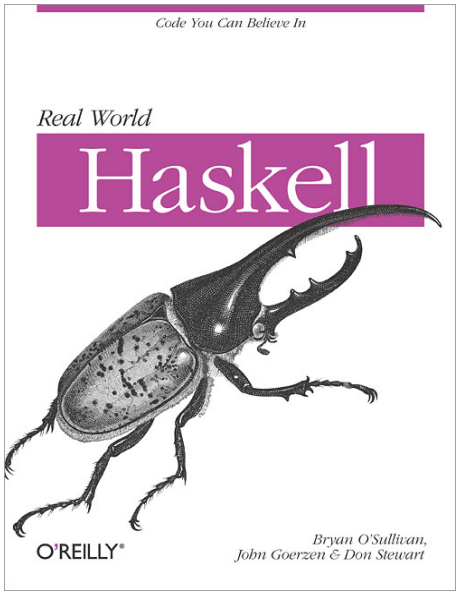
Real World Haskell 中文版 Documentation

Release 1.0

huangz1990

Oct 21, 2019

CONTENTS



本文档是《Real World Haskell》一书的简体中文翻译版本，翻译工作正在进行中，欢迎加入：https://github.com/huangz1990/real-world-haskell-cn

CHAPTER

ONE

第 1 章：入门

1.1 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 是一个解释器，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** 是生成本地原生代码的优化编译器。
- **ghci** 是一个交互解释器和调试器。
- **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。

1.2 初识解释器 ghci

ghci 程序是 GHC 的交互式解释器。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 的交互式解释器。

```
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 pieces of code, but it␣
↪can initially be surprising if you are used to , say, the interactive Python␣
↪interpreter.
```

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

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
```

现在我们就可以使用 `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。

1.3 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释器，也可以帮助我们了解更多关于 Haskell 是如何工作的。

1.3.1 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
```

1.3.2 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些怪异，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。

1.3.3 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 `True`。

```
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
```

我们再一次看到了很翔实的错误信息。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。即使遇到自己不理解的问题或者错误信息，也不必惊慌。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 ≠。

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
```

1.3.4 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
  ...
    -- 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。

1.3.5 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。让我们来看下 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，它所要表达的只是没有用 e 这个名字定义过变量。

使用 **ghci** 的 let 构造 (contruct)，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 (^) 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”里看到常规的语法形式。

1.3.6 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。

1.4 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。

1.5 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。在下面的例子中我们违反了这个规则：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"]
```

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。

1.5.1 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。

1.6 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，字符串和转义规则。

```
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"
```

1.7 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]
```

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
```

为了方便用户，ghci 允许我们对很多命令进行缩写，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:
```

(continues on next page)

(continued from previous page)

```
        (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
```

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。

1.8 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` 。]

1.9 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**:?** 以或许帮助信息。定义一个变量，比如 `let x = 1`, 然后输入`:show bindings`. 你看到了什么?
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。

CHAPTER

TWO

第 2 章：类型和函数

2.1 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约数据当作汽车租凭数据来使用。

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。

2.2 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。

2.2.1 强类型

Haskell 的强类型系统会拒绝执行任何无意义的表达式，保证程序不会因为这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。]

2.2.2 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 编译器或解释器会察觉出类型不正确的表达式，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 x 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。

2.2.3 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。

2.3 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一一展示出来，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。

2.4 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
```

2.5 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用逗号来隔开 [译注：使用空格就可以了]：

```
Prelude> compare 2 3
LT

Prelude> compare 3 3
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
```

2.6 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"
```

(continues on next page)

(continued from previous page)

```
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（notion）。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)

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。

2.7 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'
```

2.7.1 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"
```

2.8 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"]
```

结果表示，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”。]

2.9 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
```

2.10 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.

*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。）

2.10.1 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不刻都处在改变当中。

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.
```

2.10.2 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]
[]

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。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
    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
```

2.11 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。

在前面的章节里多次谈到，可以使用一个表达式去代换一个变量。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。

2.11.1 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（thunk）。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。]

2.11.2 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
```

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") 。

2.11.3 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"
```

(continues on next page)

(continued from previous page)

```
*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")`。

2.11.4 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 `(||)` 操作符的左操作对象求值：

```
*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值为 `True`。于是 `then` 分支被求值：

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

2.11.5 从递归中返回

请注意，在求值的最后一步，结果表达式 `tail "bcd"` 处于第二次对 `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。

2.11.6 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，也只执行可以确立出（establish）值的那部分表达式。[译注：比如之前提到的，`(||)` 的左操作对象的值为 `True` 时，就无需对右操作对象估值的情况。]
- 函数的返回值可能是一个块（一个被延迟计算的表达式）。

2.12 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
```

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 代换 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绑定到一个实际的值一样，Haskell 的类型也可以带有参数，并且在稍后可以将这些参数绑定到其它实际的类型上。

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。

2.12.1 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。

2.12.2 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。

2.13 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 从右到左地串联起这些箭头，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，但要说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]
```

2.14 练习

1.Haskell 提供了一个标准函数, last :: [a] -> a, 其返回一个列表的最后一个元素. 单就从他的类型看, 这个函数拥有的合理行为是怎样的 (忽略崩溃和无限循环)? 有哪些少数的事情这个函数显然不能做到?

2. 写一个函数 lastButOne, 返回列表倒数第二个元素.

3. 加载 lastButOne 函数到 ghci, 并且尝试在不同长度的列表上测试. 当你传入的函数过短时候，发生了什么?

2.15 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：

```
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有的合法行为：

- 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：每个函数都是自包容的，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。

2.16 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。

CHAPTER

THREE

第 3 章：定义类型并简化函数

3.1 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 9780135072455 "Algebra of Programming" ["Richard Bird","Oege de Moor"]
```

在 ghci 里面当然也可以创建新的 BookInfo 值：

```
*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
```

3.1.1 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 。

3.2 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🄵嗦的类型设置一个更短的名字：

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

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` 。]

3.3 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于一个值构造器：

```
-- 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"]
```

(continues on next page)

(continued from previous page)

```
*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。]

3.3.1 什么情况下该用元组，而什么情况下又该用代数数据类型？

元组和自定义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.
```

(continues on next page)

(continued from previous page)

```
*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.
```

(continues on next page)

(continued from previous page)

```
*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。

3.3.2 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

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 roygbiv {
    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
```

(continues on next page)

(continued from previous page)

```
              | 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 。

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 自己有办法弄清楚一点，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 。]

3.4 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，那么应该有办法提取这些成分。

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: ch03/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:[]
```

(continues on next page)

(continued from previous page)

```
[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，它和我们定以的 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
```

3.4.1 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` 。
- 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。]

3.4.2 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
```

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。

3.4.3 模式匹配中的变量名命名

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。

3.4.4 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，它也叫做通配符。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。

3.4.5 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
```

3.5 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同时，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
```

(continues on next page)

(continued from previous page)

```
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。

```
-- 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 类型的值来做个比较；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。

3.6 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。

```
-- 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。

3.7 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`。

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)
```

3.7.1 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。

3.8 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
```

3.8.1 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
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。

3.9 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 *最顶层* 的作用域。

3.9.1 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
      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。

3.9.2 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。

3.9.3 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"
          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内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"
```

3.10 表达式里的缩进规则和空白字符

请看 `lend` 和 `lend2` 的定义表达式，左侧空出了一大块。这并不是随随便便写的，这些空白字符是有意义的。

Haskell 依据缩进来解析代码块。这种用排版来表达逻辑结构的方式通常被称作*缩进规则*。在源码文件开始的那一行，首个顶级声明或者定义可以从该行的任意一列开始，Haskell 编译器或解释器将记住这个缩进级别，并且随后出现的所有顶级声明也必须使用相同的缩进。

以下是一个顶级缩进规则的例子。第一个文件 `GoodIndent.hs` 执行正常。

```
-- file: ch03/GoodIndent.hs
-- 这里是最左侧一列

    -- 顶级声明可以从任一列开始
    firstGoodIndentation = 1

    -- 只要所有后续声明也这么做!
    secondGoodIndentation = 2
```

第二个文件 `BadIndent.hs` 没有遵守规则，因此也不能正常执行。

```
-- file: ch03/BadIndent.hs
-- 这里是最左侧一列

    -- 第一个声明从第 4 列开始
    firstBadIndentation = 1

-- 第二个声明从第 1 列开始，这样是非法的!
secondBadIndentation = 2
```

如果我们尝试在 **ghci** 里加载这两个文件，结果如下。

```
ghci> :load GoodIndent.hs
[1 of 1] Compiling Main             ( GoodIndent.hs, interpreted )
Ok, modules loaded: Main.
ghci> :load BadIndent.hs
[1 of 1] Compiling Main             ( BadIndent.hs, interpreted )

BadIndent.hs:8:2: parse error on input `secondBadIndentation'
Failed, modules loaded: none.
```

紧跟着的（译注：一个或多个）空白行将被视作当前行的延续，比当前行缩进更深的紧跟着的行也是如此。

let 表达式和 where 从句的规则与此类似。一旦 Haskell 编译器或解释器遇到一个 let 或 where 关键字，就会记住接下来第一个标记（token）的缩进位置。然后如果紧跟着的行是空白行或向右缩进更深，则被视作是前一行的延续。而如果其缩进和前一行相同，则被视作是同一区块内的新的一行。

```
-- file: ch03/Indentation.hs
foo = let firstDefinition = blah blah
          -- 只有注释的行被视作空白行
                  continuation blah

          -- 减少缩进，于是下面这行就变成了一行新定义
          secondDefinition = yada yada

                  continuation yada
      in whatever
```

下面的例子演示了如何嵌套使用 let 和 where。

```
-- file: ch03/letwhere.hs
bar = let b = 2
          c = True
      in let a = b
         in (a, c)
```

变量 a 只在内部那个 let 表达式中可见。它对外部那个 let 是不可见的。如果我们在外部使用变量 a 就会得到一个编译错误。缩进为我们和编译器提供了视觉上的标识，让我们可以一眼就看出来作用域中包含哪些东西。

```
-- file: ch03/letwhere.hs
foo = x
    where x = y
            where y = 2
```

于此类似，第一个 where 从句的作用域即是定义 foo 的表达式，而第二个 where 从句的作用域则是第一个 where 从句。

在 let 和 where 从句中妥善地使用缩进能够更好地展现代码的意图。

3.10.1 对制表符和空格说两句

在使用诸如 Emacs 这种能够识别 Haskell 代码的编辑器时，通常的默认配置是使用空格来表示缩进。如果你的编辑器不能识别 Haskell，建议您手工设置使用空格来表示缩进。

这么做的好处是可移植性。在一个使用等宽字体的编辑器里，类 Unix 系统和 Windows 系统对制表符的默认显示宽度并不一样，前者相当于 8 个字符宽度，而后者则相当于 4 个。这就意味着无论你自己认为制表符应该显示为多宽，你并不能保证别人的编辑器设置会尊重你的看法。使用制表符来表示缩进一定会在某些人的设置那里看起来一团糟。甚至还有可能导致编译错误，因为 Haskell 语言标准的实现是按照 Unix 风格制表符宽度来的。而使用空格则完全不用担心这个问题。

3.10.2 缩进规则并不是必需

我们也可以使用显式的语法结构来代替排版，从而表达代码的意图。例如，我们先写一个左大括号，然后写几个赋值等式，每个等式之间用分号分隔，最后加上一个右大括号。下面这个两个例子所表达的意图是一模一样的。

```
-- file: ch03/Braces.hs
bar = let a = 1
          b = 2
          c = 3
      in a + b + c

foo = let { a = 1;  b = 2;
        c = 3 }
      in a + b + c
```

当我们使用显式语法结构时，普通的排版不再起作用。就像在第二个例子中那样，我们可以随意的使用缩进而仍然保持正确。

显式语法结构可以用在任何地方替换普通排版。它对 where 从句乃至顶级声明也是有效的。但是请记住，尽管你可以这样用，但在 Haseklll 编程中几乎没有人会使用显式语法结构。

3.11 Case 表达式

函数定义并不是唯一我们能使用模式匹配的地方。case 结构使得我们还能在一个表达式内部使用模式匹配。如下面的例子所示，这个函数（定义在 Data.Maybe 里）能解构一个 Maybe 值，如果该值为 Nothing 还可以返回默认值。

```
-- file: ch03/Guard.hs
fromMaybe defval wrapped =
    case wrapped of
      Nothing     -> defval
      Just value  -> value
```

case 关键字后面可以跟任意表达式，这个表达式的结果即是模式匹配的目标。of 关键字标识着表达式到此结束，以及匹配区块的开始，这个区块将用来定义每种模式及其对应的表达式。

该区块的每一项由三个部分组成：一个模式，接着一个箭头 ->，接着一个表达式；如果这个模式匹配中了，则计算并返回这个表达式的结果。这些表达式应当是同一类型的。第一个被匹配中的模式对应的表达式的计算结果即是 case 表达式的结果。匹配按自上而下的优先级进行。

如果使用通配符 _ 作为最后一个被匹配的模式，则意味着“如果上面的模式都没有被匹配中，就使用最后这个表达式”。如果所有模式匹配都没中，我们会看到前面章节里出现过的运行时错误。

3.12 新手在使用模式时常见的问题

在某些情况下，Haskell 新手会误解或误用模式。下面就是一些错误地使用模式匹配的例子。建议阅读时先自己想一想期望的结果是什么，然后看看实际的结果是否出乎你的意料。

3.12.1 错误地对变量进行匹配

```
-- file: ch03/BogusPattern.hs
data Fruit = Apple | Orange
             deriving (Show)

apple = "apple"

orange = "orange"

```

(continues on next page)

(continued from previous page)

```
whichFruit :: String -> Fruit

whichFruit f = case f of
                 apple  -> Apple
                 orange -> Orange
```

随意一瞥，这段代码的意思似乎是要检查 f 的值是 apple 还是 orange。

换一种写法可以让错误更加显而易见。(译注：原文的例子太晦涩，换了评论中一个较清楚的例子)

```
-- file: ch03/BogusPattern.hs
whichFruit2 :: String -> Fruit
whichFruit2 apple = Apple
whichFruit2 orange = Orange
```

这样写明白了吗？就是这里，显然 apple 并不是指在上层定义的那个变量名为 apple 的值，而是当前作用域的一个模式变量。

Note: 不可拒绝的模式

我们把这种永远会成功匹配的模式称作不可拒绝。普通的变量名和通配符 _ 都属于不可拒绝的模式。

上面那个函数的正确写法应该如此：

```
-- file: ch03/BogusPattern.hs
betterFruit f = case f of
                  "apple"  -> Apple
                  "orange" -> Orange
```

我们针对字符串值 "apple" 和 "orange" 进行模式匹配，于是问题得到了解决。

3.12.2 进行了错误的相等比较

如果我们想比较一个 Tree 类型的两个节点值，如果相等就返回其中一个应该怎么做？请看下面这个尝试。

```
-- file: ch03/BadTree.hs
bad_nodesAreSame (Node a _ _) (Node a _ _) = Just a
bad_nodesAreSame _            _            = Nothing
```

一个命名在一组模式绑定中只能使用一次。将同一个变量放在多个位置并不意味着 “这些变量应该相等”。要解决这类问题，我们需要使用 Haskell 的另一个重要的特性，守卫。

3.13 使用守卫实现条件求值

模式匹配针对的是值长成什么样子，因此有着诸多局限。除此之外，我们常常需要在对函数体求值之前进行各种各样的检查。Haskell 也为此提供了守卫这个特性。我们将通过改写上面这个用来比较树的两个节点是否相等的这段函数来讲解它的使用方法。

```
-- file: ch03/BadTree.hs
nodesAreSame (Node a _ _) (Node b _ _)
    | a == b     = Just a
nodesAreSame _ _ = Nothing
```

在上面的例子中，我们首先使用了模式匹配已确保值的形式是对的，然后使用了一个守卫来比较其中的一个部分。

一个模式后面可以跟着 0 个或多个 `Bool` 类型的守卫。我们引入了 `|` 这个符号来标识守卫的使用。即在它后面写一个守卫表达式，再写一个 `=` 符号（如果是 `case` 表达式则应该用 `->`），然后写上在这个守卫表达式的值为真的情况下将被使用的函数体。如果某个模式匹配成功，那么它后面的守卫将按照顺序依次被求值。如果其中的某个守卫值为真，那么使用它所对应的函数体作为结果。如果所有的守卫值都为假，那么模式匹配继续进行，即尝试匹配下一个模式。

当对守卫表达式求值时，它所对应的模式中提到的所有变量都会被绑定，因此能够使用。

下面是一个用守卫重构过的 `lend` 方法。

```
-- file: ch03/Lending.hs
lend3 amount balance
     | amount <= 0            = Nothing
     | amount > reserve * 0.5 = Nothing
     | otherwise              = Just newBalance
    where reserve    = 100
          newBalance = balance - amount
```

`otherwise` 看上去像是守卫表达式的某种特殊语法，但实际上它只是一个被绑定为值 `True` 的普通变量，这样写是为了提高可读性。

任何使用模式的地方都可以使用守卫。用模式匹配和守卫把方法写成一组等式会大大提高可读性。还记得我们在*条件求值 <>_* 一节中定义的 `myDrop` 方法吗?

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

下面是用模式和守卫重构过的例子。

```haskell
-- file: ch02/myDrop.hs
niceDrop n xs | n <= 0 = xs
niceDrop _ []          = []
niceDrop n (_:xs)      = niceDrop (n - 1) xs
```

这种格式的代码清楚的依次列举了我们对函数在不同情况下的不同用法的期望。如果使用 `if` 表达式，我们的想法很容易就被藏进某个函数里，而代码则会变得不可读。

3.14 练习

1. 写一个函数，用来计算一个列表元素的个数．出于测试要求，保证其输出的结果和标准函数 length 保持一致.

2. 添加函数的类型签名于你的源文件．出于测试要求，再次加载源文件到 ghci.

3. 写一个函数，用来计算列表的平均值，即，列表元素的总和除以列表的长度．(你可能需要用到 fromIntegral 函数将列表长度变量从 integer 类型到 float 类型进行转换．)

4. 将一个列表变成回文序列，即，他应该读起来完全一样，不管是从前往后还是从后往前．举个例子，考虑一个列表 [1,2,3]，你的函数应该返回 [1,2,3,3,2,1].

5. 写一个函数，用来确定他的输入是否是一个回文序列.

6. 创造一个函数，用于排序一个包含许多列表的列表，其排序规则基于他的子列表的长度．(你可能要看看 Data.List 模块的 sortBy 函数.)

7. 定义一个函数，其用一个分隔符将一个包含许多列表的列表连接在一起．函数类型定义如下:

```haskell
-- file: ch03/Intersperse.hs
intersperse :: a -> [[a]] -> [a]
```

这个分割应该出现于列表的元素之间，除了最后一个元素末尾之外．你的函数需运行得如下所示:

```
ghci> :load Intersperse
[1 of 1] Compiling Main             ( Intersperse.hs, interpreted )
Ok, modules loaded: Main.
ghci> intersperse ',' []
""
ghci> intersperse ',' ["foo"]
"foo"
ghci> intersperse ',' ["foo","bar","baz","quux"]
"foo,bar,baz,quux"
```

8. 使用我们在前面章节中定义的二叉树类型，写一个函数用于确定一棵二叉树的高度．高度的定义是从根节点到叶子节点经过的最大节点数．举个例子，`Empty` 这棵树的高度是 0; `Node "x" Empty Empty` 这棵树

的高度是 1; `Node "x" Empty (Node "y" Empty Empty)` 这棵树的高度是 2; 依此类推.

9. 考虑三个二维的点 a, b，和 c. 如果我们观察沿着线段 A B（由 a,b 节点组成）和线段 B C（由 b,c 节点组成）形成的角度，它或者转向（turn）左边，或者转向右边，或者组成一条直线. 定义一个 Direction（方向）的数据类型反映这些可能的情况.

10. 写一个函数，用于计算三个二维坐标点组成的转向（turn），并且返回其 Direction（方向）.

11. 定义一个函数，输入二维坐标点的序列并计算其每个连续三个的（方向）Direction. 考虑一个点的序列 [a,b,c,d,e]，他应该开始计算 [a,b,c] 的转向 (turn), 接着计算 [b,c,d] 的转向，再是 [c,d,e] 的. 你的函数应该返回一个 Direction（方向）的序列.

12. 运用前面三个练习的代码，实现 Graham 扫描算法，用于扫描由二维点集构成的凸包. 你能从**‘Wikipedia<http://en.wikipedia.org/wiki/Convex_hull>‘_** 上找到 " 什么是凸包 "，以及**‘ " Graham 扫描算法 " <http://en.wikipedia.org/wiki/Graham_scan>‘_** 的完整解释.

CHAPTER

FOUR

第 4 章：函数式编程

4.1 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。

4.2 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
```

(continues on next page)

(continued from previous page)

```
          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：

```
$ 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 。]

4.3 热身：方便地分离多行文本

Haskell 提供一个内建函数，lines，让我们在行边界上分离一段文本字符串。它返回一个忽略了行终止字符的字符串列表。

```
ghci> :type lines
lines :: String -> [String]
ghci> lines "line 1\nline 2"
["line 1","line 2"]
ghci> lines "foo\n\nbar\n"
["foo","","bar"]
```

尽管 lines 看上去有用，但它要工作的话仰赖于我们用“文本模式”去读一个文件。文本模式对于很多编程语言来说是一个很普遍的功能：当我们在 Windows 系统上读和写文件的时候它提供一个特别的行为。当我们用文本模式读取一个文件的时候，文件 I/O 库把行终止序列“\r\n”（回车后跟着一个换行）转换成“\n”（单独一个换行），当我们写一个文件的时候，它（译注：文件 I/O 库）会做相反的事情。而在类 Unix 系统上，文本模式不会做任何的转换工作。这个不同之处会导致的结果是，如果我们在 Windows 系统上读取一个在类 Unix 系统上写入的文件，行终止符很可能会变得乱七八糟。（readFile 和 writeFile 都在文本模式下操作的话）

```
ghci> lines "a\r\nb"
["a\r","b"]
```

lines 函数仅仅在换行符处分离文本，留下回车跟在行的结尾处。如果我们在 Linux 或 Unix 上读取一个 Windows 生成的文本文件，我们将在每一行的结尾处得到尾随的回车。

我们舒适地用了 Python 提供的“通用换行”支持很长时间了：它透明地为我们处理 Unix 和 Windows 的行终止惯例。我们也想要用 Haskell 提供类似的支持。

因为到目前为止我们仅仅接触了少量的 Haskell 代码，所以我们将一步一步地讨论我们用 Haskell 实现上述支持的细节。

```
-- file: ch04/SplitLines.hs
splitLines :: String -> [String]
```

我们的函数类型签名标示它接受一个字符串，可能是具有某些未知的行终止惯例的文件的内容。它返回一个包含字符串的列表，列表的每一项代表文件中的每一行文本。

```
-- file: ch04/SplitLines.hs
splitLines [] = []
splitLines cs =
    let (pre, suf) = break isLineTerminator cs
    in  pre : case suf of
                ('\r':'\n':rest) -> splitLines rest
                ('\r':rest)      -> splitLines rest
```

(continues on next page)

(continued from previous page)

```
                ('\n':rest)      -> splitLines rest
                _                -> []

isLineTerminator c = c == '\r' || c == '\n'
```

在我们深入细节之前，首先注意我们是怎么组织我们的代码的。我们首先呈现重要的代码片段，而把 isLineTerminator 函数的定义放在后面。因为我们给了这个辅助函数一个易读的名称，所以即使在我们读它的定义之前我们就能知道它是做什么用的。

Prelude 定义了一个叫做 break 的函数，我们可以用它把一个列表分成两部分。它把一个函数作为它的第一个参数。这个函数必须去检查列表中的元素，并且返回一个 Bool 值来表示列表是否在那个元素处被一分为二。break 函数返回一个对值（译注：即二元组），由一个谓词（译注：即一个返回 Bool 值的函数，下同）返回 True（译注：第一次返回 True）之前的元素构成的列表（前缀）和剩下的元素构成的列表组成（后缀）。

```
ghci> break odd [2,4,5,6,8]
([2,4],[5,6,8])
ghci> :module +Data.Char
ghci> break isUpper "isUpper"
("is","Upper")
```

因为我们一次只需要匹配单个的回车或换行，所以每次检查列表中的一个元素正是我们所需要的。

splitLines 的第一个算式标示如果我们匹配到一个空的字符串，我们就没有进一步的工作可做了。

在第二个算式中，我们首先对输入的字符串应用 break。前缀就是第一个行终止符之前的子字符串，后缀就是整个字符串余下的部分。这个后缀将包含可能存在的行终止符。

“pre :” 表达式告诉我们应该在这个代表文本行的列表的头部加上 pre 所表示的值。然后我们用一个 case 表达式去检查后缀，我们来决定下一步做什么。case 表达式的结果将被用作 (:) 函数的的二个参数来构造结果列表。

case 表达式中的第一个模式匹配以一个回车紧接着一个换行符开始的字符串。变量 rest 被绑定到这个字符串余下的部分。其它的模式是相似的，因此它们应当容易理解。

以上对 Haskell 函数的散文式的描述不一定是容易理解的。我们可以借助 ghci，观察不同情况下这个函数的行为，以获得更好的理解。

让我们从分割一个不包含任何行终止符的字符串开始。

```
ghci> splitLines "foo"
["foo"]
```

这里，我们的 break 没找到一个行终止符，所以它返回的后缀是空的。

```
ghci> break isLineTerminator "foo"
("foo","")
```

因此在 splitLines 函数中的 case 表达式一定是匹配上了第四个分支，然后完成了求值。再来一点儿更有趣的例子怎么样?

```
ghci> splitLines "foo\r\nbar"
["foo","bar"]
```

首先我们的 break 给我们一个非空的后缀。

```
ghci> break isLineTerminator "foo\r\nbar"
("foo","\r\nbar")
```

因为这个后缀以一个回车紧接着一个换行符开始，我们匹配上了 case 表达式的第一个分支。这个求值结果让我们把 pre 绑定到 “foo”，suf 绑定到 “bar”。我们递归地应用 splitLines，这一次匹配上单独的 “bar”。

```
ghci> splitLines "bar"
["bar"]
```

结果是我们构造了一个头部的元素是 “foo”，尾部的元素是 “bar” 的列表

```
ghci> "foo" : ["bar"]
["foo","bar"]
```

这种在 ghci 中的实验是一种有效的理解和调试一段代码的行为的方法。它甚至有一个更重要的好处就是非刻意的（译注：不是特别理解这句话在原英文语境中的意思，暂且按照网页中的批注直译过来，原句：It has an even more important benefit that is almost accidental in nature.）。从 ghci 测试复杂的代码是困难的，所以我们将倾向于写更小的函数。这能更进一步改善我们的代码的可读性。

这种创建和复用小的、强大的代码片段的方式是函数式编程的基础。

4.3.1 一个行终止转换程序

让我们把我们的 splitLines 函数和早先我们写的一个小框架联系起来。首先制作一份 Interact.hs 源文件的拷贝；让我们叫这个新文件 FixLines.hs。把 splitLines 加到新的源码文件中。因为我们的函数必须产出一个单独的字符串，所以我们必须把这个表示行的列表拼接起来。Prelude 提供一个 unlines 函数，它把一个字符串组成的列表串联起来，并且在每个字符串元素的末尾加上一个换行符。(译注：原文中代码注释命名有误，不是 SplitLines.hs 而是 FixLines.hs)

```
-- file: ch04/FixLines.hs
fixLines :: String -> String
fixLines input = unlines (splitLines input)
```

如果我们用 fixLines 替换这个 id 函数（译注：是指拷贝自 Interact.hs 的 FixLines.hs 中的 id 函数），我们可以把 FixLines.hs 编译成一个将文本文件的行终止转换成系统特定的行终止的可执行程序。

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

如果你在一个 Windows 系统上面，找到并下载一个在 Unix 系统上创建的文本文件（比如 gpl-3.0.txt）。在记事本程序中打开它。里面的文本行应该都跑一起去了，导致它几乎不可读。用刚才你编译得到的 FixLines 命令处理这个文件，然后在记事本程序中打开此命令输出的文件。现在这个行终止符的问题应该被修正了。

在类 Unix 的系统上，编辑器隐藏了 Windows 的行终止符。使得验证 FixLines 是否消除了它们更加困难。这里有一些命令应该能帮助你。

```
$ file gpl-3.0.txt
gpl-3.0.txt: ASCII English text
$ unix2dos gpl-3.0.txt
unix2dos: converting file gpl-3.0.txt to DOS format ...
$ file gpl-3.0.txt
gpl-3.0.txt: ASCII English text, with CRLF line terminators
```

4.4 中缀函数

通常，当我们在 Haskell 中定义和应用一个函数的时候，我们写这个函数的名称，紧接着它的参数。这种表示法作为前缀被提及，因为这个函数的名称位于它的参数前面。

如果一个函数或构造器带两个或更多的参数，我们可以选择使用中缀形式，即我们把函数（名称）放在它的第一个和第二个参数之间。这允许我们把函数作为中缀操作符来使用。

要用中缀表示法定义或应用一个函数或值构造器，我们用重音符（有时被称为反引号）包围它的名称。这里有简单的中缀函数和中缀类型的定义。

```
-- file: ch04/Plus.hs
a `plus` b = a + b

data a `Pair` b = a `Pair` b
                  deriving (Show)

-- we can use the constructor either prefix or infix
foo = Pair 1 2
bar = True `Pair` "quux"
```

因为中缀表示法纯粹是为了语法上的便利，因此它不会改变函数的行为。

```
ghci> 1 `plus` 2
3
ghci> plus 1 2
3
ghci> True `Pair` "something"
True `Pair` "something"
ghci> Pair True "something"
True `Pair` "something"
```

中缀表示法经常对代码可读性有帮助。比如，Prelude 定义了一个 elem 函数，它标示一个给定的值是否出现在一个列表中。如果我们用前缀表示法来使用 elem，它构成的代码相当易读。

```
ghci> elem 'a' "camogie"
True
```

如果我们换用中缀表示法，这段代码甚至会变得更容易理解。它清楚地标示我们正在检查左边的给定值是否出现在右边的列表里。

```
ghci> 3 `elem` [1,2,4,8]
False
```

我们在 Data.List 模块的一些有用的函数中看到了更明显的改进（译注：这里是指中缀表示法改进了代码可读性）。isPrefixOf 函数告诉我们一个列表是否匹配另一个列表的开始部分。

```
ghci> :module +Data.List
ghci> "foo" `isPrefixOf` "foobar"
True
```

相应地，isInfixOf 和 isSuffixOf 函数匹配一个列表的中间和结尾处的任何地方。

```
ghci> "needle" `isInfixOf` "haystack full of needle thingies"
True
ghci> "end" `isSuffixOf` "the end"
True
```

没有一个硬性的规则指示你什么时候应该用中缀表示法或是前缀表示法，尽管使用前缀表示法要普遍得多。在具体情况下选择使你的代码更可读的那一种表示法就是最好的。

4.5 和列表打交道

作为函数式编程的基本组件，列表应该得到足够的重视。Prelude 定义了很多函数来处理列表。它们当中的许多是不可或缺的工具，所以及早学习它们是很重要的。

不管怎样，这一节将学习很多函数。为什么要马上展示这么多函数？因为这些函数既容易学而且会经常使用。如果我们不知道有这样的工具箱，那么时间将浪费在编写那些在标准库中已经存在的简单函数上。因此深入学习列表模块中的函数是非常值得的。

Data.List 模块包含所有的列表函数。Prelude 只不过重新导出了这些在 Data.List 模块中的函数的大部分。还有一些有用的函数并没有被 Prelude 重新导出。下面学习列表函数的时候，将明确地提到那些只在 Data.List 中出现的。

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

因为这些函数没有什么复杂的或者代码量超过三行的，所有我们对它们只做简单的描述。实际上，快速和有用的学习方法是记住它们是如何定义的。

4.5.1 基本的列表操作

length 函数告诉我们一个列表中包含多少个元素。

```
ghci> :type length
length :: [a] -> Int
ghci> length []
0
ghci> length [1,2,3]
3
ghci> length "strings are lists, too"
22
```

如果需要检查列表是不是空的，用 null 函数。

```
ghci> :type null
null :: [a] -> Bool
ghci> null []
True
ghci> null "plugh"
False
```

要访问列表的第一个元素，用 head 函数。

```
ghci> :type head
head :: [a] -> a
ghci> head [1,2,3]
1
```

相反，tail 函数，返回列表中除了第一个其它所有的元素。

```
ghci> :type tail
tail :: [a] -> [a]
ghci> tail "foo"
"oo"
```

还有一个函数，last，返回列表的最后一个元素。

```
ghci> :type last
last :: [a] -> a
ghci> last "bar"
'r'
```

和 last 相反的是 init，它返回列表中除了最后一个其它所有的元素。

```
ghci> :type init
init :: [a] -> [a]
ghci> init "bar"
"ba"
```

上面的一些函数应用在空列表上时会报错，因此当不知道一个列表是否为空的时候要小心了。它们会报告什么样的错误呢?

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

在 ghci 里试一试上面的每一个函数，看看哪些应用在空列表上时会崩溃?

4.5.2 安全和明智地跟会崩溃的函数打交道

当我们想使用一个函数比如 head 时，如果我们传递一个空列表给它，很可能会破坏我们的工作，有效避免这个问题的方法是在使用 head 之前，检查传递给它的列表的长度。让我们举一个例子说明。

```
-- file: ch04/EfficientList.hs
myDumbExample xs = if length xs > 0
                   then head xs
                   else 'Z'
```

如果用过像 Perl 或者 Python 这样的编程语言，这段代码看起来就是一种很自然地编写测试的方式。在底层，Python 和 Perl 中的列表都是数组。所以它们必定能通过调用 len(foo) 或者是 scalar(@foo) 得知列表（数组）的长度。但是基于很多别的原因，把这些假设照搬到 Haskell 中不是一个好主意。

我们已经看过列表的数据类型定义好多次了，知道一个列表不会明确地存储它本身的长度。因此 length 函数的工作方式是遍历整个列表。

所以当我们只关心一个列表是不是为空时，调用 length 不是一个好的策略。如果我们处理的是一个有限的列表，它潜在地做了比我们想象的更多的事情。因为 Haskell 很容易创建无限列表，所以不小心使用了 length 函数可能导致无限循环。

一个更合适的替代方案是调用 null 函数，它执行的次数是确定不变的。更好的是，使用 null 能使我们的代码明确地标示出我们真正关心的列表的属性。下面举两个更好的例子。

```
-- file: ch04/EfficientList.hs
mySmartExample xs = if not (null xs)
                    then head xs
                    else 'Z'

myOtherExample (x:_) = x
myOtherExample [] = 'Z'
```

4.5.3 部分函数和全函数

如果一个函数只为合法输入的一个子集定义了返回值（函数返回的调用过程中产生的错误不属于返回值），这样的函数称作 *部分函数*（partial function）。类似的，对于整个输入域都能返回一个合法结果的函数，我们称之为 *全函数*（total function）。[译注：全函数是部分函数的特殊形式]

你应当了解你正在调用的函数是否为部分函数。对于部分函数而言，传入一个其无法处理的参数将导致错误，这也是 Haskell 程序能够简单明了避免错误的最主要的原因。[译注：为部分函数提供一个全函数的版本可能会导致问题处理层次的上浮，例如 head 函数在参数为空列表时抛出错误，如果我们定义 safeHead 函数对于空列表同样返回空列表，那么对于 head 函数所处的上一级函数，必须判断 head 函数返回的结果究竟是一个列表还是列表中的元素。]

有些使用 Haskell 的程序员会为部分函数加上 unsafe 的前缀，防止某些时候避免搬起石头砸自己的脚。标准的 `prelude` 定义了许多“不安全”的部分函数（例如 head），却没有提供等价的“安全”的全函数，这可以说是标准 `prelude` 的不足。[译注：此处存在争议，有人认为“不安全”的前缀指的是该函数可能会突破类型系统的限制，比如 unsafePerformIO，unsafeCoerce 等，它们可能会导致程序完全不可预知的行为，这与在空列表上调用 head 有很大的区别]

4.5.4 更多简单列表操作

Haskell 用 (++) 表示“追加”函数。

```
ghci> :type (++)
(++) :: [a] -> [a] -> [a]
ghci> "foo" ++ "bar"
"foobar"
ghci> [] ++ [1,2,3]
[1,2,3]
```

(continues on next page)

(continued from previous page)

```
ghci> [True] ++ []
[True]
```

concat 函数取一个包含列表的列表，这些列表中的元素具有相同的类型，它把这些列表连接在一起成为一个单一的列表。

```
ghci> :type concat
concat :: [[a]] -> [a]
ghci> concat [[1,2,3], [4,5,6]]
[1,2,3,4,5,6]
```

它会去掉一级的嵌套。（译注：每次调用 concat 会去除最外一层的方括号）

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

reverse 函数返回一个元素以相反的顺序排列的新列表。

```
ghci> :type reverse
reverse :: [a] -> [a]
ghci> reverse "foo"
"oof"
```

针对包含 Bool 值的列表，and 和 or 函数相当于用 && 和 || 遍历这个列表并两两求值

```
ghci> :type and
and :: [Bool] -> Bool
ghci> and [True,False,True]
False
ghci> and []
True
ghci> :type or
or :: [Bool] -> Bool
ghci> or [False,False,False,True,False]
True
ghci> or []
False
```

还有两个与 and 和 or 功能近似的函数，all 和 any，它们操作任何类型的列表。每一个带着一个谓词作为它的第一个参数；如果谓词对列表中的每个元素的判断都为真，all 函数返回 True，当对列表中的每个元素的谓词至少有一个成功了，any 函数返回 True。

```
ghci> :type all
all :: (a -> Bool) -> [a] -> Bool
ghci> all odd [1,3,5]
True
ghci> all odd [3,1,4,1,5,9,2,6,5]
False
ghci> all odd []
True
ghci> :type any
any :: (a -> Bool) -> [a] -> Bool
ghci> any even [3,1,4,1,5,9,2,6,5]
True
ghci> any even []
False
```

4.5.5 产生子列表

take 函数，在“函数应用”一节遇到过，返回一个由头 k 个元素组成的子列表。与它相反，drop，丢掉列表开头的 k 个元素。

```
ghci> :type take
take :: Int -> [a] -> [a]
ghci> take 3 "foobar"
"foo"
ghci> take 2 [1]
[1]
ghci> :type drop
drop :: Int -> [a] -> [a]
ghci> drop 3 "xyzzy"
"zy"
ghci> drop 1 []
[]
```

splitAt 函数组合了 take 和 drop 的功能，返回由一个列表产生的二元组，两部分是由原来的列表根据给定的索引分割而成。

```
ghci> :type splitAt
splitAt :: Int -> [a] -> ([a], [a])
ghci> splitAt 3 "foobar"
("foo","bar")
```

takeWhile 和 dropWhile 函数带着谓词：takeWhile 从开头遍历一个列表，抽取使谓词返回 True 的元素组成一个新列表；dropWhile 则是把使谓词返回 True 的元素丢掉。（译注：这里的表述容易引起歧义，实际上两个

函数都是走到第一个使谓词返回 False 的元素处就停止操作了，即使这个元素后面还有使谓词返回 True 的元素，两个函数也不再 take 或 drop 了）

```
ghci> :type takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
ghci> takeWhile odd [1,3,5,6,8,9,11]
[1,3,5]
ghci> :type dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
ghci> dropWhile even [2,4,6,7,9,10,12]
[7,9,10,12]
```

正如 splitAt 函数利用 take 和 drop 的结果组成一个二元组一样，break（已经在“热身：方便地分离多行文本”一节中看到过）和 span 函数利用 takeWhile 和 dropWhile 的结果组成二元组。

每个函数带着一个谓词，break 提取列表中使谓词失败的元素组成二元组的首项，而 span 提取列表中使谓词成功的元素组成二元组的首项。

```
ghci> :type span
span :: (a -> Bool) -> [a] -> ([a], [a])
ghci> span even [2,4,6,7,9,10,11]
([2,4,6],[7,9,10,11])
ghci> :type break
break :: (a -> Bool) -> [a] -> ([a], [a])
ghci> break even [1,3,5,6,8,9,10]
([1,3,5],[6,8,9,10])
```

4.5.6 搜索列表

正如我们已经看到的，elem 函数标示一个值是否出现在一个列表中。它有一个伴生的函数，notElem。

```
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
ghci> 2 `elem` [5,3,2,1,1]
True
ghci> 2 `notElem` [5,3,2,1,1]
False
```

对于更普遍的搜索操作，filter 函数带着一个谓词，返回列表中使谓词成功的每一个元素。

```
ghci> :type filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> filter odd [2,4,1,3,6,8,5,7]
[1,3,5,7]
```

在 Data.List 模块中，有三个谓词方法，isPrefixOf、isInfixOf 和 isSuffixOf，能让我们测试一下子列表在一个更大的列表中出现的位置。最容易的方式是把它们作为中缀使用。

isPrefixOf 函数告诉我们左边的列表是否出现在右边的列表的开始处。

```
ghci> :module +Data.List
ghci> :type isPrefixOf
isPrefixOf :: (Eq a) => [a] -> [a] -> Bool
ghci> "foo" `isPrefixOf` "foobar"
True
ghci> [1,2] `isPrefixOf` []
False
```

isInfixOf 函数标示左边的列表是否是右边的列表的一个子列表。

```
ghci> :module +Data.List
ghci> [2,6] `isInfixOf` [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9]
True
ghci> "funk" `isInfixOf` "sonic youth"
False
```

isSuffixOf 函数的功能不再赘述。

```
ghci> :module +Data.List
ghci> ".c" `isSuffixOf` "crashme.c"
True
```

4.5.7 一次性处理多个列表

zip 函数把两个列表压缩成一个单一的由二元组组成的列表。结果列表和被处理的两个列表中较短的那个等长。（译注：言下之意是较长的那个列表中的多出来的元素会被丢弃）

```
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]
ghci> zip [12,72,93] "zippity"
[(12,'z'),(72,'i'),(93,'p')]
```

更有用的是 zipWith 函数，它带两个列表作为参数并为从每个列表中抽取一个元素而组成的二元组提供一个函数，最后生成与较短的那个列表等长的新列表。

```
ghci> :type zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
ghci> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

Haskell 的类型系统使编写带有可变数量的参数的函数成为一个有趣的挑战。（译注：不知道此句和后面的内容有什么联系）。所以如果想把三个列表压缩在一起，要调用 zip3 或者 zipWith3，可以类推到 zip7 和 zipWith7。

4.5.8 特殊的字符串处理函数

我们已经在“热身：方便地分离多行文本”一节中遇到过 lines 函数，它有个对应的函数，unlines。注意 unlines 总是在它处理的结果（译注：列表中的每个元素）的尾部放一个换行符。

```
ghci> lines "foo\nbar"
["foo","bar"]
ghci> unlines ["foo", "bar"]
"foo\nbar\n"
```

words 函数利用任何空白字符分割一个字符串，它对应的函数，unwords，用一个空格字符把一个字符串构成的列表连接起来。

```
ghci> words "the  \r  quick \t  brown\n\n\nfox"
["the","quick","brown","fox"]
ghci> unwords ["jumps", "over", "the", "lazy", "dog"]
"jumps over the lazy dog"
```

4.5.9 练习题

1. 自己写一些安全的列表函数，确保它们不会出错。下面给一些类型定义的提示。

```
-- file: ch04/ch04.exercises.hs
safeHead :: [a] -> Maybe a
safeTail :: [a] -> Maybe [a]
safeLast :: [a] -> Maybe a
safeInit :: [a] -> Maybe [a]
```

2. 写一个和 words 功能近似的函数 splitWith，要求带一个谓词和一个任意类型元素组成的列表，在使谓词返回 False 的元素处分割这个列表。

```
-- file: ch04/ch04.exercises.hs
splitWith :: (a -> Bool) -> [a] -> [[a]]
```

3. 利用在“一个简单的命令行框架”一节中创建的命令行框架，编写一个打印输入数据的每一行的第一个单词的程序。

4. 编写一个转置一个文件中的文本的程序。比如，它应该把“hello\nworld\n”转换成“hw\neo\nlr\nll\nod\n”。

4.6 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。

4.6.1 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，渐增地计算结果。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，最终到达了列表的末尾。因此，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。

4.6.2 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];
    }
}
```

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'
```

(continues on next page)

(continued from previous page)

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

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

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

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。

4.6.3 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

*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。

4.6.4 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：

```haskell
-- 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` 。

4.6.5 处理集合并得出结果

将一个集合（collection）缩减（reduce）为一个值也是集合的常见操作之一。

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
    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（accumulator）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] & oxff)) % 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, (.&.), (.|.))
```

(continues on next page)

(continued from previous page)

```
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 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。

4.6.6 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
```

[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）。

4.6.7 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。

4.6.8 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

```

(continues on next page)

(continued from previous page)

```
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：

```
-- 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 将处初始化值放在左边，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。

这里有一种可爱的凭直觉性的关于 foldr 如何工作的解释: 他用 zero 初始值替代了空列表，并且调用 step 函数替代列表的每个值构造器.

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

乍一看, 比起 foldl, foldr 似乎不那么有用：一个从右边折叠的函数有什么用呢?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：

```haskell
-- 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：

```haskell
-- 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：

```haskell
-- 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）：

```haskell
-- 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）拼接到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。

4.6.9 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，就可以会从中看出一些问题：

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

(continues on next page)

(continued from previous page)

```
          == 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，块保存这种表达式所需的计算量，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的容量，多亏这个限制，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 (space leak)*：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

```

(continues on next page)

(continued from previous page)

```
ghci> foldl' (+) 0 [1..1000000]
500000500000
```

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]
500000000500000000
```

就是这样。] [我 (github:sancao2) 的电脑上面行为还是”Exception: stack overflow”.

```
  ch04  uname -a
Linux centos 3.10.0-229.20.1.el7.x86_64 #1 SMP Tue Nov 3 19:10:07 UTC 2015 x86_64 x86_
↪64 x86_64 GNU/Linux
  ch04  ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> foldl (+) 0 [1..100000000]
*** Exception: stack overflow
```

]

练习

1. 运用 `fold`（运用合适的 fold 将会使你的代码更简单）重写并扩展位于 " 显式递归 " 章节的 `asInt` 函数.

```
-- file: ch04/ch04.exercises.hs
asInt_fold :: String -> Int
```

你函数的效果应该是这样的.

```
ghci> asInt_fold "101"
101
ghci> asInt_fold "-31337"
-31337
ghci> asInt_fold "1798"
1798
```

扩展你的函数以处理下面当调用错误时候出现的异常的情况.

```
ghci> asInt_fold ""
0
ghci> asInt_fold "-"
0
ghci> asInt_fold "-3"
-3
ghci> asInt_fold "2.7"
*** Exception: Char.digitToInt: not a digit '.'
ghci> asInt_fold "314159265358979323846"
564616105916946374
```

2. `asInt_fold` 函数使用 `error`, 因而调用者不能处理这些错误. 重写他来修复这个问题.

```
-- file: ch04/ch04.exercises.hs
type ErrorMessage = String
asInt_either :: String -> Either ErrorMessage Int
ghci> asInt_either "33"
Right 33
ghci> asInt_either "foo"
Left "non-digit 'o'"
```

3.Prelude 下面的函数 `concat` 将一个列表的列表连接成一个单独的列表. 他的函数签名如下.

```
--file: ch04/ch04.exercises.hs
concat :: [[a]] -> [a]
```

用 foldr 写出你自己山寨的 concat.

4. 写出你自己山寨的 `takeWhile` 函数，首先用显式递归的手法，然后改成 `foldr` 形式.

5. `Data.List` 模块定义了一个函数, `groupBy`. 其拥有如下签名.

```
-- file: ch04/ch04.exercises.hs
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

运用 ghci 加载 Data.List 模块以理解 groupBy 的行为，然后写出你自己山寨的 fold 实现.

6. 下面 Prelude 函数能用 fold 系列函数重写的函数有多少?

```
any

cycle

words

unlines
```

这些函数你能用 foldl 或者 foldr 重写，请问那种情况更合适?

4.6.10 延伸阅读

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 。

4.7 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。这个故事说明，如果你要在 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。

4.8 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缀空白。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)]

```

(continues on next page)

(continued from previous page)

```
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)]
```

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
```

4.8.1 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）。

```
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`)
```

]

4.9 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"
"oobar"

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

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 ）。

在这个例子里，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。

4.10 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)
```

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个模式：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
```

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
```

]

4.11 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。

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。这强迫代码的阅读者（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。

4.12 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。

4.12.1 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 []
```

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。

4.12.2 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 的调用被隐藏了在 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 用在没有关联的两个表达式上面：

```
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 。正确的用法应该是用一个 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。

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，到时会说明更多 seq 的用法和细节。

CHAPTER

FIVE

第 5 章：编写 JSON 库

5.1 JSON 简介

在这一章，我们将开发一个小而完整的 Haskell 库，这个库用于处理和序列化 JSON 数据。

JSON （JavaScript 对象符号）是一种小型、表示简单、便于存储和发送的语言。它通常用于从 web 服务向基于浏览器的 JavaScript 程序传送数据。JSON 的格式由 www.json.org 描述，而细节由 RFC 4627 补充。

JSON 支持四种基本类型值：字符串、数字、布尔值和一个特殊值，`null` 。

```
"a string"

12345

true

null
```

JSON 还提供了两种复合类型：数组是值的有序序列，而对象则是“名字/值”对的无序集合（unordered collection of name/value pairs）。其中对象的名字必须是字符串，而对象和数组的值则可以是任何 JSON 类型。

```
[-3.14, true, null, "a string"]

{"numbers": [1,2,3,4,5], "useful": false}
```

5.2 在 Haskell 中表示 JSON 数据

要在 Haskell 中处理 JSON 数据，可以用一个代数数据类型来表示 JSON 的各个数据类型：

```
-- file: ch05/SimpleJSON.hs
data JValue = JString String
```

(continues on next page)

(continued from previous page)

```
              | JNumber Double
              | JBool Bool
              | JNull
              | JObject [(String, JValue)]
              | JArray [JValue]
                deriving (Eq, Ord, Show)
```

[译注：这里的 JObject [(String, JValue)] 不能改为 JObject [(JString, JValue)]，因为值构造器里面声明的是类构造器，不能是值构造器。

另外，严格来说，JObject 并不是完全无序的，因为它的定义使用了列表来包围，在书本的后面会介绍 Map 类型，它可以创建一个无序的键-值对结构。]

对于每个 JSON 类型，代码都定义了一个单独的值构造器。部分构造器带有参数，比如说，如果你要创建一个 JSON 字符串，那么就要给 JString 值构造器传入一个 String 类型值作为参数。

将这些定义载入到 ghci 试试看：

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

*Main> JString "the quick brown fox"
JString "the quick brown fox"

*Main> JNumber 3.14
JNumber 3.14

*Main> JBool True
JBool True

*Main> JNull
JNull

*Main> JObject [("language", JString "Haskell"), ("complier", JString "GHC")]
JObject [("language",JString "Haskell"),("complier",JString "GHC")]

*Main> JArray [JString "Haskell", JString "Clojure", JString "Python"]
JArray [JString "Haskell",JString "Clojure",JString "Python"]
```

前面代码中的构造器将一个 Haskell 值转换为一个 JValue 。反过来，同样可以通过模式匹配，从 JValue 中取出 Haskell 值。

以下函数试图从一个 JString 值中取出一个 Haskell 字符串：如果 JValue 真的包含一个字符串，那么程序返回一个用 Just 构造器包裹的字符串；否则，它返回一个 Nothing 。

```
-- file: ch05/SimpleJSON.hs
getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _           = Nothing
```

保存修改过的源码文件，然后使用 :reload 命令重新载入 SimpleJSON.hs 文件（:reload 会自动记忆最近一次载入的文件）：

```
*Main> :reload
[1 of 1] Compiling Main             ( SimpleJSON.hs, interpreted )
Ok, modules loaded: Main.

*Main> getString (JString "hello")
Just "hello"

*Main> getString (JNumber 3)
Nothing
```

再加上一些其他函数，初步完成一些基本功能：

```
-- file: ch05/SimpleJSON.hs
getInt (JNumber n) = Just (truncate n)
getInt _           = Nothing

getDouble (JNumber n) = Just n
getDouble _           = Nothing

getBool (JBool b) = Just b
getBool _         = Nothing

getObject (JObject o) = Just o
getObject _           = Nothing

getArray (JArray a) = Just a
getArray _          = Nothing

isNull v            = v == JNull
```

truncate 函数返回浮点数或者有理数的整数部分：

```
Prelude> truncate 5.8
5

Prelude> :module +Data.Ratio
```

(continues on next page)

(continued from previous page)

```
Prelude Data.Ratio> truncate (22 % 7)
3
```

5.3 Haskell 模块

一个 Haskell 文件可以包含一个模块定义，模块可以决定模块中的哪些名字可以被外部访问。

模块的定义必须放在其它定义之前：

```
-- file: ch05/SimpleJSON.hs
module SimpleJSON
    (
        JValue(..)
    ,   getString
    ,   getInt
    ,   getDouble
    ,   getBool
    ,   getObject
    ,   getArray
    ,   isNull
    ) where
```

单词 `module` 是保留字，跟在它之后的是模块的名字：模块名字必须以大写字母开头，并且它必须和包含这个模块的文件的基础名（不包含后缀的文件名）一致。比如上面定义的模块就以 `SimpleJSON` 命名，因为包含它的文件名为 `SimpleJSON.hs` 。

在模块名之后，用括号包围的是导出列表（list of exports）。`where` 关键字之后的内容为模块的体。

导出列表决定模块中的哪些名字对于外部模块是可见的，使得私有代码可以隐藏在模块的内部。跟在 `JValue` 之后的 `(..)` 符号表示导出 `JValue` 类型以及它的所有值构造器。

事实上，模块甚至可以只导出类型的名字（类构造器），而不导出这个类型的值构造器。这种能力非常重要：它允许模块对用户隐藏类型的细节，将一个类型变得抽象。如果用户看不见类型的值构造器，他就没办法对类型的值进行模式匹配，也不能使用值构造器显式创建这种类型的值 [译注：只能通过相应的 API 来创建这种类型的值]。本章稍后会说明，在什么情况下，我们需要将一个类型变得抽象。

如果省略掉模块定义中的导出部分，那么所有名字都会被导出：

```
module ExportEverything where
```

如果不想导出模块中的任何名字（通常不会这么用），那么可以将导出列表留空，仅保留一对括号：

```
module ExportNothing () where
```

5.4 编译 Haskell 代码

除了 ghci 之外，GHC 还包括一个生成本地码（native code）的编译器：ghc 。如果你熟悉 gcc 或者 cl （微软 Visual Studio 使用的 C++ 编译器组件）之类的编译器，那么你对 ghc 应该不会感到陌生。

编译一个 Haskell 源码文件可以通过 `ghc` 命令来完成：

```
$ ghc -c SimpleJSON.hs

$ ls
SimpleJSON.hi  SimpleJSON.hs  SimpleJSON.o
```

`-c` 表示让 ghc 只生成目标代码。如果省略 `-c` 选项，那么 ghc 就会试图生成一个完整的可执行文件，这会失败，因为目前的 `SimpleJSON.hs` 还没有定义 `main` 函数，而 GHC 在执行一个独立程序时会调用这个 `main` 函数。

在编译完成之后，会生成两个新文件。其中 `SimpleJSON.hi` 是接口文件（interface file），ghc 以机器可读的格式，将模块中导出名字的信息保存在这个文件。而 `SimpleJSON.o` 则是目标文件（object file），它包含了已生成的机器码。

5.5 载入模块和生成可执行文件

既然已经成功编译了 `SimpleJSON` 库，是时候写个小程序来执行它了。打开编辑器，将以下内容保存为 `Main.hs`：

```
-- file: ch05/Main.hs

module Main (main) where

import SimpleJSON

main = print (JObject [("foo", JNumber 1), ("bar", JBool False)])
```

[译注：原文说，可以不导出 `main` 函数，但是实际中测试这种做法并不能通过编译。]

放在模块定义之后的 `import` 表示载入所有 `SimpleJSON` 模块导出的名字，使得它们在 `Main` 模块中可用。

所有 `import` 指令（directive）都必须出现在模块的开头，并且位于其他模块代码之前。不可以随意摆放 `import` 。

Main.hs 的名字和 main 函数的命名是有特别含义的，要创建一个可执行文件，ghc 需要一个命名为 Main 的模块，并且这个模块里面还要有一个 main 函数，而 main 函数在程序执行时会被调用。

```
ghc -o simple Main.hs
```

这次编译没有使用 -c 选项，因此 ghc 会尝试生成一个可执行程序，这个过程被称为链接（linking）。ghc 可以在一条命令中同时完成编译和链接的任务。

-o 选项用于指定可执行程序的名字。在 Windows 平台下，它会生成一个 .exe 后缀的文件，而 UNIX 平台的文件则没有后缀。

ghc 会自动找到所需的文件，进行编译和链接，然后产生可执行文件，我们唯一要做的就是提供 Main.hs 文件。

[译注：在原文中说到，编译时必须手动列出所有相关文件，但是在新版 GHC 中，编译时提供 Main.hs 就可以了，编译器会自动找到、编译和链接相关代码。因此，本段内容做了相应的修改。]

一旦编译完成，就可以运行编译所得的可执行文件了：

```
$ ./simple
JObject [("foo",JNumber 1.0),("bar",JBool False)]
```

5.6 打印 JSON 数据

SimpleJSON 模块已经有了 JSON 类型的表示了，那么下一步要做的就是将 Haskell 值翻译（render）成 JSON 数据。

有好几种方法可以将 Haskell 值翻译成 JSON 数据，最直接的一种是编写翻译函数，以 JSON 格式来打印 Haskell 值。稍后会介绍完成这个任务的其他更有趣方法。

```
-- file: ch05/PutJSON.hs
module PutJSON where

import Data.List (intercalate)
import SimpleJSON

renderJValue :: JValue -> String

renderJValue (JString s)   = show s
renderJValue (JNumber n)   = show n
renderJValue (JBool True)  = "true"
renderJValue (JBool False) = "false"
renderJValue JNull         = "null"
```

(continues on next page)

(continued from previous page)

```
renderJValue (JObject o) = "{" ++ pairs o ++ "}"
  where pairs [] = ""
    pairs ps = intercalate ", " (map renderPair ps)
    renderPair (k,v)   = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++ "]"
  where values [] = ""
    values vs = intercalate ", " (map renderJValue vs)
```

分割纯代码和带有 IO 的代码是一种良好的 Haskell 风格。这里我们用 `putJValue` 来进行打印操作，这样就不会影响 `renderJValue` 的纯洁性：

```
putJValue :: JValue -> IO ()
putJValue v = putStrLn (renderJValue v)
```

现在打印 JSON 值变得容易得多了：

```
Prelude SimpleJSON> :load PutJSON
[2 of 2] Compiling PutJSON          ( PutJSON.hs, interpreted )
Ok, modules loaded: PutJSON, SimpleJSON.

*PutJSON> putJValue (JString "a")
"a"

*PutJSON> putJValue (JBool True)
true
```

除了风格上的考虑之外，将翻译代码和实际打印代码分开，也有助于提升灵活性。比如说，如果想在数据写出之前进行压缩，那么只需要修改 `putJValue` 就可以了，不必改动整个 `renderJValue` 函数。

将纯代码和不纯代码分离的理念非常强大，并且在 Haskell 代码中无处不在。现有的一些 Haskell 压缩模块，它们都拥有简单的接口：压缩函数接受一个未压缩的字符串，并返回一个压缩后的字符串。通过组合使用不同的函数，可以在打印 JSON 值之前，对数据进行各种不同的处理。

5.7 类型推导是一把双刃剑

Haskell 编译器的类型推导能力非常强大也非常有价值。在刚开始的时候，我们通常会倾向于尽可能地省略所有类型签名，让类型推导去决定所有函数的类型定义。

但是，这种做法是有缺陷的，它通常是 Haskell 新手引发类型错误的主要来源。

如果我们省略显式的类型信息时，那么编译器就必须猜测我们的意图：它会推导出合乎逻辑且相容的（consistent）类型，但是，这些类型可能并不是我们想要的。一旦程序员和编译器之间的想法产生了分歧，那么寻

找 bug 的工作就会变得更困难。

作为例子，假设有一个函数，它预计会返回 String 类型的值，但是没有显式地为它编写类型签名：

```
-- file: ch05/Trouble.hs

import Data.Char (toUpper)

upcaseFirst (c:cs) = toUpper c  -- 这里忘记了 ":cs"
```

这个函数试图将输入单词的第一个字母设置为大写，但是它在设置之后，忘记了重新拼接字符串的后续部分 cs 。在我们的预想中，这个函数的类型应该是 String -> String ，但编译器推导出的类型却是 String -> Char 。

现在，有另一个函数调用这个 upcaseFirst 函数：

```
-- file: ch05/Trouble.hs

camelCase :: String -> String
camelCase xs = concat (map upcaseFirst (words xs))
```

这段代码在载入 ghci 时会发生错误：

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

Trouble.hs:8:28:
    Couldn't match expected type `[Char]' with actual type `Char'
    Expected type: [Char] -> [Char]
        Actual type: [Char] -> Char
    In the first argument of `map', namely `upcaseFirst'
    In the first argument of `concat', namely            `(map upcaseFirst (words xs))
↪'
Failed, modules loaded: none.
```

请注意，如果不是 upcaseFirst 被其他函数所调用的话，它的错误可能并不会被发现！相反，如果我们之前为 upcaseFirst 编写了类型签名的话，那么 upcaseFirst 的类型错误就会立即被捕捉到，并且可以即刻定位出错误发生的位置。

为函数编写类型签名，既可以移除我们实际想要的类型和编译器推导出的类型之间的分歧，也可以作为函数的一种文档，帮助阅读和理解函数的行为。

这并不是说要巨细无遗地为所有函数都编写类型签名。不过，为所有顶层（top-level）函数添加类型签名通常是一种不错的做法。在刚开始的时候最好尽可能地为函数添加类型签名，然后随着对类型系统了解的加深，逐步放松要求。

5.8 更通用的转换方式

在前面构造 SimpleJSON 库时，我们的目标主要是按照 JSON 的格式，将 Haskell 数据转换为 JSON 值。而这些转换所得值的输出可能并不是那么适合人去阅读。有一些被称为美观打印器（pretty printer）的库，它们的输出既适合机器读入，也适合人类阅读。我们这就来编写一个美观打印器，学习库设计和函数式编程的相关技术。

这个美观打印器库命名为 Prettify ，它被包含在 `Prettify.hs` 文件里。为了让 Prettify 适用于实际需求，我们先编写一个新的 JSON 转换器，它使用 Prettify 提供的 API 。等完成这个 JSON 转换器之后，再转过头来补充 Prettify 模块的细节。

和前面的 SimpleJSON 模块不同，Prettify 模块将数据转换为一种称为 `Doc` 类型的抽象数据，而不是字符串：抽象类型允许我们随意选择不同的实现，最大化灵活性和效率，而且在更改实现时，不会影响到用户。

新的 JSON 转换模块被命名为 `PrettyJSON.hs` ，转换的工作依然由 `renderJValue` 函数进行，它的定义和之前一样简单直观：

```
-- file: ch05/PrettyJSON.hs
renderJValue :: JValue -> Doc
renderJValue (JBool True)  = text "true"
renderJValue (JBool False) = text "false"
renderJValue JNull         = text "null"
renderJValue (JNumber num) = double num
renderJValue (JString str) = string str
```

其中 `text` 、`double` 和 `string` 都由 Prettify 模块提供。

5.9 Haskell 开发诀窍

在刚开始进行 Haskell 开发的时候，通常需要面对大量崭新、不熟悉的概念，要一次性完成程序的编写，并顺利通过编译器检查，难度非常的高。

在每次完成一个功能点时，花几分钟停下来，对程序进行编译，是非常有益的：因为 Haskell 是强类型语言，如果程序能成功通过编译，那么说明程序和我们预想中的目标相去不远。

编写函数和类型的占位符（placeholder）版本，对于快速原型开发非常有效。举个例子，前文断言，`string` 、`text` 和 `double` 函数都由 `Prettify` 模块提供，如果 `Prettify` 模块里不定义这些函数，或者不定义 `Doc` 类型，那么对程序的编译就会失败，我们的“早编译，常编译”战术就没有办法施展。通过编写占位符代码，可以避免这些问题：

```
-- file: ch05/PrettyStub.hs
import SimpleJSON

```

(continues on next page)

(continued from previous page)

```
data Doc = ToBeDefined
         deriving (Show)

 string :: String -> Doc
 string str = undefined

 text :: String -> Doc
 text str = undefined

 double :: Double -> Doc
 double num = undefined
```

特殊值 undefined 的类型为 a，因此它可以让代码顺利通过类型检查。因为它只是一个占位符，没有什么实际作用，所以对它进行求值只会产生错误：

```
*Main> :type undefined
undefined :: a

*Main> undefined
*** Exception: Prelude.undefined

*Main> :load PrettyStub.hs
[2 of 2] Compiling Main             ( PrettyStub.hs, interpreted )
Ok, modules loaded: Main, SimpleJSON.

*Main> :type double
double :: Double -> Doc

*Main> double 3.14
*** Exception: Prelude.undefined
```

尽管程序里还没有任何实际可执行的代码，但是编译器的类型检查器可以保证程序中类型的正确性，这为接下来的进一步开发奠定了良好基础。

[译注：原文中 PrettyStub.hs 和 Prettify.hs 混合使用，给读者阅读带来了很大麻烦。为了避免混淆，下文统一在 Prettify.hs 中书写代码，并列出编译通过所需要的占位符代码。随着文章进行，读者只要不断将占位符版本替换为可用版本即可。]

5.10 美观打印字符串

当需要美观地打印字符串时，我们需要遵守 JSON 的转义规则。字符串，顾名思义，仅仅是一串被包含在引号中的字符而已。

```haskell
-- file: ch05/Prettify.hs
string :: String -> Doc
string = enclose '"' '"' . hcat . map oneChar

enclose :: Char -> Char -> Doc -> Doc
enclose left right x = undefined

hcat :: [Doc] -> Doc
hcat xs = undefined

oneChar :: Char -> Doc
oneChar c = undefined
```

enclose 函数把一个 Doc 值用起始字符和终止字符包起来。(<>) 函数将两个 Doc 值拼接起来。也就是说，它是 Doc 中的 ++ 函数。

```haskell
-- file: ch05/Prettify.hs
enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right

(<>) :: Doc -> Doc -> Doc
a <> b = undefined

char :: Char -> Doc
char c = undefined
```

hcat 函数将多个 Doc 值拼接成一个，类似列表中的 concat 函数。

string 函数将 oneChar 函数应用于字符串的每一个字符，然后把拼接起来的结果放入引号中。oneChar 函数将一个单独的字符进行转义（escape）或转换（render）。

```haskell
-- file: ch05/Prettify.hs
oneChar :: Char -> Doc
oneChar c = case lookup c simpleEscapes of
              Just r -> text r
              Nothing | mustEscape c -> hexEscape c
                      | otherwise    -> char c
    where mustEscape c = c < ' ' || c == '\x7f' || c > '\xff'

simpleEscapes :: [(Char, String)]
simpleEscapes = zipWith ch "\b\n\f\r\t\\\"/" "bnfrt\\\"/"
    where ch a b = (a, ['\\',b])

hexEscape :: Char -> Doc
hexEscape c = undefined
```

simpleEscapes 是一个序对组成的列表。我们把由序对组成的列表称为关联列表（*association list*），或简称为 *alist*。我们的 alist 将字符和其对应的转义形式关联起来。

```
ghci> :l Prettify.hs
ghci> take 4 simpleEscapes
[('\b',"\\b"),('\n',"\\n"),('\f',"\\f"),('\r',"\\r")]
```

case 表达式试图确定一个字符是否存在于 alist 当中。如果存在，我们就返回它对应的转义形式，否则我们就要用更复杂的方法来转义它。当两种转义都不需要时我们返回字符本身。保守地说，我们返回的非转义字符只包含可打印的 ASCII 字符。

上文提到的复杂的转义是指将一个 Unicode 字符转为一个 “\u” 加上四个表示它编码 16 进制数字。

[译注：smallHex 函数为 hexEscape 函数的一部分，只处理较为简单的一种情况。]

```
-- file: ch05/Prettify.hs
import Numeric (showHex)

smallHex :: Int -> Doc
smallHex x  = text "\\u"
           <> text (replicate (4 - length h) '0')
           <> text h
    where h = showHex x ""
```

showHex 函数来自于 Numeric 库（需要在 Prettify.hs 开头载入），它返回一个数字的 16 进制表示。

```
ghci> showHex 114111 ""
"1bdbf"
```

replicate 函数由 Prelude 提供，它创建一个长度确定的重复列表。

```
ghci> replicate 5 "foo"
["foo","foo","foo","foo","foo"]
```

有一点需要注意：smallHex 提供的 4 位数字编码仅能够表示 0xffff 范围之内的 Unicode 字符。而合法的 Unicode 字符范围可达 0x10ffff 。为了使用 JSON 字符串表示这部分字符，我们需要遵循一些复杂的规则将它们一分为二。这使得我们有机会对 Haskell 数字进行一些位操作 (bit-level manipulation)。

```
-- file: ch05/Prettify.hs
import Data.Bits (shiftR, (.&.))

astral :: Int -> Doc
astral n = smallHex (a + 0xd800) <> smallHex (b + 0xdc00)
    where a = (n `shiftR` 10) .&. 0x3ff
          b = n .&. 0x3ff
```

shiftR 函数来自 Data.Bits 模块，它把一个数字右移一位。同样来自于 Data.Bits 模块的 (.&.) 函数将两个数字进行按位与操作。

```
ghci> 0x10000 `shiftR` 4    :: Int
4096
ghci> 7 .&. 2    :: Int
2
```

有了 smallHex 和 astral ，我们可以如下定义 hexEscape：

```
-- file: ch05/Prettify.hs
import Data.Char (ord)

hexEscape :: Char -> Doc
hexEscape c | d < 0x10000 = smallHex d
            | otherwise   = astral (d - 0x10000)
    where d = ord c
```

5.11 数组和对象

跟字符串比起来，美观打印数组和对象就简单多了。我们已经知道它们两个看起来很像：以起始字符开头，中间是用逗号隔开的一系列值，以终止字符结束。我们写个函数来体现它们的共同特点：

```
-- file: ch05/Prettify.hs
series :: Char -> Char -> (a -> Doc) -> [a] -> Doc
series open close f = enclose open close
                      . fsep . punctuate (char ',') . map f
```

首先我们来解释这个函数的类型。它的参数是一个起始字符和一个终止字符，然后是一个知道怎样打印未知类型 a 的函数，接着是一个包含 a 类型数据的列表，最后返回一个 Doc 类型的值。

尽管函数的类型签名有 4 个参数，我们在函数定义中只列出了 3 个。这跟我们把 myLength xs = length xs 简化成 myLength = length 是一个道理。

我们已经有了把 Doc 包在起始字符和终止字符之间的 enclose 函数。fsep 会在 Prettify 模块中定义。它将多个 Doc 值拼接成一个，并且在需要的时候换行。

```
-- file: ch05/Prettify.hs
fsep :: [Doc] -> Doc
fsep xs = undefined
```

punctuate 函数也会在 Prettify 中定义。

```
-- file: ch05/Prettify.hs
punctuate :: Doc -> [Doc] -> [Doc]
punctuate p []     = []
punctuate p [d]    = [d]
punctuate p (d:ds) = (d <> p) : punctuate p ds
```

有了 series，美观打印数组就非常直观了。我们在 renderJValue 的定义的最后加上下面一行。

```
-- file: ch05/PrettyJSON.hs
renderJValue (JArray ary) = series '[' ']' renderJValue ary
```

美观打印对象稍微麻烦一点：对于每个元素，我们还要额外处理名字和值。

```
-- file: ch05/PrettyJSON.hs
renderJValue (JObject obj) = series '{' '}' field obj
    where field (name,val) = string name
                          <> text ": "
                          <> renderJValue val
```

5.12 书写模块头

PrettyJSON.hs 文件写得差不多了，我们现在回到文件顶部书写模块声明。

```
-- file: ch05/PrettyJSON.hs
module PrettyJSON
    (
      renderJValue
    ) where

import SimpleJSON (JValue(..))
import Prettify (Doc, (<>), char, double, fsep, hcat, punctuate, text, compact,␣
↪pretty)
```

[译注：compact 和 pretty 函数会在稍后介绍。]

我们只从这个模块导出了一个函数，renderJValue，也就是我们的 JSON 转换函数。其它的函数只是为了支持 renderJValue，因此没必要对其它模块可见。

关于载入部分，Numeric 和 Data.Bits 模块是 GHC 内置的。我们已经写好了 SimpleJSON 模块，Prettify 模块的框架也搭好了。可以看出载入标准模块和我们自己写的模块没什么区别。[译注：原文在 PrettyJSON.hs 头部载入了 Numeric 和 Data.Bits 模块。但事实上并无必要，因此在译文中删除。此处作者的说明部分未作改动。]

在每个 import 命令中，我们都显式地列出了名字，用以引入我们模块的命名空间。这并非强制：如果省略这些名字，我们就可以使用一个模块导出的所有名字。然而，通常来讲显式地载入更好。

- 一个显式列表清楚地表明了我们从哪里载入了哪个名字。如果读者碰到了不熟悉的函数，这便于他们查看文档。
- 有时候库的维护者会删除或者重命名函数。一个函数很可能在我们写完模块很久之后才从第三方库中消失并导致编译错误。显式列表提醒我们消失的名字是从哪儿载入的，有助于我们更快找到问题。
- 另外一种情况是库的维护者在模块中加入的函数与我们代码中现有的函数名字一样。如果不用显式列表，这个函数就会在我们的模块中出现两次。当我们用这个函数的时候，GHC 就会报告歧义错误。

通常情况下使用显式列表更好，但这并不是硬性规定。有的时候，我们需要一个模块中的很多名字，一一列举会非常麻烦。有的时候，有些模块已经被广泛使用，有经验的 Hashell 程序员会知道哪个名字来自那些模块。

5.13 完成美观打印库

在 Prettify 模块中，我们用代数数据类型来表示 Doc 类型。

```
-- file: ch05/Prettify.hs
data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
           deriving (Show,Eq)
```

可以看出 Doc 类型其实是一棵树。Concat 和 Union 构造器以两个 Doc 值构造一个内部节点，Empty 和其它简单的构造器构造叶子。

在模块头中，我们导出了这个类型的名字，但是不包含任何它的构造器：这样可以保证使用这个类型的模块无法创建 Doc 值和对其进行模式匹配。

如果想创建 Doc，Prettify 模块的用户可以调用我们提供的函数。下面是一些简单的构造函数。

```
-- file: ch05/Prettify.hs
empty :: Doc
empty = Empty

char :: Char -> Doc
char c = Char c

text :: String -> Doc
```

(continues on next page)

(continued from previous page)

```
text "" = Empty
text s  = Text s

double :: Double -> Doc
double d = text (show d)
```

Line 构造器表示一个换行。line 函数创建一个硬换行，它总是出现在美观打印器的输出中。有时候我们想要一个软换行，只有在行太宽，一个窗口或一页放不下的时候才用。稍后我们就会介绍这个 softline 函数。

```
-- file: ch05/Prettify.hs
line :: Doc
line = Line
```

下面是 (<>) 函数的实现。

```
-- file: ch05/Prettify.hs
(<>) :: Doc -> Doc -> Doc
Empty <> y = y
x <> Empty = x
x <> y = x `Concat` y
```

我们使用 Empty 进行模式匹配。将一个 Empty 拼接在一个 Doc 值的左侧或右侧都不会有效果。这样可以帮助我们的树减少一些无意义信息。

```
ghci> text "foo" <> text "bar"
Concat (Text "foo") (Text "bar")
ghci> text "foo" <> empty
Text "foo"
ghci> empty <> text "bar"
Text "bar"
```

Note: 来思考下数学从数学的角度来看，Empty 是拼接操作 (concatenation) 的单位元，因为任何 Doc 值和 Empty 拼接后都不变。同理，0 是加法的单位元，1 是乘法的单位元. 从数学的角度来看问题有非常实用的价值，我们在本书后面还会多次看到此类情况。

我们的 hcat 和 fsep 函数将 Doc 列表拼接成一个 Doc 值。在之前的一道题目里（fix link），我们提到了可以用 foldr 来定义列表拼接。[译注：这个例子只是为了回顾，本章代码并没有用到。]

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

因为 (<>) 类比于 (++)，empty 类比于 []，我们可以用同样的方法来定义 hcat 和 fsep 函数。

```haskell
-- file: ch05/Prettify.hs
hcat :: [Doc] -> Doc
hcat = fold (<>)

fold :: (Doc -> Doc -> Doc) -> [Doc] -> Doc
fold f = foldr f empty
```

fsep 的定义依赖于其它几个函数。

```haskell
-- file: ch05/Prettify.hs
fsep :: [Doc] -> Doc
fsep = fold (</>)

(</>) :: Doc -> Doc -> Doc
x </> y = x <> softline <> y

softline :: Doc
softline = group line

group :: Doc -> Doc
group x = undefined
```

稍微来解释一下。如果当前行变得太长，softline 函数就插入一个新行，否则就插入一个空格。Doc 并没有包含“怎样才算太长”的信息，这该怎么实现呢？答案是每次碰到这种情况，我们使用 Union 构造器来用两种不同的方式保存文档。

```haskell
-- file: ch05/Prettify.hs
group :: Doc -> Doc
group x = flatten x `Union` x

flatten :: Doc -> Doc
flatten = undefined
```

flatten 函数将 Line 替换为一个空格，把两行变成一行。

```haskell
-- file: ch05/Prettify.hs
flatten :: Doc -> Doc
flatten (x `Concat` y) = flatten x `Concat` flatten y
flatten Line           = Char ' '
flatten (x `Union` _)  = flatten x
flatten other          = other
```

我们只在 Union 左侧的元素上调用 flatten：Union 左侧元素的长度总是大于等于右侧元素的长度。下面的转换函数会用到这一性质。

5.13.1 紧凑转换

我们经常希望一段数据占用的字符数越少越好。例如，如果我们想通过网络传输 JSON 数据，就没必要把它弄得很漂亮：另外一端的软件并不关心它漂不漂亮，而使布局变漂亮的空格会增加额外开销。

在这种情况下，我们提供一个最基本的紧凑转换函数。

```
-- file: ch05/Prettify.hs
compact :: Doc -> String
compact x = transform [x]
    where transform [] = ""
          transform (d:ds) =
                case d of
                    Empty        -> transform ds
                    Char c       -> c : transform ds
                    Text s       -> s ++ transform ds
                    Line         -> '\n' : transform ds
                    a `Concat` b -> transform (a:b:ds)
                    _ `Union` b  -> transform (b:ds)
```

compact 函数把它的参数放进一个列表里，然后再对它应用 transform 辅助函数。transform 函数把参数当做栈来处理，列表的第一个元素即为栈顶。

transform 函数的 (d:ds) 模式将栈分为头 d 和剩余部分 ds。在 case 表达式里，前几个分支在 ds 上递归，每次处理一个栈顶的元素。最后两个分支在 ds 前面加了东西：Concat 分支把两个元素都加到栈里，Union 分支忽略左侧元素（我们对它调用了 flatten），只把右侧元素加进栈里。

现在我们终于可以在 **ghci** 里试试 compact 函数了。[译注：这里要对 PrettyJSON.hs 里 import Prettify 部分作一下修改才能使 PrettyJSON.hs 编译。包括去掉还未实现的 pretty 函数，增加缺少的 string, series 函数等。一个可以编译的版本如下。]

[我 (sancao2) 还是不能编译成功, 报错:

```
Prelude> :l PrettyJSON.hs

Prettify.hs:1:1:
    File name does not match module name:
    Saw: `Main'
    Expected: `Prettify'
Failed, modules loaded: none.
```

在开头加上 module Prettify where. 就能编译通过了.

```
Prelude> :l PrettyJSON
[1 of 3] Compiling SimpleJSON       ( SimpleJSON.hs, interpreted )
```

(continues on next page)

(continued from previous page)

```
[2 of 3] Compiling Prettify         ( Prettify.hs, interpreted )
[3 of 3] Compiling PrettyJSON       ( PrettyJSON.hs, interpreted )
Ok, modules loaded: PrettyJSON, SimpleJSON, Prettify.
```

]

```
-- file: ch05/PrettyJSON.hs
import Prettify (Doc, (<>), string, series, char, double, fsep, hcat, punctuate, text,
↪ compact)
```

```
ghci> let value = renderJValue (JObject [("f", JNumber 1), ("q", JBool True)])
ghci> :type value
value :: Doc
ghci> putStrLn (compact value)
{"f": 1.0,
"q": true
}
```

为了更好地理解代码，我们来分析一个更简单的例子。

```
ghci> char 'f' <> text "oo"
Concat (Char 'f') (Text "oo")
ghci> compact (char 'f' <> text "oo")
"foo"
```

当我们调用 compact 时，它把参数转成一个列表并应用 transform。

- transform 函数的参数是一个单元素列表，匹配 (d:ds) 模式。因此 d 是 Concat (Char 'f') (Text "oo")，ds 是个空列表，[]。

  因为 d 的构造器是 Concat，case 表达式匹配到了 Concat 分支。我们把 Char 'f' 和 Text "oo" 放进栈里，并递归调用 transform。

- – 这次 transform 的参数是一个二元素列表，匹配 (d:ds) 模式。变量 d 被绑定到 Char 'f'，ds 被绑定到 [Text "oo"]。case 表达式匹配到 Char 分支。因此我们用 (:) 构造一个列表，它的头是 'f'，剩余部分是对 transform 进行递归调用的结果。

  – * 这次递归调用的参数是一个单元素列表，变量 d 被绑定到 Text "oo"，ds 被绑定到 []。case 表达式匹配到 Text 分支。我们用 (++) 拼接 "oo" 和下次递归调用的结果。

  * · 最后一次调用，transform 的参数是一个空列表，因此返回一个空字符串。

  * 结果是 "oo" ++ ""。

  – 结果是 'f' : "oo" ++ ""。

5.13.2 真正的美观打印

我们的 compact 方便了机器之间的交流，人阅读起来却非常困难。我们写一个 pretty 函数来产生可读性较强的输出。跟 compact 相比, pretty 多了一个参数：每行的最大宽度 (有几列)。(假设我们使用等宽字体。)

```
-- file: ch05/Prettify.hs
pretty :: Int -> Doc -> String
pretty = undefined
```

更准确地说，这个 Int 参数控制了 pretty 遇到 softline 时的行为。只有碰到 softline 时，pretty 才能选择继续当前行还是新开一行。别的地方，我们必须严格遵守已有的打印规则。

下面是这个函数的核心部分。

```
-- file: ch05/Prettify.hs
pretty :: Int -> Doc -> String
pretty width x = best 0 [x]
    where best col (d:ds) =
        case d of
            Empty        -> best col ds
            Char c       -> c :  best (col + 1) ds
            Text s       -> s ++ best (col + length s) ds
            Line         -> '\n' : best 0 ds
            a `Concat` b -> best col (a:b:ds)
            a `Union` b  -> nicest col (best col (a:ds))
                                       (best col (b:ds))
      best _ _ = ""

      nicest col a b | (width - least) `fits` a = a
                     | otherwise                = b
                    where least = min width col

fits :: Int -> String -> Bool
fits = undefined
```

辅助函数 best 接受两个参数：当前行已经走过的列数和剩余需要处理的 Doc 列表。一般情况下，best 会简单地消耗输入更新 col。即使 Concat 这种情况也显而易见：我们把拼接好的两个元素放进栈里，保持 col 不变。

有趣的是涉及到 Union 构造器的情况。回想一下，我们将 flatten 应用到了左侧元素，右侧不变。并且，flatten 把换行替换成了空格。因此，我们的任务是看看两种布局中，哪一种（如果有的话）能满足我们的 width 限制。

我们还需要一个小的辅助函数来确定某一行已经被转换的 Doc 值是否能放进给定的宽度中。

```
-- file: ch05/Prettify.hs
fits :: Int -> String -> Bool
w `fits` _ | w < 0 = False
w `fits` ""        = True
w `fits` ('\n':_)  = True
w `fits` (c:cs)    = (w - 1) `fits` cs
```

5.13.3 理解美观打印器

为了理解这段代码是如何工作的，我们首先来考虑一个简单的 Doc 值。[译注：PrettyJSON.hs 并未载入 empty 和 </>。需要读者自行载入。]

```
ghci> empty </> char 'a'
Concat (Union (Char ' ') Line) (Char 'a')
```

我们会将 pretty 2 应用到这个值上。第一次应用 best 时，col 的值是 0。它匹配到了 Concat 分支，于是把 Union (Char ' ') Line 和 Char 'a' 放进栈里，继续递归。在递归调用时，它匹配到了 Union 分支。

这个时候，我们忽略 Haskell 通常的求值顺序。这使得在不影响结果的情况下，我们的解释最容易被理解。现在我们有两个子表达式：best 0 [Char ' ', Char 'a'] 和 best 0 [Line, Char 'a']。第一个被求值成 " a"，第二个被求值成 "\na"。我们把这些值替换进函数得到 nicest 0 " a" "\na"。

为了弄清 nicest 的结果是什么，我们再做点替换。width 和 col 的值分别是 0 和 2，所以 least 是 0，width - least 是 2。我们在 **ghci** 里试试 2 `fits` " a" 的结果是什么。

```
ghci> 2 `fits` " a"
True
```

由于求值结果为 True，nicest 的结果是 " a"。

如果我们将 pretty 函数应用到之前的 JSON 上，我们可以看到随着我们给它的宽度不同，它产生了不同的结果。

```
ghci> putStrLn (pretty 10 value)
{"f": 1.0,
"q": true
}
ghci> putStrLn (pretty 20 value)
{"f": 1.0, "q": true
}
ghci> putStrLn (pretty 30 value)
{"f": 1.0, "q": true }
```

5.13.4 练习

我们现有的美观打印器已经可以满足一定的空间限制要求，我们还可以对它做更多改进。

1. 用下面的类型签名写一个函数 `fill`。

```
-- file: ch05/Prettify.hs
fill :: Int -> Doc -> Doc
```

   它应该给文档添加空格直到指定宽度。如果宽度已经超过指定值，则不加。

2. 我们的美观打印器并未考虑嵌套（nesting）这种情况。当左括号（无论是小括号，中括号，还是大括号）出现时，之后的行应该缩进，直到对应的右括号出现为止。

   实现这个功能，缩进量应该可控。

```
-- file: ch05/Prettify.hs
nest :: Int -> Doc -> Doc
```

5.14 创建包

Cabal 是 Haskell 社区用来构建，安装和发布软件的一套标准工具。Cabal 将软件组织为包（*package*）。一个包有且只能有一个库，但可以有多个可执行程序。

5.14.1 为包添加描述

Cabal 要求你给每个包添加描述。这些描述放在一个以 `.cabal` 结尾的文件当中。这个文件需要放在你项目的顶层目录里。它的格式很简单，下面我们就来介绍它。

每个 Cabal 包都需要有个名字。通常来说，包的名字和 `.cabal` 文件的名字相同。如果我们的包叫做 `mypretty`，那我们的文件就是 `mypretty.cabal`。通常，包含 `.cabal` 文件的目录名字和包名字相同，如 `mypretty`。

放在包描述开头的是一些全局属性，它们适用于包里所有的库和可执行程序。

```
Name:           mypretty
Version:        0.1

-- This is a comment.  It stretches to the end of the line.
```

包的名字必须独一无二。如果你创建安装的包和你系统里已经存在的某个包名字相同，GHC 会搞不清楚用哪个。

全局属性中的很多信息都是给人而不是 Cabal 自己来读的。

```
Synopsis:       My pretty printing library, with JSON support
Description:
    A simple pretty printing library that illustrates how to
    develop a Haskell library.
Author:         Real World Haskell
Maintainer:     somebody@realworldhaskell.org
```

如 Description 所示，一个字段可以有多行，只要缩进即可。

许可协议也被放在全局属性中。大部分 Haskell 包使用 BSD 协议，Cabal 称之为 BSD3。（当然，你可以随意选择合适的协议。）我们可以在 License-File 这个非强制字段中加入许可协议文件，这个文件包含了我们的包所使用的协议的全部协议条款。

Cabal 所支持的功能会不断变化，因此，指定我们期望兼容的 Cabal 版本是非常明智的。我们增加的功能可以被 Cabal 1.2 及以上的版本支持。

```
Cabal-Version: >= 1.2
```

我们使用 *library* 区域来描述包中单独的库。缩进的使用非常重要：处于一个区域中的内容必须缩进。

```
library
    Exposed-Modules: Prettify
                     PrettyJSON
                     SimpleJSON
    Build-Depends:   base >= 2.0
```

Exposed-Modules 列出了本包中用户可用的模块。可选字段字段 Other-Modules 列出了内部模块。这些内部模块用来支持这个库的功能，然而对用户不可见。

Build-Depends 包含了构建我们库所需要的包，它们之间用逗号分开。对于每一个包，我们可以选择性地说明这个库可以与之工作的版本号范围。base 包包含了很多 Haskell 的核心模块，如 Prelude，因此实际上它总是被需要的。

Note: 处理依赖关系

我们并不需要猜测或者调查我们依赖于哪些包。如果我们在构建包的时候没有包含 Build-Depends 字段，编译会失败，并返回一条有用的错误信息。我们可以试试把 base 注释掉会发生什么。

```
$ runghc Setup build
Preprocessing library mypretty-0.1...
Building mypretty-0.1...

PrettyJSON.hs:8:7:
    Could not find module `Data.Bits':
        it is a member of package base, which is hidden
```

错误信息清楚地表明我们需要增加 `base` 包，尽管它已经被安装了。强制我们显式地列出所有包有一个实际好处：`cabal-install` 这个命令行工具会自动下载，构建并安装一个包和所有它依赖的包。

[译注，在运行 `runghc Setup build` 之前，Cabal 会首先要求你运行 `configure`。具体方法见下文。]

5.14.2 GHC 的包管理器

GHC 内置了一个简单的包管理器用来记录安装了哪些包以及它们的版本号。我们可以使用 **ghc-pkg** 命令来查看包数据库。

我们说*数据库*，是因为 GHC 区分所有用户都能使用的*系统包*（*system-wide packages*）和只有当前用户才能使用的*用户包*（*per-user packages*）。用户数据库（per-user database）使我们没有管理员权限也可以安装包。

ghc-pkg 命令为不同的任务提供了不同的子命令。大多数时间，我们只用到两个。`ghc-pkg list` 命令列出已安装的包。当我们想要卸载一个包时，`ghc-pkg unregister` 告诉 GHC 我们不再用这个包了。（我们需要手动删除已安装的文件。）

5.14.3 配置，构建和安装

除了 `.cabal` 文件，每个包还必须包含一个 *setup* 文件。这使得 Cabal 可以在需要的时候自定义构建过程。一个最简单的配置文件如下所示。

```
-- file: ch05/Setup.hs
#!/usr/bin/env runhaskell
import Distribution.Simple
main = defaultMain
```

我们把这个文件保存为 `Setup.hs`。

有了 `.cabal` 和 `Setup.hs` 文件之后，我们只有三步之遥。

我们用一个简单的命令告诉 Cabal 如何构建一个包以及往哪里安装这个包。

[译注：运行此命令时，Cabal 提示我没有指定 `build-type`。于是我按照提示在 `.cabal` 文件里加了 `build-type: Simple` 字段。]

```
$ runghc Setup configure
```

这个命令保证了我们的包可用，并且保存设置让后续的 Cabal 命令使用。

如果我们不给 configure 提供任何参数，Cabal 会把我们的包安装在系统包数据库里。如果想安装在指定目录下和用户包数据库内，我们需要提供更多的信息。

```
$ runghc Setup configure --prefix=$HOME --user
```

完成之后，我们来构建这个包。

```
$ runghc Setup build
```

成功之后，我们就可以安装包了。我们不需要告诉 Cabal 装在哪儿，它会使用我们在第一步里提供的信息。它会把包装在我们指定的目录下然后更新 GHC 的用户包数据库。

```
$ runghc Setup install
```

5.15 实用链接和扩展阅读

GHC 内置了一个美观打印库，`Text.PrettyPrint.HughesPJ`。它提供的 API 和我们的例子相同并且有更丰富有用的美观打印函数。与自己实现相比，我们更推荐使用它。

John Hughes 在 [Hughes95] 中介绍了 `HughesPJ` 美观打印器的设计。这个库后来被 Simon Peyton Jones 改进，也因此得名。Hughes 的论文很长，但他对怎样设计 Haskell 库的讨论非常值得一读。

本章介绍的美观打印库基于 Philip Wadler 在 [Wadler98] 中描述的一个更简单的系统。Daan Leijen 扩展了这个库，扩展之后的版本可以从 Hackage 里下载：`wl-pprint`。如果你用 **cabal** 命令行工具，一个命令即可完成下载，构建和安装：**cabal install wl-pprint**。

CHAPTER

SIX

第 6 章：使用类型类

类型类（typeclass）跻身于 Haskell 最强大功能之列：它们（typeclasses）允许你定义通用接口，而其（这些接口）为各种不同的类型（type）提供一组公共特性集。类型类是某些基本语言特性的核心，比如相等性测试（equality testing）和数值操作符（numeric operators）。在讨论到底类型类是什么之前，我想解释下他们的作用（the need for them）。

6.1 类型类的作用

假设因为某个原因，Haskell 语言的设计者拒绝实现相等性测试 == ，因此我们决定实现自己的 == 操作。你的应用由一个简单的 Color 类型组成。首先你尝试一下，像这样：

```
-- file: ch06/naiveeq.hs
data Color = Red | Green | Blue

colorEq :: Color -> Color -> Bool
colorEq Red   Red   = True
colorEq Green Green = True
colorEq Blue  Blue  = True
colorEq _     _     = False
```

让我们在 ghci 里测试一下：

```
Prelude> :l naiveeq.hs
[1 of 1] Compiling Main             ( naiveeq.hs, interpreted )
Ok, modules loaded: Main.
*Main> colorEq Green Green
True
*Main> colorEq Red Red
True
*Main> colorEq Red Green
False
```

现在，假设你想要添加 String 的相等性测试 (equality testing)。因为一个 Haskell 的 String 其实是字符们（characters）的列表 (即 [char])，所以我们可以写一个小函数来运行那个测试 (相等性测试)。为了简单（偷懒）起见，我们作一下弊：使用 == 操作符。

```
-- file: ch06/naiveeq.hs
stringEq :: [Char] -> [Char] -> Bool

-- Match if both are empty
stringEq [] [] = True

-- If both start with the same char, check the rest
stringEq (x:xs) (y:ys) = x == y && stringEq xs ys

-- Everything else doesn't match
stringEq _ _ = False
```

让我们运行一下：

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

*Main> stringEq "" ""
True

*Main> stringEq "" []
True

*Main> stringEq "" [""]
<interactive>:5:14:
Couldn't match expected type `Char' with actual type `[Char]'
In the expression: ""
In the second argument of `stringEq', namely `[""]'
In the expression: stringEq "" [""]
```

现在你应该能看出一个问题了吧：我们不得不为各个不同类型（type）实现一坨带有不同名字的函数（function），以便我们有能力用其进行比较。这种做法非常低效，而且烦人。如果我们能用 == 对比任何类型的值，就再方便不过了。

同时，我们能定义一些通用（generic）函数，比如基于 == 的 /= ，其能对几乎任何东西（anything）合法。通过写一个通用函数，其能比较所有的东西，也能使我们的代码一般化（generic）：如果一段代码仅需要比较（compare）一些东西，然后他应该就能够接受任何数据类型，而对其（这些类型）编译器是知道如何比较的。

而且更进一步，如果以后新类型被添加进来，现有的代码不应该被修改。而 Haskell 的类型类（typeclass）就是被设计成处理上面的这些破事的。

6.2 什么是类型类？

类型类定义了一系列函数，而这些函数对于不同类型的值使用不同的函数实现。它和面向对象（object-oriented）语言的对象（objects）有些类似，但是他们是完全不同的。

[huangz，labyrlnth，YenvY 等译者注：这里原文是将“面向对象编程中的对象”和 Haskell 的类型类进行类比，但实际上这种类比并不太恰当，类比成接口和多态方法更适合一点。] [sancao2 译注: 我觉得作者不是不知道类型类应该与接口和多态方法类比，他这么说的原因是下面他自己的注释”When is a class not a class?”里面说的，因为类型类的关键词是 class，传统面向对象编程里面的关键词也是 class。]

让我们使用类型类来解决我们章节前面相等性测试的困局。首先，我们定义类型类本身。我们需要一个函数，其接受两个参数。每个参数拥有相同的类型，然后返回一个 `Bool` 类型以指示他们是否相等。我们不关心这些类型到底是什么，但我需要的是同一个类型的两项 (items)。

下面是我们的类型的初定义：

```
-- file: ch06/eqclasses.hs
class BasicEq a where
    isEqual :: a -> a -> Bool
```

这个定义说，我们申明（使用 `class` 关键字）了一个类型类（typeclass），其名字叫 `BasicEq` 。接着我们将引用（refer to）实例类型（instance types），带着字母 `a` 作名字。一个类型类的实例类型可以是任何类型，只要其（实例类型）实现了类型类中定义的函数。这个类型类定义了一个函数 (`isEqual`)，而这个函数接受两个参数，他们（这俩参数）对应于实例类型即 `a` ，并且返回一个 `Bool` 型。

在定义的第一行，参数（实例类型）的名字是任选的。就是说，我们能使用任意名字。关键之处在于，当我们列出函数的类型时，我们必须使用相同的名字引用实例类型们（instance types）。比如说，我们使用 `a` 来表示实例类型，那么函数签名中也必须使用 `a` 来代表这个实例类型。

让我们在 `ghci` 看一下 `isEqual` 的类型。回想一下，在 `ghci` 我们能用 `:type` （简写 `:t` ）来查看某些东西的类型。

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

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

这种方式让我们读出：" 对于所有的类型 `a` ，只要 `a` 是 `BasicEq` 的一个实例，`isEqual` 就能接受两个类型为 `a` 的参数，并返回一个 `Bool` 。 " 让我们快速地浏览一遍为某个特定类型定义的 `isEqual` 吧。

```
-- file: ch06/eqclasses.hs
instance BasicEq Bool where
    isEqual True  True  = True
```

(continues on next page)

(continued from previous page)

```
    isEqual False False = True
    isEqual _     _     = False
```

你能用 ghci 来验证我们基于 Bool 类的 isEqual ，而不是基于其他实例类型的。

```
*Main> isEqual True True
True

*Main> isEqual False True
False

*Main> isEqual "hello" "moto"

<interactive>:5:1:
    No instance for (BasicEq [Char])
          arising from a use of `isEqual'
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "hello" "moto"
    In an equation for `it': it = isEqual "hello" "moto"
```

注意，当我们试图比较两个字符串，ghci 抱怨到，“我们没有提供基于 [Char] 实例类型的 BasicEq ，所以他不知道如何去比较 [Char] 。” 并且其建议（” Possible fix”）我们可以通过定义基于 [Char] 实例类型的 BasicEq 。

稍后的一节我们将会详细介绍定义实例（instances）。不过，首先让我们继续看定义类型类 (typeclass)。在这个例子中，一个 " 不相等 " （not-equal-to）函数可能很有用。这里我们可以做的是，定义一个带两个函数的类型类（typeclass）：

```
-- file: ch06/eqclasses.hs
class BasicEq2 a where
    isEqual2    :: a -> a -> Bool
    isNotEqual2 :: a -> a -> Bool
```

如果有人要提供一个 BasicEq2 的实例 (instance), 那么他将要定义两个函数：isEqual2 和 isNotEqual2 。当我们定义好以上的 BasicEq2 ，看起来我们为自己制造了额外的工作。从逻辑上讲，如果我们知道 isEqual2 或 isNotEqual2 返回的是什么，那么我们就可以知道另外一个函数的返回值，对于所有（输入）类型来说。为了避免让类型类的用户为所有类型都定义两个函数，我们可以提供他们（两个函数）的默认实现。然后，用户只要自己实现其中一个就可以了。这里的例子展示了如何实现这种手法。

```
-- file: ch06/eqclasses.hs
class BasicEq3 a where
    isEqual3 :: a -> a -> Bool
    isEqual3 x y = not (isNotEqual3 x y)
```

(continues on next page)

(continued from previous page)

```
    isNotEqual3 :: a -> a -> Bool
    isNotEqual3 x y = not (isEqual3 x y)
```

人们实现这个类型类必须提供至少一个函数的实现。当然他们可以实现两个，如果他们乐意，但是他们不必被强制（这么做）。虽然我们提供两个函数的默认实现，每个函数取决于另外一个来计算答案。如果我们不指定至少一个，所产生的代码将是一个无尽循环。因此，至少得有一个函数总是要被实现。

以下是将 Bool 作为 BasicEq3 实例类型的例子。

```
-- file: ch06/eqclasses.hs
instance BasicEq3 Bool where
    isEqual3 False False = True
    isEqual3 True  True  = True
    isEqual3 _     _     = False
```

我们只要定义 isEqual3 函数，就可以“免费”得到 isNotEqual3：

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

*Main> isEqual True True
True

*Main> isEqual False False
True

*Main> isNotEqual False True
True
```

用 BasicEq3 ，我们提供了一个类型类 (class)，其行为类似于 Haskell 原生的 == 和 /= 操作符。事实上，这些操作符本来就是被一个类型类定义的，其看起来几乎等价于 BasicEq3 。“Haskell 98 Report”定义了一个类型类，它实现了相等性比较 (equality comparison)。这是内建类型类 Eq 的代码。注意到他和我们的 BasicEq3 类型类多么相似呀。

```
class  Eq a  where
 (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition:
    --      (==) or (/=)
 x /= y     =  not (x == y)
 x == y     =  not (x /= y)
```

6.3 定义类型类实例

现在你知道了怎么定义一个类型类，是时候学习一下怎么定义某个类型类的实例 (instance)。回忆一下那些用于创造某个特定类型类的实例的类型们 (types)，他们是通过实现对那个类型类必须的函数来实现的。回忆一下我们位于章节前面的尝试 (attemp)，针对 `Color` 类型创造的相等性测试。

那么让我们看看我们要怎样创造同样的 `Color` 类型，作为 `BasicEq3` 类型类的一员。

```
-- file: ch06/naiveeq.hs
instance BasicEq3 Color where
    isEqual3 Red Red = True
    isEqual3 Blue Blue = True
    isEqual3 Green Green = True
    isEqual3 _ _ = False
```

注意，这里的函数定义和之前“类型类的作用”章节的 `colorEq` 函数定义实际上没有什么不同。事实上，它的实现就是等价的。然而，在本例中，我们能将 `isEqual3` 使用于 * 任何 * 类型上，只要其 (该类型) 声明成 `BasicEq3` 的一个实例 (instance)，而不仅仅限于 `Color` 一类。我们能定义相等性测试，针对任何东西，从数值到图形，通过采用相同的基本模式 (basic pattern) 的方式。事实上，我们将会在“相等性，有序和对比”章节中看到，这就是你能使 Haskell 的 == 操作符作用于你自己的类型的方式。

还要注意到，虽然 `BasicEq3` 类型类定义了两个函数 `isEqual` 和 `isNotEqual` ，但是我们只实现了其中的一个，在 `Color` 的例子中。那得归功于包含于 `BasicEq3` 中的默认实现。即使我们没有显式地定义 `isNotEqual3` ，编译器也会自动地使用 `BasicEq3` 声明中的默认实现。

6.4 重要的内置类型类

前面两节我们分别讨论了 (如何) 定义你自己的类型类 (typeclass)，以及如何创造你自己的类型类实例 (type instance)。

是时候介绍几个作为 `Prelude` 库一部分的类型类。如本章开始时所说的，类型类处于 Haskell 语言某些重要特性的中心。我们将讨论最常见的几个。更多细节，” Haskell library reference” 是一个很好的资源。其将给你介绍类型类，并且将一直告诉你什么函数是你必须要实现的以获得一份完整的定义。

6.4.1 Show

`Show` 类型类用于将值 (values) 转换为字符串 (Strings)，其最常用的（功能）可能是将数值 (numbers) 转换成字符串，但是他被定义成如此多类型以至于能转化相当多东西。如果你已经定义了你自己的类型们 (types)，创造他们 (types) `Show` 的实例，将会使他们能够在 `ghci` 中展示或者在程序中打印出来。`Show` 类型类中最重要的函数是 `show` 。其接受一个参数，以用于数据 (data) 转换，并返回一个 `String` ，以代表这个数据 (data)。

```
Main> :type show
show :: Show a => a -> String
```

让我们看看一些例子，关于转化数值到字符串的。

```
Main> show 1
"1"

Main> show [1, 2, 3]
"[1,2,3]"

Main> show (1, 2)
"(1,2)"
```

记住 ghci 显示出结果，就像你进入一个 Haskell 的程序。所以表达式 show 1 返回一个包含数字 1 的单字符的字符串。即引号不是字符串本身的一部分。我们将使用 putStrLn 明确这一点。

```
ghci> putStrLn (show 1)
1
ghci> putStrLn (show [1,2,3])
[1,2,3]
```

你也可以将 show 用在 String 上面。

```
ghci> show "Hello!"
"\"Hello!\""
ghci> putStrLn (show "Hello!")
"Hello!"
ghci> show ['H', 'i']
"\"Hi\""
ghci> putStrLn (show "Hi")
"Hi"
ghci> show "Hi, \"Jane\""
"\"Hi, \\\"Jane\\\"\""
ghci> putStrLn (show "Hi, \"Jane\"")
"Hi, \"Jane\""
```

运行 show 于 String 之上，可能使你感到困惑。因为 show 生成了一个结果，其相配（suitable）于 Haskell 的字面值 (literal)，或者说，show 添加了引号和转义符号 (“)，其适用于 Haskell 程序内部。ghci 也用 show 来显示结果，所以引号和转义符号被添加了两次。使用 putStrLn 能帮助你明确这种差异。

你能轻易地定义你自己的 Show 实例，如下。

```
-- file: ch06/naiveeq.hs
```

(continues on next page)

(continued from previous page)

```
instance Show Color where
    show Red   = "Red"
    show Green = "Green"
    show Blue  = "Blue"
```

上面的例子定义了 Show 类型类的实例，其针对我们章节前面的定义的类型 Color 。

Note: Show 类型类

show 经常用于定义数据 (data) 的字符串 (String) 表示，其非常有利于机器使用用 Read 类型类解析回来。Haskell 程序员经常写自己的函数去格式化 (format) 数据以漂亮的方式为终端用户呈现，如果这种表示方式有别于 Show 预期的输出。

因此，如果你定义了一种新的数据类型，并且希望通过 ghci 来显示它，那么你就应该将这个类型实现为 Show 类型类的实例，否则 ghci 就会向你抱怨，说它不知道该怎样用字符串的形式表示这种数据类型：

```
Main> data Color = Red | Green | Blue;

Main> show Red

<interactive>:10:1:
    No instance for (Show Color)
        arising from a use of `show'
    Possible fix: add an instance declaration for (Show Color)
    In the expression: show Red
    In an equation for `it': it = show Red

Prelude> Red

<interactive>:5:1:
    No instance for (Show Color)
        arising from a use of `print'
    Possible fix: add an instance declaration for (Show Color)
    In a stmt of an interactive GHCi command: print it
```

通过实现 Color 类型的 show 函数，让 Color 类型成为 Show 的类型实例，可以解决以上问题：

```
-- file: ch06/naiveeq.hs
instance Show Color where
    show Red   = "Red"
    show Green = "Green"
    show Blue  = "Blue"
```

当然，show 函数的打印值并不是非要和类型构造器一样不可，比如 Red 值并不是非要表示为 "Red" 不可，以下是另一种实例化 Show 类型类的方式：

```
-- file: ch06/naiveeq.hs
instance Show Color where
    show Red   = "Color 1: Red"
    show Green = "Color 2: Green"
    show Blue  = "Color 3: Blue"
```

6.4.2 Read

Read 类型类，本质上和 Show 类型类相反: 其 (Read) 最有用的函数是 read ，它接受一个字符串作为参数，对这个字符串进行解析 (parse)，并返回一个值。这个值的类型为 Read 实例类型的成员（所有实例类型中的一种）。

```
Prelude> :type read
read :: Read a => String -> a
```

这是一个例子，展示了 read 和 show 函数的用法:

```
-- file: ch06/read.hs
main = do
  putStrLn "Please enter a Double:"
  inpStr <- getLine
  let inpDouble = (read inpStr)::Double
  putStrLn ("Twice " ++ show inpDouble ++ " is " ++ show (inpDouble * 2))
```

测试结果如下：

```
Prelude> :l read.hs
[1 of 1] Compiling Main             ( read.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Please enter a Double:
123.213
Twice 123.213 is 246.426
```

这是一个简单的例子，关于 read 和 show。请注意，我们给出了一个显式的 Double 类型，当运行 read 函数的时候。

那是因为 read 会返回任意类型的值 (a value of type) Read a => a ，并且 show 期望任意类型的值 Show a => a 。存在着许许多多类型 (type)，其拥有定义于 Read 和 Show 之上的实例 (instance)。

不知道一个特定的类型，编译器必须从许多类型中猜出那个才是必须的 (needed)。在上面的这种情况下，他

可能会经常选择 Integer 类型。如果我们想要接受的是浮点输入，他就不会正常工作，所以我们提供了一个显式的类型。

Note: 关于默认值的笔记

在大多数情况下，如果显式的 Double 类型标记被忽略了，编译器会拒绝猜测一个通用的类型，并仅仅返回一个错误。他能默认以 Integer 类型这件事请是个特例。他起因于以下事实: 字面值 2 (在程序中 inpDouble * 2) 被当成 Integer 除非他得到一个不同类型的期望。]

你能看到相同的效果在起作用，如果你试着在 ghci 命令行中使用 read 。ghci 内部使用 show 来展示结果，意味着你可能同样会碰到一样会碰到模棱两可的类型问题。你将须要显式地指定类型于 read 的结果在 ghci 当中，如下。

```
Prelude> read "3"

<interactive>:5:1:
    Ambiguous type variable `a0' in the constraint:
          (Read a0) arising from a use of `read'
    Probable fix: add a type signature that fixes these type variable(s)
    In the expression: read "3"
    In an equation for `it': it = read "3"

Prelude> (read "3")::Int
3

Prelude> :type it
it :: Int

Prelude> (read "3")::Double
3.0

Prelude> :type it
it :: Double
```

注意，在第一次调用 read 的时候，我们并没有显式地给定类型签名，这时对 read "3" 的求值会引发错误。这是因为有非常多的类型都是 Read 的实例，而编译器在 read 函数读入 "3" 之后，不知道应该将这个值转换成什么类型，于是编译器就会向我们发牢骚。

因此，为了让 read 函数返回正确类型的值，必须给它指示正确的类型。

回想一下，read 函数的类型签名: (Read a) => String -> a 。a 在这里是 Read 类型类的任何实例类型。其特定的解析函数被调用取决于 read 返回值的期望类型。让我们看看他是怎么工作的。

```
ghci> (read "5.0")::Double
5.0
ghci> (read "5.0")::Integer
*** Exception: Prelude.read: no parse
```

注意到错误 (将发生) 当你试图解析 `5.0` 作为一个整数 `Integer` 。解释器选择了一个不同的 `Read` 实例：当返回值的期望是 `Integer` ，而他做的却是期望得到一个 `Double` 。`Integer` 的解析器不能接受小数点，从而抛出一个异常。

`Read` 类型提供了一些相当复杂的解析器。你可以定义一个简单的解析器，通过提供 `readsPrec` 函数的实现。你的实现能返回一个列表 (list)：该列表在解析成功时包含一个元组 (tuple)，在解析失败时为空。下面是一个实现的例子。

```
-- file: ch06/naiveeq.hs
instance Read Color where
    -- readsPrec is the main function for parsing input
    readsPrec _ value =
        -- We pass tryParse a list of pairs.  Each pair has a string
        -- and the desired return value.  tryParse will try to match
        -- the input to one of these strings.
        tryParse [("Red", Red), ("Green", Green), ("Blue", Blue)]
        where tryParse [] = []    -- If there is nothing left to try, fail
              tryParse ((attempt, result):xs) =
                   -- Compare the start of the string to be parsed to the
                   -- text we are looking for.
                   if (take (length attempt) value) == attempt
                      -- If we have a match, return the result and the
                      -- remaining input
                      then [(result, drop (length attempt) value)]
                      -- If we don't have a match, try the next pair
                      -- in the list of attempts.
                      else tryParse xs
```

运行测试一下:

```
*Main> :l naiveeq.hs
[1 of 1] Compiling Main             ( naiveeq.hs, interpreted )
Ok, modules loaded: Main.
*Main> (read "Red")::Color
Color 1: Red
*Main> (read "Green")::Color
Color 2: Green
*Main> (read "Blue")::Color
Color 3: Blue
```

(continues on next page)

(continued from previous page)

```
*Main> (read "[Red]")::Color
*** Exception: Prelude.read: no parse
*Main> (read "[Red]")::[Color]
[Color 1: Red]
*Main> (read "[Red,Green,Blue]")::[Color]
[Color 1: Red,Color 2: Green,Color 3: Blue]
*Main> (read "[Red, Green, Blue]")::[Color]
*** Exception: Prelude.read: no parse
```

注意到最后的尝试产生了错误。那是因为我们的编译器没有聪明到可以处理置位 (leading，包括前置和后置) 的空格。你可以改进他，通过些改你的 Read 实例以忽略任何置位的空格。这在 Haskell 程序中是常见的做法。

6.4.3 使用 Read 和 Show 进行序列化

很多时候，程序需要将内存中的数据保存为硬盘上的文件以备将来获取，或者通过网络发送出去。把内存中的数据转化成为，为存储目的，序列的过程，被称为 *序列化*。

通过将类型实现为 Read 和 Show 的实例类型，read 和 show 两个函数可以成为非常好的序列化工具。show 函数生成的输出是人类和机器皆可读的。大部分 show 输出也是对 Haskell 语法合法的，虽然他取决于人们如何写 Show 实例来达到这个结果。

Note: 解析超大（large）字符串

字符串处理在 Haskell 中通常是惰性的，所以 read 和 show 能被无意外地用于很大的数据结构。Haskell 中内建的 read 和 show 实例被实现成高效的纯函数。如果想知道怎么处理解析的异常，请参考”19 章错误处理”。

作为例子，以下代码将一个内存中的列表序列化到文件中：

```
Prelude> let years = [1999, 2010, 2012]

Prelude> show years
"[1999,2010,2012]"

Prelude> writeFile "years.txt" (show years)
```

writeFile 将给定内容写入到文件当中，它接受两个参数，第一个参数是文件路径，第二个参数是写入到文件的字符串内容。

观察文件 years.txt 可以看出，(show years) 所产生的文本被成功保存到了文件当中：

```
$ cat years.txt
[1999,2010,2012]
```

使用以下代码可以对 years.txt 进行反序列化操作：

```
Prelude> input <- readFile "years.txt"

Prelude> input                    -- 读入的字符串
"[1999,2010,2012]"

Prelude> (read input)::[Int]      -- 将字符串转换成列表
[1999,2010,2012]
```

readFile 读入给定的 years.txt ，并将它的内存传给 input 变量。最后，通过使用 read ，我们成功将字符串反序列化成一个列表。

6.4.4 数值类型

Haskell 有一个非常强大的数值类型集合：从速度飞快的 32 位或 64 位整数，到任意精度的有理数，无所不包。你可能知道操作符 (比如 (+)) 能作用于所有的这些类型。这个特性是用类型 (typeclass) 类实现的。作为附带的好处，他 (Haskell) 允许你定义自己的数值类型，并且把他们当做 Haskell 的一等公民 (first-class citizens)。

让我们开始讨论，关于围绕在数值类型 (numberic types) 周围的类型类们 (typeclass)，用以类型们 (type) 本身的检查 (examination)。以下表格显示了 Haskell 中最常用的一些数值类型。请注意，存在这更多数值类型用于特定的目的，比如提供接口给 C 。

表格 6.1： 部分数值类型

类型	介绍
Double	双精度浮点数。表示浮点数的常见选择。
Float	单精度浮点数。通常在对接 C 程序时使用。
Int	固定精度带符号整数；最小范围在 -2^29 至 2^29-1 。相当常用。
Int8	8 位带符号整数
Int16	16 位带符号整数
Int32	32 位带符号整数
Int64	64 位带符号整数
Integer	任意精度带符号整数；范围由机器的内存限制。相当常用。
Rational	任意精度有理数。保存为两个整数之比（ratio）。
Word	固定精度无符号整数。占用的内存大小和 `Int` 相同
Word8	8 位无符号整数
Word16	16 位无符号整数
Word32	32 位无符号整数
Word64	64 位无符号整数

这是相当多的数值类型。存在这某些操作符，比如加号（`(+)`），其能在他们中的所有之上工作。另外的一部分函数，比如 `asin` ，只能用于浮点数类型。

以下表格汇总了操作 (operate) 于不同类型的不同函数。当你读到表，记住，Haskell 操作符们 (operators) 只是函数。你可以通过 `(+) 2 3` 或者 `2 + 3` 得到相同的结果。按照惯例，当讲操作符当做函数时，他们被写在括号中，如下表 6.2。

表格 6.2： 部分数值函数和常量

项	类型	模块	描述
(+)	Num a => a -> a -> a	Prelude	加法
(-)	Num a => a -> a -> a	Prelude	减法
(*)	Num a => a -> a -> a	Prelude	乘法
(/)	Fractional a => a -> a -> a	Prelude	份数除法
(**)	Floating a => a -> a -> a	Prelude	乘幂
(^)	(Num a, Integral b) => a -> b -> a	Prelude	计算某个数的非负整数次方
(^^)	(Fractional a, Integral b) => a -> b -> a	Prelude	分数的任意整数次方
(%)	Integral a => a -> a -> Ratio a	Data.Ratio	构成比率
(.&.)	Bits a => a -> a -> a	Data.Bits	二进制并操作
(.|.)	Bits a => a -> a -> a	Data.Bits	二进制或操作
abs	Num a => a -> a	Prelude	绝对值操作
approxRational	RealFrac a => a -> a -> Rational	Data.Ratio	通过分数的分子和分母计算出近似有理数

Continued on ne

Table 1 – continued from previous page

项	类型	模块	描述
cos	Floating a => a -> a	Prelude	余弦函数。另外还有 acos 、cosh 和 acosh ，类型和 cos
div	Integral a => a -> a -> a	Prelude	整数除法，总是截断小数位。
fromInteger	Num a => Integer -> a	Prelude	将一个 Integer 值转换为任意数值类型。
fromIntegral	(Integral a, Num b) => a -> b	Prelude	一个更通用的转换函数，将任意 Integral 值转为任意数
fromRational	Fractional a => Rational -> a	Prelude	将一个有理数转换为分数。可能会有精度损失。
log	Floating a => a -> a	Prelude	自然对数算法。
logBase	Floating a => a -> a -> a	Prelude	计算指定底数对数。
maxBound	Bounded a => a	Prelude	有限长度数值类型的最大值。
minBound	Bounded a => a	Prelude	有限长度数值类型的最小值。
mod	Integral a => a -> a -> a	Prelude	整数取模。
pi	Floating a => a	Prelude	圆周率常量。
quot	Integral a => a -> a -> a	Prelude	整数除法；商数的分数部分截断为 0 。
recip	Fractional a => a -> a	Prelude	分数的倒数。
rem	Integral a => a -> a -> a	Prelude	整数除法的余数。
round	(RealFrac a, Integral b) => a -> b	Prelude	四舍五入到最近的整数。
shift	Bits a => a -> Int -> a	Bits	输入为正整数，就进行左移。如果为负数，进行右移。
sin	Floating a => a -> a	Prelude	正弦函数。还提供了 asin 、sinh 和 asinh ，和 sin 类型
sqrt	Floating a => a -> a	Prelude	平方根
tan	Floating a => a -> a	Prelude	正切函数。还提供了 atan 、tanh 和 atanh ，和 tan 类型
toInteger	Integral a => a -> Integer	Prelude	将任意 Integral 值转换为 Integer
toRational	Real a => a -> Rational	Prelude	从实数到有理数的有损转换
truncate	(RealFrac a, Integral b) => a -> b	Prelude	向着零截断
xor	Bits a => a -> a -> a	Data.Bits	二进制异或操作

“数值类型及其对应的类型类”列举在下表 6.3。

表格 6.3： 数值类型的类型类实例

类型	Bits	Bounded	Floating	Fractional	Integral	Num	Real	RealFrac
Double			X	X		X	X	X
Float			X	X		X	X	X
Int	X	X			X	X	X	
Int16	X	X			X	X	X	
Int32	X	X			X	X	X	
Int64	X	X			X	X	X	
Integer	X				X	X	X	
Rational or any Ratio				X		X	X	X
Word	X	X			X	X	X	
Word16	X	X			X	X	X	
Word32	X	X			X	X	X	
Word64	X	X			X	X	X	

表格 6.4 列举了一些数值类型之间进行转换的函数，以下表格是一个汇总：

表格 6.4： 数值类型之间的转换

源类型	目标类型			
	Double, Float	Int, Word	Integer	Rational
Double, Float Int, Word Integer Rational	fromRational . toRational fromIntegral fromIntegral fromRational	truncate * fromIntegral fromIntegral truncate *	truncate * fromIntegral N/A truncate *	toRational fromIntegral fromIntegral N/A

6.4 表中 * 代表除了 `truncate` (向着零截断) 之外，还可以使用 `round` (最近整数)、`ceiling` (上取整) 或者 `floor` (下取整) 的类型。

第十三章会说明，怎样用自定义数据类型来扩展数值类型。

6.4.5 相等性，有序和对比

我们已经讨论过了算术符号比如 `(+)` 能用到不同数字的所有类型。但是 Haskell 中还存在着某些甚至更加广泛使用的操作符。最显然地，当然，就是相等性测试: `(==)` 和 `(/=)` ，这两操作符们都定义于 `Eq` 类 (class) 中。

存在着其他的比较操作符，如 >= 和 <= ，其则由 `Ord` 类型类定义。他们 (`Ord`) 是放在于单独类中是因为存在着某些类型，比如 `Handle` ，使在这些地方相等性测试有意义 (make sense)，而表达特定的序 (ording) 一点意义都没有。

所有 Ord 实例都可以使用 Data.List.sort 来排序。

几乎所有 Haskell 内置类型都是 Eq 类型类的实例，而 Ord 类的实例类型也几乎一样多。

Tip: Ord 产生的排列顺序在某些时候是非常随意的，比如对于 Maybe 而言，Nothing 就排在 Just x 之前，这些都是随意决定的，并没有什么特殊的意义。

6.5 自动派生

对于许多简单的数据类型，Haskell 编译器可以自动将类型派生（derivation）为 Read 、Show 、Bounded 、Enum 、Eq 和 Ord 的实例 (instance)。这节省了我们大量的精力用于手动写代码进行比较或者显示他们的类型。

以下代码将 Color 类型派生为 Read 、Show 、Eq 和 Ord 的实例：

```
-- file: ch06/colorderived.hs
data Color = Red | Green | Blue
    deriving (Read, Show, Eq, Ord)
```

让我们看看这些派生实例们是怎么工作的：

```
*Main> show Red
"Red"

*Main> (read "Red")::Color
Red

*Main> (read "[Red, Red, Blue]")::[Color]
[Red,Red,Blue]

*Main> Red == Red
True

*Main> Data.List.sort [Blue, Green, Blue, Red]
[Red,Green,Blue,Blue]

*Main> Red < Blue
True
```

Note: 什么类型 (types) 能被自动派生?

Haskell 标准要求编译器能自动派生这些指定类型类的实例。

注意 Color 类型的排序位置由定义类型时值构造器的排序决定，即对应上面例子就是 Red | Green | Blue 的顺序。

自动派生并不总是可用的。比如说，如果定义类型 data MyType = MyType (Int -> Bool)，那么编译器就没办法派生 MyType 为 Show 的实例，因为它不知道该怎么渲染 (render) 一个函数。在上面这种情况下，我们会得到一个编译错误。

当我们自动派生某个类型类的一个实例时，在我们利用 data 关键词声明参考这个实例的类型时，也必须是给定类型类的实例（手动或自动地）。

举个例子，以下代码不能使用自动派生：

```
-- file: ch06/cant_ad.hs
data Book = Book

data BookInfo = BookInfo Book
                deriving (Show)
```

ghci 会给出提示，说明 Book 类型也必须是 Show 的实例，BookInfo 才能对 Show 进行自动派生 (driving)：

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

ad.hs:4:27:
    No instance for (Show Book)
          arising from the 'deriving' clause of a data type declaration
    Possible fix:
        add an instance declaration for (Show Book)
        or use a standalone 'deriving instance' declaration,
        so you can specify the instance context yourself
    When deriving the instance for (Show BookInfo)
Failed, modules loaded: none.
```

相反，以下代码可以使用自动派生，因为它对 Book 类型也使用了自动派生，使得 Book 类型变成了 Show 的实例：

```
-- file: ch06/ad.hs
data Book = Book
            deriving (Show)

data BookInfo = BookInfo Book
                deriving (Show)
```

使用 :info 命令在 ghci 中确认两种类型都是 Show 的实例：

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

*Main> :info Book
data Book = Book    -- Defined at ad.hs:1:6
instance Show Book -- Defined at ad.hs:2:23

*Main> :info BookInfo
data BookInfo = BookInfo Book    -- Defined at ad.hs:4:6
instance Show BookInfo -- Defined at ad.hs:5:27
```

6.6 类型类实战：让 JSON 更好用

我们在*在 Haskell 中表示 JSON 数据* 一节介绍的 JValue 用起来还不够简便。这里是一段由的经过截断 (truncate) 和整齐化 (tidy) 之后的实际 JSON 数据，由一个知名搜索引擎生成。

```
{
    "query": "awkward squad haskell",
    "estimatedCount": 3920,
    "moreResults": true,
    "results":
    [{
        "title": "Simon Peyton Jones: papers",
        "snippet": "Tackling the awkward squad: monadic input/output ...",
        "url": "http://research.microsoft.com/~simonpj/papers/marktoberdorf/",
    },
    {
        "title": "Haskell for C Programmers | Lambda the Ultimate",
        "snippet": "... the best job of all the tutorials I've read ...",
        "url": "http://lambda-the-ultimate.org/node/724",
    }]
}
```

这是进一步缩减片段的数据，并用 Haskell 表示：

```
-- file: ch06/SimpleResult.hs
import SimpleJSON

result :: JValue
result = JObject [
```

(continues on next page)

(continued from previous page)

```
    ("query", JString "awkward squad haskell"),
    ("estimatedCount", JNumber 3920),
    ("moreResults", JBool True),
    ("results", JArray [
        JObject [
        ("title", JString "Simon Peyton Jones: papers"),
        ("snippet", JString "Tackling the awkward ..."),
        ("url", JString "http://.../marktoberdorf/")
        ]])
    ]
```

由于 Haskell 不原生支持包含不同类型值的列表，我们不能直接表示包含不同类型值的 JSON 对象。我们需要把每个值都用 JValue 构造器包装起来。但这样我们的灵活性就受到了限制：如果我们想把数字 `3920` 转换成字符串 `"3,920"`，我们就必须改变构造器，即我们使用它 (JValue 构造器) 从 `JNumber` 构造器到 `JString` 构造器包装 (wrap) 数据。

Haskell 的类型类对这个问题提供了一个诱人的解决方案：

```
-- file: ch06/JSONClass.hs
type JSONError = String

class JSON a where
    toJValue :: a -> JValue
    fromJValue :: JValue -> Either JSONError a

instance JSON JValue where
    toJValue = id
    fromJValue = Right
```

现在，我们无需再用 `JNumber` 等构造器去包装值了，直接使用 `toJValue` 函数即可。如果我们更改值的类型，编译器会自动选择合适的 `toJValue` 实现以使用他。

我们也提供了 `fromJValue` 函数. 它试图把 `JValue` 值转换成我们希望的类型。

6.6.1 让错误信息更有用

`fromJValue` 函数的返回类型为 `Either` 。跟 `Maybe` 一样，这个类型是为我们预定义的。我们经常用它来表示可能会失败的计算。

虽然 `Maybe` 也用作这个目的，但它在错误发生时没有给我们足够有用的信息：我们只得到一个 `Nothing` 。虽然 `Either` 类型的结构相同，但是不同于 `Nothing` (相对于 `Maybe`)，“坏事情发生” 构造器命名为 `Left` ，并且其还接受一个参数。

```haskell
-- file: ch06/DataEither.hs
data Maybe a = Nothing
             | Just a
               deriving (Eq, Ord, Read, Show)

data Either a b = Left a
                | Right b
                  deriving (Eq, Ord, Read, Show)
```

我们经常使用 String 作为 a 参数值的类型，所以在出错时我们能提供有用的描述。为了说明在实际中怎么使用 Either 类型，我们来看一个简单的类型类的实例。

```haskell
-- file: ch06/JSONClass.hs
instance JSON Bool where
    toJValue = JBool
    fromJValue (JBool b) = Right b
    fromJValue _ = Left "not a JSON boolean"
```

[译注：读者若想在 **ghci** 中尝试 fromJValue ，需要为其提供类型标注，例如 (fromJValue(toJValue True))::Either JSONError Bool 。]

6.6.2 使用类型别名创建实例

Haskell 98 标准不允许我们用下面的形式声明实例，尽管它看起来没什么问题：

```haskell
-- file: ch06/JSONClass.hs
instance JSON String where
    toJValue               = JString

    fromJValue (JString s) = Right s
    fromJValue _           = Left "not a JSON string"
```

回忆一下，String 是 [Char] 的别名。因此它的类型是 [a] ，并用 Char 替换了类型变量 a 。根据 Haskell 98 的规则，我们在声明实例的时候不允许提供一个类型替代类型变量。也就是说，我们可以给 [a] 声明实例，但给 [Char] 不行。

尽管 GHC 默认遵守 Haskell 98 标准，但是我们可以在文件顶部添加特殊格式的注释来解除这个限制。

```haskell
-- file: ch06/JSONClass.hs
{-# LANGUAGE TypeSynonymInstances #-}
```

这条注释是一条编译器指令，称为*编译选项*（*pragma*），它告诉编译器允许这项语言扩展。上面的代码因为 TypeSynonymInstances（“同义类型的实例”）这项语言扩展而合法。我们在本章（本书）还会碰到更多的语言扩展。

[译注：作者举的这个例子实际上牵涉到了两个问题。第一，Haskell 98 不允许类型别名，这个问题可以通过上述方法解决。第二，Haskell 98 不允许 `[Char]` 这种形式的类型，这个问题需要通过增加另外一条编译选项 `{-# LANGUAGE FlexibleInstances #-}` 来解决。]

[sancao2 译注，若没有 `{-# LANGUAGE FlexibleInstances #-}` 这条编译选项，就会产生下面的结果。其实编译器的 `fix` 提示给大家了。

```
Prelude> :l JSONClass.hs  ../ch05/SimpleJSON.hs
[1 of 2] Compiling SimpleJSON       ( ../ch05/SimpleJSON.hs, interpreted )
[2 of 2] Compiling Main             ( JSONClass.hs, interpreted )

JSONClass.hs:16:10:
   Illegal instance declaration for `JSON String'
      (All instance types must be of the form (T a1 ... an)
      where a1 ... an are *distinct type variables*,
      and each type variable appears at most once in the instance head.
      Use -XFlexibleInstances if you want to disable this.)
   In the instance declaration for `JSON String'
Failed, modules loaded: SimpleJSON.
```

]

[Forec 译注：在 Haskell 8.0.1 中，即使不添加 `{-# LANGUAGE TypeSynonymInstances #-}` 也不会出现问题，但 `{-# LANGUAGE FlexibleInstances #-}` 这条编译选项仍然需要。]

6.7 生活在开放世界

Haskell 的有意地设计成允许我们任意创建类型类的实例，每当我们认为合适时。

```
-- file: ch06/JSONClass.hs
doubleToJValue :: (Double -> a) -> JValue -> Either JSONError a
doubleToJValue f (JNumber v) = Right (f v)
doubleToJValue _ _ = Left "not a JSON number"

instance JSON Int where
    toJValue = JNumber . realToFrac
    fromJValue = doubleToJValue round

instance JSON Integer where
    toJValue = JNumber . realToFrac
    fromJValue = doubleToJValue round

instance JSON Double where
```

(continues on next page)

(continued from previous page)

```
    toJValue = JNumber
    fromJValue = doubleToJValue id
```

我们可以在任意地方添加新实例，而不仅限于在定义了类型类的模块中。类型类系统的这个特性被称为*开放世界假设*（open world assumption）。如果我们有方法表示“这个类型类只存在这些实例”，那我们将得到一个*封闭的世界*。

我们希望把列表 (list) 转为 JSON 数组 (array)。我们现在还不用关心实现细节，所以让我们暂时使用 `undefined` 作为函数内容。

```
-- file: ch06/BrokenClass.hs
instance (JSON a) => JSON [a] where
    toJValue = undefined
    fromJValue = undefined
```

我们也希望能将键/值对列表转为 JSON 对象。

```
-- file: ch06/BrokenClass.hs
instance (JSON a) => JSON [(String, a)] where
    toJValue = undefined
    fromJValue = undefined
```

6.7.1 什么时候重叠实例（Overlapping instances）会出问题？

如果我们把这些定义放进文件中并在 **ghci** 里载入，初看起来没什么问题。

```
*JSONClass> :l BrokenClass.hs
[1 of 2] Compiling JSONClass        ( JSONClass.hs, interpreted )
[2 of 2] Compiling BrokenClass      ( BrokenClass.hs, interpreted )
Ok, modules loaded: JSONClass, BrokenClass
```

然而，一旦我们使用序对列表实例时，我们就”跑”(不是 get，体会一下) 进麻烦里面了 (run in trouble)。

```
*BrokenClass> toJValue [("foo","bar")]

<interactive>:10:1:
    Overlapping instances for JSON [([Char], [Char])]
        arising from a use of ‘toJValue’
    Matching instances:
        instance JSON a => JSON [(String, a)]
            -- Defined at BrokenClass.hs:13:10
        instance JSON a => JSON [a] -- Defined at BrokenClass.hs:8:10
```

(continues on next page)

(continued from previous page)

```
    In the expression: toJValue [("foo", "bar")]
    In an equation for ‘it’ : it = toJValue [("foo", "bar")]
```

[sancao2 译注: 上面的抱怨说的是匹配了两个实例，编译器不知道选择哪一个。`Matching instances: instance xxx, instance xxx` 。]

重叠实例问题是由 Haskell 的”开放世界假设”的一个后果 (a consequence)。以下这个例子可以把问题展现得更清楚一些。

