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。

这段代码有点啰嗦，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

WIP

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

WIP

10 Code case study: parsing a binary data format

WIP

11 Testing and quality assurance

WIP

12 Barcode recognition

WIP

13 Data structures

WIP

14 Monads

WIP

15 Programming with monads

WIP

16 The Parsec parsing library

WIP

17 The foreign function interface

WIP

18 Monad transformers

WIP

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

WIP

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