

Real World Haskell 中文版

看云文档小组



目 录

Real World Haskell 中文版

第一章：入门

第二章：类型和函数

第三章：Defining Types, Streamlining Functions

第四章：函数式编程

第五章：编写 JSON 库

第六章：类型类

第七章：I/O

第八章：高效文件处理、正则表达式、文件名匹配

第九章：I/O学习 —— 构建一个用于搜索文件系统的库

第十章：代码案例学习：解析二进制数据格式

第十一章：测试和质量保障

第十三章：数据结构

第十八章：Monad变换器

第十九章：错误处理

第二十章：使用 Haskell 进行系统编程

第二十一章：数据库的使用

第二十二章：扩展示例 —— Web 客户端编程

第二十七章：Socket 和 Syslog

第二十八章：软件事务内存 (STM)

翻译约定

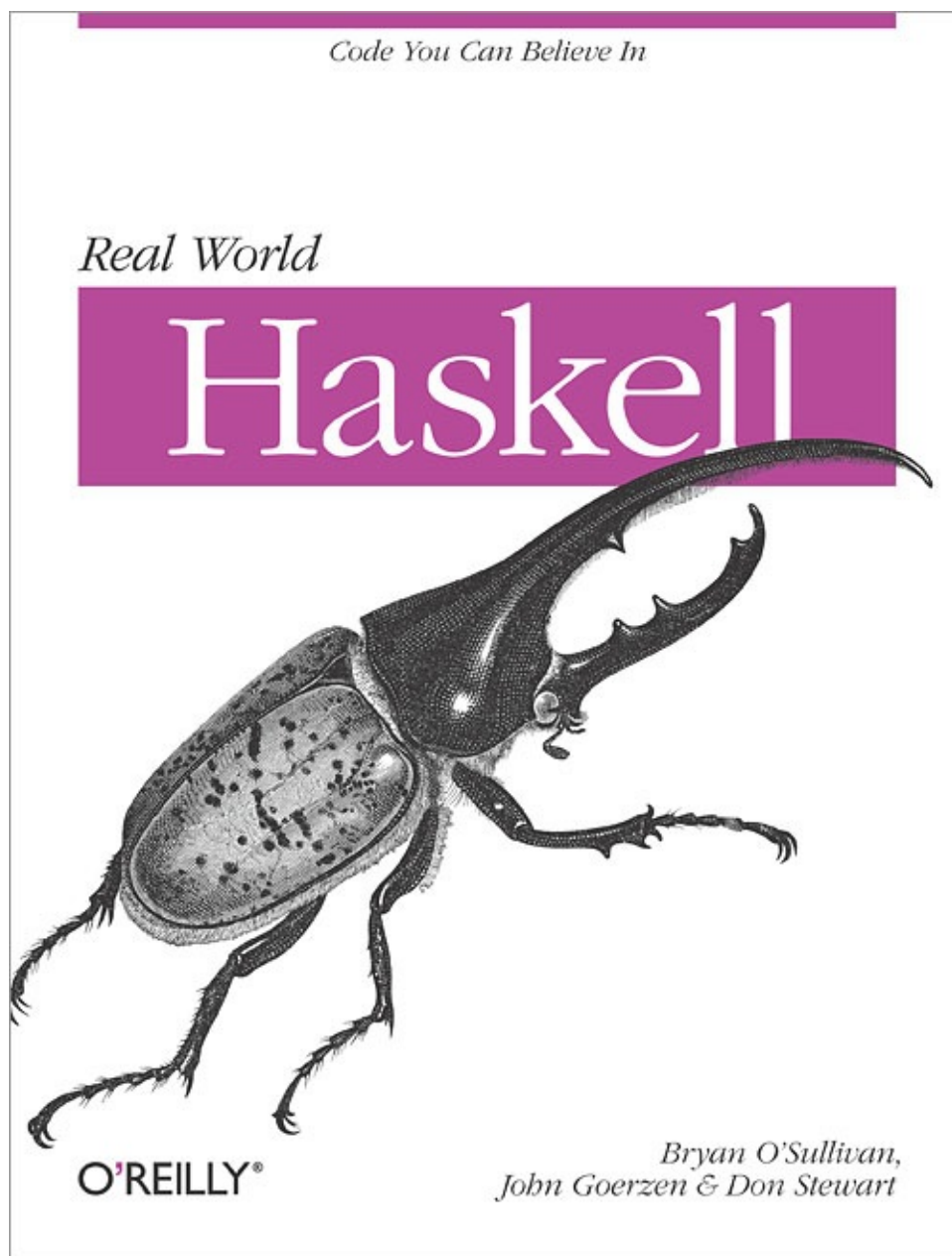
Real World Haskell 中文版

Real World Haskell 中文版

Warning

《Real World Haskell》中文版现在可以通过域名 cnhaskell.com 快速进行访问，请各位读者使用新域名访问本文档，原有的域名 rwh.rtfld.org 以及 rwh.readthedocs.org 将在一段时间之后被废弃。

2015 年 8 月 30 宣



本文档是 [Real World Haskell](https://github.com/huangz1990/real-world-haskell-cn) 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>

关于

以下人员参与了本文档的翻译工作：

- [huangz](#)
- [Haisheng, Wu](#)
- [Albert](#)
- [Guang Yang](#)

本文档使用 [看云](#) 构建

- [labyrinth](#)
- [Javran Cheng](#)
- [bladewang](#)
- [spectator](#)
- [tiancaimao](#)

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。

版权

本文档和原书一样，通过 CC 协议进行[署名-非商业性使用](#) 授权。

第一章：入门

第一章：入门 Haskell编程环境

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

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

GHC主要有三个部分组成。

- ghc是生成快速本底代码的优化编译器。
- ghci是一个交互解析器和调试器。
- runghc是一个以脚本形式(并不要首先编译)运行Haskell代码的程序，

Note

我们如何称呼GHC的各个组件

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

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

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

我们在[附录A]中提供了更多详细的信息介绍如何在各个流行平台上安装GHC。

初识解释器ghci

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

```
The ghci command has a narrow focus
We typically can not copy some code out of a haskell source file and paste it into ghci. This does not have a significant effect on debugging 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WindowsXP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 standardprelude)，因为它的内容是基于Haskell 98标准定义的。通常简称它为“序幕” (theprelude)。

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 `:setprompt` 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。基本交互: 把ghci当作一个计算器 除了能提供测试代码片段的交互功能外，ghci也可以被当作一个桌面计算器来使用。我们可以很容易的表示基本运算，同时随着对Haskell了解的深入，也可以表示更加复杂的运算。即使是以如此简单的方式来使用这个解析器，也可以帮助我们了解更多关于Haskell是如何工作的。基本算术运算 我们可以马上开始输入一些表达式，看看ghci会怎么处理它们。基本的算术表达式类似于像C或是Python这样的语言：用中缀表达式，即操作符在操作数之间。

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

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


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

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

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

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`f3`。如果我们不用括号括起一个负数，就会有非常明显的不同的方式理解`f-3`：它可以是“将函数`f`应用(`apply`)与数字`-3`”，或者是“把变量`f`减去`3`”。

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

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

下面的例子和上面有问题的负数的例子很像，然而它的错误信息并不一样。

```
ghci> 2*-3
<interactive>:1:1: Not in scope: `*-`
```

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

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

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

布尔逻辑，运算符以及值比较

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

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

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

我们再一次的遇到了很有前瞻性的错误。简单来说，错误信息告诉我们布尔类型，Bool，不是数字类型，Num的一个成员。错误信息有些长，这是因为ghci会定位出错的具体位置，并且给出了也许能解决问题的修改提示。错误信息详细分析如下。“No instance for (Num Bool)”告诉我们ghci尝试解析数字1为Bool类型但是失败。“arising from the literal 1'”表示是由于使用了数字1而引发了问题。“In the definition of it”引用了一个ghci的快捷方式。我们会在后面提到。

Tip 遇到错误信息不要胆怯 这里我们提到了很重要的一点，而且在本书的前面一些章节中我们会重复提到。如果你碰到一些你从来没遇到过的问题和错误信息，别担心(panic)。刚开始的时候，你所要的做的仅仅是找出足够的信息来帮助解决问题。随着你经验的积累，你会发现错误信息中的一部分其实很容易理解，并不会像刚开始时那么晦涩难懂。各种错误信息都有一个目的：通过提前的一些调试，帮助我们在真正运行程序之前能书写出正确的代码。如果你曾使用过其它更加宽松(permissive)的语言，这种方式可能会有些震惊(shock).所以，拿出你的耐心来。Haskell中大多数比较操作符和C语言以及受C语言影响的语言类似。

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

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

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

```
ghci> not True
False
```

运算符优先级以及结合性

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

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

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

```
ghci> :info (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
  -- 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+44和1+(44)值相同而不是(1+4)4。

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

一个右结合的操作符会以infixr表示。

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

优先级和结合性规则的组合通常称之为固定性(fixity)规则。

未定义的变量以及定义变量

Haskell的标准库prelude定义了至少一个大家熟知的数学常量。

```
ghci> pi
3.141592653589793
```

然后我们很快就会发现它对数学常量的覆盖并不是很广泛。让我们来看下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，它所表达的只是没有用e这个名字定义过变量。

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

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

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

既然e已经定义好了，我们就可以在数学表达式中使用它。我们之前用到的乘方操作符(^)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。

处理优先级以及结合性规则

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

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

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

ghci里的命令行编辑

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

[译者注：]制表符的完成功能其实在Windows下也是可以的。

Tip

哪里可以找到更多信息

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

在类Unix系统下，ghci使用功能强大并且可定制化的[GNU readline library](#)。在Windows系

统下，ghci的命令行编辑功能是由[doskeycommand](#) 提供的。

列表(Lists)

一个列表由方括号以及被逗号分隔的元素组成。

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

Note

逗号是分隔符，不是终结符

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

列表可以是任意长度。空列表表示成[]。

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

列表里所有的元素必须是相同类型。下面例子我们违反了这条规则：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”(upperbound)，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到在Haskell中无穷数列经常会用到。

Note

列举浮点数时要注意的

下面的例子看上并不那么直观

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

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

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

列表的操作符

有两个常见的用于列表的操作符。连接两个列表时使用(++)。

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

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

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

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

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，字符串和转意规则](#)。

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

函数putStrLn用于打印一个字符串。

Haskell区分单个字符和文本字符串。单个字符用单引号包含。

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

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

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

""表示空字符串，它和[]同义。

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

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

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

初识类型

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

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

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

```
Prelude> :set +t

Prelude> 'c'      -- 输入表达式
'c'              -- 输出值
it :: Char       -- 输出值的类型

Prelude> "foo"
"foo"
it :: [Char]
```

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

Ghci 打印的类型信息可以分为几个部分：

- 它打印出 it
- $x::y$ 表示表达式 x 的类型为 y
- 第二个表达式的值的类型为 [Char]。（类型 String 是 [Char]的一个别名，它通常用于代替 [Char]。）

