第四章　串

- **4.1 串的基本概念**
- **4.2 串的存储结构**
  - 顺序存储结构
  - 链式存储结构
- **4.3 串的操作算法**

串和线性表的区别:

1. 串的数据对象约束为字符集。

2. 线性表的基本操作大多以“单个元素”为操作对象，而串的基本操作通常以“串的整体”作为操作对象。

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 串的基本概念

- 串：是由零个或多个字符组成的有限序列，一般记为

$$s = \text{" } a_1a_2\ldots a_n \text{ "} \quad (n\geq 0)$$

其中：串中字符的个数n称为串的长度。

长度为0的串称为空串。

注意区分空串与空格串的区别。

- 串的相关概念：
  - 子串：串中任意个连续的字符组成的子序列。
  - 主串：包含子串的串相应地称为主串。
  - 位置：字符在序列中的序号。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。
  - 串相等：仅当两个串的长度相等，并且对应位置的字符都相等。

对于串可以定义以下运算：

1. 串赋值：StrAssign(&T，chars);
2. 串比较：StrCompare(S，T);
3. 求串长：StrLength(S);
4. 串联接：Concat(&T，S1，S2);
5. 求子串：SubString(&Sub，S，pos，len)

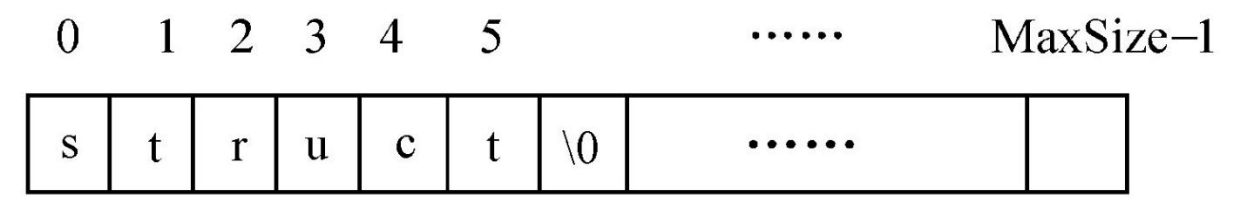
4.2 串的存储结构

1. 顺序存储结构

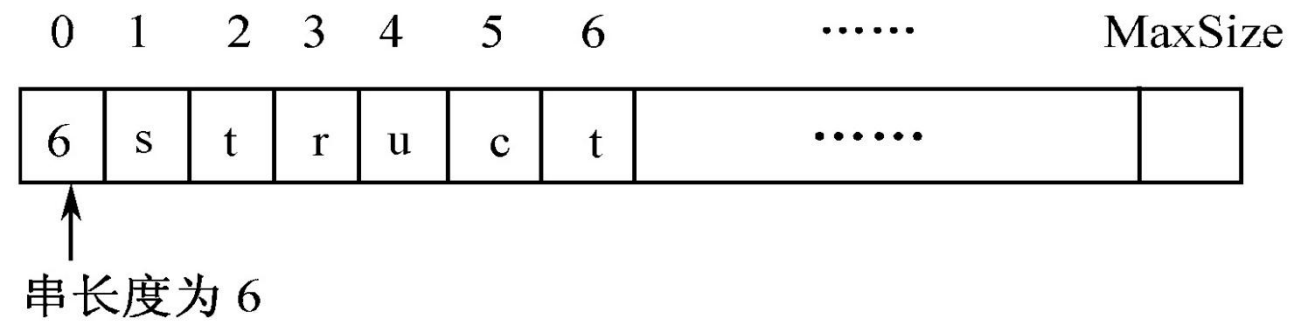
用定长的字符数组来描述串的定长顺序存储结构。

- 串长的表示方法：
  - 在串尾加一个结束标记（不计入长度）
  - 在下标为0的数组分量中存放串的长度值
  - 定义一个含有两个成员的结构体类型

(1) 在串尾加一个结束标记（不计入长度）



(2) 在下标为0的数组分量中存放串的长度值



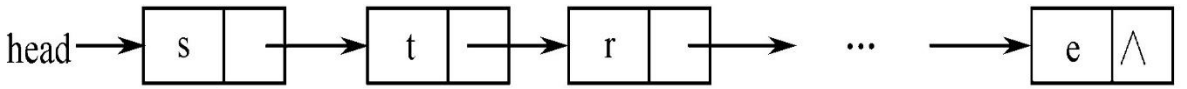
(3) 定义一个含有两个成员的结构体类型

```
const int MaxSize=1000    //串允许的最大字符个数
struct SeqString{
        char   data[MaxSize];
        int    length;        //串的长度
 };
```

