13.9　特征 7：编译器自动进行类型推断

下面我们来介绍一下函数式语言的最后一个特征——**类型推断**（type inference）。这一特征并不是所有函数式语言都具备的，而是强类型语言所特有的。

我们在第 4 章中介绍过，编程语言分为强类型方式和弱类型方式两种。在**强类型方式**的情况下，在源代码中强制声明函数的类型，在编译阶段检查错误。而在**弱类型方式**的情况下，并不会显式声明函数的类型，而是在运行时判断类型，这样有助于编写更加灵活的程序。前者安全性更强，后者灵活性更强，而类型推断结构则吸取了这两种方式的优点。

在介绍类型推断之前，我们先来介绍一下函数的类型。在函数式语言中，函数可以作为值进行处理，因此，函数本身也拥有类型。函数的类型被称为**函数类型**，使用参数和返回值的类型来表示。例如，如果是使用 Haskell 编写的从整数（Int）转换为字符串（String）的函数，那么就表示为“Int -> String”类型[1]。

> 在函数式语言中，函数本身也拥有类型。函数的类型用参数和返回值的类型来表示。

接下来，我们开始介绍类型推断。**类型推断**是一种即使不声明变量和函数的类型，编译器也能够自动判断类型的结构。

我们来看一下下面的 Haskell 的代码示例（代码清单 13.18）。

代码清单13.18　increment函数的定义

```
increment x = x + 1
```

① 在 Haskell 中，对于接收多个参数的函数类型，通常会列举所有参数和返回值的类型，像“String -> Int -> String”这样来表示。这是因为接收多个参数的函数通过柯里化可以由接收单个参数的函数构成。

这里定义了 increment 函数。左边的 increment 是函数名，x 是参数名，右边的“x + 1”表示逻辑。这里并未声明参数 x 和返回值的类型，但如果编写执行该函数的代码，结果就会像下面这样，编译器会准确判断参数的类型（代码清单 13.19）。

代码清单13.19 increment函数的应用

```
increment 3      ⇨ 编译通过      ①
increment "abc"  ⇨ 编译错误!     ②
```

①中对数值 3 应用 increment 函数，因此会正常通过编译。而②中对字符串 "abc" 应用该函数，结果发生了编译错误。

这是因为，Haskell 编译器会自动将参数 x 判断为数值型。

代码清单 13.18 的代码非常短，让人毫无头绪。这段代码到底是如何判断 x 的类型的呢？

一个线索就是对参数 x 进行的“+ 1”运算[1]。Haskell 编译器由此判断 x 是数值型，而不是字符串型和布尔型。原因就是，Haskell 中使用“+”进行加法运算的对象只有数值型的数据。

这种结构就是类型推断。在支持类型推断的语言中，如果可以从其他部分推测数据或函数的类型，那么代码中就无须显式声明类型。

> 在具有类型推断结构的语言中，编译器会自动推测数据和函数的类型。

为了编写灵活的程序，支持类型推断的很多函数式语言还提供了更加优秀的结构，那就是**多态**。在这里，多态被用来创建可以应用于非特定多数的类型的通用函数[2]。下面，我们来介绍一下该结构。

① 实际上，“+”（加号）并不是 Haskell 语言中的运算符，而是表示函数名。

② 具体地说，该多态相当于被称为“参数多态”（parametric polymorphism）的结构。Java 和 C# 中通过泛型（generics）来支持该结构。

在定义具有多态的函数的情况下，参数和返回值的类型并不指定为数值型、字符型或布尔型等具体的类型，而是指定为被称为**类型变量**的虚拟类型。使用类型变量定义的类型称为**多态类型**。例如，对于 Haskell 中提供的从列表中取出第一个元素的 head 函数，其定义如下所示（代码清单 13.20）。

代码清单13.20　使用类型变量的类型定义

```
[a] -> a
```

上面的 a 是类型变量[1]。[] 表示用于存储元素的数据结构列表。列表是一种类似于能够在内部存储多个值的数组的结构，基本上所有的函数式语言都支持列表[2]。

该 head 函数只是简单地从列表中取出开头的第一个元素。该函数具有多态，因此会根据参数动态决定返回值的类型，如果参数为数值型的列表，则返回数值型；如果参数为字符串型的列表，则返回字符串型（图 13-8）。

