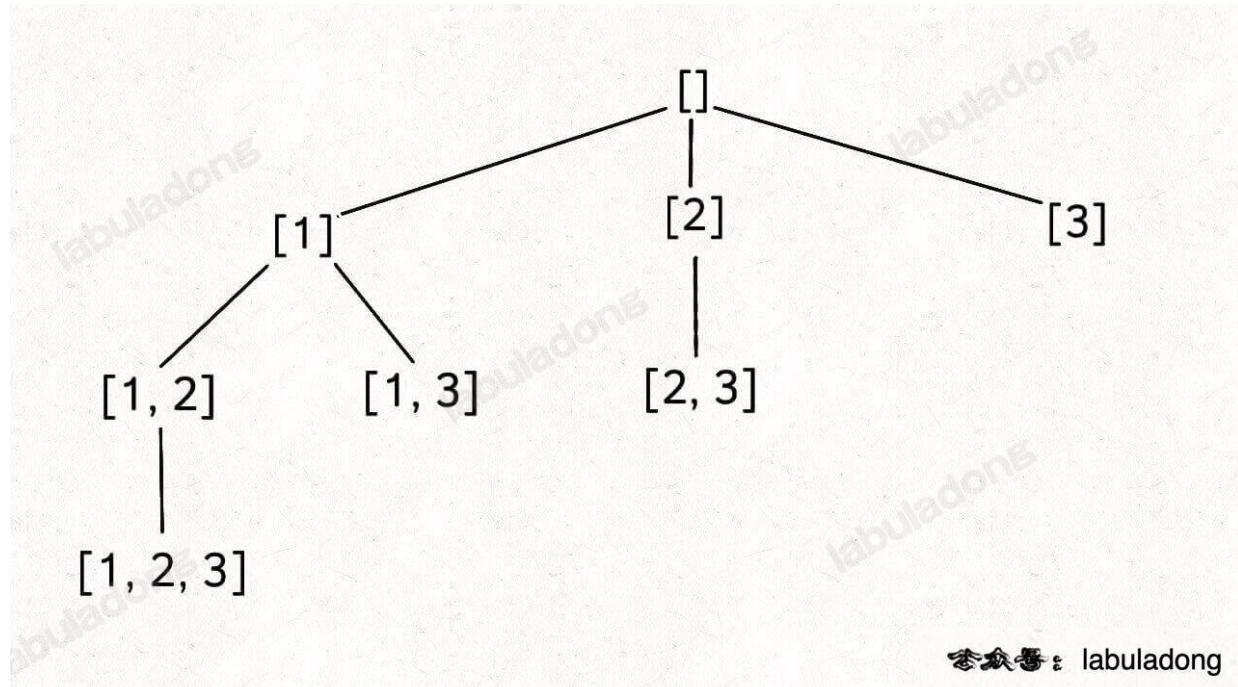
    // 主函数
    public List<List<Integer>> subsets(int[] nums) {
        backtrack(nums, 0);
        return res;
    }

    // 回溯算法核心函数，遍历子集问题的回溯树
    void backtrack(int[] nums, int start) {

        // 前序位置，每个节点的值都是一个子集
        res.add(new LinkedList<>(track));

        // 回溯算法标准框架
        for (int i = start; i < nums.length; i++) {
            // 做选择
            track.addLast(nums[i]);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(nums, i + 1);
            // 撤销选择
            track.removeLast();
        }
    }
}
```

看过前文 [回溯算法核心框架](#) 的读者应该很容易理解这段代码吧，我们使用 `start` 参数控制树枝的生长避免产生重复的子集，用 `track` 记录根节点到每个节点的路径的值，同时在前序位置把每个节点的路径值收集起来，完成回溯树的遍历就收集了所有子集：



© labuladong

最后，`backtrack` 函数开头看似没有 base case，会不会进入无限递归？

其实不会的，当 `start == nums.length` 时，叶子节点的值会被装入 `res`，但 for 循环不会执行，也就结束了递归。

---

▶ 🎃 代码可视化动画🎃

---

## 组合（元素无重不可复选）

如果你能够成功的生成所有无重子集，那么你稍微改改代码就能生成所有无重组合了。

你比如说，让你在 `nums = [1, 2, 3]` 中拿 2 个元素形成所有的组合，你怎么做？

稍微想想就会发现，大小为 2 的所有组合，不就是所有大小为 2 的子集嘛。

所以我说组合和子集是一样的：大小为 `k` 的组合就是大小为 `k` 的子集。

比如力扣第 77 题「组合」：

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

函数签名如下：

```
List<List<Integer>> combine(int n, int k)
```

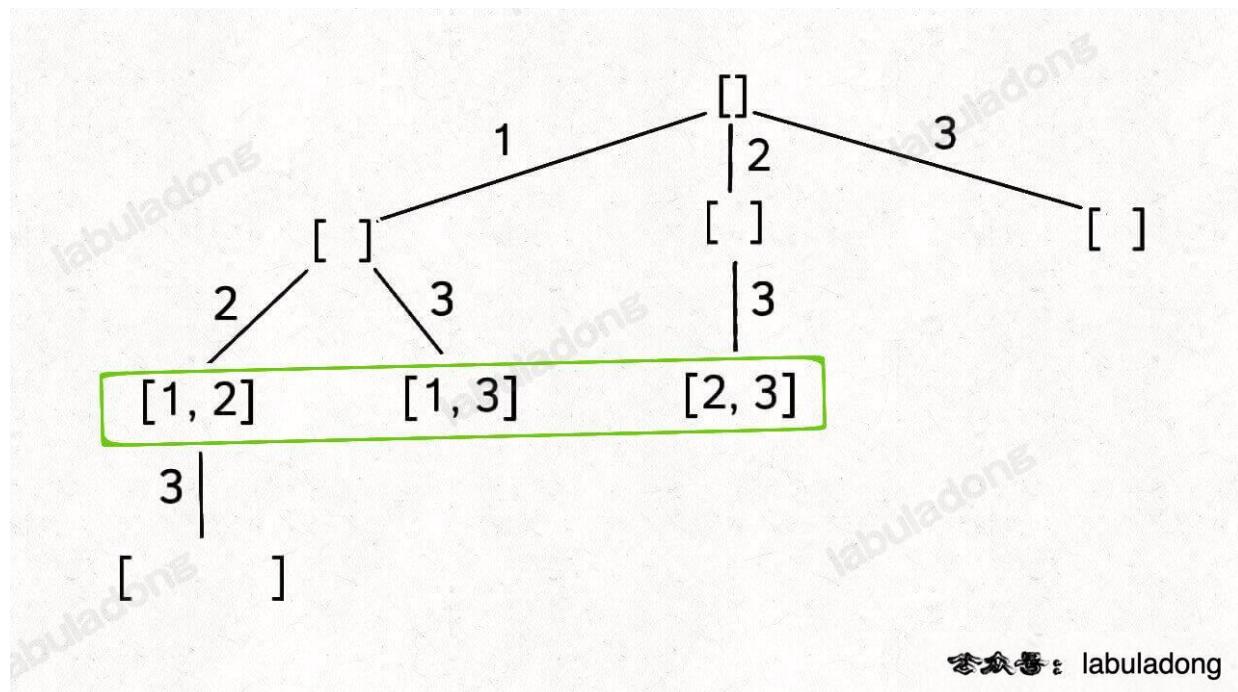
比如 `combine(3, 2)` 的返回值应该是：

```
[ [1,2], [1,3], [2,3] ]
```

这是标准的组合问题，但我给你翻译一下就变成子集问题了：

给你输入一个数组 `nums = [1, 2, ..., n]` 和一个正整数 `k`，请你生成所有大小为 `k` 的子集。

还是以 `nums = [1, 2, 3]` 为例，刚才让你求所有子集，就是把所有节点的值都收集起来；现在你只需要把第 2 层（根节点视为第 0 层）的节点收集起来，就是大小为 2 的所有组合：



反映到代码上，只需要稍改 base case，控制算法仅仅收集第 `k` 层节点的值即可：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();

    // 主函数
    public List<List<Integer>> combine(int n, int k) {
        backtrack(1, n, k);
        return res;
    }

    void backtrack(int start, int n, int k) {
        // base case
        if (k == track.size()) {
            // 遍历到了第 k 层，收集当前节点的值
            res.add(new LinkedList<>(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = start; i <= n; i++) {
            // 选择
            track.addLast(i);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(i + 1, n, k);
            // 撤销选择
            track.removeLast();
        }
    }
}
```

```
        }
    }
}
```

这样，标准的组合问题也解决了。

---

► [代码可视化动画](#)

---

## 排列（元素无重不可复选）

排列问题在前文 [回溯算法核心框架](#) 讲过，这里就简单过一下。

力扣第 46 题「全排列」就是标准的排列问题：

给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。

函数签名如下：

```
List<List<Integer>> permute(int[] nums)
```

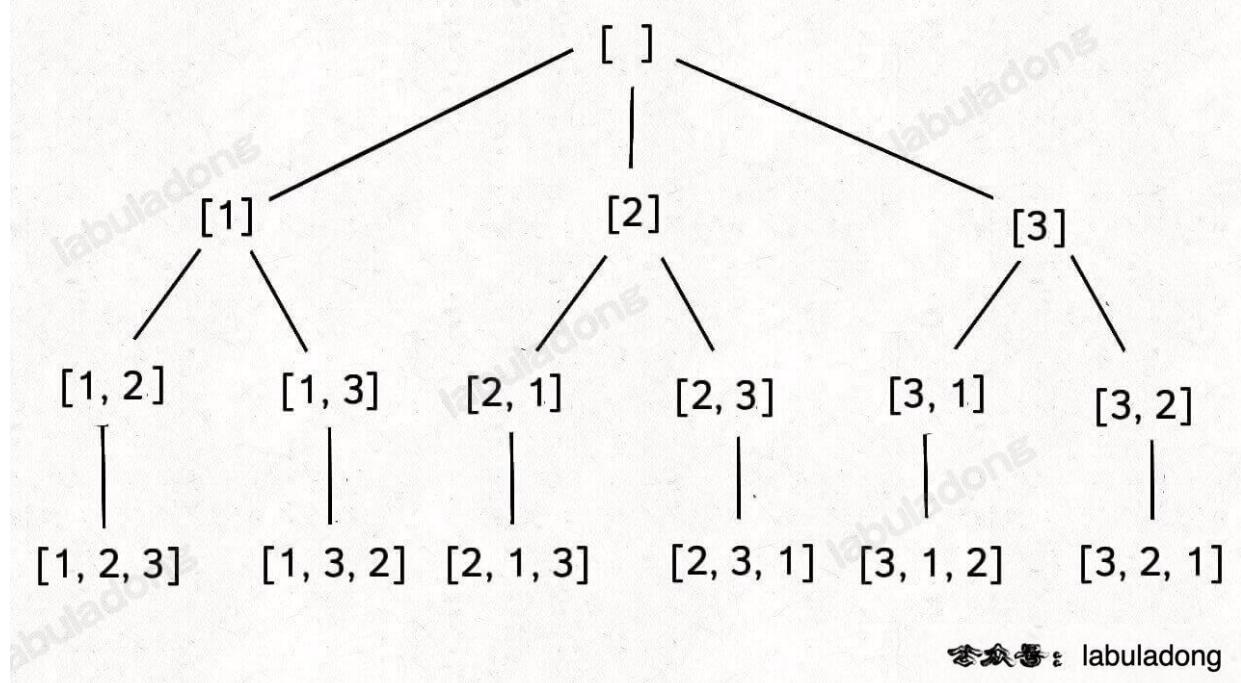
比如输入 `nums = [1,2,3]`，函数的返回值应该是：

```
[
    [1,2,3], [1,3,2],
    [2,1,3], [2,3,1],
    [3,1,2], [3,2,1]
]
```

刚才讲的组合/子集问题使用 `start` 变量保证元素 `nums[start]` 之后只会出现 `nums[start+1..]` 中的元素，通过固定元素的相对位置保证不出现重复的子集。

但排列问题本身就是让你穷举元素的位置，`nums[i]` 之后也可以出现 `nums[i]` 左边的元素，所以之前的那一套玩不转了，需要额外使用 `used` 数组来标记哪些元素还可以被选择。

标准全排列可以抽象成如下这棵多叉树：



✿✿✿✿ labuladong

我们用 `used` 数组标记已经在路径上的元素避免重复选择，然后收集所有叶子节点上的值，就是所有全排列的结果：

```

class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();
    // track 中的元素会被标记为 true
    boolean[] used;

    // 主函数，输入一组不重复的数字，返回它们的全排列
    public List<List<Integer>> permute(int[] nums) {
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }

    // 回溯算法核心函数
    void backtrack(int[] nums) {
        // base case, 到达叶子节点
        if (track.size() == nums.length) {
            // 收集叶子节点上的值
            res.add(new LinkedList(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = 0; i < nums.length; i++) {
            // 已经存在 track 中的元素，不能重复选择
            if (used[i]) {
                continue;
            }
            // 做选择
            used[i] = true;
            track.addLast(nums[i]);
            // 进入下一层回溯树
            backtrack(nums);
        }
    }
}

```

```
// 取消选择
track.removeLast();
used[i] = false;
}
}
}
```

### ▶ 😊 代码可视化动画😊

这样，全排列问题就解决了。

但如果题目不让你算全排列，而是让你算元素个数为  $k$  的排列，怎么算？

也很简单，改下 `backtrack` 函数的 base case，仅收集第  $k$  层的节点值即可：

```
// 回溯算法核心函数
void backtrack(int[] nums, int k) {
    // base case, 到达第 k 层，收集节点的值
    if (track.size() == k) {
        // 第 k 层节点的值就是大小为 k 的排列
        res.add(new LinkedList(track));
        return;
    }

    // 回溯算法标准框架
    for (int i = 0; i < nums.length; i++) {
        // ...
        backtrack(nums, k);
        // ...
    }
}
```

## 子集/组合（元素可重不可复选）

刚才讲的标准子集问题输入的 `nums` 是没有重复元素的，但如果存在重复元素，怎么处理呢？

力扣第 90 题「子集 II」就是这样一个问题：

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集。

函数签名如下：

```
List<List<Integer>> subsetsWithDup(int[] nums)
```

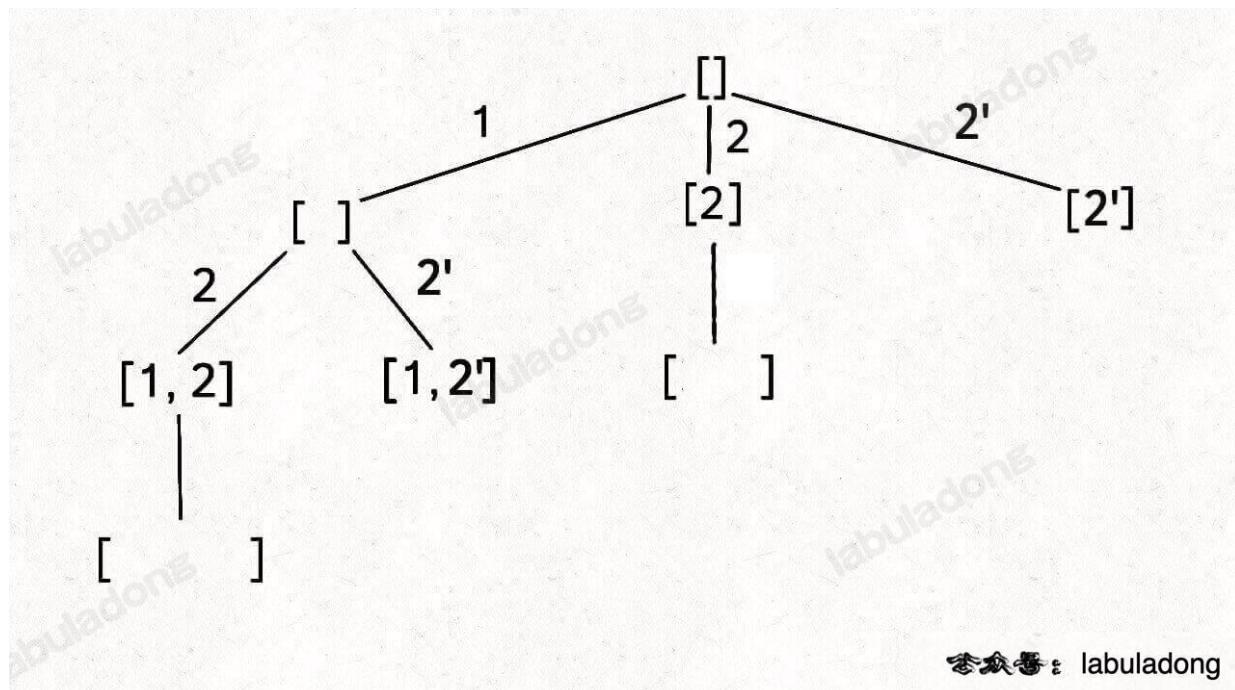
比如输入 `nums = [1,2,2]`，你应该输出：

```
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]
```

当然，按道理说「集合」不应该包含重复元素的，但既然题目这样问了，我们就忽略这个细节吧，仔细思考一下这道题怎么做才是正事。

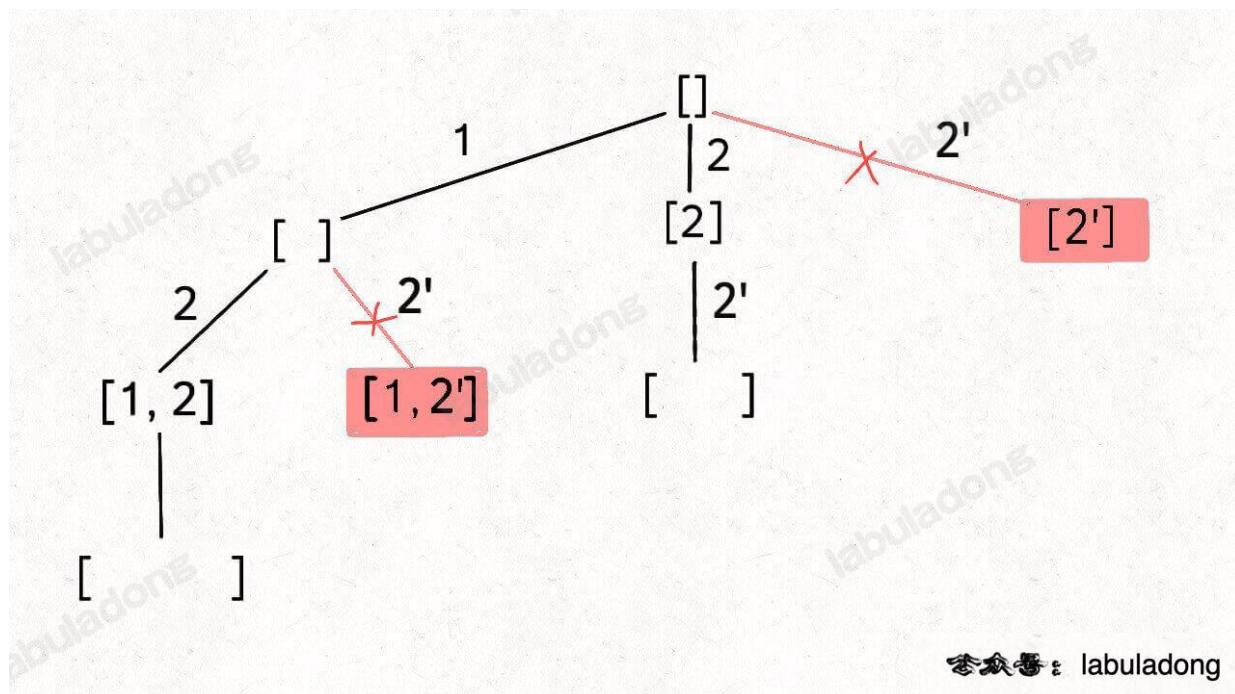
就以 `nums = [1, 2, 2]` 为例，为了区别两个 2 是不同元素，后面我们写作 `nums = [1, 2, 2']`。

按照之前的思路画出子集的树形结构，显然，两条值相同的相邻树枝会产生重复：



```
[
  [],
  [1], [2], [2'],
  [1, 2], [1, 2'], [2, 2'],
  [1, 2, 2']
]
```

你可以看到，`[2]` 和 `[1, 2]` 这两个结果出现了重复，所以我们需要进行剪枝，如果一个节点有多条值相同的树枝相邻，则只遍历第一条，剩下的都剪掉，不要去遍历：



体现在代码上，需要先进行排序，让相同的元素靠在一起，如果发现 `nums[i] == nums[i-1]`，则跳过：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> track = new LinkedList<>();

    public List<List<Integer>> subsetsWithDup(int[] nums) {
        // 先排序，让相同的元素靠在一起
        Arrays.sort(nums);
        backtrack(nums, 0);
        return res;
    }

    void backtrack(int[] nums, int start) {
        // 前序位置，每个节点的值都是一个子集
        res.add(new LinkedList<>(track));

        for (int i = start; i < nums.length; i++) {
            // 剪枝逻辑，值相同的相邻树枝，只遍历第一条
            if (i > start && nums[i] == nums[i - 1]) {
                continue;
            }
            track.addLast(nums[i]);
            backtrack(nums, i + 1);
            track.removeLast();
        }
    }
}
```

## ▶ 代码可视化动画

这段代码和之前标准的子集问题的代码几乎相同，就是添加了排序和剪枝的逻辑。

至于为什么要这样剪枝，结合前面的图应该也很容易理解，这样带重复元素的子集问题也解决了。

我们说了组合问题和子集问题是等价的，所以我们直接看一道组合的题目吧，这是力扣第 40 题「组合总和 II」：

给你输入 `candidates` 和一个目标和 `target`，从 `candidates` 中找出中所有和为 `target` 的组合。

`candidates` 可能存在重复元素，且其中的每个数字最多只能使用一次。

说这是一个组合问题，其实换个问法就变成子集问题了：请你计算 `candidates` 中所有和为 `target` 的子集。

所以这题怎么做呢？

对比子集问题的解法，只要额外用一个 `trackSum` 变量记录回溯路径上的元素和，然后将 base case 改一改即可解决这道题：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();
    // 记录 track 中的元素之和
    int trackSum = 0;
```

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {  
    if (candidates.length == 0) {  
        return res;  
    }  
    // 先排序，让相同的元素靠在一起  
    Arrays.sort(candidates);  
    backtrack(candidates, 0, target);  
    return res;  
}  
  
// 回溯算法主函数  
void backtrack(int[] nums, int start, int target) {  
    // base case, 达到目标和，找到符合条件的组合  
    if (trackSum == target) {  
        res.add(new LinkedList<>(track));  
        return;  
    }  
    // base case, 超过目标和，直接结束  
    if (trackSum > target) {  
        return;  
    }  
  
    // 回溯算法标准框架  
    for (int i = start; i < nums.length; i++) {  
        // 剪枝逻辑，值相同的树枝，只遍历第一条  
        if (i > start && nums[i] == nums[i - 1]) {  
            continue;  
        }  
        // 做选择  
        track.add(nums[i]);  
        trackSum += nums[i];  
        // 递归遍历下一层回溯树  
        backtrack(nums, i + 1, target);  
        // 撤销选择  
        track.removeLast();  
        trackSum -= nums[i];  
    }  
}
```

## ▶ 🎃 代码可视化动画🎃

## 排列（元素可重不可复选）

排列问题的输入如果存在重复，比子集/组合问题稍微复杂一点，我们看看力扣第 47 题「全排列 II」：

给你输入一个可包含重复数字的序列 `nums`，请你写一个算法，返回所有可能的全排列，函数签名如下：

```
List<List<Integer>> permuteUnique(int[] nums)
```

比如输入 `nums = [1,2,2]`，函数返回：

```
[ [1,2,2],[2,1,2],[2,2,1] ]
```

先看解法代码：

```
class Solution {

    List<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> track = new LinkedList<>();
    boolean[] used;

    public List<List<Integer>> permuteUnique(int[] nums) {
        // 先排序，让相同的元素靠在一起
        Arrays.sort(nums);
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }

    void backtrack(int[] nums) {
        if (track.size() == nums.length) {
            res.add(new LinkedList(track));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (used[i]) {
                continue;
            }
            // 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
            if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
                continue;
            }
            track.add(nums[i]);
            used[i] = true;
            backtrack(nums);
            track.removeLast();
            used[i] = false;
        }
    }
}
```

### ▶ 代码可视化动画

你对比一下之前的标准全排列解法代码，这段解法代码只有两处不同：

1、对 `nums` 进行了排序。

2、添加了一句额外的剪枝逻辑。

类比输入包含重复元素的子集/组合问题，你大概应该理解这么做是为了防止出现重复结果。

但是注意排列问题的剪枝逻辑，和子集/组合问题的剪枝逻辑略有不同：新增了 `!used[i - 1]` 的逻辑判断。

这个地方理解起来就需要一些技巧性了，且听我慢慢到来。为了方便研究，依然把相同的元素用上标<sup>'</sup>以示区别。

假设输入为 `nums = [1,2,2']`，标准的全排列算法会得出如下答案：

```
[  
    [1,2,2'], [1,2',2],  
    [2,1,2'], [2,2',1],  
    [2',1,2], [2',2,1]  
]
```

显然，这个结果存在重复，比如 `[1,2,2']` 和 `[1,2',2]` 应该只被算作同一个排列，但被算作了两个不同的排列。

所以现在关键在于，如何设计剪枝逻辑，把这种重复去除掉？

答案是，保证相同元素在排列中的相对位置保持不变。

比如说 `nums = [1,2,2']` 这个例子，我保持排列中 `2` 一直在 `2'` 前面。

这样的话，你从上面 6 个排列中只能挑出 3 个排列符合这个条件：

```
[ [1,2,2'], [2,1,2'], [2,2',1] ]
```

这也就是正确答案。

进一步，如果 `nums = [1,2,2',2'']`，我只要保证重复元素 `2` 的相对位置固定，比如说 `2 -> 2' -> 2''`，也可以得到无重复的全排列结果。

仔细思考，应该很容易明白其中的原理：

标准全排列算法之所以出现重复，是因为把相同元素形成的排列序列视为不同的序列，但实际上它们应该是相同的；而如果固定相同元素形成的序列顺序，当然就避免了重复。

那么反映到代码上，你注意看这个剪枝逻辑：

```
// 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置  
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {  
    // 如果前面的相邻相等元素没有用过，则跳过  
    continue;  
}  
// 选择 nums[i]
```

当出现重复元素时，比如输入 `nums = [1,2,2',2'']`，`2'` 只有在 `2` 已经被使用的情况下才会被选择，同理，`2''` 只有在 `2'` 已经被使用的情况下才会被选择，这就保证了相同元素在排列中的相对位置保证固定。

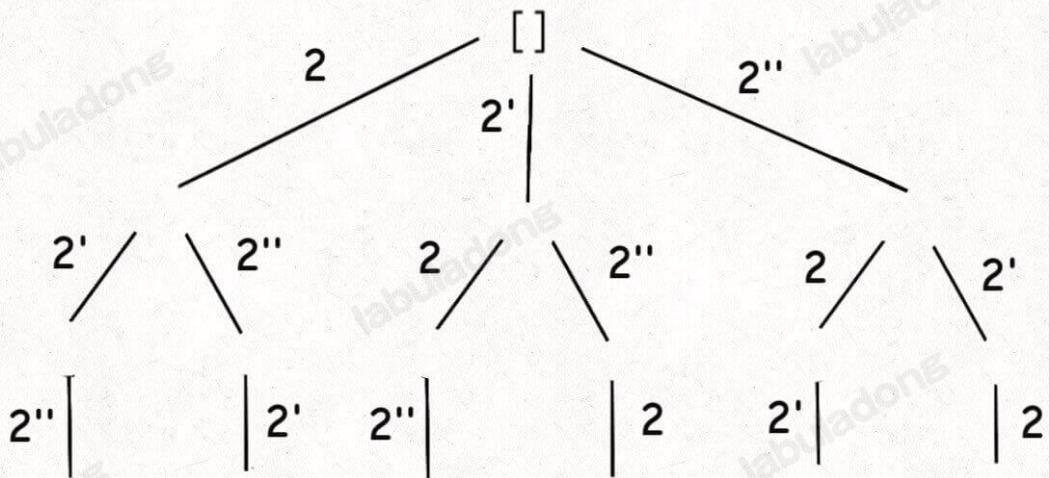
这里拓展一下，如果你把上述剪枝逻辑中的 `!used[i - 1]` 改成 `used[i - 1]`，其实也可以通过所有测试用例，但效率会有所下降，这是为什么呢？

之所以这样修改不会产生错误，是因为这种写法相当于维护了 `2'' -> 2' -> 2` 的相对顺序，最终也可以实现去重的效果。

但为什么这样写效率会下降呢？因为这个写法剪掉的树枝不够多。

比如输入 `nums = [2,2',2'']`，产生的回溯树如下：

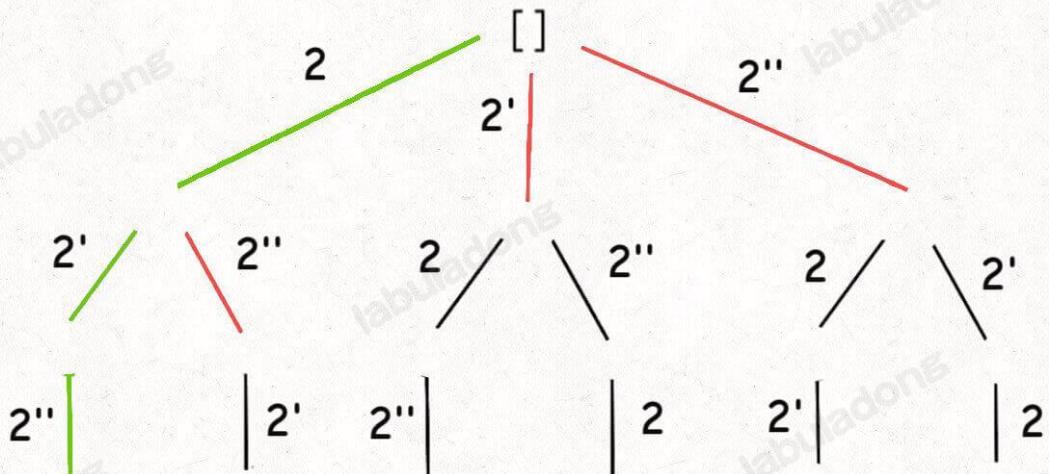
## 排列树



© labuladong

如果用绿色树枝代表 `backtrack` 函数遍历过的路径，红色树枝代表剪枝逻辑的触发，那么 `!used[i - 1]` 这种剪枝逻辑得到的回溯树长这样：

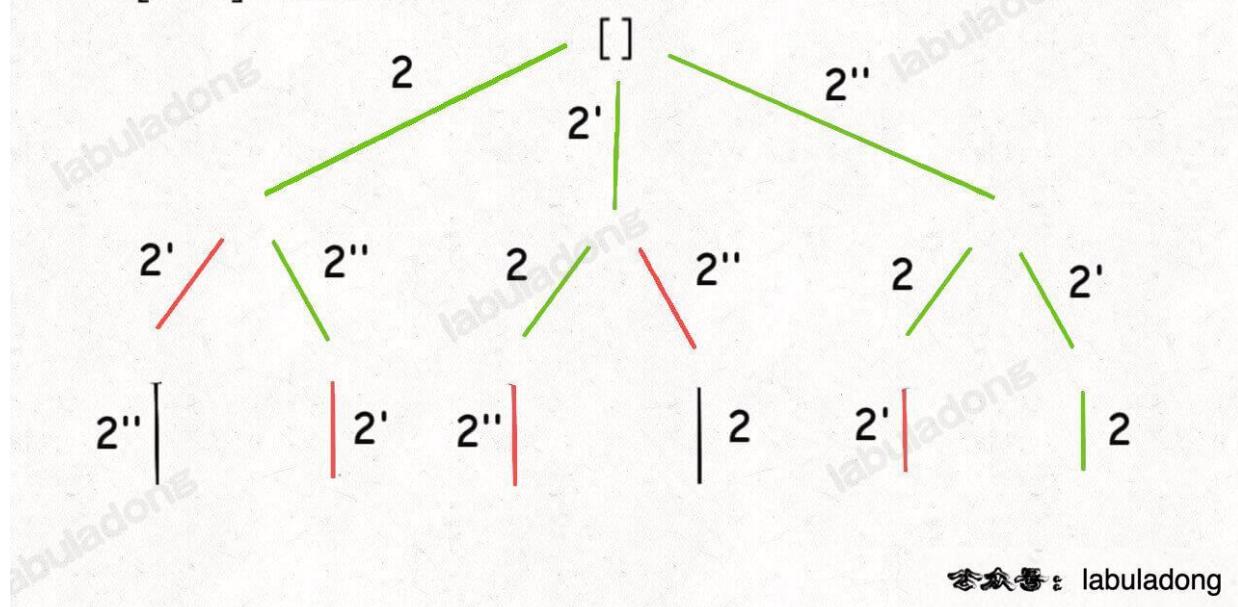
### `!used[i - 1]` 剪枝情况



© labuladong

而 `used[i - 1]` 这种剪枝逻辑得到的回溯树如下：

`used[i - 1]` 剪枝情况



可以看到，`!used[i - 1]` 这种剪枝逻辑剪得干净利落，而 `used[i - 1]` 这种剪枝逻辑虽然也能得到无重结果，但它剪掉的树枝较少，存在的无效计算较多，所以效率会差一些。

你可以使用可视化面板的「编辑」按钮自行修改代码验证一下，看看两种写法产生的回溯树有何差别：

▶ 代码可视化动画

当然，关于排列去重，也有读者提出别的剪枝思路：

```
void backtrack(int[] nums, LinkedList<Integer> track) {
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    // 记录之前树枝上元素的值
    // 题目说 -10 <= nums[i] <= 10, 所以初始化为特殊值
    int prevNum = -666;
    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            continue;
        }
        if (nums[i] == prevNum) {
            continue;
        }

        track.add(nums[i]);
        used[i] = true;
        // 记录这条树枝上的值
        prevNum = nums[i];

        backtrack(nums, track);

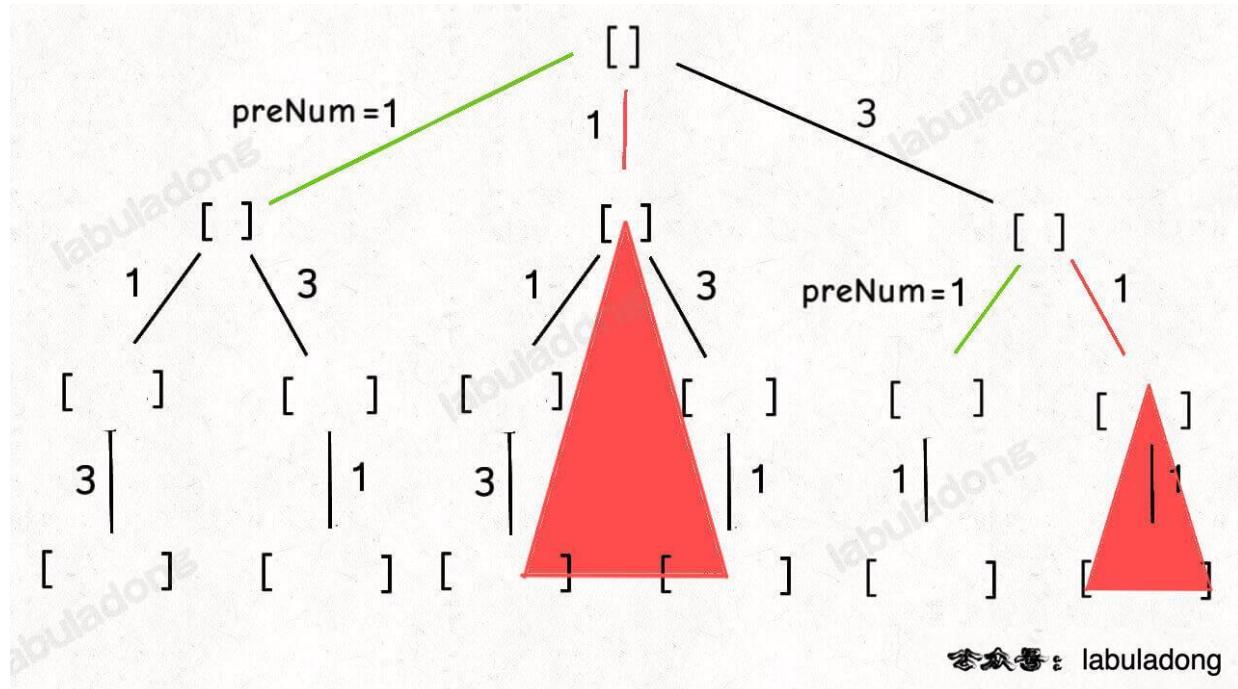
        track.removeLast();
        used[i] = false;
    }
}
```

```

    }
}

```

这个思路也是对的，设想一个节点出现了相同的树枝：



如果不作处理，这些相同树枝下面的子树也会长得一模一样，所以会出现重复的排列。

因为排序之后所有相等的元素都挨在一起，所以只要用 `prevNum` 记录前一条树枝的值，就可以避免遍历值相同的树枝，从而避免产生相同的子树，最终避免出现重复的排列。

好了，这样包含重复输入的排列问题也解决了。

## 子集/组合（元素无重可复选）

终于到了最后一种类型了：输入数组无重复元素，但每个元素可以被无限次使用。

直接看力扣第 39 题「组合总和」：

给你一个无重复元素的整数数组 `candidates` 和一个目标和 `target`，找出 `candidates` 中可以使数字和为目标数 `target` 的所有组合。`candidates` 中的每个数字可以无限制重复被选取。

函数签名如下：

```
List<List<Integer>> combinationSum(int[] candidates, int target)
```

比如输入 `candidates = [1,2,3]`, `target = 3`，算法应该返回：

```
[ [1,1,1], [1,2], [3] ]
```

这道题说是组合问题，实际上也是子集问题：`candidates` 的哪些子集的和为 `target`？

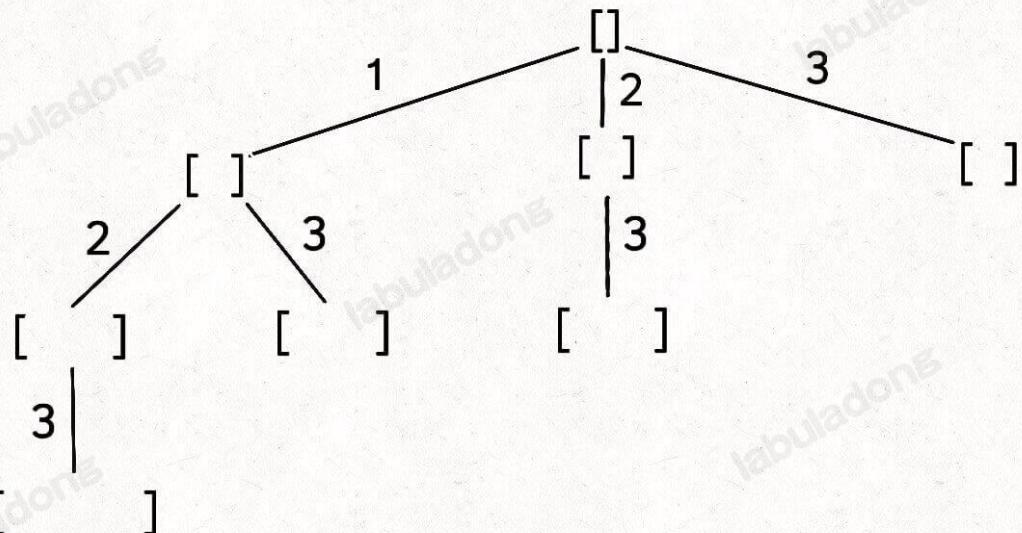
想解决这种类型的问题，也得回到回溯树上，我们不妨先思考思考，标准的子集/组合问题是如何保证不重复使用元素的？

答案在于 `backtrack` 递归时输入的参数 `start`:

```
// 无重组合的回溯算法框架
void backtrack(int[] nums, int start) {
    for (int i = start; i < nums.length; i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i + 1);
        // ...
    }
}
```

这个 `i` 从 `start` 开始，那么下一层回溯树就是从 `start + 1` 开始，从而保证 `nums[start]` 这个元素不会被重复使用：

### 组合/子集树

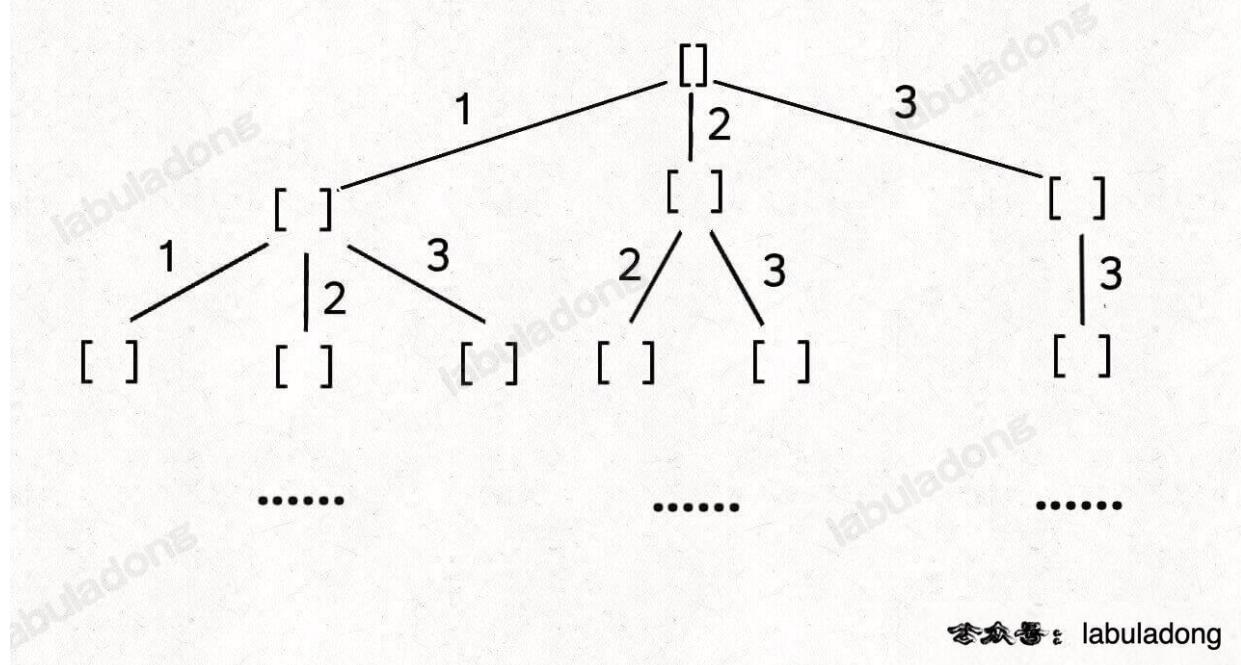


© labuladong

那么反过来，如果我想让每个元素被重复使用，我只要把 `i + 1` 改成 `i` 即可：

```
// 可重组合的回溯算法框架
void backtrack(int[] nums, int start) {
    for (int i = start; i < nums.length; i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i);
        // ...
    }
}
```

这相当于给之前的回溯树添加了一条树枝，在遍历这棵树的过程中，一个元素可以被无限次使用：



当然，这样这棵回溯树会永远生长下去，所以我们的递归函数需要设置合适的 base case 以结束算法，即路径和大于 `target` 时就没必要再遍历下去了。

这道题的解法代码如下：

```

class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();
    // 记录 track 中的路径和
    int trackSum = 0;

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        if (candidates.length == 0) {
            return res;
        }
        backtrack(candidates, 0, target);
        return res;
    }

    // 回溯算法主函数
    void backtrack(int[] nums, int start, int target) {
        // base case, 找到目标和, 记录结果
        if (trackSum == target) {
            res.add(new LinkedList<>(track));
            return;
        }
        // base case, 超过目标和, 停止向下遍历
        if (trackSum > target) {
            return;
        }
        // 回溯算法标准框架
        for (int i = start; i < nums.length; i++) {
            // 选择 nums[i]
            trackSum += nums[i];
            track.add(nums[i]);
        }
    }
}

```

```
// 递归遍历下一层回溯树
backtrack(nums, i, target);
// 同一元素可重复使用，注意参数
// 撤销选择 nums[i]
trackSum -= nums[i];
track.removeLast();
}
}
}
```

---

▶ 🎥 代码可视化动画

## 排列（元素无重可复选）

力扣上没有题目直接考察这个场景，我们不妨先想一下，`nums` 数组中的元素无重复且可复选的情况下，会有哪些排列？

比如输入 `nums = [1,2,3]`，那么这种条件下的全排列共有  $3^3 = 27$  种：

```
[  
    [1,1,1], [1,1,2], [1,1,3], [1,2,1], [1,2,2], [1,2,3], [1,3,1], [1,3,2], [1,3,3],  
    [2,1,1], [2,1,2], [2,1,3], [2,2,1], [2,2,2], [2,2,3], [2,3,1], [2,3,2], [2,3,3],  
    [3,1,1], [3,1,2], [3,1,3], [3,2,1], [3,2,2], [3,2,3], [3,3,1], [3,3,2], [3,3,3]  
]
```

标准的全排列算法利用 `used` 数组进行剪枝，避免重复使用同一个元素。如果允许重复使用元素的话，直接放飞自我，去除所有 `used` 数组的剪枝逻辑就行了。

那这个问题就简单了，代码如下：

```
class Solution {  
  
    List<List<Integer>> res = new LinkedList<>();  
    LinkedList<Integer> track = new LinkedList<>();  
  
    public List<List<Integer>> permuteRepeat(int[] nums) {  
        backtrack(nums);  
        return res;  
    }  
  
    // 回溯算法核心函数  
    void backtrack(int[] nums) {  
        // base case, 到达叶子节点  
        if (track.size() == nums.length) {  
            // 收集叶子节点上的值  
            res.add(new LinkedList(track));  
            return;  
        }  
  
        // 回溯算法标准框架  
        for (int i = 0; i < nums.length; i++) {  
            // 做选择  
            track.add(nums[i]);  
            // 进入下一层回溯树  
        }  
    }  
}
```

```
        backtrack(nums);
        // 取消选择
        track.removeLast();
    }
}
}
```

至此，排列/组合/子集问题的九种变化就都讲完了。

## 最后总结

来回顾一下排列/组合/子集问题的三种形式在代码上的区别。

由于子集问题和组合问题本质上是一样的，无非就是 base case 有一些区别，所以把这两个问题放在一起看。

**形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，`backtrack` 核心代码如下：**

```
// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i + 1);
        // 撤销选择
        track.removeLast();
    }
}

// 排列问题回溯算法框架
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 撤销选择
        track.removeLast();
        used[i] = false;
    }
}
```

**形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次，其关键在于排序和剪枝，`backtrack` 核心代码如下：**

```
Arrays.sort(nums);
// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
```

```

// 回溯算法标准框架
for (int i = start; i < nums.length; i++) {
    // 剪枝逻辑，跳过值相同的相邻树枝
    if (i > start && nums[i] == nums[i - 1]) {
        continue;
    }
    // 做选择
    track.addLast(nums[i]);
    // 注意参数
    backtrack(nums, i + 1);
    // 撤销选择
    track.removeLast();
}

Arrays.sort(nums);
// 排列问题回溯算法框架
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 剪枝逻辑，固定相同的元素在排列中的相对位置
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 撤销选择
        track.removeLast();
        used[i] = false;
    }
}

```

形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次，只要删掉去重逻辑即可，`backtrack` 核心代码如下：

```

// 组合/子集问题回溯算法框架
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i);
        // 撤销选择
        track.removeLast();
    }
}

// 排列问题回溯算法框架
void backtrack(int[] nums) {

```

```
for (int i = 0; i < nums.length; i++) {  
    // 做选择  
    track.addLast(nums[i]);  
    backtrack(nums);  
    // 撤销选择  
    track.removeLast();  
}  
}
```

只要从树的角度思考，这些问题看似复杂多变，实则改改 base case 就能解决，这也是为什么我在 [学习算法和数据结构的框架思维](#) 和 [手把手刷二叉树（纲领篇）](#) 中强调树类型题目重要性的原因。

如果你能够看到这里，真得给你鼓掌，相信你以后遇到各种乱七八糟的算法题，也能一眼看透它们的本质，以不变应万变。另外，考虑到篇幅，本文并没有对这些算法进行复杂度的分析，你可以使用我在 [算法时空复杂度分析实用指南](#) 讲到的复杂度分析方法尝试自己分析它们的复杂度。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1079. Letter Tile Possibilities</a>	<a href="#">1079. 活字印刷</a>	
<a href="#">131. Palindrome Partitioning</a>	<a href="#">131. 分割回文串</a>	
<a href="#">17. Letter Combinations of a Phone Number</a>	<a href="#">17. 电话号码的字母组合</a>	
<a href="#">254. Factor Combinations</a>	<a href="#">254. 因子的组合</a>	
<a href="#">267. Palindrome Permutation II</a>	<a href="#">267. 回文排列 II</a>	
<a href="#">368. Largest Divisible Subset</a>	<a href="#">368. 最大整除子集</a>	
<a href="#">491. Non-decreasing Subsequences</a>	<a href="#">491. 递增子序列</a>	
<a href="#">638. Shopping Offers</a>	<a href="#">638. 大礼包</a>	
<a href="#">967. Numbers With Same Consecutive Differences</a>	<a href="#">967. 连续差相同的数字</a>	
<a href="#">996. Number of Squareful Arrays</a>	<a href="#">996. 正方形数组的数目</a>	
-	<a href="#">剑指 Offer 38. 字符串的排列</a>	
-	<a href="#">剑指 Offer II 079. 所有子集</a>	
-	<a href="#">剑指 Offer II 080. 含有 k 个元素的组合</a>	
-	<a href="#">剑指 Offer II 081. 允许重复选择元素的组合</a>	
-	<a href="#">剑指 Offer II 083. 没有重复元素集合的全排列</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 球盒模型：回溯算法穷举的两种视角



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">46. Permutations</a>	<a href="#">46. 全排列</a>	
<a href="#">78. Subsets</a>	<a href="#">78. 子集</a>	

阅读本文前，你需要先学习：

- [二叉树系列算法（纲领篇）](#)
- [回溯算法核心框架](#)
- [回溯算法秒杀所有排列/组合/子集问题](#)

阅读本文之前，需要你熟悉 [回溯算法核心框架](#) 以及 [回溯算法秒杀所有排列/组合/子集问题](#)。

在上面这两篇文章中，有读者提出了不同的排列/组合/子集代码写法，比如通过 `swap` 元素实现全排列，还有没有 `for` 循环的子集解法代码。我之前不提这些不同的解法，是为了保持这些问题解法形式的一致性，如果一开始就给大家太多选择，反而容易让人迷糊。

在这篇文章，我不仅会具体介绍之前没有讲到的回溯算法写法，还会告诉你为什么可以那样写，两种写法的本质区别是什么。

1、回溯算法穷举的本质思维模式是「球盒模型」，一切回溯算法，皆从此出，别无二法。

2、球盒模型，必然有两种穷举视角，分别为「球」的视角穷举和「盒」的视角穷举，对应的，就是两种不同的代码写法。

3、从理论上分析，两种穷举视角本质上是一样的。但是涉及到具体的代码实现，两种写法的复杂度可能有优劣之分。你需要选择效率更高的写法。

球盒模型这个词是我随口编的，因为下面我会用「球」和「盒」两种视角来解释，你理解就好。

## 暴力穷举思维方法：球盒模型

一切暴力穷举算法，都从球盒模型开始，没有例外。

你懂了这个，就可以随心所欲运用暴力穷举算法，下面的内容，请你仔细看，认真想。

首先，我们回顾一下以前学过的排列组合知识：

1、 $P(n, k)$ （也有很多书写成  $A(n, k)$ ）表示从  $n$  个不同元素中拿出  $k$  个元素的排列（Permutation/Arrangement）总数； $C(n, k)$  表示从  $n$  个不同元素中拿出  $k$  个元素的组合（Combination）总数。

2、「排列」和「组合」的主要区别在于是否考虑顺序的差异。

3、排列、组合总数的计算公式：

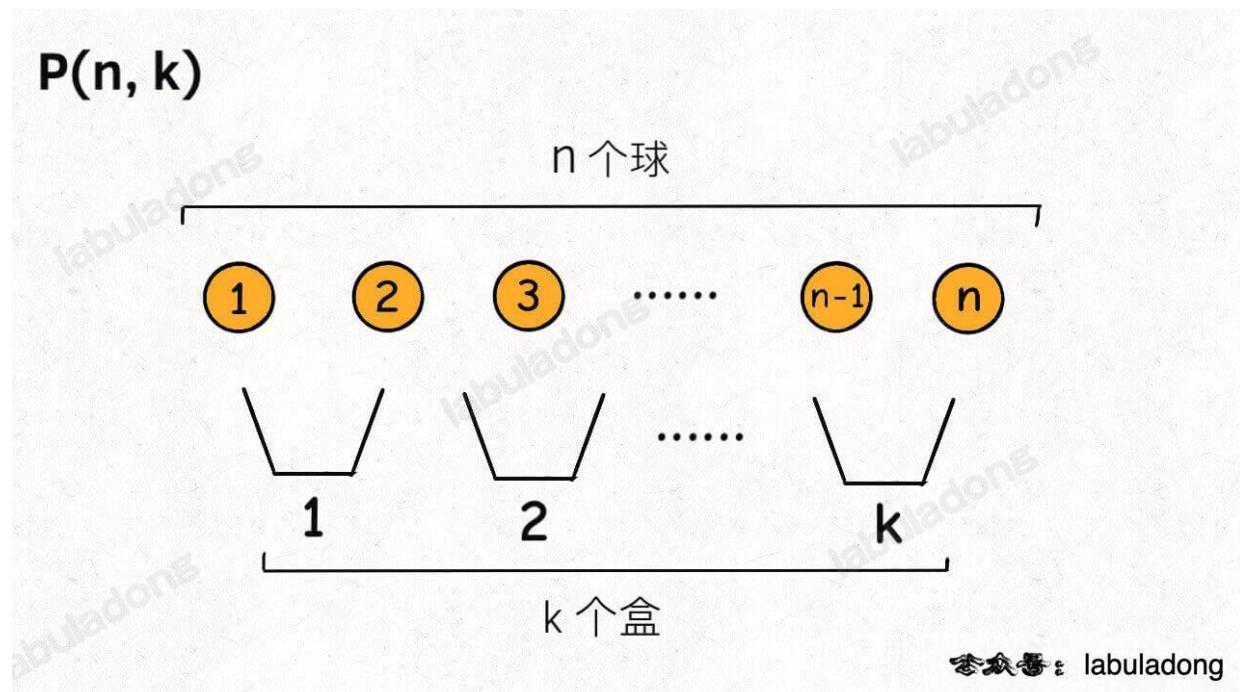
$$P(n, k) = \frac{n!}{(n - k)!}$$

$$C(n, k) = \frac{n!}{k!(n - k)!}$$

## 排列 $P(n, k)$

好，现在我问一个问题，这个排列公式  $P(n, k)$  是如何推导出来的？为了搞清楚这个问题，我需要讲一点组合数学的知识。

排列组合问题的各种变体都可以抽象成「球盒模型」， $P(n, k)$  就可以抽象成下面这个场景：

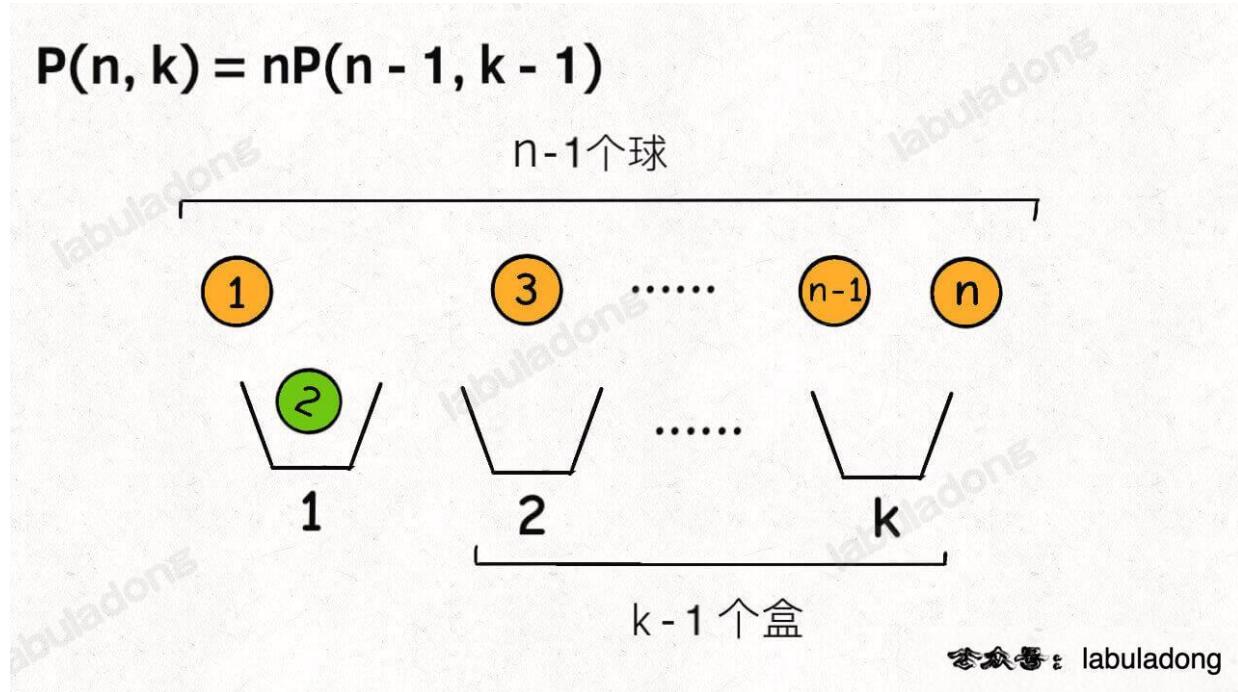


即，将  $n$  个标记了不同序号的球（标号为了体现顺序的差异），放入  $k$  个标记了不同序号的盒子中（其中  $n \geq k$ ，每个盒子最终都装有恰好一个球），共有  $P(n, k)$  种不同的方法。

现在你来，往盒子里放球，你怎么放？其实有两种视角。

首先，你可以站在盒子的视角，每个盒子必然要选择一个球。

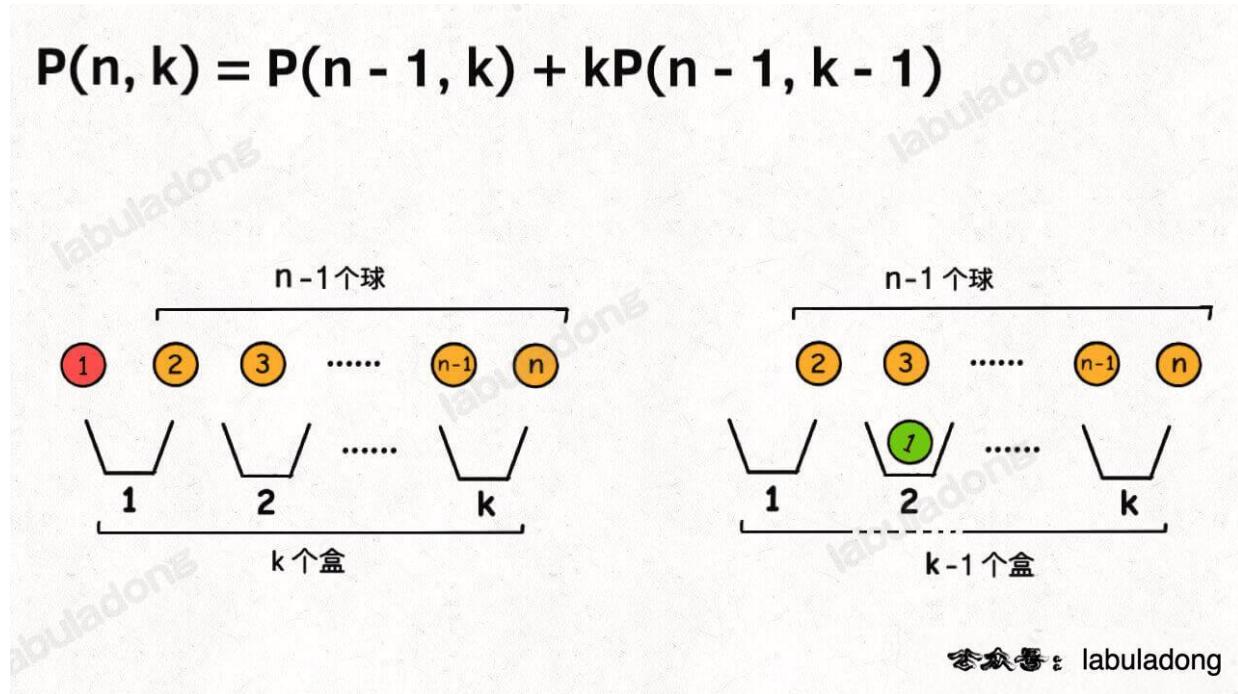
这样，第一个盒子可以选择  $n$  个球中的任意一个，然后你需要让剩下  $k - 1$  个盒子在  $n - 1$  个球中选择（这就是子问题  $P(n - 1, k - 1)$ ）：



另外，你也可以站在球的视角，因为并不是每个球都会被装进盒子，所以球的视角分两种情况：

- 1、第一个球可以不装进任何一个盒子，这样的话你就需要将剩下  $n - 1$  个球放入  $k$  个盒子。
- 2、第一个球可以装进  $k$  个盒子中的任意一个，这样的话你就需要将剩下  $n - 1$  个球放入  $k - 1$  个盒子。

结合上述两种情况，可以得到：



你看，两种视角得到两个不同的递归式，但这两个递归式解开的结果都是我们熟知的阶乘形式：

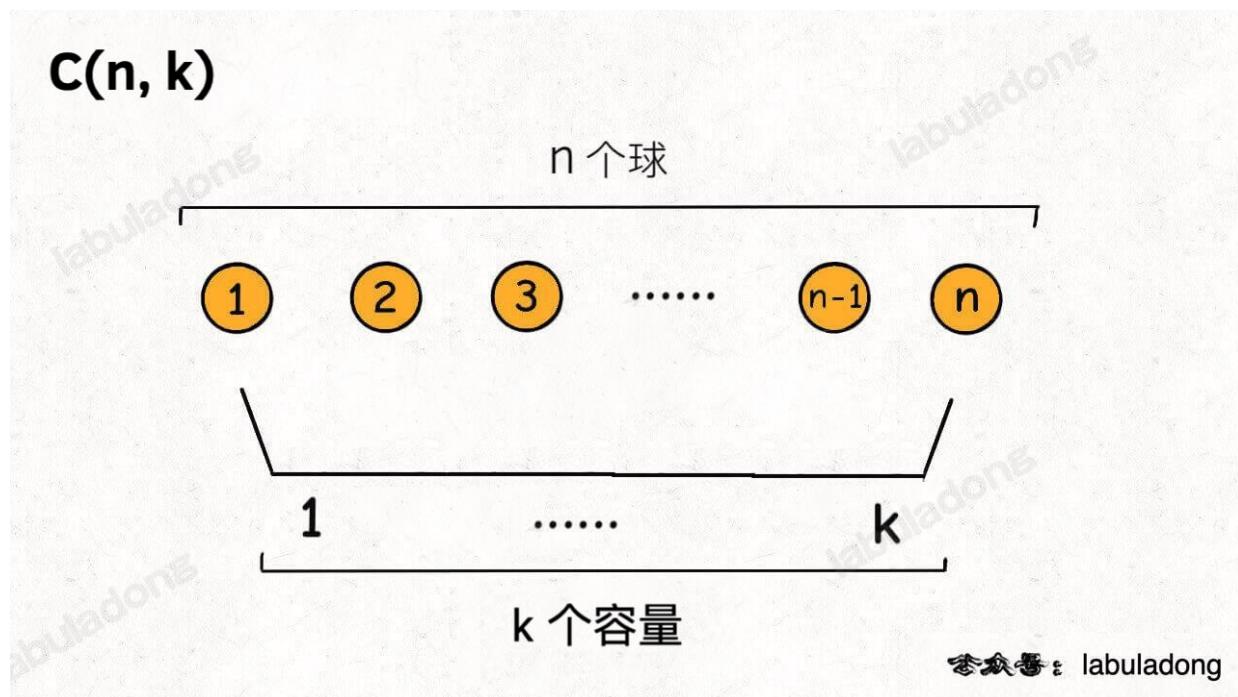
$$\begin{aligned}
 P(n, k) &= nP(n - 1, k - 1) \\
 &= P(n - 1, k) + kP(n - 1, k - 1) \\
 &= \frac{n!}{(n - k)!}
 \end{aligned}$$

至于如何解递归式，涉及数学的内容比较多，这里就不做深入探讨了，有兴趣的读者可以自行学习组合数学相关知识。

## 组合 $C(n, k)$

了解了排列的推导过程，组合的推导是类似的，也是以球和盒两种视角来做选择，也可以写出两种等价形式。你不妨停在这里思考几分钟，看看你能不能自己发明出组合的推导过程。

下面我来带你推导，组合  $C(n, k)$  可以抽象成下面这个场景：



因为在组合问题中，我们不在乎元素的顺序，即 {1, 2, 3} 和 {1, 3, 2} 视为同一种组合。所以我们不需要对盒进行编号，可以认为只有一个盒，其容量是  $k$ 。

现在你来，往盒子里放球，你怎么放？还是有两种视角。

首先，你可以站在盒子的视角，盒子必然要装  $k$  个球。

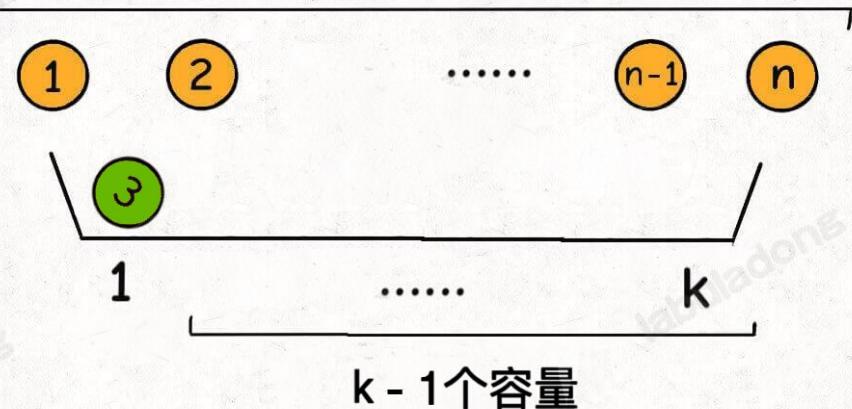
那么第一个球可以选择  $n$  个球中的任意一个，然后盒子剩余  $k - 1$  个位置，你需要在剩下的  $n - 1$  个球中选择（这就是子问题  $C(n - 1, k - 1)$ ）。

想到这里，是不是就想列出关系式  $C(n, k) = nC(n - 1, k - 1)$  了？

不对，这里有个坑，你直接  $nC(n - 1, k - 1)$  是有重复的，正确的答案应该是  $nC(n - 1, k - 1) / k$ ：

$$C(n, k) = nC(n-1, k-1) / k$$

n - 1 个球



© labuladong

举例来说就容易看出来了，比方说让你在  $[1, 2, 3, 4, 5]$  中选择 3 个元素，有多少种不同的组合？

你站在盒的视角，盒子容量为 3，开始穷举，第一个位置，你可以选择球  $1, 2, 3, 4, 5$  中的任意一个，比如说选了球 1。

接下来你需要在剩下的元素  $2, 3, 4, 5$  中选择两个元素，比方说最终你穷举出的组合是  $\{1, 3, 4\}$ 。

要想知道是否有重复，重复了几次，你就盯紧这个  $\{1, 3, 4\}$ ，它如果有重复，其他组合结果都会重复。

容易发现，如果我第一次选择的是球 3，然后我可能得到  $\{3, 1, 4\}$ ；如果我第一次选择的是球 4，那么我可能得到  $\{4, 1, 3\}$ ；这两种情况都是和  $\{1, 3, 4\}$  重复的，因此  $\{1, 3, 4\}$  共出现了三次。

有的读者会问，只有这两种情况是重复的吗？类似  $\{3, 4, 1\}$  这种情况不也是重复的吗？

不是，因为我们组合不考虑顺序， $\{3, 4, 1\}$  和  $\{3, 1, 4\}$  都是球 3 开头，所以是等价的。

综上，你不能直接用  $nC(n - 1, k - 1)$ ，因为其中每种组合都出现了  $k$  次，所以要再除以  $k$  消除重复。

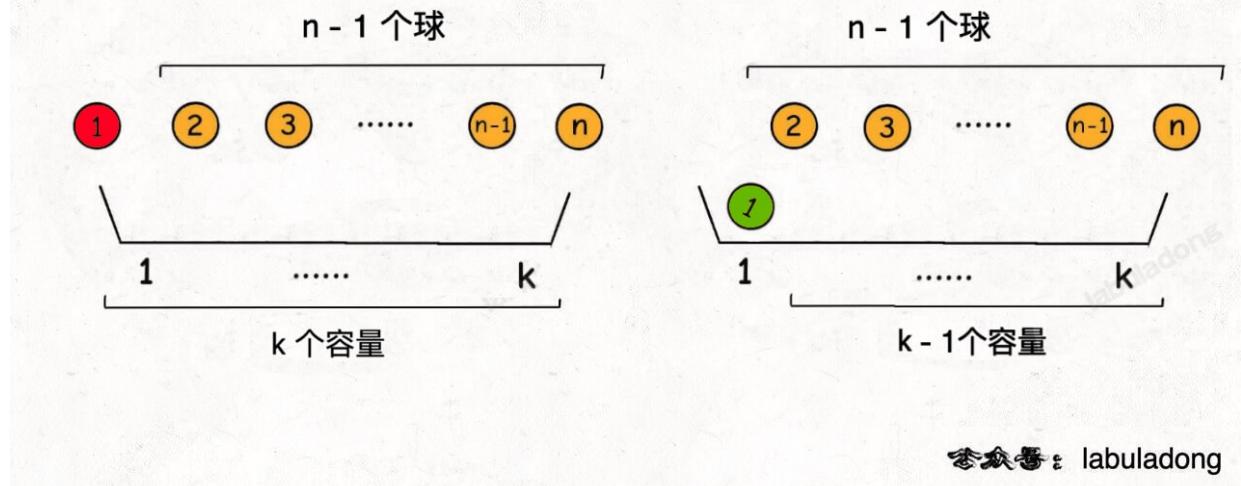
另外，你也可以站在球的视角，因为并不是每个球都会被装进盒子，所以球的视角分两种情况：

1、第一个球可以不装进盒子，这样的话你就需要将剩下  $n - 1$  个球放入  $k$  个盒子。

2、第一个球可以装进盒子，这样的话你就需要将剩下  $n - 1$  个球放入  $k - 1$  个盒子。

结合上述两种情况，可以得到：

$$C(n, k) = C(n-1, k) + C(n-1, k-1)$$



© labuladong

注意组合和前面排列的区别，因为组合不考虑顺序，所以球选择装进盒子时，只有一种选择，而不是  $k$  种不同选择，所以  $C(n-1, k-1)$  不用乘  $k$ 。

你看，两种视角得到两个不同的递归式，但这两个递归式解开的结果都是我们熟知的组合数公式：

$$\begin{aligned} C(n, k) &= C(n - 1, k) + C(n - 1, k - 1) \\ &= \frac{n \cdot C(n - 1, k - 1)}{k} \\ &= \frac{n!}{k!(n - k)!} \end{aligned}$$

至于如何解递归式，涉及数学的内容比较多，这里就不做深入探讨了。

上面的内容，我带你从头推导了排列组合的数学公式，但数学不是我的重点，重点在于，你有没有对「穷举」这个事情有了更深的理解？

## 用球盒模型重新理解全排列问题

好，上面从数学的角度介绍了全排列穷举的两种视角，现在回归到代码上，我要考你了哦。

前文 [回溯算法核心框架](#) 和 [回溯算法秒杀排列/组合/子集的九种变体](#) 都给出过全排列的代码。

就以最基本的元素无重不可复选的全排列为例，我直接把代码 copy 过来：

```

class Solution {

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();
    // track 中的元素会被标记为 true
    boolean[] used;

    // 主函数，输入一组不重复的数字，返回它们的全排列
    public List<List<Integer>> permute(int[] nums) {
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }

    // 回溯算法核心函数
    void backtrack(int[] nums) {
        // base case, 到达叶子节点
        if (track.size() == nums.length) {
            // 收集叶子节点上的值
            res.add(new LinkedList(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = 0; i < nums.length; i++) {
            // 已经存在 track 中的元素，不能重复选择
            if (used[i]) {
                continue;
            }
            // 做选择
            used[i] = true;
            track.addLast(nums[i]);
            // 进入下一层回溯树
            backtrack(nums);
            // 取消选择
            track.removeLast();
            used[i] = false;
        }
    }
}

```

请问，这个解法是以什么视角进行穷举的？是以球的视角还是盒的视角？给你三分钟思考，请回答！

这个代码是以盒的视角进行穷举的，即站在每个位置的角度来选择球，站在 `nums` 中的每个索引，来选择不同的元素放入这个索引位置。

为什么是这个答案呢？假设 `nums` 里面有  $n$  个数字，那么全排列问题相当于把  $n$  个球放到包含  $n$  个位置的盒子里，要求盒子必须装满，问你有几种不同的装法。

以盒的视角理解，盒子的第一个位置可以接收  $n$  个球的任意一个，有  $n$  种选择，第二个位置可以接收  $n - 1$  个球的任意一个，有  $n - 1$  种选择，第三个位置有  $n - 2$  种选择，以此类推。

我直接用 [算法可视化面板](#) 把递归树画出来，你一眼就可以看懂了。请你把进度条拖到最后让整棵回溯树显示出来，然后把鼠标在每一层节点上横向移动，观察递归树节点和树枝上的值：

▶   代码可视化动画 

我在 [回溯算法核心框架](#) 和 [回溯算法秒杀排列/组合/子集的九种变体](#) 中都写了上面这段代码，很多读者看了之后就跑来跟我说啊，他看的那个全排列算法是通过 `swap` 操作来计算的，不需要 `used` 数组的额外空间，比我讲解的回溯算法框架效率高，怎么怎么的。

是的，我之所以不用那个 `swap` 的解法，是因为前面那两篇文章的重点在于实践回溯算法「做选择」和「撤销选择」的思维框架，用 `used` 数组的解法更容易让初学者理解。但从算法效率上说，确实有更高效的代码实现方法。

下面就满足大家的好奇心，跟大家讲讲那个传说中的 `swap` 的解法，到底是何方神圣。

首先，我列出那个使用 `swap` 计算全排列的解法代码，请你先看一下：

