

第八章：高效文件处理、正则表达式、文件名匹配

高效文件处理

下面是个简单的基准测试，读取一个由数字构成的文本文件，并打印它们的和。

```
-- file: ch08/SumFile.hs
main = do
  contents <- getContents
  print (sumFile contents)
  where sumFile = sum . map read . words
```

尽管读写文件时，默认使用 `String` 类型，但它并不高效，所以这样简单的程序效率会很糟糕。

一个 `String` 代表一个元素类型为 `Char` 的列表；列表的每个元素被单独分配内存，并有一定的写入开销。对那些要读取文本及二进制数据的程序来说，这些因素会影响内存消耗和执行效率。在这个简单的测试中，即使是 `Python` 那样的解释型语言的表现也会大大好于使用 `String` 的 `Haskell` 代码。

`bytestring` 库是 `String` 类型的一个快速、经济的替代品。在保持 `Haskell` 代码的表现力和简洁的同时，使用 `bytestring` 编写的代码在内存占用和执行效率经常可以达到或超过 `C` 代码。

这个库提供两个模块。每个都定义了与 `String` 类型上函数对应的替代物。

`Data.ByteString` 定义了一个名为 `ByteString` 的严格类型，其将一个字符串或二进制数据或文本用一个数组表示。
`Data.ByteString.Lazy` 模块定义了一个惰性类型，同样命名为 `ByteString`。其将字符串数据表示为一个由块组成的列表，每个块是大小为 64KB 的数组。

这两种 `ByteString` 适用于不同的场景。对于大体积的文件流(几百 MB 至几 TB)，最好使用惰性的 `ByteString`。其块的大小被调整得对现代 CPU 的 L1 缓存特别友好，并且在流中已经被处理过块可以被垃圾收集器快速丢弃。

对于不在意内存占用而且需要随机访问的数据，最好使用严格的 `ByteString` 类型。

二进制 I/O 和有限载入

让我们来开发一个小函数以说明 `ByteString` API 的一些用法。我们将检测一个文件是否是 ELF object 文件：这种文件类型几乎被所有现代类 Unix 系统作为可执行文件。

这个问题可以通过查看文件头部的四个字节解决，看他们是否匹配某个特定的字节序列。表示某种文件类型的字节序列通常被称为 魔法数。

```
-- file: ch08/ElfMagic.hs
import qualified Data.ByteString.Lazy as L

hasElfMagic :: L.ByteString -> Bool
hasElfMagic content = L.take 4 content == elfMagic
  where elfMagic = L.pack [0x7f, 0x45, 0x4c, 0x46]
```

我们使用 `Haskell` 的 有限载入 语法载入 `ByteString` 模块，像上面 `import qualified` 那句那样。这样可以把一个模块关联到另一个我们选定的名字。

例如，使用到惰性 `ByteString` 模块的 `take` 函数时，要写成 `L.take`，因为我们将这个模块载入到了 `L` 这个名字下。若没有明确指明使用哪个版本的函数，如此处的 `take`，编译器会报错。

我们将一直使用有限载入语法使用 `ByteString` 模块，因为其中提供的很多函数与 `Prelude` 模块中的函数重名。

Note

有限载入使得可以方便地切换两种 `ByteString` 类型。只需要在代码的头部改变 `import` 声明；剩余的代码可能不需要任何修改。你可以方便地比较两种类型，以观察哪种类型更符合你程序的需要。

 v: latest ▼

无论是否使用有限载入，始终可以使用模块的全名来识别某些混淆。例如，`Data.ByteString.Lazy.length` 和 `L.length` 表示相同的函数，`Prelude.sum` 和 `sum` 也是如此。

`ByteString` 模块为二进制 I/O 而设计。`Haskell` 中表达字节的类型是 `Word8`；如果需要按名字引用它，需要将其从 `Data.Word` 模块载入。

`L.pack` 函数接受一个由 `Word8` 组成的列表，并将其装入一个惰性 `ByteString`（`L.unpack` 函数的作用恰好相反。）。`hasElfMagic` 函数简单地将一个 `ByteString` 的前四字节与一个魔法数相比较。

我们使用了典型的 `Haskell` 风格编写 `hasElfMagic` 函数，其并不执行 I/O。这里是如何在真正的文件上使用它。

```
-- file: ch08/ElfMagic.hs
isElfFile :: FilePath -> IO Bool
isElfFile path = do
    content <- L.readFile path
    return (hasElfMagic content)
```

`L.readFile` 函数是 `readFile` 的惰性 `ByteString` 等价物。它是惰性执行的，将文件读取为数据是需要的。它也很高效，立即读取 64KB 大小的块。对我们的任务而言，惰性 `ByteString` 是一个好选择，我们可以安全的将这个函数应用在任意大小的文件上。

文本 I/O

方便起见，`bytestring` 库提供两个具有有限文本 I/O 功能的模块，`Data.ByteString.Char8` 和 `Data.ByteString.Lazy.Char8`。它们将每个字符串的元素暴露为 `Char` 而非 `Word8`。

