暴力搜索算法

```
int ViolentMatch(char* s, char* p)  {
    int sLen = strlen(s), pLen = strlen(p);
    int i = 0,  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
   return   j == pLen ? i - j: -1;
}
```

kmp算法

KMP的算法流程

- 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。

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

- 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

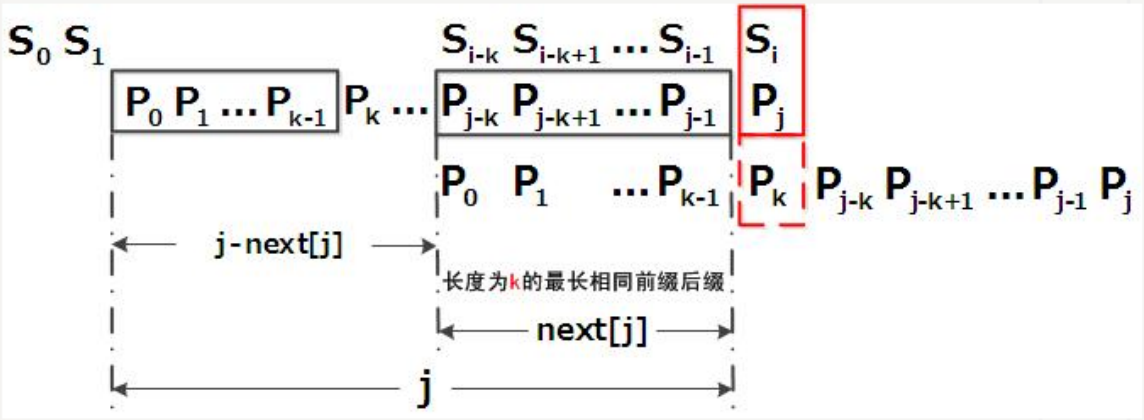
> 比如对于aba来说，第3个字符a之前的字符串ab中有长度为0的相同前缀后缀，所以第3个字符a对应的next值为0；而对于abab来说，第4个字符b之前的字符串aba中有长度为1的相同前缀后缀a，所以第4个字符b对应的next值为1（相同前缀后缀的长度为k，k = 1）。

