

4.5.2 KMP算法的原理



那么，KMP算法是如何减少无谓的字符比较的呢？

KMP算法提高效率的关键，是对已匹配前缀的有效利用。

下面我们来仔细讲一下KMP算法的思路。



给定主串和模式串如下：

主串： GTGTGAGCTGGTGTGTCFAA

模式串： GTGTCF

首先，我们把主串和模式串的首位对齐，从左到右逐个对字符进行比较。

第1轮，模式串和主串的第一个等长子串比较，发现前5个字符都是匹配的，第6个字符不匹配：

主串： GTGTGAGCTGGTGTGTCFAA

模式串： GTGTCF

对于这种不匹配的字符（主串中的字符A），我们称之为“坏字符”；而前面已经匹配上的部分“GTGTG”，我们称之为“已匹配前缀”。

我们可以发现，在已匹配前缀“GTGTG”当中，后三个字符“GTG”和前三个字符“GTG”是相同的。

在下一轮比较时，只有把这两个相同的片段对齐，才有可能出现匹配。这两个字符串片段，分别叫作最长可匹配后缀子串和最长可匹配前缀子串：

最长可匹配后缀子串

主串： G T **G T G** A G C T G G T G T G T G C F A A

模式串： **G T G** T G C F

最长可匹配前缀子串

第2轮，我们直接把模式串向后移动两位，让两个“GTG”对齐，继续从刚才主串的坏字符A开始进行比较：

主串： G T **G T G** A G C T G G T G T G T G C F A A

模式串： **G T G** T G C F

显然，主串的字符A仍然是坏字符，这时候的已匹配前缀缩短成了GTG（在上一轮，已匹配前缀是GTGTG）：

主串： G T G T G **A** G C T G G T G T G T G C F A A

模式串： G T G **T** G C F

按照第1轮的思路，我们来重新确定最长可匹配后缀子串和最长可匹配前缀子串：

最长可匹配后缀子串

主串： G T G T **G** A G C T G G T G T G T G C F A A

模式串： **G** T G T G C F

最长可匹配前缀子串

第3轮，我们再次把模式串向后移动两位，让两个“G”对齐，继续从刚才主串的坏字符A开始进行比较：

主串： G T G T **G** A G C T G G T G T G T G C F A A

模式串： **G** T G T G C F

以上就是KMP算法的整体思路：在已匹配的前缀当中寻找到最长可匹配后缀子串和最长可匹配前缀子串，在下一轮直接把两者对齐，从而实现模式串的快速移动。



那么，我们如何找到一个字符串前缀的“最长可匹配后缀子串”和“最长可匹配前缀子串”呢？难道每一轮都要重新遍历吗？

这个问题提得很好。要找到这两个子串，我们没有必要每次都去遍历，而是事先缓存到一个集合当中，用的时候再去集合里面取。



这个集合被称为next数组，如何生成next数组是KMP算法的最大难点。

接下来的内容比较烧脑，请坐稳扶好！

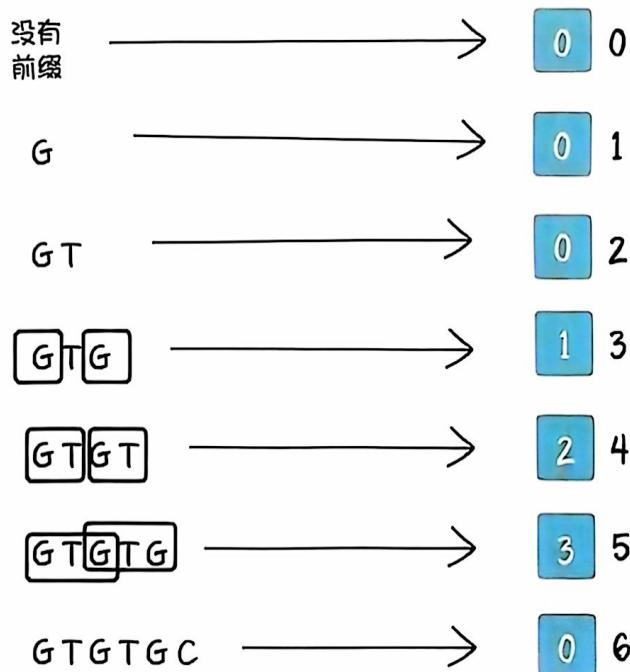


next数组到底是个什么“鬼”呢？这是一个一维整型数组，数组的下标代表了“已匹配前缀的下一个位置”，元素的值则是“最长可匹配前缀子串的下一个位置”。

或许这样的描述有些晦涩，我们来看一下图：

