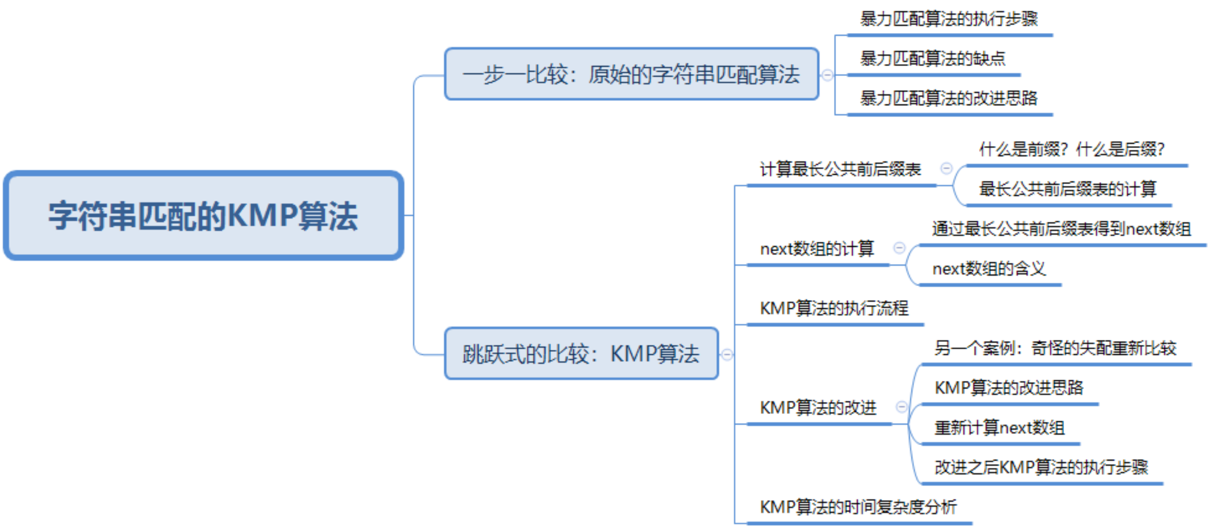

1.一步一比较：原始的字符串匹配算法

字符串比较算法，在真实开发过程中非常常见。例如，在String类当中提供的indexOf()方法，就是一个典型的字符串比较算法的实现。

字符串比较算法的含义就是：在一个较长的字符串当中，判断是否存在一个较短的字符串；如果存在，则返回这个较短字符串在较长字符串中起始下标；如果不存在，则返回-1

通常来讲，我们将较长的字符串称之为文本串S，而将较短的，要在文本串中进行查询的字符串称之为模式串P

如果仅仅从思路简单的角度出发，我们可以总结出一种极其容易理解，但是效率很低的字符串匹配算法：暴力匹配法

下面我们就从字符串的暴力匹配算法出发，一步一步的总结暴力匹配算法的执行流程以及不足

最终总结出KMP匹配算法的相关内容

①暴力匹配算法的执行步骤

暴力匹配算法的执行非常简单，下面我们以图示的方式，来演示一下暴力匹配算法的执行流程

步骤1：将文本串S和模式串P的首部（字符数组下标为0的位置）对其，并分别为文本串S和模式串P分配一个字符下标指针i和j，

分别指向文本串S和模式串P的下标为0的位置

```
         i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
         j
```

步骤2：如果文本串S的第i位上的字符和模式串P的第j位上的字符相等，即S[i] == P[j]的情况

则模式串P相对文本串S不发生位移，下标i和j分别向下移动一位，即i++，j++，并且继续比较

```
         i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
         j
```

```
           i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
           j
```

```
             i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
             j
```

步骤3：如果文本串S的第i位上的字符和模式串P的第j位上的字符不相等，即S[i] != P[j]的情况

则模式串P相对于文本串S向后移动一位，并重置j = 0，i与j对齐，并重复步骤2-3的比较过程

```
                i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P: A B C D A B D
                j
```

```
                  i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P: A B C D A B D
                  j
```

```
              i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:   A B C D A B D
              j
```

