

第三章：Defining Types, Streamlining Functions

定义新的数据类型

尽管列表和元组都非常有用，但是，定义新的数据类型也是一种常见的需求，这种能力使得我们可以为程序中的值添加结构。

而且比起使用元组，对一簇相关的值赋予一个名字和一个独一无二的类型显得更有用一些。

定义新的数据类型也提升了代码的安全性：Haskell 不会允许我们混用两个结构相同但类型不同的值。

本章将以一个在线书店为例子，展示如何去进行类型定义。

使用 `data` 关键字可以定义新的数据类型：

```
-- file: ch03/BookStore.hs
data BookInfo = Book Int String [String]
    deriving (Show)
```

跟在 `data` 关键字之后的 `BookInfo` 就是新类型的名字，我们称 `BookInfo` 为类型构造器。类型构造器用于指代（refer）类型。正如前面提到过的，类型名字的首字母必须大写，因此，类型构造器的首字母也必须大写。

接下来的 `Book` 是值构造器（有时候也称为数据构造器）的名字。类型的值就是由值构造器创建的。值构造器名字的首字母也必须大写。

在 `Book` 之后的 `Int`，`String` 和 `[String]` 是类型的组成部分。组成部分的作用，和面向对象语言的类中的域作用一致：它是一个储存值的槽。（为了方便起见，我们通常也将组成部分称为域。）

在这个例子中，`Int` 表示一本书的 ID，而 `String` 表示书名，而 `[String]` 则代表作者。

`BookInfo` 类型包含的成分和一个 `(Int, String, [String])` 类型的三元组一样，它们唯一不相同的是类型。[译注：这里指的是整个值的类型，不是成分的类型。]我们不能混用结构相同但类型不同的值。

举个例子，以下的 `MagzineInfo` 类型的成分和 `BookInfo` 一模一样，但 Haskell 会将它们作为不同的类型来区别对待，因为它们的类型构造器和值构造器并不相同：

```
-- file: ch03/BookStore.hs
data MagzineInfo = Magzine Int String [String]
    deriving (Show)
```

可以将值构造器看作是一个函数——它创建并返回某个类型值。在这个书店的例子里，我们将 `Int`、`String` 和 `[String]` 三个类型的值应用到 `Book`，从而创建一个 `BookInfo` 类型的值：

```
-- file: ch03/BookStore.hs
myInfo = Book 9780135072455 "Algebra of Programming"
    ["Richard Bird", "Oege de Moor"]
```

定义类型的工作完成之后，可以到 `ghci` 里载入并测试这些新类型：

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

再看看前面在文件里定义的 `myInfo` 变量：

```
*Main> myInfo
Book 494539463 "Algebra of Programming" ["Richard Bird", "Oege de Moor"]
```

在 `ghci` 里面当然也可以创建新的 `BookInfo` 值：

 v: latest ▾

```
*Main> Book 0 "The Book of Imaginary Beings" ["Jorge Luis Borges"]
Book 0 "The Book of Imaginary Beings" ["Jorge Luis Borges"]
```

可以用 `:type` 命令来查看表达式的值：

```
*Main> :type Book 1 "Cosmicomics" ["Italo Calvino"]
Book 1 "Cosmicomics" ["Italo Calvino"] :: BookInfo
```

请记住，在 `ghci` 里定义变量的语法和在源码文件里定义变量的语法并不相同。在 `ghci` 里，变量通过 `let` 定义：

```
*Main> let cities = Book 173 "Use of Weapons" ["Iain M. Banks"]
```

使用 `:info` 命令可以查看更多关于给定表达式的信息：

```
*Main> :info BookInfo
data BookInfo = Book Int String [String]
  -- Defined at BookStore.hs:2:6
instance Show BookInfo -- Defined at BookStore.hs:3:27
```

使用 `:type` 命令，可以查看值构造器 `Book` 的类型签名，了解它是如何创建出 `BookInfo` 类型的值的：

```
*Main> :type Book
Book :: Int -> String -> [String] -> BookInfo
```

类型构造器和值构造器的命名

在前面介绍 `BookInfo` 类型的时候，我们专门为类型构造器和值构造器设置了不同的名字（`BookInfo` 和 `Book`），这样区分起来比较容易。

在 Haskell 里，类型的名字（类型构造器）和值构造器的名字是相互独立的。类型构造器只能出现在类型的定义，或者类型签名当中。而值构造器只能出现在实际的代码中。因为存在这种差别，给类型构造器和值构造器赋予一个相同的名字实际上并不会产生任何问题。

以下是这种用法的一个例子：

```
-- file: ch03/BookStore.hs
-- 稍后就会介绍 CustomerID 的定义

data BookReview = BookReview BookInfo CustomerID String
```

以上代码定义了一个 `BookReview` 类型，并且它的值构造器的名字也同样是 `BookReview`。

类型别名

可以使用 **类型别名**，来为一个已存在的类型设置一个更具描述性的名字。

比如说，在前面 `BookReview` 类型的定义里，并没有说明 `String` 成分是用来干什么用的，通过类型别名，可以解决这个问题：

```
-- file: ch03/BookStore.hs
type CustomerID = Int
type ReviewBody = String

data BetterReview = BetterReview BookInfo CustomerID ReviewBody
```

`type` 关键字用于设置**类型别名**，其中新的类型名字放在 `=` 号的左边，而已有的类型名字放在 `=` 号的右边。这两个名字都标识同一个类型，因此，类型别名完全是为了提高可读性而存在的。

类型别名也可以用来为啰嗦的类型设置一个更短的名字：

```
-- file: ch03/BookStore.hs
type BookRecord = (BookInfo, BookReview)
```

 v: latest ▾

需要注意的是, 类型别名只是为已有类型提供了一个新名字, 创建值的工作还是由原来类型的值构造器进行。[注: 如果你熟悉 C 或者 C++, 可以将 Haskell 的类型别名看作是 typedef。]