```
-- file: ch06/Overlap.hs
{-# LANGUAGE FlexibleInstances #-}
class Borked a where
    bork :: a -> String

instance Borked Int where
    bork = show

instance Borked (Int, Int) where
    bork (a, b) = bork a ++ ", " ++ bork b

instance (Borked a, Borked b) => Borked (a, b) where
    bork (a, b) = ">>" ++ bork a ++ " " ++ bork b ++ "<<"
```

我们有两个 `Borked` 类型类实例应用于序对 (for pairs)：一个是 `Int` 序对，另一个是任意类型的序对，只要这个类型是 `Borked` 类型类的实例。

假设我们想把 `bork` 应用于 `Int` 序对。为了这样做，编译器必须选择一个实例来用。因为这些实例都是正确地紧挨着 (right next to each other)，所以它似乎可以选择更相关的 (specific) 的实例。

但是，`GHC` 在默认情况下是保守的，且坚持 (insist) 只有一个可能的 GHC 能使用的实例。因此如果我们尝试使用 `bork` 的话，那么它将报错。

Note: 什么时候重叠实例要紧 (matter)?

就像我们之前提到的，我们可以分散一个类型类的实例横跨于 (across) 几个模块中。GHC 不会抱怨重叠实例的单单存在 (mere existence)。取而代之地，他会抱怨，只有当我们试图使用受影响的类型类的函数时，只有他被迫要去做决定采用哪个实例时。

6.7.2 放松 (relex) 类型类的一些限制

通常，我们不能写一个类型类实例，(仅) 为了一个多态类型 (polymorphic type) 的特化版本 (specialized version)。`[Char]` 类型就是多态类型 `[a]` (其中的 `a`) 特化成类型 `Char` 。我们就这样被禁止声明 `[Char]` 为某个类型

类的实例。这”高度地”(highly) 不方便，因为字符串无处不在于实际的代码中。

TypeSynonymInstances (“同义类型的实例”) 语言扩展取消了这个限制，并允许我们写这样的实例。

GHC 支持另外一个有用的语言扩展，OverlappingInstances （覆盖实例）。它解决 (原文为 address) 了在处理重叠实例时候我们碰到的问题。如果存在多个重叠的实例去从中选择，这个扩展会”采摘”(pick) 最相关的（specific）那一个。

[Forec 译注：在 Haskell 8.0 后，OverlappingInstances 已被抛弃，可替代的方法是在实例中加上 {-# OVERLAPPABLE #-}，如：

```
instance {-# OVERLAPPABLE -#} Foo a => Foo [a] where
    foo = concat . intersperse ", " . map foo
```

]

我们经常使用这个扩展，同 TypeSynonymInstances 一起。这里是一个例子。

```
-- file: ch06/SimpleClass.hs
{-# LANGUAGE TypeSynonymInstances, OverlappingInstances, FlexibleInstances #-}

import Data.List

class Foo a where
    foo :: a -> String

instance Foo a => Foo [a] where
    foo = concat . intersperse ", " . map foo

instance Foo Char where
    foo c = [c]

instance Foo String where
    foo = id
```

如果我们应用 (apply) foo 于 String ，编译器会选择 String 相关的 (specific) 实现。虽然我们有一个 Foo 的实例关于 [a] 和 Char ，但关于 String 的实例更相关，所以 GHC 选择它。

即使 OverlappingInstances (覆盖实例) 扩展出于使能状态 (enabled)，GHC 仍将拒绝代码，若他找到一个以上等价地相关的（equally specific）实例。

Note: 何时去使用 OverlappingInstances 扩展?

这是一个重要的点：GHC 认为 OverlappingInstances 会影响一个实例的声明，而不是一个位置，于此(位置)我们使用一个实例。换句话说，当我们定义一个实例，其(这个实例)我们希望能(被)允许覆盖(overlap)于其他实例的时候，我们必须使能 (enable) 该扩展 (OverlappingInstances) 为这个模块，而其（这个模

块）包含着定义。当他编译这个模块的时候，GHC 会记录那个实例为 " 能被覆盖（overlap）以其他的模块 " 的。一旦我们引入 (import) 这个模块而使用他的实例，我们将不需要使能 (enable) `OverlappingInstances` 编译选项在引入模块的时候：GHC 将已经知道这个实例是被标记为 " 对覆盖友好的 " (okay to overlap)，当他被定义的时候。这种行为是很有用的，当我们在写一个库 (library) 的时候: 我们能选择去创造可覆盖的 (overlappable) 实例，但是库的用户不必须使能 (enable) 任何特殊的语言扩展。

6.7.3 show 是如何处理 String 的?

`OverlappingInstances` (覆盖实例) 和 `TypeSynonymInstances` (“同义类型的实例”) 语言扩展是特定于 GHC 的，而在定义上过去没有出现 (present) 于 “Haskell 98”。然而，大家熟悉的 `Show` 类型类，来自 “Haskell 98”，以某种方法区别地” 渲染” (render) `Char` 列表 (list) 和 `Int` 列表。它达成这个 (” 区别地渲染”) 通过一个聪明但简单的把戏 (trick)。

`Show` 类型类定义了两个方法：一个 `show` 方法，用于渲染单值 (one value) 和一个 `showList` 方法，用于渲染值的列表。而 `showList` 的默认实现，渲染一个列表，以使用中括号们和逗号们的方式。

`Show` 的实例对于 `[a]` 是使用 `showList` 实现的。`Show` 的实例为 `[Char]` 提供一个特殊的 `showList` 实现。其（该实现）使用双引号，并转义” 非 ASCII 可打印” (non-ASCII-printable) 的字符们。

[sancao2 译注: 上面那句 `[Char]` 原文没有 `[]`，应该是错了。]

作为结果，如果有人对 `[Char]` 应用 `show` 函数，那么 `showList` 的实现会被选上，并且将会正确地渲染字符串，通过使用括号们。

至少有时，因而，我们就能克制对 `OverlappingInstances` (覆盖实例) 扩展的需要，带着一点点 (时间维度的) 横向思维 (lateral thinking)。

6.8 如何给类型以新身份（new identity）

包括熟悉的 `data` 关键字以外，Haskell 提供我们另外一种方式来创建新类型，即采用 `newtype` 关键字。

```
-- file: ch06/Newtype.hs
data DataInt = D Int
    deriving (Eq, Ord, Show)

newtype NewtypeInt = N Int
    deriving (Eq, Ord, Show)
```

`newtype` 声明的目的是重命名一个存在着的类型，来给它一个独特的身份 (id)。像我们能看到的，它的用法和采用 `data` 关键字进行声明，在表面上很相似。

Note: type 和 newtype 关键字

尽管他们的名字是类似的，type 和 newtype 关键字有不同的目的。type 关键字给了我们另一种方式以引用 (refer to) 某个类型，就像昵称之于一个朋友。我们和编译器都知道 [Char] 和 String 引用的是同一个类型。

比较起来 (与 type)，newtype 关键字存在，以隐藏一个类型的本性 (nature)。考虑一个 UniqueID 类型。

```
-- file: ch06/Newtype.hs
newtype UniqueID = UniqueID Int
    deriving (Eq)
```

编译器会视 UniqueID 为一个不同的类型于 Int 。作为一个 UniqueID 的用户，我们只知道它有一个”唯一标识符”(Unique ID，英语字面意思)；我们并不知道它被实现为一个 Int 。

当我们声明一个 newtype 时，我们必须选择哪个潜在类型的类型类实例，而对其（该实例）我们想要暴露。在这里，我们决定让 NewtypeInt 提供 Int 的 Eq 、Ord 和 Show 实例。作为一个结果，我们可以比较和打印 NewtypeInt 类型的值。

```
*Main> N 1 < N 2
True
```

由于我们没有暴露 Int 的 Num 或 Integral 实例，NewtypeInt 类型的值并不是数字们。例如，我们不能加他们。

```
*Main> N 313 + N 37

<interactive>:9:7:
    No instance for (Num NewtypeInt) arising from a use of ‘+’
    In the expression: N 313 + N 37
    In an equation for ‘it’ : it = N 313 + N 37
```

跟用 data 关键字一样，我们可以用 newtype 的值构造器创建一个新值，或者模式匹配于存在的值。

如果 newtype 没用自动派生（deriving）来暴露一个类型类的潜在（underlying）类型实现的话，我们是自由的，或者去写一个新实例，或者干脆留那个类型类处于不实现状态。

6.8.1 data 和 newtype 声明之间的区别

newtype 关键字存在着（exists）为了给现有类型以一个新的身份（id）。它有更多的限制于其使用上，比起 data 关键字。说白了，newtype 只能有一个值构造器，并且那个构造器须恰有一个字段 (field)。

```
-- file: ch06/NewtypeDiff.hs
-- 可以：任意数量的构造器和字段（这里的两个 Int 为两个字段 (fields)）
data TwoFields = TwoFields Int Int
```

(continues on next page)

(continued from previous page)

```
-- 可以：恰一个字段
newtype Okay = ExactlyOne Int

-- 可以：类型变量是没问题的
newtype Param a b = Param (Either a b)

-- 可以：记录语法是友好的
newtype Record = Record {
        getInt :: Int
    }

-- 不可以：没有字段
newtype TooFew = TooFew

-- 不可以：多于一个字段
newtype TooManyFields = Fields Int Int

-- 不可以：多于一个构造器
newtype TooManyCtors = Bad Int
                     | Worse Int
```

在此之上，还有另一个重要的区别于 data 和 newtype 之间。一个类型，由 data 关键字创建，有一个簿记保持（book-keeping）的开销在运行时。例如，追踪（track）某个值是由哪个值构造器创造的。而另一方面，newtype 只能有一个构造器，所以不需要这个额外开销。这使得它在运行时更省时间和空间。

因为 newtype 的构造器只在编译时使用，运行时甚至不存在，所以对于用 newtype 定义的类型和那些用 data 定义的类型来说，类型匹配在 undefined 上的表现不同。

为了理解这个不同点，让我们首先回顾一下，我们可能期望一个普通类型的什么行为。我们已经非常熟悉，如果在运行时 undefined 被求值会导致崩溃。

```
Prelude> undefined
*** Exception: Prelude.undefined
```

这里有一个类型匹配，在其（类型匹配）中我们采用“D 构造器”构造一个 DataInt ，然后放 undefined 在内部。

```
*Main> case (D undefined) of D _ -> 1
1
```

[sancao2 译注: 做这个实验要先加载“Newtype.hs”，其中定义了 D。]

由于我们的模式匹配只对构造器而不检查载荷 (payload)，undefined 保持未被求值状态，因而不会导致一个异常被抛出。

在这个例子中，我们没有同时使用 D 构造器，因而未被保护的 undefined 会被求值。当模式匹配发生时，我们抛出异常。

```
*Main> case undefined of D _ -> 1
*** Exception: Prelude.undefined
```

当我们使用 N 构造器以得到 NewtypeInt 值时，我们看到相同的行为：没有异常，就像使用 DataInt 类型的 D 构造器。

```
*Main> case (N undefined) of N _ -> 1
1
```

决定性的（crucial）差异发生了，当我们从表达式中去掉 N ，并匹配于一个未保护的 undefined 时。

```
*Main> case undefined of N _ -> 1
1
```

我们没有崩溃！由于不存在构造器于运行时，对 N _ 的匹配实际上等效于对空白通配符 _ 的匹配：由于这个通配符（_）总可以匹配，所以表达式不需要被求值。

Note: 关于 newtype 构造器的另一种看法

虽然，我们使用值（value）构造器，以得到一个 newtype ，其方式等同于一个类型被定义而其采用 data 关键词。两者所做的是强迫一个值（value）处于（between）他的“正常”（normal）类型和他的 newtype 类型之间。

换句话说，当我们应用（apply）N 于一个表达式，我们强迫一个表达式从 Int 类型到 NewtypeInt 类型，对我们（we）和编译器（compiler）而言，但是，完全地（absolutely），没有事情发生于运行时（runtime）。

类似地，当我们匹配 N 构造器于一个模式中，我们强制一个表达式从 NewtypeInt 到 Int ，但是再次地不存在开销于运行时。

6.8.2 总结：三种命名类型的方式

这是一份简要重述（recap），关于 Haskell 的三种方式用来为类型提出（introduce）新名。

- data 关键字提出（introduce）一个真正的代数（albegraic）数据类型。
- type 关键字给我们一个别名（synonym）去用，为一个存在着的（existing）类型。我们可以交换地（interchangeably）使用这个类型和他的别名，
- newtype 关键字给予一个存在着的类型以一个独特的身份（distinct identity）。这个原类型和这个新类型是不可交换的（interchangeable）。

6.9 JSON 类型类, 不带有重叠实例

启用 GHC 的重叠实例支持是一个让我们的 JSON 库工作的既有效又快速的方法。在更复杂的场景中，我们有时被迫面对这样一种情况：某个类型类有多个相关程度相同（equally good）实例。在这种情况下，重叠实例们将不会帮助我，而我们将需要代之以几处 `newtype` 声明。为了弄明白这涉及到了什么，让我们重构（rework）我们的 JSON 类型类实例们以使用 `newtype` 代替重叠实例。

我们的第一个任务，是帮助编译器区分 `[a]` 和 `[(String,[a])]` 。前者（`[a]` ）我们用来表示 JSON 数组们（arrays），而后者（`[(String, [a])]` ）用来表示 JSON 对象们（objects）。他们是这些类型们，其给我们制造了麻烦于我们学会 `OverlappingInstances` （覆盖实例）之前。我们包装了（wrap up）列表（list）类型，以至于编译器不会视其为一个列表。

```
-- file: ch06/JSONClass.hs
newtype JAry a = JAry {
      fromJAry :: [a]
      } deriving (Eq, Ord, Show)
```

当我们从自己的模块导出这个类型时，我们会导出该类型完整的细节。我们的模块头部将看起来像这样：

```
-- file: ch06/JSONClassExport.hs
module JSONClass
    (
      JAry(..)
    ) where
```

紧跟着 `Jary` 的”`(..)` “，意思是“导出这个类型的所有细节”。

Note: 一点稍微的偏差，相比于正常使用

通常地，当我们导出一个 `newtype` 的时候，我们 不会导出这个类型的数据构造器，为了保持其细节的抽象（abstract）。取而代之，我们会定义一个函数为我们应用（apply）该数据构造器。

```
-- file: ch06/JSONClass.hs
jary :: [a] -> JAry a
jary = JAry
```

于是，我们会导出类型构造器、解构函数和我们的构造函数，除了数据构造器。

```
-- file: ch06/JSONClassExport.hs
module JSONClass
    (
      JAry(fromJAry)
    , jary
    ) where
```

当我们没有导出一个类型的数据构造器，我们库的顾客们就只能使用我们提供的函数们去构造和解构该类型的值。这个特性为我们，这些库作者们，提供了自由去改变类型的内部表示形式（represention），如果我们需要去（这么做）。

如果我们导出数据构造器，顾客们很可能开始依赖于它，比方说使用它（数据构造器）在一些模式中。如果哪天我们希望去修改这个类型的内部构造，我们将冒险打破任意代码，而其（这些代码）使用着该数据构造器。

在我们这里的情况下，我们得不到什么额外的好处，通过让数组的包装器保持抽象，所以我们就干脆地导出该类型的整个定义。

我们提供另一个包装类型，而其隐藏了一个 JSON 对象的我们的表示形式（represention）。

```
-- file: ch06/JSONClass.hs
newtype JObj a = JObj {
      fromJObj :: [(String, a)]
    } deriving (Eq, Ord, Show)
```

带着这些定义好的类型，我们制造一些小改动到我们的 `JValue` 类型的定义。

```
-- file: ch06/JSONClass.hs
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject (JObj JValue)    -- was [(String, JValue)]
            | JArray (JAry JValue)     -- was [JValue]
              deriving (Eq, Ord, Show)
```

这个改动不会影响到 `JSON` 类型类的实例们，而那些我们已经写完。但是我们还要为我们新的 `JAry` 和 `JObj` 类型编写实例。

```
-- file: ch06/JSONClass.hs
jaryFromJValue :: (JSON a) => JValue -> Either JSONError (JAry a)

jaryToJValue :: (JSON a) => JAry a -> JValue

instance (JSON a) => JSON (JAry a) where
    toJValue = jaryToJValue
    fromJValue = jaryFromJValue
```

让我们缓慢地走过各个步骤，而这些步骤会转换一个 `JAry a` 到一个 `JValue` 。给定一个列表，其中内部每一个元素都是一个 `JSON` 实例，转换它（前面的列表）到一个 `JValue` s 组成的列表是简单的。

```
-- file: ch06/JSONClass.hs
listToJValues :: (JSON a) => [a] -> [JValue]
listToJValues = map toJValue
```

取得这个值并包装他来得到一个 JAry JValue 的过程，实际上就是对其应用 newtype 的类型构造器。

```
-- file: ch06/JSONClass.hs
jvaluesToJAry :: [JValue] -> JAry JValue
jvaluesToJAry = JAry
```

(记住，这种做法没有任何性能代价。我们只是告诉编译器隐藏这个事实：我们正在使用一个列表。) 为了转化这个值成为一个 JValue ，我们应用另一个类型构造器。

:: – file: ch06/JSONClass.hs jaryOfJValuesToJValue :: JAry JValue -> JValue jaryOfJValuesToJValue = JArray

组装这些代码片段，通过使用函数组合 (function composition)，而我们得到一个简洁的单行（代码），用于转换得到一个 JValue 。

```
-- file: ch06/JSONClass.hs
jaryToJValue = JArray . JAry . map toJValue . fromJAry
```

我们有更多的工作去做来实现从 JValue 到 JAry a 的转换，但是我们把它“碎裂”（break）成一些可重用的部分。基本函数一目了然（straightforward）。

```
-- file: ch06/JSONClass.hs
jaryFromJValue (JArray (JAry a)) =
    whenRight JAry (mapEithers fromJValue a)
jaryFromJValue _ = Left "not a JSON array"
```

whenRight 函数会检查传给它的参数：如果第二个参数是用 Right 构造器创建的，以它为参数调用第一个参数指定的函数；如果第二个参数是 Left 构造器创建的，则将它保持原状返回，其它什么也不做。

```
-- file: ch06/JSONClass.hs
whenRight :: (b -> c) -> Either a b -> Either a c
whenRight _ (Left err) = Left err
whenRight f (Right a) = Right (f a)
```

mapEithers 函数要更复杂一些。它的行为就像 map 函数，但如果它遇到一个 Left 值，会直接返回该值，而不会继续积累 Right 值构成的列表。

```
-- file: ch06/JSONClass.hs
mapEithers :: (a -> Either b c) -> [a] -> Either b [c]
mapEithers f (x:xs) = case mapEithers f xs of
                        Left err -> Left err
                        Right ys -> case f x of
```

(continues on next page)

(continued from previous page)

```
                                    Left err -> Left err
                                    Right y -> Right (y:ys)
mapEithers _ _ = Right []
```

由于隐藏在 JObj 类型中的列表元素有更细碎的结构，相应的，在它和 JValue 类型之间互相转换的代码就会有点复杂。万幸的是，我们可以重用刚刚定义过的函数。

```
-- file: ch06/JSONClass.hs
import Control.Arrow (second)

instance (JSON a) => JSON (JObj a) where
    toJValue = JObject . JObj . map (second toJValue) . fromJObj

    fromJValue (JObject (JObj o)) = whenRight JObj (mapEithers unwrap o)
      where unwrap (k,v) = whenRight ((,) k) (fromJValue v)
    fromJValue _ = Left "not a JSON object"
```

6.9.1 练习题

1. 在 **ghci** 中加载 Control.Arrow 模块，弄清 second 函数的功能。
2. (,) 是什么类型？在 **ghci** 中调用它时，它的行为是什么？(,,) 呢？

6.10 可怕的单一同态限定（monomorphism restriction）

Haskell 98 有一个微妙的特性可能会在某些意想不到的情况下“咬”到我们。下面这个简单的函数展示了这个问题。

```
-- file: ch06/Monomorphism.hs
myShow = show
```

如果我们试图把它载入 **ghci**，会产生一个奇怪的错误：

```
Prelude> :l Monomorphism.hs

[1 of 1] Compiling Main             ( Monomorphism.hs, interpreted )

Monomorphism.hs:2:10:
    No instance for (Show a0) arising from a use of ‘show’
    The type variable ‘a0’ is ambiguous
    Relevant bindings include
```

(continues on next page)

(continued from previous page)

```
        myShow :: a0 -> String (bound at Monomorphism.hs:2:1)
    Note: there are several potential instances:
        instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
        instance Show Ordering -- Defined in ‘GHC.Show’
        instance Show Integer -- Defined in ‘GHC.Show’
        ...plus 22 others
    In the expression: show
    In an equation for ‘myShow’ : myShow = show
    Failed, modules loaded: none.
```

错误信息中提到的“monomorphism restriction”是 Haskell 98 的一部分。单一同态是多态（polymorphism）的反义词：它表明某个表达式只有一种类型。Haskell 有时会强制使某些声明不像我们预想的那么多态。

我们在这里提单一同态是因为尽管它和类型类没有直接关系，但类型类给它提供了产生的环境。

Tip: 在实际代码中可能很久都不会碰到单一同态，因此我们觉得你没必要记住这部分的细节，只要在心里知道有这么回事就可以了，除非 GHC 真的报告了跟上面类似的错误。如果真的发生了，记得在这儿曾读过这个错误，然后回过头来看就行了。

我们不会试图去解释单一同态限制。Haskell 社区一致同意它并不经常出现；它解释起来很棘手（tricky）；它几乎没什么实际用处；它唯一的作用就是坑人。举个例子来说明它为什么棘手：尽管上面的例子违反了这个限制，下面的两个编译起来却毫无问题。

```
-- file: ch06/Monomorphism.hs
myShow2 value = show value

myShow3 :: (Show a) => a -> String
myShow3 = show
```

上面的定义表明，如果 GHC 报告单一同态限制错误，我们有三个简单的方法来处理。

- 显式声明函数参数，而不是隐性。
- 显式定义类型签名，而不是依靠编译器去推导。
- 不改代码，编译模块的时候用上 `NoMonomorphismRestriction` 语言扩展。它取消了单一同态限制。

没人喜欢单一同态限制，因此几乎可以肯定的是下一个版本的 Haskell 会去掉它。但这并不是说加上 `NoMonomorphismRestriction` 就可以一劳永逸：有些编译器（包括一些老版本的 GHC）识别不了这个扩展，但用另外两种方法就可以解决问题。如果这种可移植性对你不是问题，那么请务必打开这个扩展。

6.11 结论

在这章，你学到了类型类有什么用以及怎么用它们。我们讨论了如何定义自己的类型类，然后又讨论了一些 Haskell 库里定义的类型类。最后，我们展示了怎么让 Haskell 编译器给你的类型自动派生出某些类型类实例。

CHAPTER

SEVEN

第 7 章：I/O

就算不是全部，绝大多数的程序员显然还是致力于从外界收集数据，处理这些数据，然后把结果传回外界。也就是说，关键就是输入输出。

Haskell 的 I/O 系统是很强大和富有表现力的。它易于使用，也很有必要去理解。Haskell 严格地把纯代码从那些会让外部世界发生事情的代码中分隔开。就是说，它给纯代码提供了完全的副作用隔离。除了帮助程序员推断他们自己代码的正确性，它还使编译器可以自动采取优化和并行化成为可能。

我们将用简单标准的 I/O 来开始这一章。然后我们要讨论下一些更强大的选项，以及提供更多 I/O 是怎么适应纯的，惰性的，函数式的 Haskell 世界的细节。

7.1 Haskell 经典 I/O

让我们开始使用 Haskell 的 I/O 吧。先来看一个程序，它看起来很像在 C 或者 Perl 等其他语言的 I/O。

```
-- file: ch07/basicio.hs
main = do
    putStrLn "Greetings!  What is your name?"
    inpStr <- getLine
    putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

你可以编译这个程序，变成一个单独的可执行文件，然后用 **runghc** 运行它，或者从 **ghci** 调用 `main` 。这里有一个使用 **runghc** 的例子：

```
$ runghc basicio.hs
Greetings!  What is your name?
John
Welcome to Haskell, John!
```

这是一个相当简单，明显的结果。你可以看到 `putStrLn` 输出一个 `string` ，后面跟了一个换行符。

`getLine` 从标准输入读取一行。<- 语法对于你可能比较新。

简单来看，它绑定一个 I/O 动作的结果到一个名字。我们用简单的列表串联运算符 ++ 来联合输入字符串和我们自己的文本。

让我们来看一下 `putStrLn` 和 `getLine` 的类型。你可以在库参考手册里看到这些信息，或者直接问 `ghci`：

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

注意，这些类型在他们的返回值里面都有 IO。现在关键的是，你要从这里知道他们可能有副作用，或者他们用相同的参数调用可能返回不同的值，或者两者都有。

> `putStrLn` 的类型看起来像一个函数，它接受一个 `String` 类型的参数，并返回 `IO ()` 类型的值。

可是 `IO ()` 是什么呢?

`IO something` 类型的所有东西都是一个 IO 动作，你可以保存它但是什么都不会发生。我可以说 `writefoo = putStrLn "foo"` 并且现在什么都不发生。但是如果我过一会在另一个 I/O 动作中间使用 `writefoo`，`writefoo` 动作将会在它的父动作被执行的时候执行 – I/O 动作可以粘合在一起来形成更大的 I/O 动作。

> `()` 是一个空的元组（读作 “unit”），表明从 `putStrLn` 没有返回值。

这和 Java 或 C 里面的 `void` 类似。

Tip: I/O 动作可以被创建，赋值和传递到任何地方，但是它们只能在另一个 I/O 动作里面被执行。

我们在 **ghci** 下看下这句代码：

```
ghci> let writefoo = putStrLn "foo"
ghci> writefoo
foo
```

在这个例子中，输出 `foo` 不是 `putStrLn` 的返回值，而是它的副作用，把 `foo` 写到终端上。

还有另一件事要注意，实际上是 **ghci** 执行的 `writefoo`。意思是，如果给 **ghci** 一个 I/O 动作，它将会在那个地方帮你执行它。

Note: 什么是 I/O 动作？* 类型是 `IO t` * 是 Haskell 的头等值，并且和 Haskell 的类型系统无缝结合。* 在运行（perform）的时候产生作用，而不是在估值（evaluate）的时候。* 任何表达式都会产生一个动作作为它的值，但是这个动作直到在另一个 I/O 动作里面被执行的时候才会运行。* 运行（执行）一个 `IO t` 类型的动作可能运行 I/O，并且最终交付一个类型 `t` 的结果。

---

getLine 的类型可能看起来比较陌生。它看起来像一个值，而不像一个函数。但实际上，有一种看它的方法：getLine 保存了一个 I/O 动作。当这个动作运行了你会得到一个 String 。

<- 运算符是用来从运行 I/O 动作中抽出结果，并且保存到一个变量中。

main 自己就是一个 I/O 动作，类型是 IO () 。你可以在其他 I/O 动作中只是运行 I/O 动作。Haskell 程序中的所有 I/O 动作都是由从 main 的顶部开始驱动的，main 是每一个 Haskell 程序开始执行的地方。然后，要说的是给 Haskell 中副作用提供隔离的机制是：你在 I/O 动作中运行 I/O，并且在那儿调用纯的（非 I/O）函数。大部分 Haskell 代码是纯的，I/O 动作运行 I/O 并且调用纯 (pure) 代码。

do 是用来定义一串动作的方便方法。你马上就会看到，还有其他方法可以用来定义。当你用这种方式来使用 do 的时候，缩进很重要，确保你的动作正确地对齐了。

只有当你有多于一个动作需要运行的时候才要用到 do 。 do 代码块的值是最后一个动作执行的结果。

想要看 do 语法的完整介绍，可以看**'do 代码块提取 '_** .

我们来考虑一个在 I/O 动作中调用纯代码的一个例子：

```
-- file: ch07/callingpure.hs
name2reply :: String -> String
name2reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ charcount ++ " characters."
    where charcount = show (length name)

main :: IO ()
main = do
       putStrLn "Greetings once again.  What is your name?"
       inpStr <- getLine
       let outStr = name2reply inpStr
       putStrLn outStr
```

注意例子中的 name2replay 函数。这是一个 Haskell 的一个常规函数，它遵守所有我们告诉过你的规则：给它相同的输入，它总是返回相同的结果，没有副作用，并且以惰性方式运行。它用了其他 Haskell 函数：(++) ，show 和 length 。

往下看到 main ，我们绑定 name2replay inpStr 的结果到 outStr 。当你在用 do 代码块的时候，你用 <- 去得到 I/O 动作的结果，用 let 得到纯代码的结果。

当你在 do 代码块中使用 let 声明的时候，不要在后面放上 in 。

你可以看到这里是怎么从键盘读取这人的名字的。然后，数据被传到一个纯函数，接着它的结果被打印出来。实际上，main 的最后两行可以被替换成 putStrLn (name2reply inpStr) 。所以，main 有副作用（比如它在终端上显示东西），name2replay 没有副作用，也不能有副作用。因为 name2replay 是一个纯函数，不是一个动作。

我们在 **ghci** 上检查一下：

```
ghci> :load callingpure.hs
[1 of 1] Compiling Main             ( callingpure.hs, interpreted )
Ok, modules loaded: Main.
ghci> name2reply "John"
"Pleased to meet you, John.\nYour name contains 4 characters."
ghci> putStrLn (name2reply "John")
Pleased to meet you, John.
Your name contains 4 characters.
```

字符串里面的 `\n` 是换行符，它让终端在输出中开始新的一行。在 **ghci** 直接调用 `name2replay "John"` 会字面上显示 `\n` ，因为使用 `show` 来显示返回值。但是使用 `putStrLn` 来发送到终端的话，终端会把 `\n` 解释成开始新的一行。

如果你就在 **ghci** 提示符那打上 `main` ，你觉得会发生什么？来试一下吧。

看完这几个例子程序之后，你可能会好奇 Haskell 是不是真正的命令式语言呢，而不是纯的，惰性的，函数式的。这些例子里的一些看起来是按照顺序的一连串的操作。这里面还有很多东西，我们会在这一章的*Haskell 实际上是命令式的吗？* 和*惰性 I/O* 章节来讨论这个问题。

7.1.1 Pure vs. I/O

这里有一个比较的表格，用来帮助理解纯代码和 I/O 之间的区别。 当我们说起纯代码的时候，我们是在说 Haskell 函数在输入相同的时候总是返回相同结果，并且没有副作用。

在 Haskell 里面只有 I/O 动作的执行违反这些规则。

表格 7.1. Pure vs. Impure

Pure	Impure
输入相同时总是产生相同结果	相同的参数可能产生不同的结果
从不会有副作用	可能有副作用
从不修改状态	可能修改程序、系统或者世界的全局状态

7.1.2 为什么纯不纯很重要？

在这一节中，我们已经讨论了 Haskell 是怎么在纯代码和 I/O 动作之间做了很明确的区分。很多语言没有这种区分。在 C 或者 Java 这样的语言中，编译器不能保证一个函数对于同样的参数总是返回同样的结果，或者保证函数没有副作用。要知道一个函数有没有副作用只有一个办法，就是去读它的文档，并且希望文档说的准确。

程序中的很多错误都是由意料之外的副作用造成的。函数在某些情况下对于相同参数可能返回不同的结果，还有更多错误是由于误解了这些情况而造成的。

多线程和其他形式的并行化变得越来越普遍，管理全局副作用变得越来越困难。

Haskell 隔离副作用到 I/O 动作中的方法提供了一个明确的界限。你总是可以知道系统中的那一部分可能修改状态哪一部分不会。你总是可以确定程序中纯的部分不会有意想不到的结果。这样就帮助你思考程序，也帮助编译器思考程序。比如最新版本的 **ghc** 可以自动给你代码纯的部分提供一定程度的并行化 – 一个计算的神圣目标。

对于这个主题，你可以在*惰性 I/O 的副作用* 一节看更多的讨论。

7.2 使用文件和句柄（Handle）

到目前为止，我们已经看了在计算机的终端里怎么和用户交互。当然，你经常会需要去操作某个特定文件，这个也很简单。

Haskell 位 I/O 定义了一些基本函数，其中很多和你在其他语言里面见到的类似。 `System.IO` 的参考手册为这些函数提供了很好的概要。

你会用到这里面某个我们在这里没有提及的某个函数。

通常开始的时候你会用到 `openFile` ，这个函数给你一个文件句柄，这个句柄用来对这个文件做特定的操作。Haskell 提供了像 `hPutStrLn` 这样的函数，它用起来和 `putStrLn` 很像，但是多一个参数（句柄），指定操作哪个文件。当操作完成之后，需要用 `hClose` 来关闭这个句柄。这些函数都是定义在 `System.IO` 中的，所以当你操作文件的时候你要引入这个模块。几乎每一个非“h”的函数都有一个对应的“h”函数，比如，`print` 打印到显示器，有一个对应的 `hPrint` 打印到文件。

我们用一种命令式的方式来开始读写文件。这有点像一个其他语言中 `while` 循环，这在 Haskell 中不是最好的方法。接着我们会看几个更加 Haskell 风格的例子。

```
-- file: ch07/toupper-imp.hs
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
    inh <- openFile "input.txt" ReadMode
    outh <- openFile "output.txt" WriteMode
    mainloop inh outh
    hClose inh
    hClose outh

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
    do ineof <- hIsEOF inh
       if ineof
```

(continues on next page)

(continued from previous page)

```
      then return ()
      else do inpStr <- hGetLine inh
              hPutStrLn outh (map toUpper inpStr)
              mainloop inh outh
```

像每一个 Haskell 程序一样，程序在 `main` 那里开始执行。两个文件被打开：`input.txt` 被打开用来读，还有一个 `output.txt` 被打开用来写。然后我们调用 `mainloop` 来处理这个文件。

`mainloop` 开始的时候检查看看我们是否在输入文件的结尾（EOF）。如果不是，我们从输入文件读取一行，把这一行转成大写，再把它写到输出文件。然后我们递归调用 `mainloop` 继续处理这个文件。

注意那个 `return` 调用。这个和 C 或者 Python 中的 `return` 不一样。在那些语言中，`return` 用来立即退出当前函数的执行，并且给调用者返回一个值。在 Haskell 中，`return` 是和 `<-` 相反。也就是说，`return` 接受一个纯的值，把它包装进 IO。因为每个 I/O 动作必须返回某个 `IO` 类型，如果你的结果来自纯的计算，你必须用 `return` 把它包装进 IO。举一个例子，如果 `7` 是一个 `Int`，然后 `return 7` 会创建一个动作，里面保存了一个 `IO Int` 类型的值。在执行的时候，这个动作将会产生结果 `7`。关于 `return` 的更多细节，可以参见*Return 的本色* 一节。

我们来尝试运行这个程序。我们已经有一个像这样的名字叫 `input.txt` 的文件：

```
This is ch08/input.txt

Test Input
I like Haskell
Haskell is great
I/O is fun

123456789
```

现在，你可以执行 `runghc toupper-imp.hs`，你会在你的目录里找到 `output.txt`。它看起来应该是这样：

```
THIS IS CH08/INPUT.TXT

TEST INPUT
I LIKE HASKELL
HASKELL IS GREAT
I/O IS FUN

123456789
```

7.2.1 关于 openFile 的更多信息

我们用 **ghci** 来检查 `openFile` 的类型：

```
ghci> :module System.IO
ghci> :type openFile
openFile :: FilePath -> IOMode -> IO Handle
```

`FilePath` 就是 `String` 的另一个名字。它在 I/O 函数的类型中使用，用来阐明那个参数是用来表示文件名的，而不是其他通常的数据。

`IOMode` 指定文件是怎么被管理的，`IOMode` 的可能值在表格 7.2 中列出来了。

表格 7.2. IOMode 可能值

IOMode	可读	可写	开始位置	备注
ReadMode	是	否	文件开头	文件必须存在
WriteMode	否	是	文件开头	如果存在，文件会被截断（完全清空）
ReadWriteMode	是	是	文件开头	如果不存在会新建文件，如果存在不会损害原来的数据
AppendMode	否	是	文件结尾	如果不存在会新建文件，如果存在不会损害原来的数据

我们在这一章里大多数是操作文本文件，二进制文件同样可以在 Haskell 里使用。如果你在操作一个二进制文件，你要用 `openBinaryFile` 替代 `openFile` 。你当做二进制文件打开，而不是当做文本文件打开的话，像 Windows 这样的操作系统会用不同的方式来处理文件。在 Linux 这类操作系统中，`openFile` 和 `openBinaryFile` 执行相同的操作。不过为了移植性，当你处理二进制数据的时候总是用 `openBinaryFile` 还是明智的。

7.2.2 关闭句柄

你已经看到 `hClose` 用来关闭文件句柄。我们花点时间思考下为什么这个很重要。

就和你将在*缓冲区（Buffering）*一节看到的一样，Haskell 为文件维护内部缓冲区，这提供了一个重要的性能提升。然而，也就是说，直到你在一个打开来写的文件上调用 `hClose` ，你的数据不会被清理出操作系统。

确保 `hClose` 的另一个理由是，打开的文件会占用系统资源。如果你的程序运行很长一段时间，并且打开了很多文件，但是没有关闭他们，你的程序很有可能因为资源耗尽而崩溃。Haskell 在这方面和其他语言没有什么不同。

当一个程序退出的时候，Haskell 通常会小心地关闭所有还打开着的文件。然而在一些情况下 Haskell 可能不会帮你做这些。所以再一次强调，最好在任何时候都由你来负责调用 `hClose` 。

Haskell 给你提供了一些工具，不管出现什么错误，用来简单地确保这些工作。你可以阅读在*扩展例子：函数式 I/O 和临时文件*一节的 `finally` 和**‘获取-使用-回收周期‘_**一节的 `bracket` 。

7.2.3 Seek and Tell

当从一个对应硬盘上某个文件句柄上读写的时候，操作系统维护了一个当前硬盘位置的内部记录。每次你做另一次读的时候，操作系统返回下一个从当前位置开始的数据块，并且增加这个位置，反映出你正在读的数据。

你可以用 `hTell` 来找出你文件中的当前位置。当文件刚新建的时候，文件是空的，这个位置为 0。在你写入 5 个字节之后，位置会变成 5，诸如此类。

> `hTell` 接受一个 `Handle` 并返回一个带有位置的 `IO Integer` 。

`hTell` 的伙伴是 `hSeek` 。 `hSeek` 让你可以改变文件位置，它有 3 个参数：一个 `Handle` ，一个 `seekMode` ，还有一个位置。

`SeekMode` 可以是三个不同值中的一个，这个值指定怎么去解析这个给的位置。 `AbsoluteSeek` 表示这个位置是在文件中的精确位置，这个和 `hTell` 给你的是同样的信息。`RelativeSeek` 表示从当前位置开始寻找，一个正数要求在文件中向前推进，一个负数要求向后倒退。

最后，`SeekFromEnd` 会寻找文件结尾之前特定数目的字节。 `hSeek handle SeekFromEnd 0` 把你带到文件结尾。

举一个 `hSeek` 的例子，参考*扩展例子：函数式 I/O 和临时文件* 一节。

不是所有句柄都是可以定位的。一个句柄通常对应于一个文件，但是它也可以对应其他东西，比如网络连接，磁带机或者终端。你可以用 `hIsSeekable` 去看给定的句柄是不是可定位的。

7.2.4 标准输入，输出和错误

先前我们指出对于每一个非“h”函数通常有一个对应的“h”函数用在句柄上的。实际上，非“h”的函数就是他们的“h”函数的一个快捷方式。

在 `System.IO` 里有 3 个预定义的句柄，这些句柄总是可用的。他们是 `stdin` ，对应标准输入；`stdout` ，对应标准输出；和 `stderr` 对应标准错误。标准输入一般对应键盘，标准输出对应显示器，标准错误一般输出到显示器。

像 `getLine` 的这些函数可以简单地这样定义：

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

Tip: 我们这里使用了局部应用。

如果不明白，可以参考**‘部分函数应用和柯里化 ‘_**

之前我们告诉你这 3 个标准文件句柄一般对应什么。那是因为一些操作系统可以让你重定向这个文件句柄到不同的地方-文件，设备，甚至是其他程序。这个功能在 POSIX（Linux，BSD，Mac）操作系统 Shell 编程中广泛使用，在 Windows 中也能使用。

使用标准输入输出经常是很有用的，这让你和终端前的用户交互。它也能让你操作输入输出文件，或者甚至让你的代码和其他程序组合在一起。

举一个例子，我们可以像这样在前面提供标准输入给 `callingpure.hs`：

```
$ echo John|runghc callingpure.hs
Greetings once again.  What is your name?
Pleased to meet you, John.
Your name contains 4 characters.
```

当 `callingpure.hs` 运行的时候，它不用等待键盘的输入，而是从 `echo` 程序接收 `John` 。注意输出也没有把 `John` 这个词放在一个分开的行，这和用键盘运行程序一样。终端一般回显所有你输入的东西给你，但这是一个技术上的输入，不会包含在输出流中。

7.2.5 删除和重命名文件

这一章到目前为止，我们已经讨论了文件的内容。现在让我们说一点文件自己的东西。`System.Directory` 提供了两个你可能觉得有用的函数。

> `removeFile` 接受一个参数，一个文件名，然后删除那个文件。`renameFile` 接受两个文件名：第一个是老的文件名，第二个是新的文件名。

如果新的文件名在另外一个目录中，你也可以把它想象成移动文件。在调用 `renameFile` 之前老的文件必须存在。如果新的文件已经存在了，它在重命名之前会被删除掉。

像很多其他接受文件名的函数一样，如果老的文件名不存在，`renameFile` 会引发一个异常。更多关于异常处理的信息你可以在**‘第十九章，错误处理 ‘_** 中找到。

在 `System.Directory` 中有很多其他函数，用来创建和删除目录，查找目录中文件列表，和测试文件是否存在。它们在**‘目录和文件信息 ‘_** 一节中讨论。

7.2.6 临时文件

程序员频繁需要用到临时文件。临时文件可能用来存储大量需要计算的数据，其他程序要使用的数据，或者很多其他的用法。

当你想一个办法来手动打开同名的多个文件，安全地做到这一点的细节在各个平台上都不相同。Haskell 提供了一个方便的函数叫做 `openTempFile`（还有一个对应的 `openBinaryTempFile` ）来为你处理这个难点。

`openTempFile` 接受两个参数：创建文件所在的目录，和一个命名文件的 “模板”。这个目录可以简单是 “.”，表示当前目录。或者你可以用 `System.Directory.getTemporaryDirectory` 去找指定机器上存

放临时文件最好的地方。这个模板用做文件名的基础，它会添加一些随机的字符来保证文件名是唯一的，从实际上保证被操作的文件具有独一无二的文件名。

`openTempFile` 返回类型是 `IO (FilePath, Handle)` 。元组的第一部分是创建的文件的名字，第二部分是用 `ReadWriteMode` 打开那个文件的一个句柄。当你处理完这个文件，你要 `hClose` 它并且调用 `removeFile` 删除它。看下面的例子中一个样本函数的使用。

7.3 扩展例子：函数式 I/O 和临时文件

这里有一个大一点的例子，它把很多这一章的还有前面几章的概念放在一起，还包含了一些没有介绍过的概念。看一下这个程序，看你是否能知道它是干什么的，是怎么做的。

```
-- file: ch07/tempfile.hs
import System.IO
import System.Directory(getTemporaryDirectory, removeFile)
import System.IO.Error(catchIOError)  -- 译注：原文为 catch ，在现在的环境中无法正确运行
import Control.Exception(finally)

-- The main entry point.  Work with a temp file in myAction.
main :: IO ()
main = withTempFile "mytemp.txt" myAction

{- The guts of the program.  Called with the path and handle of a temporary
file.  When this function exits, that file will be closed and deleted
because myAction was called from withTempFile. -}
myAction :: FilePath -> Handle -> IO ()
myAction tempname temph =
    do -- Start by displaying a greeting on the terminal
       putStrLn "Welcome to tempfile.hs"
       putStrLn $ "I have a temporary file at " ++ tempname

       -- Let's see what the initial position is
       pos <- hTell temph
       putStrLn $ "My initial position is " ++ show pos

       -- Now, write some data to the temporary file
       let tempdata = show [1..10]
       putStrLn $ "Writing one line containing " ++
           show (length tempdata) ++ " bytes: " ++
              tempdata
       hPutStrLn temph tempdata

       -- Get our new position.  This doesn't actually modify pos
```

(continues on next page)

(continued from previous page)

```
        -- in memory, but makes the name "pos" correspond to a different
        -- value for the remainder of the "do" block.
        pos <- hTell temph
        putStrLn $ "After writing, my new position is " ++ show pos

        -- Seek to the beginning of the file and display it
        putStrLn $ "The file content is: "
        hSeek temph AbsoluteSeek 0

        -- hGetContents performs a lazy read of the entire file
        c <- hGetContents temph

        -- Copy the file byte-for-byte to stdout, followed by \n
        putStrLn c

        -- Let's also display it as a Haskell literal
        putStrLn $ "Which could be expressed as this Haskell literal:"
        print c

{- This function takes two parameters: a filename pattern and another
function.  It will create a temporary file, and pass the name and Handle
of that file to the given function.

The temporary file is created with openTempFile.  The directory is the one
indicated by getTemporaryDirectory, or, if the system has no notion of
a temporary directory, "." is used.  The given pattern is passed to
openTempFile.

After the given function terminates, even if it terminates due to an
exception, the Handle is closed and the file is deleted. -}
withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
withTempFile pattern func =
    do -- The library ref says that getTemporaryDirectory may raise on
       -- exception on systems that have no notion of a temporary directory.
       -- So, we run getTemporaryDirectory under catch.  catch takes
       -- two functions: one to run, and a different one to run if the
       -- first raised an exception.  If getTemporaryDirectory raised an
       -- exception, just use "." (the current working directory).
ă ă ă  tempdir <- catchIOError (getTemporaryDirectory) (\_ -> return ".")
       (tempfile, temph) <- openTempFile tempdir pattern

       -- Call (func tempfile temph) to perform the action on the temporary
       -- file.  finally takes two actions.  The first is the action to run.
```

(continues on next page)

(continued from previous page)

```
    -- The second is an action to run after the first, regardless of
    -- whether the first action raised an exception.  This way, we ensure
    -- the temporary file is always deleted.  The return value from finally
    -- is the first action's return value.
    finally (func tempfile temph)
            (do hClose temph
                removeFile tempfile)
```

让我们从结尾开始看这个程序。 `writeTempFile` 函数证明 Haskell 当 I/O 被引入的时候没有忘记它的函数式特性。

这个函数接受一个 `String` 和另外一个函数，传给 `withTempFile` 的函数使用这个名字和一个临时文件的句柄调用。当函数退出时，这个临时文件被关闭和删除。所以甚至在处理 I/O 时，我们仍然可以发现为了方便传递函数作为参数的习惯。Lisp 程序员可能看到我们的 `withTempFile` 函数有点类似 Lisp 的 `with-open-file` 函数。

为了让程序能够更好地处理错误，我们需要为它添加一些异常处理代码。你一般需要临时文件在处理完成之后被删除，就算有错误发生。所以我们要确保删除发生。关于异常处理的更多信息，请看**‘第十九章：错误处理‘_**。

让我们回到这个程序的开头，`main` 被简单定义成 `withTempFile "mytemp.txt" myAction` 。然后，`myAction` 将会被调用，使用名字和这个临时文件的句柄作为参数。

`myAction` 显示一些信息到终端，写一些数据到文件，寻找文件的开头，并且使用 `hGetContents` 把数据读取回来。然后把文件的内容按字节地，通过 `print c` 当做 Haskell 字面量显示出来。这和 `putStrLn (show c)` 一样。

我们看一下输出：

```
$ runhaskell tempfile.hs
Welcome to tempfile.hs
I have a temporary file at /tmp/mytemp8572.txt
My initial position is 0
Writing one line containing 22 bytes: [1,2,3,4,5,6,7,8,9,10]
After writing, my new position is 23
The file content is:
[1,2,3,4,5,6,7,8,9,10]

Which could be expressed as this Haskell literal:
"[1,2,3,4,5,6,7,8,9,10]\n"
```

每次你运行这个程序，你的临时文件的名字应该有点细微的差别，因为它包含了一个随机生成的部分。看一下这个输出，你可能会问一些问题？

1. 为什么写入一行 22 个字节之后你的位置是 23？

2. 为什么文件内容显示之后有一个空行?

3. 为什么 Haskell 字面量显示的最后有一个 `\n` ?

你可能能猜到这三个问题的答案都是相关的。看看你能不能在一会内答出这些题。如果你需要帮助，这里有解释:

1. 是因为我们用 `hPutStrLn` 替代 `hPutStr` 来写这个数据。

   `hPutStrLn` 总是在结束一行的时候在结尾处写上一个 `\n` ，而这个没有出现在 `tempdata` 。

2. 我们用 `putStrLn c` 来显示文件内容 `c` 。因为数据原来使用 `hPutStrLn` 来写的，`c` 结尾处有一个换行符，并且 `putStrLn` 又添加了第二个换行符，结果就是多了一个空行。3. 这个 `\n` 是来自原始的 `hPutStrLn` 的换行符。

最后一个注意事项，字节数目可能在一些操作系统上不一样。比如 Windows，使用连个字节序列 `\r\n` 作为行结束标记，所以在 Windows 平台你可能会看到不同。

7.4 惰性 I/O

这一章到目前为止，你已经看了一些相当传统的 I/O 例子。单独请求和处理每一行或者每一块数据。

Haskell 还为你准备了另一种方法。因为 Haskell 是一种惰性语言，意思是任何给定的数据片只有在它的值必须要知道的情况下才会被计算。有一些新奇的方法来处理 I/O。

7.4.1 hGetContents

一种新奇的处理 I/O 的办法是 `hGetContents` 函数，这个函数类型是 `Handle -> IO String` 。这个返回的 `String` 表示 `Handle` 所给文件里的所有数据。

在一个严格求值（strictly-evaluated）的语言中，使用这样的函数不是一件好事情。读取一个 2KB 文件的所有内容可能没事，但是如果你尝试去读取一个 500GB 文件的所有内容，你很可能因为缺少内存去存储这些数据而崩溃。在这些语言中，传统上你会采用循环去处理文件的全部数据的机制。

但是 `hGetContents` 不一样。它返回的 `String` 是惰性估值的。在你调用 `hGetContents` 的时刻，实际上没有读任何东西。数据只从句柄读取，作为处理的一个元素（字符）列表。

`String` 的元素一直都用不到，Haskell 的垃圾收集器会自动释放那块内存。

所有这些都是完全透明地发生的。因为函数的返回值是一个如假包换的纯 String ，所以它可以被传递给非 I/O 的纯代码。让我们快速看一个例子。回到**'操作文件和句柄 '_** 一节，你看到一个命令式的程序，它把整个文件内容转换成大写。它的命令式算法和你在其他语言看到的很类似。接下来展示的是一个利用了惰性求值实现的更简单的算法。

```haskell
-- file: ch07/toupper-lazy1.hs
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       inpStr <- hGetContents inh
       let result = processData inpStr
       hPutStr outh result
       hClose inh
       hClose outh

processData :: String -> String
processData = map toUpper
```

注意到 hGetContents 为我们处理所有的读取工作。看一下 processData ，它是一个纯函数，因为它没有副作用，并且每次调用的时候总是返回相同的结果。它不需要知道，也没办法告诉它，它的输入是惰性从文件读取的。不管是 20 个字符的字面量还是硬盘上 500GB 的数据它都可以很好的工作。

你可以用 **ghci** 验证一下：

```
ghci> :load toupper-lazy1.hs
[1 of 1] Compiling Main             ( toupper-lazy1.hs, interpreted )
Ok, modules loaded: Main.
ghci> processData "Hello, there!  How are you?"
"HELLO, THERE!  HOW ARE YOU?"
ghci> :type processData
processData :: String -> String
ghci> :type processData "Hello!"
processData "Hello!" :: String
```

> **Warning:** 如果我们在 inpStr 被使用后（processData 调用那）还拿着它不放的话，那么我们的程序在内存使用上就会变的很低效。

这是因为了在以后还可以使用 inpStr 的值，编译器会被迫在内存中保留 inpStr 。这里我们知道 inpStr 将不会被重用，它一被使用完就会被释放内存。只要记住：内存只有在最后一次使用完才会被释放。

这个程序为了清楚地表明使用了纯代码，显得有点囧嗦。这里有更加简洁的版本，新版本在下一个例子里：

```haskell
-- file: ch07/toupper-lazy2.hs
```

(continues on next page)

(continued from previous page)

```
import System.IO
import Data.Char(toUpper)

main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       inpStr <- hGetContents inh
       hPutStr outh (map toUpper inpStr)
       hClose inh
       hClose outh
```

你在使用 hGetContents 的时候不要求去使用输入文件的所有数据。任何时候 Haskell 系统能决定整个 hGetContents 返回的字符串能否被垃圾收集掉，意思就是它不会再被使用，文件会自动被关闭。同样的原理适用于从文件读取的数据。当给定的数据片不会再被使用的任何时候，Haskell 会释放它保存的那块内存。严格意义上来讲，我们在这个例子中根本不必要去调用 hClose 。但是，养成习惯去调用还是个好的实践。以后对程序的修改可能让 hClose 的调用变得重要。

> **Warning:** 即使在余下的程序中不再显示引用文件句柄，你也必须在使用 hGetContents 的结果之后再关闭句柄，否则将会导致程序丢失部分或所有文件数据。因为 Haskell 是一门惰性语言，我们通常会认为，它只有在需要输出计算结果的时候才会使用输入。

[scarletsky 译注：

```
inpStr <- hGetContents inh
let outStr = map toUpper inpStr
hClose inh
```

这里的 outStr 并没有使用 hGetContents 的结果，因为它并没有输出计算结果。这种情况就属于上面提到的：在使用 hGetContents 结果前关闭文件句柄，因此这段代码是错误的。

]

7.4.2 readFile 和 writeFile

Haskell 程序员经常使用 hGetContents 作为一个过滤器。他们从一个文件读取，在数据上做一些事情，然后把结果写到其他地方。这很常见，有很多种快捷方式可以做。

readFile 和 writeFile 是把文件当做字符串处理的快捷方式。

他们处理所有细节，包括打开文件，关闭文件，读取文件和写入文件。 readFile 在内部使用 hGetContents 。

你能猜到这些函数的 Haskell 类型吗？我们用 **ghci** 检查一下：

```
ghci> :type readFile
readFile :: FilePath -> IO String
ghci> :type writeFile
writeFile :: FilePath -> String -> IO ()
```

现在有一个例子程序使用了 `readFile` 和 `writeFile`：

```
-- file: ch07/toupper-lazy3.hs
import Data.Char(toUpper)

main = do
       inpStr <- readFile "input.txt"
       writeFile "output.txt" (map toUpper inpStr)
```

看一下，这个程序的内部只有两行。 `readFile` 返回一个惰性 `String`，我们保存在 `inpStr`。

然后我们拿到它，处理它，然后把它传给 `writeFile` 函数去写入。

`readFile` 和 `writeFile` 都不提供一个句柄给你操作，所以没有东西要去 `hClose`。 `readFile` 在内部使用 `hGetContents`，底下的句柄在返回的 `String` 被垃圾回收或者所有输入都被消费之后就会被关闭。`writeFile` 会在供应给它的 `String` 全部被写入之后关闭它底下的句柄。

7.4.3 一言以蔽惰性输出

到现在为止，你应该理解了 Haskell 的惰性输入怎么工作的。但是在输入的时候惰性是怎么样的呢?

据你所知，Haskell 中的所有东西都是在需要的时候才被求值的。因为像 `writeFile` 和 `putStr` 这样的函数写传递给它们的整个 `String`，所以这整个 `String` 必须被求值。所以保证 `putStr` 的参数会被完全求值。

但是输入的惰性是什么意思呢？在上面的例子中，对 `putStr` 或者 `writeFile` 的调用会强制一次性把整个输入字符串载入到内存中吗，直接全部写出?

答案是否定的。 `putStr`（以及所有类似的输出函数）在它变得可用时才写出数据。

他们也不需要保存已经写的数据，所以只要程序中没有其他地方需要它，这块内存就可以立即释放。在某种意义上，你可以把这个在 `readFile` 和 `writeFile` 之间的 `String` 想成一个连接它们两个的管道。数据从一头进去，通过某种方式传递，然后从另外一头流出。

你可以自己验证这个，通过给 `toupper-lazy3.hs` 产生一个大的 `input.txt`。处理它可能时间要花一点时间，但是在处理它的时候你应该能看到一个常量的并且低的内存使用。

7.4.4 interact

你学习了 readFile 和 writeFile 处理读文件，做个转换，然后写到不同文件的普通情形。还有一个比他还普遍的情形：从标准输入读取，做一个转换，然后把结果写到标准输出。对于这种情形，有一个函数叫做 interact 。

interact 函数的类型是 (String -> String) -> IO () 。

也就是说，它接受一个参数：一个类型为 String -> String 的函数。 getContents 的结果传递给这个函数，也就是，惰性读取标准输入。

这个函数的结果会发送到标准输出。

我们可以使用 interact 来转换我们的例子程序去操作标准输入和标准输出。这里有一种方式：

```
-- file: ch07/toupper-lazy4.hs
import Data.Char(toUpper)

main = interact (map toUpper)
```

来看一下，一行就完成了我们的变换。要实现上一个例子同样的效果，你可以像这样来运行这个例子：

```
$ runghc toupper-lazy4.hs < input.txt > output.txt
```

或者，如果你想看输出打印在屏幕上的话，你可以打下面的命令：

```
$ runghc toupper-lazy4.hs < input.txt
```

如果你想看看 Haskell 是否真的一接收到数据块就立即写出的话，运行 runghc toupper-lazy4.hs ，不要其他的命令行参数。你可以看到每一个你输入的字符都会立马回显，但是都变成大写了。缓冲区可能改变这种行为，更多关于缓冲区的看这一章后面的**‘缓冲区 ‘_** 一节。如果你看到你输入的没一行都立马回显，或者甚至一段时间什么都没有，那就是缓冲区造成的。

你也可以用 interactive 写一个简单的交互程序。让我们从一个简单的例子开始：

```
-- file: ch07/toupper-lazy5.hs
import Data.Char(toUpper)

main = interact (map toUpper . (++) "Your data, in uppercase, is:\n\n")
```

Tip: 如果 . 运算符不明白的话，你可以参考**‘使用组合来重用代码 ‘_** 一节。

这里我们在输出的开头添加了一个字符串。你可以发现这个问题吗？

因为我们在 (++) 的结果上调用 map ，这个头自己也会显示成大写。我们可以这样来解决：

```haskell
-- file: ch07/toupper-lazy6.hs
import Data.Char(toUpper)

main = interact ((++) "Your data, in uppercase, is:\n\n" .
                 map toUpper)
```

现在把头移出了 map 。

7.4.5 interact 过滤器

interact 另一个通常的用法是过滤器。比如说你要写一个程序，这个程序读一个文件，并且输出所有包含字符“a”的行。你可能会这样用 interact 来实现：

```haskell
-- file: ch07/filter.hs
main = interact (unlines . filter (elem 'a') . lines)
```

这里引入了三个你还不熟悉的函数。让我们在 **ghci** 里检查它们的类型：

```
ghci> :type lines
lines :: String -> [String]
ghci> :type unlines
unlines :: [String] -> String
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
```

你只是看它们的类型，你能猜到它们是干什么的吗？如果不能，你可以在**‘热身：快捷文本行分割 ‘_** 一节和**‘特殊字符串处理函数 ‘_** 一节找到解释。你会频繁看到 lines 和 unlines 和 I/O 一起使用。最后，elem 接受一个元素和一个列表，如果元素在列表中出现则返回 True 。

试着用我们的标准输入例子来运行：

```
$ runghc filter.hs < input.txt
I like Haskell
Haskell is great
```

果然，你得到包含“a”的两行。惰性过滤器是使用 Haskell 强大的方式。你想想看，一个过滤器，就像标准 Unix 程序 **Grep** ，听起来很像一个函数。它接受一些输入，应用一些计算，然后生成一个意料之中的输出。

7.5 The IO Monad

这个时候你已经看了若干 Haskell 中 I/O 的例子。让我们花点时间回想一下，并且思考下 I/O 是怎么和更广阔的 Haskell 语言相关联的。

因为 Haskell 是一个纯的语言，如果你给特定的函数一个指定的参数，每次你给它那个参数这个函数将会返回相同的结果。此外，这个函数不会改变程序的总体状态的任何东西。

你可能想知道 I/O 是怎么融合到整体中去的呢？当然如果你想从键盘输入中读取一行，去读输入的那个函数肯定不可能每次都返回相同的结果。是不是？此外，I/O 都是和改变状态相关的。I/O 可以点亮终端上的一个像素，可以让打印机的纸开始出来，或者甚至是让一个包裹从仓库运送到另一个大洲。I/O 不只是改变一个程序的状态。你可以把 I/O 想成可以改变世界的状态。

7.5.1 动作（Actions）

大多数语言在纯函数和非纯函数之间没有明确的区分。Haskell 的函数有数学上的意思：它们是纯粹的计算过程，并且这些计算不会被外部所影响。此外，这些计算可以在任何时候、按需地执行。

显然，我们需要其他一些工具来使用 I/O。Haskell 里的这个工具叫做动作（Actions）。动作类似于函数，它们在定义的时候不做任何事情，而在它们被调用时执行一些任务。I/O 动作被定义在 `IO` Monad。Monad 是一种强大的将函数链在一起的方法，在**'第十四章：Monad'_** 会讲到。为了理解 I/O 你不是一定要理解 Monad，只要理解操作的返回类型都带有 `IO` 就行了。我们来看一些类型：

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

`putStrLn` 的类型就像其他函数一样，接受一个参数，返回一个 `IO ()`。这个 `IO ()` 就是一个操作。如果你想你可以在纯代码中保存和传递操作，虽然我们不经常这么干。一个操作在它被调用前不做任何事情。我们看一个这样的例子：

```
-- file: ch07/actions.hs
str2action :: String -> IO ()
str2action input = putStrLn ("Data: " ++ input)

list2actions :: [String] -> [IO ()]
list2actions = map str2action

numbers :: [Int]
numbers = [1..10]

strings :: [String]
strings = map show numbers

actions :: [IO ()]
actions = list2actions strings

printitall :: IO ()
```

(continues on next page)

(continued from previous page)

```
printitall = runall actions

-- Take a list of actions, and execute each of them in turn.
runall :: [IO ()] -> IO ()
runall [] = return ()
runall (firstelem:remainingelems) =
    do firstelem
       runall remainingelems

main = do str2action "Start of the program"
          printitall
          str2action "Done!"
```

str2action 这个函数接受一个参数并返回 IO ()，就像你在 main 结尾看到的那样，你可以直接在另一个操作里使用这个函数，它会立刻打印出一行。或者你可以保存（不是执行）纯代码中的操作。你可以在 list2actions 里看到保存的例子，我们在 str2action 用 map，返回一个操作的列表，就和操作其他纯数据一样。所有东西都通过 printall 显示出来，而 printall 是用纯代码写的。

虽然我们定义了 printall，但是直到它的操作在其他地方被求值的时候才会执行。现在注意，我们是怎么在 main 里把 str2action 当做一个 I/O 操作使用，并且执行了它。但是先前我们在 I/O Monad 外面使用它，只是把结果收集进一个列表。

你可以这样来思考：do 代码块中的每一个声明，除了 let，都要产生一个 I/O 操作，这个操作在将来被执行。

对 printall 的调用最后会执行所有这些操作。实际上，因为 HASKELL 是惰性的，所以这些操作直到这里才会被生成。实际上，因为 Haskell 是惰性的，所以这些操作直到这里才会被生成。

当你运行这个程序时，你的输出看起来像这样：

```
Data: Start of the program
Data: 1
Data: 2
Data: 3
Data: 4
Data: 5
Data: 6
Data: 7
Data: 8
Data: 9
Data: 10
Data: Done!
```

我们实际上可以写的更紧凑。来看看这个例子的修改：

```haskell
-- file: ch07/actions2.hs
str2message :: String -> String
str2message input = "Data: " ++ input

str2action :: String -> IO ()
str2action = putStrLn . str2message

numbers :: [Int]
numbers = [1..10]

main = do str2action "Start of the program"
          mapM_ (str2action . show) numbers
          str2action "Done!"
```

注意在 str2action 里对标准函数组合运算符的使用。在 main 里面，有一个对 mapM_ 的调用，这个函数和 map 类似，接受一个函数和一个列表。提供给 mapM_ 的函数是一个 I/O 操作，这个操作对列表中的每一项都执行。

> mapM_ 扔掉了函数的结果，但是如果你想要 I/O 的结果，你可以用 mapM 返回一个 I/O 结果的列表。

来看一下它们的类型：

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Tip: 这些函数其实不仅仅可以做 I/O 相关的操作，所有的 Monad 都可以使用他们。

到现在为止，你看到“M”就把它想成“IO”。还有，那些以下划线结尾的函数一般不管它们的返回值。

为什么我们有了 map 还要有一个 mapM，因为 map 是返回一个列表的纯函数，它实际上不直接执行也不能执行操作。
mapM 是一个 IO Monad 里面的可以执行操作的实用程序。

现在回到 main，mapM_ 在 numbers . show 每个元素上应用 (str2action . show)，number . show 把每个数字
mapM_ 把这些单独的操作组合成一个更大的操作，然后打印出这些行。

7.5.2 串联化（Sequencing）

do 代码块实际上是把操作连接在一起的快捷记号。有两个运算符可以用来代替 do 代码块：>> 和 >>= 。在 **ghci** 看一下它们的类型：

```
ghci> :type (>>)
(>>) :: (Monad m) => m a -> m b -> m b
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

>> 运算符把两个操作串联在一起：第一个操作先运行，然后是第二个。运算符的计算的结果是第二个操作的结果，第一个操作的结果被丢弃了。这和在 do 代码块中只有一行是类似的。你可能会写 `putStrLn "line 1" >> putStrLn "line 2"` 来测试这一点。它会打印出两行，把第一个 putStrLn 的结果丢掉了，值提供第二个操作的结果。

>>= 运算符运行一个操作，然后把它的结果传递给一个返回操作的函数。那样第二个操作可以同样运行，而且整个表达式的结果就是第二个操作的结果。例如，你写 `getLine >>= putStrLn`，这会从键盘读取一行，然后显示出来。

让我们重写例子中的一个，不用 do 代码快。还记得这一章开头的这个例子吗?

```
-- file: ch07/basicio.hs
main = do
       putStrLn "Greetings!  What is your name?"
       inpStr <- getLine
       putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

我们不用 do 代码块来重写它：

```
-- file: ch07/basicio-nodo.hs
main =
    putStrLn "Greetings!  What is your name?" >>
    getLine >>=
    (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

你定义 do 代码块的时候，Haskell 编译器内部会把它翻译成像这样。

Tip: 忘记了怎么使用 \ (lambda 表达式) 了吗？参见**‘匿名（lambda）函数‘_**一节。

7.5.3 Return 的本色

在这一章的前面，我们提到 return 很可能不是它看起来的那样。很多语言有一个关键字叫做 return，它取消函数的执行并立即给调用者一个返回值。

Haskell 的 return 函数很不一样。在 Haskell 中，return 用来在 Monad 里面包装数据。当说 I/O 的时候，return 用来拿到纯数据并把它带入 IO Monad。

为什么我们需要那样做？还记得结果依赖 I/O 的所有东西都必须在一个 IO Monad 里面吗？所以如果我们在写一个执行 I/O 的函数，然后一个纯的计算，我们需要用 `return` 来让这个纯的计算能给函数返回一个合适的值。否则，会发生一个类型错误。这儿有一个例子：

```
-- file: ch07/return1.hs
import Data.Char(toUpper)

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return ((toUpper . head $ inpStr) == 'Y')
```

我们有一个纯的计算产生一个 `Bool`，这个计算传给了 `return`，`return` 把它放进了 `IO` Monad。因为它是 `do` 代码块的最后一个值，所以它变成 `isGreen` 的返回值，而不是因为我们用了 `return` 函数。

这有一个相同程序但是把纯计算移到一个单独的函数里的版本。这帮助纯代码保持分离，并且让意图更清晰。

```
-- file: ch07/return2.hs
import Data.Char(toUpper)

isYes :: String -> Bool
isYes inpStr = (toUpper . head $ inpStr) == 'Y'

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return (isYes inpStr)
```

最后，有一个人为的例子，这个例子显示了 `return` 确实没有在 `do` 代码块的结尾出现。在实践中，通常是这样的，但是不一定需要这样。

```
-- file: ch07/return3.hs
returnTest :: IO ()
returnTest =
    do one <- return 1
       let two = 2
       putStrLn $ show (one + two)
```

注意，我们用了 `<-` 和 `return` 的组合，但是 `let` 是和简单字面量组合的。这是因为我们需要都是纯的值才能去相加它们，`<-` 把东西从 Monad 里面拿出来，实际上就是 `return` 的反作用。在 **ghci** 运行一下，你会看到和预期一样显示 3。

7.6 Haskell 实际上是命令式的吗？

这些 do 代码块可能看起来很像一个命令式语言？毕竟大部分时间你给了一些命令按顺序运行。

但是 Haskell 在它的核心上是一个惰性语言。时常在需要给 I/O 串联操作的时候，是由一些工具完成的，这些工具就是 Haskell 的一部分。Haskell 通过 I/O Monad 实现了出色的 I/O 和语言剩余部分的分离。

7.7 惰性 I/O 的副作用

本章前面你看到了 hGetContents ，我们解释说它返回的 String 可以在纯代码中使用。

关于副作用我们需要得到一些更具体的东西。当我们说 Haskell 没有副作用，这到底意味着什么？

在一定程度上，副作用总是可能的。一个写的不好的循环，就算写成纯代码形式的，也会造成系统内存耗尽和机器崩溃，或者导致数据交换到硬盘上。

当我们说没有副作用的时候，我们意思是，Haskell 中的纯代码不能运行那些能触发副作用的命令。纯函数不能修改全局变量，请求 I/O，或者运行一条关闭系统的命令。

当你有从 hGetContents 拿到一个 String ，你把它传给一个纯函数，这个函数不知道这个 String 是由硬盘文件上来的。这个函数表现地还是和原来一样，但是处理那个 String 的时候可能造成环境发出 I/O 命令。纯函数是不会发出 I/O 命令的，它们作为处理正在运行的纯函数的一个结果，就和交换内存到磁盘的例子一样。

有时候，你在 I/O 发生时需要更多的控制。可能你正在从用户那里交互地读取数据，或者通过管道从另一个程序读取数据，你需要直接和用户交流。在这些时候，hGetContents 可能就不合适了。

7.8 缓冲区（Buffering）

I/O 子系统是现代计算机中最慢的部分之一。完成一次写磁盘的时间是一次写内存的几千倍。在网络上的写入还要慢成百上千倍。就算你的操作没有直接和磁盘通信，可能数据被缓存了，I/O 还是需要一个系统调用，这个也会减慢速度。

由于这个原因，现代操作系统和编程语言都提供了工具来帮助程序当涉及到 I/O 的时候更好地运行。操作系统一般采用缓存（Cache），把频繁使用的数据片段保存在内存中，这样就能更快的访问了。

编程语言通常采用缓冲区。就是说，它们可能从操作系统请求一大块数据，就算底层代码是一次一个字节地处理数据的。通过这样，它们可以实现显著的性能提升，因为每次向操作系统的 I/O 请求带来一次处理开销。缓冲区允许我们以少得多的 I/O 请求次数去读取相同数量的数据。

7.8.1 缓冲区模式

Haskell 中有 3 种不同的缓冲区模式，它们定义成 `BufferMode` 类型：`NoBuffering`，`LineBuffering` 和 `BlockBuffering`。

`NoBuffering` 就和它听起来那样-没有缓冲区。通过像 `hGetLine` 这样的函数读取的数据是从操作系统一次一个字符读取的。写入的数据会立即写入，也是一次一个字符地写入。因此，`NoBuffering` 通常性能很差，不适用于一般目的的使用。

`LineBuffering` 当换行符输出的时候会让输出缓冲区写入，或者当缓冲区太大的时候。在输入上，它通常试图去读取块上所有可用的字符，直到它首次遇到换行符。当从终端读取的时候，每次按下回车之后它会立即返回数据。这个模式经常是默认模式。

`BlockBuffering` 让 Haskell 在可能的时候以一个固定的块大小读取或者写入数据。这在批处理大量数据的时候是性能最好的，就算数据是以行存储的也是一样。然而，这个对于交互程序不能用，因为它会阻塞输入直到一整块数据被读取。

> `BlockBuffering` 接受一个 `Maybe` 类型的参数：如果是 `Nothing`，它会使用一个自定的缓冲区大小，或者你可以使用一个像 `Just 4096` 的设定，设置缓冲区大小为 4096 个字节。

默认的缓冲区模式依赖于操作系统和 Haskell 的实现。你可以通过调用 `hGetBuffering` 查看系统的当前缓冲区模式。当前的模式可以通过 `hSetBuffering` 来设置，它接受一个 `Handle` 和 `BufferMode`。例如，你可以写 `hSetBuffering stdin (BlockBuffering Nothing)`。

7.8.2 刷新缓冲区

对于任何类型的缓冲区，你可能有时候需要强制 Haskell 去写出所有保存在缓冲区里的数据。有些时候这个会自动发生：比如，对 `hClose` 的调用。有时候你可能需要调用 `hFlush` 作为代替，`hFlush` 会强制所有等待的数据立即写入。这在句柄是一个网络套接字的时候，你想数据被立即传输，或者你想让磁盘的数据给其他程序使用，而其他程序也正在并发地读那些数据的时候都是有用的。

7.9 读取命令行参数

很多命令行程序喜欢通过命令行来传递参数。 `System.Environment.getArgs` 返回 `IO [String]` 列出每个参数。

这和 C 语言的 `argv` 一样，从 `argv[1]` 开始。程序的名字（C 语言的 `argv[0]`）用 `System.Environment.getProgName` 可以得到。

`System.Console.GetOpt` 模块提供了一些解析命令行选项的工具。如果你有一个程序，它有很复杂的选项，你会觉得它很有用。你可以在**‘命令行解析 ‘_** 一节看到一个例子和使用方法。

7.10 环境变量

如果你需要阅读环境变量，你可以使用 `System.Environment` 里面两个函数中的一个：`getEnv` 或者 `getEnvironment`。

`getEnv` 查找指定的变量，如果不存在会抛出异常。`getEnvironment` 用一个 `[(String, String)]` 返回整个环境，然后你可以用 `lookup` 这样的函数来找你想要的环境条目。

在 Haskell 设置环境变量没有采用跨平台的方式来定义。如果你在像 Linux 这样的 POSIX 平台上，你可以使用 `System.Posix.Env` 模块中的 `putEnv` 或者 `setEnv`。环境设置在 Windows 下面没有定义。

CHAPTER

EIGHT

第 8 章：高效文件处理、正则表达式、文件名匹配

8.1 高效文件处理

下面是个简单的基准测试，读取一个由数字构成的文本文件，并打印它们的和。

```
-- file: ch08/SumFile.hs
main = do
    contents <- getContents
    print (sumFile contents)
  where sumFile = sum . map read . words
```

尽管读写文件时，默认使用 String 类型，但它并不高效，所以这样简单的程序效率会很糟糕。

一个 String 代表一个元素类型为 Char 的列表；列表的每个元素被单独分配内存，并有一定的写入开销。对那些要读取文本及二进制数据的程序来说，这些因素会影响内存消耗和执行效率。在这个简单的测试中，即使是 Python 那样的解释型语言的表现也会大大好于使用 String 的 Haskell 代码。

bytestring 库是 String 类型的一个快速、经济的替代品。在保持 Haskell 代码的表现力和简洁的同时，使用 bytestring 编写的代码在内存占用和执行效率经常可以达到或超过 C 代码。

这个库提供两个模块。每个都定义了与 String 类型上函数对应的替代物。

- Data.ByteString 定义了一个名为 ByteString 的严格类型，其将一个字符串或二进制数据或文本用一个数组表示。
- Data.ByteString.Lazy 模块定义了一个惰性类型，同样命名为 ByteString 。其将字符串数据表示为一个由 块组成的列表，每个块是大小为 64KB 的数组。

这两种 ByteString 适用于不同的场景。对于大体积的文件流 (几百 MB 至几 TB)，最好使用惰性的 ByteString 。其块的大小被调整得对现代 CPU 的 L1 缓存特别友好，并且在流中已经被处理过块可以被垃圾收集器快速丢弃。

对于不在意内存占用而且需要随机访问的数据，最好使用严格的 ByteString 类型。

8.1.1 二进制 I/O 和有限载入

让我们来开发一个小函数以说明 ByteString API 的一些用法。我们将检测一个文件是否是 ELF object 文件：这种文件类型几乎被所有现代类 Unix 系统作为可执行文件。

这个简单的问题可以通过查看文件头部的四个字节解决，看他们是否匹配某个特定的字节序列。表示某种文件类型的字节序列通常被称为 魔法数。

```
-- file: ch08/ElfMagic.hs
import qualified Data.ByteString.Lazy as L

hasElfMagic :: L.ByteString -> Bool
hasElfMagic content = L.take 4 content == elfMagic
    where elfMagic = L.pack [0x7f, 0x45, 0x4c, 0x46]
```

我们使用 Haskell 的 有限载入语法载入 ByteString 模块，像上面 import qualified 那句那样。这样可以把一个模块关联到另一个我们选定的名字。

例如，使用到惰性 ByteString 模块的 take 函数时，要写成 L.take ，因为我们将这个模块载入到了 L 这个名字下。若没有明确指明使用哪个版本的函数，如此处的 take ，编译器会报错。

我们将一直使用有限载入语法使用 ByteString 模块，因为其中提供的很多函数与 Prelude 模块中的函数重名。

Note: 有限载入使得可以方便地切换两种 ByteString 类型。只需要在代码的头部改变 import 声明；剩余的代码可能无需任何修改。你可以方便地比较两种类型，以观察哪种类型更符合你程序的需要。

无论是否使用有限载入，始终可以使用模块的全名来识别某些混淆。例如，Data.ByteString.Lazy.length 和 L.length 表示相同的函数，Prelude.sum 和 sum 也是如此。

ByteString 模块为二进制 I/O 而设计。Haskell 中表达字节的类型是 Word8 ；如果需要按名字引用它，需要将其从 Data.Word 模块载入。

L.pack 函数接受一个由 Word8 组成的列表，并将其装入一个惰性 ByteString （L.unpack 函数的作用恰好相反。）。hasElfMagic 函数简单地将一个 ByteString 的前四字节与一个魔法数相比较。

我们使用了典型的 Haskell 风格编写 hasElfMagic 函数，其并不执行 I/O。这里是如何在真正的文件上使用它。

```
-- file: ch08/ElfMagic.hs
isElfFile :: FilePath -> IO Bool
isElfFile path = do
  content <- L.readFile path
  return (hasElfMagic content)
```

L.readFile 函数是 readFile 的惰性 ByteString 等价物。它是惰性执行的，将文件读取为数据是需要的。它也很高效，立即读取 64KB 大小的块。对我们的任务而言，惰性 ByteString 是一个好选择，我们可以安全的将这个函数应用在任意大小的文件上。

8.1.2 文本 I/O

方便起见，bytestring 库提供两个具有有限文本 I/O 功能的模块，Data.ByteString.Char8 和 Data.ByteSring.Lazy.Char8 。它们将每个字符串的元素暴露为 Char 而非 Word8 。

> **Warning:** 这些模块中的函数适用于单字节大小的 Char 值，所以他们仅适用于 ASCII 及某些欧洲字符集。大于 255 的值将被截断。

这两个面向字符的 bytestring 模块提供了用于文本处理的函数。以下文件包含了一家知名互联网公司在 2008 年中期每个月的股价。

如何在这一系列记录中找到最高收盘价呢？收盘价位于以逗号分隔的第四列。以下函数从单行数据中获取收盘价。

```
-- file: ch08/HighestClose.hs
import qualified Data.ByteString.Lazy.Char8 as L

closing = readPrice . (!!4) . L.split ','
```

这个函数使用 point-free 风格编写，我们要从右向左阅读。L.split 函数将一个惰性 ByteString 按某个分隔符切分为一个由 ByteString 组成的列表。(!!) 操作符检索列表中的第 k 个元素。readPrice 函数将一个表示小数的字符串转换为一个数。

```
- file: ch08/HighestClose.hs
readPrice :: L.ByteString -> Maybe Int
readPrice str =
    case L.readInt str of
      Nothing             -> Nothing
      Just (dollars,rest) ->
        case L.readInt (L.tail rest) of
          Nothing           -> Nothing
          Just (cents,more) ->
            Just (dollars * 100 + cents)
```

我们使用 L.readInt 函数来解析一个整数。当发现数字时，它会将一个整数和字符串的剩余部分一起返回。L.readInt 在解析失败时返回 Nothing ，这导致我们的函数稍有些复杂。

查找最高收盘价的函数很容易编写。

```
-- file: ch08/HighestClose.hs
highestClose = maximum . (Nothing:) . map closing . L.lines

highestCloseFrom path = do
    contents <- L.readFile path
    print (highestClose contents)
```

不能对空列表使用 maximum 函数，所以我们耍了点小把戏。

```
ghci> maximum [3,6,2,9]
9
ghci> maximum []
*** Exception: Prelude.maximum: empty list
```

我们想在没有股票数据时也不抛出异常，所以用 (Nothing:) 这个表达式来确保输入到 maximum 函数的由 Maybe Int 值构成的列表总是非空。

```
ghci> maximum [Nothing, Just 1]
Just 1
ghci> maximum [Nothing]
Nothing
```

我们的函数工作正常吗?

```
ghci> :load HighestClose
[1 of 1] Compiling Main             ( HighestClose.hs, interpreted )
Ok, modules loaded: Main.
ghci> highestCloseFrom "prices.csv"
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Just 2741
```

因为我们把逻辑和 I/O 分离开了，所以即使不创建一个空文件也可以测试无数据的情况。

```
ghci> highestClose L.empty
Nothing
```

8.2 匹配文件名

很多面向操作系统的编程语言提供了检测某个文件名是否匹配给定模式的库函数，或者返回一个匹配给定模式的文件列表。在其他语言中，这个函数通常叫做 fmatch 。尽管 Haskell 标准库提供了很多有用的系统编程设施，但是并没有提供这类用于匹配文件名的函数。所以我们可以自己开发一个。

我们需要处理的模式种类通常称为 glob 模式（我们将使用这个术语），通配符模式，或称 shell 风格模式。它们仅是一些简单规则。你可能已经了解了，但是这里将做一个简要的回顾。

Note:

- 对某个模式的匹配从字符串头部开始，在字符串尾部结束。
- 多数文本字符匹配自身。例如，文本 `foo` 作为模式匹配其自身 `foo` ，且在一个输入字符串中仅匹配 `foo` 。
- `*` (星号) 意味着“匹配所有”；其将匹配所有文本，包括空字符串。例如, 模式 `foo*` 将匹配任意以 `foo` 开头的字符串，比如 `foo` 自身，`foobar` ，或 `foo.c` 。模式 `quux*.c` 将匹配任何以 `quux` 开头且以 `.c` 结束的字符串，如 `quuxbaz.c` 。
- `?` (问号) 匹配任意单个字符。模式 `pic??.jpg` 将匹配类似 `picaa.jpg` 或 `pic01.jpg` 的文件名。
- `[` (左方括号) 将开始定义一个字符类，以] 结束。其意思是“匹配在这个字符类中的任意字符”。`[!` 开启一个否定的字符类，其意为“匹配不在这个字符类中的任意字符”。

用 - (破折号) 连接的两个字符，是一种表示范围的速记方法，表示：“匹配这个围内的任意字符”。

字符类有一个附加的条件；其不可为空。在 `[` 或 `[!` 后的字符是这个字符类的一部分，所以我们可以编写包含 `]` 的字符类，如 `[]aeiou]` 。模式 `pic[0-9].[pP][nN][gG]` 将匹配由字符串 `pic` 开始，跟随单个数字，最后是字符串 `.png` 的任意大小写形式。

尽管 Haskell 的标准库没有提供匹配 glob 模式的方法，但它提供了一个良好的正则表达式库。Glob 模式仅是一个从正则表达式中切分出来的略有不同的子集。很容易将 glob 模式转换为正则表达式，但在此之前，我们首先要了解怎样在 Haskell 中使用正则表达式。

8.3 Haskell 中的正则表达式

在这一节，我们将假设读者已经熟悉 Python、Perl 或 Java 等其他语言中的正则表达式。

为了简洁，此后我们将“regular expression”简写为 regexp。

我们将以与其他语言对比的方式介绍 Haskell 如何处理 regexp，而非从头讲解何为 regexp。Haskell 的正则表达式库比其他语言具备更加强大的表现力，所以我们有很多可以聊的。

在我们对 regexp 库的探索开始时，只需使用 `Text.Regex.Posix` 工作。一般通过在 `ghci` 进行交互是探索一个模块最方便的办法。

```
ghci> :module +Text.Regex.Posix
```

[scarletsky 注： 如果遇到 `Could not find module `Text.Regex.Posix` 的错误，那么你需要安装对应的包。

cabal 用户可以用 `$ cabal install regex-posix && cabal exec ghci`

stack 用户可以用 `$ stack install regex-posix && stack exec ghci`

]

可能正则表达式匹配函数是我们平时需要使用的唯一的函数，其以中缀运算符 `(=~)` (从 `Perl` 中借鉴) 表示。要克服的第一个障碍是 Haskell 的 regexp 库重度使用了多态。其结果就是，`(=~)` 的类型签名非常难懂，所以我们在此对其不做解释。

=~ 操作符的参数和返回值都使用了类型类。第一个参数 (=~ 左侧) 是要被匹配的文本；第二个参数 (=~ 右侧) 是准备匹配的正则表达式。对每个参数我们都可以使用 `String` 或者 `ByteString` 。

8.3.1 结果的多种类型

=~ 操作符的返回类型是多态的，所以 Haskell 编译器需要一通过一些途径知道我们想获得哪种类型的结果。实际编码中，可以通过我们如何使用匹配结果推导出它的类型。但是当我们通过 `ghci` 进行探索时，缺少类型推导的线索。如果不指明匹配结果的类型，ghci 将因其无法获得足够信息对匹配结果进行类型推导而报错。

当 `ghci` 无法推断目标的类型时，我们要告诉它想要哪种类型。若想知道正则匹配是否通过时，需要将结果类型指定为 `Bool` 型。

```
ghci> "my left foot" =~ "foo" :: Bool
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
True
ghci> "your right hand" =~ "bar" :: Bool
False
ghci> "your right hand" =~ "(hand|foot)" :: Bool
True
```

在 `regexp` 库内部，有一种类型类名为 `RegexContext` ，其描述了目标类型的行为。基础库定义了很多这个类型类的实例。`Bool` 型是这种类型类的一个实例，所以我们取回了一个可用的结果. 另一个实例是 `Int` ，可以描述正则表达式匹配了多少次。

```
ghci> "a star called henry" =~ "planet" :: Int
0
ghci> "honorificabilitudinitatibus" =~ "[aeiou]" :: Int
13
```

如果指定结果类型为 `String` ，将得到第一个匹配的子串，或者表示无匹配的空字符串。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: String
"ii"
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: String
""
```

另一个合法的返回值类型是 `[[Srtring]]`，将返回由所有匹配的的字符串组成的列表。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [[String]]
[["ii","ii"],["uu","uu"]]
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [[String]]
[]
```

Warning: 注意 String 类型的结果

指定结果为普通的字符串时，要当心。因为 (=~) 在表示“无匹配”时会返回空字符串，很明显这导致了难以处理可以匹配空字符串的正则表达式。这情况出现时，就需要使用另一种不同的结果类型，比如 [[String]]。

以上是一些“简单”的结果类型，不过还没说完。在继续讲解之前，我们先来定义一个在之后的例子中共同使用的模式串。可以在 ghci 中将这个模式串定义为一个变量，以便节省一些输入操作。

```
ghci> let pat = "(foo[a-z]*bar|quux)"
```

当模式匹配了字符串时，可以获取很多关于上下文的信息。如果指定 `(String,String,String)` 类型的元组作为结果类型，可以获取字符串中首次匹配之前的部分，首次匹配的子串，和首次匹配之后的部分。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String)
("before ","foodiebar"," after")
```

若匹配失败，整个字符串会作为“首次匹配之前”的部分返回，元组的其他两个元素将为空字符串。

```
ghci> "no match here" =~ pat :: (String,String,String)
("no match here","","")
```

使用四元组作为返回结果时，元组的第四个元素是一个包含了模式中所有分组的列表。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String,[String])
("before ","foodiebar"," after",["foodiebar"])
```

也可以获得关于匹配结果的数字信息。二元组类型的结果可以表示首次匹配在字符串中的偏移，以及匹配结果的长度。如果使用由这种二元组构成的列表作为结果类型，我们将得到所有字符串中所有匹配的此类信息。

```
ghci> "before foodiebar after" =~ pat :: (Int,Int)
(7,9)
ghci> getAllMatches  ("i foobarbar a quux" =~ pat) :: [(Int,Int)]
[(2,9),(14,4)]
```

二元组的首个元素（表示偏移的那个），其值为 -1 时，表示匹配失败。当指定返回值为列表时，空表表示失败。

```
ghci> "eleemosynary" =~ pat :: (Int,Int)
(-1,0)
ghci> getAllMatches ("mondegreen" =~ pat) :: [(Int,Int)]
[]
```

以上并非 RegexContext 类型类的内置实例的完整清单。完整的清单可以在 Text.Regex.Base.Context 模块的文档中找到。

使函数具有多态返回值的能力对于一个静态类型语言来说是个不同寻常的特性。

8.4 进一步了解正则表达式

8.4.1 不同类型字符串的混合与匹配

之前提到过，=~ 操作符的输入和返回值都使用了类型类。我们可以在正则表达式和要匹配的文本中使用 String 或者严格的 ByteString 类型。

```
ghci> :module +Data.ByteString.Char8
ghci> :type pack "foo"
pack "foo" :: ByteString
```

我们可以尝试不同的 String 和 ByteString 组合。

```
ghci> pack "foo" =~ "bar" :: Bool
False
ghci> "foo" =~ pack "bar" :: Int
0
ghci> getAllMatches (pack "foo" =~ pack "o") :: [(Int, Int)]
[(1,1),(2,1)]
```

不过，我们需要注意，文本匹配的结果必须与被匹配的字符串类型一致。让我们实践一下，看这是什么意思。

```
ghci> packChars "good food" =~ ".ood" :: [[ByteString]]
[["good"],["food"]]
```

上面的例子中，我们使用 packChars 将一个 String 转换为 ByteString 。这种情况可以通过类型检查，因为 ByteString 也是一种合法的结果类型。但是如果输入字符串类型为 String 类型，在尝试获得 ByteString 类型结果时将会失败。

[译注：原文中使用的 pack 会出现类型错误，新代码中需要使用 Data.ByteString.Internal 模块中的 packChars]

```
ghci> "good food" =~ ".ood" :: [[ByteString]]

<interactive>:55:13:
    No instance for (RegexContext Regex [Char] [[ByteString]])
      arising from a use of `=~'
    In the expression: "good food" =~ ".ood" :: [[ByteString]]
    In an equation for `it' :
        it = "good food" =~ ".ood" :: [[ByteString]]
```

将结果类型指定为与被匹配字符串相同的 String 类型就可以轻松地解决这个问题。

```
ghci> "good food" =~ ".ood" :: [[String]]
[["good"],["food"]]
```

对于正则表达式不存在这个限制。正则表达式可以是 String 或 ByteString ，而不必在意输入或结果是何种类型。

8.4.2 你要知道的其他一些事情

查阅 Haskell 的库文档，会发现很多和正则表达式有关的模块。Text.Regex.Base 下的模块定义了供其他所有正则表达式库使用的通用 API 。可以同时安装许多不同实现的正则表达式模块。写作本书时，GHC 自带一个实现，Text.Regex.Posix 。正如其名字，这个模块提供了 POSIX 语义的正则表达式实现。

Note: Perl 风格和 POSIX 风格的正则表达式

如果你此前用过其他语言，如 Perl，Python，或 Java，并且使用过其中的正则表达式，你应该知道 Text.Regex.Posix 模块处理的 POSIX 风格的正则表达式与 Perl 风格的正则表达式有一些显著的不同。

当有多个匹配结果候选时，Perl 的正则表达式引擎表现为左侧最小匹配，而 POSIX 引擎会选择贪婪匹配（最长匹配）。当使用正则表达式 (foolfo*) 匹配字符串 fooooooo 时，Perl 风格引擎将返回 foo (最左的匹配)，而 POSIX 引擎将返回的结果将包含整个字符串 (贪婪匹配)。

POSIX 正则表达式比 Perl 风格的正则表达式缺少一些格式语法。它们也缺少一些 Perl 风格正则表达式的功能，比如零宽度断言和对贪婪匹配的控制。

Hackage 上也有其他 Haskell 正则表达式包可供下载。其中一些比内置的 POSIX 引擎拥有更好的执行效率 (如 regex-tdfa); 另外一些提供了大多数程序员熟悉的 Perl 风格正则匹配 (如 regex-pcre)。它们都按照我们这节提

到的 API 编写。

8.5 将 glob 模式翻译为正则表达式

我们已经看到了用正则表达式匹配文本的多种方法，现在让我们将注意力回到 glob 模式。我们要编写一个函数，接收一个 glob 模式作为输入，返回其对应的正则表达式。glob 模式和正则表达式都以文本字符串表示，所以这个函数的类型应该已经清楚了。

```
-- file: ch08/GlobRegex.hs
module GlobRegex
    (
      globToRegex
    , matchesGlob
    ) where

import Text.Regex.Posix ((=~))

globToRegex :: String -> String
```

我们生成的正则表达式必须被锚定，所以它要对一个字符串从头到尾完整匹配。

```
-- file: ch08/GlobRegex.hs
globToRegex cs = '^' : globToRegex' cs ++ "$"
```

回想一下，`String` 仅是 `[Char]` 的同义词，一个由字符组成的数组。`:` 操作符将一个值加入某个列表头部，此处是将字符 `^` 加入 `globToRegex'` 函数返回的列表头部。

Note: 在定义之前使用一个值

Haskell 在使用某个值或函数时，并不需要其在之前的源码中被声明。在某个值首次被使用之后才定义它是很平常的。Haskell 编译器并不关心这个层面上的顺序。这使我们可以用最符合逻辑的方式灵活地组织代码，而不是为使编译器作者更轻松而遵守某种顺序。

Haskell 模块的作者们经常利用这种灵活性，将“更重要的”代码放在源码文件更靠前的位置，将繁琐的实现放在后面。这也是我们实现 `globToRegex'` 函数及其辅助函数的方法。

`globToRegex'` 将使用正则表达式做大部分的翻译工作。我们将使用 Haskell 的模式匹配特性轻松地穷举出需要处理的每一种情况

```
-- file: ch08/GlobRegex.hs

globToRegex' :: String -> String
```

(continues on next page)

(continued from previous page)

```
globToRegex' "" = ""

globToRegex' ('*':cs) = ".*" ++ globToRegex' cs

globToRegex' ('?':cs) = '.' : globToRegex' cs

globToRegex' ('[':'!':c:cs) = "[^" ++ c : charClass cs
globToRegex' ('[':c:cs)     = '['  :  c : charClass cs
globToRegex' ('[':_)        = error "unterminated character class"

globToRegex' (c:cs) = escape c ++ globToRegex' cs
```

我们的第一条规则是，如果触及 glob 模式的尾部（也就是说当输入为空字符串时），我们返回 $ ，正则表达式中表示“匹配行尾”的符号。我们按照这样一系列规则将模式串由 glob 语法转化为正则表达式语法。最后一条规则匹配所有字符，首先将可转义字符进行转义。

`escape` 函数确保正则表达式引擎不会将普通字符串解释为构成正则表达式语法的字符。

```
-- file: ch08/GlobRegex.hs
escape :: Char -> String
escape c | c `elem` regexChars = '\\' : [c]
         | otherwise = [c]
    where regexChars = "\\+()^$.{}]|"
```

`charClass` 辅助函数仅检查一个字符类是否正确地结束。这个并不改变其输入，直到遇到一个] 字符，其将控制流交还给 `globToRegex'`

```
-- file: ch08/GlobRegex.hs
charClass :: String -> String
charClass (']':cs) = ']' : globToRegex' cs
charClass (c:cs)   = c : charClass cs
charClass []       = error "unterminated character class"
```

现在我们已经完成了 `globToRegex` 函数及其辅助函数的定义，让我们在 ghci 中装载并且实验一下。

```
ghci> :load GlobRegex.hs
[1 of 1] Compiling GlobRegex        ( GlobRegex.hs, interpreted )
Ok, modules loaded: GlobRegex.
ghci> :module +Text.Regex.Posix
ghci> globToRegex "f??.c"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
```

(continues on next page)

(continued from previous page)

```
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
"^f..\\.c$"
```

果然，看上去像是一个合理的正则表达式。可以使用她来匹配某个字符串码?

```
ghci> "foo.c" =~ globToRegex "f??.c" :: Bool
True
ghci> "test.c" =~ globToRegex "t[ea]s*" :: Bool
True
ghci> "taste.txt" =~ globToRegex "t[ea]s*" :: Bool
True
```

奏效了！现在让我们在 ghci 里玩耍一下。我们可以临时定义一个 `fnmatch` 函数，并且试用它。

```
ghci> let fnmatch pat name  =  name =~ globToRegex pat :: Bool
ghci> :type fnmatch
fnmatch :: (RegexLike Regex source1) => String -> source1 -> Bool
ghci> fnmatch "d*" "myname"
False
```

[译注：在 ghc 7.10.1+ 中需要使用 `FlexibleContexts` 扩展，即 `ghci -XFlexibleContexts` ，否则定义 `fnmatch` 时会报错，也可以给 `name` 指定一个特定类型如 `String`]

但是 `fnmatch` 没有真正的 “Haskell 味道”。目前为止，最常见的 Haskell 风格是赋予函数具有描述性的，“驼峰式” 命名。将单词连接为驼峰状，首字母小写后面每个单词的首字母大写。例如，“file name matches” 这几个词将转换为 `fileNameMatch` 这个名字。“驼峰式” 这种说法来自与大写字母形成的 “驼峰”。在我们的库中，将使用 `matchesGlob` 这个函数名。

```
-- file: ch08/GlobRegex.hs
matchesGlob :: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat
```

你可能注意到目前为止我们使用的都是短变量名。从经验来看，描述性的名字在更长的函数定义中更有用，它们有助于可读性。对一个仅有两行的函数来说，长变量名价值较小。

8.5.1 练习

1. 使用 `ghci` 探索当你向 `globToRegex` 传入一个畸形的模式时会发生什么，如 “[”。编写一个小函数调用 `globToRegex` ，向其传入一个畸形的模式。发生了什么?
2. Unix 的文件系统的文件名通常是对大小写敏感的（如:” G” 和 “g” 不同），Windows 文件系统则不是。为 `globToRegex` 和 `matchesGlob` 函数添加一个参数，以控制它们是否大小写敏感。

8.6 重要的题外话：编写惰性函数

在命令式语言中，globToRegex 通常是个被我们写成循环的函数。举个例子，Python 标准库中的 fnmatch 模块包括了一个名叫 translate 的函数与我们的 globToRegex 函数做了完全相同的工作。它就被写成一个循环。

如果你了解过函数式编程语言比如 Scheme 或 ML，可能有个概念已经深入你的脑海，“模拟一个循环的方法是使用尾递归”。

观察 globToRegex'，可以发现其不是一个尾递归函数。至于原因，重新检查一下它的最后一组规则（它的其他规则也类似）。

```
-- file: ch08/GlobRegex.hs
globToRegex' (c:cs) = escape c ++ globToRegex' cs
```

其递归地执行自身，并以递归执行的结果作为 (++) 函数的参数。因为递归执行并不是这个函数的最后一个操作，所以 globToRegex' 不是尾递归函数。

为何我们的函数没有定义成尾递归的？答案是 Haskell 的非严格求值策略。在我们开始讨论它之前，先快速的了解一下为什么，传统编程语言中，这类递归定义是我们要避免的。这里有一个简化的（++）操作符定义。它是递归的，但不是尾递归的。

```
-- file: ch08/append.hs
(++) :: [a] -> [a] -> [a]

(x:xs) ++ ys = x : (xs ++ ys)
[]     ++ ys = ys
```

在严格求值语言中，如果我们执行“foo” ++ “bar”，将马上构建并返回整个列表。非严格求值将这项工作延后很久执行，直到其结果在某处被用到。

如果我们需要“foo” ++ “bar”这个表达式结果中的一个元素，函数定义中的第一个模式被匹配，返回表达式 x : (xs ++ ys)。因为 (:) 构造器是非严格的，xs ++ ys 的求值被延迟到当我们需要生成更多结果中的元素时。当生成了结果中的更多元素，我们不再需要 x ，垃圾收集器可以将其回收。因为我们按需要计算结果中的元素，且不保留已经计算出的结果，编译器可以用常数空间对我们的代码求值。

8.7 利用我们的模式匹配器

有一个函数可以匹配 glob 模式很好，但我们希望可以在实际中使用它。在类 Unix 系统中，glob 函数返回一个由匹配给定 glob 模式串的文件和目录组成的列表。让我们用 Haskell 构造一个类似的函数。按 Haskell 的描述性命名规范，我们将这个函数称为 namesMatching 。

```
-- file: ch08/Glob.hs
module Glob (namesMatching) where
```

我们将 namesMatching 指定为我们的 Glob 模块中唯一对用户可见的名字。

```
-- file: ch08/Glob.hs
import System.Directory (doesDirectoryExist, doesFileExist,
                         getCurrentDirectory, getDirectoryContents)
```

System.FilePath 抽象了操作系统路径名称的惯例。(</>) 函数将两个部分组合为一个路径。

```
ghci> :m +System.FilePath
ghci> "foo" </> "bar"
Loading package filepath-1.1.0.0 ... linking ... done.
"foo/bar"
```

dropTrailingPathSeparator 函数的名字完美地描述了其作用。

```
ghci> dropTrailingPathSeparator "foo/"
"foo"
```

splitFileName 函数以路径中的最后一个斜线将路径分割为两部分。

```
ghci> splitFileName "foo/bar/Quux.hs"
("foo/bar/","Quux.hs")
ghci> splitFileName "zippity"
("","zippity")
```

配合 Systems.FilePath 和 Systems.Directory 两个模块，我们可以编写一个在类 Unix 和 Windows 系统上都可以运行的可移植的 namesMatching 函数。

```
-- file: ch08/Glob.hs
import System.FilePath (dropTrailingPathSeparator, splitFileName, (</>))
```

在这个模块中，我们将模拟一个“for”循环；首次尝试在 Haskell 中处理异常；当然还会用到我们刚写的 matchesGlob 函数。

```
-- file: ch08/Glob.hs
import Control.Exception (handle, SomeException)
import Control.Monad (forM)
import GlobRegex (matchesGlob)
```

目录和文件存在于各种带有副作用的活动的“真实世界”，我们的 glob 模式处理函数的返回值类型中将必须带有 IO 。

如果的输入字符串中不包含模式字符，我们简单的在文件系统中检查输入的名字是否已经建立。（注意，此处使用 Haskell 的 guard 语法可以编写精细整齐的定义。“if” 语句也可以做到，但是在美学上不能令人满意。）

```
-- file: ch08/Glob.hs
isPattern :: String -> Bool
isPattern = any (`elem` "[?*")

namesMatching pat
  | not (isPattern pat) = do
    exists <- doesNameExist pat
    return (if exists then [pat] else [])
```

doesNameExist 是一个我们将要简要定义的函数的名字。

如果字符串是一个 glob 模式呢？继续定义我们的函数。

```
-- file: ch08/Glob.hs
  | otherwise = do
    case splitFileName pat of
      ("", baseName) -> do
          curDir <- getCurrentDirectory
          listMatches curDir baseName
      (dirName, baseName) -> do
          dirs <- if isPattern dirName
                  then namesMatching (dropTrailingPathSeparator dirName)
                  else return [dirName]
          let listDir = if isPattern baseName
                        then listMatches
                        else listPlain
          pathNames <- forM dirs $ \dir -> do
                           baseNames <- listDir dir baseName
                           return (map (dir </>) baseNames)
          return (concat pathNames)
```

我们使用 splitFileName 将字符串分割为目录名和文件名。如果第一个元素为空，说明我们正在当前目录寻找符合模式的文件。否则，我们必须检查目录名，观察其是否包含模式。若不含模式，我们建立一个只由目录名一个元素组成的列表。如果含有模式，我们列出所有匹配的目录。

Note: 注意事项

System.FilePath 模块稍有点诡异。上面的情况就是一个例子。splitFileName 函数在其返回值的目录名部分的结尾保留了一个斜线。

```
ghci> :module +System.FilePath
ghci> splitFileName "foo/bar"
```

(continues on next page)

(continued from previous page)

```
Loading package filepath-1.1.0.0 ... linking ... done.
("foo/","bar")
```

如果忘记（或不够了解）要去掉这个斜线，我们将在 namesMatching 函数中进行无止尽的递归匹配，看看后面演示的 splitFileName 的行为你就会明白。

```
ghci> splitFileName "foo/"
("foo/","")
```

你或许能够想象是什么促使我们加入这份注意事项。

最终，我们将每个目录中的匹配收集起来，得到一个由列表组成的列表，然后将它们连接为一个单独的由文件名组成的列表。

上面那个函数中出现的陌生的 forM 函数，其行为有些像“for”循环：它将其第二个参数（一个动作）映射到其第一个参数（一个列表），并返回由其结果组成的列表。

我们还剩余一些零散的目标需要完成。首先是上面用到过的 doesNameExist 函数的定义。System.Directory 函数无法检查一个名字是否已经在文件系统中建立。它强制我们明确要检查的是一个文件还是目录。这个 API 设计的很丑陋，所以我们必须在一个函数中完成两次检验。出于效率考虑，我们首先检查文件名，因为文件比目录更常见。

```
-- file: ch08/Glob.hs
doesNameExist :: FilePath -> IO Bool

doesNameExist name = do
    fileExists <- doesFileExist name
    if fileExists
      then return True
      else doesDirectoryExist name
```

还有两个函数需要定义，返回值都是由某个目录下的名字组成的列表。listMatches 函数返回由某目录下全部匹配给定 glob 模式的文件名组成的列表。

```
-- file: ch08/Glob.hs
listMatches :: FilePath -> String -> IO [String]
listMatches dirName pat = do
    dirName' <- if null dirName
                then getCurrentDirectory
                else return dirName
    handle (const (return []):: (SomeException->IO [String]))
           $ do names <- getDirectoryContents dirName'
                let names' = if isHidden pat
```

(continues on next page)

(continued from previous page)

```
                            then filter isHidden names
                            else filter (not . isHidden) names
                return (filter (`matchesGlob` pat) names')

isHidden ('.':_) = True
isHidden _       = False
```

listPlain 接收的函数名若存在，则返回由这个文件名组成的单元素列表，否则返回空列表。

```
-- file: ch08/Glob.hs
listPlain :: FilePath -> String -> IO [String]
listPlain dirName baseName = do
    exists <- if null baseName
              then doesDirectoryExist dirName
              else doesNameExist (dirName </> baseName)
    return (if exists then [baseName] else [])
```

仔细观察 listMatches 函数的定义，将发现一个名为 handle 的函数。之前，我们从 Control.Exception 模块中将其载入。正如其暗示的那样，这个函数让我们初次体验了 Haskell 中的异常处理。把它扔进 ghci 中看我们会发现什么。

```
ghci> :module +Control.Exception
ghci> :type handle
handle :: (Exception -> IO a) -> IO a -> IO a
```

可以看出 handle 接受两个参数。首先是一个函数，其接受一个异常值，且有副作用（其返回值类型带有 IO 标签）；这是一个异常处理器。第二个参数是可能会抛出异常的代码。

关于异常处理器，异常处理器的类型限制其必须返回与抛出异常的代码相同的类型。所以它只能选择或是抛出一个异常，或像在我们的例子中返回一个由字符串组成的列表。

const 函数接受两个参数；无论第二个参数是什么，其始终返回第一个参数。

```
ghci> :type const
const :: a -> b -> a
ghci> :type return []
return [] :: (Monad m) => m [a]
ghci> :type handle (const (return []))
handle (const (return [])) :: IO [a] -> IO [a]
```

我们使用 const 编写异常处理器忽略任何向其传入的异常。取而代之，当我们捕获异常时，返回一个空列表。

本章不会再展开任何异常处理相关的话题。然而还有更多可说，我们将在第 19 章异常处理时重新探讨这个主题。

8.7.1 练习

1. 尽管我们已经编写了一个可移植 `namesMatching` 函数，这个函数使用了我们的大小写敏感的 `globToRegex` 函数。尝试在不改变其类型签名的前提下，使 `namesMatching` 在 Unix 下大小写敏感，在 Windows 下大小写不敏感。

提示：查阅一下 `System.FilePath` 的文档，其中有一个变量可以告诉我们程序是运行在类 Unix 系统上还是在 Windows 系统上。

2. 如果你在使用类 Unix 系统，查阅 `System.Posix.Files` 模块的文档，看是否能找到一个 `doesNameExist` 的替代品。

3. `*` 通配符，仅匹配一个单独目录中的名字。很多 shell 可以提供扩展通配符语法，`**`，其将在所有目录中进行递归匹配。举个例子，`**.c` 意为“在当前目录及其任意深度的子目录下匹配一个 `.c` 结尾的文件名”。实现 `**` 通配符匹配。

8.8 通过 API 设计进行错误处理

向 `globToRegex` 传入一个畸形的正则表达式未必会是一场灾难。用户的表达式可能会有输入错误，这时我们更希望得到有意义的报错信息。

当这类问题出现时，调用 `error` 函数会有很激烈的反应（其结果在 Q: 1 这个练习中探索过。）。`error` 函数会抛出一个异常。纯函数式的 `Haskell` 代码无法处理异常，所以控制流会突破我们的纯函数代码直接交给处于距离最近一层 `IO` 中并且安装有合适的异常处理器的调用者。如果没有安装异常处理器，`Haskell` 运行时的默认动作是终结我们的程序（如果是在 ghci 中，则会打出一条令人不快的错误信息。）

所以，调用 `error` 有点像是拉下了战斗机的座椅弹射手柄。我们从一个无法优雅处理的灾难性场景中逃离，而等我们着地时会撒出很多燃烧着的残骸。

我们已经确定了 `error` 是为灾难情场景准备的，但我们仍旧在 `globToRegex` 中使用它。畸形的输入将被拒绝，但不会导致大问题。处理这种情况有更好的方式吗?

`Haskell` 的类型系统和库来救你了！我们可以使用内置的 `Either` 类型，在 `globToRegex` 函数的类型签名中描述失败的可能性。

```
-- file: ch08/GlobRegexEither.hs
type GlobError = String

globToRegex :: String -> Either GlobError String
```

`globToRegex` 的返回值将为两种情况之一，或者为 `Left " 出错信息"` 或者为 `Right " 一个合法正则表达式"`。这种返回值类型，强制我们的调用者处理可能出现的错误。(你会发现这是 `Haskell` 代码中 `Either` 类型最广泛的用途。)

8.8.1 练习

1. 编写一个使用上面那种类型签名的 `globToRegex` 版本。
2. 改变 `namesMatching` 的类型签名，使其可以处理畸形的正则表达式，并使用它重写 `globToRegex` 函数。

Tip: 你会发现牵扯到的工作量大得惊人。别怕，我们将在后面的章节介绍更多简单老练的处理错误的方式。

8.9 让我们的代码工作

`namesMatching` 函数本身并不是很令人兴奋，但它是一个很有用的构建模块。将它与稍多点的函数组合在一起，就会让我们做出有趣的东西。

这里有个例子。定义一个 `renameWith` 函数，并不简单的重命名一个文件，取而代之，对文件名执行一个函数，并将返回值作为新的文件名。

```
-- file: ch08/Useful.hs
import System.FilePath (replaceExtension)
import System.Directory (doesFileExist, renameDirectory, renameFile)
import Glob (namesMatching)

renameWith :: (FilePath -> FilePath)
           -> FilePath
           -> IO FilePath

renameWith f path = do
    let path' = f path
    rename path path'
    return path'
```

我们再一次通过一个辅助函数使用 `System.Directory` 中难看的文件/目录函数

```
-- file: ch08/Useful.hs
rename :: FilePath -> FilePath -> IO ()

rename old new = do
    isFile <- doesFileExist old
    let f = if isFile then renameFile else renameDirectory
    f old new
```

`System.FilePath` 模块提供了很多有用的函数用于操作文件名。这些函数洽好漏过了我们的 `renameWith`

和 namesMatching 函数，所以我们可以通过将他们组合起来的方式来快速的创建新函数。例如，这个简洁的函数修改了 C++ 源码文件的后缀名。

```
-- file: ch08/Useful.hs
cc2cpp =
  mapM (renameWith (flip replaceExtension ".cpp")) =<< namesMatching "*.cc"
```

cc2cpp 函数使用了几个我们已经见过多次的函数。flip 函数接受另一个函数作为参数，交换其参数的顺序（可以在 ghci 中调查 replaceExtension 的类型以了解详情）。=<< 函数将其右侧动作的结果喂给其左侧的动作。

8.9.1 练习

1. Glob 模式解释起来很简单，用 Haskell 可以很容易的直接写出其匹配器，正则表达式则不然。试一下编写正则匹配。

CHAPTER

NINE

第 9 章：I/O 学习——构建一个用于搜索文件系统的库

自从电脑有了分层文件系统以来，“我知道有这个文件，但不知道它放在哪”这个问题就一直困扰着人们。1974 年发布的 Unix 第五个版本引入的 `find` 命令，到今天仍在使用。查找文件的艺术已经走过了很长一段路：伴随现代操作系统一起不断发展的文件索引和搜索功能。

给程序员的工具箱里添加类似 `find` 这样的功能依旧非常有价值，在本章，我们将通过编写一个 Haskell 库给我们的 `find` 命令添加更多功能，我们将通过一些有着不同的健壮度的方法来完成这个库。

9.1 find 命令

如果你不曾用过类 Unix 的系统，或者你不是个重度 shell 用户，那么你很可能从未听说过 `find`，通过给定的一组目录，它递归搜索每个目录并且打印出每个匹配表达式的目录项名称。

```
-- file: ch09/RecursiveContents.hs
module RecursiveContents (getRecursiveContents) where

import Control.Monad (forM)
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))

getRecursiveContents :: FilePath -> IO [FilePath]

getRecursiveContents topdir = do
  names <- getDirectoryContents topdir
  let properNames = filter (`notElem` [".", ".."]) names
  paths <- forM properNames $ \name -> do
    let path = topdir </> name
    isDirectory <- doesDirectoryExist path
    if isDirectory
      then getRecursiveContents path
      else return [path]
  return (concat paths)
```

单个表达式可以识别像“符合这个全局模式的名称”，“目录项是一个文件”，“当前最后一个被修改的文件”以及其他诸如此类的表达式，通过 and 或 or 算子就可以把他们装配起来构成更加复杂的表达式

9.2 简单的开始：递归遍历目录

在投入设计我们的库之前，先解决一些规模稍小的问题，我们第一个问题就是递归地列出一个目录下面的所有内容和它的子目录

`filter` 表达式确保一个目录的列表不含特定的目录名（比如代表当前目录的 `.` 和上一级目录的 `..`），如果忘记过滤这些，随后的查找将陷入无限循环。

我们在之前的章节里完成了 `forM` 函数，它是参数颠倒后的 `mapM` 函数。

```
ghci> :m +Control.Monad
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type forM
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
```

循环体将检查当前目录项是否为目录，如果是，则递归调用 `getrecuresivacontents` 函数列出这个目录（的内容），如果否，则返回只含有当前目录项名字的单元素列表，不要忘记 `return` 函数在 Haskell 中有特殊的含义，他通过 monad 的类型构造器包装了一个值。

另一个值得注意的地方是变量 `isDirectory` 的使用，在命令式语言如 Python 中，我们通常用 `if os.path.isdir(path)` 来表示，然而，`doesDirectoryExist` 函数是一个动作，它的返回类型是 `IO Bool` 而非 `Bool` ，由于 `if` 表达式需要一个操作值为 `bool` 的表达式作为条件，我们使用 `<-` 来从 io 包装器上得到这个动作的的 `bool` 返回值，这样我们就能在 `if` 中使用这个干净的无包装的 `bool` 。

循环体中每一次迭代生成的结果都是名称列表,因此 `forM` 的结果是 `IO [[FilePath]]` ,我们通过 `concat` 将它转换为一个元素列表 (从以列表为元素的列表转换为不含列表元素的列表)

9.2.1 再次认识匿名和命名函数

在 Anonymous (lambda) functions 这部分，我们列举了一系列不使用匿名函数的原因，然而在这里，我们将使用它作为函数体，这是匿名函数在 Haskell 中最常见的用途之一。

我们已经在 `forM` 和 `mapM` 上看到使用函数作为参数的方式，许多循环体是程序中只出现一次的代码块。既然我们喜欢在循环中使用一个再也不会出现的循环体，那么为什么要给他们命名?

显而易见，有时候我们需要在不同的循环中嵌入相同的代码，这时候我们不应该使用匿名函数，把他们剪贴和复制进去，而是给这些匿名函数命名来调用，这样显得有意义一点

9.2.2 为什么提供 mapM 和 forM

存在两个相同的函数看起来是有点奇怪，但接受参数的顺序之间的差异使他们适用于不同的情况。

我们来考察下之前的例子，使用匿名函数作为循环体，如果我们使用 mapM 而非 forM ，我们将不得不把变量 properNames 放置到函数体的后边，而为了让代码正确解析，我们就必须将整个匿名函数用括号包起来，或者用一个不必要的命名函数将它取代，自己尝试下，拷贝上边的代码，用 mapM 代替 forM ，观察代码可读性上有什么变化

相反，如果循环体是一个命名函数，而且我们要循环的列表是通过一个复杂表达式计算的，我们就找到了 mapM 的应用场景

这里需要遵守的代码风格是无论通过 mapM 和 forM 都让你写出干净的代码，如果循环体或者循环中的表达式都很短，那么用哪个都无所谓，如果循环体很短，但数据很长，使用 mapM ，如果相反，则用 forM ，如果都很长，使用 let 或者 where 让其中一个变短，通过这样一些实践，不同情况下那个实现最好就变得显而易见

9.3 一个本地查找函数

我们可以使用 getRecursiveContents 函数作为一个内置的简单文件查找器的基础

```
-- file: ch09/SimpleFinder.hs
import RecursiveContents (getRecursiveContents)
simpleFind :: (FilePath -> Bool) -> FilePath -> IO [FilePath]
simpleFind p path = do
    names <- getRecursiveContents path
    return (filter p names)
```

上文的函数通过我们在过滤器中的谓词来匹配 getRecursiveContents 函数返回的名字，每个通过谓词判断的名称都是文件全路径，因此如何完成一个像“查找所有扩展名以 .c 结尾的文件”的功能?

System.FilePath 模块包含了许多有价值的函数来帮助我们操作文件名，在这个例子中，我们使用 takeExtension：

```
ghci> :m +System.FilePath
ghci> :type takeExtension
takeExtension :: FilePath -> String
ghci> takeExtension "foo/bar.c"
Loading package filepath-1.1.0.0 ... linking ... done.
".c"
ghci> takeExtension "quux"
""
```

下面的代码给我们一个包括获得路径，获得扩展名，然后和.c 进行比较的简单功能的函数实现，

```
ghci> :load SimpleFinder
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( SimpleFinder.hs, interpreted )
Ok, modules loaded: RecursiveContents, Main.
ghci> :type simpleFind (\p -> takeExtension p == ".c")
simpleFind (\p -> takeExtension p == ".c") :: FilePath -> IO [FilePath]
```

simpleFind 在工作中有一些非常刺眼的问题，第一个就是谓词并不能准确而完全的表达，他只关注文件夹中的目录项名称，而无法做到辨认这是个文件还是个目录此类的事情，——而我们使用 simpleFind 的尝试就是想列举所有文件和与文件一样拥有 .c 扩展名的文件夹

第二个问题是在 simpleFind 中我们无法控制它遍历文件系统的方式，这是显而易见的，想想在分布式版本控制系统中控制下的树状结构中查找一个源文件的问题吧，所有被控制的目录都含有一个 .svn 的私有文件夹，每一个包含了许多我们毫不感兴趣的子文件夹和文件，简单的过滤所有包含 .svn 的路径远比仅仅在读取时避免遍历这些文件夹更加有效。例如，一个分布式源码树包含了 45000 个文件，30000 个分布在 1200 个不同的.svn 文件夹中，避免遍历这 1200 个文件夹比过滤他们包含的 30000 个文件代价更低。

最后。simpleFind 是严格的，因为它包含一系列 IO Monad 操作执行构成的动作，如果我们有一百万个文件要遍历，我们需要等待很长一段时间才能得到一个包含一百万个名字的巨大的返回值，这对用户体验和资源消耗都是噩梦，我们更需要一个只有当他们获得结果的时才展示的结果流。

在接下来的环节里，我们将解决每个遇到的问题

9.4 谓词在保持纯粹的同时支持从贫类型到富类型

我们的谓词只关注文件名，这将一系列有趣的操作排除在外，试想下，假如我们希望列出比某个给定值更大的文件呢?

面对这个问题的第一反应是查找 IO：我们的谓词是 FilePath -> Bool 类型，为什么不把它变成 FilePath -> IO Bool 类型? 这将使我们所有的 IO 操作都成为谓词的一部分，但这在显而易见的好处之外引入一个潜在的问题，使用这样一个谓词存在各种可能的后果，比如一个有 IO a 类型返回的函数将有能力生成任何它想产生的结果。

让我们在类型系统中寻找以写出拥有更多谓词，更少 bug 的代码，我们通过避免污染 IO 来坚持谓词的纯粹，这将确保他们不会产生任何不纯的结果，同时我们给他们提供更多信息，这样他们就可以在不必诱发潜在的危险的情况下获得需要的表达式

Haskell 的 System.Directory 模块提供了一个尽管受限但仍然有用的关于文件元数据的集合

```
ghci> :m +System.Directory
```

我们可以通过 doesFileExist 和 doesDirectoryExist 来判断目录项是目录还是文件，但暂时还没有更多方式来查找这些年里出现的纷繁复杂的其他文件类型，比如管道，硬链接和软连接。

```
ghci> :type doesFileExist
doesFileExist :: FilePath -> IO Bool
ghci> doesFileExist "."
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
False
ghci> :type doesDirectoryExist
doesDirectoryExist :: FilePath -> IO Bool
ghci> doesDirectoryExist "."
True
```

getPermissions 函数让我们确定当前对于文件或目录的操作是否是合法：

```
ghci> :type getPermissions
getPermissions :: FilePath -> IO Permissions
ghci> :info Permissions
data Permissions
  = Permissions {readable :: Bool,
                 writable :: Bool,
                 executable :: Bool,
                 searchable :: Bool}
      -- Defined in System.Directory
instance Eq Permissions -- Defined in System.Directory
instance Ord Permissions -- Defined in System.Directory
instance Read Permissions -- Defined in System.Directory
instance Show Permissions -- Defined in System.Directory
ghci> getPermissions "."
Permissions {readable = True, writable = True, executable = False, searchable = True}
ghci> :type searchable
searchable :: Permissions -> Bool
ghci> searchable it
True
```

如果你无法回忆起 ghci 中变量 it 的特殊用法，回到第一章复习一下，如果我们的权限能够列出它的内容，那么这个目录就应该是可被搜索的，而文件则永远是不可搜索的

最后，getModificationTime 告诉我们目录项上次被修改的时间：

```
ghci> :type getModificationTime
getModificationTime :: FilePath -> IO System.Time.ClockTime
ghci> getModificationTime "."
Mon Aug 18 12:08:24 CDT 2008
```

[Forec 译注：在 GHC 7.6 之后，getModificationTime 不再返回 ClockTime 类型，你可以使用 UTCTime

代替：

```
import Data.Time.Clock (UTCTime(..))
```

]

如果我们像标准的 Haskell 代码一样对可移植性要求严格，这些函数就是我们手头所有的一切 (我们同样可以通过黑客手段来获得文件大小)，这些已经足够让我们明白所感兴趣领域中的原则，而非让我们浪费宝贵的时间对着一个例子冥思苦想，如果你需要写满足更多需求的代码，`System.Posix` 和 `System.Win32` 模块提供关于当代两种计算平台的更多文件元数据的细节。`Hackage` 中同样有一个 `unix-compat` 包，提供 windows 下的类 unix 的 api 。

新的富类型谓词需要关注的数据段到底有几个？自从我们可以通过 `Permissions` 来判断目录项是文件还是目录之后，我们就不再需要获得 `doesFileExist` 和 `doesDirectoryExist` 的结果，因此一个谓词需要关注的输入有四个。

```
-- file: ch09/BetterPredicate.hs
import Control.Monad (filterM)
import System.Directory (Permissions(..), getModificationTime, getPermissions)
import System.Time (ClockTime(..))
import System.FilePath (takeExtension)
import Control.Exception (bracket, handle)
import System.IO (IOMode(..), hClose, hFileSize, openFile)

-- the function we wrote earlier
import RecursiveContents (getRecursiveContents)

type Predicate =  FilePath       -- path to directory entry
               -> Permissions    -- permissions
               -> Maybe Integer  -- file size (Nothing if not file)
               -> ClockTime      -- last modified
               -> Bool
```

这一谓词类型只是一个有四个参数的函数的同义词，他将给我们节省一些键盘工作和屏幕空间。

注意这一返回值是 `Bool` 而非 `IO Bool` ，谓词需要保证纯粹，而且不能表现 IO，在拥有这种类型以后，我们的查找函数仍然显得非常整洁。

```
-- file: ch09/BetterPredicate.hs
-- soon to be defined
getFileSize :: FilePath -> IO (Maybe Integer)

betterFind :: Predicate -> FilePath -> IO [FilePath]

betterFind p path = getRecursiveContents path >>= filterM check
```

(continues on next page)

(continued from previous page)

```
    where check name = do
            perms <- getPermissions name
            size <- getFileSize name
            modified <- getModificationTime name
            return (p name perms size modified)
```

先来阅读代码，由于随后将讨论 getFileSize 的某些细节，因此现在暂时先跳过它。

我们无法使用 filter 来调用我们的谓词，因为 p 的纯粹代表他不能作为 IO 收集元数据的方式

这让我们将目光转移到一个并不熟悉的函数 filterM 上，它的动作就像普通的 filter 函数，但在这种情况下，它在 IO monad 操作中使用它的谓词，进而通过谓词表现 IO：

```
ghci> :m +Control.Monad
ghci> :type filterM
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

check 谓词是纯谓词 p 的 IO 功能包装器，替 p 完成了所有 IO 相关的脏活累活，因此我们可以使 p 对副作用免疫，在收集完元数据后，check 调用 p ，通过 return 语句包装 p 的 IO 返回结果

9.5 安全的获得一个文件的大小

即使 System.Directory 不允许我们获得一个文件的大小，我们仍可以使用 System.IO 的类似接口完成这项任务，它包含了一个名为 hFileSize 的函数，这一函数返回打开文件的字节数，下面是他的简单调用实例：

```
-- file: ch09/BetterPredicate.hs
simpleFileSize :: FilePath -> IO Integer

simpleFileSize path = do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return size
```

当这个函数工作时，他还不能完全为我们所用，在 betterFind 中，我们在目录下的任何目录项上调用 getFileSize ，如果目录项不是一个文件或者大小被 Just 包装起来，他应当返回一个空值，而当目录项不是文件或者没有被打开时（可能是由于权限不够），这个函数会抛出一个异常然后返回一个未包装的大小。

下文是安全的用法：

```
-- file: ch09/BetterPredicate.hs
saferFileSize :: FilePath -> IO (Maybe Integer)
```

(continues on next page)

(continued from previous page)

```
saferFileSize path = handle (\_ -> return Nothing) $ do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return (Just size)
```

[Forec 译注：此处在 GHC 7.8.* 中会出现编译错误，如果你想按照原文中的匿名函数格式编写，则需要做如下修改：

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Exception (bracket, handle, SomeException)

getFileSize path = handle (\(_ :: SomeException) -> return Nothing) $
```

]

函数体几乎完全一致，除了 handle 语句。

我们的异常捕捉在忽略通过的异常的同时返回一个空值，函数体唯一的变化就是允许通过 Just 包装文件大小

saferFileSize 函数现在有正确的类型签名，并且不会抛出任何异常，但他扔未能完全的正常工作，存在 openFile 会成功的目录项，但 hFileSize 会抛出异常，这将和被称作命名管道的状况一起发生，这样的异常会被捕捉，但却从未发起调用 hClose 。

当发现不再使用文件句柄，Haskell 会自动关闭它，但这只有在运行垃圾回收时才会执行，如果无法断言，则延迟到下一次垃圾回收。

文件句柄是稀缺资源，稀缺性是通过操作系统强制保证的，在 linux 中，一个进程只能同时拥有 1024 个文件句柄。

不难想象这种场景，程序调用了一个使用 saferFileSize 的 betterFind 函数，在足够的垃圾文件句柄被关闭之前，由于 betterFind 造成文件句柄数耗尽导致了程序崩溃

这是 bug 危害性的一方面：通过合并起来的不同的部分使得 bug 不易被排查，只有在 betterFind 访问足够多的非文件达到进程打开文件句柄数上限的时候才会被触发，随后在积累的垃圾文件句柄被关闭之前返回一个尝试打开另一个文件的调用。

任何程序内由无法获得数据造成的后续错误都会让事情变得更糟，直到垃圾回收为止。修正这样一个 bug 需要程序结构本身支持，文件系统内容，如何关闭当前正在运行的程序以触发垃圾回收

这种问题在开发中很容易被检查出来，然而当他在上线之后出现（这些恶心的问题一向如此），就变得非常难以发觉

幸运的是，我们可以很容易避开这种错误，同时又能缩短我们的函数。

9.5.1 请求-使用-释放循环

每当 openFile 成功之后我们就必须保证调用 hClose ，Control.Exception 模块提供了 bracket 函数来支持这个想法：

```
ghci> :type bracket
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

bracket 函数需要三个动作作为参数，第一个动作需要一个资源，第二个动作释放这个资源，第三个动作在这两个中执行，当资源被请求，我们称他为操作动作，当请求动作成功，释放动作随后总是被调用，这保证了这个资源一直能够被释放，对通过的每个请求资源文件的操作，使用和释放动作都是必要的。

如果一个异常发生在使用过程中，bracket 调用释放动作并抛出异常，如果使用动作成功，bracket 调用释放动作，同时返回使用动作返回的值。

我们现在可以写一个完全安全的函数了，他将不会抛出异常，也不会积累可能在我们程序其他地方制造失败的垃圾文件句柄数。

```
-- file: ch09/BetterPredicate.hs
getFileSize path = handle (\_ -> return Nothing) $
  bracket (openFile path ReadMode) hClose $ \h -> do
    size <- hFileSize h
    return (Just size)
```

仔细观察 bracket 的参数，首先打开文件，并且返回文件句柄，第二步关闭句柄，第三步在句柄上调用 hFileSize 并用 just 包装结果返回

为了这个函数的正常工作，我们需要使用 bracket 和 handle ，前者保证我们不会积累垃圾文件句柄数，后者保证我们免于异常。

练习

1. 调用 bracket 和 handle 的顺序重要吗，为什么

9.6 为谓词而开发的领域特定语言

深入谓词写作的内部，我们的谓词将检查大于 128kb 的 C++ 源文件：

```
-- file: ch09/BetterPredicate.hs
myTest path _ (Just size) _ =
    takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

这并不是令人感到愉快的工作，谓词需要四个参数，并且总是忽略其中的两个，同时需要定义两个等式，写一些更有意义的谓词代码，我们可以做的更好。

有些时候，这种库被用作嵌入式领域特定语言，我们通过编写代码的过程中通过编程语言的本地特性来优雅的解决一些特定问题

第一步是写一个返回当前函数的一个参数的函数，这个从参数中抽取路径并传给谓词：

```
-- file: ch09/BetterPredicate.hs
pathP path _ _ _ = path
```

如果我们不能提供类型签名，Haskell 将给这个函数提供一个通用类型，这在随后会导致一个难以理解的错误信息，因此给 pathP 一个类型：

```
-- file: ch09/BetterPredicate.hs
type InfoP a =  FilePath         -- path to directory entry
             -> Permissions      -- permissions
             -> Maybe Integer    -- file size (Nothing if not file)
             -> ClockTime        -- last modified
             -> a

pathP :: InfoP FilePath
```

我们已经创建了一个可以用做缩写的类型，相似的结构函数，我们的类型代词接受一个类型参数，如此我们可以分辨不同的结果类型：

```
-- file: ch09/BetterPredicate.hs
sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing     _ = -1
```

我们在这里做了些小动作，对那些我们无法打开的文件或者不是文件的东西我们返回的目录项大小是 -1 。

事实上，浏览中可以看出我们在本章开始处定义谓词类型的和 InfoP Bool 一样，因此我们可以合法的放弃谓词类型。

pathP 和 sizeP 的用法？通过一些线索，我们发现可以在一个谓词中使用它们（每个名称中的前缀 p 代表谓词），从这开始事情就变得有趣起来：

```
-- file: ch09/BetterPredicate.hs
equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP f k = \w x y z -> f w x y z == k
```

equalP 的类型签名值得注意，他接受一个 InfoP a ，同时兼容 pathP 和 sizeP ，他接受一个 a ，并返回一个被认为是谓词同义词的 InfoP Bool ，换言之，equalP 构造了一个谓词。

equalP 函数通过返回一个匿名函数工作，谓词接受参数之后将他们转成 f ，并将结果和 f 进行比对。

equalP 的相等强调了这一事实，我们认为它需要两个参数，在 Haskell 柯里化处理了所有函数的情况下，通过这种方式写 equalP 并无必要，我们可以避免匿名函数，同时通过柯里化来写出表现相同的函数：

```
-- file: ch09/BetterPredicate.hs
equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k
```

在继续我们的探险之前，先把写好的模块加载到 ghci 里去：

```
ghci> :load BetterPredicate
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( BetterPredicate.hs, interpreted )
Ok, modules loaded: RecursiveContents, Main.
```

让我们来看看函数中的简单谓词能否正常工作：

```
ghci> :type betterFind (sizeP `equalP` 1024)
betterFind (sizeP `equalP` 1024) :: FilePath -> IO [FilePath]
```

注意我们并没有直接调用 betterFind，我们只是确定我们的表达式进行了类型检查，现在我们需要更多的方法来列出大小为特定值的所有文件，之前的成功给了我们继续下去的勇气。

:: _avoiding-boilerplate-with-lifting:

9.6.1 多用提升（lifting）来减少样板代码

除了 equalP，我们还将能够编写其他二元函数，我们更希望不去写他们每个的具体实现，因为这看起来只是重复工作：

```
-- file: ch09/BetterPredicate.hs
liftP :: (a -> b -> c) -> InfoP a -> b -> InfoP c
liftP q f k w x y z = f w x y z `q` k

greaterP, lesserP :: (Ord a) => InfoP a -> a -> InfoP Bool
greaterP = liftP (>)
lesserP = liftP (<)
```

[Forec 译注：此处 liftP 的参数可能较容易弄混，这里我们令 k 的类型和 a 相同，令 c 的类型是 Bool，通过下面的写法可能会更好理解一些：