```
class Solution {

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

    public List<List<Integer>> permute(int[] nums) {
        backtrack(nums, 0);
        return result;
    }

    // 回溯算法核心框架
    void backtrack(int[] nums, int start) {
        if (start == nums.length) {
            // 找到一个全排列，Java 需要转化成 List 类型
            List<Integer> list = new ArrayList<>();
            for (int num : nums) {
                list.add(num);
            }
            result.add(list);
            return;
        }

        for (int i = start; i < nums.length; i++) {
            // 做选择
            swap(nums, start, i);
            // 递归调用，传入 start + 1
            backtrack(nums, start + 1);
            // 撤销选择
            swap(nums, start, i);
        }
    }

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

这个解法也可以正确计算全排列，请你思考，这段代码是以什么视角进行穷举的？是以球的视角还是盒的视角？

答案是，这个解法是以盒的视角进行穷举的。即 `nums` 数组中的每个索引位置，来选择不同的元素放入这个索引位置。

你看解法代码也可以看出来，那个 `start` 参数就是当前在选择元素的索引位置，在 `start` 之前的元素已经心有所属，被其他位置挑走了，所以 `start` 位置只能从 `nums[start..]` 中选择元素。

我可以用 [算法可视化面板](#) 把递归树画出来，你一眼就可以看懂了。请你把进度条拖到最后让整棵回溯树显示出来，然后把鼠标在每一层节点上横向移动，观察递归树节点和树枝上的值：

► 代码可视化动画

接下来一个很自然的问题，能不能写出一个以球的视角理解的全排列问题的解法？

当然可以，以球的视角来写全排列的解法代码，就是说 `nums` 中的每个元素来选择自己想去的索引，对吧。有了这个思路，代码还有何难写。

我先用 [算法可视化面板](#) 把递归树画出来，请你把进度条拖到最后让整棵回溯树显示出来，然后把鼠标在每一层节点上横向移动，观察递归树节点和树枝上的值，验证一下是不是元素在选索引：

► 代码可视化动画

当然我写的代码还有一些小优化的空间，比如说这个 `swapIndex` 其实就是 `i`，而且我们其实不用等到 `count == nums.length`，当 `count == nums.length - 1` 时就可以 `return` 了，因为最后剩的那个元素的位置不会找不到其他位置了。这些留给你优化吧。

```
class Solution {
    // 结果列表
    List<List<Integer>> res;
    // 标记元素是否已被使用
    boolean[] used;
    // 记录有多少个元素已经选择过位置
    int count;

    public List<List<Integer>> permute(int[] nums) {
        res = new ArrayList<>();
        used = new boolean[nums.length];
        count = 0;

        backtrack(nums);
        return res;
    }

    // 回溯算法框架
    void backtrack(int[] nums) {
        if (count == nums.length) {
            List<Integer> temp = new ArrayList<>();
            for (int num : nums) {
                temp.add(num);
            }
        }
    }
}
```

```
        res.add(temp);
        return;
    }

    // 找两个未被选择的位置
    int originalIndex = -1, swapIndex = -1;

    for (int i = 0; i < nums.length; i++) {
        if (used[i]) {
            continue;
        }
        if (originalIndex == -1) {
            originalIndex = i;
        }
        swapIndex = i;

        // 做选择, 元素 nums[originalIndex] 选择 swapIndex 位置
        swap(nums, originalIndex, swapIndex);
        used[swapIndex] = true;
        count++;
        // 进入下一层决策树
        backtrack(nums);
        // 撤销选择, 刚才怎么做的选择, 就原样恢复
        count--;
        used[swapIndex] = false;
        swap(nums, originalIndex, swapIndex);
    }
}

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

## 用球盒模型重新理解子集问题

有了前面的铺垫, 我又要进一步为难你了。[回溯算法秒杀排列/组合/子集的九种变体](#) 都给出过子集问题的代码。

就以最基本的元素无重不可复选的子集为例, 我直接把代码 copy 过来:

```
class Solution {
    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯算法的递归路径
    LinkedList<Integer> track = new LinkedList<>();

    // 主函数
    public List<List<Integer>> subsets(int[] nums) {
        backtrack(nums, 0);
        return res;
    }

    // 回溯算法核心函数, 遍历子集问题的回溯树
    void backtrack(int[] nums, int start) {
        // 前序位置, 每个节点的值都是一个子集
    }
}
```

```
res.add(new LinkedList<>(track));  
  
    // 回溯算法标准框架  
    for (int i = start; i < nums.length; i++) {  
        // 做选择  
        track.addLast(nums[i]);  
        // 通过 start 参数控制树枝的遍历，避免产生重复的子集  
        backtrack(nums, i + 1);  
        // 撤销选择  
        track.removeLast();  
    }  
}  
}
```

请问，这个解法是以什么视角进行穷举的？是以球的视角还是盒的视角？给你三分钟思考，请回答！

这个解法是以盒的视角穷举的，即站在 `nums` 中的每个索引的视角，来选择不同的元素放入这个索引位置。

因为刚才讲的全排列问题会考虑顺序的差异，而子集问题不考虑顺序的差异。为了方便理解，我们这里干脆不说「球盒模型」了，说「球桶模型」吧，因为放进盒子的求给人感觉是有顺序的，而丢进桶里的东西给人感觉是无所谓顺序的。

那么，以桶的视角理解，子集问题相当于把 `n` 个球丢到容量为 `n` 的桶里，桶可以不装满。

这样，桶的第一个位置可以选择 `n` 个球中的任意一个，比如选择了球 `i`，然后桶的第二个位置可以选择球 `i` 后面的球中的任意一个（通过固定相对顺序保证不会出现重复的子集），以此类推。

你看代码也能体现出来这种穷举过程：

```
// 回溯算法框架核心代码  
void backtrack(int[] nums, int start) {  
    for (int i = start; i < nums.length; i++) {  
        track.addLast(nums[i]);  
        // 通过 start 参数控制树枝的生长  
        backtrack(nums, i + 1);  
        track.removeLast();  
    }  
}
```

我继续用 [算法可视化面板](#) 来论证我的答案，请你把进度条拖到最后让整棵回溯树显示出来，然后把鼠标在每一层节点上横向移动，观察递归树节点和树枝上的值，你可以很直观地看明白，是桶的位置在选择球：

▶  [代码可视化动画](#) 

既然上面说了，我给的子集问题解法是以桶的视角理解的，那么你能不能写出一个以球的视角理解的子集问题的解法？给你十分钟写代码。

如果你有这个时间，一定要亲自动手尝试一下，不要着急看我的答案。你能认真看到这里，肯定可以写出来的，不要怀疑。

从球的视角理解，每个球都有两种选择，要么在桶中，要么不在桶中。这样，我们可以写出下面的代码：

```
class Solution {
    // 用于存储所有子集的结果
    List<List<Integer>> res;
    // 用于存储当前递归路径的子集
    List<Integer> track;

    public List<List<Integer>> subsets(int[] nums) {
        res = new ArrayList<>();
        track = new ArrayList<>();
        backtrack(nums, 0);
        return res;
    }

    void backtrack(int[] nums, int i) {
        if (i == nums.length) {
            res.add(new ArrayList<>(track));
            return;
        }

        // 做第一种选择，元素在子集中
        track.add(nums[i]);
        backtrack(nums, i + 1);
        // 撤销选择
        track.remove(track.size() - 1);

        // 做第二种选择，元素不在子集中
        backtrack(nums, i + 1);
    }
}
```

我继续用 [算法可视化面板](#) 来论证我的答案，请你把进度条拖到最后让整棵回溯树显示出来，然后把鼠标在节点上移动，观察递归树节点和树枝上的值：

### ▶ 🎃 代码可视化动画🎃

这也解释了，为什么所有子集（幂集）的数量是  $2^n$ ，因为每个元素都有两种选择，要么在子集中，要么不在子集中，所以其递归树就是一棵满二叉树，一共有  $2^n$  个叶子节点。

## 结论

照应一下开头，把几个结论再重写一遍，你现在应该更理解了。

**1、回溯算法穷举的本质思维模式是「球盒模型」，一切回溯算法，皆从此出，别无二法。**

你现在就去做 100 道回溯算法的题目，看看有没有意外，有意外你来打我。

**2、球盒模型，必然有两种穷举视角，分别为「球」的视角穷举和「盒」的视角穷举，对应的，就是两种不同的代码写法。**

暴力穷举就是如此朴实无华且枯燥，看起来花里胡哨，实则只有两种视角。

3、从理论上分析，两种穷举视角本质上是一样的。但是涉及到具体的代码实现，两种写法的复杂度可能有优劣之分。

进一步想想，为啥用「盒」的视角，即让索引去选元素的视角，可以用 `swap` 的方法把 `used` 数组给优化掉呢？

因为索引容易处理，如果你按顺序从小到大让每个索引去选元素，那么一个 `start` 变量作为分割线就能把已选元素和未选元素分开。

反过来，如果你让元素去选索引，那就只能依赖额外的数据结构来记录那些索引已经被选过了，这样就会增加额外的空间复杂度。

所以说，在开头的数学分析中，两种视角在数学上虽然是等价的，但具体到代码实现上，最优复杂度就可能不一样。

好的，最后留个悬念：只有写回溯算法时才会用到「球盒模型」这种思想吗？

你可以读一读 [动态规划算法的两种视角](#)，思考一下这个问题。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1593. Split a String Into the Max Number of Unique Substrings</a>	<a href="#">1593. 拆分字符串使唯一子字符串的数目最大</a>	
<a href="#">1723. Find Minimum Time to Finish All Jobs</a>	<a href="#">1723. 完成所有工作的最短时间</a>	
<a href="#">1849. Splitting a String Into Descending Consecutive Values</a>	<a href="#">1849. 将字符串拆分为递减的连续值</a>	
<a href="#">2850. Minimum Moves to Spread Stones Over Grid</a>	<a href="#">2850. 将石头分散到网格图的最少移动次数</a>	
<a href="#">291. Word Pattern II</a>	<a href="#">291. 单词规律 II</a>	
<a href="#">473. Matchsticks to Square</a>	<a href="#">473. 火柴拼正方形</a>	
<a href="#">526. Beautiful Arrangement</a>	<a href="#">526. 优美的排列</a>	
<a href="#">89. Gray Code</a>	<a href="#">89. 格雷编码</a>	
-	<a href="#">剑指 Offer II 079. 所有子集</a>	
-	<a href="#">剑指 Offer II 083. 没有重复元素集合的全排列</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 解答回溯算法/DFS算法的若干疑问

阅读本文前，你需要先学习：

- 二叉树系列算法（纲领篇）
- 回溯算法核心框架

本文用最简单的示例，一次性解答读者关于回溯算法、DFS 算法的若干疑问。

## 回溯算法和 DFS 算法有何区别？

经常有读者问我，网站上为啥只写了回溯算法，却没有写过 DFS 算法呢？

还有些读者有疑问，[回溯算法核心框架](#) 中说到回溯算法模板是在递归前做选择，递归后撤销选择，即这样：

```
void backtrack(...) {
    if (reached the leaf node) {
        // 到达叶子节点，结束递归
        return;
    }

    for (int i = 0; i < n; i++) {
        // 做选择
        ...

        backtrack(...)

        // 撤销选择
        ...
    }
}
```

但是为什么有些时候会看到代码在 for 循环的前面「做选择」，在 for 循环的后面「撤销选择」呢：

```
void backtrack(...) {
    if (reached the leaf node) {
        // 到达叶子节点，结束递归
        return;
    }
    // 做选择
    ...

    for (int i = 0, i < n; i++) {
        backtrack(...)
    }

    // 撤销选择
    ...
}
```

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 一文秒杀所有岛屿题目



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">695. Max Area of Island</a>	695. 岛屿的最大面积	
<a href="#">200. Number of Islands</a>	200. 岛屿数量	
<a href="#">1020. Number of Enclaves</a>	1020. 飞地的数量	
<a href="#">694. Number of Distinct Islands</a>	694. 不同岛屿的数量	
<a href="#">1905. Count Sub Islands</a>	1905. 统计子岛屿	
<a href="#">1254. Number of Closed Islands</a>	1254. 统计封闭岛屿的数目	

阅读本文前，你需要先学习：

- 二叉树系列算法（纲领篇）
- 回溯算法核心框架
- 关于回溯/DFS算法的若干疑问

岛屿系列算法问题是经典的面试高频题，虽然基本的问题并不难，但是这类问题有一些有意思的扩展，比如求子岛屿数量，求形状不同的岛屿数量等等，本文就来把这些问题一网打尽。

岛屿系列题目的核心考点就是用 DFS/BFS 算法遍历二维数组。

本文主要来讲解如何用 DFS 算法来秒杀岛屿系列题目，不过用 BFS 算法的核心思路是完全一样的，无非就是把 DFS 改写成 BFS 而已。

那么如何在二维矩阵中使用 DFS 搜索呢？如果你把二维矩阵中的每一个位置看做一个节点，这个节点的上下左右四个位置就是相邻节点，那么整个矩阵就可以抽象成一幅网状的「图」结构。

根据 [学习数据结构和算法的框架思维](#)，完全可以根据二叉树的遍历框架改写出二维矩阵的 DFS 代码框架：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}

// 二维矩阵遍历框架
void dfs(int[][] grid, int i, int j, boolean[][] visited) {
```

```

int m = grid.length, n = grid[0].length;
if (i < 0 || j < 0 || i >= m || j >= n) {
    // 超出索引边界
    return;
}
if (visited[i][j]) {
    // 已遍历过 (i, j)
    return;
}

// 进入当前节点 (i, j)
visited[i][j] = true;

// 进入相邻节点 (四叉树)
// 上
dfs(grid, i - 1, j, visited);
// 下
dfs(grid, i + 1, j, visited);
// 左
dfs(grid, i, j - 1, visited);
// 右
dfs(grid, i, j + 1, visited);
}

```

因为二维矩阵本质上是一幅「图」，所以遍历的过程中需要一个 `visited` 布尔数组防止走回头路，如果你能理解上面这段代码，那么搞定所有岛屿系列题目都很简单。

这里额外说一个处理二维数组的常用小技巧，你有时会看到使用「方向数组」来处理上下左右的遍历，和前文 [union-find 算法详解](#) 的代码很类似：

```

// 方向数组，分别代表上、下、左、右
int[][] dirs = new int[][]{{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

void dfs(int[][] grid, int i, int j, boolean[][] visited) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return;
    }
    if (visited[i][j]) {
        // 已遍历过 (i, j)
        return;
    }

    // 进入节点 (i, j)
    visited[i][j] = true;
    // 递归遍历上下左右的节点
    for (int[] d : dirs) {
        int next_i = i + d[0];
        int next_j = j + d[1];
        dfs(grid, next_i, next_j, visited);
    }
    // 离开节点 (i, j)
}

```

这种写法无非就是用 for 循环处理上下左右的遍历罢了，你可以按照个人喜好选择写法。下面就按照上述框架结合可视化面板来解题。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 回溯算法实践：数独和 N 皇后问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
37. Sudoku Solver	37. 解数独	●
51. N-Queens	51. N 皇后	●
52. N-Queens II	52. N皇后 II	●

-----

阅读本文前，你需要先学习：

- [回溯算法核心框架](#)

你已经学习过 [回溯算法核心框架](#)，那么本文就来探讨两道经典算法题：数独游戏和 N 皇后问题。

选择这两个问题，主要是它们的解法思路非常相似，而且它们都是回溯算法在实际生活中的有趣应用。

## 游戏简介

### 数独游戏

大家应该都玩过数独游戏，就是给你一个  $9 \times 9$  的棋盘，其中有一些格子预先填入了数字，让你在其余空的格子中填入数字 1~9，要求每行、每列和每个  $3 \times 3$  的九宫格内数字都不能重复。

下面是一个数独游戏的例子（来源 [Wikipedia](#)）：

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

我小的时候也尝试过玩数独游戏，但只要稍微有些难度，就搞不定了。后来我才知道做数独是有技巧的，有一些比较专业的数独游戏软件会教你玩数独的技巧，不过在我看来这些技巧都太复杂，根本就没有兴趣看下去。

现在学习了回溯算法，多困难的数独问题都拦不住我了。只要有规则，就一定可以暴力穷举出符合条件的答案来，不是吗？

下面是我用回溯算法完成数独的例子：



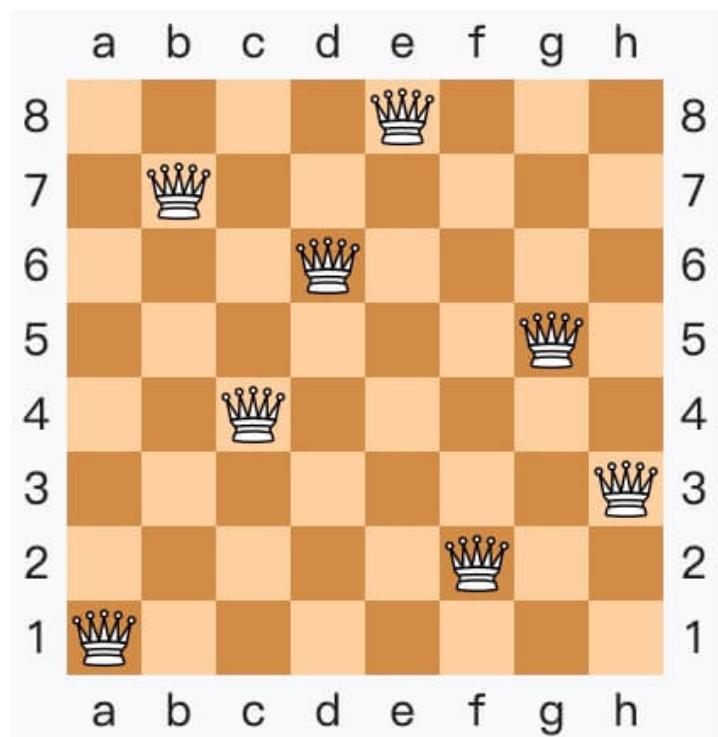
稍后我会详细讲解这道题的解法。

## N 皇后问题

在国际象棋中，皇后可以攻击同一行、同一列和同一条对角线上的任意单位。N 皇后问题是指在一个  $N \times N$  的棋盘上摆放 N 个皇后，要求任何两个皇后之间都不能互相攻击。

换句话说，就是让你在一个  $N \times N$  的棋盘上放置  $N$  个皇后，使得每行、每列和每个对角线都只有一个皇后。

比如这是 8 皇后问题的一个解（来源 [Wikipedia](#)）：



可以看到，对于任意一个皇后，它所在的行、列和对角线（左上、右上、左下、右下）都没有其他皇后，所以这就是一个符合规则的解。

在讲上述题目之前，我需要先讲一道比较简单的回溯算法问题，把这个问题作为铺垫，就能更容易理解数独游戏和  $N$  皇后问题的解法了。

## n 位二进制数的所有可能

我来给你编一道简单的题目，请你实现这样一个函数：

```
List<String> generateBinaryNumber(int n);
```

函数的输入是一个正整数  $n$ ，请你返回所有长度为  $n$  的二进制数（0、1 组成），你可以按任意顺序返回答案。

比如说  $n = 3$ ，那么你需要以字符串形式返回如下  $2^3=8$  种不同的二进制数：

```
000  
001  
010  
011  
100  
101  
110  
111
```

这道题可以认为是数独游戏和  $N$  皇后问题的简化版：

这道题相当于让你对一个长度为  $n$  的一维数组中的每一个位置进行穷举，其中每个位置可以填 0 或 1。

数独游戏相当于让你对一个  $9 \times 9$  的二维数组中的每个位置进行穷举，其中每个位置可以是数字 **1~9**，且同一行、同一列和同一个  $3 \times 3$  的九宫格内数字不能重复。

N 皇后问题相当于让你对一个 **N × N** 的二维数组中的每个位置进行穷举，其中每个位置可以不放皇后或者放置皇后（相当于 **0** 或 **1**），且不能存在多个皇后在同一行、同一列或同一对角线上。

所以，只要你把这道简化版的题目的穷举过程搞明白，其他问题都迎刃而解了，无非是规则多了一些而已。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 回溯算法实践：括号生成



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">22. Generate Parentheses</a>	<a href="#">22. 括号生成</a>	困难

阅读本文前，你需要先学习：

- 二叉树结构基础
- 二叉树的遍历框架
- 多叉树结构及遍历框架
- 回溯算法套路框架详解

括号问题可以简单分成两类，一类是前文写过的 [括号的合法性判断](#)，一类是合法括号的生成。对于括号合法性的判断，主要是借助「栈」这种数据结构，而对于括号的生成，一般都要利用 [回溯算法](#) 进行暴力穷举。

回到正题，看下力扣第 22 题「括号生成」，要求如下：

## ▼ 22. 括号生成 Leetcode | 力扣

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：

- $1 \leq n \leq 8$

函数签名如下：

```
List<String> generateParenthesis(int n);
```

有关括号问题，你只要记住以下性质，思路就很容易想出来：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 回溯算法实践：集合划分



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">698. Partition to K Equal Sum Subsets</a>	698. 划分为k个相等的子集	困难

阅读本文前，你需要先学习：

- 多叉树结构及遍历框架
- 二叉树系列算法（纲领篇）
- 回溯算法框架套路
- 球盒模型：回溯算法的两种穷举视角

我之前说过回溯算法是笔试中最好用的算法，只要你没什么思路，就用回溯算法暴力求解，即便不能通过所有测试用例，多少能过一点。回溯算法的技巧也不算难，就是穷举一棵决策树的过程，只要在递归之前「做选择」，在递归之后「撤销选择」就行了。

但是，就算暴力穷举，不同的思路也有优劣之分。本文就来看一道非常经典的回溯算法问题，力扣第 698 题「划分为k个相等的子集」。这道题可以帮你更深刻理解回溯算法的思维，得心应手地写出回溯函数。

题目非常简单：

给你输入一个数组 `nums` 和一个正整数 `k`，请你判断 `nums` 是否能够被平分为元素和相同的 `k` 个子集。

函数签名如下：

```
boolean canPartitionKSubsets(int[] nums, int k);
```

我们之前 [背包问题之子集划分](#) 写过一次子集划分问题，不过那道题只需要我们把集合划分成两个相等的集合，可以转化成背包问题用动态规划技巧解决。

为什么划分成两个相等的子集可以转化成背包问题用动态规划思路解决，而划分成 `k` 个相等的子集就不可以转化成背包问题，只能用回溯算法暴力穷举？请先尝试自己思考。

为什么划分两个相等的子集可以转化成背包问题？

背包问题之子集划分 的场景中，有一个背包和若干物品，每个物品有**两个选择**，分别是「装进背包」和「不装进背包」。把原集合  $S$  划分成两个相等子集  $S_1, S_2$  的场景下， $S$  中的每个元素也有**两个选择**，分别是「装进  $S_1$ 」和「不装进  $S_1$ （装进  $S_2$ ）」，这时候的穷举思路其实和背包问题相同。

但如果你想把  $S$  划分成  $k$  个相等的子集，相当于  $S$  中的每个元素有  $k$  个选择，这和标准背包问题的场景有本质区别，是无法套用背包问题的解题思路的。

## 题目思路

回到正题，这道算法题让我们求子集划分，子集问题和排列组合问题有所区别，但我们可以借鉴「球盒模型」的抽象，用两种不同的视角来解决这道子集划分问题。

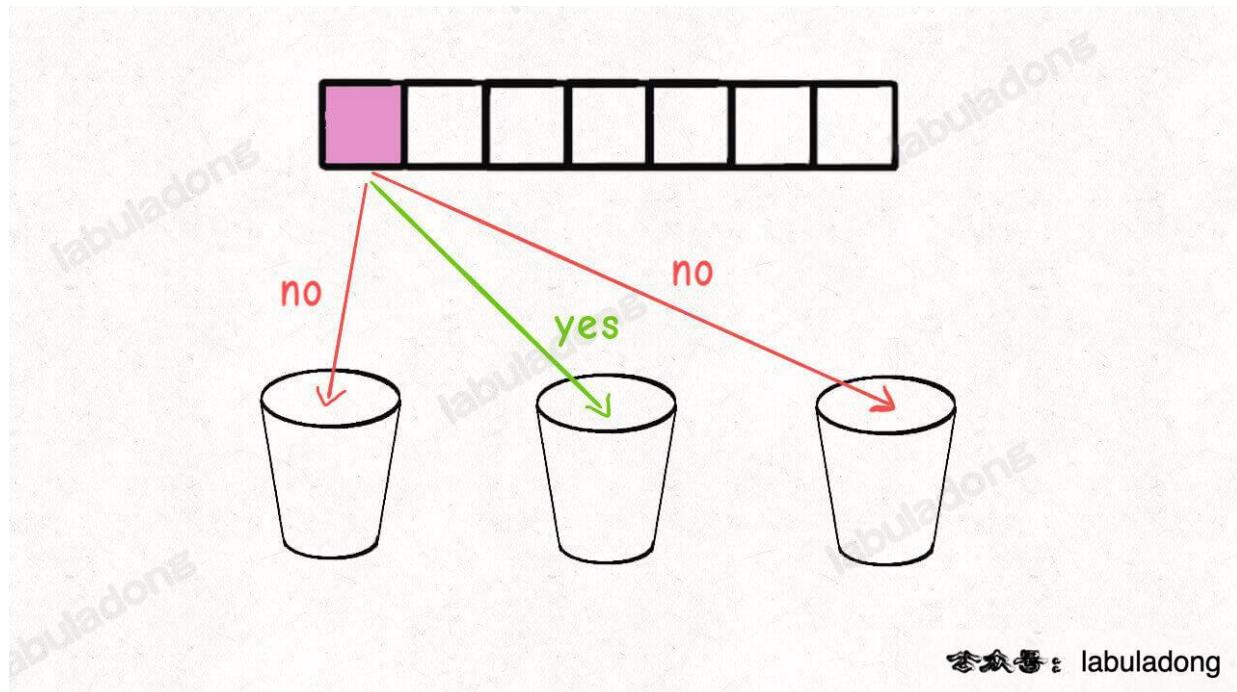
把装有  $n$  个数字的数组  $\text{nums}$  分成  $k$  个和相同的集合，你可以想象将  $n$  个数字分配到  $k$  个「桶」里，最后这  $k$  个「桶」里的数字之和要相同。

前文 [用球盒模型理解回溯算法](#) 说过，回溯算法的关键在哪里？

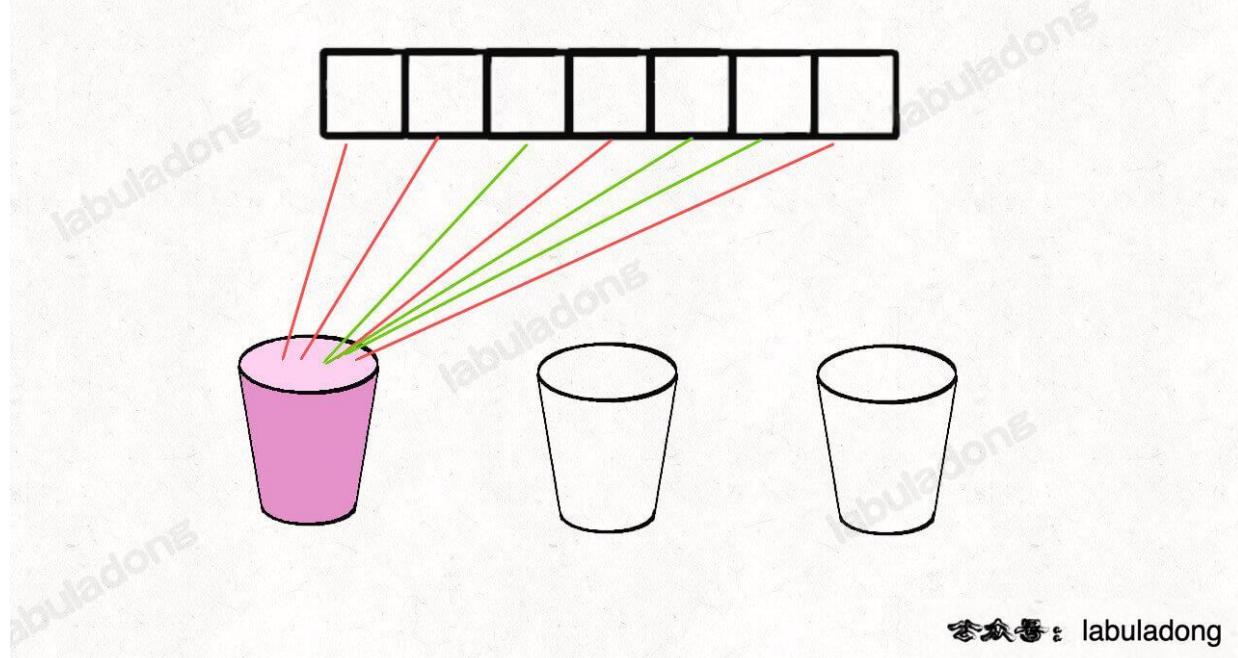
关键是要知道怎么「做选择」，这样才能利用递归函数进行穷举。

那么模仿排列公式的推导思路，将  $n$  个数字分配到  $k$  个桶里，我们也可以有两种视角：

视角一，如果我们切换到这  $n$  个数字的视角，每个数字都要选择进入到  $k$  个桶中的某一个。



视角二，如果我们切换到这  $k$  个桶的视角，对于每个桶，都要遍历  $\text{nums}$  中的  $n$  个数字，然后选择是否将当前遍历到的数字装进自己这个桶里。



© labuladong

你可能问，这两种视角有什么不同？

和前面讲的排列子集类似，用不同的视角进行穷举，虽然结果相同，但是解法代码的逻辑不同，具体的代码实现也不同，可能产生不同的时间、空间复杂度。我们需要选择复杂度更低的解法。

## 以数字的视角

用 for 循环迭代遍历 `nums` 数组大家肯定都会：

```
for (int index = 0; index < nums.length; index++) {  
    print(nums[index]);  
}
```

递归遍历数组你会不会？其实也很简单：

```
void traverse(int[] nums, int index) {  
    if (index == nums.length) {  
        return;  
    }  
    print(nums[index]);  
    traverse(nums, index + 1);  
}
```

只要调用 `traverse(nums, 0)`，和 for 循环的效果是完全一样的。

那么回到这道题，以数字的视角，选择 `k` 个桶，用 for 循环写出来是下面这样：

```
// k 个桶（集合），记录每个桶装的数字之和  
int[] bucket = new int[k];  
  
// 穷举 nums 中的每个数字  
for (int index = 0; index < nums.length; index++) {
```

```
// 穷举每个桶
for (int i = 0; i < k; i++) {
    // nums[index] 选择是否要进入第 i 个桶
    // ...
}
}
```

如果改成递归的形式，就是下面这段代码逻辑：

```
// k 个桶（集合），记录每个桶装的数字之和
int[] bucket = new int[k];

// 穷举 nums 中的每个数字
void backtrack(int[] nums, int index) {
    // base case
    if (index == nums.length) {
        return;
    }
    // 穷举每个桶
    for (int i = 0; i < bucket.length; i++) {
        // 选择装进第 i 个桶
        bucket[i] += nums[index];
        // 递归穷举下一个数字的选择
        backtrack(nums, index + 1);
        // 撤销选择
        bucket[i] -= nums[index];
    }
}
```

虽然上述代码仅仅是穷举逻辑，还不能解决我们的问题，但是只要略加完善即可：

```
class Solution {
    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 排除一些基本情况
        if (k > nums.length) return false;
        int sum = 0;
        for (int v : nums) sum += v;
        if (sum % k != 0) return false;

        // k 个桶（集合），记录每个桶装的数字之和
        int[] bucket = new int[k];
        // 理论上每个桶（集合）中数字的和
        int target = sum / k;
        // 穷举，看看 nums 是否能划分成 k 个和为 target 的子集
        return backtrack(nums, 0, bucket, target);
    }

    // 递归穷举 nums 中的每个数字
    boolean backtrack(
        int[] nums, int index, int[] bucket, int target) {
        if (index == nums.length) {
            // 检查所有桶的数字之和是否都是 target
            for (int i = 0; i < bucket.length; i++) {
                if (bucket[i] != target) {

```

```
        return false;
    }
}
// nums 成功平分成 k 个子集
return true;
}

// 穷举 nums[index] 可能装入的桶
for (int i = 0; i < bucket.length; i++) {
    // 剪枝，桶装装满了
    if (bucket[i] + nums[index] > target) {
        continue;
    }
    // 将 nums[index] 装入 bucket[i]
    bucket[i] += nums[index];
    // 递归穷举下一个数字的选择
    if (backtrack(nums, index + 1, bucket, target)) {
        return true;
    }
    // 撤销选择
    bucket[i] -= nums[index];
}

// nums[index] 装入哪个桶都不行
return false;
}
}
```

有之前的铺垫，相信这段代码是比较容易理解的，其实我们可以再做一个优化。

主要看 `backtrack` 函数的递归部分：

```
for (int i = 0; i < bucket.length; i++) {
    // 剪枝
    if (bucket[i] + nums[index] > target) {
        continue;
    }

    if (backtrack(nums, index + 1, bucket, target)) {
        return true;
    }
}
```

如果我们让尽可能多的情况命中剪枝的那个 `if` 分支，就可以减少递归调用的次数，一定程度上减少时间复杂度。

如何尽可能多的命中这个 `if` 分支呢？要知道我们的 `index` 参数是从 0 开始递增的，也就是递归地从 0 开始遍历 `nums` 数组。

如果我们提前对 `nums` 数组排序，把大的数字排在前面，那么大的数字会先被分配到 `bucket` 中，对于之后的数字，`bucket[i] + nums[index]` 会更大，更容易触发剪枝的 `if` 条件。

所以可以在之前的代码中再添加一些代码：

```
boolean canPartitionKSubsets(int[] nums, int k) {
    // 其他代码不变
```

```
// ...
// 降序排序 nums 数组
Arrays.sort(nums);
// 翻转数组，得到降序数组
for (i = 0, j = nums.length - 1; i < j; i++, j--) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
// *****
return backtrack(nums, 0, bucket, target);
}
```

这个解法可以得到正确答案，但耗时比较多，已经无法通过所有测试用例了，接下来看看另一种视角的解法。

## 以桶的视角

文章开头说了，以桶的视角进行穷举，每个桶需要遍历 `nums` 中的所有数字，决定是否把当前数字装进桶中；当装满一个桶之后，还要装下一个桶，直到所有桶都装满为止。

这个思路可以用下面这段代码表示出来：

```
// 装满所有桶为止
while (k > 0) {
    // 记录当前桶中的数字之和
    int bucket = 0;
    for (int i = 0; i < nums.length; i++) {
        // 决定是否将 nums[i] 放入当前桶中
        if (canAdd(bucket, num[i])) {
            bucket += nums[i];
        }
        if (bucket == target) {
            // 装满了一个桶，装下一个桶
            k--;
            break;
        }
    }
}
```

那么我们也可以把这个 `while` 循环改写成递归函数，不过比刚才略微复杂一些，首先写一个 `backtrack` 递归函数出来：

```
boolean backtrack(int k, int bucket, int[] nums, int start, boolean[] used, int target);
```

不要被这么多参数吓到，我会一个个解释这些参数。如果你够透彻理解了前文 [用球盒模型理解回溯算法](#)，也能得心应手地写出这样的回溯函数。

这个 `backtrack` 函数的参数可以这样解释：

现在 `k` 号桶正在思考是否应该把 `nums[start]` 这个元素装进来；目前 `k` 号桶里面已经装的数字之和为 `bucket`；`used` 标志某一个元素是否已经被装到桶中；`target` 是每个桶需要达成的目标和。

根据这个函数定义，可以这样调用 `backtrack` 函数：

```

class Solution {
    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 排除一些基本情况
        if (k > nums.length) return false;
        int sum = 0;
        for (int v : nums) sum += v;
        if (sum % k != 0) return false;

        boolean[] used = new boolean[nums.length];
        int target = sum / k;
        // k 号桶初始什么都没装，从 nums[0] 开始做选择
        return backtrack(k, 0, nums, 0, used, target);
    }
}

```

实现 `backtrack` 函数的逻辑之前，再重复一遍，从桶的视角：

- 1、需要遍历 `nums` 中所有数字，决定哪些数字需要装到当前桶中。
- 2、如果当前桶装满了（桶内数字和达到 `target`），则让下一个桶开始执行第 1 步。

下面的代码就实现了这个逻辑：

```

class Solution {
    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 见上文
    }

    boolean backtrack(int k, int bucket,
                     int[] nums, int start, boolean[] used, int target) {
        // base case
        if (k == 0) {
            // 所有桶都被装满了，而且 nums 一定全部用完了
            // 因为 target == sum / k
            return true;
        }
        if (bucket == target) {
            // 装满了当前桶，递归穷举下一个桶的选择
            // 让下一个桶从 nums[0] 开始选数字
            return backtrack(k - 1, 0, nums, 0, used, target);
        }

        // 从 start 开始向后探查有效的 nums[i] 装入当前桶
        for (int i = start; i < nums.length; i++) {
            // 剪枝
            if (used[i]) {
                // nums[i] 已经被装入别的桶中
                continue;
            }
            if (nums[i] + bucket > target) {
                // 当前桶装不下 nums[i]
                continue;
            }
            // 做选择，将 nums[i] 装入当前桶中
            used[i] = true;
            bucket += nums[i];

```

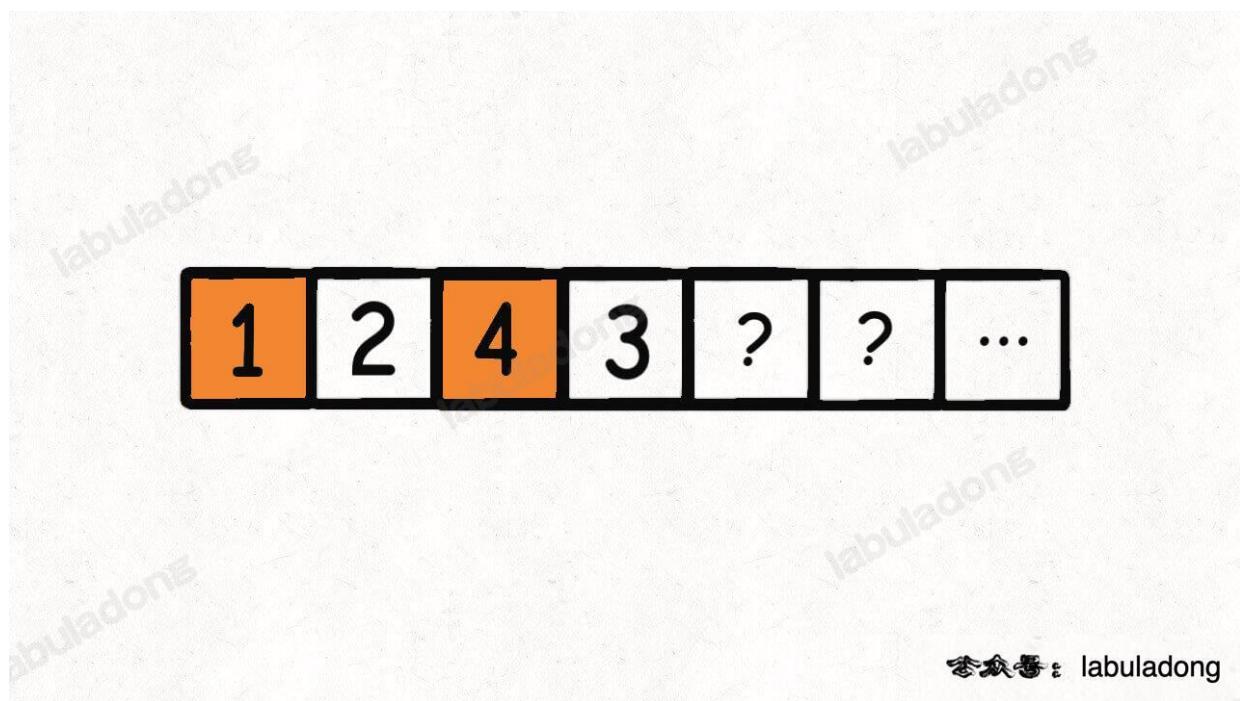
```
// 递归穷举下一个数字是否装入当前桶
if (backtrack(k, bucket, nums, i + 1, used, target)) {
    return true;
}
// 撤销选择
used[i] = false;
bucket -= nums[i];
}
// 穷举了所有数字，都无法装满当前桶
return false;
}
```

这段代码是可以得出正确答案的，但是效率很低，我们可以思考一下是否还有优化的空间。

首先，在这个解法中每个桶都可以认为是没有差异的，但是我们的回溯算法却会对它们区别对待，这里就会出现重复计算的情况。

什么意思呢？我们的回溯算法，说到底就是穷举所有可能的组合，然后看是否能找出和为 `target` 的 `k` 个桶（子集）。

那么，比如下面这种情况，`target = 5`，算法会在第一个桶里面装 `1, 4`：



现在第一个桶装满了，就开始装第二个桶，算法会装入 `2, 3`：



公众号：labuladong

然后以此类推，对后面的元素进行穷举，凑出若干个和为 5 的桶（子集）。

但问题是，如果最后发现无法凑出和为 target 的 k 个子集，算法会怎么做？

回溯算法会回溯到第一个桶，重新开始穷举，现在它知道第一个桶里装 1, 4 是不可行的，它会尝试把 2, 3 装到第一个桶里：



公众号：labuladong

现在第一个桶装满了，就开始装第二个桶，算法会装入 1, 4：



© labuladong

好，到这里你应该看出来问题了，这种情况其实和之前的那种情况是一样的。也就是说，到这里你其实已经知道不需要再穷举了，必然凑不出来和为 `target` 的 `k` 个子集。

但我们的算法还是会傻乎乎地继续穷举，因为在她看来，第一个桶和第二个桶里面装的元素不一样，那这就是两种不一样的情况呀。

那么我们怎么让算法的智商提高，识别出这种情况，避免冗余计算呢？

你注意这两种情况的 `used` 数组肯定长得一样，所以 `used` 数组可以认为是回溯过程中的「状态」。

所以，我们可以用一个 `memo` 备忘录，在装满一个桶时记录当前 `used` 的状态，如果当前 `used` 的状态是曾经出现过的，那就不用再继续穷举，从而起到剪枝避免冗余计算的作用。

有读者肯定会问，`used` 是一个布尔数组，怎么作为键进行存储呢？这其实是小问题，比如我们可以把数组转化成字符串，这样就可以作为哈希表的键进行存储了。

看下代码实现，只要稍微改一下 `backtrack` 函数即可：

```
class Solution {

    // 备忘录，存储 used 数组的状态
    HashMap<String, Boolean> memo = new HashMap<>();

    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 见上文
    }

    boolean backtrack(int k, int bucket, int[] nums, int start, boolean[] used, int target) {
        // base case
        if (k == 0) {
            return true;
        }
        // 将 used 的状态转化成形如 [true, false, ...] 的字符串
        // 便于存入 HashMap
        String state = Arrays.toString(used);
    }
}
```

```

    if (bucket == target) {
        // 装满了当前桶，递归穷举下一个桶的选择
        boolean res = backtrack(k - 1, 0, nums, 0, used, target);
        // 将当前状态和结果存入备忘录
        memo.put(state, res);
        return res;
    }

    if (memo.containsKey(state)) {
        // 如果当前状态曾今计算过，就直接返回，不要再递归穷举了
        return memo.get(state);
    }

    // 其他逻辑不变...
    return false; // This was added to complete the code, it's not a comment.
}
}

```

这样提交解法，发现执行效率依然比较低，这次不是因为算法逻辑上的冗余计算，而是代码实现上的问题。

**因为每次递归都要把 `used` 数组转化成字符串，这对于编程语言来说也是一个不小的消耗，所以我们还可以进一步优化。**

注意题目给的数据规模 `nums.length <= 16`，也就是说 `used` 数组最多也不会超过 16，那么我们完全可以用「位图」的技巧，用一个 `int` 类型的 `used` 变量来替代 `used` 数组。

具体来说，我们可以用整数 `used` 的第 `i` 位 `((used >> i) & 1)` 的 1/0 来表示 `used[i]` 的 true/false。

这样一来，不仅节约了空间，而且整数 `used` 也可以直接作为键存入 `HashMap`，省去数组转字符串的消耗。

看下最终的解法代码：

```

class Solution {
    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 排除一些基本情况
        if (k > nums.length) return false;
        int sum = 0;
        for (int v : nums) sum += v;
        if (sum % k != 0) return false;

        // 使用位图技巧
        int used = 0;
        int target = sum / k;
        // k 号桶初始什么都没装，从 nums[0] 开始做选择
        return backtrack(k, 0, nums, 0, used, target);
    }

    HashMap<Integer, Boolean> memo = new HashMap<>();

    boolean backtrack(int k, int bucket,
                     int[] nums, int start, int used, int target) {
        // base case
        if (k == 0) {
            // 所有桶都被装满了，而且 nums 一定全部用完了
            return true;
        }
        if (bucket == target) {
            // 装满了当前桶，递归穷举下一个桶的选择

```

```

    // 让下一个桶从 nums[0] 开始选数字
    boolean res = backtrack(k - 1, 0, nums, 0, used, target);
    // 缓存结果
    memo.put(used, res);
    return res;
}

if (memo.containsKey(used)) {
    // 避免冗余计算
    return memo.get(used);
}

for (int i = start; i < nums.length; i++) {
    // 剪枝
    // 判断第 i 位是否是 1
    if (((used >> i) & 1) == 1) {
        // nums[i] 已经被装入别的桶中
        continue;
    }
    if (nums[i] + bucket > target) {
        continue;
    }
    // 做选择
    // 将第 i 位置为 1
    used |= 1 << i;
    bucket += nums[i];
    // 递归穷举下一个数字是否装入当前桶
    if (backtrack(k, bucket, nums, i + 1, used, target)) {
        return true;
    }
    // 撤销选择
    // 使用异或运算将第 i 位恢复 0
    used ^= 1 << i;
    bucket -= nums[i];
}

return false;
}
}

```

## ▶ 🎥 代码可视化动画

至此，这道题的第二种思路也完成了。

## 最后总结

本文写的这两种思路都可以算出正确答案，不过第一种解法即便经过了排序优化，也明显比第二种解法慢很多，这是为什么呢？

我们来分析一下这两个算法的时间复杂度，假设 `nums` 中的元素个数为 `n`。

先说第一个解法，也就是从数字的角度进行穷举，`n` 个数字，每个数字有 `k` 个桶可供选择，所以组合出的结果个数为  $k^n$ ，时间复杂度也就是  $O(k^n)$ 。

第二个解法，每个桶要遍历 `n` 个数字，对每个数字有「装入」或「不装入」两种选择，所以组合的结果有  $2^n$  种；而我们有 `k` 个桶，所以总的时间复杂度为  $O(k \cdot 2^n)$ 。

当然，这是对最坏复杂度上界的粗略估算，实际的复杂度肯定要好很多，毕竟我们添加了这么多剪枝逻辑。不过，从复杂度的上界已经可以看出第一种思路要慢很多了。

所以，谁说回溯算法没有技巧性的？虽然回溯算法就是暴力穷举，但穷举也分聪明的穷举方式和低效的穷举方式，关键看你以谁的「视角」进行穷举。

通俗来说，我们应该尽量「少量多次」，就是说宁可多做几次选择（乘法关系），也不要给太大的选择空间（指数关系）；做  $n$  次「 $k$  选一」仅重复一次 ( $O(k^n)$ )，比  $n$  次「二选一」重复  $k$  次 ( $O(k \cdot 2^n)$ ) 效率低很多。

好了，这道题我们从两种视角进行穷举，虽然代码量看起来多，但核心逻辑都是类似的，相信你通过本文能够更深刻地理解回溯算法。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">473. Matchsticks to Square</a>	<a href="#">473. 火柴拼正方形</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】回溯算法经典习题 |

学习本章回溯算法习题之前，你必须掌握以下文章中所讲的思维方法：

- 回溯算法核心框架
- 回溯算法秒杀排列/组合/子集的九种变体
- 球盒模型：回溯算法的两种穷举视角

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】回溯算法经典习题 II

学习本章回溯算法习题之前，你必须掌握以下文章中所讲的思维方法：

- 回溯算法核心框架
- 回溯算法秒杀排列/组合/子集的九种变体
- 球盒模型：回溯算法的两种穷举视角

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】回溯算法经典习题 III

学习本章回溯算法习题之前，你必须掌握以下文章中所讲的思维方法：

- 回溯算法核心框架
- 回溯算法秒杀排列/组合/子集的九种变体
- 球盒模型：回溯算法的两种穷举视角

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# BFS 算法解题套路框架



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	<a href="#">111. 二叉树的最小深度</a>	
<a href="#">752. Open the Lock</a>	<a href="#">752. 打开转盘锁</a>	

阅读本文前，你需要先学习：

- [多叉树基础及遍历](#)
- [图结构基础及通用实现](#)

后台有很多人问起 BFS 和 DFS 的框架，今天就来说说吧。

首先，你要说我没写过 BFS 框架，这话没错，今天写个框架你背住就完事儿了。但要是说没写过 DFS 框架，那你还真是说错了，**其实 DFS 算法就是回溯算法**，我们前文[回溯算法框架套路详解](#)就写过了，而且写得不是一般得好，建议好好复习，嘿嘿嘿~

BFS 的核心思想应该不难理解的，就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，我们写 BFS 算法都是用「队列」这种数据结构，每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：**BFS 找到的路径一定是最短的，但代价就是空间复杂度可能比 DFS 大很多**，至于为什么，我们后面介绍了框架就很容易看出来了。

本文就由浅入深写两道 BFS 的典型题目，分别是「二叉树的最小高度」和「打开密码锁的最少步数」，手把手教你怎么写 BFS 算法。

## 一、算法框架

要说框架的话，我们先举例一下 BFS 出现的常见场景好吧，**问题的本质就是让你在一幅「图」中找到从起点 **start** 到终点 **target** 的最近距离**，这个例子听起来很枯燥，但是**BFS 算法问题其实都是在干这个事儿**，把枯燥的本质搞清楚了，再去欣赏各种问题的包装才能胸有成竹嘛。

这个广义的描述可以有各种变体，比如走迷宫，有的格子是围墙不能走，从起点到终点的最短距离是多少？如果这个迷宫带「传送门」可以瞬间传送呢？

再比如说两个单词，要求你通过某些替换，把其中一个变成另一个，每次只能替换一个字符，最少要替换几次？

再比如说连连看游戏，两个方块消除的条件不仅仅是图案相同，还得保证两个方块之间的最短连线不能多于两个拐点。你玩连连看，点击两个坐标，游戏是如何判断它俩的最短连线有几个拐点的？

再比如……

净整些花里胡哨的，本质上看这些问题都没啥区别，就是一幅「图」，让你从一个起点，走到终点，问最短路径。这就是 BFS 的本质，框架搞清楚了直接默写就好。

记住下面这个框架就 OK 了：

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    // 核心数据结构
    Queue<Node> q;
    // 避免走回头路
    Set<Node> visited;

    // 将起点加入队列
    q.offer(start);
    visited.add(start);

    while (q not empty) {
        int sz = q.size();
        // 将当前队列中的所有节点向四周扩散
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            // 划重点：这里判断是否到达终点
            if (cur is target)
                return step;
            // 将 cur 的相邻节点加入队列
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
    }
    // 如果走到这里，说明在图中没有找到目标节点
}
```

队列 `q` 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

## 二、二叉树的最小高度

先来个简单的问题实践一下 BFS 框架吧，判断一棵二叉树的最小高度，这也是力扣第 111 题「二叉树的最小深度」：

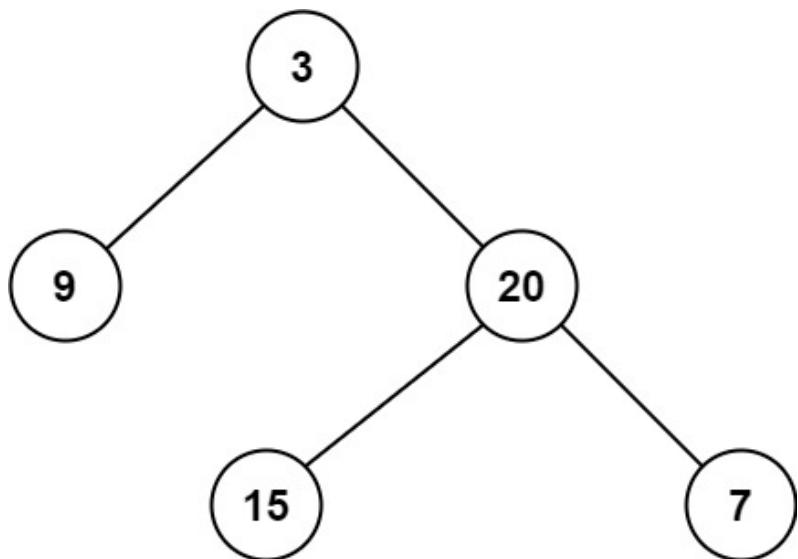
### ▼ 111. 二叉树的最小深度 [Leetcode | 力扣](#)

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

**说明：**叶子节点是指没有子节点的节点。

**示例 1：**



```
输入: root = [3,9,20,null,null,15,7]  
输出: 2
```

### 示例 2:

```
输入: root = [2,null,3,null,4,null,5,null,6]  
输出: 5
```

### 提示:

- 树中节点数的范围在  $[0, 10^5]$  内
- $-1000 \leq \text{node.val} \leq 1000$

怎么套到 BFS 的框架里呢？首先明确一下起点 `start` 和终点 `target` 是什么，怎么判断到达了终点？

显然起点就是 `root` 根节点，终点就是最靠近根节点的那个「叶子节点」嘛，叶子节点就是两个子节点都是 `null` 的节点：

```
if (cur.left == null && cur.right == null)  
    // 到达叶子节点
```

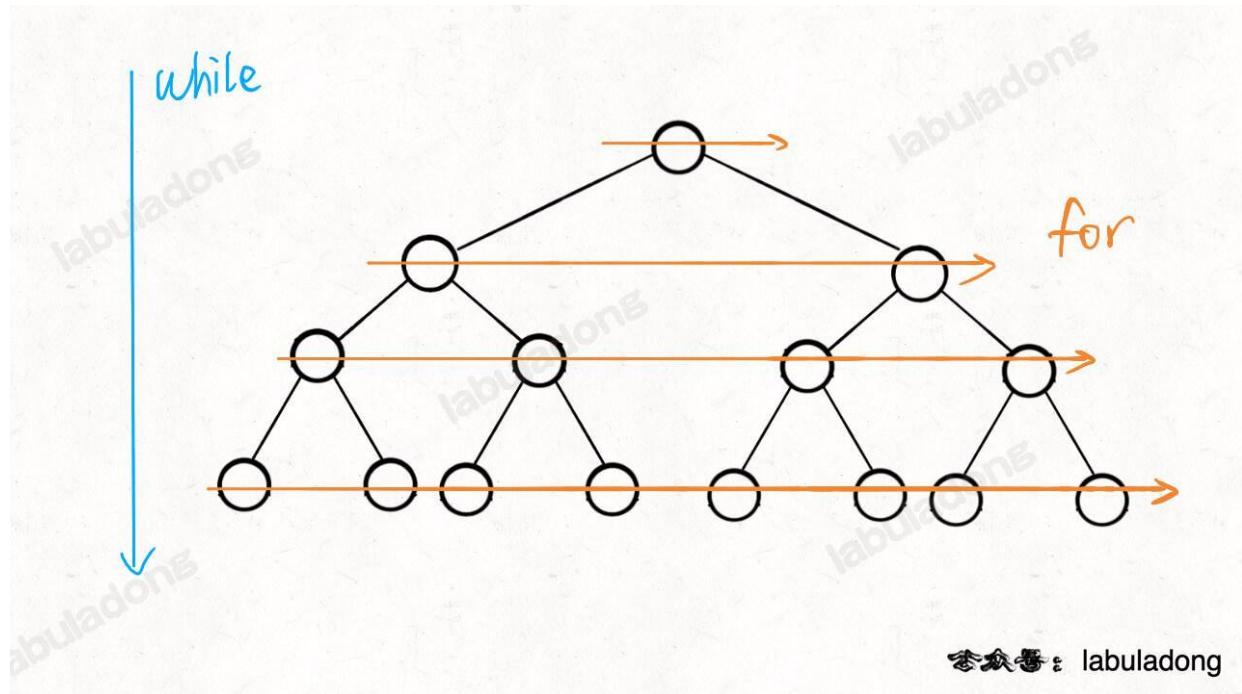
那么，按照我们上述的框架稍加改造来写解法即可：

```
class Solution {  
    public int minDepth(TreeNode root) {  
        if (root == null) return 0;  
        Queue<TreeNode> q = new LinkedList<>();  
        q.offer(root);  
        // root 本身就是一层, depth 初始化为 1  
        int depth = 1;  
  
        while (!q.isEmpty()) {  
            int sz = q.size();  
            // 将当前队列中的所有节点向四周扩散  
            for (int i = 0; i < sz; i++) {
```

```
TreeNode cur = q.poll();
// 判断是否到达终点
if (cur.left == null && cur.right == null)
    return depth;
// 将 cur 的相邻节点加入队列
if (cur.left != null)
    q.offer(cur.left);
if (cur.right != null)
    q.offer(cur.right);
}
// 这里增加步数
depth++;
}
return depth;
}
```

### ▶ 🎃 代码可视化动画🎃

这里注意这个 `while` 循环和 `for` 循环的配合，`while` 循环控制一层一层往下走，`for` 循环利用 `sz` 变量控制从左到右遍历每一层二叉树节点：



这一点很重要，这个形式在普通 BFS 问题中都很常见，但是在 Dijkstra 算法模板框架 中我们修改了这种代码模式，读完并理解本文后你可以去看看 BFS 算法是如何演变成 Dijkstra 算法在加权图中寻找最短路径的。

话说回来，二叉树本身是很简单的数据结构，我想上述代码你应该可以理解的，其实其他复杂问题都是这个框架的变形，再探讨复杂问题之前，我们解答两个问题：

#### 1、为什么 BFS 可以找到最短距离，DFS 不行吗？

首先，你看 BFS 的逻辑，`depth` 每增加一次，队列中的所有节点都向前迈一步，这保证了第一次到达终点的时候，走的步数是最少的。

DFS 不能找最短路径吗？其实也是可以的，但是时间复杂度相对高很多。你想啊，DFS 实际上是靠递归的堆栈记录走过的路径，你要找到最短路径，肯定得把二叉树中所有树杈都探索完才能对比出最短的路径有多长对不对？而 BFS 借助队列做

到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离的。

形象点说，DFS 是线，BFS 是面；DFS 是单打独斗，BFS 是集体行动。这个应该比较容易理解吧。

## 2、既然 BFS 那么好，为啥 DFS 还要存在？

BFS 可以找到最短距离，但是空间复杂度高，而 DFS 的空间复杂度较低。

还是拿刚才我们处理二叉树问题的例子，假设给你的这个二叉树是满二叉树，节点数为  $N$ ，对于 DFS 算法来说，空间复杂度无非就是递归堆栈，最坏情况下顶多就是树的高度，也就是  $O(\log N)$ 。

但是你想想 BFS 算法，队列中每次都会储存着二叉树一层的节点，这样的话最坏情况下空间复杂度应该是树的最底层节点的数量，也就是  $N/2$ ，用 Big O 表示的话也就是  $O(N)$ 。

由此观之，BFS 还是有代价的，一般来说在找最短路径的时候使用 BFS，其他时候还是 DFS 使用得多一些（主要是递归代码好写）。

好了，现在你对 BFS 了解得足够多了，下面来一道难一点的题目，深化一下框架的理解吧。

## 三、解开密码锁的最少次数

这是力扣第 752 题「打开转盘锁」，比较有意思：

### ▼ 752. 打开转盘锁 [Leetcode | 力扣](#)

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有 10 个数字： $\text{0000000000}$ ,  $\text{0000100000}$ ,  $\text{0000200000}$ ,  
 $\text{0000300000}$ ,  $\text{0000400000}$ ,  $\text{0000500000}$ ,  $\text{0000600000}$ ,  $\text{0000700000}$ ,  $\text{0000800000}$ ,  
 $\text{0000900000}$ 。每个拨轮可以自由旋转：例如把  $\text{0000900000}$  变为  $\text{0000000000}$ ,  $\text{0000000000}$  变为  
 $\text{0000900000}$ 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为  $\text{0000000000}$ ，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回  $-1$ 。

### 示例 1:

```
输入: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出: 6
解释:
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的,
因为当拨动到 "0102" 时这个锁就会被锁定。
```

### 示例 2:

```
输入: deadends = ["8888"], target = "0009"
输出: 1
解释: 把最后一位反向旋转一次即可 "0000" -> "0009"。
```

### 示例 3:

**输入:** deadends = ["8887", "8889", "8878", "8898", "8788", "8988", "7888", "9888"], target = "8888"  
**输出:** -1  
**解释:** 无法旋转到目标数字且不被锁定。

提示:

- `1 <= deadends.length <= 500`
- `deadends[i].length == 4`
- `target.length == 4`
- `target` 不在 `deadends` 之中
- `target` 和 `deadends[i]` 仅由若干位数字组成

函数签名如下:

```
int openLock(String[] deadends, String target)
```

题目中描述的就是我们生活中常见的那种密码锁，如果没有任何约束，最少的拨动次数很好算，就像我们平时开密码锁那样直奔密码拨就行了。

但现在的难点就在于，不能出现 `deadends`，应该如何计算出最少的转动次数呢？

第一步，我们不管所有的限制条件，不管 `deadends` 和 `target` 的限制，就思考一个问题：如果让你设计一个算法，穷举所有可能的密码组合，你怎么做？

穷举呗，再简单一点，如果你只转一下锁，有几种可能？总共有 4 个位置，每个位置可以向上转，也可以向下转，也就是有 8 种可能对吧。

比如说从 "0000" 开始，转一次，可以穷举出 "1000", "9000", "0100", "0900"... 共 8 种密码。然后，再以这 8 种密码作为基础，对每个密码再转一下，穷举出所有可能...

仔细想想，这就可以抽象成一幅图，每个节点有 8 个相邻的节点，又让你求最短距离，这不就是典型的 BFS 嘛，框架就可以派上用场了，先写出一个「简陋」的 BFS 框架代码再说别的：

```
// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}

// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
```

```
// BFS 框架，打印出所有可能的密码
void BFS(String target) {
    Queue<String> q = new LinkedList<>();
    q.offer("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        // 将当前队列中的所有节点向周围扩散
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            // 判断是否到达终点
            System.out.println(cur);

            // 将一个节点的相邻节点加入队列
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                String down = minusOne(cur, j);
                q.offer(up);
                q.offer(down);
            }
        }
        // 在这里增加步数
    }
    return;
}
```

这段代码当然有很多问题，但是我们做算法题肯定不是一蹴而就的，而是从简陋到完美的。不要完美主义，咱要慢慢来，好不。

这段 BFS 代码已经能够穷举所有可能的密码组合了，但是显然不能完成题目，有如下问题需要解决：

- 1、会走回头路。比如说我们从 "0000" 拨到 "1000"，但是等从队列拿出 "1000" 时，还会拨出一个 "0000"，这样的话会产生死循环。
- 2、没有终止条件，按照题目要求，我们找到 `target` 就应该结束并返回拨动的次数。
- 3、没有对 `deadends` 的处理，按道理这些「死亡密码」是不能出现的，也就是说你遇到这些密码的时候需要跳过。

如果你能够看懂上面那段代码，真得给你鼓掌，只要按照 BFS 框架在对应的位置稍作修改即可修复这些问题：

```
class Solution {
    public int openLock(String[] deadends, String target) {
        // 记录需要跳过的死亡密码
        Set<String> deads = new HashSet<>();
        for (String s : deadends) deads.add(s);
        // 记录已经穷举过的密码，防止走回头路
        Set<String> visited = new HashSet<>();
        Queue<String> q = new LinkedList<>();
        // 从起点开始启动广度优先搜索
        int step = 0;
        q.offer("0000");
        visited.add("0000");

        while (!q.isEmpty()) {
            int sz = q.size();
```

```

    // 将当前队列中的所有节点向周围扩散
    for (int i = 0; i < sz; i++) {
        String cur = q.poll();

        // 判断是否到达终点
        if (deads.contains(cur))
            continue;
        if (cur.equals(target))
            return step;

        // 将一个节点的未遍历相邻节点加入队列
        for (int j = 0; j < 4; j++) {
            String up = plusOne(cur, j);
            if (!visited.contains(up)) {
                q.offer(up);
                visited.add(up);
            }
            String down = minusOne(cur, j);
            if (!visited.contains(down)) {
                q.offer(down);
                visited.add(down);
            }
        }
    }
    // 在这里增加步数
    step++;
}
// 如果穷举完都没找到目标密码，那就是找不到了
return -1;
}

// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}

// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
}

```

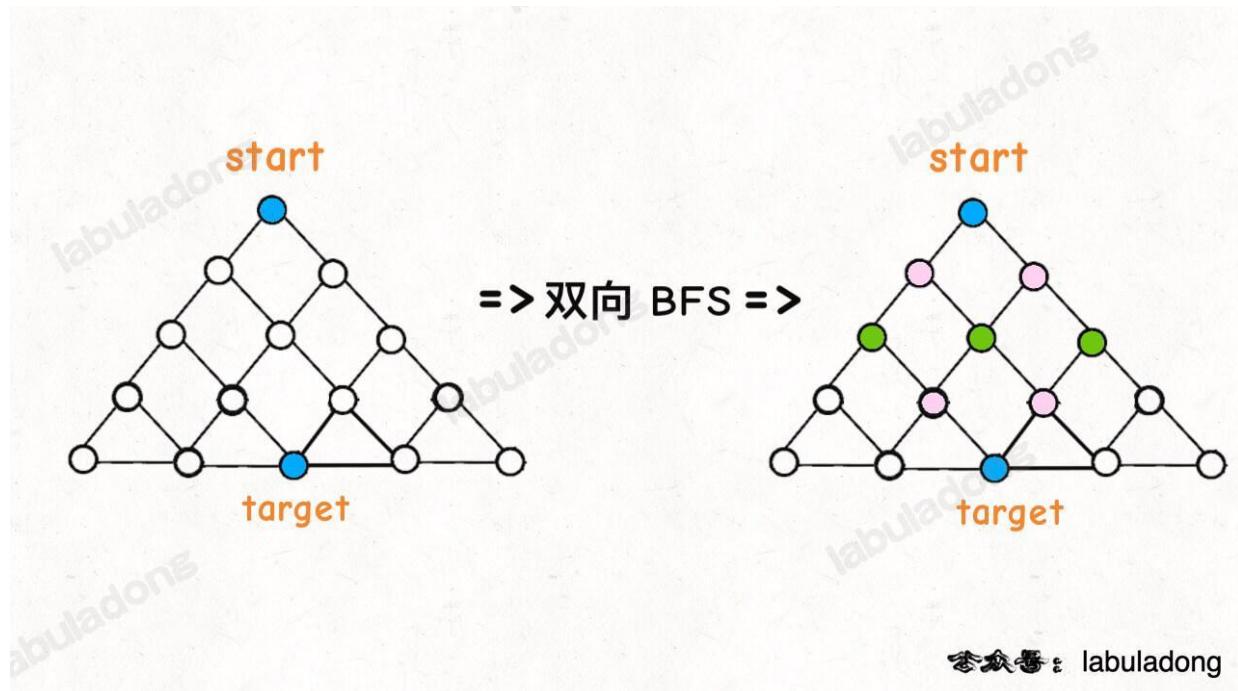
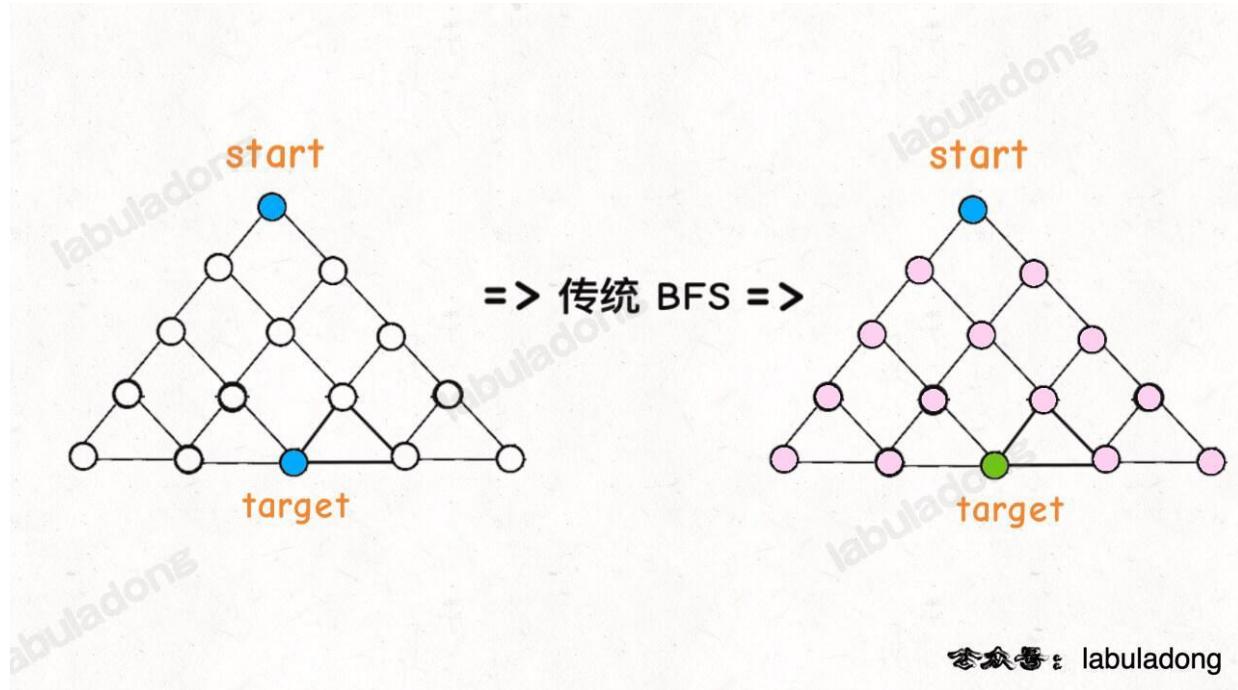
至此，我们就解决这道题目了。有一个比较小的优化：可以不需要 `dead` 这个哈希集合，可以直接将这些元素初始化到 `visited` 集合中，效果是一样的，可能更加优雅一些。

## 四、双向 BFS 优化

你以为到这里 BFS 算法就结束了？恰恰相反。BFS 算法还有一种稍微高级一点的优化思路：双向 BFS，可以进一步提高算法的效率。

篇幅所限，这里就提一下区别：传统的 BFS 框架就是从起点开始向四周扩散，遇到终点时停止；而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。

为什么这样能够提升效率呢？其实从 Big O 表示法分析算法复杂度的话，它俩的最坏复杂度都是  $O(N)$ ，但是实际上双向 BFS 确实会快一些，我给你画两张图看一眼就明白了：



图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到 **target**；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。从这个例子可以直观地感受到，双向 BFS 是要比传统 BFS 高效的。

不过，双向 BFS 也有局限，因为你必须知道终点在哪里。比如我们刚才讨论的二叉树最小高度的问题，你一开始根本就不知道终点在哪里，也就无法使用双向 BFS；但是第二个密码锁的问题，是可以使用双向 BFS 算法来提高效率的，代码稍加修改即可：

```

class Solution {
    public int openLock(String[] deadends, String target) {
        Set<String> deads = new HashSet<>();
        for (String s : deadends) deads.add(s);
        // 用集合不用队列，可以快速判断元素是否存在
        Set<String> q1 = new HashSet<>();
        Set<String> q2 = new HashSet<>();
        Set<String> visited = new HashSet<>();

        int step = 0;
        q1.add("0000");
        q2.add(target);

        while (!q1.isEmpty() && !q2.isEmpty()) {
            // 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果
            Set<String> temp = new HashSet<>();

            // 将 q1 中的所有节点向周围扩散
            for (String cur : q1) {
                // 判断是否到达终点
                if (deads.contains(cur))
                    continue;
                if (q2.contains(cur))
                    return step;

                visited.add(cur);

                // 将一个节点的未遍历相邻节点加入集合
                for (int j = 0; j < 4; j++) {
                    String up = plusOne(cur, j);
                    if (!visited.contains(up))
                        temp.add(up);
                    String down = minusOne(cur, j);
                    if (!visited.contains(down))
                        temp.add(down);
                }
            }
            // 在这里增加步数
            step++;
            // temp 相当于 q1
            // 这里交换 q1 q2，下一轮 while 就是扩散 q2
            q1 = q2;
            q2 = temp;
        }
        return -1;
    }

    // 将 s[j] 向上拨动一次
    String plusOne(String s, int j) {
        char[] ch = s.toCharArray();
        if (ch[j] == '9')
            ch[j] = '0';
        else
            ch[j] += 1;
        return new String(ch);
    }

    // 将 s[i] 向下拨动一次
}

```

```

String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
}

```

双向 BFS 还是遵循 BFS 算法框架的，只是不再使用队列，而是使用 HashSet 方便快速判断两个集合是否有交集。

另外的一个技巧点就是 while 循环的最后交换 q1 和 q2 的内容，所以只要默认扩散 q1 就相当于轮流扩散 q1 和 q2。

其实双向 BFS 还有一个优化，就是在 while 循环开始时做一个判断：

