

第十九章：错误处理

无论使用哪门语言，错误处理都是程序员最重要–也是最容易忽视–的话题之一。在Haskell中，你会发现有两类主流的错误处理：“纯”的错误处理和异常。

当我们说“纯”的错误处理，我们是指算法不依赖任何IO Monad。我们通常会利用Haskell富于表现力的数据类型系统来实现这一类错误处理。Haskell也支持异常。由于惰性求值复杂性，Haskell中任何地方都可能抛出异常，但是只会在IO monad中被捕获。在这一章中，这两类错误处理我们都会考虑。

使用数据类型进行错误处理

让我们从一个非常简单的函数来开始我们关于错误处理的讨论。假设我们希望对一系列的数字执行除法运算。分子是常数，但是分母是变化的。可能我们会写出这样一个函数：

```
-- file: ch19/divby1.hs
divBy :: Integral a => a -> [a] -> [a]
divBy numerator = map (numerator `div`)
```

非常简单，对吧？我们可以在 `ghci` 中执行这些代码：

```
ghci> divBy 50 [1,2,5,8,10]
[50,25,10,6,5]
ghci> take 5 (divBy 100 [1..])
[100,50,33,25,20]
```

这个行为跟我们预期的是一致的：50 / 1 得到50，50 / 2 得到25，等等。甚至对于无穷的链表 [1..] 它也是可以工作的。如果有个0溜进去我们的链表中了，会发生什么事呢？

```
ghci> divBy 50 [1,2,0,8,10]
[50,25,*** Exception: divide by zero
```

是不是很有意思？`ghci` 开始显示输出，然后当它遇到零时发生了一个异常停止了。这是惰性求值的作用–它只按需求值。

在这一章里接下来我们会看到，缺乏一个明确的异常处理时，这个异常会使程序崩溃。这当然不是我们想要的，所以让我们思考一下更好的方式来表征这个纯函数中的错误。

使用Maybe

可以立刻想到的一个表示失败的简单的方法是使用 `Maybe`。如果输入链表中任何地方包含了零，相对于仅仅返回一个链表并在失败的时候抛出异常，我们可以返回 `Nothing`，或者如果没有出现零我们可以返回结果的 `Just`。下面是这个算法的实现：

```
-- file: ch19/divby2.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy _ [] = Just []
divBy _ (0:_) = Nothing
divBy numerator (denom:xs) =
  case divBy numerator xs of
    Nothing -> Nothing
    Just results -> Just ((numerator `div` denom) : results)
```

如果你在 `ghci` 中尝试它，你会发现它可以工作：

```
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> divBy 50 [1,2,0,8,10]
Nothing
```

调用 `divBy` 的函数现在可以使用 `case` 语句来观察调用成功与否，就像 `divBy` 调用自己时所做的那样。

 v: latest ▾

Tip

你大概注意到，上面可以使用一个monadic的实现，像这样子：

```
-- file: ch19/divby2m.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy numerator denominators =
  mapM (numerator `safeDiv`) denominators
  where safeDiv _ 0 = Nothing
        safeDiv x y = x `div` y
```

出于简单考虑，在这章中我们会避免使用monadic实现，但是会指出有这种做法。

[译注:Tip中那段代码编译不过]

丢失和保存惰性

使用 `Maybe` 很方便，但是有代价。`divBy` 将不能够再处理无限的链表输入。由于结果是一个 `Maybe [a]`，必须要检查整个输入链表，我们才能确认不会因为存在零而返回 `Nothing`。你可以尝试在之前的例子中验证这一点：

```
ghci> divBy 100 [1..]
*** Exception: stack overflow
```

这里观察到，你没有看到部分的输出；你没得到任何输出。注意到在 `divBy` 的每一步中(除了输入链表为空或者链表开头是零的情况)，每个子序列元素的结果必须先于当前元素的结果得到。因此这个算法无法处理无穷链表，并且对于大的有限链表，它的空间效率也不高。

之前已经说过，`Maybe` 通常是一个好的选择。在这个特殊例子中，只有当我们去执行整个输入的时候我们才知道是否有问题。有时候我们可以提交发现问题，例如，在 `ghci` 中 `tail []` 会生成一个异常。我们可以很容易写一个可以处理无穷情况的 `tail`：


```
-- file: ch19/safetail.hs
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_,xs) = Just xs
```

如果输入为空，简单的返回一个 `Nothing`，其它情况返回结果的 `Just`。由于在知道是否发生错误之前，我们只需要确认链表非空，在这里使用 `Maybe` 不会破坏惰性。我们可以在 `ghci` 中测试并观察跟普通的 `tail` 有何不同：

```
ghci> tail [1,2,3,4,5]
[2,3,4,5]
ghci> safeTail [1,2,3,4,5]
Just [2,3,4,5]
ghci> tail []
*** Exception: Prelude.tail: empty list
ghci> safeTail []
Nothing
```

