# 第 08 章：Higher-order Function

```
Higher-order Function  =>  高阶函数
```

✧ Higher-order Function

A function is called **higher-order**, if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x  =  f (f x)
```

- **twice** is higher-order, because it takes a function as its first argument.
  - Obviously, **twice** also returns a function after taking the first argument.

✧ Why Higher-order Function

1. Common programming idioms can be encoded as functions within the language itself.
2. Domain specific languages can be defined as collections of higher-order functions.
3. Algebraic properties of higher-order functions can be used to reason about programs.

✧ The map Function

The higher-order library function called map applies a function to every element of a list.

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

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

The map function can be defined in a particularly simple manner using a list comprehension:

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

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

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

✧ The `filter` Function

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
ghci> filter even [1..10]
[2,4,6,8,10]
```

The filter function can be defined using a list comprehension:

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

Alternatively, it can be defined using recursion:

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

✧ The `foldr` Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []  =  v
f (x : xs)  =  x  ⊕  f xs
```

- f maps the empty list to some value v, and any non-empty list to some function (⊕) applied to its head and f of its tail.

For example:

```
sum []  =  0
sum (x:xs)  =  x + sum xs

product []  =  1
product (x:xs)  =  x * product xs

and []     = True
and (x:xs) = x && and xs
```

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum     = foldr (+) 0
product = foldr (*) 1
or      = foldr (||) False
and     = foldr (&&) True
```

The foldr function defined in Haskell:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
```

- Right-associative fold of a structure, lazy in the accumulator
- In the case of lists, foldr, when applied to a binary operator, a starting value, and a list, reduces the list using the binary operator, from right to left

  `foldr f z [x1, x2, ..., xn]  ===  x1 `f` (x2 `f` ... (xn `f` z) ...)`
- Note that since the head of the resulting expression is produced by an application of the operator to the first element of the list, given an operator lazy in its right argument,

- foldr can produce a terminating expression from an unbounded list.

The foldr on lists can be defined using recursion:

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

- However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

**Example**: The foldr on lists

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

```
    sum [1, 2, 3]
=== foldr (+) 0 [1, 2, 3]
=== foldr (+) 0 (1 : (2 : (3 : [])))
===              (1 + (2 + (3 + 0 )))
=== 6
```

```
product  =  foldr (*) 1
```

```
    product [1, 2, 3]
=== foldr (*) 1 [1, 2, 3]
=== foldr (*) 1 (1 : (2 : (3 : [])))
===              (1 * (2 * (3 * 1 )))
=== 6
```

```
length :: [a] -> Int
length []  =  0
length (_:xs)  =  1 + length xs
```

```
    length [1, 2, 3]
=== length (1 : (2 : (3 : [])))
===        (1 + (1 + (1 + 0 )))
```

```
=== 3
```

```
length :: [a] -> Int
length =  foldr (⊕) 0 where
    _ ⊕ n = 1 + n
```

```
    length [1, 2, 3]
=== foldr (⊕) 0 (1 : (2 : (3 : [])))
===             (1 ⊕ (2 ⊕ (3 ⊕ 0 )))
===             (1 + (1 + (1 + 0 )))
=== 3
```

```
reverse :: [a] -> [a]
reverse []  =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

```
    reverse [1, 2, 3]
=== reverse (1 : (2 : (3 : [])))
=== (([] ++ [3]) ++ [2]) ++ [1]
=== [3, 2, 1]
```

```
reverse :: [a] -> [a]
reverse  =  foldr (⊕) [] where
    x ⊕ xs  =  xs ++ [x]
```

```
    reverse [1, 2, 3]
=== foldr (⊕) [] (1 : (2 : (3 : [])))
===              (1 ⊕ (2 ⊕ (3 ⊕ [])))
=== (([] ++ [3]) ++ [2]) ++ [1]
=== [3, 2, 1]
```

Finally, we note that the append function (++) has a particularly compact definition using foldr:

```
(++) :: [a] -> [a] -> [a]
(++ ys) = foldr (:) ys
```

○ 遗憾的是：Haskell 似乎不支持这种定义方式

```
error: Parse error in pattern: ++ys
```

- 下面是两种可以通过编译的定义方式

```
(++) :: [a] -> [a] -> [a]
(++) xs ys = foldr (:) ys xs

(++) :: [a] -> [a] -> [a]
(++) = flip $ foldr (:)

-- flip 是 Prelude 导出的一个函数，其定义如下：
flip  ::  (a -> b -> c) -> b -> a -> c
flip f x y  =  f y x
```

Why foldr ?

- Some recursive functions on lists, such as sum, are simpler to define using foldr.
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule.
- Advanced program optimizations can be simpler if foldr is used in place of explicit recursion.

✧ The foldl Function on Lists

It is also possible to define recursive functions on lists using an operator that is assumed to *associate to the left*.

```
f v []     = v
f v (x:xs) = f (v ⊕ x) xs
```

- f maps
  - the empty list to the accumulator value v, and
  - any non-empty list to the result of recursively processing the tail using a new accumulator value obtained by applying an operator ⊕ to the current value and the head of the list.

foldl on lists itself can be defined using recursion:

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

The foldr function defined in Haskell:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

- Left-associative fold of a structure, lazy in the accumulator
- In the case of lists, foldl, when applied to a binary operator, a starting value, and a list, reduces the list using the binary operator, from left to right:

  `foldl f z [x1, x2, ..., xn]  ===  (...((z `f` x1) `f` x2) `f`...) `f` xn`

- Note that to produce the outermost application of the operator the entire input list must be traversed; as a result, foldl will diverge if given an infinite list.
- If you want an efficient strict left-fold, you probably want to use `foldl'`.

