16 最长回文子串

更新时间：2019-08-26 10:07:23

> 我们有力的道德就是通过奋斗取得物质上的成功；这种道德既适用于国家，也适用于个人。
>
> ——罗素

刷题内容

> 难度：Medium
>
> 原题链接：https://leetcode-cn.com/problems/longest-palindromic-substring/

内容描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。

示例 2：
输入: "cbbd"
输出: "bb"

题目详解

- 回文的概念我们已经在 2.3 小节中介绍过了，指的是一个字符串 `a` 和其逆序 `a[::-1]` 是相等的；
- 题目中要求我们输出的是最长的那个回文子串，而不是子序列；
- 这样的子串可能会有多个，题目说了，任意返回其中一个即可。

解题方案

思路 1：时间复杂度: O(N^2) 空间复杂度: O(N^2)

我们可以用 dp 的方式来处理这道题。

> 注：dp 解法可参看 2.5 小节最长公共前缀。

- `dp[i][j] = True` 代表 `s[i:j+1]` 是回文；
- `dp[i][j] = False` 代表 `s[i:j+1]` 不是回文；
- 那么自然而然地，如果 `s[i-1] == s[j+1]` 的时候，如果 `dp[i][j] = True` 的话， `dp[i-1 ][j+1]` 也为True。

Python beats 28.11%

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        res = ''
        dp = [[False] * len(s) for i in range(len(s))]
        for i in range(len(s)-1, -1, -1):
            for j in range(i, len(s)):
                # 如果s[i]和s[j]相等，并且要么i和j之间只有一个字符，要么中间的子串也是回文子串，此时我们的dp[i][j]就是True了
                dp[i][j] = (s[i] == s[j]) and (j - i < 3 or dp[i+1][j-1])
                if dp[i][j] and (j - i + 1 > len(res)): # 此时的回文子串长度大于res
                    res = s[i:j+1]
        return res
```

Java beats 32.14%

```java
class Solution {
    public String longestPalindrome(String s) {
        int length = s.length();
        String res = "";
        // dp[i][j] 用来表示 i 到 j 的字串是否是回文串
        boolean[][] dp = new boolean[length][length];
        for (int i = length - 1; i >= 0; i--) {
            for (int j = i; j < length; j++) {
                dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j - 1]);
                // dp[i][j] 用来判断是否为回文串， j - i + 1 > res.length() 用来判断是否是最长回文串
                if (dp[i][j] && (res == null || j - i + 1 > res.length())) {
                    res = s.substring(i, j + 1);
                }
            }
        }
        return res;
    }
}
```

c++ beats 39.37%

```
bool dp[1010][1010];
class Solution {
public:
    string longestPalindrome(string s) {
        int left = 0;
        int right = 0;
        for (int i = s.size() - 1;i >= 0;i--) {
            for (int j = i;j < s.size();j++) {
                // 如果s[i]和s[j]相等，并且要么i和j之间只有一个字符，要么中间的子串也是回文子串，此时我们的dp[i][j]就是True了
                dp[i][j] = (s[i] == s[j]) && (j - i < 3 || dp[i+1][j-1]);
                if (dp[i][j] && right - left < j - i) {
                    left = i;
                    right = j;
                }
            }
        }
        return s.substr(left, right - left + 1);
    }
};
```

Go beats 48.32%

