

## 第十八章：Monad变换器

### 动机：避免样板代码

Monad提供了一种强大途径以构建带效果的计算。虽然各个标准monad皆专一于其特定的任务，但在实际代码中，我们常常想同时使用多种效果。

比如，回忆在第十章中开发的 `Parse` 类型。在介绍monad之时，我们提到这个类型其实是乔装过的 `State monad`。事实上我们的monad比标准的 `State monad` 更加复杂：它同时也使用了 `Either` 类型来表达解析过程中可能的失败。在这个例子中，我们想在解析失败的时候就立刻停止这个过程，而不是以错误的状态继续执行解析。这个monad同时包含了带状态计算的效果和提早退出计算的效果。

普通的 `State monad`不允许我们提早退出，因为其只负责状态的携带。其使用的是 `fail` 函数的默认实现：直接调用 `error` 抛出异常 - 这一异常无法在纯函数式的代码中捕获。因此，尽管 `State monad`似乎允许错误，但是这一能力并没有什么用。（再次强调：请尽量避免使用 `fail` 函数！）

理想情况下，我们希望能使用标准的 `State monad`，并为其加上实用的错误处理能力以代替手动地大量定制各种monad。虽然在 `mtl` 库中的标准monad不可合并使用，但使用库中提供了一系列的 *monad变换器* 可以达到相同的效果。

Monad变换器和常规的monad很类似，但它们并不是独立的实体。相反，monad变换器通过修改其以为基础的monad的行为来工作。大部分 `mtl` 库中的monad都有对应的变换器。习惯上变换器以其等价的monad名为基础，加以 `T` 结尾。例如，与 `State` 等价的变换器版本称作 `StateT`；它修改下层monad以增加可变状态。此外，若将 `WriterT` monad变换器叠加于其他（或许不支持数据输出的）monad之上，在被monad修改后的monad中，输出数据将成为可能。

[注：`mtl` 意为monad变换器函数库(Monad Transformer Library)]

[译注：Monad变换器需要依附在一已有monad上来构成新的monad，在接下来的行文中将使用“下层monad”来称呼monad变换器所依附的那个monad]

### 简单的Monad变换器实例

在介绍monad变换器之前，先看看以下函数，其中使用的都是之前接触过的技术。这个函数递归地访问目录树，并返回一个列表，列表中包含树的每层的实体个数：


```
-- file: ch18/CountEntries.hs
module CountEntries
  ( listDirectory
  , countEntriesTrad
  ) where

import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
import Control.Monad (forM, liftM)

listDirectory :: FilePath -> IO [String]
listDirectory = liftM (filter notDots) . getDirectoryContents
  where notDots p = p /= "." && p /= ".."

countEntriesTrad :: FilePath -> IO [(FilePath, Int)]
countEntriesTrad path = do
  contents <- listDirectory path
  rest <- forM contents $ \name -> do
    let newName = path </> name
    isDir <- doesDirectoryExist newName
    if isDir
      then countEntriesTrad newName
      else return []
  return $ (path, length contents) : concat rest
```

现在看看如何使用 `Writer` monad 实现相同的目标。由于这个monad允许随时记下数值，所以并不需要我们显示地去构建结果。

为了遍历目录，这个函数必须在 `IO monad`中执行，因此我们无法直接使用 `Writer monad`。但我们可以用 `WriterT` 将记录  予 `IO`。一种简单的理解方法是首先理解涉及的类型。

通常 `Writer monad` 有两个类型参数，因此写作 `Writer w a` 更为恰当。其中参数 `w` 用以指明我们想要记录的数值的类型。而另一类型参数 `a` 是 `monad` 类型类所要求的。因此 `Writer [(FilePath, Int)] a` 是个记录一列目录名和目录大小的 `writer monad`。

`WriterT` 变换器有着类似的结构。但其增加了另外一个类型参数 `m`：这便是下层 `monad`，也是我们想为其增加功能的 `monad`。`WriterT` 的完整类型签名是 `WriterT w m a`。

由于所需的目录遍历操作需要访问 `IO monad`，因此我们将 `writer` 功能累加在 `IO monad` 之上。通过将 `monad` 变换器与原有 `monad` 结合，我们得到了类型签名：`WriterT [(FilePath, Int)] IO a` 这个 `monad` 变换器和 `monad` 的组合自身也是一个 `monad`：

```
-- file: ch18/CountEntriesT.hs
module CountEntriesT
  ( listDirectory
  , countEntries
  ) where

import CountEntries (listDirectory)
import System.Directory (doesDirectoryExist)
import System.FilePath ((</>))
import Control.Monad (forM_, when)
import Control.Monad.Trans (liftIO)
import Control.Monad.Writer (WriterT, tell)

countEntries :: FilePath -> WriterT [(FilePath, Int)] IO ()
countEntries path = do
  contents <- liftIO . listDirectory $ path
  tell [(path, length contents)]
  forM_ contents $ \name -> do
    let newName = path </> name
    isDir <- liftIO . doesDirectoryExist $ newName
    when isDir $ countEntries newName
```

