

第四章：函数式编程

使用 Haskell 思考

初学 Haskell 的人需要迈过两个难关：

首先，我们需要将自己的编程观念从命令式语言转换到函数式语言上面来。这样做的原因并不是因为命令式语言不好，而是因为比起命令式语言，函数式语言更胜一筹。

其次，我们需要熟悉 Haskell 的标准库。和其他语言一样，函数库可以像杠杆那样，极大地提升我们解决问题的能力。因为 Haskell 是一门层次非常高的语言，而 Haskell 的标准库也趋向于处理高层次的抽象，因此对 Haskell 标准库的学习也稍微更难一些，但这些努力最终都会物有所值。

这一章会介绍一些常用的函数式编程技术，以及一些 Haskell 特性。还会在命令式语言和函数式语言之间进行对比，帮助读者了解它们之间的区别，并且在这个过程中，陆续介绍一些基本的 Haskell 标准库。

一个简单的命令行程序

在本章的大部分时间里，我们都只和无副作用的代码打交道。为了将注意力集中在实际的代码上，我们需要开发一个接口程序，连接起带副作用的代码和无副作用的代码。

这个接口程序读入一个文件，将函数应用到文件，并且将结果写到另一个文件：

```
-- file: ch04/InteractWith.hs

import System.Environment (getArgs)

interactWith function inputFile outputFile = do
  input <- readFile inputFile
  writeFile outputFile (function input)

main = mainWith myFunction
  where mainWith function = do
    args <- getArgs
    case args of
      [input,output] -> interactWith function input output
      _ -> putStrLn "error: exactly two arguments needed"

-- replace "id" with the name of our function below
myFunction = id
```

这是一个简单但完整的文件处理程序。其中 `do` 关键字引入一个块，标识那些带有副作用的代码，比如对文件进行读和写操作。被 `do` 包围的 `<-` 操作符效果等同于赋值。第七章还会介绍更多 I/O 方面的函数。

当我们需要测试其他函数的时候，我们就将程序中的 `id` 换成其他函数的名字。另外，这些被测试的函数的类型包含 `String -> String`，也即是，这些函数应该都接受并返回字符串值。

[译注： `id` 函数接受一个值，并原封不动地返回这个值，比如 `id "hello"` 返回值 `"hello"`，而 `id 10` 返回值 `10`。]

[译注：这一段最后一句的原文是“... need to have the type `String -> String` ...”，因为 Haskell 是一种带有类型多态的语言，所以将“have the type”翻译成“包含 xx 类型”，而不是“必须是 xx 类型”。

接下来编译这个程序：

```
$ ghc --make InteractWith
[1 of 1] Compiling Main           ( InteractWith.hs, InteractWith.o )
Linking InteractWith ...
```

通过命令行运行这个程序。它接受两个文件名作为参数输入，一个用于读取，一个用于写入：

 v: latest ▾

```
$ echo "hello world" > hello-in.txt
```

```
$ ./InteractWith hello-in.txt hello-out.txt

$ cat hello-in.txt
hello world

$ cat hello-out.txt
hello world
```

[译注：原书这里的执行过程少了写入内容到 `hello-in.txt` 的一步，导致执行会出错，所以这里加上 `echo ...` 这一步。另外原书执行的是 `Interact` 过程，也是错误的，正确的执行文件名应该是 `InteractWith`。]

循环的表示

和传统编程语言不同，Haskell 既没有 `for` 循环，也没有 `while` 循环。那么，如果有一大堆数据要处理，该用什么代替这些循环呢？这一节就会给出这个问题的几种可能的解决办法。

显式递归

通过例子进行比对，可以很直观地认识有 `loop` 语言和没 `loop` 语言之间的区别。以下是一个 C 函数，它将字符串表示的数字转换成整数：

```
int as_int(char *str)
{
    int acc; // accumulate the partial result
    for (acc = 0; isdigit(*str); str++) {
        acc = acc * 10 + (*str - '0');
    }

    return acc;
}
```

既然 Haskell 没有 `loop` 的话，以上这段 C 语言代码，在 Haskell 里该怎么表达呢？

让我们先从类型签名开始写起，这不是必须的，但它对于弄清楚代码在干什么很有帮助：

```
-- file: ch04/IntParse.hs
import Data.Char (digitToInt) -- we'll need ord shortly

asInt :: String -> Int
```

C 代码在遍历字符串的过程中，递增地计算结果。Haskell 代码同样可以做到这一点，而且，在 Haskell 里，使用函数已经足以表示 `loop` 计算了。[译注：在命令式语言里，很多迭代计算都是通过特殊关键字来实现的，比如 `do`、`while` 和 `for`。]

```
-- file: ch04/IntParse.hs
loop :: Int -> String -> Int



asInt xs = loop 0 xs
```

`loop` 的第一个参数是累积器的变量，给它赋值 0 等同于 C 语言在 `for` 循环开始前的初始化操作。

在研究详细的代码前，先来思考一下我们要处理的数据：输入 `xs` 是一个包含数字的字符串，而 `String` 类型不过是 `[Char]` 的别名，一个包含字符的列表。因此，要遍历处理字符串，最好的方法是将它看作是列表来处理：它要么就是一个空字符串；要么就是一个字符，后面跟着列表的其余部分。

以上的想法可以通过对列表的构造器进行模式匹配来表达。先从最简单的情况——输入为空字符串开始：

```
-- file: ch04/IntParse.hs
loop acc [] = acc
```

一个空列表并不仅仅意味着“输入列表为空”，另一种可能的情况是，对一个非空字符串进行遍历之后，最终到达了列表  `v: latest` ，对于空列表，我们不是抛出错误，而是将累积值返回。

另一个等式处理列表不为空的情况：先取出并处理列表的当前元素，接着处理列表的其他元素。

```
-- file: ch04/IntParse.hs
loop acc (x:xs) = let acc' = acc * 10 + digitToInt x
                  in loop acc' xs
```

程序先计算出当前字符所代表的数值，将它赋值给局部变量 `acc'`。然后使用 `acc'` 和剩余列表的元素 `xs` 作为参数，再次调用 `loop` 函数。这种调用等同于在 C 代码中再次执行一次循环。

每次递归调用 `loop`，累积器的值都会被更新，并处理掉列表里的一个元素。这样一直下去，最终输入列表为空，递归调用结束。

以下是 `IntParse` 函数的完整定义：

```
-- file: ch04/IntParse.hs

-- 只载入 Data.Char 中的 digitToInt 函数
import Data.Char (digitToInt)

asInt xs = loop 0 xs

loop :: Int -> String -> Int
loop acc [] = acc
loop acc (x:xs) = let acc' = acc * 10 + digitToInt x
                  in loop acc' xs
```

[译注：书本原来的代码少了对 `Data.Char` 的引用，会造成 `digitToInt` 查找失败。]

在 `ghci` 里看看程序的表现如何：

```
Prelude> :load IntParse.hs
[1 of 1] Compiling Main           ( IntParse.hs, interpreted )
Ok, modules loaded: Main.

*Main> asInt "33"
33
```

