15 无重复字符的最长子串

更新时间：2019-08-23 10:59:47

> 书是人类进步的阶梯。
>
> ——高尔基

刷题内容

> 难度: Medium
>
> 原题连接：https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/

内容描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:
输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。

请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

题目详解

- 首先子串必须是要连续的，子序列可以不连续。比如字符串为 abcde，那么子序列可以为 ace，但是子串就不行，子串只能是 abc, abcd, abcde, bcd 等等；
- 并且我们要求的是不包含重复字符串的子串；
- 并且我们还要求返回的是满足条件的子串中最长的那个子串的长度。

解题方案

思路 1：时间复杂度: O(N) 空间复杂度: O(N)

求一个最长的子串，里面不带任何重复字符。

假设 input 为 "abcabcbb" ，我们先从第一个字符开始，只有一个字符肯定不会重复吧。“a” 满足条件，更新最大长度为 1；然后走到第二个字符，“ab” 也满足，更新最大长度为 2。
走到第三个字符，“abc” 也满足，更新最大长度为 3。
走到第四个字符，我们发现 “a” 已经出现过了，于是我们就必须要删除之前的一些字符来继续满足无重复字符的条件，但是我们不知道前面已经出现过一次的 “a” 的 index 在哪里呀，所以我们只能一个一个找了，从当前子串的 “abca” 的第一个字符开始找，删除第一个字符 “a”，发现这时候只剩下一个 “a” 了，我们又满足条件了，更新最大长度为 3，以此类推：

```
start
end
 |
 |
 v

 a b c a b c b b
```

end 指针不停往前走，只要当前子串 s[start:end+1] 不满足无重复字符条件的时候，我们就让 start 指针往前走直到满足条件为止，每次满足条件我们都要更新一下最大长度，即 res。

这就是滑动窗口的思想，也称为 sliding window，下面我们来看下代码：

Python beats 67.09%

```python
class Solution:
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        if not s:
            return 0

        start, end = 0, 0
        res, lookup = 0, set()
        while start < len(s) and end < len(s):
            if s[end] not in lookup: # 最新碰到的字符在当前子串中没有出现过
                lookup.add(s[end])  # 记录下当前子串新增的一个字符
                res = max(res, end-start+1) # 因为当前子串满足条件，更新最大长度
                end += 1 # end 指针向前走一步
            else: # 最新碰到的字符在当前子串中已经出现过了
                lookup.discard(s[start]) # 从当前子串中删除之前出现过的字符
                start += 1 # start 指针向前走一步，直到没有重复为止

        return res
```

Java beats 67.68%

```java
import java.util.HashSet;
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s == null || "".equals(s)) {
            return 0;
        }
        int start = 0;
        int end = 0;
        int res = 0;
        // HashSet 用来去重
        HashSet lookup = new HashSet();
        while (start < s.length() && end < s.length()) {
            if (!lookup.contains(s.charAt(end))){
                // end 指针所遇到的字符没有在之前遍历的字符中没有遇到过，就放到 HashSet 中
                lookup.add(s.charAt(end));
                // 满足无重复字串时更新最大长度
                res = res > (end - start + 1) ? res : (end - start + 1);
                // end 指针后移
                end++;
            } else {
                // end 指针所遇到的字符没有在之前遍历的字符中遇到过，就从 HashSet 移除
                lookup.remove(s.charAt(start));
                // start 指针后移
                start++;
            }
        }
        return res;
    }
}
```

Go beats 53.63%

```go
func lengthOfLongestSubstring(s string) int {
    if s == "" {
        return 0
    }

    start, end := 0, 0
    res, lookup := 0, map[byte]bool{}
    for start < len(s) && end < len(s) {
        _, ok := lookup[s[end]]
        if !ok { // 最新碰到的字符在当前子串中没有出现过
            lookup[s[end]] = true // 记录下当前子串新增的一个字符
            res = int(math.Max(float64(res), float64(end-start+1))) // 因为当前子串满足条件，更新最大长度
            end += 1 // end 指针向前走一步
        } else { // 最新碰到的字符在当前子串中已经出现过了
            delete(lookup, s[start]) // 从当前子串中删除之前出现过的字符
            start += 1 // start 指针向前走一步，直到没有重复为止
        }
    }

    return res
}
```