代码与其先前的版本区别不大，需要时 `liftIO` 可以将 `IO monad` 暴露出来；同时，`tell` 可以用以记下对目录的访问。

为了执行这一代码，需要选择一个 `WriterT` 的执行函数：

```
ghci> :type runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :type execWriterT
execWriterT :: Monad m => WriterT w m a -> m w
```

这些函数都可以用以执行动作，移除 `WriterT` 的包装，并将结果交给其下层 `monad`。其中 `runWriterT` 函数同时返回动作结果以及在执行过程获得的记录。而 `execWriterT` 丢弃动作的结果，只将记录返回。

因为没有 `IO T` 这样的 `monad` 变换器，所以此处我们在 `IO` 之上使用 `WriterT`。一旦要用 `IO monad` 和其他的一个或多个 `monad` 变换器结合，`IO` 一定在 `monad` 栈的最底下。

[译注：“monad栈”由 `monad` 和一个或多个 `monad` 变换器叠加而成，形成一个栈的结构。若在 `monad` 栈中需要 `IO monad`，由于没有对应的 `monad` 变换器（`IO T`），所以 `IO monad` 只能位于整个 `monad` 栈的最底下。此外，`IO` 是一个很特殊的 `monad`，它的 `IO T` 版本是无法实现的。]

## Monad和Monad变换器中的模式

在 `mtl` 库中的大部分 `monad` 与 `monad` 变换器遵从一些关于命名和类型类的模式。

为说明这些规则，我们将注意力聚焦在一个简单的 `monad` 上：`reader monad`。`reader monad` 的具体 API 位于 `MonadReader` 中。大部分 `mtl` 中的 `monad` 都有一个名称相对的类型类。例如 `MonadWriter` 定义了 `writer monad` 的 API，以此类推。

```
-- file: ch18/Reader.hs
{-# LANGUAGE FunctionalDependencies #-}
class Monad m => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

 v: latest ▾

其中类型变量 `r` 表示reader monad所附带的不变状态，`Reader r monad`是个 `MonadReader` 的实例，同时 `ReaderT r m monad`变换器也是一个。这个模式同样也在其他的 `mtl monad`中重复着：通常有个具体的monad，和它对应的monad变换器，而它们都是相应命令的类型类的实例。这个类型类定义了功能相同的monad的API。

回到我们reader monad的例子中，我们之前尚未讨论过 `local` 函数。通过一个类型为 `r -> r` 的函数，它可临时修改当前的环境，并在这一临时环境中执行其动作。举个具体的例子：

```
-- file: ch18/LocalReader.hs
import Control.Monad.Reader

myName step = do
  name <- ask
  return (step ++ ", I am " ++ name)

localExample :: Reader String (String, String, String)
localExample = do
  a <- myName "First"
  b <- local (++"dy") (myName "Second")
  c <- myName "Third"
  return (a, b, c)
```

若在 `ghci` 中执行 `localExample`，可以观察到对环境修改的效果被限制在了一个地方：

```
ghci> runReader localExample "Fred"
Loading package mtl-1.1.0.1 ... linking ... done.
("First, I am Fred", "Second, I am Freddy", "Third, I am Fred")
```

当下层monad `m` 是一个 `MonadIO` 的实例时，`mtl` 提供了关于 `ReaderT r m` 和其他类型类的实例，这里是其中的一些：

```
-- file: ch18/Reader.hs
instance (Monad m) => Functor (ReaderT r m) where
  ...

instance (MonadIO m) => MonadIO (ReaderT r m) where
  ...

instance (MonadPlus m) => MonadPlus (ReaderT r m) where
  ...
```

再次说明：为方便使用，大部分的 `mtl monad`变换器都定义了诸如此类的实例。

## 叠加多个Monad变换器

之前提到过，在常规monad上叠加monad变换器可得到另一个monad。由于混合的结果也是个monad，我们可以凭此为基础再叠加上一层monad变换器。事实上，这么做十分常见。但在什么情况下才需要创建这样的monad呢？

若代码想和外界打交道，便需要 `IO` 作为这个monad栈的基础。否则普通的monad便可以满足需求。  
 加上一层 `ReaderT`，以添加访问只读配置信息的能力。  
 叠加上 `StateT`，就可以添加可修改的全局状态。  
 若想得到记录事件的能力，可以添加一层 `WriterT`。

这个做法的强大之处在于：我们可以指定所需的计算效果，以量身定制monad栈。

举个多重叠加的moand变换器的例子，这里是之前开发的 `countEntries` 函数。我们想限制其递归的深度，并记录下它在执行过程中所到达的最大深度：

