





# 本章重点与难点

## ■ 重点：

在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映数值计算的要求，因此，字符串的处理比具体数值处理复杂。本章讨论串的存储结构及几种基本的处理。



# 第四章 串

## 4.1 串类型的定义

### 4.1.1 串的基本概念

### 4.1.2 串的抽象数据类型定义

## 4.2 串的存储表示和实现

### 4.2.1 串的定长顺序存储表示

### 4.2.2 串的堆分配存储表示

### 4.2.3 串的链式存储表示

## 4.3 串的模式匹配算法

### 4.3.1 Brute-Force模式匹配算法

### 4.3.2 模式匹配的一种改进算法



# 第四章 串

## 4.1 串类型的定义

### 4.1.1 串的基本概念

### 4.1.2 串的抽象数据类型定义

## 4.2 串的存储表示和实现

### 4.2.1 串的定长顺序存储表示

### 4.2.2 串的堆分配存储表示

### 4.2.3 串的链式存储表示

## 4.3 串的模式匹配算法

### 4.3.1 Brute-Force模式匹配算法

### 4.3.2 模式匹配的一种改进算法



## 4.1.1 串的基本概念

串是线性表的一种特殊形式，表中每个元素的类型为字符型，是一个有限的字符序列。

串的基本形式可表示成： $S = 'a_1a_2a_3\cdots a_n'$ ；

其中： $\text{char } a_i$ ； $0 \leq i \leq n$ ； $n \geq 0$ ；当  $n = 0$  时，为空串。  
 $n$  为串的长度；

- 空串(空的字符串)：长度为零的串称为空串，它不包含任何字符。
- 空格串(空白串)：构成串的所有字符都是空格的串称为空格串。
  - 注意：空串和空白串的不同，例如 “ ” 和 “ ” 分别表示长度为1的空白串和长度为0的空串。



## 4.1.1 串的基本概念

- **子串(substring)**: 串中任意个连续字符组成的子序列称为该串的子串, 包含子串的串相应地称为主串。
- **子串的序号(位置)**: 将子串在主串中**首次**出现时的该子串的首字符对应在主串中的序号, 称为子串在主串中的序号(或位置)。

✓ 例如, 设有串A和B分别是:

A="shenzhen", B="zhen"

则B是A的子串, A为主串。B在A中出现了两次, 其中首次出现所对应的主串位置是5。因此, 称B在A中的序号为5。

✓ 特别地, 空串是任意串的子串, 任意串是其自身的子串。



## 4.1.1 串的基本概念

- **串相等**：如果两个串的串值相等(相同)，称这两个串相等。换言之，只有当两个串的长度相等，且各个对应位置的字符都相同时才相等。
- 通常在程序中使用的串可分为两种：**串变量和串常量**。
- **串常量和整常数、实常数一样**，在程序中只能被引用但不能改变其值，即只能读不能写。通常串常量是由直接量来表示的，例如，语句错误(“溢出”)中“溢出”是直接量。
- **串变量和其它类型的变量一样**，其值是可以改变。



## 4.1.2 串的抽象数据类型定义

### ● 串与线性表

串的逻辑结构和线性表极为相似，区别在于串的数据对象约束为字符集。然而，串的基本操作和线性表有很大差别。在线性表的基本操作中，大多以“单个元素”作为操作对象，例如在线性表中查找某个元素、求取某个元素、在某个位置上插入一个元素和删除一个元素等；而在串的基本操作中，通常以“串的整体”作为操作对象，例如在串中查找某个子串、求取一个子串、在串的某个位置上插入一个子串以及删除一个子串等。





## 4.1.2 串的抽象数据类型定义

**ADT String{**

数据对象:  $D = \{ a_i | a_i \in \text{CharacterSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

**StrAssign(t, 'chars')**

初始条件: chars是一个字符串常量。

操作结果: 生成一个值为chars的串t。

**StrConcat(s, t)**

初始条件: 串s, t 已存在。

操作结果: 将串t联结到串s后形成新串存放到s中。

**StrCompare (s, t)**

初始条件: 串s, t 已存在。

操作结果: 若 $s > t$ , 返回值 $>0$ ; 若 $s = t$ , 返回值 $=0$ ; 若 $s < t$ , 返回值 $<0$ ;



## 4.1.2 串的抽象数据类型定义

**StrLength(t)**

初始条件：字符串t已存在。

操作结果：返回串t中的元素个数，称为串长。

**SubString (s, pos, len, sub)**

初始条件：串s, 已存在,  $1 \leq \text{pos} \leq \text{StrLength}(s)$  且

$0 \leq \text{len} \leq \text{StrLength}(s) - \text{pos} + 1$ 。

操作结果：用sub返回串s的第pos个字符起长度为len的子串。

.....

