

第4章 串

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。

4.1 串类型的定义

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: $S = \text{“}a_1a_2a_3\ldots\text{”}$, 其中 S 是串名, $a_i (1 \leq i \leq n)$ 是单个, 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。

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=“这是字符串”， B=“是”

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3。因此，称B在A中的序号为3。

特别地，空串是任意串的子串，任意串是其自身的子串。

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

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。

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

4.2 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 ;
```


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 ;

}

2 求子串操作

Status SubString (StringType s, int pos, int len,
StringType *sub)

```
{ 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-pos+1 ; /* 求得子串长度 */  
    for (j=0, k=pos ; k<=len ; k++, j++)  
        sub->str[j]=s.str[k] ; /* 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(称为“**堆**”)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。

特点是：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 ;
```

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]=s1->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

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 ;
```

(2) 块链串的类型定义

```
typedef struct
```

```
{ BNODE head;    /* 头指针 */  
  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。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，T为模式串，且不妨设：

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

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

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

这样，串匹配问题可简化为找出某给定模式T在给
定目标串S中首次出现的有效位移。

算法实现

```
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 */
```

```
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 */
```

```
}
```

该算法简单，易于理解。在一些场合的应用里，如文字处理中的文本编辑，其效率较高。

该算法的时间复杂度为 $O(n*m)$ ，其中 n 、 m 分别是主串和模式串的长度。通常情况下，实际运行过程中，该算法的执行时间近似于 $O(n+m)$ 。

理解该算法的关键点

