到右计算回文半径时，关注回文半径最右即将到达的位置（pR），一旦发现已经到达最后(pR==charArr.length)，说明必须包含最后一个字符的最长回文半径已经找到，直接退出检查过程，返回该添加的字符串即可。具体过程参看如下代码中的 shortestEnd 方法。

```
public String shortestEnd(String str) {
        if (str == null || str.length() == 0) {
                return null;
        }
        char[] charArr = manacherString(str);
        int[] pArr = new int[charArr.length];
        int index = -1;
        int pR = -1;
        int maxContainsEnd = -1;
        for (int i = 0; i != charArr.length; i++) {
                pArr[i] = pR > i ? Math.min(pArr[2 * index - i], pR - i) : 1;
                while (i + pArr[i] < charArr.length && i - pArr[i] > -1) {
                        if (charArr[i + pArr[i]] == charArr[i - pArr[i]])
                                pArr[i]++;
                        else {
                                break;
                        }
                }
                if (i + pArr[i] > pR) {
                        pR = i + pArr[i];
                        index = i;
                }
                if (pR == charArr.length) {
                        maxContainsEnd = pArr[i];
                        break;
                }
        }
        char[] res = new char[str.length() - maxContainsEnd + 1];
        for (int i = 0; i < res.length; i++) {
                res[res.length - 1 - i] = charArr[i * 2 + 1];
        }
        return String.valueOf(res);
}
```

KMP 算法

【题目】

给定两个字符串 str 和 match，长度分别为 N 和 M。实现一个算法，如果字符串 str 中含有子串 match，则返回 match 在 str 中的开始位置，不含有则返回-1。

【举例】

str="acbc"，match="bc"，返回 2。

str="acbc"，match="bcc"，返回-1。

【要求】

如果 match 的长度大于 str 的长度（$M>N$），str 必然不会含有 match，可直接返回-1。但如果 $N \geq M$，要求算法复杂度为 $O(N)$。

【难度】

将 ★★★★

【解答】

本文是想重点介绍一下 KMP 算法，该算法是由 Donald Knuth、Vaughan Pratt 和 James H. Morris 于 1977 年联合发明的。在介绍 KMP 算法之前，我们先来看普通解法怎么做。

最普通的解法是从左到右遍历 str 的每一个字符，然后看如果以当前字符作为第一个字符出发是否匹配出 match。比如 str="aaaaaaaaaaaaaaaaab"，match="aaaab"。从 str[0]出发，开始匹配，匹配到 str[4]=='a'时发现和 match[4]=='b'不一样，所以匹配失败，说明从 str[0]出发是不行的。从 str[1]出发，开始匹配，匹配到 str[5]=='a'时发现和 match[4]=='b'不一样，所以匹配失败，说明从 str[1]出发是不行的。从 str[2..12]出发，都会一直失败。从 str[13]出发，开始匹配，匹配到 str[17]=='b'时发现和 match[4]=='b'一样，match 已经全部匹配完，说明匹配成功，返回 13。普通解法的时间复杂度较高，从每个字符出发时，匹配的代价都可能是 $O(M)$，那么一共有 N 个字符，所以整体的时间复杂度为 $O(N \times M)$。普通解法的时间复杂度这么高，是因为每次遍历到一个字符时，检查工作相当于从无开始，之前的遍历检查不能优化当前的遍历检查。

下面介绍 KMP 算法是如何快速解决字符串匹配问题的。

1．首先生成 match 字符串的 nextArr 数组，这个数组的长度与 match 字符串的长度一样，nextArr[i]的含义是在 match[i]之前的字符串 match[0..i-1]中，必须以 match[i-1]结尾的后缀子串（不能包含 match[0]）与必须以 match[0]开头的前缀子串（不能包含 match[i-1]）最大匹配长度是多少。这个长度就是 nextArr[i]的值。比如，match="aaaab"字符串，nextArr[4]的值该是多少呢？match[4]=='b'，所以它之前的字符串为"aaaa"，根据定义这个字符串的后缀子串和前缀子串最大匹配为"aaa"。也就是当后缀子串等于 match[1..3]=="aaa"，前缀子串等于 match[0..2]=="aaa"时，这时前缀和后缀不仅相等，而且是所有前缀和后缀的可能性中

