

# 道语言函数式编程

September 19, 2009

(适用道语言1.1)

## 1 一般语法

道语言通过一系列内置方法支持函数式编程。这种内置方法的一般语法是这样的：

```
RESULT = METHOD( PARAMETER ) -> |VARIABLE| { EXPRESSION }  
结果 = 方法( 参数 ) -> |变量| { 表达式 }
```

这里METHOD可以是：

1. map
2. reduce/fold
3. unfold
4. select
5. index
6. count
7. each
8. repeat

也可以是string, array或list，它们可以以这种语法用作数据构造方法。其他两个采用同样语法的内置方法是sort和apply。

取决于是什么METHOD，参数PARAMETER可以是一个或多个列表，数组或其他数据类型。可选的VARIABLE可以根据METHOD和PARAMETER来申明。如果|VARIABLE|被省略，那么x,y,z和i,j将在需要的情况下被自动申明。

大括号里的 **EXPRESSION** 相当于一个内联的函数，它将被应用到 **PARAMETER** 或它（如果是列表或数组）的元素上。**PARAMETER** 或其元素将通过 **VARIABLE** 里声明的或自动声明的变量传递给 **EXPRESSION**。

一个简单的例子：

```
a = { 1, 2, 3 }
b = map( a ) -> { 10*x } # produce {10, 20, 30 }
```

它等价于：

```
a = { 1, 2, 3 }
b = map( a ) -> |x| { 10*x }
```

当多个列表或数组被用作参数时，**VARIABLE** 可以被相应的定义为那些列表数组的元素：

```
a = { 1, 2, 3 }
b = { 11, 22, 33 }
c = map( a, b ) -> |x,y| { x + y }
```

在此例中，**x** 被定义为了 **a** 的元素，而 **y** 则被定义为了 **b** 的元素。实际上这里还可再申明一个变量表示 **x+y** 被执行时当前的元素下表，如：

```
c = map( a, b ) -> |x,y,i| { x + y + i }
```

其实这里，**|x,y,i|** 也可被省略，这几个变量都将被自动申明。

大括号里也可使用多个表达式，它们必需由逗号分开，它们的值将以元组(tuple)返回：

```
a = { 1, 2, 3 }
b = map( a ) -> { 10*x, x+10 } # produce {(10, 11), (20, 12), (30, 13) }
```

要使用 **程序语句**，只需在语句后面 **EXPRESSION** 前加关键字 **return** 即可，如：

```
... -> |VARIABLE| { BLOCK return EXPRESSION }
```

这里 **BLOCK** 表示一般程序语句（不能使用再使用 **return** 关键字）。例如：

```
a = { 1, 2, 3 }
```

```

b = map( a ) -> |x| {
  u = 10*x;
  v = x+10;
  if( u < v ) u += v;
  return u*v
}

```

复合函数可以通过使用多个->|...|{...} 结构创建:

```

RESULT = METHOD( PARAMETER )
-> |VARIABLE| { EXPRESSION }
-> |VARIABLE| { EXPRESSION }
...

```

对于除了第一个之外的->|...|{...} 结构里的 VARIABLE, 变量将根据前一结构里的 EXPRESSION 来申明。例如:

```

a = { 1, 2, 3 }
b = { 11, 22, 33 }
c = map( a, b ) -> |x,y| { x + y, x - y } -> |u,v| { u * v }

```

这里,  $u$  被申明为  $x+y$  的结果,  $v$  被申明为  $x-y$  的结果。

## 2 map, reduce/fold, unfold

### 2.1 map()

map() 以一个或多个列表或数组作参数, 并将大括号里的内联函数应用到 列表数组的每个元素, 然后把所有函数应用的结果作为列表或数组返回。这些作为参数的列表数组必需有同样的长度。

如前面提到的, map() 可以以数组为参数并返回一个数组。如果 (最后的那个 大括号里的) 内联函数返回一个标量值, map() 返回的数组将和 (第一个) 参数里的数组有同样的形状, 即拥有同样的维数且每个维数上的大小一样。如果一个元组, 列表或数组被返回, map() 返回的数组将相对于参数里的数组有同样的额外的纬度, 而返回的元组列表数组等将被用来设定相应的自数组的值。

前面已经给出了一些用在列表上的map()的例子, 这里在给些使用在数组上的例子:

```

a = [ 1, 2; 3, 4 ]
b = map( a ) -> { 10*x } # produce [ 10, 20; 30, 40 ]

```

```

a = [ 1, 2; 3, 4 ]
b = [ 5, 6; 7, 8 ]
c = map( a, b ) -> { x, y }
# or
c = map( a, b ) -> { [x, y] }
# or
c = map( a, b ) -> { {x, y} }
# produce 3d array: [ [ [1, 5], [2, 6] ], [ [3, 7], [4, 8] ] ]

```

在其他拥有函数式编程方法的语言里可以经常看到zip()函数， Dao没有专门支持此函数，因为此函数可以相当容易的由map()函数实现。如：

```

a = { 1, 2, 3 }
b = { 11, 22, 33 }
c = map( a, b ) -> { x, y }