以下是另一个我们已经见过的类型：

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

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

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

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

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

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

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

```
Prelude Data.Ratio> 3.14 % 8

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

Prelude Data.Ratio> 1.2 % 3.4

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

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

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

Prelude Data.Ratio> 2
2
```

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

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

Prelude Data.Ratio> "foo"
"foo"

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

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

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

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

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

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

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

第六章会介绍更多类型签名的相关信息。

行计数程序

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

使用编辑器，输入以下内容，并将它保存为 `WC.hs`：

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

main = interact wordCount
```

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

再创建一个 quux.txt ，包含以下内容：

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

然后，在 shell 执行以下代码：

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

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

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

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
- (+)24
- sqrt16
- succ6
- succ7
- pred9
- pred8
- sin(pi/2)

- truncatepi
- round3.5
- round3.4
- floor3.7
- ceiling3.3

1. 在ghci里输入:?以或许帮助信息。定义一个变量，比如letx=1,然后输入:showbindings.你看到了什么？
2. 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。
3. 再次修改WC.hs，可以输出一个文件的字符个数。

第二章：类型和函数

第二章：类型和函数 类型是干什么用的？

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

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

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

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

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

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

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

Haskell 的类型系统

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

后面的三个小节会分别讨论这三个方面，介绍它们的长处和短处，并列举 Haskell 类型系统

的概念和其他语言里相关构思之间的相似性。

强类型

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

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

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

要在 Haskell 中进行类型转换，必须显式地使用类型转换函数。

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

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

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

静态类型

静态类型系统指的是，编译器可以在编译期（而不是执行期）知道每个值和表达式的类型。Haskell 编译器或解释器会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。

类型推导 关于类型系统，最后要说的是，Haskell 编译器可以自动推断出程序中几乎所有表达式的类型[注：有时候要提供一些信息，帮助编译器理解程序代码]。这个过程被称为类型推导（type inference）。虽然 Haskell 允许我们显式地为任何值指定类型，但类型推导使得这种工作通常是可选的，而不是非做不可的事。

正确理解类型系统 对 Haskell 类型系统能力和好处的探索会花费好几个章节。在刚开始的时候，处理 Haskell 的类型可能会让你觉得有些麻烦。比如说，在 Python 和 Ruby 里，你只要写下程序，然后测试一下程序的执行结果是否正确就够了，但是在 Haskell，你还要先确保程序能通过类型检查。那么，为什么要多走这些弯路呢？答案是，静态、强类型检查使得 Haskell 更安全，而类型推导则让它更精炼、简洁。这样得出的结果是，比起其他流行的静态语言，Haskell 要来得更安全，而比起其他流行的动态语言，Haskell 的表现力又更胜一筹。这并不是吹牛，等你看完这本书之后就会了解这一点。修复编译时的类型错误刚开始会让人觉得增加了不必要的工作量，但是，换个角度来看，这不过是提前完成了调试工作：编译器在处理程序时，会将代码中的逻辑错误一一展示出来，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。

一些常用的基本类型 以下是 Haskell 里最常用的一些基本类型，其中有些在之前的章节里已经看过了：Char 单个 Unicode 字符。Bool 表示一个布尔逻辑值。这个类型只有两个值：True 和 False。Int 带符号的定长（fixed-width）整数。这个值的准确范围由机器决定：在 32 位机器里，Int 为 32 位宽，在 64 位机器里，Int 为 64 位宽。Haskell 保证 Int 的宽度不少于 28 位。（数值类型还可以是 8 位、16 位，等等，也可以是带符号和无符号的，以后会介绍。）Integer 不限长度的带符号整数。Integer 并不像 Int 那么常用，因为它们需要更多的内存和更大的计算量。另一方面，对 Integer 的计算不会造成溢出，因此使用 Integer 的计算结果更可靠。Double 用于表示浮点数。长度由机器决定，通常是 64 位。（Haskell 也有 Float 类型，但是并不推荐使用，因为编译器都是针对 Double 来进行优化的，而 Float 类型值的计算要慢得多。）在前面的章节里，我们已经见到过 :: 符号。除了用来表示类型之外，它还可以用于进行类型签名。比如说，`exp :: T` 就是向 Haskell 表示，`exp` 的类型是 `T`，而 `:: T` 就是表达式 `exp` 的类型签名。如果一个表达式没有显式地指名类型的话，那么它的类型就通过自动推导来决定：

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

Prelude> 'a'           -- 自动推导
'a'

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

当然了，类型签名必须正确，否则 Haskell 编译器就会产生错误：

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

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

调用函数

本文档使用 [看云](#) 构建

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 `sqrt3` 和 `sqrt6` 的计算结果分别传给 `compare` 函数。如果将括号移走，Haskell 编译器就会产生一个编译错误，因为它认为我们将四个参数传给了只需要两个参数

的 `compare` 函数：

```
Prelude> compare sqrt 3 sqrt 6

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

复合数据类型：列表和元组

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

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

`head` 函数取出列表的第一个元素：

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

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

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

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

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

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

Prelude> tail [True, False]
[False]

Prelude> tail "list"
"ist"

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

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

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

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

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

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

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

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

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

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

元组的两边用括号包围，元素之间用逗号分割。元组的类型信息也使用同样的格式：

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

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

```
(4, "am", (16, True))
```

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

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

通常用元组中元素的数量作为称呼元组的前缀，比如“2-元组”用于称呼包含两个元素的元组，“5-元组”用于称呼包含五个元素的元组，诸如此类。Haskell 不能创建 1-元组，因为 Haskell 没有相应的创建 1-元组的语法（*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。

处理列表和元组的函数

前面的内容介绍了如何构造列表和元组，现在来看看处理这两种数据结构的函数。

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

`take` 返回一个包含 `l` 前 `n` 个元素的列表：

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

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

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

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

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

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

将表达式传给函数

Haskell 的函数应用是左关联的。比如说，表达式 `abcd` 等同于 `((ab)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'
```

`drop4"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"
```

函数类型

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

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

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

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

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

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

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

纯度

副作用指的是，函数的行为受系统的全局状态所影响。

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

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

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

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

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

Haskell 源码，以及简单函数的定义

既然我们已经学会了如何应用函数，那么是时候回过头来，学习怎样去编写函数。

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

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

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

[译注：原书代码里的路径为 ch03/add.hs，是错误的。]

= 号左边的 addab 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。）

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 `printx` 改为 `print(x)`，确保代码在 Python 2 和 Python 3 都可以顺利执行。]

然后，试着在 Haskell 里做同样的事：

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

但是 Haskell 并不允许做这样的多次赋值：

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

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

条件求值

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

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

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

Prelude> drop 4 "foobar"
"ar"

Prelude> drop 4 [1, 2]
[]

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，而不是新建一个。所以，不要省略缩进！

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

先保存文件，试试 `myDrop` 函数是否如我们所预期的那样工作：

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

*Main> myDrop 2 "foobar"
"obar"

*Main> myDrop 4 "foobar"
"ar"

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

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

*Main> myDrop 7 []
[]

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

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

if 关键字引入了一个带有三个部分的表达式：

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

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

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

<interactive>:2:14:
  No instance for (Num [Char])
    arising from the literal `1'
  Possible fix: add an instance declaration for (Num [Char])
  In the expression: 1
  In the expression: if True then 1 else "foo"
  In an equation for `it': it = if True then 1 else "foo"
```

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

程序里还有几个新东西需要解释。其中，`null` 函数检查一个列表是否为空：

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

Prelude> null []
True

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

```
False
```

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

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

Prelude> True || False
True

Prelude> True || True
True
```

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

最后，整个 if 表达式被分成了多行，而实际上，它也可以写成一行：

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

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

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

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

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

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

```
def myDrop(n, elts):
    while n > 0 and elts:
```

```
n = n - 1
elts = elts[1:]
return elts
```

通过示例了解求值

前面对 myDrop 的描述关注的都是表面上的特性。我们需要更进一步，开发一个关于函数是如何被应用的心智模型：为此，我们先从一些简单的示例出发，逐步深入，直到搞清楚 myDrop2"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。

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，mod32 返回 1，而 1==1 返回 True。

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

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

非严格求值通常也被称为惰性求值。[注：实际上，“非严格”和“惰性”在技术上有些细微

的差别，但这里不讨论这些细节。]

一个更复杂的例子

现在，将注意力放回 `myDrop2"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 `myDrop2"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")`。

递归

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

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

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

对 (`||`) 的左操作符的求值过程如下：

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

*Main> 2 - 1
1

*Main> 1 <= 0
False
```

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

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

*Main> tail "abcd"
"bcd"

*Main> null "bcd"
```

```
False
```

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

终止递归

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

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

再次对 `(||)` 操作符的右操作对象求值：

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

最终，我们得出了一个 `True` 值！

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

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

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

从递归中返回

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

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

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

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

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

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

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

最终计算出结果 `"cd"`：

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

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

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

学到了什么？

这一节介绍了三个重要的知识点：

- 可以通过代换（substitution）和重写（rewriting）去了解 Haskell 求值表达式的方式。
- 惰性求值可以延迟计算直到真正需要一个值为止，并且在求值时，也只执行可以给出（establish）值的那部分表达式。[译注：比如之前提到的，`(||)` 的左操作符的值为 `True` 时的情况。]
- 函数的返回值可能是一个块（一个被延迟计算的表达式）。

Haskell 里的多态

之前介绍列表的时候提到过，列表是类型多态的，这一节会说明更多这方面的细节。

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

型的列表都可以产生同样的效果：

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

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

last 的秘密就隐藏在类型签名里面：

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

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

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

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

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

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

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

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

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

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

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

关于多态还有很多东西要说，本书第六章会再次回到这个主题。

对多态函数进行推理

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

以 fst 函数为例子：

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

首先，函数签名包含两个类型变量 a 和 b，表明元组可以包含不同类型的值。

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

延伸阅读

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

多参数函数的类型

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

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

通过类型签名可以看到，`take` 函数和一个 `Int` 值以及两个列表有关。类型签名中的 `->` 符号是右关联的：Haskell 从右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]
```

为什么要对纯度斤斤计较？

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

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

以 `not` 函数为例子：

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

即使抛开函数名不说，单单函数签名就极大地限制了这个函数可能有的合法行为：

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

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

通过这种将不纯函数隔离、并尽可能简单化的编程风格，程序的漏洞将变得非常少。

回顾

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

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

第三章：Defining Types, Streamlining Functions

第三章：Defining Types, Streamlining Functions 定义新的数据类型

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

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

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

本章将以一个在线书店为例子，展示如何去进行类型定义。

使用 `data` 关键字可以定义新的数据类型：

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

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

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

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

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

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

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

```
-- file: ch03/BookStore.hs
data MagazineInfo = Magazine 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。在这个书店例子里，我们将 Int、String 和 [String] 三个类型的值应用到 Book，从而创建一个 BookInfo 类型的值：

