> 正则表达式匹配」就要求我们实现一个简单的正则匹配算法，包括「.」通配符和「*」通配符。
> 点号「.」可以匹配任意一个字符，星号「*」可以让之前的那个字符重复任意次数（包括 0 次）。

分析：

1. 如果只有.只需匹配一次即可；
2. 当将入*后，如果.之后为*则直接返回
3. 如果*之前不是.,则需要进行匹配0到多次。

状态为：字符s和p的指针i,j 选择为：字符p[j]匹配多次。 建立状态函数dp(s,i,p,j)：

- 返回true表示s[i..]可以匹配p[j..]否则为不匹配。

```
class Solution {
public:
  bool isMatch(std::string s, std::string p) {
    return dp(s, 0, p, 0);
  }

private:
  bool dp(std::string s, int i, std::string t, int j) {
    int m = s.size();
    int n = t.size();

    if (j >= n) {  // p走完，计算s是否已经遍历完成
      return i == m;
    }

    if (i >= m) {
      // s遍历完成，需要判断p余下部分是否能够匹配空串，如果能够匹配空串余下长度一定为2
的倍数
      if (1 == (n - j) % 2) {
        return false;
      }

      // 判断余下部分是否能够匹配空串
      for (; j < n - 1; j += 2) {
        if (t[j + 1] != '*') {
          return false;
        }
      }

      return true;
    }

    // 计算memo数组
    // 记录<i,j>消除重复子问题
    std::string key = std::to_string(i) + ',' + std::to_string(j);
    if (memo.count(key)) {
      return memo[key];
```

```
      }

      bool res = false;

      if (s[i] == t[j] || t[j] == '.') {
        if (j < n - 1 && t[j + 1] == '*') {
          // 通配符匹配0~多次
          res = dp(s, i, t, j + 2) || dp(s, i + 1, t, j);
        } else {
          res = dp(s, i + 1, t, j + 1);
        }
      } else {
        if (j < n - 1 && t[j + 1] == '*') {
          // 通配符匹配0次
          res = dp(s, i, t, j + 2);
        } else {
          return false;
        }
      }

      // 结果计入备忘录
      memo[key] = res;

      return res;
    }

    std::map<std::string, bool> memo;
};
```