第4章 串

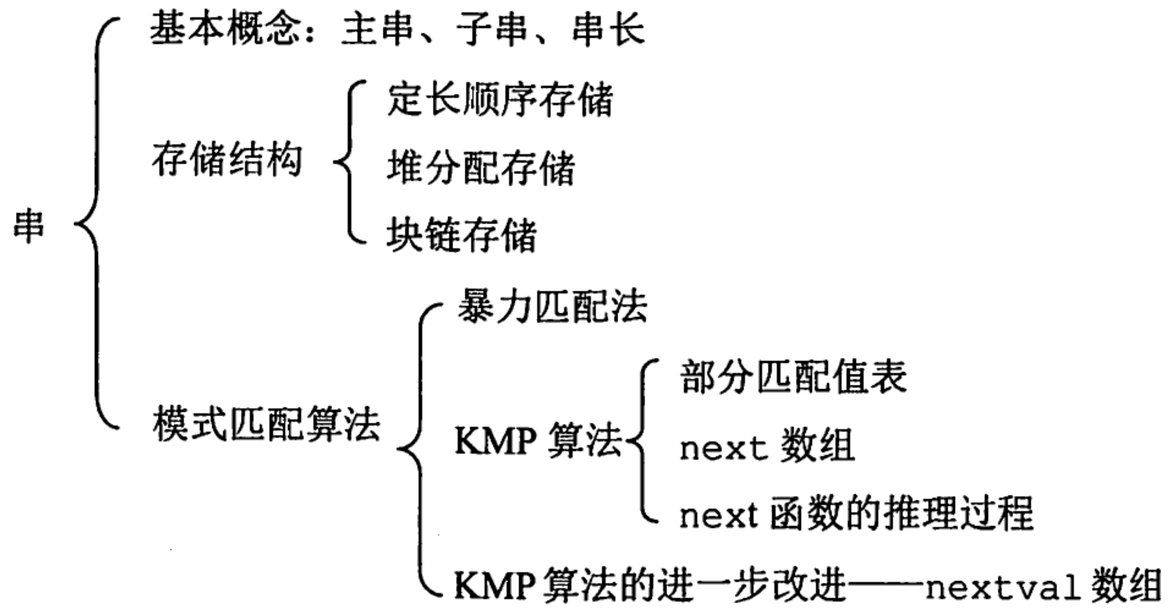
【考纲内容】

字符串模式匹配

【知识框架】

兑换后

看视频讲解



【复习提示】

本章是统考大纲第 6 章内容，采纳读者建议单独作为一章，大纲只要求掌握字符串模式匹配，重点掌握 KMP 匹配算法的原理及 next 数组的推理过程，手工求 next 数组可以先计算出部分匹配值表然后变形，或根据公式来求解。了解 nextval 数组的求解方法。

*4.1 串的定义和实现[①]

字符串简称串，计算机上非数值处理的对象基本都是字符串数据。我们常见的信息检索系统（如搜索引擎）、文本编辑程序（如 Word）、问答系统、自然语言翻译系统等，都是以字符串数据作为处理对象的。本章详细介绍字符串的存储结构及相应的操作。

4.1.1 串的定义

串（string）是由零个或多个字符组成的有限序列。一般记为

$$S='a_1a_2\cdots a_n' \quad (n\geqslant 0)$$

其中，S 是串名，单引号括起来的字符序列是串的值；a_i 可以是字母、数字或其他字符；串中字符的个数 n 称为串的长度。$n=0$ 时的串称为空串（用 $\varnothing$ 表示）。

串中任意多个连续的字符组成的子序列称为该串的子串，包含子串的串称为主串。某个字

① 本节不在统考大纲范围，仅供学习参考。

符在串中的序号称为该字符在串中的位置。子串在主串中的位置以子串的第一个字符在主串中的位置来表示。当两个串的长度相等且每个对应位置的字符都相等时，称这两个串是相等的。

例如，有串 A='China Beijing'，B='Beijing'，C='China'，则它们的长度分别为13, 7 和 5。B 和 C 是 A 的子串，B 在 A 中的位置是 7，C 在 A 中的位置是 1。

需要注意的是，由一个或多个空格（空格是特殊字符）组成的串称为空格串（注意，空格串不是空串），其长度为串中空格字符的个数。

串的逻辑结构和线性表极为相似，区别仅在于串的数据对象限定为字符集。在基本操作上，串和线性表有很大差别。线性表的基本操作主要以单个元素作为操作对象，如查找、插入或删除某个元素等；而串的基本操作通常以子串作为操作对象，如查找、插入或删除一个子串等。

4.1.2 串的存储结构

1. 定长顺序存储表示

类似于线性表的顺序存储结构，用一组地址连续的存储单元存储串值的字符序列。在串的定长顺序存储结构中，为每个串变量分配一个固定长度的存储区，即定长数组。

```
#define MAXLEN 255        //预定义最大串长为 255
typedef struct{
    char ch[MAXLEN];      //每个分量存储一个字符
    int length;           //串的实际长度
}SString;
```

串的实际长度只能小于或等于 MAXLEN，超过预定义长度的串值会被舍去，称为截断。串长有两种表示方法：一是如上述定义描述的那样，用一个额外的变量 len 来存放串的长度；二是在串值后面加一个不计入串长的结束标记字符“\0”，此时的串长为隐含值。

在一些串的操作（如插入、联接等）中，若串值序列的长度超过上界 MAXLEN，约定用“截断”法处理，要克服这种弊端，只能不限定串长的最大长度，即采用动态分配的方式。

2. 堆分配存储表示

堆分配存储表示仍然以一组地址连续的存储单元存放串值的字符序列，但它们的存储空间是在程序执行过程中动态分配得到的。

```
typedef struct{
    char *ch;                 //按串长分配存储区，ch 指向串的基地址
    int length;               //串的长度
}HString;
```

