KMP算法（快速模式匹配算法）C语言详解

< 上一页 下一页 >

快速模式匹配算法，简称 KMP 算法，是在 BF 算法基础上改进得到的算法。学习 BF 算法我们知道，该算法的实现过程就是 "傻瓜式" 地用模式串（假定为子串的串）与主串中的字符一一匹配，算法执行效率不高。

KMP 算法不同，它的实现过程接近人为进行模式匹配的过程。例如，对主串 A（"ABCABCE"）和模式串 B（"ABCE"）进行模式匹配，如果人为去判断，仅需匹配两次。

图 1 第一次人为模式匹配

第一次如图 1 所示，最终匹配失败。但在本次匹配过程中，我们可以获得一些信息，模式串中 "ABC" 都和主串对应的字符相同，但模式串中字符 'A' 与 'B' 和 'C' 不同。

因此进行下次模式匹配时，没有必要让串 B 中的 'A' 与主串中的字符 'B' 和 'C' 一一匹配（它们绝不可能相同），而是直接去匹配失败位置处的字符 'A'，如图 2 所示：

图 2 第二次人为模式匹配

至此，匹配成功。若使用 BF 算法，则此模式匹配过程需要进行 4 次。

由此可以看出，每次匹配失败后模式串移动的距离不一定是 1，某些情况下一次可移动多个位置，这就是 KMP 模式匹配算法。

那么，如何判断匹配失败后模式串向后移动的距离呢?

模式串移动距离的判断

每次模式匹配失败后，计算模式串向后移动的距离是 KMP 算法中的核心部分。

其实，匹配失败后模式串移动的距离和主串没有关系，只与模式串本身有关系。

例如，我们将前面的模式串 B 改为 "ABCAE"，则在第一次模式匹配失败，由于匹配失败位置模式串中字符 'E' 前面有两个字符 'A'，因此，第二次模式匹配应改为如图 3 所示的位置：

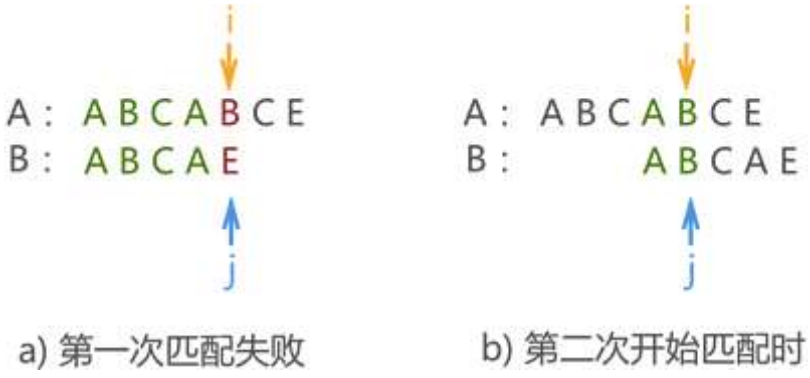

图 3 模式匹配过程示意图

结合图 1、图 2 和图 3 不难看出，模式串移动的距离只和自身有关系，和主串无关。换句话说，不论主串如何变换，只要给定模式串，则匹配失败后移动的距离就已经确定了。

不仅如此，模式串中任何一个字符都可能导致匹配失败，因此串中每个字符都应该对应一个数字用来表示匹配失败后模式串移动的距离。

注意，这里要转换一下思想，模式串向后移动等价于指针 j 前移，如图 4 中的 a) 和 b)。换句话说，模式串后移相当于对指针 j 重定位。

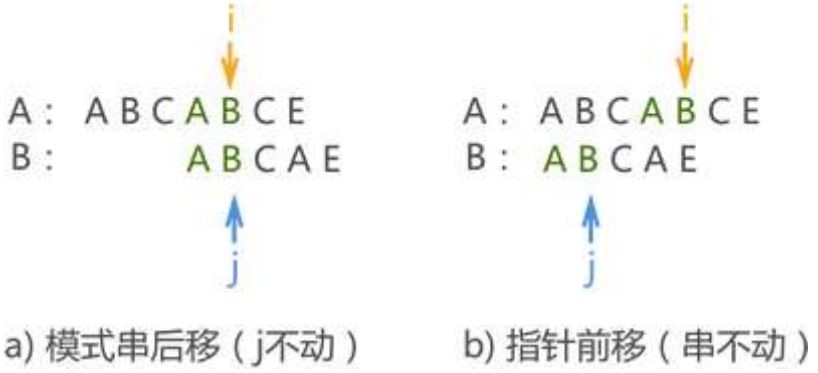

图 4 模式串后移等价于 j 前移

因此，我们可以给每个模式串配备一个数组（例如 next[]），用于存储模式串中每个字符对应指针 j 重定向的位置（也就是存储模式串的数组下标），比如 j=3，则该字符匹配失败后指针 j 指向模式串中第 3 个字符。

