

递归：如何用三行代码找到最终推荐人

推荐注册返佣金的这个功能我想你应该不陌生吧？现在很多 App 都有这个功能。这个功能中，用户 A 推荐用户 B 来注册，用户 B 又推荐了用户 C 来注册。我们可以说，用户 C 的“最终推荐人”为用户 A，用户 B 的“最终推荐人”也为用户 A，而用户 A 没有“最终推荐人”

问题：给定一个用户ID，如何查找这个用户的最终推荐人？

如何理解递归

递归是一种应用非常广泛的算法。之后很多数据结构和算法的编码实现都要永达递归，比如DFS深度优先搜索、前中后序二叉树遍历等等。

例：电影院问坐在第几排的例子，只需要问前面人是第几排，在此基础上加一。

去的过程是递，回来的过程是归。

上述问题递推公式为：

$$f(n) = f(n-1) + 1; \text{其中, } f(1) = 1;$$

有了递推公式和终止条件，则编程实现就很简单了：

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

递归满足三个条件

只要同时满足以上三个条件，就可以用递归来解决

- 1. 一个问题的解可以分成几个子问题的解** 子问题就是数据规模更小的问题
- 2. 这个问题的子问题，处理数据规模不同，求解思路完全相同** 子问题只是数据大小规模不一样，但解题思路是完全一样的。
- 3. 存在递归终止条件** 分解成子问题之后，不能存在无限循环，必须要用终止条件。

如何编写递归代码

最关键的是写出递归公式，找到终止条件。

假如这里有n个台阶，每次可以跨1个台阶或者2个台阶，请问这n个台阶有多少中走法？

仔细想想，根据第一步的走法把所有走法分为两类，第一步走1个台阶，第一步走2个台阶。所以n个台阶的走法等于先走1个台阶后，n-1个台阶的走法，加上先走2个台阶，剩下n-2个台阶的走法。则递推公式就是：

$$f(n) = f(n-1) + f(n-2)$$

此时已经有了递推公式，剩下的问题就是寻找终止条件。当只有1个台阶时，只有一种走法。所以 $f(1) = 1$ ；当 $n=2$ 时， $f(2) = f(1) + f(0)$ ，如果只要 $f(1)=1$ ，那 $f(2)$ 就不能解了，则还需要 $f(0)$ ，但 $f(0)$ 没有意义，所以我们可以将 $f(2)$ 当成一个终止条件，对于2阶台阶，有两种解法，即1+1,2,所以 $f(2) = 2$ ；则终止条件就找到了，

```
f(1) = 1;
f(2) = 2;
```

所以递归代码就是：

```
int f(int n){
    if(n == 1) return 1;
    if(n == 2) return 2;
    return f(n-1) + f(n-2);
}
```


总结：写递归代码的关键就是找到如何将大问题分解成小问题的规律，并且基于此写出递归公式，然后推敲终止条件，最后将递推公式和终止条件翻译成代码。 因此编写递归代码的关键是，只要遇到递归，我们就把它抽象成一个递推公式，不用想一层层的调用关系，不要试图用人脑去分解递归的每一个步骤。

递归代码要警惕堆栈溢出

函数调用会使用栈来保存临时变量。每调用一个函数，都会将临时变量封装为栈帧压入内存栈，等函数执行完成返回时，才出栈。如果递归层次很深，就会一直压入栈，就会有堆栈溢出的风险。

可以使用限制递归调用的最大深度来解决这个问题。但这个做法也不能完全解决问题，最大允许的递归深度跟当前线程剩余的栈空间大小有关，事先无法计算。

递归代码要警惕重复计算

递归时会出现重复计算的问题。以第二个递归代码中为例子  重复计算 其中直观的看出想要计算 $f(5)$ ，需要先计算 $f(4)$ 和 $f(3)$ ，而计算 $f(4)$ 还需要计算 $f(3)$ ，因此， $f(3)$ 就被计算了很多次，这就是重复计算问题

为了避免重复计算，我们可以通过一个数据结构（散列表）来直接保存求解过的 $f(k)$ ，当再次调用到 $f(k)$ 时，就先看小是否已经求解过了，改造刚刚代码：

```
int f(int n){
    if(n == 1) return 1;
    if(n == 2) return 2;
    if(hasSolvedList.containsKey(n)){
```

```
        return hasSolvedList.get(n);
    }
    return f(n-1) + f(n-2);
}
```

所以在分析递归代码空间复杂度时，需要额外考虑这部分的开销，比如我们前面讲到的电影院递归代码，空间复杂度并不是 $O(1)$ ，而是 $O(n)$ 。

怎样将递归代码改为非递归代码呢

利是递归代码的表达力很强，写起来非常简洁；而弊就是空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗时较多等问题

电影院的非递归：

```
int f(int n){
    int ret = 1;
    for(int i = 2; i <= n; ++i){
        ret = ret + 1;
    }
    return ret;
}
```

台阶例子；

```
int f(int n){
    if(n == 1) return 1;
    if(n == 2) return 2;

    int ret = 0;
    int pre = 2;
    int prepre = 1;
    for(int i = 3; i <= n; ++i){
        ret = pre + prepre;
        prepre = pre;
        pre = ret;
    }
    return ret;
}
```

解答开篇

最终推荐人的解决方案：

```
long findRootReferrerId(long actorId) {  
    Long referrerId = elect referrer_id from [table] where actor_id = actorId;  
    if(referrerId == null) return actorId;  
    return findRootReferrerId(referrerId);  
}
```

上面的代码在实际工程中并不能工作 第一，如果递归很深，可能会有堆栈溢出的问题 第二，如果数据库中存在脏数据，我们还需要处理由此产生的无线递归问题。

课后思考

我们平时调试代码喜欢使用 IDE 的单步跟踪功能，像规模比较大、递归层次很深的递归代码，几乎无法使用这种调试方式。对于递归代码，你有什么好的调试方法呢？

1.打印日志发现，递归值。 2.结合条件断点进行调试。

斐波那契数列：

```
public class Fibonacci {  
    private static int getItem(int i){  
        if(i == 1) return 1;  
        if(i == 2) return 1;  
        return getItem(i-1) + getItem(i - 2);  
    }  
    public static void main(String[] args){  
        for(int i = 1; i < 10; i++){  
            System.out.print(getItem(i) + " ");  
        }  
    }  
}
```