五.串

- 串，即**字符串 (String)** 是由零个或多个字符组成的有限序列。一般记为S='a1a2.....·an'(n>=0)
- 其中，S是串名，单引号括起来的字符序列是串的值;a_i可以是字母、数字或其他字符;串中字符的个数n称为串的长度。n =0时的串称为空串

例

```
S="HelloWorld!"
T='iPhone 11 Pro Max?'
```

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。eg:T是子串'iphone'的主串
字符在主串中的位置:字符在串中的序号。
子串在主串中的位置:子串的第一个字符在主串中的位置。
空串V.S空格串 eg:M=''是空串，N=' '是由三个空格字符组成的空格串，每个空格字符占1B。
串是一种特殊的线性表，数据元素之间呈线性关系
串的数据对象限定为**字符集**（如中文字符、英文字符、数字字符、标点字符等）
串的基本操作，如增删改查等通常以**子串**为操作对象。

串的基本操作

假设有串 T = '', S = 'iPhone 11 Pro Max?', W = 'Pro'

- StrAssign(&T, chars)： 赋值操作，把串T赋值为chars。
- StrCopy(&T, S):：复制操作，把串S复制得到串T。
- StrEmpty(S)：判空操作，若S为空串，则返回TRUE，否则返回False。
- StrLength(S)：求串长，返回串S的元素个数。
- ClearString(&S)：清空操作，将S清为空串。
- DestroyString(&S)：销毁串，将串S销毁（回收存储空间）。
- Concat(&T, S1, S2)：串联接，用T返回由S1和S2联接而成的新串。Concat(&T,S,W)后，T=“iPhone 11 Pro Max?Pro”
- SubString(&Sub, S, pos, len)求子串，用Sub返回串S的第pos个字符起长度为len的子串. eg：执行SubString(&T,S,4,6)后，T=“one 11”
- Index(S, T)：定位操作，若主串S中存在与串T值相同的子串，则返回它再主串S中第一次出现的位置，否则函数值为0. eg:执行Index(S,W)后，返回值11
- StrCompare(S, T)：串的比较操作，参照英文词典排序方式；若S > T,返回值>0; S = T,返回值=0(需要两个串完全相同) ; S < T,返回值<0.（从第一个字符开始往后依次对比，先出现更大字符的串

就更大；长串的前缀与短串相同时，长串更大；只有两个串完全相同时才相等）

- 每个字符在计算机中对应一个二进制数，比较字符的大小就是比较二进制数的大小。

串的存储结构

串的顺序存储

静态数组实现(定长顺序存储)

```
#define MAXLEN 255    //预定义最大串长为255
typedef struct{
    char ch[MAXLEN];    //静态数组实现（定长顺序存储）;每个char字符占1B
    int length;         //串的实际长度
}SString;
```

动态数组实现(堆分配存储)

```
typedef struct{
    char *ch;           //按串长分配存储区，ch指向串的基地址
    int length;         //串的实际长度
}HString;
HString S;
S.ch = new char[MAXLEN *sizeof(char)];
S.length = 0;
```

方案二：字符的位序和数组下标一一对应，但字符串的长度不能超过255

方案三：求串长度时，需要遍历

串的链式存储

```
typedef struct StringNode{
    char ch; //每个结点存1个字符
    struct StringNode * next;
}StringNode, * String; //缺点：每个字符1B，每个指针4B，存储密度低
//改善
typedef struct StringNode{
    char ch[4]; //每个结点存多个字符
    struct StringNode * next;
}StringNode, * String; //每个结点没存满，可用#填充
```

串的基本操作

```
#define MAXLEN 255

typedef struct{
    char ch[MAXLEN];
    int length;
}SString;
```

1. 求子串:用Sub返回串的第pos个字符起长度为len的字串

```
bool SubString(SString &Sub, SString S, int pos, int len){
    //子串范围越界
    if (pos+len-1 > S.length)
        return false;

    for (int i=pos; i<pos+len; i++)
        Sub.cn[i-pos+1] = S.ch[i];

    Sub.length = len;
    return true;
}
```

