

O'REILLY®

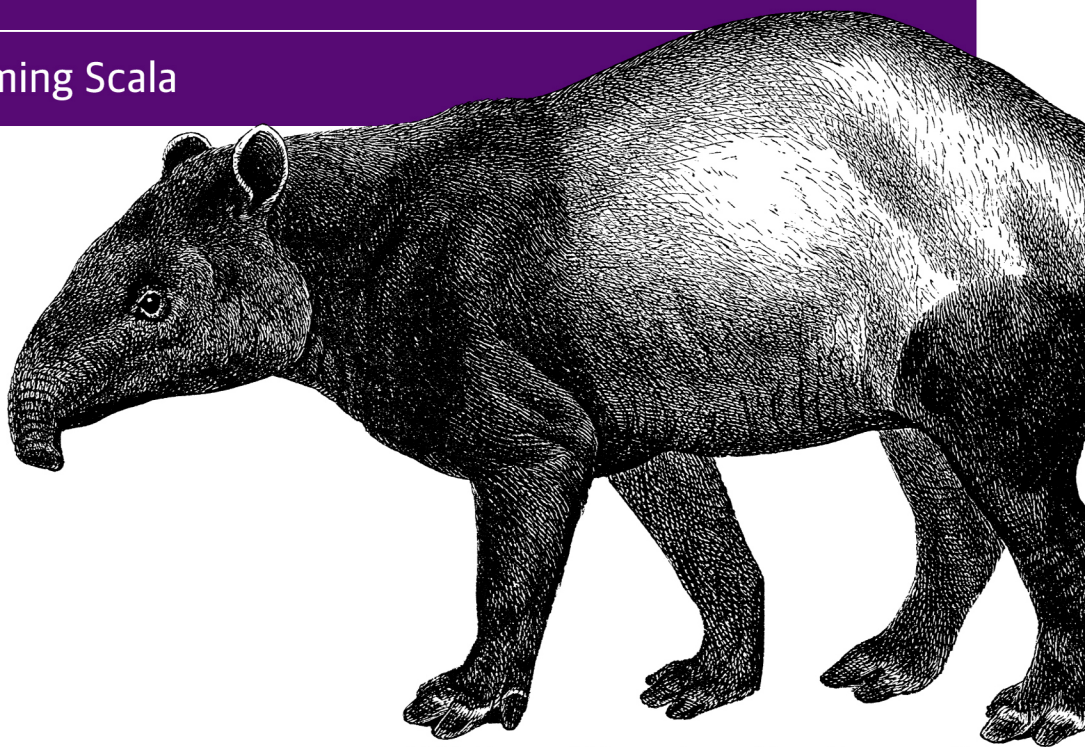


图灵程序设计丛书

第2版

Scala 程序设计

Programming Scala



[美] Dean Wampler Alex Payne 著
王渊 陈明 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

译者介绍



王渊

现任Amazon中国研发工程师，拥有十余年编码经验，曾就职于Borland、Adobe等多家公司，担任工程师、技术经理等职位，擅长分布式系统以及高性能网站开发。



陈明

毕业于天津大学，现就职于奇虎360，担任服务端工程师。主要从事C++服务器引擎的开发和海量数据的实时处理。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



图灵程序设计丛书

Scala程序设计（第2版）

Programming Scala

[美] Dean Wampler Alex Payne 著
王渊 陈明 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北 京

图书在版编目 (C I P) 数据

Scala程序设计 : 第2版 / (美) 万普勒
(Wampler, D.), (美) 佩恩 (Payne, A.) 著 ; 王渊, 陈
明译. — 北京 : 人民邮电出版社, 2016. 3
(图灵程序设计丛书)
ISBN 978-7-115-41681-0

I. ①S… II. ①万… ②佩… ③王… ④陈… III. ①
JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2016)第023309号

内 容 提 要

本书通过大量的代码示例, 全面介绍 Scala 这门针对 JVM 的编程语言, 向读者展示了如何高效地利用 Scala 语言及其生态系统, 同时解释了为何 Scala 是开发高扩展性、以数据为中心的应用程序的理想语言。

本书既适合 Scala 初学者入门, 也适合经验丰富的 Scala 开发者参考。

-
- ◆ 著 [美] Dean Wampler Alex Payne
译 王 渊 陈 明
责任编辑 岳新欣
执行编辑 刘 敏
责任印制 杨林杰
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷
 - ◆ 开本: 800×1000 1/16
印张: 31.25
字数: 761千字 2016年3月第1版
印数: 1—3 000册 2016年3月北京第1次印刷
著作权合同登记号 图字: 01-2015-6359号
-

定价: 109.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

版权声明

© 2015 by Dean Wampler and Alex Payne.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2015。

简体中文版由人民邮电出版社出版，2016。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

序	xv
前言	xvii
第 1 章 零到六十：Scala 简介	1
1.1 为什么选择 Scala	1
1.1.1 富有魅力的 Scala	2
1.1.2 关于 Java 8	3
1.2 安装 Scala	3
1.2.1 使用 SBT	5
1.2.2 执行 Scala 命令行工具	6
1.2.3 在 IDE 中运行 Scala REPL	8
1.3 使用 Scala	8
1.4 并发	17
1.5 本章回顾与下一章提要	27
第 2 章 更简洁，更强大	28
2.1 分号	28
2.2 变量声明	29
2.3 Range	31
2.4 偏函数	32
2.5 方法声明	33
2.5.1 方法默认值和命名参数列表	33
2.5.2 方法具有多个参数列表	34
2.5.3 Future 简介	35

2.5.4 嵌套方法的定义与递归	38
2.6 推断类型信息	40
2.7 保留字	44
2.8 字面量	46
2.8.1 整数字面量	46
2.8.2 浮点数字面量	47
2.8.3 布尔型字面量	48
2.8.4 字符字面量	48
2.8.5 字符串字面量	48
2.8.6 符号字面量	50
2.8.7 函数字面量	50
2.8.8 元组字面量	50
2.9 Option、Some 和 None：避免使用 null	52
2.10 封闭类的继承	53
2.11 用文件和名空间组织代码	54
2.12 导入类型及其成员	55
2.12.1 导入是相对的	56
2.12.2 包对象	57
2.13 抽象类型与参数化类型	57
2.14 本章回顾与下一章提要	59
第 3 章 要点详解	60
3.1 操作符重载 ?	60
3.2 无参数方法	63
3.3 优先级规则	64
3.4 领域特定语言	65
3.5 Scala 中的 if 语句	66
3.6 Scala 中的 for 推导式	67
3.6.1 for 循环	67
3.6.2 生成器表达式	67
3.6.3 保护式：筛选元素	67
3.6.4 Yielding	68
3.6.5 扩展作用域与值定义	69
3.7 其他循环结构	70
3.7.1 Scala 的 while 循环	71
3.7.2 Scala 中的 do-while 循环	71
3.8 条件操作符	71
3.9 使用 try、catch 和 final 子句	72
3.10 名字调用和值调用	75

3.11 惰性赋值	78
3.12 枚举	79
3.13 可插入字符串	81
3.14 Trait: Scala 语言的接口和“混入”	83
3.15 本章回顾与下一章提要	85
第 4 章 模式匹配	86
4.1 简单匹配	86
4.2 match 中的值、变量和类型	87
4.3 序列的匹配	90
4.4 元组的匹配	94
4.5 case 中的 guard 语句	94
4.6 case 类的匹配	95
4.6.1 unapply 方法	96
4.6.2 unapplySeq 方法	100
4.7 可变参数列表的匹配	101
4.8 正则表达式的匹配	103
4.9 再谈 case 语句的变量绑定	104
4.10 再谈类型匹配	104
4.11 封闭继承层级与全覆盖匹配	105
4.12 模式匹配的其他用法	107
4.13 总结关于模式匹配的评价	111
4.14 本章回顾与下一章提要	111
第 5 章 隐式详解	112
5.1 隐式参数	112
5.2 隐式参数适用的场景	115
5.2.1 执行上下文	115
5.2.2 功能控制	115
5.2.3 限定可用实例	116
5.2.4 隐式证据	120
5.2.5 绕开类型擦除带来的限制	122
5.2.6 改善报错信息	124
5.2.7 虚类型	124
5.2.8 隐式参数遵循的规则	127
5.3 隐式转换	128
5.3.1 构建独有的字符串插入器	132
5.3.2 表达式问题	134
5.4 类型类模式	135

5.5	隐式所导致的技术问题	137
5.6	隐式解析规则	139
5.7	Scala 内置的各种隐式	139
5.8	合理使用隐式	146
5.9	本章回顾与下一章提要	146
第 6 章	Scala 函数式编程	147
6.1	什么是函数式编程	148
6.1.1	数学中的函数	148
6.1.2	不可变变量	149
6.2	Scala 中的函数式编程	151
6.2.1	匿名函数、Lambda 与闭包	152
6.2.2	内部与外部的纯粹性	154
6.3	递归	154
6.4	尾部调用和尾部调用优化	155
6.5	偏应用函数与偏函数	157
6.6	Curry 化与函数的其他转换	158
6.7	函数式编程的数据结构	162
6.7.1	序列	162
6.7.2	映射表	166
6.7.3	集合	168
6.8	遍历、映射、过滤、折叠与归约	168
6.8.1	遍历	169
6.8.2	映射	170
6.8.3	扁平映射	172
6.8.4	过滤	173
6.8.5	折叠与归约	174
6.9	向左遍历与向右遍历	178
6.10	组合器：软件最佳组件抽象	183
6.11	关于复制	186
6.12	本章回顾与下一章提要	188
第 7 章	深入学习 for 推导式	189
7.1	内容回顾：for 推导式组成元素	189
7.2	for 推导式：内部机制	192
7.3	for 推导式的转化规则	194
7.4	Option 以及其他的一些容器类型	197
7.4.1	Option 容器	197
7.4.2	Either：Option 类型的逻辑扩展	200

7.4.3 Try 类型	205
7.4.4 Scalaz 提供的 Validation 类	206
7.5 本章回顾与下一章提要	209
第 8 章 Scala 面向对象编程	210
8.1 类与对象初步	211
8.2 引用与值类型	213
8.3 价值类	214
8.4 父类	217
8.5 Scala 的构造器	217
8.6 类的字段	221
8.6.1 统一访问原则	223
8.6.2 一元方法	224
8.7 验证输入	224
8.8 调用父类构造器（与良好的面向对象设计）	226
8.9 嵌套类型	230
8.10 本章回顾与下一章提要	232
第 9 章 特征	233
9.1 Java 8 中的接口	233
9.2 混入 trait	234
9.3 可堆叠的特征	238
9.4 构造 trait	243
9.5 选择类还是 trait	244
9.6 本章回顾与下一章提要	245
第 10 章 Scala 对象系统 (I)	246
10.1 参数化类型：继承转化	246
10.1.1 Hood 下的函数	247
10.1.2 可变类型的变异	250
10.1.3 Scala 和 Java 中的变异	252
10.2 Scala 的类型层次结构	253
10.3 闲话 Nothing（以及 Null）	254
10.4 Product、case 类和元组	258
10.5 Predef 对象	260
10.5.1 隐式转换	260
10.5.2 类型定义	262
10.5.3 条件检查方法	263
10.5.4 输入输出方法	263
10.5.5 杂项方法	265

10.6	对象的相等	265
10.6.1	equals 方法	266
10.6.2	== 和 != 方法	266
10.6.3	eq 和 ne 方法	267
10.6.4	数组相等和 sameElements 方法	267
10.7	本章回顾与下一章提要	268
第 11 章	Scala 对象系统 (II)	269
11.1	覆写类成员和 trait 成员	269
11.2	尝试覆写 final 声明	272
11.3	覆写抽象方法和具体方法	272
11.4	覆写抽象字段和具体字段	274
11.5	覆写抽象类型	280
11.6	无须区分访问方法和字段：统一访问原则	280
11.7	对象层次结构的线性化算法	282
11.8	本章回顾与下一章提要	287
第 12 章	Scala 集合库	288
12.1	通用、可变、不可变、并发以及并行集合	288
12.1.1	scala.collection 包	289
12.1.2	collection.concurrent 包	290
12.1.3	collection.convert 包	291
12.1.4	collection.generic 包	291
12.1.5	collection.immutable 包	291
12.1.6	scala.collection.mutable 包	292
12.1.7	scala.collection.parallel 包	294
12.2	选择集合	295
12.3	集合库的设计惯例	296
12.3.1	Builder	296
12.3.2	CanBuildFrom	297
12.3.3	Like 特征	298
12.4	值类型的特化	298
12.5	本章回顾与下一章提要	300
第 13 章	可见性规则	301
13.1	默认可见性：公有可见性	301
13.2	可见性关键字	302
13.3	Public 可见性	303
13.4	Protected 可见性	304
13.5	Private 可见性	305

13.6	作用域内私有和作用域内受保护可见性	306
13.7	对可见性的想法	312
13.8	本章回顾与下一章提要	313
第 14 章 Scala 类型系统 (I)		314
14.1	参数化类型	315
14.1.1	变异标记	315
14.1.2	类型构造器	315
14.1.3	类型参数的名称	315
14.2	类型边界	315
14.2.1	类型边界上限	316
14.2.2	类型边界下限	316
14.3	上下文边界	320
14.4	视图边界	320
14.5	理解抽象类型	322
14.6	自类型标记	325
14.7	结构化类型	329
14.8	复合类型	332
14.9	存在类型	334
14.10	本章回顾与下一章提要	335
第 15 章 Scala 类型系统 (II)		336
15.1	路径相关类型	336
15.1.1	C.this	337
15.1.2	C.super	337
15.1.3	path.x	338
15.2	依赖方法类型	339
15.3	类型投影	340
15.4	值的类型	343
15.4.1	元组类型	343
15.4.2	函数类型	343
15.4.3	中缀类型	343
15.5	Higher-Kinded 类型	344
15.6	类型 Lambda	348
15.7	自递归类型: F-Bounded 多态	350
15.8	本章回顾与下一章提要	351
第 16 章 高级函数式编程		352
16.1	代数数据类型	352