```

// ...
while (!q1.isEmpty() && !q2.isEmpty()) {
    if (q1.size() > q2.size()) {
        // 交换 q1 和 q2
        temp = q1;
        q1 = q2;
        q2 = temp;
    }
    // ...
}

```

为什么这是一个优化呢？

因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，无论传统 BFS 还是双向 BFS，无论做不做优化，从 Big O 衡量标准来看，时间复杂度都是一样的，只能说双向 BFS 是一种 trick，算法运行的速度会相对快一点，掌握不掌握其实都无所谓。最关键的是把 BFS 通用框架记下来，反正所有 BFS 算法都可以用它套出解法。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1091. Shortest Path in Binary Matrix</a>	<a href="#">1091. 二进制矩阵中的最短路径</a>	
<a href="#">117. Populating Next Right Pointers in Each Node II</a>	<a href="#">117. 填充每个节点的下一个右侧节点指针 II</a>	
<a href="#">127. Word Ladder</a>	<a href="#">127. 单词接龙</a>	
<a href="#">1926. Nearest Exit from Entrance in Maze</a>	<a href="#">1926. 迷宫中离入口最近的出口</a>	
<a href="#">2850. Minimum Moves to Spread Stones Over Grid</a>	<a href="#">2850. 将石头分散到网格图的最少移动次数</a>	
<a href="#">286. Walls and Gates</a>	<a href="#">286. 墙与门</a>	
<a href="#">310. Minimum Height Trees</a>	<a href="#">310. 最小高度树</a>	

LeetCode	力扣	难度
329. Longest Increasing Path in a Matrix	329. 矩阵中的最长递增路径	
365. Water and Jug Problem	365. 水壶问题	
431. Encode N-ary Tree to Binary Tree	431. 将 N 叉树编码为二叉树	
433. Minimum Genetic Mutation	433. 最小基因变化	
490. The Maze	490. 迷宫	
505. The Maze II	505. 迷宫 II	
542. 01 Matrix	542. 01 矩阵	
547. Number of Provinces	547. 省份数量	
773. Sliding Puzzle	773. 滑动谜题	
863. All Nodes Distance K in Binary Tree	863. 二叉树中所有距离为 K 的结点	
994. Rotting Oranges	994. 腐烂的橘子	
-	剑指 Offer II 109. 开密码锁	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 如何用 BFS 算法秒杀各种智力题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

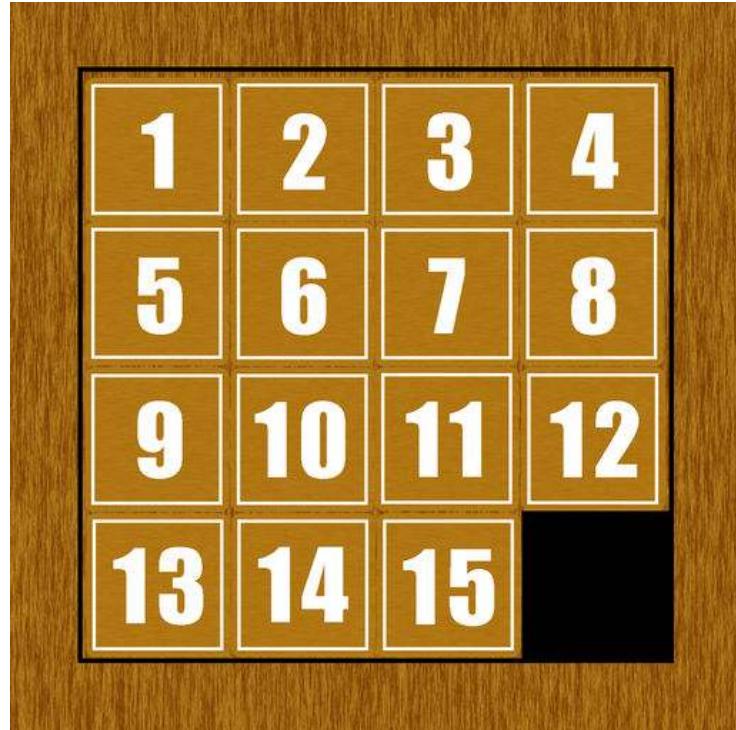
读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">773. Sliding Puzzle</a>	<a href="#">773. 滑动谜题</a>	●

阅读本文前，你需要先学习：

- 多叉树基础及遍历
- 图结构基础及通用实现
- BFS 算法核心框架

滑动拼图游戏大家应该都玩过，下图是一个 4x4 的滑动拼图：



拼图中有一个格子是空的，可以利用这个空着的格子移动其他数字。你需要通过移动这些数字，得到某个特定排列顺序，这样就算赢了。

我小时候还玩过一款叫做「华容道」的益智游戏，也和滑动拼图比较类似：



实际上，滑动拼图游戏也叫数字华容道，你看它俩挺相似的。

那么这种游戏怎么玩呢？我记得是有一些套路的，类似于魔方还原公式。但是我们今天不来研究让人头秃的技巧，**这些益智游戏通通可以用暴力搜索算法解决，所以今天我们就学以致用，用 BFS 算法框架来秒杀这些游戏。**

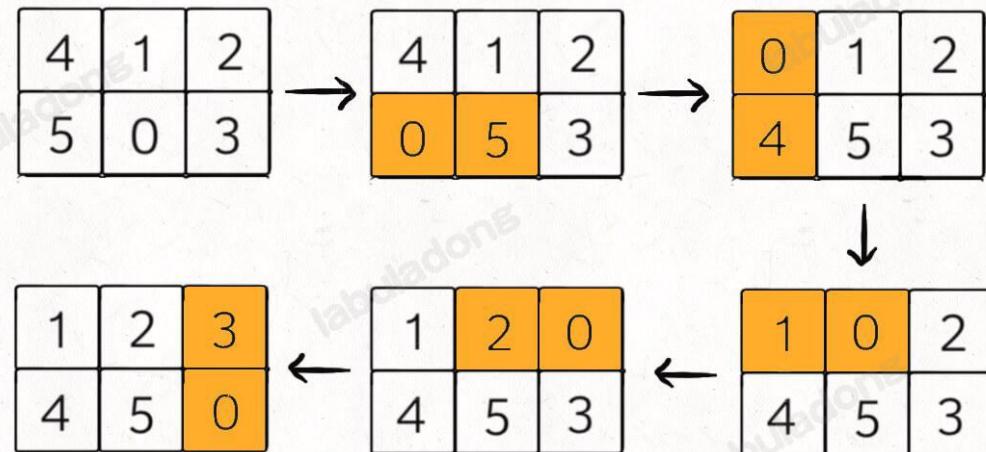
## 一、题目解析

力扣第 773 题「滑动谜题」就是这个问题，题目的要求如下：

给你一个  $2 \times 3$  的滑动拼图，用一个  $2 \times 3$  的数组 `board` 表示。拼图中有数字 0~5 六个数，其中数字 0 就表示那个空着的格子，你可以移动其中的数字，当 `board` 变为 `[[1, 2, 3], [4, 5, 0]]` 时，赢得游戏。

请你写一个算法，计算赢得游戏需要的最少移动次数，如果不能赢得游戏，返回 -1。

比如说输入的二维数组 `board = [[4, 1, 2], [5, 0, 3]]`，算法应该返回 5：



如果输入的是 `board = [[1,2,3],[5,4,0]]`, 则算法返回 -1, 因为这种局面下无论如何都不能赢得游戏。

## 二、思路分析

对于这种计算最小步数的问题, 我们就要敏感地想到 BFS 算法。

这个题目转化成 BFS 问题是有一些技巧的, 我们面临如下问题:

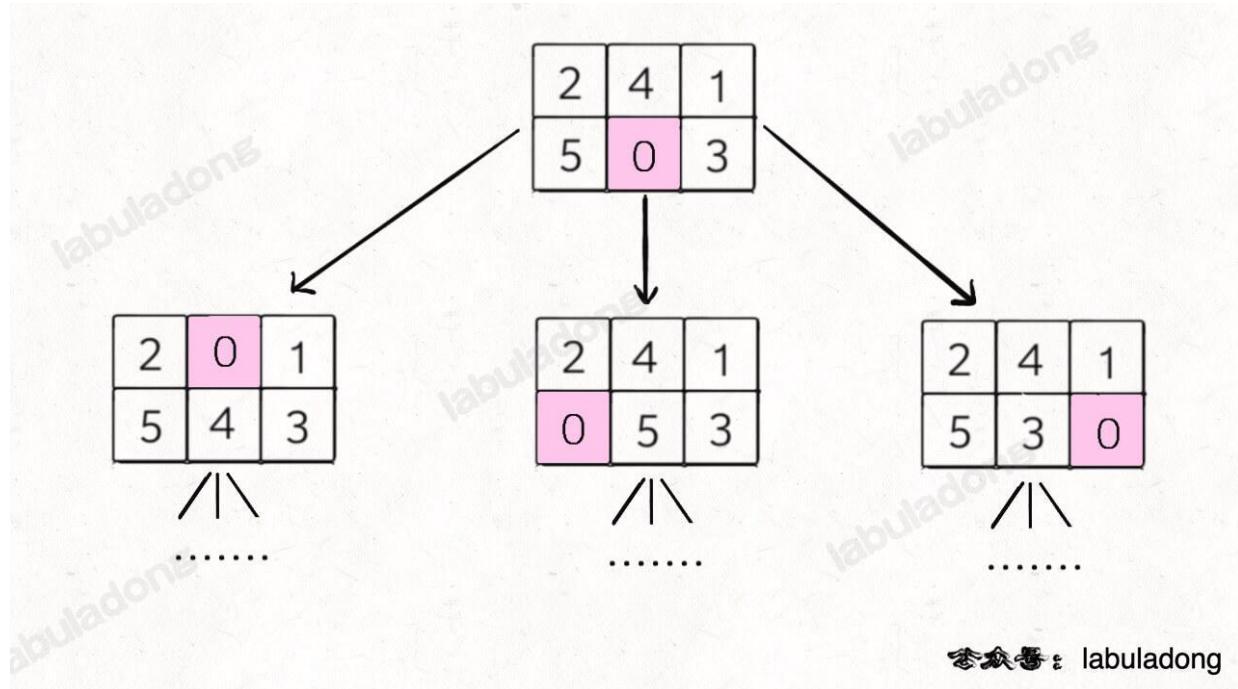
1、一般的 BFS 算法, 是从一个起点 `start` 开始, 向终点 `target` 进行寻路, 但是拼图问题不是在寻路, 而是在不断交换数字, 这应该怎么转化成 BFS 算法问题呢?

2、即便这个问题能够转化成 BFS 问题, 如何处理起点 `start` 和终点 `target`? 它们都是数组哎, 把数组放进队列, 套 BFS 框架, 想想就比较麻烦且低效。

首先回答第一个问题, **BFS 算法并不只是一个寻路算法, 而是一种暴力搜索算法**, 只要涉及暴力穷举的问题, BFS 就可以用, 而且可以最快地找到答案。

你想想计算机怎么解决问题的? 哪有什么特殊技巧, 本质上就是把所有可行解暴力穷举出来, 然后从中找到一个最优解罢了。

明白了这个道理, 我们的问题就转化成了: **如何穷举出 board 当前局面下可能衍生出的所有局面?** 这就简单了, 看数字 0 的位置呗, 和上下左右的数字进行交换就行了:



这样其实就是一个 BFS 问题, 每次先找到数字 0, 然后和周围的数字进行交换, 形成新的局面加入队列……当第一次到达 `target` 时, 就得到了赢得游戏的最少步数。

对于第二个问题, 我们这里的 `board` 仅仅是  $2 \times 3$  的二维数组, 所以可以压缩成一个一维字符串。其中比较有技巧性的点在于, 二维数组有「上下左右」的概念, 压缩成一维后, 如何得到某一个索引上下左右的索引?

对于这道题, 题目说输入的数组大小都是  $2 \times 3$ , 所以我们可以直接手动写出来这个映射:

```
// 记录一维字符串的相邻索引
int[][] neighbor = new int[][]{
    {1, 3},
    {0, 4, 2},
    {1, 5},
```

```

{0, 4},
{3, 1, 5},
{4, 2}
};

```

这个含义就是，在一维字符串中，索引  $i$  在二维数组中的相邻索引为  $\text{neighbor}[i]$ ：



$$\text{neighbor}[4] = \{1, 3, 5\}$$

© labuladong

那么对于一个  $m \times n$  的二维数组，手写它的一维索引映射肯定不现实了，如何用代码生成它的一维索引映射呢？

观察上图就能发现，如果二维数组中的某个元素  $e$  在一维数组中的索引为  $i$ ，那么  $e$  的左右相邻元素在一维数组中的索引就是  $i - 1$  和  $i + 1$ ，而  $e$  的上下相邻元素在一维数组中的索引就是  $i - n$  和  $i + n$ ，其中  $n$  为二维数组的列数。

这样，对于  $m \times n$  的二维数组，我们可以写一个函数来生成它的  $\text{neighbor}$  索引映射：

```

int[][] generateNeighborMapping(int m, int n) {
    int[][] neighbor = new int[m * n][];
    for (int i = 0; i < m * n; i++) {
        List<Integer> neighbors = new ArrayList<>();

        // 如果不是第一列，有左侧邻居
        if (i % n != 0) neighbors.add(i - 1);

        // 如果不是最后一列，有右侧邻居
        if (i % n != n - 1) neighbors.add(i + 1);

        // 如果不是第一行，有上方邻居
        if (i - n >= 0) neighbors.add(i - n);

        // 如果不是最后一行，有下方邻居
        if (i + n < m * n) neighbors.add(i + n);

        // Java 语言特性，将 List 类型转为 int[] 数组
        neighbor[i] = neighbors.stream().mapToInt(Integer::intValue).toArray();
    }
    return neighbor;
}

```

至此，我们就把这个问题完全转化成标准的 BFS 问题了，借助前文 [BFS 算法框架](#) 的代码框架，直接就可以套出解法代码了：

```

class Solution {
    public int slidingPuzzle(int[][] board) {
        int m = 2, n = 3;
        StringBuilder sb = new StringBuilder();
        String target = "123450";
        // 将 2x3 的数组转化成字符串作为 BFS 的起点
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                sb.append(board[i][j]);
            }
        }
        String start = sb.toString();

        // 记录一维字符串的相邻索引
        int[][] neighbor = new int[][]{
            {1, 3},
            {0, 4, 2},
            {1, 5},
            {0, 4},
            {3, 1, 5},
            {4, 2}
        };

        // ***** BFS 算法框架开始 *****
        Queue<String> q = new LinkedList<>();
        HashSet<String> visited = new HashSet<>();
        // 从起点开始 BFS 搜索
        q.offer(start);
        visited.add(start);

        int step = 0;
        while (!q.isEmpty()) {
            int sz = q.size();
            for (int i = 0; i < sz; i++) {
                String cur = q.poll();
                // 判断是否达到目标局面
                if (target.equals(cur)) {
                    return step;
                }
                // 找到数字 0 的索引
                int idx = 0;
                for (; cur.charAt(idx) != '0'; idx++);
                // 将数字 0 和相邻的数字交换位置
                for (int adj : neighbor[idx]) {
                    String new_board = swap(cur.toCharArray(), adj, idx);
                    // 防止走回头路
                    if (!visited.contains(new_board)) {
                        q.offer(new_board);
                        visited.add(new_board);
                    }
                }
            }
            step++;
        }
    }
}

```

```
    }
    // ***** BFS 算法框架结束 *****
    return -1;
}

private String swap(char[] chars, int i, int j) {
    char temp = chars[i];
    chars[i] = chars[j];
    chars[j] = temp;
    return new String(chars);
}
}
```

## ▶ ⭐ 代码可视化动画⭐

至此，这道题目就解决了，其实框架完全没有变，套路都是一样的，我们只是花了比较多的时间将滑动拼图游戏转化成 BFS 算法。

很多益智游戏都是这样，虽然看起来特别巧妙，但都架不住暴力穷举，常用的算法就是回溯算法或者 BFS 算法。

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">365. Water and Jug Problem</a>	<a href="#">365. 水壶问题</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】BFS 经典习题 I

阅读本文前，你需要先学习：

- 二叉树的遍历
- 多叉树的遍历
- 图结构基础及通用代码实现
- 图结构的遍历
- BFS 算法核心框架

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 【强化练习】BFS 经典习题 II

阅读本文前，你需要先学习：

- 二叉树的遍历
- 多叉树的遍历
- 图结构基础及通用代码实现
- 图结构的遍历
- BFS 算法核心框架

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 动态规划解题套路框架



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">509. Fibonacci Number</a>	<a href="#">509. 斐波那契数</a>	
<a href="#">322. Coin Change</a>	<a href="#">322. 零钱兑换</a>	

阅读本文前，你需要先学习：

- [二叉树结构基础](#)
- [二叉树的遍历框架](#)
- [多叉树结构及遍历框架](#)

tip：本文有视频版：[动态规划框架套路详解](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

这篇文章是我多年前一篇 200 多赞赏的 [动态规划详解](#) 的进阶版，我添加了更多干货内容，希望本文成为解决动态规划的一部「指导方针」。

动态规划问题（Dynamic Programming）应该是很多读者头疼的，不过这类问题也是最具有技巧性，最有意思的。本书使用了整整一个章节专门来写这个算法，动态规划的重要性也可见一斑。

本文解决几个问题：

动态规划是什么？解决动态规划问题有什么技巧？如何学习动态规划？

刷题刷多了就会发现，算法技巧就那几个套路，我们后续的动态规划系列章节，都在使用本文的解题框架思维，如果你心里有数，就会轻松很多。所以本文放在第一章，希望能够成为解决动态规划问题的一部指导方针，下面上干货。

首先，**动态规划问题的一般形式就是求最值**。动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求最长递增子序列呀，最小编辑距离呀等等。

既然是要求最值，核心问题是什么呢？**求解动态规划的核心问题是穷举**。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。

动态规划这么简单，就是穷举就完事了？我看到的动态规划问题都很难啊！

首先，虽然动态规划的核心思想就是穷举求最值，但是问题可以千变万化，穷举所有可行解其实并不是一件容易的事，需要你熟练掌握递归思维，只有列出**正确的「状态转移方程」**，才能正确地穷举。而且，你需要判断算法问题是否具备**「最优子结构」**，是否能够通过子问题的最值得到原问题的最值。另外，动态规划问题存在**「重叠子问题」**，如果暴力穷举的话效率会很低，所以需要你使用**「备忘录」**或者**「DP table」**来优化穷举过程，避免不必要的计算。

以上提到的重叠子问题、最优子结构、状态转移方程就是动态规划三要素。具体什么意思等会会举例详解，但是在实际的算法问题中，写出状态转移方程是最困难的，这也就是为什么很多朋友觉得动态规划问题困难的原因，我来提供我总结的一个思维框架，辅助你思考状态转移方程：

明确「状态」 -> 明确「选择」 -> 定义 dp 数组/函数的含义。

按上面的套路走，最后的解法代码就会是如下的框架：

```
# 自顶向下递归的动态规划
def dp(状态1, 状态2, ...):
    for 选择 in 所有可能的选择:
        # 此时的状态已经因为做了选择而改变
        result = 求最值(result, dp(状态1, 状态2, ...))
    return result

# 自底向上迭代的动态规划
# 初始化 base case
dp[0][0][...] = base case
# 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...)
```

下面通过斐波那契数列问题和凑零钱问题来详解动态规划的基本原理。前者主要是让你明白什么是重叠子问题（斐波那契数列没有求最值，所以严格来说不是动态规划问题），后者主要举集中于如何列出状态转移方程。

## 一、斐波那契数列

力扣第 509 题「斐波那契数」就是这个问题，请读者不要嫌弃这个例子简单，只有简单的例子才能让你把精力充分集中在算法背后的通用思想和技巧上，而不会被那些隐晦的细节问题搞的莫名其妙。想要困难的例子，接下来的动态规划系列里有的是。

### 暴力递归

斐波那契数列的数学形式就是递归的，写成代码就是这样：

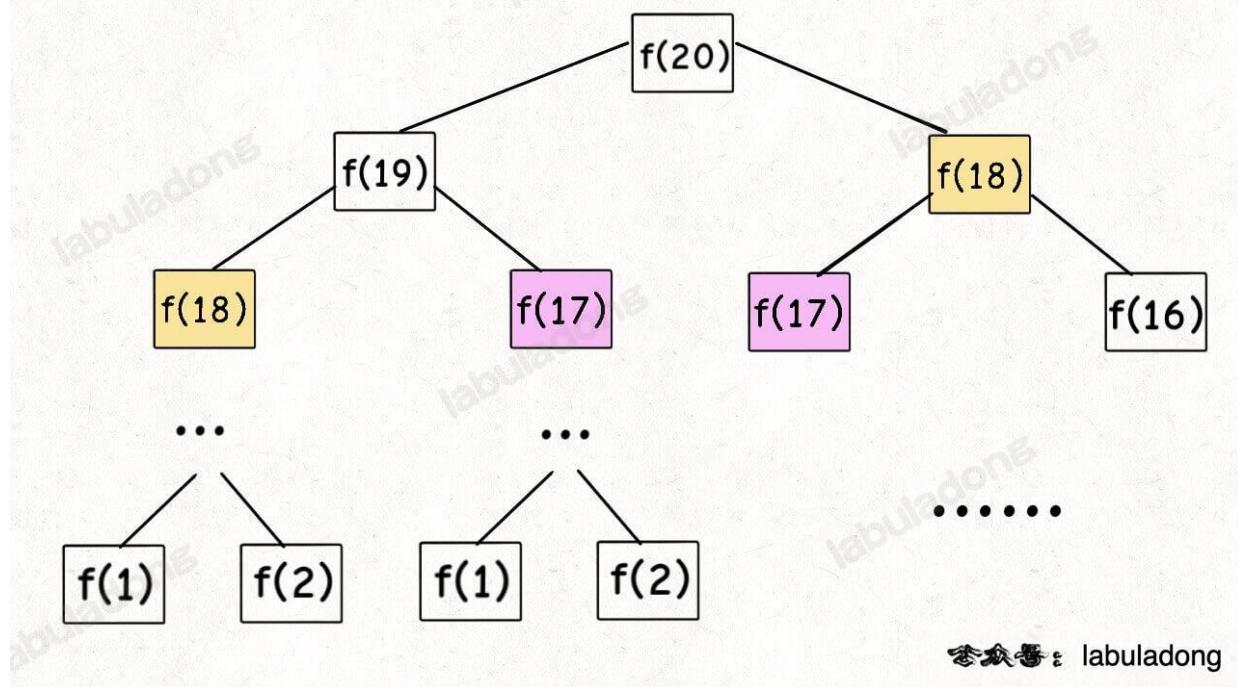
```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

---

### ▶ ⭐ 代码可视化动画⭐

---

这个不用多说了，学校老师讲递归的时候似乎都是拿这个举例。我们也知道这样写代码虽然简洁易懂，但是十分低效，低效在哪里？假设  $n = 20$ ，请画出递归树：



但凡遇到需要递归的问题，最好都画出递归树，这对你分析算法的复杂度，寻找算法低效的原因都有巨大帮助。

这个递归树怎么理解？就是说想要计算原问题  $f(20)$ ，我就得先计算出子问题  $f(19)$  和  $f(18)$ ，然后要计算  $f(19)$ ，我就得先算出子问题  $f(18)$  和  $f(17)$ ，以此类推。最后遇到  $f(1)$  或者  $f(2)$  的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

首先计算子问题个数，即递归树中节点的总数。显然二叉树节点总数为指数级别，所以子问题个数为  $O(2^n)$ 。

然后计算解决一个子问题的时间，在本算法中，没有循环，只有  $f(n - 1) + f(n - 2)$  一个加法操作，时间为  $O(1)$ 。

所以，这个算法的时间复杂度为二者相乘，即  $O(2^n)$ ，指数级别，爆炸。

观察递归树，很明显发现了算法低效的原因：存在大量重复计算，比如  $f(18)$  被计算了两次，而且你可以看到，以  $f(18)$  为根的这个递归树体量巨大，多算一遍，会耗费巨大的时间。更何况，还不止  $f(18)$  这一个节点被重复计算，所以这个算法及其低效。

这就是动态规划问题的第一个性质：重叠子问题。下面，我们想办法解决这个问题。

## 带备忘录的递归解法

明确了问题，其实就已经把问题解决了一半。既然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。

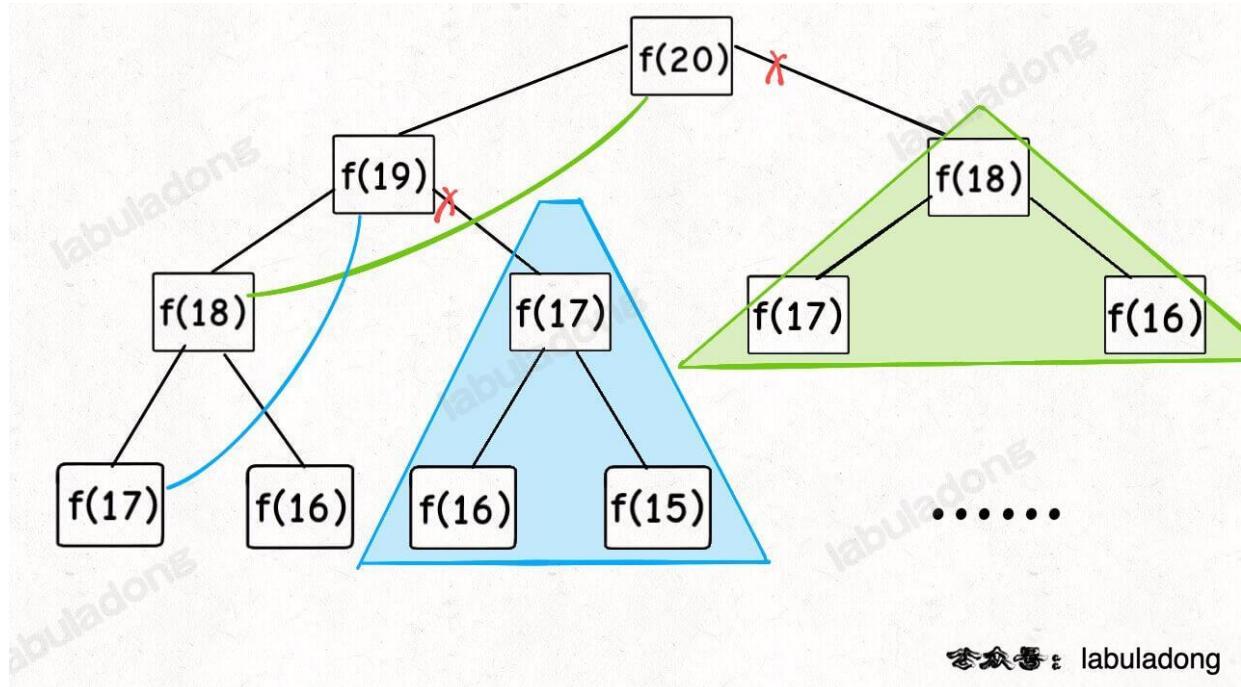
一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

```
int fib(int N) {
    // 备忘录全初始化为 0
    int[] memo = new int[N + 1];
    // 进行带备忘录的递归
    return dp(memo, N);
}
```

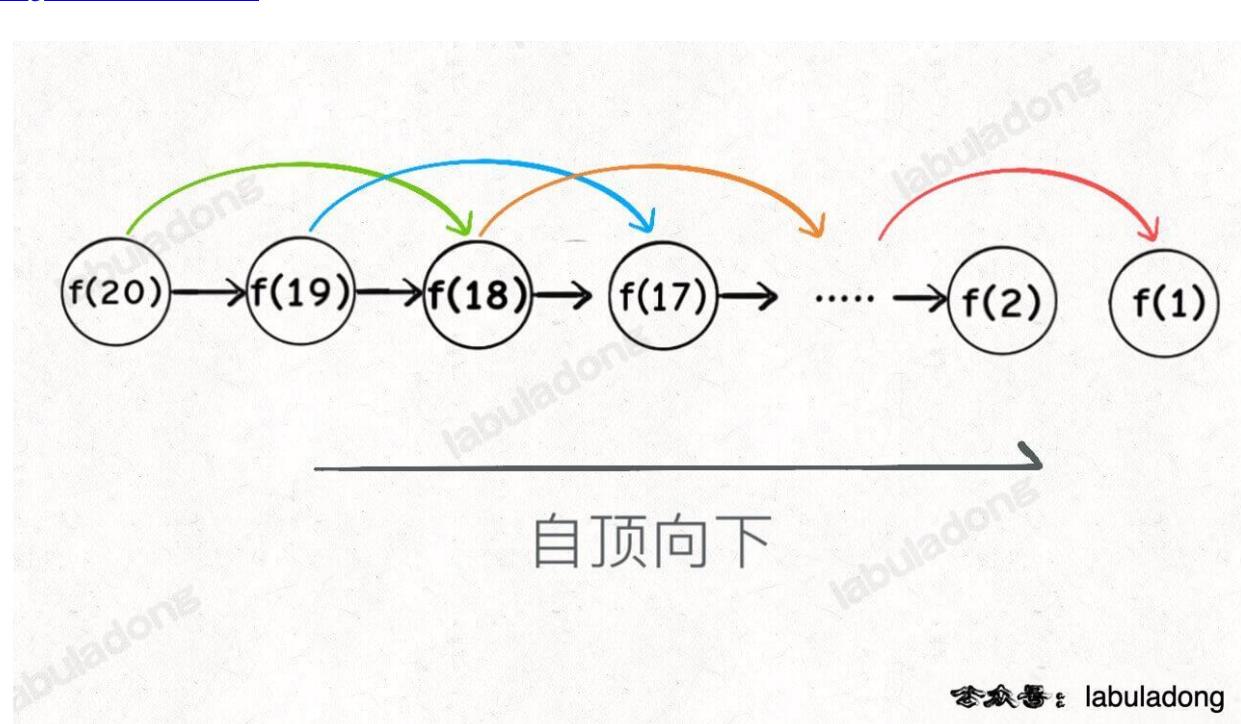
```
// 带着备忘录进行递归
int dp(int[] memo, int n) {
    // base case
    if (n == 0 || n == 1) return n;
    // 已经计算过，不用再计算了
    if (memo[n] != 0) return memo[n];
    memo[n] = dp(memo, n - 1) + dp(memo, n - 2);
    return memo[n];
}
```

### ▶ 代码可视化动画

现在，画出递归树，你就知道「备忘录」到底做了什么。



实际上，带「备忘录」的递归算法，把一棵存在巨量冗余的递归树通过「剪枝」，改造成了一幅不存在冗余的递归图，极大减少了子问题（即递归图中节点）的个数。



递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

子问题个数，即图中节点的总数，由于本算法不存在冗余计算，子问题就是  $f(1), f(2), f(3) \dots f(20)$ ，数量和输入规模  $n = 20$  成正比，所以子问题个数为  $O(n)$ 。

解决一个子问题的时间，同上，没有什么循环，时间为  $O(1)$ 。

所以，本算法的时间复杂度是  $O(n)$ ，比起暴力算法，是降维打击。

至此，带备忘录的递归解法的效率已经和迭代的动态规划解法一样了。实际上，这种解法和常见的动态规划解法已经差不多了，只不过这种解法是「自顶向下」进行「递归」求解，我们更常见的动态规划代码是「自底向上」进行「递推」求解。

啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说  $f(20)$ ，向下逐渐分解规模，直到  $f(1)$  和  $f(2)$  这两个 base case，然后逐层返回答案，这就叫「自顶向下」。

啥叫「自底向上」？反过来，我们直接从最底下、最简单、问题规模最小、已知结果的  $f(1)$  和  $f(2)$  (base case) 开始往上推，直到推到我们想要的答案  $f(20)$ 。这就是「递推」的思路，这也是动态规划一般都脱离了递归，而是由循环迭代完成计算的原因。

## dp 数组的迭代（递推）解法

有了上一步「备忘录」的启发，我们可以把这个「备忘录」独立出来成为一张表，通常叫做 DP table，在这张表上完成「自底向上」的推算岂不美哉！

```
int fib(int N) {
    if (N == 0) return 0;
    int[] dp = new int[N + 1];
    // base case
    dp[0] = 0; dp[1] = 1;
    // 状态转移
    for (int i = 2; i <= N; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}
```

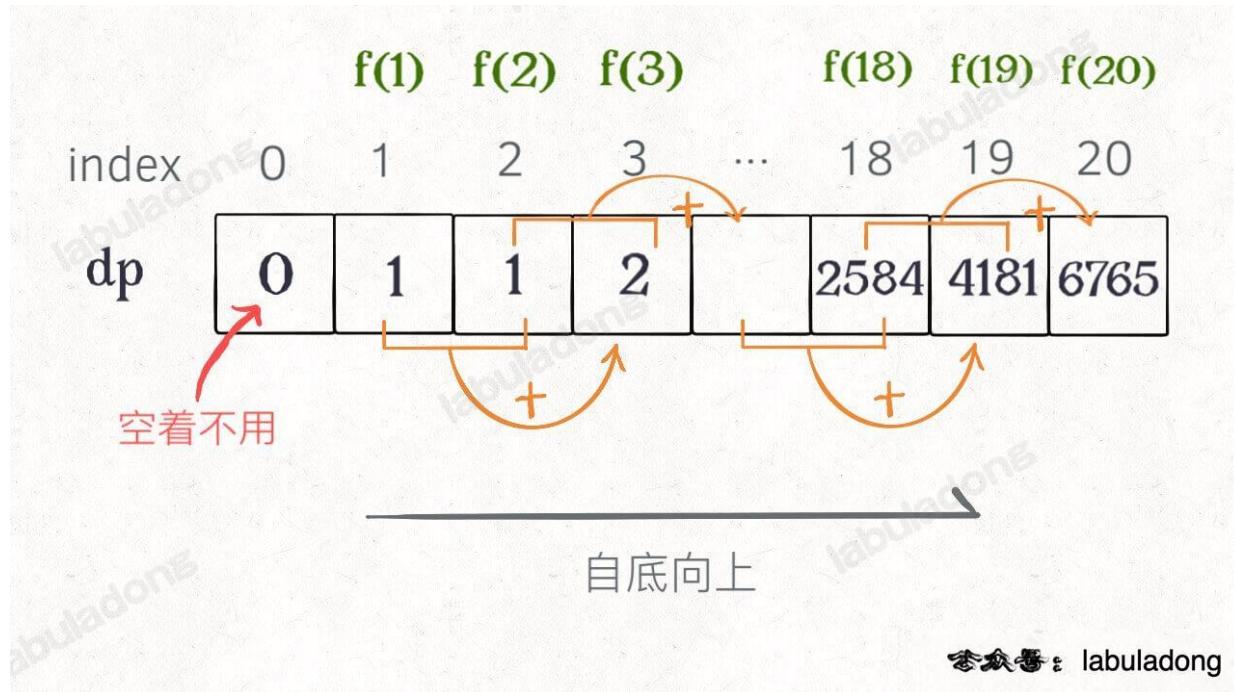
```

    return dp[N];
}

```

### ▶ 彩虹 代码可视化动画

画个图就很好理解了，而且你发现这个 DP table 特别像之前那个「剪枝」后的结果，只是反过来算而已：



实际上，带备忘录的递归解法中的那个「备忘录」`memo` 数组，最终完成后就是这个解法中的 `dp` 数组，你对比一下可视化面板中两个算法执行的过程可以更直观地看出它俩的联系。

所以说自顶向下、自底向上两种解法本质其实是差不多的，大部分情况下，效率也基本相同。

## 拓展延伸

这里，引出「状态转移方程」这个名词，实际上就是描述问题结构的数学形式：

$$f(n) = \begin{cases} 1, & n = 1, 2 \\ f(n - 1) + f(n - 2), & n > 2 \end{cases}$$

为啥叫「状态转移方程」？其实就是为了听起来高端。

`f(n)` 的函数参数会不断变化，所以你把参数 `n` 想做一个状态，这个状态 `n` 是由状态 `n - 1` 和状态 `n - 2` 转移（相加）而来，这就叫状态转移，仅此而已。

你会发现，上面的几种解法中的所有操作，例如 `return f(n - 1) + f(n - 2)`, `dp[i] = dp[i - 1] + dp[i - 2]`，以及对备忘录或 DP table 的初始化操作，都是围绕这个方程的不同表现形式。

可见列出「状态转移方程」的重要性，它是解决问题的核心，而且很容易发现，其实状态转移方程直接代表着暴力解法。

千万不要看不起暴力解，动态规划问题最困难的就是写出这个暴力解，即状态转移方程。

只要写出暴力解，优化方法无非是用备忘录或者 DP table，再无奥妙可言。

这个例子的最后，讲一个细节优化。

细心的读者会发现，根据斐波那契数列的状态转移方程，当前状态  $n$  只和之前的  $n-1$ ,  $n-2$  两个状态有关，其实并不需要那么长的一个 DP table 来存储所有的状态，只要想办法存储之前的两个状态就行了。

所以，可以进一步优化，把空间复杂度降为  $O(1)$ 。这也就是我们最常见的计算斐波那契数的算法：

```
int fib(int n) {
    if (n == 0 || n == 1) {
        // base case
        return n;
    }
    // 分别代表 dp[i - 1] 和 dp[i - 2]
    int dp_i_1 = 1, dp_i_2 = 0;
    for (int i = 2; i <= n; i++) {
        // dp[i] = dp[i - 1] + dp[i - 2];
        int dp_i = dp_i_1 + dp_i_2;
        // 滚动更新
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i_1;
}
```

## ▶ 代码可视化动画

这一般是动态规划问题的最后一步优化，如果我们发现每次状态转移只需要 DP table 中的一部分，那么可以尝试缩小 DP table 的大小，只记录必要的数据，从而降低空间复杂度。

上述例子就相当于把 DP table 的大小从  $n$  缩小到 2，即把空间复杂度下降了一个量级。我会在后文 [对动态规划发动降维打击](#) 进一步讲解这个压缩空间复杂度的技巧，一般来说用来把一个二维的 DP table 压缩成一维，即把空间复杂度从  $O(n^2)$  压缩到  $O(n)$ 。

有人会问，动态规划的另一个重要特性「最优子结构」，怎么没有涉及？下面会涉及。斐波那契数列的例子严格来说不算动态规划，因为没有涉及求最值，以上旨在说明重叠子问题的消除方法，演示得到最优解法逐步求精的过程。下面，看第二个例子，凑零钱问题。

## 二、凑零钱问题

这是力扣第 322 题「零钱兑换」：

给你  $k$  种面值的硬币，面值分别为  $c_1, c_2 \dots c_k$ ，每种硬币的数量无限，再给一个总金额  $amount$ ，问你最少需要几枚硬币凑出这个金额，如果不可能凑出，算法返回 -1。算法的函数签名如下：

```
// coins 中是可选硬币面值，amount 是目标金额
int coinChange(int[] coins, int amount);
```

比如说  $k = 3$ ，面值分别为 1, 2, 5，总金额  $amount = 11$ 。那么最少需要 3 枚硬币凑出，即  $11 = 5 + 5 + 1$ 。

你认为计算机应该如何解决这个问题？显然，就是把所有可能的凑硬币方法都穷举出来，然后找找看最少需要多少枚硬币。

## 暴力递归

首先，这个问题是动态规划问题，因为它具有「最优子结构」的。要符合「最优子结构」，子问题间必须互相独立。啥叫相互独立？你肯定不想看数学证明，我用一个直观的例子来讲解。

比如说，假设你考试，每门科目的成绩都是互相独立的。你的原问题是考出最高的总成绩，那么你的子问题就是要把语文考到最高，数学考到最高……为了每门课考到最高，你要把每门课相应的选择题分数拿到最高，填空题分数拿到最高……当然，最终就是你每门课都是满分，这就是最高的总成绩。

得到了正确的结果：最高的总成绩就是总分。因为这个过程符合最优子结构，「每门科目考到最高」这些子问题是互相独立，互不干扰的。

但是，如果加一个条件：你的语文成绩和数学成绩会互相制约，不能同时达到满分，数学分数高，语文分数就会降低，反之亦然。

这样的话，显然你能考到的最高总成绩就达不到总分了，按刚才那个思路就会得到错误的结果。因为「每门科目考到最高」的子问题并不独立，语文数学成绩互相影响，无法同时最优，所以最优子结构被破坏。

回到凑零钱问题，为什么说它符合最优子结构呢？假设你有面值为 1, 2, 5 的硬币，你想求 `amount = 11` 时的最少硬币数（原问题），如果你知道凑出 `amount = 10, 9, 6` 的最少硬币数（子问题），你只需要把子问题的答案加一（再选一枚面值为 1, 2, 5 的硬币），求个最小值，就是原问题的答案。因为硬币的数量是没限制的，所以子问题之间没有相互制约，是互相独立的。

关于最优子结构的问题，后文 [动态规划答疑篇](#) 还会再举例探讨。

那么，既然知道了这是个动态规划问题，就要思考如何列出正确的状态转移方程？

1、确定「状态」，也就是原问题和子问题中会变化的变量。由于硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 `amount`。

2、确定「选择」，也就是导致「状态」产生变化的行为。目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额。所以说所有硬币的面值，就是你的「选择」。

3、明确 `dp` 函数/数组的定义。我们这里讲的是自顶向下的解法，所以会有一个递归的 `dp` 函数，一般来说函数的参数就是状态转移中会变化的量，也就是上面说到的「状态」；函数的返回值就是题目要求我们计算的量。就本题来说，状态只有一个，即「目标金额」，题目要求我们计算凑出目标金额所需的最少硬币数量。

所以我们可以这样定义 `dp` 函数：`dp(n)` 表示，输入一个目标金额 `n`，返回凑出目标金额 `n` 所需的最少硬币数量。

那么根据这个定义，我们的最终答案就是 `dp(amount)` 的返回值。

搞清楚上面这几个关键点，解法的伪码就可以写出来了：

```
// 伪码框架
int coinChange(int[] coins, int amount) {
    // 题目要求的最终结果是 dp(amount)
    return dp(coins, amount);
}

// 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
int dp(int[] coins, int n) {
    // 做选择，选择需要硬币最少的那个结果
    for (int coin : coins) {
        res = min(res, 1 + dp(coins, n - coin));
    }
}
```

```
    return res;
}
```

根据伪码，我们加上 base case 即可得到最终的答案。显然目标金额为 0 时，所需硬币数量为 0；当目标金额小于 0 时，无解，返回 -1：

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        // 题目要求的最终结果是 dp(amount)
        return dp(coins, amount);
    }

    // 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
    int dp(int[] coins, int amount) {
        // base case
        if (amount == 0) return 0;
        if (amount < 0) return -1;

        int res = Integer.MAX_VALUE;
        for (int coin : coins) {
            // 计算子问题的结果
            int subProblem = dp(coins, amount - coin);
            // 子问题无解则跳过
            if (subProblem == -1) continue;
            // 在子问题中选择最优解，然后加一
            res = Math.min(res, subProblem + 1);
        }

        return res == Integer.MAX_VALUE ? -1 : res;
    }
}
```

这里 `coinChange` 和 `dp` 函数的签名完全一样，所以理论上不需要额外写一个 `dp` 函数。但为了后文讲解方便，这里还是另写一个 `dp` 函数来实现主要逻辑。

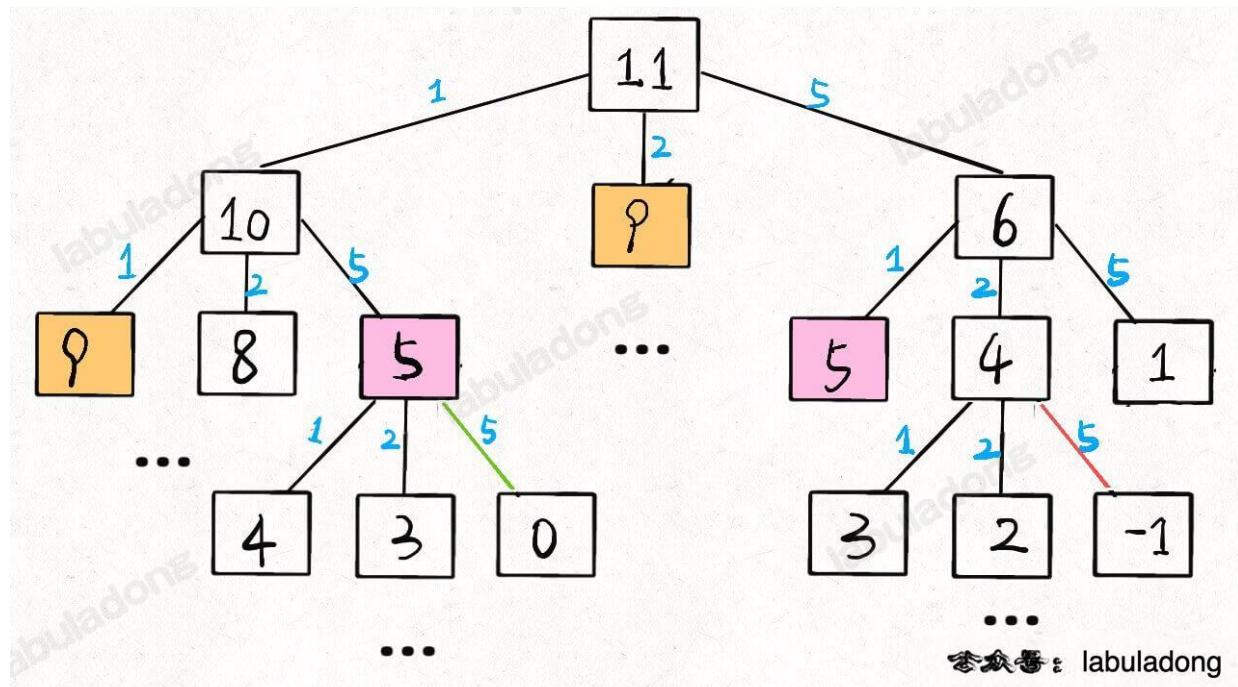
另外，我经常看到有读者留言问，子问题的结果为什么要加 1 (`subProblem + 1`)，而不是加硬币金额之类的。我这里统一提示一下，动态规划问题的关键是 `dp` 函数/数组的定义，你这个函数的返回值代表什么？你回过头去搞清楚这一点，然后就知道为什么要给子问题的返回值加 1 了。

### ▶ 🎨 代码可视化动画🎨

至此，状态转移方程其实已经完成了，以上算法已经是暴力解法了，以上代码的数学形式就是状态转移方程：

$$dp(n) = \begin{cases} 0, & n = 0 \\ -1, & n < 0 \\ \min\{dp(n - coin) + 1 \mid coin \in coins\}, & n > 0 \end{cases}$$

至此，这个问题其实就解决了，只不过需要消除一下重叠子问题，比如  $\text{amount} = 11$ ,  $\text{coins} = \{1, 2, 5\}$  时画出递归树看看：



递归算法的时间复杂度分析：子问题总数  $\times$  解决每个子问题所需的时间。

子问题总数为递归树的节点个数，但算法会进行剪枝，剪枝的时机和题目给定的具体硬币面额有关，所以可以想象，这棵树生长的并不规则，确切算出树上有多少节点是比较困难的。对于这种情况，我们一般的做法是按照最坏的情况估算一个时间复杂度的上界。

假设目标金额为  $n$ ，给定的硬币个数为  $k$ ，那么递归树最坏情况下高度为  $n$ （全用面额为 1 的硬币），然后再假设这是一棵满  $k$  叉树，则节点的总数在  $k^n$  这个数量级。

接下来看每个子问题的复杂度，由于每次递归包含一个 for 循环，复杂度为  $O(k)$ ，相乘得到总时间复杂度为  $O(k^n)$ ，指数级别。

## 带备忘录的递归

类似之前斐波那契数列的例子，只需要稍加修改，就可以通过备忘录消除子问题：

```
class Solution {
    int[] memo;

    public int coinChange(int[] coins, int amount) {
        memo = new int[amount + 1];
        // 备忘录初始化为一个不会被取到的特殊值，代表还未被计算
        Arrays.fill(memo, -666);

        return dp(coins, amount);
    }

    int dp(int[] coins, int amount) {
        if (amount == 0) return 0;
        if (amount < 0) return -1;
        // 查备忘录，防止重复计算
        if (memo[amount] != -666)
            return memo[amount];
        else
            memo[amount] = dp(coins, amount - 1) + 1;
    }
}
```

```

int res = Integer.MAX_VALUE;
for (int coin : coins) {
    // 计算子问题的结果
    int subProblem = dp(coins, amount - coin);
    // 子问题无解则跳过
    if (subProblem == -1) continue;
    // 在子问题中选择最优解，然后加一
    res = Math.min(res, subProblem + 1);
}
// 把计算结果存入备忘录
memo[amount] = (res == Integer.MAX_VALUE) ? -1 : res;
return memo[amount];
}
}

```

## ▶ 🎨 代码可视化动画

不画图了，很显然「备忘录」大大减小了子问题数目，完全消除了子问题的冗余，所以子问题总数不会超过金额数  $n$ ，即子问题数目为  $O(n)$ 。处理一个子问题的时间不变，仍是  $O(k)$ ，所以总的时间复杂度是  $O(kn)$ 。

## dp 数组的迭代解法

当然，我们也可以自底向上使用 dp table 来消除重叠子问题，关于「状态」「选择」和 base case 与之前没有区别，dp 数组的定义和刚才 dp 函数类似，也是把「状态」，也就是目标金额作为变量。不过 dp 函数体现在函数参数，而 dp 数组体现在数组索引：

**dp 数组的定义：当目标金额为  $i$  时，至少需要  $dp[i]$  枚硬币凑出。**

根据我们文章开头给出的动态规划代码框架可以写出如下解法：

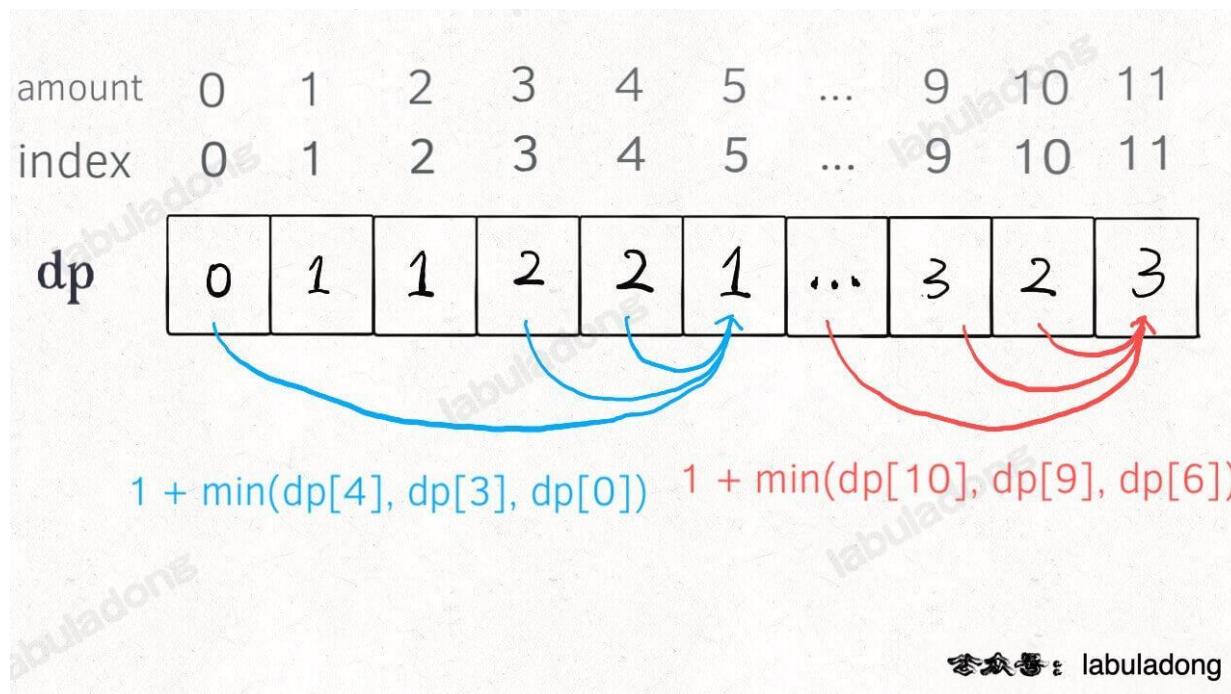
```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        // 数组大小为 amount + 1，初始值也为 amount + 1
        Arrays.fill(dp, amount + 1);

        // base case
        dp[0] = 0;
        // 外层 for 循环在遍历所有状态的所有取值
        for (int i = 0; i < dp.length; i++) {
            // 内层 for 循环在求所有选择的最小值
            for (int coin : coins) {
                // 子问题无解，跳过
                if (i - coin < 0) {
                    continue;
                }
                dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
            }
        }
        return (dp[amount] == amount + 1) ? -1 : dp[amount];
    }
}

```

为啥 `dp` 数组中的值都初始化为 `amount + 1` 呢，因为凑成 `amount` 金额的硬币数最多只可能等于 `amount`（全用 1 元面值的硬币），所以初始化为 `amount + 1` 就相当于初始化为正无穷，便于后续取最小值。为啥不直接初始化为 `int` 型的最大值 `Integer.MAX_VALUE` 呢？因为后面有 `dp[i - coin] + 1`，这就会导致整型溢出。



### 三、最后总结

第一个斐波那契数列的问题，解释了如何通过「备忘录」或者「dp table」的方法来优化递归树，并且明确了这两种方法本质上是一样的，只是自顶向下和自底向上的不同而已。

第二个凑零钱的问题，展示了如何流程化确定「状态转移方程」，只要通过状态转移方程写出暴力递归解，剩下的也就是优化递归树，消除重叠子问题而已。

如果你不太了解动态规划，还能看到这里，真得给你鼓掌，相信你已经掌握了这个算法的设计技巧。

计算机解决问题其实没有任何特殊的技巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考“如何穷举”，然后再追求“如何聪明地穷举”。

列出状态转移方程，就是在解决“如何穷举”的问题。之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不容易穷举完整。

备忘录、DP table 就是在追求“如何聪明地穷举”。用空间换时间的思路，是降低时间复杂度的不二法门，除此之外，试问，还能玩出啥花活？

之后我们会有一章专门讲解动态规划问题，如果有任何问题都可以随时回来重读本文，希望读者在阅读每个题目和解法时，多往「状态」和「选择」上靠，才能对这套框架产生自己的理解，运用自如。

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">111. Minimum Depth of Binary Tree</a>	111. 二叉树的最小深度	●
<a href="#">112. Path Sum</a>	112. 路径总和	●

LeetCode	力扣	难度
115. Distinct Subsequences	115. 不同的子序列	●
139. Word Break	139. 单词拆分	●
1696. Jump Game VI	1696. 跳跃游戏 VI	●
221. Maximal Square	221. 最大正方形	●
240. Search a 2D Matrix II	240. 搜索二维矩阵 II	●
256. Paint House	256. 粉刷房子	●
279. Perfect Squares	279. 完全平方数	●
343. Integer Break	343. 整数拆分	●
365. Water and Jug Problem	365. 水壶问题	●
542. 01 Matrix	542. 01 矩阵	●
576. Out of Boundary Paths	576. 出界的路径数	●
62. Unique Paths	62. 不同路径	●
63. Unique Paths II	63. 不同路径 II	●
70. Climbing Stairs	70. 爬楼梯	●
91. Decode Ways	91. 解码方法	●
-	剑指 Offer 04. 二维数组中的查找	●
-	剑指 Offer 10- I. 斐波那契数列	●
-	剑指 Offer 10- II. 青蛙跳台阶问题	●
-	剑指 Offer 14- I. 剪绳子	●
-	剑指 Offer 46. 把数字翻译成字符串	●
-	剑指 Offer II 091. 粉刷房子	●
-	剑指 Offer II 097. 子序列的数目	●
-	剑指 Offer II 098. 路径的数目	●
-	剑指 Offer II 103. 最少的硬币数目	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](http://labuladong.online)。

# 动态规划设计：最长递增子序列



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">300. Longest Increasing Subsequence</a>	<a href="#">300. 最长递增子序列</a>	🟡
<a href="#">354. Russian Doll Envelopes</a>	<a href="#">354. 俄罗斯套娃信封问题</a>	🔴

-----

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

也许有读者看了前文 [动态规划详解](#)，学会了动态规划的套路：找到了问题的「状态」，明确了 `dp` 数组/函数的含义，定义了 base case；但是不知道如何确定「选择」，也就是找不到状态转移的关系，依然写不出动态规划解法，怎么办？

不要担心，动态规划的难点本来就在于寻找正确的状态转移方程，本文就借助经典的「最长递增子序列问题」来讲一讲设计动态规划的通用技巧：[数学归纳思想](#)。

最长递增子序列（Longest Increasing Subsequence，简写 LIS）是非常经典的一个算法问题，比较容易想到的是动态规划解法，时间复杂度  $O(N^2)$ ，我们借这个问题来由浅入深讲解如何找状态转移方程，如何写出动态规划解法。比较难想到的是利用二分查找，时间复杂度是  $O(N \log N)$ ，我们通过一种简单的纸牌游戏来辅助理解这种巧妙的解法。

力扣第 300 题「最长递增子序列」就是这个问题：

## ▼ 300. 最长递增子序列 [Leetcode](#) | [力扣](#)

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

**子序列** 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3, 6, 2, 7]` 是数组 `[0, 3, 1, 6, 2, 2, 7]` 的子序列。

### 示例 1：

```
输入: nums = [10, 9, 2, 5, 3, 7, 101, 18]
输出: 4
解释: 最长递增子序列是 [2, 3, 7, 101]，因此长度为 4 。
```

### 示例 2：

```
输入: nums = [0,1,0,3,2,3]
输出: 4
```

### 示例 3:

```
输入: nums = [7,7,7,7,7,7,7]
输出: 1
```

### 提示:

- $1 \leq \text{nums.length} \leq 2500$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

### 进阶:

- 你能将算法的时间复杂度降低到  $O(n \log(n))$  吗?

```
// 函数签名
int lengthOfLIS(int[] nums);
```

比如说输入  $\text{nums}=[10,9,2,5,3,7,101,18]$ , 其中最长的递增子序列是  $[2,3,7,101]$ , 所以算法的输出应该是 4。

注意「子序列」和「子串」这两个名词的区别, 子串一定是连续的, 而子序列不一定是连续的。下面先来设计动态规划算法解决这个问题。

## 一、动态规划解法

动态规划的核心设计思想是数学归纳法。

相信大家对数学归纳法都不陌生, 高中就学过, 而且思路很简单。比如我们想证明一个数学结论, 那么我们先假设这个结论在  $k < n$  时成立, 然后根据这个假设, 想办法推导证明出  $k = n$  的时候此结论也成立。如果能够证明出来, 那么就说明这个结论对于  $k$  等于任何数都成立。

类似的, 我们设计动态规划算法, 不是需要一个  $dp$  数组吗? 我们可以假设  $dp[0 \dots i-1]$  都已经被算出来了, 然后问自己: 怎么通过这些结果算出  $dp[i]$ ?

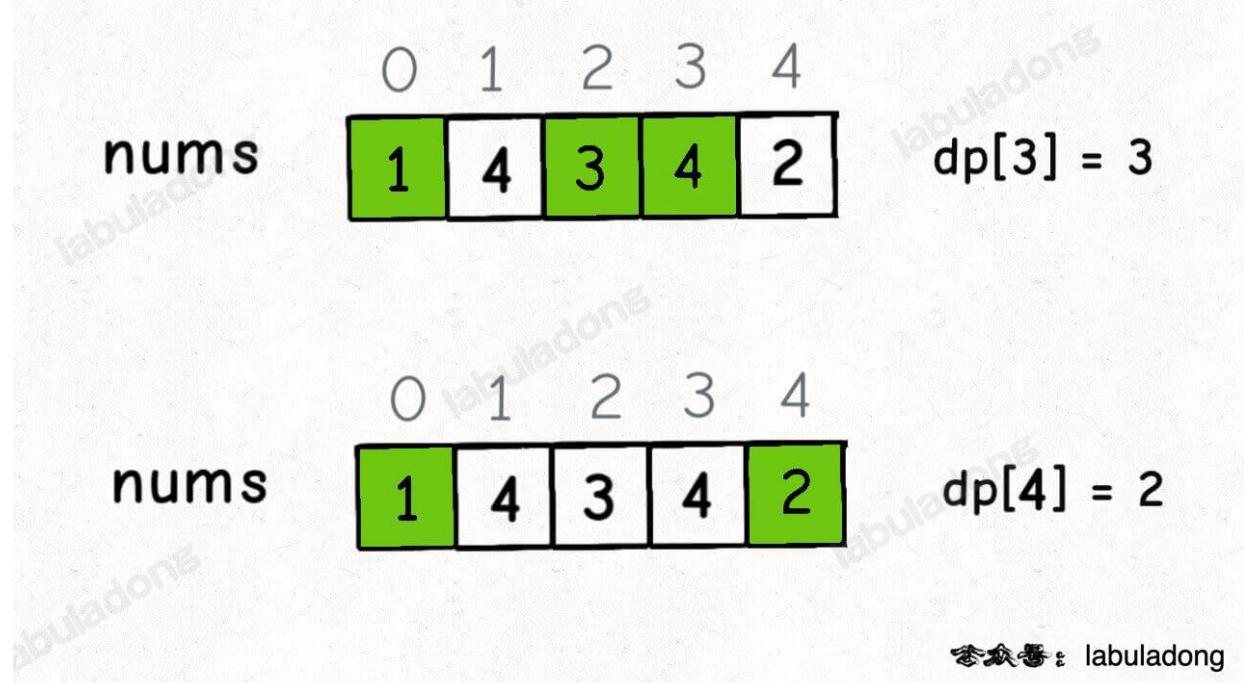
直接拿最长递增子序列这个问题举例你就明白了。不过, 首先要定义清楚  $dp$  数组的含义, 即  $dp[i]$  的值到底代表着什么?

我们的定义是这样的:  $dp[i]$  表示以  $\text{nums}[i]$  这个数结尾的最长递增子序列的长度。

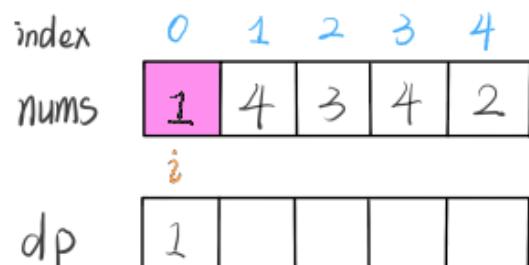
为什么这样定义呢? 这是解决子序列问题的一个套路, 后文 [动态规划之子序列问题解题模板](#) 总结了几种常见套路。你读完本章所有的动态规划问题, 就会发现  $dp$  数组的定义方法也就那几种。

根据这个定义, 我们就可以推出 base case:  $dp[i]$  初始值为 1, 因为以  $\text{nums}[i]$  结尾的最长递增子序列起码要包含它自己。

举两个例子:



这个 GIF 展示了算法演进的过程：



根据这个定义，我们的最终结果（子序列的最大长度）应该是 `dp` 数组中的最大值。

```
int res = 0;
for (int i = 0; i < dp.length; i++) {
    res = Math.max(res, dp[i]);
}
return res;
```

读者也许会问，刚才的算法演进过程中每个 `dp[i]` 的结果是我们肉眼看出来的，我们应该怎么设计算法逻辑来正确计算每个 `dp[i]` 呢？

这就是动态规划的重头戏，如何设计算法逻辑进行状态转移，才能正确运行呢？这里需要使用数学归纳的思想：

假设我们已经知道了 `dp[0..4]` 的所有结果，我们如何通过这些已知结果推出 `dp[5]` 呢？

0 1 2 3 4 5

nums

1	4	3	4	2	3
---	---	---	---	---	---

dp

1	2	2	3	2	?
---	---	---	---	---	---

公众号：labuladong

根据刚才我们对 `dp` 数组的定义，现在想求 `dp[5]` 的值，也就是想求以 `nums[5]` 为结尾的最长递增子序列。

`nums[5] = 3`，既然是递增子序列，我们只要找到前面那些结尾比 3 小的子序列，然后把 3 接到这些子序列末尾，就可以形成一个新的递增子序列，而且这个新的子序列长度加一。

`nums[5]` 前面有哪些元素小于 `nums[5]`？这个好算，用 `for` 循环比较一波就能把这些元素找出来。

以这些元素为结尾的最长递增子序列的长度是多少？回顾一下我们对 `dp` 数组的定义，它记录的正是以每个元素为末尾的最长递增子序列的长度。

以我们举的例子来说，`nums[0]` 和 `nums[4]` 都是小于 `nums[5]` 的，然后对比 `dp[0]` 和 `dp[4]` 的值，我们让 `nums[5]` 和更长的递增子序列结合，得出 `dp[5] = 3`：

0 1 2 3 4 5

nums

1	4	3	4	2	3
---	---	---	---	---	---

dp

1	2	2	3	2	3
---	---	---	---	---	---

 $\max(1+1, 2+1)$ 

公众号：labuladong

```
for (int j = 0; j < i; j++) {
    if (nums[i] > nums[j]) {
        dp[i] = Math.max(dp[i], dp[j] + 1);
    }
}
```

```

    }
}

```

当  $i = 5$  时，这段代码的逻辑就可以算出  $dp[5]$ 。其实到这里，这道算法题我们就基本做完了。

读者也许会问，我们刚才只是算了  $dp[5]$  呀， $dp[4], dp[3]$  这些怎么算呢？类似数学归纳法，你已经可以算出  $dp[5]$  了，其他的就都可以算出来：

```

for (int i = 0; i < nums.length; i++) {
    for (int j = 0; j < i; j++) {
        // 寻找 nums[0..j-1] 中比 nums[i] 小的元素
        if (nums[i] > nums[j]) {
            // 把 nums[i] 接在后面，即可形成长度为 dp[j] + 1,
            // 且以 nums[i] 为结尾的递增子序列
            dp[i] = Math.max(dp[i], dp[j] + 1);
        }
    }
}

```

结合我们刚才说的 base case，下面我们看一下完整代码：

```

class Solution {
    public int lengthOfLIS(int[] nums) {
        // 定义：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度
        int[] dp = new int[nums.length];
        // base case: dp 数组全都初始化为 1
        Arrays.fill(dp, 1);
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
        }

        int res = 0;
        for (int i = 0; i < dp.length; i++) {
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}

```

## ▶ ★ 代码可视化动画★

至此，这道题就解决了，时间复杂度  $O(N^2)$ 。总结一下如何找到动态规划的状态转移关系：

- 1、明确  $dp$  数组的定义。这一步对于任何动态规划问题都很重要，如果不得当或者不够清晰，会阻碍之后的步骤。
- 2、根据  $dp$  数组的定义，运用数学归纳法的思想，假设  $dp[0 \dots i-1]$  都已知，想办法求出  $dp[i]$ ，一旦这一步完成，整个题目基本就解决了。

但如果无法完成这一步，很可能就是 `dp` 数组的定义不够恰当，需要重新定义 `dp` 数组的含义；或者可能是 `dp` 数组存储的信息还不够，不足以推出下一步的答案，需要把 `dp` 数组扩大成二维数组甚至三维数组。

目前的解法是标准的动态规划，但对最长递增子序列问题来说，这个解法不是最优的，可能无法通过所有测试用例了，下面讲讲更高效的解法。

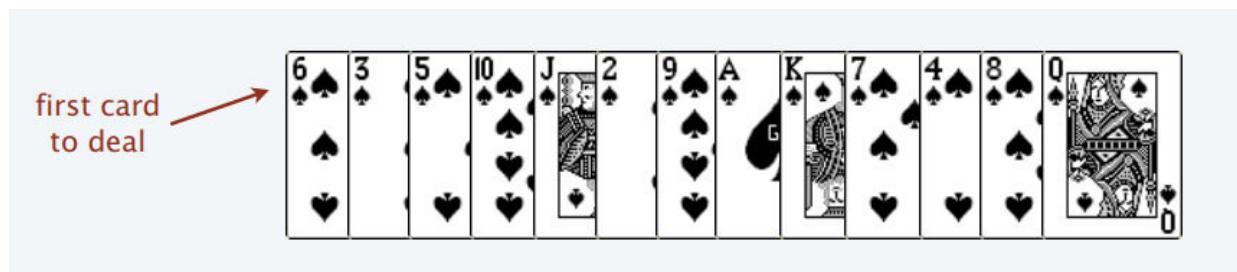
## 二、二分查找解法

这个解法的时间复杂度为  $O(N \log N)$ ，但是说实话，正常人基本想不到这种解法（也许玩过某些纸牌游戏的人可以想出来）。所以大家了解一下就好，正常情况下能够给出动态规划解法就已经很不错了。

根据题目的意思，我都很难想象这个问题竟然能和二分查找扯上关系。其实最长递增子序列和一种叫做 patience game 的纸牌游戏有关，甚至有一种排序方法就叫做 patience sorting（耐心排序）。

为了简单起见，后文跳过所有数学证明，通过一个简化的例子来理解一下算法思路。

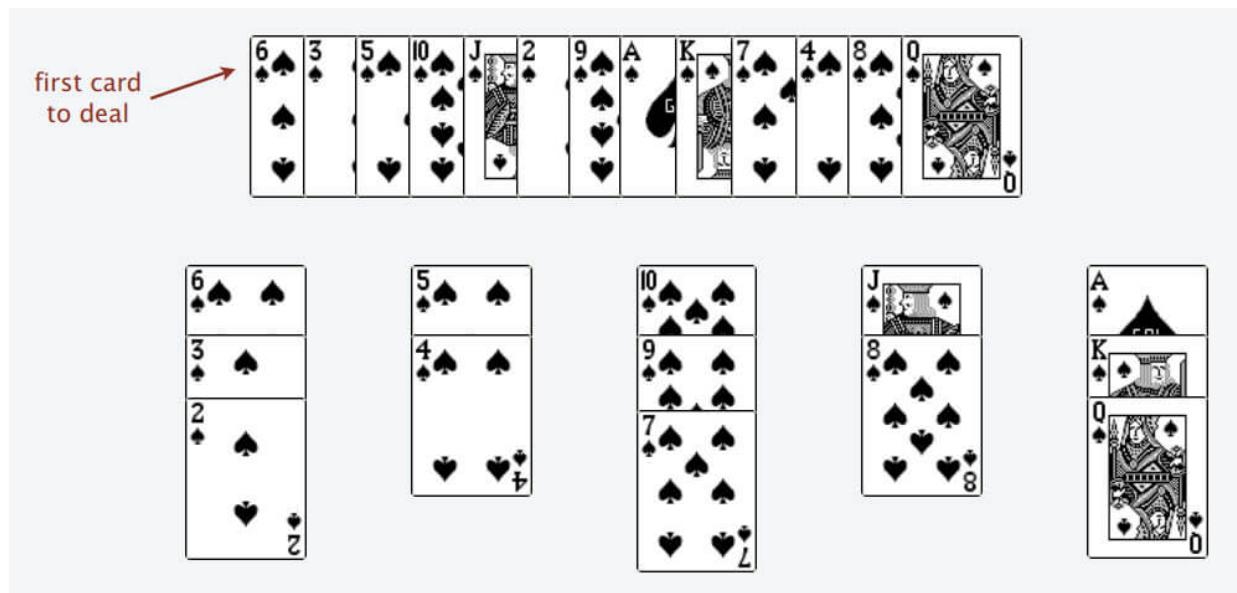
首先，给你一排扑克牌，我们像遍历数组那样从左到右一张一张处理这些扑克牌，最终要把这些牌分成若干堆。



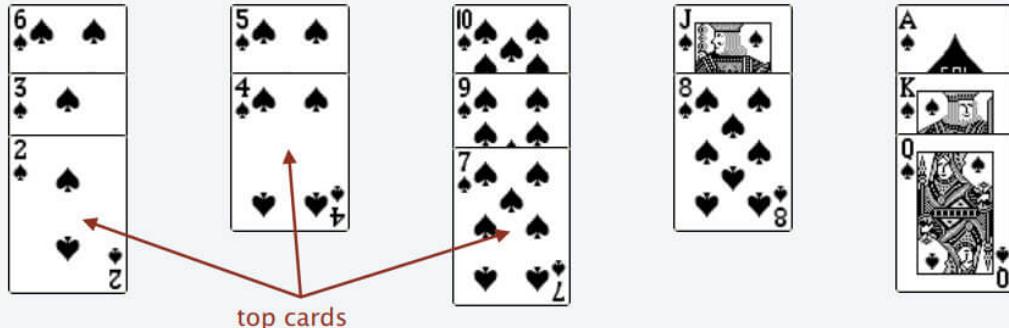
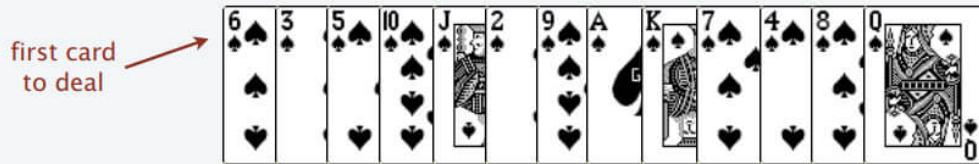
处理这些扑克牌要遵循以下规则：

只能把点数小的牌压到点数比它大的牌上；如果当前牌点数较大没有可以放置的堆，则新建一个堆，把这张牌放进去；如果当前牌有多个堆可供选择，则选择最左边的那一堆放置。

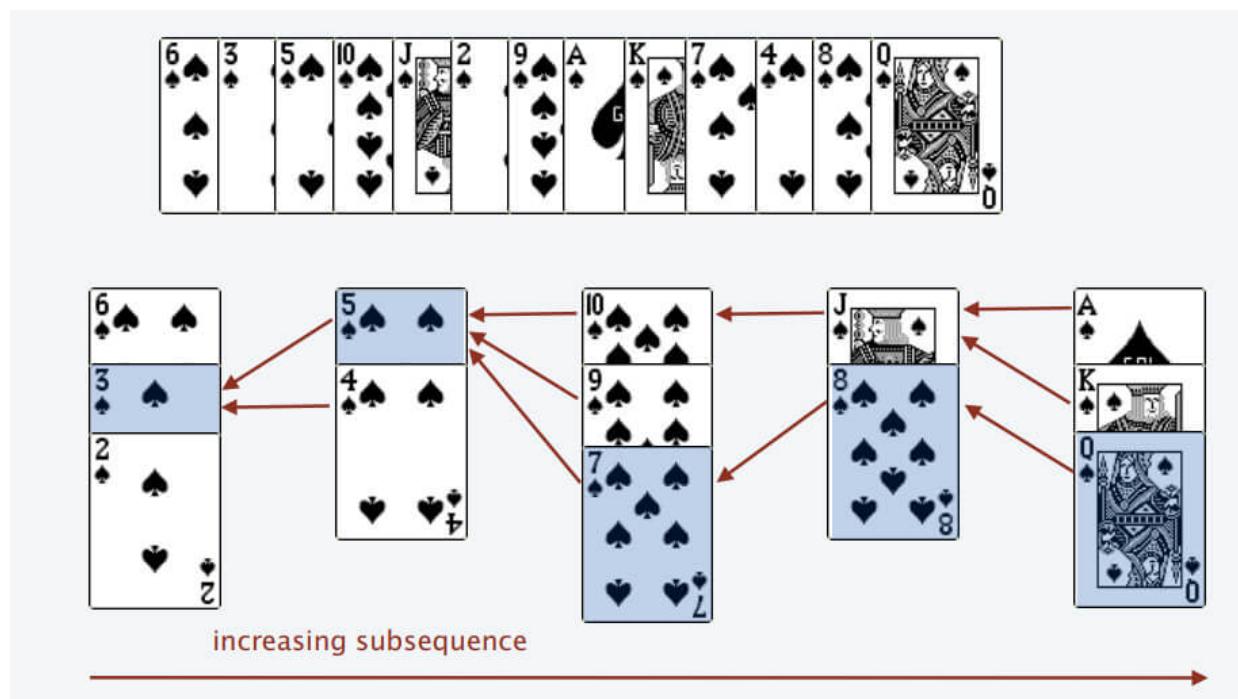
比如说上述的扑克牌最终会被分成这样 5 堆（我们认为纸牌 A 的牌面是最大的，纸牌 2 的牌面是最小的）。



为什么遇到多个可选择堆的时候要放到最左边的堆上呢？因为这样可以保证牌堆顶的牌有序  $(2, 4, 7, 8, Q)$ ，证明略。



按照上述规则执行，可以算出最长递增子序列，牌的堆数就是最长递增子序列的长度，证明略。



我们只要把处理扑克牌的过程编程写出来即可。每次处理一张扑克牌不是要找一个合适的牌堆顶来放吗，牌堆顶的牌不是有序吗，这就能用到二分查找了：用二分查找来搜索当前牌应放置的位置。

前文 [二分查找算法详解](#) 详细介绍了二分查找的细节及变体，这里就完美应用上了，如果没读过强烈建议阅读。

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] top = new int[nums.length];
        // 牌堆数初始化为 0
        int piles = 0;
        for (int i = 0; i < nums.length; i++) {
            // 要处理的扑克牌
            int poker = nums[i];
```

```
// ***** 搜索左侧边界的二分查找 *****
int left = 0, right = piles;
while (left < right) {
    int mid = (left + right) / 2;
    if (top[mid] > poker) {
        right = mid;
    } else if (top[mid] < poker) {
        left = mid + 1;
    } else {
        right = mid;
    }
}
// *****

// 没找到合适的牌堆, 新建一堆
if (left == piles) piles++;
// 把这张牌放到牌堆顶
top[left] = poker;
}
// 牌堆数就是 LIS 长度
return piles;
}
```

