第 1 章

简　介

可以在JVM上编程的语言有很多。通过这本书，我希望让你相信花时间学习Scala是值得的。

Scala语言为并发、表达性和可扩展性而设计。这门语言及其程序库可以让你专注于问题领域，而无需深陷于诸如线程和同步之类的底层基础结构细节。

如今硬件已经越来越便宜，越来越强大。很多用户的机器都装了多个处理器，每个处理器又都是多核。虽然迄今为止，Java对我们来说还不错，但它并不是为了利用我们如今手头的这些资源而设计的。而Scala可以让你运用这些资源，创建高响应的、可扩展的、高性能的应用。

本章，我们会快速浏览一下函数式编程和Scala的益处，为你展现Scala的魅力。在本书的其他部分，你将学会如何运用Scala，利用这些益处。

1.1　为何选择 Scala

Scala是适合你的语言吗？

Scala是一门混合了函数式和面向对象的语言。用Scala创建多线程应用时，你会倾向于函数式编程风格，用不变状态（immutable state）[①]编写无锁（lock-free）代码。Scala提供一个基于actor的消息传递（message-passing）模型，消除了涉及并发的痛苦问题。运用这个模型，你可以写出简洁的多线程代码，而无需顾虑线程间的数据竞争，以及处理加锁和释放带来的梦魇。把`synchronized`这个关键字从你的字典中清除，享受Scala带来的高效生产力吧。

然而，Scala的益处并不仅限于多线程应用。你可以用它构建出强大而简洁的单线程

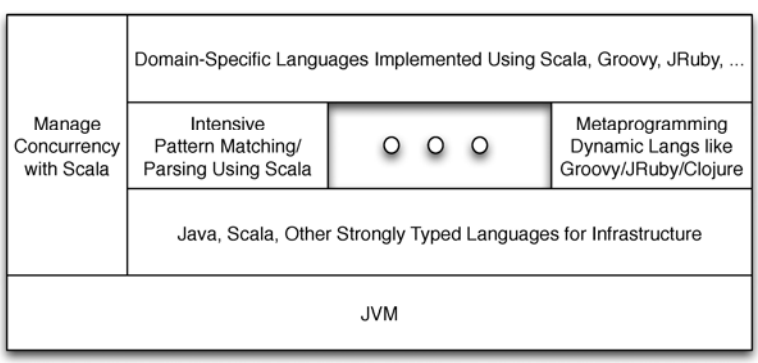
① 对象一旦创建出来，就不再改变其内容，这样的对象就是不变的。这也就无需顾虑多线程访问对象时的竞争管理。Java的`String`就是不变对象一个非常好的例子。

应用，或是多线程应用中的单线程模块。你很快就可以用上Scala的强大能力，包括自适应静态类型、闭包、不变的容器以及优雅的模式匹配。

Scala对于函数式编程的支持让你可以写出简洁而有表现力的代码。感谢更高层的抽象，它让我们可以用更少的代码做更多的事情。单线程应用和多线程应用都可以从函数式风格中受益。

函数式编程语言也为数不少。比如，Erlang就是一个很好的函数式编程语言。实际上，Scala的并发模型同Erlang的非常相似。然而，同Erlang相比，Scala有两个显著的优势。第一，Scala是强类型的，而Erlang不是。第二，不同于Erlang，Scala运行于JVM之上，可以与Java很好地互操作。

就运用在企业级应用的不同层面而言，Scala这两个特性使其成为了首选。只要你愿意，就可以用Scala构建整个企业级应用，或者，也可以把它和其他语言分别用在不同的层上。如果有些层在你的应用中至关重要，你就可以用上Scala的强类型、极佳的并发模型和强大的模式匹配能力。下图的灵感源自Ola Bini的语言金字塔（参见附录A的“Fractal Programming”），它展现了Scala在企业级应用中与其他语言的配合。