```

这将返回一个以元组zip那两个列表里元素的列表。显然可以用同样的方式来zip多个列表。

## 2.2 reduce/fold()

fold()和等同的方法reduce()，以一个列表或数组和一个初始值（也是累积值）为参数，将内联的函数依次应用到列表数组的每个元素和累积值，然后将最后那次的函数应用的结果返回。如果第二个参数被省略，那么，数组列表的第一个元素将被用作初始值和累积值，并将内联函数应用到剩余的元素。

缺省情况下，列表数组的元素将被传递给自动声明的变量 x，而初始值或累积值将被传递给自动声明的 y。这两个变量也可被显示地声明为其他变量名。

```

a = { 1, 2, 3 }
b = fold( a, 10 ) -> { x + y } # produce 16
b = fold( a ) -> { x + y } # produce 6

```

## 2.3 unfold()

unfold()以一个初始值为参数，通过反复地将函数应用到该初始值或前次函数应用的结果，生成一个包含所有函数应用的列表。此函数应用将在某条件不成立时终止：

```
a = unfold( 5 ) -> |x| { while x > 0 return x - 1 }
# produce: {4, 3, 2, 1, 0 }
```

这里条件表达式必需放在关键字while之后，return之前（这里结果表达式之前必需有return关键字）。

也可有命令语句放在while之前：

```
a = unfold( 5 ) -> |x| {
  stdout.println(x)
  while x > 0 return x - 1
}
```

### 3 select, index, count

select, index, count 这几个方法可用来通过在一个或多个列表数组的元素上应用函数来测试某个条件是否满足，并选择，或索引，或计数那些满足条件的元素。

```
a = { 10, 20, 30 }
b = select( a ) -> { x >= 20 } # produce: {20, 30 }
c = index( a ) -> { x >= 20 } # produce: {1, 2 }
d = count( a ) -> { x >= 20 } # produce: 2
```

应用到多个列表也可实现 zip 功能：

```
a = { 10, 20, 30 }
b = { 'a', 'b', 'c' }
c = select( a, b ) -> { 1 }
# effectively as zip, produce: {( 10, a ), ( 20, b ), ( 30, c ) }
```

### 4 each, repeat

each() 跟 map() 用法一样，只不过不返回结果。

repeat() 以一个整数为参数，将大括号里的内联函数执行该参数所指定的次数。缺省情况下，函数执行的序号可通过自动声明的变量 i 传递给函数。

### 5 string, array, list

基于同样的语法，string(), array(), list() 也被支持为函数式方法，用来方便地构建复

杂的字符串，列表和数组。

这些方法都以一个整数为参数，将大括号里的内联函数 执行该参数所指定的次数，并将函数执行的结果连接为或组成字符串，列表或数组。缺省情况下，函数执行的序号可通过自动声明的变量 *i* 传递给函数。当然，该变量也可被声明为其他名称。

```
a = string( 5 ) -> { (string)i + 'a' }  
# produce: 0a1a2a3a4a  
  
b = list( 5 ) -> { (string)i + 'a' }  
# produce: {0a, 1a, 2a, 3a, 4a }  
  
c = array( 5 ) -> |x| { 10 * x }  
# produce: [ 0, 10, 20, 30, 40 ]
```

更复杂一点的例子:

```
d = list( 3 ) -> { { 1 : (i+2) } }  
# produce: {{1, 2 }, {1, 2, 3 }, {1, 2, 3, 4 }}  
  
e = array( 3 ) -> { [ 10*i+1, i+2 ] }  
# produce matrix: [ [1, 2], [11, 2], [21, 3] ]
```

## 6 sort, apply

### 6.1 sort()

`sort()` 根据内联的比较函数排序，尽量使得排序后的相邻元素满足比较函数所表达的关系。它以一个列表为第一个参数，和可选的整数为第二个参数。如果有第二个参数值为 *K*，列表将被排列直到前面 *K* 个元素被恰当地排好，如果比较函数表示的是简单的大于或小于关系，那么是最大的 *K* 个或最小的 *K* 个被恰当地排好。

```
a = { 3, 2, 15, 8, 25, 16, 3, 7, 1, 12, 17, 11, 0 };  
sort( a, 6 ) -> { x < y }
```

这里 *x,y* 被自动声明为了排序时需要比较的两个元素。它们也可被显示地声明为其他变量名。

## 6.2 apply()

`apply()` 将内联函数应用到数组的每个元素，并将元素的值替换为函数的结果。缺省情况下，表示数组元素的变量将被自动申明为 `x`，而元素的下标将被申明为 `i,j`。它们也可被显示地申明。要在内联函数里使用维数大于3的数组的下标，它们必须被显示地申明。

```
a = [ 10, 20, 30 ]
apply( a ) -> { x + i }
# a becomes: [ 10, 21, 32 ]

b = [ [ 1, 2; 3, 4 ] : 3 ];
apply( b ) -> |x,i,j,k| {
    stdout.println( x, i, j, k );
    return x + i*j*k;
}
```