华章科技

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

InfoQ软件开发丛书

华章程序员书库

领域特定语言

Domain-Specific Languages

（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

本书版权登记号：图字：01–2010–6648

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
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

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

第三部分　外部 DSL 主题

第四部分　内部 DSL 主题

第五部分 其他计算模型

第六部分　代 码 生 成

第一部分

叙　　述

第①章

入门例子

落笔之初，我需要快速地解释一下本书的内容，就是解释什么是领域特定语言（Domain–Specific Language，DSL）。为达此目的，我一般都会先展示一个具体的例子，随后再给出抽象的定义。因此，我会从一个例子开始，展示 DSL 可以采用的不同形式。在第 2 章里，我会试着把这个定义概括为一些更广泛适用的东西。

1.1 哥特式建筑安全系统

在我的童年记忆里，电视上播放的那些低劣的冒险电影是模糊却持久的。通常，这些电影的场景会安排某个古旧的城堡、密室或走廊在其中起着重要的作用。为了找到它们，英雄们需要拉动楼顶的蜡烛托架，然后，轻拍两次墙壁。

想象有这样一家公司，他们决定根据这个想法构建一套安全系统。他们进入之后，设置某种无线网络，安装一些小的设备。如果发生某些有趣的事情，这些设备会发出一条四字符的消息。比如，打开抽屉，抽屉上附着的感应器就会发出一条消息：D2OP。还有一些小的控制设备，响应这样的四字符命令消息，比如，某个设备收到 D1UL 消息，就会打 开一扇门。

所有这一切的核心是一些控制器软件，它们会监听事件消息，弄清楚要做什么，然后发送命令消息。在那个网络产业崩溃的年代，这家公司买到了一堆廉价的烤面包机，它们可以用 Java 控制，所以，公司准备用它们来做控制器。因此，只要客户买了哥特式建筑安全系统，公司就会进驻其中，给这个建筑物装上一大堆设备，当然，还有一个烤面包机，里面装有 Java 编写的控制程序。

就这个例子而言，我们的关注点在于这个控制程序。每个客户都有各自的需求，但是，只要有一些好的样本，我们就不难发现其中的通用模式。为了打开密室，格兰特小姐要关上卧室房门，打开抽屉，然后，再开一盏灯。而肖瓦小姐则先要打开水龙头，再打开正确的灯，从而开启两个密室中的一个。至于史密斯小姐，她的办公室里有一个上锁的壁橱，内有一个密室。她必须先关上门，把墙上的画摘下来，再打开桌子上的灯三次，最后，打开那个满满的橱柜最上面的抽屉，这时，壁橱打开了。如果在打开里面的密室前，她忘记了关闭桌子上的灯，就会警报大作。

这是个异想天开的例子，但它所要表达的意图一点都不特别。我们有这样一系列系统，它们共享着大多数组件和行为，却彼此间差异极大。在这个例子里，对所有的客户来说，控

制器发送和接收消息的方式是相同的，但是产生的事件序列和发送的命令却不尽相同。我们要整理一下这些东西，这样，当公司安装一个全新系统时，付出的代价才会是最小的。因此，对他们而言，为控制器编写动作序列必须非常简单才行。

了解了所有这些情况，一种好的处理方式浮出水面：把控制器看做状态机（state machine）。每个感应器都可以发送事件（event），改变控制器的状态（state）。当控制器进入某种状态时，它就会在网络上发出一条命令消息。

至此，我应该坦白，写作之初，这是全然不同的。状态机是一个很好的 DSL 例子，因此，我先选择了它。之所以选择 哥特式城堡，是因为我厌倦了其他所有状态机的例子。

格兰特小姐的控制器

这家神秘的公司拥有成千上万感到满意的客户，但在这里，我们只准备强调其中的一位：格兰特小姐，我最喜欢的客户。她的卧室里有个密室，通常情况下，这个密室都会紧锁着，隐蔽在那里。要打开这个密室，她就要关上门，然后，打开柜子里的第二个抽屉，打开床边的灯——二者顺序任意。做完这些，秘密面板就会解锁，她就可以打开密室了。

我用一张状态图来表示这个序列（见图 1-1）。

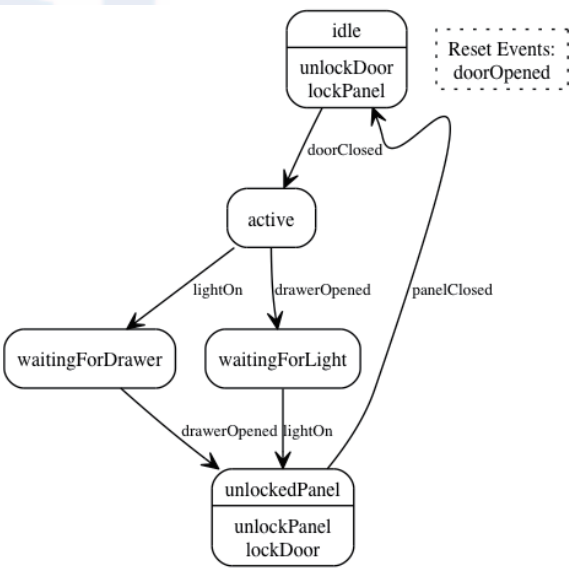

图 1-1　格兰特小姐密室的状态图

如果你没接触过状态机，其实，它们只不过是描述行为的一种常见方式而已——并非广泛适用，但对于类似这样的情况，却是再合适不过了。其基本的想法是，控制器会处于不同的状态。当它们处于某个特定的状态时，某个事件会让控制器转换为另一个状态，在那种状态下会有不同的转换（transition）。因此，一系列的事件会让控制器从一个状态进到另一个状态。在这个模型里，当进入一个状态时，会执行某个动作（比如发送消息）。（其他类型的状态机可能会在不同的地方执行动作）。

基本上，这个控制器就是一个简单的传统状态机，不过需要一些微调。客户的控制器要有一个明确的空闲（idle）态，系统会有大多数的时间处在该状态。某种特定的事件就可以让系统跳回到这个空闲态，即便它正处于一个更有趣的状态转换中间，这样就可以有效地重置整个模型了。在格兰特小姐的这个例子里，开门就是这样一个重置事件。

引入重置事件，意味着这里描述的状态机并不完全满足某种经典的状态机模型。状态机

有几种非常有名的变体，这个模型便是以其中一个为起点的，只是略做微调，增加了重置事件，也就变成了只针对这种情况。

需要特别注意的是，严格来说，未必一定要有重置事件才能表示格兰特小姐的控制器。一种替代方案是，为每个状态添加一个转换，只要触发 doorOpened，就会转换为空闲态。重置事件这个想法很有用，因为它简化了整个状态图。

1.2　状态机模型

一旦团队达成共识，认为对于指定控制器如何运作而言，状态机是一个恰当的抽象，那么，下一步就是确保这个抽象能够运用到软件自身。如果人们在考虑控制器行为时，也要考虑事件、状态和转换，那么，我们希望这些词汇也可以出现在软件代码里。从本质上说，这就是领域驱动设计（Domain–Driven Design）中的 Ubiquitous Language [Evans DDD] 原则，也就是说，我们在领域人员（那些描述建筑安全该如何运作的人）和程序员之间构建的一种共享语言。

对于 Java 程序来说，处理这种事，自然的方式就是以状态机为 Domain Model [Fowler PoEAA]。状态机框架的类图见图 1-2。

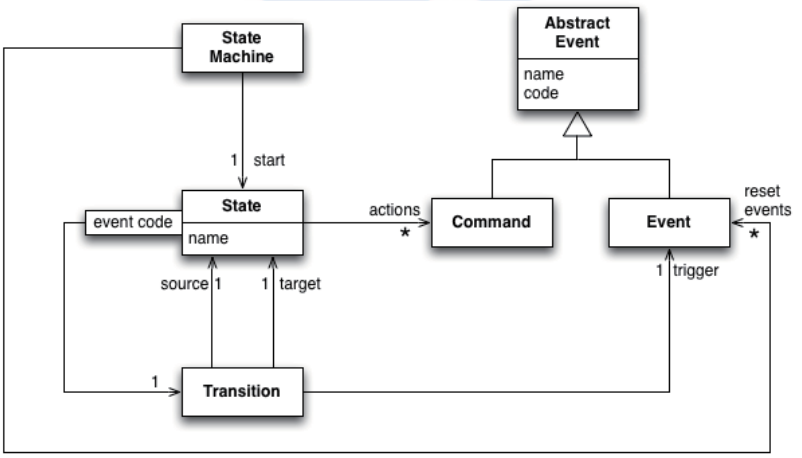

图 1-2　状态机框架的类图

通过接收事件消息和发送命令消息，控制器得以同设备通信。这些消息都是四字母编码，它们可以通过通信通道进行发送。在控制器代码里，我想用符号名（symbolic name）引用这些消息。我创建了事件类和命令类，它们都有代码（code）和名字（name）。我把它们放到单独的类里（有一个超类），因为在控制器的代码里，它们扮演着不同的角色。

```
class AbstractEvent...
 private String name, code;

 public AbstractEvent(String name, String code) {
```

```
  this.name = name;
  this.code = code;
 }
 public String getCode() { return code;}
 public String getName() { return name;}
public class Command extends AbstractEvent
public class Event extends AbstractEvent
```

状态类记录了它会发送的命令及其相应的转换。

```
class State...
 private String name;
 private List<Command> actions = new ArrayList<Command>();
 private Map<String, Transition> transitions = new HashMap<String, Transition>();

class State...
 public void addTransition(Event event, State targetState) {
  assert null != targetState;
  transitions.put(event.getCode(), new Transition(this, event, targetState));
 }

class Transition...
 private final State source, target;
 private final Event trigger;

 public Transition(State source, Event trigger, State target) {
  this.source = source;
  this.target = target;
  this.trigger = trigger;
 }
 public State getSource() {return source;}
 public State getTarget() {return target;}
 public Event getTrigger() {return trigger;}
 public String getEventCode() {return trigger.getCode();}
```

状态机保存了其起始状态。

```
class StateMachine...
 private State start;

 public StateMachine(State start) {
  this.start = start;
 }
```

这样，从这个状态可以到达状态机里的任何状态。

```
class StateMachine...
 public Collection<State> getStates() {
  List<State> result = new ArrayList<State>();
  collectStates(result, start);
  return result;
 }

 private void collectStates(Collection<State> result, State s) {
  if (result.contains(s)) return;
  result.add(s);
  for (State next : s.getAllTargets())
   collectStates(result, next);
 }
```

```
class State...
 Collection<State> getAllTargets() {
  List<State> result = new ArrayList<State>();
  for (Transition t : transitions.values()) result.add(t.getTarget());
  return result;
 }
```

为了处理重置事件，我在状态机上保存了一个列表。

```
class StateMachine...
 private List<Event> resetEvents = new ArrayList<Event>();

 public void addResetEvents(Event... events) {
  for (Event e : events) resetEvents.add(e);
 }
```

像这样用一个单独结构处理重置事件并不是必需的。简单地在状态机上声明一些额外的转换，也可以处理这种情况，如下所示：

```
class StateMachine...
 private void addResetEvent_byAddingTransitions(Event e) {
  for (State s : getStates())
   if (!s.hasTransition(e.getCode())) s.addTransition(e, start);
 }
```

我倾向于在状态机上设置显式的重置事件，这样可以更好地表现意图。虽然这样做确实使状态机有点复杂，但它也更加清晰地表现出通用状态机该如何运作，要定义特定状态机也会更加清晰。

处理完结构，再来看看行为。事实证明，这真的相当简单。控制器有个 handle 方法，它以从设备接收到的事件代码为参数。

```
class Controller...
 private State currentState;
 private StateMachine machine;

 public CommandChannel getCommandChannel() {
  return commandsChannel;
 }

 private CommandChannel commandsChannel;

 public void handle(String eventCode) {
  if (currentState.hasTransition(eventCode))
   transitionTo(currentState.targetState(eventCode));
  else if (machine.isResetEvent(eventCode))
   transitionTo(machine.getStart());
   // ignore unknown events
 }

 private void transitionTo(State target) {
  currentState = target;
  currentState.executeActions(commandsChannel);
 }

class State...
```

```
 public boolean hasTransition(String eventCode) {
  return transitions.containsKey(eventCode);
 }
 public State targetState(String eventCode) {
  return transitions.get(eventCode).getTarget();
 }
 public void executeActions(CommandChannel commandsChannel) {
  for (Command c : actions) commandsChannel.send(c.getCode());
 }

class StateMachine...
 public boolean isResetEvent(String eventCode) {
  return resetEventCodes().contains(eventCode);
 }

 private List<String> resetEventCodes() {
  List<String> result = new ArrayList<String>();
  for (Event e : resetEvents) result.add(e.getCode());
  return result;
 }
```

对于未在状态上注册的事件，它会直接忽略。对于可识别的任何事件，它就会转换为目标状态，并执行这个目标状态上定义 的命令。

1.3　为格兰特小姐的控制器编写程序

至此，我们已经实现了状态机模型，接下来，就可以为格兰特小姐的控制器编写程序了，如下所示：

```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);

idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened, waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);

waitingForLightState.addTransition(lightOn, unlockedPanelState);
```

```
waitingForDrawerState.addTransition(drawerOpened, unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);

machine.addResetEvents(doorOpened);
```

经过查看上述代码，我们发现，它与之前的代码有着很大的不同。之前的代码描述了如何构建状态机模型，而上面这段代码则是在配置一个特定的控制器。我们常常会看到这样一种划分：一方面是程序库（见图 1-3）、框架或者组件的实现代码；另一方面是配置代码或组件组装代码。从本质上说，这种做法分开了公共代码和可变代码。用公共代码构建一套组件，然后根据不同的目的进行配置。

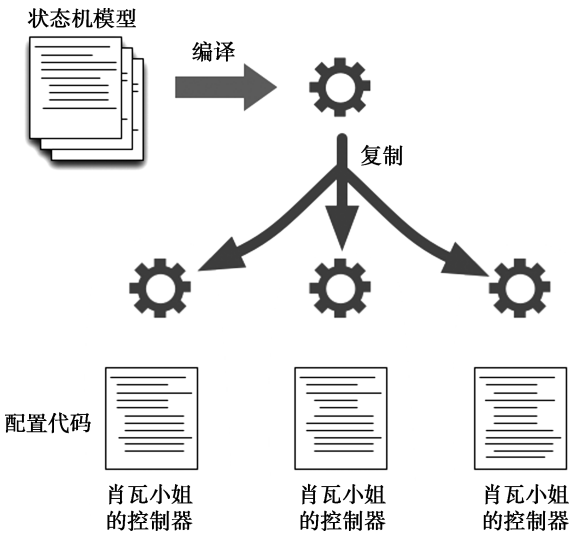

图 1-3　一个用于多种配置的程序库

配置代码还有另外一种表现形式：

```
<stateMachine start = "idle">
 <event name="doorClosed" code="D1CL"/>
 <event name="drawerOpened" code="D2OP"/>
 <event name="lightOn" code="L1ON"/>
 <event name="doorOpened" code="D1OP"/>
 <event name="panelClosed" code="PNCL"/>

 <command name="unlockPanel" code="PNUL"/>
 <command name="lockPanel" code="PNLK"/>
 <command name="lockDoor" code="D1LK"/>
 <command name="unlockDoor" code="D1UL"/>

 <state name="idle">
  <transition event="doorClosed" target="active"/>
  <action command="unlockDoor"/>
  <action command="lockPanel"/>
```

```
 </state>

 <state name="active">
  <transition event="drawerOpened" target="waitingForLight"/>
  <transition event="lightOn" target="waitingForDrawer"/>
 </state>

 <state name="waitingForLight">
  <transition event="lightOn" target="unlockedPanel"/>
 </state>

 <state name="waitingForDrawer">
  <transition event="drawerOpened" target="unlockedPanel"/>
 </state>

 <state name="unlockedPanel">
  <action command="unlockPanel"/>
  <action command="lockDoor"/>
  <transition event="panelClosed" target="idle"/>
 </state>

 <resetEvent name = "doorOpened"/>
</stateMachine>
```