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

定义类型的工作完成之后，可以到 ghci 里载入并测试这些新类型：

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

再看看前面在文件里定义的 myInfo 变量：

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

在 ghci 里面当然也可以创建新的 BookInfo 值：

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

可以用 :type 命令来查看表达式的值：

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

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

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

使用 `:info` 命令可以查看更多关于给定表达式的信息：

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

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

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

类型构造器和值构造器的命名

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

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

以下是这种用法的一个例子：

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

data BookReview = BookReview BookInfo CustomerID String
```

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

。

类型别名

可以使用类型别名，来为一个已存在的类型设置一个更具描述性的名字。

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

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

data BetterReview = BetterReview BookInfo CustomerID ReviewBody
```

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

类型别名也可以用来为啰嗦的类型设置一个更短的名字：

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

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

代数数据类型

Bool 类型是代数数据类型 (algebraic data type) 的最简单也是最常见的例子。一个代数类型可以有多于一个值构造器：

```
-- 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，创建出的值的类型都是相同的。

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

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

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

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

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

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

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

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

如果输入参数的类型不对或者数量不对，那么引发一个错误：

```
*Main> Invoice

<interactive>:7:1:
  No instance for (Show (CustomerID -> BillingInfo))
    arising from a use of `print'
```

```
Possible fix:
    add an instance declaration for (Show (CustomerID -> BillingInfo)
)
In a stmt of an interactive GHCi command: print it
```

ghci 抱怨我们没有给 Invoice 值构造器足够的参数。

[译注：原文这里的代码示例有错，译文已改正。]

什么情况下该用元组，而什么情况下又该用代数数据类型？

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

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

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

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

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

其中 a 和 b 的类型相同：

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

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

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

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

本文档使用 [看云](#) 构建

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

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

其中 c 和 d 的类型并不相同：

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

*Main> :type c
c :: Cetacean

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

以下是一个更细致的例子，它用两种不同的方式表示二维向量：

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

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

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

```
Prelude> :load AlgebraicVector.hs
[1 of 1] Compiling Main                ( AlgebraicVector.hs, interpreted )
Ok, modules loaded: Main.
*Main> Cartesian2D (sqrt 2) (sqrt 2) == Polar2D (pi / 4) 2

<interactive>:3:34:
    Couldn't match expected type `Cartesian2D'
    with actual type `Polar2D'
    In the return type of a call of `Polar2D'
```

```
In the second argument of `(==)', namely `Polar2D (pi / 4) 2'
In the expression:
  Cartesian2D (sqrt 2) (sqrt 2) == Polar2D (pi / 4) 2
```

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

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

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

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

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

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

其他语言里类似代数数据类型的东西

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

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

结构

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

以下是一个 C 结构，它等同于我们前面定义的 `BookInfo` 类型：


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

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

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

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

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

枚举

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
| 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 自己有能力弄清楚一点，它不会弄混两种不同的值。其中的原因，下一节《模式匹配》就会讲到。

[译注：原文这里将 `Poly` 写成了 `Square`。]

模式匹配

前面的章节介绍了代数数据类型的定义方法，本节将说明怎样去处理这些类型的值。

对于某个类型的值来说，应该可以做到以下两点：

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

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

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

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

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

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

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

以下是一个复杂一点的例子，这个函数计算出列表所有元素之和：

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

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

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

```
Prelude> []
[]

Prelude> 1:[]
[1]

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

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

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

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

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

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

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

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

*Main> sumList [1, 2]
```

3

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

组成和解构

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

我们通过应用值构造器来构建值：表达式 `Book9"CloseCalls"["JohnLong"]` 应用 `Book` 构造器到值 `9`、`"CloseCalls"` 和 `["JohnLong"]` 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 `Book9"CloseCalls"["JohnLong"]` 对模式 `(Bookidnameauthors)` 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 `CloseCalls`。
- 变量 `authors` 被绑定为 `["JohnLong"]`。

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 `(Circlevectordouble)` 模式时才会成功，而第二个表达式只有在 `(Polyvectors)` 时才会成功。这就是它们不会被混淆的原因。]

更进一步

对元组进行模式匹配的语法，和构造元组的语法很相似。

以下是一个可以返回三元组中最后一个元素的函数：

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

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

在 ghci 里测试这个函数：

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

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

模式匹配的“深度”并没有限制。以下模式会同时对元组和元组里的列表进行匹配：

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

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

在 ghci 里测试这个函数：

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

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

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

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

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

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

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

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 `(Book3titleauthors)` 只匹配 ID 值为 3 的那本书。

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。

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。

穷举匹配模式和通配符

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

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

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

如果我们将其作用于一个不能匹配的值，运行时就会报错：我们的软件有 bug！

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

在上面的例子中，函数定义时的方程里没有一个可以匹配 [] 这个值。

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

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

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

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

记录语法

给一个数据类型的每个成分写访问器函数是令人感觉重复而且乏味的事情。

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

我们把这种代码叫做“样板代码 (boilerplate code)”：尽管是必需的，但是又长又烦。Haskell 程序员不喜欢样板代码。幸运的是，语言的设计者提供了避免这个问题的方法：我们在定义一种数据类型的同时，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

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大型结构的过程中更方便了。

参数化类型

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

```
-- file: ch03/Nullable.hs
```

```
data Maybe a = Just a
             | Nothing
```

译注：Maybe, Just, Nothing 都是 Prelude 中已经定义好的类型

这段代码是不能在 ghci 里面执行的，它简单地展示了标准库是怎么定义 Maybe 这种类型的

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

```
-- 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 MaybeInt 或 Maybe[Bool]。如我们所希望的那样，这些都是不同的类型（译注：可能省略了“但是都可以成功传入作为参数”）。

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

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

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

递归类型

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

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

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

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

```
ghci> Nil
Nil
```

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

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

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

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

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

Tip

List 可以被当作是 list 吗？

让我们来简单的证明一下 Lista 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 Lista 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)

```

练习

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

报告错误

当我们的代码中出现严重错误时可以调用 Haskell 提供的标准函数 `error::String->a`。我们将希望打印出来的错误信息作为一个字符串参数传入。而该函数的类型签名看上去有些特别：它是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
```

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 (`[]:([]:[])`) \rightarrow (`[]:[]`) \rightarrow `[]`，`[]`，还是 2 个元素

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

引入局部变量 在函数体内部，我们可以在任何地方使用 `let` 表达式引入新的局部变量。请看下面这个简单的函数，它用来检查我们是否可以向顾客出借现金。我们需要确保剩余的保证

金不少于 100 元的情况下，才能出借现金，并返回减去出借金额后的余额。

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

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

Note

特别提示

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

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

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

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

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

屏蔽

我们可以在表达式中使用嵌套的 `let` 区块。

```
-- file: ch03/NestedLets.hs
```

```
foo = let a = 1
      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。

本文档使用 [看云](#) 构建

where 从句

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

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

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

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

局部函数与全局变量

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

```
-- file: ch03/LocalFunction.hs
pluralise :: String -> [Int] -> [String]
pluralise word counts = map plural counts
    where plural 0 = "no " ++ word ++ "s"
          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部的作用域内的任意变量：在本例中，我们使用了在外部函数 pluralise 中定义的变量 word。在 pluralise 的定义里，map 函数（我们将在下一章里再来讲解它的用法）将局部函数 plural 逐一应用于 counts 列表的每个元素。

我们也可以在代码的一开始就定义变量，语法和定义函数是一样的。

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

The Offside Rule and Whitespace in an Expression
The Case Expression
Common Beginner Mistakes with Patterns
Conditional Evaluation with Guards
Exercises

第四章：函数式编程

第四章：函数式编程 使用 Haskell 思考

初学 Haskell 的人需要迈过两个难关：

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

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

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

一个简单的命令行程序

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

这个接口程序读入一个文件，将函数应用到文件，并且将结果写到另一个文件：

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

import System.Environment (getArgs)

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

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

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

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

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

[译注：`id` 函数接受一个值，并原封不动地返回这个值，比如 `id"hello"` 返回值 `"hello"`，而 `id10` 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`。]

循环的表示

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

显式递归

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

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

    return acc;
}
```

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

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

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

asInt :: String -> Int
```

C 代码在遍历字符串的过程中，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。

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。一直这样下去，直到处理完整个列表为止。第二个等式确保计算会在列表为空时顺利终止。

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

```

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

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

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

```

以下是另一个 C 循环，它将字符串中的所有字母都设置为大写：

```

#include <ctype.h>

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

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

```

```
    return out;
}
```

以下是相等的 Haskell 版本：

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

import Data.Char (toUpper)

upperCase :: String -> String

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

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

```
Prelude> :module +Data.Char

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

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

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

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

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。

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 `myMapf1`。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。

筛选列表元素 另一种对列表的常见操作是，对列表元素进行筛选，只保留那些符合条件的元素。以下函数接受一个列表作为参数，并返回这个列表里的所有奇数元素。代码的递归情形比之前的 `map` 函数要复杂一些，它使用守卫对元素进行条件判断，只有符合条件的元素，才会被加入进结果列表里：

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

oddList :: [Int] -> [Int]

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

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

测试：

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

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

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

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

Prelude> odd 1
True

Prelude> odd 2
False

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

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

[译注：谓词函数是指那种返回 `Bool` 类型值的函数。]

稍后的章节会介绍如何定义 `filter`。

处理收集器并得出结果

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

对列表的所有元素求和，就是其中的一个例子：

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

mySum xs = helper 0 xs
  where helper acc (x:xs) = helper (acc + x) xs
        helper acc []    = acc
```

`helper` 函数通过尾递归进行计算。`acc` 是累积器参数，它记录了当前列表元素的总和。正如

我们在 `asInt` 函数看到的那样，这种递归计算是纯函数语言里表示 `loop` 最自然的方式。

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

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

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

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

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

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

以下是 Adler-32 算法的 Haskell 实现：

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

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

base = 65521

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

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

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

和前面抽取出 map 和 filter 函数类似，从 Adler32 函数里面，我们也可以抽取出一类通用的抽象，称之为折叠（fold）：它对一个列表中的所有元素做某种处理，并且一边处理元素，一边更新累积器，最后在处理完整个列表之后，返回累积器的值。

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

左折叠

以下是 foldl 函数的定义：

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

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

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

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

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

以下是使用 foldl 重写的 mySum 函数：

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

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

```
-- file: ch04/niceSum.hs
niceSum :: [Integer] -> Integer
```

```
niceSum xs = foldl (+) 0 xs
```

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

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

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

为什么使用 fold、map 和 filter？

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

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

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

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

从右边开始折叠

和 foldl 相对应的是 foldr，它从一个列表的右边开始进行折叠。

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

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

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

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

可以用 foldr 改写在《左折叠》一节定义的 niceSum 函数：

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

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

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

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

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

还记得当年大明湖畔的 filter 函数吗？它可以用显式递归来定义：

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

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

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

除此之外，filter 还可以通过 foldr 来定义：

```
-- file: ch04/myFilter.hs
myFilter p xs = foldr step [] xs
  where step x ys | p x      = x : ys
                  | otherwise = ys
```