- 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：



> next数组的值表示前个元素的最长公共元素长度

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。

解释

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

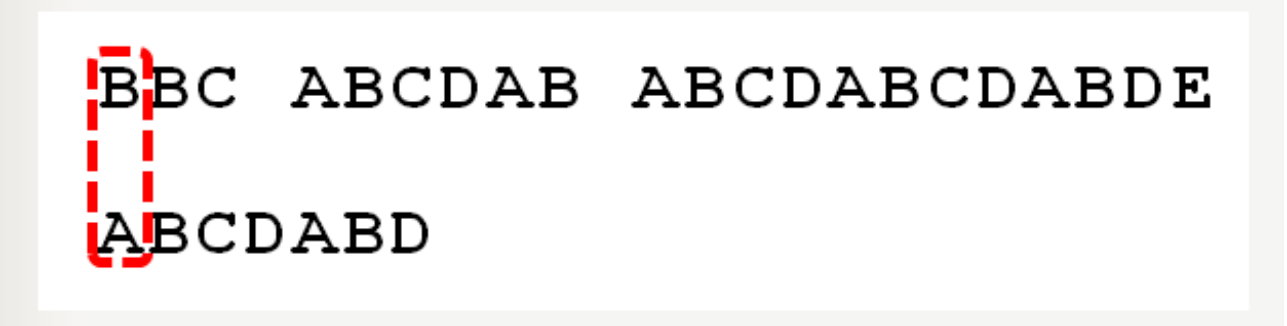
> 最大元素公共长度是由前缀和后缀得来的，考察的是模式串，而不是文本串。

基于《最大长度表》匹配

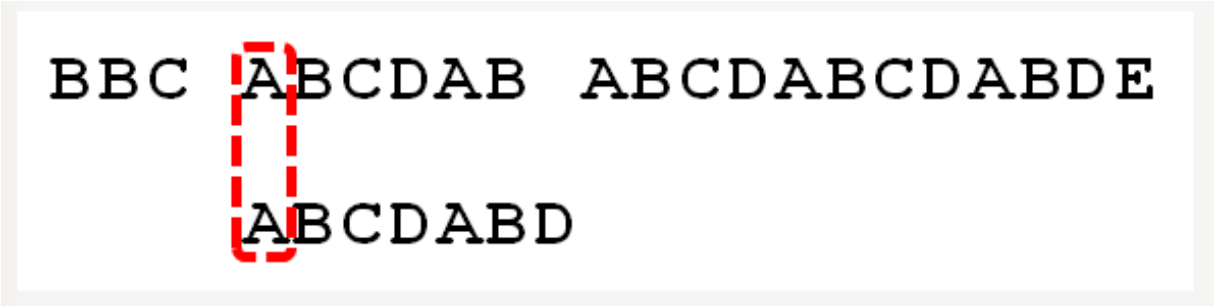
因为模式串中首尾可能会有重复的字符，故可得出下述结论：

> 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

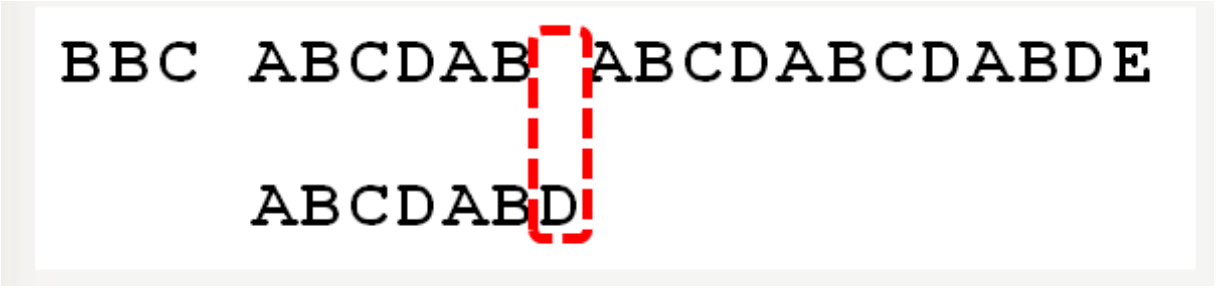
如果给定文本串“BBC ABCDAB ABCDABCDABDE”，和**模式串**“ABCDABD”，现在要拿模式串去跟文本串匹配，如下图所示：



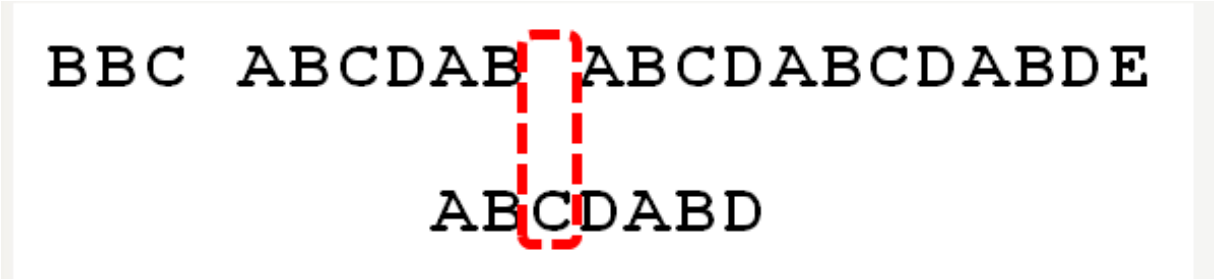
1. 因为模式串中的字符A跟文本串中的字符B、B、C、空格一开始就不匹配，所以不必考虑结论，直接将模式串不断的右移一位即可，直到模式串中的字符A跟文本串的第5个字符A匹配成功：