Warning

这些模块中的函数适用于单字节大小的 `Char` 值，所以他们仅适用于 `ASCII` 及某些欧洲字符集。大于 255 的值将被截断。

这两个面向字符的 `bytestring` 模块提供了用于文本处理的函数。以下文件包含了一家知名互联网公司在 2008 年中期每个月的股价。

如何在这一系列记录中找到最高收盘价呢？收盘价位于以逗号分隔的第四列。以下函数从单行数据中获取收盘价。

```
-- file: ch08/HighestClose.hs
import qualified Data.ByteString.Lazy.Char8 as L

closing = readPrice . (!!4) . L.split ','
```

这个函数使用 `point-free` 风格编写，我们要从右向左阅读。`L.split` 函数将一个惰性 `ByteString` 按某个分隔符切分为一个由 `ByteString` 组成的列表。`(!!)` 操作符检索列表中的第 `k` 个元素。`readPrice` 函数将一个表示小数的字符串转换为一个数。

```
- file: ch08/HighestClose.hs
readPrice :: L.ByteString -> Maybe Int
readPrice str =
    case L.readInt str of
        Nothing      -> Nothing
        Just (dollars, rest) ->
            case L.readInt (L.tail rest) of
                Nothing      -> Nothing
                Just (cents, more) ->
                    Just (dollars * 100 + cents)
```

我们使用 `L.readInt` 函数来解析一个整数。当发现数字时，它会将一个整数和字符串的剩余部分一起返回。`L.readInt` 在解析失败时返回 `Nothing`，这导致我们的函数稍有些复杂。

查找最高收盘价的函数很容易编写。

```
-- file: ch08/HighestClose.hs
highestClose = maximum . (Nothing:) . map closing . L.lines

highestCloseFrom path = do
    contents <- L.readFile path
    print (highestClose contents)
```

 v: latest ▾

不能对空列表使用 `maximum` 函数，所以我们要耍了点小把戏。

```
ghci> maximum [3,6,2,9]
9
ghci> maximum []
*** Exception: Prelude.maximum: empty list
```

我们想在没有股票数据时也不抛出异常，所以用 `(Nothing::)` 这个表达式来确保输入到 `maximum` 函数的由 `Maybe Int` 值构成的列表总是非空。

```
ghci> maximum [Nothing, Just 1]
Just 1
ghci> maximum [Nothing]
Nothing
```

我们的函数工作正常吗？

```
ghci> :load HighestClose
[1 of 1] Compiling Main             ( HighestClose.hs, interpreted )
Ok, modules loaded: Main.
ghci> highestCloseFrom "prices.csv"
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Just 2741
```

因为我们把逻辑和 I/O 分离开了，所以即使不创建一个空文件也可以测试无数据的情况。

```
ghci> highestClose L.empty
Nothing
```

匹配文件名

很多面向操作系统的编程语言提供了检测某个文件名是否匹配给定模式的库函数，或者返回一个匹配给定模式的文件列表。在其他语言中，这个函数通常叫做 `fmatch`。尽管 Haskell 标准库提供了很多有用的系统编程设施，但是并没有提供这类用于匹配文件名的函数。所以我们可以自己开发一个。

我们需要处理的模式种类通常称为 glob 模式（我们将使用这个术语），通配符模式，或称 shell 风格模式。它们仅是一些简单规则。你可能已经了解了，但是这里将做一个简要的回顾。

Note

对某个模式的匹配从字符串头部开始，在字符串尾部结束。

多数文本字符匹配自身。例如，文本 `foo` 作为模式匹配其自身 `foo`，且在一个输入字符串中仅匹配 `foo`。

`*` (星号) 意味着“匹配所有”；其将匹配所有文本，包括空字符串。例如，模式 `foo*` 将匹配任意以 `foo` 开头的字符串，比如 `foo` 自身，`foobar`，或 `foo.c`。模式 `quux*.c` 将匹配任何以 `quux` 开头且以 `.c` 结束的字符串，如 `quuxbaz.c`。

`?` (问号) 匹配任意单个字符。模式 `pic?.jpg` 将匹配类似 `picaa.jpg` 或 `pic01.jpg` 的文件名。

`[` (左方括号) 将开始定义一个字符类，以 `]` 结束。其意思是“匹配在这个字符类中的任意字符”。`[!]` 开启一个否定的字符类，其意为“匹配不在这个字符类中的任意字符”。

用 `-` (破折号) 连接的两个字符，是一种表示范围的速记方法，表示：“匹配这个范围内的任意字符”。

字符类有一个附加的条件；其不可为空。在 `[` 或 `[!` 后的字符是这个字符类的一部分，所以我们可以编写包含 `]` 的字符类，如 `[aeiou]`。模式 `pic[0-9].[pP][nN][gG]` 将匹配由字符串 `pic` 开始，跟随单个数字，最后是字符串 `.png` 的任意大小写形式。

尽管 Haskell 的标准库没有提供匹配 glob 模式的方法，但它提供了一个良好的正则表达式库。Glob 模式仅是一个从正则表达式中切分出来的略有不同的子集。很容易将 glob 模式转换为正则表达式，但在此之前，我们首先要了解怎样在 Haskell 中使用正则表达式。

