# **[数据结构20：KMP算法(快速模式匹配算法)详解](https://www.cnblogs.com/ciyeer/p/9035072.html)**

通过上一节的介绍，学习了串的普通模式匹配算法，大体思路是：模式串从主串的第一个字符开始匹配，每匹配失败，主串中记录匹配进度的指针 i 都要进行 i-j+1 的回退操作（这个过程称为“指针回溯”），同时模式串向后移动一个字符的位置。一次次的循环，直到匹配成功或者程序结束。

"KMP"算法相比于"BF"算法，优势在于：

* 在保证指针 i 不回溯的前提下，当匹配失败时，让模式串向右移动最大的距离；
* 并且可以在O(n+m)的时间数量级上完成对串的模式匹配操作；

故，"KMP"算法称为“快速模式匹配算法”。

## **模式串向右移动距离的计算**

在模式串和主串匹配时，各有一个指针指向当前进行匹配的字符（主串中是指针 i ，模式串中是指针 j ），在保证 i 指针不回溯的前提下，如果想实现功能，就只能让 j 指针回溯。

j 指针回溯的距离，就相当于模式串向右移动的距离。 j 指针回溯的越多，说明模式串向右移动的距离越长。

计算模式串向右移动的距离，就可以转化成：当某字符匹配失败后， j 指针回溯的位置。  
  
对于一个给定的模式串，其中每个字符都有可能会遇到匹配失败，这时对应的 j 指针都需要回溯，具体回溯的位置其实还是由模式串本身来决定的，和主串没有关系。  
  
模式串中的每个字符所对应 j 指针回溯的位置，可以通过算法得出，得到的结果相应地存储在一个数组中（默认数组名为 next ）。  
  
计算方法是：对于模式串中的某一字符来说，提取它前面的字符串，分别从字符串的两端查看连续相同的字符串的个数，在其基础上 +1 ，结果就是该字符对应的值。

每个模式串的第一个字符对应的值为 0 ，第二个字符对应的值为 1 。

例如：求模式串 “abcabac” 的 next 。前两个字符对应的 0 和 1 是固定的。  
  
对于字符 ‘c’ 来说，提取字符串 “ab” ，‘a’ 和 ‘b’ 不相等，相同的字符串的个数为 0 ，0 + 1 = 1 ，所以 ‘c’ 对应的 next 值为 1 ；  
  
第四个字符 ‘a’ ，提取 “abc” ，从首先 ‘a’ 和 ‘c’ 就不相等，相同的个数为 0 ，0 + 1 = 1 ，所以，‘a’ 对应的 next 值为 1 ；  
  
第五个字符 ‘b’ ，提取 “abca” ，第一个 ‘a’ 和最后一个 ‘a’ 相同，相同个数为 1 ，1 + 1 = 2 ，所以，‘b’ 对应的 next 值为 2 ；  
  
第六个字符 ‘a’ ，提取 “abcab” ，前两个字符 “ab” 和最后两个 “ab” 相同，相同个数为 2 ，2 + 1 = 3 ，所以，‘a’ 对应的 next 值为 3 ；  
  
最后一个字符 ‘c’ ，提取 “abcaba” ，第一个字符 ‘a’ 和最后一个 ‘a’ 相同，相同个数为 1 ，1 + 1 = 2 ，所以 ‘c’ 对应的 next 值为 2 ；  
  
所以，字符串 “abcabac” 对应的 next 数组中的值为（0,1,1,1,2,3,2）。  
  
上边求值过程中，每次都需要判断字符串头部和尾部相同字符的个数，而在编写算法实现时，对于某个字符来说，可以借用前一个字符的判断结果，计算当前字符对应的 next 值。  
  
具体的算法如下：  
  
模式串T为(下标从1开始)：“abcabac”  
next数组(下标从1开始)：    01  
  
第三个字符 ‘c’ ：由于前一个字符 ‘b’ 的 next 值为 1 ，取 T[1] = ‘a’ 和 ‘b’ 相比较，不相等，继续；由于 next[1] = 0，结束。 ‘c’ 对应的 next 值为1；（只要循环到 next[1] = 0 ,该字符的 next 值都为 1 ）  
  
