# KMP解析

## 模式串匹配算法：

模式串匹配算法是指在一个字符串中查找另一个字符串是否存在以及存在位置的算法。在模式串匹配算法中，被查找的字符串称为模式串(短的)，”查找于”的字符串称为主串(长的)。

显然暴力匹配时间复杂度是lena\*lenb 太慢，不多说废话了。

## KMP模式串匹配算法：

1.模式串匹配算法的产生：

主串s[]： a b c a c b c a b a c b a

模式串p[]： a c b a

在这次匹配失配后，很明显将模式串移动一个位置后是不可能匹配成功的，因为此时c字符对应的模式串中的a字符的上一个字符是b，b是不可能和c匹配成功的。所以这一次匹配根本不需要去做。

这个栗子表明，如果提前知道模式串或者主串的某两个字符之间能否匹配，就能提高效率。

Kmp大致就是这个思路。步骤是先求一个模式串next数组，它用来匹配时索引模式串移动信息，next数组长度是lenp。求完next数组，用next数组辅助匹配，具体后面说。

## 如何用next数组进行匹配

在知道如何求next数组前，应该先理解如何用next数组进行匹配的，这样有助于理解，next数组

在暴力的字符串匹配时，就是两个指针的如下循环，复杂度o(len1\*len2);

int i=0,j=0;

while(i<len1){

if(s1[i]==s2[j]){

i++;j++;

}else{

i-=j-1;//退回去

j=0;

}

if(j==len2)

return i-j;

}

s1是主串，s2是模式串

为了容易理解和KMP作对比，上述代码尽量写的和KMP的匹配相似

可见当匹配失败的时候指针i和j都要后退。这样极大的浪费了时间。

如果匹配失败时候(也就是else里面)能不回推就好了，假设s1现在匹配到了I,s2匹配到了j,

如果在能s2找到一个在j之前的位置p，使得:

s2的p位置的前缀字符串，正好与s1的i位置的前缀字符串 ，相匹配

如此做，i完全不用后退，j后在不匹配的时候需要后退。可以分析，但总复杂度变成了o(len1\*len2)

匹配的大概代码还是变成了：

int i=0,j=0;

while(i<len1){

if(s1[i]==s2[j]){

i++;j++;

}else{

//这里根据不同的next数组会有所变化，但都是根据next数组来移动j

}

if(j==len2)

return i-j;

}

设next[j]代表：在这个位置匹配失败以后，回退到哪个位置。

为什么说是大致代码，next数组有多种版本，虽然总体思路类似，但在处理上会不同，下面介绍next数组求解思路，**next数组,有多种版本，分别介绍：**

## next版本一：

### 思路：

如果在位置j匹配失败，意味着在这之前的字符串都是匹配的，主串和模式串都包含s2的j-1前缀,比如模式串ABACDABD中，看2个例子

在j=2处失败：

\*\*\*\*\*\*ABC\*\*\*\*\*\*\*\*\*\*\*\*\*

ABACABDABD

J=2处失败时，没什么好办法，j回退到0重新匹配就好。

\*\*\*\*\*\*ABC\*\*\*\*\*\*\*\*\*\*\*\*\*

ABABABDABD

-------------------------------------------------------------------------------------------------

j=3处失败

\*\*\*\*\*\*ABA**E**\*\*\*\*\*\*\*\*\*\*\*\*

ABA**B**ABDABD

J=3处失败时，ABA的首尾都有字符A，j不必回退到0，回退到1可继续匹配

\*\*\*\*\*\*ABA**E**\*\*\*\*\*\*\*\*\*\*\*\*

A**B**ABABDABD

这样就比回退到0少匹配了一次，

发现前缀ABA是可”首位相接”的,换句话说它的的” 非本身的前缀和后缀中有公共字符串”

所谓非本身的前缀和后缀就是说，找某个字符串的前缀或者后缀时，都不算和自己相同的。

ABA的非本身的前缀有：A,AB (ABA不算)

