获取客户端真实IP winrar优化

Manacher's ALGORITHM: O(n)时间求字符串的最长回文子串

OCT 13

felix021 @ 2011-10-13 12:00 [IT » 程序设计] 评论(45) , 引用(0) , 阅读(120527) | Via 本站原创

大 | 中 | 小

Translated to ENGLISH VERSION

源于这两篇文章:
http://blog.csdn.net/ggggiqnypgjg/article/details/6645824
http://zhuhongcheng.wordpress.com/2009/08/02/a-simple-linear-time-algorithm-for-finding-longest-palindrome-sub-string/

这个算法看了三天，终于理解了，在这里记录一下自己的思路，免得以后忘了又要想很久- -.

首先用一个非常巧妙的方式，将所有可能的奇数/偶数长度的回文子串都转换成了奇数长度：在每个字符的两边都插入一个特殊的符号。比如 abba 变成 #a#b#b#a#， aba变成 #a#b#a#。 为了进一步减少编码的复杂度，可以在字符串的开始加入另一个特殊字符，这样就不用特殊处理越界问题，比如$#a#b#a#
（注意，下面的代码是用C语言写就，由于C语言规范还要求字符串末尾有一个'\0'所以正好OK，但其他语言可能会导致越界）。

下面以字符串12212321为例，经过上一步，变成了 S[] = "$#1#2#2#1#2#3#2#1#";

然后用一个数组 P[i] 来记录以字符S[i]为中心的最长回文子串向左/右扩张的长度（包括S[i]，也就是把该回文串"对折"以后的长度），比如S和P的对应关系：

```
S  #  1  #  2  #  2  #  1  #  2  #  3  #  2  #  1  #
P  1  2  1  2  5  2  1  4  1  2  1  6  1  2  1  2  1
(p.s. 可以看出，P[i]-1正好是原字符串中回文串的总长度)
```

那么怎么计算P[i]呢？该算法增加两个辅助变量（其实一个就够了，两个更清晰）id和mx，其中 id 为已知的 {右边界最大} 的回文子串的中心，mx则为id+P[id]，也就是这个子串的右边界。

然后可以得到一个非常神奇的结论，这个算法的关键点就在这里了：如果mx > i，那么P[i] >= MIN(P[2 * id - i], mx - i)。就是这个串卡了我非常久。实际上如果把它写得复杂一点，理解起来会简单很多：

```
//记j = 2 * id - i，也就是说 j 是 i 关于 id 的对称点(j = id - (i - id))
if (mx - i > P[j])
    P[i] = P[j];
```

联系我

i@felix021.com

最新评论

重现了sweat
填一下最后的问题：如果是一个空串，指..
go1.12，测试了例子，没有重现呢..
支持分享知识
love
cry
我就是从您那篇go的文章开始关注到您的..
精彩!
感觉面试问这些有一些偏了，这种排查思..
膜拜大佬，还有很多需要学习啊。..

最新日志

https耗时分析
搞事：代码找茬
又是面试题？对，合并有序序列。..
Go: 关于锁的1234
踩坑记： Go 服务灵异 panic
生人勿近之Linux里养僵尸
关于写作的一点思考
Linux下删点日志也能搞死人
踩坑记#2：Go服务锁死
golang: bufio.Scanner 的坑

分类

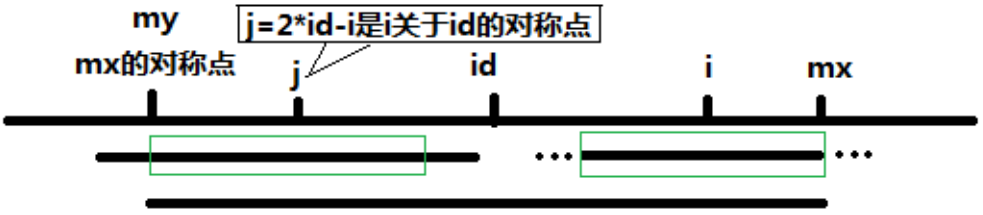
随想 [24] RSS

```
else /* P[j] >= mx - i */
    P[i] = mx - i; // P[i] >= mx - i，取最小值，之后再匹配更新。
```

