不同模式匹配算法详解

1.概述

单模式匹配是处理字符串的经典问题，指在给定字符串中寻找是否含有某一给定的字串。比较形象的是CPP中的 `strStr()` 函数，Java的String类下的 `indexOf()` 函数都实现了这个功能，本文讨论几种实现单模式匹配的方法，包括暴力匹配方法、KMP方法、以及Rabin-Karp方法（虽然Rabin-Karp方法在单模式匹配中性能一般，单其多模式匹配效率较高，且采取非直接比较的方法也值得借鉴）。

算法	预处理时间	匹配时间
暴力匹配法		O(mn)
KMP	O(m)	O(n)
Rabin-Karp	O(m)	O(mn)

2.暴力匹配

模式匹配类的问题做法都是类似使用一个匹配的滑动窗口，失配时改变移动匹配窗口，具体的暴力的做法是，两个指针分别指向长串的开始、短串的开始，依次比较字符是否相等，当不相等时，指向短串的指针移动，当短串指针已经指向末尾时，完成匹配返回结果。

以leetcode28. 实现 strStr()为例给出实现代码（下同）

```
class Solution {
    public int strStr(String haystack, String needle) {
        int m = haystack.length(), n = needle.length();
        if (needle.length() == 0) return 0;
        for (int i = 0; i <= m - n; i++) {
            for (int j = 0; j < n; j++) {
                if (haystack.charAt(i + j) != needle.charAt(j))
                    break;
                if (j == n - 1)
                    return i;
            }
        }
        return -1;
    }
}
```

值得注意的是，Java中的 `indexOf()` 方法即采用了暴力匹配方法，尽管其算法复杂度比起下面要谈到的KMP方法要高上许多。

一个可能的解释是，日常使用此方法过程中串的长度都比较短，而KMP方法预处理要生成next数组浪费时间。而一般规模较大的字符串可以由开发人员自行决定使用哪种匹配方法。

```
public static int indexOf(byte[] value, int valueCount, byte[] str, int strCount, int formIndex)
    byte first = str[0];
    int max = valueCount - strCount;

    for(int i = formIndex; i <= max; ++i){
        if(value[i] != first){
            do{
                ++i
            }while(i <= max && value[i] != first);
        }

        if(i < max){
            int j = i + 1;
            int end = j + strCount -1;

            for(int k = 1; j < end && value[j] == str[k]; ++k){
                ++j;
            }

            if(j == end){
                return i;
            }
        }
    }

    return -1;
}
```

3.KMP算法

这个算法由高德纳和沃恩·普拉特在1974年构思，同年詹姆斯·H·莫里斯也独立地设计出该算法，最终三人于1977年联合发表。

大体想法是，在暴力匹配的前提下，每次失配时，不再从待匹配串开头开始重新匹配，而是充分利用已经匹配到的部分，具体的就是使用一个部分匹配表（即在程序中经常讲的next数组），利用这一特性以避免重新检查先前匹配的字符。

比如对于待匹配串 `abcabce` 当我匹配到末尾最后一个e字母时，发现失配，一般的做法是，对于长串指针往后移动一位，然后从待匹配串开始重新匹配，但事实上，我们发现对于待匹配串失配位置以前的字

符串 abcabc 来讲，存在着一个长度为3的相同的字串 abc ，我们可以把第一个叫做前缀，第二个叫做后缀，所以对于当在后缀下一个字符失配时，我们只需要回溯到前缀的下一个字符继续匹配即可，对于此串即待匹配串移动到第四个字符（数组下标为3）开始匹配。

所以对于KMP算法，核心就是构建待匹配串的部分匹配表。其作用是当模式串第 i 个位置失配，我不必从模式串开始再重新匹配，而是移动到前 i 个字符的某个位置，具体这个位置是前 i 个字符的最长公共前后缀的长度。

依旧以 abcabce 为例，假如匹配到第 i = 5 也就是第六个字母（第二个 c ）时，失配，那么我只需要退回到 i = 2 开始匹配即可，因为匹配到第六个字母时，我们已经确定 abcab 匹配成功，很明显发现 abcab 中出现了两次 ab 且分别是前后缀，那么此时只需要从 i = 2 接着匹配即可。所以计算部分匹配表本质上就是对模式串本身做了多次匹配，或者可以理解为模式串构建了一个失配的自动机。