代数数据类型

`Bool` 类型是代数数据类型 (algebraic data type) 的最简单也是最常见的例子。一个代数类型可以有多个值构造器:

```
-- file: ch03/Bool.hs
data Bool = False | True
```

上面代码定义的 `Bool` 类型拥有两个值构造器, 一个是 `True`, 另一个是 `False`。每个值构造器使用 `|` 符号分割, 读作“或者”——以 `Bool` 类型为例子, 我们可以说, `Bool` 类型由 `True` 值或者 `False` 值构成。

当一个类型拥有一个以上的值构造器时, 这些值构造器通常被称为“备选” (alternatives) 或“分支” (case)。同一类型的所有备选, 创建出的值的类型都是相同的。

代数数据类型的各个值构造器都可以接受任意个数的参数。[译注: 不同备选之间接受的参数个数不必相同, 参数的类型也可以不一样。]以下是一个账单数据的例子:

```
-- file: ch03/BookStore.hs
type CardHolder = String
type CardNumber = String
type Address = [String]
data BillingInfo = CreditCard CardNumber CardHolder Address
                | CashOnDelivery
                | Invoice CustomerID
                deriving (Show)
```

这个程序提供了三种付款的方式。如果使用信用卡付款, 就要使用 `CreditCard` 作为值构造器, 并输入信用卡卡号、信用卡持有人和地址作为参数。如果即时支付现金, 就不用接受任何参数。最后, 可以通过货到付款的方式来收款, 在这种情况下, 只需要填写客户的 ID 就可以了。

当使用值构造器来创建 `BillingInfo` 类型的值时, 必须提供这个值构造器所需的参数:

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

*Main> :type CreditCard
CreditCard :: CardNumber -> CardHolder -> Address -> BillingInfo

*Main> CreditCard "2901650221064486" "Thomas Gradgrind" ["Dickens", "England"]
CreditCard "2901650221064486" "Thomas Gradgrind" ["Dickens", "England"]

*Main> :type it
it :: BillingInfo
```

如果输入参数的类型不对或者数量不对, 那么引发一个错误:

```
*Main> Invoice

<interactive>:7:1:
  No instance for (Show (CustomerID -> BillingInfo))
    arising from a use of `print'
  Possible fix:
    add an instance declaration for (Show (CustomerID -> BillingInfo))
  In a stmt of an interactive GHCi command: print it
```

`ghci` 抱怨我们没有给 `Invoice` 值构造器足够的参数。

[译注: 原文这里的代码示例有错, 译文已改正。]

 v: latest ▾

什么情况下该用元组, 而什么情况下又该用代数数据类型?

元组和自定域代数数据类型有一些相似的地方。比如说，可以使用一个 `(Int, String, [String])` 类型的元组来代替 `BookInfo` 类型：

```
*Main> Book 2 "The Wealth of Networks" ["Yochai Benkler"]
Book 2 "The Wealth of Networks" ["Yochai Benkler"]

*Main> (2, "The Wealth of Networks", ["Yochai Benkler"])
(2, "The Wealth of Networks", ["Yochai Benkler"])
```

代数数据类型使得我们可以在结构相同但类型不同的数据之间进行区分。然而，对于元组来说，只要元素的结构和类型都一致，那么元组的类型就是相同的：

```
-- file: ch03/Distinction.hs
a = ("Porpoise", "Grey")
b = ("Table", "Oak")
```

其中 `a` 和 `b` 的类型相同：

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

*Main> :type a
a :: ([Char], [Char])

*Main> :type b
b :: ([Char], [Char])
```

对于两个不同的代数数据类型来说，即使值构造器成分的结构和类型都相同，它们也是不同的类型：

```
-- file: ch03/Distinction.hs
data Cetacean = Cetacean String String
data Furniture = Furniture String String

c = Cetacean "Porpoise" "Grey"
d = Furniture "Table" "Oak"
```

其中 `c` 和 `d` 的类型并不相同：

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

*Main> :type c
c :: Cetacean

*Main> :type d
d :: Furniture
```

以下是一个更细致的例子，它用两种不同的方式表示二维向量：

```
-- file: ch03/AlgebraicVector.hs
-- x and y coordinates or lengths.
data Cartesian2D = Cartesian2D Double Double
    deriving (Eq, Show)

-- Angle and distance (magnitude).
data Polar2D = Polar2D Double Double
    deriving (Eq, Show)
```

`Cartesian2D` 和 `Polar2D` 两种类型的成分都是 `Double` 类型，但是，这些成分表达的是不同的意思。因为 `Cartesian2D` 和 `Polar2D` 是不同的类型，因此 Haskell 不会允许混淆使用这两种类型：

```
Prelude> :load AlgebraicVector.hs
[1 of 1] Compiling Main          ( AlgebraicVector.hs, interpreted )
```

```
Ok, modules loaded: Main.
*Main> Cartesian2D (sqrt 2) (sqrt 2) == Polar2D (pi / 4) 2

<interactive>:3:34:
    Couldn't match expected type `Cartesian2D'
with actual type `Polar2D'
    In the return type of a call of `Polar2D'
    In the second argument of `(==)', namely `Polar2D (pi / 4) 2'
    In the expression:
        Cartesian2D (sqrt 2) (sqrt 2) == Polar2D (pi / 4) 2
```

错误信息显示, `(==)` 操作符只接受类型相同的值作为它的参数, 在类型签名里也可以看出这一点:

```
*Main> :type (==)
(==) :: Eq a => a -> a -> Bool
```

另一方面, 如果使用类型为 `(Double, Double)` 的元组来表示二维向量的两种表示方式, 那么我们就有麻烦了:

```
Prelude> -- 第一个元组使用 Cartesian 表示, 第二个元组使用 Polar 表示
Prelude> (1, 2) == (1, 2)
True
```

类型系统不会察觉到, 我们正错误地对比两种不同表示方式的值, 因为对两个类型相同的元组进行对比是完全合法的!

关于该使用元组还是该使用代数数据类型, 没有一劳永逸的办法。但是, 有一个经验法则可以参考: 如果程序大量使用复合数据, 那么使用 `data` 进行类型自定义对于类型安全和可读性都有好处。而对于小规模的内部应用, 那么通常使用元组就足够了。

其他语言里类似代数数据类型的东西

代数数据类型为描述数据类型提供了一种单一且强大的方式。很多其他语言, 要达到相当于代数数据类型的表达能力, 需要同时使用多种特性。

以下是一些 C 和 C++ 方面的例子, 说明怎样在这些语言里, 怎么样实现类似于代数数据类型的功能。

结构

当只有一个值构造器时, 代数数据类型和元组很相似: 它将一系列相关的值打包成一个复合值。这种做法相当于 C 和 C++ 里的 `struct`, 而代数数据类型的成分则相当于 `struct` 里的域。

以下是一个 C 结构, 它等同于我们前面定义的 `BookInfo` 类型:

```
struct book_info {
    int id;
    char *name;
    char **authors;
};
```

目前来说, C 结构和 Haskell 的代数数据类型最大的差别是, 代数数据类型的成分是匿名且按位置排序的:

```
--file: ch03/BookStore.hs
data BookInfo = Book Int String [String]
               deriving (Show)
```

按位置排序指的是, 对成分的访问是通过位置来实行的, 而不是像 C 那样, 通过名字: 比如 `book_info->id`。

稍后的“模式匹配”小节会介绍如何访代数数据类型里的成分。在“记录”一节会介绍定义数据的新语法, 通过这种语法, 可以像 C 结构那样, 使用名字来访问相应的成分。

枚举

 v: latest ▾

C 和 C++ 里的 `enum` 通常用于表示一系列符号值排列。代数数据类型里面也有相似的东西, 一般称之为枚举类型。

以下是一个 `enum` 例子：

```
enum royg biv {
  red,
  orange,
  yellow,
  green,
  blue,
  indigo,
  violet,
};
```

以下是等价的 Haskell 代码：

```
-- file: ch03/Roygbiv.hs
data Roygbiv = Red
             | Orange
             | Yellow
             | Green
             | Blue
             | Indigo
             | Violet
             deriving (Eq, Show)
```

在 ghci 里面测试：

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

*Main> :type Yellow
Yellow :: Roygbiv

*Main> :type Red
Red :: Roygbiv

*Main> Red == Yellow
False

*Main> Green == Green
True
```

`enum` 的问题是，它使用整数值去代表元素：在一些接受 `enum` 的场景里，可以将整数传进去，C 编译器会自动进行类型转换。同样，在使用整数的场景里，也可以将一个 `enum` 元素传进去。这种用法可能会造成一些令人不爽的 bug。

另一方面，在 Haskell 里就没有这样的问题。比如说，不可能使用 `Roygbiv` 里的某个值来代替 `Int` 值[译注：因为枚举类型的每个元素都由一个唯一的值构造器生成，而不是使用整数表示。]：

```
*Main> take 3 "foobar"
"foo"

*Main> take Red "foobar"

<interactive>:9:6:
    Couldn't match expected type `Int' with actual type `Roygbiv'
    In the first argument of `take', namely `Red'
    In the expression: take Red "foobar"
    In an equation for `it': it = take Red "foobar"
```

联合

如果一个代数数据类型有多个备选，那么可以将它看作是 C 或 C++ 里的 `union`。

 v: latest ▾

以上两者的一个主要区别是，`union` 并不告诉用户，当前使用的是哪一个备选，`union` 的使用者必须自己记录这方面的信息（通常使用一个额外的域来保存），这意味着，如果搞错了备选的信息，那么对 `union` 的使用就会出错。

以下是一个 `union` 例子：

```
enum shape_type {
    shape_circle,
    shape_poly,
};

struct circle {
    struct vector centre;
    float radius;
};

struct poly {
    size_t num_vertices;
    struct vector *vertices;
};

struct shape
{
    enum shape_type type;
    union {
        struct circle circle;
        struct poly poly;
    } shape;
};
```

在上面的代码里，`shape` 域的值可以是一个 `circle` 结构，也可以是一个 `poly` 结构。`shape_type` 用于记录目前 `shape` 正在使用的结构类型。

另一方面，Haskell 版本不仅简单，而且更为安全：

```
-- file: ch03/ShapeUnion.hs
type Vector = (Double, Double)

data Shape = Circle Vector Double
           | Poly [Vector]
           deriving (Show)
```

[译注：原文的代码少了 `deriving (Show)` 一行，在 `ghci` 测试时会出错。]

注意，我们不必像 C 语言那样，使用 `shape_type` 域来手动记录 `Shape` 类型的值是由 `Circle` 构造器生成的，还是由 `Poly` 构造器生成，Haskell 自己有能力弄清楚一点，它不会弄混两种不同的值。其中的原因，下一节《模式匹配》就会讲到。

[译注：原文这里将 `Poly` 写成了 `Square`。]

模式匹配

前面的章节介绍了代数数据类型的定义方法，本节将说明怎样去处理这些类型的值。

对于某个类型的值来说，应该可以做到以下两点：

- 如果这个类型有一个以上的值构造器，那么应该可以知道，这个值是由哪个构造器创建的。
- 如果一个值构造器包含不同的成分，那么应该有能力提取这些成分。

对于以上两个问题，Haskell 有一个简单且有效的解决方式，那就是 **类型匹配**。

模式匹配允许我们查看值的内部，并将值所包含的数据绑定到变量上。以下是一个对 `Bool` 类型值进行模式匹配的例子，它的作用和 `not` 函数一样：

```
-- file: myNot.hs
myNot True = False
myNot False = True
```

