

第九章：I/O学习 —— 构建一个用于搜索文件系统的库

自从电脑有了分层文件系统以来，“我知道有这个文件，但不知道它放在哪”这个问题就一直困扰着人们。1974年发布的Unix第五个版本引入的 `find` 命令，到今天仍在使用。查找文件的艺术已经走过了很长一段路：伴随现代操作系统一起不断发展的文件索引和搜索功能。

给程序员的工具箱里添加类似 `find` 这样的功能依旧非常 valuable，在本章，我们将通过编写一个Haskell库给我们的 `find` 命令添加更多功能，我们将通过一些有着不同的健壮度的方法来完成这个库。

find命令

如果你不曾用过类Unix的系统，或者你不是个重度shell用户，那么你可能从未听说过 `find`，通过给定的一组目录，它递归搜索每个目录并且打印出每个匹配表达式的实体名称。

```
-- file: ch09/RecursiveContents.hs
module RecursiveContents (getRecursiveContents) where
import Control.Monad (forM)
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
getRecursiveContents :: FilePath -> IO [FilePath]
getRecursiveContents topdir = do
  names <- getDirectoryContents topdir
  let properNames = filter (\name -> notElem [".", ".."] name) names
  paths <- forM properNames $ \name -> do
    let path = topdir </> name
    isDirectory <- doesDirectoryExist path
    if isDirectory
      then getRecursiveContents path
      else return [path]
  return (concat paths)
```

单个表达式可以识别像“符合这个全局模式的名称”，“实体是一个文件”，“当前最后一个被修改的文件”以及其他诸如此类的表达式，通过 `and` 或 `or` 算子就可以把他们装配起来构成更加复杂的表达式

简单的开始：递归遍历目录

在投入设计我们的库之前，先解决一些规模稍小的问题，我们第一个问题就是递归地列出一个目录下面的所有内容和它的子目录

`filter` 表达式确保一个目录的列表不含特定的目录名（比如代表当前目录的 `.` 和上一级目录的 `..`），如果忘记过滤这些，随后的查找将陷入无限循环。

我们在之前的章节里完成了 `forM` 函数，它是参数颠倒后的 `mapM` 函数。

```
ghci> :m +Control.Monad
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type forM
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
```

循环体将检查当前实体是否为目录，如果是，则递归调用 `getRecursiveContents` 函数列出这个目录（的内容），如果否，则返回只含有当前实体名称一个元素的列表，不要忘记 `return` 函数在 Haskell 中有特殊的含义，他通过 `monad` 的类型构造器包装了一个值。

另一个值得注意的地方是变量 `isDirectory` 的使用，在命令式语言如 Python 中，我们通常用 `if os.path.isdir(path)` 来表示，然而，`doesDirectoryExist` 函数是一个动作，它的返回类型是 `IO Bool` 而非 `Bool`，由于 `if` 表达式需要一个操作值为 `bool` 的表达式作为条件，我们使用 `<-` 来从 `io` 包装器上得到这个动作的 `bool` 返回值，这样我们就能在 `if` 中使用这个干净的无包装的 `bool`。

循环体中每一次迭代生成的结果都是名称列表，因此 `forM` 的结果是 `IO [[FilePath]]`，我们通过 `concat` 将它转换为一个元素列表(从以列表为元素的列表转换为不含列表元素的列表)

 v: latest ▾

再次认识匿名和命名函数

在 **Anonymous (lambda) functions** 这部分，我们列举了一系列不使用匿名函数的原因，然而在这里，我们将使用它作为函数体，这是匿名函数在 Haskell 中最常见的用途之一。

我们已经在 `from` 和 `mapM` 上看到使用函数作为参数的方式，许多循环体是程序中只出现一次的代码块。既然我们喜欢在循环中使用一个再也不会出现的循环体，那么为什么要给他们命名？

显而易见，有时候我们需要在不同的循环中嵌入相同的代码，这时候我们不应该使用匿名函数，把他们剪贴和复制进去，而是给这些匿名函数命名来调用，这样显得有意义一点

为什么提供 `mapM` 和 `forM`

存在两个相同的函数看起来是有点奇怪，但接受参数的顺序之间的差异使他们适用于不同的情况。

我们来考察下之前的例子，使用匿名函数作为循环体，如果我们使用 `mapM` 而非 `forM`，我们将不得不把变量 `properNames` 放置到函数体的后边，而为了让代码正确解析，我们就必须将整个匿名函数用括号包起来，或者用一个不必要的命名函数将它取代，自己尝试下，拷贝上边的代码，用 `mapM` 代替 `forM`，观察代码可读性上有什么变化

相反，如果循环体是一个命名函数，而且我们要循环的列表是通过一个复杂表达式计算的，我们就找到了 `mapM` 的应用场景

这里需要遵守的代码风格是无论通过 `mapM` 和 `forM` 都让你写出干净的代码，如果循环体或者循环中的表达式都很短，那么用哪个都无所谓，如果循环体很短，但数据很长，使用 `mapM`，如果相反，则用 `forM`，如果都很长，使用 `let` 或者 `where` 让其中一个变短，通过这样一些实践，不同情况下那个实现最好就变得显而易见