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

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

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

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

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

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

更让人惊奇的是，foldl 甚至可以用 foldr 来表示：

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

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

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

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

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

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

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

最终产生的结果列表和原输入列表一模一样：

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

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

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

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

测试：

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

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

这个函数的效果等同于 `(++)` 操作符：

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

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

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

左折叠、惰性和内存泄漏

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

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

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

除非被显式地要求，否则最后的表达式不会被求值为 6。在表达式被求值之前，它会被保存在块里面。保存一个块比保存单独一个数字要昂贵得多，而被块保存的表达式越复杂，这个块占用的空间就越多。对于数值计算这样的廉价操作来说，块保存这种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容量，多得这个限制，我们可以在 `ghci` 里尝试求值一个大的块，而不必担心消耗掉全部内存。

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

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

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

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

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

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

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

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

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

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

ghci> :module +Data.List

ghci> foldl' (+) 0 [1..1000000]
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]
5000000000500000000
```

就是这样。]

延伸阅读 A tutorial on the universality and expressiveness of fold

[<http://www.cs.nott.ac.uk/~gmh/fold.pdf>] 是一篇关于 fold 的优秀且深入的文章。它使用了很多例子来展示如何通过简单的系统化计算技术，将一些显式递归的函数转换成 fold。

匿名 (lambda) 函数 在前面章节定义的函数中，很多函数都带有一个简单的辅助函数：

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

import Data.List (isInfixOf)

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

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

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

以下是前面的 `isInAny` 函数用 lambda 改写的版本：

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

import Data.List (isInfixOf)

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

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

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

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

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

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

而 lambda 只能覆盖其中一种情形：

```
-- file ch04/unsafeHead.hs
unsafeHead = \(x:_) -> x
```

如果一不小心，将这个函数应用到错误的模式上，它就会给我们带来麻烦：

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

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

*Main> unsafeHead [1]
1

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

因为这个 lambda 定义是完全合法的，它的类型也没有错误，所以它可以被顺利编译，而最终在运行期产生错误。这个故事说明，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。

部分函数应用和柯里化

类型签名里的 `->` 可能会让人感到奇怪：

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

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

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

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

以下是一个完全合法的 Haskell 表达式：

```
Prelude> :module +Data.Char

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

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

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

每当我们把一个参数传给一个函数时，这个函数的类型签名最前面的一个元素就会被“移除掉”。这里用函数 `zip3` 来做例子，这个函数接受三个列表，并将它们压缩成一个包含三元组的列表：

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

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

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

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

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

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

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

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

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

只要给部分函数补充上足够的参数，它就可以被成功求值：

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

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

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

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

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

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

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

import Data.List (isInfixOf)

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

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

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

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

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

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

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

测试：

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

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

节

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

```
Prelude> (1+) 2
3

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

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

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

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

```
Prelude> (2^) 3
8

Prelude> (^3) 2
8
```

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

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

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

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

```
True
```

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

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

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

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

通过这种用法，可以再一次提升 `isInAny3` 函数的可读性：

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

import Data.List (isInfixOf)

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

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

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

```
]
```

As-模式

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

```
Prelude> :m +Data.List
```

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

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种模式：我们先应用一个函数，然后又将这个函数得出的结果应用到另一个函数。可以将这个模式定义为一个函数：

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

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

compose 函数可以用于粘合两个函数：

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

*Main> :m +Data.List

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

通过柯里化，可以丢掉 xs 函数：

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

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

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

(.) 操作符并不是什么特殊语法，它只是一个普通的操作符：

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

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

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

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

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

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

```
Prelude> :module +Data.Char

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

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

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) */
```

本文档使用 [看云](#) 构建

```
#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 foldrstep[] 应用到各行当中，其中 step 辅助函数用于过滤和提取符合格式的宏名字：

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

import Data.List (isPrefixOf)

dlts :: String -> [String]

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

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

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

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

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

import Data.List (isPrefixOf)

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

]

编写可读代码的提示

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

本文档使用 [看云](#) 构建

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

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

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

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

内存泄漏和严格求值

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

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

通过 `seq` 函数避免内存泄漏

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

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

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

`seq` 函数的类型签名和之前看过的函数都有些不同，昭示了它的特殊身份：

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

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

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

]

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

比如说，对于以下表达式：

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

它展开为：

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

它强迫 `new` 求值为 3，然后返回它的第二个参数：

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

最终得到结果 3。

因为 `seq` 的存在，这个创建过程没有用到任何块。

seq 的用法

本节介绍一些更有效地使用 seq 的指导规则。

要正确地产生 seq 的作用，表达式中被求值的第一个必须是 seq：

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

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

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

为了严格求值多个值，可以连接起 seq 调用：

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

一个常见错误是，将 seq 用在没有关联的连个表达式上面：

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

stepzerox 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 stepzerox 求值，而它的结果并不会影响 badExpression 表达式内的 stepzerox。正确的用法应该是一个 let 结果保存起 stepzerox 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
本文档使用 [看云](#) 构建

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，到时会说明跟多 `seq` 的用法和细节。

第五章：编写 JSON 库

第五章：编写 JSON 库 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](http://www.ietf.org/rfc/rfc4627.txt) [http://www.ietf.org/rfc/rfc4627.txt] 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”对的无序收集器（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}
```

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
            | 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"), ("compiler", JString "GH
C")]
JObject [("language",JString "Haskell"),("compiler",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

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

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

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。

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}]
```

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。

```

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"

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。

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 xs。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 )
本文档使用 看云 构建
```



```
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 upcaseFirs
t (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给函数添加类型签名，然后随着对类型系统了解的加深，逐步放松要求。

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。

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

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，并列出生成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。]

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。

```
-- 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。

```
-- 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`）。

```
-- 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
```

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/PrettyJSON.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 `myLengthxs=lengthxs` 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
```

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，我们都列出了想要引入我们的模块的命名空间的名字。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，——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 Haskell 程序员会知道哪个名字来自那些模块。

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
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

A mathematical moment(to be added)

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。

```
-- 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。

```
-- 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。

```
-- 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。

```
-- 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。

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

importPrettify 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。]

```
-- 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"++""。

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
```

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 `pretty2` 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。现在我们有俩个子表达式：`best0[Char'',Char'a']` 和 `best0[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 `nicest0"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 }
```

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。

1. 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
```

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。

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 `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。]

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。）

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
```

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`。

[Hughes95]	John Hughes. " The design of a pretty-printing library " [http://citeseer.ist.psu.edu/hughes95design.html]". May, 1995. First International Spring School on Advanced Functional Programming Techniques. Bastad, Sweden. .
[Wadler98]	Philip Wadler. " A prettier printer " [http://citeseer.ist.psu.edu/wadler98prettier.html]". March 1998.

第六章：类型类

第六章：类型类

类型类 (typeclass) 是 Haskell 最强大的功能之一：它用于定义通用接口，为各种不同的类型提供一组公共特性集。

类型类是某些基本语言特性的核心，比如相等性测试和数值操作符。

在讨论如何使用类型类之前，先来看看它能做什么。

类型类的作用

假设这样一个场景：我们想对 Color 类型的值进行对比，但 Haskell 的语言设计者却没有实现 == 操作。

要解决这个问题，必须亲自实现一个相等性测试函数：

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

data Color = Red | Green | Blue

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

在 ghci 里测试：

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

*Main> colorEq Green Green
True

*Main> colorEq Blue Red
False
```

过了一会，程序又添加了一个新类型——职位：它对公司中的各个员工进行分类。

在执行像是工资计算这类任务是，又需要用到相等性测试，所以又需要再次为职位类型定义相等性测试函数：

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

data Role = Boss | Manager | Employee

roleEq :: Role -> Role -> Bool
roleEq Employee Employee = True
roleEq Manager  Manager  = True
roleEq Boss     Boss     = True
roleEq _        _        = False
```

测试：

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

*Main> roleEq Boss Boss
True

*Main> roleEq Boss Employee
False
```

colorEq 和 roleEq 的定义揭示了一个问题：对于每个不同的类型，我们都需要为它们专门定义一个对比函数。

这种做法非常低效，而且烦人。如果同一个对比函数（比如 == ）可以用于对比任何类型的值，这样就会方便得多。

另一方面，一般来说，如果定义了相等测试函数（比如 == ），那么不等测试函数（比如 /= ）的值就可以直接对相等测试函数取反（使用 not ）来计算得出。因此，如果可以通过相等测试函数来定义不等测试函数，那么会更方便。

通用函数还可以让代码变得更通用：如果同一段代码可以用于不同类型的输入值，那么程序的代码量将大大减少。

还有很重要的一点是，如果在之后添加通用函数对新类型的支持，那么原来的代码应该不需要进行修改。

Haskell 的类型类可以满足以上提到的所有要求。

本文档使用 [看云](#) 构建

什么是类型类？

类型类定义了一系列函数，这些函数对于不同类型的值使用不同的函数实现。它和其他语言的接口和多态方法有些类似。

[译注：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。]

我们定义一个类型类来解决前面提到的相等性测试问题：

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

类型类使用 `class` 关键字来定义，跟在 `class` 之后的 `BasicEq` 是这个类型类的名字，之后的 `a` 是这个类型类的实例类型（instance type）。

`BasicEq` 使用类型变量 `a` 来表示实例类型，说明它并不将这个类型类限定于某个类型：任何一个类型，只要它实现了这个类型类中定义的函数，那么它就是这个类型类的实例类型。

实例类型所使用的名字可以随意选择，但是它和类型类中定义函数签名时所使用的名字应该保持一致。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。

`BasicEq` 类型类只定义了 `isEqual` 一个函数——它接受两个参数作为输入，并且这两个参数都指向同一种实例类型：

```
Prelude> :load BasicEq_1.hs
[1 of 1] Compiling Main                ( BasicEq_1.hs, interpreted )
Ok, modules loaded: Main.

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

作为演示，以下代码将 `Bool` 类型作为 `BasicEq` 的实例类型，实现了 `isEqual` 函数：

```
instance BasicEq Bool where
    isEqual True  True  = True
    isEqual False False = True
    isEqual _    _     = False
```

在 ghci 里验证这个程序：

```
*Main> isEqual True True
True

*Main> isEqual False True
False
```

如果试图将不是 BasicEq 实例类型的值作为输入调用 isEqual 函数，那么就会引发错误：

```
*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"
```

错误信息提醒我们，[Char] 并不是 BasicEq 的实例类型。

稍后的一节会介绍更多关于类型类实例的定义方式，这里先继续前面的例子。这一次，除了 isEqual 之外，我们还想定义不等测试函数 isNotEqual：

```
class BasicEq a where
  isEqual    :: a -> a -> Bool
  isNotEqual :: a -> a -> Bool
```

同时定义 isEqual 和 isNotEqual 两个函数产生了一些不必要的工作：从逻辑上讲，对于任何类型，只要知道 isEqual 或 isNotEqual 的任意一个，就可以计算出另外一个。因此，一种更省事的办法是，为 isEqual 和 isNotEqual 两个函数提供默认值，这样 BasicEq 的实例类型只要实现这两个函数中的一个，就可以顺利使用这两个函数：

```
class BasicEq a where
  isEqual :: a -> a -> Bool
  isEqual x y = not (isNotEqual x y)

  isNotEqual :: a -> a -> Bool
  isNotEqual x y = not (isEqual x y)
```

以下是将 Bool 作为 BasicEq 实例类型的例子：

```
instance BasicEq Bool where
  isEqual False False = True
  isEqual True  True  = True
  isEqual _     _     = False
```

我们只要定义 isEqual 函数，就可以“免费”得到 isNotEqual：

```
Prelude> :load BasicEq_3.hs
[1 of 1] Compiling Main                ( BasicEq_3.hs, interpreted )
Ok, modules loaded: Main.