**} ADT String**



## 4.1.2 串的抽象数据类型定义

**【例2-15】** 将串T 插在串S 中第 i 个字符之后Insert( S, T, i )。

```
Void Insert( STRING &S, STRING T, int i )
{ STRING t1, t2 ;
  if ( ( i < 0 ) || ( i > LEN( S ) )
    error ‘指定位置不对’ ;
  else
    if ( IsNull( S ) ) S = T ;
    else
      if ( IsNull ( T ) )
        { t1 = Substring ( S, 1, i ) ;
          t2 = Substring ( S, i + 1, LEN( S ) );
          S = StrConcat( t1, StrConcat( T, t2 ) );
        }
  }
```



# 第四章 串

## 4.1 串类型的定义

### 4.1.1 串的基本概念

### 4.1.2 串的抽象数据类型定义

## 4.2 串的存储表示和实现

### 4.2.1 串的定长顺序存储表示

### 4.2.2 串的堆分配存储表示

### 4.2.3 串的链式存储表示

## 4.3 串的模式匹配算法

### 4.3.1 Brute-Force模式匹配算法

### 4.3.2 模式匹配的一种改进算法



## 4.2 串的存储表示和实现

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- ✓ **定长顺序存储表示**：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。
- ✓ **堆分配存储方式**：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- ✓ **块链存储方式**：是一种链式存储结构表示。



## 4.2.1 串的定长顺序存储表示

- 这种存储结构又称为**串的顺序存储结构**。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

```
#define MAX_STRLEN 255
```

```
typedef struct
```

```
{ char str[MAX_STRLEN];
```

```
    int length;
```

```
} StringType ;
```

```
Typedef char str[MAX_STRLEN+1];
```



## 4.2.1 串的定长顺序存储表示

### ① 串的联结操作

**Status StrConcat ( StringType s, StringType t)**

**/\* 将串t联结到串s之后，结果仍然保存在s中 \*/**

**{ int i, j ;**

**if ((s.length+t.length)>MAX\_STRLEN)**

**Return ERROR ; /\* 联结后长度超出范围 \*/**

**for (i=1 ; i<t.length+1 ; i++)**

**s.str[s.length+i]=t.str[i] ; /\* 串t联结到串s之后 \*/**

**s.length=s.length+t.length ; /\* 修改联结后的串长度 \*/**

**return OK ;**

**}**



## 4.2.1 串的定长顺序存储表示

### ② 求子串操作

```
Status SubString (StringType s, int pos, int len,
StringType *sub)//求在s中以位置pos起长度为len的子串
{ int k, j ;
  if (pos<1||pos>s.length||len<0||len>(s.length-pos+1))
    return ERROR ; /* 参数非法 */
  sub->length=len ; /* 求得子串长度 */
  for (j=0, k=pos ; k<=len+pos-1 ; k++, j++)
    sub->str[j]=s.str[k-1] ; /* 逐个字符复制求得子串 */
  return OK ;
}
```





## 4.2.2 串的堆分配存储表示

- **实现方法：**系统提供一个空间足够大且地址连续的存储空间供串使用。可使用C语言的动态存储分配函数`malloc()`和`free()`来管理。为每个新产生的串分配一个存储区，称串值共享的存储空间为“堆”。C语言中的串以空字符‘\0’为结束符，串长是一个隐含值。
- **特点：**仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef struct
```

```
{ char *ch; /* 若非空，按长度分配，否则为NULL */  
    int length; /* 串的长度 */  
} HString ;
```



## 4.2.2 串的堆分配存储表示

### ① 串的联结操作

**Status StrConcat(HString \*T, HString \*s1, HString \*s2)**

**/\* 用T返回由s1和s2联结而成的串 \*/**

