33 | 字符串匹配基础（中）：如何实现文本编辑器中的查找功能?

2018-12-07 王争

33 | 字符串匹配基础（中）：如何实现文本编辑器中的查找功能?
朗读人：修阳　18′20″ | 16.80M

文本编辑器中的查找替换功能，我想你应该不陌生吧？比如，我们在 Word 中把一个单词统一替换成另一个，用的就是这个功能。你有没有想过，它是怎么实现的呢?

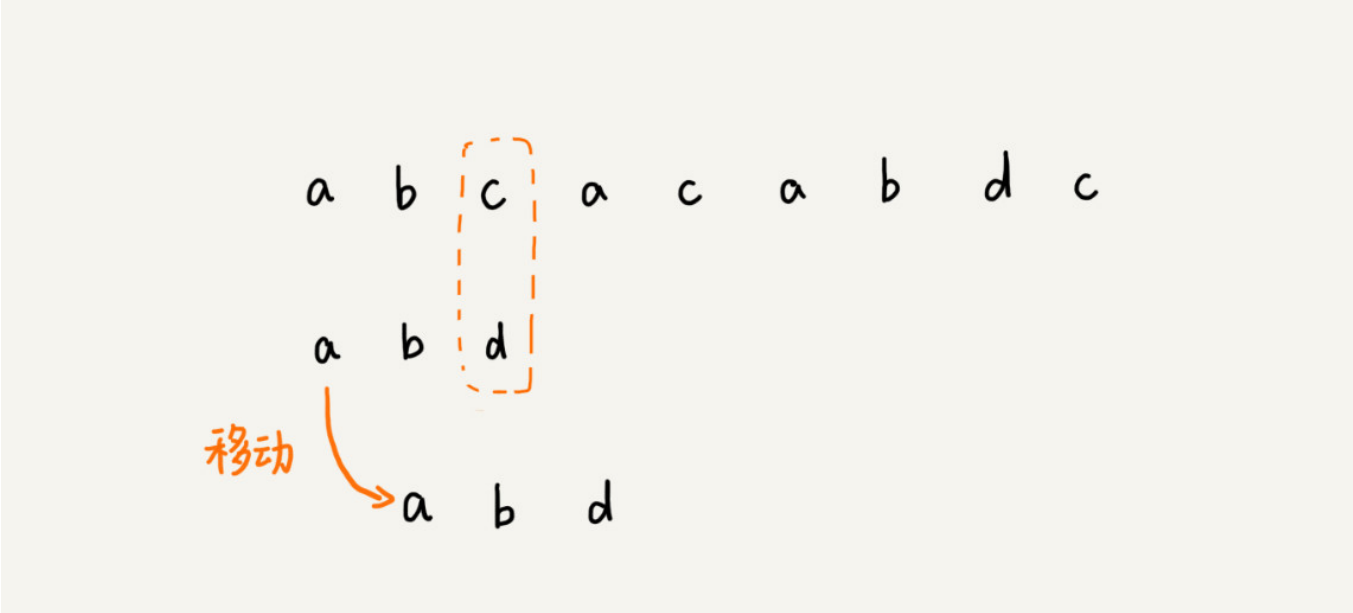
当然，你用上一节讲的 BF 算法和 RK 算法，也可以实现这个功能，但是在某些极端情况下，BF 算法性能会退化的比较严重，而 RK 算法需要用到哈希算法，而设计一个可以应对各种类型字符的哈希算法并不简单。

对于工业级的软件开发来说，我们希望算法尽可能的高效，并且在极端情况下，性能也不要退化的太严重。那么，**对于查找功能是重要功能的软件来说，比如一些文本编辑器，它们的查找功能都是用哪种算法来实现的呢？有没有比 BF 算法和 RK 算法更加高效的字符串匹配算法呢?**

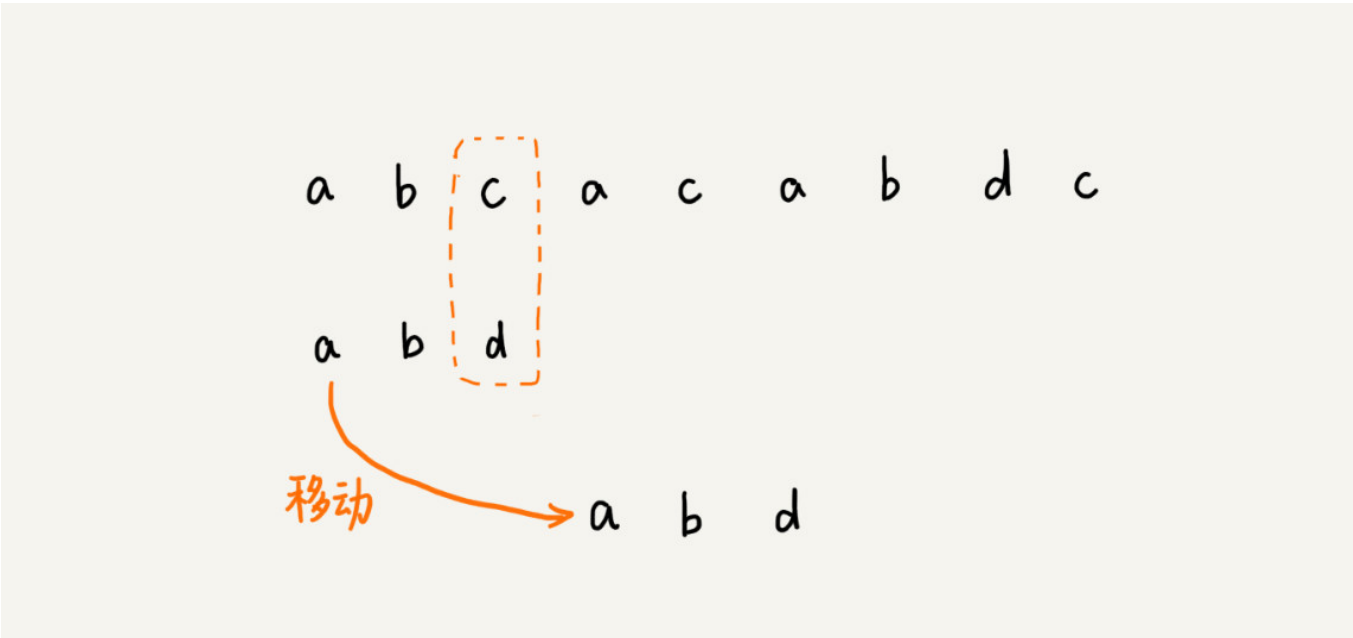
今天，我们就来学习 BM（Boyer-Moore）算法。它是一种非常高效的字符串匹配算法，有实验统计，它的性能是著名的KMP 算法的 3 到 4 倍。BM 算法的原理很复杂，比较难懂，学起来会比较烧脑，我会尽量给你讲清楚，同时也希望你做好打硬仗的准备。好，现在我们正式开始！

BM 算法的核心思想

我们把模式串和主串的匹配过程，看作模式串在主串中不停地往后滑动。当遇到不匹配的字符时，BF 算法和 RK 算法的做法是，模式串往后滑动一位，然后从模式串的第一个字符开始重新匹配。我举个例子解释一下，你可以看我画的这幅图。



在这个例子里，主串中的 c，在模式串中是不存在的，所以，模式串向后滑动的时候，只要 c 与模式串有重合，肯定无法匹配。所以，我们可以一次性把模式串往后多滑动几位，把模式串移动到 c 的后面。



由现象找规律，你可以思考一下，当遇到不匹配的字符时，有什么固定的规律，可以将模式串往后多滑动几位呢？这样一次性往后滑动好几位，那匹配的效率岂不是就提高了？

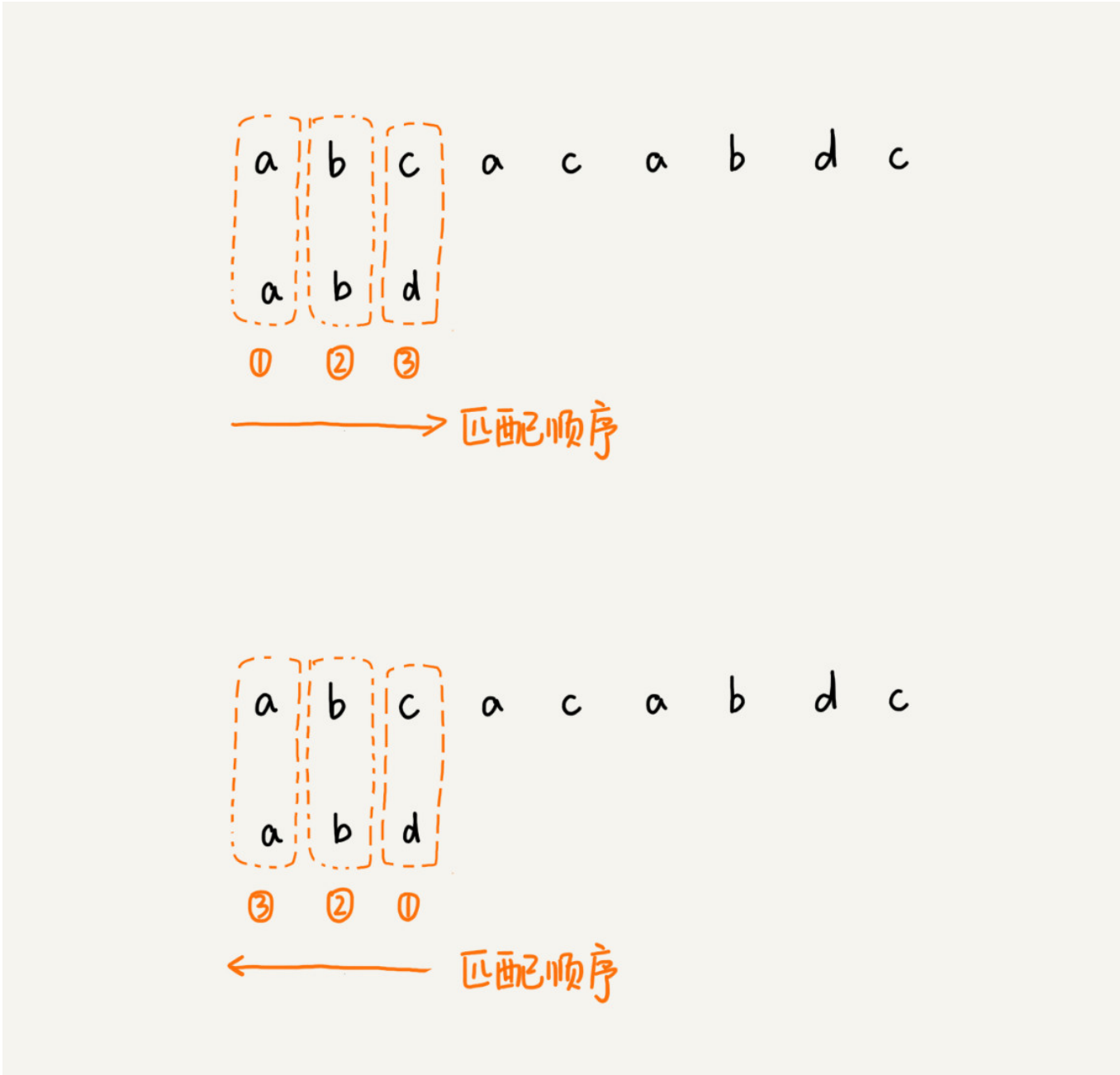
我们今天要讲的 BM 算法，本质上其实就是在寻找这种规律。借助这种规律，在模式串与主串匹配的过程中，当模式串和主串某个字符不匹配的时候，能够跳过一些肯定不会匹配的情况，将模式串往后多滑动几位。

