华章科技
HZ BOOKS
PEARSON

DSL领域的丰碑之作，软件开发“教父”Martin Fowler历时多年的心血结晶，ThoughtWorks中国翻译。

全面详尽地讲解各种DSL及其构造方式，揭示与编程语言无关的通用原则和模式，阐释如何通过DSL有效提高开发人员的生产力以及增进与领域专家的有效沟通。

华章程序员书库

Domain-Specific Languages

领域特定语言

（英）Martin Fowler 著
ThoughtWorks 中国 译

机械工业出版社
China Machine Press

华章程序员书库

Domain-Specific Languages

领域特定语言

（英）Martin Fowler 著
ThoughtWorks 中国 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

领域特定语言 /（英）福勒（Fowler，M.）著；ThoughtWorks 中国译 . —北京：机械工业出版社，2013.2
（华章程序员书库）
书名原文：Domain-Specific Languages

ISBN 978-7-111-41305-9

I. 领… II. ① 福… ② T… III. 程序语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2013）第 018068 号

本书是 DSL 领域的丰碑之作，由世界级软件开发大师和软件开发“教父”Martin Fowler 历时多年写作而成，ThoughtWorks 中国翻译。全面详尽地讲解了各种 DSL 及其构造方式，揭示了与编程语言无关的通用原则和模式，阐释了如何通过 DSL 有效提高开发人员的生产力以及增进与领域专家的有效沟通，能为开发人员选择和使用 DSL 提供有效的决策依据和指导方法。

全书共 57 章，分为六个部分：第一部分介绍了什么是 DSL，DSL 的用途，如何实现外部 DS 和内部 DSL，如何生成代码，语言工作台的使用方法；第二部分介绍了各种 DSL，分别讲述了语义模型、符号表、语境变量、构造型生成器、宏和通知的工作原理和使用场景；第三部分分别揭示分隔符指导翻译、语法指导翻译、BNF、易于正则表达式表的词法分析器、递归下降法词法分析器、解析器组合子、解析器生成器、树的构建、嵌入式语法翻译、内嵌解释器、外加代码等；第四部分介绍了表达式生成器、函数序列、嵌套函数、方法级联、对象范围、闭包、嵌套闭包、标注、解析数操作、类符号表、文本润色、字面量扩展的工作原理和使用场景；第五部分介绍了适应性模型、决策表、依赖网络、产生式规则系统、状态机等计算模型的工作原理和使用场景；第六部分介绍了基于转换器的代码生成、模板化的生成器、嵌入助手、基于模型的代码生成、无视模型的代码生成和代沟等内容。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：谢晓芳
三河市杨庄长鸣印刷装订厂印刷
2013 年 3 月第 1 版第 1 次印刷
186mm×240mm · 30.5 印张
标准书号：ISBN 978-7-111-41305-9
定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

译 者 序

2008 年，老马（Martin Fowler）在 Agile China 上做主旨发言，题目就是领域特定语言（Domain Specific Language，DSL）。老马提携后辈，愿意跟我合作完成这个演讲。而我呢，一方面，年少轻狂认为这个领域我也算个中好手，另一方面，也感激老马的信任和厚爱，就答应了。当时我已经知道老马在写一本关于这个主题的书，便跟他讨要原文来看。当时还没有成型的稿子，只有非常简略的草稿和博客片段。

2010 年年底，ThoughtWorks 技术战略委员会（ThoughtWorks Technology Advisor Board）在芝加哥开会。那时候，这本书的英文原版已然出版。晚上聚餐的时候，我心血来潮，跟老马说如果有机会，希望能将这本书翻译成中文，介绍给中国的开发者。老马听了很高兴，把最终定稿的电子版访问权限授予了我。

几个月后，同事刀哥（李剑）问我有没有兴趣参加这本书的翻译，我说当然有。后来回想起来，我还是上了刀哥的当，因为突然之间我就从参与翻译变成负责翻译了。

由于我手里已经有原稿的缘故，因此我们没有采用出版社提供的 Word 版本，而是在英文原稿上直接翻译。原稿除了文字部分之外，还有几千行代码。其中包括 XSTL、Ruby、C++、Java、图像处理脚本，甚至还有用于构建样书的 rake 脚本。惊讶之余，作为程序员的求知欲也被激发出来了。在动手翻译之前，我们花了两天彻底了解这些代码的作用。然后就发现了这个惊人的秘密：

