Scala语言编程入门

<http://www.qnr.cn/pc/java/fudao/zhuanye/201204/775522_3.html>

为什么选用 Scala

    当今的企业和互联网应用程序必须平衡一系列的要点。它们要有快速而可靠的实现。新的功能要能在短时间内通过周期渐进的方式加入。除了简单地提供商务逻辑之外，应用程序必须支持访问安全控制，数据持久化，事务，和其它的进阶功能。程序必须高度易用和可扩展，同时要求支持并发和分布式计算。应用程序会被网络化，并且提供人和机器都易于使用的接口。  
    要达成这些挑战，许多软件开发者在寻找新型的编程序言和工具。以往备受推崇的如：Java，C#，和C++ 已经不再是开发这些次世代应用程序的最佳候选。  
    如果你是一个Java 程序开发者  
    Java 是由Sun 公司在1995 年，互联网高速发展的时候正式引入的。 由于当时需要一个安全的，可移植的，开发者友好的程序语言，它被迅速地推崇为编写浏览器应用的理想语言。而当时的主流语言，C++，则并不适合这个领域。  
    今天，Java 被更多地使用在服务器端程序中。它是开发网络和企业应用的最流行的语言之一。  
    然而，Java 是它们那个时代的产物，至今也有一定年代了。在1995年，Java 为了拉拢C++开发者，提供了和C++ 足够相似的语法，同时也避开了C++ 语言上的缺陷和危险。Java 采纳了绝大多数那个时代对解决软件开发问题有用的概念，比如面向对象编程（OOP）， 同时也丢弃了一些麻烦的充满问题的技术，比如人工的内存控制。这些设计决策在最小化复杂度和最大化开发生产力上达到了一个优异的平衡。然而，从Java 出生演化到现在，许多人认为它变得越来越复杂，而且并没有显著地解决新的程序开发过程中面临的问题和挑战。  
    程序开发者想要一种更精炼和更灵活的语言去提高他们的生产效率。这也是如今所谓的Ruby ，Python 这样的脚本（Script）语言大行其道的原因之一。  
    永无休止的需求驱动着架构向大规模并发开发进行。然而，Java 的并发模型是基于对共享的，可变的信号状态的同步存取，从而导致了复杂的，易错的程序。  
    当Java 渐渐老化时，运行它的 Java 虚拟机（JVM）却持续地散发着光芒。如今JVM 的性能优化是非凡的，它允许字节码（byte code）在许多情况下得到和本地编译的代码相同的性能。今天，许多程序开发者相信使用基于JVM 的新语言是正确的前进道路。Sun 显然是拥护这个趋势的，他们雇佣了JRuby 和Jython （Ruby 和Python 在JVM 上的实现）的主要开发者。  
    Scala 的出现对于Java 开发者来说提供了一个更加新式的语言。同时保留了JVM 的惊人的性能和开发了十几年的Java 库的宝贵财富。  
    如果你是一个Ruby，Python 的开发者  
    像Ruby，Python，Groovy，JavaScript，和Smalltalk 这样的动态类型语言，通常因为它们优雅的灵活性，强大的元编程能力（metaprogramming），提供了很高的生产力。  
    如果撇开它们在高产能上的优势，动态语言也许不是个万金油，特别对于大规模和高性能程序来说，不是最佳选择。在编程社区里，有一个对于动态类型和静态类型究竟谁更占有优势进行的冗长争论。很多的比较观点多少都有些主观。我们不会在这里讨论所有的这些争论，但是我们会提供一些对此的看法以供参考。  
    相比静态语言来说，优化动态语言的性能更富有挑战性。在静态语言中，优化器可以根据类型信息来进行决策。而在动态语言中，只有很有限的信息是可用的，这使得优化器的选择更加困难。虽然近年来在动态语言优化方面的提升渐渐浮现希望，但是它们还是落在静态语言的优化艺术的后面。所以，如果你确实需要很高的性能，静态语言或许是一个更保险的选择。  
    静态语言同样可以使开发过程获得好处。集成开发环境（IDE）的一些功能，比如自动完成（有时候被称为智能感知）在静态语言中更容易完成，因为那些类型信息都是可用的。而更加明显的类型信息在静态代码中促进了代码的自我解释，随着项目的发展，这对于开发者意图的互相交流是十分重要的。  
    当使用一种静态语言时，你必须时刻考虑使用恰当的类型。这迫使你在选择设计时更加小心。这虽然会拖慢日常的设计决策，但是长此以往，在应用程序中对类型使用的思考会带来更为清晰的设计。  
    静态语言的另外一个小的好处就是编译时期的额外检查。我们通常认为这个优势被夸大了，因为类型不匹配的错误只是日常见到的运行时错误中的一小部分。编译器无法发现逻辑错误，这显然更加重要。只有一个综合的，自动的测试组可以发现逻辑错误。对于动态语言来说，测试也必须覆盖可能的类型错误。如果你以前编写过动态类型语言，你会发现你的测试组其实会小一些，但不会小很多。  
    许多开发者发现静态语言太过冗长，抱怨静态类型是冗长的元凶，而事实上真正的原因是缺少类型推断。在类型推断的情况下，编译器会根据上下文推断值的类型。例如，编译器会识别在 x = 1 + 3 中x 是一个整型。类型推断能显著地减少代码的长度，使得代码更像是用动态语言编写出来的。  
    我们都曾经在不同的时间和静态语言和动态语言打过交道。我们发现两种类型的语言都因为不同的原因被广为关注。我们相信现代软件开发者必须掌握一系列的语言和工具。有时，动态语言会是完成工作的正确工具；而有时，一个静态语言，例如Scala，会是你正需要的。  
    Scala 介绍  
    Scala 是一种迎合现代软件开发者需求的语言。它是静态的，混合范式的（mixed-paradigm），基于JVM 的语言；它在拥有简洁，优雅，灵活的语法的同时，也提供了一个久经考验的类型系统和惯用语法，所以从小巧的解释性脚本到大型的复杂系统它都可以胜任。那可是一大口蛋糕，所以，让我们详细地来了解下它的一些特性。  
    静态类型  
    正如我们在前面的章节所描述的，一个静态类型的语言在一个变量的生命周期内都会绑定一个类型。相反的，动态类型的语言则是根据变量所引用的值来绑定类型，这意味着变量的类型可以随着它引用的值改变而改变。  
    在最新的基于JVM 的语言中，Scala 是为数不多的静态类型语言，而且是最出名的一个。  
    混合范式 - 面向对象编程  
