串

▼ 抽象数据类型和概念

```
ADT List {
    数据对象：D={ai | 1≤i≤n, n>0, ai ∈ CharacterSet
    数据关系：R={<ai, ai+1> | ai, ai+1 ∈ D, i=1,2,...n-1}
    基本运算：
    StrAssign( &T, chars);
    DestroyString( &S );
    StrCopy (&T, S);
    StrEmpty (S);
    StrCompare (S,T);
    StrLength (S);
    ClearString ( &S);
    Concat (&T, S1, S2);
    Substring( &Sub, S, pos, len);
    Index (S, T, pos);
    Replace (&S, T, V).
    StrInsert (&S, pos, T);
    StrDelete (&S, pos, len);
}
```

串的长度：串中字符个数

空串：串的长度为0

子串：串中任意个连续字符组成的子序列

最小操作子集：StrAssign，Strcopy，StrCompare，StrLength，Concat，SubString

▼ 存储结构

定长顺序存储表示（静态）

```
typedef struct{
    char ch[MAXLEN+1];
    int length;
}SString;
//可以舍弃ch[0],让位序和数组下标相同
//也可以用ch[0]储存字符串长度
```

堆分配存储表示（动态）

```
typedef struct{
    char *ch;
    int length;
}HString;

bool StrInsert(HString &S,int pos,HString T){
    int n=S.length,m=T.length;
    if(pos<1||pos>n+1) return false;
    if(T.length){
        S.ch=(char*)realloc(S.ch,(n+m)*sizeof(char));
        for(int i=n+m;i>=pos+m;i--){
            S.ch[i]=S.ch[i-m];
        }
        for(int i=1;i<=m;i++){
            S.ch[i+pos-1]=T.ch[i];
        }
        S.length =n+m;
    }
    return true;
}
```

块链存储表示（动态）

```
typedef struct Chunk{
    char ch[SIZE];   //每个结点存多个字符,提高存储密度
    struct Chunk*next;
}Chunk;

typedef struct{
    Chunk *head,*tail;
    int len;
}LString;
```

▼ 基本操作

```
void Init(SString &S){
    S.length=0;
    S.ch[1]='\0';
}

int StrLength(SString S){
    return S.length;
}

bool StrAssign(SString &S, char *T) {
    int len = 0;
    while (T[len]!='\0') {
        len++;
    }
    if (len>MAXLEN) {
        return false;
    }
    for (int i=1;i<=len;i++) {
        S.ch[i]=T[i-1];
    }
    S.length=len;
```

```
    S.ch[len+1]='\0';
    return true;
}

bool StrCopy(SString &T, SString S) {
    if (S.length>MAXLEN) {
        return false;
    }
    for (int i=1;i<=S.length;i++) {
        T.ch[i]=S.ch[i];
    }
    T.length=S.length;
    T.ch[T.length+1]='\0';
    return true;
}

bool StrEmpty(SString S) {
    return (S.length==0);
}

bool ClearString(SString &S) {
    S.length=0;
    return true;
}

bool SubString(SString &Sub,SString S,int pos,int len){
    //子串范围越界
    if(pos+len-1>StrLength(S)||pos<1||len<0) return false;
    for(int i=pos;i<=pos+len-1;i++){
        Sub.ch[i-pos+1]=S.ch[i];
    }
    Sub.ch[len+1]='\0';
    Sub.length=len;
    return true;
}
```

```
int StrCompare(SString S,SString T){
    for(int i=1;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;
}

int Index(SString S,SString T,int pos){
    if(pos<=0) return 0;
    int i=pos,n=StrLength(S),m=StrLength(T);
    SString sub;
    while(i<=n-m+1){
        SubString(sub,S,i,m);
        if(StrCompare(sub,T)!=0) ++i;
        else return i;
    }
    return 0;
}

bool Concat(SString &T,SString S1,SString S2){
    int n=StrLength(S1),m=StrLength(S2);
    if(n+m<=MAXLEN){
        T.length=n+m;
        for(int i=1;i<=n;i++){
            T.ch[i]=S1.ch[i];
        }
        for(int i=n+1;i<=n+m;i++){
            T.ch[i]=S2.ch[i-n];
        }
        T.ch[n+m+1]='\0';
        T.length=n+m;
        return true;
    }
    return false;
}
```

```
//将S中的T子串替换为V
void Replace(SString &S,SString T,SString V){
   int n=StrLength(S),m=StrLength(T),v=StrLength(V),pos=1;
   SString sub,news;
   Init(news);
   int i=1;
   while(pos<=n-m+1&&i){
      i=Index(S,T,pos);  //从pos起查找T
      if(i){
         SubString(sub,S,pos,i-pos); //取无需替换的子串
         Concat(news,news,sub);
         Concat(news,news,V);      //无需替换的接换成的V
         pos=i+m;   //继续查询的起始位置
         S.length+=v-m;
      }
   }
   //没有找到T子串退出循环
   SubString(sub,S,pos,n-pos+1);    //剩余子串
   Concat(S,news,sub);            //连接news和剩余子串并赋给S
}

bool StrInsert(SString &S, int pos, SString T) {
   int n=StrLength(S),m=StrLength(T);
   if (pos<1||pos>n+1||n+m>MAXLEN) {
      return false;
   }
   for(int i=n+m;i>=pos+m;i--){
      S.ch[i]=S.ch[i-m];
   }
   for(int i=1;i<=m;i++){
      S.ch[i+pos-1]=T.ch[i];
   }
   S.length =n+m;
   S.ch[n+m+1]='\0';
   return true;
}
```

```
bool StrDelete(SString &S,int pos,int len){
    int n=StrLength(S);
    if(pos<1||len<0||pos>n||pos+len-1>n)
        return false;
    for(int i=pos;i<=n-len;i++){
        S.ch[i]=S.ch[i+len];
    }
    S.length=n-len;
    S.ch[n-len+1]='\0';
    return true;
}
```