在处理字符串表示的字符时，它运行得很好。不过，如果传给它一些不合法的输入，这个可怜的函数就没办法处理了：

```
*Main> asInt ""
0
*Main> asInt "potato"
*** Exception: Char.digitToInt: not a digit 'p'
```

在练习一，我们会想办法解决这个问题。

`loop` 函数是尾递归函数的一个例子：如果输入非空，这个函数做的最后一件事，就是递归地调用自身。这个代码还展示了另一个惯用法：通过研究列表的结构，分别处理空列表和非空列表两种状况，这种方法称之为结构递归 (structural recursion)。

非递归情形（列表为空）被称为“基本情形”（有时也叫终止情形）。当对函数进行递归调用时，计算最终会回到基本情形上。在数学上，这也称为“归纳情形”。

作为一项有用的技术，结构递归并不仅仅局限于列表，它也适用于其他代数数据类型，稍后就会介绍更多这方面的例子。

对列表元素进行转换

考虑以下 C 函数，`square`，它对数组中的所有元素执行平方计算：

```
void square(double *out, const double *in, size_t length)
{
    for (size_t i = 0; i < length; i++) {
        out[i] = in[i] * in[i];
    }
}
```

 v: latest ▾

这段代码展示了一个直观且常见的 `loop` 动作，它对输入数组中的所有元素执行同样的动作。以下是 Haskell 版本的 `square` 函数：

```
-- file: ch04/square.hs

square :: [Double] -> [Double]

square (x:xs) = x*x : square xs
square []     = []
```

`square` 函数包含两个模式匹配等式。第一个模式解构一个列表，取出它的 `head` 部分和 `tail` 部分，并对第一个元素进行加倍操作，然后将计算所得的新元素放进列表里面。一直这样做下去，直到处理完整个列表为止。第二个等式确保计算会在列表为空时顺利终止。

`square` 产生一个和输入列表一样长的新列表，其中每个新元素的值都是原本元素的平方：

```
Prelude> :load square.hs
[1 of 1] Compiling Main          ( square.hs, interpreted )
Ok, modules loaded: Main.

*Main> let one_to_ten = [1.0 .. 10.0]

*Main> square one_to_ten
[1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
```

以下是另一个 C 循环，它将字符串中的所有字母都设置为大写：

```
#include <ctype.h>

char *uppercase(const char *in)
{
    char *out = strdup(in);

    if (out != NULL) {
        for (size_t i = 0; out[i] != '\0'; i++) {
            out[i] = toupper(out[i]);
        }
    }

    return out;
}
```

以下是相等的 Haskell 版本：

```
-- file: ch04/upperCase.hs

import Data.Char (toUpper)

upperCase :: String -> String

upperCase (x: xs) = toUpper x : upperCase xs
upperCase []     = []
```

代码从 `Data.Char` 模块引入了 `toUpper` 函数，如果输入字符是一个字母的话，那么函数就将它转换成大写：

```
Prelude> :module +Data.Char

Prelude Data.Char> toUpper 'a'
'a'

Prelude Data.Char> toUpper 'A'
'A'

Prelude Data.Char> toUpper '1'
'1'

Prelude Data.Char> toUpper '*'
'*'
```

 v: latest ▾

`upperCase` 函数和之前的 `square` 函数很相似：如果输入是一个空列表，那么它就停止计算，返回一个空列表。另一方面，如果输入不为空，那么它就是对列表的第一个元素调用 `toUpper` 函数，并且递归调用自身，继续处理剩余的列表元素：

```
Prelude> :load upperCase.hs
[1 of 1] Compiling Main           ( upperCase.hs, interpreted )
Ok, modules loaded: Main.

*Main> upperCase "The quick brown fox jumps over the lazy dog"
"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
```

以上两个函数遵循了同一种处理列表的公共模式：基本情形处理（base case）空列表输入。而递归情形（recursive case）则处理列表非空时的情况，它对列表的头元素进行某种操作，然后递归地处理列表余下的其他元素。

列表映射

前面定义的 `square` 和 `upperCase` 函数，都生成一个和输入列表同等长度的新列表，并且每次只对列表的一个元素进行处理。因为这种用法非常常见，所以 Haskell 的 Prelude 库定义了 `map` 函数来更方便地执行这种操作。`map` 函数接受一个函数和一个列表作为参数，将输入函数应用到输入列表的每个元素上，并构建出一个新的列表。

以下是使用 `map` 重写的 `square` 和 `upperCase` 函数：

```
-- file: ch04/rewrite_by_map.hs

import Data.Char (toUpper)

square2 xs = map squareOne xs
  where squareOne x = x * x

upperCase2 xs = map toUpper xs
```

[译注：原文代码没有载入 `Data.Char` 中的 `toUpper` 函数。]

来研究一下 `map` 是如何实现的。通过查看它的类型签名，可以发现很多有意思的信息：

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

类型签名显示，`map` 接受两个参数：第一个参数是一个函数，这个函数接受一个 `a` 类型的值，并返回一个 `b` 类型的值[译注：这里只是说 `a` 和 `b` 类型可能不一样，但不是必须的。]。

像 `map` 这种接受一个函数作为参数、又或者返回一个函数作为结果的函数，被称为高阶函数。

因为 `map` 的抽象出现在 `square` 和 `upperCase` 函数，所以可以通过观察这两个函数，找出它们之间的共同点，然后实现自己的 `map` 函数：

```
-- file: ch04/myMap.hs

myMap :: (a -> b) -> [a] -> [b]

myMap f (x:xs) = f x : myMap f xs
myMap _ [] = []
```

[译注：在原文的代码里，第二个等式的定义为 `myMap _ [] = []`，这并不是完全正确的，因为它可以适配于第二个参数不为列表的情况，比如 `myMap f 1`。因此，这里遵循标准库里 `map` 的定义，将第二个等式修改为 `myMap _ [] = []`。]

在 `ghci` 测试这个 `myMap` 函数：

```
Prelude> :load myMap.hs
[1 of 1] Compiling Main           ( myMap.hs, interpreted )
Ok, modules loaded: Main.

*Main> :module +Data.Char
```

v: latest

```
*Main Data.Char> myMap toUpper "The quick brown fox"
"THE QUICK BROWN FOX"
```

通过观察代码，并从中提炼出重复的抽象，是复用代码的一种良好方法。尽管对代码进行抽象并不是 Haskell 的“专利”，但高阶函数使得这种抽象变得非常容易。

筛选列表元素

另一种对列表的常见操作是，对列表元素进行筛选，只保留那些符合条件的元素。

以下函数接受一个列表作为参数，并返回这个列表里的所有奇数元素。代码的递归情形比之前的 `map` 函数要复杂一些，它使用守卫对元素进行条件判断，只有符合条件的元素，才会被加入进结果列表里：

```
-- file: ch04/oddList.hs

oddList :: [Int] -> [Int]

oddList (x:xs) | odd x    = x : oddList xs
               | otherwise = oddList xs
oddList []              = []
```

[译注：这里将原文代码的 `oddList _ = []` 改为 `oddList [] = []`，原因和上一小节修改 `map` 函数的代码一样。]

测试：

```
Prelude> :load oddList.hs
[1 of 1] Compiling Main                ( oddList.hs, interpreted )
Ok, modules loaded: Main.

*Main> oddList [1..10]
[1,3,5,7,9]
```

