# 第3章 递归

递归是一种强大的方法，它允许一个对象以其自身更小的形式来定义自己。恐怕没有什么比观察神秘的自然界中出现的递归现象更好的方法来体会递归的重要意义了。想想蕨类植物的叶子，每片叶子的小枝干都是整片叶子的较小缩影；又或者两个反光的物体，相互映射对方的渐远的影像。这样的例子使我们明白尽管大自然的力量是强大的，在许多方面它那种出乎意料的简洁更让我们觉得优美。同样的道理也可用在递归算法上，从很多方面来说递归算法都是简洁而优美的，而且还非常强大。

在计算机科学领域，递归是通过递归函数来实现的。递归函数是一种可以调用自身的函数。每次成功的调用都使得输入变得更加精细，使我们越来越接近问题的答案。大多数开发者都喜欢将大型的问题分解成一些小型的问题，然后分别编写不同的函数去处理它们。然而，许多开发者却并不习惯于用单一函数递归的去解决大型的问题。诚然，以这种方式去看待问题需要一定的适应过程。本章探讨了递归是如何工作的并展示了如何以递归的方式去定义问题。本书中的一些递归方面的例子有树的遍历（见第9章），图中的广度优先和深度优先查找（见第11章），以及排序（见第12章）。

本章的内容包括：

*基本递归*

一种强大的方法，允许一个问题以其自身越来越小的形式来定义自己。在计算机科学领域，我们通过使用递归函数来解决带有递归性质的问题，也就是用函数自身调用自身。

*尾递归*

递归的一种形式，这里编译器会为此产生优化过的代码。大多数现代的编译器能够识别出尾递归。为此，只要条件允许我们都应该利用这个特性。

## 3.1 基本递归

开始吧，首先来看一个通常我们不会以递归的形式去思考的问题。假设我们想计算整数的阶乘。的阶乘可写作，其结果是从1到之间的各数之积。比如，。一种计算法方法是循环遍历其中的每一个数，然后与它之前的数相乘作为结果再参与下一次计算。这种方法称为迭代法，可以正式定义为：。看待这个问题的另一种方式是将定义为更小的阶乘形式。为了实现这一步，我们将定义为倍的的阶乘。当然，解决的过程同一样，只是变小了一些。如果我们再把看做倍的，看做倍的，一直到时，我们就计算完了。这就是递归的方式，可以正式定义为：



图3.1展示了利用递归的方法计算的过程。它也勾画出了递归过程中的两个基本阶段：递推，回归。在递推阶段，每一个递归调用通过进一步调用自己来记住这次递归过程。当其中有调用满足了终止条件时，递推结束。比如，在计算的阶乘时，终止条件是当 和 ，此时函数只需简单的返回1即可。每一个递归函数都必须拥有至少一个终止条件；否则递推阶段就永远不会结束了。一旦递推阶段结束，处理过程就进入回归阶段，在这之前的函数调用以逆序的方式回归，直到最初调用的函数返回为止，此时递归过程结束。

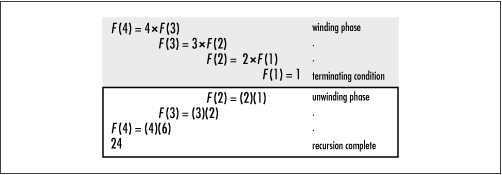


图3.1 以递归的方式计算4的阶乘

代码示例3.1展示了一个C函数fact，它接受一个整数作为参数，以递归的方式计算的阶乘。该函数按照如下的方式工作：如果小于0，函数直接返回0，这代表一个错误。如果等于0或者1，函数返回1，这是因为和都等于1，以上就是终止递归的条件。否则，函数返回倍的的阶乘。而的阶乘又会以递归的方式再次调用fact来计算，如此继续。注意观察递归实现与我们之前对递归的定义之间的相同点。

