

# 5-串

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

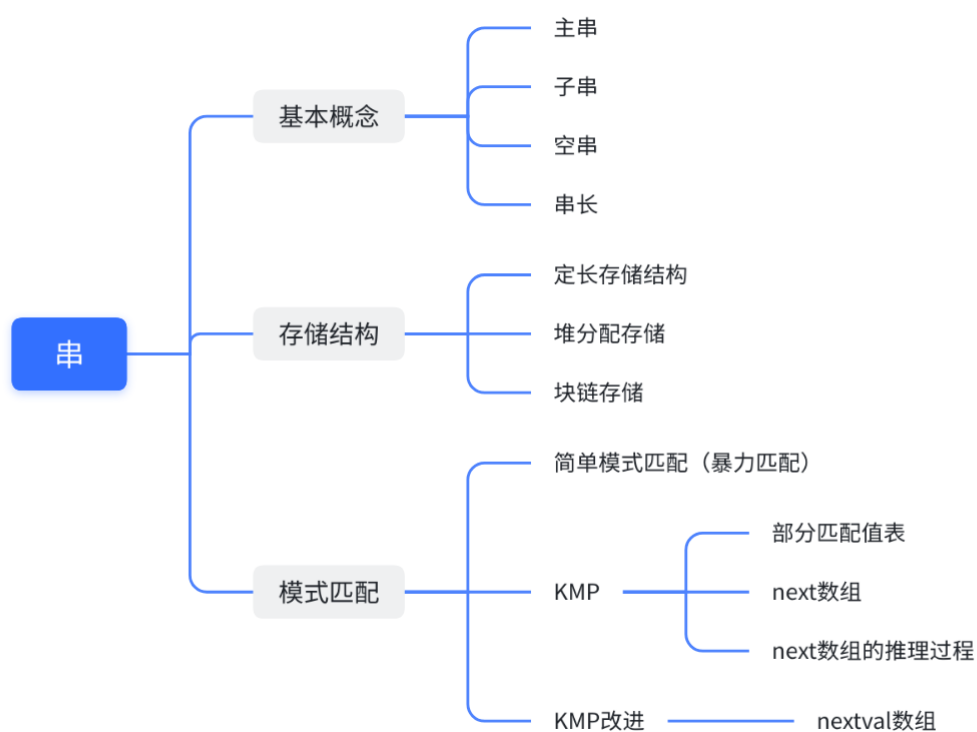
## 1、目标

📌 掌握串的概念- 主串、子串、空串、模式匹配

掌握串的存储

至少达到手动推算KMP算法（难点）

串的应用



## 2、串的概念


📌 串是由零个或多个字符组成的有限序列

串中的字符个数称为串的长度，零个字符称为空串

串中任意个**连续字符**组成的子序列称为该串的子串

包含子串的串就称为主串

### 3、串的存储

 串的操作，其实就是顺序表的操作，存储也跟顺序表一样，使用相同的结构，只是数据类型为char

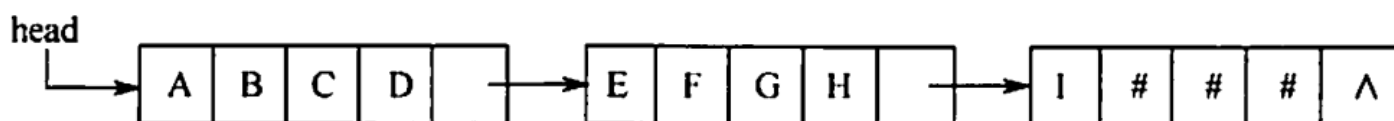
- 定长顺序存储

```
1 typedef struct SString
2 {
3     char ch[N];
4     int length;
5 }SString;
```

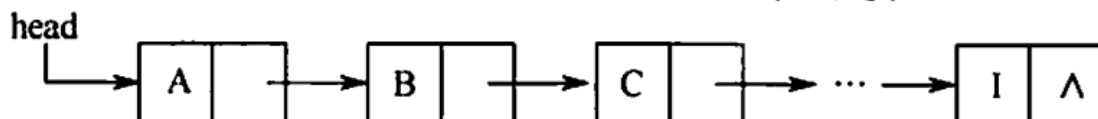
- 动态顺序存储

```
1 typedef struct HString
2 {
3     char *ch;
4     int length;
5 }HString;
```