一个本地查找函数

我们可以使用 `getRecursiveContents` 函数作为一个内置的简单文件查找器的基础

```
-- file: ch09/SimpleFinder.hs
import RecursiveContents (getRecursiveContents)
simpleFind :: (FilePath -> Bool) -> FilePath -> IO [FilePath]
simpleFind p path = do
    names <- getRecursiveContents path
    return (filter p names)
```

上文的函数通过我们在过滤器中的谓词来匹配 `getRecursiveContents` 函数返回的名字，每个通过谓词判断的名称都是文件全路径，因此如何完成一个像“查找所有扩展名以 `.c` 结尾的文件”的功能？


`System.FilePath` 模块包含了许多有价值的函数来帮助我们操作文件名，在这个例子中，我们使用 `takeExtension`：

```
ghci> :m +System.FilePath
ghci> :type takeExtension
takeExtension :: FilePath -> String
ghci> takeExtension "foo/bar.c"
Loading package filepath-1.1.0.0 ... linking ... done.
".c"
ghci> takeExtension "quux"
""
```

下面的代码给我们一个包括获得路径，获得扩展名，然后和 `.c` 进行比较的简单功能的函数实现，

```
ghci> :load SimpleFinder
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main ( SimpleFinder.hs, interpreted )
Ok, modules loaded: RecursiveContents, Main.
ghci> :type simpleFind (\p -> takeExtension p == ".c")
simpleFind (\p -> takeExtension p == ".c") :: FilePath -> IO [FilePath]
```

`simpleFind` 在工作中有一些非常刺眼的问题，第一个就是谓词并不能准确而完全的表达，他只关注文件夹中的实体名称，而无法做到辨认这是个文件还是个目录此类的事情，——而我们使用 `simpleFind` 的尝试就是想列举所文件和有和文件一样拥有 `.c` 扩展名的文件夹

第二个问题是在 `simpleFind` 中我们无法控制它遍历文件系统的方式，这是显而易见的，想想在分布式版本控制系统中  `v: latest` 中查找一个源文件的问题吧，所有被控制的目录都含有一个 `.svn` 的私有文件夹，每一个包含了许多我们毫不感兴趣的子文件夹和文

件，简单的过滤所有包含 `.svn` 的路径远比仅仅在读取时避免遍历这些文件夹更加有效。例如，一个分布式源码树包含了45000个文件，30000个分布在1200个不同的.svn文件夹中，避免遍历这1200个文件夹比过滤他们包含的30000个文件代价更低。

最后，`simpleFind` 是严格的，因为它包含一系列IO元操作执行构成的动作，如果我们有一百万个文件要遍历，我们需要等待很长一段时间才能得到一个包含一百万个名字的巨大的返回值，这对用户体验和资源消耗都是噩梦，我们更需要一个只有当他们获得结果的时才展示的结果流。

在接下来的环节里，我们将解决每个遇到的问题

谓词在保持纯粹的同时支持从贫类型到富类型

我们的谓词只关注文件名，这将一系列有趣的操作排除在外，试想下，假如我们希望列出比某个给定值更大的文件呢？

面对这个问题的第一反应是查找 `IO`：我们的谓词是 `FilePath -> Bool` 类型，为什么不把它变成 `FilePath -> IO Bool` 类型？这将使我们所有的IO操作都成为谓词的一部分，但这在显而易见的好处之外引入一个潜在的问题，使用这样一个谓词存在各种可能的后果，比如一个有 `IO a` 类型返回的函数将有能力生成任何它想产生的结果。

让我们在类型系统中寻找以写出拥有更多谓词，更少bug的代码，我们通过避免污染IO来坚持断言的纯粹，这将确保他们不会产生任何不纯的结果，同时我们给他们提供更多信息，这样他们就可以在不必诱发潜在的危险的情况下获得需要的表达式

Haskell 的 `System.Directory` 模块提供了一个尽管受限但仍然有用的关于文件元数据的集合


```
ghci> :m +System.Directory
```

我们可以通过 `doesFileExist` 和 `doesDirectoryExist` 来判断目录实体是目录还是文件，但暂时还没有更多方式来查找这些年里出现的纷繁复杂的其他文件类型，比如管道，硬链接和软连接。

```
ghci> :type doesFileExist
doesFileExist :: FilePath -> IO Bool
ghci> doesFileExist "."
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
False
ghci> :type doesDirectoryExist
doesDirectoryExist :: FilePath -> IO Bool
ghci> doesDirectoryExist "."
True
```

`getPermissions` 函数让我们确定当前对于文件或目录的操作是否是合法：

```
ghci> :type getPermissions
getPermissions :: FilePath -> IO Permissions
ghci> :info Permissions
data Permissions
  = Permissions {readable :: Bool,
                 writable :: Bool,
                 executable :: Bool,
                 searchable :: Bool}
  -- Defined in System.Directory
instance Eq Permissions -- Defined in System.Directory
instance Ord Permissions -- Defined in System.Directory
instance Read Permissions -- Defined in System.Directory
instance Show Permissions -- Defined in System.Directory
ghci> getPermissions "."
Permissions {readable = True, writable = True, executable = False, searchable = True}
ghci> :type searchable
searchable :: Permissions -> Bool
ghci> searchable it
True
```

