第5章

# 高阶函数

高阶函数是函数程序设计最重要的特性之一。所谓高阶函数就是函数的输入或者输出也是函数的函数，也可以说高阶函数表达了各种计算模式。高阶函数使得许多函数表达更简洁更灵活，而且可以大大减少了编程工作量。本章介绍常见的高阶函数以及高阶函数的应用。

## 5.1 函数也是数据

在函数语言中，函数是“一等公民”（first class citizen），即函数可以像其他数据一样出现在表达式中，作为其他函数的输入参数，或者作为返回值。

如果一个函数的输入参数也是函数，或者函数的结果也是函数，则称这样的函数为**高阶函数**（higher order function）。

### 5.1.1 map 计算模式

人们常常对一个列表的元素逐个进行某种运算，例如将一个数值列表的每个元素加倍，用列表概括表示为


高阶函数

```
Prelude> [2 * x | x <- [1..5]]
[2,4,6,8,10]
```

再例如，对一个字符串列表中每个元素求长度，得到这些串的长度列表：

```
Prelude> [length s | s <- ["Haskell", "is", "a", "functional",
"language"]] [7,2,1,10,8]
```

这些计算的共同特点是对列表的每个元素进行某种运算。我们将“某种运算”抽取出来，作为这种有共同特点计算模式的输入，由此得到一个有两个输入的计算模式：第一个输入是某个运算 f，第二个输入是一个列表 xs，其中第一个运算 f 是可以应用于第二个列表 xs 的每个元素的计算：

```
[f x | x <- xs]
```

把这种计算模式称为 map，那么 map 的类型以及定义为

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

**注 1** 函数 map 的类型可以这样来推导：假设第二个输入参数 xs :: [a]，因为第一个输入参数 f 具有函数类型，而且可应用于 xs 中每个元素，因此有 f :: a -> b，最后结果为形如 f x 的元素构成（其中 x :: a），因此结果类型为 [b]。

计算模式 map 可以用列表概括定义，也可以用递归定义：

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

我们将这种运算模式称为列表上的**映射**(map) 运算。

每当遇到这种映射运算时，只需要给 map 函数提供相应的函数参数即可。例如，前面用列表概括表示的计算 [2 * x | x <- [1..5]] 是一种特殊的映射运算，可以表示为 map double [1..5]，此时 map 的类型为 (Int -> Int) -> [Int] -> [Int]，其中 double 是一个函数：

```
double :: Int -> Int
double x = 2 * x
```

同样，计算 [length s | s <- ["Haskell", "is", "a", "functional", "language"]] 也可以写成 map length ["Haskell", "is", "a", "functional", "language"]，此时 map 的类型为 (String -> Int) -> [String] -> [Int]。

再例如，定义判断奇偶性的函数 even：

```
even :: Int -> Bool
even x = mod x 2 == 0
```

则可将 even 映射到一个整数列表：

```
*Main> map even [2,4,5,7]
[True, True, False, False]
```

这里 map 具有类型 (Int -> Bool) -> [Int] -> [Bool]。

**注 2** 函数 map 的第一个参数是一个函数，第二个参数是列表，函数和其他数据具有相等的地位，都可以作为其他函数的输入，因此称函数也是数据，函数是“一等公民”。但是，在多数命令式程序设计语言中，函数并非“一等公民”。

### 5.1.2 λ 表达式

λ 表达式是一种表示函数的方法[①]。例如，函数 double 定义如下：

① λ 表达式来自 λ 演算，它是一种计算模型，也是 Haskell 语言的理论基础。

```
double :: Int -> Int
double x = 2 * x
```

函数定义表示：对于任意整数 x，其对应的元素是 2 * x，并将该函数命名为 double。这个函数也可以用一个 λ 表达式表示：\x -> 2 * x，或者说这个 λ 表达式是该函数的匿名表示[①]。例如，在解释器下可以直接用 λ 表达式表示一个函数，将其应用于合法的输入：

```
Prelude> (\x -> 2 * x) 5
10
```

