

leetcode-master / problems / 0028.实现strStr.md

youngyangyang04 Update

History

3 contributors



819 lines (609 sloc) 26.6 KB

[PDF下载](#) [代码随想录](#) [刷题](#) [微信群](#) [B站](#) [代码随想录](#) [知识星球](#) [代码随想录](#)

欢迎大家[参与本项目](#)，贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！

在一个串中查找是否出现过另一个串，这是KMP的看家本领。

## 28. 实现 strStr()

<https://leetcode-cn.com/problems/implement-strstr/>

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

示例 1: 输入: haystack = "hello", needle = "ll" 输出: 2

示例 2: 输入: haystack = "aaaaa", needle = "bba" 输出: -1

说明: 当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。对于本题而言，**当 needle 是空字符串时我们应当返回 0**。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

### 思路

本题是KMP 经典题目。

以下文字如果看不进去，可以看我的B站视频：

```
#include<cstring>
```

```
strstr
```

```
const char * strstr ( const char * str1, const char * str2 );  
char * strstr (      char * str1, const char * str2 );
```

**Locate substring**

Returns a pointer to the first occurrence of str2 in str1, or a null pointer if str2 is not part of str1.

The matching process does not include the terminating null-characters, but it stops there.

- [帮你把KMP算法学个通透！B站（理论篇）](#)
- [帮你把KMP算法学个通透！（求next数组代码篇）](#)

KMP的经典思想就是**当出现字符串不匹配时，可以记录一部分之前已经匹配的文本内容，利用这些信息避免从头再去做匹配。**

本篇将以如下顺序来讲解KMP，

- 什么是KMP
- KMP有什么用
- 什么是前缀表
- 为什么一定要用前缀表
- 如何计算前缀表
- 前缀表与next数组
- 使用next数组来匹配
- 时间复杂度分析
- 构造next数组
- 使用next数组来做匹配
- 前缀表统一减一 C++代码实现
- 前缀表（不减一）C++实现
- 总结

读完本篇可以顺便把leetcode上28.实现strStr()题目做了。

## 什么是KMP

说到KMP，先说一下KMP这个名字是怎么来的，为什么叫做KMP呢。

是因为由这三位学者发明的：Knuth，Morris和Pratt，所以取了三位学者名字的首字母。所以叫做KMP

## KMP有什么用

**KMP主要应用在字符串匹配上。**

KMP的主要思想是**当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息避免从头再去做匹配了。**

所以如何记录已经匹配的文本内容，是KMP的重点，也是next数组肩负的重任。

其实KMP的代码不好理解，一些同学甚至直接把KMP代码的模板背下来。

没有彻底搞懂，懵懵懂懂就把代码背下来太容易忘了。

不仅面试的时候可能写不出来，如果面试官问：**next数组里的数字表示的是什么，为什么这么表示？**

估计大多数候选人都是懵逼的。

下面Carl就带大家把KMP的精髓，next数组弄清楚。

# 什么是前缀表

写过KMP的同学，一定都写过next数组，那么这个next数组究竟是个啥呢？

next数组就是一个前缀表（prefix table）。

前缀表有什么作用呢？

前缀表是用来回退的，它记录了模式串与主串(文本串)不匹配的时候，模式串应该从哪里开始重新匹配。

为了清楚的了解前缀表的来历，我们来举一个例子：

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

请记住文本串和模式串的作用，对于理解下文很重要，要不然容易看懵。所以说三遍：

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

如动画所示：



动画里，我特意把子串 aa 标记上了，这是有原因的，大家先注意一下，后面还会说道。

可以看出，文本串中第六个字符b 和 模式串的第六个字符f，不匹配了。如果暴力匹配，会发现不匹配，此时就要从头匹配了。

但如果使用前缀表，就不会从头匹配，而是从上次已经匹配的内容开始匹配，找到了模式串中第三个字符b继续开始匹配。

此时就要问了前缀表是如何记录的呢？

首先要知道前缀表的任务是当前位置匹配失败，找到之前已经匹配上的位置，在重新匹配，此也意味着在某个字符失配时，前缀表会告诉你下一步匹配中，模式串应该跳到哪个位置。  
next数组里的数字表示的是什么，为什么这么表示？

那么什么是前缀表：记录下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀。  
最长相等前后缀的长度

## 最长公共前后缀？

文章中字符串的前缀是指不包含最后一个字符的所有以第一个字符开头的连续子串。

后缀是指不包含第一个字符的所有以最后一个字符结尾的连续子串。

正确理解什么是前缀什么是后缀很重要!