C++ beats 98.17%

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int ret = 0;
        int start = 0, end = 0; // slide window的左右端点（不包括右端点）
        bool flag[128]; // 保存slide window每个字符的存在与否
        memset(flag, false, sizeof(flag));
        while (start < s.size()) {
            while (end < s.size() && !flag[s[end]]) {
                flag[s[end]] = true;
                end++;
            }
            ret = ret < end - start ? end - start : ret;
            flag[s[start]] = false;
            start++;
        }
        return ret;
    }
};
```

思路一的话其实就是典型的滑动窗口的思想，定义两个指针，右指针不停走直到范围不满足，此时通过左指针向右边的移动来使得条件重新被满足，一直重复下去直到长度超出了。

思路 2：时间复杂度: O(N) 空间复杂度: O(N)

那么为了便于解答之后 LeetCode 里面的类似题目，我们这里做一个 slide window 的模版，以后就可以重复使用了，思路 2 其实就是把思路 1 做一个总结，写成一个通用型的，下面来看代码：

Python beats 52.17%

```python
class Solution:
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        lookup = collections.defaultdict(int)
        l, r, counter, res = 0, 0, 0, 0 # counter 为当前子串中 unique 字符的数量
        while r < len(s):
            lookup[s[r]] += 1
            if lookup[s[r]] == 1: # 遇到了当前子串中未出现过的字符
                counter += 1
            r += 1
            # counter < r - l 说明有重复字符出现，否则 counter 应该等于 r - l
            while l < r and counter < r - l:
                lookup[s[l]] -= 1
                if lookup[s[l]] == 0: # 当前子串中的一种字符完全消失了
                    counter -= 1
                l += 1
            res = max(res, r - l) # 当前子串满足条件了，更新最大长度
        return res
```

Java beats 51.04%

```java
import java.util.HashMap;
class Solution {
    public int lengthOfLongestSubstring(String s) {
        HashMap<Character, Integer> lookup = new HashMap();
        int l = 0;
        int r = 0;
        //  counter 用来标记当前字串中 unique 字符的数量
        int counter = 0;
        int res = 0;
        while (r < s.length()) {
            if (lookup.get(s.charAt(r)) == null) {
                // 当前遍历到的字符如果不在 map 中需要进行判空处理
                lookup.put(s.charAt(r),  1);
            }else {
                // 否则可以直接 +1
                lookup.put(s.charAt(r), lookup.get(s.charAt(r)) + 1);
            }
            // 如果遍历到之前没有遇到的字符，则 counter++
            if (lookup.get(s.charAt(r)) == 1) {
                counter++;
            }
            // r 指针右移
            r++;
            // counter < r - l 则说明有重复字串出现，否则 counter 等于 r - l
            while (l < r && counter < r - l) {
                lookup.put(s.charAt(l), lookup.get(s.charAt(l)) - 1);
                // 当前 l 指针所代表的字符在 map 中如果为 0，说明 l 指针所代表的字符在 map 中完全被清除
                if (lookup.get(s.charAt(l)) == 0) {
                    counter--;
                }
                // l 指针右移
                l++;
            }
            // 更新最大字串长度
            res = res > (r - l) ? res : (r - l);
        }
        return res;
    }
}
```

Go beats 70.24%

```golang
func lengthOfLongestSubstring(s string) int {
    l, r, counter, res := 0, 0, 0, 0 // counter 为当前子串中 unique 字符的数量
    lookup := map[byte]int{}
    for r < len(s) {
        _, ok := lookup[s[r]]
        if !ok {
            lookup[s[r]] = 1
        } else {
            lookup[s[r]] += 1
        }
        if lookup[s[r]] == 1 { // 遇到了当前子串中未出现过的字符
            counter += 1
        }
        r += 1
        // counter < r - l 说明有重复字符出现，否则 counter 应该等于 r - l
        for l < r && counter < r - l {
            lookup[s[l]] -= 1
            if lookup[s[l]] == 0{ // 当前子串中的一种字符完全消失了
                counter -= 1
            }
            l += 1
        }
        res = int(math.Max(float64(res), float64(r-l))) // 当前子串满足条件了，更新最大长度
    }
    return res
}
```

c++ beats: 96.98%

