KMP 算法总结

看了很多遍 KMP 算法，总是似是而非，顾记录博客，总结与此，仅供自身复习与知识分享所用。

1. 暴力匹配法

假设现在我们面临这样一个问题：有一个文本串S，和一个模式串P，现在要查找P在S中的位置，怎么查找呢？

如果用暴力匹配的思路，并假设现在文本串S匹配到 i 位置，模式串P匹配到 j 位置，则有：

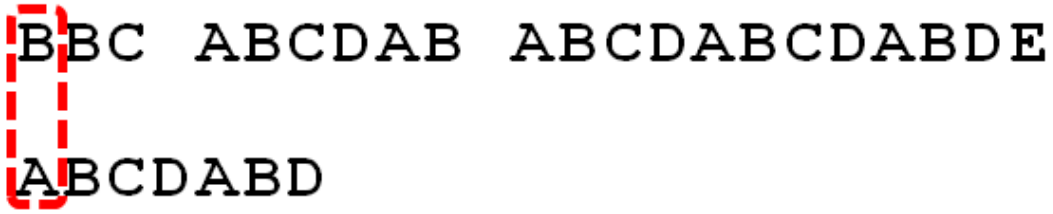
如果当前字符匹配成功（即S[i] == P[j]），则i++，j++，继续匹配下一个字符； 如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0。相当于每次匹配失败时，i 回溯，j 被置为0。

理清楚了暴力匹配算法的流程及内在的逻辑，咱们可以写出暴力匹配的代码，如下：

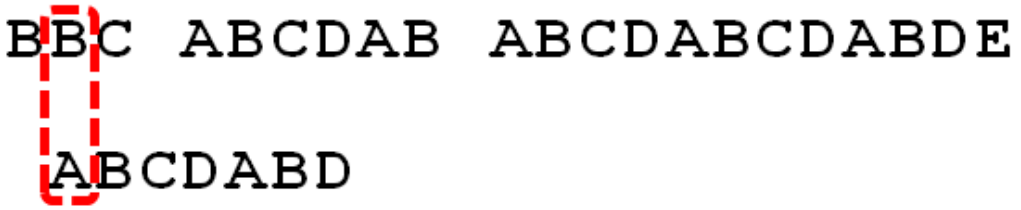
```
int ViolentMatch(char* s, char* p)
{
    int sLen = strlen(s);
    int pLen = strlen(p);

    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;
}
```

举个例子，如果给定文本串S“BBC ABCDAB ABCDABCDABDE”，和模式串P“ABCDABD”，现在要拿模式串P去跟文本串S匹配，整个过程如下所示：

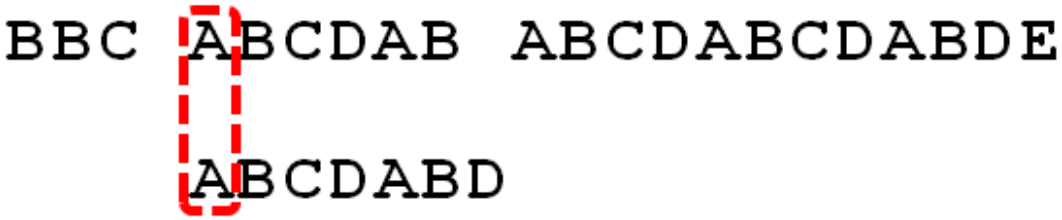
1. S[0]为B，P[0]为A，不匹配，执行第②条指令：“如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0”，S[1]跟P[0]匹配，相当于模式串要往右移动一位（i=1，j=0）



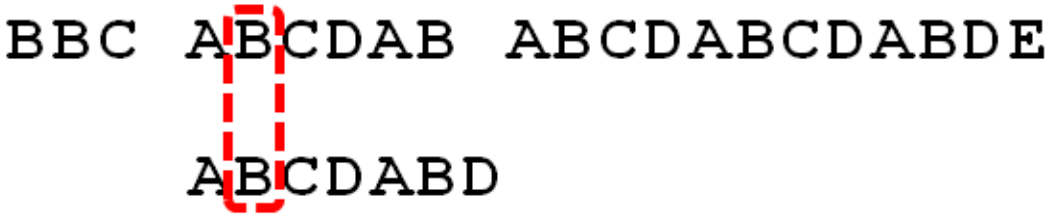
2. S[1]跟P[0]还是不匹配，继续执行第②条指令：“如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0”，S[2]跟P[0]匹配（i=2，j=0），从而模式串不断的向右移动一位（不断的执行“令i = i - (j - 1)，j = 0”，i从2变到4，j一直为0）



