摘要：

　　大师 L. Peter Deutsch 说过：To Iterate is Human, to Recurse, Divine.中文译为：人理解迭代，神理解递归。毋庸置疑地，递归确实是一个奇妙的思维方式。对一些简单的递归问题，我们总是惊叹于递归描述问题的能力和编写代码的简洁，但要想真正领悟递归的精髓、灵活地运用递归思想来解决问题却并不是一件容易的事情。本文剖析了递归的思想内涵，分析了递归与循环的联系与区别，给出了递归的应用场景和一些典型应用，并利用递归和非递归的方式解决了包括阶乘、斐波那契数列、汉诺塔、杨辉三角的存取、字符串回文判断、字符串全排列、二分查找、树的深度求解在内的八个经典问题。

版权声明：

　　本文原创作者：书呆子Rico

　　作者博客地址：http://blog.csdn.net/justloveyou\_/

友情提示：

　　若读者需要本博文相关完整代码，请移步我的Github自行获取，项目名为 SwordtoOffer，链接地址为：https://github.com/githubofrico/SwordtoOffer。

一. 引子

　　 大师 L. Peter Deutsch 说过：To Iterate is Human, to Recurse, Divine.中文译为：人理解迭代，神理解递归。毋庸置疑地，递归确实是一个奇妙的思维方式。对一些简单的递归问题，我们总是惊叹于递归描述问题的能力和编写代码的简洁，但要想真正领悟递归的精髓、灵活地运用递归思想来解决问题却并不是一件容易的事情。在正式介绍递归之前，我们首先引用知乎用户李继刚(https://www.zhihu.com/question/20507130/answer/15551917)对递归和循环的生动解释：

　　 递归：你打开面前这扇门，看到屋里面还有一扇门。你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门，你继续打开它。若干次之后，你打开面前的门后，发现只有一间屋子，没有门了。然后，你开始原路返回，每走回一间屋子，你数一次，走到入口的时候，你可以回答出你到底用这你把钥匙打开了几扇门。

　　 循环：你打开面前这扇门，看到屋里面还有一扇门。你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门（若前面两扇门都一样，那么这扇门和前两扇门也一样；如果第二扇门比第一扇门小，那么这扇门也比第二扇门小，你继续打开这扇门，一直这样继续下去直到打开所有的门。但是，入口处的人始终等不到你回去告诉他答案。

　　 上面的比喻形象地阐述了递归与循环的内涵，那么我们来思考以下几个问题：

什么是递归呢？

递归的精髓(思想)是什么？

递归和循环的区别是什么？

什么时候该用递归？

使用递归需要注意哪些问题？

递归思想解决了哪些经典的问题？

这些问题正是笔者准备在本文中详细阐述的问题。

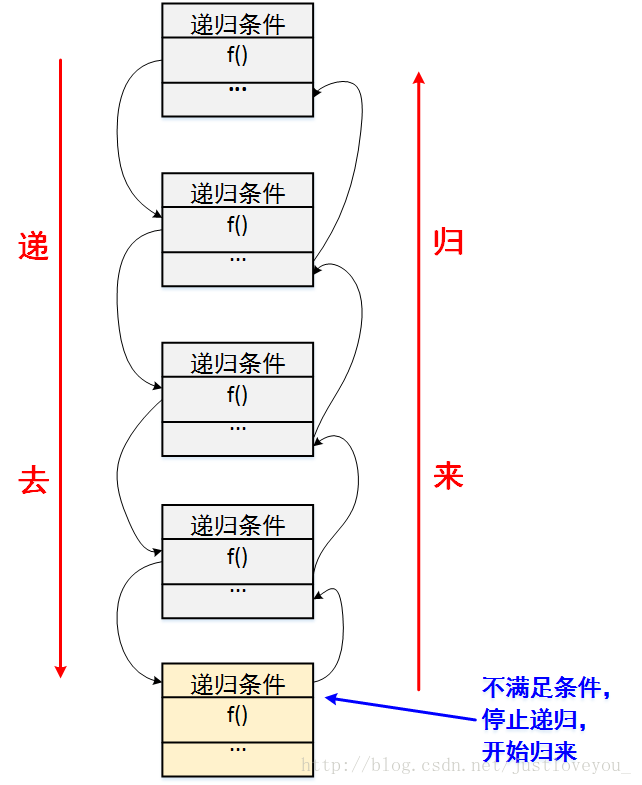
二. 递归的内涵

1、定义 (什么是递归？)

　　 在数学与计算机科学中，递归(Recursion)是指在函数的定义中使用函数自身的方法。实际上，递归，顾名思义，其包含了两个意思：递 和 归，这正是递归思想的精华所在。