```
liftP :: (a -> a -> Bool) -> InfoP a -> a -> InfoP Bool
liftP comparator getter argument w x y z = (getter w x y z) `comparator` argument
```

这里的 comparator 对应原文中的 q，为 (a -> b -> c) 类型的函数，getter 对应原文的 f，argument 对应原文的 k。getter 是 InfoP a 类的函数，这类函数根据 Predicate 类型中的元素计算出一个结果。

因此整个 `liftP` 函数接收到参数的类型签名为 `(a -> b -> c) -> InfoP a -> b -> w -> x -> y -> z`，此处 `w -> x -> y -> z` 是 `InfoP c` 展开的前四项。`getter` 从这四项中提取到一个类型为 `a` 的值，并和 `argument` 比较，最终返回 `Bool` 。在原文的代码中则返回类型为 `c` 的值，`c` 类型由比较函数决定。

]

为了完成这个，让我们使用 Haskell 的抽象功能，定义 `equalP` 代替直接调用 `==`，我们就可以把二元函数作为参数传入我们想调用的函数。

函数动作，比如 `>`，以及将它转换成另一个函数操作另一种上下文，在这里是 `greaterP`，通过提升（lifting）将它引入到上下文，这解释了当前函数名称中 lifting 出现的原因，提升让我们复用代码并降低模板的使用，在本书的后半部分的内容中，我们将大量使用这一技术

当我们提升一个函数，我们通常将它转换到原始类型和一个新版本——提升和未提升两个版本

在这里，将 `q` 作为 `liftP` 的第一个参数是经过深思熟虑的，这使得我们可能写一个对 `greaterP` 和 `lesserP` 都有意义的定义，实践中发现，相较其他语言，Haskell 中参数的最佳适配成为 api 设计中最重要的一部分。语言内部要求参数转换，在 Haskell 中放错一个参数的位置就将失去程序的所有意义。

我们可以通过组合子（combinators）恢复一些意义，比如，直到 2007 年 `forM` 才加入 `Control.Monad` 模块，在此之前，人们用的是 `flip mapM` 。

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :t forM
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
ghci> :t flip mapM
flip mapM :: (Monad m) => [a] -> (a -> m b) -> m [b]
```

9.6.2 谓词组合

如果我们希望组合谓词，我们可以循着手边最明显的路径来开始

```
-- file: ch09/BetterPredicate.hs
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z
```

现在我们知道了提升，他成为通过提升存在的布尔操作来削减代码量的更自然的选择。

```
-- file: ch09/BetterPredicate.hs
liftP2 :: (a -> b -> c) -> InfoP a -> InfoP b -> InfoP c
liftP2 q f g w x y z = f w x y z `q` g w x y z
```

(continues on next page)

(continued from previous page)

```
andP = liftP2 (&&)
orP = liftP2 (||)
```

注意 liftP2 非常像我们之前的 liftP 。事实上，liftP2 更通用，因为我们可以用它来实现 liftP：

```
-- file: ch09/BetterPredicate.hs
constP :: a -> InfoP a
constP k _ _ _ _ = k

liftP' q f k w x y z = f w x y z `q` constP k w x y z
```

Note: 组合子

在 Haskell 中，我们更希望函数的传入参数和返回值都是函数，就像组合子一样

回到之前定义的 myTest 函数，现在我们可以使用一些帮助函数了。

```
-- file: ch09/BetterPredicate.hs
myTest path _ (Just size) _ =
    takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

在加入组合子以后这个函数会变成什么样子：

```
-- file: ch09/BetterPredicate.hs
liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP`
          (sizeP `greaterP` 131072)
```

由于操作文件名是如此平常的行为，我们加入了最终组合子 liftPath 。

9.7 定义并使用新算符

可以通过特定领域语言定义新的操作：

```
-- file: ch09/BetterPredicate.hs
(==?) = equalP
(&&?) = andP
(>?) = greaterP
```

(continues on next page)

(continued from previous page)

```
myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)
```

这个括号在定义中是必要的，因为并未告诉 Haskell 有关之前和相关的操作，领域语言的操作如果没有边界（fixities）声明将会被以 `infixl 9` 之类的东西对待，计算从左到右，如果跳过这个括号，表达式将被解析成具有可怕错误的 `(((liftPath takeExtension) ==? ".cpp") &&? sizeP) >? 131072` 。

可以给操作添加边界声明，第一步是找出未提升的操作的，这样就可以模仿他们了：

```
ghci> :info ==
class Eq a where
  (==) :: a -> a -> Bool
  ...
      -- Defined in GHC.Base
infix 4 ==
ghci> :info &&
(&&) :: Bool -> Bool -> Bool        -- Defined in GHC.Base
infixr 3 &&
ghci> :info >
class (Eq a) => Ord a where
  ...
  (>) :: a -> a -> Bool
  ...
    -- Defined in GHC.Base
infix 4 >
```

学会这些就可以写一个不用括号的表达式，却和 `myTest3` 的解析结果一致的表达式了

```
-- file: ch09/BetterPredicate.hs
infix 4 ==?
infixr 3 &&?
infix 4 >?

myTest4 = liftPath takeExtension ==? ".cpp" &&? sizeP >? 131072
```

9.8 控制遍历

遍历文件系统时，我们喜欢在需要遍历的文件夹上有更多的控制权，简便方法之一是可以在函数中允许给定文件夹的部分子文件夹通过，然后返回另一个列表，这个列表可以移除元素，也可以要求和原始列表不同，或两者皆有，最简单的控制函数就是 id，原样返回未修改的列表。

为了应付多种情况，我们正在尝试改变部分表达，为了替代精心刻画的函数类型 `InfoP` ，我们将使用一个普通代数数据类型来表达相同的含义

```haskell
-- file: ch09/ControlledVisit.hs
data Info = Info {
      infoPath :: FilePath
    , infoPerms :: Maybe Permissions
    , infoSize :: Maybe Integer
    , infoModTime :: Maybe ClockTime
    } deriving (Eq, Ord, Show)

getInfo :: FilePath -> IO Info
```

记录语法给了我们一些“免费”的访问函数，例如 `infoPath`，`traverse` 函数的类型很简单，正如我们上面的提案一样，如果需要一个文件或者目录的信息，就调用 `getInfo` 函数：

```haskell
-- file: ch09/ControlledVisit.hs
traverse :: ([Info] -> [Info]) -> FilePath -> IO [Info]
```

`traverse` 的定义很短，但很有分量：

```haskell
-- file: ch09/ControlledVisit.hs
traverse order path = do
    names <- getUsefulContents path
    contents <- mapM getInfo (path : map (path </>) names)
    liftM concat $ forM (order contents) $ \info -> do
      if isDirectory info && infoPath info /= path
        then traverse order (infoPath info)
        else return [info]

getUsefulContents :: FilePath -> IO [String]
getUsefulContents path = do
    names <- getDirectoryContents path
    return (filter (`notElem` [".", ".."]) names)

isDirectory :: Info -> Bool
isDirectory = maybe False searchable . infoPerms
```

现在不再引入新技术，这就是我们遇到的最深奥的函数定义，一行行的深入他，解释它每行为何是这样，不过开始部分的那几行没什么神秘的，它们只是之前看到代码的拷贝。

观察变量 `contents` 的时候情况变得有趣起来，从左到右仔细阅读，已经知道 `names` 是一个包含目录项的列表。我们将当前目录的路径拼接到列表中每个元素的前面，再把当前目录路径加到列表里。然后再通过 `mapM` 将 `getInfo` 函数应用到产生的结果列表上。

接下来的这一行更深奥，继续从左往右看，我们看到本行的最后一个元素以一个匿名函数的定义开始，并持续到这一段的结尾，给定一个 Info 值，函数或者递归访问一个目录（有一个额外的判断条件保证我们不会重复访问 `path`），或者将当前值作为一个单元素列表返回（来匹配 `traverse` 的返回类型）。

函数通过 forM 获得 order 返回 info 列表中的每个元素，forM 是使用者提供的递归控制函数。

本行的开头，我们在一个新的上下文中使用了提升技术，liftM 函数将一个普通函数，concat ，提升到可在 IO monad 之中使用。换句话讲，liftM 将 forM 的结果值（类型为 [[Info]]）从 IO monad 中取出，把 concat 应用在其上（获得一个 [Info] 类型的返回值，这也是我们所需要的），最后将结果再放进 IO monad 里。

最后不要忘记定义 getInfo 函数：

```
-- file: ch09/ControlledVisit.hs
maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\_ -> return Nothing) (Just `liftM` act)

getInfo path = do
  perms <- maybeIO (getPermissions path)
  size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)
  modified <- maybeIO (getModificationTime path)
  return (Info path perms size modified)
```

在此唯一值得记录的事情是一个有用的组合子，maybeIO ，将一个可能抛出异常的 IO 操作转换成用 Maybe 包装的结果

[Forec 译注：在 GHC 7.6 以后的版本中，以上代码编译会出现问题，需添加 import 列表，并对部分代码做修改如下：

```
{-# LANGUAGE ScopedTypeVariables #-}
import Prelude hiding (traverse)
import Control.Monad (filterM, liftM, forM)
import System.Directory (Permissions(..),
                                                getModificationTime,
                                                getPermissions,
                                                getDirectoryContents)
import Data.Time.Clock (UTCTime(..))
import System.FilePath (takeExtension, (</>))
import Control.Exception (bracket, handle, SomeException)
import System.IO (IOMode(..), hClose, hFileSize, openFile)

maybeIO act = handle (\(_::SomeException) -> return Nothing) (liftM Just act)
```

你可以在 GHC 中导入代码并查看执行结果：

```
ghci> :l ControlledVisit.hs
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( ControlledVisit.hs, interpreted )
Ok, modules loaded: RecursiveContents, Main.
ghci> :m +Data.List
```

(continues on next page)

(continued from previous page)

```
ghci> infos<-traverse sort "."
ghci> map infoPath infos
[".",".\\BetterPredicate.hs",".\\ControlledVisit.hs",".\\RecursiveContents.hs"]
```

]

9.8.1 练习

1. 如果想要以反字母顺序 (reverse alphabetic) 遍历一棵目录树，该传给 `traverse` 什么参数?
2. 使用 `id` 作为控制函数，`traverse id` 会前序遍历一棵树，即返回值中父目录出现在子目录之前。写一个控制函数让 `traverse` 完成后序遍历，即在子目录之后返回父目录。
3. 使得《谓词组合》一节里面的谓词和组合子可以处理新的 `info` 类型。
4. 给 `traverse` 写一个包装器，可以让你通过一个谓词控制遍历，通过另一个谓词来过滤返回结果

9.9 代码密度，可读性和学习过程

`traverse` 这样密实的代码在 Haskell 中并不多见，这种代码具有显著的表达力，而且其实只需要相对很少的练习就能够以这种方式流利的阅读和编写代码:

```
-- file: ch09/ControlledVisit.hs
traverseVerbose order path = do
    names <- getDirectoryContents path
    let usefulNames = filter (`notElem` [".", ".."]) names
    contents <- mapM getEntryName ("" : usefulNames)
    recursiveContents <- mapM recurse (order contents)
    return (concat recursiveContents)
  where getEntryName name = getInfo (path </> name)
        isDirectory info = case infoPerms info of
                             Nothing -> False
                             Just perms -> searchable perms
        recurse info = do
            if isDirectory info && infoPath info /= path
                then traverseVerbose order (infoPath info)
                else return [info]
```

作为对比，这里有一个不那么复杂的代码，这也许适合一个对 Haskell 了解不那么深入的程序员

这里我们只是对部分代码做了下替换。我们在 `where` 块中定义了一些局部函数来替换原来使用的部分应用（partial application）和函数组合（function composition）。通过使用 `case` 表达式来替代 `maybe` 组合子。为了替代 `liftM` ，我们手动将 `concat` 提升。

并不是说密实的代码永远都是好的，`traverse` 函数的每一行原始代码都很短，我们引入一个局部变量和局部函数来保证代码干净且足够短，使用的名字都很有描述性，同时使用函数组合和管线化，最长的管道只含有三个元素。[译注：好像本书前面都没有介绍过何谓管线化，而且我在本章的代码里也没看到哪里是在使用管线化，姑且先把字面意思翻出来吧。]

编写可维护的 Haskell 代码核心是找到深度和可读性的折中，能否做到这点取决于你的实践层次：

- 成为 Haskell 程序员之前，Andrew 并不知道使用标准库的方式，为此付出的代价则是写了一大堆不必要的重复代码。
- Zack 是一个有数月编程经验的，并且精通通过 (`.`) 组合长管道的技巧。每当代码需要改动，就需要重构一个管道，他无法更深入的理解已经存在的管道的意义，而这些管道也太脆弱而无法修正。
- Monica 有相当时间的编程经验，她对 Haskell 库和编写整洁的代码非常熟悉，但她避免使用高深度的风格，她的代码可维护，同时她还找到了一种简单地方法来面对快速的需求变更

9.10 从另一个角度来看遍历

相比原始的 `betterFind` 函数，`traverse` 函数给我们更多控制权的同时仍存在一个问题，我们可以避免递归目录，但我们不能过滤其他文件名直到我们获得整个名称树，如果递归含有 100000 个文件的目录的同时只关注其中三个，在获得这三个需要的文件名之前需要给出一个含有 10000 个元素的表。

一个可行的方法是提供一个过滤器作为递归的新参数，我们将它应用到生成的名单中，这将允许我们获得一个只包含我们需要元素的列表

然而，这个方法也存在缺点：假如说我们只需要三个的目录项，并且这些目录项恰巧是这 10000 个我们需要遍历的元素之中的前几个，这种情况下我们会无谓的去遍历 99997 个元素，这并不是个故弄玄虚的问题，举个例子，邮箱文件夹中存放了包含许多邮件信息的文件夹——就像一个有大量文件的目录，那么代表邮箱的目录含有数千个文件就很正常。

我们可以通过从另一个角度思考问题来之前两个遍历函数的弱点：我们将遍历文件系统这件事看做对目录层级结构进行折叠（fold），这样如何？

我们所熟悉的 fold，`foldr` 和 `foldl'` ，很好地概括了如何在遍历列表的同时累计结果，把这个想法从遍历列表扩展到遍历目录树简直小菜一碟，但我们仍希望在 `fold` 中加入一个 控制的功能，我们将这个控制表达为一个代数数据类型：

```
-- file: ch09/FoldDir.hs
data Iterate seed = Done     { unwrap :: seed }
                  | Skip     { unwrap :: seed }
                  | Continue { unwrap :: seed }
                    deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed
```

Iterator 类型给函数一个便于使用的别名，它需要一个种子和一个 info 值来表达一个目录项，并返回一个新种子和一个对我们 fold 函数的指令，这个说明通过 Iterate 类型的构造器来表达：

- 如果这个构造器已经完成，遍历将立即停止，被 Done 包裹的值将作为结果返回。
- 如果这个说明被 Skip，并且当前 info 代表一个目录，遍历将不再递归搜寻这个目录。
- 否则，这个遍历仍将继续，使用包裹值作为下一个调用 fold 函数的参数。

我们的 fold 函数逻辑上来看是个左折叠的，因为我们开始从我们第一个遇到的目录项开始 fold 操作，而每步中的种子是之前一步的结果。

```
-- file: ch09/FoldDir.hs
foldTree :: Iterator a -> a -> FilePath -> IO a

foldTree iter initSeed path = do
    endSeed <- fold initSeed path
    return (unwrap endSeed)
  where
    fold seed subpath = getUsefulContents subpath >>= walk seed

    walk seed (name:names) = do
      let path' = path </> name
      info <- getInfo path'
      case iter seed info of
        done@(Done _) -> return done
        Skip seed'    -> walk seed' names
        Continue seed'
          | isDirectory info -> do
              next <- fold seed' path'
              case next of
                done@(Done _) -> return done
                seed''        -> walk (unwrap seed'') names
          | otherwise -> walk seed' names
    walk seed _ = return (Continue seed)
```

[Forec 译注：要在 GHC 中使用这段代码，你需要修改 ControlledVisit.hs 的头部，并在 FoldDir.hs 的头部加入 import ControlledVisit：

```
{-# LANGUAGE ScopedTypeVariables #-}
module ControlledVisit
(
        Info(..),
        getInfo,
        getUsefulContents,
        isDirectory
```

(continues on next page)

(continued from previous page)

```
) where

-- code left
```

]

这段代码有些有意思的地方。开始是通过使用作用域来避免通过额外的参数，最高层 `foldTree` 函数只是 `fold` 的包装器，用来揭开 `fold` 的最后结果的生成器。

由于 `fold` 是局部函数，我们不需要把 `foldTree` 的 `iter` 变量传给它，它可以直接访问外围作用域的变量，相似的，`walk` 也可以在它的外围作用域中部看到 `path` 。

另一个需要指出的点是 `walk` 是一个尾递归，在我们最初的函数中用来替代一个匿名函数调用。通过自己把控，可以在任何需要的时候停止，这使得当 `iterator` 返回 `Done` 的时候就可以退出。

虽然 `fold` 调用 `walk` ，`walk` 递归调用 `fold` 来遍历子目录，每个函数返回一个用 `Iterate` 包装起来的种子，当 `fold` 被调用，并且返回，`walk` 检查返回并观察需要继续还是退出。通过这种方式，一个 `Done` 的返回直接终止两个函数中的所有递归调用。

实践中一个 `iterator` 像什么，下面这个相对复杂的例子里会查找最多三个位图文件，同时还不会在 SVN 的元数据目录中进行查找：

```
-- file: ch09/FoldDir.hs
atMostThreePictures :: Iterator [FilePath]

atMostThreePictures paths info
    | length paths == 3
      = Done paths
    | isDirectory info && takeFileName path == ".svn"
      = Skip paths
    | extension `elem` [".jpg", ".png"]
      = Continue (path : paths)
    | otherwise
      = Continue paths
  where extension = map toLower (takeExtension path)
        path = infoPath info
```

为了使用这个需要调用 `foldTree atMostThreePictures []` ，它给我们一个 `IO [FilePath]` 类型的返回值。

当然，`iterators` 并不需要如此复杂，下面是个对目录进行计数的代码：

```
-- file: ch09/FoldDir.hs
countDirectories count info =
    Continue (if isDirectory info
```

(continues on next page)

(continued from previous page)

```
            then count + 1
            else count)
```

传递给 `foldTree` 的初始种子（seed）为数字零。

9.10.1 练习

1. 修正 `foldTree` 来允许调用改变遍历目录项的顺序。
2. `foldTree` 函数展示了前序遍历，将它修正为允许调用方决定便利顺序。
3. 写一个组合子的库允许 `foldTree` 接收不同类型的 `iterators` ，你能写出更简洁的 `iterators` 吗?

9.11 编码指南

有许多好的 Haskell 程序员的习惯来自经验，我们有一些通用的经验给你，这样你可以更快的写出易于阅读的代码。

正如已经提到的，Haskell 中永远使用空格，而不是 tab 。

如果你发现代码里有个片段聪明到炸裂，停下来，然后思考下如果你离开代码一个月是否还能懂这段代码。

对类型和变量的命名惯例是使用驼峰命名法，例如 `myVariableName` ，这种风格在 Haskell 中也同样流行，不要去想你的其他命名习惯，如果你遵循一个不标准的惯例，那么你的代码在其他人看来可能会很刺眼。

即使你已经用了 Haskell 一段时间，在你写小函数之前花费几分钟的时间查阅库函数，如果标准库并没有提供你需要的函数，你可能需要组合出一个新的函数来获得你想要的结果。

组合函数的长管道难以阅读，长意味着包含三个以上元素的序列，如果你有这样一个管道，使用 `let` 或者 `where` 语句块将它分解成若干个小部分，给每个管道元素一个有意义的名字，然后再将他们回填到代码，如果你想不出一个有意义的名字，问下自己能不能解释这段代码的功能，如果不能，简化你的代码。

即使在编辑器中很容易格式化长于八十列的代码，宽度仍然是个重要问题，宽行在 80 行之外的内容通常会被截断，这非常伤害可读性，每一行不超过八十个字符，这样你就可以写入单独的一行，这帮助你保持每一行代码不那么复杂，从而更容易被人读懂。

9.11.1 常用布局风格

只要你的代码遵守布局规范，那么他并不会给人一团乱麻的感觉，因此也不会造成误解，也就是说，有些布局风格是常用的。

`in` 关键字通常正对着 `let` 关键字，如下所示：

```
-- file: ch09/Style.hs
tidyLet = let foo = undefinedwei's
              bar = foo * 2
          in undefined
```

单独列出 in 或者让 in 在一系列等式之后跟着的写法都是正确的，但下面这种写法则会显得很奇怪：

```
-- file: ch09/Style.hs
weirdLet = let foo = undefined
               bar = foo * 2
    in undefined

strangeLet = let foo = undefined
                 bar = foo * 2 in
    undefined
```

与此相反，让 do 在行尾跟着而非在行首单独列出：

```
-- file: ch09/Style.hs
commonDo = do
  something <- undefined
  return ()

-- not seen very often
rareDo =
  do something <- undefined
     return ()
```

括号和分号即使合法也很少用到，他们的使用并不存在问题，只是让代码看起来奇怪，同时让 Haskell 写成的代码不必遵守排版规则。

```
-- file: ch09/Style.hs
unusualPunctuation =
    [ (x,y) | x <- [1..a], y <- [1..b] ] where {
                                            b = 7;
 a = 6 }

preferredLayout = [ (x,y) | x <- [1..a], y <- [1..b] ]
    where b = 7
          a = 6
```

如果等式的右侧另起一行，通常在和他本行内，相关变量名或者函数定义的下方之前留出一些空格。

```
-- file: ch09/Style.hs
normalIndent =
    undefined

strangeIndent =
                            undefined
```

空格缩进的数量有多种选择，有时候在一个文件中，二，三，四格缩进都很正常，一个缩进也合法，但不常用，而且容易被误读。

写 `where` 语句的缩进时，最好让它分辨起来比较容易：

```
-- file: ch09/Style.hs
goodWhere = take 5 lambdas
    where lambdas = []

alsoGood =
    take 5 lambdas
  where
    lambdas = []

badWhere =            -- legal, but ugly and hard to read
    take 5 lambdas
    where
    lambdas = []
```

9.12 练习

即使本章内容指导你们完成文件查找代码，但这并不意味着真正的系统编程，因为 haskell 移植的 `IO` 库并不暴露足够的消息给我们写有趣和复杂的查询。

1. 把本章代码移植到你使用平台的 `api` 上，`System.Posix` 或者 `System.Win32` 。
2. 加入查找文件所有者的功能，将这个属性对谓词可见。

CHAPTER

TEN

第 10 章：代码案例学习：解析二进制数据格式

本章将会讨论一个常见任务：解析（parsing）二进制文件。选这个任务有两个目的。第一个确实是想谈谈解析过程，但更重要的目标是谈谈程序组织、重构和消除样板代码（boilerplate code：通常指不重要，但没它又不行的代码）。我们将会展示如何清理冗余代码，并为第十四章讨论 Monad 做点准备。

我们将要用到的文件格式来自于 netpbm 库，它包含一组用来处理位图图像的程序及文件格式，它古老而令人尊敬。这种文件格式不但被广泛使用，而且还非常简单，虽然解析过程也不是完全没有挑战。对我们而言最重要的是，netpbm 文件没有经过压缩。

10.1 灰度文件

netpbm 的灰度文件格式名为 PGM（" portable grey map"）。事实上它不是一个格式，而是两个：纯文本（又名 P2）格式使用 ASCII 编码，而更常用的原始（P5）格式则采用二进制表示。

每种文件格式都包含头信息，头信息以一个“魔法”字符串开始，指出文件格式。纯文本格式是 `P2`，原始格式是 `P5`。魔法字符串之后是空格，然后是三个数字：宽度、高度、图像的最大灰度值。这些数字用十进制 ASCII 数字表示，并用空格隔开。

最大灰度值之后便是图像数据了。在原始文件中，这是一串二进制值。纯文本文件中，这些值是用空格隔开的十进制 ASCII 数字。

原始文件可包含多个图像，一个接一个，每个都有自己的头信息。纯文本文件只包含一个图像。

10.2 解析原始 PGM 文件

首先我们来给原始 PGM 文件写解析函数。PGM 解析函数是一个纯函数。它不管获取数据，只管解析。这是一种常见的 Haskell 编程方法。通过把数据的获取和处理分开，我们可以很方便地控制从哪里获取数据。

我们用 `ByteString` 类型来存储灰度数据，因为它比较节省空间。由于 PGM 文件以 ASCII 字符串开头，文件内容又是二进制数据，我们同时载入两种形式的 `ByteString` 模块。

```
-- file: ch10/PNM.hs
import qualified Data.ByteString.Lazy.Char8 as L8
import qualified Data.ByteString.Lazy as L
import Data.Char (isSpace)
```

我们并不关心 ByteString 类型是惰性的还是严格的，因此我们随便选了惰性的版本。

我们用一个直白的数据类型来表示 PGM 图像。

```
-- file: ch10/PNM.hs
data Greymap = Greymap {
      greyWidth :: Int
    , greyHeight :: Int
    , greyMax :: Int
    , greyData :: L.ByteString
    } deriving (Eq)
```

通常来说，Haskell 的 Show 实例会生成数据的字符串表示，我们还可以用 read 读回来。然而，对于一个位图图像文件来说，这可能会生成一个非常大的字符串，比如当你对一张照片调用 show 的时候。基于这个原因，我们不准备让编译器自动为我们派生 Show 实例；我们会自己实现，并刻意简化它。

```
-- file: ch10/PNM.hs
instance Show Greymap where
    show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h + " " ++ show m
```

我们的 Show 实例故意没打印位图数据，也就没必要写 Read 实例了，因为我们无法从 show 的结果重构 Greymap。

解析函数的类型显而易见。

```
-- file: ch10/PNM.hs
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
```

这个函数以一个 ByteString 为参数，如果解析成功的话，它返回一个被解析的 Greymap 值以及解析之后剩下的字符串，剩下的字符串以后会用到。

解析函数必须一点一点处理输入数据。首先，我们必须确认我们正在处理的是原始 PGM 文件；然后，我们处理头信息中的数字；最后我们处理位图数据。下面是是一种比较初级的实现方法，我们会在它的基础上不断改进。

```
-- file: ch10/PNM.hs
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString

-- "nat" here is short for "natural number"
getNat :: L.ByteString -> Maybe (Int, L.ByteString)
```

(continues on next page)

(continued from previous page)

```
getBytes :: Int -> L.ByteString
         -> Maybe (L.ByteString, L.ByteString)

parseP5 s =
  case matchHeader (L8.pack "P5") s of
    Nothing -> Nothing
    Just s1 ->
      case getNat s1 of
        Nothing -> Nothing
        Just (width, s2) ->
          case getNat (L8.dropWhile isSpace s2) of
            Nothing -> Nothing
            Just (height, s3) ->
              case getNat (L8.dropWhile isSpace s3) of
                Nothing -> Nothing
                Just (maxGrey, s4)
                  | maxGrey > 255 -> Nothing
                  | otherwise ->
                      case getBytes 1 s4 of
                        Nothing -> Nothing
                        Just (_, s5) ->
                          case getBytes (width * height) s5 of
                            Nothing -> Nothing
                            Just (bitmap, s6) ->
                              Just (Greymap width height maxGrey bitmap, s6)
```

这段代码非常直白，它把所有的解析放在了一个长长的梯形 case 表达式中。每个函数在处理完它所需要的部分后会把剩余的 ByteString 返回。我们再把这部分传给下个函数。像这样我们将结果依次解构，如果解析失败就返回 Nothing，否则便又向最终结迈近了一步。下面是我们在解析过程中用到的函数的定义。它们的类型被注释掉了因为已经写过了。

```
-- file: ch10/PNM.hs
-- L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader prefix str
    | prefix `L8.isPrefixOf` str
        = Just (L8.dropWhile isSpace (L.drop (L.length prefix) str))
    | otherwise
        = Nothing

-- L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
             Nothing -> Nothing
```

(continues on next page)

(continued from previous page)

```
            Just (num,rest)
                | num <= 0     -> Nothing
                | otherwise    -> Just (num, L8.dropWhile isSpace rest)

-- Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes n str = let count           = fromIntegral n
                     both@(prefix,_) = L.splitAt count str
                 in if L.length prefix < count
                    then Nothing
                    else Just both
```

10.3 消除样板代码

parseP5 函数虽然能用，但它的代码风格却并不令人满意。它不断挪向屏幕右侧，非常明显，再来个稍微复杂点的函数它就要横跨屏幕了。我们不断构建和解构 Maybe 值，只在 Just 匹配特定值的时候才继续。所有这些相似的 case 表达式就是“样板代码”，它掩盖了我们真正要做的事情。总而言之，这段代码需要抽象重构。

退一步看，我们能观察到两种模式。第一，很多我们用到的函数都有相似的类型，它们最后一个参数都是 ByteString，返回值类型都是 Maybe。第二，parseP5 函数不断解构 Maybe 值，然后要么失败退出，要么把展开之后的值传给下个函数。

我们很容易就能写个函数来体现第二种模式。

```
-- file: ch10/PNM.hs
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v  >>? f = f v
```

(>>?) 函数非常简单：它接受一个值作为左侧参数，一个函数 f 作为右侧参数。如果值不为 Nothing，它就把函数 f 应用在 Just 构造器中的值上。我们把这个函数定义为操作符这样它就能把别的函数串联在一起了。最后，我们没给 (>>?) 定义结合度，因此它默认为 infixl 9（左结合，优先级最高的操作符）。换言之，a >>? b >>? c 会从左向右求值，就像 (a >>? b) >>? c) 一样。

有了这个串联函数，我们来重写一下解析函数。

```
-- file: ch10/PNM.hs
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
    matchHeader (L8.pack "P5") s      >>?
    \s -> skipSpace ((), s)           >>?
    (getNat . snd)                    >>?
```

(continues on next page)

(continued from previous page)

```
    skipSpace                          >>?
    \(width, s) ->   getNat s          >>?
    skipSpace                          >>?
    \(height, s) ->  getNat s          >>?
    \(maxGrey, s) -> getBytes 1 s      >>?
    (getBytes (width * height) . snd) >>?
    \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)

skipSpace :: (a, L.ByteString) -> Maybe (a, L.ByteString)
skipSpace (a, s) = Just (a, L8.dropWhile isSpace s)
```

理解这个函数的关键在于理解其中的链。每个 (>>?) 的左侧都是一个 Maybe 值，右侧都是一个返回 Maybe 值的函数。这样，Maybe 值就可以不断传给后续 (>>?) 表达式。

我们新增了 skipSpace 函数用来提高可读性。通过这些改进，我们已将代码长度减半。通过移除样板 case 代码，代码变得更容易理解。

尽管在匿名（lambda）函数中我们已经警告过不要过度使用匿名函数，在上面的函数链中我们还是用了一些。因为这些函数太小了，给它们命名并不能提高可读性。

10.4 隐式状态

到这里还没完。我们的代码显式地用序对传递结果，其中一个元素代表解析结果的中间值，另一个代表剩余的 ByteString 值。如果我们想扩展代码，比方说记录已经处理过的字节数，以便在解析失败时报告出错位置，那我们已经有 8 个地方要改了，就为了把序对改成三元组。

这使得本来就没多少的代码还很难修改。问题在于用模式匹配从序对中取值：我们假设了我们总是会用序对，并且把这种假设编进了代码。尽管模式匹配非常好用，但如果不慎重，我们还是会误入歧途。

让我们解决新代码带来的不便。首先，我们来修改解析状态的类型。

```
-- file: ch10/Parse.hs
data ParseState = ParseState {
      string :: L.ByteString
    , offset :: Int64            -- imported from Data.Int
    } deriving (Show)
```

我们转向了代数数据类型，现在我们既可以记录当前剩余的字符串，也可以记录相对于原字符串的偏移值了。更重要的改变是用了记录语法：现在可以避免使用模式匹配来获取状态信息了，可以用 string 和 offset 访问函数。

我们给解析状态起了名字。给东西起名字方便我们推理。例如，我们现在可以这么看解析函数：它处理一个解析状态，产生新解析状态和一些别的信息。我们可以用 Haskell 类型直接表示。

```haskell
-- file: ch10/Parse.hs
simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

为了给用户更多帮助，我们可以在解析失败时报告一条错误信息。只需对解析器的类型稍作修改即可。

```haskell
-- file: ch10/Parse.hs
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

为了防患于未然，我们最好不要将解析器的实现暴露给用户。早些时候我们显式地用序对来表示状态，当我们想扩展解析器的功能时，我们马上就遇到了麻烦。为了防止这种现象再次发生，我们用一个 `newtype` 声明来隐藏解析器的细节。

```haskell
--file: ch10/Parse.hs
newtype Parse a = Parse {
    runParse :: ParseState -> Either String (a, ParseState)
    }
```

别忘了 `newtype` 只是函数在编译时的一层包装，它没有运行时开销。我们想用这个函数时，我们用 `runParser` 访问器。

如果我们的模块不导出 `Parse` 值构造器，我们就能确保没人会不小心创建一个解析器，或者通过模式匹配来观察其内部构造。

10.4.1 identity 解析器

我们来定义一个简单的 *identity* 解析器。它把输入值转为解析结果。从这个意义上讲，它有点像 `id` 函数。

```haskell
-- file: ch10/Parse.hs
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

这个函数没动解析状态，只把它的参数当成了解析结果。我们把函数体包装成 `Parse` 类型以通过类型检查。我们该怎么用它去解析呢?

首先我们得把 `Parse` 包装去掉从而得到里面的函数。这通过 `runParse` 函数实现。然后得创建一个 `ParseState`，然后对其调用解析函数。最后，我们把解析结果和最终的 `ParseState` 分开。

```haskell
-- file: ch10/Parse.hs
parse :: Parse a -> L.ByteString -> Either String a
parse parser initState
    = case runParse parser (ParseState initState 0) of
```

(continues on next page)

(continued from previous page)

```
        Left err          -> Left err
        Right (result, _) -> Right result
```

由于 identity 解析器和 parse 函数都没有检查解析状态，我们都不用传入字符串就可以试验我们的代码。

```
Prelude> :r
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, modules loaded: Main.
*Main> :type parse (identity 1) undefined
parse (identity 1) undefined :: Num a => Either String a
*Main> parse (identity 1) undefined
Right 1
*Main> parse (identity "foo") undefined
Right "foo"
```

一个不检查输入的解析器可能有点奇怪，但很快我们就可以看到它的用处。同时，我们更加确信我们的类型是正确的，基本了解了代码是如何工作的。

10.4.2 记录语法、更新以及模式匹配

记录语法的用处不仅仅在于访问函数：我们可以用它来复制或部分改变已有值。就像下面这样：

```
-- file: ch10/Parse.hs
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset =
    initState { offset = newOffset }
```

这会创建一个跟 initState 完全一样的 ParseState 值，除了 offset 字段会替换成 newOffset 指定的值。

```
*Main> let before = ParseState (L8.pack "foo") 0
*Main> let after = modifyOffset before 3
*Main> before
ParseState {string = "foo", offset = 0}
*Main> after
ParseState {string = "foo", offset = 3}
```

在大括号里我们可以给任意多的字段赋值，用逗号分开即可。

10.4.3 一个更有趣的解析器

现在来写个解析器做一些有意义的事情。我们并不好高骛远：我们只想解析单个字节而已。

```
-- file: ch10/Parse.hs
-- import the Word8 type from Data.Word
parseByte :: Parse Word8
parseByte =
    getState ==> \initState ->
    case L.uncons (string initState) of
      Nothing ->
          bail "no more input"
      Just (byte,remainder) ->
          putState newState ==> \_ ->
          identity byte
        where newState = initState { string = remainder,
                                     offset = newOffset }
              newOffset = offset initState + 1
```

定义中有几个新函数。

`L8.uncons` 函数取出 `ByteString` 中的第一个元素。

```
ghci> L8.uncons (L8.pack "foo")
Just ('f',Chunk "oo" Empty)
ghci> L8.uncons L8.empty
Nothing
```

`getState` 函数得到当前解析状态，`putState` 函数更新解析状态。`bail` 函数终止解析并报告错误。`(==>)` 函数把解析器串联起来。我们马上就会详细介绍这些函数。

Note: “悬挂”的匿名函数

> `parseByte` 呈现出的风格是我们此前没有讨论过的。它包含了一些匿名函数。这些匿名函数的参数和 `->` 都写在一行的末尾，而函数的主体部分从下一行开始。
>
> 这种风格的匿名函数布局没有官方的名字，所以我们不妨称它为“悬挂”的匿名函数。它主要的用途是为函数主体部分留出更多空间。它也让两个紧邻函数之间的关系更加明显：比如，第一个函数的结果常常作为参数传递给第二个函数。

TBD

10.4.4 获取和修改解析状态

`parseByte` 函数并不接受解析状态作为参数。相反，它必须调用 `getState` 来得到解析状态的副本，然后调用 `putState` 将当前状态更新为新状态。

```haskell
-- file: ch10/Parse.hs
getState :: Parse ParseState
getState = Parse (\s -> Right (s, s))

putState :: ParseState -> Parse ()
putState s = Parse (\_ -> Right ((), s))
```

阅读这些函数的时候，记得序对左元素为 Parse 结果，右元素为当前 ParseState。这样理解起来会比较容易。

getState 将当前解析状态展开，这样调用者就能访问里面的字符串。putState 将当前解析状态替换为一个新状态。(==>) 链中的下一个函数将会使用这个状态。

这些函数将显式的状态处理移到了需要它们的函数的函数体内。很多函数并不关心当前状态是什么，因而它们也不会调用 getState 或 putState。跟之前需要手动传递元组的解析器相比，现在的代码更加紧凑。在之后的代码中就能看到效果。

我们将解析状态的细节打包放入 ParseState 类型中，然后我们通过访问器而不是模式匹配来访问它。隐式地传递解析状态给我们带来另外的好处。如果想增加解析状态的信息，我们只需修改 ParseState 定义，以及需要新信息的函数体即可。跟之前通过模式匹配暴露状态的解析器相比，现在的代码更加模块化：只有需要新信息的代码会受到影响。

10.4.5 报告解析错误

在定义 Parse 的时候我们已经考虑了出错的情况。(==>) 组合子不断检查解析错误并在错误发生时终止解析。但我们还没来得及介绍用来报告解析错误的 bail 函数。

```haskell
-- file: ch10/Parse.hs
bail :: String -> Parse a
bail err = Parse $ \s -> Left $
           "byte offset " ++ show (offset s) ++ ": " ++ err
```

调用 bail 之后，(==>) 会模式匹配包装了错误信息的 Left 构造器，并且不会调用下一个解析器。这使得错误信息可以沿着调用链返回。

10.4.6 把解析器串联起来

(==>) 函数的功能和之前介绍的 (>>?) 函数功能类似：它可以作为“胶水”把函数串联起来。

```haskell
-- file: ch10/Parse.hs
(==>) :: Parse a -> (a -> Parse b) -> Parse b

firstParser ==> secondParser  =  Parse chainedParser
```

(continues on next page)

(continued from previous page)

```
  where chainedParser initState   =
          case runParse firstParser initState of
            Left errMessage ->
                Left errMessage
            Right (firstResult, newState) ->
                runParse (secondParser firstResult) newState
```

(==>) 函数体很有趣，还稍微有点复杂。回想一下，Parse 类型表示一个被包装的函数。既然 (==>) 函数把两个 Parse 串联起来并产生第三个，它也必须返回一个被包装的函数。

这个函数做的并不多：它仅仅创建了一个闭包（closure）用来记忆 firstParser 和 secondParser 的值。

Note: 闭包是一个函数和它所在的环境，也就是它可以访问的变量。闭包在 Haskell 中很常见。例如，(+5) 就是一个闭包。实现的时候必须将 5 记录为 (+) 操作符的第二个参数，这样得到的函数才能把 5 加给它的参数。

在应用 parse 之前，这个闭包不会被展开应用。应用的时候，它会求值 firstParser 并检查它的结果。如果解析失败，闭包也会失败。否则，它会把解析结果及 newState 传给 secondParser。

这是非常具有想象力、非常微妙的想法：我们实际上用了一个隐藏的参数将 ParseState 在 Parse 链之间传递。(我们在之后几章还会碰到这样的代码，所以现在不懂也没有关系。)

10.5 Functor 简介

现在我们对 map 函数已经有了一个比较详细的了解，它把函数应用在列表的每一个元素上，并返回一个可能包含另一种类型元素的列表。

```
ghci> map (+1) [1,2,3]
[2,3,4]
ghci> map show [1,2,3]
["1","2","3"]
ghci> :type map show
map show :: (Show a) => [a] -> [String]
```

map 函数这种行为在别的实例中可能有用。例如，考虑一棵二叉树。

```
-- file: ch10/TreeMap.hs
data Tree a = Node (Tree a) (Tree a)
            | Leaf a
              deriving (Show)
```

如果想把一个字符串树转成一个包含这些字符串长度的树，我们可以写个函数来实现：

```
-- file: ch10/TreeMap.hs
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

我们试着寻找一些可能转成通用函数的模式，下面就是一个可能的模式。

```
-- file: ch10/TreeMap.hs
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a)   = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

正如我们希望的那样，`treeLengths` 和 `treeMap length` 返回相同的结果。

```
ghci> let tree = Node (Leaf "foo") (Node (Leaf "x") (Leaf "quux"))
ghci> treeLengths tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap length tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap (odd . length) tree
Node (Leaf True) (Node (Leaf True) (Leaf False))
```

Haskell 提供了一个众所周知的类型类来进一步一般化 `treeMap`。这个类型类叫做 `Functor`，它只定义了一个函数 `fmap`。

```
-- file: ch10/TreeMap.hs
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

我们可以把 `fmap` 当做某种提升函数，就像我们在 avoiding-boilerplate-with-lifting 一节中介绍的那样。它接受一个参数为普通值 `a -> b` 的函数并把它提升为一个参数为容器 `f a -> f b` 的函数。其中 `f` 是容器的类型。

举个例子，如果我们用 `Tree` 替换类型变量 `f`，`fmap` 的类型就会跟 `treeMap` 的类型相同。事实上我们可以用 `treeMap` 作为 `fmap` 对 `Tree` 的实现。

```
-- file: ch10/TreeMap.hs
instance Functor Tree where
    fmap = treeMap
```

我们可以用 `map` 作为 `fmap` 对列表的实现。

```
-- file: ch10/TreeMap.hs
instance Functor [] where
```

(continues on next page)

(continued from previous page)

```
    fmap = map
```

现在我们可以把 fmap 用于不同类型的容器上了。

```
ghci> fmap length ["foo","quux"]
[3,4]
ghci> fmap length (Node (Leaf "Livingstone") (Leaf "I presume"))
Node (Leaf 11) (Leaf 9)
```

Prelude 定义了一些常见类型的 Functor 实例，如列表和 Maybe。

```
-- file: ch10/TreeMap.hs
instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Maybe 的这个实例很清楚地表明了 fmap 要做什么。对于类型的每一个构造器，它都必须给出对应的行为。例如，如果值被包装在 Just 里，fmap 实现把函数应用在展开之后的值上，然后再用 Just 重新包装起来。

Functor 的定义限制了我们能用 fmap 做什么。例如，如果一个类型有且仅有一个类型参数，我们才能给它实现 Functor 实例。

举个例子，我们不能给 Either a b 或者 (a, b) 写 fmap 实现，因为它们有两个类型参数。我们也不能给 Bool 或者 Int 写，因为它们没有类型参数。

另外，我们不能给类型定义添加任何约束。这是什么意思呢？为了搞清楚，我们来看一个正常的 data 定义和它的 Functor 实例。

```
-- file: ch10/ValidFunctor.hs
data Foo a = Foo a

instance Functor Foo where
    fmap f (Foo a) = Foo (f a)
```

当我们定义新类型时，我们可以在 data 关键字之后加一个类型约束。

```
-- file: ch10/ValidFunctor.hs
data Eq a => Bar a = Bar a

instance Functor Bar where
    fmap f (Bar a) = Bar (f a)
```

这意味着只有当 a 是 Eq 类型类的成员时，它才能被放进 Foo。然而，这个约束却让我们无法给 Bar 写 Functor 实例。

```
*Main> :l ValidFunctor.hs
[1 of 1] Compiling Main             ( ValidFunctor.hs, interpreted )

ValidFunctor.hs:8:6:
    Illegal datatype context (use DatatypeContexts): Eq a =>
Failed, modules loaded: none.
```

10.5.1 给类型定义加约束不好

给类型定义加约束从来就不是什么好主意。它的实质效果是强迫你给每一个用到这种类型值的函数加类型约束。假设我们现在有一个栈数据结构，我们想通过访问它来看看它的元素是否按顺序排列。下面是数据类型的一个简单实现。

```
-- file: ch10/TypeConstraint.hs
data (Ord a) => OrdStack a = Bottom
                           | Item a (OrdStack a)
                             deriving (Show)
```

如果我们想写一个函数来看看它是不是升序的（即每个元素都比它下面的元素大），很显然，我们需要 Ord 约束来进行两两比较。

```
-- file: ch10/TypeConstraint.hs
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
    | a < b     = isIncreasing rest
    | otherwise = False
isIncreasing _  = True
```

然而，由于我们在类型声明上加了类型约束，它最后也影响到了某些不需要它的地方：我们需要给 push 加上 Ord 约束，但事实上它并不关心栈里元素的顺序。

```
-- file: ch10/TypeConstraint.hs
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

如果你把 Ord 约束删掉，push 定义便无法通过类型检查。

正是由于这个原因，我们之前给 Bar 写 Functor 实例没有成功：它要求 fmap 的类型签名加上 Eq 约束。

我们现在已经尝试性地确定了 Haskell 里给类型签名加类型约束并不是一个好的特性，那有什么好的替代吗？答案很简单：不要在类型定义上加类型约束，在需要它们的函数上加。

在这个例子中，我们可以删掉 OrdStack 和 push 上的 Ord。isIncreasing 还需要，否则便无法调用 (<)。现在我们只在需要的地方加类型约束了。这还有一个更深远的好处：类型签名更准确地表示了每个函数的真

正需求。

大多数 Haskell 容器遵循这个模式。Data.Map 模块里的 Map 类型要求它的键是排序的，但类型本身却没有这个约束。这个约束是通过 insert 这样的函数来表达的，因为这里需要它，在 size 上却没有，因为在这里顺序无关紧要。

10.5.2 fmap 的中缀使用

你经常会看到 fmap 作为操作符使用：

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

也许你感到奇怪，原始的 map 却几乎从不这样使用。

我们这样使用 fmap 一个可能的原因是可以省略第二个参数的括号。括号越少读代码也就越容易。

```
ghci> fmap (1+) ([1,2,3] ++ [4,5,6])
[2,3,4,5,6,7]
```

如果你真的想把 fmap 当做操作符用，Control.Applicative 模块包含了作为 fmap 别名的 (<$>) 操作符。

当我们返回之前写的代码时，我们会发现这对解析很有用。

10.5.3 灵活实例

你可能想给 Either Int b 类型实现 Functor 实例，因为它只有一个类型参数。

```
-- file: ch10/EitherInt.hs
instance Functor (Either Int) where
    fmap _ (Left n) = Left n
    fmap f (Right r) = Right (f r)
```

然而，Haskell 98 类型系统不能保证检查这种实例的约束会终结。非终结的约束检查会导致编译器进入死循环，所以这种形式的实例是被禁止的。

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

EitherInt.hs:2:10:
    Illegal instance declaration for `Functor (Either Int)'
      (All instance types must be of the form (T a1 ... an)
       where a1 ... an are *distinct type variables*,
```

(continues on next page)

(continued from previous page)

```
       and each type variable appears at most once in the instance head.
       Use FlexibleInstances if you want to disable this.)
    In the instance declaration for ‘Functor (Either Int)’
Failed, modules loaded: none.
```

GHC 的类型系统比 Haskell 98 标准更强大。出于可移植性的考虑，默认情况下，它是运行在兼容 Haskell 98 的模式下的。我们可以通过一个编译命令允许更灵活的实例。

```
-- file: ch10/EitherIntFlexible.hs
{-# LANGUAGE FlexibleInstances #-}

instance Functor (Either Int) where
    fmap _ (Left n)  = Left n
    fmap f (Right r) = Right (f r)
```

这个命令内嵌于 LANGUAGE 编译选项。

有了 Functor 实例，我们来试试 Either Int 的 fmap 函数。

```
ghci> :load EitherIntFlexible
[1 of 1] Compiling Main             ( EitherIntFlexible.hs, interpreted )
Ok, modules loaded: Main.
ghci> fmap (== "cheeseburger") (Left 1 :: Either Int String)
Left 1
ghci> fmap (== "cheeseburger") (Right "fries" :: Either Int String)
Right False
```

[Forec 译注：在导入 GHCI 之前，你需要再添加一个 {-# LANGUAGE OverlappingInstances #-} 来通过编译，因为 Data.Either 已经为 Either a 类型定义了对应的 fmap 实例。]

10.5.4 更多关于 Functor 的思考

对于 Functor 如何工作，我们做了一些隐式的假设。把它们说清楚并当成规则去遵守非常有用，因为这会让我们把 Functor 当成统一的、行为规范的对象。规则只有两个，并且非常简单。

第一条规则是 Functor 必须保持身份（preserve identity）。也就是说，应用 fmap id 应该返回相同的值。

```
ghci> fmap id (Node (Leaf "a") (Leaf "b"))
Node (Leaf "a") (Leaf "b")
```

第二条规则是 Functor 必须是可组合的。也就是说，把两个 fmap 组合使用效果应该和把函数组合起来再用 fmap 相同。

```
ghci> (fmap even . fmap length) (Just "twelve")
Just True
ghci> fmap (even . length) (Just "twelve")
Just True
```

另一种看待这两条规则的方式是 Functor 必须保持结构（shape）。集合的结构不应该受到 Functor 的影响，只有对应的值会改变。

```
ghci> fmap odd (Just 1)
Just True
ghci> fmap odd Nothing
Nothing
```

如果你要写 `Functor` 实例，最好把这些规则记在脑子里，并且最好测试一下，因为编译器不会检查我们提到的规则。另一方面，如果你只是用 Functor，这些规则又如此自然，根本没必要记住。它们只是把一些“照我说的做”的直觉概念形式化了。下面是期望行为的伪代码表示。

```
-- file: ch10/FunctorLaws.hs
fmap id       ==  id
fmap (f . g)  ==  fmap f . fmap g
```

10.6 给 Parse 写一个 Functor 实例

对于到目前为止我们研究过的类型而言，`fmap` 的期望行为非常明显。然而由于 `Parse` 的复杂度，对于它而言 `fmap` 的期望行为并没有这么明显。一个合理的猜测是我们要 `fmap` 的函数应该应用到当前解析的结果上，并保持解析状态不变。

```
-- file: ch10/Parse.hs
instance Functor Parse where
    fmap f parser = parser ==> \result ->
                    identity (f result)
```

定义很容易理解，我们来快速做几个实验看看我们是否遵守了 Functor 规则。

首先我们检查身份是否被保持。我们在一次应该失败的解析上试试：从空字符串中解析字节（别忘了 `(<$>)` 就是 `fmap`）。

```
ghci> parse parseByte L.empty
Left "byte offset 0: no more input"
ghci> parse (id <$> parseByte) L.empty
Left "byte offset 0: no more input"
```

不错。再来试试应该成功的解析。

```
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
```

通过观察上面的结果，可以看到我们的 Functor 实例同样遵守了第二条规则，也就是保持结构。失败被保持为失败，成功被保持为成功。

最后，我们确保可组合性被保持了。

```
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
```

通过这个简单的观察，我们的 Functor 实例看起来行为规范。

10.7 利用 Functor 解析

我们讨论 Functor 是有目的的：它让我们写出简洁、表达能力强的代码。回想早先引入的 `parseByte` 函数。在重构 PGM 解析器使之使用新的解析架构的过程中，我们经常想用 ASCII 字符而不是 `Word8` 值。

尽管可以写一个类似于 `parseByte` 的 `parseChar` 函数，我们现在可以利用 `Parse` 的 Functor 属性来避免重复代码。我们的 functor 接受一个解析结果并将一个函数应用于它，因此我们需要的是一个把 `Word8` 转成 `Char` 的函数。

```
-- file: ch10/Parse.hs
w2c :: Word8 -> Char
w2c = chr . fromIntegral

-- import Control.Applicative
parseChar :: Parse Char
parseChar = w2c <$> parseByte
```

我们也可以利用 Functor 来写一个短小的“窥视”函数。如果我们在输入字符串的末尾，它会返回 `Nothing`。否则，它返回下一个字符，但不作处理（也就是说，它观察但不打扰当前的解析状态）。

```
-- file: ch10/Parse.hs
peekByte :: Parse (Maybe Word8)
peekByte = (fmap fst . L.uncons . string) <$> getState
```

定义 parseChar 时用到的提升把戏同样也可以用于定义 peekChar。

```
-- file: ch10/Parse.hs
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

注意到 peekByte 和 peekChar 分别两次调用了 fmap，其中一次还是 (<$>)。这么做的原因是 Parse (Maybe a) 类型是嵌在 Functor 中的 Functor。我们必须提升函数两次使它能进入内部 Functor。

最后，我们会写一个通用组合子，它是 Parse 中的 takeWhile：它在谓词为 True 是处理输入。

```
-- file: ch10/Parse.hs
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p = (fmap p <$> peekByte) ==> \mp ->
               if mp == Just True
               then parseByte ==> \b ->
                    (b:) <$> parseWhile p
               else identity []
```

再次说明，我们在好几个地方都用到了 Functor（doubled up, when necessary）用以化简函数。下面是相同函数不用 Functor 的版本。

```
-- file: ch10/Parse.hs
parseWhileVerbose p =
    peekByte ==> \mc ->
    case mc of
      Nothing -> identity []
      Just c | p c ->
                 parseByte ==> \b ->
                 parseWhileVerbose p ==> \bs ->
                 identity (b:bs)
             | otherwise ->
                 identity []
```

当你对 Functor 不熟悉的时候，冗余的定义应该会更好读。但是，由于 Haskell 中 Functor 非常常见，你很快就会更习惯（包括读和写）简洁的表达。

10.8 重构 PGM 解析器

有了新的解析代码，原始 PGM 解析函数现在变成了这个样子：

```
-- file: ch10/Parse.hs
parseRawPGM =
```

(continues on next page)

(continued from previous page)

```
    parseWhileWith w2c notWhite ==> \header -> skipSpaces ==>&
    assert (header == "P5") "invalid raw header" ==>&
    parseNat ==> \width -> skipSpaces ==>&
    parseNat ==> \height -> skipSpaces ==>&
    parseNat ==> \maxGrey ->
    parseByte ==>&
    parseBytes (width * height) ==> \bitmap ->
    identity (Greymap width height maxGrey bitmap)
  where notWhite = (`notElem` " \r\n\t")
```

下面是定义中用到的辅助函数，其中一些模式现在应该已经非常熟悉了：

```
-- file: ch10/Parse.hs
parseWhileWith :: (Word8 -> a) -> (a -> Bool) -> Parse [a]
parseWhileWith f p = fmap f <$> parseWhile (p . f)

parseNat :: Parse Int
parseNat = parseWhileWith w2c isDigit ==> \digits ->
           if null digits
           then bail "no more input"
           else let n = read digits
                in if n < 0
                   then bail "integer overflow"
                   else identity n

(==>&) :: Parse a -> Parse b -> Parse b
p ==>& f = p ==> \_ -> f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True  _   = identity ()
assert False err = bail err
```

类似于 `(==>)`，`(==>&)` 组合子将解析器串联起来。但右侧忽略左侧的结果。`assert` 使得我们可以检查性质，然后当性质为 `False` 时终止解析并报告错误信息。

10.9 未来方向

本章的主题是抽象。我们发现在函数链中传递显式状态并不理想，因此我们把这个细节抽象出来。在写解析器的时候发现要重复用到一些代码，我们把它们抽象成函数。我们引入了 Functor，它提供了一种映射到参数

化类型的通用方法。

关于解析，我们在第 16 章会讨论一个使用广泛并且灵活的解析库 Parsec。在第 14 章中，我们会再次讨论抽象，我们会发现用 Monad 可以进一步化简这章的代码。

Hackage 数据库中存在不少包可以用来高效解析以 `ByteString` 表示的二进制数据。在写作时，最流行的是 `binary`，它易用且高效。

10.10 练习

1. 给“纯文本”PGM 文件写解析器。
2. 在对“原始”PGM 文件的描述中，我们省略了一个细节。如果头信息中的“最大灰度”值小于 256，那每个像素都会用单个字节表示。然而，它的最大范围可达 65535，这种情况下每个像素会以大端序的形式（最高有效位字节在前）用两个字节来表示。

   重写原始 PGM 解析器使它能够处理单字节和双字节形式。
3. 重写解析器使得它能够区分“原始”和“纯文本”PGM 文件，并解析对应的文件类型。

CHAPTER

ELEVEN

第 11 章：测试和质量保障

构建真实系统意味着我们要关心系统的质量控制，健壮性和正确性。有了正确的质量保障机制，良好编写的代码才能像一架精确的机器一样，所有模块都完成它们预期的任务，并且不会有模棱两可的边界情况。最后我们得到的将是不言自明，正确无疑的代码——这样的代码往往能激发自信。

Haskell 有几个工具用来构建这样精确的系统。最明显的一个，也是语言本身就内置的，是具有强大表达力的类型系统。它使得一些复杂的不变量（invariants）得到了静态保证——绝无可能写出违反这些约束条件的代码。另外，纯度和多态也促进了模块化，易重构，易测试的代码风格。这种类型的代码通常不会出错。

测试在保证代码的正确性上起到了关键作用。Haskell 主要的测试机制是传统的单元测试（通过 HUnit 库）和由它衍生而来的更强机制：使用 Haskell 开源测试框架 QuickCheck 进行的基于类型的“性质”测试。基于性质的测试是一种层次较高的方法，它抽象出一些函数应该普遍满足的不变量，真正的测试数据由测试库为程序员产生。通过这种方法，我们可以用成百上千的测试来检验代码，从而发现一些用其他方法无法发现的微妙的边角情形 (corner cases)，而这对于手写来说是不可能的。

在这章里，我们将会学习如何使用 QuickCheck 来建立不变量，然后重新审视之前章节开发的美观打印器，并用 QuickCheck 对它进行测试。我们也会学习如何用 GHC 的内置代码覆盖工具 HPC 来指导测试过程。

11.1 QuickCheck: 基于类型的测试

为了大概了解基于性质的测试是如何工作的，我们从一个简单的情形开始：你写了一个排序算法，需要测试它的行为。

首先我们载入 QuickCheck 库和其它依赖模块：

```
-- file: ch11/QC-basics.hs
import Test.QuickCheck
import Data.List
```

然后是我们想要测试的函数——一个自定义的排序过程:

```
-- file: ch11/QC-basics.hs
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
    where lhs = filter  (< x) xs
          rhs = filter (>= x) xs
```

这是一个经典的 Haskell 排序实现：它可能不够高效（因为不是原地排序），但它至少展示了函数式编程的优雅。现在，我们来检查这个函数是否符合一个好排序算法应该符合的基本规则。很多纯函数式代码都有的一个很有用的不变量是幂等（idempotency）——应用一个函数两次和一次效果应该相同。对于我们的排序过程，一个稳定的排序算法，这当然应该满足，否则就真的出大错了！这个不变量可以简单地表示为如下性质：

```
-- file: ch11/QC-basics.hs
prop_idempotent xs = qsort (qsort xs) == qsort xs
```

依照 QuickCheck 的惯例，我们给测试性质加上 `prop_` 前缀以和普通代码区分。幂等性质可以简单地用一个 Haskell 函数表示：对于任何已排序输入，再次应用 `qsort` 结果必须相同。我们可以手动写几个例子来确保没什么问题：

[译注，运行之前需要确保自己安装了 QuickCheck 包，译者使用的版本是 2.8.1。]

```
ghci> prop_idempotent []
True
ghci> prop_idempotent [1,1,1,1]
True
ghci> prop_idempotent [1..100]
True
ghci> prop_idempotent [1,5,2,1,2,0,9]
True
```

看起来不错。但是，用手写输入数据非常无趣，并且违反了一个高效函数式程序员的道德法则：让机器干活！为了使这个过程自动化，QuickCheck 内置了一组数据生成器用来生成 Haskell 所有的基本数据类型。QuickCheck 使用 `Arbitrary` 类型类来给（伪）随机数据生成过程提供了一个统一接口，类型系统会具体决定使用哪个生成器。QuickCheck 通常会把数据生成过程隐藏起来，但我们可以手动运行生成器来看看 QuickCheck 生成的数据呈什么分布。例如，随机生成一组布尔值：

[译注：本例子根据最新版本的 QuickCheck 库做了改动。]

```
Prelude Test.QuickCheck.Gen Test.QuickCheck.Arbitrary> sample' arbitrary :: IO [Bool]
[False,False,False,True,False,False,True,True,True,True,True]
```

QuickCheck 用这种方法产生测试数据，然后通过 `quickCheck` 函数把数据传给我们要测试的性质。性质本身的类型决定了它使用哪个数据生成器。`quickCheck` 确保对于所有产生的测试数据，性质仍然成立。由于幂等测试对于列表元素类型是多态的，我们需要选择一个特定的类型来产生测试数据，我们把它作为一个类

型约束写在性质上。运行测试的时候，只需调用 quickCheck 函数，并指定我们性质函数的类型即可（否则的话，列表值将会是没什么意思的 () 类型）：

```
*Main Test.QuickCheck> :type quickCheck
quickCheck :: Testable prop => prop -> IO ()
*Main Test.QuickCheck> quickCheck (prop_idempotent :: [Integer] -> Bool)
+++ OK, passed 100 tests.
```

对于产生的 100 个不同列表，我们的性质都成立——太棒了！编写测试的时候，查看为每个测试生成的实际数据常常会很有用。我们可以把 quickCheck 替换为它的兄弟函数 verboseCheck 来查看每个测试的（完整）输出。现在，来看看我们的函数还可能满足什么更复杂的性质。

11.1.1 性质测试

好的库通常都会包含一组彼此正交而又关联的基本函数。我们可以使用 QuickCheck 来指定我们代码中函数之间的关系，从而通过一组通过有用性质相互关联的函数来提供一个好的库接口。从这个角度来说，QuickCheck 扮演了 API “lint” 工具的角色：它确保我们的库 API 能说的通。

列表排序函数的一些有趣性质把它和其它列表操作关联起来。例如：已排序列表的第一个元素应该是输入列表的最小元素。我们可以使用 List 库的 minimum 函数来指出这个性质：

```
-- file: ch11/QC-basics.hs
import Data.List
prop_minimum xs          = head (qsort xs) == minimum xs
```

测试的时候出错了：

```
*Main Test.QuickCheck> quickCheck (prop_minimum :: [Integer] -> Bool)
*** Failed! Exception: 'Prelude.head: empty list' (after 1 test):
[]
```

当对一个空列表排序时性质不满足了：对于空列表而言，head 和 minimum 没有定义，正如它们的定义所示：

```
-- file: ch11/minimum.hs
head       :: [a] -> a
head (x:_) = x
head []    = error "Prelude.head: empty list"

minimum    :: (Ord a) => [a] -> a
minimum [] =  error "Prelude.minimum: empty list"
minimum xs =  foldl1 min xs
```

因此这个性质只在非空列表上满足。幸运的是，QuickCheck 内置了一套完整的性质编写语言，使我们可以更精确地表述我们的不变量，排除那些我们不予考虑的值。对于空列表这个例子，我们可以这么说：如果列表

非空，那么被排序列表的第一个元素是最小值。这是通过（==>）函数来实现的，它在测试性质之前将无效数据排除在外：

```
-- file: ch11/QC-basics.hs
prop_minimum' xs          = not (null xs) ==> head (qsort xs) == minimum xs
```

结果非常清楚。通过把空列表排除在外，我们可以确定指定性质是成立的。

```
*Main Test.QuickCheck> quickCheck (prop_minimum' :: [Integer] -> Property)
+++ OK, passed 100 tests.
```

注意到我们把性质的类型从 Bool 改成了更一般的 Property 类型（property 函数会在测试之前过滤出非空列表，而不仅是简单地返回一个布尔常量了）。

再加上其它一些应该满足的不变量，我们就可以完成排序函数的基本性质集了：输出应该有序（每个元素应该小于等于它的后继元素）；输出是输入的排列（我们通过列表差异函数（\\）来检测）；被排序列表的最后一个元素应该是最大值；对于两个不同列表的最小值，如果我们把两个列表拼接并排序，这个值应该是第一个元素。这些性质可以表述如下：

```
-- file: ch11/QC-basics.hs
prop_ordered xs = ordered (qsort xs)
    where ordered []       = True
          ordered [x]      = True
          ordered (x:y:xs) = x <= y && ordered (y:xs)

prop_permutation xs = permutation xs (qsort xs)
    where permutation xs ys = null (xs \\ ys) && null (ys \\ xs)

prop_maximum xs         =
    not (null xs) ==>
        last (qsort xs) == maximum xs

prop_append xs ys       =
    not (null xs) ==>
    not (null ys) ==>
        head (qsort (xs ++ ys)) == min (minimum xs) (minimum ys)
```

11.1.2 利用模型进行测试

另一种增加代码可信度的技术是利用模型实现进行测试。我们可以把我们的列表排序函数跟标准列表库中的排序实现进行对比。如果它们行为相同，我们会有更多信心我们的代码的正确的。

```
-- file: ch11/QC-basics.hs
prop_sort_model xs      = sort xs == qsort xs
```

这种基于模型的测试非常强大。开发人员经常会有一些正确但低效的参考实现或原型。他们可以保留这部分代码来确保优化之后的生产代码仍具有相同行为。通过构建大量这样的测试并定期运行（例如每次提交），我们可以很容易地确保代码仍然正确。大型的 Haskell 项目通常包含了跟项目本身大小可比的性质测试集，每次代码改变都会进行成千上万项不变量测试，保证了代码行为跟预期一致。

11.2 测试案例学习：美观打印器

测试单个函数的自然性质是开发大型 Haskell 系统的基石。我们现在来看一个更复杂的案例：为第五章开发的美观打印器编写测试集。

11.2.1 生成测试数据

美观打印器是围绕 `Doc` 而建的，它是一个代数数据类型，表示格式良好的文档。

```
-- file: ch11/Prettify2.hs

data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
         deriving (Show,Eq)
```

这个库本身是由一组函数构成的，这些函数负责构建和变换 `Doc` 类型的值，最后再把它们转换成字符串。

QuickCheck 鼓励这样一种测试方式：开发人员指定一些不变量，它们对于任何代码接受的输入都成立。为了测试美观打印库，我们首先需要一个输入数据源。我们可以利用 QuickCheck 通过 `Arbitrary` 类型类提供的一套用来生成随机数据的组合子集。`Arbitrary` 类型类提供了 `arbitrary` 函数来给每种类型生成数据，我们可以利用它来给自定义数据类型写数据生成器。

```
-- file: ch11/Arbitrary.hs
import Test.QuickCheck.Arbitrary
import Test.QuickCheck.Gen
class Arbitrary a where
    arbitrary   :: Gen a
```

有一点需要注意，函数的类型签名表明生成器运行在 `Gen` 环境中。它是一个简单的状态传递 monad，用来隐藏贯穿于代码中的随机数字生成器的状态。稍后的章节会更加细致地研究 monads，现在只要知道，由于 `Gen` 被定义为一个 monad，我们可以使用 `do` 语法来定义新生成器来访问隐式的随机数字源。`Arbitrary` 类型类提供了一组可以生成随机值的函数，我们可以把它们组合起来构建出我们所关心的类型的数据结构，以便给我们的自定义类型写生成器。一些关键函数的类型如下：

```
-- file: ch11/Arbitrary.hs
    elements :: [a] -> Gen a
    choose   :: Random a => (a, a) -> Gen a
    oneof    :: [Gen a] -> Gen a
```

elements 函数接受一个列表，返回这个列表的随机值生成器。我们稍后再用 choose 和 oneof。有了 elements，我们就可以开始给一些简单的数据类型写生成器了。例如，如果我们给三元逻辑定义了一个新数据类型：

```
-- file: ch11/Arbitrary.hs
data Ternary
    = Yes
    | No
    | Unknown
    deriving (Eq,Show)
```

我们可以给 Ternary 类型实现 Arbitrary 实例：只要实现 arbitrary 即可，它从所有可能的 Ternary 类型值中随机选出一些来：

```
-- file: ch11/Arbitrary.hs
instance Arbitrary Ternary where
    arbitrary     = elements [Yes, No, Unknown]
```

另一种生成数据的方案是生成 Haskell 基本类型数据，然后把它们映射成我们感兴趣的类型。在写 Ternary 实例的时候，我们可以用 choose 生成 0 到 2 的整数值，然后把它们映射为 Ternary 值。

```
-- file: ch11/Arbitrary2.hs
instance Arbitrary Ternary where
    arbitrary     = do
        n <- choose (0, 2) :: Gen Int
        return $ case n of
                    0 -> Yes
                    1 -> No
                    _ -> Unknown
```

对于简单的和类型，这种方法非常奏效，因为整数可以很好地映射到数据类型的构造器上。对于积类型 (如结构体和元组)，我们首先得把积的不同部分分别生成（对于嵌套类型递归地生成），然后再把他们组合起来。例如，生成随机序对：

```
-- file: ch11/Arbitrary.hs
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
    arbitrary = do
        x <- arbitrary
        y <- arbitrary
```

(continues on next page)

(continued from previous page)

```
        return (x, y)
```

现在我们写个生成器来生成 Doc 类型所有不同的变种。我们把问题分解，首先先随机生成一个构造器，然后根据结果再随机生成参数。最复杂的是 union 和 concatenation 这两种情形。

[译注，作者在此处解释并实现了 Char 的 Arbitrary 实例。但由于最新 QuickCheck 已经包含此实例，故此处略去相关内容。]

现在我们可以开始给 Doc 写实例了。只要枚举构造器，再把参数填进去即可。我们用一个随机整数来表示生成哪种形式的 Doc，然后再根据结果分派。生成 concat 和 union 的 Doc 值时，我们只需要递归调用 arbitrary 即可，类型推导会决定使用哪个 Arbitrary 实例：

```
-- file: ch11/QC.hs
instance Arbitrary Doc where
    arbitrary = do
        n <- choose (1,6) :: Gen Int
        case n of
             1 -> return Empty

             2 -> do x <- arbitrary
                     return (Char x)

             3 -> do x <- arbitrary
                     return (Text x)

             4 -> return Line

             5 -> do x <- arbitrary
                     y <- arbitrary
                     return (Concat x y)

             6 -> do x <- arbitrary
                     y <- arbitrary
                     return (Union x y)
```

看起来很直观。我们可以用 oneof 函数来化简它。我们之前见到过 oneof 的类型，它从列表中选择一个生成器（我们也可以用 monadic 组合子 liftM 来避免命名中间结果）：

```
-- file: ch11/QC.hs
instance Arbitrary Doc where
    arbitrary =
        oneof [ return Empty
              , liftM  Char   arbitrary
              , liftM  Text   arbitrary
```

(continues on next page)

(continued from previous page)

```
            , return Line
            , liftM2 Concat arbitrary arbitrary
            , liftM2 Union  arbitrary arbitrary ]
```

后者更简洁。我们可以试着生成一些随机文档，确保没什么问题。

```
*QC Test.QuickCheck> sample' (arbitrary::Gen Doc)
[Text "",Concat (Char '\157') Line,Char '\NAK',Concat (Text "A\b") Empty,
Union Empty (Text "4\146~\210"),Line,Union Line Line,
Concat Empty (Text "|m  \DC4-\DLE*3\DC3\186"),Char '-',
Union (Union Line (Text "T\141\167\&3\233\163\&5\STX\164\145zI")) (Char '~'),Line]
```

从输出的结果里，我们既看到了简单，基本的文档，也看到了相对复杂的嵌套文档。每次测试时我们都会随机生成成百上千的随机文档，他们应该可以很好地覆盖各种情形。现在我们可以开始给我们的文档函数写一些通用性质了。

11.2.2 测试文档构建

文档有两个基本函数：一个是空文档常量 Empty，另一个是拼接函数。它们的类型是：

```
-- file: ch11/Prettify2.hs
empty :: Doc
(<>)  :: Doc -> Doc -> Doc
```

两个函数合起来有一个不错的性质：将空列表拼接在（无论是左拼接还是右拼接）另一个列表上，这个列表保持不变。我们可以将这个不变量表述为如下性质：

```
-- file: ch11/QC.hs
prop_empty_id x =
    empty <> x == x
  &&
    x <> empty == x
```

运行测试，确保性质成立：

```
*QC Test.QuickCheck> quickCheck prop_empty_id
+++ OK, passed 100 tests.
```

可以把 quickCheck 替换成 verboseCheck 来看看实际测试时用的是哪些文档。从输出可以看到，简单和复杂的情形都覆盖到了。如果需要的话，我们还可以进一步优化数据生成器来控制不同类型数据的比例。

其它 API 函数也很简单，可以用性质来完全描述它们的行为。这样做使得我们可以对函数的行为维护一个外部的，可检查的描述以确保之后的修改不会破坏这些基本不变量：

```
-- file: ch11/QC.hs

prop_char c   = char c   == Char c

prop_text s   = text s   == if null s then Empty else Text s

prop_line     = line     == Line

prop_double d = double d == text (show d)
```

这些性质足以测试基本的文档结构了。测试库的剩余部分还要更多工作。

11.2.3 以列表为模型

高阶函数是可复用编程的基本胶水，我们的美观打印库也不例外——我们自定义了 fold 函数，用来在内部实现文档拼接和在文档块之间加分隔符。`fold` 函数接受一个文档列表，并借助一个合并方程（combining function）把它们粘合在一起。

```
-- file: ch11/Prettify2.hs
fold :: (Doc -> Doc -> Doc) -> [Doc] -> Doc
fold f = foldr f empty
```

我们可以很容易地给某个特定 fold 实例写测试。例如，横向拼接（Horizontal concatenation）就可以简单地利用列表中的参考实现来测试。

```
-- file: ch11/QC.hs

prop_hcat xs = hcat xs == glue xs
    where
        glue []     = empty
        glue (d:ds) = d <> glue ds
```

`punctuate` 也类似，插入标点类似于列表的 interspersion 操作（`intersperse` 这个函数来自于 `Data.List`，它把一个元素插在列表元素之间）：

```
-- file: ch11/QC.hs

prop_punctuate s xs = punctuate s xs == intersperse s xs
```

看起来不错，运行起来却出了问题：

```
*QC Test.QuickCheck> quickCheck prop_punctuate
*** Failed! Falsifiable (after 5 tests and 1 shrink):
```

(continues on next page)

(continued from previous page)

```
Empty
[Text "",Text "E"]
```

美观打印库优化了冗余的空文档，然而模型实现却没有，所以我们得让模型匹配实际情况。首先，我们可以把分隔符插入文档，然后再用一个循环去掉当中的 `Empty` 文档，就像这样：

```
-- file: ch11/QC.hs
prop_punctuate' s xs = punctuate s xs == combine (intersperse s xs)
    where
        combine []           = []
        combine [x]          = [x]

        combine (x:Empty:ys) = x : combine ys
        combine (Empty:y:ys) = y : combine ys
        combine (x:y:ys)     = x `Concat` y : combine ys
```

在 **ghci** 里运行，确保结果是正确的。测试框架发现代码中的错误让人感到欣慰——因为这正是我们追求的。

```
*QC Test.QuickCheck> quickCheck prop_punctuate'
+++ OK, passed 100 tests.
```

11.2.4 完成测试框架

[译注：为了匹配最新版本的 QuickCheck，本节在原文基础上做了较大改动。读者可自行参考原文，对比阅读。]

我们可以把这些测试单独放在一个文件中，然后用 QuickCheck 的驱动函数运行它们。这样的函数有很多，包括一些复杂的并行驱动函数。我们在这里使用 `quickCheckWithResult` 函数。我们只需提供一些测试参数，然后列出我们想要测试的函数即可：

```
-- file: ch11/Run.hs
module Main where
import QC
import Test.QuickCheck

anal :: Args
anal = Args
    { replay = Nothing
    , maxSuccess = 1000
    , maxDiscardRatio = 1
    , maxSize = 1000
    , chatty = True
```

(continues on next page)

(continued from previous page)

```
    }

minimal :: Args
minimal = Args
    { replay = Nothing
    , maxSuccess = 200
    , maxDiscardRatio = 1
    , maxSize = 200
    , chatty = True
    }

runTests :: Args -> IO ()
runTests args = do
    f prop_empty_id "empty_id ok?"
    f prop_char "char ok?"
    f prop_text "text ok?"
    f prop_line "line ok?"
    f prop_double "double ok?"
    f prop_hcat "hcat ok?"
    f prop_punctuate' "punctuate ok?"
    where
        f prop str = do
            putStrLn str
            quickCheckWithResult args prop
            return ()

main :: IO ()
main = do
    putStrLn "Choose test depth"
    putStrLn "1. Anal"
    putStrLn "2. Minimal"
    depth <- readLn
    if depth == 1
        then runTests anal
    else runTests minimal
```

[译注：此代码出处为原文下 Charlie Harvey 的评论。]

我们把这些代码放在一个单独的脚本中，声明的实例和性质也有自己单独的文件，它们库的源文件完全分开。这在库项目中非常常见，通常在这些项目中测试都会和库本身分开，测试通过模块系统载入库。

这时候可以编译并运行测试脚本了：

```
$ ghc --make Run.hs
```

(continues on next page)

(continued from previous page)

```
[1 of 3] Compiling Prettify2        ( Prettify2.hs, Prettify2.o )
[2 of 3] Compiling QC               ( QC.hs, QC.o )
[3 of 3] Compiling Main             ( Run.hs, Run.o )
Linking Run ...
$ ./Run
Choose test depth
1. Anal
2. Minimal
2
empty_id ok?
+++ OK, passed 200 tests.
char ok?
+++ OK, passed 200 tests.
text ok?
+++ OK, passed 200 tests.
line ok?
+++ OK, passed 1 tests.
double ok?
+++ OK, passed 200 tests.
hcat ok?
+++ OK, passed 200 tests.
punctuate ok?
+++ OK, passed 200 tests.
```

一共产生了 1201 个测试，很不错。增加测试深度很容易，但为了了解代码究竟被测试的怎样，我们应该使用内置的代码覆盖率工具 HPC，它可以精确地告诉我们发生了什么。

11.3 用 HPC 衡量测试覆盖率

HPC(Haskell Program Coverage) 是一个编译器扩展，用来观察程序运行时哪一部分的代码被真正执行了。这在测试时非常有用，它让我们精确地观察哪些函数，分支以及表达式被求值了。我们可以轻易得到被测试代码的百分比。HPC 的内置工具可以产生关于程序覆盖率的图表，方便我们找到测试集的缺陷。

在编译测试代码时，我们只需在命令行加上 `-fhpc` 选项，即可得到测试覆盖率数据。

```
$ ghc -fhpc Run.hs --make
```

正常运行测试：

```
$ ./Run
```

测试运行时，程序运行的细节被写入当前目录下的.tix 和.mix 文件。之后，命令行工具 `hpc` 用这些文件来展示各种统计数据，解释发生了什么。最基本的交互是通过文字。首先，我们可以在 `hpc` 命令中加上 `report`

选项来得到一个测试覆盖率的摘要。我们会把测试程序排除在外（使用 `--exclude` 选项），这样就能把注意力集中在美观打印库上了。在命令行中输入以下命令：

```
$ hpc report Run --exclude=Main --exclude=QC
 93% expressions used (30/32)
100% boolean coverage (0/0)
    100% guards (0/0)
    100% 'if' conditions (0/0)
    100% qualifiers (0/0)
100% alternatives used (8/8)
100% local declarations used (0/0)
 66% top-level declarations used (10/15)
```

[译注：报告结果可能因人而异。]

在最后一行我们看到，测试时有 66% 的顶层定义被求值。对于第一次尝试来说，已经是很不错的结果了。随着被测试函数的增加，这个数字还会提升。对于快速了解结果来说文字版本的结果还不错，但为了真正了解发生了什么，最好还是看看被标记后的结果（marked up output）。用 `markup` 选项可以生成：

```
$hpc markup Run --exclude=Main --exclude=QC
```

它会对每一个 Haskell 源文件产生一个 html 文件，再加上一些索引文件。在浏览器中打开 `hpc_index.html`，我们可以看到一些非常漂亮的代码覆盖率图表：

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Prettify2	66%	10/15	100%	8/8	93%	30/32
Program Coverage Total	66%	10/15	100%	8/8	93%	30/32

还不错。打开 `Prettify2.hs.html` 可以看到程序的源代码，其中未被测试的代码用黄色粗体标记，被执行的代码用粗体标记。

```
10            | Line
11            | Concat Doc Doc
12            | Union Doc Doc
13           deriving (Show,Eq)
14
15 instance Monoid Doc where
16     mempty = empty
17     mappend = (<>)
18
19
20 (<>) :: Doc -> Doc -> Doc
21 Empty <> y = y
22 x <> Empty = x
23 x <> y = x `Concat` y
```

我们没测 Monoid 实例，还有一些复杂函数也没测。HPC 不会说谎。我们来给 Monoid 类型类实例加个测试，这个类型类支持拼接元素和返回空元素：

```
-- file: ch11/QC.hs
prop_mempty_id x =
    mempty `mappend` x == x
  &&
    x `mappend` mempty == (x :: Doc)
```

在 **ghci** 里检查确保正确：

```
*QC Test.QuickCheck> quickCheck prop_mempty_id
+++ OK, passed 100 tests.
```

我们现在可以重新编译并运行测试了。确保旧的.tix 被删除，否则当 HPC 试图合并两次测试数据时会报错：

```
$ ghc -fhpc Run.hs --make -fforce-recomp
[1 of 3] Compiling Prettify2        ( Prettify2.hs, Prettify2.o )
[2 of 3] Compiling QC               ( QC.hs, QC.o )
[3 of 3] Compiling Main             ( Run.hs, Run.o )
Linking Run ...
$ ./Run
in module 'Main'
Hpc failure: module mismatch with .tix/.mix file hash number
(perhaps remove Run.tix file?)
$rm Run.tix
$./Run
Choose test depth
1. Anal
2. Minimal
2
empty_id ok?
+++ OK, passed 200 tests.
char ok?
+++ OK, passed 200 tests.
text ok?
+++ OK, passed 200 tests.
line ok?
+++ OK, passed 1 tests.
double ok?
+++ OK, passed 200 tests.
hcat ok?
+++ OK, passed 200 tests.
punctuate ok?
+++ OK, passed 200 tests.
```

(continues on next page)

(continued from previous page)

```
prop_mempty_id ok?
+++ OK, passed 200 tests.
```

测试用例又多了两百个，我们的代码覆盖率也提高到了 80%：

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Prettify2	80%	12/15	100%	8/8	100%	32/32
Program Coverage Total	80%	12/15	100%	8/8	100%	32/32

HPC 确保我们在测试时诚实，因为任何没有被覆盖到的代码都会被标记出来。特别地，它确保程序员考虑到各种错误情形，状况不明朗的复杂分支，以及各式各样的代码。有了 QuickCheck 这样全面的测试生成系统，测试变得非常有意义，也成了 Haskell 开发的核心。

CHAPTER

TWELVE

第 12 章：条形码识别

本章我们将用第十章开发的图像分析库来制作一个条形码识别应用。只要用手机的摄像头拍下书的封底，我们就能用这个程序来提取这本书的 ISBN 编号。

12.1 条形码简介

市售绝大多数带有外包装的量产消费品上都有一个条形码。尽管有很多种不同的条形码系统在多个专业领域中被使用，但是在消费品中最典型的条形码系统还是 UPC-A 和 EAN-13 两种。UPC-A 由美国开发，而 EAN-13 最初由欧洲开发。

EAN-13 发表于 UPC-A 之后，它是 UPC-A 的超集。（事实上，尽管 UPC-A 现在在美国依然被广泛使用，但该标准早在在 2005 年已经被官方宣布作废了。) 任何可以识别 EAN-13 条形码的硬件都可以兼容 UPC-A 条形码。这样我们只介绍 EAN-13 一种标准就可以了。

正如其名字所暗示的，EAN-13 描述了一个由 13 个数字组成的序列，该序列可以分为四组：

- 最前的两个数字描述了条形码采用的 *码制*。这两位数字可以标识生产商所在国家, 或者描述该条码的类别，比方说 ISBN(国际标准书号)。

[译注 1：确切的讲，此处的“生产商所在国”实际上是指“为该生产商分配生产商代码的编码管理局所属国家”]

[译注 2：事实上，码制的长度可能为两位甚至更多位，但目前 GS1 General Specifications 中给出的最长的码制也只有 3 位。例如某条形码的类别是 ISBN 码的话，那么它的码制部分应该为 978(后文中给出的实际图中也可以看到)；如果类别是 ISSN(International Standard Serial Number，国际标准连续出版物号)，则码制为 977。其他部分的长度也比本章介绍的要更有弹性，但这些差异并不会影响对本章内容的理解。事实上，这一部分的内容在本章后面的内容中也完全用不到，因为我们在从条码的图形中提取出数字序列后，并没有进一步分离出各个分组乃至查询每个分组表示的具体信息。]

- 接下来的五个数字为厂商代码，由各国的编码规范机构分配。
- 再接下来的 5 个数字是产品代码，由生产厂商决定。(规模较小的生产商可能会使用较长的生产商 ID 和较短的产品 ID，但是两个 ID 加起总是 10 个数字。)

- 最后一个数字为 *校验码 (check digit)*，扫描设备可以通过它来校验扫描到的数字串。

EAN-13 条形码与 UPC-A 条形码的唯一不同在于后者只用一位数字表示码制。EAN-13 条形码通过将码制的第一位数字置零实现对 UPC-A 的兼容。

12.1.1 EAN-13 编码

在考虑怎样解码 EAN-13 条形码之前，我们还是得先了解它是怎样被编码出来的。EAN-13 的编码规则有一点复杂。我们先从计算校验码——即数字串的最后一位开始。

```
-- file: ch12/Barcode.hs
checkDigit :: (Integral a) => [a] -> a
checkDigit ds = productSum `mod` 10 `mod` 10
    where productSum = sum products (mapEveryOther (*3) (reverse ds))

mapEveryOther :: (a -> a) -> [a] -> [a]
mapEveryOther f = zipWith ($) (cycle [f,id])
```

[译注 1：原文对 checkDigit 函数的实现有问题，翻译时用了比较直接的方法修正了代码，并相应的修改了下面一段正文中对代码的描述]

[译注 2：你可能觉得如果把 `mapEveryOther` 中的 `f` 和 `id` 两个列表元素的位置对调的话，就可以省略掉 `checkDigit` 的 where 块中的 `reverse` 过程。事实上这个 reverse 过程是必须的，而且 `f` 和 `id` 也不能对调。因为 EAN-13 的标准规定，“将序列中的最右侧的数字规定为 *奇数位*，从最右侧开始，其余数字被交替记为 *偶数位* 和 *奇数位*，而只有奇数位的数字会被乘以 3。如果采取最开始说的方法，那么假如输入的序列包含偶数个元素的话，那么整个计算过程就是错误的。这一点的重要性在后文会有体现。]

直接看代码应该比文字描述更有助于理解校验码的计算方法。函数从数字串的最右一位数字开始，每隔一位就将该数字乘以 3，其余保持原状。接下来对处理后的列表求和，校验码就是将这个列表的和对 10 取模两次得到的结果。

条形码是一系列定宽的条纹，其中黑色的条纹表示二进制的“1”，白色的条纹表示二进制的“0”。表示相同二进制的条纹值连续排列看起来就是宽一些的条纹。

条形码中的各个二进制位的顺序如下：

- 头部保护序列，固定编码 101。
- 一个由六个数字组成的分组，其中每个数字由 7 个二进制位表示
- 另一个保护序列，固定编码 01010
- 另一个六个数字的组成的分组 (译注：每个数字也由 7 个二进制位表示)
- 尾部保护序列，固定编码 101

左右两个分组中的数字有不同的编码。左侧分组的数字编码包含了校验位 (parity bit)，而校验位编码了条形码的第 13 个数字。

[译注：请注意区分此处所提到的校验位 (parity bit) 以及后面会经常提及的校验码 (check digit)，在本文中，只需要将这个校验位理解为一种只由二进制编码模式 (pattern) 来区分 (而不是“计算”) 的信息，并且了解它只包含奇和偶两种取值即可，没必要深究“哪一位是校验位”。]

12.2 引入数组

在继续前，我们先来看看在本章接下来会用到的所有导入模块。

```
-- file: ch12/Barcode.hs
import Data.Array (Array(..), (!), bounds, elems, indices,
                                ixmap, listArray)

import Control.Applicative ((<$>))
import Control.Monad (forM_)
import Data.Char (digitToInt)
import Data.Ix (Ix(..))
import Data.List (foldl', group, sort, sortBy, tails)
import Data.Maybe (catMaybes, listToMaybe)
import Data.Ratio (Ratio)
import Data.Word (Word8)
import System.Environment (getArgs)
import qualified Data.ByteString.Lazy.Char8 as L
import qualified Data.Map as M

import Parse                    -- from chapter 11
```

条形码的编码过程基本上可以采用表驱动的形式实现，即采用保存了位模式的小规模对照表来决定如何为每个数字进行编码。Haskell 中的基本数据类型——列表和元组——都不太适合构造这种可能涉及随机访问的表。列表需要靠线性遍历才能访问到第 k 个元素。元组没有这个问题，但是 Haskell 的类型系统使我们很难编写一个接受元组和偏移量，返回该元组内指定偏移元素的函数。(我们会在下面的练习中探究为什么这很难。)

说起常见的支持常数时间随机访问的数据结构，数组 (array) 自然首当其冲。Haskell 提供了多种数组数据类型，我们可以利用它们将编码表表示为字符串构成的数组。

最简单的数组类型位于 `Data.Array` 模块，它正是我们要此处要用到的类型。该类型可以表示由任何 Haskell 类型的值构成的数组。与普通的 Haskell 类型一样，该类型的数组都是不可变的。不可变的数组的值只能在它被创建的时候填充一次，之后它的内容就无法被修改了。(标准库也提供了其他的数组类型，其中有一部分是可变的，但我们暂时还不会涉及到它们。)

```
-- file: ch12/Barcode.hs

leftOddList = ["0001101", "0011001", "0010011", "0111101", "0100011",
```

(continues on next page)

(continued from previous page)

```
              "0110001", "0101111", "0111011", "0110111", "0001011"]

rightList = map complement <$> leftOddList
        where complement '0' = '1'
              complement '1' = '0'

leftEvenList = map reverse rightList

parityList = ["111111", "110100", "110010", "110001", "101100",
                  "100110", "100011", "101010", "101001", "100101"]

listToArray :: [a] -> Array Int a
listToArray xs = listArray (0,l-1) xs
        where l = length xs

leftOddCodes, leftEvenCodes, rightCodes, parityCodes :: Array Int String

leftOddCodes = listToArray leftOddList
leftEvenCodes = listToArray leftEvenList
rightCodes = listToArray rightList
parityCodes = listToArray parityList
```

[译注：强烈建议读者在继续阅读前参考本地址中关于 EAN-13 条码二进制编码算法的介绍。如果你已经看过上面的内容，我们也稍微展开说明一下：从代码中给出的 rightList 和 leftEvenList 的计算过程可以发现，即使同一数字有多种编码形式，但他们之间还是有规律可循的。从上面地址中记录了三种数字编码的表中可以看出，每个数字无论采用哪种二进制编码，最终都会被编码为由四个颜色交错的条纹表示的形式，正因如此，每个数字虽然只有 10 种取值却需要 7 位二进制才能表示 (因为像 0001111、0101010 这种序列都是不符合 “四个条纹” 要求的)。而从 Structure of EAN-13 表格中可以看出，左侧分组中第一个数字都采用奇校验编码，而每个采用奇编码的数字编码，其第一个条纹都是白色的 (以二进制 “0” 开头)；右侧分组中的数字编码的第一个条纹都是黑色的 (以二进制 “1” 开头)。有了这些规律，条形码的分析过程可以被大大简化。上述事实在原文中没有给出，因此读者最好能留有印象，会有助于理解之后的一些内容。]

`Data.Array` 模块中的 `listArray` 函数使用列表来填充数组。第一个参数是数组的边界，第二个参数是用来填充数组的列表。

数组有一个独特的性质，它的类型由它所包含数据的类型以及索引的类型共同决定。举例来说，`String` 组成的一维数组的类型为 `Array Int String`，而二维 `String` 数组的类型则是 `Array (Int, Int) String`。

```
ghci> :m +Data.Array
ghci> :type listArray
listArray :: (Ix i) => (i, i) -> [e] -> Array i e
```

创建数组很简单。

```
ghci> listArray (0,2) "foo"
array (0,2) [(0,'f'),(1,'o'),(2,'o')]
```

注意，我们必须在构造数组时显式指定数组的边界。数组边界是闭区间，所以一个边界为 0 和 2 的数组包含 3 个元素。

```
ghci> listArray (0,3) [True,False,False,True,False]
array (0,3) [(0,True),(1,False),(2,False),(3,True)]
ghci> listArray (0,10) "too short"
array (0,10) [(0,'t'),(1,'o'),(2,'o'),(3,' '),(4,'s'),(5,'h'),(6,'o'),(7,'r'),(8,'t'),
↪(9,*** Exception: (Array.!): undefined array element
```

数组构造完成后，我们就可以借助 (!) 运算符通过索引访问元素了。

```
ghci> let a = listArray (0,14) ['a'..]
ghci> a ! 2
'c'
ghci> a ! 100
*** Exception: Error in array index
```

由于数组构造函数允许我们随意指定数组的边界，因此我们就没必要像 C 程序员一样只能用从 0 开始的索引值了。我们可以用任何便于操作的值当作数组的边界。

```
ghci> let a = listArray (-9,5) ['a'..]
ghci> a ! (-2)
'h'
```

索引值的类型可以为 `Ix` 类型的任意成员。也就是说我们就可以用像 `Char` 这种类型作为数组的索引类型。

```
ghci> let a = listArray ('a', 'h') [97..]
ghci> a ! 'e'
101
```

如需创建多维数组，可以用 `Ix` 实例组成的元组来作为数组的索引类型。`Prelude` 模块将拥有 5 个及 5 个以下元素的元组都定义为了 `Ix` 的成员。下面是一个 3 维数组的例子：

```
ghci> let a = listArray ((0,0,0), (9,9,9)) [0..]
ghci> a ! (4,3,7)
437
```

12.2.1 数组与惰性

填充数组的列表包含的元素数目至少要与数组容量相等。如果列表中没有提供足够多的元素，那么程序在运行时就可能发生错误。这个错误发生的时机取决于数组的性质。

我们这里用到的数组类型对数组的元素采用了非严格求值。如果我们想用一个包含三个元素的列表填充一个多于三个元素的数组，那么其余的元素将是未定义的。但是只有我们试图访问超过第三个元素的时候才会发生错误。

```
ghci> let a = listArray (0,5) "bar"
ghci> a ! 2
'r'
ghci> a ! 4
*** Exception: (Array.!): undefined array element
```

Haskell 也提供了严格求值的数组，它们会在上述场景中会有不同的行为。我们将在 “拆箱，抬举，和 bottom” 一章中讨论两种数组之间的取舍。

12.2.2 数组的折叠

bounds 函数返回在创建数组时用来指定边界的元组。indices 函数返回数组中各个索引值组成的列表。我们可以用它们来定义实用的折叠函数，因为 Data.Array 模块本身并没有提供用于数组的折叠函数。

```
-- file: ch12/Barcode.hs
-- | Strict left fold, similar to foldl' on lists.
foldA :: Ix k => (a -> b -> a) -> a -> Array k b -> a
foldA f s a = go s (indices a)
        where go s (j:js) = let s' = f s (a ! j)
                            in s' `seq` go s' js
              go s _ = s

-- | Strict left fold using the first element of the array as its
-- starting value, similar to foldl1 on lists.
foldA1 :: Ix k => (a -> a -> a) -> Array k a -> a
foldA1 f a = foldA f (a ! fst (bounds a)) a
```

你可能很好奇为什么数组模块不预置像折叠函数这么有用的东西。我们会发现一维数组和列表之间有一些明显的相似性。例如，都只有两种自然的方式来折叠他们：从左向右折叠或者从右向左折叠。此外，每次都只能折叠一个元素。

上述这些相似性对于二维数组就已经不再成立了。首先，在二维数组上有意义的折叠方式有很多种。我们也许仍然想要逐个元素地进行折叠，但是对二维数组，还可以逐行折叠或者逐列折叠。其次，就算同样是逐个元素折叠，在二维数组中也不再是只有两种遍历方式了。

换句话讲，对于二维数组来说，有意义操作组合太多了，可也没什么足够的理由选取其中一部分添加到标准库。这个问题只存在于多维数组，所以最好还是让开发人员自己编写合适的折叠函数。从上面的例子也可以看出，这其实没什么难度。

12.2.3 修改数组元素

尽管存在用来“修改”不可变数组的函数，但其实都不怎么实用。以 `accum` 函数为例，它接受一个数组和一个由（索引，元素值）值对构成的列表，返回一个新数组，其中所有在指定索引位置的元素都被替换为指定的元素值。

由于数组是不可变的，所以哪怕只是修改一个元素，也需要拷贝整个数组。哪怕对于中等规模的数组，这种性能开销也可能很快变得难以承受。

Data.Array.Diff 模块中的另一个数组类型 DiffArray，尝试通过保存数组的连续版本之间的变化量来减少小规模修改造成的开销。遗憾的是，在编写本书的时候它的实现还不是很高效，对于实际应用来说，它还是太慢了。

Note: 不要失望

事实上，在 Haskell 中高效地修改数组是 *可能的*——使用 `ST` monad 即可。我们以后会在第二十六章中讨论这个话题。

12.2.4 习题

让我们简单的探索一下用元组替代数组的可行性

1. 编写一个函数，它接受如下两个参数：一个由 4 个元素组成的元组，一个整数。整数参数为 0 的时候，该函数应返回元组中最左侧的元素。整数参数为 1 的时候，返回后一个元素，依此类推。为了使该函数能通过类型检查，你需要对参数的类型做怎样的限制？
2. 写一个与上面类似的函数，第一个参数改为 6 个元素组成的元组。
3. 尝试重构上面的两个函数，让它们共用尽可能多的代码。你能找到多少可以共用的代码？

12.3 生成 EAN-13 条形码

尽管我们的目标是对条形码进行 *解码*，但要是能有一个编码器做参考还是很方便的。这样我们就可以检查 `decode . encode` 的输出是否与输入相同，以此来验证代码的逻辑是否正确。

```haskell
-- file: ch12/Barcode.hs
encodeEAN13 :: String -> String
encodeEAN13 = concat . encodeDigits . map digitToInt

-- | This function computes the check digit; don't pass one in.
encodeDigits :: [Int] -> [String]
encodeDigits s@(first:rest) =
        outerGuard : lefties ++ centerGuard : righties ++ [outerGuard]
                        where (left, right) = splitAt 6 rest
    lefties = zipWith leftEncode (parityCodes ! first) left
    righties = map rightEncode (right ++ [checkDigit s])

leftEncode :: Char -> Int -> String
leftEncode '1' = (leftOddCodes !)
leftEncode '0' = (leftEvenCodes !)

rightEncode :: Int -> String
rightEncode = (rightCodes !)

outerGuard = "101"
centerGuard = "01010"
```

[译注：上面的代码中”where (left, right) = splitAt 6 rest”，在原文中写为了”where (left, right) = splitAt 5 rest”，这是错误的，因为左侧分组有最后一个数字会被分到右侧分组中。]

输入编码器的字符串包含 12 个数字，`encodeDigits` 函数会添加第 13 位数字，即校验码。

[译注：这里所指的“编码器”指的是 `encodeEAN13` 函数。]

条形码的编码分为两组，每组各 6 个数字，两个分组的中间和“外侧”各有一个保护序列。现在里面的两组共 12 个数字已经编码好了，那么剩下的那一个数字哪儿去了?

左侧分组中的每个数字都使用奇校验 (odd parity) 或偶校验 (even parity) 进行编码，具体使用的编码方式取决于数字串中的第一个数字的二进制表示。如果第一个数字中某一位为 0，则左侧分组中对应位置的数字采用偶数校验编码；如果该位为 1，则该对应数字采用奇校验编码。这是一种优雅的设计，它使 EAN-13 条形码可以向前兼容老式的 UPC-A 标准。

12.4 对解码器的约束

在讨论如何解码之前，我们先对可处理的条形码图片的种类做一些实际约束。

手机镜头和电脑摄像头通常会生成 JPEG 图像，但要写一个 JPEG 的解码器又要花上好几章的篇幅，因此我们将图片的解析工作简化为只需要处理 netpbm 文件格式。为此，我们会用到第十章中开发的解析组合子。

我们希望这个解码器能处理用低端手机上那种劣质的定焦镜头拍摄出来的图像。这些图像往往丢焦严重、噪

点多、对比度低，分辨率也很低。万幸，解决这些问题的代码并不难写。我们已经实际验证过本章中的代码，保证它能够识别用货真价实的中低端摄像头拍摄出的实体书上的条形码。

我们会绕过所有的涉及复杂的图像处理的内容，因为那又是一个需要整章篇幅来介绍的课题。我们不会去校正拍摄角度，也不会去锐化那些由于拍摄距离过近导致较窄的条纹模糊不清，或者是拍摄距离过远导致相邻的条纹都糊到一起的图像。

[译注：上面三幅图分别展示了非正对条形码拍摄、拍摄距离过近、拍摄距离过远的情况]

12.5 分而治之

我们的任务是从摄像头拍摄的图像中提取出有效的条形码。这个描述不是特别明确，我们很难以此规划如何一步步展开行动。然而，我们可以先把一个大问题拆分为一系列的独立且易处理的子问题，随后再逐个击破。

- 将颜色数据转换为易于我们处理的形式。
- 从图像中取单一扫描线，并根据该扫描线猜测这一行可能是哪些数字的编码。
- 根据上面的猜测，生成一系列有效解码结果。

我们接下来会看到，上述的子问题中有些还可以进一步分解。

在编写本章给出的代码时你可能会问，这种分而治之的实现方式与最终方案的吻合程度有多高呢？答案是——我们远不是什么图像处理的专家，因此在开始撰写这一章的时候我们也不是很确定最终的解决方案会是什么样子。

关于到底什么样的方案才是可行的，我们事先也做了一些合理的猜测，最后就得到了上面给出的子任务列表。接下来我们就可以开始着手于那些知道如何解决的部分，而在空闲时考虑那些我们没有实际经验的内容。我们当时肯定是不知道有什么既存的算法可用，也没有提前做过什么总体规划。

像这样分解问题有两大优点。首先，通过在熟悉的领域开展实施，可以让人产生“已经开始切实解决问题”的积极情绪，哪怕现在做的以后不见得用得上也是一样。其次，在处理某个子问题时，我们可能会发现可以将它进一步分解为多个我们熟悉解决思路的子问题。我们可以继续专注于其中简单的部分，而把那些还没来得及彻底想通的部分延后，一个一个的处理上面子问题列表中的项。最后，等我们把不熟悉的和未解决的问题都搞定了，对最终的解决方案也就能心中有数了。

12.6 将彩色图像转换为更容易处理的形式

这个解码器处理的对象是条形码，条形码的本质就是连续的黑白条纹序列，而我们还想让这个解码器尽可能的简单，那么最容易处理的表示形式就是黑白图像——它的每个像素都是非黑即白的。

[译注：原文此处提到的图像是 monochrome image(单色图像)，其中 monochrome(单色的) 一词虽然经常被当作 black and white 或 grayscale 的同义词使用 (在图像领域)，但实际上这个词表达比了“黑白”更广泛的颜色范围，单色图像可选的颜色并不限于黑和白，例如夜视设备生成的图像，它同样是一种单色图像，但是它生成的图像通常采用绿色为前景色。换句话说，黑白图像只是单色图像的一种。详情参见英文维基词条 *monochrome*

。由于本章节中对图像的处理的确是要将图像处理为只有黑白两种颜色像素的图像 (也确实不该考虑其他的颜色组合)，因此本章中的 monochrome image 都译为黑白图像。]

12.6.1 分析彩色图像

我们之前说过，我们的解码器将只支持 netpbm 图像。netpbm 彩色图像格式只稍微比第十章中处理的灰度图像格式复杂一点点。其头部的识别串为 “P6”，头部的其余部分都和灰度格式完全一样。在图像文件的主体部分，每个像素都由 3 个字节表示，分别对应红、绿、蓝 3 个颜色分量。

我们将图像数据表示为像素构成的二维数组。为了帮我们积累数组的使用经验，此处将完全采用数组实现。但实际上对于这个应用来说，我们用 “列表的列表” 代替数组也可以。因为数组在这里的优势不明显，它的唯一的好处就是很方便取整行。

```
-- file: ch12/Barcode.hs
type Pixel = Word8
type RGB = (Pixel, Pixel, Pixel)

type Pixmap = Array (Int,Int) RGB
```

我们定义了一些类型的同义词来提高类型签名的可读性。

Haskell 为数组提供了相当高的自由度，我们必须维数组选择一种合适的表示形式。这里我们将采取保守的方案，并遵守一个普遍的约定：索引值从 0 开始。我们不需要显式的存储图像的尺寸，因为用 `bounds` 函数可以从数组直接提取尺寸。

最终的解析器实现相当的简短，这都多亏了我们在第十章中开发的组合子。

```
-- file: ch12/Barcode.hs
parseRawPPM :: Parse Pixmap
parseRawPPM =
    parseWhileWith w2c (/= '\n') ==> \header -> skipSpaces ==>&
    assert (header == "P6") "invalid raw header" ==>&
    parseNat ==> \width -> skipSpaces ==>&
    parseNat ==> \height -> skipSpaces ==>&
    parseNat ==> \maxValue ->
    assert (maxValue == 255) "max value out of spec" ==>&
    parseByte ==>&
    parseTimes (width * height) parseRGB ==> \pxs ->
    identity (listArray ((0,0),(width-1,height-1)) pxs)

parseRGB :: Parse RGB
parseRGB = parseByte ==> \r ->
        parseByte ==> \g ->
        parseByte ==> \b ->
```

(continues on next page)

(continued from previous page)

```
        identity (r,g,b)

parseTimes :: Int -> Parse a -> Parse [a]
parseTimes 0 _ = identity []
parseTimes n p = p ==> \x -> (x:) <$> parseTimes (n-1) p
```

上面的代码中唯一需要注意的是 parseTimes 函数，它会将一个分析器调用指定的次数，最后构造出一个分析结果组成的列表。

12.6.2 灰度转换

我们需要将彩色图像的色彩数据转换为黑白的形式。其中一个步骤是将色彩数据转换为灰度数据。有一个简单并广泛应用的公式[29] 可以将彩色图像转换为灰度图像，该公式基于每个色彩通道的相对亮度来计算灰度信息。

```
-- file: ch12/Barcode.hs
luminance :: (Pixel, Pixel, Pixel) -> Pixel
luminance (r,g,b) = round (r' * 0.30 + g' * 0.59 + b' * 0.11)
    where r' = fromIntegral r
          g' = fromIntegral g
          b' = fromIntegral b
```

Haskell 中的数组都是 Functor 类型类的成员，所以我们可以直接用 fmap 函数一次性将整张图片或者单行扫描线从彩色格式转为灰度格式。

```
-- file: ch12/Barcode.hs
type Greymap = Array (Int,Int) Pixel

pixmapToGreymap :: Pixmap -> Greymap
pixmapToGreymap = fmap luminance
```

上面给出来的 pixmapToGreymap 函数只是拿来举个例子，因为我们只需要检查图片的部分行来提取可能存在的条形码，也就没必要在以后用不到的数据上做多余的转换工作了。

12.6.3 灰度二值化和类型安全

接下来要处理的子问题是如何将灰度图像转换为二值图像，二值图像的每个像素都只处于“打开”或“关闭”两种状态之一。

..FIXME: 此处的 digit 做 value 译.. FIXME: 本段里的 bit 应该是指 pixel

[29] 该公式在 ITU-R Recommendation 601 中首次提及。

在一个图像处理程序中通常需要同时处理大量的数值，有时为了方便，可能会把同一种数值类型用于不同的目的。例如，我们只要约定数字 1 表示一个像素处于“打开”状态，而 0 表示一个像素处于“关闭”状态，就可以直接使用 `Pixel` 类型表示像素的开/关状态了。

然而，这种做法有潜在的迷惑性。如果想知道某个特定的 `Pixel` 类型的值究竟是代表一个数值还是一个“开”/“关”状态，就不能靠类型签名轻易确定了。在某些场景中，我们可能很轻易的就使用了错误类型的数值，而且编译器也不会检查到错误，因为这个值的类型与签名指定的类型是吻合的。

我们可以尝试通过引入类型别名来解决这个问题。和前文中把 `Pixel` 声明为 `Word8` 的别名一样，我们也可以把 `Bit` 类型声明为 `Pixel` 类型的别名。这么做虽然可能提高一定的可读性，但类型别名还是不会让编译器替我们做什么有用的工作。

编译器将把 Pixel 和 Bit 当做完全相同的类型，所以就算把 `Pixel` 类型的值 253 传给一个接受 Bit 类型的值 (0 或 1) 的函数，编译器也不会报错。

如果另外定义一个单色 (monochrome) 类型，编译器就会阻止我们像上述的例子那样意外混用不同类型。

```
-- file: ch12/Barcode.hs
data Bit = Zero | One
           deriving (Eq, Show)

threshold :: (Ix k, Integral a) => Double -> Array k a -> Array k Bit
threshold n a = binary <$> a
    where binary i | i < pivot  = Zero
                   | otherwise  = One
          pivot    = round $ least + (greatest - least) * n
          least    = fromIntegral $ choose (<) a
          greatest = fromIntegral $ choose (>) a
          choose f = foldA1 $ \x y -> if f x y then x else y
```

`threshold` 函数会先计算输入数组中的最大值和最小值，结合参数提供的一个介于 0 到 1 之间的阈值，求出一个“枢轴”(pivot) 值。对于数组中的每个元素，如果该元素的值小于这个枢轴值，则计算结果为 `Zero`，否则结果为 `One`。注意到这里我们用到了一个在数组的折叠一节中编写的折叠函数。

[译注：针对这段代码，需要指出一个关于 RGB 颜色模式的注意点。在前面我们通过 `luminance` 函数将要给彩色图片中的像素中的三个颜色分量转换为了一个灰度值，这个灰度值可以理解为“当一个 RGB 颜色的三个分量同时取该值的时候该像素的颜色”。在这个定义下，纯白色的灰度值为 255，随着灰度值越来越小，这个颜色将会呈现为越来越深的灰色，直到灰度值为 0，此时该像素为纯黑色。可见这里有一个比较反直觉的地方，即“灰度值越大，颜色越浅”。这个特征反映到二值化函数 `threshold` 的返回值中就是“深色的像素返回值为 `Zero`，浅色的像素返回值为 `One`”。]

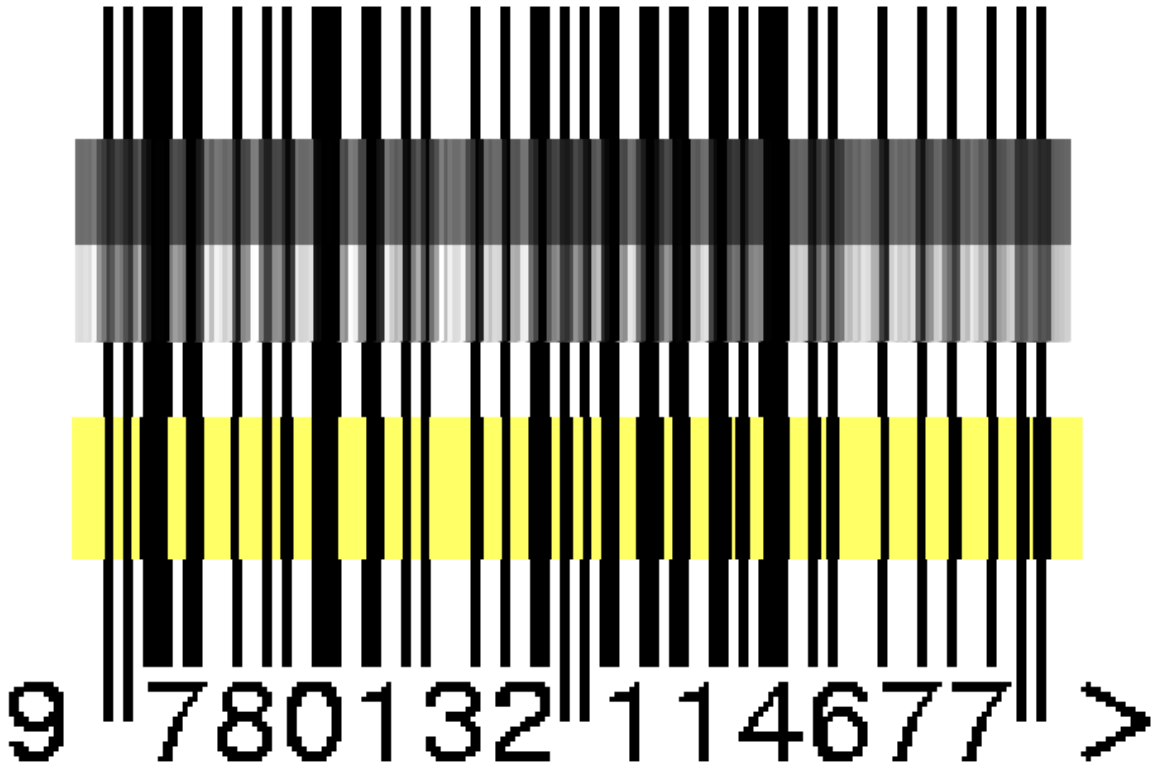
12.7 我们对图像做了哪些处理？

让我们暂时回过头来想一想，在我们把图像从彩色转换为黑白的过程中，到底对它做了哪些处理。下面是一张用 VGA 分辨率的摄像头捕获的图像。我们做的就是把这个图像“压缩”到只剩下足够识别条形码的内容。

这之中编码的数字序列，9780132114677，被打印在了条形码的下方。左侧分组编码了数字串 780132，第一位数字 9 也被隐含在该组每个数字编码的奇偶性中。右侧分组编码了数字串 114677，其中最后一个 7 为校验码。下面是这个条形码的清晰版本，是我们在一个提供免费条形码图片生成服务的网站得到的。

[译注：本段中所说的“奇偶性”指的是“左侧分组中的某个数字是采用的奇校验编码还是偶校验编码”，为了避免🄵嗦，在后文中都会采用这个说法；本章中并没有涉及自然数中“奇偶性”的概念，请注意与之区分。]

我们从捕获的图像中选择了一个行。为了便于观察，我们将这一行垂直拉伸，然后把它放在“完美图像”的上方并且做了拉伸让两幅图像对齐。



图中用深灰色标出的是经过亮度转换处理的行。可以看到，这部分图像对比度低，清晰度也很差，有多处模糊和噪点。浅灰色标出的部分来自于同一行，但是对比度经过了调整。

更靠下的一小段的显示了对亮度转换过的行进行二值化后的效果。你会发现有些条纹变得更粗了而有些更细了，有些条纹还稍微左移或者右移了一点距离。

[译注：“亮度转换”即上面的 `luminance` 函数进行的处理，将彩色图像转换为灰度图像；“二值化”即上面的 `threshold` 函数进行的处理，将灰度图像中的像素进行二值化。]

可见，要在具有这些缺陷的图像中找出匹配结果显然不是随随便便就能做到的。我们必须让代码足够健壮以应对过粗、过细或者位置有偏差的条纹。而且条纹的宽度取决于摄像头与书的距离，我们也不能对它做任何假设。

12.8 寻找匹配的数字

我们首先要面对的问题，是如何在某个 可能编码了数字的位置把这个数字找出来。在此，我们要做一些简单的假设。第一个假设是我们处理的对象是图像中的单一行，第二个假设是我们明确知道条形码左边缘位置，这个位置即条形码的起始位置。

12.8.1 游程编码

我们如何解决线条宽度的问题呢。答案就是对图像数据进行游程编码 (run length encode)。

```
-- file: ch12/Barcode.hs
type Run = Int
type RunLength a = [(Run, a)]

runLength :: Eq a => [a] -> RunLength a
runLength = map rle . group
    where rle xs = (length xs, head xs)
```

group 函数会把一个列表中所有连续的相同元素分别放入一个子列表中。

```
group [1,1,2,3,3,3,3]
[[1,1],[2],[3,3,3,3]]
```

我们的 runLength 函数将 (group 返回的列表中的) 每个子列表表示为子列表长度和首个元素组成的对。

```
ghci>  let bits = [1,1,0,0,1,1,0,0,1,1,1,1,1,1,0,0,1,1,1,1]
ghci>  runLength bits
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
[(2,1),(2,0),(2,1),(2,0),(6,1),(2,0),(4,1)]
```

[译注：上述 ghci 输出的最后一行的列表中，每一个“长度-值”对就是一个“游程”]

由于我们进行游程编码的数据只包含 0 和 1，因此编码的数字只会在 0 和 1 两个值之间变化。既然这样，我们就可以只保留长度而丢弃被编码数字，而不会丢失任何有用的信息。

```
-- file: ch12/Barcode.hs
runLengths :: Eq a => [a] -> [Run]
runLengths = map fst . runLength
```

```
ghci> runLengths bits
[2,2,2,2,6,4,4]
```

上面给出的位模式并不是我们随便编出来的；而是上面我们捕获的图像中的某一行里面的编码的左侧保护序列和第一个编码数字。如果我们丢弃表示保护序列的条纹，游程编码后的值就是 [2, 6, 4, 4]。我们怎样在“引入数组”一节中的编码表中找到匹配的位模式呢?

[译注 1：此处稍微做一下展开。首先，在灰度二值化和类型安全 一节中我们已经知道，Zero 才是代表黑色条纹的值。此处的 0 是与 Zero 对应的，它同样表示黑色条纹，相应的，1 表示白色条纹，与*EAN-13 编码* 一

节中介绍 EAN-13 条码编码格式时约定的 0/1 所代表的颜色相反，再次请读者留心这点，因为接下来的内容都会遵守这个约定。]

[译注 2：如果你实在看不出这个游程编码的值是如何表示之前捕获的图像中数字的，这里我们来详细解释一下。前文说过，由于条纹在照片中的实际宽度受到拍摄距离等因素的影响，因此我们不能对其有任何假设。而且一般来说，条形码中表示每 1 位的条纹所占的宽度几乎不可能只有 1 个像素，而是会由纵向上的复数个像素表示一位。比方说上面给出的序列，很明显就是用了两个像素来表示每个 1 个二进制位，其实际表示的二进制序列为 “01010001100” (0 为黑色，1 为白色)，当” 以 1 为黑色，0 为白色” 时，该序列即 “10101110011”。这样就可以看出，该序列就是由 “101” (左侧保护序列) 和 “0111001” (倒数第 2 位识别错误的 “7” 的奇校验编码) 以及下一位数字的第一个二进制位” 0” 组成的了。]

12.8.2 缩放游程，查找相近的匹配

一个合理的方法是缩放这些游程编码值，让它们的和为 1。我们将使用 `Ratio Int` 类型替代一般的 `Double` 类型来保存这些缩放后的值，因为 `Ratio` 值在 **ghci** 的输出中可读性更好。这点可以为交互式调试与开发提供方便。

```
-- file: ch12/Barcode.hs
type Score = Ratio Int

scaleToOne :: [Run] -> [Score]
scaleToOne xs = map divide xs
    where divide d = fromIntegral d / divisor
          divisor = fromIntegral (sum xs)
-- A more compact alternative that "knows" we're using Ratio Int:
-- scaleToOne xs = map (% sum xs) xs

type ScoreTable = [[Score]]

-- "SRL" means "scaled run length".
asSRL :: [String] -> ScoreTable
asSRL = map (scaleToOne . runLengths)

leftOddSRL = asSRL leftOddList
leftEvenSRL = asSRL leftEvenList
rightSRL = asSRL rightList
paritySRL = asSRL parityList
```

我们定义了类型别名 `Score`，这样其余的大部分代码就不需要关心 `Score` 底层的类型是什么。当我们的代码开发完毕，一头埋进 **ghci** 做后续调试的时候，只要我们愿意，我们还是能把 “`Score`” 对应的底层类型改为 `Double`，而不需要修改其它代码。

我们可以用 `scalarToOne` 函数来缩放我们所要寻找的数字序列。我们解决了拍摄距离所导致的条纹宽度不能确定的问题。现在，在缩放后的游程编码表和从图像中的提取出游程编码序列间应该有十分接近的匹配。

接下来的问题是如何将直观感觉上的“十分接近”转化为对“足够接近”的度量。给出两个缩放过的长度序列，我们可以像下面这样计算出一个大概的“差异度”(distance)。

精确匹配的两个值之间的差异度是 0，匹配程度越低，差异度的值就越大。

```
ghci> let group = scaleToOne [2,6,4,4]
ghci> distance group (head leftEvenSRL)
13%28
ghci> distance group (head leftOddSRL)
17%28
```

对给定的一个经过缩放的游程编码表，我们从中选择与输入序列最接近的几个匹配结果。

```
-- file: ch12/Barcode.hs
bestScores :: ScoreTable -> [Run] -> [(Score, Digit)]
bestScores srl ps = take 3 . sort $ scores
    where scores = zip [distance d (scaleToOne ps) | d <- srl] digits
          digits = [0..9]
```

12.8.3 列表推导式

我们在上面的例子中引入的新表示法叫做 *列表推导式 (list comprehension)*，列表推导式可以以一个或多个列表为基础创建新列表。

```
ghci> [ (a,b) | a <- [1,2], b <- "abc" ]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

竖线右侧的每一个 *生成器表达式 (generator expression)* 组合，都会代入到竖线左侧的表达式中求值。生成表达式绑定了左侧的变量 a，a 又用“<-”绑定到右侧的元素列表。正如上面的例子展示的，生成表达式的组合将按照深度优先的顺序遍历：先是第一个列表的第一个元素分别与第二个列表中的每个元素分别组合，以此类推。

我们还可以在列表推导式的右侧为生成器指定 guard。guard 是一个 `Bool` 表达式。如果 guard 的值为 `False`，则该元素被跳过。

```
ghci> [ (a,b) | a <- [1..6], b <- [5..7], even (a + b ^ 2) ]
[(1,5),(1,7),(2,6),(3,5),(3,7),(4,6),(5,5),(5,7),(6,6)]
```

其中还可以用 `let` 表达式绑定本地变量 (local variable)。

```
ghci> let vowel = (`elem` "aeiou")
ghci> [ x | a <- "etaoin", b <- "shrdlu", let x = [a,b], all vowel x ]
["eu","au","ou","iu"]
```

如果生成器表达式中的某个模式匹配失败了，那么也不会有错误发生，只会跳过未匹配的列表元素。

```
ghci> [ a | (3,a) <- [(1,'y'),(3,'e'),(5,'p')] ]
"e"
```

列表推导式功能强大用法简洁，但可能不太容易看懂。如果能小心使用，它也可以让我们的代码更容易理解。

```
-- file: ch12/Barcode.hs
-- our original
zip [distance d (scaleToOne ps) | d <- srl] digits

-- the same expression, expressed without a list comprehension
zip (map (flip distance (scaleToOne ps)) srl) digits

-- the same expression, written entirely as a list comprehension
[(distance d (scaleToOne ps), n) | d <- srl, n <- digits]
```

12.8.4 记录匹配数字的奇偶性

对左侧分组数字的每一个匹配，我们必须记录它是在奇校验编码表还是偶校验编码表中匹配到的。

```
-- file: ch12/Barcode.hs
data Parity a = Even a | Odd a | None a
                deriving (Show)

fromParity :: Parity a -> a
fromParity (Even a) = a
fromParity (Odd a) = a
fromParity (None a) = a

parityMap :: (a -> b) -> Parity a -> Parity b
parityMap f (Even a) = Even (f a)
parityMap f (Odd a) = Odd (f a)
parityMap f (None a) = None (f a)

instance Functor Parity where
    fmap = parityMap
```

我们将匹配到的数字包装在该数字的实际编码所采用的奇偶性内，并且使它成为一个 `Functor` 实体，这样我们就可以方便的操作奇偶编码值 (parity-encoded values) 了。

[译注：此处所说的“奇偶编码值”可以理解为“对同一个数字同时具有奇校验编码和偶校验编码两种形式的编码值”(即左分组中所有的编码值都是“奇偶编码值”)，为了简化描述，后文也会采用这种简称，请读者留意。]

我们可能需要对奇偶编码值按它们包含的数字进行排序。`Data.Function` 模块提供的一个好用的组合子

on 可以帮助我们实现这个功能。

```
-- file: ch12/Barcode.hs
on :: (a -> a -> b) -> (c -> a) -> c -> c -> b
on f g x y = g x `f` g y

compareWithoutParity = compare `on` fromParity
```

它的作用可能不是很明确，你可以试着去想象这样一个函数：它接受两个参数 f 和 g，返回值是一个函数，这个返回的函数也有两个参数，分别为 x 和 y 。on 将 g 分别对 x 和 y 应用，然后将 f 应用于这两个结果 (所以它的名字叫 on)。

把匹配数字装入奇偶性的方法一目了然。

```
-- file: ch12/Barcode.hs
type Digit = Word8

bestLeft :: [Run] -> [Parity (Score, Digit)]
bestLeft ps = sortBy compareWithoutParity
                  ((map Odd (bestScores leftOddSRL ps)) ++
                           (map Even (bestScores leftEvenSRL ps)))

bestRight :: [Run] -> [Parity (Score, Digit)]
bestRight = map None . bestScores rightSRL
```

一旦在奇校验表或偶校验表里找到了左侧分组某个编码的几个最佳匹配，我们就可以将他们按照匹配的质量排序。

12.8.5 键盘惰性

定义 Parity 类型时, 我们可以使用 haskell 的记录 (record) 语法来避免手写 formParity 函数。也就是说，可以这么写：

```
-- file: ch12/Barcode.hs
data AltParity a = AltEven {fromAltParity :: a}
                 | AltOdd  {fromAltParity :: a}
                 | AltNone {fromAltParity :: a}
                   deriving (Show)
```

那我们为什么没这么做呢？答案说起来有些丢人，而且与 **ghci** 的交互调试有关。当我们告诉 GHC 让它自动把一个类型派生为 Show 的实体时，GHC 会根据我们是否使用记录语法来定义这个类型而生成不同的代码。

```
ghci> show $ Even 1
"Even 1"
```

(continues on next page)

(continued from previous page)

```
ghci> show $ AltEven 1
"AltEven {fromAltParity = 1}"
ghci> length . show $ Even 1
6
ghci> length . show $ AltEven 1
27
```

使用记录语法定义生成的 Show 实体明显很“🅵嗦”，同时这也会给调试带来很大的干扰。比方说在我们检查 **ghci** 输出的奇偶编码值列表的时候，这样的输出结果会特别长以至于我们不得不一行行地扫读输出结果。

当然我们可以手动实现干扰更少的 Show 实体。避开记录语法写起来也更简洁，而且通过编写我们自己的 `formParity` 函数可以让 GHC 帮我们派生输出更简洁的 `Show` 实例。其实也并不是非这么做不可，但是程序员的惰性有时也的确会为代码引入一些特别的做法。

12.8.6 列表分块

使用列表时常常需要对它进行分块 (chunk)。例如，条形码中的每个数字都由四个连续的数字编码而成。我们可以将表示一个行的列表转换为如下这种包含四个元素的列表组成的列表。

```
-- file: ch12/Barcode.hs
chunkWith :: ([a] -> ([a], [a])) -> [a] -> [[a]]
chunkWith _ [] = []
chunkWith f xs = let (h, t) = f xs
                 in h : chunkWith f t

chunksOf :: Int -> [a] -> [[a]]
chunksOf n = chunkWith (splitAt n)
```

像这种需要手写泛型的列表操作函数的情况比较罕见。因为一般在 `Data.List` 模块里翻一翻就能找到完全符合要求或者基本满足需要的函数。

12.8.7 生成候选数字列表

这几个辅助函数一旦就绪，为每个数字分组生成候选匹配的函数也就很容易搞定了。首先，我们先得做一些前期的检查，来确定这些匹配是否都是有意义的。只有以黑色 (`Zero`) 条纹开始，并且条纹数量足够多的游程列表才是有意义的。下面是这个函数中的前几个等式。

```
-- file: ch12/Barcode.hs
candidateDigits :: RunLength Bit -> [[Parity Digit]]
candidateDigits ((_, One):_) = []
candidateDigits rle | length rle < 59 = []
```

[译注：代码中的 59 表示条形码中的条纹数，它是这样求出的：3(左侧保护序列 101)+4x6(每个数字的条纹数目 4x 左侧分组的数字数)+5(两个分组中间的保护序列 10101)+4x6(同左分组)+3(右侧保护序列) = 59。]

只要任意一次 `bestLeft` 或 `bestRight` 的应用得到一个空列表，我们都不能返回有效结果。否则，我们将丢弃 `Score` 值，返回一个由标记了编码奇偶性的候选数字列表组成的列表。外部的列表有 12 个元素，每个元素都代表条形码中的一个数字。子列表中的每个数字都根据匹配质量排序。

下面给出这个函数的其余部分

```
-- file: ch12/Barcode.hs
candidateDigits rle
    | any null match = []
    | otherwise      = map (map (fmap snd)) match
  where match = map bestLeft left ++ map bestRight right
        left = chunksOf 4 . take 24 . drop 3 $ runLengths
        right = chunksOf 4 . take 24 . drop 32 $ runLengths
        runLengths = map fst rle
```

我们看一看从上面图像中提取出的每个线条分组 (表示一个数字的四个线条算作一组) 对应的候选数字。

ghci> :type input input :: [(Run, Bit)] ghci> take 7 input [(2,Zero),(2,One),(2,Zero),(2,One),(6,Zero),(4,One),(4,Zero)] ghci> mapM_ print $ candidateDigits input [Even 1,Even 5,Odd 7,Odd 1,Even 2,Odd 5] [Even 8,Even 7,Odd 1,Odd 2,Odd 0,Even 6] [Even 0,Even 1,Odd 8,Odd 2,Odd 4,Even 9] [Odd 1,Odd 0,Even 8,Odd 2,Even 2,Even 4] [Even 3,Odd 4,Odd 5,Even 7,Even 0,Odd 2] [Odd 2,Odd 4,Even 7,Even 0,Odd 1,Even 1] [None 1,None 5,None 0] [None 1,None 5,None 2] [None 4,None 5,None 2] [None 6,None 8,None 2] [None 7,None 8,None 3] [None 7,None 3,None 8]

12.9 没有数组和散列表的日子

在命令式语言中，数组的地位就像是 Haskell 中的列表或元组，不可或缺。命令式语言中的数组通常是可变的，即我们随时可以修改数组中的元素值，我们对这点也习以为常。

正如我们在“修改数组元素”一节中提到的一样，Haskell 数组并不是可变的。这意味着如果要“修改”数组中的单个元素，整个数组都要被复制一次，被修改的元素将在复制的过程中被设置为新的值。显然，以这种方法“修改”数组不可能在性能比拼中获胜。

可变数组还被用来构建另一种命令式语言常见数据结构——散列表 (hash table)。在散列表的典型实现中，数组扮演了“脊柱”的角色：数组中的每个元素都是一个列表。在散列表中添加一个元素时，我们通过对元素进行散列 (hash)，确定这个元素在数组中的偏移，然后修改位于这个偏移的列表，把这个元素添加进去。

如果构造散列表所使用的数组不是可变的，那么如果要更新一个散列表的话，我们就不得不创建一个新的数组——先复制原数组，然后把一个新的列表放到由散列值确定的偏移位置上。我们不需要复制其他偏移位置上的列表，但是由于必须复制这个“脊柱”，性能方面已经遭到了致命打击。

不可变的数组一下就让我们的工具箱中两种命令式语言中的典型数据结构直接下岗。可见数组在纯 Haskell

代码中的确不像在许多别的语言中那么有用。不过，有很多涉及数组的代码都只是在构建阶段更新数组，构建完成后都将其当作只读的数组来使用。

[译注：此处的“构建阶段 (build phase)”并不仅限于用 `listArray` 函数或者直接调用构造器函数，还包括“原始的”数组生成完毕，进行后续的值设置的过程，这些过程中可能包含对数组的修改 (以及底层的复制) 操作。]

12.9.1 答案的森林

但事实上，用不了可变的数组和散列表并没有想象中那么悲剧。数组和散列表经常被用作由键索引的值的集合，而在 Haskell 中，我们使用 树来实现这个功能。

在 Haskell 中实现一个简单的树类型非常简单。不仅如此，更实用的树类型实现起来也是出奇的简单。比方说红黑树。红黑树这种自平衡结构，就是因为其平衡算法出了名的难写，才让几代 CS 在校生闻风丧胆。

综合运用 Haskell 的代数数据类型组合、模式匹配、guard 等特性可以把最可怕的平衡操作的代码缩减至只有短短几行。但是我们先不急着构造树类型，先来关注为什么它们在纯函数式语言中特别有用。

对函数式程序员来说，树的吸引力在于修改代价低。我们不用打破不可变原则：树就和其他东西一样不可变。然而，我们修改一棵树的时候，可以在新旧两棵树之间共享大部分的结构。举例来说，有一颗有 10000 个节点的树，我们可能想要在里面添加或者移除一个节点，这种情况下，新旧两棵树能够共享大约 9985 个节点。换句话说，每次更新树的时候所需要修改的元素数目取决于树的高度，或者说是节点数的对数。

Haskell 标准库提供了两种采用平衡树实现的集合类型：Data.Map 用于键/值对，`Data.Set` 用于集合。鉴于在下一节会用到 `Data.Map` ，我们就先简要地介绍一下这个模块。`Data.Set` 与 `Data.Map` 很相似，相信你应该也能很快掌握。

Note: 关于性能

一个具有良好实现的纯函数式树结构与散列表在性能上应该是可以一较高下的。你不应该在你的代码会付出性能代价的假设下实现树类型。

12.9.2 map 简介

`Data.Map` 模块提供了参数化类型 `Map k a` ，将键类型 k 映射到关联值类型 a。尽管其内部为一个 size-balanced tree，但是它的实现对我们是不可见的。

[译注 1：Size-Balanced Tree（SBT）是一种通过大小 (Size) 域来保持平衡的二叉搜索树，因此得名。]

[译注 2：原文对于 value 的使用有些混乱。为了明确表达，从此处开始，key 都译为“键”，而 value 在表达“map 中由 key 所映射到的值”时都译为“映射值”]

`Map` 的键是严格求值的，但是映射值却是非严格求值。换句话说，map 的 脊柱，或者说结构，是一直在更新的，但是 map 中映射的值还是要等到我们强迫对它们求值的时候才被计算出来。

记住这点很重要，因为对于不期望内存泄漏的程序员来说，Map 类型对映射值采用的惰性求值策略往往是内存泄漏的源头。

由于 Data.Map 模块包含几个与 Prelude 模块中冲突的名字，所以它通常用限定形式导入。本章靠前的部分中，我们再导入它时添加了一个前缀 M 。

12.9.3 类型约束

Map 类型并不对键值的类型做任何显式的约束，但是该模块中多数实用函数都要求键类型为 Ord 类型类的实体。需要强调的是，这里体现了 Haskell 中一个常见设计模式：类型约束的设置应该推迟到最终应用的地方，而不需要库作者为这种事情做额外劳动。

Map 类型和该模块中的函数都没有对映射值的类型设置约束。

12.9.4 部分应用时的尴尬

由于某些原因，Data.Map 模块中的某些函数的类型签名并不便于部分应用。函数操作的 map 总是作为最后一个参数，但是它们是第一个参数才更便于局部应用。结果造成使用部分应用 Map 函数的代码几乎总得通过适配函数 (adapter function) 来调整参数顺序。

12.9.5 map API 入门

Data.Map 模块有一个巨大的“暴露区”(surface area)：它导出了很多函数。而其中只有为数不多的几个函数算得上是该模块中最常用的核心部分。

如果需要创建一个空的 map ，可以使用 empty 函数。如果要创建包含一个键/值对的 map ，则应该使用 singleton 函数。

```
ghci> M.empty
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
fromList []
ghci> M.singleton "foo" True
fromList [("foo",True)]
```

由于 Map 的实现对我们是透明的，我们就无法对 Map 类型的值进行模式匹配。不过，该模块提供了一些查找函数可供我们使用，其中有两个函数应用特别广泛。查找函数有一个稍微复杂的类型签名，但是不要着急，这些很快在第 14 章中都会弄明白的。

```
ghci> :type M.lookup
M.lookup :: (Ord k, Monad m) => k -> M.Map k a -> m a
```

返回值中的类型参数 m 通常是 Maybe 类型。话句话说，如果 map 中包含具有给定键的映射值，lookup 函数会把映射值装入 `Just` 返回。否则返回 `Nothing` 。

```
ghci> let m = M.singleton "foo" 1 :: M.Map String Int
ghci> case M.lookup "bar" m of { Just v -> "yay"; Nothing -> "boo" }
"boo"
```

`findWithDefault` 函数额外指定一个参数值，如果 map 中不包含查找的键，则返回该指定值。

Note: 小心部分应用函数!

