Learn Real World Haskell

Jacob Bishop

2023-08-20

1 Getting started

略

2 Types and functions

略

3 Defining types, streamlining functions

略

4 Functional programming

略

5 Writing a library: working with JSON data

在 Haskell 中表示 JSON

首先是在 Haskell 中定义 JSON 这个数据，这里使用代数数据类型来表达 JSON 类型的范围。

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

对于每种 JSON 类型，我们都提供了独立的值构造函数。测试：

```
ghci> :l SimpleJSON
[1 of 1] Compiling Main             ( SimpleJSON.hs, interpreted )
Ok, one module loaded.
ghci> JString "foo"
JString "foo"
ghci> JNumber 2.7
JNumber 2.7
ghci> :type JBool True
JBool True :: JValue
```

构造一个从 `JValue` 获取字符串的函数：

```
getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _ = Nothing
```

测试：

```
ghci> :reload
[1 of 1] Compiling Main             ( SimpleJSON.hs, interpreted )
Ok, one module loaded.
ghci> getString (JString "hello")
Just "hello"
ghci> getString (JNumber 3)
Nothing
```

接下来是其它类型的访问函数：

```
getInt (JNumber n) = Just n
getInt _ = Nothing

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

```
6
7  getBool (JBool n) = Just n
8  getBool _ = Nothing
9
10 getObject (JObject o) = Just o
11 getObject _ = Nothing
12
13 getArray (JArray a) = Just a
14 getArray _ = Nothing
15
16 isNull v = v == JNull
```

`truncate` 函数可以让浮点类型或者有理数去掉小数点后变为整数：

```
1 ghci> truncate 5.8
2 5
3 ghci> :module +Data.Ratio
4 ghci> truncate (22 % 7)
5 3
```

Haskell 模块详解

一个 Haskell 源文件包含了单个模块的定义。模块允许我们在它其内部进行定义，并由其它模块访问：

```
1  module SimpleJSON
2    ( JValue (..),
3      getString,
4      getInt,
5      getDouble,
6      getBool,
7      getObject,
8      getArray,
9      isNull,
10   )
11 where
```

如果省略了导出（即圆括号以及其所包含的名称），那么该模块中的所有名称都会被导出。

编译 Haskell 源

编译一个源文件：

```
1 ghc -c SimpleJSON.hs
```

`-c` 选项告诉 **ghc** 仅生成对象代码。如果省略了该选项，那么编译器则会尝试生成一整个可执行文件。这会导致失败，因为我们并没有一个 `main` 函数，即 GHC 所认为的一个独立程序的执行入口。

编译后会得到两个新文件：`SimpleJSON.hi` 与 `SimpleJSON.o` 。前者是一个接口 *interface* 文件，即 **ghc** 以机器码的形式存储模块导出的名称信息；后者是一个对象 *object* 文件，其包含了生产的机器码。

生成一个 Haskell 程序，导入模块

添加一个 `Main.hs` 文件，其内容如下：

```
module Main where

import SimpleJSON

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

与原文 `module Main () where` 的不同之处在于，现在的 `Main` 后不再需要一个 `()` 。

接下来是编译 `main` 函数：

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

与原文 `ghc -o simple Main.hs SimpleJSON.o` 不同，现在没了 `SimpleJSON.o` 这个文件，加上后会报错重复 symbol 的编译错误（因为在 `Main.hs` 中已经做了 `import SimpleJSON` 导入了）。

这次省略掉了 `-c` 选项，因此编译器尝试生成一个可执行。生成可执行的过程被称为链接 *linking*（与 C++ 一样），即在一次编译中链接源文件与可执行文件。

这里给了 **ghc** 一个新选项 `-o` ，其接受一个参数：可执行文件的名称，这里是 `simple` ，执行它：

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

打印 JSON 数据

现在我们希望将 Haskell 的值渲染成 JSON 数据，创建一个 `PutJSON.hs` 文件：

```
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)
```

好的 Haskell 风格需要分隔纯代码与 I/O 代码。我们的 `renderJValue` 函数不会与外界交互，但是仍然需要一个打印的函数：

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

类型推导是把双刃剑

假设我们编写了一个自认为返回 `String` 的函数，但是并不为其写类型签名：

```
upcaseFirst (c:cs) = toUpper c -- forgot ":cs" here
```

这里希望单词首字母大写，但是忘记了将剩余的字符放进结果中。我们认为函数的类型是 `String -> String` ，但是编译器则会将其视为 `String -> Char` 。假设我们尝试在其他地方调用该函数：

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

那么当我们尝试编译该代码或者是加载进 **ghci**，我们并不会得到明显的错误信息：

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

Trouble.hs:9:27:
    Couldn't match expected type `[Char]' against inferred type `Char'
      Expected type: [Char] -> [Char]
      Inferred type: [Char] -> Char
    In the first argument of `map', namely `upcaseFirst'
    In the first argument of `concat', namely
        `(map upcaseFirst (words xs))'
Failed, modules loaded: none.
```

注意这里的报错是在 `upcaseFirst` 函数处，那么假设我们认为 `upcaseFirst` 的定义与类型是正确的，那么查找到真正的错误可能会花掉一些时间。

更加泛用的渲染

我们将更为泛用的打印模块称为 `Prettify` ，那么其源文件即 `Prettify.hs` 。

为了使 `Prettify` 满足实际需求，我们还要一个新的 JSON 渲染器来使用 `Prettify` 的 API。在 `Prettify` 模块中将使用一个抽象类型 `Doc` 。基于建立在抽象类型的泛用渲染库，我们可以选择灵活高效的实现。

`PrettyJSON.hs` 示例：

```
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
```

这里的 `text` ， `double` 以及 `string` 都会在 `Prettify` 模块中提供。

开发 Haskell 代码而不发疯

一个用于快速开发程序框架的技巧就是编写占位符，或者类型与函数的根 *stub* 版本。例如上述 `string` ， `text` 以及 `double` 函数将被 `Prettify` 模块提供。如果我们没有提供这些函数或者 `Doc` 类型，那么“早点编译，经常编译”这个尝试就会失败。为了避免这个问题现在让我们编写一个不做任何事情的根程序。

```
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` 有一个 `a` 类型，无论在哪里使用它，总是会有类型检查。如果尝试计算，则会使程序崩溃：

```
ghci> :type undefined
undefined :: a
ghci> undefined
*** Exception: Prelude.undefined
ghci> :type double
double :: Double -> Doc
ghci> double 3.14
*** Exception: Prelude.undefined
```

尽管还不能运行根代码，但是编译器的类型检查器可以确保我们的程序类型正确。

漂亮的打印字符串

当需要打印一个漂亮的字符串时，我们必须遵循 JSON 的转义规则。字符串就是一系列被包裹在引号中的字符们。`PrettyJSON.hs` ：

```
string :: String -> Doc
string = enclose '"' '"' . hcat . map oneChar
```

以及 `enclose` 函数将一个 `Doc` 值简单的包裹在一个开始与结束字符之间：

```
enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right
```

这里提供的 `<>` 函数定义在 `Prettify` 库中，它需要两个 `Doc` 值，即 `Doc` 版本的 `(++)` 。还是先在根文件中定义：

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

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

我们的库还需要提供 `hcat` ，将若干 `Doc` 值合成成为一个（类似于列表的 `concat` ）：

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

我们的 `string` 函数应用 `oneChar` 函数到字符串中的每个字符，将它们连接，然后用引号包装。而 `oneChar` 函数则是用来转义或包装一个独立的字符。在 `PrettyJSON.hs` 中：

```
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])
```

这里的 `simpleEscapes` 是一个列表的二元组，我们称其为关联 *association* 列表，或者简称 `alist` 。每个 `alist` 的元素都将字符关联了它的转义表达，测试：

```
ghci> take 4 simpleEscapes
[('\b',"\\b"),('\n',"\\n"),('\f',"\\f"),('\r',"\\r")]
```

我们的 `case` 表达式尝试查看字符是否匹配关联列表。如果匹配则返回匹配值，如若不然则需要以更复杂的方式来转义该字符。只有当两种转义都不需要时，才会返回普通字符。保守起见，我们输出的唯一未转义字符是可打印的 ASCII 字符。

更复杂的转义包含了将一个字符转为字符串 `"\u"` 并跟随四个十六进制的字符用于表达 Unicode 字符的数值。仍然是 `PrettyJSON.hs` ：

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

这里的 `showHex` 函数需要从 `Numeric` 库中加载，其用于返回一个值的十六进制：

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

`replicate` 函数则是由 Prelude 提供：

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

`smallHex` 提供的四位编码只能表示最大 `0xffff` 的 Unicode 字符，而有效的 Unicode 字符的范围可以达到 `0x10ffff` 。为了正确的表达一个超出了 `0xffff` 的 JSON 字符串，我们遵循一些复杂的规则将其分为两部分。这使我们有机会对 Haskell 的数执行一些位级操作。还是 `PrettyJSON.hs` ：

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

这里 `shiftR` 函数和 `(.&.)` 函数都源自 `Data.Bits` 模块，前者将一个数移动右一位，后者则是执行一个字节层面的两值 *and* 操作。

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

现在有了 `smallHex` 与 `astral` ，我们可以提供 `hexEscape` 的定义了：

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

```
    where
      d = ord c
```

其中 `ord` 由 `Data.Char` 模块提供。

数组与对象，以及模块头

相比于字符串的漂亮打印，数组和对象就是小菜一碟了。我们已经知道了它们两者其实很相似：都是由起始字符开始，接着是一系列由逗号分隔的值，最后接上结束字符。让我们编写一个函数捕获数组与对象的共同结构：

```
series :: Char -> Char -> (a -> Doc) -> [a] -> Doc
series open close item = enclose open close . fsep . punctuate (char ',') . map item
```

让我们首先从函数类型开始。它接受起始与结束字符，一个打印某些未知类型 `a` 值的函数，以及一个类型为 `a` 的列表，返回一个类型为 `Doc` 的值。

注意尽管我们的类型签名提及了四个参数，在函数定义中仅列出了三个。这遵循了简化定义的规则，如 `myLength xs = length xs` 等同于 `myLength = length` 。

我们已经有了之前编写过的 `enclose` ，即包装一个 `Doc` 值进起始与结束字符之间。那么 `fsep` 函数则位于 `Prettify` 模块中，其结合一个 `Doc` 值列表成为一个 `Doc` ，在输出不适合单行的情况下还需要换行。

```
fsep :: [Doc] -> Doc
fsep xs = undefined
```

那么现在，遵循上述提供的例子，你应该能够定义你自己的 `Prettify.hs` 的根文件了。这里不再显式的定义更多的根了。

`puctuate` 函数同样位于 `Prettify` 模块中：

```
punctuate :: Doc -> [Doc] -> [Doc]
punctuate p [] = []
punctuate p [d] = [d]
punctuate p (d : ds) = (d <> p) : punctuate p ds
```

通过 `series` 的定义，漂亮打印一个数组就非常直接了：

```
renderJValue (JArray ary) = series '[' ']' renderJValue ary
```

对于对象而言，还需要额外的一些工作：每个元素同时需要处理名称与值：

```
renderJValue (JObject obj) = series '{' '}' field obj
where
  field (name, val) =
    string name
      PrettyStub.<> text ": "
      PrettyStub.<> renderJValue val
```

编写一个模块头

现在已经有了 `PrettyJSON.hs` 文件，我们需要回到其顶部添加模块声明：

```
module PrettyJSON (renderJValue) where
```

这里导出了一个名称：`renderJValue` ，也就是我们的 JSON 渲染函数。模块中其它的定义都是用于支持 `renderJValue` 的，因此没有必要对其它模块可见。

充实我们的漂亮打印库

在 `Prettify` 模块中，提供了 `Doc` 类型作为一个代数数据类型：

```
data Doc
  = Empty
  | Char Char
  | Text String
  | Line
  | Concat Doc Doc
  | Union Doc Doc
  deriving (Show)
```

观察可知 `Doc` 类型实际上是一颗树。`Concat` 与 `Union` 构造函数根据其它两个 `Doc` 值创建一个内部节点，而 `Empty` 以及其它简单的构造函数用于构建叶子。

在模块的头部，我们导出该类型的名称，而不是它们的构造函数：这样可以防止使用 Doc 的构造函数被用于创建与模式匹配 `Doc` 值。

相反的，要创建一个 `Doc` ，用户需要调用我们在 `Prettify` 模块中所提供的函数：

```
empty :: Doc
empty = Empty

char :: Char -> Doc
char = Char

text :: String -> Doc
text "" = Empty
text s = Text s

double :: Double -> Doc
double = text . show
```

`Line` 构造函数代表一个换行，其创建的是一个 *hard* 换行，即总是会在漂亮的打印中出现。有时我们想要一个 *soft* 换行，即只会在窗口或者页面上过长显示时才会换行。稍后将会介绍 `softline` 函数。

```
line :: Doc
line = Line
```

另外就是用于连接两个 `Doc` 值的 `(<>)` 函数（这里使用 `(<+>)`，因为 `(<>)` 在 Prelude 中已经有定义了）：

```
(<+>) :: Doc -> Doc -> Doc
Empty <+> y = y
x <+> Empty = x
x <+> y = x `Concat` y
```

测试：

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

接下来是用于连接 `Doc` 列表的 `hcat` 函数：

```
hcat :: [Doc] -> Doc
hcat = fold (<+>)

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

以及 `fsep` 函数，它还依赖若干其它函数：

```
fsep :: [Doc] -> Doc
fsep = fold (</>)

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

softline :: Doc
softline = group line
```

这里需要解释一下，`softline` 函数应该在当前行特别宽的时候进行换行，否则插入空格。如果我们的 `Doc` 类型不包含任何关于渲染的信息，那么该如何呢？答案就是每次遇到一个软换行，通过 `Union` 构造函数来维护两个可选项：

```
group :: Doc -> Doc
group x = flatten x `Union` x
```

`flatten` 函数将一个 `Line` 替换为空格，将两行转换为一个更长的行。

```
flatten :: Doc -> Doc
flatten (x `Concat` y) = flatten x `Concat` flatten y
flatten Line = Char ' '
flatten (x `Union` _) = flatten x
flatten other = other
```

注意总是调用 `flatten` 在 `Union` 的左元素上：每个 `Union` 的左侧总是大于等于右侧宽度（字符距离）。

紧密渲染

我们需要频繁的使用包含尽可能少字符的数据。例如通过网络连接发送 JSON 数据就没有必要美观：软件并不在乎数据的美观与否，添加很多空格只会带来性能下降。

因此我们提供了一个紧密渲染的函数：

```
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` 函数将其参数包裹成一个列表，然后将帮助函数 `transform` 应用至该列表。`transform` 函数视其参数为堆叠的项用于处理，列表的第一个元素即堆的顶部。

`transform` 函数的 `(d:ds)` 模式将堆顶部的元素取出 `d` 并留下 `ds` 。在 `case` 表达式中，前面几个分支都在 `ds` 上递归，每次递归消费堆顶部的元素；最后两个分支则是在 `ds` 之前添加项：`Concat` 添加两个元素至堆，而 `Union` 分支则忽略它左侧元素，调用的是 `flatten` ，再将其右侧元素至堆。

测试 `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
}
```

为了更好的理解代码是如何运作的，让我们用一个更简单的例子来展示细节：

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

当我们应用 `compact` 时，它会将它的参数转为一个列表后应用函数 `transform` 。

- 接下来 `transform` 函数接受了一个单例列表，然后进行模式匹配 `(d:ds)` ，这里的 `d` 是 `Concat (Char 'f') (Text "oo")` ，而 `ds` 则是一个空列表。

  由于 `d` 的构造函数是 `Concat` ，其模式匹配就在 `case` 表达式中。那么在右侧，添加 `Char 'f'` 与 `Text "oo"` 值堆，然后递归的应用 `transform` 。

- – `transform` 函数接受了一个包含两项的列表，继续匹配 `d:ds` 模式。此时变量 `d` 绑定到了 `Char 'f'` ，而 `ds` 则是 `[Text "oo"]` 。

  `case` 表达式匹配到了 `Char` 分支。那么在右侧，使用 `(:)` 来构建一个列表，其头部为 `'f'` ，其余部分则是递归应用 `transform` 后的结果。

  – * 递归的调用接受到一个单例列表，其变量 `d` 绑定至 `Text "oo"` ， `ds` 绑定至 `[]` 。

    `case` 表达式匹配 `Text` 分支。那么在右侧，使用 `(++)` 来连接 `"oo"` 与递归调用 `transform` 后的结果。

    * · 最后的调用，`transform` 得到一个空列表，返回一个空字符串。

    * 结果是 `"oo" ++ ""`

  – 结果是 `'f' : "oo" ++ ""`

真实的漂亮打印

我们的 `compact` 函数对于机器之间的交流是有帮助的，但是它的结果对人类而言并不易读：每行的信息很少。为了生成一个更可读的输出，我们将要编写另一个函数 `pretty` 。相比于 `compact` ， `pretty` 接受一个额外的参数：一行的最大宽度。

```
pretty :: Int -> Doc -> String
```

确切来说，`Int` 参数控制了 `pretty` 在遇到一个 `softline` 时的行为。在一个 `softline` 时， `pretty` 会选择继续留在当前行还是另起一行。其余情况下，必须严格遵守漂亮打印函数所设定的指令。

以下是实现的代码：

```
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
```

`best` 帮助函数接受两个参数：当前行使用了的列数，以及剩余的仍需处理的 `Doc` 列表。

在简单的情况下，随着消费输入 `best` 直接更新了 `col` 变量。其中 `Concat` 情况也很明显：将两个连接过的部分推至堆叠，且不触碰 `col` 。

有趣的情况在 `Union` 构造函数。回想一下之前将 `flatten` 应用至左侧元素，且不对右侧做任何操作。还有就是 `flatten` 将新行替换成空格。因此我们则需要检查这两种布局，`flatten` 后的还是原始的，更适合我们的宽度限制。

为此需要编写一个小的帮助函数来决定 `Doc` 值的一行是否适合给定的长度：

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

遵循漂亮打印

为了理解代码如何工作的，首先考虑一个简单的 `Doc` 值：

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

我们将应用 `pretty 2` 在该值。当我们首先应用 `best` ， `col` 值为零。它匹配 `Concat` 模式，接着将 `Union (Char ' ') Line` 与 `Char 'a'` 推至堆，接着是递归应用自身，它匹配了 `Union (Char ' ') Line` 。

现在忽略 Haskell 通常的计算顺序，两个子表达式，`best 0 [Char ' ', char 'a']` 与 `best 0 [Line, Char 'a']` ，前者计算得到 `" a"` ，而后者得到 `"na "` 。接着将它们替换到外层的表达式中，得到 `nicest 0 " a" "\na"` 。

为了明白 `nicest` 的结果，我们做一个小小的替换。`width` 以及 `col` 分别是 0 与 2，那么 `least` 就是 0，`width - least` 就是 2。这里用 **ghci** 来计算一下 `2 `fits` " a"` ：

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

计算得到 `True` ，那么这里的 `nicest` 结果就是 `" a"` 。

如果将 `pretty` 函数应用到之前同样的 JSON 数据上，可以看到根据提供的最大长度，它会得到不同的结果：

```
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 }
```

创建一个库

Haskell 社区构建了一个标准工具库，名为 Cabal，其用于构建，安装，以及分发软件。Cabal 以 `包package` 的方式管理软件，一个包包含了一个库，以及若干可执行程序。

编写一个包的描述

要使用包，Cabal 需要一些描述。这些描述保存在一个文本文件中，以 `.cabal` 后缀命名。该文件应位于项目的根目录。

包描述由一系列的全局属性开始，其应用于包中所有的库以及可执行。

```
name:               pretty-json
version:            0.1.0.0
```

包名称必须是唯一的。如果你创建并安装了一个同名包在你的系统上，GHC 会感到迷惑。

```
synopsis:           My pretty printing library, with JSON support
description:
  A simple pretty printing library that illustrates how to
  develop a Haskell library.
author:             jacob xie
maintainer:         jacobbishopxy@gmail.com
```

这里还要有 license 信息，大多数 Haskell 包都使用 BSD license，Cabal 称为 BSD3。

另外就是 Cabal 的版本：

```
cabal-version:      2.4
```

在一个包中要描述一个独立的库，需要 *library* 这个部分。注意缩进在这里很重要。

```
library
  default-language: Haskell2010
  build-depends:    base
  exposed-modules:
    Prettify
    PrettyJSON
    SimpleJSON
```

`exposed-modules` 字段包含了一个模块列表，用于暴露给使用该包的用户导入。另一个可选字段 `other-modules` 包含了一个 内部 *internal* 模块的列表，它们提供给 `exposed-modules` 内的模块使用，而对用户不可见。

`build-depends` 字段包含了一个逗号分隔的包列表，它们是我们库所需的依赖。`base` 包中包含了 Haskell 很多核心模块，例如 Prelude，所以它总是必须的。

GHC 的包管理器

GHC 包含了一个简单的包管理器用于追踪安装了哪些包，以及这些包的版本号。一个名为 **ghc-pkg** 的命令行工具提供了包数据库的管理。

这里说*数据库 database* 是因为 GHC 区分了*系统级别*的包，即对所有用户可用；以及用户可见的包，即仅对当前用户可用。后者可以避免管理员权限来安装包。

ghc-pkg 提供了不同的子命令，多数时候我们仅需两个命令：`ghc-pkg list` 用于查看已安装的包；当需要删除包时则使用 `ghc-pkg unregister` 。

设置，构建与安装

除了一个 `.cabal` 文件，一个包还必须包含一个 *setup* 文件。在包需要的情况下，它允许 Cabal 的构建过程中包含大量的定制。

`Setup.hs` 示例：

```
#!/usr/bin/env runhaskell

import Distribution.Simple

main = defaultMain
```

一旦有了 `.cable` 与 `Setup.hs` 文件，那么就剩下三步了。

指导 Cabal 如何构建以及在哪里安装包，只需一个简单命令：

```
runghc Setup configure
```

这可以确保我们所需要的包都是可用的，并且保存设定为了之后的 Cabal 命令。

如果没有为 `configure` 提供任何参数，Cabal 则会将包安装在系统层的包数据库。要在 home 路径安装则需要提供一些额外的信息：

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

接下来就是包的构建：

```
runghc Setup build
```

如果成功了，我们就可以安装这个包了。我们无需指定其安装的位置：Cabal 会使用我们在 `configure` 步骤中所提供的配置。即安装在我们自己的路径，并更新 GHC 的用于层包数据库。

```
runghc Setup install
```

6 Using typeclasses

Typeclasses 是 Haskell 中最强大的特性。它们允许我们定义通用性的接口，为各种类型提供公共特性集。Typeclasses 是一些语言特性的核心，例如相等性测试和数字运算符。

对 Typeclasses 的需求

假设在没有相等性测试 `==` 的情况下需要构建一个简单的 `color` 类型，那么相等测试就应该如下：

```
data Color = Red | Green | Blue

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

现在假设我们想为 `StringS` 增加一个相等性测试。由于 Haskell 的 `String` 是字符列表，我们可以编写一个简单的函数用于测试。这里为了简化我们使用 `==` 操作符用于说明。

```
stringEq :: [Char] -> [Char] -> Bool
stringEq [] [] = True
stringEq (x:xs) (y:ys) = x == y && stringEq xs ys
stringEq _ _ = False
```

现在已经发现问题了：我们必须为每个不同的类型使用不同名称的比较函数，这是很低效且令人讨厌的。因此需要一个通用的函数可用于比较任何东西。此外，当新的数据类型之后被创建时，已经存在的代码不能被改变。

Haskell 的 typeclasses 就是设计用来解决上述问题的。

什么是 typeclasses

Typeclasses 定义了一系列的函数，它们可以根据给定的数据类型有不同的实现。

首先我们必须定义 typeclass 本身。我们希望一个函数接受同样类型的两个参数，返回一个 `Bool` 来表示它们是否相等。我们无需在意类型是什么，但需要两个参数的类型相同。下面是 typeclass 的第一个定义：

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

通过 **ghci** 的类型检查 `:type` 可以得知 `isEqual` 的类型：

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

现在可以为特定类型定义 `isEqual` ：

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

测试：

```
ghci> isEqual False False
True
ghci> isEqual False True
False
ghci> isEqual "Hi" "Hi"

<interactive>:1:0:
    No instance for (BasicEq [Char])
      arising from a use of `isEqual' at <interactive>:1:0-16
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "Hi" "Hi"
    In the definition of `it': it = isEqual "Hi" "Hi"
```

注意在尝试比较两个字符串时，**ghci** 发现我们并未给 `String` 提供 `BasicEq` 的实例。因此 `ghci` 并不知道该如何对 `String` 进行比较，同时提议我们可以通过为 `[Char]` 定义 `BasicEq` 实例来解决这个问题。

下面是定义一个包含了两个函数的 typeclass：

```
class BasicEq2 a where
  isEqual2 :: a -> a -> Bool
  isNotEqual2 :: a -> a -> Bool
```

虽然 `BasicEq2` 的定义没有问题，但是它让我们做了额外的事情。就逻辑而言，如果我们知道了 `isEqual` 或 `isNotEqual` 中的一个，我们便知道了另一个。那么与其让用户定义 typeclass 中两个函数，我们可以提供一个默认的实践。

```
class BasicEq3 a where
  isEqual3 :: a -> a -> Bool
  isEqual3 x y = not (isNotEqual3 x y)

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

声明 typeclass 实例

现在知道了如何定义 typeclasses，接下来就是直到如何定义 typeclasses 的实例。回忆一下，类型是由一个个特定 typeclass 所构成的实例所赋予了意义，这些 typeclasses 又实现了必要的函数。

之前为我们的 `Color` 类型创建了相等性测试，现在让我们试试让其成为 `BasicEq3` 的实例：

```
instance BasicEq3 Color where
  isEqual3 Red Red = True
  isEqual3 Green Green = True
  isEqual3 Blue Blue = True
  isEqual3 _ _ = False
```

注意这里提供了与之前定义的一样的函数，实际上实现是完全相同的。不过在这种情况下，我们可以使用 `isEqual3` 在任何定义了 `BasicEq3` 实例的类型上。

另外注意 `BasicEq3` 定义了 `isEqual3` 与 `isNotEqual3` ，而我们仅实现了它们其中一个。这是因为 `BasicEq3` 包含了默认实现，因此没有显式定义 `isNotEqual3` 时，编译器自动的使用了 `BasicEq3` 中的默认实现。

重要的内建 Typeclasses

Show

`Show` typeclass 用于将值转换为 `String` 。可能最常用于将数字转换为字符串，很多类型都有它的实例，因此还可以用于转换更多的类型。如果自定义类型实现了 `Show` 的实例，那么就可以在 **ghci** 上展示或者在程序中打印出来。

`Show` 中最重要的函数就是 `show` ，它接受一个参数：用于转换的数据，返回一个 `String` 来表示该数据。

```
ghci> :type show
show :: (Show a) => a -> String
```

一些其它的例子：

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

ghci 展示的结果与输入到 Haskell 程序中的结果相同。表达式 `show 1` 返回的是单个字符的字符串，其包含数字 `1` ，也就是说引号并不是字符串本身。通过 `putStrLn` 可以更清楚的看到：

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

我们也可以使用 `show` 在字符串上：

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

现在为我们自己的类型实现 `Show` 的实例:

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

Read

`Read` typeclass 基本就是相反的 `Show` : 它定义了函数接受一个 `String` , 分析它, 并返回属于 `Read` 成员的任何类型的数据。

```
ghci> :type read
read :: (Read a) => String -> a
```

以下是一个 `read` 与 `show` 的例子:

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

`read` 的类型: `(Read a) => String -> a` 。这里的 `a` 是每个 `Read` 实例的类型。也就是说特定的解析函数是根据预期的 `read` 返回类型所决定的。

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

在尝试解析 `5.0` 为 `Integer` 时异常。当预期返回值的类型是 `Integer` 时, `Integer` 的解析函数并不接受小数, 因此异常被抛出。

`Read` 提供了一些相当复杂的解析器。你可以通过实现 `readsPrec` 函数定义一个简单的解析。该实现在解析成功时, 返回只包含一个元组的列表, 如果解析失败则返回空列表。下面是一个实现 `Color` 的 `Read` 实例的例子:

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

测试：

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

注意最后一个例子的异常。这是因为我们的解析器并没有聪明到能处理空格。

Tip

Read 并没有大范围的被使用

虽然可以使用 `Read` typeclass 来构建复杂的解析器，但许多人发现使用 Parsec 会更容易，且仅依赖 `Read` 来处理简单的任务。在第 16 章会详细介绍 Parsec。

通过 Read 和 Show 进行序列化

我们经常需要存储一个内存中的数据结构至硬盘供未来使用或者通过网络发送出去，那么将内存中数据转换为一个平坦的字节序列用于存储的这个过程就被称为*序列化 serialization*。

Tip

解析大型字符串

在 Haskell 中字符串的处理通常都是惰性的，因此 `read` 与 `show` 可以被用于处理很大的数据结构而不发生异常。Haskell 内建的 `read` 与 `show` 实例是高效的，并且都是纯 Haskell。如何处理解析异常的更多细节将会在第 19 章中详细介绍。

测试：

```
ghci> let d1 = [Just 5, Nothing, Nothing, Just 8, Just 9]::[Maybe Int]
ghci> putStrLn (show d1)
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> writeFile "test" (show d1)
```

再是反序列：

```
ghci> input <- readFile "test"
"[Just 5,Nothing,Nothing,Just 8,Just 9]"
ghci> let d2 = read input

<interactive>:1:9:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:9-18
    Probable fix: add a type signature that fixes these type variable(s)
ghci> let d2 = (read input)::[Maybe Int]
ghci> print d1
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> print d2
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> d1 == d2
True
```

这里解释器并不知道 `d2` 的类型，因此抛出了异常。

以下是一些稍微复杂点的数据结构：

```
ghci> putStrLn $ show [("hi", 1), ("there", 3)]
[("hi",1),("there",3)]
ghci> putStrLn $ show [[1, 2, 3], [], [4, 0, 1], [], [503]]
[[1,2,3],[],[4,0,1],[],[503]]
ghci> putStrLn $ show [Left 5, Right "three", Left 0, Right "nine"]
[Left 5,Right "three",Left 0,Right "nine"]
ghci> putStrLn $ show [Left 0, Right [1, 2, 3], Left 5, Right []]
[Left 0,Right [1,2,3],Left 5,Right []]
```

数值类型

Haskell 拥有强大的数值类型。

选定的数值类型	
类型	描述
Double	双精度浮点数。浮点数的通常选择。
Float	单精度浮点数。通常用于与 C 交互。
Int	带方向的确定精度整数；最小范围 $[-2^29..2^29-1]$。
Int8	8-bit 带方向的整数
Int16	16-bit 带方向的整数
Int32	32-bit 带方向的整数
Int64	64-bit 带方向的整数
Integer	带方向的任意精度整数；范围仅受机器限制。很常用。
Rational	带方向的任意精度的有理数。以两个 Integers 进行存储。
Word	无方向的确定精度整数；存储大小与 Int 一致
Word8	8-bit 无方向的整数
Word16	16-bit 无方向的整数
Word32	32-bit 无方向的整数
Word64	64-bit 无方向的整数

有很多不同的数值类型。有些运算，比如加法对所有类型都适用；还有其它类型的计算例如 `asin` ，仅适用于浮点类型。

选定的数值函数与常量			
项	类型	模块	描述
(*)	Num a => a -> a -> a	Prelude	加法
(-)	Num a => a -> a -> a	Prelude	减法
(*)	Num a => a -> a -> a	Prelude	乘法
(/)	Fractional a => a -> a -> a	Prelude	除法
(**)	Floating a => a -> a -> a	Prelude	幂
(^)	(Num a, Integral b) => a -> b -> a	Prelude	非负数的幂
(i	(Fractional a, Integral b) => a -> b -> a	Prelude	分数的幂
(%)	Integral a => a -> a -> Ratio a	Data.Ratio	比例
(.&.)	Bits a => a -> a -> a	Data.Bits	Bitwise 和
(.|.)	Bits a => a -> a -> a	Data.Bits	Bitwise 或
abs	Num a => a -> a	Prelude	绝对值
approxRational	RealFrac a => a -> a -> Rational	Data.Ratio	基于分数分子与分母的近似有理组合
cos	Floating a => a -> a	Prelude	Cosine，还有 acos，cosh，与 acosh
div	Integral a => a -> a -> a	Prelude	整数除法，总是向下取整，quot 同理
fromInteger	Num a => Integer -> a	Prelude	从 Integer 类型转换为任意数值类型
fromIntegral	(Integral a, Num b) => a -> b	Prelude	比上述更泛化的转换
fromRational	Fractional a => Rational -> a	Prelude	从 Rational 转换，可能有损
log	Floating a => a -> a	Prelude	自然 log
logBase	Floating a => a -> a -> a	Prelude	显式底的 log
maxBound	Bounded a => a	Prelude	bound 类型的最大值
minBound	Bounded a => a	Prelude	bound 类型的最小值
mod	Integral a => a -> a -> a	Prelude	整数模
pi	Floating a => a	Prelude	数学常数的派
quot	Integral a => a -> a -> a	Prelude	整数除法；商的小数部分向零截断

接上表

项	类型	模块	描述
recip	Fractional a => a -> a	Prelude	倒数
rem	Integral a => a -> a -> a	Prelude	整数除法的余数
round	(RealFrac a, Integral b) => a -> b	Prelude	向最接近的整数方向取整
shift	Bits a => a -> Int -> a	Bits	左移指定 bits，右移可能为负
sin	Floating a => a -> a	Prelude	Sine，另 asin，sinh，与 asinh
sqrt	Floating a => a -> a	Prelude	平方根
tan	Floating a => a -> a	Prelude	Tangent，令 atan，tanh，与 atanh
toInteger	Integral a => a -> Integer	Prelude	转换任意 Integral 至一个 Integer
toRational	Real a => a -> Rational	Prelude	转换 Real 至 Rational
truncate	(RealFrac a, Integral b) => a -> b	Prelude	向零截断数值
xor	Bits a => a -> a -> a	Data.Bits	Bitwise 的异或

数值类型的 Typeclass 实例								
项	Bits	Bounded	Floating	Fractional	Integral	Num	Rea	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	

在数值类型间转换是另一个常用的需求。

数值类型之间的转换				
原始类型	目标类型			
	Double, Float	Int, Word	Integer	Rational
Double, Float	fromRational . toRational	truncate *	truncate *	toRational
Int, Word	fromIntegral	fromIntegral	fromIntegral	fromIntegral
Integer	fromIntegral	fromIntegral	N/A	fromIntegral
Rational	fromRational	truncate *	truncate *	N/A

自动化派生

对于很多简单的数据类型，Haskell 编译器可以自动的为我们派生出 `Read` ，`Show` ，`Bounded` ， `Enum` ， `Eq` 以及 `Ord` 的实例。这将大大的节省用户编写代码的时间。

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

测试：

```
ghci> show Red
"Red"
ghci> (read "Red")::Color
Red
ghci> (read "[Red,Red,Blue]")::[Color]
[Red,Red,Blue]
ghci> (read "[Red, Red, Blue]")::[Color]
[Red,Red,Blue]
ghci> Red == Red
True
ghci> Red == Blue
False
ghci> Data.List.sort [Blue,Green,Blue,Red]
[Red,Green,Blue,Blue]
ghci> Red < Blue
True
```

自动派生也不总是能成功。例如如果定义了一个类型 `data MyType = MyType (Int -> Bool)` ，编译器将不能为其派生 `show` 的实例，因为它不知道如何渲染一个函数。这种情况下我们会得到一个编译错误。

当我们自动派生某些 typeclass 的实例时，在数据声明中引用的类型也必须是该 typeclass 的实例（手动或自动的）。

```
data CannotShow = CannotShow
-- deriving (Show)
```

```haskell

-- will not compile, since CannotShow is not an instance of Show
data CannotDeriveShow = CannotDeriveShow CannotShow
          deriving (Show)

data OK = OK

instance Show OK where
show _ = "OK"

data ThisWorks = ThisWorks OK
  deriving (Show)
```

注意 `CannotShow` 的 `deriving (Show)` 是被注释掉的，因此 `CannotDeriveShow` 才无法派生 `Show` 。

Typeclasses 实战：令 JSON 使用起来更方便

上一章讲述的 `JValue` 处理 JSON 并不容易。例如，下面是一个实际 JSON 数据的删减整理的片段：

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

以及在 Haskell 中的表达：

```haskell
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/")
       ]])
    ]
```

因为 Haskell 并不支持列表中包含不同类型的值，我们不能直接表示一个包含了多个类型的 JSON 对象。我们必须将每个值通过 `JValue` 构造函数来进行包装。这限制了我们的灵活性：如果想要更换数值 `3920` 成为一个字符串 `"3,920"` ，我们必须更换构造函数，即 `JNumber` 变为 `JString` 。

Haskell 的 typeclasses 为此类问题提供了一个诱人的解决方案：

```
type JSONError = String

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

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

现在无需再将类似 `JNumber` 这样的构造函数应用在值上将值包裹，直接应用 `toJValue` 函数在该值上即可。

我们同样提供了一个 `fromJValue` 函数，用于将一个 `JValue` 转换成一个我们所期望的类型。

更多有帮助的错误

让我们构造一下自己的 `Maybe` 与 `Either` ：

```
data Maybe a
  = Nothing
  | Just a
  deriving (Eq, Ord, Read, Show)

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

在 `Bool` 值实例中尝试一下：

```
instance JSON Bool where
  toJValue = JBool
  fromJValue (JBool b) = Right b
  fromJValue _ = Left "not a JSON boolean"
```

通过类型同义词来创建一个实例

```
instance JSON String where
  toJValue = JString
  fromJValue (JString s) = Right s
  fromJValue _ = Left "not a JSON string"
```

活在开放世界

Haskell 的 typeclasses 允许我们在任何合适的时候创建新的 typeclass 实例。

```
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
```

我们还希望转换一个列表成为 JSON，暂时用 `undefined` 作为实例的方法。

```
instance (JSON a) => JSON [a] where
  toJValue = undefined
  fromJValue = undefined
```

对象亦是如此：

```
instance (JSON a) => JSON [(String, a)] where
  toJValue = undefined
  fromJValue = undefined
```

合适重叠实例会导致问题

如果我们将这些定义放入一个原文件中，并加载至 `ghci` ，每个初始化看起来都没问题：

```
ghci> :load BrokenClass
[1 of 2] Compiling SimpleJSON       ( ../ch05/SimpleJSON.hs, interpreted )
[2 of 2] Compiling BrokenClass      ( BrokenClass.hs, interpreted )
Ok, modules loaded: SimpleJSON, BrokenClass.
```

然而当我们尝试使用元组的列表时，错误便发生了。

```
ghci> toJValue [("foo","bar")]

<interactive>:1:0:
    Overlapping instances for JSON [([Char], [Char])]
      arising from a use of `toJValue' at <interactive>:1:0-23
    Matching instances:
      instance (JSON a) => JSON [a]
        -- Defined at BrokenClass.hs:(44,0)-(46,25)
      instance (JSON a) => JSON [(String, a)]
        -- Defined at BrokenClass.hs:(50,0)-(52,25)
    In the expression: toJValue [("foo", "bar")]
    In the definition of `it': it = toJValue [("foo", "bar")]
```

重叠实例的问题是 Haskell 的开放世界假设的结果。下面用一个更简单的例子来说明到底发生了什么：

```
class Borked a where
  bork :: a -> String

instance Borked Int where
  bork = show

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

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

我们有两个 typeclass `Borked` 的 pairs 实例：一对是 `Int`，另一对是其它任意值。

假设我们希望 `bork` 一对 `Int` 值，那么编译器必须选择一个实例来使用。由于这些实例相邻，那么看起来就能够简单的选择更加明确的实例。

然而 GHC 默认是保守的，它会坚持必须只有一个可能的实例。因此最终在使用 `bork` 时会抛出异常。

Tip

什么时候重叠的实例会有影响?

正如我们之前提到的那样，我们可以将一个 typeclass 的实例散落在若干模块中。GHC 并不会抱怨重叠实例的存在。而真正抱怨的时候就是当我们尝试使用这个受影响 typeclass 的方法时，也就是当强制要求选择哪一个实例需要使用的时候。

放宽 typeclasses 的某些限制

通常而言，我们不可以编写一个特定多态类型的 typeclass 实例。例如 `[Char]` 类型就是 `[a]` 指定 `Char` 类型的多态。因此禁止将 `[Char]` 声明为 typeclass 的实例。这个非常的不

方便，因为字符串在真实代码中处处存在。

`TypeSynonymInstances` 语言扩展移除了这个限制，允许我们编写上述的实例。

GHC 还支持另一个有用的语言扩展，`OverlappingInstances` ，专门用于处理重叠实例。当存在若干重叠实例需要选择时，该扩展使编译器选择最明确的那个。

我们通常将该扩展与 `TypeSynoymInstances` 一起使用。例如：

```
import Data.List

class Foo a where
  foo :: a -> String

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

instance {-# OVERLAPS #-} Foo Char where
  foo c = [c]

instance {-# OVERLAPS #-} Foo String where
  foo = id

main :: IO ()
main = do
  putStrLn $ "foo: " ++ foo "SimpleClass"
  putStrLn $ "foo: " ++ foo ["a", "b", "c"]
```

与原文不同之处在于，注解 `{-# LANGUAGE OverlappingInstances #-}` 在 6.8.1 后被弃用，现在则是在 `instance` 后使用 `{-# OVERLAPS #-}` 来表示重叠的实例，详见文档。

如果我们将 `foo` 应用至一个 `String` ，编译器将使用 `String` 指定的实现。尽管存在 `[a]` 与 `Char` 的 `Foo` 实例，但是 `String` 的实例更加的明确，因此 GHC 会选择它。

如何给类型一个新的身份

略。

```
data DataInt = D Int
  deriving (Eq, Ord, Show)

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

略。

总结：三总命名类型的方法：

- `data` 关键字引入了一个真实的新的代数数据类型。

- `type` 关键字给予了已存在的类型一个同义词。我们可以替换的使用类型与其同义词。
- `newtype` 关键字给予已存在类型一个新的身份。原始类型和新的类型是不可替换的。

JSON typeclasses 没有重叠的实例

现在需要帮助编译器区分 JSON 数组的 `[a]`，以及 JSON 对象的 `[(String,[a])]`。它们是造成重叠问题的原因。我们将列表类型包裹起来，这样编译器则不会视其为列表：

```
newtype JAry a = JAry {fromJAry :: [a]}
  deriving (Eq, Ord, Show)
```

当需要将其从模块中导出时，我们需要导出该类型的所有细节。我们模块头看起来像是这样：

```
module JSONClass (JAry (..)) where
```

接下来是另一个包装类型用于隐藏 JSON 对象：

```
newtype JObj a = JObj {fromJObj :: [(String, a)]}
  deriving (Eq, Ord, Show)
```

有了这些类型定义，那么就可以对 `JValue` 类型做点小修改：

```
data JValue
  = JString String
  | JNumber Double
  | JBool Bool
  | JNull
  | JObject (JObj JValue) -- was [(String, JValue)]
  | JArray (JAry JValue) -- was [JValue]
  deriving (Eq, Ord, Show)
```

这个改动并不会影响已经写过的 `JSON` typeclass 实例，不过还是需要为 `JAry` 与 `JObj` 类型编写 `JSON` typeclass 实例。

```
jaryFromJValue :: (JSON a) => JValue -> Either JSONError (JAry a)

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

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

让我们慢慢的看一下将 `JAry a` 转换成 `JValue` 的每个步骤。给定一个列表，我们知道其所有元素都是 `JSON` 实例，将其转换成一个列表的 `JValue` 很简单。

```
listToJValues :: (JSON a) => [a] -> [JValue]
listToJValues = map toJValue
```

有了上述代码后，将其变为一个 `JAry JValue` 就仅仅是应用 `newtype` 类型构造函数即可：

```
jvaluesToJAry :: [JValue] -> JAry JValue
jvaluesToJAry = JAry
```

（记住这并没有性能损耗，仅仅只是告诉编译器隐藏我们在使用一个列表的事实。）将其转换成一个 `JValue` ，我们应用赢一个类型构造函数：

```
jaryOfJValuesToJValue :: JAry JValue -> JValue
jaryOfJValuesToJValue = JArray
```

将这些部分用函数组合的方式集成起来，就获得了一个简洁的一行代码转换成一个 `JValue` ：

```
jaryToJValue = JArray . JAry . map toJValue . fromJAry
```

从 `JValue` 转换至一个 `JAry a` 则需要更多的工作，我们还是将其分解成可复用的部分。基础函数很直接：

```
jaryFromJValue (JArray (JAry a)) = whenRight JAry (mapEithers fromJValue a)
jaryFromJValue _ = Left "not a JSON array"
```

`whenRight` 函数检查它的参数：如果是由 `Right` 构造函数构建的它则调用一个函数，如果是由 `Left` 构造则保留值不变：

```
whenRight :: (b -> c) -> Either a b -> Either a c
whenRight _ (Left err) = Left err
whenRight f (Right a) = Right (f a)
```

更复杂的是 `mapEithers` ，它的行为类似于普通的 `map` 函数，但是如果遇到一个 `Left` 值，它会立刻返回，而不是继续累积一个 `Right` 值的列表。

```
mapEithers :: (a -> Either b c) -> [a] -> Either b [c]
mapEithers f (x : xs) = case mapEithers f xs of
  Left err -> Left err
  Right ys -> case f x of
    Left err -> Left err
    Right y -> Right (y : ys)
mapEithers _ _ = Right []
```

由于隐藏在 `JObj` 类型的列表中的元素有少许结构，因此它与 `JValue` 之间的转换会变得更复杂。幸运的是我们可以复用刚刚定义好的函数：

```
instance (JSON a) => JSON (JObj a) where
  toJValue = JObject . JObj . map (second toJValue) . fromJObj
  fromJValue (JObject (JObj o)) = whenRight JObj (mapEithers unwrap o)
    where
      unwrap (k, v) = whenRight (k,) (fromJValue v)
  fromJValue _ = Left "not a JSON object"
```

7 Input and output

Haskell 中的经典 I/O

略

处理文件以及句柄

Haskell 为 I/O 定义了相当多的基本函数，其中许多函数与其它编程语言中的函数相似。`System.IO` 库中提供了所有的基础 I/O 函数。

使用 `openFile` 会返回一个文件的 `Handle` ，它用于对文件执行指定的操作。Haskell 提供了例如 `hPutStrLn` 这样的函数，其类似于 `putStrLn` ，不同在于接受一个额外的参数 – 一个 `Handle` – 指定哪个文件被操作。当我们结束时，需要用 `hClose` 来结束 `Handle` 。这些函数都定义在 `System.IO` 中，“h”开头的函数对应了几乎所有的非“h”开头的函数；例如 `print` 打印在屏幕上，而 `hPrint` 打印至一个文件。

一个例子：

```haskell
import Data.Char (toUpper)
import System.IO

main :: IO ()
main = do
  inh <- openFile "input.txt" ReadMode
  outh <- openFile "output.txt" WriteMode
  mainloop inh outh
  hClose inh
  hClose outh

  putStrLn "whatever"

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
```

`mainloop` 首先检查输入是否结束（EOF），如果没有则读取一行，将该行转为大写后写入输出文件中，再递归的调用 `mainloop` 。

略

可能的 IOMode 值				
IOMode	**可读?**	**可写?**	**起始位置**	**说明**
ReadMode	Yes	No	文件起始	文件必须存在
WriteMode	No	Yes	文件起始	文件存在时会被清空
ReadWriteMode	Yes	Yes	文件起始	文件不存在时创建；否则现有数据保留
AppendMode	No	Yes	文件尾部	文件不存在时创建；否则现有数据保留

关闭句柄

上述例子中我们已经见到了 `hClose` 用于关闭文件的句柄。在之后的小节中我们将了解缓存 Buffering 这个概念，即 Haskell 在内部为文件维护了缓存。这样做能显著的提升性能，然而这就意味着在对打开的文件调用 `hClose` 之前，数据可能不会被刷新到操作系统中。

另一个原因是打开着的文件消耗系统的资源。如果程序运行的时间很长，开开了很多文件但是并没有关闭它们，那么程序很可能会因为资源枯竭而导致崩溃。

Seek 与 Tell

从一个关联了磁盘文件的 `Handle` 读写时，操作系统会维护一个关于当前位置的内部记录。每当进行下一层读取的时，操作系统返回下一个数据块，其起始点为当前位置，且增加的位置反应了读取了多少数据。

你可以使用 `hTell` 来找到文件中当前的位置。当文件被创建时，它是空的且你的位置将会是 0。在编写 5 个字节后，你的位置则变为了 5，以此类推。`hTell` 接受一个 `Handle` 并返回一个 `IO Integer` 来表示你的位置。

`hTell` 的同伴是 `hSeek` ,它允许你更改文件的位置,其三个参数为: `Handle` , `SeekMode` 以及一个位置。

`SeekMode` 有三个不同类型，用于指定如何解析给定的位置。`AbsoluteSeek` 为文件中的精确位置，等同于 `hTell` ； `RelativeSeek` 意为当前位置开始的多少位置，正值向后，负值向前； `SeekFromEnd` 则是从文件末尾往前多少的位置。

并不是所有的 `Handle` 都是可以 seek 的。一个 `Handle` 通常关联一个文件，但是它还可以关联其它的东西，比如网络连接，磁带机，或者终端。可以使用 `hIsSeekable` 来查看给定的 `Handle` 是否可以 seek。

标准输入，输出以及错误

较早之前我们指出每个非“h”函数通常都会有与其关联的“h”函数用作于任何 `Handle` 上。实际上非“h”函数只不过是“h”函数的一种缩写。

在 `System.IO` 中有三个预定义的 `Handle` ：`stdin` 标准输入，通常是键盘；`stdout` 标准输出，通常是显示器；`stderr` 标准错误，通常也是显示器。

像是 `getLine` 类似的函数可以简单的定义成：

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

删除与文件重命名

`System.Directory` 模块提供了两个比较有用的函数：`removeFile` 接受单个参数，即文件名，然后删除该文件；`renameFile` 接受两个文件名，一个旧名称以及一个新的名称，如果新的文件名是一个不同的路径，那么可以认为这是一个移动。旧的文件名必须在调用 `renameFile` 之前就存在，另外如果新文件已经存在，则重命名后将其删除。

跟很多其它接受一个文件名的函数一样，如果“旧”名称不存在，`renameFile` 则会抛出异常。

`System.Directory` 模块中还有很多其他的函数，像是创建或移除文件夹，在路径中查找文件列表，测试文件是否存在，等等。

临时文件

程序员频繁的需要临时文件。这些文件被用于存储大量等待计算的数据，可被其它程序所用的数据，等等。

通过名为 `openTempFile` 的函数（以及相关联的 `openBinaryTempFile` ）可以帮助我们解决不少问题。

`openTempFile` 接受两个参数：创建文件的路径，以及一个“template”用于命名文件。路径可以是 `.` 来代表当前路径，或者也可以使用 `System.Directory.getTemporaryDirectory` 来找到操作系统所给出的最佳放置临时文件的位置。template 则将一些随机字符添加至文件，用以确保结果是真正唯一的。实际上它保证了可以在一个唯一的文件名上工作。

`openTempFile` 的返回类型是 `IO (FilePath, Handle)` 。元组的第一部分是被创建文件的名称，第二部分则是一个模式为 `ReadWriteMode` 的打开了文件的 `Handle` 。当我们使用完文件，我们希望用 `hClose` 来操作 `Handle` 用于关闭文件，接着调用 `removeFile` 来删除它。接下来我们会看到一个例子用于展示这些函数的用法。

扩展案例：函数式 I/O 以及临时文件

```
import Control.Exception
import System.Directory (getTemporaryDirectory, removeFile)
import System.IO
```

```haskell

main :: IO ()
main = withTempFile "mytemp.txt" myAction

myAction :: FilePath -> Handle -> IO ()
myAction tempname temph = do
  -- Start by displaying a greeting on the terminal
  putStrLn "Welcome to tempfile.hs"
  putStrLn $ "I have a temporary file at " ++ tempname

  -- Let's see what the initial position is
  pos <- hTell temph
  putStrLn $ "My initial position is " ++ show pos

  -- Now, write some data to the temporary file
  let tempdata = show [1 .. 10]
  putStrLn $ "Writing one line containing " ++ show (length tempdata) ++ " bytes: " ++
   tempdata
  hPutStrLn temph tempdata

  -- Get our new position. This doesn't actually modify pos in memory,
  -- but makes the name "pos" correspond to a different value for
  -- the remainder of the "do" block.
  pos <- hTell temph
  putStrLn $ "After writing, my new position is " ++ show pos

  -- Seek to the beginning of the file and display it
  putStrLn "The file content is: "
  hSeek temph AbsoluteSeek 0

  -- hGetContents performs a lazy read of the entire file
  c <- hGetContents temph

  -- Copy the file byte-for-byte to stdout, followed by \n
  putStrLn $ "c: " ++ 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.