模式串T为：                  “abcabac”  
next数组(下标从1开始)：011  
  
第四个字符 ’a‘ ：由于前一个字符 ‘c’ 的 next 值为 1 ，取 T[1] = ‘a’ 和 ‘c’ 相比较，不相等，继续；由于 next[1] = 0 ，结束。‘a’ 对应的 next 值为 1 ；  
  
模式串T为：                  “abcabac”  
next数组(下标从1开始)：0111  
  
第五个字符 ’b’ ：由于前一个字符 ‘a’ 的 next 值为 1 ，取 T[1] = ‘a’ 和 ‘a’ 相比较，相等，结束。 ‘b’ 对应的 next 值为：1(前一个字符 ‘a’ 的 next 值) + 1 = 2 ；  
  
模式串T为：                  “abcabac”  
next数组(下标从1开始)：01112  
  
第六个字符 ‘a’ ：由于前一个字符 ‘b’ 的 next 值为 2，取 T[2] = ‘b’ 和 ‘b’ 相比较，相等，所以结束。‘a’ 对应的 next 值为：2 (前一个字符 ‘b’ 的 next 值) + 1 = 3 ；  
  
模式串T为：                  “abcabac”  
next数组(下标从1开始)：011123  
  
第七个字符 ‘c’ ：由于前一个字符 ‘a’ 的 next 值为 3 ，取 T[3] = ‘c’ 和 ‘a’ 相比较，不相等，继续；由于 next[3] = 1 ，所以取 T[1] = ‘a’ 和 ‘a’ 比较，相等，结束。‘a’ 对应的 next 值为：1 ( next[3] 的值) + 1 = 2 ；  
  
模式串T为：                  “abcabac”  
next数组(下标从1开始)：0111232  
  
算法实现：

[IMG_256](javascript:void(0);)

#include <stdio.h>

#include <string.h>  
void Next(char \*T, int \*next)  
{

　　int i = 1;

　　next[1] = 0;

　　int j = 0;

　　while (i<strlen(T))   
　　{

　　　　if (j==0 || T[i-1]==T[j-1])   
　　　　{

　　　　　　i++;

　　　　　　j++;

　　　　　　next[i] = j;

　　　　}  
　　　　else  
　　　　{

　　　　　　j = next[j];

　　　　}

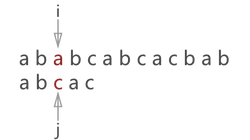
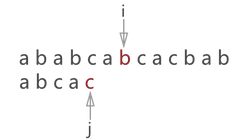
　　}

}

[IMG_257](javascript:void(0);)

注意：在此程序中，next 数组使用的下标初始值为 1 ，next[0] 没有用到（也可以存放 next 数组的长度）。而串的存储是从数组的下标 0 开始的，所以程序中为 T[i-1] 和 T[j-1]。

## **基于next的KMP算法的实现**

先看一下 KMP 算法运行流程（假设主串：ababcabcacbab，模式串：abcac）。  
  
第一次匹配：  
  
匹配失败，i 指针不动，j = 1（字符‘c’的next值）;  
  
第二次匹配：  
  
相等，继续，直到：  
  
匹配失败，i 不动，j = 2 （ j 指向的字符 ‘c’ 的 next 值）;  
  
第三次匹配：  
  
相等，i 和 j 后移，最终匹配成功。

使用普通算法，需要匹配 6 次；而使用 KMP 算法，则只匹配 3 次。

实现代码：

[IMG_262](javascript:void(0);)

int KMP(char \*S, char \*T)  
{

　　int next[10];

　　Next(T, next);　　//根据模式串T,初始化next数组

　　int i = 1;

　　int j = 1;

　　while (i<=strlen(S) && j<=strlen(T))   
　　{

　　　　//j==0:代表模式串的第一个字符就和指针i指向的字符不相等；S[i-1]==T[j-1],如果对应位置字符相等，两种情况下，指向当前测试的两个指针下标i和j都向后移

　　　　if (j==0 || S[i-1]==T[j-1])   
　　　　{

　　　　　　i++;

　　　　　　j++;

　　　　}

　　　　else  
　　　　{

　　　　　　j=next[j];//如果测试的两个字符不相等，i不动，j变为当前测试字符串的next值

　　　　}

　　}

　　if (j>strlen(T))   
　　{  
　　　　//如果条件为真，说明匹配成功

　　　　return i-(int)strlen(T);

