

KMP 算法

 **Ethson** 发布于 2017-03-05

原文链接: <https://ethsonliu.com/2018/04...>

一：背景

给定一个主串（以 S 代替）和模式串（以 P 代替），要求找出 P 在 S 中出现的位置，此即串的模式匹配问题。

Knuth-Morris-Pratt 算法（简称 KMP）是解决这一问题的常用算法之一，这个算法是由高德纳（Donald Ervin Knuth）和沃恩·普拉特在 1974 年构思，同年詹姆斯·H·莫里斯也独立地设计出该算法，最终三人于 1977 年联合发表。

在继续下面的内容之前，有必要在这里介绍下两个概念：**真前缀** 和 **真后缀**。

字符串：“china”

真前缀： c, ch, chi, chin

真后缀： hina, ina, na, a

由上图所得，“真前缀”指除了自身以外，一个字符串的全部头部组合；“真后缀”指除了自身以外，一个字符串的全部尾部组合。（网上很多博客，应该说是几乎所有的博客，也包括我以前写的，都是“前缀”。严格来说，“真前缀”和“前缀”是不同的，既然不同，还是不要混为一谈的好！）

二：朴素字符串匹配算法

初遇串的模式匹配问题，我们脑海中的第一反应，就是朴素字符串匹配（即所谓的暴力匹配），代码如下：

```
/* 子串下标始于 0 */
int NaiveStringSearch(string S, string P)
{
    int i = 0;    // S 的下标
    int j = 0;    // P 的下标
    int s_len = S.size();
    int p_len = P.size();

    while (i < s_len && j < p_len)
    {
        if (S[i] == P[j]) // 若相等，都前进一步
        {
            i++;
            j++;
        }
        else // 不相等
        {
            i = i - j + 1;
            j = 0;
        }
    }

    if (j == p_len) // 匹配成功
        return i - j;
}
```

暴力匹配的时间复杂度为 $O(nm)$ ，其中 n 为 S 的长度， m 为 P 的长度。很明显，这样的时间复杂度很难满足我们的需求。

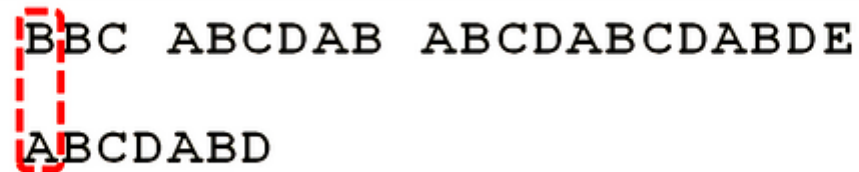
接下来进入正题：时间复杂度为 $\Theta(n + m)$ 的 KMP 算法。

三：KMP字符串匹配算法

3.1 算法流程

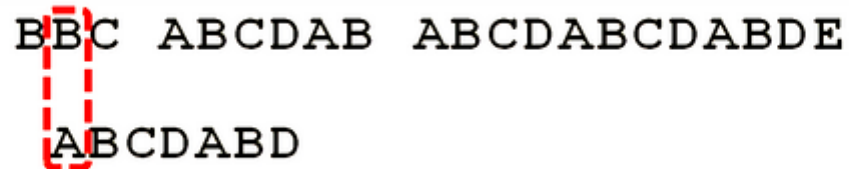
以下摘自阮一峰的[字符串匹配的KMP算法](#)，并作稍微修改。

(1)



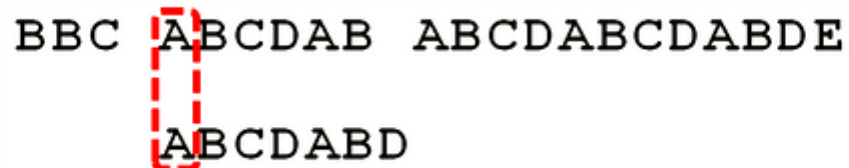
首先，主串"BBC ABCDAB ABCDABCDABDE"的第一个字符与模式串"ABCDABD"的第一个字符，进行比较。因为 B 与 A 不匹配，所以模式串后移一位。

(2)



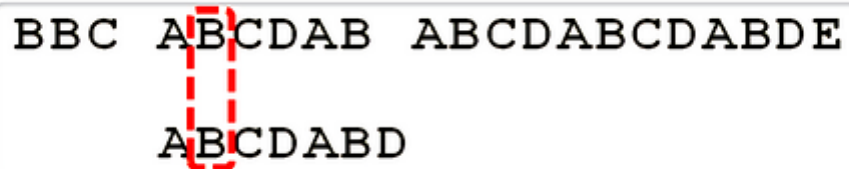
因为 B 与 A 又不匹配，模式串再往后移。

(3)



