Haskell基础

目录

- 使用函数
- List
- 高阶函数

什么是Haskell

- 一门纯粹的函数式编程语言，起源于1980s的数学研究。
- 根据科学家Haskell B.Curry的名字命名.
- 相比较命令(imperative)编程语言，函数编程语言关注于更高层次的"做什么"（What），而不是"怎么做"（How）。
- 函数编程语言的语法功能非常强，使编程的效率大幅提高。
- 优美、简洁。

函数语法

- 模式匹配
- Guards
- where
- let
- Case expressions

函数语法

- 模式匹配
- Guards
- where
- let
- Case expressions

模式匹配

- ```
  sum' :: (Num a) => [a] -> a

  sum' [] = 0

  sum' (x:xs) = x + sum' xs
  ```
- ```
  factorial :: (Integral a) => a -> a

  factorial 0 = 1

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

```
factorial 0 = 1

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

- factorial 3
- 3 * (factorial 2)
- 3 * (2 * (factorial 1))
- 3 * (2 * 1 * (factorial 0))
- 3 * 2 * 1 * 1
- 6

guards

- ```
  max' :: (Ord a) => a -> a -> a

  max' :: a b
  | a > b = a
  | otherwise = b
  ```
  ```

where

```
compare' :: (Num a) => a -> a -> a

compare' a b

  | diff > 0 = GT

  | diff = 0 = EQ

  | diff < 0 = LT

  where diff = a - b
```

Let

- let [bindings] in [expressions]
- 4 * (if 10 > 5 then 10 else 0) + 2 -> 42
- 4 * (let a = 9 in a + 1) + 2 -> 42
- [let square x = x * x in (square 5, square 3, square 2)]

  [(25,9,4)]
- (let (a,b,c) = (1,2,3) in a+b+c) * 100

  600
- let boot x y z = x * y + z in boot 3 4 2

  14

Case expressions

- 模式匹配本质上就是 case 语句的语法糖。
- case expression of pattern -> result

  pattern -> result

  pattern -> result

  ...
- sum' xs = case xs of

  [] -> 0

  [x:tail] -> x + sum' tail

List

- 最常用的数据结构，用于存储相同类型的多个元素。
- 字符串是一个Char类型的List
- Range
- List Comprehension

List常用操作

- ++

  [1,2,3] ++ [4,5,6] --> [1,2,3,4,5,6]

- :

  1:[2,3,4] -->[1,2,3,4]

- !!

  [1,2,3,4,5,6] !! 3 --> 4

- List of Lists

  [[1,2,3], [4,5,6], [7,8,9]]

- head

  head "hello" --> 'h'

- tail

  tail "hello" --> "ello"

- last

  last "hello" -- > 'o'

- init

  init "hello" -- > "ello"

- length

  length [1,2,3] -- > 3

- null

  null [] -- > True

- maximum

  maximum [1,2,3] -- > 3

- sum

  sum [1,2,3] -- > 6

- product

  product [1,2,3,4] -- > 24

- reverse

  reverse "hello" -- > "olleh"

- take

  take 3 [5,4,3,2,1] -- > [5,4,3]

- drop

  drop 3 [8,4,2,1,5,6] -- > [1,5,6]

- replicate

  replicate 3 10 , -- > [10,10,10]

- elem

  4 `elem` [3,4,5,6] -- > True

Range

- 构造List的方法之一，其中的值必须是可枚举的
- 惰性的
- [1 .. 10]

  [1,2,3,4,5,6,7,8,9,10]
- ['a' .. 'z']

  "abcdefghijklmnopqrstuvwxyz"
- ['K' .. 'Z']

  "KLMNOPQRSTUVWXYZ"

惰性

- [1..] 一个无穷自然数列
- take 10 [1..]

  [1,2,3,4,5,6,7,8,9,10]
- take 10 $ cycle [1,2,3]

  [1,2,3,1,2,3,1,2,3,1]
- take 10 (repeat 5)

  [5,5,5,5,5,5,5,5,5,5]

List Comprehension

- 来源于数学中的Set Comprehension，从一个集合产生另外一个集合
- 例如，前十个偶数的set comprehension可以表示为：

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$$

- take 10 [2,4..]

  [2,4,6,8,10,12,14,16,18,20]

- [x*2 | x <- [1..10]]

  [2,4,6,8,10,12,14,16,18,20]

Filtering

- `[ x | x <- [50..100], x `mod` 7 == 3]`

  `[52,59,66,73,80,87,94]`

- `[ x*y | x <-[2,5,10], y <- [8,10,11], x*y > 50]`

  `[55,80,100,110]`

- `removeNonUppercase st =`

  `[ c | c <- st, c `elem` ['A'..'Z'] ]`

- `removeNonUppercase "Hahaha! Ahahaha!"`

  `"HA"`

Tuple

- (x,y,z)
- 一组元素的组合，元素类型可以不同
- 长度大于1，且不可变
  - List of tuples，用来表示一组相关的元素，例如，表示二维平面上的一组坐标

  [(1,2) , (8,11), (3,5)]
- Pair(含有两个元素的tuple)的常用函数
  - fst
  - snd

zip

- 取两个 List，然后将它们交叉配对，形成一组pair的 List
- zip [1,2,3,4,5] [5,5,5,5,5]

  [(1,5),(2,5),(3,5),(4,5),(5,5)]
- zip [1 .. 5]

  ["one", "two", "three", "four", "five"]

[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]

一个示例

- 如何取得所有三边长度皆为整数且小于等于 10，周长为 24 的直角三角形?
- ```
  > let rightTriangles' =
      [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b],
                  a^2 + b^2 == c^2, a+b+c == 24 ]
  ```
- ```
  > rightTriangles'
  [(6,8,10)]
  ```

高阶函数

- 函数可以作为参数和返回值。
- 本质上，haskell的所有函数都只有一个参数，所有多个参数的函数都是柯里函数。
- 这两者等价：
  - max 3 4
  - (max 3) 4
- 利用不全调用，可以构造新的函数，并作为参数传递给其它函数。

map

- 定义:

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

- map (+3) [1,5,3,1,6]

  [4,8,6,4,9]

- 可以与List Comprehension互换
  - map (+3) [1,5,3,1,6] 与 [x+3 | x <- [1,5,3,1,6]

filter

- 定义

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

- filter (>3) [1,5,3,2,1,6,4,3,2,1]

  [5,6,4]

- List Comprehension:

  [x | x<-[1,5,3,2,1,6,4,3,2,1], x>3]

- filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"

  "uagameasadifeent"

- sum (takeWhile (<10000) (filter odd (map (^2) [1..])))

  166650

- sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])

  166650

fold

- 一个 fold 取一个二元函数，一个初始值(我喜欢管它叫累加值)和一个需要折叠的 List；有foldl、 foldr两种方式。

- sum' :: (Num a) => [a] -> a

  sum' xs = foldl (\acc x -> acc + x) 0 xs

- map' :: (a -> b) -> [a] -> [b]

  map' f xs = foldr (\x acc -> f x : acc) [] xs

Lambda

- 匿名函数，可以作为参数传给高阶函数。
- `map (\x -> x+3) [1,6,3,2]`
- `addThree :: (Num a) => a -> a -> a -> a`，下面两个等价：
  - `addThree x y z = x + y + z`
  - `addThree = \x -> \y -> \z -> x + y + z`
- 由于有了Curried functions，大部分匿名函数都可以替换掉。

  `map (+3) [1,6,3,2]`

$

- 函数调用符
- 定义:

```
($) :: (a -> b) -> a -> b

f $ x = f x
```

- 特点：右结合，优先级最低
- 减少代码中括号的数目

- f (g (z x)) 与 f $ g $ z x 等价
- f a b c 与 ((f a) b) c 等价
- sum (map sqrt [1..10]) ，等价于

  sum $ map sqrt [1..10]
- sqrt 3 + 4 + 9 vs. sqrt $ 3 + 4 + 9
- sum (filter (> 10) (map (*2) [2..10]))

  sum $ filter (> 10) $ map (*2) [2..10]

函数组合

- `(.) :: (b -> c) -> (a -> b) -> a -> c`

  `f . g = \x -> f (g x)`

- 当一个函数的返回值的类型与另一个函数输入值的类型相同时，这两个函数就可以复合。

- 复合运算满足结合律：

  `f . (g . h) = (f . g) . h`

- 将一组函数组合起来，形成更加复杂的函数。

- 函数组合是右结合的，表达式 `f (g (z x))` 与 `(f . g . z) x` 等价

- map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]

  [-5,-3,-6,-7,-3,-2,-19,-24]

- map (negate . abs) [5,-3,-6,7,-3,2,-19,24]

  [-5,-3,-6,-7,-3,-2,-19,-24]

- map (\xs -> negate (sum (tail xs)))

  [[1..5], [3..6], [1..7]]

  [-14,-15,-27]

- map (negate . sum . tail) [[1..5], [3..6], [1..7]]

  [-14,-15,-27]

- 多个参数情况下使用函数组合

```
sum (replicate 5 (max 6.7 8.9))

(sum . replicate 5 . max 6.7) 8.9

sum . replicate 5 . max 6.7 $ 8.9
```

- point free style

```
sum' :: (Num a) => [a] -> a

sum' xs = foldl (+) 0 xs

改写成: sum' = foldl (+) 0
```

- fn x = ceiling (negate (tan (cos (max 50 x)))) 等价于

```
fn = ceiling . negate . tan . cos . max 50
```