当然光看代码还是不够清晰，还是借助图来理解比较容易。

当 mx - i > P[j] 的时候，以S[j]为中心的回文子串包含在以S[id]为中心的回文子串中，由于 i 和 j 对称，以S[i]为中心的回文子串必然包含在以S[id]为中心的回文子串中，所以必有 P[i] = P[j]，见下图。

当 P[j] >= mx - i 的时候，以S[j]为中心的回文子串不一定完全包含于以S[id]为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以S[i]为中心的回文子串，其向右至少会扩张到mx的位置，也就是说 P[i] >= mx - i。至于mx之后的部分是否对称，就只能老老实实去匹配了。



对于 mx <= i 的情况，无法对 P[i]做更多的假设，只能P[i] = 1，然后再去匹配了。

于是代码如下：

```
//输入，并处理得到字符串s
int p[1000], mx = 0, id = 0;
memset(p, 0, sizeof(p));
for (i = 1; s[i] != '\0'; i++) {
    p[i] = mx > i ? min(p[2*id-i], mx-i) : 1;
    while (s[i + p[i]] == s[i - p[i]]) p[i]++;
    if (i + p[i] > mx) {
        mx = i + p[i];
        id = i;
    }
}
//找出p[i]中最大的
```

OVER.

#UPDATE@2013-08-21 14:27
@zhengyuee 同学指出，由于 P[id] = mx，所以 S[id-mx] != S[id+mx]，那么当 P[j] > mx - i 的时候，可以肯定 P[i] = mx - i ，不需要再继续匹配了。不过在具体

IT [786] RSS
» Blockchain [3] RSS
» 云 [3] RSS
» 操作系统 [141] RSS
» shell [2] RSS
» Python [14] RSS
» 探索设计模式 [5] RSS
» 软件 [93] RSS
» 硬件 [43] RSS
» 手机 [12] RSS
» 程序设计 [176] RSS
» 网络 [206] RSS
» 数据库 [21] RSS
» 病毒 [8] RSS
» 其他 [59] RSS

其他

登入
注册
RSS: 日志 | 评论
编码：UTF-8
XHTML 1.0

统计

访问次数 4724641
今日访问 491
日志数量 812
评论数量 2466
引用数量 1
留言数量 151
注册用户 419
在线人数 27

链接

默认链接组

» WHU微软俱乐部
» 谁见过风？（国际版）

朋友们的据点

» FOUR
» 笨狗又一窝
» czyhd's Blog
» Kid的原创Blog
» [GCC]Feli
» dutor

实现的时候即使不考虑这一点，也只是多一次匹配（必然会fail），但是却要多加一个分支，所以上面的代码就不改了。

欢迎扫码关注：

转载请注明出自 **https://www.felix021.com/blog/read.php?2040** ，如是转载文则注明原出处，谢谢:)
RSS订阅地址: **https://www.felix021.com/blog/feed.php** 。

苟富贵 2020-7-7 23:22

苟富贵 2020-7-7 23:21

啊啊 2020-2-1 20:08

为什么不用js写

kjkjk 2019-11-11 04:37

1233211234567 ✉ 2019-6-15 11:11

ruibinhong ✉ 2019-4-21 14:53

给定输入 character,你的算法输出是3,4,得到的回文串是ara;但有的要求输出字符串里所有能构成的回文串的字符组合（保持相对顺序一致），并计算里面最长的，这种条件下最长的回文串是carac

> **felix021 回复于 2019-4-22 18:16**

> 嗯，那是另一个问题了，相对简单一些

sarag 2019-3-3 13:35

厉害了，看一遍就懂了

..... 2019-2-12 11:52

小火汁 2018-11-7 00:08

newbee

ahlixinjie 2018-10-28 10:14

如果没看懂的话建议看一下英文版，每一步怎么来怎么去的话写的非常清楚
https://articles.leetcode.com/longest-palindromic-substring-part-ii/

