第十章：代码案例学习：解析二进制数据格式

本章将会讨论一个常见任务：解析（parsing）二进制文件。选这个任务有两个目的。 第一个确实是想谈谈解析过程，但更重要的目标是谈谈程序组织、重构和消除样板代码 （boilerplate code：通常指不重要，但没它又不行的代码）。 我们将会展示如何清理冗余代码，并为第十四章讨论 Monad 做点准备。

我们将要用到的文件格式来自于 netpbm 库，它包含一组用来处理位图图像的程序及文件格式，它古老而令人尊敬。 这种文件格式不但被广泛使用，而且还非常简单，虽然解析过程也不是完全没有挑战。 对我们而言最重要的是，netpbm 文件没有经过压缩。

灰度文件

netpbm 的灰度文件格式名为 PGM（"portable grey map"）。事实上它不是一个格式，而是两个： 纯文本（又名P2）格式使用 ASCII 编码，而更常用的原始（P5）格式则采用二进制表示。

每种文件格式都包含头信息，头信息以一个"魔法"字符串开始，指出文件格式。纯文本格式是 `P2`，原始格式是 `P5`。 魔法字符串之后是空格，然后是三个数字：宽度、高度、图像的最大灰度值。 这些数字用十进制 ASCII 数字表示，并用空格隔开。

最大灰度值之后便是图像数据了。在原始文件中，这是一串二进制值。纯文本文件中，这些值是用空格隔开的十进制 ASCII 数字。

原始文件可包含多个图像，一个接一个，每个都有自己的头信息。纯文本文件只包含一个图像。

解析原始 PGM 文件

首先我们来给原始 PGM 文件写解析函数。PGM 解析函数是一个纯函数。它不管获取数据，只管解析。 这是一种常见的 Haskell 编程方法。通过把数据的获取和处理分开，我们可以很方便地控制从哪里获取数据。

我们用 `ByteString` 类型来存储灰度数据，因为它比较节省空间。 由于 PGM 文件以 ASCII 字符串开头，文件内容又是二进制数据，我们同时载入两种形式的 `ByteString` 模块。

```
-- file: ch10/PNM.hs
import qualified Data.ByteString.Lazy.Char8 as L8
import qualified Data.ByteString.Lazy as L
import Data.Char (isSpace)
```

我们并不关心 `ByteString` 类型是惰性的还是严格的，因此我们随便选了惰性的版本。

我们用一个直白的数据类型来表示 PGM 图像。

```
-- file: ch10/PNM.hs
data Greymap = Greymap {
      greyWidth :: Int
    , greyHeight :: Int
    , greyMax :: Int
    , greyData :: L.ByteString
    } deriving (Eq)
```

通常来说，Haskell 的 Show 实例会生成数据的字符串表示，我们还可以用 `read` 读回来。 然而，对于一个位图图像文件来说，这可能会生成一个非常大的字符串，比如当你对一张照片调用 `show` 的时候。 基于这个原因，我们不准备让编译器自动为我们派生 `Show` 实例；我们会自己实现，并刻意简化它。

```
-- file: ch10/PNM.hs
instance Show Greymap where
    show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m
```

我们的 `Show` 实例故意没打印位图数据，也就没必要写 `Read` 实例了，因为我们无法从 `show` 的结果重构 `Greymap`。

解析函数的类型显而易见。

v: latest

```
-- file: ch10/PNM.hs
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
```

这个函数以一个 ByteString 为参数，如果解析成功的话，它返回一个被解析的 Greymap 值以及解析之后剩下的字符串，剩下的字符串以后会用到。

解析函数必须一点一点处理输入数据。首先，我们必须确认我们正在处理的是原始 PGM 文件； 然后，我们处理头信息中的数字；最后我们处理位图数据。下面是是一种比较初级的实现方法，我们会在它的基础上不断改进。

```
-- file: ch10/PNM.hs
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString

-- "nat" here is short for "natural number"
getNat :: L.ByteString -> Maybe (Int, L.ByteString)

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

parseP5 s =
  case matchHeader (L8.pack "P5") s of
    Nothing -> Nothing
    Just s1 ->
      case getNat s1 of
        Nothing -> Nothing
        Just (width, s2) ->
          case getNat (L8.dropWhile isSpace s2) of
            Nothing -> Nothing
            Just (height, s3) ->
              case getNat (L8.dropWhile isSpace s3) of
                Nothing -> Nothing
                Just (maxGrey, s4)
                  | maxGrey > 255 -> Nothing
                  | otherwise ->
                      case getBytes 1 s4 of
                        Nothing -> Nothing
                        Just (_, s5) ->
                          case getBytes (width * height) s5 of
                            Nothing -> Nothing
                            Just (bitmap, s6) ->
                              Just (Greymap width height maxGrey bitmap, s6)
```