BM 算法原理分析

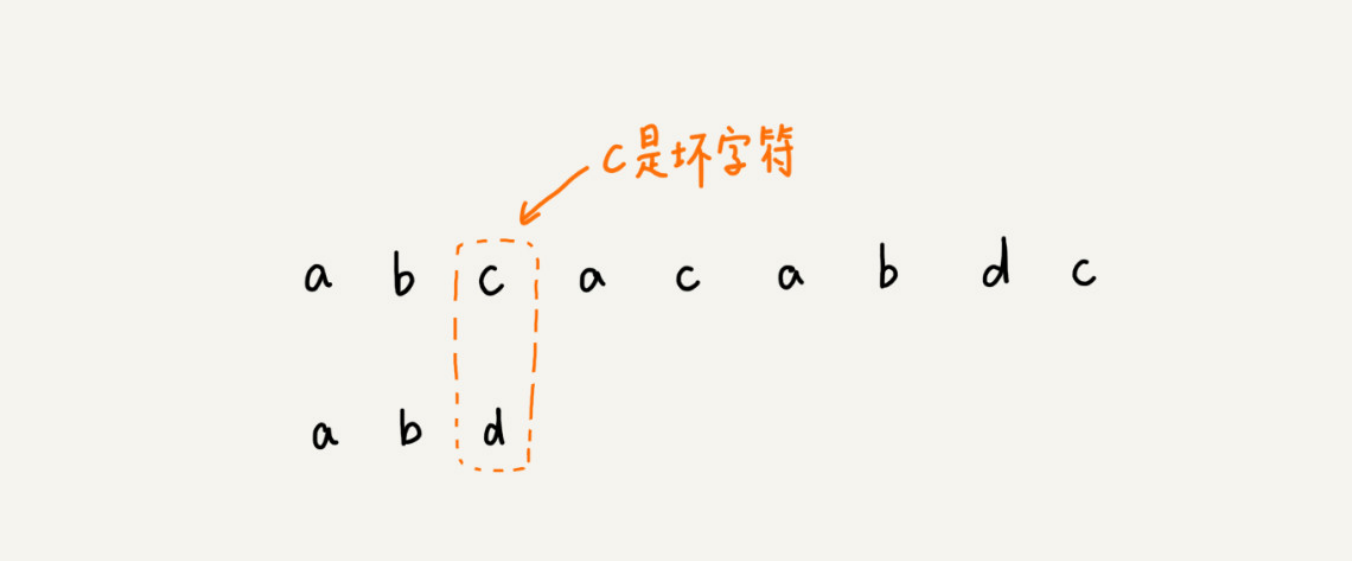
BM 算法包含两部分，分别是**坏字符规则**（bad character rule）和**好后缀规则**（good suffix shift）。我们下面依次来看，这两个规则分别都是怎么工作的。

1. 坏字符规则

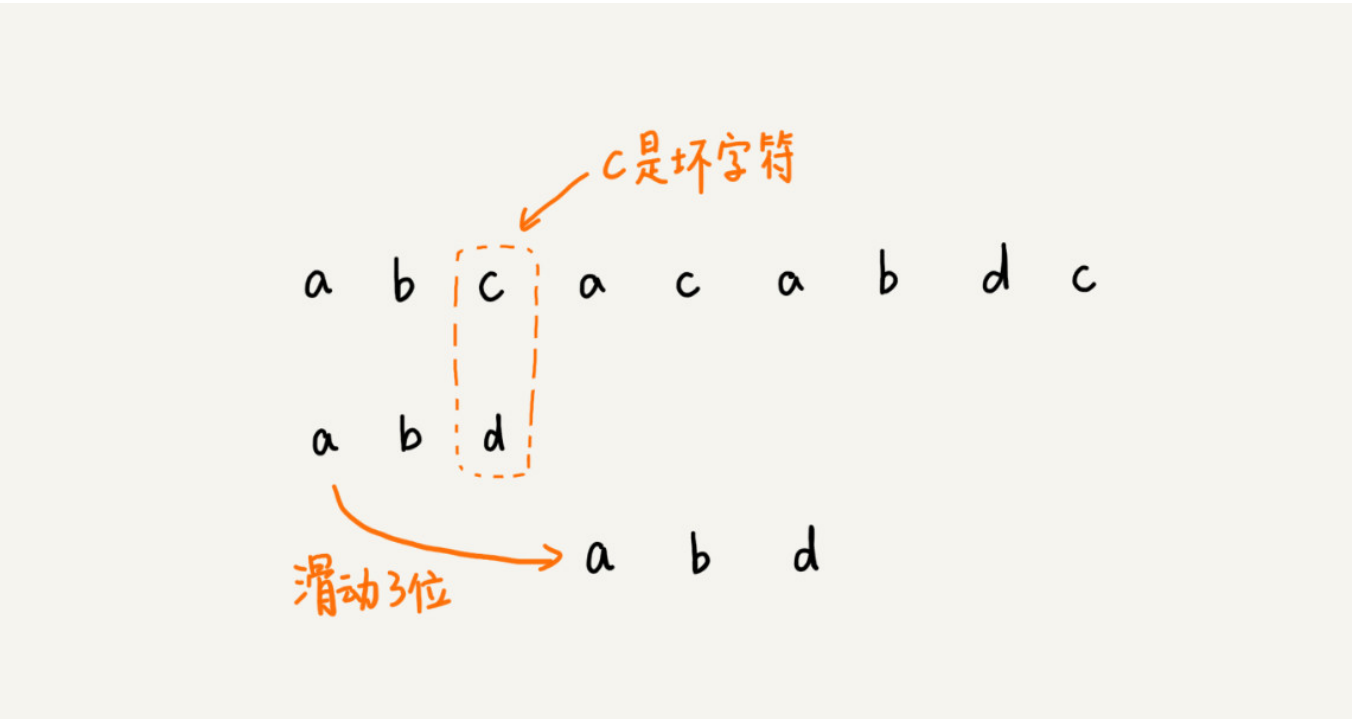
前面两节讲的算法，在匹配的过程中，我们都是按模式串的下标从小到大的顺序，依次与主串中的字符进行匹配的。这种匹配顺序比较符合我们的思维习惯，而 BM 算法的匹配顺序比较特别，它是按照模式串下标从大到小的顺序，倒着匹配的。我画了一张图，你可以看下。



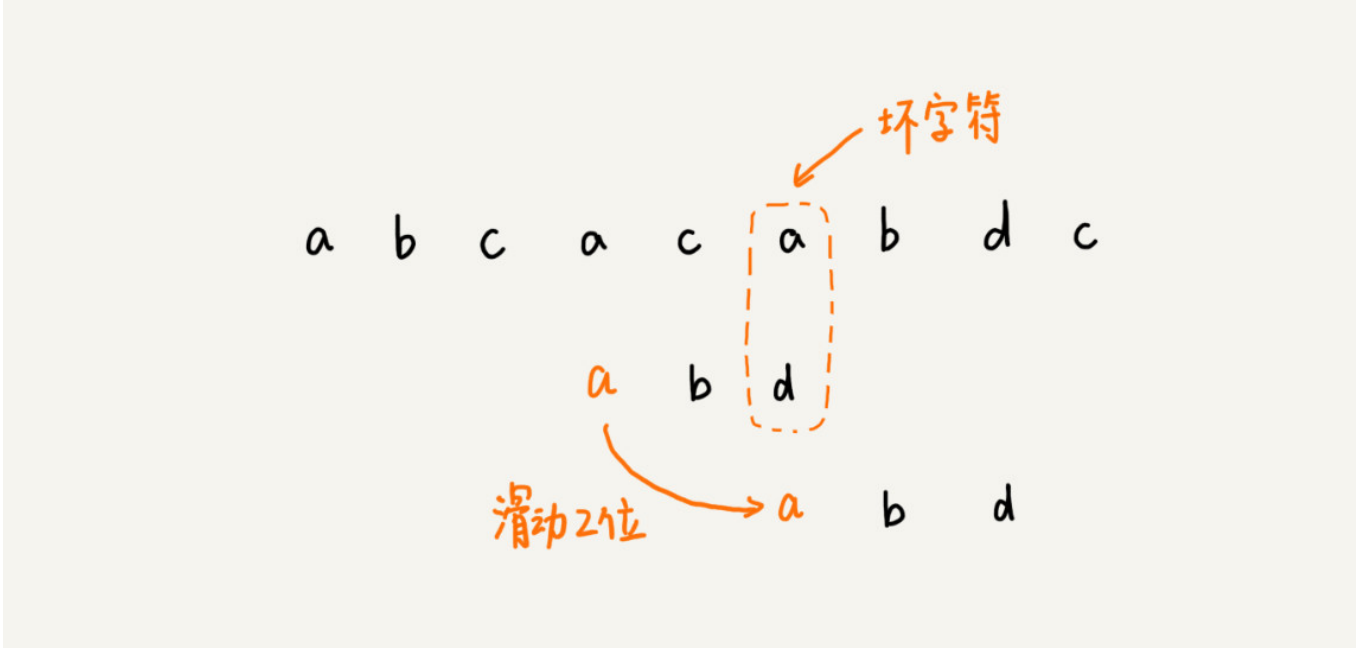
我们从模式串的末尾往前倒着匹配，当我们发现某个字符没法匹配的时候。我们把这个没有匹配的字符叫作**坏字符**（主串中的字符）。



我们拿坏字符 c 在模式串中查找，发现模式串中并不存在这个字符，也就是说，字符 c 与模式串中的任何字符都不可能匹配。这个时候，我们可以将模式串直接往后滑动三位，将模式串滑动到 c 后面的位置，再从模式串的末尾字符开始比较。

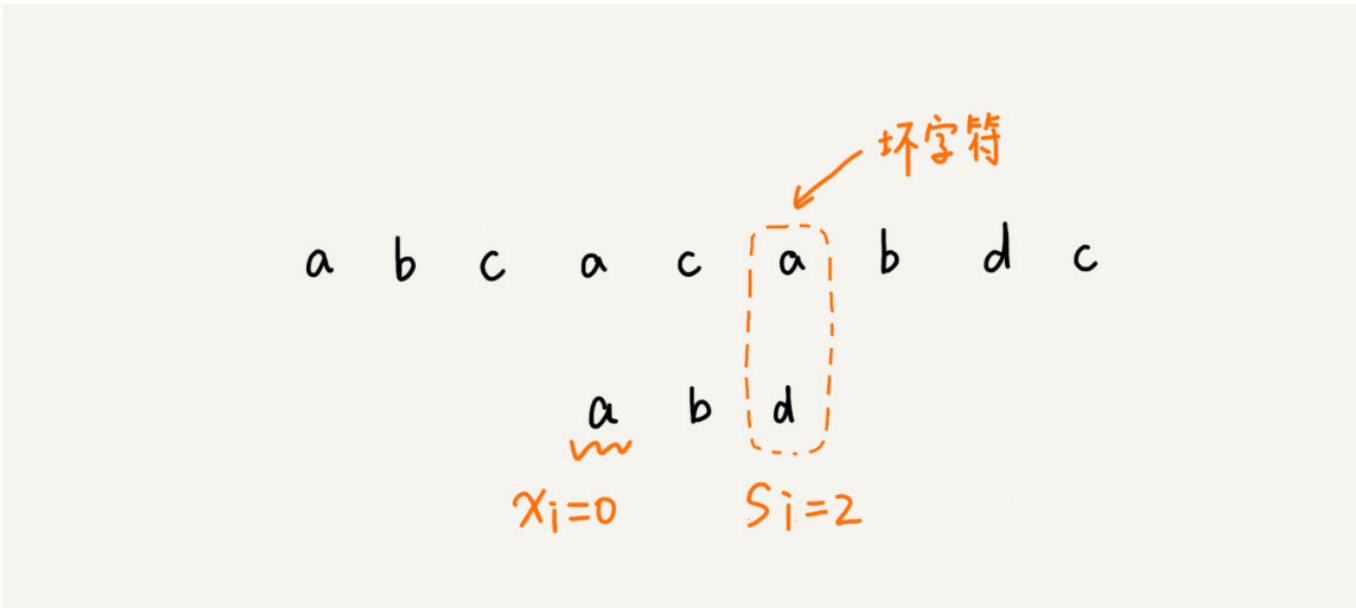


这个时候，我们发现，模式串中最后一个字符 d，还是无法跟主串中的 a 匹配，这个时候，还能将模式串往后滑动三位吗？答案是不行的。因为这个时候，坏字符 a 在模式串中是存在的，模式串中下标是 0 的位置也是字符 a。这种情况下，我们可以将模式串往后滑动两位，让两个 a 上下对齐，然后再从模式串的末尾字符开始，重新匹配。