    Scala 完全支持面向对象编程（OOP）。Scala 在改进Java 对OOP 的支持的同时，添加了traits （特性）的概念，它可以简洁地实现类之间的混合关系。Scala 的traits 和Ruby 的modules （模块）概念类似。如果你是一个Java 开发者，可以把traits 想象成interfaces （接口）和implementations （实现）的统一体。  
    在Scala 中，所有的东西实际上都是一个object （对象）。Scala 不像Java，它没有原始类型（元数据类型）。相反的，所有的数值类型都是正真的objects。 然而，为了优化性能，Scala 会实时地在底层实现中使用原始类型。另外，Scala 不支持static （静态）或者class-level members （类级别成员）的类型，因为它们并没有和一个实例（instance）关联。相反，Scala 支持单例模式，可以应用于那些一种类型只有一个实例的情况。  
    混合范式 - 函数式编程  
    Scala 完全支持函数式编程（FP）。FP 是一种比OOP 更加古老的编程范式，它被学术界的象牙塔庇护至今。FP 因为简化了某些设计，尤其是并发上的问题而受到了越来越多的关注。“纯粹”的函数式语言不允许任何易变状态（mutable state），因而避免了对易变状态的同步和共享访问。相反的，用纯函数式语言编写的程序在并发自主的进程中通过传递消息来通信。Scala 通过Actors 库来支持这种模式，但是它同时允许mutable （易变的）和immutable （不易变的）变量。  
    函数是FP 的一类公民，这意味着它们可以被赋值给变量，被传递给其他函数等，就像普通的值一样。这个特色允许通过元操作来组合一些高级行为。因为Scala 遵守所有的东西都是object 的原则，函数在Scala 中也是objects。  
    Scala 同时支持闭包，一种动态语言比如Python 和Ruby 从函数式编程世界中引用过来的特性。Java 很遗憾地没有在最近的版本中包含这个特性。闭包本质上是一个函数和其引用的变量的统一定义。这些变量不作为传入参数或者函数内的局部变量。一个闭包封闭了这些引用，所以函数调用可以安全的引用这些变量，即使它们已经超出了函数的作用域。闭包是一个强大的抽象，以至于objects 系统和基础控制结构经常是用它们实现的。  
    一种同时有JVM 和。NET 实现的语言  
    Scala 是众所周知的基于JVM 的语言，这意味着Scala 会生成JVM 字节码。一个能生成CLR 字节码的基于。NET 的Scala 版本也同时在开发中。当我们提到底层的“运行时”时，我们通常是指JVM。但是许多我们提到的概念能同时运用于两种不同的运行时。当我们讨论有关JVM 的细节时，它们大致也能应用于。NET，除非我们特别说明。  
    Scala 的编译器使用了一些聪明的技巧来映射Scala 的扩展到相应的字节码。在Scala 里，我们可以轻松地调用产生自Java 源代码（JVM）或者C# 源代码（。NET）的字节码。同样的，你也可以在Java，C# 代码里调用Scala。运行在JVM 和CLR 上，允许Scala 开发者来充分利用现有的库来和运行在这些运行时的其他语言交互。  
    简洁的，优雅的，灵活的语法  
    Java 的语法实际上有些冗长。 Scala 使用了一些技巧来减少不必要的语法，使得Scala 源码几乎和其他的动态语言一样简洁。类型推断使得显式的类型声明信息在大多数场合下减少到了最低。类型和函数的声明变得非常简洁。  
    Scala 允许函数名字包含非字母数字的字符。结合一些语法上的技巧，这些特性允许用户定义一些看起来像操作符的方法。这样，在语言核心之外的库对于用户看来就会比较自然。  
    复杂的类型系统  
    Scala 扩展了Java 的类型系统，同时提供了更灵活的类型和一些高级的类型结构。这个类型系统起初开起来可能会有些恐怖，但是大多数时候你不用担心这些高级的结构。类型推断帮助你自动推断类型签名，所以用户不用人工提供一般的类型信息。不过，当你需要它们的时候，高级类型特性可以给你提供更灵活的方式，用类型安全的方式解决设计上的问题。  
    可伸缩性 - 架构  
    Scala 被设计来服务从小的，解释性脚本到大型的，分布式系统。Scala 提供了4 种语言机制来提供更灵活的系统组合：1）显式的自我类型（selftype），2）类型成员和种类的抽象，3）嵌套的类，以及4）使用traits 的混合结构。  
    没有其它的语言同时提供所有这些机制了。这些机制一起允许用一种类型安全和简洁的方式来构建由可重用组件组成的程序。正如我们所见，许多常见的设计模式和架构技术，例如依赖注入模式，可以容易地用Scala 来实现，而不用冗长的样板代码或者XML 配置文件 -- 它们经常让Java 开发变得很枯燥。  
    可伸缩性 - 性能  
    因为Scala 代码运行在JVM 和CLR 上，它能获得所有来自这些运行时和支持性能灵活性调整的第三方工具的性能优化，例如分析器（profiler），分布式缓存库，集群机制等。如果你相信Java 和C# 的性能，那么你就能信任Scala 的性能。当然，一些特别的结构在这个语言环境下和某些库中相比其它语言会运行地特别高效或者低效。一如既往的，你应该在必要的时候分析和优化你的代码。  
    表面上看起来OOP 和FP 像是不兼容的。但实际上，Scala 的设计哲学是OOP 和FP 应该协同合作而不是对立。其中一方的特性应该能增强另一方。  
    在FP 里，函数没有副作用，变量都是不易变的。而在OOP 中，可变状态和副作用都十分常见，甚至是被鼓励的。Scala 让你来选择解决设计问题最佳的方式。函数式编程对于并发特别有用，因为它摒弃了对易变状态的同步访问。然而，“纯”函数式语言是十分严格的。有一些设计问题还是使用易变对象比较容易解决。  
    Scala 的名字来自词语：可伸缩的语言（scalable language）的缩写。这就建议Scala 的发音为scale-ah，实际上Scala 的创建者发音为scah-lah，像[意大利语](http://www.qnr.cn/waiyu/xyz/Italian/)中的“stairs”（楼梯）。也就是说，两个“a 的”的发音是一样的。  
