

## 第五章：编写 JSON 库

### JSON 简介

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

JSON（JavaScript 对象符号）是一种小型、表示简单、便于存储和发送的语言。它通常用于从 web 服务向基于浏览器的 JavaScript 程序传送数据。JSON 的格式由 [www.json.org](http://www.json.org) 描述，而细节由 [RFC 4627](https://tools.ietf.org/html/rfc4627) 补充。

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

```
"a string"
12345
true
null
```

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

```
[-3.14, true, null, "a string"]
{"numbers": [1,2,3,4,5], "useful": false}
```

### 在 Haskell 中表示 JSON 数据

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

```
-- file: ch05/SimpleJSON.hs
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
            deriving (Eq, Ord, Show)
```

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

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

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

将这些定义载入到 ghci 试试看：

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

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

*Main> JNumber 3.14
JNumber 3.14

*Main> JBool True
JBool True
```

 v: latest ▾

```
*Main> JNull
JNull

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

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

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

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

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

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

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

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

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

再加上一些其他函数，初步完成一些基本功能：

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

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

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

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

isNull v           = v == JNull
```

**truncate** 函数返回浮点数或者有理数的整数部分：

```
Prelude> truncate 5.8
5

Prelude> :module +Data.Ratio

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

## Haskell 模块

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

模块的定义必须放在其它定义之前：

 v: latest ▾

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

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

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

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

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

如果省略掉模块定义中的导出部分，那么所有名字都会被导出：

```
module ExportEverything where
```

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

```
module ExportNothing () where
```

## 编译 Haskell 代码

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

编译一个 Haskell 源码文件可以通过 `ghc` 命令来完成：

```
$ ghc -c SimpleJSON.hs

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

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

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

## 载入模块和生成可执行文件

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

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

module Main (main) where

import SimpleJSON

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

 v: latest ▾

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

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

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

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

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

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

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

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

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

一旦编译完成，就可以运行编译所得的可执行文件了：

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

## 打印 JSON 数据

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

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

```
module PutJSON where

import Data.List (intercalate)
import SimpleJSON

renderJValue :: JValue -> String

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

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

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

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

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

 v: latest ▾

现在打印 JSON 值变得容易得多了：

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

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

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

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

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

## 类型推导是一把双刃剑

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

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

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

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

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

import Data.Char (toUpper)

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

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

现在，有另一个函数调用这个 `upcaseFirst` 函数：

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

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

这段代码在载入 `ghci` 时会发生错误：

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

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

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

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

 v: latest

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

## 更通用的转换方式

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

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

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

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

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

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

## Haskell 开发诀窍

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

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

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

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

data Doc = ToBeDefined
    deriving (Show)

string :: String -> Doc
string str = undefined

text :: String -> Doc
text str = undefined

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

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

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

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

 v: latest ▾

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

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

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

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

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

## 美观打印字符串

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 v: latest ▾



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 v: latest ▾



## 数组和对象

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

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

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

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

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

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

`punctuate` 函数也会在 `Prettify` 中定义。

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

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

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

美观打印对象稍微麻烦一点：对于每个元素，我们还要额外处理名字和值。

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

## 书写模块头

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

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

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

[译注：`compact` 和 `pretty` 函数会在稍后介绍。]

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

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

在每个 `import` 命令中，我们都列出了想要引入我们的模块的命名空间的名字。这并非强制：如果省略这些名字，我们就可以使用一个模块导出的所有名字。然而，通常来讲显式地载入更好。

一个显式列表清楚地表明了我们从哪里载入了哪个名字。如果读者碰到了不熟悉的函数，这便于他们查看文档。

有时候库的维护者会删除或者重命名函数。一个函数很可能在我们写完模块很久之后才从第三方库中消失并导致编译错误。显式列表提醒我们消失的名字是从哪儿载入的，有助于我们更快找到问题。

另外一种情况是库的维护者在模块中加入的函数与我们代码中现有的函数名字一样。如果不用显式列表，这个函数就会在我们的模块中出现两次。当我们用这个函数的时候，GHC 就会报告歧义错误。

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

## 完成美观打印库

在 `Prettify` 模块中，我们用代数数据类型来表示 `Doc` 类型。

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

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

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

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

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

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

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

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

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

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

下面是 `<>` 函数的实现。

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

 v: latest ▾

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

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

### Note

A mathematical moment(to be added)

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

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

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

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

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

`fsep` 的定义依赖于其它几个函数。

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

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

softline :: Doc
softline = group line

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

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

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

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

`flatten` 函数将 `Line` 替换为一个空格，把两行变成一行。

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

 v: latest ▾

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

## 紧凑转换

我们经常希望一段数据占用的字符数越少越好。例如，如果我们想通过网络传输 JSON 数据，就没必要把它弄得很漂亮：另一端的软件并不关心它漂不漂亮，而使布局变漂亮的空格会增加额外开销。

在这种情况下，我们提供一个最基本的紧凑转换函数。

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

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

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

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

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

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

为了更好地理解代码，我们来分析一个更简单的例子。

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

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

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

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

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

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

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

结果是 `"oo" ++ ""`。

结果是 `'f' : "oo" ++ ""`。

## 真正的美观打印

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

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

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

下面是这个函数的核心部分。

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

best _ _ = ""

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

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

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

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

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


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

## 理解美观打印机

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

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

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

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

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

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

由于求值结果为 `True`，`nicest` 的结果是 `" a"`。

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

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

## 练习

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

1. 用下面的类型签名写一个函数 `fill`。

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

它应该给文档添加空格直到指定宽度。如果宽度已经超过指定值，则不加。

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

实现这个功能，缩进量应该可控。

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

## 创建包

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

### 为包添加描述

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

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

放在包描述开头的是一些全局属性，它们适用于包里所有的库和可执行程序。

```
Name:      mypretty
Version:    0.1

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

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

 v: latest ▾

全局属性中的很多信息都是给人而不是 Cabal 自己来读的。

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

如 `Description` 所示，一个字段可以有多行，只要缩进即可。

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

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

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

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

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

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

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

## Note

### 处理依赖关系

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

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

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

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

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

## GHC 的包管理器

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

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

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

 v: latest ▾



## 配置，构建和安装

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

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

我们把这个文件保存为 `Setup.hs`。

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

我们用一个简单的命令告诉 Cabal 如何构建一个包以及往哪里安装这个包。

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

```
$ runghc Setup configure
```

这个命令保证了我们的包可用，并且保存设置让后续的 Cabal 命令使用。

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

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

完成之后，我们来构建这个包。

```
$ runghc Setup build
```

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

```
$ runghc Setup install
```

## 实用链接和扩展阅读

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

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

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

[Hughes95] John Hughes. “**The design of a pretty-printing library**”. May, 1995. First International Spring School on Advanced Functional Programming Techniques. Bastad, Sweden. .

[Wadler98] Philip Wadler. “**A prettier printer**”. March 1998.

## 讨论

0条评论

Real World Haskell 中文版


1 登录

推荐

推文

分享

最新发布



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

来做第一个留言的人吧！

- 在 REAL WORLD HASKELL 中文版 上还有

第十章：代码案例学习：解析二进制数据格式 — Real World Haskell 中文版

1条评论 • 4年前

Y — 此处少个“+”：instance Show Greymap where show (Greymap w h m \_) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m

Real World Haskell 中文版

2条评论 • 6年前

yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地
- 第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

Wengel An —

Real World Haskell 中文版 — Real World Haskell 中文版

4条评论 • 6年前

Jojo — 更新好慢呀