1 上传 2个 蓝桥杯、ACM等相关的题目及其解答过程，题目自选

2 27. 移除元素（简单）

2-1 题目描述

这是 LeetCode 上的 **27. 移除元素** ，难度为 **简单** 。

Tag : 「数组」、「双指针」、「数组移除元素问题」

给你一个数组 `nums` 和一个值 `val`，你需要「原地」移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 $O(1)$ 额外空间并「原地」修改输入数组。

元素的顺序可以改变。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

示例 1：

```
输入：nums = [3,2,2,3], val = 3

输出：2, nums = [2,2]

解释：函数应该返回新的长度 2，并且 nums 中的前两个元素均为 2。你不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度为 2 ，而 nums = [2,2,3,3] 或 nums = [2,2,0,0]，也会被视作正确答案。
```

示例 2：

```
输入：nums = [0,1,2,2,3,0,4,2], val = 2

输出：5, nums = [0,1,4,0,3]

解释：函数应该返回新的长度 5，并且 nums 中的前五个元素为 0，1，3，0，4。注意这五个元素可为任意顺序。你不需要考虑数组中超出新长度后面的元素。
```

提示：

- $0 <= nums.length <= 100$
- $0 <= nums[i] <= 50$
- $0 <= val <= 100$

2-2 双指针

本解法的思路与 【题解】26. 删除排序数组中的重复项 中的「双指针解法」类似。

根据题意，我们可以将数组分成「前后」两段：

- 前半段是有效部分，存储的是不等于 `val` 的元素。
- 后半段是无效部分，存储的是等于 `val` 的元素。

最终答案返回有效部分的结尾下标。

代码：

```java
class Solution {
    public int removeElement(int[] nums, int val) {
        int j = nums.length - 1;
        for (int i = 0; i <= j; i++) {
            if (nums[i] == val) {
                swap(nums, i--, j--);
            }
        }
        return j + 1;
    }
    void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

2-3 通用解法

本解法的思路与 【题解】26. 删除排序数组中的重复项 中的「通用解法」类似。

先设定变量 `idx` ，指向待插入位置。`idx` 初始值为 0

然后从题目的「要求/保留逻辑」出发，来决定当遍历到任意元素 `x` 时，应该做何种决策：

- 如果当前元素 `x` 与移除元素 `val` 相同，那么跳过该元素。
- 如果当前元素 `x` 与移除元素 `val` 不同，那么我们将其放到下标 `idx` 的位置，并让 `idx` 自增右移。

最终得到的 `idx` 即是答案。

代码：

```java
class Solution {
    public int removeElement(int[] nums, int val) {
        int idx = 0;
        for (int x : nums) {
            if (x != val) nums[idx++] = x;
        }
        return idx;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

2-4 总结

对于诸如「相同元素最多保留 `k` 位元素」或者「移除特定元素」的问题，更好的做法是从题目本身性质出发，利用题目给定的要求提炼出具体的「保留逻辑」，将「保留逻辑」应用到我们的遍历到的每一个位置。

3 28. 实现 StrStr()（简单）

3-1 题目描述

这是 LeetCode 上的 **28. 实现 strStr()** ，难度为 **简单** 。

Tag : 「子串匹配」、「KMP」

实现 `strStr()` 函数。

给你两个字符串 `haystack` 和 `needle` ，请你在 `haystack` 字符串中找出 `needle` 字符串出现的第一个位置（下标从 0 开始）。

如果不存在，则返回 -1 。

说明：

- 当 `needle` 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。
- 对于本题而言，当 `needle` 是空字符串时我们应当返回 0 。这与 C 语言的 `strstr()` 以及 Java 的 `indexOf()` 定义相符。

示例 1：

```
输入: haystack = "hello", needle = "ll"

输出: 2
```

示例 2：

```
输入: haystack = "aaaaa", needle = "bba"

输出: -1
```

示例 3：

```
输入: haystack = "", needle = ""

输出: 0
```

提示：

- $0 <= haystack.length, needle.length <= 5 \times 10^4$
- `haystack` 和 `needle` 仅由小写英文字符组成

3-2 朴素解法

直观的解法的是：枚举原串 `ss` 中的每个字符作为「发起点」，每次从原串的「发起点」和匹配串的「首位」开始尝试匹配：

- 匹配成功：返回本次匹配的原串「发起点」。
- 匹配失败：枚举原串的下一个「发起点」，重新尝试匹配。

代码：

```
class Solution {
    public int strStr(String ss, String pp) {
        int n = ss.length(), m = pp.length();
        char[] s = ss.toCharArray(), p = pp.toCharArray();
        // 枚举原串的「发起点」
        for (int i = 0; i <= n - m; i++) {
            // 从原串的「发起点」和匹配串的「首位」开始，尝试匹配
            int a = i, b = 0;
            while (b < m && s[a] == p[b]) {
                a++;
                b++;
            }
            // 如果能够完全匹配，返回原串的「发起点」下标
            if (b == m) return i;
        }
        return -1;
    }
}
```

