[读书笔记]SICP：2[B]过程及其产生的计算过程

本章对应于书中的1.2。

- 需要区分过程和计算过程，过程是从语法形式来判断的，而计算过程是计算过程的展开形状。过程二就是递归过程产生了迭代计算过程，这就使得该递归过程只需要解释器维护状态变量，也就能在常数空间中执行递归过程，具有该特点的程序设计语言实现称为**尾递归**。Scheme的实现就具有尾递归的特点，而有些语言，比如C，即使递归过程的计算过程是迭代计算过程，解释器还是需要以递归计算过程的形式来维护，当该类语言中想要得到迭代计算过程，就需要特殊的循环形式。
- 通过增加几个状态变量将递归计算过程变为迭代计算过程

之前只考虑了程序设计中的一些要素，现在需要对**计算过程（Process）**中各种动作的执行情况作出规划，然后用一个程序去控制这一计算过程的进展。我们需要学会去看清不同种类的**过程（Procedure）**会产生什么样的计算过程。过程就是计算过程**局部演化**的模式，描述了这一计算过程中的每步是怎么基于之前的步骤建立起来的。我们这里将考察一些简单过程产生的计算过程的形状，研究这些计算过程消耗的各种资源。

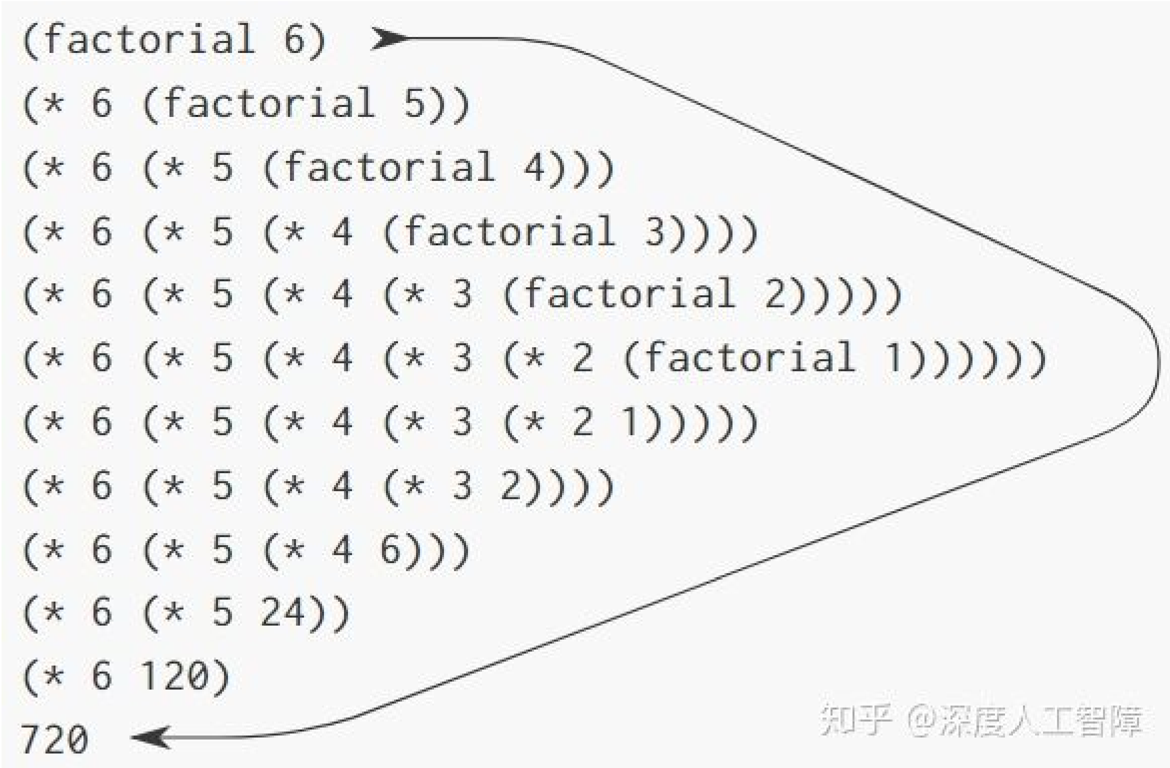
1 线性的递归和迭代计算过程

首先考虑阶乘函数$n! = n\cdot(n-1)...3\cdot2\cdot1$。

最简单的计算方式是将其计算过程转化为$n! = n\cdot(n-1)!$，由此可以得到以下代码

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

通过代换模型可以得到它的计算过程



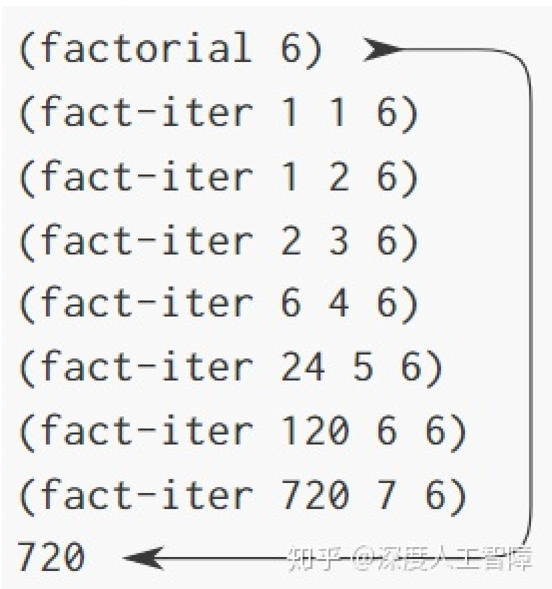
另一种计算方法是维护一个乘积结果`product`和从1到n的计数器`counter`，模拟从1到n的计算过程，则这两个参数按照以下规则变化

```
product = product * counter
counter = counter + 1
```

可以得到以下代码

```
(define (factorial n)
  (define (factorial-iter product counter max-counter)
    (if (> counter max-counter)
        product
        (factorial-iter (* counter product)
                        (+ counter 1)
                         max-counter)))
  (factorial-iter 1 1 n))
```

通过代换模型可以得到它的计算过程