```
func longestPalindrome(s string) string {
 dp := make([][]bool, len(s))
 for i := 0; i < len(s); i += 1 {
  dp[i] = make([]bool, len(s))
 }
 res := ""
 for i := len(s) - 1; i >= 0; i -= 1 {
  for j := i; j < len(s); j += 1 {
          // 如果s[i]和s[j]相等，并且要么i和j之间只有一个字符，要么中间的子串也是回文子串，此时我们的dp[i][j]就是True了
   dp[i][j] = (s[i] == s[j]) && (j - i < 3 || dp[i+1][j-1])
   if dp[i][j] && j - i + 1 > len(res) { // 此时的回文子串长度大于res
    res = s[i:j+1]
   }
  }
 }
 return res
}
```

思路 2：时间复杂度：O(N^2) 空间复杂度：O(1)

回文字符串长度为奇数和偶数是不一样的：

1. 奇数：`'xxx s[i] xxx'`，比如 `'abcdcba'`；
2. 偶数：`'xxx s[i] s[i+1] xxx'`，比如 `'abcddcba'`。

我们区分回文字符串长度为奇数和偶数的情况，然后依次把每一个字符当做回文字符串的中间字符，向左右扩展到满足回文的最大长度，不停更新满足回文条件的最长子串的左右 `index`：`l` 和 `r`，最后返回 `s[l:r+1]` 即为结果。

下面来看具体代码：

Python beats 58.44%

```python
class Solution:
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        l = 0 # left index of the current substring
        r = 0 # right index of the current substring
        max_len = 0 # length of the longest palindromic substring for now
        n = len(s)
        for i in range(n):
            # odd case: 'xxx s[i] xxx', such as 'abcdcba'
            for j in range(min(i+1, n-i)): # 向左最多移动 i 位，向右最多移动 (n-1-i) 位
                if s[i-j] != s[i+j]: # 不对称了就不用继续往下判断了
                    break
                if 2 * j + 1 > max_len: # 如果当前子串长度大于目前最长长度
                    max_len = 2 * j + 1
                    l = i - j
                    r = i + j

            # even case: 'xxx s[i] s[i+1] xxx', such as 'abcddcba'
            if i+1 < n and s[i] == s[i+1]:
                for j in range(min(i+1, n-i-1)): # s[i]向左最多移动 i 位，s[i+1]向右最多移动 [n-1-(i+1)] 位
                    if s[i-j] != s[i+1+j]: # 不对称了就不用继续往下判断了
                        break
                    if 2 * j + 2 > max_len:
                        max_len = 2 * j + 2
                        l = i - j
                        r = i + 1 + j
        return s[l:r+1]
```

Java beats 52.30%

```java
class Solution {
    public String longestPalindrome(String s) {
        if(s.equals("")){
            return "";
        }
        int l = 0; // left index of the current substring
        int r = 0; // right index of the current substring
        int maxLength = 0; // length of the longest palindromic substring for now
        int n = s.length();
        for (int i = 0; i < n; i++) {
            // odd case: 'xxx s[i] xxx', such as 'abcdcba'
            // 向左最多移动 i 位，向右最多移动 n - 1 - i 位
            for (int j = 0; j < Math.min(i + 1, n - i); j++) {
                // 不对称了就不用继续往下判断了
                if (s.charAt(i - j) != s.charAt(i + j)) {
                    break;
                }
                // 如果当前子串长度大于目前最长长度
                if (2 * j + 1 > maxLength) {
                    maxLength = 2 * j + 1;
                    l = i - j;
                    r = i + j;
                }
            }
            // even case: 'xxx s[i] s[i+1] xxx', such as 'abcddcba'
            if (i + 1 < n && s.charAt(i) == s.charAt(i + 1)) {
                // s[i]向左最多移动 i 位，s[i+1]向右最多移动 [n-1-(i+1)] 位
                for (int j = 0; j < Math.min(i + 1, n - i - 1); j++ ) {
                    // 不对称了就不用继续往下判断了
                    if (s.charAt(i - j) != s.charAt(i + 1 + j)) {
                        break;
                    }
                    if (2 * j + 2 > maxLength) {
                        maxLength = 2 * j + 2;
                        l = i - j;
                        r = i + 1 + j;
                    }
                }
            }
        }
        return s.substring(l,r + 1);
    }
}
```

c++ beats 82.83%