- 时间复杂度：`n` 为原串的长度，`m` 为匹配串的长度。其中枚举的复杂度为 $O(n-m)$，构造和比较字符串的复杂度为 $O(m)$。整体复杂度为 $O((n-m) \times m)$。
- 空间复杂度：$O(1)$。

3-3 KMP 解法

KMP 算法是一个快速查找匹配串的算法，它的作用其实就是本题问题：**如何快速在「原字符串」中找到「匹配字符串」。**

上述的朴素解法，不考虑剪枝的话复杂度是 $O(m \times n)$ 的，而 KMP 算法的复杂度为 $O(m+n)$。

KMP 之所以能够在 $O(m+n)$ 复杂度内完成查找，是因为其能在「非完全匹配」的过程中提取到有效信息进行复用，以减少「重复匹配」的消耗。

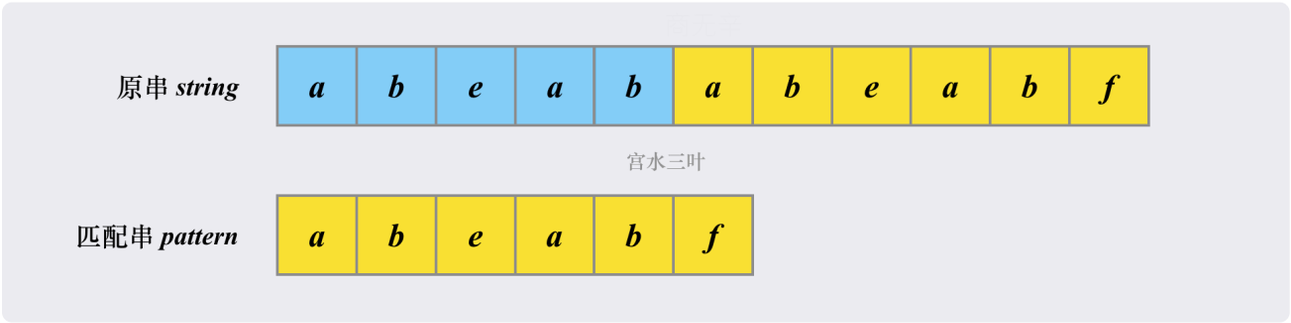
你可能不太理解，没关系，我们可以通过举 🌰 来理解 KMP。

1. 匹配过程

在模拟 KMP 匹配过程之前，我们先建立两个概念：

- 前缀：对于字符串 `abcxxxxefg`，我们称 `abc` 属于 `abcxxxxefg` 的某个前缀。
- 后缀：对于字符串 `abcxxxxefg`，我们称 `efg` 属于 `abcxxxxefg` 的某个后缀。

然后我们假设原串为 `abeababeabf`，匹配串为 `abeabf`：



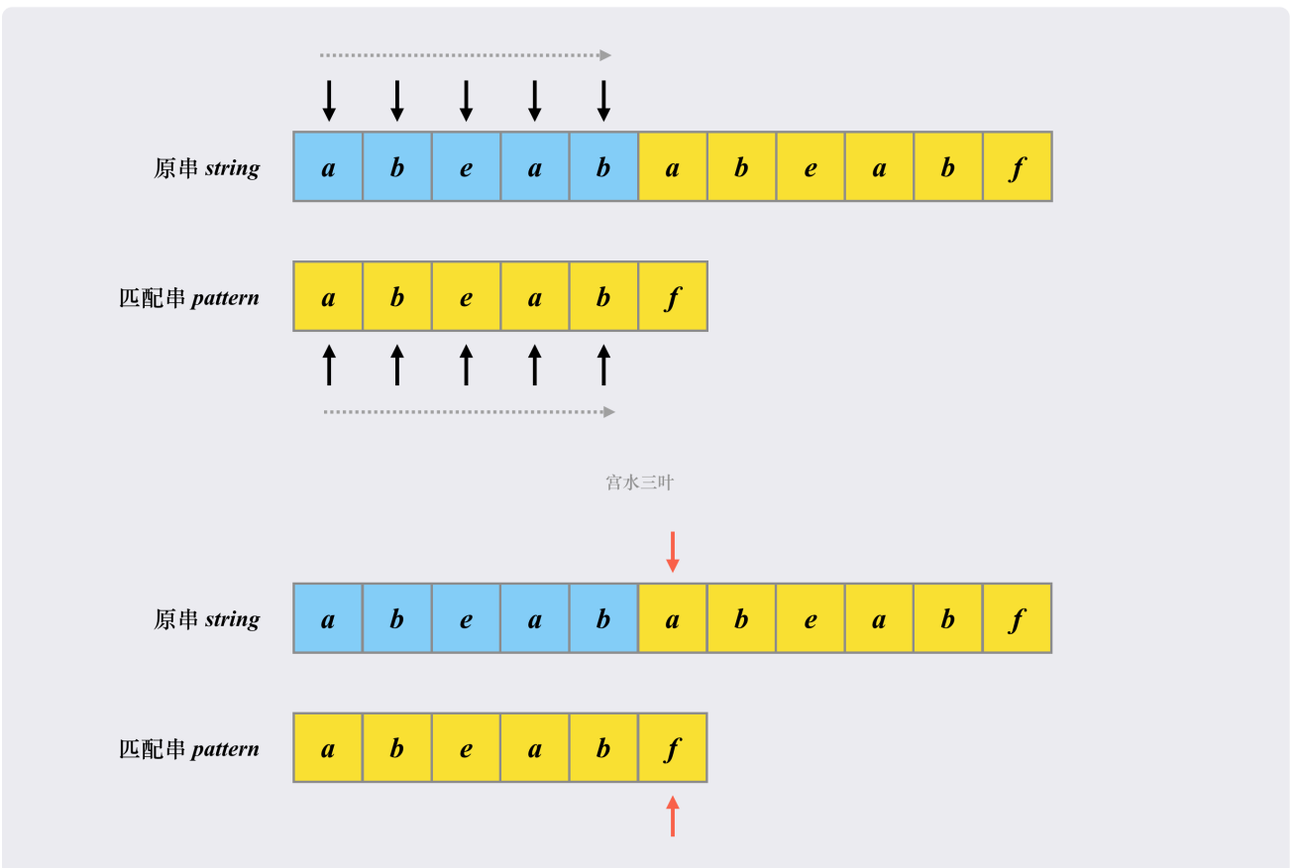
我们可以先看看如果不使用 KMP，会如何进行匹配（不使用 `substring` 函数的情况下）。