Haskell 中的正则表达式

 v: latest ▾

在这一节，我们将假设读者已经熟悉 Python、Perl 或 Java 等其他语言中的正则表达式。

为了简洁，此后我们将“regular expression”简写为 regexp。

我们将以与其他语言对比的方式介绍 Haskell 如何处理 regexp，而非从头讲解何为 regexp。Haskell 的正则表达式库比其他语言具备更加强大的表现力，所以我们有很多可以聊的。

在我们对 regexp 库的探索开始时，只需使用 `Text.Regex.Posix` 工作。一般通过在 `ghci` 进行交互是探索一个模块最方便的办法。

```
ghci> :module +Text.Regex.Posix
```

可能正则表达式匹配函数是我们平时需要使用的唯一的函数，其以中缀运算符 `(=~)`（从 Perl 中借鉴）表示。要克服的第一个障碍是 Haskell 的 regexp 库重度使用了多态。其结果就是，`(=~)` 的类型签名非常难懂，所以我们在此对其不做解释。

`=~` 操作符的参数和返回值都使用了类型类。第一个参数 `(=~` 左侧) 是要被匹配的文本；第二个参数 `(=~` 右侧) 是准备匹配的正则表达式。对每个参数我们都可以使用 `String` 或者 `ByteString`。

结果的多种类型

`=~` 操作符的返回类型是多态的，所以 Haskell 编译器需要一通过一些途径知道我们想获得哪种类型的结果。实际编码中，可以通过我们如何使用匹配结果推导出它的类型。但是当我们通过 `ghci` 进行探索时，缺少类型推导的线索。如果不指明匹配结果的类型，`ghci` 将因其无法获得足够信息对匹配结果进行类型推导而报错。

当 `ghci` 无法推断目标的类型时，我们要告诉它想要哪种类型。若想知道正则匹配是否通过时，需要将结果类型指定为 `Bool` 型。

```
ghci> "my left foot" =~ "foo" :: Bool
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
True
ghci> "your right hand" =~ "bar" :: Bool
False
ghci> "your right hand" =~ "(hand|foot)" :: Bool
True
```

在 regexp 库内部，有一种类型类名为 `RegexContext`，其描述了目标类型的行为。基础库定义了很多这个类型类的实例。`Bool` 型是这种类型类的一个实例，所以我们取回了一个可用的结果。另一个实例是 `Int`，可以描述正则表达式匹配了多少次。

```
ghci> "a star called henry" =~ "planet" :: Int
0
ghci> "honorificabilitudinitatibus" =~ "[aeiou]" :: Int
13
```

如果指定结果类型为 `String`，将得到第一个匹配的子串，或者表示无匹配的空字符串。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: String
"ii"
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: String
""
```

另一个合法的返回值类型是 `[[String]]`，将返回由所有匹配的的字符串组成的列表。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [[String]]
[["ii","ii"],["uu","uu"]]
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [[String]]
[]
```

Warning

注意 `String` 类型的结果

 v: latest ▾

指定结果为普通的字符串时，要当心。因为 `(=)` 在表示“无匹配”时会返回空字符串，很明显这导致了难以处理可以匹配空字符串的正则表达式。这情况出现时，就需要使用另一种不同的结果类型，比如 `[[String]]`。

以上是一些“简单”的结果类型，不过还没说完。在继续讲解之前，我们先来定义一个在之后的例子中共同使用的模式串。可以在 `ghci` 中将这个模式串定义为一个变量，以便节省一些输入操作。

```
ghci> let pat = "(foo[a-z]*bar|quux)"
```

当模式匹配了字符串时，可以获取很多关于上下文的信息。如果指定 `(String,String,String)` 类型的元组作为结果类型，可以获取字符串中首次匹配之前的部分，首次匹配的子串，和首次匹配之后的部分。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String)
("before ", "foodiebar", " after")
```

若匹配失败，整个字符串会作为“首次匹配之前”的部分返回，元组的其他两个元素将为空字符串。

```
ghci> "no match here" =~ pat :: (String,String,String)
("no match here", "", "")
```

使用四元组作为返回结果时，元组的第四个元素是一个包含了模式中所有分组的列表。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String,[String])
("before ", "foodiebar", " after", ["foodiebar"])
```

也可以获得关于匹配结果的数字信息。二元组类型的结果可以表示首次匹配在字符串中的偏移，以及匹配结果的长度。如果使用由这种二元组构成的列表作为结果类型，我们将得到所有字符串中所有匹配的此类信息。

```
ghci> "before foodiebar after" =~ pat :: (Int,Int)
(7,9)
ghci> getAllMatches ("i foobarbar a quux" =~ pat) :: [(Int,Int)]
[(2,9), (14,4)]
```

二元组的首个元素（表示偏移的那个），其值为 `-1` 时，表示匹配失败。当指定返回值为列表时，空表表示失败。

```
ghci> "eleemosynary" =~ pat :: (Int,Int)
(-1,0)
ghci> "mondegreen" =~ pat :: [(Int,Int)]
[]
```