```
class Solution {
public:
    string longestPalindrome(string s) {
        if(s.length() == 0){
            return "";
        }
        int l = 0; // left index of the current substring
        int r = 0; // right index of the current substring
        int maxLength = 0; // length of the longest palindromic substring for now
        int n = s.length();
        for (int i = 0; i < n; i++) {
            // odd case: 'xxx s[i] xxx', such as 'abcdcba'
            // 向左最多移动 i 位，向右最多移动 n - 1 - i 位
            for (int j = 0; j < min(i + 1, n - i); j++) {
                // 不对称了就不用继续往下判断了
                if (s[i - j] != s[i + j]) {
                    break;
                }
                // 如果当前子串长度大于目前最长长度
                if (2 * j + 1 > maxLength) {
                    maxLength = 2 * j + 1;
                    l = i - j;
                    r = i + j;
                }
            }
            // even case: 'xxx s[i] s[i+1] xxx', such as 'abcddcba'
            if (i + 1 < n && s[i] == s[i+1]) {
                // s[i]向左最多移动 i 位，s[i+1]向右最多移动 [n-1-(i+1)] 位
                for (int j = 0; j < min(i + 1, n - i - 1); j++ ) {
                    // 不对称了就不用继续往下判断了
                    if (s[i - j] != s[i + 1 + j]) {
                        break;
                    }
                    if (2 * j + 2 > maxLength) {
                        maxLength = 2 * j + 2;
                        l = i - j;
                        r = i + 1 + j;
                    }
                }
            }
        }
        return s.substr(l, r - l + 1);
    }
};
```

Go beats 44.74%

```go
func longestPalindrome(s string) string {
    // 当s=""时，此时l,r = 0,0，golang不支持rindex大于底层array的长度
    // 所以我们要提前处理
    if s == "" {
        return ""
    }
    // l: left index of the current substring
    // r: right index of the current substring
    // maxLen: length of the longest palindromic substring for now
    l, r, maxLen, n := 0, 0, 0, len(s)
    for i := 0; i < n; i++ {
        // odd case: 'xxx s[i] xxx', such as 'abcdcba'
        for j := 0; j < int(math.Min(float64(i+1), float64(n-i))); j++ { // 向左最多移动 i 位，向右最多移动 (n-1-i) 位
            if s[i-j] != s[i+j] { // 不对称了就不用继续往下判断了
                break
            }
            if 2 * j + 1 > maxLen { // 如果当前子串长度大于目前最长长度
                maxLen = 2 * j + 1
                l = i - j
                r = i + j
            }
        }
        // even case: 'xxx s[i] s[i+1] xxx', such as 'abcddcba'
        if i + 1 < n && s[i] == s[i+1] {
            // s[i]向左最多移动 i 位，s[i+1]向右最多移动 [n-1-(i+1)] 位
            for j := 0; j < int(math.Min(float64(i+1), float64(n-i-1))); j++ {
                if s[i-j] != s[i+1+j] { // 不对称了就不用继续往下判断了
                    break
                }
                if 2 * j + 2 > maxLen {
                    maxLen = 2 * j + 2
                    l = i - j
                    r = i + 1 + j
                }
            }
        }
    }
    return s[l:r+1]
}
```

思路 3：时间复杂度：O(N) 空间复杂度：O(N)

Manacher算法

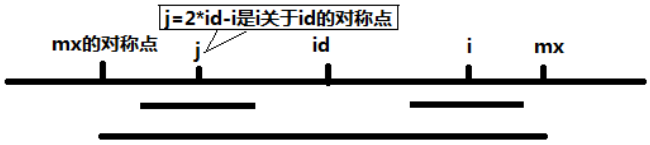
useful link

Manacher 算法增加两个辅助变量 id 和 mx，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。得到一个很重要的结论：

如果 mx > i，那么 P[i] >= Min(P[2 * id - i], mx - i) . 为什么这样说呢，下面解释：

下面，令 j = 2*id - i，也就是说 j 是 i 关于 id 的对称点。

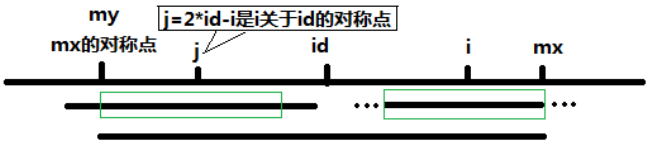
- 当 mx - i > P[j] 的时候，以 S[j] 为中心的回文子串包含在以 S[id] 为中心的回文子串中，由于 i 和 j 对称，以 S[i] 为中心的回文子串必然包含在以 S[id] 为中心的回文子串中，所以必有 P[i] = P[j]；



- 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，再具体匹配。所以`P[i] >= Min(P[2 * id - i], mx - i)`，因为以`j`为中心的绘回文子串的左边界可能会比`mx`关于`id`的对称点要大，此时只能证明`P[i]=P[j]`。

下面的图来源于上面的链接



- 此外，对于`mx <= i`的情况，因为无法对`P[i]`做更多的假设，只能让`P[i] = 0`，然后再去匹配。

在下面的程序中我的`P数组`保存的是以当前字符为回文子串中心时，该回文子串的长度（不包含当前字符自身）。

简单地用一个小例子来解释：原字符串为`'qacbcaw'`，一眼就可以看出来最大回文子串是`'acbca'`，
下面是我做的表格。

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Modified-string	^	#	q	#	a	#	c	#	b	#	c	#	a	#	w	#	$
P-array	0	0	1	0	1	0	1	0	5	0	1	0	1	0	1	0	0

所以最终代码中的`max_i`就是字符`'b'`所对应的`index8`，`start`的值就是`(max_i - P[max_i] - 1) // 2 = 1`,最终输出结果为`s[1:6]`,即`acbca'`

Python beats 92.24%