```
-- file: ch18/UglyStack.hs
import System.Directory
import System.FilePath
import System.Monad.Reader
import System.Monad.State

data AppConfig = AppConfig
  { cfgMaxDepth :: Int
```

 v: latest ▾

```

    } deriving (Show)

data AppState = AppState
  { stDeepestReached :: Int
  } deriving (Show)

```

此处使用 `ReaderT` 来记录配置数据，数据的内容表示最大允许的递归深度。同时也使用了 `StateT` 来记录在实际遍历过程中所达到的最大深度。

```

-- file: ch18/UglyStack.hs
type App = ReaderT AppConfig (StateT AppState IO)

```

我们的变换器以 `IO` 为基础，依次叠加 `StateT` 与 `ReaderT`。在此例中，栈顶是 `ReaderT` 还是 `WriterT` 并不重要，但是 `IO` 必须作为最下层 monad。

仅仅几个 monad 变换器的叠加，也会使类型签名迅速变得复杂起来。故此处以 `type` 关键字定义类型别名，以简化类型的书写。

### 缺失的类型参数呢？

或许你已注意到，此处的类型别名并没有我们为 monad 类型所常添加的类型参数 `a`：

```

-- file: ch18/UglyStack.hs
type App2 a = ReaderT AppConfig (StateT AppState IO) a

```

在常规的类型签名用例下，`App` 和 `App2` 不会遇到问题。但如果想以此类型为基础构建其他类型，两者的区别就显现出来了。

例如我们想另加一层 monad 变换器，编译器会允许 `WriterT [String] App a` 但拒绝 `WriterT [String] App2 a`。

其中的理由是：Haskell 不允许对类型别名的部分应用。`App` 不需要类型参数，故没有问题。另一方面，因为 `App2` 需要一个类型参数，若想基于 `App2` 构造其他的类型，则必须为这个类型参数提供一个类型。

这一限制仅适用于类型别名，当构建 monad 栈时，通常的做法是用 `newtype` 来封装（接下来的部分就会看到这类例子）。因此实际应用中很少出现这种问题。

[译注：类似于函数的部分应用，“类型别名的部分应用”指的是在应用类型别名时，给出的参数数量少于定义中的参数数量。在以上例子中，`App` 是一个完整的应用，因为在其定义 `type App = ...` 中，没有类型参数；而 `App2` 却是个部分应用，因为在其定义 `type App2 a = ...` 中，还需要一个类型参数 `a`。]

我们 monad 栈的执行函数很简单：

```

-- file: ch18/UglyStack.hs
runApp :: App a -> Int -> IO (a, AppState)
runApp k maxDepth =
  let config = AppConfig maxDepth
      state = AppState 0
  in runStateT (runReaderT k config) state

```

对 `runReaderT` 的应用移除了 `ReaderT` 变换器的包装，之后 `runStateT` 移除了 `StateT` 的包装，最后的结果便留在 `IO monad` 中。

和先前的版本相比，我们的修改并未使代码复杂太多，但现在函数却能记录目前的路径，和达到的最大深度：

```

constrainedCount :: Int -> FilePath -> App [(FilePath, Int)]
constrainedCount curDepth path = do
  contents <- liftIO . listDirectory $ path
  cfg <- ask
  rest <- forM contents $ \name -> do
    let newPath = path </> name
    isDir <- liftIO $ doesDirectoryExist newPath
    if isDir && curDepth < cfgMaxDepth cfg
    then do
      let newDepth = curDepth + 1
      st <- get

```

 v: latest ▾

```

    when (stDeepestReached st < newDepth) $
        put st {stDeepestReached = newDepth}
    constrainedCount newDepth newPath
else return []
return $ (path, length contents) : concat rest

```

在这个例子中如此运用monad变换器确实有些小题大做，因为这仅仅是个简单函数，其并没有因此得到太多的好处。但是这个方法的实用性在于，可以将其 *轻易扩展以解决更加复杂的问题*。

大部分指令式的应用可以使用和这里的 `App monad`类似的方法，在monad栈中编写。在实际的程序中，或许需要携带更复杂的配置数据，但依旧可以使用 `ReaderT` 以保持其只读，并只在需要时暴露配置；或许有更多可变状态需要管理，但依旧可以使用 `StateT` 封装它们。

## 隐藏细节

使用常规的 `newtype` 技术，便可将细节与接口分离开：

