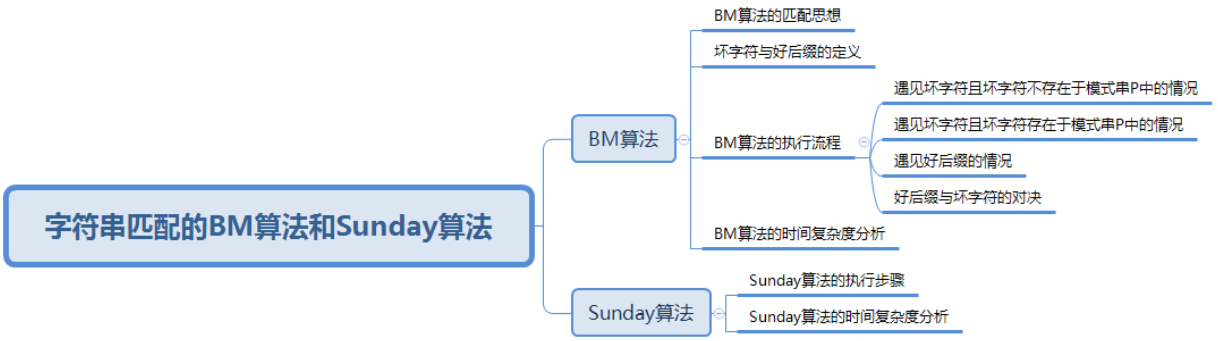


0.写在前面

在之前的章节当中，我们已经学习了字符串匹配的KMP算法。通过分析我们得知：KMP算法的时间复杂度为O(n+m)

其中，n表示文本串S的长度；m表示模式串P的长度。这个时间复杂度的由来，是根据遍历文本串S进行匹配，以及遍历模式串P计算next数组的总和得到的

实际上KMP算法在运行效率方面已经非常优秀了，但是，还有没有比KMP算法更加有效率的字符串匹配算法呢？答案是：有！

这就是BM算法和Sunday算法。

下面，就让我们一起“转换思路”，学习一下更加快速的BM算法和Sunday算法

1.BM算法

①BM算法的匹配思想

在KMP算法当中，我们采用文本串S和模式串P从前向后进行比较的方式进行字符串匹配

而BM算法则是使用以模式串P为标准的，从后向前进行匹配的方式进行字符串匹配操作。

也就是说：当文本串S与模式串P左端对齐之后，我们从模式串P的最右侧开始，逐个字符的向前与文本串S进行比较和匹配

这样的操作，可以在时间复杂程度上具有进一步的升级。

并且，BM算法从后向前的比较方式，依赖于其中定义的“坏字符”与“好后缀”两个概念

下面先让我们一起来学习一下关于这两个概念的定义

②坏字符与好后缀的定义

坏字符：当模式串P与文本串S进行从后向前进行比较的过程中，第一个遇见失配的字符，即为坏字符

好后缀：当模式串P与文本串S进行从后向前进行比较的过程中，在遇见坏字符之前，能够相互匹配的多个字符构成的子串

以及这些子串的子串，都称之为好后缀

说的简单一些就是：失配的一个字符就是坏字符，匹配的多个字符就是好后缀

注意：上述定义的坏字符和好后缀的概念，都是基于模式串P和文本串S从后向前进行比较的过程定义的

听完这些让人头晕的概念之后，下面让我们一起来看一个BM算法的实现案例

并且从其实现过程出发，理清坏字符与好后缀在BM算法中的作用

③BM算法的执行流程

下面我们借助BM算法的提出者：Boyer-Moore给出的一个案例，来对BM算法的执行流程进行讲解

假设：

文本串S=HERE_IS_A_SIMPLE_EXAMPLE

模式串P=EXAMPLE

在算法开始的时候，将文本串S和模式串P头部对齐，并且从模式串P的最右端向左端与文本串S进行逐个字符比较

```
文本串S：H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P：E X A M P L E
```