```python
class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        def preProcess(s):
            if not s:
                return ['^', '&']
            T = ['^']
            for i in s:
                T += ['#', i]
            T += ['#', '$']
            return T

        T = preProcess(s)
        P = [0] * len(T)
        id, mx = 0, 0
        for i in range(1, len(T)-1):
            j = 2 * id - i
            if mx > i:
                P[i] = min(P[j], mx-i) # 暂时先赋值，后面还有一个 while 循环接着判断的
            else:
                P[i]= 0
            while T[i+P[i]+1] == T[i-P[i]-1]:
                P[i] += 1
            if i + P[i] > mx:
                id, mx = i, i + P[i]
        max_i = P.index(max(P))    # 保存的是当前最大回文子串中心位置的index
        start = (max_i - P[max_i] - 1) // 2
        res = s[start:start+P[max_i]]
        return res
```

c++ beats 92.56%

```
class Solution {
public:
    string longestPalindrome(string s) {
        string data = "#";
        for (int i = 0;i < s.size();i++) {
            data.push_back(s[i]);
            data.push_back('#');
        }
        //半径
        vector<int> rad(data.size(), 0);
        int id = 0;
        int mx = 0;
        for (int i = 1;i < data.size();i++) {
            int last = 0;
            if (i > mx) {
                last = i;
            } else {
                if (rad[2 * id - i] < mx - i) {
                    //2*id-i为中心的最大回文被以id为中心的最大回文所覆盖，没必要继续扩展下去，直接返回
                    rad[i] = rad[2 * id - i];
                    continue;
                } else {
                    last = mx;
                }
            }
            //继续扩展
            while (last + 1 < data.size() && 2 * i - (last + 1) >= 0 && data[last + 1] == data[2 * i - (last + 1)]) {
                last++;
            }
            rad[i] = last - i;
            id = i;
            mx = last;
        }
        int left = 0, right = 0;
        for (int i = 0;i < data.size();i++) {
            //因为有#的存在，i-rad[i]必是#，也就是偶数下标，i-rad[i]+1对应的必是字母，所以(i-rad[i]+1-1)/2就是原来字母的位置
            int temp_left = (i - rad[i]) / 2;
            //同理
            int temp_right = (i + rad[i] - 2) / 2;
            if (temp_left < temp_right && right - left < temp_right - temp_left) {
                left = temp_left;
                right = temp_right;
            }
        }
        return s.substr(left, right - left + 1);
    }
};
```

Java beats 46.76%

