3. 无重复字符的最长子串.ts

> 题目策略：滑动窗口
>
> leetcode 地址：https://leetcode.cn/problems/longest-substring-without-repeating-characters/

问题描述

1. 问题：给定一个字符串 s，请你找出其中不含有重复字符的 最长子串 的长度。s 由英文字母、数字、符号和空格组成。
2. 输入输出样例

```
Input s = "abcabcbb" Output 3 ("abc")
Input s = "bbbbb" Output 1 ("b")
Input s = "pwwkew" Output 3 ("wke")
```

问题解决 - 1

1. 思路

```
- 使用字典 charFreqDict 记录字符及其索引位置，key 是字符，value 是其对应的索引。
- 每次右边界 j 移动时：
  1. 检查新字符 s[j] 是否已在字典中（即是否重复）
     char = s[j];
     isRepeated = Boolean(charFreqDict[char])
  2. 如果不重复，更新字典 charFreqDict[char] = j
  3. 如果重复，更新左边界 i 为重复字符的下一个位置，并更新字典
     i = i_new = charFreqDict[char]
     charFreqDict[i_old → i_new - 1] = undefined
     charFreqDict[char] = j,
  4. 比较当前子串长度，若更长则更新全局最长子串
```

2. 时间复杂度

时间复杂度分析：O(n)
- 假设字符串 s 的长度为 n，则抓主要矛盾，基于两个循环来分析该函数的时间复杂度
- while (true) ⟹ 每次循环窗口的右边界移动一个单位，因此该循环最多执行 n 次
- for (let c of arrCleared) ⟹ 如果移动右边界发现引入了重复字符，

则需要使用该循环调整左边界，事实上，左边界最多也只能移动 n 次，

即所有 while 循环下来，该循环的执行次数最多为 n 次
- 综上所述，整个函数的时间复杂度为 O(n)

3. 代码实现

```
function lengthOfLongestSubstring1(s: string): number {
  if (s.length ⩽ 0) return 0;

  const charFreqDict: { [key: string]: any } = { [s[0]]: 0 }; // 字符索引字典
  const currentRes = { str: s[0], length: 1 }; // 当前窗口不重复子串
  const globalRes = { str: s[0], length: 1 }; // 全局不重复子串
  let i = 0; // 窗口左边界
  let j = 0; // 窗口右边界

  while (true) {
    /* 扩展窗口 */
    const char = s[++j]; // 移动右边界

    /* 循环退出条件 */
    if (i ⩾ s.length || j ⩾ s.length) break;

    /* 是否引入重复字符检查 */
    const isRepeated = charFreqDict[char] =/= undefined; // 检查引入的新字符是否
会使得窗口字符串变重复

    /* 确保窗口子串不包含重复字符 */
    if (isRepeated) {
      // Step1. 计算新旧窗口的左边界
      const [old_i, new_i] = [i, charFreqDict[char] + 1];
      // Step2. 清空 s[old_i, new_i] 的所有字符的频率为 undefined
      const arrCleared = s.slice(old_i, new_i - 1);
      for (let c of arrCleared) charFreqDict[c] = undefined;
      // Step3. 移动左边界
      i = new_i;
    }
    charFreqDict[char] = j;

    /* 更新当前窗口子串 */
    [currentRes.str, currentRes.length] = [s.slice(i, j + 1), j - i + 1];
```

```
    /* ?更新全局不重复子串 */
    if (globalRes.length < currentRes.length)
      [globalRes.str, globalRes.length] = [currentRes.str,
currentRes.length];
  }

  return globalRes.length;
}
```

问题解决 - 2

1. 思路

在思路 1 的基础上进行优化
- 使用 charIndexMap 来记录最新索引的位置
- 字符重复判断基于 charIndexMap[char] =/= undefined
  && charIndexMap[char] ⩾ start, 避免了清理字典中的旧值

2. 时间复杂度

与问题解决 - 1 相同, O(n)

3. 代码实现

```
function lengthOfLongestSubstring2(s: string): number {
  const charIndexMap: { [key: string]: number } = {}; // 字符索引字典
  let maxLength = 0; // 不重复子串的最大长度
  let i = 0; // 窗口左边界

  /* 移动右边界 */
  for (let j = 0; j < s.length; j++) {
    /* 是否引入重复字符检查 */
    const char = s[j];
    const isRepeated =
      charIndexMap[char] =/= undefined && charIndexMap[char] ⩾ i;

    /* 确保窗口子串不包含重复字符 */
    if (isRepeated) i = charIndexMap[char] + 1;

    /* 更新引入的字符的索引 */
    charIndexMap[char] = j;
```

```
    /* 更新最大长度 */
    maxLength = Math.max(maxLength, j - i + 1);
  }

  return maxLength;
}
```

性能测试

```
/* 性能比较：虽然时间复杂度相同，但是思路 2 显然由于思路 1，可能是因为
  - 避免了不必要的操作：清理字典中的旧值。
  - 减少了变量赋值和切片操作的开销
  - 没有过多内存分配和垃圾回收
  - 简化循环 */
const performanceTest = () => {
  const inputs = ["abcabcbb", "bbbbb", "pwwkew"];
  console.time("无重复字符的最长子串 - 思路 1");
  inputs.forEach((input) => {
    console.log(lengthOfLongestSubstring1(input));
  });
  console.timeEnd("无重复字符的最长子串 - 思路 1"); // 1.681ms
  console.time("无重复字符的最长子串 - 思路 2");
  inputs.forEach((input) => {
    console.log(lengthOfLongestSubstring2(input));
  });
  console.timeEnd("无重复字符的最长子串 - 思路 2"); // 0.185ms
};
```