因为这种过滤模式也很常见，所以 Prelude 也定义了相应的函数 `filter`：它接受一个谓词函数，并将它应用到列表里的每个元素，只有那些谓词函数求值返回 `True` 的元素才会被保留：

```
Prelude> :type odd
odd :: Integral a => a -> Bool

Prelude> odd 1
True

Prelude> odd 2
False

Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]

Prelude> filter odd [1..10]
[1,3,5,7,9]
```

[译注：谓词函数是指那种返回 `Bool` 类型值的函数。]

稍后的章节会介绍如何定义 `filter`。

处理收集器并得出结果

将一个收集器（collection）简化（reduce）为一个值也是收集器的常见操作之一。

对列表的所有元素求和，就是其中的一个例子：

```
-- file: ch04/mySum.hs

mySum xs = helper 0 xs
```

v: latest

```
where helper acc (x:xs) = helper (acc + x) xs
      helper acc []      = acc
```

`helper` 函数通过尾递归进行计算。`acc` 是累积器参数，它记录了当前列表元素的总和。正如我们在 `asInt` 函数看到的那样，这种递归计算是纯函数语言里表示 `loop` 最自然的方式。

以下是一个稍微复杂一些的例子，它是一个 Adler-32 校验和的 JAVA 实现。Adler-32 是一个流行的校验和算法，它将两个 16 位校验和串联成一个 32 位校验和。第一个校验和是所有输入比特之和，加上一。而第二个校验和则是第一个校验和所有中间值之和。每次计算时，校验和都需要取模 65521。（如果你不懂 JAVA，直接跳过也没关系）：

```
public class Adler32
{
    private static final int base = 65521;

    public static int compute(byte[] data, int offset, int length)
    {
        int a = 1, b = 0;

        for (int i = offset; i < offset + length; i++) {
            a = (a + (data[i] & 0xff)) % base;
            b = (a + b) % base;
        }

        return (b << 16) | a;
    }
}
```

尽管 Adler-32 是一个简单的校验和算法，但这个 JAVA 实现还是非常复杂，很难看清楚位操作之间的关系。

以下是 Adler-32 算法的 Haskell 实现：

```
-- file: ch04/Adler32.hs

import Data.Char (ord)
import Data.Bits (shiftL, (.&.), (.|.))

base = 65521

adler32 xs = helper 1 0 xs
  where helper a b (x:xs) = let a' = (a + (ord x .&. 0xff)) `mod` base
                             b' = (a' + b) `mod` base
                             in helper a' b' xs
      helper a b []      = (b `shiftL` 16) .|. a
```

在这段代码里，`shiftL` 函数实现逻辑左移，`(.&.)` 实现二进制位的并操作，`(.|.)` 实现二进制位的或操作，`ord` 函数则返回给定字符对应的编码值。

`helper` 函数通过尾递归来进行计算，每次对它的调用，都产生新的累积变量，效果等同于 JAVA 在 `for` 循环里对变量的赋值更新。当列表处理完毕，递归终止时，程序计算出校验和并将它返回。

和前面抽取出的 `map` 和 `filter` 函数类似，从 `Adler32` 函数里面，我们也可以抽取出一种通用的抽象，称之为折叠（fold）：它对一个列表中的所有元素做某种处理，并且一边处理元素，一边更新累积器，最后在处理完整个列表之后，返回累积器的值。

有两种不同类型的折叠，其中 `foldl` 从左边开始进行折叠，而 `foldr` 从右边开始进行折叠。

左折叠

以下是 `foldl` 函数的定义：

```
-- file: ch04/foldl.hs

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

foldl step zero (x:xs) = foldl step (step zero x) xs
foldl _ zero []       = zero
```

 v: latest ▾



[译注：这个函数在载入 ghci 时会因为命名冲突而被拒绝，编写函数直接使用内置的 `foldl` 就可以了。]

`foldl` 函数接受一个步骤 (step) 函数，一个累积器的初始化值，以及一个列表作为参数。步骤函数每次使用累积器和列表中的一个元素作为参数，并计算出新的累积器值，这个过程称为步进 (stepper)。然后，将新的累积器作为参数，再次进行同样的计算，直到整个列表处理完为止。

以下是使用 `foldl` 重写的 `mySum` 函数：

```
-- file: ch04/foldlSum.hs
foldlSum xs = foldl step 0 xs
  where step acc x = acc + x
```

因为代码里的 `step` 函数执行的操作不过是相加起它的两个输入参数，因此，可以直接将一个加法函数代替 `step` 函数，并移除多余的 `where` 语句：

```
-- file: ch04/niceSum.hs
niceSum :: [Integer] -> Integer
niceSum xs = foldl (+) 0 xs
```

为了进一步看清楚 `foldl` 的执行模式，以下代码展示了 `niceSum [1, 2, 3]` 执行时的计算过程：

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

注意对比新的 `mySum` 定义比刚开始的定义节省了多少代码：新版本没有使用显式递归，因为 `foldl` 可以代替我们搞定了关于循环的一切。现在程序只要求我们回答两个问题：第一，累积器的初始化值是什么 (`foldl` 的第二个参数)；第二，怎么更新累积器的值 (`(+)` 函数)。

为什么使用 `fold`、`map` 和 `filter`？

回顾一下之前的几个例子，可以看出，使用 `fold` 和 `map` 等高阶函数定义的函数，比起显式使用递归的函数，并不总是能节约大量代码。那么，我们为什么要使用这些函数呢？

答案是，因为它们 **在 Haskell 中非常通用，并且这些函数都带有正确的、可预见的行为。**

这意味着，即使是一个 Haskell 新手，他/她理解起 `fold` 通常都要比理解显式递归要容易。一个 `fold` 并不产生任何意外动作，但一个显式递归函数的所做作为，通常并不是那么显而易见的。

以上观点同样适用于其他高阶函数库，包括前面介绍过的 `map` 和 `filter`。因为这些函数都带有定义良好的行为，我们只需要学习怎样使用这些函数一次，以后每次碰到使用这些函数的代码，这些知识都可以加快我们对代码的理解。这种优势同样体现在代码的编写上：一旦我们能熟练使用高阶函数，那么写出更简洁的代码自然就不在话下。

从右边开始折叠

和 `foldl` 相对应的是 `foldr`，它从一个列表的右边开始进行折叠。

```
-- file: ch04/foldr.hs

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr step zero (x:xs) = step x (foldr step zero xs)
foldr _ zero []       = zero
```

[译注：这个函数在载入 ghci 时会因为命名冲突而被拒绝，编写函数直接使用内置的 `foldr` 就可以了。]

可以用 `foldr` 改写在《左折叠》一节定义的 `niceSum` 函数：

v: latest ▾


```
-- file: ch04/niceSumFoldr.hs

niceSumFoldr :: [Int] -> Int
niceSumFoldr xs = foldr (+) 0 xs
```

这个 `niceSumFoldr` 函数在输入为 `[1, 2, 3]` 时，产生以下计算序列：

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

可以通过观察括号的包围方式，以及累积器初始化值摆放的位置，来区分 `foldl` 和 `foldr`：`foldl` 将初始化值放在左边，括号也是从左边开始包围。另一方面，`foldr` 将初始化值放在右边，而括号也是从右边开始包围。

