第4章 串

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(字符串)：是零个或多个字符组成的有限序列。记作： S="$a_1a_2a_3$…"，其中S是串名，$a_i(1 \leqq i \leqq n)$是单个字符，可以是字母、数字或其它字符。
- 串值：双引号括起来的字符序列是串值。
- 串长：串中所包含的字符个数称为该串的长度。
- 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白串。
  - 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：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=“shenzhenzhen”， 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

ADT String{

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

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

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。

4.1.2 串的抽象数据类型定义

操作结果：将串t联结到串s后形成新串存放到s中。

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≦pos≦StrLength(s)且
0≦len≦StrLength(s) –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

● 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。

4.2 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  256
typedef  struct
    {  char  str[MAX_STRLEN] ;
        int  length;
    } StringType ;
```

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=0 ; i<t.length ; 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  Hstring  *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 ((p=(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串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，data域可存放的字符个数称为结点的大小；
  - ◆ **next域**：存放指向下一结点的指针。

若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。如图4-1是块大小为3的串的块链式存储结构示意图。

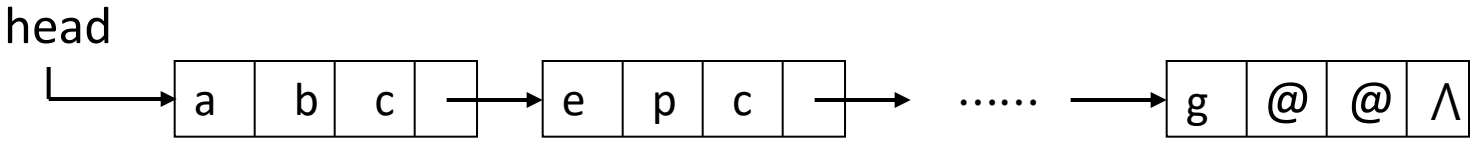


图4-1 串的块链式存储结构示意图

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  Blstrtype
    {  char  data[BLOCK_SIZE] ;
        struct  Blstrtype  *next;
    }BNODE ;
```

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 ;
```

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。

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。

4.3 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。
- 模式匹配的应用非常广泛。很多软件若有“编辑”菜单项的话，则其中必有“查找”子菜单项。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。
- 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 Brute-Force模式匹配算法

- 设S为目标串（或主串），T为模式串，且不妨设：

  S="$s_0s_1s_2\ldots s_{n-1}$"， T="$t_0t_1t_2\ldots t_{m-1}$"

- 串的匹配实际上是对合法的位置$0 \leqslant i \leqslant n-m$依次将目标串中的子串s[i…i+m-1]和模式串t[0…m-1]进行比较：

  ◆ 若s[i…i+m-1]=t[0…m-1]：则称从位置i开始的匹配成功，亦称模式t在目标s中出现；

  ◆ 若s[i…i+m-1]≠t[0…m-1]：从i开始的匹配失败。位置i称为位移，当s[i…i+m-1]=t[0…m-1]时，i称为有效位移；当s[i…i+m-1] ≠t[0…m-1]时，i称为无效位移。

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	ababcabcacbab	i=3	
	模式串	abc	j=3	匹配失败
第2趟匹配	主串	ababcabcacbab	i=2	
	模式串	abc	j=1	匹配失败
第3趟匹配	主串	ababcabcacbab	i=7	
	模式串	abcac	j=5	匹配失败
第4趟匹配	主串	ababcabcacbab	i=4	
	模式串	abc	j=1	匹配失败
第5趟匹配	主串	ababcabcacbab	i=5	
	模式串	abc	j=1	匹配失败
第6趟匹配	主串	ababcabcacbab	i=6	
	模式串	abcac	j=6	匹配成功

- 特点:主串指针需回朔，模式串指针需复位。

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和j回溯，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，返回匹配的起始比较下标；否则，匹配失败，返回0；

4.3.1 Brute-Force模式匹配算法

- 算法实现1

```
int IndexString(StringType  s , StringType  t , int pos )
  /*  采用顺序存储方式存储主串s和模式t，    */
  /*  若模式t在主串s中从第pos位置开始有匹配的子串，*/
  /*  返回位置，否则返回-1  */
   {  char *p , *q ;
      int  k , j ;
      k=pos-1 ;   j=0 ; p=s.str+pos-1 ; q=t.str ;
         /*  初始匹配位置设置  */
          /*  顺序存放时第pos位置的下标值为pos-1 */
```

4.3.1 Brute-Force模式匹配算法

