

史上比较难懂的KMP算法介绍

原创

鄙人来自大唐

2020-10-25 13:18:49

916

★ 收藏

3

版权

分类专栏:


算法

文章标签:

c++

c算法

算法



算法 专栏收录该内容

0 订阅

1 篇文章

订阅专栏

书生来自秦朝南海郡，是一秃头学子。
取经之路漫漫，沉心学习方见始终。



目录

- 前言
- 一、串匹配简介及KMP引入

1.串匹配

2.暴力法串匹配

3.KMP算法相关引入

二、KMP核心思想

三、KMP算法

后言

前言

数据结构里面有个KMP算法，**法苏ovo**是助教，他就有心做了个文章。这篇文章整体来说，是很贴心的，但是，看不懂也没有关系。

俺也没看懂XD

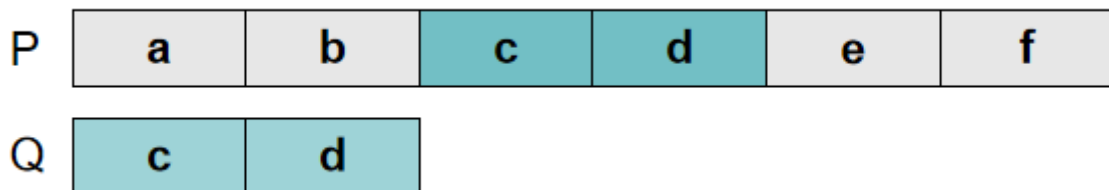
如果这篇看不懂，可以看这篇，讲的挺好的：<https://www.cnblogs.com/zzuuoo666/p/9028287.html>

众所周知，KMP这玩意挺难理解的，这里就介绍一下KMP嘛~

一、串匹配简介及KMP引入

1.串匹配

串匹配问题，就是给你一个很长的字符串P，然后给你一个目标字符串Q，现在让你在字符串P中找到字符串Q出现的位置。



2.暴力法串匹配

最简单粗暴的就是使用两个下标*i*，*j*分别指向两个字符串P和Q，一开始都从第一个字符开始逐个比较，如果相同，则同时后移一位进行下一个字符的比较，如果遇到一个不相同的字符，则将Q的下标*j*置零，也就是从头开始比较，而P的下标则重置到开始比较的下一个位置。

这里怕机器画图大家不好理解，就手动画图了ovo（不是因为懒）
就是循环查找遍历啦~

$P[]$ = 字符串: a b a b c

$Q[]$ = 匹配: a b c

round 1: a b a b c
 \downarrow
 (a)

$$P[i] = Q[j]$$

$$P[0] = Q[0]$$

round 2: a b a b c
 \downarrow
 a (b)

$$P[1] = Q[1]$$

round 3: a b a b c
 \times
 a ~~b~~ (c)

$$P[2] \neq Q[2]$$

round 4: a b a b c
 \times
 (a)

$$P[1] \neq Q[0]$$

https://blog.csdn.net/weixin_43814362

暴力法实现代码如下：

```
1  int getIndex(string p,string q)
2  {
3      int i=0;
4      int j=0;
5      while(i<(int)p.length() && j<(int)q.length())
6      {
7          if(p[i] == q[j])
8          {
9              i++;
10             j++;
11         }
12         else
13         {
14             i=i-j+1;
15             j=0;
16         }
17         if(j==q.length())
18             return i-q.length();
19     }
20     return -1;
21 }
```

3.KMP算法相关引入

在讲解KMP算法的本质时，首先引入字符串的前缀和后缀的概念，下面是我对于前缀和后缀的个人理解。

前缀：从字符串第一个字符开始的任意长度的连续子串

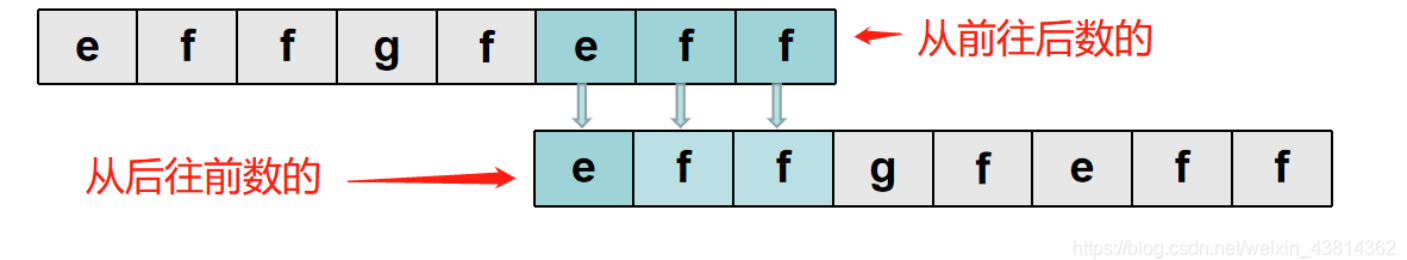
后缀：从字符串最后一个开始往前的任意长度的连续子串

举个例子，对于字符串 **"ababc"**，前后缀如下所示。

前缀	""(空串),"a","ab","aba","abab","ababc"
后缀	""(空串),"c","bc","abc","babc","ababc"

最长相同前缀后缀：顾名思义，就是对于一个字符串来说，可以列出上面的前缀和后缀集合，那么最长相同前缀后缀就是 **前缀字符串集合和后缀字符串集合的并集中的最长的字符串。**