```

```
52    After the given function terminates, even if it terminates due to an exception,
53    the Handle is closed and the file is deleted.
54  -}
55  withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
56  withTempFile pattern func = do
57    -- The library ref says that getTemporaryDirectory may raise on exception on
58    -- systems that have no notion of a temporary directory. So, we run
59    -- getTemporaryDirectory under catch. catch takes two functions: one to run,
60    -- and a different one to run if the first raised an exception.
61    -- If getTemporaryDirectory raised an exception, just use "." (the current
62    -- working directory).
63    tempdir <-
64      catch
65        getTemporaryDirectory
66        (\(_ :: IOException) -> return ".") -- explicit annotates exception type
67    (tempfile, temph) <- openTempFile tempdir pattern
68
69    -- Call (func tempfile temph) to perform the action on the temporary file.
70    -- finally takes two actions. The first is the action to run, the second is an action
71    -- to run after the first, regardless of the temporary file is always deleted.
72    -- The return value from finally is the first action's return value.
73    finally
74      (func tempfile temph)
75      ( do
76          hClose temph
77          removeFile tempfile
78      )
```

首先 `withTempFile` 函数证明了 Haskell 在 I/O 时也没有忘记其函数式的天性。该函数接受一个 `String` 与另一个函数。传递给 `withTempFile` 的函数带着临时文件的 `Handle` 被调用，当该函数退出时，临时文件关闭并被删除。

异常处理可以让程序变得更加健壮。通常我们希望在程序结束后删除临时文件，即使程序的执行过程中出现了错误。更多的错误处理会在第十九章讲述。

现在回到程序，`main` 做的事情很简单，调用函数 `withTempFile` 并传入临时文件名以及 `myAction` 函数。

`myAction` 展示了一些信息到终端上，写入了一些数据到文件中，seek 到文件的起始位置，通过 `hGetContents` 进行数据读取，接着按照字节展示文件内容，同时通过 `print c` 进行打印。这等同于 `putStrLn (show c)` 。

现在看看输出：

```
1  % runhaskell tempfiles.hs
2  Welcome to tempfile.hs
3  I have a temporary file at /var/folders/nb/w1q3ztlj139_vz9pqglmbks80000gn/T/mytemp3709-0.txt
4  My initial position is 0
5  Writing one line containing 22 bytes: [1,2,3,4,5,6,7,8,9,10]
6  After writing, my new position is 23
```

```
The file content is:
c: [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"
```

Lazy I/O

Haskell 还提供了另一种方式处理 I/O。由于 Haskell 是 lazy 语言，意味着任何数据只会在其值必须被知道时才会被计算，那么就有了一些新颖的 I/O 处理方法。

hGetContents

一个新颖的 I/O 方式就是 `hGetContents` 函数，其类型 `Handle -> IO String` ，这里的 `String` 代表着由文件的 `Handle` 所返回的所有数据。

在严格求值的语言中，使用这样的函数通常是一个坏主意。假设读一个 500GB 的文件，那么就有可能因为内存不足而导致程序崩溃。在这些语言中，传统的做法就是使用循环来处理文件的所有数据。

但是 `hGetContents` 不一样，它返回的 `String` 是惰性的，即在调用 `hGetContents` 时，并没有读取任何数据。只有在处理列表的元素（字符）时才会从 `Handle` 中读取数据。当 `String` 的元素不再使用时，Haskell 的垃圾收集器则会自动释放内存。

来看一个例子：

```haskell
import Data.Char (toUpper)
import System.IO

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
```

测试：

```
ghci> :load toupper-lazy1.hs
[1 of 1] Compiling Main             ( toupper-lazy1.hs, interpreted )
Ok, modules loaded: Main.
ghci> processData "Hello, there!  How are you?"
```

```
"HELLO, THERE!  HOW ARE YOU?"
ghci> :type processData
processData :: String -> String
ghci> :type processData "Hello!"
processData "Hello!" :: String
```

Warning

如果我们在上述例子中尝试在使用 `inpStr` 的地方（对 `processData` 的调用）之后继续使用 `inpStr` ，那么程序则会失去其内存效率。这是因为编译器将会强制保留 `inpStr` 值在内存中供未来使用。在这里编译器知道 `inpStr` 不会再被使用，那么就在其使用结束后尽快的释放内存。这里需要记住：内存只有在最后一次使用后才会被释放。

这段代码有点繁琐，我们可以让它变得更简洁一些：

```
import Data.Char (toUpper)
import System.IO

main :: IO ()
main = do
  inh <- openFile "input.txt" ReadMode
  outh <- openFile "output.txt" WriteMode
  inpStr <- hGetContents inh
  hPutStr outh (map toUpper inpStr)
  hClose inh
  hClose outh
```

在使用 `hGetContents` 时，我们甚至不需要从输入文件中消费所有的数据。每当 Haskell 系统确定 `hGetContents` 所返回的整个字符可以被垃圾回收时 – 意味着它不会再被使用 – 文件将自动被关闭。同样的原则也适用于从文件中读取的数据。每当给定的数据不再需要时，Haskell 环境就会释放存储该数据的内存。严格来说我们完全都不需要调用 `hClose` 。然而这是一个好的习惯，因为之后对程序的改动会使得 `hClose` 变得很重要。

Warning

在使用 `hGetContents` 时，必须记住即使在之后的程序可能不再显式的引用 `Handle` ，但还是要保持 `Handle` 不被关闭，直到 `hGetContents` 的结果被消费掉了。不这么做的话就可能会导致丢失部分或者全部文件数据。这是因为 Haskell 是惰性的，所以通常可以假设只有在输出了涉及输入的计算结果之后才消费了输入。

readFile 与 writeFile

Haskell 程序员经常使用 `hGetContents` 作为筛选。从一个文件读取数据，处理数据，接着将数据写到别的地方。这非常的通用所以有了一些快捷的方式：`readFile` 以及 `writeFile` 就是这样的函数，以字符串的方式处理文件。它们处理了所有关于打开文件，关闭文件，读取数据，以及写入数据的细节。`readFile` 在内部使用了 `hGetContents` 。

通过 **ghci** 查看这些函数的类型：

```
ghci> :type readFile
readFile :: FilePath -> IO String
ghci> :type writeFile
writeFile :: FilePath -> String -> IO ()
```

使用 `readFile` 与 `writeFile` 改造之前的代码：

```
import Data.Char (toUpper)

main :: IO ()
main = do
  inpStr <- readFile "input.txt"
  writeFile "output.txt" $ map toUpper inpStr
```

关于 Lazy Output

现在我们能理解惰性输入是如何在 Haskell 中工作的。那么惰性输出又是怎么样呢?

我们知道 Haskell 中所有计算都是在其值被需要时才会进行。由于函数例如 `writeFile` 以及 `putStr` 会输出所有传给它们的 `String` ，那么整个 `String` 必须被计算。因此可以保证 `putStr` 的参数将被完整求值。

那么输入惰性的意义呢? 上述例子中，对 `putStr` 或 `writeFile` 的调用是否会强制将整个输入字符串立刻加载到内存中，仅仅只是为了输出?

回答是不。`putStr` （以及其它类似的所有输出函数）在数据可同时，输出数据。它们同样也不需要保留已经输出了的数据，因此只要程序中没有其它东西需要它，内存就可以立刻释放。在某种意义上，可以将 `readFile` 与 `writeFile` 之间的 `String` 视为连接两者的管道。数据进入一端，以某种方式转换后，从另一端流出。

interact

通过 `interact` 函数可以进一步简化我们的程序：

```
import Data.Char (toUpper)

main :: IO ()
main = interact $ map toUpper
```

仅用了一行！测试：

```
$ runghc toupper-lazy4.hs < input.txt > output.txt
```

或者想要打印到屏幕上：

```
$ runghc toupper-lazy4.hs < input.txt
```

如果想要看到 Haskell 的输出确实是在接收到数据块后立刻输出到数据块，可以不加任何参数运行 `runghc toupper-lazy4.hs` ，这样就可以看到在敲击输入回车后，所有字符都会被立刻以大写的形式输出。

我们同样还可以通过 `interact` 编写在输出前添加新行的程序：

```
import Data.Char (toUpper)

main = interact $ map toUpper . (++) "Your data, in uppercase, is:\n\n"
```

由于我们在 `(++)` 后调用的 `map` ，那么新行也会变为大写，修改一下：

```
import Data.Char (toUpper)

main = interact $ (++) "Your data, in uppercase, is:\n\n" . map toUpper
```

这就将新行移到 `map` 外去了。

interact 加上过滤

`interact` 的另一个常用方法就是过滤。比如打印带有字符“a”的每行：

```
main = interact $ unlines . filter (elem 'a') . lines
```

通过 **ghci** 查看上述的三个新函数：

```
ghci> :type lines
lines :: String -> [String]
ghci> :type unlines
unlines :: [String] -> String
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
```

测试：

```
$ runghc filter.hs < input.txt
I like Haskell
Haskell is great
```

IO 单子

Actions

大多数语言不会区分纯函数和非纯函数。Haskell 的函数有其数学意义：它们是存粹的计算，且不会被任何外界的东西改变。另外，计算可以在任何时刻进行。

那么就需要另一些工具来处理 I/O 了。这个工具在 Haskell 中叫做 *actions*。Actions 类似于函数，在定义的时候不会做任何事情，而是在执行任务时才会被唤醒。I/O actions 被定义在 `IO` 单子中。我们会在第十四章中讲解单子。看看一些类型：

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

`putStrLn` 的类型跟其它函数无异。函数接受一个参数并返回一个 `IO ()` 。这个 `IO ()` 就是 action。我们可以存储以及传递 actions 至纯代码中，尽管这并不常用。一个 action 不会做任何事情直到它被唤起。看看例子：

```
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 (fst : remaining) = do
  fst
  runall remaining

main :: IO ()
main = do
  str2action "Start of the program"
  printitall
```

```
30    str2action "Done!"
```

`str2action` 这个函数接受一个参数并返回一个 `IO ()` 。在 `main` 的结尾，可以在另一个 action 中直接使用它，并打印出一行。或者可以存储 – 而不是执行 – 这个 action 到纯代码中。在 `list2actions` 中 – 使用了 `map` 应用在 `str2action` 上并返回一个列表的 actions，正如我们在纯代码中的那样。可以看到通过 `printitall` 的所有内容都是使用纯的工具构建的。

尽管定义了 `printitall` ，它不会被执行直到它的 action 在某处被计算。注意在 `main` 中是如何将 `str2action` 作为一个 I/O action 被执行的，而之前我们在 I/O 单子外使用它，并将结果转为一个列表。

可以这么理解：每个声明，除了 `let` ，在一个 `do` 代码块中必须产生一个将要执行的 I/O action。

对 `printitall` 的调用最终执行所有这些操作。实际上由于 Haskell 是惰性的，所以直到这里才会生成 actions。

测试：

```
% runhaskell actions.hs
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!
```

实际上我们可以将其写成更紧凑的形式。以下是重构：

```
str2message :: String -> String
str2message input = "Data: " ++ input

str2action :: String -> IO ()
str2action = putStrLn . str2message

numbers :: [Int]
numbers = [1 .. 10]

main :: IO ()
main = do
  str2action "Start of the program"
  mapM_ (str2action . show) numbers
  str2action "Done!"
```

注意 `str2action` 中使用了标准函数组合操作符。在 `main` 中，调用了 `mapM_` ，该函数类似于 `map` 接受一个函数与一个列表。而提供给 `mapM_` 的函数则是一个应用至列表中每个元素的 I/O action 。`mapM_` 会抛出函数的结果，不过如果有需要，也可以使用 `mapM` 返回 I/O 结果的列表。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**
> 实际上这些函数并不仅仅作用于 I/O；它们可用于任何 `Monad` 。

为什么有了 `map` 还需要 `mapM` 呢？因为 `map` 是一个返回列表的纯函数，它不能直接执行 actions，而 `mapM` 则是 `IO` 单子中的工具，可用于实际执行 actions。

回到 `main` ，`mapM_` 应用了 `(str2action . show)` 在每个 `numbers` 中的元素上。`show` 将每个数值转为 `String` ，而 `str2action` 转换每个 `String` 为一个 action。`mapM_` 结合这些独立的 actions 成为一个大的 action 并打印出来。

Sequencing

`do` 代码块其实是合并所有 actions 的简便注解。在不使用 `do` 时还有两个操作符可以使用：`>>` 与 `>>=` 。

```
ghci> :type (>>)
(>>) :: (Monad m) => m a -> m b -> m b
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

`>>` 操作符将两个 actions 串联起来：先执行第一个 action，再执行第二个 action。整个计算的结果则是第二个 action 的结果，而第一个 action 的结果则被丢弃。这类似于 `do` 代码块中的一行。可以用 `putStrLn "line 1" >> putStrLn "line 2"` 来测试一下。它将会打印出两行，将第一个 `putStrLn` 的结果丢弃，将第二个的结果作为返回。

`>>=` 操作符运行一个 action，然后将其结果传给一个返回 action 的函数。第二个 action 也将运行，整个表达式的结果还是第二个 action 的结果。例如，可以用 `getLine >>= putStrLn` 测试，它从键盘读取一行然后再将其输出。

现在不用 `do` 代码块来重写一下之前的例子：

```
main :: IO ()
main = do
  putStrLn "Greetings! What is your name?"
```

```
    inpStr <- getLine
    putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

不使用 `do` ：

```
main :: IO ()
main =
  putStrLn "Greetings! What is your name?"
    >> getLine
    >>= (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

在定义 `do` 代码块时，Haskell 编译器内部会执行类似的转换。

Return 的本质

在本章开头我们提到了 `return` 可能并不像是它长得那样。很多语言的关键字 `return` 是立刻终结函数的执行并返回一个值给到调用者。

Haskell 中的 `return` 函数则大相径庭。Haskell 中的 `return` 将数据包装进单子中。在谈及 I/O 时，`return` 用于将纯数据放入 IO 单子中。

为什么要这么做呢？记住，任何结果依赖于 I/O 的东西都必须在 IO 单子内。因此，如果我们正在编写一个执行 I/O 的函数，那么一个纯计算，我们需要使用 `return` 来使这个纯计算成为函数的正确返回值。例如：

```
import Data.Char (toUpper)

isGreen :: IO Bool
isGreen = do
  putStrLn "Is green your favorite color?"
  inpStr <- getLine
  return $ (toUpper . head $ inpStr) == 'Y'
```

我们有一个返回 `Bool` 的纯计算。计算传递给 `return` ，将其放入 `IO` 单子内。由于它是 `do` 代码块中的最后一个值，它变成了 `isGreen` 的返回值，但这并不因为我们使用了 `return` 函数。

下面是同样一个程序，只不过将纯计算剥离成为一个独立的函数。这样可以隔离纯函数，同样使得意图更加的清晰。

```
import Data.Char (toUpper)

isYes :: String -> Bool
isYes inpStr = (toUpper . head $ inpStr) == 'Y'

isGreen :: IO Bool
isGreen = do
  putStrLn "Is green your favorite color?"
  isYes <$> getLine
```

与原文不同之处在于使用了 `<$>` ，即 `fmap` 函数，替换掉了之前的 `inpStr <- getLine` 以及 `return` 。

最后是一个做作的例子用于展示 `return` 并不需要位于 `do` 代码块的结尾。

```
returnTest :: IO ()
returnTest =
    do one <- return 1
       let two = 2
       putStrLn $ show (one + two)
```

Haskell 真的是命令式的吗?

这些 `do` 代码块看起来像是命令式的语言。毕竟大多数情况下我们都是在按顺序运行命令。

但是 Haskell 本质上仍然是一种惰性语言。虽然有时有必要对 I/O 操作进行排序，但这是使用了 Haskell 中的工具来完成的。Haskell 还通过 IO 单子实现了 I/O 与语言其他部分的良好分离。

Lazy I/O 的副作用

本章提到的 `hGetContents` ，其返回的 `String` 可以用在纯代码中。

我们需要更具体的了解副作用是什么。当我们说 Haskell 没有副作用时，这到底是什么意思?

某种层度上，副作用总是可能的。一个没有优化的循环，即使是用纯代码写的，也有可能导致系统的内存耗尽然后导致程序崩溃。或者它可能导致数据被交换到磁盘。

当我们提到没有副作用，意思其实是 Haskell 的纯代码不会运行能触发副作用的命令。纯函数不会修改一个全局变量，I/O 请求，或者运行一个命令关闭系统。

当你有一个 `hGetContents` 而来的字符串被传递至一个纯函数时，函数不会知道 `String` 是从磁盘文件而来的。它只会表现得跟通常那样，但处理该 `String` 可能会导致环境发出 I/O 命令。纯函数不会这么做；它们是纯函数处理的结果，就像将内存交换到磁盘的例子一样。

在某些情况下，你可能需要更多的控制 I/O 发生的确切时间。也许是正在交互式的从用户那里读取数据，或者通过管道从另一个程序读取数据，并需要直接与用户通信。这些情况下，`hGetContents` 可能不合适。

缓存

缓存模式

Haskell 有三种不同的缓存模式：`BufferMode` 类型：`NoBuffering` ，`LineBuffering` 以及 `BlockBuffering` 。

`NoBuffering` 正如其名 – 无缓存。通过例如 `hGetLine` 等函数读取的数据将每次从操作系统重读取一个字符。写入的数据将立刻写入，也经常一次写入一个字符。通常 `NoBuffering` 的性能很差，不适合通用用途。

`LineBuffering` 使输出缓冲区在输出换行符或缓存过大时被写入。在输入时，它通常会尝试读取任何可用的数据块，直到它第一次看到换行符。当从终端读取数据时，每次按下回车键后应该立刻返回数据。这通常是合理的默认值。

`BlockBuffering` 使 Haskell 在可能的情况下以固定大小块来读写数据。在批量处理大量数据时，这是性能最好的方法，即使这些数据是以行记录的。然而它不能用于交互式程序，因为它会阻塞输入，直到读取到完整的块。`BlockBuffering` 接受 `Maybe` 类型的参数：如果 `Nothing` ，它将使用定义好的缓存大小；或者你可以使用 `Just 4096` 之类的设置将缓存设置为 4096 字节。

默认和缓存模式取决于操作系统和 Haskell 的实现。可以通过调用 `hGetBuffering` 向系统询问当前的缓存模式。当前的缓存模式可以通过 `hSetBuffering` 设置，它接受 `Handle` 和 `BufferMode` 。例如 `hSetBuffering stdin (BlockBuffering Nothing)` 。

刷新缓存

对于任何类型的缓存，有时我们会希望强制 Haskell 写出保存在缓存中的任何数据。有些时候这会自动发生：比如说调用 `hClose` 时。而调用 `hFlush` 将强制立刻写入任何挂起的数据。当 `Handle` 是一个网络 socket 且想要立刻传输数据时，或者是想要将磁盘上的数据提供给可能并发读取它的其它程序时，这会很有用。

读取命令行参数

`System.Environment.getArgs` 以 `IO [String]` 返回所有参数。类似 C 里的 `argv` ，始于 `argv[1]` 。程序名（C 中的 `argv[0]` ）则可以通过 `System.Environment.getProgName` 得到。

`System.Console.GetOpt` 模块提供了一些解析命令行选项的工具。如果我们的程序有一些复杂的选项，该模块则很有帮助。

环境变量

读取环境变量可以用 `System.Environment` 的两个函数：`getEnv` 或 `getEnvironment`。前者查找特定变量，变量不存在时抛出异常；后者将所有环境变量以 `[(String, String)]` 输出，可以使用例如 `lookup` 函数来找到所需的变量。

8 Efficient file processing, regular expressions, and file name matching

高效文件处理

下面是一个简单的微基准测试，读取一个全是数字的文本文件，打印它们的总和：

```haskell
main :: IO ()
main = do
  contents <- getContents
  print $ sumFile contents
  where
    sumFile = sum . map read . words
```

尽管 `String` 是用于读写文件的默认类型，它并不高效，因此像这样一个简单的程序性能是很差的。

`String` 代表着 `Char` 值的列表；列表中每个元素的内存分配都是独立的，需要额外的开销。这些因素会影响必须读写文本或二进制数据程序的内存消耗和性能。像这样的简单基准测试，即使是用解释性语言（如 Python）编写的程序，性能也比使用 `String` 的 Haskell 代码高出一个数量级。

`bytestring` 库提供了一个快速并且又低开销的 `String` 类型替代品。通过 `bytestring` 编写的代码通常可以媲美或超过 C 语言的性能和内存消耗，同时还保持了 Haskell 的表现力和简洁性。

该库提供了两个模块。每个定义的函数几乎都是 `String` 对应函数的直接替代品：

- `Data.ByteString` 模块定义了一个严格 *strict* 类型的 `ByteString` 。其以二进制或文本数据的数组形式来表示字符串。
- `Data.ByteString.Lazy` 模块提供了惰性 *lazy* 类型的 `ByteString` 。其以块 *chunks* 数组来表示字符串，最大大小为 64KB。

两个 `ByteString` 类型在特定环境都有更好的表现。对于流式处理一个大数据（几百 MB 至 TB），惰性 `ByteString` 类型表现的最好。它的块大小被调整为适合现代 CPU 的 L1 缓存，且垃圾回收可以快速丢弃不再使用的流数据块。

而严格的 `ByteString` 类型在不太关心内存占用或需要随机访问数据的环境下性能更好。

二进制 I/O 与 qualified imports

现在来开发一个函数来说明 `ByteString` 的一些 API。我们将决定一个文件是否为 ELF 对象：这种格式多用于现代 Unix 类型系统的可执行文件。

查看文件的首四个字节，检查它们是否匹配一个字节列表：

```haskell
import Data.ByteString.Lazy qualified as L

hasElfMagic :: L.ByteString -> Bool
hasElfMagic content = L.take 4 content == elfMagic
  where
    elfMagic = L.pack [0x7f, 0x45, 0x4c, 0x46]
```

`ByteString` 的惰性模块以及严格模块意图解决二进制 I/O。Haskell 用 `Word8` 来展示字节的数据类型；可以通过 `Data.Word` 来引入它。

上述例子的 `L.pack` 函数接受一个 `Word8` 列表，打包它们成为一个惰性的 `ByteString`。（`L.unpack` 函数作用相反。）而 `hasElfMagic` 函数则是比较 `ByteString` 的前四个字符。

以下是将其运用在一个文件上的例子：

```haskell
isElfFile :: FilePath -> IO Bool
isElfFile path = do
  content <- L.readFile path
  return $ hasElfMagic content
```

`L.readFile` 函数是 `readFile` 的惰性 `ByteString` 版本，即按需读取。它效率很高，每次读取最大 64KB。惰性 `ByteString` 很适合我们的任务：因为我们至多读取文件的前四个字节，因此我们可以安全的使用该函数在任何大小的文件上。

Text I/O

`bytestring` 库还提供了两个模块用于 text I/O 功能，`Data.ByteString.Char8` 以及 `Data.ByteString.Lazy.Char8`。它们将单独的字符串元素导出成 `Char` 而不是 `Word8`。

Warning

上述模块中的函数进作用于字节大小的 `Char` 值，也就是仅适用于 ASCII 以及某些欧洲字符集。超过 255 的字符会被截断。

面向字符的 `bytestring` 模块为文本处理提供了一些有用的函数。以下是包含了月度的股票价格的文件：

```
ghci> putStr =<< readFile "prices.csv"
Date,Open,High,Low,Close,Volume,Adj Close
2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80
2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66
2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76
2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41
```

如何找到最高价呢？收盘价在第四列，用都好分隔。下面是得到收盘价的函数：

```haskell
closing = readPrice . (!! 4) . L.split ','
```

该函数是 point-free 风格，需要从右往左阅读。`L.split` 将一个惰性的 `ByteString` 分隔开来，分隔发生在每一次找到匹配值时。`(!!)` 操作符则是获取列表中第 k 个元素。`readPrice` 函数将一个代表分数的字符串转换为数值：

```
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)
```

上面使用了 `L.readInt` 函数，即解析整数，返回整数以及字符串剩余部分。

找到最高收盘价的函数：

```
highestClose = maximum . (Nothing :) . map closing . L.lines

highestCloseFrom path = do
  contents <- L.readFile path
  print $ highestClose contents
```

这里使用了一个小技巧。当我们提供一个空列表给 `maximum` 函数会抛出异常：

```
ghci> maximum [3,6,2,9]
9
ghci> maximum []
*** Exception: Prelude.maximum: empty list
```

测试：

```
ghci> :load HighestClose
[1 of 1] Compiling Main             ( HighestClose.hs, interpreted )
Ok, modules loaded: Main.
ghci> highestCloseFrom "prices.csv"
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Just 2741
```

因为从逻辑中分离了 I/O，测试空数据时不需要创建一个空文件：

```
ghci> highestClose L.empty
Nothing
```

文件名匹配

很多面向系统的编程语言都提供了根据模式来匹配文件的功能。尽管 Haskell 的标准库有不错的系统编程工具，它并没有提供这类的匹配函数。

这类模式通常被称为 `glob 模式` ，通配符模式或者 shell 样式模式。它们有一些简单的规则：

- 将字符串与模式进行匹配，从字符串的开头开始，到末尾结束。
- 大多字面字符都匹配自己。例如，模式中的文本 foo 将匹配输入字符串中的 foo，且仅匹配 foo。
- `*` （星号）字符表示“匹配任何内容”；它将匹配任何文本，包括空字符串。
- `?` （问号）字符匹配任何单个字符。模式 `pic??.jpg` 将匹配 `picaa.jpg` 或 `pic01.jpg` 等名称。
- `[` （开方括号）字符开始一个字符类，以 `]` 结束。它的意思是“匹配这个类中的任何字符”。字符类可以在 `[` 后面加一个 `!` 来表示“匹配不属于这个类的任何字符”。

  作为一种简写，一个字符后面跟着一个 `-` （破折号），后面跟着另一个字符，表示一个范围：”匹配此集合中的任何字符“

虽然 Haskell 的标准库中没有提供 glob 模式的匹配，但是它提供了一个非常好的正则表达式库。

Haskell 中的正则表达式

首先让我们加载 `Text.Regex.Posix` 库

```
ghci> :module +Text.Regex.Posix
```

根据官方 Wiki提到的 `regex-posix` 速度非常的慢，在生产代码中需要换成别的库。

我们这里只需要正则表达式匹配的函数，一个中缀操作符 `(=~)` 即可（从 Perl 中借用）。要克服的第一个障碍是 Haskell 的 regexp 库大量使用了多态。因此 `(=~)` 操作符的类型签名很难理解，因此这里暂时不做解释。

`=~` 对它的两个参数和返回类型使用了 typeclasses。第一个参数（即 `=~` 左侧）是要匹配的文本；第二个参数（右侧）则是要匹配的正则表达式。我们可以传 `String` 或 `ByteString` 作为参数。

结果的多种类型

`=~` 操作符的结果类型是多态的，因此 Haskell 编译器需要知道其类型。在真实代码中，它有可能根据使用的场景被推导出正确的类型。但是 **ghci** 没法这么做，它没有足够的信息来进行推导。

当 **ghci** 不能推导 `target` 类型时，我们需要进行显式指定：

```
ghci> "my left foot" =~ "foo" :: Bool
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
True
ghci> "your right hand" =~ "bar" :: Bool
False
ghci> "your right hand" =~ "(hand|foot)" :: Bool
True
```

正则表达式库的核心是一个名为 `RegexContext` 的 typeclass，用于描述 `target` 类型的行为；基础库为我们定义了很多实例。`Bool` 类型就定义了该 typeclass 的实例，因此我们能得到可用的结果。另一个实例就是 `Int` ：

```
ghci> "a star called henry" =~ "planet" :: Int
0
ghci> "honorificabilitudinitatibus" =~ "[aeiou]" :: Int
13
```

如果寻求的是 `String` 结果，我们将得到第一个匹配的子字符串，或是不匹配时得到一个空字符串。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: String
"ii"
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: String
""
```

另外一个可作为结果的类型是 `[String]` ，其返回所有匹配字符串的列表：

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-50
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression:
          "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [String]
    In the definition of `it':
        it = "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" ::
             [String]
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-54
```

```
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression:
            "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [String]
    In the definition of `it':
        it = "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" ::
             [String]
```

Tip

注意字符串返回

如果期望结果是一个 `String` ，那么就要小心了。因为 `(=~)` 返回一个空字符传来表示”没有匹配“，如果空字符串也可能是 regexp 的有效匹配，那么就会难以识别。在这种情况下就应该是有一个不同的返回类型，例如 `[String]` 。

上述都是”简单的“返回类型，这远没有结束。继续往下推进前，让我们定义一个在之后例子中都会用到的匹配：

```
ghci> let pat = "(foo[a-z]*bar|quux)"
```

我们可以获得关于匹配发生的上下文的大量信息。如果期望是 `(String,String,String)` 元组，则会在第一个匹配之前获得文本，匹配成功的文本，以及紧随其后的文本。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String)
("before ","foodiebar"," after")
```

如果匹配失败，那么整个文本则会在元组中以”之前“元素作为返回，而其它两个元素为空：

```
ghci> "no match here" =~ pat :: (String,String,String)
("no match here","","")
```

请求一个四元素元组则会给我们第四个元素，它是匹配模式中所有组的列表：

```
ghci> "before foodiebar after" =~ pat :: (String,String,String,[String])
("before ","foodiebar"," after",["foodiebar"])
```

我们还可以获取匹配的数值信息。一对 `Int` 给出的第一个匹配的起始偏移量以及长度。如果我们请求这些配对的列表，我们将得到所有匹配的信息。

```
ghci> "before foodiebar after" =~ pat :: (Int,Int)
(7,9)
ghci> "i foobarbar a quux" =~ pat :: [(Int,Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-26
    Possible fix:
```

```
        add an instance declaration for
        (RegexContext Regex [Char] [(Int, Int)])
    In the expression: "i foobarbar a quux" =~ pat :: [(Int, Int)]
    In the definition of `it':
        it = "i foobarbar a quux" =~ pat :: [(Int, Int)]
```

如果请求单个元组，则用值 `-1` 作为元组的第一个元素（匹配偏移量）表示匹配失败；如果请求元组列表，则用空列表表示匹配失败：

```
ghci> "eleemosynary" =~ pat :: (Int,Int)
(-1,0)
ghci> "mondegreen" =~ pat :: [(Int,Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-18
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [(Int, Int)])
    In the expression: "mondegreen" =~ pat :: [(Int, Int)]
    In the definition of `it': it = "mondegreen" =~ pat :: [(Int, Int)]
```

更多的关于正则表达式

混合与匹配字符串类型

之前提到 `=~` 操作符使用 typeclasses 作为其参数类型以及返回类型。可以使用 `String` 或严格 `ByteString` 的值来进行正则表达式计算。

```
ghci> :module +Data.ByteString.Char8
ghci> :type pack "foo"
pack "foo" :: ByteString
```

可以尝试不同组合的 `String` 与 `ByteString`：

```
ghci> pack "foo" =~ "bar" :: Bool
False
ghci> "foo" =~ pack "bar" :: Int
0
ghci> pack "foo" =~ pack "o" :: [(Int, Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex ByteString [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-21
    Possible fix:
      add an instance declaration for
      (RegexContext Regex ByteString [(Int, Int)])
    In the expression: pack "foo" =~ pack "o" :: [(Int, Int)]
```

```
    In the definition of `it':
        it = pack "foo" =~ pack "o" :: [(Int, Int)]
```

不过需要注意的是，如果期望匹配结果中包含字符串值，那么所匹配的文本必须是同一类型的字符串：

```
ghci> pack "good food" =~ ".ood" :: [ByteString]

<interactive>:1:0:
    No instance for (RegexContext Regex ByteString [ByteString])
      arising from a use of `=~' at <interactive>:1:0-25
    Possible fix:
      add an instance declaration for
      (RegexContext Regex ByteString [ByteString])
    In the expression: pack "good food" =~ ".ood" :: [ByteString]
    In the definition of `it':
        it = pack "good food" =~ ".ood" :: [ByteString]
```

上述例子中，使用 `pack` 将 `String` 转为了 `ByteString` 。类型检查器可以接受是因为 `ByteString` 在结果类型中展示出来了。但是如果尝试获取一个 `String` ，则不会成功：

```
ghci> "good food" =~ ".ood" :: [ByteString]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [ByteString])
      arising from a use of `=~' at <interactive>:1:0-20
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [ByteString])
    In the expression: "good food" =~ ".ood" :: [ByteString]
    In the definition of `it':
        it = "good food" =~ ".ood" :: [ByteString]
```

我们可以通过将左侧的字符串类型与结果再次匹配来简单的修复这个问题：

```
ghci> "good food" =~ ".ood" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-20
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression: "good food" =~ ".ood" :: [String]
    In the definition of `it': it = "good food" =~ ".ood" :: [String]
```

其它需要注意的事项

查看 Haskell 库文档时，可以看到几个与 regexp 相关的模块。`Text.Regex.Base` 模块定义了所有其他 regexp 模块所遵循的通用 API。可以同时安装 regexp API 的多个实现。在编写本文时，GHC 与一个实现 `Text.Regex.Posix` 捆绑在一起。顾名思义，这个包提供 POSIX regexp 语义。

Tip

Perl 与 POSIX 正则表达式

如果是从 Perl、Python 或 Java 等语言来学习 Haskell 的，并且在其中一种语言中使用过正则表达式，那么应该意识到由 `Text.Regex.Posix` 模块处理的 POSIX regexp 与 Perl 风格的 `regexp` 在某些重要方面是不同的。以下是一些比较明显的区别。

Perl regexp 引擎在匹配备选项时执行左偏匹配，而 POSIX 引擎选择最贪婪的匹配。这意味着给定一个正则表达式 `(foo|fo*)` 和一个文本字符串 `fooooo`，perl 风格的引擎将给出 `foo` 的匹配（最左边的匹配），而 POSIX 引擎将匹配整个字符串（最贪婪的匹配）。

POSIX regexp 的语法不如 perl 风格的 regexp 统一。它们还缺乏 perl 风格的 regexp 提供的许多功能，例如零宽度断言和对贪婪匹配的控制。

将一个 glob 模式转为一个正则表达式

我们已经见识到了很多种将文本域正则表达式匹配的方法，让我们将注意力转回到 glob 模式。我们希望编写一个接受一个 glob 模式的函数，将其表示形式返回为正则表达式。glob 模式和正则表达式两者都是 text 字符串，因此函数的类型就很明确了：

```
module GlobRegex
  ( globToRegex,
  )
where

import Text.Regex.Posix ((=~))

globToRegex :: String -> String
```

我们生成的正则表达式必须是**固定的**，使得它可以从同到尾的匹配字符串。

```
globToRegex cs = '^' : globToRegex' cs ++ "$"
```

由于 `String` 就是 `[Char]` 的别名，这里 `:` 操作符将一个值（这里是 `^` 字符）放到一个列表的头部，而这个列表则是还未出现的 `globToRegex` 函数所返回的。

Tip

在定义一个值之前使用它

Haskell 使用一个值或者函数时，并不需要它们被声明或者在一个源文件中定义。一个定义出现在第一个被使用的地方之后是很正常的。Haskell 编译器并不关心这一层面上的顺序。这使我们可以灵活的按照逻辑来构建代码，而不是遵循编译器编写者的顺序。

Haskell 模块的编写者通常会使用这个灵活性来让“更加重要”的代码前置于源文件中。这正式我们编写 `globToRegex` 函数以及其帮助函数的方式。

有了正则表达式以后 `globToRegex'` 函数则用作于大部分的翻译工作。以下使用 Haskell 的模式匹配来进行编码：

```
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
```

第一个子句规定了如果达到了 glob 模式的末尾（到那时会看到空字符串），返回 `$` ，即正则表达式中的“匹配到了末尾”。接下来的是的子句则是一系列从 glob 语义切换到正则表达语义的模式。最后一个子句处理其它的每个字符，转义优先。

`escape` 函数确保正则引擎不会解释某些特定字符：

```
escape :: Char -> String
escape c
  | c `elem` regexChars = '\\' : [c]
  | otherwise = [c]
  where
    regexChars = "\\+()^$.{}]|"
```

`charClass` 帮助函数用于检测字符是否正确的终止，不修改输入直到遇到 `]` ：

```
charClass :: String -> String
charClass (']' : cs) = ']' : globToRegex' cs
charClass (c : cs) = c : charClass cs
charClass [] = error "unterminated character class"
```

我们已经定义好了 `globToRegex` 与其帮助函数们，现在加载进 **ghci** 中测试一下：

```
ghci> :set -package regex-posix
package flags have changed, resetting and loading new packages...
ghci> :l GlobRegex.hs
[1 of 1] Compiling GlobRegex        ( GlobRegex.hs, interpreted )
```

```
Ok, one module loaded.
ghci> globToRegex "f??.c"
"^f..\\.c$"
```

看起来像是一个合理的正则，那么用它来匹配字符串呢?

```
ghci> "foo.c" =~ globToRegex "f??.c" :: Bool
True
ghci> "test.c" =~ globToRegex "t[ea]s*" :: Bool
True
ghci> "teste.txt" =~ globToRegex "t[ea]s*" :: Bool
True
```

正常工作！我们还可以为 `fnmatch` 创建一个临时定义：

```
ghci> let fnmatch pat name = name =~ globToRegex pat :: Bool
ghci> :type fnmatch
fnmatch
  :: rgx-bs-0.94.0.2-01560e7d:Text.Regex.Base.RegexLike.RegexLike
       Text.Regex.Posix.Wrap.Regex source1 =>
     String -> source1 -> Bool
ghci> fnmatch "d*" "myname"
False
```

`fnmatch` 并不符合“Haskell 本性”，通常 Haskell 风格的函数都需要描述性的“驼峰”名称，例如“file name matches”为 `fileNameMatches` ，那么我们的库可以这样命名：

```
matchesGlob :: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat
```

一个重要的旁白：编写惰性函数

在一个命令式的语言中，`globToRegex'` 函数通常会表述为一个循环。例如 Python 的标准 fnmatch 模块中包含了一个名为 `translate` 的函数，其功能与我们的 `globToRegex` 函数一致。它的实现就是循环。

如果你有别的函数式编程的经验例如 Scheme 或 ML，那么肯定有“模仿循环的方法就是通过尾递归”这个概念。

再来看一下 `globToRegex'` 函数，并不是一个尾递归函数。可以测试一下最后的子句（其它的子句结构都类似）。

```
globToRegex' (c : cs) = escape c ++ globToRegex' cs
```

它递归的应用了自身，且递归的结果作为一个参数在 `(++)` 函数上。由于递归的应用并非作为函数的最后处理部分，那么 `globToRegex'` 并不是一个尾递归。

那么为什么这个函数并非尾递归呢? 答案就在 Haskell 的非严格计算策略。在提及这个话题之前，让我们快速的了解为什么在传统语言中，我们需要避免这样的尾递归定义。以下是 `(++)` 操作符的一个简单定义。它属于递归，但不是尾递归。

```
(++) :: [a] -> [a] -> [a]

(x:xs) ++ ys = x : (xs ++ ys)
[]     ++ ys = ys
```

在严格执行语言中，如果我们计算 `"foo" ++ "bar"` ，整个列表会被构建出来，接着再返回。非严格计算则会延迟直到真正需要被用到的时候。

如果需要表达式 `"foo" + "bar"` 的一个元素时，函数定义的第一个模式将被匹配，接着返回表达式 `x : (xs ++ ys)` 。因为 `(:)` 构造函数是非严格的，那么 `xs ++ ys` 的计算就能被递延：按需生产更多的元素。当我们生产更多的结果时，我们将不再使用 `x` ，因此垃圾回收器将会进行回收。由于是按需生产元素，且不持有已经完成的部分，编译器就可以以常量的空间来计算我们的代码。

使用我们的模式匹配器

通过一个函数用于匹配 glob 模式是很好的，但是我们希望将其用于实际应用。在 Unix 系的系统中 `glob` 函数返回所有匹配到的文件名称以及路径。让我们在 Haskell 中也构建一个类似的函数：

```
module Glob (namesMatching) where
```

`System.Directory` 模块提供了用于处理路径以及内容的标准函数。

```
import System.Directory (doesDirectoryExist, doesFileExist, getCurrentDirectory,
  getDirectoryContents)
```

而 `System.FilePath` 模块抽象了操作系统路径名称转换的细节， `(</>)` 函数将两个路径连接在一起：

```
ghci> :m +System.FilePath
ghci> "foo" </> "bar"
"foo/bar"
```

`dropTrailingPathSeparator` 函数正如其名，去除末尾的路径分隔：

```
ghci> dropTrailingPathSeparator "foo/"
"foo"
```

`splitFileName` 则分离了路径与文件：

```
ghci> splitFileName "foo/bar/Quux.hs"
("foo/bar/","Quux.hs")
```

`System.Directory` 模块结合 `System.FilePath` 的使用，我们可以编写一个同时作用于 Unix 系与 Windows 的 `namesMatching` 函数：

```
import System.FilePath (dropTrailingPathSeparator, splitFileName, (</>))
```

在这个模块中，我们将模仿一个“for”循环；首次在 Haskell 中尝试错误处理；并使用写好的 `matchesGlob` 函数。

```
import Control.Exception (handle)
import Control.Monad (forM)
import GlobRegex (matchesGlob)
```

由于路径与文件存在于拥有副作用的“真实世界”，我们的 globbing 函数的返回值则必须带上 `IO` 。

如果传入的字符串没有包含模式字符，那么就简单的检查给定的名称是否存在于文件系统重。（注意，我们在这里使用 Haskell 的函数守护语法来编写一个漂亮整洁的定义。使用“if”也是可以的，不过就没有这么美观了）

```
isPattern :: String -> Bool
isPattern = any (`elem` "[*?")

namesMatching pat
  | not (isPattern pat) = do
      exists <- doesNameExist pat
      return ([pat | exists])
```

函数 `doesNameExist` 会在稍后进行定义。

那么如果字符串是一个 glob 模式呢？继续我们的函数定义：

```
| 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)
```

我们使用了 `splitFileName` 将字符串分离成了一对“所有东西除了最终名称”以及“最终名称”。如果首个元素为空，则在当前路径寻找一个模式；否则检查路径名称是否包含模式，没有包含则创建一个包含路径名称的单例列表，包含则列举所有匹配的路径。

Tip

需要注意的事情

`System.FilePath` 模块可以变得有点棘手。上述的例子中 `splitFileName` 函数留下了末尾的斜杠

```
ghci> :module +System.FilePath
ghci> splitFileName "foo/bar"
("foo/","bar")

```

如果我们忘了（或者不了解）要移除斜杠，我们则会无限在 `namesMatching` 中递归，因为下列行为

```
ghci> splitFileName "foo/"
("foo/","")

```

你可以猜想一下之后将会发生什么！

最后，收集了所有匹配的路径，得到了一个列表的列表，再将它们打平成一个单独的列表。

这里陌生的 `forM` 函数其行为类似于一个“for”循环：它映射它的第二个参数（一个 action）在第一个参数（一个列表）上，并返回一个列表的结果。

还剩下一些需要处理的问题。首先是 `doesNameExist` 函数。`System.Directory` 模块并不能检查一个名称是否存在与文件系统中，它强制要求我们决定检查的是文件还是路径。这个 API 很丑陋，因此我们需要将两者合二为一。首先检查是否为文件，因为文件相较于路径更为普遍。

```
doesNameExist :: FilePath -> IO Bool
doesNameExist name = do
  fileExists <- doesFileExist name
  if fileExists
    then return True
    else doesDirectoryExist name
```

还有另外两个函数需要定义，它们的返回都是一个路径中的所有名称。`listMatches` 函数返回的是一个路径中匹配了给定 glob 模式的所有文件。

```
listMatches :: FilePath -> String -> IO [String]
listMatches dirName pat = do
  dirName' <-
    if null dirName
      then getCurrentDirectory
      else return dirName
  handle errorHandler $ do
    names <- getDirectoryContents dirName'
```

```
    let names' =
          if isHidden pat
            then filter isHidden names
            else filter (not . isHidden) names
    return (filter (`matchesGlob` pat) names')
  where
    errorHandler :: SomeException -> IO [String]
    errorHandler _ = return []

isHidden :: [Char] -> Bool
isHidden ('.' : _) = True
isHidden _ = False
```

与原文不同之处在于 `handle errorHandler` 这里需要显式的为 `errorHandler` 标注类型，否则原文的 `handle (const (return []))` 将会在编译时抛出类型不明确的异常，详见StackOverflow。

`listPlain` 函数的返回要么是一个空的列表要么是一个单例列表，这取决于传递的名称是否存在：

```
listPlain :: FilePath -> String -> IO [String]
listPlain dirName baseName = do
  exists <-
    if null baseName
      then doesDirectoryExist dirName
      else doesNameExist (dirName </> baseName)
  return ([baseName | exists])
```

通过 API 设计来处理错误

如果传递给我们的 `globToRegex` 的是一个错误的模式，并不会是一场灾难。这有可能使用户输入了错误的模式，这种情况下我们需要报告有意义的错误信息。

```
type GlobError = String

globToRegex :: String -> Either GlobError String
```

运行我们的代码

`namesMatching` 函数并不能很好地单独工作，我们需要将其与其它函数结合。

首先定义一个 `renameWith` 函数，并非只是简单的为一个文件重命名，而是将一个函数应用至文件名称，用函数返回的值来重命名文件。

```
import System.FilePath (replaceExtensions)
import System.Directory (doesFileExist, renameDirectory, renameFile)
import Glob (namesMatching)
```

```
4
5  renameWith :: (FilePath -> FilePath) -> FilePath -> IO FilePath
6  renameWith f path = do
7    let path' = f path
8    rename path path'
9    return path'
```

这里还是需要一个帮助函数来处理文件与路径：

```
1  rename :: FilePath -> FilePath -> IO ()
2  rename old new = do
3    isFile <- doesFileExist old
4    let f = if isFile then renameFile else renameDirectory
5    f old new
```

`System.FilePath` 模块提供了很多操作文件名的函数，它们可以很好地与 `renameWith` 与 `namesMatching` 函数结合，使得我们可以快速的使用它们来创造复杂行为的函数。例如修改 C++ 源文件的后缀。

```
1  cc2cpp :: IO [FilePath]
2  cc2cpp = mapM (renameWith (flip replaceExtension ".cpp")) =<< namesMatching "*.cc"
```

`flip` 函数接受另一个函数作为参数，交换该函数的入参。`=<<` 函数将其右侧 action 的结果喂给左侧的 action。

9 I/O case study: a library for searching the filesystem

略

简单开始：递归展示一个路径

在设计我们的库之前，让我们先解决几个小问题。我们的第一个问题就是递归的列出一个路径的所有文件与子路径。

```haskell
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)
```

`filter` 表达式确保一个路径列表中不会包含特殊的路径名称，例如 `.` 或者 `..` 。如果没有过滤它们则会有无限的递归。

这里又遇到了上一章的 `forM` ；它等同于 `mapM` 只不过参数调转了：

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
ghci> :t forM
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```

与原文（如下）不同：

```
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]
```

函数体中会检查当前输入是否为一个路径：如果是，则递归调用 `getRecursiveContents` 来列举该路径；否则，它返回一个单例列表，即当前输入的名称。（别忘了 `return` 函数在 Haskell 有独特的意义：它通过单子类型构造函数来包装一个值。）

另一个值得指出的点在于变量 `isDirectory` 的使用。一个命令式的语言例如 Python，通常会表述为 `if os.path.isdir(path)` ，然而 `doesDirectoryExist` 函数是一个 *action*；它的返回类型是 `IO Bool` 而不是 `Bool` 。又因为一个 `if` 表达式需要的是 `Bool` 类型，我们需要使用 `<-` 来获取被 `IO` 所包裹着的 action 的 `Bool` 返回，这样才能在 `if` 中使用解包过后的 `Bool` 。

每个 loop 的便利都会返回一个名称列表，因此这里 `forM` 的结果是 `IO [[FilePath]]` 。我们使用 `concat` 将其打平称为一个列表。

重访匿名与命名函数

在之前的匿名（lambda）函数的章节中，我们列举了一些不要使用匿名函数的原因，然而在这里我们使用了一个匿名函数作为 loop 的本体。这就是 Haskell 中最常见的匿名函数使用方法。

我们已经从 `forM` 与 `mapM` 的类型中得知它们接受函数作为参数。大多数的 loop 本体会是一个在程序中出现一次的代码块。因为我们只会在一处使用这个 loop 本体，那么为什么要给它一个名称呢?

