

第四章 串

刘亮亮

2020 年 4 月 13 日

(声明：本讲义为草稿，限内部使用，版权所有，未经同意，不得外传)

- 要点：本章主要介绍串的定义与存储结构，介绍串的常用的算法，以及一个重要的算法——模式匹配算法。
 - 重点：1.串的存储；2.串的模式匹配算法；
 - 难点：1.KMP算法
-

串也就是我们常说的字符串，是有零个或多个字符构成的有限序列，又称为字符串。字符串也是一种线性结构。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。我们在实际应用中需要做大量的字符串处理操作，比如在自然语言处理(Natural Language Processing, NLP)领域需要处理的都是字符串。

因此串是一种常用的数据结构，也是应用广泛的数据结构。

1 串的定义与表示

1.1 串的相关定义

一般地，我们将字符串记为 $S = "a_1a_2...a_n"$ ，其中

- (1) S 称为串名；
- (2) $a_i(1 \leq i \leq n)$ 是单个字符，可以是数字、字母或其它字符。
- (3) 双引号括起来的内容 $a_1a_2...a_n$ 称为串值。
- (4) 串值中包含的字符的个数称为串 S 的长度。

定义 1. 空串：没有任何字符的串称为空串，其长度为0。

定义 2. 空格串：又称空白串，是有一个或多个空格构成的串，空格的个数即为空格串的长度。

定义 3. 子串与主串：一个串中任意个连续字符组成的子序列称为该串的子串。对应的，包含子串的串相应地称为主串。

定义 4. 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 1. $S = \text{"How are you"}, sub = \text{"are"}$ ， S 中包含 sub ，因此 sub 是 S 的子串，其主串 S 的位置是4（注意下标是从0开始）。

定义 5. 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。

例如， $S_1 = \text{"Hello World"}$ 与 $S_2 = \text{Hello World}$ 这两个字符串就是相等的。

定义 6. 串常量与串变量：串的常量和整型常量、字符常量等一样，在运行过程中不能更改的量，在程序中只能被引用但是不能被修改，只读不可以写。而串变量和其他类型的变量一样，只是可以修改的。

如：“Hello World”就是一个串常量，存在计算机的寄存器中是不允许修改的。而 $S = \text{"Hello World"}$ ， S 就是一个变量，比如可以修改 $S = \text{"Hi"}$ 。其实 $S = \text{"Hello World"}$ 中的“Hello World”就是串常量。

误区：当将 $S = \text{"Hi"}$ ，不就修改了“Hello World”么？其实这种认识是错误的，只是定义了另一个串常量“Hi”， S 只是指向了“Hi”，“Hello World”仍然没有发生改变，只是 S 不指向它而已。

1.2 串的抽象数据类型

因为串也是一种线性结构，也是满足一对一的线性关系，因此类似于线性表，我们可以定义串的抽象数据类型，只不过串有一些特定的操作，比如求子串、求串长、串连接、串的模式匹配等算法。具体的定义如下：

```

ADT String{
    数据对象: $D = \{a_i | a_i \in CharSet\}$ 
    数据关系: $R = \{ \langle a_i, a_{i+1} \rangle \}$ 
    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,  $1 \leq pos \leq strLength(s)$  且  $0 \leq len \leq strLength(s) - pos + 1$ 。
        操作结果: 用sub返回串s的第pos个字符起长度为len的子串。
        index(s,t,pos)
        初始条件: 串s,t都存在,  $0 \leq pos \leq strLength(s) - strLength(t)$ 
        操作结果: t在s中指定pos位置开始起第一次匹配的首字符的位置,
        未找到返回-1.
    }End String.

```

串的操作有很多, 但是其中串赋值(strAssign)、串比较(strConcat)、求串长(strLength)、串连接(strConcat)、求子串(subString)构成了串的最小操作子集。所谓最小操作子集, 也就是说其它的操作都可以通过最小操作子集的中某个或某几个组合来实现。

例如, 串的模式匹配操作index(s,t,pos)就可以通过最小操作子集中的串比较、求串长、求子串来实现。具体的算法思想如下:

- 1) 从主串s从第i个位置 (i的初始值为pos) 取长度为strLength(t)的子串sub;
- 2) 将sub与t就行比较, 如果sub和t相等, 则返回i的值, 表示模式匹配成功。如果不相等, 则i增, 重复步骤1), 直到取不出和t长度相等的子串为止, 返回-1表示模式匹配失败。

从上面的算法思想可以看到, 串的模式匹配操作只用到了求串长、求

子串以及串比较等三个基本操作就可以实现。

具体的算法描述如下：

```
1  int index(string s, string t, int pos)
2  {
3      n=strLength(s);
4      m=strLength(t);
5      //t为非空串，若主串s中第pos个字符之后存在与t相等的子串，则返回
        第一个在s中的位置，否则返回-1
6      if (pos < 0 || pos > n-m)
7      {
8          return -1;
9      }
10     i=pos;
11     while (i < n-m+1)
12     {
13         subString(sub, S, i, m);
14         if (strCompare(sub, T) != 0)
15         {
16             ++i;
17         }
18         else
19         {
20             return i;
21         }
22     } //end while
23     return 0;
24 }
```

2 串的存储结构

串是一种特殊的线性表，其存储结构和线性表的类似，但又有区别，这取决于对串的进行什么的操作，不同的操作用不同的存储结构算法效率是

不一样的。

根据串的特点，串有以下三种存储结构：

- 定长顺序存储结构：将串定义成字符数组，利用串名可以直接访问串值。串的存储空间在编译时确定，其大小不能改变。
- 堆分配存储结构：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。
- 块链存储结构：是一种链式存储结构，每一个结点会分配一个一个连续的存储单元来存放一个或多个字符。如果只是存放一个字符，那么结构就是单链表。