16.1.1	加法类型与乘法类型	352
16.1.2	代数数据类型的属性	354
16.1.3	代数数据类型的最后思考	355
16.2	范畴理论	355
16.2.1	关于范畴	356
16.2.2	Functor 范畴	356
16.2.3	Monad 范畴	360
16.2.4	Monad 的重要性	362
16.3	本章回顾与下一章提要	363
第 17 章	并发工具	365
17.1	scala.sys.process 包	365
17.2	Future 类型	367
17.3	利用 Actor 模型构造稳固且可扩展的并发应用	371
17.4	Akka: 为 Scala 设计的 Actor 系统	372
17.5	Pickling 和 Spores	383
17.6	反应式编程	384
17.7	本章回顾与下一章提要	385
第 18 章	Scala 与大数据	386
18.1	大数据简史	386
18.2	用 Scala 改善 MapReduce	387
18.3	超越 MapReduce	392
18.4	数学范畴	393
18.5	Scala 数据工具列表	394
18.6	本章回顾与下一章提要	394
第 19 章	Scala 动态调用	396
19.1	一个较为激进的示例: Ruby on Rails 框架中的 ActiveRecord 库	396
19.2	使用动态特征实现 Scala 中的动态调用	397
19.3	关于 DSL 的一些思考	402
19.4	本章回顾与下一章提要	402
第 20 章	Scala 的领域特定语言	403
20.1	DSL 示例: Scala 中 XML 和 JSON DSL	404
20.2	内部 DSL	406
20.3	包含解析组合子的外部 DSL	410
20.3.1	关于解析组合子	410
20.3.2	计算工资单的外部 DSL	410
20.4	内部 DSL 与外部 DSL: 最后的思考	413

20.5 本章回顾与下一章提要	413
第 21 章 Scala 工具和库	414
21.1 命令行工具	414
21.1.1 命令行工具: scalac	414
21.1.2 Scala 命令行工具	418
21.1.3 scalap 和 javap 命令行工具	421
21.1.4 scaladoc 命令行工具	422
21.1.5 fsc 命令行工具	422
21.2 构建工具	422
21.2.1 SBT: Scala 标准构建工具	423
21.2.2 其他构建工具	425
21.3 与 IDE 或文本编辑器集成	425
21.4 在 Scala 中应用测试驱动开发	426
21.5 第三方库	427
21.6 本章回顾与下一章提要	429
第 22 章 与 Java 的互操作	430
22.1 在 Scala 代码中使用 Java 名称	430
22.2 Java 泛型与 Scala 泛型	430
22.3 JavaBean 的性质	432
22.4 AnyVal 类型与 Java 原生类型	433
22.5 Java 代码中的 Scala 名称	433
22.6 本章回顾与下一章提要	434
第 23 章 应用程序设计	435
23.1 回顾之前的内容	435
23.2 注解	437
23.3 Trait 即模块	441
23.4 设计模式	442
23.4.1 构造型模式	443
23.4.2 结构型模式	443
23.4.3 行为型模式	444
23.5 契约式设计带来更好的设计	446
23.6 帕特农神庙架构	448
23.7 本章回顾与下一章提要	453
第 24 章 元编程: 宏与反射	454
24.1 用于理解类型的工具	455
24.2 运行时反射	455

24.2.1 类型反射	455
24.2.2 ClassTag、TypeTag 与 Manifest	457
24.3 Scala 的高级运行时反射 API	458
24.4 宏	461
24.4.1 宏的示例：强制不变性	463
24.4.2 关于宏的最后思考	466
24.5 本章回顾与下一章提要	466
附录 A 参考文献	468
作者简介	473
关于封面	473

序

作为一名程序员，我的职业生涯中一直贯穿着这样的主题：寻求更好的抽象和更好的工具来编写更好的软件。经过了这些年，我认为可组合性（composability）是一项比其他特征更重要的特征。如果我们编写的代码具有很好的可组合性，这通常意味着这些代码同样具备软件工程师所看重的其他特征，如正交性（orthogonality）、松耦合性以及高聚合性（high cohesion）。这些都是互通的。

几年前，当我发现 Scala 语言时，它的可组合性便给我带来了很大的震撼。

Martin Odersky 创造 Scala 时，运用了一些简洁的设计方法以及源于面向对象和函数式编程的一些看似简单却很强大的抽象，这使得 Scala 具备高聚合性，而具备了正交性的高度抽象则给这门语言带来可用于软件设计各个方面的可组合性。Scala 是一门真正具备了可扩展性的语言，我们既能使用它编写各种脚本语言，也能使用它实现大规模企业应用和中间件。

Scala 起源于学术界，却已经成长为了一门注重实用性的语言，对于那些真实生产环境中的应用场景，Scala 已经完全准备好了。

《Scala 程序设计》一书的实用性让我感到兴奋。Dean 干得太棒了，除了使用有趣的讨论和示例对 Scala 这门语言进行讲解之外，还将这些内容套到真实世界的应用场景中。这本书是为那些希望能够解决实际问题的程序员所编写的。

几年前，我们还都是面向切面编程委员会的成员时，我认识了 Dean。我很庆幸能够认识他。Dean 拥有一个少见的混合型大脑，他既能思考高深的学术问题，也能想到如何运用实际的方法解决问题。

通过阅读这本书，你将学到如何使用 mixin 和函数组合编写可重用组件；如何运用 Akka 库编写响应式（reactive）应用；如何高效地使用 Scala 提供的一些高级特征，如宏、higher kinded 类型；如何通过 Scala 的丰富、灵活而又富有表现力的语法构造领域特定语言；如何有效地测试你的 Scala 代码；如何通过 Scala 简化大数据问题，等等。

读者们，请好好享受阅读这本书的时光，正如我所做的那样。

——Jonas Bonér

Typesafe 公司联合创始人兼技术总监，2014 年 8 月

前言

《Scala 程序设计》向读者介绍了一门既令人振奋又功能强大的语言，该门语言集合了现代对象模型、函数式编程以及先进类型系统的所有优点，同时又能应用获得产业界大量投资的 Java 虚拟机 (JVM)。这本书通过大量的代码示例，向读者全面阐述了如何使用 Scala 迅速编写代码，解释了为什么 Scala 是编写可扩展、分布式、基于组件且支持并发和分布的应用程序的最完美语言。Scala 运行在先进的 JVM 平台之上，通过阅读本书，读者还能了解到 Scala 是如何发挥 JVM 平台优势的。

如果你想了解更多内容，请访问 <http://programming-scala.org> 或查阅本书目录 (<http://shop.oreilly.com/product/0636920033073.do>)。

欢迎阅读《Scala程序设计（第2版）》

本书第 1 版出版于 2009 年秋，是当时市面上第三本讲述 Scala 的图书，仅仅因为耽误了几个月，未能成为第二本。Scala 当时的官方版本号为 2.7.5，而 2.8.0 版则接近完工。

从那时起，Scala 世界发生了巨大的变化。编写本书时，Scala 的版本号为 2.11.2。为了进一步提升 Scala 语言以及相关工具，Scala 的创建者 Martin Odersky 与基于 actor 模型的并发框架 Akka 的作者 Jonas Bonér 一同创立了 Typesafe (<http://typesafe.com>) 公司。

现在已经出版了很多 Scala 的图书，我们真的有必要再推出第 2 版吗？市场上现存不少适合初学者的 Scala 指南，也出现了一些供高级学习者使用的图书，由 Artima 出版 Odersky 等人撰写的《Scala 编程（第 2 版）》仍被视为 Scala 语言的百科全书。

然而，本书第 2 版非常完整地描述了 Scala 语言及其生态系统，既为初学者成为 Scala 高级用户提供了所需要的指导，又关注了开发人员所面对的实用性问题，这是本书独一无二的地方，也是第 1 版广受欢迎的原因所在。

与 2009 年相比，现在有更多的机构选用了 Scala，大多数 Java 开发者也都听说过这门语言。同时，也出现了一些针对 Scala 语言的持续质疑。Scala 是不是太复杂了？既然 Java 8 已经引入了一些 Scala 特性，那还有必要使用 Scala 吗？

对于真实世界中的种种质疑，我将一一解答。我常常说，Scala 的一切，包括它的不足之处，都让我为之着迷。我希望读者阅读完本书也会有同感。

如何阅读本书

本书论述全面，初级读者无须阅读所有内容便可以使用 Scala 进行编程。本书的前 3 章“零到六十：Scala 简介”“更简洁，更强大”和“要点详解”，简要概括了 Scala 的核心语言特性。第 4 章“模式匹配”和第 5 章“隐式详解”描述了使用 Scala 编程时每天都会用到的两类基本工具，通过对这两类工具的描述将读者引领到更深的领域里。

函数式编程（FP）是一种重要的软件开发方案，假如你之前从未接触过 FP，那么阅读第 6 章能通过 Scala 学习函数式编程。紧接着第 7 章，将说明 Scala 对 for 循环的扩展，以及如何使用该扩展提供的简洁语法实现复杂而又符合规范的函数式代码。

之后，第 8 章将介绍 Scala 是如何支持面向对象编程（OOP）的。为了强调 FP 对于解决软件开发问题的重要性，我将 FP 相关章节放到 OOP 章节之前。如果将 Scala 当作“更好的面向对象的 Java”，那会较容易上手，但这样会丢掉这门语言最有力的工具。第 8 章的大多数内容在概念上很容易理解，读者将学到在 Scala 中如何定义类、构造函数等与 Java 相似的概念。

第 9 章将继续探索 Scala 的功能——使用 trait 对行为进行封装。Java 8 受到了 Scala trait 机制的影响，通过对接口进行扩展，新增了部分 trait 功能。对于这部分内容而言，即便是有经验的 Java 程序员也需要花时间理解。

接下来的 4 章，从第 10 章到第 13 章，“Scala 对象系统（I）”“Scala 对象系统（II）”“Scala 集合库”以及“可见性规则”，详细地讲解了 Scala 的对象模型和库类型。由于第 10 章包含了一些必须要尽早掌握的基本知识，因此阅读时要务必仔细。第 11 章讲述了如何正确地实现普通类型层次，你可以在第一遍阅读本书时略过这一章。第 12 章讨论了集合设计问题并提供了合理使用集合的相关信息。再重申一遍，假如你初次接触 Scala，那么请先略过此章，当你试图掌握集合类 API 的详细内容时，再回来学习。最后，第 13 章详细解释了 Scala 是如何对 Java 的 public、protected 以及 private 可见性概念进行细粒度扩展的。可以快速阅览此章。

从第 14 章开始，我们将进入更高级的主题：Scala 复杂类型。这部分内容划分为两章：第 14 章包含了 Scala 新手相对容易理解的概念，而第 15 章则讲述了更高级的内容，你可以选择以后再进行阅读。

类似地，第 16 章“高级函数式编程”讲述的内容中包括了更多高级的理论概念，例如，Monad 和仿函式（Functor）这些起源于范畴论的概念。一般水平的 Scala 开发者在最初并不需要掌握这些内容。

第 17 章“并发工具”有助于开发大型服务的程序员实现并发性的可伸缩性和可扩展性。这一章既论述了 Akka 这一基于 actor 的富并发模型，又讲述了像 Future 这类有助于编写异步代码的库类型。

第 18 章“Scala 与大数据”，通常而言，在大数据以及其他以数据为中心的计算领域里，应用 Scala 和函数式编程能够构造杀手级应用。

第 19 章“Scala 动态调用”和第 20 章“Scala 的领域特定语言”是较为高级的专题，探讨了可用于构建富领域特定语言的一些工具。

第 21 章“Scala 工具和库”讨论了一些 IDE 和第三方库。假如你是 Scala 新手，那么请阅读 IDE 和编辑器支持的相关小节，同时阅读关于 Scala 公认的项目构建工具：SBT 的相关小节。本章最后列出了可以引用的库列表。第 22 章“与 Java 的互操作”对于那些需要互用 Java 和 Scala 代码的团队而言很有帮助。

第 23 章“应用程序设计”是为架构师和软件组长而写的。我在这一章分享了自己在应用设计方面的一些观点。传统模式使用了相对较大的 JAR 文件，而这些 JAR 文件又包含了复杂的对象图谱。因此我认为这种模式是一种不良模式，需要进行变更。

最后，第 24 章“元编程：宏与反射”介绍了本书最高级的主题。当然，如果你是初学者，也可以略过这一章。

本书在附录 A 中总结了一些资料，供读者进一步阅读。

本书未涉及的内容

模块化库是 Scala 最新的 2.11 版的一大焦点，它将库文件分解成更小的 JAR 文件，这样一来，在将系统部署到空间受限的环境时（如手机设备），便能很容易移除不需要的代码。除此之外，新版移除了库中一些原本被标示为“过时”（deprecated）的包和类型，还将其他的一些包和类型标示为 deprecated，这通常是因为 Scala 不再维护这些包和类型，而且有更好的第三方的替代品。

因此，我们不会在本书中讨论那些在 2.11 版本中被标示为 deprecated 的包，具体如下。

- `scala.actors` (<http://www.scala-lang.org/api/current/scala-actors/#scala.actors.package>)
一套 actor 库。请使用 Akka actor 库。（我们将在 17.3 节对该库进行描述。）
- `scala.collection.script` (<http://www.scala-lang.org/api/current/#scala.collection.script.package>)
该库用于编写监控集合以及更新集合相关“脚本”。
- `scala.text` (<http://www.scala-lang.org/api/current/#scala.text.package>)
一套用于“格式化打印”（pretty-printing）的库。

下面列举了在 Scala 2.10 中标示为 deprecated 并已从 2.11 版移除的包。

- `scala.util.automata` (<http://www.scala-lang.org/api/2.10.4/#scala.util.automata.package>)
使用正则表达式构建确定有限自动机（DFA）。
- `scala.util.grammar` (<http://www.scala-lang.org/api/2.10.4/#scala.util.grammar.package>)
属于 parsing 库。

- `scala.util.logging` (<http://www.scala-lang.org/api/2.10.4/#scala.util.logging.package>)
推荐使用某一 JVM 平台上活跃的第三方日志库。
- `scala.util.regex` (<http://www.scala-lang.org/api/2.10.4/#scala.util.regex.package>)
对正则表达式进行句式分析。`scala.util.matching` 包同样支持正则表达式，请使用功能更为强大的 `scala.util.matching` 包。
- .NET 编译器后台
Scala 团队曾在 .NET 运行的环境之上搭建编译器后台及库。不过由于大家对这次迁移的兴趣不断衰减，因此这项工作已经暂停。