```

newtype MyApp a = MyA
{ runA :: ReaderT AppConfig (StateT AppState IO) a
} deriving (Monad, MonadIO, MonadReader AppConfig,
           MonadState AppState)

runMyApp :: MyApp a -> Int -> IO (a, AppState)
runMyApp k maxDepth =
    let config = AppConfig maxDepth
        state = AppState 0
    in runStateT (runReaderT (runA k) config) state

```

若只导出 `MyApp` 类构造器和 `runMyApp` 执行函数，客户端的代码就无法知晓这个monad的内部结构是否是monad栈了。

此处，庞大的 `deriving` 子句需要 `GeneralizedNewtypeDeriving` 语言编译选项。编译器可以为我们生成这些实例，这看似十分神奇，究竟是如何做到的呢？

早先，我们提到 `mtl` 库为每个monad变换器都提供了一系列实例。例如 `IO monad`实现了 `MonadIO`，若下层monad是 `MonadIO` 的实例，那么 `mtl` 也将为其对应的 `StateT` 构建一个 `MonadIO` 的实例，类似的事情也发生在 `ReaderT` 上。

因此，这其中并无太多神奇之处：位于monad栈顶层的monad变换器，已是所有我们声明的 `deriving` 子句中的类型类的实例，我们做的只不过是重新派生这些实例。这是 `mtl` 精心设计的一系列类型类和实例完美配合的结果。除了基于 `newtype` 声明的常规的自动推导以外并没有发生什么。

[译注：注意到此处 `newtype MyApp a` 只是乔装过的 `ReaderT AppConfig (StateT AppState IO) a`。因此我们可以列出 `MyApp a` 这个monad栈的全貌（自顶向下）：

```

ReaderT AppConfig (monad变换器)
StateT AppState (monad变换器)
IO (monad)

```


注意这个monad栈和 `deriving` 子句中类型类的相似度。这些实例都可以自动派生：`MonadIO` 实例自底层派生上来，`MonadStateT` 从中间一层派生，而 `MonadReader` 实例来自顶层。所以虽然 `newtype MyApp a` 引入了一个全新的类型，其实例是可以通过内部结构自动推导的。]

## 练习

1. 修改 `App` 类型别名以交换 `ReaderT` 和 `StateT` 的位置，这一变换对执行函数 `runApp` 会带来什么影响？
2. 为 `App monad`添加 `WriterT` 变换器。相应地修改 `runApp`。
3. 重写 `constrainedCount` 函数，在为 `App` 新添加的 `WriterT` 中记录结果。

[译注：第一题中的 `StateT` 原为 `WriterT`，鉴于 `App` 定义中并无 `WriterT`，此处应该指的是 `StateT`]

## 深入Monad栈中

至今，我们了解了对monad变换器的简单运用。对 `mtl` 库的便利组合拼接使我们免于了解monad栈构造的细节。我们确  `v: latest` ！以帮助我们简化大量常见编程任务的monad变换器相关知识。

但有时，为了实现一些实用的功能，还是我们需要了解 `mtl` 库并不便利的一面。这些任务可能是将定制的monad置于monad栈底，也可能是将定制的monad变换器置于monad变换器栈中的某处。为了解其中潜在的难度，我们讨论以下例子。

假设我们有个定制的monad变换器 `CustomT`：

```
-- file: ch18/CustomT.hs
newtype CustomT m a = ...
```

在 `mtl` 提供的框架中，每个位于栈上的monad变换器都将其下层monad的API暴露出来。这是通过提供大量的类型类实例来实现的。遵从这一模式的规则，我们也可以实现一系列的样板实例：

```
-- file: ch18/CustomT.hs
instance MonadReader r m => MonadReader r (CustomT m) where
    ...

instance MonadIO m => MonadIO (CustomT m) where
    ...
```

若下层monad是 `MonadReader` 的实例，则 `CustomT` 也可作为 `MonadReader` 的实例：实例化的方法是将所有相关的API调用转接给其下层实例的相应函数。经过实例化之后，上层的代码就可以将monad栈作为一个整体，当作 `MonadReader` 的实例，而不再需要了解或关心到底是其中的哪一层提供了具体的实现。

不同于这种依赖类型类实例的方法，我们也可以显式指定想要使用的API。 `MonadTrans` 类型类定义了一个实用的函数 `lift`：

```
ghci> :m +Control.Monad.Trans
ghci> :info MonadTrans
class MonadTrans t where lift :: (Monad m) => m a -> t m a
    -- Defined in Control.Monad.Trans
```

这个函数接受来自monad栈中，当前栈下一层的monad动作，并将这个动作变成，或者说是 *抬举* 到现在的monad变换器中。每个monad变换器都是 `MonadTrans` 的实例。

`lift` 这个名字是基于此函数与 `fmap` 和 `liftM` 目的上的相似度的。这些函数都可以从类型系统的下一层中把东西提升到我们目前工作的这一层。它们的区别是：

`fmap`

将纯函数提升到functor层次

`liftM`

将纯函数提升到monad层次

`lift`

将一monad动作，从monad栈中的下一层提升到本层

[译注：实际上 `liftM` 间接调用了 `fmap`，两个函数在效果上是完全一样的。译者认为，当操作对象是monad（所有的monad都是functor）的时候，使用其中的哪一个只是思考方法上的不同。]

现在重新考虑我们在早些时候定义的 `App` monad栈 (之前我们将其包装在 `newtype` 中)：