1.遇见坏字符且坏字符不存在于模式串P中的情况

首先在第1次比较的时候我们发现：文本串S中的字符S与模式串P中末尾对齐的字符E失配，

此时我们称遇到了一个“坏字符”，这个坏字符就是文本串中的字符S，也就是说：但凡是在文本串中与模式串中失配的字符，我们都称之为坏字符

```
文本串S：H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P：E X A M P L E
```

反观这个坏字符S，在模式串P当中，并没有出现过这个字符，那么如果遇见这种：坏字符没有在模式串P中出现过的情况，我们直接将模式串P移动到坏字符后面的一位

也就是将模式串P的首字符，与坏字符的下一位字符对齐，并重新从最右侧开始比较

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:               E X A M P L E
```

2.遇见坏字符且坏字符存在于模式串P中的情况

当我们接着上一步骤再次进行比较的时候我们发现：当前与模式串P尾部字符E对齐的字符是文本串S中的字符P

那么文本串的字符P与模式串的E依然是失配的，我们认为此时文本串S中的P依然是一个坏字符

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:               E X A M P L E
```

但是经过仔细观察我们发现：此时的坏字符P在模式串中出现过，那么我们就将模式串中出现的这个字符，与当前失配的坏字符对齐

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:                   E X A M P L E
```

对齐之后，我们同样是从模式串的最右端，从后向前重新进行匹配

注意：如果坏字符在模式串P中出现多次，那么我们选择与模式串中最右侧的坏字符相对齐，这样可以避免出现跳过正确匹配的情况发生

3.遇见好后缀的情况

在上图中，文本串中的字符P与模式串中的字符P对齐后，我们重新从模式串的末尾开始向前进行比较

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:                   E X A M P L E
```

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:                   E X A M P L E
```

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:                   E X A M P L E
```

```
文本串S: H E R E _ I S _ A _ S I M P L E _ E X A M P L E
模式串P:                   E X A M P L E
```

此时我们不难发现：文本串中的子串MPLE与模式串中的子串MPLE是相互匹配的，直到文本串中的字符I与模式串中的字符A失配，才出现“坏字符”的情况

此时我们称这段匹配的子串MPLE为“好后缀”；同时，我们称这段好后缀的子串：PLE、LE和E都是好后缀

4.好后缀与坏字符的对决

如果好后缀的出现，并没有使得模式串P与文本串S中的一部分完全匹配，那么就意味着在好后缀的前面，一定存在一个坏字符，使得匹配的情况终结

就如上图所示，在好后缀MPLE的前面，文本串中的I与模式串中的A失配，出现坏字符I

那么此时，我们采取如下的策略移动模式串P：

(1).如果好后缀在模式串中重复出现，计算模式串中好后缀与文本串中好后缀相重叠时移动的位数；如果好后缀在模式串中出现多次，则以模式串中最右侧的好后缀为基准

(2).如果好后缀并没有整体在模式串中出现，我们选择好后缀在模式串中出现的最长子串，执行步骤(1)

