

第二章：类型和函数

类型是干什么用的？

Haskell 中的每个函数和表达式都带有各自的类型，通常称一个表达式拥有类型 `T`，或者说这个表达式的类型为 `T`。举个例子，布尔值 `True` 的类型为 `Bool`，而字符串 `"foo"` 的类型为 `String`。一个值的类型标识了它和该类型的其他值所共有的一簇属性（property），比如我们可以对数字进行相加，对列表进行拼接，诸如此类。

在对 Haskell 的类型系统进行更深入的探讨之前，不妨先来了解下，我们为什么要关心类型——也即是，它们是干什么用的？

在计算机的最底层，处理的都是没有任何附加结构的字节（byte）。而类型系统在这个基础上提供了抽象：它为那些单纯的字节加上了意义，使得我们可以说“这些字节是文本”，“那些字节是机票预约数据”，等等。

通常情况下，类型系统还会在标识类型的基础上更进一步：它会阻止我们混合使用不同的类型，避免程序错误。比如说，类型系统通常不会允许将一个酒店预订数据当作汽车租赁数据来使用。

引入抽象的使得我们可以忽略底层细节。举个例子，如果程序中的某个值是一个字符串，那么我不必考虑这个字符串在内部是如何实现的，只要像操作其他字符串一样，操作这个字符串就可以了。

类型系统的一个有趣的地方是，不同的类型系统的表现并不完全相同。实际上，不同类型系统有时候处理的还是不同种类的问题。

除此之外，一门语言的类型系统，还会深切地影响这门语言的使用者思考和编写程序的方式。而 Haskell 的类型系统则允许程序员以非常抽象的层次思考，并写出简洁、高效、健壮的代码。

Haskell 的类型系统

Haskell 中的类型有三个有趣的方面：首先，它们是强（strong）类型的；其次，它们是静态（static）的；第三，它们可以通过自动推导（automatically inferred）得出。

后面的三个小节会分别讨论这三个方面，介绍它们的长处和短处，并列举 Haskell 类型系统的概念和其他语言里相关构思之间的相似性。

强类型

Haskell 的强类型系统会拒绝执行任何无意义的表达式，保证程序不会因为某些表达式而引起错误：比如将整数当作函数来使用，或者将一个字符串传给一个只接受整数参数的函数，等等。

遵守类型规则的表达式被称为是“类型正确的”（well typed），而不遵守类型规则、会引起类型错误的表达式被称为是“类型不正确的”（ill typed）。

Haskell 强类型系统的另一个作用是，它不会自动地将值从一个类型转换到另一个类型（转换有时又称为强制或变换）。举个例子，如果将一个整数值作为参数传给了一个接受浮点数的函数，C 编译器会自动且静默（silently）地将参数从整数类型转换为浮点类型，而 Haskell 编译器则会引发一个编译错误。


要在 Haskell 中进行类型转换，必须显式地使用类型转换函数。

有些时候，强类型会让某种类型代码的编写变得困难。比如说，一种编写底层 C 代码的典型方式就是将一系列字节数组当作复杂的数据结构来操作。这种做法的效率非常高，因为它避免了对字节的复制操作。因为 Haskell 不允许这种形式的转换，所以要获得同等结构形式的数据，可能需要进行一些复制操作，这可能会对性能造成细微影响。

强类型的最大好处是可以让 bug 在代码实际运行之前浮现出来。比如说，在强类型的语言中，“不小心将整数当成了字符串来使用”这样的情况不可能出现。

[注意：这里说的“bug”指的是类型错误，和我们常说的、通常意义上的 bug 有一些区别。]

静态类型

静态类型系统指的是，编译器可以在编译期（而不是执行期）知道每个值和表达式的类型。Haskell 编译器或解释器会  `v: latest` 正确的表达式，并拒绝这些表达式的执行：

```
Prelude> True && "False"

<interactive>:2:9:
  Couldn't match expected type `Bool' with actual type `[Char]'
  In the second argument of `(++)', namely `"False"'
  In the expression: True && "False"
  In an equation for `it': it = True && "False"
```

类似的类型错误在之前已经看过了：编译器发现值 `"False"` 的类型为 `[Char]`，而 `(++)` 操作符要求两个操作对象的类型都为 `Bool`，虽然左边的操作对象 `True` 满足类型要求，但右边的操作对象 `"False"` 却不能匹配指定的类型，因此编译器以“类型不正确”为由，拒绝执行这个表达式。

静态类型有时候会让某种有用代码的编写变得困难。在 Python 这类语言里，duck typing 非常流行，只要两个对象的行为足够相似，那么就可以在它们之间进行互换。幸运的是，Haskell 提供的 typeclass 机制以一种安全、方便、实用的方式提供了大部分动态类型的优点。Haskell 也提供了一部分对全动态类型（truly dynamic types）编程的支持，尽管用起来没有专门支持这种功能的语言那么方便。

Haskell 对强类型和静态类型的双重支持使得程序不可能发生运行时类型错误，这也有助于捕捉那些轻微但难以发现的小错误，作为代价，在编程的时候就要付出更多的努力[译注：比如纠正类型错误和编写类型签名]。Haskell 社区有一种说法，一旦程序编译通过，那么这个程序的正确性就会比用其他语言来写要好得多。（一种更现实的说法是，Haskell 程序的小错误一般都很少。）

使用动态类型语言编写的程序，常常需要通过大量的测试来预防类型错误的发生，然而，测试通常很难做到巨细无遗：一些常见的任务，比如重构，非常容易引入一些测试没覆盖到的新类型错误。

