KMP算法

1.KMP的由来

因为是由这三位学者发明的：Knuth，Morris和Pratt，所以取了三位学者名字的首字母。所以叫做KMP

2.KMP的作用

用途：**字符串匹配**——**在文本串中查找模拟串，**如果存在，返回这个子串的起始索引，否则返回 -1。

作用：

1. 当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息**避免从头再去做匹配**。
2. 时间复杂度 **O(N)**，空间复杂度 **O(M)**。// N为文本串的长度，M为模拟串的长度

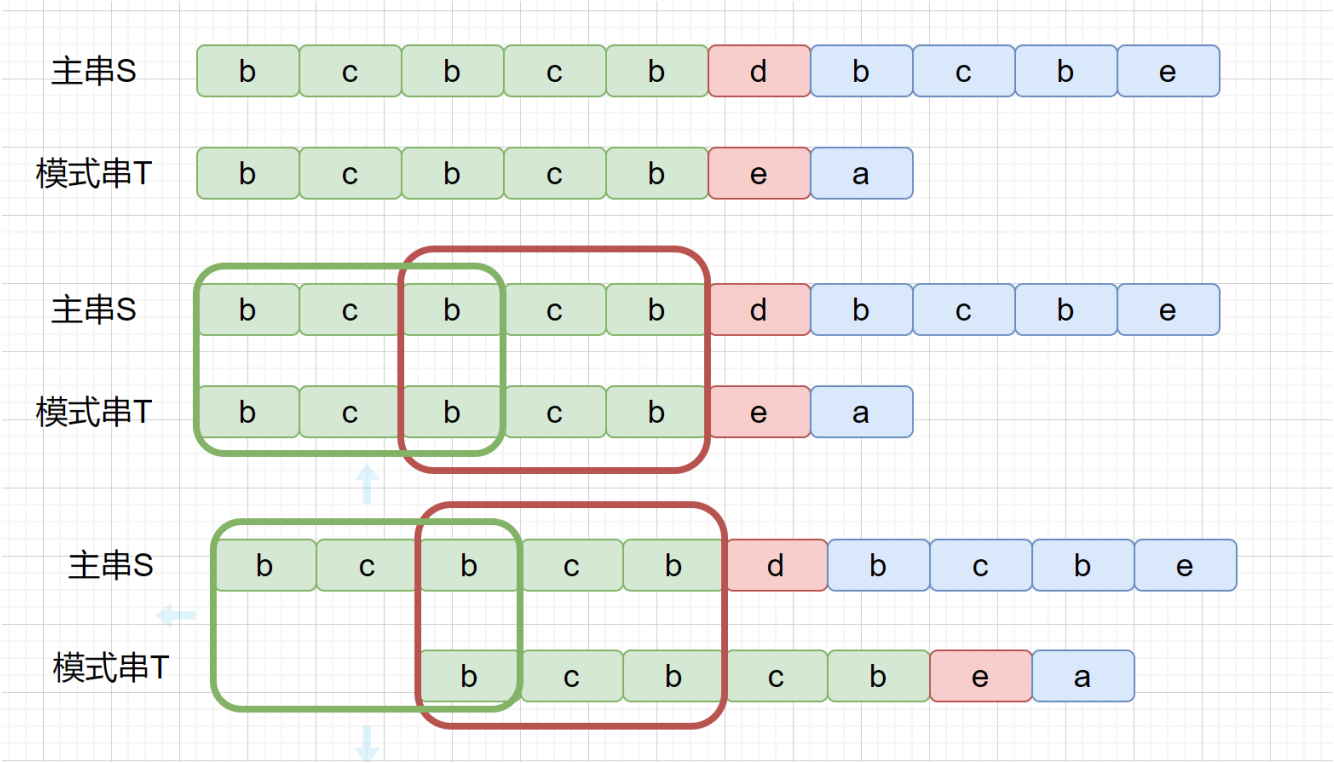
3.前缀表

3-1.定义：

> **前缀**：是指**不包含最后一个字符**的所有**以第一个字符开头的连续子串**。
>
> **后缀**：是指**不包含第一个字符**的所有**以最后一个字符结尾的连续子串**。

我们检查「前缀」和「后缀」的目的其实是「**为了确定匹配串中的下一段开始匹配的位置**」。

所以**前缀表是用来回退的，它记录了模式串与主串(文本串)不匹配的时候，判断模式串应该从哪里开始重新匹配**



3-2.作用：

char:	a	b	a	b	a	b	c	a
index:	0	1	2	3	4	5	6	7
pmt:	0	0	1	2	3	4	0	1
next:	-1	0	0	1	2	3	4	0