    Martin Oderskey  的在计算机语言方面的背景和经验是显著的。在你学习Scala 的时候，你会了解这是一个仔细考虑了设计决策，利用了所有类型理论，OOP 和FP 的艺术的语言。Martin 在JVM 方面的经验对于Scala 和JVM 平台的优雅结合有着显著的帮助。它综合了OOP 和FP 的优点，是一个优秀的两全其美的解决方案。  
    Scala 的诱惑  
    今天，我们的产业幸运地拥有许多语言方面的选择。动态语言的能力，灵活性，优雅已经使它们十分流行。但是，Java 和。NET 库，已经JVM 和CLR 的性能作为珍贵的宝藏，符合了许多实际的企业和互联网项目的需求。  
    Scala 引起众人的兴趣是因为它的简洁语法和类型推断，看起来像动态脚本语言。但是，Scala 提供了所有静态类型的优势，一个现代的对象模型，函数式编程，和先进的类型系统。这些工具允许你建立一个可伸缩的，模块化的应用程序，并且重用之前的Java 和。NET API， 充分利用JVM 和CLR 的性能。  
    Scala 是面向专业开发者的语言。相比较与Java 和Ruby，Scala 更难掌握。因为它要求OOP，FP 和静态类型方面的技能，这样才能更高效地使用它。它诱使我们偏爱动态语言的相对简洁。但是，这种简洁是一种假象。在一种动态类型语言中，使用元编程特性来实现高级设计经常是必要的。元编程十分强大，但是使用它需要经验，而且会导致代码变得难以理解，维护和调试。在Scala 中，许多类似的设计目标可以通过类型安全的方式来达到，利用它的类型系统和通过traits 实现的混合结构。  
    我们觉得在Scala 的日常使用中所需求的额外努力会迫使我们在设计时更加谨慎。久而久之，这样的几率会导致更加清晰的，模块化的，可维护的系统。幸运的是，你不必所有时候都去追逐Scala 所有复杂的功能。你的大多数代码会简单清晰，就像是用你最喜欢的动态语言写出来的一样。  
    另外一个策略是联合几种简单的语言，比如Java 来做面向对象的代码，Erlang 来做函数式，并发的代码。这样一个分解会工作的很好，如果你的系统能清晰地分解成这些不想关联的部分，并且你的团队能掌控这样一个混杂的环境。Scala 对于那些仅需要一个全能语言的情况是最具吸引力的。也就是说，Scala 代码可以和谐地与其他语言共处，尤其是基于JVM 和。NET 的语言。  
    安装 Scala  
    这个章节描述了如何安装Scala 的命令行工具， 以便可以尽快让Scala 跑起来，这也是运行本书所有范例的必要充分条件。在各种编辑器和集成开发环境（IDE）中使用Scala 的细节，请参见和IDE 集成，在第14章-Scala 工具，库和IDE 支持。本书的范例是用Scala 版本2.7.5.final 来编写和编译， 也是本书在编写时候的最新的发行版；也有部分是用Scala 版本2.8.0 的每夜编译版本，当你读到这本书的时候应该已经最终完成了。  
    注意  
    2.8 版本引入了很多新的特性，我们会在这本书中予以强调。  
    我们在这本书中会选用JVM 版本的Scala。 首先，你必须安装Java 1.4 或更高的版本（推荐1.5 或更高）。如果你需要安装Java，请登录，按照指示在你的电脑上安装Java。  
    Scala 的官方网站 。要安装Scala，去到下载（downloads）页面 。按照下载页面上的指示下载适合你系统环境的安装包。  
    最简单的跨平台安装包是IzPack 安装器。下载Scala 的jar （译注：编译完以后的Java 专属格式）文件，可以选择scala-2.7.5.final-installer.jar 或者 scala-2.8.0.N-installer.jar, N在这里是2.8.0 版本的最新发布版本。在终端窗口中（译注：或者Windwos 下的命令行），定位到下载的目录，使用java 命令来安装Scala。假设你下载了scala-2.8.0.final-installer.jar，运行如下命令会带领你完成安装过程。  
    java -jar scala-2.8.0.final-installer.jar  提示  
    在苹果系统下（Mac OS X），安装Scala 的最简单方式是使用MacPorts。按照这个页面的安装指示，然后使用 sudo port insall scala. 不用几分钟你就可以运行Scala 了。

在本书中，我们会使用符号scala-home 来指定Scala 安装路径的根目录。  
    注意  
    在Unix，Linux，和Mac OS X 系统下，你需要root 用户权限，或者sudo 命令才能把Scala 安装在一个系统目录下。比如：  
    scala-home = /usr/local/scala-2.8.0.final. 或者，你也可以下载并且展开压缩过的tar 文件（比如scala-2.8.0.final.tgz）或者zip 文件（scala-2.8.0.final.zip）。在类Unix 系统中，展开压缩文件到一个你选择的路径。然后，把scala-home/bin 子目录加入到你的PATH 环境变量中。例如，如果你安装到 /usr/local/scala-2.8.0.final，那么把/usr/local/scala-2.8.0.final/bin 加入到PATH。  
    要测试你的安装，在命令行下运行如下命令：  
    scala -version  我们会在后面学习如何使用scala 命令行。你应该能获得如下输出：  
    Scala code runner version 2.8.0.final -- Copyright 2002-2009, LAMP/EPFL  当然，你看到的版本号会根据你安装的版本而改变。从现在起，当我们展示命令行输出时候如果包含版本号，我们会使用2.8.0.final。  
    祝贺你，你已经安装了Scala！如果你在运行scala 后获得一个错误消息：command not found（无法找到命令），重新检查你的PATH 环境变量，确保它被正确地设立，并包含了正确的bin 目录。  
    注意  