当然了，有时我们需要在若干地方部署同样的代码。这种情况下与其复制黏贴一个匿名函数，不如给已存在的函数一个名称。

为什么同时提供了 `mapM` 与 `forM` ?

同时存在两个除了参数顺序不同功能却一样的函数看起来有点奇怪，实际上它们适用于不同的场景。

考虑上述例子，使用匿名函数作为函数体。如果我们使用 `mapM` 而不是 `forM` ，那么则需要将变量 `properNames` 放置在函数体之后。为了让代码能被正确的解析，我们还需要将整个匿名函数体用圆括号包裹起来，或者是用一个命名函数来代替。

相反，如果 loop 本体就是一个命名函数，且需要遍历的列表是由一个复杂的表达式计算而得的时候，那么就有更好的理由使用 `mapM` 了。

一个简单的查询函数

我们可以是用 `getRecursiveContents` 函数作为一个简单的文件查询的基础:

```
1 import RecursiveContents (getRecursiveContents)
2
3 simpleFind :: (FilePath -> Bool) -> FilePath -> IO [FilePath]
4 simpleFind p path = do
5   names <- getRecursiveContents path
6   return $ filter p names
```

这个函数接受一个谓词，用于筛选 `getRecursiveContents` 所返回的名称。每个传递到谓词的名称都是一个完整的路径，那么我们该如何执行一个类似“找到所有后缀为 `.c` 的文件”这样的常规操作呢?

`System.FilePath` 模块包含了大量的宝贵的函数可以帮助我们操作文件名。上述案例可以用 `takeExtension` 。

```
ghci> :m +System.FilePath
ghci> :t takeExtension
takeExtension :: FilePath -> String
ghci> takeExtension "foo/bar.c"
".c"
ghci> takeExtension "quux"
""
```

这样我们就可以编写一个接受路径并提取后缀再与 `.c` 进行比较的函数了:

```
ghci> :l SimpleFinder
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( SimpleFinder.hs, interpreted )
Ok, two modules loaded.
ghci> :t simpleFind (\p -> takeExtension p == ".c")
simpleFind (\p -> takeExtension p == ".c")
  :: FilePath -> IO [FilePath]
```

虽然 `simpleFind` 可以工作，不过还有一些瑕疵。首先是谓词并没有很强的表达力。它只能查看一个路径名称，而不能知晓输入的是一个文件还是路径。这就意味着尝试使用 `simpleFind` 将列出 `.c` 结尾的目录已经具有相同扩展名的文件。

其次就是 `simpleFind` 没有提供该如何遍历文件系统的控制。为什么这点很重要，可以考虑一下在拥有子版本管理的树形文件系统下，进行源文件的搜索。子版本在每个它管理的路径下都会维护一个私有的 `.svn` 路径；它们每一个包含了若干我们不需要考虑的子文件夹与文件。虽然我们可以简单的过滤掉任何包含 `.svn` 的情况，不过更有效的方式则是在第一时间就避免这些文件夹的遍历。

最后就是 `simpleFind` 是严格执行的，因为它包含了一系列在 `IO` 单子中所执行的 actions。如果我们有上百万的文件需要遍历，我们则需要等待很久并且还会得到一个巨大的包含了百万名称的结果。这对于资源管理与响应速度而言都非常的不好。我们更加倾向于一个惰性流式的结果。

谓词：从贫穷到富裕，同时保持 pure

我们的谓词只能查看文件名称。这就包含了大量有趣的行为：例如当我们想要展示的文件大于给定的大小时。

一个比较简单的做法就是修改 `IO` ：谓词的类型不再是 `FilePath -> Bool` ，而改为 `FilePath -> IO Bool` 。这使得我们可以用任意 I/O 作为谓词函数的一部分。尽管这看起来

很吸引人，但它也有潜在的问题：这样的谓词可能会有任意的副作用，因为返回类型为 I/O a 的函数可以拥有任意的副作用。

让我们利用类型系统来编写更加可以预测，更是 bug 的代码：我们将通过避免“IO”污染来保持谓词的纯粹性。这样可以确保它们不会产生任何讨厌的副作用。同时我们还会喂给它们更多的信息，这样它们就能获得我们想要的表达能力，而不会成为潜在的危险。

Haskell 的 `System.Directory` 模块提供了一组有用的文件元数据。

```
ghci> :m +System.Director
```

- 我们可以使用 `doesFileExist` 与 `doesDirectoryExist` 来判断一个输入是否是文件还是路径。近些年暂时还没有便捷的方法来查看其它的文件类型，例如命名通道，软硬链接等。

```
        ghci> :t doesFileExist
        doesFileExist :: FilePath -> IO Bool
        ghci> doesFileExist "."
        False
        ghci> :t doesDirectoryExist
        doesDirectoryExist :: FilePath -> IO Bool
        ghci> doesDirectoryExist "."
        True

```

- `getPermissions` 函数允许我们得知对于一个文件或者路径的操作是否合法

```
        ghci> :t getPermissions
        getPermissions :: FilePath -> IO Permissions
        ghci> :i Permissions
        type Permissions :: *
        data Permissions
          = directory-1.3.6.2:System.Directory.Internal.Common.Permissions {readable ::
    Bool,
                                                                           writable ::
    Bool,
                                                                           executable
    :: Bool,
                                                                           searchable
    :: Bool}
                -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Eq Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Ord Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Show Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Read Permissions
```

```
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
ghci> getPermissions "."
Permissions {readable = True, writable = True, executable = False, searchable =
True}
ghci> :t searchable
searchable :: Permissions -> Bool
ghci> searchable it

```

- 最后则是 `getModificationTime` 可以告诉我们输入是何时被修改的：

```
ghci> :t getModificationTime
getModificationTime
  :: FilePath
     -> IO time-1.11.1.1:Data.Time.Clock.Internal.UTCTime.UTCTime
ghci> getModificationTime "."
2023-10-14 13:58:10.895659105 UTC

```

那么对于新的谓词而言有多少数据是需要知道的呢？由于可以通过 `Permissions` 来查看一个输入是否为文件还是路径，因此我们不再需要 `doesFileExist` 与 `doesDirectoryExist` 。这样我们有了四个需要查看的数据：

```
import Control.Exception (bracket, handle)
import Control.Monad (filterM)
import Data.Time.Clock (UTCTime)
import RecursiveContents (getRecursiveContents)
import System.Directory (Permissions (..), getModificationTime, getPermissions)
import System.FilePath (takeExtensions)
import System.IO (IOMode (..), hClose, hFileSize, openFile)

type Predicate =
  FilePath ->
  Permissions ->
  Maybe Integer ->
  UTCTime ->
  Bool
```

与原文不同之处在于导入部分，由于 `import System.Time (ClockTime(..))` 已经废弃了，根据 `getModificationTime` 可知需要使用 `import Data.Time.Clock (UTCTime)` 。

我们的 `Predicate` 类型就是一个有着四个参数函数的别名。

注意谓词的返回类型是 `Bool` 而不是 `IO Bool` ：谓词是纯的，并不运行 I/O。有了这个类型，编写一个更具表达性的查询函数就好办了。

```
getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize = undefined

```

```haskell
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
```

我们暂时先跳过 `getFileSize` 函数，稍后会进行讲解。

我们不能使用 `filter` 来调用谓词 `p` ，因为 `p` 的纯性意味着它不能执行收集所需元数据所需的 I/O。

这时就需要一个我们并不熟悉的 `filterM` 函数了，它的行为类似于普通的 `filter` 函数，不过是在 `IO` 单子中对谓词进行计算，即允许谓词执行 I/O。

```haskell
ghci> :m +Control.Monad
ghci> :t filterM
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
```

安全的调整文件大小

虽然 `System.Directory` 没有提供文件大小的查询，不过我们可以用 `System.IO` 来做。它包含了一个名为 `hFileSize` 的函数，可以返回一个文件的字节大小。

```haskell
simpleFileSize :: FilePath -> IO Integer
simpleFileSize path = do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return size
```

虽然这个函数可以工作，但是它还不能为我们所用。在 `betterFind` 中，我们可以无条件的对任何地址输入调用 `getFileSize` ；如果输入的不是一个文件它会返回 `Nothing` ，否则返回 `Just` 。实际上该函数的输入不是一个文件或者不能打开文件（权限不够）时会抛出异常。

以下是一个安全版本：

```haskell
saferFileSize :: FilePath -> IO (Maybe Integer)
saferFileSize path = handle errorHandler $ do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return $ Just size
  where
    errorHandler :: SomeException -> IO (Maybe Integer)
    errorHandler _ = return Nothing
```

这里与原文不同在于匿名函数 `handle (_ -> return Nothing)` 改成了 `errorHandler`，并标注了类型 `errorHandler :: SomeException -> IO (Maybe Integer)` 。（与第八章的处理方式一致）

函数体内部几乎是一致的，除了 `handle` 子句。

这里的异常处理是只要有异常发生就返回 `Nothing` ，其余的改动就是返回值被包裹了 `Just` 。

`saferFileSize` 函数现在拥有了正确的类型签名，也不会再抛出任何异常了，不过它仍然不完整。这里只处理了 `openFile` 的异常，但是 `hFileSize` 仍然会抛出异常，比如说命名管道。这样的异常会被 `handle` 捕获，但是我们的 `hClose` 永远不会被执行。

Haskell 的实现会在文件句柄不再使用时自动关闭，但是这要在垃圾回收时才会发生，而接下来垃圾回收的时间却不能保证。

文件句柄是稀有资源，这份稀有性是由操作系统决定的，比如 Linux 系统中一个进程默认同时打开文件的最大数量是 1024 个。

这不难想象一个场景，在调用了使用了 `saferFileSize` 的 `betterFind` 函数因为穷尽了最大文件开启数，而垃圾回收还没开始时，程序崩溃了。

这是一种特别危险的 bug：它会在若干条件组合在一起时变得难以追踪。在 `betterFind` 访问了足够多数量的非文件时句柄没有被关闭而达到了进程最大文件开启数的上限，然后在垃圾回收还未发生之前又打开了另外的文件。

更糟糕的是，任何后续错误都是由程序中无法访问数据而引起，并且还没有垃圾回收。出现这样的 bug 依赖于程序结构，文件系统内容，以及当前程序的运行时还未触发垃圾回收。

这样的问题在开发的过程中很容易被忽视，然后后续发生时会让诊断变得异常困难。

获取-使用-释放

我们需要 `openFile` 成功时 `hClose` 能总是被调用。`Control.Exception` 模块为此提供了 `bracket` 函数。

```
ghci> :m +Control.Exception
ghci> :type bracket
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

`bracket` 函数接受三个 actions 参数。第一个 action 用于获取一个资源，第二个则是释放资源，第三个则是介于前面两者之间，让我们称其为“使用”action。如果“获取”action 成功了，“释放”action *总是*会被调用。这确保了资源总是能被释放。“使用”和“释放”actions 分别传给了“获取”action。

当“使用”action 在执行时发生了异常，`bracket` 会调用“释放”action 然后再重新抛出异常。如果“使用”action 成功了，`bracket` 会调用“释放”action 然后再返回”使用“action 所返回的值。

我们现在可以编写一个完全安全的函数了：它不会抛出异常；也不会产生句柄垃圾而导致程序崩溃。

```
getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize path = handle errorHandler $
  bracket (openFile path ReadMode) hClose $ \h -> do
    size <- hFileSize h
    return (Just size)
  where
    errorHandler :: SomeException -> IO (Maybe Integer)
    errorHandler _ = return Nothing
```

让我们看一下这里的 `bracket` ，首先是打开文件，返回打开文件的句柄，接着是关闭句柄，最后是对句柄调用 `hFileSize` 并将结果包装进 `Just` 。

我们需要同时使用 `bracket` 与 `handle` 才能达成目的。前者能保证垃圾文件句柄不会堆积，后者确保异常被处理。

用于谓词的领域特定语言

让我们尝试着写一个谓词，我们的谓词将会检查一个 C++ 源文件是否超过了 128KB。

```
myTest path _ (Just size) _ = takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

这看起来并不怎么样。谓词有四个参数，总是忽略其中两个，定义另外两个。我们当然可以做的更好，让我们编写一些让谓词变得更简洁的代码。

有时这类库被认为是*内嵌领域特定语言*：我们使用编程语言的工具（因此是*内嵌的*）来编写代码用以优雅的解决特类问题（因此是*领域特定的*）。

第一步首先是编写一个返回某个参数的函数。下面是从传入至 `Predicate` 的参数中提取 path：

```
pathP path _ _ _ = path
```

如果不提供一个类型签名，Haskell 则会为该函数推导出一个非常通用的类型。这在未来可能会带来错误而难以解析，因此我们给 `pathP` 一个类型。

```
type InfoP a =
  FilePath -> -- path to directory entry
  Permissions -> -- permissions
  Maybe Integer -> -- file size (Nothing if not file)
  UTCTime -> -- last modified
  a

pathP :: InfoP FilePath
pathP path _ _ _ = path
```

我们创建了一个类型同义词使得可以将其用于其它类似结构函数。该类型同义词接受一个类型参数，这样我们就可以指定不同的结果类型。

```
sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing _ = -1
```

实际上，本章较早之前所定义的 `Predicate` 类型与 `InfoP Bool` 是一致的。（因此我们可以去除 `Predicate` 类型。）

那么 `pathP` 与 `sizeP` 怎么用呢？通过一点点粘合，可以将它们放入谓词中使用（`P` 后缀即意味着“predicate”）。

```
equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
-- equalP f k = \w x y z -> f w x y z == k
equalP f k w x y z = f w x y z == k
```

`equalP` 的类型签名值得注意。它接受一个 `InfoP a` ，同时兼容 `pathP` 与 `sizeP` 。它接受一个 `a` ，返回一个 `InfoP Bool` ，即 `Predicate` 的同义词。换言之，`equalP` 构建一个谓语。

`equalP` 函数通过返回一个匿名函数来工作。该匿名函数接受一个谓词作为参数，将其传递给 `f` ，接着将其返回与 `k` 作比较。

由于 Haskell 的所有函数都是柯里化的，这么编写 `equalP` 实际上并不必要。我们可以省略匿名函数接着依赖柯里化来完成剩下的部分，现在让我们来编写一个相同效果的函数。

```
equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k
```

在继续探索之前，让我们将我们的模块加载至 **ghci**。

```
ghci> :l BetterPredicate.hs
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( BetterPredicate.hs, interpreted )
Ok, two modules loaded.
```

让我们看看由这些函数所构建的谓词是否生效。

```
ghci> :t betterFind (sizeP `equalP` 1024)
betterFind (sizeP `equalP` 1024) :: FilePath -> IO [FilePath]
```

注意我们并没有真正的调用 `betterFind` ，不过却能确认表达式的类型。我们现在拥有了一个更具表现力的方式来展示所有相同大小的文件。

通过 lifting 避免重复

除开 `equalP` ，我们还可以编写其它的二元函数。我们倾向于不为每个定义都编写一个完整的定义，因为这看起来冗余而没有必要。

为此让我们使用 Haskell 的抽象能力。我们将 `equalP` 的定义而不是直接调用 `(==)` ，将其作为参数传递给希望调用的二元函数中。

```
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 (<)
```

这里接受如 `(>)` 这样的函数，转换它为另一个作用于不同上下文的函数，如 `greaterP` ，这意味着将其*提升 lifting* 至该上下文。这解释了函数名中带有 `lift` 。Lifting 让我们复用代码并且减少重复的模式。在本书的后续章节中，我们将大量的使用它。

我们在提升一个函数时，通常会分别将其原始版本以及新版本称为 *unlifted* 与 *lifted*。

通过这种方式，我们的 `q` （需要提升的函数）作为 `liftP` 的第一个参数是经过深思熟虑的。这使得可以编写如 `greaterP` 以及 `lesserP` 这样简洁的定义。相较于其它的语言，偏应用使得寻找“最佳”的参数顺序成为了 Haskell 中 API 设计中更为重要的环节。在没有偏应用的语言中，参数的顺序在于使用习惯与方便性。而在 Haskell 中将参数设计在错误的位置，则会丢失偏应用所带来的简洁性。

我们可以通过组合子来复现一下这样的简洁性。例如，`forM` 在 2007 年之前并没有被加入至 `Control.Monad` 模块中，更早之前人们则会使用 `flip mapM` 来替代。

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
ghci> :t forM
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
ghci> :t flip mapM
flip mapM
  :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```

与原文不同之处在于上述函数变得更加泛化了，从之前的只对列表生效变为了对所有满足 `Traversable` typeclass 的类型生效。

将谓词们粘合在一起

如果我们希望组合谓词们，那么就可以遵循之前所做的那样：

```
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z
```

现在我们了解了 lifting，那么减少之前必须为布尔值操作符所编写的代码量就变得很自然了。

```
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

andP2 = liftP (&&)
orP2 = liftP (||)
```

注意 `liftP2` 跟之前的 `liftP` 很像。实际上前者更通用，因为我们可以根据 `liftP2` 来编写 `liftP` ：

```
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
```

Tip

组合子 Combinators

Haskell 中，我们将接受函数作为参数并返回新函数的函数成为组合子。

现在我们有了些帮助函数，可以回到较早之前定义的 `myTest` 函数了。

```
myTest :: (Ord a, Num a) => FilePath -> p1 -> Maybe a -> p2 -> Bool
myTest path _ (Just size) _ = takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

那么该函数使用新的组合子会长什么样呢?

```
liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP` (sizeP `greaterP` 131072)
```

我们添加了一个最终的组合子，`liftPath` ，因为操作文件名是一个常用的操作。

定义与使用新的操作符

我们还可以通过定义新的中缀操作符来让我们的领域特定语言更进一步。

```
-- explicit annotation
(==?) :: InfoP String -> String -> InfoP Bool
(==?) = equalP

(&&?) :: InfoP Bool -> InfoP Bool -> InfoP Bool
(&&?) = andP

-- explicit annotation
(>?) :: InfoP Integer -> Integer -> InfoP Bool
(>?) = greaterP

myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)
```

注意与原文不同之处在于 `(==?)` 以及 `(>?)` 加上了显式类型签名，因为在没有 `myTest3` 时，编译器无法正确的进行类型推导而导致编译错误。

我们选择如 `(==?)` 这样的名称作为提升函数，这样可以在视觉上与原始函数对应。

上面定义中的圆括号是必要的，因为我们没有告诉 Haskell 新操作符的优先级或结合性。语言规定没有固定声明的操作符应被视为 `infixl 9` ，即在最高优先级从左到右求值。

我们可以通过为新操作符编写固定声明，第一步是找出未提升的操作符使得可以模拟它们。

```
ghci> :i ==
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  ...
        -- Defined in ‘GHC.Classes’
infix 4 ==
ghci> :i &&
(&&) :: Bool -> Bool -> Bool    -- Defined in ‘GHC.Classes’
infixr 3 &&
ghci> :i >
type Ord :: * -> Constraint
class Eq a => Ord a where
  ...
  (>) :: a -> a -> Bool
  ...
        -- Defined in ‘GHC.Classes’
infix 4 >
```

有了它们以后，我们可以编写一个不带圆括号的表达式，解析后与 `myTest3` 一致：

```
infix 4 ==?
infixr 3 &&?
infix 4 >?

myTest4 = liftPath takeExtension ==? ".cpp" &&? sizeP >? 131072
```

控制遍历

在遍历文件系统时，我们希望得到更多在方向上的控制。一个简单的方法便是允许传递给定路径的子路径列表至函数，并返回另一个列表。该列表允许元素被移除，或者可以以其他方式排序，或是两者皆可。最简单的控制函数是 `id` ，返回的值与输入值保持一致。

为了多样性，我们将改变一些展示的方式。相较于使用一个解释性函数类型 `InfoP a` ，我们将使用一个普通的代数数据类型来展示同样的信息。

```
import Data.Time (UTCTime)
import System.Directory (Permissions)

data Info = Info
```

```
    { infoPath :: FilePath,
      infoPerms :: Maybe Permissions,
      infoSize :: Maybe Integer,
      infoModTime :: Maybe UTCTime
    }
    deriving (Eq, Ord, Show)

getInfo :: FilePath -> IO info
```

我们使用 record 语义来“自由的”访问函数，例如 `infoPath` 。`traverse` 函数的类型很简单，正如之前推测的那样。为了获取文件或者路径的 `Info` ，我们可以调用 `getInfo` action。

```
traverse' :: ([Info] -> [Info]) -> FilePath -> IO [Info]
```

与原文不同之处在于 `traverse` 已经在 `Prelude` 中定义了，因此需要另一个名称。`traverse` 的定义如下：

```
traverse' :: ([Info] -> [Info]) -> FilePath -> IO [Info]
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
```

首先是变量 `contents` 的赋值，从右至左来阅读代码。我们已经知道了 `names` 是一个路径列表，需要确保当前路径被前置放置在列表中的每个元素中，同时还要包含当前路径其本身。接着使用 `mapM` 应用 `getInfo` 函数在上述的返回值（路径列表）上。

再次从右往左阅读代码，我们可以看到该行的最后一个元素是一个匿名函数的定义。给定该匿名函数一个 `Info` 值，该函数要么递归的访问一个路径（有额外的检查可以确保我们不会再次访问 `path` ），要么返回一个单利列表（用于匹配 `traverse'` 的返回类型）。

用户提供的遍历控制函数 `order` 处理 `contents` 的结果为一个 `Info` 值列表，然后再被 `forM` 将上述匿名函数应用至这个列表的每个元素上。

最后是 `liftM` 函数，接受一个普通函数 `concat` ，提升该函数至 `IO` 单子。换言之，`liftM` 将 `forM` 的结果（类型为 `IO [[Info]]` ）从 `IO` 单子中提取出来，将 `concat` 应用在提取的结果上（返回值类型为 `[Info]` ，即我们所需要的），然后再将返回结果放入 `IO` 单子中。

最后，不要忘记定义我们的 `getInfo` 函数：

```
getInfo :: FilePath -> IO Info
getInfo path = do
  perms <- maybeIO (getPermissions path)
  size <- maybeIO (withFile path ReadMode hFileSize)
  modified <- maybeIO (getModificationTime path)
  return (Info path perms size modified)

maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\(SomeException _) -> return Nothing) $ Just `liftM` act
```

`size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)` 可以被 `withFile` 简化成 `size <- maybeIO (withFile path ReadMode hFileSize)` ，其次就是 `maybeIO` 的匿名函数的入参显式声明类型 `(\(SomeException _) -> return Nothing)` 。

这里唯一需要注意的是一个有用的组合子 `maybeIO` ，其将一个可能会抛出异常的 `IO` action，以 `Maybe` 包装结果。

密集，可读性与学习过程

Haskell 中像 `traverse'` 这样密集的代码是不常见的。代码的表现力是很重要的，它需要相关的少量代码可以被流畅的阅读以及编写。

对比一下下面相同作用的密集代码，这更像是一个经验较少的 Haskell 程序员所写的：

```
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 = maybe False searchable (infoPerms info)
  recurse info = do
    if isDirectory info && infoPath info /= path
      then traverseVerbose order (infoPath info)
      else return [info]
```

与原文不同之处在于 `isDirectory` 的 `case ... of` 被 `maybe` 替换了。

编写可维护的 Haskell 代码的关键是在密集与可读性中寻找平衡。

另一个角度看待遍历

尽管 `traverse'` 函数相较于 `betterFind` 函数，给与了我们更多的控制，但它还有一个重要的不足：我们可以在路径上进行递归，却不能过滤其它的名称，直到我们构建了整个列表的

名称树。如果遍历的路径包含了 100,000 个文件，而我们只关心其中三个，我们将分配 100,000 个元素的列表，然后才有机会修剪改列表保留三个所需的元素。

一种解决方案就是提供一个筛选函数作为 `traverse'` 的参数，这样可以将其应用在我们生成的名称列表上。这使得分配的列表只包含所需的元素。

然而这种方案同样有一个弱点：假设我们知道只需三个元素，而这三个元素在 100,000 个元素的前排，这种情况下我们无需再访问剩余的 99,997 个元素。这绝不是一个人为的例子：例如，邮箱的文件夹里的邮件，一个路径代表了邮箱所包含的千分之十的文件是非常常见的。

我们可以从另一个角度来解决上述两种方案的弱点：如果我们将文件系统的遍历视为一个路径层次的折叠 *fold* 呢?

我们熟知的 fold 有 `foldr` 以及 `foldl'` ，非常简洁的阐述了遍历一个列表并累积一个结果。延展折叠路径列表的概念并不难，不过我们所希望的是添加一个控制到折叠上。我们将这个控制表述为一个代数数据类型。

```
import ControlledVisit (Info)

data Iterate seed
  = Done {unwrap :: seed}
  | Skip {unwrap :: seed}
  | Continue {unwrap :: seed}
  deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed
```

`Iterator` 类型给与我们所想要折叠的函数了一个便捷的别名。它接受一个 `seed` 以及一个用于表示路径的 `Info` 值，返回一个新的 `seed` 以及一个折叠函数的指示，该指示代表着 `Iterate` 类型的构造函数。

- 如果指示为 `Done` ，那么遍历应该立即停止。由 `Done` 所包装的值应该作为结果被返回。
- 如果指示为 `Skip` ，同时当前 `Info` 代表着一个路径，遍历将不会在该路径上递归。
- 其余情况下的遍历为 `Continue` ，使用包装的值作为 fold 函数的下一个调用。

我们的折叠逻辑上是一个左折叠，因为折叠第一个输入，同时每一步的 seed 是上一步所返回的结果。

```
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)
```

上述代码中有几个有趣的点。首先作用域的使用避免了额外的参数传递。顶层的 `foldTree` 函数仅是 `fold` 的一个包装，从 `fold` 的构造函数中提取最终结果。

由于 `fold` 是一个本地函数，我们无需传递 `foldTree` 的 `iter` 变量给它；它本身就可以访问外层的作用域。同样的，`walk` 也可以看到外层作用域的 `path` 。

另一个值得关注的点就是 `walk` 是一个尾递归循环，而非早前定义的函数中被 `forM` 所调用的匿名函数。这使得可以在需要的时候停止循环，也就是当迭代器返回 `Done` 时退出。

尽管 `fold` 调用 `walk` ，`walk` 调用 `fold` 递归的遍历子路径。每个函数返回一个被包装在 `Iterate` 的 seed：当 `fold` 被 `walk` 调用再返回，`walk` 检测其结果并决定是否继续还是退出（`Done` ）。这种方式下，调用者提供的迭代器返回 `Done` 时会立刻终止两个函数之间的递归调用。

那么一个迭代器在实际使用上是什么样的呢？下面是一个复杂的例子，用于查看至多三个 bitmap 图像，且不会递归到 Subversion 元数据目录中。

```
atMostThreePictures :: Iterator [FilePath]
atMostThreePictures paths info
  | length paths == 3 =
      Done paths
  | isDirectory info && takeFileName path == ".svn" =
      Skip paths
  | otherwise =
      Continue paths
  where
    extension = map toLower $ takeExtension path
    path = infoPath info
```

调用 `foldTree atMostThreePictures []` ，返回值类型为 `IO [FilePath]` 。

当然了，迭代器也不需要很复杂。下面是一个统计路径数的例子：

```
countDirectories count info =
Continue (if isDirectory info then count + 1 else count)
```

这两门传递给 `foldTree` 的初始 seed 需要是数字零。

有用的编码指导

略

10 Code case study: parsing a binary data format

本章我们将要探讨一个通用任务：解析一个二进制文件。使用该任务有两个目的，第一是借助解析来探讨程序的规划，重构，以及“模板化代码移除”。我们将展示如何简化重复的代码，并为我们在第 14 章 Monads 中做准备。

我们将使用的文件格式源自 netpbm 套件，这是一个古老而可敬的用于处理位图图像的程序和文件格式集合。这些文件格式具有广泛的时候以及容易解析的双重有点。更重要的是为了方便起见，netpbm 文件没有被压缩。

灰度文件

netpbm 的灰度文件格式为 PGM (“portable grey map”)。它有两种格式构；“plain”（或“P2”）格式是由 ASCII 编码的，而更常用的“raw”（“P5”）则是二进制格式。

两种格式的文件都有 header，即一个用于描述格式的“魔力”字符串。对于一个 plain 文件而言，字符串为 `P2` ；raw 文件则是 `P5` 。紧随字符串后的则是一个空格然后再是三个数字：图片的宽度，长度，以及最大灰度。这些数字皆为 ASCII 小数，由空格分隔。

接下来的就是图片数据。raw 文件中是二进制的字符串，plain 文件中则是由单空格字符分隔的 ASCII 小数构成。

raw 文件可以包含一系列的图片，一张接着一张，每张由 header 开头；plain 文件仅包含一张图片。

解析一个原始 PGM 文件

我们的第一个解析函数仅需考虑 raw PGM 文件，同时该解析函数是一个纯函数。该函数并不对数据获取负责，仅仅用作于解析。这在 Haskell 中是一个常规操作。通过分离数据读取，我们获得了更加灵活的操作空间。

我们将使用 `ByteString` 类型来存储 greymap 数据。由于 PGM 文件的头部是 ASCII 文本，而本体是二进制的，我们需要同时引用文本以及二进制的 `ByteString` 模块。

```
import qualified Data.ByteString.Lazy as L
import qualified Data.ByteString.Lazy.Char8 as L8
import Data.Char (isSpace)
```

我们将使用一个直接的数据类型来表示 PGM 图片。

```
data Greymap = Greymap
{ greyWidth :: Int,
  greyHeight :: Int,
  greyMax :: Int,
  greyData :: L.ByteString
}
```

```
  deriving (Eq)
```

通常而言，Haskell 的 `Show` 实例应该生成一个字符串作为展示，同时我们可以通过 `read` 将其从字符串中转回。然而对于 bitmap 图像文件而言，这会潜在的生产出巨大的文本字符串，例如对一个图片执行 `show` 。因此我们不会让编译器自动派生一个 `Show` 实例：我们编写自己的实现，并将其简化。

```
instance Show Greymap where
  show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m
```

因为我们的 `Show` 实例是特意避开了打印 bitmap 数据，因此我们无法编写一个 `Read` 实例，毕竟我们不能通过 `show` 的结果来重新构建一个合法的 `Greymap` 。

接下来是我们解析函数的类型：

```
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
```

该函数接受一个 `ByteString` ，如果解析成功则返回一个单独解析好的 `Greymap` ，以及剩下还需要解析的字符串。

我们的解析函数需要需要花费一点时间。首先是确保输入的文件是 raw PGM；接着解析文件头剩下的数值；然后再是解析 bitmap 数据。下面是一个浅显的表达方式：

```
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader = undefined

getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat = undefined

getBytes :: Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes = undefined

-- parse function
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5 s =
  case matchHeader (L8.pack "PS") 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)
```

这非常字面化的代码将所有的解析放在了一个长的阶梯型的 `case` 表达式上。每个函数在消费完其所需的字符串后，都会返回剩下的 `ByteString` 。接着解构每个结果，如果解析失败则返回 `Nothing` 。下面是解析过程中所用到的函数：

```
matchHeader :: 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

getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
  Nothing -> Nothing
  Just (num, rest)
    | num <= 0 -> Nothing
    | otherwise -> Just (fromIntegral num, rest)

getBytes :: 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
```

移除样板代码

虽然 `parseP5` 函数能工作，其形态并不令人满意。我们不停地重复 `Maybe` 值的构建与解构，且仅在匹配 `Just` 时继续。所有的这些相似的 `case` 表达式就像是“模板代码”那样。

我们可以看到有两个模式。首先是很多应用的函数都有相同的类型，接受 `ByteString` 作为其最后一个参数，并返回 `Maybe` 。其次 `parseP5` 函数中的每一步“阶梯”都会解构一个 `Maybe` 值，要么失败或是传递未解包的结果给一个函数。

我们可以轻易地用一个函数捕获第二个模式。

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v >>? f = f v
```

`>>?` 函数的作用非常简单：它接受一个值作为其右参，以及一个函数作为其左参。如果值非 `Nothing` ，那么便将函数应用在 `Just` 值上。我们定义了作为一个操作符的函数，以此可

以链接所有函数在一起。最后我们并未提供给 `(>>?)` 一个优先级声明，那么默认为 `infixl 9`（左结合，最强的操作符优先级）。换言之，`a >>? b >>? c` 将会从左至右进行计算，类似于 `(a >>? b) >>? c` 。

有了这个链接函数，我们可以再尝试一下编写我们的解析函数。

```
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链条的思考。每个 `(>>?)` 的左侧总是一个 `Maybe` 值；右侧则是一个返回 `Maybe` 值的函数。那么表达式的两侧类型都是 `Maybe` ，即满足下一个 `(>>?)` 表达式。

另一个增加可读性的工作就是添加了 `skipSpace` 函数。

隐式状态

我们的代码仍然在显式的传递一对一对的值，第一个元素作为解析过程中的结果，第二个元素则是当前剩余的 `ByteString` 。当我们想要扩展代码时，例如追踪消费了多少字节数量以便报告解析异常位置的时候，我们需要修改八个不同的地方，仅仅是为了传递一个三元组。

即使在少量代码的情况下，这样的需求都会使得代码难以修改。这个问题存在于使用模式匹配从每个元组中提取值时：我们默认了在代码中直接使用元组。

让我们讲解一下新代码为何是不灵活的。首先修改解析器使用的状态类型。

```
data ParseState = ParseState
{ string :: L.ByteString,
  offset :: Int64
}
deriving (Show)
```

在我们新的代数数据类型中，我们同时拥有了追踪剩余字符串以及当前偏移量的能力。最重要的变化还是使用了 record 语义：现在可以避免对状态的模式匹配了，而是使用访问函数 `string` 以及 `offset` 。

我们给与需要解析的状态一个名称。当我们为某物命名时，它可以变得更有意义。例如，我们可以将解析视为一种函数：其消费一个解析状态，并同时生产一个新的解析状态，以及一些额外的信息。我们可以直接将其视为一个 Haskell 类型。

```
simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

为了更好的帮助用户，我们可以在解析失败时报告异常信息。这仅仅需要我们解析器一点小小的改动。

```
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

之前显式的使用状态元组时，要扩展解析器的时候我们很快就发现问题了。为了避免重复，我们将使用 `newtype` 声明来隐去解析类型的细节。

```
newtype Parse a = Parse
{ runParse :: ParseState -> Either String (a, ParseState)
}
```

记住 `newtype` 定义就是一个编译时的包装函数，因此它没有任何的运行时负荷。当我们使用该函数时，我们将应用 `runParser` 访问函数。

如果我们不从模块中导出 `Parse` 值，我们可以确保他人不会意外的创建一个解析器，也避免了通过模式匹配来查看内部实现。

唯一性解析器

让我们尝试定义一个简单的解析器，*identity* 解析器。它的功能便是将任何传递进其的参数转换为解析器的结果。这种情况下，它更像是 `id` 函数。

```
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

该函数不会改变解析状态，并使用其参数作为解析的结果。我们将函数体包装成 `Parse` 类型来满足类型检查器。那么我们该如何使用该包装后的函数来进行解析呢?

首先我们通过 `runParse` 函数去除 `Parse` 的包装，以此获取其中的函数。接着构造一个 `ParseState` ，并以此运行我们的解析函数。最后则是将解析的结果从最终的 `ParseState` 中分离。

```
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` 解析器与 `parse` 函数都不会测试状态，我们甚至不需要创建一个字符串输入来进行测试。

```
ghci> :l Parse.hs
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
ghci> :t parse (identity 1) undefined
parse (identity 1) undefined :: Num a => Either String a
ghci> parse (identity 1) undefined
Right 1
ghci> parse (identity "foo") undefined
Right "foo"
```

解析器甚至不会检查其不感兴趣的输入，之后我们将见识到其有用之处。

Record 语义，更新，以及模式匹配

Record 语义不仅仅只有访问函数有用：我们用它来拷贝以及修改现有值的一部分。使用中，概念如下：

```
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset = initState {offset = newOffset}
```

这会创造一个新的与 `initState` 一样的 `ParseState` 值，不过其 `offset` 字段被设置成了指定的 `newOffset` 。

```
ghci> :l Parse.hs
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
ghci> let before = ParseState (L8.pack "foo") 0
ghci> let after = modifyOffset before 3
ghci> before
ParseState {string = "foo", offset = 0}
ghci> after
ParseState {string = "foo", offset = 3}
```

我们可以在花括号内部设置任意数量的字段，通过逗号进行分隔。

一个更有趣的解析器

现在让我们聚焦到编写一个更有意义的解析器上。我们暂时没有太大的野心：我们需要的是解析一个单字节。

```
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
```

定义中出现了若干新函数。

`L8.uncons` 函数从一个 `ByteString` 中接受首个元素。

```
ghci> L8.uncons (L8.pack "foo")
Just ('f',"oo")
ghci> L8.uncons L8.empty
Nothing
```

`getState` 函数获取当前的解析状态，而 `putState` 则是用于替换状态；`bail` 函数则是终结解析并报告异常；`(==>)` 函数将解析器链接起来。我们将稍后对每个函数进行解析。

Tip

Hanging lambdas

`parseByte` 的定义拥有一个视觉上的风格是我们之前没有讨论过的。它包含的匿名函数为参数以及 `->` 作为一行的结尾，而函数体起始于下一行。

这种方式的匿名函数并没有一个官方的名称，所以让我们称其“hanging lambda”。它主要的作用是为函数体留下更多的空间，同样可以将函数及其后续部分的关系在视觉上进行区分。例如通常而言，首个函数的结果会被当做参数传递给下一个函数。

获取与修改解析状态

`parseByte` 函数并不将解析状态作为一个参数，而是调用 `getState` 来获取一个状态的拷贝，`putState` 则是将当前状态替换。

```
getState :: Parse ParseState
getState = Parse (\s -> Right (s, s))

putState :: ParseState -> Parse ()
putState s = Parse (\_ -> Right ((), s))
```

当读取这些函数，记住元组的左元素是 `Parse` 的结果，而右元素则是当前解析状态。这使得接下来的函数变得更加方便使用。

`getState` 函数提取当前解析中的状态，使得调用者可以访问字符串。`putState` 函数替换当前解析中的状态。该状态通过 `==>` 链在下一个函数中可见。

这些函数让我们显式的移动状态在仅需要其所需的函数中。很多函数并不需要知道当前的状态，因此它们永远也不会调用 `getState` 或者 `putState` 。这相较于之前手动使用元组的解析器，这让我们可以编写出更简洁的代码，

我们将解析状态的细节打包进 `ParseState` 类型中，通过访问函数而不是模式匹配来进行工作。现在的解析状态是被隐式的传递，这为我们带来了极大的便利。如果我们希望添加更多的信息在解析状态中，我们仅需要修改 `ParseState` 的定义，以及任何需要这些新的信息的函数。相较于早期编写的解析器，所有的状态都被模式匹配所暴露，现在的代码则更加模块化：所有被影响的代码为需要新信息的代码。

报告解析异常

我们小心的定义 `Parse` 类型来兼容所有可能的异常。`==>` 组合子检查一个解析异常，并在出现异常时停止。然而我们仍未介绍 `bail` 函数，其用于报告解析异常。

```
bail :: String -> Parse a
bail err = Parse $ \s ->
  Left $
    "byte offset " ++ show (offset s) ++ ": " ++ err
```

在我们调用 `bail` 之后，`(==>)` 将会成功的在 `Left` 构造子上进行模式匹配包装错误信息，接着唤起链条中下一个解析函数。这将导致错误信息通过链条反向传递至上一级调用者。

链起所有解析器

`(==>)` 函数类似之前的 `(>>?)` 函数：它用于“粘合”使得所有函数链接在一起。

```
(==>) :: Parse a -> (a -> Parse b) -> Parse b
firstParser ==> secondParser = Parse chainedParser
  where
    chainedParser initState =
      case runParse firstParser initState of
        Left errMessage ->
          Left errMessage
        Right (firstResult, newState) ->
          runParse (secondParser firstResult) newState
```

`(==>)` 的函数体很有意思。回忆一下 `Parse` 类型是一个被包装装的函数。因为 `(==>)` 将两个 `Parse` 值链接生产出第三个值，它必须返回一个包装后的函数。

该函数不需要真正的“做”什么：它仅仅是创建一个闭包 *closure* 来记住 `firstParser` 与 `secondParser` 的值。

Tip

闭包其实就是一个带有环境的函数，其边界变量为其可见区域。闭包在 Haskell 中很常见。例如 `(+5)` 就是一个闭包。该闭包的实现必须将值 `5` 作为操作符 `(+)` 的第二个参数，因此无论函数传递的值是什么，其返回将会加五。

闭包不会被解包并执行，直到应用了 `parse` 函数。此时它将被应用至一个 `ParseState` 上，首先是应用 `firstParser` 并检验其结果。如果解析失败，闭包同样也会失败；否则它将传递解析的结果，以及新的 `ParseState` 至 `secondParser` 。

这真的是一个美妙的东西：我们高效的将 `ParseState` 隐式的传递至 `Parse` 链。（我们在稍后的章节中继续学习。）

函子简介

我们现在已经很熟悉 `map` 函数了，它将一个函数应用在一个列表中的每个元素上，返回很可能是另一种类型的列表。

```
ghci> map (+1) [1,2,3]
[2,3,4]
ghci> map show [1,2,3]
["1","2","3"]
ghci> :t map show
map show :: Show a => [a] -> [String]
```

`map` -类似的行为在其它实例中也很有用。例如，假设有一个二元树。

```
data Tree a
= Node (Tree a) (Tree a)
| Leaf a
deriving (Show)
```

如果我们想要将一个字符串树转换为一个包含了这些字符串长度的树，我们可以这样做：

```
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

现在可以将目光转向更为通用的函数：

```
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` 与 `treeMap length` 可以得到同样的结果。

```
ghci> :l TreeMap.hs
[1 of 1] Compiling Main             ( TreeMap.hs, interpreted )
Ok, one module loaded.
```

```
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 提供了一个著名的 typeclass 来泛化 `treeMap` 。该 typeclass 就是 `Functor` ，其定义为一个函数， `fmap` 。

我们可以认为 `fmap` 类似于之前小节中所提到的*提升 lifting* 函数。它接受类型为 `a -> b` 的函数，将其提升至应用在容器上的 `f a -> f b` 函数，这里的 `f` 即容器类型。

如果我们将 `f` 替换为 `Tree` 类型，那么 `fmap` 的类型应与 `treeMap` 一致，那么实际上我们可以用 `treeMap` 作为 `Tree` 的 `fmap` 实现。

```
instance Functor Tree where
fmap = treeMap
```

`Functor` 的定义强制了一些明显的限制在 `fmap` 上。例如我们仅能实现 `Functor` 实例在只有一个类型参数的类型上。

例如我们不可以将 `fmap` 实现在 `Either a b` 或 `(a, b)` 上，因为它们有两个类型参数。同样的，我们不能实现在 `Bool` 或 `Int` 上，因为它们没有类型参数。

另外，我们不能在类型约束上添加任何约束。这是什么意思呢？为了更好的进行解释，首先看一下普通的 `data` 定义以及其 `Functor` 实例。

```
data Foo a = Foo a

instance Functor Foo where
  fmap f (Foo a) = Foo (f a)
```

当我们定义一个新的类型，我们可以添加一个类型约束在 `data` 关键字之后。

```
data Eq a => Bar a = Bar a

instance Functor Bar where
  fmap f (Bar a) = Bar (f a)
```

报错如下：

```
• No instance for (Eq a) arising from a use of ‘Bar’
  Possible fix:
    add (Eq a) to the context of
      the type signature for:
        fmap :: forall a b. (a -> b) -> Bar a -> Bar b
• In the pattern: Bar a
  In an equation for ‘fmap’: fmap f (Bar a) = Bar (f a)
  In the instance declaration for ‘Functor Bar’ typecheck(-Wdeferred-type-errors)
```

类型定义上的约束并不好

在一个类型定义上添加一个约束这种行为基本上永远不会是一个好主意。它造成的影响便是强迫用户在每个函数上也都添加类型约束。假设我们需要一个栈数据结构，使得我们可以查询其元素是否遵从某种排序。下面是该数据类型的定义：

```
data (Ord a) => OrdStack a
  = Bottom
  | Item a (OrdStack a)
  deriving (Show)
```

如果我们希望编写一个函数用于检查栈结构是否增加，我们显然需要一个 `Ord` 约束来执行一对元素的比较。

```
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
  | a < b = isIncreasing rest
  | otherwise = False
isIncreasing _ = True
```

然而因为我们在类型定义上加上了类型约束，该约束实际上影响了完全不需要的地方：我们需要添加 `Ord` 约束至 `push` ，其本身并不关心堆上的顺序。

```
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

这个时候尝试移除上述函数的 `Ord` 约束，`push` 则会在类型检查时失败。

这也是为什么较早之前我们尝试着为 `Bar` 编写一个 `Functor` 实例时失败了：它需要 `Eq` 约束在 `fmap` 上。

现在我们有了 Haskell 中在类型定义上添加类型约束是不好的认知，那么更合理的做法呢？答案就是简单的移除在类型定义上的类型约束，且放置在真正需要约束的函数上。

fmap 的中缀使用

很多时候我们会看到 `fmap` 作为操作符一样调用：

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

如果希望作为操作符那样使用 `fmap` ，那么 `Control.Applicative` 模块中有一个操作符 `(<$>)` ，它是 `fmap` 的别名。名称中 `$` 展现了一种相似性，即将一个函数应用至其参数（通过 `($)` 操作符）以及将一个函数提升至一个函子。

灵活的实例

我们可能希望能为 `Either Int b` 实现一个 `Functor` 实例，因为它仅有一个类型参数。

```
instance Functor (Either Int) where
  fmap _ (Left n) = Left n
  fmap f (Right r) = Right (f r)
```

然而 Haskell 98 的类型系统不能保证，这样的一个实例约束的检查能终止。非终止的约束检查可能会让编译器陷入无限循环，因此禁用了这样形式的实例。

```
• 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)’ typecheck
```

原书使用一下代码解决编译出错的问题：

```
{-# LANGUAGE FlexibleInstances #-}

instance Functor (Either Int) where
  fmap _ (Left n) = Left n
  fmap f (Right r) = Right (f r)
```

然而实际上新的 Haskell 仍然不能通过编译，报错如下：

```
• Overlapping instances for Functor (Either Int)
    arising from a use of ‘GHC.Base.$dm<$’
  Matching instances:
    instance Functor (Either a) -- Defined in ‘Data.Either’
    instance Functor (Either Int)
      -- Defined at <...>/EitherIntFlexible.hs:7:10
• In the expression: GHC.Base.$dm<$ @(Either Int)
  In an equation for ‘<$’ : (<$) = GHC.Base.$dm<$ @(Either Int)
  In the instance declaration for ‘Functor (Either Int)’ typecheck(-Wdeferred-type-errors)
```

实际上，Haskell 的标准库早已更新，详见上一本书的笔记《Learn Me a Haskell》中第八章的“函子 typeclass”部分。Haskell 的标准实现如下：

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

因为有了 Haskell 的标准实现，我们可以直接进行以下操作：