### ▶ 代码可视化动画

至此，二分查找的解法也讲解完毕。

这个解法确实很难想到。首先涉及数学证明，谁能想到按照这些规则执行，就能得到最长递增子序列呢？其次还有二分查找的运用，要是对二分查找的细节不清楚，给了思路也很难写对。

所以，这个方法作为思维拓展好了。但动态规划的设计方法应该完全理解：假设之前的答案已知，利用数学归纳的思想正确进行状态的推演转移，最终得到答案。

## 三、拓展到二维

我们看一个经常出现在生活中的有趣问题，力扣第 354 题「俄罗斯套娃信封问题」，先看下题目：

### ▼ 354. 俄罗斯套娃信封问题 [Leetcode](#) | 力扣

给你一个二维整数数组 `envelopes`，其中 `envelopes[i] = [wi, hi]`，表示第 `i` 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算 **最多能有多少个** 信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

**注意：**不允许旋转信封。

### 示例 1：

```
输入: envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出: 3
解释: 最多信封的个数为 3, 组合为: [2,3] => [5,4] => [6,7]。
```

示例 2：

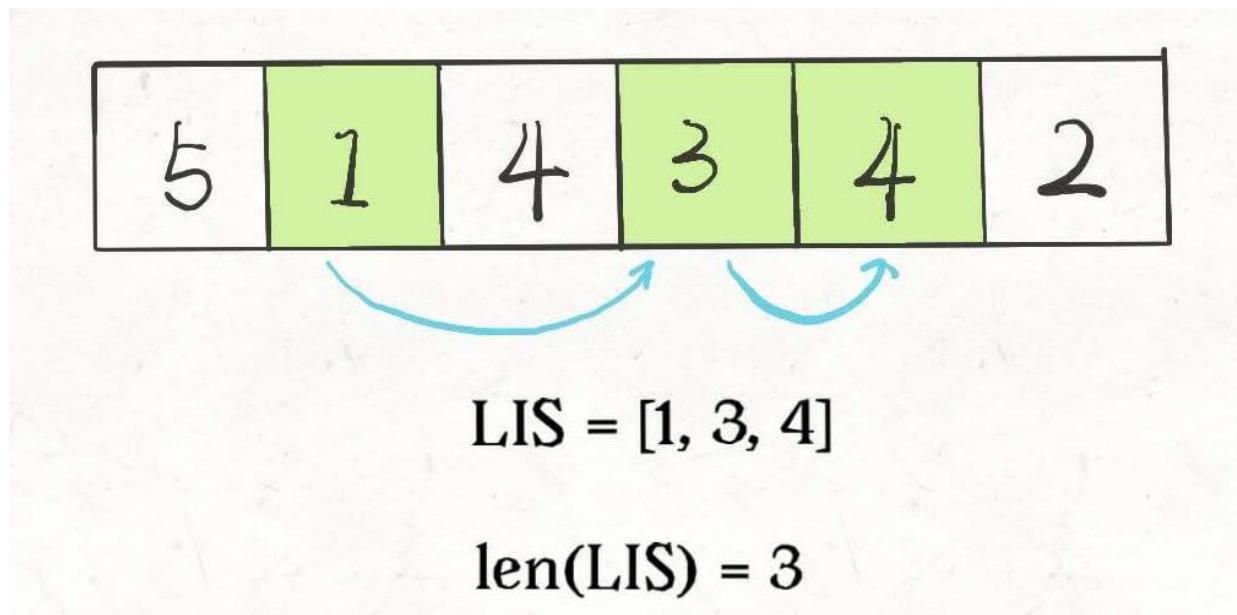
```
输入: envelopes = [[1,1],[1,1],[1,1]]  
输出: 1
```

提示：

- $1 \leq \text{envelopes.length} \leq 10^5$
- $\text{envelopes}[i].length == 2$
- $1 \leq w_i, h_i \leq 10^5$

这道题目其实是最长递增子序列的一个变种，因为每次合法的嵌套是大的套小的，相当于在二维平面中找一个最长递增的子序列，其长度就是最多能嵌套的信封个数。

前面说的标准 LIS 算法只能在一维数组中寻找最长子序列，而我们的信封是由  $(w, h)$  这样的二维数对形式表示的，如何把 LIS 算法运用过来呢？



读者也许会想，通过  $w \times h$  计算面积，然后对面积进行标准的 LIS 算法。但是稍加思考就会发现这样不行，比如  $1 \times 10$  大于  $3 \times 3$ ，但是显然这样的两个信封是无法互相嵌套的。

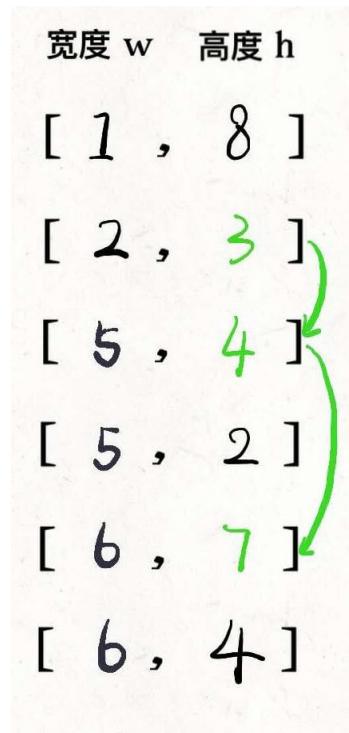
这道题的解法比较巧妙：

先对宽度  $w$  进行升序排序，如果遇到  $w$  相同的情况，则按照高度  $h$  降序排序；之后把所有的  $h$  作为一个数组，在这个数组上计算 LIS 的长度就是答案。

画个图理解一下，先对这些数对进行排序：



然后在  $h$  上寻找最长递增子序列，这个子序列就是最优的嵌套方案：



那么为什么这样就可以找到可以互相嵌套的信封序列呢？稍微思考一下就明白了：

首先，对宽度  $w$  从小到大排序，确保了  $w$  这个维度可以互相嵌套，所以我们只需要专注高度  $h$  这个维度能够互相嵌套即可。

其次，两个  $w$  相同的信封不能相互包含，所以对于宽度  $w$  相同的信封，对高度  $h$  进行降序排序，保证二维 LIS 中不存在多个  $w$  相同的信封（因为题目说了长宽相同也无法嵌套）。

下面看解法代码：

```
class Solution {
    // envelopes = [[w, h], [w, h]...]
```

```
public int maxEnvelopes(int[][] envelopes) {
    int n = envelopes.length;
    // 按宽度升序排列，如果宽度一样，则按高度降序排列
    Arrays.sort(envelopes, (int[] a, int[] b) -> {
        return a[0] == b[0] ?
            b[1] - a[1] : a[0] - b[0];
    });
    // 对高度数组寻找 LIS
    int[] height = new int[n];
    for (int i = 0; i < n; i++)
        height[i] = envelopes[i][1];

    return lengthOfLIS(height);
}

int lengthOfLIS(int[] nums) {
    // 见前文
}
}
```

### ▶ 代码可视化动画

为了复用之前的函数，我将代码分为了两个函数，你也可以合并代码，节省下 `height` 数组的空间。

由于增加了测试用例，这里必须使用二分搜索版的 `lengthOfLIS` 函数才能通过所有测试用例。这样的话算法的时间复杂度为  $O(N \log N)$ ，因为排序和计算 LIS 各需要  $O(N \log N)$  的时间，加到一起还是  $O(N \log N)$ ；空间复杂度为  $O(N)$ ，因为计算 LIS 的函数中需要一个 `top` 数组。

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1425. Constrained Subsequence Sum</a>	<a href="#">1425. 带限制的子序列和</a>	
<a href="#">256. Paint House</a>	<a href="#">256. 粉刷房子</a>	
<a href="#">368. Largest Divisible Subset</a>	<a href="#">368. 最大整除子集</a>	
-	<a href="#">剑指 Offer II 091. 粉刷房子</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# base case 和备忘录的初始值怎么定？



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">931. Minimum Falling Path Sum</a>	931. 下降路径最小和	困难

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

很多读者对动态规划问题的 base case、备忘录初始值等问题存在疑问，本文就专门讲一讲这类问题，顺便聊一聊怎么通过题目的蛛丝马迹揣测出题人的小心思，辅助我们解题。

看下力扣第 931 题「下降路径最小和」，输入为一个  $n * n$  的二维数组  $\text{matrix}$ ，请你计算从第一行落到最后一行，经过的路径和最小为多少：

## ▼ 931. 下降路径最小和 [Leetcode | 力扣](#)

给你一个  $n \times n$  的 **方形** 整数数组  $\text{matrix}$ ，请你找出并返回通过  $\text{matrix}$  的下降路径的 **最小和**。

**下降路径** 可以从第一行中的任何元素开始，并从每一行中选择一个元素。在下一行选择的元素和当前行所选元素最多相隔一列（即位于正下方或者沿对角线向左或者向右的第一个元素）。具体来说，位置  $(\text{row}, \text{col})$  的下一个元素应当是  $(\text{row} + 1, \text{col} - 1)$ 、 $(\text{row} + 1, \text{col})$  或者  $(\text{row} + 1, \text{col} + 1)$ 。

**示例 1：**

2	1	3
6	5	4
7	8	9

2	1	3
6	5	4
7	8	9

2	1	3
6	5	4
7	8	9

输入: matrix = [[2,1,3],[6,5,4],[7,8,9]]

输出: 13

解释: 如图所示, 为和最小的两条下降路径

示例 2:

-19	57
-40	-5

-19	57
-40	-5

输入: matrix = [[-19,57],[-40,-5]]

输出: -59

解释：如图所示，为和最小的下降路径

提示：

- $n == \text{matrix.length} == \text{matrix[i].length}$
- $1 \leq n \leq 100$
- $-100 \leq \text{matrix}[i][j] \leq 100$

函数签名如下：

```
int minFallingPathSum(int[][] matrix);
```

今天这道题不算是困难的题目，所以我借这道题来讲解 **base case** 的返回值、备忘录的初始值、索引越界情况的返回值如何确定。

不过还是要根据 [动态规划的标准套路](#) 讲一下这道题的解题思路。

## 解题思路

首先，我们可以定义一个 **dp** 数组：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 动态规划穷举的两种视角



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">115. Distinct Subsequences</a>	<a href="#">115. 不同的子序列</a>	●

-----

阅读本文前，你需要先学习：

- 二叉树系列算法（纲领篇）
- 动态规划核心框架
- 球盒模型：回溯算法穷举的两种视角

本文我会带大家复习一下动态规划相关问题的一系列解题套路，然后着重讨论一下动态规划穷举时不同视角的问题。

## 动态规划解题组合拳

首先，前文 [我的刷题心得](#) 讲了，我们刷的算法问题的本质是「穷举」，动态规划问题也不例外，你必须想办法穷举所有可能的解，然后从中筛选出符合题目要求的解。

另外，动态规划问题穷举的过程中会出现重叠子问题导致的冗余计算，所以前文 [动态规划核心套路框架](#) 中告诉你如何一步一步把暴力穷举解法优化成效率更高的动态规划解法。

然而，想要写出暴力解需要依据状态转移方程，状态转移方程是动态规划的解题核心，可不是那么容易想出来的。不过，后文 [动态规划设计：数学归纳法](#) 告诉你，思考状态转移方程的一个基本方法是数学归纳法，即明确 `dp` 函数或数组的定义，然后使用这个定义，从已知的「状态」中推导出未知的「状态」。

接下来就是本文要着重探讨的问题了：就算 `dp` 函数/数组的定义相同，如果你使用不同的「视角」进行穷举，效率也不见得是相同的。

关于穷举「视角」的问题，前文 [球盒模型：回溯算法穷举的两种视角](#) 讲了回溯算法中不同的穷举视角导致的不同解法，其实这种视角的切换在动态规划类型问题中依然存在。前文对排列的举例非常有助于你理解穷举视角的问题，这里再简单提一下。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 动态规划和回溯算法的思维转换



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">140. Word Break II</a>	<a href="#">140. 单词拆分 II</a>	
<a href="#">139. Word Break</a>	<a href="#">139. 单词拆分</a>	

-----

阅读本文前，你需要先学习：

- [二叉树系列算法（纲领篇）](#)
- [动态规划核心框架](#)

之前 [手把手带你刷二叉树（纲领篇）](#) 把递归穷举划分为「遍历」和「分解问题」两种思路，其中「遍历」的思路扩展延伸一下就是 [回溯算法](#)，「分解问题」的思路可以扩展成 [动态规划算法](#)。

这种思维转换不止局限于二叉树相关的算法，本文就跳出二叉树类型问题，来看看实际算法题中如何把问题抽象成树形结构，见招拆招逐步优化，从而进行「遍历」和「分解问题」的思维转换，从回溯算法顺滑地切换到动态规划算法。

先说句题外话，前文 [动态规划核心框架详解](#) 说，[标准的动态规划问题一定是求最值的](#)，因为动态规划类型问题有一个性质叫做「最优子结构」，即从子问题的最优解推导出原问题的最优解。

但在我们平常的语境中，就算不是求最值的题目，只要看见使用备忘录消除重叠子问题，我们一般都称它为动态规划算法。严格来讲这是不符合动态规划问题的定义的，说这种解法叫做「带备忘录的 DFS 算法」可能更准确些。不过咱也不用太纠结这种名词层面的细节，既然大家叫的顺口，就叫它动态规划也无妨。

本文讲解的两道题目也不是求最值的，但依然会把他们的解法称为动态规划解法，这里提前跟大家说下这个变通，免得严谨的读者疑惑。其他不多说了，直接看题目吧。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 对动态规划进行降维打击



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

空间压缩并不难，主要用来优化某些动态规划问题的空间复杂度。但是一般的笔试中对空间的要求并不高，即便不使用这个优化技巧也能通过，所以我个人认为状态压缩并不是必须掌握的技巧，有兴趣的读者可以仔细学习理解一下。

我们号之前写过十几篇动态规划文章，可以说动态规划技巧对于算法效率的提升非常可观，一般来说都能把指数级和阶乘级时间复杂度的算法优化成  $O(N^2)$ ，堪称算法界的二向箔，把各路魑魅魍魎统统打成二次元。

但是，动态规划求解的过程中也是可以进行阶段性优化的，如果你认真观察某些动态规划问题的状态转移方程，就能够把它们解法的空间复杂度进一步降低，由  $O(N^2)$  降低到  $O(N)$ 。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 最优子结构原理和 dp 数组遍历方向



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

tip：本文有视频版：[动态规划详解进阶](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

本文是旧文 [动态规划答疑篇](#) 的修订版，根据我的不断学习总结以及读者的评论反馈，我给扩展了更多内容，力求使本文成为继 [动态规划核心套路框架](#) 之后的一篇全面答疑文章。以下是正文。

这篇文章就给你讲明白以下几个问题：

- 1、到底什么才叫「最优子结构」，和动态规划什么关系。
- 2、如何判断一个问题是否是动态规划问题，即如何看出是否存在重叠子问题。
- 3、为什么经常看到将 `dp` 数组的大小设置为 `n + 1` 而不是 `n`。
- 4、为什么动态规划遍历 `dp` 数组的方式五花八门，有的正着遍历，有的倒着遍历，有的斜着遍历。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 经典动态规划：编辑距离



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">72. Edit Distance</a>	<a href="#">72. 编辑距离</a>	困难

阅读本文前，你需要先学习：

- 二叉树系列算法（纲领篇）
- 动态规划核心框架

tip：本文有视频版：[编辑距离详解动态规划](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

前几天看了一份鹅厂的面试题，算法部分大半是动态规划，最后一题就是写一个计算编辑距离的函数，今天就专门写一篇文章来探讨一下这个问题。

力扣第 72 题「编辑距离」就是这个问题，先看下题目：

## ▼ 72. 编辑距离 Leetcode | 力扣

给你两个单词 `word1` 和 `word2`，请返回将 `word1` 转换成 `word2` 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删一个字符
- 替换一个字符

示例 1：

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 %%%h%%% 替换为 %%%r%%%)
rorse -> rose (删除 %%r%%)
rose -> ros (删除 %%e%%)
```

示例 2：

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 %%%t%%%)
inention -> enention (将 %%%i%%% 替换为 %%%e%%%)
enention -> exention (将 %%%n%%% 替换为 %%%x%%%)
exention -> execution (将 %%%n%%% 替换为 %%%c%%%)
execution -> execution (插入 %%%u%%%)
```

#### 提示:

- $0 \leq \text{word1.length}, \text{word2.length} \leq 500$
- $\text{word1}$  和  $\text{word2}$  由小写英文字母组成

```
// 函数签名如下
int minDistance(String s1, String s2)
```

对于没有接触过动态规划问题的读者来说，这道题还是有一定难度的，是不是感觉完全无从下手？

但这个问题本身还是比较实用的，我曾经就在日常生活中用到过这个算法。之前有一篇公众号文章由于疏忽，写错位了一段内容，我决定修改这部分内容让逻辑通顺。但是公众号文章最多只能修改 20 个字，且只支持增、删、替换操作（跟编辑距离问题一模一样），于是我就用算法求出了一个最优方案，只用了 16 步就完成了修改。

再比如高大上一点的应用，DNA 序列是由 A,G,C,T 组成的序列，可以类比成字符串。编辑距离可以衡量两个 DNA 序列的相似度，编辑距离越小，说明这两段 DNA 越相似，说不定这俩 DNA 的主人是远古近亲啥的。

下面言归正传，详细讲解一下编辑距离该怎么算，相信本文会让你有收获。

## 一、思路

编辑距离问题就是给我们两个字符串  $s_1$  和  $s_2$ ，只能用三种操作，让我们把  $s_1$  变成  $s_2$ ，求最少的操作数。需要明确的是，不管是把  $s_1$  变成  $s_2$  还是反过来，结果都是一样的，所以后文就以  $s_1$  变成  $s_2$  举例。

解决两个字符串的动态规划问题，一般都是用两个指针  $i, j$  分别指向两个字符串的头部或尾部，然后尝试写状态转移方程。

比方说让  $i, j$  分别指向两个字符串的尾部，把  $dp[i], dp[j]$  定义为  $s_1[0..i], s_2[0..j]$  子串的编辑距离，那么  $i, j$  一步步往前移动的过程，就是问题规模（子串长度）逐步减小的过程。

当然，你想让让  $i, j$  分别指向字符串头部，然后一步步往后移动也可以，本质上并无区别，只要改一下  $dp$  函数/数组的定义即可。

设两个字符串分别为 "rad" 和 "apple"，让  $i, j$  两个指针分别指向  $s_1, s_2$  的尾部，为了把  $s_1$  变成  $s_2$ ，算法会这样进行：

## 把 s1 变成 s2

s1      r      a      d

s2      a      p      p      l      e

公众号: labuladong

至少需要 5 步

删      替      插      插      插  
s1      ~~r~~      a      p      p      l      e

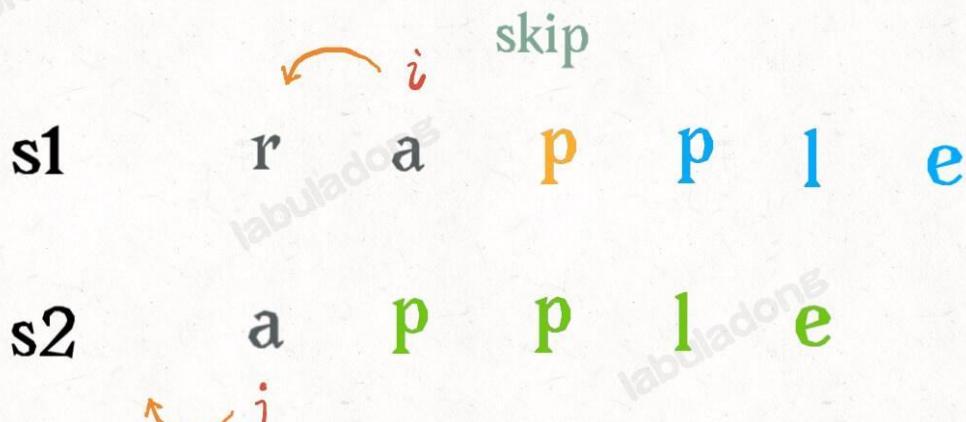
s2      a      p      p      l      e

公众  公众号: labuladong

请记住这个 GIF 过程，这样就能算出编辑距离。关键在于如何做出正确的操作，稍后会讲。

根据上面的 GIF，可以发现操作不只有三个，其实还有第四个操作，就是什么都不要做（skip）。比如这个情况：

$s1[i] == s2[j]$

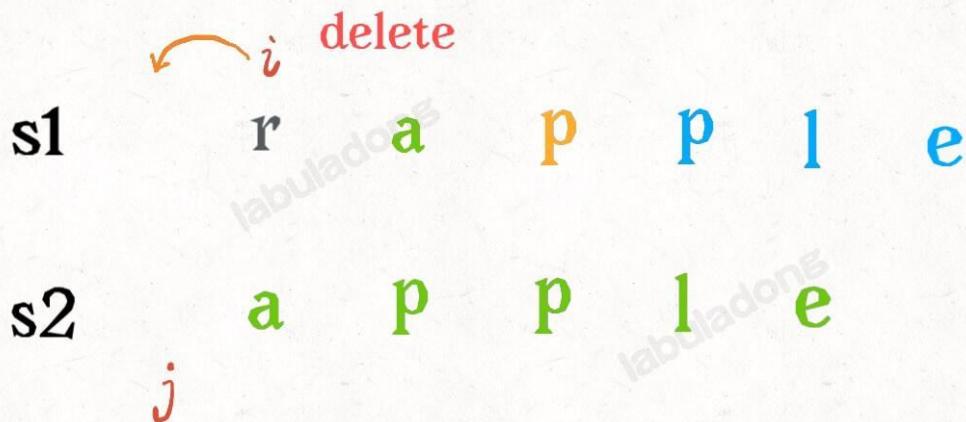


© labuladong

因为这两个字符本来就相同，为了使编辑距离最小，显然不应该对它们有任何操作，直接往前移动  $i, j$  即可。

还有一个很容易处理的情况，就是  $j$  走完  $s2$  时，如果  $i$  还没走完  $s1$ ，那么只能用删除操作把  $s1$  缩短为  $s2$ 。比如这个情况：

s2 走完了



© labuladong

类似的，如果  $i$  走完  $s1$  时  $j$  还没走完了  $s2$ ，那就只能用插入操作把  $s2$  剩下的字符全部插入  $s1$ 。等会会看到，这两种情况就是算法的 **base case**。

下面详解一下如何将思路转换成代码。

## 二、代码详解

先梳理一下之前的思路：

base case 是  $i$  走完  $s1$  或  $j$  走完  $s2$ ，可以直接返回另一个字符串剩下的长度。

对于每对儿字符  $s1[i]$  和  $s2[j]$ ，可以有四种操作：

```

if s1[i] == s2[j]:
    哪都别做 (skip)
    i, j 同时向前移动
else:
    三选一:
        插入 (insert)
        删除 (delete)
        替换 (replace)

```

有这个框架，问题就已经解决了。读者也许会问，这个「三选一」到底该怎么选择呢？很简单，全试一遍，哪个操作最后得到的编辑距离最小，就选谁。这里需要递归技巧，先看下暴力解法代码：

```

class Solution {
    public int minDistance(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // i, j 初始化指向最后一个索引
        return dp(s1, m - 1, s2, n - 1);
    }

    // 定义：返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
    int dp(String s1, int i, String s2, int j) {
        // base case
        if (i == -1) return j + 1;
        if (j == -1) return i + 1;

        if (s1.charAt(i) == s2.charAt(j)) {
            // 哪都不做
            return dp(s1, i - 1, s2, j - 1);
        }
        return min(
            // 插入
            dp(s1, i, s2, j - 1) + 1,
            // 删除
            dp(s1, i - 1, s2, j) + 1,
            // 替换
            dp(s1, i - 1, s2, j - 1) + 1
        );
    }

    int min(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
    }
}

```

下面来详细解释一下这段递归代码，base case 应该不用解释了，主要解释一下递归部分。

都说递归代码的可解释性很好，这是有道理的，只要理解函数的定义，就能很清楚地理解算法的逻辑。我们这里 `dp` 函数的定义是这样的：

```

// 定义：返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
int dp(String s1, int i, String s2, int j)

```

记住这个定义之后，先来看这段代码：

```
if s1[i] == s2[j]:  
    # 哪都不做  
    return dp(s1, i - 1, s2, j - 1)  
# 解释：  
# 本来就相等，不需要任何操作  
# s1[0..i] 和 s2[0..j] 的最小编辑距离等于  
# s1[0..i-1] 和 s2[0..j-1] 的最小编辑距离  
# 也就是说 dp(i, j) 等于 dp(i-1, j-1)
```

如果  $s1[i] \neq s2[j]$ ，就要对三个操作递归了，稍微需要点思考：

```
# 插入  
dp(s1, i, s2, j - 1) + 1,  
# 解释：  
# 我直接在 s1[i] 插入一个和 s2[j] 一样的字符  
# 那么 s2[j] 就被匹配了，前移 j，继续跟 i 对比  
# 别忘了操作数加一
```

$s1[i] \neq s2[j]$

*i* insert "p"

s1 r a d l e

s2 a p p l e

↑ j

公众号：labuladong

```
# 删除  
dp(s1, i - 1, s2, j) + 1,  
# 解释：  
# 我直接把 s[i] 这个字符删掉  
# 前移 i，继续跟 j 对比  
# 操作数加一
```

## s2 走完了

公众号: labuladong

<b>s1</b>	r	a	p	p	l	e
<b>s2</b>	a	p	p	l	e	
	j					

```
# 替换
dp(s1, i - 1, s2, j - 1) + 1
# 解释:
# 我直接把 s1[i] 替换成 s2[j], 这样它俩就匹配了
# 同时前移 i, j 继续对比
# 操作数加一
```

公众号: labuladong

<b>s1[i] != s2[j]</b>						
<b>s1</b>	r	a	d	p	l	e
<b>s2</b>	a	p	p	l	e	
	j					

现在，你应该完全理解这段短小精悍的代码了。还有点小问题就是，这个解法是暴力解法，存在重叠子问题，需要用动态规划技巧来优化。

怎么能一眼看出存在重叠子问题呢？我在[动态规划答疑篇](#)有讲过，这里再简单提一下，需要抽象出本文算法的递归框架：

```

int dp(i, j) {
    dp(i - 1, j - 1); // #1
    dp(i, j - 1);    // #2
    dp(i - 1, j);    // #3
}

```

对于子问题  $dp(i-1, j-1)$ , 如何通过原问题  $dp(i, j)$  得到呢? 有不止一条路径, 比如  $dp(i, j) \rightarrow \#1$  和  $dp(i, j) \rightarrow \#2 \rightarrow \#3$ 。一旦发现一条重复路径, 就说明存在巨量重复路径, 也就是重叠子问题。

### 三、动态规划优化

对于重叠子问题呢, 前文 [动态规划详解](#) 详细介绍过, 优化方法无非是给递归解法加备忘录, 或者把动态规划过程用 DP table 迭代实现, 下面逐个来讲。

#### 备忘录解法

既然暴力递归解法都写出来了, 备忘录是很容易加的, 原来的代码稍加修改即可:

```

class Solution {
    // 备忘录
    int[][] memo;

    public int minDistance(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // 备忘录初始化为特殊值, 代表还未计算
        memo = new int[m][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return dp(s1, m - 1, s2, n - 1);
    }

    int dp(String s1, int i, String s2, int j) {
        if (i == -1) return j + 1;
        if (j == -1) return i + 1;
        // 查备忘录, 避免重叠子问题
        if (memo[i][j] != -1) {
            return memo[i][j];
        }
        // 状态转移, 结果存入备忘录
        if (s1.charAt(i) == s2.charAt(j)) {
            memo[i][j] = dp(s1, i - 1, s2, j - 1);
        } else {
            memo[i][j] = min(
                dp(s1, i, s2, j - 1) + 1,
                dp(s1, i - 1, s2, j) + 1,
                dp(s1, i - 1, s2, j - 1) + 1
            );
        }
        return memo[i][j];
    }

    int min(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
    }
}

```

```

    }
}

```

## DP table 解法

主要说下 DP table 的解法，我们需要定义一个 `dp` 数组，然后在这个数组上执行状态转移方程。

首先明确 `dp` 数组的含义，由于本题有两个状态（索引 `i` 和 `j`），所以 `dp` 数组是一个二维数组，大概长这样：

		<i>s2</i>	'''	<i>a</i>	<i>P</i>	<i>P</i>	<i>l</i>	<i>e</i>
		<i>s1</i>	0	1	2	3	4	5
		''	0	1	2	3	4	5
		<i>r</i>	1	1	2	3	4	5
		<i>a</i>	2	1	2	3	4	5
		<i>d</i>	3	2	2	3	4	5

© labuladong

状态转移和递归解法相同，`dp[...][0]` 和 `dp[0][...]` 对应 base case，`dp[i][j]` 的含义和之前 `dp` 函数的定义类似：

```

int dp(String s1, int i, String s2, int j)
// 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离

dp[i-1][j-1]
// 存储 s1[0..i] 和 s2[0..j] 的最小编辑距离

```

`dp` 函数的 base case 是 `i`, `j` 等于 -1，而数组索引至少是 0，所以 `dp` 数组会偏移一位。

既然 `dp` 数组和递归 `dp` 函数含义一样，也就可以直接套用之前的思路写代码，唯一不同的是，递归解法是自顶向下求解（从原问题开始，逐步分解到 base case），DP table 是自底向上求解（从 base case 开始，向原问题推演）：

```

class Solution {
    public int minDistance(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // 定义：s1[0..i] 和 s2[0..j] 的最小编辑距离是 dp[i+1][j+1]
        int[][] dp = new int[m + 1][n + 1];
        // base case
        for (int i = 0; i <= m; i++)
            dp[i][0] = i;
        for (int j = 0; j <= n; j++)
            dp[0][j] = j;
        // 自底向上求解
    }
}

```

```

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (s1.charAt(i-1) == s2.charAt(j-1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = min(
                        dp[i - 1][j] + 1,
                        dp[i][j - 1] + 1,
                        dp[i - 1][j - 1] + 1
                    );
                }
            }
        }
        // 储存着整个 s1 和 s2 的最小编辑距离
        return dp[m][n];
    }

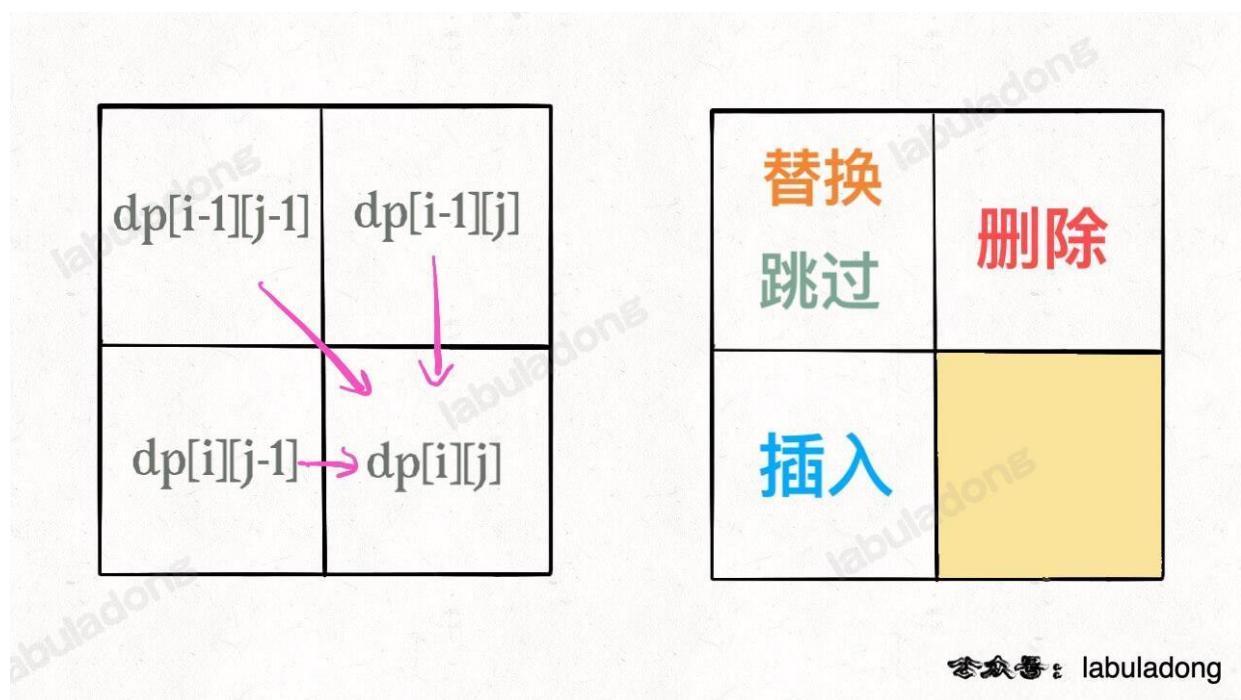
    int min(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
    }
}

```

▶ 😊 代码可视化动画😊

## 四、扩展延伸

一般来说，处理两个字符串的动态规划问题，都是按本文的思路处理，建立 DP table。为什么呢，因为易于找出状态转移的关系，比如编辑距离的 DP table：



还有一个细节，既然每个  $dp[i][j]$  只和它附近的三个状态有关，空间复杂度是可以压缩成  $O(\min(M, N))$  的（ $M, N$  是两个字符串的长度）。不难，但是可解释性大大降低，读者可以自己尝试优化一下。

你可能还会问，这里只求出了最小的编辑距离，那具体的操作是什么？你之前举的修改公众号文章的例子，只有一个最小编辑距离肯定不够，还得知道具体怎么修改才行。

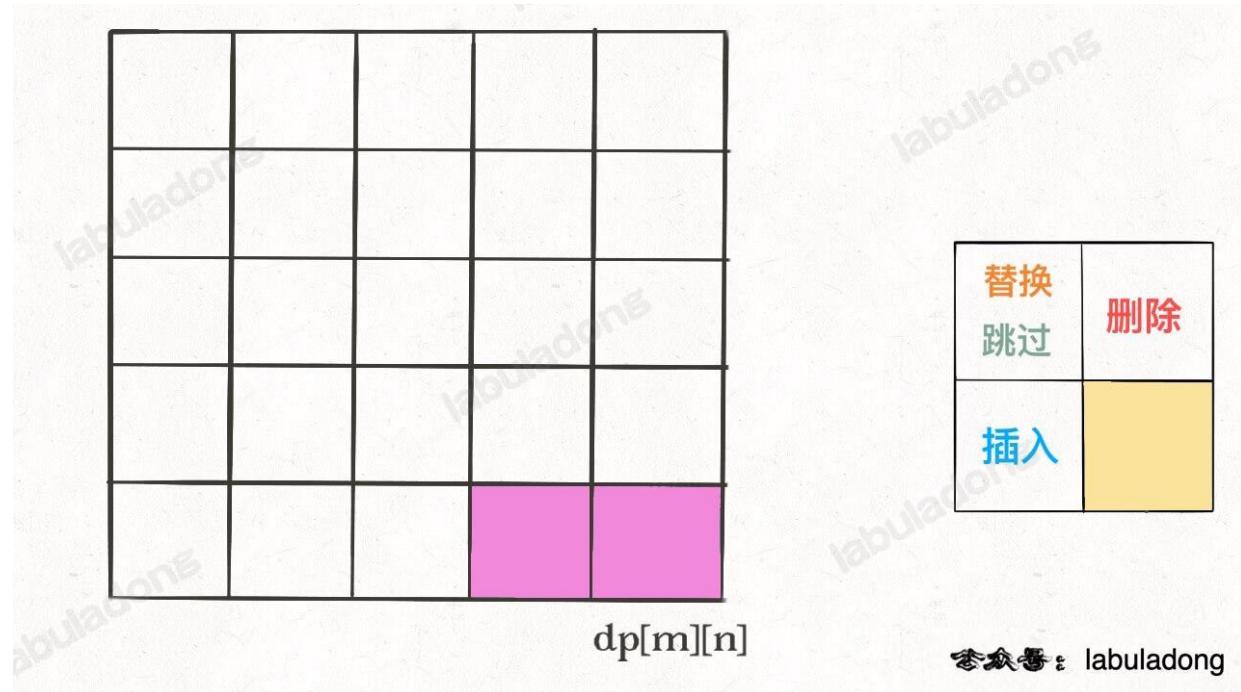
这个其实很简单，代码稍加修改，给 dp 数组增加额外的信息即可：

```
// int[][] dp;
Node[][] dp;

class Node {
    int val;
    int choice;
    // 0 代表啥都不做
    // 1 代表插入
    // 2 代表删除
    // 3 代表替换
}
```

`val` 属性就是之前的 dp 数组的数值，`choice` 属性代表操作。在做最优选择时，顺便把操作记录下来，然后就从结果反推具体操作。

我们的最终结果不是 `dp[m][n]` 吗，这里的 `val` 存着最小编辑距离，`choice` 存着最后一个操作，比如说是插入操作，那么就可以左移一格：



重复此过程，可以一步步回到起点 `dp[0][0]`，形成一条路径，按这条路径上的操作进行编辑，就是最佳方案。

删				
插	替			
		删		
		跳		
			插	插

$dp[m][n]$

替换	删除
插入	

✿✿✿✿✿ labuladong

应大家的要求，我把这个思路也写出来，你可以自己运行试一下：

```

int minDistance(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    Node[][] dp = new Node[m + 1][n + 1];
    // base case
    for (int i = 0; i <= m; i++) {
        // s1 转化成 s2 只需要删除一个字符
        dp[i][0] = new Node(i, 2);
    }
    for (int j = 1; j <= n; j++) {
        // s1 转化成 s2 只需要插入一个字符
        dp[0][j] = new Node(j, 1);
    }
    // 状态转移方程
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1.charAt(i-1) == s2.charAt(j-1)){
                // 如果两个字符相同，则什么都不需要做
                Node node = dp[i - 1][j - 1];
                dp[i][j] = new Node(node.val, 0);
            } else {
                // 否则，记录代价最小的操作
                dp[i][j] = minNode(
                    dp[i - 1][j],
                    dp[i][j - 1],
                    dp[i-1][j-1]
                );
                // 并且将编辑距离加一
                dp[i][j].val++;
            }
        }
    }
    // 根据 dp table 反推具体操作过程并打印
    printResult(dp, s1, s2);
    return dp[m][n].val;
}

```

```
// 计算 delete, insert, replace 中代价最小的操作
Node minNode(Node a, Node b, Node c) {
    Node res = new Node(a.val, 2);

    if (res.val > b.val) {
        res.val = b.val;
        res.choice = 1;
    }
    if (res.val > c.val) {
        res.val = c.val;
        res.choice = 3;
    }
    return res;
}
```

最后，`printResult` 函数反推结果并把具体的操作打印出来：

```
void printResult(Node[][] dp, String s1, String s2) {
    int rows = dp.length;
    int cols = dp[0].length;
    int i = rows - 1, j = cols - 1;
    System.out.println("Change s1=" + s1 + " to s2=" + s2 + ":\\n");
    while (i != 0 && j != 0) {
        char c1 = s1.charAt(i - 1);
        char c2 = s2.charAt(j - 1);
        int choice = dp[i][j].choice;
        System.out.print("s1[" + (i - 1) + "]:");
        switch (choice) {
            case 0:
                // 跳过，则两个指针同时前进
                System.out.println("skip '" + c1 + "'");
                i--;
                j--;
                break;
            case 1:
                // 将 s2[j] 插入 s1[i], 则 s2 指针前进
                System.out.println("insert '" + c2 + "'");
                j--;
                break;
            case 2:
                // 将 s1[i] 删除，则 s1 指针前进
                System.out.println("delete '" + c1 + "'");
                i--;
                break;
            case 3:
                // 将 s1[i] 替换成 s2[j], 则两个指针同时前进
                System.out.println(
                    "replace '" + c1 + "' with '" + c2 + "'");
                i--;
                j--;
                break;
        }
    }
    // 如果 s1 还没有走完，则剩下的都是需要删除的
    while (i > 0) {
        System.out.print("s1[" + (i - 1) + "]:");
        System.out.println("delete '" + s1.charAt(i - 1) + "'");
        i--;
    }
}
```

```
    }
    // 如果 s2 还没有走完，则剩下的都是需要插入 s1 的
    while (j > 0) {
        System.out.print("s1[0]:");
        System.out.println("insert '" + s2.charAt(j - 1) + "'");
        j--;
    }
}
```

## ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">97. Interleaving String</a>	<a href="#">97. 交错字符串</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 动态规划设计：最大子数组



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">53. Maximum Subarray</a>	<a href="#">53. 最大子数组和</a>	简单

阅读本文前，你需要先学习：

- 前缀和技巧
- 滑动窗口算法框架详解
- 动态规划核心框架

力扣第 53 题「最大子序和」问题和前文讲过的 [经典动态规划：最长递增子序列](#) 的套路非常相似，代表着一类比较特殊的动态规划问题的思路，题目如下：

给你输入一个整数数组 `nums`，请你找在其中找一个和最大的子数组，返回这个子数组的和。函数签名如下：

```
int maxSubArray(int[] nums);
```

比如说输入 `nums = [-3, 1, 3, -1, 2, -4, 2]`，算法返回 5，因为最大子数组 `[1, 3, -1, 2]` 的和为 5。

其实第一次看到这道题，我首先想到的是滑动窗口算法，因为我们前文说过嘛，滑动窗口算法就是专门处理子串/子数组问题的，这里不就是子数组问题么？

前文 [滑动窗口算法框架详解](#) 中讲过，想用滑动窗口算法，先问自己几个问题：

- 1、什么时候应该扩大窗口？
- 2、什么时候应该缩小窗口？
- 3、什么时候更新答案？

我之前认为这题用不了滑动窗口算法，理由是在包含负数的条件下对 `nums` 求和，应该无法确定什么时候扩大和缩小窗口（类似 [前缀和习题](#) 中讲解的第 560 题）。但经读者评论的启发，我发现这道题确实是可以用滑动窗口技巧解决，不过有些 case 有点过于精巧，确实不容易想到。

所以我认为滑动窗口解法可以作为思路拓展，真正遇到类似的问题，还是以动态规划/前缀和的思路来做，更容易按照模板化的思维解决问题。下面我来依次讲解这三种解法。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。



# 经典动态规划：最长公共子序列



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">712. Minimum ASCII Delete Sum for Two Strings</a>	<a href="#">712. 两个字符串的最小ASCII删除和</a>	
<a href="#">1143. Longest Common Subsequence</a>	<a href="#">1143. 最长公共子序列</a>	
<a href="#">583. Delete Operation for Two Strings</a>	<a href="#">583. 两个字符串的删除操作</a>	

阅读本文前，你需要先学习：

- 动态规划核心框架

不知道大家做算法题有什么感觉，我总结出来做算法题的技巧就是，把大的问题细化到一个点，先研究在这个小的点上如何解决问题，然后再通过递归/迭代的方式扩展到整个问题。

比如说我们前文 [手把手带你刷二叉树第三期](#)，解决二叉树的题目，我们就会把整个问题细化到某一个节点上，想象自己站在某个节点上，需要做什么，然后套二叉树递归框架就行了。

动态规划系列问题也是一样，尤其是子序列相关的问题。本文从「最长公共子序列问题」展开，总结三道子序列问题，解这道题仔细讲讲这种子序列问题的套路，你就能感受到这种思维方式了。

## 最长公共子序列

计算最长公共子序列（Longest Common Subsequence，简称 LCS）是一道经典的动态规划题目，力扣第 1143 题「最长公共子序列」就是这个问题：

给你输入两个字符串 `s1` 和 `s2`，请你找出他们俩的最长公共子序列，返回这个子序列的长度。函数签名如下：

```
int longestCommonSubsequence(String s1, String s2);
```

比如说输入 `s1 = "zabcde"`, `s2 = "acez"`, 它俩的最长公共子序列是 `lcs = "ace"`, 长度为 3, 所以算法返回 3。

如果没有做过这道题，一个最简单的暴力算法就是，把 `s1` 和 `s2` 的所有子序列都穷举出来，然后看看有没有公共的，然后在所有公共子序列里面再寻找一个长度最大的。

显然，这种思路的复杂度非常高，你要穷举出所有子序列，这个复杂度就是指数级的，肯定不实际。

正确的思路是不要考虑整个字符串，而是细化到 `s1` 和 `s2` 的每个字符。前文 [子序列解题模板](#) 中总结的一个规律：

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 动态规划之子序列问题解题模板



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">516. Longest Palindromic Subsequence</a>	516. 最长回文子序列	
<a href="#">1312. Minimum Insertion Steps to Make a String Palindrome</a>	1312. 让字符串成为回文串的最少插入次数	

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

子序列问题是常见的算法问题，而且并不好解决。

首先，子序列问题本身就相对子串、子数组更困难一些，因为前者是不连续的序列，而后两者是连续的，就算穷举你都不一定会，更别说求解相关的算法问题了。

而且，子序列问题很可能涉及到两个字符串，比如前文 [最长公共子序列](#)，如果没有一定的处理经验，真的不容易想出来。所以本文就来扒一扒子序列问题的套路，其实就有两种模板，相关问题只要往这两种思路上想，十拿九稳。

一般来说，这类问题都是让你求一个最长子序列，因为最短子序列就是一个字符嘛，没啥可问的。一旦涉及到子序列和最值，那几乎可以肯定，考察的是动态规划技巧，时间复杂度一般都是  $O(n^2)$ 。

原因很简单，你想想一个字符串，它的子序列有多少种可能？起码是指数级的吧，这种情况下，不用动态规划技巧，还想怎么着？

既然要用动态规划，那就要定义  $dp$  数组，找状态转移关系。我们说的两种思路模板，就是  $dp$  数组的定义思路。不同的问题可能需要不同的  $dp$  数组定义来解决。

## 一、两种思路

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 经典动态规划：0-1 背包问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

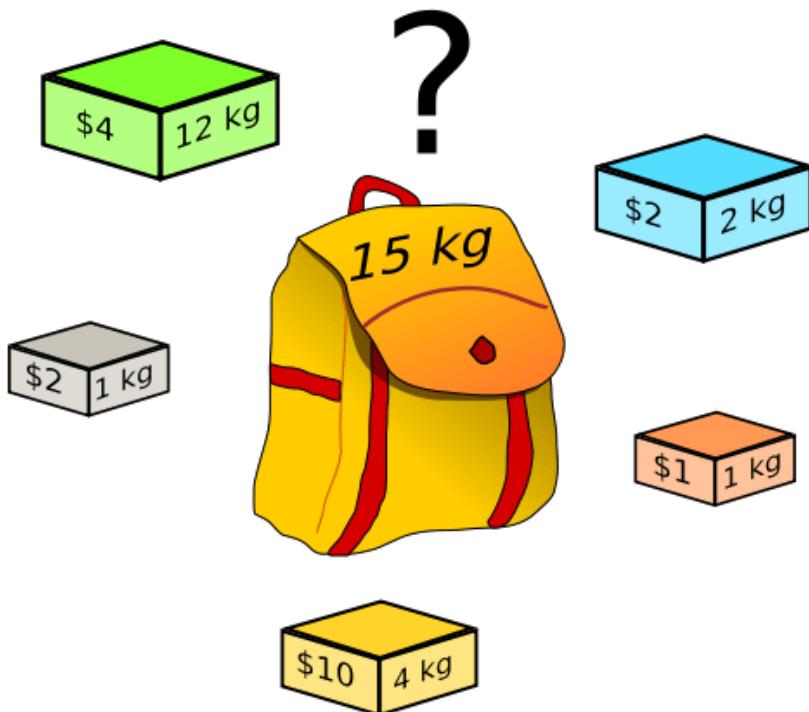
阅读本文前，你需要先学习：

- 动态规划核心框架

tip：本文有视频版：[0-1背包问题详解](#)。建议关注我的 B 站账号，我会用视频领读的方式带大家学习那些稍有难度的算法技巧。

后台天天有人问背包问题，这个问题其实不难，借助动态规划的思维框架，无非还是状态 + 选择，没啥特别之处。今天就来说一下背包问题吧，就讨论最常见的 0-1 背包问题。描述：

给你一个可装载重量为  $W$  的背包和  $N$  个物品，每个物品有重量和价值两个属性。其中第  $i$  个物品的重量为  $wt[i]$ ，价值为  $val[i]$ 。现在让你用这个背包装物品，每个物品只能用一次，在不超过被包容量的前提下，最多能装的价值是多少？



举个简单的例子，输入如下：

```
N = 3, W = 4  
wt = [2, 1, 3]  
val = [4, 2, 3]
```

算法返回 6，选择前两件物品装进背包，总重量 3 小于  $W$ ，可以获得最大价值 6。

题目就是这么简单，一个典型的动态规划问题。这个题目中的物品不可以分割，要么装进包里，要么不装，不能说切成两块装一半。这就是 0-1 背包这个名词的来历。

解决这个问题没有什么排序之类巧妙的方法，只能穷举所有可能，根据我们 [动态规划详解](#) 中的套路，直接走流程就行了。

## 动规标准套路

看来每篇动态规划文章都得重复一遍套路，历史文章中的动态规划问题都是按照下面的套路来的。

**第一步要明确两点，「状态」和「选择」。**

先说状态，如何才能描述一个问题局面？只要给几个物品和一个背包的容量限制，就形成了一个背包问题呀。所以状态有两个，就是「**背包的容量**」和「**可选择的物品**」。

再说选择，也很容易想到啊，对于每件物品，你能选择什么？**选择就是「装进背包」或者「不装进背包」嘛。**

明白了状态和选择，动态规划问题基本上就解决了，对于自底向上的思考方式，代码的一般框架是这样：

```
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][...] = 择优(选择1, 选择2...)
```

**第二步要明确  $dp$  数组的定义。**

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维  $dp$  数组。

$dp[i][w]$  的定义如下：对于前  $i$  个物品，当前背包的容量为  $w$ ，这种情况下可以装的最大价值是  $dp[i][w]$ 。

比如说，如果  $dp[3][5] = 6$ ，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 6。

为什么要这么定义？因为这样可以找到状态转移关系，或者说这就是背包问题的特殊定义方式，你当做套路记下来就行，未来遇到动态规划相关问题，都可以这样定义试一试。

根据这个定义，我们想求的最终答案就是  $dp[N][W]$ 。base case 就是  $dp[0][..] = dp[..][0] = 0$ ，因为没有物品或者背包没有空间的时候，能装的最大价值就是 0。

细化上面的框架：

```
int[][] dp[N+1][W+1]  
dp[0][..] = 0  
dp[..][0] = 0  
  
for i in [1..N]:  
    for w in [1..W]:  
        dp[i][w] = max(  
            把物品 i 装进背包,  
            不把物品 i 装进背包  
        )  
return dp[N][W]
```

### 第三步，根据「选择」，思考状态转移的逻辑。

简单说就是，上面伪码中「把物品  $i$  装进背包」和「不把物品  $i$  装进背包」怎么用代码体现出来呢？

这就要结合对  $dp$  数组的定义，看看这两种选择会对状态产生什么影响：

先重申一下刚才我们的  $dp$  数组的定义：

$dp[i][w]$  表示：对于前  $i$  个物品（从 1 开始计数），当前背包的容量为  $w$  时，这种情况下可以装下的最大价值是  $dp[i][w]$ 。

如果你没有把这第  $i$  个物品装入背包，那么很显然，最大价值  $dp[i][w]$  应该等于  $dp[i-1][w]$ ，继承之前的结果。

如果你把这第  $i$  个物品装入了背包，那么  $dp[i][w]$  应该等于  $val[i-1] + dp[i-1][w - wt[i-1]]$ 。

首先，由于数组索引从 0 开始，而我们定义中的  $i$  是从 1 开始计数的，所以  $val[i-1]$  和  $wt[i-1]$  表示第  $i$  个物品的价值和重量。

你如果选择将第  $i$  个物品装进背包，那么第  $i$  个物品的价值  $val[i-1]$  肯定就到手了，接下来你就要在剩余容量  $w - wt[i-1]$  的限制下，在前  $i - 1$  个物品中挑选，求最大价值，即  $dp[i-1][w - wt[i-1]]$ 。

综上就是两种选择，我们都已经分析完毕，也就是写出来了状态转移方程，可以进一步细化代码：

```
for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            dp[i-1][w],
            dp[i-1][w - wt[i-1]] + val[i-1]
        )
return dp[N][W]
```

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了  $w - wt[i-1]$  可能小于 0 导致数组索引越界的问题：

```
int knapsack(int W, int N, int[] wt, int[] val) {
    assert N == wt.length;
    // base case 已初始化
    int[][] dp = new int[N + 1][W + 1];
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (w - wt[i - 1] < 0) {
                // 这种情况下只能选择不装入背包
                dp[i][w] = dp[i - 1][w];
            } else {
                // 装入或者不装入背包，择优
                dp[i][w] = Math.max(
                    dp[i - 1][w - wt[i - 1]] + val[i - 1],
                    dp[i - 1][w]
                );
            }
        }
    }
}
```

```
    return dp[N][W];  
}
```

## ► 🌟 代码可视化动画🌟

其实函数签名中的物品数量 **N** 就是 **wt** 数组的长度，所以实际上这个参数 **N** 是多此一举的。但为了体现原汁原味的 0-1 背包问题，我就带上这个参数 **N** 了，你自己写的话可以省略。

至此，背包问题就解决了，相比而言，我觉得这是比较简单的动态规划问题，因为状态转移的推导比较自然，基本上你明确了 **dp** 数组的定义，就可以理所当然地确定状态转移了。

## ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1235. Maximum Profit in Job Scheduling</a>	<a href="#">1235. 规划兼职工作</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 经典动态规划：子集背包问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">416. Partition Equal Subset Sum</a>	<a href="#">416. 分割等和子集</a>	困难

阅读本文前，你需要先学习：

- 动态规划核心框架
- 0-1 背包问题详解

上篇文章 [经典动态规划：0-1 背包问题](#) 详解了通用的 0-1 背包问题，今天来看看背包问题的思想能够如何运用到其他算法题目。

读者在阅读本文之前务必读懂前文 [经典动态规划：0-1 背包问题](#) 中讲的套路，因为本文就是按照背包问题的解题模板来讲解的。

## 一、问题分析

看一下力扣第 416 题「分割等和子集」：

输入一个只包含正整数的非空数组 `nums`，请你写一个算法，判断这个数组是否可以被分割成两个子集，使得两个子集的元素和相等。

算法的函数签名如下：

```
// 输入一个集合，返回是否能够分割成和相等的两个子集
boolean canPartition(int[] nums);
```

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

# 经典动态规划：完全背包问题



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">518. Coin Change II</a>	<a href="#">518. 零钱兑换 II</a>	困难

-----

阅读本文前，你需要先学习：

- 动态规划核心框架
- 0-1 背包问题详解

力扣第 322 题「零钱兑换 I」在 [动态规划套路详解](#) 作为经典动态规划的例题讲过。本文讲的零钱兑换 II 是另一种典型背包问题的变体，我们前文已经讲了 [经典动态规划：0-1 背包问题](#) 和 [背包问题变体：相等子集分割](#)。

读本文之前，希望你已经看过前两篇文章，看过了动态规划和背包问题的套路，这篇继续按照背包问题的套路，列举一个背包问题的变形。

来看力扣第 518 题「零钱兑换 II」：

## ▼ 518. 零钱兑换 II Leetcode | 力扣

给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。

假设每一种面额的硬币有无限个。

题目数据保证结果符合 32 位带符号整数。

示例 1：

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2：

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额 2 的硬币不能凑成总金额 3。
```

### 示例 3:

```
输入: amount = 10, coins = [10]
输出: 1
```

### 提示:

- `1 <= coins.length <="300"`
- `1 <= coins[i] <="5000"`
- `coins` 中的所有值互不相同
- `0 <= amount <="5000"`

我们要完成的函数的签名如下：

```
int change(int amount, int[] coins);
```

我们可以把这个问题转化为背包问题的描述形式：

有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

这个问题和我们前面讲过的两个背包问题，有一个最大的区别就是，每个物品的数量是无限的，这也就是传说中的「完全背包问题」，没啥高大上的，无非就是状态转移方程有一点变化而已。

下面就以背包问题的描述形式，继续按照流程来分析。

## 解题思路

**第一步要明确两点，「状态」和「选择」。**

状态有两个，就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」嘛，背包问题的套路都是这样。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
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][...] = 计算(选择1, 选择2...)
```

### 第二步要明确 `dp` 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 `dp` 数组。

`dp[i][j]` 的定义如下：

若只使用前  $i$  个物品（可以重复使用），当背包容量为  $j$  时，有  $dp[i][j]$  种方法可以装满背包。

换句话说，翻译回我们题目的意思就是：

若只使用  $\text{coins}$  中的前  $i$  个（ $i$  从 1 开始计数）硬币的面值，若想凑出金额  $j$ ，有  $dp[i][j]$  种凑法。

经过以上的定义，可以得到：

base case 为  $dp[0][..] = 0$ ,  $dp[..][0] = 1$ 。 $i = 0$  代表不使用任何硬币面值，这种情况下显然无法凑出任何金额； $j = 0$  代表需要凑出的目标金额为 0，那么什么都不做就是唯一的一种凑法。

我们最终想得到的答案就是  $dp[N][amount]$ ，其中  $N$  为  $\text{coins}$  数组的大小。

大致的伪码思路如下：

```
int dp[N+1][amount+1]
dp[0][..] = 0
dp[..][0] = 1

for i in [1..N]:
    for j in [1..amount]:
        把物品 i 装进背包,
        不把物品 i 装进背包
return dp[N][amount]
```

第三步，根据「选择」，思考状态转移的逻辑。

注意，我们这个问题的特殊点在于物品的数量是无限的，所以这里和之前写的 [0-1 背包问题](#) 文章有所不同。

如果你不把这第  $i$  个物品装入背包，也就是说你不使用  $\text{coins}[i-1]$  这个面值的硬币，那么凑出金额  $j$  的方法数  $dp[i][j]$  应该等于  $dp[i-1][j]$ ，继承之前的结果。

如果你把这第  $i$  个物品装入了背包，也就是说你使用  $\text{coins}[i-1]$  这个面值的硬币，那么  $dp[i][j]$  应该等于  $dp[i-1][j-coins[i-1]]$ 。

由于定义中的  $i$  是从 1 开始计数的，所以  $\text{coins}$  的索引是  $i-1$  时表示第  $i$  个硬币的面值。

$dp[i][j-coins[i-1]]$  也不难理解，如果你决定使用这个面值的硬币，那么就应该关注如何凑出金额  $j - \text{coins}[i-1]$ 。

比如说，你想用面值为 2 的硬币凑出金额 5，那么如果你知道了凑出金额 3 的方法，再加上一枚面额为 2 的硬币，不就可以凑出 5 了嘛。

综上就是两种选择，而我们想求的  $dp[i][j]$  是「共有多少种凑法」，所以  $dp[i][j]$  的值应该是以上两种选择的结果之和：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= amount; j++) {
        if (j - coins[i-1] >= 0) {
            dp[i][j] = dp[i-1][j] + dp[i-1][j-coins[i-1]];
        }
    }
}
return dp[N][W]
```

有的读者在这里可能会有疑问，不是说可以重复使用硬币吗？那么如果我确定「使用第  $i$  个面值的硬币」，我怎么确定这个面值的硬币被使用了多少枚？简单的一个  $dp[i][j-coins[i-1]]$  可以包含重复使用第  $i$  个硬币的情况吗？

对于这个问题，建议你再仔回头细阅读一下我们对  $dp$  数组的定义，然后把这个定义代入  $dp[i][j-coins[i-1]]$  看看：

若只使用前  $i$  个物品（可以重复使用），当背包容量为  $j-coins[i-1]$  时，有  $dp[i][j-coins[i-1]]$  种方法可以装满背包。

看到了吗， $dp[i][j-coins[i-1]]$  也是允许你使用第  $i$  个硬币的，所以说已经包含了重复使用硬币的情况，你一百个放心。

**最后一步，把伪码翻译成代码，处理一些边界情况。**

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了一些边界问题：

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[][] dp = new int[n + 1][amount + 1];
        // base case
        for (int i = 0; i <= n; i++)
            dp[i][0] = 1;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= amount; j++) {
                if (j - coins[i-1] >= 0)
                    dp[i][j] = dp[i - 1][j]
                               + dp[i][j - coins[i-1]];
                else
                    dp[i][j] = dp[i - 1][j];
            }
        }
        return dp[n][amount];
    }
}
```

### ▶ 彩虹 代码可视化动画

而且，我们通过观察可以发现， $dp$  数组的转移只和  $dp[i][\dots]$  和  $dp[i-1][\dots]$  有关，所以可以使用 [动态规划空间压缩技巧](#)，进一步降低算法的空间复杂度：

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[] dp = new int[amount + 1];
        // base case
        dp[0] = 1;
        for (int i = 0; i < n; i++)
            for (int j = 1; j <= amount; j++)
                if (j - coins[i] >= 0)
                    dp[j] = dp[j] + dp[j-coins[i]];
```

```
        return dp[amount];
    }
}
```

这个解法和之前的思路完全相同，将二维 `dp` 数组压缩为一维，时间复杂度  $O(N*amount)$ ，空间复杂度  $O(amount)$ 。

至此，这道零钱兑换问题也通过背包问题的框架解决了。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 背包问题的变体：目标和



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

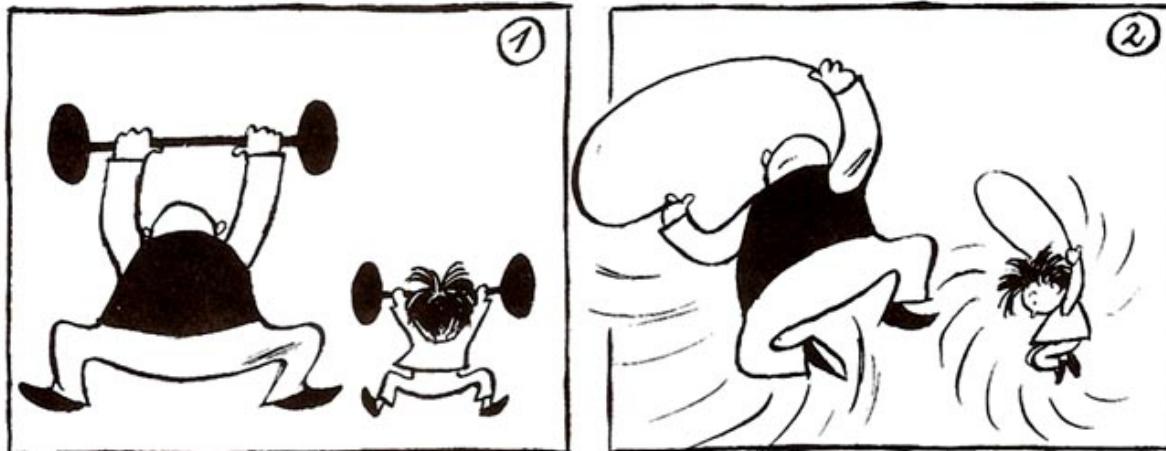
LeetCode	力扣	难度
<a href="#">494. Target Sum</a>	<a href="#">494. 目标和</a>	简单

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

我们前文经常说回溯算法和递归算法有点类似，有的问题如果实在想不出状态转移方程，尝试用回溯算法暴力解决也是一个聪明的策略，总比写不出来解法强。

那么，回溯算法和动态规划到底是啥关系？它俩都涉及递归，算法模板看起来还挺像的，都涉及做「选择」，真的酷似父与子。



那么，它俩具体有啥区别呢？回溯算法和动态规划之间，是否可能互相转化呢？

今天就用力扣第 494 题「目标和」来详细对比一下回溯算法和动态规划，题目如下：

### ▼ 494. 目标和 | 力扣

给你一个非负整数数组 `nums` 和一个整数 `target`。

向数组中的每个整数前添加 `%%%%%+%%%%%` 或 `%%%%%-%%%%%`，然后串联起所有整数，可以构造一个 表达式：

- 例如，`nums = [2, 1]`，可以在 `2` 之前添加 `%%%%%+%%%%%`，在 `1` 之前添加 `%%%%%-%%%%%`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 **target** 的不同 表达式 的数目。

### 示例 1:

输入: `nums = [1,1,1,1,1], target = 3`

输出: 5

解释: 一共有 5 种方法让最终目标和为 3 。

`-1 + 1 + 1 + 1 + 1 = 3`

`+1 - 1 + 1 + 1 + 1 = 3`

`+1 + 1 - 1 + 1 + 1 = 3`

`+1 + 1 + 1 - 1 + 1 = 3`

`+1 + 1 + 1 + 1 - 1 = 3`

### 示例 2:

输入: `nums = [1], target = 1`

输出: 1

### 提示:

- `1 <= nums.length <= 20`
- `0 <= nums[i] <= 1000`
- `0 <= sum(nums[i]) <= 1000`
- `-1000 <= target <= 1000`

函数的签名如下:

```
int findTargetSumWays(int[] nums, int target);
```

## 一、回溯思路

其实我第一眼看到这个题目，花了两分钟就写出了一个回溯解法。

任何算法的核心都是穷举，回溯算法就是一个暴力穷举算法，前文 [回溯算法解题框架](#) 就写了回溯算法框架：

```
def backtrack(路径, 选择列表):  
    if 满足结束条件:  
        result.add(路径)  
        return  
  
    for 选择 in 选择列表:  
        做选择  
        backtrack(路径, 选择列表)  
        撤销选择
```

关键就是搞清楚什么是「选择」，而对于这道题，「选择」不是明摆着的吗？对于每个数字 `nums[i]`，我们可以选择给一个正号 `+` 或者一个负号 `-`，然后利用回溯模板穷举出来所有可能的结果，数一数到底有几种组合能够凑出 `target` 不就行了嘛？

伪码思路如下：

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 动态规划之最小路径和



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">64. Minimum Path Sum</a>	<a href="#">64. 最小路径和</a>	简单

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

今天聊一道经典的动态规划题目，它是力扣第 64 题「最小路径和」：

## ▼ 64. 最小路径和 [Leetcode](#) | [力扣](#)

给定一个包含非负整数的  $m \times n$  网格 `grid`，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

**说明：**每次只能向下或者向右移动一步。

**示例 1：**

1	3	1
1	5	1
4	2	1

**输入：** `grid = [[1,3,1],[1,5,1],[4,2,1]]`

**输出：** 7

**解释：** 因为路径 `1→3→1→1→1` 的总和最小。

**示例 2：**

```
输入: grid = [[1,2,3],[4,5,6]]  
输出: 12
```

提示:

- $m == \text{grid.length}$
- $n == \text{grid[i].length}$
- $1 \leq m, n \leq 200$
- $0 \leq \text{grid}[i][j] \leq 200$

函数签名如下:

```
int minPathSum(int[][] grid);
```

其实这道题难度不算大，但你可能会遇到一些难度比较大的变体，所以统一讲一讲这种问题的通用思路。

一般来说，让你在二维矩阵中求最优化问题（最大值或者最小值），肯定需要递归 + 备忘录，也就是动态规划技巧。

就拿题目举的例子来说，我给图中的几个格子编个号方便描述：

1 <b>D</b>	3	1
1	5	A 1
4	2 <b>C</b>	B 1

我们想计算从起点 D 到达 B 的最小路径和，那你说怎么才能到达 B 呢？

题目说了只能向右或者向下走，所以只有从 A 或者 C 走到 B。

那么算法怎么知道从 A 走到 B 才能使路径和最小，而不是从 C 走到 B 呢？

难道是因为位置 A 的元素大小是 1，位置 C 的元素是 2，1 小于 2，所以一定要从 A 走到 B 才能使路径和最小吗？

其实不是的，真正的原因是，从 D 走到 A 的最小路径和是 6，而从 D 走到 C 的最小路径和是 8，6 小于 8，所以一定要从 A 走到 B 才能使路径和最小。

换句话说，我们把「从 D 走到 B 的最小路径和」这个问题转化成了「从 D 走到 A 的最小路径和」和「从 D 走到 C 的最小路径和」这两个问题。

理解了上面的分析，这不就是状态转移方程吗？所以这个问题肯定会用到动态规划技巧来解决。

比如我们定义如下一个 dp 函数：

```
int dp(int[][] grid, int i, int j);
```

这个 `dp` 函数的定义如下：

从左上角位置  $(0, 0)$  走到位置  $(i, j)$  的最小路径和为 `dp(grid, i, j)`。

根据这个定义，我们想求的最小路径和就可以通过调用这个 `dp` 函数计算出来：

```
int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 计算从左上角走到右下角的最小路径和
    return dp(grid, m - 1, n - 1);
}
```

再根据刚才的分析，很容易发现，`dp(grid, i, j)` 的值取决于 `dp(grid, i - 1, j)` 和 `dp(grid, i, j - 1)` 返回的值。

我们可以直接写代码了：

```
int dp(int[][] grid, int i, int j) {
    // base case
    if (i == 0 && j == 0) {
        return grid[0][0];
    }
    // 如果索引出界，返回一个很大的值，
    // 保证在取 min 的时候不会被取到
    if (i < 0 || j < 0) {
        return Integer.MAX_VALUE;
    }

    // 左边和上面的最小路径和加上 grid[i][j]
    // 就是到达 (i, j) 的最小路径和
    return Math.min(
        dp(grid, i - 1, j),
        dp(grid, i, j - 1)
    ) + grid[i][j];
}
```

上述代码逻辑已经完整了，接下来就分析一下，这个递归算法是否存在重叠子问题？是否需要用备忘录优化一下执行效率？

前文多次说过判断重叠子问题的技巧，首先抽象出上述代码的递归框架：

```
int dp(int i, int j) {
    dp(i - 1, j); // #1
    dp(i, j - 1); // #2
}
```

如果我想从 `dp(i, j)` 递归到 `dp(i-1, j-1)`，有几种不同的递归调用路径？

可以是 `dp(i, j) -> #1 -> #2` 或者 `dp(i, j) -> #2 -> #1`，不止一种，说明 `dp(i-1, j-1)` 会被多次计算，所以一定存在重叠子问题。

那么我们可以使用备忘录技巧进行优化：

```
class Solution {
    // 备忘录
    int[][] memo;

    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        // 构造备忘录，初始值全部设为 -1
        memo = new int[m][n];
        for (int[] row : memo)
            Arrays.fill(row, -1);

        return dp(grid, m - 1, n - 1);
    }

    int dp(int[][] grid, int i, int j) {
        // base case
        if (i == 0 && j == 0) {
            return grid[0][0];
        }
        if (i < 0 || j < 0) {
            return Integer.MAX_VALUE;
        }
        // 避免重复计算
        if (memo[i][j] != -1) {
            return memo[i][j];
        }
        // 将计算结果记入备忘录
        memo[i][j] = Math.min(
            dp(grid, i - 1, j),
            dp(grid, i, j - 1)
        ) + grid[i][j];

        return memo[i][j];
    }
}
```

## ▶ 🔍 代码可视化动画🔍

至此，本题就算是解决了，时间复杂度和空间复杂度都是  $O(MN)$ ，标准的自顶向下动态规划解法。

有的读者可能问，能不能用自底向上的迭代解法来做这道题呢？完全可以的。

首先，类似刚才的 `dp` 函数，我们需要一个二维 `dp` 数组，定义如下：

从左上角位置  $(0, 0)$  走到位置  $(i, j)$  的最小路径和为 `dp[i][j]`。

状态转移方程当然不会变的，`dp[i][j]` 依然取决于 `dp[i-1][j]` 和 `dp[i][j-1]`，直接看代码吧：

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
```

```

int[][] dp = new int[m][n];

// **** base case ****
dp[0][0] = grid[0][0];

for (int i = 1; i < m; i++)
    dp[i][0] = dp[i - 1][0] + grid[i][0];

for (int j = 1; j < n; j++)
    dp[0][j] = dp[0][j - 1] + grid[0][j];
// *****

// 状态转移
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        dp[i][j] = Math.min(
            dp[i - 1][j],
            dp[i][j - 1]
        ) + grid[i][j];
    }
}

return dp[m - 1][n - 1];
}
}

