第 4 章　串

4.1　串类型的定义
4.2　串的表示和实现
4.3　串的模式匹配算法

4.1 串类型的定义

串是由多个或零个字符组成的有限序列 ，记作

$$S = 'c_1c_2c_3…c_n' \quad (n>=0)$$

其中：

- S 是串名字
- $'c_1c_2c_3…c_n'$ 是串值
- c_i 是串中字符
- n 是串的长度，表示串中字符的数目

空串：零个字符的串称为空串记作“ **Ø**”
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
字符在串中的位置：字符在序列中的序号
子串在串中的位置：子串的第一个字符在主串中的位置

- 串相等： p70
- 空格串：由一个或多个空格组成的串
- 串的表示：用一对单引号括起来
- 串的操作：以“串的整体”为操作对象
- 串的抽象数据类型
- 基本操作集

4.2 串的表示和实现

1. 定长顺序存储表示
2. 堆分配存储表示
3. 串的块链存储表示
4. 串的基本操作

1. 定长顺序存储表示

- 静态分配
  - 每个串预先分配一个固定长度的存储区域。
  - 实际串长可在所分配的固定长度区域内变动
  - 对串长有两种表示方法
    - 以下标为 0 的数组分量存放串的实际长度—— PASCAL
    - 在串值后加入“ \0” 表示结束，此时串长为隐含值

- 用定长数组描述：

```
#define MAXSTRLEN 255 // 最大串长
typedef unsigned char SString[MAXSTRLEN + 1]
  //0 号单元存放串的长度
```

2. 堆分配存储表示

- 以一组地址连续的存储单元存放串值字符序列；
- 存储空间动态分配，用 malloc() 和 free() 来管理
- 既有顺序存储结构的特点，又可根据需要申请串的长度空间
- P75 示例

3. 串的块链存储表示

- 串的链式存储方式
- 结点大小：一个或多个字符
  - P78 图 4.2 (a) (b)
  - 存储密度 = 串值所占的存储位 / 实际分配的存储位

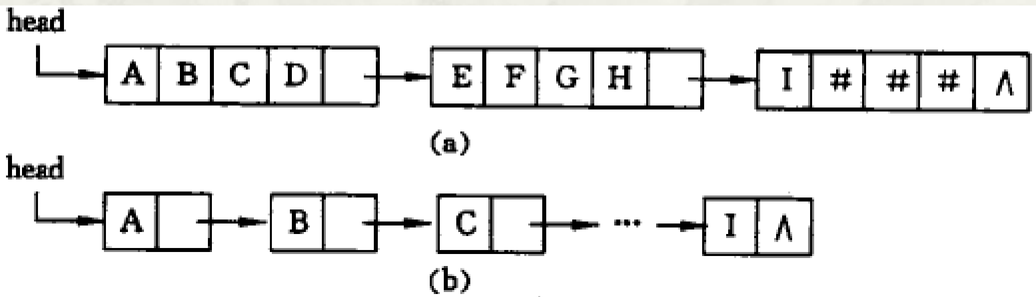

图 4.2 串值的链表存储方式

(a) 结点大小为 4 的链表；(b) 结点大小为 1 的链表

4. 串的基本操作

- 串插入　Status StrInsert(HString &S,int pos,HString T)
- 串赋值　Status StrAssign(HString &S,char *chars)
- 求串长　int StrLength(HString S)
- 串比较　int StrCompare(HString S,HString T)
- 串联接　Status Concat(HString &S,HString S1,HString S2)
- 求子串　Status SubString(HString &Sub,HString S,int pos,int len)
- 串清空　Status ClearString(HString &S)
- 串定位
- 删除
- 置换

Status StrInsert(HString &S,int pos,HString T)

```
// 在串 S 的第 pos 个位置前插入串 T
{  int i;
   if (pos<1||pos>S.length+1) return ERROR;
   if (T.length){
        if (!(S.ch=(char*) realloc(S.ch,
   (S.length+T.length)*sizeof(char))))
           exit(OVERFLOW);
        for (i=S.length-1;i>=pos-1;--i)
         { S.ch[i+T.length]=S.ch[i];}
        for (i=0; i<=T.length-1;i++)
               S.ch[pos-1+i]=T.ch[i];
        S.length+=T.length;
   } return OK;
}
```

