大数据应用与技术丛书

# Scala 和 Spark 大数据分析

## 函数式编程、数据流和机器学习

[德] 雷扎尔•卡里姆(Md. Rezaul Karim) 著
[美] 斯里达尔•阿拉(Sridhar Alla)
史跃东 译

清华大学出版社
北　京

北京市版权局著作权合同登记号　图字：01-2018-4396



图书在版编目(CIP)数据

Scala 和 Spark 大数据分析：函数式编程、数据流和机器学习 / (德)雷扎尔 • 卡里姆(Md. Rezaul Karim)，(美)斯里达尔 • 阿拉(Sridhar Alla) 著；史跃东 译. —北京：清华大学出版社，2020.5
(大数据应用与技术丛书)
书名原文：Scala and Spark for Big Data Analytics: Explore the Concepts of Functional Programming, Data Streaming, and Machine Learning
ISBN 978-7-302-55196-6

Ⅰ. ①S… Ⅱ. ①雷… ②斯… ③史… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字(2020)第 055695 号

责任编辑：王　军
装帧设计：孔祥峰
责任校对：成凤进
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　印　　张：39　　字　　数：996 千字
版　　次：2020 年 6 月第 1 版　　印　　次：2020 年 6 月第 1 次印刷
定　　价：158.00 元

产品编号：080800-01

# 译者序

和我此前翻译的几本书相比，这本书显然厚了许多，足足花了我一年多的时间来翻译和校对。

从某种程度讲，翻译就是一个再创作的过程，对于本书来说尤其如此。而想将一本专业类的书籍翻译到位，对于译者的要求也是极高的。这需要译者自身具有足够的技术能力，能够知晓原作所言；也需要译者有良好的英文阅读和理解功力，这样才能充分理解原作的本意；还需要译者具有扎实的中文功底，这样翻译出来的书才能让读者阅读的时候不至于磕磕绊绊。

回到本书，我之所以要翻译这本书，也是源于我对大数据生态圈的理解和判断。与传统的MR 计算框架相比，Spark 有着足够的性能和易编程方面的优势，并且 Spark 本身也正在形成自己的生态体系。而 Spark 的原生语言 Scala 将面向对象和函数式编程语言的优势融为一体，因此有着足够的理由让我看好。数据分析和机器学习，就更不必多言了。这都是当前 IT 领域最热门的技术方向。

正是基于上述考虑，我才决定翻译此书。当然，本书不是面向初学者的，要想发挥出本书的价值，你需要具备数据库、大数据、编程以及机器学习等方面的基础知识。换言之，如果想顺利地阅读本书，你可以是机器学习领域的一个新手，但你不能是一个 IT 领域的新手。

在本书的翻译过程中得到了诸多同事和好友的大力相助，在此一并谢过。尤其是我的妻子，在我翻译本书期间，正是她提供的始终如一的支持和理解，才让我的翻译工作能够如此持续不断地进行下去。

最后，谨以此翻译作品，献给我辛劳的父亲母亲，他们开明的态度和坚韧的精神，对我影响至深。

——史跃东

# 作者简介

**Md. Rezaul Karim** 是德国 Fraunhofer FIT 的研究学者，也是德国亚琛工业大学的博士学位研究生预科生。他拥有计算机科学的学士与硕士学位。在加盟 Fraunhofer FIT 之前，他曾作为研究员任职于爱尔兰的数据分析深入研究中心。更早之前，他还担任过三星电子公司全球研究中心的首席工程师；该研究中心分布于韩国、印度、越南、土耳其以及孟加拉。再早之前，他还在韩国庆熙大学的数据库实验室担任过助理研究员，在韩国的 BMTech21 公司担任过研发工程师，在孟加拉国的 i2 软件技术公司担任过软件工程师。

Karim 拥有超过 8 年的研发领域工作经验，并在如下算法和数据结构领域具有深厚的技术背景：C/C++、Java、Scala、R、Python、Docker、Mesos、Zeppelin、Hadoop 以及 MapReduce，并深入学习了如下技术：Spark、Kafka、DC/OS、DeepLearning4j 以及 H2O-Sparking Water。他的研究兴趣包括机器学习、深度学习、语义网络、关联数据(Linked Data)、大数据以及生物信息学。同时，他还是 Packt 出版社出版的以下两本书籍的作者：

- *Large-Scale Machine Learning with Spark*
- *Deep Learning with TensorFlow*

我非常感激我的父母，是他们一直鼓励我去不断追求新知识。也想感谢妻子 Saroar、儿子 Shadman，以及哥哥 Mamtaz 和姐姐 Josna，还有我的朋友们。因为他们总得长时间地忍受我关于本书内容的一些独白，还要鼓励我。另外，由于开源社区的令人赞叹的努力，以及 Apache Spark 和 Scala 相关的一些项目的卓越技术文档的存在，都使得本书的写作变得颇为容易。也要感谢 Packt 出版社的组稿、文稿以及技术编辑们(当然还有其他为本书做出贡献的出版社人员)，感谢他们真挚的沟通与协调。此外，若没有大量的研究人员和数据分析实践者们在出版物和各种演讲中分享自己的工作内容，公开自己的源代码，本书只怕也是无法面世的。

**Sridhar Alla** 是一位大数据专家，他曾帮助大大小小的诸多公司解决各种复杂的问题，例如数据仓库、数据治理、安全、实时数据处理、高频率的交易系统以及建立大规模的数据科学实践项目等。他也是敏捷技术的实践者，是一位获得认证的敏捷 DevOps 实践者和实施者。他在美国网域存储公司，以存储软件工程师的身份开始了自己的职业生涯。然后成为位于波士顿的 eIQNetworks 公司的 CTO，该公司是一家网络安全公司。在他的履历表中，还包括曾担任位于费城的 Comcast 公司的数据科学与工程总监。他是很多会议或者活动(如 Hadoop World、Spark 峰会等)的热心参与者，在多项技术上提供面授/在线培训。他在美国商标专利局(US PTO)也有多项专利技术，内容涉及大规模计算与分布式系统等。他还持有印度尼赫鲁科技大学计算机科学方向的学士学位。目前，他和妻子居住在新泽西州。

Alla 在 Scala、Java、C、C++、Python、R 以及 Go 语言上有超过 18 年的编程经验，他的技术研究范围也扩展到 Spark、Hadoop、Cassandra、HBase、MongoDB、Riak、Redis、Zeppelin、Mesos、Docker、Kafka、ElasticSearch、Solr、H2O、机器学习、文本分析、分布式计算以及高性能计算等领域。

我要感谢我贤惠的妻子 Rosie Sarkaria，在我写作本书的数个月中，她给了我无尽的爱与耐心，并给我写的内容进行了无数次的校订。我也想感谢父母 Ravi 和 Lakshmi Alla，他们也在一直支持我和鼓励我。也要感谢我的朋友们，尤其是 Abrar Hashmi 和 Christian Ludwig，他们不断地给我提供灵感并让我清晰地阐述书中的多个主题。如果没有神奇的 Apache 基金会，以及那些让 Spark 变得如此强大与优雅的大数据相关人员们，本书就无法付诸笔端了。我还要感谢 Packt 出版社的组稿、文稿以及技术编辑们(当然还有其他为本书做出贡献的出版社人员)，感谢他们真挚的沟通与协调。

# 审校者简介

**Andre Baianov** 是一位由经济学者转行而来的开发人员，他对数据科学有着极大的兴趣。他的学士论文是关于数据挖掘方面的，硕士论文是关于商业智能方面的。他从 2015 年开始从事 Scala 和 Spark 方面的工作。他现在是一名专业顾问，为国内和诸多国际客户提供服务，帮助这些客户建立反应式架构、机器学习框架以及函数式编程后台。

致妻子：在我们肤浅的不同之下，我们分享着同样的灵魂。

**Sumit Pal** 是 Apress 出版社出版的 *SQL on Big Data——Technology，Architecture and Innovations* 一书的作者。他在软件行业有超过 22 年的从业经历，并担任过不同职位，无论是初创公司还是大企业，他都在其中扮演过不同角色。

Pal 是一位大数据、数据可视化和数据科学领域的独立顾问，并且是建立端到端的、数据驱动的分析系统方面的软件架构师。在其 22 年的职业生涯中，他曾先后为微软(SQL Server 开发团队)、甲骨文(OLAP 开发团队)和 Verizon(大数据分析团队)等公司工作过。

目前，他正在为多个客户提供服务，在数据架构、大数据解决方案，以及如何使用 Spark、Scala、Java 和 Python 等语言进行开发方面，为客户提供咨询。

Pal 已经在如下大数据会议上发表演讲：数据峰会(纽约，2017 年 5 月)、大数据研讨会(波士顿，2017 年 5 月)、Apache Linux 基金会(加拿大温哥华，2017 年 5 月)和数据中心世界大会(拉斯维加斯，2016 年 3 月)。

# 前　言

随着数据量的持续膨胀，企业决策也变得日益复杂。因此，如果你还想使用传统的分析方法来洞察数据从而推动企业前进的话，那么，不断增长的数据将给你带来巨大障碍。现在，大数据涉及的领域太广泛了，它与各种框架之间存在千丝万缕的关系。以至于大数据的定义也与这些框架能够处理的范围产生了联系。无论你在检查来自百万访问者的点击流数据，从而优化在线广告的投放位置，还是过滤数十亿的事务数据，以便鉴定危险或欺诈信息，这些行为都需要高级的分析技术，例如机器学习和图运算，从而能在远超以往的大量数据中进行自动洞察与分析操作。