    Scala 2.7.X 以及之前的版本和JDK 1.4 以及更新的版本兼容。Scala 2.8 版本舍弃了1.4 的兼容性。注意Scala 会使用原生的JDK 类库，比如String 类。在。NET 下，Scala 使用对应的。NET 类。  
    同时，你应该能在那个下载页面找到Scala API 文档和源码的下载链接。  
    更多信息  
    在探索Scala 的时候，你会在这个网页上发现有用的资源。你会发现开发支持工具和库的链接，还有教程，语言规范【ScalaSpec2009】，和描述语言特性的学术论文。  
    Scala 工具和API 的文档特别有用。你可以在这个页面浏览API 文档。这个文档是使用scaladoc 工具生成的，类似于Java 的javadoc 工具。参见第14章- Scala 工具，库和IDE支持的“scaladoc 命令行工具”来获得更多的信息。  
    你也可以在下面页面下载一个API 文档的压缩文件来本地浏览。或者你可以用sbaz 工具来安装，如下  
    sbaz install scala-devel-docs sbaz 和scala，scalac 命令行工具安装同样的bin 目录下。安装的文档也同样包含了scala 工具集（包括sbaz）的细节和代码示例。要获取更多Scala 命令行工具和其他资源的信息，参见第14章- Scala 工具，库和IDE 支持。  
    初尝 Scala  
    是时候用一些实在的Scala 代码来刺激一下你的欲望了。在下面的范例中，我们会描述足够的细节让你明白发生了什么。这一节的目标是给你一个大致概念，让你知道用Scala 来编程是怎么一回事。我们会在以后的各个章节来探索Scala 更多的特性。  
    作为第一个实例，你可以用两种方式来运行它：交互式的，或者作为一个“脚本”。  
    让我们从交互式模式开始。我们可以通过在命令行输入scala，回车，来启动scala 解释器。你会看到如下输出。（版本号可能会有所不同。）  
    Welcome to Scala version 2.8.0.final （Java …）。  Type in expressions to have them evaluated.  Type :help for more information.  scala> 最后一行是等待你输入的提示符。交互式的scala 命令对于实验来说十分方便（参见《第14章-Scala工具，库和IDE支持》的“Scala 命令行工具”章节，来获取更多信息）。一个像这样的交互式解释器被称为REPL：读（Read），评估（Evaluate），打印（Print），循环（Loop）  
    输入如下的两行代码。  
    val book = "Programming Scala" println（book） 实际上的输入和输出看起来会像是这样。  
    scala> val book = "Programming Scala" book: java.lang.String = Programming Scala  scala> println（book）  Programming Scala  scala> 在第一行我们使用了val 关键字来声明一个只读变量 book。注意解释器的输出显示了book 的类型和值。这对理解复杂的声明会很方便。第二行打印出了book 的值 -- Programming Scala。  
    提示  
    在交互模式（REPL）模式下来测试scala 命令是学习Scala 细节的一个非常好的方式。  
    这本书里的许多例子可以像这样在解释器里运行。然而，通常使用我们提到的第二个方式会更加方便：在文本编辑器中或者IDE 中编写Scala 脚本，然后用同样的scala 命令来执行它们。我们会在这章余下的绝大多数部分使用这种方式。  
    用你选择的文本编辑器，保存下面例子中的Scala 代码到一个名为upper1-script.scala 的文件，放在你选择的目录中。  
    // code-examples/IntroducingScala/upper1-script.scala  class Upper {    def upper（strings: String\*）： Seq[String] = {      strings.map（（s:String） => s.toUpperCase（））    }  }  val up = new Upper  Console.println（up.upper（"A", "First", "Scala", "Program"）） 这段Scala 脚本把一个字符串转换到大写。  
    顺便说一下，在第一行有一句注释（在代码例子中是源文件的名字）。Scala 使用和Java，C#，C++等一样的注释方式。一个// 注释会影响整个一行，而/\* 注释 \*/ 方式则可以跨行。  
    要运行这段脚本，打开命令行窗口，定位到对应目录，然后运行如下命令。  
    scala upper1-script.scala 文件会被解释，这意味着它会被编译和执行。你会获得如下输出：  
    Array（A, FIRST, SCALA, PROGRAM） 解释 VS 编译，运行Scala 代码  
    总的来说，如果你在命令行输入scala 而不输入文件名参数，解释器会运行在交互模式。你输入的定义和语句会被立即执行。如果你附带了一个scala 文件作为命令参数，它会把文件作为脚本编译和运行，就像我们的 scala upper1-script.scala 例子一样。最后，你可以单独编译scala 文件，运行class 文件，只要你有一个main 函数，就像你通常使用java 命令一样。（我们会马上给出一个例子）  
    你需要理解有关使用解释模式的局限和单独编译运行之间的一些微妙的区别。我们会在《第14章- Scala 工具，库和IDE 支持》的“命令行工具”部分详细解释这些区别。  
    当我们提及执行一个脚本时，就是说用scala 命令运行一个Scala 源文件。  
    在这个例子里，类Upper （字面意思，没有双关） 里的upper 函数把输入字符串转换为大写，然后作为一个数组返回。最后一行把4个字符串转换完以后输出。  
