

第一章：入门

Haskell编程环境

在本书的前面一些章节里，我们有时候会以限制性的、简单的形式来介绍一些概念。由于Haskell是一本比较深的语言，所以一次性介绍某个主题的所有特性会令人难以接受。当基础巩固后，我们就会进行更加深入的学习。

在Haskell语言的众多实现中，有两个被广泛应用，Hugs和GHC。其中Hugs是一个解析器，主要用于教学。而GHC(Glasgow Haskell Compiler)更加注重实践，它编译成本地代码，支持并行执行，并带有更好的性能分析工具和调试工具。由于这些因素，在本书中我们将采用GHC。

GHC主要有三个部分组成。

ghc是生成快速本地代码的优化编译器。

ghci是一个交互解析器和调试器。

runghc是一个以脚本形式(并不要首先编译)运行Haskell代码的程序，

Note

我们如何称呼GHC的各个组件

当我们讨论整个GHC系统时，我们称之为GHC。而如果要引用到某个特定的命令，我们会直接用其名字标识，比如 **ghc**，**ghci**，**runghc**。

在本书中，我们假定你在使用最新版6.8.2版本的GHC，这个版本是2007年发布的。大多数例子不要额外的修改也能在老的版本上运行。然而，我们建议使用最新版本。如果你是Windows或者Mac OS X操作系统，你可以使用预编译的安装包快速上手。你可以从 [GHC 下载页面](#) 找到合适的二进制包或者安装包。

对于大多数的Linux版本，BSD提供版和其他Unix系列，你可以找到自定义的GHC二进制包。由于这些包要基于特性的环境编译，所以安装和使用显得更加容易。你可以在GHC的 [二进制发布包页面](#) 找到相关下载。

我们在[附录A]中提供了更多详细的信息介绍如何在各个流行平台上安装GHC。

初识解释器ghci

ghci程序是GHC的交互式解析器。它可以让用户输入Haskell表达式并对其求值，浏览模块以及调试代码。如果你熟悉Python或是Ruby，那么ghci一定程度上和python，irb很像，这两者分别是Python和Ruby的交互式解析器。

```
The ghci command has a narrow focus
We typically can not copy some code out of a haskell source file and paste it into ghci. This does not have a significant effect on debugging p
```

在类Unix系统中，我们在shell视窗下运行**ghci**。而在Windows系统下，你可以通过开始菜单找到它。比如，如果你在Windows XP下安装了GHC，你应该从“所有程序”，然后“GHC”下找到**ghci**。(参考[附录A章节Windows](#)里的截图。)

当我们运行**ghci**时，它会首先显示一个初始banner，然后就显示提示符**Prelude>**。下载例子展示的是Linux环境下的6.8.3版本。

```
$ ghci
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.
Prelude>
```

提示符**Prelude**标识一个很常用的库**Prelude**已经被加载并可以使用。同样的，当加载了其他模块或是源文件时，它们也会在出现在提示符的位子。

Tip

获取帮助信息

 v: latest ▾

在ghci提示符输入 `:?`，则会显示详细的帮助信息。

模块`Prelude`有时候被称为“标准序幕”(the standard prelude)，因为它的内容是基于Haskell 98标准定义的。通常简称它为“序幕”(the prelude)。

Note

关于ghci的提示符

提示符经常是随着模块的加载而变化。因此经常会变得很长以至在单行中没有太多可视区域用来输入。

为了简单和一致起见，在本书中我们会用字符串 `'ghci>'` 来替代ghci的默认提示符。

你可以用ghci的 `:set prompt` 来进行修改。

```
Prelude> :set prompt "ghci>"
ghci>
```

prelude模块中的类型，值和函数是默认直接可用的，在使用之前我们不需要额外的操作。然而如果需要其他模块中的一些定义，则需要使用ghci的`:module`方法预先加载。

```
ghci> :module + Data.Ratio
```

现在我们可以使用`Data.Ratio`模块中的功能了。这个模块提供了一些操作有理数的功能。

基本交互: 把ghci当作一个计算器

除了能提供测试代码片段的交互功能外，**ghci**也可以被当作一个桌面计算器来使用。我们可以很容易的表示基本运算，同时随着对Haskell了解的深入，也可以表示更加复杂的运算。即使是以如此简单的方式来使用这个解析器，也可以帮助我们了解更多关于Haskell是如何工作的。