作为大数据处理、分析以及跨越学术界和工业界的数据科学等领域的事实上的标准，Apache Spark 提供了机器学习和图运算程序包，从而能让企业基于高可扩展性及集群化的计算机设备，轻松地处理诸多复杂问题。不仅如此，Spark 还允许你使用 Scala 语言来编写分布式程序，就像为 Spark 编写普通程序那样简单。Spark 为 ETL 数据传输带来了巨大的性能提升，也能让那些原来的 MapReduce 程序员们从 Hadoop 复杂的编程模型中部分解脱出来。

在本书中，我们将竭力为你带来基于 Spark 和 Scala 的最先进数据分析技术，包括机器学习、图运算、流处理以及 Spark SQL。当然，也包括 MLlib、ML、SQL、GraphX 以及其他程序库。

我们先从 Scala 开始，然后逐步进入 Spark 部分，最后将涵盖基于 Spark 和 Scala 的大数据处理的一些高级主题。在附录中，将扩展你的 Scala 知识，介绍 SparkR、PySpark、Apache Zeppelin 以及基于内存的 Alluxio 等。本书的内容并不需要你逐章完整阅读，你可以根据自己的兴趣，随意跳跃性翻阅感兴趣的章节。

祝你阅读愉快！

## 内容简介

第 1 章“Scala 简介”将基于 Scala 语言使用 Spark 的 API，从而教给你大数据处理技术。Spark 本身就是用 Scala 编写的，因此，我们很自然地以 Scala 的简介作为本书的开始。简介包括 Scala 的历史、设计目的，以及如何在 Windows、Linux 以及 macOS 上安装 Scala。此后，我们将探讨 Scala 的 Web 框架。再后对 Java 和 Scala 做对比分析。最后将研究 Scala 程序设计从而开始使用 Scala。

第 2 章“面向对象的 Scala”讲述面向对象的程序设计(OOP)范例，提供了一个全新的抽象层。简单来说，该章描述面向对象程序设计语言的一些强大之处：可发现性、模块性和可扩展性。尤其将讲述如何处理 Scala 中的变量、方法、类和对象。还讨论包、包对象、特征以及特征的线性化。当然，还有与 Java 的互操作性。

第 3 章“函数式编程概念”将列出 Scala 中函数式编程的基本概念。具体而言，我们将学习如下几个议题，Scala 为何是数据科学的兵工厂，为何学习 Spark 范例、纯函数以及高阶函数(Higher-Order Function，HOF)很重要。同时将展示在真实世界中使用 HOF 的用户案例。然后，我们将了解如何使用 Scala 的标准库函数，来在集合之外处理高阶函数中的异常。最后，将学习函数式 Scala 如何影响对象的可变性。

第 4 章“集合 API”将介绍一个会影响大部分 Scala 用户的特性——集合 API。该特性很强大且颇具弹性。我们将展示 Scala 集合 API 的能力，以及如何有序地使用它来处理不同的数据类型，并解决各种复杂问题。在该章中，我们将探讨 Scala 的集合 API、类型以及层级，还有一些性能方面的议题，与 Java 的互操作性，还有 Scala 的隐式转换。

第 5 章“狙击大数据——Spark 加入战团”将简要描述数据分析与大数据。将讨论大数据带来的挑战如何被分布式计算以及函数式编程所处理。我们将介绍谷歌的 MapReduce、Apache Hadoop 和 Apache Spark。我们也将了解到为何 Apache Spark 会首先被创建出来；面对大数据分析与处理的挑战，Apache Spark 又能带来怎样的价值。

第 6 章“开始使用 Spark——REPL 和 RDD”将介绍 Spark 的工作原理，然后介绍 RDD(Apache Spark 的基本抽象概念)，讲述它们是怎样的分布式集合，以及如何使用类似 Scala 的 API 进行操作。我们也将了解 Apache Spark 在部署方面的一些选项，以及如何以 Spark shell 方式在本地运行。我们也将深入学习 Apache Spark 的一些内部原理，例如 RDD 的含义、DAG 和 RDD 的血统机制、transformation 算子以及 action 算子。

第 7 章“特殊 RDD 操作”将关注 RDD 是如何被冗余并满足各种不同需求的，以及 RDD 如何提供新功能。不仅如此，我们还将了解 Spark 提供的其他有用对象，如广播变量和累加器(accumulator)。我们也将学习聚合技术 shuffle。

第 8 章“介绍一个小结构——Spark SQL”将讲解如何使用 Spark 分析结构化数据，Spark 是如何将结构化数据作为 RDD 的高阶抽象来处理的，以及 Spark SQL 的 API 是如何让查询结构化数据变得简单并足够健壮。还将介绍数据集(dataset)，讲述数据集、DataFrame 以及 RDD 之间的区别。也将讨论如何使用 DataFrame API，从而利用连接操作和窗口函数执行复杂的数据分析。

第 9 章“让我流起来，Scotty——Spark Streaming”将讲述如何使用 Spark Streaming，并利用 Spark API 来高效地处理流式数据。不仅如此，在该章中，还将介绍处理实时数据流的不同方法，列举一个真实的案例来演示来自 Twitter 的信息是如何被使用和处理的。我们也将看到与 Apache Kafka 的集成。还将看到结构化的流数据，它们将能为应用提供实时查询功能。

第 10 章“万物互联——GraphX”旨在使你了解到，真实世界中的许多问题都通过图运算进行建模并加以解决。我们将看到基于 Facebook 的例子，分析如何使用图论，其中包含 Apache Spark 的图运算程序库 GraphX、VertexRDD、EdgeRDD、图操作、aggregateMessages、triangleCount、Pregel API 以及用户案例(如 PageRank 算法等)。

第 11 章“掌握机器学习——Spark MLlib 和 ML”旨在提供统计机器学习的一些概念性介绍。我们将关注 Spark 的机器学习 API(称为 Spark MLlib 和 ML)。接下来将讨论如何使用决策树和随机森林算法解决分类问题，以及如何使用线性回归算法解决回归问题。你也将看到，在训练分类模型前，我们是如何在特征提取中使用 OneHotEncoder 和降维算法来获得好处的。在该章的最后，还将一步步地展示一个例子，来讲述怎样开发一个基于协同过滤的电影推荐系统。

第 12 章“贝叶斯与朴素贝叶斯”讲述大数据与机器学习。大数据与机器学习已经成为一个激进的组合，给研究领域(无论是学术领域还是工业领域)带来巨大影响。大数据给机器学习、数据分析工具以及算法都带来了巨大挑战，因为它们都需要发现真正的价值。但是，基于现有的这些海量数据集来预测未来从来都不是容易的事情。在该章中，我们将深入研究机器学习，并找出如何使用简单但强大的方法，来创建一个具备可扩展性的分类模型，同时将介绍相关的概念，例如多元分类、贝叶斯分类、朴素贝叶斯、决策树，以及朴素贝叶斯与决策树之间的对比分析。

第 13 章“使用 Spark MLlib 对数据进行聚类分析”将介绍常见的聚类算法，并通过大量实例，让你掌握这一机器学习领域中被广泛应用的技术。

第 14 章“使用 Spark ML 进行文本分析”将简要讲解如何使用 Spark ML 进行文本分析。文本分析是机器学习领域中一个很宽广的区域，它在很多用户场景下都极为有用，例如情感分析、聊天机器人、垃圾邮件检测、自然语言处理(NLP)以及其他很多场景。我们将学到如何使用 Spark 进行文本分析，该用例来自包含 10 000 个样例集合的 Twitter 数据，我们要对其进行文本分类。我们也将学习 LDA(隐含狄利克雷分布)，这是一项颇为流行的技术，它能从文档中生成对应的主题而不必知晓真实的文档内容，然后我们会基于这些 Twitter 数据实施文本分类，看一下具体的结果会是怎样的。

第 15 章“Spark 调优”深入探究 Apache Spark 的内部机制，我们会觉得在使用 Spark 时，感觉就像是在使用另一个 Scala 集合一样，但不要忘了，Spark 实际上是在一个集群中运行的。因此，该章将关注如何监控 Spark 任务，如何进行 Spark 配置，如何处理 Spark 应用开发过程中经常遇到的错误，还将介绍一些优化技术。

第 16 章“该聊聊集群了——在集群环境中部署 Spark”将研究 Spark 及其底层架构是如何在集群中工作的。我们将了解集群中的 Spark 架构、Spark 生态系统以及集群管理等内容。当然，还有如何在独立服务器模式、Mesos、YARN 以及 AWS 集群上部署 Spark。我们也将探讨如何在一个基于云的 AWS 集群上部署应用。

第 17 章“Spark 测试与调试”将解释分布式应用的测试难度。我们将看到一些处理方法。我们也将看到如何在分布式环境中进行测试，以及如何测试和调试 Spark 应用。

第 18 章“PySpark 与 SparkR”将涵盖其他两个常用的 API。可使用 Python 或 R 语言来编写 Spark 代码，也就是 PySpark 和 SparkR。具体地说，将介绍如何使用 PySpark，以及如何与 DataFrame API 和 UDF 进行交互，然后我们就可以使用 PySpark 来执行一些数据分析了。然后将介绍如何使用 SparkR，也将介绍如何使用 SparkR 进行数据处理与操作，以及如何与 RDD 和 DataFrame 协同工作。最后，一些数据可视化工作也可以使用 SparkR。

第 19 章“高级机器学习最佳实践”从理论和实践两个方面探讨 Spark 机器学习的一些高级议题。我们将看到如何使用网格搜索、交叉检验以及超参调整等方法对机器学习模型进行调优，以获取更好的性能。接下来将研究如何使用 ALS 来开发一个可扩展的推荐系统。这里，ALS 是一个基于模型的推荐系统算法。最后将展示一个主题建模应用，作为文本聚类技术的一个示例。