第一次不匹配的时候，我们滑动了三位，第二次不匹配的时候，我们将模式串后移两位，那具体滑动多少位，到底有没有规律呢？

当发生不匹配的时候，我们把坏字符对应的模式串中的字符下标记作 si。如果坏字符在模式串中存在，我们把这个坏字符在模式串中的下标记作 xi。如果不存在，我们把 xi 记作 –1。那模式串往后移动的位数就等于 si–xi。（注意，我这里说的下标，都是字符在模式串的下标）。



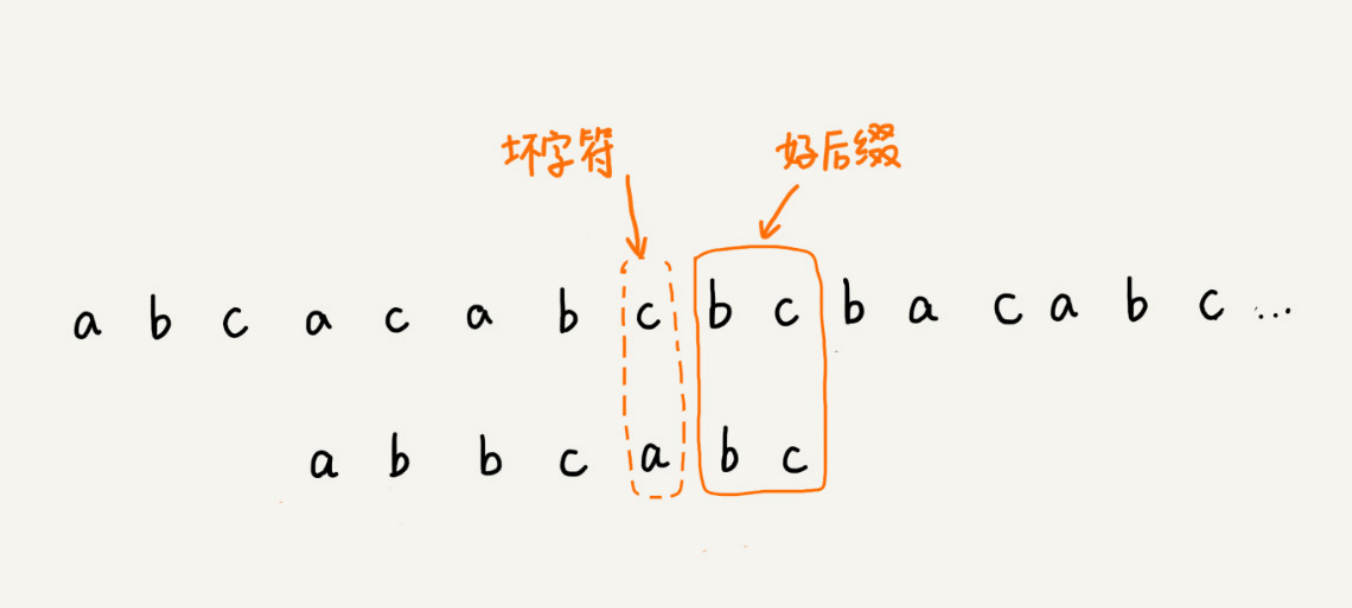
这里我要特别说明一点，如果坏字符在模式串里多处出现，那我们在计算 xi 的时候，选择最靠后的那个，因为这样不会让模式串滑动过多，导致本来可能匹配的情况被滑动略过。

利用坏字符规则，BM 算法在最好情况下的时间复杂度非常低，是 O(n/m)。比如，主串是 aaabaaabaaabaaab，模式串是 aaaa。每次比对，模式串都可以直接后移四位，所以，匹配具有类似特点的模式串和主串的时候，BM 算法非常高效。

不过，单纯使用坏字符规则还是不够的。因为根据 si-xi 计算出来的移动位数，有可能是负数，比如主串是 aaaaaaaaaaaaaaaa，模式串是 baaa。不但不会向后滑动模式串，还有可能倒退。所以，BM 算法还需要用到“好后缀规则”。

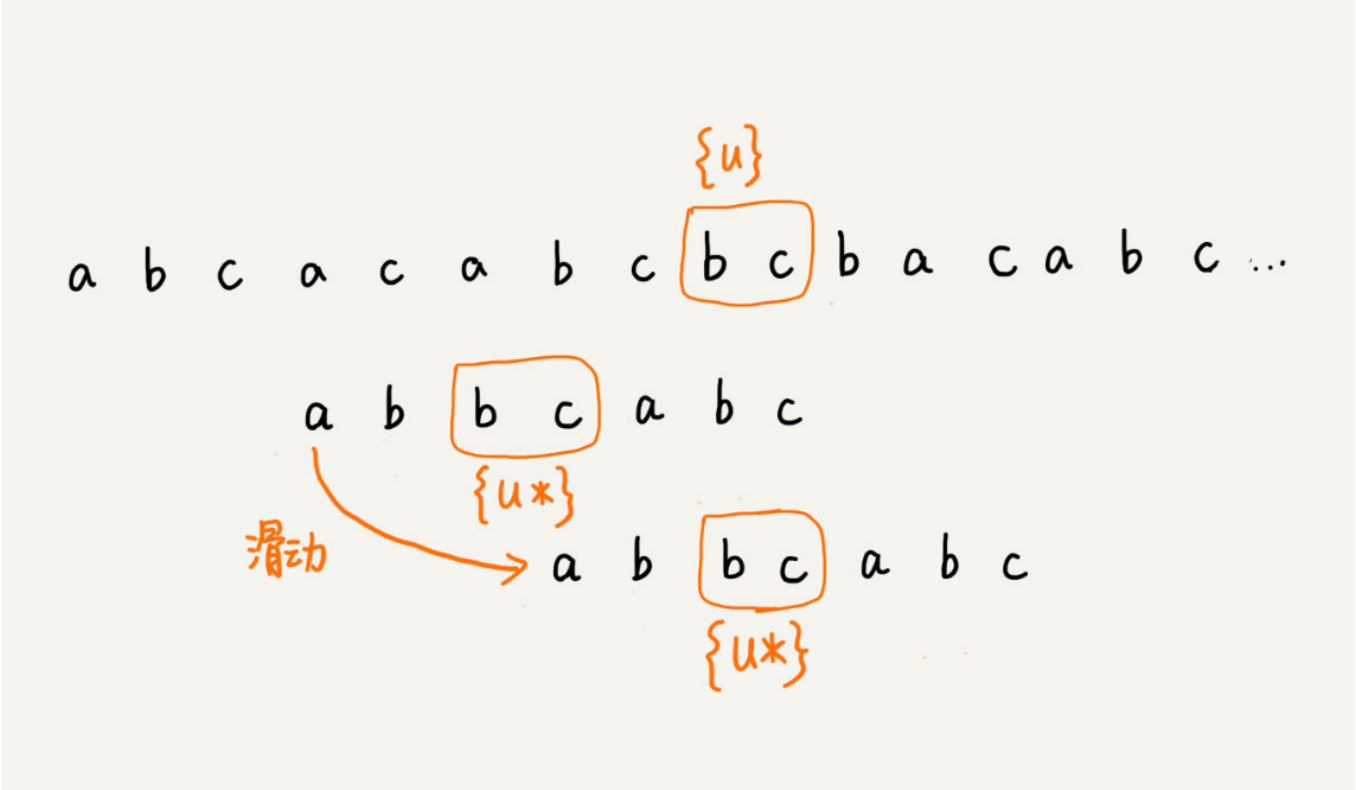
2. 好后缀规则

好后缀规则实际上跟坏字符规则的思路很类似。你看我下面这幅图。当模式串滑动到图中的位置的时候，模式串和主串有 2 个字符是匹配的，倒数第 3 个字符发生了不匹配的情况。

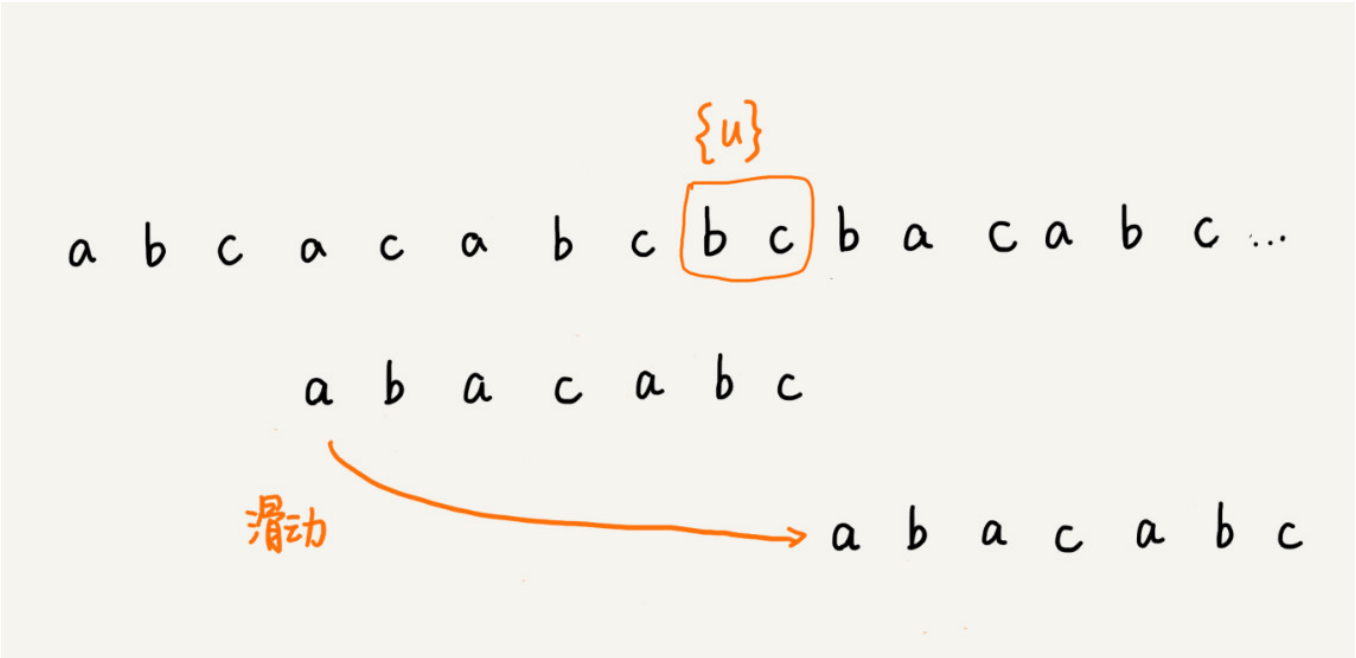


这个时候该如何滑动模式串呢？当然，我们还可以利用坏字符规则来计算模式串的滑动位数，不过，我们也可以使用好后缀处理规则。两种规则到底如何选择，我稍后会讲。抛开这个问题，现在我们来看，好后缀规则是怎么工作的？