首先在「原串」和「匹配串」分别各自有一个指针指向当前匹配的位置。

首次匹配的「发起点」是第一个字符 `a`。显然，后面的 `abeab` 都是匹配的，两个指针会同时往右移动（黑标）。

在都能匹配上 `abeab` 的部分，「朴素匹配」和「KMP」并无不同。

直到出现第一个不同的位置（红标）：



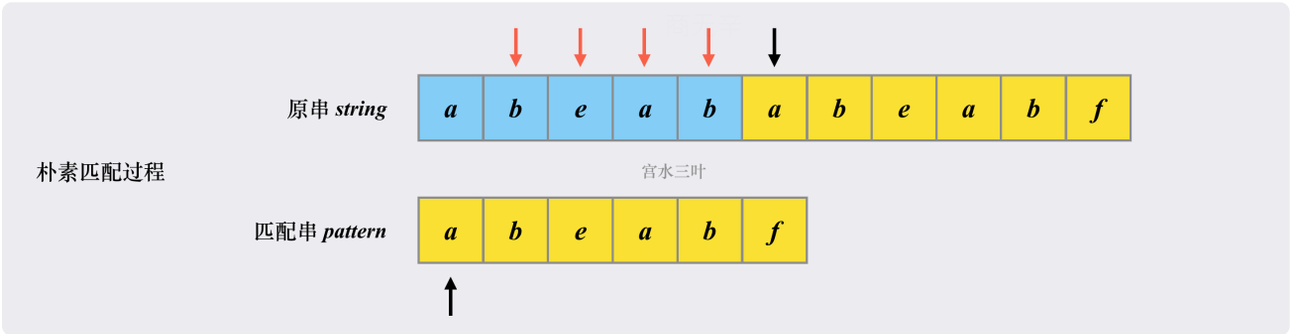
接下来，正是「朴素匹配」和「KMP」出现不同的地方：

- **先看下「朴素匹配」逻辑：**