所以对于 abcabce 很容易计算出部分失配表，特别的 i = 0 时令 next[0] = -1 。

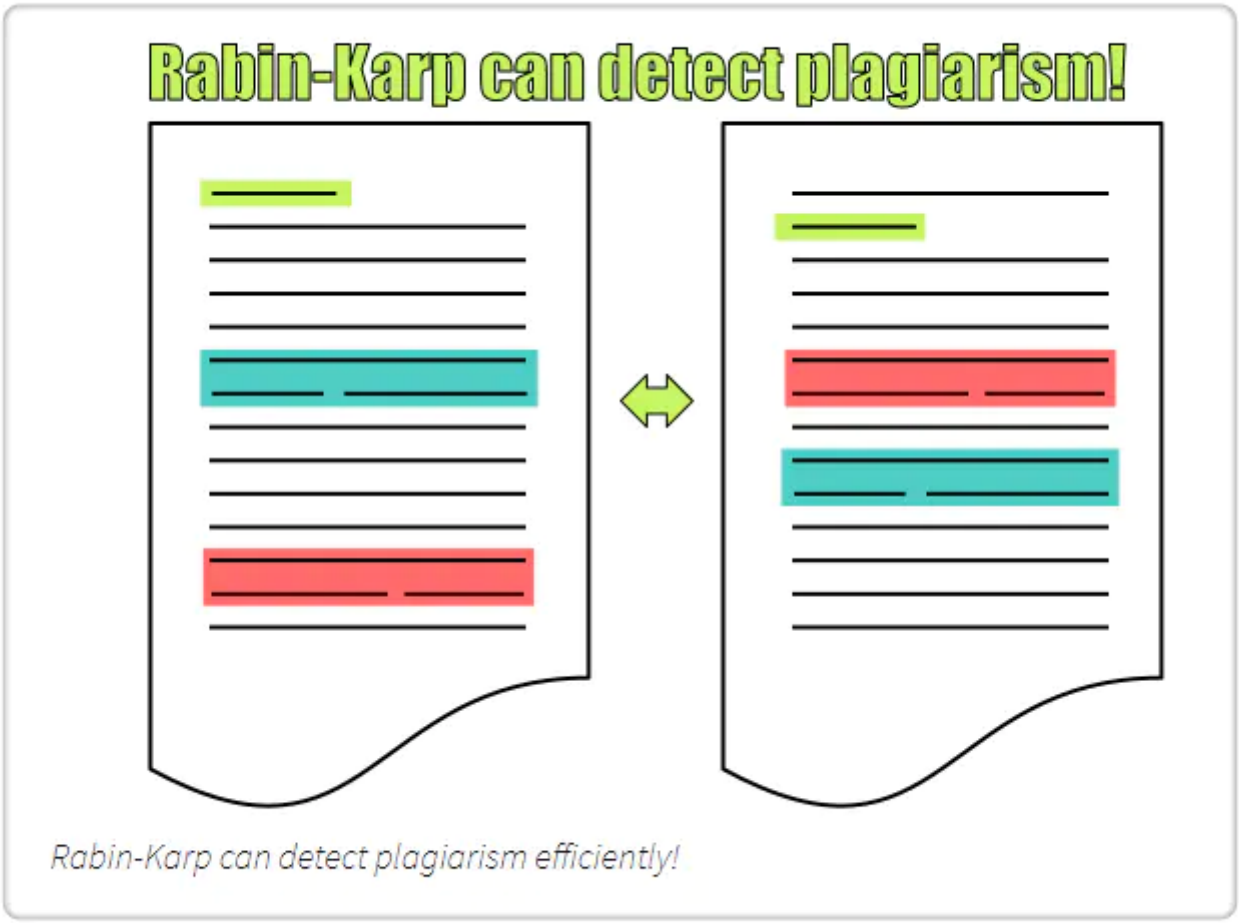
i	0	1	2	3	4	5	6
模式串	a	b	c	a	b	c	e
next[i]	-1	0	0	0	1	2	3

