

# LeetCode 例题精讲 | 18 前缀和：空间换时间的技巧

原创 nettee 面向大象编程 1周前

来自专辑

LeetCode 例题精讲



本文将教会你「前缀和」的算法套路，做出以下 LeetCode 例题：

- **LeetCode 724. Find Pivot Index** (Easy)
- **LeetCode 560. Subarray Sum Equals K** 和为K的子数组 (Medium)

在设计算法时，时间复杂度始终是我们关注的重点。我们需要让算法的时间复杂度尽可能低，追求运行效率。有些时候，我们可以通过增加空间占用的方式减少算法的运行时间，这便是空间换时间。

动态规划就是一类空间换时间的算法。动态规划通过保存所有子问题的计算结果，可以避免子问题的重复计算。这种方法的代价是 **DP 数组** 占用了较多的空间。

前缀和同样也是一种空间换时间的技巧，只不过我们使用的不是 **DP 数组**，而是「前缀和数组」。

那么，究竟什么是前缀和呢？

## 什么是前缀和

我们先用一道简单的题目理解一下「前缀和」究竟是做什么的：

## LeetCode 303. Range Sum Query - Immutable (Easy)

给定一个整数数组 `nums`，求出数组从索引  $i$  到  $j$  ( $i \leq j$ ) 范围内元素的总和，包含  $i$ 、 $j$  两点。

示例：

给定 `nums = [-2, 0, 3, -5, 2, -1]`，求和函数为 `sumRange()`

```
sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

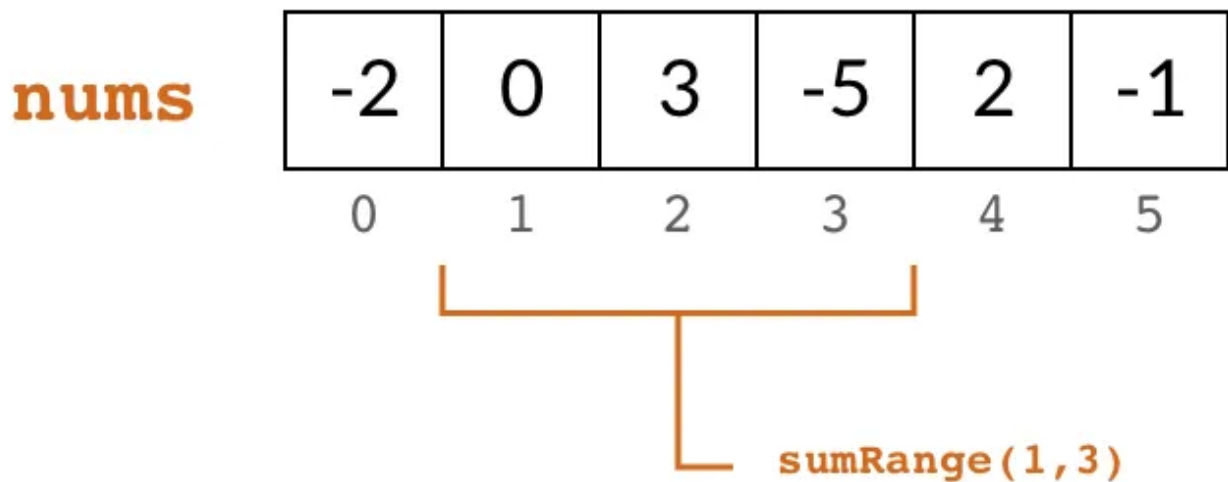
说明：

- 假设数组不可变
- 会多次调用 `sumRange` 函数

这道题目的解法很直白，难点在于如何减少时间复杂度。我们来看看不同的解法的时间、空间复杂度有何区别。

解法一：暴力法

如果用暴力解法，每次调用 `sumRange` 时，都使用 `for` 循环将  $i$  到  $j$  之间的元素相加。



解法一：暴力法

```
public int sumRange(int i, int j) {
    int sum = 0;
    for (int k = i; k <= j; k++) {
```

