函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第四讲

以序列作为程序的接口

●例1：求一棵树里值为奇数的树叶的平方和：

```
(define (square x)
  (* x x))
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

以序列作为程序的接口

- 例2：构造斐波那契数列第k项Fib(k) 的表，其中Fib(k) 是偶数且k <= n

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

以序列作为程序的接口

两个程序的流程有共同点：

例1：

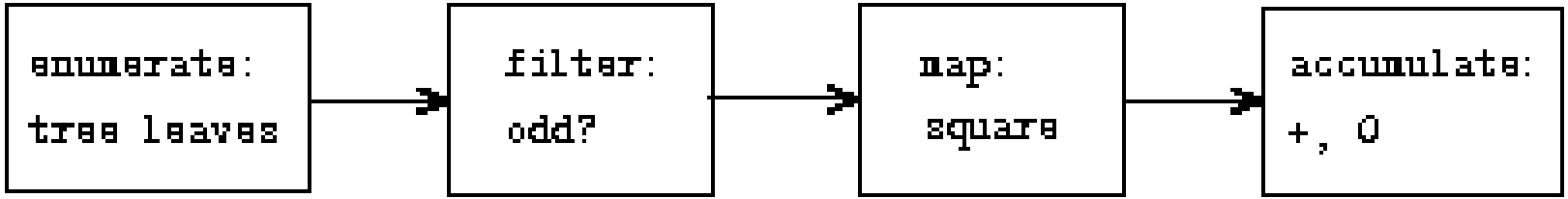
1） 枚举树中所有树叶
2） 滤出其中的奇数
3） 对选出的数求平方
4） 用**+ 累积它们，从0 开始**

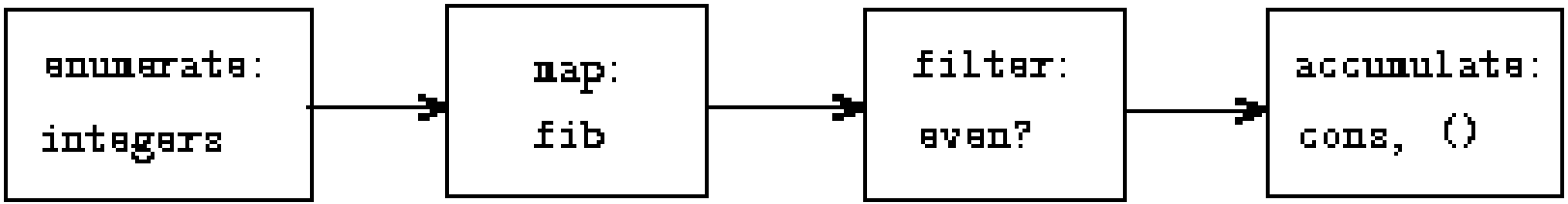
例2：

1）枚举从**0 到n 的整数**
2）对每个数**k 算出Fib(k)**
3）滤出其中的偶数
4） 用**cons 累积它们，从nil 开始**

以序列作为程序的接口

两过程实现的处理都可以看作串联起的一些步骤，每步完成一项具体工作，信息在步骤之间流动：





上面两个程序的写法，不能很好体现这个步骤。应当写出一些过程，概括和抽象出这个流程。即：过滤，然后累积过滤后的结果。

以序列作为程序的接口

对序列的过滤操作filter：

```
(define (filter predicate sequence)
  (cond ((null? sequence) '())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

序列中使得 predicate(x) 为#t的元素x被保留，形成结果序列

```
(filter even? (list 1 2 3 4 5)) => '(2 4)
```

以序列作为程序的接口

对序列的累积操作accumulate：

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

op(s0 (op s1 (op s2 (op……(op sn init))….))))

```
(accumulate + 0 (list 1 2 3 4 5)) => 15
```

以序列作为程序的接口

区间枚举操作：

```
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ low 1) high))))
```

对[low,high]这个区间内的元素，每隔1取一个，形成一个序列。

```
(enumerate-interval 2 7) ;=>(2 3 4 5 6 7)
```

以序列作为程序的接口

枚举一棵树的所有树叶，形成一个序列：

```
(define (enumerate-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))


(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
;=>(1 2 3 4 5)
```

以序列作为程序的接口

重写sum-odd-squares求一棵树里值为奇数的树叶的平方和：

```
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                   (filter odd? (enumerate-tree tree)))))
```

以序列作为程序的接口

重写even-fibs求第n项内的斐波那契数列中偶数项的和

```
(define (even-fibs n)
  (accumulate cons
              '()
              (filter even?
                      (map fib (enumerate-interval 0 n)))))
```

以序列作为程序的接口

综上所例：

有时我们可以把程序模块写成针对序列的一系列操作，是用序列（表）作为不同模块之间的标准接口

这些模块还便于重用，组合形成新的功能模块

以序列作为程序的接口

- 例3：求前n+1 个斐波纳契数的平方：

```
(define (list-fib-squares n)
  (accumulate cons '()
              (map square
                   (map fib (enumerate-interval 0 n)))))

(list-fib-squares 10)
;=> (0 1 1 4 9 25 64 169 441 1156 3025)
```

以序列作为程序的接口

- 例4：求一个序列中所有奇数的平方的乘积：

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate * 1
   (map square (filter odd? sequence))))


(product-of-squares-of-odd-elements (list 1 2 3 4 5))
;=>225
```

以序列作为程序的接口

●例5：从人事记录里找出薪金最高的程序员的工资。假定**salary** 返回记录里的工资，**programmer?** 检查是否程序员：

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max
              0
              (map salary (filter programmer? records))))
```

