第5章　串

串是一种特殊的线性表,除了具有普通线性表所具有的一切性质和操作以外,还具有一些特殊的性质和操作。串由字符构成,每个字符占用一个字节的存储空间。串可以用顺序存储结构和链式结构。其中,顺序存储结构的空间效率和时间效率更好。模式匹配是串的一个重要操作。BF 和 KMP 算法是两种经常使用的匹配算法。

5.1　串的定义

5.1.1　串的定义

串(String)是字符串的简称,它是由有限的字符组成的序列,一般记为 $s=$"$s_0 s_1 s_2 \cdots s_{n-1}$"。其中,$s$ 是串名,n 是串的长度。双引号括起来的字符序列称作串的值,每个字符可以是任意的 ASCII 码,一般是字母、数字、标点符号等可以在屏幕上显示的字符。串中字符的个数称为串的长度。零个字符的串称为**空串**(Null String)。

一个串中任意个连续的字符组成的子序列称为该串的**子串**,包含子串的串称为该子串的**主串**,例如

```
a="Hello Nanjing"
b="Hello"
c="Nanjing"
d="HelloNanjing"
```

其中,b 和 c 是 a 的子串,d 不是 a 的子串,因为它不是 a 连续的字符子序列。

一个字符在串中首次出现的位置称为该字符在串中的位置。当两个串的长度相等,且对应的字符也都相同的情况下,称作两个串相等,例如

```
a="Data Structure"
b="data Structure"
c="Data trurcture"
d="Data"
e="Data Structure"
```

其中,e 和 a 相等,字符串中的字母区分大小写,例如 d 和 D 是不同的两个字符。

串在许多软件中都有应用,例如,Word 的操作对象就是串类型对象。

5.1.2　串的抽象数据类型

串的逻辑结构和线性表十分相似,区别仅在于串的操作对象是字符,因此串的抽象数

据类型和线性表基本相同。然而串的基本操作和线性表差别很大。在线性表的基本操作中，大多以“单个元素”作为操作对象。而在串的基本操作中，通常以“串的整体”作为操作对象。

串的抽象数据类型定义如下：

```
ADT String
{
    数据对象:Di={ai|ai∈CharacterSet, i=1, 2, ...,n, n≥0}
    数据关系: R={<ai-1, ai>|ai-1,ai∈D, i=1, 2,..., n}
    基本操作:
        StrAssign(&T, chars)
         初始条件:chars 是字符串常量
         操作结果:生成一个其值等于 chars 的串
        StrCopy(&T, S)
         初始条件:串 S 存在
         操作结果:由串 S 复制得到串 T
        StrDelete(&S,pos,len)
         初始条件:串 S 存在
         操作结果:从 S 中删除自第 pos 个字符起长度为 len 的子串
        StrInsert(&S,pos,T)
         初始条件:串 S 和 T 存在
         操作结果:在 S 中第 pos 个字符之前插入串 T
        StrReplace(&S,pos,&T)
         初始条件:串 S 和 T 存在
         操作结果:在 S 中从第 pos 个字符开始用串 T 替换
        StrEmpty(S)
         初始条件:串 S 存在
         操作结果:若串 S 为空,则返回 True,否则返回 False
        StrCompare(S,T)
         初始条件:串 S 和 T 存在
         操作结果:若串 S>T,则返回值>0;若 S=T,则返回值=0;若 S<T,则返回值<0
        StrLink(S,T)
         初始条件:串 S 和 T 存在
         操作结果:将串 T 或 S 连接到 S 或 T 之后
        StrLength(S)
         初始条件:串 S 存在
         操作结果:返回 S 的元素个数
        ClearString(S)
         初始条件:串 S 存在
         操作结果:将 S 清为空串
}
```

5.2 串的存储结构与实现

5.2.1 串的顺序存储结构与实现

与线性表类似，串也可以用一组连续的存储单元(数组)依次存储串中的字符序列，串的存储空间分配在编译时完成，不能更改，因此需要事先定义一个固定长度的存储区域存储字符串。在串的顺序存储结构中，按照预先定义的尺寸，为每个定义的串变量分配一个固定长度的存储区，它的描述如下：

```
#include "stdio.h"
#define maximum 20
char str[maximum];
```