2.1 定长顺序存储结构与实现

定长顺序存储结构是一种顺序存储结构，是用一组连续的存储单元来存放串中的字符序列，所谓的定长，是直接使用定长的字符数组，数组的上界预先确定，在运行过程中不能进行修改。

存储描述如下：

```
1 #define MAX_STRLEN 256
2 typedef struct String{
3     char str[MAX_STRLEN];
4     int length;
5 }String;
```

2.1.1 串连接操作实现

利用定长顺序存储结构实现串连接操作`strConcat(s,t)`，其操作是将串`s`连接在串`t`的末尾。由于串采用定长顺序存储结构，其长度是固定的，因此在做串连接的时候有以下几种情况：

- 如果`strLength(s) == MAX_STRLEN`，则`t`的内容完成无法进行连接在`s`后面，称为全部截断`t`；

- 如果 $strLength(s) + strLength(t) > MAX_STRLEN$ ，则 t 的部分内容可以连接到 s 的后面，称为部分截断；
- 如果 $strLength(s) + strLength(t) \leq MAX_STRLEN$ ，则 t 的全部内容都可以连接到 s 的后面，串连接成功。

具体的算法描述如下：

```

1 Status strConcat (String& s, String& t)
2 {
3     //将串t联结到串s之后，结果仍然保存在s中
4     int i, j;
5     if (s.length == MAX_STRLEN)
6     {
7         uncat = false;
8     }
9     else if (s.length + t.length <= MAX_STRLEN)
10    {
11        for (i = 0; i < t.length; i++)
12        {
13            s.str[s.length + i] = t.str[i]; //t拼接在s后面
14        } //end for
15        s.length = s.length + t.length; //修改s的长度
16        uncat = true;
17    }
18    else
19    {
20        for (int i = 0; i < MAX_STRLEN - s.length; i++)
21        {
22            s.str[s.length + i] = t.str[i];
23        }
24        s.length = MAX_STRLEN;
25        uncat = false;
26    } //end if

```

```

27     return uncat;
28 }

```

时间复杂度为 $O(m)$ ， m 为串 t 的长度。

2.1.2 求子串操作实现

求子串操作是字符串最常用的操作之一，是从指定位置 pos 开始取长度为 len 的子串，算法名为 $subString(s, pos, len, sub)$ ，参数描述：

- s :主串
- pos :从 pos 位置开始取
- len :取子串的长度
- sub :返回子串

算法思想：

Step1:首先判断参数是否合法($pos < 1 || pos > s.length || len < 0 || len > (s.length - pos + 1)$)

Step2: 设置子串的长度为 len

Step 3:逐个复制字符到 sub 中，从 s 中第 pos 位置开始依次取出字符赋值到 sub 中，取长度为 len 个。

算法描述如下：

```

1  Status subString(String s, int pos, int len, String &sub)
2  {
3      int k, j;
4      //判断参数是否合法
5      if (pos < 1 || pos > s.length || len < 0 || len > (s.length - pos
6          + 1))
7      {
8          return ERROR;
9      }
10     //求得子串长度
11     sub->length=len;

```

```

11 //逐个字符复制求得子串
12 for (j=0,k=pos;k<=pos+len-1;k++,j++)
13 {
14     sub->str[j]=s.str[k];
15 }
16 return OK ;
17 }

```

算法时间复杂度为 $O(len)$ 。

2.2 堆分配存储结构

堆分配存储结构是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,也是顺序存储结构,但是存储空间是动态分配的。通过使用C语言的动态存储分配函数`malloc()`和`free()`来进行内存管理。

堆分配存储结构仍然是用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,是动态的,变长的,这是和定长顺序存储结构不一样的,当存储空间不够的时候可以通过动态分配,重新分配存储空间。因此串连接操作如果采用堆分配存储结构来实现,不会出现部分截断和全部截断的问题。

串的堆分配存储结构的类型定义:

```

1 typedef struct HString
2 {
3     char *ch; //若非空,按长度分配,否则为NULL
4     int length; //串的长度
5 } HString;

```

2.2.1 串连接操作的实现

堆分配存储结构下的串的连接操作的定义和定长顺序表示下的串连接不大相同,定义是`strConcat(t,s1,s2)`,用串`t`返回将`s1`和`s2`连接的结果,由于存储空间是动态分配,因此一开始对`t`分配存储空间的时候,分配的大小为`s1`和`s2`的长度之和,因此在堆分配存储结构下的串连接操作不会出

现截断。分配好空间后，将s1依次赋值到t，然后将s2依次赋值给t，最后对s1和s2释放内存空间。算法描述如下：

```
1 Status strConcat(HString &T,HString &s1,HString &s2)
2 {
3     int k, j , t_len;
4     if (T.ch) free (T); /* 释放旧空间 */
5     t_len=s1->length+s2->length;
6     if ((p=(char*) malloc ( sizeof ((char)*t_len ))==NULL)
7     {
8         cout<<"系统空间不够，申请空间失败!"<<endl;
9         return ERROR;
10    }
11    for (j=0;j<s->length;j++)
12    {
13        //将串s复制到串T中
14        T->ch[j]=s1->ch[j] ;
15    }
16    for (k=s1->length , j=0;j<s2->length;k++,j++)
17    {
18        // 将串s2复制到串T中
19        T->ch[k]=s1->ch[j]
20    }
21    free (s1->ch) ;
22    free (s2->ch) ;
23    return OK ;
24 }
```

2.2.2 求子串操作的实现

下面实现对分配存储下的求子串操作，和定长顺序存储的实现思想类似，首先判断参数的合法性，然后根据取子串的长度分配存储空间，然后从pos开始一一赋值就可以实现取子串操作。具体的实现如下：

```

1 Status subString(HString &s,HStrng &sub,int pos,int
    len)
2 {
3     if(pos<0||pos>=s.length||len<0||len>s.length-pos)
4     {
5         return ERROR;
6     }
7     if(sub.ch) free(sub.ch);
8     if(len==0)
9     {
10        sub.ch=NULL;
11        sub.length=0;
12    }
13    else
14    {
15        sub.ch=(cha*) malloc(len*sizeof(char));
16        for(int i=pos;i<=pos+len;i++)
17        {
18            sub.ch[i-pos]=s.ch[i];
19        }
20        sub.length=len;
21    }
22    return OK;
23 }