```

这个解法的 **base case** 看起来和递归解法略有不同，但实际上是一样的。

因为状态转移为下面这段代码：

```

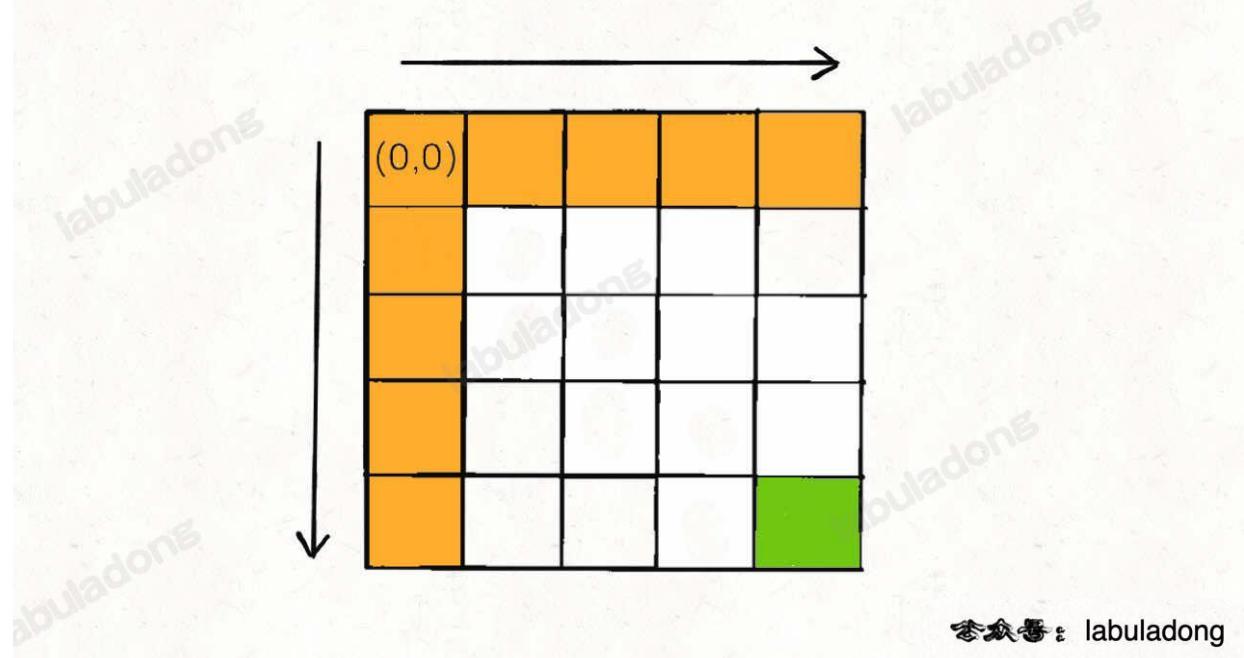
dp[i][j] = Math.min(
    dp[i - 1][j],
    dp[i][j - 1]
) + grid[i][j];

```

那如果 **i** 或者 **j** 等于 0 的时候，就会出现索引越界的错误。

所以我们需要提前计算出 **dp[0][..]** 和 **dp[..][0]**，然后让 **i** 和 **j** 的值从 1 开始迭代。

**dp[0][..]** 和 **dp[..][0]** 的值怎么算呢？其实很简单，第一行和第一列的路径和只有下面这一种情况嘛：



© labuladong

那么按照 `dp` 数组的定义，`dp[i][0] = sum(grid[0..i][0])`, `dp[0][j] = sum(grid[0][0..j])`，也就是如下代码：

```
// **** base case ****
dp[0][0] = grid[0][0];

for (int i = 1; i < m; i++)
    dp[i][0] = dp[i - 1][0] + grid[i][0];

for (int j = 1; j < n; j++)
    dp[0][j] = dp[0][j - 1] + grid[0][j];
// *****
```

到这里，自底向上的迭代解法也搞定了，那有的读者可能又要问了，能不能优化一下算法的空间复杂度呢？

前文 [动态规划的降维打击：空间压缩](#) 说过降低 `dp` 数组的技巧，这里也是适用的，不过略微复杂些，本文由于篇幅所限就不写了，有兴趣的读者可以自己尝试一下。

本文到此结束，下篇文章写一道进阶题目，更加巧妙和有趣，敬请期待~

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	<a href="#">剑指 Offer 47. 礼物的最大价值</a>	●
-	<a href="#">剑指 Offer II 099. 最小路径之和</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 动态规划帮我通关了《魔塔》



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode

力扣

难度

174. Dungeon Game    174. 地下城游戏



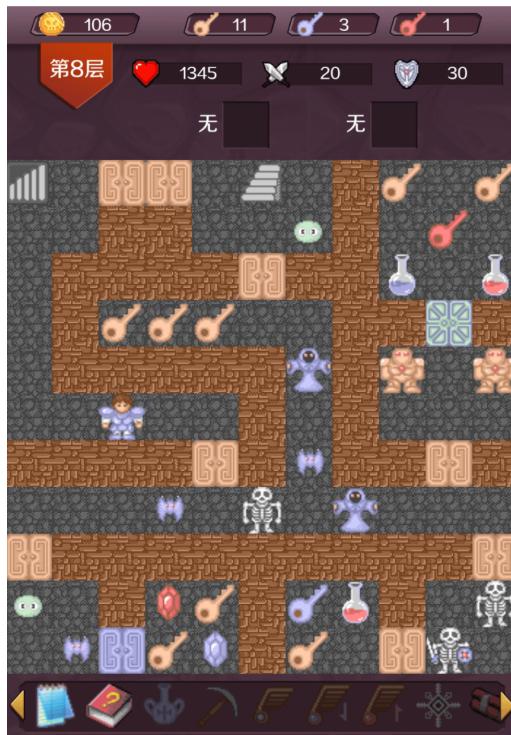
-----

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

「魔塔」是一款经典的地牢类游戏，碰怪物要掉血，吃血瓶能加血，你要收集钥匙，一层一层上楼，最后救出美丽的公主。

现在手机上仍然可以玩这个游戏：



嗯，相信这款游戏承包了不少人的童年回忆，记得小时候，一个人拿着游戏机玩，两三个人围在左右指手画脚，这导致玩游戏的人体验极差，而左右的人异常快乐 😂

力扣第 174 题「地下城游戏」是一道类似的题目：

▼ [174. 地下城游戏 Leetcode | 力扣](#)

```
.dungeon th, .dungeon td { text-align: center; height: 70px; width: 70px; }
```

恶魔们抓住了公主并将她关在了地下城 **dungeon** 的 **右下角**。地下城是由  $m \times n$  个房间组成的二维网格。我们英勇的骑士最初被安置在 **左上角** 的房间里，他必须穿过地下城并通过对抗恶魔来拯救公主。

骑士的初始健康点数为一个正整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。

有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为负整数，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 0），要么包含增加骑士健康点数的魔法球（若房间里的值为正整数，则表示骑士将增加健康点数）。

为了尽快解救公主，骑士决定每次只 **向右** 或 **向下** 移动一步。

返回确保骑士能够拯救到公主所需的最低初始健康点数。

**注意：**任何房间都可能对骑士的健康点数造成威胁，也可能增加骑士的健康点数，包括骑士进入的左上角房间以及公主被监禁的右下角房间。

**示例 1：**

-2	-3	3
-5	-10	1
10	30	-5

**输入：** dungeon = [[-2, -3, 3], [-5, -10, 1], [10, 30, -5]]

**输出：** 7

**解释：**如果骑士遵循最佳路径：右  $\rightarrow$  右  $\rightarrow$  下  $\rightarrow$  下， 则骑士的初始健康点数至少为 7。

**示例 2：**

**输入：** dungeon = [[0]]

**输出：** 1

**提示：**

- $m == \text{dungeon.length}$
- $n == \text{dungeon[i].length}$
- $1 \leq m, n \leq 200$
- $-1000 \leq \text{dungeon}[i][j] \leq 1000$

简单说，就是问你至少需要多少初始生命值，能够让骑士从最左上角移动到最右下角，且任何时候生命值都要大于 0。

函数签名如下：

```
int calculateMinimumHP(int[][] grid);
```

上篇文章 [最小路径和](#) 写过类似的问题，问你从左上角到右下角的最小路径和是多少。

我们做算法题一定要尝试举一反三，感觉今天这道题和最小路径和有点关系对吧？

想要最小化骑士的初始生命值，是不是意味着要最大化骑士行进路线上的血瓶？是不是相当于求「最大路径和」？是不是可以直接套用计算「最小路径和」的思路？

但是稍加思考，发现这个推论并不成立，吃到最多的血瓶，并不一定就能获得最小的初始生命值。

比如如下这种情况，如果想要吃到最多的血瓶获得「最大路径和」，应该按照下图箭头所示的路径，初始生命值需要 11：



但也很容易看到，正确的答案应该是下图箭头所示的路径，初始生命值只需要 1：



所以，关键不在于吃最多的血瓶，而是在于如何损失最少的生命值。

这类求最值的问题，肯定要借助动态规划技巧，要合理设计 `dp` 数组/函数的定义。类比前文 [最小路径和问题](#)，`dp` 函数签名肯定长这样：

```
int dp(int[][] grid, int i, int j);
```

但是这道题对 `dp` 函数的定义比较有意思，按照常理，这个 `dp` 函数的定义应该是：

从左上角 (`grid[0][0]`) 走到 `grid[i][j]` 至少需要 `dp(grid, i, j)` 的生命值。

这样定义的话，base case 就是 `i, j` 都等于 0 的时候，我们可以这样写代码：

```
int calculateMinimumHP(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 我们想计算左上角到右下角所需的最小生命值
    return dp(grid, m - 1, n - 1);
}

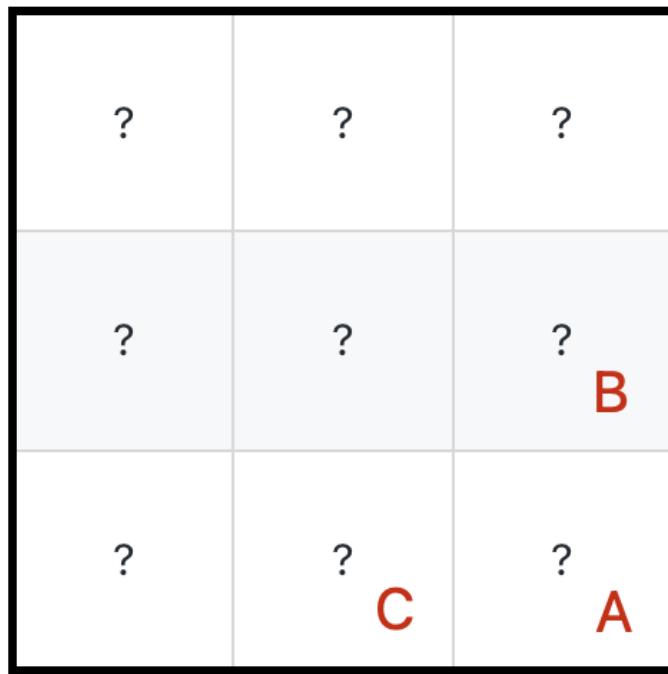
int dp(int[][] grid, int i, int j) {
    // base case
    if (i == 0 && j == 0) {
        // 保证骑士落地不死就行了
        return grid[i][j] > 0 ? 1 : -grid[i][j] + 1;
    }
    ...
}
```

为了简洁，之后 `dp(grid, i, j)` 就简写为 `dp(i, j)`，大家理解就好。

接下来我们需要找状态转移了，还记得如何找状态转移方程吗？我们这样定义 `dp` 函数能否正确进行状态转移呢？

我们希望 `dp(i, j)` 能够通过 `dp(i-1, j)` 和 `dp(i, j-1)` 推导出来，这样就能不断逼近 base case，也就能够正确进行状态转移。

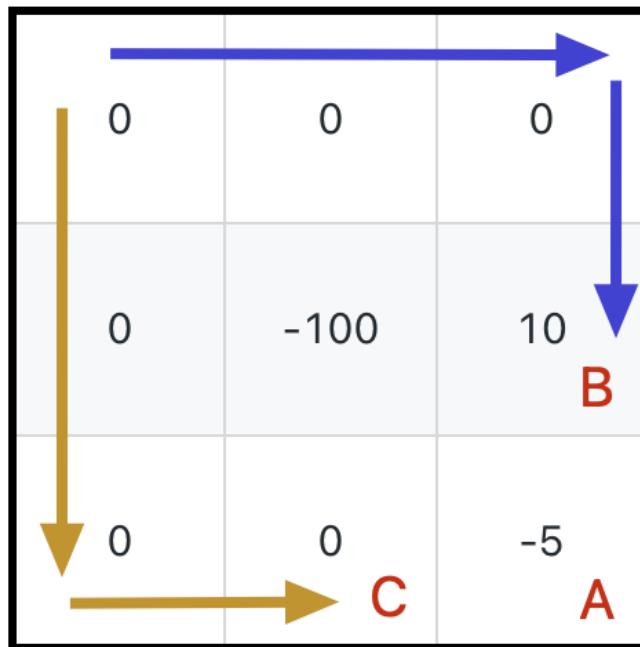
具体来说，「到达 A 的最小生命值」应该能够由「到达 B 的最小生命值」和「到达 C 的最小生命值」推导出来：



但问题是，能推出来么？实际上是不能的。

因为按照 `dp` 函数的定义，你只知道「能够从左上角到达 B 的最小生命值」，但并不知道「到达 B 时的生命值」。

「到达 B 时的生命值」是进行状态转移的必要参考，我给你举个例子你就明白了，假设下图这种情况：



你说这种情况下，骑士救公主的最优路线是什么？

显然是按照图中蓝色的线走到 B，最后走到 A 对吧，这样初始血量只需要 1 就可以；如果走黄色箭头这条路，先走到 C 然后走到 A，初始血量至少需要 6。

为什么会这样呢？骑士走到 B 和 C 的最少初始血量都是 1，为什么最后是从 B 走到 A，而不是从 C 走到 A 呢？

因为骑士走到 B 的时候生命值为 11，而走到 C 的时候生命值依然是 1。

如果骑士执意要通过 C 走到 A，那么初始血量必须加到 6 点才行；而如果通过 B 走到 A，初始血量为 1 就够了，因为路上吃到血瓶了，生命值足够抗 A 上面怪物的伤害。

这下应该说的很清楚了，再回顾我们对 dp 函数的定义，上图的情况，算法只知道  $dp(1, 2) = dp(2, 1) = 1$ ，都是一样的，怎么做出正确的决策，计算出  $dp(2, 2)$  呢？

所以说，我们之前对 dp 数组的定义是错误的，信息量不足，算法无法做出正确的状态转移。

正确的做法需要反向思考，依然是如下的 dp 函数：

```
int dp(int[][] grid, int i, int j);
```

但是我们要修改 dp 函数的定义：

从  $grid[i][j]$  到达终点（右下角）所需的最少生命值是  $dp(grid, i, j)$ 。

那么可以这样写代码：

```
int calculateMinimumHP(int[][] grid) {
    // 我们想计算左上角到右下角所需的最小生命值
    return dp(grid, 0, 0);
}

int dp(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // base case
    if (i == m - 1 && j == n - 1) {
        return grid[i][j] >= 0 ? 1 : -grid[i][j] + 1;
    }
    ...
}
```

根据新的 dp 函数定义和 base case，我们想求  $dp(0, 0)$ ，那就应该试图通过  $dp(i, j+1)$  和  $dp(i+1, j)$  推导出  $dp(i, j)$ ，这样才能不断逼近 base case，正确进行状态转移。

具体来说，「从 A 到达右下角的最少生命值」应该由「从 B 到达右下角的最少生命值」和「从 C 到达右下角的最少生命值」推导出来：

A(0,0)	B(0,1)	
?	?	?
C(1,0)		
?	?	?
	?	?
	?	?

能不能推导出来呢？这次是可以的，假设  $dp(0, 1) = 5$ ,  $dp(1, 0) = 4$ , 那么可以肯定要从 A 走向 C, 因为 4 小于 5 嘛。

那么怎么推出  $dp(0, 0)$  是多少呢？

假设 A 的值为 1, 既然知道下一步要往 C 走, 且  $dp(1, 0) = 4$  意味着走到  $grid[1][0]$  的时候至少要有 4 点生命值, 那么就可以确定骑士出现在 A 点时需要  $4 - 1 = 3$  点初始生命值, 对吧。

那如果 A 的值为 10, 落地就能捡到一个大血瓶, 超出了后续需求,  $4 - 10 = -6$  意味着骑士的初始生命值为负数, 这显然不可以, 骑士的生命值小于 1 就挂了, 所以这种情况下骑士的初始生命值应该是 1。

综上, 状态转移方程已经推出来了:

```
int res = min(
    dp(i + 1, j),
    dp(i, j + 1)
) - grid[i][j];

dp(i, j) = res <= 0 ? 1 : res;
```

根据这个核心逻辑, 加一个备忘录消除重叠子问题, 就可以直接写出最终的代码了:

```
class Solution {
    // 主函数
    public int calculateMinimumHP(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        // 备忘录中都初始化为 -1
        memo = new int[m][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
    }
```

```
    return dp(grid, 0, 0);
}

// 备忘录，消除重叠子问题
int[][] memo;

// 定义：从 (i, j) 到达右下角，需要的初始血量至少是多少
int dp(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // base case
    if (i == m - 1 && j == n - 1) {
        return grid[i][j] >= 0 ? 1 : -grid[i][j] + 1;
    }
    if (i == m || j == n) {
        return Integer.MAX_VALUE;
    }
    // 避免重复计算
    if (memo[i][j] != -1) {
        return memo[i][j];
    }
    // 状态转移逻辑
    int res = Math.min(
        dp(grid, i, j + 1),
        dp(grid, i + 1, j)
    ) - grid[i][j];
    // 骑士的生命值至少为 1
    memo[i][j] = res <= 0 ? 1 : res;

    return memo[i][j];
}
}
```

---

### ▶ 代码可视化动画

---

这就是自顶向下带备忘录的动态规划解法，参考前文 [动态规划套路详解](#) 很容易就可以改写成 `dp` 数组的迭代解法，这里就不写了，读者可以尝试自己写一写。

这道题的核心是定义 `dp` 函数，找到正确的状态转移方程，从而计算出正确的答案。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 动态规划帮我通关了《辐射4》



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">514. Freedom Trail</a>	<a href="#">514. 自由之路</a>	困难

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

本文的封面图是一款叫做《辐射4》的游戏中的一个任务剧情画面：



这个可以转动的圆盘类似是一个密码机关，中间偏上的位置有个红色的指针看到没，你只要转动圆盘可以让指针指向不同的字母，然后再按下中间的按钮就可以输入指针指向的字母。

只要转动圆环，让指针依次指向 R、A、I、L、R、O、A、D 并依次按下按钮，就可以触发机关，打开旁边的门。

至于密码为什么是这几个字母，在游戏中的剧情有暗示，这里就不多说了。

那么这个游戏场景和动态规划有什么关系呢？

我们来没事儿找事儿地想一想，拨动圆盘输入这些字母还挺麻烦的，按照什么顺序才能使得拨动圆盘所需的操作次数最少呢？

拨动圆盘的不同方法所需的操作次数肯定是不同的。

比如说你想把一个字母对准到指针上，你可以顺时针转圆盘，也可以逆时针转圆盘；而且某些字母可能不止出现一次，比如上图中大写字母 O 就在圆盘的不同位置出现了三次，你到时候应该拨哪个 O 才能使得整体的操作次数最少呢？

我们之前也多次说过，遇到求最值的问题，基本都是由动态规划算法来解决，因为动态规划本身就是运筹优化算法的一种嘛。

力扣上就有一道这个转盘游戏的算法题，难度还是 Hard，但我当时看了一眼就做出来了，因为我以前思考过生活中一个非常有意思的例子可以类比到这个问题，下面来简单介绍一下。

我曾经练习过李斯特和肖邦的几首钢琴曲，没练过钢琴的读者可能不知道，练习钢琴曲谱是需要提前确定「指法」的。

五线谱的音符七上八下的，两个手的手指必须互相配合，也就是说你必须确定好每个音符用哪只手的哪个手指来弹奏，写到谱子上。

比如说我很喜欢的一首曲子叫做《爱之梦》，这是我初学时的谱子：



音符上的数字 1 代表用大拇指，2 代表用食指，以此类推。按照确定下来的指法不断练习，形成肌肉记忆，就算是练会一首曲子了。

指法这东西因人而异，比如手大的人可以让中指跨到大拇指的左边，手小的人可能就有些别扭，那同一段谱子对应的指法可能就不一样。

那么问题来了，我应该如何设计指法，才能最小化手指切换的「别扭程度」，也就是最大化演奏的流畅度呢？

这里我就借助了动态规划算法技巧：手指的切换不就是状态的转移么？参考前文 [动态规划套路详解](#)，只要明确「状态」和「选择」就可以解决这个问题。

状态是什么？状态就是「下一个需要弹奏的音符」和「当前的手的状态」。

下一个需要弹奏的音符，无非就是钢琴上 88 个琴键中的一个；手的状态也很简单，五个手指头，每个手指头要么按下去了要么没按下去，2 的 5 次方 32 种情况，5 个二进制位就可以表示。

选择是什么？选择就是「下一个音符应该由哪个手指头来弹」，无非就是穷举五个手指头。

当然，结合当前手的状态，做出每个选择需要对应代价的，刚才说过这个代价是因人而异的，所以我需要给自己定制一个损失函数，计算不同指法切换的「别扭程度」。

现在的问题就变成了一个标准的动态规划问题，根据损失函数做出「别扭程度」最小的选择，使得整段演奏最流畅……

当然，最后这个算法时间复杂度太高了，我们刚才分析的只是单个的音符，但如果串成曲子，时空复杂度还得再乘曲子的音符数，很大。

而且，这个损失函数很难量化，钢琴的黑白键命中难度不同，而且「别扭程度」只能靠感觉，有点不严谨……

不过，本就没必要计算整首曲子的指法，只需要计算某些复杂段落的指法即可，这个算法还是比较有效的。

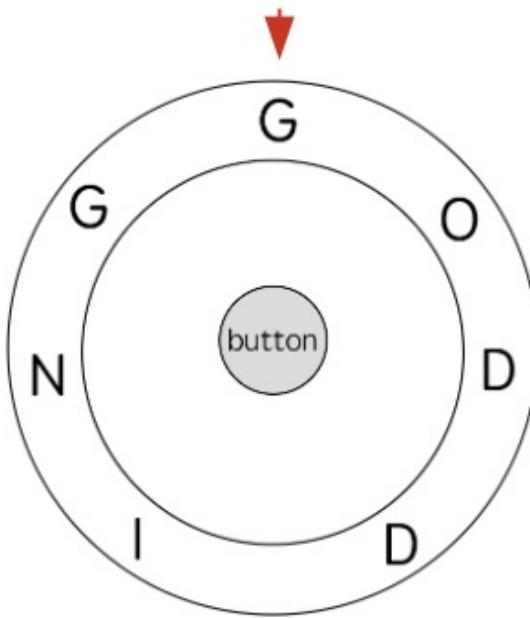
扯了这么多题外话终于要步入正题了，今天要讲的力扣第 514 题「自由之路」和钢琴指法问题有异曲同工之妙，如果你能理解钢琴的例子，相信你也能很快做出这道算法题。

题目给你输入一个字符串 `ring` 代表圆盘上的字符（指针位置在 12 点钟方向，初始指向 `ring[0]`），再输入一个字符串 `key` 代表你需要拨动圆盘输入的字符串，你的算法需要返回输入这个 `key` 至少进行多少次操作（拨动一格圆盘和按下圆盘中间的按钮都算是一次操作）。

函数签名如下：

```
int findRotateSteps(String ring, String key);
```

比如题目举的例子，输入 `ring = "godding"`, `key = "gd"`，对应的圆盘如下（大写只是为了清晰，实际上输入的字符串都是小写字母）：



我们需要输入 `key = "gd"`，算法返回 4。

因为现在指针指的字母就是字母 "`g`"，所以可以直接按下中间的按钮，然后再将圆盘逆时针拨动两格，让指针指向字母 "`d`"，然后再按一次中间的按钮。

上述过程，按了两次按钮，拨了两格转盘，总共操作了 4 次，是最少的操作次数，所以算法应该返回 4。

我们这里可以首先给题目做一个等价，转动圆盘是不是就等于拨动指针？

原题可以转化为：圆盘固定，我们可以拨动指针；现在需要我们拨动指针并按下按钮，以最少的操作次数输入 **key** 对应的字符串。

那么，这个问题如何使用动态规划的技巧解决呢？或者说，这道题的「状态」和「选择」是什么呢？

「状态」就是「当前需要输入的字符」和「当前圆盘指针的位置」。

再具体点，「状态」就是 **i** 和 **j** 两个变量。我们可以用 **i** 表示当前圆盘上指针指向的字符（也就是 **ring[i]**）；用 **j** 表示需要输入的字符（也就是 **key[j]**）。

这样我们可以写这样一个 **dp** 函数：

```
int dp(String ring, int i, String key, int j);
```

这个 **dp** 函数的定义如下：

当圆盘指针指向 **ring[i]** 时，输入字符串 **key[j...]** 至少需要 **dp(ring, i, key, j)** 次操作。

根据这个定义，题目其实就是想计算 **dp(ring, 0, key, 0)** 的值，而且我们可以把 **dp** 函数的 base case 写出来：

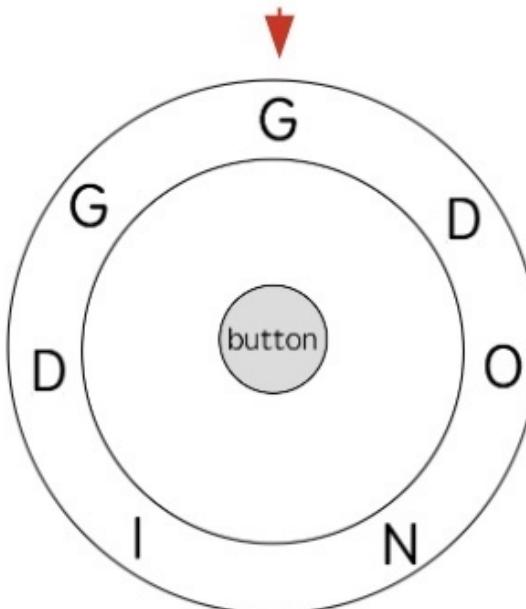
```
int dp(String ring, int i, String key, int j) {  
    // base case, 完成输入  
    if (j == key.length()) return 0;  
    // ...  
}
```

接下来，思考一下如何根据状态做选择，如何进行状态转移？

「选择」就是「如何拨动指针得到待输入的字符」。

再具体点就是，对于现在想输入的字符 **key[j]**，我们可以如何拨动圆盘，得到这个字符？

比如说输入 **ring = "gdonidg"**，现在圆盘的状态如下图：



假设我想输入的字符 `key[j] = "d"`, 圆盘中有两个字母 "`d`", 而且我可以顺时针也可以逆时针拨动指针, 所以总共有四种「选择」输入字符 "`d`", 我们需要选择操作次数最少的那个拨法。

大致的代码逻辑如下:

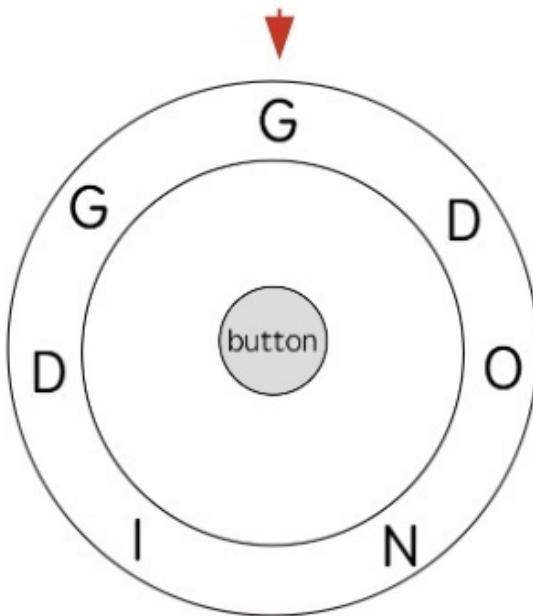
```
int dp(String ring, int i, String key, int j) {
    // base case 完成输入
    if (j == key.length()) return 0;

    // 做选择
    int res = Integer.MAX_VALUE;
    for (int k : [字符 key[j] 在 ring 中的所有索引]) {
        res = min(
            把 i 顺时针转到 k 的代价,
            把 i 逆时针转到 k 的代价
        );
    }

    return res;
}
```

至于到底是顺时针还是逆时针, 其实非常好判断, 怎么近就怎么来; 但是对于圆盘中的两个字符 "`d`", 还能是怎么近怎么来吗?

不能, 因为这和 `key[i]` 之后需要输入的字符有关, 还是上面的例子:



如果输入的是 `key = "di"`, 那么即便右边的 "`d`" 离得近, 也应该去左边的 "`d`", 因为左边的 "`d`" 旁边就是 "`i`", 「整体」的操作数最少。

那么, 应该如何判断呢? 其实就是穷举, 递归调用 `dp` 函数, 把两种选择的「整体」代价算出来, 然后再做比较就行了。

讲到这就差不多了, 直接看最终代码吧:

```
class Solution {
    // 字符 -> 索引列表
    HashMap<Character, List<Integer>> charToIndex = new HashMap<>();
```

```

// 备忘录
int[][] memo;

// 主函数
public int findRotateSteps(String ring, String key) {
    int m = ring.length();
    int n = key.length();
    // 备忘录全部初始化为 0
    memo = new int[m][n];
    // 记录圆环上字符到索引的映射
    for (int i = 0; i < ring.length(); i++) {
        char c = ring.charAt(i);
        if (!charToIndex.containsKey(c)) {
            charToIndex.put(c, new LinkedList<>());
        }
        charToIndex.get(c).add(i);
    }
    // 圆盘指针最初指向 12 点钟方向,
    // 从第一个字符开始输入 key
    return dp(ring, 0, key, 0);
}

// 计算圆盘指针在 ring[i], 输入 key[j..] 的最少操作数
int dp(String ring, int i, String key, int j) {
    // base case 完成输入
    if (j == key.length()) return 0;
    // 查找备忘录, 避免重叠子问题
    if (memo[i][j] != 0) return memo[i][j];

    int n = ring.length();
    // 做选择
    int res = Integer.MAX_VALUE;
    // ring 上可能有多个字符 key[j]
    for (int k : charToIndex.get(key.charAt(j))) {
        // 拨动指针的次数
        int delta = Math.abs(k - i);
        // 选择顺时针还是逆时针
        delta = Math.min(delta, n - delta);
        // 将指针拨到 ring[k], 继续输入 key[j+1..]
        int subProblem = dp(ring, k, key, j + 1);
        // 选择「整体」操作次数最少的
        // 加一是因为按动按钮也是一次操作
        res = Math.min(res, 1 + delta + subProblem);
    }
    // 将结果存入备忘录
    memo[i][j] = res;
    return res;
}
}

```

---

▶  代码可视化动画 

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 旅游省钱大法：加权最短路径



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">787. Cheapest Flights Within K Stops</a>	787. K 站中转内最便宜的航班	困难

阅读本文前，你需要先学习：

- 图结构基础
- 图结构遍历
- 动态规划核心框架

毕业季，对过去也许有些欢乐和感伤，对未来也许有些迷茫和向往，不过这些终究是过眼云烟，迟早会被时间淡化和遗忘。

在这段美好时光的末尾，确实应该来一场说走就走的毕业旅行，放肆一把，给青春画上一个完美的句号。

那么，本文就教给你一个动态规划算法，在毕业旅行中省钱，节约追求诗和远方的资本。

假设，你准备从学校所在的城市出发，游历多个城市，一路浪到公司入职，那么你应该如何安排旅游路线，才能最小化机票的开销？

我们来看看力扣第 787 题「K 站中转内最便宜的航班」：

## ▼ 787. K 站中转内最便宜的航班 [Leetcode](#) | [力扣](#)

有  $n$  个城市通过一些航班连接。给你一个数组  $\text{flights}$ ，其中  $\text{flights}[i] = [\text{from}_i, \text{to}_i, \text{price}_i]$ ，表示该航班都从城市  $\text{from}_i$  开始，以价格  $\text{price}_i$  抵达  $\text{to}_i$ 。

现在给定所有的城市和航班，以及出发城市  $\text{src}$  和目的地  $\text{dst}$ ，你的任务是找到出一条最多经过  $k$  站中转的路线，使得从  $\text{src}$  到  $\text{dst}$  的 **价格最便宜**，并返回该价格。如果不存在这样的路线，则输出  $-1$ 。

### 示例 1：

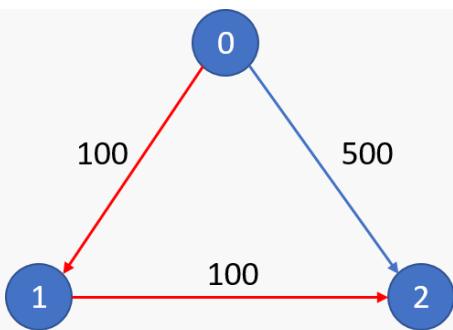
输入：

```
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]  
src = 0, dst = 2, k = 1
```

输出：200

解释：

城市航班图如下



从城市 0 到城市 2 在 1 站中转以内的最便宜价格是 200，如图中红色所示。

## 示例 2：

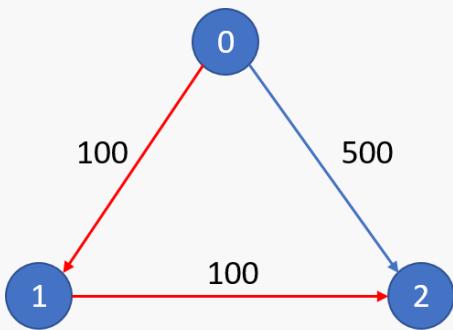
输入：

```
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]  
src = 0, dst = 2, k = 0
```

输出：500

解释：

城市航班图如下



从城市 0 到城市 2 在 0 站中转以内的最便宜价格是 500，如图中蓝色所示。

## 提示：

- $1 \leq n \leq 100$
- $0 \leq \text{flights.length} \leq (n * (n - 1) / 2)$
- $\text{flights}[i].length == 3$
- $0 \leq \text{from}_i, \text{to}_i < n$
- $\text{from}_i \neq \text{to}_i$
- $1 \leq \text{price}_i \leq 10^4$
- 航班没有重复，且不存在自环
- $0 \leq \text{src}, \text{dst}, \text{k} < n$
- $\text{src} \neq \text{dst}$

函数签名如下：

```
int findCheapestPrice(int n, int[][] flights, int src, int dst, int K);
```

很明显，这题就是个加权有向图中求最短路径的问题。

说白了，就是给你一幅加权有向图，让你求 `src` 到 `dst` 权重最小的一条路径，同时要满足，**这条路径最多不能超过 `K + 1` 条边**（经过 `K` 个节点相当于经过 `K + 1` 条边）。

我们来分析下求最短路径相关的算法。

## BFS 算法思路

我们前文 [BFS 算法框架详解](#) 中说到，求最短路径，肯定可以用 BFS 算法来解决。

因为 BFS 算法相当于从起始点开始，一步一步向外扩散，那当然是离起点越近的节点越先被遍历到，如果 BFS 遍历的过程中遇到终点，那么走的肯定是最短路径。

不过呢，我们在 [BFS 算法框架详解](#) 用的是普通的队列 `Queue` 来遍历多叉树，而对于加权图的最短路径来说，普通的队列不管用了，得用优先级队列 `PriorityQueue`。

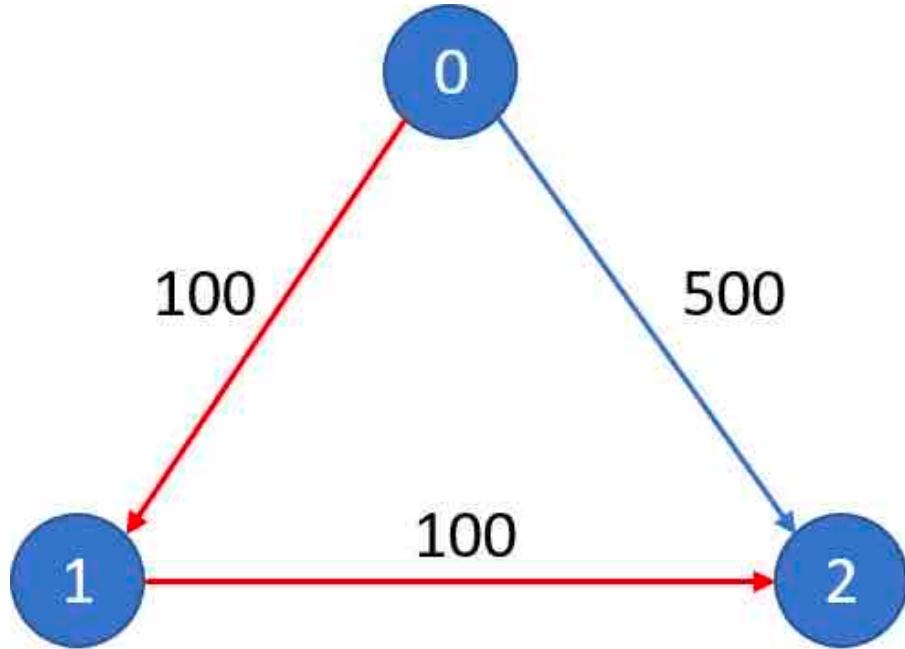
为什么呢？也好理解，在多叉树（或者扩展到无权图）的遍历中，与其说边没有权重，不如说每条边的权重都是 1，起点与终点之间的路径权重就是它们之间「边」的条数。

这样，按照 BFS 算法一步步向四周扩散的逻辑，先遍历到的节点和起点之间的「边」更少，累计的权重当然少。

换言之，先进入 `Queue` 的节点就是离起点近的，路径权重小的节点。

但对于加权图，路径中边的条数和路径的权重并不是正相关的关系了，有的路径可能边的条数很少，但每条边的权重都很大，那显然这条路径权重也会很大，很难成为最短路径。

比如题目给的这个例子：



你是可以一步从 `0` 走到 `2`，但路径权重不见得是最小的。

所以，对于加权图的场景，我们需要优先级队列「自动排序」的特性，将路径权重较小的节点排在队列前面，以此为基础施展 BFS 算法，也就变成了 [Dijkstra 算法](#)。

说了这么多 BFS 算法思路，只是帮助大家融会贯通一下，我们本文准备主要讲解如何使用动态规划来解决这道题。关于 Dijkstra 算法的实现代码，文末有写，供读者参考。

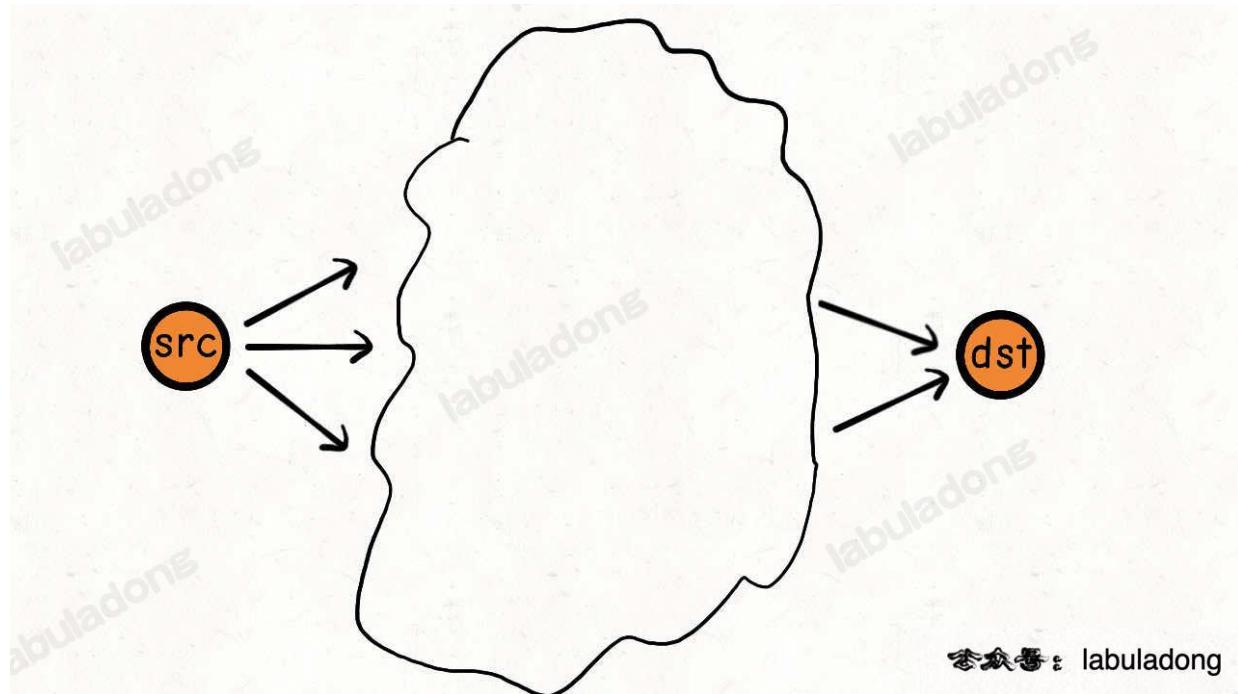
## 动态规划思路

我们前文 [动态规划核心套路详解](#) 中说过，求最值的问题，很多都可能使用动态规划来求解。

加权最短路径问题，再加个 **K** 的限制也无妨，不也是个求最值的问题嘛，动态规划统统拿下。

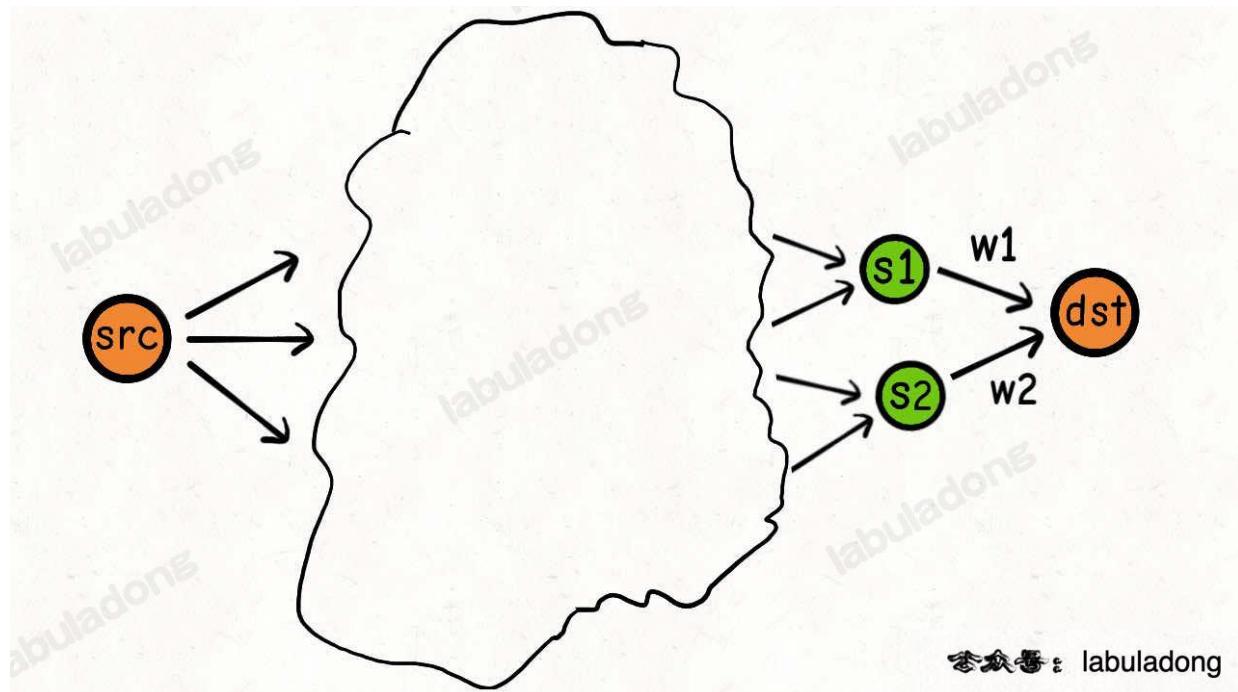
我们先不管 **K** 的限制，单就「加权最短路径」这个问题来看看，它怎么就是个动态规划问题了呢？

比方说，现在我想计算 **src** 到 **dst** 的最短路径：



最小权重是多少？我不知道。

但我可以把问题进行分解：



**s1, s2** 是指向 **dst** 的相邻节点，它们之间的权重我是知道的，分别是 **w1, w2**。

只要我知道了从 **src** 到 **s1, s2** 的最短路径，我不就知道 **src** 到 **dst** 的最短路径了吗！

```
minPath(src, dst) = min(  
    minPath(src, s1) + w1,
```

```
    minPath(src, s2) + w2  
)
```

这其实就是递归关系了，就是这么简单。

不过别忘了，题目对我们的最短路径还有个「路径上不能超过  $K + 1$  条边」的限制。

那么我们不妨定义这样一个  $dp$  函数：

```
int dp(int s, int k);
```

函数的定义如下：

从起点  $src$  出发， $k$  步之内（一步就是一条边）到达节点  $s$  的最小路径权重为  $dp(s, k)$ 。

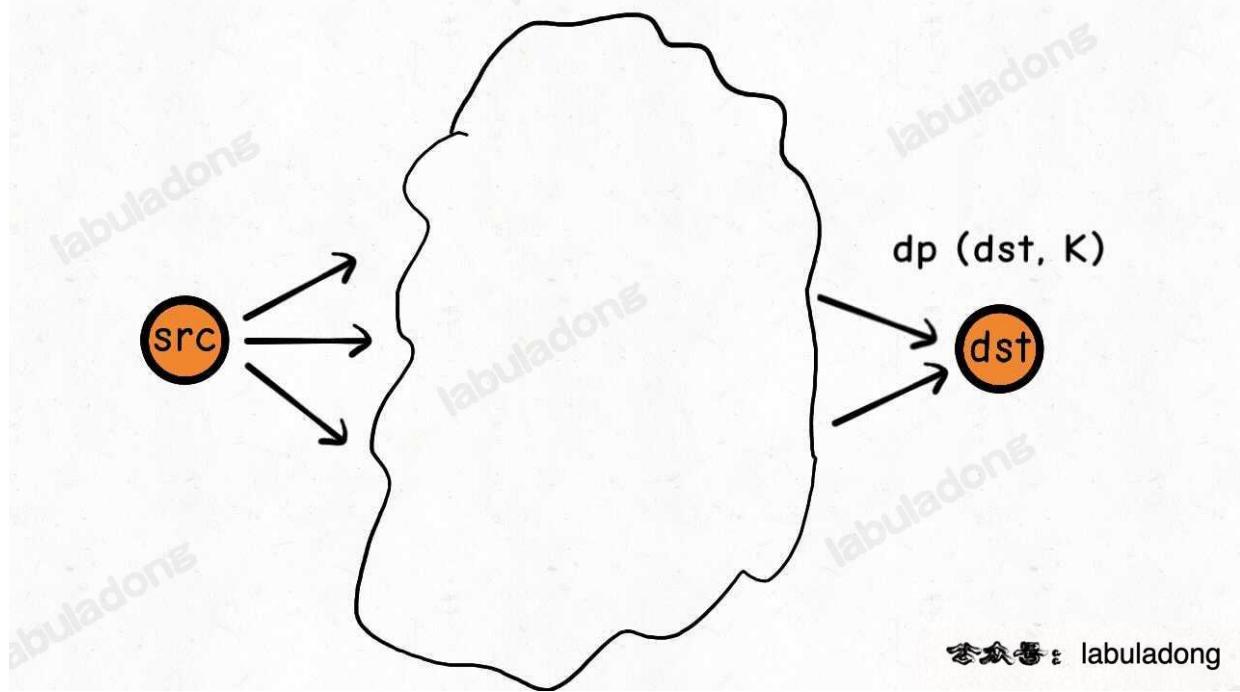
那么， $dp$  函数的 base case 就显而易见了：

```
// 定义：从 src 出发，k 步之内到达 s 的最小成本  
int dp(int s, int k) {  
    // 从 src 到 src, 一步都不用走  
    if (s == src) {  
        return 0;  
    }  
    // 如果步数用尽，就无解了  
    if (k == 0) {  
        return -1;  
    }  
  
    // ...  
}
```

题目想求的最小机票开销就可以用  $dp(dst, K+1)$  来表示：

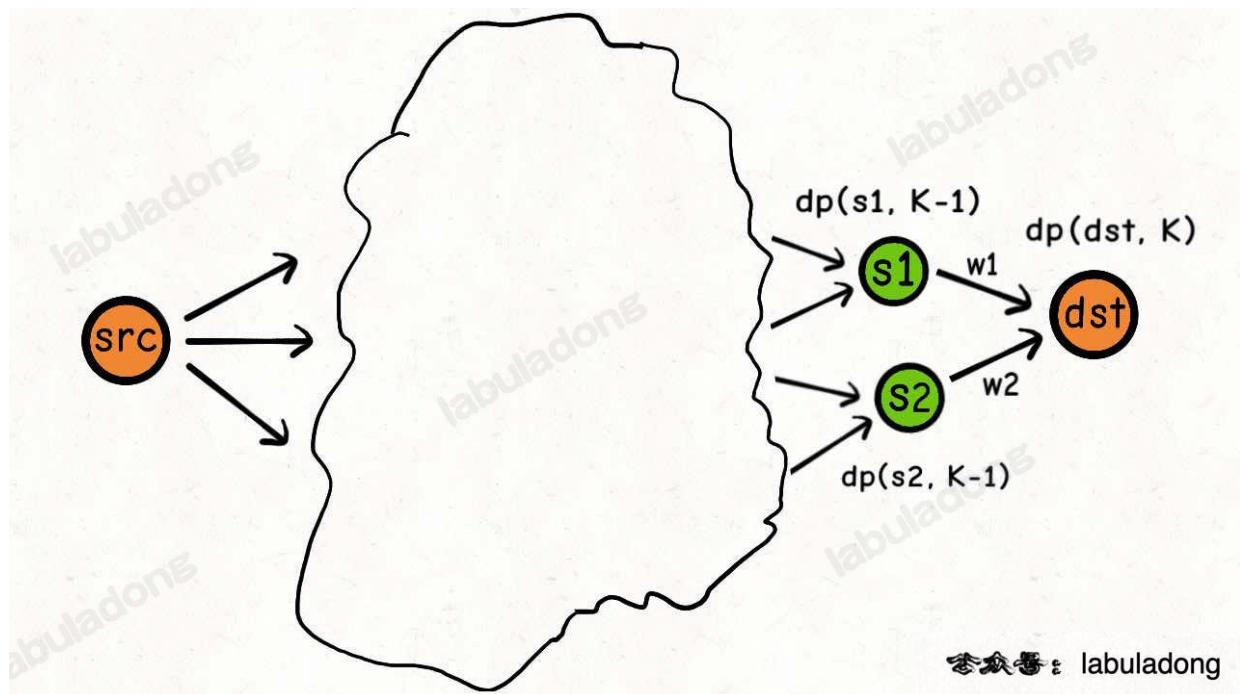
```
int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {  
    // 将中转站个数转化成边的条数  
    K++;  
    // ...  
    return dp(dst, K);  
}
```

添加了一个  $K$  条边的限制，状态转移方程怎么写呢？其实和刚才是一样的：



K 步之内从  $\text{src}$  到  $\text{dst}$  的最小路径权重是多少？我不知道。

但我可以把问题分解：



$s_1, s_2$  是指向  $\text{dst}$  的相邻节点，我只要知道  $K - 1$  步之内从  $\text{src}$  到达  $s_1, s_2$ ，那我就可以在  $K$  步之内从  $\text{src}$  到达  $\text{dst}$ 。

也就是如下关系式：

$$\begin{aligned} \text{dp}(\text{dst}, k) = \min( & \\ & \text{dp}(s_1, k - 1) + w_1, \\ & \text{dp}(s_2, k - 1) + w_2 \\ ) \end{aligned}$$

这就是新的状态转移方程，如果你能看懂这个算式，就已经可以解决这道题了。

## 代码实现

根据上述思路，我怎么知道 `s1`, `s2` 是指向 `dst` 的相邻节点，他们之间的权重是 `w1`, `w2`?

我希望给一个节点，就能知道有谁指向这个节点，还知道它们之间的权重，对吧。

专业点说，得用一个数据结构记录每个节点的「入度」`indegree`，即存储所有指向该节点的相邻节点，以及它们之间边的权重。

具体看代码吧，我们用一个哈希表 `indegree` 存储入度，然后实现 `dp` 函数：

```
class Solution {
    // 哈希表记录每个点的入度，键是节点编号，值是指向该节点的相邻节点以及之间的权重
    // to -> [from, price]
    HashMap<Integer, List<int[]>> indegree;
    int src, dst;

    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
        // 将中转站个数转化成边的条数
        K++;
        this.src = src;
        this.dst = dst;

        indegree = new HashMap<>();
        for (int[] f : flights) {
            int from = f[0];
            int to = f[1];
            int price = f[2];
            // 记录谁指向该节点，以及之间的权重
            indegree.putIfAbsent(to, new LinkedList<>());
            indegree.get(to).add(new int[] {from, price});
        }

        return dp(dst, K);
    }

    // 定义：从 src 出发，k 步之内到达 s 的最短路径权重
    int dp(int s, int k) {
        // base case
        if (s == src) {
            return 0;
        }
        if (k == 0) {
            return -1;
        }
        // 初始化为最大值，方便等会取最小值
        int res = Integer.MAX_VALUE;
        if (indegree.containsKey(s)) {
            // 当 s 有入度节点时，分解为子问题
            for (int[] v : indegree.get(s)) {
                int from = v[0];
                int price = v[1];
                // 从 src 到达相邻的入度节点所需的最短路径权重
                int subProblem = dp(from, k - 1);
                // 跳过无解的情况
                if (subProblem != -1) {
                    res = Math.min(res, subProblem + price);
                }
            }
        }
        return res;
    }
}
```

```
        }
    }
    // 如果还是初始值，说明此节点不可达
    return res == Integer.MAX_VALUE ? -1 : res;
}
}
```

有之前的铺垫，这段解法逻辑应该是很清晰的。当然，对于动态规划问题，肯定要消除重叠子问题。

为什么有重叠子问题？很简单，如果某个节点同时指向两个其他节点，那么这两个节点就有相同的一个入度节点，就会产生重复的递归计算。

怎么消除重叠子问题？找问题的「状态」。

状态是什么？在问题分解（状态转移）的过程中变化的，就是状态。

谁在变化？显然就是 `dp` 函数的参数 `s` 和 `k`，每次递归调用，目标点 `s` 和步数约束 `k` 在变化。

所以，本题的状态有两个，应该算是二维动态规划，我们可以用一个 `memo` 二维数组或者哈希表作为备忘录，减少重复计算。

我们选用二维数组做备忘录吧，注意 `K` 是从 1 开始算的，所以备忘录初始大小要再加一：

```
class Solution {
    int src, dst;
    HashMap<Integer, List<int[]>> indegree;
    // 备忘录
    int[][] memo;

    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
        K++;
        this.src = src;
        this.dst = dst;
        // 初始化备忘录，全部填一个特殊值
        memo = new int[n][K + 1];
        for (int[] row : memo) {
            Arrays.fill(row, -888);
        }

        indegree = new HashMap<>();
        for (int[] f : flights) {
            int from = f[0];
            int to = f[1];
            int price = f[2];
            indegree.putIfAbsent(to, new LinkedList<>());
            indegree.get(to).add(new int[] {from, price});
        }
    }

    return dp(dst, K);
}

// 定义：从 src 出发，k 步之内到达 s 的最小成本
int dp(int s, int k) {
    // base case
    if (s == src) {
        return 0;
    }
```

```

    if (k == 0) {
        return -1;
    }
    // 查备忘录，防止冗余计算
    if (memo[s][k] != -888) {
        return memo[s][k];
    }

    // 状态转移代码不变
    int res = Integer.MAX_VALUE;
    if (indegree.containsKey(s)) {
        for (int[] v : indegree.get(s)) {
            int from = v[0];
            int price = v[1];
            int subProblem = dp(from, k - 1);
            if (subProblem != -1) {
                res = Math.min(res, subProblem + price);
            }
        }
    }
    // 存入备忘录
    memo[s][k] = res == Integer.MAX_VALUE ? -1 : res;
    return memo[s][k];
}
}

```

## ▶ 😊 代码可视化动画😊

备忘录初始值为啥初始为 -888? 前文 [base case 和备忘录的初始值怎么定](#) 说过, 随便初始化一个无意义的值就行。

至此, 这道题就通过自顶向下的递归方式解决了。当然, 完全可以按照这个解法衍生出自底向上迭代的动态规划解法, 但由于篇幅所限, 我就不写了, 反正本质上都是一样的。

其实, 大家如果把之前的所有动态规划文章都看一遍, 就会发现我们一直在套用 [动态规划核心套路](#), 其实真没什么困难的。

最后扩展一下, 有的读者可能会问: 既然这个问题本质上是一个图的遍历问题, 为什么不需要 [visited](#) 集合记录已经访问过的节点?

这个问题我在 [Dijkstra 算法模板](#) 中探讨过, 可以去看看。另外, 这题也可以利用 Dijkstra 算法模板来解决, 代码如下:

```

class Solution {
    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
        List<int[]>[] graph = new LinkedList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new LinkedList<>();
        }
        for (int[] edge : flights) {
            int from = edge[0];
            int to = edge[1];
            int price = edge[2];
            graph[from].add(new int[]{to, price});
        }

        // 启动 dijkstra 算法
        // 计算以 src 为起点在 k 次中转到达 dst 的最短路径
    }
}

```

```

        K++;
        return dijkstra(graph, src, K, dst);
    }

    class State {
        // 图节点的 id
        int id;
        // 从 src 节点到当前节点的花费
        int costFromSrc;
        // 从 src 节点到当前节点经过的节点个数
        int nodeNumFromSrc;

        State(int id, int costFromSrc, int nodeNumFromSrc) {
            this.id = id;
            this.costFromSrc = costFromSrc;
            this.nodeNumFromSrc = nodeNumFromSrc;
        }
    }

    // 输入一个起点 src, 计算从 src 到其他节点的最短距离
    int dijkstra(List<int[][]> graph, int src, int k, int dst) {
        // 定义: 从起点 src 到达节点 i 的最短路径权重为 distTo[i]
        int[] distTo = new int[graph.length];
        // 定义: 从起点 src 到达节点 i 的最小权重路径至少要经过 nodeNumTo[i] 个节点
        int[] nodeNumTo = new int[graph.length];
        Arrays.fill(distTo, Integer.MAX_VALUE);
        Arrays.fill(nodeNumTo, Integer.MAX_VALUE);
        // base case
        distTo[src] = 0;
        nodeNumTo[src] = 0;

        // 优先级队列, costFromSrc 较小的排在前面
        Queue<State> pq = new PriorityQueue<>((a, b) -> {
            return a.costFromSrc - b.costFromSrc;
        });
        // 从起点 src 开始进行 BFS
        pq.offer(new State(src, 0, 0));

        while (!pq.isEmpty()) {
            State curState = pq.poll();
            int curNodeID = curState.id;
            int costFromSrc = curState.costFromSrc;
            int curNodeNumFromSrc = curState.nodeNumFromSrc;

            if (curNodeID == dst) {
                // 找到最短路径
                return costFromSrc;
            }
            if (curNodeNumFromSrc == k) {
                // 中转次数耗尽
                continue;
            }

            // 将 curNode 的相邻节点装入队列
            for (int[] neighbor : graph[curNodeID]) {
                int nextNodeID = neighbor[0];
                int costToNextNode = costFromSrc + neighbor[1];
                // 中转次数消耗 1
                int nextNodeNumFromSrc = curNodeNumFromSrc + 1;
                pq.offer(new State(nextNodeID, costToNextNode, nextNodeNumFromSrc));
            }
        }
    }
}

```

```
// 更新 dp table
if (distTo[nextNodeID] > costToNextNode) {
    distTo[nextNodeID] = costToNextNode;
    nodeNumTo[nextNodeID] = nextNodeNumFromSrc;
}
// 剪枝，如果中转次数更多，花费还更大，那必然不会是最短路径
if (costToNextNode > distTo[nextNodeID]
    && nextNodeNumFromSrc > nodeNumTo[nextNodeID]) {
    continue;
}

pq.offer(new State(nextNodeID, costToNextNode,
nextNodeNumFromSrc));
}
}
return -1;
}
}
```

关于这个解法这里就不多解释了，可对照前文 [Dijkstra 算法模板](#) 理解。

### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">286. Walls and Gates</a> 	<a href="#">286. 墙与门</a> 	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 经典动态规划：正则表达式



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">10. Regular Expression Matching</a>	10. 正则表达式匹配	困难

阅读本文前，你需要先学习：

- 动态规划核心框架
- 经典动态规划：编辑距离

正则表达式是一个非常强力的工具，本文就来具体看一看正则表达式的底层原理是什么。力扣第 10 题「正则表达式匹配」就要求我们实现一个简单的正则匹配算法，包括「.」通配符和「\*」通配符。

这两个通配符是最常用的，其中点号「.」可以匹配任意一个字符，星号「\*」可以让之前的那个字符重复任意次数（包括 0 次）。

比如说模式串 "`.a*b`" 就可以匹配文本 "zaaab"，也可以匹配 "cb"；模式串 "`a..b`" 可以匹配文本 "amnb"；而模式串 "`.*`" 就比较牛逼了，它可以匹配任何文本。

题目会给我们输入两个字符串 `s` 和 `p`，`s` 代表文本，`p` 代表模式串，请你判断模式串 `p` 是否可以匹配文本 `s`。我们可以假设模式串只包含小写字母和上述两种通配符且一定合法，不会出现 `*a` 或者 `b**` 这种不合法的模式串，

函数签名如下：

```
boolean isMatch(string s, string p);
```

对于我们将来要实现的这个正则表达式，难点在那里呢？

点号通配符其实很好实现，`s` 中的任何字符，只要遇到 `.` 通配符，无脑匹配就完事了。主要是这个星号通配符不好实现，一旦遇到 `*` 通配符，前面的那个字符可以选择重复一次，可以重复多次，也可以一次都不出现，这该怎么办？

对于这个问题，答案很简单，对于所有可能出现的情况，全部穷举一遍，只要有一种情况可以完成匹配，就认为 `p` 可以匹配 `s`。那么一旦涉及两个字符串的穷举，我们就应该条件反射地想到动态规划的技巧了。

## 一、思路分析

我们先脑补一下，`s` 和 `p` 相互匹配的过程大致是，两个指针 `i`, `j` 分别在 `s` 和 `p` 上移动，如果最后两个指针都能移动到字符串的末尾，那么久匹配成功，反之则匹配失败。

如果不考虑 \* 通配符，面对两个待匹配字符 s[i] 和 p[j]，我们唯一能做的就是看他俩是否匹配：

```
boolean isMatch(String s, String p) {
    int i = 0, j = 0;
    while (i < s.length() && j < p.length()) {
        // 「.」通配符就是万金油
        if (s.charAt(i) == p.charAt(j) || p.charAt(j) == '.') {
            // 匹配，接着匹配 s[i+1..] 和 p[j+1..]
            i++; j++;
        } else {
            // 不匹配
            return false;
        }
    }
    return i == j;
}
```

那么考虑一下，如果加入 \* 通配符，局面就会稍微复杂一些，不过只要分情况来分析，也不难理解。

当 p[j + 1] 为 \* 通配符时，我们分情况讨论下：

1、如果 s[i] == p[j]，那么有两种情况：

1.1 p[j] 有可能会匹配多个字符，比如 s = "aaa"，p = "a\*", 那么 p[0] 会通过 \* 匹配 3 个字符 "a"。

1.2 p[i] 也有可能匹配 0 个字符，比如 s = "aa"，p = "a\*aa"，由于后面的字符可以匹配 s，所以 p[0] 只能匹配 0 次。

2、如果 s[i] != p[j]，只有一种情况：

p[j] 只能匹配 0 次，然后看下一个字符是否能和 s[i] 匹配。比如说 s = "aa"，p = "b\*aa"，此时 p[0] 只能匹配 0 次。

综上，可以把之前的代码针对 \* 通配符进行一下改造：

```
if (s.charAt(i) == p.charAt(j) || p.charAt(j) == '.') {
    // 匹配
    if (j < p.length() - 1 && p.charAt(j + 1) == '*') {
        // 有 * 通配符，可以匹配 0 次或多次
    } else {
        // 无 * 通配符，老老实实匹配 1 次
        i++; j++;
    }
} else {
    // 不匹配
    if (j < p.length() - 1 && p.charAt(j + 1) == '*') {
        // 有 * 通配符，只能匹配 0 次
    } else {
        // 无 * 通配符，匹配无法进行下去了
        return false;
    }
}
```

整体的思路已经很清晰了，但现在的问题是，遇到 \* 通配符时，到底应该匹配 0 次还是匹配多次？多次是几次？

你看，这就是一个做「选择」的问题，要把所有可能的选择都穷举一遍才能得出结果。动态规划算法的核心就是「状态」和「选择」，「状态」无非就是 **i** 和 **j** 两个指针的位置，「选择」就是 **p[j]** 选择匹配几个字符。

## 二、动态规划解法

根据「状态」，我们可以定义一个 **dp** 函数：

```
boolean dp(String s, int i, String p, int j);
```

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 经典动态规划：高楼扔鸡蛋



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">887. Super Egg Drop</a>	<a href="#">887. 鸡蛋掉落</a>	困难

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

本文要聊一个很经典的算法问题，若干层楼，若干个鸡蛋，让你算出最少的尝试次数，找到鸡蛋恰好摔不碎的那层楼。国内大厂以及谷歌脸书面试都经常考察这道题，只不过他们觉得扔鸡蛋太浪费，改成扔杯子，扔破碗什么的。

具体的问题等会再说，但是这道题的解法技巧很多，光动态规划就好几种效率不同的思路，最后还有一种极其高效数学解法。秉承本书一贯的作风，拒绝过于诡异的技巧，因为这些技巧无法举一反三，学了也不划算。

下面就来用我们一直强调的动态规划通用思路来研究一下这道题。

## 一、解析题目

这是力扣第 887 题「鸡蛋掉落」，我描述一下题目：

你面前有一栋从 1 到  $N$  共  $N$  层的楼，然后给你  $K$  个鸡蛋 ( $K$  至少为 1)。现在确定这栋楼存在楼层  $0 \leq F \leq N$ ，在这层楼将鸡蛋扔下去，鸡蛋恰好没摔碎（高于  $F$  的楼层都会碎，低于  $F$  的楼层都不会碎，如果鸡蛋没有碎，可以捡回来继续扔）。现在问你，最坏情况下，你至少要扔几次鸡蛋，才能确定这个楼层  $F$  呢？

也就是让你找摔不碎鸡蛋的最高楼层  $F$ ，但什么叫「最坏情况」下「至少」要扔几次呢？我们分别举个例子就明白了。

比方说现在先不管鸡蛋个数的限制，有 7 层楼，你怎么去找鸡蛋恰好摔碎的那层楼？

最原始的方式就是线性扫描：我先在 1 楼扔一下，没碎，我再去 2 楼扔一下，没碎，我再去 3 楼……

以这种策略，最坏情况应该就是我试到第 7 层鸡蛋也没碎 ( $F = 7$ )，也就是我扔了 7 次鸡蛋。

先在你应该理解什么叫做「最坏情况」下了，鸡蛋破碎一定发生在搜索区间穷尽时，不会说你在第 1 层摔一下鸡蛋就碎了，这是你运气好，不是最坏情况。

现在再来理解一下什么叫「至少」要扔几次。依然不考虑鸡蛋个数限制，同样是 7 层楼，我们可以优化策略。

最好的策略是使用二分查找思路，我先去第  $(1 + 7) / 2 = 4$  层扔一下：

如果碎了说明  $F$  小于 4，我就去第  $(1 + 3) / 2 = 2$  层试……

如果没碎说明  $F$  大于等于 4，我就去第  $(5 + 7) / 2 = 6$  层试……

以这种策略，**最坏情况**应该是试到第 7 层鸡蛋还没碎 ( $F = 7$ )，或者鸡蛋一直碎到第 1 层 ( $F = 0$ )。然而无论那种最坏情况，只需要试  $\lceil \log_2 7 \rceil$  向上取整等于 3 次，比刚才尝试 7 次要少，这就是所谓的**至少要扔几次**。

实际上，如果不限制鸡蛋个数的话，二分思路显然可以得到最少尝试的次数，但问题是，**现在给你了鸡蛋个数的限制 K**，直接使用二分思路就不行了。

比如说只给你 1 个鸡蛋，7 层楼，你敢用二分吗？你直接去第 4 层扔一下，如果鸡蛋没碎还好，你可以把鸡蛋捡起来再去更高的楼层尝试；但如果碎了，你就没有鸡蛋继续测试了，无法确定鸡蛋恰好摔不碎的楼层  $F$  了。

其实这种情况下只能用线性扫描的方法，从下往上一层层尝试扔鸡蛋，那么最坏情况下需要扔 7 次，算法返回结果应该是 7。

有的读者也许会有这种想法：二分查找排除楼层的速度无疑是最快的，那干脆先用二分查找，等到只剩 1 个鸡蛋的时候再执行线性扫描，这样得到的结果是不是就是最少的扔鸡蛋次数呢？

很遗憾，并不是，比如说把楼层变高一些，100 层，给你 2 个鸡蛋，你在 50 层扔一下，碎了，那就只能线性扫描 1~49 层了，最坏情况下要扔 50 次。

如果不要「二分」，变成「五分」「十分」都会大幅减少最坏情况下的尝试次数。比方说第一个鸡蛋每隔十层楼扔，在哪里碎了第二个鸡蛋一个个线性扫描，总共不会超过 20 次。最优解其实是 14 次。最优策略非常多，而且并没有什么规律可言。

说了这么多废话，就是确保大家理解了题目的意思，而且认识到这个题目确实复杂，就连我们手算都不容易，如何用算法解决呢？

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 经典动态规划：戳气球



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">312. Burst Balloons</a>	<a href="#">312. 戳气球</a>	困难

-----

阅读本文前，你需要先学习：

- 动态规划核心框架

今天我们要聊的这道题「Burst Balloon」和之前我们写过的那篇 [经典动态规划：高楼扔鸡蛋问题](#) 分析过的高楼扔鸡蛋问题类似，知名度比较高，但难度确实也不小。

它是力扣第 312 题「戳气球」，题目如下：

### ▼ 312. 戳气球 Leetcode | 力扣

有  $n$  个气球，编号为  $0$  到  $n - 1$ ，每个气球上都标有一个数字，这些数字存在数组  $\text{nums}$  中。

现在要求你戳破所有的气球。戳破第  $i$  个气球，你可以获得  $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$  枚硬币。这里的  $i - 1$  和  $i + 1$  代表和  $i$  相邻的两个气球的序号。如果  $i - 1$  或  $i + 1$  超出了数组的边界，那么就当它是一个数字为  $1$  的气球。

求所能获得硬币的最大数量。

#### 示例 1：

```
输入: nums = [3,1,5,8]
输出: 167
解释:
  nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
  coins = 3*1*5      +      3*5*8      +      1*3*8      + 1*8*1 = 167
```

#### 示例 2：

```
输入: nums = [1,5]
输出: 10
```

#### 提示：

- `n == nums.length`
- `1 <= n <= 300`
- `0 <= nums[i] <= 100`

首先必须要说明，这个题目的状态转移方程真的比较巧妙，所以说如果你看了题目之后完全没有思路恰恰是正常的。虽然最优答案不容易想出来，但基本的思路分析是我们应该力求做到的。所以本文会先分析一下常规思路，然后再引入动态规划解法。

## 一、回溯思路

先来梳理一下解决这种问题的套路：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 经典动态规划：博弈问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">877. Stone Game</a>	<a href="#">877. 石子游戏</a>	简单
<a href="#">486. Predict the Winner</a>	<a href="#">486. 预测赢家</a>	困难

-----

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

上一篇文章 [几道智力题](#) 中讨论到一个有趣的「石头游戏」，通过题目的限制条件，这个游戏是先手必胜的。但是智力题终究是智力题，真正的算法问题肯定不会是投机取巧能搞定的。所以，本文就借石头游戏来讲讲「假设两个人都足够聪明，最后谁会获胜」这一类问题该如何用动态规划算法解决。

博弈类问题的套路都差不多，下文参考 [这个 YouTube 视频](#) 的思路讲解，其核心思路是在二维 dp 的基础上使用元组分别存储两个人的博弈结果。掌握了这个技巧以后，别人再问你什么俩海盗分宝石，俩人拿硬币的问题，你就告诉别人：我懒得想，直接给你写个算法算一下得了。

我们把力扣第 877 题「石头游戏」改的更具有一般性：

你和你的朋友面前有一排石头堆，用一个数组 `piles` 表示，`piles[i]` 表示第 `i` 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。

石头的堆数可以是任意正整数，石头的总数也可以是任意正整数，这样就能打破先手必胜的局面了。比如有三堆石头 `piles = [1, 100, 3]`，先手不管拿 1 还是 3，能够决定胜负的 100 都会被后手拿走，后手会获胜。

假设两人都很聪明，请你写一个 `stoneGame` 函数，返回先手和后手的最后得分（石头总数）之差。比如上面那个例子，先手能获得 4 分，后手会获得 100 分，你的算法应该返回 -96：

```
int stoneGame(int[] nums);
```

这样推广之后就变成了一道难度比较高的动态规划问题了，力扣第 486 题「预测赢家」就是一道类似的问题：

## ▼ 486. 预测赢家 Leetcode | 力扣

给你一个整数数组 `nums`。玩家 1 和玩家 2 基于这个数组设计了一个游戏。

玩家 1 和玩家 2 轮流进行自己的回合，玩家 1 先手。开始时，两个玩家的初始分值都是 0。每一回合，玩家从数组的任意一端取一个数字（即，`nums[0]` 或 `nums[nums.length - 1]`），取到的数字将会从数组中移除（数组长度减 1）。玩

家选中的数字将会加到他的得分上。当数组中没有剩余数字可取时，游戏结束。

如果玩家 1 能成为赢家，返回 `true`。如果两个玩家得分相等，同样认为玩家 1 是游戏的赢家，也返回 `true`。你可以假设每个玩家的玩法都会使他的分数最大化。

### 示例 1：

输入: `nums = [1, 5, 2]`

输出: `false`

解释: 一开始，玩家 1 可以从 1 和 2 中进行选择。

如果他选择 2 (或者 1)，那么玩家 2 可以从 1 (或者 2) 和 5 中进行选择。如果玩家 2 选择了 5，那么玩家 1 则只剩下 1 (或者 2) 可选。

所以，玩家 1 的最终分为  $1 + 2 = 3$ ，而玩家 2 为 5。

因此，玩家 1 永远不会成为赢家，返回 `false`。

### 示例 2：

输入: `nums = [1, 5, 233, 7]`

输出: `true`

解释: 玩家 1 一开始选择 1。然后玩家 2 必须从 5 和 7 中进行选择。无论玩家 2 选择了哪个，玩家 1 都可以选择 233。

最终，玩家 1 (234 分) 比玩家 2 (12 分) 获得更多的分数，所以返回 `true`，表示玩家 1 可以成为赢家。

### 提示：

- `1 <= nums.length <= 20`
- `0 <= nums[i] <= 107`

函数签名如下：

```
boolean predictTheWinner(int[] nums);
```

那么如果有了一个计算先手和后手分差的 `stoneGame` 函数，这道题的解法就直接出来了：

```
public boolean predictTheWinner(int[] nums) {
    // 先手的分数大于等于后手，则能赢
    return stoneGame(nums) >= 0;
}
```