模式串中各字符对应 next 值的计算方式是，取该字符前面的字符串（不包含自己），其前缀字符串和后缀字符串相同字符的最大个数再 +1 就是该字符对应的 next 值。

> 前缀字符串指的是位于模式串起始位置的字符串，例如模式串 "ABCD"，则 "A"、"AB"、"ABC" 以及 "ABCD" 都属于前缀字符串；后缀字符串指的是位于串结尾处的字符串，还拿模式串 "ABCD" 来说，"D"、"CD"、"BCD" 和 "ABCD" 为后缀字符串。

注意，模式串中第一个字符对应的值为 0，第二个字符对应 1 ，这是固定不变的（先这么认为）。因此，图 3 的模式串 "ABCAE" 中，各字符对应的 next 值如图 5 所示：

图 5 模式串对应的 next 数组

从图 5 中的数据可以看出，当字符 'E' 匹配失败时，指针 j 指向模式串数组中第 2 个字符，即 'B'，同之前讲解的图 3 不谋而合。

以上所讲 next 数组的实现方式是为了让大家对此数组的功能有一个初步的认识。接下来学习如何用编程的思想实现 next 数组。编程实现 next 数组要解决的主要问题依然是 "如何计算每个字符前面前缀字符串和后缀字符串相同的个数"。

仔细观察图 5，为什么字符 'C' 对应的 next 值为 1？因为字符串 "AB" 前缀字符串和后缀字符串相等个数为 0，0 + 1 = 1。那么，为什么字符 'E' 的 next 值为 2？因为紧挨着该字符之前的 'A' 与模式串开头字符 'A' 相等，1 + 1 = 2。

如果图 5 中模式串为 "ABCABE"，则对应 next 数组应为 [0,1,1,1,2,3]，为什么字符 'E' 的 next 值是 3？因为紧挨着该字符前面的 "AB" 与开头的 "AB" 相等，2 + 1 =3。

因此，我们可以设计这样一个算法，刚开始时令 j 指向模式串中第 1 个字符（j=1），i 指向第 2 个字符（i=2）。接下来，对每个字符做如下操作：

如果 i 和 j 指向的字符相等，则 i 后面第一个字符的 next 值为 j+1，同时 i 和 j 做自加 1 操作，为求下一个字符的 next 值做准备，如图 6 所示：

图 6 i 和 j 指向字符相等

上图中可以看到，字符 'a' 的 next 值为 j +1 = 2，同时 i 和 j 都做了加 1 操作（此时 j=2，i=3）。当计算字符 'C' 的 next 值时，还是判断 i 和 j 指向的字符是否相等，显然相等，因此令该字符串的 next 值为 j + 1 = 3，同时 i 和 j 自加 1（此次 next 值的计算使用了上一次 j 的值）。如图 7 所示：

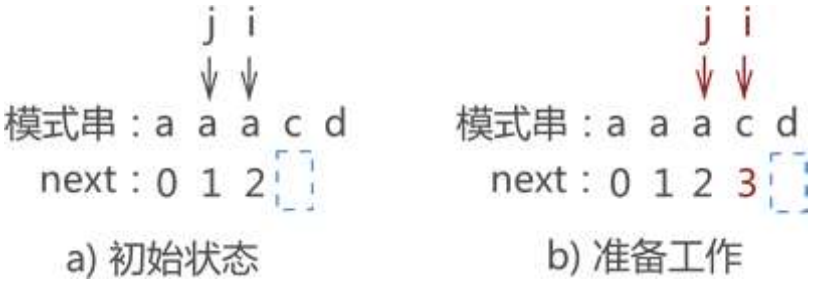

图 7 i 和 j 指向字符仍相等

如上图所示，计算字符 'd' 的 next 时，i 和 j 指向的字符不相等（此时 j=3，i=4），这表明最长的前缀字符串 "aaa" 和后缀字符串 "aac" 不相等，接下来要判断次长的前缀字符串 "aa" 和后缀字符串 "ac" 是否相等，这一步的实现可以用 j = next[j] 来实现（注意，next 数组从下标 1 开始使用，舍弃 next[0] ），如图 8 所示：

图 8 执行 j=next[j] 操作

从上图可以看到，i 和 j 指向的字符又不相同，因此继续做 j = next[j] 的操作，如图 9 所示：

图 9 继续执行 j=next[j] 的操作

此时，由于 j 和 i 指向的字符仍不相等，继续执行 j=next[j] 得到 j=0，这意味着字符 'd' 前的前缀字符串和后缀字符串相同个数为 0，因此如果字符 'd' 导致了模式匹配失败，则模式串移动的距离只能是 1。