首先，从语法形式上来看，由于这两个过程的定义中直接或间接地引用了该过程本身，所以这两个过程都是**递归过程**。其次，从计算过程方面来看，相同的递归过程却有不同的计算过程形状

- **过程一**：其代换模型展现出了一种先逐步展开而后收缩的形状
-
    - **展开阶段**：计算过程构造起一个推迟执行的操作形成的链条
    - **收缩阶段**：这些操作的实际执行
      这种计算过程由一个推迟执行的运算链条刻画的称为**递归计算过程**，运算链条隐含了计算的信息，解释器需要维护好那些以后要执行的操作的轨迹，当链条越长，解释器需要维护的信息越多。可以发现过程一中运算链条的长度与n值呈线性关系，所以该计算过程称为**线性递归计算过程**。
- **过程二**：计算过程没有任何增长或收缩，计算过程中的每一步都只需要保存`product`、`counter`和`max-counter`，这种称为**迭代计算过程**，该计算过程一般**具有以下几方面**：
-
    - 状态可用固定数目的状态变量描述
    - 存在一套固定的规则，描述了计算过程从一个状态到下一个状态转换时，状态变量的更新方式
    - 可能还存在一个结束检测，描述这一计算过程的终止条件
      可以发现，在迭代计算过程中的任意一步停下来，都能通过状态变量重启，因为状态变量提供了有关计算状态的完整描述，解释器也只需要维护这些状态变量的轨迹。可以发现过程二中的计算步骤随n值线性增长，该计算过程又称为**线性迭代计算过程**。

注意：需要区分过程和计算过程，过程是从语法形式来判断的，而计算过程是计算过程的展开形状。过程二就是递归过程产生了迭代计算过程，这就使得该递归过程只需要解释器维护状态变量，也就能在常数空间中执行递归过程，具有该特点的程序设计语言实现称为**尾递归**。Scheme的实现就具有尾递归的特点，而有些语言，比如C，即使递归过程的计算过程是迭代计算过程，解释器还是需要以递归计算过程的形式来维护，当该类语言中想要得到迭代计算过程，就需要特殊的循环形式。

例子：以下有两个过程，都基于过程`inc`和`dec`

```
; 过程一
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
; 过程二
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

首先，由于该两个过程都在内部调用了自身，所以都是递归过程。然后以`(+ 4 5)`为例查看计算过程的展开形状

```
 ; 过程一的展开形状
(+ 4 5)
(inc (+ 3 5))
(inc (inc (+ 2 5)))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9

