几种解决字符串匹配的算法

（张智雄 2021112845）

摘　要： 字符串匹配是计算机科学中最古老、研究最广泛的问题之一。本文主要介绍了 Rabin-Karp 算法、有限自动机算法（Finite Automation）、Boyer-Moore 算法、Simon 算法、Colussi 算法、Galil-Giancarlo 算法、Apostolico-Crochemore 算法、Shift-Or 算法、Sunday 算法等主要使用的匹配算法，并重点实现了使用较为广泛的 Rabin-Karp 算法以及 Sunday 算法

关键词：字符串匹配　Rabin-Karp　Sunday

一、 字符串匹配算法

字符串匹配是计算机科学中最古老、研究最广泛的问题之一。一个字符串是一个定义在有限字母表∑上的字符序列。例如，ATCTAGAGA 是字母表∑ = {A,C,G,T}上的一个字符串。字符串匹配问题就是在一个大的字符串 T 中搜索某个字符串 P 的所有出现位置。其中，T 称为文本，P 称为模式，T 和 P 都定义在同一个字母表∑上。它的应用包括生物信息学、信息检索、拼写检查、语言翻译、数据压缩、网络入侵检测等领域，在当前面对海量数据的处理情况下，如何选择一个高效的匹配算法，变得愈发重要。

1.1 Rabin-Karp 算法

Rabin-Karp 算法是子字符串查找算法中的一种，主要是利用哈希函数来进行字符串的匹配。我们不需要像朴素算法那样对字符串中的字符一个个地匹配，通过一个哈希函数$H(S)$，把字符串 S 转换为一个整数与文本串进行比较，就可以匹配比较模式串与文本串。

$$H(S)=\sum_{i=0}^{|S|-1}\alpha^{|S|-(i+1)}\times char(s_i)$$

其中，代表的是 S 的定义域大小，比如说如果 S 全是英文字母，那么的α值为 26，因为英文字母就只有 26 个。然后char( )函数是一个映射函数，映射 S 的定义域中的每一个字符到数字的函数。

算法一般步骤如下：

1.首先按照上述公式计算模式串的哈希值

2.然后计算文本串中所有可能的含有相同个字符的子字符串的哈希值并寻找匹配。

对此算法进行复杂度分析，对模式串长度为 m，文本串长度为 n，在最坏的情况下，一共会有(n-m+1)次窗口滑动。这一步的复杂度是 O(n)，而计算 p 的哈希值的时间为 O(m).

然后每一次移动窗口，都需要对窗口内的字符串计算哈希值，此时这个字符串的长度为 m，所以计算它哈希值的时间也为 O(m)。如果照这样看，算法复杂度还是 O(m*n)，和传统的朴素算法在效率上没有任何区别。

但是实际上，计算移动窗口内的哈希值并不需要 O(m)，在已知前一个窗口的哈希值的情况下，计算当前窗口的哈希值，只需要 O(1)的时间复杂度。

假设现在窗口的起点在 j 这个位置，此时窗口内的字符串哈希值为：

$$H(S,j)=\sum_{i=0}^{m-1}\alpha^{m-(i+1)}\times char(s_i)$$

那么，当计算下一个窗口的哈希值时，也就是当窗口的起点为 j+1 时，哈希函数值可由如下方法计算：

因此，Rabin-Karp 算法的时间复杂度可以缩短为 O(m+n)

注意，我们在实现算法时一般要解决两个问题：

$$H(S,j+1)=\alpha(H(S,j)-\alpha^{m-1}char(s_j))+char(s_{j+m})$$

1.**溢出问题**：因为我们计算的哈希值为数值，哪怕我们用 long 类型来存储，难免会产生溢出。于是，我们就使用**模除一个素数**操作来解决。

2.**冲突问题**：之前我们使用了模除来解决溢出问题，但不同的字符串有可能产生相同的哈希值，所以我们要对哈希值相同的串再一一匹配来确定是否正确。

1.2 有限自动机算法（Finite Automation）

有限自动机(finite automata)或称为有穷状态的机器，它由一个有限的内部状态集和一组控制规则组成，这些规则是用来控制在当前状态下读入输入符号后应转向什么状态。有限自动机是一种数学模型，它可以用来描述识别输入符号串的过程，在这个机器中，它的状态总是处于有限状态中的某一个状态，系统的当前状态概括了有关历史的信息，这些历史信息对于后来的输入所能确定的系统状态是不可少的。简单地说，也就是要根据当前系统的状态和下一个输入的符号才能确定下一个状态。当进行一系列的输入，使得状态机的状态不断变化，只要最后一个输入使得状态机处于接收节点，那么就表明当前输入可以被状态机接收。

