5 Haskell 重要概念——函数

Haskell的编程方式被称为函数式编程，所以函数的概念在Haskell中占有非常重要的地位。

5.1 函数定义

- 最简单的函数定义

```
doubleMe x = x + x
ghci>doubleMe 3
6
ghci>1 + (doubleMe 3)
7
ghci>doubleMe 3.4
6.8
```

- 限制类型的函数定义

在上面的函数定义中，我们并没有给出函数的参数类型和返回类型，所以它可以用于整数，也可以用于浮点数， 如果我们想只用于浮点数，可以这样定义

```
doubleMeFloat :: Float -> Float
doubleMeFloat x = x + x
ghci>doubleMeFloat 2.4
4.8
ghci>doubleMeFloat 2
4.0
```

Float -> Float 的含义是接受一个Float类型(第一个Float)的参数，返回一个Float类型(第二个Float)的结果

- 模式匹配(Pattern Matching)和递归(Recrusion)方式定义

```
doubleList :: (Num a) => [a] -> [a]
doubleList [] = []
doubleList (x:xs) = (x+x):(doubleList xs)
ghci>doubleList [1,2,3]
[2,4,6]
ghci>doubleList [1.1,2.2,3.3]
[2.2,4.4,6.6]
```

上面的代码在函数类型定义时使用了类型参数a,这样可以使函数适用于多种类型，另外，由于函数定义中有(x+x)， 用了加法，所以a类型必须是Num的一个实例。

5.2 高阶函数(High-order functions)

5.2.1 Curried functions

严格意义上讲，Haskell里所有的函数都只有一个参数，这是另人迷惑的地方，因为之前我们已经用到了许多函数， 不只有一个参数。比如

```
ghci>:t max
max :: Ord a => a -> a -> a
ghci>max 3 4
4
```

我们再做一些实验来观察一些现象。 首先定义我们自己的max函数。

```
maxInteger :: Integer -> Integer -> Integer
maxInteger x y
  | x >= y = x
  | otherwise = y
ghci>:t maxInteger
maxInteger :: Integer -> Integer -> Integer
ghci>maxInteger 3 4
4
```

然后我们可以通过下面的方式得到另一个函数。

```
ghci>let maxInteger' = (maxInteger 3)
ghci>:t maxInteger'
maxInteger' :: Integer -> Integer
ghci>maxInteger' 4
4
ghci>maxInteger' 5
5
ghci>maxInteger' 3
3
ghci>maxInteger' 2
3
ghci>maxInteger' 1
3
```

我们只给了maxInteger一个参数3,把结果返回,然后得到了函数maxInteger'。 这个函数接收一个参数，函数的功能是把这个参数和3比较，然后输出其中较大的数。 这就是说maxInteger接收了一个整数，返回了一个函数。我们观察一下maxInteger的类型：

```
maxInteger :: Integer -> Integer -> Integer
```

发现maxInteger不仅可以解释成接收两个整数，返回一个整数。还可以解释成接收一个整数 返回一个函数，这个返回的函数类型为Integer -> Integer,即maxInteger'的类型。从实验上看第二种解释更为正确。这也说明了我们一开始提出的Haskell的函数都只有一个参数。 如果学习过lambda演算，对这一点一定不会陌生。 基于这一点，我们可以这样定义函数：

```
max3 = max 3
ghci>max3 1
3
ghci>max3 2
3
ghci>max3 3
3
ghci>max3 4
4
ghci>max3 5
5
```

这个函数在定义时连参数都没有显式地给出，但是在使用时却可以接收参数。

5.2.2 匿名函数

有时候我们需要一个函数，但是这个函数只是临时使用一下，我们甚至不需要给它起名字。 例如下面这个函数：

```
ghci>:t map
map :: (a -> b) -> [a] -> [b]
```

map接收一个函数，一个List,返回另一个List。这里接收的函数如果只是临时用一下，我们可以这样：

```
ghci>map (\x -> x + 1) [1,2,3]
[2,3,4]
```

这里的(\x -> x + 1)就是一个匿名函数。 另外，如果匿名函数有两个参数，还可以这样

```
ghci>(\x y -> x + y) 1 2
3
```

这种匿名函数的定义方式其实就是lambda演算里函数的定义方式。

5.2.3 以函数为参数的函数

- map

- 定义

将函数作用到List的每一个元素上，将每次所得结果放到另一个List,最后返回这个"结果List"

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

- 使用

```
ghci>map (+3) [1,2,3]
[4,5,6]
ghci>map (\x -> x+3) [1,2,3]
[4,5,6]
```

- filter
  - 定义

接收一个条件函数，一个List,将List中所有满足条件的元素提取出来。

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

- 使用

```
ghci>filter (>=3) [5,2,1,4,3,7]
[5,4,3,7]
```

- foldl

如果要想把一个List里面都有的元素都加起来，你会怎么做呢？用模式匹配和递归吗？foldl提供了一种解决方案.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

先测试一下，有一个直观的印象。

```
ghci>foldl (\acc x -> acc + x) 0 [1,2,3]
```

```
6
ghci>foldl (\acc x -> acc + x) 0 [1,2,3,4]
10
ghci>foldl (\acc x -> acc + x) 0 [5,8,4]
17
```

foldl接收一个函数，一个值，一个List,返回一个值。 其工作过程是这样的：首先第二个参数是acc的初值， 每次从第三个参数中提取一个元素， acc的值和这个元素作为第一个参数（是一个函数）的参数， 第一个参数（是一个函数）返回的结果 作为acc的新值参与后面的运算。一直到第三个参数为[]为止。 以第三个例子为例：acc 初值为 0, (\acc x -> acc + x) 0 5 得到结果5，作为acc的新值。 acc 新值为 5 (\acc x -> acc + x) 5 8 得到结果13,作为acc的新值。 acc 新值为 13 (\acc x -> acc + x) 13 4 得到结果17,作为acc的新值 acc 新值为17 List:[5,8,4]每个元素都已使用过，程序结束，结果为acc的最终值 17。

5.2.4 函数组合

数学里面其实就学过函数的组合，例如 f = x + 1 g = x * 3 令 h = f o g 这时候h 就是一个新函数，如果传递给h一个参数2,首先参数给g,2*3得到6,然后6作为f的参数，6+1=7。Haskell 里也提供了这种组合方式。

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
f' :: Integer -> Integer
f' x = x + 1
g' :: Integer -> Integer
g' x = x * 3
ghci>g' 2
6
ghci>f' 6
7
ghci>let h = f' . g'
ghci>h 2
7
```