ABA的非本身的后缀有：A,BA (ABA不算)

它们都有字符串A,我们能确定

同时还要注意一件事，对于ABABAB这种：

ABABAB的非本身的前缀有：A,AB,ABA, ABAB, ABABA (ABABAB不算)

ABABAB的非本身的后缀有：B,AB, BAB, ABAB, BABAB (ABABAB不算)

它们都有字符串AB, ABAB。有多个公共字符串，就取最长的那个。意味着假如匹配时出现这种情况：

s1 \*\*\*\*\*\*ABABAB**E**\*\*\*\*\*\*\*\*\*\*\*\*

s2 ABABABDABD

应该尽可能让s2少回退，所以回退到4可继续匹配 (这比回退到2或者0，都要少匹配几次)

s1 \*\*\*\*\*\*ABABAB**E**\*\*\*\*\*\*\*\*\*\*\*\*

s2 ABABABDABD

这种情况下：

next[j]是：字符串的j位置的前缀中:公共”非本身的前缀和后缀”里最长的长度

next[j]代表：在这个j+1的位置匹配失败以后，回退到哪个位置，比如：

\*\*\*\*\*\*ABA**E**\*\*\*\*\*\*\*\*\*\*\*\*

ABA**B**ABDABD

next[2]=1

j=3处,E和B匹配失败，j怎么移动，要看next[2]，也就是next[j]是作用于s2[j+1]的

匹配的代码是：

1. **while**(i<len1){
2. **if**(s1[i]==s2[j]){
3. i++;
4. j++;
5. }**else**{
6. **if**(j==0){  //如果j是0的位置，在这个位置失败，直接移动s1的i即可
7. i++;
8. }**else**{
9. j=nex[j-1];
10. }
11. }
12. **if**(j==len2)
13. **return** i-j;
14. }

### 求Next数组求

next[j]定义是：next[j]是字符串的j位置的前缀中:公共”非本身的前缀和后缀”里最长的长度

所以可以直接按照定义求解。

**ABACDABD**

next[0]：字符串A,没有非本身的前缀后缀，next[0]=0(任何字符串的next[0]=0)

next[1]：字符串AB,前缀:A 后缀:B 没有公共字符串，next[1]=0

next[2]：字符串ABA,前缀:A,AB 后缀 公共字符串是A，长度1，next[2]=1

next[3]：字符串ABAC,前缀:A,AB,ABA 后缀:C,AC,BAC，没公共字符串,next[3]=0

next[4]：字符串ABACD,前缀:A,AB,ABA, ABAC 后缀:D,CD,ACD,BACD，没公共字符串,next[4]=0

next[5]：字符串ABACDA,前缀:A,AB,ABA,ABAC,ABACD 后缀: A,DA,CDA,ACDA,BACDA，公共字符串A,next[5]=1

next[6]：字符串ABACDAB,前缀:A,AB,ABA,ABAC,ABACD, ABACDA

后缀: B,AB,DAB,CDAB,ACDAB,BACDAB，公共字符串AB,next[6]=2

next[7]：字符串ABACDABD,前缀:A,AB,ABA,ABAC,ABACD, ABACDA, ABACDA, ABACDAB

后缀: D,BD,ABD,DABD,CDABD,ACDABD,BACDABD，没公共字符串,next[7]=0

字符串next数组是：0 0 1 0 0 1 2 0 0

这么做找前缀后缀显然很慢，不符合KMP算法初衷(线性时间完成字符串匹配)，这么做显然会让时间复杂度大于o(len1+len2),如果想让时间复杂度线性，就得连求next数组再匹配，都得是线性复杂度，要求对于每个位置的next[j]要用常数时间求得。

显然next[0]=0,且寻找ABACDABD中j位置的next值，只需要考虑j-1位置，而不需要考虑j+1以及后面的位置的字符是怎么样的，这就是所谓无后效性。

用递推思路，介绍已知next[j]，求next[j+1]的方法：

若next[j]=0,说明s2前j位置的字符串没有公共”非本身的前缀和后缀”，此时若s1[0]== s2 [j+1]则next[j+1]=next[j]+1=1,否则next[j+1]=0