这段代码非常直白，它把所有的解析放在了一个长长的梯形 case 表达式中。每个函数在处理完它所需要的部分后会把剩余的 ByteString 返回。 我们再把这部分传给下个函数。像这样我们将结果依次解构，如果解析失败就返回 Nothing，否则便又向最终结迈近了一步。 下面是我们在解析过程中用到的函数的定义。它们的类型被注释掉了因为已经写过了。

```
-- file: ch10/PNM.hs
-- L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader prefix str
    | prefix `L8.isPrefixOf` str
        = Just (L8.dropWhile isSpace (L.drop (L.length prefix) str))
    | otherwise
        = Nothing

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

-- Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes n str = let count           = fromIntegral n
                     both@(prefix,_) = L.splitAt count str
                 in if L.length prefix < count
                    then Nothing
                    else Just both
```

消除样板代码

parseP5 函数虽然能用，但它的代码风格却并不令人满意。它不断挪向屏幕右侧，非常明显，再来个稍微复杂点的函数它就要横跨屏幕了。 我们不断构建和解构 Maybe 值，只在 Just 匹配特定值的时候才继续。所有这些相似的 case 表达式就是“样板代码”，它掩盖了我们真正要做的事情。 总而言之，这段代码需要抽象重构。

退一步看，我们能观察到两种模式。第一，很多我们用到的函数都有相似的类型，它们最后一个参数都是 ByteString，返回值类型都是 Maybe。 第二，parseP5 函数不断解构 Maybe 值，然后要么失败退出，要么把展开之后的值传给下个函数。

我们很容易就能写个函数来体现第二种模式。

```
-- file: ch10/PNM.hs
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v  >>? f = f v
```

(>>?) 函数非常简单：它接受一个值作为左侧参数，一个函数 f 作为右侧参数。如果值不为 Nothing，它就把函数 f 应用在 Just 构造器中的值上。 我们把这个函数定义为操作符这样它就能把别的函数串联在一起了。最后，我们没给 (>>?) 定义结合度，因此它默认为 infixl 9 （左结合，优先级最高的操作符）。 换言之，a >>? b >>? c 会从左向右求值，就像 (a >>? b) >>? c) 一样。

有了这个串联函数，我们来重写一下解析函数。

```
-- file: ch10/PNM.hs
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
    matchHeader (L8.pack "P5") s      >>?
    \s -> skipSpace ((), s)           >>?
    (getNat . snd)                    >>?
    skipSpace                         >>?
    \(width, s) ->   getNat s         >>?
    skipSpace                         >>?
    \(height, s) ->  getNat s         >>?
    \(maxGrey, s) -> getBytes 1 s     >>?
    (getBytes (width * height) . snd) >>?
    \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)

skipSpace :: (a, L.ByteString) -> Maybe (a, L.ByteString)
skipSpace (a, s) = Just (a, L8.dropWhile isSpace s)
```

理解这个函数的关键在于理解其中的链。每个 (>>?) 的左侧都是一个 Maybe 值，右侧都是一个返回 Maybe 值的函数。 这样，Maybe 值就可以不断传给后续 (>>?) 表达式。

我们新增了 skipSpace 函数用来提高可读性。通过这些改进，我们已将代码长度减半。通过移除样板 case 代码，代码变得更容易理解。

尽管在***匿名（lambda）函数***中我们已经警告过不要过度使用匿名函数，在上面的函数链中我们还是用了一些。因为这些函数太小了，给它们命名并不能提高可读性。