在 C 语言中，存在一个称之为“堆”的自由存储区，并用 malloc()和 free()函数来完成动态存储管理。利用 malloc()为每个新产生的串分配一块实际串长所需的存储空间，若分配成功，则返回一个指向起始地址的指针，作为串的基地址，这个串由 ch 指针来指示；若分配失败，则返回 NULL。已分配的空间可用 free()释放掉。

上述两种存储表示通常为高级程序设计语言所采用。块链存储表示仅做简单介绍。

3. 块链存储表示

类似于线性表的链式存储结构，也可采用链表方式存储串值。由于串的特殊性（每个元素只有一个字符），在具体实现时，每个结点既可以存放一个字符，也可以存放多个字符。每个结点称为块，整个链表称为块链结构。图 4.1(a)是结点大小为 4（即每个结点存放 4 个字符）的链表，最后一个结点占不满时通常用“#”补上；图 4.1(b)是结点大小为 1 的链表。

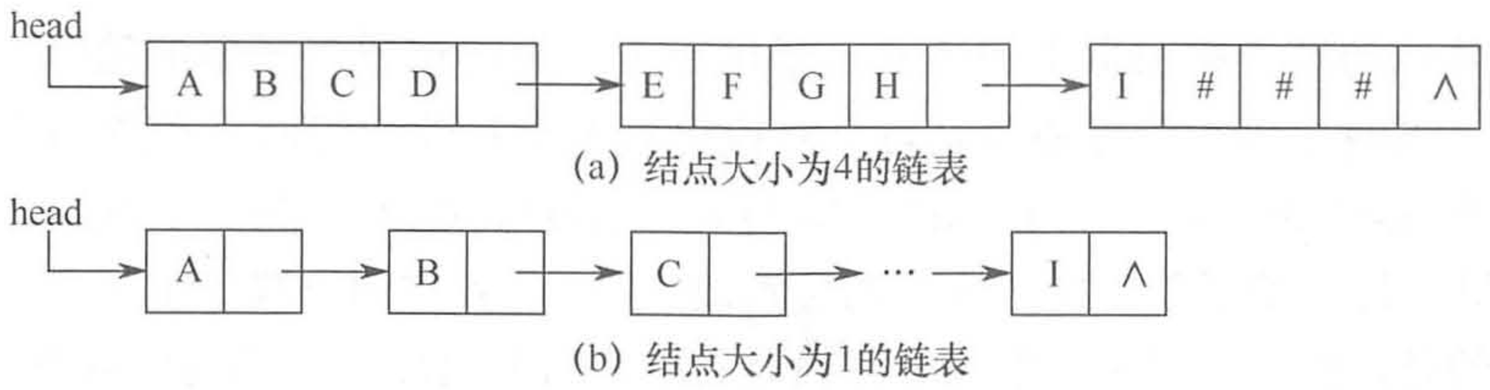


图 4.1　串值的链式存储方式

4.1.3　串的基本操作

- StrAssign(&T,chars)：赋值操作。把串 T 赋值为 chars。
- StrCopy(&T,S)：复制操作。由串 S 复制得到串 T。
- StrEmpty(S)：判空操作。若 S 为空串，则返回 TRUE，否则返回 FALSE。
- StrCompare(S,T)：比较操作。若 S>T，则返回值>0；若 S=T，则返回值=0；若 S<T，则返回值<0。
- StrLength(S)：求串长。返回串 S 的元素个数。
- SubString(&Sub,S,pos,len)：求子串。用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。
- Concat(&T,S1,S2)：串联接。用 T 返回由 S1 和 S2 联接而成的新串。
- Index(S,T)：定位操作。若主串 S 中存在与串 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 0。
- ClearString(&S)：清空操作。将 S 清为空串。
- DestroyString(&S)：销毁串。将串 S 销毁。

不同的高级语言对串的基本操作集可以有不同的定义方法。在上述定义的操作中，串赋值 StrAssign、串比较 StrCompare、求串长 StrLength、串联接 Concat 及求子串 SubString 五种操作构成串类型的最小操作子集，即这些操作不可能利用其他串操作来实现；反之，其他串操作（除串清除 ClearString 和串销毁 DestroyString 外）均可在该最小操作子集上实现。

4.2　串的模式匹配

4.2.1　简单的模式匹配算法

子串的定位操作通常称为串的模式匹配，它求的是子串（常称模式串）在主串中的位置。这里采用定长顺序存储结构，给出一种不依赖于其他串操作的暴力匹配算法。

```
int Index(SString S,SString T){
    int i=1,j=1;
    while(i<=S.length && j<=T.length){
        if(S.ch[i]==T.ch[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;
```

```
        else return 0;
    }
```

在上述算法中，分别用计数指针 i 和 j 指示主串 S 和模式串 T 中当前正待比较的字符位置。算法思想为：从主串 S 的第一个字符起，与模式 T 的第一个字符比较，若相等，则继续逐个比较后续字符；否则从主串的下一个字符起，重新和模式的字符比较；以此类推，直至模式 T 中的每个字符依次和主串 S 中的一个连续的字符序列相等，则称匹配成功，函数值为与模式 T 中第一个字符相等的字符在主串 S 中的序号，否则称匹配不成功，函数值为零。图 4.2 展示了模式 T='abcac'和主串 S 的匹配过程，每次匹配失败后，都把模式 T 后移一位。