当第一次 $s_k \neq t_j$ 时：主串要退回到 $k-j+1$ 的位置，而模式串也要退回到第一个字符（即 $j=0$ 的位置）。

比较出现 $s_k \neq t_j$ 时：则应该有 $s_{k-1}=t_{j-1}$ ， \dots ， $s_{k-j+1}=t_1$ ， $s_{k-j}=t_0$ 。

4.3.2 模式匹配的一种改进算法

该改进算法是由D.E.Knuth，J.H.Morris和V.R.Pratt提出来的，简称为KMP算法。其改进在于：

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”尽可能远的一段距离后，继续进行匹配。

例：设有串 $s = \text{“abacabab”}$ ， $t = \text{“abab”}$ 。则第一次匹配过程如图4-2所示。

$s = \text{“a b a c b b”}$ $i = 3$

|| || || \neq

$t = \text{“a b a b”}$ $j = 3$

匹配失败

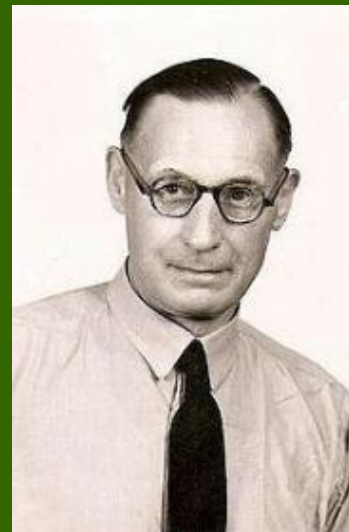


图4-2 模式匹配示例

在 $i=3$ 和 $j=3$ 时，匹配失败。但重新开始第二次匹配时，不必从 $i=1$ ， $j=0$ 开始。因为 $s_1=t_1$ ， $t_0 \neq t_1$ ，必有 $s_1 \neq t_0$ ，又因为 $t_0=t_2$ ， $s_2=t_2$ ，所以必有 $s_2=t_0$ 。由此可知，第二次匹配可以直接从 $i=3$ 、 $j=1$ 开始。

总之，在主串 s 与模式串 t 的匹配过程中，一旦出现 $s_i \neq t_j$ ，主串 s 的指针不必回溯，而是直接与模式串的 t_k ($0 \leq k < j$) 进行比较，而 k 的取值与主串 s 无关，只与模式串 t 本身的构成有关，即从模式串 t 可求得 k 值。)

不失一般性，设主串 $s = "s_1s_2 \dots s_n"$ ，模式串 $t = "t_1t_2 \dots t_m"$ 。

当 $s_i \neq t_j (1 \leq i \leq n-m, 1 \leq j < m, m < n)$ 时，主串 s 的指针 i 不必回溯，而模式串 t 的指针 j 回溯到第 $k (k < j)$ 个字符继续比较，则模式串 t 的前 $k-1$ 个字符必须满足4-1式，而且不可能存在 $k' > k$ 满足4-1式。

$$t_1t_2 \dots t_{k-1} = s_{i-(k-1)} s_{i-(k-2)} \dots s_{i-2} s_{i-1} \quad (4-1)$$

而已经得到的“部分匹配”的结果为：

$$t_{j-(k-1)} t_{j-k} \dots t_{j-1} = s_{i-(k-1)} s_{i-(k-2)} \dots s_{i-2} s_{i-1} \quad (4-2)$$

由式(4-1)和式(4-2)得：

$$t_1t_2 \dots t_{k-1} = t_{j-(k-1)} t_{j-k} \dots t_{j-1} \quad (4-3)$$

该推导过程可用图4-3形象描述。实际上，式(4-3)描述了模式串中存在相互重叠的子串的情况。

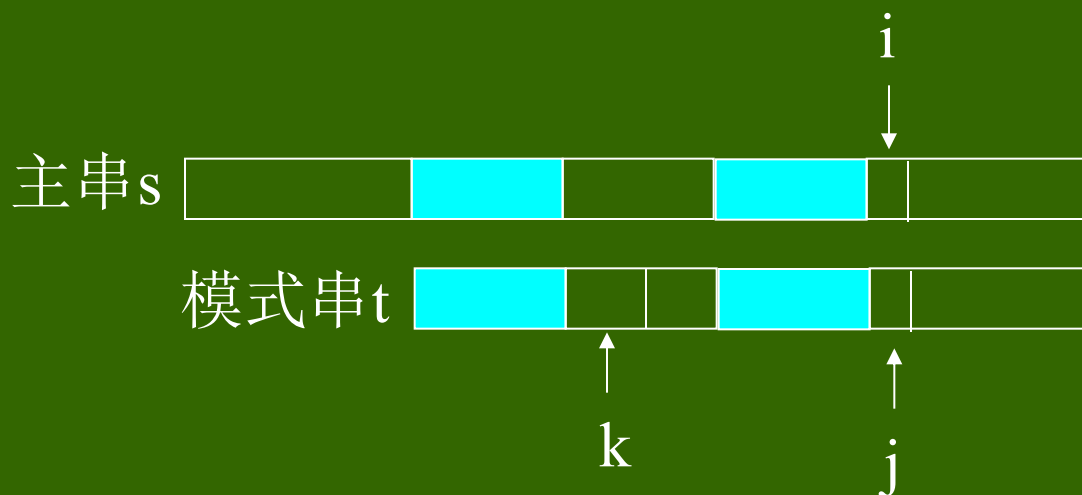


图4-3 KMP算法示例

定义next[j]函数为

$$\text{next}[j] = \begin{cases} 0 \\ \text{Max}\{k | 1 < k < j \wedge t_1 t_2 \dots t_{k-1} = t_{j-(k-1)} t_{j-k} \dots t_{j-1} \} \\ 1 \end{cases}$$

当j=1时

该集合不空时

其它情况

在求得了 $\text{next}[j]$ 值之后，KMP算法的思想是：

设目标串(主串)为 s ，模式串为 t ，并设 i 指针和 j 指针分别指示目标串和模式串中正待比较的字符，设 i 和 j 的初值均为1。若有 $s_i=t_j$ ，则 i 和 j 分别加1。否则， i 不变， j 退回到 $j=\text{next}[j]$ 的位置，再比较 s_i 和 t_j ，若相等，则 i 和 j 分别加1。否则， i 不变， j 再次退回到 $j=\text{next}[j]$ 的位置，依此类推。直到下列两种可能：

(1) j 退回到某个下一个 $[j]$ 值时字符比较相等，则指针各自加1继续进行匹配。

(2)退回到 $j=0$ ，将 i 和 j 分别加1，即从主串的下一个字符 s_{i+1} 模式串的 t_1 重新开始匹配。

KMP算法如下