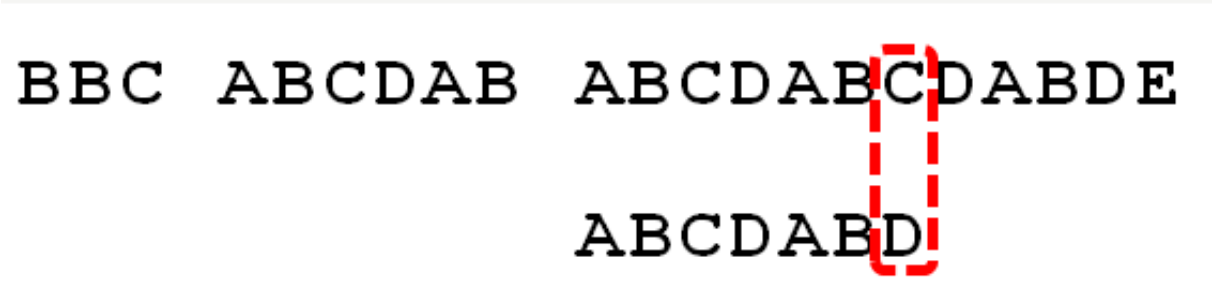
2. 继续往后匹配，当模式串最后一个字符D跟文本串匹配时失配，显而易见，模式串需要向右移动。但向右移动多少位呢？因为此时已经匹配的字符数为6个（ABCDAB），然后根据《最大长度表》可得**失配字符D的上一位字符B对应的长度值为2**（最大公共元素为AB），所以根据之前的结论，可知需要向右移动6 - 2 = 4 位。



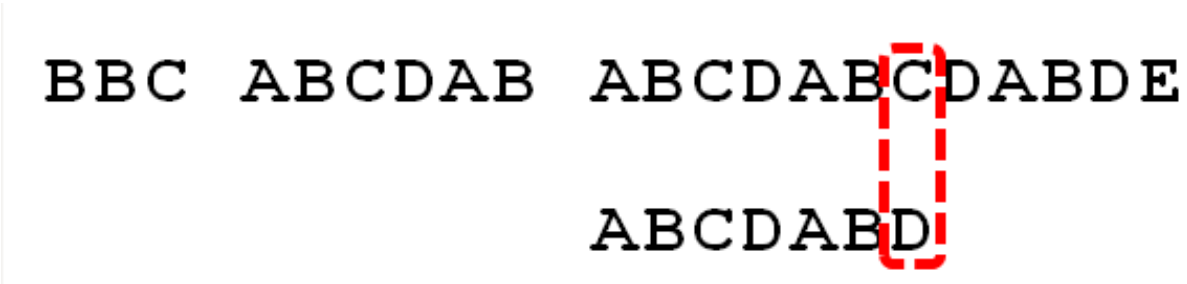
3. 模式串向右移动4位后，发现C处再度失配，因为此时已经匹配了2个字符（AB），且上一位字符B对应的最大长度值为0，所以向右移动：2 - 0 =2 位。（模式串C前面的最大公共长度为0）