直到找到模式串P在文本串S中的完全匹配情况下，i的取值；或者确定文本串S中一定不包含模式串P的情况

（如果在匹配过程中发现文本串S已经匹配完毕，但是模式串P中尚有元素没有进行匹配，那么就可以判定在文本串S中不存在模式串P）

②暴力匹配算法的缺点

通过研究上述暴力匹配算法的执行流程，暴力匹配算法的缺点也是一目了然的：匹配效率太低！

从上面的案例中我们不难看出：在模式串P当中，存在着一些重复的部分（例如上图中模式串P中两次出现AB这个子串）

模式串P：A B C D A B D

但是在暴力匹配算法中，我们并没有很好的利用这一特性，而是在某一位的字符匹配失败之后，一切从头开始

那么，我们如何利用模式串P中存在重复部分的这一特点呢？下面我们就来讨论一下

③暴力匹配算法的改进思路

在上面的章节我们已经说过，如果模式串P中存在重复的部分，我们应该尽可能的利用这一点

也就是说：如果在模式串P与文本串S匹配成功的部分当中，模式串P存在重复的部分，那么我们在进行模式串P相对于文本串S进行位移的时候

不应该每次仅移动1位，而是应该将第一次出现的重复位置，与第二次出现的重复部分相重合，并且从重合之后的部分开始进行比较（如下图所示）

```
                    i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：        A B C D A B D
                    j

                      i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：        A B C D A B D
                      j

                        i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：        A B C D A B D
                        j

                        i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：                A B C D A B D
                        j
```

这样一来有两个好处：

好处1：可以避开文本串S中，中间那一部分“一定不能够匹配”的字符串部分

好处2：重复的部分重叠，相当于重复的部分在跳转之后已经“比较”过，不需要重新比较这一部分字符串

如果你能够意识到这一点，那么说明你已经认识到了接下来我们要学习的KMP算法的精髓！

仅凭肉眼的判断我们很容易实现这个思想，但是，我们又如何通过计算机程序，有规律的实现上述思想呢？

接下来就让我们一起研究KMP算法

2.跳跃式的比较：KMP算法

KMP算法的核心思想，我们在上面分析暴力匹配算法的时候已经分析过，那就是：

如果在模式串P中存在前后重复的部分，那么就在匹配失败的时候，将模式串P中前面重复的部分与后面重复的部分重合，然后从重合的部分后面，重新开始比较

如果我们想要通过代码实现这一过程，那么需要经过如下三个步骤：1.计算最长公共前后缀表；2.通过最长公共前后缀表得到next数组；3.根据next数组实现KMP算法

那么接下来，就让我们按照这个步骤，逐步实现整个KMP算法

①计算最长公共前后缀表

1.什么是前缀？什么是后缀？

在计算模式串P的前缀和后缀的过程中，我们需要将模式串P打碎成为单个字符，然后再将这些字符按照在原来模式串P中的“出场顺序”一个一个的拼回去

并且在拼接过程中，分别计算这些子串的前后缀有哪些

以上面案例中出现的模式串P=ABCDABD为例，我们需要分别计算其子串A、AB、ABC、ABCD、ABCDA、ABCDAB和ABCDABD的前缀和后缀

那么，这些子串的前缀和后缀又是如何定义的呢？

一个子串的前缀：从一个子串的第一个字符开始，按照顺序，每添加一个后序字符，就构成这个子串的一个前缀。注意：前缀不包含当前子串本身

例如：子串ABCD的前缀就是：A、AB、ABC

一个子串的后缀：从一个子串的最后一个字符开始，按照逆序，每添加一个前序字符，就构成这个子串的一个后缀。注意：后缀同样不包含当前子串本身

例如：子串ABCD的后缀就是：D、CD、BCD

2.最长公共前后缀表的计算

根据上述对于子串前后缀的定义，我们可以得到如下一张关于模式串P=ABCDABD的情况下，各个子串的前缀和后缀的表格

子串	所有前缀	所有后缀
A	无	无
AB	A	B
ABC	A、AB	C、BC
ABCD	A、AB、ABC	D、CD、BCD
ABCDA	A、AB、ABC、ABCD	A、DA、CDA、BCDA
ABCDAB	A、AB、ABC、ABCD、ABCDA	B、AB、DAB、CDAB、BCDAB
ABCDABD	A、AB、ABC、ABCD、ABCDA、ABCDAB	D、BD、ABD、DABD、CDABD、BCDABD