当然，也可以给这个 λ 表达式起名，如double1 = \x -> 2 * x，double1 与前面的 double 表示了同一个函数：

```
Prelude> double1 = \x -> 2*x
Prelude> double1 5
10
```

一般来说，如果一个函数 f 的定义形如

```
f :: a -> b
f x = e
```

那么这个函数 f 也可以写成f = \x -> e。

同样，多个输入的函数也可以用 λ 表达式表示。例如，\x y -> x + y 表示了一个将两个数值相加的函数，可以将其应用于两个数值求和：

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

在使用 map 时，也可以直接用 λ 表达式表示 map 的第一个参数。例如，以上运算中的 even 可以直接用\x -> mod x 2 == 0 表示：

```
Prelude> map (\x -> mod x 2 == 0) [2,4,5,7]
[True, True, False, False]
```

再例如，将二元组列表转换为两个分量之和的列表，也是一种映射运算：

```
Prelude> map (\ (x, y) -> x + y) [(1,2),(1,4),(1,5),(1,7)]
[3,5,6,8]
```

注意，上面 λ 表达式表示的函数输入是二元组，因此 λ 表达式的参数直接使用了二元组模式。也可以将该函数表达为\xy -> fst xy + snd xy。

① 在 λ 演算中，double = λx.2x。


常用高阶函数

## 5.2 常用高阶函数

本节介绍一些常见的计算模式以及相应的高阶函数。

### 5.2.1 折叠计算模式 foldr

首先观察两个函数 sum 和 product，对于一个数值列表 `xs`，`sum xs` 返回列表 `xs` 元素的累加和，`product xs` 返回 xs 元素连乘之积。两个函数的定义如下：

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs

product :: Num a => [a] -> a
product [] = 1
product (x:xs) = x + product xs
```

可以发现这些计算的共同特点：将空列表对应到某个值，将非空列表的元素用某种运算连起来。例如，按照定义将 `sum [1,2,3]` 展开来，相当于 `1 + (2 + (3 + 0))`，`product [1,2,3]` 相当于 `1 * (2 * (3 * 1))`。我们也可以将这种具有共同特点的计算抽象成一个高阶函数 foldr，将对应于空列表的“某个值”和将列表元素连接起来的“某个二元运算”作为高阶函数 foldr 的两个输入参数，在不同的计算中可以取不同的值，由此得到高阶函数 foldr 的定义：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op s []     = s
foldr op s (x:xs) = op x (foldr op s xs)
```

其中，s 是空列表对应的“某个值”，op 是“某个二元运算”。

**注 3** 函数 foldr 的类型可以这样推导：假设第二个参数 s 具有类型 `s :: b`，那么这也是结果的类型，因为在最简单输入情况下函数的结果是 s。假定第三个输入参数具有类型 `[a]`，那么第一个输入参数 op 是具有两个输入参数的函数类型，并能应用于类型 a 的元素和类型 b 的元素，结果类型同 foldr 的最后结果类型 b 相同，因此有 `op :: a -> b -> b`。

有了 foldr 的定义，sum 和 product 都是 foldr 的特殊情况。例如，`sum` 相当于 `foldr (+) 0`，`product` 相当于 `foldr (*) 1`：

```
Prelude> foldr (+) 0 [1,2,3,4,5]
15
Prelude> foldr (*) 1 [1,2,3,4,5]
120
```

一个需要两个参数的函数 f 可以使用反引号转换为中缀运算，例如 `div 4 2` 等价于 `4 ‘div‘ 2`。

如果在以上定义的第二个等式中将 foldr 的第一个参数使用中缀表示，则有

```
foldr op s (x:xs) = x `op` (foldr op s xs)
```

分析 foldr 在一个列表例子的计算过程可以看出它所代表的计算模式：

```
foldr op s (x:(y:(z:[])))
  = x `op` (y `op` (z `op` (foldr op s [])))
  = x `op` (y `op` (z `op` s))
```

