

## 43 | 串的KMP模式匹配算法之改进：通过优化代码解决多次重复比较问题

2023-05-22 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

上节课介绍的 KMP 模式匹配算法是通过 next 数组参与计算来达到加速匹配的目的。但是，在 next 数组中，因为没考虑当前字符的位置情况，只考虑了字符不匹配时单纯的指针移动问题（point1 和 point2 值的改变），这很可能导致移动后将进行比较的字符仍和上一次不匹配的字符相同导致产生多次重复的无效比较。

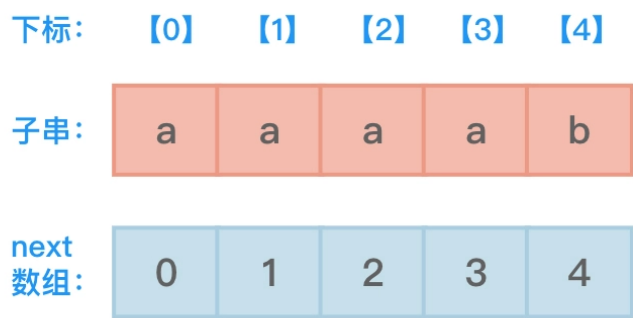
shikey.com转载分享

那这个问题要怎么解决呢？别着急，我们一步一步分析。

### next 数组使用中暴露的问题

下面我会用一个相对简单点的主串和子串，通过图示说明多次重复无效比较这件事。

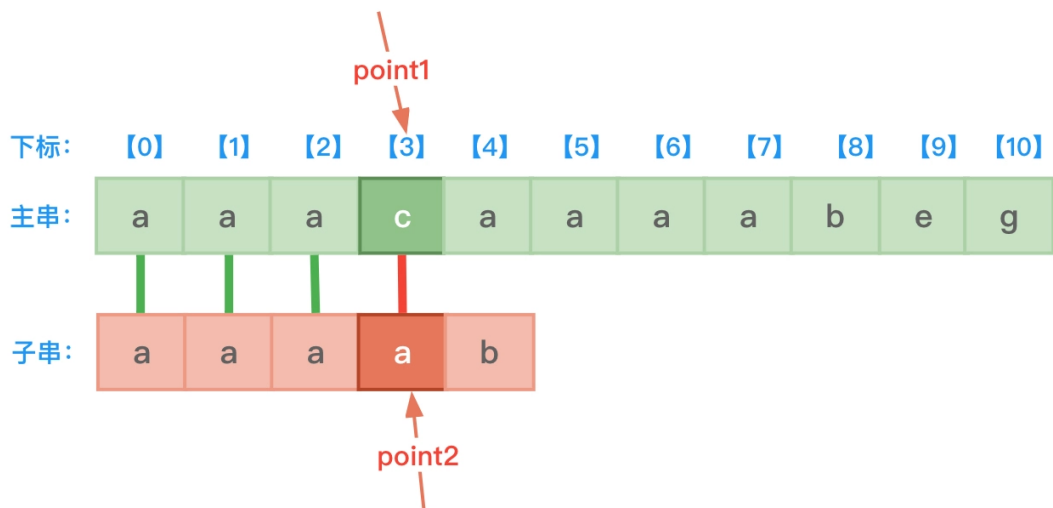
假设主串 S 为 “aaacaaaabeg”，子串 T 为 “aaaab”，可以求得子串的 next 数组为如图 1 所示：



极客时间

图1 子串 “aaaab” 对应的next数组

主串与子串开始比较。当比较到下标为 3 的位置时，子串中的字符是 a，主串中的字符是 c，两者不同了，如图 2 所示：



极客时间

图2 主串的point1指针所指向的字符c与子串point2指针所指向的字符a进行比较

依据子串的 next 数组（依据 next[3]的值），子串的 point2 指针应该恢复到子串第 3 个字符位置再用该位置的字符 a 与主串中 point1 所指向的字符 c 做比较，如图 3 所示：