有一个 (!) 运算符会查找键并且返回与该键关联的原始值（即，不是返回装在 `Maybe` 或者其他什么东西里的值)。不幸的是，这并不是一个全函数：如果该键在 map 中不存在的话，它会调用 `error` 。

要在 map 中添加一个键值对，最有用的函数是 `insert` 和 `insertWith'`。insert 函数就是简单的在 map 中插入键/值对，如果该键已经存在，则覆盖其关联的任何值。

```
ghci> :type M.insert
M.insert :: (Ord k) => k -> a -> M.Map k a -> M.Map k a
ghci> M.insert "quux" 10 m
fromList [("foo",1),("quux",10)]
ghci> M.insert "foo" 9999 m
fromList [("foo",9999)]
```

`insertWith'` 函数会额外接受一个 *组合函数 (combining function)* 。如果 map 中没有指定的键，就把该键/值对原封不动插入。否则，就先对新旧两个映射值应用组合函数，把应用的结果作为新的映射值更新到 map 中。

```
ghci> :type M.insertWith'
M.insertWith' :: (Ord k) => (a -> a -> a) -> k -> a -> M.Map k a -> M.Map k a
ghci> M.insertWith' (+) "zippity" 10 m
fromList [("foo",1),("zippity",10)]
ghci> M.insertWith' (+) "foo" 9999 m
fromList [("foo",10000)]
```

函数名最后的钩号暗示我们 `insertWith'` 将对组合函数严格求值。这个设计帮你避免了内存泄漏。该函数同时存在一个惰性的变种 (即没有最后钩号的 `insertWith`)，但你大概永远用不到它。

`delete` 函数从 map 中删除指定键。如果键不存在的话，`delete` 会将 map 原封不动返回。

```
ghci> :type M.delete
M.delete :: (Ord k) => k -> M.Map k a -> M.Map k a
ghci> M.delete "foo" m
fromList []
```

最后，还有几个常用的函数用于在 maps 上进行类似集合的操作。例如，我们接下来会用到的 `union`。这个函数是“左偏”(left biased) 的：如果两个 map 包含相同的键，返回 map 中将包含左侧 map 中对应的关联值。

```
ghci> m `M.union` M.singleton "quux" 1
fromList [("foo",1),("quux",1)]
ghci> m `M.union` M.singleton "foo" 0

fromList [("foo",1)]
```

到此我们仅仅讲到了 Data.Map 中百分之十的 API。在第 13 章中我们会更加广泛深入的讲解其中的 API。我们鼓励你自行浏览模块的文档，相信你会从中获得更多启发。这个模块滴水不漏的设计一定会让你印象深刻。

12.9.6 延伸阅读

[Okasaki99] 一书将教我们如何优雅且严密地实现纯函数式数据结构，其中包括多种平衡树。该书还中还包含了作者对于纯函数式数据结构和惰性求值的宝贵思考。

我们把 Okasaki 这本书列为为函数式程序员的必读书目。如果你不方便翻阅 Okasaki 这本书，可以去看 Okasaki 的博士论文，[Okasaki96] 是该书的一个不很完整的精简版本，在网上可以免费获得。

12.10 从成堆的数字中找出答案

我们现在又有了新的问题要解决。后十二个数字还只是一堆候选数字；此外，我们需要根据这 12 个数字中的前 6 个数字的奇偶性信息来计算第一个数字。最后，我们还需要确认求出的校验码的有效性。

这看起来很有挑战！这一大堆不确定的数据；该拿它们怎么办？采用暴力搜索是一个很合理的提议。那么，如果候选数字就是上面的 **ghci** 会话中给出的那些，我们需要测试多少种组合？

```
ghci> product . map length . candidateDigits $ input
34012224
```

可见暴力搜索要检查的组合太多了。我们还是先着眼于一个知道如何解决的子问题，晚些时候在考虑剩下的的。

12.10.1 批量求解校验码

我们暂时不考虑搜索的方案，先来关注如何计算校验码。条形码的校验码可以是十个数字中的任意一个。对于一个给定的校验码，怎样反推出它是从怎样的输入序列中计算出来的呢？

```
-- file: ch12/Barcode.hs
type Map a = M.Map Digit [a]
```

在这个 map 中，键值是一个校验码，映射值是一个可以计算出这个校验码的序列。以它为基础，我们进一步定义两种 map 类型。

我们将把这两种类型的 map 统称为“答案 map”(solution map)，因为它们包含了“求解”每个校验码对应的各个数字序列。

给定一个数字，我们可以按如下方法更新一个给定的答案 map

```
-- file: ch12/Barcode.hs
updateMap :: Parity Digit       -- ^ new digit
          -> Digit              -- ^ existing key
          -> [Parity Digit]     -- ^ existing digit sequence
          -> ParityMap          -- ^ map to update
          -> ParityMap
updateMap digit key seq = insertMap key (fromParity digit) (digit:seq)

insertMap :: Digit -> Digit -> [a] -> Map a -> Map a
insertMap key digit val m = val `seq` M.insert key' val m
    where key' = (key + digit) `mod` 10
```

从 map 中取出一个既存的校验码，一个可以求出该校验码的序列，一个新的输入数字，这个函数将可以求出新的校验码的新序列更新至 map。

这部分内容可能有点不太好消化，看一个例子应该会更明白。我们假设现在要查找数字是 `4` ，它是序列 `[1, 3]` 对应的校验码，我们想要添加到 map 的数字是 `8` 。`4+8` ，模 10 得 `2` ，那么 `2` 就是要插入到 map 中的键。能计算出新校验码 `2` 的序列就是 `[8, 1, 3]` ，这个序列就是要插入的映射值。

[译注：在实际调用 `updateMap` 函数的时候，`digit` 是一个候选数字，key 是指“在插入候选数字 `new` 之前，由这个不完整的‘猜测’序列算出的临时校验码”，`seq` 就是 `key` 所对应的“不完整的‘猜测’序列 (指条形码的编码数字序列)”。`updateMap` 的实际功能就是将 `digit` 插入到列表 `seq` 的最前面，然后由插入的 seq 再求出一个校验码的临时值。并以这个临时值和插入后的序列分别为键值和映射值，插入到指定的 map 中。这之中需要注意的地方是在 `insertMap` 函数中，`key'` 表示新求出的临时校验码，这个校验码的算法与前文 checkDigit 的算法并不相同：没有对输入序列进行 `mapEveryOther (*3) (reverse ds)` 和类似 `(10 -)` 这样的计算。实际上，这两个操作只是被推迟了，并且由于校验码只有一位数，因此校验码的值与“(10 - 校验码)”的值也是一一对应的 (即“单射”)，所以 map 中保存这个没有经过 `(10 - )` 操作的键值也是没有问题的，只要在需要提取真正的校验码时用 10 减去这个键值就可以了，除了这些之外，其计算方法与 `checkDigit` 函数中的方法是等价的。]

对候选数字序列中的每一个数字，我们都会通过当前数字和之前的 map 生成一个新的答案 map。

```
-- file: ch12/Barcode.hs
useDigit :: ParityMap -> ParityMap -> Parity Digit -> ParityMap
useDigit old new digit =
    new `M.union` M.foldWithKey (updateMap digit) M.empty old
```

我们再通过一个例子演示这段代码的实际功能。这次，我们用 ghci 交互演示。

```
ghci> let single n = M.singleton n [Even n] :: ParityMap
ghci> useDigit (single 1) M.empty (Even 1)
fromList [(2,[Even 1,Even 1])]
ghci> useDigit (single 1) (single 2) (Even 2)
fromList [(2,[Even 2]),(3,[Even 2,Even 1])]
```

[译注：这个函数的参数中，old 代表上一候选数字应用此函数时产生的 map，而 old 代表“条形码中的上一个数字位置”通过不断折叠应用此函数所产生的 map，digit 表示当前考察的候选数字。这个函数的实际作用是在某个候选数字列表中遍历的过程中，当前考察的这个候选数字插入到给定 map 的每个映射值的最前方，并求得新的临时校验码，然后将这个临时校验码和插入后的序列作为键值对插入到 map 中，并与前一候选数字应用此函数的结果 map 做“并集”操作（M.union），由于候选数字序列是按照匹配程度降序排列的，因此如果当前序列中的键值与前一候选数字产生的某个键值发生冲突，那么它就会被‘‘M.Union‘‘ 的“左偏”性质覆盖掉，而保留前一候选数字所产生的新序列。]

传给 useDigits 函数的新答案 map(即参数 new 对应的 map) 最开始是空的。其值将通过在输入数字的序列上折叠 useDigits 函数来填充。

```
-- file: ch12/Barcode.hs
incorporateDigits :: ParityMap -> [Parity Digit] -> ParityMap
incorporateDigits old digits = foldl' (useDigit old) M.empty digits
```

incooperateDigit 函数可以用旧的答案 map 生成完整的新的答案 map。

```
ghci> incorporateDigits (M.singleton 0 []) [Even 1, Even 5]
fromList [(1,[Even 1]),(5,[Even 5])]
```

[译注：incorporate 函数中，参数 old 代表条码中上一个位置的数字组成的可能的数字逆序序列以及他们对应的临时校验码组成的 map，参数 digits 表示该位置上的候选数字列表。]

最终，我们必须构造完整的答案 map。我们先创建一个空的 map，然后在条形码的数字序列上依次折叠。我们为每个位置生成一个包含截止到该位置的猜测序列的 new map。这个 map 将作为下一位置上的折叠过程的 old map 出现。

```
-- file: ch12/Barcode.hs
finalDigits :: [[Parity Digit]] -> ParityMap
finalDigits = foldl' incorporateDigits (M.singleton 0 [])
            . mapEveryOther (map (fmap (*3)))
```

(回想一下，我们在“EAN-13 编码”一节中定义 checkDigit 函数的时候，要求计算校验码的之前，数字要每隔一位乘以 3 后再进行下一步处理。)

finalDigits 函数接受的列表有多少个元素呢？我们还不知道数字序列的第一个数字是什么，所以很明显第一位数字不能计入，并且在调用‘‘finalDigits‘‘ 时校验码还只是猜测值，我们也不该把它计入。所以这个输入列表应该有 11 个元素。

从 finalDigits 返回后，答案 map 必然还不完整，因为我们还没有确定首位数字是什么。

12.10.2 用首位数字补全答案 map

我们还没说过如何从左侧分组的奇偶编码类型中提取出首位数字。其实只要直接重用我们前面编写的代码就可以了。

```
-- file: ch12/Barcode.hs
firstDigit :: [Parity a] -> Digit
firstDigit = snd
           . head
           . bestScores paritySRL
           . runLengths
           . map parityBit
           . take 6
  where parityBit (Even _) = Zero
        parityBit (Odd _) = One
```

现在这个不完整的答案 map 中的每个元素都包含一个由数字和编码奇偶性信息组成的逆序的列表。接下来的任务就是通过计算每个序列的首位数字来创建一个完整的答案 map，并通过它创建最终的答案 map(即键值都是正确的校验码，映射值都是完整的 12 位正序列表的 map)。

```
-- file: ch12/Barcode.hs
addFirstDigit :: ParityMap -> DigitMap
addFirstDigit = M.foldWithKey updateFirst M.empty

updateFirst :: Digit -> [Parity Digit] -> DigitMap -> DigitMap
updateFirst key seq = insertMap key digit (digit:renormalize qes)
  where renormalize = mapEveryOther (`div` 3) . map fromParity
        digit = firstDigit qes
        qes = reverse seq
```

[译注：mapKeys 将第一个参数指定的函数逐一应用于 map 中的每个 key，并用结果替换掉原 key 值。]

如此往复，我们最终消去了 Parity 类型，并撤销了之前乘以 3 的操作。最后一步，就是完成校验码的计算。

```
-- file: ch12/Barcode.hs
buildMap :: [[Parity Digit]] -> DigitMap
buildMap = M.mapKeys (realCheckDigit)
         . addFirstDigit
         . finalDigits
               where realCheckDigit c = (10 - c) `mod` 10
```

12.10.3 找出正确的序列

我们现在有一个包含了所有可能的校验码与对应序列映射的 map。剩下的就是逐一验证我们对校验码的猜测值，检查在答案 map 中是否存在对应的键值。

```
-- file: ch12/Barcode.hs
solve :: [[Parity Digit]] -> [[Digit]]
solve [] = []
solve xs = catMaybes $ map (addCheckDigit m) checkDigits
    where checkDigits = map fromParity (last xs)
          m = buildMap (init xs)
          addCheckDigit m k = (++[k]) <$> M.lookup k m
```

[译注：catMaybes 接受一个 Maybe 类型元素组成的列表，返回一个只由 Just 构造器的参数值构成的列表(即参数列表中的 Nothing 值会被直接忽略)。

我们用从照片上取下来的那一行来试验，看看能否得到正确的结果。

```
ghci> listToMaybe . solve . candidateDigits $ input
Just [9,7,8,0,1,3,2,1,1,4,6,7,7]
```

太棒了！这正是照片中编码的数字序列。

12.11 处理行数据

我们反复强调“处理的是图像中的一行”。下面是“处理一行”的具体的做法

```
-- file: ch12/Barcode.hs
withRow :: Int -> Pixmap -> (RunLength Bit -> a) -> a
withRow n greymap f = f . runLength . elems $ posterized
    where posterized = threshold 0.4 . fmap luminance . row n $ greymap
```

withRow 函数接受图像中的一行，将该行转换为黑白图像，然后对游程编码后的行数据应用指定函数。该函数通过 row 函数来获取行数据。

```
row :: (Ix a, Ix b) => b -> Array (a,b) c -> Array a c
row j a = ixmap (l,u) project a
where project i = (i,j)
        ((l,_), (u,_)) = bounds a
```

这个函数需要稍作解释。我们知道 fmap 用来变换数组中的元素值，而此处的 ixmap 则用来变换数组中的索引值。这个强大的函数使我们可以任意地从数组取出“切片”。

ixmap 的第一个参数是新数组的边界。边界可以与原数组有不同的维。比方说，我们可以从一个二维数组中取出一个一维数组表示的行。

[译注：此处所说的“有不同的维”包括维数不同、“维的类型”不同、以及两种都有的情况。]

第二个参数是一个映射函数。其参数为新数组的索引值，返回值为原数组的索引值。映射索引的值接下来会变为新数组原索引值处的值。例如，如果我们把 2 传给映射函数，它返回 (2, 2)。这表示新数组中索引值为 2 的元素值将取自源数组中索引值为 (2, 2) 的元素。

12.12 最终装配

candidateDigits 只要不是从条形码序列的起始处调用, 就会返回一个空结果。使用下面的函数，我们可以轻松的扫描一整行，并得到匹配结果。

```
-- file: ch12/Barcode.hs
findMatch :: [(Run, Bit)] -> Maybe [[Digit]]
findMatch = listToMaybe
          . filter (not . null)
          . map (solve . candidateDigits)
          . tails
```

..FIXME: 应该是指的 candidateDigits 的惰性求值

这里，我们利用了惰性求值的优点。tails 前面的 map 函数只会在产生非空列表的时候参会真正求值。

```
-- file: ch12/Barcode.hs
findEAN13 :: Pixmap -> Maybe [Digit]
findEAN13 pixmap = withRow center pixmap (fmap head . findMatch)
  where (_, (maxX, _)) = bounds pixmap
        center = (maxX + 1) `div` 2
```

最后，我们做了一个简单的封装，用来打印从命令行传入的 netpbm 图像文件中提取的条形码。

注意到在我们本章定义的超过 30 个函数中，只有 main 是涉及 IO 的。

12.13 关于开发方式的一些意见

你可能发现了，本章中给出的许多函数都是些放在源码顶部的小函数。这并非偶然。正如我们早些时候提到过的，当我们开始本章的撰写时，我们并不知道要怎样构造这个解决方案。

我们还经常需要找到问题域来明确解决问题的大方向。为了这个目的，我们耗费了大量的时间摆弄 **ghci**，对各种函数做小测试。这需要函数定义在源文件的最上方，否则 **ghci** 就找不到它们了。

一旦我们对这些函数的功能和行为满意了，我们就开始将它们黏合在一起，继续通过 **ghci** 观察执行的结果。这就是添加类型签名的好处——如果某些函数没法组合到一起，我们可以通过类型签名尽早发现。

最后，我们有了一大堆短小的顶层函数，每个函数都有类型签名。这可能并不是最紧凑的形式；在搞定这些函数的逻辑之后，我们其实可以将其中很多函数放到 `let` 或者 `where` 块中。然而，我们发现这种更大的纵向空间，短小的函数体，以及类型签名，都使代码变得更易读了，所以我们也不再考虑拿这些函数玩儿什么“golfing”[30] 了。

使用强静态类型的语言工作不会影响增量的流式的问题解决模式。我们发现这种“先编写函数”，再“用**ghci** 测试，获取有用信息”的模式非常快速；这为我们快速编写优质代码提供了巨大帮助。

[30] 这个 golfing 的说法来源于 Perl 程序员们玩儿的一个游戏，即程序员尝试为某种目的编写最短的代码，敲键盘次数最少的获胜。

CHAPTER

THIRTEEN

第 13 章：数据结构

13.1 关联列表

我们常常会跟一些以键为索引的无序数据打交道。

举个例子，UNIX 管理猿可能需要这么一个列表，它包含系统中所有用户的 UID ，以及和这个 UID 相对应的用户名。这个列表根据 UID 而不是数据的位置来查找相应的用户名。换句话来说，UID 就是这个数据集的键。

Haskell 里有几种不同的方法来处理这种结构的数据，最常用的两个是关联列表（association list）和 `Data.Map` 模块提供的 `Map` 类型。

关联列表非常简单，易于使用。由于关联列表由 Haskell 列表构成，因此所有列表操作函数都可以用于处理关联列表。

另一方面，`Map` 类型在处理大数据集时，性能比关联列表要好。

本章将同时介绍这两种数据结构。

关联列表就是包含一个或多个 `(key, value)` 元组的列表，`key` 和 `value` 可以是任意类型。一个处理 UID 和用户名映射的关联列表的类型可能是 `[(Integer, String)]` 。

[注：关联列表的 `key` 必须是 `Eq` 类型的成员。]

关联列表的构建方式和普通列表一样。Haskell 提供了一个 `Data.List.lookup` 函数，用于在关联列表中查找数据。这个函数的类型签名为 `Eq a => a -> [(a, b)] -> Maybe b` 。它的使用方式如下：

```
Prelude> let al = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

Prelude> lookup 1 al
Just "one"

Prelude> lookup 5 al
Nothing
```

lookup 函数的定义如下：

```
-- file: ch13/lookup.hs
myLookup :: Eq a => a -> [(a, b)] -> Maybe b
myLookup _ [] = Nothing
myLookup key ((thiskey, thisval):rest) =
    if key == thiskey
       then Just thisval
       else myLookup key rest
```

lookup 在输入列表为空时返回 Nothing 。如果输入列表不为空，那么它检查当前列表元素的 key 是否就是我们要找的 key ，如果是的话就返回和这个 key 对应的 value ，否则就继续递归处理剩余的列表元素。

再来看一个稍微复杂点的例子。在 Unix/Linux 系统中，有一个 /etc/passwd 文件，这个文件保存了用户名称，UID，用户的 HOME 目录位置，以及其他一些数据。文件以行分割每个用户的资料，每个数据域用冒号隔开：

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
jgoerzen:x:1000:1000:John Goerzen,,,:/home/jgoerzen:/bin/bash
```

以下程序读入并处理 /etc/passwd 文件，它创建一个关联列表，使得我们可以根据给定 UID ，获取相应的用户名：

```
-- file: ch13/passwd-al.hs
import Data.List
import System.IO
import Control.Monad(when)
import System.Exit
import System.Environment(getArgs)

main = do
    -- Load the command-line arguments
    args <- getArgs

    -- If we don't have the right amount of args, give an error and abort
    when (length args /= 2) $ do
```

(continues on next page)

(continued from previous page)

```
        putStrLn "Syntax: passwd-al filename uid"
        exitFailure

    -- Read the file lazily
    content <- readFile (args !! 0)

    -- Compute the username in pure code
    let username = findByUID content (read (args !! 1))

    -- Display the result
    case username of
         Just x -> putStrLn x
         Nothing -> putStrLn "Could not find that UID"

-- Given the entire input and a UID, see if we can find a username.
findByUID :: String -> Integer -> Maybe String
findByUID content uid =
    let al = map parseline . lines $ content
        in lookup uid al

-- Convert a colon-separated line into fields
parseline :: String -> (Integer, String)
parseline input =
    let fields = split ':' input
        in (read (fields !! 2), fields !! 0)

-- Takes a delimiter and a list.
-- Break up the list based on the delimiter.
split :: Eq a => a -> [a] -> [[a]]

-- If the input is empty, the result is a list of empty lists.
split _ [] = [[]]
split delimiter str =
    let -- Find the part of the list before delimiter and put it in "before".
        -- The result of the list, including the leading delimiter, goes in "remainder
↪".
        (before, remainder) = span (/= delimiter) str
        in before : case remainder of
                         [] -> []
                         x -> -- If there is more data to process,
                              -- call split recursively to process it
                              split delimiter (tail x)
```

findByUID 是整个程序的核心，它逐行读入并处理输入，使用 lookup 从处理结果中查找给定 UID：

```
*Main> findByUID "root:x:0:0:root:/root:/bin/bash" 0
Just "root"
```

parseline 读入并处理一个字符串，返回一个包含 UID 和用户名的元组：

```
*Main> parseline "root:x:0:0:root:/root:/bin/bash"
(0,"root")
```

split 函数根据给定分隔符 delimiter 将一个文本行分割为列表：

```
*Main> split ':' "root:x:0:0:root:/root:/bin/bash"
["root","x","0","0","root","/root","/bin/bash"]
```

以下是在本机执行 passwd-al.hs 处理 /etc/passwd 的结果：

```
$ runghc passwd-al.hs /etc/passwd 0
root

$ runghc passwd-al.hs /etc/passwd 10086
Could not find that UID
```

13.2 Map 类型

Data.Map 模块提供的 Map 类型的行为和关联列表类似，但 Map 类型的性能更好。

Map 和其他语言提供的哈希表类似。不同的是，Map 的内部由平衡二叉树实现，在 Haskell 这种使用不可变数据的语言中，它是一个比哈希表更高效的表示。这是一个非常明显的例子，说明纯函数式语言是如何深入地影响我们编写程序的方式：对于一个给定的任务，我们总是选择合适的算法和数据结构，使得解决方案尽可能地简单和有效，但这些（纯函数式的）选择通常不同于命令式语言处理同样问题时的选择。

因为 Data.Map 模块的一些函数和 Prelude 模块的函数重名，我们通过 import qualified Data.Map as Map 的方式引入模块，并使用 Map.name 的方式引用模块中的名字。

先来看看如何用几种不同的方式构建 Map：

```
-- file: ch13/buildmap.hs
import qualified Data.Map as Map

-- Functions to generate a Map that represents an association list
-- as a map

al = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

```

(continues on next page)

(continued from previous page)

```
-- Create a map representation of 'al' by converting the association
-- list using Map.fromList
mapFromAL =
    Map.fromList al

-- Create a map representation of 'al' by doing a fold
mapFold =
    foldl (\map (k, v) -> Map.insert k v map) Map.empty al

-- Manually create a map with the elements of 'al' in it
mapManual =
    Map.insert 2 "two" .
    Map.insert 4 "four" .
    Map.insert 1 "one" .
    Map.insert 3 "three" $ Map.empty
```

`Map.insert` 函数处理数据的方式非常『Haskell 化』：它返回经过函数应用的输入数据的副本。这种处理数据的方式在操作多个 `Map` 时非常有用，它意味着你可以像前面代码中 `mapFold` 那样使用 `fold` 来构建一个 `Map`，又或者像 `mapManual` 那样，串连起多个 `Map.insert` 调用。

[译注：这里说『Haskell 化』实际上就是『函数式化』，对于函数式语言来说，最常见的函数处理方式是接受一个输入，然后返回一个输出，输出是另一个独立的值，且原输入不会被修改。]

现在，到 ghci 中验证一下是否所有定义都如我们所预期的那样工作：

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

*Main> al
Loading package array-0.4.0.0 ... linking ... done.
Loading package deepseq-1.3.0.0 ... linking ... done.
Loading package containers-0.4.2.1 ... linking ... done.
[(1,"one"),(2,"two"),(3,"three"),(4,"four")]

*Main> mapFromAL
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]

*Main> mapFold
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]

*Main> mapManual
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
```

注意，`Map` 并不保证它的输出排列和原本的输入排列一致，对比 `mapManual` 的输入和输出可以看出这一点。

Map 的操作方式和关联列表类似。Data.Map 模块提供了一组函数，用于增删 Map 元素，对 Map 进行过滤、修改和 fold ，以及在 Map 和关联列表之间进行转换。Data.Map 模块本身的文档非常优秀，因此我们在这里不会详细讲解每个函数，而是在本章的后续内容中，通过例子来介绍这些概念。

13.3 函数也是数据

Haskell 语言的威力部分在于它可以让我们方便地创建并操作函数。

以下示例展示了怎样将函数保存到记录的域中：

```
-- file: ch13/funcrecs.hs

-- Our usual CustomColor type to play with
data CustomColor =
    CustomColor {red :: Int,
                 green :: Int,
                 blue :: Int}
    deriving (Eq, Show, Read)

-- A new type that stores a name and a function.
-- The function takes an Int, applies some computation to it,
-- and returns an Int along with a CustomColor
data FuncRec =
    FuncRec {name :: String,
             colorCalc :: Int -> (CustomColor, Int)}

plus5func color x = (color, x + 5)

purple = CustomColor 255 0 255

plus5 = FuncRec {name = "plus5", colorCalc = plus5func purple}
always0 = FuncRec {name = "always0", colorCalc = \_ -> (purple, 0)}
```

注意 colorCalc 域的类型：它是一个函数，接受一个 Int 类型值作为参数，并返回一个 (CustomColor, Int) 元组。

我们创建了两个 FuncRec 记录:plus5 和 always0 ,这两个记录的 colorCalc 域都总是返回紫色(purple)。FuncRec 自身并没有域去保存所使用的颜色，颜色的值被保存在函数当中——我们称这种用法为闭包。

以下是示例代码：

```
*Main> :l funcrecs.hs
[1 of 1] Compiling Main             ( funcrecs.hs, interpreted )
Ok, modules loaded: Main.
```

(continues on next page)

(continued from previous page)

```
*Main> :t plus5
plus5 :: FuncRec

*Main> name plus5
"plus5"

*Main> :t colorCalc plus5
colorCalc plus5 :: Int -> (CustomColor, Int)

*Main> (colorCalc plus5) 7
(CustomColor {red = 255, green = 0, blue = 255},12)

*Main> :t colorCalc always0
colorCalc always0 :: Int -> (CustomColor, Int)

*Main> (colorCalc always0) 7
(CustomColor {red = 255, green = 0, blue = 255},0)
```

上面的程序工作得很好，但我们还想做一些更有趣的事，比如说，在多个域中使用同一段数据。可以使用一个类型构造函数来做到这一点：

```
-- file: ch13/funcrecs2.hs
data FuncRec =
    FuncRec {name :: String,
             calc :: Int -> Int,
             namedCalc :: Int -> (String, Int)}

mkFuncRec :: String -> (Int -> Int) -> FuncRec
mkFuncRec name calcfunc =
    FuncRec {name = name,
             calc = calcfunc,
             namedCalc = \x -> (name, calcfunc x)}

plus5 = mkFuncRec "plus5" (+ 5)
always0 = mkFuncRec "always0" (\_ -> 0)
```

mkFuncRecs 函数接受一个字符串和一个函数作为参数，返回一个新的 FuncRec 记录。以下是对 mkFuncRecs 函数的测试：

```
*Main> :l funcrecs2.hs
[1 of 1] Compiling Main             ( funcrecs2.hs, interpreted )
Ok, modules loaded: Main.
```

(continues on next page)

(continued from previous page)

```
*Main> :t plus5
plus5 :: FuncRec

*Main> name plus5
"plus5"

*Main> (calc plus5) 5
10

*Main> (namedCalc plus5) 5
("plus5",10)

*Main> let plus5a = plus5 {name = "PLUS5A"}

*Main> name plus5a
"PLUS5A"

*Main> (namedCalc plus5a) 5
("plus5",10)
```

注意 plus5a 的创建过程：我们改变了 plus5 的 name 域，但没有修改它的 namedCalc 域。这就是为什么调用 name 会返回新名字，而 namedCalc 依然返回原本使用 mkFuncRecs 创建时设置的名字——除非我们显式地修改域，否则它们不会被改变。

13.4 扩展示例：/etc/password

以下是一个扩展示例，它展示了几种不同的数据结构的用法，根据 /etc/passwd 文件的格式，程序处理并保存它的实体（entry）：

```
-- file: ch13/passwdmap.hs
import Data.List
import qualified Data.Map as Map
import System.IO
import Text.Printf(printf)
import System.Environment(getArgs)
import System.Exit
import Control.Monad(when)

{- | The primary piece of data this program will store.
   It represents the fields in a POSIX /etc/passwd file -}
data PasswdEntry = PasswdEntry {
```

(continues on next page)

(continued from previous page)

```
    userName :: String,
    password :: String,
    uid :: Integer,
    gid :: Integer,
    gecos :: String,
    homeDir :: String,
    shell :: String}
    deriving (Eq, Ord)

{- | Define how we get data to a 'PasswdEntry'. -}
instance Show PasswdEntry where
    show pe = printf "%s:%s:%d:%d:%s:%s:%s"
                (userName pe) (password pe) (uid pe) (gid pe)
                (gecos pe) (homeDir pe) (shell pe)

{- | Converting data back out of a 'PasswdEntry'. -}
instance Read PasswdEntry where
    readsPrec _ value =
        case split ':' value of
             [f1, f2, f3, f4, f5, f6, f7] ->
                 -- Generate a 'PasswdEntry' the shorthand way:
                 -- using the positional fields.  We use 'read' to convert
                 -- the numeric fields to Integers.
                 [(PasswdEntry f1 f2 (read f3) (read f4) f5 f6 f7, [])]
             x -> error $ "Invalid number of fields in input: " ++ show x
        where
        {- | Takes a delimiter and a list.  Break up the list based on the
        -  delimiter. -}
        split :: Eq a => a -> [a] -> [[a]]

        -- If the input is empty, the result is a list of empty lists.
        split _ [] = [[]]
        split delim str =
            let -- Find the part of the list before delim and put it in
                -- "before".  The rest of the list, including the leading
                -- delim, goes in "remainder".
                (before, remainder) = span (/= delim) str
                in
                before : case remainder of
                             [] -> []
                             x -> -- If there is more data to process,
                                  -- call split recursively to process it
                                  split delim (tail x)
```

(continues on next page)

(continued from previous page)

```
-- Convenience aliases; we'll have two maps: one from UID to entries
-- and the other from username to entries
type UIDMap = Map.Map Integer PasswdEntry
type UserMap = Map.Map String PasswdEntry

{- | Converts input data to maps.  Returns UID and User maps. -}
inputToMaps :: String -> (UIDMap, UserMap)
inputToMaps inp =
    (uidmap, usermap)
    where
    -- fromList converts a [(key, value)] list into a Map
    uidmap = Map.fromList . map (\pe -> (uid pe, pe)) $ entries
    usermap = Map.fromList .
              map (\pe -> (userName pe, pe)) $ entries
    -- Convert the input String to [PasswdEntry]
    entries = map read (lines inp)

main = do
    -- Load the command-line arguments
    args <- getArgs

    -- If we don't have the right number of args,
    -- give an error and abort

    when (length args /= 1) $ do
        putStrLn "Syntax: passwdmap filename"
        exitFailure

    -- Read the file lazily
    content <- readFile (head args)
    let maps = inputToMaps content
    mainMenu maps

mainMenu maps@(uidmap, usermap) = do
    putStr optionText
    hFlush stdout
    sel <- getLine
    -- See what they want to do.  For every option except 4,
    -- return them to the main menu afterwards by calling
    -- mainMenu recursively
    case sel of
         "1" -> lookupUserName >> mainMenu maps
```

(continues on next page)

(continued from previous page)

```
        "2" -> lookupUID >> mainMenu maps
        "3" -> displayFile >> mainMenu maps
        "4" -> return ()
        _ -> putStrLn "Invalid selection" >> mainMenu maps

  where
  lookupUserName = do
      putStrLn "Username: "
      username <- getLine
      case Map.lookup username usermap of
           Nothing -> putStrLn "Not found."
           Just x -> print x
  lookupUID = do
      putStrLn "UID: "
      uidstring <- getLine
      case Map.lookup (read uidstring) uidmap of
           Nothing -> putStrLn "Not found."
           Just x -> print x
  displayFile =
      putStr . unlines . map (show . snd) . Map.toList $ uidmap
  optionText =
        "\npasswdmap options:\n\
         \\n\
         \1   Look up a user name\n\
         \2   Look up a UID\n\
         \3   Display entire file\n\
         \4   Quit\n\n\
         \Your selection: "
```

示例程序维持两个 `Map`： 一个从用户名映射到 `PasswdEntry`，另一个从 UID 映射到 `PasswdEntry`。有数据库使用经验的人可以将它们看作是两个不同数据域的索引。

根据 `/etc/passwd` 文件的格式，`PasswdEntry` 的 `Show` 和 `Read` 实例分别用于显示（display）和处理（parse）工作。

13.5 扩展示例：数字类型（Numeric Types）

我们已经讲过 Haskell 的类型系统有多强大，表达能力有多强。我们已经讲过很多利用这种能力的方法。现在我们来举一个实际的例子看看。

在数值类型 一节中，我们展示了 Haskell 的数字类型类。现在，我们来定义一些类，然后用数字类型类把它们和 Haskell 的基本数学结合起来，看看能得到什么。

我们先来想想我们想用这些新类型在 **ghci** 里干什么。首先，一个不错的选择是把数学表达式转成字符串，并确保它显示了正确的优先级。我们可以写一个 `prettyShow` 函数来实现。稍后我们就告诉你怎么写，先来看看怎么用它。

```
ghci> :l num.hs
[1 of 1] Compiling Main             ( num.hs, interpreted )
Ok, modules loaded: Main.
ghci> 5 + 1 * 3
8
ghci> prettyShow $ 5 + 1 * 3
"5+(1*3)"
ghci> prettyShow $ 5 * 1 + 3
"(5*1)+3"
```

看起来不错，但还不够聪明。我们可以很容易地把 `1 *` 从表达式里拿掉。写个函数来简化怎么样?

```
ghci> prettyShow $ simplify $ 5 + 1 * 3
"5+3"
```

把数学表达式转成逆波兰表达式（RPN）怎么样？RPN 是一种后缀表示法，它不要求括号，常见于 HP 计算器。RPN 是一种基于栈的表达式。我们把数字放进栈里，当碰到操作符时，栈顶的数字出栈，结果再被放回栈里。

```
ghci> rpnShow $ 5 + 1 * 3
"5 1 3 * +"
ghci> rpnShow $ simplify $ 5 + 1 * 3
"5 3 +"
```

能表示含有未知符号的简单表达式也很不错。

```
ghci> prettyShow $ 5 + (Symbol "x") * 3
"5+(x*3)"
```

跟数字打交道时，单位常常很重要。例如，当你看见数字 5 时，它是 5 米，5 英尺，还是 5 字节？当然，当你用 5 米除以 2 秒时，系统应该推出来正确的单位。而且，它应该阻止你用 2 秒加上 5 米。

```
ghci> 5 / 2
2.5
ghci> (units 5 "m") / (units 2 "s")
2.5_m/s
ghci> (units 5 "m") + (units 2 "s")
*** Exception: Mis-matched units in add
ghci> (units 5 "m") + (units 2 "m")
7_m
ghci> (units 5 "m") / 2
```

(continues on next page)

(continued from previous page)

```
2.5_m
ghci> 10 * (units 5 "m") / (units 2 "s")
25.0_m/s
```

如果我们定义的表达式或函数对所有数字都合法，那我们就应该能计算出结果，或者把表达式转成字符串。例如，如果我们定义 `test` 的类型为 `Num a => a`，并令 `test = 2 * 5 + 3`，那我们应该可以：

```
ghci> test
13
ghci> rpnShow test
"2 5 * 3 +"
ghci> prettyShow test
"(2*5)+3"
ghci> test + 5
18
ghci> prettyShow (test + 5)
"((2*5)+3)+5"
ghci> rpnShow (test + 5)
"2 5 * 3 + 5 +"
```

既然我们能处理单位，那我们也应该能处理一些基本的三角函数，其中很多操作都是关于角的。让我们确保角度和弧度都能被处理。

```
ghci> sin (pi / 2)
1.0
ghci> sin (units (pi / 2) "rad")
1.0_1.0
ghci> sin (units 90 "deg")
1.0_1.0
ghci> (units 50 "m") * sin (units 90 "deg")
50.0_m
```

最后，我们应该能把这些都放在一起，把不同类型的表达式混合使用。

```
ghci> ((units 50 "m") * sin (units 90 "deg")) :: Units (SymbolicManip Double)
50.0*sin(((2.0*pi)*90.0)/360.0)_m
ghci> prettyShow $ dropUnits $ (units 50 "m") * sin (units 90 "deg")
"50.0*sin(((2.0*pi)*90.0)/360.0)"
ghci> rpnShow $ dropUnits $ (units 50 "m") * sin (units 90 "deg")
"50.0 2.0 pi * 90.0 * 360.0 / sin *"
ghci> (units (Symbol "x") "m") * sin (units 90 "deg")
x*sin(((2.0*pi)*90.0)/360.0)_m
```

你刚才看到的一切都可以用 Haskell 的类型和类型类实现。实际上，你看到的正是我们马上要实现的 `num.hs`。

13.5.1 第一步

我们想想怎么实现上面提到的功能。首先，用 **ghci** 查看一下可知，(+) 的类型是 `Num a => a -> a -> a`。如果我们想给加号实现一些自定义行为，我们就必须定义一个新类型并声明它为 Num 的实例。这个类型得用符号的形式来存储表达式。我们可以从加法操作开始。我们需要存储操作符本身、左侧以及右侧内容。左侧和右侧内容本身又可以是表达式。

我们可以把表达式想象成一棵树。让我们从一些简单类型开始。

```
-- file: ch13/numsimple.hs
-- 我们支持的操作符
data Op = Plus | Minus | Mul | Div | Pow
        deriving (Eq, Show)

{- 核心符号操作类型 (core symbolic manipulation type) -}
data SymbolicManip a =
          Number a            -- Simple number, such as 5
        | Arith Op (SymbolicManip a) (SymbolicManip a)
          deriving (Eq, Show)

{- SymbolicManip 是 Num 的实例。定义 SymbolicManip 实现 Num 的函数。如 (+) 等。 -}
instance Num a => Num (SymbolicManip a) where
    a + b = Arith Plus a b
    a - b = Arith Minus a b
    a * b = Arith Mul a b
    negate a = Arith Mul (Number (-1)) a
    abs a = error "abs is unimplemented"
    signum _ = error "signum is unimplemented"
    fromInteger i = Number (fromInteger i)
```

首先我们定义了 Op 类型。这个类型表示我们要支持的操作。接着，我们定义了 `SymbolicManip a`，由于 `Num a` 约束的存在，a 可替换为任何 Num 实例。我们可以有 `SymbolicManip Int` 这样的具体类型。

SymbolicManip 类型可以是数字，也可以是数学运算。Arith 构造器是递归的，这在 Haskell 里完全合法。Arith 用一个 Op 和两个 SymbolicManip 创建了一个 SymbolicManip。我们来看一个例子：

```
Prelude> :l numsimple.hs
[1 of 1] Compiling Main             ( numsimple.hs, interpreted )
Ok, modules loaded: Main.
*Main> Number 5
Number 5
*Main> :t Number 5
Number 5 :: Num a => SymbolicManip a
*Main> :t Number (5::Int)
Number (5::Int) :: SymbolicManip Int
```

(continues on next page)

(continued from previous page)

```
*Main> Number 5 * Number 10
Arith Mul (Number 5) (Number 10)
*Main> (5 * 10)::SymbolicManip Int
Arith Mul (Number 5) (Number 10)
*Main> (5 * 10 + 2)::SymbolicManip Int
Arith Plus (Arith Mul (Number 5) (Number 10)) (Number 2)
```

可以看到，我们已经可以表示一些简单的表达式了。注意观察 Haskell 是如何把 `5 * 10 + 2` “转换”成 `SymbolicManip` 值的，它甚至还正确处理了求值顺序。事实上，这并不是真正意义上的转换，因为 `SymbolicManip` 已经是一等数字（first-class number）了。就算 `Integer` 类型的数字字面量（numeric literals）在内部也是被包装在 `fromInteger` 里的，所以 `5` 作为一个 `SymbolicManip Int` 和作为一个 `Int` 同样有效。

从这儿开始，我们的任务就简单了：扩展 `SymbolicManip`，使它能表示所有我们想要的操作；把它声明为其它数字类型类的实例；为 `SymbolicManip` 实现我们自己的 `Show` 实例，使这棵树在显示时更友好。

13.5.2 完整代码

这里是完整的 `num.hs`，我们在本节开始的 **ghci** 例子中用到了它。我们来一点一点分析这段代码。

```
-- file: ch13/num.hs
import Data.List

----------------------------------------------------
-- Symbolic/units manipulation
----------------------------------------------------

-- The "operators" that we're going to support
data Op = Plus | Minus | Mul | Div | Pow
        deriving (Eq, Show)

{- The core symbolic manipulation type.  It can be a simple number,
a symbol, a binary arithmetic operation (such as +), or a unary
arithmetic operation (such as cos)

Notice the types of BinaryArith and UnaryArith: it's a recursive
type.  So, we could represent a (+) over two SymbolicManips. -}
data SymbolicManip a =
        Number a            -- Simple number, such as 5
      | Symbol String       -- A symbol, such as x
      | BinaryArith Op (SymbolicManip a) (SymbolicManip a)
      | UnaryArith String (SymbolicManip a)
        deriving (Eq)
```

我们在这段代码中定义了 Op，和之前我们用到的一样。我们也定义了 SymbolicManip，它和我们之前用到的类似。在这个版本中，我们开始支持一元数学操作（unary arithmetic operations）（也就是接受一个参数的操作），例如 abs 和 cos。接下来我们来定义自己的 Num 实例。

```
-- file: ch13/num.hs
{- SymbolicManip will be an instance of Num.  Define how the Num
operations are handled over a SymbolicManip.  This will implement things
like (+) for SymbolicManip. -}
instance Num a => Num (SymbolicManip a) where
    a + b = BinaryArith Plus a b
    a - b = BinaryArith Minus a b
    a * b = BinaryArith Mul a b
    negate a = BinaryArith Mul (Number (-1)) a
    abs a = UnaryArith "abs" a
    signum _ = error "signum is unimplemented"
    fromInteger i = Number (fromInteger i)
```

非常直观，和之前的代码很像。注意之前我们不支持 abs，但现在可以了，因为有了 UnaryArith。接下来，我们再定义几个实例。

```
-- file: ch13/num.hs
{- 定义 SymbolicManip 为 Fractional 实例 -}
instance (Fractional a) => Fractional (SymbolicManip a) where
    a / b = BinaryArith Div a b
    recip a = BinaryArith Div (Number 1) a
    fromRational r = Number (fromRational r)

{- 定义 SymbolicManip 为 Floating 实例 -}
instance (Floating a) => Floating (SymbolicManip a) where
    pi = Symbol "pi"
    exp a = UnaryArith "exp" a
    log a = UnaryArith "log" a
    sqrt a = UnaryArith "sqrt" a
    a ** b = BinaryArith Pow a b
    sin a = UnaryArith "sin" a
    cos a = UnaryArith "cos" a
    tan a = UnaryArith "tan" a
    asin a = UnaryArith "asin" a
    acos a = UnaryArith "acos" a
    atan a = UnaryArith "atan" a
    sinh a = UnaryArith "sinh" a
    cosh a = UnaryArith "cosh" a
    tanh a = UnaryArith "tanh" a
    asinh a = UnaryArith "asinh" a
    acosh a = UnaryArith "acosh" a
```

(continues on next page)

(continued from previous page)

```
    atanh a = UnaryArith "atanh" a
```

这段代码直观地定义了 Fractional 和 Floating 实例。接下来，我们把表达式转换字符串。

```
-- file: ch13/num.hs
{- 使用常规代数表示法，把 SymbolicManip 转换为字符串 -}
prettyShow :: (Show a, Num a) => SymbolicManip a -> String

-- 显示字符或符号
prettyShow (Number x) = show x
prettyShow (Symbol x) = x

prettyShow (BinaryArith op a b) =
    let pa = simpleParen a
        pb = simpleParen b
        pop = op2str op
        in pa ++ pop ++ pb
prettyShow (UnaryArith opstr a) =
    opstr ++ "(" ++ show a ++ ")"

op2str :: Op -> String
op2str Plus = "+"
op2str Minus = "-"
op2str Mul = "*"
op2str Div = "/"
op2str Pow = "**"

{- 在需要的地方添加括号。这个函数比较保守，有时候不需要也会加。
Haskell 在构建 SymbolicManip 的时候已经处理好优先级了。-}
simpleParen :: (Show a, Num a) => SymbolicManip a -> String
simpleParen (Number x) = prettyShow (Number x)
simpleParen (Symbol x) = prettyShow (Symbol x)
simpleParen x@(BinaryArith _ _ _) = "(" ++ prettyShow x ++ ")"
simpleParen x@(UnaryArith _ _) = prettyShow x

{- 调用 prettyShow 函数显示 SymbolicManip 值 -}
instance (Show a, Num a) => Show (SymbolicManip a) where
    show a = prettyShow a
```

首先我们定义了 prettyShow 函数。它把一个表达式转换成常规表达形式。算法相当简单：数字和符号不做处理；二元操作是转换后两侧的内容加上中间的操作符；当然我们也处理了一元操作。op2str 把 Op 转为 String。在 simpleParen 里，我们加括号的算法非常保守，以确保优先级在结果里清楚显示。最后，我们声明 SymbolicManip 为 Show 的实例然后用 prettyShow 来实现。现在，我们来设计一个算法把表达式转为 RPN 形式的字符串。

```haskell
-- file: ch13/num.hs
{- Show a SymbolicManip using RPN.  HP calculator users may
find this familiar. -}
rpnShow :: (Show a, Num a) => SymbolicManip a -> String
rpnShow i =
    let toList (Number x) = [show x]
        toList (Symbol x) = [x]
        toList (BinaryArith op a b) = toList a ++ toList b ++
            [op2str op]
        toList (UnaryArith op a) = toList a ++ [op]
        join :: [a] -> [[a]] -> [a]
        join delim l = concat (intersperse delim l)
    in join " " (toList i)
```

RPN 爱好者会发现，跟上面的算法相比，这个算法是多么简洁。尤其是，我们根本不用关心要从哪里加括号，因为 RPN 天生只能沿着一个方向求值。接下来，我们写个函数来实现一些基本的表达式化简。

```haskell
-- file: ch13/num.hs
{- Perform some basic algebraic simplifications on a SymbolicManip. -}
simplify :: (Eq a, Num a) => SymbolicManip a -> SymbolicManip a
simplify (BinaryArith op ia ib) =
    let sa = simplify ia
        sb = simplify ib
        in
        case (op, sa, sb) of
                (Mul, Number 1, b) -> b
                (Mul, a, Number 1) -> a
                (Mul, Number 0, b) -> Number 0
                (Mul, a, Number 0) -> Number 0
                (Div, a, Number 1) -> a
                (Plus, a, Number 0) -> a
                (Plus, Number 0, b) -> b
                (Minus, a, Number 0) -> a
                _ -> BinaryArith op sa sb
simplify (UnaryArith op a) = UnaryArith op (simplify a)
simplify x = x
```

这个函数相当简单。我们很轻易地就能化简某些二元数学运算——例如，用 1 乘以任何值。我们首先得到操作符两侧操作数被化简之后的版本（在这儿用到了递归）然后再化简结果。对于一元操作符我们能做的不多，所以我们仅仅简化它们作用于的表达式。

从现在开始，我们会增加对计量单位的支持。增加之后我们就能表示“5 米”这种数量了。跟之前一样，我们先来定义一个类型：

```haskell
-- file: ch13/num.hs
{- 新数据类型：Units。Units 类型包含一个数字和一个 SymbolicManip，也就是计量单位。
计量单位符号可以是 (Symbol "m") 这个样子。 -}
data Units a = Units a (SymbolicManip a)
             deriving (Eq)
```

一个 Units 值包含一个数字和一个符号。符号本身是 SymbolicManip 类型。接下来，将 Units 声明为 Num 实例。

```haskell
-- file: ch13/num.hs
{- 为 Units 实现 Num 实例。我们不知道如何转换任意单位，因此当不同单位的数字相加时，我们报告错误。
对于乘法，我们生成对应的新单位。 -}
instance (Eq a, Num a) => Num (Units a) where
    (Units xa ua) + (Units xb ub)
        | ua == ub = Units (xa + xb) ua
        | otherwise = error "Mis-matched units in add or subtract"
    (Units xa ua) - (Units xb ub) = (Units xa ua) + (Units (xb * (-1)) ub)
    (Units xa ua) * (Units xb ub) = Units (xa * xb) (ua * ub)
    negate (Units xa ua) = Units (negate xa) ua
    abs (Units xa ua) = Units (abs xa) ua
    signum (Units xa _) = Units (signum xa) (Number 1)
    fromInteger i = Units (fromInteger i) (Number 1)
```

现在，我们应该清楚为什么要用 SymbolicManip 而不是 String 来存储计量单位了。做乘法时，计量单位也会发生改变。例如，5 米乘以 2 米会得到 10 平方米。我们要求加法运算的单位必须匹配，并用加法实现了减法。我们再来看几个 Units 的类型类实例。

```haskell
-- file: ch13/num.hs
{- Make Units an instance of Fractional -}
instance (Eq a, Fractional a) => Fractional (Units a) where
    (Units xa ua) / (Units xb ub) = Units (xa / xb) (ua / ub)
    recip a = 1 / a
    fromRational r = Units (fromRational r) (Number 1)

{- Floating implementation for Units.

Use some intelligence for angle calculations: support deg and rad
-}
instance (Eq a, Floating a) => Floating (Units a) where
    pi = (Units pi (Number 1))
    exp _ = error "exp not yet implemented in Units"
    log _ = error "log not yet implemented in Units"
    (Units xa ua) ** (Units xb ub)
        | ub == Number 1 = Units (xa ** xb) (ua ** Number xb)
```

(continues on next page)

(continued from previous page)

```
        | otherwise = error "units for RHS of ** not supported"
    sqrt (Units xa ua) = Units (sqrt xa) (sqrt ua)
    sin (Units xa ua)
        | ua == Symbol "rad" = Units (sin xa) (Number 1)
        | ua == Symbol "deg" = Units (sin (deg2rad xa)) (Number 1)
        | otherwise = error "Units for sin must be deg or rad"
    cos (Units xa ua)
        | ua == Symbol "rad" = Units (cos xa) (Number 1)
        | ua == Symbol "deg" = Units (cos (deg2rad xa)) (Number 1)
        | otherwise = error "Units for cos must be deg or rad"
    tan (Units xa ua)
        | ua == Symbol "rad" = Units (tan xa) (Number 1)
        | ua == Symbol "deg" = Units (tan (deg2rad xa)) (Number 1)
        | otherwise = error "Units for tan must be deg or rad"
    asin (Units xa ua)
        | ua == Number 1 = Units (rad2deg $ asin xa) (Symbol "deg")
        | otherwise = error "Units for asin must be empty"
    acos (Units xa ua)
        | ua == Number 1 = Units (rad2deg $ acos xa) (Symbol "deg")
        | otherwise = error "Units for acos must be empty"
    atan (Units xa ua)
        | ua == Number 1 = Units (rad2deg $ atan xa) (Symbol "deg")
        | otherwise = error "Units for atan must be empty"
    sinh = error "sinh not yet implemented in Units"
    cosh = error "cosh not yet implemented in Units"
    tanh = error "tanh not yet implemented in Units"
    asinh = error "asinh not yet implemented in Units"
    acosh = error "acosh not yet implemented in Units"
    atanh = error "atanh not yet implemented in Units"
```

虽然没有实现所有函数，但大部分都定义了。现在我们来定义几个跟单位打交道的工具函数。

```
-- file: ch13/num.hs
{- A simple function that takes a number and a String and returns an
appropriate Units type to represent the number and its unit of measure -}
units :: (Num z) => z -> String -> Units z
units a b = Units a (Symbol b)

{- Extract the number only out of a Units type -}
dropUnits :: (Num z) => Units z -> z
dropUnits (Units x _) = x

{- Utilities for the Unit implementation -}
deg2rad x = 2 * pi * x / 360
```

(continues on next page)

(continued from previous page)

```
rad2deg x = 360 * x / (2 * pi)
```

首先我们定义了 units，使表达式更简洁。units 5 "m" 肯定要比 Units 5 (Symbol "m") 省事。我们还定义了 dropUnits，它把单位去掉只返回 Num。最后，我们定义了两个函数，用来在角度和弧度之间转换。接下来，我们给 Units 定义 Show 实例。

```
-- file: ch13/num.hs
{- Showing units: we show the numeric component, an underscore,
then the prettyShow version of the simplified units -}
instance (Eq a, Show a, Num a) => Show (Units a) where
    show (Units xa ua) = show xa ++ "_" ++ prettyShow (simplify ua)
```

很简单。最后我们定义 test 变量用来测试。

```
-- file: ch13/num.hs
test :: (Num a) => a
test = 2 * 5 + 3
```

回头看看这些代码，我们已经完成了既定目标：给 SymbolicManip 实现更多实例；我们引入了新类型 Units,它包含一个数字和一个单位;我们实现了几个 show 函数,以便用不同的方式来转换 SymbolicManip 和 Units。

这个例子还给了我们另外一点启发。所有语言——即使那些包含对象和重载的——都有从某种角度看很独特的地方。在 Haskell 里，这个“特殊”的部分很小。我们刚刚开发了一种新的表示法用来表示像数字一样基本的东西，而且很容易就实现了。我们的新类型是一等类型，编译器在编译时就知道使用它哪个函数。Haskell 把代码复用和互换（interchangeability）发挥到了极致。写通用代码很容易，而且很方便就能把它们用于多种不同类型的东西上。同样容易的是创建新类型并使它们自动成为系统的一等功能（first-class features）。

还记得本节开头的 **ghci** 例子吗？我们已经实现了它的全部功能。你可以自己试试，看看它们是怎么工作的。

13.5.3 练习

1. 扩展 prettyShow 函数，去掉不必要的括号。

13.6 把函数当成数据来用

在命令式语言当中，拼接两个列表很容易。下面的 C 语言结构维护了指向列表头尾的指针：

```
struct list {
    struct node *head, *tail;
};
```

当我们想把列表 B 拼接到列表 A 的尾部时，我们将 A 的最后一个节点指向 B 的 `head` 节点，再把 A 的 `tail` 指针指向 B 的 `tail` 节点。

很显然，在 Haskell 里，如果我们想保持“纯”的话，这种方法是有局限性的。由于纯数据是不可变的，我们无法原地修改列表。Haskell 的 `(++)` 操作符通过生成一个新列表来拼接列表。

```
-- file: ch13/Append.hs
(++) :: [a] -> [a] -> [a]
(x:xs) ++ ys = x : xs ++ ys
_      ++ ys = ys
```

从代码里可以看出，创建新列表的开销取决于第一个列表的长度。

我们经常需要通过重复拼接列表来创建一个大列表。例如，在生成网页内容时我们可能想生成一个 `String`。每当有新内容添加到网页中时，我们会很自然地想到把它拼接到已有 `String` 的末尾。

如果每一次拼接的开销都正比与初始列表的长度，每一次拼接都把初始列表加的更长，那么我们将会陷入一个很糟糕的情况：所有拼接的总开销将会正比于最终列表长度的平方。

为了更好地理解，我们来研究一下。`(++)` 操作符是右结合的。

```
ghci> :info (++)
(++) :: [a] -> [a] -> [a]    -- Defined in GHC.Base
infixr 5 ++
```

这意味着 Haskell 在求值表达式 `"a" ++ "b" ++ "c"` 时会从右向左进行，就像加了括号一样：`"a" ++ ("b" ++ "c")`。这对于提高性能非常有好处，因为它会让左侧操作数始终保持最短。

当我们重复向列表末尾拼接时，我们破坏了这种结合性。假设我们有一个列表 `"a"` 然后想把 `"b"` 拼接上去，我们把结果存储在一个新列表里。稍后如果我们想把 `"c"` 拼接上去时，这时的左操作数已经变成了 `"ab"`。在这种情况下，每次拼接都让左操作数变得更长。

与此同时，命令式语言的程序员却在偷笑，因为他们重复拼接的开销只取决于操作的次数。他们的性能是线性的，我们的是平方的。

当像重复拼接列表这种常见任务都暴露出如此严重的性能问题时，我们有必要从另一个角度来看看问题了。

表达式 `("a"++)` 是一个节 (section)，一个部分应用的函数。它的类型是什么呢?

```
ghci> :type ("a" ++)
("a" ++) :: [Char] -> [Char]
```

由于这是一个函数，我们可以用 `(.)` 操作符把它和另一个节组合起来，例如 `("b"++)`。

```
ghci> :type ("a" ++) . ("b" ++)
("a" ++) . ("b" ++) :: [Char] -> [Char]
```

新函数的类型和之前相同。当我们停止组合函数，并向我们创造的函数提供一个 `String` 会发生什么呢?

```
ghci> let f = ("a" ++) . ("b" ++)
ghci> f []
"ab"
```

我们实现了字符串拼接！我们利用这些部分应用的函数来存储数据，并且只要提供一个空列表就可以把数据提取出来。每一个 (++) 和 (.) 部分应用都代表了一次拼接，但它们并没有真正完成拼接。

这个方法有两点非常有趣。第一点是部分应用的开销是固定的，这样多次部分应用的开销就是线性的。第二点是当我们提供一个 [] 值来从部分应用链中提取最终列表时，求值会从右至左进行。这使得 (++) 的左操作数尽可能小，使得所有拼接的总开销是线性而不是平方。

通过使用这种并不太熟悉的数据表示方式，我们避免了一个性能泥潭，并且对“把函数当成数据来用”有了新的认识。顺便说一下，这个技巧并不新鲜，它通常被称为*差异列表*（difference list）。

还有一点没讲。尽管从理论上看差异列表非常吸引人，但如果在实际中把 (++)、(.) 和部分应用都暴露在外的话，它并不会非常好用。我们需要把它转成一种更好用的形式。

13.6.1 把差异列表转成库

第一步是用 newtype 声明把底层的类型隐藏起来。我们会创建一个 DList 类型。类似于普通列表，它是一个参数化类型。

```
-- file: ch13/DList.hs
newtype DList a = DL {
    unDL :: [a] -> [a]
    }
```

unDL 是我们的析构函数，它把 DL 构造器删除掉。我们最后导出模块函数时会忽略掉构造函数和析构函数，这样我们的用户就没必要知道 DList 类型的实现细节。他们只用我们导出的函数即可。

```
-- file: ch13/DList.hs
append :: DList a -> DList a -> DList a
append xs ys = DL (unDL xs . unDL ys)
```

我们的 append 函数看起来可能有点复杂，但其实它仅仅是围绕着 (.) 操作符做了一些簿记工作，(.) 的用法和我们之前展示的完全一样。生成函数的时候，我们必须首先用 unDL 函数把它们从 DL 构造器中取出来。然后我们在把得到的结果重新用 DL 包装起来，确保它的类型正确。

下面是相同函数的另一种写法，这种方法通过模式识别取出 xs 和 ys。

```
-- file: ch13/DList.hs
append' :: DList a -> DList a -> DList a
append' (DL xs) (DL ys) = DL (xs . ys)
```

我们需要在 DList 类型和普通列表之间来回转换。

```
-- file: ch13/DList.hs
fromList :: [a] -> DList a
fromList xs = DL (xs ++)

toList :: DList a -> [a]
toList (DL xs) = xs []
```

再次声明，跟这些函数最原始的版本相比，我们在这里做的只是一些簿记工作。

如果我们想把 DList 作为普通列表的替代品，我们还需要提供一些常用的列表操作。

```
-- file: ch13/DList.hs
empty :: DList a
empty = DL id

-- equivalent of the list type's (:) operator
cons :: a -> DList a -> DList a
cons x (DL xs) = DL ((x:) . xs)
infixr `cons`

dfoldr :: (a -> b -> b) -> b -> DList a -> b
dfoldr f z xs = foldr f z (toList xs)
```

尽管 DList 使得拼接很廉价，但并不是所有的列表操作都容易实现。列表的 head 函数具有常数开销，而对应的 DList 实现却需要将整个 DList 转为普通列表，因此它比普通列表的实现昂贵得多：它的开销正比于构造 DList 所需的拼接次数。

```
-- file: ch13/DList.hs
safeHead :: DList a -> Maybe a
safeHead xs = case toList xs of
                (y:_) -> Just y
                _ -> Nothing
```

为了实现对应的 map 函数，我们可以把 DList 类型声明为一个 Functor（函子）。

```
-- file: ch13/DList.hs
dmap :: (a -> b) -> DList a -> DList b
dmap f = dfoldr go empty
    where go x xs = cons (f x) xs

instance Functor DList where
    fmap = dmap
```

当我们实现了足够多的列表操作时，我们回到源文件顶部增加一个模块头。

```
-- file: ch13/DList.hs
module DList
    (
    DList
    , fromList
    , toList
    , empty
    , append
    , cons
    , dfoldr
    ) where
```

13.6.2 列表、差异列表和幺半群（monoids）

在抽象代数中，有一类简单的抽象结构被称为*幺半群*。许多数学结构都是幺半群，因为成为幺半群的要求非常低。一个结构只要满足两个性质便可称为幺半群：

- 一个满足结合律的二元操作符。我们称之为 `(*)`：表达式 `a * (b * c)` 和 `(a * b) * c` 结果必须相同。
- 一个单位元素。我们称之为 `e`，它必须遵守两条法则：`a * e == a` 和 `e * a == a`。

幺半群并不要求这个二元操作符做什么，它只要求这个二元操作符必须存在。因此很多数学结构都是幺半群。如果我们把加号作为二元操作符，0 作为单位元素，那么整数就是一个幺半群。把乘号作为二元操作符，1 作为单位元素，整数就形成了另一个幺半群。

Haskell 中幺半群无所不在。`Data.Monoid` 模块定义了 `Monoid` 类型类。

```
-- file: ch13/Monoid.hs
class Monoid a where
    mempty  :: a                -- the identity
    mappend :: a -> a -> a      -- associative binary operator
```

如果我们把 `(++)` 当做二元操作符，`[]` 当做单位元素，列表就形成了一个幺半群。

```
-- file: ch13/Monoid.hs
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

由于列表和 `DLists` 关系如此紧密，`DList` 类型也必须是一个幺半群。

```
-- file: ch13/DList.hs
instance Monoid (DList a) where
```

(continues on next page)

(continued from previous page)

```
    mempty = empty
    mappend = append
```

在 **ghci** 里试试 `Monoid` 类型类的函数。

```
ghci> "foo" `mappend` "bar"
"foobar"
ghci> toList (fromList [1,2] `mappend` fromList [3,4])
[1,2,3,4]
ghci> mempty `mappend` [1]
[1]
```

Note: 尽管从数学的角度看，整数可以以两种不同的方式作为幺半群，但在 Haskell 里，我们却不能给 `Int` 写两个不同的 `Monoid` 实例：编译器会报告重复实例错误。

如果我们真的需要在同一个类型上实现多个 `Monoid` 实例，我们可以用 `newtype` 创建不同的类型来达到目的。

```
-- file: ch13/Monoid.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype AInt = A { unA :: Int }
    deriving (Show, Eq, Num)

-- monoid under addition
instance Monoid AInt where
    mempty = 0
    mappend = (+)

newtype MInt = M { unM :: Int }
    deriving (Show, Eq, Num)

-- monoid under multiplication
instance Monoid MInt where
    mempty = 1
    mappend = (*)
```

这样，根据使用类型的不同，我们就能得到不同的行为。

```
ghci> 2 `mappend` 5 :: MInt
M {unM = 10}
ghci> 2 `mappend` 5 :: AInt
A {unA = 7}
```

在这一节（The writer monad and lists）中，我们还会继续讨论差异列表和它的幺半群性质。

Note: 跟 functor 规则一样，Haskell 没法替我们检查幺半群的规则。如果我们定义了一个 `Monoid` 实例，我们可以很容易地写一些 QuickCheck 性质来得到一个较高的统计推断，确保代码遵守了幺半群规则。

13.7 通用序列

不论是 Haskell 内置的列表，还是我们前面定义的 `DList`，这些数据结构在不同的地方都有自己的性能短板。为此，`Data.Sequence` 模块定义了 `Seq` 容器类型，对于大多数操作，这种类型能都提供良好的效率保证。

为了避免命名冲突，`Data.Sequence` 模块通常以 `qualified` 的方式引入：

```
Prelude> import qualified Data.Sequence as Seq
Prelude Seq>
```

`empty` 函数用于创建一个空 `Seq`，`singleton` 用于创建只包含单个元素的 `Seq`：

```
Prelude Seq> Seq.empty
fromList []

Prelude Seq> Seq.singleton 1
fromList [1]
```

还可以使用 `fromList` 函数，通过列表创建出相应的 `Seq`：

```
Prelude Seq> let a = Seq.fromList [1, 2, 3]

Prelude Seq> a
fromList [1,2,3]
```

`Data.Sequence` 模块还提供了几种操作符形式的构造函数。但是，在使用 `qualified` 形式载入模块的情况下调用它们会非常难看：

[译注：操作符形式指的是那种放在两个操作对象之间的函数，比如 `2 * 2` 中的 `*` 函数。]

```
Prelude Seq> 1 Seq.<| Seq.singleton 2
fromList [1,2]
```

可以通过直接载入这几个函数来改善可读性：

```
Prelude Seq> import Data.Sequence((><), (<|), (|>))
```

现在好多了：

```
Prelude Seq Data.Sequence> Seq.singleton 1 |> 2
fromList [1,2]
```

一个帮助记忆（<|）和（|>）函数的方法是，函数的『箭头』总是指向被添加的元素：（<|）函数要添加的元素在左边，而（|>）函数要添加的元素在右边：

```
Prelude Seq Data.Sequence> 1 <| Seq.singleton 2
fromList [1,2]

Prelude Seq Data.Sequence> Seq.singleton 1 |> 2
fromList [1,2]
```

不管是从左边添加元素，还是从右边添加元素，添加操作都可以在常数时间内完成。对两个 Seq 进行追加（append）操作同样非常廉价，复杂度等同于两个 Seq 中较短的那个 Seq 的长度的对数。

追加操作由（><）函数完成：

```
Prelude Seq Data.Sequence> let left = Seq.fromList [1, 3, 3]

Prelude Seq Data.Sequence> let right = Seq.fromList [7, 1]

Prelude Seq Data.Sequence> left >< right
fromList [1,3,3,7,1]
```

反过来，如果我们想将 Seq 转换回列表，那么就需要 Data.Foldable 模块的帮助：

```
Prelude Seq Data.Sequence> import qualified Data.Foldable as Foldable
Prelude Seq Data.Sequence Foldable>
```

这个模块定义了一个类型，Foldable ，而 Seq 实现了这个类型：

```
Prelude Seq Data.Sequence Foldable> Foldable.toList (Seq.fromList [1, 2, 3])
[1,2,3]
```

Data.Foldable 中的 fold 函数可以用于对 Seq 进行 fold 操作：

```
Prelude Seq Data.Sequence Foldable> Foldable.foldl' (+) 0 (Seq.fromList [1, 2, 3])
6
```

Data.Sequence 模块还提供了大量有用的函数，这些函数都和 Haskell 列表的函数类似。模块的文档也非常齐全，还提供了函数的时间复杂度信息。

最后的疑问是，既然 Seq 的效率这么好，那为什么它不是 Haskell 默认的序列类型呢？答案是，列表类型更简单，消耗更低，对于大多数应用程序来说，列表已经足够满足需求了。除此之外，列表可以很好地处理惰性环境，而 Seq 在这方面做得还不够好。

CHAPTER

FOURTEEN

第 14 章：MONADS

14.1 简介

在第 7 章：*I/O* 中，我们讨论了 `IO` monad，那时我们刻意把精力集中在如何与外界交互上，并没有讨论 monad 是什么。

在第 7 章：*I/O* 中我们看到 `IO` Monad 确实很好用；除了在语法上不同之外，在 `IO` monad 中写代码跟其他命令式语言基本没有什么区别。

在前面的章节中，我们在解决一些实际问题的时候引入了一些数据结构，很快我们就会知道它们其实就是 monads。我们想告诉你的是，在解决某些问题的时候，monad 通常是一个非常直观且实用的工具。本章我们将定义一些 monads 并告诉你它有多么简单。

14.2 回顾之前代码

14.2.1 Maybe 链

我们先看看我们在第 10 章：代码案例学习：解析二进制数据格式 写的 `parseP5` 函数:

```
-- file: ch10/PNM.hs
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString

-- "nat" here is short for "natural number"
getNat :: L.ByteString -> Maybe (Int, L.ByteString)

getBytes :: Int -> L.ByteString
         -> Maybe (L.ByteString, L.ByteString)

    parseP5 s =
      case matchHeader (L8.pack "P5") s of
        Nothing -> Nothing
```

(continues on next page)

(continued from previous page)

```
        Just s1 ->
          case getNat s1 of
            Nothing -> Nothing
            Just (width, s2) ->
              case getNat (L8.dropWhile isSpace s2) of
                Nothing -> Nothing
                Just (height, s3) ->
                  case getNat (L8.dropWhile isSpace s3) of
                    Nothing -> Nothing
                    Just (maxGrey, s4)
                      | maxGrey > 255 -> Nothing
                      | otherwise ->
                          case getBytes 1 s4 of
                            Nothing -> Nothing
                            Just (_, s5) ->
                              case getBytes (width * height) s5 of
                                Nothing -> Nothing
                                Just (bitmap, s6) ->
                                  Just (Greymap width height maxGrey bitmap, s6)
```

这个函数要是再复杂一点，就要超出屏幕的右边了；当时我们使用 (>>?) 操作符避免了这种情况:

```
-- file: ch10/PNM.hs
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v  >>? f = f v
```

我们对 (>>?) 操作符的类型进行了精挑细选使得它能把一系列返回类型是 Maybe 的函数串联起来；只要一个函数的返回值能和下一个函数的参数类型匹配，我们就能无限串联返回类型是 Maybe 的函数。(>>?) 的函数体把细节隐藏了起来，我们不知道我们通过 (>>?) 串联的函数是由于中间某个函数返回 Nothing 而中断了还是所有函数全部执行了。

14.2.2 隐式状态

(>>?) 被用来整理 parseP5 的结构，但是在解析的时候我们还是要一点一点地处理输入字符串；这使得我们必须把当前处理的值通过一个元组传递下去 [若干个函数串联了起来，都返回 Maybe，作者称之为 Maybe 链]。Maybe 链上的每一个函数把自己处理的结果以及自己没有解析的剩下的字符串放到元组里面, 并传递下去。

```
-- file: ch10/PNM.hs
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
```

(continues on next page)

(continued from previous page)

```
    matchHeader (L8.pack "P5") s        >>?
    \s -> skipSpace ((), s)             >>?
    (getNat . snd)                      >>?
    skipSpace                           >>?
    \(width, s) ->   getNat s           >>?
    skipSpace                           >>?
    \(height, s) ->  getNat s           >>?
    \(maxGrey, s) -> getBytes 1 s       >>?
    (getBytes (width * height) . snd) >>?
    \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)

skipSpace :: (a, L.ByteString) -> Maybe (a, L.ByteString)
skipSpace (a, s) = Just (a, L8.dropWhile isSpace s)
```

我们又碰到了有着重复行为的模式：处理字符串的时候，某个函数消耗部分字符串并返回它处理的结果，同时把剩下的字符串传递给下一个函数继续处理。但是，这个模式比之前的更糟糕：如果我们要在处理链往下传递另外一些额外信息，我们必须把传递的二元组修改为三元组，这几乎要修改这个处理链上的所有元素！

我们把管理当前字符串的任务从处理链上的单个函数移出来，将它 (管理字符串) 转交给串联这些单个函数的函数完成！[译：比如上面的 (>>?)]

```
-- file: ch10/Parse.hs
(==>) :: Parse a -> (a -> Parse b) -> Parse b

firstParser ==> secondParser  =  Parse chainedParser
  where chainedParser initState   =
          case runParse firstParser initState of
            Left errMessage ->
                Left errMessage
            Right (firstResult, newState) ->
                runParse (secondParser firstResult) newState
```

我们把解析状态的细节隐藏在 ParseState 类型中，就连 getState 和 putState 都不会窥视解析状态，所以，无论对 ParseState 做怎样的修改都不会影响已有的代码。

14.3 寻找共同特征

如果我们仔细分析上面的例子，它们好像没有什么共同特点。不过有一点比较明显，它们都想把函数串联起来并试图隐藏细节以便我们写出整洁的代码。然后，我们先不管那些细节，从更粗略的层面去思考一下。

首先，我们看一看类型声明：

```haskell
-- file: ch14/Maybe.hs
data Maybe a = Nothing
             | Just a
```

```haskell
-- file: ch11/Parse.hs
newtype Parse a = Parse {
      runParse :: ParseState -> Either String (a, ParseState)
    }
```

这两个类型的共同特点是它们都有一个类型参数，因此它们都是范型，对具体的类型一无所知。

然后看一看我们给两个类型写的串联函数：

```
ghci> :type (>>?)
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
ghci> :type (==>)
(==>) :: Parse a -> (a -> Parse b) -> Parse b
```

这两个函数的类型非常相似，如果我们把它们的类型构造器替换为一个类型变量，我们会得到一个更加抽象的类型。

```haskell
-- file: ch14/Maybe.hs
chain :: m a -> (a -> m b) -> m b
```

最终，在两种情况下，我们都得到了一个获取一个普通的值，然后把它“注入”到一个目标类型里面去的函数。对于 `Maybe` 类型，这个函数就是它的一个值构造器 `Just` ，“Parse“的注入函数就略微复杂一些。

```haskell
-- file: ch10/Parse.hs
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

我们不用关心它的实现细节，也不管它有多么复杂；重要的是，这些类型都有一个“注入器”函数，它大致长这样：

```haskell
-- file: ch14/Maybe.hs
inject :: a -> m a
```

在 Haskell 里面，正是这三个属性和一些如何使用它们的规则定义了 monad。我们集中总结一下：

1. 一个类型构造器 `m`
2. 一个用于把某个函数的输出串联到另外一个函数输入上的函数，它的类型是 `m a -> (a -> m b) -> m b`

3. 一个类型是 `a -> m a` 类型的函数，它把普通值注入到调用链里面，也就是说，它把类型 `a` 用类型构造器 `m` 包装起来。

Maybe 类型的类型构造器 `Maybe a`，串联函数 `(>>?)` 以及注入函数 `Just` 使 Maybe 成为一个 monad。对于 `Parse` 类型，对应的是类型构造器 `Parse a`，串联函数 `Parse a` 以及注入函数 `identify`。

对于 Monad 的串联函数和注入函数具体应该干什么我们刻意只字未提，因为它几乎不重要。事实上，正是因为 Monad 如此简单，它在 Haskell 里面无处不在。许多常见的编程模式都用到了 monad 结构：传递隐式数据，或是短路求值链。

14.4 Monad 类型类

在 Haskell 里面我们可以使用一个类型类 (typeclass) 来表示“串联”以及“注入”的概念以及它们的类型。标准库的 Predule 模块已经包含了这样一个类型类，也就是 `Monad`。

```
-- file: ch14/Maybe.hs
class Monad m where
    -- chain
    (>>=)  :: m a -> (a -> m b) -> m b
    -- inject
    return :: a -> m a
```

在这里，`(>>=)` 就是我们的串联函数。在串联化（*Sequencing*） 中我们已经介绍了它。通常将这个操作符称呼为“绑定”，因为它把左侧运算的结果绑定到右侧运算的参数上。

我们的注入函数是 `return`，在*Return 的本色* 中讲过，选用 `return` 这个名字有点倒霉。这个关键字在命令式语言中被广泛使用并且有一个非常容易理解的含义。但是在 Haskell 里面它的含义完全不同；具体来说，在函数调用链中间使用 `return` 并不会导致调用链提前中止；我们可以这样理解它：它把纯值 (`a` 类型) 放进 (returns)monads(`m a` 类型) 里。

`(>>=)` 和 `return` 是 `Monad` 这个类型类的核心函数；除此之外，它还定义了另外两个函数。一个函数是 `(>>)`，类似于 `(>>=)`，它的作用也是串联，但是它忽略左侧的值。

```
-- file: ch14/Maybe.hs
    (>>) :: m a -> m b -> m b
        a >> f = a >>= \_ -> f
```

当我们需要按顺序执行一系列操作的，并且不关心先前的计算结果的时候，可以使用这操作符。这样也许看起来让人觉得费解：为什么我们会忽略一个函数的返回值呢，这样有什么意义？回想一下，我们之前定义了一个 `(==>&)` 组合子来专门表达这个概念。另外，考虑一下 `print` 这样的函数，它的返回结果是一个占位符，我们没有必要关心它返回值是什么。

```
ghci> :type print "foo"
print "foo" :: IO ()
```

如果我们使用普通的（>>=）来串联调用，我们必须提供一个新的函数来忽略参数（这个参数是前一个 print 的返回值。）

```
ghci> print "foo" >>= \_ -> print "bar"
"foo"
"bar"
```

但是，如果我们使用（>>）操作符，那么就可以去掉那个没什么用的函数了：

```
ghci> print "baz" >> print "quux"
"baz"
"quux"
```

正如我们上面看到的一样，（>>）的默认实现是通过（>>=）完成的。

Monad 类型类另外一个非核心函数是 fail，这个函数接受一个错误消息然后让函数调用链失败。

Warning: 许多 Monad 实现并没有重写 fail``函数的默认实现，因此在这些 Monad 里面，``fail 将由 error 函数实现。但是由于 error 函数抛出的异常对于调用者来说要么就是无法捕获的，要么就是无法预期的，所以调用 error 并不是一件好事。就算你很清楚在 Monad 使用 fail 在当前场景下是个明智之选，但是依然非常不推荐使用它。当你以后重构代码的时候，很有可能这个 fail 函数在新的语境下无法工作从而导致非常复杂的问题，这种情况太容易发生了。

回顾一下我们在*第 10 章：代码案例学习：解析二进制数据格式* 写的 parse, 里面有一个 Monad 的实例：

```
-- file: ch10/Parse.hs
instance Monad Parse where
    return = identity
    (>>=) = (==>)
    fail = bail
```

14.5 术语解释

可能你对 monad 的某些惯用语并不熟悉，虽然他们不是正式术语，但是很常见；因此有必要了解一下。

- “Monadic” 仅仅表示 “和 Monad 相关的”。一个 monadic 类型就是一个 Monad 类型类的实例；一个 monadic 值就是一个具有 monadic 类型的值。
- 当我们说某个东西 “是一个 monad” 的时候，我们其实表达的意思是 “这个类型是 Monad 这个类型类的实例”；作为 Monad 的实例就有三要素：类型构造器，注入函数，串联函数。
- 同样，当我们谈到 “Foo 这个 monad” 的时候，我们实际上指的是 Foo 这个类型，只不过 Foo 是 Monad 这个类型类的实例。

- Monadic 值的一个别称是“动作”；这个说法可能源自 `I/O Monad` 的引入，`print "foo"` 这样的 monad 值会导致副作用。返回类型是 monadic 值的函数有时候也被称之为动作，虽然这样并不太常见。

14.6 使用新的 Monad

我们在介绍 Monad 的时候，展示了一些之前的代码，并说明它们其实就是 Monad。既然我们慢慢知道 monad 是什么，而且已经见识过 `Monad` 这个类型类；现在就让我们用学到的知识来写一个 Monad 吧。我们先定义它的接口，然后使用它；一旦完成了这些，我们就写出了自己的 Monad!

纯粹的 Haskell 代码写起来非常简洁，但是它不能执行 IO 操作；有时候，我们想记下我们程序的一些操作，但是又不想直接把日志信息写入文件；就这些需求，我们开发一个小型库。

回忆一下我们在*将 glob 模式翻译为正则表达式* 中定义的 `globToRegex` 函数；我们修改它让它能够记住每次它翻译过的句子。我们又回到了熟悉的恐怖场景：比较同一份代码的 Monadic 版本和非 Monadic 版。

首先，我们可以使用一个 `Logger` 类型类把处理结果的类型包装起来。

```
-- file: ch14/Logger.hs
globToRegex :: String -> Logger String
```

14.6.1 信息隐藏

我们将刻意隐藏 `Logger` 模块的实现。

```
-- file: ch14/Logger.hs
module Logger
    (
      Logger
    , Log
    , runLogger
    , record
    ) where
```

像这样隐藏实现有两个好处：它很大程度出上保证了我们对于 Monad 实现的灵活性，更重要的是，这样有一个非常简单的接口。

`Logger` 类型就是单纯的一个类型构造器。我们并没有将它的值构造器导出，因此 `Logger` 模块的使用者没有办法自己创建一个 `Logger` 类型的值，它们对于 `Logger` 类型能做的就是把它写在类型签名上。

`Log` 类型就是一串字符串的别名，这样写是为了让它可读性更好。同时我们使用一串字符串来保持实现的简单。

```
-- file: ch14/Logger.hs
type Log = [String]
```

我们给接口的使用者提供了一个 runLogger 函数来执行某个日志操作，而不是直接给他们一个值构造器。这个函数既回传了日志纪录这个操作，同时也回传了日志信息本身。

```
-- file: ch14/Logger.hs
runLogger :: Logger a -> (a, Log)
```

14.6.2 受控的 Monad

Monad 类型类没有提供任何方法使一个 monadic 的值成为一个普通的值。我们可以使用 return 函数把一个普通的值“注入”到 monad 里面；我们也可以用 (>>=) 操作符把一个 monadic 的值提取出来，但是经过操作符处理之后还是回传一个 monadic 的值。

很多 monads 都有一个或者多个类似 runLogger 的函数；IO monad 是个例外，通常情况下我们只能退出整个程序来脱离这个 monad。

一个 Monad 函数在 monad 内部执行然后向外返回结果；一般来说这些函数是把一个 Monadic 的值脱离 Monad 成为一个普通值的唯一方法。因此，Monad 的创建者对于如何处理这个过程有着完全的控制权。

有的 Monad 有好几个执行函数。在我们这个 Logger 的例子里面，我们可以假设有一些 runLogger 的替代方法：一个仅仅返回日志信息，另外一个可能返回日志操作，然后把日志信息本身丢掉。

14.6.3 日志纪录

当我们执行一个 Logger 动作的时候，代码将调用 record 函数来纪录某些东西。

```
-- file: ch14/Logger.hs
record :: String -> Logger ()
```

由于日志纪录的过程发生在 Monad 的内部，因此 record 这个动作并不返回什么有用的信息（()）

通常 Monad 会提供一些类似 record 这样的辅助函数；这些函数也是我们访问这个 Monad 某些特定行为的方式。

我们的模块也把 Logger 定义为了 Monad 的实例。这个实例里面的定义就是使用 Logger 类型所需要的全部东西。

下面就是使用我们的 Logger 类的一个例子：

```
ghci> let simple = return True :: Logger Bool
ghci> runLogger simple
(True,[])
```

当我们使用 runLogger 函数执行被记录的操作之后，会得到一个二元组。二元组的第一个元素是我们代码的执行结果；第二个元素是我们的日志动作执行的时候纪录信息的列表。由于我们没有纪录任何东西，所以返回的列表是空；来个有日志信息的例子。

```
ghci> runLogger (record "hi mom!" >> return 3.1337)
(3.1337,["hi mom!"])
```

14.6.4 使用 Logger monad

在 Logger monad 里面我们可以剔除通配符到正则表达式的转换，代码如下：

```
-- file: ch14/Logger.hs
globToRegex cs =
    globToRegex' cs >>= \ds ->
    return ('^':ds)
```

然后我们来简单说明一下一些值得注意的代码风格。我们函数体在函数名字下面一行，要这么做，需要添加一些水平的空格；对于匿名函数，我们把它的参数放在另起的一行，这是 monadic 代码通常的组织方式。

回忆一下 (>>=) 的类型：它从 Logger 包装器中中提取出操作符 (>>=) 左边的值，然后把取出来的值传递给右边的函数。右边的操作数函数必须把这个取出来的值用 Logger 包装起来然后回传出去。这个操作正如正如 return 一样：接受一个纯值，然后用 Monad 的类型构造器包装一下返回。

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type (globToRegex "" >>=)
(globToRegex "" >>=) :: (String -> Logger b) -> Logger b
```

就算我们写一个什么都不做的函数，我们也必须使用 return 去包装具有正确类型的值。

```
-- file: ch14/Logger.hs
globToRegex' :: String -> Logger String
globToRegex' "" = return "$"
```

当我们要使用 record 函数纪录某些日志的时候，我们采用 (>>) 而不是 (>>=) 来串联一系列的日志操作。

```
-- file: ch14/Logger.hs
globToRegex' ('?':cs) =
    record "any" >>
    globToRegex' cs >>= \ds ->
    return ('.':ds)
```

(>>) 就是 (>>=) 的一个变种，只不过它会忽略左边操作的结果；由于 record 函数的返回值永远都是 () 因此获取它的返回值没有什么意义，直接使用 >> 更简洁。

另外，我们也可以使用在串联化（*Sequencing*）引入的 do 表示法来整理代码。

```
-- file: ch14/Logger.hs
globToRegex' ('*':cs) = do
    record "kleene star"
    ds <- globToRegex' cs
    return (".*" ++ ds)
```

选择使用 do 表示法还是显式使用（>>=）结合匿名函数完全取决于个人爱好，但是对于长度超过两行的代码来说，几乎所有人都会选择使用 do. 这两种风格有一个非常重要的区别，我们将会在还原 *do* 的本质 里面介绍。

对于解析单个字符的情况，monadic 的代码几乎和普通的一样：

```
-- file: ch14/Logger.hs
globToRegex' ('[':'!':c:cs) =
    record "character class, negative" >>
    charClass cs >>= \ds ->
    return ("[^" ++ c : ds)
globToRegex' ('[':c:cs) =
    record "character class" >>
    charClass cs >>= \ds ->
    return ("[" ++ c : ds)
globToRegex' ('[':_) =
    fail "unterminated character class"
```

14.7 同时使用 puer 和 monadic 代码

迄今为止我们看到的 Monad 好像有一个非常明显的缺陷：Monad 的类型构造器把一个值包装成一个 monadic 的值，这样导致在 monad 里面使用普通的纯函数有点困难。举个例子，假设我们有一段运行在 monad 里面的代码，它所做的就是返回一个字符串：

```
ghci> let m = return "foo" :: Logger String
```

如果我们想知道字符串的长度是多少，我们不能直接调用 `length` 函数：因为这个字符串被 `Logger` 这个 monad 包装起来了，因此类型并不匹配。

```
ghci> length m

<interactive>:1:7:
    Couldn't match expected type `[a]'
           against inferred type `Logger String'
    In the first argument of `length', namely `m'
```

(continues on next page)

(continued from previous page)

```
    In the expression: length m
    In the definition of `it': it = length m
```

我们能做的事情就是下面这样：

```
ghci> :type   m >>= \s -> return (length s)
m >>= \s -> return (length s) :: Logger Int
```

我们使用 (>>=) 把字符串从 monad 里面取出来，然后使用一个匿名函数调用 length 接着用 return 把这个字符串重新包装成 Logger 。

由于这种形式的代码经常在 Haskell 里面出现，因此已经有一个类似的操作符存在了。在*Functor* 简介 里面我们介绍了 *lifting* 这种技术；把一个纯函数 Lift 为一个函子通常意味着从一个带有上下文的特殊值里面取出那个值，然后使用这个普通的值调用纯函数，得到结果之后用特定的类型构造器包装成原来有着上下文的特殊值。

在 monad 里面，我们需要干同样的一件事。由于 Monad 这个类型类已经提供了 (>>=) 和 return 这两个函数处理 monadic 的值和普通值之间的转换，因此 liftM 函数不需要知道 monad 的任何实现细节。

```
-- file: ch14/Logger.hs
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= \i ->
            return (f i)
```

当我们把一个类型声明为 Functor 这个类型类的实例之后，我们必须根据这个特定的类型实现对应的 fmap 函数；但是，由于 (>>=) 和 return 对 monad 的进行了抽象，因此 ‘‘liftM‘‘ 不需要知道任何 monad 的任何实现细节。我们只需要实现一次并配上合适的类型签名即可。

在标准库的 Control.Monad 模块里面已经为我们定义好了 liftM 函数。

我们来看看使用 liftM 对于提升我们代码可读性有什么作用；先看看没有使用 liftM 的代码：

```
-- file: ch14/Logger.hs
charClass_wordy (']':cs) =
    globToRegex' cs >>= \ds ->
    return (']':ds)
charClass_wordy (c:cs) =
    charClass_wordy cs >>= \ds ->
    return (c:ds)
```

然后我们用 liftM 去掉那些 (>>=)) 和匿名函数：

```
-- file: ch14/Logger.hs
charClass (']':cs) = (']':) `liftM` globToRegex' cs
charClass (c:cs) = (c:) `liftM` charClass cs
```

正如 fmap 一样，我们通常用中缀的方式调用 liftM 。可以用这种方式来阅读这个表达式：把右边操作得到的 monadic 的值应用到左边的纯函数上。

liftM 函数实在是太有用了，因此 Control.Monad 定义了它的几个变种，这些变种可以处理更长的参数；我们可以看一看 globToRegex 这个函数的最后一个分句：

```
-- file: ch14/Logger.hs
globToRegex' (c:cs) = liftM2 (++) (escape c) (globToRegex' cs)

escape :: Char -> Logger String
escape c
    | c `elem` regexChars = record "escape" >> return ['\\',c]
    | otherwise           = return [c]
  where regexChars = "\\+()^$.{}]|"
```

上面这段代码用到的 liftM2 函数的定义如下：

```
-- file: ch14/Logger.hs
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 =
    m1 >>= \a ->
    m2 >>= \b ->
    return (f a b)
```

它首先执行第一个动作，接着执行第二个操作，然后把这两个操作的结果组合起来应用到那个纯函数上并包装返回的结果。Control.Monad 里面定义了 liftM liftM2 直到 liftM5 。

14.8 关于 Monad 的一些误解

我们已经见识过很多 Monad 的例子并且对 monad 也有一些感觉了；在继续探讨 monad 之前，有一些广为流传的关于 monad 的观念需要澄清。你肯定经常听到这些说法，因此你可能已经有一些很好的理由来反驳这些谬论了。

- *Monads 很难理解*我们已经从好几个实例的问题来说明 Monad 是如何工作的了，并且我们已经知道理解 monad 最好的方式就是先通过一些具体的例子来进行解释，然后抽象出这些这些例子共同的东西。
- *Monads 仅仅用于 I/O 操作和命令式代码*虽然我们在 Haskell 的 IO 里面使用 Monad，但是 Monad 在其他的地方也非常有用。我们已经通过 monad 串联简单的计算，隐藏复杂的状态以及纪录日志了；然而，Monad 的作用我们还只看到冰山一角。
- *只有 Haskell 才有 Monad* Haskell 有可能是显式使用 Monad 最多的语言，但是在别的语言里面也存在，从 C++ 到 OCaml。由于 Haskell 的 do 表示法，强大的类型系统以及语言的语法使得 Monad 在 Haskell 里面非常容易处理。
- *Monads 使用来控制求值顺序的*

14.9 创建 Logger Monad

Logger 类的定义非常简单：

```
-- file: ch14/Logger.hs
newtype Logger a = Logger { execLogger :: (a, Log) }
```

它其实就是一个二元组，第一个元素是执行动作的结果，第二元素是我们执行动作的时候纪录的日志信息列表。

我们使用 newtype 关键字把二元组进行了包装使它的类型更加清晰易读。runLogger 函数可以从这个 Monad 里面取出这个元组里面的值；这个函数其实是 execLogger 的一个别名。

```
-- file: ch14/Logger.hs
runLogger = execLogger
```

record 这个函数将为接收到的日志信息创建一个只包含单个元素的列表。

```
-- file: ch14/Logger.hs
record s = Logger ((), [s])
```

这个动作的结果是 () 。

让我们以 return 开始，构建 Monad 实例；先尝试一下：它什么都不记录，然后把结果存放在二元组里面。

```
-- file: ch14/Logger.hs
instance Monad Logger where
    return a = Logger (a, [])
```

(>>=) 的定义更有趣，当然它也是 monad 的核心。(>>=) 把一个普通的值和一个 monadic 的函数结合起来，得到新的运算结果和一个新的日志信息。

```
-- file: ch14/Logger.hs
    -- (>>=) :: Logger a -> (a -> Logger b) -> Logger b
    m >>= k = let (a, w) = execLogger m
                  n      = k a
                  (b, x) = execLogger n
              in Logger (b, w ++ x)
```

我们看看这段代码里面发生了什么。首先使用 runLogger 函数从动作 m 取出结果 a ，然后把它传递给 monadic 函数 k; 接着我们又取出 b ； 最后把 w 和 x 拼接得到一个新的日志。

14.9.1 顺序的日志，而不是顺序的求值

我们定义的 (>>=) 保证了新输出的日志一定在之前的输出的日志之后。但是这并不意味着 a 和 b 的求值是顺序的：(>>=) 操作符是惰性求值的。

正如 Monad 的很多其他行为一样，求值的严格性是由 Monad 的实现者控制的，并不是所有 Monad 的共同性质。事实上，有一些 Monad 同时有几种特性，每一种都有着不同程度的严格性（求值）。

14.9.2 Writer monad

我们创建的 Logger monad 实际上是标准库里面 Writer Monad 的一个特例；Writer Monad 可以在 mtl 包里面的 Control.Monad.Writer 模块找到。我们会在*第 6 章：使用类型类* 里面介绍 Writer 的用法。

14.10 Maybe monad

Maybe 应该是最简单的 Monad 了。它代表了一种可能不会产生计算结果的计算过程。

```
-- file: ch14/Maybe.hs
instance Monad Maybe where
    Just x >>= k   =  k x
    Nothing >>= _ =  Nothing

    Just _ >> k    =  k
    Nothing >> _   =  Nothing

    return x       =  Just x

    fail _         =  Nothing
```

当我们使用 (>>=) 或者 (>>) 串联一些 Maybe 计算的时候，如果这些计算中的任何一个返回了 Nothing，(>>=) 和 (>>) 就不会对余下的任何计算进行求值。

值得一提的是，整个调用链并不是完全短路的。每一个 (>>=) 或者 (>>) 仍然会匹配它左边的 Nothing 然后给右边的函数一个 Nothing, 直到达到调用链的末端。这一点很容易被遗忘：当调用链中某个计算失败的时候，之前计算的结果，余下的调用链以及使用的 Nothing 值在运行时的开销是廉价的，但并不是完全没有开销。

14.10.1 执行 Maybe monad

适合执行 Maybe Monad 的函数是 maybe (“执行” 一个 monad 意味着取出 Monad 里面包含的值，移除 Monad 类的包装)

```haskell
-- file: ch14/Maybe.hs
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x
```

如果第三个参数是 Nothing，maybe 将使用第一个参数作为返回值；而第二个参数则是在 Just 值构造器里面进行包装值的函数。

由于 Maybe 类型非常简单，直接对它进行模式匹配和调用 maybe 函数使用起来差不多，在不同的场景下，两种方式都有各自的优点。

14.10.2 使用 Maybe，以及好的 API 设计方式

下面是一个使用 Maybe 的例子。给出一个顾客的名字，找出它们手机号对应的账单地址。

```haskell
-- file: ch14/Carrier.hs
import qualified Data.Map as M

type PersonName = String
type PhoneNumber = String
type BillingAddress = String
data MobileCarrier = Honest_Bobs_Phone_Network
                   | Morrisas_Marvelous_Mobiles
                   | Petes_Plutocratic_Phones
                     deriving (Eq, Ord)

findCarrierBillingAddress :: PersonName
                          -> M.Map PersonName PhoneNumber
                          -> M.Map PhoneNumber MobileCarrier
                          -> M.Map MobileCarrier BillingAddress
                          -> Maybe BillingAddress
```

我们的第一个实现使用 case 表达式，用它完成的代码相当难看，差不多超出了屏幕的右边。

```haskell
-- file: ch14/Carrier.hs
variation1 person phoneMap carrierMap addressMap =
    case M.lookup person phoneMap of
      Nothing -> Nothing
      Just number ->
          case M.lookup number carrierMap of
            Nothing -> Nothing
            Just carrier -> M.lookup carrier addressMap
```

模块 Data.Map 的函数 lookup 返回一个 monadic 的值：

```
ghci> :module +Data.Map
ghci> :type Data.Map.lookup
Data.Map.lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

换句话说，如果给定的 key 在 map 里面存在，那么 `lookup` 函数使用 `return` 把这个值注入到 monad 里面去；否则就会调用 `fail` 函数。这是这个 API 一个有趣的实现，虽然有人觉得它很糟糕。

- 这样设计好的一方式是，根据具体 Monad 实现的不同，查找成功和失败的结果是可以根据不同需求定制的；而且，`lookup` 函数本身对于具体的这些行为完全不用关心。
- 坏处就是，在有些 Monad 里面调用 `fail` 会直接抛出让人恼火的异常；之前我们就警告过最好不要使用 `fail` 函数，这里就不在赘述了。

实际上，每个人都使用 `Maybe` 类型作为 `lookup` 函数的返回结果；这样一个简单的函数对于它的返回结果提供了它并不需要的通用性：其实 `lookup` 应该直接返回 `Maybe` 。

先放下 API 设计的问题，我们来处理一下我们之前用 `case` 编写的丑陋代码。

```
-- file: ch14/Carrier.hs
variation2 person phoneMap carrierMap addressMap = do
  number <- M.lookup person phoneMap
  carrier <- M.lookup number carrierMap
  address <- M.lookup carrier addressMap
  return address
```

如果这其中的任何一个查找失败，`(>>=)` 和 `(>>)` 的定义告诉我们整个运算的结果将会是 `Nothing`; 就和我们显式使用 `case` 表达式结果一样。

使用 Monad 的版本的代码更加整洁，但是其实 `return` 是不必要的；从风格上说，使用 `return` 让代码看起来更加有规律，另外熟悉命令式编程的程序员可能对它感觉更熟悉；但其实上它是多余的；下面是与它等价的版本：

```
-- file: ch14/Carrier.hs
variation2a person phoneMap carrierMap addressMap = do
  number <- M.lookup person phoneMap
  carrier <- M.lookup number carrierMap
  M.lookup carrier addressMap
```

14.11 List Monad

`Maybe` 类型代表有可能有值也可能没有值的计算；也有的情况下希望计算会返回一系列的结果，显然，List 正适合这个目的。List 的类型带有一个参数，这暗示它有可能能作为一个 monad 使用；事实上，我们确实能把它当作 monad 使用。

先不看标准库的 Prelude 对于 List monad 的实现，我们自己来看看一个 List 的 monad 应该是什么样的。这个过程很简单：首先看 (>>=) 和 return 的类型，然后进行一些替换操作，看看我们能不能使用一些熟悉的 list 函数。

return 和 (>>=) 这两个函数里面显然 return 比较简单。我们已经知道 return 函数接受一个类型，然后把它用类型构造器 m 包装一下然后产生一个新的类型 m a. 在 List 这种情况下，这个类型构造器就是 []. 把这个类型构造器使用 List 的类型构造器替换掉我们就得到了类型 [] a (当然，这样写是非法的!)；可以把它写成更加熟悉的形式 [a].

现在我们知道 list 的 return 函数的类型应该是 a -> [a] 。对于这种类型的函数，只有少数那么几种实现的可能：要么它返回一个空列表，要么返回一个单个元素的列表，或者一个无穷长度的列表。基于我们现在对于 Monad 的理解，最有可能的实现方式应该是返回单个元素的列表：它不会丢失已有信息，也不会无限重复。

```
-- file: ch14/ListMonad.hs
returnSingleton :: a -> [a]
returnSingleton x = [x]
```

如果我们对 (>>=) 的类型签名进行和 return 类似的替换，我们会得到：[a] -> (a -> [b]) -> [b] . 这看起来和 map 非常相似。

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

map 函数的参数顺序和它有点不对应，我们可以改成这样：

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type flip map
flip map :: [a] -> (a -> b) -> [b]
```

但是还是有一点小问题：flip map 的第二个参数的类型是 a -> b ，但是 (>>=) 的第二个参数的类型是 a -> [b] , 应该怎么办呢?

我们对类型进行更多的替换，看看会发生什么。flip map 这个函数能把任何类型 b 作为返回结果；如果我们使用 [b] 来替换 b ，这个函数的类型就成了 a -> (a -> [n]) -> [[b]]. 换句话说，如果我们使用 map ，将一个列表与一个返回列表的函数进行映射，我们会得到一个包含列表的列表。

```
ghci> flip map [1,2,3] (\a -> [a,a+100])
[[1,101],[2,102],[3,103]]
```

有趣的是，我们这么做并没有让 flip map 和 (>>=) 的类型更加匹配一点；(>>=) 的类型是 [a] -> (a -> [b]) -> [b] ；然而，flip map 如果对返回列表的函数进行 map 那么它的类型签名是 [a] -> (a

`-> [b]) -> [[b]]`. 在类型上依然是不匹配的，我们仅仅是把不匹配的类型从中间转移到了末尾。但是，我们的努力并没有白费：我们现在其实只需要一个能把 `[[b]]` 转化成 `[b]` 的函数就好了。很明显 `concat` 符合我们的要求。

```
ghci> :type concat
concat :: [[a]] -> [a]
```

`(>>=)` 的类型告诉我们应该把 `map` 的参数进行翻转，然后使用 `concat` 进行处理得到单个列表。

```
ghci> :type \xs f -> concat (map f xs)
\xs f -> concat (map f xs) :: [a] -> (a -> [a1]) -> [a1]
```

事实上 lists 的 `(>>=)` 定义就是这样：

```
-- file: ch14/ListMonad.hs
instance Monad [] where
    return x = [x]
        xs >>= f = concat (map f xs)
```

它使用函数 `f` 对列表 `xs` 的每一个元素 `x` 进行处理，然后把得到的结果拼接起来得到单个列表。

现在我们已经搞定了 List 这个 Monad 的两个核心函数，另外两个非核心函数实现起来就很容易了：

```
-- file: ch14/ListMonad.hs
    xs >> f = concat (map (\_ -> f) xs)
    fail _ = []
```

14.11.1 理解 List monad

List monad 与 Haskell 的另外一个工具——列表推导非常相似。我们可以通过计算两个列表的笛卡尔集来说明它们之间的相似性。首先，我们写一个列表推导：

```
-- file: ch14/CartesianProduct.hs
comprehensive xs ys = [(x,y) | x <- xs, y <- ys]
```

这里我们使用大括号语法来表示 monadic 代码，这样会告诉我们 monadic 代码和列表推导该有多么相似。

```
-- file: ch14/CartesianProduct.hs
monadic xs ys = do { x <- xs; y <- ys; return (x,y) }
```

唯一的一个不同点是使用 monadic 代码计算的结果在一系列表达式的末尾得到；而列表推导的结果表示在最开始。除此之外，这个函数计算的结果是完全相同的。

```
ghci> comprehensive [1,2] "bar"
[(1,'b'),(1,'a'),(1,'r'),(2,'b'),(2,'a'),(2,'r')]
ghci> comprehensive [1,2] "bar" == monadic [1,2] "bar"
True
```

一开始肯定对列表 monad 非常迷惑，我们一起看一下 monadic 代码计算笛卡尔集的过程。

```
-- file: ch14/CartesianProduct.hs
blockyDo xs ys = do
    x <- xs
    y <- ys
    return (x, y)
```

x 每次取列表 xs 的一个值，y 每次取列表 ys 的一个值，然后组合在一起得到最终结果；事实上，这就是两层嵌套循环！这也说明了关于 monad 的一个很重要的事实：除非你知道 monad 内部是如何执行的，否则你将无法预期 monadic 代码的行为。

我们再进一步观察这个代码；首先去掉 do 表示法；稍微改变一下代码的结构让它看起来更像一个嵌套循环。

```
-- file: ch14/CartesianProduct.hs
blockyPlain xs ys =
    xs >>=
    \x -> ys >>=
    \y -> return (x, y)

blockyPlain_reloaded xs ys =
    concat (map (\x ->
                 concat (map (\y ->
                              return (x, y))
                             ys))
            xs)
```

如果 xs 的值是 [1, 2, 3]，那么函数体的前两行会依次把 x 值绑定为 1，2 和 3；如果 ys 的值是 [True, False]；那么最后一行会被求值六次：一次是 x 为 1，y 值为 True；然后是 x 值为 1，y 的值为 False；一直继续下去。return 表达式把每个元组包装成一个单个列表的元素。

14.11.2 使用 List Monad

给定一个整数，找出所有的正整数对，使得它们两个积等于这个整数；下面是这个问题的简单解法：

```
-- file: ch14/MultiplyTo.hs
guarded :: Bool -> [a] -> [a]
guarded True  xs = xs
```

(continues on next page)

(continued from previous page)

```
guarded False _  = []

multiplyTo :: Int -> [(Int, Int)]
multiplyTo n = do
  x <- [1..n]
  y <- [x..n]
  guarded (x * y == n) $
    return (x, y)
```

使用 **ghci** 验证结果：

```
ghci> multiplyTo 8
[(1,8),(2,4)]
ghci> multiplyTo 100
[(1,100),(2,50),(4,25),(5,20),(10,10)]
ghci> multiplyTo 891
[(1,891),(3,297),(9,99),(11,81),(27,33)]
```

14.12 还原 do 的本质

Haskell 的 do 表示法实际上是个语法糖：它给我们提供了一种不使用（>>=）和匿名函数来写 monadic 代码的方式。去除 do 语法糖的过程就是把它翻译为（>>=）和匿名函数。

去除 do 语法糖的规则非常简单。我们可以简单的把编译器想象为机械重复地对这些 do 语句块执行这些规则直到没有任何 do 关键字为止。

do 关键字后面接单个动作（action）直接翻译为动作本身。

```
-- file: ch14/Do.hs
doNotation1 =
    do act
```

```
-- file: ch14/Do.hs
translated1 =
    act
```

do 后面包含多个动作（action）的表示是这样的：首先是第一个动作，但是接一个（>>）操作符，然后一个 do 关键字；最后接剩下的动作。当我们对 do 语句块重复应用这条规则的时候，整个 do 语句快就会被（>>）串联起来。

```
-- file: ch14/Do.hs
doNotation2 =
```

(continues on next page)

(continued from previous page)

```
    do act1
       act2
       {- ... etc. -}
       actN
```

```
-- file: ch14/Do.hs
translated2 =
    act1 >>
    do act2
       {- ... etc. -}
       actN

finalTranslation2 =
    act1 >>
    act2 >>
    {- ... etc. -}
    actN
```

<- 标记需要额外注意。在 <- 的左边是一个正常的 Haskell 模式，可以是单个变量或者更复杂的东西；但是这里不允许使用模式匹配的守卫 (guards):

```
-- file: ch14/Do.hs
doNotation3 =
    do pattern <- act1
       act2
       {- ... etc. -}
       actN
```

```
-- file: ch14/Do.hs
translated3 =
    let f pattern = do act2
                       {- ... etc. -}
                       actN
        f _       = fail "..."
    in act1 >>= f
```

这种情况会被翻译为声明了一个名字唯一的局部函数 (上面的例子里面我们仅仅使用了 `f` 这个名字) 的 `let` 表达式；<- 右边的动作会用 (>>=) 和这个局部函数串联起来。

要注意的是，如果模式匹配失败，`let` 表达式会调用 Monad 的 `fail` 函数；下面是一个使用 `Maybe` monad 的例子。

```haskell
-- file: ch14/Do.hs
robust :: [a] -> Maybe a
robust xs = do (_:x:_) <- Just xs
               return x
```

Maybe monad 里面 fail 的实现是返回一个 Nothing 。如果上面的代码模式匹配失败，那么整个计算结果就会是 Nothing .

```
ghci> robust [1,2,3]
Just 2
ghci> robust [1]
Nothing
```

当我们在 do 块里面使用 let 表达式的时候，可以省略掉 in 关键字；但是 let 后面的语句必须和它对齐。

```haskell
-- file: ch14/Do.hs
doNotation4 =
    do let val1 = expr1
           val2 = expr2
           {- ... etc. -}
           valN = exprN
       act1
       act2
       {- ... etc. -}
       actN
```

```haskell
-- file: ch14/Do.hs
translated4 =
    let val1 = expr1
        val2 = expr2
        valN = exprN
    in do act1
          act2
          {- ... etc. -}
          actN
```

14.12.1 Monads: 可编程分号

在缩进规则并不是必需 里面提到过缩进排版是 Haskell 的标准，但是这并不是必要的。我们可以使用 do 表示法来替代缩进排版。

```haskell
-- file: ch14/Do.hs
semicolon = do
```

(continues on next page)

(continued from previous page)

```
  {
    act1;
    val1 <- act2;
    let { val2 = expr1 };
    actN;
  }
```

```
-- file: ch14/Do.hs
semicolonTranslated =
    act1 >>
    let f val1 = let val2 = expr1
                 in actN
        f _ = fail "..."
    in act2 >>= f
```

虽然很少人有这么用，但是在单个表达式里面显式地使用分号容易让人产生这种感觉：monads 是一种 “可编程的分号”，因为在每个 monad 里面（>>=）和（>>）的行为都是不一样的。

14.12.2 为什么要 sugar-free

当我们在代码里面显式使用（>>=）的时候，它提醒我们在使用组合子组合函数而不是简单的序列化动作。

如果你对 monad 感觉还很陌生，那么我建议你多显式地使用（>>=）而不是 do 语法来写 monadic 的代码。这些重复对于大多数的程序员来说都能帮助理解。

当熟悉了 monad 的时候，你可以按照需要选择你自己的风格；但是永远不要再同一个函数里面混用 do 和（>>=）。

不管你用不用 do 表示法，（=<<）函数经常被使用；它就是（>>=）的参数翻转版本。

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type (=<<)
(=<<) :: (Monad m) => (a -> m b) -> m a -> m b
```

如果想把 monadic 函数按照通常 Haskell 从右往左结合起来的话，那么（=<<）非常有用。

```
-- file: ch14/CartesianProduct.hs
wordCount = print . length . words =<< getContents
```

14.13 状态 monad

在*第 10 章：代码案例学习：解析二进制数据格式*里面我们说 `Parse` 是一个 monad。`Parser` 有两个完全不同的角度像 Monad，其一是它在解析失败时候的行为——我们使用 `Either` 表达；其二是它携带这一些隐式的状态信息（每次被部分消耗的 `ByteString`.

在 Haskell 里面读写状态这种场景太常见了，因此标准库提供了一个叫做 `State` 的 monad 解决这个问题。在 `Control.Monad.State` 这个模块可以找到它。

我们的 `Parse` 类型能携带一个 `ByteString` 类型的状态，`State` monad 可以携带任意类型的状态。姑且把这个未知状态的类型记为 `s`.

我们能对一个状态做什么？给定一个状态的值，我们可以查看这个状态，产生一个结果然后返回一个新的状态。假设计算的结果类型是 `a`. 那么表达这个过程的类型就是 `s -> (a, s)`：接受一个状态 `s` 对它进行某些操作，返回结果 `a` 和新状态 `s`.

14.13.1 自己定义 State monad

我们先自己实现一个 State monad，然后看看标准库的实现是什么样的。首先我们从类型定义开始，正如上面我们已经讨论过的，State 的类型定义如下：

```
-- file: ch14/SimpleState.hs
type SimpleState s a = s -> (a, s)
```

我们定义的 monad 是把一个状态转换为另外一个状态的函数，在转化的过程中产生一个计算结果。因此，state monad 也经常被称为状态转换 monad。

在这一章的开始，我们说过 monad 有一个带单个类型参数的类型构造器，但是这里我们有两个类型参数。理解这里的关键是，我们可以把类型构造器像使用函数一样部分应用（partially apply）；下面是一个最简单的例子。

```
-- file: ch14/SimpleState.hs
type StringState a = SimpleState String a
```

这里我们把类型变量 `s` 固定为了 `String` 类型。`StringState` 还带有一个类型参数 `a`；这样就能比较明显的看出来这个类型与 Monad 类型构造器比较匹配了。换句话说，现在 monad 的类型构造器是 `SimpleState s`，而不是单独的 `SimpleState`.

实现这个 State monad 接下来要做的就是定义 `return` 函数。

```
-- file: ch14/SimpleState.hs
returnSt :: a -> SimpleState s a
returnSt a = \s -> (a, s)
```

这里 return 函数所做的就是接受一个结果和当前状态，把它包装成一个二元组，然后返回。你现在应该已经习惯了 Haskell 把带有多个参数的函数当成一系列单个参数函数的串联调用，以下是另一种更直观的写法：

```
-- file: ch14/SimpleState.hs
returnAlt :: a -> SimpleState s a
returnAlt a s = (a, s)
```

实现自定义的 State monad 最后一步就是定义 (>>=) 。下面是标准库的 State monad 对于 (>>=) 的实现：

```
-- file: ch14/SimpleState.hs
bindSt :: (SimpleState s a) -> (a -> SimpleState s b) -> SimpleState s b
bindSt m k = \s -> let (a, s') = m s
                   in (k a) s'
```

这些单个参数的变量不太容易懂，先把它们换成一些更可读的名字。

```
-- file: ch14/SimpleState.hs
-- m == step
-- k == makeStep
-- s == oldState

bindAlt step makeStep oldState =
    let (result, newState) = step oldState
    in  (makeStep result) newState
```

14.13.2 读取和修改状态

(>>=) 和 return 的定义仅仅转移状态，但是并不对状态内部做任何事情。因此我们需要一些简单的辅助函数来对状态进行操作。

```
       -- file: ch14/SimpleState.hs
       getSt :: SimpleState s s
       getSt = \s -> (s, s)

``getSt`` 函数就是接受当前状态并把它作为计算结果和状态一并返回； ``putSt`` 函数忽略当前状态并使用一个新的状态取代它。
```

14.13.3 真正的 State monad 定义

我们之前实现的 SimpleState 仅仅使用了类型别名而不是使用一个新的类型；如果我们当时就使用 newtype 包装一个新的类型，那么对于这个类型的处理会使我们的代码不太容易懂。

要定义一个 Monad 的实例，除了实现 (>>=) 和 return 还要提供一个合适的类型构造器。这正是标准库的 State Monad 的做法：

```
-- file: ch14/State.hs
newtype State s a = State {
      runState :: s -> (a, s)
    }
```

这里所做的就是把 s -> (a, s) 类型用 State 构造器包装起来。通过使用 Haskell 的纪录语法来定义新类型，我们自动获得了一个 runState 函数来从类型构造器里面提取状态值。runState 的类型是 ‘‘ State s a -> s -> (a, s)‘‘

标准库的 State monad 中 return 的定义和我们的 SimpleState 的 return 定义基本相同，只不过这里使用 State 构造器包装了一下状态函数。

```
-- file: ch14/State.hs
returnState :: a -> State s a
returnState a = State $ \s -> (a, s)
```

由于 (>>=) 要使用 runState 函数来提取 State 的值，因此它的的定义略微复杂一些。

```
-- file: ch14/State.hs
bindState :: State s a -> (a -> State s b) -> State s b
bindState m k = State $ \s -> let (a, s') = runState m s
                              in runState (k a) s'
```

这个函数与我们之前在 SimpleState 里面定义的 bindSt 函数唯一的不同是它有提取和包装一些值的操作。

同样，我们也修改了读取和修改状态的函数（提取和包装了一些值）：

```
-- file: ch14/State.hs
get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> ((), s)
```

14.13.4 使用 State monad 生成随机数

之前我们使用 Parse 解析二进制数据，当时我们把要管理的状态直接放在了 Parse 类型里面。

其实 State monad 可以接受任意的类型作为状态参数，我们可以提供这个状态类型，比如 State ByteString.

如果你有命令式编程语言的背景的话，相对于别的很多 monad，你可能对 State 这个 monad 更加熟悉。毕竟命令式语言所做的就是携带和转移一些隐式的状态，比如读写某些部分，通过赋值修改一些东西；这正是 State monad 所做的。

既然这样，我们不用费力地解释怎么使用 State monad 了，直接来个实际的例子就好：生成伪随机数。在命令式编程语言里面，通常有一些很方便使用的均匀分布的伪随机数源；比如在 C 语言标准库里面，有一个 rand 函数使用一个全局的状态生成伪随机数。

Haskell 标准库里面生成伪随机数的模块叫做 System.Random ，它可以生成任意类型的随机数，而不仅仅是数值类型。这个模块提供了一些非常实用的函数。比如与 C 语言里面 rand 等价的函数如下：

```
-- file: ch14/Random.hs
import System.Random

rand :: IO Int
rand = getStdRandom (randomR (0, maxBound))
```

(randomR 函数接受一个希望生成的随机数所在范围的闭区间。)

System.Random 模块提供了一个 RandomGen 类型类，它允许我们自行定义一个新的随机整数源。StdGen 类型是标准的 RandomGen 的实例，它可以生成伪随机数值。如果我们有一个外部的真实可靠的随机数源，我们可以创建一个 RandomGen 的实例来创建真实的随机数，而不是使用伪随机数。

Random 这个类型类展示了如何给特定的类型生成随机数值。这个模块给所有常见的简单类型创建了 Random 的实例。

顺便说下，前面定义的 rand 函数也会读取和修改 IO monad 中内置的全局随机数生成器。

14.13.5 实用纯函数生成随机数的尝试

我们一直尽量避免使用 IO monad，如果仅仅是为了生成随机数就要打破这一点就有点不好意思了。实际上，System.Random 模块里面提供了一些纯函数来生成随机数。

使用传统纯函数的缺点是，我们得获取或者手动创建一个随机数生成器，然后把它传递到需要得地方，最终调用这个纯函数的时候回传一个新的随机数生成器：要记住的是，我们是纯函数，所以不能修改已经存在的随机数生成器。

如果我们不管不变性而是直接复用原来的随机数生成器，那么每次我们调用这个函数都会得到完全一样的“随机数”。

```
-- file: ch14/Random.hs
twoBadRandoms :: RandomGen g => g -> (Int, Int)
twoBadRandoms gen = (fst $ random gen, fst $ random gen)
```

```
ghci> twoBadRandoms `fmap` getStdGen
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package random-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
(945769311181683171,945769311181683171)
```

上面的 random 函数有一个默认的随机数生成范围，而不是像 randomR 一样接受用户传递的参数范围；getStdGen 函数从 IO monad 里面获取全局的标准数据生成器的值。

不幸的是，如果我们把第一个随机数生成之后新的生成器的值正确地传递给第二个随机数的生成过程，代码就不太可读了，下面是个简单的例子：

```
-- file: ch14/Random.hs
twoGoodRandoms :: RandomGen g => g -> ((Int, Int), g)
twoGoodRandoms gen = let (a, gen') = random gen
                         (b, gen'') = random gen'
                     in ((a, b), gen'')
```

现在我们学到了 State monad, 它好像是个比较好的解决办法。state monad 允许我们整洁地管理可变的状态，并且保证这部分代码与任何诸如修改文件，连接网络等副作用操作分离开来；这样让我们能够更加容易地思考代码的行为。

14.13.6 state monad 里面的随机数值

下面是一个使用 StdGen 作为状态的 state monad：

```
-- file: ch14/Random.hs
type RandomState a = State StdGen a
```

上面的类型别名不是必要的，但是很有用；其一它可以让我们少敲几个字符，其二，如果我们想使用别的随机数生成器而不是 StdGen ，我们可以少修改一些类型签名。

有了 RandomState ，生成随机数值就是获取当前的随机数生成器，使用它然后用新的随机数生成器修改当前状态就行了。

```
-- file: ch14/Random.hs
getRandom :: Random a => RandomState a
getRandom =
  get >>= \gen ->
  let (val, gen') = random gen in
  put gen' >>
  return val
```

现在我们可以用之前学到的知识写一些 monadic 的代码来生成一对随机数：

```
-- file: ch14/Random.hs
getTwoRandoms :: Random a => RandomState (a, a)
getTwoRandoms = liftM2 (,) getRandom getRandom
```

14.13.7 练习

1. 用 do 重写 getRandom 函数

14.13.8 运行 state monad

之前提到过，每个 monad 都有他自己的求值函数；在 state monad 里面，有几个求值函数可供选择。

1. runState 返回求值结果和最终状态
2. evalState 只返回结果
3. execState 只返回最终状态

   evalState 和 execState 函数其实就是 runState 和 fst , snd 函数的简单组合。所以三个里面最重要的是要记住 runState .

下面是实现 getTwoRandoms 一个完整的例子：

```
-- file: ch14/Random.hs
runTwoRandoms :: IO (Int, Int)
runTwoRandoms = do
  oldState <- getStdGen
  let (result, newState) = runState getTwoRandoms oldState
  setStdGen newState
  return result
```

14.13.9 管理更多的状态

很难想象针对单个状态我们竟写了这么多有趣的代码，当我们想一次性纪录多个状态的时候，通常的办法是把这些状态放在一个数据结构里面管理。下面是一个纪录我们生成随机数数目的例子：

```
-- file: ch14/Random.hs
data CountedRandom = CountedRandom {
      crGen :: StdGen
    , crCount :: Int
    }
```

(continues on next page)

(continued from previous page)

```
type CRState = State CountedRandom

getCountedRandom :: Random a => CRState a
getCountedRandom = do
  st <- get
  let (val, gen') = random (crGen st)
  put CountedRandom { crGen = gen', crCount = crCount st + 1 }
  return val
```

上面的函数每次被调用的时候都会处理状态的两个元素然后返回一个全新的状态；更常见的情况是我们只需要读写整个状态的某一部分；下面的函数可以获取当前生成过的随机数的数目：

```
-- file: ch14/Random.hs
getCount :: CRState Int
getCount = crCount `liftM` get
```

这个例子也说明了我们为什么要使用纪录语法定义 CountedRandom 状态；使用纪录函数提供的访问函数，把它与 get 函数结合起来可以很方便地读取状态的特定部分。

如果想要更新整个状态的某一部分，下面的代码可能不是很吸引人：

```
-- file: ch14/Random.hs
putCount :: Int -> CRState ()
putCount a = do
  st <- get
  put st { crCount = a }
```

这一段代码我们使用了纪录更新语法而不是用一个函数。表达式 st { crCount = a } 会创建一个和 st 几乎完全相等的值，只是使用给定的 a 作为 crCount 字段的值。由于这是个语法上的小技巧，因此它没有使用函数那么灵活。纪录语法可能并没有 Haskell 通常的语法那么优雅，但是至少它能完成我们的目的。

函数 modify 组合了 get 和 put，它接受一个状态转换函数，但是依然不太令人满意：还是需要使用纪录语法。

```
-- file: ch14/Random.hs
putCountModify :: Int -> CRState ()
putCountModify a = modify $ \st -> st { crCount = a }
```

14.14 Monad 和 Functors

Functor 和 Monad 之间有非常紧密的联系，这两个术语是从数学里面的范畴论引入的，但是又与数学定义不完全相同。

在范畴论里面，monad 通过 functor 构建出来。你可能希望在 Haskell 里面也是这样，也就是 Monad 这个类型类是 Functor 类型类的子类；但是在标准库的 Prelude 里面并不是这么定义的。这是个很不幸的疏忽。

但是，Haskell 库的作者们提供了一个变通方案：一旦他们写了一个 Monad 的实例，几乎总是也给 Functor 定义一个实例。所以对于任何 monad 你都可以使用 Functor 类型类的 fmap 函数。

如果把 fmap 函数的类型签名与我们已经见到过标准库里面 Monad 的一些函数做比较，大致就知道在 monad 里面 fmap 函数是干什么的了。

```
ghci> :type fmap
fmap :: (Functor f) => (a -> b) -> f a -> f b
ghci> :module +Control.Monad
ghci> :type liftM
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
```

可以看出，fmap 函数作用和 liftM 一样，它把一个纯函数 lift 到 monad 里面。

14.14.1 换个角度看 Monad

我们已经知道了 monad 和 functor 之间的联系，如果回头再看看 List 这个 monad，会发现一些有趣的东西；具体来说，是 list 的 (>>=) 定义。

```
       -- file: ch14/ListMonad.hs
       instance Monad [] where
           return x = [x]
           xs >>= f = concat (map f xs)

``f`` 的类型是 ``a -> [a]`` 我们调用 ``map f xs`` 的时候，我们会得到一个类型是 ``[[a]]`` 的值，
然后我们必须使用 ``concat`` 把它“扁平化”(flatten).
```

想一想如果 Monad 是 Functor 的子类的时候我们能做什么；由于 list 的 fmap 定义就是 map，在 (>>=) 定义里面我们可以使用 fmap 替换 map . 这个替换本身并没有什么特殊意义，我们再进一步探讨一下。

> concat 函数的类型是 [[a]] -> [a]：正如我们提到的，它把一个嵌套的列表压平。我们可以把 list 的这函数的类型签名从 list 推广到所有 monad，也就是一个“移除一层嵌套”的类型 m (m a) -> m a; 具有这种类型前面的函数通常叫做 join .