大多数读者对这种表现形式应该更熟悉一些：表现为 XML 文件。这种做法有几个好处。第一个明显的好处是，无须为每个要实现的控制器编译一个单独的 Java 程序，相反，只要把状态机组件加上相应的解析器编译到一个公共的 JAR 里，然后，发布对应的 XML 文件，当状态机启动时读取这个文件。控制器行为的任何修改都无须发布新的 JAR。当然，我们需要为此付出一些代价，许多配置上的语法错误只能在运行时检测出来，虽然各种各样的 XML 模式系统还能帮上点忙。我还是“广泛测试”（extensive testing）的超级粉丝，广泛测试不仅可以在编译时检查就捕获到大多数错误，还可以发现一些类型检查无法确定的致命问题。有了这种及时测试，就不必担心把错误检测带到运行时。

第二个好处在于文件本身的表现力。我们不必再去考虑通过变量进行连接的细节。相反，我们拥有了一种声明方式，从许多方面来看，这种方式读起来都会更加清晰。这里还有一些限制：这个文件只能表示 配置——这种限制也是有益的，因为它会降低人们编写组装组件代码犯错的概率。

或许，你听说过声明式编程这回事。对我们而言，更常见的模型是命令式（imperative）模型，也就是，用一系列的步骤指挥计算机。“声明式”是一个非常模糊的术语，但是，它通常适用于所有远离了命令式编程的方式。在这里，我们向那个方向迈进了一步：远离变量倒换，用 XML 的子元素表示状态内的动作和转换。

正是有了这些好处，如此之多的 Java 和 C# 的框架才采用 XML 配置文件配置。现如今，有时我们会觉得自己是在用 XML 编写程序，而非自己的主编程语言。

下面是配置代码的另一个版本：

```
events
 doorClosed D1CL
 drawerOpened D2OP
```

```
 lightOn   L1ON
 doorOpened D1OP
 panelClosed PNCL
end

resetEvents
 doorOpened
end

commands
 unlockPanel PNUL
 lockPanel  PNLK
 lockDoor  D1LK
 unlockDoor D1UL
end

state idle
 actions {unlockDoor lockPanel}
 doorClosed => active
end

state active
 drawerOpened => waitingForLight
 lightOn  => waitingForDrawer
end

state waitingForLight
 lightOn => unlockedPanel
end

state waitingForDrawer
 drawerOpened => unlockedPanel
end

state unlockedPanel
 actions {unlockPanel lockDoor}
 panelClosed => idle
end
```

这确实是代码，虽然使用的不是我们所熟悉的语法。实际上，这是一种定制语法，专为这个例子而打造的。相比于 XML 语法，我认为这种语法更易写，最重要的是，更易读。它更简洁，省却了许多引号和噪音字符，这些是用 XML 所要忍受的。或许，你的做法不尽相同，但重点在于，我们可以构造自己和团队所喜欢的语法。我们依然可以在运行时加载（就像 XML 一样），但是，这么做不是必需的（XML 也不必这么做），如果想在编译时加载的话。

这样的语言就是领域专用语言，它有着 DSL 的许多特征。首先，它只适用于非常有限的用途——除了配置这种特定的状态机外，它什么都做不了。这样带来的结果就是，这个 DSL 非常简单——它没有控制结构，也没有其他的东西。它甚至不是图灵完备的。用这种语言不能编写整个应用，它所能做的只是描述应用一个小的方面。这样做的结果就是，DSL 只有同其他语言配合起来，才能完成整个工作。但是，DSL 的简单性意味着，它是易于编辑和处理的。

对于编写控制器软件的人而言，这种简单性意味着更容易理解，并且开发人员之外的人也可以理解这种行为。搭建系统的人能够查看这段代码，并且理解它的运作方式，虽然他们

无法理解控制器本身的 Java 代码。即便他们只读了 DSL，但对于指出错误或者同 Java 开发人员进行有效的交流来说，这就足够了。DSL 扮演领域专家和业务分析人员之间的交流媒介，虽然构建这种 DSL 也存在一些实际的困难，但能够在软件开发最困难的交流鸿沟上架起一座桥梁，其益处也让 这种尝试物有所值。

现在，回顾一下 XML 的表示形式。这是一种 DSL 吗？我想说，它是。只不过它是用 XML 的语法载体而已—— 但是它依旧是 DSL。这个例子引出了一个设计问题：哪种做法更好：为 DSL 定制语法，还是使用 XML 语法？ XML 更易于解析，因为人们已经熟悉了解析 XML。（然而，同为定制语法编写解析器相比，为 XML 编写解析器花了我同样多的时间。）我要声明一点，定制语法易读得多，至少在这个例子里是这样的。然而，回顾一下这个选择，我们会发现，DSL 核心部分的权衡也是一样的。的确，我们可以认为，大多数 XML 配置文件本质上都是 DSL。

看看下面这段代码，它看上去像是这个问题的 DSL 吗？

```
event :doorClosed, "D1CL"
event :drawerOpened, "D2OP"
event :lightOn, "L1ON"
event :doorOpened, "D1OP"
event :panelClosed, "PNCL"

command :unlockPanel, "PNUL"
command :lockPanel,  "PNLK"
command :lockDoor,  "D1LK"
command :unlockDoor, "D1UL"

resetEvents :doorOpened

state :idle do
 actions :unlockDoor, :lockPanel
 transitions :doorClosed => :active
end

state :active do
 transitions :drawerOpened => :waitingForLight,
       :lightOn => :waitingForDrawer
end

state :waitingForLight do
 transitions :lightOn => :unlockedPanel
end

state :waitingForDrawer do
 transitions :drawerOpened => :unlockedPanel
end

state :unlockedPanel do
 actions :unlockPanel, :lockDoor
 transitions :panelClosed => :idle
end
```

同之前的定制语言相比，它稍微有些噪音，但依旧相当清晰。与我有相近语言偏好的人可能看出来了，这是 Ruby。在创建更可读的代码方面，Ruby 给了我许多语法上的支持，因

此，我可以使它很像一门定制语言。

Ruby 开发人员会把这段代码当做一种 DSL。我用到的是 Ruby 这方面能力的一个子集，表现的想法同使用 XML 和定制语法是一样的。从本质上说，我是把 DSL 嵌入 Ruby 里，用 Ruby 的子集作为我的语法。从某种程度上来说，这更多地取决于态度，而非其他什么。我选择透过 DSL 眼镜观看 Ruby 代码。但这是一个具有长期传统的观点—— Lisp 程序员通常会考虑在 Lisp 里创建 DSL。

在此，我要指出，文本 DSL 有两种，称为外部 DSL（external DSL）和内部 DSL（internal DSL）。外部 DSL 是指，在主程序设计语言之外，用一种单独的语言表示领域专用语言。这种语言用的可能是定制语法，或者遵循另一种表现的语法，比如 XML。内部 DSL 是指，用通用语言的语法表示的 DSL。这种做法就是出于领域专用的目的，而按照某种风格来使用这种语言。

也许有人还听说一个术语，嵌入式 DSL（embedded DSL），它是内部 DSL 的同义词。虽然这个术语应用得相当广泛，但我还是会避免使用它，因为“嵌入式语言”（embedded language）同样适用于在应用中嵌入的脚本语言，比如 Excel 里的 VBA，Gimp 里的 Scheme。

回过头来考虑一下原来的 Java 配置代码。它是一种 DSL 吗？我想说，不是。这段代码感觉像是同 API 缝合在一起的，而上面的 Ruby 代码则更有声明式语言的感觉。这是否意味着无法用 Java 实现内部 DSL 呢？下面这段代码怎么样？

```
public class BasicStateMachine extends StateMachineBuilder {

 Events doorClosed, drawerOpened, lightOn, panelClosed;
 Commands unlockPanel, lockPanel, lockDoor, unlockDoor;
 States idle, active, waitingForLight, waitingForDrawer, unlockedPanel;
 ResetEvents doorOpened;

 protected void defineStateMachine() {
  doorClosed. code("D1CL");
  drawerOpened. code("D2OP");
  lightOn.  code("L1ON");
  panelClosed.code("PNCL");

  doorOpened. code("D1OP");

  unlockPanel.code("PNUL");
  lockPanel. code("PNLK");
  lockDoor.  code("D1LK");
  unlockDoor. code("D1UL");

  idle
   .actions(unlockDoor, lockPanel)
   .transition(doorClosed).to(active)
   ;

  active
   .transition(drawerOpened).to(waitingForLight)
   .transition(lightOn).  to(waitingForDrawer)
   ;
```

```
    waitingForLight
     .transition(lightOn).to(unlockedPanel)
     ;

    waitingForDrawer
     .transition(drawerOpened).to(unlockedPanel)
     ;

    unlockedPanel
     .actions(unlockPanel, lockDoor)
     .transition(panelClosed).to(idle)
     ;
   }
  }
```

虽然这段代码格式上有些奇怪，而且用到了一些不常见的编程约定，但它确实是有效的 Java。这段代码我愿意称为 DSL；虽然同 Ruby DSL 相比，它有些乱，但它还是有 DSL 所需的声明流。

是什么让内部 DSL 不同于通常的 API 呢？这是一个很难回答的问题，稍后，在 4.1 节，我会花更多的时间来讨论，但它会归结为一种流，只不过用的是一种类语言的模糊记法而已。

也许，有人还碰到过内部 DSL 的另一个术语，连贯接口（fluent interface）。这个术语强调这样一个事实：内部 DSL 实际上只是某种形式的 API，只不过其设计考虑到了连贯性难以琢磨的质量。鉴于这种差别，最好给非连贯 API 一个名字（我用的术语是，命令）查询 API（command–query API）。

1.4　语言和语义模型

在这个例子之初，我谈到了构建一个状态机模型。这种模型的存在，以及它同 DSL 的关系，是至关重要的。在这个例子里，DSL 的角色就是组装状态机模型。因此，当解析定制语法的版本时，遇到：

```
events
  doorClosed D1CL
```

会创建一个新的事件对象（`new Event("doorClosed","D1CL")`），把它保存在一边（在一个“符号表”（第 14 章）里），这样，遇到 `doorClosed=>active` 时，就可以将它包含在一个转换里（使用 addTransition）。这个模型就是个引擎，它提供了状态机的行为。事实上，可以说，这个设计的能力大多源自这样一个模型。如图 1-4 所示，DSL 所做的一切就是提供一种更可读的方式来组装这个模型——这就是与开始的命令查询 API 不同的地方。

从 DSL 的角度来看，我把这个模型称为“语义模型”（第 11 章）。谈及编程语言时，我们常常会提及语法（syntax）和语义（semantics）。语法描述程序的合法表达式，而在定制语法的 DSL 里所能描述的一切是由文法（grammar）决定的。程序的语义是指，它代表着什么，也就是说，当执行时，它能做什么。在这个例子里，模型定义了语义。如果你习惯使用

Domain Model [Fowler PoEAA]，这里就可以认为语义模型是与之非常类似的东西。

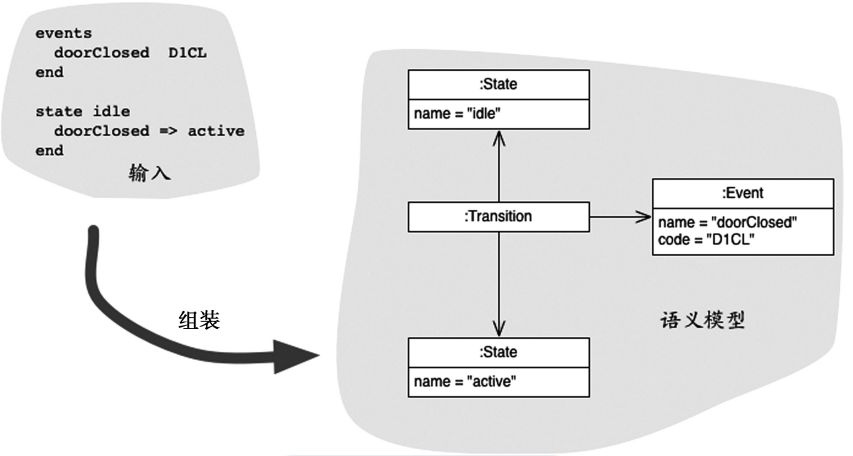

图 1-4　解析 DSL 组装语义模型

（你可以查看一下“语义模型”（第 11 章），了解一下语义模型同 Domain Model 之间的差异，还有语义模型和抽象语法树之间的不同。）

我有一个观点，对于一个设计良好的 DSL 而言，语义模型至关重要。在实际中，有的 DSL 用语义模型，而有些没有，但是我强烈建议，几乎始终（almost always）应该使用语义模型。（我发现，当说一些词比如“always”时，如果不加上限定词“almost”，几乎是不可能的。我几乎还没有找到一条广泛适用的规则。）

我提倡使用语义模型，因为它清晰地将语言解析和结果语义的关注点切分开。我可以推究出状态机的运作机制，对状态机进行增强和调试，而无须顾及语言问题。通过命令 – 查询接口，就可以组装状态机测试模型。状态机模型和 DSL 可以独立演进，即便还没想好如何通过语言表示，依然可以为模型添加新特性。也许，最关键的点在于，模型可以独立测试，而无须涉及语言。确实，上面所有 DSL 的例子都构建在相同语义模型上，基于这个模型，可以创建出完全相同的配置对象。

在这个例子里，语义模型是对象模型。语义模型还可以有其他形式。即便它只是一个纯粹的数据结构，所有的行为 都在单独的函数里，我依然愿意称之为语义模型，因为在那些函数的上下文里，数据结构表现出了 DSL 脚本特定的含义。

从这个角度来看，DSL 只是扮演着展示模型配置机制的角色。使用这种方式的益处大多源自模型，而非 DSL。为客户配置新的状态很容易，这是模型的属性，而非 DSL。控制器可以在运行时改变，无须编译，这是模型的特性，而非 DSL。可以在控制器的多次安装中重用代码，这是模型的属性，而非 DSL。由此可见，DSL 只是模型的一个薄薄的“门面”（facade）。

模型提供了诸多益处，与 DSL 如何表现无关。因此，我们一直使用它们。通过使用程序库和框架，我们可以明智地回避一些工作。在我们自己的软件里，构建模型，增进抽象，这样，就可以更快地开发。无论是作为程序库或者框架发布，或只是为自己的代码服务，即

便没有任何可见的 DSL，良好的模型都可以运作良好。

不过，DSL 可以增强模型的能力。正确的 DSL 让我们更容易理解一个特定状态机的运作机制。一些 DSL 甚至可以让我们 在运行时配置模型。因此，DSL 是对模型的一个有益补充。

DSL 所带来的益处与状态机紧密相关，其所组成的某个特定模型就扮演了系统程序的角色。要改变状态机的行为，就需要修改模型中的对象及其相互关系。这种风格的模型通常称为“适应性模型”（第 47 章）。这样得到的是一个模糊了代码和数据之间差异的系统，只看代码，是无法理解状态机行为的，还必须了解对象实例的连接方式。当然，从某种程度上说，这总是对的，任何程序对不同的数据都会给出不同的结果，但在此有个极大的差异，因为状态对象的存在会在很大程度上改变系统的行为。

适应性模型非常强大，但是通常也很难用，因为人们看不到任何定义特定行为的代码。DSL 是有价值的，它提供了一种显式的方式表现代码，这种形式让人们对状态机编程有了感觉。

状态机可以很好地适用于适应性模型，原因在于，它是另一种计算模型。常规的编程语言提供了一种为机器编程的标准思考方式，多数情况下它运作良好。但是，有时，我们需要一些不同的方式，比如“状态机”（第 51 章），“产生式规则系统”（第 50 章），或者“依赖网络”（第 49 章）。使用适应性模型是一种好的方式，它提供了另一种计算模型，DSL 则简化了为这种模型编程的方式。本书稍后会描述“其他一些计算模型”（第 7 章），在那里，你会了解到它们是什么样子，以及如何实现。或许你曾听说，有人把这种使用 DSL 的方式称为声明式编程。