我们不会对 Scala 库中每个包和类型都进行讨论。由于篇幅和其他原因，下面这些包并不会在本书中提及。

- `scala.swing` (<http://www.scala-lang.org/api/current/scala.swing/#scala.swing.package>)
对 Java Swing 库进行封装。尽管仍然有人维护该库，但已很少有人使用它。
- `scala.util.continuations` (<http://www.scala-lang.org/files/archive/api/current/scala-continuations-library/#scala.util.continuations.package>)
编译器插件，用于生成连续传递格式 (continuation-passing style, CPS) 的代码。这是一个特殊的工具，目前很少有人使用它。
- `App` (<http://www.scala-lang.org/api/current/#scala.App>) 和 `DelayedInit` (<http://www.scala-lang.org/api/current/#scala.DelayedInit>) 特征
使用这两个类型能很方便地实现 `main` 类型 (入口类型)，它们也是 Java 类中 `static main` 方法的同义词。不过由于它们有时候会导致奇怪的行为，因此我并不推荐使用它们。我会使用通用的、符合规范的 Scala 方法编写 `main` 方法。
- `scala.ref` (<http://www.scala-lang.org/api/current/#scala.ref.package>)
对某些 Java 类型进行了封装，如 `WeakReference`，这是 `java.lang.ref.WeakReference` 的封装类。
- `scala.runtime` (<http://www.scala-lang.org/api/current/#scala.runtime.package>)
用于实现类库的类型。
- `scala.util.hashing` (<http://www.scala-lang.org/api/current/#scala.util.hashing.package>)
提供了多种散列算法。

欢迎阅读《Scala程序设计（第1版）》

一门编程语言能够流行起来是有一定原因的。有时候，某一平台的程序员会青睐于某一特定语言或平台提供商所建议的语言。大多数 Mac OS 开发者习惯使用 Objective-C，大多数 Windows 平台开发者使用 C++ 和 .NET 语言，而嵌入式系统开发者则使用 C 和 C++。

有时，语言能流行起来归功于其技术上的优势，这一优势能够使其变得时尚、让人着迷。C++、Java 和 Ruby 便曾引起程序员的狂热崇拜。

有时，语言会因为适应时代的需要而流行起来。Java 最初被视为一门能够用于编写基于浏览器的富客户端应用的完美语言。当面向对象编程变得主流时，Smalltalk 抓住了这一机遇。

现在，并发、异构型、永不停止的服务以及不断缩短的开发计划，使得业内对函数式编程越来越感兴趣。似乎面向对象编程的统治地位即将结束，而混合式编程范式将流行起来，甚至会变得不可或缺。

如今我们构建的应用大多是可靠、高性能、高并发的互联网或企业级应用程序，我们也希望会有一门通用编程语言适应这一要求，Scala 能将我们从其他语言的阵营中吸引过来，便是因为它具备了许多最理想的特性。

Scala 是一门多范式语言，同时支持面向对象和函数式编程。Scala 具有可扩展性，从小脚本到基于组件的大规模应用程序，Scala 均可胜任。Scala 是深奥的，它从全世界的计算机科学系中吸收了先进的思想。Scala 又是实用的，它的创建者 Martin Odersky 参与了多年的 Java 开发，能理解专业开发人员的需求。

Scala 简洁、优雅而又富有表现力的语法，以及提供的众多工具让我们为之着迷。本书力图阐明为什么这些特性会使 Scala 引人注目、不可或缺。

假如你是一位有经验的开发者，希望能快速全面地了解 Scala，那么这本书很适合你。你也许正在思考是否改用 Scala 或将其作为另一门补充语言；抑或你已经决定使用 Scala，需要学习并很好地掌握 Scala 的特性。无论是哪种情况，我们都希望能以一种平易近人的方式阐明这门强大的语言。

我们假设你已经很好地掌握了面向对象的编程，但并没有接触过函数式编程。我们认为你熟悉一门或多门其他的语言。我们对比了 Java、C#、Ruby 等语言的特性，如果你熟悉任何一种语言，会了解 Scala 中的相似特性，以及一些全新特性。

无论你是否具有面向对象或函数式编程的背景，都会了解到 Scala 如何优雅地融合这两种编范式，展示了它们的互补性。基于众多示例，你还能明白针对不同的设计问题如何以及何时应用 OOP 和 FP 技术。

最后，我们希望你也会为 Scala 着迷。即便 Scala 未能成为你日常使用的语言，无论你使用什么语言，我们也希望你能从 Scala 中洞察到些许知识。

排版约定

本书使用以下排版约定。

- 楷体
表示新术语。

- 等宽字体 (`constant width`)
表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体 (**`constant width bold`**)
表示应该由用户输入的命令或其他文本。
- 倾斜的等宽字体 (*`constant width italic`*)
表示应该由用户输入的值或根据上下文决定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

本书就是要帮读者解决实际问题的。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。因此，使用本书中的几段代码写成一个程序不用向我们申请许可。但是，销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无需我们授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和书号，例如：“*Programming Scala, Second Edition* by Dean Wampler and Alex Payne. Copyright 2015 Dean Wampler and Alex Payne, 978-1-491-94985-6.”

如果还有其他使用代码的情形需要与我们沟通，可以随时通过 permissions@oreilly.com 与我们联系。

获得示例代码

读者可以从 GitHub (<https://github.com/deanwampler/prog-scala-2nd-ed-code-examples>) 下载代码示例，并把下载后的文件解压到指定位置。请阅读随示例发布的 README 文件，了解如何构建和使用这些示例。（第 1 章会对相关指令进行归纳说明。）

一些示例文件可以作为脚本，使用 `scala` 命令运行，而另外一些则必须编译成 `class` 文件，还有一些文件本身就包含了故意植入的错误，无法通过编译。为了表明文件类型，我采用了某种特定的文件命名方式。实际上，在学习 Scala 的过程中，你也能从文件内容中发现文件类型。在大多数情况下，本书示例文件遵循下列命名规范。

- *.scala

这是 Scala 文件的标准文件扩展名，不过你无法从该扩展名中分辨该文件是必须使用 `scalac` 进行编译的源文件，还是可以直接使用 `scala` 运行的脚本文件，或者是本书特意植入了错误的无效代码文件。因此，本书示例代码中使用了 `.scala` 扩展名的文件必须单独经过编译才能使用，编译过程与编译 Java 代码相似。

- *.sc

以 `.sc` 后缀结尾的文件可以作为脚本文件，使用 `scala` 命令运行。例如：`scala foo.sc` 命令会执行 `foo.sc` 脚本。你还可以在解释模式下启动 `scala`，并通过 `:load` 命令加载任意脚本文件。请注意，使用 `.sc` 对脚本进行命名并不是 Scala 社区的命名标准，不过由于 SBT 构建项目时会忽略 `.sc` 文件，因此我们在此处用其对脚本进行命名。与此同时，IDE 提供的 `worksheet` 新功能将 `worksheet` 文件命名为 `.sc` 文件，我们会在第 1 章中讨论这一功能。所以使用 `.sc` 对脚本命名是一个可以接受的、便利的命名方法。再次申明，通常情况下我们使用 `.scala` 扩展名为脚本文件和代码命名。

- *.scalaX 及 *.scX

某些示例文件中特意植入了某些导致编译异常的错误。为了避免导致编译出错，这些文件使用了 `.scalaX` 或 `.scX` 扩展名。`.scalaX` 表示代码文件，而 `.scX` 则表示脚本文件。再次重申，`.scalaX` 和 `.scX` 扩展名并不是业内使用的扩展名。这些文件中也嵌入了一些注释，用于说明这些文件无法执行的原因。

Safari® Books Online



Safari Books Online (<http://www.safaribooksonline.com>) 是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice

Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和发现的问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

<http://shop.oreilly.com/product/0636920033073.do>

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

<http://www.oreilly.com>

我们在 Facebook 的地址如下：<http://facebook.com/oreilly>

请关注我们的 Twitter 动态：<http://twitter.com/oreillymedia>

我们的 YouTube 视频地址如下：<http://www.youtube.com/oreillymedia>

第2版致谢

我，Dean Wampler，在编写这一版书的过程中得到了 Typesafe 公司许多同事的指导和反馈。除此之外，一些审核了早期版本的人也给我提供了有价值的反馈，非常感谢他们。我要特别感谢 Ramnivas Laddad、Kevin Kilroy、Lutz Huehnken 和 Thomas Lockney，他们帮我审阅了本书的底稿。感谢我的老同事、老朋友 Jonas Bonér，感谢他为本书作序。

特别感谢 Ann，允许我将大量的个人时间花在本书的编写工作中，感谢你长期以来对我的包容，我爱你！

第1版致谢

我们在编写这本书的时候，一些朋友阅读了早期版本，并对本书的内容提出了大量好的意见，我们在此对这些朋友表示感谢。特别要感谢 Steve Jensen、Ramnivas Laddad、Marcel Molina、Bill Venners 和 Jonas Bonér，他们给本书提供了大量的反馈。

我们在 Safari 发布初稿以及在 <http://programmingscala.com> 网站上提供在线版本时，收到了大量的反馈。在此列举了提供反馈的读者（排名不分先后），对他们表示感谢。他们是 Iulian Dragos、Nikolaj Lindberg、Matt Hellige、David Vydra、Ricky Clarkson、Alex Cruise、Josh Cronemeyer、Tyler Jennings、Alan Supynuk、Tony Hillerson、Roger Vaughn、Arbi Sookazian、Bruce Leidl、Daniel Sobral、Eder Andres Avila、Marek Kubica、Henrik Huttunen、Bhaskar Maddala、Ged Byrne、Derek Mahar、Geoffrey Wiseman、Peter Rawsthorne、Geoffrey Wiseman、Joe Bowbeer、Alexander Battisti、Rob Dickens、Tim MacEachern、Jason Harris、Steven Grady、Bob Follek、Ariel Ortiz、Parth Malwankar、Reid Hochstedler、Jason Zaugg、Jon Hanson、Mario Gleichmann、David Gates、Zef Hemel、Michael Yee、Marius Kreis、Martin Süßkraut、Javier Vegas、Tobias Hauth、Francesco Boichicchio、Stephen Duncan Jr.、Patrik Dudits、Jan Niehusmann、Bill Burdick、David Holbrook、Shalom Deitch、Jesper Nordenberg、Esa Laine、Gleb Frank、Simon Andersson、Patrik Dudits、Chris Lewis、Julian Howarth、Dirk Kuzemczak、Henri Gerrits、John Heintz、Stuart Roebuck 以及 Jungho Kim。还有很多读者也为本书提供了反馈，不过我们只知道他们的用户名，在此我们要向 Zack、JoshG、ewilligers、abcoates、brad、teto、pjcj、mkleint、dandoyon、Arek、rue、acangiano、vkelman、bryanl、Jeff、mbaxter、pjb3、kxen、hipertracker、ctran、Ram R.、cody、Nolan、Joshua、Ajay、Joe 表示感谢。除此之外，我们还要向不知道名字的贡献者表示感谢。如果名单中漏掉了谁，我们在此表示歉意！

Mike Loukides 是我们的编辑，他深知应该如何以温和的方式催促进度。他在我们写书的过程中为我提供了巨大的帮助。O'Reilly 出版社的其他员工也总能回答我们的问题，并帮助我们继续工作。

感谢 Jonas Bonér 为本书作序。Jonas 是我在“面向切面编程”（AOP，Aspect-Oriented Programming）委员会的老朋友、老同事。这些年来，他为 Java 社区做了很多前沿性的工作。现在他将精力投入到了改进 Scala 和发展 Scala 社区的工作中。

Bill Venners 很友善地评论了本书，我们将其置于封底处。他与 Martin Odersky、Lex Spoon 一起编写了第一本 Scala 图书《Scala 编程》。对于 Scala 开发人员而言，他是不可或缺的人物。Bill 同时还创造了令人惊叹的 ScalaTest 库。

除了 Jonas 和 Bill 之外，我们还从世界各地的开发人员处学到了很多，他们是 Debasish Ghosh、James Iry、Daniel Spiewak、David Pollack、Paul Snively、Ola Bini、Daniel Sobral、Josh Suereth、Robey Pointer、Nathan Hamblen、Jorge Ortiz，以及一些通过发表博文、参与论坛讨论以及私人会话给我提供帮助的朋友。

Dean 要向 Object Mentor 公司的同事表示感谢，他同时也要感谢许多客户端开发人员。他们激发了许多编程语言、软件设计以及业内实际问题的讨论。芝加哥地区 Scala 狂热者团

体（Chicago Area Scala Enthusiasts，CASE）也为本书提供了很有价值的反馈及激励。

Alex 要向他在 Twitter 的同事表示感谢，他们对 Alex 的工作给予了鼓励，并在实际工作中很好地演示了 Scala 语言的能力。Alex 同时要对湾区 Scala 爱好者（Bay Area Scala Enthusiasts，BASE）表示感谢，他们的激情以及这个社团本身为他提供了帮助。

我们要特别感谢 Martin Odersky 和他的团队，感谢他们创造了 Scala。

零到六十：Scala简介

本章将简要说明 Scala 为何应该受到重视。之后我们将会深入学习 Scala，并动手编写一些代码。

1.1 为什么选择Scala

Scala 是一门满足现代软件工程师需求的语言；它是一门静态类型语言，支持混合范式；它也是一门运行在 JVM 之上的语言，语法简洁、优雅、灵活。Scala 拥有一套复杂的类型系统，Scala 方言既能用于编写简短的解释脚本，也能用于构建大型复杂系统。这些只是它的一部分特性，下面我们来详细说明。

- 运行在 JVM 和 JavaScript 之上的语言

Scala 不仅利用了 JVM 的高性能以及最优化性，Java 丰富的工具及类库生态系统也为其所用。不过 Scala 并不是只能运行在 JVM 之上！Scala.js (<http://www.scala-js.org>) 正在尝试将其迁移到 JavaScript 世界。