如果已经有了 join 和 fmap 的定义，我们就不需要为每一个 monad 定义一个 (>>=) 函数了，因为它完全可以由 join 和 fmap 定义出来。下面是 Monad 类型类另外一种定义方式。

```
-- file: ch14/AltMonad.hs
import Prelude hiding ((>>=), return)

class Functor m => AltMonad m where
    join :: m (m a) -> m a
```

(continues on next page)

(continued from previous page)

```
    return :: a -> m a

(>>=) :: AltMonad m => m a -> (a -> m b) -> m b
xs >>= f = join (fmap f xs)
```

不能说哪一种定义比另外一种更好，因为有了 `join` 我们可以定义 `(>>=)`，反之亦然。但是这两个不同的角度给了我们对 Monad 全新的认识。

移除一层 monadic 包装实际上是非常有用的，在 `Control.Monad` 里面由一个标准的 `join` 定义。

```
-- file: ch14/MonadJoin.hs
join :: Monad m => m (m a) -> m a
join x = x >>= id
```

下面是一些使用 `join` 的例子。

```
ghci> join (Just (Just 1))
Just 1
ghci> join Nothing
Nothing
ghci> join [[1],[2,3]]
[1,2,3]
```

14.15 单子律以及代码风格

在*更多关于 Functor 的思考* 里面我们介绍了 functors 必须遵从的两条规则：

```
-- file: ch14/MonadLaws.hs
fmap id        ==   id
fmap (f . g)   ==   fmap f . fmap g
```

monads 也有它们必须遵从的规则。下面的三条规则被称为单子律。Haskell 并不会强制检查这些规则：完全由 monad 的作者保证。

单子律就是简单而正式地表达“某个单子不会表现得让人惊讶”的意思。原则上讲，我们可以完全不管这些规则定义自己的 monad，但是如果我们这么干会为人所不齿的；因为单子律有一些我们可能忽视的宝藏。

Note: 下面的每一条规则，== 左边的表达式等价于右边的表达式。

第一条规则说的是 `return` 是 `(>>=)` 的 *Left identity*.

```
-- file: ch14/MonadLaws.hs
return x >>= f            ===    f x
```

另外一种理解这条规则的方式是：如果我们仅仅是把一个纯值包装到 monad 里面然后使用（>>=）调用的话，我们就没有必要使用 return 了；使用 monad 的新手通常所犯的错误就是使用 return 把一个纯值包装为 monadic 的，然后接下来由使用（>>=）把这个值取出来。下面是使用 do 表示法表达这个规律的等价形式：

```
-- file: ch14/MonadLaws.hs
do y <- return x
   f y                    ===    f x
```

这条规则对于我们的代码风格有着实际上的指导意义：我们不想写一些不必要的代码；这条规则保证了简短的写法和冗余的写法是等价的。

单子律的第二条说的是 return 是（>>=）的 *Right identity*.

```
-- file: ch14/MonadLaws.hs
m >>= return              ===    m
```

如果你以前使用命令式编程语言，那么这一条规则对风格也有好处：如果在一系列的 action 块里面，如果最后一句就是需要返回的正确结果，那么就不需要使用 return 了；看看使用 do 表示法如何表达这条规则：

```
-- file: ch14/MonadLaws.hs
do y <- m
   return y               ===    m
```

和第一条规则一样，这条规律也能帮助我们简化代码。

单子律最后一条和结合性有关。

```
-- file: ch14/MonadLaws.hs
m >>= (\x -> f x >>= g)    ===    (m >>= f) >>= g
```

这条规则有点难理解，我们先看看等式两边括号里面的内容；等式左边可以重新表示成这样：

```
-- file: ch14/MonadLaws.hs
m >>= s
  where s x = f x >>= g
```

等式右边也做类似的处理：

```
-- file: ch14/MonadLaws.hs
t >>= g
  where t = m >>= f
```

现在我们可以把上述规律表达成如下等价形式：

```
-- file: ch14/MonadLaws.hs
m >>= s                  ===   t >>= g
```

这条规则的意思是，如果我们把一个大的 action 分解成一系列的子 action，只要我们保证子 action 的顺序，把哪些子 action 提取出来组合成一个新 action 对求值结果是没有影响的。如果我们由三个 action 串联在一起，我们可以把前两个 action 替换为它们的组合然后串联第三个，也可以用第一个 action 串联后面两个 cation 的组合。

这条较为复杂的规律对我们的代码风格也有一些意义。在软件重构里面，有一个专业术语叫做“提取方法”，它说的就是在一大段代码里面提取出一些代码片段然后组合成一个新的函数，在原始代码里面调用新的函数来取代提取出来的内容；第三条单子律保证了这种做法在 Haskell 的 monadic 代码里面也可以使用这种技术。

三条单子律都能帮助我们写出更好的 monadic 代码；前两条规则指导我们如何避免使用不必要的 `return`，第三条规则让我们能安全地把一个复杂的 action 冲构成一系列小的 action。我们现在可以不管这些细节，通过直觉我们知道在一个实现良好的 monad 里面，这些规则是不会被违背的。

顺便说一下，Haskell 编译器并不并不能保证一个 monad 是否遵守单子律。monad 的实现者必须确保自己的代码满足（最好是证明）单子律。

CHAPTER

FIFTEEN

第 15 章：使用 MONAD 编程

15.1 高尔夫训练：关联列表

Web 客户端和服务器通常通过简单的文本键值对列表来传输消息，例如：

```
name=Attila+%42The+Hun%42&occupation=Khan
```

这种编码方式被称作 `application/x-www-form-urlencoded`，这种方式非常容易理解：每个键值对通过 `&` 划分。在一个键值对中，键由一系列 URL 编码字符构成，键后紧跟着 = 和值（如果存在的话）。

很明显我们可以用一个 `String` 来表示键，但 HTTP 没有明确指出一个键是否必须有对应的值。我们可以将值用 `Maybe String` 表示以捕获歧义，当我们使用 `Nothing` 时，值不存在，当我们用 `Just` 包装一个 `String` 时，值存在。使用 `Maybe` 允许我们区分“值不存在”和“空值”。[译注：`application/x-www-form-urlencoded` 实际是在 HTML 的 Forms 部分定义的，而非原著中的 HTML]

Haskell 程序员使用关联列表 表示类型 `[(a, b)]`，你可以把列表中的每个元素理解为一个键和值的关联。关联列表的名称起源于 Lisp 社区，通常缩写为列表，因此我们可以将上述字符串表示为以下的 Haskell 值。

```
-- file: ch15/MovieReview.hs
[("name",       Just "Attila \"The Hun\""),
 ("occupation", Just "Khan")]
```

在 parsing-an-url-encoded-query-string 中，我们将解析一个 `application/x-www-form-urlencoded` 编码的字符串，并使用形如 `[(String, Maybe String)]` 的关联列表表示结果。假设我们想使用这些列表中的一个来填充一个数据结构。

```
-- file: ch15/MovieReview.hs
data MovieReview = MovieReview {
          revTitle :: String
    , revUser :: String
, revReview :: String
        }
```

我们首先用一个简单的函数说明：

```
-- file: ch15/MovieReview.hs
simpleReview :: [(String, Maybe String)] -> Maybe MovieReview
simpleReview alist =
  case lookup "title" alist of
        Just (Just title@(_:_)) ->
          case lookup "user" alist of
                Just (Just user@(_:_)) ->
                  case lookup "review" alist of
                        Just (Just review@(_:_)) ->
                                Just (MovieReview title user review)
                        _ -> Nothing -- no review
                _ -> Nothing -- no user
        _ -> Nothing -- no title
```

当且仅当作为参数的关联列表包含了所有必须的键值对，并且这些键值对中的值不为空时，函数 `simpleReview` 将返回一个 `MovieReview` 。然而，它的优点仅仅是能够验证输入是否合法，实际上它采用了应当尽量避免的“锯齿型（staircasing）”代码结构，并且它过于了解关联列表的表示细节。

我们已经对 `Maybe` monad 非常熟悉了，上面的代码可以整理一下，让它避免“锯齿化”结构。

```
-- file: ch15/MovieReview.hs
maybeReview alist = do
        title <- lookup1 "title" alist
        user <- lookup1 "user" alist
        review <- lookup1 "review" alist
        return (MovieReview title user review)

lookup1 key alist = case lookup key alist of
                                        Just (Just s@(_:_)) -> Just s
                                        _ -> Nothing
```

代码看起来整洁了许多，但其中仍存在重复的工作。我们可以利用 `MovieReview` 构造器是普通纯函数的性质，将其提升为 monad，就像我们在同时使用 *puer* 和 *monadic* 代码 讨论过的那样。

```
-- file: ch15/MovieReview.hs
liftedReview alist =
        liftM3 MovieReview (lookup1 "title" alist)
                                        (lookup1 "user" alist)
                                        (lookup1 "review" alist)
```

在上面这段代码中依旧存在很多重复的工作，但它们已经显著减少了，并且我们很难去除剩下的部分。

15.2 广义的提升

虽然使用 liftM3 让我们的代码更加整洁，但我们不能用一堆 liftM 去解决更广泛的问题，因为标准库只定义到了 liftM5。事实上我们可以根据我们的需要写出任意数字的 liftM，但那将是非常繁重的工作。

假设我们有一个构造器或者纯函数，并且接受 10 个参数，这时候再坚持用标准库，你恐怕就觉得我们没有那么好运了。

当然，标准库里面还有其他工具可用，在 Control.Monad 中，有一个函数 ap，它的类型签名（type signature）非常有趣。

```
ghci> :m +Control.Monad
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

你可能会觉得奇怪，谁会把一个接受单一参数的纯函数放到 monad 中，这么做的原因又是什么？回想一下，其实所有的 Haskell 函数本质上都是接受单一参数，MovieReview 的构造器也是这样。

```
ghci> :type MovieReview
MovieReview :: String -> String -> String -> MovieReview
```

我们可以将类型签名写成 String -> (String -> String -> MovieReview)。假如我们使用 liftM 将 MovieReview 提升为 Maybe monad，我们将得到一个类型为 Maybe (String -> (String -> (String -> MovieReview)))``的值。这个类型恰好是 ``ap 接受的参数的类型，并且 ap 的返回类型将是 Maybe (String -> (String -> MovieReview))。我们可以将 ap 返回的值继续传入 ap，直到我们结束这个定义。

```
-- file: ch15/MovieReview.hs
apReview alist =
        MovieReview `liftM` lookup1 "title" alist
                                    `ap` lookup1 "user" alist
                                    `ap` lookup1 "review" alist
```

Warning: [译注：原著这里存在错误，上面的译文直接翻译了原著，在此做出修正。使用 liftM 将 MovieReview 提升为 Maybe monad 后，得到的类型不是 Maybe (String -> (String -> (String -> MovieReview)))，而是 Maybe String -> Maybe (String -> (String -> MovieReview))。下面给出在不断应用 ap 时，类型系统的显示变化过程。]

```
-- note by translator
        MovieReview :: String -> ( String -> String -> MovieRevie )
        MovieReview `liftM` :: Maybe String -> Maybe ( String -> String ->␣
↪MovieRevie )
        MovieReview `liftM` lookup1 "title" alist :: Maybe ( String -> String ->␣
↪MovieRevie )
        MovieReview `liftM` lookup1 "title" alist `ap` :: Myabe String -> Maybe (␣
↪String -> MovieRevie )
        MovieReview `liftM` lookup1 "title" alist `ap` lookup1 "user" alist ::␣
↪Maybe ( String -> MovieRevie )
```

我们可以通过不断应用 ap 来替代 liftM 的一系列函数。

这样理解 ap 可能会对你有所帮助：ap 的 monadic 等价于我们熟悉的 ($) 运算符，你可以想象一下把 ap 读成 apply。通过观察这二者的类型签名，我们可以清晰地看到这一点。

```
ghci> :type ($)
($) :: (a -> b) -> a -> b
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

事实上，ap 通常被定义为 liftM2 id 或者 liftM2 ($)。[译注：如果你使用 :t 来观察这两种书写形式得到的类型签名，你会发现它们在类型细节上有所差异，这是由 id 和 ($) 本身类型签名的不同导致的，id``的签名是 ``a -> a，而 ($) 是 (a -> b) -> (a -> b)，当然这对于广义的类型是等同的。]

15.3 寻找替代方案

下面是通讯录中一项的简单表示。

```
-- file: ch15/VCard.hs
data Context = Home | Mobile | Business
                           deriving (Eq, Show)

type Phone = String

albulena = [(Home, "+355-652-55512")]

nils = [(Mobile, "+47-922-55-512"), (Business, "+47-922-12-121"),
                (Home, "+47-925-55-121"), (Business, "+47-922-25-551")]

twalumba = [(Business, "+260-02-55-5121")]
```

假设我们想给某个人打一个私人电话，我们必然会选择他的家庭号码（假如他有的话），而不是他的工作号码。

```
-- file: ch15/VCard.hs
onePersonalPhone :: [(Context, Phone)] -> Maybe Phone
onePersonalPhone ps = case lookup Home ps of
                                              Nothing -> lookup Mobile ps
                                              Just n -> Just n
```

在上面的代码中，我们使用 Maybe 作为生成结果的类型，这样无法处理某个人有多个符合要求的号码的情况。因此，我们将返回类型转换为一个列表。

```
-- file: ch15/VCard.hs
allBusinessPhones :: [(Context, Phone)] -> [Phone]
allBusinessPhones ps = map snd numbers
        where numbers = case filter (contextIs Business) ps of
                                          [] -> filter (contextIs Mobile) ps
                                          ns -> ns

contextIs a (b, _) = a == b
```

注意，这两个函数中 case 表达式的结构非常相似：其中一个标签处理查找结果为空的情况，剩下的处理结果非空的情况。

```
ghci> onePersonalPhone twalumba
Nothing
ghci> onePersonalPhone albulena
Just "+355-652-55512"
ghci> allBusinessPhones nils
["+47-922-12-121","+47-922-25-551"]
```

[译注：这里的代码通过需要 :l 导入 VCard.hs]

Haskell 的 Control.Monad 模块定义了一种类型类 MonadPlus ，这使我们可以将 case 表达式中的普通模式抽象出来。

```
-- file: ch15/VCard.hs
class Monad m => MonadPlus m where
   mzero :: m a
   mplus :: m a -> m a -> m a
```

值 mzero 表示了一个空结果，mplus 将两个结果合并为一个。下面是 mzero 和 mplus 针对 Maybe 和列表的标准定义。[译注：在约翰 · 休斯 1998 年发表的《Generalising Monads to Arrows》中，他提出 mzero 可理解为对失败情况的一种概括，而 mplus 则是对选择情况的概括，例如如果第一种情况失败，则尝试第二种。]

```
-- file: ch15/VCard.hs
instance MonadPlus [] where
   mzero = []
   mplus = (++)

instance MonadPlus Maybe where
   mzero = Nothing

   Nothing `mplus` ys  = ys
```

(continues on next page)

(continued from previous page)

```
  xs      `mplus` _ = xs
```

我们现在可以使用 mplus 替换掉整个 case 表达式。为了照顾情况的多样性，我们下面来获取通讯录中某人的一个工作号码和他所有的私人号码。

```
-- file: ch15/VCard.hs
oneBusinessPhone :: [(Context, Phone)] -> Maybe Phone
oneBusinessPhone ps = lookup Business ps `mplus` lookup Mobile ps

allPersonalPhones :: [(Context, Phone)] -> [Phone]
allPersonalPhones ps = map snd $ filter (contextIs Home) ps `mplus`
                                                          filter (contextIs␣
↪Mobile) ps
```

[译注：在前面的例子中，我们将 mplus 作为 case 模式的一种抽象表达来介绍，但是对于 list monad，它会产生和前面例子不同的结果。考虑前面的例子 allBusinessPhones，我们试图获取一个人的全部工作号码，当且仅当他没有工作号码时，结果中才包含私人号码。而 mplus 只是将全部工作号码和私人号码连接在一起，这和我们想要的结果有出入。]

我们已经知道 lookup 会返回一个 Maybe 类型的值，而 filter 将返回一个列表，所以对于这些函数，应当使用什么版本的 mplus 是非常显然的。

更有趣的是我们现在可以使用 mzero 和 mplus 来编写对任意 MonadPlus 实例均有效的函数。举例而言，下面是一个标准的 lookup 函数，它将返回一个 Maybe 类型的值。

```
-- file: ch15/VCard.hs
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
lookup _ []                      = Nothing
lookup k ((x,y):xys) | x == k    = Just y
                                 | otherwise = lookup k xys
```

通过下面的代码，我们可以很容易的将结果类型推广到 MonadPlus 的任意实例。

```
-- file: ch15/VCard.hs
lookupM :: (MonadPlus m, Eq a) => a -> [(a, b)] -> m b
lookupM _ []     = mzero
lookupM k ((x,y):xys)
        | x == k    = return y `mplus` lookupM k xys
        | otherwise = lookupM k xys
```

假如我们得到的结果是 Maybe 类型，那么通过这种方式我们将得到一个结果或“没有结果”；假如我们得到的结果是一个列表，那么我们将获得所有的结果；其它情况下，我们将获得一些适用于其它 MonadPlus 实例的结果。

对于一些类似我们上面展示的小函数，使用 mplus 没什么明显的优点。mplus 的优点体现在更复杂的代码

和那些独立于 monad 执行过程的代码中。即使你没有在自己的代码中碰到需要使用 MonadPlus 的情况，你也很可能在别人的项目中遇到它。

15.3.1 mplus 不意味着相加

函数 mplus 的名字中包含了“plus”，但这并不代表着我们一定是要将两个值相加。根据我们处理的 monad 的不同，有时 mplus 会实现看起来类似相加的操作。例如，列表 monad 中 mplus 等同于 (++) 运算符。

```
ghci> [1,2,3] `mplus` [4,5,6]
[1,2,3,4,5,6]
```

但是，假如我们切换到另一个 monad，mplus 和加法操作将不存在明显的相似性。

```
ghci> Just 1 `mplus` Just 2
Just 1
```

15.3.2 使用 MonadPlus 的规则

除了通常情况下 monad 的规则外，MonadPlus 类型类的实例必须遵循一些其他简单的规则。

如果一个捆绑表达式左侧出现了 mzero ，那么这个实例必须短路（short circuit）。换句话说，表达式 mzero >>= f 必须和单独的 mzero 效果相同。[译注：“短路”也用来描述严格求值语言中布尔运算符的“短路”特性，例如 B != null && B.value != "" 可以避免在 B == null 时考量 B.value]

```
-- file: ch15/MonadPlus.hs
        mzero >>= f == mzero
```

如果 mzero 出现在了一个序列表达式的右侧，则这个实例必须短路。[译注：此处存在争议，例如 Maybe monad 的一个例子 (undefined >> Nothing) = undefined /= Nothing 不满足这一条件。一种观点认为，短路特性意味着如果表达式中某个操作数的结果为某事，则不评估另一个操作数，也就是说必须首先评估一个操作数。所以，在“从左向右”和“从右向左”的短路之间，只能存在一种。]

```
-- file: ch15/MonadPlus.hs
        v >> mzero == mzero
```

15.3.3 通过 MonadPlus 安全地失败

当我们在*Monad 类型类* 中介绍 fail 函数时，我们对它的使用提出了警告：在许多 monad 中，它可能被实现为一个对错误的调用，这会导致令人不愉快的后果。

MonadPlus 类型类为我们提供了一种更温和的方法来使一个计算失败，这使我们不必面临使用 fail 和 error 带来的危险。上面介绍的规则允许我们在代码中需要的任何地方引入一个 mzero ，这样计算将在该

处短路。

在 Control.Monad 模块中，标准函数 guard 将这个想法封装成了一种方便的形式。

```
-- file: ch15/MonadPlus.hs
guard          :: (MonadPlus m) => Bool -> m ()
guard True     =  return ()
guard False    =  mzero
```

作为一个简单的例子，这里有一个函数，它接受一个数 x 作为参数，并计算 x 对于另一个数 n 的取模结果。假如结果是 0 则返回 x ，否则返回当前 monad 对应的 mzero 。

```
-- file: ch15/MonadPlus.hs
x `zeroMod` n = guard ((x `mod` n) == 0) >> return x
```

15.4 隐藏管道

在使用 *State monad* 生成随机数 中，我们展示了使用 State monad 生成随机数的简单方法。

我们编写的代码的一个缺点是它泄露了细节：使用者知道代码运行在 State monad 中。这意味着他们可以像我们这些作者一样检测并修改随机数生成器的状态。

人的本性决定了，一旦我们将工作内部细节暴露出来，就会有人试图对其做手脚。对于一个足够小的程序，这也许没什么问题，但在更大的软件项目中，如果库的某个使用者使用了其他使用者都没有预料到的方式修改库，这一举动可能导致的错误将非常严重。因为问题出现在库中，而我们通常不会怀疑库有问题，所以这些错误很难被发现，直到我们排除了所有其他可能。

更糟糕的是，一旦程序的实现细节暴露，一些人将绕过我们提供的 API 并直接采用内部实现方式。当我们需要修复某个错误或者增强某个功能时，我们等于为自己设置了一道屏障。我们要么修改内部、破坏依赖它们的代码，要么坚持现有的内部结构并寻找其他方式来做出需要的改变。

我们该如何修改随机数 monad 来隐藏我们使用了 State monad 的事实？我们需要使用某种方式来阻止用户调用 get 或 put 。要实现这一点并不难，并且在具体实现中我们将会介绍一些在日常 Haskell 编程中经常使用的技巧。

为了扩大应用的范围，我们不用随机数做例子，而是实现了一个可以提供任意类型不重复值的 monad，这个 monad 叫做 Supply 。我们将为执行函数 runSupply 提供一个值的列表，并确保列表中每个值是独一无二的。

```
-- file: ch15/Supply.hs
runSupply :: Supply s a -> [s] -> (a, [s])
```

这个 monad 并不关心这些值是什么，它们可能是随机数，或临时文件的名称，或者是 HTTP cookie 的标识符。

在这个 monad 中，每当用户要求获取一个值时，next 就会从列表中取出下一个值并将其交给用户。每个值都被 Maybe 构造器包装以防止这个列表的长度不满足需求。

```
-- file: ch15/Supply.hs
next :: Supply s (Maybe s)
```

为了隐藏我们的管道，在模块声明中我们只导出了类型构造函数，执行函数和 next 动作。

```
-- file: ch15/Supply.hs
module Supply
        (
          Supply
        , next
        , runSupply
        ) where
```

因为导入库的模块不能看到 monad 的内部，所以它不能修改我们的库。

我们的管道非常简单：使用一个 newtype 声明来包装现有的 State monad。

```
-- file: ch15/Supply.hs
import Control.Monad.State

newtype Supply s a = S (State [s] a)
```

参数 s 是我们提供的独特值的类型，a 是我们必须提供的常见类型参数，以使我们的类型成为 monad。[译注：这里类型 a 是自由的，即 a 可以是任何东西，以允许 monadic 函数返回任何可能需要的类型。例如 hGetLine :: Handle -> IO String 这样一个 monadic 函数，给定一个文件句柄，将从它读取一行并返回这一行的内容。这里，String 是 IO Monad 要返回的类型 a，程序员可以将 hGetLine 看作从句柄读取 String 的函数。]

我们通过在 Supply 类型上应用 newtype 以及定义模块头来阻止用户使用 State monad 的 get 和 set 动作。因为我们的模块并不导出 s 的构造器，所以用户没有程序化的方式来查看或访问包装在 State monad 中的内容。

现在我们有了一个类型 Supply，接下来需要定义一个 Monad 类型类的实例。我们可以遵循通常的方式如 (>>=) 和 return，但这将变成纯样板代码。我们现在所做的是通过 s 值构造器将 State monad 版本的 (>>=) 及 return 包装和展开。代码看起来应该是这个样子：

```
-- file: ch15/AltSupply.hs
unwrapS :: Supply s a -> State [s] a
unwrapS (S s) = s

instance Monad (Supply s) where
        s >>= m = S (unwrapS s >>= unwrapS . m)
```

(continues on next page)

(continued from previous page)

```
        return = S . return
```

Haskell 程序员不喜欢样板，GHC 有一个可爱的语言拓展功能消除了这一工作。我们将以下指令添加到源文件顶部（模块头之前）来使用这一功能。

```
-- file: ch15/Supply.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

通常我们只能自动导出一些标准类型类的实例如 Show 和 Eq 。顾名思义，GeneralizedNewtypeDeriving 拓展了我们派生类型类实例的能力，并且它特定于 newtype 的声明。通过下面的方式，如果我们包装的类型是任意一个类型类的实例，这个拓展可以自动让我们的新类型成为该类型类的实例。

```
-- file: ch15/Supply.hs
        deriving (Monad)
```

[译注：在 GHC 7.10 中，Monad` 是 ``Applicative 和 Functor 的子类，因此上面的 deriving 需要改为 deriving(Functor, Applicative, Monad) 。]

这需要底层类型实现 (>>=) 和 return ，通过 s 值构造器添加必要的包装和展开方法，并使用这些函数的新版本为我们导出一个 Monad 实例。[译注：这里的底层类型指的是 State 。]

我们在这里获得到的远比这个例子来的多。我们可以使用 newtype 来包装任何底层类型；我们选择性地只暴露符合我们想法的类型类实例；而且我们几乎没有花费更多的工作来创建这些更恰当、专业的类型。

现在我们已经看到了 GeneralizedNewtypeDeriving 技术，剩下的工作就是提供 next 和 runSupply 的定义。

```
-- file: ch15/Supply.hs
next = S $ do st <- get
                          case st of
                                [] -> return Nothing
                                (x:xs) -> do put xs
                                                                  return (Just x)


runSupply (S m) xs = runState m xs
```

我们可以将模块导入 **ghci** ，并用几种简单的方式尝试：

```
ghci> :load Supply
[1 of 1] Compiling Supply           ( Supply.hs, interpreted )
Ok, modules loaded: Supply.
ghci> runSupply next [1,2,3]
Loading package mtl-1.1.0.0 ... linking ... done.
(Just 1,[2,3])
```

(continues on next page)

(continued from previous page)

```
ghci> runSupply (liftM2 (,) next next) [1,2,3]
((Just 1,Just 2),[3])
ghci> runSupply (liftM2 (,) next next) [1]
((Just 1,Nothing),[])
```

我们也可以验证 State monad 是否以某种方式泄露。

```
ghci> :browse Supply
data Supply s a
next :: Supply s (Maybe s)
runSupply :: Supply s a -> [s] -> (a, [s])
ghci> :info Supply
data Supply s a          -- Defined at Supply.hs:17:8-13
instance Monad (Supply s) -- Defined at Supply.hs:17:8-13
```

15.4.1 提供随机数

如果我们想使用 Supply monad 作为随机数的源，那么有一些小困难需要克服。理想情况下，我们希望能为它提供一个无限的随机数流。我们可以在 IO monad 中获得一个 StdGen ，但完成后必须“放回”一个不同的 StdGen 。假如我们不这么做，下一段代码得到的 StdGen 将获得与之前相同的状态。这意味着它将产生与此前相同的随机数，这样的结果可能是灾难性的。

目前为止我们所看到的 System.Random 模块很难满足这些要求。我们可以使用 getStdRandom ，它的类型确保了我们可以同时得到和放回一个 StdGen 。

我们可以使用 random 在获取一个随机数的同时得到一个新的 StdGen 。我们可以用 randoms 获取一个无限的随机数列表。但我们如何同时得到一个无限的随机数列表和一个新的 StdGen?

答案在于 RandomGen 类型类的拆分函数。它接受一个随机数生成器，并将其转换为两个生成器。能够分裂这样的随机生成器是一件很不寻常的事，它在纯函数的设定中显然非常有用，但对于非纯函数语言基本不需要。[译注：stdSplit 的统计基础较差，如果随机数的质量很重要，应当尽量避免不必要的分割。]

通过使用 split 函数，我们可以用一个 StdGen 来生成一个无限长的随机数列表并将其交付 runSupply ，同时将另一个 StdGen 返还给 IO monad。

```
-- file: ch15/RandomSupply.hs
import Supply
import System.Random hiding (next)

randomsIO :: Random a => IO [a]
randomsIO =
        getStdRandom $ \g ->
```

(continues on next page)

(continued from previous page)

```
                let (a, b) = split g
                in (randoms a, b)
```

如果我们正确的书写了这个函数，我们的例子应该在每次调用时打印一个不同的随机数。

```
ghci> :load RandomSupply
[1 of 2] Compiling Supply             ( Supply.hs, interpreted )
[2 of 2] Compiling RandomSupply       ( RandomSupply.hs, interpreted )
Ok, modules loaded: RandomSupply, Supply.
ghci> (fst . runSupply next) `fmap` randomsIO

<interactive>:1:17:
        Ambiguous occurrence `next'
        It could refer to either `Supply.next', imported from Supply at RandomSupply.
↪hs:4:0-12
                                                                                ⌴
↪     (defined at Supply.hs:32:0)
                                                    or `System.Random.next', imported⌴
↪from System.Random
ghci> (fst . runSupply next) `fmap` randomsIO

<interactive>:1:17:
        Ambiguous occurrence `next'
        It could refer to either `Supply.next', imported from Supply at RandomSupply.
↪hs:4:0-12
                                                                                ⌴
↪     (defined at Supply.hs:32:0)
                                                    or `System.Random.next', imported⌴
↪from System.Random
```

Warning: [译注：此处保留了原著中的错误，原著执行的代码中可能丢失了 `hiding (next)`，因此产生歧义。下面给出正确情况下的某次执行结果。]

```
ghci> :load RandomSupply
[1 of 2] Compiling Supply             ( Supply.hs, interpreted )
[2 of 2] Compiling Main               ( RandomSupply.hs, interpreted )
Ok, modules loaded: Supply, Main.
ghci> (fst . runSupply next) `fmap` randomsIO
Just (-54705384517081531)
ghci> (fst . runSupply next) `fmap` randomsIO
Just (-2652939136952789000)
ghci> (fst . runSupply next) `fmap` randomsIO
Just (-5089130856647223466)
```

回想一下，我们的 `runSupply` 函数同时返回执行 monadic 操作的结果和列表的剩余部分。因为我们传递了一个无限的随机数列表，所以这里用 `fst` 来组合，以保证当 **ghci** 尝试打印结果时不会被随机数淹没。

15.4.2 另一轮高尔夫训练

这种将函数应用在一对元素的其中一个元素上，并且与另一个未被修改的原始元素构成新对的模式在 Haskell 代码中已经非常普遍，它已经成为了一种标准代码。

在 `Control.Arrow` 模块中有两个函数 `first` 和 `second`，它们执行这个操作。

```
ghci> :m +Control.Arrow
ghci> first (+3) (1,2)
(4,2)
ghci> second odd ('a',1)
('a',True)
```

（事实上我们已经在*JSON 类型类，不带有重叠实例* 中遇到过 `second` 了。）我们可以使用 `first` 来产生我们自己 `randomsIO` 的定义，将其转化为单线形式。

```
-- file: ch15/RandomGolf.hs
import Control.Arrow (first)

randomsIO_golfed :: Random a => IO [a]
randomsIO_golfed = getStdRandom (first randoms . split)
```

15.5 将接口与实现分离

在前面的章节中，我们看到了如何向用户隐藏我们在内部使用 `State` monad 来保持 `Supply` monad 状态的事实。

使代码更加模块化的另一个重要方式是将接口（代码可以做什么）从实现（代码具体如何做）中分离出来。

众所周知，标准的随机数生成器 `System.Random` 速度很慢。如果使用我们的 `randomsIO` 函数为它提供随机数，那么我们的 `next` 动作执行效果将不会很好。

一个简单有效的解决办法是为 `Supply` 提供一个更好的随机数源，但现在让我们把这个方法先放到一边，转而考虑一种在许多设定中都有效的替代方法。我们将使用一个类型类，对 monad 的行为和实现进行分离。

```haskell
-- file: ch15/SupplyClass.hs
class (Monad m) => MonadSupply s m | m -> s where
        next :: m (Maybe s)
```

这个类型类定义了任何 Supply monad 都必须实现的接口。因为这个类型类使用了几个我们尚不熟悉的 Haskell 语言扩展，所以我们需要对这个类型类进行详细的分析，它涉及的各个扩展将在接下来的小节中介绍。

15.5.1 多参数类型类

我们应该如何读取类型类中的代码片段 `MonadSupply s m`？如果我们添加括号，则等价表达式是 `(MonadSupply s) m`，它看起来更清晰一些。换句话说，给定一些 Monad 类型变量 `m`，我们可以让它成为类型类 `MonadSupply s` 的一个实例。与常规类型类不同，这里的类型类有一个参数。[译注：此链接 可能会帮助你更好地理解功能依赖及多参数类型类。实际上，上面给出的例子中，`s` 和 `m` 仅仅是两个参数，它们的位置可以互换。原作者通过加括号的方式，将其理解为一个参数和另一个有参数的类型类，也许在数学上正确，但译者认为此处可能导致读者的困惑。]

这个语言扩展被称作 `MultiParamTypeClasses`，因为它允许类型类有多个参数。参数 `s` 的作用与 `Supply` 类型的参数相同：它代表了 `next` 动作发出的值。

注意，我们不需要在 `MonadSupply s` 的定义中提到 `(>>=)` 或 `return`，因为类型类的上下文（超类）要求 `MonadSupply s` 必须已经是一个 `Monad`。

15.5.2 功能依赖

现在让我们回头看之前被忽略的代码段，`| m -> s` 是一个功能依赖，通常被称作 `fundep`。我们可以将竖线 `|` 读作“这样”，将箭头 `->` 读作“唯一确定”。我们的功能依赖建立了 `m` 和 `s` 之间的关系。

功能依赖由 `FunctionalDependencies` 语言编译指令管理。

我们声明 `m` 和 `s` 之间关系的目的是帮助类型检查器。回想一下，Haskell 类型检查器本质上是一个定理证明器，并且它在操作上是保守的：它坚持这个证明过程必须终止。一个非终止的证明结果将导致编译器放弃或陷入无限循环。

通过功能依赖，我们告诉类型检查器，一旦它看到一些 monad `m` 在 `MonadSupply s` 的上下文中被使用，那么类型 `s` 就是唯一可以接受的类型。假如我们省略功能依赖，类型检查器就会放弃并返回一个错误消息。

我们很难描述 `m` 和 `s` 之间的关系究竟是什么，所以让我们来看一个该类型类具体的实例。

```haskell
-- file: ch15/SupplyClass.hs
import qualified Supply as S

instance MonadSupply s (S.Supply s) where
        next = S.next
```

这里，类型变量 m 由类型 S.Supply s 替换。因为功能依赖的存在，类型检查器知道当它看到类型 S.Supply s 时，这个类型可以被当作类型类 MonadSupply s 的实例来使用。

假如没有功能依赖，类型检查器不会发现 MonadSupply s 和 Supply s 二者类型参数之间的关系，因此它将终止编译并产生一个错误。它们的定义本身将会编译，而类型错误直到我们尝试使用它的时候才会产生。

下面我们用一个例子剥离最后一层抽象：考虑类型 S.Supply Int 。我们可以不使用功能依赖而将其声明为 MonadSupply s 的一个实例。但是假如我们试图使用这个实例编写代码，编译器将无法得知类型的 Int 参数需要和类型类 s 的参数相同，因此它将报告一个错误。

功能依赖可能难以理解，并且它们被证明在实践中通常很难使用。幸运的是，功能依赖一般在和我们例子类似的简单情况下使用，此时它们不会导致什么麻烦。

15.5.3 舍入模块

假如我们将类型类和实例保存在名为 SupplyClass.hs 的源文件中，那么就需要在文件中添加一个类似这样的模块头：

```
-- file: ch15/SupplyClass.hs
{-# LANGUAGE FlexibleInstances, FunctionalDependencies,
                          MultiParamTypeClasses #-}

module SupplyClass
        (
          MonadSupply(..)
        , S.Supply
        , S.runSupply
        ) where
```

这个 FlexibleInstances 扩展是必须的，否则编译器不会接受我们的实例声明。这个扩展放宽了在某些情况下书写实例的一般规则，但同时仍然让编译器的类型检查器保证它的推导会结束。我们这里需要 FlexibleInstances 的原因是我们使用了功能依赖，具体的细节超过了本书所讨论的范畴。

Note: 如何知道是否需要一个语言扩展

假如 GHC 因为需要启用某些语言扩展而不能编译一段代码，它会提醒我们哪些扩展需要使用。比如，假如它认为我们的代码需要 flexible 的实例支持，它将提醒我们使用 -XFlexibleInstances 选项编译。-x 选项和 LANGUAGE 伪指令具有相同的效果：它们都可以启用一个特定的扩展。

最后，请注意我们正在从此模块中重新导出 runSupply 和 Supply 的名称。从一个模块中导出名称是完全合法的，即使它在另一个模块中被定义。在我们的例子中，这意味着客户端代码只需要导入 SupplyClass 模块，而不需要导入 Supply 模块。这减少了用户需要记住的“移动部件”的数量。

15.5.4 对 monad 接口编程

下面是一个简单的函数，它从我们的 Supply monad 中获取两个值，将它们格式化为字符串，并返回它们。

```
-- file: ch15/Supply.hs
showTwo :: (Show s) => Supply s String
showTwo = do
  a <- next
  b <- next
  return (show "a: " ++ show a ++ ", b: " ++ show b)
```

这份代码通过它的结果类型绑定到我们的 Supply monad。通过修改函数类型，我们可以在维持函数主体不变的情况下，将此方法推广至所有实现了 MonadSupply 接口的 monad。

```
-- file: ch15/SupplyClass.hs
showTwo_class :: (Show s, Monad m, MonadSupply s m) => m String
showTwo_class = do
  a <- next
  b <- next
  return (show "a: " ++ show a ++ ", b: " ++ show b)
```

15.6 Reader monad

State monad 让我们通过代码传递了一些可变状态。有时，我们希望能够传递一些不可变的状态，比如程序的配置数据。在这种情况下我们仍可以使用 State monad ，但这种做法有时候可能会错误地修改了不可变的数据。

让我们暂时忘记 monad 。考虑一个能满足我们要求的函数，它应当具有什么行为？它应当接收一个类型为 e （根据环境而定）的值，该值携带了我们要传入的数据；它应当返回一个其它类型 a 的值作为结果。我们希望整个函数的类型签名为 e -> a 。

为了将这个类型转化为一个方便的 Monad 实例，我们用一个 newtype 包装它。

```
-- file: ch15/SupplyInstance.hs
newtype Reader e a = R { runReader :: e -> a }
```

很容易将它转化为 Monad 实例。

```
-- file: ch15/SupplyInstance.hs
instance Monad (Reader e) where
        return a = R $ \_ -> a
        m >>= k = R $ \r -> runReader (k (runReader m r)) r
```

Warning: [译注：在 **GHC** 7.10 中，你需要同时定义 `Functor` 和 `Applicative` 的实例，其中一种实现如下。]

```
instance Functor (Reader e) where
  fmap f m = R $ \r -> f (runReader m r)

instance Applicative (Reader e) where
  pure = R . const
  f <*> x = R $ \r -> runReader f r (runReader x r)
```

我们可以将类型 `e` 表示的值理解为计算某个表达式所处的环境。因为 `return` 动作应当在任何情况下都具有相同的效果，所以这段代码不必考虑 `return` 所处的环境。

为了让环境————也即是代码中的变量 `r` ————能够在当前和接下来的计算中使用，我们使用了比较复杂的 `(>>=)` 定义。

在 monad 中执行的一段代码可以通过 `ask` 来获取环境中包含的值。

```
-- file: ch15/SupplyInstance.hs
ask :: Reader e e
ask = R id
```

在给定的一连串动作中，每次调用 `ask` 都将返回相同的值，因为存储在环境中的值不会改变。我们可以在 **ghci** 中方便地测试代码。

```
ghci> runReader (ask >>= \x -> return (x * 3)) 2
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package random-1.0.0.0 ... linking ... done.
6
```

`Reader` monad 包含在标准 `mtl` 库中，它通常与 GHC 捆绑在一起。你可以在 `Control.Monad.Reader` 模块中找到它。你可能不明白设计这个 monad 的动机是什么，实际上它通常在复杂的代码中更有用。我们时常需要访问程序深处的一段配置信息，若将这些信息作为正常参数传递，我们需要对代码做一次极为痛苦的重构。如果将这些信息隐藏在 monad 的管道中，那么不关心配置信息的中间函数就不需要看到它们。

使用 `Reader` monad 最明显的动机将在 monad-transformers 中介绍，我们将讨论如何把几个 monad 合并为一个新的 monad 。那时候，我们将学会如何利用 `Reader` monad 去更好地控制状态，以便我们的代码可以通过 `State` monad 修改一些值，而其他值则保持不变。

15.7 返回自动导出

既然我们了解了 Reader monad，下面让我们用它来创建一个 MonadSupply 类型类的实例。为了保持例子的简洁性，这里我们将违反 MonadSupply 的精神：我们的 next 动作将始终返回相同的值。

将 Reader 类型变成一个 MonadSupply 类实例是一个糟糕的想法，因为任意 Reader 都可以表现为一个 MonadSupply 。所以这样的做法通常没有任何意义。

事实上，我们在 Reader 的基础上创建一个 newtype 。这个 newtype 隐藏了内部使用 Reader 的事实。我们必须让类型成为我们关心的类型类的一个实例。通过激活 GeneralizedNewtypeDeriving 扩展功能，GHC 会帮助我们完成大部分困难的工作。

```
-- file: ch15/SupplyInstance.hs
newtype MySupply e a = MySupply { runMySupply :: Reader e a }
        deriving (Monad)

instance MonadSupply e (MySupply e) where
        next = MySupply $ do
                          v <- ask
                          return (Just v)

        -- more concise:
        -- next = MySupply (Just `liftM` ask)
```

注意，类型必须是 MonadSupply e 的实例，而不是 MonadSupply 。如果我们省略了类型变量，编译器将会报错。[译注：在类型声明中指定类型依赖的方式是在类名和实例名中使用相同的参数名。]

要尝试我们的 MySupply 类型，首先创建一个能处理任何 MonadSupply 实例的简单函数。

```
-- file: ch15/SupplyInstance.hs
xy :: (Num s, MonadSupply s m) => m s
xy = do
  Just x <- next
  Just y <- next
  return (x * y)
```

正如我们期望的那样，如果我们将 Supply monad 和 randomsIO 函数与上面这个函数结合，那么每次都会得到一个不同的结果。

```
ghci> (fst . runSupply xy) `fmap` randomsIO
-1569706427086308182544847639284191757 8
ghci> (fst . runSupply xy) `fmap` randomsIO
171829834446168344942573980423601197 26
```

因为 MySupply monad 由两层 newtype 包装，为了使其更易使用，我们可以为它编写一个自定义的执行函

数:

```
-- file: ch15/SupplyInstance.hs
runMS :: MySupply i a -> i -> a
runMS = runReader . runMySupply
```

当我们通过这个执行函数来应用 xy 动作时，每次都会得到相同的结果。虽然代码没有改动，但因为我们在不同 MonadSupply 的实现下执行，所以它的行为改变了。

```
ghci> runMS xy 2
4
ghci> runMS xy 2
4
```

就像我们的 MonadSupply 类型类和 Supply monad，几乎所有常见的 Haskell monad 都建立在接口与实现分离的情况下。举个例子，我们介绍过两个“属于” State monad 的函数 get 和 put ，它们事实上是 MonadState 类型类的方法；State 类型只是这个类的一个实例。

类似的，标准的 Reader monad 是类型类 MonadReader 的实例，它指定了 ask 方法。

我们上面讨论的接口和实现分离不止对于架构清洁有帮助，它重要的实际应用在日后将更加清楚。当我们在 monad-transformers 中开始合并 monad 时，我们会通过使用 GeneralizedNewtypeDeriving 和类型类来节约大量的工作。

15.8 隐藏 IO monad

IO monad 的“祝福”和“诅咒”都来源于它的过分强大。假如我们相信小心地使用类型能帮助我们杜绝程序错误，那么 IO monad 将成为错误的一个重要来源。因为 IO monad 对我们做的事情不加以限制，这使我们容易遭受各种意外。

我们如何驯服它的力量？假如我们想保证一段代码可以读取和写入本地文件，但它不能访问网络。这时我们不能使用 IO monad，因为它不会限制我们。

15.8.1 使用 newtype

让我们创建一个新模块，它提供了一组读取和写入文件的功能。

```
-- file: ch15/HandleIO.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module HandleIO
        (
          HandleIO
```

(continues on next page)

(continued from previous page)

```
        , Handle
        , IOMode(..)
        , runHandleIO
        , openFile
        , hClose
        , hPutStrLn
        ) where

import System.IO (Handle, IOMode(..))
import qualified System.IO
```

我们要创建有限制 IO 的第一步是使用 newtype 对其进行包装。

```
-- file: ch15/HandleIO.hs
newtype HandleIO a = HandleIO { runHandleIO :: IO a }
        deriving (Monad)
```

我们使用已经熟悉的技巧，从模块中导出类型构造函数和 runHandleIO 执行函数，但不导出数据构造器。这可以阻止在 HandleIO monad 中执行的代码获取它所包装的 IO monad。

剩下的工作就是包装我们希望 monad 允许的每个动作。这可以通过用 HandleIO 数据构造函数包装每个 IO 实现。

```
-- file: ch15/HandleIO.hs
openFile :: FilePath -> IOMode -> HandleIO Handle
openFile path mode = HandleIO (System.IO.openFile path mode)

hClose :: Handle -> HandleIO ()
hClose = HandleIO . System.IO.hClose

hPutStrLn :: Handle -> String -> HandleIO ()
hPutStrLn h s = HandleIO (System.IO.hPutStrLn h s)
```

我们现在可以使用有限制的 HandleIO monad 来执行 I/O 操作。

```
-- file: ch15/HandleIO.hs
safeHello :: FilePath -> HandleIO ()
safeHello path = do
  h <- openFile path WriteMode
  hPutStrLn h "hello world"
  hClose h
```

要执行这个动作，我们使用 runHandleIO 。

```
ghci> :load HandleIO
[1 of 1] Compiling HandleIO         ( HandleIO.hs, interpreted )
Ok, modules loaded: HandleIO.
ghci> runHandleIO (safeHello "hello_world_101.txt")
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> :m +System.Directory
ghci> removeFile "hello_world_101.txt"
```

假如我们试图用一种不被允许的方式排列一个在 HandleIO monad 中执行的动作，类型系统会加以阻止。

```
ghci> runHandleIO (safeHello "goodbye" >> removeFile "goodbye")

<interactive>:1:36:
        Couldn't match expected type `HandleIO a'
                    against inferred type `IO ()'
        In the second argument of `(>>)', namely `removeFile "goodbye"'
        In the first argument of `runHandleIO', namely
                `(safeHello "goodbye" >> removeFile "goodbye")'
        In the expression:
                runHandleIO (safeHello "goodbye" >> removeFile "goodbye")
```

15.8.2 针对意外使用情况设计

HandleIO monad 有一个重要的小问题：我们可能偶尔需要一个“逃生舱门”，而 HandleIO monad 没有考虑到这种可能性。如果定义一个这样的 monad，我们很有可能会偶尔执行 monad 设计所不允许的 I/O 动作。

设计这个 monad 的目的是让我们更容易在普通情况下编写无错代码，而不是为了杜绝特殊情况发生。让我们给自己留一条出路。

Conotrol.Monad.Trans 模块定义了一个“标准逃生舱门”：MonadIO 类型类。函数 liftIO 可以将一个 IO 动作嵌入另一个 monad 中。

```
ghci> :m +Control.Monad.Trans
ghci> :info MonadIO
class (Monad m) => MonadIO m where liftIO :: IO a -> m a
        -- Defined in Control.Monad.Trans
instance MonadIO IO -- Defined in Control.Monad.Trans
```

要实现这个类型类非常简单，只需要用我们自己的数据构造器将 IO 包装起来就可以了：

```haskell
-- file: ch15/HandleIO.hs
import Control.Monad.Trans (MonadIO(..))

instance MonadIO HandleIO where
        liftIO = HandleIO
```

通过审慎地使用 `liftIO`，我们可以逃脱束缚，并在必要的时候调用 IO 操作。

```haskell
-- file: ch15/HandleIO.hs
tidyHello :: FilePath -> HandleIO ()
tidyHello path = do
  safeHello path
  liftIO (removeFile path)
```

15.8.3 使用类型类

将 `IO` 隐藏到另一个 monad 的缺陷是我们仍然绑定到了一个具体的实现。假如要将 `HandleIO` 和一些其它的 monad 交换，就必须改变每个使用 `HandleIO` 的动作的类型。

一种替代方式是创建一个类型类。我们想从一个操作文件的 monad 中获取接口，这个类型类指定了这个接口。

```haskell
-- file: ch15/MonadHandle.hs
{-# LANGUAGE FunctionalDependencies, MultiParamTypeClasses #-}

module MonadHandle (MonadHandle(..)) where

import System.IO (IOMode(..))

class Monad m => MonadHandle h m | m -> h where
        openFile :: FilePath -> IOMode -> m h
        hPutStr :: h -> String -> m ()
        hClose :: h -> m ()
        hGetContents :: h -> m String

        hPutStrLn :: h -> String -> m ()
        hPutStrLn h s = hPutStr h s >> hPutStr h "\n"
```

现在，我们决定抽象出 monad 的类型和文件句柄的类型。为了满足类型检查器，这里添加了一个函数依赖：对于 `MonadHandle` 的任何实例，只有一个句柄类型可以使用。当 `IO` monad 成为这个类的一个实例时，我们使用一个普通的 `Handle` 。

```haskell
-- file: ch15/MonadHandleIO.hs
{-# LANGUAGE FunctionalDependencies, MultiParamTypeClasses #-}

import MonadHandle
import qualified System.IO

import System.IO (IOMode(..))
import Control.Monad.Trans (MonadIO(..), MonadTrans(..))
import System.Directory (removeFile)

import SafeHello

instance MonadHandle System.IO.Handle IO where
        openFile = System.IO.openFile
        hPutStr = System.IO.hPutStr
        hClose = System.IO.hClose
        hGetContents = System.IO.hGetContents
        hPutStrLn = System.IO.hPutStrLn
```

因为任何 MonadHandle 也必须是一个 Monad，所以我们可以使用普通的 do 标识符编写修改文件的代码，而不需要关心它最终在哪个 monad 中执行。

```haskell
-- file: ch15/SafeHello.hs
safeHello :: MonadHandle h m => FilePath -> m ()
safeHello path = do
  h <- openFile path WriteMode
  hPutStrLn h "hello world"
  hClose h
```

我们已经让 IO 成为这个类型类的一个实例，现在可以在 **ghci** 中执行这个动作：

```
ghci> safeHello "hello to my fans in domestic surveillance"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> removeFile "hello to my fans in domestic surveillance"
```

使用类型类的好处在于，我们可以不需要接触过多代码并互换一个底层的 monad，因为大多数代码不知道或不关心具体底层 monad 的实现。例如，可以用一个 monad 来替换 IO，它会在文件写入时压缩文件。

通过类型类定义一个 monad 的接口有另一个好处：它允许其他人在 newtype 包装器中隐藏我们的实现，并自动派生他们想要暴露的类型类的实例。

15.8.4 隔离和测试

事实上，因为 safeHello 函数没有使用 IO 类型，我们甚至可以使用不能执行 I/O 的 monad。通过这种方法，我们可以在纯粹且受控的环境中对那些在平常情况下会出现副作用的代码进行测试。

为此，我们将创建一个 monad，这个 monad 不会执行任何 I/O 操作，但它会记录每个与文件相关联的事件以供后续处理。

```
-- file: ch15/WriterIO.hs
data Event = Open FilePath IOMode
           | Put String String
           | Close String
           | GetContents String
                deriving (Show)
```

虽然在*使用新的 Monad* 中已经开发了一个 Logger 类型，但这里我们将使用标准的、更广泛的 Writer monad。和其它 mtl monad 类似，Writer 提供的 API 将在一个类型类中被定义，也就是接下来要展示的 MonadWriter 。MonadWriter 最有用的方法是 tell ，这个方法记录了一个值。

```
ghci> :m +Control.Monad.Writer
ghci> :type tell
tell :: (MonadWriter w m) => w -> m ()
```

虽然 tell 方法可以记录任意类型的 Monoid ，但是因为列表的类型是 Monoid ，所以 tell 方法将记录一个由 Event 构成的列表。

尽管我们可以让 Writer [Event] 成为 MonadHandle 的一个实例，但一个更廉价也更安全的做法是编写一个特殊的 monad ，并且要做到这一点也比较容易。

```
-- file: ch15/WriterIO.hs
newtype WriterIO a = W { runW :: Writer [Event] a }
        deriving (Monad, MonadWriter [Event])
```

执行函数首先会移除我们之前添加的 newtype 包装器，然后再调用普通 Writer monad 的执行函数。

```
-- file: ch15/WriterIO.hs
runWriterIO :: WriterIO a -> (a, [Event])
runWriterIO = runWriter . runW
```

当我们在 **ghci** 中执行这段代码的时候，它将反馈给我们一份日志，这份日志记录了函数对文件所做的行为。

```
ghci> :load WriterIO
[1 of 3] Compiling MonadHandle      ( MonadHandle.hs, interpreted )
[2 of 3] Compiling SafeHello        ( SafeHello.hs, interpreted )
[3 of 3] Compiling WriterIO         ( WriterIO.hs, interpreted )
```

(continues on next page)

(continued from previous page)

```
Ok, modules loaded: SafeHello, MonadHandle, WriterIO.
ghci> runWriterIO (safeHello "foo")
((),[Open "foo" WriteMode,Put "foo" "hello world",Put "foo" "\n",Close "foo"])
```

15.8.5 Writer monad 和列表

每当我们使用 `tell` 方法时，`Writer` monad 都会调用 monoid 的 `mappend` 函数。因为列表的 `mappend` 动作是 `(++)`，而不断重复附加操作是非常浪费资源的，所以在实际使用 `Writer` 时，列表并不是一个很好的选择。在上面的实例中，我们纯粹是为了追求简单才使用了列表。

在生产代码中，如果你想使用 `Writer` monad，并且需要类似列表的行为，那么你应当选择那些在执行附加操作时性能更优的类型。在把函数当成数据来用 中介绍过的差异列表就是一个这样的类型。你不需要实现自己的差异列表：Haskell 包数据库 Hackage 提供了一个调试好的库，你可以直接下载使用。除此之外，你可以使用 `Data.Sequence` 模块中的 `Seq` 类型，我们在通用序列 中介绍过这个类型。

15.8.6 任意 I/O 访问

在使用类型类方法限制 `IO` 的同时，我们可能还会希望继续保留执行任意 I/O 操作的能力。为此，我们可以尝试将 `MonadIO` 作为约束添加到类型类里面。

```
-- file: ch15/MonadHandleIO.hs
class (MonadHandle h m, MonadIO m) => MonadHandleIO h m | m -> h

instance MonadHandleIO System.IO.Handle IO

tidierHello :: (MonadHandleIO h m) => FilePath -> m ()
tidierHello path = do
  safeHello path
  liftIO (removeFile path)
```

但是这种方法会导致一个问题：添加 `MonadIO` 约束将使得我们无法再判断一个测试是否会带有副作用，我们也因此失去了在纯粹的环境中测试代码的能力。另一种方法是调整这个约束的作用域，使它只影响那些真正需要执行 I/O 操作的函数，而非所有函数。

```
-- file: ch15/MonadHandleIO.hs
tidyHello :: (MonadIO m, MonadHandle h m) => FilePath -> m ()
tidyHello path = do
  safeHello path
  liftIO (removeFile path)
```

我们可以对不受 `MonadIO` 约束的函数使用纯属性测试，对其余的函数使用传统的单元测试。

遗憾的是，这种做法只不过是将一个问题换成了另一个问题：只受 `MonadHandle` 约束的代码将无法调用同时受 `MonadIO` 和 `MonadHandle` 约束的代码。如果我们在 `MonadHandle` 约束的代码内部发现了这个问题，那么我们就必须在引发这个问题的所有代码路径上都加上 `MonadIO` 约束。

允许执行任意 I/O 操作是有风险的，而且这么做对我们开发、测试代码的流程都会有巨大的影响。相比放宽对代码行为的限制，我们通常会追求更清晰的代码逻辑和更容易的测试环境。

15.8.7 练习

1. 使用 Quick Check，为 `MonadHandle` monad 中的一个动作编写一个测试，观察它是否尝试向一个未打开的文件句柄写入。在 `safeHello` 下尝试。
2. 编写一个试图向已关闭句柄写入的动作。你的测试捕获到这个问题了吗?
3. 在表单编码的字符串中，相同的键可能出现数次，每个键具有或不具有对应的值，例如 `key&key=1&key=2` 。你会用什么类型来表示这类字符串中和键相关联的值? 编写一个正确捕获所有信息的解析器。

CHAPTER

SIXTEEN

第 16 章：使用 PARSEC

为一个文本文件或者不同类型的数据做语法分析 (parsing)，对程序员来说是个很常见的任务，在本书第 198 页“使用正则表达式”一节中，我们已经学习了 Haskell 对正则表达式的支持。对很多这样的任务，正则表达式都很好用。

不过，当处理复杂的数据格式时，正则表达式很快就会变得不实用、甚至完全不可用。比如说，对于多数编程语言来说，我们没法（只）用正则表达式去 parse 其源代码。

Parsec 是一个很有用的 parser combinator 库，使用 Parsec，我们可以将一些小的、简单的 parser 组合成更复杂的 parser。Parsec 提供了一些简单的 parser，以及一些用于将这些 parser 组合在一起的组合子。毫不意外，这个为 Haskell 设计的 parser 库是函数式的。

将 Parsec 同其他语言的 parse 工具做下对比是很有帮助的，语法分析有时会被分为两个阶段：词法分析（这方面的工具比如 `flex` ）和语法分析（比如 `bison`). Parsec 可以同时处理词法分析和语法分析。(译注：词法分析将输入的字符串序列转化为一个个的 token，而语法分析进一步接受这些 token 作为输入生成语法树）

16.1 Parsec 初步：简单的 CSV parser

让我们来写一个解析 CSV 文件的代码。CSV 是纯文本文件，常被用来表示表格或者数据库。每行是一个记录，一个记录中的字段用逗号分隔。至于包含逗号的字段，有特殊的处理方法，不过在这一节我们暂时不考虑这种情况。

下面的代码比实际需要的代码要长一些，不过接下来，我们很快就会介绍一些 Parsec 的特性，应用这些特性，整个 parser 只需要四行。

```
-- file: ch16/csv1.hs
import Text.ParserCombinators.Parsec

{- A CSV file contains 0 or more lines, each of which is terminated
   by the end-of-line character (eol). -}
csvFile :: GenParser Char st [[String]]
csvFile =
```

(continues on next page)

(continued from previous page)

```
    do result <- many line
       eof
       return result

-- Each line contains 1 or more cells, separated by a comma
line :: GenParser Char st [String]
line =
    do result <- cells
       eol                       -- end of line
       return result

-- Build up a list of cells.  Try to parse the first cell, then figure out
-- what ends the cell.
cells :: GenParser Char st [String]
cells =
    do first <- cellContent
       next <- remainingCells
       return (first : next)

-- The cell either ends with a comma, indicating that 1 or more cells follow,
-- or it doesn't, indicating that we're at the end of the cells for this line
remainingCells :: GenParser Char st [String]
remainingCells =
    (char ',' >> cells)            -- Found comma?  More cells coming
    <|> (return [])                -- No comma?  Return [], no more cells

-- Each cell contains 0 or more characters, which must not be a comma or
-- EOL
cellContent :: GenParser Char st String
cellContent =
    many (noneOf ",\n")


-- The end of line character is \n
eol :: GenParser Char st Char
eol = char '\n'

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

我们来讲解下这段代码，在这段代码中，我们并没有使用 Parsec 的特性，因此要记住这段代码还能写得更简洁!

我们自顶向下的构建了一个 CSV 的 parser，第一个函数是 `csvFile`。它的类型是 `GenParser Char st`

`[[String]]`,这表示这个函数的输入是字符序列,也就是 Haskell 中的字符串,因为 `String` 不过是 `[Char]` 的别名，而这个函数的返回类型是 `[[String]]`：一个字符串列表的列表。至于 `st`，我们暂时忽略它

Parsec 程序员经常会写一些小函数，因此他们常常懒得写函数的类型签名。Haskell 的类型推导系统能够自动识别函数类型。而在上面第一个例子中，我们写出了所有函数的类型，方便你了解函数到底在干什么。另外你可以在 `ghci` 中使用 `:t` 来查看函数的类型。

`csvFile` 函数使用了 `do` 语句，如其所示，Parsec 库是 monadic 的，它定义了用于语法分析的 [1][ref1]: `Genparser` monad。

`csvFile` 函数首先运行的是 `many line`，`many` 是一个高阶函数，它接受一个 parser 函数作为参数，不断对输入应用这个 parser，并把每次 parse 的结果组成一个列表返回。在 `csvFile` 中，我们把对 csv 文件中所有行的解析结果存储到 `result` 中，然后，当我们遇到文件终结符 EOF 时，就返回 `result`。也就是说：一个 CSV 文件有好多行组成，以 EOF 结尾。Parsec 写成的函数如此简洁，我们常常能够像这样直接用语言来解释。

上一段说，一个 CSV 文件由许多行组成，现在，我们需要说明，什么是“一行”，为此，我们定义了 `line` 函数来解析 CSV 文件中的一行，通过阅读函数代码，我们可以发现，CSV 文件中的一行，包括许多“单元格”，最后跟着一个换行符。

那么，什么是“许多单元格”呢，我们通过 `cells` 函数来解析一行中的所有单元格。一行中的所有单元格，包括一个到多个单元格。因此，我们首先解析第一个单元格的内容，然后，解析剩下的单元格，返回剩下的单元格内容组成的列表，最后，`cells` 把第一个单元格与剩余单元格列表组成一个新的单元格列表返回。

我们先跳过 `remainingCells` 函数，去看 `cellContent` 函数，`cellContent` 解析一个单元格的内容。一个单元格可以包含任意数量的字符，但每一个字符都不能是逗号或者换行符（译注：实际可以包含逗号，不过我们目前不考虑这种情况），我们使用 `noneOf` 函数来匹配这两个特殊字符，来确保我们遇到的不是这样的字符，于是，`many noneOf ",\n"` 定义了一个单元格。

然后再来看 `remainingCells` 函数，这个函数用来在解析完一行中第一个单元格之后，解析该行中剩余的单元格。在这个函数中，我们初次使用了 Parsec 中的选择操作，选择操作符是 `<|>`。这个操作符是这样定义的：它会首先尝试操作符左边的 parser 函数，如果这个 parser 没能成功消耗任何输入字符（译注：没有消耗任何输入，即是说，从输入字符串的第一个字符，就可以判定无法成功解析，例如，我们希望解析” html” 这个字符串，遇到的却是” php”，那从” php” 的第一个字符’ p’，就可以判定不会解析成功。而如果遇到的是” http”，那么我们需要消耗掉” ht” 这两个字符之后，才判定匹配失败，此时，即使已经匹配失败，” ht” 这两个字符仍然是被消耗掉了），那么，就尝试操作符右边的 parser。

在函数 `remainingCells` 中，我们的任务是去解析第一个单元格之后的所有单元格，`cellContent` 函数使用了 `noneOf ",\n"`，所以逗号和换行符不会被 `cellContent` 消耗掉，因此，如果我们在解析完一个单元格之后，见到了一个逗号，这说明这一行不止一个单元格。所以，`remainingCells` 选择操作中的第一个选择的开始是一个 `char ','` 来判断是否还有剩余单元格，`char` 这个 parser 简单的匹配输入中传入的字符，如果我们发现一个逗号，我们希望这个去继续解析剩余的单元格，这个时候，“剩下的单元格” 看上去跟一行中的所有单元格在格式上一致。所以，我们递归地调用 `cells` 去解析它们。如果我们没有发现逗号，说明这一行中再没有剩余的单元格，就返回一个空列表。

最后，我们需要定义换行符，我们将换行符设定为字符’ \n’,这个设定到目前来讲已经够用了。

在整个程序的最后，我们定义函数 parseCSV，它接受一个 String 类型的参数，并将其作为 CSV 文件进行解析。这个函数只是对 Parsec 中 parse 函数的简单封装，parse 函数返回 Either ParseError [[String]] 类型，如果输入格式有错误，则返回的是用 Left 标记的错误信息，否则，返回用 Right 标记的解析生成的数据类型。

理解了上面的代码之后，我们试着在 ghci 中运行一下来看下它：

```
ghci> :l csv1.hs
[1 of 1] Compiling Main             ( csv1.hs, interpreted )
Ok, modules loaded: Main.
ghci> parseCSV ""
Loading package parsec-2.1.0.0 ... linking ... done.
Right []
```

结果倒是合情合理，parse 一个空字符串，返回一个空列表。接下来，我们去 parse 一个单元格：

```
ghci> parseCSV "hi"
Left "(unknown)" (line 1, column 3):
unexpected end of input
expecting "," or "\n"
```

看下上面的报错信息，我们定义“一行”必须以一个换行符结尾，而在上面的输入中，我们并没有给出换行符。Parsec 的报错信息给出了错误的行号和列号，甚至告诉了我们它期望得到的输入。我们对上面的输入给出换行符，并且继续尝试新的输入：

```
ghci> parseCSV "hi\n"
Right [["hi"]]
ghci> parseCSV "line1\nline2\nline3\n"
Right [["line1"],["line2"],["line3"]]
ghci> parseCSV "cell1,cell2,cell3\n"
Right [["cell1","cell2","cell3"]]
ghci> parseCSV "l1c1,l1c2\nl2c1,l2c2\n"
Right [["l1c1","l1c2"],["l2c1","l2c2"]]
ghci> parseCSV "Hi,\n\n,Hello\n"
Right [["Hi",""],[""],["","Hello"]]
```

可以看出，parseCSV 的行为与预期一致，甚至空单元格与空行它也能正确处理。

16.2 sepBy 与 endBy 组合子

我们早先向您承诺过，上一节中的 CSV parser 可以通过几个辅助函数大大简化。有两个函数可以大幅度简化上一节中的代码。

第一个工具是 sepBy 函数，这个函数接受两个 parser 函数作为参数。第一个函数解析有效内容，第二个函

数解析一个分隔符。sepBy 首先尝试解析有效内容，然后去解析分隔符，然后有效内容与分隔符依次交替解析，直到解析完有效内容之后无法继续解析到分隔符为止。它返回有效内容的列表。

第二个工具是 endBy，它与 sepBy 相似，不过它期望它的最后一个有效内容之后，还跟着一个分隔符（译注，就是 parse “a\nb\nc\n” 这种，而 sepBy 是 parse “a,b,c” 这种）。也就是说，它将一直进行 parse，直到它无法继续消耗任何输入。

于是，我们可以用 endBy 来解析行，因为每一行必定是以一个换行字符结尾。我们可以用 sepBy 来解析一行中的所有单元格，因为一行中的单元格以逗号分割，而最后一个单元格后面并不跟着逗号。我们来看下现在的 parser 有多么简单：

```
-- file: ch16/csv2.hs
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line    = sepBy cell (char ',')
cell    = many (noneOf ",\n")
eol     = char '\n'

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

这个程序的行为同上一节中的一样，我们可以通过使用 ghci 重新运行上一节中的测试用例来验证，我们会得到完全相同的结果。然而现在的程序更短、可读性更好。你不用花太多时间就能把这段代码翻译成中文描述，当你阅读这段代码时，你将看到：

- 一个 CSV 文件包含 0 行或者更多行，每一行都是以换行符结尾。
- 一行包含一个或者多个单元格（**译者注，sepBy 应该是允许 0 个单元格的**）
- 一个单元格包含 0 个或者更多个字符，这些字符不能是逗号或者换行符
- 换行符是’\n’

16.3 选择与错误处理

不同操作系统采用不同的字符来表示换行，例如，Unix/Linux 系统中，以及 Windows 的 text mode 中，简单地用 “\n” 来表示。DOS 以及 Windows 系统，使用 “\r\n”，而 Mac 一直采用 “\r”。我们还可以添加对 “\n\r” 的支持，因为有些人可能会需要。

我们可以很容易地修改下上面的代码来适应这些不同的换行符。我们只需要做两处改动，修改下 eol 的定义，使它识别不同的换行符，修改下 cell 函数中的 noneOf 的匹配模式，让它忽略 “\r”。

这事做起来得小心些，之前 eol 的定义就是简单的 char '\n'，而现在我们使用另一个内置的 parser 函数叫做 string，它可以匹配一个给定的字符串，我们来考虑下如何用这个函数来增加对 “\n\r” 的支持。

我们的初次尝试，就像这样：

```
-- file: ch16/csv3.hs
-- This function is not correct!
eol = string "\n" <|> string "\n\r"
```

然而上面的例子并不正确，<|> 操作符总是首先尝试左边的 parser，即 `string "\n"`，但是对于“\n”和“\n\r”这两种换行符，`string "\n"` 都会匹配成功，这可不是我们想要的，不妨在 `ghci` 中尝试一下：

```
ghci> :m Text.ParserCombinators.Parsec
ghci> let eol = string "\n" <|> string "\n\r"
Loading package parsec-2.1.0.0 ... linking ... done.
ghci> parse eol "" "\n"
Right "\n"
ghci> parse eol "" "\n\r"
Right "\n"
```

看上去这个 parser 对与两种换行符都能够正常工作，不过，仅凭上面的结果我们并不能确认这一点。如果 parser 留下了一些没有解析的部分，我们也无从知晓，因为我们解析完换行符后没有再试图去消耗剩余输入。所以让我们在换行符后面加一个文件终止符 `eof`，表示我们期望在解析完换行符之后，没有剩余的带解析输入了：

```
ghci> parse (eol >> eof) "" "\n\r"
Left (line 2, column 1):
unexpected "\r"
expecting end of input
ghci> parse (eol >> eof) "" "\n"
Right ()
```

正如预期的那样，当解析“\n\r”换行符时出现了错误，所以接下来我们可能会想这样尝试：

```
-- file: ch16/csv4.hs
-- This function is not correct!
eol = string "\n\r" <|> string "\n"haskell
```

这也是不对的。回想一下，<|> 仅在左侧的选项没有消耗输入时，才会尝试在右边的 parser。但是，当我们去看在“\n”后面是不是有一个“\r”的时候，我们早就已经消耗掉了一个“\n”，我们会在 parse “\n”时遇到错误：

```
ghci> :m Text.ParserCombinators.Parsec
ghci> let eol = string "\n\r" <|> string "\n"
Loading package parsec-2.1.0.0 ... linking ... done.
ghci> parse (eol >> eof) "" "\n\r"
Right ()
```

(continues on next page)

(continued from previous page)

```
ghci> parse (eol >> eof) "" "\n"
Left (line 1, column 1):
unexpected end of input
expecting "\n\r"
```

我们在超前查看的问题上栽了跟头，看起来，在写 parser 的时候，能够在数据到来时“超前查看”是很有用的。Parsec 是支持这一特性的，不过在我们展示这一特性的时候，先来看看怎样能够不利用超前查看特性完成这个任务。你必须要自己去考虑“\n”之后的所有可能：

```
-- file: ch16/csv5.hs
eol =
    do char '\n'
       char '\r' <|> return '\n'
```

这个函数首先寻找“\n”，如果找到了，就去寻找“\r”，如果找到了“\r”，就消耗掉“\r”。既然 `char '\r'` 的返回类型是 `Char`，那么没有找到‘\r’时的行为就是简单的返回一个‘Char’而不试图 parse 任何输入。Parsec 有一个内置函数 `option` 可以将这种情况表达为 `option '\n' (char '\r')`。我们在 `ghci` 中试一下：

```
ghci> :l csv5.hs
[1 of 1] Compiling Main             ( csv5.hs, interpreted )
Ok, modules loaded: Main.
ghci> parse (eol >> eof) "" "\n\r"
Loading package parsec-2.1.0.0 ... linking ... done.
Right ()
ghci> parse (eol >> eof) "" "\n"
Right ()
```

这次结果是对的！不过，利用 Parsec 对 lookahead 的支持，代码可以更加简洁。

16.3.1 超前查看

Parsec 有一个内置函数叫做 `try` 用来支持超前查看，`try` 接受一个 parser 函数，将它应用到输入。如果这个 parser 没有成功，那么 `try` 表现地就像它不曾消耗任何输入。所以，如果你在 `<|>` 的左侧应用 `try`，那么，即使左侧 parser 在失败时会消耗掉一些输入，Parsec 仍然会去尝试右侧的 parser。`try` 只有在 `<|>` 左侧时才会有效。不过，许多函数会在内部使用 `<|>`。让我们来用 `try` 扩展对换行符的支持：

```
-- file: ch16/csv6.hs
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
```

(continues on next page)

(continued from previous page)

```
cell = many (noneOf ",\n\r")

eol =   try (string "\n\r")
    <|> try (string "\r\n")
    <|> string "\n"
    <|> string "\r"

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

这里，我们把两个包含两个字符的换行符放在开头，并且用 `try` 去检查它们。这两个换行符的 parser 都出现在 `<|>` 的左侧，因此不会有什么问题。我们也可以把 `string "\n"` 放到 `try` 中，不过这其实没什么必要，因为它只用检验一个字符，因此当解析失败时不会消耗输入，我们把代码加载进 `ghci` 去看下运行结果：

```
ghci> :l csv6.hs
[1 of 1] Compiling Main             ( csv6.hs, interpreted )
Ok, modules loaded: Main.
ghci> parse (eol >> eof) "" "\n\r"
Loading package parsec-2.1.0.0 ... linking ... done.
Right ()
ghci> parse (eol >> eof) "" "\n"
Right ()
ghci> parse (eol >> eof) "" "\r\n"
Right ()
ghci> parse (eol >> eof) "" "\r"
Right ()
```

四种换行符都能正确的处理，你也可以用不同的换行符来测试完整的 CSV parser，就像这样：

```
ghci> parseCSV "line1\r\nline2\nline3\n\rline4\rline5\n"
Right [["line1"],["line2"],["line3"],["line4"],["line5"]]
```

如你所见，现在我们的 parser 支持在单个文件中使用多种换行符啦。

16.3.2 错误处理

本章开头，我们已经看到 Parsec 的报错信息能够列出错误的具体位置以及它期望的输入。可是，当 parser 变得更加复杂的时候，Parsec 的期望输入列表会变得很复杂。不过 Parsec 也提供了一套机制让你来在解析失败时自定义出错信息。

我们来看下现在的 CSV parser 在遇到错误时给出的错误信息：

```
ghci> parseCSV "line1"
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r
```

这个报错信息有点长，并且包含了太多的技术细节。我们可以试着用 Monad 中的 `fail` 函数来改善以下：

```
-- file: ch16/csv7.hs
eol =   try (string "\n\r")
    <|> try (string "\r\n")
    <|> string "\n"
    <|> string "\r"
    <|> fail "Couldn't find EOL"
```

在 `ghci` 中测试，结果如下：

```
ghci> :l csv7.hs
[1 of 1] Compiling Main             ( csv7.hs, interpreted )
Ok, modules loaded: Main.
ghci> parseCSV "line1"
Loading package parsec-2.1.0.0 ... linking ... done.
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r"
Couldn't find EOL
```

`fail` 函数把“Couldn’t find EOL”追加到了原有的错误信息后面，而不是替换掉了原有的错误信息。Parsec 有一个内置的 <?> 操作符专门针对后一种需求。它跟 <|> 操作符很像，首先尝试操作符左边的 parser，不过，左边解析失败时并不是去尝试另一个 parser，而是呈现一段错误信息。下面是它的使用方法：

```
-- file: ch16/csv8.hs
eol =   try (string "\n\r")
    <|> try (string "\r\n")
    <|> string "\n"
    <|> string "\r"
    <?> "end of line"
```

现在，当你 parse 失败时，你会得到更有用的错误信息：

```
ghci> :l csv8.hs
[1 of 1] Compiling Main             ( csv8.hs, interpreted )
Ok, modules loaded: Main.
ghci> parseCSV "line1"
Loading package parsec-2.1.0.0 ... linking ... done.
```

(continues on next page)

(continued from previous page)

```
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting "," or end of line
```

现在报错信息很有用！通常来说，你需要在 <?> 右侧放上可读性较好的报错信息。

16.4 完整的 CSV parser

上面的 CSV parser 的例子有一个很严重的问题：它无法处理单元格中包含逗号的情况。CSV 生成程序通常会把包含逗号的单元格用引号引起。但这又产生了新问题：如果单元格中同时包含引号和逗号怎么办？在这种情况下，用两个引号来表示单元格中的一个引号。

下面是一个完整的 CSV parser，你可以在 ghci 中使用它，或者把它编译成独立的程序，它会解析从标准输入读取的 CSV 文件内容，并把它转化成另一格式的输出。

```
-- file: ch16/csv9.hs
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
cell = quotedCell <|> many (noneOf ",\n\r")

quotedCell =
    do char '"'
       content <- many quotedChar
       char '"' <?> "quote at end of cell"
       return content

quotedChar =
        noneOf "\""
    <|> try (string "\"\"" >> return '"')

eol =   try (string "\n\r")
    <|> try (string "\r\n")
    <|> string "\n"
    <|> string "\r"
    <?> "end of line"

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input

main =
```

(continues on next page)

(continued from previous page)

```
    do c <- getContents
       case parse csvFile "(stdin)" c of
            Left e -> do putStrLn "Error parsing input:"
                         print e
            Right r -> mapM_ print r
```

这是一个完整的 CSV parser，parser 部分只有 21 行代码，外加 10 行代码用来写 `parseCSV` 和 `main` 这两个函数。

我们来分析以下这个程序跟上一版本的区别。首先，一个单元格可能是一个普通的单元格或者是一个 “引用” 的单元格。在这两个选项中，我们首先用 `quotedCell` 来检查单元格是否是引用单元格，因为这可以通过检查单元格第一个字符是否是引号来实现。（译注：这样可以通过第一个字符判定单元格类型，从而避免使用 `try`）。

`quotedCell` 由引用标志双引号开始和结束，其中包含零到多个字符。不过我们不能直接获取这些字符，因为其中可能包含嵌在单元格内容之中的双引号，此时是用两个双引号表示一个嵌入双引号。所以我们定义函数 `quotedChar` 来处理 `quotedCell` 中的内容。

当我们处理一个引用单元格内的字符时，我们先考虑 `noneOf "\""`，这将会匹配并返回所有的非引号字符。而如果我们遇到了引号，我们就检查它是不是两个连续的引号，如果是，就返回一个双引号，否则报错。

注意到在 `quotedChar` 中，`try` 是出现在 `<|>` 的右侧的。而我们之前提过，`try` 只有当它出现再 `<|>` 的左侧时才会有效。事实上，这个 `try` 确实是出现在 `<|>` 的左侧的，不过是出现在 `many` 的实现中包含的 `<|>` 的左侧。（译注：虽然在 `quotedChar` 中，`try` 出现在 `<|>` 的右侧，但是当使用 `many quotedChar` 时，`many` 的实现使得 `try` 会出现在其内部的 `<|>` 的左侧。）

`try` 的使用在这里是很重要的。假如我们在解析一个引用单元格，并且这个单元格快要解析完了，在这个单元格后面还有下一个单元格。那么，在当前单元格的结尾，我们会看到一个引号，接着是一个逗号。当 parse 到单元格结尾时，调用 `quotedChar` 时，首先，`noneOf` 的测试会失败，接着会进行寻找两个连续引号的测试，这个测试也会失败，因为我们看到的是一个引号和一个逗号。如果我们不使用 `try`，parser 会在看到一个引号之后，期望下一个引号，而且此时第一个引号已经被 parser 给消耗掉了。如果我们使用了 `try`，那么这种情况就会被正确的识别为不是单元格的内容，所以 `many quotedChar` 就会终止。于是，超前查看又一次被证明是十分有用的，并且因为它用起来十分简单，它已经成为 Parsec 中十分引人注目的工具。

我们可以在 `ghci` 中用引用单元格来测试这个程序：

```
ghci> :l csv9.hs
[1 of 1] Compiling Main             ( csv9.hs, interpreted )
Ok, modules loaded: Main.
ghci> parseCSV "\"This, is, one, big, cell\"\n"
Loading package parsec-2.1.0.0 ... linking ... done.
Right [["This, is, one, big, cell"]]
ghci> parseCSV "\"Cell without an end\n"
Left "(unknown)" (line 2, column 1):
```

(continues on next page)

(continued from previous page)

```
unexpected end of input
expecting "\"\"" or quote at end of cell
```

我们来试一下真正的 CSV 文件，下面是一个电子表格程序生成的文件内容：

```
"Product","Price"
"O'Reilly Socks",10
"Shirt with ""Haskell"" text",20
"Shirt, ""O'Reilly"" version",20
"Haskell Caps",15
```

现在，我们用这个文件来测试下我们的程序：

```
$ runhaskell csv9.hs < test.csv
["Product","Price"]
["O'Reilly Socks","10"]
["Shirt with \"Haskell\" text","20"]
["Shirt, \"O'Reilly\" version","20"]
["Haskell Caps","15"]
```

16.5 Parsec 与 MonadPlus

我们在 “Looking for alternatives” 一节介绍过 `MonadPlus`，Parsec 的 `Genparser` moand 是 `MonadPlus` 类型类的一个实例。`mzero` 代表 parse 失败，而 `mplus` 则使用 `(<|>)` 把两个 parser 组合成一个。

```
-- file: ch16/ParsecPlus.hs
instance MonadPlus (GenParser tok st) where
    mzero = fail "mzero"
    mplus = (<|>)
```

16.6 解析 URL 编码查询字符串

当我们在 “Golfing practice: association lists” 一节提到 `application/x-www-form-urlencoded` 文本时，我们曾说过之后会为它写一个 parser，现在，我们可以用 Parsec 轻易的实现。

每个键-值对由 `&` 字符分隔。

```
-- file: ch16/FormParse.hs
p_query :: CharParser () [(String, Maybe String)]
p_query = p_pair `sepBy` char '&'
```

注意上面函数的类型签名，我们使用 Maybe 来表示一个值：因为 HTTP 标准中并没有规定一个键必定有一个与之对应的值。我们希望能够区分“没有值”和“空值”。

```
-- file: ch16/FormParse.hs
p_pair :: CharParser () (String, Maybe String)
p_pair = do
  name <- many1 p_char
  value <- optionMaybe (char '=' >> many p_char)
  return (name, value)
```

many1 的功能类似与 many：它反复应用一个 parser，返回 parse 的结果列表。不过，当 parser 从未成功时，many 会返回空列表，而 many1 则会失败，也就是说，many1 会返回至少包含一个元素的列表。

optionMaybe 函数接受一个 parser 作为参数，并修改它的行为，当该 parser 解析失败时，optionMaybe 返回 Nothing，成功时，则把 parser 的返回结果用 Just 封装。这就让我们能够区分“没有值”和“空值”。

译注:, 对于 optionMaybe, parser 失败时并不一定是返回 Nothing, 跟 (<|>) 类似, 只有当 optionMaybe 的 parser parse 失败，并且没有消耗任何输入时，才会返回 Nothing，否则，仍然是失败，如下列代码所示:

```
Prelude Text.ParserCombinators.Parsec> let p = string "html" :: Parser String
Prelude Text.ParserCombinators.Parsec> parseTest p "html"
"html"
Prelude Text.ParserCombinators.Parsec> parseTest p "http"
parse error at (line 1, column 1):
unexpected "t"
expecting "html"
Prelude Text.ParserCombinators.Parsec> let f = optionMaybe p
Prelude Text.ParserCombinators.Parsec> parseTest f "http"
parse error at (line 1, column 1):
unexpected "t"
expecting "html"
Prelude Text.ParserCombinators.Parsec> parseTest f "php"
Nothing
Prelude Text.ParserCombinators.Parsec>
```

单独的字符可以以如下集中方式编码

```
-- file: ch16/FormParse.hs
import Numeric
p_char :: CharParser () Char
p_char = oneOf urlBaseChars
     <|> (char '+' >> return ' ')
     <|> p_hex

urlBaseChars = ['a'..'z']++['A'..'Z']++['0'..'9']++"$-_.!*'(),"
```

(continues on next page)

(continued from previous page)

```
p_hex :: CharParser () Char
p_hex = do
  char '%'
  a <- hexDigit
  b <- hexDigit
  let ((d, _):_) = readHex [a,b]
  return . toEnum $ d
```

有些字符可以直接表示。空格需要单独表示，空格用字符 + 来表示，其他字符则用一个 % 外加两个 16 进制数字来表示，Numeric 模块中的 readHex 函数可以把一个 16 进制字符串解析为一个数字。

```
ghci> parseTest p_query "foo=bar&a%21=b+c"
Loading package parsec-2.1.0.0 ... linking ... done.
[("foo",Just "bar"),("a!",Just "b c")]
```

As appealing and readable as this parser is, we can profit from stepping back and taking another look at some of our building blocks.

16.7 用 Parsec 代替正则表达式来进行临时的 parse

在很多流行的语言中，程序员喜欢用正则表达式来进行“临时的”解析工作，不过，正则表达式既难写，又难调试，如果代码写完后几个月不管，就几乎无法理解，并且失败时没有报错信息。

如果我们用 Parsec 编写紧凑的 parser，我们的代码将拥有可读性、表现力以及有用的报错信息。虽然用 Parsec 编写的代码可能会比正则表达式更长，不过也不会长太多，大抵能够抵消正则表达式的许多诱惑了。

16.8 解析时不用变量

上面的一些 parser 使用了 do 标记语法，把一些中间的解析结果绑定到变量，以便过后使用，比如说，p_pair。

```
-- file: ch16/FormParse.hs
p_pair :: CharParser () (String, Maybe String)
p_pair = do
  name <- many1 p_char
  value <- optionMaybe (char '=' >> many p_char)
  return (name, value)
```

我们可以使用 Control.Monad 模块中的 liftM2 函数，不使用变量来完成上面的工作：

```haskell
-- file: ch16/FormParse.hs
p_pair_app1 =
    liftM2 (,) (many1 p_char) (optionMaybe (char '=' >> many p_char))
```

这个函数跟 p_pair 有相同的类型与行为，不过它只有一行。在这里，我们不使用“过程式”的风格来写 parser，而是更加强调应用 parser 以及 parser 的组合。

这种（无变量的）风格称为 applicative 风格，我们可以在编写 applicative 风格 parser 的路上走的更远一些。大多数情况下，除了刚开始要理解这种风格需要一点最初的努力之外，applicative 风格带来的代码紧凑型并不会牺牲代码的可读性。

16.9 使用 Applicative Functor 进行 parse

Haskell 标准库中包含一个叫做 Control.Applicative 的模块，我们已经在 “Infix use of fmap” 一节见识过了。这个模块定义了一个叫做 Applicative 的类型类，它表示一个 *Applicative Functor*，Applicative Functor 在结构化方面比 Functor 更强，不过比 Monad 稍弱。Control.Applicative 模块也定义了 Alternative 类型类，它跟 MonadPlus 很相似。

像往常一样，我们认为理解 Applicative Functor 的最好的方式通过使用它们来讲解。从理论上讲，每个 Monad 都是一个 Applicative functor，但不是每一个 Applicative Functor 都是一个 Monad。由于 Applicative Functor 是在 Monad 之后很久才加入标准库，我们常常不能免费获得一个 Applicative 实例，我们常常需要自己把正在使用的 Monad 声明为 Applicative。

译注：至少在我用的 GHC 7.8.1/GHC 7.10 里，Parser 已经是 Applicative 了。不需要自己实现。而且，在 GHC 7.10 中，每一个 Monad 都会强制要求声明为 Applicative，不过又据说 GHC 7.12 可能会取消这一限制。

要在 Parsec 中做到这一点，我们将写一个小模块来将 Parsec 实现为 Applicative，然后我们导入这个模块，而不是通常的 Parsec 模块。

```haskell
-- file: ch16/ApplicativeParsec.hs
module ApplicativeParsec
    (
      module Control.Applicative
    , module Text.ParserCombinators.Parsec
    ) where

import Control.Applicative
import Control.Monad (MonadPlus(..), ap)
-- Hide a few names that are provided by Applicative.
import Text.ParserCombinators.Parsec hiding (many, optional, (<|>))

-- The Applicative instance for every Monad looks like this.
```

(continues on next page)

(continued from previous page)

```
instance Applicative (GenParser s a) where
    pure  = return
    (<*>) = ap

-- The Alternative instance for every MonadPlus looks like this.
instance Alternative (GenParser s a) where
    empty = mzero
    (<|>) = mplus
```

为了方便起见，我们自己的模块导出了我们从 `Applicative` 和 `Parsec` 模块中导入的所有变量与函数名。因为我们隐藏了 Parsec 的 `(<|>)`，我们导入这个自己定义的模块后，使用的 `(<|>)` 将会是从 `Control.Applicative` 模块中导入的。

16.10 举例：使用 Applicative 进行 parse

我们将自底向上的改写上面的表单 parser，首先从 `p_hex` 开始，`p_hex` 解析一个 16 进制转义字符序列。下面是使用 do-notation 风格的代码：

```
-- file: ch16/FormApp.hs
p_hex :: CharParser () Char
p_hex = do
  char '%'
  a <- hexDigit
  b <- hexDigit
  let ((d, _):_) = readHex [a,b]
  return . toEnum $ d
```

而下面是 applicative 风格的代码：

```
-- file: ch16/FormApp.hs
a_hex = hexify <$> (char '%' *> hexDigit) <*> hexDigit
    where hexify a b = toEnum . fst . head . readHex $ [a,b]
```

虽然单独的 parser 并没有改变，仍然是 `char '%'` 与两个 `hexDigit`，把它们组合在一起的组合子却发生了变化。其中，目前我们唯一熟悉的一个就是 `(<$>)`，我们已经知道，它不过是 `fmap` 的同义词。

从我们对 `GenParser` 的 `Applicative` 实例的实现中，我们知道 `(<*>)` 就是 `ap`

剩下的我们不熟悉的组合子是 `(*>)`，它接受两个 parser 作为参数，首先应用第一个 parser，但是忽略其返回结果，而只用作消耗输入，然后应用第二个 parser，并返回其结果。换句话说，它很像 `(>>)`。

> 关于尖括号的一个小提示（此处应该是 Real World Haskell 中的 Notes）

> 我们继续之前，记住这些从 `Control.Applicative` 中导入的尖括号表示的组合子是在干什么是很有用的：如果一个尖括号指向某个方向，那么它就是返回这个方向的参数的结果。
>
> 例如，`(*>)` 返回其右侧参数的结果；`(<*>)` 返回两侧参数的结果，`(<*)`，这个组合子我们目前还没过用到，它返回其左侧参数的结果。

虽然这里涉及的多数概念在之前 Functor 和 Monad 的章节中我们已经了解过了，我们还是过一遍这下函数来解释下发生了什么。首先，为了解函数的类型，我们把 `hexify` 函数提升为全局函数，并且手动写类型签名。

```
-- file: ch16/FormApp.hs
hexify :: Char -> Char -> Char
hexify a b = toEnum . fst . head . readHex $ [a,b]
```

Parsec 的 `hexDigit` parser 会解析一个十六进制数字（译注：是 0-F 的数字，而不是十六进制数）

```
ghci> :type hexDigit
hexDigit :: CharParser st Char
```

因此，`char '%' *> hexDigit` 的类型跟 `hexDigit` 相同，而 `(*>)` 返回它右侧的结果。(`CharParser` 类型不过是 `GenParser Char` 的同义词)。

```
ghci> :type char '%' *> hexDigit
char '%' *> hexDigit :: GenParser Char st Char
```

`hexify <$> (char '%' *> hexDigit)` 这个表达式是这样一个 parser，它匹配一个 “%” 字符，紧接着匹配一个十六进制数字字符，而其结果是一个函数。(译注：, hexify 这个函数在这里被部分应用了)

```
ghci> :type hexify <$> (char '%' *> hexDigit)
hexify <$> (char '%' *> hexDigit) :: GenParser Char st (Char -> Char)
```

最后，`(<*>)` 首先应用左边的 parser，再应用右边的 parser，然后应用把右边 parser 产生的值应用到左边 parser 产生的函数上。

如果你已经能够理解下面这句话，那么你就能理解 `(<*>)` 和 `ap` 这两个组合子：`(<*>)` 就是原来的 `($)` 被提升到 Applicative Functor，而 `ap` 则是 `($)` 被提升到 Monad。

```
ghci> :type ($)
($) :: (a -> b) -> a -> b
ghci> :type (<*>)
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

接下来，我们考虑 `p_char` 这个 parser，原来的代码是这样子的：

```haskell
-- file: ch16/FormApp.hs
p_char :: CharParser () Char
p_char = oneOf urlBaseChars
     <|> (char '+' >> return ' ')
     <|> p_hex

urlBaseChars = ['a'..'z']++['A'..'Z']++['0'..'9']++"$-_.!*'(),"
```

使用 Applicative 风格的代码跟上面的代码几乎一样，不过使用了更方便的记号。

```haskell
-- file: ch16/FormApp.hs
a_char = oneOf urlBaseChars
     <|> (' ' <$ char '+')
     <|> a_hex
```

这里，(<$) 组合子会在右边的 parser 成功时，返回左边参数的值。

最后，等价的 p_pair_app1 也几乎跟原来的版本相同，下面是原来的版本：

```haskell
-- file: ch16/FormParse.hs
p_pair_app1 =
    liftM2 (,) (many1 p_char) (optionMaybe (char '=' >> many p_char))
```

我们改变的只有用来做提升的组合子：liftA 函数在这里的效果同 liftM 是一样的。

```haskell
-- file: ch16/FormApp.hs
a_pair :: CharParser () (String, Maybe String)
a_pair = liftA2 (,) (many1 a_char) (optionMaybe (char '=' *> many a_char))
```

16.11 Parse JSON 数据

为了更好的理解 Applicative Functor，并且进一步探索 Parsec，让我们来写一个满足 RFC 4627 定义的 JSON parser

在顶层，一个 JSON 值要么是一个对象，要么是一个数组。

```haskell
-- file: ch16/JSONParsec.hs
p_text :: CharParser () JValue
p_text = spaces *> text
     <?> "JSON text"
    where text = JObject <$> p_object
             <|> JArray <$> p_array
```

译注：这一节作者并没有给出 JSON 类型的定义，可以参考第六章。而且第六章的 JSON 定义也跟这里的 parser 不太一致，可以参考 Real World Haskell 网站这一节中 Alexey 的 comment:

```
-- 译注：Real World Haskell 网站这一节中 Alexey 的 comment
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject (JObj JValue)
            | JArray (JAry JValue)
            deriving (Eq, Ord, Show)
newtype JAry a = JAry {
    fromJAry :: [a]
    } deriving (Eq, Ord, Show)

newtype JObj a = JObj {
    fromJObj :: [(String, a)]
    } deriving (Eq, Ord, Show)
```

数组和对象在结构上很类似，一个字符（对数组是“[”，对对象是“{”）用作做括号，内部是用逗号分隔的数据，由另一个字符（对数组是“]”，对对象是“}”）作为右括号终结。我们可以抓住这种相似性，写一个小的辅助函数。

```
-- file: ch16/JSONParsec.hs
p_series :: Char -> CharParser () a -> Char -> CharParser () [a]
p_series left parser right =
    between (char left <* spaces) (char right) $
            (parser <* spaces) `sepBy` (char ',' <* spaces)
```

在这里，我们终于用到了 (<*) 这个我们之前介绍过的组合子。我们用它来略过一些 token 之前的空格。使用 p_series 函数，解析一个数组会很简单。

```
-- file: ch16/JSONParsec.hs
p_array :: CharParser () (JAry JValue)
p_array = JAry <$> p_series '[' p_value ']'
```

处理 JSON 的对象要复杂一点，需要一点额外的努力来为每个 object 的 field 产生一个 name-value 对。

```
-- file: ch16/JSONParsec.hs
p_object :: CharParser () (JObj JValue)
p_object = JObj <$> p_series '{' p_field '}'
    where p_field = (,) <$> (p_string <* char ':' <* spaces) <*> p_value
```

解析一个单独的值是就是调用一个现有的 Parser，然后把它的结果用相应的 JValue 构造器封装：

```haskell
-- file: ch16/JSONParsec.hs
p_value :: CharParser () JValue
p_value = value <* spaces
  where value = JString <$> p_string
            <|> JNumber <$> p_number
            <|> JObject <$> p_object
            <|> JArray  <$> p_array
            <|> JBool   <$> p_bool
            <|> JNull   <$ string "null"
            <?> "JSON value"

p_bool :: CharParser () Bool
p_bool = True <$ string "true"
     <|> False <$ string "false"
```

choice 组合子允许我们把这种很有很多选项的情况用一个列表来表示，它返回 parser 列表中第一个 parse 成功的 parser 的结果。

```haskell
-- file: ch16/JSONParsec.hs
p_value_choice = value <* spaces
  where value = choice [ JString <$> p_string
                       , JNumber <$> p_number
                       , JObject <$> p_object
                       , JArray  <$> p_array
                       , JBool   <$> p_bool
                       , JNull   <$ string "null"
                       ]
                <?> "JSON value"
```

下面是最有意思的两个 parser：数字、字符串

```haskell
-- file: ch16/JSONParsec.hs
p_number :: CharParser () Double
p_number = do s <- getInput
              case readSigned readFloat s of
                [(n, s')] -> n <$ setInput s'
                _         -> empty
```

我们的诀窍是利用 Haskell 标准库中的数字 parser 库函数，它们定义在 Numeric 库中，readFloat 函数解析一个无符号浮点数，而 readSigned 函数接受一个无符号数的 parser 作为参数，并将其转换为有符号数的 parser。

上面的那些函数都不是 Parsec 中的库函数，所以需要一点特殊处理。Parsec 的 getInput 函数可以让我们直接访问 Parsec 还不曾消耗的输入流，对这些输入流，如果 readSigned readFloat 解析成功，那么就返回解析成功的数字以及剩下的输入。这些还没有处理的输入，我们用 setInput 将他们还给 Parsec 作为新

的未消耗的输入流。

Parse 一个字符串也不困难，不过需要处理一些细节。

```
-- file: ch16/JSONParsec.hs
p_string :: CharParser () String
p_string = between (char '\"') (char '\"') (many jchar)
    where jchar = char '\\' *> (p_escape <|> p_unicode)
              <|> satisfy (`notElem` "\"\\")
```

我们可以使用刚刚介绍过的 choice 组合子来解析转义字符序列。

```
-- file: ch16/JSONParsec.hs
p_escape :: CharParser () Char
p_escape = choice (zipWith decode "bnfrt\\\"/" "\b\n\f\r\t\\\"/")
    where decode :: Char -> Char -> CharParser () Char
          decode c r = r <$ char c
```

最后，JSON 允许我们在字符串中使用 Unicode 字符：”\u” 后面跟着四个十六进制数字:

```
-- file: ch16/JSONParsec.hs
p_unicode :: CharParser () Char
p_unicode = char 'u' *> (decode <$> count 4 hexDigit)
    where decode x = toEnum code
              where ((code,_):_) = readHex x
```

相比 Monad，Applicative Functor 唯一缺少的能力，就是把一个值绑定到一个变量，而当我们需要验证我们解析的结果时，我们就需要这种能力。

基本上只有当我们需要把值绑定到变量时，我们才会需要写 Monadic 的函数，对于更复杂的 parser 也是这样的：我们不太会用到 Monad 提供的额外的力量。

我们写这本书的时候，Applicative Functor 对于 Haskell 社区还是很新的概念，人们仍然在探索它在 parser 领域之外应用的可能。

16.12 Parse HTTP 请求

这一节我们来写一个基本的 HTTP 请求的 parser，来作为 Applicative Parsing 的例子。

```
module HttpRequestParser
    (
      HttpRequest(..)
    , Method(..)
    , p_request
```

(continues on next page)

(continued from previous page)

```
    , p_query
    ) where

import ApplicativeParsec
import Numeric (readHex)
import Control.Monad (liftM4)
import System.IO (Handle)
```

一个 HTTP 请求包含一个 method，一个 identifier，一些 header，以及一个可选的 body。为了简单起见，我们只关注 HTTP 1.1 标准的六种 method 中的两种，`POST` method 包含一个 body，`GET` method 没有 body.

```
-- file: ch16/HttpRequestParser.hs
data Method = Get | Post
          deriving (Eq, Ord, Show)

data HttpRequest = HttpRequest {
      reqMethod :: Method
    , reqURL :: String
    , reqHeaders :: [(String, String)]
    , reqBody :: Maybe String
    } deriving (Eq, Show)
```

因为我们采用 application style, 我们的 parser 简洁而易读。当然，可读性好，是说你得习惯 applicative style。

```
-- file: ch16/HttpRequestParser.hs
p_request :: CharParser () HttpRequest
p_request = q "GET" Get (pure Nothing)
        <|> q "POST" Post (Just <$> many anyChar)
  where q name ctor body = liftM4 HttpRequest req url p_headers body
            where req = ctor <$ string name <* char ' '
        url = optional (char '/') *>
              manyTill notEOL (try $ string " HTTP/1." <* oneOf "01")
              <* crlf
```

简单地说，`q` 辅助函数接受一个 method 名，一个值构造器，一个对请求的可选 body 的 parser。而 `url` 辅助函数并不试图去验证一个 URL，因为 HTTP 规范没有规定 URL 能够包含哪些字符，这个函数只是消耗遇到的输入直到行尾或者遇到 HTTP 版本 identifier。

16.12.1 避免使用回溯

`try` 组合子必须记住它遇到的输入，因为要在 parse 失败时恢复消耗的输入，以便下一个 parser 使用。这被称为回溯。因为 `try` 必须保存输入，它的开销很昂贵。滥用 `try` 会拖慢 parser 的速度，甚至使性能慢到不可接受。

为了避免使用回溯，标准的做法是重构我们的 parser，手动提取（<|>）两侧 parser 的公共左因子，使我们只用一个 token 就能判断 parse 成功还是失败。在这种情况下，两个 parser 消耗相同的初始输入，最终组合为一个 parser

```
ghci> let parser = (++) <$> string "HT" <*> (string "TP" <|> string "ML")
ghci> parseTest parser "HTTP"
"HTTP"
ghci> parseTest parser "HTML"
"HTML"
```

更妙的是，使用这种写法，当输入无法匹配时，Parsec 给出的错误信息更好：

```
ghci> parseTest parser "HTXY"
parse error at (line 1, column 3):
unexpected "X"
expecting "TP" or "ML"
```

16.12.2 Parse HTTP Header

HTTP 请求的第一行之后，是零到多个 header，一个 header 以一个字段名开头，跟着是一个冒号，然后是内容。如果一行以空格开头，它被认为是上一行的延续。

```
-- file: ch16/HttpRequestParser.hs
p_headers :: CharParser st [(String, String)]
p_headers = header `manyTill` crlf
  where header = liftA2 (,) fieldName (char ':' *> spaces *> contents)
        contents = liftA2 (++) (many1 notEOL <* crlf)
                               (continuation <|> pure [])
        continuation = liftA2 (:) (' ' <$ many1 (oneOf " \t")) contents
        fieldName = (:) <$> letter <*> many fieldChar
        fieldChar = letter <|> digit <|> oneOf "-_"

crlf :: CharParser st ()
crlf = (() <$ string "\r\n") <|> (() <$ newline)

notEOL :: CharParser st Char
notEOL = noneOf "\r\n"
```

16.13 练习

1. 我们的 HTTP 请求 parser 过于简化了，没法在部署在实际应用中。它缺少重要的功能，并且无法组织最基本的拒绝服务攻击 (DOS, denial of service attack) 让我们的 parser 关注 Content-Length 这个 field, 如果

它存在的话

2. 针对不设防的 web server 的一个很流行的 DOS 攻击方式，是向它发送特别长的 header，一个 header 可能包含几百兆的垃圾信息，从而耗光服务器的内存。重构 header 的 parser，当一行超过 4096 个字符时 parse 失败。它必须在超过长度时立刻失败，而不是等到处理完一行之后。

3. 关注 Transfer-Encoding 这个 field，如果它存在的话，关于它的细节，可以查看 RFC 2616 的第 3.6.1 节

4. 另一个流行的攻击方式是开启一个链接之后，放置不管或者以十分慢的速度发送数据。使用 `IO` monad 来封装 parser，如果没有在 30 秒内完成 parse，就使用 `System.Timeout` 这个模块关闭链接。

CHAPTER

SEVENTEEN

第 18 章：MONAD 变换器

17.1 动机：避免样板代码

Monad 提供了一种强大途径以构建带效果的计算。虽然各个标准 monad 皆专一于其特定的任务，但在实际代码中，我们常常想同时使用多种效果。

比如，回忆在第十章中开发的 `Parse` 类型。在介绍 monad 之时，我们提到这个类型其实是乔装过的 `State` monad。事实上我们的 monad 比标准的 `State` monad 更加复杂：它同时也使用了 `Either` 类型来表达解析过程中可能的失败。在这个例子中，我们想在解析失败的时候就立刻停止这个过程，而不是以错误的状态继续执行解析。这个 monad 同时包含了带状态计算的效果和提早退出计算的效果。

普通的 `State` monad 不允许我们提早退出，因为其只负责状态的携带。其使用的是 `fail` 函数的默认实现：直接调用 `error` 抛出异常 - 这一异常无法在纯函数式的代码中捕获。因此，尽管 `State` monad 似乎允许错误，但是这一能力并没有什么用。（再次强调：请尽量避免使用 `fail` 函数!）

理想情况下，我们希望能使用标准的 `State` monad，并为其加上实用的错误处理能力以代替手动地大量定制各种 monad。虽然在 `mtl` 库中的标准 monad 不可合并使用，但使用库中提供了一系列的 *monad* 变换器可以达到相同的效果。

Monad 变换器和常规的 monad 很类似，但它们并不是独立的实体。相反，monad 变换器通过修改其以为基础的 monad 的行为来工作。大部分 `mtl` 库中的 monad 都有对应的变换器。习惯上变换器以其等价的 monad 名为基础，加以 `T` 结尾。例如，与 `State` 等价的变换器版本称作 `StateT`；它修改下层 monad 以增加可变状态。此外，若将 `WriterT` monad 变换器叠加于其他（或许不支持数据输出的）monad 之上，在被 monad 修改后的的 monad 中，输出数据将成为可能。

[注：`mtl` 意为 monad 变换器函数库 (Monad Transformer Library)]

[译注：Monad 变换器需要依附在一已有 monad 上来构成新的 monad，在接下来的行文中将使用 “下层 monad” 来称呼 monad 变换器所依附的那个 monad]

17.2 简单的 Monad 变换器实例

在介绍 monad 变换器之前，先看看以下函数，其中使用的都是之前接触过的技术。这个函数递归地访问目录树，并返回一个列表，列表中包含树的每层的实体个数：

```
-- file: ch18/CountEntries.hs
module CountEntries
  ( listDirectory
  , countEntriesTrad
  ) where

import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
import Control.Monad (forM, liftM)

listDirectory :: FilePath -> IO [String]
listDirectory = liftM (filter notDots) . getDirectoryContents
  where notDots p = p /= "." && p /= ".."

countEntriesTrad :: FilePath -> IO [(FilePath, Int)]
countEntriesTrad path = do
    contents <- listDirectory path
    rest <- forM contents $ \name -> do
        let newName = path </> name
        isDir <- doesDirectoryExist newName
        if isDir
          then countEntriesTrad newName
          else return []
    return $ (path, length contents) : concat rest
```

现在看看如何使用 `Writer` monad 实现相同的目标。由于这个 monad 允许随时记下数值，所以并不需要我们显示地去构建结果。

为了遍历目录，这个函数必须在 `IO` monad 中执行，因此我们无法直接使用 `Writer` monad。但我们可以用 `WriterT` 将记录信息的能力赋予 `IO` 。一种简单的理解方法是首先理解涉及的类型。

通常 `Writer` monad 有两个类型参数，因此写作 `Writer w a` 更为恰当。其中参数 `w` 用以指明我们想要记录的数值的类型。而另一类型参数 `a` 是 monad 类型类所要求的。因此 `Writer [(FilePath, Int)] a` 是个记录一列目录名和目录大小的 writer monad。

`WriterT` 变换器有着类似的结构。但其增加了另外一个类型参数 `m`： 这便是下层 monad，也是我们想为其增加功能的 monad。`WriterT` 的完整类型签名是 `Writer w m a`。

由于所需的目录遍历操作需要访问 `IO` monad，因此我们将 writer 功能累加在 `IO` monad 之上。通过将 monad 变换器与原有 monad 结合，我们得到了类型签名：`WriterT [(FilePath, Int)] IO a` 这个 monad 变换

器和 monad 的组合自身也是一个 monad：

```
-- file: ch18/CountEntriesT.hs
module CountEntriesT
  ( listDirectory
  , countEntries
  ) where

import CountEntries (listDirectory)
import System.Directory (doesDirectoryExist)
import System.FilePath ((</>))
import Control.Monad (forM_, when)
import Control.Monad.Trans (liftIO)
import Control.Monad.Writer (WriterT, tell)

countEntries :: FilePath -> WriterT [(FilePath, Int)] IO ()
countEntries path = do
    contents <- liftIO . listDirectory $ path
    tell [(path, length contents)]
    forM_ contents $ \name -> do
        let newName = path </> name
        isDir <- liftIO . doesDirectoryExist $ newName
        when isDir $ countEntries newName
```

代码与其先前的版本区别不大，需要时 `liftIO` 可以将 `IO` monad 暴露出来；同时，`tell` 可以用以记下对目录的访问。

为了执行这一代码，需要选择一个 `WriterT` 的执行函数：

```
ghci> :type runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :type execWriterT
execWriterT :: Monad m => WriterT w m a -> m w
```

这些函数都可以用以执行动作，移除 `WriterT` 的包装，并将结果交给其下层 monad。其中 `runWriterT` 函数同时返回动作结果以及在执行过程获得的记录。而 `execWriterT` 丢弃动作的结果，只将记录返回。

因为没有 `IOT` 这样的 monad 变换器，所以此处我们在 `IO` 之上使用 `WriterT` 。一旦要用 `IO` monad 和其他的一个或多个 monad 变换器结合，`IO` 一定在 monad 栈的最底下。

[译注：“monad 栈” 由 monad 和一个或多个 monad 变换器叠加而成，形成一个栈的结构。若在 monad 栈中需要 `IO` monad，由于没有对应的 monad 变换器（`IOT` ），所以 `IO` monad 只能位于整个 monad 栈的最底下。此外，`IO` 是一个很特殊的 monad，它的 `IOT` 版本是无法实现的。]

17.3 Monad 和 Monad 变换器中的模式

在 mtl 库中的大部分 monad 与 monad 变换器遵从一些关于命名和类型类的模式。

为说明这些规则，我们将注意力聚焦在一个简单的 monad 上：reader monad。reader monad 的具体 API 位于 MonadReader 中。大部分 mtl 中的 monad 都有一个名称相对的类型类。例如 MonadWriter 定义了 writer monad 的 API，以此类推。

```
-- file: ch18/Reader.hs
{-# LANGUAGE FunctionalDependencies #-}
class Monad m => MonadReader r m | m -> r where
    ask :: m r
    local :: (r -> r) -> m a -> m a
```

其中类型变量 r 表示 reader monad 所附带的不变状态，Reader r monad 是个 MonadReader 的实例，同时 ReaderT r m monad 变换器也是一个。这个模式同样也在其他的 mtl monad 中重复着：通常有个具体的 monad，和其对应的 monad 变换器，而它们都是相应命令的类型类的实例。这个类型类定义了功能相同的 monad 的 API。

回到我们 reader monad 的例子中，我们之前尚未讨论过 local 函数。通过一个类型为 r -> r 的函数，它可临时修改当前的环境，并在这一临时环境中执行其动作。举个具体的例子：

```
-- file: ch18/LocalReader.hs
import Control.Monad.Reader

myName step = do
    name <- ask
    return (step ++ ", I am " ++ name)

localExample :: Reader String (String, String, String)
localExample = do
    a <- myName "First"
    b <- local (++"dy") (myName "Second")
    c <- myName "Third"
    return (a,b,c)
```

若在 ghci 中执行 localExample，可以观察到对环境修改的效果被限制在了一个地方：

```
ghci> runReader localExample "Fred"
Loading package mtl-1.1.0.1 ... linking ... done.
("First, I am Fred","Second, I am Freddy","Third, I am Fred")
```

当下层 monad m 是一个 MonadIO 的实例时，mtl 提供了关于 ReaderT r m 和其他类型类的实例，这里是其中的一些：

```haskell
-- file: ch18/Reader.hs
instance (Monad m) => Functor (ReaderT r m) where
    ...

instance (MonadIO m) => MonadIO (ReaderT r m) where
    ...

instance (MonadPlus m) => MonadPlus (ReaderT r m) where
    ...
```

再次说明：为方便使用，大部分的 `mtl` monad 变换器都定义了诸如此类的实例。

17.4 叠加多个 Monad 变换器

之前提到过，在常规 monad 上叠加 monad 变换器可得到另一个 monad。由于混合的结果也是个 monad，我们可以凭此为基础再叠加上一层 monad 变换器。事实上，这么做十分常见。但在什么情况下才需要创建这样的 monad 呢?

- 若代码想和外界打交道，便需要 `IO` 作为这个 monad 栈的基础。否则普通的 monad 便可以满足需求。
- 加上一层 `ReaderT` ，以添加访问只读配置信息的能力。
- 叠加上 `StateT` ，就可以添加可修改的全局状态。
- 若想得到记录事件的能力，可以添加一层 `WriterT` 。

这个做法的强大之处在于：我们可以指定所需的计算效果，以量身定制 monad 栈。

举个多重叠加的 moand 变换器的例子，这里是之前开发的 `countEntries` 函数。我们想限制其递归的深度，并记录下它在执行过程中所到达的最大深度：

```haskell
-- file: ch18/UglyStack.hs
import System.Directory
import System.FilePath
import System.Monad.Reader
import System.Monad.State

data AppConfig = AppConfig
  { cfgMaxDepth :: Int
  } deriving (Show)

data AppState = AppState
  { stDeepestReached :: Int
  } deriving (Show)
```

此处使用 ReaderT 来记录配置数据，数据的内容表示最大允许的递归深度。同时也使用了 StateT 来记录在实际遍历过程中所达到的最大深度。

```
-- file: ch18/UglyStack.hs
type App = ReaderT AppConfig (StateT AppState IO)
```

我们的变换器以 IO 为基础，依次叠加 StateT 与 ReaderT 。在此例中，栈顶是 ReaderT 还是 WriterT 并不重要，但是 IO 必须作为最下层 monad。

仅仅几个 monad 变换器的叠加，也会使类型签名迅速变得复杂起来。故此处以 type 关键字定义类型别名，以简化类型的书写。

17.4.1 缺失的类型参数呢？

或许你已注意到，此处的类型别名并没有我们为 monad 类型所常添加的类型参数 a：

```
-- file: ch18/UglyStack.hs
type App2 a = ReaderT AppConfig (StateT AppState IO) a
```

在常规的类型签名用例下，App 和 App2 不会遇到问题。但如果想以此类型为基础构建其他类型，两者的区别就显现出来了。

例如我们想另加一层 monad 变换器，编译器会允许 WriterT [String] App a 但拒绝 WriterT [String] App2 a 。

其中的理由是：Haskell 不允许对类型别名的部分应用。App 不需要类型参数，故没有问题。另一方面，因为 App2 需要一个类型参数，若想基于 App2 构造其他的类型，则必须为这个类型参数提供一个类型。

这一限制仅适用于类型别名，当构建 monad 栈时，通常的做法是用 newtype 来封装（接下来的部分就会看到这类例子）。因此实际应用中很少出现这种问题。

[译注：类似于函数的部分应用，“类型别名的部分应用”指的是在应用类型别名时，给出的参数数量少于定义中的参数数量。在以上例子中，App 是一个完整的应用，因为在其定义 type App = ... 中，没有类型参数；而 App2 却是个部分应用，因为在其定义 type App2 a = ... 中，还需要一个类型参数 a 。]

我们 monad 栈的执行函数很简单：

```
-- file: ch18/UglyStack.hs
runApp :: App a -> Int -> IO (a, AppState)
runApp k maxDepth =
    let config = AppConfig maxDepth
        state = AppState 0
    in runStateT (runReaderT k config) state
```

对 runReaderT 的应用移除了 ReaderT 变换器的包装，之后 runStateT 移除了 StateT 的包装，最后的结果便留在 IO monad 中。

和先前的版本相比，我们的修改并未使代码复杂太多，但现在函数却能记录目前的路径，和达到的最大深度：

```
constrainedCount :: Int -> FilePath -> App [(FilePath, Int)]
constrainedCount curDepth path = do
    contents <- liftIO . listDirectory $ path
    cfg <- ask
    rest <- forM contents $ \name -> do
        let newPath = path </> name
        isDir <- liftIO $ doesDirectoryExist newPath
        if isDir && curDepth < cfgMaxDepth cfg
          then do
            let newDepth = curDepth + 1
            st <- get
            when (stDeepestReached st < newDepth) $
              put st {stDeepestReached = newDepth}
            constrainedCount newDepth newPath
          else return []
    return $ (path, length contents) : concat rest
```

在这个例子中如此运用 monad 变换器确实有些小题大做，因为这仅仅是个简单函数，其并没有因此得到太多的好处。但是这个方法的实用性在于，可以将其 *轻易扩展以解决更加复杂的问题*。

大部分指令式的应用可以使用和这里的 App monad 类似的方法，在 monad 栈中编写。在实际的程序中，或许需要携带更复杂的配置数据，但依旧可以使用 ReaderT 以保持其只读，并只在需要时暴露配置；或许有更多可变状态需要管理，但依旧可以使用 StateT 封装它们。

17.4.2 隐藏细节

使用常规的 newtype 技术，便可将细节与接口分离开：

```
newtype MyApp a = MyA
  { runA :: ReaderT AppConfig (StateT AppState IO) a
  } deriving (Monad, MonadIO, MonadReader AppConfig,
              MonadState AppState)

runMyApp :: MyApp a -> Int -> IO (a, AppState)
runMyApp k maxDepth =
    let config = AppConfig maxDepth
        state = AppState 0
    in runStateT (runReaderT (runA k) config) state
```

若只导出 MyApp 类构造器和 runMyApp 执行函数，客户端的代码就无法知晓这个 monad 的内部结构是否是

monad 栈了。

此处，庞大的 deriving 子句需要 GeneralizedNewtypeDeriving 语言编译选项。编译器可以为我们生成这些实例，这看似十分神奇，究竟是如何做到的呢?

早先，我们提到 mtl 库为每个 monad 变换器都提供了一系列实例。例如 IO monad 实现了 MonadIO ，若下层 monad 是 MonadIO 的实例，那么 mtl 也将为其对应的 StateT 构建一个 MonadIO 的实例，类似的事情也发生在 ReaderT 上。

因此，这其中并无太多神奇之处：位于 monad 栈顶层的 monad 变换器，已是所有我们声明的 deriving 子句中的类型类的实例，我们做的只不过是重新派生这些实例。这是 mtl 精心设计的一系列类型类和实例完美配合的结果。除了基于 newtype 声明的常规的自动推导以外并没有发生什么。

[译注：注意到此处 newtype MyApp a 只是乔装过的 ReaderT AppConfig (StateT AppState IO) a 。因此我们可以列出 MyApp a 这个 monad 栈的全貌（自顶向下）：

- ReaderT AppConfig （monad 变换器）
- StateT AppState （monad 变换器）
- IO （monad）

注意这个 monad 栈和 deriving 子句中类型类的相似度。这些实例都可以自动派生：MonadIO 实例自底层派生上来，MonadStateT 从中间一层派生，而 MonadReader 实例来自顶层。所以虽然 newtype MyApp a 引入了一个全新的类型，其实例是可以通过内部结构自动推导的。]

17.4.3 练习

1. 修改 App 类型别名以交换 ReaderT 和 StateT 的位置，这一变换对执行函数 runApp 会带来什么影响?
2. 为 App monad 栈添加 WriterT 变换器。相应地修改 runApp 。
3. 重写 contrainedCount 函数，在为 App 新添加的 WriterT 中记录结果。

[译注：第一题中的 StateT 原为 WriterT ，鉴于 App 定义中并无 WriterT ，此处应该指的是 StateT]

17.5 深入 Monad 栈中

至今，我们了解了对 monad 变换器的简单运用。对 mtl 库的便利组合拼接使我们免于了解 monad 栈构造的细节。我们确实已掌握了足以帮助我们简化大量常见编程任务的 monad 变换器相关知识。

但有时，为了实现一些实用的功能，还是我们需要了解 mtl 库并不便利的一面。这些任务可能是将定制的 monad 置于 monad 栈底，也可能是将定制的 monad 变换器置于 monad 变换器栈中的某处。为了解其中潜在的难度，我们讨论以下例子。

假设我们有个定制的 monad 变换器 CustomT：

```
-- file: ch18/CustomT.hs
newtype CustomT m a = ...
```

在 mtl 提供的框架中，每个位于栈上的 monad 变换器都将其下层 monad 的 API 暴露出来。这是通过提供大量的类型类实例来实现的。遵从这一模式的规则，我们也可以实现一系列的样板实例：

```
-- file: ch18/CustomT.hs
instance MonadReader r m => MonadReader r (CustomT m) where
    ...

instance MonadIO m => MonadIO (CustomT m) where
    ...
```

若下层 monad 是 MonadReader 的实例，则 CustomT 也可作为 MonadReader 的实例：实例化的方法是将所有相关的 API 调用转接给其下层实例的相应函数。经过实例化之后，上层的代码就可以将 monad 栈作为一个整体，当作 MonadReader 的实例，而不再需要了解或关心到底是其中的哪一层提供了具体的实现。

不同于这种依赖类型类实例的方法，我们也可以显式指定想要使用的 API。MonadTrans 类型类定义了一个实用的函数 lift：

```
ghci> :m +Control.Monad.Trans
ghci> :info MonadTrans
class MonadTrans t where lift :: (Monad m) => m a -> t m a
      -- Defined in Control.Monad.Trans
```

这个函数接受来自 monad 栈中，当前栈下一层的 monad 动作，并将这个动作变成，或者说是 抬举到现在的 monad 变换器中。每个 monad 变换器都是 MonadTrans 的实例。

lift 这个名字是基于此函数与 fmap 和 liftM 目的上的相似度的。这些函数都可以从类型系统的下一层中把东西提升到我们目前工作的这一层。它们的区别是：

fmap 将纯函数提升到 functor 层次

liftM 将纯函数提升到 monad 层次

lift 将一 monad 动作，从 monad 栈中的下一层提升到本层

[译注：实际上 liftM 间接调用了 fmap，两个函数在效果上是完全一样的。译者认为，当操作对象是 monad（所有的 monad 都是 functor）的时候，使用其中的哪一个只是思考方法上的不同。]

现在重新考虑我们在早些时候定义的 App monad 栈 (之前我们将其包装在 newtype 中)：

```
-- file: ch18/UglyStack.hs
type App = ReaderT AppConfig (StateT AppState IO)
```

若想访问 StateT 所携带的 AppState，通常需要依赖 mtl 的类型类实例来为我们处理组合工作：

```haskell
-- file: ch18/UglyStack.hs
implicitGet :: App AppState
implicitGet = get
```

通过将 get 函数从 StateT 中抬举进 ReaderT ， lift 函数也可以实现同样的效果：

```haskell
-- file: ch18/UglyStack.hs
explicitGet :: App AppState
explicitGet = lift get
```

显然当 mtl 可以为我们完成这一工作时，代码会变得更清晰。但是 mtl 并不总能完成这类工作。

17.5.1 何时需要显式的抬举？

我们必须使用 lift 的一个例子是：当在一个 monad 栈中，同一个类型类的实例出现了多次时：

```haskell
-- file: ch18/StackStack.hs
type Foo = StateT Int (State String)
```

若此时我们试着使用 MonadState 类型类中的 put 动作，得到的实例将是 StateT Int ，因为这个实例在 monad 栈顶。

```haskell
-- file: ch18/StackStack.hs
outerPut :: Int -> Foo ()
outerPut = put
```

在这个情况下，唯一能访问下层 State monad 的 put 函数的方法是使用 lift ：

```haskell
-- file: ch18/StackStack.hs
innerPut :: String -> Foo ()
innerPut = lift . put
```

有时我们需要访问多于一层以下的 monad，这时我们必须组合 lift 调用。每个函数组合中的 lift 将我们带到更深的一层。

```haskell
-- file: ch18/StackStack.hs
type Bar = ReaderT Bool Foo

barPut :: String -> Bar ()
barPut = lift . lift . put
```

正如以上代码所示，当需要用 lift 的时候，一个好习惯是定义并使用包裹函数来为我们完成抬举工作。因为这种在代码各处显式使用 lift 的方法使代码变得混乱。另一个显式 lift 的缺点在于，其硬编码了 monad 栈的层次细节，这将使日后对 monad 栈的修改变得复杂。

17.6 构建以理解 Monad 变换器

为了深入理解 monad 变换器通常是如何运作的，在本节我们将自己构建一个 monad 变换器，期间一并讨论其中的组织结构。我们的目标简单而实用：`MaybeT` 。但是 `mtl` 库意外地并没有提供它。

[译注：如果想使用现成的 `MaybeT` ，现在你可以在 Hackage 上的 `transformers` 库中找到它。]

这个 monad 变换器修改 monad 的方法是：将下层 monad `m a` 的类型参数包装在 `Maybe` 中，以得到类型 `m (Maybe a)` 。正如 `Maybe` monad 一样，若在 `MaybeT` monad 变换器中调用 `fail` ，则计算将提早结束执行。

为使 `m (Maybe a)` 成为 `Monad` 的实例，其必须有个独特的类型。这里我们通过 `newtype` 声明来实现：

```
-- file: ch18/MaybeT.hs
newtype MaybeT m a = MaybeT
  { runMaybeT :: m (Maybe a) }
```

现在需要定义三个标准的 monad 函数。其中最复杂的是 `(>>=)` ，它的实现也阐明了我们实际上在做什么。在开始研究其操作之前，不妨先看看其类型：

```
-- file: ch18/MaybeT.hs
bindMT :: (Monad m) => MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

为理解其类型签名，回顾之前在十五章中对“多参数类型类”讨论。此处我们想使 *部分类型* `MaybeT m` 成为 `Monad` 的实例。这个部分类型拥有通常的单一类型参数 `a` ，这样便能满足 `Monad` 类型类的要求。

[译注：`MaybeT` 的完整定义是 `MaybeT m a` ，因此 `MaybeT m` 只是部分应用。]

理解以下 `(>>=)` 实现的关键在于：`do` 代码块里的代码是在 *下层* monad 中执行的，无论这个下层 monad 是什么。

```
-- file: ch18/MaybeT.hs
x `bindMT` f = MaybeT $ do
    unwrapped <- runMaybeT x
    case unwrapped of
      Nothing -> return Nothing
      Just y -> runMaybeT (f y)
```

我们的 `runMaybeT` 函数解开了在 `x` 中包含的结果。进而，注意到 `<-` 符号是 `(>>=)` 的语法糖：monad 变换器必须使用其下层 monad 的 `(>>=)` 。而最后一部分对 `unwrapped` 的结构分析（`case` 表达式），决定了我们是要短路当前计算，还是将计算继续下去。最后，观察表达式的最外层。为了将下层 monad 再次藏起来，这里必须用 `MaybeT` 构造器包装结果。

刚才展示的 `do` 标记看起来更容易阅读，但是其将我们依赖下层 monad 的 `(>>=)` 函数的事实也藏了起来。下面提供一个更符合语言习惯的 `MaybeT` 的 `(>>=)` 实现：

```haskell
-- file: ch18/MaybeT.hs
x `altBindMT` f =
    MaybeT $ runMaybeT x >>= maybe (return Nothing) (runMaybeT . f)
```

现在我们了解了 (>>=) 在干些什么。关于 return 和 fail 无需太多解释，Monad 实例也不言自明：

```haskell
-- file: ch18/MaybeT.hs
returnMT :: (Monad m) => a -> MaybeT m a
returnMT a = MaybeT $ return (Just a)

failMT :: (Monad m) => t -> MaybeT m a
failMT _ = MaybeT $ return Nothing

instance (Monad m) => Monad (MaybeT m) where
  return = returnMT
  (>>=) = bindMT
  fail = failM
```

17.6.1 建立 Monad 变换器

为将我们的类型变成 monad 变换器，必须提供 MonadTrans 的实例，以使用户可以访问下层 monad：

```haskell
-- file: ch18/MaybeT.hs
instance MonadTrans MaybeT where
    lift m = MaybeT (Just `liftM` m)
```

下层 monad 以类型 a 开始：我们“注入”Just 构造器以使其变成需要的类型：Maybe a 。进而我们通过 MaybeT 藏起下层 monad。

17.6.2 更多的类型类实例

在定义好 MonadTrans 的实例后，便可用其来定义其他大量的 mtl 类型类实例了：

```haskell
-- file: ch18/MaybeT.hs
instance (MonadIO m) => MonadIO (MaybeT m) where
  liftIO m = lift (liftIO m)

instance (MonadState s m) => MonadState s (MaybeT m) where
  get = lift get
  put k = lift (put k)

-- ... 对 MonadReader, MonadWriter 等的实例定义同理 ...
```

由于一些 mtl 类型类使用了函数式依赖，有些实例的声明需要 GHC 大大放宽其原有的类型检查规则。(若我们忘记了其中任意的 LANGUAGE 指令，编译器会在其错误信息中提供建议。)

```
-- file: ch18/MaybeT.hs
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses,
             UndecidableInstances #-}
```

是花些时间来写这些样板实例呢，还是显式地使用 lift 呢？这取决于这个 monad 变换器的用途。如果我们只在几种有限的情况下使用它，那么只提供 MonadTrans 实例就够了。在这种情况下，也无妨提供一些依然有意义的实例，比如 MonadIO。另一方面，若我们需要在大量的情况下使用这一 monad 变换器，那么花些时间来完成这些实例或许也不错。

17.6.3 以 Monad 栈替代 Parse 类型

现在我们已开发了一个支持提早退出的 monad 变换器，可以用其来辅助开发了。例如，此处若想处理解析一半失败的情况，便可以用这一以我们的需求定制的 monad 变换器来替代我们在第十章“隐式状态”一节开发的 Parse 类型。

17.6.4 练习

1. 我们的 Parse monad 还不是之前版本的完美替代。因为其用的是 Maybe 而不是 Either 来代表结果。因此在失败时暂时无法提供任何有用的信息。

   构建一个 EitherT s（其中 s 是某个类型）来表示结果，并用其实现更适合的 Parse monad 以在解析失败时汇报具体错误信息。

或许在你探索 Haskell 库的途中，在 Control.Monad.Error 遇到过一个 Either 类型的 Monad 实例。我们建议不要参照它来完成你的实现，因为它的设计太局限了：虽然其将 Either String 变成一个 monad，但实际上把 Either 的第一个类型参数限定为 String 并非必要。

提示: 若你按照这条建议来做，你的定义中或许需要使用 FlexibleInstances 语言扩展。

17.7 注意变换器堆叠顺序

从早先使用 ReaderT 和 StateT 的例子中，你或许会认为叠加 monad 变换器的顺序并不重要。事实并非如此，考虑在 State 上叠加 StateT 的情况，或许会助于你更清晰地意识到：堆叠的顺序确实产生了结果上的区别：类型 StateT Int (State String) 和类型 StateT String (State Int) 或许携带的信息相同，但它们却无法互换使用。叠加的顺序决定了我们是否要用 lift 来取得状态中的某个部分。

下面的例子更加显著地阐明了顺序的重要性。假设有个可能失败的计算，而我们想记录下在什么情况下其会失败：

```haskell
-- file: ch18/MTComposition.hs
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Writer
import MaybeT

problem :: MonadWriter [String] m => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

那么这两个 monad 栈中的哪一个会带给我们需要的信息呢?

