功能函数

```
//获得字符子串
 string s;
s.substr(start_pos, len);
s.find_first_not_of(' ');
```

003无重复字符的最长子串（middle）

- **题目**
  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。

- **思路**
  left保存无重复子串中的左端，max_len保存最大的无重复子串。
  使用不对关键字进行排序的unordered_map，存储当前字符，出现的位置。
  1、通过count(关键字)函数判断当前关键字有无出现过，若出现过且出现在left和i之间的子串，则把left更新为res[s[i]]出现过的位置。
  2、res[s[i]] = i;
  3、 max_len = max_len > i-left ? max_len:i-left; :其中i-left相当于局部无重复子串的长度

- **基础知识**
  创建的哈希表无序的即可，所以用ordered_map，否则用map。

- **代码实现**

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int len = s.length();
        if(len < 2)
            return len;
        unordered_map<char, int> res;
        int max_len = 0;
        for(int i = 0, left = -1; i < len; ++i){
            if(res.count(s[i]) && res[s[i]] > left) //出现与之前重复的字符，且出现在left的右边（即出现在当前子串中）
                left = res[s[i]]; //更新left的值
            res[s[i]] = i;
            max_len = max_len > i-left ? max_len:i-left; //保留之前最长子串长度与当前子串长度之间的最大值
        }
        return max_len;
    }
};
//解法2
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int len = s.length();
        if(len < 2)
            return len;
        vector<int> res(128,-1); //键盘输入的字符范围为0到128
        int max_len = 0;
        for(int i = 0, left = -1; i < len; ++i){
            left = left > res[s[i]] ? left:res[s[i]]; //left保存当前无重复字符串最左边的位置
            res[s[i]] = i;
            max_len = max_len > i-left ? max_len:i-left;
        }
        return max_len;
    }
};
```

005 最长回文子串（middle）

- **题目**
  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。

- **思路**
  - 思路1：遍历字符串每个字符，考虑到存在奇数和偶数回文子串，以当前字符和当前字符+下个字符为中心点向左右两边搜索（搜索调用自己写的函数），复杂度平均为O（n*n）
  - 思路2：不调用子函数，用start和max_len保存最长回文子串的开始位置和长度。
    遍历每个字符，left = i，right = i，若right与right+1位置的值相等，则在边界内不断右移，即++right，
    i = right + 1;
    进行往左往右的回文判断，
    若 right - left + 1 > max_len ，则更新start和max_len；
  - 思路3：马拉车算法，时间复杂度为O(n)，需要额外的内存空间，有需要以后再看
    https://www.cnblogs.com/grandyang/p/4475985.html

- **代码实现**

```
class Solution {
public:
    string longestPalindrome(string s) {
        int len = s.length();
        if(len < 2)
            return s;
        int start = 0, max_len = 0;
        for(int i = 0; i < len - 1;){
            if(len - i < max_len >> 1) //做了优化处理
                break;
```

```
            int left = i, right = i;
            while(right < len-1 && s[right] == s[right+1]) //right右边出现重复的值
                ++right;
            i = right+1; //可直接跳过全部重复的值（此次遍历跟这些重复的值肯定包含进去了，不需要下次重复计算）
            while(left > 0 && right < len - 1 && s[left - 1] == s[right + 1])
                --left, ++right;
            if(right - left + 1> max_len){  //更新start和max_len
                max_len = right-left + 1;
                start = left;
            }
        }
        return s.substr(start, max_len);
    }
};
```

006 Z 字形变换（middle）

- **题目**
  将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。
  比如输入字符串为 "LEETCODEISHIRING" 行数为 3 时，排列如下：

  > L C I R
  > E T O E S I I G
  > E D H N
  > 其实除了第一排和最后一排，其他每行的长度都是一样的，并且是之字形摆的

- **思路**

当 n = 2 时：

0 2 4 6 8 A C E

1 3 5 7 9 B D F

当 n = 3 时：

0 4 8 C

1 3 5 7 9 B D F

2 6 A E

当 n = 4 时：

0 6 C

1 5 7 B D

2 4 8 A E

3 9 F

1、除了第一行和最后一行没有中间形成之字型的数字外，其他都有。

2、首位两行中相邻两个元素的index之差跟行数是相关的，为 *2nRows - 2*, 根据这个特点，我们可以按顺序找到**所有的黑色元素**在元字符串的位置，将他们按顺序加到新字符串里面

3、对于**红色元素**出现的位置也是有规律的，每个红色元素的位置为 *j* + 2nRows-2 - 2*i, 其中，j为前一个黑色元素的列数，i为当前行数。

- **代码实现**