;过程二的展开形状
(+ 4 5)
(+ 3 6)
(+ 2 7)
(+ 1 8)
(+ 0 9)
```

```
9
```

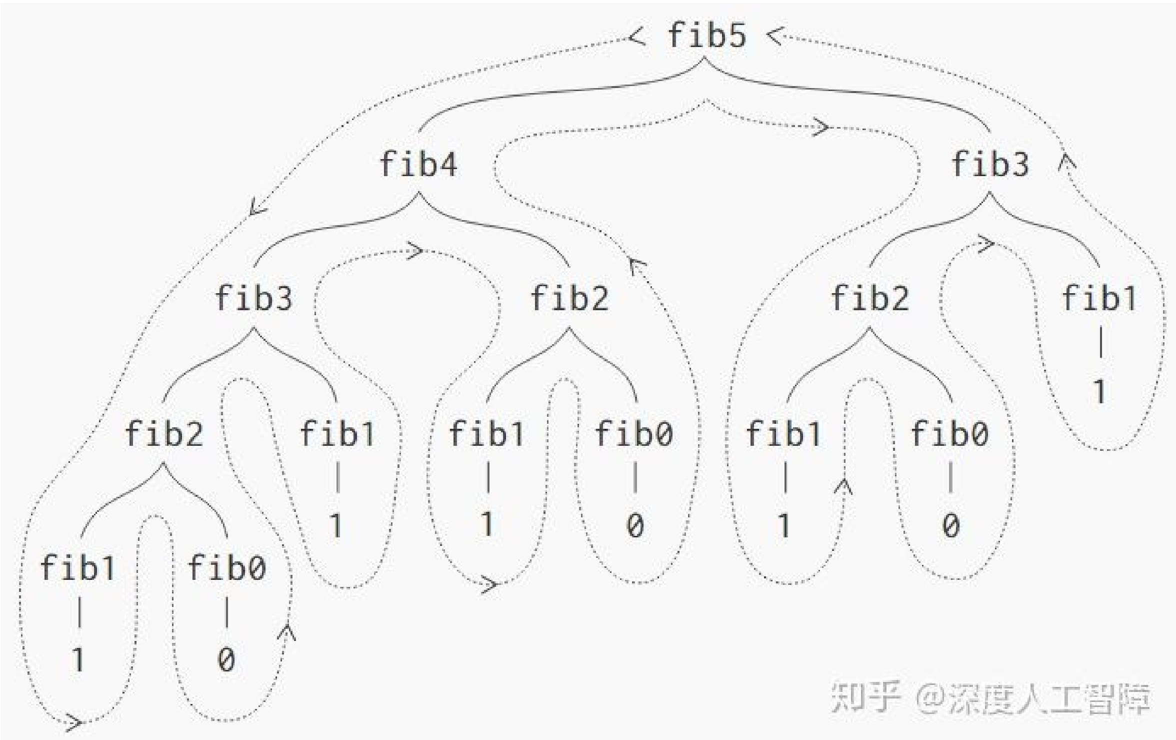
可以很明显发现，过程一有一个逐步展开而后收缩的形状，所以过程一是递归计算过程，而过程二没有任何变化，且只有两个状态变量`a`和`b`，所以过程二是迭代计算过程。

2 树形递归

以菲波那切数列为例，对应的递归过程为

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

`fib(n)`的值接近 $\phi^n/\sqrt{5}$，其中 $\phi=(1+\sqrt{5})/2$。通过代换模型可以得到它计算过程的展开形式如下所示，是一个**树形递归计算过程**



由于`fib`每次调用都会两次递归调用自身，所以每一层都会有两个分支。可以发现它的计算效率很低，计算步骤数随着输入增加而呈指数型增长，而空间需求呈线性增长，因为要计算每一个点，只需要保存树中在此之上的节点轨迹。通常可以将树形递归计算过程中所需的步骤数正比于树中的节点数，而空间需求正比于树的最大深度。

我们可以利用和之前相似的方法，通过两个状态变量分别记录`fib(n-1)`和`fib(n-2)`，将其转化为迭代计算过程，对应的代码为