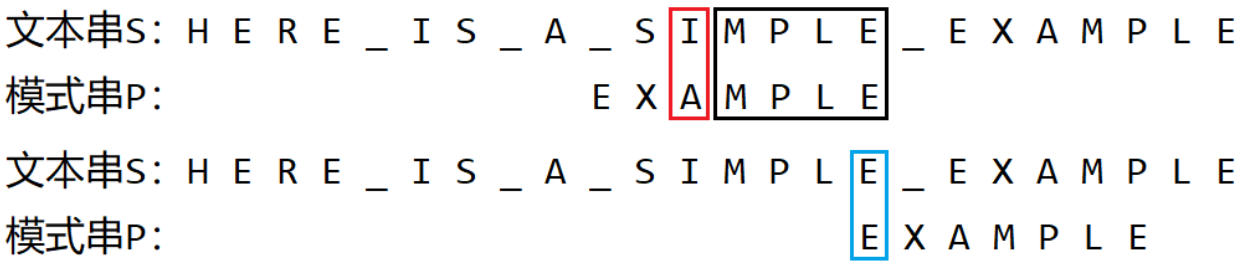
(3).计算按照失配的坏字符，模式串移动的位数

(4).取好后缀移动位数与坏字符移动位数的最大者，按照其规则进行模式串的移动

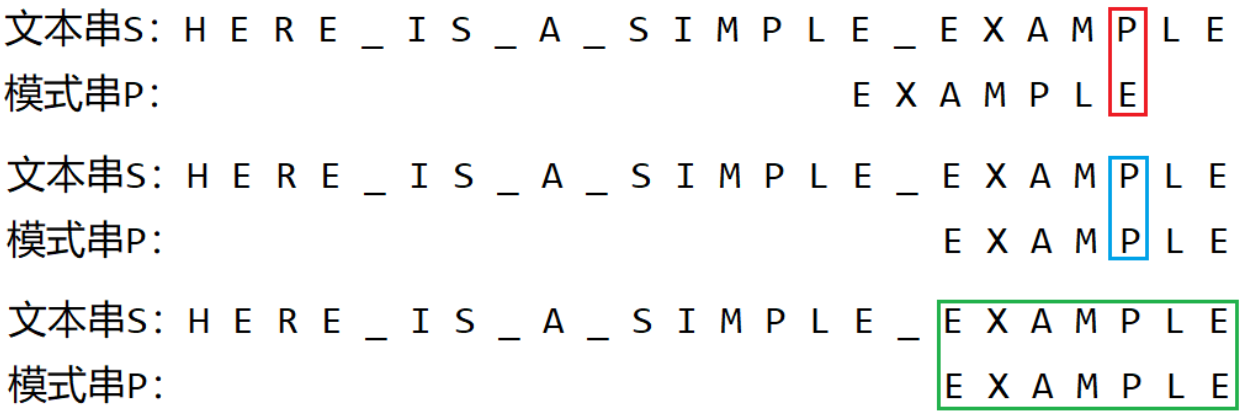
例如在上述案例中：文本串中的坏字符I与模式串中的A失配，并且坏字符I并没有在模式串中出现过，如果按照坏字符的移动规则，模式串移动位数为3位

好后缀MPLE在模式串中并没有整体出现，但是好后缀MPLE的子串，同样是好后缀的E在模式串中出现过，那么如果让文本串中的E与模式串中的E对齐

则模式串的移动位数为6位，所以最终我们选择按照好后缀的方式对模式串进行移动



最终，我们重复上述的几种情况，直到在文本串S中匹配到模式串P，或者确定文本串S中不存在模式串P为止



④BM算法的时间复杂度分析

我们假设文本串S的长度为n，模式串P的长度为m，且m <= n

在最理想情况下，假设模式串的长度等于文本串的长度，并且模式串在最开始的情况下就能够直接匹配上整个的文本串

那么我们可以得知：此时BM算法的时间复杂度等同于遍历整个文本串的时间复杂度，故而BM算法的最好时间复杂度为O(n)

在最坏情况下，我们假设每一次模式串与文本串的比较过程中，都只在最后一个字符比较的时候，也就是模式串中下标为0的字符比较的时候出现坏字符的情况

并且坏字符每次都出现在模式串中，而且通过坏字符对齐进行的比较，每一次又都无法匹配整个模式串

从这一点推断，BM算法的最坏时间复杂度是O(n*m)

在平均情况下，在每一次文本串S与模式串P失配的情况下，也就是出现坏字符的情况下，我们都假设坏字符并没有出现的模式串P中

那么我们就可以认为：每次失配的情况，都会导致模式串P向后移动m位，也就是将模式串P与坏字符之后的一个字符对齐的情况

并且我们假设，如果出现模式串P与文本串S相匹配的情况，就一定是模式串P完全匹配的情况，也就是在文本串S中找到模式串P的情况

此种情况下，我们不需要再去移动模式串P

根据上述各种说法，我们可以推断出，模式串P在文本串S中移动的次数为n/m次，每一次移动导致一次单个字符的比较

再加上最后一次匹配情况的发生，比较的次数为m次，那么算法整体的比较次数为n/m+m次

最终，我们舍弃低次幂项，得到BM算法的平均时间复杂度为：O(n/m)

