

递归函数 2.7旧版教程

阅读: 210842

在函数内部,可以调用其他函数。如果一个函数在内部调用自身本身,这个函数就是递归函数。

举个例子, 我们来计算阶乘 n! = 1 x 2 x 3 x ... x n , 用函数 fact(n) 表示, 可以看出:

```
fact(n) = n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n = (n-1)! \times n = fact(n-1) \times n
```

所以,「fact(n)可以表示为 n x fact(n-1) ,只有n=1时需要特殊处理。

于是,fact(n) 用递归的方式写出来就是:

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试:

如果我们计算 fact(5), 可以根据函数定义看到计算过程如下:

```
==> fact(5)
==> 5 * fact(4)
==> 5 * (4 * fact(3))
==> 5 * (4 * (3 * fact(2)))
==> 5 * (4 * (3 * (2 * fact(1))))
==> 5 * (4 * (3 * (2 * 1)))
==> 5 * (4 * (3 * 2))
==> 5 * (4 * 6)
==> 5 * 24
==> 120
```

递归函数的优点是定义简单,逻辑清晰。理论上,所有的递归函数都可以写成循环的方式,但循环的逻辑不如递归 清晰。

使用递归函数需要注意防止栈溢出。在计算机中,函数调用是通过栈(stack)这种数据结构实现的,每当进入一个函数调用,栈就会加一层栈帧,每当函数返回,栈就会减一层栈帧。由于栈的大小不是无限的,所以,递归调用的次数过多,会导致栈溢出。可以试试 fact(1000):

```
Traceback (most recent call last):

File "sdin>", = line 1, in <module>

File "<stdin>", line 4, in fact

...

File "<stdin>", line 4, in fact

RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过尾递归优化,事实上尾递归和循环的效果是一样的,所以,把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指,在函数返回的时候,调用自身本身,并且,return语句不能包含表达式。这样,编译器或者解释器就可以把尾递归做优化,使递归本身无论调用多少次,都只占用一个栈帧,不会出现栈溢出的情况。

上面的 [fact(n)] 函数由于 [return n \* fact(n - 1)] 引入了乘法表达式,所以就不是尾递归了。要改成尾递归方式,需要多一点代码,主要是要把每一步的乘积传入到递归函数中:

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到, return fact\_iter(num - 1, num \* product) 仅返回递归函数本身, num - 1 和 num \* product 在函数调用前就会被计算,不影响函数调用。

fact(5) 对应的 fact iter(5, 1) 的调用如下:

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时,如果做了优化,栈不会增长,因此,无论多少次调用也不会导致栈溢出。

遗憾的是,大多数编程语言没有针对尾递归做优化,Python解释器也没有做优化,所以,即使把上面的 fact(n) 函数改成尾递归方式,也会导致栈溢出。

### 小结

使用递归函数的优点是逻辑简单清晰,缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的,没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化,任何递归函数都存在栈溢出的问题。

#### 练习

汉诺塔的移动可以用递归函数非常简单地实现。

请编写[move(n, a, b, c)]函数,它接收参数[n],表示3个柱子A、B、C中第1个柱子A的盘子数量,然后打印出把所有盘子从A借助B移动到C的方法,例如:

```
# 期待输出:
# 和 -- > C
# A -- > B
# C -- > B
# A -- > C
# B -- > C
# A -- > C
move(3, 'A', 'B', 'C')
```

► Run

# 参考源码

### recur.py

感觉本站内容不错,读后有收获?

¥ 我要小额赞助,鼓励作者写出更好的教程

## 还可以分享给朋友

分享 晨星\_V5,哎哟不错耶等4人分享过







评论

发表评论

Sign In to Make a Comment



<u>廖雪峰的官方网站</u>©2015 Powered by <u>iTranswarp.js</u> 由<u>阿里云</u>托管 广告合作



友情链接: <u>中华诗词</u> - <u>阿里云</u> - <u>SICP</u> - <u>4clojure</u>