在讨论这个例子时，我采用的流程是：首先构建模型，然后在此基础之上，用 DSL 封装出一个层次，对其进行操作。之所以用这种方式进行描述，是因为我觉得这是一种简单的方式，有助于理解 DSL 如何用于软件开发。虽然模型优先的情况很常见，但它并不是唯一方式。在不同的场景下，我们可能会与领域专家交谈，假定他们可以理解状态机方式。稍后，我们和他们一起工作，创建出他们可以理解的 DSL。在这种情况下，DSL 和模型可以同步构建。

1.5　使用代码生成

在迄今为止的讨论中，要处理 DSL，组装“语义模型”（第 11 章），然后执行语义模型，提供我们希望从控制器得到的行为。在语言圈子里，这种方式称为解释（interpretation）。在解释文本时，会先解析文本，然后程序立刻产生结果。（在软件圈子里，解释是个棘手的词语，它承载了太多含义，然而，这里严格限制为立即执行的形式。）

在语言领域里，与解释相对的是编译。在编译（compilation）时，先解析程序文本，产生中间输出，然后单独处理输出，提供预期行为。在 DSL 的上下文里，编译方式通常指的是代码生成（code generation）。

用状态机解释这个差异有点困难，因此，换用另外一个小例子。想象一下，有某种规则判定人们是否符合某种资格，也许是为了满足保险资格。比如，如图 1-5 所示，一个规则是年龄在 21 ～ 40 岁。这个规则可以是一个 DSL，检查像我这样的候选人是否具备资格。

如果解释，资格判定处理器会解析规则，在执行时加载语义模型，也许是启动时加载。当检查某个候选人时，它会对 这个候选人运行语义模型，获得一个结果。

如图 1-6 所示，在编译的情况下，解析器会加载语义模型，把它当做资格判定处理器构建过程的一部分。在构建期间，DSL 处理器会产生一些代码，这些代码经过编译、打包，并且纳入资格判定处理器，也可能当做某种共享库。然后，运行这段中间代码，对候选人进行评估。

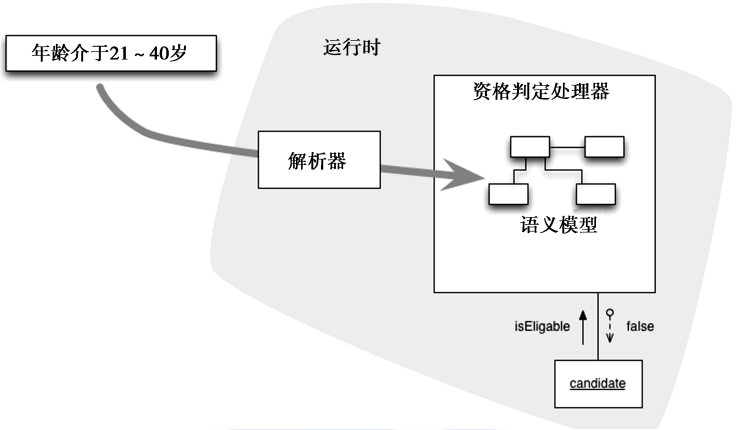

图 1-5　解释器在一个进程中解析文本，产生结果

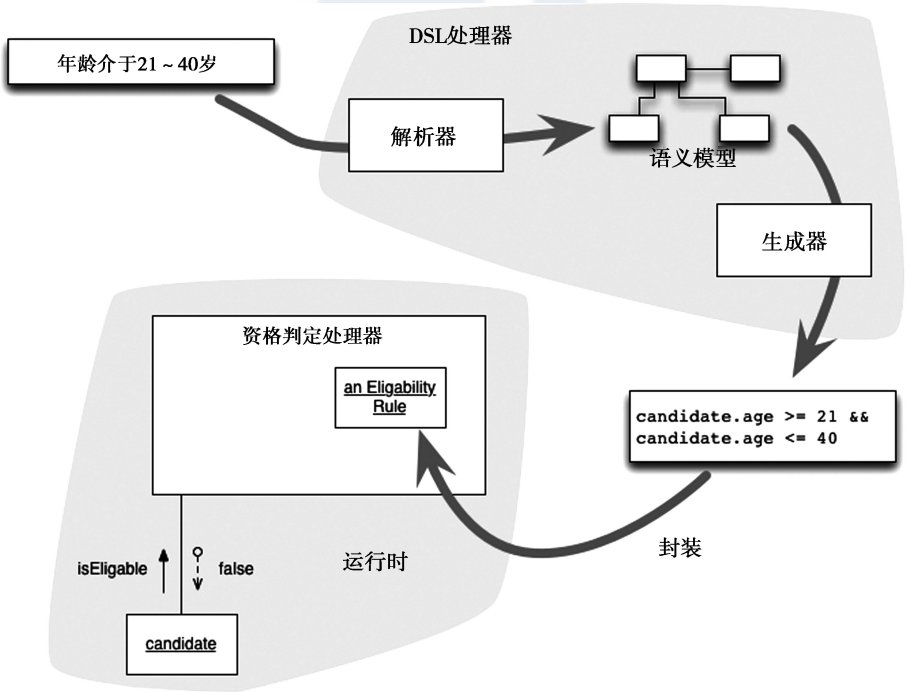

图 1-6　编译器解析文本，产生中间代码，中间代码打包后，由另外的进程执行

例子里的状态机使用的是解释：在运行时解析配置代码，并组成语义模型。但其实也可以生成一些代码，以免在烤面包机里出现解析器和模型代码。

代码生成通常很笨拙，因为它常常需要进行额外的编译步骤。为了构建程序，首先需要编译状态框架和解析器，其次运行解析器，为格兰特小姐的控制器生成源代码，然后编译生成的代码。这样做，构建过程就变得复杂许多。

然而，代码生成的一个优势在于，编写解析器和生成代码可以用不同的语言。在这个情况下，如果生成代码用的是动态语言，比如 JavaScript 或是 JRuby，第二个编译步骤就可以省略。

如果所用 DSL 的语言平台缺乏支持 DSL 的工具，代码生成的作用也会凸显出来。比如，我们不得不在一些老式的烤面包机上运行这个安全系统，而它们又只能理解编译过的 C，那我们可以这样做，实现一个代码生成器，使用组装的语义模型作为输入，产生可以编译为运行在老式烤面包机的 C 代码。在最近做的一些项目里，我们曾为 MathCAD、SQL 和 COBOL 等生成代码。

许多与 DSL 相关的作品都会关注代码生成，更有甚者，把代码生成当做这个活动的主要目标。随之而来的就是，涌现了一大批文章和书籍，赞美代码生成的优点。然而，在我看来，代码生成仅仅是一种实现机制，实际上，大多数情况下 都用不到。当然，有很多情况必须要用代码生成，但的确有很多情况确实用不到。

许多人用了代码生成之后，就舍弃了语义模型，他们在解析输入文本之后，就直接产生生成的代码。虽然对于使用代码生成的 DSL 而言，这也是一种常见的方式，但除非是最简单的情况，否则不推荐任何人这么做。语义模型的存在，可以将解析、执行语义以及代码生成分开。整个活动会因为这个划分变得简单许多。它也给了我们改变自己想法的机会；比如，无须修改代码生成的例程就可以把内部 DSL 改成外部 DSL。类似地，可以很容易产生多种输出，而无须担心解析器变得复杂。就同一种语义模型而言，既可以用解释模型，也可以选择代码生成。

因此，在本书的大部分内容里，假设存在一个语义模型，它是 DSL 工作的核心。

常见的代码生成风格有两种。第一种是“第一遍”代码，这种代码是一个模板，之后要手动修改。第二种是确保生成代码绝对不需要手动修改，也许还要排除调试期间所加的追踪信息。我几乎总是倾向于后者，因为这样可以更自由地重 新生成代码。对 DSL 而言，这点尤其正确，因为我们希望对于 DSL 所定义的逻辑而言，它是主要的表现形式。这意味着，无论何时，要修改行为，必须能够轻松修改 DSL。因此，我们必须保证，任何生成的代码都没有经过手动编辑，虽然它可以调用手写的代码，或者由手写的代码调用。

1.6　使用语言工作台

迄今所述的两种风格的 DSL（内部和外部）是思考 DSL 的一般方式。或许，它们还没有得到广泛理解和运用，虽然应该如此，但是它们拥有很长的历史，也得到了适度的应用。因此，本书余下的部分就是让你在这些方面得到起步，运用那些成熟以及容易得到的工具。

但是还有一类全新的工具已初露端倪，它们也许会极大地改变 DSL 的游戏规则——这种工具称为语言工作台（language workbench）。语言工作台是一个环境，其设计初衷就是帮助人们构建新的 DSL，以及有效地运用这些 DSL 所需的高质量工具。

使用外部 DSL 的一大劣势在于，我们会为相对有限的工具所羁绊。在文本编辑器里设置语法高亮，也就是大多数人所能到达的水平。诚然，你可以争辩，DSL 很简单，脚本规模很小巧，以此说明这样就很好。但还是有人希望拥有支持现代 IDE 的成熟工具。语言工作台不仅让定义解析器变得简单，而且让为这门语言定制一个编辑环境变得简单。

所有这些都是有价值的，但是语言工作台真正有趣的方面在于，它们让 DSL 设计者从传统的基于文本的源码编辑走向不同形式的语言。最明显的一个例子就对图表语言的支持，我们可以通过状态转换图直接指定秘密面板状态机，见图 1-7。

类似于这样的工具不仅可以定义图表语言，还可以从不同的角度来查看 DSL 脚本。在图 1-7 里，我们看到一幅图，一个列表（包括状态、时间），还有 一个表，其中是进入事件的代码（如果认为界面看上去太乱，它是可以从图中省略的）。

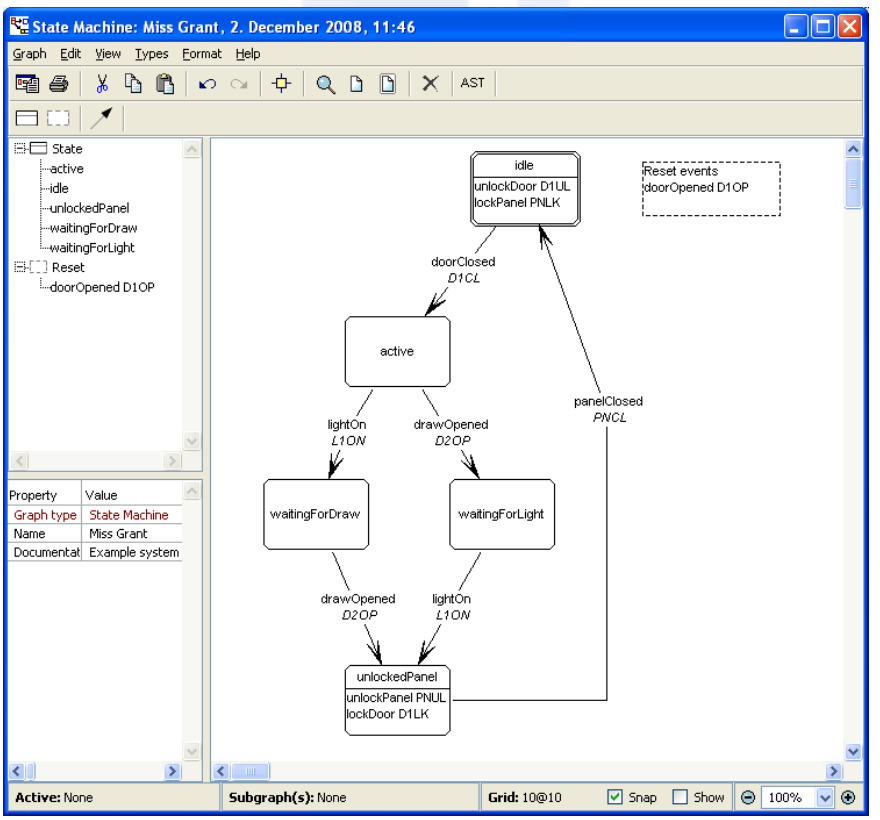

图 1-7　在 MetaEdit 语言工作台里显示的秘密面板状态机（来源：MetaCase）

许多工具都有这种多窗格的可视化编辑环境，但是自己打造一个这样的东西需要很大的工作量。语言工作台要做的一件事就是，让这件事做起来变得相当容易。确实，我第一次上手 MetaEdit 这个工具，就能很快得到像图 1-7 这样的一个例子。这个工具可以让我为状态机定义语义模型，定义图形化和表格化的编辑器，像图 1-7 这样，然后根据语义模型编写代码生成器。

然而，虽然这种工具看上去不错，但许多程序员还是本能地怀疑这种玩具式的工具。有一些非常实际的原因，使得用文本表示代码更有意义。因此，其他工具另辟蹊径，提供一种后 IntelliJ 风格的能力——为基于文本的语言提供类似于语法指导的编辑，自动补全及其他类似功能。

我的怀疑是，如果语言工作台真的流行，其所产生的语言会不同于我们常规理解的编程语言。这种工具的一大益处在于，非程序员也可以编程。对这种想法，我常嗤之以鼻，因为这就是 COBOL 最初的意图。然而，我必须承认，有一个编程环境异常成功，它给非程序员提供了一个编程工具，让这些不认为自己是程序员的人也能编程——电子表格。

许多人并不把电子表格当做编程环境，然而我们可以说，它们是迄今为止最为成功的编程环境。作为一个编程环境，电子表格有一些有趣的特征。第一个有趣的因素就是把工具紧密地集成到编程环境。没有独立于工具的文本表示，也就无须解析器处理。工具和语言紧密地结合与设计在一起。

第二个有趣的因素称为说明性编程（illustrative programming）的东西。看一下电子表格，最为可视化的东西并不是可以进行所有计算的公式；而是构成样本计算的数字。这些数字是一个图示，展现了程序执行时所做的工作。在大多数编程语言里，程序是至关重要的，只有在运行测试时，才关注其输出。在电子表格里，至关重要的是输出，只有在单击单元格时，我们才会看到其程序。

说明性编程并不是一个赢得广泛关注的概念。为了讨论它，我甚至不得不创造出一个词。对于外行程序员而言，这可能是一个非常重要的部分，有了它，他们才得以对电子表格进行操作。它也有劣势，比如，缺乏对程序结构的关注，这样会导致大量复制 – 粘贴编程，以及结构糟糕的程序。

语言工作台支持开发类似于这些全新编程平台。因此，我认为，它们所产生的 DSL 可能更接近于电子表格，而非我们通常理解的 DSL（也就是本书要讨论的内容）。

我认为，语言工作台有着非凡的潜力。如果能够达成目标，它们会完全改变软件开发的面貌。然而这个潜力，虽然深远，但尚在稍远的未来。语言工作台尚处于起步期，新的方式会定期出现，旧有的工具则进一步深化。所以，这里我不会过多地讨论，因为我觉得在本书预期的生命周期里，它们会有剧烈改变。但是，后面确实有一章是讨论它的，因为我觉得它值得关注。

1.7　可视化

语言工作台的一大优势在于它们给了 DSL 更为多样的表现形式，特别是图形化表示。然而，即便是文本化的 DSL 也可以有图形化的表示。确实，我们在本章中非常早就看到这些内容。当查看图 1-1 时，你也许已经注意到了，这个图并不像我以往所画的那些图那样整洁。原因在于，这并不是我画的图，而是我根据格兰特小姐控制器的“语义模型”（第 11 章）自动生成的。状态机类不仅可以执行，还可以用 DOT 语言对自身进行渲染。

DOT 语言是 Graphviz 包的一部分，它是一个开源工具，可以用它描述数学里的图结构（节点和边），然后自动画出来。只要告诉它，什么是节点，什么是边，用什么样的形状，以及其他一些提示，它就会算出如何对这个图进行布局。

对许多 DSL 来说，使用类似于 Graphviz 这样的工具非常有用，因为它给了我们另一种表现形式。类似于 DSL 本身，这种可视化（visualization）表现形式可以让人更好地理解模型。可视化不同于对应的源码，其本身无法编辑——但是，另一方面，它可以完成可编辑形式无法完成的操作，比如渲染出那样的图。