```

    for (int k = i; k <= j; k++) {
        sum += nums[k];
    }
    return sum;
}

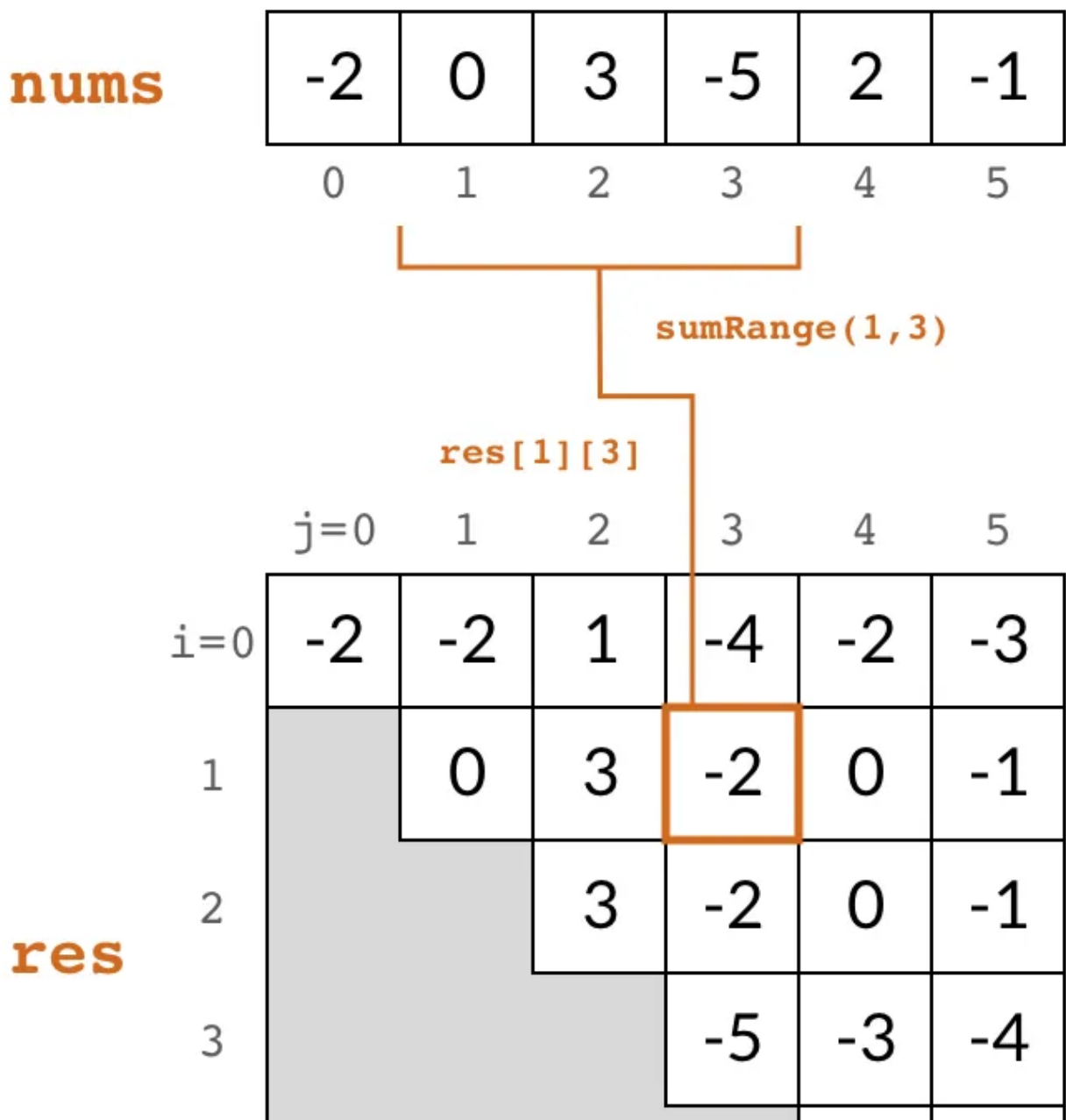
```

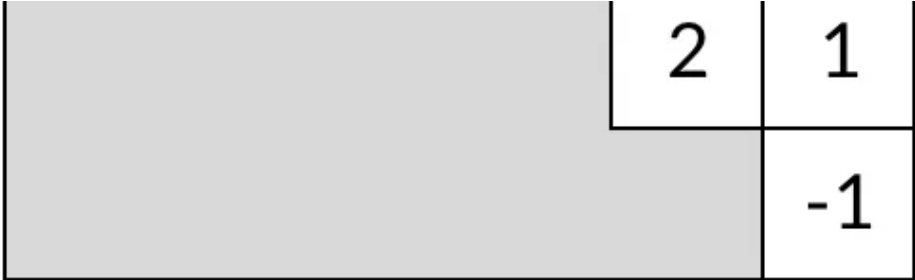
这样的话，每次查询（即每次调用 `sumRange`）平均需要  $O(n)$  的时间。由于 `sumRange` 函数会被多次调用，这种算法的时间开销会非常大。

## 解法二：空间换时间

在 `sumRange` 会被调用很多次的情况下，我们要尽可能地减少一次调用的时间。如果多次调用 `sumRange` 的参数是重复的，但还需要重新求和，就会做很多重复的计算。

为了避免重复的计算，我们可以对数组 `nums` 进行预处理，预先存储计算结果。我们使用二维数组 `res` 存储预处理的结果，`res[i][j]` 存储 `sumRange(i, j)` 的返回值。



4		
5		

解法二：空间换时间