本节用一维数组的形式保存串，maximum 是数组预定义的长度。

【算法 5-1】 字符串的初始化(生成一个字符串，存入 h 数组中)。

具体算法如程序清单 5-1 所示。

程序清单 5-1

```
void StrInitialization(char *h)
{
    gets(h);
}
```

注意：gets 函数是输入字符串的常用函数，可以输入任意长的字符串。h 是字符型数组的首地址。

【算法 5-2】 字符串的初始化(统计串的长度)。

具体算法如程序清单 5-2 所示。

程序清单 5-2

```
int StringLength(char *h)
{
    int count=0;
    while(h[count]>0)
      count++;
    return count;
}
```

注意：h[count]>0 用来判断数组中的元素是否全部统计完。0 是 ASCII 码，相当于 null。正常的字符都是大于 0 的。有时系统会产生一些随机的字符，这些字符通常小于 0。

【算法 5-3】 字符串复制操作(将串 $h2$ 复制到串 $h1$，从第一个位置开始)。

具体算法如程序清单 5-3 所示。

程序清单 5-3

```
void StrCopy(char *h1,char *h2)     /*将 h2 复制到 h1*/
{
    int len;
    len=StringLength(h2);
    for(int i=0;i<len;i++)
        h1[i]=h2[i];
}
```

注意：无论 $h2$ 和 $h1$ 哪个长，只要这个长度小于预定义的 maximum 就可以。

【算法 5-4】 字符串连接操作(将串 $h2$ 连接到串 $h1$ 之后的位置上)。

思路：首先找到 $h1$ 的最后一个字符，从其后边的第一个位置开始存储 $h2$ 的元素。

具体算法如程序清单 5-4 所示。

程序清单 5-4

```
void StrLink(char *h1,char *h2)     /*将 h2 连接到 h1 之后,假设 h1 的长度足够*/
{
    int len1,len2;
    len1=StringLength(h1);
    len2=StringLength(h2);
    for(int i=0;i<len2;i++)
        h1[i+len1]=h2[i];
}
```

注意：$h1$ 数组的下标从 0 到 len1−1，因此 $h2$ 的第一个元素在 $h1$ 中的存储位置是 len1。

【算法 5-5】 子串的提取操作(由串 $h1$ 的第 i 个字符开始的 j 个字符，产生新串)。

思路：首先找到 $h1$ 的第 i 字符，将从这个字符开始的 j 个字符提取出来，保存到新的字符串数组 $h2$ 中。

具体算法如程序清单 5-5 所示。

程序清单 5-5

```
void SubStr(char *h1,char *h2,int i,int j)
{
    if(i<1 || i>StringLength(h1)||j<0||i-1+j>StringLength(h1))
                          /*从第 i 个字符开始到第 j 个字符的位置相当于 i-1+j*/
    {
        printf("参数不合理,无法完成操作\n");
        return;
    }
    for(int k=i-1;k<i+j-1;k++)     /*第 i 个位置相当于数组中的 i-1*/
        h2[k-i+1]=h1[k];
    return;
}
```

注意：需要先判断 i 和 j 是否在 $h1$ 的合法范围内。

【算法 5-6】 字符串的插入操作(将 $h2$ 插入到 $h1$ 的第 i 个位置处，产生的新串保存到 h 中)。

思路：需要先判断 i 是否在 $h1$ 的合法范围内。然后将 $h1$ 分为两部分，第 i 个位置之前的部分先存储到 h 中，再从 h 的第 i 个位置开始存储 $h2$ 中的元素，最后存储 $h1$ 剩余的元素。

具体算法如程序清单 5-6 所示。

程序清单 5-6

```
void InsStr(char * h1, char * h2,char * h,int i)
{
    if(i<1 || i>StringLength(h1)+1 )
    {
        printf("参数错误,无法完成操作\n");
        return;
    }
    for(int n=0;n<i-1;n++)           /* 将 h1 从第 1 个到第 i-1 个元素存储进 h 中 */
        h[n]=h1[n];
    for(n=0;n<StringLength(h2);n++)    /* 从 h 的第 i 个位置开始存储 h2 的元素 */
        h[n+i-1]=h2[n];
    for(n=i-1;n<StringLength(h1);n++)  /* 把 h1 的剩余元素存储到 h2 之后 */
        h[n+StringLength(h2)]=h1[n];
    return;
}
```