字符串前缀：

next数组：



当模式串的第一个字符就和主串不匹配时，并不存在已匹配前缀子串，更不存在最长可匹配前缀子串。这种情况对应的next数组下标是0，`next[0]`的元素值也是0。

如果已匹配前缀是G、GT、GTGTGC，并不存在最长可匹配前缀子串，所以对应的next数组元素值（`next[1]`, `next[2]`, `next[6]`）同样是0。

GTG的最长可匹配前缀是G，对应数组中的`next[3]`，元素值是1。

以此类推，GTGT 对应 `next[4]`，元素值是2；GTGTG 对应 `next[5]`，元素值是3。

有了next数组，我们就可以通过已匹配前缀的下一个位置（坏字符位置），快速寻找最长可匹配前缀的下一个位置，然后把这两个位置对齐。

比如下面的场景，我们通过坏字符下标5，可以找到`next[5]=3`，即最长可匹配前缀的下一个位置是3

已匹配前缀的
下一个位置: 5

主串: G T G T G A G C T G G T G T G C F A A

模式串: G T G T G C F

最长可匹配前缀的
下一个位置: 3

next数组:

0	0	0	1	2	3	0
---	---	---	---	---	---	---

0 1 2 3 4 5 6

说完了next数组是什么，接下来我们再来思考一下，如何预生成这个next数组呢？

由于已匹配前缀在主串和模式串当中是相同的，所以我们仅仅依据模式串，就足以生成next数组。

最简单的方法是从最长的前缀子串开始，把每一种可能情况都做一次比较。

假设模式串的长度是 n ，生成next数组所需的最大总比较次数是 $1+2+3+4+\dots+n-1$ 次。

显然，这种方法的效率非常低，如何进行优化呢？

我们可以采用类似“动态规划”的方法。首先next[0]和next[1]的值肯定是0，因为这时候不存在前缀子串；从next[2]开始，next数组的每一个元素都可以由上一个元素直接或间接推导而来。

已知next[i]的值，如何推导出next[i+1]？我们看一看下面的例子：

j i

模式串: A B A B C A B A B A

next [i] = j = 3

如上图所示，在该模式串上，我们设置两个变量 j 和 i ，其中 j 表示“已匹配前缀的下一个位置”，也就是待填充的数组下标， i 表示“最长可匹配前缀子串的下一个位置”，也就是next数组对应的元素值。

在上图中， i 所在的位置之前，最长可匹配前缀子串是“ABA”，因此 $next[i] = 3$ 。

那么， $next[i+1]$ 和 $next[i]$ 的关系是什么呢？我们先让 i 指向下一个字符：

j i

模式串: A B A B C A B A B A

此时，模式串中的字符 $pattern[i+1]$ 和 $pattern[i]$ 相同，都是字母“B”，因此最长可匹配前缀子串的长度(i 的值)也增加了1。

模式串： A B A B C A B A B A

$$\text{next}[i] = j = 4$$

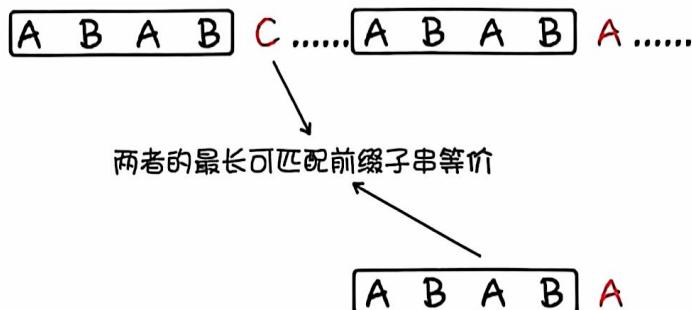
由此我们可以得出，当下一个字符相同的时候， $\text{next}[i+1] = \text{next}[i] + 1$ 。

