**什么是尾调用？**

尾调用（Tail Call）是函数式编程的一个重要概念，就是指某个函数的最后一步是调用另一个函数。

function f(x){

return g(x);

}

最后一步涉及计算，赋值等其他操作的都不是尾调用。

**尾调用优化**

尾调用之所以与其他调用不同，就在于它的特殊的调用位置。

我们知道，函数调用会在内存形成一个“调用记录”，又称“调用帧”（call frame），保存调用位置和内部变量等信息。如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，以此类推。所有的调用帧，就形成一个“调用栈”（call stack）。

尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

这就叫做“尾调用优化”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

**尾递归**

函数调用自身，称为递归。如果尾调用自身，就称为尾递归。

递归非常耗费内存，因为需要同时保存成千上百个调用帧，很容易发生“栈溢出”错误（stack overflow）。但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。

示例1 普通递归：

function factorial(n) {

if (n === 1) return 1;

return n \* factorial(n - 1);

}

factorial(5) // 120

//计算n的阶乘，最多需要保存n个调用记录，复杂度 O(n)

示例2 尾递归：

function factorial(n, total) {

if (n === 1) return total;

return factorial(n - 1, n \* total);

}

factorial(5, 1) // 120

//只保留一个调用记录，复杂度 O(1)