JVM上的其他语言Groovy，JRuby，Clojure怎么样呢？

目前为止，能够同时提供函数式风格和良好并发支持的强类型语言，唯有Scala；这正是它的卓越之处。JRuby和Groovy是动态语言，它们不是函数式的，也无法提供比Java更好的并发解决方案。另一方面，Clojure是一种混合型的函数式语言。它天生就是动态的，因此不是静态类型。而且，它的语法类似于Lisp，除非你很熟悉，否则这可不是一种易于掌握的语法。

如果你是个有经验的 Java程序员，正在头痛用Java实现多线程应用，那么你就会发现Scala非常有用。你可以相当容易地就把Java代码封装到Scala的actor中，从而实现线程

隔离。还可以用Scala的轻量级API传递消息，以达到线程通信的目的。与“启动线程，立即用同步的方式限制并发”不同，你可以通过无锁消息传递享受真正的并发。

如果你重视静态类型，喜欢编译器支持所带来的益处，你会发现，Scala提供的静态类型可以很好地为你工作，而不会阻碍你。你会因为使用这种无需键入太多代码的类型而感到惬意。

如果你喜欢寻求更高层次的抽象和具有高度表现力的代码，你会被Scala的简洁所吸引。在Scala里，你可以用更少的代码做更多的事情。了解了运算符和记法，你还会发现Scala的灵活性，这对于创建领域专用语言（domain-specific language）非常有用。

提醒一下，Scala的简洁有时会倾向于简短生硬，这会让代码变得难以理解。Scala的一些运算符和构造对初学者而言可能一时难以适应[①]。这样的语法不是为胆小之人准备的。随着你逐渐精通Scala，你会开始欣赏这种简洁，学会避免生硬，使得代码更易于维护，同时也更易于理解。

Scala不是一种超然物外的语言。你不必抛弃你已经为编写Java代码所投入的时间、金钱和努力。Scala和Java的程序库是可以混合在一起的。你可以完全用Scala构建整个应用，也可以按照你所期望的程度，将它同Java或其他JVM上的语言混合在一起。因此，你的Scala代码可以小如脚本，也可以大如全面的企业应用。Scala已经用于构建不同领域的应用，包括电信、社交网络、语义网和数字资产管理。Apache Camel用Scala做DSL创建路由规则。Lift Web Framework是一个用Scala构建的强大的Web开发框架，它充分利用了Scala的特性，比如简洁、表现力、模式匹配和并发。

1.2 何为 Scala

Scala，是Scalable Language的缩写，它是一门混合型的函数式编程语言。Martin Odersky[②]是它的创始人，2003年发布了第一个版本。下面是Scala的一些关键特性[③]：

- 它拥有基于事件的并发模型；
- 它既支持命令式风格，也支持函数式风格；
- 它是纯面向对象的；
- 它可以很好的与Java混合；
- 它强制使用自适应静态类型；

① 我着手学习一门新语言时，还没有哪门语法不让我头疼的，包括Ruby。多多练习，很快语法就变得很自然了。

② 请阅读附录A，了解更多信息。

③ 请参考附录A，获得权威的语言规范。

- 它简洁而有表现力；
- 它构建于一个微内核之上；
- 它高度可扩展，可以用更少的代码创建高性能应用。

下面的小例子突出了这些特性：

Introduction/TopStock.scala

```
import scala.actors._
import Actor._

val symbols = List( "AAPL", "GOOG", "IBM", "JAVA", "MSFT")
val receiver = self
val year = 2008

symbols.foreach { symbol =>
  actor { receiver ! getYearEndClosing(symbol, year) }
}

val (topStock, highestPrice) = getTopStock(symbols.length)

printf("Top stock of %d is %s closing at price %f\n", year, topStock,
  highestPrice)
```

不用想语法，我们先从大处着眼。symbols指向一个不变的List，其中持有股票代码。我们对这些股票代码进行循环，调用actor。每个actor在单独的线程中执行。因此，同actor关联的代码块({})运行在其自己的线程上。它调用（尚未实现的）函数getYearEndClosing()。这个调用的结果返回发起请求的actor。这由特殊的符号（!）实现。回到主线程，我们调用（尚未实现的）函数getTopStock()。在上面的代码完全实现之后，我们就可以并发地查询股票收盘价了。