由此可见，BM算法的时间复杂度O(n/m)，比KMP算法的时间复杂度O(n+m)更小，所以BM算法的执行效率高于KMP算法

2.Sunday算法

Sunday算法实际上是一种对BM算法的改进，其基本思路与BM算法相似

但是与BM算法不同的是，Sunday算法的比较是从模式串的开头进行比较的，

并且Sunday算法在不匹配情况发生时，模式串跳转的幅度更大，所以从这一点上来说，Sunday算法的效率甚至比BM算法更高

下面就让我们根据一个具体案例，来了解一下Sunday算法的具体执行流程

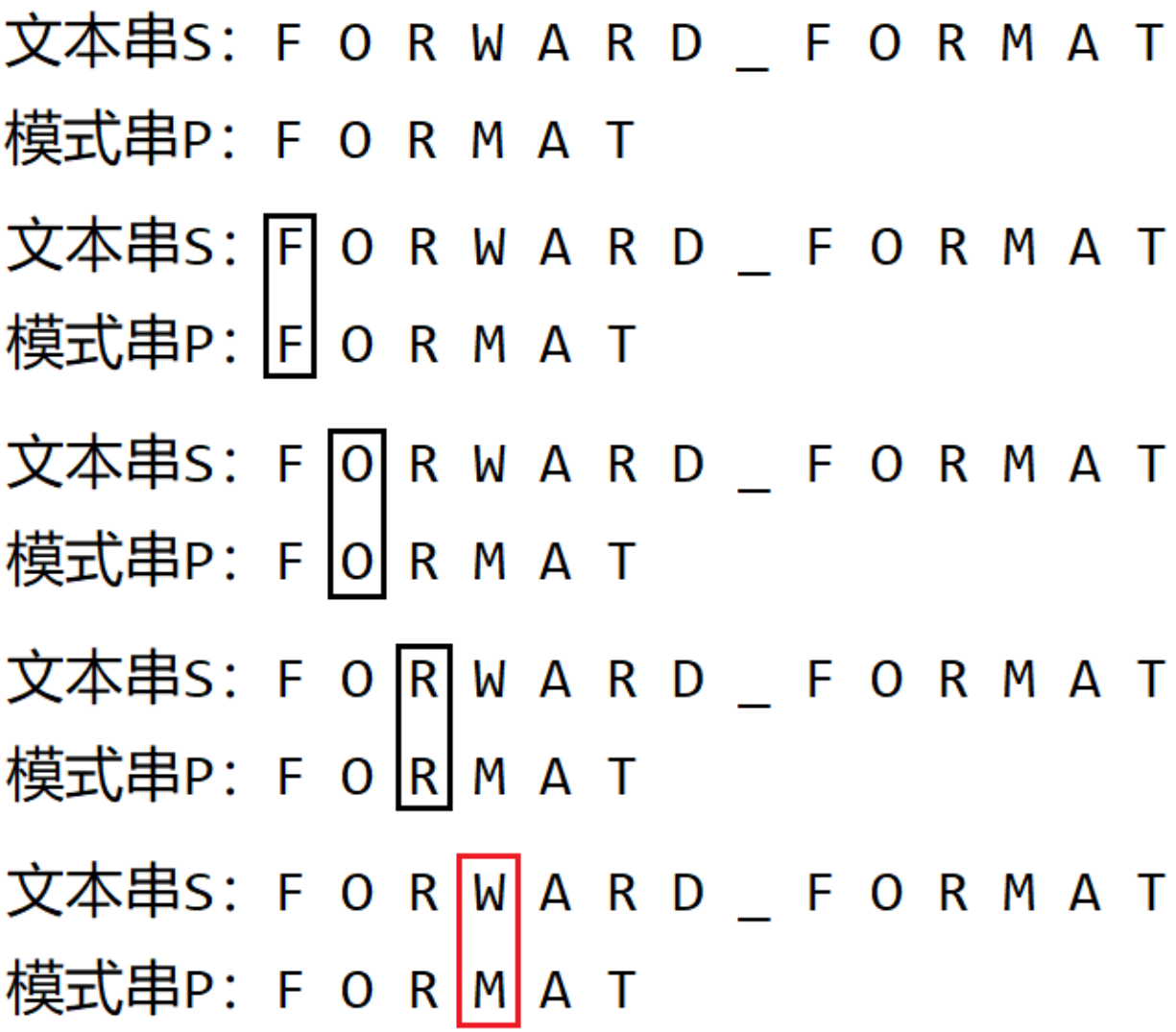
①Sunday算法的执行步骤

定义：

文本串S=FORWARD_FORMAT

模式串P=FORMAT

步骤1：首先将文本串与模式串头部对齐，从文本串与模式串的左边开始进行字符的逐个比较



步骤2：出现失配字符的情况

当出现失配字符的情况时，说明文本串失配字符之前的部分与模式串肯定不能匹配了，比较结果直接作废

按照正常的思路，我们应该观察文本串中的失配字符（本例中是W）在模式串中是否出现过