```
{ int k, j, t_len ;  
  if (T.ch) free(T); /* 释放旧空间 */  
  t_len=s1->length+s2->length ;  
  if ((T=(char *)malloc(sizeof((char)*t_len))==NULL)  
    { printf(“系统空间不够, 申请空间失败 ! \n”);  
      return ERROR ; }  
  for (j=0 ; j<s->length; j++)  
    T->ch[j]=s1->ch[j] ; /* 将串s复制到串T中 */  
  for (k=s1->length, j=0 ; j<s2->length; k++, j++)  
    T->ch[j]=s2->ch[j] ; /* 将串s2复制到串T中 */  
  free(s1->ch) ; free(s2->ch) ;  
  return OK ;  
}
```



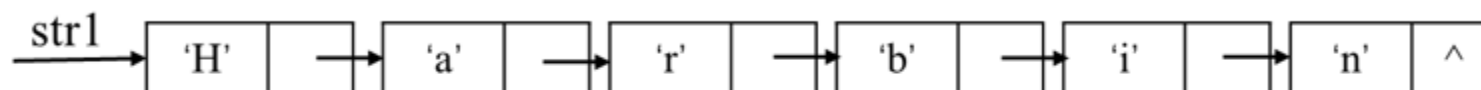
## 4.2.3 串的链式存储表示

- 串的链式存储结构和线性表的链式存储结构类似，采用**单链表来存储串**，结点的构成是：

◆ **data域**：存放字符

◆ **next域**：存放指向下一结点的指针。

```
struct node {  
    char data;  
    node *next;  假设地址指针next占用4个字节  
};  
typedef struct node *STRING1;  
STRING1 str1;
```



分配空间利用率为1/5（20%）



## 4.2.3 串的链式存储表示

串的**块链式存储的类型定义**包括：

(1) 块结点的类型定义

```
#define BLOCK_SIZE 4
```

```
Typedef struct node
```

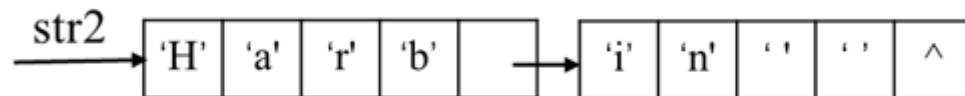
```
{ char data[BLOCK_SIZE] ;
```

```
    struct node *next;
```

```
}Bnode ;
```

```
typedef Bnode *STRING2 ;
```

```
STRING2 str2 ;
```



分配空间利用率约为1/2 (50%)

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结



## 4.2.3 串的链式存储表示

### (2) 块链串的类型定义

**typedef struct**

```
{ Bnode *head, *tail;    /* 头指针和尾指针 */  
    int Strlen;          /* 当前长度 */  
} Blstring;
```

**特点：** 连接操作简单（注意处理串尾的无效字符）

须考虑存储密度

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。



## 4.2.3 串的链式存储表示

### C 语言中的串

在C语言中，可以用两种方法表示和存放字符串：

(1) 用字符数组存放一个字符串

```
char str[10]={ 'I',' ','a','m',' ','h','a','p','p','y'};
```

可以用字符串常量来初始化字符数组（不能赋值）：

```
char str[ ]="I love China";
```

(2) 用字符指针指向一个字符串（可以赋值）

```
char* str="I love China";
```



# 第四章 串

## 4.1 串类型的定义

### 4.1.1 串的基本概念

### 4.1.2 串的抽象数据类型定义

## 4.2 串的存储表示和实现

### 4.2.1 串的定长顺序存储表示

### 4.2.2 串的堆分配存储表示

### 4.2.3 串的链式存储表示

## 4.3 串的模式匹配算法

### 4.3.1 Brute-Force模式匹配算法

### 4.3.2 模式匹配的一种改进算法



## 4.3 串的模式匹配算法

- **模式匹配 (模范匹配)**: 子串在主串中的定位称为模式匹配或串匹配 (字符串匹配)。模式匹配成功是指在主串  $S$  中能够找到模式串  $T$ , 否则, 称模式串  $T$  在主串  $S$  中不存在。
- 设  $S$  为目标串 (或主串),  $T$  为模式串, 且不妨设:

$$S = "s_1 s_2 \dots s_n", \quad T = "t_1 t_2 \dots t_m"$$

在  $S$  中寻找  $T$  的过程称为模式匹配。如果匹配成功, 返回  $T$  在  $S$  中的位置; 如果匹配失败, 返回 -1。

**模式匹配的应用非常广泛。**

**模式匹配是一个较为复杂的串操作过程。** 迄今为止, 人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。介绍两种主要的模式匹配算法。





## 4.3.1 串的模式匹配算法

使用基本操作来实现模式匹配

设主串S=“ababcabcacbab”，模式串T=“abcac”