```java
class Solution {
    public String longestPalindrome(String s) {
        String data = "#";
        for (int i = 0; i < s.length(); i++) {
            data += s.charAt(i);
            data += "#";
        }
        // 半径
        int[] rad = new int[data.length()];
        int id = 0;
        int mx = 0;
        for (int i = 1; i < data.length(); i++) {
            int last = 0;
            if (i > mx) {
                last = i;
            } else {
                if (rad[2 * id - i] < mx - i) {
                    // 2*id-i为中心的最大回文被以id为中心的最大回文所覆盖，没必要继续扩展下去，直接返回
                    rad[i] = rad[2 * id - i];
                    continue;
                } else {
                    last = mx;
                }
            }
            // 继续扩展
            while (last + 1 < data.length() && 2 * i - (last + 1) >= 0 && data.charAt(last + 1) == data.charAt(2 * i - (last + 1))) {
                last++;
            }
            rad[i] = last - i;
            id = i;
            mx = last;
        }
        int left = 0;
        int right = -1;
        for (int i = 0; i < data.length(); i++) {
            // 因为有#的存在，i-rad[i]必是#，也就是偶数下标，i-rad[i]+1对应的必是字母，所以(i-rad[i]+1-1)/2就是原来字母的位置
            int tempLeft = (i - rad[i]) / 2;
            // 同理
            int tempRight = (i + rad[i] - 2) / 2;
            if (tempLeft <= tempRight && right - left < tempRight - tempLeft) {
                left = tempLeft;
                right = tempRight;
            }
        }
        return s.substring(left, right + 1);
    }
}
```

Go beats 67.52%

```go
func preProcess(s string) []rune {
 if s == "" {
  return []rune{'^', '$'}
 }
 T := "^"
 for _, c := range s {
  T += "#" + string(c)

 }
 T += "#$"
 return []rune(T)

}
func longestPalindrome(s string) string {
 T := preProcess(s)
 P := make([]int, len(T))
 var id, mx int
 for i := 1; i < len(T) - 1; i++ {
  j := 2 * id - i
  if mx > i {
          // 暂时先赋值，后面还有一个 while 循环接着判断的
   P[i] = int(math.Min(float64(P[j]), float64(mx-i)))
  } else {
   P[i] = 0
  }
  for T[i+P[i]+1] == T[i-P[i]-1] {
   P[i] += 1
  }
  if i + P[i] > mx {
   id, mx = i, i + P[i]
  }
 }
    // maxIdx保存的是当前最大回文子串中心位置的index
 var maxP, maxIdx int
 for i, p := range P {
  if p > maxP {
   maxP = p
   maxIdx = i
  }
 }
 start := (maxIdx - P[maxIdx] - 1) / 2
 res := s[start:start+P[maxIdx]]
 return res
}
```

第647题也可以用这个算法解，可以记一下这个算法的模版，或者自己去实现一个你喜欢的版本。

小结

对于这道题来说，最简单的方法就是 dp 了，但是如果对自己要求高的话，可以记忆一下 manacher 算法，但是不需要在面试中给面试官讲明白这个算法，如果他原来不知道的话。

在面试的时候，如果你没有信心能在有限时间内给面试官讲解明白，不妨先用最简单的方式去先把题目做出来，然后跟面试官提一句你内心的最优解即可。

对于思路2 和思路3 有不理解的同学（确实是很难），可以在评论区留言问。

}