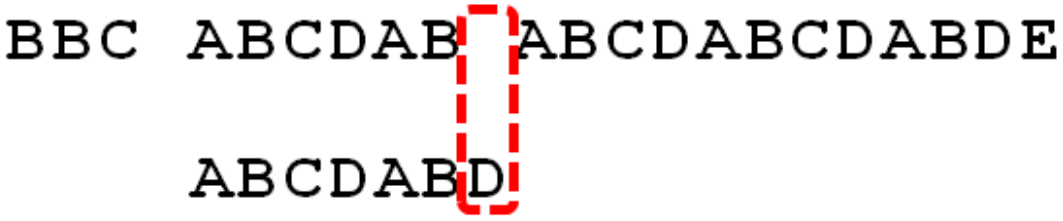
3. 直到S[4]跟P[0]匹配成功（i=4，j=0），此时按照上面的暴力匹配算法的思路，转而执行第①条指令：“如果当前字符匹配成功（即S[i] == P[j]），则i++，j++”，可得S[i]为S[5]，P[j]为P[1]，即接下来S[5]跟P[1]匹配（i=5，j=1）



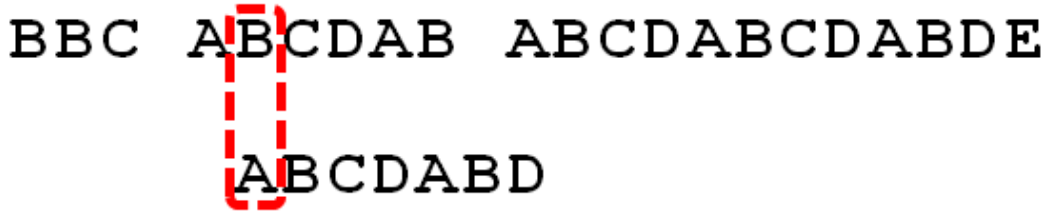
4. S[5]跟P[1]匹配成功，继续执行第①条指令：“如果当前字符匹配成功（即S[i] == P[j]），则i++，j++”，得到S[6]跟P[2]匹配（i=6，j=2），如此进行下去



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）



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[6]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。



而S[5]肯定跟P[0]失配。为什么呢？因为在之前第4步匹配中，我们已经得知S[5] = P[1] = B，而P[0] = A，即P[1] != P[0]，故S[5]必定不等于P[0]，所以回溯过去必然会导致失配。那有没有一种算法，让i 不往回退，只需要移动j 即可呢？

答案是肯定的。这种算法就是本文的主旨KMP算法，它利用之前已经部分匹配这个有效信息，保持i 不回溯，通过修改j 的位置，让模式串尽量地移动到有效的位置。

2. KMP算法

2.1 定义

Knuth-Morris-Pratt 字符串查找算法，简称为 “KMP算法”，常用于在一个文本串S内查找一个模式串P的出现位置，这个算法由Donald Knuth、Vaughan Pratt、James H. Morris三人于1977年联合发表，故取这3人的姓氏命名此算法。

下面先直接给出KMP的算法流程（如果感到一点点不适，没关系，坚持下，稍后会有具体步骤及解释，越往后看越会柳暗花明）：

假设现在文本串S匹配到 i 位置，模式串P匹配到 j 位置

- 如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++，继续匹配下一个字符；
- 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]。此举意味着失配时，模式串P相对于文本串S向右移动了j - next [j] 位。

换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值（next 数组的求解会在下文的3.3.3节中详细阐述），即移动的实际位数为：j - next[j]，且此值大于等于1。

