

滑动窗口算法解决子串问题

原创：labuladong labuladong 6月19日

预计阅读时间：9 分钟

现在写滑动窗口算法的时机应该是合适的，因为读者已经通过 [双指针技巧汇总](#) 理解了双指针的套路，通过 [单调队列解决滑动窗口问题](#) 对滑动窗口这个东西有了个印象，而且通过 [一个方法团灭股票问题](#) 看到了成体系方法举一反三的威力。

本文详解「滑动窗口」这种高级双指针技巧的算法框架，带你秒杀几道高难度的子字符串匹配问题。

LeetCode 上至少有 9 道题目可以用此方法高效解决。但是有几道是 VIP 题目，有几道题目虽不难但太复杂，所以本文只选择点赞最高，较为经典的，最能够讲明白的三道题来讲解。第一题为了让读者掌握算法模板，篇幅相对长，后两题就基本秒杀了。文章最后抽象出一个简单的算法框架。

本文代码为 C++ 实现，不会用到什么编程方面的奇技淫巧，但是还是简单介绍一下一些用到的数据结构，以免有的读者因为语言的细节问题阻碍对算法思想的理解：

`unordered_map` 就是哈希表（字典），它的一个方法 `count(key)` 相当于 `containsKey(key)` 可以判断键 `key` 是否存在。

可以使用方括号访问键对应的值 `map[key]`。需要注意的是，如果该 `key` 不存在，C++ 会自动创建这个 `key`，并把 `map[key]` 赋值为 0。

所以代码中多次出现的 `map[key]++` 相当于 Java 的 `map.put(key, map.getOrDefault(key, 0)+1)`。

PS：本文大的主要代码都是图片形式，可以点开放大，更重要的是可以左右滑动方便对比代码。

一、最小覆盖子串

给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

示例：

输入：S = "ADOBECODEBANC", T = "ABC"
输出："BANC"

说明：

- 如果 S 中不存这样的子串，则返回空字符串 ""。
- 如果 S 中存在这样的子串，我们保证它是唯一的答案。

题目不难理解，要求在串 S(source) 中找到包含串 T(target) 中全部字母的一个子串，顺序无所谓，但这个子串得是所有可能子串中最短的。

此题难度 **Hard**，但是因为很有代表性，所以放到第一道。

如果我们使用暴力解法，代码大概是这样的：

```
for (int i = 0; i < s.size(); i++)  
    for (int j = i + 1; j < s.size(); j++)  
        if s[i:j] 包含 t 的所有字母:  
            更新答案
```

思路很直接吧，但是显然，这个算法的复杂度肯定大于 $O(N^2)$ 了，不好。

滑动窗口算法的思路是这样：

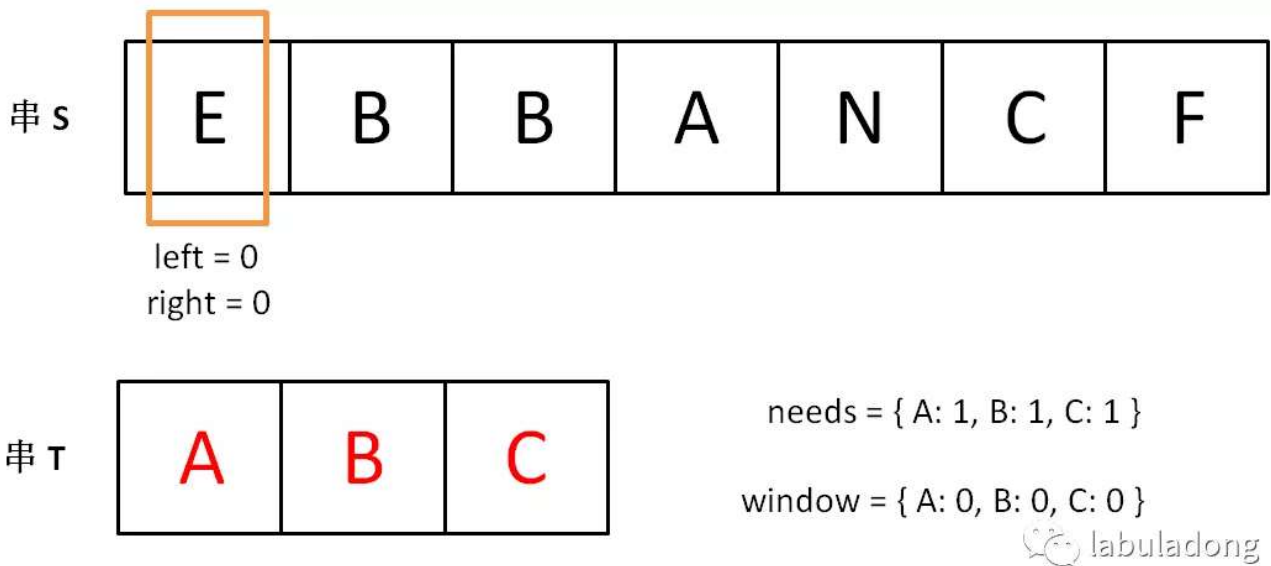
- 1、我们在字符串 S 中使用双指针中的左右指针技巧，初始化 $left = right = 0$ ，把索引闭区间 $[left, right]$ 称为一个「窗口」。
- 2、我们先不断地增加 right 指针扩大窗口 $[left, right]$ ，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。
- 3、此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 $[left, right]$ ，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 right 到达字符串 S 的尽头。