```
ghci> fmap (== "cheeseburger") (Left 1 :: Either Int String)
Left 1
ghci> fmap (== "cheeseburger") (Right "fries" :: Either Int String)
Right False
```

更多有关函子的思考

我们已经做了一些有关函子该如何工作的隐式假设。那么将它们显式的展示出来并思考它们的规则就更有帮助了，因为这让我们可以将函子视为一致的，良好行为的对象。我们仅需记住两个简单的规则。

第一条规则就是函子必须是保持恒等 *identity* 的，也就是说将 `fmap id` 应用至一值时，返回的总是相同的值。

```
ghci> fmap id (Node (Leaf "a") (Leaf "b"))
Node (Leaf "a") (Leaf "b")
```

第二条规则则是函子必须是可组合的 *composable*，也就是说将两个 `fmap` 组合起来使用与分别两次 `fmap` 使用的结果是一致的。

```
ghci> (fmap even . fmap length) (Just "twelve")
Just True
ghci> fmap (even . length) (Just "twelve")
Just True
```

另一种看待这两天规则的方式是，函子必须保持形状 *shape*。集合的结构不应该受到函子的影响；只有它所包含的值应该改变。

```
ghci> fmap odd (Just 1)
Just True
ghci> fmap odd Nothing
Nothing
```

如果你在编写一个 `Functor` 实例，那么牢记这些规则并且进行了测试是很有帮助的，因为编译器并不能上述列举的规则。另一方面，如果仅仅使用函子，那么规则是“自然而然的”无需记住它们。

为解析编写函子实例

对于已经调研过得这些类型，我们对 `fmap` 所预期的行为就显而易见了。相较于 `Parse` 因为复杂度的缘故会较难理解。一个合理的猜想就是我们所 `fmap` 的函数应该被应用在一个解析器的当前结果上，同时不对解析状态做任何修改。

```
instance Functor Parse where
fmap f parser =
  parser ==> \result ->
    identity (f result)
```

这个定义很容易阅读，让我们用几个快速的测试来检查我们是否遵循了函子的规则。

首先检查的是 identity 是否被遵循。首先是一个应该失败的解析：从一个空的字符串中解析一个字节（别忘了 `(<$>)` 就是 `fmap` ）。

```
ghci> :l Parse.hs
[1 of 2] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
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"
```

现在测试一下成功的案例：

```
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
```

通过上述结果同样可知我们的函子实例遵循了第二条规则，即保持形状。失败保持失败，成功保持成功。

最后就是组合起来也是保存原有形状的。

```
ghci> :m +Data.Char
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
```

为解析使用函子

所有关于函子的讲解都是为了一个目的：它们可以让我们编写简洁有力的代码。回忆一下之前介绍过的 `parseByte` 函数。我们总是想要处理 ASCII 字符而不是 `Word8` 值。

我们可以模仿 `parseByte` 那样编写一个 `parseChar` 函数，不过现在可以通过 `Parse` 的函子特性来避免这些重复的代码。函子接受一个解析的结果并为该结果应用一个函数，因此我们需要一个函数可以将 `Word8` 转换成一个 `Char` 。

```
w2c :: Word8 -> Char
w2c = chr . fromIntegral

parseChar :: Parse Char
parseChar = w2c <$> parseByte
```

可以使用函子来编写一个简洁的“peek”函数。它将在输入的字符串结束时返回 `Nothing` ；否则返回下一个字符，同时不消耗它（即检查时不会影响当前的解析状态）。

```
peekByte :: Parse (Maybe Word8)
peekByte = fmap fst . L.uncons . string <$> getState
```

同样的方法可以为 `parseChar` 定义一个更加简洁的 `peekChar` 版本。

```
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

注意 `peekByte` 与 `peekChar` 都调用了 `fmap`，即 `(<$>)`。这是因为 `Parse (Maybe a)` 类型是一个函子中的函子。因此我们需要将一个函数提升两次来“走进”里面的函子。

最后让我们编写另一个泛用的组合子，也就是类似于 `takeWhile` 版本的 `Parse`：消费输入直到谓词返回 `True`。

```
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p =
  (fmap p <$> peekByte) ==> \mp ->
    if mp == Just True
      then
        parseByte ==> \b ->
          (b :) <$> parseWhile p
      else identity []
```

下面是一个不使用函子的繁琐版本，不过更加直接：

```
parseWhileVerbose p =
peekByte ==> \mc ->
  case mc of
    Nothing -> identity []
    Just c
      | p c ->
          parseByte ==> \b ->
            parseWhileVerbose p ==> \bs ->
              identity (b : bs)
      | otherwise ->
          identity []
```

这个繁琐的版本在不熟悉函子时虽然看似易读，但是在 Haskell 中使用函子更为常见，它的表达更加简洁，因此应该同时成为读与写的本能。

重写 PGM 解析器

没有新的解析代码时，原始的 PGM 解析函数会是什么样子呢？

```
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")
```

该定义又使用了以下的帮助函数：

```
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 ==> const f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True _ = identity ()
assert False err = bail err
```

`(==>&)` 组合子像 `(==>)` 那样将解析器链接起来，不过右侧的解析器会无视左侧所返回的结果。`assert` 函数允许我们检查一个属性，当该属性为 `False` 时会终止解析并返回一个有用的错误信息。

最后就是检查与修改解析状态。

```
parseBytes :: Int -> Parse L.ByteString
parseBytes n =
  getState ==> \st ->
    let n' = fromIntegral n
        (h, t) = L.splitAt n' (string st)
        st' = st {offset = offset st + L.length h, string = t}
     in putState st'
          ==>& assert (L.length h == n') "end of input"
```

```
9             =>& identity h
```

未来的方向

略

11 Testing and quality assurance

构建真实系统意味着需要考虑质量控制，鲁棒性以及正确性。使用正确的质量保证机制，良好的代码类似于一个各项功能完备的精确的仪器。

Haskell 中有着若干工具帮助我们构建这样精准的系统。最显著的当然是语言自带的类型系统，它允许静态的强制执行复杂的不变量 – 使得编写的代码无法违背其约束。除此之外，纯粹性以及多态鼓励模块化，可重构和可测试的代码风格。

在维护代码的正确表达性上测试担任了很重要的角色。Haskell 中主要的测试机制是传统的单元测试（通过 HUnit 库），以及其更强大的派生品：基于类型的“属性”测试，即 QuickCheck 一个开源的 Haskell 测试框架。基于属性的测试鼓励更高层次的测试方法，是以抽象不变函数的形式测试，需要满足测试库所生成真实的测试数据。相较于手写测试用例的不足，通过这样测试的代码通常能覆盖所有的边缘用例。

本章我们学习如何使用 QuickCheck 来构建代码的不变性，并测试上一章节编写的代码。另外还会学习如何使用 GHC 代码覆盖工具：HPC。

QuickCheck：基于类型的测试

为了展示基于属性的测试是如何工作的，我们用一个简单的场景开始：用户编写了一个特殊的排序函数，想要测试其行为。

首先是导入 QuickCheck 库，正如其他所需的包那样（通过以下命令在 `.cabal` 存在的项目下安装包 `cabal install --lib QuickCheck` ）：

```
import Test.QuickCheck
import Data.List
```

以及希望被测试的函数 – 一个自定义的排序：

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort lhs ++ [x] ++ qsort rhs
  where
    lhs = filter (< x) xs
    rhs = filter (>= x) xs
```

这是一个典型的 Haskell 排序实现：学习函数式编程的优雅，不过没那么高效（这并不是一个替换排序）。现在我们希望检查该函数是否遵循基本的排序规则。一个不错的不变性测试就是幂等性 – 运行一个函数两次的返回是否一致。这个不变性的测试不难：

```
prop_idempotent xs = qsort (qsort xs) == qsort xs
```

我们将使用 QuickCheck 来转换带有 `prop_` 的前缀测试属性，用于区别普通的代码。这个幂等性的属性被简单的编写为 Haskell 函数来表明任何输入的数据被排序后是否具有相等性。我们通过手动测试若干案例的方式来检查它是否合理：

```
ghci> :l QC-basics.hs
[1 of 2] Compiling Main             ( QC-basics.hs, interpreted )
Ok, one module loaded.
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，不过手动输入数据还是太麻烦了，同时也违背了函数式编程的高效性：让机器做这事儿！QuickCheck 库带有 Haskell 所有数据类型的数据生成器可以用于自动化手动输入这个过程。通过类型系统所生成的（临时）随机数据，QuickCheck 使用 `Arbitrary` typeclass 来表示的统一接口。QuickCheck 通常会隐藏数据生成的过程，不过我们仍然可以通过手动的方式运行生成器来获取 QuickCheck 所生成的数据分布。例如，生成一个随机布尔值列表。

```
ghci> generate arbitrary :: IO [Bool]
[False,False,True,False,True,False,False]
```

注：新的 QuickCheck 的 `generate` 的函数签名是 `generate :: Gen a -> IO a` 。

QuickCheck 生成的测试数据就像是这样，并通过 `quickCheck` 函数传递给所选择的属性。属性其本身的类型决定了数据生成器的使用。`quickCheck` 接着检查属性是否满足所有生产的数据。

```
ghci> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
ghci> quickCheck (pro)
product                   properFraction            propertyForAllShrinkShow
prop_idempotent           property
ghci> quickCheck (prop_idempotent :: [Integer] -> Bool)
+++ OK, passed 100 tests.
```

对于生成的 100 个不同的列表，属性都可以适配。在开发测试时，看到每个测试用例中真实生成的数据是很有意义的。为了达到这个效果，可以使用 `verboseCheck` 函数来查看更详细的输出。

测试属性

好的库包含了一系列正交的基础，它们彼此之间都有合理的关联。使用 QuickCheck 来指定函数间的关系，通过开发属性间合理交互的函数来帮助我们找到好的库接口。QuickCheck 这时所扮演的是 API 的“lint”工具 – 其提供了确保我们库 API 合理性的机器支持。

列表排序函数当然拥有若干有意思的属性，它们与其它的列表操作紧密关联。例如：排序列表中第一个元素总是最小的元素。我们可以在 Haskell 中指定这一特性，通过 `List` 库的

`minimum` 函数：

```
prop_minimum xs = head (qsort xs) == minimum xs
```

不过会抛出异常：

```
ghci> quickCheck (prop_minimum :: [Integer] -> Bool)
*** Failed! (after 1 test):
Exception:
  Prelude.head: empty list
  CallStack (from HasCallStack):
    error, called at libraries/base/GHC/List.hs:1646:3 in base:GHC.List
    errorEmptyList, called at libraries/base/GHC/List.hs:85:11 in base:GHC.List
    badHead, called at libraries/base/GHC/List.hs:81:28 in base:GHC.List
    head, called at QC-basics.hs:21:19 in main:Main
[]
```

也就是这个属性只满足非空列表。感谢 QuickCheck 有着一整套属性使得我们可以更加精确的指定我们的不变，剔除不希望考虑到的值。对于非空列表，我们真实想表达的是：如果列表是非空的，那么排序列表中的第一个元素是最小的。根据 `(==>)` 隐式函数，在运行属性前剔除非法的数据：

```
prop_minimum' xs = not (null xs) ==> head (qsort xs) == minimum xs
```

通过分离空列表，现在可以确认属性实际上成立：

```
ghci> quickCheck (prop_minimum' :: [Integer] -> Property)
+++ OK, passed 100 tests; 18 discarded.
```

注意我们将属性的类型从简单的 `Bool` 结果变为了更泛化的 `Property` 类型（属性自身现在是一个函数，在测试前筛选非空列表，而不是简单的一个布尔常数）。

我们现在可以通过其它应该满足的不变来完成排序函数的所有基础属性了：输出是有序的（每个元素都应该小于等于其前者）；输出是输入的组合（通过列表差异函数 `(\\)` ；最后被排序的元素是最大的元素；如果找到了两个不同列表的最小元素，那么将两个列表合并后再排序，该元素还是第一个元素。这些属性可以表示为：

```
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 =
```

```
14  not (null xs)
15    ==> not (null ys)
16    ==> head (qsort (xs ++ ys))
17    == min (minimum xs) (minimum ys)
```

测试模型

另一个技巧就是测试模型的实现。可以将列表排序的实现绑定标准库中的排序函数，同时如果它们表现的一致，就可以得知自定义的排序的正确性了。

```
1  prop_sort_model xs = sort xs == qsort xs
```

这一类基于模型的测试异常强大。通常开发人员会有一个参考现实或原型，虽然不够高效，但却是正确的。这可以保存下来，并用于确保优化后的产品代码符合这套参考。通过构建一套大型的基于模型的测试，并定期运行它们（例如在每次提交时），我们可以轻松的确保代码的准确性。大型 Haskell 项目通常绑定了与项目本身大小相当的属性套件，在每次更改时都要测试数千个不变量，保证代码符合规范，并确保能按要求运行。

测试用例：指定一个漂亮打印器

在 Haskell 所构建的大型系统中中，为单独的函数测试它们的自然属性属于基础的构建模块。接下来我们将要学习更复杂的场景：为之前章节所编写的漂亮打印库构建测试套装。

生成测试数据

回忆一下漂亮打印是基于 `Doc` 所构建的程序，它是一个代表着良好格式化后的文档的代数数据结构。

```
1  data Doc
2    = Empty
3    | Char Char
4    | Text String
5    | Line
6    | Concat Doc Doc
7    | Union Doc Doc
8    deriving (Show, Eq)
```

库本身是实现的一系列函数用于构建与转换该文档类型的值，最后将完成的文档渲染成字符串。

QuickCheck 鼓励采用一种测试方法，开发人员指定的不变量可以满足任何数据类型。为了测试漂亮打印库，我们还需要输入数据源。利用 QuickCheck 中 `Arbitrary` 类所提供的组合字套件来构建随机数据。该类提供了一个函数，`arbitrary` ，用于生成每种类型的数据，通过它我们可以为自定义的数据类型定义数据生成器。

```haskell
class Arbitrary a where
  arbitrary :: Gen a
```

注：该 typeclass 由 QuickCheck 库提供。

需要注意的是生成器运行在一个 `Gen` 环境中，由类型表示。这是一个简单的状态传递单子，用于隐藏贯穿代码的随机数生成器状态。我们会在稍后的章节中讲解单子，现在我们只需要知道 `Gen` 定义为单子，可以使用 `do` 语义来编写可访问隐式随机数源的生成器。为了编写自定义类型的生成器，我们使用库中定义的一组函数来引入新的随机值，并将它们粘合在一起用以构建我们需要的数据结构类型。关键函数的类型为：

```haskell
elements :: [a] -> Gen a
choose :: Random a => (a, a) -> Gen a
oneof :: [Gen a ] -> Gen a
```

注：以上三个函数皆由 QuickCheck 库提供。

函数 `elements` ，接受一个列表，返回一个生成器，该生成器将随机返回该列表中的值。`choose` 与 `oneof` 将在稍后用到。通过它们我们可以为一个简单的类型编写生成器。例如：

```haskell
data Ternary
  = Yes
  | No
  | Unknown
  deriving (Eq, Show)
```

为其实现 `Arbitrary` typeclass：

```haskell
instance Arbitrary Ternary where
  arbitrary = elements [Yes, No, Unknown]
```

另外一种数据生成的方案就是先生成一种 Haskell 的基础类型，再转换成期望的实际类型。通过 `choose` 来选择 0 至 2 的随机整数，然后再映射成 ternary 值：

```haskell
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 *sum* 类型，这种方法可以很好的工作，因为整数可以很好的映射到数据类型的构造函数上。对于 *product* 类型（例如结构或元组），我们需要为每个成员分别生成乘积（递归嵌套类型），再将成员进行合并。例如生成一对随机值：

```haskell
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
  arbitrary = do
    x <- arbitrary
    y <- arbitrary
    return (x, y)
```

注：上述代码同样由 QuickCheck 库提供。

现在来为所有 `Doc` 类型的变体来编写一个生成器。我们需要将问题进行分解，首先是为每个类型生成随机构造函数，接着根据结果生成每个字段的组件。这里最复杂的部分就是 union 以及 concatenation 变体了。

首先，我们需要编写一个实例来生成随机字符 – QuickCheck 并没有默认的字符实例，因为有大量不同的文本编码，我们希望一个字符测试。我们无需考虑文档中的文本内容，因此简单的字母以及标点符号的生成器就足够了（更丰富的生成器可以基于该生成器进行拓展）：

```
newtype DocChar = DocChar Char

fromDocChar :: DocChar -> Char
fromDocChar (DocChar x) = x

instance Arbitrary DocChar where
  arbitrary = elements (DocChar <$> ['A' .. 'Z'] ++ ['a' .. 'z'] ++ " ~!@#$%^&*()")
```

注：与原文不同在于使用了 `newtype` ，因为 QuickCheck 已经为 `Char` 实现了 `Arbitrary` 。这里 `(<$>)` 的计算优先级是 4，而 `(++)` 的优先级是 5，因此不需要额外的括号。

有了这些现在就可以编写文档的实例了，即通过枚举构造函数，并填充其字段。我们选择随机整数来表示所需要生成的文档变体，再根据结果进行分配。生成 concat 与 union 文档节点时，只需要递归 `arbitrary` ，将类型推导交给 `Arbitrary` 的实例本身：

```
instance Arbitrary Doc where
  arbitrary = do
    n <- choose (1, 6) :: Gen Int
    case n of
      1 -> return Empty
      2 -> do
        (x :: DocChar) <- arbitrary
        return (Char $ fromDocChar x)
      3 -> Text <$> arbitrary
      4 -> return Line
      5 -> do
        x <- arbitrary
        Concat x <$> arbitrary
      6 -> do
        x <- arbitrary
        Union x <$> arbitrary
```

注：与原文不同之处在于处理 `Char` 的步骤，先是显式声明了 `x` 为 `DocChar` ，然后再使用 `fromDocChar` 取出 `Char` 值。

上述代码非常的直接，我们可以通过 `oneof` 函数对其进行重写，该函数我们较早之前见过，其作用是在列表中挑选一个后生成生成器（我们也可以使用单子化的组合子，用 `liftM` 来避免命名生成器的各种中间结果）：

```haskell
instance Arbitrary Doc where
arbitrary =
  oneof
    [ return Empty,
      fmap (Char . fromDocChar . DocChar) arbitrary,
      fmap Text arbitrary,
      return Line,
      liftM2 Concat arbitrary arbitrary,
      liftM2 Union arbitrary arbitrary
    ]
```

注：与原文不同之处在于列表第二个元素 `Char` 的处理方式上，通过 newtype `DocChar` 的 `arbitrary` 函数再提取出 `Char` 来实现调用非默认的 `Char` arbitrary。

上述两种对 `Doc` 实现 `Arbitrary` 实例的方式，后者更为简洁，仅仅是在生成器列表中进行选择，不过二种表达方式讲述的都是同一件事。我们可以通过生成一个随机文档列表来进行测试

```
ghci> generate arbitrary :: IO [Doc]
[Concat (Union (Text "1^_\FS\83431\nir^\DC1IH\t\GS\10583r\"_~t\1007943(\1091024\180519\ETX
  \53558y\998555") Line) (Union (Text "s4hxR(\1108184") Empty),Empty,Empty,Char '&',Char 'D',
  Empty,Line,Empty,Line,Char '>',Te
xt "\99986\25864;'}\CAN",Concat Line Empty,Text "\NAK",Char '&',Union Empty (Concat (Char
  '\998638') (Union
 (Text "/\t\28481\1025790\95127\1078739x;D") (Concat (Char 'Q') Line))),Empty,Concat Line (
  Text "Gwj^v\1032477\EOT\58278k},`<\1016570\59993:Y>\1074673q;A \7956\127844"),Empty,Concat
  Empty (Text "\199725\SIl(<\176480\133060,A\FSu\EM>\DC1\SYN\1102841\RS\STX\DELim\n
  \"|\1106202De\n#"),Char 'J',Union Empty Line,Concat (Union
 (Concat Line (Char 'E')) (Char '\NUL')) (Concat (Text "78\152486\194828?vy\157411\\1*\DC1\
  NAK\NULrd\SYN:\t
") Line),Text ""]
```

注：与原文不同，`generate` 的函数签名变了。

结果是各种文档变体混合的列表。之后在每次测试运行时会生成上百个结果。现在可以为文档函数编写一些更泛化的属性了。

测试文档构建

基于文档的两个基础函数是 null 文档常数（一个 nullary 函数），`empty` ，以及 append 函数。它们的类型是：

```haskell
empty :: Doc
(<>) :: Doc -> Doc -> Doc
```

总之，它们有一个很好的属性：将空列表 append 或 prepend 到第二个列表上，第二个列表保持不变。我们可以将这个不变量声明为一个属性：

```haskell
prop_empty_id x =
```

```
  empty <> x == x
    && x <> empty == x
```

注: 原有 Doc 代码在第五章的 `Prettify.hs` 文件中（改动后位于 code/Ch11/Prettify2.hs 文件下）。与第五章不一样的地方：为 `Doc` 实现了 `Semigroup` ，因此可以对 `Doc` 使用操作符 `(<>)` 。

为了确保它是正确的，测试一下：

```
ghci> quickCheck prop_empty_id
00, passed 100 tests.
```

将 `quickCheck` 改为 `verboseCheck` 可以看到生成出来的测试文档。

其它的 API 测试：

```
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)
```

以列表作模型

高阶函数是重用程序的基本粘合剂，漂亮打印库也不例外 – 自定义的 fold 函数用于内部实现文档拼接以及文档块之间的交错分隔符。文档的 `fold` 接受一个文档列表，通过提供的组合函数将它们粘合起来：

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

那么为 fold 编写测试就很简单了。

```
prop_hcat xs = hcat xs == glue xs
  where
    glue [] = empty
    glue (d : ds) = d <> glue ds
```

同理对于 `punctuate` ，我们可以通过列表散布（interspersion）的方式来插入标点符号：

```
import Data.List (intersperse)

prop_punctuate s xs = punctuate s xs == intersperse s xs
```

然而对其进行测试时，我们会得到错误：

```
ghci> quickCheck prop_punctuate
*** Failed! Falsified (after 6 tests and 1 shrink):
Char ' '
[Union (Text "\ESC") (Text "RO"),Union Empty (Char '\1097718')]
```

漂亮打印库优化掉了多余的空文档，这是模型实现无法做到的，因此我们需要增强模型以匹配实际情况。首先，可以在整个文档列表中散布标点符号文本，然后用一个小的循环来清理分散在其中的 `Empty` 文档，像是这样：

```
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
```

通过 CHCi 来确认结果。

```
ghci> :l QC.hs
[1 of 4] Compiling Prettify2        ( Prettify2.hs, interpreted )
[2 of 4] Compiling Arbitrary        ( Arbitrary.hs, interpreted )
[3 of 4] Compiling Main             ( QC.hs, interpreted )
Ok, three modules loaded.
ghci> quickCheck prop_punctuate'
+++ OK, passed 100 tests.
```

将它们合在一起

我们可以将所有的测试放到一个单独文件中，然后使用 QuickCheck 的派生函数来运行。基本的批处理驱动程序通常已经足够好了。

```
{-# LANGUAGE TemplateHaskell #-}
import Test.QuickCheck.All

-- All the prop tests before `return []`

return []
runTests = $quickCheckAll
```

注：根据StackOverflow的答案，QuickCheck 在 2.0 之后就没了 `Test.QuickCheck.Batch`，因此需要在 `QC.hs` 文件的头尾加上上述代码。而新的 `Run.hs` 代码如下：

```
import QC

main = do
  runTests
  return ()
```

通过 GHCi 测试：

```
ghci> :l Run.hs
[1 of 5] Compiling Prettify2        ( Prettify2.hs, interpreted )
[2 of 5] Compiling Arbitrary        ( Arbitrary.hs, interpreted )
```

```
[3 of 5] Compiling QC               ( QC.hs, interpreted )
[4 of 5] Compiling Main             ( Run.hs, interpreted )
Ok, four modules loaded.
ghci> main
=== prop_empty_id from ./QC.hs:14 ===
+++ OK, passed 100 tests.

=== prop_char from ./QC.hs:18 ===
+++ OK, passed 100 tests.

=== prop_text from ./QC.hs:20 ===
+++ OK, passed 100 tests.

=== prop_line from ./QC.hs:22 ===
+++ OK, passed 1 test.

=== prop_double from ./QC.hs:24 ===
+++ OK, passed 100 tests.

=== prop_hcat from ./QC.hs:28 ===
+++ OK, passed 100 tests.

=== prop_punctuate from ./QC.hs:33 ===
*** Failed! Falsified (after 4 tests):
Char 'H'
[Concat (Char '1') (Concat (Char '\119225') (Concat (Char 'r') Empty)),Text ""]

=== prop_punctuate' from ./QC.hs:35 ===
+++ OK, passed 100 tests.
```

我们在这里将代码构建为一个独立的测试脚本，实例和属性在它们自己的文件中，做到了与库的源代码分离。这是一个典型的库项目，测试与库本身分开，并通过模块系统导入库。然后可以编译和执行测试脚本：

```
ghc --make Run.hs
Loaded package environment from <xxx>
[1 of 5] Compiling Prettify2        ( Prettify2.hs, Prettify2.o, Prettify2.dyn_o )
[2 of 5] Compiling Arbitrary        ( Arbitrary.hs, Arbitrary.o, Arbitrary.dyn_o )
[3 of 5] Compiling QC               ( QC.hs, QC.o, QC.dyn_o )
[4 of 5] Compiling Main             ( Run.hs, Run.o )
[5 of 5] Linking Run
```

接着运行：

```
./Run
=== prop_empty_id from ./QC.hs:14 ===
+++ OK, passed 100 tests.

=== prop_char from ./QC.hs:18 ===
+++ OK, passed 100 tests.
```

```
=== prop_text from ./QC.hs:20 ===
+++ OK, passed 100 tests.

=== prop_line from ./QC.hs:22 ===
+++ OK, passed 1 test.

=== prop_double from ./QC.hs:24 ===
+++ OK, passed 100 tests.

=== prop_hcat from ./QC.hs:28 ===
+++ OK, passed 100 tests.

=== prop_punctuate from ./QC.hs:33 ===
*** Failed! Falsified (after 4 tests):
Union (Char '\1089590') (Char 't')
[Line,Text "\129350\167965\EOT"]

=== prop_punctuate' from ./QC.hs:35 ===
+++ OK, passed 100 tests.
```

通过 HPC 测量测试覆盖率

HPC (Haskell Program Coverage) 是编译器的扩展，用于观察给定程序运行期间实际执行的代码部分。这在测试的上下文中很有用，因为它可以让我们精确的观察哪些函数、分支以及表达式被求值。HPC 附带了一个简单的工具来生成有用的程序覆盖图，这可以放大测试套件中的弱点。

我们需要在编译时添加 `-fhpc` 标记：

```
ghc -fhpc Run.hs --make
Loaded package environment from /Users/jacobxie/.ghc/aarch64-darwin-9.4.7/environments/
  default
[1 of 5] Compiling Prettify2        ( Prettify2.hs, Prettify2.o, Prettify2.dyn_o ) [HPC flags
   changed]
[2 of 5] Compiling Arbitrary        ( Arbitrary.hs, Arbitrary.o, Arbitrary.dyn_o ) [HPC flags
   changed]
[3 of 5] Compiling QC               ( QC.hs, QC.o, QC.dyn_o ) [Source file changed]
[4 of 5] Compiling Main             ( Run.hs, Run.o ) [HPC flags changed]
[5 of 5] Linking Run [Objects changed]
```

在测试运行期间，程序的跟踪记录被写入至当前目录下的.tix 与.mix 文件中。然后命令行工具 hpc 使用这些文件来显示有关的各种统计信息。首先，可以使用 hpc 报告标志获得运行期间测试代码的摘要。我们将排除测试程序本身（使用 `--exclude` 标志），以便只关注库的代码：

```
hpc report Run --exclude=Main --exclude=QC
26% expressions used (54/202)
```

```
  0% boolean coverage (0/3)
       0% guards (0/3), 3 unevaluated
      100% 'if' conditions (0/0)
      100% qualifiers (0/0)
 21% alternatives used (8/37)
  0% local declarations used (0/4)
 44% top-level declarations used (16/36)
```

最后一行可以看到 42% 的定义在测试运行期间被评估。第一次尝试还算不错。随着我们从库中测试越来越多的函数，这个数字将会上升。文本版本的测试摘要对于快速总结很有用，弹药真正了解发生了什么，那么就可以使用 `markup` 标记：

```
hpc markup Run --exclude=Main --exclude=QC
Writing: Arbitrary.hs.html
Writing: Prettify2.hs.html
Writing: hpc_index.html
Writing: hpc_index_fun.html
Writing: hpc_index_alt.html
Writing: hpc_index_exp.html
```

略

12 Barcode recognition

本章我们将会使用第十章所编写的图像解析库来构建一个识别条形码的应用。

关于条形码

绝大多数包装好的和批量生产的消费品上面都会有条形码。尽管在各个专门的领域都有若干的条形码系统，但消费品通常使用 UPC-A 或 EAN-13。前者是在美国开发的，而后者则起源于欧洲。

EAN-13 开发晚于 UPC-A，它是 UPC-A 的超集。（实际上，UPC-A 已经在 2005 年宣布废弃了，尽管在美国它仍然被大量的使用。）任何可以识别 EAN-13 的软件或硬件也会自动处理 UPC-A 条形码。

正如其名，EAN-13 描述了一个 13 位数的序列，它被划分成四个组：

- 开头的两位数描述了*数字系统*。它可以表明制造商的国家，或是其它的一些分类，例如 ISBN（书籍身份）。
- 接下来的五位数则是生产商的 ID，通过国家的授权进行指定。
- 再接下来的五位数则是产品 ID，通过制造商指定。（小型的制造商可能有更长的制造商 ID 以及更短的产品 ID，但是它们加在一起仍然是十位数。）
- 最后一位则是*校验数*，允许一个扫描器验证它扫描的位数字符串。

EAN-13 编码

在我们关心 EAN-13 条形码的解码之前，我们需要理解它们是如何被编码的。EAN-13 所使用的系统有一点复杂。让我们首先从校验数开始，也就是字符串的最后一个位数。

```
1 checkDigit :: (Integral a) => [a] -> a
2 checkDigit ds = 10 - (sum products `mod` 10)
3   where
4     products = mapEveryOther (* 3) (reverse ds)
5
6 mapEveryOther :: (a -> a) -> [a] -> [a]
7 mapEveryOther f = zipWith ($) (cycle [f, id])
```

这是一种通过代码比口头描述更容易理解的算法。从字符串的右侧开始计算，每个连续的数字要么乘以 3，要么不动（`cycle` 函数会无限的重复它的输入列表）。校验数则是它们的和（以 10 取模）与数字 10 之间的差。

条形码是一系列固定宽度的条码，黑色代表二元“1”位，白色代表“0”位。因此，相同数字的运行看前来像一个更粗的条码。

条形码中的位序列如下所示：

- 前导保护序列，编码为 101.
- 一组六位数字，每个有七位宽。
- 另一个保护序列，编码为 01010.
- 一组六位以上的数字。
- 尾部的保护序列，编码为 101。

左边和右边组中的数字有单独的编码。左边用奇偶校验位编码，奇偶校验位编码条形码的第 13 位。

数组介绍

在继续之前，以下是本章所需的模块导入：

```
1  import Control.Applicative ((<$>))
2  import Control.Monad (forM_)
3  import Data.Array (Array (..), bounds, elems, indices, ixmap, listArray, (!))
4  import Data.ByteString.Lazy.Char8 qualified as L
5  import Data.Char (digitToInt)
6  import Data.Ix (Ix (..))
7  import Data.List (foldl', group, sort, sortBy, tails)
8  import Data.Map qualified as M
9  import Data.Maybe (catMaybes, listToMaybe)
10 import Data.Ratio (Ratio)
11 import Data.Word (Word8)
12 import Parse                              -- from Chapter 10
13 import System.Environment (getArgs)
```

注： `Parse` 库从第十章而来。

条形码的编码过程很大程度上是表格驱动的，即使用位模式的小表来决定如何对每个数字进行编码。Haskell 基本的数据类型中列表和元组都不适合用于表格，其包含的元素需要能被随机访问。列表需要线性的遍历才能访问第 k 个元素。元组没有这个问题，但是 Haskell 的类型系统使得很难编写一个函数来接受元组和元素偏移量，并返回元组中偏移量处的元素。

常用的数据类型可供常数耗时的随机访问就是数组了。Haskell 提供了若干数字类型。这里我们将编码表格表达为字符串数组。

最简单的数组类型位于 `Data.Array` 模块中，也正是我们现在所用的。与其它 Haskell 通常的类型一样，这些数组也都是不可变的。不可变数组只有在创建时才能用值填充一次，其它时候是不能修改其内容的。（标准库同样提供了其它的数组类型，有些是可变的，只不过现在我们暂时还用不上。）

```
leftOddList =
  [ "0001101",
    "0011001",
    "0010011",
    "0111101",
    "0100011",
    "0110001",
    "0101111",
    "0111011",
    "0110111",
    "0001011"
  ]

rightList = map complement <$> leftOddList
  where
    complement '0' = '1'
    complement '1' = '0'

leftEvenList = map reverse rightList

parityList =
  [ "111111",
    "110100",
    "110010",
    "110001",
    "101100",
    "100110",
    "100011",
    "101010",
    "101001",
    "100101"
  ]

listToArray :: [a] -> Array Int a
listToArray xs = listArray (0, l - 1) xs
  where
    l = length xs

leftOddCodes, leftEvenCodes, rightCodes, parityCodes :: Array Int String
leftOddCodes = listToArray leftOddList
leftEvenCodes = listToArray leftEvenList
rightCodes = listToArray rightList
parityCodes = listToArray parityList
```

`Data.Array` 模块的 `listArray` 函数将列表转为数组。第一个参数是数组所需的边界；第二个参数则是列表。

`Array` 的一个不寻常特征是，它的类型在它包含的数据和索引类型上都是参数化的。一维 `String` 数组的类型是 `Array Int String` ，而二维则是 `Array (Int,Int) String` 。

```
ghci> :m +Data.Array
ghci> :t listArray
listArray :: Ix i => (i, i) -> [e] -> Array i e
```

构造一个数组：

```
ghci> listArray (0,2) "foo"
array (0,2) [(0,'f'),(1,'o'),(2,'o')]
```

注意我们指定了数组的上下界。这些界是包含边界的，即从 0 至 2 的数组包含了 0，1 以及 2。

```
ghci> listArray (0,3) [True,False,False,True,False]
array (0,3) [(0,True),(1,False),(2,False),(3,True)]
ghci> listArray (0,10) "too short"
array (0,10) [(0,'t'),(1,'o'),(2,'o'),(3,' '),(4,'s'),(5,'h'),(6,'o'),(7,'r'),(8,'t'),(9,***
  Exception: (Array.!): undefined array element
```

当数组被构建后，可以使用 `(!)` 操作符根据索引来访问元素。

```
ghci> let a = listArray (0,14) ['a'..]
ghci> a ! 2
'c'
ghci> a ! 100
*** Exception: Ix{Integer}.index: Index (100) out of range ((0,14))
```

由于数组的构造函数允许指定数组的边界，我们无需使用类似于 C 语言那样基于零的边界。

```
ghci> let a = listArray (-9,5) ['a'..]
ghci> a ! (-2)
'h'
```

索引类型可以是任意 `Ix` 类型的成员。例如 `Char` 作为索引：

```
ghci> let a = listArray ('a','h') [97..]
ghci> a ! 'e'
101
```

创建一个高阶维度数组，我们可以使用 `Ix` 实例的元组作为索引类型。以下是一个三维数组：

```
ghci> let a = listArray ((0,0,0),(9,9,9)) [0..]
ghci> a ! (4,3,7)
437
```

数组与惰性

用于构造数组的列表，其元素数必须大于等于数组所需的元素。如果没有提供足够的元素则会在运行时抛出异常。该异常将何时出现取决于数组的具体实现。

这里我们使用的是一个非严格的数组类型。

```
ghci> let a = listArray (0,5) "bar"
ghci> a ! 2
'r'
ghci> a ! 4
*** Exception: (Array.!): undefined array element
```

Haskell 同样提供了严格数组，其行为就不同了。我们将在后续小节中讨论两者的优劣。

折叠数组

`bounds` 函数返回一个描述了创建数组时的边界元组。`indices` 函数返回列表的每个索引。我们可以用其定义一些有用的折叠，因为 `Data.Array` 模块并没有定义任何折叠函数。

```
-- Strict left fold, similar to foldl' on lists
foldA :: (Ix k) => (a -> b -> a) -> a -> Array k b -> a
foldA f s a = go s (indices a)
  where
    go s (j : js) = let s' = f s (a ! j) in s' `seq` go s' js
    go s _ = s

-- Strict left fold using the first element of the array as its starting value, similar to
  fold1 on lists
foldA1 :: (Ix k) => (a -> a -> a) -> Array k a -> a
foldA1 f a = foldA f (a ! fst (bounds a)) a
```

你可能会疑惑为什么数组模块并没有提供如此有用的折叠函数。这是因为一维数组与列表之间有一些显著的差异。例如自然而言，折叠有两个方向：从左向右，从右向左。除此之外，每次只能折叠一个元素。

而对于二维的数组来说这根本行不通。首先，这时的折叠可以有多种方案。我们可能还是想每次只折叠单个元素，但是现在有了列折叠或行折叠的可能性。最重要的是，对于一次一个元素的折叠，不再只有两个序列需要遍历。

换言之，对于二维数组，就存在若干可能的折叠方式组合，因此没有太多令人信服的理由为标准库选择少量排列。这个问题在高维情况下会更加复杂，因此最好让开发人员编写适合自身应用需求的折叠。

修改数组元素

对于不变数组而言，虽然存在“修改”函数，实际上并不常用。例如，`accum` 函数接受一个数组，以及一个 `(index, value)` 对的列表，返回一个新的被替换后的数组。

由于数组是不变的，修改一个元素就需要拷贝整个数组，这实际上是相当昂贵的。

另一个数组类型，`Data.Array.Diff` 模块中的 `DiffArray` ，尝试在少量修改时减少消耗，但是在编写这本书的时候，它在实际运用中还是太慢了。

Tip

不要放弃希望

在 Haskell 中实际上高效的修改一个数组是可能的，即通过 `ST` 单子。我们将在第 26 章对其进行学习。

对 EAN-13 条形码进行编码

尽管我们的目标是*解码*条形码，不过拥有一个编码器作为引用是很有帮助的。例如通过 `decode . encode` 来检查输出是否正确，确保我们代码的正确性。

```
encodeEAN13 :: String -> String
encodeEAN13 = concat . encodeDigits . map digitToInt

-- this function computes the check digit; don't pass one in.
encodeDigits :: [Int] -> [String]
encodeDigits s@(first : rest) =
  outerGuard : lefties ++ centerGuard : righties ++ [outerGuard]
  where
    (left, right) = splitAt 5 rest
    lefties = zipWith leftEncode (parityCodes ! first) left
    righties = map rightEncode (right ++ [checkDigit s])

leftEncode :: Char -> Int -> String
leftEncode '1' = (leftOddCodes !)
leftEncode '0' = (leftEvenCodes !)

rightEncode :: Int -> String
rightEncode = (rightCodes !)

outerGuard = "101"

centerGuard = "01010"
```

字符串的编码长度是十二位数字，通过 `encodeDigits` 来添加第十三个校验位数字。

条形码由两组六位数字进行编码，由守护序列位于两组其中以及两侧。那么其它数字呢?

左侧组的每个数字通过奇数或偶数编码，奇偶的选择取决于字符串中的第一位数字。如果第一位为零，那么左侧组则是由偶数进行编码；为一，则是由奇数进行编码。这种方式的编码很优雅，它使得 EAN-13 条形码变得向后兼容旧的 UPC-A 标准。

解码器的约束

在讨论解码钱，需要为条形码可以工作的类别设定一些约束。

手机相机或者网络相机输出的是 JPEG 图像，但是编写一个 JPEG 解码器需要占用若干章节。因此我们通过处理 netpbm 文件格式简化解析问题。我们将使用第十章节所开发的解析组合子。

分治法

我们的任务是将一个有效的条形码从相机图片中提取出来。我们可以将这个大问题切分成几个有序的子问题，它们每个都是独立可追溯的。

- 将颜色数据转换成可处理的数据。
- 从图像中采样一个扫描线，并提取一组猜测数据，用以确定这一行中的编码数字可能会是什么。
- 从猜测数据中创建一个有效的解码列表。

更多的子问题将会在后续进行分割。

将彩色图像转换成易于处理的东西

条形码实际上是黑白条的序列，我们希望编写一个简单的解析器，一个简单的表示是单色图像，其中每个像素不是黑色就是白色。

解析一个彩色图像

Netbpm 彩色图像格式仅仅比第十章所提到的灰度图像格式更复杂一点。文件头的识别字符串是“P6”，剩下的头文件与灰度格式一致。图像的文件体内，每个像素都代表三个字节，它们是红，绿，蓝。

我们将图像数据表示为二维的像素数组。这里使用数组是为了更好的学习它。对于应用而言，当然可以使用列表的列表。这里使用数组的唯一优点就是：可以高效的提取一行数据。

```
type Pixel = Word8

type RGB = (Pixel, Pixel, Pixel)

type Pixmap = Array (Int, Int) RGB
```

我们提供了一些类型同义词使得类型签名更可读。

由于 Haskell 在布局数组方面给予了相当大的自由，我们必须选择一种来表示。我们无需显式的存储图像的维度，因为可以使用 `bounds` 函数来提取它们。

实际上解析器非常简短，感谢我们在第十章所开发的组合子。

```haskell
parseRawPPM :: Parse Pixmap
parseRawPPM =
  parseWhileWith w2c (/= '\n') ==> \header ->
    skipSpaces
      ==>& assert (header == "P6") "invalid raw header"
      ==>& parseNat
      ==> \width ->
        skipSpaces
          ==>& parseNat
          ==> \height ->
            skipSpaces
              ==>& parseNat
              ==> \maxValue ->
                assert (maxValue == 255) "max value out of spec"
                  ==>& parseByte
                  ==>& parseTimes (width * height) parseRGB
                  ==> \pxs ->
                    identity (listArray ((0, 0), (width - 1, height - 1)) pxs)

parseRGB :: Parse RGB
parseRGB =
  parseByte ==> \r ->
    parseByte ==> \g ->
      parseByte ==> \b ->
        identity (r, g, b)

parseTimes :: Int -> Parse a -> Parse [a]
parseTimes 0 _ = identity []
parseTimes n p = p ==> \x -> (x :) <$> parseTimes (n - 1) p
```

上述代码中仅需要注意的是 `parseTimes`，它调用其它解析器来解析一个给定的时间，再构建一个结果的列表。

灰度转化

现在我们有了彩色图像的解析，我们需要将彩色数据转换成单色的。一个中间步骤就是转换数据为灰度。将 RGB 图像转换成灰度图像有一个广泛运用的公式，即基于每个颜色通道的感知亮度。

```haskell
luminance :: (Pixel, Pixel, Pixel) -> Pixel
luminance (r, g, b) = round (r' * 0.30 + g' * 0.59 + b' * 0.11)
  where
    r' = fromIntegral r
    g' = fromIntegral g
    b' = fromIntegral b
```

Haskell 的数组是 `Functor` typeclass 的成员，所以我们可以使用 `fmap` 来讲整个图像，或是单个扫描行，从彩色转为灰度。

```haskell
type Greymap = Array (Int, Int) Pixel

pixmapToGreymap :: Pixmap -> Greymap
pixmapToGreymap = fmap luminance
```

`pixmapToGreymap` 函数仅用于解释。因为我们将仅检查一个条形码图像的某几行，没有必要做额外的工作来转换我们永远用不上的序列。

灰度至二进制以及类型安全

下一个子问题是将灰度图像转换为一个二值图像，即每个像素要么是开要么是闭。

在一个图像处理应用中，需要处理大量的数字，因此很容易为了几个不同的目的重用相同的数字类型。例如使用 `Pixel` 类型来表述 on/off 状态，使用转换将数字一来代表“on”数字零“off”。

然而像这样为了多个目的而重用类型很快会带来潜在的混乱。为了判断一个“Pixel”是否为一个数值或是一个 on/off 值，我们不再可以简单的通过观察类型签名而知。在某些上下文中，我们可以很容易使用包含“错误类型的数字”的值，并且编译器不会对其捕获，因为类型是正确的。

我们可以尝试引入类型别称来解决这个问题。同样的声明 `Pixel` 作为 `Word8` 的同义词，再使用 `Bit` 类型作为 `Pixel` 的同义词。虽然这可以提高可读性，但是类型同义词仍然无法令编译器为我们做为何有效的检查。

编译器会将 `Pixel` 与 `Bit` 视为同样的类型，所以它不能捕获，例如在期望 `Bit` 值为零或一这样的函数中使用 `Pixel` 或是 253 值，这样的异常。

如果我们自定义单色类型，编译器则会阻止我们不小心混合了其它的类型。

```haskell
data Bit = Zero | One deriving (Eq, Show)

threshold :: (Ix k, Integral a) => Double -> Array k a -> Array k Bit
threshold n a = binary <$> a
  where
    binary i
      | i < pivot = Zero
      | otherwise = One
    pivot = round $ least + (greatest - least) * n
    least = fromIntegral $ choose (<) a
    greatest = fromIntegral $ choose (>) a
    choose f = foldA1 $ \x y -> if f x y then x else y
```

我们的 `threshold` 函数计算了作为输入的数组的最大最小值。它接受这些值和一个介于 0 到 1 之间的阈值，并计算一个“枢轴”值。接着数组中的每个值，如果其值小于枢轴值，返回 `Zero` ，否则 `One` 。

我们对图像做了什么

略

寻找匹配的位数

我们的第一个问题就是找到给定位置可能被编码的数值。然后是做几个简化的假设：首先处理的是单行，其次确切知道条形码左边缘在一行中的起始位置。

运行长度编码

我们该如何解决条形码粗细的问题呢？答案是运行图像数据的长度编码。

```
type Run = Int

type RunLength a = [(Run, a)]

runLength :: (Eq a) => [a] -> RunLength a
runLength = map rle . group
  where
    rle xs = (length xs, head xs)
```

`group` 函数接受一个一系列 id 元素的列表，将它们分组成子列表。

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

`runLength` 函数将每组表示为其长度与第一个元素的二元组。

```
ghci> let bits = [0,0,1,1,0,0,1,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0]
ghci> runLength bits
[(2,0),(2,1),(2,0),(2,1),(6,0),(4,1),(4,0)]
```

由于编码的数据都是一和零，我们可以将编码值扔了而不会失去任何有用的信息，即每次运行仅保留长度。

```
runLengths :: Eq a => [a] -> [Run]
runLengths = map fst . runLength
```

```
ghci> runLengths bits
[2,2,2,2,6,4,4]
```

上面的位模式不是随机的；它们是左外守卫也是我们捕获图像中每一行的第一个编码数字。如果丢弃守卫条形码，那么留下的长度为 `[2,6,4,4]` 。

缩放运行长度并找到近似匹配

一个可行的方法是缩放运行长度，使它们的和为一。我们将使用 `Ratio Int` 类型而不是通常的 `Double` 来管理这些缩放值，因为 `Ratio` 在 **ghci** 中有更可读的打印。这使调试和开发的交互变得更简单。

```
type Score = Ratio Int

scaleToOne :: [Run] -> [Score]
scaleToOne xs = map divide xs
  where
    divide d = fromIntegral d / divisor
    divisor = fromIntegral (sum xs)

-- A more compact alternative that "knows" we're using Ratio Int:
-- scaleToOne xs = map (% sum xs) xs

type ScoreTable = [[Score]]

-- "SRL" means "scaled run length"
asSRL :: [String] -> ScoreTable
asSRL = map (scaleToOne . runLengths)

leftOddSRL = asSRL leftOddList

leftEvenSRL = asSRL leftEvenList

rightSRL = asSRL rightList

paritySRL = asSRL parityList
```

这里使用 `Score` 类型同义词可以让大多数代码不需要考虑其本身的类型是什么。

我们可以使用 `scaleToOne` 来对正在搜寻的一系列数位进行缩放。我们现在纠正了距离而导致的条形码宽度变化，因为缩放的运行长度编码表中的条目与图像中提取的运行长度序列之间应该有相等接近的匹配。

下一个问题是如何将直觉的“相当近”转为可测量的“足够近”。给定的两个缩放运行长度序列，我们可以计算它们之间的一个近似“距离”。

```
distance :: [Score] -> [Score] -> Score
distance a b = sum . map abs $ zipWith (-) a b
```

完全匹配的距离会得到零，越弱的匹配则有越大的距离。

```
ghci> let group = scaleToOne [2,6,4,4]
ghci> distance group (head leftEvenSRL)
13 % 28
ghci> distance group (head leftOddSRL)
17 % 28
```

给定一个缩放后的运行长度表，选择表中最优的几个匹配用作输入序列。

```
type Digit = Word8

bestScores :: ScoreTable -> [Run] -> [(Score, Digit)]
bestScores srl ps = take 3 . sort $ scores
  where
    scores = zip [distance d (scaleToOne ps) | d <- srl] digits
    digits = [0 .. 9]
```

列表表达式

之前的例子中提到了一个新的概念，即*列表表达式*，根据一个或多个列表来创建一个列表。

