KMP

出处：http://blog.csdn.net/fightlei/article/details/52712461

模式匹配：

子串定位运算又称为模式匹配（Pattern Matching）或串匹配（String Matching）。在串匹配中，一般将主串称为目标串，将子串称为模式串。本篇博客统一用S表示目标串，T表示模式串，将从目标串S中查找模式串T的过程称为模式匹配。

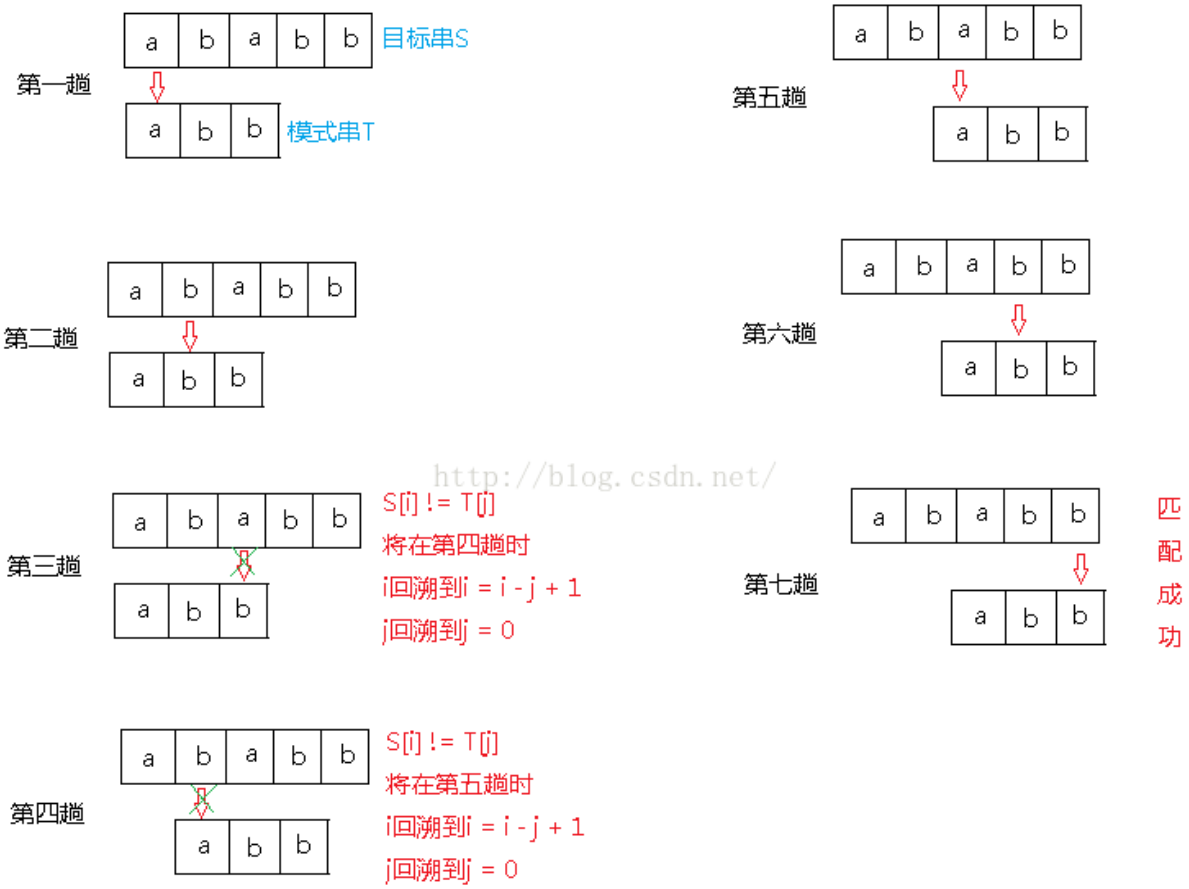
虽然我们的主角是KMP模式匹配算法，但我们还是要先从暴力匹配算法讲起，通过发现暴力匹配算法存在的问题，由此来引出KMP模式匹配算法。