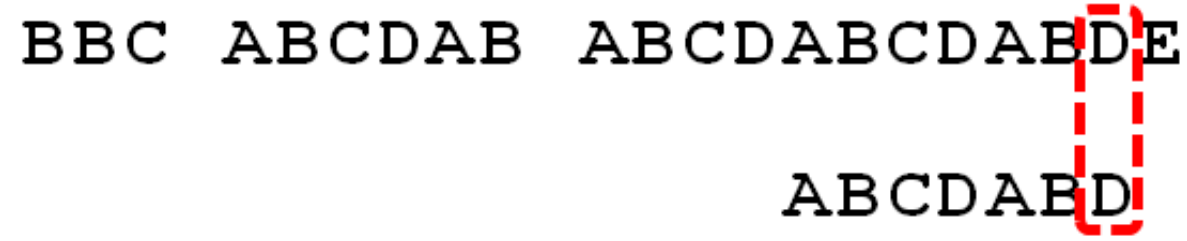
4. A与空格失配，向右移动1 位。



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**。

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

> 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

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

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

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。

通过代码递推计算next 数组

基于之前的理解，可知计算next 数组的方法可以采用递推：

1. 如果对于值k，已有p0 p1, ..., pk-1 = pj-k pj-k+1, ..., pj-1，相当于next[j] = k。
   此意味着什么呢？究其本质，next[j] = k 代表p[j] 之前的模式串子串中，有长度为k 的相同前缀和后缀。有了这个next 数组，在KMP匹配中，当模式串中j 处的字符失配时，下一步用next[j]处的字符继续跟文本串匹配，相当于模式串向右移动j - next[j] 位
2. 下面的问题是：已知next [0, ..., j]，如何求出next [j + 1]呢？

对于P的前j+1个序列字符：

> 1、若p[k] == p[j]，则next[j + 1] = next [j] + 1 = k + 1；

> 2、若p[k] ≠ p[j]，
> 如果此时p[next[k]] == p[j]，则next[j + 1] = next[k] + 1
> 否则继续递归前缀索引 k = next[k]，而后重复此过程。 相当于在字符p[j+1]之前不存在长度为k+1的前缀"p0 p1, …, pk-1 pk"跟后缀“pj-k pj-k+1, …, pj-1 pj"相等，那么是否可能存在另一个值t+1 < k+1，使得长度更小的前缀 “p0 p1, …, pt-1 pt” 等于长度更小的后缀 “pj-t pj-t+1, …, pj-1 pj” 呢？如果存在，那么这个t+1 便是next[j+1]的值，此相当于利用已经求得的next 数组（ next [0, ..., k, ..., j] ）进行P串前缀跟P串后缀的匹配。

可能还是不能很好的理解上述求解next 数组的原理，故接下来，我再来着重说明下。

- 如下图所示，假定给定模式串ABCDABCE，且已知next [j] = k（相当于“p0 pk-1” = “pj-k pj-1” = AB，可以看出k为2），现要求next [j + 1]等于多少？
  因为pk = pj = C，所以next[j + 1] = next[j] + 1 = k + 1（可以看出next[j + 1] = 3）。代表字符E前的模式串中，有长度k+1 的相同前缀后缀。

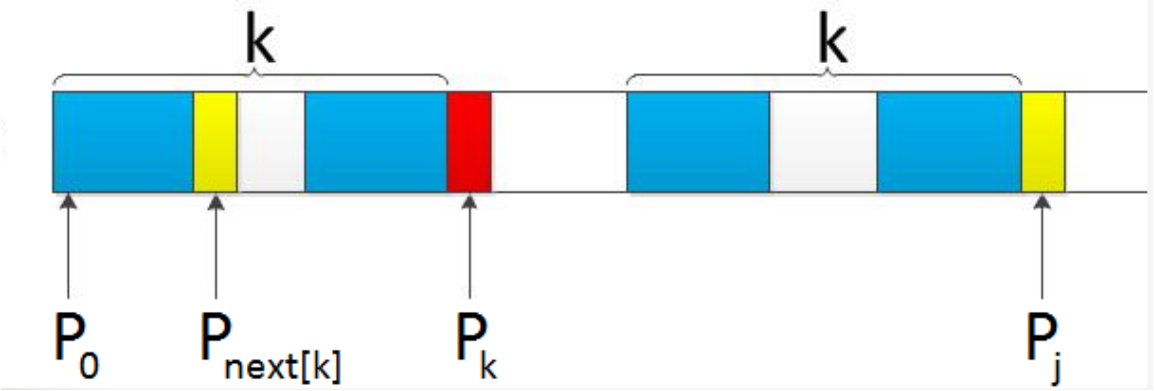
模式串	A	B	C	D	A	B	C	E
前后缀相同长度	0	0	0	0	1	2	3	0
next 值	-1	0	0	0	0	1	2	?
索引	p_0	p_{k-1}	p_k	p_{k+1}	p_{j-k}	p_{j-1}	p_j	p_{j+1}