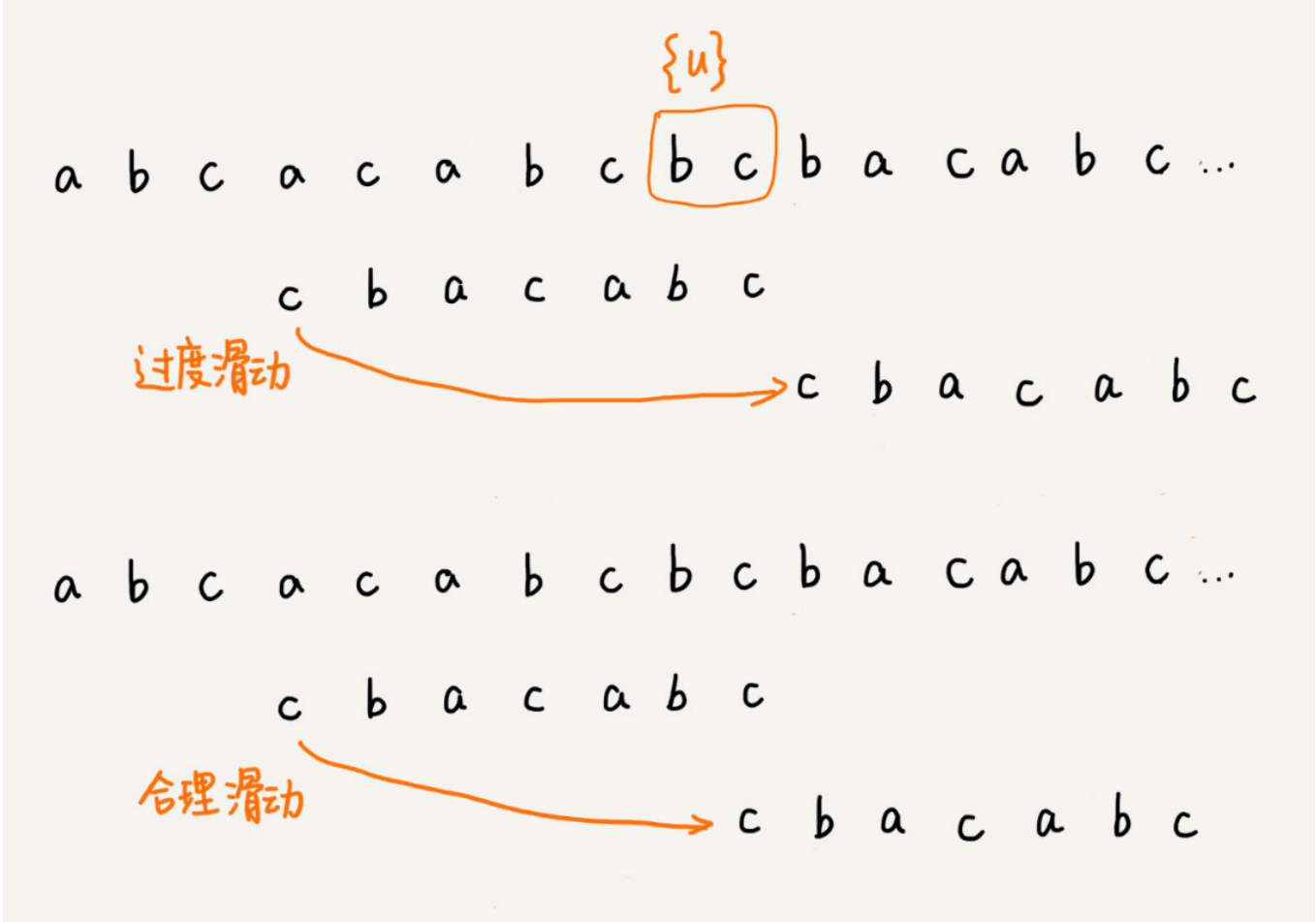
我们把已经匹配的 bc 叫作好后缀，记作{u}。我们拿它在模式串中查找，如果找到了另一个跟{u}相匹配的子串{u*}，那我们就将模式串滑动到子串{u*}与主串中{u}对齐的位置。



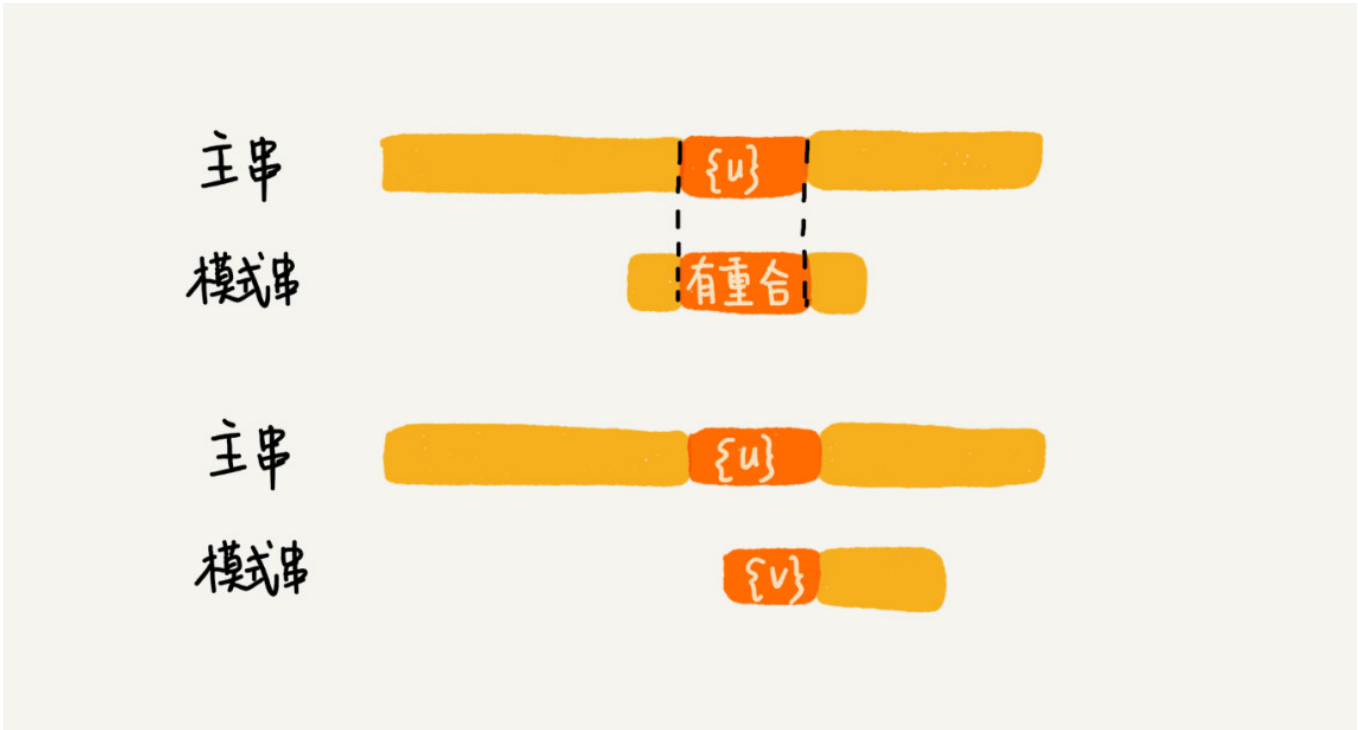
如果在模式串中找不到另一个等于{u}的子串，我们就直接将模式串，滑动到主串中{u}的后面，因为之前的任何一次往后滑动，都没有匹配主串中{u}的情况。



不过，当模式串中不存在等于{u}的子串时，我们直接将模式串滑动到主串{u}的后面。这样做是否有点太过头呢？我们来看下面这个例子。这里面 bc 是好后缀，尽管在模式串中没有另外一个相匹配的子串{u*}，但是如果我们将模式串移动到好后缀的后面，如图所示，那就会错过模式串和主串可以匹配的情况。

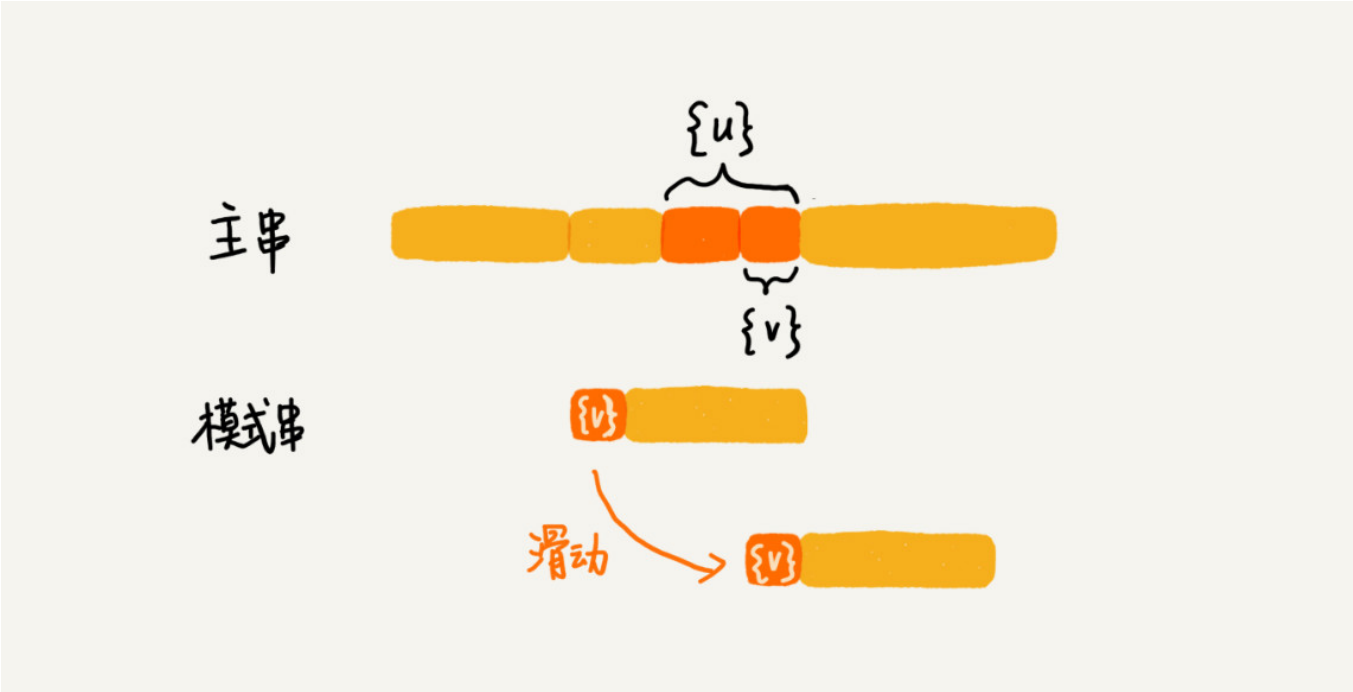


如果好后缀在模式串中不存在可匹配的子串，那在我们一步一步往后滑动模式串的过程中，只要主串中的{u}与模式串有重合，那肯定就无法完全匹配。但是当模式串滑动到前缀与主串中{u}的后缀有部分重合的时候，并且重合的部分相等的时候，就有可能会存在完全匹配的情况。



所以，针对这种情况，我们不仅要看好后缀在模式串中，是否有另一个匹配的子串，我们还要考察好后缀的后缀子串，是否存在跟模式串的前缀子串匹配的。

所谓某个字符串 s 的后缀子串，就是最后一个字符跟 s 对齐的子串，比如 abc 的后缀子串就包括 c, bc。所谓前缀子串，就是起始字符跟 s 对齐的子串，比如 abc 的前缀子串有 a，ab。我们从好后缀的后缀子串中，找一个最长的并且能跟模式串的前缀子串匹配的，假设是{v}，然后将模式串滑动到如图所示的位置。



坏字符和好后缀的基本原理都讲完了，我现在回答一下前面那个问题。当模式串和主串中的某个字符不匹配的时候，如何选择用好后缀规则还是坏字符规则，来计算模式串往后滑动的位数?