接下来，我们继续让*i*指向下一个字符：

模式串： A B A B C A B A B A

此时，模式串中的字符 $\text{pattern}[i-1]$ 和 $\text{pattern}[i]$ 不相同，之前的最长可匹配前缀子串也就变得无效了。

这时候该怎么办呢？我们有一个退而求其次的选择。由于模式串中有两段曾经的最长可匹配前缀子串（ABAB）是相同的，所以下面这两个字符串的最长可匹配前缀子串是等价的：



这样一来，我们就把问题转化成了在字符串“ABABA”中寻找最长可匹配前缀子串。要想求得“ABABA”的最长可匹配前缀子串，需要由“ABAB”的最长可匹配前缀子串（也就是 $\text{next}[i]$ ）推导出，这就需要我们把指针*j*进行回溯，回溯到 $j = \text{next}[i]$ 时候的状态：

模式串： A B A B C A B A B A

↓ 回溯

模式串： A B A B C A B [A B] A

此时，由于 $\text{pattern}[i-1]$ 和 $\text{pattern}[i]$ 相同，都是字母“A”，因此最长可匹配前缀子串的长度（*j*的值）也增加了1：

模式串： A B A B C A B [A B] A

$$\text{next}[i] = j = 3$$

接下来，我们继续重复上面的逻辑，当 $\text{pattern}[i-1]$ 和 $\text{pattern}[j]$ 相同时， $\text{next}[i+1] = \text{next}[i] + 1$ ；当 $\text{pattern}[i-1]$ 和 $\text{pattern}[j]$ 不相同时，让 j 回溯到 $\text{next}[j]$ ，直到 $j=0$ 无法进一步回溯为止。

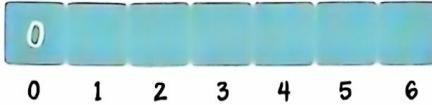
以上就是 next 数组的推导过程。

 我的天呐， next 数组的推导过程也太烧脑了……

一次看不明白很正常，这一段可以反复多看几遍。
为了加深理解，我们来演示一个完整的 next 数组填充过程：



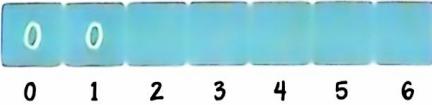
模式串： G T G T G C F
 i
 j

next数组：

0 1 2 3 4 5 6

我们回到上图的这个模式串。当已匹配前缀不存在的时候，最长可匹配前缀子串当然也不存在，所以 $i=0$, $j=0$ ，此时 $\text{next}[0] = 0$ 。

接下来，我们让已匹配前缀子串的长度(i)加1：

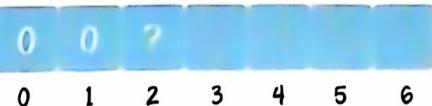
模式串： G T G T G C F
 j i

next数组：

0 1 2 3 4 5 6

此时的已匹配前缀是G，由于只有一个字符，同样不存在最长可匹配前缀子串，所以 $i=1$, $j=0$ ， $\text{next}[1] = 0$ 。

接下来，我们让已匹配前缀子串的长度继续加1：

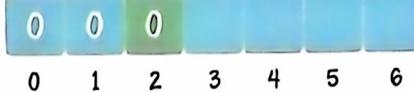
模式串： G T G T G C F
 j i

next数组：

0 1 2 3 4 5 6

此时的已匹配前缀是GT，我们需要开始做判断了：由于模式串当中 $\text{pattern}[j] \neq \text{pattern}[i-1]$ ，即 $G \neq T$ ，最长可匹配前缀子串仍然不存在。（变量 i 原本就是0，无法回溯。）