```
-- file: ch18/UglyStack.hs
type App = ReaderT AppConfig (StateT AppState IO)
```

若想访问 `StateT` 所携带的 `AppState`，通常需要依赖 `mtl` 的类型类实例来为我们处理组合工作：

```
-- file: ch18/UglyStack.hs
implicitGet :: App AppState
implicitGet = get
```

通过将 `get` 函数从 `StateT` 中抬举进 `ReaderT`，`lift` 函数也可以实现同样的效果：

```
-- file: ch18/UglyStack.hs
explicitGet :: App AppState
```

 v: latest ▾



```
explicitGet = lift get
```

显然当 `mtl` 可以为我们完成这一工作时，代码会变得更清晰。但是 `mtl` 并不总能完成这类工作。

## 何时需要显式的抬举？

我们必须使用 `lift` 的一个例子是：当在一个monad栈中，同一个类型类的实例出现了多次时：

```
-- file: ch18/StackStack.hs
type Foo = StateT Int (State String)
```

若此时我们试着使用 `MonadState` 类型类中的 `put` 动作，得到的实例将是 `StateT Int`，因为这个实例在monad栈顶。

```
-- file: ch18/StackStack.hs
outerPut :: Int -> Foo ()
outerPut = put
```

在这个情况下，唯一能访问下层 `State monad` 的 `put` 函数的方法是使用 `lift`：

```
-- file: ch18/StackStack.hs
innerPut :: String -> Foo ()
innerPut = lift . put
```

有时我们需要访问多于一层以下的monad，这时我们必须组合 `lift` 调用。每个函数组合中的 `lift` 将我们带到更深的一层。

```
-- file: ch18/StackStack.hs
type Bar = ReaderT Bool Foo

barPut :: String -> Bar ()
barPut = lift . lift . put
```

正如以上代码所示，当需要用 `lift` 的时候，一个好习惯是定义并使用包裹函数来为我们完成抬举工作。因为这种在代码各处显式使用 `lift` 的方法使代码变得混乱。另一个显式 `lift` 的缺点在于，其硬编码了monad栈的层次细节，这将使日后对monad栈的修改变得复杂。

## 构建以理解Monad变换器

为了深入理解monad变换器通常是如何运作的，在本节我们将自己构建一个monad变换器，期间一并讨论其中的组织结构。我们的目标简单而实用：`MaybeT`。但是 `mtl` 库意外地并没有提供它。

[译注：如果想使用现成的 `MaybeT`，现在你可以在 `Hackage` 上的 `transformers` 库中找到它。]

这个monad变换器修改monad的方法是：将下层monad `m a` 的类型参数 包装在 `Maybe` 中，以得到类型 `m (Maybe a)`。正如 `Maybe monad` 一样，若在 `MaybeT monad` 变换器中调用 `fail`，则计算将提早结束执行。

为使 `m (Maybe a)` 成为 `Monad` 的实例，其必须有个独特的类型。这里我们通过 `newtype` 声明来实现：

```
-- file: ch18/MaybeT.hs
newtype MaybeT m a = MaybeT
  { runMaybeT :: m (Maybe a) }
```

现在需要定义三个标准的monad函数。其中最复杂的是 `(>>=)`，它的实现也阐明了我们实际上在做什么。在开始研究其操作之前，不妨先看看其类型：

```
-- file: ch18/MaybeT.hs
bindMT :: (Monad m) => MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

为理解其类型签名，回顾之前在十五章中对“多参数类型类”讨论。此处我们想使 *部分类型* `MaybeT m` 成为 `Monad` 的实例。这个部分类型拥有通常的单一类型参数 `a`，这样便能满足 `Monad` 类型类的要求。

 v: latest ▾

[译注：`MaybeT` 的完整定义是 `MaybeT m a`，因此 `MaybeT m` 只是部分应用。]

理解以下 ( $\gg=$ ) 实现的关键在于：do 代码块里的代码是在 下层 monad 中执行的，无论这个下层 monad 是什么。

```
-- file: ch18/MaybeT.hs
x `bindMT` f = MaybeT $ do
  unwrapped <- runMaybeT x
  case unwrapped of
    Nothing -> return Nothing
    Just y -> runMaybeT (f y)
```

我们的 runMaybeT 函数解开了在 x 中包含的结果。进而，注意到 <- 符号是 ( $\gg=$ ) 的语法糖：monad 变换器必须使用其下层 monad 的 ( $\gg=$ )。而最后一部分对 unwrapped 的结构分析（case 表达式），决定了我们是要短路当前计算，还是将计算继续下去。最后，观察表达式的最外层。为了将下层 monad 再次藏起来，这里必须用 MaybeT 构造器包装结果。