我们可以分别计算好后缀和坏字符往后滑动的位数，然后取两个数中最大的，作为模式串往后滑动的位数。这种处理方法还可以避免我们前面提到的，根据坏字符规则，计算得到的往后滑动的位数，有可能是负数的情况。

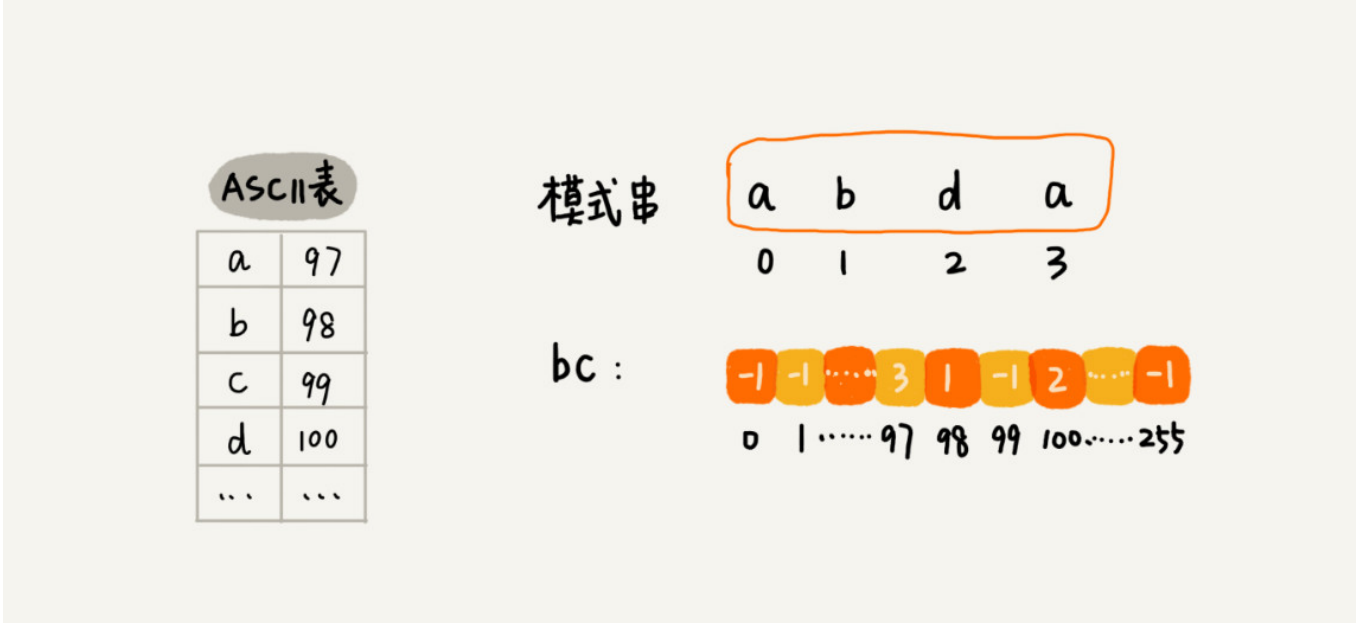
BM 算法代码实现

学习完了基本原理，我们再来看，如何实现 BM 算法?

“坏字符规则”本身不难理解。当遇到坏字符时，要计算往后移动的位数 si−xi，其中 xi 的计算是重点，我们如何求得 xi 呢？或者说，如何查找坏字符在模式串中出现的位置呢?

如果我们拿坏字符，在模式串中顺序遍历查找，这样就会比较低效，势必影响这个算法的性能。有没有更加高效的方式呢？我们之前学的散列表，这里可以派上用场了。我们可以将模式串中的每个字符及其下标都存到散列表中。这样就可以快速找到坏字符在模式串的位置下标了。

关于这个散列表，我们只实现一种最简单的情况，假设字符串的字符集不是很大，每个字符长度是 8 字节，我们用大小为 256 的数组，来记录每个字符在模式串中出现的位置。数组的下标对应字符的 ASCII 码值，数组中存储这个字符在模式串中出现的位置。



如果将上面的过程翻译成代码，就是下面这个样子。其中，变量 b 是模式串，m 是模式串的长度，bc 表示刚刚讲的散列表。

复制代码

```
private static final int SIZE = 256; // 全局变量或成员变量
private void generateBC(char[] b, int m, int[] bc) {
  for (int i = 0; i < SIZE; ++i) {
    bc[i] = -1; // 初始化 bc
  }
  for (int i = 0; i < m; ++i) {
    int ascii = (int)b[i]; // 计算 b[i] 的 ASCII 值
    bc[ascii] = i;
  }
}
```

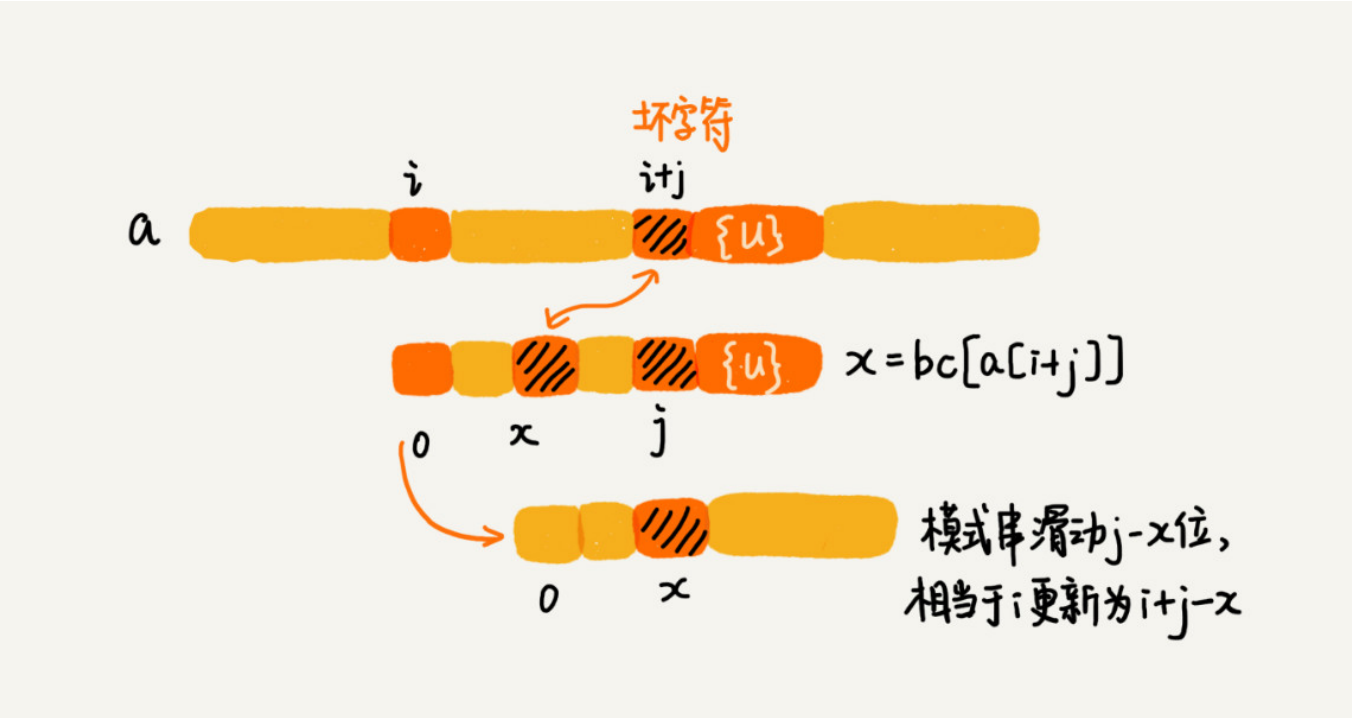
掌握了坏字符规则之后，我们先把 BM 算法代码的大框架写好，先不考虑好后缀规则，仅用坏字符规则，并且不考虑 si-xi 计算得到的移动位数可能会出现负数的情况。

复制代码

```
public int bm(char[] a, int n, char[] b, int m) {
  int[] bc = new int[SIZE]; // 记录模式串中每个字符最后出现的位置
  generateBC(b, m, bc); // 构建坏字符哈希表
  int i = 0; // i 表示主串与模式串对齐的第一个字符
  while (i <= n - m) {
    int j;
    for (j = m - 1; j >= 0; --j) { // 模式串从后往前匹配
      if (a[i+j] != b[j]) break; // 坏字符对应模式串中的下标是 j
    }
    if (j < 0) {
      return i; // 匹配成功，返回主串与模式串第一个匹配的字符的位置
    }
    // 这里等同于将模式串往后滑动 j-bc[(int)a[i+j]] 位
    i = i + (j - bc[(int)a[i+j]]);
  }
```

```
16    return -1;
17  }
```

代码里的注释已经很详细了，我就不再赘述了。不过，为了你方便理解，我画了一张图，将其中的一些关键变量标注在上面了，结合着图，代码应该更好理解。



至此，我们已经实现了包含坏字符规则的框架代码，只剩下往框架代码中填充好后缀规则了。现在，我们就来看看，如何实现好后缀规则。它的实现要比坏字符规则复杂一些。

在讲实现之前，我们先简单回顾一下，前面讲过好后缀的处理规则中最核心的内容：

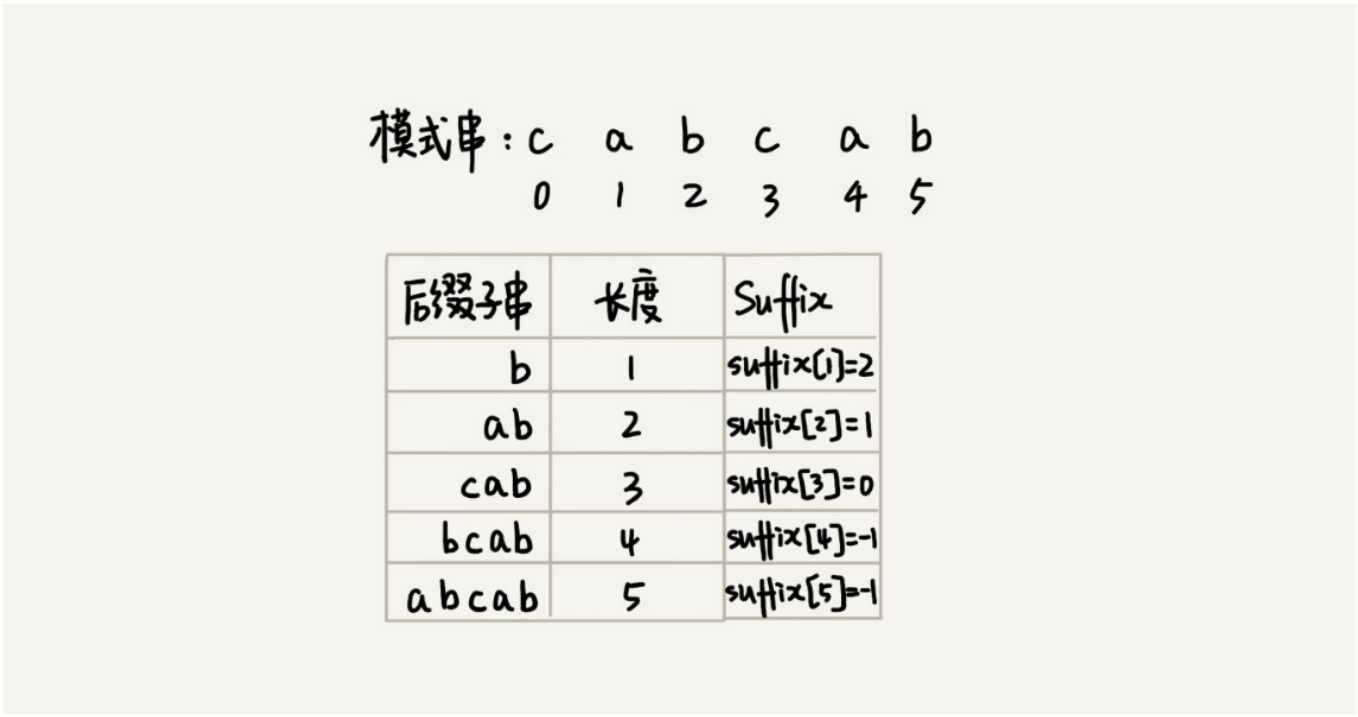
- 在模式串中，查找跟好后缀匹配的另一个子串；
- 在好后缀的后缀子串中，查找最长的、能跟模式串前缀子串匹配的后缀子串；

在不考虑效率的情况下，这两个操作都可以用很“暴力”的匹配查找方式解决。但是，如果想要 BM 算法的效率很高，这部分就不能太低效。如何来做呢？

因为好后缀也是模式串本身的后缀子串，所以，我们可以在模式串和主串正式匹配之前，通过预处理模式串，预先计算好模式串的每个后缀子串，对应的另一个可匹配子串的位置。这个预处理过程比较有技巧，很不好懂，应该是这节最难懂的内容了，你要认真多读几遍。

我们先来看，**如何表示模式串中不同的后缀子串呢？**因为后缀子串的最后一个字符的位置是固定的，下标为 m-1，我们只需要记录长度就可以了。通过长度，我们可以确定一个唯一的后缀子串。