- 块链存储



(a) 结点大小为4的链表



(b) 结点大小为1的链表

- 串的ADT

```
1 typedef struct HString
2 {
3     char *ch;
4     int length;
```

```
5 }HString;
6
7 void InitString(HString *S);
8 void PrintString(HString *S);
9
10 void StrAssign(HString *S, char *str);
11 void StrCopy(HString *S, HString *T);
12 bool StrEmpty(HString *S);
13 int StrCompare(HString *S, HString *T);
14 int StrLength(HString *S);
15 void StrConcat(HString *T, HString *s1, HString *s2);
16 void SubString(HString *S, HString *sub, int pos, int len);
17 void StrInsert(HString *S, int pos, HString *T);
18 void StrDelete(HString *S, int pos, int len);
19 void StrClear(HString *S);
20 void StrIndex(HString *S, HString *T, int pos);
```

## 4、模式匹配

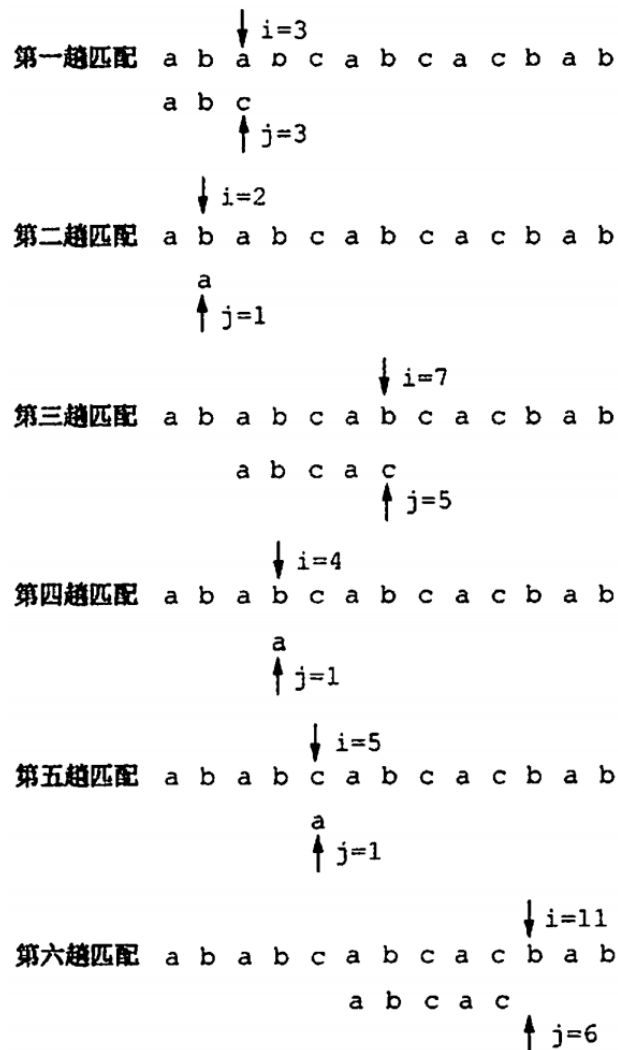


子串的定位称为模式匹配

主串：a b a b c a b c a c b a b

子串：a b c a c

### 1、朴素模式匹配（简单模式匹配）



## • 实现一

```

1 //从主串S的pos位置开始，匹配模式串T，成功返回相应位置，否则返回-1
2 //while循环控制
3 int FindIndex(const char *T, const char *P, int pos)
4 {
5     int i = pos, j = 0;
6     int t_len = strlen(T);
7     int p_len = strlen(P);
8     while(i<t_len && j<p_len)
9     {
10         if(P[j] == T[i])
11         {
12             i++;
13             j++;
14         }
15         else
16         {
17             i = i-j+1;
18             j = 0;
19         }
20     }

```

```

20     }
21     if(j >= p_len) //匹配成功
22         return i - p_len; //返回匹配成功起始位置
23     return -1;      //匹配失败
24 }
25
26 int main()
27 {
28     const char *T = "ababcabcacbab";
29     const char *P = "abcac";
30
31     int pos = FindIndex(T, P, 0);
32     printf("pos = %d\n", pos);
33
34     return 0;
35 }

```


## • 实现二

```


1 //for循环控制
2 int FindIndex(const char *T, const char *P, int pos)
3 {
4     int t_len = strlen(T);
5     int p_len = strlen(P);
6     int i, j;
7     for(i=pos; i<t_len - p_len; ++i)
8     {
9         for(j=0; j<p_len; ++j)
10         {
11             if(P[j] != T[i+j])
12                 break;
13         }
14         if(j >= p_len)
15             return i;
16     }
17     return -1;
18 }
19
20 int main()
21 {
22     const char *T = "ababcabcacbab";
23     const char *P = "abcac";
24
25     int pos = FindIndex(T, P, 0);
26     printf("pos = %d\n", pos);