*Main> isEqual True True
True

*Main> isEqual False False
True

*Main> isNotEqual False True
True
```

当然，如果闲着没事，你仍然可以自己亲手定义这两个函数。但是，你至少要定义两个函数中的一个，否则两个默认的函数就会互相调用，直到程序崩溃。

定义类型类实例

定义一个类型为某个类型类的实例，指的就是，为某个类型实现给定类型类所声明的全部函数。

比如在前面，BasicEq 类型类定义了两个函数 isEqual 和 isNotEqual：

```
class BasicEq a where
  isEqual :: a -> a -> Bool
  isEqual x y = not (isNotEqual x y)

  isNotEqual :: a -> a -> Bool
  isNotEqual x y = not (isEqual x y)
```

在前一节，我们成功将 Bool 类型实现为 BasicEq 的实例类型，要使 Color 类型也成为 BasicEq 类型类的实例，就需要另外为 Color 类型实现 isEqual 和 isNotEqual：


```
instance BasicEq Color where
    isEqual Red Red = True
    isEqual Blue Blue = True
    isEqual Green Green = True
    isEqual _ _ = True
```

注意，这里的函数定义和之前的 `colorEq` 函数定义实际上没有什么不同，唯一的区别是，它使得 `isEqual` 不仅可以对 `Bool` 类型进行对比测试，还可以对 `Color` 类型进行对比测试。

更一般地说，只要为相应的类型实现 `BasicEq` 类型类中的定义，那么 `isEqual` 就可以用于对比任何我们想对比的类型。

不过在实际中，通常并不使用 `BasicEq` 类型类，而是使用 Haskell Report 中定义的 `Eq` 类型类：它定义了 `==` 和 `/=` 操作符，这两个操作符才是 Haskell 中最常用的测试函数。

以下是 `Eq` 类型类的定义：

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

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

稍后会介绍更多使用 `Eq` 类型类的信息。

几个重要的内置类型类

前面两节分别介绍了类型类的定义，以及如何让某个类型成为给定类型类的实例类型。

正本节会介绍几个 `Prelude` 库中包含的类型类。如本章开始时所说的，类型类是 Haskell 语言某些特性的基石，本节就会介绍几个这方面的例子。

更多信息可以参考 Haskell 的函数参考，那里一般都给出了类型类的详细介绍，并且说明，要成为这个类型类的实例，需要实现那些函数。

Show

`Show` 类型类用于将值转换为字符串，它最重要的函数是 `show`。

`show` 函数使用单个参数接收输入数据，并返回一个表示该输入数据的字符串：

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

以下是一些 show 函数调用的例子：

```
Main> show 1
"1"

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

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

Ghci 输出一个值，实际上就是对这个值调用 putStrLn 和 show：

```
Main> 1
1

Main> show 1
"1"

Main> putStrLn (show 1)
1
```

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：

```
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：

```
instance Show Color where
  show Red    = "Color 1: Red"
  show Green  = "Color 2: Green"
  show Blue   = "Color 3: Blue"
```

Read

Read 和 Show 类型类的作用正好相反，它将字符串转换为值。

Read 最有用的函数是 read：它接受一个字符串作为参数，对这个字符串进行处理，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
```

以下代码展示了 read 的用法：

```
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。

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文件，又或者，反过来，需要将文件中的数据转换为内存中的数据实体。这种转换过程称为序列化和反序列化。

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。

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` 可以看到，`(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。

数字类型

Haskell 有一集非常强大的数字类型：从速度飞快的 32 位或 64 位整数，到任意精度的有理数，包罗万有。

除此之外，Haskell 还有一系列通用算术操作符，这些操作符可以用于几乎所有数字类型。而对数字类型的这种强有力的支持就是建立在类型类的基础上的。

作为一个额外的好处（side benefit），用户可以定义自己的数字类型，并且获得和内置数字类型完全平等的权利。

以下表格显示了 Haskell 中最常用的一些数字类型：

表格 6.1：部分数字类型

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

大部分算术操作都可以用于任意数字类型，少数的一部分函数，比如 `asin`，只能用于浮点数类型。

以下表格列举了操作各种数字类型的常见函数和操作符：

表格 6.2：部分数字函数和

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

cos	Floating a => a -> a	Prelude	cosh 和 acosh ，类型和 cos 一样。
div	Integral 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 值转换为任意数字类型。
fromIntegral	(Integral a, Num b) => a -> b	Prelude	一个更通用的转换函数，将任意 Integral 值转为任意数字类型。
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	Prelude	计算指定底数对数。
maxBound	Bounded a => a	Prelude	有限长度数字类型的最大值。
minBound	Bounded a => a	Prelude	有限长度数字类型的最小值。
mod	Integral a => a -> a	Prelude	整数取模。
pi	Floating a => a	Prelude	圆周率常量。
quot	Integral 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	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 类型一样。
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 类型一样。
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	向下取整
xor	Bits a => a -> a -> a	Data.Bits	二进制异或操作

数字类型及其对应的类型类列举在下表：

表格 6.3 ： 数字类型的类型类实例

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 6.2 列举了一些数字类型之间进行转换的函数，以下表格是一个汇总：

表格 6.4 ： 数字类型之间的转换

源类型	目标类型	
	Double, Float	Int, Word
Double, FloatInt, WordIntegerRational	fromRational . toRationalfromIntegralfromIntegralfromRational	truncate *fromIntegralfromIntegral*

- 除了 truncate 之外，还可以使用 round 、 ceiling 或者 float 。

第十三章会说明，怎样用自定义数据类型来扩展数字类型。

相等性，有序和对比

除了前面介绍的通用算术符号之外，相等测试、不等测试、大于和小于等对比操作也是非常常见的。

其中，Eq 类型类定义了 == 和 /= 操作，而 >= 和 <= 等对比操作，则由 Ord 类型类定义。

需要将对比操作和相等性测试分开用两个类型类来定义的原因是，对于某些类型，它们只对相等性测试和不等测试有兴趣，比如 Handle 类型，而部分有序操作（particular ordering，大于、小于等）对它来说是没有意义的。

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 实例的类型也不在少数。

自动派生

对于简单的数据类型，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。

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

```
data Color = Red | Green | Blue
    deriving (Read, Show, Eq, Ord)
```

测试：

```
*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
```

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。

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将 `MyType` 函数的输出转换成字符串，这会造成编译错误。

除此之外，当使用自动推导将某个类型设置为给定类型类的实例时，定义这个类型时所使用的其他类型，也必须是给定类型类的实例（通过自动推导或手动添加的都可以）。

举个例子，以下代码不能使用自动推导：

```
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` 进行自动推导：

```
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推导，因为它对 `Book` 类型也使用了自动推导，使得 `Book` 类型变成了 `Show` 的实例：

```
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
```

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。这里是一段由搜索引擎返回的实际 JSON 数据。删除重整之后：

```
{
  "query": "awkward squad haskell",
  "estimatedCount": 3920,
  "moreResults": true,
  "results":
  [{
    "title": "Simon Peyton Jones: papers",
    "snippet": "Tackling the awkward squad: monadic input/output ..."
  },
  {
    "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",
  }
]
```

进一步简化之，并用 Haskell 表示：

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

```

result :: JValue
result = JObject [
  ("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"，我们就必须把 JNumber 构造器换成 JString 构造器。

Haskell 的类型类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，编译器会自动选择相应的 toJValue 实现。

我们也提供了 fromJValue 函数，它把 JValue 值转换成我们希望的类型。

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。Either 类型的结构相同，但它在错误发生时会调用 Left 构造器，并且还接

受一个参数。

```
-- 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` 参数的类型，以便在出错时提供有用的描述。为了说明在实际中怎么使用 `Either` 类型，我们来看一个简单实例。

```
-- 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(toJValueTrue))::EitherJSONErrorBool`。]

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：

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

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

`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能用具体类型替代类型变量。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。

```
-- 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` 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 {-#LANGUAGEFlexibleInstances#-} 来解决。]

生活在开放世界

Haskell 的设计允许我们任意创建类型类实例。

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

我们可以在任意地方创建新实例，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。

我们希望把列表转为 JSON 数组。现在先不用关心实现细节，暂时用 undefined 替代函数内容即可。

```
-- 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
```

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，当我们使用序对列表实例时，麻烦来了。

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

重叠实例问题是由 Haskell 的开放世界假设造成的。这里有一个更简单的例子来说明发生了什么。

```
-- file: ch06/Overlap.hs
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 ++ "<<"

```

对于序对，我们有两个 Borked 类型类实例：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。编译器必须选择一个实例来用。由于这两个实例都能用，所以看上去它好像只要选那个更相关（specific）的实例就可以了。

但是，GHC 默认是保守的。它坚持只能有一个可用实例。这样，当我们试图使用 bork 时，它就会报错。

Note

重叠实例什么时候会出问题？

之前我们提到，我们可以把某个类型类的实例分散在几个模块中。GHC 并不会在意重叠实例的存在。相反，只有当我们使用受影响类型类的函数，GHC 被迫要选择使用哪个实例时，它才会报错。

取消类型类的一些限制

通常，我们不能给多态类型（polymorphic type）的特化版本（specialized version）写类型类实例。[Char] 类型就是多态类型 [a] 特化成 Char 的结果。因此我们禁止声明 [Char] 为某个类型类的实例。这非常不方便，因为字符串在代码中无处不在。

FlexibleInstances 语言扩展取消了这个限制，它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，它解决了重叠实例带来的问题。如果存在重叠实例，编译器会选择最相关的（specific）那一个。

我们经常把这个扩展和 TypeSynonymInstances 放在一起使用。下面是一个例子。

```

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

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
```

如果我们对 String 应用 foo，编译器会选择 String 的特定实现。即使 [a] 和 Char 都是 Foo 的实例，但由于 String 实例更相关，因此 GHC 选择了它。

即使开了 OverlappingInstances 扩展，如果 GHC 发现了多个同样相 (equally specific) 关的实例，它仍然会拒绝代码。

何时使用 OverlappingInstances 扩展 (to be added)

字符串的 show 是如何工作的？

OverlappingInstances 和 TypeSynonymInstances 语言扩展是 GHC 特有的，Haskell 98 并不支持。然而，Haskell 98 中的 Show 类型类在转化 Char 列表和 Int 列表时却用了不同的方法。它用了个聪明但简单的小技巧。

Show 类型类定义了转换单个值的 show 方法和转换列表的 showList 方法。showList 默认使用中括号和逗号转换列表。

[a] 的 Show 实例使用 showList 实现。Char 的 Show 实例提供了一个特殊的 showList 实现，它使用双引号，并转义非 ASCII 打印字符。

结果是，如果有人想对 [Char] 应用 show，编译器会选择 showList 的实现，并使用双引号正确转换这个字符串。

这样，换个角度看问题，我们就能避免 OverlappingInstances 扩展了。

如何给类型定义新身份 (Identity)

除了熟悉的 data 关键字外，Haskell 还允许我们用 newtype 关键字来创建新类型。

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

newtype NewtypeInt = N Int
```

```
deriving (Eq, Ord, Show)
```