这是一本用领域特定语言写就的关于领域特定语言的书。

原文的文字部分是老马在 docbook 基础上定义的领域特定语言。这种语言除了在 docbook 的基础结构上定义了章节模板之外，还有两个专用结构：patternRef 和 codeRef 。

patternRef 用于处理模式名称在不同章节中的引用。本书分为两部分，前面的概念讲述部分和后面的模式部分。它耗时 3 年写就，在草稿阶段模式名称都未确定，各章节之间交叉引用很多。一旦出现模式名称改变，更新同步成本就很高。为此，老马定义了专有的语言结构，patternRef。所有对于模式的引用，都通过 patternRef 实现。由 patternRef 解析处理应该使用那个具体的名称。

这个巧妙的做法在后来的翻译中给我们带来了很大的困扰。因为 patternRef 会处理英语中的单复数，而中文不会有这样的情况。翻译稿中出现了大量的 s 和 es。最后还是通过修改 DSL 解析器里才解决了这个问题。

codeRef 则表示代码引用。这本书属于技术领域，其中会有大量代码示例；同一份代码示例会在不同章节中引用，一旦写法变化，就需要同步检查它在上下文内是否还能起到

示范作用。老马先在示范代码的源代码中通过注释加入 XML 标注，把代码分解成一段段可引用的例子。因为是代码注释，所以不会影响源代码的编译、调试和重构。然后，再通过 codeRef，表明是哪个例子的哪段示例。最后，再通过 Ruby 和 XSLT，摘取对应的代码段，生成相应的文本。

我一直认为在澄清概念和发现模式上老马是有超能力的。通常会忘记他也是个 ThoughtWorker，而让做事情变得有趣，则是每个 ThoughtWorker 都有的超能力。

翻译这本书并不轻松，其中很多概念中文并无定译。为了呈现最好的结果，我们成立了一个翻译小组，包括熊节、郑烨、李剑、张凯峰、金明等有较多翻译经验的 ThoughtWorker 悉数在内。虽然如此，仍然难免疏漏，望读者不吝斧正。

徐昊

ThoughtWorks 中国

前　言

在我开始编程之前，DSL（Domain–Specific Language，领域特定语言）就已经成了程序世界中的一员。随便找个 UNIX 或者 Lisp 老手问问，他一定会跟你滔滔不绝地谈起 DSL 是怎么成为他的镇宅之宝的，直到你被烦得痛不欲生为止。但即便这样，DSL 却从未成为计算领域的一大亮点。大多数人都是从别人那里学到 DSL，而且只学到了有限的几种技术。

我写这本书就是为了改变这个现状。我希望通过本书介绍的大量 DSL 技术，让你有足够的信息来做出决策：是否在工作中使用 DSL，以及选择哪一种 DSL 技术。

造成 DSL 流行的原因有很多，我只着重强调两点：首先，提升开发人员的生产力；其次，增进与领域专家之间的沟通。如果 DSL 选择得当，就可以使一段复杂的代码变得清晰易懂，在使用这段代码时提高程序员的工作效率。同时，如果 DSL 选择得当，就可以使一段普通的文字既可以当做可执行的软件，又可以充当功能描述，让领域专家能理解他们的想法是如何在系统中得到体现的，开发者和领域专家的沟通也会更加顺畅。增进沟通比起工作效率提升困难了一些，但带来的效果却更为显著。因为它可以帮助我们打通软件开发中最狭窄的瓶颈——程序员和客户之间的沟通。

我不会片面夸大 DSL 的价值。我常常说，无论你什么时候谈到 DSL 的优缺点，你都可以考虑把“DSL”换成“库”。实 际上，大多数 DSL 都只是在一个框架或者库上又加了薄薄的一层外壳。于是，DSL 的成本和收益往往会比人们预想的要小，但也未曾得到过充分的认识。掌握良好的技术可以大大降低构造 DSL 的成本，我希望这本书可以帮你做到这一点。这层外壳虽薄，却也实用，值得一试。