如果你无法回忆起 `ghci` 中变量 `it` 的特殊用法，回到第一章复习一下，如果我们的权限能够列出它的内容，那么这个目录  `v: latest` 被搜索的，而文件则永远是不可搜索的

最后，`getModificationTime` 告诉我们实体上次被修改的时间：

```
ghci> :type getModificationTime
getModificationTime :: FilePath -> IO System.Time.ClockTime
ghci> getModificationTime "."
Mon Aug 18 12:08:24 CDT 2008
```

如果我们像标准的Haskell代码一样对可移植性要求严格，这些函数就是我们手头所有的一切(我们同样可以通过黑客手段来获得文件大小)，这些已经足够让我们明白所感兴趣领域中的原则，而非让我们浪费宝贵的时间对着一个例子冥思苦想，如果你需要写满足更多需求的代码，`System.Posix` 和 `System.Win32` 模块提供关于当代两种计算平台的更多文件元数据的细节。`Hackage` 中同样有一个 `unix-compat` 包，提供windows下的类unix的api。

新的富类型谓词需要关注的数据段到底有几个？自从我们可以通过 `Permissions` 来判断实体是文件还是目录之后，我们就不再需要获得 `doesFileExist` 和 `doesDirectoryExist` 的结果，因此一个谓词需要关注的输入有四个。

```
-- file: ch09/BetterPredicate.hs
import Control.Monad (filterM)
import System.Directory (Permissions(..), getModificationTime, getPermissions)
import System.Time (ClockTime(..))
import System.FilePath (takeExtension)
import Control.Exception (bracket, handle)
import System.IO (IOMode(..), hClose, hFileSize, openFile)

-- the function we wrote earlier
import RecursiveContents (getRecursiveContents)

type Predicate = FilePath      -- path to directory entry
                -> Permissions -- permissions
                -> Maybe Integer -- file size (Nothing if not file)
                -> ClockTime    -- last modified
                -> Bool
```

这一谓词类型只是一个有四个参数的函数的同义词，他将给我们节省一些键盘工作和屏幕空间。

注意这一返回值是 `Bool` 而非 `IO Bool`，谓词需要保证纯粹，而且不能表现IO，在拥有这种类型以后，我们的查找函数仍然显得非常空白。

```
-- file: ch09/BetterPredicate.hs
-- soon to be defined
getFileSize :: FilePath -> IO (Maybe Integer)

betterFind :: Predicate -> FilePath -> IO [FilePath]

betterFind p path = getRecursiveContents path >>= filterM check
  where check name = do
    perms <- getPermissions name
    size <- getFileSize name
    modified <- getModificationTime name
    return (p name perms size modified)
```

先来阅读代码，由于随后将讨论 `getFileSize` 的某些细节，因此现在暂时先跳过它。

我们无法使用 `filter` 来调用我们的谓词，因为 `p` 的纯粹代表他不能作为IO收集元数据的方式

这让我们将目光转移到一个并不熟悉的函数 `filterM` 上，它的动作就像普通的 `filter` 函数，但在这种情况下，它在 `IO monad` 操作中使用它的谓词，进而通过谓词表现IO：

```
ghci> :m +Control.Monad
ghci> :type filterM
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

`check` 谓词是纯谓词 `p` 的IO功能包装器，他执行了所有IO发生在 `p` 上的可能引起负面效果的任务，因此我们可以使 `p` 对负面效果免疫，在收集完元数据后，`check` 调用 `p`，通过 `return` 语句包装 `p` 的IO返回结果

 v: latest ▾

安全的获得一个文件的大小

即使 `System.Directory` 不允许我们获得一个文件的大小，我们仍可以使用 `System.IO` 的类似接口完成这项任务，它包含了一个名为 `hFileSize` 的函数，这一函数返回打开文件的字节数，下面是他的简单调用实例：

```
-- file: ch09/BetterPredicate.hs
simpleFileSize :: FilePath -> IO Integer

simpleFileSize path = do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return size
```

当这个函数工作时，他还不能完全为我们所用，在 `betterFind` 中，我们在目录下的任何实体上调用 `getFileSize`，如果实体不是一个文件或者大小被 `Just` 包装起来，他应当返回一个空值，而当实体不是文件或者没有被打开时（可能是由于权限不够），这个函数会抛出一个异常然后返回一个未包装的大小。

下文是安全的用法：

```
-- file: ch09/BetterPredicate.hs
saferFileSize :: FilePath -> IO (Maybe Integer)