我们发现：在上述表格中，某些子串的前后缀之中，存在着一些重复值

找到这些重复值很重要，因为下一步我们就要根据这些子串前后缀的重复值，来计算最长公共前后缀表

最长公共前后缀表的计算方式更简单：当子串中纳入一个新字符之后，其前缀和和后缀中，相等的前缀和后缀的最长长度是几，那么这个新纳入的字符的对应位置就取几

如果在纳入这个字符之后，子串的前后缀之间没有相同的取值，那么这个字符对应的位置取值为0

我们将一个子串所有前后缀中，相同取值的最长者，称之为最长公共前后缀；将根据最长公共前后缀长度得到的表格，称之为最长公共前后缀表

百说不如一见，根据上面的理论，我们得到了如下的最长公共前后缀表：

字符下标	0	1	2	3	4	5	6
纳入字符	A	B	C	D	A	B	D
最长公共前后缀长度	0	0	0	0	1	2	0

②next数组的计算

1.通过最长公共前后缀表得到next数组

在计算出最长公共前后缀表之后，我们就能够根据这张表，得到一个名为next数组的结构

next数组的计算方式为：

步骤1：将最长公共前后缀表中的最长公共前后缀长度整体右移1位

下标	0	1	2	3	4	5	6
字符	A	B	C	D	A	B	D
取值		0	0	0	0	1	2

步骤2：将第一个字符的对应的取值置为-1

下标	0	1	2	3	4	5	6
字符	A	B	C	D	A	B	D
取值	-1	0	0	0	0	1	2

这样一来，next数组就计算完毕了

看到这里，各位小伙伴可能已经彻底蒙圈了：这个next数组到底是干什么的？表格中每一个字符对应的取值又表示什么含义？

下面就让我们一起来分析一下next数组的具体含义

2.next数组的含义

实际上我们可以将上面的这个next数组看做一个“匹配失败跳转下标数组”。也就是说，当S[i]与P[j]失配，即S[i] != P[j]的时候，我们只要保持i不变，并且让P[next[j]]与S[i]对齐即可

但是在next数组中存在-1，如果我们在模式串P中取P[-1]显然是不合适的

所以，如果next[j] = -1，那么此时说明文本串S与模式串P在一开始就是失配的，所以此时我们让指向文本串S的下标i自增1，即i++，并且指向模式串P的开头处即可

根据上面的说法我们可以看出：在文本串S和模式串P失配的情况下，在对模式串P进行跳转的时候，我们始终都是在参考next数组中的取值。

下面就让我们使用上述案例，完整的讨论一下KMP算法的整体执行流程

③KMP算法的执行流程

步骤1：与暴力匹配算法的第1步相同，将文本串S与模式串P头部对齐，并且使用下标i与下标j分别指向文本串S与模式串P下标为0的位置

```
        i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
        j
```

步骤2：开始执行比较，如果文本串S能够与模式串P相匹配，即S[i] == P[j]的情况，则i++，j++，继续执行比较

```
        i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
        j

          i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
          j

            i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
            j
```

步骤3：如果文本串S与模式串P失配，即S[i] != P[j]的情况，则需要观察next数组的取值：如果next[j] != -1，则i不变，j = next[j]，即使用S[i]与P[next[j]]重新进行比较

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D 此时next[j]即next[3]=0
j

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D i不变，j取值next[3]，即j = 0，S[i]与P[j]重新开始比较
j

步骤4：如果next[j] == -1，则表示S[i]和P[j]的位置一开始就是失配的，所以可以直接略过，此时置i++，j = 0，重新开始匹配

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
j 此时next[j]即next[0]取值为-1，则i++，j = 0，S[i]与P[j]重新开始比较

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
j S[i]与P[j]继续失配，且next[j] == next[0] == -1，重复上面的步骤

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
j

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
j

i
文本串S：A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P：A B C D A B D
j

步骤5：重复执行上述步骤2-4，直到确定文本串S中不包含模式串P，或者找到模式串P在文本串S完全匹配的部分为止

```
                               i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:                 A B C D A B D
                               j

                                 i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:                 A B C D A B D
                                 j
```

......

```
                                           i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:                 A B C D A B D
                                           j 此时next[j] == next[6] == 2，所以i保持不变，置j = 2
                                           i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:                         A B C D A B D
                                           j  S[i]与P[j]对齐进行比较
```

......

```
                                                           i
文本串S: A B C _ E D C _ A B C D A B E _ A B C D A B D _ C C A D
模式串P:                                 A B C D A B D
                                                           j 匹配成功
```

到此为止，KMP算法的执行步骤就全部演示完毕了。正如上文所说的，整个KMP算法的执行步骤是仅仅围绕着next数组进行的

但是，到目前为止我们所掌握的KMP算法并不是完美的KMP算法，也就是说，KMP算法还有进一步的改进空间

下面就让我们一起找出KMP算法的不足之处，并且加以调整

④KMP算法的改进

在之前的章节中，我们已经详细的研究了KMP算法的执行流程和其中最为重要的next数组的由来

但是，上面的KMP算法并不是完美的，在一些特殊情况下，上面的KMP算法会展示出一些奇怪的特性，从而使得字符串的匹配效率下降

下面我们就来一起看一下另一个案例，并就此引发对KMP算法改进的思考

1.另一个案例：奇怪的失配重新比较

给定文本串S=ABCD_ABACAE_AAABABAC以及模式串P=ABAB

首先让我们来计算模式串P对应的next数组取值：

下标	0	1	2	3

字符	A	B	A	B
取值	-1	0	0	1

当我们使用上面的next堆文本串S进行模式串P的KMP算法匹配的时候，在执行到下列步骤时，会产生如下的一种“奇怪”的失配情况

```
              i
文本串S: A B C D _ A B A C A E _ A A A B A B A C
模式串P:          A B A B
                  j

                    i
文本串S: A B C D _ A B A C A E _ A A A B A B A C
模式串P:          A B A B
                    j

                      i
文本串S: A B C D _ A B A C A E _ A A A B A B A C
模式串P:          A B A B
                      j

                        i
文本串S: A B C D _ A B A C A E _ A A A B A B A C
模式串P:          A B A B C与B失配，此时next[3] == 1，保持i取值不变，j = next[3] = 1
                        j

                        i
文本串S: A B C D _ A B A C A E _ A A A B A B A C
模式串P:              A B A B C与B在初始状态下即为失配状态
                        j
```

正如上图所示：当执行到第一次出现C与B失配的情况时，因为此时的j取值为3，所以在跳转的过程当中，j = next[j] = next[3] = 1

也就导致了在下标i不变的情况下，下标j的取值变为1，并使用S[i]与P[j]继续进行比较

但是从图中我们不难发现P[3]取值为B，已经与文本串中的S[i]（取值为C）相失配，而P[1]的取值还是B，

那么就会导致在模式串P和下标j根据next数组的区之间进行跳转之后，一开始比较就处于失配状态，这种比较根本没有意义，并且还会导致匹配效率的下降！

那么，我们是否能够通过程序进行判断，将这些通过肉眼能够明确判断，一定不相匹配的重复判断部分排除掉，从而提升整体的比较效率呢？

2.KMP算法的改进思路

根据上文中的思路，当第一次出现文本串S中的C与模式串P中的B失配的情况之后，如果我们进行模式串P的跳转，就应该避免“在同一个位置连续失配两次”的情况发生

也就是说：如果明知跳转之后，S[i]位置上的C还要与P[j]上的B再次比较并且再次失配，那就不如直接绕过这种情况，执行i不变，j = 0，然后再去进行比较，也就是：

优化之前的情况：

```
                              i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：           A B A B
                              j
```

```
                              i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：               A B A B
                              j
```

优化之后的情况：

```
                              i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：           A B A B
                              j
```

```
                              i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：                 A B A B
                              j
```