为什么现在写这本书

DSL 已问世很长时间，但近些年来，它们掀起了一股流行风潮。与此同时，我决定用几年的时间写这本书。为什么呢？虽然我并不知道自己能否给这股风潮下一个权威的定义，但是可以分享一下自己的观点。

在千禧年到来的时候，编程语言界——至少在我的企业软件世界里——隐约出现了一种颇具统治性的标准。先是 Java，它在几年的时间里风光无限。即使后来微软推出的 C# 挑战了 Java 的统治地位，但这个新生者依然是一门跟 Java 很相似的语言。新时代的软件开发被编译型的、静态的、面向对象的、语法格式跟 C 类似的语言统治着（甚至连 VB 都被迫变得尽可能具有这些性质）。

然而人们很快发现，并不是所有的事情都能在 Java/C# 的霸权下运作良好。有些重要的逻辑用这些语言无法很好实现，这导致了 XML 配置文件的兴起。不久之后，有程序员开玩笑说，他们写的 XML 代码比 Java/C# 代码都多。这固然有一部分原因是想在运行时改变系统行为，但也体现出人们的另外一个想法：用更容易定制的方式表达系统行为的各个方面。虽然 XML 噪音很多，但确实可以让你定义自己的词汇，而且提供了非常强大的层次结构。

不过后来人们实在无法忍受 XML 那么多的噪音了。他们抱怨尖括号刺伤了他们的双眼。他们希望一方面能够享受 XML 配置文件带来的好处，另一方面又不用承受 XML 的痛苦。

我们的故事到现在讲了一半，这个时候 Ruby on Rails 横空出世，耀眼的光芒让一切都褪尽了颜色。无论 Rails 这个实用平台在历史上会占据什么样的位置（我觉得这确实是个优秀的平台），它都已经给人们对于框架和库的认识带来了深远的影响。Ruby 社区有一个很重要的做事方式：让一切显得更加连贯。换句话说，在调用库的时候，就像用一门专门的语言进行编程一样。这不禁让我们回想起一门古老的编程语言：Lisp。通过它我们也看到了 Java/C# 这片坚硬的土地上绽开的花朵：连贯接口（fluent interface）在这两门语言中都变得流行起来，这大概要归功于 JMock 和 Hamcrest 的创始人的不断努力。

回头看看这一切，我发现这里面有知识壁垒。有的时候，使用定制的语法会更容易理解，实现也不难，人们却用了 XML；有的时候，使用定制的语法会简单很多，人们却把 Ruby 用得乱七八糟；有的时候，本来在他们常用的语言中使用连贯接口就可以轻易实现的事情，人们非要使用解析器。

我觉得这些事情都是因为存在知识壁垒才发生的。经验丰富的程序员对 DSL 的相关技术所知寥寥，没法对使用哪一项技术做出明智的判断。我对打破这个壁垒很感兴趣。

为什么 DSL 很重要

本书 2.2 节会讲述更多细节，不过我觉得需要学习 DSL（以及本书中提到的其他技术）的原因主要有两个。

第一个原因是提升程序员的生产力。先看下面这段代码：

```
input =~ /\d{3}-\d{3}-\d{4}/
```

你会认出这是个正则表达式匹配，也许你还知道它匹配的是什么。正则表达式常常由于过于费解而遭受指责，但试想一下，如果你所能够使用的都是普通的正则控制代码，这段模式匹配会变成什么样子。而这段代码跟正则表达式相比，又是何等容易理解，容易修改？

DSL 的第一个优势是它擅长在程序中的某些特定地方发挥作用，并且让它们容易理解，进而提高编写、维护的速度，也会减少 bug。

DSL 的第二个优势就不仅仅限于程序员的范畴了。因为 DSL 往往短小易读，所以非程序员也能看懂这些驱动他们重要业务的代码。把这些真实的代码暴露在理解该领域的人们面前，可以确保程序员和客户之间有非常顺畅的沟通渠道。