【算法 5-7】 字符串的删除操作(删除 $h1$ 的第 i 个字符开始的 j 个字符)。

思路：需要先判断 i 是否在 $h1$ 的合法范围内。先将 $h1$ 的前 $i-1$ 个元素存储到 h 中，然后再将从 $h1$ 的第 $i+j$ 开始的所有元素存储到 h 中。

具体算法如程序清单 5-7 所示。

程序清单 5-7

```
void DelStr(char * h1, char * h,int i,int j)
{
    if(i<1 ||i>StringLength(h1)|| i-1+j>StringLength(h1))
                                                 /* 第 i 个字符在数组中的下标是 i-1 */
    {
        printf("参数错误,无法完成操作\n");
        return ;
    }
    for(int n=0;n<i-1;n++)
        h[n]=h1[n];
    for(n=i+j-1;n<StringLength(h1);n++)
                                                 /* 从 h1 的第 i+j 个元素开始存储进 h 中 */
        h[n-j]=h1[n];
```

```
    return;
}
```

【算法 5-8】 字符串的替代操作(将 $h1$ 从第 i 个字符开始用 $h2$ 替代,假设 $h1$ 足够长)。

思路:需要先判断 i 是否在 $h1$ 的合法范围内。先找到第 i 个字符,从这个字符开始用 $h2$ 替代。

具体算法如程序清单 5-8 所示。

程序清单 5-8

```
void RepStr(char * h1, char * h2,int i)
{
    if(i<1 ||i>StringLength(h1)+1)      /* 第 i 个字符在数组中的下标是 i-1 */
    {
        printf("参数错误,无法完成操作\n");
        return ;
    }
    for(int n=0;n<StringLength(h2);n++)
                                        /* 从 h1 的第 i 个位置开始存储 h2 的元素 */
        h1[n+i-1]=h2[n];
    return;
}
```

注意:判断 i 值的范围是否合法时,只需要判断 $i<1$ 和 $i>\text{StringLength}(h1)+1$ 即可,即使替代后,$h1$ 的长度超过原 $h1$ 的长度,也可以接受。当 $i=\text{StringLength}(h1)+1$ 时,表示 $h2$ 从 $h1$ 之后的位置上插入,这相当于是两个串的连接操作。而且此算法与 StrCopy 算法相比,后者相当于从 $h1$ 的第一个元素开始用 $h2$ 代替,而本算法是从任意位置开始代替。

【算法 5-9】 串的显示操作(从 h 的第一个元素开始,依次输出,直到最后一个)。

具体算法如程序清单 5-9 所示。

程序清单 5-9

```
void DisplayString(char *h)
{
    for(int i=0;i<StringLength(h);i++)
        printf("%c",h[i]);
    printf("\n");
}
```

5.2.2 串的堆存储结构与实现

在顺序存储结构中,实现串操作的原操作为"字符序列的复制",操作的时间复杂度基于复制的字符序列的长度。还有一个问题,如果在串操作中出现串的长度超过预定义的

长度的情况，约定用截尾法处理，这种情况不仅在连接操作时可能发生，在串的其他操作中，如插入、置换等情况，也有可能发生。为了解决这个问题，可以采用堆分配的存储方式。

堆存储方式的特点是：仍以一组地址的连续的存储单元存放字符序列，但它们的存储空间是在程序执行过程中动态分配的。在 C 语言中，存在一个称为“堆”的自由存储区，可由 C 语言中的 malloc 函数和 free 函数管理，这两个函数在第 2 章中介绍过，此处不再赘述。为了处理方便，规定串的长度也作为存储结构的一部分。

5.2.3 串的块链存储结构与实现

与栈和队列的链式存储结构一样，串的链式存储结构就是把串值分别存放在链表的若干个结点的数据域中。串的链式存储结构分为单字符结点链和块链两种：**单字符结点链**就是每个结点的数据域只存储一个字符，如图 5-1(a)所示；**块链**就是每个结点的数据域包括若干个字符，如图 5-1(b)所示。