例如，对于字符串"eefegeef"而言，最长相同前缀后缀为"eef"，长度为3。通过下图来理解，即两个字符串的最长相交长度。



KMP算法其实就是依靠最长相同前缀后缀进行比较的，next数组中存的值其实就是最长相同前缀后缀的长度。

在进行字符串匹配时，如果 $P[i] \neq Q[j]$ ，其实这也是第一次不匹配，这说明了之前的 $j-1$ 个字符都匹配上了，对于这 $j-1$ 个字符构成的字符串 S ，我们只需要从 S 的最长相同前缀后缀的长度处开始下一轮比较即可，不需要再回退 i 。

怎么理解上面的本质呢，首先可以肯定是前 $j-1$ 个字符是匹配的，设前 $j-1$ 个字符的最长相同前缀后缀的长度为 n ，则可以知道对于这 $j-1$ 个字符构成的字符串 S 来说，后面 n 个字符是和原串 P 相匹配的，且后面 n 个字符与 S 的前 n 个字符也是相同的，所以可以直接把前缀的 n 个长度部分放到这里来，放过来之后依旧是匹配的。

二、KMP核心思想

KMP算法的核心在于求出每个前缀的最长相同前缀后缀。在KMP算法中，这个集合被称作next数组。所以求出next数组就是整个KMP算法的核心步骤。

根据前面的分析， $next[j]$ 的值应该是前 $j-1$ 个字符构成的字符串的最长相同前缀后缀的长度。就是如下公式：

$$next[j] = \begin{cases} -1, j = 0 \\ \max\{k \mid 0 < k < j \text{ 且 } Q_0 \cdots Q_{k-1} = Q_{j-k} \cdots Q_{j-1}\} \end{cases}$$

根据上述公式，可以知道如何求出 $next$ 数组，但是这种计算方法过于繁琐，每一次都要求一次最长相同前缀后缀的长度。

这里我们可以利用KMP算法的本质思想来推导 $next$ 数组。

假设 $next[j] = k$ ，那么说明 $Q(0) \cdots Q(k-1) = Q(j-k) \cdots Q(j-1)$ ，那么接下来，如果 $Q[k] = Q[j]$ ，那么很明显可以得出 $next[j] = k+1$ 。如果 $Q[k] \neq Q[j]$ ，这里使用KMP算法的思想，令 $k = next[k]$ ，然后再去判断 $Q[k]$ 与 $Q[j]$ 的关系。

$$\begin{array}{ccccccc} Q_{j-k} & \cdots & Q_{j-next[k]} & \cdots & Q_{j-2} & Q_{j-1} & Q_j \\ Q_0 & \cdots & \cdots & \cdots & Q_{k-2} & Q_{k-1} & Q_k \\ \vdots & \cdots & Q_0 & \cdots & Q_{next[k]-2} & Q_{next[k]-1} & Q_{next[k]} \end{array}$$

从上表中可以看出，当第一次 $Q[k] \neq Q[j]$ 时，就变成了第三行和第一行的对比，类似于之前的第一行和第二行的一个比较。关键就在于 $k = next[k]$ ，充分利用了KMP算法的思想，对于 $Q(0) \cdots Q(k-1)$ ，最长相同前缀后缀长度为 $next[k]$ ，但是由于 $Q[k] \neq Q[j]$ ，所以 $next[j] \neq k+1$ 。这里根据KMP算法的思想，由于

$Q(0) \cdots Q(k-1)$ 的最长相同前缀后缀为 $Q(0) \cdots Q(next[k]-1)$ ，所以也可以将这条串与原本的串进行比较，因为这个也是原本的串的一个相同前缀后缀，只不过不是最长的。利用这一点，进行 $next[j]$ 的计算。

计算 $next[]$ 函数代码：

```

1
2  void getNext(int *next, string q)
3  {
4      int j=0;
5      int k=-1;
6      next[j] = k;
7      while(j<(int)q.size())
8      {
9          if(k== -1 || q[j] == q[k])
10

```

```
11     {
12         j++;
13         k++;
14         next[j]=k;
15     }
16     else
17     {
18         k=next[k];
19     }
20
21 }
```

三、KMP算法

在前面计算好next数组的值之后，就可以开始进行字符串匹配了，具体匹配方法完全参考第一部分KMP算法的本质，只是第一部分没有引入next数组，但是根据第二部分可知，next数组记录的就是最长相同前缀后缀的长度。具体到代码实现的时候稍微注意一些细节就可以了。

KMP算法代码：

```
1
2  int KMP(string p,string q)
3  {
4      int *next = new int[(int)q.size()];
5      getNext(next,q);
6      int i=0;
7      int j=0;
8      while(i<(int)p.size() && j<(int)q.size())
9      {
10         if(j==-1 || p[i] == q[j])
11         {
12             i++;
13             j++;
14         }
15         else
16         {
17             j=next[j];
18         }
19     }
20     if(j==(int)q.size())
21         return i-j;
22     return -1;
23
24 }
```

后言

如果看不懂的话，可以去这篇文章地址，讲的挺详细的~

跳转链接：[很详尽的KMP算法](#)

文章来自我的室友[法苏ovo](#)，下面是他的公众号，有兴趣的可以去逛逛哟~

如有问题，扫描下方二维码直接找[法苏ovo](#)询问即可~