另一方面，在 Haskell 里，编译器负责检查类型错误：编译通过的 Haskell 程序是不可能带有类型错误的。而重构 Haskell 程序通常只是移动一些代码块，编译，修复编译错误，并重复以上步骤直到编译无错为止。

要理解静态类型的好处，可以用玩拼图的例子来打比方：在 Haskell 里，如果一块拼图的形状不正确，那么它就不能被使用。另一方面，动态类型的拼图全部都是 1×1 大小的正方形，这些拼图无论放在那里都可以匹配，为了验证这些拼图被放到了正确的地方，必须使用测试来进行检查。

类型推导

关于类型系统，最后要说的是，Haskell 编译器可以自动推断出程序中几乎所有表达式的类型[注：有时候要提供一些信息，帮助编译器理解程序代码]。这个过程被称为类型推导（type inference）。

虽然 Haskell 允许我们显式地为任何值指定类型，但类型推导使得这种工作通常是可选的，而不是非做不可的事。

正确理解类型系统

对 Haskell 类型系统能力和好处的探索会花费好几个章节。在刚开始的时候，处理 Haskell 的类型可能会让你觉得有些麻烦。

比如说，在 Python 和 Ruby 里，你只要写下程序，然后测试一下程序的执行结果是否正确就够了，但是在 Haskell，你还要先确保程序能通过类型检查。那么，为什么要多走这些弯路呢？

答案是，静态、强类型检查使得 Haskell 更安全，而类型推导则让它更精炼、简洁。这样得出的结果是，比起其他流行的静态语言，Haskell 要来得更安全，而比起其他流行的动态语言，Haskell 的表现力又更胜一筹。

这并不是吹牛，等你看完这本书之后就会了解这一点。

修复编译时的类型错误刚开始会让人觉得增加了不必要的工作量，但是，换个角度来看，这不过是提前完成了调试工作：编译器在处理程序时，会将代码中的逻辑错误——展示出来，而不是一声不吭，任由代码在运行时出错。

更进一步来说，因为 Haskell 里值和函数的类型都可以通过自动推导得出，所以 Haskell 程序既可以获得静态类型带来的所有好处，而又不必像传统的静态类型语言那样，忙于添加各种各样的类型签名[译注：比如 C 语言的函数原型声明]——

在其他语言里，类型系统为编译器服务；而在 Haskell 里，类型系统为你服务。唯一的要求是，你需要学习如何在类型系统提供的框架下工作。

对 Haskell 类型的运用将遍布整本书，这些技术将帮助我们编写和测试实用的代码。

 v: latest ▾

一些常用的基本类型

以下是 Haskell 里最常用的一些基本类型，其中有些在之前的章节里已经看过了：

Char

单个 Unicode 字符。

Bool

表示一个布尔逻辑值。这个类型只有两个值：True 和 False。

Int

带符号的定长（fixed-width）整数。这个值的准确范围由机器决定：在 32 位机器里，Int 为 32 位宽，在 64 位机器里，Int 为 64 位宽。Haskell 保证 Int 的宽度不少于 28 位。（数值类型还可以是 8 位、16 位，等等，也可以是带符号和无符号的，以后会介绍。）

Integer

不限长度的带符号整数。Integer 并不像 Int 那么常用，因为它们需要更多的内存和更大的计算量。另一方面，对 Integer 的计算不会造成溢出，因此使用 Integer 的计算结果更可靠。

Double

用于表示浮点数。长度由机器决定，通常是 64 位。（Haskell 也有 Float 类型，但是并不推荐使用，因为编译器都是针对 Double 来进行优化的，而 Float 类型值的计算要慢得多。）

在前面的章节里，我们已经见到过 :: 符号。除了用来表示类型之外，它还可以用于进行类型签名。比如说，exp :: T 就是向 Haskell 表示，exp 的类型是 T，而 :: T 就是表达式 exp 的类型签名。如果一个表达式没有显式地指名类型的话，那么它的类型就通过自动推导来决定：

```
Prelude> :type 'a'
'a' :: Char

Prelude> 'a'           -- 自动推导
'a'

Prelude> 'a' :: Char   -- 显式签名
'a'
```

当然了，类型签名必须正确，否则 Haskell 编译器就会产生错误：

```
Prelude> 'a' :: Int    -- 试图将一个字符值标识为 Int 类型

<interactive>:7:1:
  Couldn't match expected type `Int' with actual type `Char'
  In the expression: 'a' :: Int
  In an equation for `it': it = 'a' :: Int
```

调用函数

要调用一个函数，先写出它的名字，后接函数的参数：

```
Prelude> odd 3
True

Prelude> odd 6
False
```

注意，函数的参数不需要用括号来包围，参数和参数之间也不需要逗号来隔开[译注：使用空格就可以了]：

```
Prelude> compare 2 3
LT

Prelude> compare 3 3
```

 v: latest ▾

```
EQ
Prelude> compare 3 2
GT
```

Haskell 函数的应用方式和其他语言差不多，但是格式要来得更简单。

因为函数应用的优先级比操作符要高，因此以下两个表达式是相等的：

```
Prelude> (compare 2 3) == LT
True

Prelude> compare 2 3 == LT
True
```

有时候，为了可读性考虑，添加一些额外的括号也是可以理解的，上面代码的第一个表达式就是这样一个例子。另一方面，在某些情况下，我们必须使用括号来让编译器知道，该如何处理一个复杂的表达式：

```
Prelude> compare (sqrt 3) (sqrt 6)
LT
```