可视化并不一定要图形化。当编写解析器时，我时常用简单的文本可视化帮我调试。我见过有人用 Excel 生成可视化的东西，帮助他们与领域专家交流。重点在于，一旦经过辛勤工作创建出语义模型，添加可视化真的就很容易。注意，可视化是根据模型产生的，而非 DSL，因此，即便不用 DSL 组装模型，依旧可以这么做。

第 2 章

使用 DSL

看过上一章的例子后，即便尚未给出 DSL 的一般定义，对于何为 DSL，你也应该已经有了自己的认识。（第 10 章中有更多例子。）现在，要开始讨论 DSL 的定义及其优势与问题。这样就可以为下一章讨论 DSL 实现提供一些上下文。

2.1 定义 DSL

“领域特定语言”是一个很有用的术语和概念，但其边界很模糊。一些东西很明显是 DSL，但另一些可能会引发争议。这一术语由来已久，不过，正如软件行业中的很多东西一样，它也从未有过一个确切的定义。然而，就本书而言，定义是非常有价值的。

领域特定语言（**名词**）：针对某一特定领域，具有受限表达性的一种计算机程序设计语言。

这一定义包含 4 个关键元素：

- **计算机程序设计语言**（computer programming language）：人们用 DSL 指挥计算机去做一些事。同大多数现代程序设计语言一样，其结构设计成便于人们理解的样子，但它应该还是可以由计算机执行的语言。
- **语言性**（language nature）：DSL 是一种程序设计语言，因此它必须具备连贯的表达能力——不管是一个表达式还是多个表达式组合在一起。
- **受限的表达性**（limited expressiveness）：通用程序设计语言提供广泛的能力：支持各种数据、控制，以及抽象结构。这些能力很有用，但也会让语言难于学习和使用。DSL 只支持特定领域所需要特性的最小集。使用 DSL，无法构建一个完整的系统，相反，却可以解决系统某一方面的问题。
- **针对领域**（domain focus）：只有在一个明确的小领域下，这种能力有限的语言才会有用。这个领域才使得这种语言值得使用。

注意，“针对领域”在这个列表中最后出现，它纯粹是受限表达性的结果。很多人按字面意思把 DSL 理解为一种用于专用领域的语言。但字面意思常常有误：比如，我们不会管硬币叫“光盘”（Compact Disk，紧凑的盘），即便它确实是“盘”，而且相比于可以用这个术语称呼的东西，更为紧凑（compact）。

DSL 主要分为三类：外部 DSL、内部 DSL，以及语言工作台。

- **外部 DSL** 是一种“不同于应用系统主要使用语言”的语言。外部 DSL 通常采用自定

义语法，不过选择其他语言的语法也很常见（XML 就是一个常见选择）。宿主应用的代码会采用文本解析技术对使用外部 DSL 编写的脚本进行解析。一些小语言的传统 UNIX 就符合这种风格。可能经常会遇到的外部 DSL 的例子包括：正则表达式、SQL、Awk，以及像 Struts 和 Hibernate 这样的系统所使用的 XML 配置文件。

- **内部 DSL** 是一种通用语言的特定用法。用内部 DSL 写成的脚本是一段合法的程序，但是它具有特定的风格，而且只用到了语言的一部分特性，用于处理整个系统一个小方面的问题。用这种 DSL 写出的程序有一种自定义语言的风格，与其所使用的宿主语言有所区别。这方面最经典的例子是 Lisp。Lisp 程序员写程序就是创建和使用 DSL。Ruby 社区也形成了显著的 DSL 文化：许多 Ruby 库都呈现出 DSL 的风格。特别是，Ruby 最著名的框架 Rails，经常被认为是一套 DSL。
- **语言工作台**是一个专用的 IDE，用于定义和构建 DSL。具体来说，语言工作台不仅用来确定 DSL 的语言结构，而且是人们编写 DSL 脚本的编辑环境。最终的脚本将编辑环境和语言本身紧密结合在一起。

多年来，这三种风格分别发展了自己的社区。你会发现，那些非常擅长使用内部 DSL 的人，完全不了解如何构造外部 DSL。我担心这可能会导致人们不能采用最适合的工具来解决问题。我曾与一个团队讨论过，他们采用了非常巧妙的内部 DSL 处理技巧来支持自定义语法，但我相信，如果他们使用外部 DSL 的话，问题会变得简单许多。但由于对如何构造外部 DSL 一无所知，他们别无选择。因此，在本书中，把内部 DSL 和外部 DSL 讲清楚对我来说格外重要，这样你就可以了解这些信息，做出适当的选择。（语言工作台稍显粗略，因为它们很新，尚在演化之中。）

另一种看待 DSL 的方式是：把它看做一种处理抽象的方式。在软件开发中，我们经常会在不同的层面上建立抽象，并处理它们。建立抽象最常见的方式是实现一个程序库或框架。操纵框架最常见的方式是通过命令 / 查询式 API 调用。从这种角度来看，DSL 就是这个程序库的前端，它提供了一种不同于命令 / 查询式 API 风格的操作方式。在这样的上下文中，程序库成了 DSL 的“语义模型”（第 11 章），因此，DSL 经常伴随着程序库出现。事实上，我认为，对于构建良好的 DSL 而言，语义模型是一个不可或缺的附属物。

谈及 DSL，人们很容易觉得构造 DSL 很难。实际上，通常是难在构造模型上，DSL 只是位于其上的一层而已。虽然让 DSL 工作良好需要花费一定的精力，但相对于构建底层模型，这一部分的付出要少多了。

2.1.1　DSL 的边界

前面说过，DSL 是一种边界模糊的概念。虽然我相信没有人会质疑正则表达式是一种 DSL，但确实有许多存在争议的情况。因此我觉得有必要在这里讨论一下这些情况，让我们更好地理解什么是 DSL。

每种风格的 DSL 都有各自的边界条件，下面会分别讨论。当讨论这些内容时，请记住，用于区分各种 DSL 特征的是语言性、针对领域以及受限表达性。根据经验，按照“针对领域”进行划分效果欠佳，因此，常常选择按语言性以及受限表达性进行划分。

我将从内部 DSL 开始。这里的边界问题，其实就是内部 DSL 与命令 / 查询式 API 之间的差异。从许多方面来说，内部 DSL 不过是一种特殊的 API（就像出自贝尔实验室的一句老话“程序库设计就是语言设计”）。在我看来，核心差异在于语言性。Mike Roberts 曾说过，命令 / 查询式 API 定义了抽象领域的词汇，而内部 DSL 则在此基础上添加了语法。

给具有命令 / 查询式 API 的类编写文档，一种常见的方式是，列出其拥有的所有方法。采用这种方法，就意味着每个方法自身都有独立含义。从这样的文档中，我们得到了一组“词汇”，每一个都有自己的完备语义。而内部 DSL 的方法名只在一个更大表达式的上下文中才有明确的含义。在前面 Java 内部 DSL 的例子中，有一个名为 `to` 的方法，定义了状态迁移的目标。这样的方法名在命令 / 查询式 API 中是一个不好的名字，但适用于这样的语句：`.transition(lightOn).to(unlockedPanel)`。

正因为这样，内部 DSL 给人的感觉是一个整句，而非一个无关命令的序列。这正是这种 API 称为连贯接口的基础。

对内部 DSL 来说，受限表达性显然不是语言的一项核心属性，因为内部 DSL 植根于一种通用语言。在这种情况下，受限表达性表现在如何使用它。当构造 DSL 表达式时，我们会限制自己只使用通用语言的一部分特性，通常不会使用条件判断、循环结构和变量。Piers Cawley 将其称为宿主语言的一种混杂用法（pidgin use）。

对外部 DSL 来说，其边界就是它与通用语言之间的边界。语言可以针对某领域，但仍是通用语言。R 语言就是一个很好的例子，它是一种用于统计的语言和平台，主要用于解决统计方面的问题，它也具备通用语言所有的表达性。因此，尽管它针对于某一领域，但是我们依然不会称其为一种 DSL。

一种更为明显的 DSL 是正则表达式。它所针对的领域（文本匹配）与其有限的特性紧密相关——那些特性刚刚好能做到易于匹配文本。DSL 的一个普遍特征是，它们不是图灵完备的。一般来说，DSL 不支持常见的命令式控制结构（条件和循环），也不能定义变量和子例程。

说到这里，可能会有很多人不同意我的看法，他们会觉得，从 DSL 的字面语义来说，像 R 这样的语言应该归为 DSL。我之所以如此强调 DSL 的有限表达性，是因为正是它使得 DSL 和通用语言之间的区别有了意义。有限表达性让 DSL 产生了独特性，无论是使用还是实现，这种独特性给了我们完全不同于通用语言的 DSL 思维方式。

如果这样的边界还不够模糊，我们再来看看 XSLT。XSLT 针对的领域是 XML 文档转换，但是它具备我们预期在一门常规程序设计语言中看到的所有特性。这样一来，XSLT 是什么样的语言已经不重要了，重要的是人们如何使用它。如果 XSLT 用于转换 XML 文档，我愿意把它称之为 DSL；但如果用以解决“八皇后”（eight queens）问题，则我认为它是一门通用语言。一门语言的特定用法可能将它置于 DSL 分界线的任意一边。

外部 DSL 同序列化的数据结构之间也存在一条边界。配置文件中的属性赋值（如 color=blue）列表算 DSL 吗？在这种情况下，边界条件是 DSL 的语言性。因为一系列赋值表达式缺乏表达上的连贯性，所以它不符合标准。

同样的理由也适用于许多配置文件。如今许多环境为用户提供了对配置文件编程的能力，这些配置文件通常采用 XML 格式。这些 XML 配置文件在很多情况下足以成为 DSL。然而，并非总是如此。有时，XML 文件是由工具生成的，此时 XML 只是一种序列化的手段，而非为人所用，因此我不会将其归为 DSL。当然，一种存储格式具备易读性肯定是有价值的，毕竟它有利于调试。其实，问题不在于 XML 是否可读，而在于其表示方式是否是人们与系统某一方面交互的主要方式。

这种配置文件的最大问题在于，虽然它们不是用来人工编辑的，但在实际中，人们却经常人工编辑。于是这种 XML 文档就意外地成为了 DSL。

对语言工作台来说，语言工作台同任意程序之间的边界在于，它允许用户设计自己的数据结构和表单——就像 Microsoft Access。毕竟，随便拿来一个状态模型，都可以用关系数据库结构来表示它（我还见过比这更糟糕的主意），然后就能创建表单，操作模型。这里有两个问题：1）Access 是一种语言工作台吗？ 2）在 Access 里定义的是一 种 DSL 吗？

先看第二个问题。既然在构建状态机的一个特定应用，我们就有了领域针对性和受限表达性，关键的问题就在于语言性。如果只是把数据放进表单，然后保存在表格里，这通常不是一种真正的语言。表格可以是语言性的一种表示法——“FIT”（10.6 节）和 Excel 都采用了表格式的表示法，又都给人一种语言的感觉（我会把 FIT 当做是领域专用的，而 Excel 是通用的）。但是绝大部分应用不会追求这种连贯性，它们只是创建表单和窗口，而不强调关联性。例如，Meta-Programming System Language Workbench 的文本界面，感觉上迥异于大部分基于表单的界面。类似地，很少有应用像 MetaEdit 那样让人通过排列图表来定义事物的整合方式。

至于 Acess 是否是一种语言工作台，我想我们可以回到其原始的设计意图上。如果真的需要，我们确实可以把它当做语言工作台用，但它并非设计成这样。想想有多少人把 Excel 当做数据库——即便它也没有设计成那样。

从更广泛的意义上说，一种纯粹在人与人之间使用的行话是否是一种 DSL 呢？ 一个广为流传的例子是人们在星巴克点咖啡所用的语言：“超大杯、低咖、脱脂、无泡沫、不搅拌的拿铁。”这种语言很不错，因为它有受限的表达性、针对领域、有词汇和语法的感觉。但它在我的定义之外，因为“领域特定语言”只用于指代计算机语言。如果要实现一种理解星巴克表达方式的计算机语言，它就真成为 DSL 了。但在我们要补充咖啡因时，从我们脱口而出的措辞是一种人类语言。这里把人们在特定领域中使用的语言称为领域语言（domain language），而 把“DSL”保留给计算机语言。

那么，这个关于 DSL 边界的讨论告诉了我们什么呢？我希望，至少有一件事情是清楚的：那就是基本上没有明确的边界。追求理性的人可以不认同我的 DSL 定义。事实上，因为

像语言性和受限表达性这样的衡量标准本身就很模糊，所以可以推断出，基于这些标准得到的结果也会同样模糊。而且，并非所有人都会采纳我的这些衡量标准。

在上面的讨论中，许多例子都被排除在 DSL 的定义之外，但这不代表它们没有价值。定义的价值在于它有助于沟通，这样，不同背景的人对于要讨论的内容有个共同的认识。对本书来说，它帮我们理清这里描述的技术是否与之相关。对我来说，有了这样的 DSL 定义之后，我就能更有效地选择一些需要讨论的技术。

2.1.2　片段 DSL 和独立 DSL

第 2 章用的那个秘密面板状态机的例子是一种独立的 DSL。这意味着，随便拿一段这种 DSL 脚本（一 般是一个文件）来看，其中全是这种 DSL。只要熟悉这种 DSL，我们就能够理解这段 DSL 在做什么，即便不了解宿主语言，因为根本就没有宿主语言（对外部 DSL 而言），或者为内部 DSL 掩盖。

DSL 出现的另一种方式是片段形式。对于这种形式，DSL 片段用于宿主语言的代码中。我们可以将其视为采用额外特性对宿主语言进行增强。在这种情况下，不了解宿主语言，就看不懂这些 DSL 在做些什么。

对外部 DSL 而言，片段 DSL 的一个典型例子是正则表达式。我们不会见到一个充斥着正则表达式的程序文件，相反，却可以见到在一个程序中点缀着正则表达式片段。还有一个例子是 SQL，在大型程序的上下文中，常常会用到 SQL 语句。

内部 DSL 也有使用片段形态的情况。单元测试领域是内部 DSL 的高产区。特别是在 mock 对象程序库中设置预期的语法，它就是在大规模宿主语言里 DSL 的集中爆发。此外，用于内部片段 DSL 的一个流行语言特性是标注，可以用它给宿主代码中的编程元素添加元数据。这种用法使注解适用于片段 DSL，而非独立 DSL。

还有同样的 DSL 既可以独立使用，也可以作为片段使用，比如 SQL。有些 DSL 设计成以片段形式使用，有些旨在独立使用，还有一些则二者均可。

2.2　为何需要 DSL

至此，我希望，对什么是 DSL，我们已经有了一个很好的共识，接下来的问题是，为何要考虑采用 DSL。

DSL 只是一种工具，关注点有限，无法像面向对象编程或敏捷方法论那样，引发软件开发思考方式的深刻变革。相反，它是在特定条件下有专门用途的一种工具。一个普通的项目可能在多个地方采用了多种 DSL ——事实上很多项目这么做了。

在 1.4 节中，一直强调，DSL 只是模型的一个薄壳，这个模型可能是程序库，也可能是框架。这句话提醒我们，当考虑 DSL 的优劣时，一定要分清它是来自 DSL 的模型，还是 DSL 本身。经常有人将二者混淆。

DSL 有其自身的价值。当考虑采用 DSL 时，要仔细衡量它的哪些价值适合于我们的情况。

2.2.1　提高开发效率

DSL 的核心价值在于，它提供了一种手段，可以更加清晰地就系统某部分的意图进行沟通。拿格兰特小姐控制器的定义来说，相比于采用命令 – 查询 API，DSL 形式对我们而言更容易理解。

这种清晰并非只是审美追求。一段代码越容易看懂，就越容易发现错误，也就越容易对系统进行修改。因此，我们鼓励变量名要有意义，文档要写清楚，代码结构要写清晰。基于同样的理由，我们应该也鼓励采用 DSL。