以上并非 `RegexContext` 类型类的内置实例的完整清单。完整的清单可以在 `Text.Regex.Base.Context` 模块的文档中找到。

使函数具有多态返回值的能力对于一个静态类型语言来说是个不同寻常的特性。

进一步了解正则表达式

不同类型字符串的混合与匹配

之前提到过，`=~` 操作符的输入和返回值都使用了类型类。我们可以在正则表达式和要匹配的文本中使用 `String` 或者严格的 `ByteString` 类型。

```
ghci> :module +Data.ByteString.Char8
ghci> :type pack "foo"
pack "foo" :: ByteString
```

我们可以尝试不同的 `String` 和 `ByteString` 组合。

```
ghci> pack "foo" =~ "bar" :: Bool
False
ghci> "foo" =~ pack "bar" :: Int
```

 v: latest ▾

```
0
ghci> getAllMatches (pack "foo" =~ pack "o") :: [(Int, Int)]
[(1, 1), (2, 1)]
```

不过，我们需要注意，文本匹配的结果必须于被匹配的字符串类型一致。让我们实践一下，看这是什么意思。

```
ghci> pack "good food" =~ ".ood" :: [[ByteString]]
[["good"], ["food"]]
```

上面的例子中，我们使用 `pack` 将一个 `String` 转换为 `ByteString`。这种情况可以通过类型检查，因为 `ByteString` 也是一种合法的结果类型。但是如果输入字符串类型为 `String` 类型，在尝试获得 `ByteString` 类型结果时将会失败。

```
ghci> "good food" =~ ".ood" :: [[ByteString]]

<interactive>:55:13:
  No instance for (RegexContext Regex [Char] [[ByteString]])
    arising from a use of ‘=~’
  In the expression: "good food" =~ ".ood" :: [[ByteString]]
  In an equation for ‘it’ :
    it = "good food" =~ ".ood" :: [[ByteString]]
```

将结果类型指定为与被匹配字符串相同的 `String` 类型就可以轻松地解决这个问题。

```
ghci> "good food" =~ ".ood" :: [[String]]
[["good"], ["food"]]
```

对于正则表达式不存在这个限制。正则表达式可以是 `String` 或 `ByteString`，而不必在意输入或结果是何种类型。

你要知道的其他一些事情

查阅 Haskell 的库文档，会发现很多和正则表达式有关的模块。`Text.Regex.Base` 下的模块定义了供其他所有正则表达式库使用的通用 API。可以同时安装许多不同实现的正则表达式模块。写作本书时，GHC 自带一个实现，`Text.Regex.Posix`。正如其名字，这个模块提供了 POSIX 语义的正则表达式实现。

Note

Perl 风格和 POSIX 风格的正则表达式

如果你此前用过其他语言，如 Perl，Python，或 Java，并且使用过其中的正则表达式，你应该知道 `Text.Regex.Posix` 模块处理的 POSIX 风格的正则表达式与 Perl 风格的正则表达式有一些显著的不同。

当有多个匹配结果候选时，Perl 的正则表达式引擎表现为左侧最小匹配，而 POSIX 引擎会选择贪婪匹配（最长匹配）。当使用正则表达式 `(foo|fo*)` 匹配字符串 `foooooo` 时，Perl 风格引擎将返回 `foo` (最左的匹配)，而 POSIX 引擎将返回的结果将包含整个字符串 (贪婪匹配)。

POSIX 正则表达式比 Perl 风格的正则表达式缺少一些格式语法。它们也缺少一些 Perl 风格正则表达式的功能，比如零宽度断言和对贪婪匹配的控制。

Hackage 上也有其他 Haskell 正则表达式包可供下载。其中一些比内置的 POSIX 引擎拥有更好的执行效率 (如 `regex-tdfa`)；另外一些提供了大多数程序员熟悉的 Perl 风格正则匹配 (如 `regex-pcre`)。它们都按照我们这节提到的 API 编写。

将 glob 模式翻译为正则表达式

我们已经看到了用正则表达式匹配文本的多种方法，现在让我们将注意力回到 glob 模式。我们要编写一个函数，接收一个 glob 模式作为输入，返回其对应的正则表达式。glob 模式和正则表达式都以文本字符串表示，所以这个函数的类型应该已经清楚了。

```
-- file: ch08/GlobRegex.hs
module GlobRegex
(
  globToRegex
, matchesGlob
```

 v: latest ▾


```

    ) where

import Text.Regex.Posix ((=))

globToRegex :: String -> String

```

我们生成的正则表达式必须被锚定，所以它要对一个字符串从头到尾完整匹配。

```

-- file: ch08/GlobRegex.hs
globToRegex cs = '^' : globToRegex' cs ++ "$"

```

回想一下，`String` 仅是 `[Char]` 的同义词，一个由字符组成的数组。`:` 操作符将一个值加入某个列表头部，此处是将字符 `^` 加入 `globToRegex'` 函数返回的列表头部。