这个表达式将 `sqrt 3` 和 `sqrt 6` 的计算结果分别传给 `compare` 函数。如果将括号移走，Haskell 编译器就会产生一个编译错误，因为它认为我们将四个参数传给了只需要两个参数的 `compare` 函数：

```
Prelude> compare sqrt 3 sqrt 6

<interactive>:17:1:
  The function `compare' is applied to four arguments,
  but its type `a0 -> a0 -> Ordering' has only two
  In the expression: compare sqrt 3 sqrt 6
  In an equation for `it': it = compare sqrt 3 sqrt 6
```

复合数据类型：列表和元组

复合类型通过其他类型构建得出。列表和元组是 Haskell 中最常用的复合数据类型。

在前面介绍字符串的时候，我们就已经见到过列表类型了：`String` 是 `[Char]` 的别名，而 `[Char]` 则表示由 `Char` 类型组成的列表。

`head` 函数取出列表的第一个元素：

```
Prelude> head [1, 2, 3, 4]
1

Prelude> head ['a', 'b', 'c']
'a'

Prelude> head []
*** Exception: Prelude.head: empty list
```

和 `head` 相反，`tail` 取出列表里除了第一个元素之外的其他元素：

```
Prelude> tail [1, 2, 3, 4]
[2,3,4]

Prelude> tail [2, 3, 4]
[3,4]

Prelude> tail [True, False]
[False]

Prelude> tail "list"
"ist"
```

 v: latest ▾

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

正如前面的例子所示，`head` 和 `tail` 函数可以处理不同类型的列表。将 `head` 应用于 `[Char]` 类型的列表，结果为一个 `Char` 类型的值，而将它应用于 `[Bool]` 类型的值，结果为一个 `Bool` 类型的值。`head` 函数并不关心它处理的是何种类型的列表。

因为列表中的值可以是任意类型，所以我们可以称列表为类型多态 (polymorphic) 的。当需要编写带有多态类型的代码时，需要使用类型变量。这些类型变量以小写字母开头，作为一个占位符，最终被一个具体的类型替换。

比如说，`[a]` 用一个方括号包围一个类型变量 `a`，表示一个“类型为 `a` 的列表”。这也就是说“我不在乎列表是什么类型，尽管给我一个列表就是了”。

当需要一个带有具体类型的列表时，就需要用一个具体的类型去替换类型变量。比如说，`[Int]` 表示一个包含 `Int` 类型值的列表，它用 `Int` 类型替换了类型变量 `a`。又比如，`[MyPersonalType]` 表示一个包含 `MyPersonalType` 类型值的列表，它用 `MyPersonalType` 替换了类型变量 `a`。

这种替换还可以递归地进行：`[[Int]]` 是一个包含 `[Int]` 类型值的列表，而 `[Int]` 又是一个包含 `Int` 类型值的列表。以下例子展示了一个包含 `Bool` 类型的列表的列表：

```
Prelude> :type [[True], [False, False]]
[[True], [False, False]] :: [[Bool]]
```

假设现在要用一个数据结构，分别保存一本书的出版年份——一个整数，以及这本书的书名——一个字符串。很明显，列表不能保存这样的信息，因为列表只能接受类型相同的值。这时，我们就需要使用元组：

```
Prelude> (1964, "Labyrinths")
(1964, "Labyrinths")
```

元组和列表非常不同，它们的两个属性刚刚相反：列表可以任意长，且只能包含类型相同的值；元组的长度是固定的，但可以包含不同类型的值。

元组的两边用括号包围，元素之间用逗号分割。元组的类型信息也使用同样的格式：

```
Prelude> :type (True, "hello")
(True, "hello") :: (Bool, [Char])

Prelude> (4, ['a', 'm'], (16, True))
(4, "am", (16, True))
```

Haskell 有一个特殊的类型 `()`，这种类型只有一个值 `()`，它的作用相当于包含零个元素的元组，类似于 C 语言中的 `void`：

```
Prelude> :t ()
() :: ()
```

通常用元组中元素的数量作为称呼元组的前缀，比如“2-元组”用于称呼包含两个元素的元组，“5-元组”用于称呼包含五个元素的元组，诸如此类。Haskell 不能创建 1-元组，因为 Haskell 没有相应的创建 1-元组的语法 (notation)。另外，在实际编程中，元组的元素太多会让代码变得混乱，因此元组通常只包含几个元素。

元组的类型由它所包含元素的数量、位置和类型决定。这意味着，如果两个元组里都包含着同样类型的元素，而这些元素的摆放位置不同，那么它们的类型就不相等，就像这样：

```
Prelude> :type (False, 'a')
(False, 'a') :: (Bool, Char)

Prelude> :type ('a', False)
('a', False) :: (Char, Bool)
```

除此之外，即使两个元组之间有一部分元素的类型相同，位置也一致，但是，如果它们的元素数量不同，那么它们的类型也不相等：

```
Prelude> :type (False, 'a')
(False, 'a') :: (Bool, Char)
```

 v: latest

```
Prelude> :type (False, 'a', 'b')
(False, 'a', 'b') :: (Bool, Char, Char)
```

只有元组中的数量、位置和类型都完全相同，这两个元组的类型才是相同的：

```
Prelude> :t (False, 'a')
(False, 'a') :: (Bool, Char)

Prelude> :t (True, 'b')
(True, 'b') :: (Bool, Char)
```