`newtype` 声明的作用是重命名现有类型，并给它一个新身份。可以看出，它的用法和使用 `data` 关键字进行类型声明看起来很相似。

Note

`type` 和 `newtype` 关键字

尽管名字类似，`type` 和 `newtype` 关键字的作用却完全不同。`type` 关键字给了我们另一种指代某个类型的方法，类似于给朋友起的绰号。我们和编译器都知道 `[Char]` 和 `String` 指的是同一个类型。

相反，`newtype` 关键字的存在是为了隐藏类型的本性。考虑这个 `UniqueID` 类型。

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

编译器会把 `UniqueID` 当成和 `Int` 不同的类型。作为 `UniqueID` 的用户，我们只知道它是一个唯一标识符；我们并不知道它是用 `Int` 来实现的。

在声明 `newtype` 时，我们必须决定暴露被重命名类型的哪些类型类实例。这里，我们让 `NewtypeInt` 提供 `Int` 类型的 `Eq`，`Ord` 和 `Show` 实例。这样，我们就可以比较和打印 `NewtypeInt` 类型的值了。

```
*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。例如，我们不能做加法。

```
*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` 的值构造器创建新值，或者对现有值进行模式匹配。如果 `newtype` 没用自动派生来暴露对应类型的类型类实现的话，我们可以自己写一个新实例或者干脆不实现那个类型类。`data` 和 `newtype` 的区别 `newtype` 关键字给现有类型一个不同的身份，相比起 `data`，它使用时的限制更多。具体来讲，`newtype` 只能有一个值构造器，并且这个构造器只能有一个字段。

```
-- file: ch06/NewtypeDiff.hs
-- 可以：任意数量的构造器和字段
data TwoFields = TwoFields Int Int

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

-- 可以：使用类型变量
newtype Param a b = Param (Either a b)

-- 可以：使用记录语法
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
```

除此之外，`data` 和 `newtype` 还有一个重要区别。由 `data` 关键字创建的类型在运行时有一个簿记开销，如记录某个值是用哪个构造器创建的。而 `newtype` 只有一个构造器，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。

由于 `newtype` 的构造器只在编译时使用，运行时甚至不存在，用 `newtype` 定义的类型和用 `data` 定义的类型在匹配 `undefined` 时会有不同的行为。

为了理解它们的不同点，我们首先回顾一下普通数据类型的行为。我们已经非常熟悉，在运行时对 `undefined` 求值会导致崩溃。

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

我们把 `undefined` 放进 `D` 构造器创建一个 `DataInt`，然后对它进行模式匹配。

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

由于我们的模式匹配只匹配构造器而不管里面的值，`undefined` 未被求值，因而不会抛出异常。

下面的例子没有使用 `D` 构造器，因而模式匹配时 `undefined` 被求值，异常抛出。

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

当我们用 `N` 构造器创建 `NewtypeInt` 值时，它的行为与使用 `DataInt` 类型的 `D` 构造器相同：没有异常。

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

但当我们把表达式中的 `N` 去掉，并对 `undefined` 进行模式匹配时，关键的不同点来了。

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

没有崩溃！由于运行时不存在构造器，匹配 `N` 实际上就是在匹配通配符：由于通配符总可以被匹配，所以表达式是不需要被求值的。

命名类型的三种方式

这里简要回顾一下 `haskell` 引入新类型名的三种方式。

- `data` 关键字定义一个真正的代数数据类型。
- `type` 关键字给现有类型定义别名。类型和别名可以通用。
- `newtype` 关键字给现有类型定义一个不同的身份（`distinct identity`）。原类型和新类型

不能通用。

JSON typeclasses without overlapping instances 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
    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.
```

[译注：译者得到的输出和原文有出入，这里提供的是使用最新版本 GHC 得到的输出。] 错误信息中提到的“monomorphism”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。Note 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 this 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。

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。

第七章：I/O

第七章：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。

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!
```

这相单简单，结果很明显。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？

`IOsomething` 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 `IOt` 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并且调用存代码。

`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。

只有当你有多余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动作中调用存代码的一个例子：

```
-- 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 `name2replayinpStr` 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` 得到存代码的结果。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(name2replyinpStr) 。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是不是真正的命令式的呢？](#) 和 [惰性I/O](#) 章节来讨论这个问题。

Pure vs. I/O

这里有一个比较的表格，用来帮助理解存代码和I/O之间的区别。当我们说起存代码的时候，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

为什么纯不纯很重要？

在这一节中，我们已经讨论了Haskell是怎么在存代码和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副作用_](#) 一节看更多的讨论。

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
       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 `IOInt` 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 `runghctoupper-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
```

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。

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。所有这些Haskell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以还打开着的文件。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，最好任何时候由你负责调用 `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`。

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 `IOInteger`。

`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。

`hSeekhandleSeekFromEnd0` 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。

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 [局部函数应用和柯里化](#)。

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
本文档使用 [看云](#) 构建

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。

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。

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。

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(catch)
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 temp =
    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 temp
        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 temp tempdata

        -- Get our new position. This doesn't actually modify pos
        -- in memory, but makes the name "pos" correspond to a different
        -- value for the remainder of the "do" block.
        pos <- hTell temp
        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 temp AbsoluteSeek 0

        -- hGetContents performs a lazy read of the entire file
        c <- hGetContents temp

        -- 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 an
      -- 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).
      tempdir <- catch (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.
      -- 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 printc 当做Haskell字面量显示出来。这和 putStrLn(showc) 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，你的临时文件的名称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 putStrLnc 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。

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。

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->IOString`。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。

```
-- 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

如果我们尝试去抓住上面例子中的 `inpStr`，在超过它被使用的地方（`processData` 调用那），内存中将没有它了。这是因为编译器会强制保存 `inpStr` 的值在内存里，为了以后的使用。这里我们知道 `inpStr` 讲不会被重用，它一被使用完就会被释放内存。只要记住：最后一次使用后释放内存。

这个程序为了清楚地表明使用了存代码，显得有点啰嗦。这里有更加简洁的版本，新版本在下一个例子里：

```
-- file: ch07/toupper-lazy2.hs
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

当使用 `hGetContents` 的时候，记住，就算你可能在剩下的程序里面不再显式引用句柄，你绝不能关闭句柄，直到在你结束对结果的使用后，这点很重要。提早关闭会造成丢失文件数据的部分或全部。因为Haskell是惰性的，一般地可以假定，你只有在包含输入的计算被算出结果输出之后，你才能使用这个输入。

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。

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。

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：

```
-- 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`。

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：

```
-- 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。

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。

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 ()
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。

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：

```
-- 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

这些函数其实可以做 I/O 更多的事情，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` 把每个数字转换成一个 `String`，`str2action` 把每个 `String` 转换成一个操作。`mapM` 把这些单独的操作组合成一个总的操作，然后打印出这些行。

串联化

`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 `do` 代码块中只有一行是类似的。你可能会写 `putStrLn"line1">>putStrLn"line2"` 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` 代码块。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。

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。

Haskell 实际上是命令式的吗？

这些 `do` 代码块可能开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。

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中的存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。

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还是需要有一个系统调用，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。缓冲区允许我们去读相同数量的数据可以用少得多的I/O请求。

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做好的，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 `Just4096` 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 `hSetBufferingstdin(BlockBufferingNothing)`。

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。

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。

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。

第八章：高效文件处理、正则表达式、文件名匹配

第八章：高效文件处理、正则表达式、文件名匹配 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。

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 importqualified 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。

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.ByteString.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 MaybeInt 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-0.10.0 ... 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
```

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：“匹配这个范围内的任意字符”。

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。

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
```

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，其以中缀预算符 (==~) (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。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-0.10.0 ... linking ... done.
Loading package containers-0.10.1 ... 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 `regex` 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. Jonsonii, 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 `[[String]]`，将返回由所有匹配的字符串组成的列表。

```
ghci> "I, B. Jonsonii, 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> "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。

进一步了解正则表达式 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于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> pack "good food" =~ ".ood" :: [[ByteString]]
```

```
["good"], ["food"]]
```

上面的例子中，我们使用 `pack` 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。

```
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。

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 `(foo|fo*)` 匹配字符串 `foooooo` 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。

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。这使我们可以用最本文档使用 [看云](#) 构建

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
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-0.10.0 ... linking ... done.
Loading package containers-0.10.1 ... 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.
"^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
```

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。

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。

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，知道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。

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"
```

The name of the `dropTrailingPathSeparator` function is perfectly descriptive. No comments
`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
```

本文档使用 [看云](#) 构建

```
ghci> splitFileName "foo/bar"
Loading package filepath-1.1.0.0 ... linking ... done.
("foo/", "bar")
```

If we didn't remember (or know enough) to remove that slash, we'd recurse endlessly in `namesMatching`, because of the following behaviour of `splitFileName`. 1 comment 如果忘记（或不够了解）要去掉这个斜线，我们将在 `namesMatching` 函数中进行无止境的递归匹配，看看后面演示的 `splitFileName` 的行为你就会明白。

```
ghci> splitFileName "foo/"
("foo/", "")
```

你或许能够想想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
            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。

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。

1. 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。
2.
 - 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，`*` 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。

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。）

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。

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。这些函数恰好漏过了我们的 `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 "*.c"
  c"
```

`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。

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。

第九章：I/O学习 —— 构建一个用于搜索文件系统的库

第九章：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。

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实体名称。

```
-- 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”，“实体是一个文件”，“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

简单的开始：递归遍历目录

本文档使用 [看云](#) 构建

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]
```

循环体将检查当前实体是否为目录，如果是，则递归调用 `getrecursivacontents` 函数列出这个目录（的内容），如果否，则返回只含有当前实体名称一个元素的列表，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 `ifos.path.isdir(path)` 来表示，然而，`doesDirectoryExist` 函数是一个动作，它的返回类型是 `IOBool` 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 `bool` 返回值，这样我们就能够在 `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)

再次认识匿名和命名函数

在 [Anonymous \(lambda\) functions](http://book.realworldhaskell.org/read/functional-programming.html#fp.anonymous) [http://book.realworldhaskell.org/read/functional-programming.html#fp.anonymous] 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

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

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整的表达，他只关注文件夹中的实体名称，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有和文件一样拥有 .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元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

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->IOBool 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 IOa 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来坚持断言的纯粹，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 来判断目录实体是目录还是文件，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, search
able = 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` 告诉我们实体上次被修改的时间：

```
ghci> :type getModificationTime
getModificationTime :: FilePath -> IO System.Time.ClockTime
ghci> getModificationTime "."
Mon Aug 18 12:08:24 CDT 2008
```