若next[j]=1,说明s2前j位置的字符串的公共”非本身的前缀和后缀”里最大长度是1，一定有结论s2[0]==s2[j]，此时若s1[1]= =s2 [j+1]则next[j+1]=next[j]+1=2,否则next[j+1]=0

若next[j]=2,说明s2前j位置的字符串的公共”非本身的前缀和后缀”里最大长度是2，一定有结论s2[0]==s2[j-1] && s2[1]==s2[j]，此时若s1[2]==s2[j+1]则next[j+1]=next[j]+1=3,否则next[j+1]=0

…….

若next[j]=n,n<j说明s2前j位置的字符串的公共”非本身的前缀和后缀”里最大长度是n，一定有结论: s2[0]==s2[j-n+1] && s2[1]==s2[j-n+2] &&s2[2]==s2[j-n+3]…..&& s2[n-1]==s2[j-n+n]

此时若s1[n]==s2[j+1] (其实是 s1[next[j]]==s2[j+1])

则next[j+1]=next[j]+1=n+1,否则next[j+1]=0

这样的next数组可以保证可以用来匹配的，时间复杂度也是正确的。

但是，在由于s1[n]!=s2[j+1]时每次都令next[j+1]=0

，意味着在j+2位置匹配失败会令j指针回退到0，这样相比接下来说的优化方法，在匹配时候还是会多匹配一些次，时间复杂度常数略高，也不完全符合next数组最初定义。

实际上s1[n]!=s2[j+1]时候，某些情况不必回退到0，

如：求abcabcabcaba中j=11时的next值

已知前几位：0 0 0 1 2 3 4 5 6 7 8

显然s1[next[10]]!=s2[11]，此时如果令next[11]=0,完全可以，但不符合next数组最初定义，因为实际abcabcabcaba的公共”非本身的前缀和后缀”是a,长度是1

当比较出s1[next[10]]!=s2[11]时，不应该急于令next[11]=0，

而是继续比较s1[next[next[10]-1]]和s2[11] => s1[5]和s2[11]显然是不相同的，

而是继续比较s1[next[next[next[10]-1]-1]]和s2[11] => s1[2]和s2[11]显然是不相同的，

而是继续比较s1[next[next[next[next[10]-1]-1]-1]]和s2[11] => s1[0]和s2[11]显然是相同的，

next[11]= next[next[next[next[10]-1]-1]-1] +1 =1

由上述规律,求解next数组的方法明确了

1. nex[0]=0;
2. **for**(j=1;j<len2;j++){
3. k=nex[j-1];
4. **while**(k>0&&s2[k]!=s2[j]){
5. k=nex[k-1];
6. }
7. **if**(s2[k]!=s2[j]){
8. nex[j]=k;
9. }**else**{
10. nex[j]=k+1;
11. }
12. }

## next版本二：

### 思路

第二版的next数组和第一版的next数组作用完全相同，它们完全是2个等价写法。

第一版中， next数组定义：next[j]是字符串的j位置的前缀中:公共”非本身的前缀和后缀”里最长的长度。 next[0]=0,且对于next[j],它的数值决定了s2[j+1]匹配失败后，j应该怎么移动。

第二版中，next数组定义变为：next[j]是字符串的j-1位置的前缀中:公共”非本身的前缀和后缀”里最长的长度。 next[0]=-1，且对于next[j],它的数值决定了s2[j]匹配失败后，j应该怎么移动。相当于版本一全都移动一位。在第二版中，循环往前找也变成了：