```
#define Max_Strlen 1024
```

```
int next[Max_Strlen];
```

```
int KMP_index (StringType s , StringType t)
```

```
/* 用KMP算法进行模式匹配，匹配返回位置，否则返回-1 */
```

```
/*用静态存储方式保存字符串， s和t分别表示主串和模式串 */
```

```
{ int k=0 , j=0 ;    /*初始匹配位置设置 */
```

```
while (k<s.length)&&(j<t.length
```

```
{ if ((j==-1)|| (s.str[k]==t.str[j])) { k++ ; j++ ; }
```

```
else j=next[j] ;
```

```
}
```

```
if (j>= t.length) return(k-t.length) ;
```

```
else return(-1) ;
```

```
}
```

很显然，KMP_index函数是在已知下一个函数值的基础上执行的，以下讨论如何求next函数值？

由式(4-3)知，求模式串的next[j]值与主串s无关，只与模式串t本身的构成有关，则可把求next函数值的问题看成是一个模式匹配问题。由next函数定义可知：

当j=1时：next[1]=0。

设next[j]=k，即在模式串中存在： $t_1t_2\dots t_{k-1}=t_{j-(k-1)}t_{j-k}\dots t_{j-1}$ ，其中下标k满足 $1<k<j$ 的某个最大值，此时求next[j+1]的值有两种可能：

(1) 若有 $t_k=t_j$ ：则表明在模式串中有：

$t_1t_2\dots t_{k-1}t_k=t_{j-(k-1)}t_{j-k}\dots t_{j-1}t_j$ ，且不可能存在 $k'>k$ 满足上式，即：next[j+1]=next[j]+1=k+1

(2) 若有 $t_k \neq t_j$ ：则表明在模式串中有： $t_1 t_2 \dots t_{k-1} t_k \neq t_{j-(k-1)} t_{j-k} \dots t_{j-1} t_j$ ，当 $t_k \neq t_j$ 时应将模式向右滑动至以模式中的第 $\text{next}[k]$ 个字符和主串中的第 j 个字符相比较。若 $\text{next}[k] = k'$ ，且 $t_j = t_{k'}$ ，则说明在主串中第 $j+1$ 字符之前存在一个长度为 k' (即 $\text{next}[k]$) 的最长子串，与模式串中从第一个字符起长度为 k' 的子串相等。即 $\text{next}[j+1] = k' + 1$

同理，若 $t_j \neq t_k$ ，应将模式继续向右滑动至将模式中的第 $\text{next}[k']$ 个字符和 t_j 对齐，...，依此类推，直到 t_j 和模式串中的某个字符匹配成功或者不存在任何 k' ($1 < k' < j$) 满足等式： $t_1 t_2 \dots t_{k-1} t_{k'} = t_{j-(k'-1)} t_{j-k'} \dots t_{j-1} t_j$

则： $\text{next}[j] + 1 = 1$

根据上述分析， 求next函数值的算法如下：

```
void next(StringType t , int next[])
```

```
/* 求模式t的next串t函数值并保存在next数组中  */
```

```
{  int  k=1 , j=0 ; next[1]=0;
```

```
  while (k<t.length)
```

```
    {  if ((j==0)|| (t.str[k]==t.str[j]))
```

```
        {  k++ ; j++ ;
```

```
            if ( t.str[k]!=t.str[j] ) next[k]=j;
```

```
            else next[k]=next[j];
```

```
        }
```

```
    else next[j]=j ;
```

```
}
```

```
}
```

习 题 四

- (1) 解释下列每对术语的区别：空串和空白串；主串和子串；目标串和模式串。
- (2) 若x和y是两个采用顺序结构存储的串，写一算法比较这两个字符串是否相等。
- (3) 写一算法void StrRelace(char *T, char *P, char *S), 将T中第一次出现的与P相等的子串替换为S，串S和P的长度不一定相等，并分析时间复杂度。