[译注：原文的文件名为 `add.hs`，这里修改成 `myNot.hs`，和函数名保持一致。]

初看上去，代码似乎同时定义了两个 `myNot` 函数，但实际情况并不是这样 —— Haskell 允许将函数定义为一系列等式：`myNot` 的两个等式分别定义了函数对于输入参数在不同模式之下的行为。对于每行等式，模式定义放在函数名之后，`=` 符号之前。

为了理解模式匹配是如何工作的，来研究一下 `myNot False` 是如何执行的：首先调用 `myNot`，Haskell 运行时检查输入参数 `False` 是否和第一个模式的值构造器匹配 —— 答案是不匹配，于是它继续尝试匹配第二个模式 —— 这次匹配成功了，于是第二个等式右边的值被作为结果返回。

以下是一个复杂一点的例子，这个函数计算出列表所有元素之和：

```
-- file:: ch03/sumList.hs
sumList (x:xs) = x + sumList xs
sumList []    = 0
```

[译注：原文代码的文件名为 `add.hs` 这里改为 `sumList.hs`，和函数名保持一致。]

需要说明的一点是，在 Haskell 里，列表 `[1, 2]` 实际上只是 `(1:(2:[]))` 的一种简单的表示方式，其中 `(:)` 用于构造列表：

```
Prelude> []
[]

Prelude> 1:[]
[1]

Prelude> 1:2:[]
[1,2]
```

因此，当需要对一个列表进行匹配时，也可以使用 `(:)` 操作符，只不过这次不是用来构造列表，而是用来分解列表。

作为例子，考虑求值 `sumList [1, 2]` 时会发生什么：首先，`[1, 2]` 尝试对第一个等式的模式 `(x:xs)` 进行匹配，结果是模式匹配成功，并将 `x` 绑定为 `1`，`xs` 绑定为 `[2]`。

计算进行到这一步，表达式就变成了 `1 + (sumList [2])`，于是递归调用 `sumList`，对 `[2]` 进行模式匹配。

这一次也是在第一个等式匹配成功，变量 `x` 被绑定为 `2`，而 `xs` 被绑定为 `[]`。表达式变为 `1 + (2 + sumList [])`。

再次递归调用 `sumList`，输入为 `[]`，这一次，第二个等式的 `[]` 模式匹配成功，返回 `0`，整个表达式为 `1 + (2 + (0))`，计算结果为 `3`。

最后要说的一点是，标准函数库里已经有 `sum` 函数，它和我们定义的 `sumList` 一样，都可以用于计算表元素的和：

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

*Main> sumList [1, 2]
3

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

组成和解构

让我们稍微慢下探索新特性的脚步，花些时间，了解构造一个值、和对这个值进行模式匹配之间的关系。

我们通过应用值构造器来构建值：表达式 `Book 9 "Close Calls" ["John Long"]` 应用 `Book` 构造器到值 `9`、`"Close Calls"` 和 `["John Long"]` 上面，从而产生一个新的 `BookInfo` 类型的值。

另一方面，当对 `Book` 构造器进行模式匹配时，我们逆转 (reverse) 它的构造过程：首先，检查这个值是否由 `Book` 构造器生成 —— 如果是的话，那么就对这个值进行探查 (inspect)，并取出创建这个值时，提供给构造器的各个值。

考虑一下表达式 `Book 9 "Close Calls" ["John Long"]` 对模式 `(Book id name authors)` 的匹配是如何进行的：

因为值的构造器和模式里的构造器相同，因此匹配成功。

变量 `id` 被绑定为 `9`。

变量 `name` 被绑定为 `Close Calls`。

 v: latest ▾

变量 `authors` 被绑定为 `["John Long"]`。

因为模式匹配的过程就像是逆转一个值的构造（`construction`）过程，因此它有时候也被称为解构（`deconstruction`）。

[译注：上一节的《联合》小节里提到，Haskell 有办法分辨同一类型由不同值构造器创建的值，说的就是模式匹配。

比如 `Circle ...` 和 `Poly ...` 两个表达式创建的都是 `Shape` 类型的值，但第一个表达式只有在匹配 `(Circle vector double)` 模式时才会成功，而第二个表达式只有在 `(Poly vectors)` 时才会成功。这就是它们不会被混淆的原因。]

更进一步

对元组进行模式匹配的语法，和构造元组的语法很相似。

以下是一个可以返回三元组中最后一个元素的函数：

```
-- file: ch03/third.hs
third (a, b, c) = c
```

[译注：原文的源码文件名为 `Tuple.hs`，这里改为 `third.hs`，和函数的名字保持一致。]

在 `ghci` 里测试这个函数：

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

*Main> third (1, 2, 3)
3
```

模式匹配的“深度”并没有限制。以下模式会同时对元组和元组里的列表进行匹配：

```
-- file: ch03/complicated.hs
complicated (True, a, x:xs, 5) = (a, xs)
```

[译注：原文的源码文件名为 `Tuple.hs`，这里改为 `complicated.hs`，和函数的名字保持一致。]

在 `ghci` 里测试这个函数：

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

*Main> complicated (True, 1, [1, 2, 3], 5)
(1, [2, 3])
```

对于出现在模式里的字面（`literal`）值（比如前面元组例子里的 `True` 和 `5`），输入里的各个值必须和这些字面值相等，匹配才有可能成功。以下代码显示，因为输入元组和模式的第一个字面值 `True` 不匹配，所以匹配失败了：

```
*Main> complicated (False, 1, [1, 2, 3], 5)
*** Exception: complicated.hs:2:1-40: Non-exhaustive patterns in function complicated
```