现在，我们看看函数getYearEndClosing()：

Introduction/TopStock.scala

```
def getYearEndClosing(symbol : String, year : Int) = {
  val url = "http://ichart.finance.yahoo.com/table.csv?s=" +
    symbol + "&a=11&b=01&c=" + year + "&d=11&e=31&f=" + year + "&g=m"

  val data = io.Source.fromURL(url).mkString
  val price = data.split("\n")(1).split(",")(4).toDouble
  (symbol, price)
}
```

在这个短小可爱的函数里面，我们向http://ichart.finance.yahoo.com发出了一个请求，收到了以CSV格式返回的股票数据。我们解析这些数据，提取年终收盘价。现在，先不必为收到数据的格式操心，它并不是我们要关注的重点。在第14章，我们还将

再用到这个例子，提供所有与Yahoo服务交流的细节。

还需要实现getTopStock()方法。在这个方法里，我们会收到收盘价，确定最高价的股票。我们看看如何用函数式风格实现它：

Introduction/TopStock.scala

```
def getTopStock(count : Int) : (String, Double) = {
  (1 to count).foldLeft("", 0.0) { (previousHigh, index) =>
    receiveWithin(10000) {
      case (symbol : String, price : Double) =>
        if (price > previousHigh._2) (symbol, price) else previousHigh
    }
  }
}
```

在这个getTopStock()方法中，没有对任何变量进行显式赋值的操作。我们以股票代码的数量作为这个方法的参数。我们的目标是找到收盘价最高的股票代码。因此，我们把初始的股票代码和高价设置为（"", 0.0），以此作为foldLeft()方法的参数。我们用foldLeft()方法去辅助比较每个股票的价格，确定最高价。通过receiveWithin()方法，我们接收来自开始那个actor的股票代码和价格。如果在指定时间间隔没有收到任何消息，receiveWithin()方法就会超时。一收到消息，我们就会判断收到的价格是否高于我们当前的高价。如果是，就用新的股票代码及其价格作为高价，与下一次接收的价格进行比较。否则，我们使用之前确定的（previousHigh）股票代码和高价。无论从附着于foldLeft()的代码块（code block）中返回什么，它都会作为参数，用于在下一元素的上下文中调用代码块。最终，股票代码和高价从foldLeft()返回。再强调一次，从大处着眼，不要管这里的方法的细节。随着学习的深入，你会逐步了解它们的详细内容。

大约25行代码，并发地访问Web，分析选定股票的收盘价。花上几分钟，分析一下代码，确保你理解了它是如何运作的。重点看方法是如何在不改变变量或对象的情况下，计算最高价的。整个代码只处理了不变状态；变量或对象在创建后就没有修改。其结果是，你不需要顾虑同步和数据竞争，代码也不需要有显式的通知和等待序列。消息的发送和接收隐式地处理了这些问题。

如果你把上面所有的代码放到一起，执行，你会得到如下输出：

```
Top stock of 2008 is GOOG closing at price 307.650000
```

假设网络延迟是d秒，需要分析的是n个股票代码。如果编写代码是顺序运行，大约要花n×d秒。因为我们并行执行数据请求，上面的代码只要花大约d秒即可。代码中最大的延迟会是网络访问，这里我们并行地执行它们，但并不需要写太多代码，花太多精力。

想象一下，用Java实现上面的例子，你会怎么做。

上面的代码的实现方式与Java截然不同，这主要体现在下面3个方面。

- 首先，代码简洁。Scala一些强大的特性包括：actor、闭包、容器（collection）、模式匹配、元组（tuple），而我们的示例就利用了其中几个。当然，我还没有介绍过它们，这还只是简介！因此，不必在此刻就试图理解一切，通读本书之后，你就能够理解它们了。
- 我们使用消息进行线程间通信。因此不再需要`wait()`和`notify()`。如果你使用传统Java线程API，代码会复杂几个数量级。新的Java并发API通过使用executor服务减轻了我们的负担。不过，相比之下，你会发现Scala基于actor的消息模型简单易用得多。
- 因为我们只处理不变状态，所以不必为数据竞争和同步花时间或精力（还有不眠夜）。