当人们谈论这类事情的时候，他们常说 DSL 可以让你不再需要程序员。我对这一论调极度不认同，毕竟那是说 COBOL 的。不过也确实有些语言是由那些自称不是程序员的人来用的，比如 CSS。对这种语言来说，读比写要重要得多。如果一个领域专家可以阅读并且理解核心业务代码中的绝大部分，那他就可以跟写这段代码的程序员进行深入细节的交流。

第二个原因是使用 DSL 并非易事，不过回报也是相当丰厚的。软件开发中最狭窄的瓶颈就是程序员和客户之间的沟通，任何可以解决这一问题的技术都值得学习。

别畏惧这本大厚书

看到这本书这么厚，你可能会吓一跳；我自己发现要写这么多内容的时候都忍不住倒吸一口冷气。我对大厚书的态度总是小心翼翼，因为我们用来阅读的时间是有限的，一本厚书就意味着时间上的大量投资。因此在这种情况下我更倾向于使用“姊妹篇”的方式。

姊妹篇实际上是关于一个主题的两本书。第一本是叙述性质的书，需要仔细阅读。我希望它可以大致地描述出这个主题的主要内容，让读者有个整体认识就好，不用深入细节。我觉得叙述部分最好不要超过 150 页，这是个比较合理的厚度。

第二本书是参考资料，不需要一页一页翻阅（虽然有些人也这样看）。用的时候再仔细看就行。有些人喜欢先读完第一本，有了整体认识之后，再去看第二本书里面感兴趣的章节。有些人喜欢一边读第一本，一边找第二本中感兴趣的地方读。我之所以采用这种划分方式，主要还是想让你们了解哪些地方可以跳过，哪些地方不能忽略，这样你也就可以有选择地深入阅读了。

我已经尽力让参考资料那部分独立成篇了，如果你想让某人使用“树的构建”（第 24 章），就让他阅读那个模式，即使他可能对 DSL 没有清楚的认识，但是也能知道怎么做。这样一来，一旦你完全理解了概述部分，这本书就变成了参考资料，想查详细资料的话，一翻开就能找到。

本书之所以这么厚，是因为我没能找到把它变薄的方法。它的一个主要目的是分析、比较 DSL 的各项技术。讨论代码生成、Ruby 元编程、“解析器生成器”（第 23 章）工具的书有很多，本书涵盖所有这些技术，让你可以了解它们的异同。它们都在广阔的舞台上扮演着自己的角色，在帮助你了解这些技术之外，我还想介绍一下这个舞台。

本书主要内容

本书旨在全面介绍各种 DSL 及其构造方式。当人们尝试 DSL 的时候，经常就只选一种技术。你可以在本书里看到对多种技术的介绍，真正用的时候就可以做出最合适的选择。本书还提供了很多 DSL 技术的实现细节和例子。当然，我无法把所有的细节都写下来，但也足以使读者入门，在早期决策时起到辅助作用。

前 3 章讲述什么是 DSL、DSL 的用途以及 DSL 与框架和库的区别。第 5 章和第 6 章可以帮你理解如何构建外部 DSL 和内部 DSL。第三部分讲述解析器所扮演的角色，“解析器生成器”（第 23 章）的作用，用解析器解析外部 DSL 的各种方式。第四部分展示了在一种 DSL 风格中所能使用的多种语言结构。虽然它不能告诉你怎样充分利用你钟爱的语言，却能帮助你理解一门语言中的技术在不同语言之间的对应关系。第五部分介绍其他计算模型，有助于读者学习如何构建模型。

第六部分列出了生成代码的各种策略，你需要的话可以看一下。第 9 章简单介绍了新一代的工具。本书所介绍的绝大部分技术都已经面世很长时间了；语言工作台更像是未来科技，虽然应当有美好的前景，但没有经验证明。

本书读者对象

本书的理想读者是那些正在思考构建 DSL 的职业软件程序员。我觉得这种类型的读者都应该有多年的工作经验，认同软件设计的基本思想。

如果你深入研究过语言设计的话题，那这本书里大概不会有什么你没有接触过的资料。我倒是希望我在书中整理并表述信息的方式对你有所帮助。虽然人们在语言设计方面做了大量的工作——尤其是在学术领域，可这些成果能够为专业编程领域服务的却寥寥无几。