有限自动机(Finite Automata)字符串匹配算法最主要的是计算出转移函数。即给定一个当前状态 k 和一个字符 x，计算下一个状态；计算方法为：找出模式 pat 的最长前缀 prefix，同时也是 pat[0...k-1]x(注意：字符串下标是从 0 开始)的后缀，则 prefix 的长度即为下一个状态。匹配的过程是比较输入文本子串和模式串的状态值，若相等则存在，若不相等则不存在。

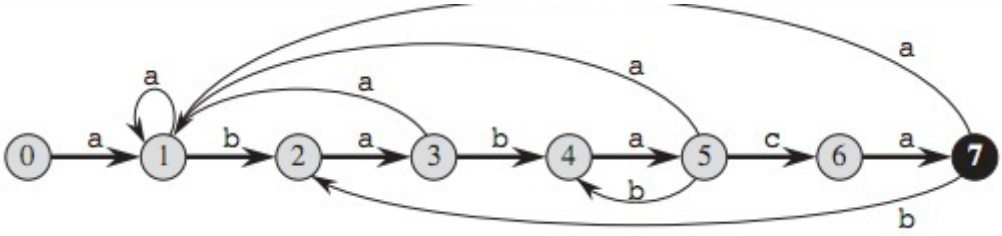
若模式串 pat 的长度为 m，则状态值为 0-m，即有 m+1 个状态，初始状态为 0。其中 n 为输入字符表的个数，计算转移函数(即预处理)的时间复杂度为 $O(m^3*n)$，匹配时间复杂度为 O(n)。该算法可以根据后面介绍的 KMP 算法进行改进，对求解转移函数的过程进行改进可以得到比较好的时间复杂度 O(m*n)。

以模式串 P=abababaca，文本串 T= abababacaba 的自动机匹配执行过程为例：

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

设计转移函数如下，0 为输入节点，7 为输出节点



具体匹配过程如下：

1. S=a, k = 1, a 是 S 的后缀
2. S=ab, k = 2, ab 是 S 的后缀
3. S=aba, k = 3, aba 是 S 的后缀
4. S=abab, k = 4, abab 是 S 的后缀
5. S=ababa, k = 5, ababa 是 S 的后缀
6. S=ababab, k = 4, abab 是 S 的后缀
7. S=abababa, k = 5, ababa 是 S 的后缀
8. S=abababac, k = 6, ababac 是 S 的后缀
9. S=ababababaca, k = 7, abababaca 是 S 的后缀
10. S=abababacab, k = 2, ab 是 S 的后缀
11. S=abababacaba, k = 3, aba 是 S 的后缀

此时，我们可以得出文本 T 包含有字符串 P，且字符串 P 在文本串中的位置为 3-9 号位。

1.3 Boyer-Moore 算法

在当前用于查找子字符串的算法中，BM(Boyer-Moore)算法是当前有效且应用比较广的一中算法，各种文本编辑器的“查找”功能，大多采用 Boyer-Moore 算法。比我们在学习的 KMP 算法快 3~5 倍。

Boyer-Moore 算法不仅效率高，而且构思巧妙，容易理解。1977 年，德克萨斯大学的 Robert S. Boyer 教授和 J Strother Moore 教授发明了这种算法。

我们知道常规的字符串匹配算法是从左往右的，这也比较符合我们一贯的思维，但是 BM 算法是从右往左的。经典的 BM 算法其实是对后缀蛮力匹配算法

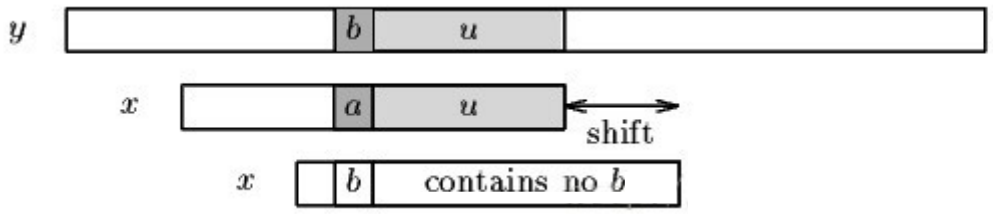
的改进，即模式串不在每次只移动一步，而是根据已经匹配的后缀信息，来判断移动的距离，从而可以跳过大量无需比较的字符，大大提高了查找效率。