还记得当年大明湖畔的 `filter` 函数吗？它可以用显式递归来定义：

```
-- file: ch04/filter.hs

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

[译注：这个函数在载入 `ghci` 时会因为命名冲突而被拒绝，编写函数直接使用内置的 `filter` 就可以了。]

除此之外，`filter` 还可以通过 `foldr` 来定义：

```
-- file: ch04/myFilter.hs

myFilter p xs = foldr step [] xs
  where step x ys | p x      = x : ys
                  | otherwise = ys
```

来仔细分析一下 `myFilter` 函数的定义：和 `foldl` 一样，`foldr` 也接受一个函数、一个基本情形和一个列表作为参数。通过阅读 `filter` 函数的类型签名可以得知，`myFilter` 函数的输入和输出都使用同类型的列表，因此函数的基本情形，以及局部函数 `step`，都必须返回这个类型的列表。

`myFilter` 里的 `foldr` 每次取出列表中的一个元素，并对他进行处理，如果这个元素经过条件判断为 `True`，那么就将它放进累积的新列表里面，否则，它就略过这个元素，继续处理列表的其他剩余元素。

所有可以用 `foldr` 定义的函数，统称为 **主递归** (primitive recursive)。很大一部分列表处理函数都是主递归函数。比如说，`map` 就可以用 `foldr` 定义：

```
-- file: ch04/myFoldrMap.hs

myFoldrMap :: (a -> b) -> [a] -> [b]

myFoldrMap f xs = foldr step [] xs
  where step x xs = f x : xs
```

更让人惊奇的是，`foldl` 甚至可以用 `foldr` 来表示：

```
-- file: ch04/myFoldl.hs

myFoldl :: (a -> b -> a) -> a -> [b] -> a

myFoldl f z xs = foldr step id xs z
  where step x g a = g (f a x)
```

一种思考 `foldr` 的方式是，将它看成是对输入列表的一种 *转换* (transform)：它的第一个参数决定了该怎么处理列表的 `head` 和 `tail` 部分；而它的第二个参数则决定了，当遇到空列表时，该用什么值来代替这个空列表。

用 `foldr` 定义的恒等 (identity) 转换，在列表为空时，返回空列表本身；如果列表不为空，那么它就将列表构造器 `(:)` 应用于每个 `head` 和 `tail` 对 (pair)：

```
-- file: ch04/identity.hs

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

最终产生的结果列表和原输入列表一模一样：

```
Prelude> :load identity.hs
[1 of 1] Compiling Main          ( identity.hs, interpreted )
Ok, modules loaded: Main.

*Main> identity [1, 2, 3]
[1, 2, 3]
```

如果将 `identity` 函数定义中，处理空列表时返回的 `[]` 改为另一个列表，那么我们就得到了列表追加函数 `append`：

```
-- file: ch04/append.hs
append :: [a] -> [a] -> [a]
append xs ys = foldr (:) ys xs
```

测试：

```
Prelude> :load append.hs
[1 of 1] Compiling Main          ( append.hs, interpreted )
Ok, modules loaded: Main.

*Main> append "the quick " "fox"
"the quick fox"
```

这个函数的效果等同于 `(++)` 操作符：

```
*Main> "the quick " ++ "fox"
"the quick fox"
```

`append` 函数依然对每个列表元素使用列表构造器，但是，当第一个输入列表为空时，它将第二个输入列表（而不是空列表元素）拼接第一个输入列表的表尾。


通过前面这些例子对 `foldr` 的介绍，我们应该可以了解到，`foldr` 函数和《列表处理》一节所介绍的基本列表操作函数一样重要。由于 `foldr` 可以增量地处理和产生列表，所以它对于惰性数据处理也非常有用。

左折叠、惰性和内存泄漏

为了简化讨论，本节的例子通常都使用 `foldl` 来进行，这对于普通的测试是没有问题的，但是，千万不要把 `foldl` 用在实际使用中。

这样做是因为，Haskell 使用的是非严格求值。如果我们仔细观察 `foldl (+) [1, 2, 3]` 的执行过程，就可以从中看出一些问题：

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

除非被显式地要求，否则最后的表达式不会被求值为 6。在表达式被求值之前，它会被保存在块里面。保存一个块比保存单独一个数字要昂贵得多，而被块保存的表达式越复杂，这个块占用的空间就越多。对于数值计算这样的廉价操作来说，块保存这种  `v: latest` 计算量，比直接求值这个表达式所需的计算量还多。最终，我们既浪费了时间，又浪费了金钱。

在 GHC 中，对块中表达式的求值在一个内部栈中进行。因为块中的表达式可能是无限大的，而 GHC 为栈设置了有限大的容量，多得这个限制，我们可以在 ghci 里尝试求值一个大的块，而不必担心消耗掉全部内存。

[译注：使用栈来执行一些可能无限大的操作，是一种常见优化和保护技术。这种用法减少了操作系统显式的上下文切换，而且就算计算量超出栈可以容纳的范围，那么最坏的结果就是栈崩溃，而如果直接使用系统内存，一旦请求超出内存可以容纳的范围，可能会造成整个程序崩溃，甚至影响系统的稳定性。]

```
Prelude> foldl (+) 0 [1..1000]
500500
```

可以推测出，在上面的表达式被求值之前，它创建了一个保存 1000 个数字和 999 个 (+) 操作的块。单单为了表示一个数字，我们就用了非常多的内存和 CPU！

[译注：这些块到底有多大？算算就知道了：对于每一个加法表达式，比如 $x + y$ ，都要使用一个块来保存。而这种操作在 `foldl (+) 0 [1..1000]` 里要执行 999 次，因此一共有 999 个块被创建，这些块都保存着像 $x + y$ 这样的表达式。]

对于一个更大的表达式——尽管它并不是真的非常庞大，`foldl` 的执行会失败：

```
ghci> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
```

对于小的表达式来说，`foldl` 可以给出正确的答案，但是，因为过度的块资源占用，它运行非常缓慢。我们称这种现象为内存泄漏：代码可以正确地执行，但它占用了比实际所需多得多的内存。

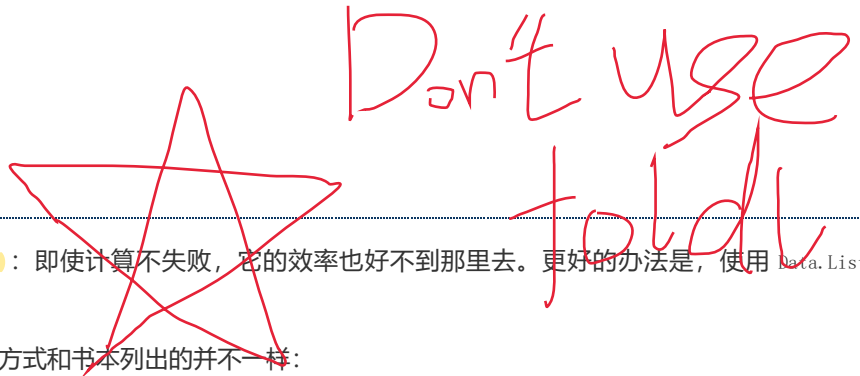
对于大的表达式来说，带有内存泄漏的代码会造成运行失败，就像前面例子展示的那样。

内存泄漏是 Haskell 新手常常会遇到的问题，幸好的是，它非常容易避免。`Data.List` 模块定义了一个 `foldl'` 函数，它和 `foldl` 的作用类似，唯一的区别是，`foldl'` 并不创建块。以下的代码直观地展示了它们的区别：

```
ghci> foldl (+) 0 [1..1000000]
*** Exception: stack overflow

ghci> :module +Data.List

ghci> foldl' (+) 0 [1..1000000]
5000000500000
```