给出算法Java实现

```
class Solution {
    public int strStr(String haystack, String needle) {
        int i = 0, j = 0;
        int sLen = haystack.length();
        int pLen = needle.length();
        if (pLen == 0) {
            return 0;
        }

        int[] next = getNext(needle);
        while (i < sLen && j < pLen) {
            if (j == -1 || haystack.charAt(i) == needle.charAt(j)) {
                i++;
                j++;
            } else {
                j = next[j];
            }
        }
        return j == pLen ? (i - j) : -1;

    }

    public int[] getNext(String p) {
        int pLen = p.length();
        int[] next = new int[pLen];
        int k = -1;
        int j = 0;
        next[0] = -1;
        while (j < pLen -1) {
            if (k == -1 || p.charAt(j) == p.charAt(k)) {
                k++;
                j++;
                next[j] = k;
            } else {
                k = next[k];
            }
        }

        return next;
    }
}
```

4.Rabin-Karp算法

Rabin–Karp算法由 Richard M. Karp 和 Michael O. Rabin 在 1987 年发表，用来解决模式匹配问题，在多模式匹配中其效率很高，常见的应用就是论文查重。



Rabin–Karp算法采用了计算字符串hash值是否相等的方法来比较字符串是否相等，当然hash算法肯定会出现冲突的可能，所以对于计算出hash相等后还需用朴素方法对比是否字符串真的相等。

但是即使计算哈希，也需要每次都计算一个长度为模式串的哈希值，真正巧妙的地方在于，RK算法采取了滚动哈希的方法，我们假设需要匹配的字符只有26个小写字母来展开讨论。

我们采取常见的多项式哈希算法来计算，底数取一个经验值31。(JDK对于String的hashCode()方法也是如此)

假设主串为 `abcdefg` ， 模式串为 `bcde` ， 首先计算模式串的hash值，基于上述假设的前提下，为了简化，我们将字母进一步做一个映射转换成整型（统一减去'a'），那么只需要计算[1,2,3,4]的哈希值即可，得到

维护一个大小为模式串长度的滑动窗口，开始从主串开头计算窗口内的hash值，比如最开始窗口内字符串为 `abcd` ， 此时有

然后此时发现　与模式串哈希值并不相等，则将窗口往后移动一个单位，此时窗口内的字符串是 `bcde` ，我们计算它的hash值

但此时显而易见的是，　可以由　计算得来，具体的

所以此时我们能够由前一个窗口的哈希值以　　的时间复杂度计算出下一个窗口的哈希值，以方便比较。

当然显然字符串过长时会存储hash值的变量会溢出，所以需要每次累加时进行一次取模运算，具体的可以选取一个大素数，素数的选择可以参考这里。

下面给出java实现

```java
class Solution {
   public static int strStr(String haystack, String needle) {
        int sLen = haystack.length(), pLen = needle.length();
        if (pLen == 0) return 0;
        if (sLen == 0) return -1;


        int MOD = 997;
        int power = 1;
        for (int i = 0; i < pLen; i++) {
           power = (power * 31) % MOD;
        }
        int hash = 0;
        for (int i = 0; i < pLen; i++) {
            hash = (hash * 31 + (needle.charAt(i) -'a')) % MOD;
        }

        int h = 0;
        for (int i = 0; i < sLen; i++) {
            h = (h * 31 + (haystack.charAt(i) - 'a')) % MOD;
            if (i < pLen - 1) {
                continue;
            }

            if (i >= pLen) {
                h = (h - (haystack.charAt(i - pLen)-'a') * power) % MOD;

                if (h < 0) {
                    h += MOD;
                }
            }

            if (hash == h) {
                int start = i - pLen + 1;
                boolean equal = true;
                for(int j = start, k = 0; j <= i; j++,k++) {
                    if (haystack.charAt(j) != needle.charAt(k))
                        equal = false;
                }

                if (equal) return start;

            }
        }

        return -1;

    }
}
```