① Haskell 中规定类型变量名以英文小写字母开头，一般习惯使用 a 或 b 这样一个英文小写字母。

② 顺便一提，最初的函数式语言 Lisp 的名称就来源于“列表处理语言”（List Processing Language）。

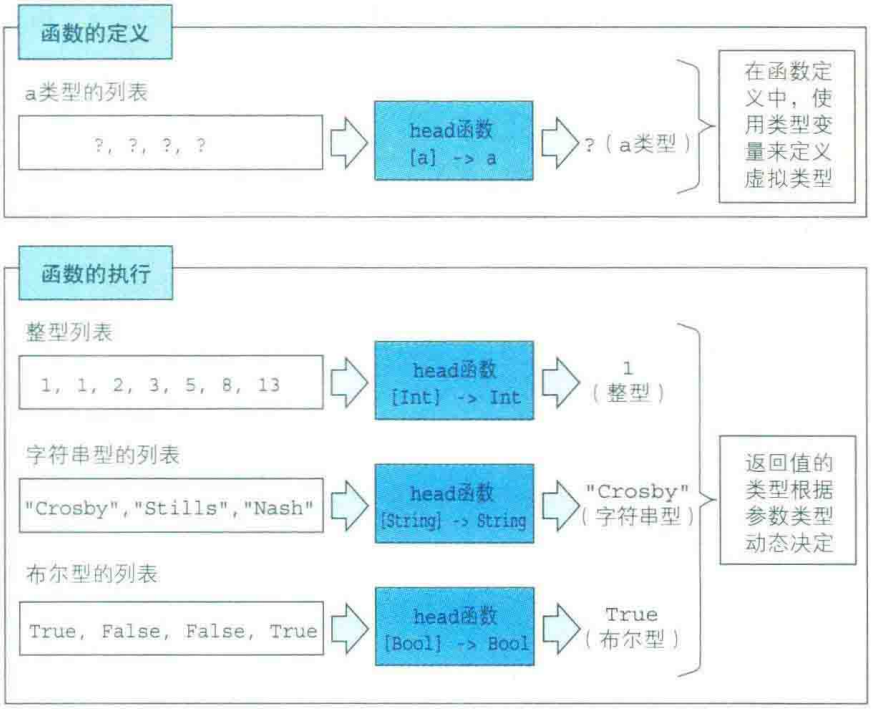

图 13-8　具有多态的函数

该多态结构与前面介绍的类型推断一起，可以发挥巨大的威力。这是因为，对于具有多态的函数的返回值，类型推断也会发挥作用。例如，在 head 函数的参数是字符串型的情况下，返回值也会默认为字符串型。如果对其返回值进行算术运算，就会发生编译错误。

像这样同时使用多态和类型推断，就可以实现强类型方式的好处，即在编译时进行错误判断，同时还可以编写出通用性高的函数。

使用类型推断和多态，可以简洁且安全地编写通用性高的函数。

“多态”一词的英文为 polymorphism，与本书中多次提及的 OOP 的“多态”一词完全相同。这两者的共同之处在于，针对不同类型可以使用同一个函数或方法，但它们的实际结构存在很大不同。函数式语言的多态将

同一个函数应用于不同类型，而 OOP 的多态则通过子类进行重写，还可以修改方法的逻辑。为了避免混淆，大家一定要将 OOP 的多态和函数式语言的多态区分开[1]。

13.10　对 7 个特征的总结

到这里为止，我们逐个介绍了函数式语言的 7 个特征。下面，我们将各个特征中介绍的表示函数式语言结构的术语汇总在表 13-3 中。

表 13-3　函数式语言的特征和术语

特征	说　明	相关术语
1	使用函数来编写程序	函数应用
2	所有的表达式都返回值	命令式语言、表达式、表达式求值、lambda 表达式
3	将函数作为值进行处理	头等函数、高阶函数
4	可以灵活组合函数和参数	部分应用、柯里化、函数组合
5	没有副作用	副作用、绑定、引用透明性、延迟求值
6	使用分类和递归来编写循环处理	模式匹配、递归
7	编译器自动进行类型推断	类型推断、多态、多态类型、类型变量