- 但如果pk != pj 呢？说明“p0 pk-1 pk” ≠ “pj-k pj-1 pj”。换言之，当pk != pj后，字符E前有多大长度的相同前缀后缀呢？
  很明显，因为C不同于D，所以ABC 跟 ABD不相同，即字符E前的模式串没有长度为k+1的相同前缀后缀，也就不能再简单的令：next[j + 1] = next[j] + 1 。
  所以，咱们只能去寻找长度更短一点的相同前缀后缀。

模式串	A	B	C	D	A	B	D	E
前后缀相同长度	0	0	0	0	1	2	0	0
next 值	-1	0	0	0	0	1	2	?
索引	p_0	p_{k-1}	p_k	p_{k+1}	p_{j-k}	p_{j-1}	p_j	p_{j+1}

  结合上图来讲，若能在前缀“ p0 pk-1 pk ” 中不断的递归前缀索引k = next [k]，找到一个字符pk’ 也为D，代表pk’ = pj，且满足p0 pk'-1 pk' = pj-k' pj-1 pj，则最大

相同的前缀后缀长度为k' + 1，从而next [j + 1] = k' + 1 = next [k'] + 1。否则前缀中没有D，则代表没有相同的前缀后缀，next [j + 1] = 0。

> 那为何递归前缀索引 k = next[k]，就能找到长度更短的相同前缀后缀呢？
> 这又归根到next数组的含义。我们拿前缀 p0、pk-1、pk 去跟后缀pj-k、pj-1、pj匹配，如果pk 跟pj 失配，
> 下一步就是用p[next[k]] 去跟pj 继续匹配，如果p[next[k]]跟pj还是不匹配，则需要寻找长度更短的相同前缀后缀，
> 即下一步用p[next[next[k]]]去跟pj匹配。
> 此过程相当于模式串的自我匹配，所以不断的递归k = next[k]，直到要么找到长度更短的相同前缀后缀，要么没有长度更短的相同前缀后缀。如下图所示：
>
> 
>
>
> 所以，因最终在前缀ABC中没有找到D，故E的next 值为0

- 举一个能在前缀中找到字符D的例子呢？OK，咱们便来看一个能在前缀中找到字符D的例子，如下图所示：

模式串	D	A	B	C	D	A	B	D	E
最长相同前缀后缀	0	0	0	0	1	2	3	?	
next 值	-1	0	0	0	0	1	2	3	?
索引	p_0	p_1	p_{k-1}	p_k	p_{j-k}	p_{j-2}	p_{j-1}	p_j	p_{j+1}

给定模式串DABCDABDE，我们很顺利的求得字符D之前的“DABCDAB”的各个子串的最长相同前缀后缀的长度分别为0 0 0 0 1 2 3，
但当遍历到字符D，要求包括D在内的“DABCDABD”最长相同前缀后缀时，我们发现pj处的字符D跟pk处的字符C不一样，换言之，前缀DABC的最后一个字符C跟后缀DABD的最后一个字符D不相同，所以不存在长度为4的相同前缀后缀。

> 怎么办呢？既然没有长度为4的相同前缀后缀，咱们可以寻找长度短点的相同前缀后缀，最终，因在p0处发现也有个字符D，p0 = pj，所以p[j]对应的长度

> 值为1，相当于E对应的next 值为1（即字符E之前的字符串“DABCDABD”中有长度为1的相同前缀和后缀）。

递推求得next 数组,代码如下：

```
void GetNext(char* p,int next[])  {
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1, 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];  //寻找前面是否有公共前缀后缀
    }
}
```

优化后的全部代码

博客上对next数组求法继续做了一次优化，我这里就直接附上代码，不作介绍了。

```
//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,  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] = next[next[k]]
                next[j] = next[k];
```