朴素的模式匹配算法

【基本思想】

从目标串S的第一个字符开始和模式串T的第一个字符进行比较，如果相等则进一步比较二者的后继字符，否则从目标串的第二个字符开始再重新与模式串T的第一个字符进行比较，以此类推，直到模式串T与目标串S中的一个子串相等，称为匹配成功，返回T在S中的位置；或者S中不存在值与T相等的子串，称匹配失败，返回-1.此算法也称为BF（Brute-Force）算法。

我们先通过一个简单的例子，来了解一下BF算法是怎么回事。假设有一个目标串S为“ababb”，模式串T为“abb”。由于例子比较简单，我们可以绘制出整个的匹配过程。如下图所示：



可以看到匹配流程完全是按照上面给出的基本思想走下来的，首先从目标串S的第一个字符开始和模式串T的第一个字符进行比较（第一趟），如果相等则进一步比较二者的后继字符（第二趟），否则从目标串的第二个字符开始再重新与模式串T的第一个字符进行比较（第三趟，第四趟）。我们重点来关注一下第三趟，此时，发现S[i] != T[j]，则要从目标串S的第二个字符再重新开始，i回溯到i = i - j + 1。因为i - j表示这一趟的起始匹配位置，i - j + 1则意为从这一趟起始比较位置的下一个位置继续进行比较。同时j要回溯到0，即重新与模式串T的第一个字符进行比较。

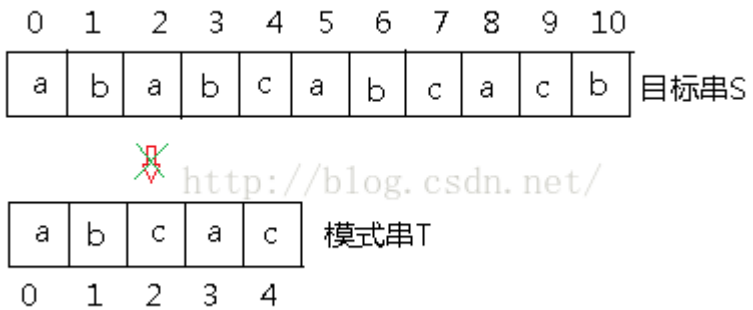
【BF算法实现】

```java
/*
 * BF匹配算法
 */
public static int violentMatching(String s, String t) {
    int i = 0;
    int j = 0;
    while (i < s.length() && j < t.length()) {
        if (s.charAt(i) == t.charAt(j)) {
            i++;
            j++;
        } else {
            //i回溯到这一趟起始匹配位置的下一个位置
            i = i - j + 1;
            j = 0;
        }
    }
    //当j==t.length()表示目标串S中的一个子串与模式串T完全匹配
    if (j == t.length()) {
        //返回这一趟起始匹配位置，即T在S中的位置
        return i - j;
    } else {
        return -1;
    }
}
```

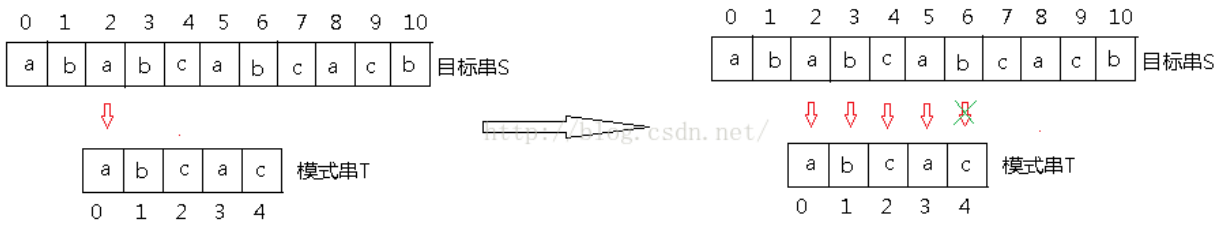
BF算法的实现比较简单，思维方式也很直接，比较容易理解。但是我们发现存在这样的问题：

