什么是递归?

递归其实就是某个函数直接或者间接的调用了自身。这种调用方式叫递归调用。说白了还是一个函数调用。

既然是函数调用，那么就有一个雷打不动的原则：所有被调用的函数都将创建一个副本，各自为调用者服务，而不受其他函数的影响。

递归的条件?

递归函数分为两个条件，边界条件和递归条件。

边界条件：就是递归中止条件，避免出现死循环。也叫做递归出口。

递归条件：也就是递归体。将一个大问题分解为一步步小问题。也是递归调用的阶段。

递归函数在具备这两个要素以后，才可以在有限次的计算后得出想要的结果。

示例

```
/**
 * 用递归方式实现1+2+3+4+5
 */
public class test {

    //定义一个变量，存放和
    private static int account = 0;

    /**
     * 主函数入口
     * @param args
     */
    public static void main(String[] args) {
        //调用递归函数
        account = add(5);
        //打印输出
        System.out.println(account);
    }

    /**
     * 递归
     * @param x
     * @return
     */
    public static int add(int x){
        //边界条件 递归中止条件
        if (x == 1){
```

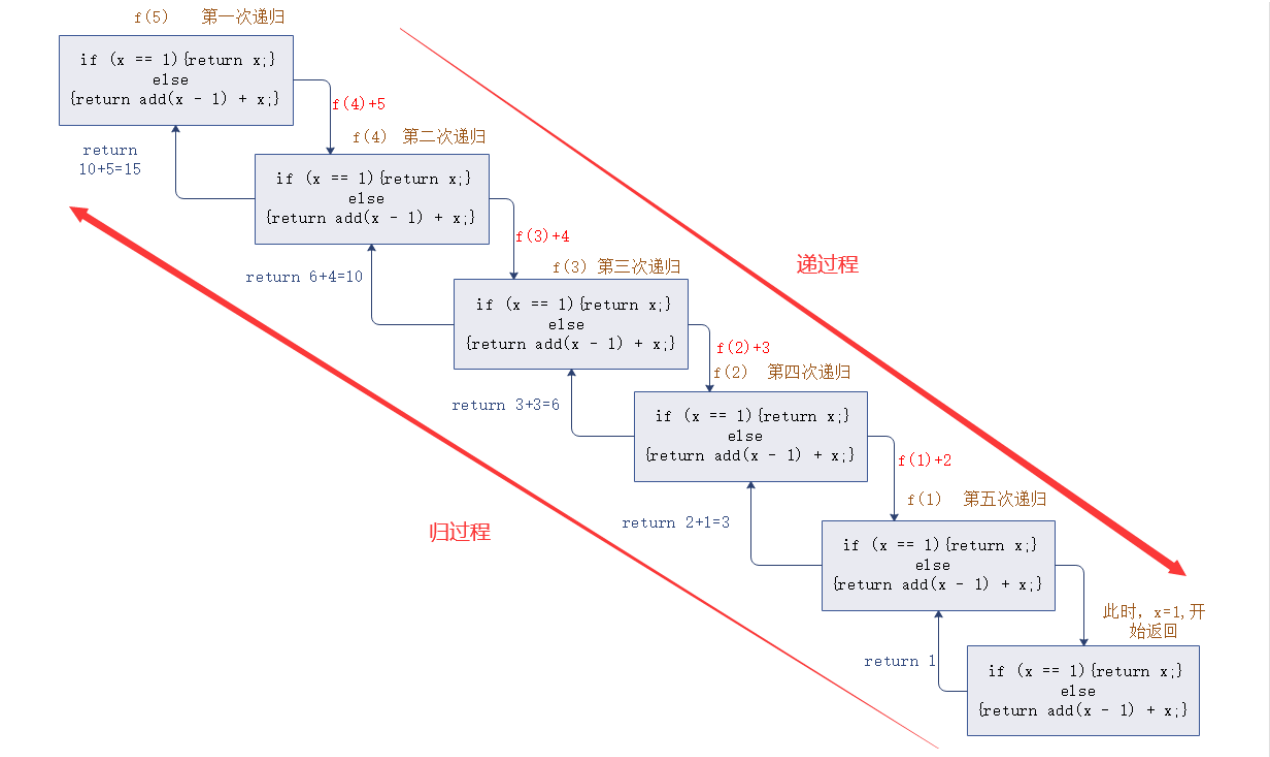
```
            //若x == 1,返回他本身
            return x;
        }else {
            //递归体 递归调用
            return add(x - 1) + x;
        }
    }
}
```