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

```
int KmpSearch(char* s, char* p)  {
    int i = 0,  j = 0;
    int sLen = strlen(s), 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] ，相当于文本串i不变，模板串由j位置到next[j]位置，值变小了，但其实相当于右移了。
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
   return j == pLen ? i - j: -1;
}
```

扩展BM算法和Sunday算法

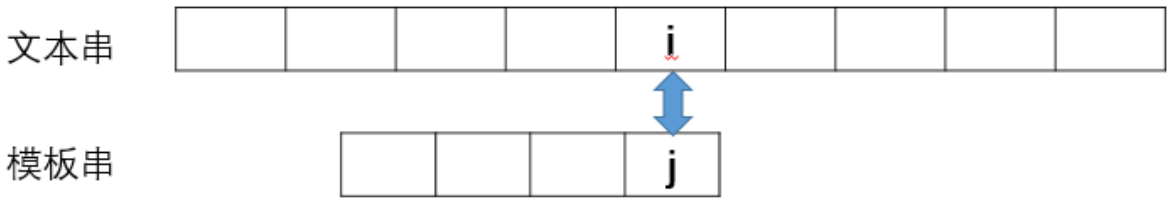
KMP算法和BM算法，这两个算法在最坏情况下均具有线性的查找时间。但实际上，KMP算法并不比最简单的c库函数strstr()快多少，而BM算法虽然通常比KMP算法快，但BM算法也还不是现有字符串查找算法中最快的算法，Sunday算法是一种比BM算法更快的查找算法。

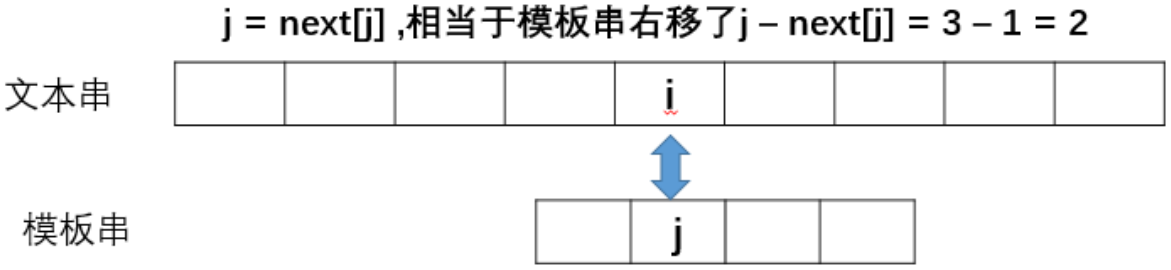
> 后续项目中用到再细看这两个算法把

个人理解

kmp算法首先求模板字符串的next数组（相对的串叫文本串，比较长），接着模板传和文本串开始比较，当出现字符不匹配的时候，文本串位置i不变，模板串位置j =

next[j]（用到之前求的next数组），看似j变小了，实际右移了j - next[k]。（比较的还是文本串i位置和模板串j位置，j指向模板串的前面）





next数组求法主要是和最长前缀后缀公共长度有关。首先得知道什么是最长前缀后缀公共长度，这幅图很清楚。

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

前缀永远从第一个字符开始，后缀永远从最后一个字符开始，找出相等的最长前缀和后缀字符串。模板穿比较的时候，就是凭借两个相等的前缀和后缀长度，进行右移，避免了重复判断。

- 在说说next数组求法
  对于P的前j+1个序列字符：
  比较模板串后缀p[j]和前缀p[k]是否相等，相等则next[j + 1] = k + 1 = next[j] + 1;

不相等时，得从k位置往前找是否还存在的公共长度的值与p[j]相等，若此时p[next[k]] == p[j]，则next[j + 1] = next[k] + 1
否则继续递归前缀索引 k = next[k]，而后重复此过程。（其实就是不断往前找跟后缀有重复的更短字符，判断有没有刚好和后缀最后一个字符相等的）