附录 A“使用 Alluxio 加速 Spark”将展示如何使用 Alluxio 结合 Spark 来加快处理速度。Alluxio 是一个开源的内存存储系统，它对于很多跨平台的应用都很有用，能提升这些应用的处理速度。我们将研究使用 Alluxio 的可能性，以及集成 Alluxio 是如何在我们每次运行 Spark 任务时，不需要缓存数据就能提供更好性能的。

附录 B“利用 Apache Zeppelin 进行交互式数据分析”从数据科学的角度出发，讲述交互式可视化数据分析的重要性。Apache Zeppelin 是一个基于 Web 的记事本，可用于交互式和大规模数据分析，它可以使用多后端和解释器。该章将研究如何使用 Apache Zeppelin，并将 Spark 作为其后端解释器，来进行大规模数据分析。

# 学习本书时所需的准备工作

本书中的所有例子都基于 Ubuntu Linux 64 位版本，使用 Python 2.7 和 Python 3.5 实现，其中使用的 TensorFlow 函数库版本为 1.0.1。本书中展示的源代码是兼容 Python 2.7 的；兼容 Python 3.5+ 的源代码可从 Packt 出版社的资料库中自行下载。你也需要如下 Python 模块(最新版本尤佳)：

- Spark 2.0.0 或更高版本
- Hadoop 2.7 或更高版本
- Java(JDK 和 JRE)1.7+/1.8+
- Scala 2.11.x 或更高版本
- Python 2.7+/3.4+
- R 3.1+以及 RStudio 1.0.143 或更高版本
- Eclipse Mars、Oxygen 或 Luna(最新版本)
- Maven Eclipse plugin (2.9 或更高版本)
- Maven compiler plugin for Eclipse (2.3.2 或更高版本)
- Maven assembly plugin for Eclipse (2.4.1 或更高版本)

**操作系统**：优先推荐 Linux 发行版(包括 Debian、Ubuntu、Fedora、RHEL 以及 CentOS)，更精确地说，对于 Ubuntu，推荐版本为 64 位 14.04(LTS)或更新版本，推荐使用 VMWare player 12 或 Virtual box。可在 Windows(XP/7/8/10)或 Mac OS X(10.4.7+)上运行 Spark 任务。

**硬件配置**：处理器核心为 i3、i5(推荐)或 i7(可获得最佳效果)。但是，多核心处理器可以提供更快的数据处理和更好的可扩展性。对于独立服务器模式，需要至少 8～16GB 的内存(推荐配置)；对于集群而言，单一的 VM 或更多的 VM 则至少需要 32GB 内存。你也需要有足够的存储来执行工作量大的任务(具体取决于要处理的数据集的大小)，建议使用的存储至少有 50GB 的空闲空间(用于支持独立服务器模式以及 SQL 仓库)。

# 本书读者对象

所有期望使用强大的 Spark 来进行数据分析的人们，都会意识到本书的内容极具价值。你不需要具备 Spark 或 Scala 的相关知识，当然，如果你此前就有编程经验的话(尤其是掌握其他 JVM 编程语言)，在学习本书的相关概念时，你掌握知识的速度就会比较快。在过去数年间，Scala 语言正逐步为人们所接纳采用，一直呈现出稳定的上升态势，在数据科学与分析领域尤其如此。与 Scala 齐头并进的是 Apache Spark，它由 Scala 语言编写而成，并且在数据分析领域得到了极广泛

的应用。本书将帮助你掌握这两种工具，从而让你在大数据处理领域大显身手。

## 约定

**注意：**
警示或重要提示。

**小技巧：**
技巧和诀窍。

## 读者反馈

我们一直希望能够收到读者的反馈意见。让我们知道你对这本书的看法——你喜欢的部分或不喜欢的部分——这对我们非常重要。因为它可以帮助我们真正编写出可以充分使用的书籍。如果想向出版社发送反馈，可以直接发送电子邮件至 feedback@packtpub.com，并在邮件中注明本书的书名即可。

## 下载样例代码

可访问 http://www.tupwk.com.cn/downpage/网站，输入本书 ISBN 或中文书名下载本书的样例代码。

一旦下载完本书的代码文件，就可以使用下列文件的最新版本来解压缩：

- 对于 Windows 平台而言，可使用 WinRAR/7-Zip。
- 对于 macOS 而言，可使用 Zipeg/iZip/UnRarX。
- 对于 Linux 而言，可使用 7-Zip/PeaZip。

另外，也可扫本书封底二维码下载相关资料。

### 勘误

尽管我们已经尽了最大努力来确保本书中内容的正确性，但错误依然无法避免。因此，如果你在阅读本书的过程中发现了错误——无论是代码错误还是文本错误，都请与我们联系，我们对此将深表感激。

### 侵权

互联网上版权材料的盗版问题一直存在。我们非常重视知识产权的保护。因此，如果你在互联网上发现了本书任何形式的非法副本，也请立即与我们联系。

## 问题

如果你在阅读本书的过程中发现了任何问题，请与我们联系。

# 目　录

# 第 1 章

# Scala 简介

“我是 Scala，我是一个可扩展的、函数式的、面向对象的编程语言。我可以和你一起成长，也可以和我一起玩耍，比如以输入一行表达式然后就能立即得到结果的方式。”

——Scala 自述

在最近几年间，Scala 语言正处于稳步上升期，它逐渐被大的开发者和相关从业者所采纳。尤其是在数据科学和分析领域。另外，由 Scala 语言所编写的 Apache Spark 则成为快速而通用的大数据处理引擎。Spark 的成功可归功于多个方面：易用的 API 接口、清晰的编程模型，性能优势等。因此，很自然，Spark 也就为 Scala 提供了更多支持；与 Python 或 Java 相比，Scala 可用的 API 更多一些。并且，在一些新的 API 出现之后，首先是支持 Scala，而后才是 Java、Python 和 R 语言。

现在，在使用 Spark 和 Scala 开始数据处理编程工作前，我们先来熟悉一下 Scala 的函数式程序设计概念、面向对象特性，以及详细了解 Scala 的集合 API(第一部分)。作为一个开始，我们将在本章简要介绍 Scala 语言。本章内容将覆盖 Scala 的历史及其设计目标等一些基本方面。然后将介绍如何在不同平台(包括 Windows、Linux 和 macOS)上安装 Scala。这样，就可以在自己喜欢的编辑器或 IDE 上进行数据分析编程活动了。本章稍后也将对 Java 和 Scala 进行对比分析。本章最后将通过一些例子来带你深入了解 Scala 程序设计。

作为概括，本章将涵盖如下主题：

- Scala 的历史与设计目标
- 平台与编辑器
- 安装与创建 Scala
- Scala：可扩展的编程语言
- 面向 Java 编程人员的 Scala
- 面向初学者的 Scala
- 本章小结

# 1.1　Scala 的历史与设计目标

Scala 是一门通用的编程语言，它支持函数式编程，同时也是一个强静态类型的语言系统。Scala 的源代码可被编译成 Java 的字节码，因此生成的可执行代码能在 JVM 上运行。

Martin Odersky 于 2001 年在瑞士洛桑联邦理工学院(EPFL)开始了 Scala 语言的设计。该语言基于他当时使用的 Funnel 语言，并进行了扩展。Funnel 也是一种程序设计语言，使用了函数式程序设计以及 Petri 网。Scala 语言的第一个版本于 2004 年发布，但只支持 Java 平台。稍后，在 2004 年 6 月，.NET 框架面世了。

Scala 语言很快就变得流行起来，并被广泛采用，因为它不仅支持面向对象的程序设计范例，还拥抱了函数式程序设计的概念。此外，尽管与 Java 相比，Scala 中的符号算子(symbolic operator)很难读懂，但大多数 Scala 的开发人员还是认为 Scala 更简洁更易读——Java 则显得过于繁杂。

与其他程序设计语言一样，Scala 语言也是为了某个特定的目标而建立并发展起来的。现在的问题是，Scala 为什么会被设计出来？它当时是为了解决什么问题？为了回答这些问题，Odersky 在他的博客中写道：

> “Scala 的设计工作，源于我们想开发出一种用于更好地支持组件软件(component software)的语言。我们想在 Scala 语言中验证两个假设。第一个假设是用于组件软件的编程语言具备可扩展性，也就是说，无论该语言用来表述小对象或大对象，其概念都应该是相同的。因此，我们将注意力放在了抽象、组成以及分解等实现机制上。我们并未给 Scala 语言添加一大堆原语(primitive)，因为这些东西可能在某些扩展级别上很有用，但在其他扩展级别上就未必了。第二个假设是为组件软件提供的这种可扩展性支持能由编程语言提供，该编程语言应该是面向对象和函数式编程的统一和归纳。对于静态类型的语言来说，Scala 语言则是一个实例，现在这两种范例之间的差别已经很大了。”

无论如何，现在 Scala 语言中也开始提供模式匹配和高阶函数等内容。这不是用来填平函数式编程与面向对象编程之间的鸿沟，而是因为这些本来就是函数式编程中的典型特性。因此，Scala 语言就有了一些令人惊奇的强大的模式匹配特性，它们是一种基于角色(actor-based)的并发处理框架，还支持一阶或高阶函数。总的来说，其名称 Scala 是可扩展语言(**Sca**lable **la**nguage)的一个合成词，预示了该语言被设计为会随着用户需求的变化而不断增长。