这个 `stoneGame` 函数怎么写呢？博弈问题的难点在于，两个人要轮流进行选择，而且都贼精明，应该如何编程表示这个过程呢？其实不难，还是按照 动态规划核心框架 中强调多次的套路，首先明确 `dp` 数组的含义，然后只要找到「状态」和「选择」，一切就水到渠成了。

## 一、定义 `dp` 数组的含义

定义 `dp` 数组的含义是很有技术含量的，同一问题可能有多种定义方法，不同的定义会引出不同的状态转移方程，不过只要逻辑没有问题，最终都能得到相同的答案。

我建议不要迷恋那些看起来很牛逼，代码很短小的解法思路，最好是稳一点，采取可解释性最好，最容易推广的解法思路。本文就给出一种博弈问题的通用设计框架。

介绍 `dp` 数组的含义之前，我们先看一下 `dp` 数组最终的样子：

**piles = [2, 8, 3, 5]**

end start \ end	0	1	2	3
0	(2, 0)	(8, 2)	(5, 8)	(13, 5)
1		(8, 0)	(8, 3)	(11, 5)
2			(3, 0)	(5, 3)
3				(5, 0)

下文讲解时，认为元组是包含 `first` 和 `second` 属性的一个类，而且为了节省篇幅，将这两个属性简写为 `fir` 和 `sec`。比如按上图的数据，我们说 `dp[1][3].fir = 11, dp[0][1].sec = 2`。

先回答几个读者可能提出的问题：

这个二维 `dp` table 中存储的是元组，怎么编程表示呢？这个 `dp` table 有一半根本没用上，怎么优化？很简单，都不要管，先把解题的思路想明白了再谈也不迟。

**以下是对 `dp` 数组含义的解释：**

`dp[i][j].fir = x` 表示，对于 `piles[i...j]` 这部分石头堆，先手能获得的最高分数为 `x`。

`dp[i][j].sec = y` 表示，对于 `piles[i...j]` 这部分石头堆，后手能获得的最高分数为 `y`。

举例理解一下，假设 `piles = [2, 8, 3, 5]`，索引从 0 开始，那么：

`dp[0][1].fir = 8` 意味着：面对石头堆 `[2, 8]`，先手最多能够获得 8 分；`dp[1][3].sec = 5` 意味着：面对石头堆 `[8, 3, 5]`，后手最多能够获得 5 分。

我们想求的答案是先手和后手最终分数之差，按照这个定义也就是 `dp[0][n-1].fir - dp[0][n-1].sec`，即面对整个 `piles`，先手的最优得分和后手的最优得分之差。

## 二、状态转移方程

写状态转移方程很简单，首先要找到所有「状态」和每个状态可以做的「选择」，然后择优。

根据前面对 `dp` 数组的定义，状态显然有三个：开始的索引 `i`，结束的索引 `j`，当前轮到的人。

```
dp[i][j][fir or sec]
其中：
```

```
0 <= i < piles.length
i <= j < piles.length
```

对于这个问题的每个状态，可以做的选择有两个：选择最左边的那堆石头，或者选择最右边的那堆石头。我们可以这样穷举所有状态：

```
n = piles.length
for 0 <= i < n:
    for j <= i < n:
        for who in {fir, sec}:
            dp[i][j][who] = max(left, right)
```

上面的伪码是动态规划的一个大致的框架，这道题的难点在于，两人足够聪明，而且是交替进行选择的，也就是说先手的选择会对后手有影响，这怎么表达出来呢？

根据我们对 `dp` 数组的定义，很容易解决这个难点，写出状态转移方程：

```
dp[i][j].fir = max(piles[i] + dp[i+1][j].sec, piles[j] + dp[i][j-1].sec)
dp[i][j].fir = max(      选择最左边的石头堆      ,      选择最右边的石头堆      )
# 解释：我作为先手，面对 piles[i...j] 时，有两种选择：

# 要么我选择最左边的那一堆石头 piles[i]，局面变成了 piles[i+1...j]，
# 然后轮到对方选了，我变成了后手，此时我作为后手的最优得分是 dp[i+1][j].sec

# 要么我选择最右边的那一堆石头 piles[j]，局面变成了 piles[i...j-1]
# 然后轮到对方选了，我变成了后手，此时我作为后手的最优得分是 dp[i][j-1].sec

if 先手选择左边：
    dp[i][j].sec = dp[i+1][j].fir
if 先手选择右边：
    dp[i][j].sec = dp[i][j-1].fir
# 解释：我作为后手，要等先手先选择，有两种情况：

# 如果先手选择了最左边那堆，给我剩下了 piles[i+1...j]
# 此时轮到我，我变成了先手，此时的最优得分是 dp[i+1][j].fir

# 如果先手选择了最右边那堆，给我剩下了 piles[i...j-1]
# 此时轮到我，我变成了先手，此时的最优得分是 dp[i][j-1].fir
```

根据 `dp` 数组的定义，我们也可以找出 **base case**，也就是最简单的情况：

```
dp[i][j].fir = piles[i]
dp[i][j].sec = 0
其中 0 <= i == j < n
# 解释：i 和 j 相等就是说面前只有一堆石头 piles[i]
# 那么显然先手的得分为 piles[i]
# 后手没有石头拿了，得分为 0
```

**piles = [2, 8, 3, 5]**

end start \	0	1	2	3
0	(2, 0)			
1		(8, 0)		
2			(3, 0)	
3				(5, 0)

这里需要注意一点，我们发现 base case 是斜着的，而且我们推算  $dp[i][j]$  时需要用到  $dp[i+1][j]$  和  $dp[i][j-1]$ ：

**piles = [2, 8, 3, 5]**

end start \	0	1	2	3
0	(2, 0)			
1		(8, 0)	(8, 3) → (11, 5)	
2			(3, 0)	(5, 3)
3				(5, 0)

根据前文 动态规划答疑篇 判断  $dp$  数组遍历方向的原则，算法应该倒着遍历  $dp$  数组：

```

for (int i = n - 2; i >= 0; i--) {
    for (int j = i + 1; j < n; j++) {
        dp[i][j] = ...
    }
}

```

**piles = [2, 8, 3, 5]**

start \ end	0	1	2	3
0	(2, 0)	(8, 2)	(5, 8)	(13, 5)
1		(8, 0)	(8, 3)	(11, 5)
2			(3, 0)	(5, 3)
3				(5, 0)

### 三、代码实现

如何实现这个 fir 和 sec 元组呢，你可以用 python，自带元组类型；或者使用 C++ 的 pair 容器；或者用一个三维数组  $dp[n][n][2]$ ，最后一个维度就相当于元组；或者我们自己写一个 Pair 类：

```
class Pair {
    int fir, sec;
    Pair(int fir, int sec) {
        this.fir = fir;
        this.sec = sec;
    }
}
```

然后直接把我们的状态转移方程翻译成代码即可，注意我们要倒着遍历数组：

```
// 返回游戏最后先手和后手的得分之差
int stoneGame(int[] piles) {
    int n = piles.length;
    // 初始化 dp 数组
    Pair[][] dp = new Pair[n][n];
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            dp[i][j] = new Pair(0, 0);
    // 填入 base case
    for (int i = 0; i < n; i++) {
        dp[i][i].fir = piles[i];
        dp[i][i].sec = 0;
    }
    // 倒着遍历数组
    for (int i = n - 2; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            // 先手选择最左边或最右边的分数
            if (piles[i] > piles[j])
                dp[i][j].fir = piles[i] + dp[i + 1][j].sec;
                dp[i][j].sec = dp[i + 1][j].fir;
            else
                dp[i][j].fir = piles[j] + dp[i][j + 1].sec;
                dp[i][j].sec = dp[i][j + 1].fir;
        }
    }
    return dp[0][0].fir - dp[0][0].sec;
}
```

```
int left = piles[i] + dp[i+1][j].sec;
int right = piles[j] + dp[i][j-1].sec;
// 套用状态转移方程
// 先手肯定会选择更大的结果，后手的选择随之改变
if (left > right) {
    dp[i][j].fir = left;
    dp[i][j].sec = dp[i+1][j].fir;
} else {
    dp[i][j].fir = right;
    dp[i][j].sec = dp[i][j-1].fir;
}
}
Pair res = dp[0][n-1];
return res.fir - res.sec;
}
```

动态规划解法，如果没有状态转移方程指导，绝对是一头雾水，但是根据前面的详细解释，读者应该可以清晰理解这一大段代码的含义。

而且，注意到计算 `dp[i][j]` 只依赖其左边和下边的元素，所以说肯定有优化空间，转换成一维 `dp`，想象一下把二维平面压扁，也就是投影到一维。但是，一维 `dp` 比较复杂，可解释性比较差，大家就不必浪费这个时间去理解了。

## 四、最后总结

本文给出了解决博弈问题的动态规划解法。博弈问题的前提一般都是在两个聪明人之间进行，编程描述这种游戏的一般方法是二维 `dp` 数组，数组中通过元组分别表示两人的最优决策。

之所以这样设计，是因为先手在做出选择之后，就成了后手，后手在对方做完选择后，就变成了先手。**这种角色转换使得我们可以重用之前的结果，典型的动态规划标志。**

读到这里的朋友应该能理解算法解决博弈问题的套路了。学习算法，一定要注重算法的模板框架，而不是一些看起来牛逼的思路，也不要奢求上来就写一个最优的解法。不要舍不得多用空间，不要过早尝试优化，不要惧怕多维数组。`dp` 数组就是存储信息避免重复计算的，随便用，直到咱满意为止。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 一个方法团灭 LeetCode 打家劫舍问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">198. House Robber</a>	198. 打家劫舍	简单
<a href="#">337. House Robber III</a>	337. 打家劫舍 III	困难
<a href="#">213. House Robber II</a>	213. 打家劫舍 II	困难

阅读本文前，你需要先学习：

- 二叉树系列算法（纲领篇）
- 动态规划核心框架

今天来讲「打家劫舍」系列问题（英文版叫 House Robber），这个系列是比较有代表性和技巧性的动态规划题目。

打家劫舍系列总共有三道，难度设计比较合理，层层递进。第一道是比较标准的动态规划问题，而第二道融入了环形数组的条件，第三道更绝，把动态规划的自底向上和自顶向下解法和二叉树结合起来，我认为很有启发性。

下面，我们从第一道开始分析。

## 打家劫舍 I

力扣第 198 题「打家劫舍」的题目如下：

街上有一排房屋，用一个包含非负整数的数组 `nums` 表示，每个元素 `nums[i]` 代表第 `i` 间房子中的现金数额。现在你是一名专业盗贼，你希望尽可能多的盗窃这些房子中的现金，但是，相邻的房子不能被同时盗窃，否则会触发报警器，你就凉凉了。

请你写一个算法，计算在不触动报警器的前提下，最多能够盗窃多少现金呢？函数签名如下：

```
int rob(int[] nums);
```

比如说输入 `nums=[2,1,7,9,3,1]`，算法返回 12，小偷可以盗窃 `nums[0]`, `nums[3]`, `nums[5]` 三个房屋，得到的现金之和为  $2 + 9 + 1 = 12$ ，是最优的选择。

题目很容易理解，而且动态规划的特征很明显。我们前文 [动态规划详解](#) 做过总结，解决动态规划问题就是找「状态」和「选择」，仅此而已。

本文为 labuladong.online 网站会员内容，请 [点这里](#) 查看。

# 一个方法团灭 LeetCode 股票买卖问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">188. Best Time to Buy and Sell Stock IV</a>	<a href="#">188. 买卖股票的最佳时机 IV</a>	●
<a href="#">123. Best Time to Buy and Sell Stock III</a>	<a href="#">123. 买卖股票的最佳时机 III</a>	●
<a href="#">121. Best Time to Buy and Sell Stock</a>	<a href="#">121. 买卖股票的最佳时机</a>	●
<a href="#">714. Best Time to Buy and Sell Stock with Transaction Fee</a>	<a href="#">714. 买卖股票的最佳时机含手续费</a>	●
<a href="#">309. Best Time to Buy and Sell Stock with Cooldown</a>	<a href="#">309. 最佳买卖股票时机含冷冻期</a>	●
<a href="#">122. Best Time to Buy and Sell Stock II</a>	<a href="#">122. 买卖股票的最佳时机 II</a>	●

阅读本文前，你需要先学习：

- 动态规划核心框架

很多读者抱怨力扣上的股票系列问题的解法太多，如果面试真的遇到这类问题，基本不会想到那些巧妙的办法，怎么办？所以本文不讲那些过于巧妙的思路，而是稳扎稳打，只用一种通用方法解决所有问题，以不变应万变。

这篇文章参考 [英文版高赞题解](#) 的思路，用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

先随便抽出一道题，看看别人的解法：

```
int maxProfit(int[] prices) {
    if(prices.empty()) return 0;
    int s1 = -prices[0], s2 = INT_MIN, s3 = INT_MIN, s4 = INT_MIN;

    for(int i = 1; i < prices.size(); ++i) {
        s1 = max(s1, -prices[i]);
        s2 = max(s2, s1 + prices[i]);
        s3 = max(s3, s2 - prices[i]);
        s4 = max(s4, s3 + prices[i]);
    }
    return max(0, s4);
}
```

能看懂吧？会做了吗？不可能的，你看不懂，这才正常。就算你勉强看懂了，下一个问题你还是做不出来。为什么别人能写出这么诡异却又高效的解法呢？因为这类问题是有关框架的，但是人家不会告诉你的，因为一旦告诉你，你五分钟就学会了，该算法题就不再神秘，变得不堪一击了。

本文就来告诉你这个框架，然后带着你一道一道秒杀。这篇文章用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

这 6 道题目是有共性的，我们只需要抽出来力扣第 188 题「买卖股票的最佳时机 IV」进行研究，因为这道题是最泛化的形式，其他的问题都是这个形式的简化，看下题目：

#### ▼ 188. 买卖股票的最佳时机 IV [Leetcode](#) | 力扣

给你一个整数数组 `prices` 和一个整数 `k`，其中 `prices[i]` 是某支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 `k` 笔交易。也就是说，你最多可以买 `k` 次，卖 `k` 次。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入: `k = 2, prices = [2, 4, 1]`

输出: 2

解释: 在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。

示例 2：

输入: `k = 2, prices = [3, 2, 6, 5, 0, 3]`

输出: 7

解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出，这笔交易所能获得利润 = 6-2 = 4 。

随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。

提示：

- `1 <= k <= 100`
- `1 <= prices.length <= 1000`
- `0 <= prices[i] <= 1000`

第一题是只进行一次交易，相当于 `k = 1`；第二题是不限交易次数，相当于 `k = +infinity`（正无穷）；第三题是只进行 2 次交易，相当于 `k = 2`；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

下面言归正传，开始解题。

## 一、穷举框架

首先，还是一样的思路：如何穷举？

[动态规划核心套路](#) 说过，动态规划算法本质上就是穷举「状态」，然后在「选择」中选择最优解。

那么对于这道题，我们具体到每一天，看看总共有几种可能的「状态」，再找出每个「状态」对应的「选择」。我们要穷举所有「状态」，穷举的目的是根据对应的「选择」更新状态。听起来抽象，你只要记住「状态」和「选择」两个词就

行，下面实操一下就很容易明白了。

```
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][...] = 择优(选择1, 选择2...)
```

比如说这个问题，**每天都有三种「选择」**：买入、卖出、无操作，我们用 `buy`, `sell`, `rest` 表示这三种选择。

但问题是，并不是每天都可以任意选择这三种选择的，因为 `sell` 必须在 `buy` 之后，`buy` 必须在 `sell` 之后。那么 `rest` 操作还应该分两种状态，一种是 `buy` 之后的 `rest`（持有了股票），一种是 `sell` 之后的 `rest`（没有持有股票）。而且别忘了，我们还有交易次数 `k` 的限制，就是说你 `buy` 还只能在 `k > 0` 的前提下操作。

注意我在本文会频繁使用「交易」这个词，**我们把一次买入和一次卖出定义为一次「交易」**。

很复杂对吧，不要怕，我们现在的目的只是穷举，你有再多的状态，老夫要做的就是一把梭全部列举出来。

这个问题的「状态」有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 `rest` 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合：

```
dp[i][k][0 or 1]  
0 <= i <= n - 1, 1 <= k <= K  
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。  
此问题共 n × K × 2 种状态，全部穷举就能搞定。  
  
for 0 <= i < n:  
    for 1 <= k <= K:  
        for s in {0, 1}:  
            dp[i][k][s] = max(buy, sell, rest)
```

而且我们可以用自然语言描述出每一个状态的含义，比如说 `dp[3][2][1]` 的含义就是：今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易。再比如 `dp[2][3][0]` 的含义：今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易。很容易理解，对吧？

我们想求的最终答案是 `dp[n - 1][K][0]`，即最后一天，最多允许 `K` 次交易，最多获得多少利润。

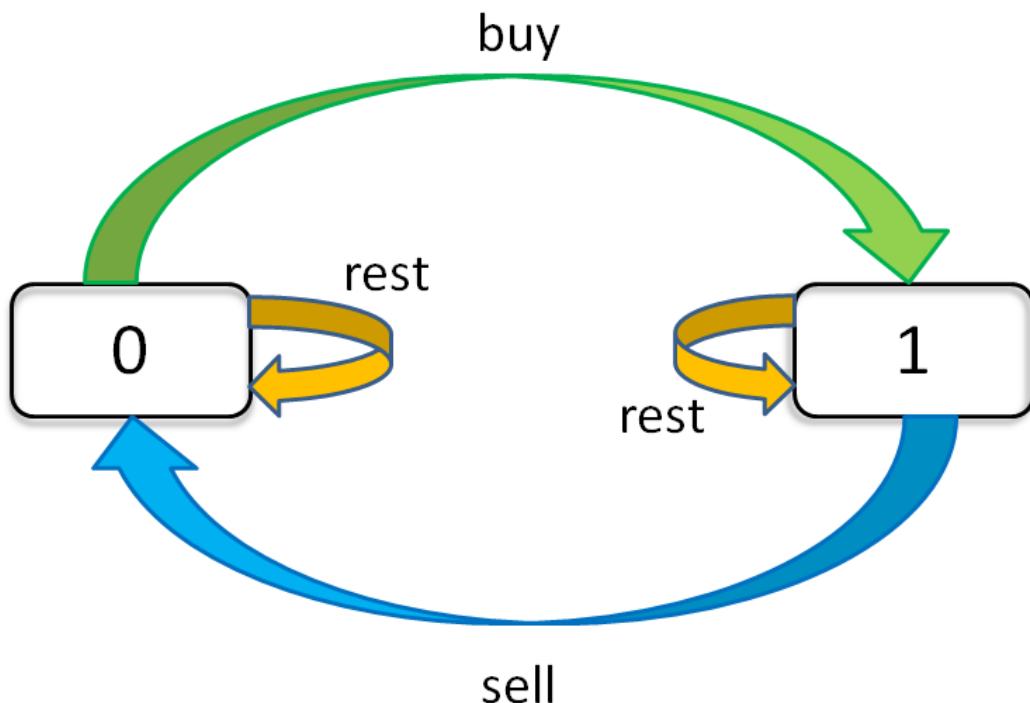
读者可能问为什么不是 `dp[n - 1][K][1]`？因为 `dp[n - 1][K][1]` 代表到最后一天手上还持有股票，`dp[n - 1][K][0]` 表示最后一天手上的股票已经卖出去了，很显然后者得到的利润一定大于前者。

记住如何解释「状态」，一旦你觉得哪里不好理解，把它翻译成自然语言就容易理解了。

## 二、状态转移框架

现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。

只看「持有状态」，可以画个状态转移图：



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下状态转移方程：

$$\text{dp}[i][k][0] = \max(\text{dp}[i-1][k][0], \text{dp}[i-1][k][1] + \text{prices}[i])$$

max( 今天选择 rest, 今天选择 sell )

解释：今天我没有持有股票，有两种可能，我从这两种可能中求最大利润：

- 1、我昨天就没有持有，且截至昨天最大交易次数限制为  $k$ ；然后我今天选择 **rest**，所以我今天还是没有持有，最大交易次数限制依然为  $k$ 。
- 2、我昨天持有股票，且截至昨天最大交易次数限制为  $k$ ；但是今天我 **sell** 了，所以我今天没有持有股票了，最大交易次数限制依然为  $k$ 。

$$\text{dp}[i][k][1] = \max(\text{dp}[i-1][k][1], \text{dp}[i-1][k-1][0] - \text{prices}[i])$$

max( 今天选择 rest, 今天选择 buy )

解释：今天我持有着股票，最大交易次数限制为  $k$ ，那么对于昨天来说，有两种可能，我从这两种可能中求最大利润：

- 1、我昨天就持有着股票，且截至昨天最大交易次数限制为  $k$ ；然后今天选择 **rest**，所以我今天还持有着股票，最大交易次数限制依然为  $k$ 。
- 2、我昨天本没有持有，且截至昨天最大交易次数限制为  $k - 1$ ；但今天我选择 **buy**，所以今天我就持有股票了，最大交易次数限制为  $k$ 。

这里着重提醒一下，**时刻牢记「状态」的定义**，状态  $k$  的定义并不是「已进行的交易次数」，而是「最大交易次数的上限限制」。如果确定今天进行一次交易，且要保证截至今天最大交易次数上限为  $k$ ，那么昨天的最大交易次数上限必须是  $k - 1$ 。举个具体的例子，比方说要求你的银行卡里今天至少有 100 块钱，且你确定你今天可以赚 10 块钱，那么你就要保证昨天的银行卡要至少剩下 90 块钱。

这个解释应该很清楚了，如果 `buy`，就要从利润中减去 `prices[i]`，如果 `sell`，就要给利润增加 `prices[i]`。今天最大利润就是这两种可能选择中较大的那个。

注意 `k` 的限制，在选择 `buy` 的时候相当于开启了一次交易，那么对于昨天来说，交易次数的上限 `k` 应该减小 1。

这里补充修正一点，以前我以为在 `sell` 的时候给 `k` 减小 1 和在 `buy` 的时候给 `k` 减小 1 是等效的，但细心的读者向我提出质疑，经过深入思考我发现前者确实是错误的，因为交易是从 `buy` 开始，如果 `buy` 的选择不改变交易次数 `k` 的话，会出现交易次数超出限制的错误。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。**如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了。**不过还差最后一点点，就是定义 base case，即最简单的情况。

`dp[-1][...][0] = 0`

解释：因为 `i` 是从 `0` 开始的，所以 `i = -1` 意味着还没有开始，这时候的利润当然是 `0`。

`dp[-1][...][1] = -infinity`

解释：还没开始的时候，是不可能持有股票的。

因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。

`dp[...][0][0] = 0`

解释：因为 `k` 是从 `1` 开始的，所以 `k = 0` 意味着根本不允许交易，这时候利润当然是 `0`。

`dp[...][0][1] = -infinity`

解释：不允许交易的情况下，是不可能持有股票的。

因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。

把上面的状态转移方程总结一下：

base case:

`dp[-1][...][0] = dp[...][0][0] = 0`

`dp[-1][...][1] = dp[...][0][1] = -infinity`

状态转移方程：

`dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])`

`dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])`

读者可能会问，这个数组索引是 `-1` 怎么编程表示出来呢，负无穷怎么表示呢？这都是细节问题，有很多方法实现。现在完整的框架已经完成，下面开始具体化。

### 三、秒杀题目

#### 121. 买卖股票的最佳时机

第一题，先说力扣第 121 题「买卖股票的最佳时机」，相当于 `k = 1` 的情况：

▼ 121. 买卖股票的最佳时机 [Leetcode | 力扣](#)

给定一个数组 `prices`，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。

### 示例 1：

输入： [7,1,5,3,6,4]

输出： 5

解释： 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6 - 1 = 5。

注意利润不能是 7 - 1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

### 示例 2：

输入： prices = [7,6,4,3,1]

输出： 0

解释： 在这种情况下，没有交易完成，所以最大利润为 0。

### 提示：

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$

直接套状态转移方程，根据 base case，可以做一些化简：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
= max(dp[i-1][1], -prices[i])
```

解释：  $k = 0$  的 base case，所以  $dp[i-1][0][0] = 0$ 。

现在发现  $k$  都是 1，不会改变，即  $k$  对状态转移已经没有影响了。

可以进行进一步化简去掉所有  $k$ ：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])
```

直接写出代码：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

显然  $i = 0$  时  $i - 1$  是不合法的索引，这是因为我们没有对  $i$  的 base case 进行处理，可以这样给一个特化处理：

```
if (i - 1 == -1) {
    dp[i][0] = 0;
    // 根据状态转移方程可得：
    // dp[i][0]
```

```

// = max(dp[-1][0], dp[-1][1] + prices[i])
// = max(0, -infinity + prices[i]) = 0
// = max(dp[-1][0], dp[-1][1] + prices[i])
// = max(0, -infinity + prices[i]) = 0

dp[i][1] = -prices[i];
// 根据状态转移方程可得:
//   dp[i][1]
// = max(dp[-1][1], dp[-1][0] - prices[i])
// = max(-infinity, 0 - prices[i])
// = -prices[i]
// = max(dp[-1][1], dp[-1][0] - prices[i])
// = max(-infinity, 0 - prices[i])
// = -prices[i]
continue;
}

```

第一题就解决了，但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，所以可以用前文 动态规划的降维打击：空间压缩技巧，不需要用整个 `dp` 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1)：

```

// 原始版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    // base case: dp[-1][0] = 0, dp[-1][1] = -infinity
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        // dp[i][1] = max(dp[i-1][1], -prices[i])
        dp_i_1 = Math.max(dp_i_1, -prices[i]);
    }
    return dp_i_0;
}

```

两种方式都是一样的，不过这种编程方法简洁很多，但是如果没有前面状态转移方程的引导，是肯定看不懂的。后续的题目，你可以对比一下如何把 `dp` 数组的空间优化掉。

## 122. 买卖股票的最佳时机 II

第二题，看一下力扣第 122 题「买卖股票的最佳时机 II」，也就是  $k$  为正无穷的情况：

### ▼ 122. 买卖股票的最佳时机 II Leetcode | 力扣

给你一个整数数组  $\text{prices}$ ，其中  $\text{prices}[i]$  表示某支股票第  $i$  天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

返回 你能获得的 **最大 利润**。

示例 1：

输入:  $\text{prices} = [7, 1, 5, 3, 6, 4]$

输出: 7

解释: 在第 2 天 (股票价格 = 1) 的时候买入，在第 3 天 (股票价格 = 5) 的时候卖出，这笔交易所能获得利润 =  $5 - 1 = 4$ 。

随后，在第 4 天 (股票价格 = 3) 的时候买入，在第 5 天 (股票价格 = 6) 的时候卖出，这笔交易所能获得利润 =  $6 - 3 = 3$ 。

最大总利润为  $4 + 3 = 7$ 。

示例 2：

输入:  $\text{prices} = [1, 2, 3, 4, 5]$

输出: 4

解释: 在第 1 天 (股票价格 = 1) 的时候买入，在第 5 天 (股票价格 = 5) 的时候卖出，这笔交易所能获得利润 =  $5 - 1 = 4$ 。

最大总利润为 4。

示例 3：

输入:  $\text{prices} = [7, 6, 4, 3, 1]$

输出: 0

解释: 在这种情况下，交易无法获得正利润，所以不参与交易可以获得最大利润，最大利润为 0。

提示：

- $1 \leq \text{prices.length} \leq 3 * 10^4$
- $0 \leq \text{prices}[i] \leq 10^4$

题目还专门强调可以在同一天出售，但我觉得这个条件纯属多余，如果当天买当天卖，那利润当然就是 0，这不是和没有进行交易是一样的吗？这道题的特点在于没有给出交易总数  $k$  的限制，也就相当于  $k$  为正无穷。

如果  $k$  为正无穷，那么就可以认为  $k$  和  $k - 1$  是一样的。可以这样改写框架：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])
```

我们发现数组中的  $k$  已经不会改变了，也就是说不需要记录  $k$  这个状态了：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

直接翻译成代码：

```
// 原始版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], dp[i-1][0] - prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
    }
    return dp_i_0;
}
```

## 309. 最佳买卖股票时机含冷冻期

第三题，看力扣第 309 题「最佳买卖股票时机含冷冻期」，也就是  $k$  为正无穷，但含有交易冷冻期的情况：

### ▼ 309. 买卖股票的最佳时机含冷冻期 [Leetcode | 力扣](#)

给定一个整数数组  $\text{prices}$ ，其中第  $\text{prices}[i]$  表示第  $i$  天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 卖出股票后，你无法在第二天买入股票（即冷冻期为 1 天）。

**注意：**你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

### 示例 1：

```
输入: prices = [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

## 示例 2:

```
输入: prices = [1]
输出: 0
```

## 提示:

- $1 \leq \text{prices.length} \leq 5000$
- $0 \leq \text{prices[i]} \leq 1000$

和上一道题一样的，只不过每次 `sell` 之后要等一天才能继续交易，只要把这个特点融入上一题的状态转移方程即可：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释: 第 i 天选择 buy 的时候, 要从 i-2 的状态转移, 而不是 i-1。
```

翻译成代码：

```
// 原始版本
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case 1
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        if (i - 2 == -1) {
            // base case 2
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            // i - 2 小于 0 时根据状态转移方程推出对应 base case
            dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
            // dp[i][1]
            // = max(dp[i-1][1], dp[-1][0] - prices[i])
            // = max(dp[i-1][1], 0 - prices[i])
            // = max(dp[i-1][1], -prices[i])
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], dp[i-2][0] - prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    // 代表 dp[i-2][0]
    int dp_pre_0 = 0;
    for (int i = 0; i < n; i++) {
```

```
    int temp = dp_i_0;
    dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
    dp_i_1 = Math.max(dp_i_1, dp_pre_0 - prices[i]);
    dp_pre_0 = temp;
}
return dp_i_0;
}
```

## 714. 买卖股票的最佳时机含手续费

第四题，看力扣第 714 题「买卖股票的最佳时机含手续费」，也就是 k 为正无穷且考虑交易手续费的情况：

### ▼ 714. 买卖股票的最佳时机含手续费 [Leetcode | 力扣](#)

给定一个整数数组 `prices`，其中 `prices[i]` 表示第 `i` 天的股票价格；整数 `fee` 代表了交易股票的手续费。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

#### 示例 1：

```
输入: prices = [1, 3, 2, 8, 4, 9], fee = 2
输出: 8
解释: 能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8
```

#### 示例 2：

```
输入: prices = [1,3,7,5,10,3], fee = 3
输出: 6
```

#### 提示：

- $1 \leq \text{prices.length} \leq 5 * 10^4$
- $1 \leq \text{prices}[i] < 5 * 10^4$
- $0 \leq \text{fee} < 5 * 10^4$

每次交易要支付手续费，只要把手续费从利润中减去即可，改写方程：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
```

解释：相当于买入股票的价格升高了。

在第一个式子里减也是一样的，相当于卖出股票的价格减小了。

如果直接把 `fee` 放在第一个式子里减，会有一些测试用例无法通过，错误原因是整型溢出而不是思路问题。一种解决方案是把代码中的 `int` 类型都改成 `long` 类型，避免 `int` 的整型溢出。

直接翻译成代码，注意状态转移方程改变后 base case 也要做出对应改变：

```
// 原始版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i] - fee;
            // dp[i][1]
            // = max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee)
            // = max(dp[-1][1], dp[-1][0] - prices[i] - fee)
            // = max(-inf, 0 - prices[i] - fee)
            // = -prices[i] - fee
            continue;
        }
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i] - fee);
    }
    return dp_i_0;
}
```

## 123. 买卖股票的最佳时机 III

第五题，看力扣第 123 题「买卖股票的最佳时机 III」，也就是 `k = 2` 的情况：

### ▼ 123. 买卖股票的最佳时机 III [Leetcode | 力扣](#)

给定一个数组，它的第 `i` 个元素是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **两笔** 交易。

**注意：**你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

**示例 1：**

输入: prices = [3,3,5,0,0,3,1,4]

输出: 6

解释: 在第 4 天 (股票价格 = 0) 的时候买入, 在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 =  $3 - 0 = 3$ 。

随后, 在第 7 天 (股票价格 = 1) 的时候买入, 在第 8 天 (股票价格 = 4) 的时候卖出, 这笔交易所能获得利润 =  $4 - 1 = 3$ 。

## 示例 2:

输入: prices = [1,2,3,4,5]

输出: 4

解释: 在第 1 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 5) 的时候卖出, 这笔交易所能获得利润 =  $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票, 之后再将它们卖出。

因为这样属于同时参与了多笔交易, 你必须在再次购买前出售掉之前的股票。

## 示例 3:

输入: prices = [7,6,4,3,1]

输出: 0

解释: 在这个情况下, 没有交易完成, 所以最大利润为 0。

## 示例 4:

输入: prices = [1]

输出: 0

### 提示:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^5$

$k = 2$  和前面题目的情况稍微不同, 因为上面的情况都和  $k$  的关系不太大: 要么  $k$  是正无穷, 状态转移和  $k$  没关系了; 要么  $k = 1$ , 跟  $k = 0$  这个 base case 挨得近, 最后也没有存在感。

这道题  $k = 2$  和后面要讲的  $k$  是任意正整数的情况下, 对  $k$  的处理就凸显出来了, 我们直接写代码, 边写边分析原因。

原始的状态转移方程, 没有可化简的地方

$\text{dp}[i][k][0] = \max(\text{dp}[i-1][k][0], \text{dp}[i-1][k][1] + \text{prices}[i])$   
 $\text{dp}[i][k][1] = \max(\text{dp}[i-1][k][1], \text{dp}[i-1][k-1][0] - \text{prices}[i])$

按照之前的代码, 我们可能想当然这样写代码 (错误的) :

```
int k = 2;
int[][][] dp = new int[n][k + 1][2];
for (int i = 0; i < n; i++) {
    if (i - 1 == -1) {
```

```

    // 处理 base case
    dp[i][k][0] = 0;
    dp[i][k][1] = -prices[i];
    continue;
}
dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
}
return dp[n - 1][k][0];

```

为什么错误？我这不是照着状态转移方程写的吗？

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中  $k$  都被化简掉了。

比如说第一题， $k = 1$  时的代码框架：

```

int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];

```

但当  $k = 2$  时，由于没有消掉  $k$  的影响，所以必须要对  $k$  进行穷举：

```

// 原始版本
int maxProfit_k_2(int[] prices) {
    int max_k = 2, n = prices.length;
    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++) {
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
    }
    // 穷举了  $n \times max\_k \times 2$  个状态，正确。
    return dp[n - 1][max_k][0];
}

```

这里肯定会有读者疑惑， $k$  的 base case 是 0，按理说应该从  $k = 1$ ,  $k++$  这样穷举状态  $k$  才对？而且如果你真的这样从小到大遍历  $k$ ，提交发现也是可以的。

这个疑问很正确，因为我们前文 [动态规划答疑篇](#) 有介绍 `dp` 数组的遍历顺序是怎么确定的，主要是根据 base case，以 base case 为起点，逐步向结果靠近。

但为什么我从大到小遍历 `k` 也可以正确提交呢？因为你注意看，`dp[i][k][..]` 不会依赖 `dp[i][k - 1][..]`，而是依赖 `dp[i - 1][k - 1][..]`，而 `dp[i - 1][..][..]`，都是已经计算出来的，所以不管你是 `k = max_k, k--`，还是 `k = 1, k++`，都是可以得出正确答案的。

那为什么我使用 `k = max_k, k--` 的方式呢？因为这样符合语义：

你买股票，初始的「状态」是什么？应该是从第 0 天开始，而且还没有进行过买卖，所以最大交易次数限制 `k` 应该是 `max_k`；而随着「状态」的推移，你会进行交易，那么交易次数上限 `k` 应该不断减少，这样一想，`k = max_k, k--` 的方式是比较合乎实际场景的。

当然，这里 `k` 取值范围比较小，所以也可以不用 for 循环，直接把 `k = 1` 和 `2` 的情况全部列举出来也可以：

```
// 状态转移方程：  
// dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])  
// dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])  
// dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])  
// dp[i][1][1] = max(dp[i-1][1][1], -prices[i])  
  
// 空间复杂度优化版本  
int maxProfit_k_2(int[] prices) {  
    // base case  
    int dp_i10 = 0, dp_i11 = Integer.MIN_VALUE;  
    int dp_i20 = 0, dp_i21 = Integer.MIN_VALUE;  
    for (int price : prices) {  
        dp_i20 = Math.max(dp_i20, dp_i21 + price);  
        dp_i21 = Math.max(dp_i21, dp_i10 - price);  
        dp_i10 = Math.max(dp_i10, dp_i11 + price);  
        dp_i11 = Math.max(dp_i11, -price);  
    }  
    return dp_i20;  
}
```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 `a, b, c, d`。这样当别人看到你的代码时就会大惊失色，对你肃然起敬。

第六题，着力扣第 188 题「买卖股票的最佳时机 IV」，即 `k` 可以是题目给定的任何数的情况：

#### ▼ 188. 买卖股票的最佳时机 IV [Leetcode | 力扣](#)

给你一个整数数组 `prices` 和一个整数 `k`，其中 `prices[i]` 是某支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 `k` 笔交易。也就是说，你最多可以买 `k` 次，卖 `k` 次。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入： `k = 2, prices = [2, 4, 1]`

输出： `2`

解释： 在第 1 天（股票价格 = 2）的时候买入，在第 2 天（股票价格 = 4）的时候卖出，这笔交易所能获得利润 =  $4 - 2 = 2$ 。

## 示例 2：

输入:  $k = 2$ ,  $\text{prices} = [3, 2, 6, 5, 0, 3]$

输出: 7

解释: 在第 2 天 (股票价格 = 2) 的时候买入, 在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 =  $6 - 2 = 4$ 。

随后, 在第 5 天 (股票价格 = 0) 的时候买入, 在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 =  $3 - 0 = 3$ 。

## 提示:

- $1 \leq k \leq 100$
- $1 \leq \text{prices.length} \leq 1000$
- $0 \leq \text{prices}[i] \leq 1000$

有了上一题  $k = 2$  的铺垫, 这题应该和上题的第一个解法没啥区别, 你把上题的  $k = 2$  换成题目输入的  $k$  就行了。

但试一下发现会出一个内存超限的错误, 原来是传入的  $k$  值会非常大,  $\text{dp}$  数组太大了。那么现在想想, 交易次数  $k$  最多有多大呢?

一次交易由买入和卖出构成, 至少需要两天。所以说有效的限制  $k$  应该不超过  $n/2$ , 如果超过, 就没有约束作用了, 相当于  $k$  没有限制的情况, 而这种情况是之前解决过的。

所以我们可以直接把之前的代码重用:

```
int maxProfit_k_any(int max_k, int[] prices) {
    int n = prices.length;
    if (n <= 0) {
        return 0;
    }
    if (max_k > n / 2) {
        // 复用之前交易次数 k 没有限制的情况
        return maxProfit_k_inf(prices);
    }

    // base case:
    // dp[-1][...][0] = dp[...][0][0] = 0
    // dp[-1][...][1] = dp[...][0][1] = -infinity
    int[][][] dp = new int[n][max_k + 1][2];
    // k = 0 时的 base case
    for (int i = 0; i < n; i++) {
        dp[i][0][1] = Integer.MIN_VALUE;
        dp[i][0][0] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 i = -1 时的 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
}
```

```
        }
    return dp[n - 1][max_k][0];
}
```

至此，6道题目通过一个状态转移方程全部解决。

## 万法归一

如果你能看到这里，已经可以给你鼓掌了，初次理解如此复杂的动态规划问题想必消耗了你不少的脑细胞，不过这是值得的，股票系列问题已经属于动态规划问题中较困难的了，如果这些题你都能搞懂，试问，其他那些虾兵蟹将又何足道哉？

现在你已经过了九九八十一难中的前八十难，最后我还要再难为你一下，请你实现如下函数：

```
int maxProfit_all_in_one(int max_k, int[] prices, int cooldown, int fee);
```

输入股票价格数组 `prices`，你最多进行 `max_k` 次交易，每次交易需要额外消耗 `fee` 的手续费，而且每次交易之后需要经过 `cooldown` 天的冷冻期才能进行下一次交易，请你计算并返回可以获得的最大利润。

怎么样，有没有被吓到？如果你直接给别人出一道这样的题目，估计对方要当场吐血，不过我们这样一步步做过来，你应该很容易发现这道题目就是之前我们探讨的几种情况的组合体嘛。

所以，我们只要把之前实现的几种代码掺和到一块，在 **base case** 和状态转移方程中同时加上 `cooldown` 和 `fee` 的约束就行了：

```
// 同时考虑交易次数的限制、冷冻期和手续费
int maxProfit_all_in_one(int max_k, int[] prices, int cooldown, int fee) {
    int n = prices.length;
    if (n <= 0) {
        return 0;
    }
    if (max_k > n / 2) {
        // 交易次数 k 没有限制的情况
        return maxProfit_k_inf(prices, cooldown, fee);
    }

    int[][][] dp = new int[n][max_k + 1][2];
    // k = 0 时的 base case
    for (int i = 0; i < n; i++) {
        dp[i][0][1] = Integer.MIN_VALUE;
        dp[i][0][0] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // base case 1
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i] - fee;
                continue;
            }

            // 包含 cooldown 的 base case
            if (i - cooldown - 1 < 0) {
                // base case 2
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i] - fee;
            } else {
                dp[i][k][0] = Math.max(dp[i - 1][k][0], dp[i - 1][k][1] + prices[i]);
                dp[i][k][1] = Math.max(dp[i - 1][k - 1][0], dp[i - 1][k - 1][1] - prices[i] - fee);
            }
        }
}
```

```

        dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
        // 别忘了减 fee
        dp[i][k][1] = Math.max(dp[i-1][k][1], -prices[i] - fee);
        continue;
    }
    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
    // 同时考虑 cooldown 和 fee
    dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-cooldown-1][k-1][0] -
prices[i] - fee);
}
return dp[n - 1][max_k][0];
}

// k 无限制, 包含手续费和冷冻期
int maxProfit_k_inf(int[] prices, int cooldown, int fee) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case 1
            dp[i][0] = 0;
            dp[i][1] = -prices[i] - fee;
            continue;
        }
        // 包含 cooldown 的 base case
        if (i - cooldown - 1 < 0) {
            // base case 2
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            // 别忘了减 fee
            dp[i][1] = Math.max(dp[i-1][1], -prices[i] - fee);
            continue;
        }
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        // 同时考虑 cooldown 和 fee
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - cooldown - 1][0] - prices[i] -
fee);
    }
    return dp[n - 1][0];
}

```

你可以用这个 `maxProfit_all_in_one` 函数去完成之前讲的 6 道题目，因为我们无法对 `dp` 数组进行优化，所以执行效率上不是最优的，但正确性上肯定是没有问题的。

最后总结一下吧，本文给大家讲了如何通过状态转移的方法解决复杂的问题，用一个状态转移方程秒杀了 6 道股票买卖问题，现在回头去看，其实也不算那么可怕对吧？

关键就在于列举出所有可能的「状态」，然后想想怎么穷举更新这些「状态」。一般用一个多维 `dp` 数组储存这些状态，从 `base case` 开始向后推进，推进到最后的状态，就是我们想要的答案。想想这个过程，你是不是有点理解「动态规划」这个名词的意义了呢？

具体到股票买卖问题，我们发现了三个状态，使用了一个三维数组，无非还是穷举 + 更新，不过我们可以说的高大上一点，这叫「三维 DP」，听起来是不是很厉害？

## ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
-	剑指 Offer 63. 股票的最大利润	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 老司机加油算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
134. Gas Station	134. 加油站	困难

今天讲一个贪心的老司机的故事，就是力扣第 134 题「加油站」：

## ▼ 134. 加油站 [Leetcode](#) | [力扣](#)

在一条环路上有  $n$  个加油站，其中第  $i$  个加油站有汽油  $gas[i]$  升。

你有一辆油箱容量无限的的汽车，从第  $i$  个加油站开往第  $i+1$  个加油站需要消耗汽油  $cost[i]$  升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组  $gas$  和  $cost$ ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回  $-1$ 。如果存在解，则 **保证** 它是 **唯一** 的。

### 示例 1:

```
输入: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
输出: 3
解释:
从 3 号加油站(索引为 3 处)出发, 可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站, 此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站, 此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站, 此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站, 此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站, 你需要消耗 5 升汽油, 正好足够你返回到 3 号加油站。
因此, 3 可为起始索引。
```

### 示例 2:

```
输入: gas = [2,3,4], cost = [3,4,3]
输出: -1
解释:
你不能从 0 号或 1 号加油站出发, 因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发, 可以获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站, 此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站, 此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站, 因为返程需要消耗 4 升汽油, 但是你的油箱只有 3 升汽油。
因此, 无论怎样, 你都不可能绕环路行驶一周。
```

提示:

- `gas.length == n`
- `cost.length == n`
- `1 <= n <= 105`
- `0 <= gas[i], cost[i] <= 104`

题目应该不难理解，就是每到达一个站点 `i`，可以加 `gas[i]` 升油，但离开站点 `i` 需要消耗 `cost[i]` 升油，问你从哪个站点出发，可以兜一圈回来。

要说暴力解法，肯定很容易想到，用一个 `for` 循环遍历所有站点，假设为起点，然后再套一层 `for` 循环，判断一下是否能够转一圈回到起点：

```
int n = gas.length;
for (int start = 0; start < n; start++) {
    for (int step = 0; step < n; step++) {
        int i = (start + step) % n;
        tank += gas[i];
        tank -= cost[i];
        // 判断油箱中的油是否耗尽
    }
}
```

很明显时间复杂度是  $O(N^2)$ ，这么简单粗暴的解法一定不是最优的，我们试图分析一下是否有优化的余地。

暴力解法是否有重复计算的部分？是否可以抽象出「状态」，是否对同一个「状态」重复计算了多次？

我们前文 [动态规划详解](#) 说过，变化的量就是「状态」。那么观察这个暴力穷举的过程，变化的量有两个，分别是「起点」和「当前油箱的油量」，但这两个状态的组合肯定有不下  $O(N^2)$  种，显然没有任何优化的空间。

所以说这道题肯定不是通过简单的剪枝来优化暴力解法的效率，而是需要我们发现一些隐藏较深的规律，从而减少一些冗余的计算。

下面我们介绍两种方法巧解这道题，分别是数学图像解法和贪心解法。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 贪心算法之区间调度问题



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">452. Minimum Number of Arrows to Burst Balloons</a>	<a href="#">452. 用最少量的箭引爆气球</a>	
<a href="#">435. Non-overlapping Intervals</a>	<a href="#">435. 无重叠区间</a>	

-----

什么是贪心算法呢？贪心算法可以认为是动态规划算法的一个特例，相比动态规划，使用贪心算法需要满足更多的条件（贪心选择性质），但是效率比动态规划要高。

比如说一个算法问题使用暴力解法需要指数级时间，如果能使用动态规划消除重叠子问题，就可以降到多项式级别的时间，如果满足贪心选择性质，那么可以进一步降低时间复杂度，达到线性级别的。

什么是贪心选择性质呢，简单说就是：每一步都做出一个局部最优的选择，最终的结果就是全局最优。注意哦，这是一种特殊性质，其实只有一部分问题拥有这个性质。

比如你面前放着 100 张人民币，你只能拿十张，怎么才能拿最多的面额？显然每次选择剩下钞票中面值最大的一张，最后你的选择一定是最优的。

然而，大部分问题明显不具有贪心选择性质。比如打斗地主，对手出对儿三，按照贪心策略，你应该出尽可能小的牌刚好压制住对方，但现实情况我们甚至可能会出王炸。这种情况就不能用贪心算法，而得使用动态规划解决，参见前文 [动态规划解决博弈问题](#)。

## 一、问题概述

言归正传，本文解决一个很经典的贪心算法问题 Interval Scheduling（区间调度问题），也就是力扣第 435 题「无重叠区间」：

给你很多形如 `[start, end]` 的闭区间，请你设计一个算法，算出这些区间中最多有几个互不相交的区间。

```
int intervalSchedule(int[][] intvs);
```

举个例子，`intvs = [[1,3], [2,4], [3,6]]`，这些区间最多有 2 个区间互不相交，即 `[[1,3], [3,6]]`，你的算法应该返回 2。注意边界相同并不算相交。

这个问题在生活中的应用广泛，比如你今天有好几个活动，每个活动都可以用区间 `[start, end]` 表示开始和结束的时间，请问你今天最多能参加几个活动呢？显然你一个人不能同时参加两个活动，所以说这个问题就是求这些时间区间的最大不相交子集。

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。



# 扫描线技巧：安排会议室



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">253. Meeting Rooms II</a>	<a href="#">253. 会议室 II</a>	

之前面试，被问到一道非常经典且非常实用的算法题目：会议室安排问题。

力扣上类似的问题是会员题目，你可能没办法做，但对于这种经典的算法题，掌握思路还是必要的。

先说下题目，力扣第 253 题「会议室 II」：

给你输入若干形如 `[begin, end]` 的区间，代表若干会议的开始时间和结束时间，请你计算至少需要申请多少间会议室。

函数签名如下：

```
// 返回需要申请的会议室数量
int minMeetingRooms(int[][] meetings);
```

比如给你输入 `meetings = [[0,30],[5,10],[15,20]]`，算法应该返回 2，因为后两个会议和第一个会议时间是冲突的，至少申请两个会议室才能让所有会议顺利进行。

如果会议之间的时间有重叠，那就得额外申请会议室来开会，想求至少需要多少间会议室，就是让你计算同一时刻最多有多少会议在同时进行。

换句话说，如果把每个会议的起始时间看做一个线段区间，那么题目就是让你求最多有几个重叠区间，仅此而已。

我们之前也学习过区间相关的算法，如果你对 [差分数组技巧](#) 有印象，应该首先能想到用那个技巧来解决这个题。

这道题相当于是说，给你一个原本全是 0 的数组，然后给你若干区间，让你对每个区间中的元素都加 1，问你最后整个数组中的最大值是多少。这就是经典的差分数组实用场景对吧，直接套用前文给的 `Difference` 类就可以解决这个问题了。

但是差分数组技巧有一个问题，就是你必须把那个全是 0 的初始数组构造出来。由于我们用数组的索引表示时间，所以这个数组的长度取决于时间区间的最大值。

比如输入 `meetings = [[0,30],[5,10],[15,20]]`，那么你得构造一个长度为 30 的数组。那如果输入 `meetings = [[0,30],[5,10],[10^8,10^9]]`，这样的话你就得构造一个长度为  $10^9$  的数组，这显然是有问题的。不过这道题给的数据规模是时间的取值最多为  $10^6$ ，不算是特别大，用差分数组的方法应该可以通过。

但本文再教你另外的一个处理区间的技巧，不用构造这么大的数组，也能巧妙解决这个问题。

## 题目延伸

我们之前写过很多区间调度相关的文章，这里就顺便帮大家梳理一下这类问题的思路：

**第一个场景**，假设现在只有一个会议室，还有若干会议，你如何将尽可能多的会议安排到这个会议室里？

这个问题需要将这些会议（区间）按结束时间（右端点）排序，然后进行处理，详见前文 [贪心算法做时间管理](#)。

**第二个场景**，给你若干较短的视频片段，和一个较长的视频片段，请你从较短的片段中尽可能少地挑出一些片段，拼接出较长的这个片段。

这个问题需要将这些视频片段（区间）按开始时间（左端点）排序，然后进行处理，详见前文 [剪视频剪出一个贪心算法](#)。

**第三个场景**，给你若干区间，其中可能有些区间比较短，被其他区间完全覆盖住了，请你删除这些被覆盖的区间。

这个问题需要将这些区间按左端点排序，然后就能找到并删除那些被完全覆盖的区间了，详见前文 [删除覆盖区间](#)。

**第四个场景**，给你若干区间，请你将所有有重叠部分的区间进行合并。

这个问题需要将这些区间按左端点排序，方便找出存在重叠的区间，详见前文 [合并重叠区间](#)。

**第五个场景**，有两个部门同时预约了同一个会议室的若干时间段，请你计算会议室的冲突时段。

这个问题就是给你两组区间列表，请你找出这两组区间的交集，这需要你将这些区间按左端点排序，详见前文 [区间交集问题](#)。

**第六个场景**，假设现在只有一个会议室，还有若干会议，如何安排会议才能使这个会议室的闲置时间最少？

这个问题需要动动脑筋，说白了这就是个 0-1 背包问题的变形：

会议室可以看做一个背包，每个会议可以看做一个物品，物品的价值就是会议的时长，请问你如何选择物品（会议）才能最大化背包中的价值（会议室的使用时长）？

当然，这里背包的约束不是一个最大重量，而是各个物品（会议）不能互相冲突。把各个会议按照结束时间进行排序，然后参考前文 [0-1 背包问题详解](#) 的思路和 TreeMap 即可解决。

力扣第 1235 题「规划兼职工作」就是类似的题目，我在插件思路中给出了详细的解答，你可以安装我的 [Chrome 插件](#) 去查看，我在这里就不花费篇幅了。

**第七个场景**，就是本文想讲的场景，给你若干会议，让你最小化申请会议室的数量。

好了，举例了这么多，来看看今天的这个问题如何解决。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 剪视频剪出一个贪心算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode

力扣

难度

[1024. Video Stitching](#)

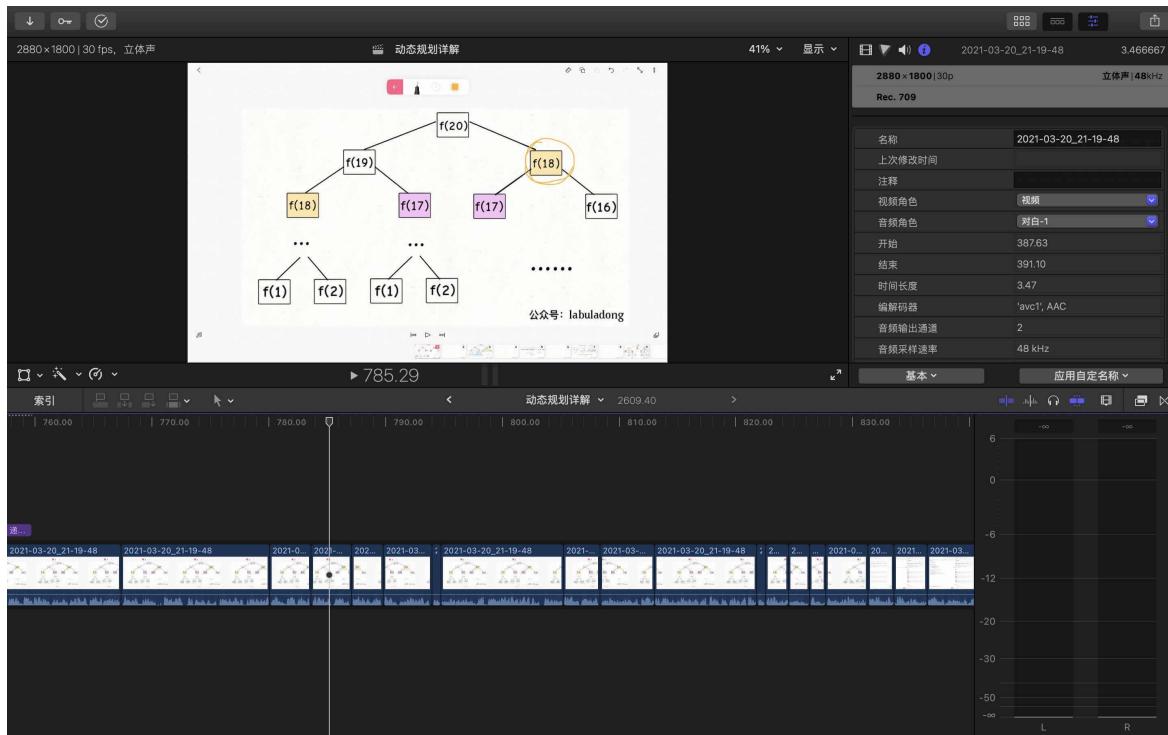
[1024. 视频拼接](#)



前面发过几个视频，也算是对视频剪辑入了个门。像我这种非专业剪辑玩家，不做什么宏大特效电影镜头，只是做个视频教程，其实也没啥难度，只需要把视频剪流畅，所以用到最多的功能就是切割功能，然后删除和拼接视频片接。

没有剪过视频的读者可能不知道，在常用的剪辑软件中视频被切割成若干片段之后，每个片段都可以还原成原始视频。

就比如一个 10 秒的视频，在中间切一刀剪成两个 5 秒的视频，这两个五秒的视频各自都可以还原成 10 秒的原视频。就好像蚯蚓，把自己切成 4 段就能搓麻，把自己切成 11 段就可以凑一个足球队。



剪视频时，每个视频片段都可以抽象成了一个个区间，时间就是区间的端点，这些区间有的相交，有的不相交... ...

假设剪辑软件不支持将视频片段还原成原视频，那么如果给我若干视频片段，我怎么将它们还原成原视频呢？

本文为 [labuladong.online](http://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

# 如何运用贪心思想玩跳跃游戏



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
45. Jump Game II	45. 跳跃游戏 II	困难
55. Jump Game	55. 跳跃游戏	困难

-----

阅读本文前，你需要先学习：

- [动态规划核心框架](#)

经常有读者在后台问，动态规划和贪心算法到底有啥关系。我们之前的文章 [贪心算法之区间调度问题](#) 就说过一个常见的区间调度的贪心算法问题。

说白了，贪心算法可以理解为一种特殊的动态规划问题，拥有一些更特殊的性质，可以进一步降低动态规划算法的时间复杂度。那么这篇文章，就讲 LeetCode 上两道经典的贪心算法：跳跃游戏 I 和跳跃游戏 II。

我们可以对这两道题分别使用动态规划算法和贪心算法进行求解，通过实践，你就能更深刻地理解贪心和动规的区别和联系了。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 一行代码就能解决的算法题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
877. Stone Game	877. 石子游戏	●
292. Nim Game	292. Nim 游戏	●
319. Bulb Switcher	319. 灯泡开关	●

-----

下文是我在刷题过程中总结的三道有趣的「脑筋急转弯」题目，可以使用算法编程解决，但只要稍加思考，就能找到规律，直接想出答案。

## 一、Nim 游戏

力扣第 292 题「Nim 游戏」给了这样一个游戏规则：

你和你的朋友面前有一堆石子，你们轮流拿，一次至少拿一颗，最多拿三颗，谁拿走最后一颗石子谁获胜。

假设你们都很聪明，由你第一个开始拿，请你写一个算法，输入一个正整数  $n$ ，返回你是否能赢（true 或 false）。

比如现在有 4 颗石子，算法应该返回 false。因为无论你拿 1 颗 2 颗还是 3 颗，对方都能一次性拿完，拿走最后一颗石子，所以你一定会输。

首先，这道题肯定可以使用动态规划，因为显然原问题存在子问题，且子问题存在重复。但是因为你们都很聪明，涉及到你和对手的博弈，动态规划会比较复杂。

我们解决这种问题的思路一般都是反着思考：

如果我能赢，那么最后轮到我取石子的时候必须要剩下 1~3 颗石子，这样我才能一把拿完。

如何营造这样的一个局面呢？显然，如果对手拿的时候只剩 4 颗石子，那么无论他怎么拿，总会剩下 1~3 颗石子，我就能赢。

如何逼迫对手面对 4 颗石子呢？要想办法，让我选择的时候还有 5~7 颗石子，这样的话我就有把握让对方不得不面对 4 颗石子。

如何营造 5~7 颗石子的局面呢？让对手面对 8 颗石子，无论他怎么拿，都会给我剩下 5~7 颗，我就能赢。

这样一直循环下去，我们发现只要踩到 4 的倍数，就落入了圈套，永远逃不出 4 的倍数，而且一定会输。所以这道题的解法非常简单：

```
boolean canWinNim(int n) {  
    // 如果上来就踩到 4 的倍数，那就认输吧  
}
```

```
// 否则，可以把对方控制在 4 的倍数，必胜  
return n % 4 != 0;  
}
```

## 二、石头游戏

力扣第 877 题「石子游戏」的规则是这样的：

你和你的朋友面前有一排石头堆，用一个数组 `piles` 表示，`piles[i]` 表示第 `i` 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。

假设你们都很聪明，由你第一个开始拿，请你写一个算法，输入一个数组 `piles`，返回你是否能赢（true 或 false）。

注意，石头的堆的数量为偶数，所以你们两人拿走的堆数一定是相同的。石头的总数为奇数，也就是你们最后不可能拥有相同多的石头，一定有胜负之分。

举个例子，`piles=[2, 1, 9, 5]`，你先拿，可以拿 2 或者 5，你选择 2。

`piles=[1, 9, 5]`，轮到对手，可以拿 1 或 5，他选择 5。

`piles=[1, 9]` 轮到你拿，你拿 9。

最后，你的对手只能拿 1 了。

这样下来，你总共拥有  $2 + 9 = 11$  颗石头，对手有  $5 + 1 = 6$  颗石头，你是可以赢的，所以算法应该返回 true。

你看到了，并不是简单的挑数字大的选，为什么第一次选择 2 而不是 5 呢？因为 5 后面是 9，你要想贪图一时的利益，就把 9 这堆石头暴露给对手了，那你就输了。

这也是强调双方都很聪明的原因，算法也是求最优决策过程中你是否能赢。

这道题又涉及到两人的博弈，也可以用动态规划算法暴力试，比较麻烦。但我们只要对规则深入思考，就会大惊失色：只要你足够聪明，你是必胜无疑的，因为你是先手。

```
boolean stoneGame(int[] piles) {  
    return true;  
}
```

这是为什么呢，因为题目有两个条件很重要：一是石头总共有偶数堆，石头的总数是奇数。这两个看似增加游戏公平性的条件，反而使该游戏成为了一个割韭菜游戏。我们以 `piles=[2, 1, 9, 5]` 讲解，假设这四堆石头从左到右的索引分别是 1, 2, 3, 4。

如果我们把这四堆石头按索引的奇偶分为两组，即第 1、3 堆和第 2、4 堆，那么这两组石头的数量一定不同，也就是说一堆多一堆少。因为石头的总数是奇数，不能被平分。

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。

你最开始可以选择第 1 堆或第 4 堆。如果你想要偶数堆，你就拿第 4 堆，这样留给对手的选择只有第 1、3 堆，他不管怎么拿，第 2 堆又会暴露出来，你就可以拿。同理，如果你想拿奇数堆，你就拿第 1 堆，留给对手的只有第 2、4 堆，他不管怎么拿，第 3 堆又给你暴露出来了。

也就是说，你可以在第一步就观察好，奇数堆的石头总数多，还是偶数堆的石头总数多，然后步步为营，就一切尽在掌控之中了。知道了这个漏洞，可以整一整不知情的同学了。

## 三、电灯开关问题

力扣第 319 题「灯泡开关」的规则是这样的：

有  $n$  盏电灯，最开始时都是关着的。现在要进行  $n$  轮操作：

第 1 轮操作是把每一盏电灯的开关按一下（全部打开）。

第 2 轮操作是把每两盏灯的开关按一下（就是按第 2, 4, 6... 盏灯的开关，它们被关闭）。

第 3 轮操作是把每三盏灯的开关按一下（就是按第 3, 6, 9... 盏灯的开关，有的被关闭，比如 3，有的被打开，比如 6）...

如此往复，直到第  $n$  轮，即只按一下第  $n$  盏灯的开关。

现在给你输入一个正整数  $n$  代表电灯的个数，问你经过  $n$  轮操作后，这些电灯有多少盏是亮的？

我们当然可以用一个布尔数组表示这些灯的开关情况，然后模拟这些操作过程，最后去数一下就能出结果。但是这样显得没有灵性，最好的解法是这样的：

```
int bulbSwitch(int n) {
    return (int)Math.sqrt(n);
}
```

什么？这个问题跟平方根有什么关系？其实这个解法挺精妙，如果没人告诉你解法，还真不好想明白。

首先，因为电灯一开始都是关闭的，所以某一盏灯最后如果是点亮的，必然要被按奇数次开关。

我们假设只有 6 盏灯，而且我们只看第 6 盏灯。需要进行 6 轮操作对吧，请问对于第 6 盏灯，会被按下几次开关呢？这不难得出，第 1 轮会被按，第 2 轮，第 3 轮，第 6 轮都会被按。

为什么第 1、2、3、6 轮会被按呢？因为  $6=1*6=2*3$ 。一般情况下，因子都是成对出现的，也就是说开关被按的次数一般是偶数次。但是有特殊情况，比如说总共有 16 盏灯，那么第 16 盏灯会被按几次？

$16 = 1*16 = 2*8 = 4*4$

其中因子 4 重复出现，所以第 16 盏灯会被按 5 次，奇数次。现在你应该理解这个问题为什么和平方根有关了吧？

不过，我们不是要算最后有几盏灯亮着吗，这样直接平方根一下是啥意思呢？稍微思考一下就能理解了。

就假设现在总共有 16 盏灯，我们求 16 的平方根，等于 4，这就说明最后会有 4 盏灯亮着，它们分别是第  $1*1=1$  盏、第  $2*2=4$  盏、第  $3*3=9$  盏和第  $4*4=16$  盏。

就算有的  $n$  平方根结果是小数，强转成 int 型，也相当于一个最大整数上界，比这个上界小的所有整数，平方后的索引都是最后亮着的灯的索引。所以说我们直接把平方根转成整数，就是这个问题的答案。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 常用的位操作



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">231. Power of Two</a>	<a href="#">231. 2 的幂</a>	
<a href="#">136. Single Number</a>	<a href="#">136. 只出现一次的数字</a>	
<a href="#">268. Missing Number</a>	<a href="#">268. 丢失的数字</a>	
<a href="#">191. Number of 1 Bits</a>	<a href="#">191. 位1的个数</a>	

-----  
位操作（Bit Manipulation）可以有很多技巧，有一个叫做 Bit Twiddling Hacks 的网站收集了几乎所有位操作的黑科技玩法，网址如下：

<http://graphics.stanford.edu/~seander/bithacks.html>

但是这些技巧大部分都过于晦涩，我觉得可以作为字典查阅，没必要逐条深究。但我认为那些有趣的、有用的位运算技巧，是我们每个人需要掌握的。

所以本文由浅入深，先展示几个有趣（但没卵用）的位运算技巧，然后再汇总一些在算法题以及工程开发中常用的位运算技巧。

## 一、几个有趣的位操作

```
// 1. 利用或操作 `|` 和空格将英文字符转换为小写
('a' | ' ') = 'a'
('A' | ' ') = 'a'

// 2. 利用与操作 `&` 和下划线将英文字符转换为大写
('b' & '_') = 'B'
('B' & '_') = 'B'

// 3. 利用异或操作 `^` 和空格进行英文字符大小写互换
('d' ^ ' ') = 'D'
('D' ^ ' ') = 'd'

// 以上操作能够产生奇特效果的原因在于 ASCII 编码
// ASCII 字符其实就是数字，恰巧空格和下划线对应的数字通过位运算就能改变大小写
// 有兴趣的读者可以查 ASCII 码表自己算算，我就不展开讲了

// 4. 不用临时变量交换两个数
int a = 1, b = 2;
```

```
a ^= b;
b ^= a;
a ^= b;
// 现在 a = 2, b = 1

// 5. 加一
int n = 1;
n = ~n;
// 现在 n = 2

// 6. 减一
int n = 2;
n = ~n;
// 现在 n = 1

// 7. 判断两个数是否异号
int x = -1, y = 2;
boolean f = ((x ^ y) < 0); // true

int x = 3, y = 2;
boolean f = ((x ^ y) < 0); // false
```

如果说前 6 个技巧的用处不大，这第 7 个技巧还是比较实用的，利用的是补码编码的符号位。整数编码最高位是符号位，负数的符号位是 1，非负数的符号位是 0，再借助异或的特性，可以判断出两个数字是否异号。

当然，如果不用位运算来判断是否异号，需要使用 if else 分支，还挺麻烦的。你可能想利用乘积来判断两个数是否异号，但是这种处理方式容易造成整型溢出，从而出现错误。

## index & (arr.length - 1) 的运用

我在 [单调栈解题套路](#) 中介绍过环形数组，其实就是利用求模（余数）的方式让数组看起来头尾相接形成一个环形，永远都走不完：

```
int[] arr = {1,2,3,4};
int index = 0;
while (true) {
    // 在环形数组中转圈
    print(arr[index % arr.length]);
    index++;
}
// 输出: 1,2,3,4,1,2,3,4,1,2,3,4...
```

但模运算 % 对计算机来说其实是一个比较昂贵的操作，所以我们可以用 & 运算来求余数：

```
int[] arr = {1,2,3,4};
int index = 0;
while (true) {
    // 在环形数组中转圈
    print(arr[index & (arr.length - 1)]);
    index++;
}
```

```

}
// 输出: 1,2,3,4,1,2,3,4,1,2,3,4...

```

注意这个技巧只适用于数组长度是 2 的幂次方的情况，比如 2、4、8、16、32 以此类推。至于如何将数组长度扩展为 2 的幂次方，这也是有比较巧妙的位运算算法的，可以参考  
<https://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2>

简单说，`& (arr.length - 1)` 这个位运算能够替代 `% arr.length` 的模运算，性能会更好一些。

那问题来了，现在是不断地 `index++`，你做到了循环遍历。但如果不断地 `index--`，还能做到环形数组的效果吗？

答案是，如果你使用 `%` 求模的方式，那么当 `index` 小于 0 之后求模的结果也会出现负数，你需要特殊处理。但通过 `&` 与运算的方式，`index` 不会出现负数，依然可以正常工作：

```

int[] arr = {1,2,3,4};
int index = 0;
while (true) {
    // 在环形数组中转圈
    print(arr[index & (arr.length - 1)]);
    index--;
}
// 输出: 1,4,3,2,1,4,3,2,1,4,3,2,1...

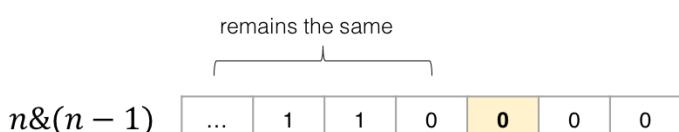
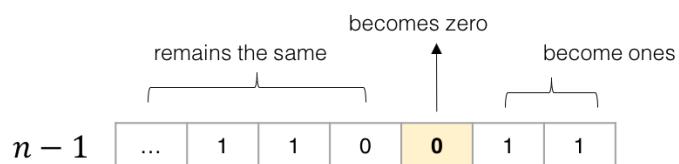
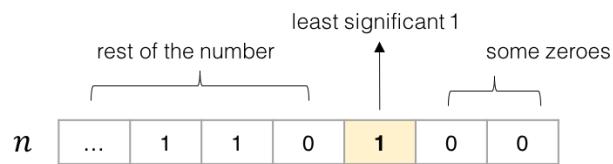
```

我们自己写代码一般用不到这个技巧，但在学习一些其他代码库时可能会经常看到，这里留个印象，到时候就不会懵逼了。

## n & (n-1) 的运用

`n & (n-1)` 这个操作在算法中比较常见，作用是消除数字 `n` 的二进制表示中的最后一个 1。

看个图就很容易理解了：



其核心逻辑就是， $n - 1$  一定可以消除最后一个 1，同时把其后的 0 都变成 1，这样再和  $n$  做一次  $\&$  运算，就可以仅仅把最后一个 1 变成 0 了。

## 计算汉明权重 (Hamming Weight)

这是力扣第 191 题「位 1 的个数」：

### ▼ 191. 位1的个数 [Leetcode | 力扣](#)

编写一个函数，获取一个正整数的二进制形式并返回其二进制表达式中 设置位 的个数（也被称为[汉明重量](#)）。

示例 1：

```
输入: n = 11
输出: 3
解释: 输入的二进制串 1011 中, 共有 3 个设置位。
```

示例 2：

```
输入: n = 128
输出: 1
解释: 输入的二进制串 10000000 中, 共有 1 个设置位。
```

示例 3：

```
输入: n = 2147483645
输出: 30
解释: 输入的二进制串 1111111111111111111111111101 中, 共有 30 个设置位。
```

提示：

- $1 \leq n \leq 2^{31} - 1$

进阶：

- 如果多次调用这个函数，你将如何优化你的算法？

就是让你返回  $n$  的二进制表示中有几个 1。因为  $n \& (n - 1)$  可以消除最后一个 1，所以可以用一个循环不停地消除 1 同时计数，直到  $n$  变成 0 为止。

```
int hammingWeight(int n) {
    int res = 0;
    while (n != 0) {
        n = n & (n - 1);
        res++;
    }
    return res;
}
```

判断 2 的指数

力扣第 231 题「2 的幂」就是这个问题。

一个数如果是 2 的指数，那么它的二进制表示一定只含有一个 1：

```
2^0 = 1 = 0b0001  
2^1 = 2 = 0b0010  
2^2 = 4 = 0b0100
```

如果使用 `n & (n-1)` 的技巧就很简单了（注意运算符优先级，括号不可以省略）：

```
boolean isPowerOfTwo(int n) {  
    if (n <= 0) return false;  
    return (n & (n - 1)) == 0;  
}
```

## a ^ a = 0 的运用

异或运算的性质是需要我们牢记的：

一个数和它本身做异或运算结果为 0，即 `a ^ a = 0`；一个数和 0 做异或运算的结果为它本身，即 `a ^ 0 = a`。

## 查找只出现一次的元素

这是力扣第 136 题「只出现一次的数字」：

### ▼ 136. 只出现一次的数字 [Leetcode | 力扣](#)

给你一个 **非空** 整数数组 `nums`，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

### 示例 1：

```
输入: nums = [2,2,1]  
输出: 1
```

### 示例 2：

```
输入: nums = [4,1,2,1,2]  
输出: 4
```

### 示例 3：

```
输入: nums = [1]  
输出: 1
```

提示：

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$
- 除了某个元素只出现一次以外，其余每个元素均出现两次。

对于这道题目，我们只要把所有数字进行异或，成对儿的数字就会变成 0，落单的数字和 0 做异或还是它本身，所以最后异或的结果就是只出现一次的元素：

```
int singleNumber(int[] nums) {  
    int res = 0;  
    for (int n : nums) {  
        res ^= n;  
    }  
    return res;  
}
```

## 寻找缺失的元素

这是力扣第 268 题「丢失的数字」：

### ▼ 268. 丢失的数字 [Leetcode | 力扣](#)

给定一个包含  $[0, n]$  中  $n$  个数的数组  $\text{nums}$ ，找出  $[0, n]$  这个范围内没有出现在数组中的那个数。

#### 示例 1：

```
输入: nums = [3,0,1]  
输出: 2  
解释: n = 3, 因为有 3 个数字, 所以所有的数字都在范围 [0,3] 内。2 是丢失的数字, 因为它没有出现在  
nums 中。
```

#### 示例 2：

```
输入: nums = [0,1]  
输出: 2  
解释: n = 2, 因为有 2 个数字, 所以所有的数字都在范围 [0,2] 内。2 是丢失的数字, 因为它没有出现在  
nums 中。
```

#### 示例 3：

```
输入: nums = [9,6,4,2,3,5,7,0,1]  
输出: 8  
解释: n = 9, 因为有 9 个数字, 所以所有的数字都在范围 [0,9] 内。8 是丢失的数字, 因为它没有出现在  
nums 中。
```

#### 示例 4：

```
输入: nums = [0]  
输出: 1
```

解释： $n = 1$ ，因为有 1 个数字，所以所有的数字都在范围  $[0, 1]$  内。 $1$  是丢失的数字，因为它没有出现在  $\text{nums}$  中。

提示：

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $0 \leq \text{nums}[i] \leq n$
- $\text{nums}$  中的所有数字都 独一无二

进阶：你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题？

给一个长度为  $n$  的数组，其索引应该在  $[0, n)$ ，但是现在你要装进去  $n + 1$  个元素  $[0, n]$ ，那么肯定有一个元素装不下嘛，请你找出这个缺失的元素。

这道题不难的，我们应该很容易想到，把这个数组排个序，然后遍历一遍，不就很容易找到缺失的那个元素了吗？

或者说，借助数据结构的特性，用一个 HashSet 把数组里出现的数字都储存下来，再遍历  $[0, n]$  之间的数字，去 HashSet 中查询，也可以很容易查出那个缺失的元素。

排序解法的时间复杂度是  $O(N\log N)$ ，HashSet 的解法时间复杂度是  $O(N)$ ，但是还需要  $O(N)$  的空间复杂度存储 HashSet。

这个问题其实还有一个特别简单的解法：等差数列求和公式。

题目的意思可以这样理解：现在有个等差数列  $0, 1, 2, \dots, n$ ，其中少了某一个数字，请你把它找出来。那这个数字不就是  $\text{sum}(0, 1, \dots, n) - \text{sum}(\text{nums})$  嘛？

```
int missingNumber(int[] nums) {  
    int n = nums.length;  
    // 虽然题目给的数据范围不大，但严谨起见，用 long 类型防止整型溢出  
    // 求和公式：(首项 + 末项) * 项数 / 2  
    long expect = (0 + n) * (n + 1) / 2;  
    long sum = 0;  
    for (int x : nums) {  
        sum += x;  
    }  
    return (int)(expect - sum);  
}
```

不过，本文的主题是位运算，我们来讲讲如何利用位运算技巧来解决这道题。

再回顾一下异或运算的性质：一个数和它本身做异或运算结果为 0，一个数和 0 做异或运算还是它本身。

而且异或运算满足交换律和结合律，也就是说：

$$2 \wedge 3 \wedge 2 = 3 \wedge (2 \wedge 2) = 3 \wedge 0 = 3$$

而这也题索就可以通过这些性质巧妙算出缺失的那个元素，比如说  $\text{nums} = [0, 3, 1, 4]$ ：

index 0 1 2 3

nums 0 3 1 4

© labuladong

为了容易理解，我们假设先把索引补一位，然后让每个元素和自己相等的索引相对应：

index 0 1 2 3 4

nums 0 1      3 4

装了个寂寞...