隐式状态

到这里还没完。我们的代码显式地用序对传递结果，其中一个元素代表解析结果的中间值，另一个代表剩余的 ByteString 值。 如果我们想扩展代码，比方说记录已经处理过的字节数，以便在解析失败时报告出错位置，那我们已经有8个地方要改了，就为了把序对改成三元组。

这使得本来就没多少的代码还很难修改。问题在于用模式匹配从序对中取值：我们假设了我们总是会用序对，并且把这种假设编进了代码。 尽管模式匹配非常好用，但如果不慎重，我们还是会误入歧途。

让我们解决新代码带来的不便。首先，我们来修改解析状态的类型。

```
-- file: ch10/Parse.hs
data ParseState = ParseState {
      string :: L.ByteString
    , offset :: Int64            -- imported from Data.Int
    } deriving (Show)
```

我们转向了代数数据类型，现在我们既可以记录当前剩余的字符串，也可以记录相对于原字符串的偏移值了。 更重要的改变是用了记录语法：现在可以避免使用模式匹配来获取状态信息了，可以用 `string` 和 `offset` 访问函数。

我们给解析状态起了名字。给东西起名字方便我们推理。例如，我们现在可以这么看解析函数：它处理一个解析状态，产生新解析状态和一些别的信息。 我们可以用 Haskell 类型直接表示。

```
-- file: ch10/Parse.hs
simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

为了给用户更多帮助，我们可以在解析失败时报告一条错误信息。只需对解析器的类型稍作修改即可。

```
-- file: ch10/Parse.hs
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

为了防患于未然，我们最好不要将解析器的实现暴露给用户。早些时候我们显式地用序对来表示状态，当我们想扩展解析器的功能时，我们马上就遇到了麻烦。 为了防止这种现象再次发生，我们用一个 `newtype` 声明来隐藏解析器的细节。

```
--file: ch10/Parse.hs
newtype Parse a = Parse {
    runParse :: ParseState -> Either String (a, ParseState)
    }
```

别忘了 `newtype` 只是函数在编译时的一层包装，它没有运行时开销。我们想用这个函数时，我们用 `runParser` 访问器。

如果我们的模块不导出 `Parse` 值构造器，我们就能确保没人会不小心创建一个解析器，或者通过模式匹配来观察其内部构造。

identity 解析器

我们来定义一个简单的 *identity* 解析器。它把输入值转为解析结果。从这个意义上讲，它有点像 `id` 函数。

```
-- file: ch10/Parse.hs
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

这个函数没动解析状态，只把它的参数当成了解析结果。我们把函数体包装成 `Parse` 类型以通过类型检查。 我们该怎么用它去解析呢？

首先我们得把 `Parse` 包装去掉从而得到里面的函数。这通过 `runParse` 函数实现。 然后得创建一个 `ParseState`，然后对其调用解析函数。最后，我们把解析结果和最终的 `ParseState` 分开。

```
-- file: ch10/Parse.hs
parse :: Parse a -> L.ByteString -> Either String a
parse parser initState
    = case runParse parser (ParseState initState 0) of
        Left err          -> Left err
        Right (result, _) -> Right result
```

由于 `identity` 解析器和 `parse` 函数都没有检查解析状态，我们都不用传入字符串就可以试验我们的代码。

```
Prelude> :r
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, modules loaded: Main.
*Main> :type parse (identity 1) undefined
parse (identity 1) undefined :: Num a => Either String a
*Main> parse (identity 1) undefined
Right 1
*Main> parse (identity "foo") undefined
Right "foo"
```

一个不检查输入的解析器可能有点奇怪，但很快我们就可以看到它的用处。 同时，我们更加确信我们的类型是正确的，基本上解析代码是如何工作的。

记录语法、更新以及模式匹配

记录语法的用处不仅仅在于访问函数：我们可以用它来复制或部分改变已有值。就像下面这样：

```
-- file: ch10/Parse.hs
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset =
    initState { offset = newOffset }
```

这会创建一个跟 `initState` 完全一样的 `ParseState` 值，除了 `offset` 字段会替换成 `newOffset` 指定的值。

```
*Main> let before = ParseState (L8.pack "foo") 0
*Main> let after = modifyOffset before 3
*Main> before
ParseState {string = "foo", offset = 0}
*Main> after
ParseState {string = "foo", offset = 3}
```

在大括号里我们可以给任意多的字段赋值，用逗号分开即可。

一个更有趣的解析器

现在来写个解析器做一些有意义的事情。我们并不好高骛远：我们只想解析单个字节而已。