```
int Index(SString S, SString T){  
    int i=1, n=StrLength(S), m=StrLength(T);  
    SString sub;    //用于暂存子串  
    while(i<=n-m+1){  
        SubString(sub, S, i, m);  
        if(StrCompare(sub, T)!=0) ++i;  
        else return i;    //返回子串在主串中的位置  
    }  
    return 0;    //S中不存在与T相等的子串  
}
```



### 4.3.1 Brute-Force模式匹配算法

- 朴素模式匹配算法(Brute-Force算法)：枚举法
- 基本思想：
  - 从主串S的第一个字符开始和模式T的第一个字符进行比较，若相等，则继续比较两者的后续字符；否则，从主串S的第二个字符开始和模式T的第一个字符进行比较，重复上述过程，直到T中的字符全部比较完毕，则说明本趟匹配成功；或S中字符全部比较完，则说明匹配失败。



### 4.3.1 Brute-Force模式匹配算法

设主串S=“ababcabcacbab”，模式串T=“abcac”

第1趟匹配	主串	ab <b>a</b> bcabcacbab	i=3	
	模式串	ab <b>c</b> ac	j=3	匹配失败
第2趟匹配	主串	a <b>b</b> abcabcacbab	i=2	
	模式串	<b>a</b> bcac	j=1	匹配失败
第3趟匹配	主串	ababca <b>b</b> cacbab	i=7	
	模式串	abc <b>a</b> c	j=5	匹配失败
第4趟匹配	主串	abab <b>c</b> abcacbab	i=4	
	模式串	<b>a</b> bcac	j=1	匹配失败
第5趟匹配	主串	abab <b>c</b> abcacbab	i=5	
	模式串	<b>a</b> bcac	j=1	匹配失败
第6趟匹配	主串	ababc <b>a</b> bcacbab	i=11	//返回i- T.length
	模式串	<b>a</b> bcac	j=6	匹配成功

- 特点:主串指针需回溯( $i - j + 2$ )，模式串指针需复位(1)。



## 4.3.1 Brute-Force模式匹配算法

- **BF算法实现的详细步骤:**
  - 1. 在串S和串T中设比较的起始下标i和j;
  - 2. 循环直到S或T的所有字符均比较完;
    - 2.1 如果 $S[i]=T[j]$ , 继续比较S和T的下一个字符;
    - 2.2 否则, 将i回溯( $i=i-j+2$ ), j复位, 准备下一趟比较;
  - 3. 如果T中所有字符均比较完, 则匹配成功, 返回主串匹配的**起始比较下标**; 否则, 匹配失败, 返回-1;



### 4.3.1 Brute-Force模式匹配算法

