老司机开车，教会女朋友什么是「马拉车算法」

点击蓝色“五分钟学算法”关注我哟

加个“星标”，天天中午 12:15，一起学算法

投稿作者 | 李威

来源 | https://www.liwei.party

整理 | 五分钟学算法

马拉车算法（ Manacher‘s Algorithm ）是程序员小吴最喜欢的算法之一，因为，它真的很牛逼！

马拉车算法是用来 **查找一个字符串的最长回文子串的线性方法** ，由一个叫 Manacher 的人在 1975 年发明的，这个方法的牛逼之处在于将时间复杂度提升到了 **线性** 。

事实上，马拉车算法在思想上和 **KMP** 字符串匹配算法有相似之处，都避免做了很多重复的工作。

动画：七分钟理解什么是KMP算法

Manacher 算法本质上还是 **中心扩散法** ，只不过它使用了类似 KMP 算法的技巧，充分挖掘了已经进行回文判定的子串的特点，提高算法的效率。

下面介绍 Manacher 算法的运行流程。

首先还是“中心扩散法”的思想：回文串可分为奇数回文串和偶数回文串，它们的区别是：奇数回文串关于它的“中点”满足“中心对称”，偶数回文串关于它“中间的两个点”满足“中心对称”。

为了避免对于回文串字符个数为奇数还是偶数的套路，首先对原始字符串进行预处理，方法也很简单：*添加分隔符*。

第 1 步：预处理

第一步是对原始字符串进行预处理，也就是 *添加分隔符* 。

首先在字符串的首尾、相邻的字符中插入分隔符，例如 `"babad"` 添加分隔符 `"#"` 以后得到 `"#b#a#b#a#d#"` 。

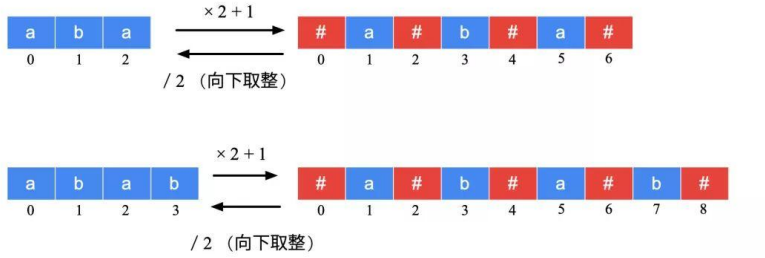
对这一点有如下说明：

1、分隔符是一个字符，种类也只有一个，并且这个字符一定不能是原始字符串中出现过的字符；

与之对应，因此对新字符串的回文子串的研究就能得到原始字符串的回文子串；

3、新字符串的回文子串的长度一定是奇数；

4、新字符串的回文子串一定以分隔符作为两边的边界，因此分隔符起到“哨兵”的作用。



五分钟学算法：原始字符串与新字符串的对应关系

第 2 步：计算辅助数组 p

辅助数组 `p` 记录了新字符串中以每个字符为中心的回文子串的信息。

手动的计算方法仍然是“中心扩散法”，此时记录以当前字符为中心，向左右两边同时扩散，记录能够扩散的最大步数。

以字符串 `"abbabb"` 为例，说明如何手动计算得到辅助数组 `p`，我们要填的就是下面这张表。

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12

第 1 行数组 `char` ：原始字符串**加上分隔符以后**的每个字符。

第 2 行数组 `index` ：这个数组是新字符串的索引数组，它的值是从 0 开始的索引编号。

- 我们首先填 `p[0]` 。

以 `char[0] = '#'` 为中心，同时向左边向右扩散，走 1 步就碰到边界了，因此能扩散的步数为 0 ，因此 `p[0] = 0` ；

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12
p	0												

- 下面填写 `p[1]` 。

以 `char[1] = 'a'` 为中心，同时向左边向右扩散，走 1 步，左右都是 `"#"` ，构成回文子串，于是再继续同时向左边向右边扩散，左边就碰到边界了，最多能扩散的步数”为 1，因此 `p[1] = 1` ；

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12

- 下面填写 `p[2]` 。

以 `char[2] = '#'` 为中心，同时向左边向右扩散，走 1 步，左边是 `"a"`，右边是 `"b"`，不匹配，最多能扩散的步数为 0，因此 `p[2] = 0`；