▼ 模式匹配

▼ 朴素模式匹配算法

- 最坏 $O(m \times n)$ $m = length(S)$ $n = length(T)$

```
int Index(SString S,SString T,int pos){
    int i=pos,j=1;
    while(i<=S.length && j<=T.length){
        if(S.ch[i]==T.ch[j]){
            ++i;++j;
        }
        else{
            i=i-j+2;  //先退回j进行的个数(j-1)再+1
            j=1;
        }
    }
    if(j>T.length) return i-T.length;  //匹配成功,返回位序
    else return 0;
}
```

▼ KMP算法

- 最坏 $O(m + n)$

```
void get_next(SString T,int *next){
    int j=1,k=0;
    next[1]=0;
    while(j<T.length){
        if(k==0||T.ch[j]==T.ch[k]){
            ++j;++k;
            next[j]=k;
        }
        else k=next[k];
    }
}

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

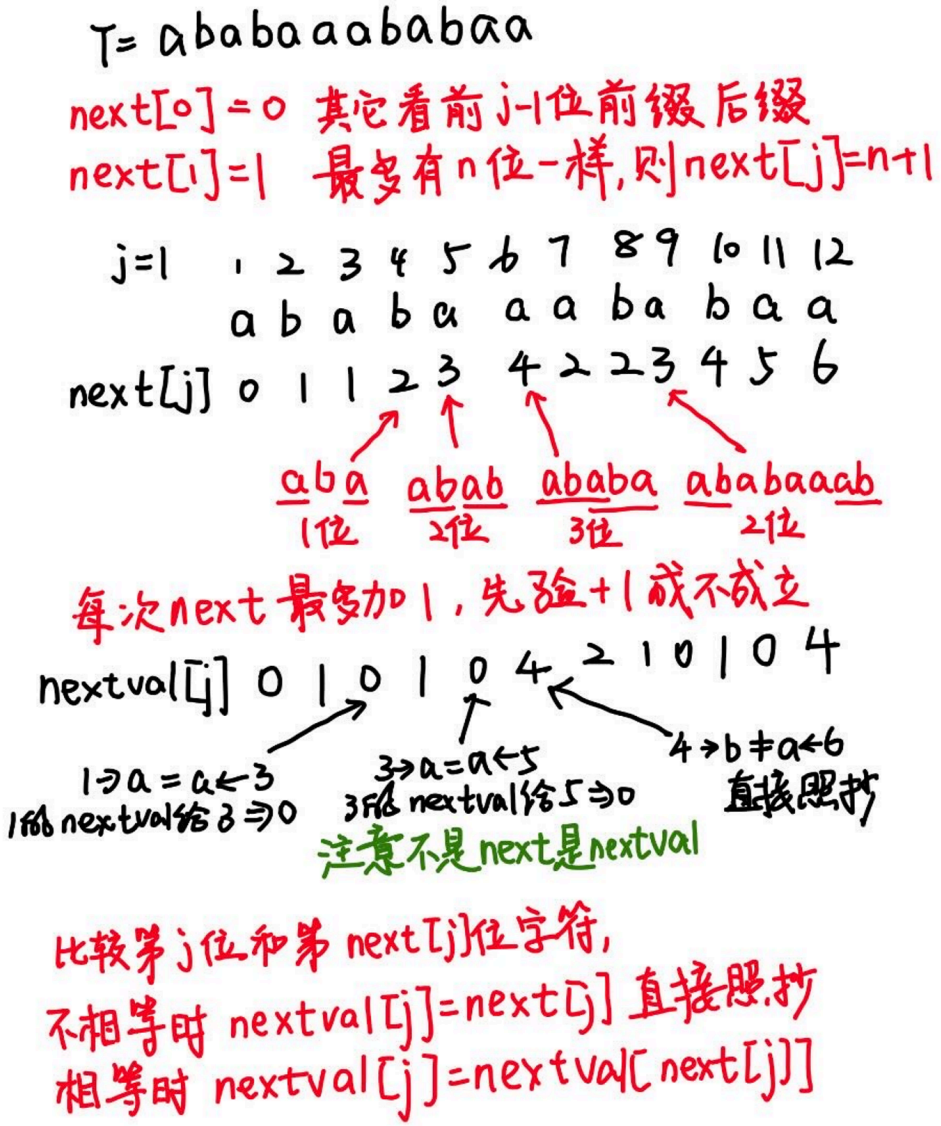
优化

```
void get_nextval(SString T,int *next){
    int j=1,k=0;
    next[1]=0;
    while(j<T.length){
        //把j看成再主串的指针