Note

在定义之前使用一个值

Haskell 在使用某个值或函数时，并不需要其在之前的源码中被声明。在某个值首次被使用之后才定义它是很平常的。Haskell 编译器并不关心这个层面上的顺序。这使我们用最符合逻辑的方式灵活地组织代码，而不是为使编译器作者更轻松而遵守某种顺序。

Haskell 模块的作者们经常利用这种灵活性，将“更重要的”代码放在源码文件更靠前的位置，将繁琐的实现放在后面。这也是我们实现 `globToRegex'` 函数及其辅助函数的方法。

`globToRegex'` 将使用正则表达式做大部分的翻译工作。我们将使用 Haskell 的模式匹配特性轻松地穷举出需要处理的每一种情况

```

-- file: ch08/GlobRegex.hs

globToRegex' :: String -> String
globToRegex' "" = ""

globToRegex' ('*':cs) = "." ++ globToRegex' cs

globToRegex' ('.':cs) = '.' : globToRegex' cs

globToRegex' ('[':c:cs) = "[" ++ c : charClass cs
globToRegex' ('[':c:cs) = '[' : c : charClass cs
globToRegex' ('[':_) = error "unterminated character class"

globToRegex' (c:cs) = escape c ++ globToRegex' cs

```

我们的第一条规则是，如果触及 `glob` 模式的尾部（也就是说当输入为空字符串时），我们返回 `$`，正则表达式中表示“匹配行尾”的符号。我们按照这样一系列规则将模式串由 `glob` 语法转化为正则表达式语法。最后一条规则匹配所有字符，首先将可转义字符进行转义。

`escape` 函数确保正则表达式引擎不会将普通字符串解释为构成正则表达式语法的字符。

```

-- file: ch08/GlobRegex.hs
escape :: Char -> String
escape c | c `elem` regexChars = '\\' : [c]
         | otherwise = [c]
  where regexChars = "\\+()$.{}|\""

```

`charClass` 辅助函数仅检查一个字符类是否正确地结束。这个并不改变其输入，直到遇到一个 `]` 字符，其将控制流交还给 `globToRegex'`

```

-- file: ch08/GlobRegex.hs
charClass :: String -> String
charClass (']':cs) = ']' : globToRegex' cs
charClass (c:cs) = c : charClass cs
charClass [] = error "unterminated character class"

```

 v: latest ▾

现在我们已经完成了 `globToRegex` 函数及其辅助函数的定义，让我们在 `ghci` 中装载并且实验一下。

```
ghci> :load GlobRegex.hs
[1 of 1] Compiling GlobRegex      ( GlobRegex.hs, interpreted )
Ok, modules loaded: GlobRegex.
ghci> :module +Text.Regex.Posix
ghci> globToRegex "f??.c"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
"^f..\\..c$"
```

果然，看上去像是一个合理的正则表达式。可以使用她来匹配某个字符串码？

```
ghci> "foo.c" =~ globToRegex "f??.c" :: Bool
True
ghci> "test.c" =~ globToRegex "t[ea]s*" :: Bool
True
ghci> "taste.txt" =~ globToRegex "t[ea]s*" :: Bool
True
```

奏效了！现在让我们在 ghci 里玩耍一下。我们可以临时定义一个 `fnmatch` 函数，并且试用它。

```
ghci> let fnmatch pat name = name =~ globToRegex pat :: Bool
ghci> :type fnmatch
fnmatch :: (RegexLike Regex source1) => String -> source1 -> Bool
ghci> fnmatch "d*" "myname"
False
```

但是 `fnmatch` 没有真正的“Haskell 味道”。目前为止，最常见的 Haskell 风格是赋予函数具有描述性的，“驼峰式”命名。将单词连接为驼峰状，首字母小写后面每个单词的首字母大写。例如，“file name matches”这几个词将转换为 `fileNameMatch` 这个名字。“驼峰式”这种说法来自与大写字母形成的“驼峰”。在我们的库中，将使用 `matchesGlob` 这个函数名。

```
-- file: ch08/GlobRegex.hs
matchesGlob :: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat
```

你可能注意到目前为止我们使用的都是短变量名。从经验来看，描述性的名字在更长的函数定义中更有用，它们有助于可读性。对一个仅有两行的函数来说，长变量名价值较小。

练习

1. 使用 `ghci` 探索当你向 `globToRegex` 传入一个畸形的模式时会发生什么，如 `[`。编写一个小函数调用 `globToRegex`，向其传入一个畸形的模式。发生了什么？
2. Unix 的文件系统的文件名通常是对大小写敏感的（如：“G”和“g”不同），Windows 文件系统则不是。为 `globToRegex` 和 `matchesGlob` 函数添加一个参数，以控制它们是否大小写敏感。

重要的题外话：编写惰性函数

在命令式语言中，`globToRegex` 通常是个被我们写成循环的函数。举个例子，Python 标准库中的 `fnmatch` 模块包括了一个名叫 `translate` 的函数与我们的 `globToRegex` 函数做了完全相同的工作。它就被写成一个循环。