输出结果:

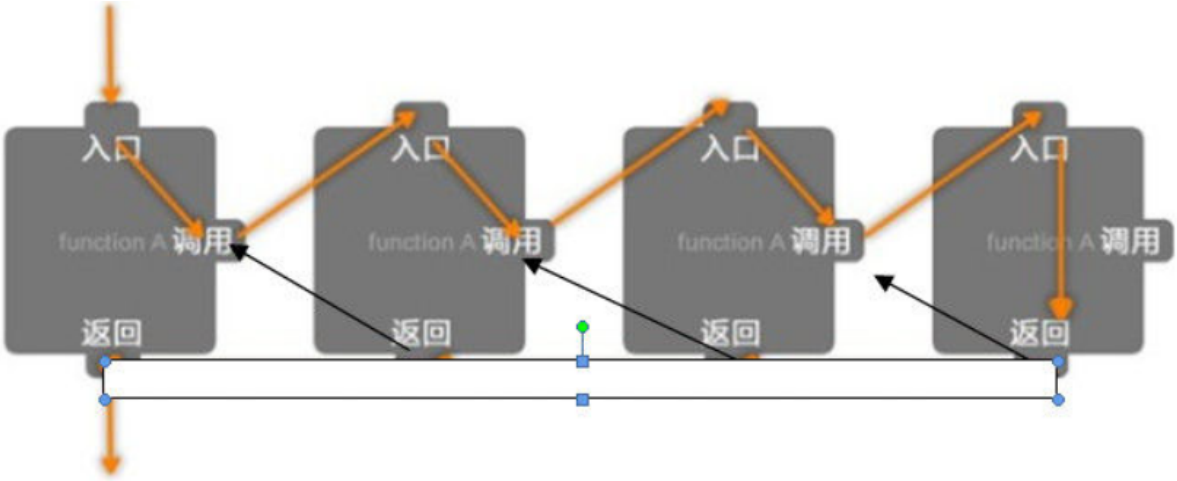
```
"E:\JetBrains\IntelliJ IDEA 2019.2\jbr\bin\java.exe" "-
15

Process finished with exit code 0
```

递归图示:



从上面的描述我们可以看出，递归其实就是在自己调自己的过程中，存在前进和退回的阶段。我们这里用褚跃跃的一张图也能比较清楚地反映出这个过程。



是否真的需要递归

不建议滥用。要分清楚具体场景。不建议使用，无非就是怕你太菜了，写不对边界条件，直接把栈写穿了。

在很多特定场景下。肯定是要用递归的。比如无限级树。解析xml文件。

我没听说过有这样的规定。你在日常能接触到的领域如果需要用到递归，那么特征应该是很明显的，比如遍历文件系统或任意层次的树结构。对于普通业务逻辑，如果你能把非递归的问题硬要用递归来解决，那我也是服的，但目前为止我还没有遇到过这样自找麻烦的人。其实大多数意外的递归问题（比如说错误把消息发送给自己导致无限循环）是由于对架构不熟悉、底层原理不了解导致的，这需要靠加强学习和引导来避免，并不是制定一条规范就能解决的。

有几个人用event driven写xml解析的？高性能的部分就那么一点点，专门优化就好了，其他部分，能怎么懒，能怎么方便就怎么写。工程上阅读性比奇怪的优化策略要有用得多，不是每个人都能搞个怪胎sqrt函数去优化性能的，你不是那种大佬，写这种代码唯一的结果就是被reviewer砍死。