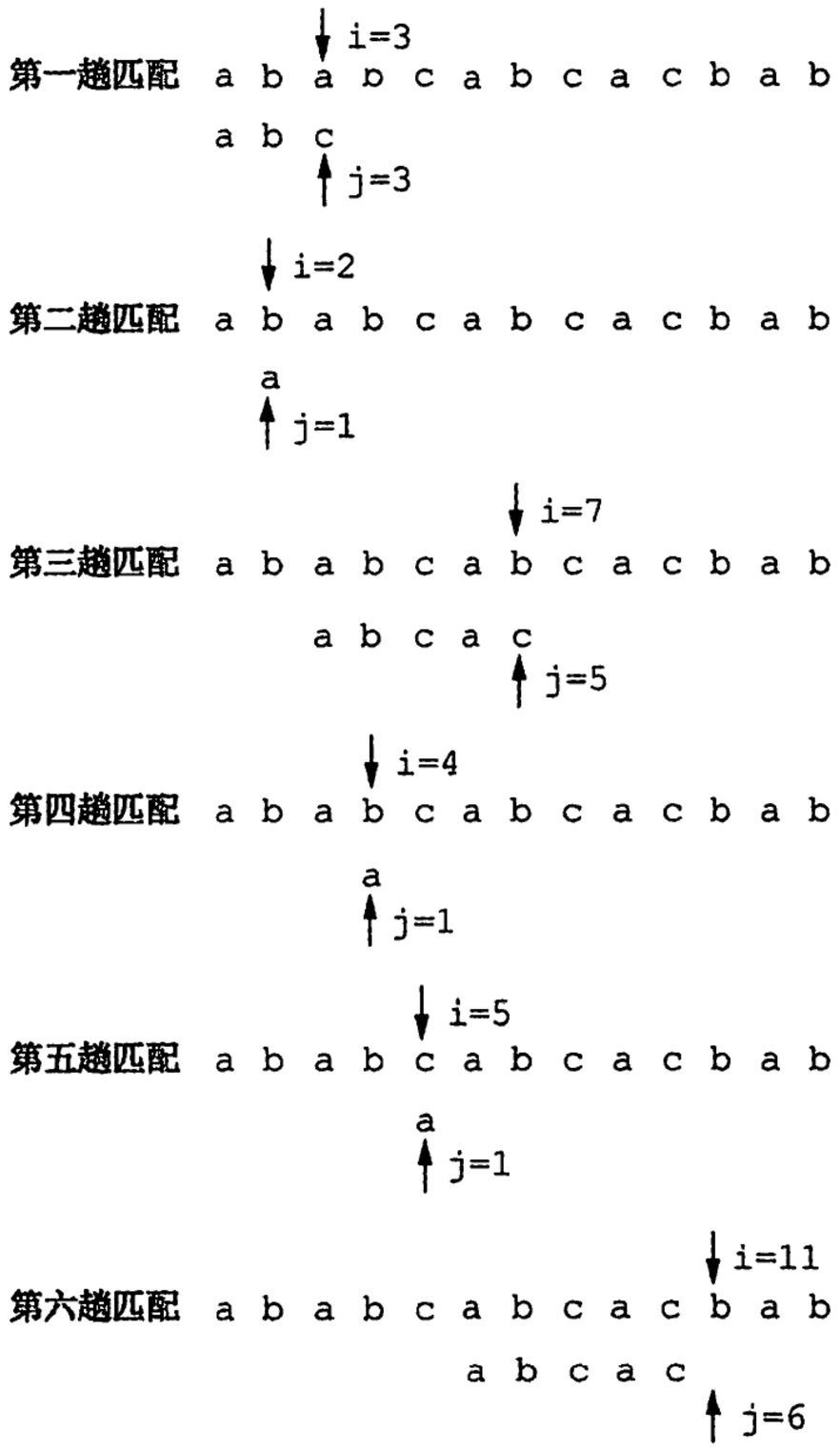


图 4.2　简单模式匹配算法举例

简单模式匹配算法的最坏时间复杂度为 $O(nm)$，其中 n 和 m 分别为主串和模式串的长度。例如，当模式串为'0000001'，而主串为'00000000000000000000000000000000000000000000000001'时，由于模式中前 6 个字符均为“0”，主串中前 45 个字符均为“0”，每趟匹配都是比较到模式的最后一个字符时才发现不等，指针 i 需回溯 39 次，总比较次数为 40×7 = 280 次。

4.2.2　串的模式匹配算法——KMP 算法

图 4.2 的匹配过程，在第三趟匹配中，i=7、j=5 的字符比较不等，于是又从 i=4、j=1 重新开始比较。然而，仔细观察会发现，i=4 和 j=1，i=5 和 j=1 及 i=6 和 j=1 这三次比较都是不必进行的。从第三趟部分匹配的结果可知，主串中第 4、5 和 6 个字符是'b'、'c'和'a'（即模式中第 2、3 和 4 个字符），因为模式中第一个字符是'a'，因此它无须再和这 3 个字符进行比较，而仅需将模式向右滑动 3 个字符的位置，继续进行 i=7、j=2 时的比较即可。

在暴力匹配中，每趟匹配失败都是模式后移一位再从头开始比较。而某趟已匹配相等的字符序列是模式的某个前缀，这种频繁的重复比较相当于模式串在不断地进行自我比较，这就是其低效率的根源。因此，可以从分析模式本身的结构着手，如果已匹配相等的前缀序列中有某个后缀正好是模式的前缀，那么就可以将模式向后滑动到与这些相等字符对齐的位置，主串 i 指针无须回溯，并从该位置开始继续比较。而模式向后滑动位数的计算仅与模式本身的结构有关，与主串无关（这里理解起来会比较困难，没关系，带着这个问题继续往后看）。

1. 字符串的前缀、后缀和部分匹配值

要了解子串的结构，首先要弄清楚几个概念：前缀、后缀和部分匹配值。前缀指除最后一个字符以外，字符串的所有头部子串；后缀指除第一个字符外，字符串的所有尾部子串；部分匹配值则为字符串的前缀和后缀的最长相等前后缀长度。下面以'ababa'为例进行说明：

- 'a'的前缀和后缀都为空集，最长相等前后缀长度为 0。
- 'ab'的前缀为{a}，后缀为{b}，{a}∩{b}=∅，最长相等前后缀长度为 0。
- 'aba'的前缀为{a,ab}，后缀为{a,ba}，{a,ab}∩{a,ba}={a}，最长相等前后缀长

度为 1。

- 'abab'的前缀{a,ab,aba}∩后缀{b,ab,bab}={ab}，最长相等前后缀长度为 2。
- 'ababa'的前缀{a,ab,aba,abab}∩后缀{a,ba,aba,baba}={a,aba}，公共元素有两个，最长相等前后缀长度为 3。