叙述部分的前几章还可以澄清一些困惑，比如什么是 DSL，DSL 有什么用途。通读第一部分以后，你就可以更全面地掌握 DSL 的不同实现技术。

这是本 Java 书或者 C# 书吗

本书和我曾写过的大部分书一样，与编程语言没有多大关系。我最想做的事情是揭示一些与编程语言无关的通用原则和模式。因此，不管你用的是哪种流行的面向对象语言，本书里的思想都会为你提供帮助。

函数式语言可能会是一条代沟。虽然我觉得很多内容依然对函数式语言适用，但我在函数式编程中的经验并不足以让我做出判断，它们的编程范式到底会从多大程度上影响到书中的建议。本书对于过程式语言（即非面向对象的语言，例如 C）的作用也很有限，因为我讲的很多技术都依赖于面向对象技术。

虽然我写的是通用原则，但为了能够把它们恰当地讲述出来，我还需要一些例子——于是需要一门具体的编程语言。在选择用哪门语言来写例子的时候，我的首要标准是有多少人能读懂它。于是绝大多数例子都是用 Java 或 C# 写的。这两门语言在业界广泛使用，有很多相似之处：类 C 的语法、内存管理，为人们提供各种便利的类库。但我的意思可不是说它们就是写 DSL 的最佳选择了（这里需要特别强调一下，因为我根本就不认为它们是最佳选择），只是说它们最能够帮助读者理解我讲的通用概念。我尽力让二者出现的机会均等，只

有当使用某种语言更方便的时候，我的天平才会稍稍倾斜一下。虽然内部 DSL 的良好运用常常要用到某些另类的语法特色，但我也尽力避免使用一些需要太多语法知识才能理解的语言元素，着实挺困难的。

还有一些思想是必须使用动态语言才能满足的，没法用 Java 或 C# 实现。在那些情况下我就换用 Ruby，因为这是我最熟悉的动态语言。它为我提供了很多帮助，因为它的特性完美地契合了编写 DSL 的需求。另外再强调一点，虽然我个人更熟悉某种语言，在选择时也考虑了个人偏好，但这并不能推断出这些技术换个地方就不能用了。我很喜欢 Ruby，但如果你想看看我对语言的偏执，那只有贬低 Smalltalk 才行。

值得一提的是，另外有许多语言都适合构建 DSL，尤其还有一些是专门为了写内部 DSL 而设计出来的。我之所以没有提到它们，是因为我对它们所知不多，没有足够的信心进行评价。你不要认为我对它们有什么负面观点。

另外还要提一句，在写这本书的时候，本来试图和语言无关，可大多数技术的实用性偏偏都要直接依赖于某种语言的特性。这是让我最苦恼的地方了。我为了实现大范围内的通用性做出了很多权衡，但你必须意识到，这些权衡可能会因具体的语言环境而彻底改变。

本书缺少什么

在写这样一本书的过程中，要说什么时候最让人垂头丧气、濒临崩溃，莫过于自己意识到必须停笔的那一刻了。我为这本书花费了几年的时间，我相信这里面有很多值得你阅读的内容。但我也知道我留了很多疏漏之处。我本来想弥补这些疏漏，可这得占用大量时间。我的信念是，宁可出版一本未完成的书，也不要再等上几年把书写完——即便是真的能够写完的话。下面简单介绍一下我实在没时间补充的内容。

前面曾提到过一点——函数式语言。实际上，在当代那些基于 ML 和 / 或 Haskell 的函数式语言中，构建 DSL 已经是广为人知的事实了。我在书中基本没提这部分内容。我也很想知道，当我熟悉了函数式语言及其 DSL 应用之后，本书的内容安排会发生多大的改变。

有一件事情最让我心神不安，那就是没有把近期有关诊断和错误处理的讨论加进去。我记得上大学的时候曾学到过，诊断这件事情在写编译器的过程中是何等艰难，因此忽略这一点让我觉得自己像是在作表面文章。