这些益处为你卸下了沉重的负担。要详细地了解使用线程到底有多痛苦，请参考Brian Goetz的*Java Concurrency in Practice* [Goe06]。运用Scala，你可以专注于你的应用逻辑，而不必为低层的线程操心。

你看到了Scala并发的益处。Scala也并发地[①]为单线程应用提供了益处。Scala让你拥有选择和混合两种编程风格的自由：Java所用的命令式风格和无赋值的纯函数式风格。Scala允许混合这两种风格，这样，你可以在一个线程范围内使用你最舒服的风格。Scala使你能够调用和混合已有的Java代码。

在Scala里，一切皆对象。比如，`2.toString()`在Java里会产生编译错误。然而，在Scala里，这是有效的——我们调用`Int`实例的`toString()`方法。同时，为了能给Java提供良好性能和互操作性，在字节码层面上，Scala将`Int`的实例映射为32位的基本类型`int`。

Scala编译为字节码。你可以按照运行Java语言程序相同的方式运行它。[②]也可以很好的将它同Java混合起来。你可以用Scala类扩展Java类，反之亦然。你也可以在Scala里使用Java类，在Java里使用Scala类。你可以用多种语言编写应用，成为真正的多语言程序员[③]——在Java应用里，在需要并发和简洁的地方，就用Scala（比如创造领域特定语言）吧！

Scala是一个静态类型语言，但是，不同于Java，它拥有自适应的静态类型。Scala

① 这里一语双关。——编者注
② 你可以把它当作脚本运行。
③ 参见附录A，也请阅读Neal Ford著的*The Productive Programmer* [For08]。

在力所能及的地方使用类型推演。因此，你不必重复而冗繁地指定类型，而可以依赖语言来了解类型，在代码的剩余部分强制执行。不是你为编译器工作；相反，编译器为你工作。比如，我们定义`var i = 1`，Scala立即就能推演出变量`i`是`Int`类型。现在，如果我们将某个字符串赋给那个变量，比如，`i = "haha"`，编译器就会给出如下的错误：

```
error: type mismatch;
 found   : java.lang.String("haha")
 required: Int
       i = "haha"
```

在本书后面，你会看到类型推演超越了简单类型定义，也进一步超越了函数参数和返回值。

Scala偏爱简洁。在语句结尾放置分号是Java程序的第二天性。Scala可以为你的小拇指能从多年的虐待中提供一个喘息之机——分号在Scala中是可选的。但是，这只是个开始。在Scala中，根据上下文，点运算符（.）也是可选的，括号也是。因此，不用写成`s1.equals(s2);`，我们可以这么写`s1 equals s2`。去掉了分号、括号和点，代码会有一个高信噪比。它会变成更易编写的领域特定语言。

Scala最有趣的一个方面是可扩展性。你可以很好享受到函数式编程构造和强大的Java程序库之间的相互作用，创建高度可扩展的、并发的Java应用，运用Scala提供的功能，充分发挥多核处理器的多线程优势。

Scala真正的魅力在于它内置规则极少。相比于Java，C#和C++，Scala语言只内置了一套非常小的内核规则。其余的，包括运算符，都是Scala程序库的一部分。这种差异具有深远的影响。因为语言少做一些，你就能用它多做一些。这是真正的可扩展，它的程序库就是一个很好的研究案例。

1.3 函数式编程

我已经提过几次，Scala可以用作函数式编程语言。我想花几页的篇幅给你一些函数式编程的感觉。让我们从对比Java编程的命令式风格开始吧！如果我们想找到给定日期的最高气温，可能写出这样的Java代码：

```
//Java code
public static int findMax(List<Integer> temperatures) {
  int highTemperature = Integer.MIN_VALUE;
  for(int temperature : temperatures) {
   highTemperature = Math.max(highTemperature, temperature);
  }
  return highTemperature;
}
```