这里我们可以看到，我们的 `safeTail` 执行结果符合预期。但是对于无穷链表呢？我们不想打印无穷的结果的数字，所以我们用 `take 5 (tail [1..])` 以及一个类似的`saftTail`构建测试：

```
ghci> take 5 (tail [1..])
[2,3,4,5,6]
ghci> case safeTail [1..] of {Nothing -> Nothing; Just x -> Just (take 5 x)}
Just [2,3,4,5,6]
ghci> take 5 (tail [])
*** Exception: Prelude.tail: empty list
ghci> case safeTail [] of {Nothing -> Nothing; Just x -> Just (take 5 x)}
Nothing
```

这里你可以看到 `tail` 和 `safeTail` 都可以处理无穷链表。注意我们可以更好地处理空的输入链表；而不是抛出异常，我们  `v: latest` 返回 `Nothing`。我们可以获得错误处理能力却不会失去惰性。

但是我们如何将它应用到我们的 `divBy` 的例子中呢？让我们思考下现在的情况：失败是单个坏的输入的属性，而不是输入链表自身。那么将失败作为单个输出元素的属性，而不是整个输出链表怎么样？也就是说，不是一个类型为 `a -> [a] -> Maybe [a]` 的函数，取而代之我们使用 `a -> [a] -> [Maybe a]`。这样做的好处是可以保留惰性，并且调用者可以确定是在链表中的哪里出了问题—或者甚至是过滤掉有问题的结果，如果需要的话。这里是一个实现：

```
-- file: ch19/divby3.hs
divBy :: Integral a => a -> [a] -> [Maybe a]
divBy numerator denominators =
  map worker denominators
  where worker 0 = Nothing
        worker x = Just (numerator `div` x)
```

看下这个函数，我们再次回到使用 `map`，这无论对简洁和惰性都是件好事。我们可以在 `ghci` 中测试它，并观察对于有限和无限链表它都可以正常工作：

```
ghci> divBy 50 [1,2,5,8,10]
[Just 50,Just 25,Just 10,Just 6,Just 5]
ghci> divBy 50 [1,2,0,8,10]
[Just 50,Just 25,Nothing,Just 6,Just 5]
ghci> take 5 (divBy 100 [1..])
[Just 100,Just 50,Just 33,Just 25,Just 20]
```

我们希望通过这个讨论你可以明白这点，不符合规范的（正如 `safeTail` 中的情况）输入和包含坏的数据的输入(`divBy` 中的情况)是有区别的。这两种情况通常需要对结果采用不同的处理。

Maybe Monad的用法

回到 *使用Maybe* 这一节，我们有一个叫做 `divby2.hs` 的示例程序。这个例子没有保存惰性，而是返回一个类型为 `Maybe [a]` 的值。用 monadic 风格也可以表达同样的算法。更多信息和 monad 相关背景，参考 **第14章 Monads**。这是我们新的 monadic 风格的算法：

```
-- file: ch19/divby4.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy _ [] = return []
divBy _ (0:_) = fail "division by zero in divBy"
divBy numerator (denom:xs) =
  do next <- divBy numerator xs
  return ((numerator `div` denom) : next)
```

`Maybe monad` 使得这个算法的表示看上去更好。对于 `Maybe monad`，`return` 就跟 `Just` 一样，并且 `fail _ = Nothing`，因此我们看到任何的错误说明的字段串。我们可以用我们在 `divby2.hs` 中使用过的测试来测试这个算法：

```
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> divBy 50 [1,2,0,8,10]
Nothing
ghci> divBy 100 [1..]
*** Exception: stack overflow
```

我们写的代码实际上并不限于 `Maybe monad`。只要简单地改变类型，我们可以让它对于任何 monad 都能工作。让我们试一下：

```
-- file: ch19/divby5.hs
divBy :: Integral a => a -> [a] -> Maybe [a]
divBy = divByGeneric

divByGeneric :: (Monad m, Integral a) => a -> [a] -> m [a]
divByGeneric _ [] = return []
divByGeneric _ (0:_) = fail "division by zero in divByGeneric"
divByGeneric numerator (denom:xs) =
  do next <- divByGeneric numerator xs
  return ((numerator `div` denom) : next)
```

 v: latest ▾

函数 `divByGeneric` 包含的代码 `divBy` 之前所做的一样；我们只是给它一个更通用的类型。事实上，如果不给出类型，这个类型是由 `ghci` 自动推导的。我们还为特定的类型定义了一个更方便的函数 `divBy`。

让我们在 `ghci` 中运行一下。

```
ghci> :l divby5.hs
[1 of 1] Compiling Main             ( divby5.hs, interpreted )
Ok, modules loaded: Main.
ghci> divBy 50 [1,2,5,8,10]
Just [50,25,10,6,5]
ghci> (divByGeneric 50 [1,2,5,8,10]) :: (Integral a => Maybe [a])
Just [50,25,10,6,5]
ghci> divByGeneric 50 [1,2,5,8,10]
[50,25,10,6,5]
ghci> divByGeneric 50 [1,2,0,8,10]
*** Exception: user error (division by zero in divByGeneric)
```