```
(define (fib n)
  (define (fib-iter a b count)
    (if (= count n)
        b
        (fib-iter (+ a b) a (- count 1))))
  (fib-iter 1 0 n))
```

它是一个线性迭代计算过程，所需的步骤数相对n为线性的。

虽然树形递归计算过程计算效率较低，但在层次结构性的数据上操作时，会更加自然且强大，比如解释器就是用树形的通用计算过程来求值表达式的。

树形递归的思想在于将大的复杂问题拆解为小的子问题，根据小的子问题来得到最终问题的答案。比如总共有几种方法将总数为`a`的现金换成`n`种硬币，此时如果能减少现金或减少硬币总数，都是原始问题的简化问题，对应的两个分支为：

- 减少硬币：现金`a`换成除第一种硬币以外的其他硬币，总共的方法数量
- 减少现金：现金`a-d`换成所有种类的硬币的不同方式数量，`d`为第一种硬币的值

通过将这两个分支相加就得到了最终答案，其实也是树形递归模式。

3 增长的阶

我们这里使用增长的阶来衡量当输入变大时。过程所需资源的粗略度量情况。`n`是一个参数，作为问题规模的一种度量，可以是函数需要计算的数等等，总会存在某个有关问题特性的数值，我们可以根据它来分析给定的计算过程。`R(n)`是一个计算过程在处理规模为`n`的问题所需的资源量，可以是寄存器数目，可以是执行步骤，由于计算机在每个时刻只能执行固定数目的操作，所以所需的时间将正比于需要执行的基本机器指令条数。

我们规定`R(n)`具有`O(f(n))`的增长阶，如果存在与`n`无关的整数`k1`和`k2`使得 $k_1 f(n) \le R(n) \le k_2 f(n)$ 。比如之前介绍的计算阶乘时，递归计算过程所需步骤的增长阶为`O(n)`，所需空间的增长阶为`O(n)`，而迭代计算过程所需步骤的增长阶为`O(n)`，而所需口空间的增长阶为`O(1)`。

接下来将考察两个增长阶为对数型的算法，也就是当问题规模增加一倍，所需的资源量只增加一个常数。

3.1 求幂

我们先要求一个幂 b^n ，可以用递归计算过程

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

它需要`O(n)`步和`O(n)`空间（解释器需要维护运算链）。我们也可以得到迭代计算过程

```
(define (expt b n)
  (define (expt-iter b counter product)
    (if (= counter 0)
        product
        (expt-iter b
                   (- counter 1)
                   (* product b))))
  (expt-iter b n 1))
```

它需要`O(n)`步和`O(1)`空间。跟进一步，我们可以用**连续求平方思想**，将求幂过程转化为：如果`n`为偶数，则 $b^n = (b^{n/2})^2$ ，如果`n`为奇数，则 $b^n = b \cdot b^{n-1}$ 。对应的代码为

```
(define (fast-expt b n)
  (define (even? n)
    (= (remainder n 2) 0))
  (cond ((= n 0) 1)
        ((even? n) (fast-expt (square b) (/ n 2)))
        (else (* b (fast-expt b (- n 1))))))
```

当我们用迭代计算过程计算 b^n 需要`O(n)`步，而当我们用连续求平方来计算 b^{2n} 时，只是多了一步乘积计算，可以发现问题规模增加了一倍，而步骤数只增加了1，所以该计算过程的增长阶为`O(logn)`。

上面这个计算过程是递归计算过程，我们同样可以将其转化为迭代计算过程

```
(define (fast-expt-i b n)
  (define (even? n)
    (= (remainder n 2) 0))
  (define (fast-expt-iter a b n)
    (cond ((= n 0) a)
          ((even? n) (fast-expt-iter a
                                     (square b)
                                     (/ n 2)))
          (else (fast-expt-iter (* b a)
                                b
                                (- n 1)))))
  (fast-expt-iter 1 b n))
```

注意：在迭代计算过程中，通常需要定义一个不变量，要求其在状态转移时不变，比如这里将 $a \cdot b^n$ 作为不变量。

其实这种**折半思想**可以应用到很多方面，比如可以用反复加法的方式求解乘积，可以将`a*b`的计算过程转化为：如果`b`为偶数，则 $a \times b = 2(a \times b/2)$，如果`b`为奇数，则 $a \times b = a \times (b-1) + a$，对此我们需要定义一个加倍的运算`double`和一个减半的运算`halve`，对应的加入折半思想的递归计算过程代码为

```
(define (even ? b)
    (= (remainder b 2) 0))
  (define (double n)
    (+ n n))
  (define (halve n)
    (/ n 2))

(define (* a b)
  (cond
    ((= b 0) 0)
    ((even? b) (* (double a) (halve b)))
    (else (+ (* a (- b 1)) a))))
```

同样也可以将其转化为迭代计算过程