✧ Some other higher-order library functions

1. The library function (.) returns the composition of two functions as a single function.

   ```
   (.) :: (b -> c) -> (a -> b) -> a -> c
   (.) f g = \x -> f $ g x
   ```

   - For example

   ```
   odd :: Int -> Bool
   odd = not . even
   ```

2. The all function determines whether all elements of the structure satisfy the predicate.

   ```
   all :: Foldable t => (a -> Bool) -> t a -> Bool
   ```

   - On lists, all can be defined as:

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

3. The any function determines whether any element of the structure satisfy the predicate.

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
```

   ○ On lists, any can be defined as:

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

4. The function takeWhile selects elements from a list while a predicate holds of all the elements

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

```
ghci> takeWhile (/= ' ') "abc def"
 "abc"
```

5. Dually, the function dropWhile removes elements while a predicate holds of all the elements.

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

✧ 应用 1：Binary String Transmitter

**2 进制数 转换到 10 进制数**

○ 效果：

```
ghci> bin2int [1, 0, 1, 1]
13
```

- 定义方式一

```
type Bit = Int

bin2int :: [Bit] -> Int
bin2int bits  =  sum [ w * b | (w, b) <- zip weights bits ]
   where weights = iterate (* 2) 1

-- iterate is defined in Prelude
-- iterate :: (a -> a) -> a -> [a]
-- iterate f x  =  x : iterate f (f x)
```

- 定义方式二

```
type Bit = Int

bin2int :: [Bit] -> Int
bin2int  =  foldr (\x y -> x + 2 * y) 0
```

**10 进制数 转换到 8 位 2 进制数**

- 效果：

```
ghci> int2bin8 13
[1, 0, 1, 1, 0, 0, 0, 0]
```

```
int2bin :: Int -> [Bit]
int2bin 0  =  []
int2bin n  =  mod n 2 : int2bin (div n 2)

make8 :: [Bit] -> [Bit]
make8 bits  =  take 8 $ bits ++ repeat 0
-- repeat is defined in Prelude
-- repeat :: a -> [a]
-- repeat x = xs where xs = x : xs

int2bin8 :: Int -> [Bit]
int2bin8  =  make8 . int2bin
```

**文字序列编码**

- 效果：

```
ghci> encode "abc"
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2bin . ord)
```

**2 进制序列解码**

- 效果：

```
ghci> decode [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

```
decode :: [Bit] -> String
decode = map (chr . bin2int) . chop8

chop8 :: [Bit] -> [[Bit]]
chop8 []   = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

✧ 应用 2.1：投票算法 之 First past the post

In this system, each person has one vote, and the candidate with the largest number of votes is declared the winner.

```
votes :: [String]
votes =  ["Red", "Blue", "Green", "Blue", "Blue", "Red"]
```

```
ghci> result votes
[(1,”Green"),(2,"Red"),(3,"Blue")]

ghci> :type result
result :: Ord a => [a] -> [(Int, a)]

ghci> winner votes
"Blue"

ghci> :type winner
result :: Ord a => [a] -> a
```

```
result :: Ord a => [a] -> [(Int, a)]
result vs  =  sort [ (count v vs, v) | v <- rmdups vs ]
-- The sort function is defined in Data.List
```

```
rmdups :: Eq a => [a] -> [a]
rmdups []  =  []
rmdups (x:xs)  =  x : filter (/= x) (rmdups xs)

count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)

winner :: Ord a => [a] -> a
winner = snd . last . result
```

✧ 应用 2.2：投票算法 之 Alternative vote

In this voting system, each person can vote for as many or as few candidates as they wish, listing them in preference order on their ballot (1st choice, 2nd choice, and so on).

```
ballots :: [[String]]
ballots  =  [["Red", "Green"],
             ["Blue"],
             ["Green", "Red", "Blue"],
             ["Blue", "Green", "Red"],
             ["Green"]]
```

```
ghci> winner ballots
"Green"
ghci> :type winner
winner :: Ord a => [[a]] -> a
```

To decide the winner:

1. any empty ballots are first removed,
2. then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
3. and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots  =  [["Red", "Green"],
             ["Blue"],
             ["Green", "Red", "Blue"],
             ["Blue", "Green", "Red"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Red", "Green"],
             ["Blue"],
             ["Green", "Red", "Blue"],
             ["Blue", "Green", "Red"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Green"],
             ["Blue"],
             ["Green", "Blue"],
             ["Blue", "Green"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Green"],
             ["Blue"],
             ["Green", "Blue"],
             ["Blue", "Green"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Green"],
             [],
             ["Green"],
             ["Green"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Green"],
             [],
             ["Green"],
             ["Green"],
             ["Green"]]
```

```
ballots :: [[String]]
ballots  =  [["Green"],
             ["Green"],
             ["Green"],
             ["Green"]]
```

- 定义方式

```
winner :: Ord a => [[a]] -> a
winner bs = case rank $ filter (/= []) bs of
    [c] -> c
    (c:cs) -> winner $ map (filter (/= c)) bs

rank :: Ord a => [[a]] -> [a]
rank = map snd . result . map head
```

**作业 01**

Express the list comprehension `[f x | x <- xs, p x]` using the functions `map` and `filter`.

**作业 02**

Redefine `map f` and `filter p` using `foldr`.

作业 **03**

Modify the binary string transmitter example to detect simple transmission errors using the concept of parity bits.

- That is, each eight-bit binary number produced during encoding is extended with a parity bit,
  - set the parity bit to one if the number contains an odd number of ones, and to zero otherwise.
- In turn, each resulting nine-bit binary number consumed during decoding is checked to ensure that its parity bit is correct, with the parity bit being discarded if this is the case, and a parity error being reported otherwise.

Hint: the library function `error :: String -> a` displays the given string as an error message and terminates the program; the polymorphic result type ensures that error can be used in any context.