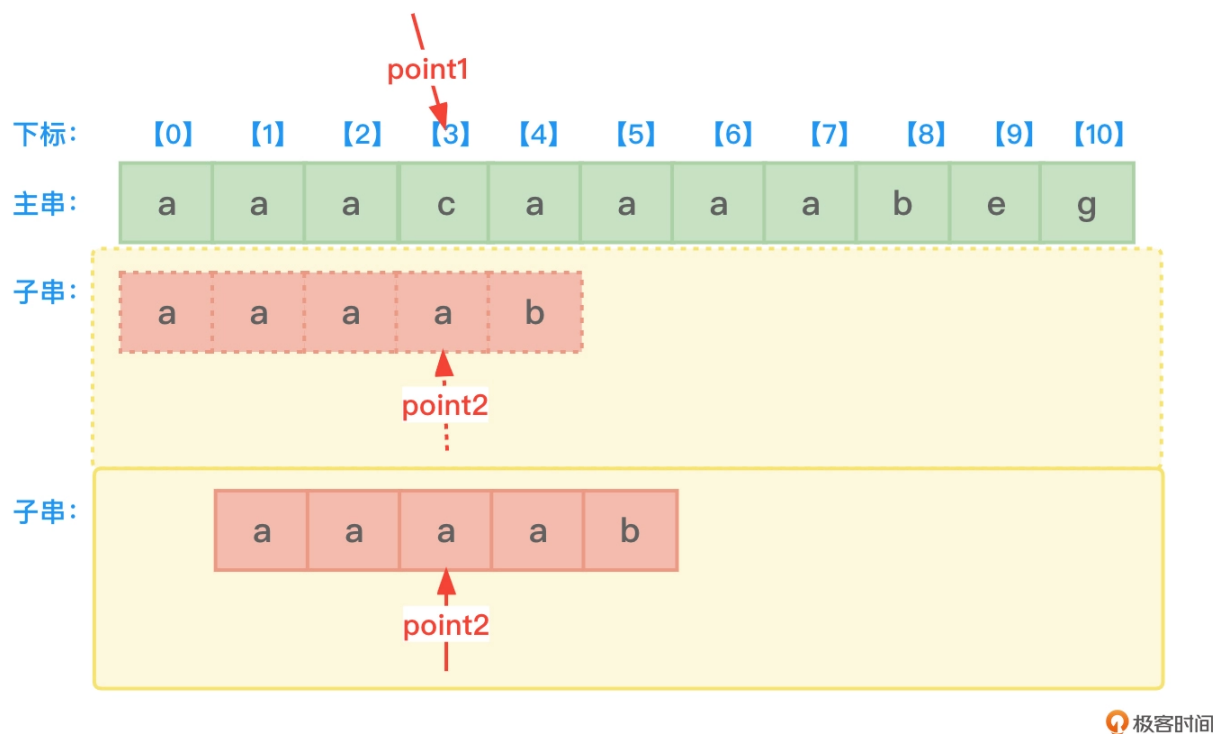
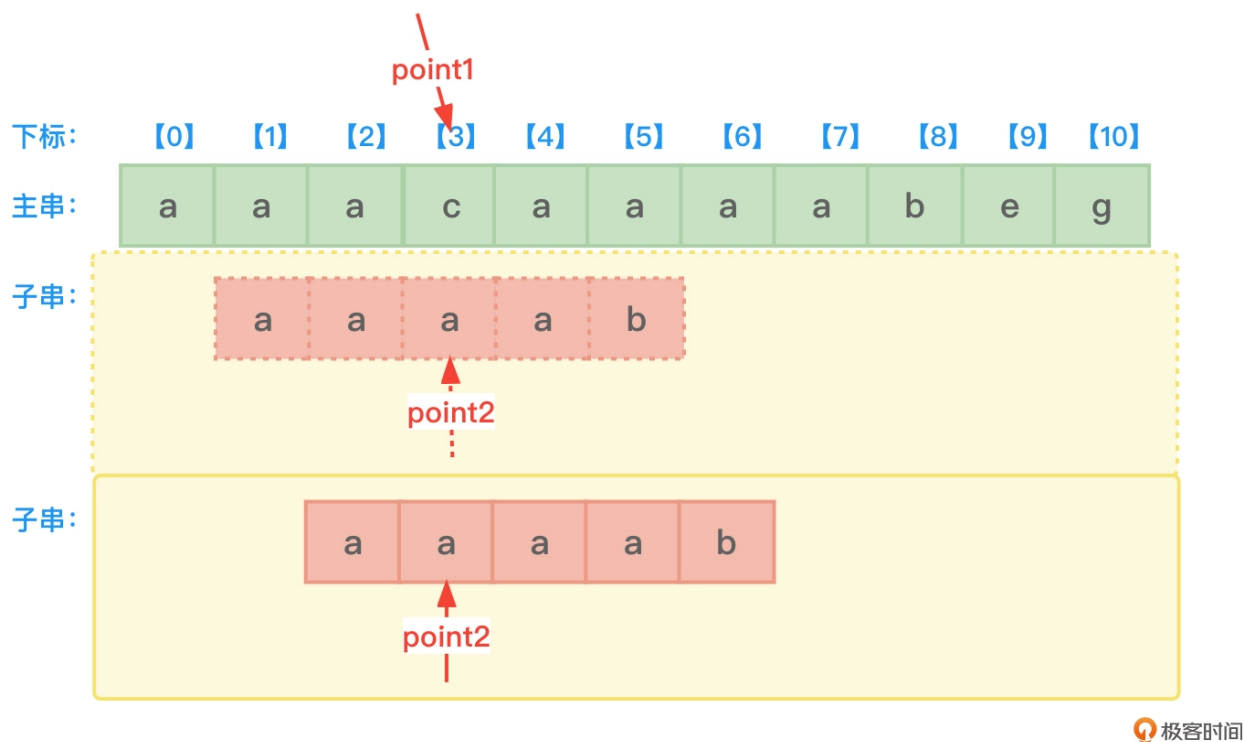


图3 主串的point1指针所指向的字符c与子串point2指针所指向的字符a进行比较

图 2 中已经进行了主串中的 c 和子串中的 a 比较，而图 3 又再次进行了主串中的 c 和子串中的 a 比较，显然这第 2 次比较毫无意义。

第 2 次主串中的 c 和子串中的 a 再比较，两者还是不同，依据子串的 next 数组（依据 next[2]的值），子串的 point2 指针应该恢复到子串第 2 个字符位置再用该位置的字符 a 与主串中 point1 所指向的字符 c 作比较，如图 4 所示：