最大的匹配。所以 nextArr[4]的值等于 3。再如，match="abc1abc1"字符串，nextArr[7]的值该是多少呢？match[7]=='1'，所以它之前的字符串为"abc1abc"，根据定义这个字符串的后缀子串和前缀子串最大匹配为"abc"。也就是当后缀子串等于 match[4..6]=="abc"，前缀子串等于 match[0..2]=="abc"时，这时前缀和后缀不仅相等，而且是所有前缀和后缀的可能性中最大的匹配。所以 nextArr[7]的值等于 3。关于如何快速得到 nextArr 数组的问题，我们在把 KMP 算法的大概过程介绍完毕之后再详细说明，接下来先看如果有了 match 的 nextArr 数组，如何加速进行 str 和 match 的匹配过程。

2．假设从 str[i]字符出发时，匹配到 j 位置的字符发现与 match 中的字符不一致。也就是说，str[i]与 match[0]一样，并且从这个位置开始一直可以匹配，即 str[i..j-1]与 match[0..j-i-1]一样，直到发现 str[j]!=match[j-i]，匹配停止。如图 9-21 所示。

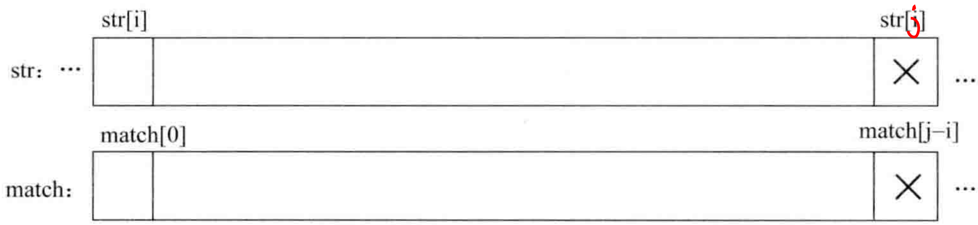

图 9-21

因为现在已经有了 match 字符串的 nextArr 数组，nextArr[j-i]的值表示 match[0..j-i-1]这一段字符串前缀与后缀的最长匹配。假设前缀是图 9-22 中的 a 区域这一段，后缀是图 9-22 中的 b 区域这一段，再假设 a 区域的下一个字符为 match[k]，如图 9-22 所示。

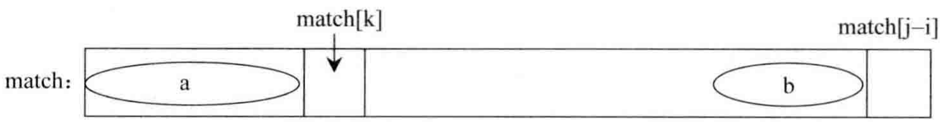

图 9-22

那么下一次的匹配检查不再像普通解法那样退回到 str[i+1]重新开始与 match[0]的匹配过程，而是直接让 str[j]与 match[k]进行匹配检查，如图 9-23 所示。

在图 9-23 中，在 str 中要匹配的位置仍是 j，而不进行退回。对 match 来说，相当于向右滑动，让 match[k]滑动到与 str[j]同一个位置上，然后进行后续的匹配检查。普通解法 str 要退回到 i+1 位置，然后让 str[i+1]与 match[0]进行匹配，而我们的解法在匹配的过程中一直进行这样的滑动匹配的过程，直到在 str 的某一个位置把 match 完全匹配完，就说明 str 中有 match。如果 match 滑到最后也没匹配出来，就说明 str 中没有 match。那么为什么这