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` 来判断实体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 `IOBool`，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空白。

```
-- 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
  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 `IOmonad` 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，他执行了所有IO发生在 `p` 上的可能引起负面效果的任务，因此我们可以使 `p` 对负面效果免疫，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

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实体上调用 `getFileSize`，如果实体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，而当实体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)

saferFileSize path = handle (\_ -> return Nothing) $ do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return (Just size)
```

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，但他仍未能完全的正常工作，存在 `openFile` 会成功的目录实体，但 `hFileSize` 会抛出异常，这将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。

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 and 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

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，断言需要四个参数，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实体大小是 -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 InfoPBool 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代表断言），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 InfoPa，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 InfoPBool，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。

多用提升 (lifting) 少用模板

除了 `equalP`，我们还将能够编写其他二进制函数，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 (<)
```

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 `==`，我们就可以把二进制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。

我们可以通过组合字（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 `flipmapM`。

```
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]
```

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

andP = liftP2 (&&)
orP = liftP2 (||)
```

注意 `liftP2` 非常像我们之前的 `liftP`，事实上，这更加通用，因为我们可以用 `liftP` 代替 `liftP2`：

```
-- 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
```

在加入组合字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字 `liftPath`。

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

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 `infixl9` 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

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，简便方法之一是在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

```

-- 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

```

记录语法给我们自由控制函数的权限，如 infoPath，traverse 函数中的这种类型是简单地，正如我们之前期望的那样，如果需要一个文件或者目录的信息，就调用 getInfo 函数：

```

-- file: ch09/ControlledVisit.hs
traverse :: ([Info] -> [Info]) -> FilePath -> IO [Info]

```

traverse 的定义很短，但很有分量：

```
-- 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 是目录实体的列表，同时确定当前目录的所有元素都在这个列表中，这时通过 mapM 将 getInfo 附加到结果返回的路径上。

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（有额外的方法保证我们不在访问这个路径），或者返回当前值作为列表唯一元素的列表（来匹配递归的返回类型）。

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。

本行的新上下文中使用提升技术，liftM 函数需要一个规则函数，concat，并且提升到 IO 的monad操作，换言之，他需要 forM 通过 IO monad 操作的返回值，并将 concat 附加其上（获得一个 [Info] 类型的返回值，这也是我们所需要的）并将结果值返回给 IO monad。

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字，`maybeIO`，将一个可能抛出异常的 IO 操作转换成用 `Maybe` 包装的结果

练习

1. 在以代数顺序遍历一个目录树时如何确定需要通过的内容。
2. 使用 `id` 作为控制函数，`traverseid` 扮演一个前序递归树，在子目录之前他返回一个父目录，写一个控制函数让 `traverse` 表现为一个后序遍历，返回子目录在父目录之前。
3. 使得《谓词组合》一节里面的断言和组合字可以处理新的 `info` 类型。
4. 给 `traverse` 写一个包装器，让你通过谓词控制递归，并通过谓词过滤返回结果

代码深度，可读性和学习过程

`traverse` 这样深度的代码在 Haskell 中并不多见，在这种表达方式中里学习的收获是巨大的，同时也并不需要大量的练习才能以这种方式流利的阅读和写作代码：

```
-- 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

这里所做的一切都是创建一个新的替代，通过部分应用（partial application）和函数组合（function composition）替代liberally，在 where 块中我们已经定义了一些本地函数，在 maybe 组合子中，使用了 case 表达式，为了替代 liftM，我们手动将 concat 提升。

并不是说深度是一个不好的特征，traverse 函数的每一行原始代码都很短，我们引入一个本地变量和本地函数来保证代码干净且足够短，命名注意可读性，同时使用函数组合管道，最长的管道只含有三个元素。

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，他对Haskell库和编写整洁的代码非常熟悉，但他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

观察迭代函数的另一种方法

相比原始的 betterFind 函数，迭代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。

一个可能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，假如说我们知道需要比三个多很多的实体组，并且这些实体组是这10000个我们需要遍历实体中的前几个，这种情况下就不需要访问剩下的实体，这并不是个故弄玄虚的问题，举个栗子，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。

从另一个角度看，我们尝试定位之前两个遍历函数的弱点：我们如何看待文件系统遍历阶级目录下的一个文件夹？

相似的文件夹，foldr 和 foldl'，干净的生成遍历并计算出一个结果，很难把这个想法从列

表扩展到目录树，但我们仍乐于在 `fold` 中加入一个控制元素，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` 值来表达这个目录实体，并返回一个新种子和一个我们 `fold` 函数的说明，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，遍历将立即释放，被 `Done` 包裹的值将作为结果返回。
- 如果这个说明被跳过，并且当前 `info` 代表一个目录，遍历将不在递归寻找这个目录。
- 其他，这个便利仍将继续，使用包裹值作为下一个调用 `fold` 函数的参数。

目录逻辑上是左序的，因为我们开始从我们第一个遇到的实体开始 `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)
```

这部分代码中有意思的部分很少，开始是通过 `scoping` 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` 是本地函数，我们不需要通过 `foldTree` 的 `iter` 变量来进入他，可以从外部进入，相似的，`walk` 也可以在外看到 `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。通过外部控制，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。

即使 `fold` 调用 `walk`，`walk` 调用 `fold` 这样的递归来遍历子目录，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，下面是一个观察三个位图文件（至多）的同时并不逆向递归元数据目录的复杂例子：

```
-- 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
              then count + 1
              else count)
```

传递给 `foldTree` 的初始种子 (`seed`) 为数字零。

练习

1. 修正 `foldTree` 来允许调用改变遍历目录实体的顺序。
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？

代码指南

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。

常规命名类型和变量一般是骆驼法，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，那么你的代码将会对其他人的眼睛造成折磨。

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。

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，通常在和它本行内，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 = []
```

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。

第十章：代码案例学习：解析二进制数据格式

第十章：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。

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。

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，比如当你调对一张照片调用 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)

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进了一步。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
    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
```

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 infixl9（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)                    >>?
  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。

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。

```
-- 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。

```
-- 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。

```
--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。

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。

本文档使用 [看云](#) 构建

```
-- 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。

```
-- file: ch10/Parse.hs
parse :: Parse a -> L.ByteString -> Either String a
parse parser initState
  = case runParse parser (ParseState initState 0) of
      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。

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。

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

Hanging lambdas

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。

```
-- 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。

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。

```
-- file: ch10/Parse.hs
bail :: String -> Parse a
```

本文档使用 [看云](#) 构建


```
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。

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。

```
-- file: ch10/Parse.hs
(==>) :: Parse a -> (a -> Parse b) -> Parse b

firstParser ==> secondParser = Parse chainedParser
  where chainedParser initState =
    case runParse firstParser initState of
      Left errorMessage ->
        Left errorMessage
      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。）

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 `treeMaplength` 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*(fix link) 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 `fa->fb` 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
    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.
```

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。

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。

灵活实例

你可能想给 EitherIntb 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*,
   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 EitherInt 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
```

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 fmapid 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
```

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。

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)` 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。

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 =
  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。

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。

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。

1. 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。

第十一章：测试和质量保障

第十一章：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。

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。

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 (\Rightarrow) 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 x
s
```

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，如果我们将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)
```

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是正确的。

```
-- 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。

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。

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
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
      , 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。

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。

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 interspersions 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):
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.
```

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
  }

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
[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。

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 <code>Prettify2</code>	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。

```

9      | Line
10      | Concat Doc Doc
11      | Union Doc Doc
12      deriving (Show, Eq)
13
14 instance Monoid Doc where
15     mempty = empty
16     mappend = (<>)
17
18
19 (<>) :: Doc -> Doc -> Doc
20 Empty <> y = y
21 x <> Empty = x
22 x <> y = x `Concat` y
23

```

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.
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	<div><div></div></div>	100%	8/8	<div><div></div></div>	100%	32/32	<div><div></div></div>
Program Coverage Total	80%	12/15	<div><div></div></div>	100%	8/8	<div><div></div></div>	100%	32/32	<div><div></div></div>

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。

第十三章：数据结构

第十三章：数据结构 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。换句话说来说，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
```

```

    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
```

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")]

-- 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。

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

*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/funcrcs2.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 funcrcs2.hs
```

```
[1 of 1] Compiling Main                ( funcsecs2.hs, interpreted )
Ok, modules loaded: Main.

*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。

扩展示例：/etc/passwd

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 {
    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 conver
t
                -- 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 th
e
        - 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 i
t
                        split delim (tail x)

        -- 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
    "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。

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 [数字类型](#) 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
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 `Numa=>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> rpShow $ 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就要实现的 num.hs。

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 $\text{Num} \Rightarrow a \rightarrow a \rightarrow a$ 。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，由于 Num 约束的存在，a 可替换为任何 Num 实例。我们可以有 SymbolicManipInt 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
*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 SymbolicManipInt 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。

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
    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。

```

-- 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。

```
-- 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：

```
-- 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。

```
-- 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。

```

-- 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 1)
      | 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
rad2deg x = 360 * x / (2 * pi)

```

首先我们定义了 `units`，使表达式更简洁。`units5"m"` 肯定要比 `Units5(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，以使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。

练习

1. 扩展 `prettyShow` 函数，去掉不必要的括号。

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面的话，它并不会非常好用。我们需要把它转成一种更好用的形式。

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`

dfolder :: (a -> b -> b) -> b -> DList a -> b
dfolder 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
```

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 $a(bc)$ 和 $(ab)*c$ 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： $ae==a$ 和 $ea==a$ 。

么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么半群。如果我们把加号作为二元操作符，0作为单位元素，那么整数就是一个么半群。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么半群。

Haskell 中么半群无所不在。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，列表就形成了一个么半群。

```
-- file: ch13/Monoid.hs
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

由于列表和 DLists 关系如此紧密，DList 类型也必须是一个么半群。

```
-- file: ch13/DList.hs
instance Monoid (DList a) where
    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么半群，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。

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 [
本文档使用 看云 构建
```

```
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。

第十八章：Monad变换器

第十八章：Monad变换器

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]

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 Writerw 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 Writerwma。

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)]IOa 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。]

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，ReaderT r m 是个 MonadReader 的实例，同时 ReaderT 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 `ReaderTm` 和其他类型类的实例，这里是其中的一些：

```
-- 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。

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monad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：

```
-- 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。

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]Appa` 但拒绝 `WriterT[String]App2a`。

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 `typeApp=...` 中，没有类型参数；而 `App2` 却是个部分应用，因为在其定义 `typeApp2a=...`

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。

隐藏细节

本文档使用 [看云](#) 构建

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 newtypeMyAppa 只是乔装过的 ReaderTAppConfig(StateTAppStateIO)a。因此我们可以列出 MyAppa 这个monad栈的全貌（自顶向下）：

- ReaderTAppConfig (monad变换器)
- StateTAppState (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 newtypeMyAppa 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。]

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 constrainedCount 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]

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：

```
-- 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：