故字符串'ababa'的部分匹配值为 00123。

这个部分匹配值有什么作用呢？

回到最初的问题，主串为 a b a b c a b c a c b a b，子串为 a b c a c。

利用上述方法容易写出子串'abcac'的部分匹配值为 00010，将部分匹配值写成数组形式，就得到了部分匹配值（Partial Match, PM）的表。

编号	1	2	3	4	5
S	a	b	c	a	c
PM	0	0	0	1	0

下面用 PM 表来进行字符串匹配：

主串　a　b　a　b　c　a　b　c　a　c　b　a　b

子串　a　b　c

第一趟匹配过程：

发现 c 与 a 不匹配，前面的 2 个字符'ab'是匹配的，查表可知，最后一个匹配字符 b 对应的部分匹配值为 0，因此按照下面的公式算出子串需要向后移动的位数：

移动位数 = 已匹配的字符数 - 对应的部分匹配值

因为 2 - 0 = 2，所以将子串向后移动 2 位，如下进行第二趟匹配：

主串　a　b　a　b　c　a　b　c　a　c　b　a　b

子串　　a　b　c　a　c

第二趟匹配过程：

发现 c 与 b 不匹配，前面 4 个字符'abca'是匹配的，最后一个匹配字符 a 对应的部分匹配值为 1，4 - 1 = 3，将子串向后移动 3 位，如下进行第三趟匹配：

主串　a　b　a　b　c　a　b　c　a　c　b　a　b

子串　　　　　a　b　c　a　c

第三趟匹配过程：

子串全部比较完成，匹配成功。整个匹配过程中，主串始终没有回退，故 KMP 算法可以在 $O(n+m)$的时间数量级上完成串的模式匹配操作，大大提高了匹配效率。

某趟发生失配时，如果对应的部分匹配值为 0，那么表示已匹配相等序列中没有相等的前后缀，此时移动的位数最大，直接将子串首字符后移到主串当前位置进行下一趟比较；如果已匹配相等序列中存在最大相等前后缀（可理解为首尾重合），那么将子串向右滑动到和该相等前后缀对齐（这部分字符下一趟显然不需要比较），然后从主串当前位置进行下一趟比较。

2．KMP 算法的原理是什么？

我们刚刚学会了怎样计算字符串的部分匹配值、怎样利用子串的部分匹配值快速地进行字符串匹配操作，但公式“移动位数 = 已匹配的字符数 - 对应的部分匹配值”的意义是什么呢？

如图 4.3 所示，当 c 与 b 不匹配时，已匹配'abca'的前缀 a 和后缀 a 为最长公共元素。已知前缀 a 与 b、c 均不同，与后缀 a 相同，故无须比较，直接将子串移动“已匹配的字符数 - 对应的部分匹配值”，用子串前缀后面的元素与主串匹配失败的元素开始比较即可，如图 4.4 所示。

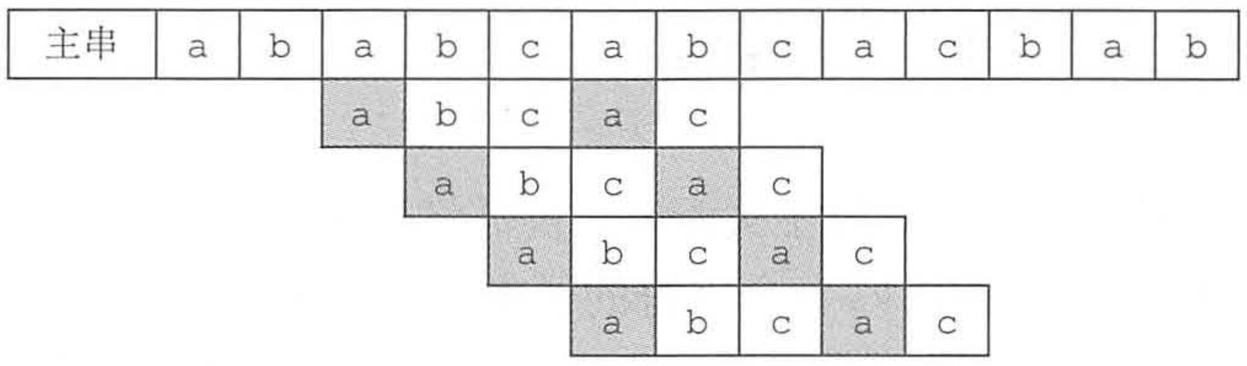


图 4.3　失配后移动情况

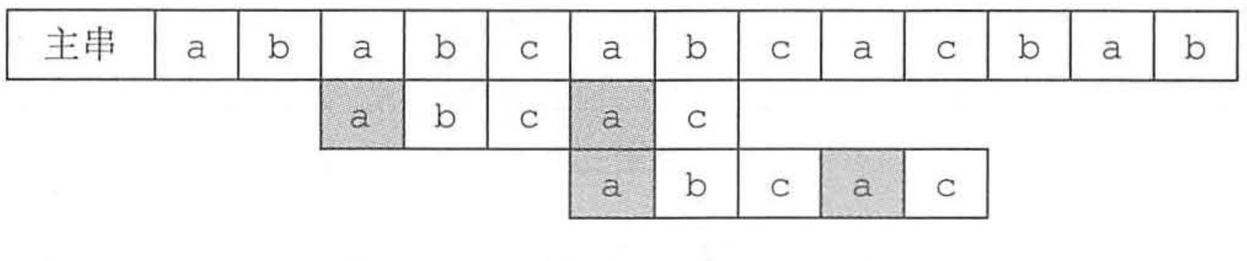


图 4.4　直接移动到合适位置

对算法的改进方法：

已知：右移位数 = 已匹配的字符数 - 对应的部分匹配值。

写成：Move=(j-1)-PM[j-1]。

使用部分匹配值时，每当匹配失败，就去找它前一个元素的部分匹配值，这样使用起来有些不方便，所以将 PM 表右移一位，这样哪个元素匹配失败，直接看它自己的部分匹配值即可。