人们经常低估缺陷对生产率的影响。缺陷不仅损害软件的外部质量，还浪费开发人员的时间去调查以及修复，降低开发效率，并使系统的行为异常，播下混乱的种子。DSL 的受限表达性，使其难于犯错，纵然犯错，也易于发现。

模型本身可以极大地提升生产率。通过把公共代码放在一起，它可以避免重复。首先，它提供了一种“用于思考问题”的抽象，这样，更容易用一种可理解的方式指定系统行为。DSL 提供了一种“对阅读和操作抽象”更具表达性的形式，从而增强了这种抽象。DSL 还可以帮助人们更好地学习使用 API，因为它将人们的关注点转移到怎样将 API 方法整合在一起。

我还遇到过一个有趣的例子，使用 DSL 封装一个棘手的第三方程序库。当命令 – 查询接口设计得很糟糕时，DSL 惯常的连贯性就得以凸现。此外，DSL 只须支持客户真正用到的部分，这大大降低了客户开发人员学习的成本。

2.2.2　与领域专家的沟通

我相信，软件项目中最困难的部分，也是项目失败最常见的原因，就是开发团队与客户以及软件用户之间的沟通。DSL 提供了一种清晰而准确的语言，可以有效地改善这种沟通。

相比于关于生产率的简单争论，改善沟通所带来的好处显得更加微妙。首先，很多 DSL 并不适用于沟通领域问题，比如，用于正则表达式或构建依赖关系的 DSL，在这些情况下就不合适。只有一部分独立 DSL 确实应用这种沟通手段。

当在这样的场景下讨论 DSL 时，经常会有人说：“好吧，现在我们不需要程序员了，领域专家可以自己指定业务规则。”我把这种论调称为“COBOL 谬论”——因为 COBOL 曾被人寄予这样的厚望。这种争论很常见，不过，我觉得这种争论不值得在此重复。

虽然存在“COBOL 谬论”，我仍然觉得 DSL 可以改善沟通。不是让领域专家自己去写 DSL，但他们可以读懂，从而理解系统做了什么。能够阅读 DSL 代码，领域专家就可以指出问题所在。他们还可以同编写业务规则的程序员更好地交流，也许，他们还可以编写一些草稿，程序员们可以将其细化成适当的 DSL 规则。

但我不是说领域专家永远不能编写 DSL。我遇见过很多团队，他们成功地让领域专家用 DSL 编写了大量系统功能。但我仍然认为，使用 DSL 的最大价值在于，领域专家能够读懂。所以编写 DSL 的第一步，应该专注于易读性，这样即便后续的目标达不到，我们也不会失去什么。

使用 DSL 是为了让领域专家能够看懂，这就引出了一个值得争议的问题。如果希望领域专家理解一个“语义模型”（第 11 章）的内容，可以将模型可视化。这时就要考虑一下，相比于支持一种 DSL，是不是只使用可视化会是一种更有效的办法。可视化对于 DSL 而言，是一种有益的补充。

让领域专家参与构建 DSL，与让他们参与构建模型是同样的道理。我发现，与领域专家一起构建模型能够带来很大的好处，在构建一种 Ubiquitous Language [Evans DDD] 的过程中，程序员与领域专家之间可以深入沟通。DSL 提供了另一种增进沟通的手段。随着项目的不同，我们可能发现，领域专家可能会参与模型和 DSL，也可能只参与 DSL。

实际上，有些人发现，即便不实现 DSL，有一种描述领域知识的 DSL，也能带来很大的好处。即使只把它当做沟通平台也可以获益。

总的来说，让领域专家参与构建 DSL 比较难，但回报很高。即使最终不能让领域专家参与，但是开发人员在生产率方面的提升，也足以让我们大受裨益，因此，DSL 值得投入。

2.2.3　执行环境的改变

当谈及将状态机表述为 XML 的理由时，一个重要的原因是，状态机定义可以在运行时解析，而非编译时。在这种情况下，我们希望将代码运行于不同的环境，这类理由也是使用 DSL 一个常见的驱动力。对于 XML 配置文件而言，将逻辑从编译时移到运行时就是一个这样的理由。

还有一些需要迁移执行环境的情况。我曾见过一个项目，它要从数据库里找出所有满足某种条件的合同，给它们打上标签。他们编写了一种 DSL，以指定这些条件，并用它以 Ruby 语言组装“语义模型”（第 11 章）。如果用 Ruby 将所有合同读入内存，再运行查询逻辑，那会非常慢，但是团队可以用语义模型的表示生成 SQL，在数据库里做处理。直接用 SQL 编写规则，对开发人员都很困难，遑论业务人员。然而，业务人员可以读懂（在这种情况下，甚至编写）DSL 里有关的表达式。

这样用 DSL 常常可以弥补宿主语言的局限性，将事物以适宜的 DSL 形式表现出来，然后，生成可用于实际执行环境的代码。

模型的存在有助于这种迁移。一旦有了一个模型，或者直接执行它，或者根据它产生代码都很容易。模型可以由表单风格的界面创建，也可以由 DSL 创建。DSL 相对于表单有一些优势。在表述复杂逻辑方面，DSL 比表单做得更好。而且，可以用相同的代码管理工具，比如版本控制系统，管理这些规则。当规则经由表单输入，存入数据库中，版本控制就无能为力了。

下面会谈及 DSL 的一个伪优点。我听说，有人宣称 DSL 的一个好处是，它能够在不同的语言环境下执行相同的行为。一个人编写了业务规则，然后生成 C# 或 Java 代码，或者，描述校验逻辑之后，在服务器端以 C# 形式运行，在客户端则是 JavaScript。这是一个伪优势，因为仅仅使用模型就可以做到这一点，根本无需 DSL。当然，DSL 有助于理解这些规则，但那是另外一个问题。

2.2.4　其他计算模型

几乎所有主流的编程语言都采用命令式的计算模型。这意味着，我们要告诉计算机做什么事情，按照怎样的顺序来做。通过条件和循环处理控制流，还要使用变量——确实，还有很多我们以为理所当然的东西。命令式计算模型之所以流行，是因为它们相对容易理解，也容易应用到许多问题上。然而，它并不总是最佳选择。

状态机是这方面的一个良好例子。可以采用命令式代码和条件处理这种行为，也确实可以很好地构建出这种行为。但如果直接把它当做“状态机”来思考，效果会更好。另外一个常见的例子是，定义软件构建方式。我们固然可以用命令式逻辑实现它，但后来，人们发现用“依赖网络”（第 49 章）（比如，运行测试必须依赖于最新的编译结果）解决会更容易。结果，人们设计出了专用于描述构建的语言（比如 Make 和 Ant），其中将任务间的依赖关系作为主要的结构化机制。

你可能经常听到，人们把非命令式方式称为声明式编程。之所以叫做声明式，是因为这种风格让人定义做什么，而不是用一堆命令语句来描述怎么做。

采用其他计算模型，并不一定非要有 DSL。其他编程模型的核心行为也源自“语义模型”（第 11 章），正如前面所讲的状态机。然而，DSL 还是能够带来很大的转变，因为操作声明式程序，组装语义模型会容易一些。

2.3　DSL 的问题

前面已经讨论了何时该采用 DSL，接下来就该谈论什么时候不该采用 DSL，或者至少是使用 DSL 应注意的问题。

从根本上说，不使用 DSL 的唯一原因就是，在你的场景下，使用 DSL 得不到任何好处，或者，至少是 DSL 的好处不足以抵消构建它的成本。

虽然 DSL 在有些场合下适用，但同样会带来一些问题。总的来说，我认为通常是高估了这些问题，一般人们不太熟悉如何构造 DSL，以及 DSL 如何适应更为广阔的软件开发图景。还有，许多常提及的 DSL 问题混淆了 DSL 和模型，这也伤及了 DSL 的优势。

许多 DSL 问题只是与某种特定 DSL 风格相关，要理解这些问题，我们需要深入理解这些 DSL 是如何实现的。所以，这些问题留待后面讨论，在这里，我们只看宽泛的问题，这同当前讨论的问题是一致的。

2.3.1　语言噪音

在反对 DSL 的观点中，我最常听到的是称为语言噪音的问题：担心语言难于学习，因此，使用多种语言会比使用一种语言复杂得多。必须了解多种语言，会让工作更为困难，新人加入的门槛也提升了。

当人们谈及这种担心时，他们都会有一些共同的误解。首先，他们通常混淆了学习一门 DSL 的心血与学习一门通用语言的心血。DSL 远比一门通用语言容易，因此，学习起来也要容易得多。

许多批评者知道这一点，但依然反对 DSL，即便它们相对容易学习，在一个项目上有多种 DSL 也增加了理解的难度。这里的误解在于，他们忘了一点，一个项目总有一些复杂的地方，难于学习。即便不用 DSL，代码库中仍然有许多需要理解的抽象。通常，这些抽象应该在程序库里，以便于掌握。即使不必学习多种 DSL，也不得不学习多个程序库。

所以，真正的问题在于，相比于学习 DSL 底层模型而言，学习 DSL 会难多少。我认为，相对于理解模型而言，学习 DSL 所增加的成本相当小。确实，因为 DSL 的的价值就在于，让人们理解和使用模型更容易，所以使用 DSL 就应该能降低学习成本。

2.3.2　构建成本

相对于底层的程序库而言，DSL 增加的成本并不大，但这始终是成本。代码需要写，尤其是还要维护。所以，同其他代码一样，它也要做好自己的本职工作。并非所有程序库都值得用 DSL 封装。如果命令 – 查询 API 够用，就没有必要在上面提供额外的 API。即便 DSL 有用，就边界效应而言，构建和维护也需要花费太多的工作量。

DSL 的可维护性是一项重要的考量因素。如果团队中的大多数人都觉得难以理解，即使是一种简单的内部 DSL，也会带来很大的麻烦。外部 DSL 更是让许多人望而却步，一个解释器就足以让很多程序员打退堂鼓。

人们不习惯构建 DSL，这也让添加 DSL 的成本变得更高。人们要学习新技术。虽然不应该忽略这些成本，但我们也应该清楚，这个学习曲线的成本能够分摊到未来使用 DSL 的过程中。

还有一点要清楚，DSL 的成本大于构建模型的成本。任何复杂的地方都需要某种机制管理其复杂性，如果复杂到要考虑 DSL，几乎肯定复杂到可以从模型中获益的程度。DSL 有助于思考模型，降低构建成本。

这会带来一个相关问题，鼓励使用 DSL 会导致构建出一堆糟糕的 DSL。实际上，我盼着构建出一堆糟糕的 DSL，就像有很多糟糕的命令 – 查询 API 的程序库一样。问题在于，DSL 会不会把事情弄得更糟。一个好的 DSL 可以封装一个糟糕的程序库，把它变得更易用（如果可能的话，我更愿意修正程序库本身）。糟糕的 DSL 对于构建和维护而言，就是浪费资源，但这种说法对任何代码都适用。

2.3.3　集中营语言

集中营语言（ghetto language）问题与语言噪音问题正好相反。比如，一家公司用一种内部语言编写公司内的很多系统，这种语言在其他地方根本用不上。这种做法会让他们很难找到新人，跟上技术变化。

在分析这个问题时，首先要澄清一点，如果整个系统都是用一种语言编写的，那它就不是一种 DSL（至少按我的定义），而是一种通用语言。虽然可以用许多 DSL 技术构建通用语言，但我强烈建议，不要这样做。构建和维护一种通用语言是一个巨大的负担，它会迫使你在这个集中营中做大量工作，甚至挣扎一生。不要这么做。

我相信，集中营语言问题并非空穴来风，它隐含了一些现实问题。首先是，一种 DSL 总是存在着无意中演化成一种通用语言的危险。我们有一种 DSL，然后，逐步为它添加新功能；今天添加条件表达式，明天又添加循环，最终图灵完备了。

对此，唯一的抵御就是坚决防范。确保我们对 DSL 针对问题的受限范围有个清晰的认识。质疑任何不在此范畴内的新特性。如果想做得更多，可以考虑采用多种语言，综合运用，而非强求一种 DSL 不断膨胀。

框架也面临着同样的问题。好的程序库都有一个明确的目的。如果产品定价库包含 HTTP 协议的实现，从本质上说，我们也就要忍受同样错误之苦：未能分离关注点。

第二个问题是，自行构造本应从外部获得的东西。这个问题同样适用于 DSL 和程序库。比如，如今，很少有要自己构造对象 – 关系映射（object–relational mapping）系统。我有一条关于软件的通用规则，不是自己的业务，不要自己写——总要先看看是否从别的地方可以找到。特别是，随着开源工具的崛起，基于既有开源工作量进行扩展，肯定比从头打造更有意义。

2.3.4　“一叶障目”的抽象

DSL 的有用之处在于，它提供了一种抽象，我们可以基于这种抽象来思考领域问题。这种抽象非常有价值，我们更容易表述领域行为，效果远胜于依据底层构造进行思考。

然而，任何抽象（包括 DSL 和程序库）总是伴随着风险——它可能让我们“一叶障目，不见泰山”。有了这种“一 叶障目”的抽象，我们就会苦苦思索，竭尽全力把外部世界塞入抽象之中，而非另寻它路。我们常常会见到这种情况：遇到一种不符合抽象的事物，殚精竭虑地让其符合，而不是修改抽象，让抽象更容易接纳新的行为。一旦我们满意了这个抽象，觉得尘埃落定，“一叶障目”也就随之而来。到这种时候，对于颠覆性的变化，难免心生忧虑。

“一叶障目”是任何抽象都会面临的问题，不仅是 DSL，但 DSL 可能让这个问题变得更严重。因为 DSL 提供了一种更为舒适的方式操作抽象，一旦适应，更不愿意做出改变。如果采用 DSL 与领域专家交流，问题可能会更严重，通常，他 们在习惯之后更不愿意做出改变。

如同对待任何抽象一样，应该视 DSL 为一种“不断演化，尚未完结”的事物。

2.4　广义的语言处理

本书是关于领域专用语言的，但它也涉及语言处理技术。之所以二者重合，是因为在普通的开发团队里，用到语言处理技术的情况，90% 都是为了 DSL。但是，这些技术也可以用于其他方面，若不讨论这些情况，将是我的失职。

我曾遇到过这方面一个很好的例子，那是在一次拜访 ThoughtWorks 项目团队时。他们有一个任务，要与某第三方系统通信，发送的消息以 COBOL copybook 定义。COBOL copybook 是一种用来描述记录的数据结构格式。因为系统中有很多地方要用到，所以我的同事 Brian Egge 决定编写一个解析器，支持 COBOL copybook 语法的子集，为要连接这些记录生成 Java 类。一旦解析器写好，他就可以很高兴地连接所需的 copybook。其他代码无须了解 COBOL 数据结构，一旦有变化，只要简单地重新生成即可。很难将 COBOL copybook 称为一种 DSL，但是，可以用与处理外部 DSL 同样的技术解决这个问题。

因此，虽然我是在 DSL 的上下文中讨论这些技术，但不妨碍你把它们用在其他地方。一旦掌握了关于语言处理的概念，就可以在许多地方用到它们。

2.5　DSL 的生命周期

为了介绍 DSL，开篇先描述一个框架，及其命令 – 查询 API，基于这个 API，构建一层 DSL 以简化操作。我用这种方式，是因为我觉得这种方式有助于理解 DSL，但这并不是人们在实际中使用 DSL 的唯一方式。

另一种常见的方式是先定义 DSL。在这种模式下，可以先从一些场景开始，按照期望 DSL 的样子，把这些场景写下来。如果语言是业务功能的一部分，最好和领域专家一起做——这是一个好的开始，使用 DSL 作为一种沟通媒介。

有人喜欢从语句开始，他们期待这些语句能够语法正确。这意味着，对内部 DSL，要符合宿主语言的语法；对外部 DSL，语句要能够解析。其他 DSL 开始时比较非正式，然后再对照 DSL 进行修改，以得到一种合理的语法。