saferFileSize path = handle (\_ -> return Nothing) $ do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return (Just size)
```

函数体几乎完全一致，除了 `handle` 语句。

我们的异常捕捉在忽略通过的异常的同时返回一个空值，函数体唯一的变化就是允许通过 `Just` 包装文件大小

`saferFileSize` 函数现在有正确的类型签名，并且不会抛出任何异常，但他仍未能完全的正常工作的，存在 `openFile` 会成功的目录实体，但 `hFileSize` 会抛出异常，这将和被称作命名管道的状况一起发生，这样的异常会被捕捉，但却从未发起调用 `hClose`。

当发现不再使用文件句柄，Haskell会自动关闭它，但这只有在运行垃圾回收时才会执行，如果无法断言，则延迟到下一次垃圾回收。

文件句柄是稀缺资源，稀缺性是通过操作系统强制保证的，在linux中，一个进程只能同时拥有1024个文件句柄。

不难想象这种场景，程序调用了使用 `saferFileSize` 的 `betterFind` 函数，在足够的垃圾文件句柄被关闭之前，由于 `betterFind` 造成文件句柄数耗尽导致了程序崩溃

这是bug危害性的一方面：通过合并起来的不同的部分使得bug不易被排查，只有在 `betterFind` 访问足够多的非文件达到进程打开文件句柄数上限的时候才会被触发，随后在积累的垃圾文件句柄被关闭之前返回一个尝试打开另一个文件的调用。

任何程序内由无法获得数据造成的后续错误都会让事情变得更糟，直到垃圾回收为止。修正这样一个bug需要程序结构本身支持，文件系统内容，如何关闭当前正在运行的程序以触发垃圾回收

这种问题在开发中很容易被检查出来，然而当他在上线之后出现（这些恶心的问题一向如此），就变得非常难以发觉

幸运的是，我们可以很容易避开这种错误，同时又能缩短我们的函数。

请求-使用-释放循环

每当 `openFile` 成功之后我们就必须保证调用 `hClose`，`Control.Exception` 模块提供了 `bracket` 函数来支持这个想法：

```
ghci> :type bracket
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

`bracket` 函数需要三个动作作为参数，第一个动作需要一个资源，第二个动作释放这个资源，第三个动作在这两个中执行。我们称他为操作动作，当请求动作成功，释放动作随后总是被调用，这保证了这个资源一直能够被释放，对通过的操作，使用和释放动作都是必要的。

如果一个异常发生在使用过程中，`bracket` 调用释放动作并抛出异常，如果使用动作成功，`bracket` 调用释放动作，同时返回使用动作返回的值。

我们现在可以写一个完全安全的函数了，他将不会抛出异常，也不会积累可能在我们程序其他地方制造失败的垃圾文件句柄数。

```
-- file: ch09/BetterPredicate.hs
getFileSize path = handle (\_ -> return Nothing) $
  bracket (openFile path ReadMode) hClose $ \h -> do
    size <- hFileSize h
    return (Just size)
```

仔细观察 `bracket` 的参数，首先打开文件，并且返回文件句柄，第二步关闭句柄，第三步在句柄上调用 `hFileSize` 并用 `just` 包装结果返回。为了这个函数的正常工作，我们需要使用 `bracket` 和 `handle`，前者保证我们不会积累垃圾文件句柄数，后者保证我们免于异常。

练习

1. 调用 `bracket` 和 `handle` 的顺序重要吗，为什么

为谓词而开发的领域特定语言

深入谓词写作的内部，我们的谓词将检查大于128kb的C++源文件：

```
-- file: ch09/BetterPredicate.hs
myTest path _ (Just size) _ =
  takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

这并不是令人感到愉快的工作，断言需要四个参数，并且总是忽略其中的两个，同时需要定义两个等式，写一些更有意义的谓词代码，我们可以做的更好。

有些时候，这种库被用作嵌入式领域特定语言，我们通过编写代码的过程中通过编程语言的本地特性来优雅的解决一些特定问题

第一步是写一个返回当前函数的一个参数的函数，这个从参数中抽取路径并传给谓词：

```
-- file: ch09/BetterPredicate.hs
pathP path _ _ _ = path
```

如果我们不能提供类型签名，Haskell 将给这个函数提供一个通用类型，这在随后会导致一个难以理解的错误信息，因此给 `pathP` 一个类型：

```
-- file: ch09/BetterPredicate.hs
type InfoP a = FilePath      -- path to directory entry
              -> Permissions  -- permissions
              -> Maybe Integer -- file size (Nothing if not file)
              -> ClockTime    -- last modified
              -> a

pathP :: InfoP FilePath
```

我们已经创建了一个可以用做缩写的类型，相似的结构函数，我们的类型代词接受一个类型参数，如此我们可以分辨不同的结果类型：

```
-- file: ch09/BetterPredicate.hs
sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing _ = -1
```

我们在这里做了些小动作，对那些我们无法打开的文件或者不是文件的东西我们返回的实体大小是 `-1`。

事实上，浏览中可以看出我们在本章开始处定义谓词类型的和 `InfoP Bool` 一样，因此我们可以合法的放弃谓词类型。

 v: latest ▾

`pathP` 和 `sizeP` 的用法？通过一些线索，我们发现可以在一个谓词中使用它们（每个名称中的前缀`p`代表断言），从这开始事情就变得有趣起来：

```
-- file: ch09/BetterPredicate.hs
equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP f k = \w x y z -> f w x y z == k
```

`equalP` 的类型签名值得注意，他接受一个 `InfoP a`，同时兼容 `pathP` 和 `sizeP`，他接受一个 `a`，并返回一个被认为是谓词同义词的 `InfoP Bool`，换言之，`equalP` 构造了一个谓词。

`equalP` 函数通过返回一个匿名函数工作，谓词接受参数之后将他们转成 `f`，并将结果和 `k` 进行比对。

`equalP` 的相等强调了这一事实，我们认为它需要两个参数，在 Haskell 柯里化处理了所有函数的情况下，通过这种方式写 `equalP` 并无必要，我们可以避免匿名函数，同时通过柯里化来写出表现相同的函数：