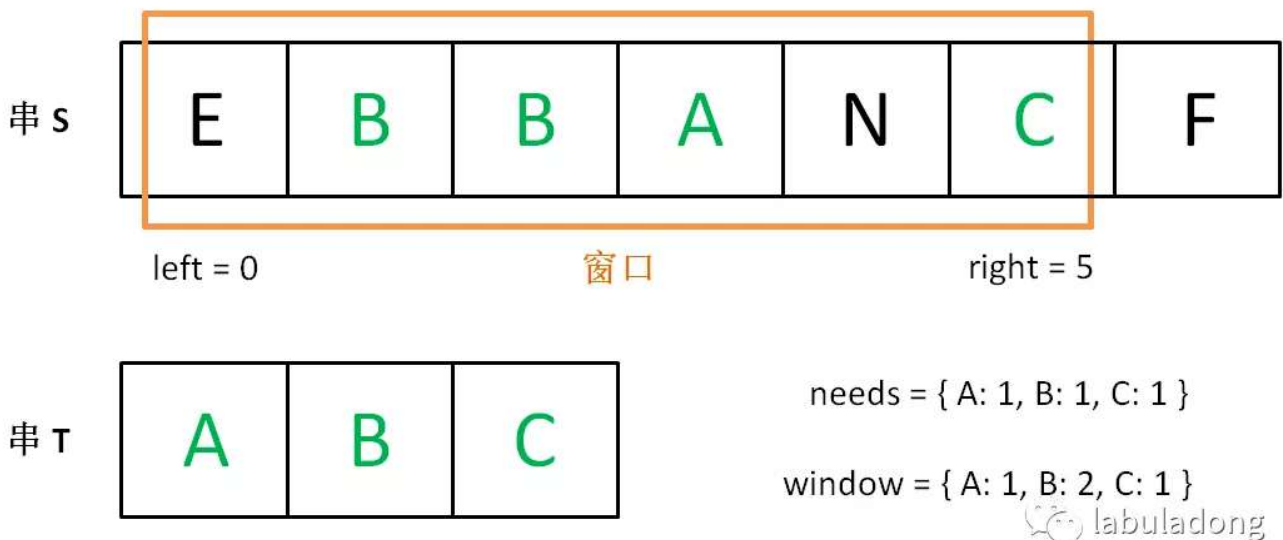
这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