以这种方式实现状态机，我们要和了解客户需求的人一起坐下来，想出一套关于控制器行为的例子，这些例子可以基于人们过去的需求，也可以基于我们对他们期望的理解。对于每个例子，尝试用 DSL 的形式把它们写下来。随着处理到不同的情况，我们就要修改 DSL，支持新的能力。最后，我们就会得到一套合理的用例，以及对这些用例的伪 DSL 描述。

如果用的是语言工作台，就要在工作台之外完成这一阶段，用一个纯文本编辑器，或者一个普通的绘图软件，也可以是纸和笔。

一旦有了一套典型的伪 DSL，就可以着手实现它了。这里的实现包括以宿主语言设计的

状态机模型、模型的命令－查询 API、DSL 的具体语法以及 DSL 和命令－查询 API 之间的转换。实现方式有很多种。有人喜欢一次做一点，横跨所有元素：构建少量模型、添加 DSL 驱动模型，然后用测试把所有东西串起来；也有人倾向于先构建和测试整个框架，然后在其上构建一层 DSL；还有人喜欢先准备好 DSL，然后构建程序库，再使它们适配。作为一个增量主义者，我倾向于端到端地实现薄薄几层的功能，因此，我会采用三种做法中的第一种。

我会从我所见的最简单的一个用例出发。采用测试驱动开发的方式，编写一个支持这个用例的程序库。然后，通过 DSL 实现这个用例，把它同之前构建的框架连起来。我很乐于对 DSL 做出一些修改，使其易于修改，当然，我会拿着这些修改与领域专家沟通，确保我们对于共同的沟通媒介有同样的理解。做完一个，我会继续做下一个。就这样，我逐渐演化着框架、测试优先，然后逐渐演化着 DSL。

这并不是说模型优先的路线是一个糟糕的选择，事实上，这种做法往往可以做得很漂亮。模型优先常常发生在一开始没有考虑 DSL，或者不确定是否需要 DSL 的情况下。这时，我们会先构建框架，用了一段时间之后，我们觉得 DSL 是一 个有益的补充。在这个例子里，已经有一个可用的状态机模型，而且许多客户已经开始使用。这时，我们意识到，要增加一个新客户困难重重，因此决定尝试 DSL。

基于模型发展 DSL 的方法有两种。对于“语言生长”（language–seeded）的方式，要慢慢地在模型之上构建 DSL，把模型几乎视为黑盒。首先看看目前所有的控制器，然后草拟出每个控制器的伪 DSL。然后，就像前面提及的情况那样，一个场景一个场景地实现 DSL，通常，我们不会对模型做任何深入的修改，尽管给模型添加一些方法能够更好地支持 DSL。

对于“模型生长”（model–seeded）的方式，要先给模型加入一些连贯方法（fluent method），让模型更易于配置，然后逐渐把它们抽成 DSL。这种方式更适用于内部 DSL，可以视之为模型的一次重量级重构，派生出内部 DSL。“模型生长”的方式最吸引人的方面在于，它是逐步进行的，构建 DSL 并不需要显著的成本。

当然，在很多情况下，我们甚至连框架都没有。写了几个控制器之后，我们才意识到，有很多通用功能。然后，就会重构系统，拆分模型与控制代码。这个拆分是至关重要的一步。虽然做这件事时，脑海中已经有了 DSL，但我依然倾向于首先完成拆分，然后在其上构建 DSL。

至此，我要强调一件事，希望我的担心是多余的。一定要保证所有的 DSL 脚本都保存在某种形式的版本控制系统里。DSL 脚本是代码的一部分，所以，就像其他东西一样，它也应该放到版本控制里。文本化 DSL 的一大优势在于，它们可以很好地与版本控制系统协作，也就很容易跟踪系统行为的变化。

2.6　设计优良的 DSL 从何而来

审核本书的人常常问我，有没有一些“设计优良语言”的技巧。毕竟，语言设计很特

别，我们不希望这个世界上充斥着糟糕的语言。我想分享一些好的建议，但我坦白，真没什么好办法。

如同任何写作一样，DSL 的总体目标就是对读者要清晰。我们希望本书的典型读者，可能是程序员，抑或是领域专家，能够尽可能快速清晰地理解 DSL 里句子的意图。虽然我觉得可能无法告诉你如何做到这一点，但是我的确觉得在工作中牢记这一点非常有价值。

总的说来，我是迭代设计的粉丝，这次也不例外。尽早从目标受众那里获得反馈。准备多种方案，看看人们的反应。要得到一种好的语言，总会经历尝试和失败，不要怕走弯路。尝试得越多，越有可能走上正确的道路。

不要担心在 DSL 以及“语义模型”（第 11 章）里用到领域中的术语。如果 DSL 的用户熟悉这些术语，他们就应该在 DSL 里看见。术语有助于增进领域内的沟通，即便对外人来说有些奇怪。

此外，还要记得采用我们日常生活里的通用约定。如果每个人都用 Java 或 C#，就用“//”表示注释，用“{”和“}”表示分层结构。

还有一点，我觉得需要特别警示一下：不要试图让 DSL 读起来像自然语言。为了这个效果，有人用通用语言做过各种尝试，Applescript 是其中最著名的例子。问题在于，这种尝试会引入很多语法糖，这会增加语义理解的难度。记住，DSL 是一种程序设计语言，所以，使用它感觉应该像编程，同自然语言相比，程序设计语言应该更加简洁且准确。尝试让程序设计语言看上去像自然语言，只会让大脑陷入错误的上下文；同程序打交道时，务必记住，你得按程序的规矩办事儿。

第③章

实现 DSL

至此，对于什么是 DSL，以及为何要用 DSL，我们已经透彻理解。如果要开始构建 DSL，那么现在该深入研究所用的技术了。虽然构建内部 DSL 和外部 DSL 所用的技术有所不同，但它们还是有一些共通之处的。本章主要关注内部 DSL 和外部 DSL 的一些共通问题，而下一章再讨论各自具体的问题。本章先不谈语言工作台，留待后续探讨。

3.1 DSL 处理之架构

关于 DSL 实现的大体结构（见图 3-1），也就是所谓的 DSL 系统架构——可能是我们要谈论的最重要的内容之一。

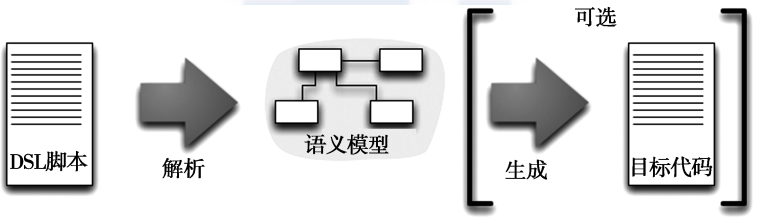

图 3-1 我喜欢的 DSL 处理的总体架构

迄今为止，你应该已经厌倦了听我说了无数次的“DSL 是模型上面薄薄的一层结构”。这里所说的“模型”，称为“语义模型”（第 11 章）模式。这个模式背后的概念是：所有重要的语义行为都可以在模型中捕获，而 DSL 的任务就是通过解析来填充模型。所以，根据我的理解，语义模型在 DSL 中扮演着核心角色——事实上，全书都会首先假设，我们在使用语义模型。（当然，在本节的最后，在我们有足够的上下文去讨论它们的章节，我会谈谈语义模型的替代方案。）

因为我是一个面向对象偏执狂，所以我理解的语义模型首先是一个对象模型。我喜欢既有数据又有行为的多功能对象模型 (Rich Object Model)，但语义模型不必拘泥于此，它也可以仅仅是一种数据结构。虽然我坚持我们应该使用一个合适的对象，但有数据模型来表示语义模型总好过没有。所以，在本书的讨论中，尽管我会将其假定为有合适行为的对象，但事实上，数据结构也是可以用来描述语义模型的选择之一。

很多系统都使用 Domain Model [Fowler PoEAA] 捕获系统的核心行为，而且通常 DSL 就是负责组装 Domain Model 的重要部分，但我依然坚持把 Domain Model 和语义模型区分

开。DSL 的语义模型通常是一个系统的 Domain Model 的子集，因为并不是 Domain Model 的所有部分都适合用 DSL 处理。另外，DSL 的任务不仅仅是填充 Domain Model，它还用于其他任务。

语义模型完全就是一个普通的对象模型，可以像操作其他所有对象模型一样操作它。在前面关于状态（state）的例子中，用状态模型的命令 – 查询 API 组装一个状态机，然后运行它，获取状态对象的行为。从某种意义上说，它与 DSL 是相互独立的，但在现实中，它们又是焦不离孟，孟不离焦。

（如果有编译器背景知识，你可能会将 Domain Model 等同于抽象语法树。简而言之，二者不同。我们会在 3.2 节中再进行分析。）

分离语义模型和 DSL 有几个好处。首先，我们可以暂时不纠结于 DSL 的语法和解析器，而专注于当前领域的语义。如果用上 DSL，这就说明我们所表达的东西已经非常复杂，复杂到了要拥有自己的模型来表示。

特别是，这样一来，就可以直接创建语义模型中的对象，操作它们进行测试。比如，可以创建一堆状态和迁移（transition），测试事件（event）和命令（command）是否运行良好，而无须处理解析。如果状态机的执行存在问题，问题必然在模型中，无须理解解析如何运作。

对于显式的语义模型而言，可以用多种 DSL 组装它。比如，从简单的内部 DSL 开始，稍后再给出一个更加易读的外部 DSL 版本。考虑到既有的脚本和用户，也许我们希望保留既有的内部 DSL，同时支持内部和外部 DSL。因为二者都可以解析成相同的语义模型，所以这么做并不困难。它还可以帮助我们避免语言之间的重复。

更为重要的是，拥有一个独立的语义模型，模型和语言就可以独立演化。如果要改变模型，无须修改 DSL 就可以探索做法，模型能够工作之后，给 DSL 添加必要的语言构造即可。同样，如果需要尝试不同的 DSL 语法，只要验证它们是否可以创建相同的模型对象即可。比较它们组装语义模型方式的不同，就可以知道两种语法之间的区别。

从许多方面来看，语义模型和 DSL 语法的分离实际上反映了领域对象及其表现的分离，这种做法在企业级软件设计中是很常见的。有时，我甚至把 DSL 视为另一种类型的用户界面。

对比 DSL 和表现，也有一些局限性。DSL 和语义模型始终还是关联的。如果要给 DSL 增加新构造，就需要保证语义模型中有支持，这也就意味着二者需要同时修改才可以。但是另一方面，二者的分离可以将语义问题和解析问题分开思考，从而简化了很多事情。

内部 DSL 和外部 DSL 的不同就在于解析这一步，既包括解析的目标，也包括解析的方式。两种风格的 DSL 都会产生同样的语义模型，正如前文暗示的一样，没有理由不使用单独的语义模型。事实上，这也正是我在编写状态机的例子中所使用的策略：多种 DSL，一个语义模型。

当使用外部 DSL 时，DSL 脚本、解析器和语义模型之间有条清晰的界限。DSL 脚本由一种独立的语言编写，解析器读取这些脚本，然后组装语义模型。而使用内部 DSL 时，它们之间更容易混杂在一起。我提倡使用一个显式的对象层次（“表达式生成器”（第 32 章）），

后者提供必要的连贯接口作为一种语言，然后，运行 DSL 脚本，调用表达式生成器的方法组装语义模型。所以对内部 DSL 而言，DSL 脚本的处理是由宿主语言解析器和表达式生成器共同完成的。

这让我们想到另一个有趣的问题：在内部 DSL 中使用“解析”这个词可能令人感觉古怪，我承认，我对此也并不感到十分舒服。然而，我发现，对内部 DSL 和外部 DSL 的处理进行类比思考其实是非常有用的。传统的解析方式是，获得文本流，将文本组织为解析树，处理解析树产生有用的输出。而解析内部 DSL，输入就是一系列方法调用，将它们组织到一个层次结构中（通常是隐式地在栈中组织），以便后续产生有用的输出。

这里使用“解析”这个词还涉及另一个因素，在很多场景下它并不直接处理文本。在内部 DSL 中，宿主语言解析器处理文本，而 DSL 处理器处理的更多是语言构造。但是 XML DSL 也有同样的情况：XML 解析器把文本翻译成 XML 元素，而 DSL 处理器基于这个基础进行工作。

是时候重温一下内部 DSL 和外部 DSL 的区别了。之前采用的区分标准——是否以应用的基础语言编写——通常而言是对的，但不绝对。一个极端的例子是，应用以 Java 编写，而 DSL 以 JRuby 编写。在这种情况下，我依然会将其归为内部 DSL，而我们用到的技术都是来自本书关于内部 DSL 的章节。

二者之间的真正区别在于，内部 DSL 使用一种可执行的语言编写，然后，通过在那种语言中执行 DSL 进行解析。无论 是 JRuby 还是 XML，DSL 都是嵌入载体语法中，但不同之处在于，JRuby 代码是要执行的，而 XML 数据结构只是读取而已。当然，大多数时候，内部 DSL 都是以应用的主要语言所实现的，因此，通常来说，这个定义还是有用的。

一旦有了语义模型，就需要让模型按照期望工作。在状态机的例子中，这个任务就是控制安全系统。有两种方法可以达到这个目的。最简单的，通常也是最好的办法就是：执行语义模型，因为语义模型本身就是代码，可以根据需要执行它。

另一个方法是代码生成。代码生成是指，生成单独编译和运行的代码。在一些圈子中，代码生成被视为 DSL 的根本。我看到一些讨论，认为任何和 DSL 相关的工作都需要通过生成代码来实现，代码生成不可或缺。在个别情况下，我甚至发现一些人在谈论或编写“解析器生成器”（第 23 章）时，总是不可避免地要谈论代码生成。DSL 同代码生成并没有本质关联，大多数情况下，执行语义模型是最好的选择。

代码生成在一种情况下最为有用，就是运行模型同解析 DSL 不在同一个地方的时候。一个很恰当的例子就是，代码执行在一个语言选择有限的环境中，比如在硬件有限制，或者在关系数据库中。你肯定不希望在一个烤箱或者在 SQL 中运行解析器，因此用一种更加合适的语言生成解析器和语义模型，然后再用它生成 C 或者 SQL。另一个类似的场景是，解析器依赖于生产环境用不到的程序库。这种情况很普遍，比如，为了 DSL 使用一个复杂工具，这也就是语言工作台倾向于使用代码生成的原因。

对于这些情况，在解析环境下，有一个“无须生成代码也可以运行”的语义模型还是很

有用的。运行语义模型，就可以在不了解代码生成如何运作的情况下，也能够体验 DSL 的执行。如果不生成代码，就能够测试解析和语义，就可以更快测试以及隔离问题。基于语义模型进行验证，可以在生成代码之前就捕获一些错误。

即便在一个能够很好地解释语义模型的环境里，关于代码生成，也存在一个争论，许多开发人员发现，富语义模型中的某些逻辑非常难以理解。从语义模型生成的代码会让一切变得更加明显，不再如魔法般隐晦。在一个缺乏高水平开发人员的团队中，这点至关重要。

就代码生成而言，务必记住一点，它只是 DSL 蓝图中的一个可选项。用到时，它极为关键，但大多数情况用不到。在我看来，DSL 如同雪地靴，在雪地中远足，当然要有一双保暖防水的靴子，而在炎炎夏日，却完全用不着。

使用语义模型生成代码还有一个好处，它解耦了代码生成器和解析器。纵然对解析过程一无所知，我也能够写出一个代码生成器，并对其进行独立测试。单凭这一点，语义模型就已值回票价了。另外，它也更容易支持生成多种目标代码，如果需要的话。

3.2　解析器的工作方式

所以，内部 DSL 和外部 DSL 的差别主要体现在解析上。虽然二者确实存在一些细节上的不同，但它们也有很多共通之处。

一个最重要的共同点就是，解析都是一个很强的层级操作。当解析文本时，把数据块组织成一个树结构。考虑一个简单结构，状态机中的事件列表。在外部 DSL 语法中，它看起来如下所示：

```
events
  doorClosed D1CL
  drawerOpened D2OP
end
```