```c++
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int ret = 0;
        int lookup[128];
        memset(lookup, 0, sizeof(lookup));

        int l = 0, counter = 0;
        for (int r = 0;r < s.size();r++) {
            lookup[s[r]]++;
            if (lookup[s[r]] == 1) {
                counter++;
            }
            while (l <= r && counter < r - l + 1) {
                lookup[s[l]]--;
                if (lookup[s[l]] == 0) {
                    counter--;
                }
                l++;
            }
            ret = ret > counter ? ret : counter;
        }
        return ret;
    }
};
```

有兴趣的同学可以用这个模版去做一下第 159 题和第 340 题，会发现改几个数字就完全OK了。

思路 3：时间复杂度：O(N) 空间复杂度：O(N)

刚才思路 1 中有这样一句话：但是我们不知道前面已经出现过一次的“a”的index在哪里呀，所以我们只能一个一个找了。

我们可以对这里做一个优化，就不需要一个个去找了。只需要用一个字典，对当前子串中的每一个字符，将其在 input 中的来源 index 记录下来即可。

我们先从第一个字符开始，只要碰到已经出现过的字符，我们就必须从之前出现该字符的 index 开始重新往后看。

例如 'xyzxlkjh'，当看到第二个 'x' 时，我们就应该从第一个 x 后面的 y 开始重新往后看了。

我们将每一个已经阅读过的字符作为 key，而它的值就是它在原字符串中的 index。如果我们现在的字符不在字典里面，我们就把它加进字典中去。因此，只要 end 指针指向的这个字符 c，在该字典中的值大于等于了当前子串首字符的 index 时，就说明 c 在当前子串中已经出现过了，我们就将当前子串的首字符的 index 加 1，即从后一位又重新开始读，此时当前子串已经满足条件了，然后我们更新 res。

程序变量解释

- start 是当前无重复字符的子串首字符的 index；
- maps 放置每一个字符的 index，如果 maps.get(s[i], -1) 大于等于 start 的话，就说明字符重复了，此时就要重置 res 和 start 的值了。

Python beats 74.35%

```
class Solution:
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        # start 指针指向的是当前子串首字符在 input 中对应的index
        res, start, n = 0, 0, len(s)
        maps = {}
        for i in range(n):
            start = max(start, maps.get(s[i], -1) + 1) # 找到当前子串新的起点
            res = max(res, i - start + 1) # 当前子串满足条件了，更新结果
            maps[s[i]] = i # 将当前字符与其在 input 中的 index 记录下来
        return res
```

Java beats 90.87%

```java
import java.util.HashMap;
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int res = 0;
        // start 标记当前字串首字符在 s 中对应的索引位置
        int start = 0;
        int length = s.length();
        HashMap<Character, Integer> map = new HashMap();
        for (int i = 0; i < length; i++) {
            int temp = -1;
            if (map.get(s.charAt(i)) != null) {
                // 可能当前遍历到的字符是一个全新的字符，没有存在过 map 中，所以需要判空处理
                temp = map.get(s.charAt(i));
            }
            // 找到字串新的起点
            start = start > (temp + 1) ? start : (temp + 1);
            // 更新字串的长度
            res = res > (i - start + 1) ? res : (i - start + 1);
            // 将当前遍历到的字符记录在 map 中
            map.put(s.charAt(i), i);
        }
        return res;
    }
}
```

Go beats 99.89%

```go
func lengthOfLongestSubstring(s string) int {
    res, start := 0, 0 // start 指针指向的是当前子串首字符在 input 中对应的index
    lookup := map[byte]int{}

    for i, _ := range s {
        idx, ok := lookup[s[i]]
        if !ok {
            start = int(math.Max(float64(start), float64(-1 + 1))) // 找到当前子串新的起点
        } else {
            start = int(math.Max(float64(start), float64(idx + 1))) // 找到当前子串新的起点
        }
        res = int(math.Max(float64(res), float64(i-start+1))) // 当前子串满足条件了，更新结果
        lookup[s[i]] = i // 将当前字符与其在 input 中的 index 记录下来
    }
    return res
}
```

c++ beats 96.98%

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int ret = 0;
        int lookup[128];
        memset(lookup, -1, sizeof(lookup));
        int start = 0;
        for (int i = 0;i < s.size();i++) {
            if (lookup[s[i]] != -1) {
                start = start < lookup[s[i]] + 1 ? lookup[s[i]] + 1 : start;
            }
            lookup[s[i]] = i;
            ret = i - start + 1 > ret ? i - start + 1 : ret;
        }
        return ret;
    }
};
```

思路3的话又可以省一些时间了，因为相当于我们的左指针不需要一步一步走了，而是知道下一步在哪里。

总结

1. 分清子串和子序列的概念；
2. 学会从当前做不到的一些东西去优化，想想怎么利用基础数据结构去做优化；
3. 学会从一类题中总结出模版。

}

14 两数相加

16 最长回文子串