```
ghci> [(a,b) | a <- [1,2], b <- "abc"]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

在竖线左侧的表达式会在右侧*生成器表达式*的每个组合后计算。生成器表达式通过 `a <-` 以及右侧列表中的一个元素进行变量的绑定。上述例子中，生成器的组合子按照深度的顺序进行计算：首先是第一个列表的第一个元素，再计算第二个列表中的每个元素，以此类推。

另外关于生成器，我们还可以在列表表达式的右侧指定守护。一个守护就是一个 `Bool` 表达式，如果结果为 `False` ，那么元素被跳过。

```
ghci> [(a,b) | a <- [1..6], b <- [5..7], even (a + b ^ 2)]
[(1,5),(1,7),(2,6),(3,5),(3,7),(4,6),(5,5),(5,7),(6,6)]
```

我们还可以用 `let` 表达式来绑定本地变量。

```
ghci> let vowel = (`elem` "aeiou")
ghci> [x | a <- "etaoin", b <- "shrdlu", let x = [a,b], all vowel x]
["eu","au","ou","iu"]
```

如果在一个生成器表达式中模式匹配失败了，不会有异常出现。相反，列表的元素被跳过。

```
ghci> [a | (3,a) <- [(1, 'y'),(3,'e'),(5,'p')]]
"e"
```

列表表达式非常的强大已经简洁。然而它们不利于阅读。

记住匹配的奇偶

对于左侧组的每个匹配，我们需要记住其是否在偶数表格还是奇数表格。

```
data Parity a = Even a | Odd a | None a deriving (Show)

fromParity :: Parity a -> a
fromParity (Even a) = a
fromParity (Odd a) = a
fromParity (None a) = a
```

```haskell

parityMap :: (a -> b) -> Parity a -> Parity b
parityMap f (Even a) = Even (f a)
parityMap f (Odd a) = Odd (f a)
parityMap f (None a) = None (f a)

instance Functor Parity where
  fmap = parityMap
```

我们将一个编码好的值包装进 parity 内，并为其实现 `Functor` 实例，这样就可以轻松的操控 parity 内部的值了。

我们希望根据 parity 所包含的值进行排序。`Data.Function` 模块为此还提供了一个方便的组合子，`on` 。

```haskell
import Data.Function (on)

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

compareWithoutParity :: Parity (Score, Digit) -> Parity (Score, Digit) -> Ordering
compareWithoutParity = compare `on` fromParity
```

注：文件头引入了 `Data.Function` 的 `on` ，另外显式标注了 `compareWithoutParity` 的函数签名。

为了避免理解不清楚，尝试将 `on` 想象为一个带有两个参数的函数，`f` 与 `g` ，该函数返回另一个带两个参数的函数，`x` 与 `y` 。然后将 `g` 分别应用至 `x` 与 `y` ，再将 `f` 应用至上述两个结果。

将匹配包装进一个 parity 值很直接。

```haskell
bestLeft :: [Run] -> [Parity (Score, Digit)]
bestLeft ps =
  sortBy compareWithoutParity $
    map Odd (bestScores leftOddSRL ps) ++ map Even (bestScores leftEvenSRL ps)

bestRight :: [Run] -> [Parity (Score, Digit)]
bestRight = map None . bestScores rightSRL
```

有了奇偶表格的最佳左侧匹配后，可以仅基于每个匹配的质量对它们进行排序

另一种惰性，键盘上的

`Parity` 类型可以使用 Haskell 的 record 语法来避免编写 `fromParity` 函数。换言之，可以这么写：

```haskell
data AltParity a
  = AltEven {fromAltParity :: a}
  | AltOdd {fromAltParity :: a}
```

```
    | AltNone {fromAltParity :: a}
    deriving (Show)
```

那么为什么不这么做呢？答案有点令人羞愧，这与 **ghci** 中的交互式调试有关。当我们告诉 GHC 自动派生类型的 `Show` 实例时，它会根据是否使用记录语法声明该类型而生成不同的代码。

```
ghci> show $ Even 1
"Even 1"
ghci> show $ AltEven 1
"AltEven {fromAltParity = 1}"
ghci> length . show $ Even 1
6
ghci> length . show $ AltEven 1
27
```

`Show` 实例对于使用 record 语义而言显然更加冗长。

列表切分

处理列表的通常做法就是对它们进行切分。例如，条形码中的每个位数是由四个位数所编码的。我们可以将平整的列表表示为四个元素所构成的列表的列表。

```
chuckWith :: ([a] -> ([a], [a])) -> [a] -> [[a]]
chuckWith _ [] = []
chuckWith f xs = let (h, t) = f xs in h : chuckWith f t

chunksOf :: Int -> [a] -> [[a]]
chunksOf n = chuckWith $ splitAt n
```

生成候选位数列表

有了这些帮助函数后，编写一个用于生成每个数位组的候选匹配列表函数就很简单了。首先需要提前检查匹配偶数是否合理。所有的运行都需要以黑（`Zero`）条码开始，并包含足够的条码。

```
candidateDigits :: RunLength Bit -> [[Parity Digit]]
candidateDigits ((_, One) : _) = []
candidateDigits rle | length rle < 59 = []
```

对于 `bestLeft` 或 `bestRight` 返回的空列表，并不能被匹配。否则返回一个由校验编码的候选数字列表组成的列表。外部列表由十二个元素构成，条形码的每个数位有一个元素。每个子列表中的数字按匹配质量排序。

```
candidateDigits rle
  | any null match = []
  | otherwise = map (map (fmap snd)) match
```

```
    where
      match = map bestLeft left ++ map bestRight right
      left = chunksOf 4 . take 24 . drop 3 $ runLengths
      right = chunksOf 4 . take 24 . drop 32 $ runLengths
      runLengths = map fst rle
```

测试：

```
ghci> let input = zip (runLengths $ encodeEAN13 "978013211467") (cycle [Zero, One])
ghci> :t input
input :: [(Run, Bit)]
ghci> take 7 input
[(1,Zero),(1,One),(1,Zero),(1,One),(3,Zero),(1,One),(2,Zero)]
ghci> mapM_ print $ candidateDigits input
[Odd 7,Even 1,Even 2,Odd 3,Even 4,Odd 8]
[Even 8,Odd 0,Odd 1,Odd 2,Even 6,Even 7]
[Even 0,Even 1,Odd 2,Odd 4,Odd 6,Even 9]
[Odd 1,Odd 0,Even 1,Odd 2,Even 2,Even 4]
[Even 3,Odd 4,Odd 5,Even 7,Even 0,Odd 1]
[Odd 1,Even 1,Odd 2,Even 2,Odd 0,Even 0]
[None 1,None 0,None 2]
[None 1,None 0,None 2]
[None 4,None 2,None 5]
[None 6,None 8,None 2]
[None 7,None 3,None 8]
[None 7,None 3,None 8]
```

注：原文缺失：`let input = ...` 。

没有数组或哈希表的世界

在命令式语言中，数组就跟 Haskell 的列表和元组那样，作为核心。在命令式语言中数组通常是可变的。

而 Haskell 中的数组是不可变的。这就意味着“修改”数组中的一个元素，整个数组会被拷贝，同时将那个元素设为新值。很明显，这样性能很差。

可变数组是由另一个常用的命令式数据结构所构建的，即哈希表。经典的实现中，数组像是在“旋转”表，每个元素包含了一个列表。为哈希表添加一个元素时，将哈希元素来找到列表的偏移量，并在该处修改列表并添加元素。

如果数组不是可变的，更新一个哈希表则需要创建一个新的。拷贝数组，将一个新的列表放在偏移处。我们无需拷贝其它偏移处的列表，但是我们已经拷贝了整个哈希表导致了更差的性能。

不可变数组从我们的工具箱中消除了两个规范的命令式数据结构。相较于很多其它语言，在纯 Haskell 中数组显得没有这么有用。不过很多数组代码只会在构建时更新，然后以只读的方式使用它。

树作为解决方案

尽管情况看上去好像并没有那么不好。数组和哈希表通常作为由键作为索引的集合，而在 Haskell 中我们则使用树。

在 Haskell 实现一个简单的树类型非常的容易。除此之外，更有用的树类型也非常容易实现。自平衡结构，如红黑树，其平衡算法是出奇的难以被正确的使用。

Haskell 的代数数据类型组合，模式匹配，以及守护，将最复杂的平衡操作简化成了几行代码。

对于函数式程序员而言树的魅力则是其廉价的修改。我们不会打破不变的规则：树是不变的。当需要修改一个树时，则创建一个新树，这一过程可以将新旧版本的树中绝大多数的结构进行共享。例如一个树包含了 10,000 个节点，那么在添加或删除一个元素时，可以预期新旧版本共享了 9,985 个元素。换言之，每次更新需要改变元素的数量是取决于树的高度，或者树大小的对数。

Haskell 标准库提供了两种由平衡树所实现的集合类型：`Data.Map` 用于键值对，`Data.Set` 用于值集合。

Tip

关于性能

相较于哈希表，一个实现良好的存函数树数据结构将具有竞争力。因此不应该假设代码在处理树时会损失性能。

maps 的简介

`Data.Map` 模块提供了一个参数化类型 `Map k a` 。尽管其内部是一个大小平衡的二元树，其实现并不对我们可见。

`Map` 的键是严格的 strict，但是值确实非严格的 non-strict。换言之，map 的结构总是维持更新状态，但是值只会在使用时才会被计算。

记住这点很重要，因为 `Map` 在于值的惰性是导致空间泄漏的常见来源。

由于 `Data.Map` 模块包含了一系列与 Prelude 重名的名称，通常需要通过 qualified 的方式进行引入。

类型约束

`Map` 类型再起键类型上并没有任何显式的约束，但是模块中大多数有用的函数都需要键是 `Ord` 的实例。这是值得注意的，因为这是 Haskell 代码中常见的设计模式：类型约束只会在真实需要时才进行声明。

`Map` 类型和模块中的任何函数都不会约束用作值的类型。

部分应用程序的笨拙

由于某些原因，`Data.Map` 中的函数类型签名对部分应用 partial application 而言并不友好。map 的参数总是最后出现，毕竟先出现对于部分应用会更为友好。因此，使用部分应用的映射函数的代码几乎总是包含适配器函数来调整参数顺序。

从 API 开始

`Data.Map` 模块有一个巨大的“表层面”：即导出了大量的函数。

使用 `empty` 可以创建一个空的 map。使用 `singleton` 可以创建只包含一对键值的 map。

```
ghci> import qualified Data.Map as M
ghci> M.empty
fromList []
ghci> M.singleton "foo" True
fromList [("foo",True)]
```

由于实现是抽象的，我们无法对 `Map` 值进行模式匹配。相反，它提供了一系列的查询函数，其中有两个是被广泛使用的。`lookup` 函数的函数签名会稍微比较严格，不过无需担心，我们会在第十四章中进行详细的说明。

```
ghci> :t M.lookup
M.lookup :: Ord k => k -> M.Map k a -> Maybe a
```

通常而言，类型签名 `m` 会是 `Maybe` 。换言之，如果 map 包含给定键的值，那么 `lookup` 则会返回被 `Just` 所包含的值；否则返回 `Nothing` 。

```
ghci> let m = M.singleton "foo" 1 :: M.Map String Int
ghci> case M.lookup "bar" m of {Just v -> "yay"; Nothing -> "boo"}
"boo"
```

`findWithDefault` 函数接受一个值，在键不存在时作为返回。

Warning

小心偏函数！

存在一个 `(!)` 操作符，它执行查找并返回与键相关的未修饰值（即不包装在 `Maybe` 或其它）。不幸的是，它并不是一个完整的函数：它在键不存在时调用 `error` 。

要添加一对键值至 map，最有用的函数就是 `insert` 与 `insertWith` 了。前者插入一个值，且覆盖可能存在的值。**注**：原文中的 `insertWith ‘` 已废弃，现直接使用 `insertWith` 即可。

```
ghci> :t M.insert
M.insert :: Ord k => k -> a -> M.Map k a -> M.Map k a
```

```
ghci> M.insert "quux" 10 m
fromList [("foo",1),("quux",10)]
ghci> M.insert "foo" 9999 m
fromList [("foo",9999)]
```

`insertWith` 函数接受一个*合并函数 combining function* 作为其参数。如果没有匹配的键，则直接插入该值；否则，合并函数被调用用于新旧值，并将函数返回插入 map 中。

```
ghci> :t M.insertWith
M.insertWith
  :: Ord k => (a -> a -> a) -> k -> a -> M.Map k a -> M.Map k a
ghci> M.insertWith (+) "zippity" 10 m
fromList [("foo",1),("zippity",10)]
ghci> M.insertWith (+) "foo" 9999 m
fromList [("foo",10000)]
```

`delete` 函数删除 map 中给定的键值。如果键不存在，那么 map 不变。

```
ghci> :t M.delete
M.delete :: Ord k => k -> M.Map k a -> M.Map k a
ghci> M.delete "foo" m
fromList []
```

最后还有若干高效的函数类似集合类型的操作用于 map。例如下面的 `union` ，该函数是“偏向左值的”：如果两个 map 包含同样的键，那么返回的结果中则是包含左 map 的值。

```
ghci> m `M.union` M.singleton "quux" 1
fromList [("foo",1),("quux",1)]
ghci> m `M.union` M.singleton "foo" 0
fromList [("foo",1)]
```

以上我们仅覆盖大约百分之十的 `Data.Map` 的 API。我们将会在下一章进一步讲解 map 的更多细节。

将位数变为答案

现在还需要解决另有一个问题。有了很多条形码最后 12 位数字的候选码。此外还需要使用前六位数的奇偶来计算第一位数，最后需要确保答案符合校验码。

这看起来很有挑战性！我们仍有大量的未确定数据；那么我们该怎么做呢？通过 **ghci** 可以知道有多少组合需要测试。

```
ghci> product . map length . candidateDigits $ input
34012224
```

注：`input` 在上文有交代过。

这个想法到此为止。同样，我们将先关注子问题，再去考虑后续。

并行的方式解决校验位数

校验条形码的数组可以假设十分之一的数。对于一个给定的奇偶数，什么样的输入序列会导致该位数被计算?

```
type Map a = M.Map Digit [a]
```

该 map 的键是一个校验数，值则是该校验书所计算的序列值。另外两个基于该 map 定义的 map:

```
type DigitMap = Map Digit

type ParityMap = Map (Parity Digit)
```

我们通常将它们称为“解映射”，因为它们展示了“解出”每个校验数的数字序列。

```
updateMap ::
  Parity Digit -> -- new digit
  Digit -> -- existing key
  [Parity Digit] -> -- existing digit sequence
  ParityMap -> -- map to update
  ParityMap
updateMap digit key seq = insertMap key (fromParity digit) (digit : seq)

insertMap :: Digit -> Digit -> [a] -> Map a -> Map a
insertMap key digit val m = val `seq` M.insert key' val m
  where
    key' = (key + digit) `mod` 10
```

通过从 map 中得到的现有校验数，求解该校验数的序列和一个新的输入数字，该函数用指向新校验数的新序列更新 map。

这可能看起来难以理解，不过使用一个例子就很清楚了。假设需要找的校验数是 `4` ，其序列为 `[1,3]` ，同时需要添加至该 map 的数是 `8` 。那么 `4` 与 `8` 之和，取模 `10` ，即为 `2` ，也是需要插入至 map 的键。那么新校验数 `2` 所携带的序列就是 `[8,1,3]` ，即所需插入至 map 的值。

对于一个序列中的每个位数，将使用位数与旧的解映射来生成新的解映射。

```
useDigit :: ParityMap -> ParityMap -> Parity Digit -> ParityMap
useDigit old new digit = new `M.union` M.foldrWithKey (updateMap digit) M.empty old
```

注：原文中的 `M.foldWithKey` 已废弃，先用 `M.foldrWithKey` 替代。

通过一些例子来测试一下:

```
ghci> let single n = M.singleton n [Even n] :: ParityMap
ghci> useDigit (single 1) M.empty (Even 1)
fromList [(2,[Even 1,Even 1])]
ghci> useDigit (single 1) (single 2) (Even 2)
fromList [(2,[Even 2]),(3,[Even 2,Even 1])]
```

我们提供给 `useDigits` 新的解映射为空，通过 `useDigits` fold 输入序列来进行求解。

```
incorporateDigits :: ParityMap -> [Parity Digit] -> ParityMap
incorporateDigits old = foldl' (useDigit old) M.empty
```

从旧的解映射生成一个新的：

```
ghci> incorporateDigits (M.singleton 0 []) [Even 1, Even 5]
fromList [(1,[Even 1]),(5,[Even 5])]
```

最后是构建一个完整的解映射。从空 map 开始，fold 每个位数的位置。在每个位置，创建一个新的 map。该 map 在下一轮的 fold 中又变成了旧的 map。

```
finalDigits :: [[Parity Digit]] -> ParityMap
finalDigits = foldl' incorporateDigits (M.singleton 0 []) . mapEveryOther (map $ fmap (* 3))
```

那么调用 `finalDigits` 所需的列表有多长呢？我们暂且不知道序列的第一个位数，显然还不能提供它。同时我们并不想在校验位数是包含猜测。因此该列表必须有十一位元素那么长。

一旦从 `finalDigits` 返回答案，解映射又是不完整的，因为我们仍未知第一位数是什么。

通过第一位数完成求解 map

我们仍未讨论如何从左侧组的奇偶性中提取第一个位数。实际上只需要复用之前写过的代码即可。

```
firstDigit :: [Parity a] -> Digit
firstDigit =
  snd
    . head
    . bestScores paritySRL
    . runLengths
    . map parityBit
    . take 6
  where
    parityBit (Even _) = Zero
    parityBit (Odd _) = One
```

奇偶解映射中的每个元素现在包含了一个倒序的数值列表以及奇偶性数据。下一个任务就是创建一个完整的解映射，通过计算每个序列的第一个位数，再使用它创建最后一个解映射。

```
addFirstDigit :: ParityMap -> DigitMap
addFirstDigit = M.foldrWithKey updateFirst M.empty

updateFirst :: Digit -> [Parity Digit] -> DigitMap -> DigitMap
updateFirst key seq = insertMap key digit $ digit : renormalize qes
  where
    renormalize = mapEveryOther (`div` 3) . map fromParity
    digit = firstDigit qes
    qes = reverse seq
```

注：与前文相同，这里使用的是 `M.foldrWithKey` 。

通过这样的方式，去除了 `Parity` 类型，同时恢复了之前乘上的 `3` 。最后一步就是完成校验码的计算。

```
buildMap :: [[Parity Digit]] -> DigitMap
buildMap =
  M.mapKeys (10 -)
    . addFirstDigit
    . finalDigits
```

寻找正确的序列

现在有了所有可能的校验和以及对应序列的 map。剩下的就是猜测校验位，看看是否拥有与之对应的解映射。

```
import Data.Maybe (mapMaybe)

...

solve :: [[Parity Digit]] -> [[Digit]]
solve [] = []
solve xs = mapMaybe (addCheckDigit m) checkDigits
  where
    checkDigits = map fromParity $ last xs
    m = buildMap $ init xs
    addCheckDigit m k = (++ [k]) <$> M.lookup k m
```

注：原文的 `catMaybes $ map (addCheckDigit m) checkDigits` 被 `mapMaybe` 简化。

测试：

```
ghci> listToMaybe . solve . candidateDigits $ input
Just [7,2,0,0,1,3,1,1,1,4,6,7,3]
```

处理行数据

反复的提到从图像中取出一行：

```
withRow :: Int -> Pixmap -> (RunLength Bit -> a) -> a
withRow n greymap f = f . runLength . elems $ posterized
  where
    posterized = threshold 0.4 . fmap luminance . row n $ greymap
```

`withRow` 函数接受一行，转换成黑白图像，接着调用另一个函数在 run length 编码的行数据上。通过调用 `row` ，来获取行数据：

```
row :: (Ix a, Ix b) => b -> Array (a, b) c -> Array a c
row j a = ixmap (l, u) project a
```

```
    where
      project i = (i, j)
      ((l, _), (u, _)) = bounds a
```

该函数需要详细讲解一下。`fmap` 函数用于转换数组中的*值*，而 `ixmap` 则用于转换数组中的*索引*。这是一个极为强大的函数，可以让我们随心所欲的堆数组进行“切片”。

`ixmap` 的第一个参数是新数组的边界。该边界可以是与源数组不同维度的。`row` 函数则是从一个二维数组中提取一个意为数组。

第二个参数则是*投影*函数，它从新数组中获取一个索引，并返回源数组的索引。该投影索引处的值将成为新数组中位于原始索引处的值。例如，传入 `2` 至投影函数，其返回 `(2,2)`，在新数组的索引 `2` 处的元素将被源数组的 `(2,2)` 处的元素所替代。

合并所有

除非是在条形码序列一开始就调用 `candidateDigits` 函数，否则返回一个空结果。我们可以轻松的扫描整个行，直到得到一个匹配的结果。

```
import Data.List (find)

...

findMatch :: [(Run, Bit)] -> Maybe [[Digit]]
findMatch =
  find (not . null)
    . map (solve . candidateDigits)
    . tails
```

注：原文的 `listToMaybe . filter` 可以被 `find` 简化。

这里利用了惰性求值。对 `tails` 调用 `map` 只会在其结果为非空列表时才会被计算。

接下来从图像中选择一行，并尝试从中找到条形码。

```
findEAN13 :: Pixmap -> Maybe [Digit]
findEAN13 pixmap = withRow center pixmap $ fmap head . findMatch
  where
    (_, (maxX, _)) = bounds pixmap
    center = (maxX + 1) `div` 2
```

最后是一个简单的封装，用于从命令行中获取 netpbm 图像文件，并打印条形码

```
main :: IO ()
main = do
  args <- getArgs
  forM_ args $ \arg -> do
    e <- parse parseRawPPM <$> L.readFile arg
    case e of
      Left err -> print $ "error: " ++ err
      Right pixmap -> print $ findEAN13 pixmap
```

注意，本章我们已经定义了超过三十个函数，而只有 `main` 是唯一需要 `IO` 的。

一些开发风格的评论

略

13 Data structures

关联列表

通常我们需要处理由键索引的无序数据。例如 Unix 管理员会有一系列的数值 UIDs 以及其关联的用户名。该列表的价值在于能够查找给定 UID 的用户名，而不是按照数据的顺序。换言之，UID 是数据库的键。

在 Haskell 中，有若干方法可以处理类似这样数据的结构。两个最常见的就是关联列表以及由 `Data.Map` 模块提供的 `Map` 类型。关联列表很方便，因为它们很简单。它们就是 Haskell 的列表，因此所有熟悉的列表函数同样可用于关联列表。然而对于较大的数据集，`Map` 则比关联列表具有更大的性能优势。

关联列表就是一个包含了（键，值）元组的普通列表。对于 UID 映射至用户名的类型就是 `[(Integer, String)]` 。

Haskell 拥有一个称为 `Data.List.lookup` 的内建函数用于关联列表的查询。其类型为 `Eq a => a -> [(a, b)] -> Maybe b` 。

```
ghci> let al = [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
ghci> lookup 1 al
Just "one"
ghci> lookup 5 al
Nothing
```

`lookup` 函数实际很简单，我们可以自己编写一个：

```
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
```

让我们来看一个更复杂的关联列表。在 Unix/Linux 机器中存在一个名为 `/etc/passwd` 的文件用于存储用户名，UIDs，home 路径，已经其它数据。我们将编写一个程序用于解析该文件，创建一个关联列表，令用户通过给定的 UID 来查找用户名。

```
import Control.Monad (when)
import Data.List
import System.Environment (getArgs)
import System.Exit
import System.IO

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 $ head args

    -- 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"

    putStrLn "whatever"

-- 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 . filter (('#' /=) . head) . 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), head fields)

-- Takes a delimiter and a list. Break up the list based on the delimiter.
split :: (Eq a) => a -> [a] -> [[a]]
-- If the input is empty, the result is a list of empty lists.
split _ [] = [[]]
split delim str =
  -- 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".
  let (before, remainder) = span (/= delim) str
   in before : case remainder of
        [] -> []
        -- If there is more data to precess,
        -- call split recursively to process it
        x ->
          split delim $ tail x
```

注：原文的 `findByUID` 函数中的 `let al = map parseLine . lines $ content` 需要修改成 `let al = map parseLine . filter (('##' /=) . head) . lines $ content` 才能对 `/etc/passwd` 文件生效。否则会抛出异常 `Prelude.!!: index too large` 。

测试：

```
% runhaskell passwd-al.hs /etc/passwd 0
root
% runhaskell passwd-al.hs /etc/passwd 3
Could not find that UID
```

映射

`Data.Map` 模块提供了一个与关联列表行为相似的 `Map` 类型，不过有更优异的性能。

映射提供了其它语言中与哈希表一样功能。其内部的实现为一个平衡二叉树。相较于哈希表，在不可变数据上的表现而言，映射更加的高效。这也是纯函数式编程如何深刻影响我们编写代码的最明显的例子：我们选择的数据结构和算法可以清晰的表达并且高效的执行，但是我们对特定任务的选择通常与命令式语言中的对应选项不同。

`Data.Map` 中有些函数与 Prelude 同名，因此需要 `import qualified Data.Map as Map` 并使用 `Map.name` 来引用模块中对应的名称。

```haskell
import Data.Map qualified 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
```

类似 `Map.insert` 的函数通常在 Haskell 中这么工作：它们返回输入数据的拷贝，并应用所需的修改。这对于映射而言很轻松。这意味着可以使用 `foldl` 来构建一个如 `mapFold` 案

例中那样的映射。或者是调用 `Map.insert` 串联在一起如 `mapManual` 案例中那样。使用 **ghci** 来查看是否如同预期：

```
ghci> :l buildmap.hs
[1 of 2] Compiling Main             ( buildmap.hs, interpreted )
Ok, one module loaded.
ghci> mapFromAL
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
ghci> mapFold
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
ghci> mapManual
fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
```

注意 `mapManual` 的输出与之前使用列表来构建映射的输出是不一样的。映射并不确保原始的顺序。

函数也是数据

Haskell 一部分的能力就是创建于操作函数非常的容易。下面是一个将函数存储为 record 字段的例子：

```
data CustomColor = CustomColor {red :: Int, green :: Int, blue :: Int}
  deriving (Eq, Show, Read)

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 = const (purple, 0)}
```

注意 `colorCalc` 的类型：它是一个函数，接受一个 `Int` 并返回 `(CustomColor, Int)` 的元组。我们创建了两个 `FuncRec` 的 records：`plus5` 与 `always0` 。注意两者的 `colorCalc` 总是返回紫色。`FuncRec` 其本身并没有字段用于存储颜色，而是以某种方式使值成为了函数本身。这就是**闭包** *closure*。

```
ghci> :l funcrecs.hs
[1 of 2] Compiling Main             ( funcrecs.hs, interpreted )
Ok, one module loaded.
ghci> :t plus5
plus5 :: FuncRec
ghci> name plus5
"plus5"
ghci> :t colorCalc plus5
colorCalc plus5 :: Int -> (CustomColor, Int)
```

```
ghci> (colorCalc plus5) 7
(CustomColor {red = 255, green = 0, blue = 255},12)
ghci> :t colorCalc always0
colorCalc always0 :: Int -> (CustomColor, Int)
ghci> (colorCalc always0) 7
(CustomColor {red = 255, green = 0, blue = 255},0)
```

更高级的方式，例如将数据用于若干地方，使用类型构造函数则会大有帮助：

```
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" (const 0)
```

这里有一个名为 `mkFuncRec` 的函数，其接受一个 `String` 以及另一个函数作为参数，返回一个新的 `FuncRec` record。注意 `mkFuncRec` 的两个参数在若干地方都被用到了。

```
ghci> :l funcrecs2.hs
[1 of 2] Compiling Main             ( funcrecs2.hs, interpreted )
Ok, one module loaded.
ghci> :t plus5
plus5 :: FuncRec
ghci> name plus5
"plus5"
ghci> (calc plus5) 5
10
ghci> (namedCalc plus5) 5
("plus5",10)
ghci> let plus5a = plus5 {name = "PLUS5A"}
ghci> name plus5a
"PLUS5A"
ghci> (namedCalc plus5a) 5
("plus5",10)
```

注意 `plus5a` 的创建。我们修改了 `name` 字段而不是 `namedCalc` 字段。这是为什么 `name` 拥有一个新的名称，而 `namedCalc` 仍然返回的是传入 `mkFuncRec` 的名称；它不会改变，除非我们显式的修改它。

拓展案列：/etc/passwd

以下案例从 `/etc/passwd` 文件中解析并存储数据。

```haskell
import Control.Monad (when)
import Data.List
import Data.Map qualified as Map
import System.Environment (getArgs)
import System.Exit
import System.IO
import Text.Printf (printf)

{-
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] ->
```

```
46          -- Generate a 'PasswdEntry' the shorthand way:
47          -- using the positional fields. We use 'read' to
48          -- convert the numeric fields to Integers.
49          [(PasswdEntry f1 f2 (read f3) (read f4) f5 f6 f7, [])]
50        x -> error $ "Invalid number of fields in input: " ++ show x
51      where
52        -- Takes a delimiter and a list.
53        -- Break up the list based on the delimiter.
54        split :: (Eq a) => a -> [a] -> [[a]]
55        -- If the input is empty, the result is a list of empty lists.
56        split _ [] = []
57        split delim str =
58          -- Find the part of the list before delim and put it in "before".
59          -- The rest of the list, including the leading delim,
60          -- goes in "remainder".
61          let (before, remainder) = span (/= delim) str
62           in before : case remainder of
63                []  -> []
64                x ->
65                  -- If there is more data to process,
66                  -- call split recursively to process it
67                  split delim (tail x)
68
69  -- Convenience aliases; we'll have two maps: one from UID to entries
70  -- and the other from username to entries
71  type UIDMap = Map.Map Integer PasswdEntry
72
73  type UserMap = Map.Map String PasswdEntry
74
75  {-
76  Converts input data to maps. Returns UID and User maps.
77  -}
78  inputToMaps :: String -> (UIDMap, UserMap)
79  inputToMaps inp = (uidmap, usermap)
80    where
81      -- fromList converts a [(key, value)] list into a Map
82      uidmap = Map.fromList . map (\pe -> (uid pe, pe)) $ entries
83      usermap = Map.fromList . map (\pe -> (userName pe, pe)) $ entries
84      -- Convert the input String to [PasswdEntry]
85      entries = map read $ lines inp
86
87  mainMenu maps@(uidmap, usermap) = do
88    putStr optionText
89    hFlush stdout
90    sel <- getLine
91    -- See what they want to do. For every option except 4,
92    -- return them to the main menu afterwards by calling
93    -- mainMenu recursively
94    case sel of
```

```
95      "1" -> lookupUserName >> mainMenu maps
96      "2" -> lookupUID >> mainMenu maps
97      "3" -> displayFile >> mainMenu maps
98      "4" -> return ()
99      _ -> putStrLn "Invalid selection" >> mainMenu maps
100   where
101     lookupUserName = do
102       putStrLn "Username: "
103       username <- getLine
104       case Map.lookup username usermap of
105         Nothing -> putStrLn "Not found."
106         Just x -> print x
107     lookupUID = do
108       putStrLn "UID: "
109       uidstring <- getLine
110       case Map.lookup (read uidstring) uidmap of
111         Nothing -> putStrLn "Not found."
112         Just x -> print x
113     displayFile =
114       putStr . unlines . map (show . snd) . Map.toList $ uidmap
115     optionText =
116       "\npasswdmap options:\n\
117       \\n\
118       \1    Look up a user name\n\
119       \2    Look up a UID\n\
120       \3    Display entire file\n\
121       \4    Quit\n\n\
122       \Your selection: "
123
124 main = do
125   -- Load the command-line arguments
126   args <- getArgs
127
128   -- If we don't have the right number of args,
129   -- give an error and abort
130
131   when (length args /= 1) $ do
132     putStrLn "Syntax: passwdmap filename"
133     exitFailure
134
135   -- Read the file lazily
136   content <- readFile $ head args
137   let maps = inputToMaps content
138   mainMenu maps
```

该案例维护了两个 maps：username 与 `PasswdEntry` ，以及 UID 与 `PasswdEntry` 。可以将此视为数据库的两个不同的数据索引，用于加速搜索不同的字段。

拓展案列：数值类型

略

第一步

如果希望为加号操作符实现一些自定义行为，则必须定义一个 newtype，并使其成为 Num 的一个实例。该类型需要以符号方式存储表达式。为此我们需要存储操作本身，其左侧以及右侧。其中，左右侧也可以是表达式。

```haskell
-- The "operators" that we're going to support
data Op = Plus | Minus | Mul | Div | Pow
  deriving (Eq, Show)

{- The core symbolic manipulation type -}
data SymbolicManip a
  = Number a -- Simple number, such as 5
  | Arith Op (SymbolicManip a) (SymbolicManip a)
  deriving (Eq, Show)

{-
  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 = Arith Plus a b
  a - b = Arith Minus a b
  a * b = Arith Mul a b
  negate = Arith Mul (Number (-1))
  abs a = error "abs is unimplemented"
  signum _ = error "signum is unimplemented"
  fromInteger i = Number (fromInteger i)
```

这里定义了名为 `Op` 的类型。该类型代表了某些的操作符。接着是 `SymbolicManip a` 的定义，由于有 `Num a` 约束，任何 `Num` 都可用作于 `a` 。一个完整类型会像 `SymbolicManip Int` 这样。

一个 `SymbolicManip` 类型可以是一个简单的数字，或者是某些计算操作。`Arith` 构造函数的类型是递归的，这在 Haskell 中很合理。`Arith` 用一个 `Op` 和另外两个 `SymbolicManip` 创建了一个新的 `SymbolicManip` 。

```
ghci> :l numsimple.hs
[1 of 2] Compiling Main             ( numsimple.hs, interpreted )
Ok, one module loaded.
ghci> Number 5
Number 5
ghci> :t Number 5
```

```
Number 5 :: Num a => SymbolicManip a
ghci> :t Number (5::Int)
Number (5::Int) :: SymbolicManip Int
ghci> Number 5 * Number 10
Arith Mul (Number 5) (Number 10)
ghci> (5 * 10)::SymbolicManip Int
Arith Mul (Number 5) (Number 10)
ghci> (5 * 10 + 2)::SymbolicManip Int
Arith Plus (Arith Mul (Number 5) (Number 10)) (Number 2)
```

可以看到非常基础的表达式是如何工作的。注意 Haskell 是如何“转换”`5 * 10 + 2` 成一个 `SymbolicManip` 的，以及其中的处理顺序。这并非一个真实的转换；`SymbolicManip` 如今是一个 first-class 数值。整数数值的字面量会被视为包裹在 `fromInteger` 内，因此 `5` 等同于 `SymbolicManip Int` 等同于 `Int` 。

至此我们的任务就简单了：拓展 `SymbolicManip` 类型来表达所有需求的操作，为其实现其他数值 typeclasses 的实例，并实现自身的 `Show` 。

完整代码

以下是完整的 `num.hs` 代码：

```
import Data.List

-- Symbolic/units manipulation
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 SymbolicManips a
  = Number a -- Simple number, such as 5
  | Symbol String -- A symbol, such as x
  | BinaryArith Op (SymbolicManips a) (SymbolicManips a)
  | UnaryArith String (SymbolicManips a)
  deriving (Eq)
```

上述代码定义了之前使用过的 `Op` ，以及 `SymbolicManips` ，与之前类似。这个版本支持了单元算法符（即仅接受一个参数）例如 `abs` 以及 `cos` 。接下来定义 `Num` 。

```
{-
  SymbolicManips will be an instance of Num.
  Define how the Num operations are handled over a SymbolicManips.
  This will implement things like (+) for SymbolicManips.
-}
instance (Num a) => Num (SymbolicManips a) where
  a + b = BinaryArith Plus a b
  a - b = BinaryArith Minus a b
  a * b = BinaryArith Mul a b
  negate = BinaryArith Mul (Number (-1))
  abs = UnaryArith "abs"
  signum _ = error "signum is unimplemented"
  fromInteger = Number . fromInteger
```

以上代码简单直接。注意之前的版本并不支持 `abs` ，不过现在有了 `UnaryArith` 构造函数就能支持了。接下来是更多的实例：

```
{-
  Make SymbolicManips an instance of Fractional
-}
instance (Fractional a) => Fractional (SymbolicManips a) where
  a / b = BinaryArith Div a b
  recip = BinaryArith Div $ Number 1
  fromRational = Number . fromRational

{-
  Make SymbolicManips an instance of Floating
-}
instance (Floating a) => Floating (SymbolicManips a) where
  pi = Symbol "pi"
  exp = UnaryArith "exp"
  log = UnaryArith "log"
  sqrt = UnaryArith "sqrt"
  a ** b = BinaryArith Pow a b
  sin = UnaryArith "sin"
  cos = UnaryArith "cos"
  tan = UnaryArith "tan"
  asin = UnaryArith "asin"
  acos = UnaryArith "acos"
  atan = UnaryArith "atan"
  sinh = UnaryArith "sinh"
  cosh = UnaryArith "cosh"
  tanh = UnaryArith "tanh"
  asinh = UnaryArith "asinh"
  acosh = UnaryArith "acosh"
  atanh = UnaryArith "atanh"
```

以上代码实现的是 `Fractional` 与 `Floating` 。接下来是将表达式以字符串形式展示的实现：

```
{-
  Show a SymbolicManips as a String, using conventional algebraic notation
-}
prettyShow :: (Show a, Num a) => SymbolicManips a -> String
-- Show a number or symbol as a bare number or serial
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 ++ "(" ++ prettyShow a ++ ")"

op2str :: Op -> String
op2str Plus = "+"
op2str Minus = "-"
op2str Mul = "*"
op2str Div = "/"
op2str Pow = "**"

{-
  Add parenthesis where needed. This function is fairly conservative and will
  add parenthesis when not needed in some cases.

  Haskell will have already figured out precedence for us while building up
  the SymbolicManips.
-}
simpleParen :: (Show a, Num a) => SymbolicManips a -> String
simpleParen (Number x) = prettyShow (Number x)
simpleParen (Symbol x) = prettyShow (Symbol x)
simpleParen x@(BinaryArith _ _ _) = "(" ++ prettyShow x ++ ")"
simpleParen x@(UnaryArith _ _) = prettyShow x

{-
  Showing a SymbolicManips calls the prettyShow function on it
-}
instance (Show a, Num a) => Show (SymbolicManips a) where
  show = prettyShow
```

首先是定义 `prettyShow` 函数，它用于渲染表达式。接下来实现将一个表达式转换为 RPN 格式字符串的算法：

```
{-
  Show a SymbolicManip using RPN. HP calculator users may find this familiar.
-}
```

```
rnpShow :: (Show a, Num a) => SymbolicManips a -> String
rnpShow 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 = intercalate
  in join " " (toList i)
```

接下来实现一个函数来对表达式进行一些基本的简化：

```
{-
  Perform some basic algebraic simplifications on a SymbolicManip
-}
simplify :: (Num a, Eq a) => SymbolicManips a -> SymbolicManips 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, _) -> Number 0
        (Mul, _, 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
```

现在需要为已构建好的库来增加单位测量。这将允许我们表达例如“5 米”这样的数量。首先是定义一个类型：

```
{-
  New data type: Units.
  A Units type contains a number and a SymbolicManip, which represents the units of measure.
  A simple label would be something like (Symbol "m")
-}
data Units a = Units a (SymbolicManips a) deriving (Eq)
```

一个 `Units` 包含一个数值与一个标签。标签本身是一个 `SymbolicManip` 。接下来就是 `Units` 的 `Num` 实例了：

```
{-
  Implement Units for Num.
  We don't know how to convert between arbitrary units, so we generate an error if we try to
   add
  numbers with different units. For multiplication, generate the appropriate new units.
```

```
-}
instance (Num a, Eq 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)
```

这里就能明白为什么使用 `SymbolicManip` 而不是 `String` 来存储测量单位了。例如在乘法计算出现时，测量的单位也同样发生了变化。例如 5 米乘以 2 米，那么得到的是 10 平方米。我们强制加法时的单位匹配，并由加法来实现减法。下面是更多的 typeclass：

```
{-
  Make Units an instance of Fractional
-}
instance (Fractional a, Eq 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 (Floating a, Eq 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"
  sqrt (Units xa ua) = Units (sqrt xa) (sqrt ua)
  (Units xa ua) ** (Units xb ub)
    | ub == Number 1 = Units (xa ** xb) (ua ** Number xb)
    | otherwise = error "units for RHS of ** not supported"
  logBase _ = error "logBase not yet implemented in Units"
  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 empty"
  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 empty"
  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 empty"
```

```
  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"
```

接下来是一些处理单位的公用函数:

```
{-
  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)
```

这里首先是 `units` 用于简化了表达式的构建。接着是 `dropUnits` 用于去掉单位返回裸值 `Num` 。最后是定义了之前所需的角度与弧度之间的转换函数。接下来是 `Units` 的 `Show` 实例:

```
{-
  Showing units: we show the numeric component, an underscore, then the prettyShow version of
   the simplified units
-}
instance (Show a, Num a, Eq a) => Show (Units a) where
  show (Units xa ua) = show xa ++ "_" ++ prettyShow (simplify ua)
```

最后是测试:

```
test :: (Num a) => a
test = 2 * 5 + 3
```

函数作为数据的优点

在命令式语言中，两个列表的追加廉价且容易。这里是一个简单的 C 结构，其中维护了两个列表的头尾指针：

```
struct list {
  struct node *head, *tail;
};
```

当我们想将一个列表附加到另一个列表时，需要修改前一个列表的最后一个节点并指向后一个列表的 `head` 节点，接着更新尾指针指向尾结点。

显然在 Haskell 中，如果想要保持纯粹性，这种方法是不行的。因为数据是不可变的，所以不能随意修改列表。Haskell 用来附加列表的 `(++)` 操作符：

```
(++) :: [a] -> [a] -> [a]
(x : xs) ++ ys = x : xs ++ ys
_ ++ ys = ys
```

从代码中可知，创建一个新的列表取决于初始列表的长度。

如果是不断的附加，那么这个代价是巨大的。`(++)` 操作符是右关联的：

```
ghci> :info (++)
(++) :: [a] -> [a] -> [a]   -- Defined in GHC.Base
infixr 5 ++
```

这就意味着 Haskell 将表达式 `"a" ++ "b" ++ "c"` 转换为 `"a" ++ ("b" ++ "c")` 。这样可以使性能达到最优，因为它令左侧的操作尽可能的简短。

当我们重复的对一个列表进行附加时，我们便破坏了这种结合性。

与此同时，命令式程序员却在呵呵笑，因为他们重复追加的成本只取决于他们执行追加的次数。它们的性能是线性的，而我们则是平方函数。

像这样重复对列表进行附加的常见操作会造成性能损失时，就需要从另一个角度来看待这个问题了。

表达式 `"a"++` 是一个偏应用函数。它的类型是什么?

```
ghci> :type ("a" ++)
("a" ++) :: [Char] -> [Char]
```

由于这是一个函数，我们可以使用 `(.)` 操作符与其它操作进行组合，例如 `("b"++)` 。

```
ghci> :type ("a" ++) . ("b" ++)
("a" ++) . ("b" ++) :: [Char] -> [Char]
```

新的函数拥有同样的类型。那么当停止组合函数时，并将一个 `String` 提供给函数会得到什么呢?

```
ghci> let f = ("a" ++) . ("b" ++)
ghci> f []
"ab"
```

我们附加了字符串！我们用这些偏应用函数来存储数据，通过一个空列表来获取它。每个偏应用的 `(++)` 与 `(.)` 代表着一个附加，但是又不实际 执行附加。

关于这个方法有两点非常有趣的地方。首先偏应用的成本是恒定的，因此很多偏应用的成本是线性的。其次当我们最终提供 `[]` 值来获取偏应用中的最终列表时，应用的顺序是从右至左的。这使得 `(++)` 的左侧操作最小，从而让附加操作的整体成本是线性而不是二次方程的。

将差异列表变为一个合理的库

第一步是使用 `newtype` 声明对用户隐藏基础类型。我们将创建一个名为 `DList` 的新类型。

```
newtype DList a = DL
{ unDL :: [a] -> [a]
}
```

`unDL` 函数即解构函数，即移除 `DL` 构造函数。

```
append :: DList a -> DList a -> DList a
append xs ys = DL (unDL xs . unDL ys)
```

`append` 函数看起来有点复杂，但它实际上只是之前展示的 `(.)` 操作符的相同用法。对函数进行组合，首先需要从 `DL` 构造函数中解包，因此使用了 `unDL` 。接着通过 `DL` 构造函数重新将结果包装起来。

下面是另一种写法，这里使用了模式匹配来解包 `xs` 与 `ys` ：

```
append' :: DList a -> DList a -> DList a
append' (DL xs) (DL ys) = DL (xs . ys)
```

`DList` 类型如果不能与正常列表互相转换的话，那么用处也不会很大：

```
fromList :: [a] -> DList a
fromList xs = DL (xs ++)

toList :: DList a -> [a]
toList (DL xs) = xs []
```

如果我们还想要 `DList` 变得跟普通列表一样好用，那么还需要提供一些列表的常规操作。

```
empty :: DList a
empty = DL id

```

```haskell
-- equivalent of the list type's (:) operator
cons :: a -> DList a -> DList a
cons x (DL xs) = DL $ (x :) . xs

dfoldr :: (a -> b -> b) -> b -> DList a -> b
dfoldr f z xs = foldr f z $ toList xs
```

尽管 `DList` 令列表附加的成本下降，但并不是所有的类列表操作都是可用的。`head` 函数的成本对于普通列表是常数，而 `DList` 需要将整个 `DList` 转换为正常列表才能进行，因此这使成本变得额外高昂。

```haskell
safeHead :: DList a -> Maybe a
safeHead xs = case toList xs of
  (y : _) -> Just y
  _ -> Nothing
```

对于 `map` 而言，我们需要将 `DList` 类型成为一个函子。

```haskell
dmap :: (a -> b) -> DList a -> DList b
dmap f = dfoldr go empty
  where
    go x = cons (f x)

instance Functor DList where
  fmap = dmap
```

最终可在源文件头部添加模块头进行导出：

```haskell
module DList
  ( DList,
    fromList,
    toList,
    empty,
    append,
    cons,
    dfoldr,
  )
where
```

列表，差异列表，以及幺半群

在抽象代数中，存在一个名为*幺半群* *monoid* 的简单抽象结构。很多数学对象就是幺半群，它们必须有两个属性：

- 一个二维操作符。名为 `(*)` ：表达式 `a * (b * c)` 的结果必须等同于 `(a * b) * c` 的结果。
- 一个 id 值。如果是 `e` ，它必须遵循两个规则：`a * e == a` 以及 `e * a == a` 。

具体实现：

```haskell
instance Semigroup (DList a) where
  (<>) = append

instance Monoid (DList a) where
  mempty = empty
```

注：与原文不同，`Monoid` 现在需要 `Semigroup` 作为约束。

通用序列

Haskell 内建的列表类型和我们所构建的 `DList` 类型在某些特定场景下的性能非常的差。而 `Data.Sequence` 模块中定义的 `Seq` 容器类型则是在大量的操作下具有优异的性能。

```haskell
import Data.Sequence qualified as Seq

import Data.Sequence ((<|), (><), (|>))

import Data.Foldable qualified as Foldable
```

示例：

```
ghci> :l DataSequence.hs
[1 of 2] Compiling Main             ( DataSequence.hs, interpreted )
Ok, one module loaded.
ghci> Seq.empty
fromList []
ghci> Seq.singleton 1
fromList [1]
ghci> let a = Seq.fromList [1,2,3]
ghci> Seq.singleton 1 |> 2
fromList [1,2]
ghci> let left = Seq.fromList [1,3,3]
ghci> let right = Seq.fromList [7,1]
ghci> left >< right
fromList [1,3,3,7,1]
ghci> Foldable.toList (Seq.fromList [1,2,3])
[1,2,3]
ghci> Foldable.foldl' (+) 0 (Seq.fromList [1,2,3])
6
```

14 Monads

简介

在第七章 I/O 中，我们讨论了 `IO` 单子，然而当时只局限于如何它与外界进行交互，并没有讨论什么是单子。

...