就这样，直到主串有一个字符，与模式串的第一个字符相同为止。

(4)



接着比较主串和模式串的下一个字符，还是相同。

(5)

BBC ABCDAB ABCDABCDABDE
ABCDABD

直到主串有一个字符，与模式串对应的字符不相同为止。

(6)

BBC ABCDAB ABCDABCDABDE
ABCDABD

这时，最自然的反应是，将模式串整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

(7)

BBC ABCDAB ABCDABCDABDE
ABCDABD

一个基本事实是，当空格与 D 不匹配时，你其实是已经知道前面六个字符是"ABCDAB"。KMP 算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，而是继续把它向后移，这样就提高了效率。

(8)

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|----|---|---|---|---|---|---|------|
| 模式串 | A | B | C | D | A | B | D | '\0' |
| next[i] | -1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

怎么做到这一点呢？可以针对模式串，设置一个跳转数组 `int next[]`，这个数组是怎么计算出来的，后面再介绍，这里只要会用就可以了。

(9)

BBC ABCDAB ABCDABCDABDE
ABCDABD

已知空格与 D 不匹配时，前面六个字符"ABCDAB"是匹配的。根据跳转数组可知，不匹配处 D 的 next 值为 2，因此接下来**从模式串下标为 2 的位置开始匹配。**

(10)

BBC ABCDAB ABCDABCDABDE

ABCDABD

因为空格与 C 不匹配，C 处的 next 值为 0，因此接下来模式串从下标为 0 处开始匹配。

(11)

BBC ABCDAB ABCDABCDABDE

ABCDABD

因为空格与 A 不匹配，此处 next 值为 -1，表示模式串的第一个字符就不匹配，那么直接往后移一位。

(12)

BBC ABCDAB ABCDABCDABDE

ABCDABD

逐位比较，直到发现 C 与 D 不匹配。于是，下一步从下标为 2 的地方开始匹配。

(13)

BBC ABCDAB ABCDABCDABDE

ABCDABD

逐位比较，直到模式串的最后一位，发现完全匹配，于是搜索完成。

3.2 next 数组是如何求出的

next 数组的求解基于“真前缀”和“真后缀”，即next[i]等于P[0]...P[i - 1]最长的相同真前后缀的长度（请暂时忽视 i 等于 0 时的情况，下面会有解释）。我们依旧以上述的表格为例，为了方便阅读，我复制在下方了。

| | | | | | | | | |
|---------|----|---|---|---|---|---|---|------|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 模式串 | A | B | C | D | A | B | D | '\0' |
| next[i] | -1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

1. i = 0, 对于模式串的首字符，我们统一为next[0] = -1;
2. i = 1, 前面的字符串为A, 其最长相同真前后缀长度为 0, 即next[1] = 0;
3. i = 2, 前面的字符串为AB, 其最长相同真前后缀长度为 0, 即next[2] = 0;
4. i = 3, 前面的字符串为ABC, 其最长相同真前后缀长度为 0, 即next[3] = 0;
5. i = 4, 前面的字符串为ABCD, 其最长相同真前后缀长度为 0, 即next[4] = 0;
6. i = 5, 前面的字符串为ABCD A, 其最长相同真前后缀为A, 即next[5] = 1;
7. i = 6, 前面的字符串为ABCDAB, 其最长相同真前后缀为AB, 即next[6] = 2;
8. i = 7, 前面的字符串为ABCDABD, 其最长相同真前后缀长度为 0, 即next[7] = 0。

那么，为什么根据最长相同真前后缀的长度就可以实现在不匹配情况下的跳转呢？举个代表性的例子：假如 $i = 6$ 时不匹配，此时我们是知道其位置前的字符串为 **ABCDAB**，仔细观察这个字符串，首尾都有一个 **AB**，既然在 $i = 6$ 处的 **D** 不匹配，我们为何不直接把 $i = 2$ 处的 **C** 拿过来继续比较呢，因为都有一个 **AB** 啊，而这个 **AB** 就是 **ABCDAB** 的最长相同真前后缀，其长度 2 正好是跳转的下标位置。

有的读者可能存在疑问，若在 $i = 5$ 时匹配失败，按照我讲解的思路，此时应该把 $i = 1$ 处的字符拿过来继续比较，但是这两个位置的字符是一样的啊，都是 **B**，既然一样，拿过来比较不就是无用功了么？其实不是我讲解的有问题，也不是这个算法有问题，而是这个算法还未优化，关于这个问题在下面会详细说明，不过建议读者不要在这里纠结，跳过这个，下面你自然会恍然大悟。