Entropy 2018-10-21 05:26

感谢楼主 果然一遍就懂~

dengjiangzhou ✉ 2018-10-14 14:35

倒着讲解的，讲的很好

3344aaxgl 2018-10-12 11:29

确实一遍就懂了

657657 2018-9-28 03:18

234324923432 2018-9-25 12:53

无 2018-8-22 21:14

给跪 2018-8-20 07:50

大家都看懂了，厉害厉害，已放弃，打扰了 

flashhu 2018-1-28 22:36

讲的很清楚，一下就懂了，感谢！

Coder 2017-11-3 03:22

解釋的清楚明白多謝.供大家參考：articles.leetcode.com的Longest Palindromic Substring Part II文後的useful links有提到這篇的網址.在youtube-IDeserve的Longest Palindromic Substring O(N) Manacher's Algorithm的useful references可間接連到這網頁.

pikachu 2017-10-28 14:00

memset应该初始化为1，id和mx也应该为1啊

Cava 2017-10-25 22:32

不错！思路清晰，代码简洁，非常容易理解。不过感觉写成这个样子更符合我的思维方式：mx > P[j] + i 受教了，感谢~~

ff 2017-10-11 10:09

prince123 2017-10-7 03:11

写得真好～（因为我看懂了。。

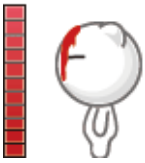
han0111 2017-9-19 13:34

很好

端木wx 2017-8-29 15:50

写的很不错，看了一遍就懂了

123 2017-8-21 15:39



cxh007 2017-8-9 17:56

555 2017-7-26 09:40

文章写的不错，很有助于理解此算法，是我目前看到的解释最清楚地。最后 @zhengyuee 同学指出的为题也确实值得考虑，从更深层次解释理解这个算法的复杂度为啥是O（n），但是，对于这种情况却考虑的不完整，如果以id为中心的回文字符串左边界已经到最边上，这样的话就需要再继续检测，不能直接 P[i] = mx - i 。可能说的不太明白，大概就这个意思。最后，感谢作者。

mayahw 2017-6-19 17:45

这个是我目前读到的最清楚的描述了， 感谢。

ctg诺 2016-10-10 10:17

0x3A2B 2016-7-29 10:06

"//记j = 2 * id - i，也就是说 j 是 i 关于 id 的对称点(j = id + (id - i))"这里应该是"j = id - (i - id)"吧原文写的是"int i_mirror = 2*C-i; // equals to i' = C - (i-C)"

> **felix021 回复于 2016-7-30 21:50**
>
> 嗯，是的，j在id的左边，j = id - (i - id) 这个写法更合理。

曙光 2016-1-20 15:26

真棒！不但表述棒，代码也写得简洁。受教，多谢！

zhs 2015-11-2 19:59

“其中id表示最大回文子串中心的位置，mx则为id+P[id]，也就是最大回文子串的边界。”这句话有误啊！并不是最大回文子串，只是这个回文子串的右边界最靠右而已，被你误导了，卡TLE卡了几天。。。

> **felix021 回复于 2015-11-5 09:33**
>
> sorry 表述有问题:D

Wei 2015-10-25 08:20

这个算法不好的地方就是要改原始数据。但如果用两个端点之和来存储最大的子回文串的长度的话，就不需要改原始数据了。

6 2015-10-22 22:06

1

exr 2015-10-14 16:38

winterstasis 2015-9-24 02:22

有一个问题是如果字符串里面本身就含有#怎么办呢

> **felix021 回复于 2015-10-16 11:44**
>
> 换另一个□字符就好了，比如ASCII 01

winterstasis 2015-9-23 13:38

感觉写的简单易懂，真棒

孙小九 2015-8-25 15:23

文中提到的：#UPDATE@2013-08-21 14:27@zhengyuee 同学指出，由于 P[id] = mx，所以 S[id-mx] != S[id+mx]，那么当 P[j] > mx - i 的时候，可以肯定 P[i] = mx - i，不需要再继续匹配了。不过在具体实现的时候即使不考虑这一点，也只是多一次匹配（必然会fail），但是却要多加一个分支，所以上面的代码就不改了。是有问题的考虑到特殊情况，当mx-i == i-id的时候（即i正好是[id,mx]的中点的时候），不能肯定P[i] = mx - i，仍然需要扩大范围继续搜索

