递归

递归（recursion）是一种算法策略，通过函数调用自身来解决问题。它主要包含两个阶段。

- **递**：程序不断深入地调用自身，通常传入更小或更简化的参数，直到达到“终止条件”。
- **归**：触发“终止条件”后，程序从最深层的递归函数开始逐层返回，汇聚每一层的结果。

而从实现的角度看，递归代码主要包含三个要素。

- **终止条件**：用于决定什么时候由“递”转“归”。
- **递归调用**：对应“递”，函数调用自身，通常输入更小或更简化的参数。
- **返回结果**：对应“归”，将当前递归层级的结果返回至上一层。

观察以下代码，我们只需调用函数 `recur(n)` ，就可以完成 `1+2+3 +...+ n` 的计算

```
/* 递归 */
int recur(int n) {
    // 终止条件
    if (n == 1)
        return 1;
    // 递：递归调用
    int res = recur(n - 1);
    // 归：返回结果
    return n + res;
}
```

从计算角度看，迭代与递归可以得到相同的结果，**但它们代表了两种完全不同的思考和解决问题的范式。**

- **迭代**：“自下而上”地解决问题。从最基础的步骤开始，然后不断重复或累加这些步骤，直到任务完成。
- **递归**：“自上而下”地解决问题。将原问题分解为更小的子问题，这些子问题和原问题具有相同的形式。

  接下来将子问题继续分解为更小的子问题，直到基本情况时停止（基本情况的解是已知的）。

以上述求和函数为例，设问题 $f(n) = 1 + 2 + \cdots + n$

- **迭代**：在循环中模拟求和过程，从 1 遍历到 n ，每轮执行求和操作，即可求得 $f(n)$ 。
- **递归**：将问题分解为子问题 $f(n) = n + f(n-1)$ ，不断（递归地）分解下去，直至基本情况 $f(1) = 1$ 时终止。

动态规划

动态规划(dynamic programming)是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法。

动态规划算法通常基于一个**递推公式**及一个或多个**初始状态**。 当前**子问题的解将由上一次子问题的解推出。**

基本思想：

- 要解决一个给定的问题，我们需要解决其不同部分（即**解决子问题**），再合并子问题的解以得出原问题的解。
- 通常许多子问题非常相似，为此动态规划法试图**只解决每个子问题一次**，从而减少计算量。

动态规划的意义是什么?

https://www.zhihu.com/question/23995189

动态经典题目之爬楼梯

题目描述：有 10 阶楼梯，每次可以上一阶或者两阶，求有多少种上楼梯的方法？请用 golang 来实现一波

首先根据直觉来想一下：

- 每次走1级台阶，一共走10步，这是其中一种走法。
- 每次走2级台阶，一共走5步，这是另一种走法。

这样只能一个一个穷举，实在是太麻烦了，如果是100节楼梯呢？是 n 节楼梯呢？

算法思想

我们可以分解成局部问题逐步求解：

- 前面8节楼梯的走法有 f(8) 种 ， 最后走二节楼梯 。这一共是种 f(8) 种走法
- 前面9节楼梯的走法有f(9)种，最后走一节楼梯，这一共是种 f(9) 种走法
- 所有走完10节楼梯的走法一共是 f(10) = f(8) + f(9) 种走法
- 依次递推 f9 = f(8) + f(7)   f8= f(7) + f(6) f(7)=f(6)+f(5) f(6)=f(5)+f(4)
- 得出递归方程， **f(n) = f(n-1) + f(n-2);** 其实就是斐波那契数列
- 初始值 **f(1) = 1; f2) = 2;**

一节楼梯时，当然只有一种走法：

- 走一节

两节楼梯时，两种走法：

- 走一节，走一节
- 走两节

三节楼梯时，三种走法：

- 走一节，走一节，走一节
- 走一节，走两节
- 走两节，走一节

```
package main

import (
    "fmt"
    "os"
)

/*
算法一：斐波那契数列 - 递归求解
Refer https://segmentfault.com/a/1190000015944750
```

```
Refer https://zhuanlan.zhihu.com/p/72734380
每次计算n的时候，其实n-1 和 n-2都被计算出来了。递归过程中，很多值都要被重新计算一次。
这种方法的时间复杂度为O(2^n)
*/

func climbStairs1(n int) int {
    if n<3{
        return n
    }
    return climbStairs1(n-1) + climbStairs1(n-2)
}


/*
算法二：自底向上
这种算法，其实就是通过循环，以及多重赋值的方式，在一个函数调用里面，循环进行重新赋值。内存开销较小
*/

func climbStairs2(n int) int {
   if n < 3 {
        return n
    }
    a, b := 1, 2
    // 在循环中，多重赋值，向上叠加
    for i := 3; i < n; i++ {
       a, b = b, a + b
    }
    return a + b
}




func main() {
    var n int  = 20
    // 10946
    fmt.Println(climbStairs1(n))
    // 10946
    fmt.Println(climbStairs2(n))

}
```