重构早期代码

Maybe 链

回忆一下第十章的 `PNM.hs` 代码中的 `parseP5` 函数：

```haskell
-- header
matchHeader :: 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

-- nat = natural number
getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
  Nothing -> Nothing
  Just (num, rest)
    | num <= 0 -> Nothing
    | otherwise -> Just (fromIntegral num, rest)

getBytes :: 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

-- parse function
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5 s =
  case matchHeader (L8.pack "PS") 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)
```

接着又使用了 `(>>?)` 函数来减少模式匹配：

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v >>? f = f v
```

...

隐式状态

简化后的代码如下：

```
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)
```

我们仍然遇到了重复的行为模式：消费一些字符串，返回一个结果，接着将剩下的字符串传入下一个函数进行消费。然而这套模式有隐藏的风险：如果希望将其余部分的信息在链中传递，那么就需要将链中的每个元素都进行修改，将每个二元元组改为三元元组！

我们解决这个问题的方法是将管理当前字符串的责任从链中的单个函数移动到将函数进行串联的函数中：

```haskell
(==>) :: Parse a -> (a -> Parse b) -> Parse b
firstParser ==> secondParser = Parse chainedParser
  where
    chainedParser initState =
      case runParse firstParser initState of
        Left errMessage ->
          Left errMessage
        Right (firstResult, newState) ->
          runParse (secondParser firstResult) newState
```

我们同样将解析状态的细节隐藏进 `ParseState` 类型中。即使 `getState` 与 `putState` 函数都不会检查解析状态，因此对 `ParseState` 的任何修改都不会影响现有的代码。

寻找共有模式

...

单子 typeclass

我们可以在 Haskell 的 typeclass 中捕获链接和注入的概念，同时又希望它们具有类型。标准的 Prelude 已经定义了这样的一个 typeclass，即 `Monad` 。

```haskell
class Monad m where
  -- chain
  (>>=)  :: m a -> (a -> m b) -> m b
  -- inject
  return :: a -> m a
```

注：新的 Haskell 版本中 Monad 还需要先实现 `Applicative` 实例，而 `Applicative` 又需要先实现 `Functor` 实例。

这里的 `>>=` 是链接函数，而 `return` 则是注入函数。

...

行话时间

- ”Monadic” 即为“与单子有关”。一个 monadic *类型*就是 `Monad` typeclass 的一个实例；一个 monadic *值*拥有一个 monadic 类型。
- 当我们说一个类型“是一个单子”，这实际上在简述它是 `Monad` typeclass 的一个实例。作为 `Monad` 的一个实例则拥有必要的 monadic 的类型构造函数，注入函数，以及链接函数。
- 同样的，引用“`Foo` 单子”意味着正在谈论名为 `Foo` 的类型，且它是 `Monad` 的一个实例。

- “动作 action”则是一个 monadic 值的别称。这个词的这种用法可能起源于用于 I/O 单子的引入，其中像 `print "foo"` 这样的单子值可能具有可观察到的副作用。具有 monadic 返回类型的函数也可以称为操作，尽管这种情况不太常见。

使用一个新的单子：展示你的成果！

在介绍单子时，我们展示了一些已经具备 monadic 形态的预定义代码。现在让我们定义其接口。

纯 Haskell 代码编写很简单，但是它不能执行 I/O。有时我们希望在记录一些决策时，不必将 log 信息写入文件中。现在来开发一个这样的库。

回忆在之前将 glob 模式转为正则表达式的小节中所开发的 `globToRegex` 函数。我们将修改它，使其保存它所翻译的每个特殊模式序列的记录。

作为开始，首先将返回类型通过 `Logger` 类型构造函数进行包装。

```
globToRegex :: String -> Logger String
```

隐藏信息

```
module Logger
  ( Logger,
    Log,
    runLogger,
    record,
  )
where
```

像这样隐藏细节有两个好处：它为我们实现单子提供了相当大的灵活性，更重要的是，它为用户提供了一个简单的界面。

我们的 `Logger` 类型存粹是一个类型构造函数。没有导出用户创建这种类型的值所需的值构造函数，用户仅仅可以使用 `Logger` 来编写类型签名。

`Log` 类型即字符串列表的同义词，将若干签名变得更有可读性。使用列表字符串是为了实现起来更加的方便。

```
type Log = [String]
```

相较于提供用户一个值构造函数，更好的是提供一个 `runLogger` 函数，用于记录操作。它将同时返回操作的结果，以及计算时的日志。

```
runLogger :: Logger a -> (a, Log)
```

控制退出

`Monad` typeclass 不提供任何方法让值脱离它们的 monadic 束缚。可以使用 `return` 将值注入进一个单子中，可以使用 `(>>=)` 将值从单子中提取出来，而位于其右侧的函数将会得到解包的值，该函数再将其结果重新进行包装。

多数单子拥有一个或多个类似 `runLogger` 的函数。最典型的就是 `IO` ，通常只会在退出程序时才退出。

一个单子执行函数会将代码运行在单子内部并展开其结果。这样的函数通常是提供给值从其一元包装器中转义的唯一方法。因此单子的作者可以完全控制单子内发生的任何事物。

留痕

当在一个 `Logger` 中操作时，用户代码调用 `record` 来记录事物。

```
record :: String -> Logger ()
```

由于记录发生在单子的管道中，所以操作的结果不提供任何信息。

通常来说，一个单子会提供一个或多个像 `record` 一样的帮助函数。它们是访问该单子的特殊行为的方法。

模块中也为 `Logger` 类型提供了 `Monad` 实例。它们是客户端模块为了能使用这个单子所需的全部定义。

使用 Logger 单子

下面是将 glob-to-regexp 在 `Logger` 单子中转换的方法：

```
globToRegex :: String -> Logger String
globToRegex cs =
  globToRegex' cs >>= \ds -> return $ '^' : ds
```

注意 `(>>=)` 的类型：它从左侧的 `Logger` 包裹中中提取值，然后将解包的值传递给右侧的函数。在其右侧的函数则需要使用 `Logger` 包装其结果。这也正是 `return` 所做的：获取一个纯值，包裹该值进入一个单子类型的构造函数中。

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type (globToRegex "" >>=)
(globToRegex "" >>=) :: (String -> Logger b) -> Logger b
```

即使编写一个不做任何事情的函数也要调用 `return` 将结果包装成正确的类型。

```
globToRegex' :: String -> Logger String
globToRegex' "" = return "$"
```

当调用 `record` 来存储一个日志，需要用 `(>>)` 而不是 `(>>=)` 来进行串联。

```
globToRegex' ('?' : cs) =
record "any"
  >> globToRegex' cs
  >>= \ds -> return ('.' : ds)
```

记住它是 `(>>=)` 的一种变体，它会忽略左侧的结果。我们知道 `record` 的结果总是 `()`，因此无需捕获该结果。

我们可以使用 `do` 符号来整洁一下代码：

```
globToRegex'' ('*' : cs) = do
  record "kleene star"
  ds <- globToRegex'' cs
  return $ ".*" ++ ds
```

解析字符类主要遵循以上这种模式：

```
globToRegex' ('[' : '!' : c : cs) =
  record "character class, negative"
    >> charClass cs
    >>= \ds -> return $ "[^" ++ c : ds
globToRegex' ('[' : c : cs) =
  record "character class"
    >> charClass cs
    >>= \ds -> return $ "[" ++ c : ds
globToRegex' ('[' : _) =
  fail "unterminated character class"
```

混合纯代码与单子代码

根据目前所见到的代码，单子看起来有一个很大的缺点：使用普通的纯函数处理被包装后的值时，会变得很棘手。以下是这个浅显问题的一个简单说明。假设有一个简单的代码运行在 `Logger` 单子中并返回一个字符串。

```
ghci> let m = return "foo" :: Logger String
```

如果希望得知该字符串的长度，我们无法直接调用 `length`：因为字符串被封装了，类型并不匹配。

迄今为止我们所能做的就是类似这样：

```
ghci> :type   m >>= \s -> return (length s)
m >>= \s -> return (length s) :: Logger Int
```

通过 `(>>=)` 解包该字符串，接着用一个小的匿名函数来调用 `length` 并通过 `return` 重新进行包装。

由于 `Monad` typeclass 已经提供了 `(>>=)` 与 `return` 函数用于解包与打包一个值，`liftM` 函数则不需要知道任何单子实现的细节。

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= \i -> return $ f i
```

当我们为一个类型声明 `Functor` typeclass 实例时，我们需要编写自己的 `fmap` 。相反的，`liftM` 不需要了解单子的内部，因为它们抽象出来了 `(>>=)` 与 `return` 。我们只需要通过适当的类型约束编写它一次。

`liftM` 函数预定义在 `Control.Monad` 模块中。

为了检测 `liftM` 是如何提高可读性的，我们来比较连个相同作用的代码。首先是不使用 `liftM` 的情况：

```
charClass_wordy (']' : cs) =
  globToRegex' cs >>= \ds -> return $ ']' : ds
charClass_wordy (c : cs) =
  charClass_wordy cs >>= \ds -> return $ c : ds
```

然后是使用 `liftM` 来去除 `(>>=)` 以及匿名函数：

```
charClass (']' : cs) = (']' :) `liftM` globToRegex' cs
charClass (c : cs) = (c :) `liftM` globToRegex' cs
```

与 `fmap` 一样，我们经常以中缀形式使用 `liftM` 。阅读这种表达式的一种简单方法是“将左侧的纯函数应用于右侧一元操作的结果”。

`liftM` 太有用了以致于 `Control.Monad` 定义了若干变体，用于结合更长的操作链。来看一下 `globToRegex'` 函数的最后一个分句：

```
globToRegex' (c : cs) = liftM2 (++) (escape c) (globToRegex' cs)

escape :: Char -> Logger String
escape c
  | c `elem` regexChars = record "escape" >> return ['\\', c]
  | otherwise = return [c]
  where
    regexChars = "\\+()^$.{}]|"
```

而上述的 `liftM2` 函数定义如下：

```
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 =
  m1 >>= \a ->
    m2 >>= \b ->
      return (f a b)
```

它首先执行第一个操作，接着第二个，再使用纯函数 `f` 组合它们的结果，并封装该结果。除了 `liftM2` ， `Control.Monad` 中的变体达到了 `liftM5` 。

消除一些误解

略。

构建一个 Logger 单子

`Logger` 类型的定义非常简单：

```
newtype Logger a = Logger {execLogger :: (a, Log)}
```

它一个二元元组，第一个元素是具体操作的结果，而第二个元素则是该操作中产生的所有日志的列表。

我们将该元组包装进了一个 `newtype` 中使其变为独立类型。`runLogger` 函数则是从包装中提取出该元组。而暴露出来执行一个日志行为的函数，`runLogger`，则是 `execLogger` 的一个别名。

```
runLogger = execLogger
```

`record` 帮助函数则是通过传入的信息，创建了一个单例列表：

```
record s = Logger ((), [s])
```

该操作的结果为 `()`，这也是为什么在结果位置放置它的缘故。

现在开始 `Monad` 实例的 `return`，它很简单：不记录日志，并将其入参存储值结果位置。

```
instance Functor Logger where
  fmap f (Logger (a, l)) = Logger (f a, l)

instance Applicative Logger where
  pure a = Logger (a, [])
  (<*>) (Logger (h, w)) (Logger (a, x)) = Logger (h a, w ++ x)

instance Monad Logger where
  m >>= k =
    let (a, w) = execLogger m
        n = k a
        (b, x) = execLogger n
    in Logger (b, w ++ x)
```

注：现代版本 Haskell 在实现 Monad 实例之前还需要先实现 Functor 与 Applicative 实例。

序列化日志，而不是序列化计算

我们定义的 `(>>=)` 确保了左侧的日志信息将会附加到右侧的新的日志信息中。然而，它并没有涉及 `a` 与 `b` 的计算：`(>>=)` 是惰性的。

正如其它大多数的单子行为一样，严格程度是由单子的实现者所控制的。它不是所有单子都共享的常量。实际上一些单子由多种风格，每一种都会有不同的严格程度。

Writer 单子

我们的 `Logger` 单子实际上是标准 `Writer` 单子的特殊版本，可以在 `mtl` 库的 `Control.Monad.Writer` 模块中找到。关于 `Writer` 的例子将会在下一章中讲到。

Maybe 单子

`Maybe` 类型是最简单的 `Monad` 实例。它代表着计算可能不会返回一个结果。

当我们将一系列返回 `Maybe` 的计算通过 `(>>=)` 或 `>>` 串联起时，它们任意一个返回 `Nothing` 时，后续的计算都不再进行。

注意，尽管链条并不是完全短路的。链路中的每个 `(>>=)` 或 `(>>)` 仍然会匹配左侧的 `Nothing` ，并生产一个 `Nothing` 在右侧，直至最后。容易遗忘的点：当链路中的一个计算失败时，剩下的链路以及 `Nothing` 值的消费在运行时是廉价的，但并不是全免。

执行 Maybe 单子

执行 `Maybe` 单子的函数名为 `maybe` 。（记住“执行”一个单子包括对单子求值并返回一个去掉单子类型包装的结果。）

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x
```

它第一个参数是当结果为 `Nothing` 时，给定的返回；第二个参数则是当结果为 `Just` 时，应用到解包值上的函数。

Maybe 作为优秀的 API 设计

下面是一个 `Maybe` 作为单子使用的例子。给定一个用户名，希望通过移动电话运营商找到账单地址。

```
import Data.Map qualified as M

type PersonName = String

type PhoneNumber = String

type BillingAddress = String

data MobileCarrier
  = Honest_Bobs_Phone_Network
  | Morrisas_Marvelous_Mobiles
  | Petes_Plutocratic_Phones
  deriving (Eq, Ord)

findCarrierBillingAddress ::
  PersonName ->
  M.Map PersonName PhoneNumber ->
  M.Map PhoneNumber MobileCarrier ->
  M.Map MobileCarrier BillingAddress ->
  Maybe BillingAddress
```

我们的第一个版本是可怕的梯形代码，带着一堆 `case` 表达式：

```
variation1 :: (Ord k1, Ord k2, Ord k3) => k1 -> M.Map k1 k2 -> M.Map k2 k3 -> M.Map k3 a ->
  Maybe a
variation1 person phoneMap carrierMap addressMap =
  case M.lookup person phoneMap of
    Nothing -> Nothing
    Just number ->
      case M.lookup number carrierMap of
        Nothing -> Nothing
        Just carrier -> M.lookup carrier addressMap
```

`Data.Map` 模块的 `lookup` 函数拥有一个 monadic 返回类型

```
ghci> :module +Data.Map
ghci> :type Data.Map.lookup
Data.Map.lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

换言之，如果给定的键在 map 中，`lookup` 通过 `return` 将值注入至单子；反之，调用 `fail` 。这是很有意思的 API 设计，尽管我们会认为这是一个糟糕的选择。

- 正面来看，成功与失败的行为是根据调用 `lookup` 所产生的单子自动定值的。更妙的的是，`lookup` 本身并不需要知道或关系这些行为。
- 反面来看，问题在于错误的单子中使用 `file` 会抛出令人厌恶的异常。

使用 `do` 简化：

```
variation2 :: (Ord k1, Ord k2, Ord k3) => k1 -> M.Map k1 k2 -> M.Map k2 k3 -> M.Map k3 b ->
  Maybe b
variation2 person phoneMap carrierMap addressMap = do
  number <- M.lookup person phoneMap
  carrier <- M.lookup number carrierMap
  M.lookup carrier addressMap
```

如果这些 lookup 中任意一个失败了，那么 `(>>=)` 以及 `(>>)` 的定义意味着函数的整个结果将会变为 `Nothing` ，正如第一次尝试中显式的使用 `case` 那样。

通过 `flip` 还可以将函数体变为一行：

```
variation3 :: Ord a => a -> M.Map a a -> M.Map a a -> M.Map a b -> Maybe b
variation3 person phoneMap carrierMap addressMap =
  lookup phoneMap person >>= lookup carrierMap >>= lookup addressMap
  where
    lookup = flip M.lookup
```

列表单子

虽然 `Maybe` 类型即可以表示没有值，也可以表示一个值，但在很多情况下，我们希望返回一些事先不知道的结果。显然列表非常适合这个目的。列表的类型表明我们可以将它用作单子，因为他的类型构造函数有一个自由变量。当然，我们可以使用列表作为单子。

略。

如果将 `(>>=)` 应用至列表，我们会发现其类型为 `[a] -> (a -> [b]) -> [b]` 。这看起来就跟 `map` 类似：

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

看上去并不匹配，不过可以使用 `flip` ：

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type flip map
flip map :: [a] -> (a -> b) -> [b]
```

仍有一个问题：`flip map` 的第二个参数的类型是 `a -> b` ，而 `(>>=)` 对于列表的第二个参数是 `a -> [a]` 。我们该怎么做呢?

函数 `flip map` 可以返回任何 `b` 类型作为其结果。如果在签名中将 `b` 替换为 `[b]` ，那么类型签名就变为 `a -> (a -> [a]) -> [[b]]` 。换言之，如果我们 map 一个函数返回列表的列表，我们则会得到列表的列表作为返回。

```
ghci> flip map [1,2,3] (\a -> [a,a+100])
[[1,101],[2,102],[3,103]]
```

有趣的是，我们并没有真正改变类型签名。`(>>=)` 的类型是 `[a] -> (a -> [b]) -> [b]` ，而当映射函数返回列表时，`flip map` 的类型为 `[a] -> (a -> [b]) -> [[b]]` 。这里仍有一个类型不匹配；我们仅仅将一项从类型签名的中间移到了末尾。不过这并没有白费功夫：现在需要一个接受 `[[b]]` 并返回 `[b]` 的函数，这里推荐 `concat` ：

```
ghci> :type concat
concat :: [[a]] -> [a]
```

该类型建议我们翻转参数给到 `map` ，接着 `concat` 结果得到一个非嵌套的列表。

```
ghci> :type \xs f -> concat (map f xs)
\xs f -> concat (map f xs) :: [a] -> (a -> [a1]) -> [a1]
```

这正是列表的 `(>>=)` 的定义：

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

注：以上是旧版的定义。略。

理解列表单子

列表单子类似于我们熟悉的 Haskell 工具：列表表达式。我们可以通过两个列表的笛卡尔积来说明它们的相似性。首先是写一个列表表达式：

```
comprehensive xs ys = [(x, y) | x <- xs, y <- ys]
```

这一次，我们将对一元代码使用花括号表示法。这将突出一元代码在结构上与列表表达式的相似之处：

```
monadic xs ys = do x <- xs; y <- ys; return (x, y)
```

注：原本的 `monadic xs ys = do { x <- xs; y <- ys; return (x, y)}` 会被 HLS 简化成上述代码。

这里的唯一真正的区别是，我们构造的值出现在一系列表达式的末尾，而不是像列表表达式那样出现在开头。两者的结果是完全一致的：

```
ghci> comprehensive [1,2] "bar"
[(1,'b'),(1,'a'),(1,'r'),(2,'b'),(2,'a'),(2,'r')]
ghci> comprehensive [1,2] "bar" == monadic [1,2] "bar"
True
```

刚开始的时候很容易对列表单子感到困惑，让我们再次看一下单子笛卡尔乘积代码。这次将重新排列函数：

```
blockyDo :: Monad m => m a -> m b -> m (a, b)
blockyDo xs ys = do
  x <- xs
  y <- ys
  return (x, y)
```

对于列表 `xs` 中的每个元素，函数的其余部分求值一次，每次都将 `x` 绑定到列表中的不同的值上。然后对于列表 `ys` 中的每个元素，函数的剩余部分求值一次，每次都将 `y` 绑定到列表中的不同值。

这里我们真正拥有的是一个双重嵌套循环！这突出了一个关于单子的重要事实：除非知道一个单子代码块将在那个单子中执行，否则无法预测它的行为。

现在让嵌套的循环更加明显一些：

```
blockyPlain :: (Monad m) => m a -> m b -> m (a, b)
blockyPlain xs ys =
  xs
    >>= \x ->
      ys
        >>= \y -> return (x, y)

```

```haskell
blockyPlain_reloaded :: [a] -> [b] -> [(a, b)]
blockyPlain_reloaded xs ys =
  concatMap (\x -> concatMap (\y -> return (x, y)) ys) xs
```

注：这里与文中不同之处在于使用了 `concatMap` 而不是 `concat (map (...` 。

令列表单子生效

下面是一个简单的暴力约束求解。给定一个整数，它会找到所有正整数对，这些正整数相乘后会得到该值（这就是要解决的约束）。

```haskell
guarded :: Bool -> [a] -> [a]
guarded True xs = xs
guarded False _ = []

multiplyTo :: Int -> [(Int, Int)]
multiplyTo n = do
  x <- [1 .. n]
  y <- [x .. n]
  guarded (x * y == n) $ return (x, y)
```

测试：

```
ghci> multiplyTo 8
[(1,8),(2,4)]
ghci> multiplyTo 100
[(1,100),(2,50),(4,25),(5,20),(10,10)]
ghci> multiplyTo 891
[(1,891),(3,297),(9,99),(11,81),(27,33)]
```

do 代码块的脱糖

略。

单子作为可编程的分号

略。

为何选择无糖?

略。

State 单子

我们早在第十章节中就发现了 `Parse` 是一个单子。它有着两个独立的角度：一个是当解析失败时，提供带有细节的错误信息（通过 `Either` 类型达成）；另一个就是它携带着隐式状态，该例子中为 `ByteString` 。

这种读写状态的需求在 Haskell 程序中很常见，因此标准库提供了一个名为 `State` 的单子，其位于 `Control.Monad.State` 模块中。

当我们的 `Parse` 类型携带一个 `ByteString` 作为它的状态片段时， `state` 单子可以携带任何类型的状态。我们将状态的未知类型称为 `s` 。

给定一个状态值，我们检查它，然后产生一个结果和一个新的状态值。假设结果可以是任何类型 `a` 。捕获这个想法的类型签名是 `s -> (a, s)` ；获取一个状态 `s` ，对其做某些操作，然后返回结果 `a` ，以及一个新的状态 `s` 。

状态单子

我们先来开发一个 `State` 单子

```
type SimpleState s a = s -> (a, s)
```

我们的单子是一个函数，用于转换一个状态，并返回一个值。正因如此，状态单子有时候会被称为状态转换单子。

这是一个类型别名，而不是一个新类型，所以这里有点作弊，不过这可以简化很多之后的描述。

早在本章开头，我们提到了一个单子有一个单类型入参的类型构造函数，而这里则是带两个参数的类型。这里的关键是要理解我们可以部分应用类型，正如部分应用普通函数那样。接下来是一个简单的示例：

```
type StringState a = SimpleState String a
```

这里将变量 `s` 绑定至 `String` 。 `StringState` 类型仍有一个类型参数 `a` 。现在有了一个适合单子的类型构造函数，换言之，单子的类型构造函数是 `SimpleState s` ，而不仅仅只是 `SimpleState` 。

接下来是单子所需的 `return` 函数：

```
returnSt :: a -> SimpleState s a
returnSt a = \s -> (a, s)
```

它所做的就是获取结果和当前状态，并将它们“元组化”。到目前为止，我们可能已经习惯了这样的想法，即带有多个参数的 Haskell 函数知识一个单参数的函数链，不过以防万一，这里有一种更熟悉的编写 `returnSt` 的方法，它使这个函数更明显且简单：

```
returnAlt :: a -> SimpleState s a
returnAlt a s = (a, s)
```

单子最后一块定义就是 `(>>=)`，这里使用了标准库中 `State` 定义中的实际变量名：

```
bindSt :: SimpleState s a -> (a -> SimpleState s b) -> SimpleState s b
bindSt m k = \s -> let (a, s') = m s in k a s'
```

这些单字母变量名对可读性并没有什么好处，所以让我们看看是否可以替换一些更有意义的名称。

```
-- m == step
-- k == makeStep
-- s == oldState
bindAlt step makeStep oldState =
  let (result, newState) = step oldState
   in makeStep result newState
```

为了理解这个定义，记住 `step` 是一个类型为 `s -> (a, s)` 的函数。当需要计算它时，将会得到一个元组，我们需要使用它返回一个新的 `s -> (a, s)` 类型的函数。这恐怕比从 `bindAlt` 类型中去掉 `SimpleState` 类型别名，以及测试类型的入参与返回值，要来的更简单。

```
bindAlt :: (s -> (a, s)) -> (a -> s -> (b, s)) -> (s -> (b, s))
```

读取与修改状态

状态单子的 `(>>=)` 与 `return` 定义很简单：移动状态，但并不触碰该状态。

```
getSt :: SimpleState s s
getSt = \s -> (s, s)

putSt :: s -> SimpleState s ()
putSt s = const ((), s)
```

注：原文的 `putSt s = _ -> ((), s)` 现改为 `putSt s = const ((), s)` 。

真正的状态单子可用吗?

我们在前一节中使用的唯一简化技巧是为 `SimpleState` 使用类型别名而不是类型定义。如果引入了一个线类型的包装器，那么额外的包装和解包将使代码变得更难理解。

为了定义一个 `Monad` 实例，我们需要提供合适的类型构造函数，以及 `(>>=)` 与 `return` 的定义。这就是*真实的* `State` 定义。

```
newtype State s a = State
  { runState :: s -> (a, s)
  }
```

这里所做的是包装 `s -> (a, s)` 类型进一个 `State` 构造函数中。使用 Haskell 的 record 语义来进行类型定义，自动获得一个 `runState` 函数用来从其构造函数中解包一个 `State` 值。`runState` 的类型是 `State s a -> s -> (a, s)` 。

`return` 的定义与 `SimpleState` 几乎相同，除了需要通过 `State` 构造函数来进行包装。

```
returnState :: a -> State s a
returnState a = State (a,)
```

注：原文为 `returnState a = State $ s-> (a, s)` ，这里进行了简化。

`(>>=)` 的定义有点复杂，因为它需要使用 `runState` 来移除 `State` 包装器。

```
bindState :: State s a -> (a -> State s b) -> State s b
bindState m k = State $ \s ->
  let (a, s') = runState m s
   in runState (k a) s'
```

该函数与早前定义的 `bindSt` 仅在多了包装和解包动作上有所不同。

读与修改状态也是同理，添加包装：

```
get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ const ((), s)
```

注：原文为 `put s = State $ _ -> ((), s)` 。

使用状态单子：生成随机值

我们已经使用过状态单子的前身 `Parse` 用于解析二进制数据。这个例子中，是将要操作的状态类型直接连接到 `Parse` 类型中。

`State` 单子，相反，接受任意状态类型作为参数，`State ByteString` 。

如果拥有命令式语言的背景，那么状态单子可能会比其它单子更易接受。因为命令式语言都是关于携带一些隐式状态，读取一部分，并通过赋值修改其它部分，而这正是状态单子的作用。

Haskell 生成随机值的标准库名为 `System.Random` 。它允许任何类型的随机值生成，不仅仅是数值。该模块包含了若干方便的函数于 `IO` 单子中。例如，等同于 C 的 `rand` 函数：

```
import System.Random

rand :: IO Int
rand = getStdRandom $ randomR (0, maxBound)
```

（`randomR` 函数接受一个闭合的区间作为随机值的范围）

`System.Random` 模块提供一个 typeclass，`RandomGen` ，其供我们定义随机的 `Int` 源。`StdGen` 类型则是标准 `RandomGen` 的实例。它生成伪随机值。如果我们拥有真正随机数据的外部来源，便可以将其作为 `RandomGen` 的实例，并获得真正随机的值，而不仅仅是伪随机值。

另一个 typeclass，`Random` ，则指示如何生成特定类型的随机值。模块定义了所有常用类型的 `Random` 实例。

顺带一提，上面 `rand` 的定义读取并修改了一个内置的全局随机生成器，该生成器位于 `IO` 单子中。

尝试纯粹性

到目前为止，我们一直强调尽可能避免使用 `IO` 单子，如果只是为了生成一些随机值而使用它，那就很可惜。实际上 `System.Random` 包含了纯随机数生成函数。

纯函数的缺点是，我们必须获得或创建一个随机生成器，在调用它时，它返回一个新的随机数生成器：记住，在纯函数中我们不能修改现有生成器的状态。

如果我们忘记不可变性，并且在函数中重用相同的生成器，那么则会得到完全相同的“随机”数：

```
twoBadRandoms :: (RandomGen g) => g -> (Int, Int)
twoBadRandoms gen = (fst $ random gen, fst $ random gen)
```

`random` 函数使用了一个隐式范围而不是 `randomR` 里用户提供的范围。`getStdGen` 函数从 `IO` 单子中获取当前的全局标准数值生成器。

不幸的是，正确的传递和使用生成器的连续版本并不会产生令人满意的数。

```
twoGoodRandoms :: (RandomGen g) => g -> ((Int, Int), g)
twoGoodRandoms gen =
  let (a, gen') = random gen
      (b, gen'') = random gen'
   in ((a, b), gen'')
```

现在我们了解了状态单子，但是它看起来是隐藏生成器的一个很好的候选。状态单子允许我们规整的管理可变状态，同时保证我们的代码不会有其它意想不到的副作用，比如修改文件或建立网络连接。这使得我们更容易推断代码的行为。

状态单子中的随机值

下面是一个状态单子，携带一个 `StdGen` 作为它的状态：

```
type RandomState a = State StdGen a
```

生成一个随机值现在只需要获取当前的生成器，使用它，然后修改状态，用新的生成器替换它。

```haskell
getRandom :: (Random a) => RandomState a
getRandom =
  get >>= \gen ->
    let (val, gen') = random gen
     in put gen' >> return val
```

我们现在可以使用之前看到的一些一元机制来编写一个更简洁的函数，用于给出一对随机数：

```haskell
getTwoRandoms :: (Random a) => RandomState (a, a)
getTwoRandoms = liftM2 (,) getRandom getRandom
```

运行状态单子

正如之前提到的那样，每个单子都有其特化的计算函数。本例的状态单子，有若干种选择：

- `runState` 同时返回结果和最终状态；
- `evalState` 仅返回结果，丢弃最终状态；
- `execState` 丢弃结果，仅返回最终状态。

`evalState` 与 `execState` 函数仅仅是 `runState` 以及 `fst` 与 `snd` 的组合。因此最需要记住的是 `runState` 函数。

以下是如何实现 `getTwoRandoms` 函数的完整用例：

```haskell
runTwoRandoms :: IO (Int, Int)
runTwoRandoms = do
  oldState <- getStdGen
  let (result, newState) = runState getTwoRandoms oldState
  setStdGen newState
  return result
```

`runState` 的调用遵循一个标准模式：传递它至一个包含在状态单子中的函数，以及一个初始状态；它返回函数的结果以及最终状态。

围绕 `runState` 调用的代码仅仅获取当前全局 `StdGen` 值，然后替换它，以便后续对 `runTwoRandoms` 或其他随机生成函数的调用可以获得更新的状态。

更多特性

很难想象编写只有一个状态值要传递的代码。当我们想要一次跟踪多个状态片段时，通常的技巧是在一个数据类型中维护它们。这里有一个例子：跟踪我们分发的随机数的数量：

```haskell
data CountedRandom = CountedRandom
  { crGen :: StdGen,
```

```
    crCount :: Int
  }

type CRState = State CountedRandom

getCountedRandom :: (Random a) => CRState a
getCountedRandom = do
  st <- get
  let (val, gen') = random (crGen st)
  put CountedRandom {crGen = gen', crCount = crCount st + 1}
  return val
```

该例子碰巧消耗了状态的两个元素，并构造了一个全新的状态。更普遍的场景是，可能只读取或修改状态的一部分。该函数获取到目前为止生成的随机值数量。

```
getCount :: CRState Int
getCount = gets crCount
```

注：原文为 `getCount = crCount `liftM` get`，这里用 `gets crCount` 进行简化。

如果我们希望更新部分状态，那么代码可能就没那么显而易见了。

```
putCount :: Int -> CRState ()
putCount a = do
  st <- get
  put st {crCount = a}
```

还存在一个名为 `modify` 的函数，其组合了 `get` 与 `put` 步骤。它将状态转换函数作为参数，但它并不令人满意：我们仍然无法摆脱笨拙的记录更新语法：

```
putCountModify :: Int -> CRState ()
putCountModify a = modify $ \st -> st {crCount = a}
```

单子与函子

略（原文 Haskell 版本过旧，已不适合现在的学习）。

15 Programming with monads

练习：关联列表

Web 客户端与服务端之间经常通过简单的键值对列表进行信息传输。

```
name=Attila+%42The+Hun%42&occupation=Khan
```

这里的编码名为 `application/x-www-form-urlencoded`，同时非常便于理解。每个键值对都被一个“&”符号分隔。在一个兼职对中，键为“=”符号之前的所有字符，而值为之后的所有字符。

显然可以将一个 `String` 作为键，但是 HTTP 并不清楚该键是否必须跟着一值。可以通过 `Maybe String` 来表示一个模糊的值。如果值为 `Nothing`，即无值可展示。当使用 `Just` 包裹一个值时则以为着有值。使用 `Maybe` 让我们可以区分“无值”与“空值”。

Haskell 程序员使用类型为 `[(a, b)]` 的*关联列表*，可以视作关联列表中的每个元素都是键与值的关联。

假设我们想用这些列表中的一个来填充一个数据结构。

```
data MovieReview = MovieReview
  { revTitle :: String,
    revUser :: String,
    revReview :: String
  }
```

从一个朴素的函数开始：

```
simpleReview :: [(String, Maybe String)] -> Maybe MovieReview
simpleReview alist =
  case lookup "title" alist of
    Just (Just title@(_ : _)) ->
      case lookup "user" alist of
        Just (Just user@(_ : _)) ->
          case lookup "review" alist of
            Just (Just review@(_ : _)) ->
              Just (MovieReview title user review)
            _ -> Nothing -- no review
        _ -> Nothing -- no user
    _ -> Nothing -- no title
```

当关联列表包含了所有必要值且不为空值时，它将返回一个 `MovieReview`。

我们对 `Maybe` 单子已经很熟悉了，因此可以简化一下上述的阶梯式代码：

```
maybeReview :: [(String, Maybe [Char])] -> Maybe MovieReview
maybeReview alist = do
  title <- lookup1 "title" alist
  user <- lookup1 "user" alist
  review <- lookup1 "review" alist
```

```
    return $ MovieReview title user review

lookup1 :: Eq a1 => a1 -> [(a1, Maybe [a2])] -> Maybe [a2]
lookup1 key alist =
  case lookup key alist of
    Just (Just s@(_ : _)) -> Just s
    _ -> Nothing
```

尽管这看起来简洁多了，但仍然在重复自身。我们可以利用 `MoviewReview` 构造函数作为一个普通的纯函数，通过 *lifting* 它至单子：

```
liftedReview :: [(String, Maybe [Char])] -> Maybe MovieReview
liftedReview alist =
  liftM3
    MovieReview
    (lookup1 "title" alist)
    (lookup1 "user" alist)
    (lookup1 "review" alist)
```

这里仍然有一些重复，不过更难简化。

泛化的 lifting

尽管使用 `liftM3` 得以简化代码，但是我们无法使用 liftM 家族的函数来解决更泛化的问题，因为标准库仅定义到了 `liftM5` 。虽然可以自定义此类型的变体函数，但是这个数量仍然是一个问题。

假设一个构造函数或者纯函数接收十个参数，并决定坚持使用标准库，这可能就不太合适了。

在 `Control.Monad` 中，有一个名为 `ap` 的函数拥有着有趣的类型签名。

```
ghci> :m +Control.Monad
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

我们可能会疑惑谁会将单参数纯函数放在单子中，且为什么。不过回想一下所有的 Haskell 函数实际上只接受一个参数，这里开始将看到其与 `MovieReview` 构造函数的管理。

```
ghci> :type MovieReview
MovieReview :: String -> String -> String -> MovieReview
```

我们当然可以简单的将类型写作 `String -> (String -> (String -> MovieReview))` 。如果使用旧的 `liftM` 将 `MovieReview` 提升至 `Maybe` 单子，那么我们将会得到类型为 `Maybe (String -> (String -> (String -> MovieReview)))` 的值。现在可以看出来该类型适用于单个参数的 `ap` 。我们可以一次将这个传递给 `ap` ，并继续链式执行，直到得到该定义。

```haskell
apReview :: [(String, Maybe [Char])] -> Maybe MovieReview
apReview alist =
  MovieReview
    `liftM` lookup1 "title" alist
    `ap` lookup1 "user" alist
    `ap` lookup1 "review" alist
```

注：以下为步骤拆解后的类型变化。

```haskell
MovieReview :: String -> ( String -> String -> MovieReview )
MovieReview `liftM` :: Maybe String -> Maybe ( String -> String -> MovieReview )
MovieReview `liftM` lookup1 "title" alist :: Maybe ( String -> String -> MovieReview )
MovieReview `liftM` lookup1 "title" alist `ap` :: Maybe String -> Maybe ( String ->
  MovieReview )
MovieReview `liftM` lookup1 "title" alist `ap` lookup1 "user" alist :: Maybe ( String ->
  MovieReview )
```

我们可以像这样把 `ap` 的应用链接起来，只要有需要就可以多次链接，从而绕过 `liftM` 系列函数。

看待 `ap` 的另一种有用的方式是，它是我们熟悉的 `($)` 操作符的一元等价物：可以把 `ap` 读作 *apply*。当比较两者的函数签名时可知：

```haskell
ghci> :type ($)
($) :: (a -> b) -> a -> b
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

实际上，`ap` 通常被定义为 `liftM2 id` 或是 `liftM2 ($)` 。

寻找其它方案

以下是某人的电话号码：

```haskell
data Context = Home | Mobile | Business deriving (Eq, Show)

type Phone = String

albulena :: [(Context, String)]
albulena = [(Home, "+355-652-55512")]

nils :: [(Context, String)]
nils =
  [ (Mobile, "+47-922-55-512"),
    (Business, "+47-922-12-121"),
    (Home, "+47-925-55-121"),
    (Business, "+47-922-25-551")
  ]

twalumba :: [(Context, String)]
```

```
17  twalumba = [(Business, "+260-02-55-5121")]
```

假设我们想要通过打电话来联系某人。我们不希望通过商务号码，更倾向于使用家庭号码（如果存在的话）而不是移动电话。

```
1  onePersonalPhone :: [(Context, Phone)] -> Maybe Phone
2  onePersonalPhone ps =
3    case lookup Home ps of
4      Nothing -> lookup Mobile ps
5      Just n -> Just n
```

当然我们可以使用 `Maybe` 作为返回类型，我们无法考虑到某人可能拥有多个号码的可能性。因此我们需要一个列表：

```
1  allBusinessPhones :: [(Context, Phone)] -> [Phone]
2  allBusinessPhones ps = map snd numbers
3    where
4      numbers =
5        case filter (contextIs Business) ps of
6          [] -> filter (contextIs Mobile) ps
7          ns -> ns
8
9  contextIs :: Eq a => a -> (a, b) -> Bool
10 contextIs a (b, _) = a == b
```

注意这两个函数的 `case` 表达式结构类似：一个替代方法处理第一次查找返回空值，而另一个方法处理非空情况。

```
1  ghci> onePersonalPhone twalumba
2  Nothing
3  ghci> onePersonalPhone albulena
4  Just "+355-652-55512"
5  ghci> allBusinessPhones nils
6  ["+47-922-12-121","+47-922-25-551"]
```

Haskell 的 `Control.Monad` 模块定义了一个 typeclass，`MonadPlus` ，它让我们可以出 `case` 表达式中抽象出公共模式。

```
1  ghci> import Control.Monad
2  ghci> :i MonadPlus
3  type MonadPlus :: (* -> *) -> Constraint
4  class (GHC.Base.Alternative m, Monad m) => MonadPlus m where
5    mzero :: m a
6    mplus :: m a -> m a -> m a
7          -- Defined in ‘GHC.Base’
8  instance MonadPlus IO -- Defined in ‘GHC.Base’
9  instance MonadPlus [] -- Defined in ‘GHC.Base’
10 instance MonadPlus Maybe -- Defined in ‘GHC.Base’
```

`mzero` 代表一个空值，而 `mplus` 则是将两个结果合并成一个。我们现在可以使用 `mplus` 来完全的移除 `case` 表达式了。

```
oneBusinessPhone :: [(Context, Phone)] -> Maybe Phone
oneBusinessPhone ps = lookup Business ps `mplus` lookup Mobile ps

allPersonalPhones :: [(Context, Phone)] -> [Phone]
allPersonalPhones ps =
  map snd $
    filter (contextIs Home) ps
      `mplus` filter (contextIs Mobile) ps
```

这些函数中由于我们知道 `lookup` 返回一个类型为 `Maybe` 的值，以及 `filter` 返回一个列表，那么 `mplus` 使用的版本就很明显了。

更有趣的是，我们可以使用 `mzero` 与 `mplus` 来编写任何对 `MonadPlus` 实例都有用的函数。以下是标准查找函数的例子：

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
lookup _ []                     = Nothing
lookup k ((x,y):xys) | x == k    = Just y
                     | otherwise = lookup k xys
```

我们可以轻易地泛化返回类型至任意 `MonadPlus` 的实例：

```
lookupM :: (MonadPlus m, Eq a) => a -> [(a, b)] -> m b
lookupM _ [] = mzero
lookupM k ((x, y) : xys)
  | x == k = return y `mplus` lookupM k xys
  | otherwise = lookupM k xys
```

如果结果类型是 `Maybe` ，即要么没有结果，要么一个结果；如果类型是列表，即所有结果；或者其它更适合于 `MonadPlus` 的奇异实例。

mplus 并不是加法

尽管 `mplus` 函数包含了“plus”，但这并不意味着将两个值求和。

根据单子定义 `mplus` 可能会实现类似于加法的操作。例如立标单子中的 `mplus` 是作为 `(++)` 操作符实现的。

```
ghci> [1,2,3] `mplus` [4,5,6]
[1,2,3,4,5,6]
```

然而切换到其他单子，类似加法的行为并不成立：

```
ghci> Just 1 `mplus` Just 2
Just 1
```

MonadPlus 的规则

`MonadPlus` typeclass 的实例相较于普通的单子规则还需要遵循某些其他简单的规则。

如果 `mzero` 出现在绑定表达式的左侧，则实例必须短路。换言之，表达式 `mzeor >>= f` 的计算结果必须与 `mzero` 单独计算的结果相同。

```
mzero >>= f == mzero
```

如果 `mzero` 出现在序列表达式的*右侧*，那么该实例必须短路。

```
v >> mzero == mzero
```

失败安全的 MonadPlus

早在“The Monad typeclass”章节中提到的 `fail` 函数，被告知不要使用它：在很多 `Monad` 中，它被实现为对 `error` 的调用，这会产生令人不快的后果。

`MonadPlus` typeclass 为我们提供了一种更温和的方式来失败计算，而不会出现 `fail` 或 `error` 。上面介绍的规则允许我们在任何需要的地方在代码中引入 `mzero` ，并在该点上短路。

在 `Control.Monad` 模块中，标准函数 `guard` 将此理念打包成了方便的样式。

```
guard        :: (MonadPlus m) => Bool -> m ()
guard True   =  return ()
guard False  =  mzero
```

下面是个简单的例子，一个函数接受一个值 `x` 并计算它对另一个数字 `n` 取模。如果结果为零返回 `x` ，否则返回当前单子的 `mzero` 。

```
x `zeroMod` n = guard ((x `mod` n) == 0) >> return x
```

隐藏管道的冒险

略。

我们给单子取名为 `Supply` ，将执行函数 `runSupply` 提供一个列表；需要确保列表中每一个元素都是唯一的。

```
runSupply :: Supply s a -> [s] -> (a, [s])
```

单子并不会关心内部的值：它们有可能是随机数，临时文件的名称，或是 HTTP cookies 的 ID。

单子内部，消费者每次需求一个值，`next` 则会从列表中获取下一个元素并给到消费者。每个值都会被 `Maybe` 构造函数包装，以防列表长度不够。

```
next :: Supply s (Maybe s)
```

为了隐藏管道，模块声明时仅导出类型构造函数，执行函数，以及 `next` 操作函数：

```
module Supply (Supply, next, runSupply) where
```

管道非常的简单：使用 `newtype` 声明来包装一个 `State` 单子：

```haskell
import Control.Monad.State

newtype Supply s a = S (State [s] a)
```

这里型参 `s` 是我们将要提供的唯一值类型，而 `a` 是为了类型成为单子而必须提供的通常类型参数。

我们对 `Supply` 类型的 `newtype` 的使用和模块头文件联合起来防止用户使用 `State` 单子的 `get` 和 `set` 操作。由于模块没有导出 `S` 的构造函数，所以外部无法通过编程的方式看到包装的 `State` 单子，也无法访问它。

此刻有了类型 `Supply`，我们需要它来创建 `Monad` typeclass 实例。我们可以遵循 `(>>=)` 以及 `return` 的通常模式，但这只是存粹的样板代码。我们所要做的是包装盒解包 `State` 单子的 `(>>=)` 版本，并使用 `S` 值构造函数返回。

```haskell
unwrapS :: Supply s a -> State [s] a
unwrapS (S s) = s

instance Functor (Supply s) where
  fmap f s = S $ fmap f $ unwrapS s

instance Applicative (Supply s) where
  pure = S . return
  f <*> a = S $ unwrapS f <*> unwrapS a

instance Monad (Supply s) where
  s >>= m = S $ unwrapS s >>= unwrapS . m
```

注：与原文不同，实现单子实例前还需分别实现函子实例与应用函子实例。

Haskell 程序员不喜欢样板文件，且可以肯定的是，GHC 有一个可爱的语言扩展用于消除这些样板。使用它需要再源文件顶部添加：

```haskell
{-# LANGUAGE GeneralisedNewtypeDeriving #-}
```

通常而言，我们只能自动派生一些标准 typeclass 的实例，例如 `Show` 和 `Eq` 。顾名思义，`GeneralisedNewtypeDeriving` 扩展了派生 typeclass 实例的能力，且它是特定于 `newtype` 声明的。如果我们包装的类型是任何 typeclass 的实例，扩展可以自动将我们的新类型作为该 typeclass 的实例，如下所示：

```haskell
newtype Supply s a = S (State [s] a)
  deriving (Monad)
```

那么接下来就是 `next` 与 `runSupply` 的定义了：

```haskell
runSupply :: Supply s a -> [s] -> (a, [s])
runSupply (S m) xs = runState m xs

next :: Supply s (Maybe s)
```

```
next = S $ do
  st <- get
  case st of
    [] -> return Nothing
    (x : xs) -> do
      put xs
      return $ Just x
```

加载模块至 **ghci** 测试一下：

```
ghci> :l Supply
[1 of 1] Compiling Supply           ( Supply.hs, interpreted )
Ok, one module loaded.
ghci> runSupply next [1,2,3]
(Just 1,[2,3])
ghci> import Control.Monad
ghci> runSupply (liftM2 (,) next next) [1,2,3]
((Just 1,Just 2),[3])
ghci> runSupply (liftM2 (,) next next) [1]
((Just 1,Nothing),[])
```

我们还可以验证 `State` 单子是否泄漏了。

```
ghci> :browse Supply
type role Supply nominal nominal
type Supply :: * -> * -> *
newtype Supply s a = S (State [s] a)
runSupply :: Supply s a -> [s] -> (a, [s])
next :: Supply s (Maybe s)
ghci> :info Supply
type role Supply nominal nominal
type Supply :: * -> * -> *
newtype Supply s a = S (State [s] a)
        -- Defined at Supply.hs:12:1
instance Applicative (Supply s) -- Defined at Supply.hs:32:10
instance Functor (Supply s) -- Defined at Supply.hs:29:10
instance Monad (Supply s) -- Defined at Supply.hs:36:10
```

支持随机数

如果想要使用 `Supply` 单子作为随机数的源，那么我们将会遇到一个小困难。理想情况下，我们希望能够为它提供无限流式的随机数。我们可以在 `IO` 单子中获得一个 `StdGen` ，但是当完成时必须“放回”一个不同的 `StdGen` 。如果不这么做，那么下一次获取 `StdGen` 的代码将获得相同的状态，即产生一样的随机数，这可是灾难性的事故。

从 `System.Random` 模块中可知，目前的需求很难被调和。我们可以使用 `getStdRandom` ，其类型可以确保获得一个 `StdGen` 时，又会放回一个。

```
ghci> :type getStdRandom
```

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

在给到一个随机值后，可以使用 `random` 来获取一个新的 `StdGen` ；可以使用 `randoms` 来获取一个随机数的无限列表。但是我们该怎么得到一个无限的随机数列表和一个新的 `StdGen` 呢？

答案就在 `RandomGen` typeclass 的 `split` 函数内，该函数接受一个随机数生成器，并将其转换为两个生成器。像这样拆分随机生成器是最不寻常的事：它在纯函数设置中显然非常有用，但本质上既不是必须的，也不是由非纯语言提供的。

使用 `split` 函数时，可以使用 `StdGen` 来生成一个无限随机数列表用于 `runSupply` ，另一个则是用于 `IO` 单子：

```
import Supply
import System.Random hiding (next)

randomsIO :: (Random a) => IO [a]
randomsIO = getStdRandom $ \g ->
  let (a, b) = split g in (randoms a, b)
```

如果我们正确的编写了这个函数，那么示例应该在每次调用时打印一个不同的随机数：

```
ghci> :l RandomSupply.hs
[1 of 2] Compiling Supply           ( Supply.hs, interpreted )
[2 of 2] Compiling RandomSupply     ( RandomSupply.hs, interpreted )
Ok, two modules loaded.
ghci> (fst . runSupply next) `fmap` randomsIO
Just (-8154423328023582499)
ghci> (fst . runSupply next) `fmap` randomsIO
Just (-4209314352233312889)
```

回忆一下， `runSupply` 函数即返回执行一元操作的结果，也返回列表中未使用的剩余部分。由于我们向它传递了一个随机数的无限列表，因此使用 `fst` 以确保在 **ghci** 尝试打印结果时不会被随机数淹没。

再一次尝试

将函数应用与一对元组中的一个元素，并在不改变另一个原始元素的情况下构造一个新元组的模式，这在 Haskell 代码中很常见，以至于它已经变成了标准代码。

`Control.Arrow` 模块中有两个函数， `first` 与 `second` ，即实现了该操作。