元组通常用于以下两个地方：

如果一个函数需要返回多个值，那么可以将这些值都包装到一个元组中，然后返回元组作为函数的值。
当需要使用定长容器，但又没有必要使用自定义类型的时候，就可以使用元组来对值进行包装。

处理列表和元组的函数

前面的内容介绍了如何构造列表和元组，现在来看看处理这两种数据结构的函数。

函数 `take` 和 `drop` 接受两个参数，一个数字 `n` 和一个列表 `l`。

`take` 返回一个包含 `l` 前 `n` 个元素的列表：

```
Prelude> take 2 [1, 2, 3, 4, 5]
[1, 2]
```

`drop` 则返回一个包含 `l` 丢弃了前 `n` 个元素之后，剩余元素的列表：

```
Prelude> drop 2 [1, 2, 3, 4, 5]
[3, 4, 5]
```

函数 `fst` 和 `snd` 接受一个元组作为参数，返回该元组的第一个元素和第二个元素：

```
Prelude> fst (1, 'a')
1

Prelude> snd (1, 'a')
'a'
```

将表达式传给函数

Haskell 的函数应用是左关联的。比如说，表达式 `a b c d` 等同于 `((a b) c) d`。要将一个表达式用作另一个表达式的参数，那么就必须显式地使用括号来包围它，这样编译器才会知道我们的真正意思：

```
Prelude> head (drop 4 "azety")
'y'
```

`drop 4 "azety"` 这个表达式被一对括号显式地包围，作为参数传入 `head` 函数。

如果将括号移走，那么编译器就会认为我们试图将三个参数传给 `head` 函数，于是它引发一个错误：

```
Prelude> head drop 4 "azety"

<interactive>:26:6:
  Couldn't match expected type `[t1 -> t2 -> t0]'
    with actual type `Int -> [a0] -> [a0]'
  In the first argument of `head`, namely `drop`
  In the expression: head drop 4 "azety"
  In an equation for `it`: it = head drop 4 "azety"
```

 v: latest ▾

函数类型

使用 `:type` 命令可以查看函数的类型[译注：缩写形式为 `:t`]：

```
Prelude> :type lines
lines :: String -> [String]
```

符号 `->` 可以读作“映射到”，或者（稍微不太精确地），读作“返回”。函数的类型签名显示，`lines` 函数接受单个字符串，并返回包含字符串值的列表：

```
Prelude> lines "the quick\nbrown fox\njumps"
["the quick","brown fox","jumps"]
```

结果表明，`lines` 函数接受一个字符串作为输入，并将这个字符串按行转义符号分割成多个字符串。

从 `lines` 函数的这个例子可以看出：函数的类型签名对于函数自身的功能有很大的提示作用，这种属性对于函数式语言的类型来说，意义重大。

[译注：`String -> [String]` 的实际意思是指 `lines` 函数定义了一个从 `String` 到 `[String]` 的函数映射，因此，这里将 `->` 的读法 to 翻译成“映射到”。]

纯度

副作用指的是，函数的行为受系统的全局状态所影响。

举个命令式语言的例子：假设有某个函数，它读取并返回某个全局变量，如果程序中的其他代码可以修改这个全局变量的话，那么这个函数的返回值就取决于这个全局变量在某一时刻的值。我们说这个函数带有副作用，尽管它并不亲自修改全局变量。

副作用本质上是函数的一种不可见的（invisible）输入或输出。Haskell 的函数在默认情况下都是无副作用的：函数的结果只取决于显式传入的参数。

我们将带副作用的函数称为“不纯（impure）函数”，而将不带副作用的函数称为“纯（pure）函数”。

从类型签名可以看出一个 Haskell 函数是否带有副作用 —— 不纯函数的类型签名都以 `IO` 开头：

```
Prelude> :type readFile
readFile :: FilePath -> IO String
```

Haskell 源码，以及简单函数的定义

既然我们已经学会了如何应用函数，那么是时候回过头来，学习怎样去编写函数。

因为 `ghci` 只支持 Haskell 特性的一个非常受限的子集，因此，尽管可以在 `ghci` 里面定义函数，但那里并不是编写函数最适当的环境。更关键的是，`ghci` 里面定义函数的语法和 Haskell 源码里定义函数的语法并不相同。综上所述，我们选择将代码写在源码文件里。

Haskell 源码通常以 `.hs` 作为后缀。我们创建一个 `add.hs` 文件，并将以下定义添加到文件中：

```
-- file: ch02/add.hs
add a b = a + b
```

[译注：原书代码里的路径为 `ch03/add.hs`，是错误的。]

= 号左边的 `add a b` 是函数名和函数参数，而右边的 `a + b` 则是函数体，符号 `=` 表示将左边的名字（函数名和函数参数）定义为右边的表达式（函数体）。

将 `add.hs` 保存之后，就可以在 `ghci` 里通过 `:load` 命令（缩写为 `:l`）载入它，接着就可以像使用其他函数一样，调用 `add` 函数了：

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

 v: latest ▾

```
*Main> add 1 2 -- 包载入成功之后 ghci 的提示符会发生变化
3
```

[译注：你的当前文件夹（CWD）必须是 `ch02` 文件夹，否则直接载入 `add.hs` 会失败]

当以 `1` 和 `2` 作为参数应用 `add` 函数的时候，它们分别被赋值给（或者说，绑定到）函数定义中的变量 `a` 和 `b`，因此得出的结果表达式为 `1 + 2`，而这个表达式的值 `3` 就是本次函数应用的结果。