那么网上清一色都说“~~kmp 最长公共前后缀~~”又是怎么回事呢?

我查了一遍 算法导论 和 算法4里KMP的章节, 都没有提到“~~最长公共前后缀~~”这个词, 也不知道从哪里来了, 我理解是用“**最长相等前后缀**”更准确一些。


因为前缀表要求的就是**相同前后缀的长度**。

而最长公共前后缀里面的“公共”, 更像是说前缀和后缀公共的长度。这其实并不是前缀表所需要的。

所以字符串a的最长相等前后缀为0。字符串aa的最长相等前后缀为1。**字符串aaa的最长相等前后缀为2。** 等等.....。

## 为什么一定要用前缀表

这就是前缀表那为啥就能告诉我们 上次匹配的位置, 并跳过去呢?

回顾一下, 刚刚匹配的过程在下标5的地方遇到不匹配, 模式串是指向f, 如图: 

然后就找到了下标2, 指向b, 继续匹配: 如图: 

以下这句话, 对于理解**为什么使用前缀表可以告诉我们匹配失败之后跳到哪里重新匹配**非常重要!

下标5之前这部分的字符串 (也就是**字符串aabaa**) 的最长相等的**前缀** 和 **后缀**字符串是 **子字符串aa**, 因为找到了最长相等的前缀和后缀, **匹配失败的位置是后缀子串的后面**, 那么我们**找到与其相同的前缀的后面从新匹配**就可以了。

所以前缀表具有告诉我们当前位置匹配失败, 跳到之前已经匹配过的地方的能力。

很多介绍KMP的文章或者视频并没有把为什么要用前缀表? 这个问题说清楚, 而是直接默认使用前缀表。

## 如何计算前缀表

接下来就要说一说怎么计算前缀表。

如图:



长度为前1个字符的子串 a, 最长相同前后缀的长度为0。(注意字符串的**前缀是指不包含最后一个字符的所有以第一个字符开头的连续子串**; **后缀是指不包含第一个字符的所有以最后一个字符结尾的连续子串。**)

 长度为前2个字符的子串 aa, 最长相同前后缀的长度为1。

 长度为前3个字符的子串 aab, 最长相同前后缀的长度为0。

下表:	0	1	2	3	4	5
模式串:	a	a	b	a	a	f



下表:	0	1	2	3	4	5
模式串:	a	a	b	a	a	f



下表:	0	1	2	3	4	5
模式串:	a	a	b	a	a	f

下表:	0	1	2	3	4	5
模式串:	a	a	b	a	a	f

下表:	0	1	2	3	4	5
模式串:	a	a	b	a	a	f

以此类推：长度为前4个字符的子串 aaba，最长相同前后缀的长度为1。长度为前5个字符的子串 aabaa，最长相同前后缀的长度为2。长度为前6个字符的子串 aabaaf，最长相同前后缀的长度为0。

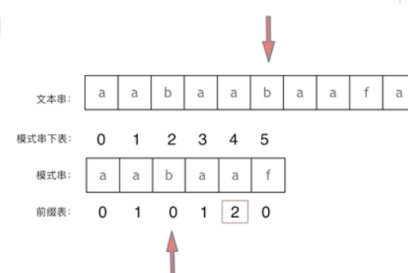
那么把求得的最长相同前后缀的长度就是对应前缀表的元素，如图：

下表：            0        1        2        3        4        5

模式串：

a	a	b	a	a	f
---	---	---	---	---	---

前缀表：        0        1        0        1        2        0



可以看出模式串与前缀表对应位置的数字表示的就是：**下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀。**