```
-- file: ch09/BetterPredicate.hs
equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k
```

在继续我们的探险之前，先把写好的模块加载到 `ghci` 里去：

```
ghci> :load BetterPredicate
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main ( BetterPredicate.hs, interpreted )
Ok, modules loaded: RecursiveContents, Main.
```

让我们来看看函数中的简单谓词能否正常工作：

```
ghci> :type betterFind (sizeP `equalP` 1024)
betterFind (sizeP `equalP` 1024) :: FilePath -> IO [FilePath]
```

注意我们并没有直接调用 `betterFind`，我们只是确定我们的表达式进行了类型检查，现在我们需要更多的方法来列出大小为特定值的所有文件，之前的成功给了我们继续下去的勇气。

多用提升 (lifting) 少用模板

除了 `equalP`，我们还将能够编写其他二进制函数，我们更希望不去写他们每个的具体实现，因为这看起来只是重复工作：

```
-- file: ch09/BetterPredicate.hs
liftP :: (a -> b -> c) -> InfoP a -> b -> InfoP c
liftP q f k w x y z = f w x y z `q` k

greaterP, lesserP :: (Ord a) => InfoP a -> a -> InfoP Bool
greaterP = liftP (>)
lesserP = liftP (<)
```

为了完成这个，让我们使用 Haskell 的抽象功能，定义 `equalP` 代替直接调用 `==`，我们就可以把二进制函数作为参数传入我们想调用的函数。

函数动作，比如 `>`，以及将它转换成另一个函数操作另一种上下文，在这里是 `greaterP`，通过提升 (lifting) 将它引入到上下文，这解释了当前函数名称中 `lifting` 出现的原因，提升让我们复用代码并降低模板的使用，在本书的后半部分的内容中，我们将大量使用这一技术

当我们提升一个函数，我们通常将它转换到原始类型和一个新版本——提升和未提升两个版本

在这里，将 `q` 作为 `liftP` 的第一个参数是经过深思熟虑的，这使得我们可能写一个对 `greaterP` 和 `lesserP` 都有意义的定义，实践中发现，相较其他语言，Haskell 中参数的最佳适配成为 api 设计中最重要的一部分。语言内部要求参数转换，在 Haskell 中放错一个参数的位置就将失去程序的所有意义。

我们可以通过组合字 (combinators) 恢复一些意义，比如，直到2007年 `forM` 才加入 `Control.Monad` 模块，在此之前， `lip` `mapM`。

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :t forM
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
ghci> :t flip mapM
flip mapM :: (Monad m) => [a] -> (a -> m b) -> m [b]
```

谓词组合

如果我们希望组合谓词，我们可以循着手边最明显的路径来开始

```
-- file: ch09/BetterPredicate.hs
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z
```

现在我们知道了解决提升，他成为通过提升存在的布尔操作来削减代码量的更自然的选择。

```
-- file: ch09/BetterPredicate.hs
liftP2 :: (a -> b -> c) -> InfoP a -> InfoP b -> InfoP c
liftP2 q f g w x y z = f w x y z `q` g w x y z

andP = liftP2 (&&)
orP = liftP2 (||)
```

注意 liftP2 非常像我们之前的 liftP，事实上，这更加通用，因为我们可以用 liftP 代替 liftP2：

```
-- file: ch09/BetterPredicate.hs
constP :: a -> InfoP a
constP k _ _ _ _ = k

liftP' q f k w x y z = f w x y z `q` constP k w x y z
```

Note

组合子

在Haskell中，我们更希望函数的传入参数和返回值都是函数，就像组合子一样

回到之前定义的 myTest 函数，现在我们可以使用一些帮助函数了。

```
-- file: ch09/BetterPredicate.hs
myTest path _ (Just size) _ =
    takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

在加入组合字以后这个函数会变成什么样子：

```
-- file: ch09/BetterPredicate.hs
liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP`
    (sizeP `greaterP` 131072)
```

由于操作文件名是如此平常的行为，我们加入了最终组合字 liftPath。

定义并使用新算符

可以通过特定领域语言定义新的操作：


```
-- file: ch09/BetterPredicate.hs
(==?) = equalP
(&&?) = andP
(>?) = greaterP

myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)
```

这个括号在定义中是必要的，因为并未告诉Haskell有关之前和相关的操作，领域语言的操作如果没有边界（fixities）声明将会被以 `infixl 9` 之类的东西对待，计算从左到右，如果跳过这个括号，表达式将被解析成具有可怕错误的 `((liftPath takeExtension) ==? ".cpp") &&? sizeP) >? 131072`。

可以给操作添加边界声明，第一步是找出未提升的操作的，这样就可以模仿他们了：

```
ghci> :info ==
class Eq a where
  (==) :: a -> a -> Bool
  ...
  -- Defined in GHC.Base
infix 4 ==
ghci> :info &&
(&&) :: Bool -> Bool -> Bool      -- Defined in GHC.Base
infixr 3 &&
ghci> :info >
class (Eq a) => Ord a where
  ...
  (>) :: a -> a -> Bool
  ...
  -- Defined in GHC.Base
infix 4 >
```

学会这些就可以写一个不用括号的表达式，却和 `myTest3` 的解析结果一致的表达式了

控制遍历

遍历文件系统时，我们喜欢在需要遍历的文件夹上有更多的控制权，简便方法之一是在函数中允许给定文件夹的部分子文件夹通过，然后返回另一个列表，这个列表可以移除元素，也可以要求和原始列表不同，或两者皆有，最简单的控制函数就是 `id`，原样返回未修改的列表。

为了应付多种情况，我们正在尝试改变部分表达，为了替代精心刻画的函数类型 `InfoP`，我们将使用一个普通代数数据类型来表达相同的含义