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。

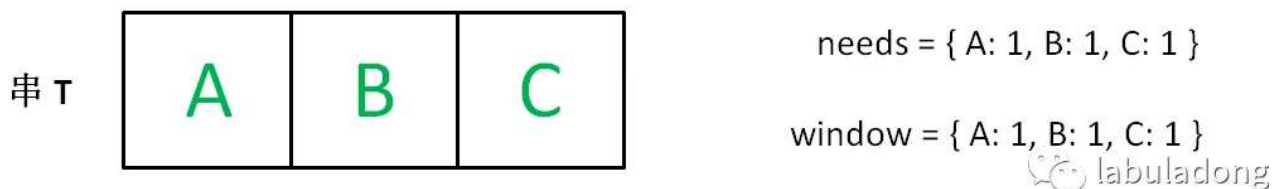
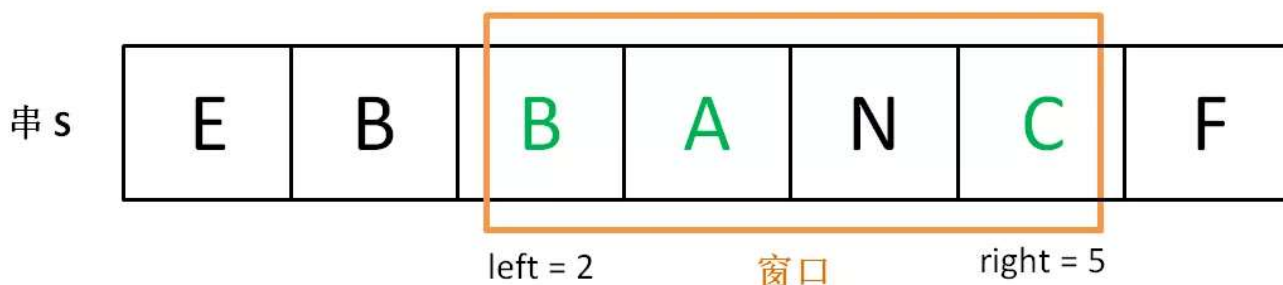
初始状态：



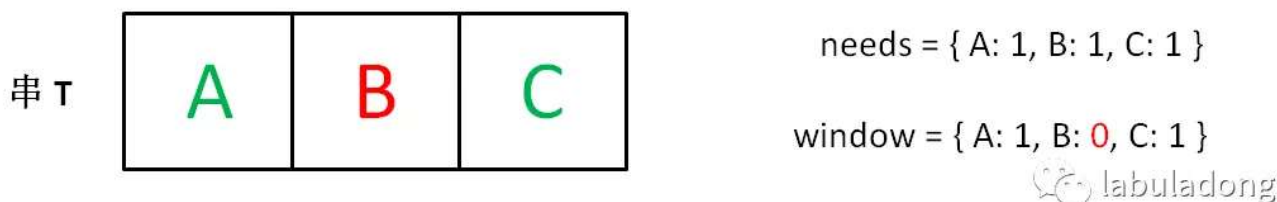
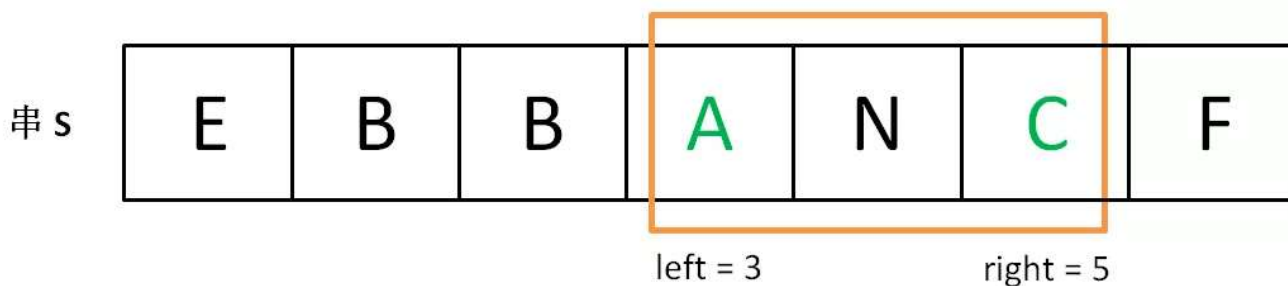
增加 right，直到窗口 [left, right] 包含了 T 中所有字符：



现在开始增加 left，缩小窗口 [left, right]（此时窗口里的字符串是最优解）：



直到窗口中的字符串不再符合要求，left 不再继续移动：



之后重复上述过程，再移动 right 试图使窗口中的字符再次符合要求，之后移动 left 缩小窗口... 直到 right 指针到达字符串 S 的末端，算法结束。