```


```
27
28     return 0;
29 }
```

 以上两种实现方式，其原理是一样的，只是循环控制的方式有所不同而已

## 2、KMP算法

-  1、首先知道模式匹配需要解决什么问题
- 2、掌握前缀、后缀、部分匹配值(PM)概念
  - 3、会计算PM
  - 4、会计算next[j]
  - 5、会利用next[j]匹配
  - 6、掌握KMP的优化理论
  - 7、针对考研，最低要求达到会手工计算next[]，以及nextval[]

 KMP模式匹配需要解决什么问题: 让主串指针不回退，提高匹配效率

 前缀后缀只要注意一点：就是字符串不能包含完整字符串本身，否则最大的匹配长度就是自身字符串

前缀：除最后一个字符以外，字符串的所有头部子串

后缀：除第一个字符以外，字符串的所有尾部子串

部分匹配值：为字符串的前缀和后缀匹配的最大长度值

以abccac为例：

1. a 的前缀和后缀都为空集，PM=0
2. ab的前缀为{a}, 后缀为{b}, PM=0
3. abc的前缀为{a, ab}, 后缀为{c, bc}, PM=0
4. abca的前缀为{a, ab, abc}, 后缀为{a, ca, bca}, PM=1
5. abccac的前缀为{a, ab, abc, abca}, 后缀为{c, ac, cac, bcac}, PM=0

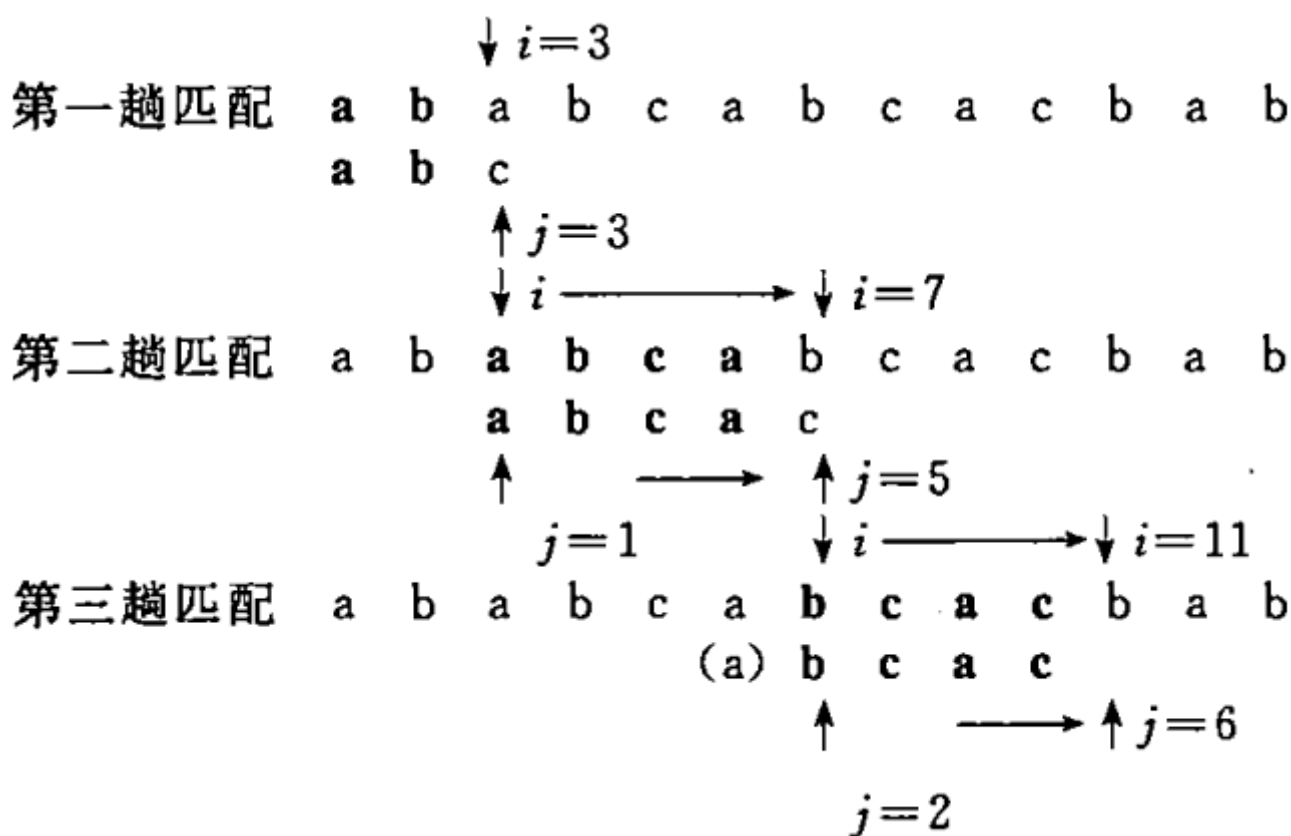
### 📌 next[j]的计算:

将PM表右移一位即可得到next[j]表, 右移时左边补-1, 右边丢弃

j	1	2	3	4	5
S	a	b	c	a	c
PM	0	0	0	1	0

j	1	2	3	4	5
S	a	b	c	a	c
next	-1	0	0	0	1

### 📌 利用next[j]匹配



```
1 void get_next(const char *P, int next[])
2 {
3     int j = 0, k = -1;
4     next[0] = -1;
5
6     int p_len = strlen(P);
7     while(j < p_len-1)
8     {
9         if(k==-1 || P[j]==P[k])
10             next[++j] = ++k;
11         else
12             k = next[k];
13     }
14 }
15 int FastFindIndex(const char *T, const char *P, int pos, int next[])
16 {
17     int i = pos, j = 0;
18     int t_len = strlen(T);
19     int p_len = strlen(P);
20     while(i<t_len && j<p_len)
21     {
22         if(j==-1 || P[j]==T[i])
23         {
24             i++;
25             j++;
26         }
27         else
28             j = next[j];
29     }
30     if(j >= p_len)
31         return i-p_len;
32     return -1;
33 }
34
35 int main()
36 {
37     const char *T = "ababcabcacbab";
38     const char *P = "abcac";
39     int next[5];
40
41     get_next(P, next);
42     int pos = FastFindIndex(T, P, 0, next);
43     printf("pos = %d\n", pos);
```



```
44
45     return 0;
46 }
```

## KMP优化

### 为什么KMP算法需要优化？

举例：

主串s: a a a b a a a b

子串p: a a a a b

next[j] = {-1, 0, 1, 2, 3}

在上述的匹配过程中，i=3, j=3是，b跟a匹配失败，根据next[j],会继续使用p[2],p[1],p[0]跟s[3]进行比较，由于(p[0]==p[1]==p[2]==p[3]==a) != (s[3]==b),所以p[2],p[1],p[0]跟s[3]的匹配必然失败，相当于这样的比价是毫无意义的，因此需要优化。

优化后的nextval[j] = {-1, -1, -1, -1, 3}

### KMP的优化其实就是next[j]的优化，即如何得到nextval[j]数组：

有如下串： aaaaaaab

next[j] = {-1,0,1,2,3,4,5,6,7}

优化后的nextval[j] = {-1, -1, -1, -1, -1, -1, -1, -1, 7}

练习：模式串 t= 'abcaabbcabcaabdab'，该模式串的 next 数组的值为(D)，nextval 数组的值为(F)

- A. 0 1 1 1 2 2 1 1 1 2 3 4 5 6 7 1 2    B. 0 1 1 1 2 1 2 1 1 2 3 4 5 6 1 1 2  
C. 0 1 1 1 0 0 1 3 1 0 1 1 0 0 7 0 1    D. 0 1 1 1 2 2 3 1 1 2 3 4 5 6 7 1 2  
E. 0 1 1 0 0 1 1 1 0 1 1 0 0 1 7 0 1    F. 0 1 1 0 2 1 3 1 0 1 1 0 2 1 7 0 1

```
1 void get_nextval(const char *P, int next[])
2 {
3     int j = 0, k = -1;
4     next[0] = -1;
5
6     int p_len = strlen(P);
7     while(j < p_len-1)
```

```
8      {
9          if(k==-1 || P[j]==P[k])
10         {
11             j++; k++;
12             if(P[k] != P[j])
13                 next[j] = k;
14             else
15                 next[j] = next[k];
16         }
17         else
18             k = next[k];
19     }
20 }
```

### 字符串必会题型：

字符串压缩 <https://leetcode.cn/problems/compress-string-lcci/>

字符串旋转 <https://leetcode.cn/problems/rotate-string/>

字符串相加 <https://leetcode.cn/problems/add-strings/>

字符串相乘 <https://leetcode.cn/problems/multiply-strings/>