    为了学习Scala 语法，让我们来更详细地解释一下代码。在这仅有的6行代码里面有许多细节！我们会解释一下基础的概念。这个例子的所有的概念会在这被书的后面几个章节被详细地讲解。  
    在这个例子里，Upper 类以class 关键字开始。类的主体被概括在最外面的大括号中 {…}。  
    upper 方法的定义在二行，以def 关键字开始，紧接着是方法名，参数列表，和方法的返回类型，最后是等于号“=”，和方法的主体。  
    在括号里的参数列表实际上是一个String（字符串）类型的可变长度参数列表，由冒号后面后面的String\* 类型决定。也就是说，你可以传入任意多的，以分号分隔的字符串（包括空的列表）。这些字符串会被存在一个名为strings 的参数中。在这个方法里面，strings 实际上是一个Array（数组）。  
    注意  
    当在代码里显式地为变量指定类型信息时，类型注解应该跟在变量名的冒号后面（也就是类Pascal 语法）。Scala 为什么不遵照Java 的惯例呢？ 回想一下，类型信息在Scala 中经常是被推断出来的（不像Java），这意味着我们并不总是需要显式的声明类型。和Java 的类型习惯比较，item: type 模式在你忽略掉冒号和类型注解的时候，更容易被编译器清楚地分析。  
    方法的返回类型在参数列表的最后出现。在这个例子里，返回类型是Seq[String]，Seq（sequence）是一种特殊的集合。它是参数化的类型（像Java 中的泛型），在这里String 是参数。注意，Scala 使用方括号[…] 来指定参数类型，而Java 使用尖括号<…>。  
    注意  
    Scala 允许在方法名中使用尖括号，比如命名“小于”方法为<，这很常见。所以，为了避免二义性，Scala 使用了方括号来声明参数类型。它们不能被用于方法名。这就是为什么Scala 不允许像Java 那样的使用尖括号的习惯。  
    upper 方法的主体跟在等于号“=”后面。为什么是一个等于号？为什么不像Java 一样直接使用大括号{…} 呢？因为分号，函数返回类型，方法参数列表，甚至大括号都经常会被省略，使用等于号可以避免几种可能的二义性。使用等于号也提醒了我们，即使是函数，在Scala 里面也是值。这和Scala 对函数是编程的支持是一致的。我们会在《第8章，Scala 函数式编程》里讨论更多的细节。  
    函数的主体调用了strings 数组的map 方法，它接受一个字面函数（Function Literal）作为参数。字面函数也就是“匿名”函数。它们类似于其它语言中的Lambda 表达式，闭包，块，或者过程。在Java 里，你可能会在这里使用一个匿名内部类来实现一个接口（interface）定义的方法。  
    在这个例子里，我们传入这样的一个字面函数。  
    （s:String） => s.toUpperCase（） 它接受一个单独的名为s 的String 类型参数。 函数的主体在“箭头” => 的后面。它调用了s 的toUpperCase（） 方法。调用的结果会被函数返回。在Scala 中，函数的最后一个表达式就是返回值，尽管你也可以在其它地方使用return 语句。return 关键字在这里是可选的，而且很少被用到，除非在一段代码中间返回（比如在一个if 语句块中）。  
    注意  
    最后一个表达式的值是默认的返回值。不需要显式的return。  
    继续，map 把strings 里面的每一个String 传递给字面函数，从而用这些返回值创建了一个新的集合。  
    要运行这些代码，我们创建一个新的Upper 实例，然后把它赋值给一个名为up 的变量。和Java，C#，以及其它类似的语言一样，语法new Upper 创建了一个新的实例。变量up 被val 关键字定义为一个只读的值。  
    最后，我们对一个字符串列表调用upper 方法，然后用Console.println（…） 方法打印出来。这和Java 的System.out.println（…） 等效。  
    实际上，我们可以更加简化我们的代码。来看下面这一段简化版的脚本。  
    // code-examples/IntroducingScala/upper2-script.scala   object Upper {    def upper（strings: String\*） = strings.map（\_.toUpperCase（））  }  println（Upper.upper（"A", "First", "Scala", "Program"））  这段代码做了一模一样的事情，但是用了更少的字符。  
    在第一行，Upper 被定义为一个object，也就是单体模式。实际上我们是定义了一个class，但是Scala 运行时仅会创建Upper 的一个实例。（比如，你就不能写new Upper了。）Scala 的objects 被使用在其他语言需要“类级别”的成员的时候，比如Java 的statics （静态成员）。我们实际上并不需要更多的实例，所以单体模式也不错。  
    注意  
    Scala 为什么不支持statics？因为在Scala 中，所有的东西都是一个object，object 结构使得这样的政策保持了一致。Java 的static 方法和字段并不绑定到一个实际的实例。  
    注意这样的代码是完全线程安全的。我们没有定义任何会引起线程安全问题的变量。我们使用的API 方法也是线程安全的。所以，我们不需要多个实例。单体模式工作的很好。  
    在第二行的upper 方法的实现也变简单了。Scala 通常可以推断出方法的返回值（但是方法参数的类型就不行了），所以我们不用显式声明。而且，因为在方法的主体中只有一个表达式，我们也省略了括号，把整个方法的定义放到一行中。方法主体前面的等于号告诉编译器函数的主体从这里开始，就像我们看到的一样。  
    我们也在字面函数里利用一些简写。之前我们像这样写一个函数：  
    （s:String） => s.toUpperCase（） 我们可以简化成如下表达式：  