从该表也可以看出，函数式语言的结构与命令式语言有很大不同，因此存在许多特殊术语。为了充分理解函数式语言的结构，除了实际的编程语言的语法之外，还需要充分掌握这些术语的含义和目的。在大家被各个技术搞糊涂以至于无法看到全貌时，可以查阅一下该表。

① 更详细地说，OOP 的多态被称为子类型多态（subtype polymorphism）。另外还存在第 3 种多态，即针对多个特定类型的随意多态（ad hoc polymorphism）。在 Haskell 中，将其使用类型类的结构来实现。

13.11 函数式语言的分类

接下来，我们来介绍一下函数式语言的分类。函数式语言通常基于确定类型的方式和纯粹性两个轴进行分类。基于这两个轴的分类和代表性的语言如表 13-4 所示。

表 13-4 函数式语言的分类和代表性的语言

确定类型的方式 / 纯粹性	强（静态）	弱（动态）
纯函数式语言	Haskell、Miranda	Lazy K
非纯函数式语言	Scala、OCaml、F#、ML	Common Lisp、Scheme、Erlang

横轴的确定类型的方式是根据类型检查的结构不同进行分类的。强类型语言在程序编译时进行类型检查，弱类型语言在程序运行时进行类型检查。很多强类型语言都支持特征 7 中介绍的类型推断。

纵轴的纯粹性是针对函数式语言是否纯粹的分类。换言之，就是语言规格说明中是否允许副作用的分类。纯函数式语言中基本上不允许副作用。而在非纯函数式语言中，语言规格说明中允许副作用。它除了提供函数式语言的结构之外，还与传统的命令式语言一样，支持将值赋给变量的语法。

不过，即使是纯函数式语言，也需要实现画面、网络和数据库等外部输入输出。关于这一点，Haskell 中是使用名为 Monad 的结构实现的[1]。

13.12 函数式语言的优势

到这里为止，我们介绍了函数式语言的结构，下面，我们再来思考一下函数式语言的优势。

---

① Monad 是可以存储各种值的通用容器，作为名为 Monad 的类型类提供。输入输出处理被封装为名为 IO Monad 的专用类型，使用连接函数（>>=）来控制输入输出的顺序。

第一个优势就是可以简洁地编写通用性高的程序。在使用函数式语言编写程序的情况下，相比传统的命令式语言，相同的程序使用十分之一到二分之一的代码量就可以实现。代码量少，程序就容易理解，开发和维护也会变轻松。函数式语言中拥有很多传统语言中没有的结构，比如高阶函数、部分应用、模式匹配和类型推断等，如果能够掌握熟练，就可以去除冗余，编写出简洁的程序。

还有一个优势是可以简洁地编写通用的可重用构件。利用没有副作用的函数或者将函数作为头等函数进行处理的性质，我们可以编写出之前无法实现的高通用性的构件。另外，由于重用的基本单位是函数，比类的粒度小，所以有助于编写灵活的构件。

另外，“与分散并发处理兼容性好”也被认为是函数式语言的优势。没有副作用的表达式无论从何处按什么顺序执行，整体的结果都是一样的，原理上可以说是适合并发处理的。灵活应用没有副作用的函数来编写多线程应用程序，就可以防止不慎修改多个线程共享的内存区域。不过，函数式语言仅针对多个线程的并发处理，对多个进程或者多个硬件的并发处理并没有特别大的作用[①]。

虽然不是什么实用的优势，但是函数式语言对技术人员来说很有魅力，这一点也不可忽视。传统的命令式语言中没有的各种结构、使用这些结构编写的优雅的代码、运行方式与命令式语言不同的延迟求值，等等，这些有趣的结构会刺激技术人员的求知欲。尽管函数式语言在实际系统中的应用还不是很广泛，但相关图书已经出版了很多，技术人员的社区活动和信息发送也都很活跃，这都说明了函数式语言很有魅力。