4.2 串的存储结构

2. 链式存储结构

- 顺序串上的插入和删除操作不方便，需要移动大量的字符。因此，我们可用单链表方式来存储串值，串的这种链式存储结构简称为链串。



- 这种结构便于进行插入和删除运算，但存储空间利用率低。

- 为了提高存储密度，可使每个结点存放多个字符。通常将结点数据域存放的字符个数定义为结点的大小。

串的块链存储表示：

```
const int  CHUNKSIZE=4     // 可由用户定义的块大小
struct Chunk {
   char data[CHUNKSIZE];
   Chunk *next;
} ;
struct LString {
    Chunk *head, *tail;       // 串的头和尾指针
    int  curlen;              // 串的当前长度
 };
```

串的压缩存储结构的应用

- 在编辑系统中，整个文本编辑区可以看成是一个文本串，每一行是一个子串，构成一个结点。即: 同一行的串用定长结构(80个字符), 行和行之间用指针相联接。

4.3 串的操作算法

- **4.3.1 串的基本操作算法**
- **4.3.2 串的模式匹配**

4.3.1 串的基本操作算法

1．串连接

在串s1的后面连接串s2，s1改变，s2不改变。

```
void StrCat (char *s1, char *s2){
        len1=strlen(s1);
        len2=strlen(s2);
        if  (len1+len2>MaxSize-1)
        { cerr<<"超长";  exit(1); }
        i=0;
        while(s2[i]!='\0') {
                s1[len1+i]=s2[i];
                i++;
        }
        s1[i+len1]='\0';
}
```

2. 串比较

```
int StrCmp(char *s1, char *s2) {
    i=0;
    while (s1[i]==s2[i] && s1[i]!='\0')
        i++;
    return (s1[i]-s2[i]);
}
```

3．串复制

```
void StrCpy(char *s1, char *s2)
{
    int len=strlen(s2);
    if ( len>MaxSize-1 )
    {cerr<<"超长"; exit(1);}
    while (*s1++ = *s2++);
}
```

4.3.2 串的模式匹配算法

• **模式匹配：**子串（又称*模式串*）在主串（*目标串*）中的定位操作。

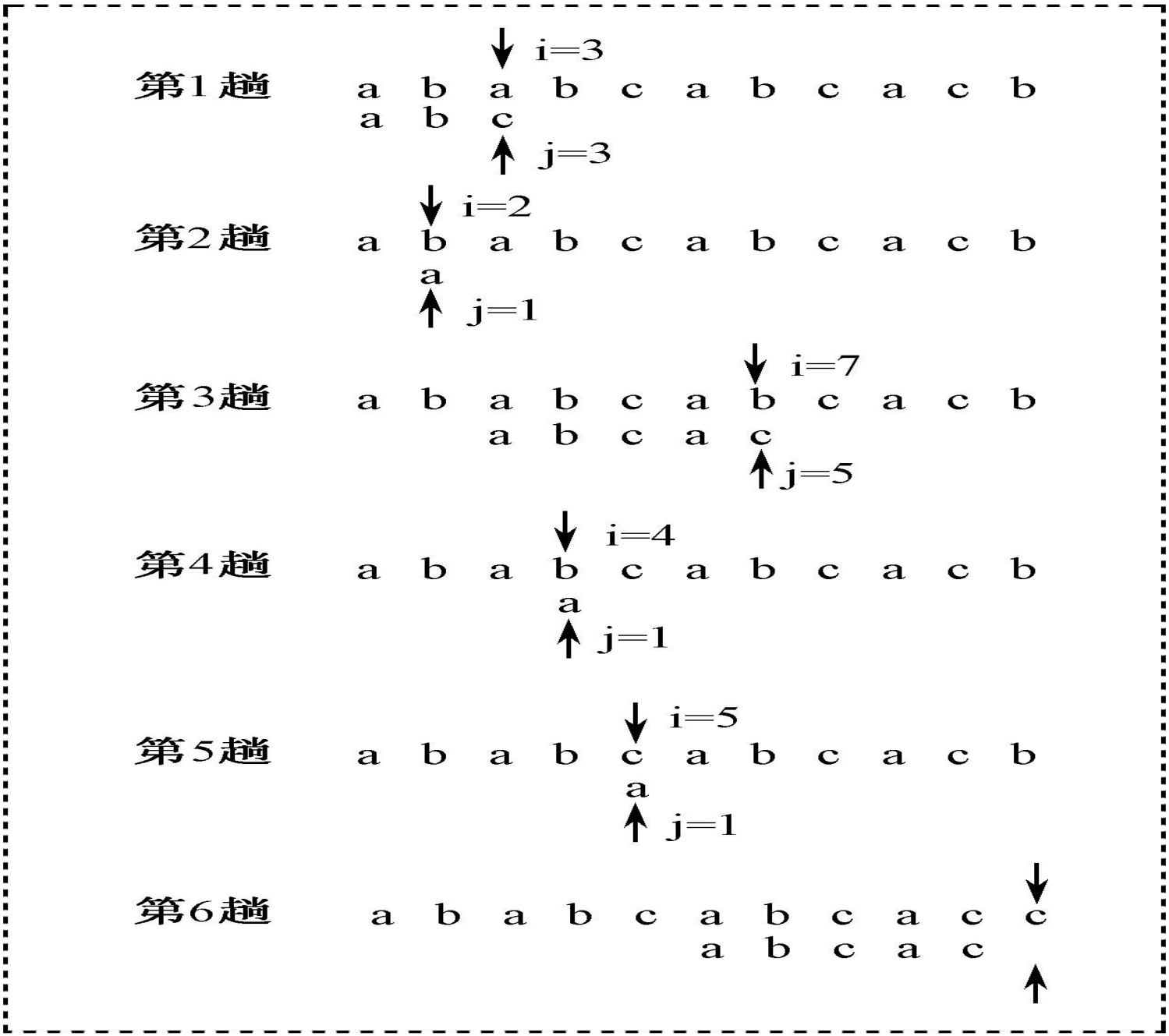
int Index (char *S, char *T)