;max是scheme基本过程

(max r0 (max r1 (max r2 (max ……(max rn 0)))…)

嵌套的映射

●例6：找出所有不同的i 和j 使1 <= j < i <= n 且i + j 是素数。

n = 6 时：

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
i + j	3	5	5	7	7	7	11

嵌套的映射

- 例6：找出所有不同的i 和j 使1 <= j < i <= n 且i + j 是素数。

- 思路：先找出所有对子(i,j) (其中j<i)，形成一个序列，然后再在该序列上过滤

```
(define (make-list n)
  (map (lambda (i) ;对 [1,n]中的每个元素i
       (map (lambda (j) ; 然后对[1,i-1]这个区间里的每个元素j
              (list i j)) ;生成一个序列 (i j)
            (enumerate-interval 1 (- i 1)))) ;先生成一个区间 [1,i-1]
    (enumerate-interval 1 n))) ;生成一个区间 [1,n]

(make-list 4)
=>'( () ((2 1)) ((3 1) (3 2)) ((4 1) (4 2) (4 3)) )
```

嵌套的映射

- 希望把 (() ((2 1)) ((3 1) (3 2)) ((4 1) (4 2) (4 3)))

平摊成： **((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))** 才好过滤

```
(define (make-pairs n)
  (accumulate append
              '()
              (make-list n)))

(make-pairs 4)
=>'((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
```

(append s1 (append s2 (append……(append sn `())…)))

s1: () s2: ((2 1)) s3: ((3 1) (3 2)) ……

嵌套的映射

- 编写通用的平摊映射过程 flapmap:

```
(define (flatmap proc seq)
  (accumulate append '() (map proc seq)))

;假定proc作用在seq的每个元素上ei，得到的结果都是一个序列 si
;flatmap 把所有si中的元素抽取出来，拼成一个序列

;即 (map proc seq)得到的结果可能是:  (( 1 2) (3 4 ) (5 6))
;而经过flatmap后，即得到 ( 1 2 3 4 5 6)
```

嵌套的映射

- 用flapmap重新定义make-pairs:

```
(define (make-pairs n)
  (flatmap (lambda (i) ;对 [1,n]中的每个元素i
             (map (lambda (j) ; 然后对[1,i-1]这个区间里的每个元素j
                    (list i j)) ;生成一个序列 (i j)
                  (enumerate-interval 1 (- i 1))))
           (enumerate-interval 1 n)))

(make-pairs 4)
```

```
=>'((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))
```

嵌套的映射

●例6：找出所有不同的i 和j 使1 <= j < i <= n 且i + j 是素数。最终结果：

```
(define (prime-sum? pair) (prime? (+ (car pair) (cadr pair))))
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))


(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
               (make-pairs n))))

(prime-sum-pairs 6)
=>'((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7) (6 1 7) (6 5 11))
```

嵌套的映射

- 例7：生成全排列的O(n!)的算法

生成集合S里所有元素的全排列。对集合**S** 里的每个**x** 生成**S – {x}** 的所有排序的序列，而后将**x** 加在这些序列的最前面，就得到以**x** 开头的所有排序序列。把对**S** 里每个**x** 生成的以**x** 开头的序列连起来，就得到了结果

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))
(define (permutations s)
  (if (null? s) ; empty set?
      (list '()) ; sequence containing empty set
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

(permutations (list 1 2 3 4)) ;=> ((1 2 3 4) (1 2 4 3) .....)

嵌套的映射

- 如果把 (list '()) 换成 '()会怎样?

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))
(define (permutations s)
  (if (null? s) ; empty set?
      '()   ;******** changed
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

(permutations (list 1 2 3 4)) ;=> ?

嵌套的映射

- 如果把 (list '()) 换成 '()会怎样?

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))
(define (permutations s)
  (if (null? s) ; empty set?
      '()   ;******** changed
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

(permutations (list 1 2 3 4)) ;=> '()

嵌套的映射

- 为什么要用 flatmap而不能用map?

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))
(define (permutations s)
  (if (null? s) ; empty set?
      (list '())
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

嵌套的映射

```
s = (1)
(remove 1 s) = ()
(permutations (remove 1 s)) => (())

则: (map (lambda (p) (cons x p))
                         (permutations (remove 1 s))))
=> ((1))

则:   (map (lambda (x)
                  (map (lambda (p) (cons x p))
                       (permutations (remove x s))))
                s)))
=> (((1)))   ,应该是 ((1))才对，所以要用 flatmap
```

嵌套的映射

```
s = ( 1 2 3 )
(remove 2 s) = (1 3)

假定 (permutations (remove 2 s)) => ((1 3) (3 1))
则: (map (lambda (p) (cons x p))
                        (permutations (remove 2 s))))
=> ((2 1 3) (2 3 1))
则:   (map (lambda (x)
                  (map (lambda (p) (cons x p))
                       (permutations (remove x s))))
                s)))
=> (  ((2 1 3)(2 3 1))   ((1 2 3) (1 3 2))  ((3 1 2)(3 2 1))   )
```

不是想要的结果，应该将其拉平。用 **flatmap** 则无此问题

流水线

更加抽象和清晰的做法是定义流水线过程 **pipeline:**

```
(pipeline operand op1 op2 ... opn) ; 任意多个参数 operand是序列

(define (even-fibs n)
  (pipeline (enumerate-interval 1 n)
            (lambda (lst) (map fib lst))
            (lambda (lst) (filter even? lst))
            (lambda (lst) (accumulate cons '() lst)) ))
```

留作思考题