```
ghci> :m +Control.Arrow
ghci> first (+3) (1,2)
(4,2)
ghci> second odd ('a',1)
('a',True)
```

从实现中分离接口

之前的小节中，我们见识到了在使用 `State` 用于维护 `Supply` 的状态时，是如何隐藏实现的。

另一个让代码更模块化的方式则是分离其*接口* – 即代码可以做的，与*实现* – 即代码如何做的。

`System.Random` 模块中的标准随机数生成器是很低效的。如果使用 `randomsIO` 函数来提供随机数，那么 `next` 操作的性能不会很好。

一个简单高效的处理方式就是为 `Supply` 提供一个更好的随机数据源。现在让我们将这个想法放在一旁，而去考虑另一种方式，一种在很多设置中有用的方式。我们将单子可以执行的操作与它使用的 typeclass 的工作方式分离开来。

```
class (Monad m) => MonadSupply s m | m -> s where
  next :: m (Maybe s)
```

该 typeclass 定义了任何 supply 单子必须实现的接口。他需要仔细检查，因为它使用了几个暂不熟悉的 Haskell 语言扩展。我们将在接下来的章节中逐一介绍。

若干参数的 typeclasses

我们该如何阅读代码切片 `MonadSupply s m` 这个 typeclass 呢? 如果添加圆括号，那么一个相同的表达式就是 `(MonadSupply s) m` ，这更清晰一点。换言之，给定某身为 `Monad` 的类型变量 `m` ，我们可以使其成为 `MonadSupply s` 的实例。有别于通常的 typeclass，它有一个*参数*。

语言扩展允许一个 typeclass 拥有多个参数，其名称为 `MultiParamTypeClasses` 。参数 `s` 的作用与同名的 `Supply` 类型参数相同：它表示下一个函数传递的值的类型。

注意，我们不需要在 `MonadSupply s` 的定义中提到 `(>>=)` 或 `return` ，因为 typeclass 的上下文（superclass）要求 `MonadSupply s` 必须是 `Monad` 。

函数式依赖

回忆一下早前忽略的代码片段，`| m -> s` 是一个*函数式依赖*，通常称为 *fundep*。我们可以将竖线 `|` 读作“such that”，而箭头 `->` 读作“uniquely determines”。函数式依赖建立了关于 `m` 与 `s` 之间的*关系*。

功能依赖是由 `FunctionalDependencies` 的语言 pragma 控制的。

声明关系的目的是帮助类型检查器。回想一下，Haskell 类型检查器本质上是一个定理证明器，且他的操作方式是保守的：它坚持它的证明必须终止。非终止证明会导致编译器放弃或陷入无限循环。

通过函数式依赖，我们告诉类型检查器，每当它看到 `MonadSupply s` 的上下文中使用某些单子 `m` 时，类型 `s` 是唯一可接受的类型。如果我们忽略函数式依赖，类型检查器将简单的放弃并显示一条错误信息。

现在看一下这个 typeclass 的实例：

```
import qualified Supply as S

instance MonadSupply s (S.Supply s) where
  next = S.next
```

这里类型变量 `m` 被类型 `S.Supply s` 取代。由于函数式依赖，类型检查器知道了当它看到 `S.Supply s` 时，该类型可以用作 typeclass `MonadSupply s` 的实例。

为了去掉最后一层抽象，试着考虑一下类型 `S.Supply Int` 。如果没有函数式依赖，我们可以将其声明为 `MonadSupply s` 的实例。然而在尝试使用该实例时，编译器将无法确定类型的 `Int` 参数需要与 typeclass 的 `s` 参数相同，同时编译器将报告一个错误。

函数式依赖关系可能很难理解，一旦超出了简单使用，它们在实践中将很难使用。幸运的是，函数依赖关系最常见的使用方式就是像上述这样的简单情况，它们造成的麻烦会少很多。

完善我们的模块

当我们将 typeclass 与实例保存在一个名为 `SupplyClass.hs` 的源文件中，我们需要添加一个模块头如下：

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module SupplyClass
  ( MonadSupply (..),
    S.Supply,
    S.runSupply,
  )
where
```

为了让编译器接受我们的实例声明，必须使用 `FlexibleInstances` 扩展。这个扩展放宽了某些情况下编写实例的正常规则，在某种程度上仍然让编译器的类型检查器保证它被终止。这里还需要使用 `FlexibleInstances` 是因为使用了函数式依赖的关系，不过这些细节超出了本书的范围。

> **Tip**
> 如何知道哪个语言扩展是被需要的

如果 GHC 因为需要某些语言扩展被开启才能对某部分代码进行编译，它会直接告诉用户需要那些扩展。例如，如果它认为我们的代码需要灵活实例的支持，它会建议我们使用 `-XFlexibleInstances` 选项来进行编译。一个 `-X` 选项与 `LANGUAGE` 的效果一样：开启特定的扩展。

最后请注意，我们将重新导出这个模块中的 `runSupply` 以及 `Supply` 。从一个模块导出名称是完全合法的，即使它是在另一个模块中定义的。这个例子中，意味着用户只需要导入 `SupplyClass` 模块，而不需要导入 `Supply` 模块。这减少了用户需要记住的“移动部件”的数量。

对单子接口进行编程

以下是一个简单的函数用于提取 `Supply` 单子的两个值，并格式化它们成为一个字符串进行返回：

```
showTwo :: (Show s) => Supply s String
showTwo = do
  a <- next
  b <- next
  return $ show "a: " ++ show a ++ ", b: " ++ show b
```

这段代码通过其结果类型绑定到 `Supply` 单子。通过修改函数类型，我们可以很容易的泛化到任何实现 `MonadSupply` 接口的单子。注意函数体保持不变：

```
showTwoClass :: (Show s, Monad m, MonadSupply s m) => m String
showTwoClass = do
  a <- next
  b <- next
  return $ show "a: " ++ show a ++ ", b: " ++ show b
```

reader 单子

`State` 单子允许我们在代码中探索可变状态。有时我们希望能够传递一些不可变的状态，例如程序的配置数据。我们可以使用 `State` 单子来实现这个目的，但是我们可能会发现自己意外的修改了应该保持不变的数据。

现在考虑一下具有上述期望的特征的函数应该做什么。它应该接受我们传入的数据某种类型 `e` 的值，并返回另一种类型 `a` 的值作为结果，即 `e -> a` 。

将该类型转换为一个方便的 `Monad` 实例仅需 `newtype` 包装：

```
newtype Reader e a = R {runReader :: e -> a}
```

使其成为 `Monad` 实例并不需要做太多的工作：

```haskell
instance Functor (Reader a) where
  fmap f m = R $ f . runReader m

instance Applicative (Reader a) where
  pure = R . const
  f <*> m = R $ \r -> runReader f r $ runReader m r

instance Monad (Reader e) where
  m >>= k = R $ \r -> runReader (k $ runReader m r) r
```

注：原文代码过旧。

我们可以把类型 `e` 看作是计算某个表达式的环境。无论环境是什么，`return` 操作都应该具有相同效果，即忽略它的环境。

`(>>=)` 的定义稍微复杂一些，仅仅是因为需要将环境 – 这里是变量 `r` – 在当前计算与链接的计算上都适用。

在单子中执行的一段代码如何知道它的环境中有什么？仅需 `ask` ：

```haskell
ask :: Reader e e
ask = R id
```

在给定的操作链中，每次调用 `ask` 都将返回相同的值，因为存储在环境中的值不会改变。在 **ghci** 中进行测试：

```
ghci> :l SupplyClass.hs
[1 of 2] Compiling Supply           ( Supply.hs, interpreted )
[2 of 2] Compiling SupplyClass      ( SupplyClass.hs, interpreted )
Ok, two modules loaded.
ghci> runReader (ask >>= \x -> return (x * 3)) 2
6
```

`Reader` 单子包含在标准的 `mtl` 库中，这个库通常与 GHC 捆绑在一起。我们可以在 `Control.Monad.Reader` 模块中找到它。这个单子的动机最初看起来可能有点单薄，但是它在复杂的代码中很有用。我们经常要访问程序内部深处的配置信息；将这些信息作为普通参数传递进来，则需要对代码进行痛苦的重构。将这些信息隐藏在单子中，不关心配置信息的中间函数则不需要看到它。

`Reader` 单子的最明确动机将出现在第 18 章*单子转换*中的合并几个单子来构建一个新的单子的讨论中。这里我们则是看到了如何更好的控制状态，以便我们的代码可以通过 `State` 单子修改一些值，而其他值则通过 `Reader` 单子保持不变。

返回自动推导

现在我们了解了 `Reader` 单子，让我们用它创建 `MonadSupply` typeclass。为了保持示例的简单性，在这里违背 `MonadSupply` 的精神：下一个动作总是返回相同的值，而不是总返回不同的值。

直接将 `Reader` 类型转为 `MonadSupply` 类的实例是一个坏主意，因为这样任何 `Reader` 都可以充当 `MonadSupply` ，这通常没有任何意义。

创建一个基于 `Reader` 的 `newtype` ，它隐藏了内部使用 `Reader` 的事实。现在必须使类型成为我们所关心的两个 typeclasses 实例。启用了 `GeneralizedNewtypeDeriving` 扩展后，GHC 将会帮我们做大部分的苦力活。

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}

import Control.Monad
import SupplyClass

newtype MySupply e a = MySupply {runMySupply :: Reader e a}
  deriving (Functor, Applicative, Monad)

instance MonadSupply e (MySupply e) where
  next = MySupply $ Just `liftM` ask
```

注：`deriving Monad` 之前还要加上 `Functor` 和 `Applicative` 。

注意我们必须让类型成为 `MonadSupply e` 的一个实例，而不是 `MonadSupply` 。如果省略类型变量，编译器会报错。

测试 `MySupply` 类型：

```
xy :: (Num s, MonadSupply s m, MonadFail m) => m s
xy = do
  Just x <- next
  Just y <- next
  return $ x * y
```

注：与原文不同，这里 HLS 加上了 `MonadFail m` 的约束。

略（原文此处难以理解）。

隐藏 IO 单子

`IO` 单子的优点和缺点是它非常强大。如果谨慎使用类型可以帮助我们避免编程错误，那么 `IO` 单子应该是一个很大的不安来源。因为 `IO` 单子对我们能做的事情没有任何限制，它使我们容易受到各种事故的影响。

我们怎样才能驯服它呢？假设我们想要保证一段代码可以读写本地文件系统上的文件，但它不能访问网络。我们不能使用普通的 `IO` 单子，因为它不会限制我们。

使用 newtype

现在让我们创建一个模块提供一些用于读写文件的功能。首先是创造一个约束版本的 `IO` 将其用 `newtype` 包裹。

```
newtype HandleIO a = HandleIO {runHandleIO :: IO a}
  deriving (Functor, Applicative, Monad)
```

从模块导出类型构造函数和 `runHandleIO` 执行函数，但不导出数据构造函数。这可以防止在 `HandleIO` 单子内运行的代码获得它所包装的 `IO` 单子。

剩下要做的就是包装希望 `monad` 允许的每个动作。这是一个用 `HandleIO` 数据构造函数包装每个 `IO` 。

```
import qualified System.IO as IO

openFile :: FilePath -> IO.IOMode -> HandleIO IO.Handle
openFile path mode = HandleIO $ IO.openFile path mode

hClose :: IO.Handle -> HandleIO ()
hClose = HandleIO . IO.hClose

hPutStrLn :: IO.Handle -> String -> HandleIO ()
hPutStrLn h s = HandleIO $ IO.hPutStrLn h s
```

现在我们可以使用被约束的 `HandleIO` 单子了：

```
safeHello :: FilePath -> HandleIO ()
safeHello path = do
  h <- openFile path IO.WriteMode
  hPutStrLn h "hello world"
  hClose h
```

测试：

```
ghci> :l HandleIO.hs
[1 of 1] Compiling HandleIO         ( HandleIO.hs, interpreted )
Ok, one module loaded.
ghci> runHandleIO (safeHello "hello_world_101.txt")
```

为意想不到的用途设计

`HandleIO` 单子有一个小但却重要的问题：它没有考虑到我们可能偶尔需要一个逃生舱口的可能性。如果像这样定义一个单子，可能会偶尔需要执行单子不允许的 I/O 操作。

我们这样定义单子的目的是为了更容易在常见情况下编写可靠的代码，而不是让极端情况变得不可能。

`Control.Monad.Trans` 模块定义了一个“标准逃生舱”，`MonadIO` typeclass。它定义了一个单独的函数，`liftIO` ，让我们可以将一个 `IO` 操作镶嵌进另一个单子中。

```
ghci> :m +Control.Monad.Trans
ghci> :i MonadIO
type MonadIO :: (* -> *) -> Constraint
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
  {-# MINIMAL liftIO #-}
        -- Defined in ‘Control.Monad.IO.Class’
instance [safe] MonadIO IO -- Defined in ‘Control.Monad.IO.Class’
```

这个 typeclass 的实现很简单：仅用构造函数来包装 `IO` 。

```
import Control.Monad.Trans (MonadIO (..))

instance MonadIO HandleIO where
  liftIO = HandleIO
```

通过明智的使用 `liftIO` ，我们可以摆脱束缚并在必要时调用 `IO` 操作。

```
import System.Directory (removeFile)

tidyHello :: FilePath -> HandleIO ()
tidyHello path = do
  safeHello path
  liftIO $ removeFile path
```

Tip

自动派生与 MonadIO

通过将 typeclass 添加到 `HandleIO` 的派生子句中，可以让编译器自动派生 `MonadIO` 实例。实际上在生产代码中，这是常用的策略。我们在这里避免了这点，仅仅是为了将早期材料的呈现与 `MonadIO` 分开。

使用 typeclasses

将 `IO` 隐藏在另一个单子中的缺点就是：它仍然与具体的实现绑定到了一起。如果想要将 `HandleIO` 替换为其他单子，则必须更改使用 `HandelIO` 的每个操作类型。

作为替换方法，我们可以创建一个 typeclass 指定从操作文件的单子中获得接口。

```
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module MonadHandle
  ( MonadHandle (..),
  )
where

```

```haskell
import qualified System.IO as IO

class (Monad m) => MonadHandle h m | m -> h where
  openFile :: FilePath -> IO.IOMode -> m h
  hPutStr :: h -> String -> m ()
  hClose :: h -> m ()
  hGetContents :: h -> m String

  hPutStrLn :: h -> String -> m ()
  hPutStrLn h s = hPutStr h s >> hPutStr h "\n"
```

这里我们选择抽象单子的类型和文件句柄的类型。为了满足类型检查器的要求，我们添加了一个函数依赖：对于 `MonadHandle` 的任何实例，只能使用一种句柄类型。当我们将 `IO` 单子作为该类的一个实例时，我们使用一个普通的 `Handle` 。

```haskell
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}

import Control.Monad.Trans (MonadIO (..), MonadTrans (..))
import MonadHandle
import SafeHello
import System.Directory (removeFile)
import System.IO (IOMode (..))
import qualified System.IO as IO

instance MonadHandle IO.Handle IO where
  openFile = IO.openFile
  hPutStr = IO.hPutStr
  hClose = IO.hClose
  hGetContents = IO.hGetContents
  hPutStrLn = IO.hPutStrLn
```

由于任何 `MonadHandle` 也必须同样是一个 `Monad` ，因此可以使用 `do` 来操作文件，而无需关心它最终会在那个单子中执行。

```haskell
safeHello :: (MonadHandle h m) => FilePath -> m ()
safeHello path = do
  h <- openFile path WriteMode
  hPutStrLn h "hello world"
  hClose h
```

因为我们让 `IO` 成为这个 typeclass 的一个实例，所以可以从 **ghci** 执行这个动作：

```
ghci> :l MonadHandleIO.hs
[1 of 2] Compiling MonadHandle      ( MonadHandle.hs, interpreted )
[2 of 2] Compiling MonadHandleIO    ( MonadHandleIO.hs, interpreted )
Ok, two modules loaded.
ghci> safeHello "hello to my fans in domestic surveillance"
ghci> removeFileo "hello to my fans in domestic surveillance"
```

typeclass 方法的美妙之处在于，我们可以在不涉及太多代码的情况下将一个底层单子交换为另一个单子，因为大多数代码都不知道或不关心实现。例如，可以用某个写入文件时压缩文件的单子来代替 `IO` ，

通过 typeclass 定义单子的接口还有一个好处。它允许其他人将我们的实现隐藏在新的类型包装器中，并自动派生他们想要公开的 typeclass 实例。

隔离与测试

实际上，由于 `safeHello` 函数并没有使用 `IO` 类型，我们甚至可以用一个无法执行 I/O 的单子。这允许我们在一个完全纯净、可控的环境中测试通常会产生副作用的代码。

为此我们将创建一个无法执行 I/O 的单子，不过将会记录每个文件相关的事件。

```
data Event
  = Open FilePath IOMode
  | Put String String
  | Close String
  | GetContents String
  deriving (Show)
```

我们在之前的章节中开发了 `Logger` 类型，不过这里将使用标准且更泛用的 `Writer` 单子。正如其它 `mtl` 单子，由 `Writer` 提供的 API 定义在一个 typeclass 中，即 `MonadWriter` 。它最有用的方法是 `tell` ，即记录一个值。

```
ghci> :m +Control.Monad.Writer
ghci> :type tell
tell :: (MonadWriter w m) => w -> m ()
```

我们记录的值可以是任何 `Monoid` 类型。由于列表类型是一个 `Monoid` ，那么记录到 `Event` 列表。

我们可以让 `Writer [Event]` 成为 `MonadHandle` 的一个实例，但它更便宜，更容易，也更安全。

注：书中遗漏了 `MonadHandle` 的 `WriterIO` 实例：

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

import MonadHandle (MonadHandle (..))
import MonadHandleIO

instance MonadHandle FilePath WriterIO where
  openFile path mode = tell [Open path mode] >> return path
  hPutStr h str = tell [Put h str]
  hClose h = tell [Close h]
  hGetContents h = tell [GetContents h] >> return "fake contents"
```

通过 **ghci** 测试：

```
ghci> :l WriterIO.hs
[1 of 3] Compiling MonadHandle      ( MonadHandle.hs, interpreted )
[2 of 3] Compiling MonadHandleIO    ( MonadHandleIO.hs, interpreted )
[3 of 3] Compiling WriterIO         ( WriterIO.hs, interpreted )
Ok, three modules loaded.
ghci> runWriterIO (safeHello "foo")
((),[Open "foo" WriteMode,Put "foo" "hello world",Put "foo" "\n",Close "foo"])
```

writer 单子与列表

每次使用 `tell` 时，writer 单子都会使用单子的 `mappend` 函数，由于列表的 `mappend` 是 `(++)` ，因此列表不是与 `Writer` 一起使用的使用选择：重复追加的代价很高。上面的示例使用列表存粹是为了简单。

在生产代码中，如果使用 `Writer` 单子同时需要一个类似列表这样的方式记录日志，那么请使用更高效的追加特征的容器。这里有一种选择就是之前介绍的差异列表（十三章）。我们无需自己造轮子，而是可以在 Haskell 的库中去寻找一个合适的替代。另一种方案就是使用由 `Data.Sequence` 模块所提供的 `Seq` 类型（十三章）。

任意 I/O 的重新访问

如果使用 typeclass 方式来限制 `IO` ，我们可能仍然希望保留执行任意 I/O 操作的能力。我们可以尝试在 typeclass 上添加 `MonadIO` 作为约束。

```
class (MonadHandle h m, MonadIO m) => MonadHandleIO h m | m -> h

instance MonadHandleIO Handle IO

tidierHello :: (MonadHandleIO h m) => FilePath -> m ()
tidierHello path = do
  safeHello path
  liftIO $ removeFile path
```

不过这种方式存在一个问题：添加的 `MonadIO` 约束使我们失去了在纯环境中测试代码的能力，因为我们不能再判断测试是否可能具有破坏性的副作用。另一种方法是将这个约束从 typeclass（它影响所有函数）移到那些真正需要执行 I/O 的函数。

```
tidyHello :: (MonadIO m, MonadHandle h m) => FilePath -> m ()
tidyHello path = do
  safeHello path
  liftIO $ removeFile path
```

我们可以对缺少 `MonadIO` 约束的函数使用纯函数式测试，而对其余的函数使用传统的单元测试。

不幸的是，我们用一个问题替代了另一个问题：我们不能从单独具有 `MonadHandle` 约束的代码中调用具有 `MonadIO` 和 `MonadHandle` 约束的代码。如果我们发现在 `MonadHandle` 代码深处的某个地方，我们确实需要 `MonadIO` 约束，我们必须将它添加到通向这一点的所有代码路径中。

允许任意 I/O 是有风险的，并且会对我们如何开发和测试代码产生深远的影响。当我们必须在宽容和更容易的推理和测试之间做出选择时，我们通常会选择后者。

16 The Parsec parsing library

WIP

17 The foreign function interface

WIP

18 Monad transformers

动机：避免样板

单子提供了一种强大的方式来构建具有效果的计算。每个标准单子都专门做一件事。在实际代码中，我们经常需要能够一次使用几个效果。

回忆一下第十章开发的 `Parse` 类型。在介绍单子的时候，我们提到了这个类型是一个伪装的状态单子。我们的单子比标准的 `State` 单子更加的复杂，因为它使用了 `Either` 类型来容忍解析错误。这个例子中，如果一个解析提前出现了，我们是希望停止解析而不是继续带着破损的状态。我们的单子结合了携带状态的效果和提前退出的效果。

正常的 `State` 单子不允许我们以这种方式逃离；它仅携带状态。它使用 `fail` 的默认实现：即调用 `error` ，它跑出一个在纯代码中无法捕获的异常。因此，`State` 单子允许失败，而这种能力实际上没有任何用处。（再次声明，我们建议尽量避免使用 `fail` ！）

如果能够以某种方式采用标准的 `State` 单子，并在其中添加错误处理，而不需要大量手工构建自定义单子，那将是最为理想的。`mtl` 库中的标准单子不允许我们组合它们。相反，该库提供了一组*单子转换*来实现相同的结果。

一个单子转换类似于普通的单子，但它不是一个独立的实体：相反，它修改底层单子的行为。`mtl` 库中的大多数单子都有等效的转换。根据惯例，单子转换的版本具有相同的名称，末尾有一个 `T` 。例如，与 `State` 等价的转换就是 `StateT` ；它将可变状态添加到底层单子中。`WriterT` 单子转换使得在堆叠在另一个单子上时写入数据成为可能。

简单单子转换案例

在介绍单子转换之前，看一下用已经熟悉的技术编写的函数。下面的函数递归到一个目录树，并返回它在树的每级找到的条目数列表。

```
1
2  module CountEntries
3    ( listDirectory,
4      countEntriesTrad,
5    )
6  where
7
8  import Control.Monad (forM, liftM)
9  import System.Directory (doesDirectoryExist, getDirectoryContents)
10 import System.FilePath ((</>))
11
12 listDirectory :: FilePath -> IO [FilePath]
13 listDirectory = liftM (filter notDots) . getDirectoryContents
14   where
15     notDots p = p /= "." && p /= ".."
16
```

```haskell
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
```

现在我们来看看如何使用 writer 单子来实现同样的目的。由于这个单子允许我们记录任何需要的值，因此不需要显式的构建结果。

因为我们的函数必须在 `IO` 单子中执行，这样它才能遍历目录，所以不能直接使用 `Writer` 单子。相反，我们使用 `WriterT` 向 `IO` 添加记录功能。

普通的 `Writer` 单子有两个类型参数，因此这里更为合适的写法是 `Writer w a` 。第一个参数 `w` 是要记录的值的类型，而 `a` 则是 `Monad` typeclass 通常所需要的类型。因此 `Writer [(FilePath, Int)] a` 是一个记录目录名称和大小列表的 writer 单子。

`WriterT` 转换拥有类似的结构，不过它添加了另一个类型参数 `m` ：这正是增强其行为的底层单子。`WriterT` 的完整签名为 `WriterT w m a` 。

因为要遍历目录，这需要访问 `IO` 单子，所以将 Writer 堆栈在 `IO` 单子的顶部。单子转换和底层单子的组合将具有 `WriterT [(FilePath, Int)] IO a` 类型。这个单子转换和单子的堆栈本身就是一个单子。

```haskell
module CountEntriesT
  ( countEntries,
  )
where

import Control.Monad
import Control.Monad.Trans
import Control.Monad.Writer
import CountEntries (listDirectory)
import System.Directory (doesDirectoryExist)
import System.FilePath ((</>))

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
```

这段代码与之前的版本没有太大的不同。我们使用 `liftIO` 在必要的地方公开 `IO` 单子，并使用 `tell` 来记录对目录的访问。

要使代码能被运行还必须使用 `WriterT` 的一个执行函数：

```
ghci> :t runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :t execWriterT
execWriterT :: Monad m => WriterT w m a -> m w
```

这些函数执行操作，移除 `WriteT` 包装并给出一个封装在底层单子中的结果。`runWriterT` 函数既提供操作的结果也提供运行时记录的内容，而 `execWriteT` 会丢弃结果，值提供记录的内容。

```
ghci> :l CountEntriesT.hs
[1 of 2] Compiling CountEntries     ( CountEntries.hs, interpreted )
[2 of 2] Compiling CountEntriesT    ( CountEntriesT.hs, interpreted )
Ok, two modules loaded.
ghci> :t countEntries ".."
countEntries ".." :: WriterT [(FilePath, Int)] IO ()
ghci> :t execWriterT (countEntries "..")
execWriterT (countEntries "..") :: IO [(FilePath, Int)]
ghci> take 4 `liftM` execWriterT (countEntries "..")
[("..",6),("../dist-newstyle",2),("../dist-newstyle/cache",8),("../dist-newstyle/tmp",0)]
```

我们在 `IO` 上使用 `WriterT` ，因为没有 `IOT` 单子转换。每当我们使用一个或多个单子转换的 `IO` 单子时，`IO` 总是在堆栈的底部。

单子和单子转换中的常见模式

`mtl` 库中大多数的单子和单子转换都遵循着一些关于名称和 typeclasses 的通用模式。

为了解释这些规则，我们将专注于一个单子：reader 单子。reader 的 API 由 `MonadReader` typeclass 详细说明。大多数 `mtl` 单子都有着类似命名的 typeclasses：`MonadWriter` 定义了 writer 单子，以此类推。

```
class (Monad m) => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a
```

类型变量 `r` 表示 reader 单子所携带的不可变状态。`Reader r` 单子是 `MonadReader` 类的一个实例，即 `ReaderT r m` 单子转换。这个模式同样也被其他 `mtl` 单子所用：通常存在实体单子以及一个单子转换，它们每个都是用来定义单子 API 的 typeclass 实例。

回到 reader 单子，我们尚未接触到 `local` 函数。它通过 `r -> r` 函数临时改变当前环境，同时在修改后的环境中执行其操作。为了更好的理解，以下是一个简单的示例：

```
{-# LANGUAGE FlexibleContexts #-}

```

```haskell
import Control.Monad.Reader

myName :: (MonadReader String m) => String -> m String
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)
```

注：需要添加 `FlexibleContexts` 扩展，否则 `myName` 无法编译。

在 **ghci** 中执行 `localExample` 操作时，我们可以看到改变环境的效果局限于一处：

```
ghci> :l LocalReader.hs
[1 of 2] Compiling Main             ( LocalReader.hs, interpreted )
Ok, one module loaded.
ghci> runReader localExample "Fred"
("First, I am Fred","Second, I am Freddy","Third, I am Fred")
```

当底层单子 `m` 是 `MonadIO` 的一个实例时，`mtl` 库提供了 `ReaderT r m` 的实例，以及其它的一些 typeclasses。例如：

```haskell
instance (Monad m) => Functor (ReaderT r m) where
  ...

instance (MonadIO m) => MonadIO (ReaderT r m) where
  ...

instance (MonadPlus m) => MonadPlus (ReaderT r m) where
  ...
```

再次声明，大多数 `mtl` 单子转换都定义了这样的实例，以便我们更容易的使用它们。

堆叠多个单子转换

正如之前提到的，当我们将单子转换堆叠在普通单子上时，结果便是另一个单子。这表明我们可以再次将单子转换堆叠在合并的单子之上，以提供新的单子，实际上这是一件很常见的事。在什么情况下可能需要创建这样一个堆栈?

1. 如果需要与外界沟通，我们将在堆栈的基础上使用 `IO` ；否则得到的是一些正常的单子。
2. 如果添加了一层 `ReaderT` ，我们将获得对配置信息只读的权限。
3. 添加了一层 `StateT` ，将会得到可控修改的全局状态。

4. 当需要将事件输出到日志时，添加一层 `WriterT` 。

这个方法的强大之处在于，我们可以根据确切需求定制堆栈，指定我们想要支持的效果类型。

下面是一个堆叠单子转换操作的例子，这里是之前开发的 `countEntries` 函数。我们将修改它，使其递归到目录树的深度不超过给定的量，并记录它达到的最大深度。

```
import Control.Monad.Reader
import Control.Monad.State
import System.Directory
import System.FilePath

data AppConfig = AppConfig
  { cfgMaxDepth :: Int
  }
  deriving (Show)

data AppState = AppState
  { stDeepestReached :: Int
  }
  deriving (Show)
```

我们使用 `ReaderT` 来存储配置数据，即将要实现的最大的递归深度。同样使用 `StateT` 来记录递归过程时所达到的最大深度。

```
type App = ReaderT AppConfig (StateT AppState IO)
```

我们的转换堆叠由 `IO` 打底，接着是 `StateT` ，最后是 `ReaderT` 在最上层。这种情况下无论是 `ReaderT` 还是 `WriterT` 在上层都没有区别，但是 `IO` 必须在最底层。

即使是一小堆单子转换也会很快产生一个笨拙的类型名称。我们可以使用类型别名来减少编写类型签名的长度。

Tip

缺失的类型参数去哪里了?

你可能已经注意到了 `type` 别名并没有通常的类型参数 `a` 用于 monadic 类型:

```
type App2 a = ReaderT AppConfig (StateT AppState IO) a

```

`App` 与 `App2` 两者作为普通的类型签名都是能正常工作。差别出现在当我们尝试使用它们去构建另一个类型时：假设我们希望添加另一个单子转换到堆栈上：编译器将允许 `WriterT [String] App a` ，但拒绝 `WriterT [String] App2 a` 。

这是因为 Haskell 并不允许我们偏应用一个类型别名。`App` 的别名并不带有一个类型参数，这就不会造成问题。但是由于 `App2` 接受一个类型参数，那么当我们希望使用 `App2` 来创建

另一个类型时，我们就必须为此提供某些类型用于类型参数。

这个约束仅局限于类型别名。当创建一个单子转换堆栈是，我们通常会通过 `newtype`（将在后面见到）来包装。这样的话，我们就杜绝了此类型的问题。

单子堆栈的执行函数很简单:

```
runApp :: App a -> Int -> IO (a, AppState)
runApp k maxDepth =
  let cfg = AppConfig maxDepth
      stt = AppState 0
   in runStateT (runReaderT k cfg) stt
```

应用中的 `runReaderT` 移除了 `ReaderT` 转换的包装，而 `runStateT` 移除了 `StateT` 的包装，最后将结果放置到 `IO` 单子上。

```
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
```

我们可以在单子堆栈中编写应用程序的大部分命令式代码，类似于我们的 `App` 单子。在实际的程序中，我们会携带更复杂的配置数据，但是我们仍然会使用 `ReaderT` 来使其保持只读和需要时的隐藏。我们将有更多的可变状态需要管理，但我们仍然使用 `StateT` 来封装它。

隐藏工序

我们可以使用 `newtype` 技巧在自定义单子的实现和它的接口之间建立一个坚实的屏障。

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype MyApp a = MyA
  { runA :: ReaderT AppConfig (StateT AppState IO) a
  }
  deriving
    ( Functor,
```

```
      Applicative,
      Monad,
      MonadIO,
      MonadReader AppConfig,
      MonadState AppState
    )

runMyApp :: MyApp a -> Int -> IO (a, AppState)
runMyApp k maxDepth =
  let cfg = AppConfig maxDepth
      stt = AppState 0
  in runStateT (runReaderT (runA k) cfg) stt
```

如果我们从模块中导出 `MyApp` 类型构造函数与 `runMyApp` 执行函数，客户端代码将无法判断单子的内部是一对单子转换。

这里的 `deriving` 需要 `GeneralizedNewtypeDeriving` 语言扩展。它以某种方式让编译器为我们派生出了这些实例。那么这时如何工作的呢?

前面我们提到 `mtl` 库为每个单子转换都提供了许多 typeclass 的实例。例如 `IO` 单子实现了 `MonadIO` 。如果底层单子是 `MonadIO` 的一个实例，`mtl` 也会使 `StateT` 成为一个实例，同理 `ReaderT` 也是这样。

因此这里并没有什么魔法：堆栈中的顶层单子转换是我们用派生子句重新派生的所有 typeclass 的实例。这是 `mtl` 提供了一组 typeclass 和实例的结果，它们很好的结合在了一起。除了可以用 `newtype` 声明执行的通常的自动派生之外，我们无需做其它任何操作。

向下移动堆栈

目前为止，我们对单子转换的使用还很简单，并且 `mtl` 库让我们能够避免构造单子堆栈的细节。实际上我们已经对单子转换有了足够的了解，可以简化很多常见的编程任务。

有一些有用的方法可以让我们脱离 `mtl` 带来的舒适。大多数情况下，自定义单子位于堆栈底层，或者自定义单子转换位于堆栈中某层。为了理解这里潜在的困难，让我们看一个例子。

假设我们有个自定义的单子转换，`CustomT`

```
newtype CustomT m a = ...
```

`mtl` 提供的框架中，堆栈中的每个单子转换通过提供一些列 typeclass 的实例来使较低层级的 API 可用。我们可以遵循这个模式，并编写一些样板实例。

```
instance MonadReader r m => MonadReader r (CustomT m) where
  ...

instance MonadIO r m => MonadIO r (CustomT m) where
  ...
```

如果底层单子是 `MonadReader` 的一个实例，我们将为 `CustomT` 编写一个 `MonadReader` 实例，其中 API 中的每个函数将传递给底层实例中的相应函数。这将允许更高层的代码只关心整个堆栈是 `MonadReader` 的一个实例，而不必知道或关心哪一层提供了真正的实现。

我们可以显式的实现，而不依赖所有这些 typeclass 实例在幕后为我们工作。`MonadTrans` typeclass 定义了一个名为 `lift` 的有用函数。

```
ghci> :m +Control.Monad.Trans
ghci> :i MonadTrans
type MonadTrans :: ((* -> *) -> * -> *) -> Constraint
class (forall (m :: * -> *). Monad m => Monad (t m)) =>
      MonadTrans t where
  lift :: Monad m => m a -> t m a
  {-# MINIMAL lift #-}
        -- Defined in ‘transformers-0.6.1.0:Control.Monad.Trans.Class’
```

该函数从堆栈的一层接受单元操作，并将其转换为当前单子转换中的操作，换言之，将 `lift` 为单子转换中的操作。每个单子转换都是 `MonadTrans` 的一个实例。

基于 `fmap` 和 `liftM` 的用途相似性，我们使用 `lift` 这个名称。在每种情况下，我们都将一些东西从类型系统的较低层级提升到当前工作的层级。

1. `fmap` 将纯函数提升到函子级别；
2. `liftM` 将纯函数提升到单子级别；
3. `lift` 将一个 monadic 操作从低层级的堆栈转换提升到当前层级。

让我们回到之前定义的 `App` 单子堆栈：

```
type App = ReaderT AppConfig (StateT AppState IO)
```

如果想要访问由 `StateT` 携带的 `AppState` ，我们通常需要依靠 `mtl` 所提供的 typeclass 以及实例。

```
implicitGet :: App AppState
implicitGet = get
```

`lift` 函数拥有同样的效果，通过 `get` 将 `StateT` 提升至 `ReaderT` 。

```
explicitGet :: App AppState
explicitGet = lift get
```

显然，当可以使用 `mtl` 时，我们可以拥有更整洁的代码。

当需要显式 lifting 时

我们必须使用 `lift` 的一种情况是，当我们创建一个单子转换堆栈时，同一个 typeclass 的实例出现在多个层级上。

```haskell
type Foo = StateT Int (State String)
```

如果我们尝试使用 `MonadState` typeclass 的 `put` 操作，我们将获得实例 `StateT Int`，因为它位于堆栈的顶层。

```haskell
outerPut :: Int -> Foo ()
outerPut = put
```

这种情况下，访问底层 `State` 单子的 `put` 方法的仅有方式是通过 `lift` 。

```haskell
innerPut :: String -> Foo ()
innerPut = lift . put
```

有时我们需要访问堆栈下面不止一层的单子，这种情况下，我们必须组合调用 `lift` 。每个组合的 `lift` 提供更深一层的访问。

```haskell
type Bar  = ReaderT Bool Foo

barPut :: String -> Bar ()
barPut = lift . lift . put
```

当我们需要使用 `lift` 时，像上面那样编写包装器函数为执行提升并使用这些函数是一种很好的风格。在整个代码中显式的使用 `lift` 往往看起来很混乱。更糟糕的是，它将单子堆栈的布局细节硬连到了代码中，这会是的后续的修改变得复杂。

通过构建一个单子转换来了解它

为了让我们深入了解单子转换的工作原理，我们将创建一个单子转换，并在此过程中描述其机制。我们的目标简单有用。令人惊讶的，它居然没有在 `mtl` 库中：`MaybeT` 。

这个单子转换通过 `Maybe` 包装其类型参数修改底层单子的行为，以给出 `m (Maybe a)` 。与 `Maybe` 单子一样，如果在 `MaybeT` 单子转换中调用 `fail` ，执行将提前终止。

为了将 `m (Maybe a)` 变为一个 `Monad` 实例，我们必须通过 `newtype` 声明将其变为独立类型。

```haskell
newtype MaybeT m a = MaybeT
  { runMaybeT :: m (Maybe a)
  }
```

我们现在需要定义三个标准单子函数。最复杂的就是 `(>>=)` ，它的内部结构最能说明实际在做什么。在深入研究它之前，让我们先看一下它的类型：

```haskell
bindMT :: (Monad m) => MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

为了理解这个类型签名，回忆一下之前的多参数 typeclasses 讨论。我们意图构建的一个 `Monad` 实例是偏类型的 `MaybtT m` ：它带有单类型参数，`a` ，满足 `Monad` typeclass 的要求。

理解 `(>>=)` 实现的技巧是，`do` 块中的所有内容都在底层单子 `m` 中执行，无论它是什么。

```
x `bindMT` f = MaybeT $ do
  unwrapped <- runMaybeT x
  case unwrapped of
    Nothing -> return Nothing
    Just y -> runMaybeT (f y)
```

我们的 `runMaybeT` 函数解包了包含在 `x` 中的结果。接着，`<-` 符号脱糖为 `(>>=)`：一个单子转换的 `(>>=)` 必须使用底层单子的 `(>>=)`。接着则是判断是否要短路或是串联计算。最后看一下函数体的最上方：这里我们必须通过 `MaybeT` 的构造函数来包装结果，为的就是再一次隐藏底层单子。

上述代码的 `do` 声明便于阅读，但是它隐藏了依赖底层单子的 `(>>=)` 实现这个事实。以下是 `MaybeT` 的一个更理想的 `(>>=)` 版本，它令逻辑更为清晰。

```
altBindMT :: (Monad m) => MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
x `altBindMT` f =
  MaybeT $ runMaybeT x >>= maybe (return Nothing) (runMaybeT . f)
```

现在我们理解了 `(>>=)` 是做什么的了，那么 `return` 与 `fail` 的实现则无需解释了，`Monad` 实例亦是如此。

```
instance (Functor f) => Functor (MaybeT f) where
  fmap f x = MaybeT $ fmap f <$> runMaybeT x

instance (Applicative a) => Applicative (MaybeT a) where
  pure = MaybeT . pure . Just
  f <*> x = MaybeT $ fmap (<*>) (runMaybeT f) <*> runMaybeT x

instance (Monad m) => Monad (MaybeT m) where
  (>>=) = bindMT

instance (MonadFail m) => MonadFail (MaybeT m) where
  fail _ = MaybeT $ return Nothing
```

创建一个单子转换

为了将类型转为一个单子转换，我们必须提供 `MonadTrans` 类的实例，这样用户才能访问底层单子：

```
import Control.Monad
import Control.Monad.Trans

instance MonadTrans MaybeT where
  lift m = MaybeT $ Just `liftM` m
```

底层单子从 `a` 的类型参数开始：我们“注入”`Just` 构造函数，这样它就会获得我们需要的类型 `Maybe a` 。然后我们用 `MaybeT` 构造函数隐藏单子。

更多的 typeclass 实例

一旦我们定义了 `MonadTrans` 的实例，我们就可以用它来定义无数其他 `mtl` typeclass 的实例。

```
instance (MonadIO m) => MonadIO (MaybeT m) where
  liftIO = lift . liftIO

instance (MonadState s m) => MonadState s (MaybeT m) where
  get = lift get
  put = lift . put
```

由于有几个 `mtl` typeclasses 使用函数依赖，所有一些实例声明要求我们放宽 GHC 严格的检查规则。(如果我们忘记了这些指令中的任何一个，编译器会在它的错误消息中公司我们需要添加哪些指令。)

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}
```

是显式的使用 `lift` 更好，还是花时间编写这些样板实例更好？这取决于我们期望用单子转换做什么。如果只在一些受限的情况下使用它，我们可以单独为 `MonadTrans` 提供一个实例。这种情况下，多几个实例可能仍然有意义，比如 `MonadIO` 。另一方面，如果单子转换将在整个代码中不同的情况下出现，那么花上数小时来编写这些实例可能是一项很好的投资。

将 Parse 类型替换为单子堆栈

现在我们需要开发一个单子转换用于短路退出，例如如果一个解析中途失败了，可以使用它来退出。这样就可以替换掉之前名为“隐式状态”小节中的自定义单子了。

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module MaybeTParse
( Parse,
  evalParse,
)
where

import Control.Monad.State
import qualified Data.ByteString.Lazy as L
import Data.Int (Int64)
import MaybeT

```

```haskell
data ParseState = ParseState
  { string :: L.ByteString,
    offset :: Int64
  }
  deriving (Show)

newtype Parse a = P
  { runP :: MaybeT (State ParseState) a
  }
  deriving (Functor, Applicative, Monad, MonadState ParseState)

evalParse :: Parse a -> L.ByteString -> Maybe a
evalParse m s = evalState (runMaybeT $ runP m) (ParseState s 0)
```

单子转换的堆栈顺序很重要

从前面使用 `ReaderT` 与 `StateT` 等单子转换的例子中，很容易得出这样的结论：单子转换的堆栈顺序无关紧要。

当我们将 `StateT` 堆叠到 `State` 上时，我们很清楚的就能知道顺序是有影响的。类型 `StateT Int (State String)` 与 `StateT String (State Int)` 可能携带了相同的信息，但是我们并不能替换的使用他们。顺序决定了我们什么时候需要 `lift` 来获取一个或是另一个状态。

下面这个例子更生动的说明了顺序的重要性。假设我们有一个可能失败的计算，且想要记录它失败的情况。

```haskell
{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Writer
import MaybeT

problem :: (MonadWriter [String] m, MonadFail m) => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

下面这些单子堆栈的哪个是我们需要的?