- 静态类型

在 Scala 语言中，静态类型（static typing）是构建健壮应用系统的一个工具。Scala 修正了 Java 类型系统中的一些缺陷，此外通过类型推演（type inference）也免除了大量的冗余代码。

- 混合式编程范式——面向对象编程

Scala 完全支持面向对象编程（OOP）。Scala 引入特征（trait）改进了 Java 的对象模型。trait 能通过使用混合结构（mixin composition）简洁地实现新的类型。在 Scala 中，一切都是对象，即使是数值类型。

- 混合式编程范式——函数式编程

Scala 完全支持函数式编程 (FP)，函数式编程已经被视为解决并发、大数据以及代码正确性问题的最佳工具。使用不可变值、被视为一等公民的函数、无副作用的函数、高阶函数以及函数集合，有助于编写出简洁、强大而又正确的代码。

- 复杂的类型系统

Scala 对 Java 类型系统进行了扩展，提供了更灵活的泛型以及一些有助于提高代码正确性的改进。通过使用类型推演，Scala 编写的代码能够和动态类型语言编写的代码一样精简。

- 简洁、优雅、灵活的语法

使用 Scala 之后，Java 中冗长的表达式不见了，取而代之的是简洁的 Scala 方言。Scala 提供了一些工具，这些工具可用于构建领域特定语言 (DSL)，以及对用户友好的 API 接口。

- 可扩展的架构

使用 Scala，你能编写出简短的解释性脚本，并将其粘合成大型的分布式应用。以下四种语言机制有助于提升系统的扩展性：1) 使用 trait 实现的混合结构；2) 抽象类型成员和泛型；3) 嵌套类；4) 显式自类型 (self type)。

Scala 实际上是 Scalable Language 的缩写，意为可扩展的语言。Scala 的发音为 scah-lah，像意大利语中的 staircase (楼梯)。也就是说，两个 a 的发音是一样的。

早在 2001 年，Martin Odersky 便开始设计 Scala，并在 2004 年 1 月 20 日推出了第一个公开版本 (参见 <http://article.gmane.org/gmane.comp.lang.scala/17>)。Martin 是瑞士洛桑联邦理工大学 (EPFL) 计算机与通信科学学院的一名教授。在就读研究生时，Martin 便加入了由 Niklaus Wirth¹ 领导的 PASCAL fame 项目组。Martin 曾任职于 Pizza 项目组，Pizza 是运行在 JVM 平台上早期的函数式语言。之后与 Haskell 语言设计者之一 Philip Wadler 一起转战 GJ。GJ 是一个原型系统，最终演变为 Java 泛型。Martin 还曾受雇于 Sun 公司，编写了 javac 的参考编译器，这套系统后来演化成了 JDK 中自带的 Java 编译器。

1.1.1 富有魅力的Scala

自从本书第 1 版出版之后，Scala 用户数量急剧上升，这也证实了我的观点：Scala 适应当前时代。当前我们会遇到很多技术挑战，如大数据、通过并发实现高扩展性、提供高可用并健壮的服务。Scala 语法简洁但却富有表现力，能够满足这些技术挑战。在享受 Scala 最先进的语言特性的同时，你还可以拥有成熟的 JVM、库以及生产工具给你带来的便利。

在那些需要努力才能成功的领域里，专家们往往都需要掌握复杂强大的工具和技术。也许掌握这些工具技能需要花费一些时间，但是掌握它们是你事业成功的关键，所以花费这些时间都是值得的。

注 1：PASCAL 之父。——译者注

我确信对于专家级开发者而言，Scala 就是这样一门语言。并不是所有的用户都能称得上是专家，而 Scala 却是属于技术专家的语言。Scala 包含丰富的语言特性，具有很好的性能，能够解决多种问题。虽然需要花一些时间才能掌握 Scala 语言，但是一旦你掌握了它，便不会被它束缚。

1.1.2 关于Java 8

自从 Java 5 引入泛型之后，再也没有哪次升级能比 Java 8 引入更多的特性了。现在可以使用真正的匿名函数了，我们称之为 Lambda。通过本书你将了解到这些匿名函数的巨大作用。Java 8 还改进了接口，允许为声明的方法提供默认实现。这一变化使得我们能够像使用混合结构那样使用接口，这也使接口变得更有用了，而 Scala 则是通过 trait 实现这种用法的。在 Java 8 推出之前，Scala 便已为 Java 提供了这两个被公认为 Java 8 最重要的新特性。现在是不是能说服自己切换到 Scala 了？

由于 Java 语言向后兼容的缘故，Scala 新增了一些改进，而 Java 也许永远不会包含。即便 Java 最终会拥有这些改进，那也需要漫长的等待。举例来说，较 Java 而言，Scala 能提供更为强大的类型推演、强大的模式匹配 (pattern matching) 和 for 推导式 (for comprehension)，善用模式匹配和 for 推导式能够极大地减少代码量以及类型耦合。随着深入学习，你会发现这些特性的巨大价值。

另外，一些组织对升级 JVM 设施抱有谨慎态度，这是可以理解的。对于他们而言，目前并不允许部署 Java 8 虚拟机。为了使用这些 Java 8 特性，这些组织可以在 Java 6 或 Java 7 的虚拟机上运行 Scala。

你也许因为当前使用 Java 8，就认为 Java 8 是最适合团队的选择。即便如此，本书仍然能给你传授一些有用的技术，而且这些技术可以运用在 Java 8 中。不过，我认为 Scala 具有一些额外的特性，能够让你觉得值得为之改变。

好吧，让我们开始吧！

1.2 安装Scala

为了能够尽可能快地安装并运行 Scala，本节将讲述如何安装命令行工具，使用这些工具便能运行本书列举的所有示例²。本书示例中的代码使用了 Scala 2.11.2 进行编写及编译。这也是编写本书时最新的版本。绝大多数代码无须修改便能运行在早期版本 2.10.4 上，而一些团队也仍在使用这一版本。



相较于 2.10，Scala 2.11 引入了一些新的特性，不过此次发布更侧重于整体性能的提升以及库的重构。Scala 2.10 与 2.9 版本相比，也引入了一些新的特性。也许你们部门正在使用其中的某一版本，而随着学习的深入，我们会讨论这些版本间最重要的差别。（参阅 <http://docs.scala-lang.org/scala/2.11/> 了解 2.11 版本，参阅 http://www.scala-lang.org/download/2.10.4.html#Release_Notes 了解 2.10 版本。）

注 2：第 21 章会详细讲解这些工具。

安装步骤如下。

- 安装 Java

针对 Scala 2.12 之前的版本，你可以选择 Java 6、7、8 三个版本，在安装 Scala 之前，你必须确认你的电脑上已经安装了 Java。（Scala 2.12 计划于 2016 年年初发布，该版本将只支持 Java 8。）假如你需要安装 Java，请登录 Oracle 的网站（<http://www.oracle.com/technetwork/java/javase/downloads/index.html>），遵循指示安装完整的 Java 开发工具包（JDK）。

- 安装 SBT

请遵循 [scala-sbt.org](http://www.scala-sbt.org/release/tutorial/Setup.html)（<http://www.scala-sbt.org/release/tutorial/Setup.html>）网页上的指示安装 SBT，它是业内公认的构建工具。安装完成后，便可以在 Linux、OS X 终端和 Windows 命令窗口中运行 `sbt` 命令。（你也可以选择其他的构建工具，21.2.2 节将介绍这些工具。）

- 获取本书源代码

本书前言中描述了如何下载示例代码。压缩包可以解压到你电脑中的任何文件夹。

- 运行 SBT

打开 shell 或命令行窗口，进入示例代码解压后的目录，敲入命令 `sbt test`，该命令会下载所有的依赖项，包括 Scala 编译器及第三方库，请确保网络连接正常，并耐心等待该命令执行。下载完毕后，`sbt` 会编译代码并运行单元测试。此时你能看到很多的输出信息，该命令最后会输出 `success` 信息。再次运行 `sbt test` 命令，由于该命令不需要执行任何事情，你会发现命令很快就结束了。

祝贺你！你已经真正开始了 Scala 的学习。不过，你也许会想安装其他一些有用的工具。



在学习本书的大多数时候，通过使用 SBT，你便能使用其他工具。SBT 会自动下载指定版本的 Scala 编译器、标准库以及需要的第三方资源。

不使用 SBT，也能很方便地单独下载 Scala 工具。我们会提供一些 SBT 外使用 Scala 的例子。

请遵循 Scala 官方网站（<http://www.scala-lang.org>）中的链接安装 Scala，还可以选择安装 Scaladoc。Scaladoc 是 Scala 版的 Javadoc（在 Scala 2.11 中，Scala 库和 Scaladoc 被切分为许多较小的库）。你也可以在线查阅 Scaladoc（<http://www.scala-lang.org/api/current>）。为了方便你使用，本书中出现的 Scala 库中的类型，大部分都附上了连接到 Scaladoc 页面的链接。

Scaladoc 在页面左侧类型列表上面提供了搜索栏，这有助于快速查找类型。同时，每个类型的入口处都提供了一个指向 Scala GitHub 库中对应代码的链接（<https://github.com/scala/scala>），这能很好地帮助用户学习这些库的实现。这个链接位于类型概述讨论的底部，链接所在行标注着 `Source` 字样。

你可以选用任何文本编辑器或 IDE 来处理这些示例，也可以为这些主流编辑器或 IDE 安装 Scala 支持插件。具体方法，请参见 21.3 节。通常而言，访问你所青睐的编辑器的社区，能最及时地发现 Scala 相关的支持信息。

1.2.1 使用 SBT

21.2.1 节将介绍 SBT 是如何工作的。下面，我们介绍当前需要掌握的一些基本指示。

当你启动 `sbt` 命令时，假如不指定任何任务，SBT 将启动一个交互式 REPL（REPL 是 Read、Eval、Print、Loop 的简写，代表了“读取 - 求值 - 打印 - 循环”）。下面我们将运行该命令，并尝试运行一些可用的任务。

下面列举的代码中，`$` 表示 shell 命令提示符（如 `bash` 命令提示符），你可以在该提示符下运行 `sbt` 命令；`>` 是 SBT 默认的交互提示符，可以在 `#` 符号后编写 `sbt` 注释。你可以以任意顺序输入下面列举的大多数 `sbt` 命令。

```
$ sbt
> help          # 描述命令
> tasks         # 显示最常用的、当前可用的任务
> tasks -v      # 显示所有的可用任务
> compile       # 增量编译代码
> test          # 增量编译代码，并执行测试
> clean         # 删除所有已经编译好的构建
> ~test         # 一旦有文件保存，执行增量编译并运行测试
                # 适用于任何使用了~前缀的命令
> console       # 运行Scala REPL
> run           # 执行项目的某一主程序
> show x        # 显示变量x的定义
> eclipse       # 生成Eclipse项目文件
> exit          # 退出REPL(也可以通过control-d的方式退出)
```

为了能编译更新后的代码并运行对应测试，我通常会执行 `~test` 命令。SBT 使用了增量的编译器和测试执行器，因此每次执行时不用等待完全构建所需时间。假如你希望执行其他任务或退出 `sbt`，只需要按一下回车键即可。

假如你使用安装了 Scala 插件的 Eclipse 进行开发，便能很方便地执行 `eclipse` 任务。运行 `eclipse` 任务将生成对应的项目文件，这些生成的代码作为 Eclipse 项目文件进行加载。如果你想使用 Eclipse 来处理示例代码，请执行 `eclipse` 任务。

假如你使用最近发布的 Scala 插件 IntelliJ IDEA 进行开发，直接导入 SBT 项目文件便能生成 IntelliJ 项目。

Scala 中已经包含了 REPL 环境，你可以执行 `console` 命令启动该环境。如果你希望在 REPL 环境下运行本书中的代码示例，那么通常情况下，你首先需要运行 `console` 命令：

```
$ sbt
> console
[info] Updating {file:/.../prog-scala-2nd-ed/}prog-scala-2nd-ed...
[info] ...
[info] Done updating.
[info] Compiling ...
```

```
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.2 (Java HotSpot(TM) 64-Bit Server VM, Java ...).
Type in expressions to have them evaluated .
Type :help for more information.

scala> 1 + 2
res0: Int = 3

scala> :quit
```

此处省去若干输出，与 SBT REPL 一样，你也可以使用 Ctrl-D 退出系统。

运行 `console` 时，SBT 首先会构建项目，并通过设置 `CLASSPATH` 使该项目可用。因此，你也可以使用 REPL 编写代码进行试验。



使用 Scala REPL 能有效地对你编写的代码进行试验，也可以通过 REPL 来学习 API，即便是 Java API 亦可。在 SBT 上使用 `console` 任务执行代码时，`console` 任务会很体贴地为你在 `classpath` 中添加项目依赖项以及编译后的项目代码。

1.2.2 执行Scala命令行工具

如果你单独安装了 Scala 命令行工具，会发现与 Java 编译器 `javac` 相似，Scala 编译器叫作 `scalac`。我们会使用 SBT 执行编译工作，而不会直接使用 `scalac`。不过如果你曾运行过 `javac` 命令，会发现 `scalac` 语法也很直接。

在命令行窗口中运行 `-version` 命令，便可查看到当前运行的 `scalac` 版本以及命令行参数帮助信息。与之前一样，在 `$` 提示符后输入文本。之后生成的文本便是命令输出。

```
$ scalac -version
Scala compiler version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scalac -help
Usage: scalac <options> <source files>
where possible standard options include:
  -Dproperty=value      Pass -Dproperty=value directly to the runtime system.
  -J<flag>              Pass <flag> directly to the runtime system.
  -P:<plugin>:<opt>     Pass an option to a plugin
  ...
```

与之类似，执行下列 `scala` 命令也可以查看 Scala 版本及命令参数帮助。

```
$ scala -version
Scala code runner version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scala -help
Usage: scala <options> [<script|class|object|jar> <arguments>]
      or scala -help

All options to scalac (see scalac -help) are also allowed.
...
```