2. 比较两个串的大小:若S>T,则返回值>0；若S=T,则返回值=0

```
int StrCompare(SString S, SString T){
    for (int i; i<S.length && i<T.length; i++){
        if(S.ch[i] != T.ch[i])
            return S.ch[i] - T.ch[i];
    }
    //扫描过的所有字符都相同，则长度长的串更大
    return S.length - T.length;
}
```

3. 定位操作：若主串S中存在与串T值相同的子串，则返回它在主串S中第一次出现的位置；否则函数返回值为0

```
int Index(SString S, SString T){
    int i=1;
    n = StrLength(S);
    m = StrLength(T); //串T的长度
    SString sub;        //用于暂存子串

    while(i<=n-m+1){ //长度为n的主串中有多少个m长度的子串：n-m+1
        SubString(Sub,S,i,m);
        if(StrCompare(Sub,T)!=0)  ++i;
        else return i;    // 返回子串在主串中的位置
```

```
    }
    return 0;            //S中不存在与T相等的子串
}
```

串的模式匹配算法

串的朴素模式匹配

串的模式匹配：在**主串**中找到与**模式串**相同的子串，并返回其所在主串中的位置。

朴素模式匹配算法(简单模式匹配算法) 思想:

- 将主串中与模式串长度相同的子串搞出来，挨个与模式串对比当子串与模式串某个对应字符不匹配时，就立即放弃当前子串，转而检索下一个子串。
- 若模式串长度为m，主串长度为n，则直到匹配成功/匹配失败最多需要$(n-m+1)*m$ 次比较
  最坏时间复杂度: $0(nm)$
- 最坏情况:每个子串的前m-1个字符都和模式串匹配，只有第m个字符不匹配。
- 比较好的情况:每个子串的第1个字符就与模式串不匹配

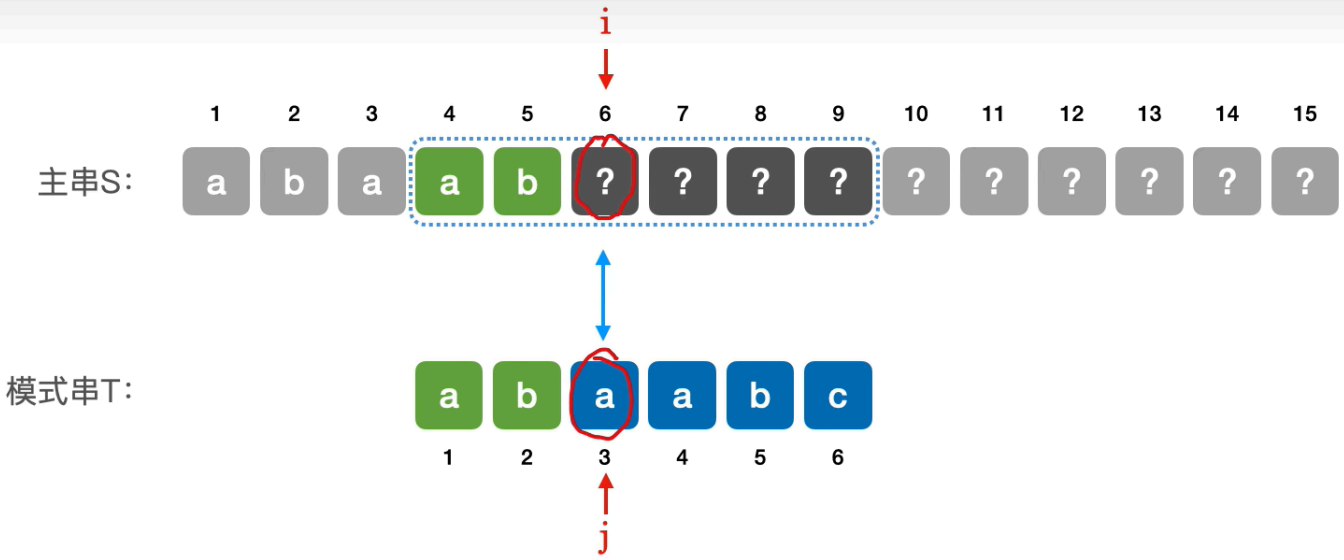
方法1：
字符串的基本操作：Index(SString S, SString T)