再来看一下如何利用 前缀表找到 当字符不匹配的时候应该指针应该移动的位置。如动画所示：



找到的不匹配的位置，那么此时我们要看它的前一个字符的前缀表的数值是多少。

为什么要前一个字符的前缀表的数值呢，**因为要找前面字符串的最长相同的前缀和后缀。**

所以要看前一位的 前缀表的数值。

前一个字符的前缀表的数值是2，所有把下标移动到下标2的位置继续匹配。可以再反复看一下上面的动画。

最后就在文本串中找到了和模式串匹配的子串了。

## 前缀表与next数组

实现

很多KMP算法的~~时间~~都是使用next数组来做回退操作，那么next数组与前缀表有什么关系呢？

next数组就可以是前缀表，但是很多实现都是把前缀表统一减一（右移一位，初始位置为-1）之后作为next数组。

为什么这么做呢，其实也是很多文章视频没有解释清楚的地方。

其实**这并不涉及到KMP的原理，而是具体实现，next数组即可以就是前缀表，也可以是前缀表统一减一（右移一位，初始位置为-1）。**

后面我会提供两种不同的实现代码，大家就明白了。

## 使用next数组来匹配

以下我们以前缀表统一减一之后的next数组来做演示。

有了next数组，就可以根据next数组来匹配文本串s，和模式串t了。

注意next数组是新前缀表（旧前缀表统一减一了）。

匹配过程动画如下：



## 时间复杂度分析

其中n为文本串长度，m为模式串长度，因为在匹配的过程中，根据前缀表不断调整匹配的位置，可以看出匹配的过程是 $O(n)$ ，之前还要单独生成next数组，时间复杂度是 $O(m)$ 。所以整个KMP算法的时间复杂度是 $O(n+m)$ 的。

暴力的解法显而易见是 $O(n * m)$ ，所以KMP在字符串匹配中极大的提高了搜索的效率。

为了和力扣题目28.实现strStr保持一致，方便大家理解，以下文章统称haystack为文本串，needle为模式串。

都知道使用KMP算法，一定要构造next数组。

## 构造next数组

我们定义一个函数getNext来构建next数组，函数参数为指向next数组的指针，和一个字符串。代码如下：

```
void getNext(int* next, const string& s)
```

构造next数组其实就是计算模式串s，前缀表的过程。主要有如下三步：

1. 初始化
2. 处理前后缀不相同的情况
3. 处理前后缀相同的情况

next

接下来我们详解详解一下。

1. 初始化：  
j指向前缀末尾位置，i指向后缀末尾位置

定义两个指针i和j，j指向前缀起始位置，i指向后缀起始位置。

然后还要对next数组进行初始化赋值，如下：

```
int j = -1;
next[0] = j;
```

j为什么要初始化为-1呢，因为之前说过 前缀表要统一减一的操作仅仅是其中的一种实现，我们这里选择j初始化为-1，下文我还会给出j不初始化为-1的实现代码。

next[i] 表示 i（包括i）之前最长相等的前后缀长度（其实就是j）

所以初始化next[0] = j。

## 2. 处理前后缀不相同的情况

因为j初始化为-1，那么i就从1开始，进行s[i] 与 s[j+1]的比较。

所以遍历模式串s的循环下标i 要从 1开始，代码如下：

```
for(int i = 1; i < s.size(); i++) {
```

如果 s[i] 与 s[j+1]不相同，也就是遇到 前后缀末尾不相同的情况，就要向前回退。

怎么回退呢？

next[j]就是记录着j（包括j）之前的子串的相同前后缀的长度。

那么 s[i] 与 s[j+1] 不相同，就要找 j+1前一个元素在next数组里的值（就是next[j]）。

所以，处理前后缀不相同的情况代码如下：

```
while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
    j = next[j]; // 向前回退
}
```

## 3. 处理前后缀相同的情况

如果s[i] 与 s[j + 1] 相同，那么就同时向后移动i 和j 说明找到了相同的前后缀，同时还要将j（前缀的长度）赋给next[i], 因为next[i]要记录相同前后缀的长度。

代码如下：

```
if (s[i] == s[j + 1]) { // 找到相同的前后缀
    j++;
}
next[i] = j;
```

最后整体构建next数组的函数代码如下：

```
void getNext(int* next, const string& s){
    int j = -1;
    next[0] = j;
    for(int i = 1; i < s.size(); i++) { // 注意i从1开始
```



```

while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
    j = next[j]; // 向前回退
}
if (s[i] == s[j + 1]) { // 找到相同的前后缀
    j++;
}
next[i] = j; // 将j（前缀的长度）赋给next[i]
}
}