有时我们会使用 `scala` 来运行 Scala “脚本” 文件，而 `java` 命令行却没有提供类似的功能。下面将要执行的脚本来源于我们的示例代码：

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

我们将调用 `scala` 命令执行该脚本。也请读者尝试运行该示例。上述代码使用的文件路径适用于 Linux 和 Mac OS X 系统。我假设，当前的工作目录位于代码示例所在的根目录。如果使用 Windows 系统，请在路径中使用反斜杠。

```
$ scala src/main/scala/progscala2/introscala/upper1.sc
ArrayBuffer(HELLO, WORLD!)
```

现在我们终于满足了编程图书或向导的一条不成文的规定：第一个程序必须打印 “Hello World!”。

最后提一下，执行 `scala` 命令时，如果未指定主程序或脚本文件，那么 `scala` 将进入 REPL 模式，这与在 `sbt` 中运行 `console` 命令类似。（不过，运行 `scala` 时的 classpath 与执行 `console` 任务的 classpath 不同。）下面列出的 REPL 会话中讲解了一些有用的命令。（如果你未独立安装 Scala，在 `sbt` 中执行 `console` 任务也能进入 Scala REPL 环境）。此时，REPL 提示符是 `scala>`（此处省略了一些输出信息）。

```
$ scala
Welcome to Scala version 2.11.2 (Java HotSpot(TM)...).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
:cp <path>          add a jar or directory to the classpath
:edit <id>|<line>    edit history
:help [command]      print this summary or command-specific help
:history [num]       show the history (optional num is commands to show)
... 其他消息

scala> val s = "Hello, World!"
s: String = Hello, World!

scala> println("Hello, World!")
Hello, World!

scala> 1 + 2
res3: Int = 3

scala> s.con<tab>
```

```
concat    contains    contentEquals
```

```
scala> s.contains("el")
res4: Boolean = true
```

```
scala> :quit
$      #返回shell提示符
```

我们为变量 `s` 赋予了 `string` 值 `"Hello, World!"`，通过使用 `val` 关键字，我们将变量 `s` 声明成不可变值。`println` 函数 ([http://www.scala-lang.org/api/current/index.html#scala.Console\\$](http://www.scala-lang.org/api/current/index.html#scala.Console$)) 将在控制台中打印一个字符串，并会在字符串结尾处打印换行符。

`println` 函数与 Java 中的 `System.out.println` (<http://docs.oracle.com/javase/8/docs/api/java/lang/System.html>) 作用一致。同样，Scala 也使用了 Java 提供的 `String` 类型 (<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>)。

接下来，请注意我们要将两个数字相加，由于我们并未将运算的结果赋予任何一个变量，因此 REPL 帮我们将变量命名为 `res3`，我们可以在随后的表达式中运用该变量。

REPL 支持 `tab` 补全。例子中显示输入命令 `s.con<tab>` 表示的是在 `s.con` 后输入 `tab` 符。REPL 将列出一组可能会被调用的方法名。在本例中表达式最后调用了 `contains` 方法。

最后，调用 `:quit` 命令退出 REPL。也可以使用 `Ctrl-D` 退出。

接下来，我们将看到更多 REPL 命令，在 21.1 节中，我们将更深入地探索 REPL 的各个命令。

1.2.3 在IDE中运行Scala REPL

下面我们将讨论另外一种执行 REPL 的方式。特别是当你使用 Eclipse、IntelliJ IDEA 或 NetBeans 时，这种方式会更加实用。Eclipse 和 IDEA 支持 `worksheet` 功能，当你编辑 Scala 代码时，感觉不到它与正常地编辑编译代码或脚本代码有什么区别。不过一旦将该文件保存，代码便会立刻被执行。因此，假如你需要修改并重新运行重要的代码片段，使用这种开发方式比使用 REPL 更为方便。NetBeans 也提供了一种类似的交互式控制台功能。

假如你想要使用上述的某个 IDE，可以参考 21.3 节，掌握 Scala 插件、`worksheet` 以及交互式控制台的相关信息。

1.3 使用Scala

在本章的剩余篇幅和之后的两章中，我们将对 Scala 的一些特性进行快速讲解。在学习这些内容时会涉及一些语言细节，这些细节仅用于理解这些内容，更多的细节会在后续章节中提供。你可以将这几章内容视为 Scala 语法入门书，并从中感受 Scala 编程的魅力。



当提到某一 Scala 库类型时，我们可以阅读 Scaladoc 中的相关信息进行学习。如果你想访问当前版本的 Scala 对应的 Scaladoc 文档，请查看 <http://www.scala-lang.org/api/current/>。请注意，左侧类型列表区域的上方有一搜索栏，应用该搜索栏能很方便地快速查找类型。与 Javadoc 不同，Scaladoc 按照 `package` 来排列类型，而不是按照字母顺序全部列出。

本书多数情况下会使用 Scala REPL，因此我们在这儿再温习一遍运行 REPL 的三种方式。你可以不指定脚本或 `main` 参数直接输入 `scala` 命令，也可以使用 `SBT console` 命令，还可以在那些流行的 IDE 中使用 `worksheet` 特性。

假如你不想使用任何 IDE，我建议你尽量使用 SBT，尤其是当你的工作固定在某一特定项目时。本书也将使用 SBT 进行讲解，这些操作步骤同样适用于直接运行 `scala` 命令或者在 IDE 中创建 `worksheet` 的情况。请自行选择开发工具。事实上，即便你青睐于使用 IDE，我还是希望你能尝试在命令行窗口运行一下 SBT，了解 SBT 环境。我个人很少使用 IDE，不过是否选择 IDE 只是个人的偏好罢了。

打开 shell 窗口，切换到代码示例所在的根文件夹并运行 `sbt`。在 `>` 提示符后输入 `console`。从现在开始，本书将省略关于 `sbt` 和 `scala` 输出的一些“程序化”的语句。

在 `scala>` 提示符中输入下列两行：

```
scala> val book = "Programming Scala"
book: java.lang.String = Programming Scala

scala> println(book)
Programming Scala
```

第一行代码中的 `val` 关键字用于声明不变变量 `book`。可变数据是错误之源，因此我推荐使用不变值。

请注意，解释器返回值列出了 `book` 变量的类型和数值。Scala 从字面量 `"Programming Scala"` 中推导出 `book` 属于 `java.lang.String` 类型 (<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>)。

显示类型信息或在声明中显式指明类型信息时，这些类型标注紧随冒号，出现在相关项之后。为什么 Scala 不遵循 Java 的习惯呢？Scala 常常能推导出类型信息，因此，我们在代码中总是看不到显式的类型标注。如果代码中省略了冒号和类型标注信息，那么与 Java 的类型习惯相比，`item: type` 这一模式更有助于编译器正确地分析代码。

一般来说，当 Scala 语法与 Java 语法存在差异时，通常都会有一个充分的理由。比如说，Scala 支持了一个新的特性，而这个特性很难使用 Java 的语法表达出来。



REPL 中显示了类型信息，这有助于学习 Scala 是如何为特定表达式推导类型的。透过这个例子，可以了解到 REPL 提供了哪些功能。

仅使用 REPL 来编辑或提交大型的示例代码会比较枯燥，而使用文本编辑器或 IDE 来编写 Scala 脚本则会方便得多。编写完成之后，你可以执行脚本，也可以复制粘贴大段代码再执行。

我们再回顾一下之前编写的 `upper1.sc` 文件。

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

本书的下载示例压缩包中的每个示例的第一行均为注释，该注释列出了示例文件在压缩包中的路径。Scala 遵循 Java、C#、C 等语言的注释规则，`// comment` 只能作用到本行行尾，而 `/* comment */` 则可以跨行。

我们再回顾一下前言中的内容。依照命名规范，脚本文件的扩展名为 `.sc`，而编译后的文件的扩展名为 `.scala`，这一命名规范仅适用于本书。通常，脚本文件往往也使用 `.scala` 扩展名。不过如果使用 SBT 构建项目，SBT 会尝试编译这些以 `scala` 命名的文件，而这些脚本文件却无法编译（我们稍后便会讲到这些）。

我们首先运行该脚本，具体代码细节稍后讨论。启动 `sbt` 并执行 `console` 命令以开启 Scala 环境。然后使用 `:load` 命令加载（编译并运行）文件：

```
scala> :load src/main/scala/progscala2/introscala/upper1.sc
Loading src/main/scala/progscala2/introscala/upper1.sc...
defined class Upper
up: Upper = Upper@4ef506bf    // 调用Java的Object.toString方法。
ArrayBuffer(HELLO, WORLD!)
```

上述脚本中，只有最后一行才是 `println` 命令的输出，其他行则是 REPL 提供的一些反馈信息。

那么这些脚本为什么无法编译呢？脚本设计的初衷是为了简化代码，无须将声明（变量和函数）封装在对象中便是一种简化。而将 Java 和 Scala 代码编译后，声明必须封装在对象中（这是 JVM 字节码的需求）。`scala` 命令通过一个聪明的技巧解决了冲突：将脚本封装在一个你看不到的匿名对象中。

假如你的确希望能将脚本文件编译为 JVM 的字节码（一组 `.class` 文件），可以在 `scalac` 命令中传入 `-Xscript <object>` 参数，`<object>` 表示你所选中的 `main` 类，它是生成的 Java 应用程序的入口点。

```
$ scalac -Xscript Upper1 src/main/scala/progscala2/introscala/upper1.sc
$ scala Upper1
ArrayBuffer(HELLO, WORLD!)
```

执行完毕后检查当前文件夹，你会发现一些命名方式有趣的 `.class` 文件。（提示：一些匿名函数也被转换成了对象！）我们稍后会再讨论这些名字，`Upper1.class` 文件中包含了主程序，我们将使用 `javap` 和 Scala 对应工具 `scalap`，对该文件实施逆向工程！

```
$ javap -cp . Upper1
Compiled from "upper1.sc"
public final class Upper1 {
```

```

    public static void main(java.lang.String[]);
  }
$ scalap -cp . Upper1
object Upper1 extends scala.AnyRef {
  def this() = { /* compiled code */ }
  def main(argv : scala.Array[scala.Predef.String]) : scala.Unit =
    { /* compiled code */ }
}

```

最后，我们将对代码本身进行讨论，代码如下：

```

// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))

```

Upper 类中的 upper 方法将输入字符串转换成大写字符串，并返回一个包含这些字符串的 Seq (Seq 表示“序列”，<http://www.scala-lang.org/api/current/index.html#scala.collection.Seq>) 对象。最后两行代码创建了 Upper 对象的一个实例，并调用这一实例将字符串“Hello”和“World!”转换为大写字符串，并最终打印出产生的 Seq 对象。

在 Scala 中定义类时需要输入 class 关键字，整个类定义体包含在最外层的一对大括号中 ({...})。事实上，这个类定义体同样也是这个类的主构造函数。假如需要将参数传递给这个构造函数，就要在类名 Upper 之后输入参数列表。

下面这小段代码声明了一个方法：

```
def upper(strings: String*): Seq[String] = ...
```

定义方法时需要先输入 def 关键字，之后输入方法名称以及可选的参数列表。再输入可选的返回类型（有时候，Scala 能够推导出返回类型），返回类型由冒号加类型表示。最后使用等于号 (=) 将方法签名和方法体分隔开。

实际上，圆括号中的参数列表代表了变长的 String 类型参数列表，修饰 strings 参数的 String 类型后面的 * 号指明了这一点。也就是说，你可以传递任意多的字符串（也可以传递空列表），而这些字符串由逗号分隔。在这个方法中，strings 参数的类型实际上是 WrappedArray (<http://www.scala-lang.org/api/current/index.html#scala.collection.mutable.WrappedArray>)，该类型对 Java 数组进行了封装。

参数列表后列出了该方法的返回类型 Seq[String]，Seq（代表 Sequence）是集合的一种抽象，你可以依照固定的顺序（不同于遍历 Set 和 Map 对象那样的随机顺序和未定义顺序，遍历那类容器无法保证遍历顺序）遍历这类结合抽象。实际上，该方法返回的类型是 scala.collection.mutable.ArrayBuffer (<http://www.scala-lang.org/api/current/#scala.collection.mutable.ArrayBuffer>)，不过绝大多数情况下，调用者无须了解这点。

值得一提的是，Seq 是一个参数化类型，就好象 Java 中的泛型类型。Seq 代表着“某类事物的序列”，上面代码中的 Seq 表示的是一个字符串序列。请注意，Scala 使用方括号 ([...]) 表示参数类型，而 Java 使用角括号 (<...>)。



Scala 的标识符，如方法名和变量名，中允许出现尖括号，例如定义“小于”方法时，该方法常被命名为 <，这在 Scala 语言中是允许的，而 Java 则不允许标识符中出现这样的字符。因此，为了避免出现歧义，Scala 使用方括号而不是尖括号表示参数化类型，并且不允许在标识符中使用方括号。

upper 方法的定义体出现在等号 (=) 之后。为什么使用等号呢？而不像 Java 那样，使用花括号表示方法体呢？

避免歧义是原因之一。当你在代码中省略分号时，Scala 能够推断出来。在大多数时候，Scala 能够推导出方法的返回类型。假如方法不接受任何参数，你还可以在方法定义中省略参数列表。

使用等号也强调了函数式编程的一个准则：值和函数是高度对齐的概念。正如我们所看到的那样，函数可以作为参数传递给其他函数，也能够返回函数，还能被赋给某一变量。这与对象的行为是一致的。

最后提一下，假如方法体仅包含一个表达式，那么 Scala 允许你省略花括号。所以说，使用等号能够避免可能的解析歧义。

函数方法体中对字符串集合调用了 map 方法 (<http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableLike>)，map 方法的输入参数为函数数字面量 (function literal)。而这些函数数字面量便是“匿名”函数。在其他语言中，它们也被称为 Lambda、闭包 (closure)、块 (block) 或过程 (proc)。Java 8 最终也提供了真正的匿名方法 Lambda。但 Java 8 之前，你只能通过接口实现的方式实现匿名方法，我们通常会在接口中定义一个匿名的内部类，并在内部类中声明执行真正工作的方法。因此，即便是在 Java 8 之前，你也能够实现匿名函数的功能：通过传入某些嵌套行为，将外部行为参数化。不过这些繁琐的语法着实损害并掩盖了匿名方法这门技术的优势。