综上所述，最好不要在实际代码中使用 `foldl`：即使计算不失败，它的效率也好不到那里去。更好的办法是，使用 `Data.List` 里面的 `foldl'` 来代替。

[译注：在我的电脑上，超出内存的 `foldl` 失败方式和书本列出的并不一样：

```
Prelude> foldl (+) 0 [1..1000000000]
<interactive>: internal error: getMBlock: mmap: Operation not permitted
(GHC version 7.4.2 for i386_unknown_linux)
Please report this as a GHC bug: http://www.haskell.org/ghc/reportabug
已放弃
```

从错误信息看，GHC/GHci 处理 `foldl` 的方式应该已经发生了变化。

如果使用 `foldl'` 来执行计算，就不会出现任何问题：

```
Prelude> :module +Data.List

Prelude Data.List> foldl' (+) 0 [1..1000000000]
5000000000500000000
```

就是这样。]

延伸阅读

v: latest ▾

A tutorial on the universality and expressiveness of fold 是一篇关于 fold 的优秀且深入的文章。它使用了很多例子来展示如何通过简单的系统化计算技术，将一些显式递归的函数转换成 fold。

匿名 (lambda) 函数

在前面章节定义的函数中，很多函数都带有一个简单的辅助函数：

```
-- file: ch04/isInAny.hs

import Data.List (isInfixOf)

isInAny needle haystack = any inSequence haystack
  where inSequence s = needle `isInfixOf` s
```

Haskell 允许我们编写完全匿名的函数，这样就不必再费力地为辅助函数想名字了。因为匿名函数从 lambda 演算而来，所以匿名函数通常也被称为 lambda 函数。

在 Haskell 中，匿名函数以反斜杠符号 `\` 为开始，后跟函数的参数（可以包含模式），而函数体定义在 `->` 符号之后。其中，`\` 符号读作 *lambda*。

以下是前面的 `isInAny` 函数用 lambda 改写的版本：

```
-- file: ch04/isInAny2.hs

import Data.List (isInfixOf)

isInAny2 needle haystack = any (\s -> needle `isInfixOf` s) haystack
```

定义使用括号包裹了整个匿名函数，确保 Haskell 可以知道匿名函数体在那里结束。

匿名函数各个方面的行为都和带名称的函数基本一致，但是，匿名函数的定义受到几个严格的限制，其中最重要的一点是：普通函数可以通过多条语句来定义，而 lambda 函数的定义只能有一条语句。

只能使用一条语句的局限性，限制了在 lambda 定义中可使用的模式。一个普通函数，通常要使用多条定义，来覆盖各种不同的模式：

```
-- file: ch04/safeHead.hs

safeHead (x:_) = Just x
safeHead [] = Nothing
```

而 lambda 只能覆盖其中一种情形：

```
-- file ch04/unsafeHead.hs

unsafeHead = \ (x:_) -> x
```


如果一不小心，将这个函数应用到错误的模式上，它就会给我们带来麻烦：

```
Prelude> :load unsafeHead.hs
[1 of 1] Compiling Main          ( unsafeHead.hs, interpreted )
Ok, modules loaded: Main.

*Main> :type unsafeHead
unsafeHead :: [t] -> t

*Main> unsafeHead [1]
1

*Main> unsafeHead []
*** Exception: unsafeHead.hs:2:14-24: Non-exhaustive patterns in lambda
```

因为这个 lambda 定义是完全合法的，它的类型也没有错误，所以它可以被顺利编译，而最终在运行期产生错误。这个  `v: latest` 如果你在 lambda 函数里使用模式，请千万小心，必须确保你的模式不会匹配失败。

另外需要注意的是，在前面定义的 `isInAny` 函数和 `isInAny2` 函数里，带有辅助函数的 `isInAny` 要比使用 `lambda` 的 `isInAny2` 要更具可读性。带有名字的辅助函数不会破坏程序的代码流（flow），而且它的名字也可以传达更多的相关信息。

相反，当在一个函数定义里面看到 `lambda` 时，我们必须慢下来，仔细阅读这个匿名函数的定义，弄清楚它都干了些什么。为了程序的可读性和可维护性考虑，我们在很多情况下都会避免使用 `lambda`。

当然，这并不是说 `lambda` 函数完全没用，只是在使用它们的时候，必须小心谨慎。

很多时候，部分应用函数可以很好地代替 `lambda` 函数，避免不必要的函数定义，粘合起不同的函数，并产生更清晰和更可读的代码。下一节就会介绍部分应用函数。

部分函数应用和柯里化

类型签名里的 `->` 可能会让人感到奇怪：

```
Prelude> :type dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
```

初看上去，似乎 `->` 既用于隔开 `dropWhile` 函数的各个参数（比如括号里的 `a` 和 `Bool`），又用于隔开函数参数和返回值的类型（`(a -> Bool) -> [a]` 和 `[a]`）。

但是，实际上 `->` 只有一种作用：它表示一个函数接受一个参数，并返回一个值。其中 `->` 符号的左边是参数的类型，右边是返回值的类型。

理解 `->` 的含义非常重要：在 Haskell 中，所有函数都只接受一个参数。尽管 `dropWhile` 看上去像是一个接受两个参数的函数，但实际上它是一个接受一个参数的函数，而这个函数的返回值是另一个函数，这个被返回的函数也只接受一个参数。

以下是一个完全合法的 Haskell 表达式：

```
Prelude> :module +Data.Char

Prelude Data.Char> :type dropWhile isSpace
dropWhile isSpace :: [Char] -> [Char]
```

表达式 `dropWhile isSpace` 的值是一个函数，这个函数移除一个字符串的所有前置空白。作为一个例子，可以将它应用到一个高阶函数：

```
Prelude Data.Char> map (dropWhile isSpace) ["a", "f", "   e"]
["a", "f", "e"]
```

每当我们将一个参数传给一个函数时，这个函数的类型签名最前面的一个元素就会被“移除掉”。这里用函数 `zip3` 来做例子，这个函数接受三个列表，并将它们压缩成一个包含三元组的列表：

```
Prelude> :type zip3
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]

Prelude> zip3 "foo" "bar" "quux"
[('f', 'b', 'q'), ('o', 'a', 'u'), ('o', 'r', 'u')]
```