```

代码构造next数组的逻辑流程动画如下：

next[i]:

模式串:

a	a	b	a	a	f
---	---	---	---	---	---

下表i:

0 1 2 3 4 5



得到了next数组之后，就要用这个来做匹配了。

## 使用next数组来做匹配

在文本串s里 找是否出现过模式串t。

定义两个下标j 指向模式串起始位置，i指向文本串起始位置。

那么j初始值依然为-1，为什么呢？依然因为next数组里记录的起始位置为-1。

i就从0开始，遍历文本串，代码如下：

```
for (int i = 0; i < s.size(); i++)
```

接下来就是 s[i] 与 t[j + 1]（因为j从-1开始的） 进行比较。

如果 s[i] 与 t[j + 1] 不相同，j就要从next数组里寻找下一个匹配的位置。

代码如下：

```

while(j >= 0 && s[i] != t[j + 1]) {
    j = next[j];
}

```



```
}
```

如果  $s[i]$  与  $t[j + 1]$  相同，那么  $i$  和  $j$  同时向后移动，代码如下：

```
if (s[i] == t[j + 1]) {  
    j++; // i的增加在for循环里  
}
```

如何判断在文本串  $s$  里出现了模式串  $t$  呢，如果  $j$  指向了模式串  $t$  的末尾，那么就说明模式串  $t$  完全匹配文本串  $s$  里的某个子串了。

本题要在文本串字符串中找出模式串出现的第一个位置 (从0开始)，所以返回当前在文本串匹配模式串的位置  $i$  减去 模式串的长度，就是文本串字符串中出现模式串的第一个位置。

代码如下：

```
if (j == (t.size() - 1) ) {  
    return (i - t.size() + 1);  
}
```

那么使用 `next` 数组，用模式串匹配文本串的整体代码如下：

```
int j = -1; // 因为next数组里记录的起始位置为-1  
for (int i = 0; i < s.size(); i++) { // 注意i就从0开始  
    while(j >= 0 && s[i] != t[j + 1]) { // 不匹配  
        j = next[j]; // j 寻找之前匹配的位置  
    }  
    if (s[i] == t[j + 1]) { // 匹配，j和i同时向后移动  
        j++; // i的增加在for循环里  
    }  
    if (j == (t.size() - 1) ) { // 文本串s里出现了模式串t  
        return (i - t.size() + 1);  
    }  
}
```

此时所有逻辑的代码都已经写出来了，力扣 28.实现strStr 题目的整体代码如下：

## 前缀表统一减一 C++ 代码实现

```
class Solution {  
public:  
    void getNext(int* next, const string& s) {  
        int j = -1;  
        next[0] = j;  
        for(int i = 1; i < s.size(); i++) { // 注意i从1开始  
            while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了  
                j = next[j]; // 向前回退  
            }  
            if (s[i] == s[j + 1]) { // 找到相同的前后缀
```

```

        j++;
    }
    next[i] = j; // 将j（前缀的长度）赋给next[i]
}
}
int strStr(string haystack, string needle) {
    if (needle.size() == 0) {
        return 0;
    }
    int next[needle.size()];
    getNext(next, needle);
    int j = -1; // 因为next数组里记录的起始位置为-1
    for (int i = 0; i < haystack.size(); i++) { // 注意i就从0开始
        while(j >= 0 && haystack[i] != needle[j + 1]) { // 不匹配
            j = next[j]; // j 寻找之前匹配的位置
        }
        if (haystack[i] == needle[j + 1]) { // 匹配，j和i同时向后移动
            j++; // i的增加在for循环里
        }
        if (j == (needle.size() - 1)) { // 文本串s里出现了模式串t
            return (i - needle.size() + 1);
        }
    }
    return -1;
}
};

```

## 前缀表（不减一）C++实现

那么前缀表就不减一了，也不右移的，到底行不行呢？

行！

我之前说过，这仅仅是KMP算法实现上的问题，如果就直接使用前缀表可以换一种回退方式，找  $j = \text{next}[j - 1]$  来进行回退。

主要就是  $j = \text{next}[x]$  这一步最为关键！

我给出的getNext的实现为：（前缀表统一减一）

```

void getNext(int* next, const string& s) {
    int j = -1;
    next[0] = j;
    for(int i = 1; i < s.size(); i++) { // 注意i从1开始
        while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
            j = next[j]; // 向前回退
        }
        if (s[i] == s[j + 1]) { // 找到相同的前后缀
            j++;
        }
        next[i] = j; // 将j（前缀的长度）赋给next[i]
    }
}