**代码示例3.1 以递归方式计算阶乘的代码实现**

#include "fact.h"

int fact(int n) {

if (n < 0)

return 0;

else if (n == 0)

return 1;

else if (n == 1)

return 1;

else

return n \* fact(n - 1);

}

为了理解递归究竟是如何工作的，有必要先看看C语言中函数的执行方式。基于这点，我们需要了解一点C程序在内存中的组织方式。基本上来说一个可执行程序由4个区域组成：代码段，静态数据区，堆，堆栈（见图3.2a）。代码段包含了程序运行时所执行的机器指令。静态数据区包含了在程序生命期内一直持久的数据，比如全局变量和静态局部变量。堆中包含了程序运行时动态分配的存储空间，比如用malloc分配的内存。堆栈中包含了函数调用的信息。按照惯例，堆的增长方向为从程序低地址到高地址向上增长，而堆栈的增长方向刚好相反（实际情况可能不是这样，与CPU的体系结构有关）。注意，这里用到的术语堆与我们在第10章中介绍的数据结构堆没有什么关系。

当C程序中调用了一个函数时，堆栈中会分配一块空间来保存与这个调用相关的信息。每一个调用都被当作是活跃的。堆栈上的那块存储空间被称为活跃记录，或者称为栈帧。栈帧由5个区域组成：输入参数、返回值空间、计算表达式时用到的临时存储空间、函数调用时保存的上下文信息以及输出参数（见图3.2b）。输入参数是传递到活跃记录中的参数；输出参数是传递给在活跃记录中调用的函数所使用的。一个活跃记录中的输出参数就成为堆栈中下一个活跃记录的输入参数。函数调用产生的活跃记录将一直存在于堆栈中直到这个函数调用结束返回。

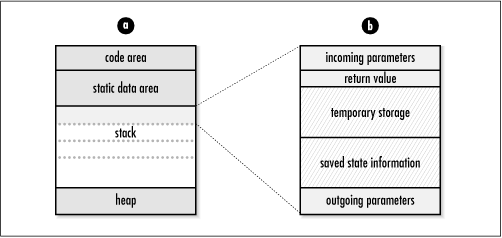


图3.2 C程序在内存中的组织形式（a），以及一份活跃记录（b）

回到示例3.1，考虑一下当计算时堆栈中都发生了些什么。初始调用fact会在堆栈中产生一个活跃记录，输入参数（见图3.3，第1步）。由于这个调用没有满足终止条件，fact将继续以为参数递归调用。这将在堆栈上创建另一个活跃记录，但这次输入参数（见图3.3，第2步）。这里，也是第一个活跃期中的输出参数，因为正是在第一个活跃期内调用fact产生了第二个活跃期。这个过程将一直继续，直到的值变为1，此时满足了终止条件，fact将返回1（见图3.3，第4步）。

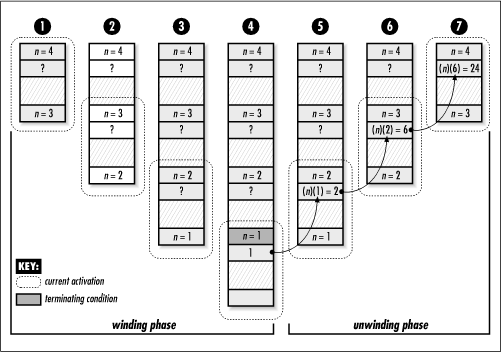


图3.3 递归计算时的堆栈情况

一旦当时的活跃期结束，那么时的递归计算结果就是，因而时的活跃期也将结束，返回值为2（见图3.3，第5步）。结果就是时的递归计算结果表示为，因此时的活跃期结束，返回值为6（见图3.3，第6步）。最终，当时的递归计算结果将表示为，时的活跃期将结束，返回值为24（见图3.3， 第7步）。此时，函数已经返回到了最初的调用中，递归过程结束。