在这个示例中，我们向 map 方法传递了下列函数数字面量：

```
(s:String) => s.toUpperCase()
```

此函数数字面量的参数表中只包含了一个字符串参数 s。它的函数体位于箭头 => 之后 (UTF8 也允许使用 =>)。该函数体调用了 s 的 UpperCase() 方法。此次调用的返回值会自动被这个函数数字面量返回。在 Scala 中，函数或方法中把最后一条表达式的返回值作为自己的返回值。尽管 Scala 中存在 return 关键字，但只能在方法中使用，上面这样的匿名函数则不允许使用。事实上，方法中也很少用到这个关键字。

方法和函数

对于大多数的面向对象编程语言而言，方法指的是类或对象中定义的函数。当调用方法时，方法中的 `this` 引用会隐性地指向某一对象。当然，在大多数的 OOP 语言中，方法调用的语法通常是 `this.method_name(other_args)`。本书中的“方法”也满足这一常用规范。我们提到的“函数”尽管不是方法，但在某些时候通常会将方法也归入函数。当前上下文能够认清它们的区别。

`upper1.sc` 中表达式 `(s:String) => s.toUpperCase()` 便是一个函数，它并不是方法。

我们对序列对象 `strings` 调用了 `map` 方法，该方法会把每个字符串依次传递给函数数字量，并将函数数字量返回的值组成一个新的集合。举个例子，假如在原先的列表中有五个元素，那么新生成的列表也将包含五个元素。

继续上面的示例，为了进一步练习代码，我们会创建一个新的 `Upper` 实例并将它赋给变量 `up`。与 Java、C# 等类似语言一样，`new Upper` 语法将创建一个新的实例。由于主构造函数并不接受任何参数，因此并不需要传递参数列表。通过 `val` 关键字，`up` 参数被声明为只读值。`up` 的行为与 Java 中的 `final` 变量相似。

最后，我们调用 `upper` 方法，并使用 `println(...)` 方法打印结果。

我们可以进一步简化代码，请思考下面更简洁的版本。

```
// src/main/scala/progscala2/introscala/upper2.sc

object Upper {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}

println(Upper.upper("Hello", "World!"))
```

这段代码同样实现了相同的功能，但使用的字符却相对较少，简洁度排名第三。

在第一行中，`Upper` 被声明为单例对象，Scala 将单例模式视为本语言的第一等级成员。尽管我们声明了一个类，不过 Scala 运行时只会创建 `Upper` 的一个实例。也就是说，你无法通过 `new` 创建 `Upper` 对象。就好像 Java 使用静态类型一样，其他语言使用类成员（class-level member），Scala 则使用对象进行处理。由于 `Upper` 中并不包含状态信息，所以我们此处的确不需要多个实例，使用单例便能满足需求。

单例模式具有一些弊端，也因此常被指责。例如在那些需要将对象值进行 `double` 的单元测试中，如果使用了单例对象，便很难替换测试值。而且如果对一个实例执行所有的计算，会引发线程安全和性能的问题。不过正如静态方法或静态值有时适用于 Java 这样的语言一样，单例有时候在 Scala 中也是适用的。上述示例便是一个证明，由于无须维护状态而且对象也不需要与外界交互，单例模式适用于上述示例。因此，使用 `Upper` 对象时我们没有必要考虑测试双倍值的问题，也没有必要担心线程安全。



Scala 为什么不支持静态类型呢？与那些允许静态成员（或类似结构）的语言相比，Scala 更信奉万物皆应为对象。相较于混入了静态成员和实例成员的语言，采用对象结构的 Scala 更坚定地贯彻了这一方针。回想一下，Java 的静态方法和静态域并未绑定到类型的实际实例中，而 Scala 的对象则是某一类型的单例。

第二行中 `upper` 的实现同样简洁。尽管 Scala 无法推断出方法的参数类型，却常常能够推断出方法的返回类型，因此我们在此省略返回类型的显式声明。同时，由于方法体中仅包含了一句表达式，我们可以省略括号，并在一行内完成整个方法的定义。除了能提示读者之外，方法体之前的等号也告诉编译器方法体的起始位置。

Scala 为什么无法推导出方法参数类型呢？理论上类型推理算法执行了局部类型推导，这意味着该推导无法作用于整个程序全局，而只能局限在某一特定域内。因此，尽管无法分辨出参数所必须使用的类型，但由于能够查看整个函数体，Scala 大多数情况下却能推导出方法的返回值类型。递归函数是个例外，由于它的执行域超越了函数体的范围，因此必须声明返回类型。

任何时候，参数列表中的返回类型都为读者提供了有用信息。仅仅是因为 Scala 能推导出函数的返回类型，我们就放弃为读者提供返回类型信息吗？对于简单的函数而言，读者能够很清楚地发现返回类型，显式列出的返回类型也许还不是特别重要。不过有时候由于 bug 或某些特定输入或函数体中的某些表达式所触发的某些微妙行为，推导出的类型可能并不是我们所期望的类型。显式返回类型代表了你所期望的返回类型，它们同时还为读者提供了有用信息，因此我推荐添加返回类型，而不要省略它们。这尤其适用于公有 API。

我们对函数数字面量进行了进一步的简化，之前我们的代码如下：

```
(s:String) => s.toUpperCase()
```

我们将其简化为下列表达式：

```
_ .toUpperCase()
```

`map` 方法接受单一函数参数，而单一函数也只接受单一参数。在这种情况下，函数体只使用一次该参数，所以我们使用占位符 `_` 来替代命名参数。也就是说：`_` 起到了匿名参数的作用，在调用 `toUpperCase` 方法之前，`_` 将被字符串替换。Scala 同时也为我们推断出了该变量的类型为 `String` 类型。

最后一行代码中，由于使用了对象而不是类，此次调用变得更加简单。无须通过 `new Upper` 代码创建实例，我们只需直接调用 `Upper` 对象的 `upper` 方法。调用语法与调用 Java 类静态方法时的语法一样。

最后，Scala 会自动加载一些像 `println` ([http://www.scala-lang.org/api/current/index.html#scala.Console\\$](http://www.scala-lang.org/api/current/index.html#scala.Console$)) 这样的 I/O 方法，`println` 方法实际是 `scala` 包 (<http://www.scala-lang.org/api/current/scala/package.html>) 中 `Console` 对象 ([http://www.scala-lang.org/api/current/index.html#scala.Console\\$](http://www.scala-lang.org/api/current/index.html#scala.Console$)) 的一个方法。与 Java 中的包一样，Scala 通过包提供“命名空间”并界定作用域。

因此，使用 `println` 方法时，我们无需调用 `scala.Console.println` 方法 ([http://www.scala-lang.org/api/current/index.html#scala.Console\\$](http://www.scala-lang.org/api/current/index.html#scala.Console$))，直接输入 `println` 即可。`println` 方法只是众多被自动加载的方法和类型中的一员，有一个叫作 `Predef` 的库对象 ([http://www.scala-lang.org/api/current/index.html#scala.Predef\\$](http://www.scala-lang.org/api/current/index.html#scala.Predef$)) 对这些自动加载的方法和类型进行定义。

我们再进行一次重构，把这个脚本转化成编译好的一个命令行工具。也就是说，我们将创建一个包含了 `main` 方法的更为经典的 JVM 应用程序。

```
// src/main/scala/progscala2/introscala/upper1.scala
package progscala2.introscala

object Upper {
  def main(args: Array[String]) = {
    args.map(_.toUpperCase()).foreach(printf("%s ", _))
    println("")
  }
}
```

回顾一下前面的内容，如果代码具有 `.scala` 扩展名，那就表示我们会使用 `scalac` 编译它。现在 `upper` 方法被改名成了 `main` 方法。由于 `Upper` 是一个对象，`main` 方法就像是 Java 类的静态 `main` 方法一样。它就是 `Upper` 应用的入口点。



在 Scala 中，`main` 方法必须为对象方法。（在 Java 中，`main` 方法必须是类静态方法。）应用程序的命令行参数将作为一组字符串传递给 `main` 方法。举例来说，输入参数是 `args: Array[String]`。

`upper1.scala` 文件中的第一行代码定义了名为 `introscala` 的包，用于装载所定义的类型。在 `Upper.main` 方法中的表达式使用了 `map` 方法的简写形式，这与我们之前代码中出现的简写形式一致。

```
args.map(_.toUpperCase())...
```

`map` 方法会返回一个新的集合。对该集合我们将使用 `foreach` 方法进行遍历。我们向 `foreach` 方法中传递另一个使用了 `_` 占位符的函数字面量。在这段代码中，集合中的每一个字符串都将作为参数传递给 `scala.Console.printf` 方法 ([http://www.scala-lang.org/api/current/index.html#scala.Console\\$](http://www.scala-lang.org/api/current/index.html#scala.Console$))，该方法也是 `Predef` 对象导入的方法，它会接受代表格式的字符串参数以及一组将嵌入到格式字符串的参数。

```
args.map(_.toUpperCase()).foreach(printf("%s ", _))
```

在此澄清一下，上述代码有两处使用了 `_`，这两个 `_` 分别位于不同的作用域中，彼此之间没有任何关联。

你需要花一些时间才能掌握这样的链式函数以及函数字面量中的一些简写方式，不过一旦熟悉了它们，你便能应用它们编写出可读性强、简洁强大的代码，这些代码能最大程度地避免使用临时变量和其他一些样板代码。如果你是一名 Java 程序员，可以想象一下使用早于 Java 8 的 Java 版本编写代码，这时你需要使用匿名内部类才能实现相同的功能。

`main` 方法的最后一行在输出中增加了一个最终换行符。

为了运行代码，你必须首先使用 `scalac`，将代码编译成一个能在 JVM 下运行的 `.class` 文件（下文中的 `$` 代表命令提示符）。

```
$ scalac src/main/scala/progscala2/introscala/upper1.scala
```

现在，你应该会看到一个名为 `progscala2/introscala` 的新文件夹，该文件夹里包含了一些 `.class` 文件，`Upper.class` 便是其中的一个文件。Scala 生成的代码必须满足 JVM 字节代码的合法性要求，文件夹目录必须与包结构吻合是要求之一。

Java 在源代码级也遵循这一规定，Scala 则要更灵活一些。请注意，在我们下载的代码示例中，文件 `Upper.class` 位于一个叫作 `IntroScala` 的文件夹中，这与它的包名并不一致。Java 同时要求必须为每一个最顶层类创建一个单独的文件，而 Scala 则允许在文件中创建任意多个类型。虽然开发 Scala 代码可以不用遵循 Java 关于源代码目录结构的规范（源代码目录结构应吻合包结构，而且为每个顶层类创建一个单独的文件），不过一些开发团队依然遵循这些规范，这主要因为他们熟悉这些 Java 规范，而且遵循这些规范有利于追踪代码位置。

现在，你可以输入任意长度的字符串参数并执行命令，如下所示：

```
$ scala -cp . progscala2.introscala.Upper Hello World!
HELLO WORLD!
```

我们通过选项 `-cp .` 将当前目录添加到查询类路径（`classpath`）中，不过本示例其实并不需要该选项。

请尝试使用其他输入参数来执行程序。另外，你可以查看 `progscala2/introscala` 文件夹中还有哪些其他的类文件，像之前例子那样使用 `javap` 或 `scalap` 命令查看这些类中包含了什么定义。

最后，由于 SBT 会帮助我们编译文件，我们实际上并不需要手动编译这些文件。在 SBT 提示符下，我们可以使用下列命令运行程序。

```
> run-main progscala2.introscala.Upper Hello World!
```

使用 `scala` 命令运行程序时，我们需要指明 SBT 生成的类文件的正确路径。

```
$ scala -cp target/scala-2.11/classes progscala2.introscala.Upper Hello World!
HELLO WORLD!
```

解释运行 Scala 与编译运行 Scala

概括地说，假如在命令行输入 `scala` 命令时不指定文件参数，REPL 将启动。在 REPL 中输入的命令、表达式和语句都会被直接执行。假如输入 `scala` 命令时指定 Scala 源文件，`scala` 命令将会以脚本的形式编译并运行文件。另外，假如你提供了 JAR 文件或是一个定义了 `main` 方法的类文件，`scala` 会像 Java 命令那样执行该文件。

我们接下来对这些代码再进行最后一次重构：


```
// src/main/scala/progscala2/introscala/upper2.scala
package progscala2.introscala

object Upper2 {
  def main(args: Array[String]) = {
    val output = args.map(_.toUpperCase()).mkString(" ")
    println(output)
  }
}
```

将输入参数映射为大写格式字符串之后，我们并没有使用 `foreach` 方法迭代并依次打印每个词，而是通过一个更便利的集合方法生成字符串。`mkString` 方法（<http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableOnce>）只接受一个输入参数，该参数指定了集合元素间的分隔符。另外一个 `mkString` 方法（重构版本）则接受三个参数，分别表示最左边的前缀字符串、分隔符和最右边的后缀字符串。你可以尝试将代码修改为使用 `mkSting("[", ", ", ", "]")`，并观察修改后代码的输出。

我们把 `mkString` 方法的输出保存到一个变量之中，再调用 `println` 方法打印这个变量。我们本可以在整个 `map` 方法之外再封装 `println` 方法进行打印，不过此处引入新变量能增强代码的可读性。

1.4 并发

Scala 有许多诱人之处，能够使用 Akka API 通过直观的 actor 模式构建健壮的并发应用便是其中之一（请参考 <http://akka.io>）。

下面的示例有些激进，不过却能让我们体会到 Scala 的强大和优雅。将 Scala 与一套直观的并发 API 相结合，便能以如此简洁优雅的方式实现并发软件。你之前研究 Scala 的一个原因可能是寻求更好的并发之道，以便更好地利用多核 CPU 和集群中的服务器来实现并发。使用 actor 并发模型便是其中的一种方法。

在 actor 并发模型中，actor 是独立的软件实体，它们之间并不共享任何可变状态信息。actor 之间无须共享信息，通过交换消息的方式便可进行通信。通过消除同步访问那些共享可变状态，编写健壮的并发应用程序变得非常简单。尽管这些 actor 也许需要修改状态，但是假如这些可变状态对外不可访问，并且 actor 框架确保 actor 相关代码调用是线程安全的，开发者就无须再费力编写枯燥而又容易出错的同步原语（`synchronization primitive`）了。