这里给出使用上述思想实现 next 数组的 C 语言代码：

```
01.  void Next(char*T,int *next){
02.      next[1]=0;
03.      int i=1;
04.      int j=0;
05.      //next[2]=1 可以通过第一次循环直接得出
06.      while (i<strlen(T)) {
07.          if (j==0||T[i-1]==T[j-1]) {
08.              i++;
09.              j++;
10.              next[i]=j;
11.          }else{
12.              j=next[j];
13.          }
```

```
    }
}
```

代码中 j=next[j] 的运用可以这样理解，每个字符对应的next值都可以表示该字符前 "同后缀字符串相同的前缀字符串最后一个字符所在的位置"，因此在每次匹配失败后，都可以轻松找到次长前缀字符串的最后一个字符与该字符进行比较。

Next函数的缺陷

图 10 Next 函数的缺陷

例如，在图 10a) 中，当匹配失败时，Next 函数会由图 10b) 开始继续进行模式匹配，但是从图中可以看到，这样做是没有必要的，纯属浪费时间。

出现这种多余的操作，问题在当 T[i-1]==T[j-1] 成立时，没有继续对 i++ 和 j++ 后的 T[i-1] 和 T[j-1] 的值做判断。改进后的 Next 函数如下所示：

```
void Next(char*T,int *next){
    next[1]=0;
    int i=1;
    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];
        }
    }
}
```

注意，这里只设定了 next[1] 的值为 0，而 next[1] 的值，需要经过判断之后，才能最终得出，所以它的值不一定是 1。

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，会大大提高效率，如图 11 所示，精简前为 next1，精简后为 next2：