基本算术运算

我们可以马上开始输入一些表达式，看看**ghci**会怎么处理它们。基本的算术表达式类似于像C或是Python这样的语言：用中缀表达式，即操作符在操作数之间。

```
ghci> 2 + 2
4
ghci> 31337 * 101
3165037
ghci> 7.0 / 2.0
3.5
```

用中缀表达式是为了书写方便：我们同样可以用前缀表达式，即操作符在操作数之前。在这种情况下，我们需要用括号将操作符括起来。

```
ghci> 2 + 2
4
ghci> (+) 2 2
4
```

上述的这些表达式暗示了一个概念，Haskell有整数和浮点数类型。整数的大小是随意的。下面例子中的`(^)`表示了整数的乘方。

```
ghci> 313 ^ 15
27112218957718876716220410905036741257
```

 v: latest ▾

算术奇事(quirk),负数的表示

在如何表示数字方面Haskell提供给我们一个特性：通常需要将负数写在括号内。当我们要表示不是最简单的表达式时，这个特性就开始发挥影响。

我们先开始表示简单的负数

```
ghci> -3
-3
```

上述例子中的`-`是一元表达式。换句话说，我们并不是写了一个数字“-3”；而是一个数字“3”，然后作用于操作符`-`。`-`是Haskell中唯一的一元操作符，而且我们也不能将它和中缀运算符一起使用。

```
ghci> 2 + -3

<interactive>:1:0:
  precedence parsing error
    cannot mix `(+)' [infixl 6] and prefix `-' [infixl 6] in the same infix expression
```

如果需要在同一个中缀操作符附近使用一元操作符，则需要将一元操作符以及其操作数包含的括号内。

```
ghci> 2 + (-3)
-1
ghci> 3 + (-(13 * 37))
-478
```

如此可以避免解析的不确定性。当在Haskell应用(`apply`)一个函数时，我们先写函数名，然后随之其参数，比如`f 3`。如果我们不用括号括起一个负数，就会有非常明显的不同的方式理解`f-3`：它可以是“将函数`f`应用(`apply`)与数字-3”，或者是“把变量`f`减去3”。

大多数情况下，我们可以省略表达式中的空格(“空”字符比如空格或制表符`tab`)，Haskell也同样能正确的解析。但并不是所有的情况。

```
ghci> 2*3
6
```

下面的例子和上面有问题的负数的例子很像，然而它的错误信息并不一样。

```
ghci> 2*-3

<interactive>:1:1: Not in scope: `*-'
```

这里Haskell把`*-`理解成单个的操作符。Haskell允许用户自定义新的操作符（这个主题我们随后会讲到），但是我们未曾定义过`*-`。

```
ghci> 2*(-3)
-6
```

相比较其他的编程语言，这种对于负数不太一样的行为可能会很怪异，然后它是一种合理的折中方式。Haskell允许用户在任何时候自定义新的操作符。这是一个并不深奥的语言特性，我们会在以后的章节中看到许多用户定义的操作符。语言的设计者们为了拥有这个表达式强项而接受了这个有一点累赘的负数表达语法。

布尔逻辑，运算符以及值比较

Haskell中表示布尔逻辑的值有这么两个：`True`和`False`。名字中的大写很重要。作用于布尔值得操作符类似于C语言的情况：`(&&)`表示“逻辑与”，`(||)`表示“逻辑或”。

```
ghci> True && False
False
ghci> False || True
True
```

有些编程语言中会定义数字0和`False`同义，但是在Haskell中并没有这么定义，同样的，也Haskell也没有定义非0的值为

```
ghci> True && 1

<interactive>:1:8:
  No instance for (Num Bool)
    arising from the literal `1' at <interactive>:1:8
  Possible fix: add an instance declaration for (Num Bool)
  In the second argument of `(&&)', namely `1'
  In the expression: True && 1
  In the definition of `it': it = True && 1
```

我们再一次的遇到了很有前瞻性的错误。简单来说，错误信息告诉我们布尔类型，`Bool`，不是数字类型，`Num`的一个成员。错误信息有些长，这是因为`ghci`会定位出错的具体位置，并且给出了也许能解决问题的修改提示。

错误信息详细分析如下。

“No instance for (Num Bool)” 告诉我们`ghci`尝试解析数字`1`为`Bool`类型但是失败。

“arising from the literal `1’” 表示是由于使用了数字`1`而引发了问题。

“In the definition of `it’” 引用了一个`ghci`的快捷方式。我们会在后面提到。

Tip

遇到错误信息不要胆怯

这里我们提到了很重要的一点，而且在本书的前面一些章节中我们会重复提到。如果你碰到一些你从来没遇到过的问题和错误信息，别担心(panic)。刚开始的时候，你所要的做的仅仅是找出足够的信息来帮助解决问题。随着你经验的积累，你会发现错误信息中的一部分其实很容易理解，并不会像刚开始时那么晦涩难懂。