上述计算模式表明，列表中的 (:) 被运算 `op`代替，空列表被 s 代替。

### 5.2.2　过滤计算模式 filter

一种常见的计算模式是在列表中选出满足某种性质的元素。例如，在一个整数列表中选出其中的偶数：

```
Prelude> [x | x<- [1..5], even x]
[2,4]
```

其中，even :: Int -> Bool 是判断一个整数是否偶数的函数。

再例如，在一个串中选出其中的数字：

```
Prelude> [x | x <- "born on 15 April 1986", Data.Char.isDigit x]
"151986"
```

其中，isDigit :: Char -> Bool 是模块 Data.Char 定义的函数。

**注 4**　*在解释器中，可以直接使用“模块名. 函数名”的方式调用一个函数。*

以上两种计算均具有形式 [x | x <- xs, p x]，其中，p 是某种性质。这种在一个列表 xs 上选出满足某种性质 p 的元素的计算模式称为**过滤**，用 filter 表示，其一般类型及定义如下：

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

函数 filter 的第一个输入是类型 a -> Bool 的函数，因此，它表达了类型 a 的元素是否具有某种性质。

函数 filter 的递归定义留作习题。

### 5.2.3　前缀处理函数 takeWhile 和 dropWhile

函数 take 可以取得列表给定长度的前缀。有时需要在列表中取得满足某种条件的前缀子列表，即从列表的第一个元素开始，如果满足指定条件则保留，直至遇到不满足条件的元素，然后返回保留下来的满足条件的前缀子列表。例如，在字符串"Haskell Curry" 中取出第一个词，可以认为是“保留每个字母，直至遇到非字母符号”。这里的条件是“元素是字母”，它是一个类型为 Char -> Bool 的函数。

**注 5** 判断某种类型 a 的数据是否满足某种条件的函数具有类型 a -> Bool，或者说这种类型的函数表达了类型 a 的数据的某种性质。例如，一个字符是否大写的性质具有类型 Char -> Bool，一个整数是否素数的性质具有类型 Int -> Bool。

一般地，这种计算模式需要两个输入：一个是表示条件的函数；另一个是列表，其结果也是列表。这种函数在标准库中称为 takeWhile，其类型和定义如下：

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

因此，如果将在一个串中提取第一单词的函数称为 getWord，则可以用 takeWhile 定义之：

```
getWord :: String -> String
getWord s = takeWhile isAlpha s
```

其中，isAlpha 是模块 Data.Char 提供的一个函数。

类似于 drop，我们也可以按照某个条件，舍弃满足条件的元素，直至遇到不满足条件的元素，返回剩下的尾列表。这种函数在标准库中称为 dropWhile，其类型和定义如下：

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

例如，如果要在“Haskell Curry”中取得第二个单词，可以先舍弃第一个单词，利用 dropWhile isAlpha "Haskell Curry"，得到串" Curry"，然后再用 getWord 取得第二个单词。不过，要注意，直接使用 getWord " Curry" 并不能得到单词“Curry”，而是得到一个空串（为什么？）。为此，要得到第二个单词还需要再次舍弃前面的非字母字符，例如 dropWhile isSpace " Curry"（isSpace 是模块 Data.Char 定义的函数），然后进一步使用 getWord 得到单词“Curry”。

### 5.2.4 函数的复合

在数学上，如果有两个函数 $f : A \to C$ 和 $g : B \to C$，那么可以定义这两个函数的复合 $g \circ f$，它是 $A$ 到 $C$ 的函数，而且 $g \circ f(x) = g(f(x))$。在 Haskell 中，这个**函数复合**运算用点 (.) 表示，其类型为

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

例如，定义两个函数：

```
f :: Int -> Int
f x = 2 * x
g :: Int -> Int
g x = x + 1
```

那么 g . f 具有类型 Int -> Int，而且 (g . f) x 的结果是 g(f x) 或者 g (2 * x)，即 2 * x + 1，也就是复合函数 (g . f) 应用于 x 的结果是 2*x + 1。