第一趟比较结束后，我们可以发现信息：S[0] =T[0]，第二趟比较结束后，得到信息：S[1] = T[1]，第三趟后得到信息：S[2] != T[2]。接下来我们通过观察模式串T可以发现T[0] !=T[1]。因此可以立即得出结论T[0] != S[1]，所以根本无需进行第四趟的比较。可能由于例子比较简单，无法鲜明的体现出KMP算法的优势，下面我们举一个稍微复杂些的例子来看看：

假设有一个目标串S为“ababcabcacb”，模式串为“abcac”，当比较到到S[2]与T[2]时出现失配



如果是按照BF算法，则下一趟应从S[1]与T[0]进行比较开始。但是通过上一趟的比较我们是可以发现：S[0] = T[0]，S[1] = T[1]，S[2] != T[2]。再观察模式串T自身我们发现T[0] != T[1]，因此可以立即得出结论S[1] != T[0]，所以可以省略它们的比较，直接从S[2]与T[0]进行比较开始：



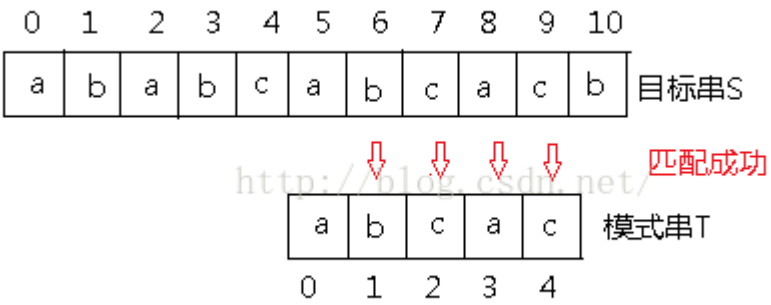
从图中可以看到，当比较到S[6]和T[4]时，再次出现失配情况。如果继续按照BF算法，显然又会多进行几次不必要的比较。那么又应该从目标串和模式串的哪两个位置开始进行比较呢？

从上图可以看出比较结束时，有如下信息：S[2] = T[0]，S[3] = T[1]，S[4] = T[2]，S[5] = T[3]，S[6] != T[4]。然后我们在观察模式串T，可以得到：

（1）T[0] != T[1]，因此T[0] != S[3]，所以可以省略它们的比较。

（2）T[0] != T[2]，因此T[0] != S[4]，省略它们的比较。

（3）T[0] = T[3]，因此T[0] = S[5]，当相等时继续比较两个串的后继字符，所以从S[6]和T[1]开始进行比较。



可以看到应用此方法，只发生了三次重新匹配，就得到了匹配成功的结论，加快了匹配的执行速度。

上面的例子只是大概描述了方法的思路，但是这种方法到底是什么，到底如何精确的进行描述，以及如何用代码实现呢？下面就来解决这些问题。

KMP模式匹配算法

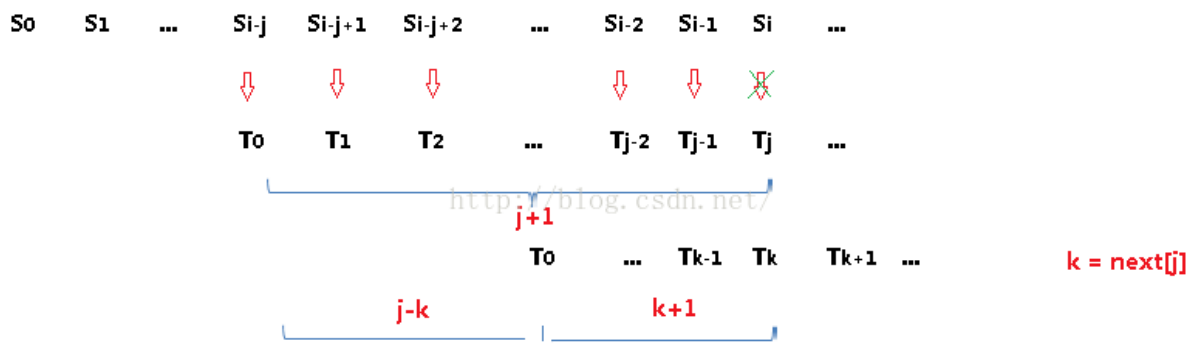
此算法是由D.E.Knuth，J.H.Morris和V.R.Pratt同时发现的，因此该算法被称为克努斯-莫里斯-普拉特操作，简称为KMP算法。