在这个简单示例中，我们会将表示几何图形的一组类的实例发送给一个 actor，该 actor 再将这组实例绘制到显示器上。你可以想象这样一个场景：渲染工厂（`rendering farm`）在为动画生成场景。一旦场景渲染完毕，构成场景的几何图形便会被发送给某一 actor 进行展示。

首先，我们将定义 `Shape` 类。

```
// src/main/scala/progscala2/introscala/shapes/Shapes.scala
package progscala2.introscala.shapes

case class Point(x: Double = 0.0, y: Double = 0.0) // ❶
```

```

abstract class Shape() {                                     // ❷
  /**
   * draw方法接受一个函数参数。每个图形对象都会将自己的字符格式传给函数f，
   * 由函数f执行绘制工作。
   */
  def draw(f: String => Unit): Unit = f(s"draw: ${this.toString}") // ❸
}

case class Circle(center: Point, radius: Double) extends Shape // ❹

case class Rectangle(lowerLeft: Point, height: Double, width: Double) // ❺
  extends Shape

case class Triangle(point1: Point, point2: Point, point3: Point) // ❻
  extends Shape

```

- ❶ 此处声明了一个表示二维点的类。
- ❷ 此处声明了一个表示几何形状的抽象类。
- ❸ 此处实现了一个“绘制”形状的 draw 方法，该方法中仅输出了一个格式化的字符串。
- ❹ Circle 类由圆心和半径组成。
- ❺ 位于左下角的点、高度和宽度这三个属性构成了矩形。为了简化问题，我们规定矩形的各条边分别与横坐标或纵坐标平行。
- ❻ 三角形由三个点所构成。

Point 类名列出的参数列表就是类构造函数参数列表。在 Scala 中，整个类主体便是这个类的构造函数，因此你能在类名之后、类主体之前列出主构造函数的参数。在本示例中，Point 类并没有类主体。由于我们在 Point 类声明的前面输入了 case 关键字，因此每一个构造函数参数都自动转化为 Point 实例的某一只读（不可变）字段。也就是说，假如要实例化一个名为 point 的 Point 实例，你可以使用 point.x 和 point.y 读取 point 的字段，但无法修改它们的值。尝试运行 point.y = 3.0 会触发编译错误。

你也可以设置参数默认值。每个参数定义后出现 = 0.0 会把 0.0 设置为该参数的默认值。因此用户无须明确给出参数值，Scala 便会推导出参数值。不过这些参数值会按照从左到右的顺序进行推导。下面我们运用 SBT 项目去进一步探索参数默认值：

```

$ sbt
...
> compile
Compiling ...
[success] Total time: 15 s, completed ...
> console
[info] Starting scala interpreter...

scala> import progscala2.intro.shapes._
import progscala2.intro.shapes._

scala> val p00 = new Point
p00: intro.shapes.Point = Point(0.0,0.0)

```

```
scala> val p20 = new Point(2.0)
p20: intro.shapes.Point = Point(2.0,0.0)

scala> val p20b = new Point(2.0)
p20b: intro.shapes.Point = Point(2.0,0.0)

scala> val p02 = new Point(y = 2.0)
p02: intro.shapes.Point = Point(0.0,2.0)

scala> p00 == p20
res0: Boolean = false

scala> p20 == p20b
res1: Boolean = true
```

因此，当我们不指定任何参数时，Scala 会使用 0.0 作为参数值。当我们只设定了一个参数值时，Scala 会把这个值赋予最左边的参数 x，而剩下的参数则使用默认值。我们还可以通过名字指定参数。对于 p02 对象，当我们想使用 x 的默认值却为 y 赋值时，可以使用 Point(y = 2.0) 的语句。

由于 Point 类并没有类主体，case 关键字的另一个特征便是让编译器自动为我们生成许多方法，其中包括了类似于 Java 语言中 String、equals 和 hashCode 方法。每个点显示的输出信息，如 Point(2.0,0.0)，其实是 toString 方法的输出。大多数开发者很难正确地实现 equals 方法和 hashCode 方法，因此自动生成这些方法具有实际的意义。

Scala 调用生成的 equals 方法，以判断 p00 == p20 和 p20 == p20b 是否成立。这与 Java 的做法不同，Java 通过比较引用是否相同来判断 == 是否成立。在 Java 中如果希望执行一次逻辑比较，你需要明确地调用 equals 方法。

现在我们要谈论 case 类的最后一个特性，编译器同时会生成一个伴生对象（companion object），伴生对象是一个与 case 类同名的单例对象（本示例中，Point 对象便是一个伴生对象）。



你可以自己定义伴生对象。任何时候只要对象名和类名相同并且定义在同一个文件中，这些对象就能称作伴生对象。

随后可以看到，我们可以在伴生对象中添加方法。不过伴生对象中已经自动添加了不少方法，apply 方法便是其中之一。该方法接受的参数列表与构造函数接受的参数列表一致。

任何时候只要你在输入对象后紧接着输入一个参数列表，Scala 就会查找并调用该对象的 apply 方法，这也意味着下面两行代码是等价的。

```
val p1 = Point.apply(1.0, 2.0)
val p2 = Point(1.0, 2.0)
```

如果对象中未定义 apply 方法，系统将抛出编译错误。与此同时，输入参数必须与预期输入相符。

`Point.apply` 方法实际上是构建 `Point` 对象的工厂方法，它的行为很简单；调用该方法就好像是不通过 `new` 关键字调用 `Point` 的构造函数一样。伴生对象其实与下列代码生成的对象无异。

```
object Point {  
  def apply(x: Double = 0.0, y: Double = 0.0) = new Point(x, y)  
  ...  
}
```

不过，伴生对象 `apply` 方法也可以用于决定相对复杂的类继承结构。父类对象需判断参数列表与哪个数据类型最为吻合，并依此选择实例化的子类型。比方说，某一数据类型必须分别为元素数量少的情况和元素数量多的情况各提供一个不同的最佳实现，此时选用工厂方法可以屏蔽这一逻辑，为用户提供统一的接口。



紧挨着对象名输入参数列表时，Scala 会查找并调用匹配该参数列表的 `apply` 方法。换句话说，Scala 会猜想该对象定义了 `apply` 方法。从句法角度上说，任何包含了 `apply` 方法的对象的行为都很像函数。

在伴生对象中安置 `apply` 方法是 Scala 为相关类定义工厂方法的一个便利写法。在类中定义而不是在对象中定义的 `apply` 方法适用于该类的实例。例如，调用 `Seq.apply(index: Int)` 方法将获得序列中指定位置的元素（从 0 开始计数）。

`Shape` 是一个抽象类。在 Java 中我们无法实例化一个抽象类，即使该抽象类中没有抽象成员。该类定义了 `Shape.draw` 方法，不过我们只希望能够实例化具体的形状：圆形、矩形或三角形。

请注意传给 `draw` 方法的参数，该参数是一个类型为 `String => Unit` 的函数。也就是说，函数 `f` 接受字符串参数输入并返回 `Unit` 类型。`Unit` 是一个实际存在的类型，它的表现却与 Java 中的 `void` 类型相似。在函数式编程中，大家将 `void` 类型称为 `Unit` 类型。

具体做法是 `draw` 方法的调用者将传入一个函数，该函数会接受表示具体形状的字符串，并执行实际的绘图工作。



假如某函数返回 `Unit` 对象，那么该函数肯定是有副作用的。`Unit` 对象没有任何作用，因此该函数只能对某些状态产生副作用。副作用可能会造成全局范围的影响，比如执行一次输入或输出操作（I/O），也可能只会影响某些局部对象。

通常在函数式编程中，人们更青睐于那些没有任何副作用的纯函数，这些纯函数的返回值便是它们的工作成果。纯函数容易阐述、易于测试，也很方便重用，而副作用往往是错误之源。不过最起码现实中的程序离不开 I/O。

`Shape.draw` 阐明了这样一个观点：与 `Strings`、`Ints`、`Points` 和其他对象无异，函数也是第一等级的值。和其他值一样，我们可以将函数赋给变量，将函数作为参数传递给其他函数，就好像 `draw` 方法一样。函数还能作为其他函数的返回值。我们将利用函数这一特性构

建可组合并且灵活的软件。

假如某函数接受其他函数参数并返回函数，我们称之为高阶函数（higher-order function, HOF）。

我们可以认为 `draw` 方法定义了一个所有形状类都必须支持的协议，而用户可以自定义这个协议的实现。各个形状类可以通过 `toString` 方法决定如何将状态信息序列化为字符串。`draw` 方法会调用 `f` 函数，而 `f` 函数通过 Scala 2.10 引入的新特性插值字符串（interpolated string）构建了最终的字符串。



如果你忘了在“插值字符串”前输入 `s` 字符，`draw: ${this.toString}` 将原封不动地返回给你。也就是说，字符串不会被篡改。

`Circle`、`Rectangle` 和 `Triangle` 类都是 `Shape` 类的具体子类。这些类并没有类主体，这是因为 `case` 关键字为它们定义好了所有必须的方法，如 `Shape.draw` 所需要的 `toString` 方法。

为了简化问题，我们规定矩形的各条边平行于 x 或 y 轴。因此，我们使用一个点（左侧最低点即可）、矩形的高度和宽度便能描述矩阵。而 `Triangle` 类（三角形）的构造函数则接受三个 `Pointer` 对象参数。

在简化后的程序中，传递给 `draw` 方法的 `f` 函数只会在控制台中输出一条字符串，不过你也许有机会构建一个真实的图形程序，该程序将使用 `f` 函数将图形绘制到显示器上。

既然已经定义好了形状类型，我们便可以回到 `actor` 上。其中，`Typesafe` (<http://typesafe.com>) 贡献的 `Akka` 类库 (<http://akka.io>) 会被使用到。项目文件 `build.sbt` 中已经将该类库设定为项目依赖项。

下面列出 `ShapesDrawingActor` 类的实现代码：

```
// src/main/scala/progscala2/introscala/shapes/ShapesDrawingActor.scala
package progscala2.introscala.shapes

object Messages {                                     // ❶
  object Exit                                         // ❷
  object Finished
  case class Response(message: String)                // ❸
}

import akka.actor.Actor                               // ❹

class ShapesDrawingActor extends Actor {              // ❺
  import Messages._                                   // ❻
  def receive = {                                     // ❼
    case s: Shape =>
      s.draw(str => println(s"ShapesDrawingActor: $str"))
```

```

        sender ! Response(s"ShapesDrawingActor: $s drawn")
    case Exit =>
        println(s"ShapesDrawingActor: exiting...")
        sender ! Finished
    case unexpected => // default. Equivalent to "unexpected: Any"
        val response = Response(s"ERROR: Unknown message: $unexpected")
        println(s"ShapesDrawingActor: $response")
        sender ! response
    }
}

```

- ❶ 此处声明了对象 Messages，该对象定义了大多数 actor 之间进行通信的消息。这些消息就好像信号量一样，触发了彼此的行为。将这些消息封装在一个对象中是一个常见的封装方式。
- ❷ Exit 和 Finished 对象中不包含任何状态，它们起到了标志的作用。
- ❸ 当接收到发送者发送的消息后，模板类（case class）Response 会随意构造字符串消息，并将消息返回给发送者。
- ❹ 导入 akka.actor.Actor 类型（<http://doc.akka.io/api/akka/current/#akka.actor.Actor>）。Actor 类型是一个抽象基类，我们将继承该类定义 actor。
- ❺ 此处定义了一个 actor 类，用于绘制图形。
- ❻ 此处导入了 Messages 对象中定义的三个消息。Scala 支持嵌套导入（nesting import），嵌套导入会限定这些值的作用域。
- ❼ 此处实现了抽象方法 Actor.receive。该方法是 Actor 的子类必须实现的方法，定义了如何处理接收到的消息。

包括 Akka 在内的大多数 actor 系统中，每一个 actor 都会有一个关联邮箱（mailbox）。关联邮箱中存储着大量消息，而这些消息只有经过 actor 处理后才会被提取。Akka 确保了消息处理的顺序与接收顺序相同，而对于那些正在被处理的消息，Akka 保证不会有其他线程抢占该消息。因此，使用 Akka 编写的消息处理代码天生具有线程安全的特性。

需要注意的是，Akka 支持一种奇特的 receive 方法实现方式。该实现不接受任何参数，而实现体中也只包含了一组由 case 关键字开头的表达式。

```

def receive = {
  case first_pattern =>
    first_pattern_expressions
  case second_pattern =>
    second_pattern_expressions
}

```

偏函数（PartialFunction，<http://www.scala-lang.org/api/current/#scala.PartialFunction>）是一类较为特殊的函数，上述函数体所用的语法就是典型的偏函数语法。偏函数实际类型是 PartialFunction[Any,Unit]，这说明偏函数接受单一的 Any 类型参数并返回 Unit 值。Any 是 Scala 类层次级别的根类，因此该函数可以接受任何参数。由于该函数返回 Unit 对象，因此函数体一定会产生副作用。由于 actor 系统采用了异步消息机制，它必须依靠副作用。通常情况下由于传递消息后无法返回任何信息，我们的代码块中便会发送一些其他消息，

包括给发送者的返回信息。

偏函数中仅包含了一些 `case` 子句，这些子句会对传递给函数的消息执行模式匹配。代码中并没有任何表示消息的函数参数，内部实现需要处理这些消息。

当匹配上某一模式时，系统将执行从箭头符 (`=>`) 到下一个 `case` 子句（也有可能是函数结尾处）之间的表达式。由于箭头符和下一个 `case` 关键字能够无误地标识代码区间，因此无须使用大括号包住表达式。另外，假如 `case` 关键字后只有一句简短的表达式，可以不用换行，直接将表达式放在箭头后面。

尽管听上去挺复杂，实际上偏函数是一个简单的概念。单参数函数会接受某一类型的输入值并返回相同或不同类型的值。而选用偏函数相当于明确地告诉其他人：“我也许无法处理所有你输入给我的值。”除法 x/y 是数学上的一个经典偏函数例子，当分母 y 为 0 时， x/y 的值是不确定的。因此，除法是一个偏函数。

