剑指48 最长不含重复字符的子字符串

思路 类似 offer42 连续子数组的最大和， 但是需要使用hash来存储以便判断每一步的字符重复项。

动态规划 或 滑动窗口

法一 动态规划

由于hash使用不熟练，导致很多用法思路不对，复杂度依然高。
而子串长度计算应该优先考虑下标相减，而不是存储子串然后算length。

关键点在于，字典中元素无需删除，出现重复元素则直接更新为最新的下标值，而重复的原下标值则需要判断是在原子字符串内还是外边，在内部要从这个点开始重新计算子串长度值；在外部则没有影响，子串长度值加一。

动态规划解析：

- **状态定义：** 设动态规划列表 dp ， $dp[j]$ 代表以字符 $s[j]$ 为结尾的 “最长不重复子字符串” 的长度。
- **转移方程：** 固定右边界 j ，设字符 $s[j]$ 左边距离最近的相同字符为 $s[i]$ ，即 $s[i] = s[j]$ 。
  1. 当 $i < 0$ ，即 $s[j]$ 左边无相同字符，则 $dp[j] = dp[j-1] + 1$ ；
  2. 当 $dp[j-1] < j - i$ ，说明字符 $s[i]$ 在子字符串 $dp[j-1]$ **区间之外** ，则 $dp[j] = dp[j-1] + 1$ ；
  3. 当 $dp[j-1] \geq j - i$ ，说明字符 $s[i]$ 在子字符串 $dp[j-1]$ **区间之中** ，则 $dp[j]$ 的左边界由 $s[i]$ 决定，即 $dp[j] = j - i$ ；

  > 当 $i < 0$ 时，由于 $dp[j-1] \leq j$ 恒成立，因而 $dp[j-1] < j - i$ 恒成立，因此分支 1. 和 2. 可被合并。

$$dp[j] = \begin{cases} dp[j-1] + 1 & , dp[j-1] < j - i \\ j - i & , dp[j-1] \geq j - i \end{cases}$$

- **返回值：** $\max(dp)$ ，即全局的 “最长不重复子字符串” 的长度。

```cpp
// 我的答案1,时间空间复杂度不符合要求。（动规+遍历）

class Solution {
public:
    int lengthOfLongestSubstring(string s) {
```

```
6
7       //dp[n] = dp[n - 1] + 1  (如果s[n]和dp[n-1]方案中的子串中字符不重复)
8       //      =  从s[n]往前找到第一个和s[n]重复的字符，计算长度          (否则)
9       int maxlen = 0;
10      string subs = "";
11      for (int i = 0; i < s.size(); i++) {
12          int pos = subs.rfind(s[i]);
13          subs += s[i];
14          if (pos != string::npos) {
15              subs = subs.substr(pos + 1);
16          }
17          maxlen = subs.size() > maxlen ? subs.size() : maxlen;
18      }
19      return maxlen;
20  }
21 };
22
```

```
1  // k神翻译为c++，好难！！  (动规+hash)
2  class Solution {
3  public:
4      int lengthOfLongestSubstring(string s) {
5          unordered_map<char, int> dic;
6          int res = 0, tmp = 0;
7          for (int j = 0; j < s.length(); j++) {
8              int i = dic.find(s[j]) != dic.end() ? dic[s[j]] : -1; // 获取索引 i
9              dic[s[j]] = j; // 更新哈希表
10             tmp = tmp < j - i ? tmp + 1 : j - i; // dp[j - 1] -> dp[j]
11             res = max(res, tmp); // max(dp[j - 1], dp[j])
12         }
13         return res;
14     }
15 };
16
17
18 // 我的易于理解的版本：
19 class Solution {
20 public:
21     int lengthOfLongestSubstring(string s) {
22         unordered_map<char, int> dic;
23         int res = 0, tmp = 0; // res最大长度，tmp为以当前s[j]结尾的符合要求的最长子串长度
24         for (int j = 0; j < s.length(); j++) {
25             if (dic.find(s[j]) == dic.end()) { // 前面没有和s[j]重复的
26                 // dp[j] = dp[j - 1] + 1
27                 dic[s[j]] = j;
28                 tmp += 1;
29             }
30             else if (j - dic[s[j]] > tmp) { // 前面有和s[j]重复的但是不被上次的最长子串包含
31                 dic[s[j]] = j;
32                 tmp += 1;
33             }
34             else { // 前面有重复的且被上次的子串包含，重新计算子串长度。
35                 tmp = j - dic[s[j]];
36                 dic[s[j]] = j;
37             }
38             res = max(res, tmp); // max(dp[j - 1], dp[j])
39         }
```

```
        return res;
    }
};
```

法二 双指针滑动窗口

以第一个字符为起点(左指针)，遍历直到发现有重复的(移动右指针)，子串长度记为k。
然后以第二个字符为起点(左指针+1)，此时可以确定往后的k-1个字符都不重复，直接从右指针+1 开始寻找重复的。
这样不断一移动左右指针。

滑动窗口技巧详解！！

https://www.bilibili.com/video/BV1hd4y1r7Gq/?vd_source=7d2082262d19abdd2e30ff5f60dc1fdf

```
// 我的答案是否别扭
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> dic;
        int l = -1, r = -1, res = 0;
        while (l + 1 < s.size()) {
            l++;
            while (r + 1 < s.size()) {
                r++;
                if (dic.find(s[r]) == dic.end() || dic[s[r]] < l) {
                    dic[s[r]] = r;
                }
                else {
                    l = dic[s[r]];
                    dic[s[r]] = r;
                    break;
                }
                res = res > r - l + 1 ? res : r - l + 1;
            }
        }
        return res;
    }
};
```