现在，我们要**引入最关键的变量 suffix 数组**。suffix 数组的下标 k，表示后缀子串的长度，下标对应的数组值存储的是，在模式串中跟好后缀{u}相匹配的子串{u*}的起始下标值。这句话不好理解，我举一个例子。



但是，如果模式串中有多个（大于 1 个）子串跟后缀子串{u}匹配，那 suffix 数组中该存储哪一个子串的起始位置呢？为了避免模式串往后滑动得过头了，我们肯定要存储模式串中最靠后的那个子串的起始位置，也就是下标最大的那个子串的起始位置。不过，这样处理就足够了吗？

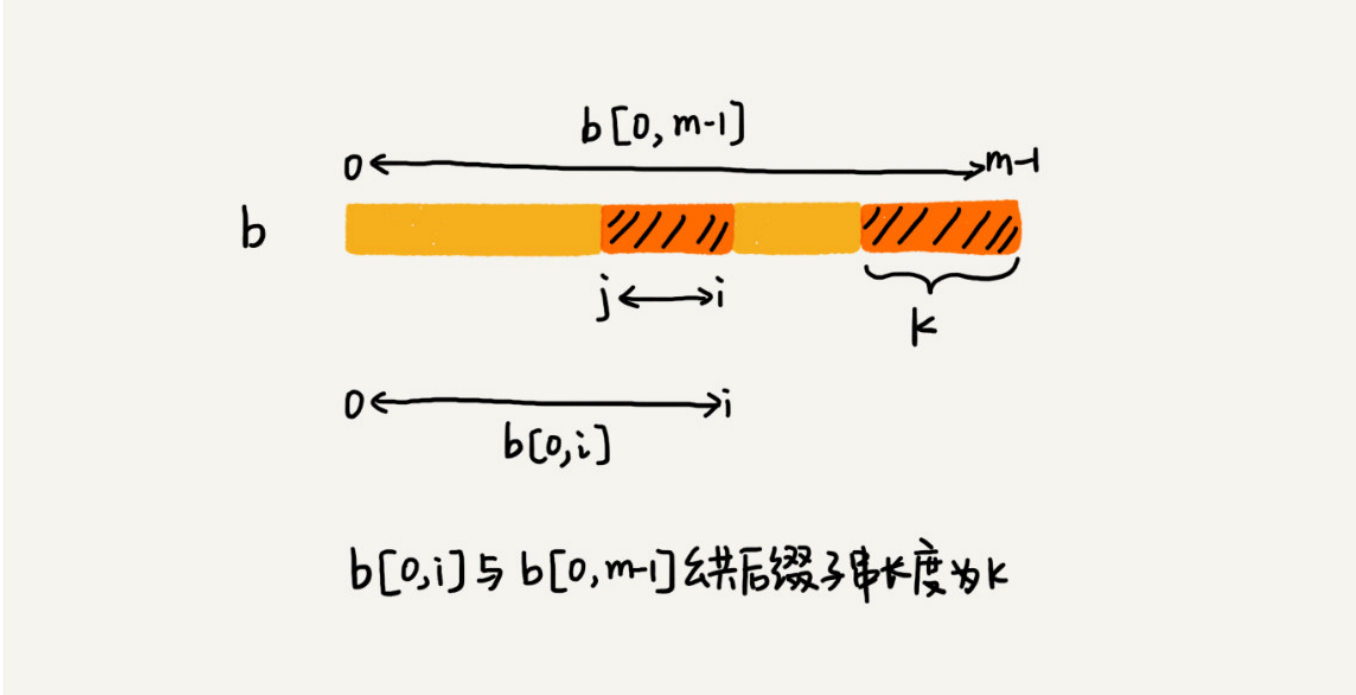
实际上，仅仅是选最靠后的子串片段来存储是不够的。我们再回忆一下好后缀规则。

我们不仅要在模式串中，查找跟好后缀匹配的另一个子串，还要在好后缀的后缀子串中，查找最长的能跟模式串前缀子串匹配的后缀子串。

如果我们只记录刚刚定义的 suffix，实际上，只能处理规则的前半部分，也就是，在模式串中，查找跟好后缀匹配的另一个子串。所以，除了 suffix 数组之外，我们还需要另外一个 boolean 类型的 prefix 数组，来记录模式串的后缀子串是否能匹配模式串的前缀子串。

现在，我们来看下，**如何来计算并填充这两个数组的值**？这个计算过程非常巧妙。

我们拿下标从 0 到 i 的子串（i 可以是 0 到 m-2）与整个模式串，求公共后缀子串。如果公共后缀子串的长度是 k，那我们就记录 suffix[k]=j（j 表示公共后缀子串的起始下标）。如果 j 等于 0，也就是说，公共后缀子串也是模式串的前缀子串，我们就记录 prefix[k]=true。



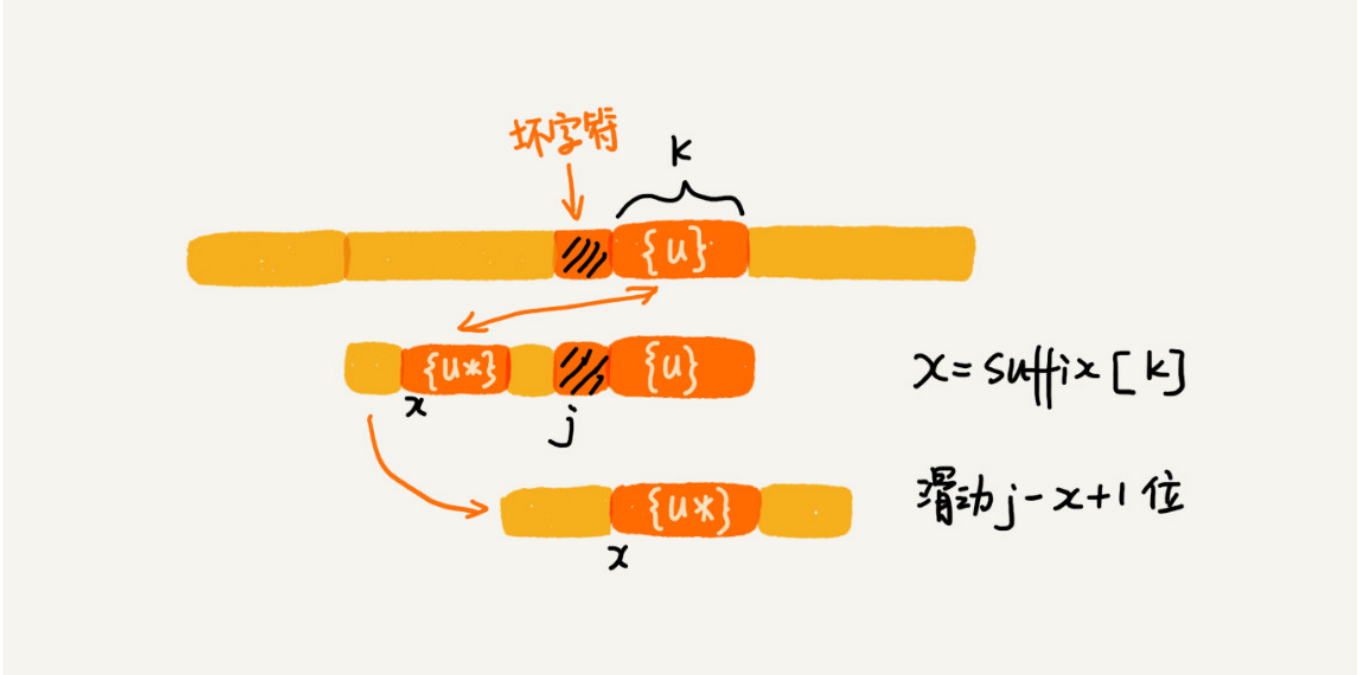
我们把 suffix 数组和 prefix 数组的计算过程，用代码实现出来，就是下面这个样子：

复制代码

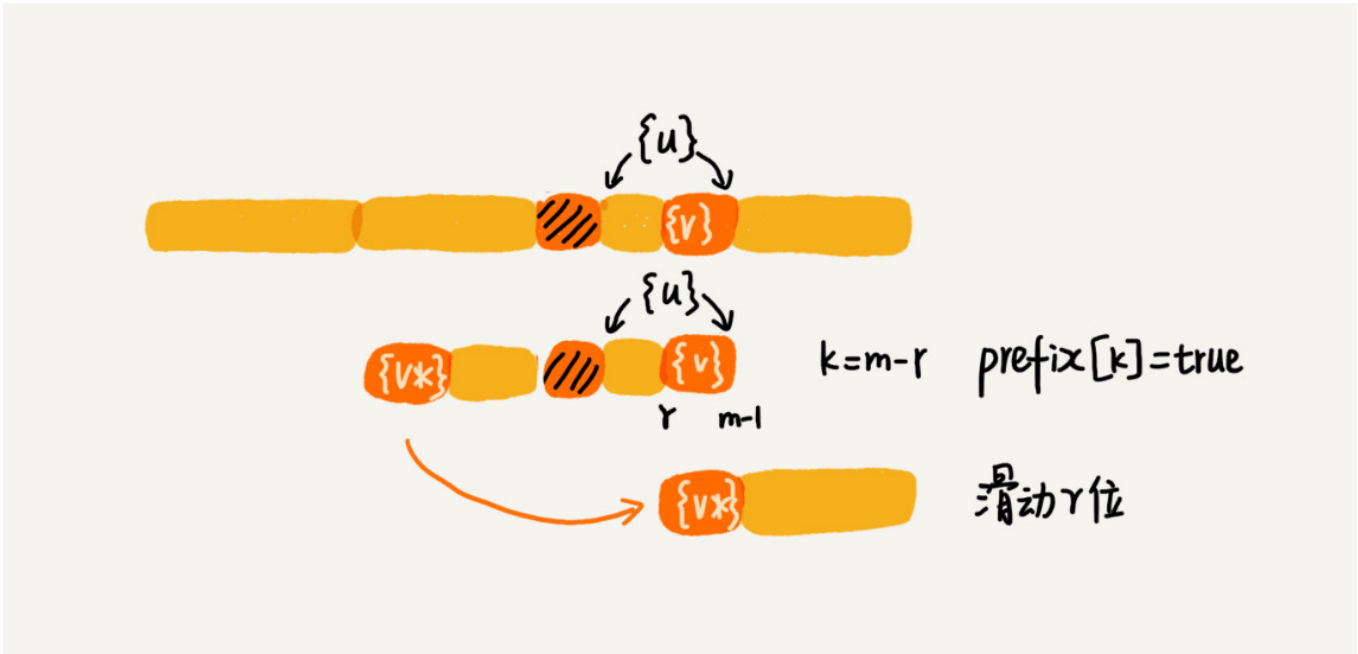
```
// b 表示模式串，m 表示长度，suffix，prefix 数组事先申请好了
private void generateGS(char[] b, int m, int[] suffix, boolean[] prefix) {
  for (int i = 0; i < m; ++i) { // 初始化
    suffix[i] = -1;
    prefix[i] = false;
  }
  for (int i = 0; i < m - 1; ++i) { // b[0, i]
    int j = i;
    int k = 0; // 公共后缀子串长度
    while (j >= 0 && b[j] == b[m-1-k]) { // 与 b[0, m-1] 求公共后缀子串
      --j;
      ++k;
      suffix[k] = j+1; //j+1 表示公共后缀子串在 b[0, i] 中的起始下标
    }
    i
    if (j == -1) prefix[k] = true; // 如果公共后缀子串也是模式串的前缀子串
  }
}
```