如果只将一个参数应用到 `zip3` 函数，那么它就会返回一个接受两个参数的函数。无论之后将什么参数传给这个复合函数，之前传给它的第一个参数的值都不会改变。

```
Prelude> :type zip3
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]

Prelude> :type zip3 "foo"
zip3 "foo" :: [b] -> [c] -> [(Char, b, c)]

Prelude> :type zip3 "foo" "bar"
zip3 "foo" "bar" :: [c] -> [(Char, Char, c)]

Prelude> :type zip3 "foo" "bar" "quux"
zip3 "foo" "bar" "quux" :: [(Char, Char, Char)]
```

 v: latest ▾

传入参数的数量，少于函数所能接受参数的数量，这种情况被称为函数的 **部分应用** (partial application of the function)：函数正被它的其中几个参数所应用。

在上面的例子中，`zip3 "foo"` 就是一个部分应用函数，它以 `"foo"` 作为第一个参数，部分应用了 `zip3` 函数；而 `zip3 "foo" "bar"` 也是另一个部分应用函数，它以 `"foo"` 和 `"bar"` 作为参数，部分应用了 `zip3` 函数。

只要给部分函数补充上足够的参数，它就可以被成功求值：

```
Prelude> let zip3foo = zip3 "foo"

Prelude> zip3foo "bar" "quux"
[('f','b','q'),('o','a','u'),('o','r','u')]

Prelude> let zip3foobar = zip3 "foo" "bar"

Prelude> zip3foobar "quux"
[('f','b','q'),('o','a','u'),('o','r','u')]

Prelude> zip3foobar [1, 2, 3]
[('f','b',1),('o','a',2),('o','r',3)]
```

部分函数应用 (partial function application) 让我们免于编写烦人的一次性函数，而且它比起之前介绍的匿名函数要来得更有用。回顾之前的 `isInAny` 函数，以下是一个部分应用函数改写的版本，它既不需要匿名函数，也不需要辅助函数：

```
-- file: ch04/isInAny3.hs

import Data.List (isInfixOf)

isInAny3 needle haystack = any (isInfixOf needle) haystack
```

表达式 `isInfixOf needle` 是部分应用函数，它以 `needle` 变量作为第一个参数，传给 `isInfixOf`，并产生一个部分应用函数，这个部分应用函数的作用等同于 `isInAny` 定义的辅助函数，以及 `isInAny2` 定义的匿名函数。

部分函数应用被称为柯里化 (currying)，以逻辑学家 Haskell Curry 命名 (Haskell 语言的命名也是来源于他的名字)。

以下是另一个使用柯里化的例子。先来回顾《左折叠》章节的 `niceSum` 函数：

```
-- file: ch04/niceSum.hs
niceSum :: [Integer] -> Integer
niceSum xs = foldl (+) 0 xs
```

实际上，并不需要完全应用 `foldl` [译注：完全应用是指提供函数所需的全部参数]，`niceSum` 函数的 `xs` 参数，以及传给 `foldl` 函数的 `xs` 参数，这两者都可以被省略，最终得到一个更紧凑的函数，它的类型也和原本的一样：

```
-- file: ch04/niceSumPartial.hs
niceSumPartial :: [Integer] -> Integer
niceSumPartial = foldl (+) 0
```

测试：

```
Prelude> :load niceSumPartial.hs
[1 of 1] Compiling Main          ( niceSumPartial.hs, interpreted )
Ok, modules loaded: Main.

*Main> niceSumPartial [1..10]
55
```

节

Haskell 提供了一种方便的符号快捷方式，用于对中序函数进行部分应用：使用括号包围一个操作符，通过在括号里面提供左操作对象或者右操作对象，可以产生一个部分应用函数。这种类型的部分函数应用称之为节 (section)。

 v: latest ▾

```
Prelude> (1+) 2
3

Prelude> map (*3) [24, 36]
[72, 108]

Prelude> map (2^) [3, 5, 7, 9]
[8, 32, 128, 512]
```

如果向节提供左操作对象，那么得出的部分函数就会将接收到的参数应用为右操作对象，反之亦然。

以下两个表达式都计算 2 的 3 次方，但是第一个节接受的是左操作对象 2，而第二个节接受的则是右操作对象 3。

```
Prelude> (2^) 3
8

Prelude> (^3) 2
8
```

之前提到过，通过使用反括号来包围一个函数，可以将这个函数用作中序操作符。这种用法可以让节使用函数：

```
Prelude> :type (`elem` ['a' .. 'z'])
(`elem` ['a' .. 'z']) :: Char -> Bool
```

上面的定义将 ['a' .. 'z'] 传给 elem 作为第二个参数，表达式返回的函数可以用于检查一个给定字符值是否属于小写字母：

```
Prelude> (`elem` ['a' .. 'z']) 'f'
True

Prelude> (`elem` ['a' .. 'z']) '1'
False
```

还可以将这个节用作 all 函数的输入，这样就得到了一个检查给定字符串是否整个字符串都由小写字母组成的函数：

```
Prelude> all (`elem` ['a' .. 'z']) "Haskell"
False

Prelude> all (`elem` ['a' .. 'z']) "haskell"
True
```

通过这种用法，可以再一次提升 isInAny3 函数的可读性：

```
-- file: ch04/isInAny4.hs

import Data.List (isInfixOf)

isInAny4 needle haystack = any (needle `isInfixOf`) haystack
```

[译注：根据前面部分函数部分提到的技术，这个 isInAny4 的定义还可以进一步精简，去除 haystack 参数：

```
import Data.List (isInfixOf)
isInAny4Partial needle = any (needle `isInfixOf`)
```

]

As-模式

Data.List 模块里定义的 tails 函数是 tail 的推广，它返回一个列表的所有“尾巴”：

```
Prelude> :m +Data.List

Prelude Data.List> tail "foobar"
```

 v: latest ▾


```
"oobar"

Prelude Data.List> tail (tail "foobar")
"oobar"

Prelude Data.List> tails "foobar"
["foobar", "oobar", "obar", "bar", "ar", "r", ""]
```

`tails` 返回一个包含字符串的列表，这个列表保存了输入字符串的所有后缀，以及一个额外的空列表（放在结果列表的最后）。`tails` 的返回值总是带有额外的空列表，即使它的输入为空时：

```
Prelude Data.List> tails ""
[""]
```

如果想要一个行为和 `tails` 类似，但是并不包含空列表后缀的函数，可以自己写一个：

```
-- file: ch04/suffixes.hs

suffixes :: [a] -> [[a]]
suffixes xs@(_:xs') = xs : suffixes xs'
suffixes [] = []
```

[译注：在稍后的章节就会看到，有简单得多的方法来完成这个目标，这个例子主要用于展示 `as`-模式的作用。]

源码里面用到了新引入的 `@` 符号，模式 `xs@(_:xs')` 被称为 `as`-模式，它的意思是：如果输入值能匹配 `@` 符号右边的模式（这里是 `(_:xs')`），那么就将这个值绑定到 `@` 符号左边的变量中（这里是 `xs`）。

在这个例子中，如果输入值能够匹配模式 `(_:xs')`，那么这个输入值这就被绑定为 `xs`，它的 `tail` 部分被绑定为 `xs'`，而它的 `head` 部分因为使用通配符 `_` 进行匹配，所以这部分没有被绑定到任何变量。

```
*Main Data.List> tails "foo"
["foo", "oo", "o", ""]

*Main Data.List> suffixes "foo"
["foo", "oo", "o"]
```