为了实现更快的移动模式串，BM 定义了两个规则，坏后缀规则和好后缀规则。坏字符即这两个规则分别计算我们能够向后移动模式串长度，然后选取这两个规则中移动大的，作为我们真正移动的距离。

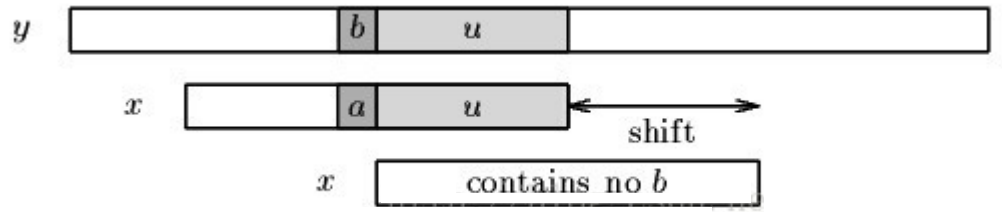
1.3.1 坏字符规则

坏字符为与文本串不匹配的字符，当出现一个坏字符时,BM 算法向右移动模式串，让模式串中最靠右的对应字符与坏字符相对，然后继续匹配。

1. 如图，y 为文本串，x 为模式串，模式串中有对应的坏字符时，将模式串 x 向右移动直至此坏字符位置首次匹配。



2. 若模式串中不存在坏字符，则将模式串整体移动到坏字符的下一个字符，继续比较。



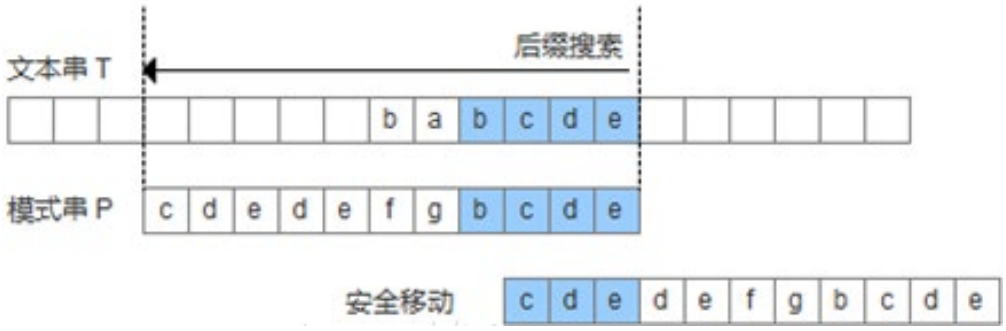
1.3.2 好后缀规则

好后缀为当前与文本串已经匹配的后缀子串，当匹配不完全成功时，向右移动字符串，具体情况如下

1. 模式串中有子串匹配上好后缀，此时移动模式串，让该子串和好后缀对齐即可，如果超过一个子串匹配上好后缀，则选择最靠右边的子串对齐，防止有漏匹配的。

2. 模式串中没有子串匹配上后后缀，此时需要寻找模式串的一个最长前缀，并让该前缀等于好后缀的后缀，寻找到该前缀后，让该前缀和好后缀对齐即可。

3. 模式串中没有子串匹配上后后缀，并且在模式串中找不到最长前缀，让



该前缀等于好后缀的后缀。此时，直接移动模式到好后缀的下一个字符。

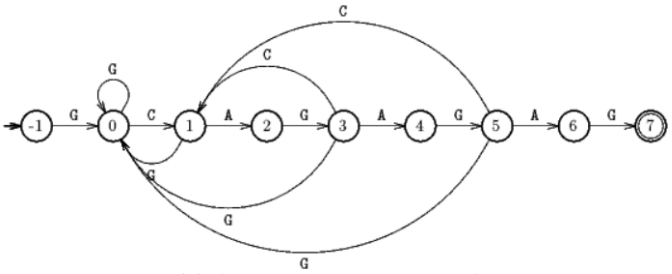
1.4 Simon 算法

Simon 算法和上述有限自动机算法类似，主要分为预处理阶段以及搜索阶段。其主要特点是简化转移函数的生成，较为高效的完成预处理阶段。

西蒙注意到$A(x)$中只有几条重要边:

1. 从长度为k的x的前缀到长度$k+1$的前缀，对于$0 \le k < m$的正向边。正好有m这样的边缘;

2. 从长度为 k的x的前缀到较小的非零长度前缀的后向边缘。此类边的数量以m为界。



状态由与它们关联的前缀的长度减去 1 来标记。

i	0	1	2	3	4	5	6
$L[i]$	(0)	(0)	∅	(0,1)	∅	(0,1)	∅

如图，其他边通向初始状态，然后可以推导。因此，有效边的数量以$2m$为界。然后，对于自动机的每个状态，只需要存储其重要传出边缘的列表。

每个状态都由其关联前缀的长度减去1表示，以便每个指向状态i的边$(-1 \le i \le m-1)$由$x[i]$标记，因此不必存储边的标签。正边可以很容易地从图案中推断出来，因此它们不会被存储。它只保留用于存储重要的后向边缘。

西蒙算法的预处理阶段可以在$O(m)$空间和时间复杂性中完成。搜索阶段类似于使用自动机的搜索阶段，当找到该模式的出现时，将使用状态更新为当前状态，时间复杂度为$O(m+n)$。

1.5 Colussi 算法

Colussi 算法改进了 KMP 算法，将模式串划分为两个不相交的子串;从左到右扫描第一组子串的位置是否正确，当没有发生不匹配时，从右到左扫描第二个子串的位置。即：

a) 在第一阶段，从左到右执行比较，文本字符与 KMP 算法中的 Next 函数的值严格大于-1 的模式位置(noholes)对齐;

b) 第二阶段包括从右到左比较剩余的位置（holes）。

此策略具有两个优点：

1. 当在第一阶段发生不匹配时，在适当的偏移之后，没有必要比较与上一次尝试中比较的 noholes 对齐的文本字符;

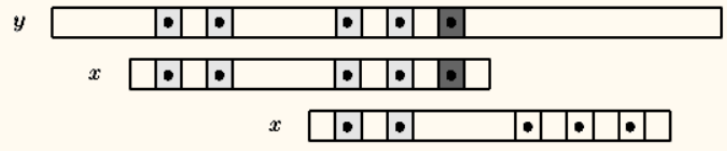


Figure 9.1: Mismatch with a nohole. Noholes are black circles and are compared from left to right.
In this situation, after the shift, it is not necessary to compare the first two noholes again.

2. 当在第二阶段发生不匹配时，这意味着模式串的部分后缀与文本串匹配，在相应的移位后，模式的前缀仍与文本串匹配，则无需再次比较此字符。

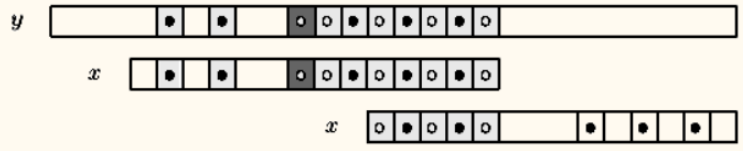


Figure 9.2: Mismatch with a hole. Noholes are black circles and are compared from left to right while holes are white circles and are compared from right to left.
In this situation, after the shift, it is not necessary to compare the matched prefix of the pattern again.

1.6 Galil-Giancarlo 算法

Galil-Giancarlo 算法是 Colussi 算法的变体，主要增加在搜索阶段中的优化。设l为模式串中的最后一个索引，使得对于 $0 \le i < l$，$x[0] = x[i]$和 $x[0] \neq x[l+1]$。假设在上一次尝试中，所有 nohole 都已匹配，并且模式串的后缀已经匹配，这意味着在相应的移位之后，模式串的前缀仍将匹配文本串的一部分。因此，窗口位于文本串 $y[j..j+m-1]$的部分，同时$y[j..last]$与$x[0..last-j]$匹配。然后在下一次尝试期间，算法将扫描以$y[last+1]$开头的文本字符，直到到达文本串的末尾或找到字符满足$x[0] \neq y[j+k]$。