在算法执行过程中，就可以找到长度最短符合条件的子串（窗口）。

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。至于如何具体到问题，如何得出此题的答案，都是编程问题，很容易解决。

上述过程可以简单地写出如下伪码框架：

```
string s, t;  
// 在 s 中寻找 t 的「最小覆盖子串」  
int left = 0, right = 0;
```

```

string res = s;
// 先移动 right 寻找可行解
while(right < s.size()) {
    window.add(s[right]);
    right++;
    // 找到可行解后，开始移动 left 缩小窗口
    while (window 符合要求) {
        // 如果这个窗口的子串更短，则更新结果
        res = minLen(res, window);
        window.remove(s[left]);
        left++;
    }
}
return res;

```

如果上述代码你也能够理解，那么你离解题更近了一步。现在就剩下一个比较棘手的问题：

如何判断 window 即子串 `s[left...right]` 是否符合要求，即是否包含 `t` 的所有字符呢？

可以用两个哈希表当作计数器解决。用一个哈希表 `needs` 记录字符串 `t` 中包含的字符及出现次数，用另一个哈希表 `window` 记录当前「窗口」中包含的字符及出现的次数。如果 `window` 包含所有 `needs` 中的键，且这些键对应的值都大于等于 `needs` 中的值，那么就可以知道当前「窗口」符合要求了，可以开始移动 `left` 指针了。

现在将上面的框架继续细化：

```

string s, t;
// 在 s 中寻找 t 的「最小覆盖子串」
int left = 0, right = 0;
string res = s;

// 相当于两个计数器
unordered_map<char, int> window;
unordered_map<char, int> needs;
for (char c : t) needs[c]++;

// 记录 window 中已经有多少字符符合要求了
int match = 0;

```

```

while (right < s.size()) {
    char c1 = s[right];
    if (needs.count(c1)) {
        window[c1]++; // 加入 window
        if (window[c1] == needs[c1])
            // 字符 c1 的出现次数符合要求了
            match++;
    }
    right++;

    // window 中的字符串已符合 needs 的要求了
    while (match == needs.size()) {
        // 更新结果 res
        res = minLen(res, window);
        char c2 = s[left];
        if (needs.count(c2)) {
            window[c2]--; // 移出 window
            if (window[c2] < needs[c2])
                // 字符 c2 出现次数不再符合要求
                match--;
        }
        left++;
    }
}
return res;

```

上述代码已经具备完整的逻辑了，只有一处伪码，即更新最短子串结果 `res` 的地方，不过这个问题太好解决了，直接看完整解法吧！

```

string minWindow(string s, string t) {
    // 记录最短子串的开始位置和长度
    int start = 0, minLen = INT_MAX;
    int left = 0, right = 0;

```

```

unordered_map<char, int> window;
unordered_map<char, int> needs;
for (char c : t) needs[c]++;

int match = 0;

while (right < s.size()) {
    char c1 = s[right];
    if (needs.count(c1)) {
        window[c1]++;
        if (window[c1] == needs[c1])
            match++;
    }
    right++;

    while (match == needs.size()) {
        if (right - left < minLen) {
            // 更新最小子串的位置和长度
            start = left;
            minLen = right - left;
        }
        char c2 = s[left];
        if (needs.count(c2)) {
            window[c2]--;
            if (window[c2] < needs[c2])
                match--;
        }
        left++;
    }
}
return minLen == INT_MAX ?
    "" : s.substr(start, minLen);
}

```


如果直接甩给你这么一大段代码，我想你的心态是爆炸的。但是通过之前的步步跟进，你应该能够理解这个算法的内在逻辑，能清晰看出该算法的结构了。

这个算法的时间复杂度是 $O(M + N)$ ， M 和 N 分别是字符串 S 和 T 的长度。因为我们先用 `for` 循环遍历了字符串 T 来初始化 `needs`，时间 $O(N)$ ，之后的两个 `while` 循环最多执行 $2M$ 次，时间 $O(M)$ 。

读者也许认为嵌套的 `while` 循环复杂度应该是平方级，但是你这样想，`while` 执行的次数就是双指针 `left` 和 `right` 走的总路程，最多是 $2M$ 嘛。