KMP算法，是不需要对目标串S进行回溯的模式匹配算法。读者可以回顾上面的例子，整个过程中完全没有对目标串S进行回溯，而只是对模式串T进行了回溯。通过前面的分析，我们发现这种匹配算法的关键在于当出现失配情况时，应能够决定将模式串T中的哪一个字符与目标串S的失配字符进行比较。所以呢，那三位前辈就通过研究发现，使用模式串T中的哪一个字符进行比较，仅仅依赖于模式串T本身，与目标串S无关。

这里就要引出KMP算法的关键所在next数组，next数组的作用就是当出现失配情况S[i] != T[j]时，next[j]就指示使用T中的以next[j]为下标的字符与S[i]进行比较（注意在KMP算法中，i是永远不会进行回溯的）。还需要说明的是当next[j] = -1时，就表示T中的任何字符都不与S[i]进行比较，下一轮比较从T[0]与S[i+1]开始进行。由此可见KMP算法在进行模式匹配之前需要先求出关于模式串T各个位置上的next函数值。即next[j]，j = 0,1,2,3,...n-1。

求解next数组

根据next数组的特性，匹配过程中一旦出现S[i] != T[j]，则用T[next[j]]与S[i]继续进行比较，这就相当于将模式串T向右滑行j - next[j]个位置，示意图如下：



理解上面这幅图是理解next数组的关键，为了绘图简单，使用k 来表示next[j]。图中，j+1表示模式串T的字符个数，当出现失配情况时，使用T[next[j]]与S[i]进行比较，即图中T[k]与S[i]进行比较。因此右边括起来的是T[0]~T[k]共k+1个字符，因此左边括起来的是j + 1 - (k + 1) = j - k个字符，即向右滑行了j-next[j]个位置。

当上图中出现失配后可以得到如下信息：

S[i-j] = T[0]，S[i-j+1] = T[1]，...，S[i-k] = T[j-k]，S[i-k+1] = T[j-k+1]，...，S[i-2] = T[j-2]，S[i-1] = T[j-1]

模式串T进行右滑后，如图中所示必须保证：

S[i-k] = T[0]，S[i-k+1] = T[1]，S[i-k+2] = T[2]，...，S[i-2] = T[k-2]，S[i-1] = T[k-1]

通过上面两个式子可得：

T[0] = T[j-k]，T[1] = T[j-k+1]，T[2] = T[j-k+2]，...，T[k-2] = T[j-2]，T[k-1] = T[j-1]

它的含义表示对于模式串T中的一个子串T[0]~T[j-1]，K的取值需要满足前K个字符构成的子序列（即T[0]~T[k-1]，称为前缀子序列）与后K个字符构成的子序列（即T[j-1]~T[j-k]，称为后缀子序列）相等。满足这个条件的K值有多个，取最大的那个值。

由此求解next数组问题，便被转化成了求解最大前缀后缀子序列问题。

再通过一个例子来说明最大前缀后缀子序列分别是什么？

对于子串“aaabcdbaaa”，满足条件的K值有1,2,3，取最大K值即3做为next[j]的函数值。此时的最大前缀子序列为“aaa”，最大后缀子序列为“aaa”。

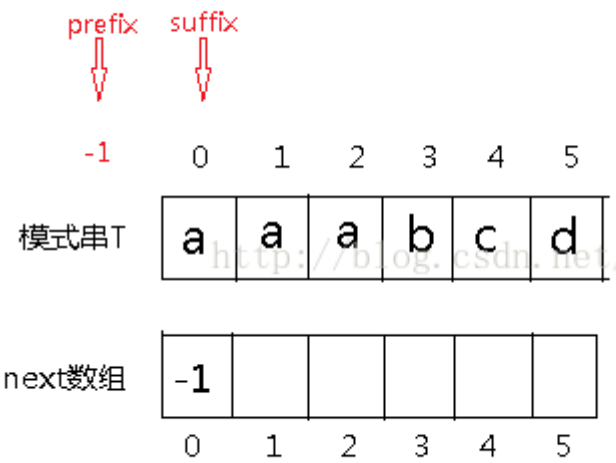
再比如子串“abcabca”，其相等的最大前缀后缀子序列即为“abca”