```
while (k<s.length)&&(j<t.length)
   {  if (*p==*q)   { p++ ; q++ ; k++ ; j++ ; }
      else { k=k-j+1 ; j=0 ; q=t.str ; p=s.str+k ;  }
      /*  重新设置匹配位置    */
      }
if (j==t.length)
    return(k-t.length) ;   / *   匹配，返回位置   */
else return(-1) ;      /*   不匹配，返回-1   */
}
```

4.3.1 Brute-Force模式匹配算法

- 算法实现2

```
int Index_BF ( char* S, char* T, int pos=1)
{  /* S为主串，T为模式，串的第0位置存放串长度；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[0] && j<=T[0]) {
        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[0]) return i-T[0];          // 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算法的时间复杂性**

主串S长n,模式串T长m。可能匹配成功的位置（1~n-m+1)。

（1）最好的情况下，模式串的第1个字符失配

设匹配成功在S的第i个字符，则在前i-1趟匹配中共比较了i-1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：

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

最好情况下算法的平均时间复杂性O(n+m)。

4.3.1 Brute-Force模式匹配算法

- **Brute-Force算法的时间复杂性**

主串S长n,模式串T长m。可能匹配成功的位置（1~n-m+1)。

（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-1趟匹配中共比较了(i-1)*m次，第i趟成功匹配共比较了m次，总共比较了(i*m）次。共需要n-m+1趟比较，所以在等概率情况下的平均比较次 数：