前两个例子产生的输出都跟我们之前看到的一样。由于 `divByGeneric` 没有指定返回的类型，我们要么指定一个，要么让解释器从环境中推导得到。如果我们不指定返回类型，`ghci` 推荐得到 `IO monad`。在第三和第四个例子中你可以看出来。在第四个例子中你可以看到，`IO monad` 将 `fail` 转化成了一个异常。

`mtl` 包中的 `Control.Monad.Error` 模块也将 `Either String` 变成了一个 `monad`。如果你使用 `Either`，你可以得到保存了错误信息的纯的结果，像这样子：

```
ghci> :m +Control.Monad.Error
ghci> (divByGeneric 50 [1,2,5,8,10]) :: (Integral a => Either String [a])
Loading package mtl-1.1.0.0 ... linking ... done.
Right [50,25,10,6,5]
ghci> (divByGeneric 50 [1,2,0,8,10]) :: (Integral a => Either String [a])
Left "division by zero in divByGeneric"
```

这让我们进入到下一个话题的讨论：使用 `Either` 返回错误信息。

使用Either

`Either` 类型跟 `Maybe` 类型类似，除了一处关键的不同：对于错误或者成功（“`Right` 类型”），它都可以携带数据。尽管语言没有强加任何限制，按照惯例，一个返回 `Either` 的函数使用 `Left` 返回值来表示一个错误，`Right` 来表示成功。如果你觉得这样有助于记忆，你可以认为 `Right` 表式正确结果。我们可以改一下前面小节中关于 `Maybe` 时使用的 `divby2.hs` 的例子，让 `Either` 可以工作：

```
-- file: ch19/divby6.hs
divBy :: Integral a => a -> [a] -> Either String [a]
divBy _ [] = Right []
divBy _ (0:_) = Left "divBy: division by 0"
divBy numerator (denom:xs) =
  case divBy numerator xs of
    Left x -> Left x
    Right results -> Right ((numerator `div` denom) : results)
```

这份代码跟 `Maybe` 的代码几乎是完全一样的；我们只是把每个 `Just` 用 `Right` 替换。`Left` 对应于 `Nothing`，但是现在它可以携带一条信息。让我们在 `ghci` 里面运行一下：

```
ghci> divBy 50 [1,2,5,8,10] Right [50,25,10,6,5] ghci> divBy 50 [1,2,0,8,10] Left "divBy: division by 0"
```

为错误定制数据类型

尽管用 `String` 类型来表示错误的原因对今后很有好处，自定义的错误类型通常会更有帮助。使用自定义的错误类型我们可以知道到底是出了什么问题，并且获知是什么动作引发的这个问题。例如，让我们假设，由于某些原因，不仅仅是除0，我们还不想除以10或者20。我们可以像这样子自定义一个错误类型：

```
-- file: ch19/divby7.hs
data DivByError a = DivBy0
                  | ForbiddenDenominator a
                  deriving (Eq, Read, Show)

divBy :: Integral a => a -> [a] -> Either (DivByError a) [a]
divBy _ [] = Right []
```

 v: latest ▾

```
divBy _ (0:_) = Left DivBy0
divBy _ (10:_) = Left (ForbiddenDenominator 10)
divBy _ (20:_) = Left (ForbiddenDenominator 20)
divBy numerator (denom:xs) =
  case divBy numerator xs of
    Left x -> Left x
    Right results -> Right ((numerator `div` denom) : results)
```

现在，在出现错误时，可以通过 `Left` 数据检查导致错误的准确原因。或者，可以简单的只是通过 `show` 打印出来。下面是这个函数的应用：

```
ghci> divBy 50 [1,2,5,8]
Right [50,25,10,6]
ghci> divBy 50 [1,2,5,8,10]
Left (ForbiddenDenominator 10)
ghci> divBy 50 [1,2,0,8,10]
Left DivBy0
```

Warning

所有这些 `Either` 的例子都跟我们之前的 `Maybe` 一样，都会遇到失去惰性的问题。我们将在这一章的最后用一个练习题来解决这个问题。

Monadic地使用Either

回到 *Maybe Monad的用法* 这一节，我们向你展示了如何在一个monad中使用 `Maybe`。`Either` 也可以在monad中使用，但是可能会复杂一点。原因是 `fail` 是硬编码的只接受 `String` 作为失败代码，因此我们必须有一种方法将这样的字符串映射成我们的 `Left` 使用的类型。正如你前面所见，`Control.Monad.Error` 为 `Either String a` 提供了内置的支持，它没有涉及到将参数映射到 `fail`。这里我们可以将我们的例子修改为monadic风格使得 `Either` 可以工作：

```
-- file: ch19/divby8.hs
{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Error

data Show a =>
  DivByError a = DivBy0
              | ForbiddenDenominator a
              | OtherDivByError String
              deriving (Eq, Read, Show)

instance Error (DivByError a) where
  strMsg x = OtherDivByError x

divBy :: Integral a => a -> [a] -> Either (DivByError a) [a]
divBy = divByGeneric

divByGeneric :: (Integral a, MonadError (DivByError a) m) =>
  a -> [a] -> m [a]
divByGeneric _ [] = return []
divByGeneric _ (0:_) = throwError DivBy0
divByGeneric _ (10:_) = throwError (ForbiddenDenominator 10)
divByGeneric _ (20:_) = throwError (ForbiddenDenominator 20)
divByGeneric numerator (denom:xs) =
  do next <- divByGeneric numerator xs
  return ((numerator `div` denom) : next)
```