```
-- file: ch10/Parse.hs
-- import the Word8 type from Data.Word
parseByte :: Parse Word8
parseByte =
    getState ==> \initState ->
    case L.uncons (string initState) of
      Nothing ->
          bail "no more input"
      Just (byte,remainder) ->
          putState newState ==> \_ ->
          identity byte
        where newState = initState { string = remainder,
                                     offset = newOffset }
              newOffset = offset initState + 1
```

定义中有几个新函数。

`L8.uncons` 函数取出 `ByteString` 中的第一个元素。

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

`getState` 函数得到当前解析状态，`putState` 函数更新解析状态。`bail` 函数终止解析并报告错误。`(==>)` 函数把解析器串联起来。我们马上就会详细介绍这些函数。

> **Note**
>
> Hanging lambdas

获取和修改解析状态

`parseByte` 函数并不接受解析状态作为参数。相反，它必须调用 `getState` 来得到解析状态的副本，然后调用 `putState` 将当前状态更新为新状态。


```
-- file: ch10/Parse.hs
getState :: Parse ParseState
```

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

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

阅读这些函数的时候，记得序对左元素为 Parse 结果，右元素为当前 ParseState。这样理解起来会比较容易。

getState 将当前解析状态展开，这样调用者就能访问里面的字符串。 putState 将当前解析状态替换为一个新状态。 (==>) 链中的下一个函数将会使用这个状态。

这些函数将显式的状态处理移到了需要它们的函数的函数体内。很多函数并不关心当前状态是什么，因而它们也不会调用 getState 或 putState。 跟之前需要手动传递元组的解析器相比，现在的代码更加紧凑。在之后的代码中就能看到效果。

我们将解析状态的细节打包放入 ParseState 类型中，然后我们通过访问器而不是模式匹配来访问它。 隐式地传递解析状态给我们带来另外的好处。如果想增加解析状态的信息，我们只需修改 ParseState 定义，以及需要新信息的函数体即可。 跟之前通过模式匹配暴露状态的解析器相比，现在的代码更加模块化：只有需要新信息的代码会受到影响。

报告解析错误

在定义 Parse 的时候我们已经考虑了出错的情况。 (==>) 组合子不断检查解析错误并在错误发生时终止解析。 但我们还没来得及介绍用来报告解析错误的 bail 函数。

```
-- file: ch10/Parse.hs
bail :: String -> Parse a
bail err = Parse $ \s -> Left $
           "byte offset " ++ show (offset s) ++ ": " ++ err
```

调用 bail 之后， (==>) 会模式匹配包装了错误信息的 Left 构造器，并且不会调用下一个解析器。 这使得错误信息可以沿着调用链返回。

把解析器串联起来

(==>) 函数的功能和之前介绍的 (>>?) 函数功能类似：它可以作为“胶水”把函数串联起来。

```
-- file: ch10/Parse.hs
(==>) :: Parse a -> (a -> Parse b) -> Parse b

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

(==>) 函数体很有趣，还稍微有点复杂。回想一下， Parse 类型表示一个被包装的函数。 既然 (==>) 函数把两个 Parse 串联起来并产生第三个，它也必须返回一个被包装的函数。

这个函数做的并不多：它仅仅创建了一个*闭包*（closure）用来记忆 firstParser 和 secondParser 的值。

> **Note**
>
> 闭包是一个函数和它所在的*环境*，也就是它可以访问的变量。闭包在 Haskell 中很常见。 例如， (+5) 就是一个闭包。实现的时候必须将 5 记录为 (+) 操作符的第二个参数，这样得到的函数才能把 5 加给它的参数。

在应用 parse 之前，这个闭包不会被展开应用。 应用的时候，它会求值 firstParser 并检查它的结果。如果解析失败，闭包也会失败。否则，它会把解析结果及 newState 传给 secondParser。

这是非常具有想象力、非常微妙的想法：我们实际上用了一个隐藏的参数将 ParseState 在 Parse 链之间传递。 （我们在之后几章还会碰到这样的代码，所以现在不懂也没有关系。）