刚才展示的 do 标记看起来更容易阅读，但是其将我们依赖下层 monad 的 ( $\gg=$ ) 函数的事实也藏了起来。下面提供一个更符合语言习惯的 MaybeT 的 ( $\gg=$ ) 实现：

```
-- file: ch18/MaybeT.hs
x `altBindMT` f =
  MaybeT $ runMaybeT x >>= maybe (return Nothing) (runMaybeT . f)
```

现在我们了解了 ( $\gg=$ ) 在干些什么。关于 return 和 fail 无需太多解释，Monad 实例也不言自明：

```
-- file: ch18/MaybeT.hs
returnMT :: (Monad m) => a -> MaybeT m a
returnMT a = MaybeT $ return (Just a)

failMT :: (Monad m) => t -> MaybeT m a
failMT _ = MaybeT $ return Nothing

instance (Monad m) => Monad (MaybeT m) where
  return = returnMT
  (>>=) = bindMT
  fail = failM
```

## 建立Monad变换器

为将我们的类型变成 monad 变换器，必须提供 MonadTrans 的实例，以使用户可以访问下层 monad：

```
-- file: ch18/MaybeT.hs
instance MonadTrans MaybeT where
  lift m = MaybeT (Just `liftM` m)
```

下层 monad 以类型 a 开始：我们“注入” Just 构造器以使其变成需要的类型：Maybe a。进而我们通过 MaybeT 藏起下层 monad。

## 更多的类型类实例

在定义好 MonadTrans 的实例后，便可用其来定义其他大量的 mtl 类型类实例了：

```
-- file: ch18/MaybeT.hs
instance (MonadIO m) => MonadIO (MaybeT m) where
  liftIO m = lift (liftIO m)

instance (MonadState s m) => MonadState s (MaybeT m) where
  get = lift get
  put k = lift (put k)

-- ... 对 MonadReader, MonadWriter 等的实例定义同理 ...
```

由于一些 mtl 类型类使用了函数式依赖，有些实例的声明需要 GHC 大大放宽其原有的类型检查规则。(若我们忘记了其中任意的 LANGUAGE 指令，编译器会在其错误信息中提供建议。)

 v: latest ▾



```
-- file: ch18/MaybeT.hs
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses,
       UndecidableInstances #-}
```

是花些时间来写这些样板实例呢，还是显式地使用 `lift` 呢？这取决于这个monad变换器的用途。如果我们只在几种有限的情况下使用它，那么只提供 `MonadTrans` 实例就够了。在这种情况下，也无妨提供一些依然有意义的实例，比如 `MonadIO`。另一方面，若我们需要在大量的情况下使用这一monad变换器，那么花些时间来完成这些实例或许也不错。

## 以Monad栈替代Parse类型

现在我们已开发了一个支持提早退出的monad变换器，可以用其来辅助开发了。例如，此处若想处理解析一半失败的情况，便可以用这一以我们的需求定制的monad变换器来替代我们在第十章“隐式状态”一节开发的 `Parse` 类型。

## 练习

1. 我们的Parse monad还不是之前版本的完美替代。因为其用的是 `Maybe` 而不是 `Either` 来代表结果。因此在失败时暂时无法提供任何有用的信息。

构建一个 `EitherT s`（其中 `s` 是某个类型）来表示结果，并用其实现更适合的 `Parse monad`以在解析失败时汇报具体错误信息。

或许在你探索Haskell库的途中，在 `Control.Monad.Error` 遇到过一个 `Either` 类型的 `Monad` 实例。我们建议不要参照它来完成你的实现，因为它的设计太局限了：虽然其将 `Either String` 变成一个monad，但实际上把 `Either` 的第一个类型参数限定为 `String` 并非必要。

提示: 若你按照这条建议来做，你的定义中或许需要使用 `FlexibleInstances` 语言扩展。

## 注意变换器堆叠顺序

从早先使用 `ReaderT` 和 `StateT` 的例子中，你或许会认为叠加monad变换器的顺序并不重要。事实并非如此，考虑在 `State` 上叠加 `StateT` 的情况，或许会助你更清晰地意识到：堆叠的顺序确实产生了结果上的区别：类型 `StateT Int (State String)` 和类型 `StateT String (State Int)` 或许携带的信息相同，但它们却无法互换使用。叠加的顺序决定了我们是否要用 `lift` 来取得状态中的某个部分。

下面的例子更加显著地阐明了顺序的重要性。假设有个可能失败的计算，而我们想记录下在什么情况下其会失败：

```
-- file: ch18/MTComposition.hs
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Writer
import MaybeT

problem :: MonadWriter [String] m => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

那么这两个monad栈中的哪一个会带给我们需要的信息呢？