初始条件：串S和T存在，T是非空串

操作结果：若主串S中存在和串T值相同的子串，
则返回它在主串S中第一次出现的位置；
否则函数值为-1。

• **模式匹配是串的一种重要操作，很多软件，若有“编辑”菜单项的话，则其中必有“查找”子菜单项。**

Example


i=3
第1趟
a b a b c a b c a c b
a b c
j=3
i=2
第2趟
a b a b c a b c a c b
a
j=1
i=7
第3趟
a b a b c a b c a c b
a b c a c
j=5
i=4
第4趟
a b a b c a b c a c b
a
j=1
i=5
第5趟
a b a b c a b c a c b
a
j=1
第6趟
a b a b c a b c a c c
a b c a c

朴素的模式匹配算法:

串长存放在0号单元

```
int BFmatching(char *s, char *t)
{
        i=1; j=1;
        n=s[0];  m=t[0];
        while(i<=n&&j<=m)
        {
                if(s[i]==t[j]) { i++; j++; }
                else { i=i-j+2; j=1; }
        }
        if (j>m) return i-j+1;
        else return 0;
}
```


朴素的模式匹配算法的时间复杂度分析：

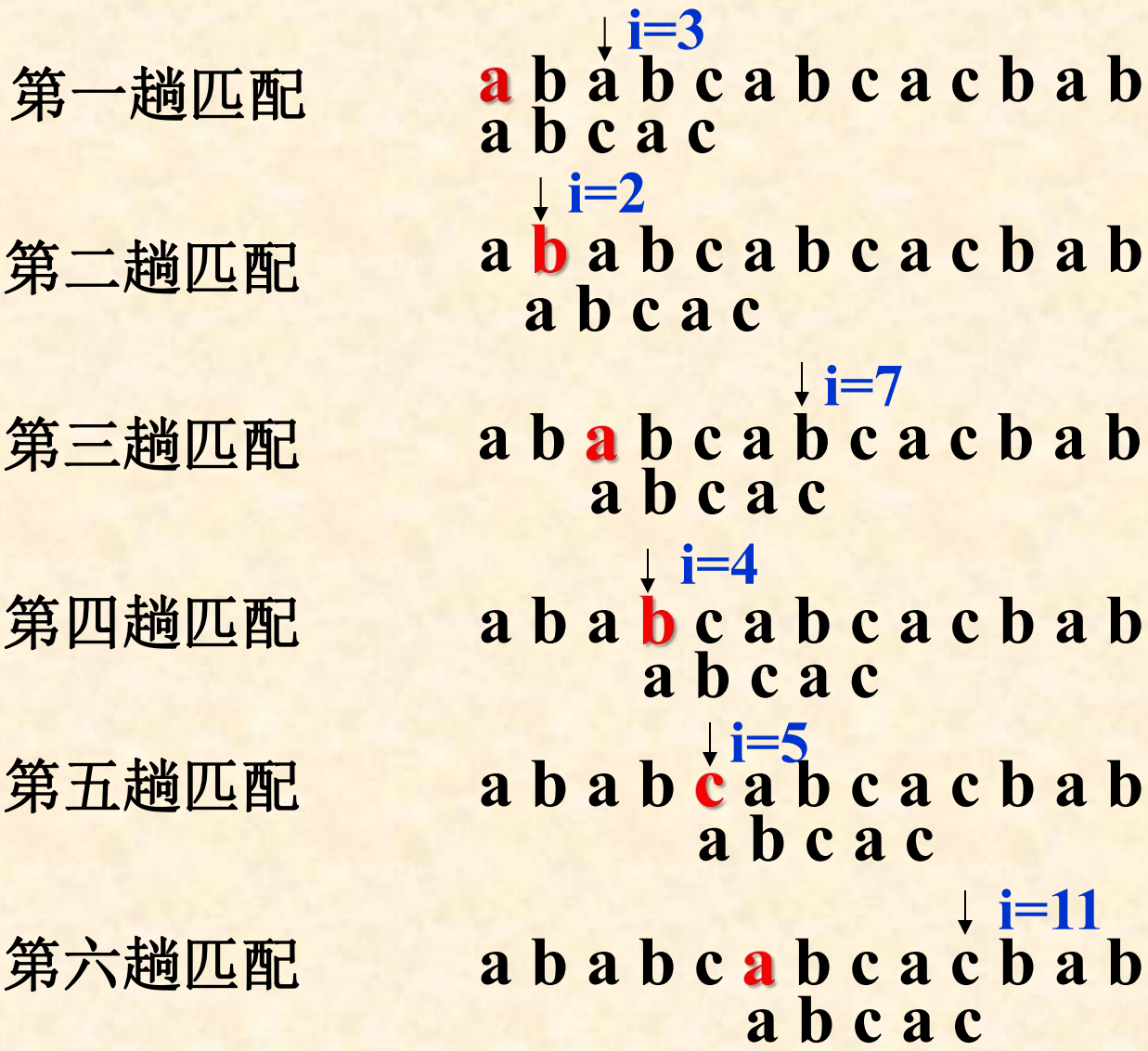
- 该算法的思想简单，易于理解，但算法的效率不高，原因是回溯，分析如下：
  - 最好情况下： $O(n+m)$
  - 最坏情况下： $O(n*m)$

4.3.2 改进算法

由D.E.Knuth与J.H.Morris 和V.R.Pratt同时发现的。简称 KMP算法。

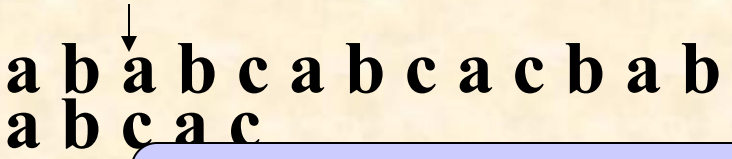
设主串S=“ababcabcacbab”, 模式串T=“abcac”。则朴素算法的匹配过程如下：

未改进时的匹配情况：


第一趟匹配
i=3
a b a b c a b c a c b a b
a b c a c
第二趟匹配
i=2
a b a b c a b c a c b a b
a b c a c
第三趟匹配
i=7
a b a b c a b c a c b a b
a b c a c
第四趟匹配
i=4
a b a b c a b c a c b a b
a b c a c
第五趟匹配
i=5
a b a b c a b c a c b a b
a b c a c
第六趟匹配
i=11
a b a b c a b c a c b a b
a b c a c

