递归（Recursion）

- 递归：函数（方法）直接或间接调用自身。是一种常用的编程技巧

```
int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

```
void a(int v) {
    if (v < 0) return;
    b(--v);
}

void b(int v) {
    a(--v);
}
```

递归现象

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？【从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？『从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……』】

GNU 是 GNU is Not Unix 的缩写

GNU → GNU is Not Unix → GNU is Not Unix is Not Unix → GNU is Not Unix is Not Unix is Not Unix

假设A在一个电影院，想知道自己坐在哪一排，但是前面人很多，
A 懒得数，于是问前一排的人 B【你坐在哪一排？】，只要把 B 的答案加一，就是 A 的排数。
B 懒得数，于是问前一排的人 C【你坐在哪一排？】，只要把 C 的答案加一，就是 B 的排数。
C 懒得数，于是问前一排的人 D【你坐在哪一排？】，只要把 D 的答案加一，就是 C 的排数。
……
直到问到最前面的一排，最后大家都知道自己在哪一排了

小码哥教育 SEEMYGO

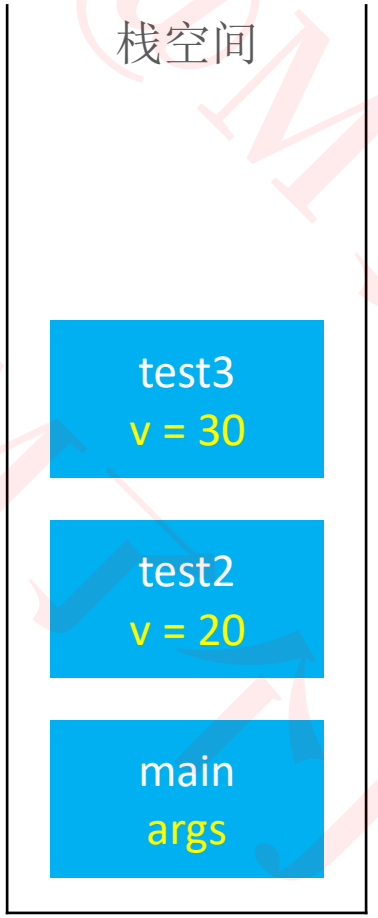
函数的调用过程

```
public static void main(String[] args) {
    test1(10);
    test2(20);
}

private static void test1(int v) {}

private static void test2(int v) {
    test3(30);
}

private static void test3(int v) {}
```

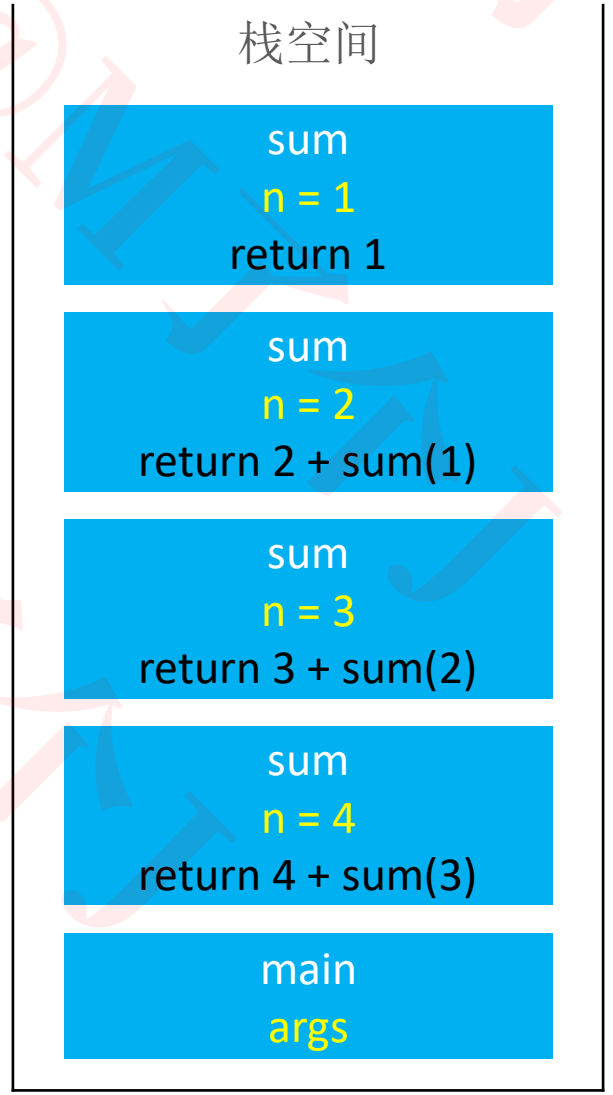


函数的递归调用过程