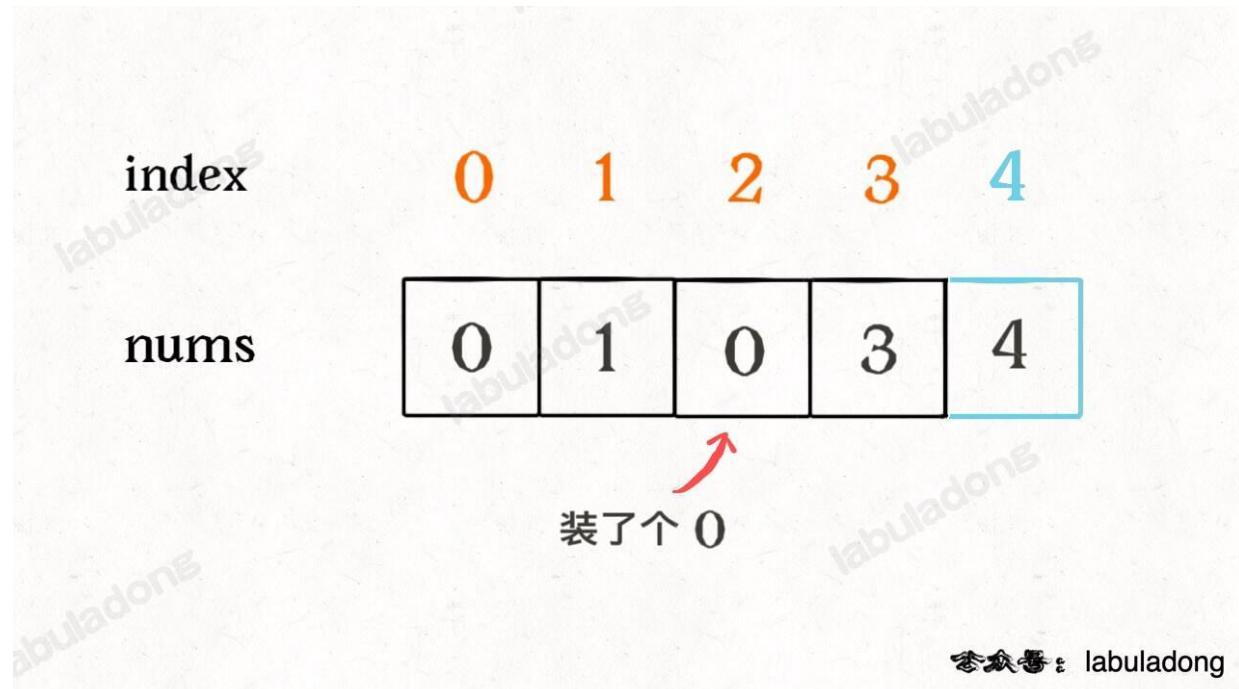
© labuladong

这样做了之后，就可以发现除了缺失元素之外，所有的索引和元素都组成一对儿了，现在如果把这个落单的索引 2 找出来，也就找到了缺失的那个元素。

如何找这个落单的数字呢，只要把所有的元素和索引做异或运算，成对儿的数字都会消为 0，只有这个落单的元素会剩下，也就达到了我们的目的：

```
class Solution {
    public int missingNumber(int[] nums) {
        int n = nums.length;
        int res = 0;
        // 先和新补的索引异或一下
        res ^= n;
        // 和其他的元素、索引做异或
        for (int i = 0; i < n; i++)
            res ^= i ^ nums[i];
    }
}
```

```
        res ^= i ^ nums[i];
    return res;
}
```



由于异或运算满足交换律和结合律，所以总是能把成对儿的数字消去，留下缺失的那个元素。

到这里，常见的位运算差不多都讲完了。这些技巧就是会者不难难者不会，也不需要死记硬背，只要有个印象就完全够用了。

#### ► 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">1457. Pseudo-Palindromic Paths in a Binary Tree</a>	<a href="#">1457. 二叉树中的伪回文路径</a>	●
<a href="#">389. Find the Difference</a>	<a href="#">389. 找不同</a>	●
-	<a href="#">剑指 Offer 15. 二进制中1的个数</a>	●

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 谈谈游戏中的随机算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

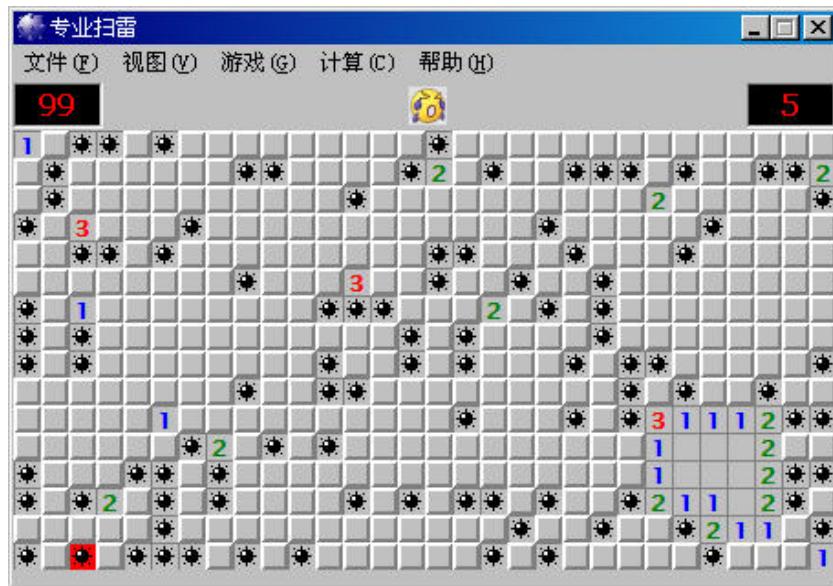
读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">384. Shuffle an Array</a>	<a href="#">384. 打乱数组</a>	●
<a href="#">382. Linked List Random Node</a>	<a href="#">382. 链表随机节点</a>	●
<a href="#">398. Random Pick Index</a>	<a href="#">398. 随机数索引</a>	●

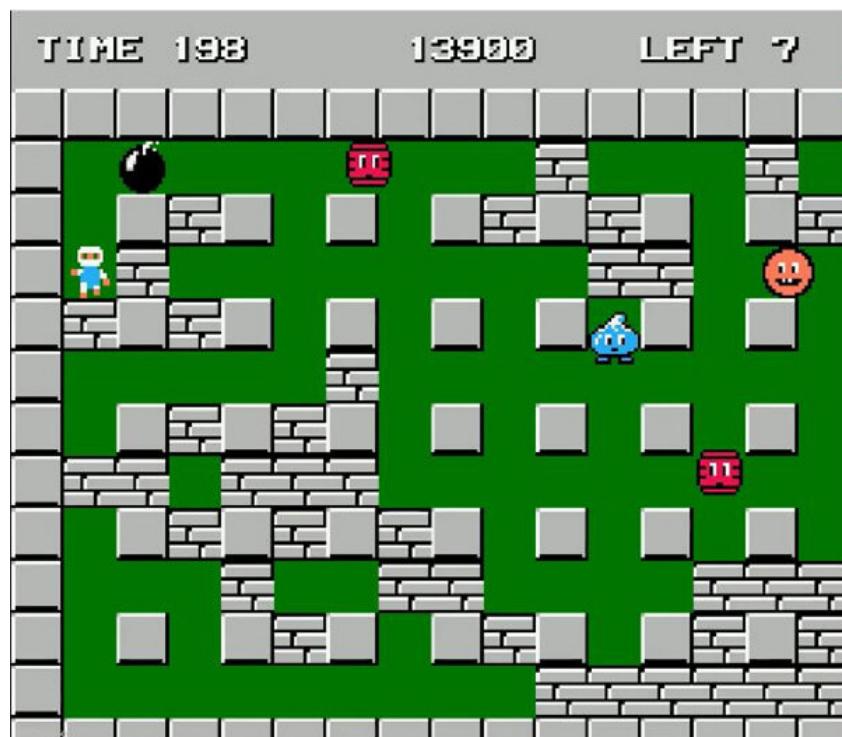
阅读本文前，你需要先学习：

- [数组基础](#)

没事儿的时候我喜欢玩玩那些经典的 2D 网页小游戏，我发现很多游戏都要涉及地图的随机生成，比如扫雷游戏中雷的位置应该是随机分布的：



再比如经典炸弹人游戏，障碍物的位置也是有一定随机性的：



这些 2D 游戏相较现在的大型 3D 游戏虽然看起来有些简陋，但依然用到很多有趣算法技巧，本文就来深入研究一下地图的随机生成算法。

2D 游戏的地图肯定可以抽象成一个二维矩阵，就拿扫雷举例吧，我们可以用下面这个类表示扫雷的棋盘：

```
class Game {  
    int m, n;  
    // 大小为 m * n 的二维棋盘  
    // 值为 true 的地方代表有雷，false 代表没有雷  
    boolean[][] board;  
}
```

如果你想在棋盘中随机生成  $k$  个雷，也就是说你需要在 `board` 中生成  $k$  个不同的  $(x, y)$  坐标，且这里面  $x, y$  都是随机生成的。

对于这个需求，首先一个优化就是对二维矩阵进行「降维打击」，把二维数组转化成一维数组：

```
class Game {  
    int m, n;  
    // 长度为 m * n 的一维棋盘  
    // 值为 true 的地方代表有雷，false 代表没有雷  
    boolean[] board;  
  
    // 将二维数组中的坐标 (x, y) 转化为一维数组中的索引  
    int encode(int x, int y) {  
        return x * n + y;  
    }  
  
    // 将一维数组中的索引转化为二维数组中的坐标 (x, y)  
    int[] decode(int index) {  
        return new int[] {index / n, index % n};  
    }  
}
```

这样，我们只要在  $[0, m * n)$  中选取一个随机数，就相当于在二维数组中随机选取了一个元素。

但问题是，我们现在需要随机选出  $k$  个不同的位置放雷。你可能说，那在  $[0, m * n)$  中选出来  $k$  个随机数不就行了？

是的，但实际操作起来有些麻烦，因为你很难保证随机数不重复。如果出现重复的随机数，你就得再随机选一次，直到找到  $k$  个不同的随机数。

如果  $k$  比较小  $m * n$  比较大，那出现重复随机数的概率还比较低，但如果  $k$  和  $m * n$  的大小接近，那么出现重复随机数的概率非常高，算法的效率就会大幅下降。

那么，我们有没有更好的办法能够在线性的时间复杂度解决这个问题？其实是有的，而且有很多种解决方案。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

## 讲两道常考的阶乘算法题



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">793. Preimage Size of Factorial Zeroes Function</a>	793. 阶乘函数后 K 个零	
<a href="#">172. Factorial Trailing Zeroes</a>	172. 阶乘后的零	

阅读本文前，你需要先学习：

- [二分查找框架详解](#)

笔试题中经常看到阶乘相关的题目，今天说两个最常见的题目：

1、输入一个非负整数  $n$ ，请你计算阶乘  $n!$  的结果末尾有几个 0。

这也是力扣第 172 题「阶乘后的零」，比如说输入  $n = 5$ ，算法返回 1，因为  $5! = 120$ ，末尾有一个 0。

函数签名如下：

```
int trailingZeroes(int n);
```

2、输入一个非负整数  $K$ ，请你计算有多少个  $n$ ，满足  $n!$  的结果末尾恰好有  $K$  个 0。

这也是力扣第 793 题「阶乘后  $K$  个零」，比如说输入  $K = 1$ ，算法返回 5，因为  $5!, 6!, 7!, 8!, 9!$  这 5 个阶乘的结果最后只有一个 0，即有 5 个  $n$  满足条件。

函数签名如下：

```
int preimageSizeFZF(int K);
```

我把这两个题放在一起，肯定是因为它们有共性，下面我们来逐一分析。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 如何高效寻找素数



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">204. Count Primes</a>	<a href="#">204. 计数质数</a>	困难

-----

素数的定义看起来很简单，如果一个数如果只能被 1 和它本身整除，那么这个数就是素数。

虽然素数的定义并不复杂，恐怕没多少人真的能把素数相关的算法写得高效。

比如力扣第 204 题「计数质数」，让你写这样一个函数：

```
// 返回区间 [2, n) 中有几个素数
int countPrimes(int n)

// 比如 countPrimes(10) 返回 4
// 因为 2,3,5,7 是素数
```

你会如何写这个函数？我想大家应该会这样写：

```
int countPrimes(int n) {
    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrime(i)) count++;
    return count;
}

// 判断整数 n 是否是素数
boolean isPrime(int n) {
    for (int i = 2; i < n; i++)
        if (n % i == 0)
            // 有其他整除因子
            return false;
    return true;
}
```

这样写的话时间复杂度  $O(n^2)$ ，问题很大。首先你用 `isPrime` 函数来辅助的思路就不够高效；而且就算你要用 `isPrime` 函数，这样写算法也是存在计算冗余的。

先来说下如果你要判断一个数是不是素数，应该如何写算法。只需稍微修改一下上面的 `isPrime` 代码中的 for 循环条件：

```
boolean isPrime(int n) {
    for (int i = 2; i * i <= n; i++)
    ...
}
```

换句话说，`i` 不需要遍历到 `n`，而只需要到 `sqrt(n)` 即可。为什么呢，我们举个例子，假设 `n = 12`。

```
12 = 2 × 6
12 = 3 × 4
12 = sqrt(12) × sqrt(12)
12 = 4 × 3
12 = 6 × 2
```

可以看到，后两个乘积就是前面两个反过来，反转临界点就在 `sqrt(n)`。

换句话说，如果在 `[2, sqrt(n)]` 这个区间之内没有发现可整除因子，就可以直接断定 `n` 是素数了，因为在区间 `[sqrt(n), n]` 也一定不会发现可整除因子。

现在，`isPrime` 函数的时间复杂度降为  $O(\sqrt{N})$ ，但是我们实现 `countPrimes` 函数其实并不需要这个函数，以上只是希望读者明白 `sqrt(n)` 的含义，因为等会还会用到。

## 高效实现 `countPrimes`

接下来介绍的方法叫做「素数筛选法」，这个方法是古希腊一位名叫埃拉托色尼的大佬发明的，我们在中学的教课书上见过他的大名，因为他就是第一个通过物体的影子正确计算地球周长的人，被推崇为「地理学之父」。

回到正题，素数筛选法的核心思路是和上面的常规思路反着来：

首先从 2 开始，我们知道 2 是一个素数，那么  $2 \times 2 = 4, 3 \times 2 = 6, 4 \times 2 = 8 \dots$  都不可能是素数了。

然后我们发现 3 也是素数，那么  $3 \times 2 = 6, 3 \times 3 = 9, 3 \times 4 = 12 \dots$  也都不可能是素数了。

Wikipedia 的这个 GIF 很形象：

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

看到这里，你是否有点明白这个排除法的逻辑了呢？先看我们的第一版代码：

```
class Solution {
    public int countPrimes(int n) {
        boolean[] isPrime = new boolean[n];
        // 将数组都初始化为 true
        Arrays.fill(isPrime, true);

        for (int i = 2; i < n; i++) {
            if (isPrime[i]) {
                // i 的倍数不可能是素数了
                for (int j = 2 * i; j < n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        int count = 0;
        for (int i = 2; i < n; i++) {
            if (isPrime[i]) count++;
        }

        return count;
    }
}
```

如果上面这段代码你能够理解，那么你已经掌握了整体思路，但是还有两个细微的地方可以优化。

首先，回想本文开头介绍的 `isPrime` 素数判定函数，由于因子的对称性，其中的 `for` 循环只需要遍历  $[2, \sqrt{n}]$  就够了。这里也是类似的，我们外层的 `for` 循环也需要遍历到  $\sqrt{n}$ ：

```
for (int i = 2; i * i < n; i++)
    if (isPrime[i])
        ...
```

除此之外，很难注意到内层的 `for` 循环也可以优化。我们之前的做法是：

```
for (int j = 2 * i; j < n; j += i)
    isPrime[j] = false;
```

这样可以把 `i` 的整数倍都标记为 `false`，但是仍然存在计算冗余。

比如 `n = 25, i = 5` 时算法会标记  $5 \times 2 = 10, 5 \times 3 = 15$  等等数字，但是这两个数字已经被 `i = 2` 和 `i = 3` 的  $2 \times 5$  和  $3 \times 5$  标记了。

我们可以稍微优化一下，让 `j` 从 `i * i` 开始遍历，而不是从 `2 * i` 开始：

```
for (int j = i * i; j < n; j += i)
    isPrime[j] = false;
```

这样，素数计数的算法就高效实现了，其实这个算法有一个名字，叫做 Sieve of Eratosthenes。看下完整的最终代码：

```
class Solution {
    public int countPrimes(int n) {
        boolean[] isPrime = new boolean[n];
        Arrays.fill(isPrime, true);
        for (int i = 2; i * i < n; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j < n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        int count = 0;
        for (int i = 2; i < n; i++) {
            if (isPrime[i]) count++;
        }

        return count;
    }
}
```

### ▶ 代码可视化动画

该算法的时间复杂度比较难算，显然时间跟这两个嵌套的 for 循环有关，其操作数应该是：

$$n/2 + n/3 + n/5 + n/7 + \dots = n \times (1/2 + 1/3 + 1/5 + 1/7\dots)$$

括号中是素数的倒数。其最终结果是  $O(N * \log \log N)$ ，有兴趣的读者可以查一下该算法的时间复杂度证明。

以上就是素数算法相关的全部内容。怎么样，是不是看似简单的问题却有不少细节可以打磨呀？

### ▶ 引用本文的题目

安装 [我的 Chrome 刷题插件](#) 点开下列题目可直接查看解题思路：

LeetCode	力扣	难度
<a href="#">264. Ugly Number II</a>	<a href="#">264. 丑数 II</a>	
-	<a href="#">剑指 Offer 49. 丑数</a>	

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 如何高效进行模幂运算



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">372. Super Pow</a>	<a href="#">372. 超级次方</a>	困难

今天来聊一道与数学运算有关的题目，力扣第 372 题「超级次方」，让你进行巨大的幂运算，然后求余数。

```
int superPow(int a, int[] b);
```

要求你的算法返回幂运算  $a^b$  的计算结果与 1337 取模（mod，也就是余数）后的结果。就是你先得计算幂  $a^b$ ，但是这个  $b$  会非常大，所以  $b$  是用数组的形式表示的。

这个算法其实就是广泛应用于离散数学的模幂算法，至于为什么要对 1337 求模我们不管，单就这道题可以有三个难点：

**一是如何处理用数组表示的指数**，现在  $b$  是一个数组，也就是说  $b$  可以非常大，没办法直接转成整型，否则可能溢出。你怎么把这个数组作为指数，进行运算呢？

**二是如何得到求模之后的结果**？按道理，起码应该先把幂运算结果算出来，然后做  $\% 1337$  这个运算。但问题是，指数运算你懂得，真实结果肯定会大得吓人，也就是说，算出来真实结果也没办法表示，早都溢出报错了。

**三是如何高效进行幂运算**，进行幂运算也是有算法技巧的，如果你不了解这个算法，后文会讲解。

那么对于这几个问题，我们分开思考，逐个击破。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 如何同时寻找缺失和重复的元素



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">645. Set Mismatch</a>	<a href="#">645. 错误的集合</a>	●

今天就聊一道很看起来简单却十分巧妙的问题，寻找缺失和重复的元素。之前的一篇文章 [常用的位操作](#) 中也写过类似的问题，不过这次的和上次的问题使用的技巧不同。

这是力扣第 645 题「错误的集合」，我来描述一下这个题目：

给一个长度为  $N$  的数组  $\text{nums}$ ，其中本来装着  $[1..N]$  这  $N$  个元素，无序。但是现在出现了一些错误， $\text{nums}$  中的一个元素出现了重复，也就同时导致了另一个元素的缺失。请你写一个算法，找到  $\text{nums}$  中的重复元素和缺失元素的值。

```
// 返回两个数字，分别是 {dup, missing}
int[] findErrorNums(int[] nums);
```

比如说输入： $\text{nums} = [1, 2, 2, 4]$ ，算法返回  $[2, 3]$ 。

其实很容易解决这个问题，先遍历一次数组，用一个哈希表记录每个数字出现的次数，然后遍历一次  $[1..N]$ ，看看那个元素重复出现，那个元素没有出现，就 OK 了。

但问题是，这个常规解法需要一个哈希表，也就是  $O(N)$  的空间复杂度。你看题目给的条件那么巧，在  $[1..N]$  的几个数字中恰好有一个重复，一个缺失，事出反常必有妖，对吧。

$O(N)$  的时间复杂度遍历数组是无法避免的，所以我们可以想想办法如何降低空间复杂度，是否可以在  $O(1)$  的空间复杂度之下找到重复和缺失的元素呢？

本文为 [labuladong.online](https://labuladong.online) 网站会员内容，请 [点这里](#) 查看。

# 几个反直觉的概率问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](http://labuladong.online)。

-----  
上篇文章 [谈谈游戏中的随机算法](#) 讲到了验证概率算法的蒙特卡罗方法，今天聊点轻松的内容：几个和概率相关的有趣问题。

计算概率有下面两个最简单的原则：

原则一、计算概率一定要有一个参照系，称作「样本空间」，即随机事件可能出现的所有结果。事件 A 发生的概率 = A 包含的样本点 / 样本空间的样本总数。

原则二、计算概率一定要明白，概率是一个连续的整体，不可以把连续的概率分割开，也就是所谓的条件概率。

上述两个原则高中就学过，但是我们还是很容易犯错，而且犯错的流程也有异曲同工之妙：

先是忽略了原则二，错误地计算了样本空间，然后通过原则一算出了错误的答案。

下面介绍几个简单却具有迷惑性的问题，分别是男孩女孩问题、生日悖论、三门问题。当然，三门问题可能是大家最耳熟的，所以就多说一些有趣的思考。

## 一、男孩女孩问题

假设有一个家庭，有两个孩子，现在告诉你其中有一个男孩，请问另一个也是男孩的概率是多少？

很多人，包括我在内，不假思索地回答：1/2 啊，因为另一个孩子要么是男孩，要么是女孩，而且概率相等呀。但是实际上，答案是 1/3。

上述思想为什么错误呢？因为没有正确计算样本空间，导致原则一计算错误。有两个孩子，那么样本空间为 4，即哥哥妹妹，哥哥弟弟，姐姐妹妹，姐姐弟弟这四种情况。已知有一个男孩，那么排除姐姐妹妹这种情况，所以样本空间变成 3。另一个孩子也是男孩只有哥哥弟弟这 1 种情况，所以概率为 1/3。

为什么计算样本空间会出错呢？因为我们忽略了条件概率，即混淆了下面两个问题：

这个家庭只有一个孩子，这个孩子是男孩的概率是多少？

这个家庭有两个孩子，其中一个是男孩，另一个孩子是男孩的概率是多少？

根据原则二，概率问题是连续的，不可以把上述两个问题混淆。第二个问题需要用条件概率，即求一个孩子是男孩的条件下，另一个也是男孩的概率。运用条件概率的公式也很好算，就不多说了。

通过这个问题，读者应该理解两个概率计算原则的关系了，最具有迷惑性的就是条件概率的忽视。为了不要被迷惑，最简单的办法就是把所有可能结果穷举出来。

最后，对于此问题我见过一个很奇葩的质疑：如果这两个孩子是双胞胎，不存在年龄上的差异怎么办？

我竟然觉得有那么一丝道理！但其实，我们只是通过年龄差异来表示两个孩子的独立性，也就是说即便两个孩子同性，也有两种可能。所以不要用双胞胎抬杠了。

## 二、生日悖论

生日悖论是由这样一个问题引出的：一个屋子里需要有多少人，才能使得存在至少两个人生日是同一天的概率达到 50%？

答案是 23 个人，也就是说房子里如果有 23 个人，那么就有 50% 的概率会存在两个人生日相同。这个结论看起来不可思议，所以被称为悖论。按照直觉，要得到 50% 的概率，起码得有 183 个人吧，因为一年有 365 天呀？其实不是的，觉得这个结论不可思议主要有两个思维误区：

**第一个误区是误解「存在」这个词的含义。**

读者可能认为，如果 23 个人中出现相同生日的概率就能达到 50%，是不是意味着：

假设现在屋子里坐着 22 个人，然后我走进去，那么有 50% 的概率我可以找到一个人和我生日相同。但这怎么可能呢？

并不是的，你这种想法是以自我为中心，而题目的概率是在描述整体。也就是说「存在」的含义是指 23 人中的任意两个人，涉及排列组合，大概率和你没啥关系。

如果你非要计算存在和自己生日相同的人的概率是多少，可以这样计算：

$$1 - P(22 \text{ 个人都和我的生日不同}) = 1 - (364/365)^{22} = 0.06$$

这样计算得到的结果是不是看起来合理多了？生日悖论计算对象的不是某一个人，而是一个整体，其中包含了所有人的排列组合，它们的概率之和当然会大得多。

**第二个误区是认为概率是线性变化的。**

读者可能认为，如果 23 个人中出现相同生日的概率就能达到 50%，是不是意味着 46 个人的概率就能达到 100%？

不是的，就像中奖率 50% 的游戏，你玩两次的中奖率就是 100% 吗？显然不是，你玩两次的中奖率是 75%：

$$P(\text{两次能中奖}) = P(\text{第一次就中了}) + P(\text{第一次没中但第二次中了}) = 1/2 + 1/2 * 1/2 = 75\%$$

那么换到生日悖论也是一个道理，概率不是简单叠加，而要考虑一个连续的过程，所以这个结论并没有什么不合常理之处。

那为什么只要 23 个人出现相同生日的概率就能大于 50% 了呢？我们先计算 23 个人生日都唯一（不重复）的概率。只有 1 个人的时候，生日唯一的概率是  $365/365$ ，2 个人时，生日唯一的概率是  $365/365 \times 364/365$ ，以此类推可知 23 人的生日都唯一的概率：

$$P(A') = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{343}{365}$$

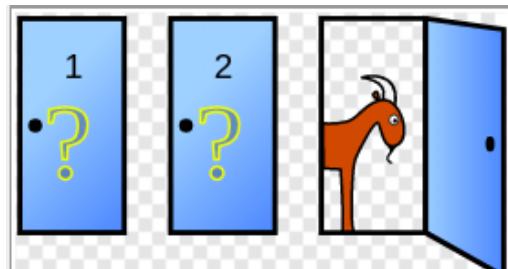
算出来大约是 0.493，所以存在相同生日的概率就是 0.507，差不多就是 50% 了。实际上，按照这个算法，当人数达到 70 时，存在两个人生日相同的概率就上升到了 99.9%，基本可以认为是 100% 了。所以从概率上说，一个几十人的小团体中存在生日相同的人真没啥稀奇的。

## 三、三门问题

这个游戏很经典了：游戏参与者面对三扇门，其中两扇门后面是山羊，一扇门后面是跑车。参与者只要随便选一扇门，门后面的东西就归他（跑车的价值当然更大）。但是主持人决定帮一下参与者：在他选择之后，先不急着打开这扇门，而是由主持人打开剩下两扇门中的一扇，展示其中的山羊（主持人知道每扇门后面是什么），然后给参与者一次换门的机会，此时参与者应该换门还是不换门呢？

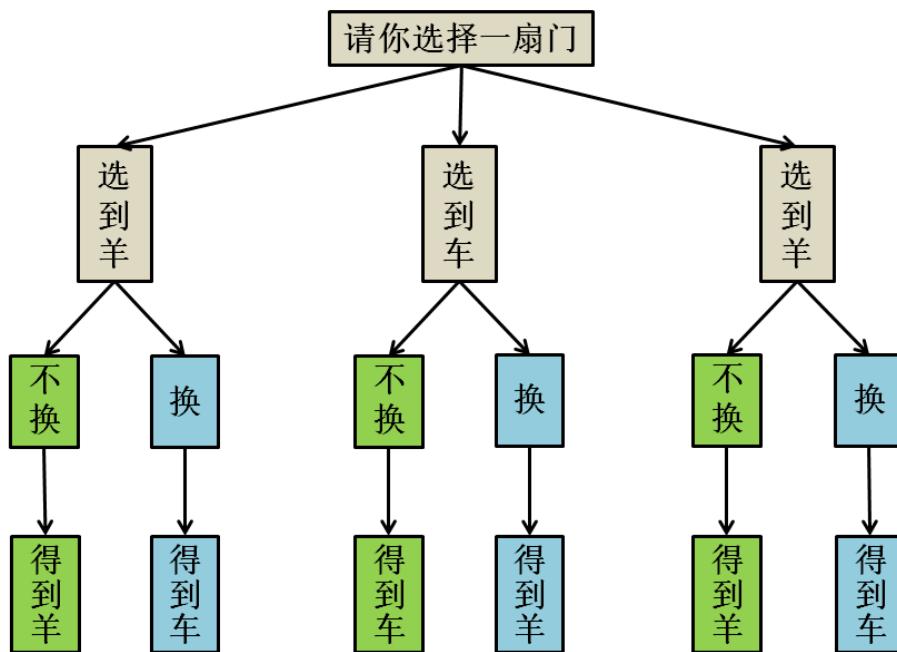
为了防止第一次看到这个问题的读者迷惑，再具体描述一下这个问题：

你是游戏参与者，现在有门 1,2,3，假设你随机选择了门 1，然后主持人打开了门 3 告诉你那后面是山羊。现在，你是坚持你最初的选择门 1，还是选择换成门 2 呢？



答案是应该换门，换门之后抽到跑车的概率是  $2/3$ ，不换的话是  $1/3$ 。又一次反直觉，感觉换不换的中奖概率应该都一样啊，因为最后肯定就剩两个门，一个是羊，一个是跑车，这是事实，所以不管选哪个的概率不都是  $1/2$  吗？

类似前面说的男孩女孩问题，最简单稳妥的方法就是把所有可能结果穷举出来：



很容易看到选择换门中奖的概率是  $2/3$ ，不换的话是  $1/3$ 。

关于这个问题还有更简单的方法：主持人开门实际上在「浓缩」概率。一开始你选择到跑车的概率当然是  $1/3$ ，剩下两个门中包含跑车的概率当然是  $2/3$ ，这没啥可说的。但是主持人帮你排除了一个含有山羊的门，相当于把那  $2/3$  的概率浓缩到了剩下的这一扇门上。那么，你说你是抱着原来那扇  $1/3$  的门，还是换成那扇经过「浓缩」的  $2/3$  概率的门呢？

再直观一点，假设你三选一，剩下 2 扇门，再给你加入 98 扇装山羊的门，把这 100 扇门随机打乱，问你换不换？肯定不换对吧，这明摆着把概率稀释了，肯定抱着原来的那扇门是最可能中跑车的。再假设，初始有 100 扇门，你选了一扇，然后主持人在剩下 99 扇门中帮你排除 98 个山羊，问你换不换一扇门？肯定换对吧，你手上那扇门是  $1\%$ ，另一扇门是  $99\%$ ，或者也可以这样理解，不换只是选择了 1 扇门，换门相当于选择了 99 扇门，这样结果很明显了吧？

以上思想，也许有的读者都思考过，下面我们思考这样一个问题：假设你在决定是否换门的时候，小明破门而入，要求帮你做出选择。他完全不知道之前发生的事，他只知道面前有两扇门，一扇是跑车一扇是山羊，那么他抽中跑车的概率有多大？

当然是  $1/2$ ，这也是很多人做错三门问题的根本原因。类似生日悖论，人们总是容易以自我为中心，通过这个小明的视角来计算是否换门，这显然会进入误区。

就好比有两个箱子，一号箱子有 4 个黑球 2 个红球，二号箱子有 2 个黑球 4 个红球，随便选一个箱子，随便摸一个球，问你摸出红球的概率。

对于不知情的小明，他会随机选择一个箱子，随机摸球，摸到红球的概率是： $1/2 \times 2/6 + 1/2 \times 4/6 = 1/2$

对于知情的你，你知道在二号箱子摸球概率大，所以只在二号箱摸，摸到红球的概率是： $0 \times 2/6 + 1 \times 4/6 = 2/3$

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

## 【强化练习】数学技巧相关习题

---

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 算法笔试「骗分」套路



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

首先回答一个问题，刷力扣题是直接在网页上刷比较好还是在本地 IDE 上刷比较好？

如果是牛客网笔试那种自己处理输入输出的判题形式，一定要在 IDE 上写，这个没啥说的，但像力扣这种判题形式，我个人偏好直接在网页上刷，原因有二：

## 1、方便

因为力扣有的数据结构是自定的，比如说 `TreeNode`, `ListNode` 这种，在本地你还得把这个类 copy 过去。

而且在 IDE 上没办法测试，写完代码之后还得粘贴到网页上跑测试数据，那还不如直接网页上写呢。

算法又不是工程代码，量都比较小，IDE 的自动补全带来的收益基本可以忽略不计。

## 2、实用

到时候面试的时候，面试官给你出的算法题大都是希望你直接在网页上完成的，最好是边写边讲你的思路。

如果平时练习的时候就习惯没有 IDE 的自动补全，习惯手写代码大脑编译，到时候面试的时候写代码就能更快更从容。

之前我面快手的时候，有个面试官让我 [实现 LRU 算法](#)，我直接把双链表的实现、哈希链表的实现，在网页上全写出来了，而且一次无 bug 跑通，可以看到面试官惊讶的表情😊

我秋招能当 offer 收割机，很大程度上就是因为手写算法这一关超出面试官的预期，其实都是因为之前在网页上刷题练出来的。

当然，实在不想在网页上刷，也可以用我的 vscode 刷题插件或者 JetBrains 刷题插件，插件和我的网站内容都有完美的融合：

## labuladong 的刷题全家桶（2023 新版）



公众号



算法秘籍/刷题笔记PDF



更多精品课程



Chrome 插件手册



vscode 插件手册



JetBrains 插件手册

算法学习网站：<https://labuladong.gitbook.io/algo/> 或 <https://labuladong.github.io/algo/>

二维码太多，长按识别不到？放大图片 -> 让对应二维码充满屏幕 -> 长按识别该二维码

接下来介绍几个很实用的「投机取巧」的办法和调试技巧，全方位提高你通过笔试的概率。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 如何高效解决接雨水问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">42. Trapping Rain Water</a>	42. 接雨水	●
<a href="#">11. Container With Most Water</a>	11. 盛最多水的容器	●

阅读本文前，你需要先学习：

- 数组双指针技巧汇总

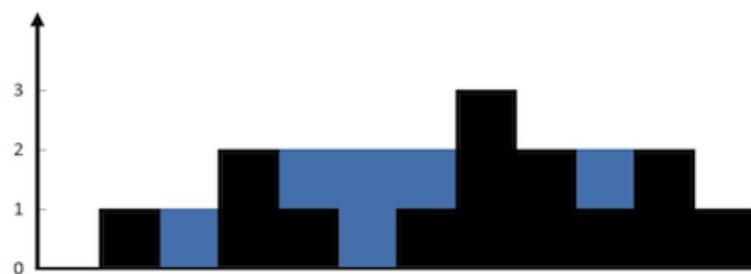
力扣第 42 题「接雨水」挺有意思，在面试题中出现频率还挺高的，本文就来步步优化，讲解一下这道题。

先看一下题目：

## ▼ 42. 接雨水 Leetcode | 力扣

给定  $n$  个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：



输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]

输出：9

提示：

- $n == \text{height.length}$
- $1 \leq n \leq 2 * 10^4$
- $0 \leq \text{height}[i] \leq 10^5$

就是用一个数组表示一个条形图，问你这个条形图最多能接多少水。

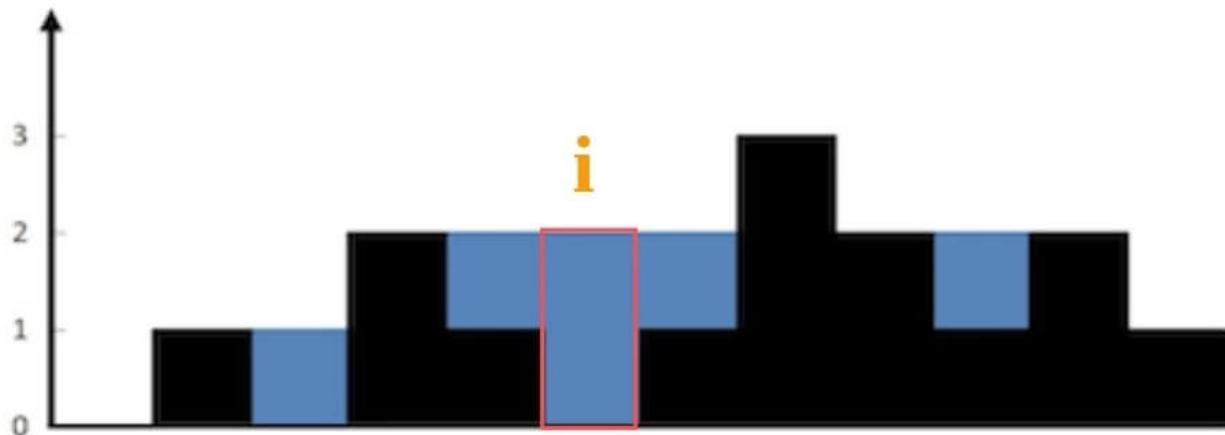
```
int trap(int[] height);
```

下面就来由浅入深介绍暴力解法 -> 备忘录解法 -> 双指针解法，在  $O(N)$  时间  $O(1)$  空间内解决这个问题。

## 一、核心思路

做算法题，如果对题目提出的问题没有思路，不妨尝试化简问题，先从局部思考，先写出最简单粗暴的解法，也许会有突破点。逐步优化后也许就能找到最优解。

比如这道题，先不考虑整个柱状图能装多少水，仅仅考虑位置  $i$  这一个位置能装下多少水？



能装 2 格水，因为  $\text{height}[i]$  的高度为 0，而这里最多能盛 2 格水， $2-0=2$ 。

为什么位置  $i$  最多能盛 2 格水呢？因为，位置  $i$  能达到的水柱高度和其左边的最高柱子、右边的最高柱子有关，我们分别称这两个柱子高度为  $l_{\max}$  和  $r_{\max}$ ；位置  $i$  最大的水柱高度就是  $\min(l_{\max}, r_{\max})$ 。

也就是说，对于位置  $i$ ，能够装的水为：

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 一文秒杀所有丑数系列问题



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">264. Ugly Number II</a>	<a href="#">264. 丑数 II</a>	●
<a href="#">313. Super Ugly Number</a>	<a href="#">313. 超级丑数</a>	●
<a href="#">1201. Ugly Number III</a>	<a href="#">1201. 丑数 III</a>	●
<a href="#">263. Ugly Number</a>	<a href="#">263. 丑数</a>	●

阅读本文前，你需要先学习：

- [链表双指针技巧汇总](#)
- [二分查找框架详解](#)

最近读者群里有个读者跟我私信，说去面试微软遇到了一系列和数学相关的算法题，直接懵圈了。我看了下题目发现这些题其实就是 LeetCode 上面「丑数」系列问题的修改版。

首先，「丑数」系列问题属于会者不难难者不会的类型，因为会用到些数学定理嘛，如果没有专门学过，靠自己恐怕是想不出来的。

另外，这类问题而且非常考察抽象联想能力，因为它不仅仅要用到数学定理，还需要你把题目抽象成链表相关的题目运用双指针技巧，或者抽象成数组相关的题目运用二分搜索技巧。

那么今天我就来用一篇文章把所有丑数相关的问题一网打尽，看看这类问题能够如何变化，应该如何解决。

## 丑数 I

首先是力扣第 263 题「丑数」，题目给你输入一个数字  $n$ ，请你判断  $n$  是否为「丑数」。所谓「丑数」，就是只包含质因数 2、3 和 5 的正整数。

函数签名如下：

```
boolean isUgly(int n)
```

比如  $12 = 2 \times 2 \times 3$  就是一个丑数，而  $42 = 2 \times 3 \times 7$  就不是一个丑数。

这道题其实非常简单，前提是你知道算术基本定理（正整数唯一分解定理）：

任意一个大于 1 的自然数，要么它本身就是质数，要么它可以分解为若干质数的乘积。

既然任意一个大于一的正整数都可以分解成若干质数的乘积，那么丑数也可以被分解成若干质数的乘积，且这些质数只能是 2, 3 或 5。

有了这个思路，就可以实现 `isUgly` 函数了：

```
class Solution {
    public boolean isUgly(int n) {
        if (n <= 0) return false;
        // 如果 n 是丑数，分解因子应该只有 2, 3, 5
        while (n % 2 == 0) n /= 2;
        while (n % 3 == 0) n /= 3;
        while (n % 5 == 0) n /= 5;
        // 如果能够成功分解，说明是丑数
        return n == 1;
    }
}
```

## 丑数 II

接下来提升难度，看下力扣第 264 题「丑数 II」，现在题目不是让你判断一个数是不是丑数，而是给你输入一个 `n`，让你计算第 `n` 个丑数是多少，函数签名如下：

```
int nthUglyNumber(int n)
```

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 分治算法详解：运算优先级



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">241. Different Ways to Add Parentheses</a>	<a href="#">241. 为运算表达式设计优先级</a>	简单

在 [手把手带你刷二叉树（纲领篇）](#) 中，我说递归算法主要有两种思路，一种是「遍历」的思路，另一种是「分解问题」的思路。我说「遍历」思路的典型代表是回溯算法，「分解问题」思路的典型代表是动态规划。

之所以说动态规划是「分解问题」思路典型代表，主要还是因为大家比较熟悉这类算法。但实际上，很多算法问题不具备动态规划问题的重叠子问题、最优子结构等特性，但都可以用「分解问题」的思路来解决，我们可以给这些算法一个高大上的名字，统称为「分治算法」。

最典型的分治算法就是 [归并排序](#) 了，核心逻辑如下：

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

「对数组排序」是一个可以运用分治思想的算法问题，只要我先把数组的左半部分排序，再把右半部分排序，最后把两部分合并，不就是对整个数组排序了吗？

下面来看一道具体的算法题。

## 添加括号的所有方式

我来借力扣第 241 题「为运算表达式设计优先级」来讲讲什么是分治算法，先看看题目：

### ▼ 241. 为运算表达式设计优先级 [Leetcode](#) | [力扣](#)

给你一个由数字和运算符组成的字符串 `expression`，按不同优先级组合数字和运算符，计算并返回所有可能组合的结果。你可以 [按任意顺序](#) 返回答案。

生成的测试用例满足其对应输出值符合 32 位整数范围，不同结果的数量不超过  $10^4$ 。

示例 1：

```
输入: expression = "2-1-1"
输出: [0,2]
解释:
((2-1)-1) = 0
(2-(1-1)) = 2
```

示例 2：

```
输入: expression = "2*3-4*5"
输出: [-34,-14,-10,-10,10]
解释:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

提示：

- $1 \leq \text{expression.length} \leq 20$
- $\text{expression}$  由数字和算符 `%%%%`、`%%%%-%` 和 `%%%%*%` 组成。
- 输入表达式中的所有整数值在范围 `[0, 99]`

简单说，就是给你输入一个算式，你可以给它随意加括号，请你穷举出所有可能的加括号方式，并计算出对应的结果。

函数签名如下：

```
// 计算所有加括号的结果
List<Integer> diffWaysToCompute(String input);
```

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 一个方法解决三道区间问题



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">1288. Remove Covered Intervals</a>	<a href="#">1288. 删除被覆盖区间</a>	●
<a href="#">986. Interval List Intersections</a>	<a href="#">986. 区间列表的交集</a>	●
<a href="#">56. Merge Intervals</a>	<a href="#">56. 合并区间</a>	●

经常有读者问区间相关的问题，今天写一篇文章，秒杀三道区间相关的问题。

所谓区间问题，就是线段问题，让你合并所有线段、找出线段的交集等等。主要有两个技巧：

**1、排序。**常见的排序方法就是按照区间起点排序，或者先按照起点升序排序，若起点相同，则按照终点降序排序。当然，如果你非要按照终点排序，无非对称操作，本质都是一样的。

**2、画图。**就是说不要偷懒，勤动手，两个区间的相对位置到底有几种可能，不同的相对位置我们的代码应该怎么去处理。

废话不多说，下面我们来做题。

本文为 [labuladong.online](#) 网站会员内容，请 [点这里](#) 查看。

# 谁能想到，斗地主也能玩出算法



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">659. Split Array into Consecutive Subsequences</a>	<a href="#">659. 分割数组为连续子序列</a>	困难

-----

斗地主中，大小连续的牌可以作为顺子，有时候我们把对子拆掉，结合单牌，可以组合出更多的顺子，可能更容易赢。

那么如何合理拆分手上的牌，合理地拆出顺子呢？我们今天看一道非常有意思的算法题，连续子序列的划分问题。

这是力扣第 659 题「分割数组为连续子序列」，题目很简单：

给你输入一个升序排列的数组 `nums`（可能包含重复数字），请你判断 `nums` 是否能够被分割成若干个长度至少为 3 的子序列，每个子序列都由连续的整数组成。

函数签名如下：

```
boolean isPossible(int[] nums);
```

比如题目举的例子，输入 `nums = [1,2,3,3,4,4,5,5]`，算法返回 `true`。

因为 `nums` 可以被分割成 `[1,2,3,4,5]` 和 `[3,4,5]` 两个包含连续整数子序列。

但如果输入 `nums = [1,2,3,4,4,5]`，算法返回 `false`，因为无法分割成两个长度至少为 3 的连续子序列。

对于这种涉及连续整数的问题，应该条件反射地想到排序，不过题目说了，输入的 `nums` 本就是排好序的。

那么，我们如何判断 `nums` 是否能够被划分成若干符合条件的子序列呢？

类似前文 回溯算法进行集合划分，我们想把 `nums` 的元素划分到若干个子序列中，其实就是下面这个代码逻辑：

```
for (int v : nums) {  
    if (...) {  
        // 将 v 分配到某个子序列中  
    } else {  
        // 实在无法分配 v  
        return false;  
    }  
    return true;  
}
```

关键在于，我们怎么知道当前元素  $v$  如何进行分配呢？

肯定得分情况讨论，把情况讨论清楚了，题目也就做出来了。

总共有两种情况：

1、当前元素  $v$  自成一派，「以自己开头」构成一个长度至少为 3 的序列。

比如输入  $\text{nums} = [1, 2, 3, 6, 7, 8]$ ，遍历到元素 6 时，它只能自己开头形成一个符合条件的子序列  $[6, 7, 8]$ 。

2、当前元素  $v$  接到已经存在的子序列后面。

比如输入  $\text{nums} = [1, 2, 3, 4, 5]$ ，遍历到元素 4 时，它只能接到已经存在的子序列  $[1, 2, 3]$  后面。它没办法自成开头形成新的子序列，因为少了个 6。

但是，如果这两种情况都可以，应该如何选择？

比如说，输入  $\text{nums} = [1, 2, 3, 4, 5, 5, 6, 7]$ ，对于元素 4，你说它应该形成一个新的子序列  $[4, 5, 6]$  还是接到子序列  $[1, 2, 3]$  后面呢？

显然， $\text{nums}$  数组的正确划分方法是分成  $[1, 2, 3, 4, 5]$  和  $[5, 6, 7]$ ，所以元素 4 应该优先判断自己是否能够接到其他序列后面，如果不可以，再判断是否可以作为新的子序列开头。

这就是整体的思路，想让算法代码实现这两个选择，需要两个哈希表来做辅助：

$\text{freq}$  哈希表帮助一个元素判断自己是否能够作为开头， $\text{need}$  哈希表帮助一个元素判断自己是否可以被接到其他序列后面。

$\text{freq}$  记录每个元素出现的次数，比如  $\text{freq}[3] == 2$  说明元素 3 在  $\text{nums}$  中出现了 2 次。

那么如果我发现  $\text{freq}[3]$ ,  $\text{freq}[4]$ ,  $\text{freq}[5]$  都是大于 0 的，那就说明元素 3 可以作为开头组成一个长度为 3 的子序列。

$\text{need}$  记录哪些元素可以被接到其他子序列后面。

比如说现在已经组成了两个子序列  $[1, 2, 3, 4]$  和  $[2, 3, 4]$ ，那么  $\text{need}[5]$  的值就应该是 2，说明对元素 5 的需求为 2。

明白了这两个哈希表的作用，我们就可以看懂解法了：

```
class Solution {
    public boolean isPossible(int[] nums) {
        Map<Integer, Integer> freq = new HashMap<>();
        Map<Integer, Integer> need = new HashMap<>();

        // 统计 nums 中元素的频率
        for (int v : nums) {
            freq.put(v, freq.getOrDefault(v, 0) + 1);
        }

        for (int v : nums) {
            if (freq.get(v) == 0) {
                // 已经被用到其他子序列中
                continue;
            }

            // 先判断 v 是否能接到其他子序列后面
            if (need.containsKey(v) && need.get(v) > 0) {
```

```

        // v 可以接到之前的某个序列后面
        freq.put(v, freq.get(v) - 1);
        // 对 v 的需求减一
        need.put(v, need.get(v) - 1);
        // 对 v + 1 的需求加一
        need.put(v + 1, need.getOrDefault(v + 1, 0) + 1);
    } else if (freq.getOrDefault(v, 0) > 0 &&
                freq.getOrDefault(v + 1, 0) > 0 &&
                freq.getOrDefault(v + 2, 0) > 0) {
        // 将 v 作为开头，新建一个长度为 3 的子序列 [v, v+1, v+2]
        freq.put(v, freq.get(v) - 1);
        freq.put(v + 1, freq.get(v + 1) - 1);
        freq.put(v + 2, freq.get(v + 2) - 1);
        // 对 v + 3 的需求加一
        need.put(v + 3, need.getOrDefault(v + 3, 0) + 1);
    } else {
        // 两种情况都不符合，则无法分配
        return false;
    }
}

return true;
}
}

```

至此，这道题就解决了。

那你可能会说，斗地主里面顺子至少要 5 张连续的牌，我们这道题只计算长度最小为 3 的子序列，怎么办？

很简单，把我们的 else if 分支修改一下，连续判断 v 之后的连续 5 个元素就行了。

那么，我们再难为自己，如果我想要的不只是一个布尔值，我想要你给我把子序列都打印出来，怎么办？

其实这也很好实现，只要修改 need，不仅记录对某个元素的需求个数，而且记录具体是哪些子序列产生的需求：

```

// need[6] = 2 说明有两个子序列需要 6
Map<Integer, Integer> need = new HashMap<>();

// need[6] = {
//     {3,4,5},
//     {2,3,4,5},
// }
// 记录哪两个子序列需要 6
Map<Integer, List<List<Integer>>> need = new HashMap<>();

```

这样，我们稍微修改一下之前的代码就行了：

```

class Solution {
    public boolean isPossible(int[] nums) {
        Map<Integer, Integer> freq = new HashMap<>();
        Map<Integer, List<List<Integer>>> need = new HashMap<>();

        // 统计 nums 中元素的频率
        for (int v : nums) {
            freq.put(v, freq.getOrDefault(v, 0) + 1);
        }
    }
}

```

```

    }

    for (int v : nums) {
        if (freq.get(v) == 0) {
            continue;
        }

        if (need.containsKey(v) && need.get(v).size() > 0) {
            // v 可以接到之前的某个序列后面
            freq.put(v, freq.get(v) - 1);
            // 随便取一个需要 v 的子序列
            List<Integer> seq = need.get(v).remove(need.get(v).size() - 1);
            // 把 v 接到这个子序列后面
            seq.add(v);
            // 这个子序列的需求变成了 v + 1
            need.computeIfAbsent(v + 1, k -> new ArrayList<>()).add(seq);
        } else if (freq.getOrDefault(v, 0) > 0 &&
                   freq.getOrDefault(v + 1, 0) > 0 &&
                   freq.getOrDefault(v + 2, 0) > 0) {
            // 可以将 v 作为开头
            freq.put(v, freq.get(v) - 1);
            freq.put(v + 1, freq.get(v + 1) - 1);
            freq.put(v + 2, freq.get(v + 2) - 1);
            // 新建一个长度为 3 的子序列 [v, v + 1, v + 2]
            List<Integer> seq = new ArrayList<>(Arrays.asList(v, v + 1, v +
2));
            // 对 v + 3 的需求加一
            need.computeIfAbsent(v + 3, k -> new ArrayList<>()).add(seq);
        } else {
            return false;
        }
    }

    // 打印切分出的所有子序列
    for (Map.Entry<Integer, List<List<Integer>>> entry : need.entrySet()) {
        for (List<Integer> seq : entry.getValue()) {
            for (int val : seq) {
                System.out.print(val + " ");
            }
            System.out.println();
        }
    }

    return true;
}
}

```

这样，我们记录具体子序列的需求也实现了。

如果本文对你有帮助，点个赞，微信会给你推荐更多相似文章。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 烧饼排序算法



微信搜一搜

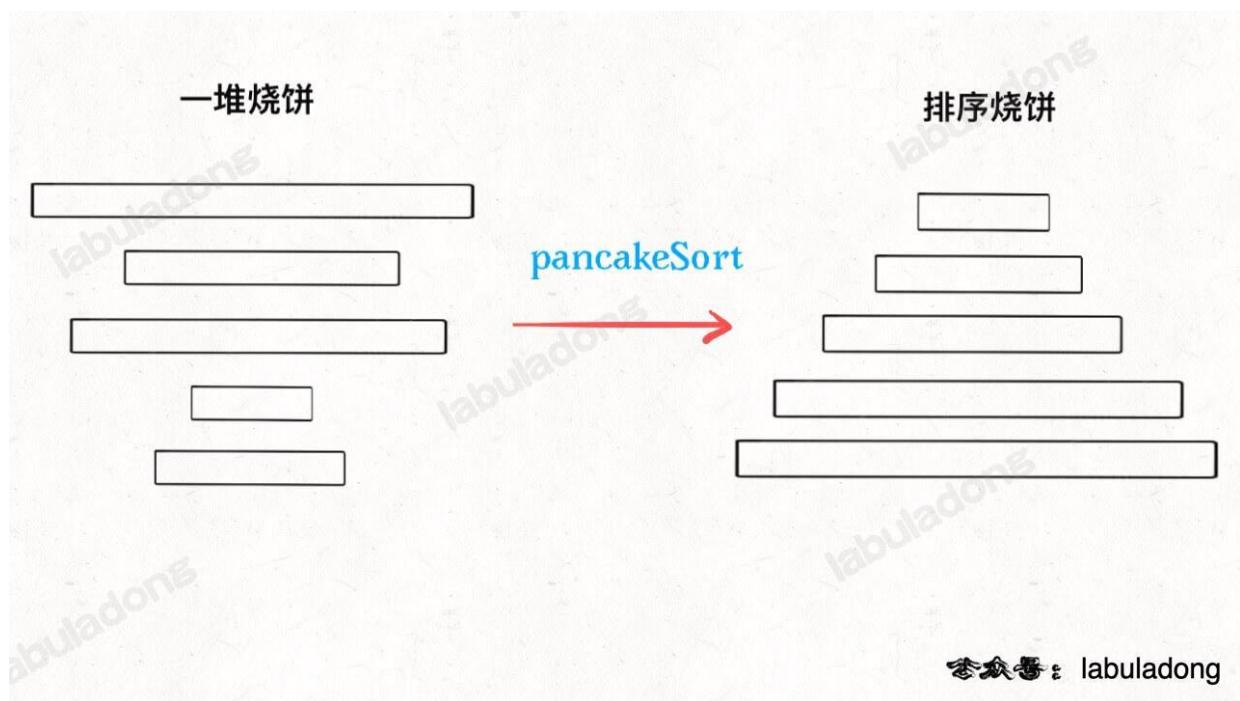
Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

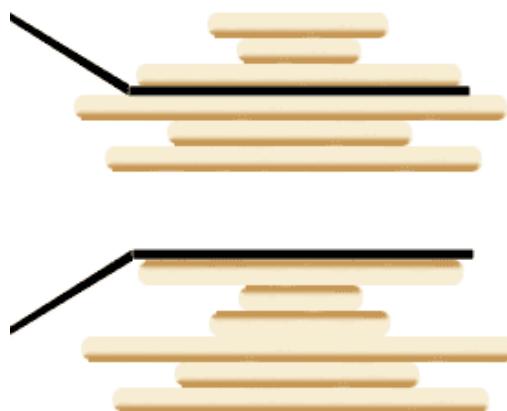
读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">969. Pancake Sorting</a>	<a href="#">969. 煎饼排序</a>	●

力扣第 969 题「煎饼排序」是个很有意思的实际问题：假设盘子上有  $n$  块面积大小不一的烧饼，你如何用一把锅铲进行若干次翻转，让这些烧饼的大小有序（小的在上，大的在下）？



设想一下用锅铲翻转一堆烧饼的情景，其实是有一点限制的，我们每次只能将最上面的若干块饼子翻转：



我们的问题是，如何使用算法得到一个翻转序列，使得烧饼堆变得有序？

首先，需要把这个问题抽象，用数组来表示烧饼堆：

▼ 969. 煎饼排序 [Leetcode](#) | 力扣

给你一个整数数组 `arr`，请使用 `煎饼翻转` 完成对数组的排序。

一次煎饼翻转的执行过程如下：

- 选择一个整数 `k`，`1 <= k <= arr.length`
- 反转子数组 `arr[0...k-1]`（下标从 0 开始）

例如，`arr = [3, 2, 1, 4]`，选择 `k = 3` 进行一次煎饼翻转，反转子数组 `[3, 2, 1]`，得到 `arr = [1, 2, 3, 4]`。

以数组形式返回能使 `arr` 有序的煎饼翻转操作所对应的 `k` 值序列。任何将数组排序且翻转次数在 `10 * arr.length` 范围内的有效答案都将被判断为正确。

示例 1：

```
输入: [3,2,4,1]
输出: [4,2,4,3]
解释:
我们执行 4 次煎饼翻转, k 值分别为 4, 2, 4, 和 3。
初始状态 arr = [3, 2, 4, 1]
第一次翻转后 (k = 4) : arr = [1, 4, 2, 3]
第二次翻转后 (k = 2) : arr = [4, 1, 2, 3]
第三次翻转后 (k = 4) : arr = [3, 2, 1, 4]
第四次翻转后 (k = 3) : arr = [1, 2, 3, 4], 此时已完成排序。
```

示例 2：

```
输入: [1,2,3]
输出: []
解释:
输入已经排序, 因此不需要翻转任何内容。
请注意, 其他可能的答案, 如 [3, 3] , 也将被判断为正确。
```

提示：

- `1 <= arr.length <= 100`
- `1 <= arr[i] <= arr.length`
- `arr` 中的所有整数互不相同（即，`arr` 是从 1 到 `arr.length` 整数的一个排列）

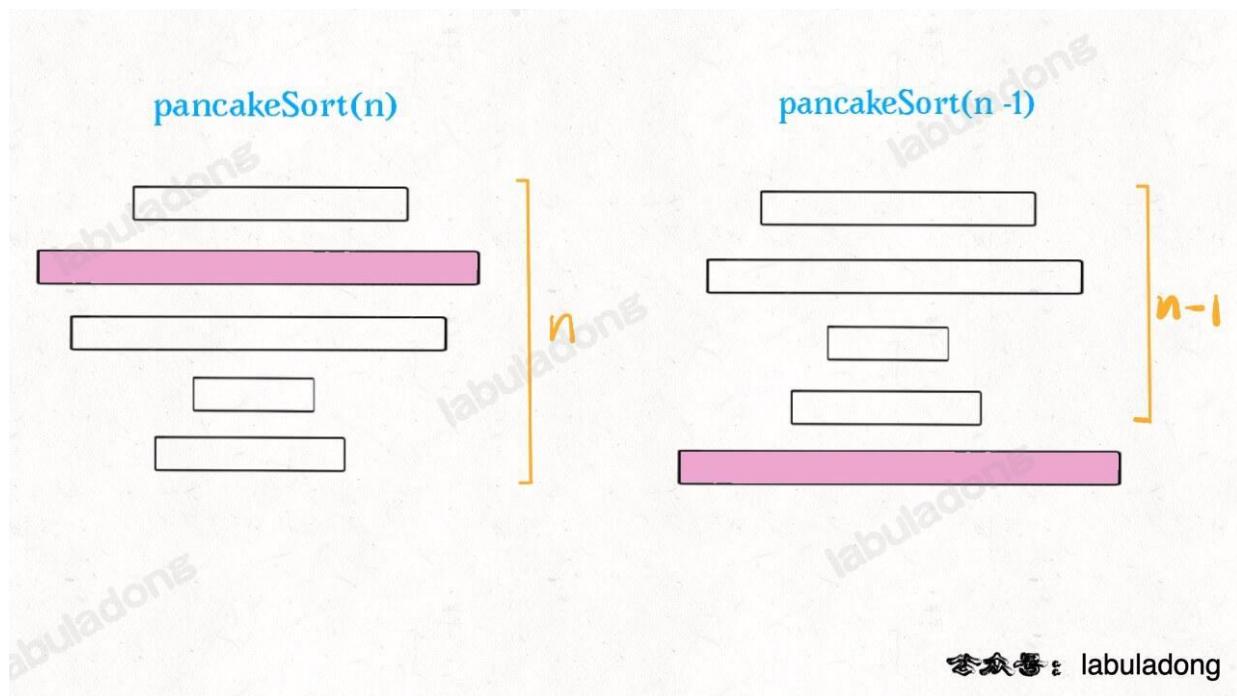
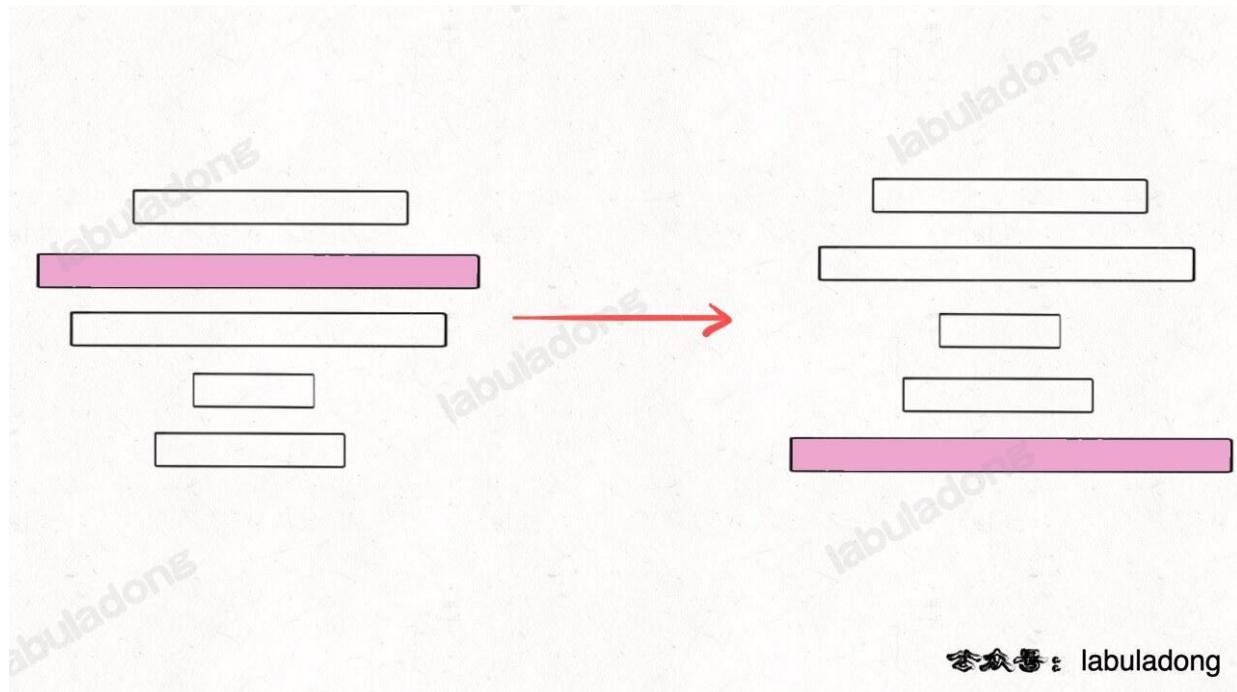
如何解决这个问题呢？其实类似上篇文章 [递归反转链表的一部分](#)，这也是需要递归思想的。

## 一、思路分析

为什么说这个问题有递归性质呢？比如说我们需要实现这样一个函数：

```
// cakes 是一堆烧饼, 函数会将前 n 个烧饼排序
void sort(int[] cakes, int n);
```

如果我们找到了前  $n$  个烧饼中最大的那个，然后设法将这个饼子翻转到最底下：



你看，这就是递归性质，总结一下思路就是：

- 1、找到  $n$  个饼中最大的那个。
- 2、把这个最大的饼移到最底下。
- 3、递归调用 `pancakeSort(A, n - 1)`。

base case:  $n == 1$  时，排序 1 个饼时不需要翻转。

那么，最后剩下个问题，如何设法将某块烧饼翻到最后呢？

其实很简单，比如第 3 块饼是最大的，我们想把它换到最后，也就是换到第  $n$  块。可以这样操作：

1、用锅铲将前 3 块饼翻转一下，这样最大的饼就翻到了最上面。

2、用锅铲将前  $n$  块饼全部翻转，这样最大的饼就翻到了第  $n$  块，也就是最后一块。

以上两个流程理解之后，基本就可以写出解法了，不过题目要求我们写出具体的反转操作序列，这也很简单，只要在每次翻转烧饼时记录下来就行了。

## 二、代码实现

只要把上述的思路用代码实现即可，唯一需要注意的是，数组索引从 0 开始，而我们要返回的结果是从 1 开始算的。

```
class Solution {
    // 记录反转操作序列
    LinkedList<Integer> res = new LinkedList<>();

    public List<Integer> pancakeSort(int[] cakes) {
        sort(cakes, cakes.length);
        return res;
    }

    void sort(int[] cakes, int n) {
        // base case
        if (n == 1) return;

        // 寻找最大饼的索引
        int maxCake = 0;
        int maxCakeIndex = 0;
        for (int i = 0; i < n; i++) {
            if (cakes[i] > maxCake) {
                maxCakeIndex = i;
                maxCake = cakes[i];
            }
        }

        // 第一次翻转，将最大饼翻到最上面
        reverse(cakes, 0, maxCakeIndex);
        res.add(maxCakeIndex + 1);
        // 第二次翻转，将最大饼翻到最下面
        reverse(cakes, 0, n - 1);
        res.add(n);

        // 递归调用
        sort(cakes, n - 1);
    }

    void reverse(int[] arr, int i, int j) {
        while (i < j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++; j--;
        }
    }
}
```

## ▶ 代码可视化动画

通过刚才的详细解释，这段代码应该是很清晰了。

算法的时间复杂度很容易计算，因为递归调用的次数是  $n$ ，每次递归调用都需要一次 for 循环，时间复杂度是  $O(n)$ ，所以总的复杂度是  $O(n^2)$ 。

**最后，我们可以思考一个问题：**按照我们这个思路，得出的操作序列长度应该为  $2(n - 1)$ ，因为每次递归都要进行 2 次翻转并记录操作，总共有  $n$  层递归，但由于 base case 直接返回结果，不进行翻转，所以最终的操作序列长度应该是固定的  $2(n - 1)$ 。

显然，这个结果不是最优的（最短的），比如说一堆煎饼  $[3, 2, 4, 1]$ ，我们的算法得到的翻转序列是  $[3, 4, 2, 3, 1, 2]$ ，但是最快捷的翻转方法应该是  $[2, 3, 4]$ ：

```
初始状态 : [3, 2, 4, 1]
翻前 2 个: [2, 3, 4, 1]
翻前 3 个: [4, 3, 2, 1]
翻前 4 个: [1, 2, 3, 4]
```

如果要求你的算法计算排序烧饼的最短操作序列，你该如何计算呢？或者说，解决这种求最优解法的问题，核心思路什么，一定需要使用什么算法技巧呢？

不妨分享一下你的思考。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 字符串乘法计算



微信搜一搜

Q labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">43. Multiply Strings</a>	<a href="#">43. 字符串相乘</a>	简单

对于比较小的数字，做运算可以直接使用编程语言提供的运算符，但是如果相乘的两个因数非常大，语言提供的数据类型可能就会溢出。一种替代方案就是，运算数以字符串的形式输入，然后模仿我们小学学习的乘法算术过程计算出结果，并且也用字符串表示。

看下力扣第 43 题「字符串相乘」：

▼ 43. 字符串相乘 [Leetcode](#) | [力扣](#)

给定两个以字符串形式表示的非负整数 `num1` 和 `num2`，返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

注意：不能使用任何内置的 `BigInteger` 库或直接将输入转换为整数。

示例 1:

```
输入: num1 = "2", num2 = "3"
输出: "6"
```

示例 2:

```
输入: num1 = "123", num2 = "456"
输出: "56088"
```

提示：

- `1 <= num1.length, num2.length <= 200`
- `num1` 和 `num2` 只能由数字组成。
- `num1` 和 `num2` 都不包含任何前导零，除了数字0本身。

需要注意的是，`num1` 和 `num2` 可以非常长，所以不可以把他们直接转成整型然后运算，唯一的思路就是模仿我们手算乘法。

比如说我们手算  $123 \times 45$ ，应该会这样计算：

$$\begin{array}{r} 123 \\ \times 45 \\ \hline 615 \\ 492 \\ \hline 5535 \end{array}$$

计算  $123 \times 5$ , 再计算  $123 \times 4$ , 最后错一位相加。这个流程恐怕小学生都可以熟练完成, 但是你是否能把这个运算过程进一步机械化, 写成一套算法指令让没有任何智商的计算机来执行呢?

你看这个简单过程, 其中涉及乘法进位, 涉及错位相加, 还涉及加法进位; 而且还有一些不易察觉的问题, 比如说两位数乘以两位数, 结果可能是四位数, 也可能是三位数, 你怎么想出一个标准化的处理方式? 这就是算法的魅力, 如果没有计算机思维, 简单的问题可能都没办法自动化处理。

首先, 我们这种手算方式还是太「高级」了, 我们要再「低级」一点,  $123 \times 5$  和  $123 \times 4$  的过程还可以进一步分解, 最后再相加:

$$\begin{array}{r} 123 \\ \times 45 \\ \hline 15 \\ 10 \\ 05 \\ 12 \\ 08 \\ \hline 04 \\ \hline 5535 \end{array}$$

现在  $123$  并不大, 如果是个很大的数字的话, 是无法直接计算乘积的。我们可以用一个数组在底下接收相加结果:

The diagram illustrates the multiplication of 123 by 45 using a vertical multiplication algorithm. The numbers are aligned vertically:

1 2 3					
	4 5				
<hr/>					
	1 5				
	1 0				
	0 5				
	1 2				
	0 8				
	0 4				
<hr/>					
res					

整个计算过程大概就是这样，有两个指针 `i`, `j` 在 `num1` 和 `num2` 上游走，计算乘积，同时将乘积叠加到 `res` 的正确位置，如下 GIF 图所示：

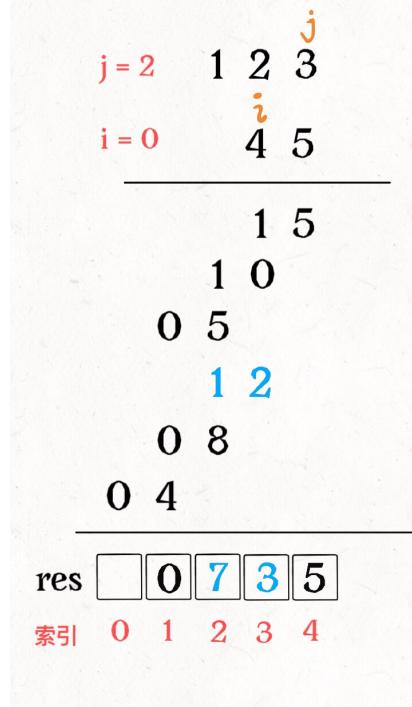
The Gif shows the step-by-step process of multiplying 123 by 45 using a vertical multiplication algorithm. It highlights the addition of partial products to the result array 'res'. The numbers are aligned vertically:

1 2 3					
	4 5				
<hr/>					
	1 5				
	1 0				
	0 5				
	1 2				
	0 8				
	0 4				
<hr/>					
res					

公众号: labuladong

现在还有一个关键问题，如何将乘积叠加到 `res` 的正确位置，或者说，如何通过 `i`, `j` 计算 `res` 的对应索引呢？

其实，细心观察之后就发现，`num1[i]` 和 `num2[j]` 的乘积对应的就是 `res[i+j]` 和 `res[i+j+1]` 这两个位置。



明白了这一点，就可以用代码模仿出这个计算过程了：