1. **while**(k>-1&&s2[k]!=s2[j])
2. k=nex[k];

next[0]=-1是说，s2在j=0这个位置匹配失败，j移动到-1，也就是字符串的前驱

\*\*\*\*\*\***A**BC\*\*\*\*\*\*\*\*\*\*\*\*\* =》 \*\*\*\*\*\***A**BC\*\*\*\*\*\*\*\*\*\*\*\*\*

**B**BACABDABD =》 BBACABDABD

\*\*\*\*\*\*A**B**C\*\*\*\*\*\*\*\*\*\*\*\*\*

**B**BACABDABD

前驱是空字符，一定不和任何字符匹配，所以处理方法是再令i++;j++;

### 求解next数组示例

1. nex[0]=-1;
2. **for**(j=1;j<len2;j++){
3. k=nex[j-1];
4. **while**(k>-1&&s2[k]!=s2[j-1]){
5. k=nex[k];
6. }
7. nex[j]=k+1;
8. }

求解abcabcabcaba的next数组

k=0 ;j=1, next[0]=-1

求next[1]：j=1,k=next[0]=-1 ; next[1]=0

求next[2]：j=2,k=next[1]=0 ; s2[k]!=s2[1] => k=-1, next[2]=0

求next[3]：j=3,k=next[2]=0 ; s2[k]!=s2[2] => k=-1, next[3]=0

求next[4]：j=4,k=next[3]=0 ; s2[k]==s2[3] ,next[4]=k+1=1

求next[5]：j=5,k=next[4]=1 ; s2[k]==s2[4] ,next[5]=k+1=2

求next[6]：j=6,k=next[5]=2 ; s2[k]==s2[5] ,next[6]=k+1=3

求next[7]：j=7,k=next[6]=3 ; s2[k]==s2[6] ,next[7]=k+1=4

求next[8]：j=8,k=next[7]=4 ; s2[k]==s2[7] ,next[8]=k+1=5

求next[9]：j=9,k=next[8]=5 ; s2[k]==s2[8] ,next[9]=k+1=6

求next[10]：j=10,k=next[9]=6 ; s2[k]==s2[9] ,next[10]=k+1=7

求next[11]：j=11,k=next[10]=7 ; s2[k]!=s2[10] => k=next[7]=4 s2[k]!=s2[10] => k=next[4]=1

s2[k]!=s2[10] => k=next[1]=0 s2[k]==s2[10] next[11]=k+1=1

### 说明：

next版本二是广为流传的next数组，它实际就是版本一的每一位”向后挪动”

有两种形式，next[0]=0和next[0]=-1，这两种下第一种可以通过第二种每位加一得到，本部分的说明都是拿next[0]=-1为例子的，实际使用上next[0]=-1的方便一点。

next[0]=0也是考研数据结构的next[]数组

## next版本三：

### 思路：

版本三是对前二者的改进，不论前2者哪个版本的kmp作用是一样的，只是相互等价，这里next数组一第二版为例子,观看以下示例abababa的next是：-1 0 0 1 2 3 4

\*\*\*\*\*\*\*abababz\*\*\*\*\*\*\*\*

abababa

匹配到a和z时，发现不匹配，移动s2

\*\*\*\*\*\*\*abababz\*\*\*\*\*\*\*\*

abababa

这时发现还不匹配，还要移动，最终实际上这个字符串任何一个位置，都根本不和z匹配，因为abababa就里没有z.

所以这里的next[6]=4在某些情况下，一点意义没有(需要增多匹配次数)，不如t[6]=0实用

回顾求next数组的next[6]的过程，此时可以发现，比较s2[5]和s2[3]发现它们相等，令next[6]=5。

现在回顾定义：next数组的定义是什么？答：”next[j]是字符串的j位置的前缀中:公共”非本身的前缀和后缀”里最长的长度””。它的目的是什么？答：”尽可能少的让s2移动，尽可能能多的预先知道，并且规避掉那些，s1[i]和s2[j]明显就不匹配的情况的”