我们创建了一个可变的变量highTemperature，在循环中不断修改它。当你拥有可变变量时，你就必须保证正确地初始化它们，在正确的地方将它们改成正确的值。

函数式编程是声明式风格，使用这种风格，你要说明做什么，而不是如何去做。如果你用过XSLT，规则引擎，或是ANTLR，那么你就已经用过函数式风格了。我们用函数式风格重写上面的代码，不用可变变量，如下代码所示：

Introduction/FindMaxFunctional.scala

```
def findMax(temperatures : List[Int]) = {
  temperatures.foldLeft(Integer.MIN_VALUE) { Math.max }
 }
```

上面代码里，你看到了Scala的简洁和函数式编程风格的相互作用。这是段高密度的代码。用几分钟时间沉淀一下。

我们创建了一个函数findMax()，接收一个不变的容器（temperatures）为参数，表示温度值。圆括号和花括号之间的“=”告诉Scala推演这个函数的返回类型（这里是Int）。

在这个函数里，我们调用这个collection的foldLeft()方法，对容器中的每个元素运用Math.max()。正如你所知道的，java.lang.Math类的max()方法接收两个参数，就是我们要确定最大值的两个值。在上面的代码里，这两个参数是隐式传递的。max()的第一个隐式参数是之前的高值，第二个参数是foldLeft()正在迭代的容器中的当前元素。foldLeft()取回调用max的结果，这就是当前的高值，在接下来调用max()时把它传进去，同下一个元素比较。foldLeft()的参数就是高温的初始值。

foldLeft()方法需要花些功夫来掌握。稍稍做个假设，把容器中的元素当作是站成一排的人，我们要找出年纪最大的人的年龄。我们在笔记上写上0，把它传给这排的第一个人。第一个丢弃这个笔记（因为他比0岁年龄大）；用他的年龄20创建一个新的笔记；把它传给这排的下一个人。第二个人，他比20岁年轻，简单把笔记传给下一个挨着他的人。第三个人，32岁，丢弃这个笔记，创建一个新的传递下去。我们从最后一个人获得的笔记就会包含年纪最大的人的年龄。把这一系列过程可视化，你就知道foldLeft()背后做了些什么。

上面的代码是不是感觉像喝了一小口红牛？Scala代码高度简洁，非常紧凑。你不得不花些功夫学习这个语言。但是，一旦你掌握了它，你就能够利用它的威力和表现力了。

我们来看另外一个函数式风格的例子。假定我们想要一个List，其元素就是将原List值的翻倍。我们不会对每个元素进行循环来实现，只要简单的说，我们要元素翻倍，让语言来循环，如下所示：

Introduction/DoubleValues.scala

```
val values = List(1, 2, 3, 4, 5)

val doubleValues = values.map(_ * 2)
```

关键字val理解为“不变的”。我们告诉Scala，变量values和doubleValues一旦创建就不会改变。

尽管看上去不像，但_*2确实是一个函数。它是个匿名函数，这表示这个函数只有函数体，而没有函数名。下划线（_）表示传给这个函数的参数。函数本身作为参数传给map函数。map()函数在容器上迭代，对于容器中的每个元素，都会调用以参数给出的匿名函数。其结果是创建一个新的List，包含的元素就是原List元素值的翻倍。

看见怎么把函数（这里就是把一个数翻倍）当作普通参数和变量了吧？在Scala里面，函数是一等公民。

因此，虽然获得了一个将原List元素值翻倍的List，但我们并没有修改任何变量和对象。这种不变的方式是一个关键概念，它让函数式编程成为一种非常有吸引力的并发编程风格。在函数式编程中，函数是纯粹的。它们产生的输出只是基于其接收到的输入，它们不会受任何状态影响或也不会影响任何状态，无论是全局还是局部的。