13.13　函数式语言的课题

函数式语言存在什么课题呢？

运行时的性能曾是一个大课题。函数式语言的历史比面向对象还要古

① Scala、Erlang 等语言与函数式语言的结构不同，提供了 Actor 结构来控制多线程。

老，据说最早的函数式语言 Lisp 出现于 1957 年。但在之后很长一段时间，都只有一部分技术人员使用，其中一个主要原因是，函数式语言基于不直接依赖于计算机命令的结构，因此，匮乏的硬件环境无法充分实现其性能。不过，在运行于虚拟机上的 Java 和 .NET 已经普及的现在，运行速度已经不再是大问题了。

更重要的课题是函数式语言难以理解。函数和数据的区分比较模糊，无须声明类型，可以将函数指定为参数，使用模式匹配和递归来实现循环处理等，虽然这样写出来的代码很简洁，但是对于习惯传统语言的许多程序员来说，这样的代码看起来就像是无法理解的暗号。在函数式语言广泛普及之前，还需要时间让更多的人来接受、熟悉该技术。

13.14 函数式语言和面向对象的关系

最后，我们来介绍一下函数式语言与面向对象的关系。

函数式语言和面向对象在提高软件的可维护性和可重用性这一点上是相同的，但实现该目的的方法却大不相同。我们以第 3 章介绍的解决全局变量问题的结构为例，来比较一下它们的不同。

在面向对象语言中，通过将变量和过程汇总、隐藏到类中来解决全局变量问题。另外，类是组成软件的基本构件。

与此相对，函数式语言中禁止修改变量，而是使用参数和返回值传递信息来解决全局变量问题。程序由函数的网络构成，以函数为基本的软件构件来实现整个程序。

像这样，面向对象语言和函数式语言的基本思想和结构存在很大不同（图 13-9）。

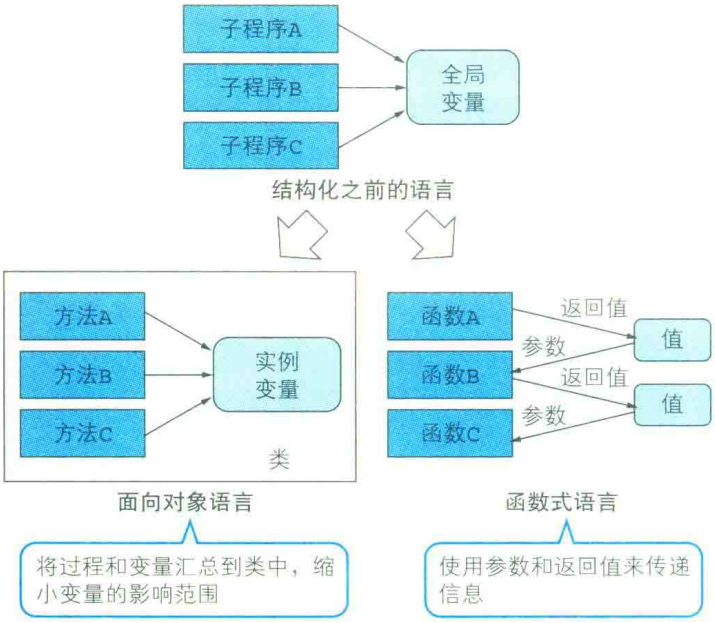

图 13–9　面向对象和函数式语言的不同

另外，面向对象语言和函数式语言也是互补的关系。

函数式语言中存在拥有面向对象和函数类型这两种结构的混合语言，例如 Scala、OCaml 和 F# 等。另外，面向对象语言的 C#、Ruby 中已经引入了头等函数等结构，Java 在将来的语言规格扩展中，也极有可能会引入函数式语言的结构。

另外，纯函数式语言 Haskell 中也引入了一部分面向对象的结构。Haskell 中为了对不同的数据类型应用共同的函数，导入了“类型类”结构，该结构将仅可以应用于特定类型的函数定义为“方法”。用于实现该结构的各个技术中使用了面向对象中常见的术语，如“实例”“继承”“超类”“子类”等[1]。

① 实际上，Haskell 的类型类结构与面向对象语言的类结构是不同的。在面向对象语言中，类是类型，而 Haskell 的类型类能汇总多个类型中共同的性质，类型类的实例是类型。另外，Haskell 的类型类还支持将某种类型作为多个类型类的实例的多重分类。

13.15 函数式语言会普及吗

函数式语言会普及吗？会像 Java、C 语言和 COBOL 那样，很多人都在系统开发中使用吗？会成为新员工培训中一定会涉及的技术吗？

坦率地说，笔者认为这种情况还不能立马实现。这是因为，某种编程语言是否会普及并不只是由该语言功能的优劣决定的。好东西并不一定会流行，这种现象并不仅限于 IT 领域。函数式语言与命令式语言的思想和结构有很大不同。要想真正普及，除了人们需要时间来熟悉之外，相关技术的发展也必不可少，比如支持类型推断结构的开发环境、类似于设计模式的设计技术窍门集等。

回顾过去，之前面向对象语言也只是一部分人使用的小规模的技术。随着实际应用不断增多，设计模式和类库等相关技术越发成熟，在 20 世纪 90 年代后半期互联网普及的大环境下，随着 Java 的出现，面向对象语言一下子跃升为主要技术。

现在函数式语言的状况在某种意义上与 Java 出现之前的面向对象非常相似。有各种语言被提出，很多技术人员都很关注，在一部分系统中也开始实际应用……在不久的将来，如果能够出现像 Java 那样容易掌握的、符合时代潮流的技术，那么函数式语言或许也会一下子普及开来。

笔者个人认为函数式语言一定会普及。原因是函数式语言非常有趣。作为软件技术人员，工作的最大成就感就在于自己参与的系统对这个世界有用。而除此之外，自己进行编程和设计的智慧享受也是非常有意义的。

不管是面向对象语言，还是函数式语言，都是能够刺激人们求知欲的有魅力的技术。

深入学习的参考书籍

[1] 青木峰郎. ふつうの Haskell プログラミング——ふつうのプログラマのための関数型言語入門 [M]. 东京：SoftBank Creative，2006.

☆☆☆

该书介绍了纯函数式语言 Haskell 的概要。作者以通俗易懂的语言进行了详细的讲解，使读者能够非常容易地理解书中的内容。该书从“Hello，world”示例代码开始，最终介绍到 IO Monad。推荐大家在最开始学习函数式语言时阅读这本书。

[2] Bryan O'Sullivan，John Goerzen，Don Stewart. Real World Haskell[M]. New York：O'Reilly Media，2008.

☆

该书以将近 700 页的篇幅，使用大量示例代码，详细介绍了 Haskell 编程的相关内容。可以说这是笔者读过的对 Monad 结构和 Monad 规则的讲解最易懂的一本书。不过，这本书无法作为语言规格说明手册使用，再加上有许多示例代码也比较难以理解，所以这本书主要面向的是真正想要学习 Haskell 的群体。

[3] Martin Odersky，Lex Spoon，Bill Venners. Scala 编程 [M]. 黄海旭，高宇翔，译. 北京：电子工业出版社，2010.

☆☆

该书介绍了在 Java 虚拟机上运行的、作为 Java 进化形式的 Scala。Scala 是面向对象语言和函数式语言的混合语言，支持许多能改善 Java 缺点的结构，所以它的语言规格说明非常庞大。该书以超过 600 页的篇幅，为读者讲解了 Scala 的性质和语法。

[4] Dean Wampler，Alex Payne. Scala 程序设计（第 2 版）[M]. 王渊，陈明，译. 北京：人民邮电出版社，2016.

☆☆

该书讲解了 Scala 的概要，从基本语法开始，逐步介绍了面向对象编程、函数式编程、使用 Actor 的并发处理、XML 协作、领域特定语言（Domain-Specific Language，DSL）和类型系统等诸多课题，这样的构成便于熟悉 OOP 的技术人员理解书中的内容。

[5] 五十嵐淳. プログラミング in OCaml——関数型プログラミングの基礎から GUI 構築まで [M]. 东京：技术评论社，2007.

☆☆

该书以基于函数式语言、同时支持面向对象功能的 OCaml（Objective Caml 的缩写）为题材，讲解了函数式语言的结构和思想。虽然目前 OCaml 与 Haskell 和 Scala 相比规模还比较小，但是它没有 Haskell 的类型类和 Monad 等复杂结构，因此易于掌握函数式语言的全貌。该书的作者是一名大学老师，详细易懂地讲解了函数式语言的程序设计。

[6] 荒井省三，いげ太. 实践 F# 関数型プログラミング入門 [M]. 东京：技术评论社，2011.

☆☆

这是一本关于微软公司在 .NET 平台上使用的函数式语言 F# 的入门书。F# 是汲取了 OCaml 流程的函数式语言，同时还具有面向对象语言的功能。从函数式语言的概要和基本结构，到 F# 的语法及其与 .NET 框架的协作等，该书都进行了介绍。

[7] 竹内郁雄. 初めての人のための LISP［増補改訂版］初めての人のための LISP［増補改訂版］[M]. 东京：翔泳社，2010.

☆☆

这是一本以独具一格的方式介绍函数式语言的鼻祖 Lisp 的入门书。

Lisp 出现于计算机技术诞生早期的 20 世纪 50 年代。这种编程语言具有非常独特的观点，它只具有最低限度的语法，函数和数据都用列表表示。该书通过出场人物之间轻松有趣的对话，带领读者了解 S 表达式和 5 个基本函数等 Lisp 的结构和基本思想。

后　记

日语中有一句杂俳，大意是“看不懂药品的疗效说明书，反而让人觉得药很有效”。这句杂俳反映了人们的一种心理，不明白疗效说明书，反而觉得药很高级，很有效。

面向对象也有类似之处。

“封装、多态和继承三种结构”“现实世界和软件是无缝的”“在需求定义和设计之前开始编码的 XP”等，当第一次听到这些时，相信不少人都会觉得“虽然不是很明白，但好像很厉害的样子”。

然而，计算机终究只是使用 0 和 1 来表示所有内容的机器，软件是驱动计算机的结构。即使让人觉得不可思议，或者无法充分理解，但只要是可以实际使用的技术，就一定可以用理论来说明。

IT 领域的技术革新速度非常快，今后也会不断出现新的技术。有的技术或许会颠覆之前的常识，或者非常复杂。即便如此，IT 技术人员也必须充分理解这些技术，并熟练掌握。

在学习这些技术的过程中感到迷惑时，大家除了考虑“如何使用该技术”（How）之外，还要考虑“该技术究竟是什么样的”（What）和“该技术为何存在”（Why）。充分掌握新技术的捷径就在于此。

致　谢

在本书执笔期间，笔者得到了许多人的帮助。

平锅健儿接着第 1 版为本书写了精彩的推荐序。冈本和己、大冢庸史和近栋稔审读了新编写的第 13 章的草稿。和田卓人告诉笔者 TDD 的最新动向。再次感谢第 1 版执笔时参与审读并提出宝贵意见的重见刚、向井丞、小笠原记子、高桥英一郎、渡边博之和梅泽真史等。衷心感谢笔者目前任职的 UL Systems 公司以及之前任职的 OGIS-RI、日本 UNISYS 公司的前辈、同事和客户等。

衷心感谢在博客、SNS 和网上书店对本书第 1 版留下评论和感想的各位。每当读到好评时，笔者就会有满满的成就感和喜悦感。而读者提出的差评和技术错误则成为第 2 版执笔时的重要信息来源。

日经 BP 社出版局的高畠知子在百忙之中为笔者提供了悉心的指导，提出了很多优秀的意见，并进行了原稿审读。也要感谢负责图书制作的 Kunimedia 公司的诸位和叶波高人，以及为本书出版创造机会的日经软件杂志的历届主编真岛馨和柳田俊彦。

时间过得好快，在本书第 1 版出版时的 2004 年，笔者慈爱的父亲过世了，距今已经过了 7 年。7 年前还在上小学和幼儿园的两个女儿现在已经是高中生和初中生了。最后，还要感谢支持笔者在家写作的妻子。