```java
public static void main(String[] args) {
    sum(4);
}

private static int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

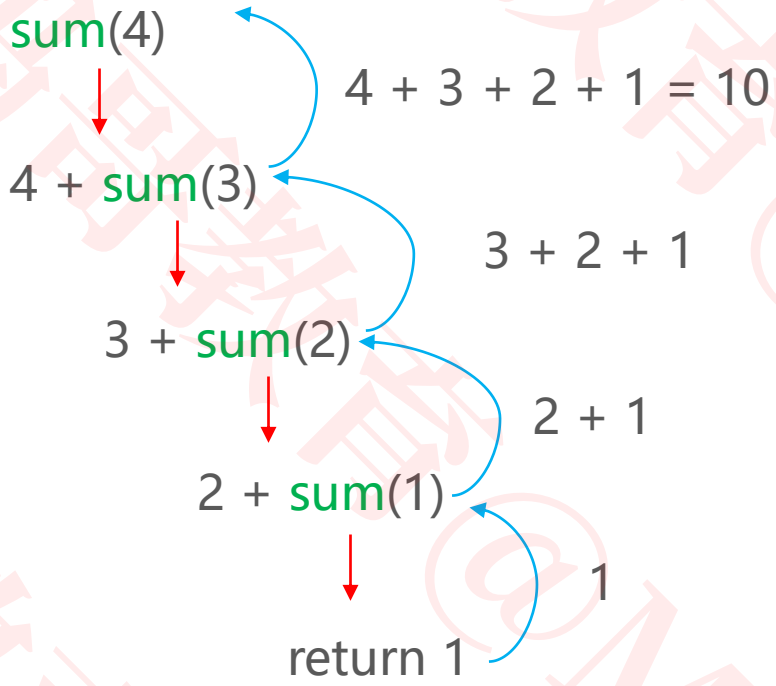
空间复杂度：$O(n)$



- 如果递归调用没有终止，将会一直消耗栈空间
  - 最终导致栈内存溢出（Stack Overflow）

- 所以必需要有一个明确的结束递归的条件
  - 也叫作边界条件、递归基

函数的递归调用过程



实例分析

- 求 1+2+3+...+(n-1)+n 的和（n>0）

```
int sum(int n) {
    if (n <= 1) return n;
    return n + sum(n - 1);
}
```

- 总消耗时间 $T(n) = T(n-1) + O(1)$，因此
  - 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

```
int sum(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

- 时间复杂度：$O(n)$，空间复杂度：$O(1)$

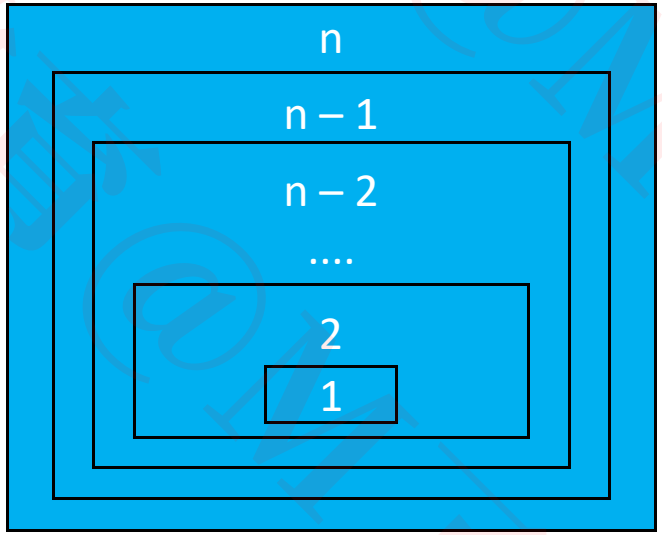
```
int sum(int n) {
    if (n <= 1) return n;
    return (1 + n) * n >> 1;
}
```

- 时间复杂度：$O(1)$，空间复杂度：$O(1)$

- 注意：使用递归不是为了求得最优解，是为了简化解决问题的思路，代码会更加简洁
- 递归求出来的很有可能不是最优解，也有可能是最优解

递归的基本思想

- 拆解问题
  - 把规模大的问题变成规模较小的同类型问题
  - 规模较小的问题又不断变成规模更小的问题
  - 规模小到一定程度可以直接得出它的解



- 求解
  - 由最小规模问题的解得出较大规模问题的解
  - 由较大规模问题的解不断得出规模更大问题的解
  - 最后得出原来问题的解

- 凡是可以利用上述思想解决问题的，都可以尝试使用递归
  - 很多链表、二叉树相关的问题都可以使用递归来解决
  - ✓ 因为链表、二叉树本身就是递归的结构（链表中包含链表，二叉树中包含二叉树）

递归的使用套路

① 明确函数的功能

- 先不要去思考里面代码怎么写，首先搞清楚这个函数的干嘛用的，能完成什么功能？

② 明确原问题与子问题的关系

- 寻找 f(n) 与 f(n – 1) 的关系

③ 明确递归基（边界条件）

- 递归的过程中，子问题的规模在不断减小，当小到一定程度时可以直接得出它的解
- 寻找递归基，相当于是思考：问题规模小到什么程度可以直接得出解？