我个人最喜欢第 7 章。我本来可以写很多内容的，只可惜时不我予。最后我只好决定少写一些——希望它依然可以激发你主动探索更多模型的兴趣。

章节引用

虽然本书的结构比较普通，但引用章节的结构还是需要稍稍介绍一下的。我把引用章节分成一系列主题，按照相似性组成不同的章节。我的想法是每个主题都可以独立成篇，于是

你读完第一部分以后，就可以任选一个主题深入了解，无须再涉及其他章节。如果有例外情况的话，我会在对应主题的开篇提到。

大部分主题都以模式的形式呈现。模式是对于一再重复出现的问题的通用解决方案。所以如果你有一个常见的问题："我该怎么处理我的解析器结构呢？"对这个问题的两种可行模式是"分隔符指导翻译"（第 17 章）和"语法指导翻译"（第 18 章）。

在过去的二十年间，人们写了很多关于软件开发模式的书，不同的人有不同的视角。我的看法是，模式给我提供了一种组织参考资料的良好方式。第一部分告诉你如果想要解析文本，可以考虑上面两种模式。模式本身提供了更多的信息以供选择和具体实施。

引用章节大都是以模式的结构来写的，但也有些例外：并不是所有的主题在我眼中都是解决方案。比如"嵌套的运算符表达式"（第 29 章），它的重点就不是解决方案，也不符合模式的结构，因此我没采用模式风格的描述方式。还有一些情况很难称为模式，比如"宏"（第 15 章）和"BNF"（第 19 章），可是用模式结构来描述它们却很合适。总的来说，只要是模式结构——尤其是把"如何起作用"和"何时使用模式"分离开这种形式——能够帮我描述概念，我就一直在使用它。

模式结构

大多数作者在写模式的时候都用了一些标准模板。我也不例外，既用了一个标准模板，又与别人用的不一样。我所用的模板，或称模式形态，是我第一次用在企业应用架构模式（P of EAA，[Fowler PoEAA]）中的模式。它的形式如下。

模板中最重要的元素大概要数名字。我喜欢用模式来描述各个引用主题，最大的原因就是它可以帮我创建一个强大的词汇表，方便展开讨论。虽然这个词汇表不一定能得到广泛应用，但至少可以让我的写作保持一致性，也可以在别人想要用这个模式的时候，给他提供一个起始点。

接下来的两个元素是意图和概要。它们对模式进行简要的概括。它们还能起到提醒作用，如果你已"将模式纳入囊中"，但忘了名字，它们可以轻轻拨动你的记忆。意图用一句话总结模式，而概要是模式的一种可视化表示——有时候是一张草图，有时候是代码示例，不管是什么形式，只要能够快速解释模式的本质就好。我有时候使用图表，有时候会用 UML 画图，不过要是有其他方式可以更容易表达意图，我也是很乐意用的。

接下来就是稍长一些的摘要了。我一般会在这一部分给出一个例子，用来说明模式的用途。摘要由几段话组成，同样是为了让读者在深入细节之前先了解模式全貌。

模式有两个主体部分：工作原理和使用场景。这两部分没有固定顺序，如果你想了解是否该用某个模式，可能就只想读"使用场景"这一节。不过，一般来说，不了解工作原理的话，只看"使用场景"是没什么作用的。

最后一部分是例子。我尽力在"工作原理"这一节把模式的工作原理讲清楚，但人们一

般还是需要通过代码来理解。然而，代码示例是有危险的，它们演示的只是模式的一种应用场景，可有些人却会以为模式就是这个用法，没有理解背后的概念。你可以把一种模式用上千百遍，每次稍稍有些差异，可我没有地方容纳那么多代码，所以请记住，模式的含义远远不止从特定示例中看到的那么多。

所有的例子都设计得非常简单，只关注要讨论的模式本身。我用的例子都是相互独立的，因为我的目标是每一个引用章节都独立成篇。一般来说，在实际应用模式的时候还需要处理其他大量问题，但在一个简单的例子中，起码可以让你有机会理解问题的核心。丰富的例子更贴近现实，可它们也会引入大量与当前模式无关的问题。于是我只会展示一些片段，你需要自己把它们组装起来，满足特定的需求。