2、递归思想的内涵(递归的精髓是什么？)

　　 正如上面所描述的场景，递归就是有去（递去）有回（归来），如下图所示。“有去”是指：递归问题必须可以分解为若干个规模较小，与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决，就像上面例子中的钥匙可以打开后面所有门上的锁一样；“有回”是指 : 这些问题的演化过程是一个从大到小，由近及远的过程，并且会有一个明确的终点(临界点)，一旦到达了这个临界点，就不用再往更小、更远的地方走下去。最后，从这个临界点开始，原路返回到原点，原问题解决。



　　 更直接地说，递归的基本思想就是把规模大的问题转化为规模小的相似的子问题来解决。特别地，在函数实现时，因为解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数调用它自身的情况，这也正是递归的定义所在。格外重要的是，这个解决问题的函数必须有明确的结束条件，否则就会导致无限递归的情况。

3、用归纳法来理解递归

　　 数学都不差的我们，第一反应就是递归在数学上的模型是什么，毕竟我们对于问题进行数学建模比起代码建模拿手多了。观察递归，我们会发现，递归的数学模型其实就是 数学归纳法，这个在高中的数列里面是最常用的了，下面回忆一下数学归纳法。

　　 数学归纳法适用于将解决的原问题转化为解决它的子问题，而它的子问题又变成子问题的子问题，而且我们发现这些问题其实都是一个模型，也就是说存在相同的逻辑归纳处理项。当然有一个是例外的，也就是归纳结束的那一个处理方法不适用于我们的归纳处理项，当然也不能适用，否则我们就无穷归纳了。总的来说，归纳法主要包含以下三个关键要素：

步进表达式：问题蜕变成子问题的表达式

结束条件：什么时候可以不再使用步进表达式

直接求解表达式：在结束条件下能够直接计算返回值的表达式

事实上，这也正是某些数学中的数列问题在利用编程的方式去解决时可以使用递归的原因，比如著名的斐波那契数列问题。

4、递归的三要素

　　 在我们了解了递归的基本思想及其数学模型之后，我们如何才能写出一个漂亮的递归程序呢？笔者认为主要是把握好如下三个方面：

明确递归终止条件；

给出递归终止时的处理办法；

提取重复的逻辑，缩小问题规模。

1). 明确递归终止条件

　　 我们知道，递归就是有去有回，既然这样，那么必然应该有一个明确的临界点，程序一旦到达了这个临界点，就不用继续往下递去而是开始实实在在的归来。换句话说，该临界点就是一种简单情境，可以防止无限递归。

2). 给出递归终止时的处理办法

　　 我们刚刚说到，在递归的临界点存在一种简单情境，在这种简单情境下，我们应该直接给出问题的解决方案。一般地，在这种情境下，问题的解决方案是直观的、容易的。

3). 提取重复的逻辑，缩小问题规模\*

　　 我们在阐述递归思想内涵时谈到，递归问题必须可以分解为若干个规模较小、与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决。从程序实现的角度而言，我们需要抽象出一个干净利落的重复的逻辑，以便使用相同的方式解决子问题。

5、递归算法的编程模型

　　 在我们明确递归算法设计三要素后，接下来就需要着手开始编写具体的算法了。在编写算法时，不失一般性，我们给出两种典型的递归算法设计模型，如下所示。

模型一： 在递去的过程中解决问题

function recursion(大规模){

if (end\_condition){ // 明确的递归终止条件

end; // 简单情景

}else{ // 在将问题转换为子问题的每一步，解决该步中剩余部分的问题

solve; // 递去

recursion(小规模); // 递到最深处后，不断地归来

}

}

模型二： 在归来的过程中解决问题

function recursion(大规模){

if (end\_condition){ // 明确的递归终止条件

end; // 简单情景

}else{ // 先将问题全部描述展开，再由尽头“返回”依次解决每步中剩余部分的问题

recursion(小规模); // 递去

solve; // 归来

}

}

6、递归的应用场景

　　 在我们实际学习工作中，递归算法一般用于解决三类问题：

　　 (1). 问题的定义是按递归定义的（Fibonacci函数，阶乘，…）；

　　 (2). 问题的解法是递归的（有些问题只能使用递归方法来解决，例如，汉诺塔问题，…）；

　　 (3). 数据结构是递归的（链表、树等的操作，包括树的遍历，树的深度，…）。

　　在下文我们将给出递归算法的一些经典应用案例，这些案例基本都属于第三种类型问题的范畴。

---------------------

作者：书呆子Rico

来源：CSDN

原文：https://blog.csdn.net/justloveyou\_/article/details/71787149

版权声明：本文为博主原创文章，转载请附上博文链接！