堆栈是用来存储函数调用信息的绝好方案，这正是由于其后进先出的特点（见第6章）精确满足了函数调用和返回的顺序。然而，使用堆栈也有一些缺点。堆栈维护了每个函数调用的信息直到函数返回后才释放，这需要占用相当大的空间，尤其是在使用了许多递归调用的情况下。除此之外，因为有大量的信息需要被保存和恢复，因此建立和销毁活跃记录需要耗费一定的时间。如此一来当函数调用的开销变得很大时，我们就需要考虑应该采用迭代的方案。幸运的是，我们可以采用一种称为尾递归的特殊递归方式来避免前面提到的这些缺点。

## 3.2 尾递归

如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。当递归调用是整个函数体中最后被执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。尾递归函数的特点是在回归过程中不用做任何操作，这个特性很重要，因为大多数现代的编译器会利用这种特点自动生成优化过的代码。

当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活跃记录而不是在堆栈中去创建一个新的。编译器可以做到这点，因为递归调用是最后一条被执行到的语句，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，这样所使用的堆栈空间就被大大缩减了，这使得实际的运行效率会变的更高。因此，只要有可能我们就需要将递归函数写成尾递归的形式。

为了理解尾递归是如何工作的，让我们再次以递归的形式计算阶乘。首先，这可以很容易让我们理解为什么之前所定义的递归不是尾递归。回忆之前对计算的定义：在每个活跃期计算倍的的值，让并持续这个过程直到为止。这种定义不是尾递归的，因为每个活跃期的返回值都依赖于用乘以下一个活跃期的返回值，因此每次调用产生的栈帧将不得不保存在堆栈上直到下一个子调用的返回值被确定。现在让我们考虑以尾递归的形式来定义计算的过程。我们可以定义成如下形式：



这种定义还需要接受第二个参数，除此之外并没有太大区别。（初始化为1）维护着递归层次的深度。这就让我们避免了每次还需要将返回值再乘以。图3.4说明了用尾递归计算的过程。注意在回归的过程中不需要做任何操作，这是所有尾递归函数的标志。

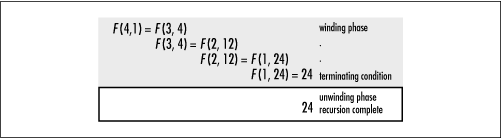


图3.4 以尾递归的方式计算

代码实例3.2给出了一个C函数facttail，它接受一个整数并以尾递归的形式计算的阶乘。这个函数还接受一个参数，的初始值为1。facttail使用来维护递归层次的深度，除此之外它和fact很相似。读者可以注意一下函数的具体实现和尾递归定义的相似之处。

**代码示例3.2 以尾递归的形式计算阶乘**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ------------------------- facttail.c ----------------------------\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "facttail.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* --------------------------facttail ------------------------------\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int facttail(int n, int a) {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*

\* 以尾递归的方式计算阶乘

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (n < 0)

return 0;

else if (n == 0)

return 1;

else if (n == 1)

return a;

else

return facttail(n - 1, n \* a);

}

示例3.2中的函数是尾递归的，因为对facttail的递归调用是函数返回前的最后一条语句。在facttail中碰巧最后一条语句也是对facttail的调用，但这并不是必须的。换句话说，在递归调用之后还可以有其他的语句执行，只是它们只能在递归调用没有执行时才可以执行。图3.5展示了当使用尾递归计算时堆栈使用的情况，读者可以和图3.3展示的堆栈使用情况做下对比。

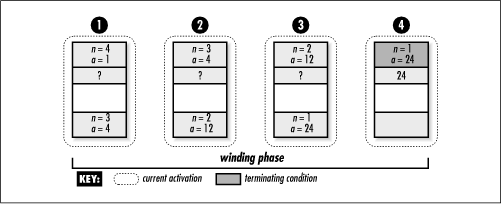


图3.5 以尾递归形式计算时堆栈的情况

## 3.3 问与答

问：下面的递归定义有一个错误。请指出错误并想想该如何修正？给定一个正整数，其正式的定义通常用来计算分治算法的运行时间，比如归并排序（见第12章）。归并排序将一组数据一分为二，然后分别将两份数据各自再进行分半处理，一直持续这个过程直到每一份中都只含一个元素。然后在回归过程中完成各份数据的合并最终产生一个有序的集合。