Haskell 不使用 `return` 关键字来返回函数值：因为一个函数就是一个单独的表达式（`expression`），而不是一组陈述（`statement`），求值表达式所得的结果就是函数的返回值。（实际上，Haskell 有一个名为 `return` 的函数，但它和命令式语言里的 `return` 不是同一回事。）

变量

在 Haskell 里，可以使用变量来赋予表达式名字：一旦变量绑定了（也即是，关联起）某个表达式，那么这个变量的值就不会改变——我们总能用这个变量来指代它所关联的表达式，并且每次都会得到同样的结果。

如果你曾经用过命令式语言，就会发现 Haskell 的变量和命令式语言的变量很不同：在命令式语言里，一个变量通常用于标识一个内存位置（或者其他类似的东西），并且在任何时候，都可以随意修改这个变量的值。因此在不同时间点上，访问这个变量得出的值可能是完全不同的。

对变量的这两种不同的处理方式产生了巨大的差别：在 Haskell 程序里面，当变量和表达式绑定之后，我们总能将变量替换成相应的表达式。但是在声明式语言里面就没有办法做这样的替换，因为变量的值可能无时无刻都处在改变当中。

举个例子，以下 Python 脚本打印出值 `11`：

```
x = 10
x = 11
print(x)
```

[译注：这里将原书的代码从 `print x` 改为 `print(x)`，确保代码在 Python 2 和 Python 3 都可以顺利执行。]

然后，试着在 Haskell 里做同样的事：

```
-- file: ch02/Assign.hs
x = 10
x = 11
```

但是 Haskell 并不允许做这样的多次赋值：

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

Assign.hs:3:1:
  Multiple declarations of `x`
    Declared at: Assign.hs:2:1
               Assign.hs:3:1
Failed, modules loaded: none.
```

条件求值

和很多语言一样，Haskell 也有自己的 `if` 表达式。本节先说明怎么用这个表达式，然后再慢慢介绍它的详细特性。

我们通过编写一个个人版本的 `drop` 函数来熟悉 `if` 表达式。先来回顾一下 `drop` 的行为：

```
Prelude> drop 2 "foobar"
"obar"

Prelude> drop 4 "foobar"
"ar"

Prelude> drop 4 [1, 2]
[]
```

 v: latest ▾


```
Prelude> drop 0 [1, 2]
[1, 2]

Prelude> drop 7 []
[]

Prelude> drop (-2) "foo"
"foo"
```

从测试代码的反馈可以看到。当 `drop` 函数的第一个参数小于或等于 0 时，`drop` 函数返回整个输入列表。否则，它就从列表左边开始移除元素，一直到移除元素的数量足够，或者输入列表被清空为止。

以下是带有同样行为的 `myDrop` 函数，它使用 `if` 表达来决定该做什么。而代码中的 `null` 函数则用于检查列表是否为空：

```
-- file: ch02/myDrop.hs
myDrop n xs = if n <= 0 || null xs
              then xs
              else myDrop (n - 1) (tail xs)
```

在 Haskell 里，代码的缩进非常重要：它会延续 (continue) 一个已存在的定义，而不是新创建一个。所以，不要省略缩进！

变量 `xs` 展示了一个命名列表的常见模式：`s` 可以视为后缀，而 `xs` 则表示“复数个 `x`”。

先保存文件，试试 `myDrop` 函数是否如我们所预期的那样工作：

```
[1 of 1] Compiling Main          ( myDrop.hs, interpreted )
Ok, modules loaded: Main.

*Main> myDrop 2 "foobar"
"obar"

*Main> myDrop 4 "foobar"
"ar"

*Main> myDrop 4 [1, 2]
[]

*Main> myDrop 0 [1, 2]
[1, 2]

*Main> myDrop 7 []
[]

*Main> myDrop (-2) "foo"
"foo"
```

好的，代码正如我们所想的那样运行，现在是时候回过头来，说明一下 `myDrop` 的函数体里都干了些什么：

`if` 关键字引入了一个带有三个部分的表达式：

- 跟在 `if` 之后的是一个 `Bool` 类型的表达式，它是 `if` 的条件部分。
- 跟在 `then` 关键字之后的是另一个表达式，这个表达式在条件部分的值为 `True` 时被执行。
- 跟在 `else` 关键字之后的又是另一个表达式，这个表达式在条件部分的值为 `False` 时被执行。

我们将跟在 `then` 和 `else` 之后的表达式称为“分支”。不同分支之间的类型必须相同。[译注：这里原文还有一句“the `if` expression will also have this type”，这是错误的，因为条件部分的表达式只要是 `Bool` 类型就可以了，没有必要和分支的类型相同。]像是 `if True then 1 else "foo"` 这样的表达式会产生错误，因为两个分支的类型并不相同：

```
Prelude> if True then 1 else "foo"

<interactive>:2:14:
  No instance for (Num [Char])
    arising from the literal `1'
  Possible fix: add an instance declaration for (Num [Char])
  In the expression: 1
```

 v: latest ▾

```
In the expression: if True then 1 else "foo"
In an equation for `it`: it = if True then 1 else "foo"
```

记住，Haskell 是一门以表达式为主导（expression-oriented）的语言。在命令式语言中，代码由陈述（statement）而不是表达式组成，因此在省略 if 语句的 else 分支的情况下，程序仍是有意义的。但是，当代码由表达式组成时，一个缺少 else 分支的 if 语句，在条件部分为 False 时，是没有办法给出一个结果的，当然这个 else 分支也不会有任何类型，因此，省略 else 分支对于 Haskell 是无意义的，编译器也不会允许这么做。

程序里还有几个新东西需要解释。其中，null 函数检查一个列表是否为空：