shikey.com转载分享



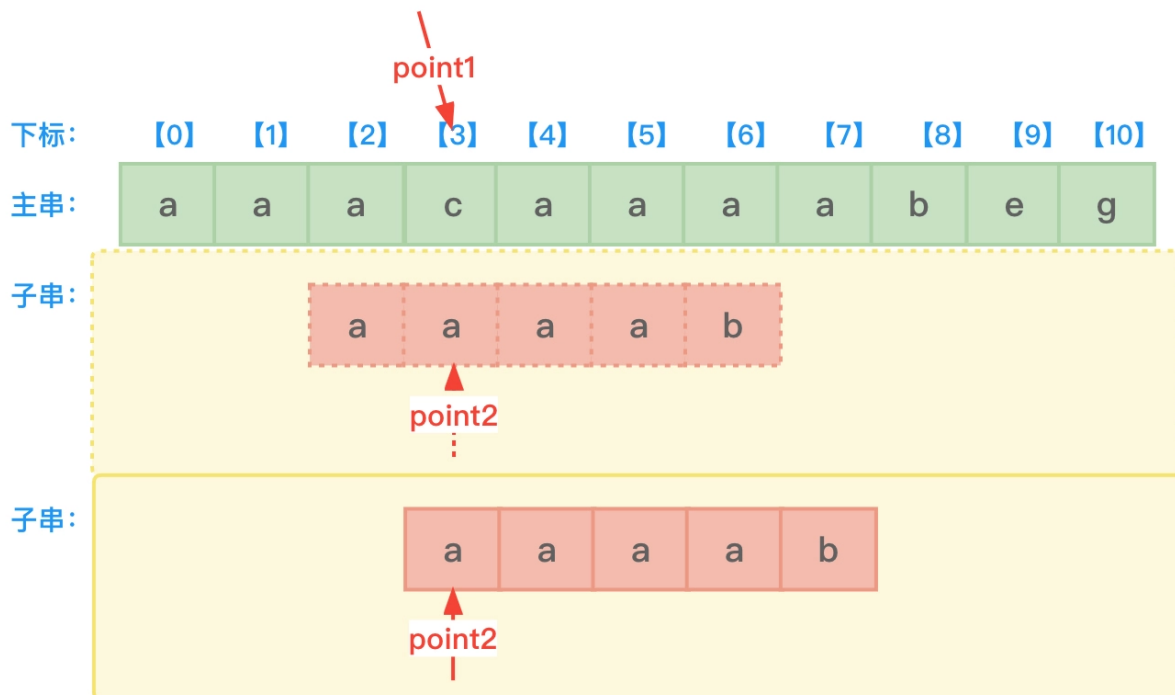
极客时间

图4 主串的point1指针所指向的字符c与子串point2指针所指向的字符a进行比较

这是第 3 次主串中的字符 c 和子串中的字符 a 进行比较，这次比较也毫无意义。

第 3 次主串中的 c 和子串中的 a 再比较，两者还是不同，依据子串的 next 数组（依据 next[1]的值），子串的 point2 指针应该恢复到子串第 1 个字符位置再用该位置的字符 a 与主串中 point1 所指向的字符 c 作比较，如图 5 所示：

shikey.com转载分享

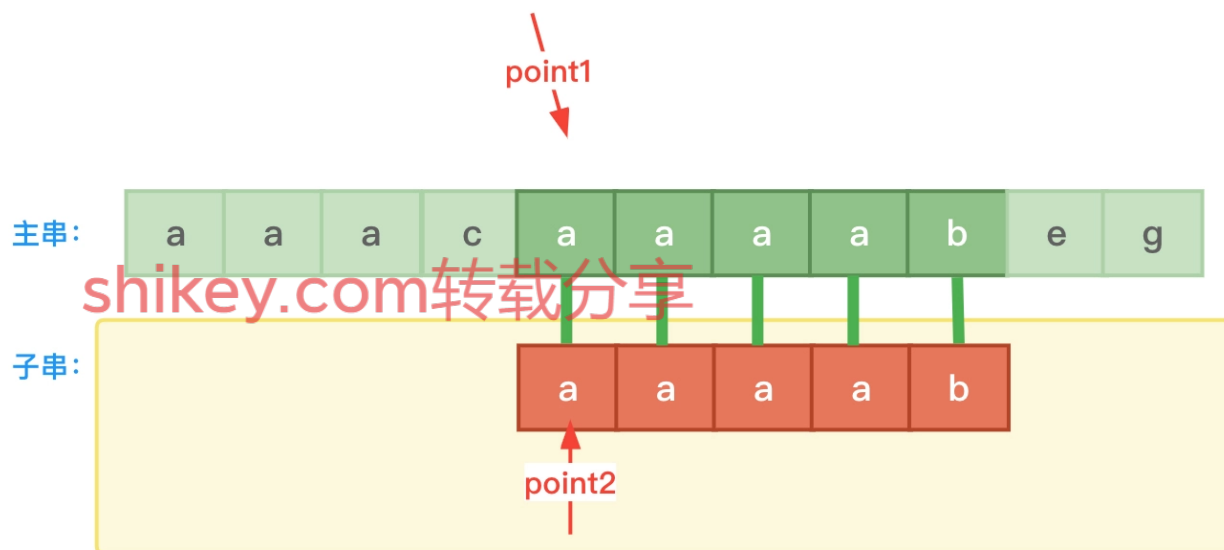


极客时间

图5 主串的point1指针所指向的字符c与子串point2指针所指向的字符a进行比较

这是第 4 次主串中的字符 c 和子串中的字符 a 进行比较，这次比较也毫无意义。

此时主串中 point1 的值会指向下个位置（依据 next[0]的值）。接着进行比较，最终在主串中找到子串。如图 6 所示：



极客时间

图6 主串point1指针指向下个位置，与子串比较并在主串中找到了子串

好了，我们总结一下使用 next 数组的字符串比较过程。

第 1 次比较：point1=0，point2=0；到 point1=point2=3 时出现主串字符 c 和子串 a 不匹配。

第 2 次比较：point2=2；主串字符 c 和子串 a 比较，多余的一次比较。

第 3 次比较：point2=1；主串字符 c 和子串 a 比较，多余的一次比较。

第 4 次比较：point2=0；主串字符 c 和子串 a 比较，多余的一次比较。

第 5 次比较：point1=4；point2=0，开始了新的主串与子串字符比较。

后续的描述步骤省略。

综合而言，第 2、3、4 次主串中的字符 c 与子串中的字符 a 比较是多余的。因为子串中的第 4 个字符与第 3 个、第 2 个、第 1 个字符都相同——都是 a。