这里，我们需要打开 `FlexibleContexts` 语言扩展以提供 `divByGeneric` 的类型签名。`divBy` 函数跟之前的工作方式完全一致。对于 `divByGeneric`，我们将 `DivByError` 做为 `Error` 类型类的成员，通过定义调用 `fail` 时的行为（`strMsg` 函数）。我们还将 `Right` 转化成 `return`，将 `Left` 转化成 `throwError` 进行泛化。

异常

 v: latest ▾

许多语言中都有异常处理，包括Haskell。异常很有用，因为当发生故障时，它提供了一种简单的处理方法，即使故障离发生的地方沿着函数调用链走了几层。有了异常，不需要检查每个函数调用的返回值是否发生了错误，不需要注意去生成表示错误的返回值，像C程序员必须这么做。在Haskell中，由于有 `monad` 以及 `Either` 和 `Maybe` 类型，你通常可以在纯的代码中达到同样的效果而不需要使用异常和异常处理。

有些问题—尤其是涉及到IO调用—需要处理异常。在Haskell中，异常可能会在程序的任何地方抛出。然而，由于计算顺序是不确定的，异常只可以在 `IO monad` 中捕获。Haskell异常处理不涉及像Python或者Java中那样的特殊语法。捕获和处理异常的技术是—真令人惊讶—函数。

异常第一步

在 `Control.Exception` 模块中，定义了各种跟异常相关的函数和类型。`Exception` 类型是在那里定义的；所有的异常的类型都是 `Exception`。还有用于捕获和处理异常的函数。让我们先看一看 `try`，它的类型是 `IO a -> IO (Either Exception a)`。它将异常处理包装在 `IO` 中。如果有异常抛出，它会返回一个 `Left` 值表示异常；否则，返回原始结果到 `Right` 值。让我们在 `ghci` 中运行一下。我们首先触发一个未处理的异常，然后尝试捕获它。

```
ghci> :m Control.Exception
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> print x
*** Exception: divide by zero
ghci> print y
5
ghci> try (print x)
Left divide by zero
ghci> try (print y)
5
Right ()
```

注意到在 `let` 语句中没有抛出异常。这是意料之中的，是因为惰性求值；除以零只有到打印 `x` 的值的时候才需要计算。还有，注意 `try (print y)` 有两行输出。第一行是由 `print` 产生的，它在终端上显示5。第二个是由 `ghci` 生成的，这个表示 `print y` 的返回值为 `()` 并且没有抛出异常。

惰性和异常处理

既然你知道了 `try` 是如何工作的，让我们试下另一个实验。让我们假设我们想捕获 `try` 的结果用于后续的计算，这样我们可以处理除的结果。我们大概会这么做：

```
ghci> result <- try (return x)
Right *** Exception: divide by zero
```

这里发生了什么？让我们拆成一步一步看，先试下另一个例子：

```
ghci> let z = undefined
ghci> try (print z)
Left Prelude.undefined
ghci> result <- try (return z)
Right *** Exception: Prelude.undefined
```

跟之前一样，将 `undefined` 赋值给 `z` 没什么问题。问题的关键，以及前面的迷惑，都在于惰性求值。准确地说，是在于 `return`，它没有强制它的参数的执行；它只是将它包装了一下。这样，`try (return undefined)` 的结果应该是 `Right undefined`。现在，`ghci` 想要将这个结果显示在终端上。它将运行到打印“Right”，但是 `undefined` 无法打印（或者说除以零的结果无法打印）。因此你看到了异常信息，它是来源于 `ghci` 的，而不是你的程序。

这是一个关键点。让我们想想为什么之前的例子可以工作，而这个不可以。之前，我们把 `print x` 放在了 `try` 里面。打印一些东西的值，固然是需要执行它的，因此，异常在正确的地方被检测到了。但是，仅仅是使用 `return` 并不会强制计算的执行。为了解决这个问题，`Control.Exception` 模块中定义了一个 `evaluate` 函数。它的行为跟 `return` 类似，但是会让参数立即执行。让我们试一下：

```
ghci> let z = undefined
ghci> result <- try (evaluate z)
Left Prelude.undefined
```

 v: latest ▾


```
ghci> result <- try (evaluate x)
Left divide by zero
```

看，这就是我们想要的答案。无论对于 `undefined` 还是除以零的例子，都可以正常工作。

Tip

记住：任何时候你想捕获纯的代码中抛出的异常，在你的异常处理函数中使用 `evaluate` 而不是 `return`。

使用 `handle`

通常，你可能希望如果一块代码中没有任何异常发生，就执行某个动作，否则执行不同的动作。对于像这种场合，有一个叫做 `handle` 的函数。这个函数的类型是 `(Exception -> IO a) -> IO a -> IO a`。即是说，它需要两个参数：前一个是一个函数，当执行后一个动作发生异常的时候它会被调用。下面是我们使用的一种方式：

```
ghci> :m Control.Exception
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> handle (\_ -> putStrLn "Error calculating result") (print x)
Error calculating result
ghci> handle (\_ -> putStrLn "Error calculating result") (print y)
5
```

像这样，如果计算中没有错误发生，我们可以打印一条好的信息。这当然要比除以零出错时程序崩溃要好。

选择性地处理异常

上面的例子的一个问题是，对于任何异常它都是打印 “Error calculating result”。可能会有些其它不是除零的异常。例如，显示输出时可能会发生错误，或者纯的代码中可能抛出一些其它的异常。

`handleJust` 函数就是处理这种情况的。它让你指定一个测试来决定是否对给定的异常感兴趣。让我们看一下：

```
-- file: ch19/hj1.hs
import Control.Exception

catchIt :: Exception -> Maybe ()
catchIt (ArithException DivideByZero) = Just ()
catchIt _ = Nothing

handler :: () -> IO ()
handler _ = putStrLn "Caught error: divide by zero"

safePrint :: Integer -> IO ()
safePrint x = handleJust catchIt handler (print x)
```

`catchIt` 定义了一个函数，这个函数会决定我们对给定的异常是否感兴趣。如果是，它会返回 `Just`，否则返回 `Nothing`。还有，`Just` 中附带的值会被传到我们的处理函数中。现在我们可以很好地使用 `safePrint` 了：

```
ghci> :l hj1.hs [1 of 1] Compiling Main ( hj1.hs, interpreted ) Ok, modules loaded: Main. ghci> let x = 5 `div` 0 ghci> let y = 5
`div` 1 ghci> safePrint x Caught error: divide by zero ghci> safePrint y 5
```

`Control.Exception` 模块还提供了一些可以在 `handleJust` 中使用的函数，以便于我们将异常的范围缩小到我们所关心的类别。例如，有个函数 `arithExceptions` 类型是 `Exception -> Maybe ArithException` 可以挑选出任意的 `ArithException` 异常，但是会忽略掉其它。我们可以像这样使用它：

```
-- file: ch19/hj2.hs
import Control.Exception

handler :: ArithException -> IO ()
handler e = putStrLn $ "Caught arithmetic error: " ++ show e

safePrint :: Integer -> IO ()
safePrint x = handleJust arithExceptions handler (print x)
```

 v: latest ▾

用这种方式，我们可以捕获所有 `ArithException` 类型的异常，但是仍然让其它的异常通过，不捕获也不修改。我们可以看到它是这样工作的：

```
ghci> :l hj2.hs
[1 of 1] Compiling Main           ( hj2.hs, interpreted )
Ok, modules loaded: Main.
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> safePrint x
Caught arithmetic error: divide by zero
ghci> safePrint y
5
```

其中特别感兴趣的是，你大概注意到了 `ioErrors` 测试，这是跟一大类的I/O相关的异常。

I/O异常

大概在任何程序中异常最大的来源就是I/O。在处理外部世界的时候所有事情都可能出错：磁盘满了，网络断了，或者你期望文件里面有数据而文件却是空的。在Haskell中，I/O异常就跟其它的异常一样可以用 `Exception` 数据类型来表示。另一方面，由于有这么多类型的I/O异常，有一个特殊的模块— `System.IO.Error` 专门用于处理它们。

`System.IO.Error` 定义了两个函数：`catch` 和 `try`，跟 `Control.Exception` 中的类似，它们都是用于处理异常的。然而，不像 `Control.Exception` 中的函数，这些函数只会捕获I/O错误，而不处理其它类型异常。在Haskell中，所有I/O错误有一个共同类型 `IOError`，它的定义跟 `IOException` 是一样的。

Tip

当心你使用的哪个名字 因为 `System.IO.Error` 和 `Control.Exception` 定义了同样名字的函数，如果你将它们都导入你的程序，你将收到一个错误信息说引用的函数有歧义。你可以通过 `qualified` 引用其中一个或者另一个，或者将其中一个或者另一个的符号隐藏。

注意 Prelude 导出的是 `System.IO.Error` 中的 `catch`，而不是 `ControlException` 中提供的。记住，前者只捕获I/O错误，而后者捕获所有的异常。换句话说，你要的几乎总是 `Control.Exception` 中的那个 `catch`，而不是默认的那个。