```
private int[][] res;

// 预处理阶段
public NumArray(int[] nums) {
    int n = nums.length;
    res = new int[n][n];
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += nums[j];
            res[i][j] = sum;
        }
    }
}

public int sumRange(int i, int j) {
    return res[i][j];
}
```

这个解法的复杂度分析要区分「预处理阶段」和「查询阶段」：

- 预处理阶段：时间复杂度  $O(n^2)$ ，空间复杂度  $O(n^2)$ ；
- 查询阶段：每次查询需要  $O(1)$  时间。

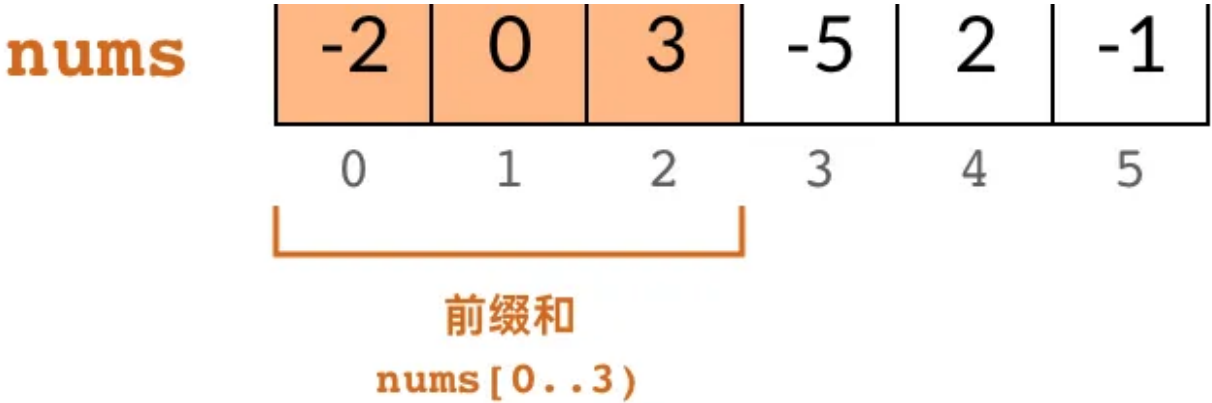
通过预处理，我们实现了空间换时间，每次查询的时间开销降到了最小。然而这种解法要存储所有可能的结果，空间占用太大。有没有空间占用小一点的方法呢？

解法三：前缀和

上一个解法中的预处理方法过于暴力，会空间占用太大。我们还可以使用另一个更聪明的预处理方法：前缀和。