```

class Solution {
    public String multiply(String num1, String num2) {
        int m = num1.length(), n = num2.length();
        // 结果最多为 m + n 位数
        int[] res = new int[m + n];
        // 从个位数开始逐位相乘
        for (int i = m - 1; i >= 0; i--) {
            for (int j = n - 1; j >= 0; j--) {
                int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
                // 乘积在 res 对应的索引位置
                int p1 = i + j, p2 = i + j + 1;
                // 叠加到 res 上
                int sum = mul + res[p2];
                res[p2] = sum % 10;
                res[p1] += sum / 10;
            }
        }
        // 结果前缀可能存的 0 (未使用的位)
        int i = 0;
        while (i < res.length && res[i] == 0)
            i++;
        // 将计算结果转化成字符串
        StringBuilder str = new StringBuilder();
        for (; i < res.length; i++)
            str.append(res[i]);

        return str.length() == 0 ? "0" : str.toString();
    }
}

```

至此，字符串乘法算法就完成了。

总结一下，我们习以为常的一些思维方式，在计算机看来是非常难以做到的。比如说我们习惯的算术流程并不复杂，但是如果让你再进一步，翻译成代码逻辑，并不简单。算法需要将计算流程再简化，通过边算边叠加的方式来得到结果。

俗话教育我们，不要陷入思维定式，不要程序化，要发散思维，要创新。但我觉得程序化并不是坏事，可以大幅提高效率，减小失误率。算法不就是一套程序化的思维吗，只有程序化才能让计算机帮助我们解决复杂问题呀！

也许算法就是一种寻找思维定式的思维吧，希望本文对你有帮助。

---

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 如何判定完美矩形



微信搜一搜

Q labuladong 公众号

学算法认准 labuladong，致力于把算法讲清楚！查看完整教程、习题请访问我的网站 [labuladong.online](https://labuladong.online)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

LeetCode	力扣	难度
<a href="#">391. Perfect Rectangle</a>	<a href="#">391. 完美矩形</a>	●

今天讲一道非常有意思，而且比较有难度的题目。

我们知道一个矩形有四个顶点，但是只要两个顶点的坐标就可以确定一个矩形了（比如左下角和右上角两个顶点坐标）。

今天来看看力扣第 391 题「完美矩形」，题目会给我们输入一个数组 `rectangles`，里面装着若干四元组  $(x_1, y_1, x_2, y_2)$ ，每个四元组就是记录一个矩形的左下角和右上角坐标。

也就是说，输入的 `rectangles` 数组实际上就是很多小矩形，题目要求我们输出一个布尔值，判断这些小矩形能否构成一个「完美矩形」。函数签名如下：

```
boolean isRectangleCover(int[][] rectangles)
```

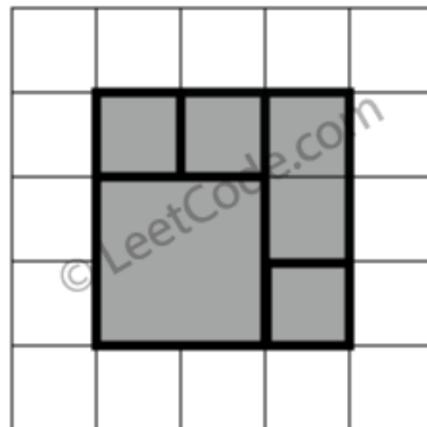
所谓「完美矩形」，就是说 `rectangles` 中的小矩形拼成图形必须是一个大矩形，且大矩形中不能有重叠和空缺。

比如说题目给我们举了几个例子：

**Example 1:**

```
rectangles = [
    [1,1,3,3],
    [3,1,4,2],
    [3,2,4,4],
    [1,3,2,4],
    [2,3,3,4]
]
```

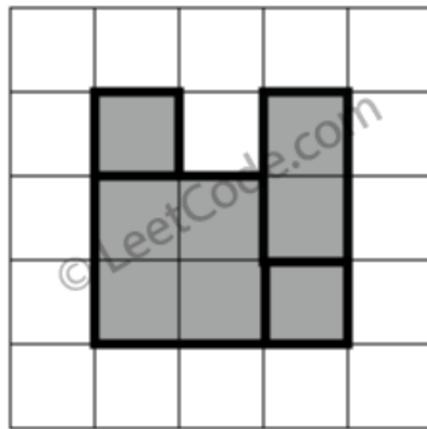
返回 `true`，因为最终形成的图形中没有空缺和重叠。



### Example 2:

```
rectangles = [
    [1,1,3,3],
    [3,1,4,2],
    [1,3,2,4],
    [3,2,4,4]
]
```

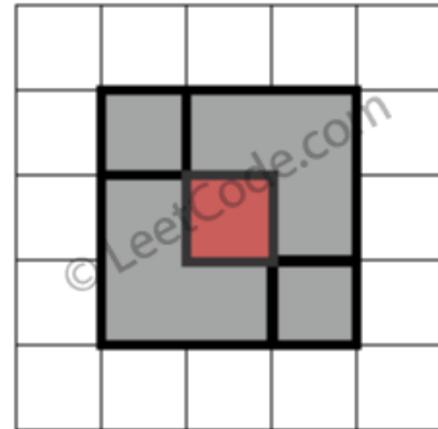
返回 `false`, 因为最终形成的图形中有空缺。



### Example 3:

```
rectangles = [
    [1,1,3,3],
    [3,1,4,2],
    [1,3,2,4],
    [2,2,4,4]
]
```

返回 `false`, 因为最终形成的图形存在重叠。



这个题目难度是 Hard, 如果没有做过类似的题目, 还真做不出来。

常规的思路, 起码要把最终形成的图形表示出来吧, 而且你要有方法去判断两个矩形是否有重叠, 是否有空隙, 虽然可以做到, 不过感觉异常复杂。

其实, 想判断最终形成的图形是否是完美矩形, 需要从「面积」和「顶点」两个角度来处理。

先说说什么叫从「面积」的角度。

`rectangles` 数组中每个元素都是一个四元组 (`x1, y1, x2, y2`), 表示一个小矩形的左下角顶点坐标和右上角顶点坐标。

那么假设这些小矩形最终形成了一个「完美矩形」, 你会不会求这个完美矩形的左下角顶点坐标 (`X1, Y1`) 和右上角顶点的坐标 (`X2, Y2`)?

这个很简单吧, 左下角顶点 (`X1, Y1`) 就是 `rectangles` 中所有小矩形中最靠左下角的那个小矩形的左下角顶点; 右上角顶点 (`X2, Y2`) 就是所有小矩形中最靠右上角的那个小矩形的右上角顶点。

注意我们用小写字母表示小矩形的坐标, 大写字母表示最终形成的完美矩形的坐标, 可以这样写代码:

```
// 左下角顶点, 初始化为正无穷, 以便记录最小值
double X1 = Double.POSITIVE_INFINITY, Y1 = Double.POSITIVE_INFINITY;

// 右上角顶点, 初始化为负无穷, 以便记录最大值
double X2 = Double.NEGATIVE_INFINITY, Y2 = Double.NEGATIVE_INFINITY;

for(int[] rectangle : rectangles){
```

```

int x1 = rectangle[0], y1 = rectangle[1], x2 = rectangle[2], y2 = rectangle[3];

// 取小矩形左下角顶点的最小值
X1 = Math.min(X1, x1);
Y1 = Math.min(Y1, y1);

// 取小矩形右上角顶点的最大值
X2 = Math.max(X2, x2);
Y2 = Math.max(Y2, y2);
}

```

这样就能求出完美矩形的左下角顶点坐标 ( $X1, Y1$ ) 和右上角顶点的坐标 ( $X2, Y2$ ) 了。

计算出的  $X1, Y1, X2, Y2$  坐标是完美矩形的「理论坐标」，如果所有小矩形的面积之和不等于这个完美矩形的理论面积，那么说明最终形成的图形肯定存在空缺或者重叠，肯定不是完美矩形。

代码可以进一步：

```

boolean isRectangleCover(int[][] rectangles) {
    int X1 = Integer.MAX_VALUE, Y1 = Integer.MAX_VALUE;
    int X2 = Integer.MIN_VALUE, Y2 = Integer.MIN_VALUE;
    // 记录所有小矩形的面积之和
    int actualArea = 0;
    for (int[] rect : rectangles) {
        int x1 = rect[0], y1 = rect[1], x2 = rect[2], y2 = rect[3];
        // 计算完美矩形的理论坐标
        X1 = Math.min(X1, x1);
        Y1 = Math.min(Y1, y1);
        X2 = Math.max(X2, x2);
        Y2 = Math.max(Y2, y2);
        // 累加所有小矩形的面积
        actualArea += (x2 - x1) * (y2 - y1);
    }

    // 计算完美矩形的理论面积
    int expectedArea = (X2 - X1) * (Y2 - Y1);
    // 面积应该相同
    if (actualArea != expectedArea) {
        return false;
    }

    return true;
}

```

这样，「面积」这个维度就完成了，思路其实不难，无非就是假设最终形成的图形是个完美矩形，然后比较面积是否相等，如果不相等的话说明最终形成的图形一定存在空缺或者重叠部分，不是完美矩形。

但是反过来说，如果面积相同，是否可以证明最终形成的图形是完美矩形，一定不存在空缺或者重叠？

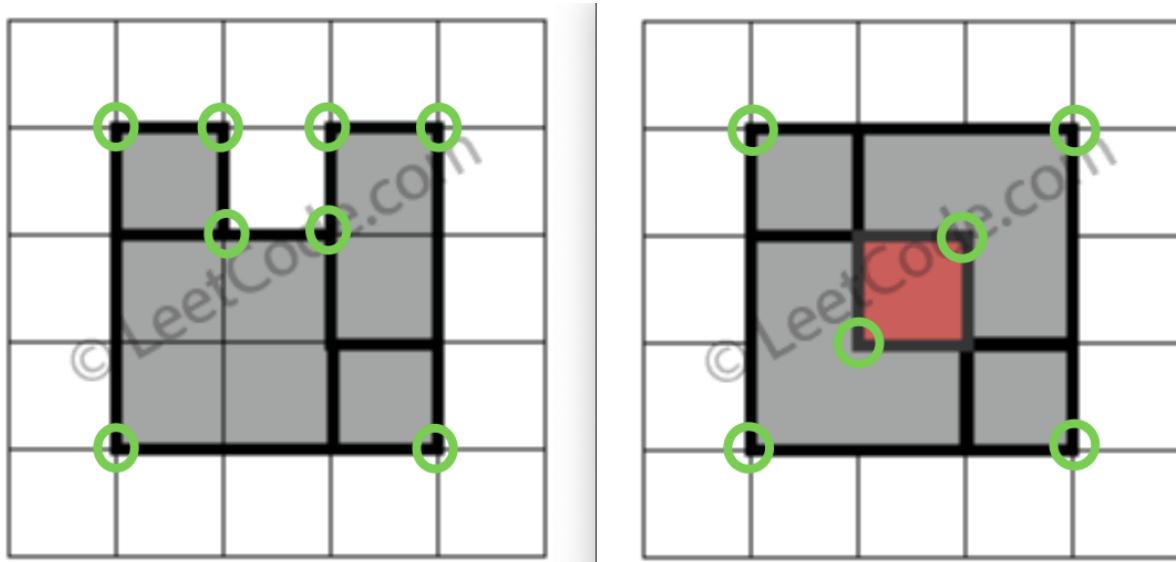
肯定是不行的，举个很简单的例子，你假想一个完美矩形，然后我在它中间挖掉一个小矩形，把这个小矩形向下平移一个单位。这样小矩形的面积之和没变，但是原来的完美矩形中就空缺了一部分，也重叠了一部分，已经不是完美矩形了。

综上，即便面积相同，并不能完全保证不存在空缺或者重叠，所以我们需要从「顶点」的维度来辅助判断。

记得小学的时候有一道智力题，给你一个矩形，切一刀，剩下的图形有几个顶点？答案是，如果沿着对角线切，就剩 3 个顶点；如果横着或者竖着切，剩 4 个顶点；如果只切掉一个小角，那么会出现 5 个顶点。

回到这道题，我们接下来的分析也有那么一点智力题的味道。

显然，完美矩形一定只有四个顶点。矩形嘛，按理说应该有四个顶点，如果存在空缺或者重叠的话，肯定不是四个顶点，比如说题目的这两个例子就有不止 4 个顶点：

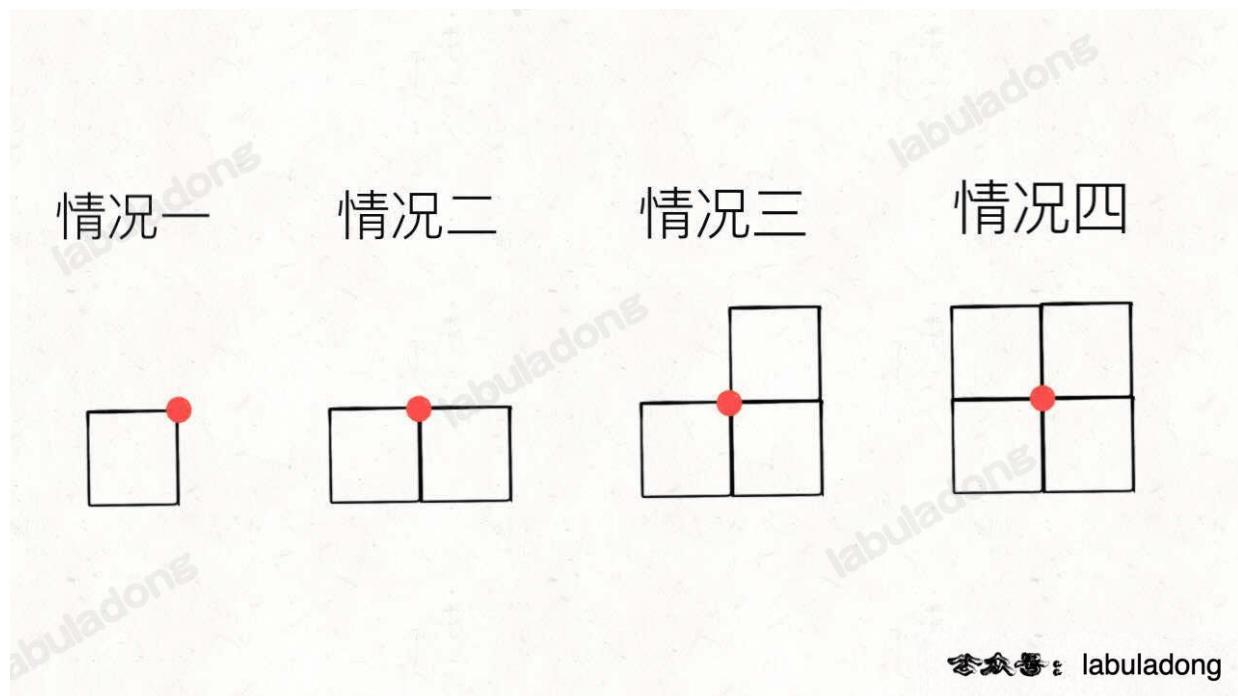


我也不知道应该用「顶点」还是「角」来形容，好像都不太准确，本文统一用「顶点」来形容，大家理解就好~

只要我们想办法计算 `rectangles` 中的小矩形最终形成的图形有几个顶点，就能判断最终的图形是不是一个完美矩形了。

那么顶点是如何形成的呢？我们倒是一眼就可以看出来顶点在哪里，问题是如何让计算机，让算法知道某一个点是不是顶点呢？这也是本题的难点所在。

看下图的四种情况：



图中画红点的地方，什么时候是顶点，什么时候不是顶点？显然，情况一和情况三的时候是顶点，而情况二和情况四的时候不是顶点。

也就是说，当某一个点同时是 2 个或者 4 个小矩形的顶点时，该点最终不是顶点；当某一个点同时是 1 个或者 3 个小矩形的顶点时，该点最终是一个顶点。

注意，2 和 4 都是偶数，1 和 3 都是奇数，我们想计算最终形成的图形中有几个顶点，也就是要筛选出那些出现了奇数次的顶点，可以这样写代码：

```
boolean isRectangleCover(int[][] rectangles) {
    int X1 = Integer.MAX_VALUE, Y1 = Integer.MAX_VALUE;
    int X2 = Integer.MIN_VALUE, Y2 = Integer.MIN_VALUE;

    int actualArea = 0;
    // 哈希集合，记录最终图形的顶点
    Set<String> points = new HashSet<>();
    for (int[] rect : rectangles) {
        int x1 = rect[0], y1 = rect[1], x2 = rect[2], y2 = rect[3];
        X1 = Math.min(X1, x1);
        Y1 = Math.min(Y1, y1);
        X2 = Math.max(X2, x2);
        Y2 = Math.max(Y2, y2);

        actualArea += (x2 - x1) * (y2 - y1);
        // 先算出小矩形每个点的坐标，用字符串表示，方便存入哈希集合
        String p1 = x1 + "," + y1;
        String p2 = x1 + "," + y2;
        String p3 = x2 + "," + y1;
        String p4 = x2 + "," + y2;
        // 对于每个点，如果存在集合中，删除它；
        // 如果不存在集合中，添加它；
        // 在集合中剩下的点都是出现奇数次的点
        for (String p : new String[]{p1, p2, p3, p4}) {
            if (points.contains(p)) {
                points.remove(p);
            } else {
                points.add(p);
            }
        }
    }

    int expectedArea = (X2 - X1) * (Y2 - Y1);
    if (actualArea != expectedArea) {
        return false;
    }

    // 检查顶点个数
    if (points.size() != 4 ||
        !points.contains(X1 + "," + Y1) ||
        !points.contains(X1 + "," + Y2) ||
        !points.contains(X2 + "," + Y1) ||
        !points.contains(X2 + "," + Y2)) {
        return false;
    }

    return true;
}
```

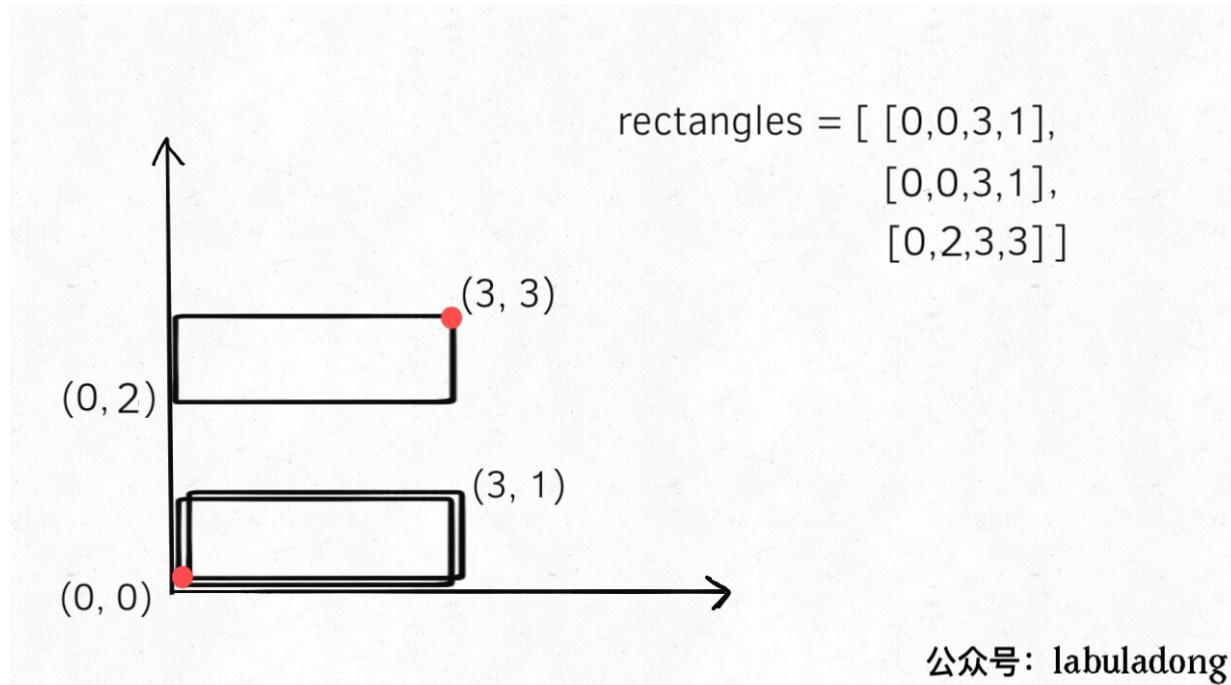
这段代码中，我们用一个 `points` 集合记录 `rectangles` 中小矩形组成的最终图形的顶点坐标，关键逻辑在于如何向 `points` 中添加坐标：

如果某一个顶点 `p` 存在于集合 `points` 中，则将它删除；如果不存在于集合 `points` 中，则将它插入。

这个简单的逻辑，让 `points` 集合最终只会留下那些出现了 1 次或者 3 次的顶点，那些出现了 2 次或者 4 次的顶点都被消掉了。

那么首先想到，`points` 集合中最后应该只有 4 个顶点对吧，如果 `len(points) != 4` 说明最终构成的图形肯定不是完美矩形。

但是如果 `len(points) == 4` 是否能说明最终构成的图形肯定是完美矩形呢？也不行，因为题目并没有说 `rectangles` 中的小矩形不存在重复，比如下面这种情况：



下面两个矩形重复了，按照我们的算法逻辑，它们的顶点都被消掉了，最终是剩下了四个顶点；再看面积，完美矩形的理论坐标是图中红色的点，计算出的理论面积和实际面积也相同。但是显然这种情况不是题目要求完美矩形。

所以不仅要保证 `len(points) == 4`，而且要保证 `points` 中最终剩下的点坐标就是完美矩形的四个理论坐标，直接看代码吧：

```
class Solution {  
    public boolean isRectangleCover(int[][] rectangles) {  
        int X1 = Integer.MAX_VALUE, Y1 = Integer.MAX_VALUE;  
        int X2 = Integer.MIN_VALUE, Y2 = Integer.MIN_VALUE;  
  
        Set<String> points = new HashSet<>();  
        int actualArea = 0;  
        for (int[] rect : rectangles) {  
            int x1 = rect[0], y1 = rect[1], x2 = rect[2], y2 = rect[3];  
            // 计算完美矩形的理论顶点坐标  
            X1 = Math.min(X1, x1);  
            Y1 = Math.min(Y1, y1);  
            X2 = Math.max(X2, x2);  
            Y2 = Math.max(Y2, y2);  
            // 累加小矩形的面积  
            actualArea += (x2 - x1) * (y2 - y1);  
            String point = x1 + " " + y1;  
            if (points.contains(point)) {  
                points.remove(point);  
            } else {  
                points.add(point);  
            }  
        }  
        return points.size() == 4 && X1 == 0 && Y1 == 0 && X2 == 3 && Y2 == 3;  
    }  
}
```

```
// 记录最终形成的图形中的顶点
String p1 = x1 + "," + y1;
String p2 = x1 + "," + y2;
String p3 = x2 + "," + y1;
String p4 = x2 + "," + y2;
for (String p : new String[]{p1, p2, p3, p4}) {
    if (points.contains(p)) {
        points.remove(p);
    } else {
        points.add(p);
    }
}
// 判断面积是否相同
int expectedArea = (X2 - X1) * (Y2 - Y1);
if (actualArea != expectedArea) {
    return false;
}
// 判断最终留下的顶点个数是否为 4
if (points.size() != 4) {
    return false;
}
// 判断留下的 4 个顶点是否是完美矩形的顶点
if (!points.contains(X1 + "," + Y1)) return false;
if (!points.contains(X1 + "," + Y2)) return false;
if (!points.contains(X2 + "," + Y1)) return false;
if (!points.contains(X2 + "," + Y2)) return false;
// 面积和顶点都对应，说明矩形符合题意
return true;
}
```

---

▶ 🎨 代码可视化动画🎨

---

这就是最终的解法代码，从「面积」和「顶点」两个维度来判断：

- 1、判断面积，通过完美矩形的理论坐标计算出一个理论面积，然后和 `rectangles` 中小矩形的实际面积做对比。
  - 2、判断顶点，`points` 集合中应该只剩下 4 个顶点且剩下的顶点必须都是完美矩形的理论顶点。
- 

成体系的算法教程、习题以及配套刷题插件可查看 [我的网站](#)。

# 习题汇总页面

我在 [本站阅读方法](#) 中提到，对于希望临阵磨枪的读者，可以着重学习标有【强化练习】的习题章节。

但是这些习题章节分散在目录中的不同位置，有读者反馈说担心遗漏。所以我将本站 33 篇【强化练习】章节入口按顺序整理汇总在这里，方便有需要的读者查阅。

本页面提供的目录的作用仅仅是帮助临阵磨枪的读者做一个快速索引。

如果有足够的时间或精力，更建议按照本站目录顺序学习，稳扎稳打，效果最佳。

每个习题章节开头都有「前置阅读」部分，指明练习该章节必须得算法框架和技巧，练习之前务必要先学习这些内容。

另外，本站内容持续更新优化中，可能会调整章节顺序或加入新的章节。

[【强化练习】链表双指针经典习题](#)

[【强化练习】数组双指针经典习题](#)

[【强化练习】前缀和技巧经典习题](#)

[【强化练习】滑动窗口算法经典习题](#)

[【强化练习】二分搜索算法经典习题](#)

[【强化练习】用「遍历」思维解题 I](#)

[【强化练习】用「遍历」思维解题 II](#)

[【强化练习】用「遍历」思维解题 III](#)

[【强化练习】用「分解问题」思维解题 I](#)

[【强化练习】用「分解问题」思维解题 II](#)

[【强化练习】同时运用两种思维解题](#)

[【强化练习】利用后序位置解题 I](#)

[【强化练习】利用后序位置解题 II](#)

[【强化练习】利用后序位置解题 III](#)

[【强化练习】运用层序遍历解题 I](#)

[【强化练习】运用层序遍历解题 II](#)

[【强化练习】二叉搜索树经典例题 I](#)

[【强化练习】二叉搜索树经典例题 II](#)

[【强化练习】栈的经典习题](#)

[【强化练习】括号类问题汇总](#)

[【强化练习】队列的经典习题](#)

[【强化练习】单调栈的几种变体及经典习题](#)

[【强化练习】单调队列的通用实现及经典习题](#)

[【强化练习】哈希表更多习题](#)

[【强化练习】优先级队列经典习题](#)

[【强化练习】Trie 树算法习题](#)

[【强化练习】并查集经典习题](#)

[【强化练习】更多经典设计习题](#)

[【强化练习】回溯算法经典习题 I](#)

[【强化练习】回溯算法经典习题 II](#)

[【强化练习】回溯算法经典习题 III](#)

[【强化练习】BFS 经典习题 I](#)

[【强化练习】BFS 经典习题 II](#)

[【强化练习】数学技巧相关习题](#)

动态规划相关题目请查看动态规划章节的文章，未在这里列出。

# labuladong.online 更新日志

2024/11/14

精简 [回溯算法核心框架](#)，将 N 皇后问题和数独问题整理到 [回溯算法实践：数独和 N 皇后问题](#)。

2024/11/10

- 增加右上角搜索框能够显示的结果条数。
- [可视化面板简介](#) 中添加每个数据结构的 API 文档。

2024/11/3

- 添加 [红黑树的完美平衡及可视化](#)。

2024/10/30

可视化面板支持颜色系统，同时支持通过注释关键词和页面调色板修改任意对象的颜色。

详见 [可视化面板的颜色系统](#)。

2024/10/27

- 修复可视化面板显示 [404 资源不存在](#) 的 bug。
- 修复部分文章显示 [Fail to fetch data](#) 的 bug。

若你依然遇到上述问题，请尝试刷新页面，或清除浏览器缓存，即可修复。

2024/10/21

- 修复可视化面板不能显示排序算法的 bug。

2024/10/19

- 添加 [并查集 Union Find 核心原理](#) 和 [线段树核心原理](#)。
- [Trie 树基础](#) 添加可视化面板辅助理解。

2024/10/13

- 优化 [二叉堆](#) 的内容，改为使用索引 0 开始存储元素，并支持使用 [showArray](#) 方法同时展示底层数组和二叉树结构。
- 更新 [堆排序算法](#)。

2024/10/3

快速入门章节新增 [系列排序算法](#) 的讲解，并配套可排序算法可视化。

▶  [代码可视化动画](#) 

2024/10/1

- 为 [回溯算法习题 I](#), [回溯算法习题 II](#), [回溯算法习题 III](#) 的所有题目添加可视化面板。
- 修复打卡日历样式错误的问题。

2024/9/24

- 修复习题章节中的代码块缺少复制按钮的 bug。
- 为 [BFS 习题章节 I](#), [BFS 习题章节 II](#) 中的所有解法代码添加可视化面板。
- 修复某些章节的上一页、下一页会跳到主页的 bug。

2024/9/22

- 修复小鹅通会员权限迁移的 bug。
- 修复某些情况下微信支付失败的 bug。
- 优化了网站总是弹出语言切换选项的问题。

2024/9/21

- 修复部分多语言代码 tab 中缺少 JavaScript 代码的问题。
- 将基础知识章节中的 C++ 代码进行验证，修复了一些错误。
- 修复题目面板缺少力扣跳转链接的问题。
- 修复了少部分题目缺少多语言解法代码的问题。
- 全新的通知组件，优化若干已知问题。

2024/9/19

- 感谢大家的反馈，修复了个别文章中代码格式错乱、内容缺失的问题。
- 对每篇文章添加「前置知识」，并在标题下方显示本文能够解决的题目数量。

2024/9/18

感谢大家的反馈，修复了若干 bug：

- 优化有时 GitHub 登录失败的问题。
- 优化海外读者的网络访问速度。
- 修复有些读者无法修改评论的问题。

2024/9/14

- 增加 [【强化练习】数学技巧相关习题](#) 和 [【强化练习】并查集相关习题](#)。

2024/9/8

- 增加几十道习题。
  - 增加 [习题汇总页面](#)，方便有需要的读者查看所有强化练习章节。

2024/9/1

网站右上角的搜索栏支持按照题号、中英文题目名称、题目链接等信息直接搜索相关的文章。

2024/8/25

- 文中每篇文章都添加了阅读进度标记，方便判断该文章是否已经学过。

- 文章开头添加了前置知识点，方便读者更流畅地学习理解文章内容。

2024/8/18

- 校准本站所有文章中的多语言代码片段，保证代码的准确性。
- 网站搜索框支持直接搜索力扣题目名称、题号。

2024/7/17

修复可视化面板构建二叉树/多叉树时，节点不显示的问题。

将网站主要语言全部统一为 Java。

2024/7/1

- 修复习题部分可视化面板无法加载的 bug。

大幅提升习题中多语言解法的准确性。

2024/6/15

- 优化阅读体验，更新代码样式。
- 更新二叉堆、二叉树的基础知识，具体请看主站目录的「极速入门」章节。
- 优化 [订单页面](#)，显示赠送老用户的网站会员订单及有效期。

修复了网站评论区点赞/点踩功能导致数量清零的 bug。

修复可视化面板功能按钮缺失的 bug。

2024/5/10

- 统一网站图标风格，若干其他细节优化。
- 修复移动端登录成功后重定向到 404 页面的 bug。

2024/5/1

- 增加若干基础知识，优化若干内容。
- 添加基础知识章节。

2024/3/26

- 修复网站会员文章中部分链接依然指向旧版课程的 bug。
- 修复了登录后也无法加载评论的 bug。
- 修复了 GitHub 登录后无法进行评论的 bug。

2024/2/26

支持海外读者使用 GitHub 账户登录网站。

## 2024/2/22

- 支持文章的阅读历史。在侧边栏中，学完的文章会显示 标记，未学完文章会显示 标记，方便你了解自己的学习进度。
- 支持读者修改和删除自己的评论。
- 修复部分链接 404 的问题。

## 2024/2/21

- 支持微信内直接调起支付。

PC 端网站登录需要使用微信扫码，但如果想用手机端登录网站就麻烦了，还得把登录二维码截图到微信里才能扫码登录。

现在支持了移动端一键登录，只需要在手机微信里点开网站，点击「登录」按钮，授权网站读取用户昵称，就可以直接登录网站了。

## 2024/2/17

修复优化网站文章的 URL 生成算法导致站内搜索 404 的 bug。

## 2024/2/15

很多海外读者反馈，无法使用国内的微信支付宝支付。现在支持 PayPal 支付，方便海外读者购买课程。

注意国内用户不要用 PayPal 支付，因为 PayPal 不允许两个中国账户之间进行交易，会报错「为遵守国际法规，这笔交易已被拒绝」。

## 2024/2/10

优化网站文章的 URL 生成算法，避免目录结构变化导致的 URL 变化。

以前收藏的文章链接可能失效；如果某些页面跳转出现 404，可能是因为浏览器存在旧的缓存，请清除浏览器缓存再试。

- 全面去除新网站、插件中指向小鹅通的过时链接。

## 2024/2/5

[打卡挑战](#) 的打卡日历在海外时区可能错误地显示缺卡，现在已修复。

- 课程视频支持键盘控制：左右方向键快进快退，上下方向键调整音量，空格键暂停播放。
- 其他若干小 bug 修复，若干体验优化。

## 2024/2/1

新网站 [labuladong.online](#) 正式上线，全面优化使用体验。小鹅通课程全部迁往新网站，以前的付费读者可以迁移课程权限到新网站，迁移指南 [见这里](#)。基于新网站解锁插件中课程专属题解的方法，已经在课程的第一章进行了更新。

新网站中的算法可视化面板添加「编辑」按钮，读者可以修改算法可视化面板的代码并执行，例子：

▶  代码可视化动画 

2024/1/20

修复每次登录的用户 ID 都会变化的 bug。

2024/1/15

修复网站微信/支付宝支付成功后课程权限有延迟的 bug。

- 修复课程/会员解锁状态显示错误的问题。
- 修复编辑器若干 bug，如登录失败、代码无法执行等。

2023/12/31

- 可视化在线编辑器上线：<https://labuladong.online/algo-visualize>

2023/10/29

- 优化网站中 力扣/LeetCode 题目的显示和跳转。
- 修复课程中可视化面板无法全屏显示的 bug。
- 添加部分可视化面板帮助理解。

2023/10/2

- 添加 《labuladong 的算法笔记》纸质书 的详情介绍。
- 根据学员反馈，优化 30 天打卡挑战 的使用体验。

2023/8/29

- 全面优化可视化面板，修复若干 bug，大幅降低多余步骤。
- 全面更新 可视化面板使用指南，添加情景教学实操部分。

2023/8/24

- 可视化面板中间添加可拖拽的分界线。
- 可视化面板可高亮显示递归树上的递归路径。

The screenshot shows a code editor on the left with Java code for generating permutations using backtracking. The right side features a visualization of a binary tree where each node represents a partial permutation of the array [1, 2, 3]. The root node is an empty list. Its left child is (1), and its right child is (2). Node (1) has children (1, 2) and (1, 3). Node (2) has children (2, 1) and (2, 3). The bottom of the visualization shows the original array [1, 2, 3] and the current state of the track [2, 3, 1]. A variable `backtrack` is also shown.

## 算法可视化

可交互的算法可视化面板，所有题目代码下方都有对应的可视化面板。

- 算法执行流程可视化

左侧是算法代码，右侧是可视化面板，正在执行的代码行会高亮，当前作用域的变量和数据结构会在右侧显示。

- 数据结构可视化

包括链表、数组、二叉树、哈希表等数据结构。

- 递归算法可视化

结合我一直强调的「框架思维」，从树的角度理解递归算法，将递归树可视化展现出来。

- 网站和所有配套插件均已适配此功能

2023/8/21

- ✓ 修复在网页上学习课程时，无法显示多语言代码和代码行内注释的 bug。
- ✓ 根据会员用户的反馈，优化网站会员的提示信息，新增「遇到问题」按钮。

2023/8/19

- ✓ 添加了系列刷题插件的视频介绍。
- ✓ 给算法可视化面板添加更显眼的样式。

2023/8/10

- ✓ 算法可视化面板功能升级，代码可直接点击，跳转对应的执行步骤：

The screenshot shows the website's navigation bar and a sidebar with various course links. The main area displays a code editor for an in-order tree traversal problem. The right side features a visualization of a binary tree with nodes labeled 1 through 5. A variable `root` points to node 1. A temporary variable `tmp` points to node 3. The code editor highlights a specific line of code: `root.left = root.right = ___bhmg8p`. A callout box points to this line with the number 10. Other numbered callouts (1-12) point to various UI elements: 1 (play/pause button), 2 (step back button), 3 (step forward button), 4 (progress bar), 5 (refresh button), 6 (copy URL button), 7 (help button), 8 (mouse click highlight), 9 (variable value display), 10 (data structure display), 11 (function stack display), and 12 (function parameters display).

你把前序位置的代码移到后序位置也可以，但是直接移到中序位置是不行的，需要稍作修改，这应该很容易看出来吧，我就不说啦。

2023/7/27

- 修复了会员解锁内容中多语言 tab 无法切换的问题。
- 将 **刷题打卡** 的内容目录移动到网站上，方便大家查看。
- 添加网站首页，点击左上角的 logo 即可返回查看。

2023/7/19

- 修复评论区主题和网站主题不一致的 bug。

2023/7/13

- 在第一章添加算法可视化功能的 [使用手册](#)。
- 在可视化面板右上角添加两个按钮，支持复制算法可视化面板的 URL；支持刷新算法可视化面板（适用于动画加载失败时重新刷新面板）：

① 播放 / 停按钮。点击后开始播放算法可视化过程，再次点击可暂停。

② 上一步 / 下一步按钮。非播放过程中点击可跳转到算法流程的上一步，播放过程中点击可放慢播放速度。

③ 下一步 / 加速按钮。非播放过程中点击可跳转到算法流程的下一步，播放过程中点击可加快播放速度。

④ 进度条，拖动可调整动画进度。

⑤ 点击可跳转到帮助页面。

⑥ 刷新按钮，可视化面板不显示或遇到 bug 时可尝试刷新。

⑦ 复制可视化面板的 URL，方便分享。

⑧ 正在执行的代码会高亮显示。

⑨ 鼠标点击可直接快进到对应代码执行。

⑩ 变量区域，显示当前作用域内的变量及其值。

⑪ 数据结构区域，用于可视化复杂数据结构。

⑫ 函数堆栈区域，显示递归的堆栈情况及函数参数。

**labuladong 网站地址：**  
<https://labuladong.online/algo/>

- 优化每篇文章最后的「相关文章」显示样式。

### ▶ 引用本文的题目

### ▼ 引用本文的文章

- [Dijkstra 算法模板及应用](#)
- [Git 原理之最近公共祖先](#)
- [东哥带你刷二叉树（后序篇）](#)
- [东哥带你刷二叉树（序列化篇）](#)
- [东哥带你刷二叉树（构造篇）](#)
- [二叉树（递归）专题课](#)
- [前缀树算法模板秒杀五道算法题](#)
- [动态规划和回溯算法的思维转换](#)
- [后序遍历的妙用](#)
- [回溯算法秒杀所有排列/组合/子集问题](#)
- [回溯算法解题套路框架](#)
- [图论基础及遍历算法](#)
- [在插件中解锁二叉树专属题解](#)
- [归并排序详解及应用](#)
- [我的刷题心得](#)
- [算法可视化功能简介](#)
- [算法学习和心流体验](#)

- 添加网站以及各个插件的更新日志，方便大家了解更新情况。

2023/7/1

- 可视化功能升级，支持以树的视角可视化所有递归算法：

The screenshot shows a code editor on the left and a visualization panel on the right. The code is a JavaScript implementation of the N-Queens problem:

```
var permute = function (nums) {
    let res = [];
    let track = [];
    let used = new Array(nums.length).fill(false);

    backtrack(nums, track, used, res);
    return res;
}

// 路径：记录在 track 中
// 选择列表：nums 中不存在于 track 的那些元素 (used[i] 为 false)
// 结束条件：nums 中的元素全都在 track 中出现
// @visualize status(track)
var backtrack = function (nums, track, used, res) {
    // 触发结束条件
    if (track.length === nums.length) {
        res.push([...track]);
        return;
    }

    for (let i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (used[i]) {
            // nums[i] 已经在 track 中，跳过
            continue;
        }
        // 做选择
        track.push(nums[i]);
        used[i] = true;
        // 进入下一层决策树
        backtrack(nums, track, used, res);
        // 取消选择
        track.pop();
        used[i] = false;
    }
}

let result = permute([1, 2, 3]);

```

The visualization panel on the right shows a backtracking tree for the N-Queens problem with 3 queens. The root node is labeled 0. Node 1 has children (1,2) and (1,3). Node 2 has children (2,1), (2,2), and (2,3). Node 3 has children (3,1) and (3,2). The final result is shown as a 3x3 grid where each row and column contains exactly one queen.

2023/6/27

- 修复算法可视化的 bug，添加 50 道题目的算法可视化。
- 支持 Chrome 插件和网站联动，在网页右上方显示插件连接状态。

2023/6/25

- 添加阅读历史功能，阅读完的文章会显示 标记，未读完的文章会显示 标记。

2023/6/20

- 新主题上线，全面优化使用体验。

# 可视化面板更新日志

2024/11/6

- 修复 线段树 的 Bug。
- 优化几种树结构的创建方法。

2024/11/3

- 可视化面板支持 红黑树 结构。

2024/10/31

可视化面板支持颜色系统，同时支持通过注释关键词和页面调色板修改任意对象的颜色。

详见 [可视化面板的颜色系统](#)。

2024/10/21

- 修复可视化面板不能显示排序算法的 bug。

2024/10/19

- 支持可视化 并查集 Union Find 结构。
- 支持可视化 线段树 结构。
- 支持可视化 Trie 树 结构。
- 支持变量绑定、设置节点颜色等功能。

具体用法参见 [可视化面板使用说明](#)。

2024/10/13

- 优化体验：点击代码跳转到下次执行时，数据结构不再会闪动。
- 二叉堆结构可以使用 `showArray` 展示底层数组。

2024/10/3

添加 `@visualize cert` 注释支持按照直方图的形式显示数组元素，辅助本站 [排序算法章节](#) 的学习。

▶ 🎃 代码可视化动画🎃

2024/7/17

添加 `@visualize bfs` 注释支持可视化 BFS 算法，具体使用方法参见 [可视化面板简介](#)。

▶ 😊 代码可视化动画😊

2024/7/17

修复构建二叉树/多叉树时，节点不显示的问题。

2024/6/10

使用 `@visualize` 标签对递归函数生成的递归树支持区分「已完成」和「未完成」状态，更直观地显示当前递归函数是否已经计算完成：

▶  [代码可视化动画](#) 

2024/6/2

支持可视化 [二叉堆](#) 结构，具体使用方法参见 [可视化面板简介](#)。

2024/5/19

- 点击 Console Log 面板后，对齐辅助线的颜色更加明显，方便查看递归层级。
- 给面板的各个部分添加滚动条，可以用鼠标拖动查看。

2024/5/1

支持查看 `console` 输出，在代码中直接使用 `console.log` 即可。

根据 [笔试「骗分」技巧](#) 中介绍的递归调试方法，原生的 `console.log` 方法被我加强了，可以自动根据递归深度给输出内容加上缩进，方便你观察递归过程。如果你不希望自动添加缩进，可以使用 `console._log` 方法。

具体介绍见 [可视化面板简介](#)。

2024/4/6

鼠标悬停在代码上方，支持跳转到上一次执行位置。

2024/4/3

- 支持 `@visualize hide` 和 `@visualize global` 注释，具体介绍见 [可视化面板简介](#)。

2024/3/18

- 支持 GitHub 登录。

2024/3/12

- 优化编译器效率，优化动画渲染速度。
- 提升动画的稳定性。

2024/3/5

- 可视化面板右下角添加刷新和全屏按钮，方便读者在插件中刷新面板或全屏显示。

- 优化高亮代码的跳转逻辑，当播放算法执行步骤时，避免左侧代码跳来跳去影响体验。

2024/2/17

修复部分代码编译时出现报错: `Error: No substitution given for "__X34"` 的 bug。

2024/2/1

新网站 [labuladong.online](#)、[vscode](#) 刷题插件、[Jetbrains](#) 插件、[Chrome](#) 插件 中的算法可视化面板添加「编辑」按钮，读者可以直接修改算法可视化面板的代码并执行。

2024/1/10

支持 `@visualize choice(nums[i])` 和 `@visualize unchoice(nums[i])` 注释，用于可视化回溯/动态规划算法的选择和撤销选择过程。鼠标移动到递归树节点上，会显示递归路径上所做的所有选择：

The screenshot shows a code editor and a visualization panel. The code editor contains a JavaScript implementation of the coin change problem using dynamic programming and recursion. The visualization panel shows a recursive tree where each node represents a choice of coins. Nodes are labeled with their state (e.g., choice = 2, amount = 1) and value (e.g., res = Infinity). The tree structure is as follows:

- Root node: coinChange = (1), amount = 1, res = Infinity.
  - Child node (4): choice = 2, amount = 1, res = Infinity.
    - Grandchild node (0): choice = 2, amount = 0, res = 1.
      - Leaf node (2): coins = [2]
      - Leaf node (-1): coins = [-1]
    - Grandchild node (1): choice = 2, amount = 1, res = 1.
      - Leaf node (1): coins = [1]
    - Grandchild node (-2): choice = 2, amount = 2, res = 2.
      - Leaf node (-2): coins = [-2]
  - Child node (6): choice = 2, amount = 2, res = 2.

网站/插件所有的可视化面板已更新这个新功能。具体介绍见 [可视化面板简介](#)。

- `@visualize` 关键词会用蓝色显示，方便用户确认递归追踪是否生效。

2024/1/7

- 修复可视化面板「上一步」按钮导致数据结构错乱的 bug。
- 模仿 IDE 代码调试的实用体验，优化可视化面板的代码显示效果：注释用灰色表示，已完成执行的代码用暗黄色显示，当前执行到的代码用绿色高亮显示，还未执行到的代码用白色显示。

The screenshot shows a code editor on the left and a visualization panel on the right. The code editor contains several snippets of JavaScript code related to data structures and algorithms. Annotations with red arrows point to specific parts of the code:

- An arrow points to a yellowed-out section of code: `// @visualize status(n)`.
- An arrow points to a grayed-out section of code: `注释为灰色`.
- An arrow points to a green-highlighted section of code: `当前步骤的代码为绿色高亮`.
- An arrow points to a white section of code: `还会执行的代码为白色`.

The visualization panel on the right shows a binary tree for the Fibonacci sequence (fibonacci.js) and a linked list (list.js). The tree has root node 5, with children 4 and 3. Node 4 has children 3 and 2, and so on. The linked list arr contains nodes 1, 4, 3, 2, 8. The global variable s is set to "你学会了吗？".

- 支持双链表的可视化。

The screenshot shows a code editor on the left and a visualization panel on the right. The code editor contains code for creating and manipulating a double-linked list (doubleList.js).

The visualization panel on the right shows a double-linked list with nodes 1, 2, 3, 4, 5. It also shows variables: list1 = 1, d3 = 3, node = 3, d2 = 2, nodeVal = 3, list2 = 4, d1 = 1, and s = "你学会了吗？".

2024/1/1

可视化面板正式上线，使用地址：

<https://labuladong.online/algo-visualize/>

# Chrome 刷题插件更新日志

本页面为 Chrome 插件的更新日志，插件的详细使用指南见[Chrome 插件说明书](#)。

## v5.0.6

- 修复部分题目不显示思路按钮的 bug。

## v5.0.5

- 修复 dev 版本的 Chrome 浏览器无法使用插件的 bug。
- 提升海外用户拉取网站数据的速度。

## v5.0.3

- 修复部分题目详情页中，思路/题解按钮显示错位的问题。

## v5.0.2

- 修复 dev 版本的 Chrome 浏览器无法使用插件的 bug。

## v5.0.1

- 修复收藏题目列表无法渲染思路/题解按钮的 bug。

## v4.4.7

大幅提升思路中多语言解法的准确性。

## v4.4.6

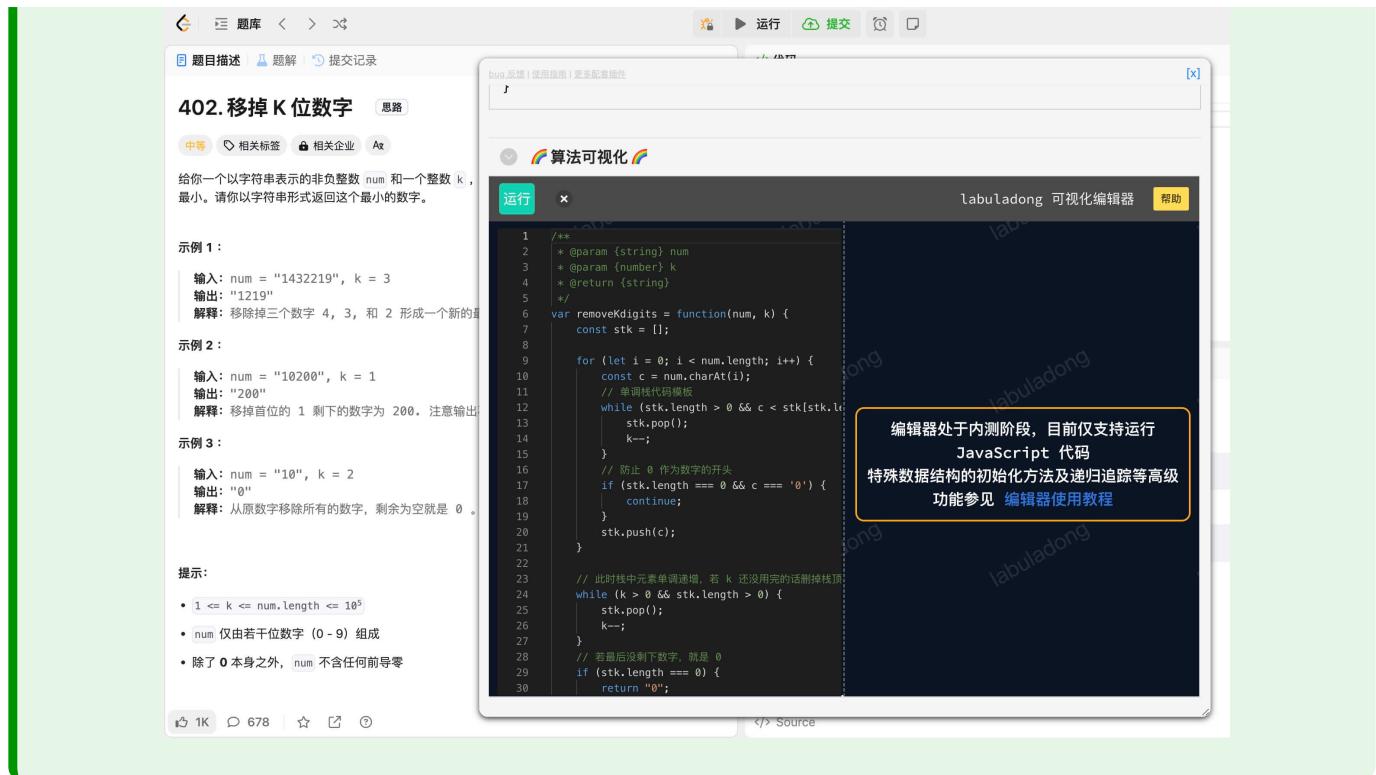
修复了插件思路中，文章链接还引用旧版课程链接的问题。按照[Chrome 插件说明书](#)中的方法，重新拉取数据即可。

## v4.4.5

- 优化数据拉取速度，优化若干用户体验。
- 修复 studyplan 插件无法生效的 bug。

## v4.4.0

可视化面板新增「编辑」按钮，支持修改我的解法代码后直接在面板上运行，方便验证你的奇思妙想：



插件适配了我的新网站 [labuladong.online](https://labuladong.online)，将不再支持小鹅通平台。

之前购买了小鹅通的读者，可以迁移课程权限到新网站，迁移指南 [见这里](#)。基于新网站解锁插件中课程专属题解的方法，已经在课程的第一章进行了更新。

## v4.3.7

- 修复 study plan 页面的 bug。
- 修复调整灵动布局后就无法显示思路题解按钮的 bug。

## v4.3.6

- 优化用户体验，自动判断网速快的数据源，并在安装后自动拉取数据。
- 适配新 UI 灵动布局（Dynamic layout）。
- 优化插件弹窗的显示状态，修复了之前版本显示错误的问题：

## labuladong 的刷题插件

如果插件好用, 请推荐给你的朋友。

基本信息:			
当前版本	最新版本	使用指南	bug 反馈
4.3.6	4.3.5	<a href="#">点这里</a>	<a href="#">点这里</a>

数据拉取信息:				
元数据	思路/题解	二叉树课程	数据结构课程	可视化面板
<input checked="" type="checkbox"/>				

多语言解法信息:				
java	cpp	python	javascript	go
<input checked="" type="checkbox"/>				

表示刷新成功, 表示刷新失败, 可以尝试修改数据源再刷新。  
[data-structure](#) 和 [tree](#) 是课程数据, 购买并登录后才能刷新成功。  
 仅在插件有异常时尝试点击刷新按钮。

[手动刷新数据](#)    数据源: Auto

付费课程权益信息:		
网站会员	数据结构精品课	二叉树递归专题课
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

表示刷新成功, 表示未购买或未登录课程。  
 刷新网站会员/课程, 在网页端解锁对应的内容。

[刷新网站 VIP/课程](#)

### v4.3.5

- 修复在网页写代码时卡顿的 bug。
- 支持 LeetCode/力扣 study plan 页面的渲染。

### v4.3.2

- Chrome 插件可以连接数据结构/二叉树课程, 在网页上学习课程:

首先, 要确保你登录了小鹅通平台, 然后点击「刷新网站课程」的按钮。  
 如果你已经购买了对应的课程, 就会显示内容解锁成功。  
 接下来, 刷新一下页面。  
 仅在插件有异常时尝试刷新, 表示未购买或未登录课程。  
 刷新网站 VIP 状态, 解锁全部网页文章。  
 刷新网站课程, 在网站解锁付费课程。  
 数据结构精品课 刷新成功, 网站内容已解锁。  
 二叉树递归专题课 刷新成功, 网站内容已解锁。

### v4.3.0

- 修复 Java 代码渲染颜色错误的 bug。

### v4.2.0

- 修复 LeetCode 和力扣的新 UI 导致插件失效的问题。

- 思路弹窗中能够保存用户的编程语言偏好，下次会默认打开对应语言的解法代码，不必频繁切换。
- 支持 400 道题的算法可视化，持续增加中：

The screenshot shows the 'Circular Linked List II' problem (142) on the labuladong online platform. The problem statement asks to find the entry node of a cycle in a linked list. The code provided uses快慢指针 (fast and slow pointers) to detect the cycle. A detailed diagram illustrates the cycle detection process, showing the movement of the pointers and the meeting point at the cycle's entrance.

## v4.1.0

- 修复剑指 offer 题目的思路解法无法显示的 bug。
- 支持所有主流编程语言的解法代码。

The screenshot shows the 'Circular Linked List II' problem (142) on the labuladong online platform. It displays a different solution for detecting a cycle in a linked list using the two-pointer technique. A detailed diagram illustrates the cycle detection process, showing the movement of the pointers and the meeting point at the cycle's entrance. The code is annotated with comments explaining the logic.

# vscode 刷题插件更新日志

本页面为 vscode 插件的更新日志，插件的详细使用指南见[vscode 插件说明书](#)。

## v2.0.3

- 修复中文力扣无法登录的问题 #1825。
- 修复了每次都会弹出报错窗口的问题 #1739。

## v2.0.2

- 修复优化了若干细节问题。
- 默认代码文件的命名规范改为题目名称，并在[vscode 插件说明书](#)中添加了相关说明。
- 提升海外用户拉取网站数据的速度。

## v2.0.0

- 借鉴了开源插件 [ccagml/leetcode-extension](#) 中的部分功能，包括自运行自定义测试用例、curl 登录等，修复英文版 LeetCode 无法登录的问题。
- 国际化：支持将 labuladong 题解及思路设置为英文。

现在 vscode 统一使用 curl 命令登录 力扣/LeetCode 账号，具体操作方法见[vscode 插件说明书](#)。

请使用 Chrome/Edge 等主流浏览器复制 curl 命令，因为非主流浏览器的开发者工具不完善，复制的 curl 命令可能是错误的。

2024/6/29

大幅提升思路中多语言解法的准确性。

2024/6/25

修复了插件思路中，文章链接还引用旧版课程链接的问题。按照[vscode 插件说明书](#)中的方法，重新拉取数据即可。

## v1.5.6

优化插件数据拉取速度。

## v1.5.5

支持[2024 新版网站会员](#)。

## v1.5.4

由于英文版 leetcode.com 添加了 cloudflare 的保护，导致插件原先的网络请求被拦截，无法正常登录和使用。本次更新修复了这个问题。

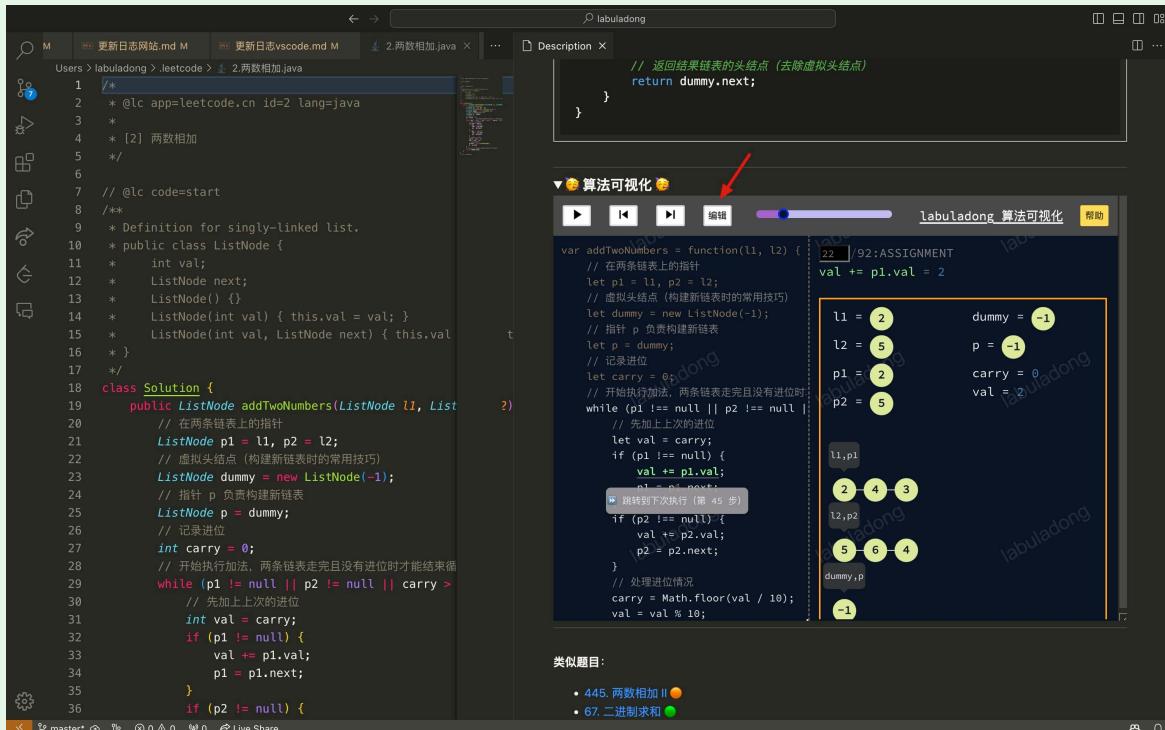
## v1.5.3

- 优化文章跳转链接的响应速度，修复部分用户反馈题解链接无法跳转的问题。
- 可视化面板支持全屏和刷新按钮。

## v1.5.1

放弃小鹅通和旧网站的支持，适配新网站 [labuladong.online](#)。在插件中解锁课程专属题解的方法有些变化，课程的第一章有具体介绍。

可视化面板新增「编辑」按钮，支持修改我的解法代码后直接在面板上运行，方便验证你的奇思妙想：



## v1.4.5

- 自动测网速，选择拉取可视化数据最快的端点。
- 修复了课程数据拉取无效的问题，优化用户提示。

## v1.4.3

- 优化部分用户提示，比如课程数据拉取失败时给出更用户友好的提示指引。
- 根据网速情况自动选择 [labuladong.gitee.io](#) 和 [labuladong.github.io](#) 作为数据源，解决部分国内用户拉取数据太慢的问题。

## v1.4.2

- 给中文力扣添加了 cookie 登录的方式，解决部分用户反馈在登录时遇到 `invalid password` 的问题。
- 修复小鹅通 cookie 无法拉取课程题解的 bug。

## v1.4.0

支持算法代码可视化：

The screenshot shows a Java code editor with the file `19.删除链表的倒数第-n个结点.java`. The code implements a solution for removing the  $n$ -th node from the end of a singly-linked list. It uses a dummy head node and two pointers, `p1` and `p2`, to find the target node. A separate panel displays the algorithm's logic and a visual representation of the list nodes.

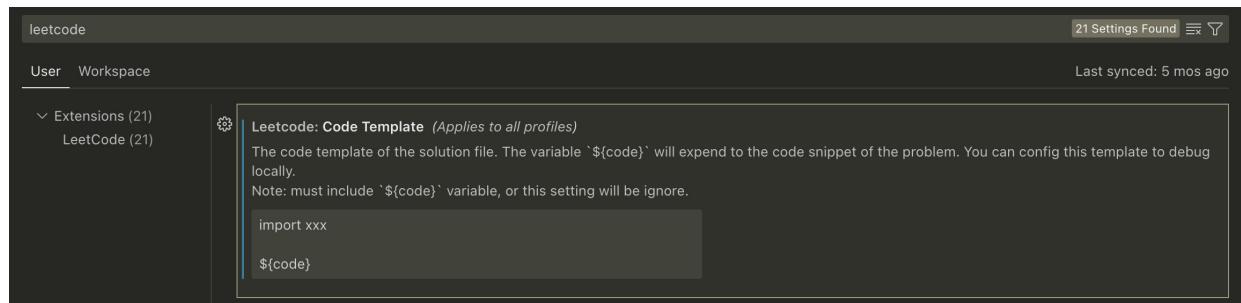
```

16     *      ListNode(int val, ListNode next) { this.val = val; this.next = next; }
17     *
18     */
19 class Solution {
20     // 主函数
21     public ListNode removeNthFromEnd(ListNode head, int n) {
22         // 虚拟头结点
23         ListNode dummy = new ListNode(-1);
24         dummy.next = head;
25         // 删掉倒数第 n 个，要先找倒数第 n + 1 个节点
26         ListNode x = findFromEnd(dummy, n + 1);
27         // 删掉倒数第 n 个节点
28         x.next = x.next.next;
29         return dummy.next;
30     }
31
32     // 返回链表的倒数第 k 个节点
33     ListNode findFromEnd(ListNode head, int k) {
34         ListNode p1 = head;
35         // p1 先走 k 步
36         for (int i = 0; i < k; i++) {
37             p1 = p1.next;
38         }
39         ListNode p2 = head;
40         // p1 和 p2 同时走 n - k 步
41         while (p1 != null) {
42             p2 = p2.next;
43             p1 = p1.next;
44         }
45         // p2 现在指向第 n - k 个节点
46         return p2;
47     }
48 }

```

## v1.3.3

- 多语言解法上面给出提示语，允许用户提交 pr 修正错误
- 添加 `codeTemplate` 设置，允许用户自定义代码模板进行本地调试：



## v1.3.2

- 每道题的思路都支持所有主流编程语言的代码。
- 修复了无法解锁课程题目思路的问题。

# Jetbrain 刷题插件更新日志

本页面为 Jetbrain 插件的更新日志，插件的详细使用指南见 [Jetbrain 插件说明书](#)。

## v2.0.2

- 将登录方式统一为 cookie 登录，修复了部分用户无法登录的问题。
- 提升海外用户拉取网站数据的速度。

## v2.0.0

- 国际化：支持将 labuladong 题解及思路设置为英文。

## v1.9.1

- 修复了之前版本无法正常登录/提交代码的问题。
- 修复了拉取网站会员数据报错 labuladong 网站会员 专属题解拉取失败，可能是网站 cookie 格式错误或者已过期，请更新网站 cookie 后尝试 ✗ 的问题。

插件兼容性改变为 2022.2+，即不再支持 2022.2 以下版本的 IDE。如无法更新插件，请升级 IDE 版本。

## v1.8.9

- 修复部分情况下登录力扣账号速度较慢的问题。

2024/6/29

大幅提升思路中多语言解法的准确性。

2024/6/25

- 修复了插件思路中，文章链接还引用旧版课程链接的问题。按照 [Jetbrain 插件说明书](#) 中的方法，重新拉取数据即可。

## v1.8.8

- 部分用户反馈 IDE 中可视化面板无法使用鼠标滚动。暂时不知道原因，但是我在可视化面板上添加了滚动条，如果你遇到这个问题，可以尝试用鼠标拖动滚动条滚动页面。
- 优化用户体验，提升数据拉取速度。

## v1.8.6

由于英文版 leetcode.com 添加了 cloudflare 的保护，导致插件原先的网络请求被拦截，无法正常登录和使用。本次更新修复了这个问题。

## v1.8.4

修复部分情况下无法点开插件配置页面的 bug。

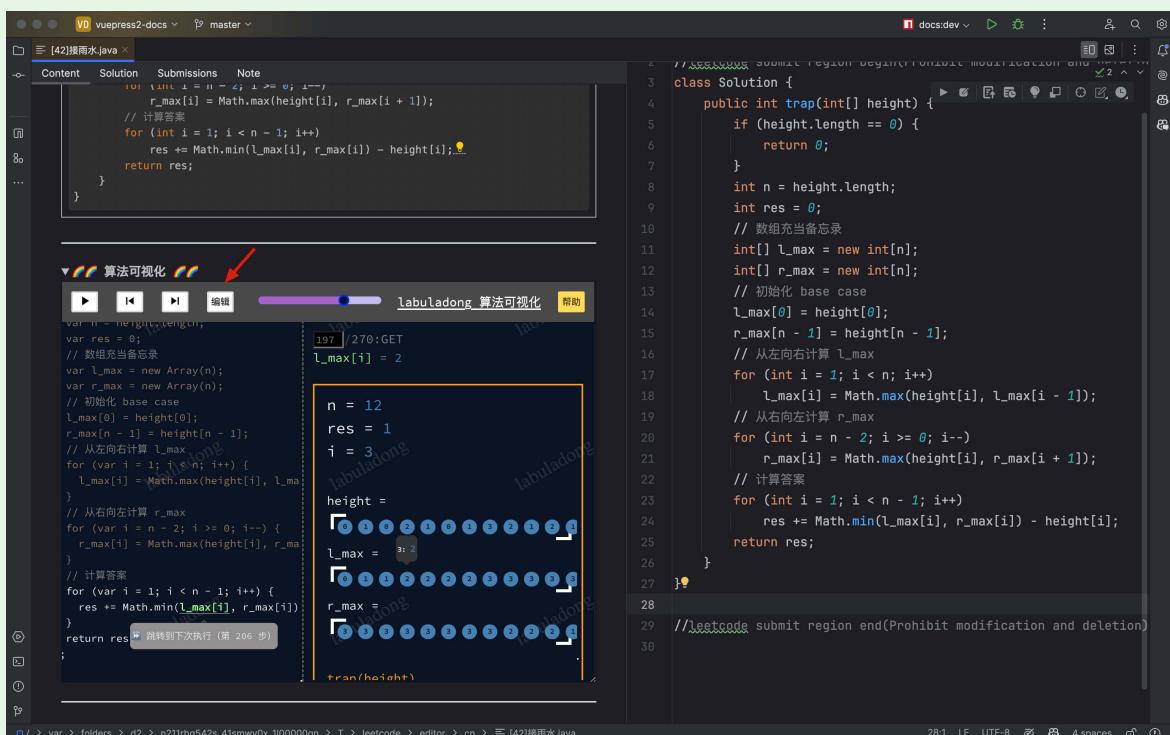
## v1.8.1

修复部分情况下可视化面板加载时会意外跳转到浏览器的情况！

## v1.8.0

放弃小鹅通和旧网站的支持，适配新网站 [labuladong.online](https://labuladong.online)。在插件中解锁课程专属题解的方法有些变化，课程的第一章有具体介绍。

可视化面板新增「编辑」按钮，支持修改我的解法代码后直接在面板上运行，方便验证你的奇思妙想：



## v1.7.6

修复最新版 2023.3 IDE 开启 new UI 后无法使用插件的 bug。

## v1.7.5

- 优化部分用户提示，比如课程数据拉取失败时给出更用户友好的提示指引。
- 有用户反馈，编辑器打开的代码题目页面的标题是力扣的题号和题目，不方便摸鱼刷题 😅。本次更新后，代码文件名的模板（Code File Name 配置）将同时作用于打开的页面标题。

## v1.7.3

修复可视化面板样式错乱的问题

```
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    return 1 + Math.max(leftMax, rightMax);
}
```

```
Definition for a binary tree node.
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }

    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 后序遍历位置顺便计算最大直径
        maxDiameter = Math.max(maxDiameter, leftMax + rightMax);
        return 1 + Math.max(leftMax, rightMax);
    }

    // 这是一种简单粗暴，但是效率不高的解法
    class BadSolution {
        public int diameterOfBinaryTree(TreeNode root) {
            if (root == null) {
                return 0;
            }
            // 计算出左右子树的最大高度
            int leftMax = maxDepth(root.left);
            int rightMax = maxDepth(root.right);
            // root 这个节点的直径
            int res = leftMax + rightMax;
            // 递归遍历 root.left 和 root.right 两个子树
            return Math.max(res,
                Math.max(diameterOfBinaryTree(root.left),
                    diameterOfBinaryTree(root.right)));
        }
    }
}
```

## v1.7.2

- 优化了可视化面板的展示逻辑，优化了数据拉取逻辑。

## v1.7.0

- 支持了算法可视化，但是还是有些偶发性 bug，未来逐步优化。

## v1.6.2

- 思路中支持显示所有主流编程语言的解法。

# 网站/插件问题反馈

随着我对算法教程的持续迭代，本站以及配套插件的功能越来越丰富，但也越来越复杂，难免出现 bug。

所以我需要及时接收大家的问题反馈，快速修复问题，避免给大家带来不良的使用体验。

大部分问题可能都是本地缓存了过期的内容导致的，所以在你反馈问题前，请先查看下面的常见问题列表，看看是否有解决方案。

## 邮箱/微信直接向我反馈

支付、权限、网站可用性，这几种直接影响使用的问题，可以通过邮件或微信反馈，我会第一时间处理：

- 邮箱：[labuladong@foxmail.com](mailto:labuladong@foxmail.com)
- 微信：仅支持付费会员添加，在[网站会员](#)页面可以添加我的微信或者进入本站的微信讨论群。

请具体描述你遇到的问题，最好带上截图和报错信息。支付/权限问题请带上你的[用户 ID](#) 和订单号。

## Github Issue 反馈

网站、配套工具的任何 bug、功能建议等，可以在 GitHub Issue 进行反馈，便于其他读者查看和讨论。

点击如下链接即可查看已有的 Issue：

<https://github.com/labuladong/fucking-algorithm/issues/>

点击 Issue 列表上方的 **New Issue** 按钮即可创建新的 Issue。

提 Issue 时注意以下几点：

- GitHub Issue 是公开访问的，切记不要暴露任何私密信息，包括用户 ID、订单号、账号密码、网站 token/cookie 等。
- 提出 Issue 前可以先搜索，看看是否有其他人遇到类似的问题，是否已有解决方案。

## 网站留言反馈

本站每篇文章下方都有留言区。对于笔误、代码小 bug 等不紧急的问题，可以直接在留言区评论反馈，我会定期查看留言并修复。

## 插件常见问题

插件的常见问题及解决方法在各个插件的使用说明中介绍，具体见[Chrome 插件使用说明](#), [vscode 插件使用说明](#), [JetBrains 插件使用说明](#)。

## 本站常见问题

### 页面卡死/加载不出来？

这一般是网络问题，海外读者和挂了海外代理的读者小概率会遇到。

因为本站更新较为频繁，如果由于本地缓存或 CDN 缓存的原因导致未加载最新资源，就会小概率出现无法访问某些页面的情况。

**解决方法：**请尝试清除浏览器缓存后刷新页面。如果还是不能解决问题，可以尝试更换网络，已确定是否是客户端网络的问题。

对于挂了海外代理的国内读者，请关闭代理访问本站，或者把本站 [labuladong.online](#) 移出代理规则，就不会出现这类问题了。

若依然无法解决，请向我反馈。

## 本站会员内容一直转圈，无法加载？

一般也是网络问题，解法同上。

## 可视化面板一直黑屏？

一般也是网络问题，导致未加载最新资源，解法同上。

## 购买本站会员后，插件专属题解未解锁？

插件中的专属题解需要你手动刷新数据才能解锁，具体看 [网站会员](#) 介绍的具体操作方法。

## 登录时报错 **仅海外 IP 可以使用 GitHub 登录**

由于网络原因，登录时会进行 IP 检测，仅支持海外 IP 使用 GitHub 登录方式。国内用户请使用微信登录的方式。

目前不同登录方式会产生不同的网站用户，购买记录不互通。如需要将权限在 GitHub 账号/微信账号之间转移，请联系我后台处理。