【求解next数组的算法实现】

```
    public static int[] getNext(String t) {
        int[] next = new int[t.length()];
        next[0] = -1;
        int suffix = 0;  // 后缀
        int prefix = -1;  // 前缀
        while (suffix < t.length() - 1) {
            //若前缀索引为-1或相等，则前缀后缀索引均+1
            if (prefix == -1 || t.charAt(prefix) == t.charAt(suffix)) {
                ++prefix;
                ++suffix;
                next[suffix] = prefix;  //1
            } else {
                prefix = next[prefix];  //2
            }
        }
        return next;
    }
```

代码其实并不复杂，整体思路是分别以T[0]~T[suffix]为子串，依次求这些子串的相等的最大前缀后缀子序列，即next[suffix]的值。

比较难理解的应该是有两处，我分别用1和2标示了出来。我们依次来看。初始化的过程如下图所示，prefix指向-1，suffix指向0，next[0] = -1。



if条件中prefix = -1成立，所以进入if语句，prefix = prefix+1，suffix = suffix+1，此时直接将next[suffix]赋值为prefix。即next[1] = 0。prefix+1到底代表的是什么？next[suffix]又代表的是什么？

next[suffix]表示的是不包括suffix即T[0]~T[suffix-1]这个子串的相等最长前缀后缀子序列的长度。意思是这个子串前面有next[suffix]个字符，与后面的next[suffix]个字符相等。

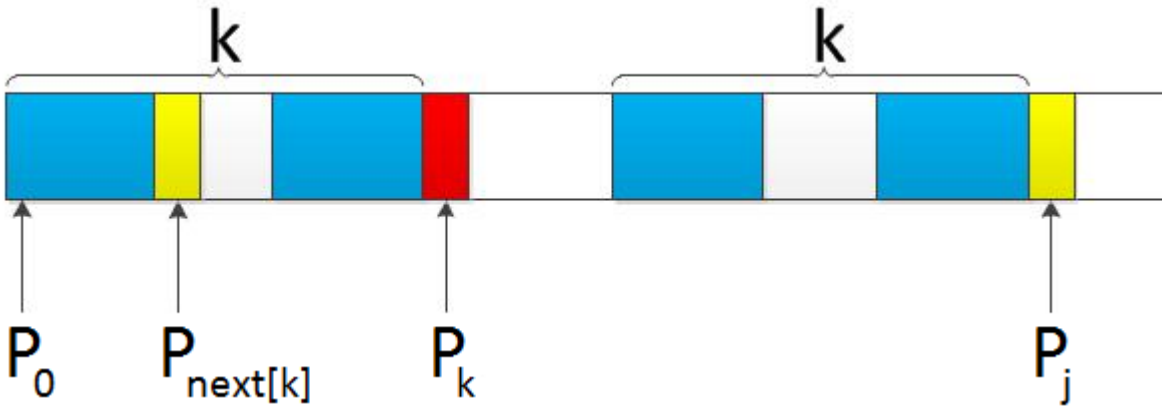
代码中suffix+1以后值为1，prefix+1以后值为0，next[1]表示的是对于子串“a”，它的相等最长前缀后缀子序列的长度，即为prefix，0。prefix一直表示的就是对于子串T[0]~T[suffix-1]前面有prefix个字符与后面prefix个字符相等，就是next[suffix]

继续往下走，满足if条件T[0] = T[1]，则suffix+1值为2，prefix+1以后值为1，next[2] = 1，表示子串“aa”，有长度为1的相等最长前缀后缀子序列“a”。

继续往下走，满足if条件T[1] = T[2]，则suffix+1值为3，prefix+1以后值为2，next[3] = 2，表示子串“aaa”，有长度为2的相等最长前缀后缀子序列“aa”。