这个复合结构是一个事件列表，包含一系列事件，每个事件都有名字和代码。

用 Ruby 编写的内部 DSL 与上述代码很类似：

```
event :doorClosed "D1CL"
event :drawerOpened "D2OP"
```

对于整个列表，这里没有显式的标记，但是每一个事件本身仍是一个层级 ：每个事件都有表示名字的符号和表示代 码的字符串。

无论何时看到这样的脚本，都可以把它想象为一个层级，这样的层级称为语法树（或者解析树）。任何脚本都可以转化为许多潜在的语法树——这取决于如何分解它。相对于单词（word），语法树是一种更有效的脚本表现形 式，因为可以遍历语法树，使用各种不同的方式来对它进行操作。

如果用到“语义模型”（第 11 章），可以把一个语法树翻译成语义模型（见图 3-2）。如果经常读一些语言社区的资料，我们会发现，语法树得到了非常多的关注——人们通常直接执行语法树，或者基于语法树生成代码。更有效的做法是，语法树可以直接当做语义模型来使用。但大多数时候，我不会这么做，因为语法树同 DSL 脚本关联非常紧密，这样做只会让 DSL 的处理同语法产生耦合。

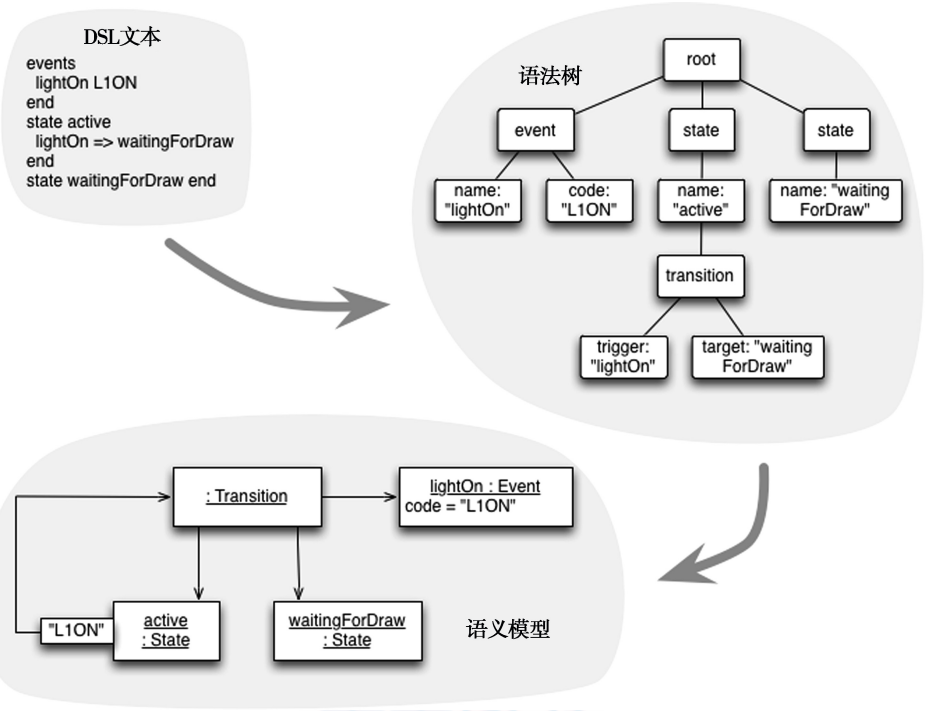

图 3-2　语法树和语义模型是 DSL 脚本的不同表现形式

目前为止，我都一直都在谈论语法树，仿佛它是系统里一种有形的数据结构，就像 XML DOM 一样。有时候，它的确是，但更多的时候，它不是。很多时候，语法树在调用栈中形成，在遍历的过程中得到处理。所以，我们看不到整个树，而只能看到当前处理的分支（类似于 XML SAX 的工作方式）。尽管如此，尝试理解隐匿于调用栈中鬼魅般的语法树总是有帮助的。对一个内部 DSL 而言，语法树的形成有赖于方法调用（“嵌套函数”（第 34 章））的实参和嵌套对象（“方法级联”（第 35 章））。有时候，我们看不到一个很明显的层次结构，不得不进行模拟（有层次结构的“函数序列”（第 33 章）可以由“语境变量”（第 13 章）模拟）。语法树或许形似鬼魅，但它依然是一种有益的脑力工具。使用外部 DSL 会产生一个更加显式的语法树，事实上，有时候我们确实生成了一个完完全全的语法树数据结构（“树的构建”（第 24 章））。但即使是外部 DSL，通常在处理过程中，也是在调用栈中不断形成和修剪着语法树。（这里引用了几个尚未描述的模式，如果是第一次读到，放心略过即可，但以后再读，这些引用会很有帮助。）

3.3　文法、语法和语义

如果要处理一种语言的语法，文法是一种很重要的工具。文法是一组规则，用以描述如何将文本流转化为语法树。大多数程序员都会在生命中的某一刻接触文法，因为文法常用以描述我们日常使用的程序设计语言。文法由一系列产生式规则组成，每个生产规则都有一个名字（term）以及一个描述如何分解它的语句（statement）。所以，一个加法语句可能看起来就像这样：`additionStatement:=number'+'number`。它告诉我们，如果遇到语句 `5+3`，解析器能够将其识别为加法语句。因为规则是可以相互引用的，所以也会有一条针对数字的规则，告诉我们如何识别合法数字。通过这些规则，我们就可以得到一种语言的文法。

一种语言可以由多种不同的文法来定义，认识到这点很重要。世界上不存在某种语言的唯一文法。一种文法就定义了语言所生成语法树的一种结构，对于一段特定的文本，可能会识别出许多不同的语法树结构。一种文法只定义一种形式的语法树；选择何种文法和语法树取决于很多因素，包括语言的文法特性以及处理语法树的方式等。

文法只定义一种语言的语法——它在语法树中如何表现。而这与语义（也就是表达式的含义）无关。根据上下文不同，`5+3` 可能等于 `8`，也可能等于 `53`，语法相同，但语义可能截然不同。在“语义模型”（第 11 章）中，语义的定义浓缩为如何根据语法树组装语义模型，以及如何处理语义模型。特别是，如果两个表达式产生相同结构的语义模型，即使语法不同，它们的语义其实也是相同的。

如果在使用外部 DSL，特别是，用到了“语法指导翻译”（第 18 章），我们很可能会显式地使用文法来构建解析器。如果用的是内部 DSL，可能没有显式的文法，但是从文法的角度思考 DSL 仍然是有用的，文法有助于我们在众多内部 DSL 模式中进行选择。

对于内部 DSL，谈论文法显得有些奇怪，原因之一是，这里解析了两遍，所以包含了两种文法。第一种是宿主语言本身的解析，这显然要依赖于宿主语言的文法。这一遍解析创建宿主语言的执行指令。当宿主语言所构建的 DSL 执行时，鬼魅般的语法树就会在调用栈中创建。只有在第二遍解析时，才会出现这个名义上的 DSL 语法。

3.4　解析中的数据

当解析器执行时，它需要存储解析过程中的数据。这些数据可能是一个完整的语法树，但大多数情况下不是这样的。即使这种情况出现了，还是需要存储其他的一些数据，以便解析工作可以正常进行。

解析本质上是一种树遍历（见图 3-3），当处理某一部分 DSL 脚本时，对于正在处理的语法树分支，我们可以得到其上下文的一些相关信息。然而，通常我们还会用到这个分支以外的信息。我们再从状态机的例子里选取一段代码看看：

```
commands
  unlockDoor D1UL
end

state idle
  actions {unlockDoor}
end
```

我们在这里看到了一种常见的情况：命令定义在语言的某个地方，然后在其他地方引用。当命令在语句的行为中引用时，我们所在的语法树分支不同于命令定义的分支。如果语法树的表示只存在于调用栈中，那么到这里，命令定义就已经消失了。因此，要把命令对象保存下来以备后用，这样，在行为代码中就可以引用了。

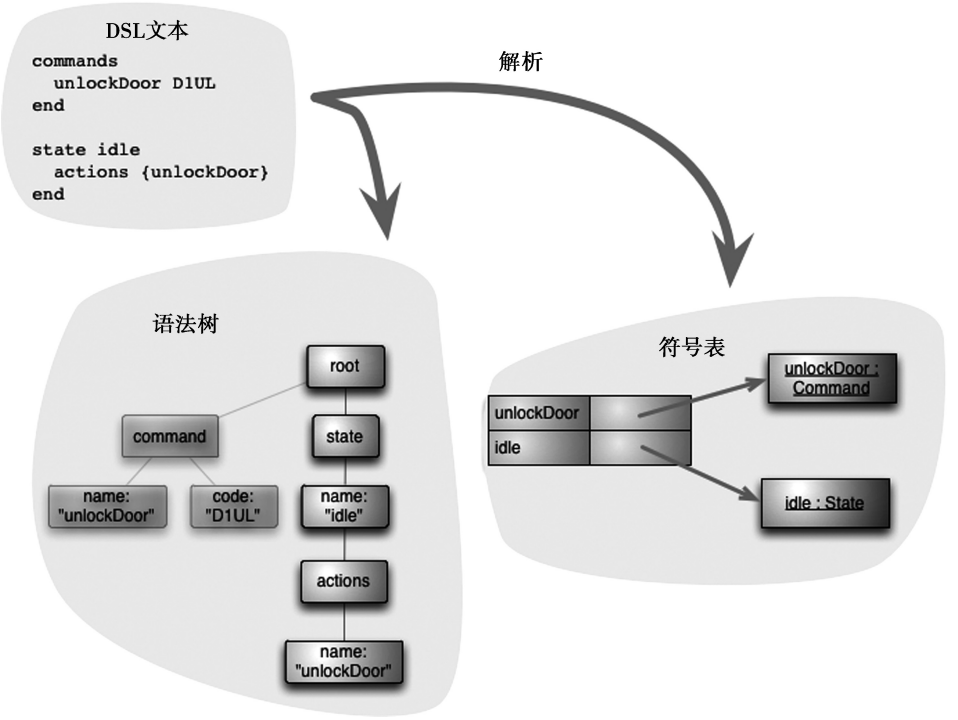

图 3-3　解析不仅创建解析树，还创建一个符号表

为了做到这一点，我们用到了“符号表”（第 12 章），它本质上是一个字典，其键是标识符 unlockDoor，值是在解析中表示命令的对象。当处理文本 unlockDoor D1UL 时，创建一个对象持有数据，然后，把它存放在符号表里，键为 unlockDoor。存放的对象可能是命令的语义对象，也可能针对局部语法树的中间对象。稍后，当处理 actions {un lockDoor} 时，我们会通过符号表查找这个对象，以获得状态同行为之间的关系。因此，符号表对于交叉引用至关重要。如果在解析中创建一棵完整的语法树，理论上，可以省略符号表，虽然通常它依然是一个有用的结构，可以把事物关联起来。

在本节结束之前，我们看两个具体的模式。之所以在这里提起它们，是因为内部 DSL 和外部 DSL 都会用到。所 以，这里是个合适的地方，即便本章讨论的大多是一些高层次的内容。

当进行解析时，要保存结果。有时，所有结果都可以放到符号表里 ；有时，许多信息要保存在调用栈里 ；还有时，要在解析器里有额外的数据结构。在所有这些情况里，最明显要做的一件事是，创建“语义模型”（第 11 章）对象保存结果。然而，在很多情况下，要到解析的最后时刻才能创建语义模型，所以，还要创建一些中间对象。对于这种中间对象，一个常见的例子是“构造型生成器”（第 14 章），它是一个对象，包含语义模型所需的全部数据。如果语义模型在创建后就是只读的，这种做法就非常有用了，可以在解析过程中逐步地为它收集数据。构造型生成器拥有同语义模型一样的字段，但这些字段是可读写的，这样，就有地方保存数据了。一旦有了所有数据，就可以创建语义模型对象了。使用构造型生成器会让解析器变得复杂，但相比于改变语义模型的只读属性，我宁愿选择这么做。

事实上，有时候，我们会在处理完所有 DSL 脚本时，再创建语义模型对象。在这种情况下，解析就会有不同的阶段：首先，读取 DSL 脚本，创建中间的解析数据，其次，处理中间数据，组装语义模型。在文本处理阶段做多少工作，后面做什么，这取决于语义模型如何组装。

表达式的解析方式取决于我们处理的上下文。查看下面这段文本：

```
state idle
  actions {unlockDoor}
end

state unlockedPanel
  actions {lockDoor}
end
```

当处理 `actions{lockDoor}` 时，有一点很重要，它处于 `unlockedPanel` 状态的上下文中，而非空闲态。通常，解析器构建以及遍历解析树的方式，就提供了这个上下文，但还有很多情况，很难做到这一点。如果检查解析树无法获得上下文，那么一种好的做法就是，持有上下文，对于这个例子，我们可以把当前状态保存在一个变量里。我将这种变量称为“语境变量”（第 13 章）。这种语境变量类似于符号表，可以持有语义模型对象，或者一些中间对象。

虽然语境变量用起来很简单，但一般来说，我倾向于尽可能避免使用。语境变量会让解析代码难于理解，正如大量的可变变量会让过程式代码变得复杂。当然，肯定会有无法避免使用语境变量的情况，但我更倾向于将其视为应该避免的坏味道。

3.5　宏

“宏”（第 15 章）是一种工具，既可以用于内部 DSL，也可以用于外部 DSL。宏曾经得到广泛应用，但如今已不那么常见了。在大多数情况下，我建议尽量避免使用宏，但偶尔，

它也有一些用处。所以，接下来，就谈论宏的运作方式以及使用时机。

宏有两种风格：文本宏和语法宏。文本宏最容易理解，简单说就是文本替换。使用文本宏会带来便利，一个很好的例子就是，在 CSS 文件中指定颜色。除了少数几种特定情况外，CSS 强制我们以颜色代码指定颜色，比如 `#FFB595`。这样的代码并不表意，更糟糕的是，如果要在多个地方使用同一种颜色，就要重复同样的代码。任何形式的代码重复都是坏味道。我们可以给它一个在上下文中有意义的名字，比如 `MEDIUM_SHADE`，在一个地方定义它，比如，`MEDIUM_SHADE` 等于 `#FFB595`。

虽然 CSS（至少目前为止）并不允许这么做，但可以用一个宏处理器处理这种情况。创建一个文件，它就是 CSS 文件，只不过其中用 `MEDIUM_SHADE` 表示所需的颜色。然后，用宏处理器做一次简单的文本替换，把 `MEDIUM_SHADE` 替换为 `#FFB595`。

这只是一个非常简单的宏处理例子。更复杂的宏还可以有参数。一个经典的例子是 C 的预处理器，比如，定义一 个宏 `sqr(x)`，它可以替换为 `x*x`。

通过宏创建 DSL 有很多方式，可以使用宿主语言本身（比如，C 预处理器），也可以提供单独的一个文件，将其转换为宿主语言。宏的缺点在于，它有很多诡异的问题，这使得它在实际中难于使用。所以，文本宏现在已经不再受宠，很多专业人士（比如我）都反对使用它。

语法宏也是通过替换实现的，但是它处理的是宿主语言中有效的元素，将一种表达式转换为另一种。在以“大量使用语法宏”而闻名的语言中，Lisp 最为著名，虽然 C++ 模板更广为人知。使用语法宏编写 DSL，是 Lisp 编写内部 DSL 的核心技术，但这种技术也仅限于支持宏的语言。因此，本书不会过多谈及，因为仅有很少的语言可以支持宏。

3.6　测试 DSL

过去二十年，我变得越来越不想谈论测试。我已然成为一个忠实粉丝，迷恋着测试驱动开发 [Beck TDD] 以及类似的技术：将测试置于程序设计之前。所以，我已无法脱离测试思考 DSL。