    \_.toUpperCase（） 因为map 接受一个参数，即一个函数，我们可以用 \_ 占位符来替代有名参数。也就是说，\_ 像是一个匿名变量，在调用 toUpperCase 之前每一个字符串都会被赋值给它。注意，String 类型是被推断出来的。将来我们会看到，Scala 还会在某些上下文中充当通配符。  
    你可以在一些更复杂的字面函数中使用这种简化的语法，就像我们将在《第3章 - 要点详解》中看到的那样。  
    在最后一行，我们使用了一个object 而不是一个class 来简化代码。我们只要在Upper object 上直接调用upper 方法，而不用new Upper 来创建一个新的实例。（注意，这样的语法看起来很像在Java 类中调用一个静态方法。  
    最后，Scala 自动导入了许多用以输入输出的方法，比如println，所以我们不用写成Console.println（）。我们只使用println 本身就可以了。（参见《第7章 - Scala Obejct 系统》的“预定义Object ”章节来获取更多有关自动导入类型和方法的信息。）

让我们来做最后一次重构；让我们把这段脚本变成一个编译好的命令行工具。  
    // code-examples/IntroducingScala/upper3.scala  object Upper {    def main（args: Array[String]） = {      args.map（\_.toUpperCase（））。foreach（printf（"%s ",\_））      println（""）    }  } 现在upper 方法被重命名为main。因为Upper 是一个object，这个main 方法就像Java 类里的static main 方法一样。这个Upper 程序的入口。  
    注意  
    在Scala，main 必须是一个object 的函数。（在Java，main 必须是一个类的静态方法。）命令行参数会作为一个字符串数组被传入应用程序，比如 args: Array[String]。  
    main 方法的第一行使用了和我们刚才产看过的map 方法一样的简写。  
    args.map（\_.toUpperCase（））… 调用map 会返回一个新的集合。我们用foreach 来遍历它。我们在传给foreach 的这个字面函数中再一次使用了一个 \_ 占位符。这样，集合的每一个字符串会被作为printf 的参数传入。  
    …foreach（printf（"%s ",\_）） 更清楚地说明一下，这两个“\_”是完全相互独立的。这个例子里的连锁方法（Method Chaining）和简写字面函数需要花一些时间来习惯，但是一旦你熟悉了它们，他们用最少的临时变量来产生可读性很高的代码。  
    main 的最后一行在输出中加入了一个换行。  
    在这次，你必须先用scalac 来把代码编译成JVM 可认的。class 文件。  
    scalac upper3.scala  
    你现在应该有一个名为Upper.class 的文件，就像你刚编译了一个Java 类一样。  
    注意  
    你可能已经注意到编译器并没有因为文件名为upper3.scala 而object 名为Upper 而抱怨。不像Java，这里文件名不用和公开域内的类型名字一致。（我们会在《第5章 - Scala 基础面向对象编程》的“可见性规则”章节来探索可见性规则。）实际上，和Java 不同，你可以在一个单独文件中有很多公开类型。此外，文件的地址也不用和包的声明一致。不过，如果你愿意，你可以依旧遵循Java 的规则。  
    现在，你可以传入任意多个字符串来执行这个命令了。比如：  
    scala -cp . Upper Hello World! -cp 选项会把当前目录加入到“类路径”的搜索中去。你会得到如下输出：  
    HELLO WORLD! 这样，我们已经满足了一本编程语言书籍必须以一个“hello world ”程序开始的基本要求。  
    初尝并发  
    被Scala 吸引有很多原因。 其中一个就是Scala 库的Actors API。它基于Erlang [Haller2007] 强大的Actors 并发模型建立。这里有一个例子来满足你的好奇心。  
    在Actor 并发模型[Agha1987] 中， 被称为执行者（Actor） 的独立软件实体不会互相之间共享状态信息。 相反， 它们通过交换消息来通信。 没有了共享易变状态的需要， 就更容易写出健壮的并发应用程序。  
    在这个例子里， 不同的图形的实例被发送到执行者（Actor ）来进行绘画和显示。 想象这样一个场景： 一个渲染集群在为动画生成场景。 在场景渲染完成之后， 场景中的元图形会被发送到一个执行者中由显示子系统处理。  
    我们从定义一系列的Shape （形状） 类开始。  
    // code-examples/IntroducingScala/shapes.scala  package shapes {    class Point（val x: Double, val y: Double） {      override def toString（） = "Point（" + x + "," + y + "）"    }    abstract class Shape（） {      def draw（）： Unit    }   class Circle（val center: Point, val radius: Double）     extends Shape {      def draw（） = println（"Circle.draw: " + this）      override def toString（） =          "Circle（" + center + "," + radius + "）"    }    class Rectangle（val lowerLeft: Point, val height: Double, val width: Double）          extends Shape {      def draw（） = println（"Rectangle.draw: " + this）      override def toString（） =        "Rectangle（" + lowerLeft + "," + height + "," + width + "）"    }    class Triangle（val point1: Point, val point2: Point, val point3: Point）          extends Shape {      def draw（） = println（"Triangle.draw: " + this）      override def toString（） =        "Triangle（" + point1 + "," + point2 + "," + point3 + "）"    }  } 类Shape 的继承结构在shapes 包（package）中定义。