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

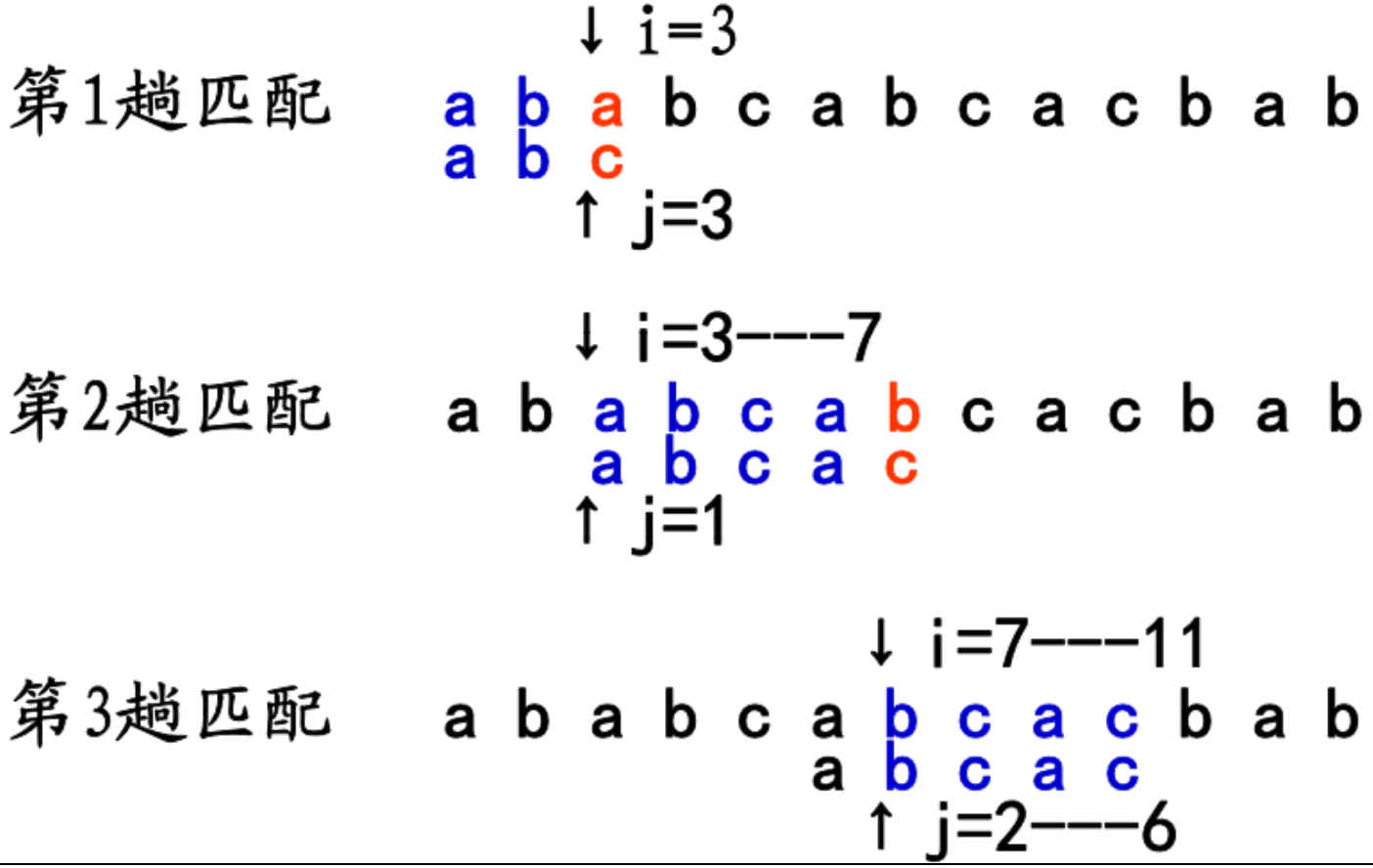
设n>>m，最坏情况下的平均时间复杂性为O(n*m)。

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：**
  - 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。
  - 如何在匹配不成功时主串不回溯？
    - 主串不回溯，模式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

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



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）？
- 进一步思考：
  - 假定主串中第i个字符与模式串第k个字符相比较，则应有
    - $T_1T_2…T_{k-1}= S_{i-(k-1)}S_{i-(k-2)}…S_{i-1}$
    - 问题可能有多个k，取哪一个？
  - 而根据已有的匹配，有
    - $T_{j-(k-1)}T_{j-(k-2)}…T_{j-1}= S_{i-(k-1)}S_{i-(k-2)}…S_{i-1}$
  - 因此：
    - $T_1T_2…T_{k-1} = T_{j-(k-1)}T_{j-(k-2)}…T_{j-1}$

4.3.2 模式匹配的一种改进算法

- $T_1 T_2 \cdots T_{k-1} = T_{j-(k-1)} T_{j-(k-2)} \cdots T_{j-1}$说明了什么？
  - k与j具有函数关系，由当前失配位置j，可以计算出滑 动位置k（即比较的新起点）；
  - 滑动位置k仅与模式串自身T有关，而与主串无关。
- $T_1 T_2 \cdots T_{k-1} = T_{j-(k-1)} T_{j-(k-2)} \cdots T_{j-1}$的物理意义是什么？

第1位往右数k-1位

第j-1位往左数k-1位

- 模式应该向右滑多远才是最高效率的?

$$k = \max\{\, k \mid 1<k<j \text{ 且} T_1 T_2 \ldots T_{k-1} = T_{j-(k-1)}\ T_{j-(k-2)} \ldots T_{j-1}\}$$

4.3.2 模式匹配的一种改进算法

- 令k = next[j]，则：

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

- k = next[j]实质是找$T_1T_2\cdots T_{j-1}$中的最长相同的前缀（$T_1T_2\cdots T_{k-1}$）和后缀（$T_{j-(k-1)}T_{j-(k-2)}\cdots T_{j-1}$）。
- 模式中相似部分越多，则next[j]函数越大
  - 表示模式T 字符之间的相关度越高，
  - 模式串向右滑动得越远，
  - 与主串进行比较的次数越少，时间复杂度就越低。
- 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

- **计算next[j]的方法：**
- **当j=1时，next[j]=0；**
  - **//next[j]=0表示根本不进行字符比较**
- **当j>1 时，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首尾相同的子串的最大长度加1。**
- **当无首尾相同的子串时next[j]的值为1。**
  - **//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

- 函数k=next[j]的实现：

```
void GetNext(char* t, int
Next[])
{
    int j= 1,k= 0; Next[1]= 0;
    while ( j <  t[0] )
        if (k==0||T[j]==T[k]){
            j++;  k++;   Next[j]= k;
        }
        else  k=Next[k];
}
```

4.3.2 模式匹配的一种改进算法

- 计算next[j]的方法：

模式串 T：	a	b	a	a	b	c	a	c
可能失配位 j：	1	2	3	4	5	6	7	8
新匹配位k=next[j]：	0	1	1	2	2	3	1	2

j=1时, next[j] = 0;

j=2时, next[j] = 1;

j=3时, $T_1 \neq T_2$， 因此， k=1;

j=4时, $T_1 = T_3$， 因此， k=2;

j=5时, $T_1 = T_4$， 因此， k=2;

以此类推。

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

- **KMP算法的实现：**

```
int Index_KMP(char* S, char* T, int pos=1)
{ /*S为主串T为模式，串的第0位置存放串长度；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[0] && j<=T[0]) {
     if (S[i] == T[j]) {++i; ++j;}            //继续比较后继字符
     else j = Next[j];                        // 模式串向右移
   }
  if (j > T[0]) return i-T[0];                // 返回与模式第一字符相等
   else return 0;     // 匹配不成功           // 的字符在主串中的序号
}
```

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