前缀表（pmt）中存储的是：是**字符串的前缀集合与后缀集合的交集中最长元素的长度**

index=4（a）时的pmt：a&a / ba&ba / **aba&aba**

index=5（b）时的pmt：ab&ab / b&b / bab&bab / **abab&abab**

3-3.前缀表的计算过程

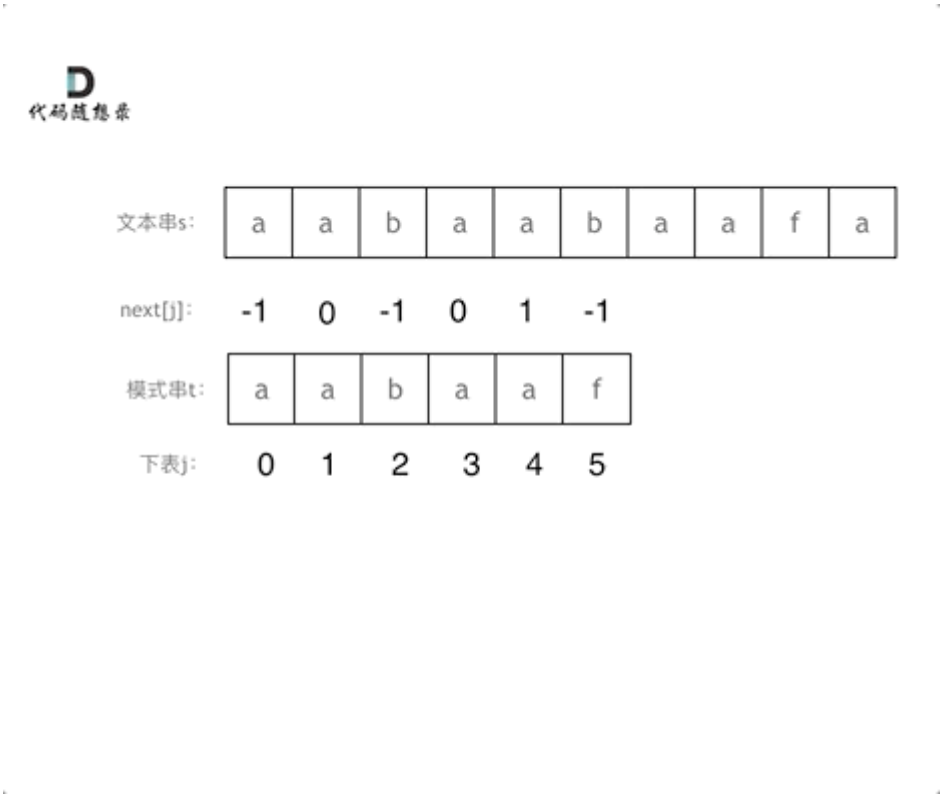
B站上的一个视频，3分钟左右，方便快速理解。

https://www.bilibili.com/video/BV12J411m74v?from=search&seid=18387765513244869997&spm_id_from=333.337.0.0

3-4.next的数组

> next数组即可以就是前缀表，也可以是前缀表统一减一（右移一位，初始位置为-1）。

这是我之前最头痛的一部分，因为我看见网上的有些人用-1的，用0的，还有用2的，后来看见Carl哥说这并没有什么区别，才发现浪费了大把的时间，呜呜呜。



个人建议不用-1开始的方式，用0开始的方式，因为更便于理解。

其实next里面存储的相当于回退的下标，图中的例子，用0的方式的话，回退的下标就是2，就是b的值。

4.代码

4-1.暴力破解：与indexOf()速度相同

```
class Solution {
```

```
    public int strStr(String haystack, String needle) {

        int M = needle.length();
        int N = haystack.length();
        if(M==0)return 0;//模拟串为空字符串
        if(M>N)return -1;//如果模拟串的长度>文本串的长度，那么文本串中就不存在模拟串

        char[] txt=haystack.toCharArray();
        char[] pat=needle.toCharArray();

        for (int i = 0; i <= N - M; i++) {// N-M的含义表示模拟串的尾部不能超出文本串的尾部
            int j;
            for (j = 0; j < M; j++) {
                //i表示文本串的指针，j表示模拟串的指针，i+j其实表示i指针向后移动
                if (pat[j] != txt[i+j])
                    break;
            }
            // pat 全都匹配了
            if (j == M) return i;
        }
        // txt 中不存在 pat 子串
        return -1;
        }
}
```

4-2.KMP算法：

```
class Solution {
    public int strStr(String haystack, String needle) {
        // KMP算法：如果已经匹配的字符串包含相同的前缀和后缀，遇到下一个不匹配的位置时，指向needle的指针跳转到前缀的后一个位置，还是不匹配的话，再往前跳转后继续比较；
        int n=haystack.length();
        int m=needle.length();
        if(m==0)return 0;
        if(m>n)return -1;

        char[] pat=needle.toCharArray();
        char[] txt=haystack.toCharArray();

        //构造一个next数组来记录needle指针跳转的位置
        int[] next=new int[m];
        next[0]=0;//第一个默认没有前后缀所以为0,也可以不写，因为默认为0;
```

```
        //i表示next数组下标，同时表示后缀的最后一个下标，j表示前缀起始。
        for(int i=1,j=0;i<m;i++){
            // 一直和前一位置的值比较，直到遇到相等的字符或者j=0；j通过next[j-1]来回退
            while(j>0&&pat[i]!=pat[j])j=next[j-1];
            if(pat[i]==pat[j])j++;//如果相等i和j同时后移
            next[i]=j;//将最长前缀表赋值给next数组
        }


        for(int i=0,j=0;i<n;i++){
            while(j>0&&txt[i]!=pat[j])j=next[j-1];
            if(txt[i]==pat[j])j++;
            if(j==m){//如果文本串中存在模拟串就退出
                return i-j+1;
            }
        }
        return -1;
    }
}
```