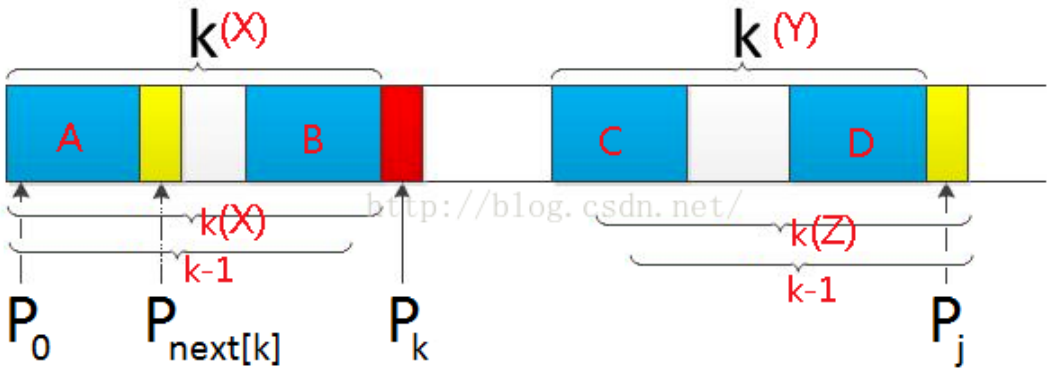
当再继续往下走时会发现T[2] != T[3]，不满足条件，则进入了else语句，prefix进行了回溯，prefix = next[prefix]，这就遇到了第二个难点，为什么要如此进行回溯呢？

借用网上的一张图来回答这个问题



这张图网上很多，但是详细描述这张图的具体含义的却很少。图中的j就对应代码中的suffix，k就对应代码中的prefix，模式串T图中用的P表示。

现在它们也遇到了这个问题，Pj] != P[k]，然后k进行了回溯，变为next[k]。既然prefix能走到k，suffix能走到j，则至少能保证对于子串P[0] ~ P[j-1]，前面有k个字符与后面k个字符相等。即图中前后方的蓝色区域。要是满足条件P[k] = P[j]则说明对于子串P[0] ~ P[j]，前面有k+1个字符与后面k+1个字符相等。但是现在不满足，则说明对于子串P[0] ~ P[j]不存在长度为k+1的相等最长前缀后缀子序列，可能存在比k+1小的最长前缀后缀子序列，可能是k，可能是k-1，k -2 , k -3 ...或者根本就没有是0。那么我们的正常思路应该是回溯到k再进行判断，不存在k个则再回溯到k-1个，以此类推，那么算法中为什么是直接回溯到next[k]呢？



为了便于描述，我将图中的不同区域使用大写字母进行标注。前面说过正常思路是不存在k+1个，就回溯到k个进行判断，现在我们来看为什么不回溯到k？当回溯到k时，需要满足的条件是X区域的字符与Z区域的字符相等，而我们已知的是X区域的字符与Y区域的字符相等，若要满足条件，则需要Y区域字符与Z区域字符相等，从图中可以看到，Y区域与Z区域的字符仅相差一位，实际上比较的是Y区域的第一个字符与第二个字符，第二个字符与第三个字符等等，所以除非是Y区域的字符全部相等，是同一个字符，否则是不可能满足条件的。然而当Y区域的字符全部相等时，则X区域的字符也全部相等，那么next[k]就等于k，所以不如直接就回溯到next[k]。

那为什么直接回溯到next[k]就一定会满足条件呢？

已知的是X区域等于Y区域，所以B区域一定等于D区域，因为B区域表示X区域的后next[k]个字符，D区域表示Y区域的后next[k]个字符。

而next[k]的含义就是对于子串P[0] ~ P[k-1]，前面有next[k]个字符与后面next[k]个字符相等，即A区域等于B区域，所以可以得到A区域一定等于D区域，因此当下次比较满足条件P[next[k]] = P[j]时，就一定有长度为next[k] + 1的相等最长前缀后缀子序列。