1. 将原串的指针移动至本次「发起点」的下一个位置（`b` 字符处）；匹配串的指针移动至起始位置。

2. 尝试匹配，发现对不上，原串的指针会一直往后移动，直到能够与匹配串对上位置。

如图：

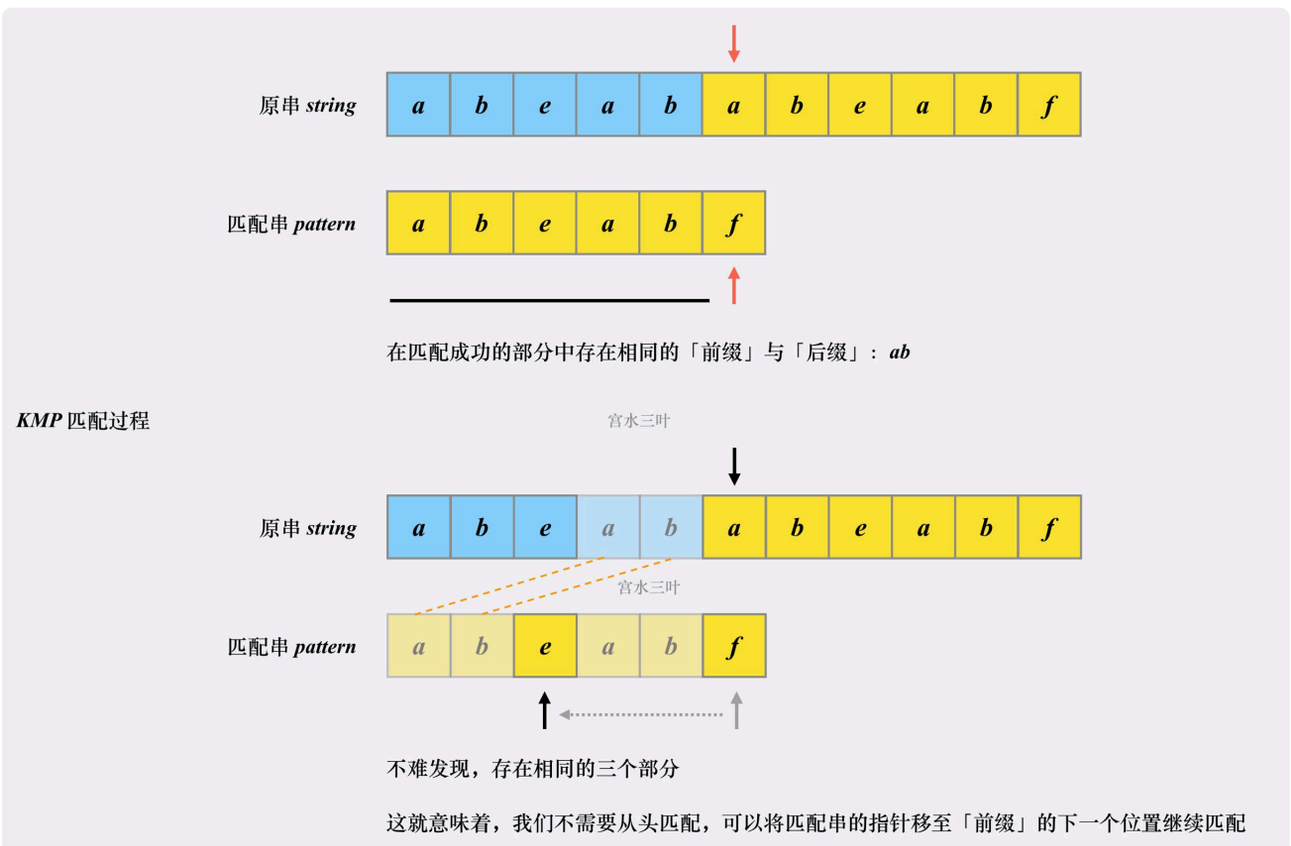


也就是说，对于「朴素匹配」而言，一旦匹配失败，将会将原串指针调整至下一个「发起点」，匹配串的指针调整至起始位置，然后重新尝试匹配。

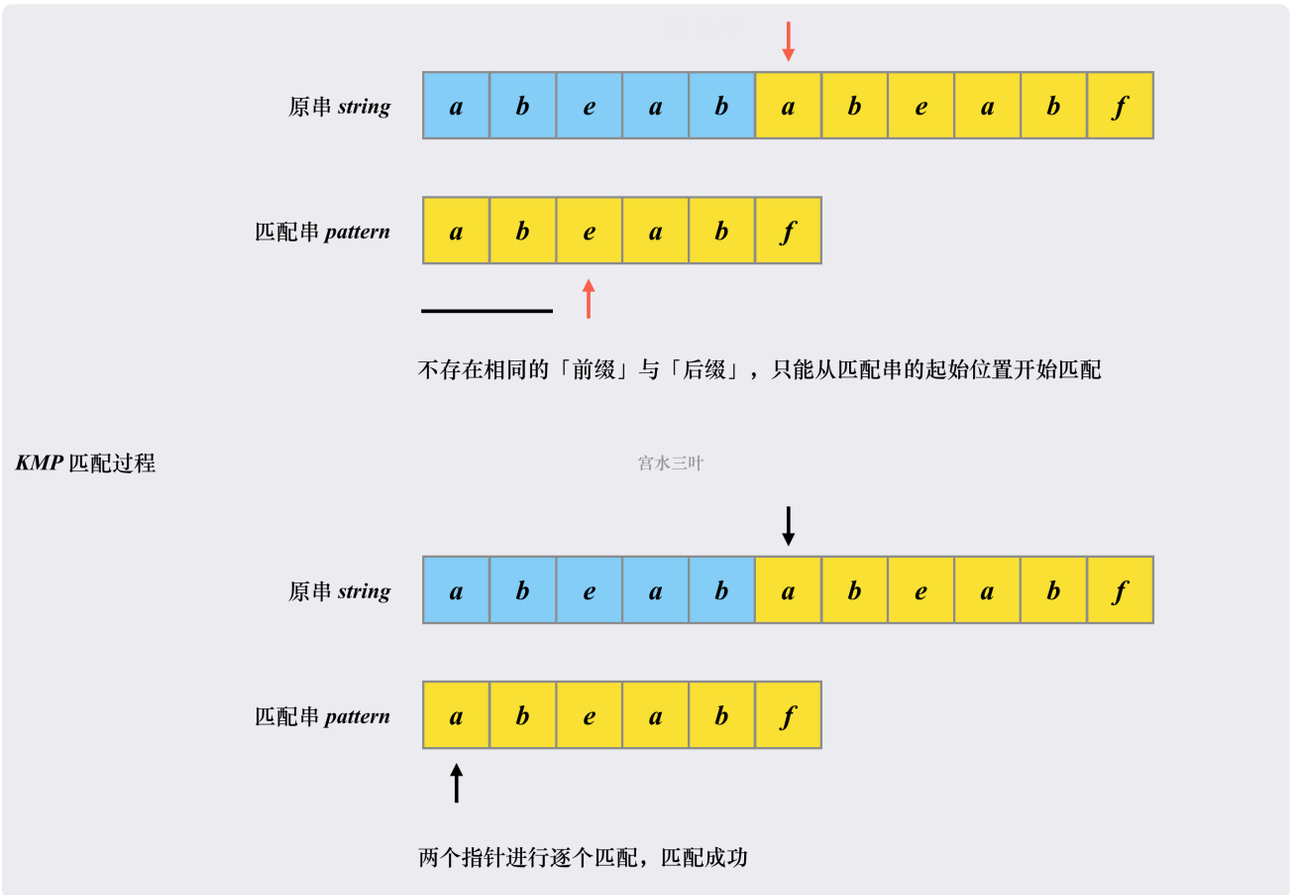
这也就不难理解为什么「朴素匹配」的复杂度是 $O(m \times n)$ 了。

