滑动窗口算法通用思想

文章目录

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

最小覆盖子串
找到字符串中所有字母异位词
无重复字符的最长子串

最后抽象出一个简单的滑动窗口算法框架。

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。

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)。

本文大部分代码都是图片形式，可以点开放大，更重要的是可以左右滑动方便对比代码。下面进入正题。

一、最小覆盖子串

题目链接
题目不难理解，就是说要在 S(source) 中找到包含 T(target) 中全部字母的一个子串，顺序无所谓，但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.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。

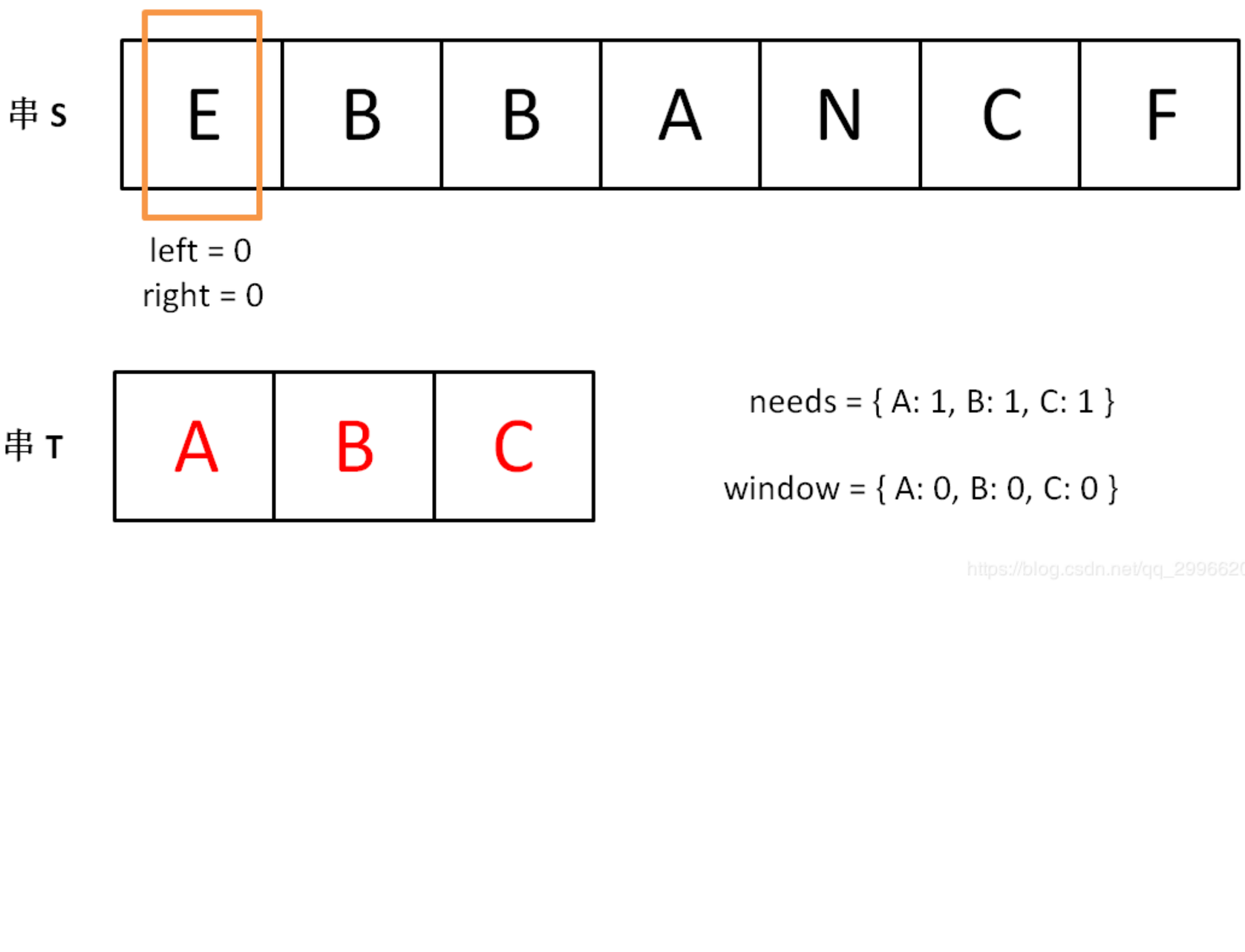
滑动窗口算法的思路是这样：

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。

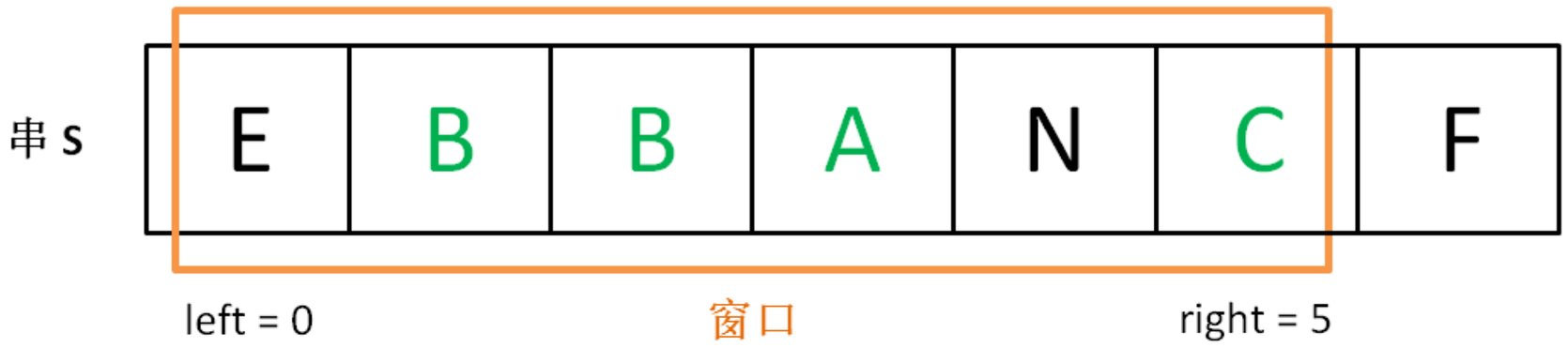
这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