```
(define (* a b)
  (define (*-iter a b n)
    (cond
      ((= b 0) n)
      ((even? b) (*-iter (double a)
                         (halve b)
                         n))
      (else (*-iter a
                    (- b 1)
                    (+ n a)))))
  (*-iter a b 0))
```

综上所述：

- **加入折半思想**：
-
    - 将计算过程展开，比如 $b^n = b \times b \times b ... \times b$ 或 $b * n = b + b + b ... + b$

-
  - 如果个数`n`为偶数，则没有余项，就将其两两结合并组合起来，比如 $b^n=(b\times b)\times(b\times b)\times\ldots\times(b\times b)=(b^2)^{n/2}$ 或 $b*n=(b+b)+(b+b)+\ldots+(b+b)=(2b)\times(n/2)$
  - 如果个数`n`为奇数，则将其化为偶数个整体和一个余项的形式，比如 $b^n=b^{n-1}\times b$ 或 $b\times n=b\times(n-1)+b$
 - 将代码分成这两种情况分别递归
- **将递归计算过程转变为迭代计算过程**：通常会引入一个不变量，使得它和当前子问题的结果复合后就为真实结果。

3.2 最大公约数

求解最大公约数可以用[欧几里得算法]进行求解，对应的迭代计算过程为

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcb b (remainder a b))))
```

而这里存在一个**Lame定理**：如果欧几里得算法需要用k步计算出一对整数的GCD，那么这对数中较小的那个数必然大于或等于第k个斐波那契额数。

当`n`为较小的那个值，而对应的第`k`个斐波那契额数为 $\phi^k/\sqrt{5}$，则 $n\geq\phi^k/\sqrt{5}$，则欧几里得算法的增长阶为`O(logn)`。

3.3 素数检测

3.3.1 寻找因子

最简单的检查某个数`n`是否为素数，直接遍历2到 $\sqrt{n}$ 查看是否有`n`的因子，如果有则不是素数

```
(define (prime? n)
  (define (divides? a b)
    (= (remainder b a ) 0))
  (define (find-divisor n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divides? test-divisor n) test-divisor)
          (else (find-divisor n (+ test-divisor 1)))))
  (define (smallest-divisor n)
    (find-divisor n 2))
  (= n (smallest-divisor n)))
```

该算法的步数具有 $O(\sqrt{n})$ 增长阶。此外，可以发现当一个数如果不能被2整除，那它就无法被其他的偶数整除，可以减少许多判断条件，可以修改为以下形式

```
(define (prime? n)
  (define (divides? a b)
    (= (remainder b a ) 0))
  (define (next n)
    (if (= n 2)
        3
        (+ n 2)))
  (define (find-divisor n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divides? test-divisor n) test-divisor)
```

```
        (else (find-divisor n (next test-divisor))))) ;这里
  (define (smallest-divisor n)
    (find-divisor n 2))
  (= n (smallest-divisor n)))
```

大约可以提速1.5倍。

3.3.2 费马检查

这主要基于**费马小定理**：如果`n`是一个素数，`a`是小于`n`的任意正整数，则`a`的`n`次方与`a`模`n`同余（由于`a，所以余数就是`a`）。如果`n`不是素数，则大部分的`a都不满足这个关系。

我们可以随机选取一个`a，然后计算出 ![[公式]](https://www.zhihu.com/equation?tex=a%5En) 取模`n`的余数，如果不为`a`，则`n`一定不是素数，否则`n`很大可能会是素数。当我们判断一个`a`满足关系，则`n`是素数的概率大于50%，当我们取的`a`越多，`n`是素数的概率就越大，这种算法称为**概率算法**，该算法称为**费马检查**。这里也存在一些特殊的数`a`会使得不是素数的`n`也满足以上关系，称为**Carmichael数**，但是这类数特别少，所以费马检查基本是很可靠的。

```
; 首先要有一个过程计算一个数的幂对另一个数取模的结果，这里使用连续求平方的方式计算幂
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m))
        (else (remainder (* base (expmod base (- exp 1) m)) m))))

; 可以定义只有一个随机a的费马检查
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

; 包装成检查若干次的费马检查
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

这里之所以不使用以下形式来求解一个数的幂对另一个数取模，是因为以上的结果会不断对结果中间结果通过`remainder`过程保持在较小的数，而以下使用`faster-expt`会产生很大的中间结果，在较大的数上进行计算比较耗时。

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

参考：http://community.schemewiki.org