KMP

问题

文本串S,模式串P，寻找P在S中的位置

方法

暴力匹配

```
int ViolentMatch(char* s, char* p)
{
    int sLen = strlen(s);
    int pLen = strlen(p);

    int i = 0;
    int j = 0;
    while (i < sLen && j < pLen)
    {
        if (s[i] == p[j])
        {
            //①如果当前字符匹配成功（即S[i] == P[j]），则i++, j++
            i++;
            j++;
        }
        else
        {
            //②如果失配（即S[i]！= P[j]），令i = i - (j - 1), j = 0
            i = i - j + 1;
            j = 0;
        }
    }
    //匹配成功，返回模式串p在文本串s中的位置，否则返回-1
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

匹配与不匹配的解决方案

- S[i]与P[j]匹配成功　i++,j++
- S[i]与P[j]匹配不成功　回溯
  - i=i-(j-1),j=0
  - 也就是i回退当前i所走的长度，然后j指向第一个元素，重新开始

例子

5. 直到S[10]为空格字符，P[6]为字符D（i=10，j=6），因为不匹配，重新执行第②条指令：“如果失配（即S[i] ≠ P[j]），令i = i - (j - 1), j = 0”，相当于S[5]跟P[0]匹配（i=5，j=0）

```
BBC ABCDAB ABCDABCDABDE
     ABCDABD
```

6. 至此，我们可以看到，如果按照暴力匹配算法的思路，尽管之前文本串和模式串已经分别匹配到了S[9]、P[5]，但因为S[10]跟P[0]不匹配，所以文本串回溯到S[5]，模式串回溯到P[0]，从而让S[5]跟P[0]匹配。

```
BBC ABCDAB ABCDABCDABDE
     ABCDABD
```

缺点

这种回退方法会导致重复的计算，因为在前面的匹配过程中，我们已经肯定的知道S[5]=P[1]!=P[0],因此引入KMP，来回退i而不是i

KMP

思想

利用之前匹配过的有效信息，来保存i不回退，而回退j。让模式串移动到有效的位置：引入了next数组来记录下一次i应该回退的位置

算法

```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++, j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

1.求解next数组的方法

1.首先求解最大前缀后缀公共元素长度

3.3.1 寻找最长前缀后缀

如果给定的模式串是：“ABCDABD”，从左至右遍历整个模式串，其各个子串的前缀后缀分别如下表格所示：

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

也就是说，原模式串子串对应的各个前缀后缀的公共元素的最大长度表为（**下简称《最大长度表》**）：

字符	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0

2.然后进行变化求得next数组

- next 数组考虑的是除当前字符外的最长相同前缀后缀，所以通过第①步骤求得各个前缀后缀的公共元素的最大长度后，只要稍作变形即可：将第①步骤中求得的值整体右移一位，然后初值赋为-1，如下表格所示：

模式串	a	b	a	b
next数组	-1	0	0	1

```
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
```

2.还可以用递推方式来求next数组，已知next[0,...,j]，如何求next[j+1]，这里就略了

例子

继续拿之前的例子来说，当S[10]跟P[6]匹配失败时，KMP不是跟暴力匹配那样简单的把模式串右移一位，而是执行第②条指令：“如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令i 不变，j = next[j]”，即j从6变到2（后面我们将求得P[6]，即字符D对应的next 值为2），所以相当于模式串向右移动的位数为j - next[j]（j - next[j] = 6-2 = 4）。

```
BBC ABCDAB ABCDABCDABDE
     ABCDABD
```

向右移动4位后，S[10]跟P[2]继续匹配。为什么要向右移动4位呢，因为移动4位后，模式串中又有个“AB”可以继续跟S[8]S[9]对应着，从而不用让i 回溯。相当于在除去字符D的模式串子串中寻找相同的前缀和后缀，然后根据前缀后缀求出next 数组，最后基于next 数组进行匹配（不关心next 数组是怎么求来的，只想看匹配过程是咋样的，可直接跳到下文3.3.4节）。

```
BBC ABCDAB ABCDABCDABDE
        ABCDABD
```

```
//优化过后的next 数组求法
void GetNextval(char* p, int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            //较之前next数组求法，改动在下面4行
            if (p[j] != p[k])
                next[j] = k;   //之前只有这一行
            else
                //因为不能出现p[j] = p[ next[j ]]，所以当出现时需要继续递归，k = next[k] = ne
xt[next[k]]
                next[j] = next[k];
        }
        else
        {
            k = next[k];
        }
    }
}
```

next数组的优化　出现p[j]=p[next[j]]时，需要再次递归，令next[j]=next[next[j]]

时间复杂度分析　如果文本串的长度为 n，模式串的长度为 m，那么匹配过程的时间复杂度为 O(n)，算上计算 next 的 O(m) 时间，KMP 的整体时间复杂度为 O(m + n)。

参考链接：http://wiki.jikexueyuan.com/project/kmp-algorithm/define.html