如果你了解过函数式编程语言比如 Scheme 或 ML，可能有个概念已经深入你的脑海，“模拟一个循环的方法是使用尾递归”。

观察 `globToRegex'`，可以发现其不是一个尾递归函数。至于原因，重新检查一下它的最后一组规则（它的其他规则也类似）。

```
-- file: ch08/GlobRegex.hs
globToRegex' (c:cs) = escape c ++ globToRegex' cs
```

其递归地执行自身，并以递归执行的结果作为 `(++)` 函数的参数。因为递归执行并不是这个函数的最后一个操作，所以 `globToRegex'` 不是尾递归函数。

 v: latest ▼

为何我们的函数没有定义成尾递归的？答案是 Haskell 的非严格求值策略。在我们开始讨论它之前，先快速的了解一下为什么，传统编程语言中，这类递归定义是我们要避免的。这里有一个简化的 `(++)` 操作符定义。它是递归的，但不是尾递归的。

```
-- file: ch08/append.hs
(+++) :: [a] -> [a] -> [a]

(x:xs) ++ ys = x : (xs ++ ys)
[]      ++ ys = ys
```

在严格求值语言中，如果我们执行 `“foo” ++ “bar”`，将马上构建并返回整个列表。非严格求值将这项工作延后很久执行，知道其结果在某处被用到。

如果我们需要 `“foo” ++ “bar”` 这个表达式结果中的一个元素，函数定义中的第一个模式被匹配，返回表达式 `x : (xs ++ ys)`。因为 `(:)` 构造器是非严格的，`xs ++ ys` 的求值被延迟到当我们需要生成更多结果中的元素时。当生成了结果中的更多元素，我们不再需要 `x`，垃圾收集器可以将其回收。因为我们按需要计算结果中的元素，且不保留已经计算出的结果，编译器可以用常数空间对我们的代码求值。

利用我们的模式匹配器

有一个函数可以匹配 glob 模式很好，但我们希望可以在实际中使用它。在类 Unix 系统中，glob 函数返回一个由匹配给定 glob 模式串的文件和目录组成的列表。让我们用 Haskell 构造一个类似的函数。按 Haskell 的描述性命名规范，我们将这个函数称为 `namesMatching`。

```
-- file: ch08/Glob.hs
module Glob (namesMatching) where
```

我们将 `namesMatching` 指定为我们的 Glob 模块中唯一对用户可见的名字。

```
-- file: ch08/Glob.hs
import System.Directory (doesDirectoryExist, doesFileExist,
                        getCurrentDirectory, getDirectoryContents)
```

`System.FilePath` 抽象了操作系统路径名称的惯例。(`</>`) 函数将两个部分组合为一个路径。

```
ghci> :m +System.FilePath
ghci> "foo" </> "bar"
Loading package filepath-1.1.0.0 ... linking ... done.
"foo/bar"
```

The name of the `dropTrailingPathSeparator` function is perfectly descriptive. No comments `dropTrailingPathSeparator` 函数的名字完美地描述了其作用。


```
ghci> dropTrailingPathSeparator "foo/"
"foo"
```

`splitFileName` 函数以路径中的最后一个斜线将路径分割为两部分。

```
ghci> splitFileName "foo/bar/Quux.hs"
("foo/bar/", "Quux.hs")
ghci> splitFileName "zippity"
("", "zippity")
```

配合 `Systems.FilePath` 和 `Systems.Directory` 两个模块，我们可以编写一个在类 Unix 和 Windows 系统上都可以运行的可移植的 `namesMatching` 函数。

```
-- file: ch08/Glob.hs
import System.FilePath (dropTrailingPathSeparator, splitFileName, (</>))
```

在这个模块中，我们将模拟一个 `“for”` 循环；首次尝试在 Haskell 中处理异常；当然还会用到我们刚写的 `matchesGlob` 函数  [v: latest](#) ▼

```

-- file: ch08/Glob.hs
import Control.Exception (handle, SomeException)
import Control.Monad (forM)
import GlobRegex (matchesGlob)

```

目录和文件存在于各种带有副作用的活动的“真实世界”，我们的 glob 模式处理函数的返回值类型中将必须带有 IO。

如果的输入字符串中不包含模式字符，我们简单的在文件系统中检查输入的名字是否已经建立。（注意，此处使用 Haskell 的 guard 语法可以编写精细整齐的定义。“if”语句也可以做到，但是在美学上不能令人满意。）

```

-- file: ch08/Glob.hs
isPattern :: String -> Bool
isPattern = any (`elem` "[?*")

namesMatching pat
  | not (isPattern pat) = do
    exists <- doesNameExist pat
    return (if exists then [pat] else [])

```

doesNameExist 是一个我们将要简要定义的函数的名字。

如果字符串是一个 glob 模式呢？继续定义我们的函数。

```

-- file: ch08/Glob.hs
| otherwise = do
  case splitFileName pat of
    ("", baseName) -> do
      curDir <- getDirectory
      listMatches curDir baseName
    (dirName, baseName) -> do
      dirs <- if isPattern dirName
        then namesMatching (dropTrailingPathSeparator dirName)
        else return [dirName]
      let listDir = if isPattern baseName
        then listMatches
        else listPlain
      pathNames <- forM dirs $ \dir -> do
        baseNames <- listDir dir baseName
        return (map (dir </>) baseNames)
      return (concat pathNames)