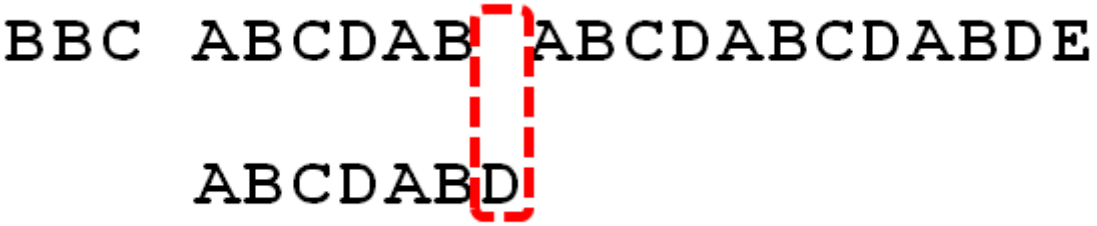
很快，你也会意识到next 数组各值的含义：代表当前字符之前的字符串中，有多大长度的相同前缀后缀。例如如果next [j] = k，代表j 之前的字符串中有最大长度为k 的相同前缀后缀。

此也意味着在某个字符失配时，该字符对应的next 值会告诉你下一步匹配中，模式串应该跳到哪个位置（跳到next [j] 的位置）。如果next [j] 等于0或-1，则跳到模式串的开头字符，若next [j] = k 且 k > 0，代表下次匹配跳到j 之前的某个字符，而不是跳到开头，且具体跳过了k 个字符。

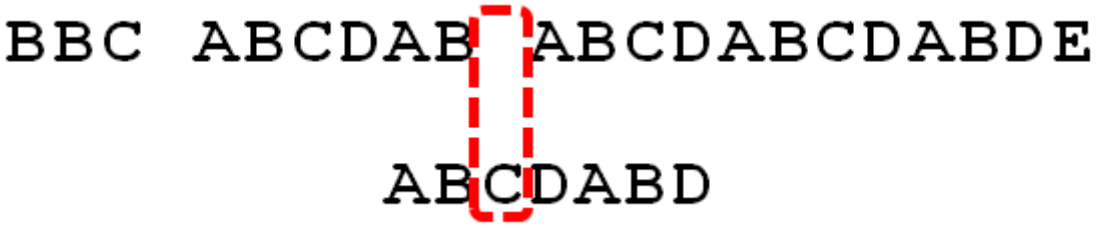
```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
```

```
        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;
}
```

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）。



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）。



2.2 步骤

1. 寻找前缀后缀最长公共元素长度 对于P = p0 p1 ...pj-1 pj，寻找模式串P中长度最大且相等的前缀和后缀。如果存在p0 p1 ...pk-1 pk = pj- k pj-k+1...pj-1 pj，那么在包含pj的模式串中有最大长度为k+1的相同前缀后缀。举个例子，如果给定的模式串为“abab”，那么它的各个子串的前缀后缀的公共元素的最大长度如下表格所示：

模式串	a	b	a	b
最大前缀后缀公共元素长度	0	0	1	2

比如对于字符串aba来说，它有长度为1的相同前缀后缀a；而对于字符串abab来说，它有长度为2的相同前缀后缀ab（相同前缀后缀的长度为k + 1，k + 1 = 2）。