`As`-模式可以提升代码的可读性，作为对比，以下是一个没有使用 `as`-模式的 `suffixes` 定义：

```
-- file: noAsPattern.hs

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

可以看到，使用 `as`-模式的定义同时完成了模式匹配和变量绑定两项工作。而不使用 `as`-模式的定义，则需要在对列表进行结构之后，在函数体里又重新对列表进行组合。

除了增强可读性之外，`as`-模式还有其他作用：它可以对输入数据进行共享，而不是复制它。在 `noAsPattern` 函数的定义中，当 `(x:xs)` 匹配时，在函数体里需要复制一个 `(x:xs)` 的副本。这个动作会引起内存分配。虽然这个分配动作可能很廉价，但它并不是免费的。相反，当使用 `suffixes` 函数时，我们通过变量 `xs` 重用匹配了 `as`-模式的输入值，因此就避免了内存分配。

通过组合函数来进行代码复用

前面的 `suffixes` 函数实际上有一种更简单的实现方式。

回忆前面在《使用列表》一节里介绍的 `init` 函数，它可以返回一个列表中除了最后一个元素之外的其他元素。而组合使用 `init` 和 `tails`，可以给出一个 `suffixes` 函数的更简单实现：

```
-- file: ch04/suffixes.hs

import Data.List (tails)

suffixes2 xs = init (tails xs)
```

v: latest ▾

`suffixes2` 和 `suffixes` 函数的行为完全一样，但 `suffixes2` 的定义只需一行：

```
Prelude> :load suffixes2.hs
[1 of 1] Compiling Main           ( suffixes2.hs, interpreted )
Ok, modules loaded: Main.

*Main> suffixes2 "foobar"
["foobar", "oobar", "obar", "bar", "ar", "r"]
```

如果仔细地观察，就会发现这里隐含着一种模式：我们先应用一个函数，然后又将这个函数得出的结果应用到另一个函数。可以将这个模式定义为一个函数：

```
-- file: ch04/compose.hs

compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)
```

`compose` 函数可以用于粘合两个函数：

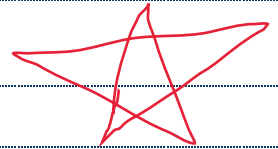
```
Prelude> :load compose.hs
[1 of 1] Compiling Main           ( compose.hs, interpreted )
Ok, modules loaded: Main.

*Main> :m +Data.List

*Main Data.List> let suffixes3 xs = compose init tails xs
```

通过柯里化，可以丢掉 `xs` 函数：

```
*Main Data.List> let suffixes4 = compose init tails
```



更棒的是，其实我们并不需要自己编写 `compose` 函数，因为 Haskell 已经内置在了 `Prelude` 里面，使用 `(.)` 操作符就可以组合起两个函数：

```
*Main Data.List> let suffixes5 = init . tails
```

`(.)` 操作符并不是什么特殊语法，它只是一个普通的操作符：

```
*Main Data.List> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c

*Main Data.List> :type suffixes5
suffixes5 :: [a] -> [[a]]

*Main Data.List> suffixes5 "foobar"
["foobar", "oobar", "obar", "bar", "ar", "r"]
```

在任何时候，都可以通过使用 `(.)` 来组合函数，并产生新函数。组合链的长度并没有限制，只要 `(.)` 符号右边函数的输出值类型适用于 `(.)` 符号左边函数的输入值类型就可以了。

也即是，对于 `f . g` 来说，`g` 的输出值必须是 `f` 能接受的类型，这样的组合就是合法的，`(.)` 的类型签名也显示了这一点。

作为例子，再来解决一个非常常见的问题：计算字符串中以大写字母开头的单词的个数：

```
Prelude> :module +Data.Char

Prelude Data.Char> let capCount = length . filter (isUpper . head) . words

Prelude Data.Char> capCount "Hello there, Mon!"
2
```

v: latest

来逐步分析 `capCount` 函数的组合过程。因为 `(.)` 操作符是右关联的，因此我们从组合链的最右边开始研究：

```
Prelude Data.Char> :type words
words :: String -> [String]
```

`words` 返回一个 `[String]` 类型值，因此 `(.)` 的左边的函数必须能接受这个参数。

```
Prelude Data.Char> :type isUpper . head
isUpper . head :: [Char] -> Bool
```

上面的组合函数在输入字符串以大写字母开头时返回 `True`，因此 `filter (isUpper . head)` 表达式会返回所有以大写字母开头的字符串：

```
Prelude Data.Char> :type filter (isUpper . head)
filter (isUpper . head) :: [[Char]] -> [[Char]]
```

因为这个表达式返回一个列表，而 `length` 函数用于统计列表的长度，所以 `length . filter (isUpper . head)` 就计算出了所有以大写字母开头的字符串的个数。

以下是另一个例子，它从 `libpcap` —— 一个流行的网络包过滤库中提取 C 文件头中给定格式的宏名字。这些头文件带有很多以下格式的宏：

```
#define DLT_EN10MB    1    /* Ethernet (10Mb) */
#define DLT_EN3MB     2    /* Experimental Ethernet (3Mb) */
#define DLT_AX25      3    /* Amateur Radio AX.25 */
```

我们的目标是提取出所有像 `DLT_AX25` 和 `DLT_EN3MB` 这种名字。以下是程序的定义，它将整个文件看作是一个字符串，先使用 `lines` 对文件进行按行分割，再将 `foldr step []` 应用到各行当中，其中 `step` 辅助函数用于过滤和提取符合格式的宏名字：

```
-- file: ch04/dlts.hs

import Data.List (isPrefixOf)

dlts :: String -> [String]

dlts = foldr step [] . lines
  where step l ds
        | "#define DLT_" `isPrefixOf` l = secondWord l : ds
        | otherwise                     = ds
        secondWord = head . tail . words
```

程序通过守卫表达式来过滤输入：如果输入字符串符合给定格式，就将它加入到结果列表里；否则，就略过这个字符串，继续处理剩余的输入字符串。

至于 `secondWord` 函数，它先取出一个列表的 `tail` 部分，得出一个新列表。再取出新列表的 `head` 部分，等同于取出一个列表的第二个元素。

[译注：书本的这个程序弱爆了，以下是 `dlts` 的一个更直观的版本，它使用 `filter` 来过滤输入，只保留符合格式的输入，而不是使用复杂且难看的显式递归和守卫来进行过滤：