二、找到字符串中所有字母异位词

给定一个字符串 s 和一个非空字符串 t ，找到 s 中所有是 t 的字母异位词的字串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 t 的长度都不超过 20100。

说明：

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

示例 1:

输入：

`s: "cbaebabacd" t: "abc"`

输出：

`[0, 6]`

解释：

起始索引等于 0 的子串是 "cba"，它是 "abc" 的字母异位词。

起始索引等于 6 的子串是 "bac"，它是 "abc" 的字母异位词。

示例 2:

输入：

`s: "abab" t: "ab"`

输出：

`[0, 1, 2]`

解释：

起始索引等于 0 的子串是 "ab"，它是 "ab" 的字母异位词。

起始索引等于 1 的子串是 "ba"，它是 "ab" 的字母异位词。

起始索引等于 2 的子串是 "ab"，它是 "ab" 的字母异位词。

这道题的难度是 **Easy**，但是评论区点赞最多的一条是这样：

How can **this** problem be marked **as** easy?

实际上，这个 **Easy** 是属于了解双指针技巧的人的，只要把上一道题的代码改中更新结果的代码稍加修改就成了这道题的解：

```
vector<int> findAnagrams(string s, string t) {  
    // 用数组记录答案  
    vector<int> res;  
    int left = 0, right = 0;  
    unordered_map<char, int> needs;  
    unordered_map<char, int> window;  
    for (char c : t) needs[c]++;  
    int match = 0;  
  
    while (right < s.size()) {  
        char c1 = s[right];  
        if (needs.count(c1)) {  
            window[c1]++;  
            if (window[c1] == needs[c1])  
                match++;  
        }  
        right++;  
  
        while (match == needs.size()) {  
            // 如果 window 的大小合适  
            // 就把起始索引 left 加入结果  
            if (right - left == t.size()) {  
                res.push_back(left);  
            }  
            char c2 = s[left];  
            if (needs.count(c2)) {  
                window[c2]--;  
                if (window[c2] < needs[c2])  
                    match--;  
            }  
            left++;  
        }  
    }  
    return res;  
}
```

因为这道题和上一道的场景类似，也需要 window 中包含串 t 的所有字符，但上一道题要找长度最短的子串，这道题要找长度相同的子串，也就是「字母异位词」嘛。

三、无重复字符的最长子串

给定一个字符串，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: "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: "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。

此题难度 **Medium**，遇到子串问题，首先想到的就是滑动窗口技巧。

类似之前的思路，使用 window 作为计数器记录窗口中的字符出现次数，然后先向右移动 right，当 window 中出现重复字符时，开始移动 left 缩小窗口，如此往复：

```

int lengthOfLongestSubstring(string s) {
    int left = 0, right = 0;
    unordered_map<char, int> window;
    int res = 0; // 记录最长长度

    while (right < s.size()) {
        char c1 = s[right];
        window[c1]++;
        right++;
        // 如果 window 中出现重复字符
        // 开始移动 left 缩小窗口
        while (window[c1] > 1) {
            char c2 = s[left];
            window[c2]--;
            left++;
        }
        res = max(res, right - left);
    }
    return res;
}

```

需要注意的是，因为我们要求的是最长子串，所以需要在每次移动 right 增大窗口时更新 res，而不是像之前的题目在移动 left 缩小窗口时更新。

四、模板总结

通过上面三道题，我们可以总结出滑动窗口算法的抽象思想：

```

int left = 0, right = 0;

while (right < s.size()) {
    window.add(s[right]);
    right++;
}

```

```
while (valid) {  
    window.remove(s[left]);  
    left++;  
}  
}
```

其中 window 的数据类型可以视具体情况而定，比如上述题目都使用哈希表充当计数器，当然你也可以用一个数组实现同样效果，因为我们只处理英文字母。

另外，滑动窗口技巧也可以运用在数组中，比如给一个数组，求其中 sum 最大的子数组，或者求平均数最大的子数组，都可以用滑动窗口技巧解决。

稍微麻烦的地方就是这个 valid 条件，为了实现这个条件的实时更新，我们可能会写很多代码。比如前两道题，看起来解法篇幅那么长，实际上思想还是很简单，只是大多数代码都在处理这个问题而已。

[点击这里进入留言板](#)