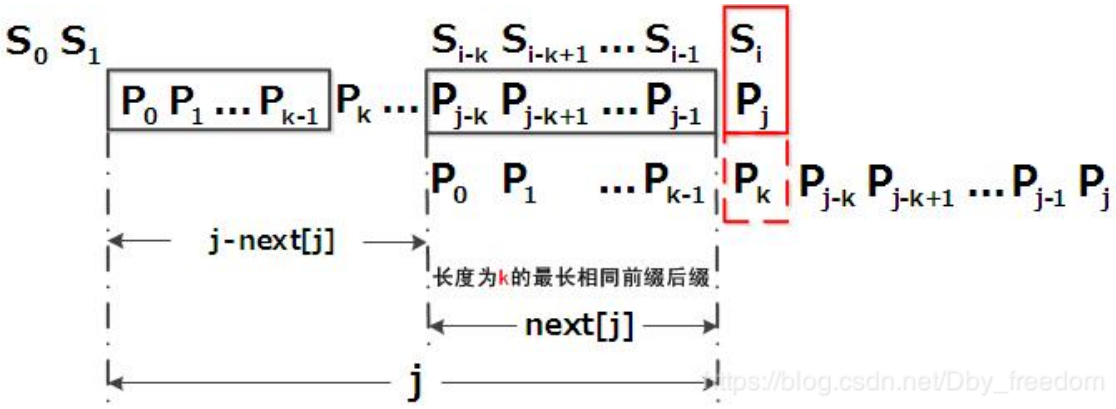
2. 求next数组

   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

   比如对于aba来说，第3个字符a之前的字符串ab中有长度为0的相同前缀后缀，所以第3个字符a对应的next值为0；而对于abab来说，第4个字符b之前的字符串aba中有长度为1的相同前缀后缀a，所以第4个字符b对应的next值为1（相同前缀后缀的长度为k，k = 1）。

3. 根据next数组进行匹配 匹配失配，j = next [j]，模式串向右移动的位数为：j - next[j]。换言之，当模式串的后缀pj-k pj-k+1, ..., pj-1 跟文本串si-k si-k+1, ..., si-1匹配成功，但pj 跟si匹配失败时，因为next[j] = k，相当于在不包含pj的模式串中有最大长度为k 的相同前缀后缀，即p0 p1 ...pk-1 = pj-k pj-k+1...pj-1，故令j = next[j]，从而让模式串右移j - next[j] 位，使得模式串的前缀p0 p1, ..., pk-1对应着文本串 si-k si-k+1, ..., si-1，而后让pk 跟si 继续匹配。如下图所示：



   综上，KMP的next 数组相当于告诉我们：当模式串中的某个字符跟文本串中的某个字符匹配失配时，模式串下一步应该跳到哪个位置。如模式串中在j 处的字符跟文本串在i 处的字符匹配失配时，下一步用next [j] 处的字符继续跟文本串i 处的字符匹配，相当于模式串向右移动 j - next[j] 位。