让我们看一下对我们有益的一个在I/O系统中使用异常的方法。在 [使用文件和句柄](#) 这一节里，我们展示了一个使用命令式风格从文件中一行一行的读取的程序。尽管我们后面也示范过更简洁的，更“Haskelly”的方式解决那个问题，让我们在这里重新审视这个例子。在 `mainloop` 函数中，在读一行之前，我们必须明确地测试我们的输入文件是否结束。这次，我们可以检查尝试读一行是否会导致一个EOF错误，像这样子：

```
-- file: ch19/toupper-impch20.hs
import System.IO
import System.IO.Error
import Data.Char(toUpper)

main :: IO ()
main = do
    inh <- openFile "input.txt" ReadMode
    outh <- openFile "output.txt" WriteMode
    mainloop inh outh
    hClose inh
    hClose outh

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
    do input <- try (hGetLine inh)
    case input of
        Left e ->
            if isEOFError e
            then return ()
            else ioError e
        Right inpStr ->
            do hPutStrLn outh (map toUpper inpStr)
               mainloop inh outh
```

 v: latest ▾

这里，我们使用 `System.IO.Error` 中的 `try` 来检测是否 `hGetLine` 抛出一个 `IOError`。如果是，我们使用 `isEOFError`（在 `System.IO.Error` 中定义）来看是否抛出异常表明我们到达了文件末尾。如果是的，我们退出循环。如果是其它的异常，我们调用 `ioError` 重新抛出它。

有许多的这种测试和方法可以从 `System.IO.Error` 中定义的 `IOError` 中提取信息。我们推荐你在需要的时候去查一下库的参考页。

抛出异常

到现在为止，我们已经详细地讨论了异常处理。还有另外一个困惑：抛出异常。到目前为止这一章我们所接触到的例子中，都是由 Haskell 为你抛出异常的。然后你也可以自己抛出任何异常。我们会告诉你怎么做。

你将会注意到这些函数大部分似乎返回一个类型为 `a` 或者 `IO a` 的值。这意味着这个函数似乎可以返回任意类型的值。事实上，由于这些函数会抛出异常，一般情况下它们决不“返回”任何东西。这些返回值让你可以在各种各样的上下文中使用这些函数，不同的上下文需要不同的类型。

让我们使用函数 `Control.Exception` 来开始我们的抛出异常的教程。最通用的函数是 `throw`，它的类型是 `Exception -> a`。这个函数可以抛出任何的 `Exception`，并且可以用于纯的上下文中。还有一个类型为 `Exception -> IO a` 的函数 `throwIO` 在 `IO monad` 中抛出异常。这两个函数都需要一个 `Exception` 用于抛出。你可以手工制作一个 `Exception`，或者重用之前创建的 `Exception`。

还有一个函数 `ioError`，它在 `Control.Exception` 和 `System.IO.Error` 中定义都是相同的，它的类型是 `IOError -> IO a`。当你想生成任意的 I/O 相关的异常的时候可以使用它。

动态异常

这需要使用两个很不常用的 Haskell 模块：`Data.Dynamic` 和 `Data.Typeable`。我们不会讲太多关于这些模块，但是告诉你当你需要制作自己的动态异常类型时，可以使用这些工具。

在 [第二十一章使用数据库](http://book.realworldhaskell.org/read/using-databases.html) <http://book.realworldhaskell.org/read/using-databases.html> 中，你会看到 HDBC 数据库库使用动态异常来表示 SQL 数据库返回给应用的错误。数据库引擎返回的错误通常有三个组件：一个表示错误码的整数，一个状态，以及一条人类可读的错误消息。在这一章中我们会创建我们自己的 HDBC `SqlError` 实现。让我们从错误自身的数据结构表示开始：

```
-- file: ch19/dynexc.hs
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Dynamic
import Control.Exception

data SqlError = SqlError {seState :: String,
                          seNativeError :: Int,
                          seErrorMsg :: String}
    deriving (Eq, Show, Read, Typeable)
```

通过继承 `Typeable` 类型类，我们使这个类型可用于动态的类型编程。为了让 GHC 自动生成一个 `Typeable` 实例，我们要开启 `DeriveDataTypeable` 语言扩展。

现在，让我们定义一个 `catchSql` 和一个 `handleSql` 用于捕获一个 `SqlError` 异常。注意常规的 `catch` 和 `handle` 函数无法捕获我们的 `SqlError`，因为它不是 `Exception` 类型的。

```
-- file: ch19/dynexc.hs
{- | Execute the given IO action.

If it raises a 'SqlError', then execute the supplied
handler and return its return value. Otherwise, proceed
as normal. -}
catchSql :: IO a -> (SqlError -> IO a) -> IO a
catchSql = catchDyn

{- | Like 'catchSql', with the order of arguments reversed. -}
handleSql :: (SqlError -> IO a) -> IO a -> IO a
handleSql = flip catchSql
```

[译注：原文中文件名是 `dynexc.hs`，但是跟前面的冲突了，所以这里重命名为 `dynexc1.hs`]

 v: latest ▾

这些函数仅仅是在 `catchDyn` 外面包了很薄的一层，类型是 `Typeable exception => IO a -> (exception -> IO a) -> IO a`。这里我们简单地限定了它的类型使得它只捕猎SQL异常。