```

自行实现串赋值操作strAssign(HString& s,char* chrs)。

2.3 块链存储结构

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，结点也分为data域和next域。但字符串由很多字符构成，每个字符占用1个字节，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。

图1为块大小为3的串的块链式存储结构示意图。

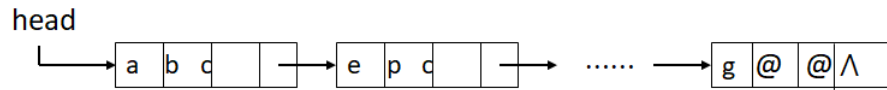


图 串的块链式存储结构示意图

Figure 1: 块链存储结构示意图

串的块链式存储的类型定义包括块结点和块链串两种结构，如下所示：

```
1 #define BLOCK_SIZE 4
2 typedef struct BNODE
3 {
4     char data[BLOCK_SIZE];
5     struct BNODE *next;
6 }BNODE;
7 typedef struct LString
8 {
9     BNODE *head,*tail; //头、尾指针
10    int len ; //当前长度
11 }LString;
```

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，因此，在这种存储结构下进行插入和删除等操作就会相当复杂。

3 串的模式匹配算法

定义 7. 模式匹配：指模式串 T 在主串 S 中的定位，也称为串匹配或字符串匹配。在主串 S 中能够找到模式串 T 则匹配成功，否则，称模式串 T 在主串 S 中不存在。

串的模式匹配应用非常广泛，在很多软件或应用里面广泛使用，比如在文本编辑程序（如微软的word）的查找功能，查找某个单词在文本中出现的位置等。

串的模式匹配算法是一个较为复杂的串操作过程，也提出了很多模式匹配算法，这里主要介绍两种模式匹配算法，一种是简单模式匹配算法（即：Brute-Force算法），一种是KMP算法，KMP算法是一种简单模式匹配算法的改进算法，是由D.E.Knuth，J.H.Morris和V.R.Pratt共同提出。

下面介绍这两个算法。首先定义相关的符号。

S 为目标串（主串）： $S = "s_0s_1s_2...s_{n-1}"$ （长度为 n ）

T 为模式串（子串）： $T = "t_0t_1t_2...t_{m-1}"$ （长度为 m ）

一般地，主串的长度要大于等于子串的长度，即： $n \geq m$

3.1 简单模式匹配算法

简单模式算法，即Brute-Force算法，暴力搜索，是比较简单的一种字符串匹配算法，在处理简单的数据时候就可以用这种算法，完全匹配。

算法思想：从目标串 S 的第一个字符起和模式串 T 的第一个字符进行比较，若相等，则继续逐个比较后续字符，否则从串 S 的第二个字符起再重新和串 T 进行比较。依此类推，直至串 T 中的每个字符依次和串 S 的一个连续的字符序列相等，则称模式匹配成功，此时串 T 的第一个字符在串 S 中的位置就是 T 在 S 中的位置，否则模式匹配不成功。

图2为简单模式匹配过程的示意图。



Figure 2: 简单模式匹配过程

算法描述如下：

```

1  int index(String S, String T)
2  {
3      if (S.length < 1 || T.length < 1)
4          return -1;
5      if (S.length < T.length)
6      {
7          return -1;
8      }
9      int i=0, j=0;
10     while (i < S.length && j < T.length)
11     {
12         if (S.ch[i] == T.ch[j])

```

```

13      {
14          ++i;
15          ++j;
16      }
17      else
18      {
19          i = i-j+1;
20          j = 0;
21      }
22  }
23  if(j > T.length)
24  {
25      return i-T.length;
26  }
27  return -1;
28  }

```

算法分析：当模式匹配失败的时候，指针需要回溯，i指针回溯到S的 $i - j + 2$ 的位置上，而j要回溯到T的开始位置。简单模式匹配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。但是在实际运行过程中，该算法的执行时间近似于 $O(n + m)$ ，最坏的情况下（也就是每一轮比较比较对最后才不相等，j每次都回溯到最开始的位置），该算法的时间复杂度为 $O(n * m)$ ，因此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。

但是有些情况下，算法效率低下，比如说，S=”00000000000000000000
000000000000000000000000000001”,T=”00000001”，每一趟比较到T最后一个字符时候，指针i需要回溯到i-6的位置，然后重新开始比较，整个过程i需要回溯46次，循环 $46 * 8(\text{index} * \text{m})$ 。

根据上面的分析，简单模式匹配算法如果匹配不成功，模式串的指针需要不断去回溯，这是算法效率低的地方，那么能否改进呢？——也就是说模式串的指针能否不回溯或者不回溯到开始位置。

3.2 KMP算法

KMP算法是简单模式匹配算法的改进之一，该算法是由D.E.Knuth，J.H.Morris和 V.R.Pratt同时发现。算法的时间复杂度可以达到 $O(m+n)$ 。

思路：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，继续进行比较。为什么可以回溯，这“部分匹配”的结果如何得到？

3.2.1 示例图解

首先，看一个例子：

S="BBC ABCDAB ABCDABCDABDE"

T="ABCDABD"

(1)首先，主串"BBCABCDABABCDABCDABDE"的第一个字符(指针i指向,i=0)与模式串"ABCDABD"的第一个字符（指针j指向,j=0），进行比较。因为'B'与'A'不相等，所以主串指针i后移一位(i=1)，模式串保持不动(j=0)。