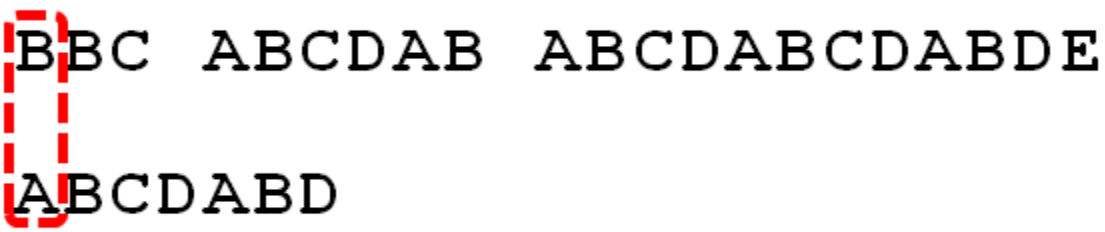
   接下来，分别具体解释上述3个步骤。

3. 编程思路

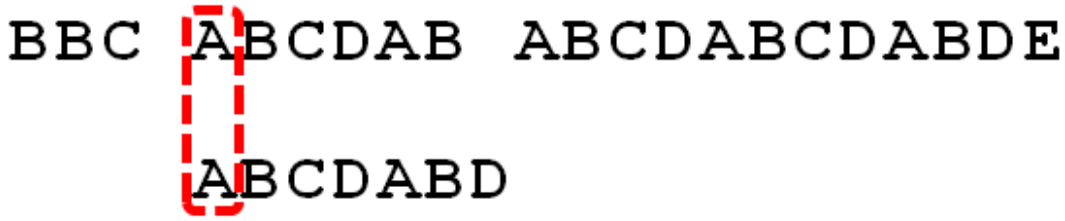
3.1 基于《最大长度表》匹配

因为模式串中首尾可能会有重复的字符，故可得出下述结论：

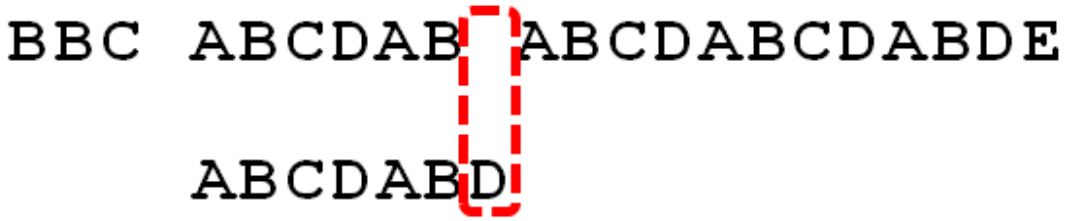
失配时，模式串向右移动的位数为：已匹配字符数 - 失配字符的上一位字符所对应的最大长度值 下面，咱们就结合之前的《最大长度表》和上述结论，进行字符串的匹配。如果给定文本串“BBC ABCDAB ABCDABCDABDE”，和模式串“ABCDABD”，现在要拿模式串去跟文本串匹配，如下图所示：



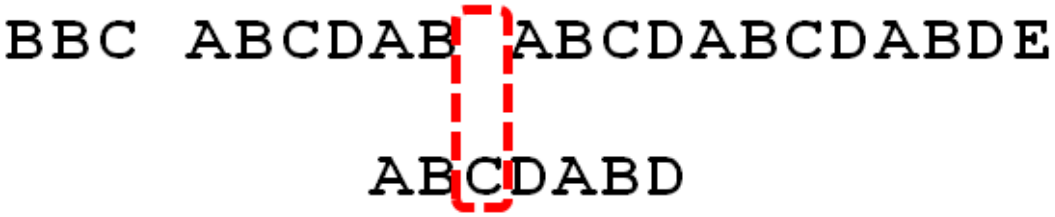
1. 因为模式串中的字符A跟文本串中的字符B、B、C、空格一开始就不匹配，所以不必考虑结论，直接将模式串不断的右移一位即可，直到模式串中的字符A跟文本串的第5个字符A匹配成功：



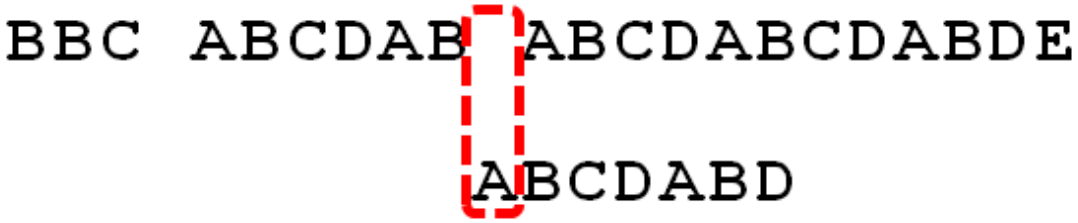
2. 继续往后匹配，当模式串最后一个字符D跟文本串匹配时失配，显而易见，模式串需要向右移动。但向右移动多少位呢？因为此时已经匹配的字符数为6个（ABCDAB），然后根据《最大长度表》可得失配字符D的上一位字符B对应的长度值为2，所以根据之前的结论，可知需要向右移动6 - 2 = 4 位。



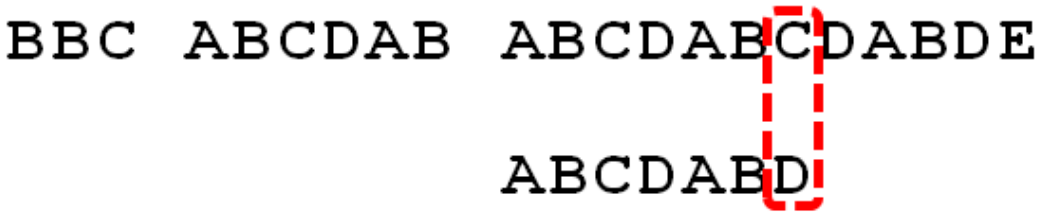
3. 模式串向右移动4位后，发现C处再度失配，因为此时已经匹配了2个字符（AB），且上一位字符B对应的最大长度值为0，所以向右移动：2 - 0 =2 位。