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12
p	0	1	0										

- 下面填写 `p[3]` 。

以 `char[3] = 'b'` 为中心，同时向左边向右扩散，走 1 步，左右两边都是 “#”，构成回文子串，继续同时向左边向右扩散，左边是 `"a"`，右边是 `"b"`，不匹配，最多能扩散的步数为 1，因此 `p[3] = 1`；

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12
p	0	1	0	1									

- 下面填写 `p[4]` 。

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12
p	0	1	0	1	4								

- 继续填完 p 数组剩下的部分。

分析到这里，后面的数字不难填出，最后写成如下表格：

Char	#	A	#	B	#	B	#	A	#	B	#	B	#
index	0	1	2	3	4	5	6	7	8	9	10	11	12
p	0	1	0	1	4	1	0	5	0	1	2	1	0

说明：有些资料将辅助数组 p 定义为回文半径数组，即 p[i] 记录了以新字符串第 i 个字符为中心的回文字符串的半径（包括第 i 个字符），与我们这里定义的辅助数组 p 有一个字符的偏差，本质上是一样的。

下面是辅助数组 p 的结论：辅助数组 p 的最大值是 5，对应了原字符串 "abbabb" 的 “最长回文子串”："bbabb"。这个结论具有一般性，即：

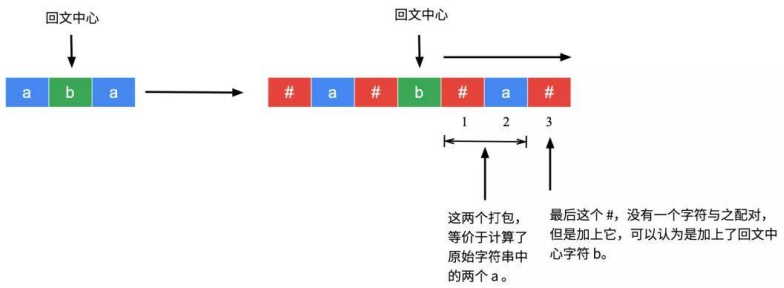
辅助数组 `p` 的最大值就是“最长回文子串”的长度

个最大值，并且记录最长回文子串。

简单说明一下这是为什么：

如果新回文子串的中心是一个字符，那么原始回文子串的中心也是一个字符，在新回文子串中，向两边扩散的特点是：“先分隔符，后字符”，同样扩散的步数因为有分隔符 # 的作用，在新字符串中每扩散两步，虽然实际上只扫到一个有效字符，但是相当于在原始字符串中相当于计算了两个字符。

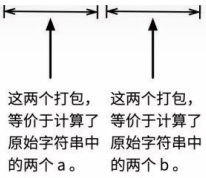
因为最后一定以分隔符结尾，还要计算一个，正好这个就可以把原始回文子串的中心算进去；



五分钟学算法：理解辅助数组的数值与原始字符串回文子串的等价性-1

如果新回文子串的中心是 # ，那么原始回文子串的中心就是一个“空隙”。在新回文子串中，向两边扩散的特点是：“先字符，后分隔符”，扩散的步数因为有分隔符 # 的作用，在新字符串中每扩散两步，虽然实际上只扫到一个有效字符，但是相当于在原始字符串中相当于计算了两个字符。