如果出现过，那么将模式串移动至两个字符相匹配的位置对齐，然后再次进行比较

如果没有出现过，那么直接将模式串移动至失配字符的下一位对齐，重新进行比较

这么做的目的很明显，是为了避免跳过可能的匹配情况

但是Sunday算法的做法是：

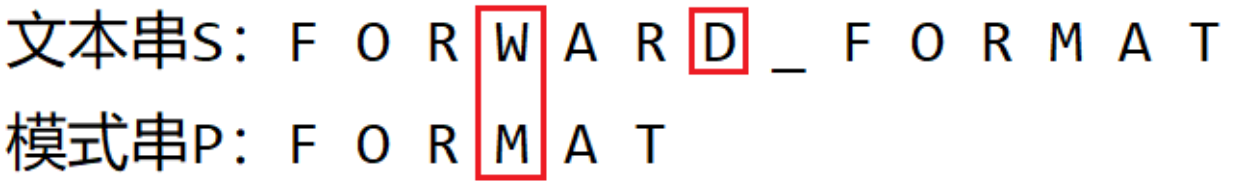
观察文本串中，与模式串对齐的最后一位字符的下一位字符（本例中就是文本串中的D）是否出现在模式串中

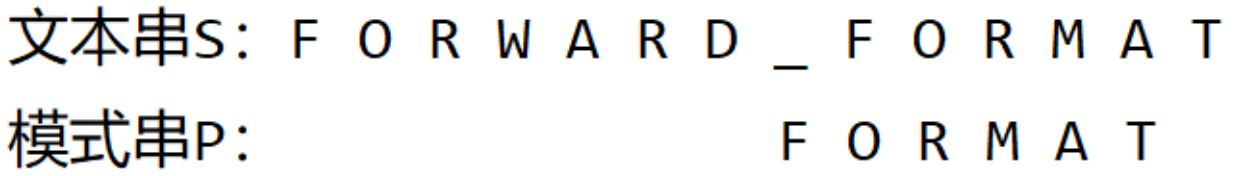
以本图为例，这一做法的意图是：如果文本串中，与模式串对齐位置上的下一位字符，也就是文本串中的D并没有出现在文本串中的话

那么就可以确定，从失配字符W向后，一直到字符D中间的这段内容与模式串肯定是不可能匹配了，所以我们可以直接跳过这段内容，

直接将模式串跳转到字符D后面的位置进行对齐，然后重新开始比较

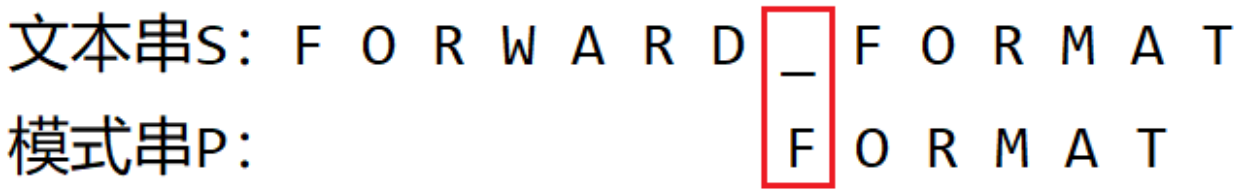
这样一来，我们就加大了跳转的范围，每一次跳转就能够多跳过一些失配的字符，提升算法效率