```
type A = WriterT [String] Maybe

type B = MaybeT (Writer [String])

a :: A ()
a = problem

b :: B ()
b = problem
```

我们在 `ghci` 中试试看：

```
ghci> runWriterT a
Loading package mtl-1.1.0.1 ... linking ... done.
Nothing
```

 v: latest ▾

```
ghci> runWriter $ runMaybeT b
(Nothing, ["this is where i fail"])
```

看看执行函数的类型签名，其实结果并不意外：

```
ghci> :t runWriterT
runWriterT :: WriterT w m a -> m (a, w)
ghci> :t runWriter . runMaybeT
runWriter . runMaybeT :: MaybeT (Writer w) a -> (Maybe a, w)
```

在 `Maybe` 上叠加 `WriterT` 的策略使 `Maybe` 成为下层monad，因此 `runWriterT` 必须给我们以 `Maybe` 为类型的结果。在测试样例中，我们只会在不出现任何失败的情况下才能获得日志！

叠加monad变换器类似于组合函数：如果我们改变函数应用的顺序，那么我们并不会对得到不同的结果感到意外。同样的道理也适用于对monad变换器的叠加。

## 纵观Monad与Monad变换器

本节，让我们暂别细节，讨论一下用monad和monad变换器编程的优缺点。

### 对纯代码的干涉

在实际编程中，使用monad的最恼人之处或许在于其阻碍了我们使用纯代码。很多实用的纯函数需要一个monad版的类似函数，而其monad版只是加上一个占位参数 `m` 供monad类型构造器填充：

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> :i filterM
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
-- Defined in Control.Monad
```

然而，这种覆盖是有限的：标准库中并不总能提供纯函数的monad版本。

其中有一部分历史原因：Eugenio Moggi于1988年引入了使用monad编程的思想。而当时Haskell 1.0标准尚在开发中。现今版本的Prelude中的大部分函数可以追溯到1990发布的Haskell 1.0。在1991年，Philip Wadler开始为更多的函数式编程听众作文，阐述monad的潜力。从那时起，monad开始用于实践。

直到1996年Haskell 1.3标准发布之时，monad才得到了支持。但是在那时，语言的设计者已经受制于维护向前兼容性：它们无法改变Prelude中的函数签名，因为那会破坏现有的代码。

从那以后，Haskell社区学会了很多合适的抽象。因此我们可以写出不受这一纯函数 / monad函数分裂影响的代码。你可以在 `Data.Traversable` 和 `Data.Foldable` 中找到这些思想的精华。

尽管它们极具吸引力，由于版面的限制。我们不会在本书中涵盖相关内容。但如果你能轻易理解本章内容，自行理解它们也不会有问题。

在理想世界里，我们是否会与过去断绝，并让Prelude包含 `Traversable` 和 `Foldable` 类型呢？或许不会，因为学习Haskell本身对新手来说已经是个相当刺激的历程了。在我们已经了解functor和monad之后，`Foldable` 和 `Traversable` 的抽象是十分容易理解的。但是对学习来说这意味着摆在他们面前的是更多纯粹的抽象。若以教授语言为目的，`map` 操作的最好是列表，而不是functor。

[译注：实际上，自GHC 7.10开始，`Foldable` 和 `Traversable` 已经进入了Prelude。一些函数的类型签名会变得更加抽象（以GHC 7.10.1为例）：

```
ghci-7.10.1> :t mapM
mapM :: (Monad m, Traversable t) => (a -> m b) -> t a -> m (t b)
ghci-7.10.1> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

这并不是一个对初学者友好的改动，但由于新的函数只是旧有函数的推广形式，使用旧的函数签名依旧可以通过类型检查

 v: latest ▾

```
ghci-7.10.1> :t (mapM :: Monad m => (a -> m b) -> [a] -> m [b])
(mapM :: Monad m => (a -> m b) -> [a] -> m [b])
:: Monad m => (a -> m b) -> [a] -> m [b]
ghci-7.10.1> :t (foldl :: (b -> a -> b) -> b -> [a] -> b)
(foldl :: (b -> a -> b) -> b -> [a] -> b)
:: (b -> a -> b) -> b -> [a] -> b
```

若在学习过程中遇到障碍，不妨暂且以旧的类型签名来理解它们。]

## 对次序的过度限定

我们使用monad的一个基本原因是：其允许我们指定效果发生的次序。再看看我们早先写的一小段代码：

```
-- file: ch18/MTComposition.hs
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Writer
import MaybeT

problem :: MonadWriter [String] m => m ()
problem = do
  tell ["this is where i fail"]
  fail "oops"
```

因为我们在monad中执行，`tell` 的效果可以保证发生在 `fail` 之前。这里的问题在于，这个次序并不必要，但是我们却得到了这样的次序保证。编译器无法任意安排monad式代码的次序，即便这么做能使代码效率更高。

[译注：解释一下这里的“次序并不必要”。回顾之前对叠加次序问题的讨论：