Functor 简介

现在我们对 map 函数已经有了一个比较详细的了解，它把函数应用在列表的每一个元素上，并返回一个可能包含另一种类型元素的列表。

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

map 函数这种行为在别的实例中可能有用。例如，考虑一棵二叉树。

```
-- file: ch10/TreeMap.hs
data Tree a = Node (Tree a) (Tree a)
            | Leaf a
              deriving (Show)
```

如果想把一个字符串树转成一个包含这些字符串长度的树，我们可以写个函数来实现：

```
-- file: ch10/TreeMap.hs
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

我们试着寻找一些可能转成通用函数的模式，下面就是一个可能的模式。

```
-- file: ch10/TreeMap.hs
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a)   = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

正如我们希望的那样，treeLengths 和 treeMap length 返回相同的结果。

```
ghci> let tree = Node (Leaf "foo") (Node (Leaf "x") (Leaf "quux"))
ghci> treeLengths tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap length tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap (odd . length) tree
Node (Leaf True) (Node (Leaf True) (Leaf False))
```

Haskell 提供了一个众所周知的类型类来进一步一般化 treeMap。这个类型类叫做 Functor，它只定义了一个函数 fmap。

```
-- file: ch10/TreeMap.hs
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

我们可以把 fmap 当做某种提升函数，就像我们在 Avoiding boilerplate with lifting(fix link) 一节中介绍的那样。 它接受一个参数为普通值 a -> b 的函数并把它提升为一个参数为容器 f a -> f b 的函数。 其中 f 是容器的类型。

举个例子，如果我们用 Tree 替换类型变量 f，fmap 的类型就会跟 treeMap 的类型相同。 事实上我们可以用 treeMap 作为 fmap 对 Tree 的实现。

```
-- file: ch10/TreeMap.hs
instance Functor Tree where
    fmap = treeMap
```

我们可以用 map 作为 fmap 对列表的实现。

```
-- file: ch10/TreeMap.hs
instance Functor [] where
    fmap = map
```

现在我们可以把 fmap 用于不同类型的容器上了。

```
ghci> fmap length ["foo","quux"]
[3,4]
ghci> fmap length (Node (Leaf "Livingstone") (Leaf "I presume"))
Node (Leaf 11) (Leaf 9)
```

Prelude 定义了一些常见类型的 Functor 实例，如列表和 Maybe。

```
-- file: ch10/TreeMap.hs
instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Maybe 的这个实例很清楚地表明了 fmap 要做什么。对于类型的每一个构造器，它都必须给出对应的行为。 例如，如果值被包装在 Just 里，fmap 实现把函数应用在展开之后的值上，然后再用 Just 重新包装起来。

Functor 的定义限制了我们能用 fmap 做什么。例如，如果一个类型有且仅有一个类型参数，我们才能给它实现 Functor 实例。

举个例子，我们不能给 Either a b 或者 (a, b) 写 fmap 实现，因为它们有两个类型参数。 我们也不能给 Bool 或者 Int 写，因为它们没有类型参数。

另外，我们不能给类型定义添加任何约束。这是什么意思呢？为了搞清楚，我们来看一个正常的 data 定义和它的 Functor 实例。

```
-- file: ch10/ValidFunctor.hs
data Foo a = Foo a

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

当我们定义新类型时，我们可以在 data 关键字之后加一个类型约束。

```
-- file: ch10/ValidFunctor.hs
data Eq a => Bar a = Bar a

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

这意味着只有当 a 是 Eq 类型类的成员时，它才能被放进 Foo。 然而，这个约束却让我们无法给 Bar 写 Functor 实例。

```
*Main> :l ValidFunctor.hs
[1 of 1] Compiling Main             ( ValidFunctor.hs, interpreted )

ValidFunctor.hs:8:6:
    Illegal datatype context (use DatatypeContexts): Eq a =>
Failed, modules loaded: none.
```

给类型定义加约束不好

给类型定义加约束从来就不是什么好主意。它的实质效果是强迫你给每一个用到这种类型值的函数加类型约束。 假设我们现在有一个栈数据结构，我们想通过访问它来看看它的元素是否按顺序排列。下面是数据类型的一个简单实现。

```
-- file: ch10/TypeConstraint.hs
data (Ord a) => OrdStack a = Bottom
                           | Item a (OrdStack a)
                             deriving (Show)
```