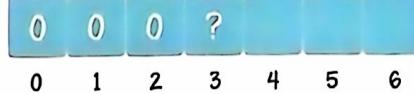
所以当 $i=2$ 时， j 仍然是0， $\text{next}[2] = 0$ 。

j i
模式串: G T G T G C F

next数组: 
0 1 2 3 4 5 6

接下来，我们让已匹配前缀子串的长度继续加1：

j i
模式串: G T G T G C F

next数组: 
0 1 2 3 4 5 6

此时的已匹配前缀是GTG，由于模式串当中 $\text{pattern}[j] = \text{pattern}[i-1]$ ，即G=G，最长可匹配前缀子串出现了，是G。

所以当 $i=3$ 时， $j=1$ ， $\text{next}[3] = \text{next}[2]+1 = 1$ 。

j i
模式串: G T G T G C F

next数组: 
0 1 2 3 4 5 6

接下来，我们让已匹配前缀子串的长度继续加1：

j i
模式串: G T G T G C F

next数组: 
0 1 2 3 4 5 6

此时的已匹配前缀是GTGT，由于模式串当中 $\text{pattern}[j] = \text{pattern}[i-1]$ ，即T=T，最长可匹配前缀子串又增加了一位，是GT。

所以当 $i=4$ 时， $j=2$ ， $\text{next}[4] = \text{next}[3]+1 = 2$ 。

j i
模式串: G T G T G C F

next数组: 
0 1 2 3 4 5 6

接下来，我们让已匹配前缀子串的长度继续加1：

j i
模式串： G T G T G C F

next数组：

0	0	0	1	2	?	
0	1	2	3	4	5	6

此时的已匹配前缀是GTGTG，由于模式串当中 $\text{pattern}[j] = \text{pattern}[i-1]$ ，即 $G=G$ ，最长可匹配前缀子串又增加了一位，是GTG。

所以当 $i=5$ 时， $j=3$ ， $\text{next}[5] = \text{next}[4]+1 = 3$ 。

j i
模式串： G T G T G C F

next数组：

0	0	0	1	2	3	?
0	1	2	3	4	5	6

接下来，我们让已匹配前缀子串的长度继续加1：

j i
模式串： G T G T G C F

next数组：

0	0	0	1	2	3	?
0	1	2	3	4	5	6

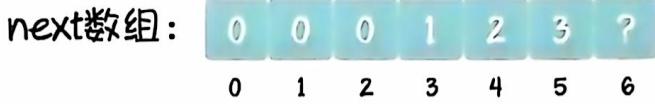
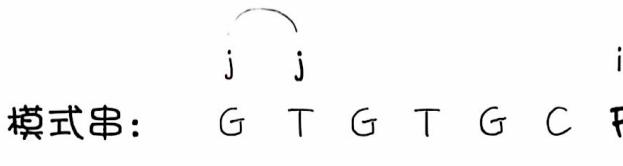
此时的已匹配前缀是GTGTGC，这时候需要注意了，模式串当中 $\text{pattern}[j] \neq \text{pattern}[i-1]$ ，即 $T \neq C$ ，我们需要把变量 j 回溯到 $\text{next}[j]$ ，也就是 $j=1$ 的局面（ i 值不变）：

j j i
模式串： G T G T G C F

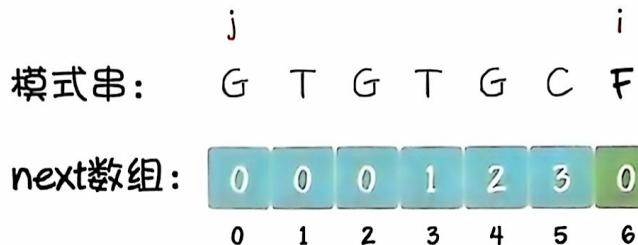
next数组：

0	0	0	1	2	3	?
0	1	2	3	4	5	6

回溯后，情况仍然是 $\text{pattern}[j] \neq \text{pattern}[i-1]$ ，即 $T \neq C$ 。我们继续把变量 j 回溯到 $\text{next}[j]$ ，也就是 $j=0$ 的局面：