方法2：
直接通过数组下标实现：

```
int Index(SString S, SString T){
    int i=1;                //扫描主串S
    int j=1;                //扫描模式串T
    while(i<=S.length && j<=T.length){
        if(S.ch[i] == T.ch[j]){
            ++i;
            ++j;             //继续比较后继字符
        }
        else{                //当前子串匹配失败
            i = i-j+2;      //主串指针i指向下一个子串的第一个位置
            j=1;             //模式串指针j回到模式串第一个位置
        }
    }
    if(j>T.length)          //当前子串匹配成功
        return i-T.length;  //返回当前子串第一个字符的位置
    else
        return 0;
}
```

KMP算法

- 朴素模式匹配算法的缺点:当某些子串与模式串能部分匹配时，主串的扫描指针 i 经常回溯，导致时间开销增加。最坏时间复杂度O(mn)。
- KMP算法:当子串和模式串不匹配时，主串指针i不回溯，模式串指针j = next[j]算法平均时间复杂度：O(m+n)



可以直接从这里继续匹配

对于模式串 T = 'abaabc'，当第6个元素匹配失败时，可令主串指针 i 不变，模式串指针 j=3

对于模式串T='abaabc'
当第6个元素匹配失败时，可令主串指针i不变，模式串指针j=3
当第5个元素匹配失败时，可令主串指针i不变，模式串指针j=2
当第4个元素匹配失败时，可令主串指针i不变，模式串指针j=2
当第3个元素匹配失败时，可令主串指针i不变，模式串指针j=1
当第2个元素匹配失败时，可令主串指针i不变，模式串指针j=1
当第1个元素匹配失败时，可令主串指针i不变，模式串指针j=0，i++，j++

next数组:

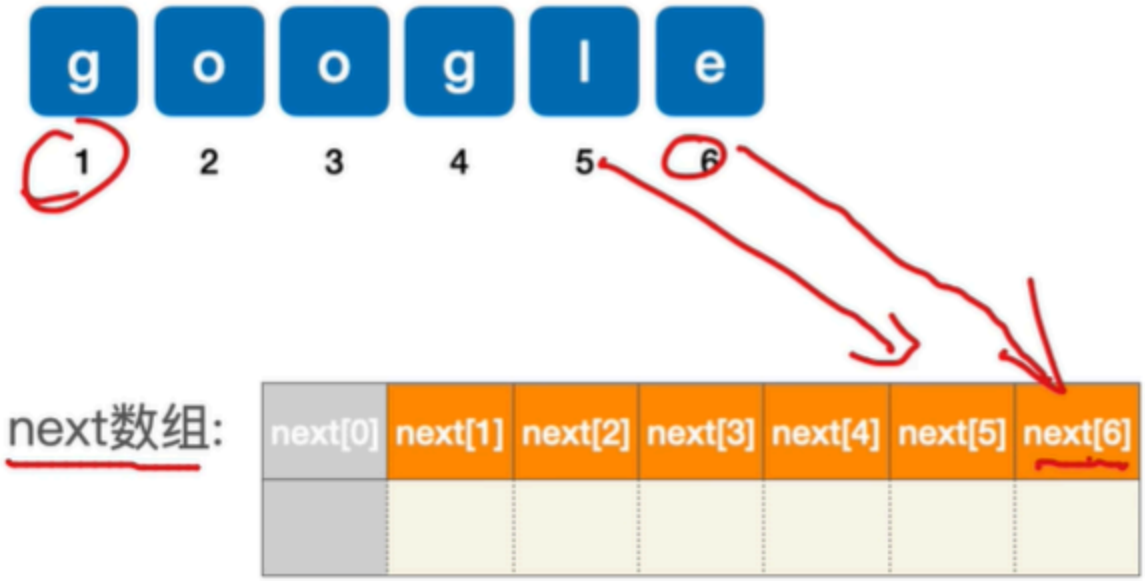
next[0]	next[1]	next[2]	next[3]	next[4]	next[5]	next[6]
	0	1	1	2	2	3

```
if (S[i] !=T[j])    j=next[j];

if (j==0)    { i++; j++ }
```