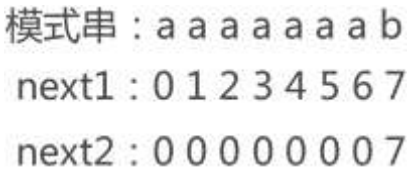

图 11 改进后的 Next 函数

KMP 算法的实现

假设主串 A 为 "ababcabcacbab"，模式串 B 为 "abcac"，则 KMP 算法执行过程为：

- 第一次匹配如图 12 所示，匹配结果失败，指针 j 移动至 next[j] 的位置；

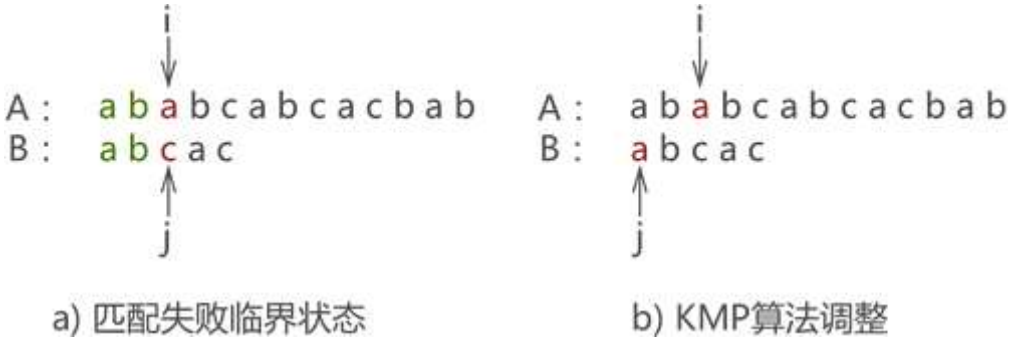

图 12 第一次匹配示意图

- 第二次匹配如图 13 所示，匹配结果失败，依旧执行 j=next[j] 操作：

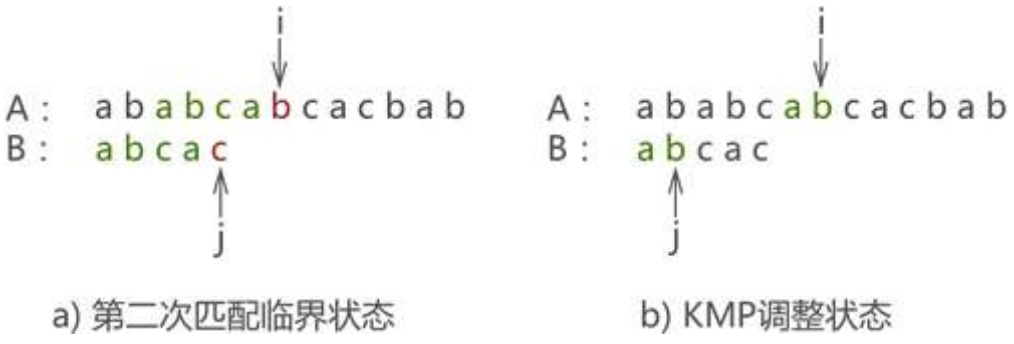

图 13 第二次匹配示意图

- 第三次匹配成功，如图 14 所示：

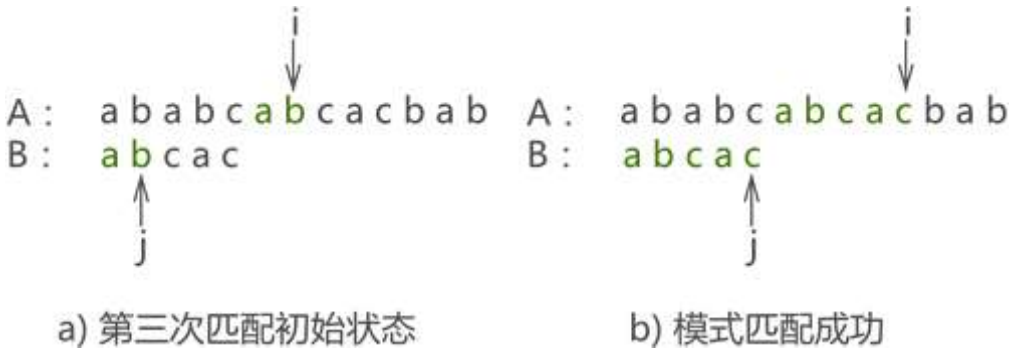

图 14 第三次匹配示意图

很明显，使用 KMP 算法只需匹配 3 次，而同样的问题使用 BF 算法则需匹配 6 次才能完成。

KMP 算法的完整 C 语言实现代码为：

```
01.  #include <stdio.h>
02.  #include <string.h>
03.  //调用了普通求 next 的方式，这里并未直接对 next[1] 赋值为 1，但通过函数第一次运行，也可以得出它的值为 1
04.  void Next(char*T,int *next){
05.      int i=1;
06.      next[1]=0;
07.      int j=0;
08.      while (i<strlen(T)) {
09.          if (j==0||T[i-1]==T[j-1]) {
10.              i++;
11.              j++;
12.              next[i]=j;
13.          }else{
14.              j=next[j];
15.          }
16.      }
17.  }
18.  int KMP(char * S,char * T){
19.      int next[10];
20.      Next(T,next);//根据模式串T,初始化next数组
21.      int i=1;
22.      int j=1;
23.      while (i<=strlen(S)&&j<=strlen(T)) {
24.          //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
25.          if (j==0 || S[i-1]==T[j-1]) {
26.              i++;
27.              j++;
28.          }
29.          else{
30.              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
```

```
31.             }
32.         }
33.         if (j>strlen(T)) {//如果条件为真，说明匹配成功
34.             return i-(int)strlen(T);
35.         }
36.         return -1;
37.     }
38.     int main() {
39.         int i=KMP("ababcabcacbab","abcac");
40.         printf("%d",i);
41.         return 0;
42.     }
```

运行结果为：

```
6
```

KMP 算法优秀文章推荐：

KMP算法推荐表

软文推荐	软文特点
KMP算法详解	此教程对KMP算法中 next 数组的实现做了详细地讲解，其实现代码与本文中有些出路，但两种实现都正确，只是出发点不同。
彻底理解KMP算法	此教程详细介绍了 BF算法和 KMP 算法，如果你能耐下心来读完，那么模式匹配算法肯定能彻底学会。
知乎KMP算法	此页面中存在对 KMP 算法形象地描述，只不过是用 Java 语言实现，但是其理论知识的讲解堪称精彩。
KMP算法入门	这篇软文对 KMP 算法的实现过程进行了最详细地描述，有图有真相，如果你阅读完本文，对 KMP 还是一知半解，可以看这篇文章。
KMP算法	这也是一篇详细介绍 KMP算法的文章，推荐给大家。

< 上一页 下一页 >

所有教程

设计模式 C语言入门 C语言编译器 C语言项目案例 数据结构 C++

STL C++11 socket GCC GDB Makefile OpenCV Qt教程
Unity 3D UE4 游戏引擎 Python Python并发编程 TensorFlow
Django NumPy Linux Shell Java教程 Java Swing Servlet
JSP教程 Struts2 Maven Spring Spring MVC Spring Boot
Spring Cloud Hibernate Mybatis MySQL教程 MySQL函数 NoSQL
Redis MongoDB HBase Go语言 C# MATLAB JavaScript
Bootstrap HTML CSS教程 PHP 汇编语言 TCP/IP vi命令
Android教程 区块链 Docker 大数据 云计算 编程笔记 资源下载
VIP视频 一对一答疑 关于我们

优秀文章

=与==的区别，C语言=与==的区别详解
Linux系统服务及其分类
MySQL INT、TINYINT、SMALLINT、MEDIUMINT、BIGINT（整数类型）
Unity 3D Game View视图
顺序表和链表的优缺点（区别、特点）详解
JSP PageContext.setAttribute()方法：设置属性
Vim配置文件（.vimrc）详解
Java分割字符串（spilt()）
PHP实现倒计时功能
JS获取元素的偏移位置