```
Prelude> :type null
null :: [a] -> Bool

Prelude> null []
True

Prelude> null [1, 2, 3]
False
```

而 (||) 操作符对它的 Bool 类型参数执行一个逻辑或（logical or）操作：

```
Prelude> :type (||)
(||) :: Bool -> Bool -> Bool

Prelude> True || False
True

Prelude> True || True
True
```

另外需要注意的是，myDrop 函数是一个递归函数：它通过调用自身来解决问题。关于递归，书本稍后会做更详细的介绍。

最后，整个 if 表达式被分成了多行，而实际上，它也可以写成一行：

```
-- file: ch02/myDropX.hs
myDropX n xs = if n <= 0 || null xs then xs else myDropX (n - 1) (tail xs)
```

[译注：原文这里的文件名称为 myDrop.hs，为了和之前的 myDrop.hs 区别开来，这里修改文件名，让它和函数名 myDropX 保持一致。]

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

*Main> myDropX 2 "foobar"
"obar"
```

这个一行版本的 myDrop 比起之前的定义要难读得多，为了可读性考虑，一般来说，总是应该通过分行来隔开条件部分和两个分支。

作为对比，以下是一个 Python 版本的 myDrop，它的结构和 Haskell 版本差不多：

```
def myDrop(n, elts):
    while n > 0 and elts:
        n = n - 1
        elts = elts[1:]
    return elts
```

通过示例了解求值

前面对 myDrop 的描述关注的都是表面上的特性。我们需要更进一步，开发一个关于函数是如何被应用的心智模型：为此，我们先从一些简单的示例出发，逐步深入，直到搞清楚 myDrop 2 "abcd" 到底是怎样求值为止。

 v: latest ▾

在前面的章节里多次谈到，可以使用一个表达式去替换一个变量。在这部分的内容里，我们也会看到这种替换能力：计算过程需要多次对表达式进行重写，并将变量替换为表达式，直到产生最终结果为止。为了帮助理解，最好准备一些纸和笔，跟着书本的说明，自己计

算一次。

惰性求值

先从一个简单的、非递归例子开始，其中 `mod` 函数是典型的取模函数：

```
-- file: ch02/isOdd.hs
isOdd n = mod n 2 == 1
```

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

我们的第一个任务是，弄清楚 `isOdd (1 + 2)` 的结果是如何求值出的。

在使用严格求值的语言里，函数的参数总是在应用函数之前被求值。以 `isOdd` 为例子：子表达式 `(1 + 2)` 会首先被求值，得出结果 `3`。接着，将 `3` 绑定到变量 `n`，应用到函数 `isOdd`。最后，`mod 3 2` 返回 `1`，而 `1 == 1` 返回 `True`。

Haskell 使用了另外一种求值方式——**非严格求值**。在这种情况下，求值 `isOdd (1 + 2)` 并不会即刻使得子表达式 `1 + 2` 被求值为 `3`，相反，编译器做出了一个“承诺”，说，“当真正有需要的时候，我有办法计算出 `isOdd (1 + 2)` 的值”。

用于追踪未求值表达式的记录被称为块（chunk）。这就是事情发生的经过：编译器通过创建块来延迟表达式的求值，直到这个表达式的值真正被需要为止。如果某个表达式的值不被需要，那么从始至终，这个表达式都不会被求值。

非严格求值通常也被称为**惰性求值**。[注：实际上，“非严格”和“惰性”在技术上有些细微的差别，但这里不讨论这些细节。]

一个更复杂的例子

现在，将注意力放回 `myDrop 2 "abcd"` 上面，考察它的结果是如何计算出来的：

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

*Main> myDrop 2 "abcd"
"cd"
```

当执行表达式 `myDrop 2 "abcd"` 时，函数 `myDrop` 应用于值 `2` 和 `"abcd"`，变量 `n` 被绑定为 `2`，而变量 `xs` 被绑定为 `"abcd"`。将这两个变量代换到 `myDrop` 的条件判断部分，就得出了以下表达式：

```
*Main> :type 2 <= 0 || null "abcd"
2 <= 0 || null "abcd" :: Bool
```

编译器需要对表达式 `2 <= 0 || null "abcd"` 进行求值，从而决定 `if` 该执行哪一个分支。这需要对 `(||)` 表达式进行求值，而要求值这个表达式，又需要对它的左操作符进行求值：

```
*Main> 2 <= 0
False
```

将值 `False` 代换到 `(||)` 表达式当中，得出以下表达式：

```
*Main> :type False || null "abcd"
False || null "abcd" :: Bool
```

如果 `(||)` 左操作符的值为 `True`，那么 `(||)` 就不需要对右操作符进行求值，因为整个 `(||)` 表达式的值已经由左操作符决定了。[译注：在逻辑或计算中，只要有一个变量的值为真，那么结果就为真。]另一方面，因为这里左操作符的值为 `False`，那么 `(||)` 表达式的值由右操作符的值来决定：

```
*Main> null "abcd"
False
```

 v: latest ▾

最后，将左右两个操作对象的值分别替换回 `(||)` 表达式，得出以下表达式：

```
*Main> False || False
False
```

这个结果表明，下一步要求值的应该是 `if` 表达式的 `else` 分支，而这个分支包含一个对 `myDrop` 函数自身的递归调用：`myDrop (2 - 1) (tail "abcd")`。

递归

当递归地调用 `myDrop` 的时候，`n` 被绑定为块 `2 - 1`，而 `xs` 被绑定为 `tail "abcd"`。

于是再次对 `myDrop` 函数进行求值，这次将新的值替换到 `if` 的条件判断部分：