样做是正确的呢？如图 9-24 所示。

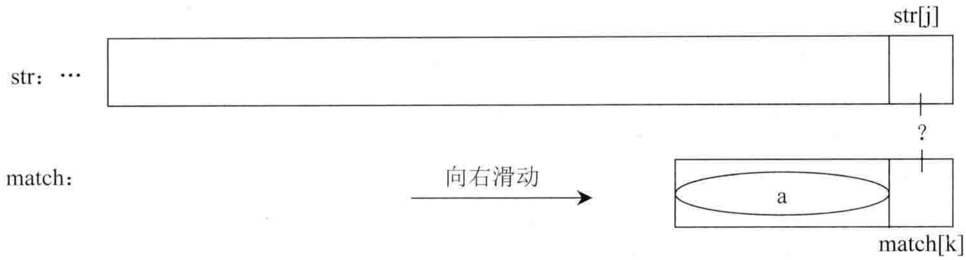

图 9-23

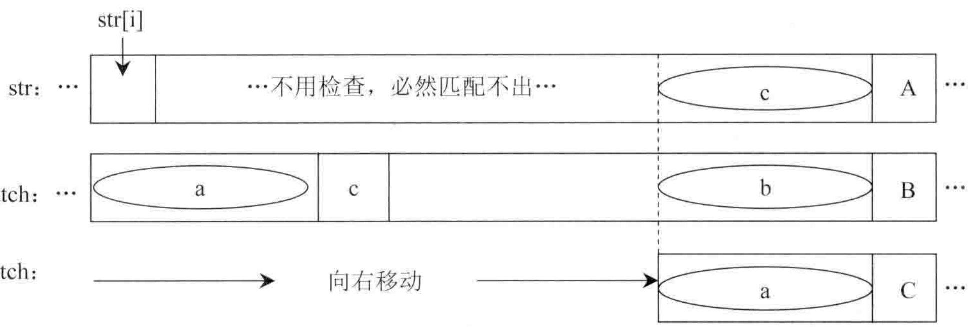

图 9-24

在图 9-24 中，匹配到 A 字符和 B 字符才发生的不匹配，所以 c 区域等于 b 区域，b 区域又与 a 区域相等(因为 nextArr 的含义如此)，所以 c 区域和 a 区域是不需要检查的，必然会相等。所以直接把字符 C 滑到字符 A 的位置开始检查即可。其实这个过程相当于是从 str 的 c 区域中第一个字符重新开始的匹配过程(c 区域的第一个字符和 match[0]匹配，并往右的过程)，只不过因为 c 区域与 a 区域一定相等，所以省去了这个区域的匹配检查而已，直接从字符 A 和字符 C 往后继续匹配检查。读者看到这里肯定会问，为什么开始的字符从 str[i]直接跳到 c 区域的第一个字符呢？中间的这一段为什么是“不用检查”的区域呢？因为在这个区域中，从任何一个字符出发都肯定匹配不出 match，下面还是图解来解释这一点。如图 9-25 所示。

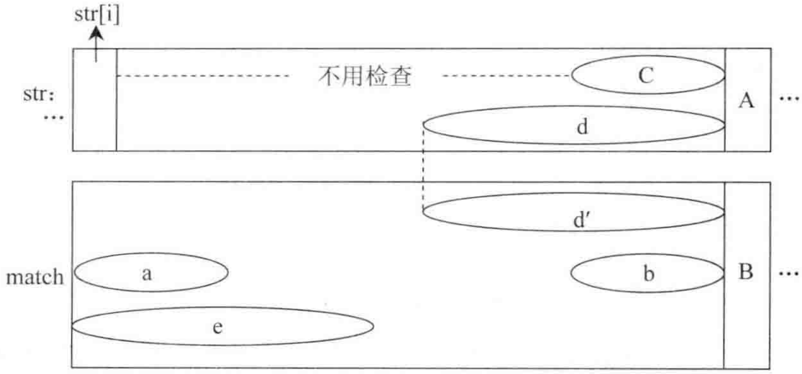

图 9-25

在图 9-25 中，假设 d 区域开始的字符是“不用检查”区域的其中一个位置，如果从这个位置开始能够匹配出 match，那么毫无疑问，起码整个 d 区域应该和从 match[0]开始的 e 区域匹配，即 d 区域与 e 区域长度一样，且两个区域的字符都相等。同时我们注意到，d 区域比 c 区域大，e 区域比 a 区域大。如果这种情况发生了，假设 d 区域对应到 match 字符串中是 d'区域，也就是字符 B 之前的字符串的后缀，而 e 区域本身就是 match 的前缀，所以对 match 来说，相当于找到了 B 这个字符之前的字符串(match[0..j-i-1])的一个更大的前缀与后缀匹配，一个比 a 区域和 b 区域更大的前缀后缀匹配，e 区域和 d'区域。这与 nextArr[j-i]的值是自相矛盾的，因为 nextArr[j-i]的值代表的含义就是 match[0..j-i-1]字符串上最大的前缀与后缀匹配长度。所以如果 match 字符串的 nextArr 数组计算正确，这种情况绝不会发生。也就是说，根本不会有更大的 d'区域和 e 区域，所以 d 区域与 e 区域也必然不会相等。