有了这两个数组之后，我们现在来看，**在模式串跟主串匹配的过程中，遇到不能匹配的字符时，如何根据好后缀规则，计算模式串往后滑动的位数？**

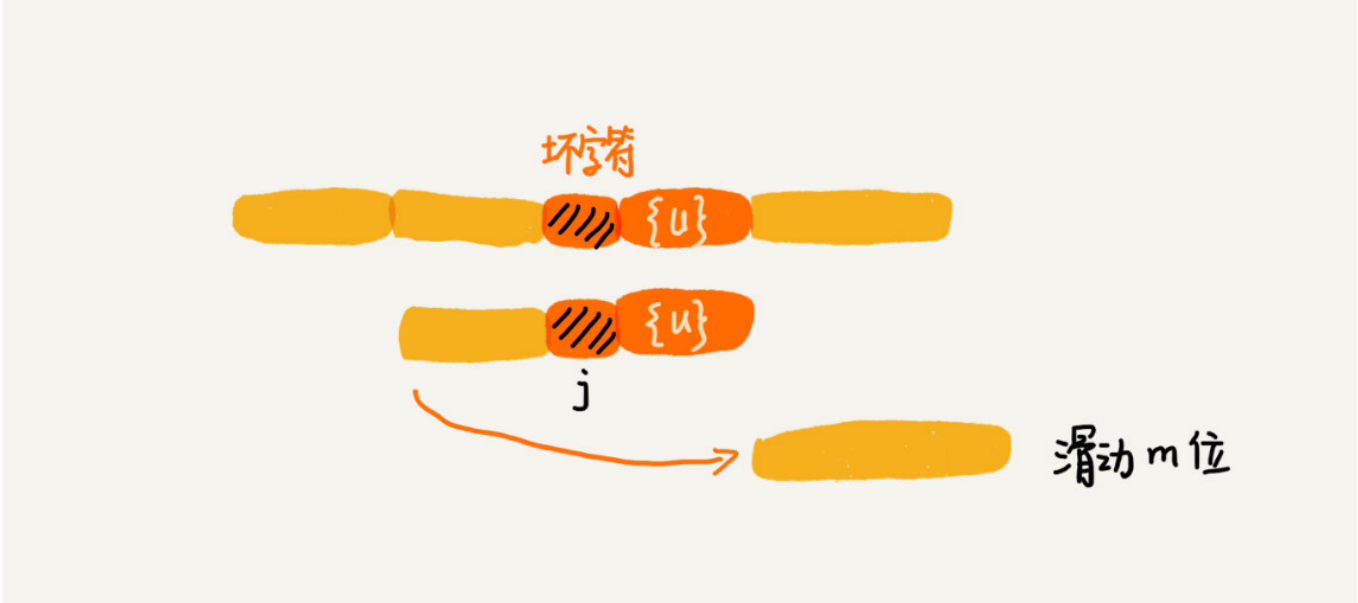
假设好后缀的长度是 k。我们先拿好后缀，在 suffix 数组中查找其匹配的子串。如果 suffix[k] 不等于 -1（-1 表示不存在匹配的子串），那我们就将模式串往后移动 j-suffix[k]+1 位（j 表示坏字符对应的模式串中的字符下标）。如果 suffix[k] 等于 -1，表示模式串中不存在另一个跟好后缀匹配的子串片段。我们可以用下面这条规则来处理。



好后缀的后缀子串 b[r, m–1]（其中，r 取值从 j+2 到 m–1）的长度 k=m–r，如果 prefix[k] 等于 true，表示长度为 k 的后缀子串，有可匹配的前缀子串，这样我们可以把模式串后移 r 位。



如果两条规则都没有找到可以匹配好后缀及其后缀子串的子串，我们就将整个模式串后移 m 位。



至此，好后缀规则的代码实现我们也讲完了。我们把好后缀规则加到前面的代码框架里，就可以得到 BM 算法的完整版代码实现。

复制代码

```
// a,b 表示主串和模式串；n，m 表示主串和模式串的长度。
public int bm(char[] a, int n, char[] b, int m) {
  int[] bc = new int[SIZE]; // 记录模式串中每个字符最后出现的位置
  generateBC(b, m, bc); // 构建坏字符哈希表
  int[] suffix = new int[m];
  boolean[] prefix = new boolean[m];
  generateGS(b, m, suffix, prefix);
  int i = 0; // j 表示主串与模式串匹配的第一个字符
  while (i <= n - m) {
    int j;
    for (j = m - 1; j >= 0; --j) { // 模式串从后往前匹配
      if (a[i+j] != b[j]) break; // 坏字符对应模式串中的下标是 j
    }
    if (j < 0) {
      return i; // 匹配成功，返回主串与模式串第一个匹配的字符的位置
    }
    int x = j - bc[(int)a[i+j]];
    int y = 0;
    if (j < m-1) { // 如果有好后缀的话
      y = moveByGS(j, m, suffix, prefix);
    }
    i = i + Math.max(x, y);
  }
  return -1;
}

// j 表示坏字符对应的模式串中的字符下标 ; m 表示模式串长度
private int moveByGS(int j, int m, int[] suffix, boolean[] prefix) {
  int k = m - 1 - j; // 好后缀长度
  if (suffix[k] != -1) return j - suffix[k] +1;
  for (int r = j+2; r <= m-1; ++r) {
    if (prefix[m-r] == true) {
```

```
33        return r;
34      }
35    }
36    return m;
37  }
```

BM 算法的性能分析及优化

我们先来分析 BM 算法的内存消耗。整个算法用到了额外的 3 个数组，其中 bc 数组的大小跟字符集大小有关，suffix 数组和 prefix 数组的大小跟模式串长度 m 有关。

如果我们处理字符集很大的字符串匹配问题，bc 数组对内存的消耗就会比较多。因为好后缀和坏字符规则是独立的，如果我们运行的环境对内存要求苛刻，可以只使用好后缀规则，不使用坏字符规则，这样就可以避免 bc 数组过多的内存消耗。不过，单纯使用好后缀规则的 BM 算法效率就会下降一些了。

对于执行效率来说，我们可以先从时间复杂度的角度来分析。

实际上，我前面讲的 BM 算法是个初级版本。为了让你能更容易理解，有些复杂的优化我没有讲。基于我目前讲的这个版本，在极端情况下，预处理计算 suffix 数组、prefix 数组的性能会比较差。

比如模式串是 aaaaaaa 这种包含很多重复的字符的模式串，预处理的时间复杂度就是 O(m^2)。当然，大部分情况下，时间复杂度不会这么差。关于如何优化这种极端情况下的时间复杂度退化，如果感兴趣，你可以自己研究一下。

实际上，BM 算法的时间复杂度分析起来是非常复杂，这篇论文“A new proof of the linearity of the Boyer–Moore string searching algorithm”证明了在最坏情况下，BM 算法的比较次数上限是 5n。这篇论文“Tight bounds on the complexity of the Boyer–Moore string matching algorithm”证明了在最坏情况下，BM 算法的比较次数上限是 3n。你可以自己阅读看看。

解答开篇 & 内容小结

今天，我们讲了一种比较复杂的字符串匹配算法，BM 算法。尽管复杂、难懂，但匹配的效率却很高，在实际的软件开发中，特别是一些文本编辑器中，应用比较多。如果一遍看不懂的话，你就多看几遍。

BM 算法核心思想是，利用模式串本身的特点，在模式串中某个字符与主串不能匹配的时候，将模式串往后多滑动几位，以此来减少不必要的字符比较，提高匹配的效率。BM 算法构建的规则有两类，坏字符规则和好后缀规则。好后缀规则可以独立于坏字符规则使用。因为坏字符规则的实现比较耗内存，为了节省内存，我们可以只用好后缀规则来实现 BM 算法。

课后思考

你熟悉的编程语言中的查找函数，或者工具、软件中的查找功能，都是用了哪种字符串匹配算法呢？

欢迎留言和我分享，也欢迎点击“请朋友读”，把今天的内容分享给你的好友，和他一起讨论、学习。

上一篇 32 | 字符串匹配基础（上）：如何借助哈希算法实现高效字符串匹配?

下一篇 34 | 字符串匹配基础（下）：如何借助BM算法轻松理解KMP算法?

写留言

精选留言

Liam 5

好后缀原则下，最后一种情况为什么移到坏字符后面呢，不能移到好后缀的后面吗？即m+1，而不是j + 1

2018-12-07

作者回复

你说的对 👍 我改下

2018-12-07

seniusen 3

好后缀原则中，最后一种情况，应该是移动 m 位吧，移动整个模式串的长度。

2018-12-07

作者回复

是的

2018-12-10

Liam 3

好后缀原则中，最后一种情况，为什么是移动j + 1 位，而不是m+1位

2018-12-07

作者回复

移动到坏字符后面 移动m+1位是怎么理解的呢

2018-12-07

五岳寻仙 3

老师好！今天讲的BM算法确实有点复杂，不过听的时候有熟悉的感觉，似乎跟之前接触过的Boyer Moore算法很像，查了一下才发现原来是同一种算法😂

在工作中遇到过这样的情况，需要在一个长度为n (比如十亿级)的巨大的主串中查找长度为m(比如几百)的模式串。主串是固定的，从直观上讲，要加快搜索速度，就需要对主串建索引。BWT-FM算法是解决这类问题最经典的算法，刚接触时也是不好理解，但感觉非常神奇，可以将搜索的时间复杂度降到O(m)，是我认为最伟大的算法之一。

2018-12-07

cygnus 2

generateGS函数里suffix和prefix的赋值应该放到while循环内，即每次k变动时都要赋值。
另外请问下：好后缀的后缀子串 b[r, m-1]，这里的r的初值j+2是怎么得来的啊?

2018-12-08

作者回复

j表示坏字符的下标 好狗追其实下标j+1

2018-12-10

Jerry银银 2

曾经一度觉得字符串匹配的几大算法，都是高山仰止的，难以理解。

但是前阵子受两句话启发，从此以后对字符串匹配问题，至少在战略层面藐视了它：
1. 善用之前信息(从信息论的角度：消除信息的不确定性，就是引入信息)

2. 增加效率，在资源有限的情况下，只有想办法少做事情

2018-12-07

P@tricK 2

老师，suffix和prefix的赋值那里有BUG，应该在每一次k的变动都要有suffix赋值。

2018-12-07

作者回复

是的 多谢

2018-12-10

meng 1

我对这次课的内容一知半解，于是在网上搜到一个文档，里面的图挺好的，跟大家分享一下：http://www.cs.jhu.edu/~langmea/resources/lecture_notes/boyer_moore.pdf

2018-12-09

Smallfly 1

BM 算法分析着实比较复杂，不过按照老师的思路，一步一步走，看懂应该没问题的。但其实有些代码实现细节看不懂关系也不大。我们学算法主要目的是学习算法的思想，能在需要的时候加以应用就好。

但对于平时工作，几乎不可能遇到，需要自己手写一个字符串匹配算法的场景。那我们还要学，图的是什么？

我认为文章中值得学习借鉴的思想有：

1、要有优化意识，前面的 BF，RK 算法已经能够满足我们需求了，为什么发明 BM 算法？是为了减少时间复杂度，但是带来的弊端是，优化代码变得复杂，维护成本变高。

2、需要查找，需要减少时间复杂度，应该想到什么？散列表。

3、如果某个表达式计算开销比较大，又需要频繁的使用怎么办？预处理，并缓存。

（一点拙见，可能文中还有其它优秀的思想，没能 Get 到）

2018-12-08

作者回复

2018-12-10

weizhe 1

老师，为什么suffix数组的代码实现中只记录了模式串中最长的后缀且在子串出现的情况，而没有记录其他子串{u*}的起始下标值？
（感觉generateGS方法中的每一条while判断成功后都应该在suffix数组中记录下来）

2018-12-08

作者回复

已改正 不好意思

2018-12-10

中午要吃鱼摆摆 1

老师，您好，文中得generateGS函数，似乎不能求得模式串中存在多个好后缀得时候，靠右好后缀的起始下标。

2018-12-08

作者回复

已经改正

2018-12-10

P@tricK 1

高票那个留言，是移动m位，不是m+1位。

这节课细节上小问题有点多，不过瑕不掩瑜，思想重要，细节自己钻研。

2018-12-08

作者回复

是的

2018-12-10

blacknhole 1

有几点疑问：

1，“BM 算法的性能分析及优化”小节中说“suffix 数组的大小跟字符集大小有关”，这是书写错误吗？suffix数组长度应该与字符集大小无关，只跟模式串长度有关。与字符集大小有关的是bc数组。

2，BM 算法的完整版代码实现中的语句for (int r = j+2; r < m-1; ++r) { if (prefix[m-r+1] == true) { return r; } }有误吧？ 应为r < m和prefix[m-r]，即for (int r = j+2; r < m; ++r) { if (prefix[m-r] == true) { return r; } }。

2018-12-07

作者回复

1 笔误 是bc数组

2 我改下 多谢指正

2018-12-07

杨伟 1

这个算法用的多么？老师为什么讲解这个算法？

2018-12-07

作者回复

怎么讲呢 平常不大可能会自己去实现一个bm算法 顶多就用个bf算法。不过bm算法号称最高效的 比如grep命令就是用它实现的 所以有必要讲一下 不然不完整啊 你全当思维训练吧

2018-12-07

dapaul 1

坏字符规则那，算xi的位置时，应该只从坏字符对齐时模式串往前的字符中匹配，这样就不会出现si-xi为负了

2018-12-07

作者回复

代码实现就难了 也没那么高效了

2018-12-07

深蓝...