```
-- file: ch09/ControlledVisit.hs
data Info = Info {
  infoPath :: FilePath
  , infoPerms :: Maybe Permissions
  , infoSize :: Maybe Integer
  , infoModTime :: Maybe ClockTime
} deriving (Eq, Ord, Show)

getInfo :: FilePath -> IO Info
```

记录语法给我们自由控制函数的权限，如 `infoPath`，`traverse` 函数中的这种类型是简单地，正如我们之前期望的那样，如果需要一个文件或者目录的信息，就调用 `getInfo` 函数：

```
-- file: ch09/ControlledVisit.hs
traverse :: ([Info] -> [Info]) -> FilePath -> IO [Info]
```

`traverse` 的定义很短，但很有分量：

```
-- file: ch09/ControlledVisit.hs
traverse order path = do
  names <- getUsefulContents path
  contents <- mapM getInfo (path : map (path </>) names)
  liftM concat $ forM (order contents) $ \info -> do
```

 v: latest ▾

```

if isDirectory info && infoPath info /= path
then traverse order (infoPath info)
else return [info]

getUsefulContents :: FilePath -> IO [String]
getUsefulContents path = do
  names <- getDirectoryContents path
  return (filter (`notElem` [".", ".."]) names)

isDirectory :: Info -> Bool
isDirectory = maybe False searchable . infoPerms

```

现在不再引入新技术，这就是我们遇到的最深奥的函数定义，一行行的深入他，解释它每行为何是这样，不过开始部分的那几行没什么神秘的，它们只是之前看到代码的拷贝。

观察变量 `contents` 的时候情况变得有趣起来，从左到右仔细阅读，已经知道 `names` 是目录实体的列表，同时确定当前目录的所有元素都在这个列表中，这时通过 `mapM` 将 `getInfo` 附加到结果返回的路径上。

接下来的这一行更深奥，继续从左往右看，我们看到本行的最后一个元素以一个匿名函数的定义开始，并持续到这一段的结尾，给定一个 `Info` 值，函数或者递归访问一个目录（有额外的方法保证我们不在访问这个路径），或者返回当前值作为列表唯一元素的列表（来匹配递归的返回类型）。

函数通过 `forM` 获得 `order` 返回 `info` 列表中的每个元素，`forM` 是使用者提供的递归控制函数。

本行的新上下文中使用提升技术，`liftM` 函数需要一个规则函数，`concat`，并且提升到 `IO` 的 `monad` 操作，换言之，他需要 `forM` 通过 `IO monad` 操作的返回值，并将 `concat` 附加其上（获得一个 `[Info]` 类型的返回值，这也是我们所需要的）并将结果值返回给 `IO monad`。

最后不要忘记定义 `getInfo` 函数：

```

-- file: ch09/ControlledVisit.hs
maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\_ -> return Nothing) (Just `liftM` act)

getInfo path = do
  perms <- maybeIO (getPermissions path)
  size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)
  modified <- maybeIO (getModificationTime path)
  return (Info path perms size modified)

```

在此唯一值得记录的事情是一个有用的组合字，`maybeIO`，将一个可能抛出异常的 `IO` 操作转换成用 `Maybe` 包装的结果

练习

1. 在以代数顺序遍历一个目录树时如何确定需要通过的内容。
2. 使用 `id` 作为控制函数，`traverse id` 扮演一个前序递归树，在子目录之前他返回一个父目录，写一个控制函数让 `traverse` 表现为一个后序遍历，返回子目录在父目录之前。
3. 使得《谓词组合》一节里面的断言和组合字可以处理新的 `info` 类型。
4. 给 `traverse` 写一个包装器，让你通过谓词控制递归，并通过谓词过滤返回结果

代码深度，可读性和学习过程

`traverse` 这样深度的代码在 `Haskell` 中并不多见，在这种表达方式中里学习的收获是巨大的，同时也并不需要大量的练习才能以这种方式流利的阅读和写作代码：

```

-- file: ch09/ControlledVisit.hs
traverseVerbose order path = do
  names <- getDirectoryContents path
  let usefulNames = filter (`notElem` [".", ".."]) names
  contents <- mapM getEntryName ("":usefulNames)
  recursiveContents <- mapM recurse (order contents)
  return (concat recursiveContents)
where getEntryName name = getInfo (path </> name)
      isDirectory info = case infoPerms info of
        Nothing -> False

```

 v: latest ▾

```

Just perms -> searchable perms

recurse info = do
  if isDirectory info && infoPath info /= path
    then traverseVerbose order (infoPath info)
    else return [info]

```

作为对比，这里有一个不那么复杂的代码，这也许适合一个对Haskell了解不那么深入的程序员

这里所做的一切都是创建一个新的替代，通过部分应用（partial application）和函数组合（function composition）替代liberally，在where块中我们已经定义了一些本地函数，在maybe组合子中，使用了case表达式，为了替代liftM，我们手动将concat提升。

并不是说深度是一个不好的特征，traverse函数的每一行原始代码都很短，我们引入一个本地变量和本地函数来保证代码干净且足够短，命名注意可读性，同时使用函数组合管道，最长的管道只含有三个元素。

编写可维护的Haskell代码核心是找到深度和可读性的折中，能否做到这点取决于你的实践层次：

成为Haskell程序员之前，Andrew并不知道使用标准库的方式，为此付出的代价则是写了一大堆不必要的重复代码。

Zack是一个有数月编程经验的，并且精通通过(.)组合长管道的技巧。每当代码需要改动，就需要重构一个管道，他无法更深入的理解已经存在的管道的意义，而这些管道也太脆弱而无法修正。

Monica有相当时间的编程经验，他对Haskell库和编写整洁的代码非常熟悉，但他避免使用高深度的风格，她的代码可维护，同时她还找到了一种简单地方法来面对快速的需求变更

观察迭代函数的另一种方法

相比原始的betterFind函数，迭代函数给我们更多控制权的同时仍存在一个问题，我们可以避免递归目录，但我们不能过滤其他文件名直到我们获得整个名称树，如果递归含有100000个文件的目录的同时只关注其中三个，在获得这三个需要的文件名之前需要给出一个含有10000个元素的表。

一个可能的方法是提供一个过滤器作为递归的新参数，我们将它应用到生成的名单中，这将允许我们获得一个只包含我们需要元素的列表

然而，这个方法也存在缺点，假如说我们知道需要比三个多很多的实体组，并且这些实体组是这10000个我们需要遍历实体中的前几个，这种情况下就不需要访问剩下的实体，这并不是个故弄玄虚的问题，举个栗子，邮箱文件夹中存放了包含许多邮件信息的文件夹——就像一个有大量文件的目录，那么代表邮箱的目录含有数千个文件就很正常。

从另一个角度看，我们尝试定位之前两个遍历函数的弱点：我们如何看待文件系统遍历阶级目录下的一个文件夹？

相似的文件夹，foldr和foldl'，干净的生成遍历并计算出一个结果，很难把这个想法从列表扩展到目录树，但我们仍乐于在fold中加入一个控制元素，我们将这个控制表达为一个代数数据类型：

```

-- file: ch09/FoldDir.hs
data Iterate seed = Done      { unwrap :: seed }
                  | Skip      { unwrap :: seed }
                  | Continue { unwrap :: seed }
                  deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed

```

Iterator类型给函数一个便于使用的别名，它需要一个种子和一个info值来表达这个目录实体，并返回一个新种子和一个我们fold函数的说明，这个说明通过Iterate类型的构造器来表达：

如果这个构造器已经完成，遍历将立即释放，被Done包裹的值将作为结果返回。

如果这个说明被跳过，并且当前info代表一个目录，遍历将不在递归寻找这个目录。

其他，这个便利仍将继续，使用包裹值作为下一个调用fold函数的参数。

目录逻辑上是左序的，因为我们开始从我们第一个遇到的实体开始fold操作，而每步中的种子是之前一步的结果。

```

-- file: ch09/FoldDir.hs
foldTree :: Iterator a -> a -> FilePath -> IO a

foldTree iter initSeed path = do
  endSeed <- fold initSeed path
  return (unwrap endSeed)
where
  fold seed subpath = getUsefulContents subpath >>= walk seed

```

 v: latest ▾

```
walk seed (name:names) = do
  let path' = path </> name
  info <- getInfo path'
  case iter seed info of
    done@(Done _) -> return done
    Skip seed'    -> walk seed' names
    Continue seed'
      | isDirectory info -> do
          next <- fold seed' path'
          case next of
            done@(Done _) -> return done
            seed''        -> walk (unwrap seed'') names
      | otherwise -> walk seed' names
  walk seed _ = return (Continue seed)
```

这部分代码中有意思的部分很少，开始是通过 `scoping` 避免通过额外的参数，最高层 `foldTree` 函数只是 `fold` 的包装器，用来揭开 `fold` 的最后结果的生成器。

由于 `fold` 是本地函数，我们不需要通过 `foldTree` 的 `iter` 变量来进入他，可以从外部进入，相似的，`walk` 也可以在外看到 `path`。

另一个需要指出的是 `walk` 是一个尾递归，在我们最初的函数中用来替代一个匿名函数调用。通过外部控制，可以在任何需要的时候停止，这使得当 `iterator` 返回 `Done` 的时候就可以退出。

即使 `fold` 调用 `walk`，`walk` 调用 `fold` 这样的递归来遍历子目录，每个函数返回一个用 `Iterate` 包装起来的种子，当 `fold` 被调用，并且返回，`walk` 检查返回并观察需要继续还是退出，通过这种方式，一个 `Done` 的返回直接终止两个函数中的所有递归调用。

实践中一个 `iterator` 像什么，下面是一个观察三个位图文件（至多）的同时并不逆向递归元数据目录的复杂例子：