各种错误信息都有一个目的：通过提前的一些调试，帮助我们在真正运行程序之前能书写出正确的代码。如果你曾使用过其它更加宽松(permissive)的语言，这种方式可能会有些震惊(shock)。所以，拿出你的耐心来。

Haskell中大多数比较操作符和C语言以及受C语言影响的语言类似。

```
ghci> 1 == 1
True
ghci> 2 < 3
True
ghci> 4 >= 3.99
True
```

有一个操作符和C语言的相应的不一样，“不等于”。C语言中是用`!=`表示的，而Haskell是用`/=`表示的，它看上去很像数学中的 \neq 。

另外，类C的语言中通常用`!`表示逻辑非的操作，而Haskell中用函数`not`。

```
ghci> not True
False
```

运算符优先级以及结合性

类似于代数或是使用中缀操作符的编程语言，Haskell也有操作符优先级的概念。我们可以使用括号将部分表达显示的组合在一起，同时操作符优先级允许省略掉一些括号。比如乘法比加法优先级高，因此以下两个表达式效果是一样的。

```
ghci> 1 + (4 * 4)
17
ghci> 1 + 4 * 4
17
```

Haskell给每个操作符一个数值型的优先级值，从1表示最低优先级，到9表示最高优先级。高优先级的操作符先于低优先级的操作符被应用(apply)。在`ghci`中我们可以用命令`:info`来查看某个操作符的优先级。

```
ghci> :info (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
```

 v: latest ▾

```
...
-- Defined in GHC.Num
infixl 6 +
ghci> :info (*)
class (Eq a, Show a) => Num a where
...
(*) :: a -> a -> a
...
-- Defined in GHC.Num
infixl 7 *
```

这里我们需要找的信息是“infixl 6 +”，表示(+)的优先级是6。（其他信息我们稍后介绍。）“infixl 7 *”表示(*)的优先级为7。由于(*)比(+)优先级高，所以我们看到为什么 $1 + 4 * 4$ 和 $1 + (4 * 4)$ 值相同而不是 $(1 + 4) * 4$ 。

Haskell也定义了操作符的结合性(associativity)。它决定了当一个表达式中多次出现某个操作符时是否是从左到右求值。(+)和(*)都是左结合，在上述的ghci输出结果中以infixl表示。一个右结合的操作符会以infixr表示。

```
ghci> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a -- Defined in GHC.Real
infixr 8 ^
```

优先级和结合性规则的组合通常称之为固定性(fixity)规则。

未定义的变量以及定义变量

Haskell的标准库prelude定义了至少一个大家熟知的数学常量。

```
ghci> pi
3.141592653589793
```

然后我们很快就会发现它对数学常量的覆盖并不是很广泛。让我们看下Euler数，e。

```
ghci> e
<interactive>:1:0: Not in scope: `e`
```

啊哈，看上去我们必须得自己定义。

不要担心错误信息
以上“not in the scope”的错误信息看上去有点令人畏惧的。别担心，它所表达的只是没有用e这个名字定义过变量。

使用ghci的let构造器(construct)，我们可以定义一个临时变量e。

```
ghci> let e = exp 1
```

这是指数函数 \exp 的一个应用，也是如何调用一个Haskell函数的第一个例子。像Python这些语言，函数的参数是位于括号内的，但Haskell不要那样。

既然e已经定义好了，我们就可以在数学表达式中使用它。我们之前用到的乘方操作符(\wedge)是对于整数的。如果要用浮点数作为指数，则需要操作符(**)。

```
ghci> (e ** pi) - pi
19.99909997918947
```

Note

这是ghci的特殊语法

ghci 中 let 的语法和常规的“top level”的Haskell程序的使用不太一样。我们会在章节“初识类型”里看到常规  v: latest ▼

处理优先级以及结合性规则

有时候最好显式地加入一些括号，即使Haskell允许省略。它们会帮助将来的读者，包括我们自己，更好的理解代码的意图。

更加重要的，基于操作符优先级的复杂的表达式经常引发bug。对于一个简单的、没有括号的表达式，编译器和人总是很容易的对其意图产生不同的理解。

不需要去记住所有优先级和结合性规则：在你不确定的时候，加括号是最简单的方法。

ghci里的命令行编辑

在大多数系统中，**ghci**有些命令行编辑的功能。如果你对命令行编辑还不熟悉，它将会帮你节省大量的时间。基本操作对于类Unix系统和Windows系统都很常规。按下**向上**方向键会显示你输入的上一条命令；重复输入**向上**方向键则会找到更早的一些输入。可以使用**向左**和**向右**方向键在当前行移动。在类Unix系统中(很不幸，不是Windows)，**制表键**(tab)可以完成输入了一部分的标示符。