`receive` 方法会尝试将接收到的各条消息与这三个模式匹配表达式进行匹配，并执行最先被匹配上的表达式。接下来我们对 `receive` 方法进行分解。

```
def receive = {  
  case s: Shape =>                                // ❶  
    ...  
  case Exit =>                                    // ❷  
    ...  
  case unexpected =>                              // ❸  
    ...  
}
```

- ❶ 如果收到的信息是 `Shape` 的一个实例，那说明该消息匹配了第一条 `case` 子句。我们也会将 `Shape` 对象引用赋给变量 `s`。也就是说，虽然输入消息的类型为 `Any`，但 `s` 类型却是 `Shape`。
- ❷ 判断消息是否为 `Exit` 消息体。`Exit` 消息用于标识已经完成。
- ❸ 这是一条“默认”子句，可以匹配任何输入。该子句等同于 `unexpected: Any` 子句，对于那些未能与前两个子句模式匹配的任何输入，该子句都会匹配。而变量 `unexpected` 会被赋予消息值。

最后一条匹配规则能匹配任何消息，因此该规则必须放到最后一位。假如你尝试将其放置到某些规则之前，你将看到 `unreachable code` 的错误信息。这是因为这些后续的 `case` 表达式不可访问。

值得注意的是，由于我们添加了“默认”子句，这个“偏”函数其实变成了“完整的”，这意味着该函数能正确处理任何输入。

下面让我们查看每个匹配点调用的表达式：

```
def receive = {  
  case s: Shape =>  
    s.draw(str => println(s"ShapesDrawingActor: $str")) // ❶  
    sender ! Response(s"ShapesDrawingActor: $s drawn") // ❷  
  case Exit =>  
    println(s"ShapesDrawingActor: exiting...")          // ❸
```

```

        sender ! Finished // ❹
    case unexpected =>
        val response = Response(s"ERROR: Unknown message: $unexpected") // ❺
        println(s"ShapesDrawingActor: $response")
        sender ! response // ❻
    }

```

- ❶ 调用了形状 `s` 的 `draw` 方法并传入一个匿名函数，该匿名函数了解如何处理 `draw` 方法生成的字符串。在这段代码中，此匿名函数仅打印了生成的字符串。
- ❷ 向“发信方”回复了一个消息。
- ❸ 打印了一条表示正在退出的消息。
- ❹ 向“发信方”发送了一条结束信息。
- ❺ 根据错误信息生成 `Response` 对象，并打印错误信息。
- ❻ 向“发信方”回复了这条信息。

代码 `sender ! Response(s"ShapesDrawingActor: $s drawn")` 创建了回复信息，并将该信息发送给了 `shape` 对象的发送方。`Actor.sender` 函数返回了 `actor` 发送消息接收方的对象引用，而 `!` 方法则用于发送异步消息。是的，`!` 是一个方法名，使用 `!` 遵循了之前 Erlang 的消息发送规范，值得一提的是，Erlang 是一门推广 `actor` 模型的语言。

我们也可以在 Scala 允许范围内使用一些语法糖。下面两行代码是等价的：

```

sender ! Response(s"ShapesDrawingActor: $s drawn")
sender.!(Response(s"ShapesDrawingActor: $s drawn"))

```

假如某一方法只接受单一参数，你可以省略掉对象后的点号和参数周边的括号。请注意，第一行代码看起来更清晰，这也是 Scala 支持这种语法的原因。表示法 `sender ! Response` 被称为中置表示法，这是因为操作符 `!` 位于对象和参数中间。



Scala 的方法名可以是操作符。调用接受单一参数的方法时可以省略对象后的点号和参数周边的括号。不过有时候省略它们会导致解析二义性，这时你需要保留点号或保留括号，有时候两者都需要保留。

在进入最后一个 `actor` 之前还有最后一个值得注意的地方。使用面向对象编程时，有一条经常被人提及的原则：永远不要在 `case` 语句上进行类型匹配。这是因为如果继承层次结构发生了变化，`case` 表达式也会失效。作为替代方案，你应该使用多态函数。这是不是意味着我们之前讨论的模式匹配代码只是一个反模式呢？

回顾一下，我们之前定义的 `Shape.draw` 方法调用了 `Shape` 类的 `toString` 方法，由于 `Shape` 类的那些子类是 `case` 类，因此这些子类中实现了 `toString` 方法。第一个 `case` 语句中的代码调用了多态的 `toString` 操作，而我們也没有与 `Shape` 的某一具体子类进行匹配。这意味着即便修改了 `Shape` 类层次结构，我们的代码也不会失效。其他的 `case` 子句所匹配的条件也与类层次无关，即便这些条件真会发生变化，变化也不会频繁。

由此，我们将面向对象编程中的多态与函数式编程中的劳模——模式匹配结合到了一起。

这是 Scala 优雅地集成这两种编程范式的方式之一。

模式匹配与子类型多态

模式匹配在函数式编程中扮演了重要的角色，而子类型多态（即重写子类型中的方法）在面向对象编程的世界中同样不可或缺。函数式编程中的模式匹配的重要性和复杂度都要远超过大多数命令式语言中对应的 switch/case 语句。我们将在第 4 章深入探讨模式匹配。在此处的示例中，我们开始了解到函数风格的模式匹配和多态调度之间的结合会产生强大的组合效果，而这也是像 Scala 这样的混合范式语言能提供的一大益处。

最后，我将列出运行此示例的 ShapesDrawingDriver 对象的代码：

```
// src/main/scala/progscala2/introscala/shapes/ShapesActorDriver.scala
package progscala2.introscala.shapes
import akka.actor.{Props, Actor, ActorRef, ActorSystem}
import com.typesafe.config.ConfigFactory

// 仅用于本文件的消息：
private object Start // ❶

object ShapesDrawingDriver { // ❷
  def main(args: Array[String]) { // ❸
    val system = ActorSystem("DrawingActorSystem", ConfigFactory.load())
    val drawer = system.actorOf(
      Props(new ShapesDrawingActor), "drawingActor")
    val driver = system.actorOf(
      Props(new ShapesDrawingDriver(drawer)), "drawingService")
    driver ! Start // ❹
  }
}

class ShapesDrawingDriver(drawerActor: ActorRef) extends Actor { // ❺
  import Messages._

  def receive = {
    case Start => // ❻
      drawerActor ! Circle(Point(0.0,0.0), 1.0)
      drawerActor ! Rectangle(Point(0.0,0.0), 2, 5)
      drawerActor ! 3.14159
      drawerActor ! Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))
      drawerActor ! Exit
    case Finished => // ❼
      println(s"ShapesDrawingDriver: cleaning up...")
      context.system.shutdown()
    case response: Response => // ❸
      println("ShapesDrawingDriver: Response = " + response)
    case unexpected => // ❹
      println("ShapesDrawingDriver: ERROR: Received an unexpected message = "
        + unexpected)
  }
}
```

- ❶ 定义仅用于本文件的消息（私有消息），该消息用于启动。使用一个特殊的开始消息是一个普遍的做法。
- ❷ 定义“驱动”actor。
- ❸ 定义了用于驱动应用的主方法。主方法先后构建了一个 `akka.actor.ActorSystem` 对象 (<http://doc.akka.io/api/akka/current/#akka.actor.ActorSystem>) 和两个 actor 对象：我们之前讨论过的 `ShapesDrawingActor` 对象和即将讲解的 `ShapesDrawingDriver` 对象。我们暂时先不讨论设置 Akka 的方法，在 17.3 节将详细讲述。现在只需要知道我们把 `ShapesDrawingActor` 对象传递给了 `ShapesDrawingDriver` 即可，事实上我们向 `ShapesDrawingDriver` 对象传递的对象属于 `akka.actor.ActorRef` 类型 (<http://doc.akka.io/api/akka/current/#akka.actor.ActorRef>，actor 的引用类型，指向实际的 actor 实例)。
- ❹ 向驱动对象发送 `Start` 命令，启动应用！
- ❺ 定义了 actor 类：`ShapesDrawingDriver`。
- ❻ 当 `receive` 方法接收到 `Start` 消息时，它将向 `ShapesDrawingActor` 发送五个异步消息：包含了三个形状类对象，`Pi` 值（将被视为错误信息）和 `Exit` 消息。从这能看出，这是一个生命周期很短的 actor 系统！
- ❼ 假如 `ShapesDrawingDriver` 发送 `Exit` 消息后接收到了返回的 `Finished` 消息（请回忆一下 `ShapesDrawingActor` 类处理 `Exit` 消息的逻辑），那么我们将访问 `Actor` 类提供的 `context` 字段来关闭 actor 系统。
- ❽ 简单地打印出其他错误的回复信息。
- ❾ 与之前所见的默认子句一样，该子句用于处理预料之外的消息。

让我们尝试运行该程序！在 `sbt` 提示符后输入 `run`，`sbt` 将按需编译代码并列出了所有定义了 `main` 方法的代码示例程序：

```
> run
[info] Compiling ...

Multiple main classes detected, select one to run:

[1] progscala2.introscala.shapes.ShapesDrawingDriver
...

Enter number:
```

输入数字 **1**，之后我们便能看到下列输出（为了方便显示，已对输出内容进行排版）：

```
...
Enter number: 1

[info] Running progscala2.introscala.shapes.ShapesDrawingDriver
ShapesDrawingActor: draw: Circle(Point(0.0,0.0),1.0)
ShapesDrawingActor: draw: Rectangle(Point(0.0,0.0),2.0,5.0)
ShapesDrawingActor: Response(ERROR: Unknown message: 3.14159)
ShapesDrawingActor: draw: Triangle(
  Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0))
ShapesDrawingActor: exiting...
```

```
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Circle(Point(0.0,0.0),1.0) drawn)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Rectangle(Point(0.0,0.0),2.0,5.0) drawn)
ShapesDrawingDriver: Response = Response(ERROR: Unknown message: 3.14159)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Triangle(
    Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0)) drawn)
ShapesDrawingDriver: cleaning up...
[success] Total time: 10 s, completed Aug 2, 2014 7:45:07 PM
>
```

由于所有的消息都是以异步的方式发送的，你可以看到驱动 actor 和绘图 actor 的消息交织在一起。不过处理消息的顺序与发送消息的顺序相同。运行多次应用程序，你会发现每次输出都会不同。

到现在为止，我们已经尝试了基于 actor 的并发编程，同时也掌握了一些很有威力的 Scala 特性。

1.5 本章回顾与下一章提要

我们首先介绍了 Scala，之后分析了一些重要的 Scala 代码，其中包含一些 Akka 的 actor 并发库相关代码。

你在学习 Scala 的过程中，也可以访问 <http://scala-lang.org> 网站获取其他一些有用的资源。在该网站上，能找到一些指向 Scala 类库、教程以及一些描述这门语言特性相关文章的链接。

Typesafe 是一家为 Scala 以及包括 Akka (<http://akka.io>)、Play (<http://www.playframework.com>) 在内的许多基于 JVM 的开发工具和框架提供支持的商业公司。在该公司的网站上 (<http://typesafe.com>) 也能找到一些有用的资源。尤其是 Typesafe Activator 工具 (<http://typesafe.com/activator>)，该工具会根据不同类型的 Scala 或 Java 应用程序模版，执行分析、下载和构建工作。Typesafe 公司还提供了订购支持、咨询及培训服务。

在后续的部分，我们将继续介绍 Scala 的特性，着重介绍如何使用 Scala 简洁有效地完成工作。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈 ……



—— QQ联系我们 ——

图灵读者官方群I: 218139230

图灵读者官方群II: 164939616



—— 微博联系我们 ——

官方账号: @图灵教育 @图灵社区 @图灵新知

市场合作: @图灵袁野

写作本版书: @图灵小花 @图灵张霞 @毛倩倩-图灵

翻译英文书: @朱巍ituring @楼伟珊

翻译日文书或文章: @图灵乐馨

翻译韩文书: @图灵陈曦

电子书合作: @hi_jeanne

图灵访谈/《码农》杂志: @李盼ituring

加入我们: @王子是好人



—— 微信联系我们 ——



图灵教育
turingbooks



图灵访谈
ituring_interview

Scala 程序设计(第2版)

Scala具备现代对象模型、函数式编程以及先进类型系统的所有优势，是一门可以满足现代软件工程师需求的语言。本书通过大量的代码示例，向读者全面展示了在Scala语言生态环境下如何高效地编写代码，同时阐明了Scala是目前编写高扩展性和以数据为中心的应用软件的最佳语言。

在第1版的基础之上，第2版介绍了Scala的最新语言特性，新添了模式匹配、推导式以及高级函数式编程等知识。通过本书，读者还能学会如何使用Scala命令行工具、第三方工具、库以及适用于编辑器和IDE的Scala相关插件。本书既适合Scala初学者入门，也适合经验丰富的Scala开发者参考。

通过阅读本书，你可以：

- 利用Scala简洁灵活的语法，提高编程效率；
- 深入学习函数式编程的基本技能和高级技能；
- 使用Scala函数式组合器，构造“杀手级”大数据应用；
- 使用Scala提供的trait类型实现mixin组合，使用模式匹配实现数据抽取功能；
- 学习Scala语言中复杂的类型系统，了解函数式编程和面向对象编程中的概念；
- 深入学习包括Akka的Scala并发工具；
- 掌握如何开发丰富的领域特定语言。

作为一本强调数据科学的图书，本书中出现的代码示例均保存在公开的Github仓库中。通过立即可启动的虚拟机，这些示例代码可以很容易地获得。该虚拟机中预装了一组IPython Notebook，为我们提供方便的交互式学习环境。

Dean Wampler博士，Typesafe公司的大数据架构师。Typesafe使用Scala、函数式编程、Spark、Hadoop以及Akka技术编写数据处理与分析的工具和服务。Dean是《面向Java开发者的函数式编程》的作者，同时也与他人合著了《Hive编程指南》一书。

Alex Payne是Twitter的平台组长。在Alex开发的服务基础上，其他的程序开发者构造了一套备受欢迎的社交消息服务。此前，Alex还为政治竞选、公益性组织以及初创企业编写过一些Web应用。

SCALA/JAVA/PROGRAMMING LANGUAGES

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机 / 程序设计 / Scala

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc. 授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-41681-0



ISBN 978-7-115-41681-0

定价：109.00元

欢迎加入 图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn