第六章：类型类

类型类（typeclass）是 Haskell 最强大的功能之一：它用于定义通用接口，为各种不同的类型提供一组公共特性集。

类型类是某些基本语言特性的核心，比如相等性测试和数值操作符。

在讨论如何使用类型类之前，先来看看它能做什么。

类型类的作用

假设这样一个场景：我们想对 `Color` 类型的值进行对比，但 Haskell 的语言设计者却没有实现 `==` 操作。

要解决这个问题，必须亲自实现一个相等性测试函数：

```
-- file: ch06/colorEq.hs

data Color = Red | Green | Blue

colorEq :: Color -> Color -> Bool
colorEq Red   Red   = True
colorEq Green Green = True
colorEq Blue  Blue  = True
colorEq _     _     = False
```

在 ghci 里测试：

```
Prelude> :load colorEq.hs
[1 of 1] Compiling Main             ( colorEq.hs, interpreted )
Ok, modules loaded: Main.

*Main> colorEq Green Green
True

*Main> colorEq Blue Red
False
```

过了一会，程序又添加了一个新类型 —— 职位：它对公司中的各个员工进行分类。

在执行像是工资计算这类任务是，又需要用到相等性测试，所以又需要再次为职位类型定义相等性测试函数：

```
-- file: ch06/roleEq.hs

data Role = Boss | Manager | Employee

roleEq :: Role -> Role -> Bool
roleEq Employee Employee = True
roleEq Manager  Manager  = True
roleEq Boss     Boss     = True
roleEq _        _        = False
```

测试：

```
Prelude> :load roleEq.hs
[1 of 1] Compiling Main             ( roleEq.hs, interpreted )
Ok, modules loaded: Main.

*Main> roleEq Boss Boss
True

*Main> roleEq Boss Employee
False
```

v: latest

`colorEq` 和 `roleEq` 的定义揭示了一个问题：对于每个不同的类型，我们都需要为它们专门定义一个对比函数。

这种做法非常低效，而且烦人。如果同一个对比函数（比如 == ）可以用于对比任何类型的值，这样就会方便得多。

另一方面，一般来说，如果定义了相等测试函数（比如 == ），那么不等测试函数（比如 /= ）的值就可以直接对相等测试函数取反（使用 not ）来计算得出。因此，如果可以通过相等测试函数来定义不等测试函数，那么会更方便。

通用函数还可以让代码变得更通用：如果同一段代码可以用于不同类型的输入值，那么程序的代码量将大大减少。

还有很重要的一点是，如果在之后添加通用函数对新类型的支持，那么原来的代码应该不需要进行修改。

Haskell 的类型类可以满足以上提到的所有要求。

什么是类型类?

类型类定义了一系列函数，这些函数对于不同类型的值使用不同的函数实现。它和其他语言的接口和多态方法有些类似。

[译注：这里原文是将“面向对象编程中的对象”和 Haskell 的类型类进行类比，但实际上这种类比并不太恰当，类比成接口和多态方法更适合一点。]

我们定义一个类型类来解决前面提到的相等性测试问题：

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

类型类使用 class 关键字来定义，跟在 class 之后的 BasicEq 是这个类型类的名字，之后的 a 是这个类型类的实例类型（instance type）。

BasicEq 使用类型变量 a 来表示实例类型，说明它并不将这个类型类限定于某个类型：任何一个类型，只要它实现了这个类型类中定义的函数，那么它就是这个类型类的实例类型。

实例类型所使用的名字可以随意选择，但是它和类型类中定义函数签名时所使用的名字应该保持一致。比如说，我们使用 a 来表示实例类型，那么函数签名中也必须使用 a 来代表这个实例类型。

BasicEq 类型类只定义了 isEqual 一个函数 —— 它接受两个参数作为输入，并且这两个参数都指向同一种实例类型：

```
Prelude> :load BasicEq_1.hs
[1 of 1] Compiling Main             ( BasicEq_1.hs, interpreted )
Ok, modules loaded: Main.

*Main> :type isEqual
isEqual :: BasicEq a => a -> a -> Bool
```

作为演示，以下代码将 Bool 类型作为 BasicEq 的实例类型，实现了 isEqual 函数：

```
instance BasicEq Bool where
    isEqual True  True  = True
    isEqual False False = True
    isEqual _     _     = False
```

在 ghci 里验证这个程序：

```
*Main> isEqual True True
True

*Main> isEqual False True
False
```

如果试图将不是 BasicEq 实例类型的值作为输入调用 isEqual 函数，那么就会引发错误：

```
*Main> isEqual "hello" "moto"

<interactive>:5:1:
    No instance for (BasicEq [Char])
          arising from a use of `isEqual'
```

```
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "hello" "moto"
    In an equation for `it': it = isEqual "hello" "moto"
```

错误信息提醒我们， `[Char]` 并不是 `BasicEq` 的实例类型。

稍后的一节会介绍更多关于类型类实例的定义方式，这里先继续前面的例子。这一次，除了 `isEqual` 之外，我们还想定义不等测试函数 `isNotEqual` ：

```
class BasicEq a where
    isEqual    :: a -> a -> Bool
    isNotEqual :: a -> a -> Bool
```

同时定义 `isEqual` 和 `isNotEqual` 两个函数产生了一些不必要的工作：从逻辑上讲，对于任何类型，只要知道 `isEqual` 或 `isNotEqual` 的任意一个，就可以计算出另外一个。因此，一种更省事的办法是，为 `isEqual` 和 `isNotEqual` 两个函数提供默认值，这样 `BasicEq` 的实例类型只要实现这两个函数中的一个，就可以顺利使用这两个函数：

```
class BasicEq a where
    isEqual :: a -> a -> Bool
    isEqual x y = not (isNotEqual x y)

    isNotEqual :: a -> a -> Bool
    isNotEqual x y = not (isEqual x y)
```

以下是将 `Bool` 作为 `BasicEq` 实例类型的例子：

```
instance BasicEq Bool where
    isEqual False False = True
    isEqual True  True  = True
    isEqual _     _     = False
```

我们只要定义 `isEqual` 函数，就可以“免费”得到 `isNotEqual` ：

```
Prelude> :load BasicEq_3.hs
[1 of 1] Compiling Main             ( BasicEq_3.hs, interpreted )
Ok, modules loaded: Main.

*Main> isEqual True True
True

*Main> isEqual False False
True

*Main> isNotEqual False True
True
```

当然，如果闲着没事，你仍然可以自己亲手定义这两个函数。但是，你至少要定义两个函数中的一个，否则两个默认的函数就会互相调用，直到程序崩溃。

定义类型类实例

定义一个类型为某个类型类的实例，指的就是，为某个类型实现给定类型类所声明的全部函数。

比如在前面， `BasicEq` 类型类定义了两个函数 `isEqual` 和 `isNotEqual` ：

```
class BasicEq a where
    isEqual :: a -> a -> Bool
    isEqual x y = not (isNotEqual x y)

    isNotEqual :: a -> a -> Bool
    isNotEqual x y = not (isEqual x y)
```

在前一节，我们成功将 Bool 类型实现为 BasicEq 的实例类型，要使 Color 类型也成为 BasicEq 类型类的实例，就需要另外为 Color 类型实现 isEqual 和 isNotEqual：

```
instance BasicEq Color where
    isEqual Red Red = True
    isEqual Blue Blue = True
    isEqual Green Green = True
    isEqual _ _ = True
```

注意，这里的函数定义和之前的 colorEq 函数定义实际上没有什么不同，唯一的区别是，它使得 isEqual 不仅可以对 Bool 类型进行对比测试，还可以对 Color 类型进行对比测试。

更一般地说，只要为相应的类型实现 BasicEq 类型类中的定义，那么 isEqual 就可以用于对比*任何*我们想对比的类型。

不过在实际中，通常并不使用 BasicEq 类型类，而是使用 Haskell Report 中定义的 Eq 类型类：它定义了 == 和 /= 操作符，这两个操作符才是 Haskell 中最常用的测试函数。

以下是 Eq 类型类的定义：

```
class  Eq a  where
    (==), (/=) :: a -> a -> Bool

-- Minimal complete definition:
--      (==) or (/=)
x /= y     =  not (x == y)
x == y     =  not (x /= y)
```

稍后会介绍更多使用 Eq 类型类的信息。

几个重要的内置类型类

前面两节分别介绍了类型类的定义，以及如何让某个类型成为给定类型类的实例类型。

正本节会介绍几个 Prelude 库中包含的类型类。如本章开始时所说的，类型类是 Haskell 语言某些特性的奠基石，本节就会介绍几个这方面的例子。

更多信息可以参考 Haskell 的函数参考，那里一般都给出了类型类的详细介绍，并且说明，要成为这个类型类的实例，需要实现那些函数。

Show

Show 类型类用于将值转换为字符串，它最重要的函数是 show 。

show 函数使用单个参数接收输入数据，并返回一个表示该输入数据的字符串：

```
Main> :type show
show :: Show a => a -> String
```

以下是一些 show 函数调用的例子：

```
Main> show 1
"1"

Main> show [1, 2, 3]
"[1,2,3]"

Main> show (1, 2)
"(1,2)"
```

Ghci 输出一个值，实际上就是对这个值调用 putStrLn 和 show：

```
Main> 1
1

Main> show 1
"1"

Main> putStrLn (show 1)
1
```

因此，如果你定义了一种新的数据类型，并且希望通过 ghci 来显示它，那么你就应该将这个类型实现为 Show 类型类的实例，否则 ghci 就会向你抱怨，说它不知道该怎样用字符串的形式表示这种数据类型：

```
Main> data Color = Red | Green | Blue;

Main> show Red

<interactive>:10:1:
    No instance for (Show Color)
        arising from a use of `show'
    Possible fix: add an instance declaration for (Show Color)
    In the expression: show Red
    In an equation for `it': it = show Red

Prelude> Red

<interactive>:5:1:
    No instance for (Show Color)
        arising from a use of `print'
    Possible fix: add an instance declaration for (Show Color)
    In a stmt of an interactive GHCi command: print it
```

通过实现 Color 类型的 show 函数，让 Color 类型成为 Show 的类型实例，可以解决以上问题：

```
instance Show Color where
    show Red   = "Red"
    show Green = "Green"
    show Blue  = "Blue"
```

当然， show 函数的打印值并不是非要和类型构造器一样不可，比如 Red 值并不是非要表示为 "Red" 不可，以下是另一种实例化 Show 类型类的方式：

```
instance Show Color where
    show Red   = "Color 1: Red"
    show Green = "Color 2: Green"
    show Blue  = "Color 3: Blue"
```

Read

Read 和 Show 类型类的作用正好相反，它将字符串转换为值。

Read 最有用的函数是 read ： 它接受一个字符串作为参数，对这个字符串进行处理，并返回一个值，这个值的类型为 Read 实例类型的成员（所有实例类型中的一种）。

```
Prelude> :type read
read :: Read a => String -> a
```

以下代码展示了 read 的用法：

```
Prelude> read "3"

<interactive>:5:1:
    Ambiguous type variable `a0' in the constraint:
          (Read a0) arising from a use of `read'
    Probable fix: add a type signature that fixes these type variable(s)
```

```
    In the expression: read "3"
    In an equation for `it': it = read "3"

Prelude> (read "3")::Int
3

Prelude> :type it
it :: Int

Prelude> (read "3")::Double
3.0

Prelude> :type it
it :: Double
```

注意在第一次调用 read 的时候，我们并没有显式地给定类型签名，这时对 read "3" 的求值会引发错误。这是因为有非常多的类型都是 Read 的实例，而编译器在 read 函数读入 "3" 之后，不知道应该将这个值转换成什么类型，于是编译器就会向我们发牢骚。

因此，为了让 read 函数返回正确类型的值，必须给它指示正确的类型。

使用 Read 和 Show 进行序列化

很多时候，程序需要将内存中的数据保存为文件，又或者，反过来，需要将文件中的数据转换为内存中的数据实体。这种转换过程称为*序列化*和*反序列化*.

通过将类型实现为 Read 和 Show 的实例类型， read 和 show 两个函数可以成为非常好的序列化工具。

作为例子，以下代码将一个内存中的列表序列化到文件中：

```
Prelude> let years = [1999, 2010, 2012]

Prelude> show years
"[1999,2010,2012]"

Prelude> writeFile "years.txt" (show years)
```

writeFile 将给定内容写入到文件当中，它接受两个参数，第一个参数是文件路径，第二个参数是写入到文件的字符串内容。

观察文件 years.txt 可以看到， (show years) 所产生的文本被成功保存到了文件当中：

```
$ cat years.txt
[1999,2010,2012]
```

使用以下代码可以对 years.txt 进行反序列化操作：

```
Prelude> input <- readFile "years.txt"

Prelude> input                  -- 读入的字符串
"[1999,2010,2012]"

Prelude> (read input)::[Int]    -- 将字符串转换成列表
[1999,2010,2012]
```

readFile 读入给定的 years.txt ，并将它的内存传给 input 变量，最后，通过使用 read ，我们成功将字符串反序列化成一个列表。

数字类型

Haskell 有一集非常强大的数字类型：从速度飞快的 32 位或 64 位整数，到任意精度的有理数，包罗万有。

除此之外，Haskell 还有一系列通用算术操作符，这些操作符可以用于几乎所有数字类型。而对数字类型的这种强有力的支持就是建立在类型类的基础上的。

作为一个额外的好处（side benefit），用户可以定义自己的数字类型，并且获得和内置数字类型完全平等的权利。

以下表格显示了 Haskell 中最常用的一些数字类型：

表格 6.1： 部分数字类型

类型	介绍
Double	双精度浮点数。表示浮点数的常见选择。
Float	单精度浮点数。通常在对接 C 程序时使用。
Int	固定精度带符号整数；最小范围在 -2^29 至 2^29-1 。相当常用。
Int8	8 位带符号整数
Int16	16 位带符号整数
Int32	32 位带符号整数
Int64	64 位带符号整数
Integer	任意精度带符号整数；范围由机器的内存限制。相当常用。
Rational	任意精度有理数。保存为两个整数之比（ratio）。
Word	固定精度无符号整数。占用的内存大小和 `Int` 相同
Word8	8 位无符号整数
Word16	16 位无符号整数
Word32	32 位无符号整数
Word64	64 位无符号整数

大部分算术操作都可以用于任意数字类型，少数的一部分函数，比如 `asin` ，只能用于浮点数类型。

以下表格列举了操作各种数字类型的常见函数和操作符：

表格 6.2： 部分数字函数和

项	类型	模块	描述
(+)	Num a => a -> a -> a	Prelude	加法
(-)	Num a => a -> a -> a	Prelude	减法
(*)	Num a => a -> a -> a	Prelude	乘法
(/)	Fractional a => a -> a -> a	Prelude	份数除法
(**)	Floating a => a -> a -> a	Prelude	乘幂
(^)	(Num a, Integral b) => a -> b -> a	Prelude	计算某个数的非负整数次方
(^^)	(Fractional a, Integral b) => a -> b -> a	Prelude	分数的任意整数次方
(%)	Integral a => a -> a -> Ratio a	Data.Ratio	构成比率
(.&.)	Bits a => a -> a -> a	Data.Bits	二进制并操作
(.|.)	Bits a => a -> a -> a	Data.Bits	二进制或操作
abs	Num a => a -> a	Prelude	绝对值操作
approxRational	RealFrac a => a -> a -> Rational	Data.Ratio	通过分数的分子和分母计算出近似有理数
cos	Floating a => a -> a	Prelude	余弦函数。另外还有 acos 、 cosh 和 acosh ， 类型和 cos 一样。
div	Integral a => a -> a -> a	Prelude	整数除法，总是截断小数位。
fromInteger	Num a => Integer -> a	Prelude	将一个 Integer 值转换为任意数字类型。
fromIntegral	(Integral a, Num b) => a -> b	Prelude	一个更通用的转换函数，将任意 Integral 值转为任意数字类型。
fromRational	Fractional a => Rational -> a	Prelude	将一个有理数转换为分数。可能会有精度损失。
log	Floating a => a -> a	Prelude	自然对数算法。
logBase	Floating a => a -> a -> a	Prelude	计算指定底数对数。
maxBound	Bounded a => a	Prelude	有限长度数字类型的最大值。
minBound	Bounded a => a	Prelude	有限长度数字类型的最小值。
mod	Integral a => a -> a -> a	Prelude	整数取模。
pi	Floating a => a	Prelude	圆周率常量。
quot	Integral a => a -> a -> a	Prelude	整数除法；商数的分数部分截断为 0 。
recip	Fractional a => a -> a	Prelude	分数的倒数。

项	类型	模块	描述
rem	Integral a => a -> a -> a	Prelude	整数除法的余数。
round	(RealFrac a, Integral b) => a -> b	Prelude	四舍五入到最近的整数。
shift	Bits a => a -> Int -> a	Bits	输入为正整数，就进行左移。如果为负数，进行右移。
sin	Floating a => a -> a	Prelude	正弦函数。还提供了 asin 、 sinh 和 asinh ， 和 sin 类型一样。
sqrt	Floating a => a -> a	Prelude	平方根
tan	Floating a => a -> a	Prelude	正切函数。还提供了 atan 、 tanh 和 atanh ， 和 tan 类型一样。
toInteger	Integral a => a -> Integer	Prelude	将任意 Integral 值转换为 Integer
toRational	Real a => a -> Rational	Prelude	从实数到有理数的有损转换
truncate	(RealFrac a, Integral b) => a -> b	Prelude	向下取整
xor	Bits a => a -> a -> a	Data.Bits	二进制异或操作

数字类型及其对应的类型类列举在下表：

表格 6.3： 数字类型的类型类实例

类型	Bits	Bounded	Floating	Fractional	Integral	Num	Real	RealFrac
Double			X	X		X	X	X
Float			X	X		X	X	X
Int	X	X			X	X	X	
Int16	X	X			X	X	X	
Int32	X	X			X	X	X	
Int64	X	X			X	X	X	
Integer	X				X	X	X	
Rational or any Ratio				X		X	X	X
Word	X	X			X	X	X	
Word16	X	X			X	X	X	
Word32	X	X			X	X	X	
Word64	X	X			X	X	X	

表格 6.2 列举了一些数字类型之间进行转换的函数，以下表格是一个汇总：

表格 6.4： 数字类型之间的转换

源类型	目标类型			
	Double, Float	Int, Word	Integer	Rational
Double, Float Int, Word Integer Rational	fromRational . toRational fromIntegral fromIntegral fromRational	truncate * fromIntegral fromIntegral truncate *	truncate * fromIntegral N/A truncate *	toRational fromIntegral fromIntegral N/A

* 除了 `truncate` 之外，还可以使用 `round` 、 `ceiling` 或者 `float` 。

第十三章会说明，怎样用自定义数据类型来扩展数字类型。

相等性，有序和对比

除了前面介绍的通用算术符号之外，相等测试、不等测试、大于和小于等对比操作也是非常常见的。

其中， `Eq` 类型类定义了 `==` 和 `/=` 操作，而 `>=` 和 `<=` 等对比操作，则由 `Ord` 类型类定义。

需要将对比操作和相等性测试分开用两个类型类来定义的原因是，对于某些类型，它们只对相等性测试和不等测试有兴趣，比如 `Handle` 类型，而部分有序操作（particular ordering， 大于、小于等）对它来说是没有意义的。

所有 `Ord` 实例都可以使用 `Data.List.sort` 来排序。

几乎所有 Haskell 内置类型都是 Eq 类型类的实例，而 Ord 实例的类型也不在少数。

自动派生

对于简单的数据类型， Haskell 编译器可以自动将类型派生（derivation）为 Read 、 Show 、 Bounded 、 Enum 、 Eq 和 Ord 的实例。

以下代码将 Color 类型派生为 Read 、 Show 、 Eq 和 Ord 的实例：

```
data Color = Red | Green | Blue
    deriving (Read, Show, Eq, Ord)
```

测试：

```
*Main> show Red
"Red"

*Main> (read "Red")::Color
Red

*Main> (read "[Red, Red, Blue]")::[Color]
[Red,Red,Blue]

*Main> Red == Red
True

*Main> Data.List.sort [Blue, Green, Blue, Red]
[Red,Green,Blue,Blue]

*Main> Red < Blue
True
```

注意 Color 类型的排序位置由定义类型时值构造器的排序决定。

自动派生并不总是可用的。比如说，如果定义类型 data MyType = MyType (Int -> Bool) ，那么编译器就没办法派生 MyType 为 Show 的实例，因为它不知道该怎么将 MyType 函数的输出转换成字符串，这会造成编译错误。

除此之外，当使用自动推导将某个类型设置为给定类型类的实例时，定义这个类型时所使用的其他类型，也必须是给定类型类的实例（通过自动推导或手动添加的都可以）。

举个例子，以下代码不能使用自动推导：

```
data Book = Book

data BookInfo = BookInfo Book
                deriving (Show)
```

Ghci 会给出提示，说明 Book 类型也必须是 Show 的实例， BookInfo 才能对 Show 进行自动推导：

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

ad.hs:4:27:
    No instance for (Show Book)
          arising from the 'deriving' clause of a data type declaration
    Possible fix:
        add an instance declaration for (Show Book)
        or use a standalone 'deriving instance' declaration,
        so you can specify the instance context yourself
    When deriving the instance for (Show BookInfo)
Failed, modules loaded: none.
```

相反，以下代码可以使用自动推导，因为它对 Book 类型也使用了自动推导，使得 Book 类型变成了 Show 的实例：


```
data Book = Book
            deriving (Show)
```

```
data BookInfo = BookInfo Book
                deriving (Show)
```

使用 `:info` 命令在 ghci 中确认两种类型都是 `Show` 的实例：

```
Prelude> :load ad.hs
[1 of 1] Compiling Main             ( ad.hs, interpreted )
Ok, modules loaded: Main.

*Main> :info Book
data Book = Book     -- Defined at ad.hs:1:6
instance Show Book -- Defined at ad.hs:2:23

*Main> :info BookInfo
data BookInfo = BookInfo Book    -- Defined at ad.hs:4:6
instance Show BookInfo -- Defined at ad.hs:5:27
```

类型类实战：让 JSON 更好用

我们在 ***在 Haskell 中表示 JSON 数据*** 一节介绍的 `JValue` 用起来还不够简便。这里是一段由搜索引擎返回的实际 JSON 数据。删除重整之后：

```
{
    "query": "awkward squad haskell",
    "estimatedCount": 3920,
    "moreResults": true,
    "results":
    [{
        "title": "Simon Peyton Jones: papers",
        "snippet": "Tackling the awkward squad: monadic input/output ...",
        "url": "http://research.microsoft.com/~simonpj/papers/marktoberdorf/",
    },
    {
        "title": "Haskell for C Programmers | Lambda the Ultimate",
        "snippet": "... the best job of all the tutorials I've read ...",
        "url": "http://lambda-the-ultimate.org/node/724",
    }]
}
```

进一步简化之，并用 Haskell 表示：

```
-- file: ch06/SimpleResult.hs
import SimpleJSON

result :: JValue
result = JObject [
    ("query", JString "awkward squad haskell"),
    ("estimatedCount", JNumber 3920),
    ("moreResults", JBool True),
    ("results", JArray [
        JObject [
        ("title", JString "Simon Peyton Jones: papers"),
        ("snippet", JString "Tackling the awkward ..."),
        ("url", JString "http://.../marktoberdorf/")
        ]])
    ]
```

由于 Haskell 不原生支持包含不同类型值的列表，我们不能直接表示包含不同类型值的 JSON 对象。 我们需要把每个值都用 JValue 构造器包装起来。但这样我们的灵活性就受到了限制：如果我们想把数字 `3920` 转换成字符串 `"3,920"`， 我们就必须把 `JNumber` 构造器换成 `JString` 构造器。

Haskell 的类型类提供了一个诱人的解决方案：

```haskell
-- file: ch06/JSONClass.hs
type JSONError = String

class JSON a where
    toJValue :: a -> JValue
    fromJValue :: JValue -> Either JSONError a

instance JSON JValue where
    toJValue = id
    fromJValue = Right
```

现在，我们无需再用 `JNumber` 等构造器去包装值了，直接使用 `toJValue` 函数即可。 如果我们更改值的类型，编译器会自动选择相应的 `toJValue` 实现。

我们也提供了 `fromJValue` 函数，它把 `JValue` 值转换成我们希望的类型。

让错误信息更有用

`fromJValue` 函数的返回类型为 `Either`。跟 `Maybe` 一样，这个类型是预定义的。我们经常用它来表示可能会失败的计算。

虽然 `Maybe` 也用作这个目的，但它在错误发生时没有给我们足够有用的信息：我们只得到一个 `Nothing`。 `Either` 类型的结构相同，但它在错误发生时会调用 `Left` 构造器，并且还接受一个参数。

```haskell
-- file: ch06/DataEither.hs
data Maybe a = Nothing
             | Just a
               deriving (Eq, Ord, Read, Show)

data Either a b = Left a
                | Right b
                  deriving (Eq, Ord, Read, Show)
```

我们经常使用 `String` 作为 `a` 参数的类型，以便在出错时提供有用的描述。 为了说明在实际中怎么使用 `Either` 类型，我们来看一个简单实例。

```haskell
-- file: ch06/JSONClass.hs
instance JSON Bool where
    toJValue = JBool
    fromJValue (JBool b) = Right b
    fromJValue _ = Left "not a JSON boolean"
```

[译注：读者若想在 **ghci** 中尝试 `fromJValue`，需要为其提供类型标注，例如 `(fromJValue(toJValue True))::Either JSONError Bool`。]

使用类型别名创建实例

Haskell 98标准不允许我们用下面的形式声明实例，尽管它看起来没什么问题：

```haskell
-- file: ch06/JSONClass.hs
instance JSON String where
    toJValue              = JString

    fromJValue (JString s) = Right s
    fromJValue _           = Left "not a JSON string"
```

`String` 是 `[Char]` 的别名，因此它的类型是 `[a]`，并用 `Char` 替换了类型变量 `a`。 根据 Haskell 98的规则，我们在声明实例的时候不能用具体类型替代类型变量。 也就是说，我们可以给 `[a]` 声明实例，但给 `[Char]` 不行。

尽管 GHC 默认遵守 Haskell 98标准，但是我们可以在文件顶部添加特殊格式的注释来解除这个限制。

```haskell
-- file: ch06/JSONClass.hs
{-# LANGUAGE TypeSynonymInstances #-}
```

这条注释是一条编译器指令，称为*编译选项（pragma）*，它告诉编译器允许这项语言扩展。 上面的代码因为 ``TypeSynonymInstances`` 这项语言扩展而合法。 我们在本章（本书）还会碰到更多的语言扩展。

[译注：作者举的这个例子实际上牵涉到了两个问题。第一，Haskell 98不允许类型别名，这个问题可以通过上述方法解决。 第二，Haskell 98不允许 `[Char]` 这种形式的类型，这个问题需要通过增加另外一条编译选项 `{-# LANGUAGE FlexibleInstances #-}` 来解决。]

生活在开放世界

Haskell 的设计允许我们任意创建类型类实例。

```
-- file: ch06/JSONClass.hs
doubleToJValue :: (Double -> a) -> JValue -> Either JSONError a
doubleToJValue f (JNumber v) = Right (f v)
doubleToJValue _ _ = Left "not a JSON number"

instance JSON Int where
    toJValue = JNumber . realToFrac
    fromJValue = doubleToJValue round

instance JSON Integer where
    toJValue = JNumber . realToFrac
    fromJValue = doubleToJValue round

instance JSON Double where
    toJValue = JNumber
    fromJValue = doubleToJValue id
```

我们可以在任意地方创建新实例，而不仅限于在定义了类型类的模块中。类型类系统的这个特性被称为*开放世界假设*（open world assumption）。 如果有方法表示“这个类型类只存在这些实例”，那我们将得到一个*封闭的*世界。

我们希望把列表转为 JSON 数组。现在先不用关心实现细节，暂时用 `undefined` 替代函数内容即可。

```
-- file: ch06/BrokenClass.hs
instance (JSON a) => JSON [a] where
    toJValue = undefined
    fromJValue = undefined
```

我们也希望能将键/值对列表转为 JSON 对象。

```
-- file: ch06/BrokenClass.hs
instance (JSON a) => JSON [(String, a)] where
    toJValue = undefined
    fromJValue = undefined
```

什么时候重叠实例（Overlapping instances）会出问题?

如果我们把这些定义放进文件中并在 **ghci** 里载入，初看起来没什么问题。

```
*JSONClass> :l BrokenClass.hs
[1 of 2] Compiling JSONClass        ( JSONClass.hs, interpreted )
[2 of 2] Compiling BrokenClass      ( BrokenClass.hs, interpreted )
Ok, modules loaded: JSONClass, BrokenClass
```

然而，当我们使用序对列表实例时，麻烦来了。

```
*BrokenClass> toJValue [("foo","bar")]

<interactive>:10:1:
    Overlapping instances for JSON [([Char], [Char])]
        arising from a use of ‘toJValue’
    Matching instances:
        instance JSON a => JSON [(String, a)]
            -- Defined at BrokenClass.hs:13:10
```

```
        instance JSON a => JSON [a] -- Defined at BrokenClass.hs:8:10
    In the expression: toJValue [("foo", "bar")]
    In an equation for ‘it’ : it = toJValue [("foo", "bar")]
```

*重叠实例*问题是由 Haskell 的开放世界假设造成的。 这里有一个更简单的例子来说明发生了什么。

```
-- file: ch06/Overlap.hs
class Borked a where
    bork :: a -> String

instance Borked Int where
    bork = show

instance Borked (Int, Int) where
    bork (a, b) = bork a ++ ", " ++ bork b

instance (Borked a, Borked b) => Borked (a, b) where
    bork (a, b) = ">>" ++ bork a ++ " " ++ bork b ++ "<<"
```

对于序对，我们有两个 `Borked` 类型类实例：一个是 `Int` 序对，另一个是任意类型的序对，只要这个类型是 `Borked` 类型类的实例。

假设我们想把 `bork` 应用于 `Int` 序对。编译器必须选择一个实例来用。由于这两个实例都能用，所以看上去它好像只要选那个更相关（specific）的实例就可以了。

但是，`GHC` 默认是保守的。它坚持只能有一个可用实例。这样，当我们试图使用 `bork` 时，它就会报错。

> **Note**
>
> 重叠实例什么时候会出问题?
>
> 之前我们提到，我们可以把某个类型类的实例分散在几个模块中。GHC 并不会在意重叠实例的存在。 相反，只有当我们使用受影响的类型类的函数，GHC 被迫要选择使用哪个实例时，它才会报错。

取消类型类的一些限制

通常，我们不能给多态类型（polymorphic type）的特化版本（specialized version）写类型类实例。 `[Char]` 类型就是多态类型 `[a]` 特化成 `Char` 的结果。因此我们禁止声明 `[Char]` 为某个类型类的实例。 这非常不方便，因为字符串在代码中无处不在。

`FlexibleInstances` 语言扩展取消了这个限制，它允许我们写这样的实例。

GHC 支持另外一个有用的语言扩展，`OverlappingInstances`，它解决了重叠实例带来的问题。 如果存在重叠实例，编译器会选择最相关的（specific）那一个。

我们经常把这个扩展和 `TypeSynonymInstances` 放在一起使用。下面是一个例子。

```
-- file: ch06/SimpleClass.hs
{-# LANGUAGE TypeSynonymInstances, OverlappingInstances #-}

import Data.List

class Foo a where
    foo :: a -> String

instance Foo a => Foo [a] where
    foo = concat . intersperse ", " . map foo

instance Foo Char where
    foo c = [c]

instance Foo String where
    foo = id
```

如果我们对 `String` 应用 `foo`，编译器会选择 `String` 的特定实现。 即使 `[a]` 和 `Char` 都是 `Foo` 的实例，但由于 `String` 实例更[illegible] GHC 选择了它。

即使开了 OverlappingInstances 扩展，如果 GHC 发现了多个同样相（equally specific）关的实例，它仍然会拒绝代码。

> 何时使用 OverlappingInstances 扩展（to be added）

字符串的 show 是如何工作的?

OverlappingInstances 和 TypeSynonymInstances 语言扩展是 GHC 特有的，Haskell 98 并不支持。 然而，Haskell 98 中的 Show 类型类在转化 Char 列表和 Int 列表时却用了不同的方法。它用了一个聪明但简单的小技巧。

Show 类型类定义了转换单个值的 show 方法和转换列表的 showList 方法。 showList 默认使用中括号和逗号转换列表。

[a] 的 Show 实例使用 showList 实现。Char 的 Show 实例提供了一个特殊的 showList 实现， 它使用双引号，并转义非 ASCII 打印字符。

结果是，如果有人想对 [Char] 应用 show，编译器会选择 showList 的实现，并使用双引号正确转换这个字符串。

这样，换个角度看问题，我们就能避免 OverlappingInstances 扩展了。

如何给类型定义新身份（Identity）

除了熟悉的 data 关键字外，Haskell 还允许我们用 newtype 关键字来创建新类型。

```
-- file: ch06/Newtype.hs
data DataInt = D Int
    deriving (Eq, Ord, Show)

newtype NewtypeInt = N Int
    deriving (Eq, Ord, Show)
```

newtype 声明的作用是重命名现有类型，并给它一个新身份。可以看出，它的用法和使用 data 关键字进行类型声明看起来很相似。

> **Note**
>
> type 和 newtype 关键字
>
> 尽管名字类似，type 和 newtype 关键字的作用却完全不同。 type 关键字给了我们另一种指代某个类型的方法，类似于给朋友起的绰号。 我们和编译器都知道 [Char] 和 String 指的是同一个类型。
>
> 相反，newtype 关键字的存在是为了隐藏类型的本性。考虑这个 UniqueID 类型。
>
> ```
> -- file: ch06/Newtype.hs
> newtype UniqueID = UniqueID Int
>     deriving (Eq)
> ```
>
> 编译器会把 UniqueID 当成和 Int 不同的类型。 作为 UniqueID 的用户，我们只知道它是一个唯一标识符；我们并不知道它是用 Int 来实现的。

在声明 newtype 时，我们必须决定暴露被重命名类型的哪些类型类实例。 这里，我们让 NewtypeInt 提供 Int 类型的 Eq， Ord 和 Show 实例。 这样，我们就可以比较和打印 NewtypeInt 类型的值了。

```
*Main> N 1 < N 2
True
```

由于我们没有暴露 Int 的 Num 或 Integral 实例，NewtypeInt 类型的值并不是数字。 例如，我们不能做加法。

```
*Main> N 313 + N 37

<interactive>:9:7:
    No instance for (Num NewtypeInt) arising from a use of ‘+’
    In the expression: N 313 + N 37
    In an equation for ‘it’ : it = N 313 + N 37
```


跟用 data 关键字一样，我们可以用 newtype 的值构造器创建新值，或者对现有值进行模式匹配。

如果 newtype 没用自动派生来暴露对应类型的类型类实现的话，我们可以自己写一个新实例或者干脆不实现那个类型类。

data 和 newtype 的区别

newtype 关键字给现有类型一个不同的身份，相比起 data，它使用时的限制更多。具体来讲，newtype 只能有一个值构造器， 并且这个构造器只能有一个字段。

```
-- file: ch06/NewtypeDiff.hs
-- 可以：任意数量的构造器和字段
data TwoFields = TwoFields Int Int

-- 可以：一个字段
newtype Okay = ExactlyOne Int

-- 可以：使用类型变量
newtype Param a b = Param (Either a b)

-- 可以：使用记录语法
newtype Record = Record {
        getInt :: Int
    }

-- 不可以：没有字段
newtype TooFew = TooFew

-- 不可以：多于一个字段
newtype TooManyFields = Fields Int Int

-- 不可以：多于一个构造器
newtype TooManyCtors = Bad Int
                     | Worse Int
```

除此之外，data 和 newtype 还有一个重要区别。由 data 关键字创建的类型在运行时有一个簿记开销， 如记录某个值是用哪个构造器创建的。而 newtype 只有一个构造器，所以不需要这个额外开销。 这使得它在运行时更省时间和空间。

由于 newtype 的构造器只在编译时使用，运行时甚至不存在， 用 newtype 定义的类型和用 data 定义的类型在匹配 undefined 时会有不同的行为。

为了理解它们的不同点，我们首先回顾一下普通数据类型的行为。我们已经非常熟悉，在运行时对 undefined 求值会导致崩溃。

```
Prelude> undefined
*** Exception: Prelude.undefined
```

我们把 undefined 放进 D 构造器创建一个 DataInt，然后对它进行模式匹配。

```
*Main> case (D undefined) of D _ -> 1
1
```

由于我们的模式匹配只匹配构造器而不管里面的值，undefined 未被求值，因而不会抛出异常。

下面的例子没有使用 D 构造器，因而模式匹配时 undefined 被求值，异常抛出。

```
*Main> case undefined of D _ -> 1
*** Exception: Prelude.undefined
```

当我们用 N 构造器创建 NewtypeInt 值时，它的行为与使用 DataInt 类型的 D 构造器相同：没有异常。

```
*Main> case (N undefined) of N _ -> 1
1
```

但当我们把表达式中的 N 去掉，并对 undefined 进行模式匹配时，关键的不同点来了。

```
*Main> case undefined of N _ -> 1
1
```

没有崩溃！由于运行时不存在构造器，匹配 `N _` 实际上就是在匹配通配符 `_`：由于通配符总可以被匹配，所以表达式是不需要被求值的。

命名类型的三种方式

这里简要回顾一下 haskell 引入新类型名的三种方式。

`data` 关键字定义一个真正的代数数据类型。
`type` 关键字给现有类型定义别名。类型和别名可以通用。
`newtype` 关键字给现有类型定义一个不同的身份（distinct identity）。原类型和新类型不能通用。

JSON typeclasses without overlapping instances

可怕的单一同态限定（monomorphism restriction）

Haskell 98 有一个微妙的特性可能会在某些意想不到的情况下“咬”到我们。下面这个简单的函数展示了这个问题。

```
-- file: ch06/Monomorphism.hs
myShow = show
```

如果我们试图把它载入 **ghci**，会产生一个奇怪的错误：

```
Prelude> :l Monomorphism.hs

[1 of 1] Compiling Main             ( Monomorphism.hs, interpreted )

Monomorphism.hs:2:10:
    No instance for (Show a0) arising from a use of ‘show’
    The type variable ‘a0’ is ambiguous
    Relevant bindings include
        myShow :: a0 -> String (bound at Monomorphism.hs:2:1)
    Note: there are several potential instances:
        instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
        instance Show Ordering -- Defined in ‘GHC.Show’
        instance Show Integer -- Defined in ‘GHC.Show’
        ...plus 22 others
    In the expression: show
    In an equation for ‘myShow’: myShow = show
    Failed, modules loaded: none.
```

[译注：译者得到的输出和原文有出入，这里提供的是使用最新版本 GHC 得到的输出。]

错误信息中提到的 “monomorphism” 是 Haskell 98 的一部分。 *单一同态*是多态（polymorphism）的反义词：它表明某个表达式只有一种类型。 Haskell 有时会强制使某些声明不像我们预想的那么多态。

我们在这里提单一同态是因为尽管它和类型类没有直接关系，但类型类给它提供了产生的环境。

> **Note**
>
> 在实际代码中可能很久都不会碰到单一同态，因此我们觉得你没必要记住这部分的细节， 只要在心里知道有这么回事就可以了，除非 GHC 真的报告了跟上面类似的错误。 如果真的发生了，记得在这儿曾读过这个错误，然后回过头来看就行了。

我们不会试图去解释单一同态限制。Haskell 社区一致同意它并不经常出现；它解释起来很棘手（tricky）； 它几乎没什么实际用处；它唯一的作用就是坑人。举个例子来说明它为什么棘手：尽管上面的例子违反了这个限制， 下面的两个编译起来却毫无问题。


```
-- file: ch06/Monomorphism.hs
myShow2 value = show value
```

```
myShow3 :: (Show a) => a -> String
myShow3 = show
```

上面的定义表明，如果 GHC 报告单一同态限制错误，我们有三个简单的方法来处理。

显式声明函数参数，而不是隐性。
显式定义类型签名，而不是依靠编译器去推导。
不改代码，编译模块的时候用上 `NoMonomorphismRestriction` 语言扩展。它取消了单一同态限制。

没人喜欢单一同态限制，因此几乎可以肯定的是下一个版本的 Haskell 会去掉它。 但这并不是说加上 `NoMonomorphismRestriction` 就可以一劳永逸：有些编译器（包括一些老版本的 GHC） 识别不了这个扩展，但用另外两种方法就可以解决问题。如果这种可移植性对你不是问题，那么请务必打开这个扩展。

结论

在这章，你学到了类型类有什么用以及怎么用它们。我们讨论了如何定义自己的类型类，然后又讨论了一些 Haskell 库里定义的类型类。 最后，我们展示了怎么让 Haskell 编译器给你的类型自动派生出某些类型类实例。

讨论