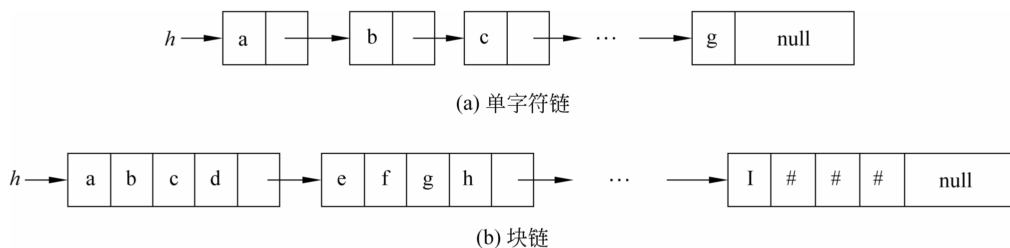


图 5-1　字符的链式存储

如图 5-1(b)所示，由于串的大小不一定等于结点大小的整数倍，则链表中的最后一个结点不一定全部被串值占满，此时可以补上“♯”或其他非串值字符。在链式存储方式中，结点大小的选择和顺序存储方式的格式选择一样都很重要，直接影响到串处理的效率。在各种串处理系统中，所处理的串一般都很长，这就要求考虑串值的存储密度问题，串的存储密度定义如下：

存储密度＝串值所占有的存储位/实际分配的存储位

显然，块链结构的存储密度高于一个结点仅存放一个字符的单链结构，通常，串的链式存储结构多采用块链结构。但是块链结构对字符的插入、删除操作极不方便，因此并不常用。块链的 C 语言类型定义如下：

```
#include "stdio.h"
#include "malloc.h"
#define maximum 5
typedef struct  blocklink
{
    char data[maximum];                    /* 数据域 */
    struct blocklink * next;               /* 指针域 */
```

```
} BLOCKLINK;                          /* 块链类型名 */
```

由上述定义可以看出,块链和单字符节点链的定义基本一样,区别仅在于块链的数据域是 char 型数组,且长度不为 1,可以自己定义。

5.3 串的模式匹配算法

设 s 和 t 是给定的两个串,且 s 的长度大于 t 的长度,在 s 中找到等于 t 的子串的过程称为**串的模式匹配**,如果找到,则称为匹配成功,否则,匹配失败。模式匹配是一种重要的串操作,在文字处理等软件中有着广泛的应用。串的模式匹配也称为子串的定位操作。

5.3.1 简单的模式匹配算法——BF 算法

BF 算法,又称**蛮力算法**,它的主要思想是:从主串 $s=$"$s_0 s_1 s_2 \cdots s_{n-1}$"的第一个字符开始和模式串 $t=$"$t_0 t_1 \cdots t_{m-1}$"的第一个字符开始比较,若相等则继续比较后续字符;否则从主串 s 的第二个字符开始重新与模式串 t 的第一个字符比较,按照这个方式,继续下去。如果模式串 t 和主串 s 的某一段连续子串相等,则匹配成功,并返回模式串 t 的第一个字符在主串中的位置;若匹配不成功,则返回-1。

【例 5-1】 设主串 $s=$"abceabcabcacbab",模式串 $t=$"abcab",s 的长度是 15,t 的长度是 5。用指针 i 指示主串 s 的当前比较字符位置,用指针 j 指向模式串 t 的当前比较字符的位置,分析其模式匹配过程。

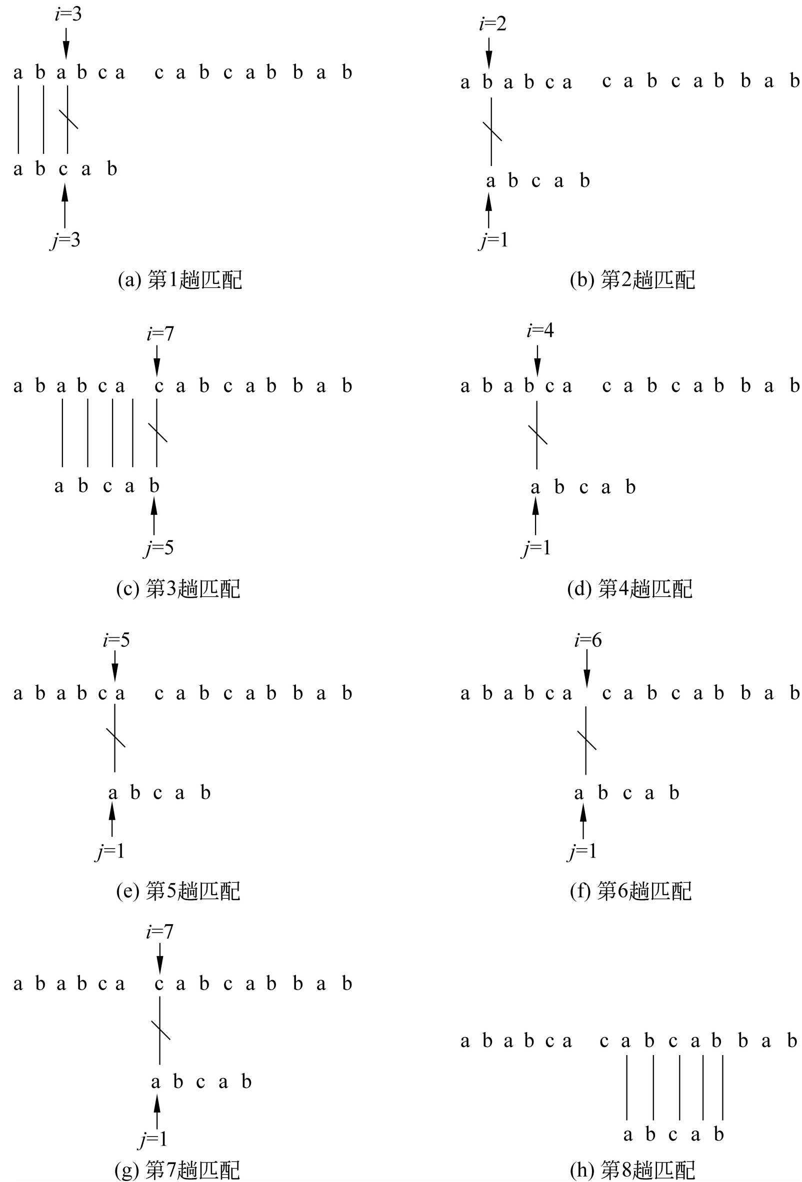
分析:其模式匹配的过程如图 5-2 所示。

由图 5-2 可知,BF 算法简单、易于理解,但时间效率低。这是因为在主串和子串已有相当多个字符相等的情况下,只要有一个字符不相等,就需要重新将主串的比较位置后移一位,之前做过的比较工作也需要重新进行。因此,BP 算法最好情况的时间复杂度是 $O(m)$,即主串的前 m 个字符正好与模式串完全匹配;最坏情况的时间复杂度是 $O(n\times m)$,这种情况是模式串的前$(m-1)$个字符序列和主串的相应位置总是相等,而模式串的第 m 个字符和主串的相应位置处的元素总是不等。

具体实现如程序清单 5-10 所示。

程序清单 5-10

```
int BF_Match(char *s, char *t)            /* BF匹配方法 */
{
    int i=0,j=0;                          /* i表示主串的指针,j表示模式串的指针 */
    int k;
    while(i<StringLength(s) && j<StringLength(t))
    {
        if(s[i]==t[j])                    /* s[i]和t[j]相等,继续向下匹配 */
        { i++; j++;}
        else
        {j=0;i=i-j+1;}
```



(a) 第1趟匹配 (b) 第2趟匹配

(c) 第3趟匹配 (d) 第4趟匹配

(e) 第5趟匹配 (f) 第6趟匹配

(g) 第7趟匹配 (h) 第8趟匹配

图 5-2 BF 算法匹配流程

```
/* 如果不相等,则主串退回到第 i-j+1 个元素;模式串 t 从第一个位置开始重新匹配 */
    }
    if(j>=StringLength(t))
/* 匹配成功,返回模式串的第一个元素在在主串中的位置,用 k 返回 */
         k=i-StringLength(t)+1;
    else                                        /* 匹配不成功,返回-1 */
        k=-1;
    return k;
}
```