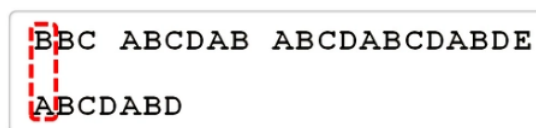


Figure 3: KMP匹配过程

(2)主串的当前的i指向的'B'与模式串j指向的'A'又不相等，主串指针i再往后移(i=2)。

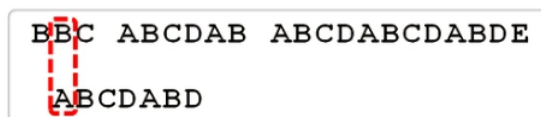


Figure 4: KMP匹配过程

(3)就这样，直到主串有一个字符，与模式串的第一个字符相同为止，此时i=3,j=0

Figure 5: KMP匹配过程

(4)接着移动i指针($i=4$)和j指针($j=1$),进行比较,相等则移动指针i和j。

Figure 6: KMP匹配过程

(5)直到主串有一个字符,与模式串对应的字符不相同为止或者所有元素都相等表示匹配成功。

Figure 7: KMP匹配过程

(6)如果遇到不相等,最自然的反应是,主串指针i回到当前起始位置的后面一个位置($i-j+1$),再和模式串的开始(j回到0)逐个比较。这样做虽然可行,但是效率很差,因为你要把"搜索位置"移到已经比较过的位置,重比一遍。

Figure 8: KMP匹配过程

(7)其实一个基本事实是，当主串中的空格与模式串中的'D'不匹配时，其实是已经知道前面六个字符是"ABCDAB",否则也不会匹配到最后一个才不相等。KMP 算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，而是继续把它向后移，这样就提高了效率。

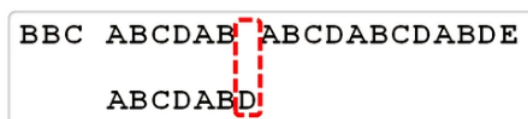


Figure 9: KMP匹配过程

(8)问题是：如何做到这一点呢？可以针对模式串，设置一个跳转数组int next[]，这个数组记录i和j指向的字符不相等时，j需要滑动到哪个位置上重新开始比较。这个数组是怎么计算出来的，后面再介绍，这里只要会用就可以了。

j	0	1	2	3	4	5	6	7
模式串	A	B	C	D	A	B	D	'\0'
next[j]	-1	0	0	0	0	1	2	0

(9)当主串的空格与模式串的D不相等时，前面六个字符"ABCDAB"是匹配的。根据跳转数组可知，不匹配处D的next值为2，因此接下来从模式串j=next[j] (j=6) 从下标为2的位置开始匹配。

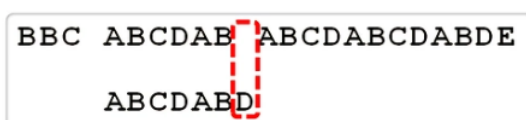


Figure 10: KMP匹配过程

(10)此时空格与字符C不相等，C处的next值为0，因此接下来模式串从下标为0处开始匹配。

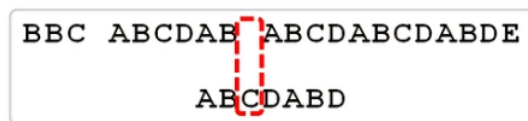


Figure 11: KMP匹配过程

(11)因为空格与A不匹配，此处next值为-1，表示模式串的第一个字符就不匹配，那么主串指针i直接往后移一位。

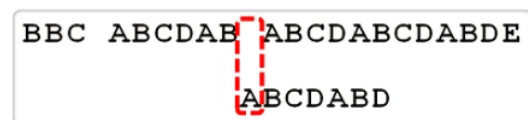


Figure 12: KMP匹配过程

(12)逐位进行比较，直到发现主串的C（ $i=17$ ）与子串的D（ $j=6$ ）不匹配。于是，下一步子串的j从下标为 $2(\text{next}[6])$ 的地方开始匹配。

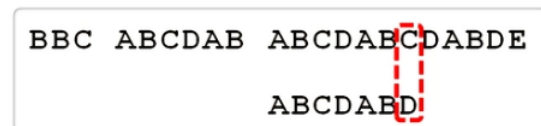


Figure 13: KMP匹配过程

(13)逐位比较，直到模式串的最后一位，发现完全匹配，于是搜索完成。

3.2.2 next数组求解方法

问题是：next数组是如何求出来的？

next数组的求解基于“前缀”和“后缀”，即 $\text{next}[j]$ 等于 $T[0]...T[j-1]$ 最长的相同前后缀的长度。上面定义的next表格如下。

j	0	1	2	3	4	5	6	7
模式串	A	B	C	D	A	B	D	'\0'
next[j]	-1	0	0	0	0	1	2	0

- $j = 0$ ，对于模式串的首字符，我们统一为 $\text{next}[0] = -1$ ；
- $j = 1$ ，前面的字符串为A，其最长相同真前后缀长度为 0，即 $\text{next}[1] = 0$ ；
- $j = 2$ ，前面的字符串为AB，其最长相同真前后缀长度为 0，即 $\text{next}[2] = 0$ ；
- $j = 3$ ，前面的字符串为ABC，其最长相同真前后缀长度为 0，即 $\text{next}[3] = 0$ ；
- $j = 4$ ，前面的字符串为ABCD，其最长相同真前后缀长度为 0，即 $\text{next}[4] = 0$ ；
- $j = 5$ ，前面的字符串为ABCDA，其最长相同真前后缀为A，即 $\text{next}[5] = 1$ ；
- $j = 6$ ，前面的字符串为ABCDAB，其最长相同真前后缀为AB，即 $\text{next}[6] = 2$ ；
- $j = 7$ ，前面的字符串为ABCDABD，其最长相同真前后缀长度为 0，即 $\text{next}[7] = 0$ 。

大家此时有疑问：为什么根据最长相同的前后缀的长度就可以实现在不匹配情况下的跳转呢？举个代表性的例子：假如当 $j = 6$ 时不匹配，此时我们是知道其位置前的字符串为”ABCDAB”，仔细观察这个字符串，首尾都有一个”AB”，既然在 $j=6$ 处的'D'和主串对应位置不匹配，我们为何不直接把 $j=2$ 处的字符'C'拿过来继续比较呢，因为模式串从开始有一个最大前缀”AB”和最大后缀”AB”相等，而这个”AB”就是”ABCDAB”的最长相同前后缀，其长度2正好是跳转的下标位置。

这里可能存在疑问，若在 $j=5$ 时匹配失败，按照分析的思路，此时应该把 $j=1$ 处的字符拿过来继续比较，但是这两个位置的字符是一样的啊，都是B，既然一样，拿过来比较不就是无用功了么？因此这个next算法还未优化，这里先不介绍。

先看next数组的代码实现，如下：

```
1 // P 为模式串，下标从0开始
```

```

2 void GetNext(String T, int next[])
3 {
4     int len = T.length;
5     int i = 0;    //T的下标
6     int j = -1;
7     next[0] = -1;
8
9     while (i < len)
10    {
11        if (j == -1 || T[i] == T[j])
12        {
13            i++;
14            j++;
15            next[i] = j;
16        }
17        else
18            j = next[j];
19    }
20 }

```

上面的算法就是求next函数，分析一下这个算法：

(1) 算法中的i和j的作用是什么？

i 和 j 就像是两个“指针”，一前一后，通过移动它们来找到最长的相同前后缀。

(2) if...else...语句里做了什么？

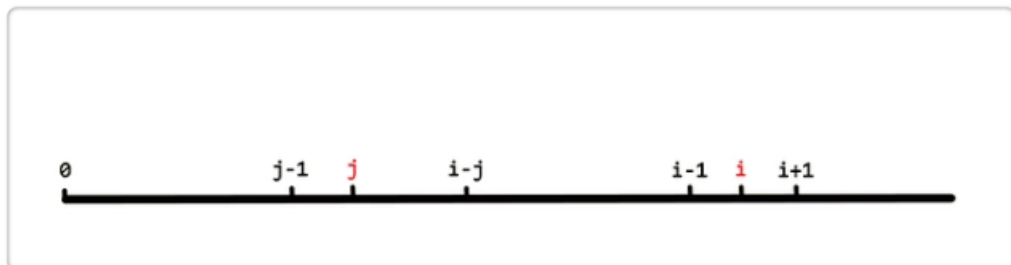


Figure 15: KMP匹配过程

假设 i 和 j 的位置如图15，由 $\text{next}[i] = j$ 得，也就是对于位置 i 来说，区段 $[0, i - 1]$ 的最长相同前后缀分别是 $[0, j - 1]$ 和 $[i - j, i - 1]$ ，即这两区段内容相同。

按照算法流程，if ($P[i] == P[j]$)，则 $i++$; $j++$; $\text{next}[i] = j$ ；若不等，则 $j = \text{next}[j]$ ，见图16：

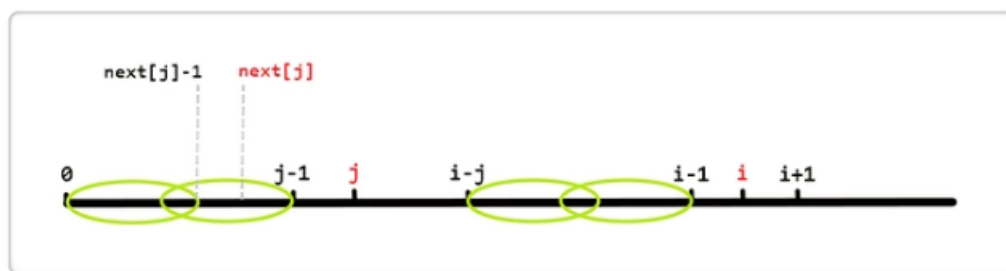


Figure 16: KMP匹配过程

$\text{next}[j]$ 代表 $[0, j - 1]$ 区段中最长相同前后缀的长度。如图16，用左侧两个椭圆来表示这个最长相同前后缀，即这两个椭圆代表的区段内容相同；同理，右侧也有相同的两个椭圆。所以else语句就是利用第一个椭圆和第四个椭圆内容相同来加快得到 $[0, i - 1]$ 区段的相同前后缀的长度。

if 语句中条件 $j == -1$ 是什么作用？第一，程序刚运行时， j 是被初始为 -1 ，直接进行 $P[i] == P[j]$ 判断无疑会边界溢出；第二，else 语句中 $j = \text{next}[j]$ ， j 是不断后退的，若 j 在后退中被赋值为 -1 （也就是 $j = \text{next}[0]$ ），在 $P[i] == P[j]$ 判断也会边界溢出。综上两点，其意义就是为了特殊边界判断。

其实本质上，next数组求解算法就是模式匹配过程，其主串和模式串都是T本身，在T找最长的前后缀的过程。

3.2.3 KMP算法实现

完整的KMP算法如下所示：

```
1 #include <iostream>
2
3 using namespace std;
```

```

4
5 //P为模式串,下标从0开始
6 void GetNext(String T, int next[])
7 {
8     int len = T.length;
9     int i = 0;    //T的下标
10    int j = -1;
11    next[0] = -1;
12
13    while (i < len)
14    {
15        if (j == -1 || T[i] == T[j])
16        {
17            i++;
18            j++;
19            next[i] = j;
20        }
21        else
22            j = next[j];
23    }
24 }
25
26 /* 在 S 中找到 P 第一次出现的位置 */
27 int KMP(String S, String T)
28 {
29     int i = 0;    // S 的下标
30     int j = 0;    // T 的下标
31     int s_len = S.length;
32     int t_len = T.length;
33     int next[t_len];
34     GetNext(T, next);
35
36     while (i < s_len && j < t_len)