将上例中字符串'abcac'的 PM 表右移一位，就得到了 next 数组：

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

我们注意到：

1）第一个元素右移以后空缺的用-1 来填充，因为若是第一个元素匹配失败，则需要将子串向右移动一位，而不需要计算子串移动的位数。

2）最后一个元素在右移的过程中溢出，因为原来的子串中，最后一个元素的部分匹配值是其下一个元素使用的，但显然已没有下一个元素，故可以舍去。

这样，上式就改写为

```
Move=(j-1)-next[j]
```

相当于将子串的比较指针 j 回退到

```
j=j-Move=j-((j-1)-next[j])=next[j]+1
```

有时为了使公式更加简洁、计算简单，将 next 数组整体+1。

因此，上述子串的 next 数组也可以写成

编号	1	2	3	4	5
S	a	b	c	a	c
next	0	1	1	1	2

最终得到子串指针变化公式 j=next[j]。在实际匹配过程中，子串在内存里是不会移动的，而是指针在变化，画图举例只是为了让问题描述得更为形象。next[j]的含义是：在子串的第 j 个字符与主串发生失配时，则跳到子串的 next[j]位置重新与主串当前位置进行比较。

如何推理 next 数组的一般公式？设主串为'$s_1s_2\cdots s_n$'，模式串为'$p_1p_2\cdots p_m$'，当主串中第 i 个字符与模式串中第 j 个字符失配时，子串应向右滑动多远，然后与模式中的哪个字符比较？

假设此时应与模式中第 k（k<j）个字符继续比较，则模式中前 k-1 个字符的子串必须满足

下列条件，且不可能存在 k′>k 满足下列条件：

$$'p_1p_2\cdots p_{k-1}'='p_{j-k+1}p_{j-k+2}\cdots p_{j-1}'$$

若存在满足如上条件的子串，则发生失配时，仅需将模式向右滑动至模式中第 k 个字符和主串第 i 个字符对齐，此时模式中前 k-1 个字符的子串必定与主串中第 i 个字符之前长度为 k-1 的子串相等，由此，只需从模式第 k 个字符与主串第 i 个字符继续比较即可，如图 4.5 所示。

主串	s_1	…	…	…	…	…	…	s_{i-k+1}	…	s_{i-1}	s_i	…	…	…	…	s_n
子串			p_1	…	p_{k-1}	…		p_{j-k+1}	…	p_{j-1}	p_j	…	p_m			
右移								p_1	…	p_{k-1}	p_k	…	…	p_m		

图 4.5　模式串右移到合适位置（阴影对齐部分表示上下字符相等）

当模式串已匹配相等序列中不存在满足上述条件的子串时（可以看成 k=1），显然应该将模式串右移 j-1 位，让主串第 i 个字符和模式第一个字符进行比较，此时右移位数最大。

当模式串第一个字符（j=1）与主串第 i 个字符发生失配时，规定 next[1]=0[①]。将模式串右移一位，从主串的下一个位置（i+1）和模式串的第一个字符继续比较。

通过上述分析可以得出 next 函数的公式：

$$\text{next}[j]=\begin{cases}0, & j=1\\ \max\{k \mid 1<k<j \text{且} 'p_1\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\}, & \text{当此集合不空时}\\ 1, & \text{其他情况}\end{cases}$$

上述公式不难理解，实际做题求 next 值时，用之前的方法也很好求，但如果想用代码来实现，貌似难度还真不小，我们来尝试推理求解的科学步骤。

首先由公式可知

$$\text{next}[1]=0$$

设 next[j]=k，此时 k 应满足的条件在上文中已描述。

此时 next[j+1]=？可能有两种情况：

(1)若 $p_k=p_j$，则表明在模式串中

$$'p_1\cdots p_{k-1}p_k'='p_{j-k+1}\cdots p_{j-1}p_j'$$

并且不可能存在 k′>k 满足上述条件，此时 next[j+1]=k+1，即

$$\text{next}[j+1]=\text{next}[j]+1$$

(2)若 $p_k\neq p_j$，则表明在模式串中

$$'p_1\cdots p_{k-1}p_k'\neq'p_{j-k+1}\cdots p_{j-1}p_j'$$

此时可以把求 next 函数值的问题视为一个模式匹配的问题。用前缀 $p_1\cdots p_k$ 去跟后缀 $p_{j-k+1}\cdots p_j$ 匹配，则当 $p_k\neq p_j$ 时应将 $p_1\cdots p_k$ 向右滑动至以第 next[k]个字符与 p_j 比较，如果 $p_{\text{next}[k]}$ 与 p_j 还是不匹配，那么需要寻找长度更短的相等前后缀，下一步继续用 $p_{\text{next}[\text{next}[k]]}$ 与 p_j 比较，以此类推，直到找到某个更小的 k′=next[next⋯[k]]（1<k′<k<j），满足条件

$$'p_1\cdots p_{k'}'='p_{j-k'+1}\cdots p_j'$$

则 next[j+1]=k′+1。

也可能不存在任何 k′满足上述条件，即不存在长度更短的相等前缀后缀，令 next[j+1]=1。

理解起来有一点费劲？下面举一个简单的例子。

图 4.6 的模式串中已求得 6 个字符的 next 值，现求 next[7]，因为 next[6]=3，又 $p_6\neq p_3$，

① 可理解为将主串第 i 个字符和模式串第一个字符的前面空位置对齐，也即模式串右移一位。

则需比较 p_6 和 p_1（因 next[3]=1），由于 $p_6 \neq p_1$，而 next[1]=0，所以 next[7]=1；求 next[8]，因 $p_7=p_1$，则 next[8]=next[7]+1=2；求 next[9]，因 $p_8=p_2$，则 next[9]=3。

j	1	2	3	4	5	6	7	8	9
模式	a	b	a	a	b	c	a	b	a
next[j]	0	1	1	2	2	3	?	?	?