Status StrAssign(HString &S,char *chars)
生成一个值等于 chars 的串 S

```
{  int i,j;  char *c;
   for (i=0,c=chars;*c;++i,++c);
   if (!i) {S.ch=NULL;  S.length=0;}
   else {
         if (!(S.ch=(char *)malloc(i * sizeof(char))))
                  exit(OVERFLOW);
         for (j=0;j<=i-1;j++){
                  S.ch[j]=chars[j];}
         S.length=i;
        }
   return OK;
}
```

int StrLength(HString S)
求串的长度

```
{
  return S.length;
}
```

int StrCompare(HString S,HString T) 比较两个串，若相等返回 0

```
{
   int i;
   for (i=0;i<S.length && i<T.length; ++i)
        if (S.ch[i] != T.ch[i]) return S.ch[i]-T.ch[i];
        return S.length-T.length;
}
```

Status Concat(HString &S,HString S1,HString S2)

用 S 返回由 S1 和 S2 联接而成的新串

```
{ int j;
  if (!(S.ch = (char*)malloc((S1.length+S2.length)*sizeof(char))))
      exit(OVERFLOW);
  for (j=0;j<=S1.length-1;j++)
    { S.ch[j]=S1.ch[j]; }
  S.length=S1.length+S2.length;
  for (j=0;j<=S2.length-1;j++)
    { S.ch[S1.length+j]=S2.ch[j]; }
  return OK;
}
```

Status SubString(HString &Sub,HString S,int pos,int len)

用 Sub 返回串 S 的第 pos 个字符开始长度为 len 的子串

```
{
    if (pos<1 || pos>S.length || len<0 ||len>S.length-pos+1)
        return ERROR;
    if (!len) { Sub.ch=NULL; Sub.length=0;}
    else {
        Sub.ch=(char *)malloc(len*sizeof(char));
        for (int j=0;j<=len-1;j++){
                Sub.ch[j]=S.ch[pos-1+j];}
        Sub.length=len;
    }
    return OK;
}
```

Status ClearString(HString &S)
将 S 清为空串

```
{
  if (S.ch) { free(S.ch); S.ch=NULL;}
  S.length=0;
  return OK;
}
```

4.3 串的模式匹配算法

- 定义　在串中寻找子串（第一个字符）在串中的位置
- 词汇　在模式匹配中，子串称为模式，串称为目标。
- 示例　目标 S : “Beijing”

　　模式 P : “jin”

匹配结果 ＝ 4

1. 穷举模式匹配

- 设 S=s1,s2,…,sn(主串)P=p1,p2,…,pm(模式串)

i 为指向 S 中字符的指针， j 为指向 P 中字符的指针

匹配失败： si≠pj 时，

$(s_{i-j+1} \ \ldots \ s_{i-1})=(p_1 \ \ldots \ p_{j-1})$

回溯： i=i-j+2；j=1

重复回溯太多， O(m*n)

第 1 趟　　S　a b b a b a　　穷举的模式

P　a b a　　匹配过程

第 2 趟　　S　a b b a b a

P　　a b a

第 3 趟　　S　a b b a b a

P　　　a b a

第 4 趟　　S　a b b a b a

P　　　　a b a

√

求子串位置的定位函数

```
int Index(SString S, SString T,int pos) {
// 穷举的模式匹配
   int i=pos;      int j=1;
   while (i<=S[0] && j<=T[0]) {// 当两串未检测完，S[0]、 S[0] 为串长
       if (S[i]==T[j]) {++i; ++j;}
       else {i=i-j+2;  j=1;}
   }
   if (j>T[0]) return i-T[0]; // 匹配成功
   else return 0;
}
```

2.KMP 快速模式匹配

- D.E.Knuth, J.H.Morris, V.R.Pratt 同时发现
- 无回溯的模式匹配

$$
\begin{array}{lccccccccccc}
\mathbf{S} & s_1 & \cdots & s_{i-j-1} & s_{i-j} & s_{i-j+1} & s_{i-j+2} & \cdots & s_{i-1} & s_i & s_{i+1} & \cdots\ s_n \\
 & & & & \| & \| & \| & \| & \times & & & \\
\mathbf{P} & & & & p_1 & p_2 & \cdots & p_{j-1} & p_j & p_{j+1} & \cdots & p_m
\end{array}
$$

则有 $s_{i-j+1}\ s_{i-j+2} \cdots s_{i-1} = p_1\ p_2 \cdots p_{j-1}$ (1)