# 1.2　平台与编辑器

Scala 可在 JVM 上运行，这就使其成为 Java 开发人员的一个好选择，开发人员可在代码中添加一些函数式编程的风格。在编写 Scala 程序时，有足够多的编辑器供选择。可以花费一些时间，对可用的编辑器进行比较排序。如果你已经习惯了某个 IDE，那么再使用其他编辑器的话，可能会比较痛苦。所以，你要花时间好好地挑选一下编辑器。如下是一些可供挑选的编辑器：

- Scala IDE
- Scala plugin for Eclipse
- IntelliJ IDEA

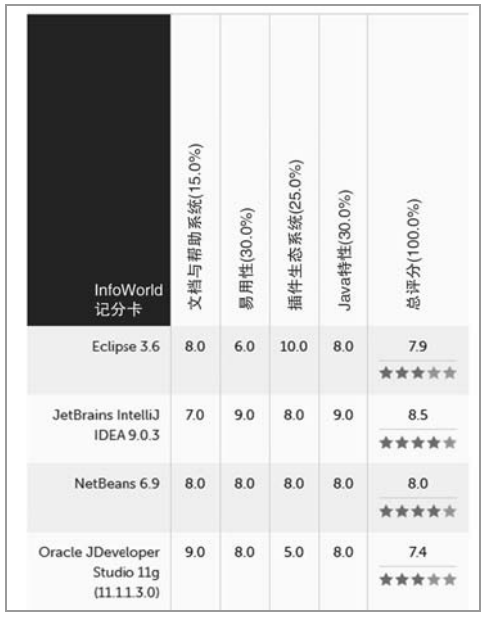
- Emacs
- VIM

使用 Eclipse 编写 Scala 程序有一些很明显的优势，可以使用相当数量的测试版插件。Eclipse 提供了一些非常优秀的特性，例如本地、远程以及高级别的调试功能，包括语义上的高亮显示，以及 Scala 中的代码辅助完成等。可使用 Eclipse 像编写 Java 代码一样编写 Scala 程序。但我也推荐你使用 Scala IDE(http://scala-ide.org/)——它是一个基于 Eclipse 的完全成熟的 Scala 编辑器，并且定制了一些很有趣的特性(如 Scala 工作单、ScalaTest 支持、Scala 重构等)。

在我看来，第二个最好用的编辑器是 IntelliJ IDEA。其第一个版本于 2001 年发布，是第一个集成了高级代码导航和重构功能的 Java IDE。由于 InfoWorld 的支持(可查看网站 http://www.infoworld.com/article/2683534/development-environments/infoworld-review--top-java-programming-tools.html)，使得在 4 个最受欢迎的 Java 编程 IDE 工具(Eclipse、IntelliJ IDEA、NetBeans 和 JDeveloper)中，IntelliJ 获得了最高分 8.5 分(满分 10 分)。

具体评分情况如图 1.1 所示。

从图 1.1 来看，你可能也会对使用其他 IDE(如 NetBeans 或 Jdeveloper)感兴趣。从根本上讲，究竟哪个编辑器更好用，是开发人员之间永恒的争论话题。这意味着，最终使用哪个 IDE，完全由你自己来定。



| InfoWorld 记分卡 | 文档与帮助系统(15.0%) | 易用性(30.0%) | 插件生态系统(25.0%) | Java特性(30.0%) | 总评分(100.0%) |
|---|---|---|---|---|---|
| Eclipse 3.6 | 8.0 | 6.0 | 10.0 | 8.0 | 7.9 ★★★★☆ |
| JetBrains IntelliJ IDEA 9.0.3 | 7.0 | 9.0 | 8.0 | 9.0 | 8.5 ★★★★★ |
| NetBeans 6.9 | 8.0 | 8.0 | 8.0 | 8.0 | 8.0 ★★★★☆ |
| Oracle JDeveloper Studio 11g (11.1.1.3.0) | 9.0 | 8.0 | 5.0 | 8.0 | 7.4 ★★★★☆ |

图 1.1　最适合 Scala/Java 开发人员的 IDE

## 1.3　安装与创建 Scala

由于 Scala 要使用 JVM，因此需要确保你的机器上已经安装了 Java。如果没有，可以参考后面的章节，其中展示了如何在 Ubuntu 上安装 Java。在这一节中，首先，将展示如何在 Ubuntu 上安装 Java 8。然后，我们来看看如何在 Windows、macOS 和 Linux 上安装 Scala。

### 1.3.1　安装 Java

为简单起见，在此仅展示如何在 Ubuntu 14.04 LTS 64 位机器上安装 Java 8。至于如何在 Windows 或 macOS 上安装 Java，你最好还是花时间使用 Google 搜索一番吧。对于 Windows 用户来说，可以访问 https://java.com/en/download/help/windows_manual_download.xml 来了解更多内容。

现在，让我们在 Ubuntu 上使用分步命令和说明来安装 Java 8。首先检查 Java 是否已经被安装：

```
$ java -version
```

如果返回消息 the program java cannot be found in the following packages，则表明尚未安装 Java。可执行如下命令进行安装：

```
$ sudo apt-get install default-jre
```

该命令将安装 JRE。但如果不想使用 JRE，也可以使用 JDK，它通常在 Apache Ant、Apache Maven、Eclipse 以及 IntelliJ IDEA 上编译 Java 应用时会用到。

Oracle JDK 是官方的 JDK，但 Oracle 已不再将其提供为 Ubuntu 的默认安装选项。你仍然可以使用 apt-get 命令来安装它。要安装任意版本的 JDK，需要先执行如下命令：

```
$ sudo apt-get install python-software-properties
$ sudo apt-get update
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
```

然后，基于想安装的版本，执行如下命令：

```
$ sudo apt-get install oracle-java8-installer
```

安装完成后，不要忘记设置 JAVA_HOME 环境变量。只需要使用如下命令即可(为简单起见，我们假设 Java 安装在/usr/lib/jvm/java-8-oracle 目录下)：

```
$ echo "export JAVA_HOME=/usr/lib/jvm/java-8-oracle" >> ~/.bashrc
$ echo "export PATH=$PATH:$JAVA_HOME/bin" >> ~/.bashrc
$ source ~/.bashrc
```