```

此时如果输入的模式串为aabaaf，对应的next为-1 0 -1 0 1 -1。

这里j和next[0]初始化为-1，整个next数组是以 前缀表减一之后的效果来构建的。

那么前缀表不减一来构建next数组，代码如下：

```
void getNext(int* next, const string& s) {
    int j = 0;
    next[0] = 0;
    for(int i = 1; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) { // j要保证大于0，因为下面有取j-1作为数组下标的操作
            j = next[j - 1]; // 注意这里，是要找前一位的对应的回退位置了
        }
        if (s[i] == s[j]) {
            j++;
        }
        next[i] = j;
    }
}
```

此时如果输入的模式串为aabaaf，对应的next为 0 1 0 1 2 0，（其实这就是前缀表的数值了）。

那么用这样的next数组也可以用来做匹配，代码要有所改动。

实现代码如下：

next数组就是前缀表，不用减1，遇到冲突就到前一位找就可以

```
class Solution {
public:
    void getNext(int* next, const string& s) {
        int j = 0;
        next[0] = 0;
        for(int i = 1; i < s.size(); i++) {
            while (j > 0 && s[i] != s[j]) {
                j = next[j - 1];
            }
            if (s[i] == s[j]) {
                j++;
            }
            next[i] = j;
        }
    }
    int strStr(string haystack, string needle) {
        if (needle.size() == 0) {
            return 0;
        }
        int next[needle.size()];
        getNext(next, needle);
        int j = 0;
        for (int i = 0; i < haystack.size(); i++) {
            while(j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if (haystack[i] == needle[j]) {
                j++;
            }
        }
    }
};
```

```

    }
    if (j == needle.size() ) {
        return (i - needle.size() + 1);
    }
}
return -1;
}
};

```

## 总结

我们介绍了什么是KMP，KMP可以解决什么问题，然后分析KMP算法里的next数组，知道了next数组就是前缀表，再分析为什么要是前缀表而不是什么其他表。

接着从给出的模式串中，我们一步一步的推导出了前缀表，得出前缀表无论是统一减一还是不减一得到的next数组仅仅是kmp的实现方式的不同。

其中还分析了KMP算法的时间复杂度，并且和暴力方法做了对比。

然后先用前缀表统一减一得到的next数组，求得文本串s里是否出现过模式串t，并给出了具体分析代码。

又给出了直接用前缀表作为next数组，来做匹配的实现代码。

可以说把KMP的每一个细微的细节都扣了出来，毫无遮掩的展示给大家了！

## 其他语言版本

Java:

```

class Solution {
    /**
     * 基于窗口滑动的算法
     * <p>
     * 时间复杂度: O(m*n)
     * 空间复杂度: O(1)
     * 注: n为haystack的长度, m为needle的长度
     */
    public int strStr(String haystack, String needle) {
        int m = needle.length();
        // 当 needle 是空字符串时我们应当返回 0
        if (m == 0) {
            return 0;
        }
        int n = haystack.length();
        if (n < m) {
            return -1;
        }
        int i = 0;
        int j = 0;
        while (i < n - m + 1) {
            // 找到首字母相等

```

```

        while (i < n && haystack.charAt(i) != needle.charAt(j)) {
            i++;
        }
        if (i == n) {// 没有首字母相等的
            return -1;
        }
        // 遍历后续字符，判断是否相等
        i++;
        j++;
        while (i < n && j < m && haystack.charAt(i) == needle.charAt(j)) {
            i++;
            j++;
        }
        if (j == m) {// 找到
            return i - j;
        } else {// 未找到
            i -= j - 1;
            j = 0;
        }
    }
    return -1;
}
}