在后一种情况下，可能会出现两个子情况：

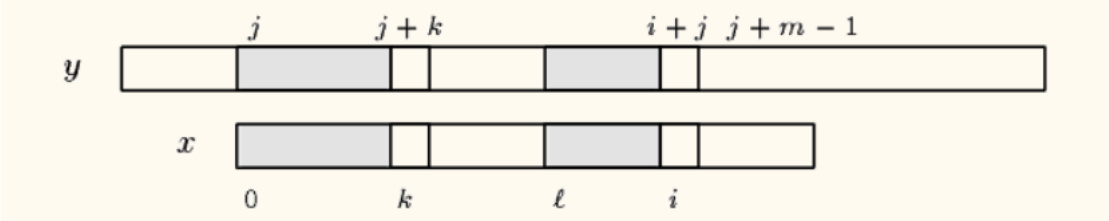
a) $x[l+1] \neq y[j+k]$或太小的$x[0]$已被找到$(k \leq l)$，然后窗口被移位并定位在文本串 $y[k+1..k+m]$，文本的扫描恢复（如在 Colussi 算法中），第一个 nohole 和模式的记忆前缀是空字。

b) $x[l+1] = y[j+k]$并且已经找到了足够的$x[0]$ $(k > l)$，然后窗口被移位并定位在文本串$y[k-l-1..k-l+m-2]$，文本的扫描恢复（如在 Colussi 算法中），第二个空孔（$x[l+1]$是第一个），并且模式的记忆前缀是 $x[0..l+1]$。

预处理阶段与 Colussi 算法相同，可以在$O(m)$空间和时间复杂度内完成。

1.7 Apostolico-Crochemore 算法

Apostolico-Crochemore 算法通过 KMP 算法中的 Next 移位表来计算移位，通过i,j,k三个参数，按照$l, l+1, \ldots, m-2, m-1, 0, 1, \ldots, l-1$的顺序进行模式串与文本串的比较，如下图所示。



其中，i,j,k满足下述条件：窗口位于文本串$y[k-l-1..k-l+m-2]$，$0 \leq k \leq l$ 且 $x[0,..,k-1] = y[j,..,j+k-1]$， $l \leq i < m$ 且 $x[l,..,i-1] = y[j+l,..,i+j-1]$。初始化参数为$(l,0,0)$。

之后(i,j,k)的更新分为以下三种情况：

a). $i = l$

如果$x[i] = y[i+j]$，则下一个三元组是$(i+1, j, k)$;

如果$x[i] \neq y[i+j]$，则下一个三元组是$(l, j+1, \max\{0, k-1\})$.

b). $l < i < m$

如果$x[i] = y[i+j]$，则下一个三元组是$(i+1, j, k)$。

如果$x[i] \neq y[i+j]$，则根据 KMPnext 的值出现两种情况：

1.$KMPnext\ [i] \leq l$: 则下一个三元组是$(i+1, j, k)$

2.$KMPnext\ [i] > l$: 那么下一个三元组是$(KMPnext\ [i], i+j-KMPnext\ [i], l)$

c). $i = m$

如果 $k < l$且$x[k] = y[j+k]$，则下一个三元组是$(i, j, k+1)$。

否则，要么$k < l$且 $x[k] \neq y[j+k]$，要么$k = l$。在此两种情况下，下一个三元组的计算方式与$l < i < m$的情况相同。

1.8 Shift-Or 算法

Shift-or 算法利用位运算的优势，大大提高了时间与空间复杂度，匹配过程时间复杂度为 O(n)，而预处理所花费的时间与空间均为 O(m+σ)。

具体过程为事先计算出字母表中每个字符在模式中出现的位置，用位的方式存在整数里，出现的地方标为 0，不出现的地方标为 1，这样总共使用 σ 个整数；同样，我用一个整数来表示状态，模式串位于某个状态就标为 0，否则标为 1，模式串的匹配就恰好可以用“移位”来表示，当检查位置的时候只需要与表示状态的整数“或”1 次，所以整个算法时间复杂度为 O(n)了。Shift-Or 算法名字就是这样来的。