因此，“辅助数组 p 的最大值就是“最长回文子串”的长度”这个结论是成立的，可以看下面的图理解上面说的 2 点。



五分钟学算法：理解辅助数组的数值与原始字符串回文子串的等价性-2

写到这里，其实已经能写出一版代码。（注：本文的代码是结合了 LeetCode 第 5 题「最长回文子串」进行讲解）

参考代码

```
public class Solution {

    public String longestPalindrome(String
        int len = s.length();
        if (len < 2) {
            return s;
        }
        String str = addBoundaries(s, '#')
        int sLen = 2 * len + 1;
        int maxLen = 1;

        int start = 0;
        for (int i = 0; i < sLen; i++) {
            int curLen = centerSpread(str,
            if (curLen > maxLen) {
                maxLen = curLen;
                start = (i - maxLen) / 2;
            }
        }
        return s.substring(start, start +
    }

    private int centerSpread(String s, int
        // left = right 的时候，此时回文中心
        // right = left + 1 的时候，此时回文
        int len = s.length();
        int i = center - 1;
```

```
            j++;
            step++;
        }
        return step;
    }


    /**
     * 创建预处理字符串
     *
     * @param s       原始字符串
     * @param divide 分隔字符
     * @return 使用分隔字符处理以后得到的字符
     */
    private String addBoundaries(String s,
        int len = s.length();
        if (len == 0) {
            return "";
        }
        if (s.indexOf(divide) != -1) {
            throw new IllegalArgumentExcep
        }
        StringBuilder stringBuilder = new
        for (int i = 0; i < len; i++) {
            stringBuilder.append(divide);
            stringBuilder.append(s.charAt(
        }
        stringBuilder.append(divide);
        return stringBuilder.toString();
    }
}
```

复杂度分析

- 时间复杂度：$O(N^2)$，这里 N 是原始字符串的长度。新字符串的长度是 2 * N + 1，不计系数与常数项，因此时间复杂度仍为 $O(N^2)$。
- 空间复杂度：$O(N)$。

科学家的工作

上面的代码不太智能的地方是，对新字符串每一个位置进行中心扩散，会导致原始字符串的每一个字符被访问多次，一个比较极端的情况就是：`#a#a#a#a#a#a#a#` 。

事实上，计算机科学家 Manacher 就改进了这种算法，使得在填写新的辅助数组 `p` 的值的时候，能够参考已经填写过的辅助数组 `p` 的值，使得新字符串每个字符只访问了一次，整体时间复杂度由 $O(N^2)$ 改进到 $O(N)$ 。

具体做法是：在遍历的过程中，除了循环变量 `i` 以外，我们还需要记录两个变量，它们是 `maxRight` 和 `center` ，它们分别的含义如下：

maxRight

`maxRight` 表示记录当前向右扩展的最远边界，即从开始到现在使用“中心扩散法”能得到的回文子串，它能延伸到的最右端的位置 。

对于 `maxRight` 我们说明 3 点：

1. “向右最远”是在计算辅助数组 `p` 的过程中，向右边扩散能走的索引最大的位置，注意：得到一个 `maxRight` 所对应的回文子串，并不一定是当前得到的“最长回文子串”，很可能的一种情况是，某个回文子串可能比较短，但是它正好在整个字符串比较靠后的位置；
2. `maxRight` 的下一个位置可能是被程序看到的，停止的原因有 2 点：（1）左边界不能扩散，导致右边界受限制也不能扩散， `maxRight`

3. 为什么 maxRight 很重要？因为扫描是从左向右进行的， maxRight 能够提供的信息最多，它是一个重要的分类讨论的标准，因此我们需要一个变量记录它。

center

center 是与 maxRight 相关的一个变量，它是上述 maxRight 的回文中心的索引值。对于 center 的说明如下：

center 的形式化定义：

$$center = argmax\{x + p[x] \mid 0 \leq x < i\}$$

说明： x + p[x] 的最大值就是我们定义的 maxRight ， i 是循环变量， 0<= x< i 表示是在 i 之前的所有索引里得到的最大值 maxRight ，它对应的回文中心索引就是上述式子。

maxRight 与 center 的关系

maxRight 与 center 是一一对应的关系，即一个 center 的值唯一对应了一个 maxRight 的值；因此 maxRight **与** center **必须要同时更新**。

下面的讨论就根据循环变量 i 与 maxRight 的关系展开讨论：

maxRight ；

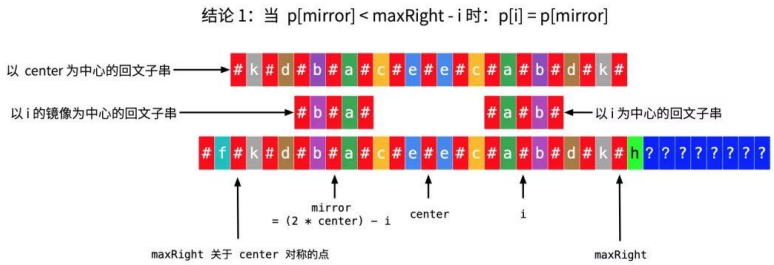
情况 2：当 i < maxRight 的时候，根据新字符的回文子串的性质，循环变量关于 center 对称的那个索引（记为 mirror ）的 p 值就很重要。