```
-- file: ch09/FoldDir.hs
atMostThreePictures :: Iterator [FilePath]

atMostThreePictures paths info
  | length paths == 3
  = Done paths
  | isDirectory info && takeFileName path == ".svn"
  = Skip paths
  | extension `elem` [".jpg", ".png"]
  = Continue (path : paths)
  | otherwise
  = Continue paths
where extension = map toLower (takeExtension path)
      path = infoPath info
```

为了使用这个需要调用 `foldTree atMostThreePictures []`，它给我们一个 `IO [FilePath]` 类型的返回值。

当然，`iterators` 并不需要如此复杂，下面是个对目录进行计数的代码：

```
-- file: ch09/FoldDir.hs
countDirectories count info =
  Continue (if isDirectory info
    then count + 1
    else count)
```

传递给 `foldTree` 的初始种子（seed）为数字零。

练习

1. 修正 `foldTree` 来允许调用改变遍历目录实体的顺序。
2. `foldTree` 函数展示了前序遍历，将它修正为允许调用方决定便利顺序。
3. 写一个组合子的库允许 `foldTree` 接收不同类型的 `iterators`，你能写出更简洁的 `iterators` 吗？

代码指南

 v: latest ▾

有许多好的Haskell程序员的习惯来自经验，我们有一些通用的经验给你，这样你可以更快的写出易于阅读的代码。

正如已经提到的，Haskell中永远使用空格，而不是tab。

如果你发现代码里有个片段聪明到炸裂，停下来，然后思考下如果你离开代码一个月是否还能懂这段代码。

常规命名类型和变量一般是骆驼法，例如 `myVariableName`，这种风格在Haskell中也同样流行，不要去想你的其他命名习惯，如果你遵循一个不标准的惯例，那么你的代码将会对其他人的眼睛造成折磨。

即使你已经用了Haskell一段时间，在你写小函数之前花费几分钟的时间查阅库函数，如果标准库并没有提供你需要的函数，你可能需要组合出一个新的函数来获得你想要的结果。

组合函数的长管道难以阅读，长意味着包含三个以上元素的序列，如果你有这样一个管道，使用 `let` 或者 `where` 语句块将它分解成若干小部分，给每个管道元素一个有意义的名字，然后再将他们回填到代码，如果你想不出一个有意义的名字，问下自己 能不能解释这段代码的功能，如果不能，简化你的代码。

即使在编辑器中很容易格式化长于八十列的代码，宽度仍然是个重要问题，宽行在80行之外的内容通常会被截断，这非常伤害可读性，每一行不超过八十个字符，这样你就可以写入单独的一行，这帮助你保持每一行代码不那么复杂，从而更容易被人读懂。

常用布局风格

只要你的代码遵守布局规范，那么他并不会给人一团乱麻的感觉，因此也不会造成误解，也就是说，有些布局风格是常用的。

`in` 关键字通常正对着 `let` 关键字，如下所示：

```
-- file: ch09/Style.hs
tidyLet = let foo = undefinedwei's
          bar = foo * 2
          in undefined
```

单独列出 `in` 或者让 `in` 在一系列等式之后跟着的写法都是正确的，但下面这种写法则会显得很奇怪：

```
-- file: ch09/Style.hs
weirdLet = let foo = undefined
           bar = foo * 2
           in undefined

strangeLet = let foo = undefined
              bar = foo * 2 in
              undefined
```

与此相反，让 `do` 在行尾跟着而非在行首单独列出：

```
-- file: ch09/Style.hs
commonDo = do
  something <- undefined
  return ()

-- not seen very often
rareDo =
  do something <- undefined
  return ()
```

括号和分号即使合法也很少用到，他们的使用并不存在问题，只是让代码看起来奇怪，同时让Haskell写成的代码不必遵守排版规则。

```
-- file: ch09/Style.hs
unusualPunctuation =
  [ (x,y) | x <- [1..a], y <- [1..b] ] where {
                                     b = 7;

a = 6 }

preferredLayout = [ (x,y) | x <- [1..a], y <- [1..b] ]
  where b = 7
        a = 6
```

 v: latest ▾

如果等式的右侧另起一行，通常在和他本行内，相关变量名或者函数定义的下方之前留出一些空格。

```
-- file: ch09/Style.hs
normalIndent =
    undefined

strangeIndent =
    undefined
```

空格缩进的数量有多种选择，有时候在一个文件中，二，三，四格缩进都很正常，一个缩进也合法，但不常用，而且容易被误读。

写 `where` 语句的缩进时，最好让它分辨起来比较容易：

```
-- file: ch09/Style.hs
goodWhere = take 5 lambdas
    where lambdas = []

alsoGood =
    take 5 lambdas
    where
        lambdas = []


badWhere =
    take 5 lambdas
    where
        lambdas = []
-- legal, but ugly and hard to read
```

练习

即使本章内容指导你们完成文件查找代码，但这并不意味着真正的系统编程，因为haskell移植的 `IO` 库并不暴露足够的消息给我们写有趣和复杂的查询。

1. 把本章代码移植到你使用平台的 `api` 上，`System.Posix` 或者 `System.Win32`。
2. 加入查找文件所有者的功能，将这个属性对谓词可见。

讨论



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

Pearls of Functional Algorithm Design

1条评论 • 6年前

Tonghua Su — where is the content?
- 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 —
- 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。
- 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