如果我们想写一个函数来看看它是不是升序的（即每个元素都比它下面的元素大），很显然，我们需要 Ord 约束来进行两两比较。

```
-- file: ch10/TypeConstraint.hs
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
```

```
    | a < b     = isIncreasing rest
    | otherwise = False
isIncreasing _  = True
```

然而，由于我们在类型声明上加了类型约束，它最后也影响到了某些不需要它的地方：我们需要给 `push` 加上 `Ord` 约束，但事实上它并不关心栈里元素的顺序。

```
-- file: ch10/TypeConstraint.hs
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

如果你把 `Ord` 约束删掉，`push` 定义便无法通过类型检查。

正是由于这个原因，我们之前给 `Bar` 写 `Functor` 实例没有成功：它要求 `fmap` 的类型签名加上 `Eq` 约束。

我们现在已经尝试性地确定了 Haskell 里给类型签名加类型约束并不是一个好的特性，那有什么好的替代吗？ 答案很简单：不要在类型定义上加类型约束，在需要它们的函数上加。

在这个例子中，我们可以删掉 `OrdStack` 和 `push` 上的 `Ord`。 `isIncreasing` 还需要，否则便无法调用 `(<)`。现在我们只在需要的地方加类型约束了。 这还有一个更深远的好处：类型签名更准确地表示了每个函数的真正需求。

大多数 Haskell 容器遵循这个模式。`Data.Map` 模块里的 `Map` 类型要求它的键是排序的，但类型本身却没有这个约束。 这个约束是通过 `insert` 这样的函数来表达的，因为这里需要它，在 `size` 上却没有，因为在这里顺序无关紧要。

fmap 的中缀使用

你经常会看到 `fmap` 作为操作符使用：

```
ghci> (1+) `fmap` [1,2,3] ++ [4,5,6]
[2,3,4,4,5,6]
```

也许你感到奇怪，原始的 `map` 却几乎从不这样使用。

我们这样使用 `fmap` 一个可能的原因是可以省略第二个参数的括号。括号越少读代码也就越容易。

```
ghci> fmap (1+) ([1,2,3] ++ [4,5,6])
[2,3,4,5,6,7]
```

如果你真的想把 `fmap` 当做操作符用，`Control.Applicative` 模块包含了作为 `fmap` 别名的 `(<$>)` 操作符。

当我们返回之前写的代码时，我们会发现这对解析很有用。

灵活实例

你可能想给 `Either Int b` 类型实现 `Functor` 实例，因为它只有一个类型参数。

```
-- file: ch10/EitherInt.hs
instance Functor (Either Int) where
    fmap _ (Left n) = Left n
    fmap f (Right r) = Right (f r)
```

然而，Haskell 98 类型系统不能保证检查这种实例的约束会终结。非终结的约束检查会导致编译器进入死循环，所以这种形式的实例是被禁止的。

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

EitherInt.hs:2:10:
    Illegal instance declaration for ‘Functor (Either Int)’
      (All instance types must be of the form (T a1 ... an)
       where a1 ... an are *distinct type variables*,
       and each type variable appears at most once in the instance head.
```

```
      Use FlexibleInstances if you want to disable this.)
    In the instance declaration for ‘Functor (Either Int)’
Failed, modules loaded: none.
```

GHC 的类型系统比 Haskell 98 标准更强大。出于可移植性的考虑，默认情况下，它是运行在兼容 Haskell 98 的模式下的。 我们可以通过一个编译命令允许更灵活的实例。

```
-- file: ch10/EitherIntFlexible.hs
{-# LANGUAGE FlexibleInstances #-}

instance Functor (Either Int) where
    fmap _ (Left n)  = Left n
    fmap f (Right r) = Right (f r)
```

这个命令内嵌于 `LANGUAGE` 编译选项。

有了 `Functor` 实例，我们来试试 `Either Int` 的 `fmap` 函数。

```
ghci> :load EitherIntFlexible
[1 of 1] Compiling Main             ( EitherIntFlexible.hs, interpreted )
Ok, modules loaded: Main.
ghci> fmap (== "cheeseburger") (Left 1 :: Either Int String)
Left 1
ghci> fmap (== "cheeseburger") (Right "fries" :: Either Int String)
Right False
```