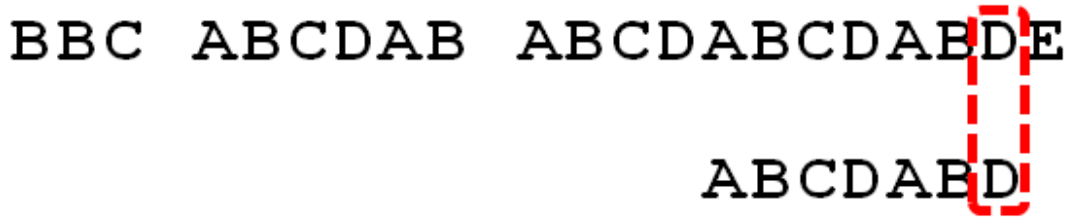
4. A与空格失配，向右移动1 位。



5. 继续比较，发现D与C 失配，故向右移动的位数为：已匹配的字符数6减去上一位字符B对应的最大长度2，即向右移动6 - 2 = 4 位。



6. 经历第5步后，发现匹配成功，过程结束。



通过上述匹配过程可以看出，问题的关键就是寻找模式串中最大长度的相同前缀和后缀，找到了模式串中每个字符之前的前缀和后缀公共部分的最大长度后，便可基于此匹配。而这个最大长度便正是next 数组要表达的含义。

3.2 根据《最大长度表》求next 数组

由上文，我们已经知道，字符串“ABCDABD”各个前缀后缀的最大公共元素长度分别为：

模式串	A	B	C	D	A	B	D
前后缀最大公共元素长度	0	0	0	0	1	2	0

而且，根据这个表可以得出下述结论

失配时，模式串向右移动的位数为：已匹配字符数 - 失配字符的上一位字符所对应的最大长度值

上文利用这个表和结论进行匹配时，我们发现，当匹配到一个字符失配时，其实没必要考虑当前失配的字符，更何况我们每次失配时，都是看的失配字符的上一位字符对应的最大长度值。如此，便引出了next 数组。

给定字符串“ABCDABD”，可求得它的next 数组如下：

模式串	A	B	C	D	A	B	D
next	-1	0	0	0	0	1	2

把next 数组跟之前求得的最大长度表对比后，不难发现，next 数组相当于“最大长度值” 整体向右移动一位，然后初始值赋为-1。意识到了这一点，你会惊呼原来next 数组的求解竟然如此简单：就是找最大对称长度的前缀后缀，然后整体右移一位，初值赋为-1（当然，你也可以直接计算某个字符对应的next 值，就是看这个字符之前的字符串中有多大长度的相同前缀后缀）。

换言之，对于给定的模式串：ABCDABD，它的最大长度表及next 数组分别如下：

模式串	A	B	C	D	A	B	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

根据最大长度表求出了next 数组后，从而有

失配时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值

而后，你会发现，无论是基于《最大长度表》的匹配，还是基于next 数组的匹配，两者得出来的向右移动的位数是一样的。为什么呢？因为：

根据《最大长度表》，失配时，模式串向右移动的位数 = 已经匹配的字符数 - 失配字符的上一位字符的最大长度值 而根据《next 数组》，失配时，模式串向右移动的位数 = 失配字符的位置 - 失配字符对应的next 值 其中，从0开始计数时，失配字符的位置 = 已经匹配的字符数（失配字符不计数），而失配字符对应的next 值 = 失配字符的上一位字符的最大长度值，两相比较，结果必然完全一致。

所以，你可以把《最大长度表》看做是next 数组的雏形，甚至就把它当做next 数组也是可以的，区别不过是怎么用的问题。

编程策略：

1. 当前字符的前面所有字符的对称程度为0的时候，只要将当前字符与前面这个子串的第一个字符进行比较。这个很好理解啊，前面所有字符串的对称值都是0，说明都不对称了，如果多加了一个字符，要对称的话只能是当前字符和前面字符串的第一个字符对称。比如“ABCDA”这个里面"ABCD"的最大对称值是0，那么后面的A的对称程度只需要看它是不是等于前面字符串的第一个字符A相等，如果相等就增加1，如果不相等那就保持不变，显然还是为0。
2. 按照这个推理，我们就可以总结一个规律，不仅前面是0呀，如果前面字符串的的最大对称值是1（k），那么我们就把当前字符与前面字符串的第二（k）个字符即P[1]（P[k]）进行比较，因为前面的是1（k），说明前面的字符已经和第一（k）个字符相等了，如果这个又与第二（k+1）个相等了，说明对称程度就是2（k+1）了。有两（k+1）个字符对称了。比如上面“ABCDA”的最大对称值是1，说明它只和第一个A对称了，接着我们就把下一个字符“B”与P[1]（即第二个字符）比较，又相等，自然对称程度就累加了，就是2了。