```
int Index_BF ( char* S, char* T, int pos=1)
{ /* S为主串，T为模式，串采用顺序存储结构 */

    i = pos;    j = 1;                // 从第一个位置开始比较
    while (i<=S.length && j<=T.length) {
        if (S[i] == T[j]) {++i; ++j;} // 继续比较后继字符
        else {i = i - j + 2;  j = 1;} // 指针后退重新开始匹配
    }
    if (j > T.length) return i - T.length; // 返回与模式第一字符相等
    else                                     // 的字符在主串中的序号
        return 0;                          // 匹配不成功
}
```



## 4.3.1 Brute-Force模式匹配算法

### Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0 ~ n-m)。

①最好的情况下，模式串的第0个字符失配

设匹配成功在S的第i个字符，则在前i趟匹配中共比较了i次，第i趟成功匹配共比较了m次，总共比较了(i+m)次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i(i+m) = \frac{1}{n-m+1} \sum_{i=0}^{n-m} (i+m) = \frac{1}{2}(n+m)$$

最好情况下算法的平均时间复杂度O(n+m)。



## 4.3.1 Brute-Force模式匹配算法

### Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0 ~ n-m)。

②最坏的情况下，模式串的最后1个字符失配

设匹配成功在S的第i个字符，则在前i趟匹配中共比较了*i*\*m次，第i趟成功匹配共比较了m次，总共比较了(*i*+1)\*m次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i (i+1)m = \frac{m}{n-m+1} \sum_{i=0}^{n-m} (i+1) = \frac{1}{2}m(n-m+2)$$

最坏情况下的平均时间复杂度为O(n\*m)。



## 4.3.2 模式匹配的一种改进算法

- 为什么BF算法时间性能低？
  - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。
- KMP算法----改进的模式匹配算法：
- 如何在匹配不成功时主串不回溯？
  - 主串不回溯，模式串就需要向右滑动一段距离。
- 如何确定模式的滑动距离？
  - 利用已经得到的“部分匹配”的结果。
  - 将模式向右“滑动”尽可能远的一段距离(next[j])后，继续进行比较。





## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2...S_n$ ，模式串为 $T_1T_2...T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

g	o	o	g	l	o	g	o	o	g	l	e	g	o	o	g	l	o
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

g	o	o	g	l	e
1	2	3	4	5	6

若  $j=6$  时发生不匹配，则应让  $j$  回到 1  
也可以理解为模式串往右移动了5步



## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2...S_n$ ，模式串为 $T_1T_2...T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

X	X	X	X	X	X	g	o	o	g	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

若  $j=5$  时发生不匹配, 则应让  $j$  回到 2  
也可以理解为模式串往右移动了3步

g	o	o	g	l	e
1	2	3	4	5	6



## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2...S_n$ ，模式串为 $T_1T_2...T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

X	X	X	X	X	X	g	o	o	?	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

若  $j=4$  时发生不匹配, 则应让  $j$  回到 1  
也可以理解为模式串往右移动了3步

g	o	o	g	l	e
1	2	3	4	5	6



## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2...S_n$ ，模式串为 $T_1T_2...T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

	X	X	X	X	X	g	o	?	?	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

若  $j=3$  时发生不匹配, 则应让  $j$  回到 1  
也可以理解为模式串往右移动了2步

g	o	o	g	l	e
1	2	3	4	5	6



## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2...S_n$ ，模式串为 $T_1T_2...T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

X	X	X	X	X	X	g	?	?	?	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

若  $j=2$  时发生不匹配, 则应让  $j$  回到 1  
也可以理解为模式串往右移动了1步

g	o	o	g	l	e
1	2	3	4	5	6



## 4.3.2 模式匹配的一种改进算法

- 思考的开始：
  - 假定主串为 $S_1S_2\dots S_n$ ，模式串为 $T_1T_2\dots T_m$
  - 无回溯匹配问题变为：当主串中的第 $i$ 个字符和模式串中的第 $j$ 个字符出现不匹配，主串中的第 $i$ 个字符应该与模式串中的哪个字符匹配（无回溯）？

X	X	X	X	X	o	?	?	?	?	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

若  $j=1$  时发生不匹配, 则应让  $i$  后移一步,  $j$  仍然是 1

g	o	o	g	l	e
1	2	3	4	5	6



## 4.3.2 模式匹配的一种改进算法

如果  $j=k$  时才发现匹配失败, 说明  $1\sim k-1$  都匹配成功。

g	o	o	g	l	e
1	2	3	4	5	6

↑  
j

int next[7];

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

若当前两个字符匹配, 则  $i++$ ,  $j++$

若  $j=1$  时发生不匹配, 则应让  $i++$ ,  $j$  依然是 1

若  $j=2$  时发生不匹配, 则应让  $j$  回到 1

若  $j=3$  时发生不匹配, 则应让  $j$  回到 1

若  $j=4$  时发生不匹配, 则应让  $j$  回到 1

若  $j=5$  时发生不匹配, 则应让  $j$  回到 2

若  $j=6$  时发生不匹配, 则应让  $j$  回到 1

模式串向右滑动距离可由模式串本身决定。  
当  $j=k$  且发现字符不匹配时, 令  $j=\text{next}[k]$



## 4.3.2 模式匹配的一种改进算法

- 假设主串ababcabcacbab,模式abcac(01112),改进算法的匹配过程如下:

第1趟匹配

↓ i=3

a	b	a	b	c	a	b	c	a	c	b	a	b
a	b	c										

↑ j=3

第2趟匹配

↓ i=3---7

a	b	a	b	c	a	b	c	a	c	b	a	b
		a	b	c	a	c						

↑ j=1

第3趟匹配

↓ i=7---11

a	b	a	b	c	a	b	c	a	c	b	a	b
					a	b	c	a	c			

↑ j=2---6





## 4.3.2 模式匹配的一种改进算法

### ● KMP算法实现步骤:

- 1. 在串S和串T中分别设比较的起始下标i和j;
- 2. 循环直到S中所剩字符长度小于T的长度或T中所有字符均比较完毕
  - 2.1 如果 $S[i]=T[j]$ , 继续比较S和T的下一个字符; 否则
  - 2.2 将j向右滑动到 $next[j]$ 位置, 即 $j=next[j]$ ;
  - 2.3 如果 $j=0$ , 则将i和j分别加1, 准备下一趟比较;
- 3. 如果T中所有字符均比较完毕, 则返回匹配的起始下标; 否则返回0;



## 4.3.2 模式匹配的一种改进算法