```haskell
type A = WriterT [String] Maybe

type B = MaybeT (Writer [String])

a :: A ()
a = problem

b :: B ()
b = problem
```

我们在 ghci 中试试看:

```
ghci> runWriterT a
Loading package mtl-1.1.0.1 ... linking ... done.
Nothing
ghci> runWriter $ runMaybeT b
(Nothing,["this is where i fail"])
```

看看执行函数的类型签名，其实结果并不意外:

```
ghci> :t runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :t runWriter . runMaybeT
runWriter . runMaybeT :: MaybeT (Writer w) a -> (Maybe a, w)
```

在 Maybe 上叠加 WriterT 的策略使 Maybe 成为下层 monad，因此 runWriterT 必须给我们以 Maybe 为类型的结果。在测试样例中，我们只会在不出现任何失败的情况下才能获得日志!

叠加 monad 变换器类似于组合函数: 如果我们改变函数应用的顺序，那么我们并不会对得到不同的结果感到意外。同样的道理也适用于对 monad 变换器的叠加。

17.8 纵观 Monad 与 Monad 变换器

本节，让我们暂别细节，讨论一下用 monad 和 monad 变换器编程的优缺点。

17.8.1 对纯代码的干涉

在实际编程中，使用 monad 的最恼人之处或许在于其阻碍了我们使用纯代码。很多实用的纯函数需要一个 monad 版的类似函数，而其 monad 版只是加上一个占位参数 `m` 供 monad 类型构造器填充：

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> :i filterM
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
      -- Defined in Control.Monad
```

然而，这种覆盖是有限的：标准库中并不总能提供纯函数的 monad 版本。

其中有一部分历史原因：Eugenio Moggi 于 1988 年引入了使用 monad 编程的思想。而当时 Haskell 1.0 标准尚在开发中。现今版本的 `Prelude` 中的大部分函数可以追溯到于 1990 发布的 Haskell 1.0。在 1991 年，Philip Wadler 开始为更多的函数式编程听众作文，阐述 monad 的潜力。从那时起，monad 开始用于实践。

直到 1996 年 Haskell 1.3 标准发布之时，monad 才得到了支持。但是在那时，语言的设计者已经受制于维护向前兼容性: 它们无法改变 `Prelude` 中的函数签名，因为那会破坏现有的代码。

从那以后，Haskell 社区学会了很多合适的抽象。因此我们可以写出不受这一纯函数 / monad 函数分裂影响的代码。你可以在 `Data.Traversable` 和 `Data.Foldable` 中找到这些思想的精华。

尽管它们极具吸引力，由于版面的限制。我们不会在本书中涵盖相关内容。但如果你能轻易理解本章内容，自行理解它们也不会有问题。

在理想世界里，我们是否会与过去断绝，并让 `Prelude` 包含 `Traversable` 和 `Foldable` 类型呢？或许不会，因为学习 Haskell 本身对新手来说已经是个相当刺激的历程了。在我们已经了解 functor 和 monad 之后，`Foldable` 和 `Traversable` 的抽象是十分容易理解的。但是对学习者来说这意味着摆在他们面前的是更多纯粹的抽象。若以教授语言为目的，`map` 操作的最好是列表，而不是 functor。

[译注：实际上，自 GHC 7.10 开始，`Foldable` 和 `Traversable` 已经进入了 `Prelude` 。一些函数的类型签名会变得更加抽象（以 GHC 7.10.1 为例）：

```
ghci-7.10.1> :t mapM
mapM :: (Monad m, Traversable t) => (a -> m b) -> t a -> m (t b)
ghci-7.10.1> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

这并不是一个对初学者友好的改动，但由于新的函数只是旧有函数的推广形式，使用旧的函数签名依旧可以通过类型检查：

```
ghci-7.10.1> :t (mapM :: Monad m => (a -> m b) -> [a] -> m [b])
(mapM :: Monad m => (a -> m b) -> [a] -> m [b])
  :: Monad m => (a -> m b) -> [a] -> m [b]
ghci-7.10.1> :t (foldl :: (b -> a -> b) -> b -> [a] -> b)
(foldl :: (b -> a -> b) -> b -> [a] -> b)
  :: (b -> a -> b) -> b -> [a] -> b
```

若在学习过程中遇到障碍，不妨暂且以旧的类型签名来理解它们。]

17.8.2 对次序的过度限定

我们使用 monad 的一个基本原因是：其允许我们指定效果发生的次序。再看看我们早先写的一小段代码：

```
-- file: ch18/MTComposition.hs
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Writer
import MaybeT

problem :: MonadWriter [String] m => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

因为我们在 monad 中执行，`tell` 的效果可以保证发生在 `fail` 之前。这里的问题在于，这个次序并不必要，但是我们却得到了这样的次序保证。编译器无法任意安排 monad 式代码的次序，即便这么做能使代码效率更高。

[译注：解释一下这里的“次序并不必要”。回顾之前对叠加次序问题的讨论：

```
type A = WriterT [String] Maybe

type B = MaybeT (Writer [String])

a :: A ()
a = problem
-- runWriterT a == Nothing

b :: B ()
b = problem
-- runWriter (runMaybeT b) == (Nothing, ["this is where i fail"])
```

下面把注意力集中于 `a`：注意到 `runWriterT a == Nothing`，`tell` 的结果并不需要，因为接下来的 `fail` 取消了计算，将之前的结果抛弃了。利用这个事实，可以得知让 `fail` 先执行效率更高。同时注意对

`fail` 和 `tell` 的实际处理来自 monad 栈的不同层，所以在一定限制下调换某些操作的顺序会不影响结果。但是由于这个 monad 栈本身也要是个 monad，使这种本来可以进行的交换变得不可能了。]

17.8.3 运行时开销

最后，当我们使用 monad 和 monad 变换器时，需要付出一些效率的代价。例如 `State` monad 携带状态并将其放在一个闭包中。在 Haskell 的实现中，闭包的开销或许廉价但绝非免费。

Monad 变换器把其自身的开销附加在了其下层 monad 之上。每次我们使用 (>>=) 时，MaybeT 变换器便需要包装和解包。而由 `ReaderT` ，`StateT` 和 `MaybeT` 依次叠加组成的 monad 栈，在每次使用 (>>=) 时，更是有一系列的簿记工作需要完成。

一个足够聪明的编译器或许可以将这些开销部分，甚至于全部消除。但是那种深度的复杂工作尚未广泛适用。

但是依旧有些相对简单技术可以避免其中的一些开销，版面的限制只允许我们在此做简单描述。例如，在 continuation monad 中，对 (>>=) 频繁的包装和解包可以避免，仅留下执行效果的开销。所幸的是使用这种方法所要考虑的大部分复杂问题，已经在函数库中得到了处理。

这一部分的工作在本书写作时尚在积极的开发中。如果你想让你对 monad 变换器的使用更加高效，我们推荐在 Hackage 中寻找相关的库或是在邮件列表或 IRC 上寻求指引。

17.8.4 缺乏灵活性的接口

若我们只把 `mtl` 当作黑盒，那么所有的组件将很好地合作。但是若我们开始开发自己的 monad 和 monad 变换器，并想让它们于 `mtl` 提供的组件配合，这种缺陷便显现出来了。

例如，我们开发一个新的 monad 变换器 `FooT` ，并想沿用 `mtl` 中的模式。我们就必须实现一个类型类 `MonadFoo` 。若我们想让其更好地和 `mtl` 配合，那么便需要提供大量的实例来支持 `mtl` 中的类型类。

除此之外，还需要为每个 `mtl` 中的变换器提供 `MonadFoo` 的实例。大部分的实例实现几乎是完全一样的，写起来也十分乏味。若我们想在 `mtl` 中集成更多的 monad 变换器，那么我们需要处理的各类活动部件将达到引入的 monad 变换器数量的 *平方级别*！

公平地看来，这个问题会只影响到少数人。大部分 `mtl` 的用户并不需要开发新的 monad。

造成这一 `mtl` 设计缺陷的原因在于，它是第一个 monad 变换器的函数库。想像其设计者投入这个未知的领域，完成了大量的工作以使这个强大的函数库对于大部分用户来说做到简便易用。

一个新的关于 monad 和变换器的函数库 `monadLib` ，修正了 `mtl` 中大量的设计缺陷。若在未来你成为了一个 monad 变换器的中坚骇客，这值得你一试。

平方级别的实例定义实际上是使用 monad 变换器带来的问题。除此之外另有其他的手段来组合利用 monad。虽然那些手段可以避免这类问题，但是它们对最终用户而言仍不及 monad 变换器便利。幸运的是，并没有太多基础而泛用的 monad 变换器需要去定义实现。

17.8.5 综述

Monad 在任何意义下都不是处理效果和类型的终极途径。它只是在我们探索至今，处理这类问题最为实用的技术。语言的研究者们一直致力于找到可以扬长避短的替代系统。

尽管在使用它们时我们必须做出妥协，monad 和 monad 变换器依旧提供了一定程度上的灵活度和控制，而这在指令式语言中并无先例。仅仅几个声明，我们就可以给分号般基础的东西赋予崭新的意义。

[译注：此处的分号应该指的是 `do` 标记中使用的分号。]

CHAPTER

EIGHTEEN

第 19 章：错误处理

无论使用哪门语言，错误处理都是程序员最重要 – 也是最容易忽视 – 的话题之一。在 Haskell 中，你会发现有两类主流的错误处理：“纯”的错误处理和异常。

当我们说“纯”的错误处理，我们是指算法不依赖任何 IO Monad。我们通常会利用 Haskell 富于表现力的数据类型系统来实现这一类错误处理。Haskell 也支持异常。由于惰性求值复杂性，Haskell 中任何地方都可能抛出异常，但是只会在 IO monad 中被捕获。在这一章中，这两类错误处理我们都会考虑。

18.1 使用数据类型进行错误处理

让我们从一个非常简单的函数来开始我们关于错误处理的讨论。假设我们希望对一系列的数字执行除法运算。分子是常数，但是分母是变化的。可能我们会写出这样一个函数：

```
-- file: ch19/divby1.hs
divBy :: Integral a => a -> [a] -> [a]
divBy numerator = map (numerator `div`)
```

非常简单，对吧？我们可以在 ghci 中执行这些代码：

```
ghci> divBy 50 [1,2,5,8,10]
[50,25,10,6,5]
ghci> take 5 (divBy 100 [1..])
[100,50,33,25,20]
```

这个行为跟我们预期的是一致的：50 / 1 得到 50，50 / 2 得到 25，等等。甚至对于无穷的链表 [1..] 它也是可以工作的。如果有个 0 溜进去我们的链表中了，会发生什么事呢？

```
ghci> divBy 50 [1,2,0,8,10]
[50,25,*** Exception: divide by zero
```

是不是很有意思？ghci 开始显示输出，然后当它遇到零时发生了一个异常停止了。这是惰性求值的作用 – 它只按需求值。

在这一章里接下来我们会看到，缺乏一个明确的异常处理时，这个异常会使程序崩溃。这当然不是我们想要的，所以让我们思考一下更好的方式来表征这个纯函数中的错误。

18.1.1 使用 Maybe

可以立刻想到的一个表示失败的简单的方法是使用 Maybe 。如果输入链表中任何地方包含了零，相对于仅仅返回一个链表并在失败的时候抛出异常，我们可以返回 Nothing ，或者如果没有出现零我们可以返回结果的 Just。下面是这个算法的实现：

```
-- file: ch19/divby2.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy _ [] = Just []
divBy _ (0:_) = Nothing
divBy numerator (denom:xs) =
    case divBy numerator xs of
      Nothing -> Nothing
      Just results -> Just ((numerator `div` denom) : results)
```

如果你在 ghci 中尝试它，你会发现它可以工作：

```
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> divBy 50 [1,2,0,8,10]
Nothing
```

调用 divBy 的函数现在可以使用 case 语句来观察调用成功与否，就像 divBy 调用自己时所做的那样。

Tip: 你大概注意到，上面可以使用一个 monadic 的实现，像这样子：

```
-- file: ch19/divby2m.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy numerator denominators =
    mapM (numerator `safeDiv`) denominators
    where safeDiv _ 0 = Nothing
          safeDiv x y = Just (x `div` y)
```

出于简单考虑，在这章中我们会避免使用 monadic 实现，但是会指出有这种做法。

[译注: 原 Tip 中代码错误，在除以非 0 数字的情况下没有返回正确类型，因此编译不过，可以使用 return 或者 Just 来修正，此处使用 Just]

丢失和保存惰性

使用 Maybe 很方便，但是有代价。divBy 将不能够再处理无限的链表输入。由于结果是一个 Maybe [a]，必须要检查整个输入链表，我们才能确认不会因为存在零而返回 Nothing 。你可以尝试在之前的例子中验证这一点：

```
ghci> divBy 100 [1..]
*** Exception: stack overflow
```

这里观察到，你没有看到部分的输出；你没得到任何输出。注意到在 divBy 的每一步中 (除了输入链表为空或者链表开头是零的情况)，每个子序列元素的结果必须先于当前元素的结果得到。因此这个算法无法处理无穷链表，并且对于大的有限链表，它的空间效率也不高。

之前已经说过，Maybe 通常是一个好的选择。在这个特殊例子中，只有当我们去执行整个输入的时候我们才知道是否有问题。有时候我们可以提交发现问题，例如，在 ghci 中 tail [] 会生成一个异常。我们可以很容易写一个可以处理无穷情况的 tail：

```
-- file: ch19/safetail.hs
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

如果输入为空，简单的返回一个 Nothing，其它情况返回结果的 Just 。由于在知道是否发生错误之前，我们只需要确认链表非空，在这里使用 Maybe 不会破坏惰性。我们可以在 ghci 中测试并观察跟普通的 tail 有何不同：

```
ghci> tail [1,2,3,4,5]
[2,3,4,5]
ghci> safeTail [1,2,3,4,5]
Just [2,3,4,5]
ghci> tail []
*** Exception: Prelude.tail: empty list
ghci> safeTail []
Nothing
```

这里我们可以看到，我们的 safeTail 执行结果符合预期。但是对于无穷链表呢？我们不想打印无穷的结果的数字，所以我们用 take 5 (tail [1..]) 以及一个类似的 saftTail 构建测试：

```
ghci> take 5 (tail [1..])
[2,3,4,5,6]
ghci> case safeTail [1..] of {Nothing -> Nothing; Just x -> Just (take 5 x)}
Just [2,3,4,5,6]
ghci> take 5 (tail [])
*** Exception: Prelude.tail: empty list
```

(continues on next page)

(continued from previous page)

```
ghci> case safeTail [] of {Nothing -> Nothing; Just x -> Just (take 5 x)}
Nothing
```

这里你可以看到 `tail` 和 `safeTail` 都可以处理无穷链表。注意我们可以更好地处理空的输入链表；而不是抛出异常，我们决定这种情况返回 `Nothing` 。我们可以获得错误处理能力却不会失去惰性。

但是我们如何将它应用到我们的 `divBy` 的例子中呢？让我们思考下现在的情况：失败是单个坏的输入的属性，而不是输入链表自身。那么将失败作为单个输出元素的属性，而不是整个输出链表怎么样？也就是说，不是一个类型为 `a -> [a] -> Maybe [a]` 的函数，取而代之我们使用 `a -> [a] -> [Maybe a]` 。这样做的好处是可以保留惰性，并且调用者可以确定是在链表中的哪里出了问题 – 或者甚至是过滤掉有问题的结果，如果需要的话。这里是一个实现：

```
-- file: ch19/divby3.hs
divBy :: Integral a => a -> [a] -> [Maybe a]
divBy numerator denominators =
    map worker denominators
    where worker 0 = Nothing
          worker x = Just (numerator `div` x)
```

看下这个函数，我们再次回到使用 `map` ，这无论对简洁和惰性都是件好事。我们可以在 `ghci` 中测试它，并观察对于有限和无限链表它都可以正常工作：

```
ghci> divBy 50 [1,2,5,8,10]
[Just 50,Just 25,Just 10,Just 6,Just 5]
ghci> divBy 50 [1,2,0,8,10]
[Just 50,Just 25,Nothing,Just 6,Just 5]
ghci> take 5 (divBy 100 [1..])
[Just 100,Just 50,Just 33,Just 25,Just 20]
```

我们希望通过这个讨论你可以明白这点，不符合规范的（正如 `safeTail` 中的情况）输入和包含坏的数据的输入 (`divBy` 中的情况) 是有区别的。这两种情况通常需要对结果采用不同的处理。

Maybe Monad 的用法

回到 使用 Maybe 这一节，我们有一个叫做 `divby2.hs` 的示例程序。这个例子没有保存惰性，而是返回一个类型为 `Maybe [a]` 的值。用 monadic 风格也可以表达同样的算法。更多信息和 monad 相关背景，参考 第 14 章 Monads 。这是我们新的 monadic 风格的算法：

```
-- file: ch19/divby4.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy _ [] = return []
divBy _ (0:_) = fail "division by zero in divBy"
```

(continues on next page)

(continued from previous page)

```
divBy numerator (denom:xs) =
    do next <- divBy numerator xs
       return ((numerator `div` denom) : next)
```

Maybe monad 使得这个算法的表示看上去更好。对于 Maybe monad，return 就跟 Just 一样，并且 fail _ = Nothing，因此我们看到任何的错误说明的字段串。我们可以用我们在 divby2.hs 中使用过的测试来测试这个算法：

```
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> divBy 50 [1,2,0,8,10]
Nothing
ghci> divBy 100 [1..]
*** Exception: stack overflow
```

我们写的代码实际上并不限于 Maybe monad。只要简单地改变类型，我们可以让它对于任何 monad 都能工作。让我们试一下：

```
-- file: ch19/divby5.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy = divByGeneric

divByGeneric :: (Monad m, Integral a) => a -> [a] -> m [a]
divByGeneric _ [] = return []
divByGeneric _ (0:_) = fail "division by zero in divByGeneric"
divByGeneric numerator (denom:xs) =
    do next <- divByGeneric numerator xs
       return ((numerator `div` denom) : next)
```

函数 divByGeneric 包含的代码 divBy 之前所做的一样；我们只是给它一个更通用的类型。事实上，如果不给出类型，这个类型是由 ghci 自动推导的。我们还为特定的类型定义了一个更方便的函数 divBy。

让我们在 ghci 中运行一下。

```
ghci> :l divby5.hs
[1 of 1] Compiling Main             ( divby5.hs, interpreted )
Ok, modules loaded: Main.
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> (divByGeneric 50 [1,2,5,8,10])::(Integral a => Maybe [a])
Just [50,25,10,6,5]
ghci> divByGeneric 50 [1,2,5,8,10]
[50,25,10,6,5]
ghci> divByGeneric 50 [1,2,0,8,10]
```

(continues on next page)

(continued from previous page)

```
*** Exception: user error (division by zero in divByGeneric)
```

前两个例子产生的输出都跟我们之前看到的一样。由于 divByGeneric 没有指定返回的类型，我们要么指定一个，要么让解释器从环境中推导得到。如果我们不指定返回类型，ghic 推荐得到 IO monad。在第三和第四个例子中你可以看出来。在第四个例子中你可以看到，IO monad 将 fail 转化成了一个异常。

mtl 包中的 Control.Monad.Error 模块也将 Either String 变成了一个 monad。如果你使用 Either，你可以得到保存了错误信息的纯的结果，像这样子：

```
ghci> :m +Control.Monad.Error
ghci> (divByGeneric 50 [1,2,5,8,10])::(Integral a => Either String [a])
Loading package mtl-1.1.0.0 ... linking ... done.
Right [50,25,10,6,5]
ghci> (divByGeneric 50 [1,2,0,8,10])::(Integral a => Either String [a])
Left "division by zero in divByGeneric"
```

这让我们进入到下一个话题的讨论：使用 Either 返回错误信息。

18.1.2 使用 Either

Either 类型跟 Maybe 类型类似，除了一处关键的不同：对于错误或者成功（“Right 类型”），它都可以携带数据。尽管语言没有强加任何限制，按照惯例，一个返回 Either 的函数使用 Left 返回值来表示一个错误，Right 来表示成功。如果你觉得这样有助于记忆，你可以认为 Right 表式正确结果。我们可以改一下前面小节中关于 Maybe 时使用的 divby2.hs 的例子，让 Either 可以工作：

```
-- file: ch19/divby6.hs
divBy :: Integral a => a -> [a] -> Either String [a]
divBy _ [] = Right []
divBy _ (0:_) = Left "divBy: division by 0"
divBy numerator (denom:xs) =
    case divBy numerator xs of
      Left x -> Left x
      Right results -> Right ((numerator `div` denom) : results)
```

这份代码跟 Maybe 的代码几乎是完全一样的；我们只是把每个 Just 用 Right 替换。Left 对应于 Nothing，但是现在它可以携带一条信息。让我们在 ghci 里面运行一下：

ghci> divBy 50 [1,2,5,8,10] Right [50,25,10,6,5] ghci> divBy 50 [1,2,0,8,10] Left “divBy: division by 0”

为错误定制数据类型

尽管用 String 类型来表示错误的原因对今后很有好处，自定义的错误类型通常会更有帮助。使用自定义的错误类型我们可以知道到底是出了什么问题，并且获知是什么动作引发的这个问题。例如，让我们假设，由于某些原因，不仅仅是除 0，我们还不想除以 10 或者 20。我们可以像这样子自定义一个错误类型：

```
-- file: ch19/divby7.hs
data DivByError a = DivBy0
                  | ForbiddenDenominator a
                    deriving (Eq, Read, Show)

divBy :: Integral a => a -> [a] -> Either (DivByError a) [a]
divBy _ [] = Right []
divBy _ (0:_) = Left DivBy0
divBy _ (10:_) = Left (ForbiddenDenominator 10)
divBy _ (20:_) = Left (ForbiddenDenominator 20)
divBy numerator (denom:xs) =
    case divBy numerator xs of
      Left x -> Left x
      Right results -> Right ((numerator `div` denom) : results)
```

现在，在出现错误时，可以通过 Left 数据检查导致错误的准确原因。或者，可以简单的只是通过 show 打印出来。下面是这个函数的应用：

```
ghci> divBy 50 [1,2,5,8]
Right [50,25,10,6]
ghci> divBy 50 [1,2,5,8,10]
Left (ForbiddenDenominator 10)
ghci> divBy 50 [1,2,0,8,10]
Left DivBy0
```

> **Warning:** 所有这些 Either 的例子都跟我们之前的 Maybe 一样，都会遇到失去惰性的问题。我们将在这一章的最后用一个练习题来解决这个问题。

Monadic 地使用 Either

回到 Maybe Monad 的用法这一节，我们向你展示了如何在一个 monad 中使用 Maybe 。Either 也可以在 monad 中使用，但是可能会复杂一点。原因是 fail 是硬编码的只接受 String 作为失败代码，因此我们必须有一种方法将这样的字符串映射成我们的 Left 使用的类型。正如你前面所见，Control.Monad.Error 为 Either String a 提供了内置的支持，它没有涉及到将参数映射到 fail 。这里我们可以将我们的例子修改为 monadic 风格使得 Either 可以工作：

```haskell
-- file: ch19/divby8.hs
{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Error

data Show a =>
    DivByError a = DivBy0
                 | ForbiddenDenominator a
                 | OtherDivByError String
                   deriving (Eq, Read, Show)

instance Error (DivByError a) where
    strMsg x = OtherDivByError x

divBy :: Integral a => a -> [a] -> Either (DivByError a) [a]
divBy = divByGeneric

divByGeneric :: (Integral a, MonadError (DivByError a) m) =>
                a -> [a] -> m [a]
divByGeneric _ [] = return []
divByGeneric _ (0:_) = throwError DivBy0
divByGeneric _ (10:_) = throwError (ForbiddenDenominator 10)
divByGeneric _ (20:_) = throwError (ForbiddenDenominator 20)
divByGeneric numerator (denom:xs) =
    do next <- divByGeneric numerator xs
       return ((numerator `div` denom) : next)
```

这里，我们需要打开 FlexibleContexts 语言扩展以提供 divByGeneric 的类型签名。divBy 函数跟之前的工作方式完全一致。对于 divByGeneric ，我们将 divByError 做为 Error 类型类的成员，通过定义调用 fail 时的行为（strMsg 函数）。我们还将 Right 转化成 return ，将 Left 转化成 throwError 进行泛化。

18.2 异常

许多语言中都有异常处理，包括 Haskell。异常很有用，因为当发生故障时，它提供了一种简单的处理方法，即使故障离发生的地方沿着函数调用链走了几层。有了异常，不需要检查每个函数调用的返回值是否发生了错误，不需要注意去生成表示错误的返回值，像 C 程序员必须这么做。在 Haskell 中，由于有 monad 以及 Either 和 Maybe 类型，你通常可以在纯的代码中达到同样的效果而不需要使用异常和异常处理。

有些问题 – 尤其是涉及到 IO 调用 – 需要处理异常。在 Haskell 中，异常可能会在程序的任何地方抛出。然而，由于计算顺序是不确定的，异常只可以在 IO monad 中捕获。Haskell 异常处理不涉及像 Python 或者 Java 中那样的特殊语法。捕获和处理异常的技术是 – 真令人惊讶 – 函数。

18.2.1 异常第一步

在 Control.Exception 模块中，定义了各种跟异常相关的函数和类型。Exception 类型是在那里定义的；所有的异常的类型都是 Exception 。还有用于捕获和处理异常的函数。让我们先看一看 try ，它的类型是 IO a -> IO (Either Exception a) 。它将异常处理包装在 IO 中。如果有异常抛出，它会返回一个 Left 值表示异常；否则，返回原始结果到 Right 值。让我们在 ghci 中运行一下。我们首先触发一个未处理的异常，然后尝试捕获它。

```
ghci> :m Control.Exception
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> print x
*** Exception: divide by zero
ghci> print y
5
ghci> try (print x)
Left divide by zero
ghci> try (print y)
5
Right ()
```

注意到在 let 语句中没有抛出异常。这是意料之中的，是因为惰性求值；除以零只有到打印 x 的值的时候才需要计算。还有，注意 try (print y) 有两行输出。第一行是由 print 产生的，它在终端上显示 5。第二个是由 ghci 生成的，这个表示 print y 的返回值为 () 并且没有抛出异常。

18.2.2 惰性和异常处理

既然你知道了 try 是如何工作的，让我们试下另一个实验。让我们假设我们想捕获 try 的结果用于后续的计算，这样我们可以处理除的结果。我们大概会这么做：

```
ghci> result <- try (return x)
Right *** Exception: divide by zero
```

这里发生了什么？让我们拆成一步一步看，先试下另一个例子：

```
ghci> let z = undefined
ghci> try (print z)
Left Prelude.undefined
ghci> result <- try (return z)
Right *** Exception: Prelude.undefined
```

跟之前一样，将 undefined 赋值给 z 没什么问题。问题的关键，以及前面的迷惑，都在于惰性求值。准确地说，是在于 return ，它没有强制它的参数的执行；它只是将它包装了一下。这样，try (return

undefined) 的结果应该是 Right undefined 。现在，ghci 想要将这个结果显示在终端上。它将运行到打印”Right”，但是 undefined 无法打印（或者说除以零的结果无法打印）。因此你看到了异常信息，它是来源于 ghci 的，而不是你的程序。

这是一个关键点。让我们想想为什么之前的例子可以工作，而这个不可以。之前，我们把 print x 放在了 try 里面。打印一些东西的值，固然是需要执行它的，因此，异常在正确的地方被检测到了。但是，仅仅是使用 return 并不会强制计算的执行。为了解决这个问题，Control.Exception 模块中定义了一个 evaluate 函数。它的行为跟 return 类似，但是会让参数立即执行。让我们试一下：

```
ghci> let z = undefined
ghci> result <- try (evaluate z)
Left Prelude.undefined
ghci> result <- try (evaluate x)
Left divide by zero
```

看，这就是我们想要的答案。无论对于 undefiined 还是除以零的例子，都可以正常工作。

Tip: 记住：任何时候你想捕获纯的代码中抛出的异常，在你的异常处理函数中使用 evaluate 而不是 return 。

18.2.3 使用 handle

通常，你可能希望如果一块代码中没有任何异常发生，就执行某个动作，否则执行不同的动作。对于像这种场合，有一个叫做 handle 的函数。这个函数的类型是 (Exception -> IO a) -> IO a -> IO a 。即是说，它需要两个参数：前一个是一个函数，当执行后一个动作发生异常的时候它会被调用。下面是我们使用的一种方式：

```
ghci> :m Control.Exception
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> handle (\_ -> putStrLn "Error calculating result") (print x)
Error calculating result
ghci> handle (\_ -> putStrLn "Error calculating result") (print y)
5
```

像这样，如果计算中没有错误发生，我们可以打印一条好的信息。这当然要比除以零出错时程序崩溃要好。

18.2.4 选择性地处理异常

上面的例子的一个问题是，对于任何异常它都是打印 “Error calculating result”。可能会有些其它不是除零的异常。例如，显示输出时可能会发生错误，或者纯的代码中可能抛出一些其它的异常。

handleJust 函数就是处理这种情况的。它让你指定一个测试来决定是否对给定的异常感兴趣。让我们看一下：

```
-- file: ch19/hj1.hs
import Control.Exception

catchIt :: Exception -> Maybe ()
catchIt (ArithException DivideByZero) = Just ()
catchIt _ = Nothing

handler :: () -> IO ()
handler _ = putStrLn "Caught error: divide by zero"

safePrint :: Integer -> IO ()
safePrint x = handleJust catchIt handler (print x)
```

cacheIt 定义了一个函数，这个函数会决定我们对给定的异常是否感兴趣。如果是，它会返回 Just ，否则返回 Nothing 。还有，Just 中附带的值会被传到我们的处理函数中。现在我们可以很好地使用 safePrint 了：

> ghci> :l hj1.hs [1 of 1] Compiling Main (hj1.hs, interpreted) Ok, modules loaded: Main. ghci> let x = 5 *div* 0 ghci> let y = 5 *div* 1 ghci> safePrint x Caught error: divide by zero ghci> safePrint y 5

Control.Exception 模块还提供了一些可以在 handleJust 中使用的函数，以便于我们将异常的范围缩小到我们所关心的类别。例如，有个函数 arithExceptions 类型是 Exception -> Maybe ArithException 可以挑选出任意的 ArithException 异常，但是会忽略掉其它。我们可以像这样使用它：

```
-- file: ch19/hj2.hs
import Control.Exception

handler :: ArithException -> IO ()
handler e = putStrLn $ "Caught arithmetic error: " ++ show e

safePrint :: Integer -> IO ()
safePrint x = handleJust arithExceptions handler (print x)
```

用这种方式，我们可以捕获所有 ArithException 类型的异常，但是仍然让其它的异常通过，不捕获也不修改。我们可以看到它是这样工作的：

```
ghci> :l hj2.hs
[1 of 1] Compiling Main             ( hj2.hs, interpreted )
Ok, modules loaded: Main.
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
```

(continues on next page)

(continued from previous page)

```
ghci> safePrint x
Caught arithmetic error: divide by zero
ghci> safePrint y
5
```

其中特别感兴趣的是，你大概注意到了 ioErrors 测试，这是跟一大类的 I/O 相关的异常。

18.2.5 I/O 异常

大概在任何程序中异常最大的来源就是 I/O。在处理外部世界的时候所有事情都可能出错：磁盘满了，网络断了，或者你期望文件里面有数据而文件却是空的。在 Haskell 中，I/O 异常就跟其它的异常一样可以用 Exception 数据类型来表示。另一方面，由于有这么多类型的 I/O 异常，有一个特殊的模块 – System.IO.Error 专门用于处理它们。

System.IO.Error 定义了两个函数：catch 和 try ，跟 Control.Exception 中的类似，它们都是用于处理异常的。然而，不像 Control.Exception 中的函数，这些函数只会捕获 I/O 错误，而不处理其它类型异常。在 Haskell 中，所有 I/O 错误有一个共同类型 IOError ，它的定义跟 IOException 是一样的。

Tip: 当心你使用的哪个名字因为 System.IO.Error 和 Control.Exception 定义了同样名字的函数，如果你将它们都导入你的程序，你将收到一个错误信息说引用的函数有歧义。你可以通过 qualified 引用其中一个或者另一个，或者将其中一个或者另一个的符号隐藏。

注意 Prelude 导出的是 System.IO.Error 中的 catch ，而不是 ControlException 中提供的。记住，前者只捕获 I/O 错误，而后者捕获所有的异常。换句话说，你要的几乎总是 Control.Exception 中的那个 catch ，而不是默认的那个。

让我们看一下对我们有益的一个在 I/O 系统中使用异常的方法。在 使用文件和句柄 这一节里，我们展示了一个使用命令式风格从文件中一行一行的读取的程序。尽管我们后面也示范过更简洁的，更” Haskelly” 的方式解决那个问题，让我们在这里重新审视这个例子。在 mainloop 函数中，在读一行之前，我们必须明确地测试我们的输入文件是否结束。这次，我们可以检查尝试读一行是否会导致一个 EOF 错误，像这样子：

```
-- file: ch19/toupper-impch20.hs
import System.IO
import System.IO.Error
import Data.Char(toUpper)

main :: IO ()
main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       mainloop inh outh
```

(continues on next page)

(continued from previous page)

```
       hClose inh
       hClose outh

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
    do input <- try (hGetLine inh)
       case input of
         Left e ->
             if isEOFError e
                then return ()
                else ioError e
         Right inpStr ->
             do hPutStrLn outh (map toUpper inpStr)
                mainloop inh outh
```

这里，我们使用 System.IO.Error 中的 try 来检测是否 hGetLine 抛出一个 IOError 。如果是，我们使用 isEOFError （在 System.IO.Error 中定义）来看是否抛出异常表明我们到达了文件末尾。如果是的，我们退出循环。如果是其它的异常，我们调用 ioError 重新抛出它。

有许多的这种测试和方法可以从 System.IO.Error 中定义的 IOError 中提取信息。我们推荐你在需要的时候去查一下库的参考页。

18.2.6 抛出异常

到现在为止，我们已经详细地讨论了异常处理。还有另外一个困惑：抛出异常。到目前为止这一章我们所接触到的例子中，都是由 Haskell 为你抛出异常的。然后你也可以自己抛出任何异常。我们会告诉你怎么做。

你将会注意到这些函数大部分似乎返回一个类型为 a 或者 IO a 的值。这意味着这个函数似乎可以返回任意类型的值。事实上，由于这些函数会抛出异常，一般情况下它们决不“返回”任何东西。这些返回值让你可以在各种各样的上下文中使用这些函数，不同的上下文需要不同的类型。

让我们使用函数 Control.Exception 来开始我们的抛出异常的教程。最通用的函数是 throw ，它的类型是 Exception -> a 。这个函数可以抛出任何的 Exception ，并且可以用于纯的上下文中。还有一个类型为 Exception -> IO a 的函数 throwIO 在 IO monad 中抛出异常。这两个函数都需要一个 Exception 用于抛出。你可以手工制作一个 Exception ， 或者重用之前创建的 Exception 。

还有一个函数 ioError ， 它在 Control.Exception 和 System.IO.Error 中定义都是相同的，它的类型是 IOError -> IO a 。当你想生成任意的 I/O 相关的异常的时候可以使用它。

18.2.7 动态异常

这需要使用两个很不常用的 Haskell 模块: Data.Dynamic 和 Data.Typeable 。我们不会讲太多关于这些模块，但是告诉你当你需要制作自己的动态异常类型时，可以使用这些工具。

在 http://book.realworldhaskell.org/read/using-databases.html 中，你会看到 HDBC 数据库库使用动态异常来表示 SQL 数据库返回给应用的错误。数据库引擎返回的错误通常有三个组件：一个表示错误码的整数，一个状态，以及一条人类可读的错误消息。在这一章中我们会创建我们自己的 HDBC `SqlError` 实现。让我们从错误自身的数据结构表示开始：

```
-- file: ch19/dynexc.hs
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Dynamic
import Control.Exception

data SqlError = SqlError {seState :: String,
                          seNativeError :: Int,
                          seErrorMsg :: String}
                deriving (Eq, Show, Read, Typeable)
```

通过继承 `Typeable` 类型类，我们使这个类型可用于动态的类型编程。为了让 GHC 自动生成一个 `Typeable` 实例，我们要开启 `DeriveDataTypeable` 语言扩展。

现在，让我们定义一个 `catchSql` 和一个 `handleSql` 用于捕猎一个 `SqlError` 异常。注意常规的 `catch` 和 `handle` 函数无法捕猎我们的 `SqlError` ，因为它不是 `Exception` 类型的。

```
-- file: ch19/dynexc.hs
{- | Execute the given IO action.

If it raises a 'SqlError', then execute the supplied
handler and return its return value.  Otherwise, proceed
as normal. -}
catchSql :: IO a -> (SqlError -> IO a) -> IO a
catchSql = catchDyn

{- | Like 'catchSql', with the order of arguments reversed. -}
handleSql :: (SqlError -> IO a) -> IO a -> IO a
handleSql = flip catchSql
```

[译注：原文中文件名是 dynexc.hs，但是跟前面的冲突了，所以这里重命名为 dynexc1.hs]

这些函数仅仅是在 `catchDyn` 外面包了很薄的一层，类型是 `Typeable exception => IO a -> (exception -> IO a) -> IO a` 。这里我们简单地限定了它的类型使得它只捕猎 SQL 异常。

正常地，当一个异常抛出，但是没有在任何地方被捕获，程序会崩溃并显示异常到标准错误输出。然而，对于动态异常，系统不会知道该如何显示它，因此你将仅仅会看到一个的”unknown exception”消息，这可能没太大帮助。我们可以提供一个辅助函数，这样应用可以写成，比如说 `main = handleSqlError $ do ...` ，使抛出的异常可以显示。下面是如何写 `handleSqlError` ：

```
-- file: ch19/dynexc.hs
{- | Catches 'SqlError's, and re-raises them as IO errors with fail.
Useful if you don't care to catch SQL errors, but want to see a sane
error message if one happens.  One would often use this as a
high-level wrapper around SQL calls. -}
handleSqlError :: IO a -> IO a
handleSqlError action =
    catchSql action handler
    where handler e = fail ("SQL error: " ++ show e)
```

[译注：原文中是 dynexc.hs，这里重命名过文件]

最后，让我们给出一个如何抛出 SqlError 异常的例子。下面的函数做的就是这件事：

```
-- file: ch19/dynexc.hs
throwSqlError :: String -> Int -> String -> a
throwSqlError state nativeerror errormsg =
    throwDyn (SqlError state nativeerror errormsg)

throwSqlErrorIO :: String -> Int -> String -> IO a
throwSqlErrorIO state nativeerror errormsg =
    evaluate (throwSqlError state nativeerror errormsg)
```

Tip: 提醒一下，evaluate 跟 return 类似但是会立即计算它的参数。

这样我们的动态异常的支持就完成了。代码很多，你大概不需要这么多代码，但是我们想要给你一个动态异常自身的例子以及和它相关的工具。事实上，这里的例子几乎就反映在 HDBC 库中。让我们在 ghci 中试一下：

```
ghci> :l dynexc.hs
[1 of 1] Compiling Main             ( dynexc.hs, interpreted )
Ok, modules loaded: Main.
ghci> throwSqlErrorIO "state" 5 "error message"
*** Exception: (unknown)
ghci> handleSqlError $ throwSqlErrorIO "state" 5 "error message"
*** Exception: user error (SQL error: SqlError {seState = "state", seNativeError = 5,␣
↪seErrorMsg = "error message"})
ghci> handleSqlError $ fail "other error"
*** Exception: user error (other error)
```

这里你可以看出，ghci 自己并不知道如何显示 SQL 错误。但是，你可以看到 handleSqlError 帮助做了这些，不过没有捕获其它的错误。最后让我们试一个自定义的 handler：

```
ghci> handleSql (fail . seErrorMsg) (throwSqlErrorIO "state" 5 "my error")
*** Exception: user error (my error)
```

这里，我们自定义了一个错误处理抛出一个新的异常，构成 SqlError 中的 seErrorMsg 域。你可以看到它是按预想中那样工作的。

18.3 练习

1. 将 Either 修改成 Maybe 例子中的那种风格，使它保存惰性。

18.4 monad 中的错误处理

因为我们必须捕获 IO monad 中的异常，如果我们在一个 monad 中或者在 monad 的转化栈中使用它们，我们将跳出到 IO monad。这几乎肯定不是我们想要的。

在 构建以理解 Monad 变换器 中我们定义了一个 MaybeT 的变换，但是它更像是一个有助于理解的东西，而不是编程的工具。幸运的是，已经有一个专门的 – 也更有用的 –monad 变换：ErrorT，它是定义在 Control.Monad.Error 模块中的。

ErrorT 变换器使我们可以向 monad 中添加异常，但是它使用了特殊的方法，跟 Control.Exception 模块中提供的不一样。它提供给我们一些有趣的能力。

- 如果我们继续用 ErrorT 接口，在这个 monad 中我们可以抛出和捕获异常。
- 根据其它 monad 变换器的命名规范，这个执行函数的名字是 runErrorT 。当它遇到 runErrorT 之后，未被捕获的 ErrorT 异常将停止向上传递。我们不会被踢到 IO monad 中。
- 我们可以控制我们的异常的类型。

Warning: 不要把 ErrorT 跟普通异常混淆如果我们在 ErrorT 内面使用 Control.Exception 中的 throw 函数，我们仍然会弹出到 IO monad。

正如其它的 mtl monad 一样，ErrorT 提供的接口是由一个类型类定义的。

```
-- file: ch19/MonadError.hs
class (Monad m) => MonadError e m | m -> e where
    throwError :: e             -- error to throw
               -> m a

    catchError :: m a           -- action to execute
```

(continues on next page)

(continued from previous page)

```
              -> (e -> m a)     -- error handler
              -> m a
```

类型变量 e 代表我们想要使用的错误类型。不管我们的错误类型是什么，我们必须将它做成 Error 类型类的实例。

```
-- file: ch19/MonadError.hs
class Error a where
    -- create an exception with no message
    noMsg  :: a

    -- create an exception with a message
    strMsg :: String -> a
```

ErrorT 实现 fail 时会用到 strMsg 函数。它将 strMsg 作为一个异常抛出，将自己接收到的字符串参数传递给这个异常。对于 noMsg ，它是用于提供 MonadPlus 类型类中的 mzero 的实现。

为了支持 strMsg 和 noMsg 函数，我们的 ParseError 类型会有一个 Chatty 构造器。这个将用作构造器如果，比如说，有人在我们的 monad 中调用 fail 。

我们需要知道的最后一块是关于执行函数 runErrorT 的类型。

```
ghci> :t runErrorT
runErrorT :: ErrorT e m a -> m (Either e a)
```

18.4.1 一个小的解析构架

为了说明 ErrorT 的使用，让我们开发一个类似于 Parsec 的解析库的基本的骨架。

```
-- file: ch19/ParseInt.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Monad.Error
import Control.Monad.State
import qualified Data.ByteString.Char8 as B

data ParseError = NumericOverflow
                | EndOfInput
                | Chatty String
                  deriving (Eq, Ord, Show)

instance Error ParseError where
```

(continues on next page)

(continued from previous page)

```
    noMsg  = Chatty "oh noes!"
    strMsg = Chatty
```

对于我们解析器的状态，我们会创建一个非常小的 monad 变换器栈。一个 State monad 包含了需要解析的 ByteString，在栈的顶部是 ErrorT 用于提供错误处理。

```
-- file: ch19/ParseInt.hs
newtype Parser a = P {
      runP :: ErrorT ParseError (State B.ByteString) a
    } deriving (Monad, MonadError ParseError)
```

和平常一样，我们将我们的 monad 栈包装在一个 newtype 中。这样做没有任意性能损耗，但是增加了类型安全。我们故意避免继承 MonadState B.ByteString 的实例。这意味着 Parser monad 用户将不能够使用 get 或者 put 去查询或者修改解析器的状态。这样的结果是，我们强制自己去做一些手动提升的事情来获取在我们栈中的 State monad。

```
-- file: ch19/ParseInt.hs
liftP :: State B.ByteString a -> Parser a
liftP m = P (lift m)

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
  s <- liftP get
  case B.uncons s of
    Nothing         -> throwError EndOfInput
    Just (c, s')
        | p c       -> liftP (put s') >> return c
        | otherwise -> throwError (Chatty "satisfy failed")
```

catchError 函数对于我们的任何非常有用，远胜于简单的错误处理。例如，我们可以很轻松地解除一个异常，将它变成更友好的形式。

```
-- file: ch19/ParseInt.hs
optional :: Parser a -> Parser (Maybe a)
optional p = (Just `liftM` p) `catchError` \_ -> return Nothing
```

我们的执行函数仅仅是将各层连接起来，将结果重新组织成更整洁的形式。

```
-- file: ch19/ParseInt.hs
runParser :: Parser a -> B.ByteString
          -> Either ParseError (a, B.ByteString)
runParser p bs = case runState (runErrorT (runP p)) bs of
                   (Left err, _) -> Left err
```

(continues on next page)

(continued from previous page)

```
                    (Right r, bs) -> Right (r, bs)
```

如果我们将它加载到 ghci 中，我们可以对它进行了一些测试。

```
ghci> :m +Data.Char
ghci> let p = satisfy isDigit
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> runParser p (B.pack "x")
Left (Chatty "satisfy failed")
ghci> runParser p (B.pack "9abc")
Right ('9',"abc")
ghci> runParser (optional p) (B.pack "x")
Right (Nothing,"x")
ghci> runParser (optional p) (B.pack "9a")
Right (Just '9',"a")
```

18.4.2 练习

1. 写一个 many 解析器，类型是 `Parser a -> Parser [a]` 。它应该执行解析直到失败。
2. 使用 many 写一个 int 解析器，类型是 `Parser Int` 。它应该既能接受负数也能接受正数。
3. 修改你们 int 解析器，如果在解析时检测到了一个数值溢出，抛出一个 NumericOverflow 异常。

注

CHAPTER

NINETEEN

第 20 章：使用 HASKELL 进行系统编程

目前为止，我们讨论的大多数是高阶概念。Haskell 也可以用于底层系统编程。完全可以使用 Haskell 编写使用操作系统底层接口的程序。

本章中，我们将尝试一些很有野心的东西：编写一种类似 Perl 实际上是合法的 Haskell 的“语言”，完全使用 Haskell 实现，用于简化编写 shell 脚本。我们将实现管道，简单命令调用，和一些简单的工具用于执行由 `grep` 和 `sed` 处理的任务。

有些模块是依赖操作系统的。本章中，我们将尽可能使用不依赖特殊操作系统的通用模块。不过，本章将有很多内容着眼于 POSIX 环境。POSIX 是一种类 Unix 标准，如 Linux ，FreeBSD ，MacOS X ，或 Solaris 。Windows 默认情况下不支持 POSIX ，但是 Cygwin 环境为 Windows 提供了 POSIX 兼容层。

19.1 调用外部程序

Haskell 可以调用外部命令。为了这么做，我们建议使用 `System.Cmd` 模块中的 `rawSystem` 。其用特定的参数调用特定的程序，并将返回程序的退出状态码。你可以在 ghci 中练习一下。

```
ghci> :module System.Cmd
ghci> rawSystem "ls" ["-l", "/usr"]
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package unix-2.3.0.0 ... linking ... done.
Loading package process-1.0.0.0 ... linking ... done.
total 124
drwxr-xr-x   2 root root  49152 2008-08-18 11:04 bin
drwxr-xr-x   2 root root   4096 2008-03-09 05:53 games
drwxr-sr-x  10 jimb guile  4096 2006-02-04 09:13 guile
drwxr-xr-x  47 root root   8192 2008-08-08 08:18 include
drwxr-xr-x 107 root root  32768 2008-08-18 11:04 lib
lrwxrwxrwx   1 root root      3 2007-09-24 16:55 lib64 -> lib
```

(continues on next page)

(continued from previous page)

```
drwxrwsr-x  17 root staff  4096 2008-06-24 17:35 local
drwxr-xr-x   2 root root   8192 2008-08-18 11:03 sbin
drwxr-xr-x 181 root root   8192 2008-08-12 10:11 share
drwxrwsr-x   2 root src    4096 2007-04-10 16:28 src
drwxr-xr-x   3 root root   4096 2008-07-04 19:03 X11R6
ExitSuccess
```

此处，我们相当于执行了 shell 命令 `ls -l /usr` 。`rawSystem` 并不从字符串解析输入参数或是扩展通配符[43] 。取而代之，其接受一个包含所有参数的列表。如果不想提供参数，可以像这样简单地输入一个空列表。

```
ghci> rawSystem "ls" []
calendartime.ghci  modtime.ghci    rp.ghci    RunProcessSimple.hs
cmd.ghci       posixtime.hs    rps.ghci   timediff.ghci
dir.ghci       rawSystem.ghci  RunProcess.hs  time.ghci
ExitSuccess
```

19.2 目录和文件信息

`System.Directory` 模块包含了相当多可以从文件系统获取信息的函数。你可以获取某目录包含的文件列表，重命名或删除文件，复制文件，改变当前工作路径，或者建立新目录。`System.Directory` 是可移植的，在可以跑 GHC 的平台都可以使用。

System.Directory 的库文档 中含有一份详尽的函数列表。让我们通过 ghci 来对其中一些进行演示。这些函数大多数简单的等价于其对应的 C 语言库函数或 shell 命令。

```
ghci> :module System.Directory
ghci> setCurrentDirectory "/etc"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
ghci> getCurrentDirectory
"/etc"
ghci> setCurrentDirectory ".."
ghci> getCurrentDirectory
"/"
```

此处我们看到了改变工作目录和获取当前工作目录的命令。它们类似 POSIX shell 中的 `cd` 和 `pwd` 命令。

[43] 也有一个 `system` 函数，接受单个字符串为参数，并将其传入 shell 解析。我们推荐使用 `rawSystem` ，因为某些字符在 shell 中具有特殊含义，可能会导致安全隐患或者意外的行为。

```
ghci> getDirectoryContents "/"
[".","..","lost+found","boot","etc","media","initrd.img","var","usr","bin","dev","home
↪","lib","mnt","proc","root","sbin","tmp","sys","lib64","srv","opt","initrd","vmlinuz
↪",".rnd","www","ultra60","emul",".fonts.cache-1","selinux","razor-agent.log",".svn",
↪"initrd.img.old","vmlinuz.old","ugid-survey.bulkdata","ugid-survey.brief"]
```

getDirectoryContents 返回一个列表，包含给定目录的所有内容。注意，在 POSIX 系统中，这个列表通常包含特殊值 “.” 和 “..”。通常在处理目录内容时，你可能会希望将他们过滤出去，像这样：

```
ghci> getDirectoryContents "/" >>= return . filter (`notElem` [".", ".."])
["lost+found","boot","etc","media","initrd.img","var","usr","bin","dev","home","lib",
↪"mnt","proc","root","sbin","tmp","sys","lib64","srv","opt","initrd","vmlinuz",".rnd
↪","www","ultra60","emul",".fonts.cache-1","selinux","razor-agent.log",".svn",
↪"initrd.img.old","vmlinuz.old","ugid-survey.bulkdata","ugid-survey.brief"]
```

Tip: 更细致的讨论如何过滤 getDirectoryContents 函数的结果，请参考第八章：高效文件处理、正则表达式、文件名匹配

filter (`notElem` [".", ".."]) 这段代码是否有点莫名其妙? 也可以写作 filter (c -> not $ elem c [".", ".."]) 。反引号让我们更有效的将第二个参数传给 notElem ; 在 “中序函数” 一节中有关于反引号更详细的信息。

也可以向系统查询某些路径的位置。这将向底层操作系统发起查询相关信息。

```
ghci> getHomeDirectory
"/home/bos"
ghci> getAppUserDataDirectory "myApp"
"/home/bos/.myApp"
ghci> getUserDocumentsDirectory
"/home/bos"
```

19.3 终止程序

开发者经常编写独立的程序以完成特定任务。这些独立的部分可能会被组合起来完成更大的任务。一段 shell 脚本或者其他程序将会执行它们。发起调用的脚本需要获知被调用程序是否执行成功。Haskell 自动为异常退出的程序分配一个 “不成功” 的状态码。

不过，你需要对状态码进行更细粒度的控制。可能你需要对不同类型的错误返回不同的代码。System.Exit 模块提供一个途径可以在程序退出时返回特定的状态码。通过调用 exitWith ExitSuccess 表示程序执行成功 (POSIX 系统中的 0)。或者可以调用 exitWith (ExitFailure 5) ，表示将在程序退出时向系统返回 5 作为状态码。

19.4 日期和时间

从文件时间戳到商业事务的很多事情都涉及到日期和时间。除了从系统获取日期时间信息之外，Haskell 提供了很多关于时间日期的操作方法。

19.4.1 ClockTime 和 CalendarTime

在 Haskell 中，日期和时间主要由 `System.Time` 模块处理。它定义了两个类型：`ClockTime` 和 `CalendarTime` 。

`ClockTime` 是传统 POSIX 中时间戳的 Haskell 版本。`ClockTime` 表示一个相对于 UTC 1970 年 1 月 1 日零点的时间。负值的 `ClockTime` 表示在其之前的秒数，正值表示在其之后的秒数。

`ClockTime` 便于计算。因为它遵循协调世界时（Coordinated Universal Time，UTC），其不必调整本地时区、夏令时或其他时间处理中的特例。每天是精确的 (60 * 60 * 24) 或 86,400 秒[44]，这易于计算时间间隔。举个例子，你可以简单的记录某个程序开始执行的时间和其结束的时间，相减即可确定程序的执行时间。如果需要的话，还可以除以 3600，这样就可以按小时显示。

使用 `ClockTime` 的典型场景：

- 经过了多长时间?
- 相对此刻 14 天前是什么时间?
- 文件的最后修改时间是何时?
- 当下的精确时间是何时?

ClockTime 善于处理这些问题，因为它们使用无法混淆的精确时间。但是，`ClockTime` 不善于处理下列问题：

- 今天是周一吗?
- 明年 5 月 1 日是周几?
- 在我的时区当前是什么时间，考虑夏令时。

`CalendarTime` 按人类的方式存储时间：年，月，日，小时，分，秒，时区，夏令时信息。很容易的转换为便于显示的字符串，或者以上问题的答案。

你可以任意转换 `ClockTime` 和 `CalendarTime` 。Haskell 将 `ClockTime` 可以按本地时区转换为 `CalendarTime` ，或者按 `CalendarTime` 格式表示的 UTC 时间。

[44] 可能有人会注意到 UTC 定义了不规则的闰秒。在 Haskell 使用的 POSIX 标准中，规定了在其表示的时间中，每天必须都是精确的 86,400 秒，所以在执行日常计算时无需担心闰秒。精确的处理闰秒依赖于系统而且复杂，不过通常其可以被解释为一个“长秒”。这个问题大体上只是在执行精确的亚秒级计算时才需要关心。

使用 ClockTime

ClockTime 在 System.Time 中这样定义：

```
data ClockTime = TOD Integer Integer
```

第一个 Integer 表示从 Unix 纪元开始经过的秒数。第二个 Integer 表示附加的皮秒数。因为 Haskell 中的 ClockTime 使用无边界的 Integer 类型，所以其能够表示的数据范围仅受计算资源限制。

让我们看看使用 ClockTime 的一些方法。首先是按系统时钟获取当前时间的 getClockTime 函数。

```
ghci> :module System.Time
ghci> getClockTime
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Mon Aug 18 12:10:38 CDT 2008
```

如果一秒钟再次运行 getClockTime，它将返回一个更新后的时间。这条命令会输出一个便于观察的字符串，补全了周相关的信息。这是由于 ClockTime 的 Show 实例。让我们从更底层看一下 ClockTime：

```
ghci> TOD 1000 0
Wed Dec 31 18:16:40 CST 1969
ghci> getClockTime >>= (\(TOD sec _) -> return sec)
1219079438
```

这里我们先构建一个 ClockTime，表示 UTC 时间 1970 年 1 月 1 日午夜后 1000 秒这个时间点。在你的时区这个时间相当于 1969 年 12 月 31 日晚。

第二个例子演示如何从 getClockTime 返值中将秒数取出来。我们可以像这样操作它：

```
ghci> getClockTime >>= (\(TOD sec _) -> return (TOD (sec + 86400) 0))
Tue Aug 19 12:10:38 CDT 2008
```

这将显精确示你的时区 24 小时后的时间，因为 24 小时等于 86,400 秒。

使用 CalendarTime

正如其名字暗示的，CalendarTime 按日历上的方式表示时间。它包括年、月、日等信息。CalendarTime 和其相关类型定义如下：

```
data CalendarTime = CalendarTime
   {ctYear :: Int,          -- Year (post-Gregorian)
    ctMonth :: Month,
    ctDay :: Int,           -- Day of the month (1 to 31)
```

(continues on next page)

(continued from previous page)

```
    ctHour :: Int,          -- Hour of the day (0 to 23)
    ctMin :: Int,           -- Minutes (0 to 59)
    ctSec :: Int,           -- Seconds (0 to 61, allowing for leap seconds)
    ctPicosec :: Integer,   -- Picoseconds
    ctWDay :: Day,          -- Day of the week
    ctYDay :: Int,          -- Day of the year (0 to 364 or 365)
    ctTZName :: String,     -- Name of timezone
    ctTZ :: Int,            -- Variation from UTC in seconds
    ctIsDST :: Bool         -- True if Daylight Saving Time in effect
   }

data Month = January | February | March | April | May | June
             | July | August | September | October | November | December

data Day = Sunday | Monday | Tuesday | Wednesday
           | Thursday | Friday | Saturday
```

关于以上结构有些事情需要强调：

- ctWDay, ctYDay, ctTZName 是被创建 CalendarTime 的库函数生成的，但是并不参与计算。如果你手工创建一个 CalendarTime ，不必向其中填写准确的值，除非你的计算依赖于它们。
- 这三个类型都是 Eq, Ord, Read, Show 类型类的成员。另外，Month 和 Day 都被声明为 Enum 和 Bounded 类型类的成员。更多的信息请参考“重要的类型类”这一章节。

有几种不同的途径可以生成 CalendarTime 。可以像这样将 ClockTime 转换为 CalendarTime ：

```
ghci> :module System.Time
ghci> now <- getClockTime
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Mon Aug 18 12:10:35 CDT 2008
ghci> nowCal <- toCalendarTime now
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10,␣
↪ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYDay = 230, ctTZName = "CDT
↪", ctTZ = -18000, ctIsDST = True}
ghci> let nowUTC = toUTCTime now
ghci> nowCal
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10,␣
↪ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYDay = 230, ctTZName = "CDT
↪", ctTZ = -18000, ctIsDST = True}
ghci> nowUTC
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 17, ctMin = 10,␣
↪ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYDay = 230, ctTZName = "UTC
↪", ctTZ = 0, ctIsDST = False}
```

用 getClockTime 从系统获得当前的 ClockTime 。接下来，toCalendarTime 按本地时间区将 ClockTime 转换为 CalendarTime 。toUTCtime 执行类似的转换，但其结果将以 UTC 时区表示。

注意，toCalendarTime 是一个 IO 函数，但是 toUTCTime 不是。原因是 toCalendarTime 依赖本地时区返回不同的结果，但是针对相同的 ClockTime ， toUTCTime 将始终返回相同的结果。

很容易改变一个 CalendarTime 的值

```
ghci> nowCal {ctYear = 1960}
CalendarTime {ctYear = 1960, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10,␣
↪ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYDay = 230, ctTZName = "CDT
↪", ctTZ = -18000, ctIsDST = True}
ghci> (\(TOD sec _) -> sec) (toClockTime nowCal)
1219079435
ghci> (\(TOD sec _) -> sec) (toClockTime (nowCal {ctYear = 1960}))
-295685365
```

此处，先将之前的 CalendarTime 年份修改为 1960 。然后用 toClockTime 将其初始值转换为一个 ClockTime ，接着转换新值，以便观察其差别。注意新值在转换为 ClockTime 后显示了一个负的秒数。这是意料中的，ClockTime 表示的是 UTC 时间 1970 年 1 月 1 日午夜之后的秒数。

也可以像这样手工创建 CalendarTime：

```
ghci> let newCT = CalendarTime 2010 January 15 12 30 0 0 Sunday 0 "UTC" 0 False
ghci> newCT
CalendarTime {ctYear = 2010, ctMonth = January, ctDay = 15, ctHour = 12, ctMin = 30,␣
↪ctSec = 0, ctPicosec = 0, ctWDay = Sunday, ctYDay = 0, ctTZName = "UTC", ctTZ = 0,␣
↪ctIsDST = False}
ghci> (\(TOD sec _) -> sec) (toClockTime newCT)
1263558600
```

注意，尽管 2010 年 1 月 15 日并不是一个周日 – 并且也不是一年中的第 0 天 – 系统可以很好的处理这些情况。实际上，如果将其转换为 ClockTime 后再转回 CalendarTime ，你将发现这些域已经被正确的处理了。

```
ghci> toUTCTime . toClockTime $ newCT
CalendarTime {ctYear = 2010, ctMonth = January, ctDay = 15, ctHour = 12, ctMin = 30,␣
↪ctSec = 0, ctPicosec = 0, ctWDay = Friday, ctYDay = 14, ctTZName = "UTC", ctTZ = 0,␣
↪ctIsDST = False}
```

ClockTime 的 TimeDiff

以对人类友好的方式难于处理 ClockTime 值之间的差异，System.Time 模块包括了一个 TimeDiff 类型。TimeDiff 用于方便的处理这些差异。其定义如下：

```
data TimeDiff = TimeDiff
   {tdYear :: Int,
    tdMonth :: Int,
    tdDay :: Int,
    tdHour :: Int,
    tdMin :: Int,
    tdSec :: Int,
    tdPicosec :: Integer}
```

diffClockTimes 和 addToClockTime 两个函数接收一个 ClockTime 和一个 TimeDiff 并在内部将 ClockTime 转换为一个 UTC 时区的 CalendarTime ， 在其上执行 TimeDiff ， 最后将结果转换回一个 ClockTime 。

看看它怎样工作:

```
ghci> :module System.Time
ghci> let feb5 = toClockTime $ CalendarTime 2008 February 5 0 0 0 0 Sunday 0 "UTC" 0␣
↪False
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
ghci> feb5
Mon Feb  4 18:00:00 CST 2008
ghci> addToClockTime (TimeDiff 0 1 0 0 0 0 0) feb5
Tue Mar  4 18:00:00 CST 2008
ghci> toUTCTime $ addToClockTime (TimeDiff 0 1 0 0 0 0 0) feb5
CalendarTime {ctYear = 2008, ctMonth = March, ctDay = 5, ctHour = 0, ctMin = 0, ctSec␣
↪= 0, ctPicosec = 0, ctWDay = Wednesday, ctYDay = 64, ctTZName = "UTC", ctTZ = 0,␣
↪ctIsDST = False}
ghci> let jan30 = toClockTime $ CalendarTime 2009 January 30 0 0 0 0 Sunday 0 "UTC" 0␣
↪False
ghci> jan30
Thu Jan 29 18:00:00 CST 2009
ghci> addToClockTime (TimeDiff 0 1 0 0 0 0 0) jan30
Sun Mar  1 18:00:00 CST 2009
ghci> toUTCTime $ addToClockTime (TimeDiff 0 1 0 0 0 0 0) jan30
CalendarTime {ctYear = 2009, ctMonth = March, ctDay = 2, ctHour = 0, ctMin = 0, ctSec␣
↪= 0, ctPicosec = 0, ctWDay = Monday, ctYDay = 60, ctTZName = "UTC", ctTZ = 0,␣
↪ctIsDST = False}
ghci> diffClockTimes jan30 feb5
TimeDiff {tdYear = 0, tdMonth = 0, tdDay = 0, tdHour = 0, tdMin = 0, tdSec = 31104000,
↪ tdPicosec = 0}
ghci> normalizeTimeDiff $ diffClockTimes jan30 feb5
TimeDiff {tdYear = 0, tdMonth = 12, tdDay = 0, tdHour = 0, tdMin = 0, tdSec = 0,␣
↪tdPicosec = 0}
```

首先我们生成一个 ClockTime 表示 UTC 时间 2008 年 2 月 5 日。注意，若你的时区不是 UTC，按你本地时区的格式，当其被显示的时候可能是 2 月 4 日晚。

其次，我们用 addToClockTime 在其上加一个月。2008 是闰年，但系统可以正确的处理，然后我们得到了一个月后的相同日期。使用 toUTCTime ，我们可以看到以 UTC 时间表示的结果。

第二个实验，设定一个表示 UTC 时间 2009 年 1 月 30 日午夜的时间。2009 年不是闰年，所以我们可能很好奇其加上一个月是什么结果。因为 2009 年没有 2 月 29 日和 2 月 30 日，所以我们得到了 3 月 2 日。

最后，我们可以看到 diffClockTimes 怎样通过两个 ClockTime 值得到一个 TimeDiff ，尽管其只包含秒和皮秒。normalizeTimeDiff 函数接受一个 TimeDiff 将其重新按照人类的习惯格式化。

19.4.2 文件修改日期

很多程序需要找出某些文件的最后修改日期。ls 和图形化的文件管理器是典型的需要显示文件最后变更时间的程序。System.Directory 模块包含一个跨平台的 getModificationTime 函数。其接受一个文件名，返回一个表示文件最后变更日期的 ClockTime 。例如：

```
ghci> :module System.Directory
ghci> getModificationTime "/etc/passwd"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Fri Aug 15 08:29:48 CDT 2008
```

POSIX 平台不仅维护变更时间 (被称为 mtime)，还有最后读或写访问时间 (atime) 以及最后状态变更时间 (ctime)。这是 POSIX 平台独有的，所以跨平台的 System.Directory 模块无法访问它。取而代之，需要使用 System.Posix.Files 模块中的函数。下面有一个例子：

```
-- file: ch20/posixtime.hs
-- posixtime.hs

import System.Posix.Files
import System.Time
import System.Posix.Types

-- | Given a path, returns (atime, mtime, ctime)
getTimes :: FilePath -> IO (ClockTime, ClockTime, ClockTime)
getTimes fp =
    do stat <- getFileStatus fp
       return (toct (accessTime stat),
               toct (modificationTime stat),
               toct (statusChangeTime stat))
```

(continues on next page)

(continued from previous page)

```haskell
-- | Convert an EpochTime to a ClockTime
toct :: EpochTime -> ClockTime
toct et =
    TOD (truncate (toRational et)) 0
```

注意对 getFileStatus 的调用。这个调用直接映射到 C 语言的 stat() 函数。其返回一个包含了大量不同种类信息的值，包括文件类型、权限、属主、组、和我们感性去的三种时间值。System.Posix.Files 提供了 accessTime 等多个函数，可以将我们感兴趣的时间从 getFileStatus 返回的 FileStatus 类型中提取出来。

> accessTime 等函数返回一个 POSIX 平台特有的类型，称为 EpochTime ，可以通过 toct 函数转换 ClockTime 。System.Posix.Files 模块同样提供了 setFileTimes 函数，以设置文件的 atime 和 mtime 。[45]

19.5 延伸的例子: 管道

我们已经了解了如何调用外部程序。有时候需要更多的控制。比如获得程序的标准输出、提供输入，甚至将不同的外部程序串起来调用。管道有助于实现所有这些需求。管道经常用在 shell 脚本中。在 shell 中设置一个管道，会调用多个程序。第一个程序的输入会做为第二个程序的输入。其输出又会作为第三个的输入，以此类推。最后一个程序通常将输出打印到终端，或者写入文件。下面是一个 POSIX shell 的例子，演示如何使用管道:

```
$ ls /etc | grep 'm.*ap' | tr a-z A-Z
IDMAPD.CONF
MAILCAP
MAILCAP.ORDER
MEDIAPRM
TERMCAP
```

这条命令运行了三个程序，使用管道在它们之间传输数据。它以 ls/etc 开始，输出是 /etc 目录下全部文件和目录的列表。ls 的输出被作为 grep 的输入。我们想 grep 输入一条正则使其只输出以‘m’开头并且在某处包含“ap”的行。最后，其结果被传入 tr 。我们给 tr 设置一个选项，使其将所有字符转换为大写。tr 的输出没有特殊的去处，所以直接在屏幕显示。

这种情况下，程序之间的管道线路由 shell 设置。我们可以使用 Haskell 中的 POSIX 工具实现同的事情。

在讲解如何实现之前，要提醒你一下，System.Posix 模块提供的是很低阶的 Unix 系统接口。无论使用何种编程语言，这些接口都可以相互组合，组合的结果也可以相互组合。这些低阶接口的完整性质可以用一整本书来讨论，这章中我们只会简单介绍。

[45] POSIX 系统上通常无法设置 ctime 。

19.5.1 使用管道做重定向

POSIX 定义了一个函数用于创建管道。这个函数返回两个文件描述符（FD），与 Haskell 中的句柄概念类似。一个 FD 用于读端，另一个用于写端。任何从写端写入的东西，都可以从读端读取。这些数据就是“通过管道推送”的。在 Haskell 中，你可以通过 `createPipe` 使用这个接口。

在外部程序之间传递数据之前，要做的第一步是建立一个管道。同时还要将一个程序的输出重定向到管道，并将管道做为另一个程序的输入。Haskell 的 `dupTo` 函数就是做这个的。其接收一个 FD 并将其拷贝为另一个 FD 号。POSIX 的标准输入、标准输出和标准错误的 FD 分别被预定义为 0, 1, 2 。将管道的某一端设置为这些 FD 号，我们就可以有效的重定向程序的输入和输出。

不过还有问题需要解决。我们不能简单的只是在某个调用比如 `rawSystem` 之前使用 `dupTo` ，因为这回混淆我们的 Haskell 主程序的输入和输出。此外，`rawSystem` 会一直阻塞直到被调用的程序执行完毕，这让我们无法启动并行执行的进程。为了解决这个问题，可以使用 `forkProcess` 。这是一个很特殊的函数。它实际上生成了一份当前进程的拷贝，并使这两份进程同时运行。Haskell 的 `forkProcess` 函数接收一个函数，使其在新进程（称为子进程）中运行。我们让这个函数调用 `dupTo` 。之后，其调用 `executeFile` 调用真正希望执行的命令。这同样也是一个特殊的函数：如果一切顺利，他将不会返回。这是因为 `executeFile` 使用一个不同的程序替换了当前执行的进程。最后，初始的 Haskell 进程调用 `getProcessStatus` 以等待子进程结束，并获得其状态码。

在 POSIX 系统中，无论何时你执行一条命令，不关是在命令上上敲 `ls` 还是在 Haskell 中使用 `rawSystem` ，其内部机理都是调用 `forkProcess` , `executeFile` , 和 `getProcessStatusa` (或是它们对应的 C 函数)。为了使用管道，我们复制了系统启动程序的进程，并且加入了一些调用和重定向管道的步骤。

还有另外一些辅助步骤需要注意。当调用 `forkProcess` 时，“几乎”和程序有关的一切都被复制[46] 。包括所有已经打开的文件描述符（句柄）。程序通过检查管道是否传来文件结束符判断数据接收是否结束。写端进程关闭管道时，读端程序将收到文件结束符。然而，如果同一个写端文件描述符在多个进程中同时存在，则文件结束符要在所有进程中都被关闭才会发送文件结束符。因此，我们必须在子进程中追踪打开了哪些文件描述符，以便关闭它们。同样，也必须尽早在主进程中关闭子进程的写管道。

下面是一个用 Haskell 编写的管道系统的初始实现：

```
-- file: ch20/RunProcessSimple.hs

{-# OPTIONS_GHC -XDatatypeContexts #-}
{-# OPTIONS_GHC -XTypeSynonymInstances #-}
{-# OPTIONS_GHC -XFlexibleInstances #-}

module RunProcessSimple where

--import System.Process
import Control.Concurrent
import Control.Concurrent.MVar
```

(continues on next page)

[46] 线程是一个主要例外，其不会被复制，所以说“几乎”。

(continued from previous page)

```
import System.IO
import System.Exit
import Text.Regex.Posix
import System.Posix.Process
import System.Posix.IO
import System.Posix.Types
import Control.Exception

{- | The type for running external commands.  The first part
of the tuple is the program name.  The list represents the
command-line parameters to pass to the command. -}
type SysCommand = (String, [String])

{- | The result of running any command -}
data CommandResult = CommandResult {
    cmdOutput :: IO String,              -- ^ IO action that yields the output
    getExitStatus :: IO ProcessStatus    -- ^ IO action that yields exit result
    }

{- | The type for handling global lists of FDs to always close in the clients
-}
type CloseFDs = MVar [Fd]

{- | Class representing anything that is a runnable command -}
class CommandLike a where
    {- | Given the command and a String representing input,
         invokes the command.  Returns a String
         representing the output of the command. -}
    invoke :: a -> CloseFDs -> String -> IO CommandResult

-- Support for running system commands
instance CommandLike SysCommand where
    invoke (cmd, args) closefds input =
        do -- Create two pipes: one to handle stdin and the other
           -- to handle stdout.  We do not redirect stderr in this program.
           (stdinread, stdinwrite) <- createPipe
           (stdoutread, stdoutwrite) <- createPipe

           -- We add the parent FDs to this list because we always need
           -- to close them in the clients.
           addCloseFDs closefds [stdinwrite, stdoutread]

           -- Now, grab the closed FDs list and fork the child.
```

(continues on next page)

(continued from previous page)

```
            childPID <- withMVar closefds (\fds ->
                            forkProcess (child fds stdinread stdoutwrite))

            -- Now, on the parent, close the client-side FDs.
            closeFd stdinread
            closeFd stdoutwrite

            -- Write the input to the command.
            stdinhdl <- fdToHandle stdinwrite
            forkIO $ do hPutStr stdinhdl input
                        hClose stdinhdl

            -- Prepare to receive output from the command
            stdouthdl <- fdToHandle stdoutread

            -- Set up the function to call when ready to wait for the
            -- child to exit.
            let waitfunc =
                 do status <- getProcessStatus True False childPID
                    case status of
                        Nothing -> fail $ "Error: Nothing from getProcessStatus"
                        Just ps -> do removeCloseFDs closefds
                                              [stdinwrite, stdoutread]
                                      return ps
            return $ CommandResult {cmdOutput = hGetContents stdouthdl,
                                    getExitStatus = waitfunc}

        -- Define what happens in the child process
        where child closefds stdinread stdoutwrite =
                do -- Copy our pipes over the regular stdin/stdout FDs
                   dupTo stdinread stdInput
                   dupTo stdoutwrite stdOutput

                   -- Now close the original pipe FDs
                   closeFd stdinread
                   closeFd stdoutwrite

                   -- Close all the open FDs we inherited from the parent
                   mapM_ (\fd -> catch (closeFd fd) (\(SomeException e) -> return␣
↪()))) closefds

                   -- Start the program
                   executeFile cmd True args Nothing
```

(continues on next page)

(continued from previous page)

```
-- Add FDs to the list of FDs that must be closed post-fork in a child
addCloseFDs :: CloseFDs -> [Fd] -> IO ()
addCloseFDs closefds newfds =
    modifyMVar_ closefds (\oldfds -> return $ oldfds ++ newfds)

-- Remove FDs from the list
removeCloseFDs :: CloseFDs -> [Fd] -> IO ()
removeCloseFDs closefds removethem =
    modifyMVar_ closefds (\fdlist -> return $ procfdlist fdlist removethem)

    where
    procfdlist fdlist [] = fdlist
    procfdlist fdlist (x:xs) = procfdlist (removefd fdlist x) xs

    -- We want to remove only the first occurance ot any given fd
    removefd [] _ = []
    removefd (x:xs) fd
        | fd == x = xs
        | otherwise = x : removefd xs fd

{- | Type representing a pipe.  A 'PipeCommand' consists of a source
and destination part, both of which must be instances of
'CommandLike'. -}
data (CommandLike src, CommandLike dest) =>
     PipeCommand src dest = PipeCommand src dest

{- | A convenient function for creating a 'PipeCommand'. -}
(-|-) :: (CommandLike a, CommandLike b) => a -> b -> PipeCommand a b
(-|-) = PipeCommand

{- | Make 'PipeCommand' runnable as a command -}
instance (CommandLike a, CommandLike b) =>
         CommandLike (PipeCommand a b) where
    invoke (PipeCommand src dest) closefds input =
        do res1 <- invoke src closefds input
           output1 <- cmdOutput res1
           res2 <- invoke dest closefds output1
           return $ CommandResult (cmdOutput res2) (getEC res1 res2)

{- | Given two 'CommandResult' items, evaluate the exit codes for
both and then return a "combined" exit code.  This will be ExitSuccess
if both exited successfully.  Otherwise, it will reflect the first
```

(continues on next page)

(continued from previous page)

```
error encountered. -}
getEC :: CommandResult -> CommandResult -> IO ProcessStatus
getEC src dest =
    do sec <- getExitStatus src
       dec <- getExitStatus dest
       case sec of
            Exited ExitSuccess -> return dec
            x -> return x

{- | Execute a 'CommandLike'. -}
runIO :: CommandLike a => a -> IO ()
runIO cmd =
    do -- Initialize our closefds list
       closefds <- newMVar []

       -- Invoke the command
       res <- invoke cmd closefds []

       -- Process its output
       output <- cmdOutput res
       putStr output

       -- Wait for termination and get exit status
       ec <- getExitStatus res
       case ec of
            Exited ExitSuccess -> return ()
            x -> fail $ "Exited: " ++ show x
```

在研究这个函数的运作原理之前，让我们先来在 ghci 里面尝试运行它一下：

```
ghci> runIO $ ("pwd", []::[String])
/Users/Blade/sandbox

ghci> runIO $ ("ls", ["/usr"])
NX
X11
X11R6
bin
include
lib
libexec
local
sbin
share
```

(continues on next page)

(continued from previous page)

```
standalone

ghci> runIO $ ("ls", ["/usr"]) -|- ("grep", ["^l"])
lib
libexec
local

ghci> runIO $ ("ls", ["/etc"]) -|- ("grep", ["m.*ap"]) -|- ("tr", ["a-z", "A-Z"])
COM.APPLE.SCREENSHARING.AGENT.LAUNCHD
```

我们从一个简单的命令 pwd 开始，它会打印当前工作目录。我们将 [] 做为参数列表，因为 pwd 不需要任何参数。由于使用了类型类，Haskell 无法自动推导出 [] 的类型，所以我们说明其类型为字符串组成的列表。

下面是一个更复杂些的例子。我们执行了 ls ，将其输出传入 grep 。最后我们通过管道，调用了一个与本节开始处 shell 内置管道的例子中相同的命令。不像 shell 中那样舒服，但是相对于 shell 我们的程序始终相对简单。

让我们读一下程序。起始处的 OPTIONS_GHC 语句，作用与 ghc 或 ghci 开始时传入 -fglasgow-exts 参数相同。我们使用了一个 GHC 扩展，以允许使用 (String, [String]) 类型作为一个类型类的实例[47] 。将此类声明加入源码文件，就不用在每次调用这个模块的时候都要记得手工打开编译器开关。

在载入了所需模块之后，定义了一些类型。首先，定义 type SysCommand = (String, [String]) 作为一个别名。这是系统将接收并执行的命令的类型。例子中的每条领命都要用到这个类型的数据。CommandResult 命令用于表示给定命令的执行结果，CloseFDs 用于表示必须在新的子进程中关闭的文件描述符列表。

接着，定义一个类称为 CommandLike 。这个类用来跑“东西”，这个“东西”可以是独立的程序，可以是两个程序之间的管道，未来也可以跑纯 Haskell 函数。任何一个类型想为这个类的成员，只需实现一个函数 – invoke 。这将允许以 runIO 启动一个独立命令或者一个管道。这在定义管道时也很有用，因为我们可以拥有某个管道的读写两端的完整调用栈。

我们的管道基础设施将使用字符串在进程间传递数据。我们将通过 hGetContents 获得 Haskell 在延迟读取方面的优势，并使用 forkIO 在后台写入。这种设计工作得不错，尽管传输速度不像将两个进程的管道读写端直接连接起来那样快[48] 。但这让实现很简单。我们仅需要小心，不要做任何会让整个字符串被缓冲的操作，把接下来的工作完全交给 Haskell 的延迟特性。

接下来，为 SysCommand 定义一个 CommandLike 实例。我们创建两个管道：一个用来作为新进程的标准输入，另一个用于其标准输出。将产生两个读端两个写端，四个文件描述符。我们将要在子进程中关闭的文件描述符加入列表。这包括子进程标准输入的写端，和子进程标准输出的读端。接着，我们 fork 出子进程。然后可以在父进程中关闭相关的子进程文件描述符。fork 之前不能这样做，因为那时子进程还不可用。获取 stdinwrite 的句柄，并通过 forkIO 启动一个现成向其写入数据。接着定义 waitfunc , 其中定义了

[47] Haskell 社区对这个扩展支持得很好。Hugs 用户可以通过 hugs -98 +o 使用。

[48] Haskell 的 HSH 库提供了与此相近的 API , 使用了更高效 (也更复杂) 的机构将外部进程使用管道直接连接起来, 没有要传给 Haskell 处理的数据。shell 采用了相同的方法，而且这样可以降低处理管道的 CPU 负载。

调用这在准备好等待子进程结束时要执行的动作。同时，子进程使用 `dupTo` ，关闭其不需要的文件描述符。并执行命令。

然后定义一些工具函数用来管理文件描述符。此后，定义一些工具用于建立管道。首先，定义一个新类型 `PipeCommand` ，其有源和目的两个属性。源和目的都必须是 `CommandLike` 的成员。为了方便，我们还定义了 `-|-` 操作符。然后使 `PipeCommand` 成为 `CommandLike` 的实例。它调用第一个命令并获得输出，将其传入第二个命令。之后返回第二个命令的输出，并调用 `getExitStatus` 函数等待命令执行结束并检查整组命令执行之后的状态码。

最后以定义 `runIO` 结束。这个函数建立了需要在子进程中关闭的 FDS 列表，执行程序，显示输出，并检查其退出状态。

19.5.2 更好的管道

上个例子中解决了一个类似 shell 的管道系统的基本需求。但是为它加上下面这些特点之后就更好了：

- 支持更多的 shell 语法。
- 使管道同时支持外部程序和正规 Haskell 函数，并使二者可以自由的混合使用。
- 以易于 Haskell 程序利用的方式返回标准输出和退出状态码。

幸运的是，支持这些功能的代码片段已经差不多就位了。只需要为 `CommandLike` 多加入几个实例，以及一些类似 `runIO` 的函数。下面是修订后实现了以上功能的例子代码：

```
-- file: ch20/RunProcess.hs
{-# OPTIONS_GHC -XDatatypeContexts #-}
{-# OPTIONS_GHC -XTypeSynonymInstances #-}
{-# OPTIONS_GHC -XFlexibleInstances #-}

module RunProcess where

import System.Process
import Control.Concurrent
import Control.Concurrent.MVar
import Control.Exception
import System.Posix.Directory
import System.Directory(setCurrentDirectory)
import System.IO
import System.Exit
import Text.Regex
import System.Posix.Process
import System.Posix.IO
import System.Posix.Types
import Data.List
```

(continues on next page)

(continued from previous page)

```
import System.Posix.Env(getEnv)

{- | The type for running external commands.  The first part
of the tuple is the program name.  The list represents the
command-line parameters to pass to the command. -}
type SysCommand = (String, [String])

{- | The result of running any command -}
data CommandResult = CommandResult {
    cmdOutput :: IO String,              -- ^ IO action that yields the output
    getExitStatus :: IO ProcessStatus    -- ^ IO action that yields exit result
    }

{- | The type for handling global lists of FDs to always close in the clients
-}
type CloseFDs = MVar [Fd]

{- | Class representing anything that is a runnable command -}
class CommandLike a where
    {- | Given the command and a String representing input,
         invokes the command.  Returns a String
         representing the output of the command. -}
    invoke :: a -> CloseFDs -> String -> IO CommandResult

-- Support for running system commands
instance CommandLike SysCommand where
    invoke (cmd, args) closefds input =
        do -- Create two pipes: one to handle stdin and the other
           -- to handle stdout.  We do not redirect stderr in this program.
           (stdinread, stdinwrite) <- createPipe
           (stdoutread, stdoutwrite) <- createPipe

           -- We add the parent FDs to this list because we always need
           -- to close them in the clients.
           addCloseFDs closefds [stdinwrite, stdoutread]

           -- Now, grab the closed FDs list and fork the child.
           childPID <- withMVar closefds (\fds ->
                       forkProcess (child fds stdinread stdoutwrite))

           -- Now, on the parent, close the client-side FDs.
           closeFd stdinread
           closeFd stdoutwrite
```

(continues on next page)

(continued from previous page)

```
            -- Write the input to the command.
            stdinhdl <- fdToHandle stdinwrite
            forkIO $ do hPutStr stdinhdl input
                        hClose stdinhdl

            -- Prepare to receive output from the command
            stdouthdl <- fdToHandle stdoutread

            -- Set up the function to call when ready to wait for the
            -- child to exit.
            let waitfunc =
                 do status <- getProcessStatus True False childPID
                    case status of
                        Nothing -> fail $ "Error: Nothing from getProcessStatus"
                        Just ps -> do removeCloseFDs closefds
                                                [stdinwrite, stdoutread]
                                      return ps
            return $ CommandResult {cmdOutput = hGetContents stdouthdl,
                                    getExitStatus = waitfunc}

        -- Define what happens in the child process
        where child closefds stdinread stdoutwrite =
                do -- Copy our pipes over the regular stdin/stdout FDs
                   dupTo stdinread stdInput
                   dupTo stdoutwrite stdOutput

                   -- Now close the original pipe FDs
                   closeFd stdinread
                   closeFd stdoutwrite

                   -- Close all the open FDs we inherited from the parent
                   mapM_ (\fd -> catch (closeFd fd) (\(SomeException e) -> return␣
↪())) closefds

                   -- Start the program
                   executeFile cmd True args Nothing

{- | An instance of 'CommandLike' for an external command.  The String is
passed to a shell for evaluation and invocation. -}
instance CommandLike String where
    invoke cmd closefds input =
        do -- Use the shell given by the environment variable SHELL,
```

(continues on next page)

(continued from previous page)

```
            -- if any.  Otherwise, use /bin/sh
            esh <- getEnv "SHELL"
            let sh = case esh of
                       Nothing -> "/bin/sh"
                       Just x -> x
            invoke (sh, ["-c", cmd]) closefds input

-- Add FDs to the list of FDs that must be closed post-fork in a child
addCloseFDs :: CloseFDs -> [Fd] -> IO ()
addCloseFDs closefds newfds =
    modifyMVar_ closefds (\oldfds -> return $ oldfds ++ newfds)

-- Remove FDs from the list
removeCloseFDs :: CloseFDs -> [Fd] -> IO ()
removeCloseFDs closefds removethem =
    modifyMVar_ closefds (\fdlist -> return $ procfdlist fdlist removethem)

    where
    procfdlist fdlist [] = fdlist
    procfdlist fdlist (x:xs) = procfdlist (removefd fdlist x) xs

    -- We want to remove only the first occurance ot any given fd
    removefd [] _ = []
    removefd (x:xs) fd
        | fd == x = xs
        | otherwise = x : removefd xs fd

-- Support for running Haskell commands
instance CommandLike (String -> IO String) where
    invoke func _ input =
       return $ CommandResult (func input) (return (Exited ExitSuccess))

-- Support pure Haskell functions by wrapping them in IO
instance CommandLike (String -> String) where
    invoke func = invoke iofunc
        where iofunc :: String -> IO String
              iofunc = return . func

-- It's also useful to operate on lines.  Define support for line-based
-- functions both within and without the IO monad.

instance CommandLike ([String] -> IO [String]) where
    invoke func _ input =
```

(continues on next page)

(continued from previous page)

```
          return $ CommandResult linedfunc (return (Exited ExitSuccess))
      where linedfunc = func (lines input) >>= (return . unlines)

instance CommandLike ([String] -> [String]) where
    invoke func = invoke (unlines . func . lines)

{- | Type representing a pipe.  A 'PipeCommand' consists of a source
and destination part, both of which must be instances of
'CommandLike'. -}
data (CommandLike src, CommandLike dest) =>
     PipeCommand src dest = PipeCommand src dest

{- | A convenient function for creating a 'PipeCommand'. -}
(-|-) :: (CommandLike a, CommandLike b) => a -> b -> PipeCommand a b
(-|-) = PipeCommand

{- | Make 'PipeCommand' runnable as a command -}
instance (CommandLike a, CommandLike b) =>
         CommandLike (PipeCommand a b) where
    invoke (PipeCommand src dest) closefds input =
        do res1 <- invoke src closefds input
           output1 <- cmdOutput res1
           res2 <- invoke dest closefds output1
           return $ CommandResult (cmdOutput res2) (getEC res1 res2)

{- | Given two 'CommandResult' items, evaluate the exit codes for
both and then return a "combined" exit code.  This will be ExitSuccess
if both exited successfully.  Otherwise, it will reflect the first
error encountered. -}
getEC :: CommandResult -> CommandResult -> IO ProcessStatus
getEC src dest =
    do sec <- getExitStatus src
       dec <- getExitStatus dest
       case sec of
            Exited ExitSuccess -> return dec
            x -> return x

{- | Different ways to get data from 'run'.

 * IO () runs, throws an exception on error, and sends stdout to stdout

 * IO String runs, throws an exception on error, reads stdout into
   a buffer, and returns it as a string.
```

(continues on next page)

(continued from previous page)

```
 * IO [String] is same as IO String, but returns the results as lines

 * IO ProcessStatus runs and returns a ProcessStatus with the exit
   information.  stdout is sent to stdout.  Exceptions are not thrown.

 * IO (String, ProcessStatus) is like IO ProcessStatus, but also
   includes a description of the last command in the pipe to have
   an error (or the last command, if there was no error)

 * IO Int returns the exit code from a program directly.  If a signal
   caused the command to be reaped, returns 128 + SIGNUM.

 * IO Bool returns True if the program exited normally (exit code 0,
   not stopped by a signal) and False otherwise.

-}
class RunResult a where
    {- | Runs a command (or pipe of commands), with results presented
       in any number of different ways. -}
    run :: (CommandLike b) => b -> a

-- | Utility function for use by 'RunResult' instances
setUpCommand :: CommandLike a => a -> IO CommandResult
setUpCommand cmd =
    do -- Initialize our closefds list
       closefds <- newMVar []

       -- Invoke the command
       invoke cmd closefds []

instance RunResult (IO ()) where
    run cmd = run cmd >>= checkResult

instance RunResult (IO ProcessStatus) where
    run cmd =
        do res <- setUpCommand cmd

           -- Process its output
           output <- cmdOutput res
           putStr output

           getExitStatus res
```

(continues on next page)

(continued from previous page)

```
instance RunResult (IO Int) where
    run cmd = do rc <- run cmd
                 case rc of
                   Exited (ExitSuccess) -> return 0
                   Exited (ExitFailure x) -> return x
                   (Terminated x _) -> return (128 + (fromIntegral x))
                   Stopped x -> return (128 + (fromIntegral x))

instance RunResult (IO Bool) where
    run cmd = do rc <- run cmd
                 return ((rc::Int) == 0)

instance RunResult (IO [String]) where
    run cmd = do r <- run cmd
                 return (lines r)

instance RunResult (IO String) where
    run cmd =
        do res <- setUpCommand cmd

           output <- cmdOutput res

           -- Force output to be buffered
           evaluate (length output)

           ec <- getExitStatus res
           checkResult ec
           return output

checkResult :: ProcessStatus -> IO ()
checkResult ps =
    case ps of
         Exited (ExitSuccess) -> return ()
         x -> fail (show x)

{- | A convenience function.  Refers only to the version of 'run'
that returns @IO ()@.  This prevents you from having to cast to it
all the time when you do not care about the result of 'run'.
-}
runIO :: CommandLike a => a -> IO ()
runIO = run
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------
-- Utility Functions
------------------------------------------------------------
cd :: FilePath -> IO ()
cd = setCurrentDirectory

{- | Takes a string and sends it on as standard output.
The input to this function is never read. -}
echo :: String -> String -> String
echo inp _ = inp

-- | Search for the regexp in the lines.  Return those that match.
grep :: String -> [String] -> [String]
grep pat = filter (ismatch regex)
    where regex = mkRegex pat
          ismatch r inp = case matchRegex r inp of
                            Nothing -> False
                            Just _ -> True

{- | Creates the given directory.  A value of 0o755 for mode would be typical.
An alias for System.Posix.Directory.createDirectory. -}
mkdir :: FilePath -> FileMode -> IO ()
mkdir = createDirectory

{- | Remove duplicate lines from a file (like Unix uniq).
Takes a String representing a file or output and plugs it through
lines and then nub to uniqify on a line basis. -}
uniq :: String -> String
uniq = unlines . nub . lines

-- | Count number of lines.  wc -l
wcL, wcW :: [String] -> [String]
wcL inp = [show (genericLength inp :: Integer)]

-- | Count number of words in a file (like wc -w)
wcW inp = [show ((genericLength $ words $ unlines inp) :: Integer)]

sortLines :: [String] -> [String]
sortLines = sort

-- | Count the lines in the input
countLines :: String -> IO String
countLines = return . (++) "\n" . show . length . lines
```

主要改变是：

- `String` 的 `CommandLike` 实例，以便在 shell 中对字符串进行求值和调用。
- `String -> IO String` 的实例，以及其它几种相关类型的实现。这样就可以像处理命令一样处理 Haskell 函数。
- `RunResult` 类型类，定义了一个 `run` 函数，其可以用多种不同方式返回命令的相关信息。
- 一些工具函数，提供了用 Haskell 实现的类 Unix shell 命令。

现在来试试这些新特性。首先确定一下之前例子中的命令是否还能工作。然后，使用新的类 shell 语法运行一下。

```
ghci> :load RunProcess.hs
[1 of 1] Compiling RunProcess       ( RunProcess.hs, interpreted )
Ok, modules loaded: RunProcess.

ghci> runIO $ ("ls", ["/etc"]) -|- ("grep", ["m.*ap"]) -|- ("tr", ["a-z", "A-Z"])
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package containers-0.5.5.1 ... linking ... done.
Loading package filepath-1.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package unix-2.7.0.1 ... linking ... done.
Loading package directory-1.2.1.0 ... linking ... done.
Loading package process-1.2.0.0 ... linking ... done.
Loading package transformers-0.3.0.0 ... linking ... done.
Loading package mtl-2.1.3.1 ... linking ... done.
Loading package regex-base-0.93.2 ... linking ... done.
Loading package regex-posix-0.95.2 ... linking ... done.
Loading package regex-compat-0.95.1 ... linking ... done.
COM.APPLE.SCREENSHARING.AGENT.LAUNCHD

ghci> runIO $ "ls /etc" -|- "grep 'm.*ap'" -|- "tr a-z A-Z"
COM.APPLE.SCREENSHARING.AGENT.LAUNCHD
```

输入起来容易多了。试试使用 Haskell 实现的 grep 来试一下其它的新特性：

```
ghci> runIO $ "ls /usr/local/bin" -|- grep "m.*ap" -|- "tr a-z A-Z"
DUMPCAP
MERGECAP
NMAP
```

(continues on next page)

(continued from previous page)

```
ghci> run $ "ls /usr/local/bin" -|- grep "m.*ap" -|- "tr a-z A-Z" :: IO String
"DUMPCAP\nMERGECAP\nNMAP\n"

ghci> run $ "ls /usr/local/bin" -|- grep "m.*ap" -|- "tr a-z A-Z" :: IO [String]
["DUMPCAP","MERGECAP","NMAP"]

ghci> run $ "ls /usr" :: IO String
"X11\nX11R6\nbin\ninclude\nlib\nlibexec\nlocal\nsbin\nshare\nstandalone\ntexbin\n"

ghci> run $ "ls /usr" :: IO Int
X11
X11R6
bin
include
lib
libexec
local
sbin
share
standalone
texbin
0

ghci> runIO $ echo "Line1\nHi, test\n" -|- "tr a-z A-Z" -|- sortLines
HI, TEST
LINE1
```

19.5.3 关于管道，最后说几句

我们开发了一个精巧的系统。前面时醒过，POSIX 有时会很复杂。另外要强调一下：要始终注意确保先将这些函数返回的字符串求值，然后再尝试获取子进程的退出状态码。子进程经常要等待写出其所有输出之后才能退出，如果搞错了获取输出和退出状态码的顺序，你的程序会挂住。

本章中，我们从零开始开发了一个精简版的 HSH 。如果你希望使程序具有这样类 shell 的功能，我们推荐使用 HSH 而非上面开发的例子，因为 HSH 的实现更加优化。HSH 还有一个数量庞大的工具函数集和更多功能，但其背后的代码也更加庞大和复杂。其实例子中很多工具函数的代码我们是直接从 HSH 抄过来的。可以从 http://software.complete.org/hsh 访问 HSH 的源码。

注

CHAPTER

TWENTY

第 21 章：数据库的使用

网上论坛、播客抓取器（podcatchers）甚至备份程序通常都会使用数据库进行持久化储存。基于 SQL 的数据库非常常见：这种数据库具有速度快、伸缩性好、可以通过网络进行操作等优点，它们通常会负责处理加锁和事务，有些数据库甚至还提供了故障恢复（failover）功能以提高应用程序的冗余性（redundancy）。市面上的数据库有很多不同的种类：既有 Oracle 这样大型的商业数据库，也有 PostgreSQL 、MySQL 这样的开源引擎，甚至还有 Sqlite 这样的可嵌入引擎。

因为数据库是如此的重要，所以 Haskell 也必须对数据库进行支持。本章将介绍其中一个与数据库进行互动的 Haskell 框架，并使用这个框架去构建一个播客下载器（podcast downloader），本书的 23 章还会对这个博客下载器做进一步的扩展。

20.1 HDBC 简介

数据库引擎位于数据库栈（stack）的最底层，引擎负责将数据实际地储存到硬盘里面，常见的数据库引擎有 PostgreSQL 、MySQL 和 Oracle 。

大多数现代化的数据库引擎都支持 SQL ，也即是结构化查询语言（Structured Query Language），并将这种语言用作读取和写入关系式数据库的标准方式。不过本书并不会提供 SQL 或者关系式数据库管理方面的教程[49]。

在拥有了支持 SQL 的数据库引擎之后，用户还需要寻找一种方法与引擎进行通信。虽然每个数据库都有自己的独有协议，但是因为各个数据库处理的 SQL 几乎都是相同的，所以通过为不同的协议提供不同的驱动，以此来创建一个通用的接口是完全可以做到的。

Haskell 有几种不同的数据库框架可用，其中某些框架在其他框架的基础上提供了更高层次的抽象，而本章将对 HDBC ——也即是 Haskell DataBase Connectivity 系统进行介绍。通过 HDBC ，用户可以在只需进行少量修改甚至无需进行修改的情况下，访问储存在任意 SQL 数据库里面的数据[50]。即使你并不需要更换底层的数据引擎，由多个驱动构成的 HDBC 系统也使得你在单个接口上面有更多选择可用。

[49] O’Reilly 出版的《Learning SQL and SQL in a Nutshell》对于没有 SQL 经验的读者来说可能会有所帮助。

[50] 假设你只能使用标准的 SQL 。

HSQL 是 Haskell 的另一个数据库抽象库，它与 HDBC 具有相似的构想。除此之外，Haskell 还有一个名为 HaskellDB 的高层次框架，这个框架可以运行在 HDBC 或是 HSQL 之上，它被设计于用来为程序员隔离处理 SQL 时的相关细节。因为 HaskellDB 的设计无法处理一些非常常见的数据库访问模式，所以它并未被广泛引用。最后，Takusen 是一个使用左折叠（left fold）方式从数据库里面读取数据的框架。

20.2 安装 HDBC 和驱动

为了使用 HDBC 去连给定的数据库，用户至少需要用到两个包：一个包是 HDBC 的通用接口，而另一个包则是针对给定数据库的驱动。HDBC 包和所有其他驱动都可以通过 Hackage[51] 获得，本章将使用 1.1.3 版本的 HDBC 作为示例。

除了 HDBC 包之外，用户还需要准备数据库后端和数据库驱动。本章会使用 Sqlite 3 作为数据库后端，这个数据库是一个嵌入式数据库，因此它不需要独立的服务器，并且也非常容易设置。很多操作系统本身就内置了 Sqlite 3 ，如果你的系统里面没有提供这一数据库，那么你可以到 http://www.sqlite.org/ 里面进行下载。HDBC 的主页上面列出了指向已有 HDBC 后端驱动的链接，针对 Sqlite 3 的驱动也可以通过 Hackage 下载到。

[Forec 译注：配置此章环境需要加载 `HDBC` 和 `HDBC-Sqlite3` 两个包。Windows 用户在安装 `HDBC-Sqlite3` 时需要 Sqlite3 的源码和动态链接库。以 cabal 用户为例，在 http://sqlite.org/download.html 下载 `sqlite-amalgamation-xxxx.zip` ，将其解压到某个目录如 `/usr/hdbc`，下载 `sqlite-dll-xxxx.zip`，解压到 `/usr/hdbc_lib`。执行 `cabal install HDBC-Sqlite3 --extra-include-dirs=/usr/hdbc --extra-lib-dirs=/usr/hdbc_lib` 。Linux 用户以 Debian 源为例，需先通过 `apt` 安装 sqlite3 和 libsqlite3-dev，之后再执行 `cabal install` 。如果这里的译注对你的环境仍不适用，可以在 https://github.com/hdbc/hdbc/wiki/FrequentlyAskedQuestions 寻找有无对应解决方案。]

如果读者打算使用 HDBC 去处理其他数据库，那么可以在 http://software.complete.org/hdbc/wiki/KnownDrivers 查看 HDBC 已有的驱动：上面展示的 ODBC 绑定（binding）基本上可以让你在任何平台（Windows、POSIX 等等）上面连接任何数据库；针对 PostgreSQL 的绑定也是存在的；而 MySQL 同样可以通过 ODBC 绑定进行支持，具体的信息可以在 HDBC-ODBC API 文档里面找到。

20.3 连接数据库

连接至数据库需要用到数据库后端驱动提供的连接函数。每个数据库都有自己独特的连接方法。用户通常只会在初始化连接的时候直接调用从后端驱动模块载入的函数。

数据库连接函数会返回一个数据库连接，不同驱动的数据库连接类型可能并不相同，但它们总是 `IConnection` 类型类的一个实例，并且所有数据库操作函数都能够与这种类型的实例进行协作。

在完成了与数据库的通信指挥，用户只要调用 `disconnect` 函数就可以断开与数据库的连接。以下代码展示了怎样去连接一个 Sqlite 数据库：

[51] 想要了解更多关于安装 Haskell 软件的相关信息，请阅读本书的《安装 Haskell 软件》一节。

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3

ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package HDBC-1.1.5 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.

ghci> :type conn
conn :: Connection

ghci> disconnect conn
```

20.4 事务

大部分现代化 SQL 数据库都具有事务的概念。事务可以确保一项修改的所有组成部分都会被实现，又或者全部都不实现。更进一步来说，事务可以避免访问相同数据库的多个进程看见正在进行的修改动作所产生的不完整数据。

大多数数据库都要求用户通过显式的提交操作来将所有修改储存到硬盘上面，又或者在“自动提交”模式下运行：这种模式在每一条语句的后面都会进行一次隐式的提交。“自动提交”模式可能会给不熟悉事务数据库的程序员带来一些方便，但它对于那些真正想要执行多条语句事务的人来说却是一个阻碍。

HDBC 有意地不对自动提交模式进行支持。当用户在修改数据库的数据之后，它必须显式地将修改提交到硬盘上面。有两种方法可以在 HDBC 里面做到这件事：第一种方法就是在准备好将数据写入到硬盘的时候，调用 `commit` 函数；而另一种方法则是将修改数据的代码包裹到 `withTransaction` 函数里面。`withTransaction` 会在被包裹的函数成功执行之后自动执行提交操作。

在将数据写入到数据库里面的时候，可能会出现问题。也许是因为数据库出错了，又或者数据库发现正在提交的数据出现了问题。在这种情况下，用户可以“回滚”事务进行的修改：回滚动作会撤销最近一次提交或是最近一次回滚之后发生的所有修改。在 HDBC 里面，你可以通过 `rollback` 函数来进行回滚。如果你使用 `withTransaction` 函数来包裹事务，那么函数将在事务发生异常时自动进行回滚。

要记住，回滚操作只会撤销掉最近一次 `commit` 函数、`rollback` 函数或者 `withTransaction` 函数引发的修改。数据库并不会像版本控制系统那样记录全部历史信息。本章稍后将展示一些 `commit` 函数的使用示例。

20.5 简单的查询示例

最简单的 SQL 查询语句都是一些不返回任何数据的语句，这些查询可以用于创建表、插入数据、删除数据、又或者设置数据库的参数。

run 函数是向数据库发送查询的最基本的函数，这个函数接受三个参数，它们分别是一个 IConnection 实例、一个表示查询的 String 以及一个由列表组成的参数。以下代码展示了如何使用这个函数去将一些数据储存到数据库里面。

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3

ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package HDBC-1.1.5 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.

ghci> run conn "CREATE TABLE test (id INTEGER NOT NULL, desc VARCHAR(80))" []
0

ghci> run conn "INSERT INTO test (id) VALUES (0)" []
1

ghci> commit conn

ghci> disconnect conn
```

在连接到数据库之后，程序首先创建了一个名为 test 的表，接着向表里面插入了一个行。最后，程序将修改提交到数据库，并断开与数据库的连接。记住，如果程序不调用 commit 函数，那么修改将不会被写入到数据库里面。

run 函数返回因为查询语句而被修改的行数量。在上面展示的代码里面，第一个查询只是创建一个表，它并没有修改任何行；而第二个查询则向表里面插入了一个行，因此 run 函数返回了数字 1 。

20.6 SqlValue

在继续讨论后续内容之前，我们需要先了解一种由 HDBC 引入的数据类型：SqlValue 。因为 Haskell 和 SQL 都是强类型系统，所以 HDBC 会尝试尽可能地保留类型信息。与此同时，Haskell 和 SQL 类型并不是一一对应的。更进一步来说，日期和字符串里面的特殊字符这样的东西，在每个数据库里面的表示方法都是不相同

的。

SqlValue 类型具有 SqlString 、SqlBool 、SqlNull 、SqlInteger 等多个构造器，用户可以通过使用这些构造器，在传给数据库的参数列表里面表示各式各样不同类型的数据，并且仍然能够将这些数据储存到一个列表里面。除此之外，SqlValue 还提供了 toSql 和 fromSql 这样的常用函数。如果你非常关心数据的精确表示的话，那么你还是可以在有需要的时候，手动地构造 SqlValue 数据。

20.7 查询参数

HDBC 和其他数据库一样，都支持可替换的查询参数。使用可替换参数主要有几个好处：它可以预防 SQL 注射攻击、避免因为输入里面包含特殊字符而导致的问题、提升重复执行相似查询时的性能、并通过查询语句实现简单且可移植的数据插入操作。

假设我们想要将上千个行插入到新的表 test 里面，那么我们可能会执行像 INSERT INTO test VALUES (0, 'zero') 和 INSERT INTO test VALUES (1, 'one') 这样的查询上千次，这使得数据库必须独立地分析每条 SQL 语句。但如果我们将被插入的两个值替换为占位符，那么服务器只需要对 SQL 查询进行一次分析，然后就可以通过重复地执行这个查询来处理不同的数据了。

使用可替换参数的第二个原因和特殊字符有关。因为 SQL 使用单引号表示域（field）的末尾，所以如果我们想要插入字符串 "I don't like 1"，那么大多数 SQL 数据库都会要求我们把这个字符串写成 I don''t like1'，并且不同的特殊字符（比如反斜杠符号）在不同的数据库里面也会需要不同的转移规则。但是只要使用 HDBC，它就会帮你自动完成所有转义动作，以下展示的代码就是一个例子：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> run conn "INSERT INTO test VALUES (?, ?)" [toSql 0, toSql "zero"]
1

ghci> commit conn

ghci> disconnect conn
```

在这个示例里面，INSERT 查询包含的问号是一个占位符，而跟在占位符后面的就是要传递给占位符的各个参数。因为 run 函数的第三个参数接受的是 SqlValue 组成的列表，所以我们使用了 toSql 去将列表中的值转换为 SqlValue 。HDBC 会根据目前使用的数据库，自动地将 String "zero" 转换为正确的表示方式。

在插入大量数据的时候，可替换参数实际上并不会带来任何性能上的提升。因此，我们需要对创建 SQL 查询的过程做进一步的控制，具体的方法在接下来的一节里面就会进行讨论。

Note: 使用可替换参数

当服务器期望在查询语句的指定部分看见一个值的时候，用户才能使用可替换参数：比如在执行 SELECT 语

句的 WHERE 子句时就可以使用可替换参数；又或者在执行 INSERT 语句的时候就可以把要插入的值设置为可替换参数；但执行 run "SELECT * from ?" [toSql "tablename"] 是无法运行的。这是因为表的名字并非一个值，所以大多数数据库都不允许这种语法。因为在实际中很少人会使用这种方式去替换一个不是值的事物，所以这并不会带来什么大的问题。

20.8 预备语句

HDBC 定义了一个 prepare 函数，它可以预先准备好一个 SQL 查询，但是并不将查询语句跟具体的参数绑定。prepare 函数返回一个 Statement 值来表示已编译的查询。

在拥有了 Statement 值之后，用户就可以对它调用一次或多次 execute 函数。在对一个会返回数据的查询执行 execute 函数之后，用户可以使用任意的获取函数去取得查询所得的数据。诸如 run 和 quickQuery' 这样的函数都会在内部使用查询语句和 execute 函数；为了让用户可以更快捷妥当地执行常见的任务，像是 run 和 quickQuery' 这样的函数都会在内部使用 Statement 值和 execute 函数。当用户需要对查询的具体执行过程有更多的控制时，就可以考虑使用 Statement 而不是 run 函数。

以下代码展示了如何通过 Statement 值，在只使用一条查询的情况下插入多个值：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"

ghci> execute stmt [toSql 1, toSql "one"]
1

ghci> execute stmt [toSql 2, toSql "two"]
1

ghci> execute stmt [toSql 3, toSql "three"]
1

ghci> execute stmt [toSql 4, SqlNull]
1

ghci> commit conn

ghci> disconnect conn
```

在这段代码里面，我们创建了一个预备语句并使用 stmt 函数去调用它。我们一共执行了那个语句四次，每次都向它传递了不同的参数，这些参数会被用于替换原有查询字符串中的问号。在代码的最后，我们提交了修改并断开数据库。

为了方便地重复执行同一个预备语句，HDBC 还提供了 executeMany 函数，这个函数接受一个由多个数据

行组成的列表作为参数，而列表中的数据行就是需要调用预备语句的数据行。正如以下代码所示：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"

ghci> executeMany stmt [[toSql 5, toSql "five's nice"], [toSql 6, SqlNull]]

ghci> commit conn

ghci> disconnect conn
```

Note: 更高效的查询执行方法

在服务器上面，大多数数据库都会对 `executeMany` 函数进行优化，使得查询字符串只会被编译一次而不是多次。[52]在一次插入大量数据的时候，这种优化可以带来极为有效的性能提升。有些数据库还可以将这种优化应用到执行查询语句上面，并并非所有数据库都能做到这一点。

20.9 读取结果

本章在前面已经介绍过如何通过查询语句，将数据插入到数据库；在接下来的内容中，我们将学习从数据库里面获取数据的方法。`quickQuery'` 函数的类型和 `run` 函数非常相似，只不过 `quickQuery'` 函数返回的是一个由查询结果组成的列表而不是被改动的行数量。`quickQuery'` 函数通常与 `SELECT` 语句一起使用，正如以下代码所示：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> quickQuery' conn "SELECT * from test where id < 2" []
[[SqlString "0",SqlNull],[SqlString "0",SqlString "zero"],[SqlString "1",SqlString
↪"one"]]

ghci> disconnect conn
```

正如之前展示过的一样，`quickQuery'` 函数能够接受可替换参数。上面的代码没有使用任何可替换参数，所以在调用 `quickQuery'` 的时候，我们没有在函数调用的末尾给定任何的可替换值。`quickQuery'` 返回一个由行组成的列表，其中每个行都会被表示为 `[SqlValue]` ，而行里面的值会根据数据库返回时的顺序进行排列。在有需要的时候，用户可以使用 `fromSql` 可以将这些值转换为普通的 Haskell 类型。

因为 `quickQuery'` 的输出有一些难读，我们可以对上面的示例进行一些扩展，将它的结果格式化得更美观一些。以下代码展示了对结果进行格式化的具体方法：

[52] 对于不支持这一优化的数据库，HDBC 会通过模拟这一行为来为用户提供一致的 API ，以便执行重复的查询。

```haskell
-- file: ch21/query.hs
import Database.HDBC.Sqlite3 (connectSqlite3)
import Database.HDBC

{- | 定义一个函数，它接受一个表示要获取的最大 id 值作为参数。
函数会从 test 数据库里面获取所有匹配的行，并以一种美观的方式将它们打印到屏幕上面。 -}
query :: Int -> IO ()
query maxId =
    do -- 连接数据库
       conn <- connectSqlite3 "test1.db"

       -- 执行查询并将结果储存在 r 里面
       r <- quickQuery' conn
           "SELECT id, desc from test where id <= ? ORDER BY id, desc"
           [toSql maxId]

       -- 将每个行转换为 String
       let stringRows = map convRow r

       -- 打印行
       mapM_ putStrLn stringRows

       -- 断开与服务器之间的连接
       disconnect conn

    where convRow :: [SqlValue] -> String
          convRow [sqlId, sqlDesc] =
              show intid ++ ": " ++ desc
              where intid = (fromSql sqlId)::Integer
                    desc = case fromSql sqlDesc of
                                Just x -> x
                                Nothing -> "NULL"
          convRow x = fail $ "Unexpected result: " ++ show x
```

这个程序所做的工作和本书之前展示过的 **ghci** 示例差不多，唯一的区别就是新添加了一个 `convRow` 函数。这个函数接受来自数据库行的数据，并将它转换为一个易于打印的 `String` 值。

注意，这个程序会直接通过 `fromSql` 取出 `intid` 值，但是在处理 `fromSql sqlDesc` 的时候却使用了 `Maybe String` 。不知道你是否还记得，我们在定义表的时候，曾经将表的第一列设置为不准包含 `NULL` 值，但是第二列却没有进行这样的设置。所以，程序不需要担心第一列是否会包含 `NULL` 值，只要对第二行进行处理就可以了。虽然我们也可以使用 `fromSql` 去将第二行的值直接转换为 `String` ，但是这样一来的话，程序只要遇到 `NULL` 值就会出现异常。因此，我们需要把 SQL 的 `NULL` 转换为字符串 `"NULL"` 。虽然这个值在打印的时候可能会与字符串 `'NULL'` 出现混淆，但对于这个例子来说，这样的问题还是可以接受的。让我们尝试在 **ghci** 里面调用这个函数：

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

ghci> query 2
0: NULL
0: zero
1: one
2: two
```

20.9.1 使用语句进行数据读取操作

正如前面的《预备语句》一节所说，用户可以使用预备语句进行读取操作，并且在一些环境下，使用不同的方法从这些语句里面读取出数据将是一件非常有用的事情。像 `run` 、`quickQuery'` 这样的常用函数实际上都是使用语句去完成任务的。

为了创建一个执行读取操作的预备语句，用户只需要像之前执行写入操作那样使用 `prepare` 函数来创建预备语句，然后使用 `execute` 去执行那个预备语句就可以了。在语句被执行之后，用户就可以使用各种不同的函数去读取语句中的数据。`fetchAllRows'` 函数和 `quickQuery'` 函数一样，都返回 `[[SqlValue]]` 类型的值。除此之外，还有一个名为 `sFetchAllRows'` 的函数，它在返回每个列的数据之前，会先将它们转换为 `Maybe String` 。最后，`fetchAllRowsAL'` 函数对于每个列返回一个 `(String, SqlValue)` 二元组，其中 `String` 类型的值是数据库返回的列名。本章接下来的《数据库元数据》一节还会介绍其他获取列名的方法。

通过 `fetchRow` 函数,用户可以每次只读取一个行上面的数据,这个函数会返回 `IO (Maybe [SqlValue])` 类型的值：当所有行都已经被读取了之后，函数返回 `Nothing` ；如果还有尚未读取的行，那么函数返回一个行。

20.9.2 惰性读取

前面的《惰性 I/O》一节曾经介绍过如何对文件进行惰性 I/O 操作，同样的方法也可以用于读取数据库中的数据，并且在处理可能会返回大量数据的查询时，这种特性将是非常有用的。通过惰性地读取数据，用户可以继续使用 `fetchAllRows` 这样的方便的函数，不必再在行数据到达时手动地读取数据。通过以谨慎的方式使用数据，用户可以避免将所有结构都缓存到内存里面。

不过要注意的是，针对数据库的惰性读取比针对文件的惰性读取要负责得多。用户在以惰性的方式读取完整个文件之后，文件就会被关闭，不会留下什么麻烦的事情。另一方面，当用户以惰性的方式从数据库读取完数据之后，数据库的连接仍然处于打开状态，以便用户继续执行其他操作。有些数据库甚至支持同时发送多个查询，所以 HDBC 是无法在用户完成一次惰性读取之后就关闭连接的。

在使用惰性读取的时候，有一点是非常重要的：在尝试关闭连接或者执行一个新的查询之前，一定要先将整个数据集读取完。我们推荐你使用严格（strict）函数又或者以一行接一行的方式进行处理，从而尽量避免惰

性读取带来的复杂的交互行为。

Tip: 如果你是刚开始使用 HDBC，又或者对惰性读取的概念并不熟悉，但是又需要读取大量数据，那么可以考虑通过反复调用 `fetchRow` 来获取数据。这是因为惰性读取虽然是一种非常强大而且有用的工具，但是正确地使用它并不是那么容易的。

要对数据库进行惰性读取，只需要使用不带单引号版本的数据库函数就可以了。比如 `fetchAllRows` 就是 `fetchAllRows'` 的惰性读取版本。惰性函数的类型和对应的严格版本函数的类型一样。以下代码展示了一个惰性读取示例：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "SELECT * from test where id < 2"

ghci> execute stmt []
0

ghci> results <- fetchAllRowsAL stmt
[[("id",SqlString "0"),("desc",SqlNull)],[("id",SqlString "0"),("desc",SqlString "zero
↪")],[("id",SqlString "1"),("desc",SqlString "one")]]

ghci> mapM_ print results
[("id",SqlString "0"),("desc",SqlNull)]
[("id",SqlString "0"),("desc",SqlString "zero")]
[("id",SqlString "1"),("desc",SqlString "one")]

ghci> disconnect conn
```

虽然使用 `fetchAllRowsAL'` 函数也可以达到取出所有行的效果，但是如果需要读取的数据集非常大，那么 `fetchAllRowsAL'` 函数可能就会消耗非常多的内存。通过以惰性的方式读取数据，我们同样可以读取非常大的数据集，但是只需要使用常数数量的内存。惰性版本的数据库读取函数会把结果放到一个块里面进行求值；而严格版的数据库读取函数则会直接获取所有结果，把它们储存到内存里面，接着打印。

20.10 数据库元数据

在一些情况下，能够知道一些关于数据库自身的信息是非常有用的。比如说，一个程序可能会想要看看数据库里面目前已有的表，然后自动创建缺失的表或者对数据库的模式（schema）进行更新。而在另外一些情况下，程序可能会需要根据正在使用的数据库后端对自己的行为进行修改。

通过使用 `getTables` 函数，我们可以取得数据库目前已定义的所有列表；而 `describeTable` 函数则可以告诉我们给定表的各个列的定义信息。

调用 dbServerVer 和 proxiedClientName 可以帮助我们了解正在运行的数据库服务器，而 dbTransactionSupport 函数则可以让我们了解到数据库是否支持事务。以下代码展示了这三个函数的调用示例：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> getTables conn
["test"]

ghci> proxiedClientName conn
"sqlite3"

ghci> dbServerVer conn
"3.5.9"

ghci> dbTransactionSupport conn
True

ghci> disconnect conn
```

describeResult 函数返回一组 [(String, SqlColDesc)] 类型的二元组，二元组的第一个项是列的名字，第二个项则是与列相关的信息：列的类型、大小以及这个列能够为 NULL 等等。完整的描述可以参考 HDBC 的 API 手册。

需要注意一点是，某些数据库并不能提供所有这些元数据。在这种情况下，程序将引发一个异常。比如 Sqlite3 就不支持前面提到的 describeResult 和 describeTable 。

20.11 错误处理

HDBC 在错误出现时会引发异常，异常的类型为 SqlError 。这些异常会传递来自底层 SQL 引擎的信息，比如数据库的状态、错误信息、数据库的数字错误代号等等。

因为 **ghci** 并不清楚应该如何向用户展示一个 SqlError ，所以这个异常将导致程序停止，并打印一条没有什么用的信息。就像这样：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> quickQuery' conn "SELECT * from test2" []
*** Exception: (unknown)

ghci> disconnect conn
```

上面的这段代码因为使用了 SELECT 去获取一个不存在的表，所以引发了错误，但 **ghci** 返回的的错误信息并没有说清楚这一点。通过使用 handleSqlError 辅助函数，我们可以捕捉 SqlError 并将它重新抛出为

IOError 。这种格式的错误可以被 **ghci** 打印，但是这种格式会使得用户比较难于通过编程的方式来获取错误信息的指定部分。以下是一个使用 handleSqlError 处理异常的例子：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> handleSqlError $ quickQuery' conn "SELECT * from test2" []
*** Exception: user error (SQL error: SqlError {seState = "", seNativeError = 1,␣
↪seErrorMsg = "prepare 20: SELECT * from test2: no such table: test2"})

ghci> disconnect conn
```

这个新的错误提示具有更多信息，它甚至包含了一条说明 test2 表并不存在的消息，这比之前的错误提示有用得多了。作为一种标准实践（standard practice），很多 HDBC 程序员都将 main = handleSqlError $ do 放到程序的开头，确保所有未被捕获的 SqlError 都会以更有效的方式被打印。

除了 handleSqlError 之外，HDBC 还提供了 catchSql 和 handleSql 这两个函数，它们类似于标准的 catch 函数和 handle 函数，主要的区别在于 catchSql 和 handleSql 只会中断 HDBC 错误。想要了解更多关于错误处理的信息，可以参考本书第 19 章《错误处理》一章。

CHAPTER

TWENTYONE

第 22 章：扩展示例——WEB 客户端编程

到目前为止，我们已经了解过如何与数据库进行交互、如何进行语法分析（parse）以及如何处理错误。接下来，让我们更进一步，通过引入一个 web 客户端库来将这些知识结合在一起。

在这一章，我们将要构建一个实际的程序：一个播客下载器（podcast downloader），或者叫“播客抓取器”（podcatcher）。这个博客抓取器的概念非常简单，它接受一系列 URL 作为输入，通过下载这些 URL 来得到一些 RSS 格式的 XML 文件，然后在这些 XML 文件里面找到下载音频文件所需的 URL 。

播客抓取器常常会让用户通过将 RSS URL 添加到配置文件里面的方法来订阅播客，之后用户就可以定期地进行更新操作：播客抓取器会下载 RSS 文档，对它们进行检查以寻找音频文件的下载链接，并为用户下载所有目前尚未存在的音频文件。

Tip: 用户通常将 RSS 文件称之为“广播”（podcast）或是“广播源”（podcast feed），而每个单独的音频文件则是播客的其中一集（episode）。

为了实现具有类似功能的播客抓取器，我们需要以下几样东西：

- 一个用于下载文件的 HTTP 客户端库；
- 一个 XML 分析器；
- 一种能够记录我们感兴趣的广播，并将这些记录永久地储存起来的方法；
- 一种能够永久地记录已下载广播分集（episodes）的方法。

这个列表的后两样可以通过使用 HDBC 设置的数据库来完成，而前两样则可以通过本章介绍的其他库模块来完成。

Tip: 本章的代码是专为本书而写的，但这些代码实际上是基于 hpodder ——一个使用 Haskell 编写的播客抓取器来编写的。hpodder 拥有的特性比本书展示的播客抓取器要多得多，因此本书不太可能详细地对它进行介绍。如果读者对 hpodder 感兴趣的话，可以在 http://software.complete.org/hpodder 找到 hpodder 的源代码。

本章的所有代码都是以自成一体的方式来编写的，每段代码都是一个独立的 Haskell 模块，读者可以通过 **ghci** 独立地运行这些模块。本章的最后会写出一段代码，将这些模块全部结合起来，构成一个完整的程序。我们首先要做的就是写出构建博客抓取器需要用到的基本类型。

21.1 基本类型

为了构建播客抓取器，我们首先需要思考抓取器需要引入（important）的基本信息有那些。一般来说，抓取器关心的都是记录用户感兴趣的博客的信息，以及那些记录了用户已经看过和处理过的分集的信息。在有需要的时候改变这些信息并不困难，但是因为我们在整个抓取器里面都要用到这些信息，所以我们最好还是先定义它们：

```
-- file: ch22/PodTypes.hs
module PodTypes where

data Podcast =
    Podcast {castId :: Integer,  -- ^ 这个播客的数字 ID
             castURL :: String   -- ^ 这个播客的源 URL
            }
    deriving (Eq, Show, Read)

data Episode =
    Episode {epId :: Integer,      -- ^ 这个分集的数字 ID
             epCast :: Podcast,    -- ^ 这个分集所属播客的 ID
             epURL :: String,      -- ^ 下载这一集所使用的 URL
             epDone :: Bool        -- ^ 记录用户是否已经看过这一集
            }
    deriving (Eq, Show, Read)
```

这些信息将被储存到数据库里面。通过为每个播客和博客的每一集都创建一个独一无二的 ID ，程序可以更容易找到分集所属的播客，也可以更容易地从一个特定的播客或者分集里面载入信息，并且更好地应对将来可能会出现的“博客 URL 改变”这类情况。

21.2 数据库

接下来，我们需要编写代码，以便将信息永久地储存到数据库里面。我们最感兴趣的，就是通过数据库，将 `PodTypes.hs` 文件定义的 Haskell 结构中的数据储存到硬盘里面。并在用户首次运行程序的时候，创建储存数据所需的数据库表。

我们将使用 21 章介绍过的 HDBC 与 Sqlite 数据库进行交互。Sqlite 非常轻量，并且是自包含的（self-contained），因此它对于这个小项目来说简直是再合适不过了。HDBC 和 Sqlite 的安装方法可以在 21 章的《安装 HDBC 和驱动》一节看到。

```haskell
-- file: ch22/PodDB.hs
module PodDB where

import Database.HDBC
import Database.HDBC.Sqlite3
import PodTypes
import Control.Monad(when)
import Data.List(sort)

-- | Initialize DB and return database Connection
connect :: FilePath -> IO Connection
connect fp =
    do dbh <- connectSqlite3 fp
       prepDB dbh
       return dbh

{- | 对数据库进行设置，做好储存数据的准备。

这个程序会创建两个表，并要求数据库引擎为我们检查某些数据的一致性：

* castid 和 epid 都是独一无二的主键 (unique primary keys)，它们的值不能重复
* castURL 的值也应该是独一无二的
* 在记录分集的表里面，对于一个给定的播客 (epcast)，每个给定的 URL 或者分集 ID 只能出现一次
-}
prepDB :: IConnection conn => conn -> IO ()
prepDB dbh =
    do tables <- getTables dbh
       when (not ("podcasts" `elem` tables)) $
           do run dbh "CREATE TABLE podcasts (\
                      \castid INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,\
                      \castURL TEXT NOT NULL UNIQUE)" []
              return ()
       when (not ("episodes" `elem` tables)) $
           do run dbh "CREATE TABLE episodes (\
                      \epid INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,\
                      \epcastid INTEGER NOT NULL,\
                      \epurl TEXT NOT NULL,\
                      \epdone INTEGER NOT NULL,\
                      \UNIQUE(epcastid, epurl),\
                      \UNIQUE(epcastid, epid))" []
              return ()
       commit dbh

{- | 将一个新的播客添加到数据库里面。
```

(continues on next page)

(continued from previous page)

```
在创建播客时忽略播客的 castid ，并返回一个包含了 castid 的新对象。

尝试添加一个已经存在的播客将引发一个错误。 -}
addPodcast :: IConnection conn => conn -> Podcast -> IO Podcast
addPodcast dbh podcast =
    handleSql errorHandler $
        do -- Insert the castURL into the table.  The database
           -- will automatically assign a cast ID.
           run dbh "INSERT INTO podcasts (castURL) VALUES (?)"
               [toSql (castURL podcast)]
           -- Find out the castID for the URL we just added.
           r <- quickQuery' dbh "SELECT castid FROM podcasts WHERE castURL = ?"
                [toSql (castURL podcast)]
           case r of
             [[x]] -> return $ podcast {castId = fromSql x}
             y -> fail $ "addPodcast: unexpected result: " ++ show y
    where errorHandler e =
              do fail $ "Error adding podcast; does this URL already exist?\n"
                     ++ show e

{- | 将一个新的分集添加到数据库里面。

因为这一操作是自动执行而非用户请求执行的，我们将简单地忽略创建重复分集的请求。
这样的话，在对播客源进行处理的时候，我们就可以把遇到的所有 URL 到传给这个函数，
而不必先检查这个 URL 是否已经存在于数据库当中。

这个函数在创建新的分集时同样不会考虑如何创建新的 ID ，
因此它也没有必要去考虑如何去获取这个 ID 。 -}
addEpisode :: IConnection conn => conn -> Episode -> IO ()
addEpisode dbh ep =
    run dbh "INSERT OR IGNORE INTO episodes (epCastId, epURL, epDone) \
            \VALUES (?, ?, ?)"
            [toSql (castId . epCast $ ep), toSql (epURL ep),
             toSql (epDone ep)]
    >> return ()

{- | 对一个已经存在的播客进行修改。
根据 ID 来查找指定的播客，并根据传入的 Podcast 结构对数据库记录进行修改。 -}
updatePodcast :: IConnection conn => conn -> Podcast -> IO ()
updatePodcast dbh podcast =
    run dbh "UPDATE podcasts SET castURL = ? WHERE castId = ?"
            [toSql (castURL podcast), toSql (castId podcast)]
    >> return ()
```

(continues on next page)

(continued from previous page)

```
{- | 对一个已经存在的分集进行修改。
根据 ID 来查找指定的分集，并根据传入的 episode 结构对数据库记录进行修改。 -}
updateEpisode :: IConnection conn => conn -> Episode -> IO ()
updateEpisode dbh episode =
    run dbh "UPDATE episodes SET epCastId = ?, epURL = ?, epDone = ? \
            \WHERE epId = ?"
            [toSql (castId . epCast $ episode),
             toSql (epURL episode),
             toSql (epDone episode),
             toSql (epId episode)]
    >> return ()

{- | 移除一个播客。 这个操作在执行之前会先移除这个播客已有的所有分集。 -}
removePodcast :: IConnection conn => conn -> Podcast -> IO ()
removePodcast dbh podcast =
    do run dbh "DELETE FROM episodes WHERE epcastid = ?"
         [toSql (castId podcast)]
       run dbh "DELETE FROM podcasts WHERE castid = ?"
         [toSql (castId podcast)]
       return ()

{- | 获取一个包含所有播客的列表。 -}
getPodcasts :: IConnection conn => conn -> IO [Podcast]
getPodcasts dbh =
    do res <- quickQuery' dbh
              "SELECT castid, casturl FROM podcasts ORDER BY castid" []
       return (map convPodcastRow res)

{- | 获取特定的广播。
函数在成功执行时返回 Just Podcast ；在 ID 不匹配时返回 Nothing 。 -}
getPodcast :: IConnection conn => conn -> Integer -> IO (Maybe Podcast)
getPodcast dbh wantedId =
    do res <- quickQuery' dbh
              "SELECT castid, casturl FROM podcasts WHERE castid = ?"
              [toSql wantedId]
       case res of
         [x] -> return (Just (convPodcastRow x))
         [] -> return Nothing
         x -> fail $ "Really bad error; more than one podcast with ID"

{- | 将 SELECT 语句的执行结果转换为 Podcast 记录 -}
convPodcastRow :: [SqlValue] -> Podcast
```

(continues on next page)

(continued from previous page)

```
convPodcastRow [svId, svURL] =
    Podcast {castId = fromSql svId,
             castURL = fromSql svURL}
convPodcastRow x = error $ "Can't convert podcast row " ++ show x

{- | 获取特定播客的所有分集。 -}
getPodcastEpisodes :: IConnection conn => conn -> Podcast -> IO [Episode]
getPodcastEpisodes dbh pc =
    do r <- quickQuery' dbh
            "SELECT epId, epURL, epDone FROM episodes WHERE epCastId = ?"
            [toSql (castId pc)]
       return (map convEpisodeRow r)
    where convEpisodeRow [svId, svURL, svDone] =
              Episode {epId = fromSql svId, epURL = fromSql svURL,
                       epDone = fromSql svDone, epCast = pc}
```

PodDB 模块定义了连接数据库的函数、创建所需数据库表的函数、将数据添加到数据库里面的函数、查询数据库的函数以及从数据库里面移除数据的函数。以下代码展示了一个与数据库进行交互的 **ghci** 会话，这个会话将在当前目录里面创建一个名为 poddbtest.db 的数据库文件，并将广播和分集添加到这个文件里面。

```
ghci> :load PodDB.hs
[1 of 2] Compiling PodTypes         ( PodTypes.hs, interpreted )
[2 of 2] Compiling PodDB            ( PodDB.hs, interpreted )
Ok, modules loaded: PodDB, PodTypes.

ghci> dbh <- connect "poddbtest.db"

ghci> :type dbh
dbh :: Connection

ghci> getTables dbh
["episodes","podcasts","sqlite_sequence"]

ghci> let url = "http://feeds.thisamericanlife.org/talpodcast"

ghci> pc <- addPodcast dbh (Podcast {castId=0, castURL=url})
Podcast {castId = 1, castURL = "http://feeds.thisamericanlife.org/talpodcast"}

ghci> getPodcasts dbh
[Podcast {castId = 1, castURL = "http://feeds.thisamericanlife.org/talpodcast"}]

ghci> addEpisode dbh (Episode {epId = 0, epCast = pc, epURL = "http://www.example.com/
↪foo.mp3", epDone = False})
```

(continues on next page)

(continued from previous page)

```
ghci> getPodcastEpisodes dbh pc
[Episode {epId = 1, epCast = Podcast {castId = 1, castURL = "http://feeds.
↪thisamericanlife.org/talpodcast"}, epURL = "http://www.example.com/foo.mp3", epDone␣
↪= False}]

ghci> commit dbh

ghci> disconnect dbh
```

21.3 分析器

在实现了抓取器的数据库部分之后，我们接下来就需要实现抓取器中负责对广播源进行语法分析的部分，这个部分要分析的是一些包含着多种信息的 XML 文件，例子如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:itunes="http://www.itunes.com/DTDs/Podcast-1.0.dtd" version="2.0">
<channel>
<title>Haskell Radio</title>
<link>http://www.example.com/radio/</link>
<description>Description of this podcast</description>
<item>
<title>Episode 2: Lambdas</title>
<link>http://www.example.com/radio/lambdas</link>
<enclosure url="http://www.example.com/radio/lambdas.mp3"
type="audio/mpeg" length="10485760"/>
</item>
<item>
<title>Episode 1: Parsec</title>
<link>http://www.example.com/radio/parsec</link>
<enclosure url="http://www.example.com/radio/parsec.mp3"
type="audio/mpeg" length="10485150"/>
</item>
</channel>
</rss>
```

在这些文件里面，我们最关心的是两样东西：广播的标题以及它们的附件（enclosure）URL 。我们将使用 HaXml 工具包来对 XML 文件进行分析，以下代码就是这个工具包的源码：

```
-- file: ch22/PodParser.hs
module PodParser where
```

(continues on next page)

(continued from previous page)

```
import PodTypes
import Text.XML.HaXml
import Text.XML.HaXml.Parse
import Text.XML.HaXml.Html.Generate(showattr)
import Data.Char
import Data.List

data PodItem = PodItem {itemtitle :: String,
                        enclosureurl :: String
                        }
          deriving (Eq, Show, Read)

data Feed = Feed {channeltitle :: String,
                  items :: [PodItem]}
            deriving (Eq, Show, Read)

{- | 根据给定的广播和 PodItem ，产生一个分集。 -}
item2ep :: Podcast -> PodItem -> Episode
item2ep pc item =
    Episode {epId = 0,
             epCast = pc,
             epURL = enclosureurl item,
             epDone = False}

{- | 从给定的字符串里面分析出数据，给定的名字在有需要的时候会被用在错误消息里面。 -}
parse :: String -> String -> Feed
parse content name =
    Feed {channeltitle = getTitle doc,
          items = getEnclosures doc}

    where parseResult = xmlParse name (stripUnicodeBOM content)
          doc = getContent parseResult

          getContent :: Document -> Content
          getContent (Document _ _ e _) = CElem e

          {- | Some Unicode documents begin with a binary sequence;
          strip it off before processing. -}
          stripUnicodeBOM :: String -> String
          stripUnicodeBOM ('\xef':'\xbb':'\xbf':x) = x
          stripUnicodeBOM x = x

{- | 从文档里面提取出频道部分 (channel part)
```

(continues on next page)

(continued from previous page)

```
注意 HaXml 会将 CFilter 定义为:

> type CFilter = Content -> [Content]
-}
channel :: CFilter
channel = tag "rss" /> tag "channel"

getTitle :: Content -> String
getTitle doc =
    contentToStringDefault "Untitled Podcast"
        (channel /> tag "title" /> txt $ doc)

getEnclosures :: Content -> [PodItem]
getEnclosures doc =
    concatMap procPodItem $ getPodItems doc
    where procPodItem :: Content -> [PodItem]
          procPodItem item = concatMap (procEnclosure title) enclosure
              where title = contentToStringDefault "Untitled Episode"
                                (keep /> tag "title" /> txt $ item)
                    enclosure = (keep /> tag "enclosure") item

          getPodItems :: CFilter
          getPodItems = channel /> tag "item"

          procEnclosure :: String -> Content -> [PodItem]
          procEnclosure title enclosure =
              map makePodItem (showattr "url" enclosure)
              where makePodItem :: Content -> PodItem
                    makePodItem x = PodItem {itemtitle = title,
                                             enclosureurl = contentToString [x]}

{- | 将 [Content] 转换为可打印的字符串,
如果传入的 [Content] 为 [] ，那么向用户说明此次匹配未成功。 -}
contentToStringDefault :: String -> [Content] -> String
contentToStringDefault msg [] = msg
contentToStringDefault _ x = contentToString x

{- | 将 [Content] 转换为可打印的字符串，并且小心地对它进行反解码 (unescape)。

一个没有反解码实现的实现可以简单地定义为:

> contentToString = concatMap (show . content)
```

(continues on next page)

(continued from previous page)

```
因为 HaXml 的反解码操作只能对 Elements 使用，
我们必须保证每个 Content 都被包裹为 Element ，
然后使用 txt 函数去将 Element 内部的数据提取出来。 -}
contentToString :: [Content] -> String
contentToString =
    concatMap procContent
    where procContent x =
              verbatim $ keep /> txt $ CElem (unesc (fakeElem x))

          fakeElem :: Content -> Element
          fakeElem x = Elem "fake" [] [x]

          unesc :: Element -> Element
          unesc = xmlUnEscape stdXmlEscaper
```

让我们好好看看这段代码。它首先定义了两种类型：`PodItem` 和 `Feed` 。程序会将 XML 文件转换为 `Feed` ，而每个 `Feed` 可以包含多个 `PodItem` 。此外，程序还提供了一个函数，它可以将 `PodItem` 转换为 `PodTypes.hs` 文件中定义的 `Episode` 。

接下来，程序开始定义与语法分析有关的函数。`parse` 函数接受两个参数，一个是 `String` 表示的 XML 文本，另一个则是用于展示错误信息的 `String` 表示的名字，这个函数也会返回一个 `Feed` 。

HaXml 被设计成一个将数据从一种类型转换为另一种类型的“过滤器”，它是一个简单直接的转换操作，可以将 XML 转换为 XML 、将 XML 转换为 Haskell 数据、或者将 Haskell 数据转换为 XML 。HaXml 拥有一种名为 `CFilter` 的数据类型，它的定义如下：

```
type CFilter = Content -> [Content]
```

一个 `CFilter` 接受一个 XML 文档片段（fragments），然后返回 0 个或多个片段。`CFilter` 可能会被要求找出指定标签（tag）的所有子标签、所有具有指定名字的标签、XML 文档某一部分包含的文本，又或者其他几样东西（a number of other things）。操作符 `(/>)` 可以将多个 `CFilter` 函数组合在一起。抓取器想要的是那些包围在 `<channel>` 标签里面的数据，所以我们首先要做的就是找出这些数据。以下是实现这一操作的一个简单的 `CFilter` ：

```
channel = tag "rss" /> tag "channel"
```

当我们将一个文档传递给 `channel` 函数时，函数会从文档的顶层（top level）查找名为 `rss` 的标签。并在发现这些标签之后，寻找 `channel` 标签。

余下的程序也会遵循这一基本方法进行。`txt` 函数会从标签中提取出文本，然后通过使用 `CFilter` 函数，程序可以取得文档的任意部分。

21.4 下载

构建抓取器的下一个步骤是完成用于下载数据的模块。抓取器需要下载两种不同类型的数据：它们分别是广播的内容以及每个分集的音频。对于前者，程序需要对数据进行语法分析并更新数据库；而对于后者，程序则需要将数据写入到文件里面并储存到硬盘上。

抓取器将通过 HTTP 服务器进行下载，所以我们需要使用一个 Haskell HTTP 库。为了下载广播源，抓取器需要下载文档、对文档进行语法分析并更新数据库。对于分集音频，程序会下载文件、将它写入到硬盘并在数据库里面将该分集标记为“已下载”。以下是执行这一工作的代码：

```
-- file: ch22/PodDownload.hs
module PodDownload where
import PodTypes
import PodDB
import PodParser
import Network.HTTP
import System.IO
import Database.HDBC
import Data.Maybe
import Network.URI

{- | 下载 URL 。
函数在发生错误时返回 (Left errorMessage) ；
下载成功时返回 (Right doc) 。 -}
downloadURL :: String -> IO (Either String String)
downloadURL url =
    do resp <- simpleHTTP request
       case resp of
         Left x -> return $ Left ("Error connecting: " ++ show x)
         Right r ->
             case rspCode r of
               (2,_,_) -> return $ Right (rspBody r)
               (3,_,_) -> -- A HTTP redirect
                 case findHeader HdrLocation r of
                   Nothing -> return $ Left (show r)
                   Just url -> downloadURL url
               _ -> return $ Left (show r)
    where request = Request {rqURI = uri,
                             rqMethod = GET,
                             rqHeaders = [],
                             rqBody = ""}
          uri = fromJust $ parseURI url

{- | 对数据库中的广播源进行更新。 -}
```

(continues on next page)

(continued from previous page)

```
updatePodcastFromFeed :: IConnection conn => conn -> Podcast -> IO ()
updatePodcastFromFeed dbh pc =
    do resp <- downloadURL (castURL pc)
       case resp of
         Left x -> putStrLn x
         Right doc -> updateDB doc

    where updateDB doc =
              do mapM_ (addEpisode dbh) episodes
                 commit dbh
              where feed = parse doc (castURL pc)
                    episodes = map (item2ep pc) (items feed)

{- | 下载一个分集，并以 String 表示的形式，将储存该分集的文件名返回给调用者。
函数在发生错误时返回一个 Nothing 。 -}
getEpisode :: IConnection conn => conn -> Episode -> IO (Maybe String)
getEpisode dbh ep =
    do resp <- downloadURL (epURL ep)
       case resp of
         Left x -> do putStrLn x
                      return Nothing
         Right doc ->
             do file <- openBinaryFile filename WriteMode
                hPutStr file doc
                hClose file
                updateEpisode dbh (ep {epDone = True})
                commit dbh
                return (Just filename)
          -- This function ought to apply an extension based on the filetype
    where filename = "pod." ++ (show . castId . epCast $ ep) ++ "." ++
                     (show (epId ep)) ++ ".mp3"
```

这个函数定义了三个函数：

- `downloadURL` 函数对 URL 进行下载，并以 `String` 形式返回它；
- `updatePodcastFromFeed` 函数对 XML 源文件进行下载，对文件进行分析，并更新数据库；
- `getEpisode` 下载一个给定的分集，并在数据库里面将该分集标记为“已下载”。

> **Warning:** 这里使用的 HTTP 库并不会以惰性的方式读取 HTTP 结果，因此在下载诸如广播这样的大文件的时候，这个库可能会消耗掉大量的内容。其他一些 HTTP 库并没有这一限制。我们之所以在这里使用这个有缺陷的库，是因为它稳定、易于安装并且也易于使用。对于正式的 HTTP 需要，我们推荐使用

mini-http 库，这个库可以从 Hackage 里面获得。

21.5 主程序

最后，我们需要编写一个程序来将上面展示的各个部分结合在一起。以下是这个主模块（main module）：

```
-- file: ch22/PodMain.hs
module Main where

import PodDownload
import PodDB
import PodTypes
import System.Environment
import Database.HDBC
import Network.Socket(withSocketsDo)

main = withSocketsDo $ handleSqlError $
    do args <- getArgs
       dbh <- connect "pod.db"
       case args of
         ["add", url] -> add dbh url
         ["update"] -> update dbh
         ["download"] -> download dbh
         ["fetch"] -> do update dbh
                         download dbh
         _ -> syntaxError
    disconnect dbh

add dbh url =
    do addPodcast dbh pc
       commit dbh
    where pc = Podcast {castId = 0, castURL = url}

update dbh =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
    where procPodcast pc =
              do putStrLn $ "Updating from " ++ (castURL pc)
                 updatePodcastFromFeed dbh pc

download dbh =
    do pclist <- getPodcasts dbh
```

(continues on next page)

(continued from previous page)

```
       mapM_ procPodcast pclist
    where procPodcast pc =
              do putStrLn $ "Considering " ++ (castURL pc)
                 episodelist <- getPodcastEpisodes dbh pc
                 let dleps = filter (\ep -> epDone ep == False)
                             episodelist
                 mapM_ procEpisode dleps
          procEpisode ep =
              do putStrLn $ "Downloading " ++ (epURL ep)
                 getEpisode dbh ep

syntaxError = putStrLn
  "Usage: pod command [args]\n\
  \\n\
  \pod add url       Adds a new podcast with the given URL\n\
  \pod download      Downloads all pending episodes\n\
  \pod fetch         Updates, then downloads\n\
  \pod update        Downloads podcast feeds, looks for new episodes\n"
```

这个程序使用了一个非常简单的命令行解释器，并且这个解释器还包含了一个用于展示命令行语法错误的函数，以及一些用于处理不同命令行参数的小函数。

通过以下命令，可以对这个程序进行编译：