所谓前缀和（prefix sum），就是数组开头的若干连续元素的和。





前缀和的定义

在预处理的时候，我们求出数组 `nums` 的全部前缀和，放在数组 `preSum` 中。`preSum[k]` 表示 `nums` 的前  $k$  个元素（即 `nums[0..k)`）的元素之和，其中  $0 \leq k \leq n$ 。

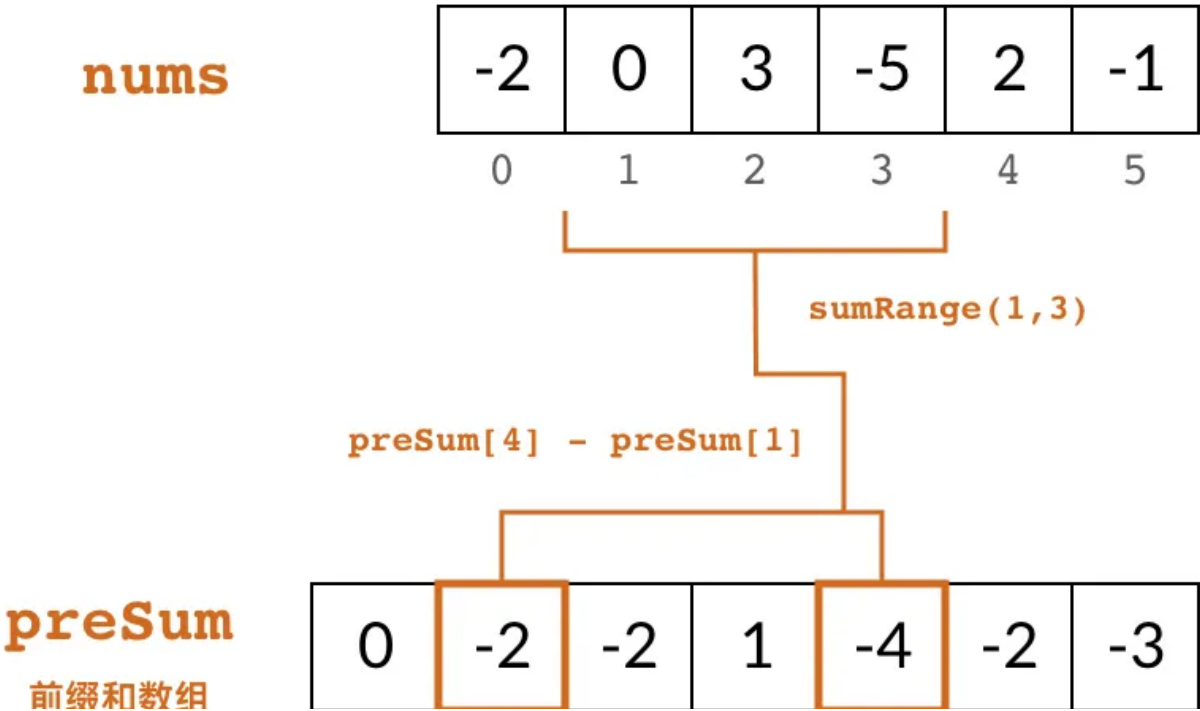
语言小贴士：

前缀和数组的定义中使用了左闭右开区间。这种表示方法的优点之一是很容易做区间的减法。例如：`nums[0..j) - nums[0..i)` 可以得到 `nums[i..j)`。在滑动窗口类题目中也经常使用左闭右开区间。

前缀和数组的聪明之处在哪里呢？

首先，通过两个前缀和相减就可以很快求出数组中从  $i$  到  $j$  的元素之和，只需要  $O(1)$  的时间：

sumRange(i, j) = preSum[j+1] - preSum[i]



0 1 2 3 4 5 6

两个前缀和相减求出元素之和

其次，前缀和数组只占用  $O(n)$  的空间，计算前缀和数组也很简单，只需要  $O(n)$  的时间：