```
int Index_KMP ( char* S, char* T, int next[])
```

```
{ /* S为主串，T为模式，串采用顺序存储结构 */
```

```
    int i = 1;
```

```
    int j = 1;    // 从第一个位置开始比较
```

```
    while (i<=S.length && j<=T.length) {
```

```
        if (j==0 || S[i] == T[j]) {++i; ++j;} // 继续比较后继字符
```

```
        else { j=next[j]; } // 指针后退重新开始匹配
```

```
    }
```

```
    if (j > T.length) return i- T.length;    // 返回与模式第一字符相等
```

```
    else                                     // 的字符在主串中的序号
```

```
        return 0;                          // 匹配不成功
```

g	o	o	g	l	e
1	2	3	4	5	6

↑  
j

```
int next[7];
```

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



## 4.3.2 模式匹配的一种改进算法

- **KMP算法**----改进的模式匹配算法：
  - 如何在匹配不成功时主串不回溯？
    - 主串不回溯，模式串就需要向右滑动一段距离。
  - 如何确定模式的滑动距离？
    - 利用已经得到的“**部分匹配**”的结果。
    - 将模式向右“滑动”尽可能远的一段距离( $\text{next}[j]$ )后，继续进行比较。

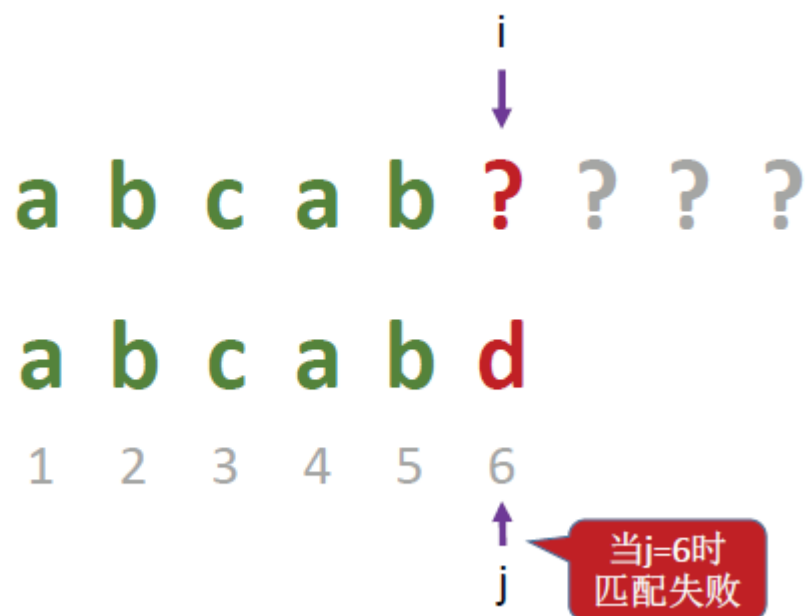


## 4.3.2 模式匹配的一种改进算法

### 求模式串的next数组

next 数组：当模式串的第j 个字符匹配失败时，令模式串跳到 next[j] 再继续匹配

模式串：'abcabd'