通过上面的图示我们看到，在优化之后，我们成功的避免了出现两次S[i]=C与P[j]=B相比较的情况

这是因为我们明知S[i]=C与P[j]=B是失配的，并且这种情况一定可以确定模式串P在文本串S的这个位置不会完整适配

所以我们绕过了这一次没有意义的比较，从而提升了算法整体的效率

但是上面都是我们根据经验和肉眼判断得出的结论，那么通过程序是如何实现上述的优化的呢？这就需要我们重新计算next数组了

3.重新计算next数组

从本案例出发，当我们知道文本串S中的S[i]=C与模式串P中的P[j]=B是失配的情况下，如果在执行j = next[j]之后我们发现：P[j] == P[next[j]] == B

那么我们就可以马上判断出：如果执行跳转，跳转之后依然是使用S[i]=C与P[j]=B进行比较，我们应该再次执行跳转

也就是说：我们是如何处理上一个P[j]=B失配情况的，我们就应该如何处理这个P[j]=B的失配情况

最终，我们得出了对next数组进行优化的最终方案：遍历整个next数组，使得如果P[n] == P[m]，则next[n] == next[m]（n > m）

说的简单一些就是：在模式串P当中，所有取值相同的字符在next数组中对应的取值也相同

根据上述理论，我们对本例中模式串P=ABAB的next数组进行了优化，得到如下结果：

下标	0	1	2	3
字符	A	B	A	B
取值	-1	0	0	1
优化	-1	0	-1	0

下面让我们按照优化之后的next数组取值，重新执行一遍上面的案例

4.改进之后KMP算法的执行步骤

改进之后的KMP算法的执行流程依然没有变化，唯一变化的是在文本串S与模式串P之间失配的情况下，参考的next数组的取值

所以改进之后KMP算法的执行步骤在此不再赘述，让我们直接以图示的方式，来演示参考新的next数组所得到的KMP算法，在上述案例中的执行流程

```
            i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P： A B A B
          j

              i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P： A B A B
            j

                i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P： A B A B next[j] == next[2] == -1，所以：i++，j=0，重新比较
              j

                  i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：         A B A B
                  j
```

……

```
                      i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：             A B A B
                      j

                        i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：             A B A B
                        j

                          i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：             A B A B
                          j

                            i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：             A B A B next[j] == next[3] == 0，所以：i不变，j = 0，重新比较
                            j

                            i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：                   A B A B
                            j
```

……

```
                                          i
文本串S： A B C D _ A B A C A E _ A A A B A B A C
模式串P：                               A B A B 匹配成功
                                          j
```

从上述案例中我们不难看出：我们在执行KMP算法的同时，规避开了很多重复且一定会失配的情况，这就使得KMP算法的整体执行效率有所提升

各位读者如果想要更加详细的理解KMP算法在优化之后和优化之前比较方式的区别，那么可以使用上述案例分别按照原有的next数组与重新计算过的next数组的取值

从头演练一遍整体的KMP算法的运作流程，并且比较二者之间分别需要执行多少次比较，才能够得到最终匹配成功的结果。

⑤KMP算法的时间复杂度分析

KMP算法是从原始的字符串暴力匹配算法发展而来的。

KMP算法与原始的暴力匹配算法最大的不同点在于：KMP算法中用于遍历文本串S的下标i是不会回溯的

这也就是说：假设文本串S的长度为n，则在不回溯的情况下，遍历文本串S的时间复杂度是O(n)

因为在匹配过程中，使用下标j遍历模式串P的过程包含在了遍历文本串S的过程中

所以：假设模式串P的长度为m，则遍历文本串S的时间复杂度O(n)中本身包含了每一次遍历模式串P的时间复杂度O(m)

但是我们并不能忽略计算next数组所用的时间复杂度，即：假设模式串P的长度为m，则通过遍历模式串P计算next数组的时间复杂度是O(m)

所以最终我们得到KMP算法的时间复杂度 = 计算模式串P对应next数组的时间复杂度 + 遍历文本串S进行匹配的时间复杂度

即：KMP算法的时间复杂度是O(n + m)（n为文本串S的长度，m为模式串P的长度，且一般情况下n > m）