思路如此简单，接下来就是代码实现了，如下：

```
/* P 为模式串，下标从 0 开始 */
void GetNext(string P, int next[])
{
    int p_len = P.size();
    int i = 0;    // P 的下标
    int j = -1;
    next[0] = -1;

    while (i < p_len)
    {
        if (j == -1 || P[i] == P[j])
        {
            i++;
            j++;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

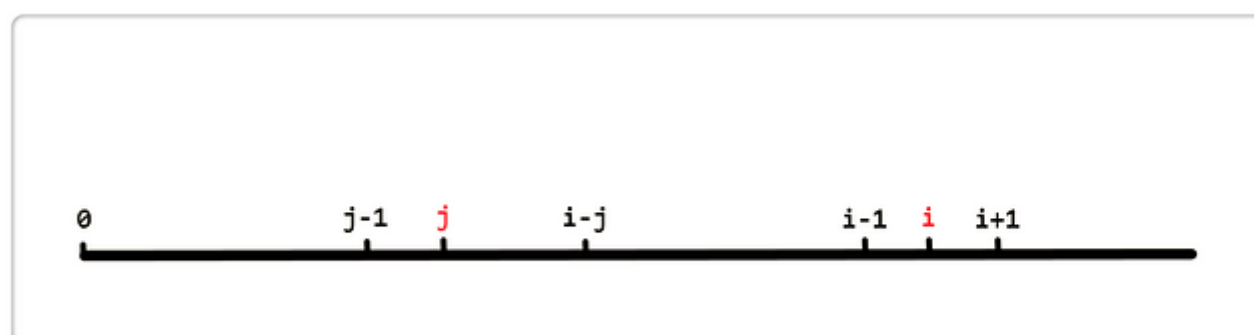
一脸懵逼，是不是。。。上述代码就是用来求解模式串中每个位置的 `next[]` 值。

下面具体分析，我把代码分为两部分来讲：

(1)：i 和 j 的作用是什么？

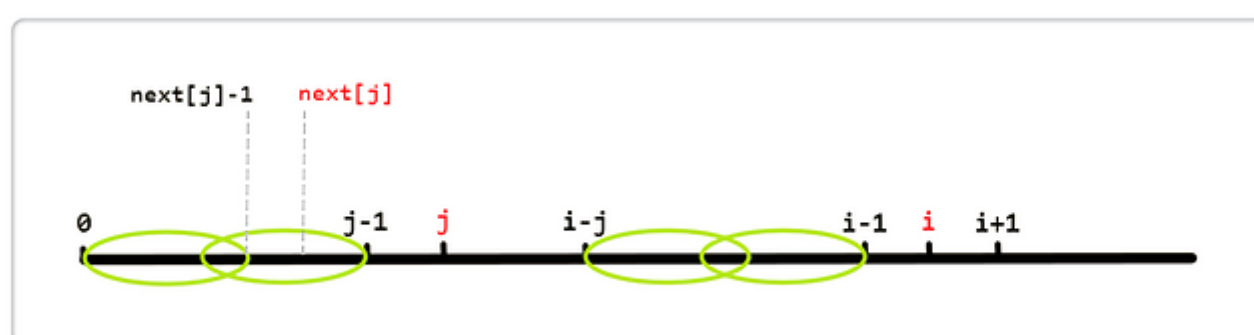
i 和 j 就像是两个“指针”，一前一后，通过移动它们来找到最长的相同真前后缀。

(2)：if...else...语句里做了什么？



假设 i 和 j 的位置如上图，由 `next[i] = j` 得，也就是对于位置 i 来说，**区段 $[0, i - 1]$ 的最长相同真前后缀分别是 $[0, j - 1]$ 和 $[i - j, i - 1]$ ，即这两区段内容相同。**

按照算法流程，`if (P[i] == P[j])`，则 `i++`；`j++`；`next[i] = j`；若不等，则 `j = next[j]`，见下图：



`next[j]`代表 $[0, j - 1]$ 区段中最长相同真前后缀的长度。如图，用左侧两个椭圆来表示这个最长相同真前后缀，即这两个椭圆代表的区段内容相同；同理，右侧也有相同的两个椭圆。所以 `else` 语句就是利用第一个椭圆和第四个椭圆内容相同来加快得到 $[0, i - 1]$ 区段的相同真前后缀的长度。

细心的朋友会问 `if` 语句中 `j == -1` 存在的意义是何？第一，程序刚运行时，`j` 是被初始为 `-1`，直接进行 `P[i] == P[j]` 判断无疑会边界溢出；第二，`else` 语句中 `j = next[j]`，`j` 是不断后退的，若 `j` 在后退中被赋值为 `-1`（也就是 `j = next[0]`），在 `P[i] == P[j]` 判断也会边界溢出。综上两点，其意义就是为了特殊边界判断。

四：完整代码