5.3.2 改进的模式匹配算法——KMP 算法

模式匹配的另一种算法是由 D. E. Knuth、J. H. Morris 以及 V. R. Pratt 共同发现的，因此称为**克努特—莫里斯—普拉特算法**，简称 **KMP 算法**。它是一种改进型的 BF 算法。改进之处在于：该算法主要消除了主串指针（i 指针）的回溯，利用已经得到的部分匹配结果将模式串右滑尽可能远的一段距离再继续比较，从而使算法效率有某种程度的提高。KMP 算法的时间复杂度是 $O(n+m)$。

BF 算法速度慢的主要原因是回溯，而这些步骤并不是必需的。而 KMP 算法中，串指针 i 不回溯，由模式串指针 j 退到某一个位置 k 上，使模式串 t 中的 k 之前的($k-1$)个字符与 s 中的 i 之前的($k-1$)个字符相等，这样可以减少匹配的次数，从而提高效率。

在图 5-3 中，第一次匹配时，s_3 和 t_3 不相等，BF 算法需要将 s_3 回溯到 s_s，再重新和 t 比较，而由图 5-3(a)可知，s_2 和 t 也不相等，因此，图 5-3(b)中的第二次匹配是没有必要的，应该直接从 s_3 开始和 t 比较。与此相同的是，图 5-3(c)中的 s_7 和 t_5 不相等，之后，回溯到 s_4，再重新与 t_1 比较，按这个过程一直持续到 s_8，才和 t_1 又重新相等，因此从 s_4 到 s_7 的比较都是多余的，完全可以省去，让 t_1 在和 s_3 比较之后，直接和 s_8 进行比较。因此如何省略 s_2 到 s_7，直接找到 s_8 是算法的关键。可将如图 5-3 所示的匹配过程改进为如

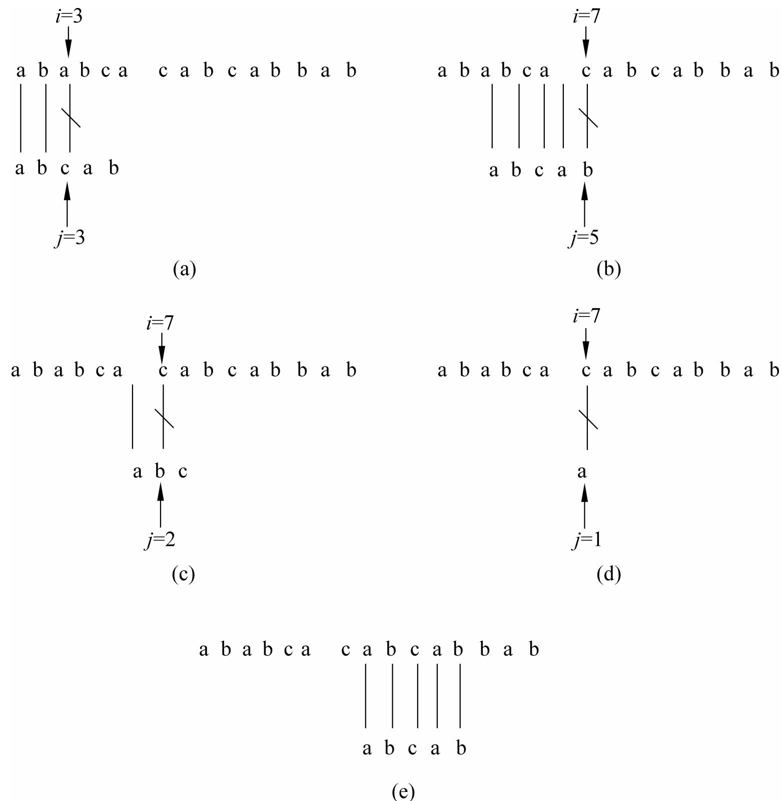


图 5-3 KMP 算法的匹配过程