```
int n = nums.length;
// 计算前缀和数组
int[] preSum = new int[n+1];
preSum[0] = 0;
for (int i = 0; i < n; i++) {
    preSum[i+1] = preSum[i] + nums[i];
}
```

最终的题解代码如下所示：

```
class NumArray {

    private int[] preSum;

    // 预处理阶段
    public NumArray(int[] nums) {
        int n = nums.length;
        // 计算前缀和数组
        preSum = new int[n+1];
        preSum[0] = 0;
        for (int i = 0; i < n; i++) {
            preSum[i+1] = preSum[i] + nums[i];
        }
    }

    public int sumRange(int i, int j) {
        return preSum[j+1] - preSum[i];
    }
}
```

我们可以对比一下三种解法的时间、空间复杂度。

解法	空间复杂度	查询时间复杂度
解法一：暴力法	$O(1)$	$O(n)$
解法二：暴力时间换空间	$O(n^2)$	$O(1)$
解法三：前缀和	$O(n)$	$O(1)$

可以看到，前缀和方法的特点是：能优化时间复杂度，同时让空间复杂度不会太大。这让前缀和成为一个很实用的数组预处理手段。

## 前缀和的应用

下面，我们用两道典型题目来看看前缀和的应用场景。这两道题分别是「寻找枢纽元素」以及「和为K的子数组」。

前缀和方法的典型使用场景是数组类题目。当看到题目与「子数组求和」有关，就要想想能不能使用前缀和来求解。

### 例题一：寻找枢纽元素

**LeetCode 724. Find Pivot Index** 寻找枢纽元素（Easy）

给定一个整数类型的数组 `nums`，返回数组的「枢纽元素」。

数组的「枢纽元素」定义为：对于数组中的某个元素  $x$ ，若  $x$  左侧所有元素之和等于右侧所有元素之和，则  $x$  为枢纽元素。

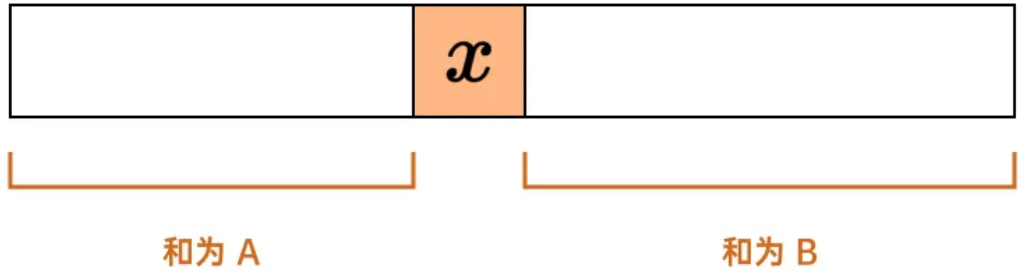
如果数组不存在枢纽元素，则返回 -1。如果数组有多个中心索引，则返回最左边一个。

示例：

输入：`nums = [1, 7, 3, 6, 5, 6]`  
输出：3  
解释：索引 3 (`nums[3] = 6`) 的左侧数之和 (`1 + 7 + 3 = 11`)，与右侧数之和 (`5 + 6 = 11`) 相等。

对于题目中定义的「枢纽元素」，我们可以用一张图来理解：

nums



枢纽元素的含义

设枢纽元素  $x$  左侧的元素之和为  $A$ ，右侧的元素之和为  $B$ 。 $x$  为枢纽元素需要  $A = B$ 。

这道题关注的是  $x$  左右两侧的「元素之和」，因此可以考虑用前缀和的技巧来求解。我们发现， $x$  左侧的元素之和  $A$  就已经满足前缀和的定义，那么我们以  $A$  为核心思考解题方法。

$x$  右侧的元素之和  $B$  可以直接由  $A$  求出来。我们设数组的所有元素之和为  $S$ （这个值可以一开始先求出来），则  $B$  可以表示为  $S - A - x$ 。枢纽元素  $x$  又需要  $A = B$ ，那么我们可以得到

$$A = B = S - A - x$$

化简得到

$$2A + x = S$$

也就是说，前缀和（ $A$ ）与下一个元素（ $x$ ）满足以上的关系时，元素  $x$  即为枢纽元素。我们可以在不断求前缀和的过程中判断以上关系是否满足。

最终得到的题解代码如下：

```
public int pivotIndex(int[] nums) {
    // 首先计算所有元素之和 S
    int S = 0;
    for (int n : nums) {
        S += n;
    }

    int A = 0; // A 为前缀和
    // 迭代计算前缀和
    for (int i = 0; i < nums.length; i++) {
        int x = nums[i];
        if (2 * A + x == S) {
            // 满足公式中的关系, x 是枢纽元素
            return i;
        }
        A += x; // 计算前缀和
    }
    return -1;
}
```



}

这道题的重点在于意识到  $A$  就是前缀和，然后应用前缀和的技巧就可以迎刃而解了。

## 例题二：和为K的子数组

### LeetCode 560. Subarray Sum Equals K 和为K的子数组（Medium）

给定一个整数数组 `nums` 和一个整数  $k$ ，返回该数组中「和为  $k$  的连续子数组」的个数。

示例：

输入：nums = [1,1,1], k = 2

输出：2

解释：[1,1] 与 [1,1] 为两种不同的情况。

这道题关注就是「子数组的元素之和」，显然又是一道可以使用前缀和技巧的题目。

我们可以首先求出所有的前缀和，然后根据前缀和求出所有可能的子数组之和，题解代码如下所示：

```
public int subarraySum(int[] nums, int k) {  
    int N = nums.length;  
  
    // 计算前缀和数组  
    // presum[k] 表示元素 nums[0..k) 之和  
    int[] presum = new int[N+1];  
    int sum = 0;  
    for (int i = 0; i < N; i++) {  
        presum[i] = sum;  
        sum += nums[i];  
    }  
    presum[N] = sum;  
  