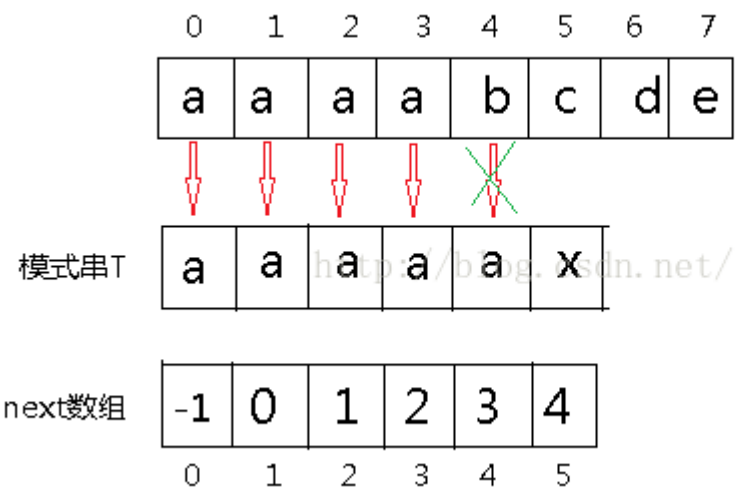
next数组的求解搞清楚了，接下来我们就可以给出KMP算法的完整实现了

【KMP模式匹配算法实现】

```
public static int KMP(String s, String t) {
    int i = 0;
    int j = 0;
    //得到next数组
    int[] next = getNext(t);
    while (i < s.length() && j < t.length()) {
        if (j == -1 || s.charAt(i) == t.charAt(j)) {
            i++;
            j++;
        } else {
            //根据next数组的指示j进行回溯，而i永远不会回溯
            j = next[j];
        }
    }
    if (j == t.length()) {
        return i - j;
    } else {
        return -1;
    }
}
```

KMP算法优化

上面给出的KMP算法还有一些小问题，比如有一个模式串“aaaaax”，目标串“aaaabcde”。通过上面给出的算法我们很容易得到模式串T的next数组，匹配过程如下图所示：



可以看到当匹配到S[4]和T[4]时出现失配情况，而根据next数组的指示，next[4]，下一步应该比较S[4]和T[3]，再次失配，再根据指示，比较S[4]和T[2]，再失配，再比较S[4]和T[2]，又失配，直到比较到S[0]和T[0]仍然失配，然后next[0] = -1，则表示T中的任何字符都不与S[4]进行比较，下一轮比较从S[5]与T[0]开始进行。对于这种特殊情况，虽然我们使用了next数组，但效率仍然是低下的。当S[4] != T[4]时，由于T[4] = T[3] = T[2] = T[1] = T[0]，所以它们都不会与S[4]相等，因此应该直接用T[0]与S[5]进行比较。

针对这种情况，只需改进next数组的求解过程即可

【next数组求解算法优化实现】

```
public static int[] getNext(String t) {
    int[] next = new int[t.length()];
    next[0] = -1;
    int suffix = 0;  // 后缀
    int prefix = -1;  // 前缀
    while (suffix < t.length() - 1) {
        //若相等或前缀索引为-1，则前缀后缀索引均+1
        if (prefix == -1 || t.charAt(prefix) == t.charAt(suffix)) {
            ++prefix;
            ++suffix;
            //改进的地方
            if (t.charAt(prefix) == t.charAt(suffix)) {
                next[suffix] = next[prefix];
            } else {
                next[suffix] = prefix;
            }
        } else {
            prefix = next[prefix];
        }
    }
    return next;
}
```

改进的地方在于，如果T[suffix] != T[prefix]，则仍然遵从之前的处理，next[suffix] = prefix。

若T[suffix] = T[prefix]则可能会出现上面例子所说的特殊情况，使next[suffix] = next[prefix]。其实这就是一个回溯过程，原本应该是next[suffix] = prefix，这里由prefix回溯到了next[prefix]。可以这样理解，当再T[suffix]位置出现失配时，本来按照next数组的指示应采用T[next[suffix]] = T[prefix]进行下一轮比较，但是我们已经得知T[suffix] = T[prefix]所以一定会再次出现失配情况，所以我们下一轮直接使用T[next[prefix]]进行比较。