我们先看 mirror 的值是多少，因为 center 是中心， i 和 mirror 关于 center 中心对称，因此 (mirror + i) / 2 = center ，所以 mirror = 2 * center - i 。

根据 p[mirror] 的数值从小到大，具体可以分为如下 3 种情况：

情况 2（1）： p[mirror] 的数值比较小，不超过 maxRight - i 。

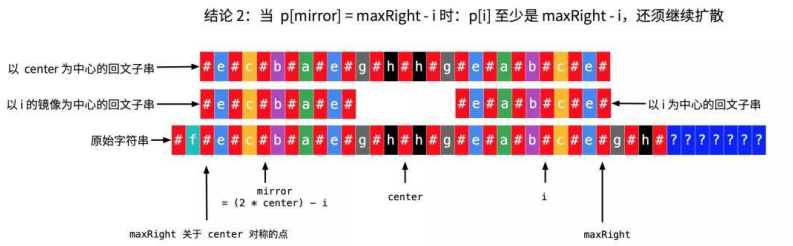
说明： maxRight - i 的值，就是从 i 关于 center 的镜像点开始向左走（不包括它自己），到 maxRight 关于 center 的镜像点的步数



五分钟学算法：Manacher 算法分类讨论情况 2（1）

从图上可以看出，由于“以 center 为中心的回文子串”的对称性，导致了“以 i 为中心的回文子串”与“以 center 为中心的回文子串”也具有对称性，“以 i 为中心的回文子串”与“以 center 为中心的回

情况 2（2）：`p[mirror]` 的数值恰好等于 `maxRight - i`。



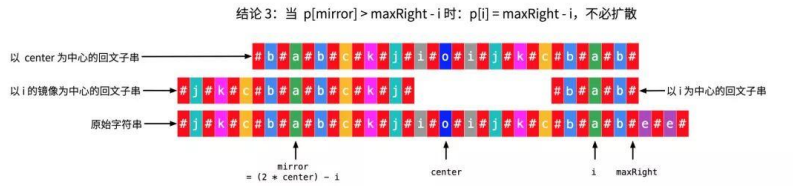
五分钟学算法：Manacher 算法分类讨论情况 2（2）

说明：仍然是依据“以 `center` 为中心的回文子串”的对称性，导致了“以 `i` 为中心的回文子串”与“以 `center` 为中心的回文子串”也具有对称性。

1. 因为靠左边的 `f` 与靠右边的 `g` 的原因，导致“以 `center` 为中心的回文子串”不能继续扩散；
2. 但是“以 `i` 为中心的回文子串”还可以继续扩散。

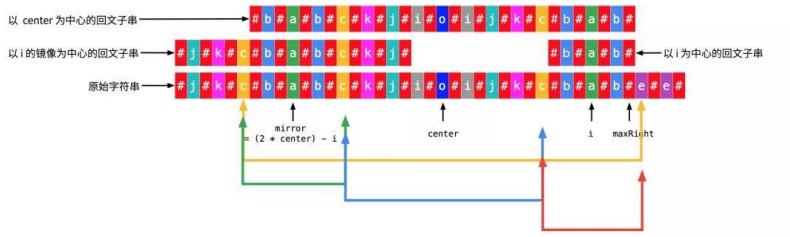
因此，可以先把 `p[mirror]` 的值抄过来，然后继续“中心扩散法”，继续增加 `maxRight`。

情况 2（3）：`p[mirror]` 的数值大于 `maxRight - i`。



五分钟学算法：Manacher 算法分类讨论情况 2（3）

下面证明， `p[i] = maxRight - i` ，证明的方法还是利用三个回文子串的对称性。



五分钟学算法：Manacher 算法分类讨论情况 2（3）的证明

① 由于“以 `center` 为中心的回文子串”的对称性，黄色箭头对应的字符 `c` 和 `e` 一定不相等；

② 由于“以 `mirror` 为中心的回文子串”的对称性，绿色箭头对应的字符 `c` 和 `c` 一定相等；

③ 又由于“以 `center` 为中心的回文子串”的对称性， 蓝色箭头对应的字符 `c` 和 `c` 一定相等；

推出“以 `i` 为中心的回文子串”的对称性， 红色箭头对应的字符 `c` 和 `e` 一定不相等。

因此， `p[i] = maxRight - i` ，不可能再大。上面是因为我画的图，可能看的朋友会觉得理所当然。事实上，可以使用反证法证明：

如果“以 `i` 为中心的回文子串”再向两边扩散的两个字符 `c` 和 `e` 相等，就能够推出黄色、绿色、蓝色、红色箭头所指向的 8 个变量的值都相等，此时“以 `center` 为中心的回文子串”就可以再同时向左边和右边扩散 1 格，与 `maxRight` 的最大性矛盾。

