**深入理解递归函数**

    刚开始接触编程对递归调用都是比较头痛，很多年前我也会一样。昨天晚上睡觉突然想起了谭浩强C语言的汉诺塔递归调用，记得当时是在高中的时候，我表姐在上大学，她把谭浩强的C语言给了我，只看书不实践，现在想起来效果还真差。其中递归调用汉诺塔看了好久都没有整明白，直到上大学学习C语言也还没有搞明白，当学到递归调用了，我就去问老师，老是说回去看看，下周告诉我。谁知到老师真的很忙，下周也没有结果。后来自己什么时候明白的也忘记了。

    刚开始接触递归都会告诉你，递归占用资源，使程序复杂，最好不要使用；还有人说，如果这个人一来就是用递归，我肯定不会聘用他。但是我认为这些观点太片面。递归算法的目的降低程序的复杂度，解放大脑负担，让大脑更加专注于问题本身。程序的性能跟递归没有什么关系，更重要的时算法本身，我们会在稍后讲解一下同一种算法同样是递归，性能的差异巨大。设计模式中，很多模式都存在递归。

    刚开始接触递归的，往往都会在里面打圈圈，自己越绕越晕，觉得递归太复杂。其实看待递归的时候，也是要分层面看待，不要把自己的大脑当做是电脑，可以绕很多的圈圈，有人说人的大脑同时能处理7个左右的变量，绕一圈就多几个变量，能绕几圈啊。呵呵。找到一个算法，在编写算法的时候，只考虑一次递归所做的事情，如果遇到到递归调用函数的时候，把他当做一个函数整体考虑，他能完成他要完成的事情，要相信他，也要相信自己。我们所在的层面就是算法的层面，或者一次执行的层面。如果在算法层面和递归调用层面来回穿插的思考，读懂递归算法将非常困难，递归的复杂度就在于压栈会导致大量的变量需要存储，对我们的大脑来说负担太重，但是对计算机来说是小意思，相对来说算法层面往往很简单，所以我们一定要站在算法层面考虑问题，而不是递归层面。

    下面来看我如何一步一步实现汉诺塔：(VS 2010 C# 控制台程序)

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {

7. }
9. **static** **void** Move(**int** n, **char** a, **char** b, **char** c)
10. {
11. **if** (n < 1) **return**;
12. **if** (n == 1)
13. {
14. MoveTo(a, c);
15. }
16. **else**
17. {
19. //以下三句可能存在问题，只是为了快速思考，先写上，看着代码找感觉。
20. Move(n - 1, a, b, c);
21. MoveTo(a, c);
22. Move(n - 1, a, b, c);
23. }
24. }
26. **private** **static** **void** MoveTo(**char** a, **char** c)
27. {
28. **throw** **new** NotImplementedException();
29. }
30. }

    凭着感觉直接写了个Move方法，第一个参数为盘子的个数，将所有盘子从 a 移动到 c。

    第一个判断，防止错误的参数。

    第二个判断，当n 等于1 时，也就是一个盘子，直接盘子从a移动到c上。

    如果多余一个，将n-1个盘子从a移到b，再将最下面一个从a移动到c，最后从b上将n-1个盘子移动到c上。

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. //这三句肯定有问题的，只是为了快速把大脑里的想法写出看，看着来找感觉。
2. Move(n - 1, a, b, c);  //这句要实现将A座n-1的盘子移动到B上。
3. MoveTo(a, c);            //这句实现将A座最下面的一个盘子移动到C上
4. Move(n - 1, a, b, c);  //这句要实现将移动到B座的盘子在移动到C座上，这就OK了。

    因为Move这个方法就是要把盘子从 a  移动到 c，所以我们在递归调用时仅仅记住这个函数的这个功能就行了，这是一个函数的整体功能。

    我们分别来讲解当N>1的3个步骤：

     第一步，讲 n-1 个盘子从 a 移动到 b 。把n-1盘子全部移动到 b 位置和 c 的方法是一样的，也就是算法是一样的。只是规模比原来少1。所以我们可以递归调用Move方法来解决将 n-1 个盘子从 a 移动到 b。 这时 b 就像相当于原来的c位置了,那么就这样调用   Move(n-1, a, c, b).

     第二步，当 n-1 个盘子从 a 移动到 b 之后，a  上就一个盘子，我们就可以直接将盘子移动到 c 上面。调用MoveTo(a, c)实现盘子的移动。在这一步，其实就是相当于移动只有一个盘子的情况，我们还可以递归调算法本身 Move(1, a, b, c);  这样调用也是可以的。在执行完成后，a 位置上是空的， b 位置上有 n-1 个盘子， c 位置上有一个最大的盘子。

    第三步，在第二步之后，我们只需要将 b 位置上的盘子都移动到 c 位置就可以了。这个和第一步类似，只是位置变了。 b 相当于原来的 a 位置（因为盘子在b上）， a 位置相当于原来的 b 位置，因为移动到 c ，所以 c 还是相当于与原来的 c 位置。 语句调用这样写 Move(n - 1, b, a, c);

    完善MoveTo方法，输出结果就好了。如果是图形移动，在这里写移动图形的方法即可。

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. **private** **static** **void** MoveTo(**char** from, **char** to)
2. {
3. Console.WriteLine(from + " -> " + to);
4. }

在Main函数中写代码测试一下：

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. Move(3, 'A', 'B', 'C');
2. Console.ReadKey();

运行一下没有问题。

将代码进行重构和调整：

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. Hanor(4, "A", "B", "C");
6. Console.ReadKey();
7. }
9. **static** **void** Hanor(**int** n, **string** platOne, **string** platTwo, **string** platThree)
10. {
11. **if** (n < 1) **return**;
12. **if** (n == 1)
13. {
14. MoveTo(platOne, platThree);
15. }
16. **else**
17. {
18. Hanor(n - 1, platOne, platThree, platTwo);
19. MoveTo(platOne, platThree);
20. Hanor(n - 1, platTwo, platOne, platThree);
21. }
22. }
24. **private** **static** **void** MoveTo(**string** a, **string** c)
25. {
26. Console.WriteLine(a + " -> " + c);
27. }
28. }

    函数执行对不对，最好不要用大脑去测试，用电那运行测试，看看运行结果正常就OK了。如果好奇，感兴趣，或者测试一下大脑，可以自己绕一绕。

    下面在举一个递归算法的例子,用于说明他递归算法的性能，代码如下：

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. **static** **int** Fun(**int** n)
2. {
3. **if** (n <= 1)
4. {
5. **return** 1;
6. }
7. **return** Fun(n - 1) + Fun(n - 1);
8. }

    这个函数的表示：当n<=1时， f(n) =1；当n>1时，f(n) = f(n-1) + f(n-1)

    这个函数这样写性能会随n的增大成指数下降，汉诺塔的性能和这个是一样的。这都是算法导致的，对于这个问题，他会调用2的（n-1）次方次，但是改变一下解决问题的算法，那么情况会是怎样？ f(n-1) +f(n-1) 其实相当于f(n-1)\*2。如果算法改成这样，结果不会有任何问题，性能却非常高，函数被调用次数变为n次。所以算法才是对新能影响的关键。修改该后的代码如下：

**[csharp]** [view plaincopyprint?](http://blog.csdn.net/dw14132124/article/details/8970539)[在CODE上查看代码片](https://code.csdn.net/snippets/97397)

1. **static** **int** Fun(**int** n)
2. {
3. **if** (n <= 1)
4. {
5. **return** 1;
6. }
7. **return** Fun(n - 1) \* 2;
8. }

    在这里有一点要注意，可能有些人担心前后两个f(n-1)的结果可能不一样，其实这种担心是因为递归的原因。之前说了，在递归调用时要把函数作为整体来看待，当做一般的函数来看待。所以这里的结果肯定会是一样的。对于汉诺塔能不能也能不性能提升到这么高呢？这也在于算法，能不能提高性能在于能不能找到算法。