   但是如果不相等呢？那么这个对称值显然要减少，并且我们只能到前面去寻找对称值，而在找的过程我们同时也利用前缀函数表快速搜索找到与当前字符匹配的位置。比如假设是“（AGCTAGC）（AGCTAGC）T”（请无视字符串中的括号，只为方便看出对称），显然最后一个T的前一个位置的对称度是7,说明T的前一个位置的7个字符的后缀必与7个字符的前缀相等，然而T!=P[7]，说明T位置的对称度只能是比7小的长度的前缀，所以递减k值，递减为多少呢？当前字符前一个位置的对称度为k=next[13]=7，显然必须以7为基准减少，即在前缀长度为7以内的范围重新寻找以T结尾的前缀，所以k=next[6]，再接着判断是否相等,
3. 按照上面的推理，我们总是在找当前字符P[q]（q为遍历到的位置下标，见下面程序）通过其前一个位置的对称值判断是否与P[k]相等，如果相等，那么加，如果不相等，那么就减少k值，重新寻找与与P[q]相等的元素位置

```
  void makeNext(const char p[], int next[]) {
      int k;    //k:最大对称长度
      int length = strlen(P);        //模版字符串长度
      next[0] = 0;  //模版字符串的第一个字符的最大对称值必为0
      for (int cur = 1, k = 0; cur < length; ++cur) {        //for循环，从第二个字符开始，依次计算每一个字符对应的next值
          while (cur > 0 && P[cur] != P[k]) {        //while循环是整段代码的精髓所在，
              cur = next[cur-1];
          }
          if (P[k] == P[cur]) { //如果相等，那么最大相同前后缀长度加1
              k ++; //增加k的唯一方式
          }
          next[cur] = k;
      }
  }
```

4. KMP 代码

求 next 数组时，相当于自身与自身做匹配，以 k 当做自身匹配 string 索引，q 为自身做待匹配 string 索引，有如下代码：

```
#include "vector"
#include "string"
#include <iostream>
#include "algorithm"

using namespace std;

//计算模式P的部分匹配值，保存在next数组中
void MakeNext(const string& P, vector<int>& next)
{
    int q, k;//k记录所有前缀的对称值
    int m = P.size();//模式字符串的长度
    next[0] = 0;//首字符的对称值肯定为0
    for (q = 1, k = 0; q < m; ++q)//计算每一个位置的对称值
    {
        //k总是用来记录上一个前缀的最大对称值
        while (k > 0 && P[q] != P[k])
            k = next[k - 1];//k将循环递减，值得注意的是next[k]<k总是成立
        if (P[q] == P[k])
            k++;//增加k的唯一方法
        next[q] = k;//获取最终值
    }
}


void KmpMatch(const string & T, const string & P, vector<int> & next)
{
    int n, m;
    n = T.size();
    m = P.size();
    MakeNext(P, next);
    for (int i = 0, q = 0; i < n; ++i)
    {
        while (q > 0 && P[q] != T[i])
            q = next[q - 1];
        if (P[q] == T[i])
            q++;
        if (q == m)
        {
            cout << "模式文本的偏移为：" << (i - m + 1) << endl;
            q = next[q - 1];//寻找下一个匹配
        }
    }
}

int main()
{
    system("color 0A");
    vector<int> next(20, 0);//保存待搜索字符串的部分匹配表（所有前缀函数的对称值）
    string T = "xyxababcaxxxababca";//文本
    string P = "ababca";//待搜索字符串
    cout << "文本字符串：" << T << endl;
```

```
    cout << "模式字符串：" << P << endl;
    KmpMatch(T, P, next);
    cout << "模式字符串的前缀函数表：" << endl;
    for (int i = 0; i < P.size(); i++)
        cout << next[i];
    cout << endl;
    system("pause");
    return 0;
}
```

参考文献

[1] 字符串匹配---KMP算法 [2] 从头到尾彻底理解KMP（2014年8月22日版）