```

我们使用 splitFileName 将字符串分割为目录名和文件名。如果第一个元素为空，说明我们正在当前目录寻找符合模式的文件。否则，我们必须检查目录名，观察其是否包含模式。若不含模式，我们建立一个只由目录名一个元素组成的列表。如果含有模式，我们列出所有匹配的目录。

Note

注意事项

System.FilePath 模块有点诡异。上面的情况就是一个例子。splitFileName 函数在其返回值的目录名部分的结尾保留了一个斜线。

```

ghci> :module +System.FilePath
ghci> splitFileName "foo/bar"
Loading package filepath-1.1.0.0 ... linking ... done.
("foo/", "bar")

```

If we didn't remember (or know enough) to remove that slash, we'd recurse endlessly in namesMatching, because of the following behaviour of splitFileName. 1 comment 如果忘记（或不够了解）要去掉这个斜线，我们将在 namesMatching 函数中进行无止境的递归匹配，看看后面演示的 splitFileName 的行为你就会明白。

```

ghci> splitFileName "foo/"
("foo/", "")

```

 v: latest ▾

你或许能够想想想象是什么促使我们加入这份注意事项。

最终，我们将每个目录中的匹配收集起来，得到一个由列表组成的列表，然后将它们连接为一个单独的由文件名组成的列表。

上面那个函数中出现的陌生的 `forM` 函数，其行为有些像 “for” 循环：它将其第二个参数（一个动作）映射到其第一个参数（一个列表），并返回由其结果组成的列表。

我们还剩余一些零散的目标需要完成。首先是上面用到过的 `doesNameExist` 函数的定义。`System.Directory` 函数无法检查一个名字是否已经在文件系统中建立。它强制我们明确要检查的是一个文件还是目录。这个 API 设计的很丑陋，所以我们必须在一个函数中完成两次检验。出于效率考虑，我们首先检查文件名，因为文件比目录更常见。

```
-- file: ch08/Glob.hs
doesNameExist :: FilePath -> IO Bool

doesNameExist name = do
  fileExists <- doesFileExist name
  if fileExists
    then return True
    else doesDirectoryExist name
```

还有两个函数需要定义，返回值都是由某个目录下的名字组成的列表。`listMatches` 函数返回由某目录下全部匹配给定 glob 模式的文件名组成的列表。

```
-- file: ch08/Glob.hs
listMatches :: FilePath -> String -> IO [String]
listMatches dirName pat = do
  dirName' <- if null dirName
    then getCurrentDirectory
    else return dirName
  handle (const (return [])) :: (SomeException -> IO [String])
    $ do names <- getDirectoryContents dirName'
        let names' = if isHidden pat
            then filter isHidden names
            else filter (not . isHidden) names
        return (filter (`matchesGlob` pat) names')

isHidden ('.' : _) = True
isHidden _         = False
```

`listPlain` 接收的函数名若存在，则返回由这个文件名组成的单元列表，否则返回空列表。

```
-- file: ch08/Glob.hs
listPlain :: FilePath -> String -> IO [String]
listPlain dirName baseName = do
  exists <- if null baseName
    then doesDirectoryExist dirName
    else doesNameExist (dirName </> baseName)
  return (if exists then [baseName] else [])
```

仔细观察 `listMatches` 函数的定义，将发现一个名为 `handle` 的函数。之前，我们从 `Control.Exception` 模块中将其载入。正如其暗示的那样，这个函数让我们初次体验了 Haskell 中的异常处理。把它扔进 `ghci` 中看我们会发现什么。

```
ghci> :module +Control.Exception
ghci> :type handle
handle :: (Exception -> IO a) -> IO a -> IO a
```

可以看出 `handle` 接受两个参数。首先是一个函数，其接受一个异常值，且有副作用（其返回值类型带有 IO 标签）；这是一个异常处理器。第二个参数是可能会抛出异常的代码。

关于异常处理器，异常处理器的类型限制其必须返回与抛出异常的代码相同的类型。所以它只能选择或是抛出一个异常，或像在我们的例子中返回一个由字符串组成的列表。

 v: latest ▾

`const` 函数接受两个参数；无论第二个参数是什么，其始终返回第一个参数。

```
ghci> :type const
const :: a -> b -> a
ghci> :type return []
return [] :: (Monad m) => m [a]
ghci> :type handle (const (return []))
handle (const (return [])) :: IO [a] -> IO [a]
```

我们使用 `const` 编写异常处理器忽略任何向其传入的异常。取而代之，当我们捕获异常时，返回一个空列表。

本章不会再展开任何异常处理相关的话题。然而还有更多可说，我们将在第 19 章异常处理时重新探讨这个主题。

练习

1. 尽管我们已经编写了一个可移植 `namesMatching` 函数，这个函数使用了我们的大小写敏感的 `globToRegex` 函数。尝试在不改变其类型签名的前提下，使 `namesMatching` 在 Unix 下大小写敏感，在 Windows 下大小写不敏感。