正常地，当一个异常抛出，但是没有在任何地方被捕获，程序会崩溃并显示异常到标准错误输出。然而，对于动态异常，系统不会知道该如何显示它，因此你将仅仅会看到一个的“unknown exception”消息，这可能没太大帮助。我们可以提供一个辅助函数，这样应用可以写成，比如说 `main = handleSqlError $ do ...`，使抛出的异常可以显示。下面是如何写 `handleSqlError`：

```
-- file: ch19/dynexc.hs
{- | Catches 'SqlError's, and re-raises them as IO errors with fail.
Useful if you don't care to catch SQL errors, but want to see a sane
error message if one happens. One would often use this as a
high-level wrapper around SQL calls. -}
handleSqlError :: IO a -> IO a
handleSqlError action =
  catchSql action handler
  where handler e = fail ("SQL error: " ++ show e)
```

[译注：原文中是dynexc.hs，这里重命名过文件]

最后，让我们给出一个如何抛出 `SqlError` 异常的例子。下面的函数做的就是这件事：

```
-- file: ch19/dynexc.hs
throwSqlError :: String -> Int -> String -> a
throwSqlError state nativeerror errormsg =
  throwDyn (SqlError state nativeerror errormsg)

throwSqlErrorIO :: String -> Int -> String -> IO a
throwSqlErrorIO state nativeerror errormsg =
  evaluate (throwSqlError state nativeerror errormsg)
```

Tip

提醒一下，`evaluate` 跟 `return` 类似但是会立即计算它的参数。

这样我们的动态异常的支持就完成了。代码很多，你大概不需要这么多代码，但是我们想要给你一个动态异常自身的例子以及和它相关的工具。事实上，这里的例子几乎就反映在HDBC库中。让我们在 `ghci` 中试一下：

```
ghci> :l dynexc.hs
[1 of 1] Compiling Main             ( dynexc.hs, interpreted )
Ok, modules loaded: Main.
ghci> throwSqlErrorIO "state" 5 "error message"
*** Exception: (unknown)
ghci> handleSqlError $ throwSqlErrorIO "state" 5 "error message"
*** Exception: user error (SQL error: SqlError {seState = "state", seNativeError = 5, seErrorMsg = "error message"})
ghci> handleSqlError $ fail "other error"
*** Exception: user error (other error)
```

这里你可以看出，`ghci` 自己并不知道如何显示SQL错误。但是，你可以看到 `handleSqlError` 帮助做了这些，不过没有捕获其它的错误。最后让我们试一个自定义的handler：

```
ghci> handleSql (fail . seErrorMsg) (throwSqlErrorIO "state" 5 "my error")
*** Exception: user error (my error)
```

这里，我们自定义了一个错误处理抛出一个新的异常，构成 `SqlError` 中的 `seErrorMsg` 域。你可以看到它是按预想中那样工作的。

练习

1. 将 `Either` 修改成 `Maybe` 例子中的那种风格，使它保存惰性。

monad中的错误处理

因为我们必须捕获 `IO monad` 中的异常，如果我们在一个 `monad` 中或者在 `monad` 的转化栈中使用它们，我们将跳回到 `IO monad`。这几乎肯定不是我们想要的。

在 **构建以理解Monad变换器** 中我们定义了一个 `MaybeT` 的变换，但是它更像是一个有助于理解的东西，而不是编程的工具。幸运的是，已经有一个专门的—也更有用的—`monad` 变换：`ErrorT`，它是定义在 `Control.Monad.Error` 模块中的。

`ErrorT` 变换器使我们可以向 `monad` 中添加异常，但是它使用了特殊的方法，跟 `Control.Exception` 模块中提供的不一样。它提供给我们一些有趣的能力。

如果我们继续用 `ErrorT` 接口，在这个 `monad` 中我们可以抛出和捕获异常。

根据其它 `monad` 变换器的命名规范，这个执行函数的名字是 `runErrorT`。当它遇到 `runErrorT` 之后，未被捕获的 `ErrorT` 异常将停止向上传递。我们不会被踢到 `IO monad` 中。

我们可以控制我们的异常的类型。

Warning

不要把 `ErrorT` 跟普通异常混淆 如果我们在 `ErrorT` 内面使用 `Control.Exception` 中的 `throw` 函数，我们仍然会弹出到 `IO monad`。

正如其它的 `mtl monad` 一样，`ErrorT` 提供的接口是由一个类型类定义的。

```
-- file: ch19/MonadError.hs
class (Monad m) => MonadError e m | m -> e where
  throwError :: e          -- error to throw
             -> m a

  catchError :: m a        -- action to execute
             -> (e -> m a) -- error handler
             -> m a
```

类型变量 `e` 代表我们想要使用的错误类型。不管我们的错误类型是什么，我们必须将它做成 `Error` 类型类的实例。

```
-- file: ch19/MonadError.hs
class Error a where
  -- create an exception with no message
  noMsg :: a

  -- create an exception with a message
  strMsg :: String -> a
```

`ErrorT` 实现 `fail` 时会用到 `strMsg` 函数。它将 `strMsg` 作为一个异常抛出，将自己接收到的字符串参数传递给这个异常。对于 `noMsg`，它是用于提供 `MonadPlus` 类型类中的 `mzero` 的实现。