这个例子也显示了，如果所有给定等式的模式都匹配失败，那么返回一个运行时错误。

对代数数据类型的匹配，可以通过这个类型的值构造器来进行。拿之前我们定义的 `BookInfo` 类型为例子，对它的模式匹配可以使用它的 `Book` 构造器来进行：

```
-- file: ch03/BookStore.hs
bookID      (Book id title authors) = id
bookTitle   (Book id title authors) = title
bookAuthors (Book id title authors) = authors
```

 v: latest ▾

在 `ghci` 里试试：

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

*Main> let book = (Book 3 "Probability Theory" ["E. T. H. Jaynes"])

*Main> bookID book
3

*Main> bookTitle book
"Probability Theory"

*Main> bookAuthors book
["E. T. H. Jaynes"]
```

字面值的比对规则对于列表和值构造器的匹配也适用：`(3:xs)` 模式只匹配那些不为空，并且第一个元素为 `3` 的列表；而 `(Book 3 title authors)` 只匹配 ID 值为 `3` 的那本书。

模式匹配中的变量命名

当你阅读那些进行模式匹配的函数时，经常会发现像是 `(x:xs)` 或是 `(d:ds)` 这种类型的名字。这是一个流行的命名规则，其中的 `s` 表示“元素的复数”。以 `(x:xs)` 来说，它用 `x` 来表示列表的第一个元素，剩余的列表元素则用 `xs` 表示。

通配符模式匹配

如果在匹配模式中我们不在乎某个值的类型，那么可以用下划线字符 “`_`” 作为符号来进行标识，它也被称为*通配符*。它的用法如下。

```
-- file: ch03/BookStore.hs
nicerID    (Book id _ _ ) = id
nicerTitle (Book _ title _ ) = title
nicerAuthors (Book _ _ authors) = authors
```

于是，我们将之前介绍过的访问器函数改得更加简明了。现在能很清晰的看出各个函数究竟使用到了哪些元素。

在模式匹配里，通配符的作用和变量类似，但是它并不会绑定成一个新的变量。就像上面的例子展示的那样，在一个模式匹配里可以使用一个或多个通配符。

使用通配符还有另一个好处。如果我们在一个匹配模式中引入了一个变量，但没有在函数体中用到它的话，Haskell 编译器会发出一个警告。定义一个变量但忘了使用通常意味着存在潜在的 bug，因此这是个有用的功能。假如我们不准备使用一个变量，那就不要用变量，而是用通配符，这样编译器就不会报错。

穷举匹配模式和通配符

在给一个类型写一组匹配模式时，很重要的一点就是一定要涵盖构造器的所有可能情况。例如，如果我们需要探查一个列表，就应该写一个匹配非空构造器 `(:)` 的方程和一个匹配空构造器 `[]` 的方程。

假如我们没有涵盖所有情况会发生什么呢。下面，我们故意漏写对 `[]` 构造器的检查。

```
-- file: ch03/BadPattern.hs
badExample (x:xs) = x + badExample xs
```

如果我们将其作用于一个不能匹配的值，运行时就会报错：我们的软件有 bug！

```
ghci> badExample []
*** Exception: BadPattern.hs:4:0-36: Non-exhaustive patterns in function badExample
```

在上面的例子中，函数定义时的方程里没有一个可以匹配 `[]` 这个值。

如果在某些情况下，我们并不在乎某些特定的构造器，我们就可以用通配符匹配模式来定义一个默认的行为。

```
-- file: ch03/BadPattern.hs
goodExample (x:xs) = x + goodExample xs
goodExample []    = 0
```

上面例子中的通配符可以匹配 `[]` 构造器，因此应用这个函数不会导致程序崩溃。

```
ghci> goodExample []
0
ghci> goodExample [1,2]
3
```

记录语法

给一个数据类型的每个成分写访问器函数是令人感觉重复而且乏味的事情。

```
-- file: ch03/BookStore.hs
nicerID    (Book id _ _ ) = id
nicerTitle (Book _ title _ ) = title
nicerAuthors (Book _ _ authors) = authors
```

我们把这种代码叫做“样板代码（boilerplate code）”：尽管是必需的，但是又长又烦。Haskell 程序员不喜欢样板代码。幸运的是，语言的设计者提供了避免这个问题的方法：我们在定义一种数据类型的时候，就可以定义好每个成分的访问器。（逗号的位置是一个风格问题，如果你喜欢的话，也可以把它放在每行的最后。）

```
-- file: ch03/BookStore.hs
data Customer = Customer {
  customerID    :: CustomerID
  , customerName :: String
  , customerAddress :: Address
} deriving (Show)
```

以上代码和下面这段我们更熟悉的代码的意义几乎是完全一致的。

```
-- file: ch03/AltCustomer.hs
data Customer = Customer Int String [String]
  deriving (Show)

customerID :: Customer -> Int
customerID (Customer id _ _) = id

customerName :: Customer -> String
customerName (Customer _ name _) = name

customerAddress :: Customer -> [String]
customerAddress (Customer _ _ address) = address
```

Haskell 会使用我们在定义类型的每个字段时的命名，相应生成与该命名相同的该字段的访问器函数。

```
ghci> :type customerID
customerID :: Customer -> CustomerID
```

我们仍然可以如往常一样使用应用语法来新建一个此类型的值。

```
-- file: ch03/BookStore.hs
customer1 = Customer 271828 "J.R. Hacker"
  ["255 Syntax Ct",
   "Milpitas, CA 95134",
   "USA"]
```

记录语法还新增了一种更详细的标识法来新建一个值。这种标识法通常都会提升代码的可读性。

 v: latest ▾

```
-- file: ch03/BookStore.hs
customer2 = Customer {
    customerID = 271828
  , customerAddress = ["1048576 Disk Drive",
                      "Milpitas, CA 95134",
                      "USA"]
  , customerName = "Jane Q. Citizen"
}
```

如果使用这种形式，我们还可以调换字段列表的顺序。比如在上面的例子里，name 和 address 字段的顺序就被移动过，和定义类型时的顺序不一样了。

当我们使用记录语法来定义类型时，还会影响到该类型的打印格式。