为使模式 **P** 与目标 **S** 匹配，必须满足

$p_1\ p_2 \cdots p_{j-1}\ p_j \cdots p_m = s_{i-j+1}\ s_{i-j+2} \cdots s_{i-1}\ s_i \cdots s_{i-j+m}$

如果 $p_1 \cdots p_{j-2} \neq p_2\ p_3 \cdots p_{j-1}$ (2)

由 **(1)(2)** 则立刻可以断定

$$p_1 \cdots p_{j-2} \neq s_{i-j+2}\ s_{i-j+3} \cdots s_{i-1}$$

下一趟必不匹配

同样，若 $p_1\ p_2 \ldots p_{j-3} \neq p_3\ p_4 \ldots p_{j-1}$

则再下一趟也不匹配，因为有

$$p_1 \ldots p_{j-3} \neq s_{i-j+3} \ldots s_{i-1}$$

直到对于某一个“ k” 值，使得

$$p_1 \ldots p_k \neq p_{j-k}\ p_{i-k+1} \ldots p_{j-1}$$

且 $p_1 \ldots p_{k-1} = p_{j-k+1}\ p_{j-k+2} \ldots p_{j-1}$

则 $p_1 \ldots p_{k-1} = s_{i-k+1}\ s_{i-k+2} \ldots s_{i-1}$

$$\Vert \quad \Vert \quad \Vert$$

$$p_{j-k+1}\ p_{j-k+3} \ldots p_{j-1}$$

模式右滑 **j-k** 位

next 数组值

- 假设当模式中第 j 个字符与主串中相应字符“失配”时，可以拿第 k 个字符来继续比较，则令 next[j]=k

next 函数定义：

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

手工求 **next** 数组的方法

- 序号 j 1 2 3 4 5 6 7 8
- 模式 P a b a a b c a c
- k 1 1 2 2 3 1 2
- Pk==Pj ≠ = ≠ = ≠ = ≠
- next[j] 0 1 1 2 2 3 1 2
- Nextval[j] 0 1 0 2 1 3 0 2

运用 **KMP** 算法的匹配过程

第 **1** 趟 目标 **a c a b a a b a a b c a c a a b c**

模式 **a b a a b c a c**

× **next(2) = 1**

第 **2** 趟 目标 **a c a b a a b a a b c a c a a b c**

模式 **a b a a b c a c** **next(1)=0**

第 **3** 趟 目标 **a c a b a a b a a b c a c a a b c**

模式 **a b a a b c a c**

×**next(6) = 3**

第 **4** 趟 目标 **a c a b a a b a a b c a c a a b c**

模式 **(a b) a a b c a c**

√

KMP 算法

```
int Index_KMP(SString S, SString T, int
   *next){ int i,j;
   i=1;  j=1;
   while (i<=S[0] && j<=T[0]){
        if (j==0 || S[i]==T[j]){++i;++j;}
        else j=next[j];
   }
   if (j>T[0]) return i-T[0];
   else return 0;
}
```

求 **next** 数组的步骤

(1)next[1]=0

i=1; j=0;

(2) 设 next[i]=j

若 j=0, next[i+1]=1

若 Pi=Pj, next[i+1]=j+1=next[i]+1

若 Pi ≠ Pj, next[i+1]=next[j]+1

参看教材 p82~83 递推过程

求 next 数组的函数

```
void get_next(SString S, int *next){
   int i,j;
   i=1;      next[1]=0;      j=0;
   while (i<S[0]) {
       if (j==0 || S[i]==S[j]) {++i; ++j;
   next[i]=j;}
       else j=next[j];
   }
}
```

改进的求 **next** 数组方法

设 next[i]=j

若 P[i]=P[j]，则 nextval[i]=nextval[j]

改进的求 **next** 数组的函数

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

- 穷举的模式匹配算法时间代价

最坏情况比较 **n-m+1** 趟，每趟比较 **m** 次

，

总比较次数达 **(n-m+1)*m**

- 原因在于每趟重新比较时，目标串的检测指针要回退。改进的模式匹配算法可使目标串的检测指针每趟不回退。

- 改进的模式匹配 **(KMP)** 算法的时间代价
  - 若每趟第一个不匹配，比较 n-m+1 趟，总比较次数最坏达 (n-m)+m = n
  - 若每趟第 m 个不匹配，总比较次数最坏亦达到 n
  - 求 next 函数的比较次数为 m，所以总的时间复杂度是 O(n+m)