匹配过程分析完毕，我们知道，str 中匹配的位置是不退回的，match 则一直向右滑动，如果在 str 中的某个位置完全匹配出 match，整个过程停止。否则 match 滑到 str 的最右侧过程也停止，所以滑动的长度最大为 N，所以时间复杂度为 $O(N)$。匹配的全部过程参看如下代码中的 getIndexOf 方法。

```
public int getIndexOf(String s, String m) {
    if (s == null || m == null || m.length() < 1 || s.length() < m.length()) {
        return -1;
    }
    char[] ss = s.toCharArray();
    char[] ms = m.toCharArray();
    int si = 0;
    int mi = 0;
    int[] next = getNextArray(ms);
    while (si < ss.length && mi < ms.length) {
        if (ss[si] == ms[mi]) {
```

```
                si++;
                mi++;
            } else if (next[mi] == -1) {
                si++;
            } else {
                mi = next[mi];
            }
        }
        return mi == ms.length ? si - mi : -1;
    }
```

最后需要解释如何快速得到 match 字符串的 nextArr 数组，并且要证明得到 nextArr 数组的时间复杂度为 $O(M)$。对 match[0]来说，在它之前没有字符，所以 nextArr[0]规定为-1。对 match[1]来说，在它之前有 match[0]，但 nextArr 数组的定义要求任何子串的后缀不能包括第一个字符(match[0])，所以 match[1]之前的字符串只有长度为 0 的后缀字符串，所以 nextArr[1]为 0。之后对 match[i](i>1)来说，求解过程如下：

1．因为是左到右依次求解 nextArr，所以在求解 nextArr[i]时，nextArr[0..i-1]的值都已经求出。假设 match[i]字符为图 9-26 中的 A 字符，match[i-1]为图 9-26 中的 B 字符，如图 9-26 所示。

通过 nextArr[i-1]的值可以知道 B 字符前的字符串的最长前缀与后缀匹配区域，图 9-26 中的 l 区域为最长匹配的前缀子串，k 区域为最长匹配的后缀子串，图 9-26 中字符 C 为 l 区域之后的字符。然后看字符 C 与字符 B 是否相等。

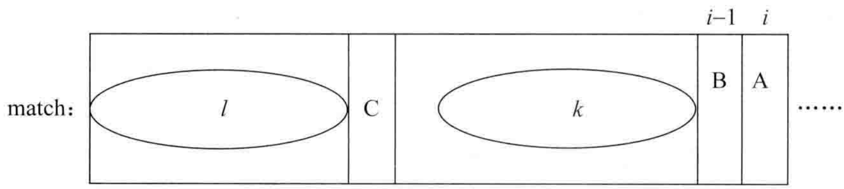

图 9-26

2．如果字符 C 与字符 B 相等，那么 A 字符之前的字符串的最长前缀与后缀匹配区域就可以确定，前缀子串为 l 区域+C 字符，后缀子串为 k 区域+B 字符，即 nextArr[i]=nextArr[i-1]+1。

3．如果字符 C 与字符 B 不相等，就看字符 C 之前的前缀和后缀匹配情况，假设字符 C 是第 cn 个字符（match[cn]），那么 nextArr[cn]就是其最长前缀和后缀匹配长度，如图 9-27 所示。