```
ghci> customer1
Customer {customerID = 271828, customerName = "J.R. Hacker", customerAddress = ["255 Syntax Ct", "Milpitas, CA 95134", "USA"]}
```

让我们打印一个 BookInfo 类型的值来做比较；这是没有使用记录语法时的打印格式。

```
ghci> cities
Book 173 "Use of Weapons" ["Iain M. Banks"]
```

我们在使用记录语法的时候“免费”得到的访问器函数，实际上都是普通的 Haskell 函数。

```
ghci> :type customerName
customerName :: Customer -> String
ghci> customerName customer1
"J.R. Hacker"
```

标准库里的 System.Time 模块就是一个使用记录语法的好例子。例如其中定义了这样一个类型：

```
data CalendarTime = CalendarTime {
    ctYear      :: Int,
    ctMonth     :: Month,
    ctDay, ctHour, ctMin, ctSec :: Int,
    ctPicosec   :: Integer,
    ctWDay      :: Day,
    ctYDay      :: Int,
    ctTZName    :: String,
    ctTZ        :: Int,
    ctIsDST     :: Bool
}
```

假如没有记录语法，从一个如此复杂的类型中抽取某个字段将是一件非常痛苦的事情。这种标识法使我们在使用大型结构的过程中更方便了。

参数化类型

我们曾不止一次地提到列表类型是多态的：列表中的元素可以是任何类型。我们也可以给自定义的类型添加多态性。只要在类型定义中使用类型变量就可以做到这一点。Prelude 中定义了一种叫做 Maybe 的类型：它用来表示这样一种值——既可以有值也可能空缺，比如数据库中某行的某字段就可能为空。

```
-- file: ch03/Nullable.hs
data Maybe a = Just a
              | Nothing
译注：Maybe, Just, Nothing 都是 Prelude 中已经定义好的类型
这段代码是不能在 ghci 里面执行的，它简单地展示了标准库是怎么定义 Maybe 这种类型的
```

这里的变量 a 不是普通的变量：它是一个类型变量。它意味着 Maybe 类型使用另一种类型作为它的参数。从而使得 Maybe 可以作用于任何类型的值。

 v: latest ▼

```
-- file: ch03/Nullable.hs
someBool = Just True
someString = Just "something"
```

和往常一样，我们可以在 **ghci** 里试着用一下这种类型。

```
ghci> Just 1.5
Just 1.5
ghci> Nothing
Nothing
ghci> :type Just "invisible bike"
Just "invisible bike" :: Maybe [Char]
```

Maybe 是一个多态，或者称作泛型的类型。我们向 **Maybe** 的类型构造器传入某种类型作为参数，例如 `Maybe Int` 或 `Maybe [Bool]`。如我们所希望的那样，这些都是不同的类型（译注：可能省略了“但是都可以成功传入作为参数”）。

我们可以嵌套使用参数化的类型，但要记得使用括号标识嵌套的顺序，以便 **Haskell** 编译器知道如何解析这样的表达式。

```
-- file: ch03/Nullable.hs
wrapped = Just (Just "wrapped")
```

再补充说明一下，如果和其它更常见的语言做个类比，参数化类型就相当于 **C++** 中的模板（**template**），和 **Java** 中的泛型（**generics**）。请注意这仅仅是个大概的比喻。这些语言都是在被发明之后很久再加上模板和泛型的，因此在使用时会感到有些别扭。**Haskell** 则是从诞生之日起就有了参数化类型，因此更简单易用。

递归类型

列表这种常见的类型就是**递归**的：即它用自己来定义自己。为了深入了解其中的含义，让我们自己来设计一个与列表相仿的类型。我们将用 **Cons** 替换 **(:)** 构造器，用 **Nil** 替换 **[]** 构造器。

```
-- file: ch03/ListADT.hs
data List a = Cons a (List a)
            | Nil
            deriving (Show)
```

List a 在 **=** 符号的左右两侧都有出现，我们可以说该类型的定义引用了它自己。当我们使用 **Cons** 构造器创建一个值的时候，我们必须提供一个 **a** 的值作为参数一，以及一个 **List a** 类型的值作为参数二。接下来我们看一个实例。

我们能创建的 **List a** 类型的最简单的值就是 **Nil**。请将上面的代码保存为一个文件，然后打开 **ghci** 并加载它。

```
ghci> Nil
Nil
```

由于 **Nil** 是一个 **List a** 类型（译注：原文是 **List** 类型，可能是漏写了 **a**），因此我们可以将它作为 **Cons** 的第二个参数。

```
ghci> Cons 0 Nil
Cons 0 Nil
```

然后 **Cons 0 Nil** 也是一个 **List a** 类型，我们也可以将它作为 **Cons** 的第二个参数。

```
ghci> Cons 1 it
Cons 1 (Cons 0 Nil)
ghci> Cons 2 it
Cons 2 (Cons 1 (Cons 0 Nil))
ghci> Cons 3 it
Cons 3 (Cons 2 (Cons 1 (Cons 0 Nil)))
```

我们可以一直这样写下去，得到一个很长的 **Cons** 链，其中每个子链的末位元素都是一个 **Nil**。

 v: latest ▾

Tip

List 可以被当作是 list 吗？

让我们来简单的证明一下 List a 类型和内置的 list 类型 [a] 拥有相同的构型。让我们设计一个函数能够接受任何一个 [a] 类型的值作为输入参数，并返回 List a 类型的一个值。

```
-- file: ch03/ListADT.hs
fromList (x:xs) = Cons x (fromList xs)
fromList []    = Nil
```

通过查看上述实现，能清楚的看到它将每个 (:) 替换成 Cons，将每个 [] 替换成 Nil。这样就涵盖了内置 list 类型的全部构造器。因此我们可以说二者是同构的，它们有着相同的构型。