```
#include <iostream>
#include <string>

using namespace std;

/* P 为模式串，下标从 0 开始 */
void GetNext(string P, int next[])
{
    int p_len = P.size();
    int i = 0;    // P 的下标
    int j = -1;
    next[0] = -1;

    while (i < p_len)
    {
        if (j == -1 || P[i] == P[j])
        {
            i++;
            j++;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

五：KMP优化

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|----|---|---|---|---|---|---|------|
| 模式串 | A | B | C | D | A | B | D | '\0' |
| next[i] | -1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

以 3.2 的表格为例（已复制在上方），若在 `i = 5` 时匹配失败，按照 3.2 的代码，此时应该把 `i = 1` 处的字符拿过来继续比较，但是这两个位置的字符是一样的，都是 `B`，既然一样，拿过来比较不就是无用功了么？这我在 3.2 已经解释过，之所以会这样是因为 KMP 还未优化。那怎么改写就可以解决这个问题呢？很简单。

```
/* P 为模式串，下标从 0 开始 */
void GetNextval(string P, int nextval[])
{
    int p_len = P.size();
    int i = 0;    // P 的下标
    int j = -1;
    nextval[0] = -1;

    while (i < p_len)
    {
        if (j == -1 || P[i] == P[j])
        {
            i++;
            j++;

            if (P[i] != P[j])
                nextval[i] = j;
            else
                nextval[i] = nextval[j];    // 既然相同就继续往前找真前缀
        }
        else
            j = nextval[j];
    }
}
```

六：参考文献

- 严蔚敏. 数据结构（C 语言版）
- 阮一峰. [字符串匹配的KMP算法](#)

[c++](#) [算法](#) [数据结构](#) [kmp](#)

阅读 58.5k • 更新于 2020-01-12

 赞 31

 收藏 26

 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



经典算法与数据结构

2019-12-27 文章正在重新编辑...

关注专栏



Ethson

1.9k 声望 61 粉丝

关注作者

0 条评论

得票 · 时间



撰写评论 ...

 提交评论

推荐阅读

扩展 KMP 算法

问题定义：给定两个字符串S和T（长度分别为n和m），下标从0开始，定义extend[i]等于S[i]...S[n-1]与T的最长相同前缀的长度，求...

Ethson · 阅读 16.6k · 12 赞 · 7 评论

KMP算法及优化

今天看到同学在复习数据结构书上的KMP算法，忽然发觉自己又把KMP算法忘掉了，以前就已经忘过一次，看样子还是没有真正的...

疯狂的爱因斯坦 · 阅读 16.6k · 11 赞 · 17 评论

KMP字符串匹配算法

KMP算法是一种改进的字符串匹配算法，由D.E.Knuth，J.H.Morris和V.R.Pratt提出的，因此人们称它为克努特—莫里斯—普拉特操...

CodeNeil · 阅读 735 · 7 赞

KMP算法

KMP算法是一种改进的字符串匹配算法，由D.E.Kunth,J.H.Morris和V.R.Pratt提出，KMP算法的功能是在一个主文本字符串s中查找模...

lioneey · 阅读 875 · 4 赞

【数据结构】41 KMP字串查找算法

问题 如何在目标字符串 S 中，查找是否存在子串 P？ 朴素解法 {代码...} 朴素算法的一个线索优化 因为，pa != pb 且 pb == sb;所以...

TianSong · 阅读 844 · 3 赞

KMP 算法

字符串匹配算法是常见的一种字符串操作，其是在一个主字符串中查找一个子字符串（也叫模式串），即判断模式串是否是主字符...

guozhchun · 阅读 1.7k · 1 赞

[Leetcode] Shortest Palindrome 最短回文拼接法

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest pali...

ethannnli · 阅读 4.5k · 1 赞

【第5期】算法精选-你应该知道的KMP算法

先说一下这个算法出现的背景，也就是解决什么问题。每个算法和框架都有它出现的原因和要解决的问题，很多时候你不会一个技...

耳东 · 阅读 450 · 1 赞

产品

热门问答

热门专栏

热门课程

最新活动

技术圈

酷工作

移动客户端

课程

Java 开发课程

PHP 开发课程

Python 开发课程

前端开发课程

移动开发课程

资源

每周精选

用户排行榜

徽章

帮助中心

声望与权限

社区服务中心

合作

关于我们

广告投放

职位发布

讲师招募

联系我们

合作伙伴

关注

产品技术日志

社区运营日志

市场运营日志

团队日志

社区访谈

条款

服务条款

隐私政策

下载 App