```
*Main> :type (2 - 1) <= 0 || null (tail "abcd")
(2 - 1) <= 0 || null (tail "abcd") :: Bool
```

对 `(||)` 的左操作符的求值过程如下：

```
*Main> :type (2 - 1)
(2 - 1) :: Num a => a

*Main> 2 - 1
1

*Main> 1 <= 0
False
```

正如前面“惰性求值”一节所说的那样，`(2 - 1)` 只有在真正需要的时候才会被求值。同样，对右操作符 `(tail "abcd")` 的求值也会被延迟，直到真正有需要时才被执行：

```
*Main> :type null (tail "abcd")
null (tail "abcd") :: Bool

*Main> tail "abcd"
"bcd"

*Main> null "bcd"
False
```

因为条件判断表达式的最终结果为 `False`，所以这次执行的也是 `else` 分支，而被执行的表达式为 `myDrop (1 - 1) (tail "bcd")`。

终止递归

这次递归调用将 `1 - 1` 绑定到 `n`，而 `xs` 被绑定为 `tail "bcd"`：

```
*Main> :type (1 - 1) <= 0 || null (tail "bcd")
(1 - 1) <= 0 || null (tail "bcd") :: Bool
```

再次对 `(||)` 操作符的右操作对象求值：

```
*Main> :type (1 - 1) <= 0
(1 - 1) <= 0 :: Bool
```

最终，我们得出了一个 `True` 值！

```
*Main> True || null (tail "bcd")
True
```

因为 `(||)` 的右操作符 `null (tail "bcd")` 并不影响表达式的计算结果，因此它没有被求值，而整个条件判断部分的最终分支被求值：

```
*Main> :type tail "bcd"
tail "bcd" :: [Char]
```

从递归中返回

请注意，在求值的最后一步，结果表达式 `tail "bcd"` 处于两次对 `myDrop` 的递归调用当中。

因此，表达式 `tail "bcd"` 作为结果值，被返回给对 `myDrop` 的第二次递归调用：

```
*Main> myDrop (1 - 1) (tail "bcd") == tail "bcd"
True
```

接着，第二次递归调用所得的值（还是 `tail "bcd"`），它被返回给第一次递归调用：

```
*Main> myDrop (2 - 1) (tail "abcd") == tail "bcd"
True
```

然后，第一次递归调用也将 `tail "bcd"` 作为结果值，返回给最开始的 `myDrop` 调用：

```
*Main> myDrop 2 "abcd" == tail "bcd"
True
```

最终计算出结果 `"cd"`：

```
*Main> myDrop 2 "abcd"
"cd"

*Main> tail "bcd"
"cd"
```

注意，在从递归调用中退出并传递结果值的过程中，`tail "bcd"` 并不会被求值，只有当它返回到最开始的 `myDrop` 之后，`ghci` 需要打印这个值时，`tail "bcd"` 才会被求值。

学到了什么？

这一节介绍了三个重要的知识点：

可以通过代换（substitution）和重写（rewriting）去了解 Haskell 求值表达式的方式。

惰性求值可以延迟计算直到真正需要一个值为止，并且在求值时，也只执行可以给出（establish）值的那部分表达式。[译注：比如之前提到的，`(||)` 的左操作符的值为 `True` 时的情况。]

函数的返回值可能是一个块（一个被延迟计算的表达式）。

Haskell 里的多态

之前介绍列表的时候提到过，列表是类型多态的，这一节会说明更多这方面的细节。

如果想要取出一个列表的最后一个元素，那么可以使用 `last` 函数。`last` 函数的返回值和列表中的元素的类型是相同的，但是，`last` 函数并不介意输入的列表是什么类型，它对于任何类型的列表都可以产生同样的效果：

```
Prelude> last [1, 2, 3, 4, 5]
5

Prelude> last "baz"
'z'
```

`last` 的秘密就隐藏在类型签名里面：

```
Prelude> :type last
last :: [a] -> a
```

 v: latest ▾

这个类型签名可以读作“`last` 接受一个列表，这个列表里的所有元素的类型都为 `a`，并返回一个类型为 `a` 的元素作为返回值”，其中 `a` 是类型变量。

如果函数的类型签名里包含类型变量，那么就表示这个函数的某些参数可以是任意类型，我们称这些函数是多态的。

如果将一个类型为 `[Char]` 的列表传给 `last`，那么编译器就会用 `Char` 替换 `last` 函数类型签名中的所有 `a`，从而得出一个类型为 `[Char] -> Char` 的 `last` 函数。而对于 `[Int]` 类型的列表，编译器则产生一个类型为 `[Int] -> Int` 类型的 `last` 函数，诸如此类。

这种类型的多态被称为参数多态。可以用一个类比来帮助理解这个名字：就像函数的参数可以被其他实际的值绑定一样，Haskell 的类型也可以带有参数，并且这些参数也可以被其他实际的类型绑定。

当看见一个参数化类型（parameterized type）时，这表示代码并不在乎实际的类型是什么。另外，我们还可以给出一个更强的陈述：没有办法知道参数化类型的实际类型是什么，也不能操作这种类型的值；不能创建这种类型的值，也不能对这种类型的值进行探查（inspect）。

参数化类型唯一能做的事，就是作为一个完全抽象的“黑箱”而存在。稍后的内容会解释为什么这个性质对参数化类型来说至关重要。

参数多态是 Haskell 支持的多态中最明显的一个。Haskell 的参数多态直接影响了 Java 和 C# 等语言的泛型（generic）功能的设计。Java 泛型中的类型变量和 Haskell 的参数化类型非常相似。而 C++ 的模板也和参数多态相去不远。

为了弄清楚 Haskell 的多态和其他语言的多态之间的区别，以下是一些被流行语言所使用的多态形式，这些形式的多态都没有在 Haskell 里出现：

在主流的面向对象语言中，子类多态是应用得最广泛的一种。C++ 和 Java 的继承机制实现了子类多态，使得子类可以修改或扩展父类所定义的行为。Haskell 不是面向对象语言，因此它没有提供子类多态。

另一个常见的多态形式是强制多态（coercion polymorphism），它允许值在类型之间进行隐式的转换。很多语言都提供了对强制多态的某种形式的支持，其中一个例子就是：自动将整数类型值转换成浮点数类型值。既然 Haskell 坚决反对自动类型转换，那么这种多态自然也不会出现在 Haskell 里面。

关于多态还有很多东西要说，本书第六章会再次回到这个主题。

对多态函数进行推理

前面的《函数类型》小节介绍过，可以通过查看函数的类型签名来了解函数的行为。这种方法同样适用于对多态类型进行推理。

以 `fst` 函数为例子：

```
Prelude> :type fst
fst :: (a, b) -> a
```

首先，函数签名包含两个类型变量 `a` 和 `b`，表明元组可以包含不同类型的值。

其次，`fst` 函数的结果值的类型为 `a`。前面提到过，参数多态没有办法知道输入参数的实际类型，并且它也没有足够的信息构造一个 `a` 类型的值，当然，它也不可以将 `a` 转换为 `b`。因此，这个函数唯一合法的行为，就是返回元组的第一个元素。



延伸阅读

前一节所说的 `fst` 函数的类型推导行为背后隐藏着非常高深的数学知识，并且可以延伸出一系列复杂的多态函数。有兴趣的话，可以参考 Philip Wadler 的 Theorems for free 论文。

多参数函数的类型

截至目前为止，我们已经见到过一些函数，比如 `take`，它们接受一个以上的参数：

```
Prelude> :type take
take :: Int -> [a] -> [a]
```

通过类型签名可以看到，`take` 函数和一个 `Int` 值以及两个列表有关。类型签名中的 `->` 符号是右关联的：Haskell 从右  `v: latest`  这些箭头，使用括号可以清晰地标示这个类型签名是怎样被解释的：


```
-- file: ch02/Take.hs
take :: Int -> ([a] -> [a])
```

从这个新的类型签名可以看出，`take` 函数实际上只接受一个 `Int` 类型的参数，并返回另一个函数，这个新函数接受一个列表作为参数，并返回一个同类型的列表作为这个函数的结果。

以上的说明都是正确的，但要清楚隐藏在这种变换背后的重要性并不容易，在《部分函数应用和柯里化》一节，我们会再次回到这个主题上。目前来说，可以简单地将类型签名中最后一个 `->` 右边的类型看作是函数结果的类型，而将前面的其他类型看作是函数参数的类型。

了解了这些之后，现在可以为前面定义的 `myDrop` 函数编写类型签名了：

```
myDrop :: Int -> [a] -> [a]
```

为什么要对纯度斤斤计较？

很少有语言像 `Haskell` 那样，默认使用纯函数。这个选择不仅意义深远，而且至关重要。

因为纯函数的值只取决于输入的参数，所以通常只要看看函数的名字，还有它的类型签名，就能大概知道函数是干什么用的。

以 `not` 函数为例子：

```
Prelude> :type not
not :: Bool -> Bool
```

即使抛开函数名不说，单单函数签名就极大地限制了这个函数可能的合法行为：

- 函数要么返回 `True`，要么返回 `False`
- 函数直接将输入参数当作返回值返回
- 函数对它的输入值求反

除此之外，我们还能肯定，这个函数不会干以下这些事情：读取文件，访问网络，或者返回当前时间。

纯度减轻了理解一个函数所需的工作量。一个纯函数的行为并不取决于全局变量、数据库的内容或者网络连接状态。纯代码（pure code）从一开始就是模块化的：每个函数都是自包含的，并且都带有定义良好的接口。

将纯函数作为默认的另一个不太明显的好处是，它使得与不纯代码之间的交互变得简单。一种常见的 `Haskell` 风格就是，将带有副作用的代码和不带副作用的代码分开处理。在这种情况下，不纯函数需要尽可能地简单，而复杂的任务则交给纯函数去做。

软件的大部分风险，都来自于与外部世界进行交互：它需要程序去应付错误的、不完整的数据，并且处理恶意的攻击，诸如此类。`Haskell` 的类型系统明确地告诉我们，哪一部分的代码带有副作用，让我们可以对这部分代码添加适当的保护措施。

通过这种将不纯函数隔离、并尽可能简单化的编程风格，程序的漏洞将变得非常少。

回顾

这一章对 `Haskell` 的类型系统以及类型语法进行了快速的概览，了解了基本类型，并学习了如何去编写简单的函数。这章还介绍了多态、条件表达式、纯度和惰性求值。

这些知识必须被充分理解。在第三章，我们就会在这些基本知识的基础上，进一步加深对 `Haskell` 的理解。

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 HASKELL 中文版 上还有

Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

forlice — ...

Real World Haskell 中文版 — Real World Haskell 中文版

4条评论 • 6年前

Jojo — 更新好慢呀
- 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

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 —