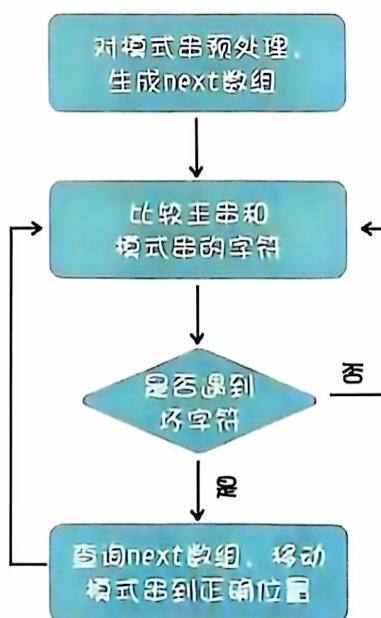


回溯后，情况仍然是 $\text{pattern}[j] \neq \text{pattern}[i-1]$ ，即 $G \neq C$ 。j已经不能再次回溯了，所以我们得出结论： $i=6$ 时， $j=0$ ， $\text{next}[6]=0$ 。



就这样，完整的next数组就生成了。

最后，让我们梳理一下KMP算法的全过程：



那么，KMP算法的代码如何实现呢？

代码已经写好了，我们一起看看吧！



```

// KMP算法主体逻辑。str是主串，pattern是模式串
public static int kmp(String str, String pattern) {
    //预处理，生成next数组
  
```

```

        int[] next = getNexts(pattern);
        int j = 0;
        //主循环，遍历主串字符
        for (int i = 0; i < str.length(); i++) {
            while ((j > 0) && (str.charAt(i) != pattern.charAt(j))) {
                //遇到坏字符时，查询next数组并改变模式串的起点
                j = next[j];
            }
            if (str.charAt(i) == pattern.charAt(j)) {
                j++;
            }
            if (j == pattern.length()) {
                //匹配成功，返回下标
                return i - pattern.length() + 1;
            }
        }
        return -1;
    }

    // 生成next数组
    private static int[] getNexts(String pattern) {
        int[] next = new int[pattern.length()];
        int j = 0;

        for (int i = 2; i < pattern.length(); i++) {
            while ((j != 0) && (pattern.charAt(j) != pattern.
charAt(i - 1))) {
                //从next[i+1]的求解回溯到 next[j]
                j = next[j];
            }
            if (pattern.charAt(j) == pattern.charAt(i - 1)) {
                j++;
            }
            next[i] = j;
        }
        return next;
    }

    public static void main(String[] args) {
        String str = "ATGTGAGCTGGTGTGTGCFAA";
        String pattern = "GTGTGCF";
        int index = kmp(str, pattern);
        System.out.println("首次出現位置: " + index);
    }
}

```

小灰，你来说一说，KMP算法的时间复杂度和空间复杂度分别是多少？



让我想想啊……

我知道啦！首先空间复杂度很好计算，KMP算法开辟的额外空间只有next数组，假设模式串长度是 n ，那么算法的空间复杂度就是 $O(n)$ 。

至于时间复杂度，KMP算法包括两部分，第一部分生成next数组，时间复杂度可以估算为 $O(n)$ ，第二部分的主循环是对主串的遍历，时间复杂度可以估算为 $O(m)$ 。

因此，KMP算法的整体时间复杂度是 $O(m+n)$ ，其中 n 是模式串的长度， m 是主串长度。

分析得很好，关于KMP算法我们就介绍到这里。
下一节，我们会对查找算法做一个整体的回顾总结。



4.6 小结

- 在有序数组中查找元素，可以使用二分查找算法，查找时间复杂度是 $O(\log n)$ 。
- 在有序链表中查找元素，可以把链表改造成跳表，查找时间复杂度是 $O(\log n)$ 。
- BF算法是朴素的字符串匹配算法，效率较低，时间复杂度是 $O(m \times n)$ 。
- RK算法利用hash进行字符串匹配，效率较高但不稳定，平均时间复杂度是 $O(m+n)$ 。
- KMP算法减少了字符串匹配过程中的无谓比较，效率较高且稳定，时间复杂度是 $O(m+n)$ 。