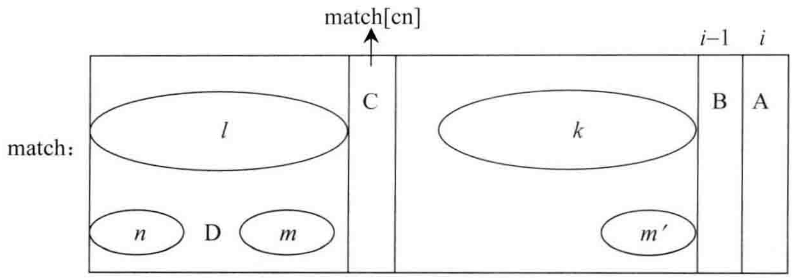

图 9-27

在图 9-27 中，*m* 区域和 *n* 区域分别是字符 C 之前的字符串的最长匹配的后缀与前缀区域，这是通过 nextArr[cn]的值确定的，当然两个区域是相等的，*m'*区域为 *k* 区域最右的区域且长度与 *m* 区域一样，因为 *k* 区域和 *l* 区域是相等的，所以 *m* 区域和 *m'*区域也相等，字符 D 为 *n* 区域之后的一个字符，接下来比较字符 D 是否与字符 B 相等。

1）如果相等，A 字符之前的字符串的最长前缀与后缀匹配区域就可以确定，前缀子串为 *n* 区域+D 字符，后缀子串为 m'区域+B 字符，则令 nextArr[i]=nextArr[cn]+1。

2）如果不等，继续往前跳到字符 D，之后的过程与跳到字符 C 类似，一直进行这样的跳过程，跳的每一步都会有一个新的字符和 B 比较（就像 C 字符和 D 字符一样），只要有相等的情况，nextArr[i]的值就能确定。

4．如果向前跳到最左位置（即 match[0]的位置），此时 nextArr[0]==-1，说明字符 A 之前的字符串不存在前缀和后缀匹配的情况，则令 nextArr[i]=0。用这种不断向前跳的方式可以算出正确的 nextArr[i]值的原因还是因为每跳到一个位置 cn，nextArr[cn]的意义就表示它之前字符串的最大匹配长度。求解 nextArr 数组的具体过程请参看如下代码中的 getNextArray 方法，先看代码，然后分析这个过程的时间复杂度为什么为 $O(M)$。

```
public int[] getNextArray(char[] ms) {
        if (ms.length == 1) {
                return new int[] { -1 };
        }
        int[] next = new int[ms.length];
        next[0] = -1;
        next[1] = 0;
        int pos = 2;
        int cn = 0;
        while (pos < next.length) {
                if (ms[pos - 1] == ms[cn]) {
                        next[pos++] = ++cn;
```

```
            } else if (cn > 0) {
                cn = next[cn];
            } else {
                next[pos++] = 0;
            }
        }
        return next;
    }
```

getNextArray 方法中的 while 循环就是求解 nextArr 数组的过程，现在证明这个循环发生的次数不会超过 $2M$ 这个数量。先来看两个量，一个为 pos 量，一个为（pos-cn）的量。对 pos 量来说，从 2 开始又必然不会大于 match 的长度，即 pos<M。对（pos-cn）量来说，pos 最大为 M-1，cn 最小为 0，所以(pos-cn)<=M。

循环的第一个逻辑分支会让 pos 的值增加，(pos-cn)的值不变。循环的第二个逻辑分支为 cn 向左跳的过程，所以会让 cn 减小，pos 值在这个分支中不变，所以（pos-cn）的值会增加。循环的第三个逻辑分支会让 pos 的值增加，(pos-cn)的值也增加。如下表所示：

	Pos	pos-cn
循环的第一个逻辑分支	增加	不变
循环的第二个逻辑分支	不变	增加
循环的第三个逻辑分支	增加	增加

因为 pos+(pos-cn)<$2M$，又有上表的关系，所以循环发生的总体次数小于 pos 量和（pos-cn）量的增加次数，也必然小于 $2M$，证明完毕。

所以整个 KMP 算法的复杂度为 $O(M)$(求解 nextArr 数组的过程)+$O(N)$（匹配的过程），因为有 $N \geqslant M$，所以时间复杂度为 $O(N)$。

丢棋子问题

【题目】

一座大楼有 0~N 层，地面算作第 0 层，最高的一层为第 N 层。已知棋子从第 0 层掉落肯定不会摔碎，从第 i 层掉落可能会摔碎，也可能不会摔碎($1 \leqslant i \leqslant N$)。给定整数 N 作为楼层数，再给定整数 K 作为棋子数，返回如果想找到棋子不会摔碎的最高层数，即使在最差的情况下扔的最少次数。一次只能扔一个棋子。