你可以用Java 的语法定义包，但是Scala 也支持类似于C# 的名称空间的语法，就是把整个声明都包含在大括号的域中，就像这里所做的。Java 风格的包声明语法并不经常用到，然而，它们都一样精简和可读。  
    类Point（点）表示了在一个平面上的二位点。注意类名字后面的参数列表。它们是构造函数的参数。在Scala 中，整个类的主体就是构造函数，所以你可以在类名字后面，类实体之前的主构造函数里列出所有参数。（在《第5章 - Scala 的基本面向对象编程》的“Scala 的构造函数”章节中，我们会看到如何定义辅助的构造函数。）因为我们在每一个参数声明前放置了val 关键字，它们会被自动地转换为有同样名字的只读的字段，并且伴有同样名字的公开读取方法。也就是说，当你初始化一个Point 的实例时，比如point， 你可以通过point.x 和point.y 来读取字段。如果你希望有可变的字段，那么使用var 关键字。我们会在《第2章 - 打更少的字，做更多的事》的“变量声明”章节来探索如何使用val 和var 关键字声明变量。  
    Point 类的主体定义了一个方法，类似于Java 的toString 方法的重写（或者C# 的ToString 方法）。主意，Scala 像C# 一样，在重写一个具体方法时需要显式的override 关键字。不过和C# 不一样的是，你不需要一个virtual （虚拟）关键字在原来的具体方法上。实际上，在Scala 中没有virtual 关键字。像之前一样，我们省略了toString 方法主体两边的大括号“｛…｝”，因为我们只有一个表达式。  
    Shape 是一个抽象类。Scala 中的抽象类和Java 以及C# 中的很像。我们不能实例化一个抽象类，即使它们的字段和方法都是具体的。  
    在这个例子里，Shape 声明了一个抽象的draw （绘制）方法。我们说它抽象是因为它没有方法主体。在方法上不用写abstract （抽象）关键字。Scala 中的抽象方法就像Java 和C# 中的一样。（参见《第6章 - Scala 高级面向对象编程》的“重写Classes 和Traits 的成员”章节获取更多细节。）  
    draw 方法返回Unit，这种类型和Java 这样的C 后继语言中的void 大体一致。（参见《第7章 - Scala Object 系统》的“Scala 类型组织”来获取更多细节。）  
    Circle （圆）被声明为Shape 的一个具体的子类。 它定义了draw 方法来简单地打印一条消息到控制台。Circle 也重写了toString。  
    Rectangle 也是Shape 得一个具体子类，定义了draw 方法，重写了toString。为了简单起见，我们假设它不会相对X 或Y 轴旋转。于是，我们所需要的就是一个点，左下角的点就可以，以及长方形的高度和宽度。  
    Triangle （三角形）遵循了同样的模式。它获取3个点作为它的构造函数参数。  
    在Circle，Rectangle 和Triangle 的所有draw 方法里都用到了this。和Java，C# 一样，this 是一个实例引用自己的方式。在这里的上下文中，this 在一个String 的链接表达式（使用加号）的右边，this.toString 被隐式地调用了。  
    注意  
    当然，在一个真正的程序中，你不会像这样实现一个域模型里的drawing 方法，因为实现会依赖于操作系统平台，绘图API 等细节。我们会在《第4章 - Traits》里看到一个更好地设计方式。  
    既然我们已经定义了我们的形状类型，让我们回过头来看Actors。我们定义了一个Actor 来接受消息（需要绘制的Shape）。  
    // code-examples/IntroducingScala/shapes-actor.scala  package shapes {    import scala.actors.\_    import scala.actors.Actor.\_    object ShapeDrawingActor extends Actor {      def act（） {        loop {          receive {            case s: Shape => s.draw（）            case "exit"   => println（"exiting…"）； exit            case x: Any   => println（"Error: Unknown message! " + x）          }        }      }    }  } Actor 被声明为shapes 包的一部分。接着，我们有两个import （导入）表达式。  
    第一个import 表达式导入了所有在scala.actors 包里的类型。在Scala 中，下划线\_ 的用法和Java 中的星号\* 的用法一致。  
    注意  
    因为\* 是方法名允许的合法字符，它不能在import 被用作通配符。所以，\_ 被保留来作为替代。  
    Actor 的所有方法和公开域内的字段会被导入。Actor 类型中没有静态导入类型，虽然Java 中会。不过，它们会被导入为一个object，名字一样为Actor。类和object 可以使用同样的名字，就像我们会在《第6章 - Scala 高级面向对象编程》的“伴随实体”章节中看到的那样。  