```

```

37     {
38         //T的第一个字符不匹配或S[i] == T[j]
39         if (j == -1 || S[i] == T[j])
40         {
41             i++;
42             j++;
43         }
44         else
45             j = next[j];    //当前字符匹配失败，进行跳转
46     }
47
48     if (j == t_len)    // 匹配成功
49         return i - j;
50
51     return -1;
52 }

```

3.2.4 next数组的优化

j	0	1	2	3	4	5	6	7
模式串	A	B	C	D	A	B	D	'\0'
next[j]	-1	0	0	0	0	1	2	0

根据前面的分析，我们知道若在 $i = 5$ 时匹配失败，按照next代码，此时应该把 $j = 1$ 处的字符拿过来继续比较，但是这两个位置的字符是一样的，都是B，既然一样，拿过来比较就没有意义了？因此next函数的实现还需要进一步优化。

```

1  /* P 为模式串，下标从 0 开始 */
2  void GetNextval(string T, int nextval[])
3  {
4      int t_len = T.size();
5      int i = 0;    // T 的下标
6      int j = -1;

```

```

7      nextval[0] = -1;
8
9      while (i < t_len)
10     {
11         if (j == -1 || T[i] == T[j])
12         {
13             i++;
14             j++;
15
16             if (T[i] != T[j])
17                 nextval[i] = j;
18             else
19                 nextval[i] = nextval[j]; //相同就继续往前
                                           找前缀
20         }
21         else
22             j = nextval[j];
23     }
24 }

```

4 小结

本章介绍了串的定义、存储结构与操作，主要包括：

- 串的定义与特点：也是一种线性结构
- 串的三种存储结构：定长顺序存储、堆分配存储、块链存储结构
- 串的基本操作的实现：串连接、取子串等操作的实现
- 串的模式匹配算法与实现：简单模式匹配算法、KMP算法
- 串的应用

5 Next

下一章介绍另外一种数据结构———数组与广义表，主要内容如下：

- (1)数组的定义与表示
- (2)矩阵的存储与表示
- (3)广义表的定义与存储

6 习题

一、填空题

- (1) 两个串相等的充分必要条件是_____。
- (2) 空串是_____，其长度为_____；空格串是_____，其长度为_____。
- (3) 串的存储结构主要包括_____、_____、_____。
- (4) 设有两个串s和t,则求t在s中首次出现的位置的操作是_____。

二、求下面串的next数组：

- (1)T="ababaaaba",next=?
- (2)T="aaabaacab" next=?
- (3)T="abcdabbcd" next=?

三、算法题：

- 1. 字符串采用堆存储结构，实现字符串赋值操作strAssign(HString &s,char* ch)。
- 2. 串s采用定长顺序存储结构，实现删除s中第pos个字符开始的len个字符操作delete(String &s,int pos,int len)。
- 3. 输入两个字符串s和t，编写一个函数，设计算法求串s和t的最长公共子串。

7 附录

```
1 #pragma once
2 #include "predefine.h"
3 #define MAX_STRLEN 256
4 class String
5 {
6 public:
7     String();
8     ~String();
9     String(const char* ch);
10
11 public:
12     Status concat(String& t);
13     int size();
14     void print();
15     Status strcpy(String t);
16     static Status strcpy(String& t, String& s);
17     Status empty();
18     int strempy(String t);
19     Status clear();
20     Status substring(String& sub, int pos, int len
21         );
22     int index(String t, int pos);
23 private:
24     char str[MAX_STRLEN];
25     int length;
26 };
27 #include "String.h"
28
29
30
```

```

31 String::String()
32 {
33 }
34
35
36 String::~~String()
37 {
38 }
39 String::String(const char *ch)
40 {
41     if (strlen(ch) > MAXSTRLEN)
42         return;
43     else
44     {
45         length = strlen(ch);
46         for (int i = 0; i < length; i++)
47             str[i] = *(ch + i);
48     }
49 }
50 // 初始条件: 串S存在。
51 // 操作结果: 返回串S的元素个数, 称为串的长度。
52 int String::size() {
53     return length;
54 }
55 Status String::concat(String& t)
56 {
57     bool uncat = false;
58     if (length == MAXSTRLEN)
59     {
60         uncat = false;
61     }
62     else if (length + t.length <= MAXSTRLEN)
63     {

```

```

64         for (int i = 0; i < t.length; i++)
65         {
66             str[length + i] = t.str[i];
67         }
68         this->length = t.length + this->length
69         ;
70         uncat = true;
71     }
72     else
73     {
74         for (int i = 0; i < MAXSTRLEN -
75             length; i++)
76         {
77             str[length + i] = t.str[i];
78         }
79         length = MAXSTRLEN;
80         uncat = false;
81     }
82     return uncat;
83 }
84 void String::print()
85 {
86     for (int i = 0; i < length; i++)
87     {
88         cout << str[i];
89     }
90     cout << endl;
91 }
92 /* 由串复制得串ST */
93 Status String::strcpy(String& t, String& s)
94 {
95     for (int i = 0; i < s.length; i++)

```

```

95         t.str[i] = s.str[i];
96     t.length = s.length;
97     return OK;
98 }
99
100 /* 若为空串S则返回,TRUE否则返回,FALSE */
101 Status String::empty()
102 {
103     if (length == 0)
104         return TRUE;
105     else
106         return FALSE;
107 }
108
109 /* 初始条件 : 串和存在ST */
110 /* 操作结果 : 若S>T则返回值若,>0;S=T则返回值若,=0;S<T则返回值,<0 */
111 int String::strcmpy(String t)
112 {
113     if (t.length != length)
114     {
115         return length - t.length;
116     }
117     for (int i = 0; i < length && i < t.length; ++i)
118     {
119         if (str[i] != t.str[i])
120         {
121
122             return str[i] - t.str[i];
123         }
124     }
125     //return S[0] - T[0];
126 }