```
ghci> fromList "durian"
Cons 'd' (Cons 'u' (Cons 'r' (Cons 'i' (Cons 'a' (Cons 'n' Nil)))))
ghci> fromList [Just True, Nothing, Just False]
Cons (Just True) (Cons Nothing (Cons (Just False) Nil))
```

为了说明什么是递归类型，我们再来看第三个例子——定义一个二叉树类型。

```
-- file: ch03/Tree.hs
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving (Show)
```

二叉树是指这样一种节点：该节点有两个子节点，这两个子节点要么也是二叉树节点，要么是空节点。

这次我们将和另一种常见的语言进行比较来寻找灵感。以下是在 Java 中实现类似数据结构的类定义。

```
class Tree<A>
{
    A value;
    Tree<A> left;
    Tree<A> right;

    public Tree(A v, Tree<A> l, Tree<A> r)
    {
        value = v;
        left = l;
        right = r;
    }
}
```

稍有不同的是，Java 中使用特殊值 null 表示各种“没有值”，因此我们可以使用 null 来表示一个节点没有左子节点或没有右子节点。下面这个简单的函数能够构建一个有两个叶节点的树（叶节点这个词习惯上是指没有子节点的节点）。

```
class Example
{
    static Tree<String> simpleTree()
    {
        return new Tree<String>(
            "parent",
            new Tree<String>("left leaf", null, null),
            new Tree<String>("right leaf", null, null));
    }
}
```

Haskell 没有与 null 对应的概念。尽管我们可以使用 Maybe 达到类似的效果，但后果是模式匹配将变得十分臃肿。因此我们决定使用一个没有参数的 Empty 构造器。在上述 Tree 类型的 Java 实现中使用到 null 的地方，在 Haskell 中都改用 Empty。

```
-- file: ch03/Tree.hs
simpleTree = Node "parent" (Node "left child" Empty Empty)
              (Node "right child" Empty Empty)
```

 v: latest ▾

练习

1. 请给 `List` 类型写一个与 `fromList` 作用相反的函数：传入一个 `List a` 类型的值，返回一个 `[a]`。
2. 请仿造 Java 示例，定义一种只需要一个构造器的树类型。不要使用 `Empty` 构造器，而是用 `Maybe` 表示节点的子节点。

报告错误

当我们的代码中出现严重错误时可以调用 Haskell 提供的标准函数 `error :: String -> a`。我们将希望打印出来的错误信息作为一个字符串参数传入。而该函数的类型签名看上去有些特别：它是怎么做到的仅从一个字符串类型的值就生成任意类型 `a` 的返回值的呢？

由于它的结果是返回类型 `a`，因此无论我们在哪里调用它都能得到正确类型的返回值。然而，它并不像普通函数那样返回一个值，而是立即中止求值过程，并将我们提供的错误信息打印出来。

`mySecond` 函数返回输入列表参数的第二个元素，假如输入列表长度不够则失败。

```
-- file: ch03/MySecond.hs
mySecond :: [a] -> a

mySecond xs = if null (tail xs)
               then error "list too short"
               else head (tail xs)
```

和之前一样，我们来看看这个函数在 `ghci` 中的使用效果如何。

```
ghci> mySecond "xi"
'i'
ghci> mySecond [2]
*** Exception: list too short
ghci> head (mySecond [[9]])
*** Exception: list too short
```

注意上面的第三种情况，我们试图将调用 `mySecond` 的结果作为参数传入另一个函数。求值过程也同样中止了，并返回到 `ghci` 提示符。这就是使用 `error` 的最主要的问题：它并不允许调用者根据错误是可修复的还是严重到必须中止的来区别对待。

正如我们之前所看到的，模式匹配失败也会造成类似的不可修复错误。

```
ghci> mySecond []
*** Exception: Prelude.tail: empty list
```

让过程更可控的方法

我们可以使用 `Maybe` 类型来表示有可能出现错误的情况。

如果我们想指出某个操作可能会失败，可以使用 `Nothing` 构造器。反之则使用 `Just` 构造器将值包裹起来。

让我们看看如果返回 `Maybe` 类型的值而不是调用 `error`，这样会给 `mySecond` 函数带来怎样的变化。

```
-- file: ch03/MySecond.hs
safeSecond :: [a] -> Maybe a

safeSecond [] = Nothing
safeSecond xs = if null (tail xs)
                  then Nothing
                  else Just (head (tail xs))
```

当传入的列表太短时，我们将 `Nothing` 返回给调用者。然后由他们来决定接下来做什么，假如调用 `error` 的话则会强制程序崩溃。

```
ghci> safeSecond []
Nothing
ghci> safeSecond [1]
Nothing
ghci> safeSecond [1,2]
Just 2
```

 v: latest ▾


```
ghci> safeSecond [1,2,3]
Just 2
```

复习一下前面的章节，我们还可以使用模式匹配继续增强这个函数的可读性。

```
-- file: ch03/MySecond.hs
tidySecond :: [a] -> Maybe a

tidySecond (_,x:_) = Just x
tidySecond _       = Nothing
```

译注: `(_:x:_)` 相当于 `(_: (x:_))`，考虑到列表的元素只能是同一种类型
 假想第一个 `_` 是 `a` 类型，那么这个模式匹配的是 `(a:(a:[a, a, ...]))` 或 `(a:(a:[]))`
 即元素是 `a` 类型的值的一个列表，并且至少有 2 个元素
 那么如果第一个 `_` 匹配到了 `[]`，有没有可能使最终匹配到得列表只有一个元素呢？
`([]:(x:_))` 说明 `a` 是列表类型，那么 `x` 也必须是列表类型，`x` 至少是 `[]`
 而 `([]:([]:[])) -> ([]: [[]]) -> [[], []]`，还是 2 个元素

第一个模式仅仅匹配那些至少有两个元素的列表（因为它有两个列表构造器），并将列表的第二个元素的值绑定给变量 `x`。如果第一个模式匹配失败了，则匹配第二个模式。

引入局部变量

在函数体内部，我们可以在任何地方使用 `let` 表达式引入新的局部变量。请看下面这个简单的函数，它用来检查我们是否可以向顾客出借现金。我们需要确保剩余的保证金不少于 100 元的情况下，才能出借现金，并返回减去出借金额后的余额。