    我们的Actor 类定义，ShapeDrawingActor，是继承自Actor （类型，不是实体）的一个实体。它的act 方法被重写来执行Actor 的实际工作。因为act 是一个抽象方法，我们不需要显式地用override 关键字来重写。我们的Actor 会无限循环来等待进来的消息。  
    在每一次循环中，receive 方法会被调用。它会阻塞当前线程直到一个新的消息到来。为什么在receive 后面的代码被包含在大括号｛｝中而不是小括号（）呢？我们会在后面学到，有些情况下这样的替代是被允许的，而且十分有用（参见《第3章 - Scala 本质》）。现在，我们需要知道的是，在括号中的表达式组成了一个字面函数，并且传递给了receive。这个字面函数给消息做了一个模式匹配来决定它被如何处理。由于case 语句的存在，它看上去像Java 中的一个典型的switch 表达式，实际上它们的行为也很相像。  
    第一个case 给消息做了一个类型比较。（在代码中没有为消息实体做显式变量声明；它是被推断出来的。）如果消息是Shape 类型的，第一个case 会被满足。消息实体会被转换成Shape 并且赋值给变量s，然后s 的draw 方法会被调用。  
    如果消息不是一个Shape，第二个case 会被尝试。如果消息是字符串 exit ，Actor 会打印一条消息然后结束执行。Actors 通常需要一个优雅退出的方式。  
    最后一个case 处理所有其它任何类型的消息实例，作用和default （默认）case 一样。Actor 会报告一个错误然后丢弃这个消息。Any 是Scala 类型结构中所有类型的父类型，就像Java 和其他类型语言中的Object 根类型一样。所以，这个case 块会匹配任何类型的消息。模式匹配是头饥饿的怪兽，我们必须把这个case 块放在最后，这样它才不会把我们需要的消息也都吃掉！  
    回想一样我们在Shape 类里定义draw 为一个抽象方法，然后我们在具体的子类里实现它。所以，在第一个case 块中的代码执行了一个多态操作。  
    模式匹配 vs. 多态  
    模式匹配在函数式编程中扮演了中心角色， 就好像多态在面向对象编程中扮演着中心角色一样。函数式的模式匹配比绝大多数像Java 这样的命令式语言中的switch/case 语句更加重要和成熟。我们会在《第8章 - Scala 函数式编程》了解更多Scala 对于模式匹配支持的细节。在我们的这个例子里，我们可以开始看到，函数式模式匹配和面向对象多态调度的有力结合会给Scala 这样的混合范式语言带来巨大好处。  
    最后，这里有一段脚本来使用ShapeDrawingActor。  
    // code-examples/IntroducingScala/shapes-actor-script.scala  import shapes.\_  ShapeDrawingActor.start（）  ShapeDrawingActor ! new Circle（new Point（0.0,0.0）， 1.0）  ShapeDrawingActor ! new Rectangle（new Point（0.0,0.0）， 2, 5）  ShapeDrawingActor ! new Triangle（new Point（0.0,0.0），                                   new Point（1.0,0.0），                                   new Point（0.0,1.0））  ShapeDrawingActor ! 3.14159  ShapeDrawingActor ! "exit" 在shapes 包里的所有形状类会被导入。  
    ShapeDrawingActor 会被启动。默认情况下，它会运行在它自己的线程中（也有另外的选择，我们会在《第9章 - 使用Actor 的健壮的，可伸缩的并发编程》中讨论），等待消息。  
    有5个消息通过使用语法 actor ! message 被送到Actor。第一个消息发送了一个Circle 实例。Actor 会“画”出这个圆。第二个消息发送了Rectangle 消息。Actor 会“画”出这个长方形。第三个消息对一个三角形做了同样的事情。第四个消息发送了一个约等于Pi 的Double （双精度浮点数）值。这对于Actor 来说是一个未知消息，所以它只是打印了一个错误消息。最后一个消息发送了exit 字符串，它会导致Actor 退出。  
    要实验这个Actor 例子，从编译这两个源文件开始。你可以从O'Reilly 下载网站获取源代码（参见前言中获取代码示例的部分来取得更多细节信息），或者你也可以自己创建它们。  
    使用下面的命令来编译文件。  
    scalac shapes.scala shapes-actor.scala 虽然源文件的名字和位置并不和文件内容匹配，你会发现生成的class 文件被写入到一个shape 文件夹内，每一个类都会有一个class 文件对应。这些class 文件的名字和位置必须和JVM 的需求相吻合。  
    现在你可以运行这个脚本来看看Actor 的实际运行。  
    scala -cp . shapes-actor-script.scala 你应该可以看到如下输出。  
    Circle.draw: Circle（Point（0.0,0.0），1.0）  Rectangle.draw: Rectangle（Point（0.0,0.0），2.0,5.0）  Triangle.draw: Triangle（Point（0.0,0.0），Point（1.0,0.0），Point（0.0,1.0））  Error: Unknown message! 3.14159  exiting… 要知道更多关于Actor 的细节，参加《第9章 - 使用Actor 的强壮的，可伸缩的并发编程》。  
    概括  
    我们通过Scala 的示例来让你开始对Scala 有所了解，其中一个还给出了Scala Actors 库的强大并发编程体验。下面，我们会更深入Scala 语法，强调各种各样快速完成大量任务的“键盘金融”方式。