　　}

　　return -1;

}

[IMG_263](javascript:void(0);)

## **KMP算法完整代码**

[IMG_264](javascript:void(0);)

#include <stdio.h>

#include <string.h>  
void Next(char \*T, int \*next)  
{

　　int i = 1;

　　next[1] = 0;

　　int j = 0;

　　while (i<strlen(T))   
　　{

　　　　if (j==0 || T[i-1]==T[j-1])   
　　　　{

　　　　　　i++;

　　　　　　j++;

　　　　　　next[i] = j;

　　　　}  
　　　　else  
　　　　{

　　　　　　j = next[j];

　　　　}

　　}

}  
int KMP(char \*S, char \*T)  
{

　　int next[10];

　　Next(T, next);　　//根据模式串T,初始化next数组

　　int i = 1;

　　int j = 1;

　　while (i<=strlen(S)&&j<=strlen(T))   
　　{

　　　　//j==0:代表模式串的第一个字符就和当前测试的字符不相等；S[i-1]==T[j-1],如果对应位置字符相等，两种情况下，指向当前测试的两个指针下标i和j都向后移

　　　　if (j==0 || S[i-1]==T[j-1])   
　　　　{

　　　　　　i++;

　　　　　　j++;

　　　　}

　　　　else  
　　　　{

　　　　　　j = next[j];//如果测试的两个字符不相等，i不动，j变为当前测试字符串的next值

　　　　}

　　}

　　if (j>strlen(T))   
　　{  
　　　　//如果条件为真，说明匹配成功

　　　　return i-(int)strlen(T);

　　}

　　return -1;

}  
int main()   
{

　　int i = KMP("ababcabcacbab", "abcac");

　　printf("%d", i);

　　return 0;

}

运行结果：6

[IMG_265](javascript:void(0);)

## **升级版的next**

注意：KMP 算法的关键在于 next 数组的确定，其实对于上边的KMP算法中的next数组，不是最精简的，还可以简化。  
  
例如：

模式串T：a b c a c  
    next  ：0 1 1 1 2

在模式串“abcac”中，有两个字符 ‘a’，我们假设第一个为 a1，第二个为 a2。在程序匹配过程中，如果 j 指针指向 a2 时匹配失败，那么此时，主串中的 i 指针不动，j 指针指向 a1 ，很明显，由于 a1==a2，而 a2！=S[i]，所以 a1 也肯定不等于 S[i]。  
  
为了避免不必要的判断，需要对 next 数组进行精简，对于“abcac”这个模式串来说，由于 T[4] == T[next[4]] ，所以，可以将next数组改为：

模式串T：a b c a c  
    next  ：0 1 1 0 2

这样简化，如果匹配过程中由于 a2 匹配失败，那么也不用再判断 a1 是否匹配，因为肯定不可能，所以直接绕过 a1，进行下一步。  
  
实现代码：

void Next(char \*T, int \*next)  
{

　　int i = 1;

　　next[1] = 0;

　　int j = 0;

　　while (i<strlen(T))   
　　{

　　　　if (j==0 || T[i-1]==T[j-1])   
　　　　{

　　　　　　i++;

　　　　　　j++;

　　　　　　if (T[i-1] != T[j-1])   
　　　　　　{

　　　　　　　　next[i] = j;

　　　　　　}

　　　　　　else  
　　　　　　{

　　　　　　　　next[i] = next[j];

　　　　　　}

　　　　}  
　　　　else  
　　　　{

　　　　　　j = next[j];

　　　　}

　　}

}

[IMG_267](javascript:void(0);)

使用精简过后的 next 数组在解决例如模式串为“aaaaaaab”这类的问题上，会减少很多不必要的判断次数，提高了KMP算法的效率。  
  
例如：精简前为 next1，精简后为 next2：

模式串：a a a a a a a b  
  next1：0 1 2 3 4 5 6 7  
  next2：0 0 0 0 0 0 0 7

## **总结**

KMP 算法，之所以比 BF 算法快的根本原因在于：KMP 算法其实也和 BF 算法一样，都是从主串开头开始匹配，但是在匹配过程中，KMP算法记录了一些必要的信息。根据这些信息，在后续的匹配过程中，跳过了一些无意义的匹配过程。