图 4.6 求模式串的 next 值

通过上述分析写出求 next 值的程序如下：

```
void get_next(SString T,int next[]){
    int i=1,j=0;
    next[1]=0;
    while(i<T.length){
        if(j==0||T.ch[i]==T.ch[j]){
            ++i; ++j;
            next[i]=j;  //若 pi=pj，则 next[j+1]=next[j]+1
        }
        else
            j=next[j];  //否则令 j=next[j]，循环继续
    }
}
```

计算机执行起来效率很高，但对于我们手工计算来说会很难。因此，当我们需要手工计算时，还是用最初的方法。

与 next 数组的求解相比，KMP 的匹配算法相对要简单很多，它在形式上与简单的模式匹配算法很相似。不同之处仅在于当匹配过程产生失配时，指针 i 不变，指针 j 退回到 next[j]的位置并重新进行比较，并且当指针 j 为 0 时，指针 i 和 j 同时加 1。即若主串的第 i 个位置和模式串的第一个字符不等，则应从主串的第 i+1 个位置开始匹配。具体代码如下：

```
int Index_KMP(SString S,SString T,int next[]){
    int i=1, j=1;
    while(i<=S.length&&j<=T.length){
        if(j==0||S.ch[i]==T.ch[j]){
            ++i; ++j;                   //继续比较后继字符
        }
        else
            j=next[j];                  //模式串向右移动
    }
    if(j>T.length)
        return i-T.length;              //匹配成功
    else
        return 0;
}
```

尽管普通模式匹配的时间复杂度是 $O(mn)$，KMP 算法的时间复杂度是 $O(m+n)$，但在一般情况下，普通模式匹配的实际执行时间近似为 $O(m+n)$，因此至今仍被采用。KMP 算法仅在主串与子串有很多“部分匹配”时才显得比普通算法快得多，其主要优点是主串不回溯。

4.2.3 KMP 算法的进一步优化

前面定义的 next 数组在某些情况下尚有缺陷，还可以进一步优化。如图 4.7 所示，模式'aaaab'在和主串'aaabaaaab'进行匹配时：

主串	a	a	a	b	a	a	a	a	b
模式	a	a	a	a	b				
j	1	2	3	4	5				
next[j]	0	1	2	3	4				
nextval[j]	0	0	0	0	4				

图 4.7　KMP 算法进一步优化示例

当 i=4、j=4 时，s_4 跟 p_4（b≠a）失配，如果用之前的 next 数组还需要进行 s_4 与 p_3、s_4 与 p_2、s_4 与 p_1 这 3 次比较。事实上，因为 $p_{next[4]=3}=p_4=a$、$p_{next[3]=2}=p_3=a$、$p_{next[2]=1}=p_2=a$，显然后面 3 次用一个和 p_4 相同的字符跟 s_4 比较毫无意义，必然失配。那么问题出在哪里呢？

问题在于不应该出现 $p_j=p_{next[j]}$。理由是：当 $p_j\neq s_j$ 时，下次匹配必然是 $p_{next[j]}$ 跟 s_j 比较，如果 $p_j=p_{next[j]}$，那么相当于拿一个和 p_j 相等的字符跟 s_j 比较，这必然导致继续失配，这样的比较毫无意义。那么如果出现了 $p_j=p_{next[j]}$ 应该如何处理呢？

如果出现了，则需要再次递归，将 next[j]修正为 next[next[j]]，直至两者不相等为止，更新后的数组命名为 nextval。计算 next 数组修正值的算法如下，此时匹配算法不变。

```
void get_nextval(SString T,int nextval[]){
    int i=1, j=0;
    nextval[1]=0;
    while(i<T.length){
        if(j==0||T.ch[i]==T.ch[j]){
            ++i; ++j;
            if(T.ch[i]!=T.ch[j])  nextval[i]=j;
            else  nextval[i]=nextval[j];
        }
        else
            j=nextval[j];
    }
}
```

KMP 算法对于初学者来说可能不太容易理解，读者可以尝试多读几遍本章的内容，并参考一些其他教材的相关内容来巩固这个知识点。

4.2.4　本节试题精选

一、单项选择题

01. 设有两个串 S_1 和 S_2，求 S_2 在 S_1 中首次出现的位置的运算称为（　）。

A. 求子串　　B. 判断是否相等　　C. 模式匹配　　D. 连接

02. KMP 算法的特点是在模式匹配时指示主串的指针（　）。

A. 不会变大　　B. 不会变小　　C. 都有可能　　D. 无法判断

03. 设主串的长度为 n，子串的长度为 m，则简单的模式匹配算法的时间复杂度为（　），KMP 算法的时间复杂度为（　）。

A. $O(m)$　　B. $O(n)$　　C. $O(mn)$　　D. $O(m+n)$

04. 已知串 S='aaab'，其 next 数组值为（　）。

A. 0123　　B. 0112　　C. 0231　　D. 1211

05. 串'abababaababaa'的 next 数组值为（　）。

A. 01234567899　　B. 012121111212

C. 011234223456　　D. 0123012322345

06. 串'abababaabaa'的 next 数组为（ ）。

A. −1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8　　B. −1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1

C. −1, 0, 0, 1, 2, 3, 1, 1, 2, 3, 4, 5　　D. −1, 0, 1, 2, −1, 0, 1, 2, 1, 1, 2, 3

07. 串'abababaabaa'的 nextval 数组为（ ）。

A. 0, 1, 0, 1, 1, 2, 0, 1, 0, 1, 0, 2　　B. 0, 1, 0, 1, 1, 4, 1, 1, 0, 1, 0, 2

C. 0, 1, 0, 1, 0, 4, 2, 1, 0, 1, 0, 4　　D. 0, 1, 1, 1, 0, 2, 1, 1, 0, 1, 0, 4

08.【2015 统考真题】已知字符串 S 为'abaabaabacacaabaabcc'，模式串 t 为'abaabc'。采用 KMP 算法进行匹配，第一次出现“失配”(s[i]≠t[j]) 时，i=j=5，则下次开始匹配时，i 和 j 的值分别是（ ）。

A. i=1, j=0　　B. i=5, j=0　　C. i=5, j=2　　D. i=6, j=2

09.【2019 统考真题】设主串 T='abaabaabcabaabc'，模式串 S='abaabc'，采用 KMP 算法进行模式匹配，到匹配成功时为止，在匹配过程中进行的单个字符间的比较次数是（ ）。

A. 9　　B. 10　　C. 12　　D. 15

二、综合应用题

01. 在字符串模式匹配的 KMP 算法中，求模式的 next 数组值的定义如下：

$$\text{next}[j]=\begin{cases}0, & j=1\\ \max\{k \mid 1<k<j \text{且} 'p_1 \cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\}, & \text{此集合不空时}\\ 1, & \text{其他情况}\end{cases}$$

1）当 j=1 时，为什么要取 next[1]=0?