## nextval 数组的求解步骤

看到了上述这么多次多余比较的产生，你就会发现 KMP 模式匹配算法仍有改进空间，也就是说，它可以进一步减少无意义的比较来提升效率。

这里我先揭晓答案，**KMP 算法是使用 nextval 数组取代 next 数组来达成提效的目的。**

nextval 数组可以通过 next 数组求得。nextval 数组被看作是优化过的 next 数组，首先看一看如何通过 next 数组推导出 nextval 数组。

这里以前面的一个稍复杂点的子串 T=“ababaaababaa”以及该子串的 next 数组为例叙述一下求解该子串的 nextval 数组的步骤，如图 7 所示：

下标:	<b>[0]</b>	<b>[1]</b>	<b>[2]</b>	<b>[3]</b>	<b>[4]</b>	<b>[5]</b>	<b>[6]</b>	<b>[7]</b>	<b>[8]</b>	<b>[9]</b>	<b>[10]</b>	<b>[11]</b>
子串:	a	b	a	b	a	a	a	b	a	b	a	a
next 数组:	0	1	1	2	3	4	2	2	3	4	5	6

极客时间

图7 子串T= “ababaaababaa” 及其对应的next数组

nextval[0]: 值固定是 0, 和 next[0]同值;

nextval[1]:  $\text{next}[1]-1=1-1=0$ , 因为  $T[0] \neq T[1]$ , 所以 nextval[1]保持 next[1]不变 =1;

nextval[2]:  $\text{next}[2]-1=1-1=0$ , 因为  $T[0]=T[2]$ , 所以 nextval[2]要等于 nextval[0]=0;

nextval[3]:  $\text{next}[3]-1=2-1=1$ , 因为  $T[1]=T[3]$ , 所以 nextval[3]要等于 nextval[1]=1;

nextval[4]:  $\text{next}[4]-1=3-1=2$ , 因为  $T[2]=T[4]$ , 所以 nextval[4]要等于 nextval[2]=0;

nextval[5]:  $\text{next}[5]-1=4-1=3$ , 因为  $T[3] \neq T[5]$ , 所以 nextval[5]保持 next[5]不变 =4

(在  $T[3] \neq T[5]$ 时保持 nextval[5]和 next[5]值相同可以称为**保留不同，可以看作是 nextval 数组的计算原则**) ;

nextval[6]:  $\text{next}[6]-1=2-1=1$ , 因为  $T[1] \neq T[6]$ , 所以 nextval[6]保持 next[6]不变 =2;

nextval[7]:  $\text{next}[7]-1=2-1=1$ , 因为  $T[1]=T[7]$ , 所以 nextval[7]要等于 nextval[1]=1;

nextval[8]:  $\text{next}[8]-1=3-1=2$ , 因为  $T[2]=T[8]$ , 所以 nextval[8]要等于 nextval[2]=0;

nextval[9]:  $\text{next}[9]-1=4-1=3$ , 因为  $T[3]=T[9]$ , 所以 nextval[9]要等于 nextval[3]=1;

nextval[10]:  $\text{next}[10]-1=5-1=4$ , 因为  $T[4]=T[10]$ , 所以 nextval[10]要等于 nextval[4]=0;

nextval[11]:  $\text{next}[11]-1=6-1=5$ , 因为  $T[5]=T[11]$ , 所以 nextval[11]要等于 nextval[5]=4;

观察一下上面这行 (其他行也同样有这个规律), 可知  $T[5]$ 和  $T[11]$ 是什么关系, 正好是子串中下标 11 左侧的 0~10 共 11 个字符的公共前后缀 “ababa” 的下一个字符。

为了方便观察，我们简单地把上述这些文字描述表示成图。但因为线条太密集，所以这里我只绘制了  $\text{nextval}[0] \sim \text{nextval}[7]$  的求值过程，箭头中的数字代表下标，比如求解  $\text{nextval}[1]$  时， $\text{next}[1]$  向上指向的两个红色箭头表示指向的  $T[0]$  和  $T[1]$  不等（相等用绿色箭头）， $\text{next}[1]$  指向  $\text{nextval}[1]$  的箭头表示  $\text{nextval}[1]$  的值来自于  $\text{next}[1]$ ，如图 8 所示：