这同时也意味着我在代码中主要追求的是可读性。我没有考虑性能、错误处理等因素，它们只会把你的注意力从模式的核心转移到别处。

也基于这个原因，我力图避免写一些我觉得很难借鉴的代码，即便是更符合该语言的惯例也不予考虑。这个折衷在内部 DSL 上会显得有些笨拙，因为内部 DSL 经常要靠语言的小窍门来强化语言的连贯性。

很多模式都缺少上面讲的一两个部分，因为确实没什么可写的。有些模式没有例子，因为最合适的例子在其他模式里面用到了——在发生这种情况的时候，我会把它指出来。

致谢

当我每次写书的时候，很多人都为我提供了大量帮助。虽然作者那里写的是我的名字，但许多朋友都为提升本书的质量作出了巨大的贡献。

我首先要感谢的是我的同事 Rebecca Parsons。我对 DSL 这个主题曾有很多顾虑，例如，它会涉及很多学术背景的知识，而那些是我所不熟悉的。Rebecca 有深厚的语言理论背景，她在这方面给了我很多帮助。此外，她也是我们公司的首席技术探路人和战略家，她可以将她的学术背景和大量的实践经验合二为一。她有能力，并且也愿意为本书付出更多心血，但 ThoughtWorks 在其他方面更需要她。我很高兴，我们曾关于 DSL 这个主题滔滔不绝。

作者总是希望（并且带着小小的恐惧）审校者可以通读全书，找出不计其数的大大小小的错误。我幸运地找到了 Michael Hunger，他的审校工作做得极其出色。从本书刚刚出现在我网站上的时候，他就开始不断地给我挑错，催促我改正——这正是我需要的态度。他同时也催促我详细介绍使用静态类型的技术，尤其是静态类型的“符号表”（第 12 章）。他给我提供了无数建议，足以再写两本书了。我希望有朝一日可以把这些想法写下来。

在过去的几年里，我和同事们包括 Rebecca Parsons、Neal Ford、Ola Bini，写过很多这方面的文章。在本书里，我也借鉴了他们的一些成型的想法。

ThoughtWorks 慷慨地给予我大量时间来写这本书。在花了很长时间决定不再为某一家公司工作之后，我很高兴找到一家让我愿意留下来，并且积极地参与其建设的公司。

本书还有很多正式的审校者，他们为本书提供了大量建议，找出了很多错误。他们是：

David Bock
Gilad Bracha
Aino Corry
Sven Efftinge
Eric Evans
Jay Fields
Steve Freeman
Brian Goetz
Steve Hayes
Clifford Heath
Michael Hunger
David Ing
Jeremy Miller
Ravi Mohan
Terance Parr
Nat Pryce
Chris Sells
Nathaniel Schutta
Craig Taverner
Dave Thomas
Glenn Vanderburg

我还要感谢 David Ing，是他提出了第 10 章的章名。

成为一个系列丛书的编辑之后，我就有了些美妙的特权，比如，我拥有了一个很出色的作者团队，他们可以帮我出谋划策。其中我尤其要感谢 Elliotte Rusty Harold，他提供了很多精彩的建议和评论。

我在 ThoughtWorks 的很多同事也成为我想法的源泉。我要谢谢过去几年里允许我在项目中闲逛的每个人。我能写下来的远不及所看到的，能在这样广袤的海洋中徜徉，我感到无比愉悦。

有些人给 Safari 联机丛书的初稿提供了很多建议，我在正式付梓之前也参考了他们的想法。这些人是：Pavel Bernhauser、Mocky、Roman Yakovenko、tdyer。

我还要谢谢本书出版商 Pearson 的工作人员。Greg Doench 是本书的组稿编辑，他负责出版的整体流程。John Fuller 是本书的执行编辑，他监管生产流程。

Dmitry Kirsanov 调整了我拙劣的英语，让本书语言流畅。Alina Kirsanova 组织了本书的布局。

目　录

第二部分 通 用 主 题