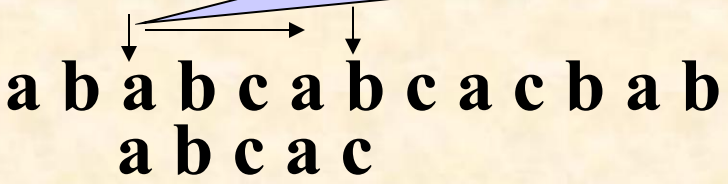
改进后的匹配情况：

第一趟匹配



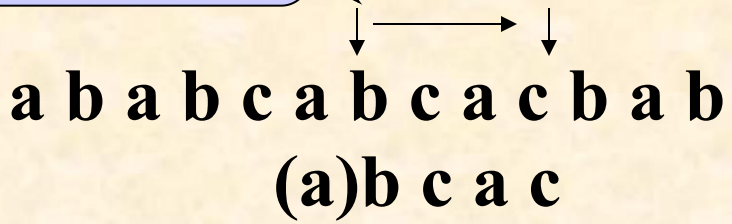
保持主串指针位置不动,而模式串向右滑动若干个字符

第二趟匹配



保持主串指针位置不动,而模式串向右滑动若干个字符

第三趟匹配



改进算法的一般情况：

设主串为“$s_1s_2\ldots s_n$”，模式串为“$t_1t_2\ldots t_m$”,当在某一趟匹配过程中出现“失配”情况时，即当

$$\begin{cases} s_i \neq t_j \\ \text{“}s_{i-j+1}s_{i-j+2}\ldots s_{i-1}\text{”} = \text{“}t_1t_2\ldots t_{j-1}\text{”} \end{cases}$$

要能立即确定模式右移的位数，即确定 s_i（i指针不回溯）应与模式串的哪一个字符继续进行比较？

假设此时 s_i 应与模式串的第 k（k<j）个字符t_k比较，则应有如下关系式存在：

“$t_1t_2\ldots t_{k-1}$” = “$t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$”

由于“失配”时，存在下列关系：

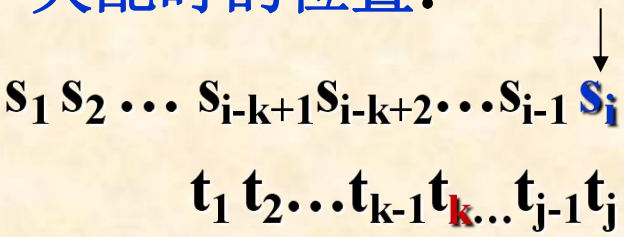
“$t_{j-k+1}t_{j-k+2}\ldots t_{j-1}$” = “$s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$”

于是有：

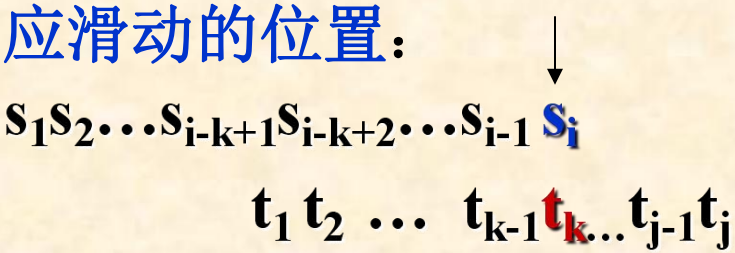
“$t_1t_2\ldots t_{k-1}$” = “$s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$”

当主串中第 i 个字符与模式串中第 j 个字符不等时，将模式串向右滑动至第k个字符和主串中第i个字符比较。

失配时的位置：



应滑动的位置：



• 若令next[j] = k, 则next[j]表明当模式中第j个字符与主串中相应字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符的位置。由此可以得出next函数的定义：

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时}\\ \mathrm{Max}\{k \mid 1<k<j \text{且} 't_1t_2\cdots t_{k-1}'='t_{j-k+1}\cdots t_{j-1}'\} & \\ 1 & \text{其它情况}\end{cases}$$

Example

- 【例】 设有模式串t= " abaabcac " ，则它的next数组值为

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[*j*]	0	1	1	2	2	3	1	2

j	0	1	2	3	4	5	6	7
模式串	a	b	a	a	b	c	a	c
next[*j*]	-1	0	0	1	1	2	0	1

KMP 算法

```
int Index_KMP(char *s, char *t)
 {
      i = 1;   j = 1;
     n=s[0];  m=t[0];
     while (i <= n && j <= m) {
        if (j == 0 || s[i] == t[j])
        { ++i;  ++j; }              // 继续比较后继字符
        else  j = next[j];          // 模式串向右移动
      }
      if (j > m)  return  i-m;              // 匹配成功
      else return 0;
 }
```