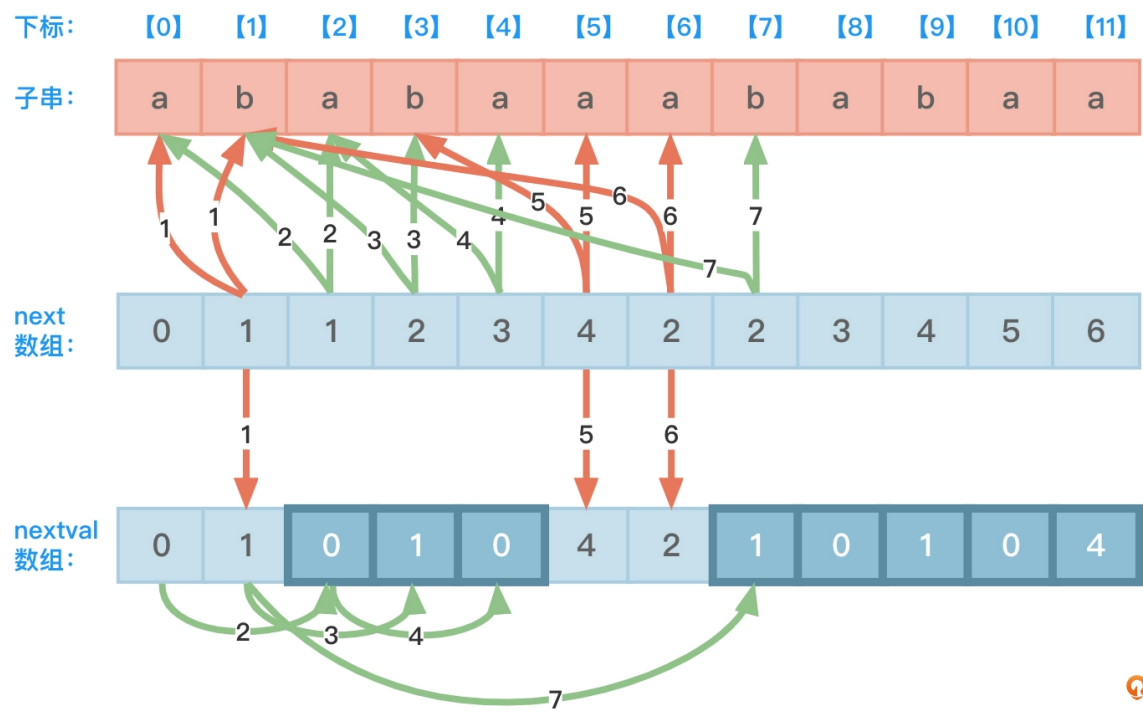


图8 子串 $T = \text{"ababaaababaa"}$  及对应的 $\text{next}$ 数组和 $\text{nextval}$ 数组（展示 $\text{nextval}$ 数组求解步骤）

我们可以通过上面的描述总结出求解  $\text{nextval}$  数组的规则（数组下标用  $i$  表示）。

- 1. 数组  $\text{nextval}$  下标  $i$  为 0 的元素值固定为 0。
- 2. 数组  $\text{nextval}$  下标  $i > 0$  时，分两种情况：

如果  $T[i] \neq T[\text{next}[i]-1]$ ，则  $\text{nextval}[i]$  等于  $\text{next}[i]$ 。

如果  $T[i] = T[\text{next}[i]-1]$ ，则  $\text{nextval}[i]$  等于  $\text{nextval}[\text{next}[i]-1]$ 。

前面说过两个重要的观点：



1. next 数组存在的意义是指示当子串中某个位置的字符与主串相应位置字符不匹配时，应该将子串中的哪一位字符与主串中的当前位 / 下一位字符做比较。
2. 在 next 数组中，因为没考虑当前字符的位置情况，只考虑了字符不匹配时单纯的指针移动问题（point1 和 point2 值的改变），这很可能导致移动后将进行比较的字符仍和上一次不匹配的字符相同导致产生多次重复无效比较。

显然，nextval 数组存在的意义与 next 数组相同。而 nextval 数组对 next 数组的值进行了进一步修正，排除掉了一些无效的移动和比较，保证子串中下一个即将（与主串当前位置）参与比较的字符和上次参与比较且不匹配的字符不同。

这里为了方便观察，再说回前面图 1 所示的子串和其 next 数组，优化该 next 数组得到 nextval 数组，这次我们换一种方式叙述一下通过 next 数组求解 nextval 数组的步骤。

1. 先把 next 数组的各个元素值原封不动的赋给 nextval 数组。
2. 然后从左到右依次检查 nextval 数组的各个元素。

nextval[0]值是 0 固定不变。

nextval[1]:  $\text{next}[1]-1=1-1=0$ ，因为  $T[0]=T[1]$ ，所以 nextval[1]要等于 nextval[0]=0；这意思应该就是子串中下标为 1 的字符和下标为 0 的字符相同（都是 a），因此如果和下标为 0 的字符不等，则就不必和下标为 1 的字符比较了。

nextval[2]:  $\text{next}[2]-1=2-1=1$ ，因为  $T[1]=T[2]$ ，所以 nextval[2]要等于 nextval[1]=0；这意思应该就是子串中下标为 2 的字符和下标为 1 的字符相同，因此如果和下标为 1 的字符不等，则就不必和下标为 2 的字符比较了。

nextval[3]:  $\text{next}[3]-1=3-1=2$ ，因为  $T[2]=T[3]$ ，所以 nextval[3]要等于 nextval[2]=0；这意思应该就是子串中下标为 3 的字符和下标为 2 的字符相同，因此如果和下标为 2 的字符不等，则就不必和下标为 3 的字符比较了。

nextval[4]:  $\text{next}[4]-1=4-1=3$ ，因为  $T[3]\neq T[4]$ ，所以 nextval[4]保持 next[4]不变 =4；

3. 最终 nextval 数组值如图 9 所示：

下标:	<b>[0]</b>	<b>[1]</b>	<b>[2]</b>	<b>[3]</b>	<b>[4]</b>
子串:	a	a	a	a	b
next 数组:	0	1	2	3	4
nextval 数组:	0	0	0	0	4

图9 子串“aaaab”对应的next数组和nextval数组

现在，在执行 KMP 模式匹配算法从主串中查找子串时，将不再使用 next 数组，而是使用 nextval 数组，我们来看看具体的操作步骤。

首先，前面图 2 中，主串与子串开始进行比较，当比较到下标为 3 的位置时，子串中的字符是 a，主串中的字符是 c，两者不同了。

其次，依据子串的 nextval[3]的值 0，代表着主串中 point1 会指向下个位置，同时子串的 point2 会指向子串的开始位置。接着主串和子串的当前位置字符再做比较，如图 10 所示使用 next 数组和使用 nextval 数组的区别：