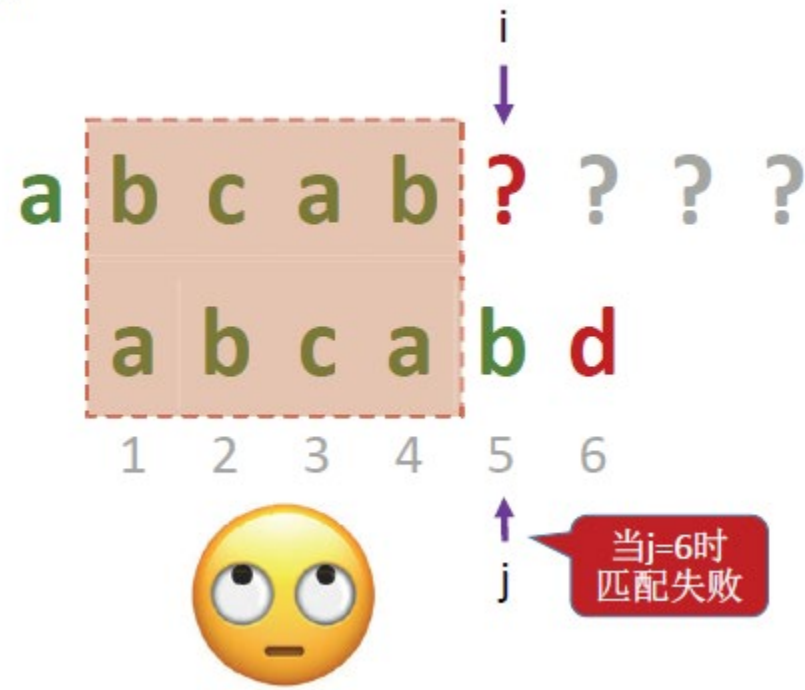


## 4.3.2 模式匹配的一种改进算法

### 求模式串的next数组

next 数组：当模式串的第j 个字符匹配失败时，令模式串跳到 next[j] 再继续匹配

模式串：'abcbabd'



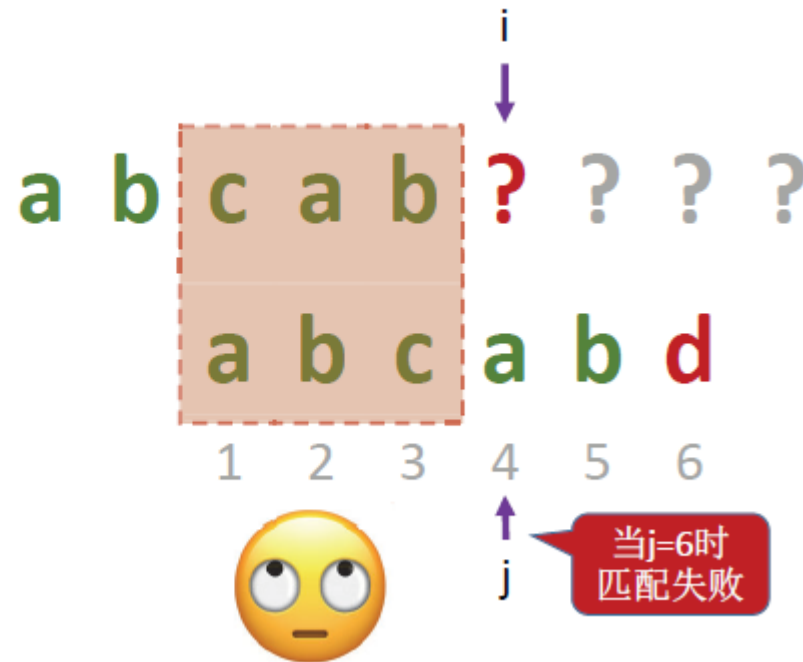


## 4.3.2 模式匹配的一种改进算法

### 求模式串的next数组

next 数组：当模式串的第j 个字符匹配失败时，令模式串跳到 next[j] 再继续匹配

模式串：'abcbabd'



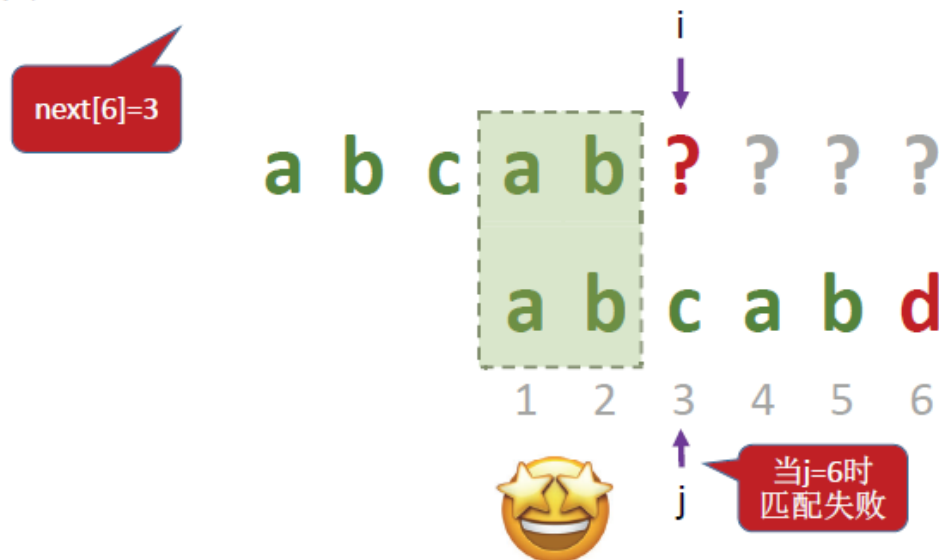


## 4.3.2 模式匹配的一种改进算法

### 求模式串的next数组

next 数组：当模式串的第j 个字符匹配失败时，令模式串跳到 next[j] 再继续匹配

模式串：'abcbabd'



串的**前缀**：包含第一个字符，且不包含最后一个字符的子串

串的**后缀**：包含最后一个字符，且不包含第一个字符的子串

- $k = \text{next}[j]$  实质是找  $T_1 T_2 \dots T_{j-1}$  中的最长相同的**前缀** ( $T_1 T_2 \dots T_{k-1}$ ) 和**后缀** ( $T_{j-(k-1)} T_{j-(k-2)} \dots T_{j-1}$ )。
- 仍是一个模式匹配的过程,只是主串和模式串在同一个串T中



## 4.3.2 模式匹配的一种改进算法

$$\text{next}[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \quad // \text{不比较} \\ \max \{ k \mid 1 < k < j \text{ 且 } T_1 \dots T_{k-1} = T_{j-(k-1)} \dots T_{j-1} \} & \\ 1 & \text{其他情况} \end{cases}$$

- 计算 $\text{next}[j]$ 的方法:
- 当 $j=1$ 时,  $\text{next}[j]=0$ ;