```

```

127
128
129 /* 初始条件串:存在。操作结果S将:清为空串S */
130 Status String::clear()
131 {
132     length = 0; /* 令串长为零 */
133     return OK;
134 }
135 /* 用返回串的第个字符起长度为的子串。SubSposlen */
136 Status String::substring(String& sub, int pos, int len)
137 {
138     if (pos<0 || pos>=length || len<0 || len>
        length - pos + 1)
139         return ERROR;
140     for (int i = 0; i < len; i++)
141         sub.str[i] = str[pos + i];
142     sub.length = len;
143     return OK;
144 }
145
146 /* 返回子串在主串中第个字符之后的位置。若不存在TSpos则函数返回值为。0 */
147 /* 其中,非空T≤,1≤posStrLength(S)。 */
148 int String::index(String t, int pos)
149 {
150     int i = pos; /* 用于主串中当前位置下标值,若不为,
        则从位置开始匹配iSpos1pos */
151     int j = 0; /* 用于
        子串中当前位置下标值jT */
152     while (i <length && j <t.length) /* 若小于的长度
        并且小于的长度时,循环继续iSjT */
153     {
154         if (str[i] == t.str[j]) /* 两字
            符相等则继续 */
155         {

```

```

156         ++i;
157         ++j;
158     }
159     else /* 指针
        后退重新开始匹配 */
160     {
161         i = i - j + 1; /* 退回
            到上次匹配首位的下一位i */
162         j = 0; /* 退回
            到子串的首位jT */
163     }
164 }
165 if (j >= t.length)
166     return i - t.length;
167 else
168     return -1;
169 }
170 #include <cstring>
171
172 //int main()
173 int main()
174 {
175     //char *c;
176     //strcpy(c, "XYZNMP");
177     String s("武汉");
178     s.print();
179     String t("加油");
180     s.concat(t);
181     s.print();
182
183     String str;
184     String::strcpy(str, s);
185     str.print();
186     str.clear();
187     str.print();

```

```

188         if (str.empty())
189         {
190             cout << "是空的str" << endl;
191         }
192         String sub;
193         s.substring(sub, 0, 4);
194         sub.print();
195         cout << "在中的位置是:
            ts" << s.index(t, 0) << endl;
196
197         String s1("Hello World");
198         String s2("Hello2");
199         cout << s1.index(s2, 0) << endl;
200         system("pause");
201         return 0;
202     }

```

下面是字符串堆分配存储的实现:

```

1  #pragma once
2  #include "predefine.h"
3  class HString
4  {
5  public:
6      HString();
7      HString(const char *p);
8      ~HString();
9
10 public:
11     Status assign(const char *p);
12     Status concat(HString s1, HString s2); //字符串
        联接
13     Status print(); //打印字符串
14     Status size();

```



```

15 |         Status del(int pos, int len); //删除字符串中某个
    |         位置固定长度的子串
16 |         Status insert(int pos, HString& t);
17 |         Status index(HString& t, int pos); //在字符串S中
    |         索引位置pos之后的子串t
18 |         Status substring(HString &sub, int pos, int
    |         len);
19 |         Status kmp(HString t);
20 |         Status empty(); //字符串判空
21 |         Status replace(HString s1, HString s2); //将字
    |         符串T中等于S1的子串替换成为S2
22 |         Status substring(HString *sub, int pos, int len
    |         );
23 |     protected:
24 |         void getNext(HString t, int next[]);
25 |     private:
26 |         char *ch;
27 |         int length;
28 | };
29 | #include "HString.h"
30 |
31 |
32 |
33 | HString::HString()
34 | {
35 |     ch = NULL;
36 |     length = 0;
37 | }
38 | HString::HString(const char* p)
39 | {
40 |     int i, len = strlen(p);
41 |     ch = (char *)malloc(len * sizeof(char));
42 |     if (!ch)
43 |     {
44 |         exit(OVERFLOW);

```

```

45     }
46     for (i = 0; i < len; i++)
47     {
48         ch[i] = p[i];
49     }
50     length = len;
51 }
52
53 HString::~~HString()
54 {
55     ch = NULL;
56     free(ch);
57 }
58
59 Status HString::assign(const char *p)
60 {
61     //1.判断T是否已有内容,有则释放
62     //2.判断赋值的内容是否为空,为空则不赋值
63     //3.根据长度向内存申请空间,遍历赋值给T,长度等于字符串长
        度
64     //p.s.在这里赋值不赋
        0,在打印时通过长度来判断字符串结尾
65     int len = strlen(p);
66     if (ch)
67     {
68         free(ch);
69     }
70     if (!len)
71     {
72         ch = NULL;
73         length = 0;
74         return ERROR;
75     }
76     else

```

```

77     {
78         ch = (char *)malloc(len * sizeof(char)
79                                );
80
81         if (!ch)
82         {
83             exit(OVERFLOW);
84         }
85         for (int i = 0; i < len; i++)
86         {
87             ch[i] = p[i];
88         }
89         length = len;
90     }
91     return OK;
92 }
93
94 Status HString::concat(HString s1, HString s2)
95 {
96     //1.申请长度为S1和S2之和的字符串空间
97     //2.先将S1的元素逐个赋值到T中
98     //3.再将S2的元素逐个赋值到T中
99     ch = (char *)malloc((s1.length + s2.length) *
100                          sizeof(char));
101
102     if (!ch)
103     {
104         exit(OVERFLOW);
105     }
106     for (int i = 0; i < s1.length; ++i)
107     {
108         ch[i] = s1.ch[i];
109     }
110     for (int i = 0; i < s2.length; ++i)

```

```

108         {
109             ch[i + s1.length] = s2.ch[i];
110         }
111         length = s1.length + s2.length;
112         return OK;
113     }
114     Status HString::print() //打印字符串
115     {
116         for (int i = 0; i < length; ++i)
117         {
118             cout << ch[i];
119         }
120         cout << endl;
121         return OK;
122     }
123     Status HString::size()
124     {
125         return length;
126     }
127     Status HString::del(int pos, int len)
128     {
129         //pos是字符串中的位置,删除包括pos的len长度
130         if (pos >= length)
131         {
132             return ERROR;
133         }
134         if (pos + len > length)
135         {
136             len = length - pos + 1;
137         }
138         for (int i = pos; i < length - len; i++)
139         {
140             ch[i] = ch[i + len];

```

```

141     }
142     length -= len;
143     ch = (char *)realloc(ch, length * sizeof(char)
144         );
145     if (!ch)
146     {
147         exit(OVERFLOW);
148     }
149     return OK;
150 }
151 Status HString::insert(int pos, HString& t)
152 {
153     //pos是字符串中的位置,插入时原来的元素(包括pos位)后移
154     int i, len;
155     //-pos;
156     len = t.size();
157     ch = (char *)realloc(ch, (length + len) *
158         sizeof(char));
159     if (!ch)
160     {
161         exit(OVERFLOW);
162     }
163     if (pos > length)
164         pos = length;
165     for (i = length - 1; i > pos - 1; --i)
166         ch[i + len] = ch[i];
167     for (i = 0; i < len; ++i)
168         ch[i + pos] = t.ch[i];
169     length += len;
170     return OK;
171 }

```

```

172  /* 返回子串在主串中第个字符之后的位置。若不存在TSpos则函数返回值为。 ,0 */
173  /* 其中,非空T≤,1≤posStrLength(S)。 */
174  int HString::index(HString& t, int pos)
175  {
176      int i = pos; /* 用于主串中当前位置下标值,若不为,
177                  则从位置开始匹配iSpos1pos */
178      int j = 0; /* 用于子串中当前位置下标值jT */
179      while (i < length && j < t.length) /* 若小于的长度并且小于的长度时,循环继续iSjT */
180      {
181          if (ch[i] == t.ch[j]) /* 两字符相等则继续 */
182          {
183              ++i;
184              ++j;
185          }
186          else /* 指针后退重新开始匹配 */
187          {
188              i = i - j + 1; /* 退回到上次匹配首位的下一位i */
189              j = 0; /* 退回到子串的首位jT */
190          }
191      }
192      if (j >= t.length)
193          return i - t.length;
194      else
195          return -1;
196  }
197  Status HString::substring(HString& sub, int pos, int len)
198  {
199      if (pos < 0 || pos >= length || len < 0 || len >

```

```

length - pos + 1)
199         return ERROR;
200     for (int i = 0; i < len; i++)
201         sub.ch[i] = ch[pos + i];
202     sub.length = len;
203     return OK;
204 }
205 //求模式t的next串t函数值并保存在next数组中
206 void HString::getNext(HString t, int next[])
207 {
208     int k = 0, j = -1;
209     next[0] = -1;
210     while (k < t.length-1)
211     {
212         if ((j == -1) || (t.ch[k] == t.ch[j]))
213         {
214             k++;
215             j++;
216             next[k] = j;
217         }
218         else
219         {
220             j = next[j];
221         }
222     } //end while
223 } //end next
224 Status HString::kmp(HString t)
225 {
226     //T-主串,P-模式串
227     int * next = new int[t.length];
228     getNext(t, next);
229     int i = 0;
230     int j = 0;
231     while (j < t.length && i < length ) {

```

```

231         if (j == -1 || t.ch[j] == ch[i]) {
232             i++;
233             j++;
234         }
235         else {
236             j = next[j];
237         }
238
239     }
240     delete [] next;
241     if (j >= t.length)
242     {
243         return i - t.length;
244     }
245     else
246     {
247         return -1;
248     }
249 }
250 Status HString::empty()
251 {
252     if (length == 0)
253     {
254         return true;
255     }
256     return false;
257 }
258 Status HString::replace(HString s1, HString s2)
259 {
260     //循环索引子串S1在字符串T中的位置(每次的位置从上一次位置后开始查找)
261     //从查找到的位置-1开始替换
262     //p.s.初始状态下S1为非空串
263     int pos = 0;

```



```

264         if (s1.empty())
265             return ERROR;
266         //当pos存在时循环,当全部索引完毕后pos为0
267         //将索引到的该位置对应的子串删除后再插入新的子串
268         do
269         {
270             pos = index(s1, pos);
271             if (pos >= 0)
272             {
273                 del(pos, s1.size());
274                 insert(pos, s2);
275             }
276         } while (pos >= 0);
277
278         return OK;
279     }
280     Status HString::substring(HString *sub, int pos, int
        len)
281     {
282         int i;
283         if (pos < 1 || len > length || len < 0 || len
            > length - pos + 1)
284         {
285             exit(OVERFLOW);
286         }
287         if (sub->ch)
288         {
289             free(sub->ch);
290         }
291         //如果查询的长度为0,则子串置空
292         if (len == 0)
293         {
294             sub->ch = NULL;

```

```

295         sub->length = 0;
296     }
297     else
298     {
299         sub->ch = (char *)malloc(len * sizeof(
300             char));
301         for (i = 0; i < len; ++i)
302         {
303             sub->ch[i] = ch[pos + i - 1];
304         }
305         sub->length = len;
306     }
307     return OK;
308 }
309 int main()
310 {
311     HString s1("Hello");
312     HString s2("World");
313     HString t;
314     t.concat(s1, s2);
315     t.print();
316     HString s("ABABABABABD");
317     HString sub("ABABD");
318     cout<<s.kmp(sub)<<endl;
319     const char *p = "Hello,String!";
320     const char *q = "Bye,Bye!";
321     HString str1(p);
322     HString str2(q);
323
324     str1.insert(13, str2);
325     str1.print();
326     HString oldStr("Bye");

```

```
327         HString newStr("Welcome");
328         str1.replace(oldStr , newStr);
329         str1.print();
330         cout << str1.kmp(oldStr) << endl;
331         cout << str1.kmp(newStr) << endl;
332         system("pause");
333         return 0;
334     }
```