1.4 本书的内容

我写这本书的目标是让你快速理解Scala，可以用它编写并发、可伸缩、有表现力的程序。为了做到这些，你需要学很多东西，但是还有很多你也不必了解。如果你的目标是了解Scala的全部，本书满足不了你。我们已经有了这样一本书，叫*Programming in Scala* [OSV08]，由Martin Odersky、Lex Spoon和Bill Venners编写，它相当深入的介绍了这门语言，非常值得一读。本书里讲述的是开始使用Scala所需的一些必要概念。

我假定你非常熟悉Java。因此，你并不会从这本书里面学到基本的编程概念。然而，我并不假定你拥有函数式编程的知识，或是了解Scala语言本身——你会在本书里学到。

我是为忙碌的Java开发者编写的这本书，因此我的目标是让你很快觉得Scala很舒服，以便你可以很快地开始用它构建真实的应用。你会看到概念介绍得相当快，但会提供很多例子帮助你理解它们的。

本书的其余部分按照如下方式组织。

在每章里，你都会学到一些必需的知识点，让你更接近于用Scala编写并发代码。

第2章，起步。这一章我会带着你安装Scala，让你的第一个Scala代码执行起来。我会为你展示如何把Scala当作脚本用，如何像传统的Java代码一样编译，以及如何使用Java工具运行它。

第3章，Scala步入正轨。从这一章开始，你会拥有一次快速Scala之旅，了解它的简洁，了解它如何处理Java类和基本类型，如何在已有Java知识的基础上学习新内容。对于那些毫无戒心的Java程序员而言，Scala还是有些惊奇的，你会在这章看到这些惊奇。

第4章，Scala的类。作为一门纯粹的面向对象语言，Scala处理类的方式与Java有相当大的差异。比如，它没有`static`关键字，然而你可以用伴生对象创建类成员。你会在这一章中学到Scala的OO编程方式。

第5章，自适应类型[①]。Scala是一种静态类型语言。它提供了编译时检查，但是与其他静态类型语言不同，它没有繁文缛节的[②]语法。在这一章中，你会学到Scala轻量级自适应类型。

第6章，函数值和闭包。函数值和闭包是函数式编程的核心概念，也是Scala的一个最常见特征。在这一章中，我会带你领略如何善用它们。

第7章，Trait和类型转换。你会学到如何抽象行为，并将其混入任意的类中，也会了解到Scala的隐式类型转换。

第8章，使用容器，Scala提供可变和不变的容器。你可以很简洁的创建它们，通过闭包进行迭代，正如你在这一章所见到的。

第9章，模式匹配和正则表达式。从这章开始，你会开始探索模式匹配功能。它是Scala最强大的特性之一，也是你需要在并发编程中依赖的一个特性。

第10章，并发编程。在这一章，你会读到本书中最令人期待的特性。你会学习到强大的基于事件的并发模型和用做支撑的actor API。

第11章，与Java混合。一旦你解决了如何使用并发的问题，你就会想把它用到你的Java应用里面了。这一章会为你展示如何做到这一点。

第12章，Scala的单元测试。如果你想确保你键入的代码确实做了你想做的事情，那

---

① 自适应类型（Sensible Typing），并不是Scala本身的术语，而是作者为了形容Scala类型的特性而想出的一个说法。这章主要是描述Scala的类型的推演能力和类型体系。从字面理解，是有意识或有知觉的确定类型，把Scala比作一个生命体，用以形容Scala的类型特征。这里把它译作自适应类型。

——译者注

② 参见附录A。

么Scala拥有的单元测试可以提供良好的支持。在这一章中，你会学习如何使用JUnit、TestNG和基于Scala的测试工具，测试Scala和Java代码。

第13章，异常处理。我知道你能写出很好的代码。然而，你还是不得不处理你调用代码所抛出的异常。Scala有一种不同于Java的异常处理方法，你会在这一章中看到。

第14章，使用Scala。在这章中，我会把这本书的概念放到一起，为你展示如何善用Scala构建真实世界的应用。

最后，在附录A中，你会找到本书所引用的一些Web上的文章和blog。

1.5 本书面向的读者

这本书是为有经验的Java程序员准备的。也就是说你要相当熟悉Java语言的语法和Java API，而且你也要有扎实的面向对象编程知识。基于这样的前提，你就可以快速领会到Scala的精髓，用它构建真实的应用。