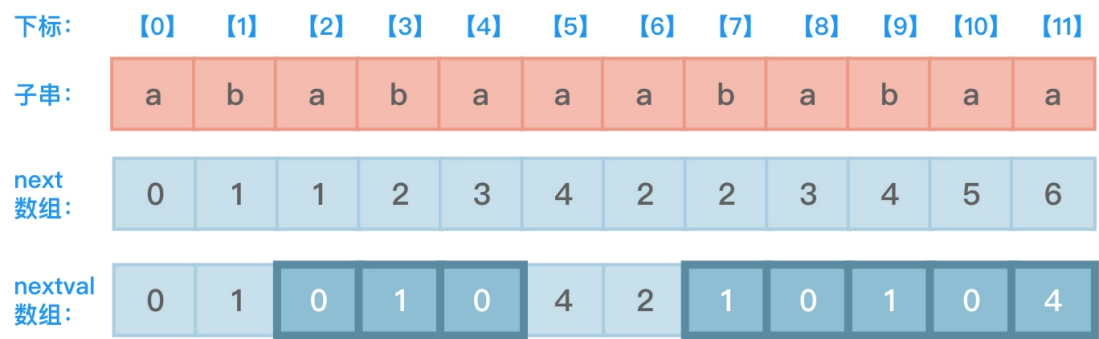


图10 当发生主串与子串字符不匹配时使用next数组和使用nextval数组的主串和子串指针point1、point2指向差别

最后，从图 10 可以看到，使用 nextval 数组子串向右跳跃的位置远比使用 next 数组大（大量减少了对比次数的发生）。使用 KMP 算法的效率得到进一步提升。

# nextval 数组的工作原理

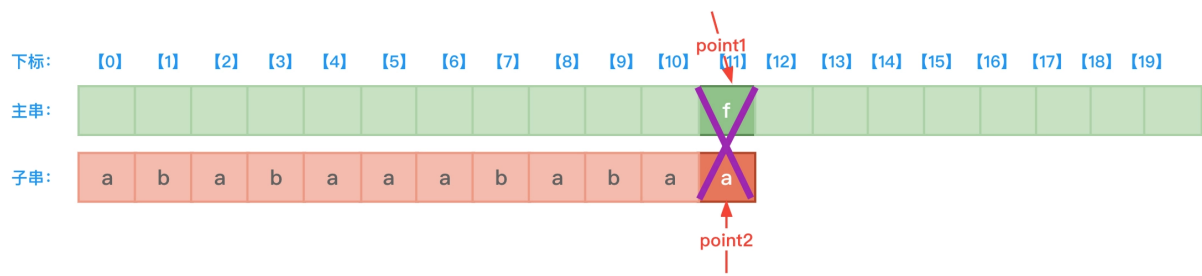
基于 next 数组来求解 nextval 数组。可能许多同学会疑惑为什么会这样来求 nextval 数组。也许你把这个问题理解复杂了，其实是挺简单的，就以前面图 8 这个子串以及它对应的 next 数组和 nextval 数组来解释，重新绘制如下图 11 所示：



极客时间

图11 子串T= “ababaaababaa” 及对应的next数组和nextval数组

这里就以下标 11 这个位置的子串字符 a 来说明 next 数组和 nextval 数组的用途。从下标 0 到 10，主串和子串的对应字符都相等，但当主串和子串比较到下标为 11 的字符时，发现两者不等。如图 12 所示：



极客时间

图12 下标11位置的主串和子串字符不同

在图 12 中，主串的字符 f 和子串的字符 a 不匹配，如果使用 next 数组，则 next[11]值为 6，这意味着子串中下标为 11 的字符 a 之前有宽度为 5 个字符的最长公共前后缀，所以子串中的 point2 指针应该如下图 13 这样调整：

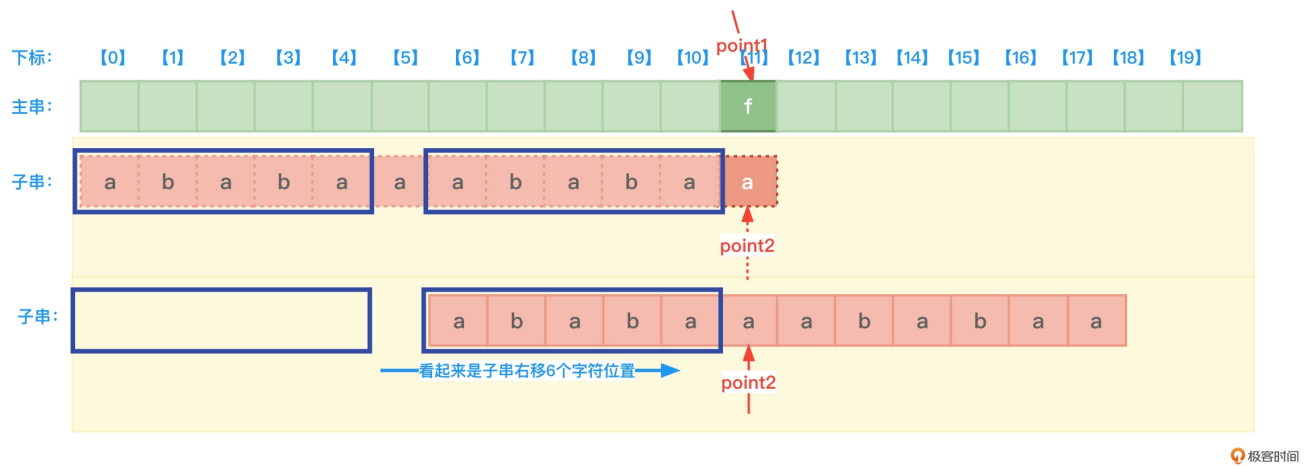


图13 KMP算法使用next数组进行字符串匹配

从图 13 可以看到，point2 指针指向了**针对于子串的下标为 5 (6-1) 的位置**，看上去好像是子串针对主串右移了 6 个字符位置。next[11] = 6，这个数字 6 其实与子串中下标为 11 的字符（第 12 个字符）a 没有任何关系。这就是 next 数组记录数据的不足之处——只能记录该位置之前的所有字符的公共前后缀长度信息。接下来，还是要进行子串中第 6 个字符 a 与主串中的字符 f 比较，这显然是重复比较。

KMP 优化算法不再使用 next 数组，而是使用了 nextval 数组。通过前面的图 11 可以看到，nextval[11]=4。下面看一看当主串和子串第 11 个下标位置字符不等时使用 nextval 数组的效果。子串中的 point2 指针应该如下图 14 这样调整：

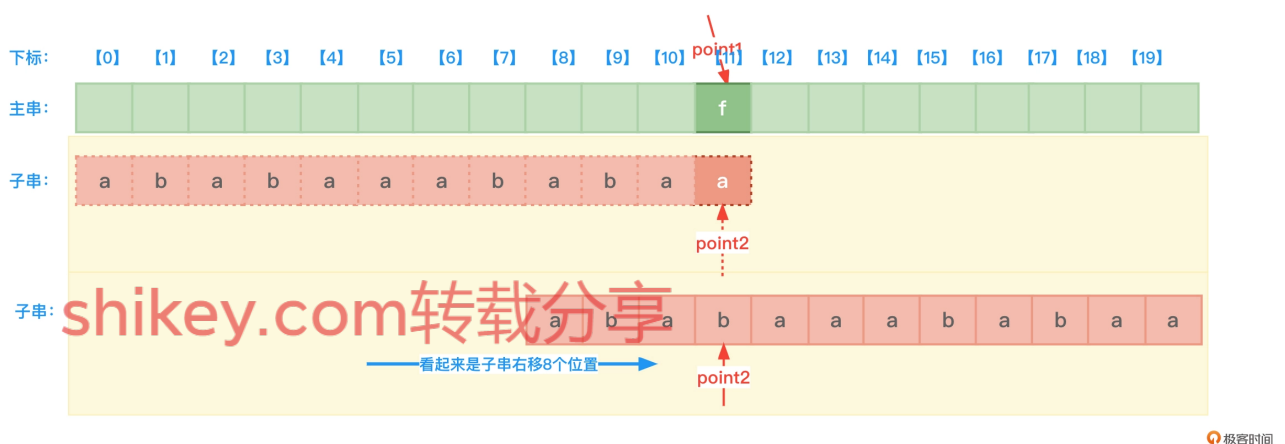


图14 KMP优化算法使用nextval数组进行字符串匹配

从图 14 可以看到，point2 指针指向了**针对于子串的下标为 3 (4-1) 的位置**，看上去好像是子串针对主串右移了 4 个字符位置。nextval[11] = 4。这个数字 4 怎么来的，通过前面的学

习你应该很清楚，如果忘记了不妨回顾前面的内容。这里我也给你一个提示：nextval[11]要等于 nextval[5]=4。

最后，我们比较一下 next[11]和 nextval[11]。

首先，next[11]=6，而 nextval[11]=4，两者相差了 2。

其次，next[11]=6，这个数字 6 与子串 T 中的字符 T[11]没有任何关系。该数字只代表下标 11 这个位置之前的所有字符的公共前后缀长度信息（6-1=5）。


接着，nextval[11]=4，根据 next 数组推导 nextval 数组的步骤你应该知道，这个数字 4 就与子串 T 中的 T[11]有关系了，我不得不佩服该优化算法设计者超越常人的思考能力。因为 4-1=3，这个 3 可以理解成包含下面两个信息。

1. 下标 11 这个位置之前的所有字符的公共前后缀长度信息长度是 5 个。
2. 额外还有 2 个字符与当前 T[11]位置的字符相同，在比较的时候应该略过。5-2=3，所以 point2 直接跳到了针对于子串的下标为 3（4-1）的位置，从而也就避免了图 13 中再次进行子串的字符 a 与主串字符 f 比较的浪费。

## nextval 数组的求解实现代码

先求出 next 数组，然后根据求解 nextval 数组的规则来求得 nextval 数组值当然是可以的。但是，通过修改求 next 数组的代码，求 next 数组的同时也求得 nextval 数组也是完全可以做到的。

仿照前面求解 next 数组的代码 getNextArray\_Classic() 来实现同时求解 next 和 nextval 数组的 getNextAndNextValArray\_Classic()，代码如下：

 复制代码

```
1 //根据给的nextval的下标值，求nextval[idx]的值
2 void getNextValArray(int i, int next[], int nextval[])
3 {
4     //数组nextval下标i为0的元素值固定为0
5     if (i == 0)
```

```

6  {
7      nextval[i] = 0;
8  }
9  else //数组nextval下标i > 0时, 分两种情况:
10 {
11     if (ch[i] != ch[next[i] - 1])
12     {
13         //如果T[i]≠T[next[i]-1], 则nextval[i] 等于next[i]
14         nextval[i] = next[i];
15     }
16     else
17     {
18         //如果T[i] = T[next[i] - 1], 则nextval[i]等于nextval[next[i] - 1]
19         nextval[i] = nextval[next[i] - 1];
20     }
21 }
22 return;
23 }
24
25 //求本串的next和nextval数组
26 void getNextAndNextValArray_Classic(int next[], int nextval[])
27 {
28     if (length < 1)
29         return;
30
31     //next数组的前两个元素肯定是0和1
32     if (length == 1) //只有一个字符
33     {
34         next[0] = 0;
35         return;
36     }
37
38     next[0] = 0;
39     next[1] = 1;
40
41     getNextValArray(0, next, nextval);
42     getNextValArray(1, next, nextval);
43
44     if (length == 2) //只有二个字符
45     {
46         return;
47     }
48
49     //至少三个字符
50     int next_idx = 2;    //要求的next数组中下标为2的元素值
51     int qz_tail_idx = 0; //前缀末尾位置
52
53     while (next_idx < length)
54     {

