Learn Real World Haskell

Jacob Bishop

2023-08-20

1 Getting started

WIP

2 Types and functions

WIP

3 Defining types, streamlining functions

WIP

4 Functional programming

WIP

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。

```haskell
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：

```haskell
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：

```haskell
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

WIP

7 Input and output

WIP

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