```
ghc --make -O2 -o pod -package HTTP -package HaXml -package network \
    -package HDBC -package HDBC-sqlite3 PodMain.hs
```

你也可以通过《创建包》一节介绍的方法，使用 Cabal 文件来构建这个项目：

```
-- ch23/pod.cabal
Name: pod
Version: 1.0.0
Build-type: Simple
Build-Depends: HTTP, HaXml, network, HDBC, HDBC-sqlite3, base

Executable: pod
Main-Is: PodMain.hs
GHC-Options: -O2
```

除此之外，我们还需要一个简单的 `Setup.hs` 文件：

```
import Distribution.Simple
main = defaultMain
```

如果你是使用 Cabal 进行构建的话，那么只要运行以下代码即可：

```
runghc Setup.hs configure
runghc Setup.hs build
```

程序的输出将被放到一个名为 dist 的文件及里面。要将程序安装到系统里面的话，可以运行 run runghc Setup.hs install 。

CHAPTER

TWENTYTWO

第 23 章：用 GTK2HS 进行图形界面编程

在本书前面的内容中，我们开发了一系列简单的文本工具。尽管这些工具提供的文本接口在大部分情况下都能令人满意，但在某些情况下，我们还是需要用到图形用户界面（GUI）。有很多可供 Haskell 使用的图形界面工具。在这一章中，我们将使用其中的一个，gtk2hs[53] 。

22.1 安装 gtk2hs

在我们研究如何使用 gtk2hs 工作前，需要先安装它。在大多数 Linux，BSD，或者其它 POSIX 平台，有已经打包好的 gtk2hs 安装包。你一般需要安装 GTK+ 开发环境，Glade，和 gtk2hs。安装的细节不同版本各有不同。

使用 Windows 和 Mac 的开发者应该查阅 gtk2hs 下载站 。从下载 gtk2hs 开始，然后你需要 Glade version 3 的版本。Mac 开发者可以从 macports 找到，Windows 开发者应该查阅 sourceforge 。

22.2 概览 GTK+ 开发栈

在深入代码前，让我们暂停一会考虑一下我们将要使用的系统的架构。首先，我们使用的 GTK+ 是一个跨平台的，用 C 语言来实现的 GUI 工具集。可以跑在 Windows，Mac，Linux，BSD 等等操作系统上。Gnome 桌面环境的下层就是用了它。

然后，我们使用的 Glade 是一个用户界面设计工具，可以让你用图形化的方式来设计你应用的窗口和对话框等。Glade 把你的设计保存在 XML 文件中，你的应用程序会在运行时加载这些 XML 文件。

最后使用的是 gtk2hs。这是一个 GTK+，Glade 以及一些依赖库的 Haskell 绑定。它只是很多编程语言对 GTK+ 绑定的一种。

[53] 还有很多别的选择，除了 gtk2hs 之外，wxHaskell 也是非常杰出的跨平台图形界面工具集。

22.3 使用 Glade 进行用户界面设计

在这一小节中，我们将为 第 22 章中开发的播客下载器 开发一个图形界面版本。我们的第一项任务就是在 Glade 中设计图形界面。当我们完成设计时，我们将编写 Haskell 代码集成进应用中。

因为这是一本 Haskell 书，而不是一本图形界面设计书，所以我们快速带过前面的步骤。需要更多关于使用 Glade 设计图形界面的信息，你可以参考下面的资源：

- Glade 主页 ，包含了 Glade 的文档。
- GTK+ 的主页 包含了不同窗口小部件的信息。参考文档章节，然后进入稳定版 GTK 文档的区域。
- gtk2hs 主页 也有很有用的文档，包含了 gtk2hs 的 API 参考和一个 glade 的教程。

22.4 Glade 基本概念

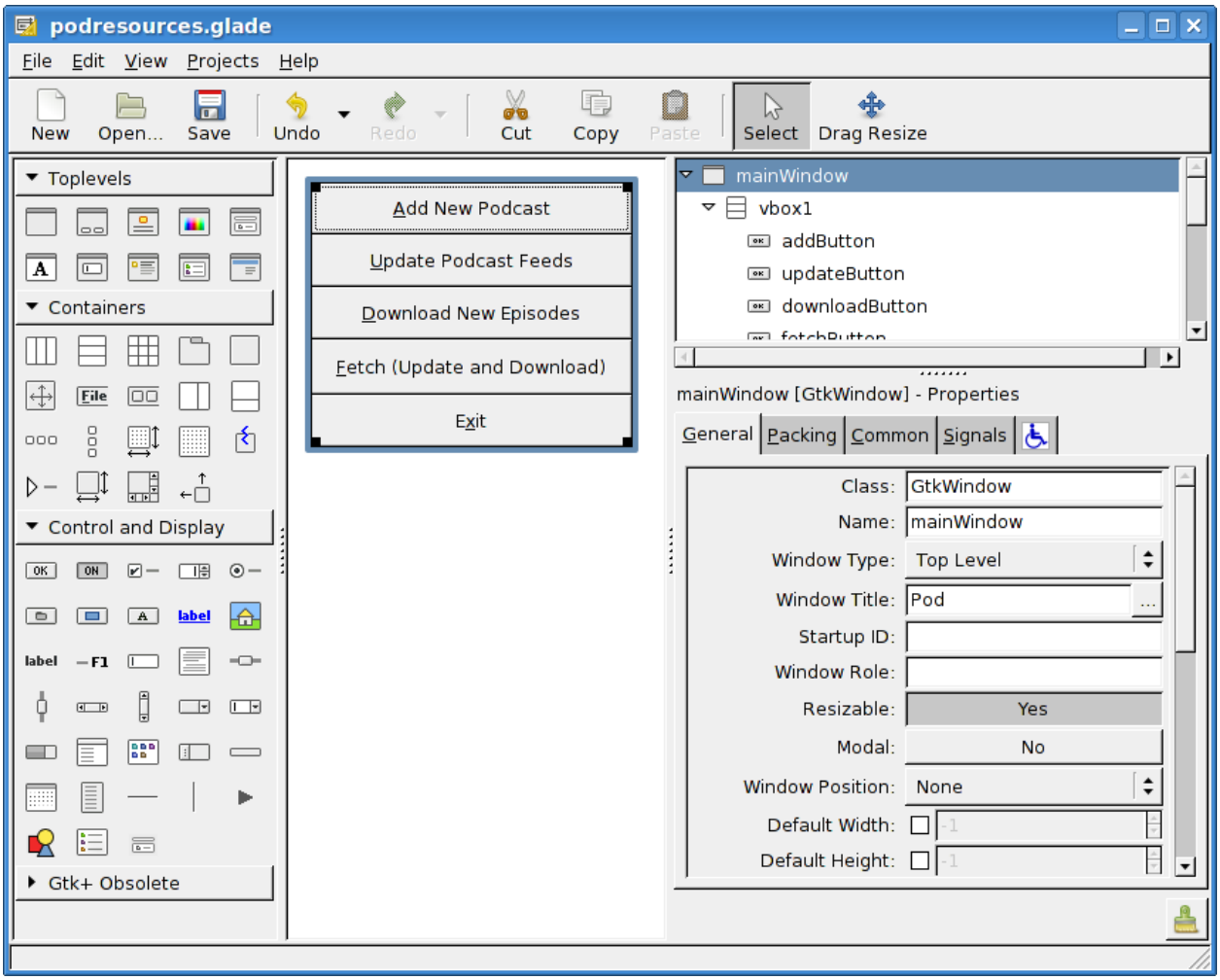
Glade 是一个图形界面设计工具。让我们用图形界面的方式来设计图形界面。我们可以使用一堆 GTK+ 的函数来创建窗口组件，但更简单的方式是使用 Glade。

我们要使用 GTK+ 来开发的基础的东西叫**窗口小部件**。一个窗口小部件代表了 GUI 的一部分，可能这个小部件还包含了别的小部件。比如一些小部件包含了一个窗口，对话框，按钮，以及带文字的按钮。

我们在 Glade 中初始化小部件树，最高级的窗口在树的根部。你可以把 Glade 和小部件想象成 HTML：你可以像 table 布局一样排列组件，然后设置 padding 规则，然后组织完整的继承逻辑。

Glade 把组件描述保存在 XML 文件中。我们的程序在运行时加载这些文件。我们通过指定名字从 Glade 运行时库中加载对应的组件。

下面是一个使用 Glade 设计我们应用主界面的截图：



在本书的附加下载材料中，你可以找到完整的 Glade XML 文件 (podresources.glade)，然后你可以加载它或者按你希望的修改它。

22.5 基于事件编程

GTK+ 就像其它的 GUI 工具集一样，是事件驱动的工具集。这就意味着，我们不是要显示一个对话框，然后等待用户点击按钮，相反的，我们是要告诉 gtk2hs 当点击某个按钮时要调用什么函数，而不是坐在那儿等待点击对话框。

这跟传统的控制台编程是不同的模式。一个 GUI 程序应该有多个窗口打开，但坐在那儿编写代码来组合输入特性组合的打开窗口是一个复杂的命题。

事件驱动编程很好的补充了 Haskell。就像我们在书中一遍又一遍的讨论，函数是语言通过传递函数来繁荣昌盛。所以当某些事件发生时，我们将调用传给 gtk2hs 的函数。这种做法被称为回调函数。

GTK+ 程序的核心是主循环 (**main loop**)。这部分程序等待用户或者程序命令运行，然后执行它们。GTK+ 的主循环由 GTK+ 来掌控。对于我们来说，它看起来就像一个 I/O 操作，我们执行命令，然后知道主循环执行

到我们的命令才返回结果 (即不立即返回)。

因为主循环负责响应一切的点击鼠标重绘窗口事件，所以它必须始终是可用状态的。我们不能执行一个很耗时的任务 – 比如在主循环中下载一个播客节目。这会使得 GUI 出于无法响应的状态，所有的动作比如点击取消按钮将不会被及时的执行。

所以，我们将使用多线程来处理这些耗时任务。更多关于多线程的信息请查看 [本书第 24 章]()。现在，你只需要知道我们将使用 *forkIO* 来创建新的线程来处理像下载播客的节目单和节目。对于很快的任务，像是添加一个播客到数据库里，就不用新开一个线程来处理了，因为它快到用户无法感知。

22.6 初始化 GUI

第一步我们先来初始化我们的 GUI 项目。我们将创建一个小文件 *PodLocalMain.hs* 然后加载 *PodMain* 然后把它的路径传到 *podresources.glade*，这个被 Glade 保存的 XML 文件提供了我们的 GUI 组件的信息，这么做的原因我们将在 [使用 Cabal]() 这一章中解释。

```
-- file: ch23/PodLocalMain.hs
module Main where
import qualified PodMainGUI
main = PodMainGUI.main "podresources.glade"
```

现在让我们来考虑一下 *PodMainGUI.hs* 该怎么写。这个文件是我们在 第 22 章 的例子基础上唯一要修改的文件，我们修改它以便于让它可以作为 GUI 工作。我们先把 *PodMainGUI.hs* 重命名为 *PodMain.hs* 使它更加清晰。

```
-- file: ch23/PodMainGUI.hs
module PodMainGUI where

import PodDownload
import PodDB
import PodTypes
import System.Environment
import Database.HDBC
import Network.Socket(withSocketsDo)

-- GUI libraries

import Graphics.UI.Gtk hiding (disconnect)
import Graphics.UI.Gtk.Glade

-- Threading

import Control.Concurrent
```

PodMainGUI.hs 的第一部分跟非 GUI 版本基本相同。我们引入三个附加的组件，首先，我们引入 *Graphics.UI.Gtk*，它提供了我们需要使用的大部分 GTK+ 函数。这个模块和叫 *Database.HDBC* 的模块都提供了一个函数叫 *disconnect*。我们将使用 HDBC 版本提供的，而不是 GTK+ 版本的，所以我们不从 *Graphics.UI.Gtk* 导入这个函数。*Graphics.UI.Gtk.Glade* 包含了需要加载的函数且可以跟我们的 Glade 文件协同工作。

然后我们引入 *Control.Concurrent*，它提供了多线程编程的基础。我们从这里开始将使用少量的函数来描述上面提到的功能。接下来，让我们定义一个类型来存储我们的 GUI 信息。

```
-- file: ch23/PodMainGUI.hs
-- | Our main GUI type
data GUI = GUI {
      mainWin :: Window,
      mwAddBt :: Button,
      mwUpdateBt :: Button,
      mwDownloadBt :: Button,
      mwFetchBt :: Button,
      mwExitBt :: Button,
      statusWin :: Dialog,
      swOKBt :: Button,
      swCancelBt :: Button,
      swLabel :: Label,
      addWin :: Dialog,
      awOKBt :: Button,
      awCancelBt :: Button,
      awEntry :: Entry}
```

我们的新 GUI 类型存储所有我们在程序中需要关心的组件。即使是规模较大的程序，通常也不会用到这么单一而庞大的类型。但是对于这个小示例来说，单一类型更容易在函数之间传递，并使得我们可以随时拿到所需的信息，因此我们不妨在这里开个特例。

这个类型记录中，我们有 *Window*(顶层窗口)，*Dialog*(对话框窗口)，*Button*(可被点击的按钮)，*Label*(文本)，以及 *Entry*(用户输入文本的地方)。让我们马上看一下 *main* 函数：

```
-- file: ch23/PodMainGUI.hs
main :: FilePath -> IO ()
main gladepath = withSocketsDo $ handleSqlError $
    do initGUI                  -- Initialize GTK+ engine

       -- Every so often, we try to run other threads.
       timeoutAddFull (yield >> return True)
                      priorityDefaultIdle 100

       -- Load the GUI from the Glade file
       gui <- loadGlade gladepath
```

(continues on next page)

(continued from previous page)

```
        -- Connect to the database
        dbh <- connect "pod.db"

        -- Set up our events
        connectGui gui dbh

        -- Run the GTK+ main loop; exits after GUI is done
        mainGUI

        -- Disconnect from the database at the end
        disconnect dbh
```

注意这里的 *main* 函数的类型与通常的优点区别，因为它被 **PodLocalMain.hs** 中的 *main* 调用。我们一开始调用了 *initGUI* 来初始化 GTK+ 系统。接下来我们调用了 *timeoutAddFull*。这个调用只有在进行多线程 GTK+ 编程才需要。它告诉 GTK+ 的主循环时不时地给其它线程机会去执行。

之后，我们调用 *loadGlade* 函数 (见下面的代码) 来加载我们的 Glade XML 文件。接着，我们连接数据库并调用 *connectGui* 函数来设置我们的回调函数。然后，我们启动 GTK+ 主循环。我们期望它在 *mainGUI* 返回之前可能执行数分钟，数小时，甚至是数天。当 *mainGUI* 返回时，它表示用户已经关闭了主窗口或者是点击了退出按钮。这时，我们关闭数据库连接并且结束程序。现在，来看看 *loadGlade* 函数：

```
-- file: ch23/PodMainGUI.hs
loadGlade gladepath =
    do -- Load XML from glade path.
       -- Note: crashes with a runtime error on console if fails!
       Just xml <- xmlNew gladepath

       -- Load main window
       mw <- xmlGetWidget xml castToWindow "mainWindow"

       -- Load all buttons

       [mwAdd, mwUpdate, mwDownload, mwFetch, mwExit, swOK, swCancel,
        auOK, auCancel] <-
           mapM (xmlGetWidget xml castToButton)
           ["addButton", "updateButton", "downloadButton",
            "fetchButton", "exitButton", "okButton", "cancelButton",
            "auOK", "auCancel"]

       sw <- xmlGetWidget xml castToDialog "statusDialog"
       swl <- xmlGetWidget xml castToLabel "statusLabel"

       au <- xmlGetWidget xml castToDialog "addDialog"
```

(continues on next page)

(continued from previous page)

```
      aue <- xmlGetWidget xml castToEntry "auEntry"

      return $ GUI mw mwAdd mwUpdate mwDownload mwFetch mwExit
             sw swOK swCancel swl au auOK auCancel aue
```

这个函数从调用 *xmlNew* 开始来加载 Glade XML 文件。当发生错误时它返回 *Nothing*。当执行成功时我们用模式匹配来获取结果值。如果失败，那么命令行将会有异常被输出；这是这一章结束的练习题之一。

现在 Glade XML 文件已经被加载了，你将看到一大堆 *xmlGetWidget* 的函数调用。这个 Glade 函数被用来加载一个组件的 XML 定义，同时返回一个 GTK+ 组件类型给对应的组件。我们将传给这个函数一个值来指出我们期望的 GTK+ 类型 – 当类型不匹配的时候会得到一个运行时错误。

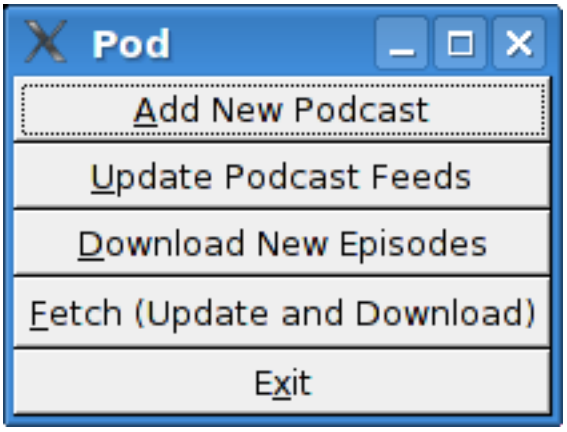
我们开始在主窗口创建一个组件。它在 XML 里被定义为 *mainWindow* 并被加载，然后存到 *mw* 这个变量里。接着我们通过模式匹配和 *mapM* 来加载所有的按钮。然后，我们有了两个对话框，一个标签，和一个被加载的实体。最后，我们使用所有的这些来建立 GUI 类型并且返回。接下来，我们设置回调函数作为事件控制器：

```
-- file: ch23/PodMainGUI.hs
connectGui gui dbh =
    do -- When the close button is clicked, terminate GUI loop
       -- by calling GTK mainQuit function
       onDestroy (mainWin gui) mainQuit

       -- Main window buttons
       onClicked (mwAddBt gui) (guiAdd gui dbh)
       onClicked (mwUpdateBt gui) (guiUpdate gui dbh)
       onClicked (mwDownloadBt gui) (guiDownload gui dbh)
       onClicked (mwFetchBt gui) (guiFetch gui dbh)
       onClicked (mwExitBt gui) mainQuit

       -- We leave the status window buttons for later
```

我们通过调用 *onDestroy* 来开始调用 *connectGui* 函数。这意味着当某个人点击了操作系统的关闭按钮 (在 Windows 或者 Linux 上是标题栏上面的 X 标志，在 Mac OS X 上是红色的圆点)，我们在主窗口调用 *mainQuit* 函数。*mainQuit* 关闭所有的 GUI 窗口然后结束 GTK+ 主循环。

接下来，我们调用 *onClicked* 对五个不同按钮的点击来注册事件控制器。对于每个按钮，当用户通过键盘选择按钮时控制器同样会被触发。点击这些按钮将会调用比如 *guiAdd* 这样的函数，传递 GUI 记录以及一个对数据库的调用。

现在，我们完整地定义了我们 GUI 播客的主窗口。它看起来像下面的截图。



22.7 增加播客窗口

现在，我们已经完整介绍了主窗口，让我们来介绍别的需要呈现的窗口，从增加播客窗口开始。当用户点击增加一个播客的时候，我们需要弹出一个对话框来提示输入播客的 URL。我们已经在 Glade 中定义了这个对话框，所以接下来需要做的就是设置它：

```
-- file: ch23/PodMainGUI.hs
guiAdd gui dbh =
    do -- Initialize the add URL window
       entrySetText (awEntry gui) ""
       onClicked (awCancelBt gui) (widgetHide (addWin gui))
       onClicked (awOKBt gui) procOK

       -- Show the add URL window
       windowPresent (addWin gui)
       where procOK =
           do url <- entryGetText (awEntry gui)
              widgetHide (addWin gui) -- Remove the dialog
              add dbh url             -- Add to the DB
```

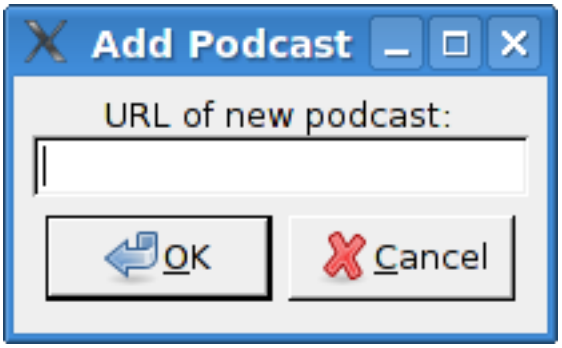
我们通过调用 *entrySetText* 来设置输入框 (用户填写播客 URL 的地方) 的内容，让我们先设置为一个空字符串。这是因为这个组件在我们程序的生命周期中会被复用，所以我们不希望用户最后添加的 URL 被留在输入框中。接下来，我们设置对话框中两个按钮的事件。如果用户点击取消按钮，我们就调用 *widgetHide* 函数来从屏幕上移除这个对话框。如果用户点击了 OK 按钮，我们调用 *procOK*。

procOK 先获取输入框中提供的 URL。接下来，它用 *widgetHide* 函数来隐藏输入框，最后它调用 *add* 函数来往输入库里增加 URL。这个 *add* 函数跟我们没有 GUI 版本的程序中的一样。

我们在 *guiAdd* 里做的最后一件事是弹出窗口，这个通过调用 *windowPresent* 来做，这个函数功能正好跟 *widgetHide* 相反。

注意 *guiAdd* 函数会立即返回。它只是设置组件并且让输入框显示出来；它不会阻塞自己等待输入。下图显示

了对话框看起来是什么样的。



22.8 长时间执行的任务

在主窗口的按钮中，有三个点击之后的任务是需要等一会才会完成的，这三个分别是更新 (update)，下载 (download)，已经获取 (fetch)。当这些操作发生时，我们希望做两件事：提供给用户当前操作的进度，以及可以取消当前正在执行的操作的功能。

因为这些操作都非常类似，所以可以提供一个通用的处理方式来处理这些交互。我们已经在 Glade 文件中定义了一个状态窗口组件，这个组件将会被这三个操作使用。在我们的 Haskell 代码中，我们定义了一个通用的 *statusWindow* 函数来同时被这三个操作使用。

statusWindow 需要 4 个参数：GUI 信息，数据库信息，表示该窗口标题的字符串，一个执行操作的函数。这个函数自己将会被当做参数传递给汇报进度的那个函数。下面是代码：

```
-- file: ch23/PodMainGUI.hs
statusWindow :: IConnection conn =>
                GUI
             -> conn
             -> String
             -> ((String -> IO ()) -> IO ())
             -> IO ()
statusWindow gui dbh title func =
    do -- Clear the status text
       labelSetText (swLabel gui) ""

       -- Disable the OK button, enable Cancel button
       widgetSetSensitivity (swOKBt gui) False
       widgetSetSensitivity (swCancelBt gui) True

       -- Set the title
       windowSetTitle (statusWin gui) title

       -- Start the operation
```

(continues on next page)

(continued from previous page)

```
      childThread <- forkIO childTasks

      -- Define what happens when clicking on Cancel
      onClicked (swCancelBt gui) (cancelChild childThread)

      -- Show the window
      windowPresent (statusWin gui)
  where childTasks =
            do updateLabel "Starting thread..."
               func updateLabel
               -- After the child task finishes, enable OK
               -- and disable Cancel
               enableOK

         enableOK =
    do widgetSetSensitivity (swCancelBt gui) False
       widgetSetSensitivity (swOKBt gui) True
       onClicked (swOKBt gui) (widgetHide (statusWin gui))
       return ()

updateLabel text =
    labelSetText (swLabel gui) text
cancelChild childThread =
    do killThread childThread
       yield
       updateLabel "Action has been cancelled."
       enableOK
```

这个函数一开始清理了它上次运行时的标签内容。接下来，我们使 OK 按钮不可被点击 (变灰色)，同时使取消按钮可被点击。当操作在进行中时，点击 OK 按钮不起任何作用，当操作结束后，点击取消按钮不起任何作用。

接着，我们设置窗口的标题。这个标题会出现在系统显示的窗口标题栏中。最后，我们启动一个新的线程 (通过调用 *childTasks*)，然后保存这个线程 ID。然后，我们定义当用户点击取消按钮之后的行为 – 我们调用 *cancelChild* 传入线程 ID。最后，我们调用 *windowPresent* 来显示进度窗口。

在子任务中，我们显示一条信息来说明我们正在启动线程。然后我们调用真正的工作函数，传入 *updateLabel* 函数来显示状态信息。注意命令行版本的程序可以传入 *putStrLn* 函数。

最后，当工作函数退出后，我们调用 *enableOK* 函数。这个函数使取消按钮变得不可被点击，并且让 OK 按钮变得可点击，顺便定义在点击 OK 按钮时候的行为 – 让进度窗口消失。

updateLabel 简单地调用在标签组件上的 *labelSetText* 函数来更新标签显示信息。最后，*cancelChild* 函数被调用来杀死执行任务的线程，更新标签信息，并且使 OK 按钮可被点击。

现在我们需要的基础功能都就位了。他们看起来像下面这样：

```
-- file: ch23/PodMainGUI.hs
guiUpdate :: IConnection conn => GUI -> conn -> IO ()
guiUpdate gui dbh =
    statusWindow gui dbh "Pod: Update" (update dbh)

guiDownload gui dbh =
    statusWindow gui dbh "Pod: Download" (download dbh)

guiFetch gui dbh =
    statusWindow gui dbh "Pod: Fetch"
                     (\logf -> update dbh logf >> download dbh logf)
```

我们只给出了第一个函数的类型，但是其实三个函数类型都是相同的，Haskell 可以通过类型推断来推导出它们的类型。注意我们实现的 *guiFetch* 函数，我们不用调用两次 *statusWindow* 函数，相反，我们在它的操作中组合函数来实现。

最后一点构成三个函数的部分是真正做想要的工作。*add* 函数是命令行版本直接拿过来的，没有任何修改。*update* 和 *download* 函数仅仅修改了一小部分 – 通过一个记录函数 (logging function) 来取代调用 *putStrLn* 函数来更新进度状态。

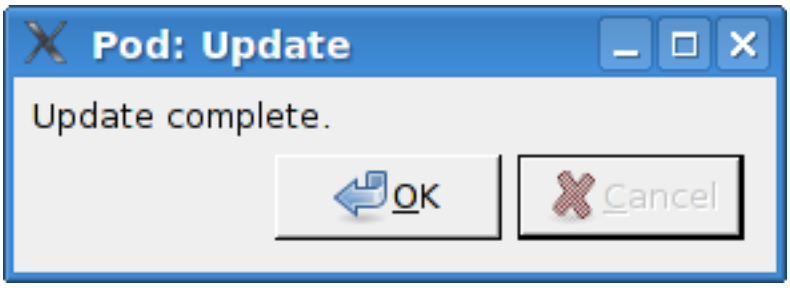
```
-- file: ch23/PodMainGUI.hs
add dbh url =
    do addPodcast dbh pc
       commit dbh
    where pc = Podcast {castId = 0, castURL = url}

update :: IConnection conn => conn -> (String -> IO ()) -> IO ()
update dbh logf =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
       logf "Update complete."
    where procPodcast pc =
              do logf $ "Updating from " ++ (castURL pc)
                 updatePodcastFromFeed dbh pc

download dbh logf =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
       logf "Download complete."
    where procPodcast pc =
              do logf $ "Considering " ++ (castURL pc)
                 episodelist <- getPodcastEpisodes dbh pc
                 let dleps = filter (\ep -> epDone ep == False)
```

(continues on next page)

(continued from previous page)

```
                          episodelist
                mapM_ procEpisode dleps
        procEpisode ep =
            do logf $ "Downloading " ++ (epURL ep)
               getEpisode dbh ep
```

下图展示了更新操作执行完成的结果是什么样子的。



22.9 使用 Cabal

我们通过一个 Cabal 文件来构建我们命令行版本的项目。我们需要做一些修改来让它支持构建我们 GUI 版本的项目。首先我们需要增加 gtk2hs 包的依赖。当然还有 Glade XML 文件的问题。

在前面，我们写了 **PodLocalMain.hs** 文件来假定配置文件叫 *podresources.glade*，然后把它存到当前目录下。但是对于真正的系统安装来说，我们不能做这个假设。而且，不同的操作系统会把文件放到不同的路径下。

Cabal 提供了处理这个问题的方法。它自动生成一个模块，这个模块可以通过导出函数来查询环境变量。我们必须在 Cabal 依赖文件里增加一行 *Data-files*。这个文件名称表示了所有需要一同安装的数据文件。然后，Cabal 将会导出一个 *Paths_pod* 模块 (pod 部分来自 Cabal 文件中的 Name 行)，我们可以使用这个模块来在运行时查看文件路径。下面是我们新的 Cabal 依赖文件：

```
-- ch24/pod.cabal
name: pod
Version: 1.0.0
Build-type: Simple
Build-Depends: HTTP, HaXml, network, HDBC, HDBC-sqlite3, base,
               gtk, glade
Data-files: podresources.glade

Executable: pod
Main-Is: PodCabalMain.hs
GHC-Options: -O2
```

当然还有 *PodCabalMain.hs*：

```haskell
-- file: ch23/PodCabalMain.hs
module Main where

import qualified PodMainGUI
import Paths_pod(getDataFileName)

main =
    do gladefn <- getDataFileName "podresources.glade"
       PodMainGUI.main gladefn
```

22.10 练习

1. 如果调用 *xmlNew* 返回了 *Nothging*，显示一个图形话的出错信息。
2. 修改项目来实现在同一个代码仓库既可以使用图形界面的方式，又可以选择命令行模式来运行程序。提示：把通用代码移出 *PodMainGUI.hs*，然后创建两个 *main* 模块，一个为图形界面服务，一个为命令行服务。
3. 为什么 *guiFetch* 函数组合工作函数而不是调用 *statusWindow* 两次？

CHAPTER

TWENTYTHREE

第 24 章：并发和多核编程

在撰写此书时，CPU 架构的景观正以几十年来最快的速度发生变化。

23.1 定义并发和并行

一个并发程序需要同时处理多个互不相关的任务。考虑一下游戏服务器的例子：典型做法是将数十个组件组合起来，其中的每一个都与外部有复杂交互。可能其中某个组件负责多个用户间聊天；另外一些负责处理玩家的输入，并且将更新后的状态返回给客户端；同时还有其他程序执行物理计算。

并发程序的正确运转并不需要多核，尽管多核可以提高执行效率和响应速度。

相比之下，一个并行程序仅解决一个单独的问题。假设一个金融模型尝试计算并预测下一分钟某支股票的价格波动。如果想在某个交易所列出的所有股票上执行这个模型，例如计算一下那些股票应该买入或卖出，我们希望在五百个核上可以比仅有一个核的时候跑得更快。这表明，并行程序通常不需要通过多核来保证正确性。

另一个有效区分并行和并发的点在于他们如何与外部世界交互。由定义，并发程序连续不断的处理网络协议和数据库之类的东西。典型的并行程序可能更专注：其接收流入的数据，咀嚼一会儿（间或有点 I/O），然后将需要返回的数据流吐出来。

许多传统编程语言进一步模糊了并发和并行之间已经难以辨认的边界，这些语言强制程序员使用相同的基础设施投监这两种程序。

本章将涉及在单个操作系统进程内进行并发和并行编程。

23.2 用线程进行并发编程

作为并发编程的基础，大多数语言提供了创建多个多线程的方法。`Haskell` 也不例外，尽管使用 `Haskell` 进行线程编程看起来和其他语言有些不同。

In Haskell, a thread is an IO action that executes independently from other threads. To create a thread, we import the Control.Concurrent module and use the forkIO function Haskell 中，线程是互相独立的 IO 动作。为创建线程，需要导入 Control.Concurrent 模块并使用其中的 forkIO 函数

```
ghci> :m +Control.Concurrent
ghci> :t forkIO
forkIO :: IO () -> IO ThreadId
ghci> :m +System.Directory
ghci> forkIO (writeFile "xyzzy" "seo craic nua!") >> doesFileExist "xyzzy"
True
```

新线程几乎立即开始执行，创建它的线程同时继续向下执行。新线程将在它的 IO 动作结束后停止执行。

23.2.1 线程的不确定性

GHC 的运行时组件并不按特定顺序执行多个线程。所以，上面的例子中，文件 xyzzy 的创建时间在初始线程检查其是否存在之前或之后都有可能。如果删除 xyzzy 并且再执行一次，我们可能得到完全相反的结果。

23.2.2 隐藏延迟

假设我们要将一个大文件压缩并写入磁盘，但是希望快速处理用户输入以使他们感觉程序是立即响应的。如果使用 forkIO 来开启一个单独的线程去写文件，这样就可以同时做这两件事。

```
-- file: ch24/Compressor.hs
import Control.Concurrent (forkIO)
import Control.Exception (handle)
import Control.Monad (forever)
import qualified Data.ByteString.Lazy as L
import System.Console.Readline (readline)

-- http://hackage.haskell.org/ 上的 zlib 包提供了压缩功能
import Codec.Compression.GZip (compress)

main = do
    maybeLine <- readline "Enter a file to compress> "
    case maybeLine of
      Nothing -> return ()       -- 用户输入了 EOF
      Just "" -> return ()       -- 不输入名字按 “想要退出” 处理
      Just name -> do
           handle
             (print :: (SomeException->IO ()))
             $ do
                 content <- L.readFile name
```

(continues on next page)

(continued from previous page)

```
                forkIO (compressFile name content)
                return ()
          main
  where compressFile path = L.writeFile (path ++ ".gz") . compress
```

[Forec 译注：原著代码稍微有点瑕疵，上面是修正后的版本。此外，在部分 GHC 中执行上述程序可能遇到 System.Console.Readline 包无法导入的情况。Readline 是 GNU readline 库的 Haskell 绑定，你可以在 http://www.gnu.org/software/readline/ 获取稳定版本的 readline 库。另一种解决方法是使用 haskeline 代替 readline ，它的文档位于 http://hackage.haskell.org/package/haskeline-0.7.3.1/docs/System-Console-Haskeline.html ：

```
import Control.Concurrent (forkIO)
import Control.Exception (handle, SomeException)
import qualified Data.ByteString.Lazy as L
import System.Console.Haskeline (runInputT, defaultSettings, getInputLine)

-- Provided by the 'zlib' package on http://hackage.haskell.org/
import Codec.Compression.GZip (compress)

main :: IO ()
main = do
  maybeLine <- runInputT defaultSettings $ getInputLine "Enter a file to compress> "
  case maybeLine of
    Nothing -> return () -- user entered EOF
    Just "" -> return () -- treat no name as "want to quit"
    Just name -> do
      handle (print :: SomeException -> IO ()) $ do
        content <- L.readFile name
        forkIO (compressFile name content)
        return ()
      main
```

]

因为使用了惰性的 ByteString I/O ，主线程中做仅仅是打开文件。真正读取文件内容发生在子线程中。

当用户输入的文件名并不存在时将发生异常，handle (print :: (SomeException-> IO ())) 是一个低成本的打印错误信息的方式。

23.3 线程间的简单通信

在两个线程之间共享信息最简单的方法是，让它们使用同一个变量。上面文件压缩的例子中，main 线程与子线程共享了文件名和文件内容。Haskell 的数据默认是不可变的，所以这样共享不会有问题，两个线程

都无法修改另一个线程中的文件名和文件内容。

线程经常需要和其他线程进行活跃的通信。例如，GHC 没有提供查看其他线程是否还在执行、执行完毕、或者崩溃的方法[54] 。可是，其提供了同步变量类型，MVar ，我们可以通过它自己实现上述功能。

MVar 的行为类似一个单元素的箱子：其可以为满或空。将一些东西扔进箱子，使其填满，或者从中拿出一些东西，使其变空。

```
ghci> :t putMVar
putMVar :: MVar a -> a -> IO ()
ghci> :t takeMVar
takeMVar :: MVar a -> IO a
```

尝试将一个值放入非空的 MVar ，将会导致线程休眠直到其他线程从其中拿走一个值使其变空。类似的，如果尝试从一个空的 MVar 取出一个值，线程也将休眠，直到其他线程向其中放入一个值。

```
-- file: ch24/MVarExample.hs
import Control.Concurrent

communicate = do
  m <- newEmptyMVar
  forkIO $ do
    v <- takeMVar m
    putStrLn ("received " ++ show v)
  putStrLn "sending"
  putMVar m "wake up!"
```

newEmptyMVar 函数的作用从其名字一目了然。要创建一个初始状态非空的 MVar ，需要使用 newMVar 。

```
ghci> :t newEmptyMVar
newEmptyMVar :: IO (MVar a)
ghci> :t newMVar
newMVar :: a -> IO (MVar a)
```

在 ghci 运行一下上面例子。

```
ghci> :load MVarExample
[1 of 1] Compiling Main             ( MVarExample.hs, interpreted )
Ok, modules loaded: Main.
ghci> communicate
sending
rece
```

如果有使用传统编程语言编写并发程序的经验，你会想到 MVar 有助于实现两个熟悉的效果。

[54] 在稍后将展示，GHC 的线程异常轻量。如果运行时提供检查每个线程状态的方法，每个线程的开销将增加，哪怕永远不会用到这些信息。

- 从一个线程向另一个线程发送消息，例如：一个提醒。
- 对线程间共享的可变数据提供互斥。在数据没有被任何线程使用时，将其放入 `MVar`，某线程需要读取或改变它时，将其临时从中取出。

23.4 主线程等待其他线程

GHC 的运行时系统对主线程的控制与其他线程不同。主线程结束时，运行时系统认为整个程序已经跑完了。其他没有执行完毕的线程，会被强制终止。

所以，如果线程执行时间非常长，且必须不被杀死，必须对主线程做特殊安排，以使得主线程在其他线程完成前都不退出。让我们来开发一个小库实现这一点。

```
-- file: ch24/NiceFork.hs
import Control.Concurrent
import Control.Exception (Exception, try)
import qualified Data.Map as M

data ThreadStatus = Running
                  | Finished          -- 正常退出
                  | Threw Exception   -- 被未捕获的异常终结
                    deriving (Eq, Show)

-- | 创建一个新线程管理器
newManager :: IO ThreadManager

-- | 创建一个被管理的线程
forkManaged :: ThreadManager -> IO () -> IO ThreadId

-- | 立即返回一个被管理线程的状态
getStatus :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)

-- | 阻塞，直到某个特定的被管理线程终结
waitFor :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)

-- | 阻塞，直到所有被管理线程终结
waitAll :: ThreadManager -> IO ()
```

[Forec 译注：需要对代码做一些改动。在新版本 `Control.Exception` 中，`Exception` 的 kind 是 `* -> *`，需要提供一个具体型别作为参数。可以将代码中的两处 `Exception` 替换为 `SomeException`。]

我们使用一个常见的方法来实现 `ThreadManager` 的类型抽象：将其包裹进一个 `newtype`，并防止使用者直接创建这个类型的值。在模块的导出声明中，我们列出了一个创建线程管理器的 IO 动作，但是并不直接导出类型构造器。

```
-- file: ch24/NiceFork.hs
module NiceFork
    (
      ThreadManager
    , newManager
    , forkManaged
    , getStatus
    , waitFor
    , waitAll
    ) where
```

ThreadManager 的实现中维护了一个线程 ID 到线程状态的 map 。我们将此作为线程 map 。

```
-- file: ch24/NiceFork.hs
newtype ThreadManager =
    Mgr (MVar (M.Map ThreadId (MVar ThreadStatus)))
    deriving (Eq)

newManager = Mgr `fmap` newMVar M.empty
```

此处使用了两层 MVar 。首先将 Map 保存在 MVar 中。这将允许通过使用新版本替换来“改变”map 中的值。同样确保了每个使用这个 Map 的线程可以看到一致的内容。

对每个被管理的线程，都维护一个对应的 MVar 。这种 MVar 从空状态开始，表示这个线程正在执行。当线程被杀死或者发生未处理异常导致退出时，我们将此类信息写入这个 MVar 。

为了创建一个线程并观察它的状态，必须做一点簿记。

```
-- file: ch24/NiceFork.hs
forkManaged (Mgr mgr) body =
    modifyMVar mgr $ \m -> do
      state <- newEmptyMVar
      tid <- forkIO $ do
        result <- try body
        putMVar state (either Threw (const Finished) result)
      return (M.insert tid state m, tid)
```

[Forec 译注：上面这段代码中有一些对读者而言可能相对生疏的函数，在此稍作解释：try 的型别声明是 Exception e => IO a -> IO (Either e a) ，它执行一个 IO 操作，若执行过程中发生异常则返回 Left e，否则返回 Right。either 的型别声明是 (a -> c) -> (b -> c) -> Either a b -> c，如果 try 返回的是 Left 类型，either 会用 Threw 将异常值包裹，否则无论 Right 中包含的值是什么，都返回 Finished 的状态。关于 modifyMVar ，请看下一节的介绍。它的返回值是一个 tuple，这个 tuple 的第一个元素将被放回到 mgr 中，而第二个元素会作为返回值。]

23.4.1 安全的修改 MVar

forkManaged 中使用的 modifyMVar 函数很实用：它将 takeMVar 和 putMVar 安全的组合在一起。

```
ghci> :t modifyMVar
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

其从一个 MVar 中取出一个值，并传入一个函数。这个函数生成一个新的值，且返回一个结果。如果函数抛出一个异常，modifyMVar 会将初始值重新放回 MVar ，否则其会写入新值。它还会返回另一个返回值。

使用 modifyMVar 而非手动使用 takeMVar 和 putMVar 管理 MVar ，可以避免两类并发场景下的问题。

- 忘记将一个值放回 MVar 。有的线程会一直等待 MVar 中被放回一个值，如果一致没有等到，就将导致死锁。
- 没有考虑可能出现的异常，扰乱了某端代码的控制流。这可能导致一个本应执行的 putMVar 没有执行，进而导致死锁。

因为这些美妙的安全特性，尽可能的使用 modifyMVar 是明智的选择。

23.4.2 安全资源管理：一个相对简单的好主意。

modifyMVar 遵循的模式适用很多场景。下面是这些模式：

1. 获得一份资源。
2. 将资源传入一个将处理它函数。
3. 始终释放资源，即使函数抛出异常。如果发生异常，重新抛出异常，以便使其被程序捕获。

除了安全性，这个方法还有其他好处：可以是代码更简短且容易理解。正如前面的 forkManaged ，Hakell 的简洁语法和匿名函数使得这种风格的代码看起来一点都不刺眼。

下面是 modifyMVar 的定义，从中可以了解这个模式的细节：

```
-- file: ch24/ModifyMVar.hs
import Control.Concurrent (MVar, putMVar, takeMVar)
import Control.Exception (block, catch, throw, unblock)
import Prelude hiding (catch) -- use Control.Exception's version

modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
modifyMVar m io =
  block $ do
    a <- takeMVar m
    (b,r) <- unblock (io a) `catch` \e ->
             putMVar m a >> throw e
```

(continues on next page)

(continued from previous page)

```
    putMVar m b
    return r
```

这种模式很容易用于你的特定需求，无论是处理网络连接，数据库句柄，或者被 C 库函数管理的数据。

[Forec 译注：block 和 unblock 在很久以前就已经被弃置了。最新 base 包中 modifyMVar 的实现如下：

```
modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
modifyMVar m io =
  mask $ \restore -> do
  a      <- takeMVar m
  (a',b) <- restore (io a >>= evaluate) `onException` putMVar m a
  putMVar m a'
  return b
```

]

23.4.3 查看线程状态

我们编写的 getStatus 函数用于获取某个线程的当前状态。若某线程已经不被管理（或者未被管理），它返回 Nothing 。

```
-- file: ch24/NiceFork.hs
getStatus (Mgr mgr) tid =
  modifyMVar mgr $ \m ->
    case M.lookup tid m of
      Nothing -> return (m, Nothing)
      Just st -> tryTakeMVar st >>= \mst -> case mst of
                   Nothing -> return (m, Just Running)
                   Just sth -> return (M.delete tid m, Just sth)
```

若线程仍在运行，它返回 Just Running 。否则，它指出将线程为何被终止，并停止管理这个线程。

若 tryTakeMVar 函数发现 MVar 为空，它将立即返回 Nothing 而非阻塞等待。

```
ghci> :t tryTakeMVar
tryTakeMVar :: MVar a -> IO (Maybe a)
```

否则，它将从 MVar 取到一个值。

waitFor 函数的行为较简单，其会阻塞等待给定线程终止，而非立即返回。

```
-- file: ch24/NiceFork.hs
waitFor (Mgr mgr) tid = do
```

(continues on next page)

(continued from previous page)

```
  maybeDone <- modifyMVar mgr $ \m ->
    return $ case M.updateLookupWithKey (\_ _ -> Nothing) tid m of
      (Nothing, _) -> (m, Nothing)
      (done, m') -> (m', done)
  case maybeDone of
    Nothing -> return Nothing
    Just st -> Just `fmap` takeMVar st
```

首先读取保存线程状态的 MVar，若其存在。Map 类型的 updateLookupWithKey 函数很有用：它将查找某个值与更新或移除组合起来。

```
ghci> :m +Data.Map
ghci> :t updateLookupWithKey
updateLookupWithKey :: (Ord k) =>
                       (k -> a -> Maybe a) -> k -> Map k a -> (Maybe a, Map k a)
```

在此处，我们希望若保存线程状态的 MVar 存在，则将其从 Map 中移除，这样线线程管理器将不在管理这个线程。若从其中取到了值，则从中取出线程的退出状态，并将其返回。

我们的最后一个实用函数简单的等待所有当前被管理的线程完成，且忽略他们的退出状态。

```
-- file: ch24/NiceFork.hs
waitAll (Mgr mgr) = modifyMVar mgr elems >>= mapM_ takeMVar
    where elems m = return (M.empty, M.elems m)
```

[Forec 译注：注意 waitAll 函数其实是有缺陷的，它仅仅能够等待在执行 waitAll 之前创建的所有线程。如果在等待期间存在某个线程异步启动，waitAll 是无法获知其状态的。

至此，这个简单的 ThreadManager 基本可以运行了，你可以在 GHCI 中通过如下方式检测一下：

```
Prelude> :l NiceFork.hs
[1 of 1] Compiling NiceFork         ( NiceFork.hs, interpreted )
Ok, modules loaded: NiceFork.
*NiceFork> let calc = do { calc; return ()}
*NiceFork> manager <- newManager
*NiceFork> tid <- forkManaged manager calc
*NiceFork> ans <- getStatus manager tid
*NiceFork> :m +Data.Maybe
*NiceFork Data.Maybe> fromJust ans
Threw stack overflow
```

我们通过一个反复调用自身的 calc 函数构造栈溢出，线程管理器成功地返回了这一结果。

]

23.4.4 编写更紧凑的代码

我们在上面定义的 waitFor 函数有点不完善，因为或多或少执行了重复的模式分析：在 modifyMVar 内部的回调函数，以及处理其返回值时。

当然，我们可以用一个函数消除这种重复。这是 Control.Monad 模块中的 join 函数。

```
ghci> :m +Control.Monad
ghci> :t join
join :: (Monad m) => m (m a) -> m a
```

这是个有趣的主意：可以创建一个 monadic 函数或纯代码中的 action ，然后一直带着它直到最终某处有个 monad 可以使用它。一旦我们了解这种写法适用的场景，就可以更灵活的编写代码。

```
-- file: ch24/NiceFork.hs
waitFor2 (Mgr mgr) tid =
  join . modifyMVar mgr $ \m ->
    return $ case M.updateLookupWithKey (\_ _ -> Nothing) tid m of
      (Nothing, _) -> (m, return Nothing)
      (Just st, m') -> (m', Just `fmap` takeMVar st)
```

23.5 使用频道通信

对于线程间的一次性通信，MVar 已经足够好了。另一个类型，Chan 提供了单向通信频道。此处有一个使用它的简单例子。

```
-- file: ch24/Chan.hs
import Control.Concurrent
import Control.Concurrent.Chan

chanExample = do
  ch <- newChan
  forkIO $ do
    writeChan ch "hello world"
    writeChan ch "now i quit"
  readChan ch >>= print
  readChan ch >>= print
```

若一个 Chan 未空，readChan 将一直阻塞，直到读到一个值。writeChan 函数从不阻塞：它会立即将一个值写入 Chan 。

23.6 注意事项

23.6.1 MVar 和 Chan 是非严格的

正如大多数 Haskell 容器类型，MVar 和 Char 都是非严格的：从不对其内容求值。我们提到它，并非因为这是一个问题，而是因为这通常是一个盲点：人们倾向于假设这些类型是严格的，这大概是因为它们被用在 IO monad 中。

正如其他容器类型，误认为 MVar 和 Chan 是严格的会导致空间和性能的泄漏。考虑一下这个很可能发生的情况：

我们分离一个线程以在另一个核上执行一些开销较大的计算

```
-- file: ch24/Expensive.hs
import Control.Concurrent

notQuiteRight = do
  mv <- newEmptyMVar
  forkIO $ expensiveComputation_stricter mv
  someOtherActivity
  result <- takeMVar mv
  print result
```

它看上去做了一些事情并将结果存入 MVar 。

```
-- file: ch24/Expensive.hs
expensiveComputation mv = do
  let a = "this is "
      b = "not really "
      c = "all that expensive"
  putMVar mv (a ++ b ++ c)
```

当我们在父线程中从 MVar 获取结果并尝试用它做些事情时，我们的线程开始疯狂的计算，因为我们从未强制指定在其他线程中的计算真正发生。

照旧，一旦我们知道了有个潜在问题，解决方案就很简单：未分离的线程添加严格性，以确保计算确实发生。这个严格性最好加在一个位置，以避免我们忘记添加过它。

```
-- file: ch24/ModifyMVarStrict.hs
{-# LANGUAGE BangPatterns #-}

import Control.Concurrent (MVar, putMVar, takeMVar)
import Control.Exception (block, catch, throw, unblock)
import Prelude hiding (catch) -- 使用 Control.Exception's 中的 catch 而非 Prelude 中的。
```

(continues on next page)

(continued from previous page)

```
modifyMVar_strict :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_strict m io = block $ do
  a <- takeMVar m
  !b <- unblock (io a) `catch` \e ->
        putMVar m a >> throw e
  putMVar m b
```

[Forec 译注：block 和 unblock 不被建议使用，更好的方式是使用 mask 。此外，上面的模式匹配仅仅将表达式求值为 WHNF（弱首范式），关于弱首范式的内容将在本章后半部分讨论。你暂时可以简单地将其理解为“剥去表达式的一层”，例如 1 + 2 将被求值为 3，而 "ab" ++ "bc" 则仅仅被求值为 ('a': ("b" ++ "bc"))。所以这里代码的“严格”是有缺陷的。一个可行但不太符合工程应用的做法是使用 Control.DeepSeq 中的 rnf 方法，该方法将表达式求值为范式。建议将原著代码修改如下：

```
-- file: ch24/ModifyMVarStrict.hs
{-# LANGUAGE BangPatterns #-}

import Control.Concurrent (MVar, putMVar, takeMVar)
import Control.Exception (catch, throw, mask)
import Prelude hiding (catch)

modifyMVar_strict :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_strict m io = mask $ \restore -> do
  a <- takeMVar m
  b <- restore (io a) `catch` \e ->
        putMVar m a >> throw e
rnf b `seq` putMVar m b
```

]

Note: 查看 Hackage 始终是值得的。

在 Hackage 包数据库，你将发现一个库，strict-concurrency ，它提供了严格版本的 MVar 和 Chan 类型

上面代码中的！模式用起来很简单，但是并不总是足以确保我们的数据已经被求值。更完整的方法，请查看下面的段落“从求值中分离算法”。

23.6.2 Chan 是无边界的

因为 writeChan 总是立即成功，所以在使用 Chan 时有潜在风险。若对某个 Chan 的写入多于其读取，Chan 将用不检查的方法增长：对未读消息的读取将远远落后于其增长。

23.7 共享状态的并发仍不容易

尽管 Haskell 拥有与其他语言不同的基础设施用于线程间共享数据，它仍需克服相同的基本问题：编写正确的并发程序极端困难。真的，一些其他语言中的并发编程陷阱也会在 Haskell 中出现。其中为人熟知的两个是死锁和饥饿。

23.7.1 死锁

死锁的情况下，两个或多个线程永远卡在争抢共享资源的访问权上。制造多线程程序死锁的一个经典方法是不按顺序加锁。这种类型的 bug 很常见，它有个名字：锁顺序倒置。Haskell 没有提供锁，但 MVar 类型可能会有顺序倒置问题。这有一个简单例子：

```
-- file: ch24/LockHierarchy.hs
import Control.Concurrent

nestedModification outer inner = do
    modifyMVar_ outer $ \x -> do
        yield -- 强制当前线程让出 CPU
        modifyMVar_ inner $ \y -> return (y + 1)
        return (x + 1)
    putStrLn "done"

main = do
    a <- newMVar 1
    b <- newMVar 2
    forkIO $ nestedModification a b
    forkIO $ nestedModification b a
```

在 ghci 中运行这段程序，它通常会（但不总是）不打印任何信息，表明两个线程已经卡住了。

容易看出 nestedModification 函数的问题。在第一个线程中，我们先取出 MVar a，接着取出 b。在第二个线程中，先取出 b 然后取出 a，若第一个线程成功取出了 a 然后要取出 b，这是两个线程都会阻塞：每个线程都尝试获取一个 MVar，而这个 MVar 已经被另一个线程取空了，所以二者都不能完成整个流程。

无论何种语言，通常解决倒序问题的方法是申请资源时一直遵循一致的顺序。因为这需要人工遵循编码规范，在实践中很容易遗忘。

更麻烦的是，这种倒序问题在实际代码中很难被发现。获取 MVar 的动作经常跨越不同文件中的不同函数，这使得通过观察源码检查时更加棘手。更糟糕的是，这类问题通常是间歇性的，这使得它们难于重现，更不要说隔离和修复了。

23.7.2 饥饿

并发软件通常可能会导致饥饿问题，某个线程霸占了共享资源，阻止其他线程使用。很容易想象这是如何发生的：一个线程调用 `modifyMVar` 执行一个 100 毫秒的代码段，稍后另外一个线程对同一个 `MVar` 调用 `modifyMVar` 执行一个 1 毫秒的代码段。第二个线程在第一个线程完成前将无法执行。

`MVar` 类型的非严格性质使会导致或恶化饥饿的问题。若我们将一个求值开销很大的 `thunk` 写入一个 `MVar`，在一个看上去开销较小的线程中取出并求值，这个线程的执行开销马上会变大。所以我们在 “MVar 和 Chan 是非严格的” 一章中特地给出了一些建议。

23.7.3 没希望了吗？

幸运的是，我们已经提及的并发 `API` 并不是故事的全部。最近加入 Haskell 中的一个设施，软件事务内存，使用起来更加容易和安全。我们将在第 28 章，软件事务内存中介绍。

23.8 练习

1. `Chan` 类型是使用 `MVar` 实现的。使用 `MVar` 来开发一个有边界的 `Chan` 库。
2. 你开发的 newBoundedChanfunction 接受一个 `Int` 参数，限制单独 `BoundedChan` 中的未读消息数量。
3. 达到限制是，调用 `writeBoundedChanfunction` 要被阻塞，知道某个读取者使用 `readBoundedChan` 函数消费掉队列中的一个值。
4. 尽管我们已经提到过 Hackage 库中的 `strict-concurrency` 包，试着自己开发一个，作为内置 `MVar` 类型的包装。按照经典的 `Haskell` 实践，使你的库类型安全，让用户不会混淆严格和非严格的 `MVar` 。

23.9 在 GHC 中使用多核

默认情况下，`GHC` 生成的程序只使用一个核，甚至在编写并发代码时也是如此。要使用多核，我们必须明确指定。当生成可执行程序时，要在链接阶段指定这一点。

- “non-threaded” 运行时库在一个操作系统线程中运行所有 `Haskell` 线程。这个运行时在创建线程和通过 MVar 传递数据时很高效。
- “threaded” 库使用多个操作系统线程运行 `Haskell` 线程。它在创建线程和使用 `MVar` 时具有更高的开销。

若我们在向编译器传递 `-threadedoption` 参数，它将使用 `threaded` 运行时库链接我们的程序。在编译库和源码文件时无需指定 `-threaded` ，只是在最终生成可执行文件时需要指定。

即使为程序指定了 `threaded` 运行时，默认情况下它仍将只使用一个核运行。必须明确告诉运行时使用多少个核。

23.9.1 运行时选项

运行程序时可以向 GHC 的运行时系统传递命令行参数。在将控制权交给我们的代码前，运行时扫描程序的参数，看是否有命令行选项 `+RTS` 。其后跟随的所有选项都被运行时解释，直到特殊的选项 `-RTS` ，这些选项都是提供给运行时系统的，不为我们的程序。运行时会对我们的代码隐藏所有这些选项。当我们使用 `System.Environment` 模块的 `getArgsfunction` 来获得我们的命令行参数是，我们不会在其中获得运行时选项。

`threaded` 运行时接受参数 `-N`[55] 。其接受一个参数，指定了 GHC 的运行时系统将使用的核数。这个选项对输入很挑剔：`-N` 和参数之间必须没有空格。`-N4` 可被接受，`-N 4` 则不被接受。

23.9.2 找出 Haskell 可以使用多少核

`GHC.Conc` 模块输出一个变量，`numCapabilities` ，它会告诉我们运行时系统被 `-NRTS` 选项指定了多少核。

```
-- file: ch24/NumCapabilities.hs
import GHC.Conc (numCapabilities)
import System.Environment (getArgs)

main = do
    args <- getArgs
    putStrLn $ "command line arguments: " ++ show args
    putStrLn $ "number of cores: " ++ show numCapabilitie
```

若编译上面的程序，我们可以看到运行时系统的选项对于程序来说是不可见的，但是它可以看其运行在多少核上。

```
$ ghc -c NumCapabilities.hs
$ ghc -threaded -o NumCapabilities NumCapabilities.o $ ./NumCapabilities +RTS -N4 -
↪RTS foo
command line arguments: ["foo"]
number of cores: 4
```

23.9.3 选择正确的运行时

选择正确的运行时需要花点心思。`threaded` 运行时可以使用多核，但是也有相应的代价：线程间共享数据的成本比 `non-threaded` 运行时更大。

更关键的是，GHC 6.8.3 版本使用的还是单线程的垃圾收集器：它在执行时会暂停其他所有线程，然后在单个核上执行垃圾收集工作。这限制了我们在使用多核的时候希望看到的性能改进[56] 。

[55] `non-threaded` 运行时不接受这个选项，会用一条错误信息拒绝它。

[56] 此书撰写时，垃圾收集器已经开始重新编写以利用多核，但是我们不确定它在未来的效果。

很多真实世界中的并发程序中，一个单独的线程多数时间实在等待一个网络请求或响应。这些情况下，若以一个单独的 Haskell 程序为数万并发客户端提供服务，使用低开销的 non-threaded 运行时很可能是合适的。例如，与其用 4 个核跑 threaded 运行时的单个服务器程序，可能同时跑 4 个 non-threaded 运行时的相同服务器程序性能更好。

我们的目的并不是阻止你使用 threaded 运行时。相对于 non-threaded 运行时它并没有特别大的开销：相对于其他编程语言，线程依旧惊人的轻量。我们仅是希望说明 threaded 运行时并不是在所有场景都是最佳选择。

23.10 Haskell 中的并行编程

现在让我们来关注一下并行编程。对很多计算密集型问题，可以通过分解问题，并在多个核上求值来更快的计算出结果。多核计算机已经普及，甚至在最新的笔记本上都有，但是很少有程序可以利用这一优势。

大部分原因是因为传统观念认为并行编程非常困难。在一门典型的编程语言中，我们将用处理并发程序相同的库和设施处理并发程序。这是我们的注意力集中在处理一些熟悉的问题比如死锁、竞争条件、饥饿和陡峭的复杂性。

但是我们可以确定，使用 Haskell 的并发特性开发并行代码时，有许多更简单的方法。在一个普通的 Haskell 函数上稍加变化，就可以并行求值。

23.10.1 范式和首范式

seq 函数将一个表达式求值为首范式（简称 HNF）。seq 一旦到达最外层的构造函数（也就是“首部”）就会停止，这与范式不同（NF），被称作范式的表达式必然是被完全求值的，而非仅仅“剥离”掉最外层的构造函数。

你可能会经常听到 Haskell 程序员提到弱首范式（WHNF）。对一般数据来说，弱首范式和首范式相同。它们仅仅在功能上有些许区别，这里我们就不过多关注了。

[Forec 译注：读者只需要记住范式和弱首范式这两个概念的区别，HNF 几乎可以忽略。以下两个链接可以帮助更好的理解，这部分内容与并行编程关系不是非常紧密，因此不在此处过多叙述：* Haskell Wiki: https://en.wikibooks.org/wiki/Haskell/Laziness#Thunks_and_Weak_head_normal_form * StackOverflow: http://stackoverflow.com/questions/6872898/haskell-what-is-weak-head-normal-form]

23.10.2 排序

这是一个使用分治算法实现的 Haskell 排序函数：

```
-- file: ch24/Sorting.hs
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
```

(continues on next page)

(continued from previous page)

```
  where lesser  = sort [y | y <- xs, y <  x]
        greater = sort [y | y <- xs, y >= x]
sort _ = []
```

sort 函数实现了著名的快速排序算法，很多 Haskell 程序员将其视作经典：在早期的 Haskell 教程中，用一行代码实现的 sort 经常作为示例向读者展示 Haskell 强大的表达能力。这里我们将代码切分为几行以方便比较串行和并行版本。

下面是对 sort 工作流程的简要介绍：

1. 从列表中取出一个元素，这个元素被称为 “轴心”（或哨兵）。每个元素都要和轴心比较。上面的代码简单地通过模式匹配选取列表的第一个元素作为轴心；
2. 使用原始列表中除轴心外的其它元素构造一个子列表，子列表中元素的值全部小于轴心，并递归地处理子列表；
3. 与 2 类似，构造另一个子列表，但子列表中元素的值均大于或等于轴心的值，递归处理这个子列表；
4. 将 2、3 两步中排序后的子列表通过轴心进行连接。

23.10.3 将代码变换为并行版本

并行版本的排序函数相对要复杂一些：

```
-- file: ch24/Sorting.hs
module Sorting where

import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x:xs)    = force greater `par` (force lesser `pseq` (lesser ++ x:greater))
  where lesser  = parSort [y | y <- xs, y <  x]
        greater = parSort [y | y <- xs, y >= x]
parSort _         = []
```

不过我们并没有改变代码的结构。parSort 仅仅多使用了三个函数：par、pseq 和 force。

par 函数由 Control.Parallel 模块提供。它与 seq 目的类似：将左侧参数求值为弱首范式并返回右侧参数。par 的名字很好地阐述了它的功能：par 能够在其它运算执行的同时并行地对其左侧参数求值。

pseq 也和 seq 类似：它在返回右侧表达式之前将左侧表达式求值为弱首范式。这二者之间的区别很微妙，但对于并行编程而言非常重要：编译器不保证 seq 的求值顺序，即如果编译器认为首先对右侧参数求值能够提高性能，则它将先计算右侧参数。对于单核执行的程序来说，这种灵活性很有必要，但对多核代码而言不够健壮。相比之下，编译器能够保证 pseq 左侧参数的求值过程早于右侧参数。

以上修改将对一些我们没有提到的方面产生巨大的影响，比如说：

- 使用多少个核心
- 线程间如何通信
- 如何将工作分配给多个可用核心
- 明确哪些数据将在线程间共享，哪些属于线程私有
- 如何确定所有任务均已完成

23.10.4 明确在并行中执行什么

并行的 Haskell 代码之所以能够表现出更优秀的性能，是因为计算过程中有大量重复、独立、可并行计算的工作。非严格求值会阻碍并行程序的执行，因此我们会在并行代码中使用 `force` 函数。下面通过一个错误的例子解释 `force` 函数的功能：

```
-- file: ch24/Sorting.hs
sillySort (x:xs) = greater `par` (lesser `pseq` (lesser ++ x:greater))
  where lesser  = sillySort [y | y <- xs, y <  x]
        greater = sillySort [y | y <- xs, y >= x]
sillySort _     = []
```

注意，我们在 `par` 和 `pseq` 两处用普通求值取代了 `force lesser` 和 `force greater`。

回忆一下，对弱首范式的求值会在“看到”表达式的外部构造器时停止。在错误的例子中，我们将每个有序的子列表求值为 WHNF。因为最外层的构造器仅仅是一个列表构造器，上面的代码实际上仅仅强制对每个排序子列表的第一个元素做了求值，每个排序子列表剩余的元素仍未被完全求值。换句话说，上面代码在并行部分执行时几乎没有做任何有效计算，`sillySort` 的执行过程和完全顺序的代码没什么差别。

[Forec 译注：考虑此前译注给出的例子，`"ab" ++ "bc"` 的弱首范式仅仅是 `'a' : ("b" ++ "bc")`。以待排序列表 `[3, 2, 5, 1, 3]` 为例，`sillySort` 试图在 `par` 包装的并行操作中计算出排序好的两个子列表。假设轴心元素为 3，则小于轴心的待排序子列表应为 `[2, 1]`，而 `par` 中的并行操作在计算出第一个满足条件的元素 2 后，得到 `lesser = (2: _)`，已经遇到了最外层的列表构造器，因此停止计算。真正求出完整待排序子列表是在后续顺序操作 `++` 时，因为需要列表中所有元素，这时程序才开始计算。

这里给出一个译者认为更容易理解弱首范式的例子：在 GHCI 中执行如下指令。

```
> let list = [1..10]
> let whnfList = map (+1) list
> :sprint whnfList
whnfList = _
> length whnfList
10
```

(continues on next page)

(continued from previous page)

```
> :sprint whnfList
whnfList = [_, _, _, _, _, _, _, _, _, _]
```

可以看出，length 操作仅需要 whnfList 中元素的数量，并没有对其中元素进行更深层次的求值。

]

我们使用 force 函数，在构造函数返回前遍历整个列表以避免这种情况出现。

```
-- file: ch24/Sorting.hs
force :: [a] -> ()
force xs = go xs `pseq` ()
  where go (_:xs) = go xs
        go [] = 1
```

注意，我们并不在乎列表中具体有什么，而是仅仅把列表遍历一遍，遍历之后再调用 pseq 。因为我们会在 par 或者 pseq 的左侧使用 force，所以返回值无所谓。

当然，很多情况下我们会需要对列表中的个别元素强制求值。下面会介绍一个基于类型类的解决方案。

23.10.5 par 提供什么保证？

实际上，par 函数并不保证会将表达式并行求值，它只在对表达式并行求值有意义的时候才这么做。在并行编程中，这种行为比保证并行执行更有效。它允许运行时系统遇到 par 时智能调度。

举个例子，运行时系统可能发现表达式过于简单，并行求值带来的性能提升远低于并行操作本身的额外开销。或者，运行时系统发现所有的计算核心均正在工作，而启动一个新的并行运算仅仅会增加待运行线程的数量。

这个潜规则影响了我们如何编写并行代码。假设系统性能不会因为线程间争夺核心资源下降，考虑到 par 在运行时可以智能调度，我们就可以将它应用到任何想应用的地方。

23.10.6 运行并测试性能

将 sort、parSort 和 parSort2 保存到 Sorting.hs 中并封装为 Sorting 模块。我们创建一个驱动程序以计算这些排序函数的性能：

```
-- file: ch24/SortMain.hs

module Main where

import Data.Time.Clock (diffUTCTime, getCurrentTime)
import System.Environment (getArgs)
import System.Random (StdGen, getStdGen, randoms)
```

(continues on next page)

(continued from previous page)

```
import Sorting

-- testFunction = sort
-- testFunction = seqSort
testFunction = parSort
-- testFunction = parSort2 2

randomInts :: Int -> StdGen -> [Int]
randomInts k g = let result = take k (randoms g)
         in force result `seq` result

main = do
  args <- getArgs
  let count | null args = 500000
            | otherwise = read (head args)
  input <- randomInts count `fmap` getStdGen
  putStrLn $ "We have " ++ show (length input) ++ " elements to sort."
  start <- getCurrentTime
  let sorted = testFunction input
  putStrLn $ "Sorted all " ++ show (length sorted) ++ " elements."
  end <- getCurrentTime
  putStrLn $ show (end `diffUTCTime` start) ++ " elapsed."
```

简单起见，我们使用 `testFunction` 变量选择用于基准测试的排序函数。

上面的程序接受一个可选的命令行参数，用于指定待排序随机数组的长度。

非严格求值是性能测量和分析中要注意的“雷区”。下面是驱动程序中特别要避免的一些潜在问题：

- 非严格求值会使测量单一行为变为测量多个行为。 Haskell 默认的随机数产生器（PRNG）很慢，而且 `random` 函数仅仅在需要的时候才产生下一个随机数。我们在记录开始时间之前对输入列表的每个元素进行了强制求值，并且打印了列表的长度：这保证了程序在计算之前就已经生成了全部的随机数。如果我们忽略了这一步，则并行计算的过程会包含随机数的生成，进而导致测量出的时间变为生成随机数和数据排序所用时间之和，而非数据排序本身的耗时。

- 隐含的数据依赖。在产生随机数列表时，只打印列表的长度并不会对列表完全求值。`length` 函数只会遍历列表的结构而非列表内的每个元素。因此，在排序操作执行前，列表中并没有生成好的随机数。这一行为可能严重拖慢性能。每个随机数的产生都取决于列表中前一个随机数的值，但并行的数据排序已经将列表元素分散到了不同的处理器内核中。如果排序前没有对输入的随机数列表完全求值，那么运行时就会遭遇可怕的“乒乓”效应：计算会在核心之间不停跳跃，导致性能的迅速下降。尝试删除 `main` 函数中应用在随机数列表上的 `force` ：你会发现并行代码比顺序执行的程序慢了三倍。[Forec 译注：原著作者似乎忘记在代码中使用 `force`。可以在运算开始前加入 `force input`。]

- 让我们错误地认为代码执行了有意义的工作。为了保证数据排序的执行，我们在记录结束时间之前将

结果列表的长度打印到了屏幕上。如果没有 putStrLn 强制要求列表长度，排序压根就不会执行。

在构建程序时开启优化和 GHC 的运行时线程支持：

```
$ ghc -threaded -O2 --make SortMain
[1 of 2] Compiling Sorting          ( Sorting.hs, Sorting.o )
[2 of 2] Compiling Main             ( SortMain.hs, SortMain.o )
Linking SortMain ...
```

在程序运行时告知 GHC 的运行时系统使用多少核心。首先测试最原始的 sort 函数，看看基础性能如何：

```
$ ./Sorting +RTS -N1 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
3.178941s elapsed.
```

启用两个核心并不会提升顺序执行代码的性能：

```
$ ./Sorting +RTS -N2 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
3.259869s elapsed.
```

如果重新编译，测试 parSort 的性能，就会发现结果还不如顺序代码：

```
$ ./Sorting +RTS -N1 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
3.915818s elapsed.
$ ./Sorting +RTS -N2 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
4.029781s elapsed.
```

性能上没有任何提升。这看起来是以下两个原因之一造成的：要么是 par 本身的开销过大，要么是我们滥用了 par 。为了鉴别究竟哪个才是罪魁祸首，我们编写了一个类似 parSort 的函数 seqSort，它使用 pseq 代替 par：

```
seqSort :: (Ord a) => [a] -> [a]
seqSort (x:xs) = lesser `pseq` (greater `pseq` (lesser ++ x:greater))
  where lesser  = seqSort [y | y <- xs, y <  x]
        greater = seqSort [y | y <- xs, y >= x]
seqSort _ = []
```

我们还删去了 parSort 中对 force 的调用。所以将 seqSort 和 sort 比较就可以观察到只应用 pseq 的性能：

```
$ ./Sorting +RTS -N1 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
3.848295s elapsed.
```

上面的运行结果说明 par 和 pseq 耗时相似。我们该如何改进性能呢?

23.10.7 性能调优

在 parSort 中，调用 par 的次数是待排序数据数量的两倍。尽管 par 的开销很小，但它毕竟也不是“免费”的。当递归调用 parSort 时，我们最终将 par 应用到了单个元素上。在这种细微的粒度下，使用 par 的成本远超其带来的增益。为了减少这种影响，我们在待排序数据数量小于某个阈值时采用非并行排序。

```
-- file: ch24/Sorting.hs
parSort2 :: (Ord a) => Int -> [a] -> [a]
parSort2 d list@(x:xs)
  | d <= 0     = sort list
  | otherwise  = force greater `par` (force lesser `pseq` (lesser ++ x:greater))
    where lesser      = parSort2 d' [y | y <- xs, y <  x]
          greater     = parSort2 d' [y | y <- xs, y >= x]
          d' = d - 1
parSort2 _ _              = []
```

parSort2 在到达可控深度时停止递归以及创建新的并行计算。如果待处理数据的规模已知，程序就能在剩余工作数量足够少的时刻停止并行计算并切换到非并行代码。

```
$ ./Sorting +RTS -N2 -RTS 700000
We have 700000 elements to sort.
Sorted all 700000 elements.
2.947872s elapsed.
```

在双核系统中，这一改进将运行速度提升了大约 25%。性能的提升并不高，但想想我们对代码所做的改变：几句注解而已就让速度提升了四分之一。

上面的排序函数难以获得良好的并行性能。它执行了大量内存分配，导致垃圾回收器频繁运行。-sstderr 这一 RTS 选项能够将垃圾回收统计信息打印到屏幕，反馈信息显示程序大约有 40% 的时间用于回收垃圾。由于 GHC 6.8 中的垃圾回收器会停止所有线程并运行在单个内核上，这一点也就成为了程序的瓶颈。

将 par 应用在内存分配不频繁的代码中可能会带来明显的性能改进。我们已经看到，相比单核运算，上面的基准测试在双核系统上能够取得 1.8 倍加速。在这本书编写时，GHC 正在开发一个并行的垃圾回收器，届时在多核系统上执行大量内存分配的代码也能获得令人满意的性能。

[Forec 译注：翻译此处时 GHC 的最新版本为 8.0.2，已支持多核系统的垃圾回收。实际测试表明，即使支持多核 GC，并行编程的主要瓶颈在很多时候还是垃圾回收。]

23.10.8 练习

1. 决定什么时候从 parSort2 切换到 sort 不是一件容易的事。我们上面实现的方法是根据递归的深度选择，另一种方法是根据待排序子列表的长度决定。重写 parSort2 使其在待排序子列表长度小于某个数的时候切换到 sort。

2. 测试基于列表长度切换的 parSort2 的性能，并于基于递归深度的版本比较。哪一个性能更好?

23.11 并行策略和 Map Reduce

在编程社区中，Google 用于并行处理海量数据的 MapReduce 框架是受函数式编程启发的著名软件系统之一。

我们可以用 Haskell 轻松构建一个简单而实用的“山寨版”MapReduce。我们将把重点放在 Web 服务器日志文件的处理上，这些文件通常规模较大并且包含丰富的信息[57] 。

下面这个例子是 Apache Web 服务器记录的一条访问日志。这条日志原本是一行，这里为了阅读方便将其切分成多行。

```
201.49.94.87 - - [08/Jun/2008:07:04:20 -0500] "GET / HTTP/1.1"
200 2097 "http://en.wikipedia.org/wiki/Mercurial_(software)"
"Mozilla/5.0 (Windows; U; Windows XP 5.1; en-GB; rv:1.8.1.12)
Gecko/20080201 Firefox/2.0.0.12" 0 hgbook.red-bean.com
```

虽然直接处理这些日志文件的实现非常简单，但我们显然不能满足于此。如果换个思路，考虑解决一类问题而非一个问题，我们就可能得到适用范围更广的代码。

在开发并行程序时，无论使用何种底层语言，我们都总会遇到一些“坏钱币”问题:

[Forec 译注：坏钱币，原文为“Bad penny”，意味着一个不好的东西总会一次接一次地出现。]

- 算法很快就会被分区和通信的细节所掩盖，从而导致代码变得难以理解和修改;
- 对“粒度”的选择将变得困难——我们将无法轻易地决定应该分配多少工作给每个核心。如果粒度太小了，那么核心将会在簿记工作上花费大量时间，导致并行程序跑得比串行程序还要慢；但如果粒度太大的话，某些核心又会因为糟糕的负载平衡而被闲置。

23.11.1 将算法和求值分离

并行 Haskell 代码不存在传统语言中通信部分代码产生的混乱，取而代之的，是 par 和 pseq 对应的繁杂注释。举个例子，下面这个函数的操作方式与 map 类似，但会以并行方式将每个元素求值为弱首范式 (WHNF)。[Forec 译注：弱首范式的概念在上文的译注中已经介绍过。]

[57] 这个想法是 Tim Bray 提出的。

```haskell
-- file: ch24/ParMap.hs
import Control.Parallel (par)

parallelMap :: (a -> b) -> [a] -> [b]
parallelMap f (x:xs) = let r = f x
                       in r `par` r : parallelMap f xs
parallelMap _ _      = []
```

类型 b 可以是一个列表，或者是其它容易求值为弱首范式的类型。我们希望得到万精油式的解法，而不必为列表或者其他任何特殊类型编写定制的 parallelMap 。

要解决这个问题，首先需要考虑如何强制求值。下面这个函数将列表中的每个元素强制求值为弱首范式：

```haskell
-- file: ch24/ParMap.hs
forceList :: [a] -> ()
forceList (x:xs) = x `pseq` forceList xs
forceList _      = ()
```

上面的函数并不对列表做任何计算（实际上，该函数的类型签名就已经说明它无法做任何计算，因为它并不知晓列表中的具体元素）。这个函数的唯一目的是确保列表被求值为首范式，它仅能作为 seq 或 par 函数的第一个参数，例如：

```haskell
-- file: ch24/ParMap.hs
stricterMap :: (a -> b) -> [a] -> [b]
stricterMap f xs = forceList xs `seq` map f xs
```

上面的 stricterMap 仅仅使列表中的元素被求值为弱首范式。我们可以添加一个函数作为参数，从而强制每个元素被求值至更深的层次：

```haskell
-- file: ch24/ParMap.hs
forceListAndElts :: (a -> ()) -> [a] -> ()
forceListAndElts forceElt (x:xs) =
    forceElt x `seq` forceListAndElts forceElt xs
forceListAndElts _        _      = ()
```

模块 Control.Parallel.Strategies 将这个想法浓缩为一个库，它引入了“求值策略”（Evaluation Strategy）这个概念：

```haskell
-- file: ch24/Strat.hs
type Done = ()
type Strategy a = a -> Done
```

一个求值策略不会执行任何计算操作，它仅仅保证求值的程度。最简单的策略是 r0，也就是什么都不做：

```haskell
-- file: ch24/Strat.hs
r0 :: Strategy a
r0 _ = ()
```

下面是 rwhnf ，它会将参数求值为弱首范式：

```haskell
-- file: ch24/Strat.hs
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

这个模块还提供了一个类型类 NFData，它提供了方法 rnf ，该方法能够将一个值求值为范式：

```haskell
-- file: ch24/Strat.hs
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

[Forec 译注：NFData 类型类已经被迁移至 Control.DeepSeq ，rnf 现在已经不再是策略，取而代之的是 rdeepseq 。]

对于基本的类型（如 Int ），弱首范式和范式完全相同，这也是 NFData 类型类将 rwhnf 用作 rnf 默认实现的原因。Control.Parallel.Strategies 模块也为许多常见类型提供了 NFData 的实例。

```haskell
-- file: ch24/Strat.hs
instance NFData Char
instance NFData Int

instance NFData a => NFData (Maybe a) where
    rnf Nothing  = ()
    rnf (Just x) = rnf x

{- ... and so on ... -}
```

通过这些例子，你应该已经大致清楚如何为自定义类型编写 NFData 实例了。你的 rnf 应当能够处理每个构造器，并且将 rnf 应用到构造器的每个字段上。

23.11.2 将算法和策略分离

我们可以通过这些基本策略构造更多复杂策略。Control.Parallel.Strategies 提供了很多更复杂的策略。比如说，parList 会将一个求值策略并行地应用到列表的每个元素上：

```haskell
-- file: ch24/Strat.hs
parList :: Strategy a -> Strategy [a]
```

(continues on next page)

(continued from previous page)

```
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

该模块用 parList 定义了并行的 map 函数：

```
-- file: ch24/Strat.hs
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

这正是代码的有趣之处。在 using 的左边是普通形式的 map ，而右侧是求值策略。组合器 using 告诉我们如何对一个值使用策略，从而使代码和具体的求值方式分离：

```
-- file: ch24/Strat.hs
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

Control.Parallel.Strategies 还包含了许多其它函数，它们可以更精确地求值。例如，parZipWith 通过求值策略并行地应用 zipWith ：

```
-- file: ch24/Strat.hs
vectorSum' :: (NFData a, Num a) => [a] -> [a] -> [a]
vectorSum' = parZipWith rnf (+)
```

[Forec 译注：函数 parZipWith 似乎并不存在，也许这里是原作自己实现的某个函数。]

23.11.3 编写简单的 MapReduce 定义

只需要考虑一下 mapReduce 函数需要做什么，就能推导出它的类型。首先，map 部件是必须的，因此有类型 a -> b ；其次需要 reduce ，也就是 fold 的同义词，这里我们使用更普适的类型 [b] -> c 。这个类型同时兼容左/右 fold，因此我们可以根据数据和处理过程灵活地选择。

把这些类型整合到一起，就得到了以下这个完整的类型：

```
-- file: ch24/MapReduce.hs
simpleMapReduce
    :: (a -> b)       -- map function
    -> ([b] -> c)     -- reduce function
    -> [a]            -- list to map over
    -> c
```

代码非常简单：

```
simpleMapReduce mapFunc reduceFunc = reduceFunc . map mapFunc
```

23.11.4 MapReduce 和策略

上面定义的 simpleMapReduce 实在太简单了。我们希望能够指定一些并行执行的工作，让它变得更加实用。这一目标可以通过分别在 map 和 reduce 阶段传入策略实现。

```
-- file: ch24/MapReduce.hs
mapReduce
    :: Strategy b    -- evaluation strategy for mapping
    -> (a -> b)      -- map function
    -> Strategy c    -- evaluation strategy for reduction
    -> ([b] -> c)    -- reduce function
    -> [a]           -- list to map over
    -> c
```

修改后的类型和函数体大小都有所增长。

```
-- file: ch24/MapReduce.hs
mapReduce mapStrat mapFunc reduceStrat reduceFunc input =
    mapResult `pseq` reduceResult
  where mapResult    = parMap mapStrat mapFunc input
        reduceResult = reduceFunc mapResult `using` reduceStrat
```

23.11.5 适当调整工作量

要取得更好的性能，必须保证 par 中每个应用所做的工作量远大于维护并行计算所需额外数据的开销。例如处理一个巨大的文件时，若按行切割，那么真正有意义的工作将远小于切割、合并的额外开销。

我们将在后面的章节开发一种处理大块文件的方式。这些大块会由什么组成？由于 Web 服务器日志文件应该只包含 ASCII 文本，我们将会看到 ByteString 的出色性能：这种类型极其高效，并且以流方式传输时仅消耗非常少的内存。

```
-- file: ch24/LineChunks.hs
module LineChunks
    (
      chunkedReadWith
    ) where

import Control.Exception (bracket, finally)
import Control.Monad (forM, liftM)
import Control.Parallel.Strategies (NFData, rnf)
```

(continues on next page)

(continued from previous page)

```
import Data.Int (Int64)
import qualified Data.ByteString.Lazy.Char8 as LB
import GHC.Conc (numCapabilities)
import System.IO

data ChunkSpec = CS {
      chunkOffset :: !Int64
    , chunkLength :: !Int64
    } deriving (Eq, Show)

withChunks :: (NFData a) =>
              (FilePath -> IO [ChunkSpec])
           -> ([LB.ByteString] -> a)
           -> FilePath
           -> IO a
withChunks chunkFunc process path = do
  (chunks, handles) <- chunkedRead chunkFunc path
  let r = process chunks
  (rnf r `seq` return r) `finally` mapM_ hClose handles

chunkedReadWith :: (NFData a) =>
                   ([LB.ByteString] -> a) -> FilePath -> IO a
chunkedReadWith func path =
    withChunks (lineChunks (numCapabilities * 4)) func path
```

这个程序会以并行的方式处理各个块，并谨慎地利用惰性 I/O 的优势以便确保程序可以安全地传输这些块。

23.11.6 减轻惰性 I/O 的风险

惰性 I/O 带来了一些众所周知的风险，以下是一些我们需要避免的地方：

- 如果不强制要求程序从文件中拉取数据并计算，那么惰性 I/O 的特性会隐式地延长文件句柄打开的时间。操作系统会限制同时打开的文件数量，这个限制的数量较小且固定。如果这个风险不解决，可能会导致程序中使用到文件句柄的其他部分发生饥饿现象。
- 如果程序没有显式地关闭文件句柄，那么文件句柄将由垃圾回收器负责自动关闭，但等到 GC 注意到这个泄漏的文件句柄可能需要很长的时间。这与上面的饥饿风险同理。
- 显式关闭句柄能够避免饥饿，但如果关闭得太早，惰性计算可能还需要从这个被关闭的句柄中读取更多数据，这将导致该计算的失败。

除了这些常见的风险外，我们无法通过单个文件句柄为多个线程提供数据。一个文件句柄只有一个“寻址指针”，它指明了当前应当读取的位置。如果我们想要读取多个块，那么每一块都需要从文件中的不同位置读取，而单个句柄显然是无法满足这种要求的。

把上面这些思考都整合到一起，我们就得到了以下的惰性 I/O 解决方案：

```
-- file: ch24/LineChunks.hs
chunkedRead :: (FilePath -> IO [ChunkSpec])
            -> FilePath
            -> IO ([LB.ByteString], [Handle])
chunkedRead chunkFunc path = do
  chunks <- chunkFunc path
  liftM unzip . forM chunks $ \spec -> do
    h <- openFile path ReadMode
    hSeek h AbsoluteSeek (fromIntegral (chunkOffset spec))
    chunk <- LB.take (chunkLength spec) `liftM` LB.hGetContents h
    return (chunk, h)
```

上面的代码通过显式关闭文件句柄来避免饥饿问题。程序会为读取同一个文件的多个线程分别提供独立的句柄，从而允许这些线程同时读取该文件的不同块。

我们要解决的最后一个问题，就是如何防止句柄在计算结束之前被关闭。程序使用 `rnf` 保证所有的处理过程在 `withChunks` 函数返回前完成，从而可以显式关闭文件句柄（可以确保这些句柄不会再被读取）。如果你必须在程序中使用惰性 I/O，那么最好用这种方式构建一道 “防火墙”，从而避免在意想不到的位置发生错误。

23.11.7 高效寻找行对齐的块

由于服务器的日志文件是面向行的，所以需要寻找一种高效的方式，保证在文件切分为大块后，每块均在行的边界上结束。我们不想通过扫描一个块的所有数据来确定边界，因为每一块数据可能达到几十兆字节。

无论是固定块的大小还是固定块的数量，我们的方法都能奏效，这里选择后者。首先寻求一个大块结尾的大致位置，之后向前扫描直到换行符，然后在换行符后开始寻找下一个块，并重复该过程。

```
-- file: ch24/LineChunks.hs
lineChunks :: Int -> FilePath -> IO [ChunkSpec]
lineChunks numChunks path = do
  bracket (openFile path ReadMode) hClose $ \h -> do
    totalSize <- fromIntegral `liftM` hFileSize h
    let chunkSize = totalSize `div` fromIntegral numChunks
        findChunks offset = do
          let newOffset = offset + chunkSize
          hSeek h AbsoluteSeek (fromIntegral newOffset)
          let findNewline off = do
                eof <- hIsEOF h
                if eof
                  then return [CS offset (totalSize - offset)]
                  else do
                    bytes <- LB.hGet h 4096
```

(continues on next page)

(continued from previous page)

```
                    case LB.elemIndex '\n' bytes of
                      Just n -> do
                        chunks@(c:_) <- findChunks (off + n + 1)
                        let coff = chunkOffset c
                        return (CS offset (coff - offset):chunks)
                      Nothing -> findNewline (off + LB.length bytes)
          findNewline newOffset
    findChunks 0
```

最后一块的大小会比前面的块稍小一些，但这种差异在实际应用中无关紧要。

23.11.8 计算行数

下面这个简单的例子展示了如何使用我们构建的“脚手架”。

```
-- file: ch24/LineCount.hs
module Main where

import Control.Monad (forM_)
import Data.Int (Int64)
import qualified Data.ByteString.Lazy.Char8 as LB
import System.Environment (getArgs)

import LineChunks (chunkedReadWith)
import MapReduce (mapReduce, rnf)

lineCount :: [LB.ByteString] -> Int64
lineCount = mapReduce rnf (LB.count '\n')
                      rnf sum

main :: IO ()
main = do
  args <- getArgs
  forM_ args $ \path -> do
    numLines <- chunkedReadWith lineCount path
    putStrLn $ path ++ ": " ++ show numLines
```

若使用 `ghc -O2 --make-threaded` 命令编译此程序，它在运行过一次后性能会更好（文件系统已缓存数据）。在双核笔记本上，处理 248MB （110 万行）的数据只需要 0.361 秒（加上 `+RTS -N2` 参数），单核需要 0.576 秒。

23.11.9 找到最受欢迎的 URL

下面这个例子将计算每个 URL 被访问的次数，它来自 [Google08]（Google 讨论 MapReduce 的原始论文）。在 `map` 阶段，我们为每个数据块创建从 URL 映射到访问次数的 `Map` 。在 `reduce` 阶段，合并这些映射。

[Forec 译注：该链接需科学上网后访问。]

```
-- file: ch24/CommonURLs.hs
module Main where

import Control.Parallel.Strategies (NFData(..), rwhnf)
import Control.Monad (forM_)
import Data.List (foldl', sortBy)
import qualified Data.ByteString.Lazy.Char8 as L
import qualified Data.ByteString.Char8 as S
import qualified Data.Map as M
import Text.Regex.PCRE.Light (compile, match)

import System.Environment (getArgs)
import LineChunks (chunkedReadWith)
import MapReduce (mapReduce)

countURLs :: [L.ByteString] -> M.Map S.ByteString Int
countURLs = mapReduce rwhnf (foldl' augment M.empty . L.lines)
                      rwhnf (M.unionsWith (+))
  where augment map line =
            case match (compile pattern []) (strict line) [] of
              Just (_:url:_) -> M.insertWith' (+) url 1 map
              _ -> map
        strict  = S.concat . L.toChunks
        pattern = S.pack "\"(?:GET|POST|HEAD) ([^ ]+) HTTP/"
```

为了从日志文件的每一行中提取出 URL，我们使用了在 chapter_17 中开发的 PCRE 正则表达式库。

驱动程序打印了十条最受欢迎的 URL。与统计行数的例子类似，这个程序在双核上的速度是单核的 1.8 倍，处理 110 万条日志只需要 1.7 秒。

23.11.10 总结

给定一个适合 MapReduce 模型的问题，MapReduce 编程模型可以帮助我们使用 Haskell 以极小的工作量编写“临时”但高效的并行程序。此外，这个想法可以轻松扩展到其它数据源，比如文件集以及网络上的数据资源。

在许多情况下，为了跟上计算核心的处理速度，数据流的速度必须足够快，这一点将成为性能瓶颈。例如，若我们尝试将上述某个示例程序应用于一个没有缓存在主存的文件，或者是一个通过高带宽存储阵列传输的文件，那么大部分时间都将浪费在等待磁盘 I/O 上，而无法利用多核的优势。

CHAPTER

TWENTYFOUR

第 25 章：性能剖析与优化

Haskell 是一门高级编程语言，一门真正的高级编程语言。我们可以一直使用抽象概念、幺半群、函子、以及多态进行编程，而不必与任何特定的硬件模型打交道。Haskell 在语言规范方面下了很大的功夫，力求语言可以不受制于某个特定的求值模型。这几层抽象使得我们可以把 Haskell 作为计算本身的记号，让编程人员关心他们问题的关键点，而不用操心低层次的实现细节，使得人们可以心无旁骛地进行编程。

但是，本书介绍的是真实世界中的编程行为，而真实世界中的代码都运行在资源有限的硬件之上，并且程序也会有时间和空间上的限制。因此，我们需要掌握好程序数据的底层结构，准确地理解使用惰性求值和严格求值策略带来的后果，并学会如何分析和控制程序在时间和空间上的行为。

在这一章，我们将会去看一些 Haskell 编程中常见的空间和时间问题，并且学习如何对它们进行有条理地分析，最后理解并解决它们。为此我们将研究使用一系列的技术：时间和空间性能剖析，运行时统计，以及对严格求值和惰性求值进行推断。我们也会看下编译器优化对性能的影响，以及在纯函数式编程语言中可行的高级优化技术的应用。那么，让我们用一个挑战开始吧：调查一个看似无害的程序中出乎意料的内存使用的问题。

24.1 Haskell 程序性能剖析

请看下面这个列表处理程序，它用于计算某个超长列表的平均值。这里展示的只是程序的其中一部分代码（并且具体的实现算法我们并不关心），这是我们经常会在真实的 Haskell 程序中看到的典型的简单列表操作代码，这些代码大量地使用标准库函数，并且包含了一些因为疏忽大意而导致的性能问题。这里也展示了几种因疏忽而易出现的性能问题。

```
-- file: ch25/A.hs
import System.Environment
import Text.Printf

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean [1..d])
```

(continues on next page)

(continued from previous page)

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

这个程序非常简单：我们引用了访问系统环境的函数（即 `getArgs`），和 Haskell 版的 `printf` 函数来格式化输出。接着这个程序从命令行读入一个数字来构建一个由浮点数组成的列表。我们用这个列表的和除以列表的长度得到平均值，然后以字符串的形式打印出来。我们来将此代码编译成机器代码（打开优化开关）然后用 `time` 命令执行它看看情况吧：

```
$ ghc --make -rtsopts -O2 A.hs
[1 of 1] Compiling Main             ( A.hs, A.o )
Linking A ...
$ time ./A 1e5
50000.5
./A 1e5  0.05s user 0.01s system 102% cpu 0.059 total
$ time ./A 1e6
500000.5
./A 1e6  0.26s user 0.04s system 99% cpu 0.298 total
$ time ./A 1e7
5000000.5
./A 1e7  63.80s user 0.62s system 99% cpu 1:04.53 total
```

程序在处理少量元素时运行得非常好，但是当输入的列表元素数量达到一千万个时，程序的运行速度就会变得相当缓慢。从这点我们就能感觉到应该有什么地方做得不对，但我们并不清楚它的资源使用情况。我们需要研究下。

24.1.1 收集运行时统计信息

为了获得这些信息，GHC 支持直接向 Haskell 运行时传入一些标志 (flags)，如 +RTS 和 -RTS。这些特殊的标志是传递给 Haskell 的运行时系统的保留参数，会直接被运行时系统处理掉。而应用程序本身并不会看到这些标志。

特别地，我们可以用 `-s` 标志来让运行时系统收集内存和垃圾收集器 (garbage collector，以下也会用 GC 来简称) 的性能参数（并可以用 `-N` 来控制系统线程的数量，或调整栈和堆的大小）。我们将用各种运行时标志来启动不同的性能剖析。Haskell 运行时能够接受的所有标志列表可以参见 GHC 用户手册。

那么让我们用 `+RTS -sstderr` 来运行程序取得我们所需要的结果吧。

```
$ ./A 1e7 +RTS -sstderr
5000000.5
1,689,133,824 bytes allocated in the heap
697,882,192 bytes copied during GC (scavenged)
465,051,008 bytes copied during GC (not scavenged)
382,705,664 bytes maximum residency (10 sample(s))
```

(continues on next page)

(continued from previous page)

```
     3222 collections in generation 0 (  0.91s)
       10 collections in generation 1 ( 18.69s)

      742 Mb total memory in use

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.63s  (  0.71s elapsed)
  GC    time   19.60s  ( 20.73s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time   20.23s  ( 21.44s elapsed)

  %GC time      96.9%  (96.7% elapsed)

  Alloc rate    2,681,318,018 bytes per MUT second

  Productivity   3.1% of total user, 2.9% of total elapsed
```

当使用 `-sstderr` 时，程序的性能数字会输出到标准错误流里，告诉我们很多关于程序在具体做什么的信息。尤其是，它告诉了我们 GC 占用了多少时间，以及最大活动内存的使用情况。原来，为了计算一千万个元素的平均值，程序在堆上使用了多达 742M 的内存，并且 96.9% 的时间是花费到垃圾收集器上的！总的来说，只有 3.1% 的时间是程序用来干正事的。

那么为什么我们的程序运行情况这么差？我们如何来提高它呢？毕竟，Haskell 是一个惰性语言：它不应该只用恒定的内存空间来处理列表吗？

24.1.2 时间剖析

幸运的是，GHC 为我们提供了多种工具来剖析程序的时间和空间使用情况。我们可以在编译程序时打开性能剖析标志，这样程序在执行时会生成每个函数的资源使用信息。性能剖析可以按照下面三个步骤来进行：用性能剖析标志编译程序；执行程序时用运行时标志打开特定的性能剖析模式；最后分析收集到的统计信息。

编译程序时，我们可以使用 `-prof` 性能剖析标志来得到基本的时间和空间消耗信息。我们也需要给感兴趣的函数标记为“消耗集中点 (cost centres)”以便让性能剖析程序知晓。一个消耗集中点即是一个信息收集点，GHC 会生成代码来计算在这些地方执行的表达式的消耗情况。我们可以用 `SCC` 编译指示 (pragma) 把任何表达式设为消耗集中点。

```
-- file: ch25/SCC.hs
mean :: [Double] -> Double
mean xs = {-# SCC "mean" #-} sum xs / fromIntegral (length xs)
```

或者，我们也可以用 `-auto-all` 标志来让编译器将所有顶层函数设为消耗集中点。然后在我们识别出某个性能热点 (hot spot) 的函数之后，把手动添加消耗集中点作为一个十分有用的补充，就可以更为精确地去侦

测该函数的子表达式了。

需要注意的一个复杂的地方：在 Haskell 这类惰性、纯函数式编程语言里，没有参数的值只会被计算一次（比如之前计算超长列表的程序），然后计算的结果会在之后共享。于是这种函数在调用关系图 (call graph) 中记录的统计值并不准确，因为它们并不是每次调用都执行。然而，我们仍然想要知道它们一次执行的消耗情况是怎么样。为了得到这种被称为“常量函数体 (Constant Applicative Forms)”或 CAF 的确切值，我们可以使用 `-caf-all` 标志。

那么以性能剖析的方式来编译我们的程序吧（用 `-fforce-recomp` 标志来强制重新编译所有部分）：

```
$ ghc -O2 --make A.hs -prof -auto-all -caf-all -fforce-recomp
[1 of 1] Compiling Main             ( A.hs, A.o )
Linking A ...
```

现在我们可以执行这个标记了性能剖析点的程序了（标记了性能剖析的程序会变慢，所以我们用一个较小的输入来执行）：

```
$ time ./A  1e6 +RTS -p
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize' to increase it.
./A 1e6 +RTS -p  1.11s user 0.15s system 95% cpu 1.319 total
```

程序竟然把栈空间耗完了！这就是使用性能剖析时需要注意的主要影响：给程序加消耗集中点会改变它的优化方式，进而可能影响它的运行时表现，因为每一个被标记的表达式都会被附加一段额外的代码，以此来跟踪它们的执行轨迹。从某种意义上说，观察程序执行会影响它的执行。对于我们这样情况，修正起来很简单——只需要通过 GHC 运行时标志 `-K` 来增加栈空间上限即可（要附带指示存储单位的后缀）：

```
$ time ./A 1e6 +RTS -p -K100M
500000.5
./A 1e6 +RTS -p -K100M  4.27s user 0.20s system 99% cpu 4.489 total
```

运行时会将性能信息生成到一个名字为“A.prof“（以可执行程序的名字命名）的文件中。其中含有以下信息：

```
$ cat A.prof

Time and Allocation Profiling Report  (Final)

       A +RTS -p -K100M -RTS 1e6

    total time  =        0.28 secs   (14 ticks @ 20 ms)
    total alloc = 224,041,656 bytes  (excludes profiling overheads)

COST CENTRE  MODULE                %time %alloc

CAF:sum      Main                   78.6   25.0
```

(continues on next page)

(continued from previous page)

```
CAF           GHC.Float              21.4   75.0

                                          individual     inherited
COST CENTRE MODULE          no.     entries  %time %alloc   %time %alloc

MAIN        MAIN              1           0   0.0    0.0   100.0  100.0
 main       Main            166           2   0.0    0.0     0.0    0.0
  mean      Main            168           1   0.0    0.0     0.0    0.0
 CAF:sum    Main            160           1  78.6   25.0    78.6   25.0
 CAF:lvl    Main            158           1   0.0    0.0     0.0    0.0
  main      Main            167           0   0.0    0.0     0.0    0.0
 CAF        Numeric         136           1   0.0    0.0     0.0    0.0
 CAF        Text.Read.Lex   135           9   0.0    0.0     0.0    0.0
 CAF        GHC.Read        130           1   0.0    0.0     0.0    0.0
 CAF        GHC.Float       129           1  21.4   75.0    21.4   75.0
 CAF        GHC.Handle      110           4   0.0    0.0     0.0    0.0
```

这些信息给我们描述了程序在运行时的行为。里面包含了程序的名字以及执行程序时用到的标志和参数。“total time”是运行时系统视角所见的程序运行的确切总时长。“total alloc”是程序在运行过程中分配的内存总字节数（不是程序运行时内存使用的峰值；那个峰值大概是 700MB）

报告中的第二段是各个函数所消耗的时间和空间部分。第三段是消耗集中点的报告，它被组织成了调用关系图的格式（比如，我们可以看出 `mean` 是被 `main` 调用的）。“individual”和“inherited”列提供了每个消耗集中点其本身、以及它和它的子部分所消耗的资源。此外，最下面那些 `CAF` 是常量执行的一次性消耗（例如超长列表中浮点数以及列表本身）。

我们能从这些信息得出什么结论呢？我们可以看出两个 `CAF` 占用了大多数时间：一个与计算总和相关，另一个与浮点数相关。光是它们就几乎占据了程序运行期间的所有消耗。结合我们之前观察到 GC 的压力问题，看起来像是在列表节点的内存分配和浮点数值上发生了问题。

简单的性能热点检测，特别是对于我们难以知道时间花费点的大型程序，这个时间剖析会突出某些问题模块或顶层函数。这往往已足够显示出问题所在了。跟上面展示的程序一样，一旦我们缩小了问题代码的范围，我们就可以用更加尖端的剖析工具来拿到更多的信息。

24.1.3 空间剖析

GHC 除了可以进行基本的时间和空间统计外，还能为程序整个执行期间的堆内存使用情况生成图表。这能完美检测内存泄露问题。内存泄露是指不再需要的内存没有被释放。这会对 GC 造成压力，就像在我们的程序中见到的那样。

构建堆内存剖析和构建一般的时间剖析的步骤是一样的，都需要用到 `-prof -auto-all -caf-all` 编译标志。但当执行程序时，我们会让运行时系统收集关于堆使用的最多细节。堆使用信息能够以几种方式分解：消耗集中点、模块、构造器和数据类型。每个都有各自的洞见。对 `A.hs` 进行堆内存剖析所得的原始数据会

被记录到一个名为 A.hp 的文件里面，之后只要使用 hp2ps 处理这个文件，就可以得到一个堆内存占用历史图像的 PostScript 文件。

想要使用标准的堆内存剖析的话，可以在运行程序时使用 -hc 作为运行时的性能剖析标志：

```
$ time ./A 1e6 +RTS -hc -p -K100M
500000.5
./A 1e6 +RTS -hc -p -K100M  4.15s user 0.27s system 99% cpu 4.432 total
```

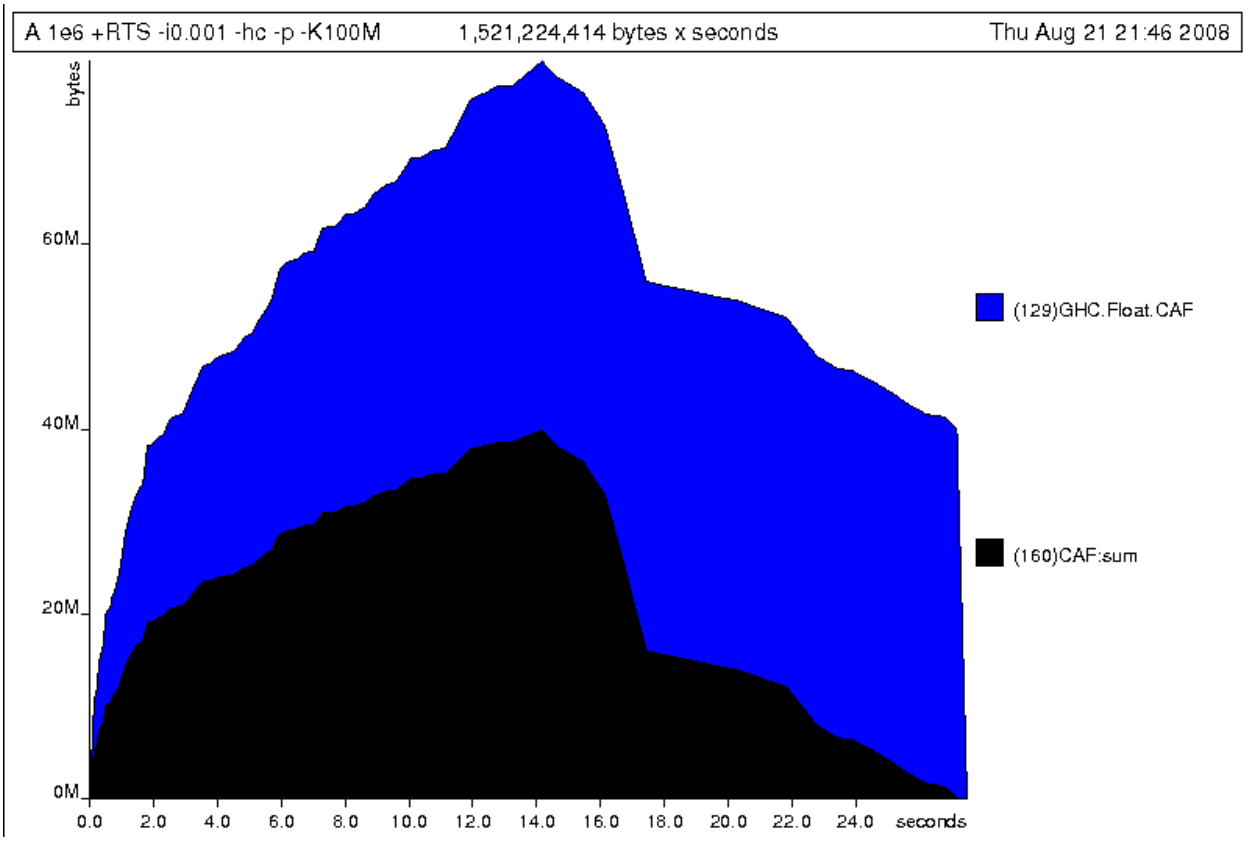
一个堆内存剖析的日志文件 A.hp 会创建出来，其内容为以下形式：

```
JOB "A 1e6 +RTS -hc -p -K100M"
SAMPLE_UNIT "seconds"
VALUE_UNIT "bytes"
BEGIN_SAMPLE 0.00
END_SAMPLE 0.00
BEGIN_SAMPLE 0.24
(167)main/CAF:lvl   48
(136)Numeric.CAF    112
(166)main   8384
(110)GHC.Handle.CAF 8480
(160)CAF:sum    10562000
(129)GHC.Float.CAF  10562080
END_SAMPLE 0.24
```

这些样本是在程序运行期间以固定的间隔采样出来的。我们可以用 -iN 标志来增加采样频率，这里的 N 是内存采样之间相隔的秒数（如 0.01 秒）。很明显，采样越频繁，得到的结果越精确，但程序也会执行得越慢。我们可以用 hp2ps 将剖析结果生成一张图表：

```
$ hp2ps -e8in -c A.hp
```

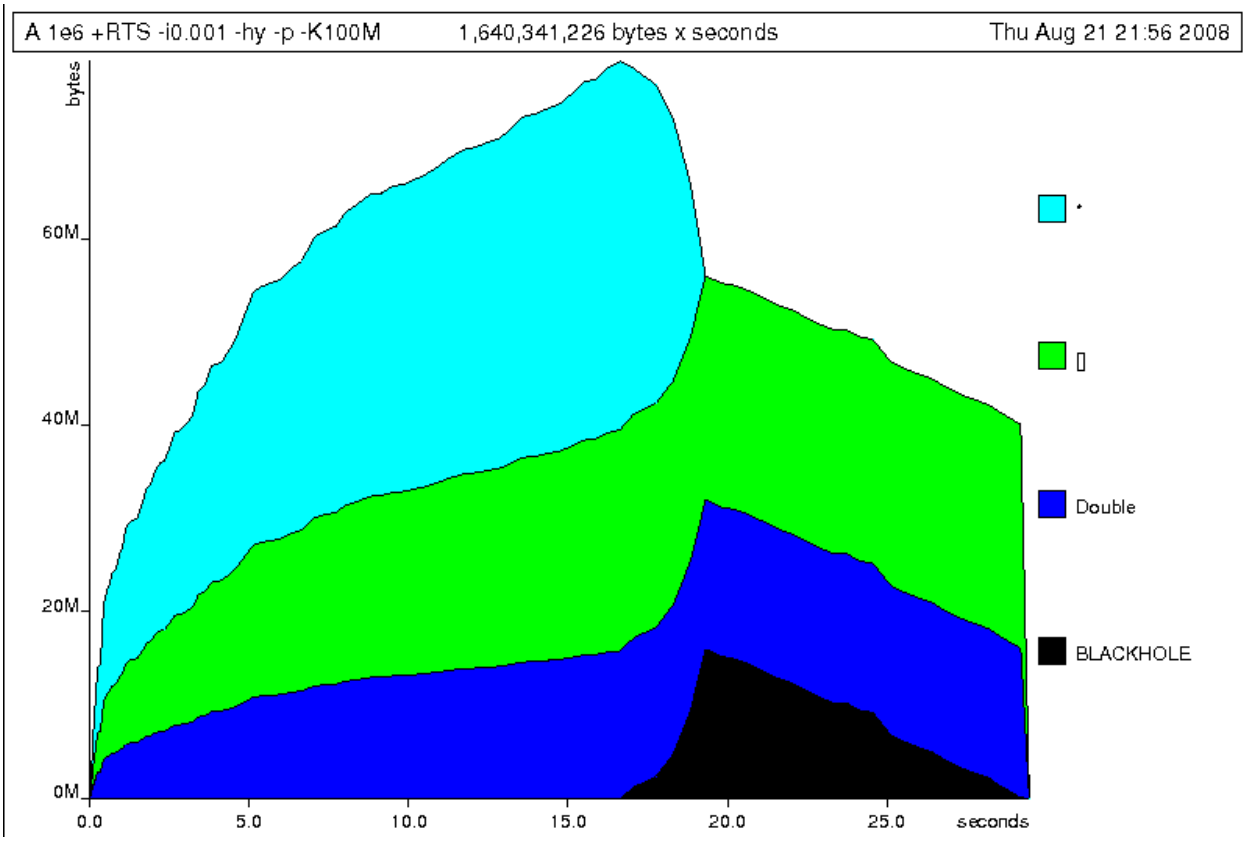
这就是生成的图表 A.ps：



我们能从图片中看出什么？举个例子，程序的执行分为两个阶段：前一阶段在计算数值总和的同时不断分配大量的内存，后一阶段清理释放这些内存。内存初始化分配的同时，sum 也开始工作，并消耗大量的内存。如果用性能剖析标志 -hy 来按类型分解的话，我们会得到一个稍有不同的图像：

```
$ time ./A 1e6 +RTS -hy -p -K100M
500000.5
./A 1e6 +RTS -i0.001 -hy -p -K100M  34.96s user 0.22s system 99% cpu 35.237 total
$ hp2ps -e8in -c A.hp
```

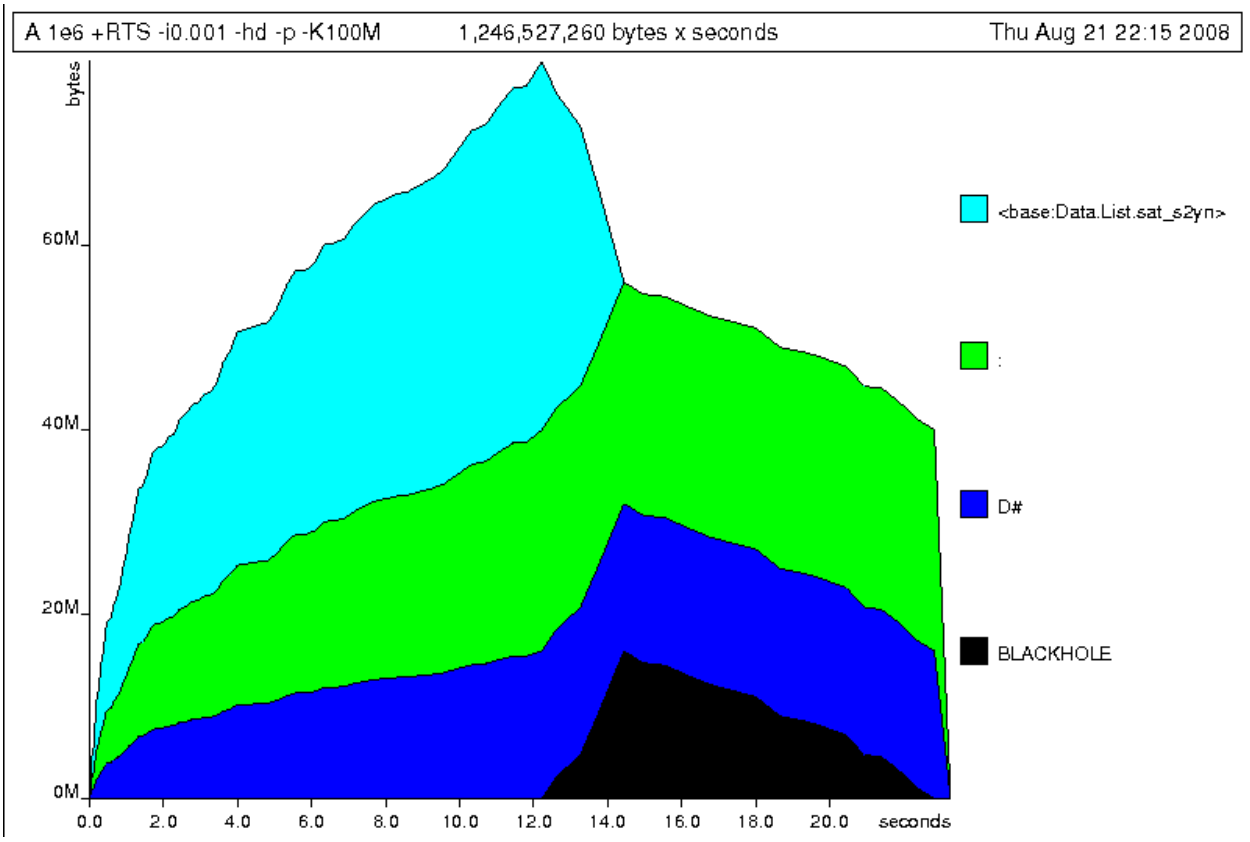
以下是生成的图像：



这里最有趣的是很大部分的内存都被 list 类型（即“[]”）和 Double 类型所占用；我们看到未知类型（图中用“*”标记）也占用了一些内存。最后，再让我们用 `-hd` 标志，根据构造器对结果进行分解：

```
$ time ./A 1e6 +RTS -hd -p -K100M
$ time ./A 1e6 +RTS -i0.001 -hd -p -K100M
500000.5
./A 1e6 +RTS -i0.001 -hd -p -K100M  27.85s user 0.31s system 99% cpu 28.222 total
```

下面就是能够展示程序执行的所有情况的最终图像：



程序在分配双精度浮点数列表上面花了不少功夫。列表在 Haskell 语言中是惰性的，所以含有上百万个元素的列表都是在程序执行过程中一点点地构建出来的。但这些元素在被遍历的同时并没有被逐步释放，所以导致内存占用变得越来越大。最终，在程序执行稍稍超过一半时，终于将列表总和计算出来，并开始计算其长度。如果看下关于 `mean` 的程序片断的话，我们就会知道内存没被释放的确切原因：

```
-- file: ch25/Fragment.hs
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

首先我们计算列表的总和，这会使得所有列表元素被分配到内存。但我们现在还不能释放列表元素，因为 `length` 还需要整个列表。一旦 `sum` 结束，`length` 会马上开始访问列表，同时 GC 会跟进，逐步释放列表元素，直到 `length` 结束。这两个计算阶段展示了两种明显不同的分配与释放，并指出我们需要改进的确切思路：只对列表遍历一次，遍历过程中同时计算总和与平均值。

24.2 控制求值

如果我们有很多方式来实现一个只遍历一次的循环。例如，我们可以写一个对列表折叠 (fold) 的循环，也可以对列表进行显式的递归。本着使用更高级的方法去解决问题的原则，我们决定先尝试折叠的方式：

```
-- file: ch25/B.hs
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)      = foldl k (0, 0) xs
    k (n, s) x = (n+1, s+x)
```

这次，我们不再采取“计算列表的总和，并保留这个列表直到获取它的长度为止”这一方案，而是左折叠 (left-fold) 整个列表，累加当前的总和及长度到 pair(对组) 上（我们必须采用左折叠，因为右折叠 (right-fold) 会先带我们到列表的结尾，然后倒回来计算，这恰恰是我们想要避免的）。

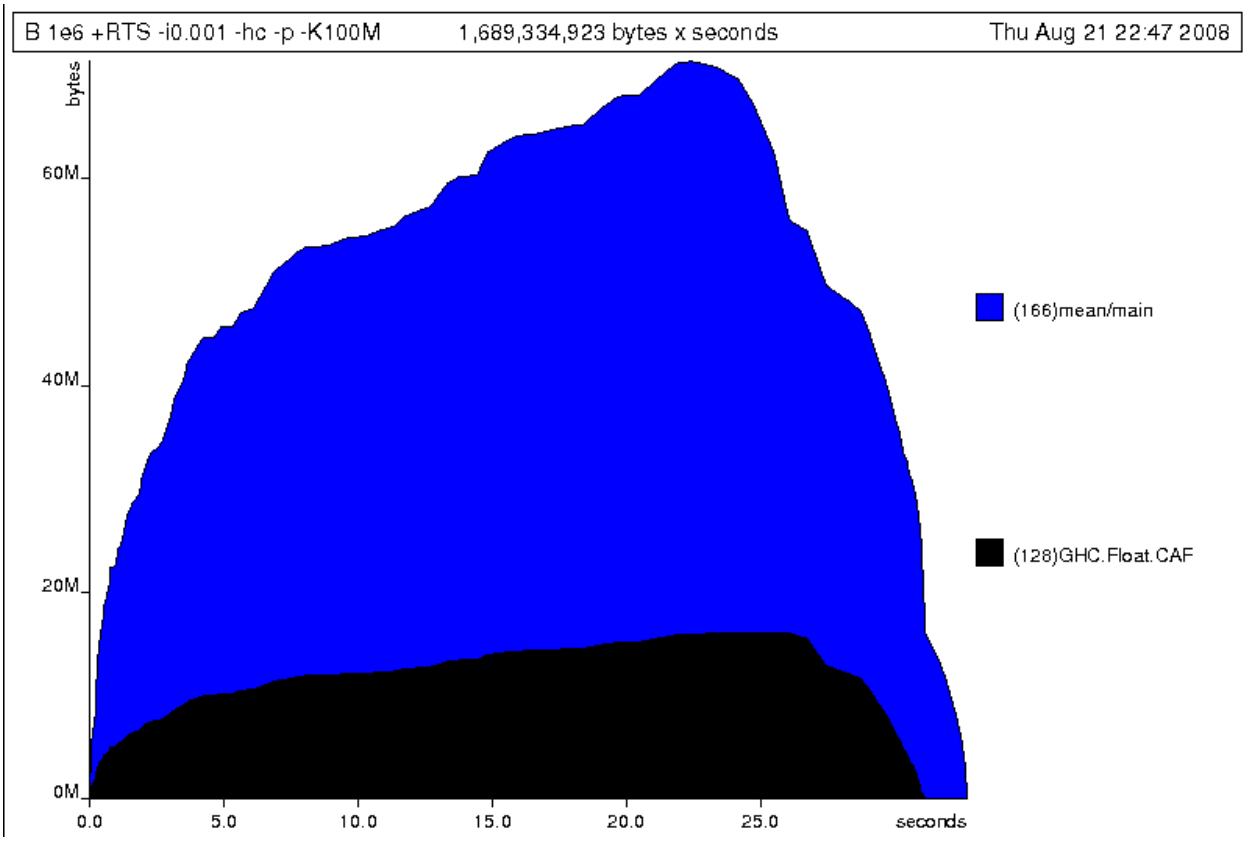
循环的主体是 k 函数，把即时的循环状态和当前的列表元素作为参数，然后返回新状态——长度增一、总和加当前元素。然而，当我们运行它时发生了栈溢出：

```
$ ghc -O2 --make B.hs -fforce-recomp
$ time ./B 1e6
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize' to increase it.
./B 1e6  0.44s user 0.10s system 96% cpu 0.565 total
```

我们把堆消耗换成栈消耗了！事实上，如果我们用 -K 标志增加栈的大小到前面堆的大小，程序就能够运行完成，并生成相似的内存分配图：

```
$ ghc -O2 --make B.hs -prof -auto-all -caf-all -fforce-recomp
[1 of 1] Compiling Main             ( B.hs, B.o )
Linking B ...
$ time ./B 1e6 +RTS -i0.001 -hc -p -K100M
500000.5
./B 1e6 +RTS -i0.001 -hc -p -K100M  38.70s user 0.27s system 99% cpu 39.241 total
```

从生成的堆剖析结果中，我们可以看到现在 mean 的整个内存分配状况：



问题是：为什么即使采用折叠的方式，程序仍然会引发越来越多的内存分配呢？其实，这就是典型的极度惰性 (excessive laziness) 带来的空间泄露问题。

24.2.1 严格执行和尾递归

产生问题的原因是，我们的左折叠函数 `foldl` 是惰性的。我们想要的是一个尾递归循环，实现的像 goto 一样高效而没有保留在栈上的状态。而现在的情况并不是在每一步都会消掉状态元组，而是产生一个 thunk 的长链，只会在程序结束时才会求值。在任何时候我们都没有要求减少循环状态，所以编译器无法推断出什么时候必须严格执行，以减少纯惰性的值。

所以，我们要稍微地调整求值的策略：惰性地展开列表，但是严格地累加折叠状态。标准方法是使用 `Data.List` 模块的 `foldl'` 替换 `foldl`：

```
-- file: ch25/C.hs
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)      = foldl' k (0, 0) xs
    k (n, s) x = (n+1, s+x)
```

然而这个程序的运行方式仍然和我们想象中的不一样：

```
$ ghc -O2 --make C.hs
[1 of 1] Compiling Main             ( C.hs, C.o )
Linking C ...
$ time ./C 1e6
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize' to increase it.
./C 1e6  0.44s user 0.13s system 94% cpu 0.601 total
```

计算还是不够严格！我们的循环还是继续在栈上累积没有求值的折叠状态。这里的问题是 `foldl'` 只在外部严格执行：

```
-- file: ch25/Foldl.hs
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z xs = lgo z xs
    where lgo z []     = z
          lgo z (x:xs) = let z' = f z x in z' `seq` lgo z' xs
```

这个循环虽然使用 `seq` 消减每步的累加状态，但是只对循环状态上最外部的 pair 构造器进行了严格执行。也就是说，`seq` 把一个表达式消减到 “weak head normal form”，仅仅对第一个匹配的构造器严格求值。在这种情况下，对于最外部的构造器元组 `(,)` 来说深度是不够的。也就是说，现在的问题是 pair 中的元素仍然在未求值状态。

24.2.2 增加严格执行

有很多方式可以使函数完全地严格执行。例如，我们可以自己在 pair 的内部补充上严格求值的代码，就可以得到一个真正的尾递归循环：

```
-- file: ch25/D.hs
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)      = foldl' k (0, 0) xs
    k (n, s) x = n `seq` s `seq` (n+1, s+x)
```

在这次修改中，我们深入到状态元组中，明确地告诉编译器状态的各个部分在每一步都应该消耗掉。最终，我们得到一个常量内存空间的版本：

```
$ ghc -O2 D.hs --make
[1 of 1] Compiling Main             ( D.hs, D.o )
Linking D ...
```

在打开内存分配统计的情况下运行这个程序，我们终于得到了满意的结果：

```
$ time ./D 1e6 +RTS -sstderr
./D 1e6 +RTS -sstderr
500000.5
256,060,848 bytes allocated in the heap
     43,928 bytes copied during GC (scavenged)
     23,456 bytes copied during GC (not scavenged)
     45,056 bytes maximum residency (1 sample(s))

        489 collections in generation 0 (  0.00s)
          1 collections in generation 1 (  0.00s)

          1 Mb total memory in use

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.12s  (  0.13s elapsed)
  GC    time    0.00s  (  0.00s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    0.13s  (  0.13s elapsed)

  %GC time       2.6%  (2.6% elapsed)

  Alloc rate    2,076,309,329 bytes per MUT second

  Productivity  97.4% of total user, 94.8% of total elapsed

./D 1e6 +RTS -sstderr  0.13s user 0.00s system 95% cpu 0.133 total
```

不像我们的第一个版本那样，这个程序的计算效率是 97.4%，在 GC 上仅花费 2.6% 的时间，并且运行的内存是常量 1 兆。它很好地展示了如何在同时使用严格计算和惰性计算之间取得平衡：对大型列表惰性展开，而在展开中严格求值。这样就能得到一个使用常量的空间，并且运行速度很快的程序。

24.2.3 Normal form reduction

我们也有许多其他方法可以解决这里的严格执行问题。比如对于深度严格求值，我们也可以使用并行策略库 (parallel strategies library) 里的 `rnf` 函数（以及与之相搭配的 `using`）。不像 `seq` 那样，`rnf` 能够对表达式彻底求值成“normal form”（正如它的名字，即“reduced normal form”）。使用“深度 `seq`”我们可以这样重写折叠代码：

```
-- file: ch25/E.hs
import System.Environment
import Text.Printf
import Control.Parallel.Strategies

```

(continues on next page)

(continued from previous page)

```
main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean [1..d])

foldl'rnf :: NFData a => (a -> b -> a) -> a -> [b] -> a
foldl'rnf f z xs = lgo z xs
    where
        lgo z []     = z
        lgo z (x:xs) = lgo z' xs
            where
                z' = f z x `using` rnf

mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)      = foldl'rnf k (0, 0) xs
    k (n, s) x = (n+1, s+x) :: (Int, Double)
```

我们修改了 `foldl'` 的实现，使用 `rnf` 策略把状态规约到 normal form。这样也引入了一个我们先前可以避免的问题：循环状态的类型推导。程序原来可以通过默认的类型推导，把循环状态中列表的长度推导成数值整型。但是在切换到 `rnf` 后，由于引入了 `NFData` 类型类的约束，我们就无法再依赖默认的类型推导了。

24.2.4 Bang patterns

为一个极其惰性的代码添加严格执行，从语法修改的成本上来讲，大概最廉价的方式就是“bang patterns”了（它的名字来自符号“!”，发音是“bang”）。我们可以用下面的编译指示引入的这个语言扩展：

```
-- file: ch25/F.hs
{-# LANGUAGE BangPatterns #-}
```

通过 bang patterns 我们可以在把严格执行指示到任何 binding form 上，从而使函数在那个变量上严格执行。和显示的类型标注可以指导类型推断一样，bang patterns 可以帮助指导推断严格执行。现在我们可以把循环状态重写成更为简单的形式：

```
-- file: ch25/F.hs
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)       = foldl' k (0, 0) xs
    k (!n, !s) x = (n+1, s+x)
```

因为循环状态中的临时变量都是严格执行的，所以这个循环可以在常量空间中执行：

```
$ ghc -O2 F.hs --make
$ time ./F 1e6 +RTS -sstderr
./F 1e6 +RTS -sstderr
500000.5
256,060,848 bytes allocated in the heap
     43,928 bytes copied during GC (scavenged)
     23,456 bytes copied during GC (not scavenged)
     45,056 bytes maximum residency (1 sample(s))

        489 collections in generation 0 (  0.00s)
          1 collections in generation 1 (  0.00s)

          1 Mb total memory in use

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.14s  (  0.15s elapsed)
  GC    time    0.00s  (  0.00s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    0.14s  (  0.15s elapsed)

  %GC time       0.0%  (2.3% elapsed)

  Alloc rate    1,786,599,833 bytes per MUT second

  Productivity 100.0% of total user, 94.6% of total elapsed

./F 1e6 +RTS -sstderr  0.14s user 0.01s system 96% cpu 0.155 total
```

在大型项目里，当我们正在侦查内存分配的热点时，bang patterns 是最简单的一种试探性地修改代码的严格执行属性的方式。因为它对语法的侵略性与其他方法相比更小。

24.2.5 严格的数据类型

严格的数据类型 (strict data type) 是另一个有效的方式提供给编译器严格执行的信息。默认 Haskell 的数据类型都是惰性的，但是很容易通过为数据类型的字段添加严格执行的信息，使严格执行推广到整个程序中。例如，我们可以声明一个新的严格的 pair 类型：

```
-- file: ch25/G.hs
data Pair a b = Pair !a !b
```

这样定义的 pair 类型总是会将字段保存成 weak head normal form。我们现在重写循环：

```haskell
-- file: ch25/G.hs
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    Pair n s       = foldl' k (Pair 0 0) xs
    k (Pair n s) x = Pair (n+1) (s+x)
```

这个实现再次不仅同样高效，而且只占用常量空间。到了这个地方，为了能够从这个代码里榨干最后几滴性能，我们还需要进一步深入下去。

24.3 理解核心语言

除了查看运行时剖析的数据外，一个可靠地了解你的程序运行过程的方法是，查看编译器优化后的最终程序源码。特别是在 Haskell 编译器会对源码进行各种非常激进的转换 (transform) 的情况下，这一方法将非常有效。GHC 使用一种被幽默地称为“一门简单的函数式语言”的核心语言 (Core，以下也会译成“核心代码”) 来作为编译器的中间表示。它实质上是适合代码生成的 Haskell 语言的子集，并扩增了一种未封装数据类型 (unboxed data type)，也就是原生机器类型。就像 C 语言一样直接对应着机器的基本数据类型 (primitive data type)。GHC 通过转换来优化 Haskell，反复重写源码以得到更有效率的形式。在被翻译成底层命令式代码之前，核心代码就是你的代码在函数式上的最终版。换句话说，核心代码具有最终发言权。如果你的目标是让程序到达终极性能的话，那么你很值得去掌握它。

[译注：对于未封装类型 (unboxed types) 的解释，从 wiki 上做以下简单摘要：GHC 中的大多数类型是封装的 (boxed)，也就是意味着这个类型的值实际上是用一个指向堆对象 (heap object) 的指针表示的。比如 Haskell 中的 Int，其实是一个占用两个字大小的堆对象。而对于一个未封装类型，它的值就是它本身，既没有指针也没有在堆上的内存分配。]

为了查看我们的 Haskell 程序的核心代码，需要编译时使用 `-ddump-simpl` 标志。也可以使用一个叫做 ghc-core 的第三方工具，使我们能够分页地查看核心代码。

```
$ ghc -O2 -ddump-simpl G.hs
```

生成了满屏幕的文本。仔细看一下，我们会发现一个循环（在此为了能够看得清晰，就稍微清理了一下）

```haskell
lgo :: Integer -> [Double] -> Double# -> (# Integer, Double #)

lgo = \ n xs s ->
    case xs of
      []       -> (# n, D# s #);
      (:) x ys ->
        case plusInteger n 1 of
            n' -> case x of
                D# y -> lgo n' ys (+## s y)
```

这就是 `foldl'` 的最终版本，它告诉了我们很多下一步的优化信息。折叠已经完全内联，生成一个显式的列表递归循环。循环状态，也就是我们的严格执行的 pair 已经完全消失。现在函数已经把列表、长度以及总和一起作为参数使用。

列表元素的总和被表示成未封装的 Double# 类型，在原生机器里，会被保存在一个浮点寄存器上。这样做非常好！因为这样就可以避免使用堆上的变量而带来的一系列内存操作。然而对于列表的长度，因为我们没有给它显式的类型标注，所以它被推断为分配在堆上的 Integer，并附带了一个非基本数据类型需要的 `plusInteger` 实施相加操作。如果在算法的合理性上我们可以用 Int 替代 Integer，那么通过使用类型标注，GHC 就会用一个原生机器的 Int# 类型作为长度。通过确保循环状态的分量都是保存在寄存器上的未封装类型，程序的时间和空间性能将有机会获得更大的提升。

循环的基础 case 分支，也就是循环的结束，生成一个未封装 pair（只分配在寄存器上的 pair），保存着列表的最终长度以及累加的总和。注意返回类型是堆分配的 Double 类型，这点可以从 D# 构造器看的出来。所以它在堆内存上占据一个原生的双精度值大小的空间。再次，这里也会对性能产生影响，因为在为它分配内存和返回循环之前，GHC 需要检查是否有足够的堆空间。

我们可以避免这个最终的堆检查，通过使 GHC 返回一个未封装的 Double# 类型。这点可以用一个自定义的 pair 类型实现。此外，GHC 提供了一个对数据类型中的严格字段不封装的优化，这样就保证了新 pair 类型的字段会被保存在寄存器上。这个优化通过 `-funbox-strict-fields` 标志开启。

对于上面提到的两个代码改进，我们可以通过使用固定有 Int 和 Double 类型的字段的 pair 数据类型，来替换原先的多态严格的 pair 来一并完成。

```
-- file: ch25/H.hs
data Pair = Pair !Int !Double

mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    Pair n s       = foldl' k (Pair 0 0) xs
    k (Pair n s) x = Pair (n+1) (s+x)
```

带着优化标志和 `-funbox-strict-fields -ddump-simpl` 编译程序，我们得到一个拥有更紧密的内循环的核心代码：

```
lgo :: Int# -> Double# -> [Double] -> (# Int#, Double# #)
lgo = \ n s xs ->
    case xs of
      []       -> (# n, s #)
      (:) x ys ->
        case x of
            D# y -> lgo (+# n 1) (+## s y) ys
```

现在，循环状态的 pair 已经作为未封装的基本类型来表示和返回了，也就是说它将会保存在寄存器上使用。最终版本的程序只剩下列表的节点被分配在堆内存上，这样做是为了满足列表的惰性计算需要。编译和运行

这个调优过的版本，我们可以和原来的程序对比一下内存分配和时间的性能。

```
$ time ./H 1e7 +RTS -sstderr
./H 1e7 +RTS -sstderr
5000000.5
1,689,133,824 bytes allocated in the heap
    284,432 bytes copied during GC (scavenged)
         32 bytes copied during GC (not scavenged)
     45,056 bytes maximum residency (1 sample(s))

       3222 collections in generation 0 (  0.01s)
          1 collections in generation 1 (  0.00s)

          1 Mb total memory in use

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.63s  (  0.63s elapsed)
  GC    time    0.01s  (  0.02s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    0.64s  (  0.64s elapsed)

  %GC time       1.0%  (2.4% elapsed)

  Alloc rate    2,667,227,478 bytes per MUT second

  Productivity  98.4% of total user, 98.2% of total elapsed

./H 1e7 +RTS -sstderr  0.64s user 0.00s system 99% cpu 0.644 total
```

当操作的列表包含 1 千万个元素时，原来的程序运行了超过 1 分钟的时间，并且分配了超过 700M 的内存。然而使用一个简单的高阶折叠，以及一个严格的数据类型的最终版本的程序，却运行了大概半秒钟，并分配了总共 1M 内存。这可真是相当大的进步啊！

我们可以从剖析和优化过程中学到的通用规则有：

- 编译成本地程序时，要打开优化标志
- 当有疑虑，就使用运行时统计和时间剖析
- 如果怀疑有内存分配的问题，就使用堆剖析
- 小心翼翼地结合严格和惰性执行会产生最好的结果
- 最好对原子数据类型（如 Int、Double 等相似类型）的字段标记成严格执行
- 要使用能够表示成原生机器的基本数据类型的数据类型（最好用 Int 而不是 Integer）

这些简单的规则足以定位和消除麻烦的内存占用问题，正确地使用这些规则还能够有效地防范这些问题。

24.4 高级技术：fusion(融合)

这个程序最后的性能瓶颈是惰性列表本身。虽然我们避免了立刻把它所需的内存全部分配出来，但也因此带来了每次循环时内存操作的消耗。因为在循环中我们要取下一个列表的头元素，在堆上分配内存给它，操作，然后继续。这个列表类型也是多态的，所以它的元素将表示成分配在堆上的 Double 值。

我们想做的是完全去掉列表，仅在寄存器上保存我们需要的下一个元素。可能会使人感到意外，GHC 能够把一个包含列表的程序转换成无列表的形式，用一个被称为 deforestation 的优化。这个优化是指一类去掉中间数据结构的优化。得益于 Haskell 代码不包含副作用，Haskell 编译器有时会非常激进的大规模地重排、调整顺序和转换代码。这次我们使用的 deforestation 优化叫做 stream fusion(流融合)。

这个优化可以把迭代列表生成 (recursive list generation) 和转换函数 (transformation functions) 转换成非迭代的展开 (non-recursive unfolds)。当一个展开靠近一个折叠时，它们之间的结构会被去掉，然后生成一个无需堆内存分配的单一紧凑的循环。这个优化默认是没有开启的，因为它会急剧地增加一段代码的复杂度。但是它在许多数据结构库上都开启了。通过编译器提供的“重写规则”，自定义优化会应用到这些库的导出函数上。

我们将使用 `uvector` 库，这个库提供了具有一整套类似列表操作的数据类型。但与列表不同的是这些操作使用 stream fusion 移除了中间的数据结构。把我们前面的程序改写成使用 streams 优化的代码并不困难：

```
-- file: ch25/I.hs
import System.Environment
import Text.Printf
import Data.Array.Vector

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean (enumFromToFracU 1 d))

data Pair = Pair !Int !Double

mean :: UArr Double -> Double
mean xs = s / fromIntegral n
  where
    Pair n s       = foldlU k (Pair 0 0) xs
    k (Pair n s) x = Pair (n+1) (s+x)
```

从 Hackage 上安装过 `uvector` 库后，编译程序，使用 `-O2 -funbox-strict-fields`，再检查核心代码：

```
fold :: Int# -> Double# -> Double# -> (# Int#, Double# #)
fold = \ n s t ->
    case >## t limit of {
      False -> fold (+# n 1) (+## s t) (+## t 1.0)
      True  -> (# n, s #)
```

这次真的是最优化的结果了！我们的列表结构现在已经完全融化成一个紧凑的循环了。它的列表生成是和循

环状态累加相互交错 (interleaved)，所有的输入和输出变量都保存在寄存器里。运行这个程序，我们会看到程序性能再一次猛进，运行时消耗也下降到另一个数量级：

```
$ time ./I 1e7
5000000.5
./I 1e7  0.06s user 0.00s system 72% cpu 0.083 total
```

24.4.1 调整生成的汇编代码

考虑到现在我们的核心代码已是最优化了，继续深入下去就只剩下汇编代码了。当然，到了这个地步可以获得的优化就剩下非常少了。为了查看生成的汇编，我们可以使用比如像 `ghc-core` 这样的工具，或者对 GHC 使用 `-ddump-asm` 标志把汇编代码生成到标准输出上。我们几乎没有“控制杆”调整生成的汇编，但是我们可以在后端是 C 语言还是原生代码之间做选择。以及，如果我们选 C 语言后端，我们还可以选择把哪些优化标志传给 GCC。特别是在有浮点数代码的情况下，通过 C 语言编译，并开启特定的高性能的 C 编译器优化，会很有用。

例如，通过 `-funbox-strict-fields -fvia-C -optc-O2` 这些标志，我们还可以从最终融合过的循环代码中榨出最后几滴性能。它可以再次减少一半运行时间（因为 C 编译器能够优化掉一些内循环中冗余的移动指令）：

```
$ ghc -fforce-recomp --make -O2 -funbox-strict-fields -fvia-C -optc-O2 I.hs
[1 of 1] Compiling Main             ( I.hs, I.o )
Linking I ...
$ time ./I 1e7
5000000.5
./I 1e7  0.04s user 0.00s system 98% cpu 0.047 total
```

检查最终 x86_64 汇编（通过 `-keep-tmp-files`），我们看到生成的循环只包含 6 个指令：

```
go:
  ucomisd     5(%rbx), %xmm6
  ja  .L31
  addsd       %xmm6, %xmm5
  addq        $1, %rsi
  addsd       .LC0(%rip), %xmm6
  jmp go
```

通过多种源码层的优化我们已经有力地蹂躏了程序，所有这些也促成了最终的汇编代码。至此就没有其他路可以走了。很少有必要优化代码到这个级别。当然，典型情况下只有在写底层的库时才有意义，又或者在已经决定好最终算法的情况下优化特别重要的代码。对日常的代码，选择更好的算法总是更有效的策略。但是清楚如果有必要我们可以优化到底线的程度也是很有用的。

24.5 结论

这一章向我们展示了如何使用一套工具和技术去跟踪和定位代码的问题区域，以及列出了一些可以保持代码简洁高效的惯例。我们的目标是能够从源码、编译器直到底层为止的各个层面上了解代码的具体行为，并在有需要时将注意力集中到特定的层面上。

通过坚持简单的规则、选择正确的数据结构并避免不谨慎的陷阱，即便是在非常高的层次之上进行开发，使用 Haskell 获得高性能也是完全有可能的。最终的结果将会是一个在生产率和无情的性能之间美妙的平衡。

CHAPTER

TWENTYFIVE

第 26 章高级库设计：构建一个布隆过滤器

25.1 布隆过滤器介绍

布隆过滤器（Bloom Filter）是类似集合的一种数据结构，它的特点是空间利用的高效性。布隆过滤器只支持两种操作：插入和成员查询。与常规的集合数据结构不同，布隆过滤器可能会给出不正确的结果。如果我们查询的某个元素存在，布隆过滤器会返回肯定的结果。但是如果我们查询一个之前没有插入过的元素，那么布隆过滤器可能会返回错误的结果，即声称它是存在的。

对大多数应用来说，低概率的误判是可以容忍的。举个例子，网络流量整形（traffic shaper）的主要工作是限制批量传输（比如 BitTorrent），使得一些交互式会话（比如 `ssh` 或者游戏）可以得到优秀的响应时间。流量整形可能会使用布隆过滤器来判断一个特定会话的数据包是批量的还是交互的。如果布隆过滤器在 10000 个批量数据包中误判其中的 1 个为交互式数据包且没有截止，也不会造成任何问题。

布隆过滤器吸引人的地方在于它的空间效率。举个例子，假设现在有一个包含一百万个单词的词典，我们想基于这个词典构建一个拼写检查器，若使用集合数据结构则可能会消耗 20MB 的空间。相比之下，布隆过滤器会消耗大约 0.5MB，代价是漏掉大约 1% 拼错的单词。

布隆过滤器的内部非常简单。它由一个位数组（bit array）和少数哈希函数组成。我们使用 k 表示哈希函数的数量。向布隆过滤器中插入数据时，先用哈希函数为数据计算出 k 个哈希值，然后在位数组中将这些位设置为 1。如果我们想要看看某个数据是否存在，那么就为这个数据计算出 k 个哈希值，然后检查位数组中这些哈希值的位是否都为 1。

下面通过一个例子理解整个过程。现在我们想向布隆过滤器中插入字符串 `"foo"` 和 `"bar"`，这个布隆过滤器有 8 位宽，并且我们有两个哈希函数：

1. 假设用两个哈希函数分别计算 `"foo"` 的哈希值，得到 `1` 和 `6`
2. 在位数组中置位 `1` 和 `6`
3. 同样用 1 中的两个哈希函数计算 `"bar"` 的哈希值，得到 `6` 和 `3`
4. 在位数组中置位 `6` 和 `3`

这个例子解释了为什么我们不能从布隆过滤器中移除一个元素：插入 `"foo"` 和 `"bar"` 都会导致位数组中的第 `6` 位被置位。

假设我们现在想要查询布隆过滤器中 "quux" 和 "baz" 是否存在：

1. 用和之前相同的两个哈希函数计算 "quux" 的哈希值，得到 4 和 0
2. 检查位数组中的位 4，位 4 没有被置位，所以 "quux" 不可能存在，我们不需要检查位 0
3. 计算 “baz” 的两个哈希值，得到 1 和 3
4. 检查位数组中的位 1 ，位 1 被置位；同样，位 3 也被置位。所以我们认为 "baz" 存在。但是实际上 "bar" 并不存在，这里我们得到了一个误判。

如果你想了解布隆过滤器的一些使用案例，请参阅 [Broder02]

25.2 使用场景与封装设计

不是所有布隆过滤器的使用需求都完全相同。在某些使用场景中，只需要一次性创建布隆过滤器，之后只有查询。对于其他应用，我们可能需要在创建布隆过滤器之后持续更新。我们通过把可变和不可变的 API 放在不同的模块中来对它们实施分离，其中 BloomFilter 用于实现不可变的布隆过滤器，而 BloomFilter.Mutable 则用于实现可变的布隆过滤器。

我们将可变与不可变的 API 分离，通过把他们放在不同的模块中：BloomFilter 用于不可变的代码，BloomFilter.Mutable 用于可变代码。

另外，我们将创建一些辅助模块，这些模块不会在公开的 API 中出现，但它们可以让内部代码变得更清晰。

最后，我们让 API 的使用者提供用来产生多个哈希的函数。这个函数的类型是 a -> [Word32] 。我们将使用这个函数返回的全部哈希值，所以这个函数返回的列表不能为无穷的。

25.3 基本设计

跟前面介绍布隆过滤器实现原理时提到的数据结构一样，我们的 Haskell 版布隆过滤器也会用到一个位数组和一个能够计算出多个哈希值的函数。

```
-- file: BloomFilter/Internal.hs
module BloomFilter.Internal
    (
      Bloom(..)
    , MutBloom(..)
    ) where

import Data.Array.ST (STUArray)
import Data.Array.Unboxed (UArray)
import Data.Word (Word32)
```

(continues on next page)

(continued from previous page)

```
data Bloom a = B {
      blmHash  :: (a -> [Word32])
    , blmArray :: UArray Word32 Bool
    }
```

因为 BloomFilter.Internal 模块纯粹是为了控制名称的可见性而存在的，所以在创建 Cabal 包时，我们将不会导出这个模块。我们把 BloomFilter.Internal 导入可变和不可变的模块中，但是我们会从各个模块中重新导出和模块 API 相关的类型。

25.3.1 拆箱，提升和 bottom

与其他 Haskell 的数组不同，UArray 包含未装箱的值。

对于一个常规的 Haskell 类型来说，它的值既可以是完全求值的（full evaluated），也可以是未求值的形式程序（thunk），又或者特殊值 ，发音为（有时候也写作）“bottom”。值 是一个用来表示计算未成功的占位符。这里的计算可以有多种形式。它可能是一个无限循环，一个 error 应用，或者特殊值 undefined 。

一个可以包含 bottom 的类型被称为已提升的。所有常规 Haskell 类型都是已提升的。实际中，这意味着我们可以写 error "eek!" 或者 undefined 来代替常规表达式。

存储形式程序和 bottom 的能力会带来性能上的损耗：这种能力增加了额外的间接层。为了理解为什么我们需要这种间接，考虑 Word32 类型。这种类型的值是全 32 位宽的，所以在 32 位系统上，没有办法直接用 32 位来编码 bottom。运行时系统不得不维护，并且检查一些额外的数据来跟踪这个值是不是 。

一个未装箱的值没有这种间接性。通过未装箱可以获得性能，但是牺牲了表示形式程序或者 bottom 的能力。因为未装箱的数组可以比常规 Haskell 的数组更加紧凑，所以这对于大量数据和位来说是一个非常好的选择。

GHC 通过将 8 个数组元素组装成 1 个字节，实现了一种 Bool 类型的 UArray 数组，这种数组非常适合我们的需求。

25.4 ST monad

正如前面的修改数组元素 部分所说，因为修改一个不可变数组需要对整个数组进行复制，所以这种修改的代价是非常高的。即使使用 UArray ，这一问题仍然会存在。那么我们如何才能将复制不可变数组的代价降低至我们可以承受的水平呢?