```haskell
{-# OPTIONS_GHC -Wno-orphans #-}

type A = WriterT [String] Maybe

type B = MaybeT (Writer [String])

a :: A ()
a = problem

```

```
instance MonadFail Identity where
  fail _ = Identity $ error "oops"

b :: B ()
b = problem
```

注：由于 `fail` 函数不再属于 `Monad` 的一部分（属于 `MonadFail` ）因此还需要独立为 `Identity` 实现 `MonadFail` 实例，另外还需要表头 `{-## OPTIONS_GHC -Wno-orphans ##-}` 。

ghci:

```
ghci> :l MTComposition.hs
[1 of 3] Compiling MaybeT           ( MaybeT.hs, interpreted )
[2 of 3] Compiling Main             ( MTComposition.hs, interpreted )
Ok, two modules loaded.
ghci> runWriterT a
Nothing
ghci> runWriterT $ runMaybeT b
Identity (Nothing,["this is where i fail"])
```

结果的差异并不令人惊讶：只需要看看执行函数的签名就知道了：

```
ghci> :t runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :t runWriter . runMaybeT
runWriter . runMaybeT
  :: MaybeT (WriterT w Identity) a -> (Maybe a, w)
```

我们的 `WriterT` 叠加 `Maybe` 的堆栈是由 `Maybe` 作为底层单子，因此 `runWriterT` 必须返回一个 `Maybe` 类型。我们的测试用例中，只有在没有实际出错的情况下才能看到日志!

堆叠单子转换类似于组合函数。如果我们改变应用函数的顺序，然后得到不同的结果，这并不会让人惊讶，而单子转换亦是如此。

透视单子和单子转换

现在暂时抛开细节，看看使用单子和单子转换进行编程的优缺点。

对于纯代码的干扰

使用单子最大的实际问题可能是，当我们想用纯代码时，单子的类型构造函数经常会妨碍我们。许多有用的纯函数需要 monadic 对应，只是为了给一些 monadic 类型构造函数附加一个占位符参数 `m` 。

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> import Control.Monad
```

```
ghci> :i filterM
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
        -- Defined in ‘Control.Monad’
```

然而这覆盖范围并不完整：标准库并不总是提供 monadic 版本的纯函数。

略。

超出定义的顺序

我们使用单子的一个主要原因是它允许我们指定效果的顺序。回顾一下早前写的代码：

```
{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Writer
import MaybeT

problem :: (MonadWriter [String] m, MonadFail m) => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

因为我们是在单子中执行的，所以可以保证 `tell` 的效果在 `fail` 之前发生。问题是即使在不一定需要它的时候，我们也得到了这种排序保证：编译器不能自由的重新排列 monadic 代码，即使这样做会提高效率。

运行时开支

最后，当我们使用单子和单子转换时，我们得支付效率税。例如 `State` 单子在闭包中携带其状态。Haskell 中的闭包可能是廉价的，但是它并不是无开销的。

单子转换将自己的开销添加到下面的任何开销中。`MaybeT` 转换必须在每次使用 `(>>=)` 时包装并解包 `Maybe` 值。因此 `StateT` 之上的 `MaybeT` 堆栈对于每个 `(>>=)` 都有大量的工作需要做。

一个足够聪明的编译器可能会使这些成本部分或全部消失，但是这种复杂程度还没有广泛应用。

有一些相对简单的技术可以避免这些成本。例如通过使用延续单子，可以避免在 `(>>=)` 中进行不断的包装和解包，仅仅在使用时才支付开销。这种方法的许多复杂性已经被打包在库中。

笨拙的接口

如果我们将 `mtl` 库当做黑盒使用，那么它的所有组件都可以很好的啮合在一起。然而，当我们开始开发自己的单子和单子转换时，并将它们与 `mtl` 提供的单子转换一起使用时，就会发现一些不足之处。

例如，如果我们创建了一个新的单子转换 `FooT` ，并遵循 `mtl` 相同的模式，那么则需要实现 typeclass `MonadFoo` 。如果还想要集成 `mtl` ，那么还需要提供一大堆 `mtl` typeclasses 的实例。

除此之外，我们还必须为每个 `mtl` 单子转换声明 `MonadFoo` 实例。这些实例中大多数几乎是相同的，写起来相当的枯燥。如果想继续将新的单子转换集成到 `mtl` 框架中，我们必须处理的组件的数量则随着新单子转换的*平方*而增加!

略。

整合一下

在处理副作用和类型时，单子绝不是终点。它们是迄今为止我们所达到的最实际的休息点。

尽管我们在使用它时必须做出妥协，但单子和单子转换仍然提供了一定程度的灵活性和控制，这在命令式语言中是没有先例的。只需几个声明，我们就可以重新连接像分号这样基本的东西，并赋予它新的含义。

19 Error handling

WIP

20 Systems programming

WIP

21 Working with databases

WIP

22 Web client programming

WIP

23 GUI programming

WIP

24 Basic concurrent and parallel programming

定义并发与并行

一个*并发*程序需要在同一时刻执行若干不相关的任务。假设一个游戏服务器：它由多个组件构成，每个组件都要与外界进行复杂的交互。某个组件可能是处理多用户聊天；若干个则是处理用户的输入，并将更新后的状态返回给用户；另外一些则是执行物理计算。

一个并发程序的正确操作并不需要多核，不过多核可以增强性能与响应。

相反的，一个*并行*程序解决单个问题。假设一个金融模型用于预测单个股票下一分钟的价格。如果我们希望将此模型应用在交易所中的每个股票上，例如预计哪只票可以进行买卖，那么通过五百个核进行计算肯定比一个核计算更能快速的得到一个答案。这种情况下，并行程序通常不依靠多核也能正确工作。

另一个用于区分两者差别的方法在于它们如何与外界进行交互。根据定义，一个并发程序会持续不断地进行网络协议，数据库等处理。而一个典型的并行程序则更为集中：它流式的接受数据，处理一段时间（进行少许 I/O），然后流式的输出数据。

很多传统语言会模糊这两者之间的边界，因为它们强制程序员使用相同的方式构建这两种程序。

使用线程的并发编程

作为并发程序的构建块，大多数编程语言都提供了创建多个独立控制线程的方法。Haskell 也不例外，尽管在 Haskell 中使用线程编程看起来与其他语言有些不同。

Haskell 中，一个线程就是一个独立其它线程执行的 `IO` 操作。要创建一个线程，我们需要导入 `Control.Concurrent` 模块并使用 `forkIO` 函数。

```
ghci> :m +Control.Concurrent
ghci> :t forkIO
forkIO :: IO () -> IO ThreadId
ghci> :m +System.Directory
ghci> forkIO (writeFile "xyzzy" "seo craic nua!") >> doesFileExist "xyzzy"
False
```

线程是不确定的

GHC 的运行时组件没有指定执行线程的顺序。因此，在上面的示例中，新线程创建的文件 *xyzzy* 在原始线程检查其存在时可能已经创建，也可能还没有创建。如果我们尝试此示例一次，然后删除 *xyzzy* 并再次尝试，那么第二次可能会得到不同的结果。

隐藏延迟

假设有一个很大的文件需要被压缩并写入磁盘，但是又想要迅速的处理用户的输入，使用户感受到程序的快速响应。可以使用 `forkIO` 在另一个线程中进行文件写入。

```
import Codec.Compression.GZip ( compress )
import Control.Concurrent (forkIO)
import Control.Exception (SomeException, handle)
import qualified Data.ByteString.Lazy as L

main :: IO ()
main = do
  putStrLn "Enter a file to compress>"
  name <- getLine
  handle (print :: SomeException -> IO ()) $ do
    content <- L.readFile name
    _ <- forkIO $ compressFile name content
    return ()
  where
    compressFile path = L.writeFile (path ++ ".gz") . compress
```

注：原文使用的 `readline` 库不再适用，另外 `print` 也需要显式声明类型。

因为这里使用了惰性的 `ByteString` I/O，剩下的只需要在主线程中打开文件。实际的读取则是在另一条线程中进行。

`handle print` 的使用提供了廉价打印错误信息的方法，例如文件名不存在。

线程间的简单通讯

在两个线程之间共享信息的最简单方法是让它们都是用一个变量。在我们的文件压缩例子中，主线程与另一个线程共享文件的名称以及内容。因为 Haskell 数据在默认情况下是不可变的，所以这没有任何的风险：两个线程都不能修改另一个线程所见的文件名或内容。

我们通常需要线程主动地与其他线程进行交互。例如，GHC 并没有为线程提供查看另一个线程是否正在执行、已经完成或是崩溃的方法。不过它提供了一个同步变量类型，即 `MVar` ，允许我们使用它来达到目的。

`MVar` 的作用像是一个单元素的盒子：它要么是满的要么是空的。我们可以将某个东西放入盒子，使其满上，或是拿出使其空置。

```
ghci> import Control.Concurrent
ghci> :t putMVar
putMVar :: MVar a -> a -> IO ()
ghci> :t takeMVar
takeMVar :: MVar a -> IO a
```

如果我们尝试将一个值放入已经满的 `MVar` 中，线程则会进入睡眠直到其它线程将该值取出。同样的，如果尝试从一个空置的 `MVar` 中取值，线程也会进入睡眠直到其它的线程将值放

入。

```
import Control.Concurrent

communicate :: IO ()
communicate = do
  m <- newEmptyMVar
  _ <- forkIO $ do
    v <- takeMVar m
    putStrLn $ "received" ++ show v
  putStrLn "sending"
  putMVar m "wake up"
```

`newEmptyMVar` 函数有一个描述性的名称；要创建一个非空的 `MVar` 则使用 `newMVar` 。

```
ghci> :t newEmptyMVar
newEmptyMVar :: IO (MVar a)
ghci> :t newMVar
newMVar :: a -> IO (MVar a)
```

测试：

```
ghci> :l MVarExample.hs
[1 of 2] Compiling Main             ( MVarExample.hs, interpreted )
Ok, one module loaded.
ghci> communicate
sending
received "wake up"
```

如果你拥有传统语言中并发编程的经验，那么可以将 `MVar` 看做是用有两个目的的助力。

1. 将一个信息从一个线程发送至另一个线程，例如，消息。
2. 为线程间共享的可变数据提供*互斥*。当数据没有被任何线程使用时，将其放入 `MVar` 中，另一个线程将其临时取出进行读取或者修改。

主线程与其它线程的等待

GHC 的运行时系统将程序的原始控制线程与其它线程区别对待。当该线程完成执行，运行时系统认为程序作为一个整体已经完成。如果又其它线程仍在执行，它们则会被终止。

因此，当我们有不能杀死的长时间运行线程时，我们必须做出特殊安排以确保主线程在其他线程完成之前不会完成。下面是一个小库使其易于实现：

```
import Control.Concurrent
import Control.Exception
import Control.Monad
import Data.Map as M

data ThreadStatus
```

```
    = Running
    | Finished -- terminated normally
    | Threw IOException -- changed from `Exception`
    deriving (Eq, Show)

    -- | Create a new thread manager.
    newManager :: IO ThreadManager

    -- | Create a new managed thread.
    forkManaged :: ThreadManager -> IO () -> IO ThreadId

    -- | Immediately return the status of a managed thread.
    getStatus :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)

    -- | Block until a specific managed thread terminates.
    waitFor :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)

    -- | Block until all managed threads terminate.
    waitAll :: ThreadManager -> IO ()
```

我们使用通常的方法来保持 `ThreadManager` 类型的抽象：我们将其包装在 `newtype` 中，并防止用于创建该类型的值。在模块导出中，给出类型构造函数以及构造管理者的 IO 操作，并不导出数据构造函数。

```
module NiceFork
  ( ThreadManager,
    newManager,
    forkManaged,
    getStatus,
    waitFor,
    waitAll,
  )
where
```

对于 `ThreadManager` 的实现，我们维护一个从线程 ID 到线程状态的映射：

```
-- Thread map
newtype ThreadManager
  = Mgr (MVar (M.Map ThreadId (MVar ThreadStatus)))
  deriving (Eq)

-- Create a new thread manager
newManager :: IO ThreadManager
newManager = Mgr <$> newMVar M.empty
```

这里有两个级别的 `MVar` 使用。首先是 `Map` ，通过替换一个新版本允许我们对其进行“修改”。同时还可以确保使用 `Map` 的任何线程看到的是一致的视图。

对于管理的每个线程，我们维护一个 `MVar` 。每个线程的 `MVar` 开始时为空，即表明线程正在执行。当线程结束或被未捕获的异常杀死时，将该信息放入 `MVar` 中。

要创建一个线程并观察其状态，我们必须执行一点记录性的操作。

```
-- Create a new managed thread and watch its status
forkManaged :: ThreadManager -> IO () -> IO ThreadId
forkManaged (Mgr mgr) body =
  -- safely modify an MVar
  modifyMVar mgr $ \m -> do
    state <- newEmptyMVar
    tid <- forkIO $ do
      result <- try body
      putMVar state $ either Threw (const Finished) result
    return (M.insert tid state m, tid)
```

安全的修改 MVar

在上述 `forkManaged` 中使用的 `modifyMVar` 函数非常有用：它是一个安全的 `takeMVar` 与 `putMVar` 组合。

```
ghci> :t modifyMVar
ved "wake up!"
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

它从 `MVar` 中取值，将其传递给一个函数。该函数可以同时生成一个新值并返回一个结果。如果该函数抛出异常，那么 `modifyMVar` 则会将原有值放回 `MVar` 中，否则将新值放入其中。它返回函数的另一个元素作为自己的结果。

当我们使用 `modifyMVar` 而不是通过 `takeMVar` 与 `putMVar` 来手动管理一个 `MVar` 时，我们避免了两种常见的并行错误：

1. 忘记将值放回 `MVar` 。这可能导致*死锁*，在这种情况下，一些线程永远在等待一个永远不会向其放入值的 `MVar` 。
2. 未能考虑到抛出异常的可能，从而中断了代码流程。这可能导致*应该*发生的对 `putVar` 的调用实际上并没有发生，从而再次导致死锁。

因为这些优秀的安全属性，尽可能的使用 `mofigyMVar` 是一个明智的选择。

我们可以采用 `modifyMVar` 遵循的模式，将其应用于许多其它资源管理的情况中。以下是该模式的步骤：

1. 获取资源
2. 将资源传递给函数进行处理
3. 总是释放资源，即使该函数抛出了异常。如果出现这种情况，将异常重新抛出使其能被应用代码捕获。

除了安全之外，这种方法还有另一个好处：它可以使我们的代码更简短且更容易理解。从上面的 `forkManaged` 中可以看出，Haskell 对于匿名函数的轻量级语法使得这种风格的编码在视觉上不那么突兀。

以下是 `modifyMVar` 的定义，我们可以看到这种模式的特定样式：

```
import Control.Concurrent (MVar, putMVar, takeMVar)
import Control.Exception (evaluate, mask, onException)

modifyMVar' :: MVar a -> (a -> IO (a, b)) -> IO b
modifyMVar' m io =
  mask $ \restore -> do
    a <- takeMVar m
    (a', b) <- restore (io a >>= evaluate) `onException` putMVar m a
    putMVar m a'
    return b
```

注：详见此处。

无论是在处理网络连接、数据库句柄还是 C 库管理的数据，它都能轻松的适应用户的特定需求。

查找线程的状态

`getStatus` 函数能告诉我们一个线程的当前状态。如果该线程不再被管理（或者是从来没有被管理过），它返回 `Nothing` 。

```
-- Immediately return the status of a managed thread
getStatus :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)
getStatus (Mgr mgr) tid =
  modifyMVar mgr $ \m ->
    case M.lookup tid m of
      Nothing -> return (m, Nothing)
      Just st -> tryTakeMVar st >>= mst m
  where
    mst m' mm = case mm of
      Nothing -> return (m', Just Running)
      Just sth -> return (M.delete tid m', Just sth)
```

如果线程仍在运行，其返回 `Just Running` 。否则，它表示该线程为何被终结，同时停止管理该线程。

如果 `tryTakeMVar` 函数找到 `MVar` 是空的，它便立刻返回 `Nothing` 而不是阻塞。

```
ghci> :t tryTakeMVar
tryTakeMVar :: MVar a -> IO (Maybe a)
```

否则，它像通常那样从 `MVar` 中提取值。

`waitFor` 函数的行为类似，不过并不是立刻返回，它会阻塞，直到给定的线程在返回之前终止。

```
-- Block until a specific managed thread terminates
waitFor :: ThreadManager -> ThreadId -> IO (Maybe ThreadStatus)
waitFor (Mgr mgr) tid = do
  maybeDone <- modifyMVar mgr $ \m ->
    return $ case M.updateLookupWithKey (\_ _ -> Nothing) tid m of
      (Nothing, _) -> (m, Nothing)
      (done, m') -> (m', done)
  case maybeDone of
    Nothing -> return Nothing
    Just st -> Just <$> takeMVar st
```

它首先提取保存线程状态的 `MVar` （如果存在）。`Map` 类型的 `updateLookupWithKey` 函数很有用：它将查找与修改或删除值结合起来。

```
ghci> :m +Data.Map
ghci> :t updateLookupWithKey
updateLookupWithKey :: (Ord k) =>
                       (k -> a -> Maybe a) -> k -> Map k a -> (Maybe a, Map k a)
```

这种情况下，我们希望总是删除持有线程状态的 `MVar` （如果它存在），这样我们的线程管理器将不再管理线程。如果有要提取的值，我们从 `MVar` 中获取线程的退出状态并返回它。

最后一个有用的函数就是简单的等待所有管理的线程完成，并忽略它们的退出状态：

```
-- Block until all managed threads terminate
waitAll :: ThreadManager -> IO ()
waitAll (Mgr mgr) = modifyMVar mgr es >>= mapM_ takeMVar
  where
    es m = return (M.empty, M.elems m)
```

编写更紧凑的代码

上面 `waitFor` 的定义不太令人满意，因为两个地方执行了类似的代码：在由 `modifyMVar` 调用的函数内部，以及在它的返回值上。

当然，我们可以应用前面遇到的一个函数来消除这种重复。由 `Control.Monad` 模块带来的 `join` 可以解决这个问题。

```
ghci> :m +Control.Monad
ghci> :t join
join :: (Monad m) => m (m a) -> m a
```

这里的技巧就是通过让第一个表达式返回应该从 `modifyMVar` 返回后执行的 IO 操作来摆脱第二个 `case` 表达式。这里使用了 `join` ：

```
waitFor (Mgr mgr) tid =
```

```
  join . modifyMVar mgr $ \m -> return $
    case M.updateLookupWithKey (\_ _ -> Nothing) tid m of
      (Nothing, _) -> (m, return Nothing)
      (Just st, m') -> (m', Just <$> takeMVar st)
```

这是一个有趣的想法：我们可以在纯函数代码中创建 monadic 函数或操作，然后传递它，直到最终可用的单元中结束。一旦我们的开发令其有意义，这可能是一种灵活的编写代码的方式。

通过隧道进行交流

对于线程之间的一次性通信，`MVar` 是非常好的选择。另一种类型 `Chan` 提供单向通信通道。下面是一个简单的例子：

```
import Control.Concurrent (forkIO, newChan, readChan, writeChan)

chanExample :: IO ()
chanExample = do
  ch <- newChan
  _ <- forkIO $ do
    writeChan ch "hello world"
    writeChan ch "now i quit"
  readChan ch >>= print
  readChan ch >>= print
```

如果 `Chan` 是空的，`readChan` 会阻塞知道有值可以读取。`writeChan` 函数永远不会阻塞：它会立刻向 `Chan` 中写入一个新值。

几点重要的须知

MVar 与 Chan 是 non-strict 的

正如大多数 Haskell 容器类型一样，`MVar` 与 `Chan` 都是 non-strict 的：它们都不计算其内容。提到这点并不是因为它是一个问题，而是因为它是一个常见的盲点：人们倾向认为这些类型是 strict 的，也许是因为它们在 `IO` 单子中使用。

对于其它容器类型，错误猜测 `MVar` 或 `Chan` 类型的 strict 性的结果通常是空间或性能泄漏。这里有一个合理的设想。

fork 一个线程，在另一个核执行昂贵计算。

```
import Control.Concurrent

notQuiteRight :: IO ()
notQuiteRight = do
  mv <- newEmptyMVar
  _ <- forkIO $ expensiveComputation_stricter mv
```

```
  -- some other activity
  result <- takeMVar mv
  print result

expensiveComputation_stricter :: MVar [Char] -> IO ()
expensiveComputation_stricter mv = do
  let a = "this is"
      b = "not really"
      c = "all that expensive"
  putMVar mv (a ++ b ++ c)
```

这像是做某事儿，同时将其结果返回给 `MVar` ：

```
expensiveComputation_stricter :: MVar [Char] -> IO ()
expensiveComputation_stricter mv = do
  let a = "this is"
      b = "not really"
      c = "all that expensive"
  putMVar mv (a ++ b ++ c)
```

当我们从父线程的 `MVar` 中获取结果并试图对它做一些事情时，我们的线程开始疯狂的计算，因为我们从未强迫计算发生在另一个线程内！

像往常一样，解决方案很简单，一旦我们知道存在潜在的问题：为 fork 的线程增加 strict 性，以确保计算在原线程发生。strict 最好放在一个地方，以免遗忘。

```
{-# LANGUAGE BangPatterns #-}

import Control.Concurrent (MVar, putMVar, takeMVar)
import Control.Exception (IOException, catch, mask, onException, throw)

modifyMVar_strict :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_strict m io = mask $ \restore -> do
  a <- takeMVar m
  !a' <- restore (io a) `onException` putMVar m a
  putMVar m a'

modifyMVar_strict' :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_strict' m io = mask $ \restore -> do
  a <- takeMVar m
  !a' <-
    restore (io a) `catch` \e ->
      putMVar m a >> throw (e :: IOException)
  putMVar m a'
```

注：原文中 `Control.Exception` 库提供的 `block` 与 `unblock` 已被废弃，现在改用 `mask` 来实现。

上述代码中的 `!` 模式简单易用，但这并不总保证数据总能被计算。

Chan 是无边界的

由于 `writeChan` 总能立即成功，所以使用 `Chan` 有着潜在的风险。如果一个线程的写比另一个线程的读要多，那么这个线程将以一种不受约束的方式增长：未读的消息将随着读取越来越落后而堆积起来。

共享状态仍然很困难

尽管与其它语言相比，Haskell 在线程间共享数据有不同的原语，但它仍然面临相同的基本问题：编写正确的并发程序非常困难。实际上，其它语言中并发编程的一些缺陷同样适用于 Haskell。两个比较著名的问题就是*死锁*与*饥饿*。

死锁

在*死锁*的情况下，两个或多个线程在访问共享资源时永远的卡住。一个典型的多线程死锁问题就是忘记申请锁的顺序。这类型的 bug 太常见了，即：*锁序反转*。虽然 Haskell 不提供锁，但是 `MVar` 类型仍然容易出现顺序反转的问题。下面是一个简单的例子：

```haskell
import Control.Concurrent (MVar, forkIO, modifyMVar_, newMVar, yield)

nestedModification :: (Num a1, Num a2) => MVar a2 -> MVar a1 -> IO ()
nestedModification outer inner = do
  modifyMVar_ outer $ \x -> do
    yield -- force this thread to temporarily yield the CPU
    modifyMVar_ inner $ \y -> return $ y + 1
    return $ x + 1
  putStrLn "done"

main :: IO ()
main = do
  a <- newMVar (1 :: Int)
  b <- newMVar (2 :: Int)
  _ <- forkIO $ nestedModification a b
  _ <- forkIO $ nestedModification b a

  return ()
```

如果将此运行在 **ghci** 中，通常（并不总是）打印不了任何东西，这表明两个线程都被卡住了。

`nestedModification` 函数的问题很容易发现。在第一个线程中，我们取 `MVar a` ，接着取 `b` ；在第二个线程中，我们取 `b` 然后取 `a` 。如果第一个线程成功取到了 `a` ，第二个线程取 `b` ，两个线程都将阻塞：每个线程都试图取另一个线程已经清空了的 `MVar` ，所以两个线程都不会有进展。

各种语言中，解决顺序反转的常用方法是在获取资源时始终遵循一致的顺序。由于这种方法需要手动遵循编码约定，因此在实践中很容易被忽略。

更复杂的是，这些反转问题在实际代码很难被发现。`MVar` 的获取通常分布在不同文件中的若干函数中，这使得肉眼检测变得更加棘手。更糟糕的是，这些问题往往是 *间歇性的*，这让它们变得很难复现，更不用说隔离和修复了。

饥饿

并发软件也很容易出现*饥饿*，即一个线程"霸占"共享资源，阻止另一个线程使用它。很容易想象这是如何发生的：一个线程用执行 100 毫秒的主体调用 `modifyMVar` ，而另一个线程用执行 1 毫秒的主体在同一个 `MVar` 上调用 `modifyMVar` 。在第一个线程将一个值放回 `MVar` 之前，第二个线程无法取得任何进展。

`MVar` 类型的 non-strict 性质可能加剧或导致饥饿问题。如果我们在 `MVar` 中放入一个计算成本很高的东西，然后把它从 `MVar` 中取出，放到一个看起来很轻量的线程中，那么这个线程就会突然变得高计算成本，因为它必须计算这个东西。

还有任何的希望吗?

幸运的是，在这里介绍的用于并发性的 api 并不是结局。Haskell 有一个名为软件事务性内存的功能，使用起来更容易也更安全。我们将会在第二十八中进行讨论。

使用多核 GHC

默认情况下，GHC 生成的程序只使用一个内核，即使我们显式的编写并发代码也是如此。要使用多核，我们必须明确的选择这样做。在生成可执行程序时，我们在链接时做出此选择。

- "非线程"运行库在单个操作系统线程中运行所有的 Haskell 线程。这个运行时对于 `MVar` 中创建线程和传递数据非常高效。
- "线程"运行库使用多个操作系统线程来运行 Haskell 线程。它在创建线程和使用 `MVar` 方面有更多的开销。

如果我们将 `-thread` 选项传递给编译器，它将把我们的程序链接到线程运行时库。在编译库或源文件时不需要使用 `-thread` ，只有在最终生成可执行文件时才需要使用 `-thread` 。

即使我们为程序选择了线程运行时，当我们运行它时，它仍然默认只使用一个内核。我们必须明确的告诉运行时要使用多少内核。

运行时选项

我们可以在程序的命令行上将选项传递给 GHC 的运行时系统。在将控制权交给代码之前，运行时扫描程序的参数，以查找特殊的命令行选项 `+RTS` 。它将之后的所有内容，直到特殊选项 `-RTS` ，解释为运行时系统的选项，而不是我们的程序。它在代码中隐藏了所有这些选项。当我们使用 `System.Environment` 模块的 `getArgs` 函数来获取我们的命令行参数，我们不会在列表中找到任何运行时选项。

线程运行时接受一个选项 `-N` 。它有一个参数，指定 `GHC` 运行时系统应该使用的内核数量。选项解析器是挑剔的：`-N` 和后面的数字之间不能有空格。选项 `-N4` 是可以接受的，而 `-N 4` 则不行。

为 Haskell 寻找有效核术

`GHC.Conc` 模块导出了一个变量，`numCapabilities` ，可以告诉我们运行时系统通过 `-N` RTS 选项分配了多少核数。

```
import GHC.Conc (numCapabilities)
import System.Environment (getArgs)

main :: IO ()
main = do
  args <- getArgs
  putStrLn $ "command line arguments: " <> show args
  putStrLn $ "number of cores: " <> show numCapabilities
```

如果编译并运行上述程序，就可以看到运行时的选项在程序中并不可见，但是可以看到多少核可以用于运行。

```
$ ghc -c NumCapabilities.hs
$ ghc -threaded -o NumCapabilities NumCapabilities.o
$ ./NumCapabilities +RTS -N4 -RTS foo
command line arguments: ["foo"]
number of cores: 4
```

使用合适的运行时

使用哪个运行时的决定并不完全明确。虽然线程运行时可以使用多个内核，但它有一个代价：线程和在它们之间共享数据比使用非线程运行时更昂贵。

在许多现实世界的并发程序中，单个线程将花费大部分时间等待网络请求或响应。这些情况下，如果一个 Haskell 程序为成千上万个并发客户提供服务，那么非线程运行时的较低开销可能会有所帮助。例如，与其让单个服务器程序在四个核上使用线程运行时，不如将服务器设计成可以同时运行它的四个副本，并使用非线程运行时。如此可能会得到更好的性能。

Haskell 中的并行编程

现在我们将重点转向并行编程。对于许多计算成本很高的问题，如果我们可以将解决方案分离，并同时在多个核上进行计算，那么就可以更快的计算出结果。

Normal form 以及 head normal form

熟知的 `seq` 函数将表达式求值为我们所说的头范式（缩写 HNF）。它一旦到达最外层的构造函数（“头部”）就会停止。这与正常范式（NF）不同，在正常范式（NF）中，表达式是完全求值的。

我们还会听到 Haskell 程序员提到的弱头范式（WHNF）。对于正常数据，若头正常范式与头正常范式相同。这种差别只出现在函数上，不过这过于深奥了，在这里不进行讨论。

序列化排序

以下是一个使用分治方法的普通 Haskell 函数，其用于列表排序。

```
1 -- divide-and-conquer
2 sort :: (Ord a) => [a] -> [a]
3 sort (x : xs) = lesser ++ x : greater
4   where
5     lesser = sort [y | y <- xs, y < x]
6     greater = sort [y | y <- xs, y >= x]
7 sort _ = []
```

这个函数的灵感来源于著名的快排算法，它是 Haskell 程序员中的经典：它经常在 Haskell 教程的早期以一行代码的形式出现，用 Haskell 的表达性来戏弄读者。这里，我们将代码分成几行，以便于比较串行与并行的版本。

下面是 `sort` 操作的简单介绍：

1. 它从列表中选择一个元素。这被称为*轴心点 pivot*。任何元素都可以是轴心点；第一种模式是最容易匹配的。
2. 它为小于轴心点的所有元素创建一个子列表，并对它们进行递归排序。
3. 它为大于或等于轴心点的所有元素创建一个子列表，并对它们进行递归排序。
4. 合并两个排序好的子列表。

转换代码为并行代码

并行版本的函数比初版稍微复杂了一些：

```haskell
import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x : xs) = force greater `par` (force lesser `pseq` (lesser ++ x : greater))
  where
    lesser = parSort [y | y <- xs, y < x]
    greater = parSort [y | y <- xs, y >= x]
parSort _ = []
```

我们几乎没有扰乱代码：我们所添加的只是三个函数，`par`，`pseq` 以及 `force`。

`par` 函数由 `Control.Parallel` 模块提供。它的左右与 `seq` 类似：将左参求值为弱头范式，并返回右参。顾名思义，`par` 可以与正在发生的任何其它求值并行的求其左参。

至于 `pseq`，它与 `seq` 类似：先将左边的表达式求值为 WHNF，然后返回右边的表达式。两者之前的区别很微妙，但对于并行程序而言很重要：如果编译器看到先求右参会提高性能，则不会承诺求 `seq` 的左参。这种灵活性对于在一个核上执行的程序而言很好，但是对于在多个核上运行的代码而言就不够强劲了。相反编译器保证 `pseq` 会先求左参，再求右参。

这些对于我们代码的修改而言是值得关注的，因为我们不需要说明所有的事情。

- 使用多少核
- 什么线程用于彼此间的通讯
- 如何在可用核中拆分任务
- 什么数据是线程间共享的，什么是私有的
- 所有部分完成后如何收尾

知道并行计算什么

从并行 Haskell 代码中获得良好性能的关键是找到有意义的工作块来并行执行。Non-strict 的计算会妨碍到这点，这就是在并行排序中使用 `force` 函数的原因。为了更好的解释 `force` 函数的作用，我们先来看一个错误的例子：

```haskell
sillySort :: (Ord a) => [a] -> [a]
sillySort (x : xs) = greater `par` (lesser `pseq` (lesser ++ x : greater))
  where
    lesser = sillySort [y | y <- xs, y < x]
    greater = sillySort [y | y <- xs, y >= x]
sillySort _ = []
```

看一下每个使用 `par` 处的小改变。没有用 `force lesser` 与 `force greater`，而是直接计算 `lesser` 与 `greater`。

请记住，对 WHNF 的求值只计算足以查看其最外层构造函数的表达式。这个错误示例中，我们将每个排序的子列表求值为 WHNF。由于每种情况下最外层的构造函数都只是一个列表构造函数，因此我们实际上只强制对每个排序子列表的第一个元素求值！每个列表的其它所有元素保持未求值。换言之，几乎没有并行的做任何有用的工作：`sillySort` 几乎是完全顺序的。

我们通过 `force` 函数强制整个列表，在返回给构造函数之前进行计算，避免上述情况发生：

```
force :: [a] -> ()
force xs = go xs `pseq` ()
  where
    go :: [a] -> Integer
    go (_ : xs') = go xs'
    go [] = 1
```

注意我们并不在乎列表中的是什么；我们遍历列表至最后，使用 `pseq` 。这里显然没有魔法：只是使用我们对 Haskell 计算模型的理解。又因为我们将在 `par` 或 `pseq` 的左侧使用 `force` ，我们不需要返回一个有意义的值。

当然很多情况下，我们也需要强制对列表中的单个元素求值。下面，我们将讨论这个问题的基于类型的解决方案。

par 做了什么承诺

`par` 函数实际上并没有保证一个表达式与另一个表达式并行求值。相反，它承诺如果这样做“有意义”才会这么做。这种模棱两可的不承诺实际上比总是并行求值表达式的保证更有用。它使运行时系统在遇到 `par` 的使用时可以自由的采取智能行动。

例如，运行时可能会认为表达式太廉价而不值得并行计算，或者它可能会注意到所有内核当前都很忙，因此“触发”一个新的并行计算将导致可运行线程的数量超过可执行的内核数量。

这种松散的规范反过来又影响了我们编写并行代码的方式。由于 `par` 在运行时可能有点智能，我们几乎可以在任何喜欢的地方使用它，假设性能不会因为线程争用繁忙的内核而陷入困境。

允许代码并测量性能

为了测试代码，让我们将 `sort` ，`parSort` 与 `parSort2` 保存到一个名为 `Sorting.hs` 的模块中。我们创建了一个小的驱动程序，我们可以使用它来计时其中一个排序函数的性能。

```

module Main where

import Data.Time.Clock
import qualified Sorting as S
import System.Environment (getArgs)
```

```haskell
import System.Random

testFunction :: Int -> [Int] -> [Int]
testFunction n
  | n == 1 = S.sort
  | n == 2 = S.parSort
  | n == 3 = S.seqSort
  | n == 4 = S.parSort2 100000 -- temporary used input
  | otherwise = error "choose n from 1 to 4"

randomInts :: Int -> StdGen -> [Int]
randomInts k g =
  let result = take k (randoms g)
   in S.force result `seq` result

-- test:
-- cabal run sort-main 3 500000
main :: IO ()
main = do
  args <- getArgs

  case args of
    (sortMethod : count : _) -> do
      input <- randomInts (read count) `fmap` getStdGen
      putStrLn $ "We have " ++ show (length input) ++ " elements to sort."
      start <- getCurrentTime
      let sorted = testFunction (read sortMethod) input
      putStrLn $ "Sorted all " ++ show (length sorted) ++ " elements."
      end <- getCurrentTime
      putStrLn $ show (end `diffUTCTime` start) ++ " elapsed."
    _ -> error "SortMain <sortMethod:Int> <count:Int>"
```

简单起见，我们通过 `testFunction` 变量选择在编译时进行基准测试的排序函数。

我们的程序接受一个可选的命令行参数，即要生成的随机列表的长度。

Non-strict 的计算会把性能测量和分析变成雷区。这里有一些潜在的问题，需要我们避免在我们的驱动程序中出现。

- *当我们认为在看待一件事物时，实际上却要衡量几件事。*Haskell 的默认伪随机数生成器（PRNG）很慢，`random` 函数根据需要生成随机数。

  在记录开始时间之前，我们强制计算列表中的每个元素，并打印列表的长度：这确保我们提前创建了所有需要的随机数。

  如果我们省略这一步，那么将把随机数的生成与并行处理随机数交织在一起。因此，既要测量排序数字的成本，也要测量生成数字的成本（不太明显）。

- *不可见的数据依赖。*当我们生成随机数列表时，简单的打印列表的长度将无法执行足够的

求值。这将计算列表的主体，而不是其元素。实际的随机数在排序比较它们之前不会被计算。

这可能会对性能产生严重的影响。随机数的值取决于列表中前面随机数的值，但是我们在处理器内核中随机分散了列表元素。如果我们在排序之前没有对列表元素进行计算，那么我们将遭受可怕的“乒乓”效应：计算不仅会从一个核跳到另一个核，性能也会受到影响。

试着从上面的 `main` 主体中去掉 `force` 的应用：我们会发现并行代码很容易比非并行代码慢三倍。

- *当我们相信代码正在执行有意义的工作时，基准测试是一种思考*。为了强制进行排序，在记录结束时间之前打印结果列表的长度。如果没有 `putStrLn` 要求列表的长度来打印它，那么排序就根本不会发生。

当我们构建程序时，我们启用了优化和 GHC 的线程运行时。

```
ghc -threaded -O2 -package parallel -package time --make SortMain
Loaded package environment from /Users/jacobxie/.ghc/aarch64-darwin-9.8.2/environments/
  default
[1 of 3] Compiling Sorting          ( Sorting.hs, Sorting.o )
[2 of 3] Compiling Main             ( SortMain.hs, SortMain.o )
[3 of 3] Linking SortMain
```

注：也可以通过 cabal 或 stack 配置，以下是 cabal 设置：

```
executable sort-main
import:           warnings
default-language: Haskell2010
main-is:          SortMain.hs
ghc-options:      -threaded -O2
build-depends:
  , base
  , my-concurrent
  , parallel
  , random
  , time
```

当我们运行程序时，必须告诉 GHC 的运行时需要使用多少内核。开始我们尝试原始排序

```
# 使用 GHC 编译后的可执行文件
./SortMain +RTS -N1 -RTS 1 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
10.959195s elapsed.

# 或者使用 cabal
cabal run sort-main +RTS -N1 -RTS 1 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
10.884864s elapsed.
```

注：与原文不同，`SortMain.hs` 改造后的第一个 arg 为 1 时，为单线程运行；为 2 时，为并行运行。

启用第二个核并不会太影响效率。

```
./SortMain +RTS -N2 -RTS 1 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
11.082317s elapsed.
```

接下来试试 `parSort` ，结果并不理想。

```
cabal run sort-main +RTS -N1 -RTS 2 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
13.192825s elapsed.

cabal run sort-main +RTS -N2 -RTS 2 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
12.038354s elapsed.
```

我们在效率上毫无提高，这可能是由于以下两个因素之一：要么 `par` 本身就很昂贵，要么是我们使用的太多了。为了区分这两种可能性，这里有一个与 `parSort` 相同的排序，使用 `pseq` 而不是 `par` 。

```
seqSort :: (Ord a) => [a] -> [a]
seqSort (x:xs) = lesser `pseq` (greater `pseq` (lesser ++ x:greater))
  where
    lesser = seqSort [y | y <- xs, y < x]
    greater = seqSort [y|y <- xs, y >= x]
seqSort _ = []
```

我们也放弃了使用 `force` ，所以与最初的 `sort` 相比，我们应该只测量使用 `pseq` 的成本。单独的 `pseq` 对性能有什么影响?

```
cabal run sort-main +RTS -N1 -RTS 3 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
12.348779s elapsed.
```

这说明 `par` 与 `pseq` 的成本相似。该怎么样提高性能呢?

性能调优

在 `parSort` 中，执行的 `par` 应用程序数量是要排序的元素数量的两倍。虽然 `par` 很廉价，但是它并不是免费的。当递归的应用 `parSort` 时，最终会对单个列表元素应用 `par` 。在这种细颗粒度下，使用 `par` 的成本超过了任何可能的有用性。为了减少这种影响，我们在超过某个阈值后切换到非并行排序。

```haskell
parSort2 :: (Ord a) => Int -> [a] -> [a]
parSort2 d lst@(x : xs)
  | d <= 0 = sort lst
  | otherwise = force greater `par` (force lesser `pseq` (lesser ++ x : greater))
  where
    lesser = parSort2 d' [y | y <- xs, y < x]
    greater = parSort2 d' [y | y <- xs, y >= x]
    d' = d - 1
parSort2 _ _ = []
```

这里，我们停止递归并在可控深度触发新的并行计算。如果我们知道正在处理的数据大小，那么就可以停止细分并在剩余工作量足够小时切换到非并行代码。

```
ghc -threaded -O2 -package parallel -package time --make SortMain

./SortMain +RTS -N2 -RTS 4 7000000
We have 7000000 elements to sort.
Sorted all 7000000 elements.
7.802378s elapsed.
```

注：`testFunction` 的函数体中 `S.parSort2 100000` 参数并不是最优。

略。

并行策略以及 MapReduce

编程社区中，最受函数式编程启发的著名软件系统之一是 Google 的 MapReduce 基础设施，用于并行处理大量数据。

虽然我们可以创建一个简单的实现而不需耗费太多的努力，但是我们需要抵制这个诱惑。如果考虑解决一类问题而不是单个问题，我们可能会得到更广泛适用的代码。

- 我们的算法很快就会被划分与通信的细节所掩盖。这使得理解代码变得困难，这反过来又使得修改代码变得危险。
- 选择一个“颗粒大小”– 分配到一个核的最小工作单元 – 是很困难的。如果颗粒度太小，内核则要花费大量时间进行记录，因此并行程序很容易变得比串行慢。如果颗粒度过大，可能会导致部分内核因负载不均衡而处于空闲状态。

从计算中分离算法

在并行的 Haskell 代码中，传统语言中通信代码产生的混乱被 `par` 与 `pseq` 所取代。例如，以下的函数操作类似于 `map` ，但在执行时并行的将每个元素求值为弱头范式（WHNF）。

```haskell
import Control.Parallel

parallelMap :: (a -> b) -> [a] -> [b]
```

```
parallelMap f (x : xs) = let r = f x in r `par` r : parallelMap f xs
parallelMap _ _ = []
```

类型 `b` 可能是一个列表，或者其它类型，对 `WHNF` 的求值不会做大量有用的工作。我们希望不必为列表和其他需要特殊处理的类型编写特殊的 `parallelMap` 。

为了解决这个问题，我们将从考虑一个更简单的问题开始：如何强制求值。下面是一个函数，它强制将列表中的每个元素求值为 WHNF。

```
forceList :: [a] -> ()
forceList (x:xs) = x `pseq` forceList xs
forceList _ = ()
```

我们的函数不对列表执行任何计算。（事实上，通过检查它的类型签名，我们可以知道它不能执行任何计算，因为它对列表的元素一无所知。）其唯一目的是确保列表被计算为头范式。应用这个函数唯一有意义的地方是 `seq` 或 `par` 的第一个参数，例如如下所示：

```
stricterMap :: (a -> b) -> [a] -> [b]
stricterMap f xs = forceList xs `seq` map f xs
```

这仍然给我们留下了只对 WHNF 求值的列表元素。我们通过添加一个函数作为参数来解决这个问题，该函数可以强制对元素进行更深入的求值。

```
forceListAndElts :: (a -> ()) -> [a] -> ()
forceListAndElts forceElt (x : xs) = forceElt x `seq` forceListAndElts forceElt xs
forceListAndElts _ _ = ()
```

`Control.Parallel.Strategies` 模块将这个想法概括为我们可以用做库的东西。它引入了计算策略的概念。

```
type Done = ()

type Strategy a = a -> Done
```

求值策略不执行计算；它只是确保一个值在某种程度上被求值。最简单的策略被称为 `r0` ，它什么都不做。

```
r0 :: Strategy a
r0 _ = ()
```

接下来是 `rwhnf` ，它计算弱头范式的一个值。

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

为了计算正常范式的一个值，模块提供了一个 typeclass 名为 `rnf` 的方法。

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

Tip

记住这些名称

如果记不住这些函数和类型的名称，将它们视为同义词即可。`rwhnf` 意为“reduce to weak head normal form”；而 `NFData` 则是“normal form data”；以此类推。

对于基本类型，比如 `Int` ，弱头范式和正常范式是一样的，这就是 `NFData` typeclass 使用 `rwhnf` 作为 `rnf` 的默认实现的原因。对于许多常见类型， `Control.Parallel.Strategies` 模块提供了 `NFData` 实例。

```
instance NFData Char

instance NFData Int

instance (NFData a) => NFData (Maybe a) where
  rnf Nothing = ()
  rnf (Just x) = rnf x

{- ... and so on ... -}
```

从这些实例中，应该可以清楚的看到如何为自己的类型编写 `NFData` 实例。用户的 `rnf` 实现必须处理每个构造函数，并将 `rnf` 应用于构造函数的每个字段。

从策略中分离算法

从这些策略构建块中可以构建更复杂的策略。许多已由 `Control.Parallel.Strategies` 提供。例如 `parList` 并行的对列表的每个元素应用求值策略。

```
parList :: Strategy a -> Strategy [a]
parList _ [] = ()
parList strat (x : xs) = strat x `par` (parList strat xs)
```

该模块使用它来定义并行的 `map` 函数。

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

这就是代码变得有趣的地方。在 `using` 的左侧，有一个普通的 `map` 应用。右侧则是计算策略。`using` 组合子告诉我们如何将策略应用于一个值，允许我们将代码与计算其的求值方式分离。

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

`Control.Parallel.Strategies` 模块提供了许多其他函数，这些函数提供了对求值的精细控制。例如，使用求值策略并行应用 `zipWith` 的 `parZipWith` 。

```haskell
vectorSum' :: (NFData a, Num a) => [a] -> [a] -> [a]
vectorSum' = parZipWith rnf (+)
```

编写一个简单的 MapReduce 定义

对于一个 `mapReduce` 函数我们可以快速的了解到其需要的类型。即一个 *map* 组件，给与它一个通常的类型 `a -> b` 。同时还需要一个 *reduce*；该项为 *fold* 的同义词。与其去编写一个特殊类型的 fold，我们将使用一个更泛化的类型，`[b] -> c` 。这个类型允许我们是有左或右折叠，这样我们可以选择一个符合我们数据的再进行处理。

如果我们将这些类型放在一起，那么完全的类型将类似于：

```haskell
simpleMapReduce ::
  (a -> b) -> -- map function
  ([b] -> c) -> -- reduce function
  [a] -> -- list to map over
  c
```

函数本身非常的简单：

```haskell
simpleMapReduce mapFunc reduceFunc = reduceFunc . map mapFunc
```

MapReduce 与策略

我们 `simpleMapReduce` 的定义太简单了。要使其变得有用，我们希望能指定某些任务可以并行。通过策略可以达到目的，传入一个 `map` 的策略以及 `reduce` 的策略。

```haskell
mapReduce ::
  Strategy b -> -- evaluation strategy for mapping
  (a -> b) -> -- map function
  Strategy c -> -- evaluation strategy for reduction
  ([b] -> c) -> -- reduce function
  [a] -> -- list to map over
  c
```

函数的类型和函数体都需要增大来适应策略参数。

```haskell
mapReduce mapStrat mapFunc reduceStrat reduceFunc input =
  mapResult `pseq` reduceResult
  where
    mapResult = parMap mapStrat mapFunc input
    reduceResult = reduceFunc mapResult `using` reduceStrat
```

合理规划

为了获得良好的性能，我们必须确保每个使用 `par` 的应用都能使用理论的成本。如果是处理一个大文件，通过行切分任务对比任务整体而言只占很小部分的负载。

我们将在之后的小节中开发一种处理大块文件的方式。那么这些大块是由什么构成的呢？一个 web 服务的日志文件应该只包含 ASCII 文本，我们将看到通过惰性的 `ByteString` 所带来的优良性能：该类型是高效的，且在流式处理文件时仅消耗少量内存。

```
module LineChunks
  ( chunkedReadWith,
  )
where

import Control.Exception (bracket, finally)
import Control.Parallel.Strategies (NFData, rdeepseq)
import qualified Data.ByteString.Lazy.Char8 as LB
import Data.Int (Int64)
import GHC.Conc (numCapabilities)
import GHC.IO.Handle (Handle, hClose)

data ChunkSpec = CS
  { chunkOffset :: !Int64,
    chunkLength :: !Int64
  }
  deriving (Eq, Show)

withChunks ::
  (NFData a) =>
  (FilePath -> IO [ChunkSpec]) ->
  ([LB.ByteString] -> a) ->
  FilePath ->
  IO a
withChunks chunkFunc process path = do
  (chunks, handles) <- chunkedRead chunkFunc path
  let r = process chunks
  (rdeepseq r `seq` return r) `finally` mapM_ hClose handles

chunkedReadWith :: (NFData a) => ([LB.ByteString] -> a) -> FilePath -> IO a
chunkedReadWith func path =
  withChunks (lineChunks (numCapabilities * 4)) func path

chunkedRead :: (FilePath -> IO [ChunkSpec]) -> FilePath -> IO ([LB.ByteString], [Handle])
chunkedRead = undefined

lineChunks :: Int -> FilePath -> IO [ChunkSpec]
lineChunks = undefined
```

我们并行的消费每个数据块，通过惰性 I/O 的优势确保可以安全的以流的方式，处理这些数据块。

降低惰性 I/O 的风险

惰性 I/O 会带来一些我们希望避免的众所周知的危害。

- 通过不强制从文件句柄中提取数据的计算进行评估，我们可以无形的使文件句柄打开的时间超过必要的时间。由于操作系统通常会对一次可以打开的文件数量设置一个小的固定限制，如果不解决这个风险，可能会意外的使程序的其它部分缺少文件句柄。
- 如果没有显式关闭文件句柄，垃圾收集器将自动为我们关闭它。可能需要很长时间才能注意到应该关闭。这就造成了与上一条一样的饥饿风险。
- 我们可以通过显式关闭文件句柄来避免饥饿。但是如果太早这么做，惰性计算期望能够从关闭的句柄中提取更多数据，则会导致惰性计算失败。

除了这些众所周知的风险外，我们不能使用单个文件句柄向多线程提供数据。文件句柄有一个单独的“查找指针”来跟踪它应该读取的位置，但是当我们想要读取多个块时，每个块都需要从文件中的不同位置消费数据。

有了上述这些概念，让我们来填写一下惰性 I/O。

```
chunkedRead :: (FilePath -> IO [ChunkSpec]) -> FilePath -> IO ([LB.ByteString], [Handle])
chunkedRead chunkFunc path = do
  chunks <- chunkFunc path
  liftM unzip . forM chunks $ \spec -> do
    h <- openFile path ReadMode
    hSeek h AbsoluteSeek (fromIntegral $ chunkOffset spec)
    chunk <- LB.take (chunkLength spec) `liftM` LB.hGetContents h
    return (chunk, h)
```

我们通过显式的关闭文件句柄来避免饥饿问题。我们允许多个线程一次读取不同的块，方法是为每个线程提供一个不同的文件句柄，所有线程读取同一个文件。

最后一个需要解决的问题是，一个惰性计算有一个文件句柄在其后关闭。我们使用 `rnf` 来强制从 `withChunks` 返回之前完成所有的处理。然后，我们可以显式的关闭文件句柄，因为它们不应该再被读取。如果必须在程序中使用惰性 I/O，通常最好是像这样做成“防火墙”，这样便不会在代码没有预料到的部分发生问题。

高效的寻找行对称的代码块

由于服务器日志文件是面向行的，因此我们需要一种有效的方法将文件分成大块，同时确保每个块以行边界结束。由于块的大小可能有几十兆字节，因此我们不希望扫描块中的所有数据来确定其最终边界的位置。

无论选择固定的块大小还是固定数量的块，我们的方法都是有效的。这里我们选择后者。首先寻找块末尾的大致位置，然后向前扫描，直到找到换行符；然后在换行符之后开始下一块，并重复该过程。

```haskell
lineChunks :: Int -> FilePath -> IO [ChunkSpec]
lineChunks numChunks path = do
  bracket (openFile path ReadMode) hClose $ \h -> do
    totalSize <- fromIntegral `liftM` hFileSize h
    let chunkSize = totalSize `div` fromIntegral numChunks
        findChunks offset = do
          let newOffset = offset + chunkSize
          hSeek h AbsoluteSeek (fromIntegral newOffset)
          let findNewline off = do
                eof <- hIsEOF h
                if eof
                  then return [CS offset (totalSize - offset)]
                  else do
                    bytes <- LB.hGet h 4096
                    case LB.elemIndex '\n' bytes of
                      Just n -> do
                        chunks@(c : _) <- findChunks (off + n + 1)
                        let coff = chunkOffset c
                        return (CS offset (coff - offset) : chunks)
                      Nothing -> findNewline (off + LB.length bytes)
          findNewline newOffset
    findChunks 0
```

最后一个块最终会比之前的块短一点，但在实践中这种差异是微不足道的。

记录行数

下面这个简单的例子说明了如何使用刚刚构建好的脚手架：

```haskell
module Main where

import Control.Monad (forM_)
import Control.Parallel.Strategies (rdeepseq)
import qualified Data.ByteString.Lazy.Char8 as LB
import Data.Int (Int64)
import LineChunks
import MapReduce
import System.Environment (getArgs)

lineCount :: [LB.ByteString] -> Int64
lineCount = mapReduce rdeepseq (LB.count '\n') rdeepseq sum

main :: IO ()
main = do
  args <- getArgs
  forM_ args $ \path -> do
    numLines <- chunkedReadWith lineCount path
    putStrLn $ path <> ": " <> show numLines
```

注：原文的 `rnf` 已由 `rdeepseq` 替代。

如果编译项目时带上 `ghc -O2 --make -threaded` ，在初始运行“预热”文件系统缓存后，它应该可以良好的运行（运行时使用 `+RTS -N2` 双核）。

查找最流行的 URLs

下面的例子中，我们统计每个 URL 被访问了多少次。

```haskell
module Main where

import Control.Monad (forM_)
import Control.Parallel.Strategies (rseq)
import qualified Data.ByteString.Char8 as S
import qualified Data.ByteString.Lazy.Char8 as L
import Data.List (foldl', sortBy)
import qualified Data.Map as M
import LineChunks (chunkedReadWith)
import MapReduce (mapReduce)
import System.Environment (getArgs)
import Text.Regex.PCRE.Light (compile, match)

countURLs :: [L.ByteString] -> M.Map S.ByteString Int
countURLs = mapReduce rseq (foldl' augment M.empty . L.lines) rseq M.unions
  where
    augment m line =
      case match (compile pttn []) (strict line) [] of
        Just (_ : url : _) -> M.insertWith (+) url 1 m
        _ -> m
    strict = S.concat . L.toChunks
    pttn = S.pack "\"(?:GET|POST|HEAD) ([^ ]+) HTTP/"

main :: IO ()
main = do
  args <- getArgs
  forM_ args $ \path -> do
    m <- chunkedReadWith countURLs path
    let mostPopular (_, a) (_, b) = compare b a
    mapM_ print . take 10 . sortBy mostPopular . M.toList $ m
```

略。

总结

给定一个非常合适模型的问题，`MapReduce` 编程模型允许我们在 Haskell 中编写具有良好性能的“任意”并行程序，并只需要很少的额外工作量。我们可以很容易的将这个想法扩展到使用其他数据源，例如文件集合或者通过网络获取的数据。

很多情况下，性能瓶颈将以足够高的速率流式传输数据，以跟上核心的处理能力。例如，如果我们尝试在没有缓存在内存中或从高带宽存储阵列流式传输的文件上使用上述任意一程序，我们将花费大部分时间等待磁盘 I/O，而不会从多核中获得任何好处。

25 Profiling and tuning for performance

WIP

26 Advanced library design: building a Bloom filter

WIP

27 Network programming

WIP

28 Software transactional memory

WIP