现在，让我们使用以下方式查看一下 JAVA_HOME：

```
$ echo $JAVA_HOME
```

你在终端上应能看到如下结果：

```
/usr/lib/jvm/java-8-oracle
```

现在，让我们使用如下命令来确认 Java 已经被成功安装(这里，你可能会看到最新版本)：

```
$ java -version
```

你将得到如下输出：

```
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

现在你已经成功地在机器上安装了 Java，一旦安装好，就可以准备写 Scala 代码了。我们将在接下来的章节中完成该内容。

### 1.3.2 Windows

这一部分将关注如何在 Windows 7 的电脑上安装 Scala。但是在最后，无论你使用的 Windows 是什么版本，其实都没什么关系。

(1) 第一步，需要去官网下载 Scala 的安装压缩包。可在 https://www.Scala-lang.org/download/all.html 上找到它。在该页面的其他资源部分下面，可看到你能够安装的 Scala 归档文件。这里选择下载 Scala 2.11.8 安装包，如图 1.2 所示。

| Archive | System | Size |
|---|---|---|
| scala-2.11.8.tgz | Mac OS X, Unix, Cygwin | 27.35M |
| scala-2.11.8.msi | Windows (msi installer) | 109.35M |
| scala-2.11.8.zip | Windows | 27.40M |
| scala-2.11.8.deb | Debian | 76.02M |
| scala-2.11.8.rpm | RPM package | 108.16M |
| scala-docs-2.11.8.txz | API docs | 46.00M |
| scala-docs-2.11.8.zip | API docs | 84.21M |
| scala-sources-2.11.8.tar.gz | Sources | |

图 1.2　在 Windows 上安装 Scala

(2) 下载完成后，对文件进行解压操作，并将其放置到你想要的目录下。可将该文件命名为 Scala，这样以后你查找起来也会比较方便。最后，需要在电脑上为 Scala 设置一个名为 PATH 的全局可见的变量。因此，需要导航到“计算机”|“属性”，如图 1.3 所示。


图 1.3　Windows 上的“环境变量”标签

(3) 从这里设置环境变量，获取 Scala 的 bin 目录的位置，将其扩展到 PATH 环境变量的后面。然后应用修改并单击“确定”，如图 1.4 所示。

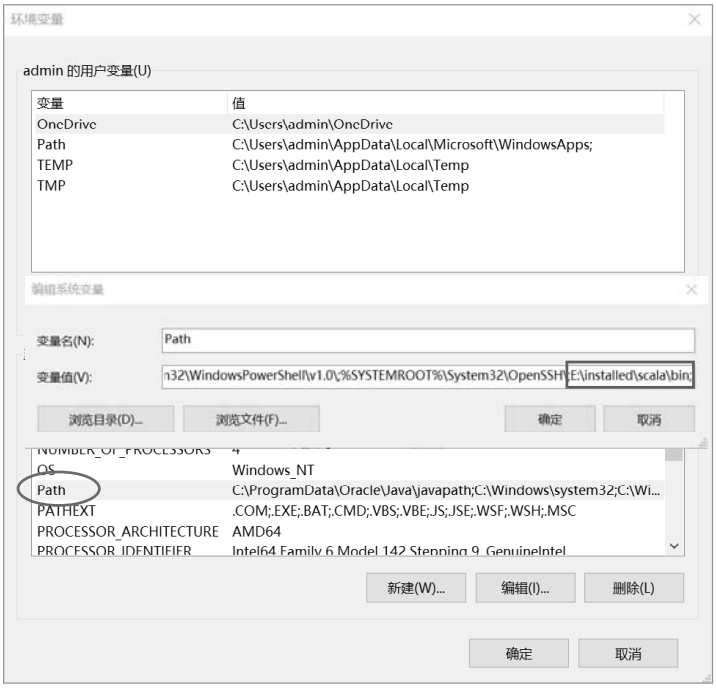



图 1.4　为 Scala 添加环境变量

(4) 现在，你已在 Windows 上做好了安装准备。打开 cmd 命令行，然后输入 scala。如果你在前面的安装和配置都是成功的，就能在屏幕上看到类似于图 1.5 的输出。

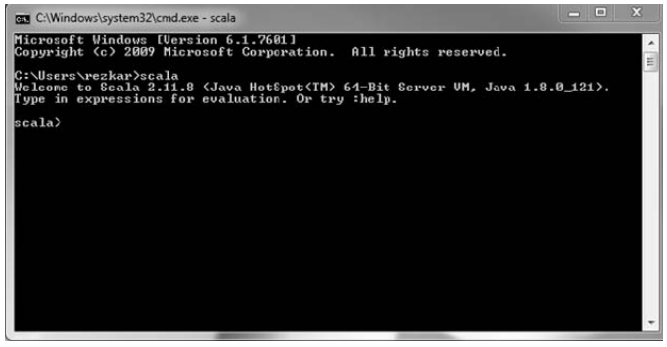



图 1.5　从 Scala shell 访问 Scala

## 1.3.3　macOS

现在可以在你的 macOS 上安装 Scala 了。在 macOS 上安装 Scala 有很多种方式。在这里，我们打算为你介绍其中的两种。

### 1. 使用 Homebrew 安装工具

(1) 首先，检查系统是否安装了 Xcode，因为这一步需要这一组件。可以从苹果公司的 Apple Store 上免费下载并安装。

(2) 接下来，需要在终端上执行如下命令来从内部安装 Homebrew。

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

**注意：**

上述安装命令可能会被 Homebrew 的工作人员随时修改。如果该命令无法正常工作，可以通过 Homebrew 的网站(http://brew.sh/)获取最新的安装“咒语”。

(3) 现在，可在终端上执行 brew install scala 命令来安装 Scala。

(4) 最后，可在终端上简单地输入一些 Scala 代码了(第二行)，之后就可以看到如图 1.6 所示的内容。

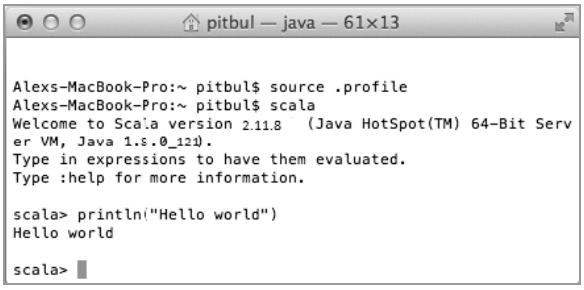



图 1.6　macOS 上的 Scala shell 窗口

### 2. 手工安装

在手工安装 Scala 之前，可在 http://www.Scala-lang.org/download/网站上选择你喜欢的 Scala 版本，并下载对应的.tgz 文件。在下载完你喜欢的 Scala 版本后，可按如下方式解压：

```
$ tar xvf scala-2.11.8.tgz
```

然后，以如下方式将其移到/usr/local/share 目录下：

```
$ sudo mv scala-2.11.8 /usr/local/share
```

现在，为保证安装结果能够持久，请执行如下命令：

```
$ echo "export SCALA_HOME=/usr/local/share/scala-2.11.8" >> ~/.bash_profile
$ echo "export PATH=$PATH: $SCALA_HOME/bin" >> ~/.bash_profile
```

这样就行了。现在，介绍一下在 Linux 发行版(例如 Ubuntu)上如何安装 Scala。

### 3. Linux

在这一章节中，我们将为你展示在 Linux 发行版 Ubuntu 上安装 Scala 的流程。在开始之前，让我们先确保 Scala 能够被正确安装。可以使用如下命令执行检查：

```
$ scala -version
```

如果你的系统上已经安装了 Scala，你将在终端上看到如下消息：

```
Scala code runner version 2.11.8 --Copyright 2002-2016, LAMP/EPFL
```

要注意这一点，在写作此安装过程时，我们使用的是当时最新的 Scala 版本。也就是 2.11.8。

如果你还没有在系统上安装 Scala，则在进行下一步之前要确保先安装完毕。可从 Scala 的官网下载最新版本(为清楚起见，可参考图 1.2)。为简便起见，我们下载 Scala 2.11.8，如下：

```
$ cd Downloads/
$ wget https://downloads.lightbend.com/scala/2.11.8/scala-2.11.8.tgz
```

等到下载完毕，就可在下载的文件夹中找到 Scala 的 tar 文件了。

**注意：**

用户应该使用如下命令进入 Downloads 目录：$ cd /Downloads/。当然，基于你的系统，选择的语言可能不同，Downloads 文件夹的名称也可能有所不同。

为将 Scala tar 文件从它所在的位置中抽取出来，可执行如下命令：

```
$ tar -xvzf scala-2.11.8.tgz
```

现在，通过执行如下命令将 Scala 移动到用户选定的位置(如/usr/local/share)，或者是手动移动：

```
$ sudo mv scala-2.11.8 /usr/local/share/
```

使用如下命令移动到用户的主目录：

```
$ cd ~
```

然后，使用如下命令设置 Scala 的主目录：

```
$ echo "export SCALA_HOME=/usr/local/share/scala-2.11.8" >> ~/.bashrc
$ echo "export PATH=$PATH:$SCALA_HOME/bin" >> ~/.bashrc
```

接下来，为让上述修改在会话中变得持久，可执行如下命令：

```
$ source ~/.bashrc
```

当安装完成后，你最好执行如下命令来检查安装结果：

```
$ scala -version
```

如果 Scala 在你的系统上已经配置成功，就可在终端上看到如下信息：

```
Scala code runner version 2.11.8 --Copyright 2002-2016, LAMP/EPFL
```

现在，让我们在终端上输入 scala 命令来进入 Scala shell。如图 1.7 所示。

```
asif@ubuntu:~$ cd /usr/local/share/
asif@ubuntu:/usr/local/share$ ls
ca-certificates  fonts  man  scala-2.11.8  sgml  texmf  xml
asif@ubuntu:/usr/local/share$ cd ~
asif@ubuntu:~$ echo "export SCALA_HOME=/usr/local/share/scala-2.11.8" >> ~/.bash
rc
asif@ubuntu:~$ echo "export PATH=$PATH:$SCALA_HOME/bin" >> ~/.bashrc
asif@ubuntu:~$ source  ~/.bashrc
asif@ubuntu:~$ scala
Welcome to Scala version 2.9.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_12
1).
Type in expressions to have them evaluated.
Type :help for more information.
```

图 1.7　Linux 上的 Scala shell(Ubuntu 发行版)

最后，也可以使用 apt-get 命令来安装 Scala，如下所示：

```
$ sudo apt-get install scala
```

该命令会下载最新版的 Scala(这里是 2.12.x)。但 Spark 尚未支持 Scala 2.12 版本(至少是在我们写作本章时)。因此，我们推荐使用前面的手动安装方式。

# 1.4　Scala：可扩展的编程语言

Scala 的代码从规模上可以很好地扩展到大型程序。对于其他编程语言可能需要编写数十行代码才能实现的功能，在 Scala 中能简便地实现；你能以简洁有效的方式获得表达编程的一般模式和概念的能力。本节将介绍 Odersky(Martin Odersky，Scala 语言之父)为我们创造的这些令人激动的特性。

## 1.4.1　Scala 是面向对象的

Scala 是面向对象编程语言的一个非常好的例子。为给你的对象定义类型或行为，需要使用类和特质的概念，这些内容将在第 2 章中展开讲述。Scala 并不支持直接多态集成，但是要实现这种架构，也可以使用 Scala 的子类扩展，或者是混合基成分(mixing-based composition)。这些内容也将在后续章节中讲述。

## 1.4.2　Scala 是函数式的

函数式编程通常将函数视为一等公民。在 Scala 中，则是通过语法糖(syntactic sugar)和扩展其特质的对象(例如 Function2)来实现的。但这是在 Scala 中实现函数式编程的方法。此外，Scala 也设计了一种简便的方式来定义匿名函数。并且 Scala 支持高阶函数，也允许函数嵌套。后面将深入介绍这些概念的语法。

另外，Scala 也能帮助你以不可变的方式进行编码，并且基于此，可以轻松地使用同步与并发机制来实现并行处理。

## 1.4.3　Scala 是静态类型的

不同于其他静态类型语言，例如 Pascal、Rust 等，Scala 并不期望你提供冗余的类型信息。在绝大部分情况下，你不必指定类型。更重要的是，你甚至不用再重复它们。

**注意：**

如果该语言的类型在编译时就是知道的，则该编程语言被称为是静态类型的。这也意味着，作为一名编程人员，需要确定每个变量的类型。Scala、Java、C、OCaml、Haskell 和 C++等都是静态类型语言。另一方面，Perl、Ruby、Python 等则是动态类型语言；这些语言中的类型，并不与变量或字段相关，而与运行时的值相关。

Scala 的静态类型天性，确保了所有种类的检查都是被编译器完成的。这是 Scala 的一个极其强大的特性，因为它可以帮助你及早发现/捕获大多数 bug 或错误。

### 1.4.4　在 JVM 上运行 Scala

与 Java 一样，Scala 代码也可被编译成字节码并在 JVM 上简单执行。这意味着 Scala 和 Java 的运行时平台是一样的。由于它们都是以字节码作为编译的输出结果，因此可以轻松地从 Java 转向 Scala，也可以轻松地将二者集成起来，甚至在你的 Android 应用中使用 Scala 来添加一个函数。

**小技巧：**
在 Scala 程序中使用 Java 代码相当容易，但反过来就很难了。这主要是因为 Scala 的语法糖特性。

另外，和使用 javac 命令将 Java 代码编译成字节码一样，Scala 也有一条 scalas 命令，它可将 Scala 代码编译成字节码。

### 1.4.5　Scala 可以执行 Java 代码

正如在前面提到的，Scala 也可以用来执行 Java 代码。它不仅可安装 Java 代码，甚至也允许你使用来自 Java SDK 的所有类，即便是你在 Scala 环境中已经有了自己的预定义类、项目或包。

### 1.4.6　Scala 可以完成并发与同步处理

对于 Scala 来说，你能以简洁有效的方式获得表达编程的一般模式和概念的能力。另外，Scala 也能帮助你以不可变的方式进行编码，并且基于此，可轻松地使用同步与并发机制来实现并行处理。

## 1.5　面向 Java 编程人员的 Scala

Scala 有着与 Java 完全不同的一些特性。本节将讨论其中一部分特性。对于那些具备 Java 编程背景，或者是对基本的 Java 语法有了解的编程人员而言，这一部分的内容将非常有用。

### 1.5.1　一切类型都是对象

Scala 中的每一个值看起来都像一个对象。这句话意味着一切看起来都像对象。但其中一部分并非真正的对象，你将在接下来的章节中看到对这些内容的解释。例如，在 Scala 中，引用类型和原生类型的区别依然存在，但大部分区别都已经被隐藏起来了；在 Scala 中，字符串已经被隐式地转换为字符集合，而在 Java 中却非如此！

## 1.5.2　类型推导

如果你不熟悉这个术语，这没什么，但是在编译时会出现类型推导(deduction of types)。等一下，难道这不是指动态类型吗？不是。注意这里我所说的类型推导，这与动态类型语言的处理完全不同，另一件事情是，这一推导是在编译时而非运行时完成。很多编程语言都内置了这一功能，但具体实现则完全不同。一开始，这可能造成疑惑，但当看到代码示例时就会清晰起来。让我们跳到 Scala REPL 来分析一些示例吧。

## 1.5.3　Scala REPL

Scala REPL 是一个非常强大的特性，它让我们在 Scala shell 中编写 Scala 代码变得直接又简单。REPL 指的是 Read-Eval-Print-Loop(读取-求值-输出-循环)，也被称为交互式解释器。这意味着对于一段程序来说，REPL 可以：

(1) 读取你输入的表达式。

(2) 使用 Scala 编译器求出在第(1)步输入的表达式的值。

(3) 打印出在第(2)步求出的结果。

(4) 以循环方式等待你输入更多表达式。

从图 1.8 中可以看到，这里并没有什么魔法，在编译时，变量会被自动推导出最合适的类型。如果尝试声明：

```
i:Int = "hello"
```

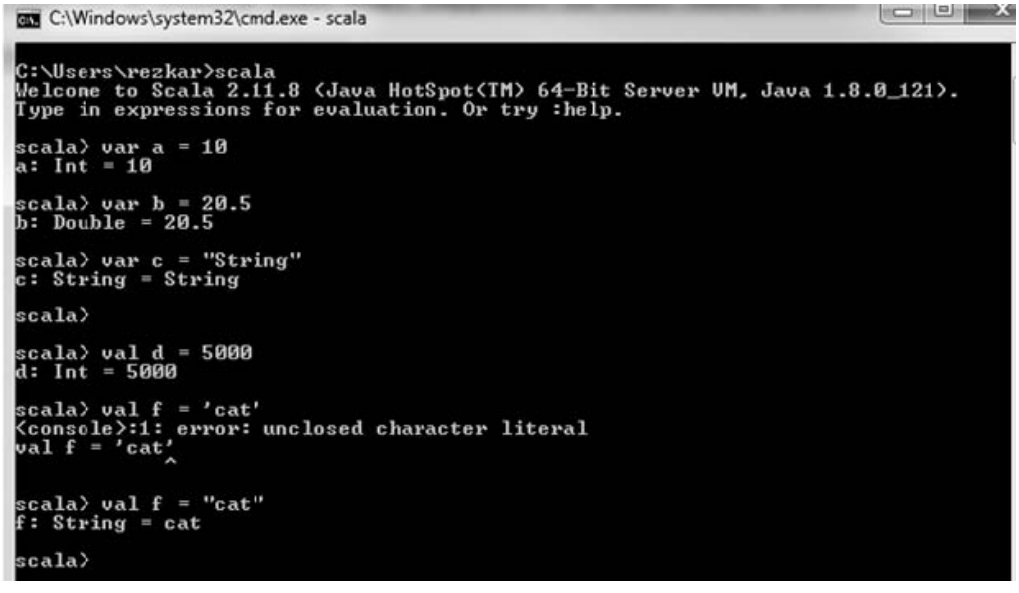



图 1.8　Scala REPL 样例 1

Scala 会抛出如下错误：

```
<console>:11: error: type mismatch;
 found : String("hello")
```

```
required: Int
     val i:Int = "hello"
                 ^
```

按照 Odersky 的说法，“应该通过富字符串(RichString)将字符映射到字符映射从而再次生成一个富字符串，一如与 Scala REP 的交互”。这句话可使用下面的代码来证明：

```
scala> "abc" map (x => (x + 1).toChar)
res0: String = bcd
```

但如果有人使用了一个方法，将字符转换为整型或者字符串，将发生什么？在这个例子中，Scala 会将其作为一个整数向量进行转换，这也称为不可变性。这是 Scala 集合的一个特性。我们将在第 9 章讲解它。我们也将在第 4 章讲述 Scala 集合的更多细节。

```
"abc" map (x => (x + 1))
res1: scala.collection.immutable.IndexedSeq[Int] = Vector(98, 99, 100)
```

在这里，无论是对象的静态方法还是动态方法都是可用的。例如，如果你将字符串 hello 声明为 x，然后尝试访问 x 的静态方法和动态方法，则它们也都是可用的。在 Scala shell 中输入 x，接着输入.再输入<tab>，就可以找到这些可用的方法：

```
scala> val x = "hello"
x:  java.lang.String = hello
scala> x.re<tab>
reduce              reduceRight         replaceAll            reverse
reduceLeft          reduceRightOption   replaceAllLiterally
reverseIterator
reduceLeftOption    regionMatches       replaceFirst          reverseMap
reduceOption        replace             repr
scala>
```

这一切都是通过反射即时完成的。因此，甚至是你刚刚定义的匿名类同样可以访问：

```
scala> val x = new AnyRef{def helloWord = "Hello, world!"}
x: AnyRef{def helloWord: String} = $anon$1@58065f0c
  scala> x.helloWord
    def helloWord: String
  scala> x.helloWord
  warning: there was one feature warning; re-run with -feature for details
  res0: String = Hello, world!
```

上述两个例子也可以在 Scala shell 中展示出来，如图 1.9 所示。

“所以这就证明了，Scala 能根据传入函数参数的结果类型来匹配不同的类型!”

——Odersky

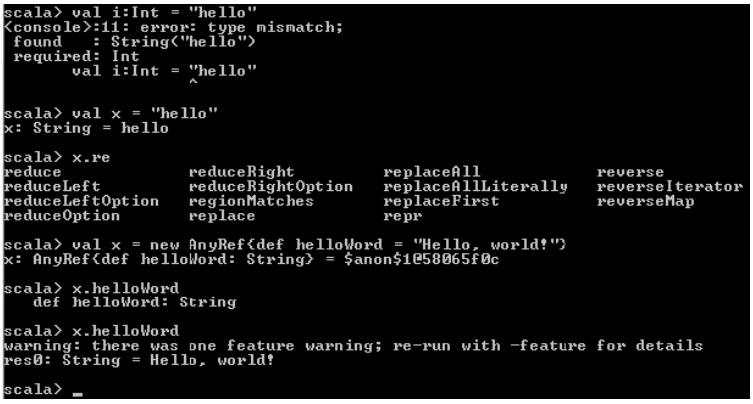



图 1.9　Scala REPL 样例 2

## 1.5.4　嵌套函数

编程语言为何需要支持嵌套函数？大多数情况下，是因为我们想让维护的方法一直都具有较少的代码行数，并避免重写大函数。Java 中，对此的一个典型解决方法是在类级别定义所有小函数，但其他方法可很容易地引用或访问这些函数。尽管这些只是一些辅助方法。但在 Scala 中，情况却有所不同。可在函数内部再定义函数。通过这种方式，阻止了对这些函数的外部访问：

```
def sum(vector: List[Int]): Int = {
  //Nested helper method (won't be accessed from outside this function
  def helper(acc: Int, remaining: List[Int]): Int = remaining match {
    case Nil => acc
    case _=> helper(acc + remaining.head, remaining.tail)
  }
  //Call the nested method
  helper(0, vector)
}
```

在此，我们并不期望你立刻就能理解这些代码片段，它只是为了展示 Scala 与 Java 之间的不同之处。

## 1.5.5　导入语句

在 Java 中，你只能在代码文件头部导入包，其位置是在包语句的右边。但是在 Scala 中情况就有所不同。可在源文件内部的几乎任意位置编写导入包的语句(例如，甚至可在一个类或方法的内部编写导入语句)。你只需要注意导入语句的作用域即可。因为它继承了方法中的本地变量或类成员的作用域。Scala 中的_(下画线)可用作导入时的通配符，它与你在 Java 中使用的*(星号)类似：

