

图解算法二

递归——一种优雅的问题解决方案

结论：

- 递归条件：递归函数调用自己的条件
- 基线条件：递归函数结束调用自己的条件
- **递归只是让解决方案更清晰，并没有性能上的优势。**（不考虑尾递归）

递归的实现中，系统会对变量重新分配内存空间（普通循环是覆盖）

递归只有走到最后结束函数（基线条件）时才会释放栈内存

因此递归次数多时可能造成内存分配崩溃 **Stack overflow**

注：避免内存崩溃可以使用高级递归中尾递归，但受使用语言和场景约束，这种方式极少使用，故这里不考虑。

实践出真知

- 为了验证在大多数场景下，循环性能优于递归所以使用测试用例
- 以JS为例使用常用调试函数测量一个JS脚本程序执行消耗的时间

`console.time()`开始代码执行计时

`console.timeEnd()`停止计时，输出脚本执行的时间。

```
/*-----递归-----*/

console.time('testRecursive')
function Recursive(num) {
  if (num > 1) {
    return num * Recursive(num - 1)
  } else
    return num;
}
Recursive(8000);
console.timeEnd('testRecursive')

/*-----传统循环-----*/

console.time('testFor')
var flag = 1;
for (var x = 8000; x > 1; x--) {
  flag *= x;
}
console.timeEnd('testFor')
```

Chrome 81.0.4044.122	Recursive	For	结果
第一次	testRecursive: 2.110107421875ms	testFor: 0.942138671875ms	For
第二次	testRecursive: 2.089111328125ms	testFor: 1.848876953125ms	For
第三次	testRecursive: 2.156005859375ms	testFor: 0.7099609375ms	For
第四次	testRecursive: 3.24902343475ms	testFor: 1.453857421875ms	For
第五次	testRecursive: 1.301025390625ms	testFor: 1.299072265625ms	For
第六次	testRecursive: 1.69580078125ms	testFor: 0.6347655625ms	For

由以上可知：常用的for循环性能确实是优于递归。

原理：递归的实现是通过重复调用函数本身，而调用的时候每次都要保存局部变量、形参、返回值等，这些都会影响代码执行的效率。

栈：调用栈 (call stack) 在编程中是一个非常重要的概念。

计算机在内部使用被称之为调用栈的栈

调用栈主要功能：保存调用的返回地址。

- 举个例子

```
function Poo(name){
  console.log(`Hi,I'M ${name}`);
  getHobby(name);
  Bye(name);
}
function getHobby (name){
  console.log(`${name}的爱好是编程~`);
}
function Bye(name){
  console.log(`欢迎你下次再来看${name}的文章`);
}
Poo('potato')
```

1. 在这个函数中，调用Poo()时，计算机首先为该函数调用分配一块内存
2. 比如使用这块内存，首先变量 name 被设置成 potato, 这需要存储在内存中

- 每当调用函数的时候，计算机都像这样把函数涉及到的变量值存储到内存内存中

初始调用 Poo('potato')

- 执行Poo函数时,先赋值存储变量到内存中
- 然后打印 Hi,I'M potato

Poo

Poo	
name	potato

再调用 `getHobby(name)` 同样计算机也为它分配内存，第二个内存块位于第一个内存块上方**（压栈）**

getHobby	
name	potato
Poo	
name	potato

`getHobby(name)` 执行完毕，打印 `'potato的爱好是编程'` 函数调用返回，此时栈顶的内存块会**弹出**

getHobby	
name	potato
◦ 此时上方的内存块会分离 （弹出）	
Poo	
name	potato

此时栈顶的函数是 `Poo()` ,意味着返回到了函数 `Poo('potato')` , 总体来看函数 `Poo('potato')` 只执行了一部分

- **注：这是一个很重要的概念，在函数中调用另一个函数时，当前函数暂停，并处于未完成的状态，该函数的所有变量的值都还在内存中，如例子中一样，执行完 `getHobby(name)` 又回到了 `Poo('potato')`，然后从离开的地方接着往下执行。**

然后调用函数 `Bye(name)` 在栈顶添加这个函数的内存块，重复上述步骤，执行打印后从栈顶弹出。

Bye	
name	potato
Poo	
name	potato

这个栈用于存储多个函数的变量，因此被称之为**调用栈**

总结

由以上特性可知：

- 递归在性能上并没有优势，只是代码更易解读而已。

- 在实际使用场景中调用栈有可能会很长将占用大量的内存，造成内存崩溃隐患