```
-- 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。

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：

```
-- 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 StateTInt，因为这个实例在monad栈顶。

```
-- 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：

```
-- 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。

```
-- 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。

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 ma 的类型参数包装在 Maybe 中，以得到类型 m(Maybea)。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(Maybea) 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 MaybeTm 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 MaybeTma，因此 MaybeTm 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：

```
-- 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：

```
-- 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
```

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：

```
-- 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：`MaybeT`。进而我们通过 `MaybeT` 藏起下层monad。

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：

```
-- 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。

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。

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 `EitherTs`（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 `EitherString` 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。

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 `StateTInt(StateString)` 和类型 `StateTString(StateInt)` 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：

```
-- 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？

```
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。

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。

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)
```

```
(foldl1 :: (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。]

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 runWriterTa==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。]

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。

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。

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。]

第十九章：错误处理

第十九章：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。

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。

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 = 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:Tip中那段代码编译不过]

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 take5(tail[1..]) 以及一个类似的safeTail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
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](#)

[<http://rwh.readthedocs.org/en/latest/chp/14.html>]。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"
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]
*** 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，`ghci` 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 `EitherString` 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。

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es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 EitherStringa 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：

```
-- 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。

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。

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 IOa->IO(EitherExceptiona)。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(printy) 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 printy 的返回值为 () 并且没有抛出异

常。

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(returnundefined)` 的结果应该是 `Rightundefined`。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 `printx` 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`。

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。

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 0ghci> let y = 5 div 1ghci> safePrint xCaught error:
divide by zeroghci> safePrint y5
```

`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 -> MaybeArithException` 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
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。

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](http://rwh.readthedocs.org/en/latest/chp/7.html#handle) [http://rwh.readthedocs.org/en/latest/chp/7.html#handle] 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
    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。

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 `IOa` 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 -> IOa` 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 -> IOa`。当你想生成任意的I/O相关的异常的时候可以使用它。

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 [第二十一章使用数据库](http://book.realworldhaskell.org/read/using-databases.html) <http://book.realworldhaskell.org/read/using-databases.html> 中，你会看到HDBC数据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` 用于捕获一个 `SqlError` 异常。注意常规的 `catch` 和 `handle` 函数无法捕获我们的 `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 `Typeableexception=>IOa->(exception->IOa)->IOa`。这里我们简单地限定了它的类型使得它只捕获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", seNati
```



```

veError = 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。

练习

1. 将 Either 修改成 Maybe 例子中的那种风格，使它保存惰性。

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](http://rwh.readthedocs.org/en/latest/chp/18.html#id9) [http://rwh.readthedocs.org/en/latest/chp/18.html#id9] 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
               -> (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)
```

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
    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
    (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")
```

级习

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

这里我们使用的是整数的除法，因此 <code>50 / 8</code> 显示是 6 而不是 6.25。在这个例子中我们没有
--

	使用浮点算术是因为对一个 Double 除以零会返回一个特殊的 Infinity 而不是一个错误。
[39]	关于 Maybe 的介绍，参考 <让过程更可控的方法 [http://rwh.readthedocs.org/en/latest/chp/3.html#id2] –
[40]	更多关于 Either 的信息，参考 <通过 API 设计进行错误处理 [http://rwh.readthedocs.org/en/latest/chp/8.html] –
[41]	在一些其它语言中，抛出异常是叫做 raising 。
[42]	可以手动继承 Typeable 实例，但是那样很麻烦。

第二十章：使用 Haskell 进行系统编程

第二十章：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。

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
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
```

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](http://hackage.haskell.org/package/directory-1.0.0.0/docs/System-Directory.html) [http://hackage.haskell.org/package/directory-1.0.0.0/docs/System-Directory.html] 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。

```
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.bulldata","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$elemc[".",".."])`。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"
```

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 `exitWithExitSuccess` 表示程序执行成功（POSIX 系统中的 0）。或者可以调用 `exitWith(ExitFailure5)`，表示将在程序退出时向系统返回 5 作为状态码。

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。

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。

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` 返回值中将秒数取出来。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)
    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, c
tMin = 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, c
tMin = 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, c
tMin = 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, c
tMin = 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。

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))

-- | 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 。 [\[45\]](#)

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 工具实现同样的事情。

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。

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，不关是在命令上敲 `ls` 还是在 Haskell 中使用 `rawSystem`，其内部机理都是调用 `forkProcess`，`executeFile`，和 `getProcessStatus`（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
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 o
    getExitStatus :: IO ProcessStatus -- ^ IO action that yields exit
    result
}

{- | The type for handling global lists of FDs to always close in the cli
ents
-}
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 progra
m.

            (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

            -- 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 getProcessS
tatus"
        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

-- 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 removethe
m)

where
    procfdlist fdlist [] = fdlist
    procfdlist fdlist (x:xs) = procfdlist (removefd fdlist x) xs

    -- We want to remove only the first occurrence of 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
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
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。例子中的每条命令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。

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
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 o
utput
    getExitStatus :: IO ProcessStatus -- ^ IO action that yields exit
result
}

{- | The type for handling global lists of FDs to always close in the cli
ents
-}
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
```

```

m.      -- to handle stdout. We do not redirect stderr in this progra
      (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

      -- 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 getProcessS
tatus"
               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,
           -- 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 occurrence of 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 =
        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.

* 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

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

-----
-- 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->IOString 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-compatible-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

```

ghci> run $ "ls /usr/local/bin" -|- grep "m.*ap" -|- "tr a-z A-Z" :: IO S
tring
"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\n
texbin\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

```

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

也有一个 <code>system</code> 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
--

[43]	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。
[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。
[45]	POSIX 系统上通常无法设置 ctime。
[46]	线程是一个主要例外，其不会被复制，所以说“几乎”。
[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。

第二十一章：数据库的使用

第二十一章：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。

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[\[49\]](#)。

[\[49\]](#)

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。

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。

[\[50\]](#)

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。

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](http://hackage.haskell.org/) [<http://hackage.haskell.org/>][51](#)获得，本章将使用 1.1.3 版本的 HDBC 作为示例。

[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。

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 后端驱动的连接，针对 Sqlite 3 的驱动也可以通过 Hackage 下载到。

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](#) [<http://software.complete.org/static/hdbc-odbc/doc/HDBC-odbc/>]里面找到。

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：


```

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

```

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。

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。

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。

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 INSERTINTOtestVALUES(0,'zero') 和 INSERTINTOtestVALUES(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 "Idon'tlike1"，那么大多数 SQL 数据库都会要求我们把这个字符串写成 Idon''tlike1'，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。

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。

[\[52\]](#)

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。

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，只不过本文档使用 [看云](#) 构建

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：

```
-- 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。 -}
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 `MaybeString`。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

```

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

'fetchAllRows' 的函数，它在返回每个列的数据之前，会先将它们转换为 MaybeString。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。

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，以使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容。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。

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`。

错误处理

`HDBC` 在错误出现时会引发异常，异常的类型为 `SQLException`。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 `SQLException`，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` 返回的错误信息并没有说清楚这一点。通过使用 `handleSQLException` 辅助函数，我们可以捕捉 `SQLException` 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 `handleSQLException` 处理异常的例子：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> handleSQLException $ quickQuery' conn "SELECT * from test2" []
*** Exception: user error (SQL error: SQLException {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。

第二十二章：扩展示例 —— Web 客户端编程

第二十二章：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）。这个播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。

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) 的基本信息有哪些。一般来说，抓取器关心的都是记录用户感兴趣的博文的信息，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)
```

这些信息将被储存在数据库里面。通过为每个播客和博文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“博文 URL 改变”这类情况。

数据库

接下来，我们需要编写代码，以便将信息永久地储存在数据库里面。我们最感兴趣的，就是通过数据库，将 PodTypes.hs 文件定义的 Haskell 结构中的数据储存在硬盘里面。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。

```
-- 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
T,\
               \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。

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 castUR
L = ?"
        [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 exis
t?\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 ()

```

{- | 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
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})

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

```

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](http://www.cs.york.ac.uk/fp/HaXml/) [http://www.cs.york.ac.uk/fp/HaXml/]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

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)

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)
```

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 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。

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。 -}
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。

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
    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 `runrunghcSetup.hsinstall`。

第二章：Socket 和 Syslog

第二章：Socket 和 Syslog 基本网络

本书的前几张，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。

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。

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
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 hand
le
      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)

{- | 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) mess
    | AUTH                   -- ^ Authentication or security m
    | SYSLOG                 -- ^ Internal syslog messages
    | LPR                    -- ^ Printer messages
    | NEWS                   -- ^ Usenet news
    | UUCP                   -- ^ UUCP messages
    | CRON                   -- ^ Cron messages
    | AUTHPRIV               -- ^ Private authentication messa
    | FTP                    -- ^ FTP messages
    | LOCAL0
    | LOCAL1
    | LOCAL2
    | 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

```

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 = SocketAddr -> 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

    -- 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:permissiondenied` 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。

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。

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。

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

    -- 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 connected"
       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
  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 connected
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。

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
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
le
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"
ghci> syslog sl USER INFO "This is my TCP message"
ghci> syslog sl USER INFO "This is my TCP message again"

```

```
ghci> closelog s1
```

结束时，服务器上将看到这样的输出：

```
From 127.0.0.1:46319: syslogtcpserver.hs: client connected
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
```

是优先级和设施代码，和之前 UDP 例子中的意思一样。

第二十八章：软件事务内存 (STM)

第二十八章：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。

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。

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可能的纯函数化。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。

```
-- 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
```

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

```
-- 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。

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)
  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`。

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。

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 `returnFalse` 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
```

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
  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。

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/STMI0.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。

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，`newtypeTMVara=TMVar(TVar(Maybea))`，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。]

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,
      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
    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
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)的连接打印到 `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)
```

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)
  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

-- 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
```

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)
```

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。

```
-- 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。

```
-- 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HTML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:"

```

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。

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。

```
-- 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。

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。

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。

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。

```
-- 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

```
-- 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。

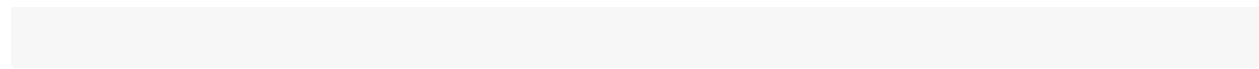
```
-- file: ch28/GameInventory.hs
consistentBalance :: [Player] -> STM (STM ())
consistentBalance players = do
  initialTotal <- totalBalance
  return $ do
    curTotal <- totalBalance
    when (curTotal /= initialTotal) $
      error "inconsistent global balance"
  where totalBalance = foldM addBalance 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
```



翻译约定

翻译约定

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 branche

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

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

第四章

折叠 fold

收集器 collection

主递归 primitive recursive

部分应用 parital application

部分函数应用 parital 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

第五章

导出 export

本地码 native code

目标代码 object code

指令 directive

顶层 top-level

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

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

第八章

字符类 character class

多态 polymorphism

第十九章

错误处理 error handling

惰性求值 lazy evaluation

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 fineo-grained

簿记 book-keeping