```
//Import everything from the package math
import math._
```

如果想使用{ }来表明导入的对象集合来自于同一个父包，只需要使用一行代码即可。在 Java 中，你可能需要使用多行代码来完成这样的工作。

```
//Import math.sin and math.cos
import math.{sin, cos}
```

与 Java 不同，Scala 中并没有静态导入的概念。换句话说，Scala 中并不存在静态的概念。但作为一名开发人员，很明显，可以使用一条正常的导入语句来导入一个或者多个对象。前面的例子已经展示了这样的内容。我们从一个名为 Math 的包对象中导入 sin 和 cos 方法。为展示这个例子，如果是从 Java 编程人员的角度出发，则应该按照如下方式定义前面的代码片段：

```
import static java.lang.Math.sin;
import static java.lang.Math.cos;
```

Scala 的另一个优美之处在于，在 Scala 中，也可以重命名你导入的包，以免与包含相似对象的包出现类型冲突。如下语句在 Scala 中是有效的：

```
//Import Scala.collection.mutable.Map as MutableMap
import Scala.collection.mutable.{Map => MutableMap}
```

最后，在导入时，你可能想排除包中的一些对象。可以使用通配符来完成这一操作：

```
//Import everything from math, but hide cos
import math.{cos => _, _}
```

## 1.5.6　作为方法的操作符

值得提醒的是，Scala 并不支持操作符重载。有人可能因此认为，Scala 中没有操作符。

作为调用只有一个参数的方法的一种替代，可以使用中缀语法。该语法为你提供了一个新的玩法，就像是你使用了操作符重载，跟你在 C++中的动作类似。例如：

```
val x = 45
val y = 75
```

在下面的例子中，+指的是类 Int 中的一个方法。如下代码是一个无转换方法的调用语法：

```
val add1 = x.+(y)
```

更正式一点，也可以使用中缀语法来完成这个操作，如下：

```
val add2 = x + y
```

不仅如此，还可以使用中缀语法。但该方法只有一个参数，如下：

```
val my_result = List(3, 6, 15, 34, 76) contains 5
```

这是使用中缀语法的一个特例。也就是说，如果方法的名称以:(冒号)结束，则调用就是向右关联(right associative)。这意味着该方法是在右边的参数上调用(左边的表达式作为参数)，而不是以另一种方式被调用。例如，下面的句子在 Scala 中是有效的：

```
val my_list = List(3, 6, 15, 34, 76)
```

是 my_list.+:(5)，而不是 5.+:(my_list)。更正式的写法是：

```
val my_result = 5 +: my_list
```

现在，让我们在 Scala REPL 中看一下前面这个例子：

```
scala> val my_list = 5 +: List(3, 6, 15, 34, 76)
 my_list: List[Int] = List(5, 3, 6, 15, 34, 76)
scala> val my_result2 = 5+:my_list
 my_result2: List[Int] = List(5, 5, 3, 6, 15, 34, 76)
scala> println(my_result2)
 List(5, 5, 3, 6, 15, 34, 76)
scala>
```

这里的操作符仅是方法，因此它们也像方法那样，可以被简单地进行重载。

## 1.5.7　方法与参数列表

在 Scala 中，一个方法可以有多个参数列表，或者是一个参数列表都没有。另外，在 Java 中，一个方法通常都有一个参数列表，可以是 0 个或者多个参数。例如，在 Scala 中，如下的方法定义语句是有效的(使用 currie 符号)，该方法有两个参数列表：

```
def sum(x: Int)(y: Int) = x + y
```

该语句不能写成如下形式：

```
def sum(x: Int, y: Int) = x + y
```

一个方法(这里将其称为 sum2)也可以没有一个参数列表，如下：

```
def sum2 = sum(2) _
```

现在，可调用方法 add2，它会返回带有一个参数的函数。然后，它使用参数 5 调用该函数，如下：

```
val result = add2(5)
```

## 1.5.8　方法内部的方法

有时，你想让应用代码更模块化一些，从而避免出现一些太长或太复杂的方法。Scala 为你提供了这样的功能，以免方法变得过于庞大。这样可将它们拆分成几个较小的方法。

另外，Java 则只允许你在类级别定义方法。例如，假设你有如下的方法定义：

```
def main_method(xs: List[Int]): Int = {
  //This is the nested helper/auxiliary method
  def auxiliary_method(accu: Int, rest: List[Int]): Int = rest match {
    case Nil => accu
    case _=> auxiliary_method(accu + rest.head, rest.tail)
  }
}
```

现在，可以按照如下方式来调用该辅助方法：

```
auxiliary_method(0, xs)
```

考虑一下上面的内容，如下是一个有效的完整代码段：

```
def main_method(xs: List[Int]): Int = {
  //This is the nested helper/auxiliary method
  def auxiliary_method(accu: Int, rest: List[Int]): Int = rest match {
    case Nil => accu
    case _=> auxiliary_method(accu + rest.head, rest.tail)
  }
  auxiliary_method(0, xs)
}
```

## 1.5.9　Scala 中的构造器

Scala 中令人惊奇的一件事情，就是 Scala 类体本身就是一个构造器。但事实上，Scala 是以更明确的方式来实现这一点的。此后，该类的一个实体就被创建出来并运行。不仅如此，在类的声明行里，还可以指定参数。

结论是，该类中定义的所有方法都可访问构造器的参数。例如，如下的类和构造器定义在 Scala 中都是有效的：

```
class Hello(name: String) {
  //Statement executed as part of the constructor
  println("New instance with name: " + name)
  //Method which accesses the constructor argument
  def sayHello = println("Hello, " + name + "!")
}
```

等价的 Java 类就像下面这样：

```
public class Hello {
  private final String name;
  public Hello(String name) {
    System.out.println("New instance with name: " + name);
    this.name = name;
  }
  public void sayHello() {
    System.out.println("Hello, " + name + "!");
  }
}
```

## 1.5.10　代替静态方法的对象

正如在前面提到的，Scala 中不存在静态。你无法进行静态导入，或给类添加静态方法。在 Scala 中，当你在同一个源文件中定义对象，并且该对象的名称与类名相同时，该对象就被称为类的伴生(companion)对象。你在类的伴生对象中定义的函数和 Java 中类的静态方法类似：

```
class HelloCity(CityName: String) {
  def sayHelloToCity = println("Hello, " + CityName + "!")
}
```

这里展示了你如何为类 Hello 定义一个伴生对象：

```
object HelloCity {
  //Factory method
  def apply(CityName: String) = new Hello(CityName)
}
```

在 Java 中等价的类则像下面这样：

```
public class HelloCity {
  private final String CityName;
  public HelloCity(String CityName) {
    this.CityName = CityName;
  }
  public void sayHello() {
    System.out.println("Hello, " + CityName + "!");
  }
  public static HelloCity apply(String CityName) {
    return new Hello(CityName);
  }
}
```

所以，这个简单的类中就有了这么冗长的内容，不是吗？Scala 中的应用方法是以不同方式被处理的。这样就可以找到一种特殊的快捷方式来调用它。如下是你熟悉的调用方法：

```
val hello1 = Hello.apply("Dublin")
```

下面则使用快捷方式来调用，与上面的方式等价：

```
val hello2 = Hello("Dublin")
```

要注意，这只有在代码中使用应用方法时才有效，因为 Scala 以这种特殊方法来对待指定的应用。

### 1.5.11　特质

Scala 提供了一个卓越功能来扩展并丰富类的行为。这些特质(trait)与你定义函数原型(function prototype)或签名的接口类似。因此，有了这项功能，就拥有了来自不同特质的混合功能，并基于此种方式，就丰富了类的行为。所以，Scala 中的特质有什么好处？因为它们使得来自不同特质的类进行组合成为可能，而这些特质可用来构建基块。与往常一样，我们来看一个例子。这是 Java 中常规的建立日志记录的方法。

要注意，尽管可按自己的想法混合任意数量的特质。但与 Java 一样，Scala 确实不支持多重继承。但无论是 Java 还是 Scala，一个子类都只能扩展一个超类。例如，在 Java 中：

```
class SomeClass {
  //First, to have to log for a class, you must initialize it
  final static Logger log = LoggerFactory.getLogger(this.getClass());
  ...
  //For logging to be efficient, you must always check, if logging level
  //for current message is enabled
```

```
  //BAD, you will waste execution time if the log level is an error, fatal,
  //etc.
  log.debug("Some debug message");
  ...
  //GOOD, it saves execution time for something more useful
  if (log.isDebugEnabled()) { log.debug("Some debug message"); }
  //BUT looks clunky, and it's tiresome to write this construct every time
  //you want to log something.
}
```

要了解关于此内容的更详细的讨论，请浏览 https://stackoverflow.com/questions/963492/in-log4j-does-checking-isdebugenabled-before-logging-improve-performance/963681#963681。

但特质则有所不同。经常检查被启用的日志级别其实是一件很无聊的事情。但如果你写了一次这种例行检查的代码，然后就在任意类中予以复用，这个就比较好了。Scala 中的特质使得这样的想法成为可能，例如：