为了支持 `strMsg` 和 `noMsg` 函数，我们的 `ParseError` 类型会有一个 `Chatty` 构造器。这个将用作构造器如果，比如说，有人在我们的 `monad` 中调用 `fail`。

我们需要知道的最后一块是关于执行函数 `runErrorT` 的类型。

```
ghci> :t runErrorT
runErrorT :: ErrorT e m a -> m (Either e a)
```

一个小的解析构架

为了说明 `ErrorT` 的使用，让我们开发一个类似于 `Parsec` 的解析库的基本的骨架。

```
-- file: ch19/ParseInt.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Monad.Error
import Control.Monad.State
import qualified Data.ByteString.Char8 as B

data ParseError = NumericOverflow
```

 v: latest ▾

```

    | EndOfInput
    | Chatty String
    deriving (Eq, Ord, Show)

instance Error ParseError where
  noMsg  = Chatty "oh noes!"
  strMsg = Chatty

```

对于我们解析器的状态，我们会创建一个非常小的monad变换器栈。一个 `State monad` 包含了需要解析的 `ByteString`，在栈的顶部是 `ErrorT` 用于提供错误处理。

```

-- file: ch19/ParseInt.hs
newtype Parser a = P {
  runP :: ErrorT ParseError (State B.ByteString) a
} deriving (Monad, MonadError ParseError)

```

和平常一样，我们将我们的monad栈包装在一个 `newtype` 中。这样做没有任意性能损耗，但是增加了类型安全。我们故意避免继承 `MonadState B.ByteString` 的实例。这意味着 `Parser monad` 用户将不能够使用 `get` 或者 `put` 去查询或者修改解析器的状态。这样的结果是，我们强制自己去做一些手动提升的事情来获取在我们栈中的 `State monad`。

```

-- file: ch19/ParseInt.hs
liftP :: State B.ByteString a -> Parser a
liftP m = P (lift m)

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
  s <- liftP get
  case B.uncons s of
    Nothing    -> throwError EndOfInput
    Just (c, s') -> liftP (put s') >> return c
    | p c      -> liftP (put s') >> return c
    | otherwise -> throwError (Chatty "satisfy failed")

```

`catchError` 函数对于我们的任何非常有用，远胜于简单的错误处理。例如，我们可以很轻松地解除一个异常，将它变成更友好的形式。

```

-- file: ch19/ParseInt.hs
optional :: Parser a -> Parser (Maybe a)
optional p = (Just `liftM` p) `catchError` \_ -> return Nothing

```

我们的执行函数仅仅是将各层连接起来，将结果重新组织成更整洁的形式。

```

-- file: ch19/ParseInt.hs
runParser :: Parser a -> B.ByteString
  -> Either ParseError (a, B.ByteString)
runParser p bs = case runState (runErrorT (runP p)) bs of
  (Left err, _) -> Left err
  (Right r, bs) -> Right (r, bs)

```

如果我们将它加载到 `ghci` 中，我们可以对它进行了一些测试。

```

ghci> :m +Data.Char
ghci> let p = satisfy isDigit
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
ghci> runParser p (B.pack "x")
Left (Chatty "satisfy failed")
ghci> runParser p (B.pack "9abc")
Right ('9', "abc")
ghci> runParser (optional p) (B.pack "x")
Right (Nothing, "x")
ghci> runParser (optional p) (B.pack "9a")
Right (Just '9', "a")

```

 v: latest ▾

级习

1. 写一个 `many` 解析器，类型是 `Parser a -> Parser [a]`。它应该执行解析直到失败。
2. 使用 `many` 写一个 `int` 解析器，类型是 `Parser Int`。它应该既能接受负数也能接受正数。
3. 修改你们 `int` 解析器，如果在解析时检测到了一个数值溢出，抛出一个 `NumericOverflow` 异常。

注

- [38] 这里我们使用的是整数的除法，因此 `50 / 8` 显示是 6 而不是 6.25。在这个例子中我们没有使用浮点算术是因为对一个 `Double` 除以零会返回一个特殊的 `Infinity` 而不是一个错误。
- [39] 关于 `Maybe` 的介绍，参考`<让过程更可控的方法 <http://rwh.readthedocs.org/en/latest/chp/3.html#id21>>`_
- [40] 更多关于 `Either` 的信息，参考`<通过 API 设计进行错误处理 <http://rwh.readthedocs.org/en/latest/chp/8.html#api>>`_
- [41] 在一些其它语言中，抛出异常是叫做 `raising`。
- [42] 可以手动继承 `Typeable` 实例，但是那样很麻烦。

讨论

0条评论 Real World Haskell 中文版

1 登录 ▾

♥ 推荐 🐦 推文 f 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

2条评论 • 6年前

 yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地


Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

 forlice — ...


第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

 Wengel An —

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

📧 订阅 🌐 在您的网站上使用 Disqus添加 Disqus添加 🔒 Disqus 隐私政策隐私政策隐私

📄 v: latest ▾