        if(k==0||T.ch[j]==T.ch[k]){
            ++j;
            ++k;
            //匹配失败意味着主串该位置肯定不是T.ch[k]的值
            //因此更新的主串指针j对应T.ch[j]==T.ch[k]则必然匹配不上
            if(T.ch[j]!=T.ch[k]) next[j]=k;
```

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

手算：

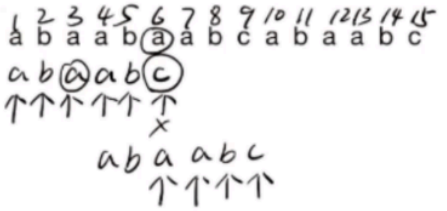
- 找模式串前 k-1 个字符（可以盖住第 j 位）的前缀后缀重合个数+1赋给 next [k]
- nextval求解方法是对比 j 与 next[j] 的字符
    1. 一样则 nextval [j] = nextval [next [j]]
    2. 不一样则 nextval [j] = next [j]

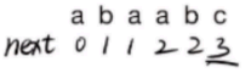


检验一下会了没

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	a	b	c	a	a	b	b	a	b	c	a	b	a	a	c	b	a	c	b	a
next[j]	0	1	1	1	2	2	3	1	2	3	4	5	3	2	2	1	1	2	1	1
nval[j]	0	1	1	0	2	1	3	0	1	1	0	5	3	2	2	1	0	2	1	0

设主串T="abaabaabcabaabc",模式串S="abaabc"，采用KMP算法进行模式匹配，到匹配成功为止，在匹配过程中进行的单个字符间的比较次数是 10





当匹配到第六位的时候，匹配失败

然后回去查找C的next值（3）

从模式串的第三位开始匹配，从主串的第六位开始匹配。

然后就匹配成功，一共比较次数为10次。