而abababa求next[6]时，这时按照定义，是通过next[5]递推转移得到，而next[5]=3是表明

ababab的最长公共的前缀后缀是abab

求next[6]时，除了之前比较的s2[5]和s2[3]之外，还发现一件事：s2[6]和s2[4]也相等

既然s1[i]!=s2[6]了，那也必然s1[i]!=s2[4].这说明：尽管s2[5]==s2[3]，但3这个位置继续比较不可能使s1和s2匹配，应该当做s2[5]!=s2[3]处理。(相当于代码里的k=next[k])

所以代码变成了：

1. nex[0]=-1;k=-1;
2. **for**(j=1;j<len2;j++){
3. **while**(k>-1&&s2[k]!=s2[j-1]){
4. k=nex[k];
5. }
6. k++;
7. nex[j]=s2[k]==s2[j]?nex[k]:k;
8. }

上述代码是在s2[k]!=s2[j-1]或者k==-1时，继续比较s2[k+1]和s2[j]，如果他们不等说明k这个位置还有可能使s1和s2匹配，就照常按照版本二的方式处理，否则就当做s2[k]!=s2[j-1]

来处理，这里不必循环匹配到最开始，因为求next是一个递推的过程，前一部分保证

s2[k+1]!=s2[j]。所以直接领next[j]= next[k]即可

### 说明：

这个版本下匹配s1和s2的代码和版本二相同。但定义上，这种改进的next数组不再和之前next数组定义相同，网上也叫nextval数组

nextval有两种形式，nextval [0]=0和nextval [0]=-1，这两种下第一种可以通过第二种每位加一得到，本部分的说明都是拿nextval [0]=-1为例子的，实际使用上nextval [0]=-1的方便一点。nextval [0]=0也是考研数据结构的nextval []数组

## KMP记忆方法

KMP的next数组不尽相同，考研主要是对next和nextval记忆，统一记忆next[0]=-1的形式，之后再进行转化，只要记住了next数组是

1. nex[0]= k=-1;
2. **for**(j=1;j<len2;j++){
3. k=nex[j-1];
4. **while**(k>-1&&s2[k]!=s2[j-1]){
5. k=nex[k];
6. }
7. k++;
8. nex[j]=k;
9. }

而nextval和next代码不同点

1. 循环里k不需要初始值而是沿用上次的
2. 对nextval [j]赋值时判断s2[k]==s2[j]相等说明：有可能匹配，就赋值成nextval [k]；不等说明不可能在这个位置匹配，就和next的一样赋值成k

## KMP扩展

以下都按照next版本三进行解释

### 匹配多个字符：

就是如果s1中出现了很多个的s2，如何找到所以位置?

每次找到一个匹配的点时，把这个点存起来或者打印出来，按照没匹配处理，所谓”按照没匹配处理”，就是指代码里的：k=next[k]

1. **void** KMP2(**char** \*s1,**char** \*s2,**int** \*nex){//在s1中找s2,找全部出现位置
2. **int** i=0,j=0,len1=(**int**)strlen(s1),len2=(**int**)strlen(s2);
3. **while**(i<len1&&j<len2){
4. **if**(j==-1||s1[i]==s2[j]){
5. i++;
6. j++;
7. }**else**{
8. j=nex[j];
9. }
10. **if**(j==len2){
11. printf("%d\n",i-len2);
12. j=nex[j];
13. }
14. }
15. }

### 字符串最小循环节

如果是next[0]=0的数组看next[len-1]的数值，如果是next[0]=-1的数组，随便加一个字母，看next[len]的数值。设这个数值是a，有如下定理

定理：假设S的长度为len，且S存在最小循环节，循环节的长度L=len-a

（1）如果len可以被len - a整除，则表明字符串S可以完全由循环节循环组成，循环周期T=len/L。

（2）如果不能，说明还需要再添加几个字母才能补全。需要补的个数是循环个数L-len%L=L-(len-L)%L=L-a %L，L=len-a。

### 与AC自动机失败指针的联系

Ac自动机fail指针就是KMP的next数组再树上的表示形式