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。

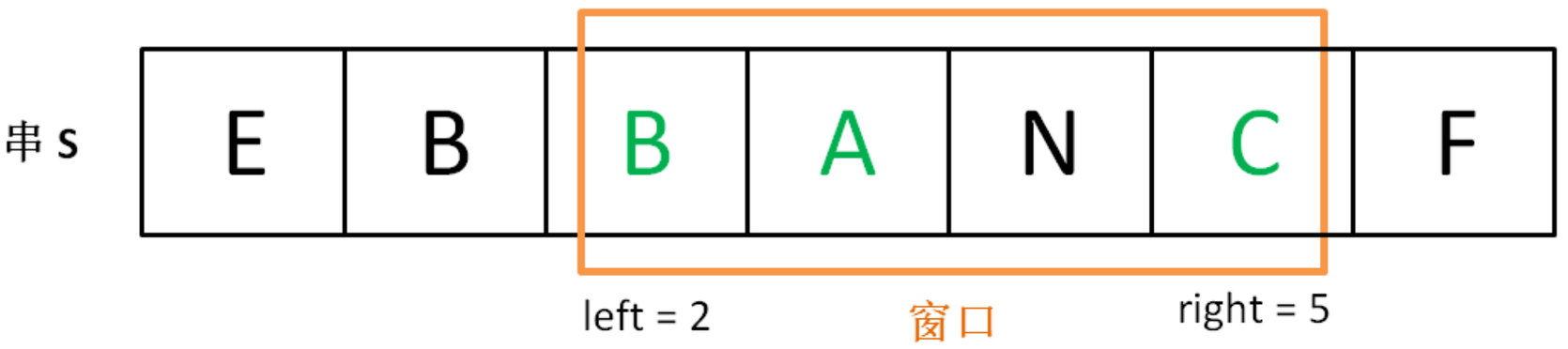
初始状态：



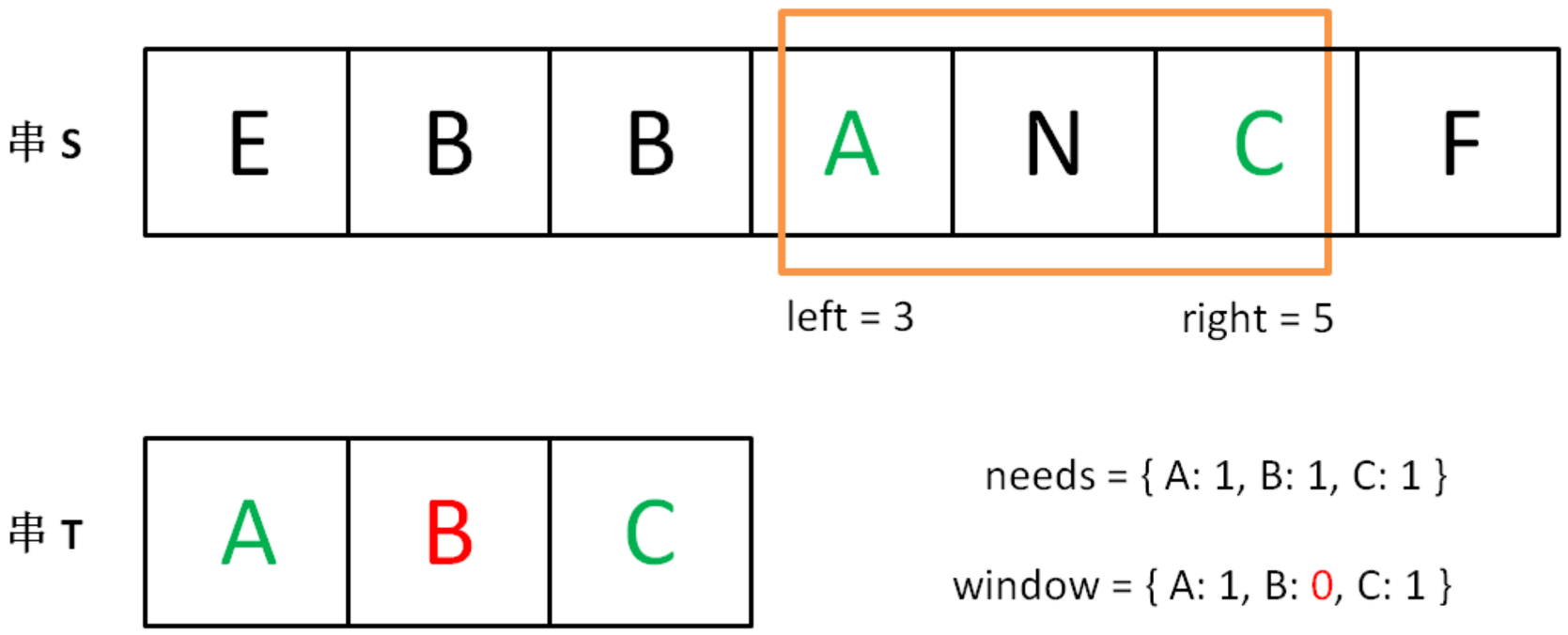
增加 right，直到窗口 [left, right] 包含了 T 中所有字符：



现在开始增加 left，缩小窗口 [left, right]。



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，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。至于如何具体到问题，如何得出此题的答案，都是编程问题，等会提供一套模板，理解一下就会了。

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;

while(right < s.size()) {
    window.add(s[right]);
    right++;
    // 如果符合要求，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 res
        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，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 res 的地方，不过这个问题太好解决了，直接看解法吧！

```
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，你是否能够理解这个算法的内在逻辑呢？你是否能清晰看出该算法的结构呢？

这个算法的时间复杂度是 O(M + N)O(M+N)，MM 和 NN 分别是字符串 SS 和 TT 的长度。因为我们先用 forfor 循环遍历了字符串 TT 来初始化 needsneeds，时间 O(N)O(N)，之后的两个 whilewhile 循环最多执行 2M2M 次，时间 O(M)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

题目连接

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 res 部分的代码稍加修改就成了这道题的解：

C++

```
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。如果本文对你有帮助，关注我的众公号 labuladong 看更多精彩算法文章~

三、无重复字符的最长子串

题目链接

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 res。

最后总结

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。

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。