```

// 方法一

```

class Solution {
    public void getNext(int[] next, String s){
        int j = -1;
        next[0] = j;
        for (int i = 1; i<s.length(); i++){
            while(j>=0 && s.charAt(i) != s.charAt(j+1)){
                j=next[j];
            }

            if(s.charAt(i)==s.charAt(j+1)){
                j++;
            }
            next[i] = j;
        }
    }
    public int strStr(String haystack, String needle) {
        if(needle.length()==0){
            return 0;
        }

        int[] next = new int[needle.length()];
        getNext(next, needle);
        int j = -1;
        for(int i = 0; i<haystack.length();i++){
            while(j>=0 && haystack.charAt(i) != needle.charAt(j+1)){
                j = next[j];
            }
            if(haystack.charAt(i)==needle.charAt(j+1)){
                j++;
            }
            if(j==needle.length()-1){
                return (i-needle.length()+1);
            }
        }
    }
}

```

```

    }
    }

    return -1;
}
}

```

Python:

```

// 方法一
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        a=len(needle)
        b=len(haystack)
        if a==0:
            return 0
        next=self.getNext(a,needle)
        p=-1
        for j in range(b):
            while p>=0 and needle[p+1]!=haystack[j]:
                p=next[p]
            if needle[p+1]==haystack[j]:
                p+=1
            if p==a-1:
                return j-a+1
        return -1

    def getNext(self,a,needle):
        next=['' for i in range(a)]
        k=-1
        next[0]=k
        for i in range(1,len(needle)):
            while (k>-1 and needle[k+1]!=needle[i]):
                k=next[k]
            if needle[k+1]==needle[i]:
                k+=1
            next[i]=k
        return next

```

```

// 方法二
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        a=len(needle)
        b=len(haystack)
        if a==0:
            return 0
        i=j=0
        next=self.getNext(a,needle)
        while(i<b and j<a):
            if j== -1 or needle[j]==haystack[i]:
                i+=1
                j+=1
            else:
                j=next[j]
        if j==a:

```

```

        return i-j
    else:
        return -1

def getNext(self,a,needle):
    next=['' for i in range(a)]
    j,k=0,-1
    next[0]=k
    while(j<a-1):
        if k== -1 or needle[k]==needle[j]:
            k+=1
            j+=1
            next[j]=k
        else:
            k=next[k]
    return next

```

Go:

```

// 方法一:前缀表使用减1实现

// getNext 构造前缀表next
// params:
//          next 前缀表数组
//          s 模式串
func getNext(next []int, s string) {
    j := -1 // j表示 最长相等前后缀长度
    next[0] = j

    for i := 1; i < len(s); i++ {
        for j >= 0 && s[i] != s[j+1] {
            j = next[j] // 回退前一位
        }
        if s[i] == s[j+1] {
            j++
        }
        next[i] = j // next[i]是i（包括i）之前的最长相等前后缀长度
    }
}

func strStr(haystack string, needle string) int {
    if len(needle) == 0 {
        return 0
    }
    next := make([]int, len(needle))
    getNext(next, needle)
    j := -1 // 模式串的起始位置 next为-1 因此也为-1
    for i := 0; i < len(haystack); i++ {
        for j >= 0 && haystack[i] != needle[j+1] {
            j = next[j] // 寻找下一个匹配点
        }
        if haystack[i] == needle[j+1] {
            j++
        }
        if j == len(needle)-1 { // j指向了模式串的末尾
            return i - len(needle) + 1
        }
    }
}

```



```
        return -1
    }
}
```

// 方法二：前缀表无减一或者右移

```
// getNext 构造前缀表next
// params:
//         next 前缀表数组
//         s 模式串
func getNext(next []int, s string) {
    j := 0
    next[0] = j
    for i := 1; i < len(s); i++ {
        for j > 0 && s[i] != s[j] {
            j = next[j-1]
        }
        if s[i] == s[j] {
            j++
        }
        next[i] = j
    }
}

func strStr(haystack string, needle string) int {
    n := len(needle)
    if n == 0 {
        return 0
    }
    j := 0
    next := make([]int, n)
    getNext(next, needle)
    for i := 0; i < len(haystack); i++ {
        for j > 0 && haystack[i] != needle[j] {
            j = next[j-1] // 回退到j的前一位
        }
        if haystack[i] == needle[j] {
            j++
        }
        if j == n {
            return i - n + 1
        }
    }
    return -1
}
```

- 
- 作者微信：[程序员Carl](#)
  - B站视频：[代码随想录](#)
  - 知识星球：[代码随想录](#)