提示：查阅一下 `System.FilePath` 的文档，其中有一个变量可以告诉我们程序是运行在类 Unix 系统上还是在 Windows 系统上。

2. 如果你在使用类 Unix 系统，查阅 `System.Posix.Files` 模块的文档，看是否能找到一个 `doesNameExist` 的替代品。
3. `*` 通配符，仅匹配一个单独目录中的名字。很多 shell 可以提供扩展通配符语法，`**`，其将在所有目录中进行递归匹配。举个例子，`**.*` 意为“在当前目录及其任意深度的子目录下匹配一个 `.c` 结尾的文件名”。实现 `**` 通配符匹配。

通过 API 设计进行错误处理

向 `globToRegex` 传入一个畸形的正则表达式未必会是一场灾难。用户的表达式可能会有输入错误，这时我们更希望得到有意义的报错信息。

当这类问题出现时，调用 `error` 函数会有很激烈的反应（其结果在 Q: 1 这个练习中探索过。）。`error` 函数会抛出一个异常。纯函数式的 Haskell 代码无法处理异常，所以控制流会突破我们的纯函数代码直接交给处于距离最近一层 `IO` 中并且安装有合适的异常处理器的调用者。如果没有安装异常处理器，Haskell 运行时的默认动作是终结我们的程序（如果是在 `ghci` 中，则会打出一条令人不快的错误信息。）

所以，调用 `error` 有点像是拉下了战斗机的座椅弹射手柄。我们从一个无法优雅处理的灾难性场景中逃离，而等我们着地时会撒出很多燃烧着的残骸。

我们已经确定了 `error` 是为灾难情场景准备的，但我们仍旧在 `globToRegex` 中使用它。畸形的输入将被拒绝，但不会导致大问题。处理这种情况有更好的方式吗？

Haskell 的类型系统和库来救你了！我们可以使用内置的 `Either` 类型，在 `globToRegex` 函数的类型签名中描述失败的可能性。

```
-- file: ch08/GlobRegexEither.hs
type GlobError = String

globToRegex :: String -> Either GlobError String
```

`globToRegex` 的返回值将为两种情况之一，或者为 `Left` “出错信息”或者为 `Right` “一个合法正则表达式”。这种返回值类型，强制我们的调用者处理可能出现的错误。（你会发现这是 Haskell 代码中 `Either` 类型最广泛的用途。）

练习

1. 编写一个使用上面那种类型签名的 `globToRegex` 版本。
2. 改变 `namesMatching` 的类型签名，使其可以处理畸形的正则表达式，并使用它重写 `globToRegex` 函数。

Tip

你会发现牵扯到的工作量大得惊人。别怕，我们将在后面的章节介绍更多简单老练的处理错误的方式。

让我们的代码工作

`namesMatching` 函数本身并不是很令人兴奋，但它是一个很有用的构建模块。将它与稍多点的函数组合在一起，就会让我们做出有趣的东西。

这里有个例子。定义一个 `renameWith` 函数，并不简单的重命名一个文件，取而代之，对文件名执行一个函数，并将返回值作为新的文件名。

```
-- file: ch08/Useful.hs
import System.FilePath (replaceExtension)
import System.Directory (doesFileExist, renameDirectory, renameFile)
import Glob (namesMatching)

renameWith :: (FilePath -> FilePath)
            -> FilePath
            -> IO FilePath

renameWith f path = do
    let path' = f path
    rename path path'
    return path'
```

我们再一次通过一个辅助函数使用 `System.Directory` 中难看的文件/目录函数

```
-- file: ch08/Useful.hs
rename :: FilePath -> FilePath -> IO ()

rename old new = do
    isFile <- doesFileExist old
    let f = if isFile then renameFile else renameDirectory
    f old new
```

`System.FilePath` 模块提供了很多有用的函数用于操作文件名。这些函数恰好漏过了我们的 `renameWith` 和 `namesMatching` 函数，所以我们可以通过将他们组合起来的方式来快速的创建新函数。例如，这个简洁的函数修改了 C++ 源码文件的后缀名。

```
-- file: ch08/Useful.hs
cc2cpp =
    mapM (renameWith (flip replaceExtension ".cpp")) =<< namesMatching "*.cc"
```

`cc2cpp` 函数使用了几个我们已经见过多次的函数。`flip` 函数接受另一个函数作为参数，交换其参数的顺序（可以在 `ghci` 中调查 `replaceExtension` 的类型以了解详情）。`=<<` 函数将其右侧动作的结果喂给其左侧的动作。

练习

1. `Glob` 模式解释起来很简单，用 `Haskell` 可以很容易的直接写出其匹配器，正则表达式则不然。试一下编写正则匹配。

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 HASKLL 中文版 上还有


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。

第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前


 Wengel An —
- 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

1条评论 • 6年前

 Tonghua Su — where is the content?