```
-- file: ch04/dlts2.hs

import Data.List (isPrefixOf)

dlts2 :: String -> [String]
dlts2 = map (head . tail . words) . filter ("#define DLT_" `isPrefixOf`) . lines
```

]

编写可读代码的提示

目前为止，我们知道 Haskell 有两个非常诱人的特性：尾递归和匿名函数。但是，这两个特性通常并不被使用。

 v: latest ▾

对列表的处理操作一般可以通过组合库函数比如 `map`、`take` 和 `filter` 来进行。当然，熟悉这些库函数需要一定的时间，不过掌握这些函数之后，就可以使用它们写出更快更好更少 bug 的代码。

库函数比尾递归更好的原因很简单：尾递归和命令式语言里的 loop 有同样的问题——它们太通用（general）了。在一个尾递归里，你可以同时执行过滤（filtering）、映射（mapping）和其他别的动作。这强迫代码的读者（可能是你自己）必须弄懂整个递归函数的定义，才能理解这个函数到底做了些什么。与此相反，`map` 和其他很多列表函数，都只专注于做一件事。通过这些函数，我们可以很快理解某段代码到底做了什么，以及整个程序想表达什么意思，而不是将时间浪费在关注细节方面。

折叠（fold）操作处于（完全通用化的）尾递归和（只做一件事的）列表处理函数之间的中间地带。折叠也很值得我们花时间去好好理解，它的作用跟组合起 `map` 和 `filter` 函数差不多，但比起显式递归来说，折叠的行为要来得更有规律，而且更可控。一般来说，可以通过组合函数来解决的问题，就不要使用折叠。另一方面，如果问题用组合函数没办法解决，那么使用折叠要比使用显式递归要好。

另一方面，匿名函数通常会对代码的可读性造成影响。一般来说，匿名函数都可以用 `let` 或者 `where` 定义的局部函数来代替。而且带名字的局部函数可以达到一箭双雕的效果：它使得代码更具可读性，且函数名本身也达到了文档化的作用。

内存泄漏和严格求值

前面介绍的 `foldl` 函数并不是 Haskell 代码里唯一会造成内存泄漏的地方。

在这一节，我们使用 `foldl` 来展示非严格求值在什么情况下会造成问题，以及如何去解决这些问题。

通过 `seq` 函数避免内存泄漏

我们称非惰性求值的表达式为严格的（strict）。`foldl'` 就是左折叠的严格版本，它使用特殊的 `seq` 函数来绕过 Haskell 默认的非严格求值：

```
-- file: ch04/strictFoldl.hs

foldl' _ zero [] = zero
foldl' step zero (x:xs) =
    let new = step zero x
    in new `seq` foldl' step new xs
```

`seq` 函数的类型签名和之前看过的函数都有些不同，昭示了它的特殊身份：

```
ghci> :type seq
seq :: a -> t -> t
```

[译注：在 7.4.2 版本的 GHCi 里，`seq` 函数的类型签名不再使用 `t`，而是像其他函数一样，使用 `a` 和 `b`。

```
Prelude> :type seq
seq :: a -> b -> b
```

]

实际上，`seq` 函数的行为并没有那么神秘：它强迫（force）求值传入的第一个参数，然后返回它的第二个参数。

比如说，对于以下表达式：

```
foldl' (+) 1 (2:[])
```

它展开为：

```
let new = 1 + 2
in new `seq` foldl' (+) new []
```

它强迫 `new` 求值为 3，然后返回它的第二个参数：

```
foldl' (+) 3 []
```

 v: latest ▾

最终得到结果 3。

因为 `seq` 的存在，这个创建过程没有用到任何块。

seq 的用法

本节介绍一些更有效地使用 `seq` 的指导规则。

要正确地产生 `seq` 的作用，表达式中被求值的第一个必须是 `seq`：

```
-- 错误：因为表达式中第一个被求值的是 someFunc 而不是 seq
-- 所以 seq 的调用被隐藏在了 someFunc 调用之下
hiddenInside x y = someFunc (x `seq` y)

-- 错误：原因和上面一样
hiddenByLet x y z = let a = x `seq` someFunc y
                    in anotherFunc a z

-- 正确：seq 被第一个求值，并且 x 被强迫求值
onTheOutside x y = x `seq` someFunc y
```

为了严格求值多个值，可以连接起 `seq` 调用：

```
chained x y z = x `seq` y `seq` someFunc z
```

一个常见错误是，将 `seq` 用在没有关联的连个表达式上面：

```
badExpression step zero (x:xs) =
  seq (step zero x)
    (badExpression step (step zero x) xs)
```

`step zero x` 分别出现在 `seq` 的第一个参数和 `badExpression` 的表达式内，`seq` 只会对第一个 `step zero x` 求值，而它的结果并不会影响 `badExpression` 表达式内的 `step zero x`。正确的用法应该是一个 `let` 结果保存起 `step zero x` 表达式，然后将它分别传给 `seq` 和 `badExpression`，做法可以参考前面的 `foldl'` 的定义。

`seq` 在遇到像数字这样的值时，它会对值进行求值，但是，一旦 `seq` 碰到构造器，比如 `(:)` 或者 `(,)`，那么 `seq` 的求值就会停止。举个例子，如果将 `(1+2):[]` 传给 `seq` 作为它的第一个参数，那么 `seq` 不会对这个表达式进行求值；相反，如果将 `1` 传给 `seq` 作为第一个参数，那么它会被求值为 `1`。

[译注：

原文说，对于 `(1+2):[]` 这样的表达式，`seq` 在求值 `(1+2)` 之后，碰到 `:`，然后停止求值。但是根据原文网站上的评论者测试，`seq` 并不会对 `(1+2)` 求值，而是在碰到 `(1+2):[]` 时就直接停止求值。

这一表现可能的原因如下：虽然 `:` 是中序操作符，但它实际上只是函数 `(:)`，而 Haskell 的函数总是前序的，因此 `(1+2):[]` 实际上应该表示为 `(:)(1+2) []`，所以原文说“`seq` 在碰到构造器时就会停止求值”这一描述并没有出错，只是给的例子出了问题。

因为以上原因，这里对原文进行了修改。

]

如果有需要的话，也可以绕过这些限制：

```
strictPair (a,b) = a `seq` b `seq` (a,b)

strictList (x:xs) = x `seq` x : strictList xs
strictList []     = []
```

`seq` 的使用并不是无成本的，知道这一点很重要：它需要在运行时检查输入值是否已经被求值。必须谨慎使用 `seq`。比如说，上面定义的 `strictPair`，尽管它能顺利对元组进行强制求值，但它在求值元组所需的计算量上，加上了一次模式匹配、两次 `seq` 调用和一次构造新元组的计算量。如果我们检测这个函数的性能的话，就会发现它降低了程序的处理速度。

 v: latest ▼

即使不考虑性能的问题，seq 也不是处理内存泄漏的万能药。可以进行非严格求值，但并不意味着非用它不可。对 seq 的不小心使用可能对内存泄漏并没有帮助，在更糟糕的情况下，它还会造成新的内存泄漏。

第二十五章会介绍关于性能和优化的内容，到时会说明跟多 seq 的用法和细节。

讨论

0条评论

Real World Haskell 中文版

1 登录

推荐

推文

分享

最新发布



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

来做第一个留言的人吧!

在 REAL WORLD HASKLL 中文版 上还有

Real World Haskell 中文版

2条评论 • 6年前

 yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地

Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

 forlice — ...

第八章：高效文件处理、正则表达式、文件名匹配 — Real World Haskell 中文版

1条评论 • 4年前

 Yutong Zhang —

第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

 Wengel An —

订阅

在您的网站上使用 Disqus添加 Disqus添加

Disqus 隐私政策隐私政策隐私