```
type A = WriterT [String] Maybe

type B = MaybeT (Writer [String])

a :: A ()
a = problem
-- runWriterT a == Nothing

b :: B ()
b = problem
-- runWriter (runMaybeT b) == (Nothing, ["this is where i fail"])
```

下面把注意力集中于 `a`：注意到 `runWriterT a == Nothing`，`tell` 的结果并不需要，因为接下来的 `fail` 取消了计算，将之前的结果抛弃了。利用这个事实，可以得知让 `fail` 先执行效率更高。同时注意对 `fail` 和 `tell` 的实际处理来自monad栈的不同层，所以在一定限制下调换某些操作的顺序会不影响结果。但是由于这个monad栈本身也要是个monad，使这种本来可以进行的交换变得不可能了。]

## 运行时开销

最后，当我们使用monad和monad变换器时，需要付出一些效率的代价。例如 `State monad` 携带状态并将其放在一个闭包中。在Haskell的实现中，闭包的开销或许廉价但绝非免费。

Monad变换器把其自身的开销附加在了其下层monad之上。每次我们使用 `(>>=)` 时，`MaybeT` 变换器便需要包装和解包。而由 `ReaderT`，`StateT` 和 `MaybeT` 依次叠加组成的monad栈，在每次使用 `(>>=)` 时，更是有一系列的簿记工作需要完成。

一个足够聪明的编译器或许可以将这些开销部分，甚至于全部消除。但是那种深度的复杂工作尚未广泛适用。

但是依旧有些相对简单技术可以避免其中的一些开销，版面的限制只允许我们在此做简单描述。例如，在continuation monad中，对 `(>>=)` 频繁的包装和解包可以避免，仅留下执行效果的开销。所幸的是使用这种方法所要考虑的大部分复杂问题，已经在函数库中得到了处理。

这一部分的工作在本书写作时尚在积极的开发中。如果你想让你对monad变换器的使用更加高效，我们推荐在Hackage中寻找相关的库或是在邮件列表或IRC上寻求指引。

 v: latest ▾

## 缺乏灵活性的接口

若我们只把 `mtl` 当作黑盒，那么所有的组件将很好地合作。但是若我们开始开发自己的monad和monad变换器，并想让它们于 `mtl` 提供的组件配合，这种缺陷便显现出来了。

例如，我们开发一个新的monad变换器 `FooT`，并想沿用 `mtl` 中的模式。我们就必须实现一个类型类 `MonadFoo`。若我们想让其更好地和 `mtl` 配合，那么便需要提供大量的实例来支持 `mtl` 中的类型类。

除此之外，还需要为每个 `mtl` 中的变换器提供 `MonadFoo` 的实例。大部分的实例实现几乎是完全一样的，写起来也十分乏味。若我们想在 `mtl` 中集成更多的monad变换器，那么我们需要处理的各类活动部件将达到引入的monad变换器数量的 *平方级别*！

公平地看来，这个问题会只影响到少数人。大部分 `mtl` 的用户并不需要开发新的monad。

造成这一 `mtl` 设计缺陷的原因在于，它是第一个monad变换器的函数库。想像其设计者投入这个未知的领域，完成了大量的工作以使这个强大的函数库对于大部分用户来说做到简便易用。

一个新的关于monad和变换器的函数库 `monadLib`，修正了 `mtl` 中大量的设计缺陷。若在未来你成为了一个monad变换器的中坚骇客，这值得你一试。

平方级别的实例定义实际上是使用monad变换器带来的问题。除此之外另有其他的手段来组合利用monad。虽然那些手段可以避免这类问题，但是它们对最终用户而言仍不及monad变换器便利。幸运的是，并没有太多基础而泛用的monad变换器需要去定义实现。

### 综述

Monad在任何意义下都不是处理效果和类型的终极途径。它只是在我们探索至今，处理这类问题最为实用的技术。语言的研究者们一直致力于找到可以扬长避短的替代系统。

尽管在使用它们时我们必须做出妥协，monad和monad变换器依旧提供了一定程度上的灵活度和控制，而这在指令式语言中并无先例。仅仅几个声明，我们就可以给分号般基础的东西赋予崭新的意义。

[译注：此处的分号应该指的是 `do` 标记中使用的分号。]

### 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

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

第八章：高效文件处理、正则表达式、文件名匹配 — Real World Haskell 中文版

1条评论 • 4年前

Yutong Zhang —

Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

forlice — ...

第九章：I/O学习 —— 构建一个用于搜索文件系统的库 — Real World Haskell 中文版

3条评论 • 4年前

在原佐为 — “继续从左往右看” 这里英文是“from right to left”，是从右到左吧。

v: latest