  - $//\text{next}[j]=0$ 表示根本不进行字符比较
- 当 $j>1$  时,  $\text{next}[j]$ 的值为: 模式串的位置从1到 $j-1$ 构成的串中所出现的最长相等的**前后缀长度**加1。
- 当无首尾相同的子串时 $\text{next}[j]$ 的值为1。
  - $//\text{next}[j]=1$ 表示从模式串头部开始进行字符比较





## 4.3.2 模式匹配的一种改进算法

- 计算 $\text{next}[j]$ 的方法:

序号j	1	2	3	4	5	6
模式串	a	b	a	b	a	a
$\text{next}[j]$						

- 当 $j=1$ 时,  $\text{next}[j]=0$ ;
- 当 $j>1$  时,  $\text{next}[j]$ 的值为: 模式串的位置从1到 $j-1$ 构成的串中所出现的最长相等的**前后缀长度**加1。

$\text{next}[j] =$  [填空1]

序号j	1	2	3	4	5
模式串	a	a	a	a	b
next[j]					

正常使用填空题需3.0以上版本雨课堂

$\text{next}[j] =$  [填空1]

j	1	2	3	4	5	6	7	8
模式串	b	a	b	b	a	b	a	b
Next[j]								

正常使用填空题需3.0以上版本雨课堂



## 4.3.2 模式匹配的一种改进算法

- 进一步的改进算法:

序号j	1	2	3	4	5
模式串	a	a	a	a	b
next[j]	0	1	2	3	4
Nextval[j]	0	0	0	0	4



## 4.3.2 模式匹配的一种改进算法

- 朴素模式匹配算法的缺点：当某些子串与模式串能部分匹配时，主串的扫描指针*i*经常回溯，导致时间开销增加。最坏时间复杂度  $O(nm)$
- KMP算法：当子串和模式串不匹配时，主串指针*i*不回溯，模式串指针  $j = \text{next}[j]$ ，算法平均时间复杂度：  $O(n+m)$
- 如果不会经常出现子串与模式串部分匹配问题，那么KMP算法也没厉害多少

哼 不过如此嘛





# 本章小结

- ✓ 熟练掌握：
  - 串的基本概念
  - 串的存储表示和实现
  - 串的模式匹配算法