```
trait Logging {
  lazy val log = LoggerFactory.getLogger(this.getClass.getName)
  //Let's start with info level...
  ...
  //Debug level here...
  def debug() {
    if (log.isDebugEnabled) log.info(s"${msg}")
}
  def debug(msg: => Any, throwable: => Throwable) {
    if (log.isDebugEnabled) log.info(s"${msg}", throwable)
  }
  ...
  //Repeat it for all log levels you want to use
}
```

如果你查看了上述代码，就会看到一个以 s 开头的字符串的例子。Scala 提供了一种从你提供的数据中创建字符串的机制，称为字符串插值(string interpolation)。

**注意：**

字符串插值允许你将被引用的变量直接插入字符串文本中。例如：

```
scala> val name = "John Breslin"
scala> println(s"Hello, $name") //Hello, John Breslin.
```

现在，我们就能获得一种高效的日志程序，并采用一种更传统的可复用代码块的形式。为对任意类都启用日志记录，只需要将其混入 Logging 特质即可！太神奇了！现在，下面就是将 Logging 特质添加进类的全部内容：

```
class SomeClass extends Logging {
  ...
  //With logging trait, no need for declaring a logger manually for every
  //class
  //And now, your logging routine is either efficient and doesn't litter
  //the code!
```

```
  log.debug("Some debug message")
  ...
}
```

并且，将多个特质进行混合也是可能的。例如，对于前述的特质(也就是 Logging)，可按如下顺序进行扩展：

```
trait Logging {
  override def toString = "Logging "
}
class A extends Logging {
  override def toString = "A->" + super.toString
}
trait B extends Logging {
  override def toString = "B->" + super.toString
}
trait C extends Logging {
  override def toString = "C->" + super.toString
}
class D extends A with B with C {
  override def toString = "D->" + super.toString
}
```

但需要注意，Scala 的类可一次性扩展多个特质，而 JVM 类一次则只能扩展一个父类。

现在，为调用上述特质和类，我们在 Scala REPL 中使用 new D() ，如图 1.10 所示。

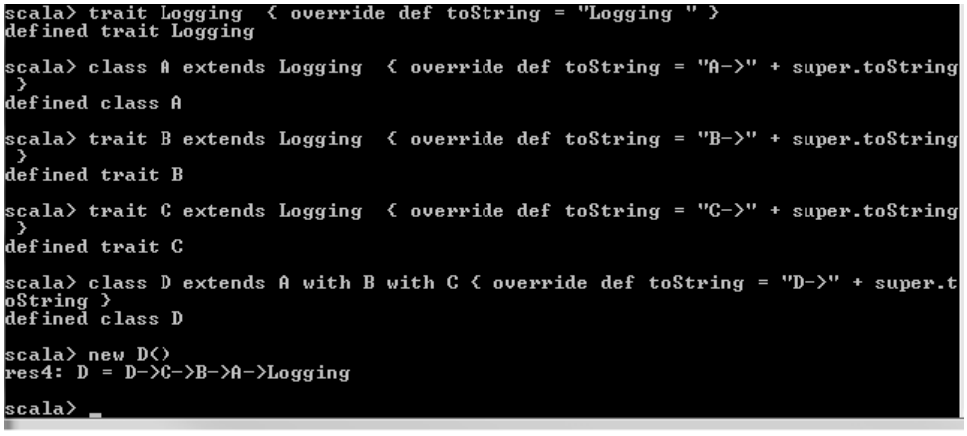



图 1.10　混合多个特质

到目前为止，本章的一切内容看起来都很顺利。现在，让我们开始一个新的部分，这里将讨论一些与初学者相关的主题。

## 1.6　面向初学者的 Scala

在这部分，你将发现，我们会假设你已经对其他编程语言有了初步的理解。如果 Scala 是你

进入代码世界所接触到的第一门编程语言，你将在互联网上找到大量专门为初学者解释 Scala 的材料、辅导内容、视频和课程。

**小技巧：**

在 Coursera 上，有专门的关于 Scala 的内容：https://www.coursera.org/specializations/scala。该课程由 Scala 的创建者 Martin Odersky 亲自讲授。该在线课程采用了一种带有学院派风格的授课方式来讲解函数式编程的基础知识。通过完成一些编程的作业，你将学到很多关于 Scala 的知识。不仅如此，该网站上的这一部分还包含了 Apache Spark 相关的课程。此外，Kojo(http://www.kogics.net/sf:kojo)是一个交互式学习环境，在这里可使用 Scala 进行编程，从而探索数学、艺术、音乐、动画，当然还有游戏。

## 1.6.1 你的第一行代码

作为第一个例子，我们将使用相当流行的“Hello, world!”程序来展示如何使用 Scala 以及相关的工具，当然你不必对这些了解太多就可上手。打开你喜爱的编辑器(这里的例子是运行在 Windows 7 上的，当然也可运行在 Ubuntu 或 macOS 上)，也就是 Notepad++，然后输入如下代码行：

```
object HelloWorld {
  def main(args: Array[String]){
    println("Hello, world!")
  }
}
```

现在，将这些代码保存，并指定一个名字：HelloWorld.scala，如图 1.11 所示。

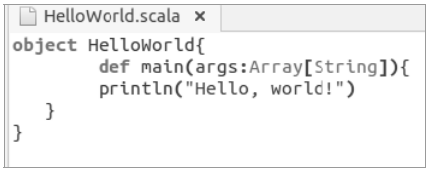



图 1.11　使用 Notepad++保存你的第一个 Scala 源代码

然后以如下方式编译该源文件：

```
C:\>scalac HelloWorld.scala
  C:\>scala HelloWorld
  Hello, world!
  C:\>
```

对于具备一些编程经验的人员而言，Scala 的程序看起来应该也是比较熟悉的。它有一个 main 方法用于在控制台上打印字符串“Hello, world!”。接下来，为了看清是如何定义 main 函数的，我们使用了 def main()这样奇怪的语法来定义它。def 是 Scala 的关键字，用于声明/定义一个方法。然后，我们使用 Array[String]作为该方法的一个参数。该参数是一个字符串的数组，可用来对程序进行初始化配置。如果省略它也是允许的。接下来使用通用的 println()方法，它带有一个字符串(可能是一个已经格式化的字符串)并将其打印到控制台上。一个简单的 HelloWorld 就开启了许

多需要学习的主题。主要有三个：

- 方法(在稍后的章节中进行讲解)
- 对象与类(在稍后的章节中进行讲解)
- 类型推导(前面已解释过)

## 1.6.2　交互式运行 Scala！

scala 命令可以为你开始交互式 shell，这样就可以交互式地解释 Scala 的表达式：

```
> scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_121).
Type in expressions for evaluation.Or try :help.
scala>
scala> object HelloWorld {
    |   def main(args: Array[String]){
    |     println("Hello, world!")
    |   }
    | }
defined object HelloWorld
scala> HelloWorld.main(Array())
Hello, world!
scala>
```

**注意：**
快捷方式:q 代表内部 shell 命令 quit，用于退出解释器。

## 1.6.3　编译

与 javac 命令相似，scalac 命令可用来编译一个或多个 Scala 源文件，并生成字节码作为输出，然后就可以在任意 JVM 上执行。为了编译 HelloWorld 对象，可使用如下命令：

```
> scalac HelloWorld.scala
```

默认情况下，scalac 会在当前工作目录下生成类文件。也可以使用-d 选项来指定不同的输出目录：

```
> scalac -d classes HelloWorld.scala
```

但要注意，这里的 classes 目录必须要在执行该命令之前就已创建完毕。

### 使用 scala 命令执行它

scala 命令执行由解释器生成的字节码：

```
$ scala HelloWorld
```

scala 命令允许我们指定不同的命令选项，例如-classpath(别名-cp)选项：

```
$ scala -cp classes HelloWorld
```

在使用 scala 命令执行源文件之前，你应该有一个 main 方法，它会作为应用程序的入口点。换言之，你应该有一个对象来扩展 Trait Scala.App，然后该对象中包含的所有代码都将被该命令执行。如下是同样的“Hello, world!”例子，但使用了 App 特质：

```
#!/usr/bin/env Scala
object HelloWorld extends App {
  println("Hello, world!")
}
HelloWorld.main(args)
```

上述脚本可以直接在命令 shell 中运行：

```
./script.sh
```

注意，这里假设文件 script.sh 具有执行权限：

```
$ sudo chmod +x script.sh
```

然后，scala 命令的搜索路径也已在$PATH 环境变量中予以设置。

## 1.7　本章小结

本章介绍了 Scala 编程语言的基础知识、特性和可用的编辑器，也简要探讨了 Scala 及其语法。对于初学者，尤其是初次接触 Scala 编程语言的人员来说，我们也展示了如何安装并建立开发环境。本章还讲述了如何编写、编译以及执行一段简单的 Scala 代码。不仅如此，为便于具有 Java 背景的人员学习，还进行了 Scala 和 Java 之间的比较。

Scala 是静态类型，但 Python 是动态类型。Scala 绝大部分情况下遵循函数式编程范例，而 Python 却非如此。Python 具有独特的语法，并且大部分情况下不使用括号，但是 Scala 大部分情况下则需要它们。在 Scala 中，几乎所有东西都是表达式，但在 Python 中显然并非如此。但有些特点看起来则错综复杂。另外，根据 https://stackoverflow.com/questions/1065720/what-is-the-purpose-of-scala-programming-language/5828684#5828684 所提供的文档，Scala 编译器就像一个自由测试工具，其文档太过复杂。如果能恰当地部署 Scala，它几乎能完成所有事情，原因在于 Scala 拥有其一致且连贯的 API。

第 2 章将指导你进一步掌握基础知识，以便你了解 Scala 如何实现面向对象的范例，该范例允许我们创建模块化的软件系统。