在指令式语言中，我们可以简单地原地修改数组元素，并且在 Haskell 里面也可以这样做。

Haskell 提供了一个特殊的 Monad，叫做 ST[58] （*State Transformer*）。ST 允许我们安全地工作在可变状态下。与 State Monad 相比，ST Monad 有一些额外的强大功能。

[58] ST 是“状态变换器”（state transformer）的缩写。

- 解冻一个不可变数组并得到一个可变数组，接着原地对可变数组进行修改，然后在修改完成之后冻结出一个新的不可变数组。
- 通过 可变引用（*mutable references*）可以构建出一种数据结构，这种数据结构允许用户像命令式语言一样随时对其进行修改。对于那些尚未找到高效纯函数替代的命令式数据结构和算法来说，这个功能尤为重要。

`IO` Monad 同样提供了这些功能。两者的主要区别在于，`ST` Monad 是为了让用户能够从 Monad 中回退到纯 Haskell 代码中而设计的。和大部分 Haskell Monad（当然除了 `IO` ）一样，我们通过执行函数 `runST` 进入 `ST` Monad，然后通过从 `runST` 中 return 来退出。

当我们应用一个 Monad 的执行函数的时候，我们希望它可以反复运行：如果给予相同的函数体（body）和参数，我们每次都能得到相同的结果。这同样可以应用于 `runST` 。为了达到这种可重复性（repeatablility），`ST` Monad 比 `IO` Monad 更加严格。我们不能读写文件，创建全局变量，或者创建线程。甚至，即使我们可以创建并且使用可变的引用和数组，类型系统也不允许它们逃逸到 `runST` 的调用方。在返回数据之前，可变数组必须被冻结（frozen）为不可变数组，并且可变引用不可以逃逸。

25.5 设计一个合格的输入 API

我们需要讨论一下用来处理布隆过滤器的公开接口。

```
-- file: BloomFilter/Mutable.hs
module BloomFilter.Mutable
    (
      MutBloom
    , elem
    , notElem
    , insert
    , length
    , new
    ) where

import Control.Monad (liftM)
import Control.Monad.ST (ST)
import Data.Array.MArray (getBounds, newArray, readArray, writeArray)
import Data.Word (Word32)
import Prelude hiding (elem, length, notElem)

import BloomFilter.Internal (MutBloom(..))
```

在我们导出的函数当中，有几个函数和 Prelude 导出的函数具有相同的名称。这么做是经过考虑的：我们希望用户使用限制名称导入我们的模块，这减轻了用户记忆的负担，因为他们对 Prelude 中的 `elem` ， `notElem` 和 `length` 函数已经相当熟悉了。

在导入这种风格的模块时，我们通常会使用单个字母来作为前缀。例如，用户在代码中使用 `import qualified BloomFilter.Mutable as M` 导入模块，此时用户可以将导入模块中的 `length` 写为 `M.length`，这保持了代码的紧凑型和可读性。

我们也可以不使用限制名称导入模块，但这样一来的话，我们就需要通过 `import Prelude hiding (length)` 来隐藏 Prelude 与模块相冲突的函数。我们不建议使用这种做法，因为它使读者容易忽视代码中的 `length` 并非 Prelude 模块的 `length`。

当然，我们在上面定义的模块头中违背了这个规则：我们导入了 Prelude 并且隐藏了它的一些函数名。这是因为我们在模块中定义了自己的函数 `length`，如果不先隐藏 Prelude 包中的同名函数，编译器将无法确定它该导出我们自定义的 `length` 还是 Prelude 中的 `length`。

虽然导出完全限定名称 `BloomFilter.Mutable.length` 能够消除歧义，但它看起来更丑陋。这个决定对使用模块的用户没有影响，它仅仅针对我们自己——黑盒的设计者，所以这里一般不会导致混淆。

25.6 创建一个可变的布隆过滤器

我们将可变布隆过滤器和不可变的 `Bloom` 类型均声明在 `BloomFilter.Internal` 模块中。

```
-- file: BloomFilter/Internal.hs
data MutBloom s a = MB {
          mutHash :: (a -> [Word32])
        , mutArray :: STUArray s Word32 Bool
        }
```

`STUArray` 类型提供了可以在 `ST` monad 中使用的可变数组，我们可以使用 `newArray` 函数创建一个 `STUArray`。下面的 `new` 函数属于 `BloomFilter.Mutable` 模块（译注：此处应为 `module`，原著中此处为 `function`）。

```
-- file: BloomFilter/Mutable.hs
new :: (a -> [Word32]) -> Word32 -> ST s (MutBloom s a)
new hash numBits = MB hash `liftM` newArray (0,numBits-1) False
```

`STUArray` 的大多数方法实际上是 `MArray` 类型类的实现，这个类型类在 `Data.Array.MArray` 模块中定义。

有两个因素导致我们自己定义的 `length` 函数略显复杂：函数依赖于位数组对自己边界的记录，且 `MArray` 实例的 `getBounds` 函数有一个 monadic 类型。此外最终的结果还需要加 1，因为数组的上限比实际长度小 1。

布隆过滤器在添加元素时，需要将哈希函数计算出的所有位置位。`mod` 函数确保了所有计算出的哈希值都限制在位数组范围之内，并将计算位数组偏移量的代码独立为一个函数。（译注：这里使用 `mod` 函数最好保证散列的范围是取模的倍数，否则使用 `mod` 会使散列结果倾向于某种概率分布。由于布隆过滤器和散列通常基于概率，因此应当避免概率分布过分偏离平均）

```haskell
-- file: BloomFilter/Mutable.hs
insert :: MutBloom s a -> a -> ST s ()
insert filt elt = indices filt elt >>=
                                  mapM_ (\bit -> writeArray (mutArray filt) bit True)

indices :: MutBloom s a -> a -> ST s [Word32]
indices filt elt = do
  modulus <- length filt
  return $ map (`mod` modulus) (mutHash filt elt)
```

判断一个元素是否属于布隆过滤器的成员非常简单：如果根据元素计算出的哈希值对应的每一位都已经被置位，则可以认为这个元素已经位于布隆过滤器中。

```haskell
-- file: BloomFilter/Mutable.hs
elem, notElem :: a -> MutBloom s a -> ST s Bool

elem elt filt = indices filt elt >>=
                                allM (readArray (mutArray filt))

notElem elt filt = not `liftM` elem elt filt
```

我们需要再编写一个简单的支持函数：monadic 版本的 `all` ，这里将其命名为 `allM` 。

```haskell
-- file: BloomFilter/Mutable.hs
allM :: Monad m => (a -> m Bool) -> [a] -> m Bool
allM p (x:xs) = do
  ok <- p x
  if ok
        then allM p xs
        else return False
allM _ [] = return True
```

25.7 不可变的 API

我们为可变布隆过滤器保留的接口与不可变布隆过滤器的 API 拥有相同的结构：

```haskell
-- file: ch26/BloomFilter.hs
module BloomFilter
        (
          Bloom
        , length
        , elem
```

(continues on next page)

(continued from previous page)

```
        , notElem
        , fromList
        ) where

import BloomFilter.Internal
import BloomFilter.Mutable (insert, new)
import Data.Array.ST (runSTUArray)
import Data.Array.IArray ((!), bounds)
import Data.Word (Word32)
import Prelude hiding (elem, length, notElem)

length :: Bloom a -> Int
length = fromIntegral . len

len :: Bloom a -> Word32
len = succ . snd . bounds . blmArray

elem :: a -> Bloom a -> Bool
elt `elem` filt   = all test (blmHash filt elt)
  where test hash = blmArray filt ! (hash `mod` len filt)

notElem :: a -> Bloom a -> Bool
elt `notElem` filt = not (elt `elem` filt)
```

我们还提供了一个易于使用的方法，用户可以通过 `fromList` 函数创建不可变的布隆过滤器。这个函数对用户隐藏了 `ST` monad，因此他们只能看到不可变类型。

```
-- file: ch26/BloomFilter.hs
fromList :: (a -> [Word32])     -- family of hash functions to use
                 -> Word32              -- number of bits in filter
                 -> [a]                 -- values to populate with
                 -> Bloom a
fromList hash numBits values =
        B hash . runSTUArray $
          do mb <- new hash numBits
                 mapM_ (insert mb) values
                 return (mutArray mb)
```

[Forec 译注：上面的代码在 **GHC** 7.x 中无法通过编译，可以作如下修改来通过编译。

```
fromList hash numBits values =
        (B hash . runSTUArray) (new hash numBits >>= \mb -> do
        mapM_ (insert mb) values
        return (mutArray mb))
```

]

fromList 函数的关键在于 runSTUArray 。前面提过，为了从 ST monad 返回一个不可变数组，我们必须冻结一个可变数组，而 runSTUArray 函数将执行和冻结相结合。给定一个返回 STUArray 的动作，runSTUArray 会使用 runST 执行这个动作，之后冻结返回的 STUArray 并将结果作为 UArray 返回。

MArray 类型类同样提供了一个可用的冻结函数，不过 runSTUArray 更方便，也更有效。这是因为冻结必须将底层数据从 STUArray 复制到新的 UArray 以确保对 STUArray 的后续修改不会影响 UArray 。因为类型系统的存在，runSTUArray 可以在创建 UArray 的同时保证 STUArray 不能被访问。因此 runSTUArray 无需复制也可以共享两个数组之间的底层内容。

25.8 创建友好的接口

在创建了布隆过滤器之后，我们就可以直接使用上面提到的不可变布隆过滤器 API 。需要注意的是，fromList 函数还遗留了一些重要的决策没有完成。我们仍然要选择一个合适的哈希函数，并确定布隆过滤器的容量。

```
-- file: BloomFilter/Easy.hs
easyList :: (Hashable a)
                 => Double          -- false positive rate (between 0 and 1)
                 -> [a]             -- values to populate the filter with
                 -> Either String (B.Bloom a)
```

这里有一种更“友好”的方式创建布隆过滤器：这种方式将计算哈希值的任务交给了 Hashable 类型类，并且允许我们将可容忍的错误率作为参数配置布隆过滤器。它还可以根据容错率和输入列表中的元素数量为我们自动选择合适的过滤器大小。

当然，这种方式不是始终可用的。例如，它可能在输入列表的长度过长时失败。但是这种方法的简便性比起我们之前提供的其他接口都要更胜一筹：它使得接口的用户能够对布隆过滤器的整个创建过程进行一系列控制，并将原来彻头彻尾的命令式接口变成了完完全全的声明式接口。

25.8.1 导出更方便的名称

在模块的导出列表中，我们从基本的 BloomFilter 模块中重新导出了一些名称。这允许临时用户只导入 BloomFilter.Easy 模块，并访问他们可能需要的所有类型和功能。

你可能会好奇，同时导入一个被 BloomFilter.Easy 和 BloomFilter 二者均导出的名称会带来什么后果。我们知道，如果不使用 qualified 导入 BloomFilter 并调用 length 函数，GHC 会发出一个有关歧义的错误，因为 Prelude 中也包含一个同名函数。

Haskell 标准的实现要能够分辨出指向同一个“事物”的多个不同名称。例如，BloomFilter 和 BloomFilter.Easy 均导出了 Bloom 类型，如果我们同时导入了这两个模块并使用 Bloom ，GHC 将能够发现这两个模块导出的 Bloom 相同，并且不会报告歧义。

25.8.2 哈希值

一个布隆过滤器的性能取决于快速、高质量的哈希函数，然而编写一个兼具这两种属性的哈希函数非常困难。

幸运的是，一个名为 Bob Jenkins 的开发人员编写了一些具有这些属性的哈希函数，并公开了代码（网址为 http://burtleburtle.net/bob/hash/doobs.html[59]）。这些哈希函数使用 C 语言编写，可以通过 FFI 创建它们的绑定。在该网站上，我们需要的特定源文件名为 `lookup3.c`，在本地创建一个 `cbits` 目录并将这个文件下载到该目录。

还剩下最后一个难题没有解决：我们可能经常需要七个、十个，甚至更多个散列函数，但又不想把这些不同功能的哈希函数混杂到一起。幸运的是，在实际应用中我们多数情况下只需要两个哈希函数，下面很快就会讲到如何实现。Jenkins 的散列库包含两个函数 `hashword2` 和 `hashlittle2`，它们计算两个哈希值。这里有一个 C 语言的头文件，它描述了这两个函数的 API，我们将它保存为 `cbits/lookup3.h`。

```
/* save this file as lookup3.h */

#ifndef _lookup3_h
#define _lookup3_h

#include <stdint.h>
#include <sys/types.h>

/* only accepts uint32_t aligned arrays of uint32_t */
void hashword2(const uint32_t *key,  /* array of uint32_t */
               size_t length,            /* number of uint32_t values */
               uint32_t *pc,             /* in: seed1, out: hash1 */
               uint32_t *pb);            /* in: seed2, out: hash2 */

/* handles arbitrarily aligned arrays of bytes */
void hashlittle2(const void *key,   /* array of bytes */
                 size_t length,     /* number of bytes */
                 uint32_t *pc,      /* in: seed1, out: hash1 */
                 uint32_t *pb);     /* in: seed2, out: hash2 */

#endif /* _lookup3_h */
```

“盐”是在计算哈希值时加入的干扰值。如果我们用某哈希函数求一个值的散列，并分别加入两个不同的盐，那么将会计算出两个不同的结果。因为即使是同一个哈希函数，接收了两个不同的盐值后，计算结果也会相去甚远。

下面的代码是对这两个函数的绑定：

[59] 与流行的非加密哈希函数（如 FNV 和 hashpjw）相比，Jenkins 的哈希函数的混合属性要好得多，因此我们建议避免使用那些非加密哈希函数。

```
-- file: BloomFilter/Hash.hs
{-# LANGUAGE BangPatterns, ForeignFunctionInterface #-}
module BloomFilter.Hash
        (
          Hashable(..)
        , hash
        , doubleHash
        ) where

import Data.Bits ((.&.), shiftR)
import Foreign.Marshal.Array (withArrayLen)
import Control.Monad (foldM)
import Data.Word (Word32, Word64)
import Foreign.C.Types (CSize)
import Foreign.Marshal.Utils (with)
import Foreign.Ptr (Ptr, castPtr, plusPtr)
import Foreign.Storable (Storable, peek, sizeOf)
import qualified Data.ByteString as Strict
import qualified Data.ByteString.Lazy as Lazy
import System.IO.Unsafe (unsafePerformIO)

foreign import ccall unsafe "lookup3.h hashword2" hashWord2
        :: Ptr Word32 -> CSize -> Ptr Word32 -> Ptr Word32 -> IO ()

foreign import ccall unsafe "lookup3.h hashlittle2" hashLittle2
        :: Ptr a -> CSize -> Ptr Word32 -> Ptr Word32 -> IO ()
```

[Forec 译注：上面的代码在 **GHC** 7.6 后无法通过编译，解决方法是将 `import Foreign.C.Types (CSize)` 修改为 `import Foreign.C.Types (CSize(..))` 或者 `import Foreign.C.Types (CSize(CSize))` 。]

函数的定义可以查看我们刚刚创建的 `lookup3.h` 。

出于对效率和便捷的考虑，我们将 Jenkins 散列函数所需的 32 位盐值和计算出的散列值组成单个 64 位值：

```
-- file: BloomFilter/Hash.hs
hashIO :: Ptr a    -- value to hash
          -> CSize    -- number of bytes
          -> Word64   -- salt
          -> IO Word64
hashIO ptr bytes salt =
        with (fromIntegral salt) $ \sp -> do
          let p1 = castPtr sp
                p2 = castPtr sp `plusPtr` 4
          go p1 p2
```

(continues on next page)

(continued from previous page)

```
        peek sp
  where go p1 p2
                | bytes .&. 3 == 0 = hashWord2 (castPtr ptr) words p1 p2
                | otherwise        = hashLittle2 ptr bytes p1 p2
              words = bytes `div` 4
```

[Forec 译注：with 在下面的段落中会有解释，castPtr 没有介绍过，你可以在 http://hackage.haskell.org/package/base-4.6.0.1/docs/Foreign-Marshal-Utils.html#v:with 查看 with 的文档，在 http://hackage.haskell.org/package/base-4.6.0.1/docs/Foreign-Ptr.html#v:castPtr 查看 castPtr 的文档。此外，这里使用 castPtr 并对 p1 和 p2 使用类型推断虽然简短了代码，但也降低了代码的可读性。]

上面的代码如果没有明确的类型来描述其功能，那么可能看起来就不是很清晰。with 函数在 C 程序的堆栈段中为盐值分配了空间，并存储了当前的盐值，所以 sp 的类型是 Ptr Word64 。指针 p1 和 p2 的类型是 Ptr Word32 ； p1 指向了 sp 的低位字，p2 指向了 sp 的高位字。这就是我们将一个 Word64 的盐值切分为两个 Ptr Word32 参数的方法。

因为所有的数据指针均来自 Haskell 堆，所以它们会在一个能够安全传递给 hashWord2 （只接受 32 位对齐地址）或者 hashLittle2 的地址上对齐。由于 hashWord2 是两个哈希函数中较快的，所以我们会在数据为 4 字节的倍数时调用 hashWord2 ，否则调用 hashLittle2 。[Forec 译注：这里原著拼写错误，将 hashWord2 误拼写为 hashWord32]

C 语言编写的哈希函数会将计算出的哈希值写入 p1 和 p2 指向的地址，我们可以通过 sp 直接检索计算结果。

使用这个模块的客户不应当被低级细节困扰，所以我们通过类型类来提供一个干净、高级的接口：

```
-- file: BloomFilter/Hash.hs
class Hashable a where
        hashSalt :: Word64        -- ^ salt
                 -> a             -- ^ value to hash
                 -> Word64

hash :: Hashable a => a -> Word64
hash = hashSalt 0x106fc397cf62f64d3
```

我们还为这个类型类提供了一些实用的实现。要计算基本类型的哈希值，必须先编写一点样板代码：

```
-- file: BloomFilter/Hash.hs
hashStorable :: Storable a => Word64 -> a -> Word64
hashStorable salt k = unsafePerformIO . with k $ \ptr ->
                                        hashIO ptr (fromIntegral (sizeOf k)) salt

instance Hashable Char   where hashSalt = hashStorable
instance Hashable Int    where hashSalt = hashStorable
instance Hashable Double where hashSalt = hashStorable
```

下面的代码使用 Storable 类型类将声明减少到一个：

```
-- file: BloomFilter/Hash.hs
instance Storable a => Hashable a where
        hashSalt = hashStorable
```

[Forec 译注：上面使用 Storable 的代码需要添加 {-# LANGUAGE FlexibleInstances #-} 和 {-# LANGUAGE UndecidableInstances #-} 两个编译选项后才能通过编译。]

不幸的是，Haskell 不允许编写这种形式的实例，因为它们会使类型系统无法判定：编译器的类型检查器可能会陷入无限循环中。对不可确定类型的限制使我们必须单独列出声明，但它对于上面的定义并不会造成什么影响。[Forec 译注：上面的例子中如果存在 instance Hashable a => Storable a 这样的代码（虽然这样的代码没什么意义），则编译器会陷入循环。但如果程序开发者能够保证这种情况不会发生，则可以开启编译选项并使用这一扩展功能。]

```
-- file: BloomFilter/Hash.hs
hashList :: (Storable a) => Word64 -> [a] -> IO Word64
hashList salt xs =
        withArrayLen xs $ \len ptr ->
          hashIO ptr (fromIntegral (len * sizeOf x)) salt
  where x = head xs

instance (Storable a) => Hashable [a] where
        hashSalt salt xs = unsafePerformIO $ hashList salt xs
```

编译器会接受这个实例，因而我们能够对多种列表类型计算哈希值[60]。最重要的是，由于 Char 是 Storable 的一个实例，所以 String 类型的哈希值同样可以被计算。

利用函数组合可以计算元组的哈希值：在组合管道的一端取盐，并将元组中每个元素的散列结果作为计算该元组中下一个元素使用的盐值。

```
-- file: BloomFilter/Hash.hs
hash2 :: (Hashable a) => a -> Word64 -> Word64
hash2 k salt = hashSalt salt k

instance (Hashable a, Hashable b) => Hashable (a,b) where
        hashSalt salt (a,b) = hash2 b . hash2 a $ salt

instance (Hashable a, Hashable b, Hashable c) => Hashable (a,b,c) where
        hashSalt salt (a,b,c) = hash2 c . hash2 b . hash2 a $ salt
```

要计算 ByteString 类型的哈希值，我们可以编写一个直接插入到 ByteString 类型内部的特殊实例，其效率非常出色：

[60] 遗憾的是，详细讨论这些情况能否判断不属于本书范畴。

```
-- file: BloomFilter/Hash.hs
hashByteString :: Word64 -> Strict.ByteString -> IO Word64
hashByteString salt bs = Strict.useAsCStringLen bs $ \(ptr, len) ->
                                                  hashIO ptr (fromIntegral len) salt

instance Hashable Strict.ByteString where
        hashSalt salt bs = unsafePerformIO $ hashByteString salt bs

rechunk :: Lazy.ByteString -> [Strict.ByteString]
rechunk s
        | Lazy.null s = []
        | otherwise   = let (pre,suf) = Lazy.splitAt chunkSize s
                                        in  repack pre : rechunk suf
        where repack    = Strict.concat . Lazy.toChunks
                  chunkSize = 64 * 1024

instance Hashable Lazy.ByteString where
        hashSalt salt bs = unsafePerformIO $
                                             foldM hashByteString salt (rechunk bs)
```

由于惰性的 ByteString 类型是由一系列块表示的，我们必须留意块之间的边界。举个例子，字符串 foobar 可以通过五种不同方式表示，如 ["foob", "ar"] 或者 ["fo", "obar"] 。这一点对于多数用户不可见，但我们直接使用了底层的块。rechunck 函数能够确保传递给 C 语言代码的块大小统一为 64 KB，所以无论原始边界在哪里，计算出的哈希值都是一致的。

25.8.3 将两个哈希值转换为多个

正如前面所述，我们需要两个以上的哈希函数才能有效地使用布隆过滤器。双重哈希技术能够组合 Jenkins 哈希函数计算出的两个值，并产生更多的哈希值。使用双重哈希技术产生的多个哈希值足够满足我们的需要，并且比计算多个不同的哈希值更容易。

```
-- file: BloomFilter/Hash.hs
doubleHash :: Hashable a => Int -> a -> [Word32]
doubleHash numHashes value = [h1 + h2 * i | i <- [0..num]]
        where h   = hashSalt 0x9150a946c4a8966e value
                  h1  = fromIntegral (h `shiftR` 32) .&. maxBound
                  h2  = fromIntegral h
                  num = fromIntegral numHashes
```

[Forec 译注：上面代码中的 maxBound 可以通过在 GHCI 中执行 maxBound::Word32 查看，结果为 4294967295。]

25.8.4 实现简单的创建函数

在 BloomFilter.Easy 模块中，我们使用新的 doubleHash 函数来定义之前已经定义过类型的 easyList 函数。

```
-- file: BloomFilter/Easy.hs
module BloomFilter.Easy
        (
          suggestSizing
        , sizings
        , easyList

        -- re-export useful names from BloomFilter
        , B.Bloom
        , B.length
        , B.elem
        , B.notElem
        ) where

import BloomFilter.Hash (Hashable, doubleHash)
import Data.List (genericLength)
import Data.Maybe (catMaybes)
import Data.Word (Word32)
import qualified BloomFilter as B

easyList errRate values =
        case suggestSizing (genericLength values) errRate of
          Left err            -> Left err
          Right (bits,hashes) -> Right filt
                where filt = B.fromList (doubleHash hashes) bits values
```

上面的代码依赖于一个 suggestSizing 函数，这个函数能够根据用户要求的错误率和期望滤波器包含元素的最大数量来估计滤波器的大小以及要计算的哈希值数量：

```
-- file: BloomFilter/Easy.hs
suggestSizing
        :: Integer       -- expected maximum capacity
        -> Double        -- desired false positive rate
        -> Either String (Word32,Int) -- (filter size, number of hashes)
suggestSizing capacity errRate
        | capacity <= 0                = Left "capacity too small"
        | errRate <= 0 || errRate >= 1 = Left "invalid error rate"
        | null saneSizes               = Left "capacity too large"
        | otherwise                    = Right (minimum saneSizes)
```

(continues on next page)

(continued from previous page)

```
  where saneSizes = catMaybes . map sanitize $ sizings capacity errRate
                sanitize (bits,hashes)
                  | bits > maxWord32 - 1 = Nothing
                  | otherwise            = Just (ceiling bits, truncate hashes)
                  where maxWord32 = fromIntegral (maxBound :: Word32)

sizings :: Integer -> Double -> [(Double, Double)]
sizings capacity errRate =
        [(((-k) * cap / log (1 - (errRate ** (1 / k)))), k) | k <- [1..50]]
  where cap = fromIntegral capacity
```

[Forec 译注：关于上面代码中 errRate 的推导，可以参考维基百科上布隆过滤器的词条 http://en.wikipedia.org/wiki/Bloom_filter 。根据维基百科，有式 errRate = (1-e^(-k*cap/size))^k ，因为 suggestSizing 函数接受 k 、cap 和 errRate ，我们可以重新整理方程，并得到 size = -k*cap/log(1 - errRate^(1/k)) ，这就是代码中使用的公式。]

我们对参数做了一定的规范。例如，sizings 函数虽然受到数组大小和哈希值数量的影响，但它并不验证这两个值。由于使用了 32 位哈希值，我们必须过滤掉太大的数组。

在 suggestSizing 函数中，我们仅仅尝试最小化位数组的大小，而不考虑哈希值的数量。现在让我们通过 GHCI 交互地探索一下数组大小和哈希值数量的关系，并解释这种做法的缘由：

假设要将一千万个元素插入布隆过滤器中，并希望误报率不超过 0.1 %。

```
ghci> let kbytes (bits,hashes) = (ceiling bits `div` 8192, hashes)
ghci> :m +BloomFilter.Easy Data.List
Could not find module `BloomFilter.Easy':
  Use -v to see a list of the files searched for.
ghci> mapM_ (print . kbytes) . take 10 . sort $ sizings 10000000 0.001

(17550,10.0)
(17601,11.0)
(17608,9.0)
(17727,12.0)
(17831,8.0)
(17905,13.0)
(18122,14.0)
(18320,7.0)
(18368,15.0)
(18635,16.0)
```

[Forec 译注：上面交互式代码在原著中是有误的，原著没有纠正这一错误，上面的结果由译者修改后计算。要想得到上面的结果，可以参考如下步骤：

```
$ cd cbits
$ gcc -c -fPIC lookup3.c -o lookup3.o
$ gcc -shared -Wl,-soname,liblookup3.so.1 -o liblookup3.so.1.0.1 lookup3.o
$ ln -s liblookup3.so.1.0.1 liblookup3.so
$ cd ..
$ ghci -L./cbits -llookup3
Prelude> :l BloomFilter.Easy
*BloomFilter.Easy> :m +Data.List
*BloomFilter.Easy Data.List> let kb (bits,hashes) = (ceiling bits `div` 8192, hashes)
*BloomFilter.Easy Data.List> mapM_ (print . kb) . take 10 . sort $ sizings 10000000 0.
↪001
Loading package array-0.4.0.0 ... linking ... done.
Loading package bytestring-0.9.2.1 ... linking ... done.
(17550,10.0)
(17601,11.0)
(17608,9.0)
(17727,12.0)
(17831,8.0)
(17905,13.0)
(18122,14.0)
(18320,7.0)
(18368,15.0)
(18635,16.0)
```

]

通过计算 10 个哈希值，我们得到了一个非常紧凑的表（刚好超过 17 KB）。如果真的对数据进行反复的散列，则哈希值的数量可以减少到 7 个，空间消耗可以减少到 5%。因为 Jenkins 的哈希函数在一轮计算中得到两个哈希值，并通过双重哈希产生额外的哈希值，因此我们计算额外哈希值的成本非常小，所以选择最小的表大小。

如果将最高可容忍误报率增加十倍，变为 1%，则所需的空间和哈希值数量都会下降，尽管下降的幅度不太容易预测。

```
ghci> mapM_ (print . kbytes) . take 10 . sort $ sizings 10000000 0.01
(11710,7.0)
(11739,6.0)
(11818,8.0)
(12006,9.0)
(12022,5.0)
(12245,10.0)
(12517,11.0)
(12810,12.0)
(12845,4.0)
(13118,13.0)
```

[Forec 译注：上面的代码在原著中同样有误，计算结果由译者修改后给出，步骤同上。]

25.9 创建一个 Cabal 包

至此我们已经创建了一个不算太复杂的库，它包括四个公共模块和一个内部模块。现在创建一个 `rwh-bloomfilter.cabal` 文件，将这个库打包成容易发布的格式。

Cabal 允许我们在一个包中描述几个库的信息。`.cabal` 文件的头部包含了所有库通用的信息，后面跟着各个库不同的部分。

```
Name:                rwh-bloomfilter
Version:             0.1
License:             BSD3
License-File:        License.txt
Category:            Data
Stability:           experimental
Build-Type:          Simple
```

由于 C 语言代码 `lookup3.c` 和库捆绑在一起，所以我们要将这个 C 语言源文件的信息告知 Cabal。

```
Extra-Source-Files: cbits/lookup3.c cbits/lookup3.h
```

`Extra-Source-Files` 指令对包的构建没有影响：它仅仅在我们运行 `runhaskell Setup sdist` 时指导 Cabal 绑定一些额外的文件，这条指令将创建一个用于发布的源码包。

25.9.1 处理不同的构建设置

在 2007 年以前，Haskell 标准库被组织在少数几个规模较大的包中，其中最大的一个被命名为 `base` 。这个包将许多互不相关的库绑定到一起，因此 Haskell 社区将 `base` 包拆分成了几个模块化程度更高的库。

Cabal 包需要指明自己构建时依赖的其它包，这些信息帮助 Cabal 的命令行接口在必要的情况下自动下载并构建包的依赖。我们希望，不管用户使用的 GHC 版本是否具备 `base` 和其它包的现代布局，我们的代码都能尽量兼容。举个例子，我们的代码要能够在 `array` 包存在的时候说明自己依赖它，否则就只能依赖 `base` 包。

Cabal 提供了一个通用的配置功能，它允许我们选择性地启用一个 `.cabal` 文件的某些部分。构建的配置信息由布尔类型的标识控制，标识为 `True` 时使用 `if flag` 之后的文本，而标识为 `False` 时则使用跟在 `else` 之后的文本。

```
Cabal-Version:       >= 1.2

Flag split-base
  Description: Has the base package been split up?
```

(continues on next page)

(continued from previous page)

```
  Default: True

Flag bytestring-in-base
  Description: Is ByteString in the base or bytestring package?
  Default: False
```

- 配置功能在 Cabal 的 1.2 版本中引入，因此指定 Cabal 版本不能低于 1.2。
- `split-base` 标识的含义不言而喻。[Forec 译注：该标识表示 `base` 包是否被划分]
- `bytestring-in-base` 标识源于一段更为曲折的历史：`bytestring` 包在创建之初是和 GHC 6.4 捆绑的，并且它始终独立于 `base` 包；在 GHC 6.6 中，它被合并到了 `base` 包中；到了 GHC 6.8.1 版本，它又再次被独立出去。
- 上面这些标识对构建包的开发者来说通常是不可见的，因为 Cabal 会自动处理它们。在我们进行下一步分析前，了解它们能够帮助理解 `.cabal` 文件中 `Library` 部分开头的内容。

```
Library
  if flag(bytestring-in-base)
        -- bytestring was in base-2.0 and 2.1.1
        Build-Depends: base >= 2.0 && < 2.2
  else
        -- in base 1.0 and 3.0, bytestring is a separate package
        Build-Depends: base < 2.0 || >= 3, bytestring >= 0.9

  if flag(split-base)
        Build-Depends: base >= 3.0, array
  else
        Build-Depends: base < 3.0
```

Cabal 使用标识的默认值来创建包描述（该标识的默认值为 `True` ）。如果当前的配置能够构建成功（比如所有需要的包版本都可用）则这个配置将被采用，否则 Cabal 将尝试多种方式组合标识，直到它寻找到一个能够构建成功的配置，又或者所有备选的配置都无法生效为止。

例如，如果我们将 `split-base` 和 `bytestring-in-base` 设置为 `True`，Cabal 会选择以下的包依赖项：

```
Build-Depends: base >= 2.0 && < 2.2
Build-Depends: base >= 3.0, array
```

`base` 包的版本无法同时又高于 3.0 又低于 2.2，所以 Cabal 出于一致性考虑会拒绝这个配置。对于现代版本的 GHC，在几次尝试后，它将产生如下配置：

```
-- in base 1.0 and 3.0, bytestring is a separate package
Build-Depends: base < 2.0 || >= 3, bytestring >= 0.9
Build-Depends: base >= 3.0, array
```

在运行 `runhaskell Setup configure` 时，我们可以使用 `--flag` 选项手动指定各标识的值，虽然实际中很少需要这么做。

25.9.2 编译选项和针对 C 的接口

下面让我们继续分析 `.cabal` 文件，并完成与 Haskell 相关的剩余细节。如果在构建过程中启用分析，我们希望所有的顶级函数都显示在分析的输出中。

```
GHC-Prof-Options: -auto-all
```

`Other-Modules` 属性列出了库中私有的 Haskell 模块，这些模块对使用此包的代码不可见。

在 GHC 构建这个包时，Cabal 会将 `GHC-Options` 属性中的选项传递给编译器。

`-O2` 选项使 GHC 尽可能地优化我们的代码。不加以优化编译出的代码效率很低，所以在编译生产代码时应当始终使用 `-O2` 选项。

为了写出更清晰的代码，我们通常添加 `-Wall` 选项，这个选项会启用 GHC 的所有警告。这将导致 GHC 在遇到潜在问题（例如重叠的模式匹配、未使用的函数参数等其它潜在障碍）时提出警告。尽管忽略这些警告一般是安全的，但我们应该尽量完善代码以消除它们。这一点小小的努力，将催生更容易阅读和维护的代码。

普通情况下 GHC 会直接生成汇编语言代码，而在使用 `-fvia-C` 编译时，GHC 会生成 C 语言代码并使用系统的 C 编译器来编译它。这会减慢编译速度，但有时 C 编译器能够进一步改善 GHC 优化的代码，所以这也是值得的。

我们这里提到 `-fiva-C` 主要是为了展示如何使用它编译。

```
C-Sources:         cbits/lookup3.c
CC-Options:        -O3
Include-Dirs:      cbits
Includes:          lookup3.h
Install-Includes:  lookup3.h
```

对于 `C-Sources` 属性，我们只需要列出必须编译到库中的文件。`CC-Options` 属性包含了提供给 C 编译器的选项（其中选项 `-O3` 用于指定最高级别的优化）。因为对 Jenkins 散列函数的 FFI 绑定引用了 `lookup3.h` 头文件，我们需要告诉 Cabal 在哪里可以找到该头文件。`Install-Includes` 用来告诉 Cabal 安装这个头文件，否则在构建时客户端代码将无法找到头文件。

[Forec 译注：遗憾的是，在较新版本的 GHC 中 `-fvia-C` 不会产生任何作用，并且它将在未来的 GHC 发布中被移除。所以本节关于 `-fvia-C` 选项的介绍已经成为历史了。]

25.10 用 QuickCheck 测试

在进一步考虑性能之前，我们要确保布隆过滤器的正确性。使用 QuickCheck 可以轻松测试一些基本的属性。

```haskell
-- file: examples/BloomCheck.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Main where

import BloomFilter.Hash (Hashable)
import Data.Word (Word8, Word32)
import System.Random (Random(..), RandomGen)
import Test.QuickCheck
import qualified BloomFilter.Easy as B
import qualified Data.ByteString as Strict
import qualified Data.ByteString.Lazy as Lazy
```

普通的 quickCheck 函数对布隆过滤器属性的测试帮助不大，因为它产生的 100 个测试输入样例无法完整覆盖布隆过滤器的功能。这里我们编写自己的测试函数：

```haskell
-- file: examples/BloomCheck.hs
handyCheck :: Testable a => Int -> a -> IO ()
handyCheck limit = check defaultConfig {
                                        configMaxTest = limit
                                      , configEvery   = \_ _ -> ""
                                      }
```

[Forec 译注：在较新版本的 QuickCheck 中，上面的代码应该写成：

```haskell
handyCheck :: Testable a => Int -> a -> IO ()
handyCheck limit = quickCheckWith (stdArgs { maxSuccess = limit })
```

]

下面我们要完成的第一个任务是确保：无论用户选择多大的容错率，只要向布隆过滤器添加了一个任意值，则之后针对该值的成员测试都应得到“值已存在”的结果。

我们将使用 easyList 函数来创建一个布隆过滤器。Double 的 Random 实例能够生成 0 到 1 之间的随机数，因此 QuickCheck 可以提供任意大小的错误率。

然而，测试生成的错误率应当排除 0 和 1。QuickCheck 提供了两种方法：

- 通过结构：指定要生成的有效值的范围。QuickCheck 为此提供了 forAll 组合器。
- 通过过滤：当 QuickCheck 生成一个任意值时，用 (=~>) 运算符过滤掉不符合标准的值。如果布隆过滤器通过这种方式拒绝一个输入值，测试将显示成功。

如果以上两个方法都可以选择，那么最好采用通过结构的方法：假设 QuickCheck 生成了 1000 个任意值，其中 800 个由于某些原因被过滤掉。看起来我们似乎运行了 1000 次测试，但实际上只有 200 次做了有意义的事。

出于这个原因，当需要产生错误率时，我们不会去消除 QuickCheck 提供的 0 或 1，而是在一个始终有效的区间中构造值：

```
falsePositive :: Gen Double
falsePositive = choose (epsilon, 1 - epsilon)
        where epsilon = 1e-6

(=~>) :: Either a b -> (b -> Bool) -> Bool
k =~> f = either (const True) f k

prop_one_present _ elt =
        forAll falsePositive $ \errRate ->
          B.easyList errRate [elt] =~> \filt ->
                elt `B.elem` filt
```

[Forec 译注：原著作者似乎在这里犯了一点错误，根据代码，`prop_one_present` 的型别声明应为 `(Hashable a) => t -> a -> Property`，但这无法通过编译，因为 `prop_one_present` 的第一个参数 `_` 隐藏着对类型 `t` 和 `a` 的约束，它们二者必须相等。有两种解决方法：一是不指定这个多余的 `_` 参数，二是将型别声明显式地指定为 `(Hashable a) => a -> a -> Property`。]

组合器 `(=~>)` 过滤了 `easyList` 失败的情况：如果失败了，测试会自动通过。

25.10.1 多态测试

[Forec 译注：以下几节，原著作者给出的代码在新版本 GHC 中无法通过编译，如需在 GHC 中运行，请按照译注中的说明修改对应文件。译文仅对代码和部分运行结果进行一定修正。]

QuickCheck 要求属性必须是单型的。鉴于目前有多种可散列类型需要测试，我们有必要设计一个方法，避免对每种类型都编写同样的测试。

注意，`prop_one_present` 这个函数是多态的，但它忽略了第一个参数。我们可以借助这一点模拟单型性质：

```
ghci> :load BloomCheck
BloomCheck.hs:9:17:
        Could not find module `BloomFilter.Easy':
          Use -v to see a list of the files searched for.
Failed, modules loaded: none.
ghci> :t prop_one_present
<interactive>:1:0: Not in scope: `prop_one_present'
ghci> :t prop_one_present (undefined :: Int)
<interactive>:1:0: Not in scope: `prop_one_present'
```

[Forec 译注：原著给出的代码无法正确运行，要想正确运行需要对 `BloomCheck.hs` 做如下修改：

- 按上面译注里所述，将 handyCheck limit 部分修改；
- 移除 Random Word8 、Arbitary Word8 、Random Word32 以及 Arbitary Word32 的实例；
- 删除 Arbitary 的 Lazy.ByteString 实例中 coarbitrary 所属行；
- 删除 Arbitary 的 Strict.ByteString 实例中 coarbitrary 所属行；

]

任何值都可以作为 prop_one_present 的第一个参数。第二个参数中第一个元素的类型需要和第一个参数保持一致。

```
ghci> handyCheck 5000 $ prop_one_present (undefined :: Int)
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_one_present'
ghci> handyCheck 5000 $ prop_one_present (undefined :: Double)
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_one_present'
```

[Forec 译注：请留意本节译注中对 prop_one_present 的修正，我们需要显式声明 prop_one_present 的类型，或者移除 prop_one_present 的 _ 参数。这里采取第二种方法，定义函数 prop_one_present' ，它与 prop_one_present 的唯一区别在于它没有匿名参数：

```
prop_one_present' elt =
        forAll falsePositive $ \errRate ->
        B.easyList errRate [elt] =~> \filt ->
        elt `B.elem` filt
```

载入修改后的文件，并显式指定 prop_one_present' 的型别：

```
ghci> handyCheck 2 $ (prop_one_present' :: String -> Property)
Passed:
""
7.053216229843191e-2
Passed:
"\n9UP'\161O%~S"
0.5021342445896073
```

本节剩余部分在 GHCI 中执行 handyCheck 指令均需做类似的修正。下面的译文不再对原著的运行结果予以更正。]

在向布隆过滤器添加多个元素之后，这些元素应该都能够被识别出来：

```
-- file: examples/BloomCheck.hs
prop_all_present _ xs =
        forAll falsePositive $ \errRate ->
```

(continues on next page)

(continued from previous page)

```
          B.easyList errRate xs =~> \filt ->
                all (`B.elem` filt) xs
```

测试依然成功：

```
ghci> handyCheck 2000 $ prop_all_present (undefined :: Int)
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_all_present
```

25.10.2 为 ByteString 编写任意实例

QuickCheck 库没有为 ByteString 类型提供 Arbitary 的实例，因此我们必须自己编写。pack 函数可基于 [Word8] 创建一个 ByteString 。

```
-- file: examples/BloomCheck.hs
instance Arbitrary Lazy.ByteString where
        arbitrary = Lazy.pack `fmap` arbitrary
        coarbitrary = coarbitrary . Lazy.unpack

instance Arbitrary Strict.ByteString where
        arbitrary = Strict.pack `fmap` arbitrary
        coarbitrary = coarbitrary . Strict.unpack
```

[Forec 译注：原著编写时的 Arbitary 类型类到今天已经发生了变化，coarbitrary 函数现在属于 CoArbitrary 类型类。上面的代码需要修正为：

```
instance Arbitrary Lazy.ByteString where
arbitrary = Lazy.pack `fmap` arbitrary

instance CoArbitrary Lazy.ByteString where
coarbitrary = coarbitrary . Lazy.unpack
```

Strict.ByteString 也要做同样的修改。]

QuickCheck 中还缺少针对 Data.Word 和 Data.Int 中固定宽度类型的 Arbitary 实例。我们至少需要为 Word8 实现 Arbitary 实例：

```
-- file: examples/BloomCheck.hs
instance Random Word8 where
  randomR = integralRandomR
  random = randomR (minBound, maxBound)
```

(continues on next page)

(continued from previous page)

```
instance Arbitrary Word8 where
        arbitrary = choose (minBound, maxBound)
        coarbitrary = integralCoarbitrary
```

[Forec 译注：Word8 的实例不需要定义。当前的 QuickCheck 库已经默认实现了这些实例。]

为这些实例编写几个通用函数，以便在之后为其他整型类型编写实例时重用它们：

```
-- file: examples/BloomCheck.hs
integralCoarbitrary n =
        variant $ if m >= 0 then 2*m else 2*(-m) + 1
  where m = fromIntegral n

integralRandomR (a,b) g = case randomR (c,d) g of
                                                    (x,h) -> (fromIntegral x, h)
        where (c,d) = (fromIntegral a :: Integer,
                                  fromIntegral b :: Integer)

instance Random Word32 where
  randomR = integralRandomR
  random = randomR (minBound, maxBound)

instance Arbitrary Word32 where
        arbitrary = choose (minBound, maxBound)
        coarbitrary = integralCoarbitrary
```

[Forec 译注：上面这部分代码也是不需要的，QuickCheck 库已经实现了它们。]

创建了这些 Arbitary 实例后，我们就可以在 ByteString 类型上尝试现有的属性：

```
ghci> handyCheck 1000 $ prop_one_present (undefined :: Lazy.ByteString)
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_one_present'
<interactive>:1:49:
        Failed to load interface for `Lazy':
          Use -v to see a list of the files searched for.
ghci> handyCheck 1000 $ prop_all_present (undefined :: Strict.ByteString)
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_all_present'
<interactive>:1:49:
        Failed to load interface for `Strict':
          Use -v to see a list of the files searched for.
```

25.10.3 推荐大小是正确的吗？

随着待运行测试数量的增加，用于测试 easyList 性能的开销也在快速增长。我们希望输入数据规模对 easyList 的性能没有影响。直接测试是不现实的，所以这里使用另一个问题来衡量：面对极端的输入规模时，suggestSizing 是否仍能给出敏感的数组大小以及哈希值？

检查这一特性略微有些棘手：我们需要同时改变期望的错误率和预期容量。根据 sizings 函数给出的结果，这些值之间的关系较难预测。

我们可以尝试忽略复杂性：

```
-- file: examples/BloomCheck.hs
prop_suggest_try1 =
  forAll falsePositive $ \errRate ->
        forAll (choose (1,maxBound :: Word32)) $ \cap ->
          case B.suggestSizing (fromIntegral cap) errRate of
                Left err -> False
                Right (bits,hashes) -> bits > 0 && bits < maxBound && hashes > 0
```

正如我们所料，这一做法只会带来一个没有实际作用的测试：

```
ghci> handyCheck 1000 $ prop_suggest_try1
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_suggest_try1'
ghci> handyCheck 1000 $ prop_suggest_try1
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_suggest_try1'
```

[Forec 译注：一个仅供参考的运行结果如下：

```
ghci> handyCheck 1000 $ prop_suggest_try1
Passed:
0.840272094122386
1533634864
Failed:
0.13172750223946617
3002287708
*** Failed! Falsifiable (after 2 tests):
0.13172750223946617
3002287708
```

]

将 QuickCheck 打印的反例交给 suggestSizing 时，我们发现这些输入被拒绝了，因为它们会导致一个过于庞大的位数组。

```
ghci> B.suggestSizing 1678125842 8.501133057303545e-3
<interactive>:1:0:
        Failed to load interface for `B':
          Use -v to see a list of the files searched for.
```

[Forec 译注：运行结果如下：

```
ghci> B.suggestSizing 1678125842 8.501133057303545e-3
Left "capacity too large"
```

]

由于无法预测哪些组合会导致此问题，我们只能通过限制大小和错误率来防止异常：

```
-- file: examples/BloomCheck.hs
prop_suggest_try2 =
        forAll falsePositive $ \errRate ->
          forAll (choose (1,fromIntegral maxWord32)) $ \cap ->
                let bestSize = fst . minimum $ B.sizings cap errRate
                in bestSize < fromIntegral maxWord32 ==>
                   either (const False) sane $ B.suggestSizing cap errRate
  where sane (bits,hashes) = bits > 0 && bits < maxBound && hashes > 0
                maxWord32 = maxBound :: Word32
```

对其加以测试，看起来效果不错：

```
ghci> handyCheck 1000 $ prop_suggest_try2
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:18: Not in scope: `prop_suggest_try2'
```

[Forec 译注：预期的运行结果为：

```
ghci> handyCheck 1000 $ prop_suggest_try2
+++ OK, passed 1000 tests.
```

]

在过大的测试中，许多组合都被过滤掉了：

```
ghci> handyCheck 10000 $ prop_suggest_try2
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:19: Not in scope: `prop_suggest_try2'
```

为了解决此问题，我们要尝试降低生成无效输入的可能性：

```
-- file: examples/BloomCheck.hs
prop_suggestions_sane =
        forAll falsePositive $ \errRate ->
          forAll (choose (1,fromIntegral maxWord32 `div` 8)) $ \cap ->
                let size = fst . minimum $ B.sizings cap errRate
                in size < fromIntegral maxWord32 ==>
                   either (const False) sane $ B.suggestSizing cap errRate
  where sane (bits,hashes) = bits > 0 && bits < maxBound && hashes > 0
                maxWord32 = maxBound :: Word32
```

最终，我们得到了更加健壮的性能：

```
ghci> handyCheck 40000 $ prop_suggestions_sane
<interactive>:1:0: Not in scope: `handyCheck'
<interactive>:1:19: Not in scope: `prop_suggestions_sane'
```

25.11 性能分析和调优

我们可以将程序通过 QuickCheck 测试视为一条证明代码正确的“基准线”。在调整性能时，随时重新运行测试能够防止修改过程中不小心导致的破坏。

第一步，编写一个用于计时的小程序：

```
-- file: examples/WordTest.hs
module Main where

import Control.Parallel.Strategies (NFData(..))
import Control.Monad (forM_, mapM_)
import qualified BloomFilter.Easy as B
import qualified Data.ByteString.Char8 as BS
import Data.Time.Clock (diffUTCTime, getCurrentTime)
import System.Environment (getArgs)
import System.Exit (exitFailure)

timed :: (NFData a) => String -> IO a -> IO a
timed desc act = do
        start <- getCurrentTime
        ret <- act
        end <- rnf ret `seq` getCurrentTime
        putStrLn $ show (diffUTCTime end start) ++ " to " ++ desc
        return ret

instance NFData BS.ByteString where
```

(continues on next page)

(continued from previous page)

```
        rnf _ = ()

instance NFData (B.Bloom a) where
        rnf filt = B.length filt `seq` ()
```

[Forec 译注：编译过时的代码总是不尽人意，需要做如下修改：

- NFData 已经被移动到 Control.DeepSeq ，因此需将首行的 import 修改为 import Control.DeepSeq (NFData(..)) ；
- 将 rnf 替换为 rdeepsep ，具体原因可参考 http://hackage.haskell.org/package/parallel-2.2.0.1/docs/Control-Parallel-Strategies.html ；
- 在 GHC 7.6 之后，所有的 ByteString 均已成为 NFData 的实例，所以上面针对 BS.ByteString 的实例定义需要被移除。

]

上面的小程序中，我们使用了 24 章中 “将求值从算法中分离” 一节介绍的 rnf 函数来实现简单的时间约束。输出用时这一动作能够确保计算完成，从而准确地评估运算成本。

[Note: 24 章该节尚未翻译，翻译完成后应当将此处替换为 reference。]

主程序根据指定文件创建一个布隆过滤器，它将文件中每一行视为一个要添加到布隆过滤器中的元素：

```
-- file: examples/WordTest.hs
main = do
  args <- getArgs
  let files | null args = ["/usr/share/dict/words"]
                        | otherwise = args
  forM_ files $ \file -> do

        words <- timed "read words" $
          BS.lines `fmap` BS.readFile file

        let len = length words
                errRate = 0.01

        putStrLn $ show len ++ " words"
        putStrLn $ "suggested sizings: " ++
                           show (B.suggestSizing (fromIntegral len) errRate)

        filt <- timed "construct filter" $
          case B.easyList errRate words of
                Left errmsg -> do
                  putStrLn $ "Error: " ++ errmsg
```

(continues on next page)

(continued from previous page)

```
                exitFailure
              Right filt -> return filt

        timed "query every element" $
          mapM_ print $ filter (not . (`B.elem` filt)) words
```

[Forec 译注：显然，原著给出的程序需要在类 Linux 环境下运行，并且 /usr/share/dict 路径下要存在 words 文件，该文件用于为程序提供输入。]

timed 函数用来计算程序执行中三个不同阶段的成本；读取并按行分割数据、填充布隆过滤器和查询其中的每个元素。

如果将上述程序编译运行几次，就可以发现执行时间比较长，但多次运行之间的时长差距很小。至此，我们创建了一个看似可信的微基准测试：

```
$ ghc -O2  --make WordTest
[1 of 1] Compiling Main             ( WordTest.hs, WordTest.o )
Linking WordTest ...
$ ./WordTest
0.196347s to read words
479829 words
1.063537s to construct filter
4602978 bits
0.766899s to query every element
$ ./WordTest
0.179284s to read words
479829 words
1.069363s to construct filter
4602978 bits
0.780079s to query every element
```

25.11.1 配置驱动的性能调优

下面重新构建这个程序，并在启用分析的情况下运行，以观察哪些调优能够改善它的性能。

因为此前我们已经构建了 WordTest 且没有对源码做任何改动，如果仅仅重新运行 GHC，GHC 会认为已存在的二进制文件足够新，从而跳过重新构建。我们必须强制重新构建此程序，这里可以通过编辑源文件以更新文件系统来实现。

[Forec 译注：也可以使用 =fforce-recomp 标识，它会强迫 GHC 重新构建。]

```
$ touch WordTest.hs
$ ghc -O2 -prof -auto-all --make WordTest
```

(continues on next page)

(continued from previous page)

```
[1 of 1] Compiling Main             ( WordTest.hs, WordTest.o )
Linking WordTest ...

$ ./WordTest +RTS -p
0.322675s to read words
479829 words
suggested sizings: Right (4602978,7)
2.475339s to construct filter
1.964404s to query every element

$ head -20 WordTest.prof
total time  =          4.10 secs   (205 ticks @ 20 ms)
total alloc = 2,752,287,168 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE               %time %alloc

doubleHash                     BloomFilter.Hash      48.8   66.4
indices                        BloomFilter.Mutable   13.7   15.8
elem                           BloomFilter            9.8    1.3
hashByteString                 BloomFilter.Hash       6.8    3.8
easyList                       BloomFilter.Easy       5.9    0.3
hashIO                         BloomFilter.Hash       4.4    5.3
main                           Main                   4.4    3.8
insert                         BloomFilter.Mutable    2.9    0.0
len                            BloomFilter            2.0    2.4
length                         BloomFilter.Mutable    1.5    1.0
```

可以看出，doubleHash 占用了巨大的时空资源。

回忆一下，doubleHash 函数的主体功能是无副作用的列表解析：

```
-- file: BloomFilter/Hash.hs
doubleHash :: Hashable a => Int -> a -> [Word32]
doubleHash numHashes value = [h1 + h2 * i | i <- [0..num]]
        where h   = hashSalt 0x9150a946c4a8966e value
              h1  = fromIntegral (h `shiftR` 32) .&. maxBound
              h2  = fromIntegral h
              num = fromIntegral numHashes
```

鉴于 doubleHash 的返回值是列表，它占这么大内存似乎有点道理。但是这么简单的代码却表现出如此之差的性能，难免让人怀疑。

面对这么一个性能上的谜团，我们自然会想到检查编译器的输出。这里并不需要通过汇编语言转储来分析，从更高层次开始会更容易。

GHC 的 -ddump-simpl 选项会打印出执行所有高级优化后生成的代码：

```
$ ghc -O2 -c -ddump-simpl --make BloomFilter/Hash.hs > dump.txt
[1 of 1] Compiling BloomFilter.Hash ( BloomFilter/Hash.hs )
```

产生的 dump.txt 大约有一千行，其中多数名字是根据原始 Haskell 表示自动生成的。即使如此，搜索 doubleHash 仍然可以帮助我们立刻定位到函数的定义。下面这个例子说明了如何在 Unix Shell 中寻找到函数的定义：

```
$ less +/doubleHash dump.txt
```

刚开始阅读 GHC 简化器的输出会有些困难。这些输出包含了许多自动生成的名称，并且代码中有许多不明显的注释。我们可以忽略掉自己不了解的东西，将注意力集中在看起来很熟悉的部分上。普通的 Haskell 和 Core 语言在语法特性上有一定相似之处，尤其是类型签名、用于变量绑定的 let 和模式匹配的 case 。

如果去除 doubleHash 定义之外的部分，我们将得到类似如下所示的代码：

```
__letrec { <1>
  go_s1YC :: [GHC.Word.Word32] -> [GHC.Word.Word32] <2>
  [Arity 1
   Str: DmdType S]
  go_s1YC =
        \ (ds_a1DR :: [GHC.Word.Word32]) ->
          case ds_a1DR of wild_a1DS {
        [] -> GHC.Base.[] @ GHC.Word.Word32; <3>
        : y_a1DW ys_a1DX -> <4>
          GHC.Base.: @ GHC.Word.Word32 <5>
                (case h1_s1YA of wild1_a1Mk { GHC.Word.W32# x#_a1Mm -> <6>
                 case h2_s1Yy of wild2_a1Mu { GHC.Word.W32# x#1_a1Mw ->
                 case y_a1DW of wild11_a1My { GHC.Word.W32# y#_a1MA ->
                 GHC.Word.W32# <7>
                   (GHC.Prim.narrow32Word#
                  (GHC.Prim.plusWord# <8>
                         x#_a1Mm (GHC.Prim.narrow32Word#
                                                    (GHC.Prim.timesWord# x#1_
↪a1Mw y#_a1MA))))
                 }
                 }
                 })
                (go_s1YC ys_a1DX) <9>
          };
} in
  go_s1YC <10>
        (GHC.Word.$w$dmenumFromTo2
           __word 0 (GHC.Prim.narrow32Word# (GHC.Prim.int2Word# ww_s1X3)))
```

[Forec 译注：原著中给出的代码包含了图片，译文使用 < 编号 > 这样的标识代替图片，并在下面给出对应的注释。]

这是列表分析部分的主体。看起来似乎令人生畏，但我们可以逐步分析它。你会发现它并非那么复杂：

1. __letrec 等价于 Haskell 中的 let ；
2. GHC 将列表解析的主体部分编译成了一个名为 go_s1YC 的循环；
3. 如果 case 表达式匹配了空列表，我们就返回空列表。是不是看起来很熟悉？
4. 这个模式在 Haskell 中读作 (y_a1DW:ys_a1DX) 。(:) 构造器之所以出现在操作数之前，是因为 Core 语言出于简单起见使用了前缀表达式；
5. 这是 (:) 构造器的一种应用。符号 @ 表明第一个操作数的类型是 Word32 ；
6. 三个 case 表达式分别对一个 Word32 值拆箱以取出其中包含的原始值。首先处理的是 h1 （这里命名为 h1_s1YA ），然后是 h2 ，最后是当前列表元素 y 。拆箱是通过模式匹配实现的：W32# 是用于将原始值装箱的构造函数。按照惯例，原始类型、值以及使用它们的函数在命名时都会包含一个 #；
7. 这里我们将 W32# 构造器应用于 Word32# 类型的原始值，从而给出类型为 Word32 的正常值；
8. plusWord# 和 timesWord# 函数分别对原始的无符号整数做添加和相乘操作；
9. 这是 (:) 构造器的第二个参数，其中 go_s1YC 函数以递归的方式调用自身；
10. 这里调用了列表解析函数。它的参数是用 Core 语言表示的 [0..n] 。

阅读这段代码，我们发现了两处有趣的行为：

- 我们创建了一个列表，并且立刻在 go_s1YC 循环中解构它。GHC 通常可以检查出这种生产后立刻消费的模式，并将其转化为一个不包含资源分配的循环。这类变换称为 融合，因为生产者和消费者被融合到了一起。不幸的是，在上面的代码中 GHC 并没有为我们实现这一点。
- 在循环体中重复对 h1 和 he 开箱的行为非常浪费资源。

为了解决这些问题，我们对 doubleHash 函数做了一些细微的修改：

```
-- file: BloomFilter/Hash.hs
doubleHash :: Hashable a => Int -> a -> [Word32]
doubleHash numHashes value = go 0
        where go n | n == num  = []
                   | otherwise = h1 + h2 * n : go (n + 1)

              !h1 = fromIntegral (h `shiftR` 32) .&. maxBound
              !h2 = fromIntegral h

              h   = hashSalt 0x9150a946c4a8966e value
              num = fromIntegral numHashes
```

上面的代码中，我们手动将 `[0..num]` 表达式和“消费”它的代码合并成单个循环，并为 `h1` 和 `h2` 添加了严格注释。这些修改将 6 行代码变成 8 行，除此之外没有任何其它变动。它们会对 Core 的输出有什么影响呢?

```
__letrec {
  $wgo_s1UH :: GHC.Prim.Word# -> [GHC.Word.Word32]
  [Arity 1
   Str: DmdType L]
  $wgo_s1UH =
        \ (ww2_s1St :: GHC.Prim.Word#) ->
          case GHC.Prim.eqWord# ww2_s1St a_s1T1 of wild1_X2m {
        GHC.Base.False ->
          GHC.Base.: @ GHC.Word.Word32
                (GHC.Word.W32#
                 (GHC.Prim.narrow32Word#
                  (GHC.Prim.plusWord#
                   ipv_s1B2
                   (GHC.Prim.narrow32Word#
                (GHC.Prim.timesWord# ipv1_s1AZ ww2_s1St)))))
                ($wgo_s1UH (GHC.Prim.narrow32Word#
                                              (GHC.Prim.plusWord# ww2_s1St __word␣
↪1)));
        GHC.Base.True -> GHC.Base.[] @ GHC.Word.Word32
          };
} in  $wgo_s1UH __word 0
```

新函数被编译成了简单的计数循环。多么令人兴奋！来看看它的实际运行效果：

```
$ touch WordTest.hs
$ ghc -O2 -prof -auto-all --make WordTest
[1 of 1] Compiling Main             ( WordTest.hs, WordTest.o )
Linking WordTest ...

$ ./WordTest +RTS -p
0.304352s to read words
479829 words
suggested sizings: Right (4602978,7)
1.516229s to construct filter
1.069305s to query every element
~/src/darcs/book/examples/ch27/examples $ head -20 WordTest.prof
total time  =        3.68 secs    (184 ticks @ 20 ms)
total alloc = 2,644,805,536 bytes (excludes profiling overheads)

COST CENTRE                    MODULE                %time %alloc

doubleHash                     BloomFilter.Hash       45.1   65.0
```

(continues on next page)

(continued from previous page)

indices	BloomFilter.Mutable	19.0	16.4
elem	BloomFilter	12.5	1.3
insert	BloomFilter.Mutable	7.6	0.0
easyList	BloomFilter.Easy	4.3	0.3
len	BloomFilter	3.3	2.5
hashByteString	BloomFilter.Hash	3.3	4.0
main	Main	2.7	4.0
hashIO	BloomFilter.Hash	2.2	5.5
length	BloomFilter.Mutable	0.0	1.0

针对 doubleHash 的调整使性能提高了约 11%。对于仅仅添加两行代码的变动来说已经相当不错了。

25.12 练习

- easyList 中使用的 genericLength 在处理无限列表时会导致函数陷入无限循环。如何修正此问题?
- 困难：编写一个 QuickCheck 属性以检查观察到的错误率是否接近用户可容忍的错误率。

CHAPTER

TWENTYSIX

第 27 章：SOCKET 和 SYSLOG

26.1 基本网络

本书的前几章，我们讨论了在网络上进行操作的服务。其中两个例子是数据库客户端/服务器和 web 服务。当需要设计新的协议，或者使用没有现成 Haskell 库的协议通信时，将需要使用 Haskell 库函数提供的底层网络工具。

本章中，我们将讨论这些底层工具。网络通讯是个大题目，可以用一整本书来讨论。本章中，我们将展示如何使用 Haskell 应用你已经掌握的底层网络知识。

Haskell 的网络函数几乎始终与常见的 C 函数调用相符。像其他在 C 上层的语言一样，你将发现其接口很眼熟。

26.2 使用 UDP 通信

UDP 将数据拆散为数据包。其不保证数据到达目的地，也不确保同一个数据包到达的次数。其用校验和的方式确保到达的数据包没有损坏。UDP 适合用在对性能和延迟敏感的应用中，此类场景中系统的整体性能比单个数据包更重要。也可以用在 TCP 表现性能不高的场景，比如发送互不相关的短消息。适合使用 UDP 的系统的例子包括音频和视频会议、时间同步、网络文件系统、以及日志系统。

26.2.1 UDP 客户端例子：syslog

传统 Unix syslog 服务允许程序通过网络向某个负责记录的中央服务器发送日志信息。某些程序对性能非常敏感，而且可能会生成大量日志消息。这样的程序，将日志的开销最小化比确保每条日志被记录更重要。此外，在日志服务器无法访问时，使程序依旧可以操作或许是一种可取的设计。因此，UDP 是一种 syslog 支持的日志传输协议。这种协议比较简单，这里有一个 Haskell 实现的客户端：

```
-- file: ch27/syslogclient.hs
import Data.Bits
import Network.Socket
```

(continues on next page)

(continued from previous page)

```
import Network.BSD
import Data.List
import SyslogTypes

data SyslogHandle =
    SyslogHandle {slSocket :: Socket,
                  slProgram :: String,
                  slAddress :: SockAddr}

openlog :: HostName             -- ^ Remote hostname, or localhost
        -> String               -- ^ Port number or name; 514 is default
        -> String               -- ^ Name to log under
        -> IO SyslogHandle      -- ^ Handle to use for logging
openlog hostname port progname =
    do -- Look up the hostname and port.  Either raises an exception
       -- or returns a nonempty list.  First element in that list
       -- is supposed to be the best option.
       addrinfos <- getAddrInfo Nothing (Just hostname) (Just port)
       let serveraddr = head addrinfos

       -- Establish a socket for communication
       sock <- socket (addrFamily serveraddr) Datagram defaultProtocol

       -- Save off the socket, program name, and server address in a handle
       return $ SyslogHandle sock progname (addrAddress serveraddr)

syslog :: SyslogHandle -> Facility -> Priority -> String -> IO ()
syslog syslogh fac pri msg =
    sendstr sendmsg
    where code = makeCode fac pri
          sendmsg = "<" ++ show code ++ ">" ++ (slProgram syslogh) ++
                    ": " ++ msg

          -- Send until everything is done
          sendstr :: String -> IO ()
          sendstr [] = return ()
          sendstr omsg = do sent <- sendTo (slSocket syslogh) omsg
                                    (slAddress syslogh)
                            sendstr (genericDrop sent omsg)

closelog :: SyslogHandle -> IO ()
closelog syslogh = sClose (slSocket syslogh)
```

(continues on next page)

(continued from previous page)

```
{- | Convert a facility and a priority into a syslog code -}
makeCode :: Facility -> Priority -> Int
makeCode fac pri =
    let faccode = codeOfFac fac
        pricode = fromEnum pri
        in
          (faccode `shiftL` 3) .|. pricode
```

这段程序需要 SyslogTypes.hs，代码如下：

```
-- file: ch27/SyslogTypes.hs
module SyslogTypes where
{- | Priorities define how important a log message is. -}

data Priority =
            DEBUG                   -- ^ Debug messages
          | INFO                    -- ^ Information
          | NOTICE                  -- ^ Normal runtime conditions
          | WARNING                 -- ^ General Warnings
          | ERROR                   -- ^ General Errors
          | CRITICAL                -- ^ Severe situations
          | ALERT                   -- ^ Take immediate action
          | EMERGENCY               -- ^ System is unusable
                  deriving (Eq, Ord, Show, Read, Enum)

{- | Facilities are used by the system to determine where messages
are sent. -}

data Facility =
              KERN                      -- ^ Kernel messages
              | USER                    -- ^ General userland messages
              | MAIL                    -- ^ E-Mail system
              | DAEMON                  -- ^ Daemon (server process) messages
              | AUTH                    -- ^ Authentication or security messages
              | SYSLOG                  -- ^ Internal syslog messages
              | LPR                     -- ^ Printer messages
              | NEWS                    -- ^ Usenet news
              | UUCP                    -- ^ UUCP messages
              | CRON                    -- ^ Cron messages
              | AUTHPRIV                -- ^ Private authentication messages
              | FTP                     -- ^ FTP messages
              | LOCAL0
              | LOCAL1
              | LOCAL2
```

(continues on next page)

(continued from previous page)

```
              | LOCAL3
              | LOCAL4
              | LOCAL5
              | LOCAL6
              | LOCAL7
                deriving (Eq, Show, Read)

facToCode = [
                        (KERN, 0),
                        (USER, 1),
                        (MAIL, 2),
                        (DAEMON, 3),
                        (AUTH, 4),
                        (SYSLOG, 5),
                        (LPR, 6),
                        (NEWS, 7),
                        (UUCP, 8),
                        (CRON, 9),
                        (AUTHPRIV, 10),
                        (FTP, 11),
                        (LOCAL0, 16),
                        (LOCAL1, 17),
                        (LOCAL2, 18),
                        (LOCAL3, 19),
                        (LOCAL4, 20),
                        (LOCAL5, 21),
                        (LOCAL6, 22),
                        (LOCAL7, 23)
            ]

codeToFac = map (\(x, y) -> (y, x)) facToCode


{- | We can't use enum here because the numbering is discontiguous -}
codeOfFac :: Facility -> Int
codeOfFac f = case lookup f facToCode of
                Just x -> x
                _ -> error $ "Internal error in codeOfFac"

facOfCode :: Int -> Facility
facOfCode f = case lookup f codeToFac of
                Just x -> x
                _ -> error $ "Invalid code in facOfCode"
```

可以用 ghci 向本地的 syslog 服务器发送消息。服务器可以使用本章实现的例子，也可以使用其它的在 Linux 或者 POSIX 系统中的 syslog 服务器。注意，这些服务器默认禁用了 UDP 端口，你需要启用 UDP 以使 syslog 接收 UDP 消息。

可以使用下面这样的命令向本地 syslog 服务器发送一条消息：

```
ghci> :load syslogclient.hs
[1 of 2] Compiling SyslogTypes      ( SyslogTypes.hs, interpreted )
[2 of 2] Compiling Main             ( syslogclient.hs, interpreted )
Ok, modules loaded: SyslogTypes, Main.
ghci> h <- openlog "localhost" "514" "testprog"
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
ghci> syslog h USER INFO "This is my message"
ghci> closelog h
```

26.2.2 UDP Syslog 服务器

UDP 服务器会在服务器上绑定某个端口。其接收直接发到这个端口的包，并处理它们。UDP 是无状态的，面向包的协议，程序员通常使用 `recvFrom` 这个调用接收消息和发送机信息，在发送响应时会用到发送机信息。

```
-- file: ch27/syslogserver.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List

type HandlerFunc = SockAddr -> String -> IO ()

serveLog :: String              -- ^ Port number or name; 514 is default
         -> HandlerFunc         -- ^ Function to handle incoming messages
         -> IO ()
serveLog port handlerfunc = withSocketsDo $
    do -- Look up the port.  Either raises an exception or returns
       -- a nonempty list.
       addrinfos <- getAddrInfo
                    (Just (defaultHints {addrFlags = [AI_PASSIVE]}))
                    Nothing (Just port)
       let serveraddr = head addrinfos

       -- Create a socket
       sock <- socket (addrFamily serveraddr) Datagram defaultProtocol
```

(continues on next page)

(continued from previous page)

```
        -- Bind it to the address we're listening to
        bindSocket sock (addrAddress serveraddr)

        -- Loop forever processing incoming data.  Ctrl-C to abort.
        procMessages sock
    where procMessages sock =
              do -- Receive one UDP packet, maximum length 1024 bytes,
                 -- and save its content into msg and its source
                 -- IP and port into addr
                 (msg, _, addr) <- recvFrom sock 1024
                 -- Handle it
                 handlerfunc addr msg
                 -- And process more messages
                 procMessages sock

-- A simple handler that prints incoming packets
plainHandler :: HandlerFunc
plainHandler addr msg =
    putStrLn $ "From " ++ show addr ++ ": " ++ msg
```

这段程序可以在 ghci 中执行。执行 `serveLog "1514" plainHandler` 将建立一个监听 1514 端口的 UDP 服务器。其使用 `plainHandler` 将每条收到的 UDP 包打印出来。按下 `Ctrl-C` 可以终止这个程序。

Note: 处理错误。执行时收到了 `bind: permission denied` 消息？要确保端口值比 1024 大。某些操作系统不允许 root 之外的用户使用小于 1024 的端口。

26.3 使用 TCP 通信

TCP 被设计为确保互联网上的数据尽可能可靠地传输。TCP 是数据流传输。虽然流在传输时会被操作系统拆散为一个个单独的包，但是应用程序并不需要关心包的边界。TCP 负责确保如果流被传送到应用程序，它就是完整的、无改动、仅传输一次且保证顺序。显然，如果线缆被破坏会导致流量无法送达，任何协议都无法克服这类限制。

与 UDP 相比，这带来一些折衷。首先，在 TCP 会话开始必须传递一些包以建立连接。其次，对于每个短会话，UDP 将有性能优势。另外，TCP 会努力确保数据到达。如果会话的一端尝试向远端发送数据，但是没有收到响应，它将周期性的尝试重新传输数据直至放弃。这使得 TCP 面对丢包时比较健壮可靠。可是，它同样意味着 TCP 不是实时传输协议（如实况音频或视频传输）的最佳选择。

26.3.1 处理多个 TCP 流

TCP 的连接是有状态的。这意味着每个客户机和服务器之间都有一条专用的逻辑“频道”，而不是像 UDP 一样只是处理一次性的数据包。这简化了客户端开发者的工作。服务器端程序几乎总是需要同时处理多条 TCP 连接。如何做到这一点呢?

在服务器端，首先需要创建一个 socket 并绑定到某个端口，就像 UDP 一样。但这回不是重复监听从任意地址发来的数据，取而代之，你的主循环将围绕 `accept` 调用编写。每当有一个客户机连接，服务器操作系统为其分配一个新的 socket 。所以我们的主 socket 只用来监听进来的连接，但从不发送数据。我们也获得了多个子 socket 可以同时使用，每个子 socket 从属于一个逻辑上的 TCP 会话。

在 Haskell 中，通常使用 `forkIO` 创建一个单独的轻量级线程以处理与子 socket 的通信。对此，Haskell 拥有一个高效的内部实现，执行得非常好。

26.3.2 TCP Syslog 服务器

让我们使用 TCP 的实现来替换 UDP 的 syslog 服务器。假设一条消息并不是定义为单独的包，而是以一个尾部的字符‘n’结束。任意客户端可以使用 TCP 连接向服务器发送 0 或多条消息。我们可以像下面这样实现:

```
-- file: ch27/syslogtcpserver.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List
import Control.Concurrent
import Control.Concurrent.MVar
import System.IO

type HandlerFunc = SockAddr -> String -> IO ()

serveLog :: String              -- ^ Port number or name; 514 is default
         -> HandlerFunc         -- ^ Function to handle incoming messages
         -> IO ()
serveLog port handlerfunc = withSocketsDo $
    do -- Look up the port.  Either raises an exception or returns
       -- a nonempty list.
       addrinfos <- getAddrInfo
                    (Just (defaultHints {addrFlags = [AI_PASSIVE]}))
                    Nothing (Just port)
       let serveraddr = head addrinfos

       -- Create a socket
       sock <- socket (addrFamily serveraddr) Stream defaultProtocol
```

(continues on next page)

(continued from previous page)

```
        -- Bind it to the address we're listening to
        bindSocket sock (addrAddress serveraddr)

        -- Start listening for connection requests.  Maximum queue size
        -- of 5 connection requests waiting to be accepted.
        listen sock 5

        -- Create a lock to use for synchronizing access to the handler
        lock <- newMVar ()

        -- Loop forever waiting for connections.  Ctrl-C to abort.
        procRequests lock sock

    where
        -- | Process incoming connection requests
        procRequests :: MVar () -> Socket -> IO ()
        procRequests lock mastersock =
            do (connsock, clientaddr) <- accept mastersock
               handle lock clientaddr
                  "syslogtcpserver.hs: client connnected"
               forkIO $ procMessages lock connsock clientaddr
               procRequests lock mastersock

        -- | Process incoming messages
        procMessages :: MVar () -> Socket -> SockAddr -> IO ()
        procMessages lock connsock clientaddr =
            do connhdl <- socketToHandle connsock ReadMode
               hSetBuffering connhdl LineBuffering
               messages <- hGetContents connhdl
               mapM_ (handle lock clientaddr) (lines messages)
               hClose connhdl
               handle lock clientaddr
                  "syslogtcpserver.hs: client disconnected"

        -- Lock the handler before passing data to it.
        handle :: MVar () -> HandlerFunc
        -- This type is the same as
        -- handle :: MVar () -> SockAddr -> String -> IO ()
        handle lock clientaddr msg =
            withMVar lock
               (\a -> handlerfunc clientaddr msg >> return a)

-- A simple handler that prints incoming packets
```

(continues on next page)

(continued from previous page)

```
plainHandler :: HandlerFunc
plainHandler addr msg =
    putStrLn $ "From " ++ show addr ++ ": " ++ msg
```

SyslogTypes 的实现，见*UDP 客户端例子：syslog* 。

让我们读一下源码。主循环是 procRequests ，这是一个死循环，用于等待来自客户端的新连接。accept 调用将一直阻塞，直到一个客户端来连接。当有客户端连接，我们获得一个新 socket 和客户机地址。我们向处理函数发送一条关于新连接的消息，接着使用 forkIO 建立一个线程处理来自客户机的数据。这条线程执行 procMessages 。

处理 TCP 数据时，为了方便，通常将 socket 转换为 Haskell 句柄。我们也同样处理，并明确设置了缓冲 – 一个 TCP 通信的要点。接着，设置惰性读取 socket 句柄。对每个传入的行，我们都将其传给 handle 。当没有更多数据时 – 远端已经关闭了 socket – 我们输出一条会话结束的消息。

因为可能同时收到多条消息，我们需要确保没有将多条消息同时写入一个处理函数。那将导致混乱的输出。我们使用了一个简单的锁以序列化对处理函数的访问，并且编写了一个简单的 handle 函数处理它。

你可以使用下面我们将展示的客户机代码测试，或者直接使用 telnet 程序来连接这个服务器。你向其发送的每一行输入都将被服务器原样返回。我们来试一下：

```
ghci> :load syslogtcpserver.hs
[1 of 1] Compiling Main             ( syslogtcpserver.hs, interpreted )
Ok, modules loaded: Main.
ghci> serveLog "10514" plainHandler
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
```

此处，服务器从 10514 端口监听新连接。在有某个客户机过来连接之前，它什么事儿都不做。我们可以使用 telnet 来连接这个服务器：

```
~$ telnet localhost 10514
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Test message
^]
telnet> quit
Connection closed.
```

于此同时，在我们运行 TCP 服务器的终端上，你将看到如下输出：

```
From 127.0.0.1:38790: syslogtcpserver.hs: client connnected
From 127.0.0.1:38790: Test message
From 127.0.0.1:38790: syslogtcpserver.hs: client disconnected
```

其显示一个客户端从本机 (127.0.0.1) 的 38790 端口连上了主机。连接之后，它发送了一条消息，然后断开。当你扮演一个 TCP 客户端时，操作系统将分配一个未被使用的端口给你。通常这个端口在你每次运行程序时都不一样。

26.3.3 TCP Syslog 客户端

现在，为我们的 TCP syslog 协议编写一个客户端。这个客户端与 UDP 客户端类似，但是有一些变化。首先，因为 TCP 是流式协议，我们可以使用句柄传输数据而不需要使用底层的 socket 操作。其次，不在需要在 `SyslogHandle` 中保存目的地址，因为我们将使用 `connect` 建立 TCP 连接。最后，我们需要一个途径，以区分不同的消息。UDP 中，这很容易，因为每条消息都是不相关的逻辑包。TCP 中，我们将仅使用换行符 'n' 来作为消息结尾的标识，尽管这意味着不能在单条消息中发送多行信息。这是代码：

```
-- file: ch27/syslogtcpclient.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List
import SyslogTypes
import System.IO

data SyslogHandle =
    SyslogHandle {slHandle :: Handle,
                  slProgram :: String}

openlog :: HostName                -- ^ Remote hostname, or localhost
        -> String                  -- ^ Port number or name; 514 is default
        -> String                  -- ^ Name to log under
        -> IO SyslogHandle         -- ^ Handle to use for logging
openlog hostname port progname =
    do -- Look up the hostname and port.  Either raises an exception
       -- or returns a nonempty list.  First element in that list
       -- is supposed to be the best option.
       addrinfos <- getAddrInfo Nothing (Just hostname) (Just port)
       let serveraddr = head addrinfos

       -- Establish a socket for communication
       sock <- socket (addrFamily serveraddr) Stream defaultProtocol

       -- Mark the socket for keep-alive handling since it may be idle
       -- for long periods of time
       setSocketOption sock KeepAlive 1

       -- Connect to server
```

(continues on next page)

(continued from previous page)

```
        connect sock (addrAddress serveraddr)

        -- Make a Handle out of it for convenience
        h <- socketToHandle sock WriteMode

        -- We're going to set buffering to BlockBuffering and then
        -- explicitly call hFlush after each message, below, so that
        -- messages get logged immediately
        hSetBuffering h (BlockBuffering Nothing)

        -- Save off the socket, program name, and server address in a handle
        return $ SyslogHandle h progname

syslog :: SyslogHandle -> Facility -> Priority -> String -> IO ()
syslog syslogh fac pri msg =
    do hPutStrLn (slHandle syslogh) sendmsg
       -- Make sure that we send data immediately
       hFlush (slHandle syslogh)
    where code = makeCode fac pri
          sendmsg = "<" ++ show code ++ ">" ++ (slProgram syslogh) ++
                    ": " ++ msg

closelog :: SyslogHandle -> IO ()
closelog syslogh = hClose (slHandle syslogh)

{- | Convert a facility and a priority into a syslog code -}
makeCode :: Facility -> Priority -> Int
makeCode fac pri =
    let faccode = codeOfFac fac
        pricode = fromEnum pri
        in
          (faccode `shiftL` 3) .|. pricode
```

可以在 ghci 中试着运行它。如果还没有关闭之前的 TCP 服务器，你的会话看上去可能会像是这样：

```
ghci> :load syslogtcpclient.hs
Loading package base ... linking ... done.
[1 of 2] Compiling SyslogTypes      ( SyslogTypes.hs, interpreted )
[2 of 2] Compiling Main             ( syslogtcpclient.hs, interpreted )
Ok, modules loaded: Main, SyslogTypes.
ghci> openlog "localhost" "10514" "tcptest"
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
ghci> sl <- openlog "localhost" "10514" "tcptest"
```

(continues on next page)

(continued from previous page)

```
ghci> syslog sl USER INFO "This is my TCP message"
ghci> syslog sl USER INFO "This is my TCP message again"
ghci> closelog sl
```

结束时，服务器上将看到这样的输出：

```
From 127.0.0.1:46319: syslogtcpserver.hs: client connnected
From 127.0.0.1:46319: <9>tcptest: This is my TCP message
From 127.0.0.1:46319: <9>tcptest: This is my TCP message again
From 127.0.0.1:46319: syslogtcpserver.hs: client disconnected
```

<9> 是优先级和设施代码，和之前 UDP 例子中的意思一样。

CHAPTER

TWENTYSEVEN

第 28 章：软件事务内存 (STM)

在并发编程的传统线程模型中，线程之间的数据共享需要通过锁来保持一致性 (consistentBalance)，当数据产生变化时，还需要使用条件变量 (condition variable) 对各个线程进行通知。

某种程度上，Haskell 的 `MVar` 机制对上面提到的工具进行了改进，但是，它仍然带有和这些工具一样的缺陷：

- 因为忘记使用锁而导致条件竞争 (race condition)
- 因为不正确的加锁顺序而导致死锁 (deadblock)
- 因为未被捕捉的异常而造成程序崩溃 (corruption)
- 因为错误地忽略了通知，造成线程无法正常唤醒 (lost wakeup)

这些问题即使在很小的并发程序里也会经常发生，而在更加庞大的代码库或是高负载的情况下，这些问题会引发更加糟糕的难题。

比如说，对一个只有几个大范围锁的程序进行编程并不难，只是一旦这个程序在高负载的环境下运行，锁之间的相互竞争就会变得非常严重。另一方面，如果采用细粒度 (fineo-grained) 的锁机制，保持软件正常工作将会变得非常困难。除此之外，就算在负载不高的情况下, 加锁带来的额外的簿记工作 (book-keeping) 也会对性能产生影响。

27.1 基础知识

软件事务内存 (Software transactional memory) 提供了一些简单但强大的工具。通过这些工具我们可以解决前面提到的大多数问题。通过 `atomically` 组合器 (combinator), 我们可以在一个事务内执行一批操作。当这一组操作开始执行的时候，其他线程是觉察不到这些操作所产生的任何修改，直到所有操作完成。同样的，当前线程也无法察觉其他线程的所产生的修改。这些性质表明的操作的隔离性 (isolated)。

当从一个事务退出的时候，只会发生以下情况中的一种:

- 如果没有其他线程修改了同样的数据，当前线程产生的修改将会对所有其他线程可见。
- 否则，当前线程的所产生的改动会被丢弃，然后这组操作会被重新执行。

`atomically` 这种全有或全无 (all-or-nothing) 的天性被称之为原子性 (atomic)，`atomically` 也因为得名。如果你使用过支持事务的数据库，你会觉得 STM 使用起来非常熟悉。

27.2 一些简单的例子

在多玩家角色扮演的游戏里，一个玩家的角色会有许多属性，比如健康，财产以及金钱。让我们从基于游戏人物属性的一些简单的函数和类型开始去了解 STM 的精彩内容。随着学习的深入，我们也会不断地改进我们的代码。

STM 的 API 位于 `stm` 包，模块 `Control.Concurrent.STM` 。

```
-- file: ch28/GameInventory.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Concurrent.STM
import Control.Monad

data Item = Scroll
          | Wand
          | Banjo
            deriving (Eq, Ord, Show)

newtype Gold = Gold Int
    deriving (Eq, Ord, Show, Num)

newtype HitPoint = HitPoint Int
    deriving (Eq, Ord, Show, Num)

type Inventory = TVar [Item]
type Health = TVar HitPoint
type Balance = TVar Gold

data Player = Player {
      balance :: Balance,
      health :: Health,
      inventory :: Inventory
}
```

参数化类型 `TVar` 是一个可变量，可以在 `atomically` 块中读取或者修改。为了简单起见，我们把玩家的背包 (Inventory) 定义为物品的列表。同时注意到，我们用到了 `newtype` ，这样不会混淆财富和健康属性。

当需要在两个账户 (Balance) 之间转账，我们所要的做的就只是调整下各自的 `Tvar` 。

```
-- file: ch28/GameInventory.hs
basicTransfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  toQty   <- readTVar toBal
  writeTVar fromBal (fromQty - qty)
  writeTVar toBal   (toQty + qty)
```

让我们写个简单的测试函数

```
-- file: ch28/GameInventory.hs
transferTest = do
  alice <- newTVar (12 :: Gold)
  bob   <- newTVar 4
  basicTransfer 3 alice bob
  liftM2 (,) (readTVar alice) (readTVar bob)
```

如果我们在 ghci 里执行下这个函数，应该有如下的结果

```
ghci> :load GameInventory
[1 of 1] Compiling Main             ( GameInventory.hs, interpreted )
Ok, modules loaded: Main.
ghci> atomically transferTest
Loading package array-0.4.0.0 ... linking ... done.
Loading package stm-2.3 ... linking ... done.
(Gold 9,Gold 7)
```

原子性和隔离性保证了当其他线程同时看到 bob 的账户和 alice 的账户被修改了。

即使在并发程序里，我们也努力保持代码尽量的纯函数化。这使得我们的代码更加容易推导和测试。由于数据并没有事务性，这也让底层的 STM 做更少的事。以下的纯函数实现了从我们来表示玩家背包的数列里移除一个物品。

```
-- file: ch28/GameInventory.hs
removeInv :: Eq a => a -> [a] -> Maybe [a]
removeInv x xs =
    case takeWhile (/= x) xs of
      (_:ys) -> Just ys
      []     -> Nothing
```

这里返回值用了 Maybe 类型，它可以用来表示物品是否在玩家的背包里。

下面这个事务性的函数实现了把一个物品给另外一个玩家。这个函数有一点点复杂因为需要判断给予者是否有这个物品。

```haskell
-- file: ch28/GameInventory.hs
maybeGiveItem item fromInv toInv = do
  fromList <- readTVar fromInv
  case removeInv item fromList of
    Nothing      -> return False
    Just newList -> do
      writeTVar fromInv newList
      destItems <- readTVar toInv
      writeTVar toInv (item : destItems)
      return True
```

27.3 STM 的安全性

既然我们提供了有原子性和隔离型的事务，那么保证我们不能有意或是无意的从 `atomically` 执行块从脱离显得格外重要。借由 `STM` monad，Haskell 的类型系统保证了我们这种行为。

```
ghci> :type atomically
atomically :: STM a -> IO a
```

`atomically` 接受一个 `STM` monad 的动作，然后执行并让我们可以从 `IO` monad 里拿到这个结果。`STM` monad 是所有事务相关代码执行的地方。比如这些操作 `TVar` 值的函数都在 `STM` monad 里被执行。

```
ghci> :type newTVar
newTVar :: a -> STM (TVar a)
ghci> :type readTVar
readTVar :: TVar a -> STM a
ghci> :type writeTVar
writeTVar :: TVar a -> a -> STM ()
```

我们之前定义的事务性函数也有这个特性

```haskell
-- file: ch28/GameInventory.hs
basicTransfer :: Gold -> Balance -> Balance -> STM ()
maybeGiveItem :: Item -> Inventory -> Inventory -> STM Bool
```

在 `STM` monad 里是不允许执行 I/O 操作或者是修改非事务性的可变状态，比如 `MVar` 的值。这就使得我们可以避免那些违背事务完整的操作。

27.4 重试一个事务

maybeGiveItem 这个函数看上去稍微有点怪异。只有当角色有这个物品时才会将它给另外一个角色，这看上去还算合理，然后返回一个 Bool 值使调用这个函数的代码变得复杂。下面这个函数调用了 maybeGiveItem，它必须根据 maybeGiveItem 的返回结果来决定如何继续执行。

```
maybeSellItem :: Item -> Gold -> Player -> Player -> STM Bool
maybeSellItem item price buyer seller = do
  given <- maybeGiveItem item (inventory seller) (inventory buyer)
  if given
    then do
      basicTransfer price (balance buyer) (balance seller)
      return True
    else return False
```

我们不仅要检查物品是否给到了另一个玩家，而且还得把是否成功这个信号传递给调用者。这就意味了复杂性被延续到了更外层。

下面我们来看看如何用更加优雅的方式处理事务无法成功进行的情况。STM API 提供了一个 retry 函数，它可以立即中断一个无法成功进行的 atomically 执行块。正如这个函数名本身所指明的意思，当它发生时，执行块会被重新执行，所有在这之前的操作都不会被记录。我们使用 retry 重新实现了 maybeGiveItem 。

```
-- file: ch28/GameInventory.hs
giveItem :: Item -> Inventory -> Inventory -> STM ()

giveItem item fromInv toInv = do
    fromList <- readTVar fromInv
    case removeInv item fromList of
        Nothing -> retry
        Just newList -> do
            writeTVar fromInv newList
            readTVar toInv >>= writeTVar toInv . (item :)
```

我们之前实现的 basicTransfer 有一个缺陷：没有检查发送者的账户是否有足够的资金。我们可以使用 retry 来纠正这个问题并保持方法签名不变。

```
-- file: ch28/GameInventory.hs
transfer :: Gold -> Balance -> Balance -> STM ()

transfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  when (qty > fromQty) $
    retry
  writeTVar fromBal (fromQty - qty)
```

(continues on next page)

(continued from previous page)

```
  readTVar toBal >>= writeTVar toBal . (qty +)
```

使用 retry 后，销售物品的函数就显得简单很多。

```
sellItem :: Item -> Gold -> Player -> Player -> STM ()
sellItem item price buyer seller = do
  giveItem item (inventory seller) (inventory buyer)
  transfer price (balance buyer) (balance seller)
```

这个实现和之前的稍微有点不同。如果有必要会会阻塞以至卖家有东西可卖并且买家有足够的余额支付，而不是在发现卖家没这个物品可销售时马上返回 False 。

27.4.1 retry 时到底发生了什么?

retry 不仅仅使得代码更加简洁：它似乎有魔力般的内部实现。当我们调用 retry 的时候，它并不是马上重启事务，而是会先阻塞线程，一直到那些在 retry 之前被访问过的变量被其他线程修改。

比如，如果我们调用 transfer 而发现余额不足，retry 会自发的等待，直到账户余额的变动，然后会重新启动事务。同样的，对于函数 giveItem ，如果卖家没有那个物品，线程就会阻塞直到他有了那个物品。

27.5 选择替代方案

有时候我们并不总是希望重启 atomically 操作即使调用了 retry 或者由于其他线程的同步修改而导致的失败。比如函数 sellItem 会不断地重试，只要没有满足其条件：要有物品并且余额足够。然而我们可能更希望只重试一次。

orElse 组合器允许我们在主操作失败的情况下，执行一个”备用”操作。

```
ghci> :type orElse
orElse :: STM a -> STM a -> STM a
```

我们对 sellItem 做了一点修改：如果 sellItem 失败，则 orElse 执行 return False 的动作从而使这个 sale 函数立即返回。

```
trySellItem :: Item -> Gold -> Player -> Player -> STM Bool
trySellItem item price buyer seller =
   sellItem item price buyer seller >> return True
  `orElse`
   return False
```

27.5.1 在事务中使用高阶代码

假设我们想做稍微有挑战的事情，从一系列的物品中，选取第一个卖家拥有的并且买家能承担费用的物品进行购买，如果没有这样的物品则什么都不做。显然我们可以很直观的给出实现。

```
-- file: ch28/GameInventory.hs
crummyList :: [(Item, Gold)] -> Player -> Player
              -> STM (Maybe (Item, Gold))
crummyList list buyer seller = go list
    where go []                           = return Nothing
          go (this@(item,price) : rest) = do
              sellItem item price buyer seller
              return (Just this)
           `orElse`
              go rest
```

在这个实现里，我们有碰到了一个熟悉的问题：把我们的需求和如果实现混淆在一个。再深入一点观察，则会发现两个可重复使用的模式。

第一个就是让事务失败而不是重试。

```
-- file: ch28/GameInventory.hs
maybeSTM :: STM a -> STM (Maybe a)
maybeSTM m = (Just `liftM` m) `orElse` return Nothing
```

第二个，我们要对一系列的对象执行否一个操作，直到有一个成功为止。如果全部都失败，则执行 `retry` 操作。由于 `STM` 是 `MonadPlus` 类型类的一个实例，所以显得很方便。

```
-- file: ch28/STMPlus.hs
instance MonadPlus STM where
  mzero = retry
  mplus = orElse
```

`Control.Monad` 模块定义了一个 `msum` 函数，而它就是我们所需要的。

```
-- file: ch28/STMPlus.hs
msum :: MonadPlus m => [m a] -> m a
msum =  foldr mplus mzero
```

有了这些重要的工具，我们就可以写出更加简洁的实现了。

```
-- file: ch28/GameInventory.hs
shoppingList :: [(Item, Gold)] -> Player -> Player
              -> STM (Maybe (Item, Gold))
shoppingList list buyer seller = maybeSTM . msum $ map sellOne list
```

(continues on next page)

(continued from previous page)

```
    where sellOne this@(item,price) = do
            sellItem item price buyer seller
            return this
```

既然 STM 是 MonadPlus 类型类的实例，我们可以改进 maybeSTM，这样就可以适用于任何 MonadPlus 的实例。

```
-- file: ch28/GameInventory.hs
maybeM :: MonadPlus m => m a -> m (Maybe a)
maybeM m = (Just `liftM` m) `mplus` return Nothing
```

这个函数会在很多不同情况下显得非常有用。

27.6 I/O 和 STM

STM monad 禁止任意的 I/O 操作，因为 I/O 操作会破坏原子性和隔离性。当然 I/O 的操作还是需要的，只是我们需要非常的谨慎。

大多数时候，我们会执行 I/O 操作是由于我们在 atomically 块中产生的一个结果。在这些情况下，正确的做法通常是 atomically 返回一些数据，在 I/O monad 里的调用者则根据这些数据知道如何继续下一步动作。我们甚至可以返回需要被操作的动作 (action)，因为他们是第一类值 (First Class vaules)。

```
-- file: ch28/STMIO.hs
someAction :: IO a

stmTransaction :: STM (IO a)
stmTransaction = return someAction

doSomething :: IO a
doSomething = join (atomically stmTransaction)
```

我们偶尔也需要在 STM 里进行 I/O 操作。比如从一个肯定存在的文件里读取一些非可变数据，这样的操作并不会违背 STM 保证原子性和隔离性的原则。在这些情况，我们可以使用 unsafeIOToSTM 来执行一个 IO 操作。这个函数位于偏底层的一个模块 GHC.Conc，所以要谨慎使用。

```
ghci> :m +GHC.Conc
ghci> :type unsafeIOToSTM
unsafeIOToSTM :: IO a -> STM a
```

我们所执行的这个 IO 动作绝对不能打开另外一个 atomically 事务。如果一个线程尝试嵌套的事务，系统就会抛出异常。

由于类型系统无法帮助我们确保 IO 代码没有执行一些敏感动作，最安全的做法就是我们尽量的限制使用 unsafeIOToSTM 。下面的例子展示了在 atomically 中执行 IO 的典型错误。

```
-- file: ch28/STMIO.hs
launchTorpedoes :: IO ()

notActuallyAtomic = do
  doStuff
  unsafeIOToSTM launchTorpedoes
  mightRetry
```

如果 mightRetry 会引发事务的重启，那么 launchTorpedoes 会被调用多次。事实上，我们无法预见它会被调用多少次，因为重试是由运行时系统所处理的。解决方案就是在事务中不要有这种类型的 non-idempotent I/O 操作。

27.7 线程之间的通讯

正如基础类型 TVar 那样，stm 包也提供了两个更有用的类型用于线程之间的通讯，TMVar 和 TChan 。TMVar 是 STM 世界的 MVar ，它可以保存一个 Maybe 类型的值，即 Just 值或者 Nothing 。TChan 则是 STM 世界里的 Chan ，它实现了一个有类型的先进先出 (FIFO) 通道。

[译者注：为何说 TMVar 是 STM 世界的 MVar 而不是 TVar ？是因为从实践意义上理解的。MVar 的特性是要么有值要么为空的一个容器，所以当线程去读这个容器时，要么读到值继续执行，要么读不到值就等待。而 TVar 并没有这样的特性，所以引入了 TMVar 。它的实现是这样的，newtype TMVar a = TMVar (TVar (Maybe a)) ，正是由于它包含了一个 Maybe 类型的值，这样就有了”要么有值要么为空”这样的特性，也就是 MVar 所拥有的特性。]

27.8 并发网络链接检查器

作为一个使用 STM 的实际例子，我们将开发一个检查 HTML 文件里不正确链接的程序，这里不正确的链接是指那些链接指向了一个错误的网页或是无法访问到其指向的服务器。用并发的方式解决这个问题非常得合适：如果我们尝试和已经下线的服务器 (dead server) 通讯，需要有两分钟的超时时间。如果使用多线程，即使有一两个线程由于和响应很慢或者下线的服务器通讯而停住 (stuck)，我们还是可以继续进行一些有用的事情。

我们不能简单直观的给每一个 URL 新建一个线程，因为由于（也是我们预想的）大多数链接是正确的，那么这样做就会导致 CPU 或是网络连接超负荷。因此，我们只会创建固定数量的线程，这些线程会从一个队列里拿 URL 做检查。

```
-- file: ch28/Check.hs
{-# LANGUAGE FlexibleContexts, GeneralizedNewtypeDeriving,
```

(continues on next page)

(continued from previous page)

```
             PatternGuards #-}

import Control.Concurrent (forkIO)
import Control.Concurrent.STM
import Control.Exception (catch, finally)
import Control.Monad.Error
import Control.Monad.State
import Data.Char (isControl)
import Data.List (nub)
import Network.URI
import Prelude hiding (catch)
import System.Console.GetOpt
import System.Environment (getArgs)
import System.Exit (ExitCode(..), exitWith)
import System.IO (hFlush, hPutStrLn, stderr, stdout)
import Text.Printf (printf)
import qualified Data.ByteString.Lazy.Char8 as B
import qualified Data.Set as S

-- 这里需要 HTTP 包, 它并不是 GHC 自带的.
import Network.HTTP

type URL = B.ByteString

data Task = Check URL | Done
```

main 函数显示了这个程序的主体脚手架 (scaffolding)。

```
-- file: ch28/Check.hs
main :: IO ()
main = do
    (files,k) <- parseArgs
    let n = length files

    -- count of broken links
    badCount <- newTVarIO (0 :: Int)

    -- for reporting broken links
    badLinks <- newTChanIO

    -- for sending jobs to workers
    jobs <- newTChanIO

    -- the number of workers currently running
```

(continues on next page)

(continued from previous page)

```
    workers <- newTVarIO k

    -- one thread reports bad links to stdout
    forkIO $ writeBadLinks badLinks

    -- start worker threads
    forkTimes k workers (worker badLinks jobs badCount)

    -- read links from files, and enqueue them as jobs
    stats <- execJob (mapM_ checkURLs files)
                     (JobState S.empty 0 jobs)

    -- enqueue "please finish" messages
    atomically $ replicateM_ k (writeTChan jobs Done)

    waitFor workers

    broken <- atomically $ readTVar badCount

    printf fmt broken
                (linksFound stats)
                (S.size (linksSeen stats))
                n
  where
    fmt    = "Found %d broken links. " ++
             "Checked %d links (%d unique) in %d files.\n"
```

当我们处于 `IO` monad 时，可以使用 `newTVarIO` 函数新建一个 `TVar` 值。同样的，也有类似的函数可以新建 `TMVar` 和 `TChan` 值。

在程序用了 `printf` 函数打印出最后的结果。和 C 语言里类似函数 `printf` 不同的是 Haskell 这个版本会在运行时检查参数的个数以及其类型。

```
ghci> :m +Text.Printf
ghci> printf "%d and %d\n" (3::Int)
3 and *** Exception: Printf.printf: argument list ended prematurely
ghci> printf "%s and %d\n" "foo" (3::Int)
foo and 3
```

在 **ghci** 里试试 `printf "%d" True`，看看会得到什么结果。

支持 `main` 函数的是几个短小的函数。

```
-- file: ch28/Check.hs
```

(continues on next page)

(continued from previous page)

```
modifyTVar_ :: TVar a -> (a -> a) -> STM ()
modifyTVar_ tv f = readTVar tv >>= writeTVar tv . f

forkTimes :: Int -> TVar Int -> IO () -> IO ()
forkTimes k alive act =
  replicateM_ k . forkIO $
    act
    `finally`
    (atomically $ modifyTVar_ alive (subtract 1))
```

forkTimes 函数新建特定数量的相同的工作线程，每当一个线程推出时，则”活动”线程的计数器相应的减一。我们使用 finally 组合器确保无论线程是如何终止的，都会减少”活动”线程的数量。

下一步，writeBadLinks 会把每个失效或者死亡 (dead) 的链接打印到 stdout 。

```
-- file: ch28/Check.hs
writeBadLinks :: TChan String -> IO ()
writeBadLinks c =
  forever $
    atomically (readTChan c) >>= putStrLn >> hFlush stdout
```

上面我们使用了 forever 组合器使一个操作永远的执行。

```
ghci> :m +Control.Monad
ghci> :type forever
forever :: (Monad m) => m a -> m ()
```

waitFor 函数使用了 check ，当它的参数是 False 时会调用 retry 。

```
-- file: ch28/Check.hs
waitFor :: TVar Int -> IO ()
waitFor alive = atomically $ do
  count <- readTVar alive
  check (count == 0)
```

27.8.1 检查一个链接

这个原生的函数实现了如何检查一个链接的状态。代码和 [第二十二章 Chapter 22, Extended Example: Web Client Programming] 里的 podcatcher 相似但有一点不同。

```
-- file: ch28/Check.hs
getStatus :: URI -> IO (Either String Int)
getStatus = chase (5 :: Int)
```

(continues on next page)

(continued from previous page)

```
  where
    chase 0 _ = bail "too many redirects"
    chase n u = do
      resp <- getHead u
      case resp of
        Left err -> bail (show err)
        Right r ->
          case rspCode r of
            (3,_,_) ->
               case findHeader HdrLocation r of
                 Nothing -> bail (show r)
                 Just u' ->
                   case parseURI u' of
                     Nothing -> bail "bad URL"
                     Just url -> chase (n-1) url
            (a,b,c) -> return . Right $ a * 100 + b * 10 + c

    bail = return . Left

getHead :: URI -> IO (Result Response)
getHead uri = simpleHTTP Request { rqURI = uri,
                                   rqMethod = HEAD,
                                   rqHeaders = [],
                                   rqBody = "" }
```

为了避免无尽的重定向相应，我们只允许固定次数的重定向请求。我们通过查看 HTTP 标准 HEAD 信息来确认链接的有效性，比起一个完整的 GET 请求，这样做可以减少网络流量。

这个代码是典型的”marching off the left of the screen”风格。正如之前我们提到的，需要谨慎使用这样的风格。下面我们用 `ErrorT` monad transformer 和几个通用一点的方法进行了重新实现，它看上去简洁了很多。

```
-- file: ch28/Check.hs
getStatusE = runErrorT . chase (5 :: Int)
  where
    chase :: Int -> URI -> ErrorT String IO Int
    chase 0 _ = throwError "too many redirects"
    chase n u = do
      r <- embedEither show =<< liftIO (getHead u)
      case rspCode r of
        (3,_,_) -> do
            u'  <- embedMaybe (show r)  $ findHeader HdrLocation r
            url <- embedMaybe "bad URL" $ parseURI u'
            chase (n-1) url
        (a,b,c) -> return $ a*100 + b*10 + c
```

(continues on next page)

(continued from previous page)

```
-- Some handy embedding functions.
embedEither :: (MonadError e m) => (s -> e) -> Either s a -> m a
embedEither f = either (throwError . f) return

embedMaybe :: (MonadError e m) => e -> Maybe a -> m a
embedMaybe err = maybe (throwError err) return
```

27.8.2 工作者线程

每个工作者线程 (Worker Thread) 从一个共享队列里拿一个任务，这个任务要么检查链接有效性，要么让线程推出。

```
-- file: ch28/Check.hs
worker :: TChan String -> TChan Task -> TVar Int -> IO ()
worker badLinks jobQueue badCount = loop
  where
    -- Consume jobs until we are told to exit.
    loop = do
        job <- atomically $ readTChan jobQueue
        case job of
            Done  -> return ()
            Check x -> checkOne (B.unpack x) >> loop

    -- Check a single link.
    checkOne url = case parseURI url of
        Just uri -> do
            code <- getStatus uri `catch` (return . Left . show)
            case code of
                Right 200 -> return ()
                Right n   -> report (show n)
                Left err  -> report err
        _ -> report "invalid URL"

        where report s = atomically $ do
                           modifyTVar_ badCount (+1)
                           writeTChan badLinks (url ++ " " ++ s)
```

27.8.3 查找链接

我们构造了基于 IO monad 的状态 monad transformer 栈用于查找链接。这个状态会记录我们已经找到过的链接 (避免重复)、链接的数量以及一个队列，我们会把需要做检查的链接放到这个队列里。

```haskell
-- file: ch28/Check.hs
data JobState = JobState { linksSeen :: S.Set URL,
                           linksFound :: Int,
                           linkQueue :: TChan Task }

newtype Job a = Job { runJob :: StateT JobState IO a }
    deriving (Monad, MonadState JobState, MonadIO)

execJob :: Job a -> JobState -> IO JobState
execJob = execStateT . runJob
```

严格来说，对于对立运行的小型程序，我们并不需要用到 newtype ，然后我们还是将它作为一个好的编码实践的例子放在这里。(毕竟也只多了几行代码)

main 函数实现了对每个输入文件调用一次 checkURLs 方法，所以 checkURLs 的参数就是单个文件。

```haskell
-- file: ch28/Check.hs
checkURLs :: FilePath -> Job ()
checkURLs f = do
    src <- liftIO $ B.readFile f
    let urls = extractLinks src
    filterM seenURI urls >>= sendJobs
    updateStats (length urls)

updateStats :: Int -> Job ()
updateStats a = modify $ \s ->
    s { linksFound = linksFound s + a }

-- | Add a link to the set we have seen.
insertURI :: URL -> Job ()
insertURI c = modify $ \s ->
    s { linksSeen = S.insert c (linksSeen s) }

-- | If we have seen a link, return False.  Otherwise, record that we
-- have seen it, and return True.
seenURI :: URL -> Job Bool
seenURI url = do
    seen <- (not . S.member url) `liftM` gets linksSeen
    insertURI url
    return seen

sendJobs :: [URL] -> Job ()
sendJobs js = do
    c <- gets linkQueue
    liftIO . atomically $ mapM_ (writeTChan c . Check) js
```

extractLinks 函数并没有尝试去准确的去解析一个 HTMP 或是文本文件，而只是匹配那些看上去像 URL 的字符串。我们认为这样做就够了。

```
-- file: ch28/Check.hs
extractLinks :: B.ByteString -> [URL]
extractLinks = concatMap uris . B.lines
  where uris s      = filter looksOkay (B.splitWith isDelim s)
        isDelim c   = isControl c || c `elem` " <>\"{}|\\^[]`"
        looksOkay s = http `B.isPrefixOf` s
        http        = B.pack "http:"
```

27.8.4 命令行的实现

我们使用了 System.Console.GetOpt 模块来解析命令行参数。这个模块提供了很多解析命令行参数的很有用的方法，不过使用起来稍微有点繁琐。

```
-- file: ch28/Check.hs
data Flag = Help | N Int
            deriving Eq

parseArgs :: IO ([String], Int)
parseArgs = do
    argv <- getArgs
    case parse argv of
        ([], files, [])                    -> return (nub files, 16)
        (opts, files, [])
            | Help `elem` opts             -> help
            | [N n] <- filter (/=Help) opts -> return (nub files, n)
        (_,_,errs)                         -> die errs
  where
    parse argv = getOpt Permute options argv
    header     = "Usage: urlcheck [-h] [-n n] [file ...]"
    info       = usageInfo header options
    dump       = hPutStrLn stderr
    die errs   = dump (concat errs ++ info) >> exitWith (ExitFailure 1)
    help       = dump info                  >> exitWith ExitSuccess
```

getOpt 函数接受三个参数

- 参数顺序的定义。它定义了选项 (Option) 是否可以和其他参数混淆使用 (就是我们上面用到的 Permute) 或者是选项必须出现在参数之前。
- 选项的定义。每个选项有这四个部分组成：简称，全称，选项的描述 (比如是否接受参数) 以及用户说明。

- 参数和选项数组，类似于 `getArgs` 的返回值。

这个函数返回一个三元组，包括用户输入的选项，参数以及错误信息 (如果有的话)。

我们使用 `Flag` 代数类型 (Algebraic Data Type) 表示程序所能接收的选项。

```
-- file: ch28/Check.hs
options :: [OptDescr Flag]
options = [ Option ['h'] ["help"] (NoArg Help)
                  "Show this help message",
            Option ['n'] []       (ReqArg (\s -> N (read s)) "N")
                  "Number of concurrent connections (default 16)" ]
```

`options` 列表保存了每个程序能接收选项的描述。每个描述必须要生成一个 `Flag` 值。参考上面例子中是如何使用 `NoArg` 和 `ReqArg` 。`GetOpt` 模块的 `ArgDescr` 类型有很多构造函数 (Constructors)。

```
-- file: ch28/GetOpt.hs
data ArgDescr a = NoArg a
                | ReqArg (String -> a) String
                | OptArg (Maybe String -> a) String
```

- `NoArg` 接受一个参数用来表示这个选项。在我们这个例子中，如果用户在调用程序时输入 `-h` 或者 `--help` , 我们就用 `Help` 值表示。
- `ReqArg` 的第一个函数作为参数，这个函数把用户输入的参数转化成相应的值；第二个参数是用来说明的。这里我们是将字符串转换为数值 (integer)，然后再给类型 `Flag` 的构造函数 `N` 。
- `OptArg` 和 `ReqArg` 很相似，但它允许选项没有对应的参数。

27.8.5 模式守卫 (Pattern guards)

函数 `parseArgs` 的定义里其实潜在了一个语言扩展 (Language Extension), Pattern guards。用它可以写出更加简要的 guard expressions. 它通过语言扩展 `PatternGuards` 来使用。

一个 Pattern Guard 有三个组成部分：一个模式 (Pattern)，一个 `<-` 符号以及一个表达式。表达式会被解释然后和模式相匹配。如果成功，在模式中定义的变量会被赋值。我们可以在一个 guard 里同时使用 pattern guards 和普通的 `Bool` guard expressions。

```
-- file: ch28/PatternGuard.hs
{-# LANGUAGE PatternGuards #-}

testme x xs | Just y <- lookup x xs, y > 3 = y
            | otherwise                    = 0
```

在上面的例子中，当关键字 `x` 存在于 alist `xs` 并且大于等于 3，则返回它所对应的值。下面的定义实现了同样的功能。

```haskell
-- file: ch28/PatternGuard.hs
testme_noguards x xs = case lookup x xs of
                         Just y | y > 3 -> y
                         _              -> 0
```

Pattern guards 使得我们可以把一系列的 guards 和 `case` 表达式组合到单个 guard，从而写出更加简洁并容易理解的 guards。

27.9 STM 的实践意义

至此我们还并未提及 STM 所提供的特别优越的地方。比如它在做组合 (*composes*) 方面就表现的很好：当需要向一个事务中增加逻辑时，只需要用到常见的函数 `(>>=)` 和 `(>>)`。

组合的概念在构建模块化软件是显得格外重要。如果我们把俩段都没有问题的代码组合在一起，也应该是能很好工作的。常规的线程编程技术无法实现组合，然而由于 STM 提供了一些很关键的前提，从而使在线程编程时使用组合变得可能。

STM monad 防止了我们意外的非事务性的 I/O。我们不再需要关心锁的顺序，因为代码里根本没有锁机制。我们可以忘记丢失唤醒，因为不再有条件变量了。如果有异常发生，我们则可以用函数 `catchSTM` 捕捉到，或者是往上级传递。最后，我们可以用 `retry` 和 `orElse` 以更加漂亮的方式组织代码。

采用 STM 机制的代码不会死锁，但是导致饥饿还是有可能的。一个长事务导致另外一个事务不停的 `retry`。为了解决这样的问题，需要尽量的短事务并保持数据一致性。

27.9.1 合理的放弃控制权

无论是同步管理还是内存管理，经常会遇到保留控制权的情况：一些软件需要对延时或是内存使用记录有很强的保证，因此就必须花很多时间和精力去管理和调试显式的代码。然后对于软件的大多数实际情况，垃圾回收 (Garbage Collection) 和 STM 已经做的足够好了。

STM 并不是一颗完美的灵丹妙药。当我们选择垃圾回收而不是显式的内存管理，我们是放弃了控制权从而获得更加安全的代码。同样的，当使用 STM 时，我们放弃了底层的细节，从而希望代码可读性更好，更加容易理解。

27.9.2 使用不变量

STM 并不能消除某些类型的 bug。比如，我们在一个 `atomically` 事务中从某个账号中取钱，然后返回到 `IO` monad，然后在另一个 `atomically` 事务中把钱存到另一个账号，那么代码就会产生不一致性，因为会在某个特定时刻，这部分钱不会出现的任意一个账号里。

```haskell
-- file: ch28/GameInventory.hs
bogusTransfer qty fromBal toBal = do
  fromQty <- atomically $ readTVar fromBal
  -- window of inconsistency
  toQty   <- atomically $ readTVar toBal
  atomically $ writeTVar fromBal (fromQty - qty)
  -- window of inconsistency
  atomically $ writeTVar toBal   (toQty + qty)

bogusSale :: Item -> Gold -> Player -> Player -> IO ()
bogusSale item price buyer seller = do
  atomically $ giveItem item (inventory seller) (inventory buyer)
  bogusTransfer price (balance buyer) (balance seller)
```

在同步程序中，这类问题显然很难而且不容易重现。比如上述例子中的不一致性问题通常只存在一段很短的时间内。在开发阶段通常不会出现这类问题，而往往只有在负载很高的产品环境才有可能发生。

我们可以用函数 `alwaysSucceeds` 定义一个不变量，它是永远为真的一个数据属性。

```
ghci> :type alwaysSucceeds
alwaysSucceeds :: STM a -> STM ()
```

当创建一个不变量时，它马上会被检查。如果要失败，那么这个不变量会抛出异常。更有意思的是，不变量会在经后每个事务完成时自动被检查。如果在任何一个点上失败，事务就会推出，不变量抛出的异常也会被传递下去。这就意味着当不变量的条件被违反时，我们就可以马上得到反馈。

比如，下面两个函数给本章开始时定义的游戏世界增加玩家

```haskell
-- file: ch28/GameInventory.hs
newPlayer :: Gold -> HitPoint -> [Item] -> STM Player
newPlayer balance health inventory =
    Player `liftM` newTVar balance
           `ap` newTVar health
           `ap` newTVar inventory

populateWorld :: STM [Player]
populateWorld = sequence [ newPlayer 20 20 [Wand, Banjo],
                           newPlayer 10 12 [Scroll] ]
```

下面的函数则返回了一个不变量，通过它我们可以保证整个游戏世界资金总是平衡的：即任何时候的资金总量和游戏建立时的总量是一样的。

```haskell
-- file: ch28/GameInventory.hs
consistentBalance :: [Player] -> STM (STM ())
consistentBalance players = do
```

(continues on next page)

(continued from previous page)

```
    initialTotal <- totalBalance
    return $ do
      curTotal <- totalBalance
      when (curTotal /= initialTotal) $
        error "inconsistent global balance"
  where totalBalance   = foldM addBalance 0 players
        addBalance a b = (a+) `liftM` readTVar (balance b)
```

下面我们写个函数来试验下。

```
-- file: ch28/GameInventory.hs
tryBogusSale = do
  players@(alice:bob:_) <- atomically populateWorld
  atomically $ alwaysSucceeds =<< consistentBalance players
  bogusSale Wand 5 alice bob
```

由于在函数 bogusTransfer 中不正确地使用了 atomically 而会导致不一致性，当我们在 **ghci** 里运行这个方法时则会检测到这个不一致性。

```
ghci> tryBogusSale
*** Exception: inconsistent global balance
```

CHAPTER

TWENTYEIGHT

翻译约定

28.1 翻译约定

28.1.1 第二章

强类型 strong type

静态类型 static type

自动推导 automatically infer

类型正确 well type

类型不正确 ill type

类型推导 type inference

列表 list

元组 tuple

表达式 expression

陈述 statement

分支 branch

严格求值 strict evaluation

非严格求值 non-strict evaluation

惰性求值 lazy evaluation

块 chunk

代换 substitution

28.1.2 第三章

类构造器 type constructor

值构造器 value constructor

类型别名 type synonym

代数数据类型 algebraic data type

备选 alternative

分支 case

复合数据/复合值 compound value

枚举类型 enumeration type

解构 deconstruction

字面 literal

结构递归 structural recursion

递归情形 recursive case

基本情形 base case

高阶 high-order

公式化 boilerplate

样板代码 boilerplate code

访问器函数 accessor function

28.1.3 第四章

折叠 fold

收集器 collection

主递归 primitive recursive

部分应用 partial application

部分函数 partial function

全函数 total function

部分函数应用 partial function application

柯里化 currying

组合函数 composition

内存泄漏 space leak

严格 strict

非严格 non-strict

28.1.4 第五章

导出 export

本地码 navie code

目标代码 object code

指令 directive

顶层 top-level

28.1.5 第六章

通用函数 generic function

部分有序 particular ordering

编译选项 pragma

重叠实例 overlapping instances

身份 identity

单一同态 monomorphism

28.1.6 第八章

字符类 character class

多态 polymorphism

28.1.7 第十五章

提升 lift

类型签名 type signature

锯齿化 staircasing

捆绑表达式 bind expression

短路 short circuit

拆分函数 split function

单线 one-liner

多参数类型类 MultiParamTypeClasses

超类 superclass

功能依赖 functional dependency

28.1.8 第十八章

monad 变换器 monad transformer

monad 栈 monad transformer stack / monad stack

下层 monad underlying monad

派生 derive / deriving

类型类 typeclass

抬举 lift

28.1.9 第十九章

错误处理 error handling

惰性求值 lazy evaluation

28.1.10 第二十四章

求值策略 evaluation strategy

28.1.11 第二十五章

性能剖析 profiling

消耗集中点 cost centre

热点 hot spot

内存分配 allocations

编译指示 pragma

顶层函数 top level function

标志 flag

节点 node

核心语言 Core

未封装数据类型 unboxed data types

基本数据类型 primitive data types

范式 normal form

弱首范式 weak head normal form

融合 fusion

28.1.12 第二十六章

布隆过滤器 Bloom Filter

网络流量整形 traffic shaper

可变引用 mutable references

冻结 frozen

28.1.13 第二十八章

软件事务内存 Software transactional memory

一致性 consistent

条件变量 condition variable

条件竞争 race condition

死锁 deadlock

程序崩溃 corruption

细粒度 finer-grained

簿记 book-keeping

CHAPTER

TWENTYNINE

关于

以下人员参与了本文档的翻译工作：

- huangz
- Haisheng, Wu
- Albert
- Guang Yang
- labyrlnth
- Javran Cheng
- bladewang
- spectator
- tiancaiamao
- Yao Lu
- YenvY
- Lvwenlong
- sancao2
- Damon Zhao
- tianws
- Hugo Wang
- xnnyygn
- Allan Zyne
- Forec

除了进行翻译之外，本文档还在原书的基础上做了以下改进：

- 修正原文正文和代码中的错误
- 更新代码以符合最新的 Haskell 规范
- 在一些比较复杂的地方添加注释，帮助理解

关注项目进度 / 反馈意见或建议 / 提交你的翻译贡献，请访问项目的 github 页面。

本文档的部分内容参考了 AlbertLee 的译本，在此对他表示感谢。

CHAPTER

THIRTY

版权

BIBLIOGRAPHY

[Hughes95] John Hughes. "The design of a pretty-printing library" . May, 1995. First International Spring School on Advanced Functional Programming Techniques. Bastad, Sweden. .

[Wadler98] Philip Wadler. "A prettier printer" . March 1998.

[Broder02] Andrei Broder. Michael Mitzenmacher. "Network applications of Bloom filters: a survey" . Internet Mathematics. 1. 4. 2005. 485-509. A K Peters Ltd..