函数的复合实际上是常见的运算。例如，在 5.2.3 节定义的函数 getWord，提取一个串中的第一个单词：getWord "Haskell Curry" 的结果是"Haskell"，但是，如果输入串前面有非字母字符，则会得到空串：getWord " Curry" 为空串。这是因为 getWord 的定义实际上假定输入串的第一个词前面不含空格等非字母字符。为此，为了确保在各种情况下都能取得第一个单词，可以先使用 dropWhile 将前面的非字母字符舍弃，然后开始取第一个单词。由此得到如下第二个版本定义：

```
getWord2 :: String -> String
getWord2 s = getWord (dropSpaces s)
```

其中，dropSpace 舍弃了非字母字符，定义为

```
dropSpaces :: String -> String
dropSpaces s = dropWhile (\x -> not (isApha x)) s
```

由此可见，getWord2 是 dropSpaces 与 getWord 的复合，因此，也可直接用复函运算定义 getWords2：

```
getWord2 = getWord . dropSpaces
```

在函数 dropSpaces 定义中，函数参数\x -> not (isApha x) 也包含了复合运算，可以写成\x -> (not . isApha) x，这个 λ 表达式表示的函数实际上就是函数 isAlpha 与 not 的复合，因此，dropSpaces 可以定义为 dropWhile (not . isApha)：

```
dropSpaces = dropWhile (not . isApha)
```

**注 6**　如果一个函数 f 定义形如：

```
f :: a -> b
f x = g x
```

其中，g 是与 f 同类型的一个函数，则称 f 和 g 是两个相等的函数，并可用如下形式定义：

```
f :: a -> b
f = g
```

称这种函数的定义方式为 f 的**无参数表示**。

实际上，函数复合是人们解决问题常用的方法。例如，在定义一个函数 f :: a -> b 完成某个计算时，往往将其分解成多步计算，例如，第一步（step1）将原输入类型转换为 b1 类型的数据，第二步（step2）将第一步的结果转换为 b2 类型的数据，第三步（step3）将第二步的结果转换为 b 类型的结果，也是最后希望得到的结果。因此，函数 f 便是这三个函数的复合：

```
f :: a -> b
step1 :: a -> b1
step2 :: b1 -> b2
step3 :: b2 -> b
f x = step3 (step2 (step1 x))
```

也可以直接写成 f = step3 . (step2 . step1)。

函数复合运算是右结合的,因此,上述定义也可写成 f = step3. step2. step1。

### 5.2.5 卡瑞化

Haskell 中习惯将数学上的二元函数 $f : A \times B \to C$ 写成 f :: A -> B ->C。这种将一个以二元组为输入的函数转换为依次取两个输入的二元函数称为**卡瑞化**（Curried）[①]：

```
curry :: ((a, b) ->c) -> a -> b -> c
curry f x y = f (x,y)
```

例如，定义二元函数 add：

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

那么 curry add 具有类型 Int -> Int ->Int，仍然表示将两个输入相加：

```
*Main> (curry add) 1 2
3
```

相反，将类型为 a -> b -> c 的函数转换为类型 (a, b) -> c 的函数称为**去卡瑞化**（uncurry）：

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

例如，uncurry (\x y -> x + y) 相当于函数 add：

```
*Main> (uncurry (\x y -> x + y)) (1, 2)
3
```

① 为纪念著名数学家和逻辑学家 Haskell Curry 命名，又译柯里化。

### 5.2.6　部分应用

在 Haskell 中可以将一个二元运算（函数）应用于一个输入。例如，(+) 是一个二元运算，通常需要将其应用于两个同类型数值。但是，也可以只为 (+) 提供一个输入，如 (+1)，其结果是类型 `Int -> Int` 的函数，因此，(+1) 可以进一步应用于另一个数值，结果就是给后一个数值加一：

```
Prelude> (+1) 2
3
```