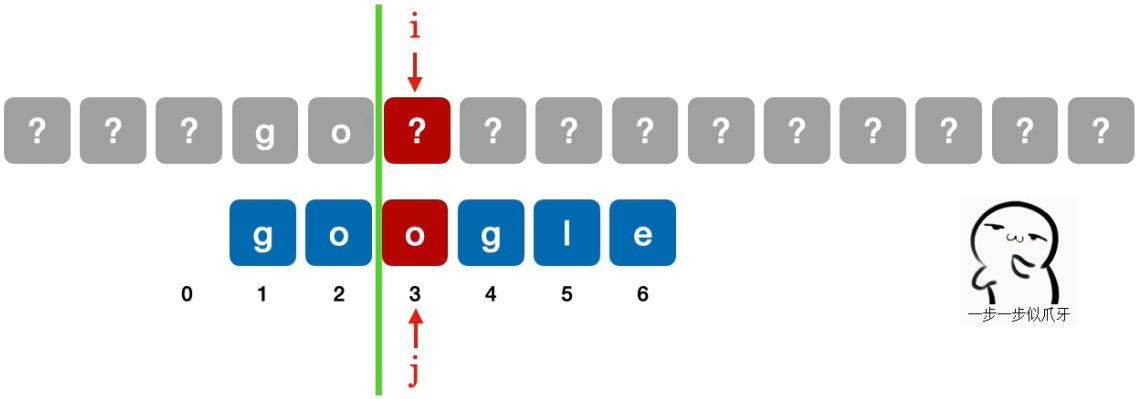
综上所述:

进行KMP算法，先进行预处理：根据模式串T，求出next数组。



任何模式串都一样，第一个字符不匹配时，只能匹配下一个子串。因此，next[1]恒为0

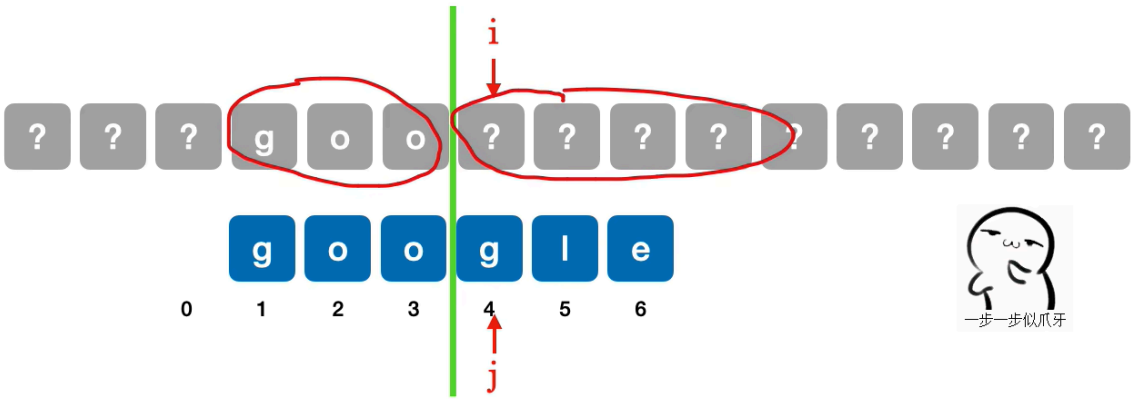
任何模式串都一样，第二个字符不匹配时，应尝试匹配模式串的第一个字符。因此，next[2]恒为1



next数组:

next[0]	next[1]	next[2]	next[3]	next[4]	next[5]	next[6]
	0	1				

在不匹配的位置前边，划一根美丽的分界线
模式串一步一步往后退，直到分界线之前
“能对上”，或模式串完全跨过分界线为止

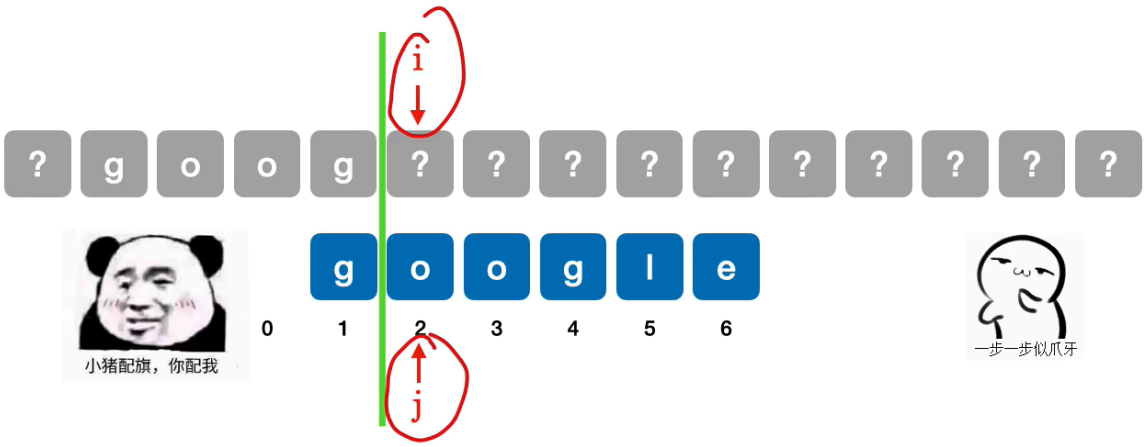


next数组:

next[0]	next[1]	next[2]	next[3]	next[4]	next[5]	next[6]
	0	1	1			

在不匹配的位置前边，划一根美丽的分界线
模式串一步一步往后退，直到分界线之前
“能对上”，或模式串完全跨过分界线为止

此时 j 指向哪儿，next数组值就是多少



next数组:

next[0]	next[1]	next[2]	next[3]	next[4]	next[5]	next[6]
	0	1	1	1		

在不匹配的位置前边，划一根美丽的分界线
模式串一步一步往后退，直到分界线之前
“能对上”，或模式串完全跨过分界线为止

此时 j 指向哪儿，next数组值就是多少

```
// 获取模式串T的next[]数组
void getNext(SString T, int next[]){
    int i=1, j=0;  //i表示当前处理的字符位置，j表示当前最长公共前后缀的长度
    next[1]=0;    //模式串的第一个字符没有前缀和后缀。
    while(i<T.length){
        if(j==0 || T.ch[i]==T.ch[j]){
            ++i; ++j;
            next[i]=j;
        }else
            j=next[j]; //跳过已经比较过的字符
    }

// KPM算法，求主串S中模式串T的位序，没有则返回0
int Index_KPM(SString S, SString T){
    int i=1, j=1;
    int next[T.length+1];
    getNext(T, next);
    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;
}
```

最坏时间复杂度：O(m+n)。其中，求next数组O(m)，模式匹配过程最坏时间复杂度o(n)

KMP算法的进一步优化

根据模式串T，求出 next 数组 → 利用next数组进行匹配（主串指针不回溯）

T = 'abaabc'

next数组:

next[0]	next[1]	next[2]	next[3]	next[4]	next[5]	next[6]
	0	1	1	2	2	3

优化

nextval数组:

nextval[0]	nextval[1]	nextval[2]	nextval[3]	nextval[4]	nextval[5]	nextval[6]
	0	1	0	2	1	3

使用nextval数组

```
int Index_KMP(SString S,SString T,int next[]){
    int i=1, j=1;
    while(i<=S.length&&j<=T.length){
        if(j==0||S.ch[i]==T.ch[j]){
            ++i;
            ++j;                    //继续比较后继字符
        }
        else
            j=next[j];              //模式串向右移动
    }
    if(j>T.length)
        return i-T.length;          //匹配成功
    else
        return 0;
}
```

先求next数组，再由next数组求nextval数组

```
void getNextval(SString T, int nextval[]){
    int i=1,j=0;
    nextval[1]=0;
    while(i<T.length){
        if(j==0 || T.ch[i]==T.ch[j]){
            ++i; ++j;
            if(T.ch[i]!=T.ch[j])
                nextval[i]=j;
            else
                nextval[i]=nextval[j];
        }else
            j=nextval[j];
    }
}
```