- **然后我们再看看「KMP 匹配」过程：**

首先匹配串会检查之前已经匹配成功的部分中里是否存在相同的「前缀」和「后缀」。如果存在，则跳转到「前缀」的下一个位置继续往下匹配：



跳转到下一匹配位置后，尝试匹配，发现两个指针的字符对不上，并且此时匹配串指针前面不存在相同的「前缀」和「后缀」，这时候只能回到匹配串的起始位置重新开始：



到这里，你应该清楚 KMP 为什么相比于朴素解法更快：

- **因为 KMP 利用已匹配部分中相同的「前缀」和「后缀」来加速下一次的匹配。**
- **因为 KMP 的原串指针不会进行回溯（没有朴素匹配中回到下一个「发起点」的过程）。**

第一点很直观，也很好理解。

我们可以把重点放在第二点上，原串不回溯至「发起点」意味着什么？

其实是意味着：随着匹配过程的进行，原串指针的不断右移，我们本质上是在不断地在否决一些「不可能」的方案。

当我们的原串指针从 `i` 位置后移到 `j` 位置，不仅仅代表着「原串」下标范围为 $[i,j)$ 的字符与「匹配串」匹配或者不匹配，更是在否决那些以「原串」下标范围为 $[i,j)$ 为「匹配发起点」的子集。

II 2. 分析实现

到这里，就结束了吗？要开始动手实现上述匹配过程了吗？

我们可以先分析一下复杂度。如果严格按照上述解法的话，最坏情况下我们需要扫描整个原串，复杂度为 $O(n)$。同时在每一次匹配失败时，去检查已匹配部分的相同「前缀」和「后缀」，跳转到相应的位置，如果不匹配则再检查前面部分是否有相同「前缀」和「后缀」，再跳转到相应的位置 ... 这部分的复杂度是 $O(m^2)$ ，因此整体的复杂度是 $O(n \times m^2)$，而我们的朴素解法是 $O(m \times n)$ 的。

说明还有一些性质我们没有利用到。

显然，扫描完整原串操作这一操作是不可避免的，我们可以优化的只能是「**检查已匹配部分的相同前缀和后缀**」这一过程。

再进一步，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」。

同时我们发现，对于匹配串的任意一个位置而言，由该位置发起的下一个匹配点位置其实与原串无关。

举个 🌰，对于匹配串 `abcabd` 的字符 `d` 而言，由它发起的下一个匹配点跳转必然是字符 `c` 的位置。因为字符 `d` 位置的相同「前缀」和「后缀」字符 `ab` 的下一位置就是字符 `c`。