[译者注：]制表符的完成功能其实在Windows下也是可以的。

Tip

哪里可以找到更多信息

我们只是蜻蜓点水般的介绍了下命令行编辑功能。因为命令行编辑系统可以让你更加有效的工作，你可能会觉得进一步的学习会有帮助。

在类Unix系统下，**ghci**使用功能强大并且可定制化的**GNU readline library**。在Windows系统下，**ghci**的命令行编辑功能是由**doskey command**提供的。

列表(Lists)

一个列表由方括号以及被逗号分隔的元素组成。

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

Note

逗号是分隔符，不是终结符

有些语言在表示列表时会在右中括号前多一个逗号，但是Haskell没有这样做。如果多出一个逗号(比如 `[1, 2,]`)，则会导致编译错误。

列表可以是任意长度。空列表表示成`[]`。

```
ghci> []
[]
ghci> ["foo", "bar", "baz", "quux", "fnord", "xyzzy"]
["foo", "bar", "baz", "quux", "fnord", "xyzzy"]
```

列表里所有的元素必须是相同类型。下面例子我们违反了 this 规则：列表中前面两个是Bool类型，最后一个是字符类型。

```
ghci> [True, False, "testing"]

<interactive>:1:14:
    Couldn't match expected type `Bool' against inferred type `[Char]'
      Expected type: Bool
      Inferred type: [Char]
    In the expression: "testing"
    In the expression: [True, False, "testing"]
```

 v: latest ▾

这次**ghci**的错误信息也是同样的很详细。它告诉我们无法把字符串转换为布尔类型，因此无法定义这个列表表达式的类型。

如果用**列举符号(enumeration notation)**来表示一系列元素，Haskell则会自动填充内容。

```
ghci> [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

字符`..`在这里表示列举(enumeration)。它只能用于那些可以被列举的类型。因此对于字符类型来说这就没意义了。比如对于`["foo".."quux"]`，没有任何意思，也没有通用的方式来对其进行列举。

顺便提一下，上面例子生成了一个闭区间，列表包含了两个端点的元素。

当使用列举时，我们可以通过最初两个元素之间步调的大小，来指明后续元素如何生成。

```
ghci> [1.0, 1.25..2.0]
[1.0, 1.25, 1.5, 1.75, 2.0]

ghci> [1, 4..15]
[1, 4, 7, 10, 13]

ghci> [10, 9..1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

上述的第二个例子中，终点元素并未包含的列表内，是由于它不属于我们定义的系列元素。

我们可以省略列举的终点(end point)。如果类型没有自然的“上限”(upper bound)，那么会生成无穷列表。比如，如果在`ghci`终端输入`[1..]`，那么就会输出一个无穷的连续数列，因此你不得不强制关闭或是杀掉`ghci`进程。在后面的章节章节中我们会看在Haskell中无穷数列经常会用到。

Note

列举浮点数时要注意的

下面的例子看上并不那么直观

```
ghci> [1.0..1.8]
[1.0, 2.0]
```

为了避免浮点数舍入的问题，Haskell就从`1.0`到`1.8+0.5`进行了列举。

对浮点数的列举有时候会有点特别，如果你不得不用，要注意。浮点数在任何语言里都显得有些怪异(quirky)，Haskell也不例外。

列表的操作符

有两个常见的用于列表的操作符。**连接两个列表时使用`(++)`**。

```
ghci> [3, 1, 3] ++ [3, 7]
[3, 1, 3, 3, 7]
ghci> [] ++ [False, True] ++ [True]
[False, True, True]
```

更加基础的操作符是`(:)`，用于**增加一个元素到列表的头部**。它读成“cons”（即“construct”的简称）。

```
ghci> 1 : [2, 3]
[1, 2, 3]
ghci> 1 : []
[1]
```

你可能会尝试`[1, 2]:3`给列表末尾增加一个元素，然而`ghci`会拒绝这样的表达式并给出错误信息，因为`(:)`的第一个参数必须是单个元素同时第二个必须是一个列表。

 v: latest ▾

字符串和字符

如果你熟悉Perl或是C语言，你会发现Haskell里表示字符串的符号很熟悉。

双引号所包含的就表示一个文本字符串。

```
ghci> "This is a string."  
"This is a string."
```

像其他语言一样，那些不显而易见的字符(hard-to-see)需要“转意”(escaping)。Haskell中需要转意的字符以及转意规则绝大部分是和C语言中的情况一样的。比如 `'\n'` 表示换行，`'\t'` 表示制表符。完整的详细列表可以参照[附录B：字符，字符串和转意规则](#)。

```
ghci> putStrLn "Here's a newline -->\n<-- See?"  
Here's a newline -->  
<-- See?
```

函数`putStrLn`用于打印一个字符串。

Haskell区分单个字符和文本字符串。单个字符用单引号包含。

```
ghci> 'a'  
'a'
```

事实上，文本字符串是单一字符的列表。下面例子展示了表示一个短字符串的痛苦方式，而`ghci`的显示结果却是我们很熟悉的形式。

```
ghci> let a = ['l', 'o', 't', 's', ' ', 'o', 'f', ' ', 'w', 'o', 'r', 'k']  
ghci> a  
"lots of work"  
ghci> a == "lots of work"  
True
```

`""`表示空字符串，它和`[]`同义。

```
ghci> "" == []  
True
```

既然字符串就是单一字符的列表，那么我们就可以用列表的操作符来构造一个新的字符串。

```
ghci> 'a' : "bc"  
"abc"  
ghci> "foo" ++ "bar"  
"foobar"
```

初识类型

尽管前面的内容里提到了一些类型方面的事情，但直到目前为止，我们还没有使用`ghci`进行过任何类型方面的交互：即使不告诉`ghci`输入是什么类型，它也会很高兴地接受传给它的输入。

需要提醒的是，在Haskell里，所有类型名字都以大写字母开头，而所有变量名字都以小写字母开头。紧记这一点，你就不会弄错类型和变量。

我们探索类型世界的第一步是修改`ghci`，让它在返回表达式的求值结果时，打印出这个结果的类型。使用`ghci`的`:set`命令可以做到这一点：

```
Prelude> :set +t  
  
Prelude> 'c'      -- 输入表达式  
'c'             -- 输出值  
it :: Char       -- 输出值的类型  
  
Prelude> "foo"  
"foo"  
it :: [Char]
```

 v: latest ▾

注意打印信息中那个神秘的 `it`：这是一个有特殊用途的变量，`ghci` 将最近一次求值所得的结果保存在这个变量里。（这不是 Haskell 语言的特性，只是 `ghci` 的一个辅助功能而已。）

Ghci 打印的类型信息可以分为几个部分：

它打印出 `it`

`x :: y` 表示表达式 `x` 的类型为 `y`

第二个表达式的值的类型为 `[Char]`。（类型 `String` 是 `[Char]` 的一个别名，它通常用于代替 `[Char]`。）

以下是另一个我们已经见过的类型：

```
Prelude> 7 ^ 80
40536215597144386832065866109016673800875222251012083746192454448001
it :: Integer
```

Haskell 的整数类型为 `Integer`。`Integer` 类型值的长度只受限于系统的内存大小。

分数和整数看上去不太相同，它使用 `%` 操作符构建，其中分子放在操作符左边，而分母放在操作符右边：

```
Prelude> :m +Data.Ratio
Prelude Data.Ratio> 11 % 29
11 % 29
it :: Ratio Integer
```

这里的 `:m` 是 `:module` 的缩写，用于载入一个给定模块。`Ghci` 还提供了很多这类缩写，方便使用者。

为了方便起见，`ghci` 给很多命令都提供了缩写，这里的 `:m` 就是 `:module` 的缩写，它用于载入给定的模块。

注意这个分数的类型信息：在 `::` 的右边，有两个单词，分别是 `Ratio` 和 `Integer`，可以将这个类型读作“由整数构成的分数”。这说明，分数的分子和分母必须都是整数类型，如果用一些别的类型值来构建分数，就会造成出错：

```
Prelude Data.Ratio> 3.14 % 8

<interactive>:8:1:
  Ambiguous type variable `a0' in the constraints:
    (Fractional a0)
      arising from the literal `3.14' at <interactive>:8:1-4
    (Integral a0) arising from a use of `%` at <interactive>:8:6
    (Num a0) arising from the literal `8' at <interactive>:8:8
  Probable fix: add a type signature that fixes these type variable(s)
  In the first argument of `(%)`, namely `3.14'
  In the expression: 3.14 % 8
  In an equation for `it': it = 3.14 % 8

Prelude Data.Ratio> 1.2 % 3.4

<interactive>:9:1:
  Ambiguous type variable `a0' in the constraints:
    (Fractional a0)
      arising from the literal `1.2' at <interactive>:9:1-3
    (Integral a0) arising from a use of `%` at <interactive>:9:5
  Probable fix: add a type signature that fixes these type variable(s)
  In the first argument of `(%)`, namely `1.2'
  In the expression: 1.2 % 3.4
  In an equation for `it': it = 1.2 % 3.4
```

尽管每次都打印出值的类型很方便，但这实际上有点小题大作了。因为在一般情况下，表达式的类型并不难猜，或者我们并非对每个表达式的类型都感兴趣。所以这里用 `:unset` 命令取消对类型信息的打印：

```
Prelude Data.Ratio> :unset +t

Prelude Data.Ratio> 2
2
```

 v: latest ▾

取而代之的是，如果现在我们对某个值或者表达式的类型不清楚，那么可以用 `:type` 命令显式地打印它的类型信息：

```
Prelude Data.Ratio> :type 'a'
'a' :: Char

Prelude Data.Ratio> "foo"
"foo"

Prelude Data.Ratio> :type it
it :: [Char]
```

注意 `:type` 并不实际执行传给它的表达式，它只是对输入进行检查，然后将输入的类型信息打印出来。以下两个例子显示了其中的区别：

```
Prelude Data.Ratio> 3 + 2
5

Prelude Data.Ratio> :type it
it :: Integer

Prelude Data.Ratio> :type 3 + 2
3 + 2 :: Num a => a
```

在前两个表达式中，我们先求值 `3+2`，再使用 `:type` 命令打印 `it` 的类型，因为这时 `it` 已经是 `3+2` 的结果 `5`，所以 `:type` 打印这个值的类型 `it :: Integer`。

另一方面，最后的表达式中，我们直接将 `3+2` 传给 `:type`，而 `:type` 并不对输入进行求值，因此它返回表达式的类型 `3 + 2 :: Num a => a`。

第六章会介绍更多类型签名的相关信息。

行计数程序

以下是一个用 Haskell 写的行计数程序。如果暂时看不太懂源码也没关系，先照着代码写写程序，热热身就行了。

使用编辑器，输入以下内容，并将它保存为 `WC.hs`：

```
-- file: ch01/WC.hs
-- lines beginning with "--" are comments.

main = interact wordCount
  where wordCount input = show (length (lines input)) ++ "\n"
```


再创建一个 `quux.txt`，包含以下内容：

```
Teignmouth, England
Paris, France
Ulm, Germany
Auxerre, France
Brunswick, Germany
Beaumont-en-Auge, France
Ryazan, Russia
```

然后，在 shell 执行以下代码：

```
$ runghc WC < quux.txt
7
```

恭喜你！你刚完成了一个非常有用的行计数程序（尽管它非常简单）。后面的章节会继续介绍更多有用的知识，帮助你（读者）写出属于自己的程序。

[译注：可能会让人有点迷惑，这个程序明明是一个行计数（line count）程序，为什么却命名为 WC（word count）呢？实际上，在接下来的练习小节中，读者需要对这个程序进行修改，将它的功能从行计数改为单词计数，因此这里程序被命名为 `WC.hs`， [v: latest](#) ▼]

练习

1. 在ghci里尝试下以下的这些表达式看看它们的类型是什么？

```
5 + 8
3 * 5 + 8
2 + 4
(+) 2 4
sqrt 16
succ 6
succ 7
pred 9
pred 8
sin (pi / 2)
truncate pi
round 3.5
round 3.4
floor 3.7
ceiling 3.3
```

2. 在ghci里输入:?
3. 函数words计算一个字符串中的单词个数。修改例子WC.hs，使得可以计算一个文件中的单词个数。
4. 再次修改WC.hs，可以输出一个文件的字符个数。

讨论

0条评论

Real World Haskell 中文版

1 登录

推荐 1 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

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。

第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

Wengel An — 平面上任意三个点a,b,c，假设固定a,b，c点出现的位置只有三种可能，在ab线段所在直线的右边，在ab线段所在直线的左边，或者在ab线段所在的直线上

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