答：该定义的问题在于当的初始值大于等于0时将永远无法满足终止条件。为了解决问题，我们需要一个满足要求的终止条件。这个条件就能很好满足，这意味着我们也要修改函数中的第二个条件。合适的递归定义应该是这样的：



这恰好是归并排序运行时间的正确定义形式。这样一种函数称为“递推”。在更正式的分析中，递推常用来描述递归算法的运行时间。

问：以递归的思想描述一种求解整数质因子的方法。分析该方法是否是尾递归的并解释原因。

答：递归是一种很自然的求解整数质因子的方法，因为因子分解无非就是不断的解决同样的问题。每当我们确定了一个因子则剩余因子的集合就变得越来越小。针对这个问题的递归方法可以定义为如下的式子：



这个定义的意思是说：为了递归的确定整数的质因子，先确定它的最小质因子并把它记录到集合中，然后对整数重复这个过程直到本身成为质数为止，这就是终止条件。这个定义是尾递归的，因为在回归过程中不需要做任何处理，如图3.6所示。

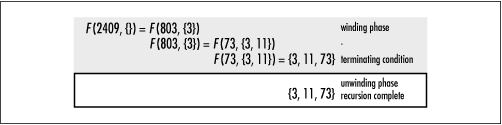


图3.6 以尾递归的方式计算整数2409的质数因子

问：思考当执行递归函数时堆栈的使用情况，当递归的递推永远不会终止时会出现什么情况？也许如同第一个问题中那样，终止条件是畸形的。

答：如果递归函数的终止条件永远得不到满足，最终堆栈的增长会超过可接受的值，程序会因为堆栈溢出而终止运行。当程序执行时，一个被称为堆栈指针的特殊指针会记录栈顶的位置。它也可用来确定堆栈是否增长到太大的地步，此时会产生一个中断来通报这个错误。

问：递归函数常常能够以简洁而精确的方式描述计算问题。请描述下面的递归定义式所要表达的计算是什么：



答：这个递归定义计算出了我们称之为调和级数的数列之和。对于给定的正整数，该函数计算出到第位的调和级数数列之和。（计算过程与下面给出的调和级数序列是相反的，但下面的方式更好看一些）



问：上一个问题中的函数是尾递归的吗？如果是，请解释原因。如果不是，解释原因并给出一个尾递归的版本。

答：上个问题中定义的函数不是尾递归的。因为递归调用的返回值在一个表达式中。这个表达式称为当前调用的返回值。因此，每一个栈帧都必须保留在堆栈上直到下一个子调用得到它的返回值为止。为了使这个函数成为尾递归，我们可以利用本章前面提到的以尾递归方式计算阶乘的方法来处理。我们增加一个参数来记录当前递归中计算的数列之和。最终，上个问题的尾递归版本可以表示成：



## 3.4 相关主题

编译器设计

代码编译背后的基本思想最终都是要求程序能够高效的运行，至少得在机器指令这个级别。而一般在算法设计中我们主要精力都集中在控制算法的复杂度上（见第4章），理解编译器编译代码过程中的林林总总能够帮助我们在实践中调整程序的性能。理解尾递归就是个很好的例子。

尾递归消除

在函数中将最终的尾递归调用替换为等同的迭代控制结构，这不会改变函数的输出结果但能够避免一次额外的函数调用开销。尾递归消除在编译器设计中是一个重要的基本法则。

递归树

画图表能帮助我们形象化的理解递归函数的调用顺序。递归树在形式上有所不同，展示递归计算阶乘的图3.1和图3.4与表示确定整数质因子的图3.6都是递归树。递归树最常用在表示包含2个或更多的递归调用的函数中。