    // sum of nums[i..j) = sum of nums[0..j) - sum of nums[0..i)  
    int count = 0;  
    for (int i = 0; i <= N; i++) {  
        for (int j = i+1; j <= N; j++) {  
            // 前缀和相减求子数组之和  
            if (presum[j] - presum[i] == k) {
```

```

        count++;
    }
}
return count;
}

```

这个解法的时间复杂度是  $O(n^2)$ ，空间复杂度是  $O(n)$ ，并不是最优的解法。

虽然我们用前缀和简化了子数组求和的过程，但是因为要穷举所有可能的子数组，还是使用了二重循环，时间复杂度达到了平方级别的  $O(n^2)$ 。要进一步减少时间复杂度，需要用到哈希表的技巧。

### 注意：

哈希表技巧不是本文的重点，这里只是介绍本题的最优解法。关于哈希表的相关技巧在后面的文章中会有专门的讲解。

我们再仔细看一下上面代码中的二重循环：

```

for (int i = 0; i <= N; i++) {
    for (int j = i+1; j <= N; j++) {
        // 前缀和相减求子数组之和
        if (presum[j] - presum[i] == k) {
            count++;
        }
    }
}

```

为了减少时间复杂度，我们的目标是把二重循环变为一重循环。

我们将条件判断  $\text{presum}[j] - \text{presum}[i] == k$  简单移项，可以得到  $\text{presum}[i] == \text{presum}[j] - k$ 。那么我们的循环完全可以把  $i$ 、 $j$  颠倒，写成这样：

```

for (int j = 1; j <= N; j++) {
    for (int i = 0; i < j; i++) {
        if (presum[i] == presum[j] - k) {
            count++;
        }
    }
}

```

内层循环实际上是在求「有多少个  $i$  满足  $\text{presum}[i]$  的值为  $\text{presum}[j] - k$ 」。而我们可以通过用哈希表存储每一个  $\text{presum}[i]$  的值，直接找到满足条件的  $\text{presum}[i]$  的个数，而不需要写一个循环。

我们使用一个哈希表，在计算前缀和的同时把前缀和的每个值出现的次数都记录在哈希表中，得到以下的题解代码：

```
public int subarraySum(int[] nums, int k) {  
    // 前缀和 -> 该前缀和（的值）出现的次数  
    Map<Integer, Integer> presum = new HashMap<>();  
    // base case, 前缀和 0 出现 1 次  
    presum.put(0, 1);  
  
    int sum = 0; // 前缀和  
    int res = 0;  
    for (int n : nums) {  
        sum += n; // 计算前缀和  
        // 查找有多少个 sum[i] 等于 sum[j] - k  
        if (presum.containsKey(sum - k)) {  
            res += presum.get(sum - k);  
        }  
        // 更新 sum[j] 的个数  
        if (presum.containsKey(sum)) {  
            presum.put(sum, presum.get(sum) + 1);  
        } else {  
            presum.put(sum, 1);  
        }  
    }  
    return res;  
}
```

这样，我们就把时间复杂度优化到了  $O(n)$ ，空间复杂度依然保持在  $O(n)$ 。

## 总结

本文介绍了前缀和的技巧，以及相关的两道例题：LeetCode 724. 寻找枢纽元素、LeetCode 560. 和为K的子数组。这两道题目都有一个共同点：对子数组求和。我们在做题的时候，只要遇到与「子数组求和」相关的题目，就考虑一下使用前缀和方法会如何，一定没有错。

前缀和技巧掌握起来并不难，但是需要一定的经验才能在题目中灵活运用。还没有用过前缀和的同学，建议自己做一遍这两道例题，体会一下前缀和在时间复杂度的优化。

前缀和是一个「小方法」，在解题过程中往往是进行局部的优化，一般还需要和其他方法一起使用。例如 560 题就还需要利用哈希表做进一步的优化，以消除不必要的循环。与哈希表相关的技巧将在后续的文章中进一步介绍。

## 往期文章

- [经典动态规划：「换硬币」系列三道问题详解](#)
- [一文教你股票买卖问题实用而装逼的解法](#)
- [LeetCode 例题精讲 | 17 动态规划如何拆分子问题，简化思路](#)

我是 nettee，致力于分享面试算法的解题套路，让你真正掌握解题技巧，做到举一反三。我的《LeetCode 例题精讲》系列文章正在写作中，关注我的公众号，获取最新文章。

# 面向大象编程

带你刷 LeetCode  
让算法题不再难



扫码关注公众号

原创不易，欢迎分享、点赞和「在看」 ↴