更多关于 Functor 的思考

对于 Functor 如何工作，我们做了一些隐式的假设。 把它们说清楚并当成规则去遵守非常有用，因为这会让我们把 Functor 当成统一的、行为规范的对象。 规则只有两个，并且非常简单。

第一条规则是 Functor *必须保持身份*（preserve identity）。也就是说，应用 `fmap id` 应该返回相同的值。

```
ghci> fmap id (Node (Leaf "a") (Leaf "b"))
Node (Leaf "a") (Leaf "b")
```

第二条规则是 Functor *必须是可组合的*。 也就是说，把两个 `fmap` 组合使用效果应该和把函数组合起来再用 `fmap` 相同。

```
ghci> (fmap even . fmap length) (Just "twelve")
Just True
ghci> fmap (even . length) (Just "twelve")
Just True
```

另一种看待这两条规则的方式是 Functor *必须保持结构*（shape）。集合的结构不应该受到 Functor 的影响，只有对应的值会改变。

```
ghci> fmap odd (Just 1)
Just True
ghci> fmap odd Nothing
Nothing
```

如果你要写 `Functor` 实例，最好把这些规则记在脑子里，并且最好测试一下，因为编译器不会检查我们提到的规则。 另一方面，如果你只是用 Functor，这些规则又如此自然，根本没必要记住。 它们只是把一些“照我说的做”的直觉概念形式化了。下面是期望行为的伪代码表示。

```
-- file: ch10/FunctorLaws.hs
fmap id       ==  id
fmap (f . g)  ==  fmap f . fmap g
```

给 Parse 写一个 Functor 实例

对于到目前为止我们研究过的类型而言，`fmap` 的期望行为非常明显。然而由于 `Parse` 的复杂度，对于它而言 `fmap` 的期望行为并没有这么明显。 一个合理的猜测是我们要 `fmap` 的函数应该应用到当前解析的结果上，并保持解析状态不变。

```
-- file: ch10/Parse.hs
instance Functor Parse where
    fmap f parser = parser ==> \result ->
                    identity (f result)
```

定义很容易理解，我们来快速做几个实验看看我们是否遵守了 Functor 规则。

首先我们检查身份是否被保持。我们在一次应该失败的解析上试试：从空字符串中解析字节（别忘了 `(<$>)` 就是 `fmap`）。

```
ghci> parse parseByte L.empty
Left "byte offset 0: no more input"
ghci> parse (id <$> parseByte) L.empty
Left "byte offset 0: no more input"
```

不错。再来试试应该成功的解析。

```
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
```

通过观察上面的结果，可以看到我们的 Functor 实例同样遵守了第二条规则，也就是保持结构。 失败被保持为失败，成功被保持为成功。

最后，我们确保可组合性被保持了。

```
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
```

通过这个简单的观察，我们的 Functor 实例看起来行为规范。

利用 Functor 解析

我们讨论 Functor 是有目的的：它让我们写出简洁、表达能力强的代码。回想早先引入的 `parseByte` 函数。 在重构 PGM 解析器使之使用新的解析架构的过程中，我们经常想用 ASCII 字符而不是 `Word8` 值。

尽管可以写一个类似于 `parseByte` 的 `parseChar` 函数，我们现在可以利用 `Parse` 的 Functor 属性来避免重复代码。 我们的 functor 接受一个解析结果并将一个函数应用于它，因此我们需要的是一个把 `Word8` 转成 `Char` 的函数。

```
-- file: ch10/Parse.hs
w2c :: Word8 -> Char
w2c = chr . fromIntegral

-- import Control.Applicative
parseChar :: Parse Char
parseChar = w2c <$> parseByte
```

我们也可以利用 Functor 来写一个短小的“窥视”函数。如果我们在输入字符串的末尾，它会返回 `Nothing`。 否则，它返回下一个字符，但不作处理（也就是说，它观察但不打扰当前的解析状态）。

```
-- file: ch10/Parse.hs
peekByte :: Parse (Maybe Word8)
peekByte = (fmap fst . L.uncons . string) <$> getState
```