是保守的，即二者之中较小的那个值：

```
p[i] = min(maxRight - i, p[mirror]);
```

参考代码

```
public class Solution {

    public String longestPalindrome(String
        // 特判
        int len = s.length();
        if (len < 2) {
            return s;
        }

        // 得到预处理字符串
        String str = addBoundaries(s, '#')
        // 新字符串的长度
        int sLen = 2 * len + 1;

        // 数组 p 记录了扫描过的回文子串的信
        int[] p = new int[sLen];

        // 双指针，它们是一一对应的，须同时更
        int maxRight = 0;
        int center = 0;

        // 当前遍历的中心最大扩散步数，其值等
        int maxLen = 1;
        // 原始字符串的最长回文子串的起始位置
        int start = 0;

        for (int i = 0; i < sLen; i++) {
            if (i < maxRight) {
                int mirror = 2 * center -
                // 这一行代码是 Manacher 算
                p[i] = Math.min(maxRight -
```

```
            int right = i + (i - p[i]);

            // left >= 0 && right < sLen 
            // str.charAt(left) == str.cha
            while (left >= 0 && right < sL
                p[i]++;
                left--;
                right++;

            }
            // 根据 maxRight 的定义，它是遍
            // 如果 maxRight 的值越大，进入
            if (i + p[i] > maxRight) {
                // maxRight 和 center 需要
                maxRight = i + p[i];
                center = i;
            }
            if (p[i] > maxLen) {
                // 记录最长回文子串的长度和
                maxLen = p[i];
                start = (i - maxLen) / 2;
            }
        }
        return s.substring(start, start +
    }


    /**
     * 创建预处理字符串
     *
     * @param s       原始字符串
     * @param divide 分隔字符
     * @return 使用分隔字符处理以后得到的字
     */
    private String addBoundaries(String s,
        int len = s.length();
        if (len == 0) {
            return "";
        }
        if (s.indexOf(divide) != -1) {
            throw new IllegalArgumentExcep
        }
        StringBuilder stringBuilder = new
        for (int i = 0; i < len; i++) {
            stringBuilder.append(divide);
            stringBuilder.append(s.charAt(
        }
        stringBuilder.append(divide);
        return stringBuilder.toString();
```

复杂度分析

- 时间复杂度：$O(N)$ ，由于 Manacher 算法只有在遇到还未匹配的位置时才进行匹配，已经匹配过的位置不再匹配，因此对于字符串 S 的每一个位置，都只进行一次匹配，算法的复杂度为 $O(N)$ 。
- 空间复杂度：$O(N)$ 。

后记

Manacher 算法我个人觉得没有必要记住，如果真有遇到，查资料就可以了。

最后，再推荐一篇用漫画的形式讲解马拉车算法的文章：**漫画：如何找到字符串中的最长回文子串？**

—

以上，便是今日分享，觉得不错，还请点个在看，谢谢~

推荐阅读：

有了这套模板，女朋友再也不用担心我刷不动 LeetCode 了

图解 LeetCode 第 421 题：数组中两个数的最大异或值

互联网人职业发展之路：三年升高工，七年做架构，十年送外卖

我的个人博客大改版啦，欢迎点击阅读原文进行访问~

分享到：

上一篇
五分钟学算法之经典算法题：二分查找

下一篇
五分钟学算法之经典算法题：排序算法（某东算法工程师比赛）

相关推荐

程序员吴师兄

独乐乐不如众乐乐，如何装逼的求众数

程序员吴师兄

使用位运算处理一道难题：获取所有钥匙的最短路径

程序员吴师兄

算法科普：有趣的游程编码

程序员吴师兄

数据结构与算法： 三十张图弄懂「图的两种遍历方式」