熟悉其他语言的程序员也可以使用这本书，但是不得不读一些Java的好书，补充一些营养。

对Scala有几分了解的程序员也可以使用本书，了解一些他们尚未得到机会探索的语言特性。已经熟悉Scala的人可以用这本书来培训他们组织中的其他程序员。

1.6 致谢

编写本书的过程中，我拥有着一些特权，这些特权使我能够从很多智者那里获得帮助。这群非常有激情的人都是在百忙之中贡献出他们的时间评论本书，告诉我哪里不足，哪里做得好，鼓励我继续前行。这本书能够变得更好，我需要鸣谢Al Scherer、Andres Almiray、Arild Shirazi、Bill Venners、Brian Goetz、Brian Sam-bodden、Brian Sletten、Daniel Hinojosa、Ian Roughley、John D. Heintz、Mark Richards、Michael Feathers、Mike Mangino、Nathaniel Schutta、Neal Ford、Raju Gandhi、Scott Davis和Stuart Halloway。他们影响着这本书向许多好的方面进步。你在本书中发现的任何错误，责任完全在我。

特别要鸣谢Scott Leberknight；他是我遇到过最细心的评论者。他的评论如此详尽且见解深刻，他花时间运行了书中的每一段代码。在一些我需要帮忙的地方，他总是非常友好的帮我再过一遍。

一本编程语言书的作者所能要求的，还有什么能比让语言的创造者对书进行审校更好的呢？我诚挚的感谢Martin Odersky，感谢他那无价的评论、修正和建议。

你在读的这本书经过了良好的打磨、修正、细化和重构。有一个人勇于阅读和编辑每个单词，就如同是它们只是通过我指尖流露出来的一般。他做到了，唯一的延迟是互联网强加给我们的。他为我展示一个人可能对你是如何的严格，与此同时，又能够不断地激励你。我承诺再写一本书，如果他承诺再编辑的话。我从心底里感谢Daniel Steinberg。

我要特别鸣谢Pragmatic Programmers，Andy Hunt和Dave Thomas，他们开启了这本书的项目，并支撑着完成它。感谢你们提供了如此敏捷的环境和设置了如此高的标准。很高兴再次为你们写书。感谢Janet Furlow、Kim Wimpsett、Steve Peter以及整个Pragmatic Bookshelf团队，有了你们的协助，才有了这本书。

我还要鸣谢Dustin Whitney、Jonathan Smith、Josh McDonald、Fred Jason、Vladimir Kelman和Jeff Sack，感谢他们在本书论坛（参见附录A）和email交流中给予我的鼓励。我还要鸣谢本书beta版的读者，他们提供了很有价值的评论和反馈。感谢Daniel Glauser、David Bailey、Kai Virkki、Leif Jantzen、Ludovic Kuty、Morris Jones、 Peter Olsen和Renaud Florquin为beta版报告的错误。

感谢Jay Zimmerman，NFJS系列大会（http://www.nofluffjuststuff.com）的主管，他为我提供了机会，展现一些想法和主题，正是这些内容帮我塑成了本书。感谢与会的geek——演讲者和参会者——让我有机会与你们交流。你们是灵感之源，我从你们身上学到了很多。

我还要“并发”地鸣谢Martin Odersky和Scala社区，他们的付出让我们拥有了如此美妙的语言。

感谢我的妻子Kavitha同两个儿子Karthik和Krupakar，没有你们的巨大支持、耐心和鼓励，编写本书是不可能的。这本书始于Krupa问“爸爸，Scala是什么？”，止于Karthik说“我今年夏天要学Scala”，以及我妻子在其间不断稳定提供的垃圾食品、咖啡因饮料与刨根问底的问题。下面这段完全函数式的Scala代码是献给他们的：`("thank you! " * 3) foreach print`。