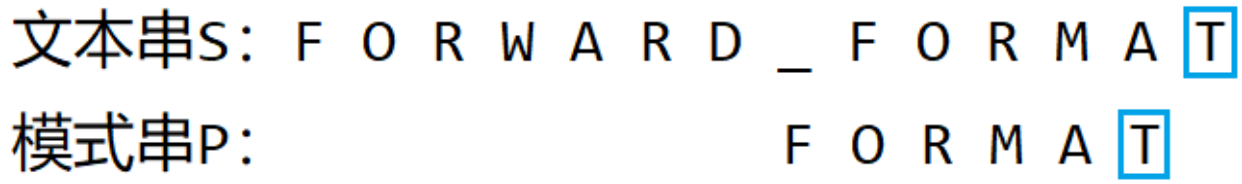


步骤3：失配后，文本串中与模式串对齐的下一位字符出现在模式串中的情况

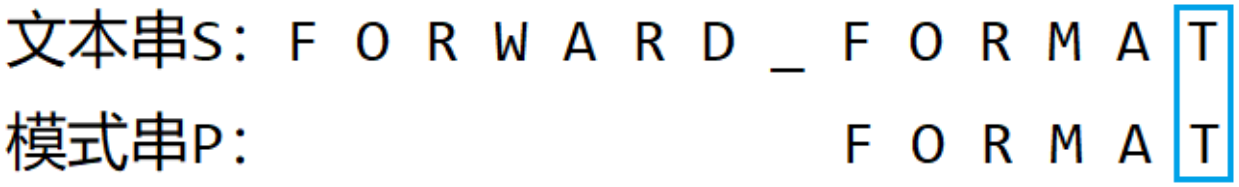
依然以上图为例，在进行模式串跳转之后，我们发现文本串中的_与模式串中的F首先就是不匹配的



但是此时在文本串中，与模式串对齐的下一位字符T在模式串中出现过



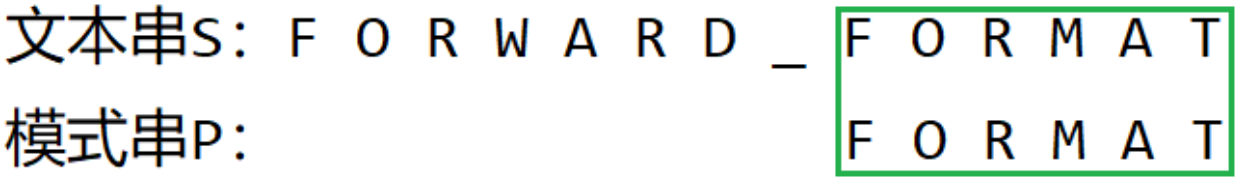
此时，我们将与模式串对齐的下一位字符与模式串中出现的位置对齐，然后继续进行比较



与BM算法相同的是，如果这个字符在模式串中出现多次，那么与模式串中最后一次出现的该字符对齐

防止跳过正确匹配的情况

最终我们成功的在文本串中匹配到模式串



从上述步骤不难看出，我们仅仅将模式串进行了2次跳转，就成功的完成了匹配操作，操作效率比BM算法还要高

②Sunday算法的时间复杂度分析

Sunday算法因为是在BM算法的基础上进行改进的，所以其理论时间复杂程度与BM算法是相似的，即：

最好时间复杂度为O(n)，最坏时间复杂度为O(n*m)，平均时间复杂度为O(n/m)

但是从实际运行的角度来看，Sunday算法的运行效率，一定是比BM算法更高的