2）为什么要取 max{k}，k 最大是多少?

3）其他情况是什么情况，为什么取 next[j]=1?

02. 设有字符串 S='aabaabaabaac'，P='aabaac'。

1）求出 P 的 next 数组。

2）若 S 作主串，P 作模式串，试给出 KMP 算法的匹配过程。

4.2.5 答案与解析

一、单项选择题

01. C

求子串操作是从串 S 中截取第 i 个字符起长度为 l 的子串，A 错。BD 明显错。选 C。

02. B

在 KMP 算法的比较过程中，主串不会回溯，所以主串的指针不会变小。选 B。

03. C、D

尽管实际应用中，一般情况下简单的模式匹配算法的时间复杂度近似为 $O(m+n)$，但它的理论时间复杂度还是 $O(mn)$，选 C。KMP 算法的时间复杂度为 $O(m+n)$，选 D。

04. A

1）设 next[1]=0，next[2]=1。

编号	1	2	3	4
S	a	a	a	b
next	0	1		

2）j=3 时 k=next[j-1]=next[2]=1，观察 S[j-1]（S[2]）与 S[k](S[1])是否相等，

S[2]=a，S[1]=a，S[2]=S[1]，所以 next[j]=k+1=2。

```
     ↓j-1=2
 a   a   a   b
     a   a   a   b
     ↑k=1
```

3）j=4 时 k=next[j-1]=next[3]=2，观察 S[j-1]（S[3]）与 S[k](S[2])是否相等，S[3]=a，S[2]=a，S[3]=S[2]，所以 next[j]=k+1=3。

```
         ↓j-1=3
 a   a   a   b
     a   a   a   b
         ↑k=2
```

最后的结果如下，选 A。

编号	1	2	3	4
S	a	a	a	b
next	0	1	2	3

本题采用 next 的推理原理求解，如果字符串较长，求解过程会比较烦琐。

05. C

这道题采用手工求 next 数组的方法。先求串 S='ababaaababaa'的部分匹配值：

- 'a'的前后缀都为空，最长相等前后缀长度为 0。
- 'ab'的前缀{a}∩后缀{b}=∅，最长相等前后缀长度为 0。
- 'aba'的前缀{a,ab}∩后缀{a,ba}={a}，最长相等前后缀长度为 1。
- 'abab'的前缀{a,ab,aba}∩后缀{b,ab,bab}={ab}，最长相等前后缀长度为 2。
- ……

依次求出的部分匹配值如下表第三行所示，将其整体右移一位，低位用-1 填充，如下表第四行所示。

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	a	b	a	a	a	b	a	b	a	a
PM	0	0	1	2	3	1	1	2	3	4	5	6
next	-1	0	0	1	2	3	1	1	2	3	4	5

选项中 next[1]等于 0，故将 next 数组整体加 1，答案选 C。

06. C

解析见上题，选 C。注意，next 数组是否整体加 1 都正确，需根据题意具体分析。

注意：在实际 KMP 算法中，为了使公式更简洁、计算简单，如果串的位序是从 1 开始的，则 next 数组才需要整体加 1；如果串的位序是从 0 开始的，则 next 数组不需要整体加 1。

07. C

nextval 从 0 开始，可知串的位序从 1 开始。第一步，令 nextval[1]=next[1]=0。

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	a	b	a	a	a	b	a	b	a	a
next	0	1	1	2	3	4	2	2	3	4	5	6
nextval	0	1	0	1	0	4	2	1	0	1	0	4

从 j=2 开始，依次判断 p_j 是否等于 $p_{next[j]}$？若是则将 next[j]修正为 next[next[j]]，直至两者不相等为止。由下述推理可知，答案选 C。

第 2 步：p_2=b、$p_{next[2]}$=a，$p_2 \neq p_{next[2]}$，nextval[2]=next[2]=1；

第 3 步：p_3=a、$p_{next[3]}$=a，$p_3=p_{next[3]}$，nextval[3]=nextval[next[3]]=nextval[1]=0；

第 4 步：p_4=b、$p_{next[4]}$=b，$p_4=p_{next[4]}$，nextval[4]=nextval[next[4]]=nextval[2]=1；

第 5 步：p_5=a、$p_{next[5]}$=a，$p_5=p_{next[5]}$，nextval[5]=nextval[next[5]]=nextval[3]=0；

第 6 步：p_6=a、$p_{next[6]}$=b，$p_6 \neq p_{next[6]}$，nextval[6]=next[6]=4；

第 7 步：p_7=a、$p_{next[7]}$=b，$p_7 \neq p_{next[7]}$，nextval[7]=next[7]=2；

第 8 步：p_8=b、$p_{next[8]}$=b，$p_8=p_{next[8]}$，nextval[8]=nextval[next[8]]=nextval[2]=1；

第 9 步：p_9=a、$p_{next[9]}$=a，$p_9=p_{next[9]}$，nextval[9]=nextval[next[9]]=nextval[3]=0；

第 10 步：p_{10}=b、$p_{next[10]}$=b，$p_{10}=p_{next[10]}$，nextval[10]=nextval[next[10]]=nextval[4]=1；

第 11 步：p_{11}=a、$p_{next[11]}$=a，$p_{11}=p_{next[11]}$，nextval[11]=nextval[next[11]]=nextval[5]=0；