```
class Solution {
public:
    string convert(string s, int numRows) {
        if(s.empty() || numRows == 1)
            return s;
        string res("");
        int step = 2 * numRows - 2; //step为相邻黑色字符的步长
        for(int i = 0; i < numRows; ++i) //按行处理
            for(int j = i; j < s.length(); j += step){
                res += s[j];
                int temp_pos = j + step - 2*i; //黑色字符旁边的红色字符位置
                if(i != 0 && i != numRows-1 && temp_pos < s.length())
                    res += s[temp_pos];
```

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

010 正则表达式匹配（hard）

- **题目**
  给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

  > '.' 匹配任意单个字符
  > '*' 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

- **思路**
  1、若p为空，若s也为空，返回true，反之返回false。
  2、若p的长度为1，若s长度也为1，且相同或是p为'.'则返回true，反之返回false。
  3、若p的第二个字符不为，若此时*s*为空返回*false*，否则判断首字符是否匹配，且从各自的第二个字符开始调用递归函数匹配。
  4、若*p*的第二个字符为，进行下列循环，条件是若s不为空且首字符匹配（包括p[0]为点），调用递归函数匹配s和去掉前两个字符的p（这样做的原因是假设此时的星号的作用是让前面的字符出现0次，验证是否匹配），若匹配返回true，否则s去掉首字母（因为此时首字母匹配了，我们可以去掉s的首字母，而p由于星号的作用，可以有任意个首字母，所以不需要去掉），继续进行循环。
  5、返回调用递归函数匹配s和去掉前两个字符的p的结果（这么做的原因是处理星号无法匹配的内容，比如s="ab", p="a*b*"，直接进入*while*循环后，我们发现*"ab"*和*"b"*不匹配，所以*s*变成*"b"*，那么此时跳出循环后，就到最后的*return*来比较*"b"*和*"b"*了，返回*true*。再举个例子，比如*s=""*, *p="a*"，由于s为空，不会进入任何的if和while，只能到最后的return来比较了，返回true，正确）。

- **代码实现**

```
class Solution {
public:
    bool isMatch(string s, string p) {
        if(p.empty())
```

```
            return s.empty();
        if(p.size() == 1)
            return s.size()==1 && (s[0] == p[0] || p[0] == '.');
        if(p[1] != '*'){
            if(s.empty())
                return false;
            if(s[0] == p[0] || p[0] == '.')
                return isMatch(s.substr(1), p.substr(1));
            else
                return false;
        }
        while(!s.empty() && (p[0] == s[0] || p[0] == '.')){
            if(isMatch(s, p.substr(2))) //假设出现*前面的值出现0次
                return true;
            s = s.substr(1);//s后移一位，*前面的数有效
        }
        return isMatch(s, p.substr(2));
    }
};
```

012 整数转罗马数字（middle）

- **题目**
  罗马数字包含以下七种字符： I， V， X， L，C，D 和 M。

> 字符 数值
> I 1
> V 5
> X 10
> L 50
> C 100
> D 500
> M 1000
> 例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。
> 通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情

> 况：
> I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
> X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
> C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个整数，将其转为罗马数字。输入确保在 1 到 3999 的范围内。

- **思路**
  定义两个数组，分别把1000,对应的字符M，900对应的字符CM，按这个规律全存到数组里，然后num去遍历计算。
- **代码实现**

```
class Solution {
public:
    string intToRoman(int num) {
        string res = "";
        vector<int> val{1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        vector<string> str{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
        for (int i = 0; i < val.size(); ++i) {  //不断减去最大位上的值
            while (num >= val[i]) {
                num -= val[i];
                res += str[i];
            }
        }
        return res;
    }
};
```