1

有点掉队的节奏

2018-12-07

作者回复

正常的。但凡是上难度的 都会有掉队的 就看谁能跟上了

2018-12-07

纯洁的憎恶

0

大体思路应该是看懂了，不过具体实现和代码细节还需要时间消化。BM算法的核心思想是通过将模式串沿着主串大踏步的向后滑动，从而大大减少比较次数，降低时间复杂度。而算法的关键在于如何兼顾步子迈得足够大与无遗漏，同时要尽量提高执行效率。这就需要模式串在向后滑动时，遵守坏字符规则与好后缀规则，同时采用一些技巧。

坏字符规则：从后往前逐位比较模式串与主串的字符，当找到不匹配的坏字符时，记录模式串的下标值si，并找到坏字符在模式串中，位于下标si前的最近位置xi（若无则记为-1），si-xi即为向后滑动距离。（PS：我觉得加上xi必须在si前面，也就是比si小的条件，就不用担心计算出的距离为负了）。但是坏字符规则向后滑动的步幅还不够大，于是需要好后缀规则。

好后缀规则：从后往前逐位比较模式串与主串的字符，当出现坏字符时停止。若存在已匹配成功的子串 {u} ，那么在模式串的 {u} 前面找到最近的 {u} ，记作 {u'} 。再将模式串后移，使得模式串的 {u'} 与主串的 {u} 重叠。若不存在 {u'} ，则直接把模式串移到主串的 {u} 后面。为了没有遗漏，需要找到最长的、能够跟模式串的前缀子串匹配的，好后缀的后缀子串（同时也是模式串的后缀子串）。然后把模式串向右移到其左边界，与这个好后缀的后缀子串在主串中的左边界对齐。

何时使用坏字符规则和好后缀规则呢？首先在每次匹配过程中，一旦发现坏字符，先执行坏字符规则，如果发现存在好后缀，还要执行好后缀规则，并从两者中选择后移距离最大的方案执行。

技巧：
1.通过散列表实现，坏字符在模式串中下标位置的快速查询。
2.每次执行好后缀原则时，都会计算多次能够与模式串前缀子串相匹配的好后缀的最长后缀子串。为了提高效率，可以预先计算模式串的所有后缀子串，在模式串中与之匹配的另一个子串的位置。同时预计算模式串中（同长度的）后缀子串与前缀子串是否匹配并记录。在具体操作中直接使用，大大提高效率。
3.如何快速记录模式串后缀子串匹配的另一个子串位置，以及模式串（相同长度）前缀与后缀子串石否匹配呢？先用一个suffix数组，下标值k为后缀子串的长度，从模式串下标为i（0~m-2）的字符为最后一个字符，查找这个子串是否与后缀子串匹配，若匹配则将子串起始位置的下标值j赋给suffix[k]。若j为0，说明这个匹配子串的起始位置为模式串的起始位置，则用一个数组prefix，将prefix[k]设为true，否则设为false。k从0到m（模式串的长度）于是就得到了模式串所有前缀与后缀子串的匹配情况。

2018-12-10

距离 0

对于还没毕业的我有点坚持不下去了

2018-12-10

编辑回复

再坚持一下

2018-12-10

传说中的成大大 0

在用一个256的数组 用字符的ascii码做下标 记录该字符出现的位置 如果存在相同字符怎么办呢？之前的会被新的覆盖掉的把！

2018-12-10

他在她城断了弦 0

因为根据 si-xi 计算出来的移动位数，有可能是负数，比如主串是 aaaaaaaaaaaaaaa，模式串是 baaa。不但不会向后滑动模式串，还有可能倒退。

这里不太懂，老师能解释下吗？

2018-12-10

纯洁的憎恶 0

太精妙啦～不知道是怎么想出来的

2018-12-10

sarahsnow 0

例子中，prefix数组的值有误吧？

suffix b和ab，可以在模式串cabcab的前缀中找到

应该是：

prefix[1] = true

prefix[2] = true

2018-12-09

作者回复

必须是前缀才是true

2018-12-10

微秒 0

老师，我们从好后缀的后缀子串中，找一个最长的并且能跟模式串的前缀子串匹配的。为什么会是最长呢？万一公共的后缀子串中出现更小的公共子串，这种情况也有可能出现匹配的情况吧。

2018-12-09

ZX 0

用js写了一遍

```
/**
```

```
* BM算法，取好坏部分，进行模式串的移动
* 1.主串从左往右匹配模式串
* 2.模式串从右往左每个字符匹配
* 3.如果出现不匹配的字符，取坏字符规则和好字符规则中较大的值进行移动
* 4.坏字符规则：看不匹配的值在模式串前面是否还有出现过，进行相应的移动
* 5.好字符规则：先看好字符串在前面是否有出现，没有就看好字符的后缀子串是否是模式串的前缀子串，进行相应的移动
* 6.技巧，模式串是固定的，可以预先构建一个查找某个字符在模式串位置的对象(方便后面查找坏字符)，构建一个后缀串是否在模式串出现和后缀串是否是前缀串的数组(方便进行好字符规则进行匹配)
*/

/**
* @param {*} s 模式串
* @param {*} m 主串
*/
function BM(s, m) {
const bc = {}
generateBC(bc, s)
const suffix = [] // 模式串后缀串在模式串前面是否有出现
const prefix = [] // 模式串后缀串是否是模式串前缀串
generateSP(s, suffix, prefix)

const mLength = m.length
const sLength = s.length

let i = sLength - 1
while (i < mLength) {
let j = sLength - 1
while (j >=0 && m[i] === s[j]) {
i--
j--
}
if (j === -1) return i + 1
const badMove = j - bc[m[i]]
const goodMove = moveByGood(s, j, suffix, prefix)
i += Math.max(badMove, goodMove)
}
return false
}

/**
* 计算好后缀规则需要移动的步数
```

```
* @param {} s
* @param {*} j
* @param {*} suffix
* @param {*} prefix
*/
function moveByGood(s, j, suffix, prefix) {
const m = s.length
const k = m - j - 1
if (typeof suffix[k] !== 'undefined') return j - suffix[k]
for (let i = k; i > 0 ; i--) {
if (prefix[i] === true) return m - i
}
return m
}

/**
* 构建坏字符
* @param {*} bc
* @param {*} s
*/
function generateBC(bc, s) {
const length = s.length
for(let i = 0; i< length - 1; i++) {
bc[s[i]] = i
}
}

/**
* 构建模式串后缀和前缀的数组
* @param {*} s
* @param {*} suffix
* @param {*} prefix
*/
function generateSP(s, suffix, prefix) {
const length = s.length
for(let i = 0; i < length - 1; i++) {
let j = i
let k = 0
while(j >= 0 && s[j] === s[length - k - 1]) {
j--
k++
}
if (k !== 0) suffix[k] = j
```

```
if (j === -1) prefix[k] = true
}
}
```

2018-12-08

作者回复

👍

2018-12-10

DADDYHINS 0

这 倒是能看懂，但是代码实现自己写还是一脑袋浆糊

2018-12-08

作者回复

看懂就行了

2018-12-10

nopsky 0

讲shuffix的第一个图中shuffix[4] = -1，这个-1怎么来的，不能理解，能不能再讲一下

2018-12-07

喵吉豆豆 0

如果我们处理字符集很大的字符串匹配问题，suffix 数组对内存的消耗就会比较多。因为好后缀和坏字符规则是独立的，如果我们运行的环境对内存要求苛刻，可以只使用好后缀规则，不使用坏字符规则，这样就可以避免 suffix 数组过多的内存消耗。不过，单纯使用好后缀规则的 BM 算法效率就会下降一些了。
这里应该是bc数组吧

2018-12-07

作者回复

是的 已经改正

2018-12-10

🐱您的好友William🐱 0

懂了是懂了，但是你让我自己实现是不可能实现的，这辈子。。。。之后有可能实现。。。。
其实精髓都在最后那三张移动的图里，记住两原则取最大的，好后缀按“suffix”，“prefix”，“都没对上”，三个顺序输出。其中一旦在原则中出现了匹配到多次的情况，都按最保守最接近右侧的取。

2018-12-07

作者回复

看懂就够了 能看懂我就没白画这文章中的22张图

2018-12-10

拉欧 0

花了一下午时间全弄明白了，感觉字符串匹配算法方面精进不少

2018-12-07

作者回复

👍

2018-12-10

吴月月鸟 0

0 1 2 3 4 5 0 1 2 3 4 5

c a b c a b c a b c a b

b c

a b c a

c a b c a b

b c a b c a b c

a b c a b c a b c a

这是我在excel中画的suffix和prefix值的图，重点要理解suffix和prefix公式的含义，后续的内容才好理解，希望能帮助大家方便理解。

2018-12-07

槛外人 0

最后为什么是从这两个移动位数中取最大值而不是最小值来避免错过了?

2018-12-07

作者回复

两个算法是独立的 坏字符移动3 是对的 好后缀移动5也是对的 那我们就选最大的移动

2018-12-10

walor 0

BM 算法的性能分析及优化

我们先来分析 BM 算法的内存消耗。整个算法用到了额外的 2 个数组，其中 suffix 数组的大小跟

字符集大小有关，prefix 数组的大小跟模式串长度 m 有关。

问题：

这里说的 suffix 数组，应该是 bc 数组吧

2018-12-07

作者回复

是的 已经改正

2018-12-10

sherry 0

上面有个同学提到，“坏字符规则计算xi的位置时，从坏字符对齐时模式串往前的字符中匹配，这样就不会出现si-xi为负”。我觉得这样是可行的，并且这样操作之后只使用坏字符规则就能完成字符串的匹配了，但是效率不一定会更高。因为老师在讲解代码实现时提到：为了更高效地查找坏字符的xi，所以使用了散列表，从例子中可以看出，相同字符在散列之后前面的就被覆盖掉了（只留下相同字符的最后位置在散列表中，数组的下标为字符的ASCII码，值为

该字符出现的最后位置），这样获取xi简单有效。如果从对齐位置往前查找模式串中是否有该字符，那么要么使用顺序遍历查找这种低效的方法，要么就要解决散列冲突，所以这种方法不一定会更好。

另外我的问题是老师给了两篇文章的链接，证明了在最坏情况下BM的比较次数上限分别为5n和3n，这两个证明都是正确的吗？还没有读论文，但是第一感觉看起来这句话是矛盾的呀，对于确定问题怎么会有不同结果呢？

2018-12-07

作者回复

不矛盾 比如有10个人 我量了一下这10人的身高 说身高上限是1.8m。你又量了一遍说是1.79m。我俩都没错 你更精确

2018-12-10

任雪龙 0

坏字节匹配时，构造的 bc 数组这里是不是有问题，模式串中出现两个 a 的情况下，数组下标为 97 的地方值只会保存模式串中最后一个 a 字符在串中的位置，这样出现坏字符时怎么能确定上一个 a 的位置呢？

2018-12-07

作者回复

就是只保存最后一个 再看看文章

2018-12-07

MSN 0

suffix数组的下标是后缀字符的长度，那不应该和模式串的长度m有关系吗，跟字符集有什么关系？希望老师解答

2018-12-07

作者回复

是bc 已经改正

2018-12-10

fumeck.com🍋🌴summer sk... 0

看得有点懵逼哎

2018-12-07

作者回复

那就多看几遍啊 内容的难度摆在那里 要么多思考搞定它 要么就放弃这篇 直接看后面的 其他没啥办法了

2018-12-07