因此，(+1) 相当于“加 1”函数：`\x -> x + 1`。

同样 (1+) 也是类型 `Int -> Int` 的函数：

```
Prelude> (1+) 2
3
```

一般地，如果 $\oplus$ 是一个二元运算符，那么给它提供左运算数 $a$ 后，$a\oplus$ 就变成一个一元运算，这个一元运算可以进一步应用于一个值 $b$，结果就是 $a \oplus b$。

这种将一个多元函数应用于部分输入的现象叫**部分应用**(section)。部分应用也构成一种表示函数的方法。例如，(1+) 就是加一函数。再例如，

```
sum xs = foldr (+) 0 xs
```

也可以只给 foldr 提供前两个输入：`foldr (+) 0`，它表示函数 sum，是 sum 的无参数表示。

一般地，一个多元函数可以应用于它的前几个输入，形成部分应用。例如，假设 `f :: Int -> Bool -> Int -> Bool`，那么 f 可应用于第一个输入类型的值，如 `f 3`，它是类型为 `Bool -> Int -> Bool` 的函数。同样 f 3 可以应用于一个类型 Bool 的值，如 f 3 True，它是一个类型为 `Int -> Bool` 的函数。

在 Haskell 中，函数通常用卡瑞化的形式定义，这为使用部分应用表达函数提供了方便。


词频统计

## 5.3　词频统计

通常在设计完成一个计算任务的方法时，将计算任务分成多步进行，每一步的输出是下一步的输入，每一步设计一个函数完成，最后这些函数的复合构成最终解。本节以词频统计为例，说明如何使用任务分解、步骤复合和高阶函数解决问题。

### 5.3.1　问题分析及解决步骤

给定一个多行文本串，例如`"hello clouds\n hello sky\n"`，要求统计所有单词及其出现次数，并按照单词字典序排列打印在屏幕上。例如，

```
clouds:1
hello:2
sky:1
```

对于一个计算任务，首先分析输入数据类型和输出数据类型。这个问题的输入是字符串，那么输出是打印在屏幕上的词频统计数据。实际上，这个计算任务的关键是统计所有单词出现的次数，将这些数据打印在屏幕上可以仿照 3.3.3 节的打印清单函数完成。因此，首先考虑输出数据是所有单词及其出现次数。对于前面的输入例子，期望的输出形如

```
[("clouds",1),("hello",2),("sky",1)],
```

因此核心计算任务的类型为 `String -> [(Sting, Int)]`。

以输入`"hello clouds\n hello sky\n"` 和输出 `[("clouds",1),("hello",2),("sky",1)]` 为例，将计算任务分成下列步骤。

（1）提取输入串的所有单词，结果是所有单词的列表：

```
["hello","clouds","hello","sky"]
```

为此，设计一个类型 `String -> [String]` 的函数。

（2）将单词列表排序，这样相同的单词应该连续排列，由此得到所有单词的有序列表，如 `["clouds","hello","hello","sky"]`。为此，需要一个类型为 `[String] -> [String]` 的排序函数。

（3）将有序单词列表中相同的单词组织成一个列表，由此得到一个新的列表，其中每个元素是相同单词构成的列表：

```
[["clouds"],["hello","hello"],["sky"]]
```

为此，需要一个类型为 `[String] ->[[String]]` 的函数。

（4）将上一步结果列表中每个元素（同一个单词构成的列表）转换为该单词及其出现次数的二元组，如 `[("clouds",1),("hello",2),("sky",1)]`。这一步是一个 map 计算模式，可以用 map 完成。

以上步骤完成了核心计算任务。接下来为每一步设计相应的函数。

### 5.3.2 设计分步函数

在设计完成一个计算任务时，首先考虑使用已有的库函数，其次考虑自定义函数。事实上，每一种程序设计语言都提供了许多完成不同类型任务的库函数。特别是，对于常见的计算任务，都存在预定义的库函数。

对于 5.3.1 节第（1）步，在 hoogle 查找发现存在多个类型 `String -> [String]` 的函数，其中 words 是解决该问题的预定义函数。