但是此算法缺陷是：模式不能超过机器字长。按现在普遍的 32 位机，机器字长就是 32，也就是只能用来匹配不大于 32 个字符的模式。

1.9 Sunday 算法

著名的 KMP 算法，将字符串匹配的时间复杂度降低到了 O(m+n)，但在现代文字处理器中，却很少使用 KMP 算法来做字符串匹配。而主流的算法是 BM 算法，平均时间复杂度为 O(m/n)，而 Sunday 算法，则是对 BM 算法的进一步小幅优化。Sunday 算法是 Daniel M.Sunday 于 1990 年提出的字符串模式匹配。其效率在匹配随机的字符串时比其他匹配算法还要更快。Sunday 算法的实现可比 KMP、BM 算法的实现更为简单。此算法平均性能的时间复杂度为 O(n)；最差情况的时间复杂度为 O(n∗m)。

Sunday 算法和 BM 算法稍有不同的是，Sunday 算法是从前往后匹配，在匹配失败时关注的是**主串中参加匹配的最末位字符的下一位字符**。

如果该字符没有在模式串中出现则直接跳过，即移动位数 = 模式串长度+1；

否则，其移动位数 = 模式串长度 - 该字符最右出现的位置(以 0 开始)= 模式串中该字符最右出现的位置到尾部的距离 + 1。

二、 部分算法实现

2.1 Sunday 算法

按照上述思路，以 shift[]数组存储文本串中每一个字符若不匹配窗口移动的位数，随后多次循环嵌套条件判断，总体实现较为简单，运行速度也较快，具体实现如下：

```cpp
//储存移动步数
int shift[maxNum];

int Sunday(const string T, const string P) {
    int n = T.length();
    int m = P.length();

    // 模式串长度+1
    for(int i = 0; i < maxNum; i++)
        shift[i] = m + 1;
    // 模式串长度 - 该字符最右出现的位置(以0开始) = 模式串中该字符最右出现的位置到尾部的距离 + 1
    for(int i = 0; i < m; i++)
        shift[P[i]] = m - i;
    // 模式串窗口在主串的起始位置
    int s = 0;
    // 模式串已经匹配到的位置
    int j;
    while(s <= n - m) {
        j = 0;
        while(T[s + j] == P[j]) {
            j++;
            // 字符串全部匹配，返回当前模式串的位置
            if(j >= m)
                return s;
        }
        //移动窗口位置
        s += shift[T[s + m]];
    }
    return -1;
}
```

2.2 Rabin-Karp 算法

按照上述思路，char( )对应为字符串的 ACSII 码，采用简化复杂度后的哈希值计算方式，同时为确保准确性，增加对哈希值相同子串的逐个检查，具体实现如下：

```cpp
    int p = 0; // pattern哈希值
    int t = 0; // txt哈希值
    int h = ((int)pow(d, M-1)) % q;

    // 计算模式串和文本串的第一个窗口的哈希值
    for (i = 0; i < M; i++)
    {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    for (i = 0; i <= N - M; i++)
    {
        // 如果哈希值相等，则逐个检查每个字符是否匹配
        if (p == t)
        {
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
            // 字符串全部匹配，返回当前模式串的起始位置
            if (j == M)
                return i;
        }
        // 计算文本串下一个窗口的哈希值
        if (i < N-M)
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;
            //负值处理
            if (t < 0)
                t = (t + q);
        }
    }
    return -1;
```

三、 总结

字符串匹配是模式匹配中最简单的一个问题，但在文本处理领域中字符匹配是一个非常重要的主题。它可用于数据处理、数据压缩、文本编辑、信息检索等多种应用中，大多数操作系统中软件实现的字符匹配算法是基本组件之一。字符串匹配技术通常也和其他字符问题有一定关联。在实际应用中字符串匹配技术不仅适用于计算机科学，在语义学、分子生物学等领域也具有相当重要的应用，在以模式匹配为特征的网络安全应用中也发挥了举足轻重的作用。掌握多种算法，针对不同应用场景选择适合的算法有助于提高解决问题的效率以及正确率，对今后学习数据结构的过程中也有一定帮助。

四、 参考文献

[1] 母泽平.字符串匹配算法探讨[J].重庆工商大学学报(自然科学版),2014,31(08):79-82.

[2] 精确字符串匹配算法 https://monge.univ-mlv.fr/~lecroq/string/

[3]Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest,等. 算法导论(原书第 3 版)[M]. 机械工业出版社, 2012.