可见从匹配串某个位置跳转下一个匹配位置这一过程是与原串无关的，我们将这一过程称为找 `next` 点。

显然我们可以预处理出 `next` 数组，数组中每个位置的值就是该下标应该跳转的目标位置（`next` 点）。

当我们进行了这一步优化之后，复杂度是多少呢？

预处理 `next` 数组的复杂度未知，匹配过程最多扫描完整个原串，复杂度为 $O(n)$。

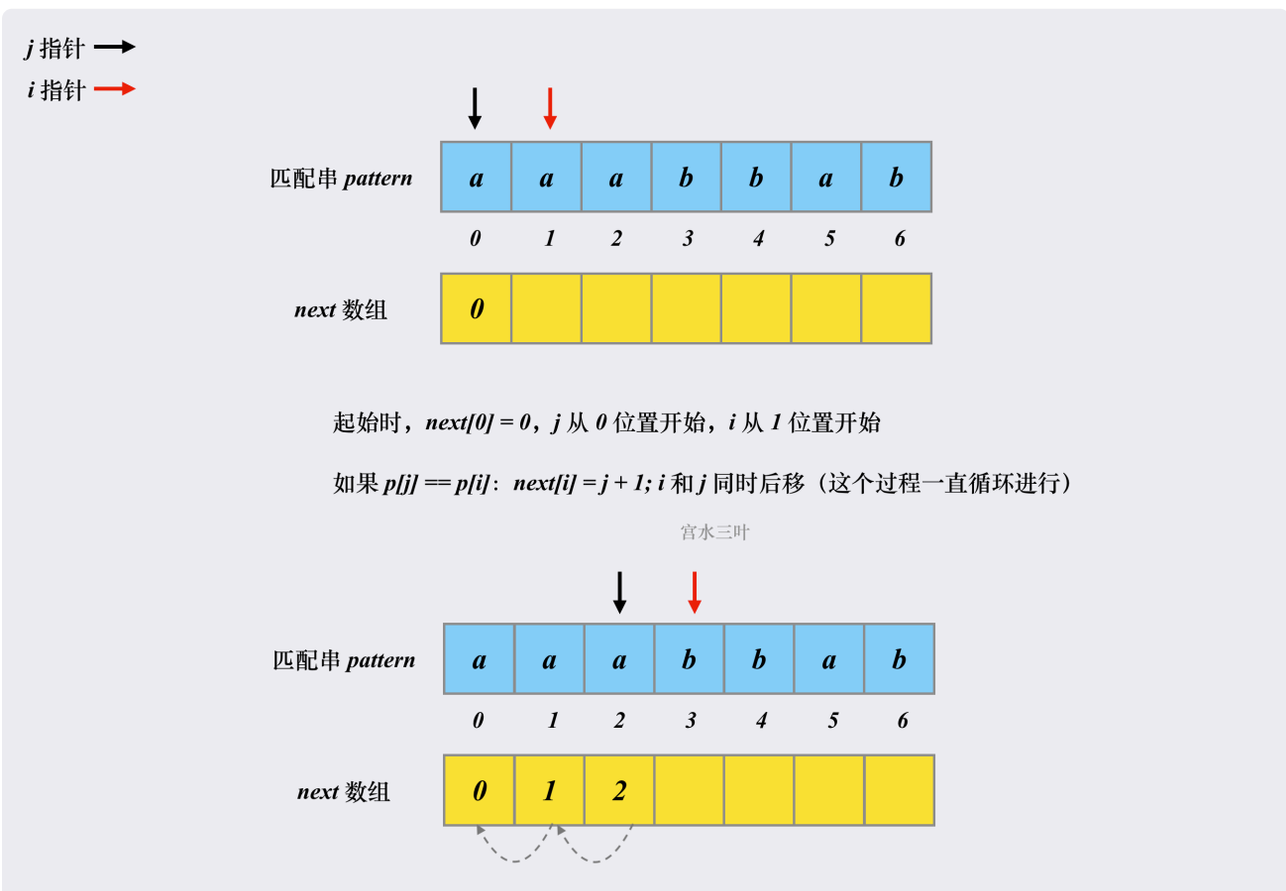
因此如果我们希望整个 KMP 过程是 $O(m+n)$ 的话，那么我们需要在 $O(m)$ 的复杂度内预处理出 `next` 数组。

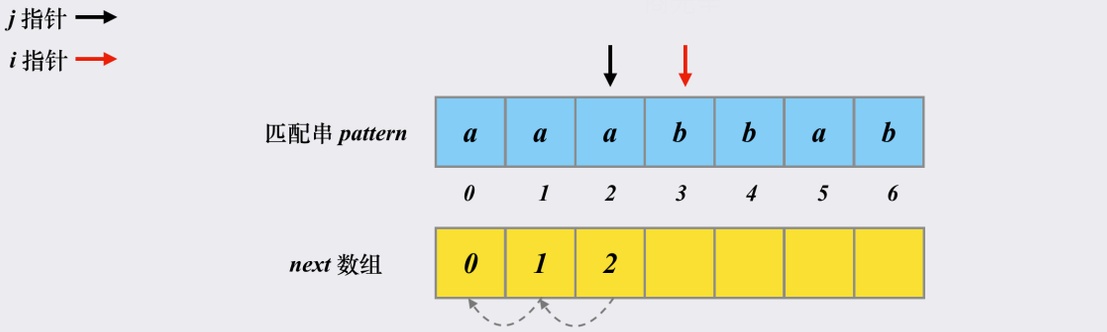
所以我们的重点在于如何在 $O(m)$ 复杂度内处理处 `next` 数组。

III 3. `Next` 数组的构建

接下来，我们看看 `next` 数组是如何在 $O(m)$ 的复杂度内被预处理出来的。

假设有匹配串 `aaabbab`，我们来看看对应的 `next` 是如何被构建出来的。

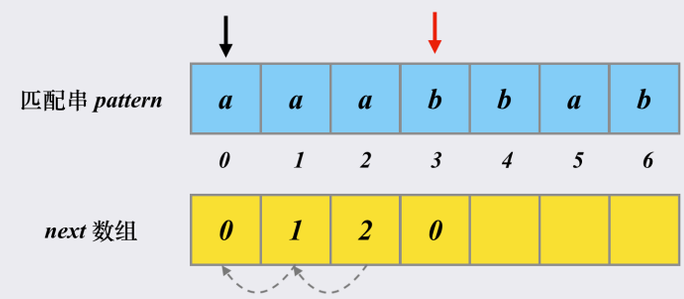


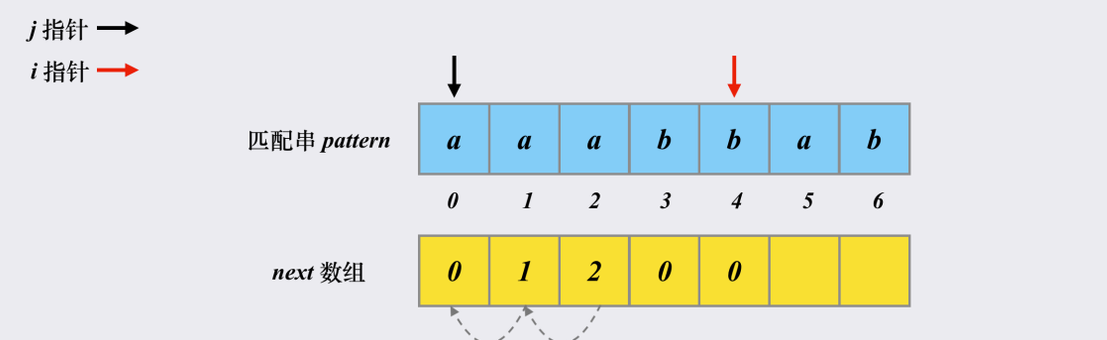


如果 *p[j] != p[i]*：将 *j* 指针指向前一位置的 *next* 数组所对应的值，即 *j = next[j - 1]*

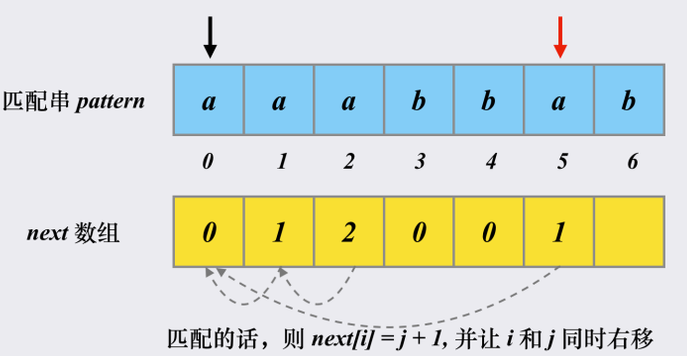
这个过程同样是循环进行，直到 *p[j] == p[i]* 或者 *j == 0*