定义 parseChar 时用到的提升把戏同样也可以用于定义 peekChar。

```
-- file: ch10/Parse.hs
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

注意到 peekByte 和 peekChar 分别两次调用了 fmap，其中一次还是 (<$>)。 这么做的原因是 Parse (Maybe a) 类型是嵌在 Functor 中的 Functor。 我们必须提升函数两次使它能进入内部 Functor。

最后，我们会写一个通用组合子，它是 Parse 中的 takeWhile：它在谓词为 True 是处理输入。

```
-- file: ch10/Parse.hs
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p = (fmap p <$> peekByte) ==> \mp ->
               if mp == Just True
               then parseByte ==> \b ->
                    (b:) <$> parseWhile p
               else identity []
```

再次说明，我们在好几个地方都用到了 Functor（doubled up, when necessary）用以化简函数。 下面是相同函数不用 Functor 的版本。

```
-- file: ch10/Parse.hs
parseWhileVerbose p =
    peekByte ==> \mc ->
    case mc of
      Nothing -> identity []
      Just c | p c ->
                 parseByte ==> \b ->
                 parseWhileVerbose p ==> \bs ->
                 identity (b:bs)
             | otherwise ->
                 identity []
```

当你对 Functor 不熟悉的时候，冗余的定义应该会更好读。 但是，由于 Haskell 中 Functor 非常常见，你很快就会更习惯（包括读和写）简洁的表达。

重构 PGM 解析器

有了新的解析代码，原始 PGM 解析函数现在变成了这个样子：

```
-- file: ch10/Parse.hs
parseRawPGM =
    parseWhileWith w2c notWhite ==> \header -> skipSpaces ==>&
    assert (header == "P5") "invalid raw header" ==>&
    parseNat ==> \width -> skipSpaces ==>&
    parseNat ==> \height -> skipSpaces ==>&
    parseNat ==> \maxGrey ->
    parseByte ==>&
    parseBytes (width * height) ==> \bitmap ->
    identity (Greymap width height maxGrey bitmap)
  where notWhite = (`notElem` " \r\n\t")
```

下面是定义中用到的辅助函数，其中一些模式现在应该已经非常熟悉了：

```
-- file: ch10/Parse.hs
parseWhileWith :: (Word8 -> a) -> (a -> Bool) -> Parse [a]
parseWhileWith f p = fmap f <$> parseWhile (p . f)

parseNat :: Parse Int
parseNat = parseWhileWith w2c isDigit ==> \digits ->
           if null digits
           then bail "no more input"
           else let n = read digits
                in if n < 0
```

```
                then bail "integer overflow"
                else identity n

(==>&) :: Parse a -> Parse b -> Parse b
p ==>& f = p ==> \_ -> f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True  _   = identity ()
assert False err = bail err
```

类似于 `(==>)`，`(==>&)` 组合子将解析器串联起来。但右侧忽略左侧的结果。 `assert` 使得我们可以检查性质，然后当性质为 `False` 时终止解析并报告错误信息。

未来方向

本章的主题是抽象。我们发现在函数链中传递显式状态并不理想，因此我们把这个细节抽象出来。 在写解析器的时候发现要重复用到一些代码，我们把它们抽象成函数。 我们引入了 Functor，它提供了一种映射到参数化类型的通用方法。

关于解析，我们在第16章会讨论一个使用广泛并且灵活的解析库 Parsec。 在第14章中，我们会再次讨论抽象，我们会发现用 Monad 可以进一步化简这章的代码。

Hackage 数据库中存在不少包可以用来高效解析以 `ByteString` 表示的二进制数据。在写作时，最流行的是 `binary`，它易用且高效。

练习

1. 给“纯文本” PGM 文件写解析器。
2. 在对“原始” PGM 文件的描述中，我们省略了一个细节。如果头信息中的“最大灰度”值小于256，那每个像素都会用单个字节表示。然而，它的最大范围可达65535，这种情况下每个像素会以大端序的形式（最高有效位字节在前）用两个字节来表示。

   重写原始 PGM 解析器使它能够处理单字节和双字节形式。
3. 重写解析器使得它能够区分“原始”和“纯文本” PGM 文件，并解析对应的文件类型。

讨论