lonriyao ✉ 2015-4-4 15:28

```
int palinLen(const char *s)
{
    if (!s) {
        return 0;
    }
    char *front,*back, *head =s;
                        //这里的front back分别指向 s前后
                        //head记录s的开始位置 防止front访问非法元素

    int len = 0, i ;    //len记录最后的长度 i只是记录一次的长度

    s++;                //第一个肯定不是 所以s加1
    while(*s){          //以s为中心进行 front back分别指向两边 查找
        front = s - 1;
        back =  s + 1;

        i = 0;          //从这里开始计数

        while((&*head != &*front) && *back  ){
                            //这里有问题front可能访问非法区域
                            //可以使用head指向的位置的地址来判断
                            //是否front越界

            if(*front == *back){
                i++;
            }else{
```

```
                break;
            }
            back++;
            front--;
        }

        if (len < i) {
            len = i;
        }

        s++;
    }
    return len;
}
```

这个算法 行吗？？ 这是我自己写的不知道 对不对

> **felix021 回复于 2015-4-10 20:11**
>
> 多造几个case试试看喽

newwayy 2015-3-31 20:48

while (s[i + p[i]] == s[i - p[i]]) p[i]++;这句话是会下溢或者上溢， 除非限定了s中字符的取值范围（如alnum）。 否则， 不管构造多精美的卫兵字符， 都可能溢出，如：^#1#2# ==> ^###^# 将1替换成#， 将2替换成^ 下溢；
#1#2# ==> ##### 将1、2都替换成#^1#2# ==> ^##^#

> **felix021 回复于 2015-4-1 09:17**
>
> 嗯，是的，严谨地说应该加上一个边界判断的条件，不过这个并不影响对算法的理解吧:)

maplainfly ✉ 2015-2-8 12:09

while (s[i + p[i]] == s[i - p[i]]) p[i]++;后面应该有个p[i]--吧?

> **felix021 回复于 2015-2-9 15:57**
>
> 好像不用哟。

大头 2014-11-13 04:19

你的测试过了？ while (s[i + p[i]] == s[i - p[i]]) p[i]++;s[i+p[i]]可能会越界吧

> **felix021 回复于 2014-11-14 16:32**
>
> 不会越界的

迷失的杰克 2014-6-10 18:35

如果P[i]包括S[i]，那么为啥 P[1] = 2 (i 从0开始) 呢？应该等于3吧，因为 S[0] = \'#\'S[1] = \'1\'S[2] = \'#\'从S[1]向左向右扩张，还包括S[1]的话，那么P[1] = 3?如果我说的正确，那么例子中的后面有好几个错误

> **felix021 回复于 2014-6-15 23:15**
>
> “P[i] 来记录以字符S[i]为中心的最长回文子串向左/右扩张的长度（包括S[i]）”P[1]对应的回文串是 "#1#"，从S[1]起向左/右扩张1个字符，加上S[1]，就是2个，所以P[1] = 2.

wayne 2011-10-13 21:17

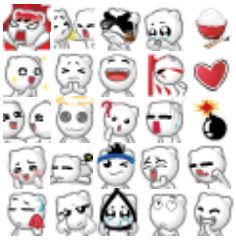
瞄了一眼，有点像用后缀数组的那种方法的感觉

> **felix021 回复于 2011-10-14 08:22**
>
> 曾经试图去看后缀数组，最后看晕了-。-

分页： 1/1 1

发表评论

表情



昵称 密码 *非必须

网址 电邮

[注册]

B I U

打开HTML
打开UBB
打开表情
隐藏
记住我

提交 重置