```
-- file: ch03/Lending.hs
lend amount balance = let reserve    = 100
                        newBalance = balance - amount
                        in if balance < reserve
                           then Nothing
                           else Just newBalance
```

这段代码中使用了 `let` 关键字标识一个变量声明区块的开始，用 `in` 关键字标识这个区块的结束。每行引入了一个局部变量。变量名在 `=` 的左侧，右侧则是该变量所绑定的表达式。

Note

特别提示

请特别注意我们的用词：在 `let` 区块中，变量名被绑定到了一个表达式而不是一个值。由于 Haskell 是一门惰性求值的语言，变量名所对应的表达式一直到被用到时才会求值。在上面的例子里，如果没有满足保证金的要求，就不会计算 `newBalance` 的值。

当我们在一个 `let` 区块中定义一个变量时，我们称之为“`let` 范围内的变量”。顾名思义即是：我们将这个变量限制在这个 `let` 区块内。

另外，上面这个例子中对空白和缩进的使用也值得特别注意。在下一节“The offside rule and white space in an expression”中我们会着重讲解其中的奥妙。

在 `let` 区块内定义的变量，既可以在定义区内使用，也可以在紧跟着 `in` 关键字的表达式中使用。

一般来说，我们将代码中可以使用一个变量名的地方称作这个变量名的作用域 (scope)。如果我们能使用，则说明在*作用域*内，反之则说明在作用域外。如果一个变量名在整个源代码的任意处都可以使用，则说明它位于最高层的作用域。

屏蔽

我们可以在表达式中使用嵌套的 `let` 区块。

```
-- file: ch03/NestedLets.hs
foo = let a = 1
```

 v: latest ▾

```
in let b = 2
    in a + b
```

上面的写法是完全合法的；但是在嵌套的 `let` 表达式里重复使用相同的变量名并不明智。

```
-- file: ch03/NestedLets.hs
bar = let x = 1
      in ((let x = "foo" in x), x)
```

如上，内部的 `x` 隐藏了，或称作屏蔽 (*shadowing*)，外部的 `x`。它们的变量名一样，但后者拥有完全不同的类型和值。

```
ghci> bar
("foo",1)
```

我们同样也可以屏蔽一个函数的参数，并导致更加奇怪的结果。你认为下面这个函数的类型是什么？

```
-- file: ch03/NestedLets.hs
quux a = let a = "foo"
        in a ++ "eek!"
```

在函数的内部，由于 `let`-绑定的变量名 `a` 屏蔽了函数的参数，使得参数 `a` 没有起到任何作用，因此该参数可以是任何类型的。

```
ghci> :type quux
quux :: t -> [Char]
```

Tip

编译器警告是你的朋友

显然屏蔽会导致混乱和恶心的 bug，因此 GHC 设置了一个有用的选项 `-fwarn-name-shadowing`。如果你开启了这个功能，每当屏蔽某个变量名时，GHC 就会打印出一条警告。

where 从句

还有另一种方法也可以用来引入局部变量：`where` 从句。`where` 从句中的定义在其所跟随的主句中有效。下面是和 `lend` 函数类似的一个例子，不同之处是使用了 `where` 而不是 `let`。

```
-- file: ch03/Lending.hs
lend2 amount balance = if amount < reserve * 0.5
                      then Just newBalance
                      else Nothing
  where reserve      = 100
        newBalance = balance - amount
```

尽管刚开始使用 `where` 从句通常会有异样的感觉，但它对于提升可读性有着巨大的帮助。它使得读者的注意力首先能集中在表达式的一些重要的细节上，而之后再补上支持性的定义。经过一段时间以后，如果再用回那些没有 `where` 从句的语言，你就会怀念它的存在了。

与 `let` 表达式一样，`where` 从句中的空白和缩进也十分重要。在下一节 “The offside rule and white space in an expression” 中我们会着重讲解其中的奥妙。

局部函数与全局变量

你可能已经注意到了，在 Haskell 的语法里，定义变量和定义函数的方式非常相似。这种相似性也存在于 `let` 和 `where` 区块里：定义局部函数就像定义局部变量那样简单。

```
-- file: ch03/LocalFunction.hs
pluralise :: String -> [Int] -> [String]
pluralise word counts = map plural counts
  where plural 0 = "no " ++ word ++ "s"
```

 v: latest ▼

```
plural 1 = "one " ++ word
plural n = show n ++ " " ++ word ++ "s"
```

我们定义了一个由多个等式构成的局部函数 `plural`。**局部函数可以自由地使用其被封装在的作用域内的任意变量**：在本例中，我们使用了在外部函数 `pluralise` 中定义的变量 `word`。在 `pluralise` 的定义里，`map` 函数（我们将在下一章里再来讲解它的用法）将局部函数 `plural` 逐一应用于 `counts` 列表的每个元素。

我们也可以在代码的一开始就定义变量，语法和定义函数是一样的。

```
-- file: ch03/GlobalVariable.hs
itemName = "Weighted Companion Cube"
```

The Offside Rule and Whitespace in an Expression

The Case Expression

Common Beginner Mistakes with Patterns

Conditional Evaluation with Guards

Exercises

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

第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

Wengel An —

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
- 第九章：I/O学习 —— 构建一个用于搜索文件系统的库 — Real World Haskell 中文版

3条评论 • 4年前

在原佐为 — “继续从左往右看” 这里英文是“from right to left”，是从右到左吧。
- 第十章：代码案例学习：解析二进制数据格式 — Real World Haskell 中文版

1条评论 • 4年前

Y — 此处少个“+”：instance Show Greymap where show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m