```

```

55     if (ch[qz_tail_idx] == ch[next_idx - 1]) //next_idx-1代表后缀末尾位置
56     {
57         next[next_idx] = (qz_tail_idx + 1) + 1;    //qz_tail_idx+1就是前缀的宽度
58
59         //这里求nextval元素值：求next元素值后就可以求nextval元素值
60         getNextValArray(next_idx, next, nextval);
61
62         next_idx++;
63         qz_tail_idx++; //前缀末尾位置：其实这样写也OK: qz_tail_idx = next[next_idx - 1]
64     }
65     else
66     {
67         //qz_tail_idx = next[qz_tail_idx] - 1; //这句是最难理解的代码
68         qz_tail_idx = nextval[qz_tail_idx] - 1; //注意可以用这句替换上一句，其实不替换也可
69
70         //qz_tail_idx允许等于0，等于0有机会下次while时再比较一次，所以下面只判断qz_tail_idx
71         if (qz_tail_idx < 0)
72         {
73             //没找到前缀
74             qz_tail_idx = 0;
75             next[next_idx] = 1;
76
77             //这里求nextval元素值：求next元素值后就可以求nextval元素值
78             getNextValArray(next_idx, next, nextval);
79
80             ++next_idx;
81         }
82     }
83 } //end while (next_idx < length)
84 return;
85 }

```

接着，就可以利用 nextval 数组取代原来的 next 数组来进行子串查找了。

但要正常使用 nextval 数组，还需要对 StrKMPIndex() 实现代码升级优化一下。这里提示一下，优化后是可以兼容以往对该成员函数的调用的。因为以往 StrKMPIndex 实现中 next 数组除下标为 0 的数组元素外，其他数组元素的值都大于 0，而采用 nextval 数组后，任何一个 nextval 数组元素值都可能等于 0。

升级后的 StrKMPIndex 实现代码如下：


```

1 //KMP模式匹配算法接口，返回子串中第一个字符在主串中的下标，如果没找到子串，则返回-1
2 //nextornextval：下一步数组（前缀表/前缀数组）
3 //pos：从主串的什么位置开始匹配子串，默认从位置0开始匹配子串
4 int StrKMPIndex(const MySString& substr, int nextornextval[], int pos = 0)
5 {
6     if (length < substr.length) //主串还没子串长，那不可能找到
7         return -1;
8
9     int point1 = pos; //指向主串
10    int point2 = 0; //指向子串
11
12    while (ch[point1] != '\0' && substr.ch[point2] != '\0')
13    {
14        if (ch[point1] == substr.ch[point2])
15        {
16            //两个指针都向后走
17            point1++;
18            point2++;
19        }
20        else //两者不同
21        {
22            //point1和point2两个指针的处理
23            if (point2 == 0) //图7.5.3_1可以看到，下标0号位置子串的字符如果与主串字符不匹配则后缀
24            {
25                point1++; //主串指针指向下一位
26            }
27            else
28            {
29                //新增对nextval数组的处理
30                if (nextornextval[point2] == 0) //因为nextval数组的任何元素都可能等于0
31                {
32                    point1++; //主串指针指向下一位
33                    point2 = 0; //子串的point2值会指向子串的开始位置
34                }
35                else
36                {
37                    //走到这个分支的，主串指针point1不用动，只动子串指针point2
38                    point2 = nextornextval[point2]; //第这些个子串中的字符与主串当前位字符做比
39                }
40            }
41        }
42    } //end while
43
44    if (substr.ch[point2] == '\0')
45    {
46        //找到了子串
47        return point1 - point2;
48    }
49    return -1;

```



在 main 主函数中，增加如下测试代码：

 复制代码

```
1 //求本串的next和nextval数组—典型的KMP算法求解next数组的代码写法
2 MySString mys15sub; //子串
3 mys15sub.StrAssign("ababaaababaa");
4 int* mynextarray15 = new int[mys15sub.length]; //next数组
5 int* mynextvalarray15 = new int[mys15sub.length]; //nextval数组
6 cout <<"本次采用典型的KMP算法求解next和nextval数组然后利用nextval数组进行模式串匹配查找：--
7 mys15sub.getNextAndNextValArray_Classic(mynextarray15, mynextvalarray15); //求nex
8 MySString mys15master; //主串
9 mys15master.StrAssign("abbabbababaaababaaa");
10 cout <<"StrKMPIIndex()结果为"<< mys15master.StrKMPIIndex(mys15sub, mynextvalarray15)
11 delete[]mynextarray15;
12 delete[]mynextvalarray15;
```

新增代码执行结果如下：

```
本次采用典型的KMP算法求解next和nextval数组然后利用nextval数组进行模式串匹配查找：--
--
StrKMPIIndex()结果为6
```

改进后的 KMP 算法，引入了 nextval 数组，虽然 nextval 数组的值是借助 next 数组的值来计算的，但通过适当的修改代码，不通过 next 数组而是直接求得 nextval 数组的值也是可以的。这说明改进后的 KMP 算法执行效率更高，但空间复杂度方面并没有发生改变。

## 小结 [shikey.com](https://shikey.com)转载分享

这节课，我带你详细理解了“KMP 模式匹配算法通过 next 数组参与计算来达到加速查找子串”的过程中，**next 数组存在的局限性在于多次重复无效比较导致的效率降低。**

于是，我们提出了 KMP 模式匹配算法的改进，也就是用 nextval 数组取代 next 数组来进一步减少无意义的比较以提升效率。

这节课我们详细讨论了通过 next 数组来求解 nextval 数组，并解释了之所以这样求解 nextval 数组的原理，相信对你理解算法的改进思路有着不小的帮助。

最后，我给出了求解 nextval 数组的实现代码，也给出了通过 nextval 数组实现 KMP 模式匹配算法的范例，最终使得在主串中寻找子串的效率更高。

## 思考题

1. 请分析 KMP 模式匹配算法中 nextval 数组的含义和作用，并与使用 next 数组时的算法进行比较。
2. 试着比较改进后的 KMP 模式匹配算法和原算法的时间复杂度和空间复杂度，并说明改进后的算法的优势和不足。

欢迎你在留言区和我交流，如果觉得有所收获，也可以把课程分享给更多的朋友一起学习。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

shikey.com转载分享