尾递归

在传统的递归中，典型的模型是首先执行递归调用，然后获取递归调用的返回值并计算结果。以这种方式，在每次递归调用返回之前，您不会得到计算结果。传统地递归过程就是函数调用，涉及返回地址、函数参数、寄存器值等压栈（在x86-64上通常用寄存器保存函数参数），这样做的缺点有二：

效率低，占内存

如果递归链过长，可能会statck overflow

若函数在尾位置调用自身（或是一个尾调用本身的其他函数等等），则称这种情况为尾递归。尾递归也是递归的一种特殊情形。尾递归是一种特殊的尾调用，即在尾部直接调用自身的递归函数。对尾递归的优化也是关注尾调用的主要原因。尾调用不一定是递归调用，但是尾递归特别有用，也比较容易实现。

尾递归的原理

当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。编译器可以做到这点，因为递归调用是当前活跃期内最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其他事情可做，因此也就没有保存栈帧的必要了。通过覆盖当前的栈帧而不是在其之上重新添加一个，这样所使用的栈空间就大大缩减了，这使得实际的运行效率会变得更高。

```
/**
 * 用递归方式实现1+2+3+4+5
 */
public class test {

    //定义一个变量，存放和
    private static int account = 0;

    /**
     * 主函数入口
     * @param args
     */
    public static void main(String[] args) {
        //调用递归函数
        account = add(5);
        //打印输出
        System.out.println(account);
    }

    /**
     * 递归
     * @param x
     * @return
     */
    public static int add(int x,int sum){
        //边界条件 递归中止条件
        if (x == 1){
            //若x == 1,返回他本身
            return sum;
        }else {
            //递归体 递归调用
            return add(x - 1,sum + x);
        }
    }
}
```

特点

尾递归在普通尾调用的基础上，多出了2个特征：

在尾部调用的是函数自身 （Self-called）；

可通过优化，使得计算仅占用常量栈空间 (Stack Space)。

说明

传统模式的编译器对于尾调用的处理方式就像处理其他普通函数调用一样，总会在调用时创建一个新的栈帧（stack frame）并将其推入调用栈顶部，用于表示该次函数调用。

当一个函数调用发生时，电脑必须 “记住” 调用函数的位置 —— 返回位置，才可以在调用结束时带着返回值回到该位置，返回位置一般存在调用栈上。在尾调用这种特殊情形中，电脑理论上可以不需要记住尾调用的位置而从被调用的函数直接带着返回值返回调用函数的返回位置（相当于直接连续返回两次）。尾调用消除即是在不改变当前调用栈（也不添加新的返回位置）的情况下跳到新函数的一种优化（完全不改变调用栈是不可能的，还是需要校正调用栈上形式参数与局部变量的信息。）

由于当前函数帧上包含局部变量等等大部分的东西都不需要了，当前的函数帧经过适当的更动以后可以直接当作被尾调用的函数的帧使用，然后程序即可以跳到被尾调用的函数。产生这种函数帧更动代码与 “jump”（而不是一般常规函数调用的代码）的过程称作尾调用消除(Tail Call Elimination)或尾调用优化(Tail Call Optimization, TCO)。尾调用优化让位于尾位置的函数调用跟 goto 语句性能一样高，也因此使得高效的结构编程成为现实。

然而，对于 C++ 等语言来说，在函数最后 return g(x); 并不一定是尾递归——在返回之前很可能涉及到对象的析构函数，使得 g(x) 不是最后执行的那个。这可以通过返回值优化来解决。

在尾递归中，首先执行计算，然后执行递归调用，将当前步骤的结果传递给下一个递归步骤。这导致最后一个语句采用的形式(return (recursive-function params))。基本上，任何给定递归步骤的返回值与下一个递归调用的返回值相同。