如果 *j == 0* , *p[i]* 和 *p[j]* 还是不相等，则 *next[i] = 0*，*i* 往后移动，*j* 不变

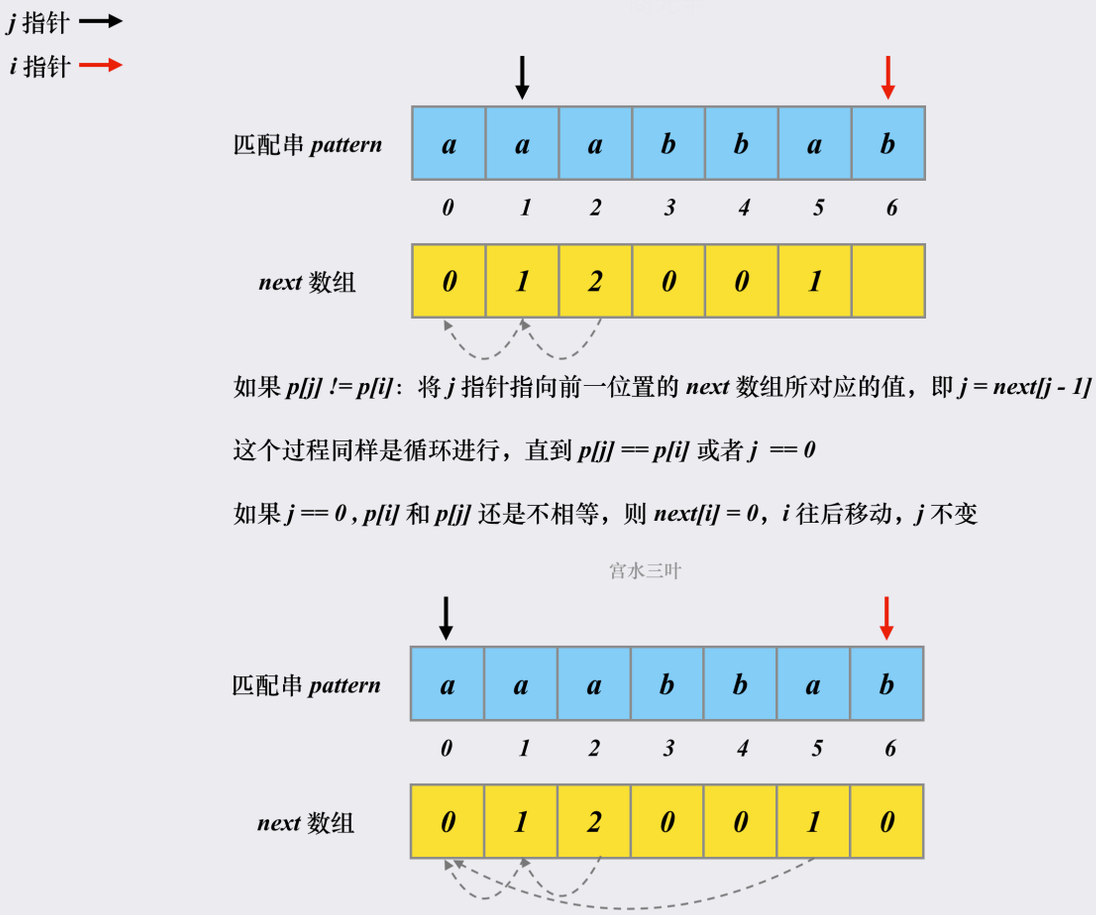




继续这个过程，不匹配，而且 *j = 0*, 直接让 *next[i] = 0*, *i* 指针后移



匹配的话，则 *next[i] = j + 1*, 并让 *i* 和 *j* 同时右移



这就是整个 `next` 数组的构建过程，时空复杂度均为 $O(m)$。

至此整个 KMP 匹配过程复杂度是 $O(m+n)$ 的。

IV 4. 代码实现

在实际编码时，通常我会往原串和匹配串头部追加一个空格（哨兵）。

目的是让 `j` 下标从 `0` 开始，省去 `j` 从 `-1` 开始的麻烦。

整个过程与上述分析完全一致，一些相关的注释我已经写到代码里。

代码：

```java
class Solution {
    // KMP 算法
    // ss: 原串(string)  pp: 匹配串(pattern)
    public int strStr(String ss, String pp) {
        if (pp.isEmpty()) return 0;

        // 分别读取原串和匹配串的长度
        int n = ss.length(), m = pp.length();
        // 原串和匹配串前面都加空格，使其下标从 1 开始
        ss = " " + ss;
        pp = " " + pp;

        char[] s = ss.toCharArray();
        char[] p = pp.toCharArray();

        // 构建 next 数组，数组长度为匹配串的长度（next 数组是和匹配串相关的）
        int[] next = new int[m + 1];
        // 构造过程 i = 2，j = 0 开始，i 小于等于匹配串长度 【构造 i 从 2 开始】
        for (int i = 2, j = 0; i <= m; i++) {
            // 匹配不成功的话，j = next(j)
            while (j > 0 && p[i] != p[j + 1]) j = next[j];
            // 匹配成功的话，先让 j++
            if (p[i] == p[j + 1]) j++;
            // 更新 next[i]，结束本次循环，i++
            next[i] = j;
        }

        // 匹配过程，i = 1，j = 0 开始，i 小于等于原串长度 【匹配 i 从 1 开始】
        for (int i = 1, j = 0; i <= n; i++) {
            // 匹配不成功 j = next(j)
            while (j > 0 && s[i] != p[j + 1]) j = next[j];
            // 匹配成功的话，先让 j++，结束本次循环后 i++
            if (s[i] == p[j + 1]) j++;
            // 整一段匹配成功，直接返回下标
            if (j == m) return i - m;
        }

        return -1;
    }
}
```

- 时间复杂度：`n` 为原串的长度，`m` 为匹配串的长度。复杂度为 $O(m+n)$。
- 空间复杂度：构建了 `next` 数组。复杂度为 $O(m)$。

3-4 总结

KMP 算法的应用范围要比 Manacher 算法广，Manacher 算法只能应用于「回文串」问题，相对比较局限，而「子串匹配」问题还是十分常见的。

背过这样的算法的意义在于：相当于大脑里有了一个时间复杂度为 $O(n)$ 的 api 可以使用，这个 api 传入一个原串和匹配串，返回匹配串在原串的位置。

因此，三叶十分建议大家在「理解 KMP」的基础上，对模板进行背过 ~