第 12 步：p_{12}=a、$p_{next[12]}$=a，$p_{10}=p_{next[12]}$，nextval[12]=nextval[next[12]]=nextval[6]=4；

在第 5 步的推理中，$p_5=p_{next[5]}$=a，按前面的讲解部分，应该继续让 p_3 和 $p_{next[3]}$ 比较（恰好 $p_3=p_{next[3]=1}$），注意到此时 nextval[3]的值已存在，故直接将 nextval[5]赋值为 nextval[3]。

对于一般情况，nextval 数组是从前往后逐步求解的，发生 $p_j=p_{next[j]}$ 时，因为 nextval[next[j]]早已求得，所以直接将 nextval[j]赋值为 nextval[next[j]]。

08. C

由题中“失配 s[i]≠t[j]时，i=j=5”，可知题中的主串和模式串的位序都是从 0 开始的（要注意灵活应变）。按照 next 数组生成算法，对于 t 有

编号	0	1	2	3	4	5
t	a	b	a	a	b	c
next	-1	0	0	1	1	2

发生失配时，主串指针 i 不变，子串指针 j 回退到 next[j]位置重新比较，当 s[i]≠t[j]时，i=j=5，由 next 表得知 next[j]=next[5]=2（位序从 0 开始）。因此，i=5，j=2。

09. B

假设位序从 0 开始的，按照 next 数组生成算法，对于 S 有

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

第一趟连续比较 6 次，在模式串的 5 号位和主串的 5 号位匹配失败，模式串的下一个比较位置为 next[5]，即下一次比较从模式串的 2 号位和主串的 5 号位开始，然后直到模式串 5 号位和主串 8 号位匹配，第二趟比较 4 次，匹配成功。单个字符的比较次数为 10 次，因此选 B。

二、综合应用题

01. 【解答】

1）当模式串中的第一个字符与主串的当前字符比较不相等时，next[1]=0，表示模式串应右移一位，主串当前指针后移一位，再和模式串的第一字符进行比较。

2）当主串的第 i 个字符与模式串的第 j 个字符失配时，主串 i 不回溯，则假定模式串的第 k 个字符与主串的第 i 个字符比较，k 值应满足条件 1<k<j 且'$p_1 \cdots p_{k-1}$'='$p_{j-k+1} \cdots p_{j-1}$'，

即 k 为模式串的下次比较位置。k 值可能有多个，为了不使向右移动丢失可能的匹配，右移距离应该取最小，由于 j-k 表示右移的距离，所以取 max{k}。k 的最大值为 j-1。

3）除上面两种情况外，发生失配时，主串指针 i 不回溯，在最坏情况下，模式串从第 1 个字符开始与主串的第 i 个字符比较。

02.【解答】

1）P='aabaac'，按照 next 数组生成算法，对于 P 有：

① 设 next[1]=0，next[2]=1。

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1				

② j=3 时 k=next[j-1]=next[2]=1，观察 S[j-1](S[2])与 S[k](S[1])是否相等，S[2]=a，S[1]=a，S[2]=S[1]，所以 next[j]=k+1=2。

```
      ↓j-1=2
  a   a   b   a   a   c
      a   a   b   a   a   c
      ↑k=1
```

③ j=4 时 k=next[j-1]=next[3]=2，观察 S[j-1](S[3])与 S[k](S[2])是否相等，S[3]=b，S[2]=a，S[3]!=S[2]。

```
          ↓j-1=3
  a   a   b   a   a   c
      a   a   b   a   a   c
          ↑k=2
```

此时 k=next[k]=1，观察 S[3]与 S[k](S[1])是否相等，S[3]=b，S[1]=a，S[3]!=S[1]。k=next[k]=0，因为 k=0，所以 next[j]=1。

```
          ↓j-1=3
  a   a   b   a   a   c
          a   a   b   a   a   c
      ↑k=0
```

④ j=5 时 k=next[j-1]=next[4]=1，观察 S[j-1](S[4])与 S[k](S[1])是否相等，S[4]=a，S[1]=a，S[4]=S[1]，所以 next[j]=k+1=2。

```
              ↓j-1=4
  a   a   b   a   a   c
              a   a   b   a   a   c
              ↑k=1
```

⑤ j=6 时 k=next[j-1]=next[5]=2，观察 S[j-1](S[5])与 S[k](S[2])是否相等，S[5]=a，S[2]=a，S[5]=S[2]，所以 next[j]=k+1=3。

```
                  ↓j-1=5
  a   a   b   a   a   c
              a   a   b   a   a   c
                  ↑k=2
```


最后的结果为

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1	2	1	2	3

也可以通过求部分匹配值表的方法来求 next 数组。

2）利用 KMP 算法的匹配过程如下。

第一趟：从主串和模式串的第一个字符开始比较，失配时 i=6，j=6。

主串	a	a	b	a	a	b	a	a	b	a	a	c
	a	a	b	a	a	c						

第二趟：next[6]=3，主串当前位置和模式串的第 3 个字符继续比较，失配时 i=9，j=6。

主串	a	a	b	a	a	b	a	a	b	a	a	c
				a	a	b	a	a	c			

第三趟：next[6]=3，主串当前位置和模式串的第 3 个字符继续比较，匹配成功。

主串	a	a	b	a	a	b	a	a	b	a	a	c
							a	a	b	a	a	c

归纳总结

在学习 KMP 算法时，应从分析暴力法的弊端入手，思考通过何种方式来优化。实际上已匹配相等的序列就是模式串的某个前缀，因此每次回溯就相当于是模式串与模式串某个前缀在比较，这种频繁的重复比较是效率低的原因所在，可以从分析模式串本身的结构入手，然后就能得知当匹配到某个字符不等时，应向后滑动到什么位置，即已匹配相等的前缀和模式串若有首尾重合，对齐它们，对齐部分显然无需再比较，下一步直接从主串的当前位置继续比较。

思维拓展

编程实现：模式串在主串中有多少个完全匹配的子串？（注：统考不会考 KMP 算法题）