对 DSL 而言，我把其测试分为三个独立的部分：“语义模型”（第 11 章）的测试，解析器的测试，以及脚本的测试。

3.6.1　语义模型的测试

我首先想到的部分是“语义模型”（第 11 章）的测试。这些测试用来保证语义模型能够如预期般工作，也就是说，当执行模型时，根据编写的代码，它能够产生正确的输出。这是一个标准的测试实践，同测试任何框架里的对象一样。对于这种测试，根本无需 DSL。使用模型本身的基本接口就可以组装模型。这种做法很好，因为可以独立测试模型，无须 DSL 和解析器。

我们用秘密面板控制器说明这种做法。在这个例子中，语义模型就是状态机。下面测试语义模型，用 1.3 节的例子提及的命令 – 查询 API 组装模型，无需任何 DSL。

```
@Test
public void event_causes_transition() {
  State idle = new State("idle");
  StateMachine machine = new StateMachine(idle);
  Event cause = new Event("cause", "EV01");
  State target = new State("target");
  idle.addTransition(cause, target);
  Controller controller = new Controller(machine, new CommandChannel());
  controller.handle("EV01");
  assertEquals(target, controller.getCurrentState());
}
```

上面的代码演示了如何独立测试语义模型。然而，需要说明的是，这个例子的真实测试代码会更复杂，也应该更好地分解。

有两种方法来分解这类代码。首先，创建一堆小的状态机，提供最小的测试夹具，以便测试语义模型的各种特性。比如，要测试“事件触发转换”（event triggers a transition），只要创建一个简单状态机，它处于空闲态，并且可以转换（transition）为另外两个状态。

```
class TransitionTester...
  State idle, a, b;
  Event trigger_a, trigger_b, unknown;

  protected StateMachine createMachine() {
    idle = new State("idle");
    StateMachine result = new StateMachine(idle);
    trigger_a = new Event("trigger_a", "TRGA");
    trigger_b = new Event("trigger_b", "TRGB");
    unknown = new Event("Unknown", "UNKN");
    a = new State("a");
    b = new State("b");
    idle.addTransition(trigger_a, a);
    idle.addTransition(trigger_b, b);
    return result;
  }
```

如果要测试命令（command），也许只要一个更小的状态机，它只有一个空闲态。

```
class CommandTester...
  Command commenceEarthquake = new Command("Commence Earthquake", "EQST");
  State idle = new State("idle");
  State second = new State("second");
  Event trigger = new Event("trigger", "TGGR");

  protected StateMachine createMachine() {
    second.addAction(commenceEarthquake);
    idle.addTransition(trigger, second);
    return new StateMachine(idle);
  }
```

这些不同的夹具可以用类似的方法运行，给它们创建一个共同的超类会让这一切更加容易。这个超类首先应该能够创建公用夹具 ——在这个初始化过程里，包括一个控制器

（controller）、一个命令通道（command channel），还有子类提供的状态机。

```
class AbstractStateTesterLib...
  protected CommandChannel commandChannel = new CommandChannel();
  protected StateMachine machine;
  protected Controller controller;

  @Before
  public void setup() {
    machine = createMachine();
    controller = new Controller(machine, commandChannel);
  }

  abstract protected StateMachine createMachine();
```

下面编写测试，在控制器中触发事件，然后检查状态。

```
class TransitionTester...
  @Test
  public void event_causes_transition() {
    fire(trigger_a);
    assertCurrentState(a);
  }
  @Test
  public void event_without_transition_is_ignored() {
    fire(unknown);
    assertCurrentState(idle);
  }

class AbstractStateTesterLib...
  //-------- Utility methods --------------------------
  protected void fire(Event e) {
    controller.handle(e.getCode());
  }
  //------- Custom asserts --------------------------
  protected void assertCurrentState(State s) {
    assertEquals(s, controller.getCurrentState());
  }
```

超类提供的 Test Utility Method [Meszaros] 和 Custom Assertion [Meszaros] 让测试更易读。

另一种测试语义模型的方法是组装一个拥有很多特性的大模型，然后进行多方面的测试。在下面的例子里，我用格兰特小姐的控制器作为测试夹具。

```
class ModelTest...
  private Event doorClosed, drawerOpened, lightOn, doorOpened, panelClosed;
  private State activeState, waitingForLightState, unlockedPanelState,
                idle, waitingForDrawerState;
  private Command unlockPanelCmd, lockDoorCmd, lockPanelCmd, unlockDoorCmd;
  private CommandChannel channel = new CommandChannel();
  private Controller con;
  private StateMachine machine;

@Before
public void setup() {
  doorClosed = new Event("doorClosed", "D1CL");
  drawerOpened = new Event("drawerOpened", "D2OP");
  lightOn = new Event("lightOn", "L1ON");
```

```
    doorOpened = new Event("doorOpened", "D1OP");
    panelClosed = new Event("panelClosed", "PNCL");
    unlockPanelCmd = new Command("unlockPanel", "PNUL");
    lockPanelCmd = new Command("lockPanel", "PNLK");
    lockDoorCmd = new Command("lockDoor", "D1LK");
    unlockDoorCmd = new Command("unlockDoor", "D1UL");

    idle = new State("idle");
    activeState = new State("active");
    waitingForLightState = new State("waitingForLight");
    waitingForDrawerState = new State("waitingForDrawer");
    unlockedPanelState = new State("unlockedPanel");

    machine = new StateMachine(idle);

    idle.addTransition(doorClosed, activeState);
    idle.addAction(unlockDoorCmd);
    idle.addAction(lockPanelCmd);

    activeState.addTransition(drawerOpened, waitingForLightState);
    activeState.addTransition(lightOn, waitingForDrawerState);

    waitingForLightState.addTransition(lightOn, unlockedPanelState);
    waitingForDrawerState.addTransition(drawerOpened, unlockedPanelState);

    unlockedPanelState.addAction(unlockPanelCmd);
    unlockedPanelState.addAction(lockDoorCmd);
    unlockedPanelState.addTransition(panelClosed, idle);

    machine.addResetEvents(doorOpened);
    con = new Controller(machine, channel);
    channel.clearHistory();
  }
  @Test
  public void event_causes_state_change() {
    fire(doorClosed);
    assertCurrentState(activeState);
  }

  @Test
  public void ignore_event_if_no_transition() {
    fire(drawerOpened);
    assertCurrentState(idle);
  }
```

在这个例子里，我又一次用到了自己的命令 – 查询接口组装语义模型。然而，随着测试夹具变得复杂，我会考虑用 DSL 创建测试夹具，以简化测试。如果我的解析器有测试，我就可以这么做。

3.6.2　解析器的测试

当使用“语义模型”（第 11 章）时，解析器的工作就是组装语义模型。所以，解析器的

测试就是，编写一小段 DSL，确保它们生成结构正确的语义模型。

```
@Test
public void loads_states_with_transition() {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end " +
    "state target end ";
  StateMachine actual = StateMachineLoader.loadString(code);

  State idle = actual.getState("idle");
  State target = actual.getState("target");
  assertTrue(idle.hasTransition("TGGR"));
  assertEquals(idle.targetState("TGGR"), target);
}
```

这样使用语义模型不太合适，而且可能破坏语义模型对象的封装。所以，还有一种方法是，定义一些方法，比较语义模型，使用这些方法来测试解析器的输出。

```
@Test
public void loads_states_with_transition_using_compare() {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end " +
    "state target end ";
  StateMachine actual = StateMachineLoader.loadString(code);

  State idle = new State("idle");
  State target = new State("target");
  Event trigger = new Event("trigger", "TGGR");
  idle.addTransition(trigger, target);
  StateMachine expected = new StateMachine(idle);

  assertEquivalentMachines(expected, actual);
}
```

相比于常规的相等性判定，复杂结构的相等性判定更为复杂。要了解对象之间的具体差异，一个布尔（Boolean）类型的答案是远远不够的。所以，要用“通知”（第 16 章）进行比较。

```
class StateMachine...
  public Notification probeEquivalence(StateMachine other) {
    Notification result = new Notification();
    probeEquivalence(other, result);
    return result;
  }

  private void probeEquivalence(StateMachine other, Notification note) {
    for (State s : getStates()) {
      State otherState = other.getState(s.getName());
      if (null == otherState) note.error("missing state: %s", s.getName()) ;
      else s.probeEquivalence(otherState, note);
```

```
    }
    for (State s : other.getStates())
      if (null == getState(s.getName())) note.error("extra state: %s",
         s.getName());
    for (Event e : getResetEvents()) {
      if (!other.getResetEvents().contains(e))
        note.error("missing reset event: %s", e.getName());
    }
    for (Event e : other.getResetEvents()) {
      if (!getResetEvents().contains(e))
        note.error("extra reset event: %s", e.getName());
    }
  }
class State...
  void probeEquivalence(State other, Notification note) {
    assert name.equals(other.name);
    probeEquivalentTransitions(other, note);
    probeEquivalentActions(other, note);
  }

  private void probeEquivalentActions(State other, Notification note) {
    if (!actions.equals(other.actions))
      note.error("%s has different actions %s vs %s", name, actions, other.
          actions);
  }

  private void probeEquivalentTransitions(State other, Notification note) {
    for (Transition t : transitions.values())
      t.probeEquivalent(other.transitions.get(t.getEventCode()), note);
    for (Transition t : other.transitions.values())
      if (!this.transitions.containsKey(t.getEventCode()))
        note.error("%s has extra transition with %s", name, t.getTrigger());
  }
```

这种检测方式就是遍历语义模型中的对象，然后把差异记录在通知中。这样，就可以找出所有的差异，而不是找到 第一个就停下来。断言只要检查通知中是否有错误即可。

```
class AntlrLoaderTest...
  private void assertEquivalentMachines(StateMachine left, StateMachine right) {
    assertNotificationOk(left.probeEquivalence(right));
    assertNotificationOk(right.probeEquivalence(left));
  }

  private void assertNotificationOk(Notification n) {
    assertTrue(n.report(), n.isOk());
  }

class Notification...
  public boolean isOk() {return errors.isEmpty();}
```

你可能会认为我是一个偏执狂，要从两个方向进行相等性断言，但事实上，代码经常会出乎所料。

无效输入的测试

刚才讨论的是正向测试，保证有效的 DSL 输入可以生成结构正确的“语义模型”（第 11 章）。测试的另一种类别是负向测试，用于检测在无效输入的情况下会发生什么。这

还会涉及错误处理和诊断等技术，这些内容超出了本书的范围，但我还是要在这里简单地讨论对无效输入的测试。

无效输入的测试的基本想法，就是把各式各样的无效输入抛给解析器。第一次进行这样的测试会非常有趣。我们经常会看到一些不起眼却很极端的错误。得到这样的结果可能已经足够了，除非我们要对错误诊断提供更多的支持。更糟糕的情况是，提供无效输入、解析，根本没有任何错误。这违反了“快速失败”（fail fast）原则——也就是说，错误应该尽快、尽可能明显地暴露出来。如果用无效状态组装一个模型，又没有任何检查，那么可能要到很晚才会发现问题。到了那个时候，原始的错误（加载无效输入）和后来的失败之间已然相去甚远，这段距离会让错误定位难上加难。

状态机例子只有很少的错误处理机制——这是本书中一个典型的例子。用下面这个测试来测试解析器例子，看看会发生什么：

```
@Test public void targetStateNotDeclaredNoAssert () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end ";
  StateMachine actual = StateMachineLoader.loadString(code);
}
```

虽然测试通过了，但情况非常糟糕。稍后，我尝试用模型做些事情，即便只是简单的打印工作，它都会抛出空指针异常。这个例子有些粗糙，我可以接受，毕竟它只是用于教学，但是，输入 DSL 中的一个拼写错误都要耗费大量时间调试。这是我的时间，我喜欢假装时间很宝贵，所以，我希望它能够快速失败。

问题在于，创建了一个无效结构的语义模型，所以，检查这个错误也是语义模型的职责所在——在这个例子中，就是要在给状态（state）添加转换（transition）的方法里进行处理。添加一个断言检测这个问题。

```
class State...
  public void addTransition(Event event, State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(), new Transition(this, event, targetState));
  }
```

现在，就可以修改测试捕获异常了。如果更改输入的行为，它就会告诉我，还会记录是怎样的非法输入带来的问题。

```
@Test public void targetStateNotDeclared () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end ";
  try {
    StateMachine actual = StateMachineLoader.loadString(code);
    fail();
```

```
  } catch (AssertionError expected) {}
```

你会注意到，我只给目标状态添加了断言，而没有断言触发事件，它同样也可能为空。这么做的原因是，空事件在调用 `event.getCode()` 时，会立即抛出空指针异常。这就满足了快速失败的要求。可以用另外一个测试检查这个问题。

```
@Test public void triggerNotDeclared () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "wrongTrigger => target " +
    "end " +
    "state target end ";
  try {
    StateMachine actual = StateMachineLoader.loadString(code);
    fail();
  } catch (NullPointerException expected) {}
```

空指针异常确实能够快速失败，但是它不如断言那么清晰。一般来说，我不会对方法实参进行非空断言，我觉得，因为要额外阅读代码，所以这种做法带来的好处有些不值得。除非一段为空的处理不能立即失败，就像上面的空目标状态一样。

3.6.3　脚本的测试

“语义模型”（第 11 章）和解析器的测试就是对普通代码进行单元测试。然而，DSL 脚本也是代码，我们也应该考虑对它们进行测试。我经常听到这样的观点：“DSL 脚本过于简单和明显，不值得测试”，但我本能地对这种想法存疑。我把测试视 为一种“双重检查”（double–check）机制。当编写代码和测试时，其实是用两种非常不同的方式确定同一行为，一种是用抽象的方式（代码），另一种是用样例的方式（测试）。对任何有持久价值的东西，我们都应该进行双重确认。

脚本测试的细节很大程度上取决于要测试的东西。基本的方法是，提供一个测试环境，在其中创建文本夹具，运行 DSL，比较结果。准备这样的环境需要花费一些精力，不过，DSL 易读，并不意味人们就不会犯错误。如果不提供这样的环境，没有双重检查机制，会极大地增加在 DSL 中犯错误的风险。

脚本测试也扮演着集成测试的角色，因为解析器或者语义模型的任何错误都会让它失败。所以，选择一些 DSL 脚本用于此目的是值得的。

通常，对于测试和调试 DSL 脚本而言，脚本可视化是一种非常有用的辅助手段。如果脚本已经置入语义模型，那么对脚本的逻辑，生成不同的可视化方式（文本或图形化）相对容易。以多种方式呈现有助于人们发现错误，确实，这种双重检查的想法，就是自动测试代码如此有价值的核心原因。

对于状态机这个例子，我会先想出几个对于这类状态机来说有意义的场景。对我来说，合理的方法就是运行这些场景，每个场景都是一连串发送给状态机的事件。然后，检查每个

状态机的最终状态，以及发出的命令。以更加可读的方式构造这样的测试，其实就创建了另一套 DSL。这并不奇怪，测试脚本其实就是一种 DSL，因为它很好地满足了受限的、声明式的语言要求。

```
events("doorClosed", "drawerOpened", "lightOn")
        .endsAt("unlockedPanel")
        .sends("unlockPanel", "lockDoor");
```

3.7　错误处理

每次写作一本书，我都会到达一个时刻，我意识到，如同做软件一样，只有削减范围，书才能出版。然而，这就意味某些重要的主题无法妥善涵盖。但是，一本有用却不完整的书总好过一本完整却永远也不能完成的书。本书有很多主题，我想进一步探索，排在第一位的就是错误处理。

在大学的编译课上，我记得老师说过：解析和生成输出是编写编译器中容易的部分——真正的难点在于，给出更好的错误消息。相应地，错误诊断就超出了那堂课的范围，对于本书也一样。

体面的错误消息“超出范围”在现实中会更进一步。即便是一些成功的 DSL，良好的诊断信息也极其罕见。许多广泛使用的 DSL 包在提供有用信息方面所做有限。Graphviz 是我最喜欢的 DSL 工具之一，当错误发生时，它只会简单地告诉我 `syntax error near line 4`，我还感到几分幸运，因为它竟然还能给出行号。我确实遇到了很多工具，只是直接失败，我只能通过注释代码，以二分查找的方式定位问题。

我们可以批评一个系统没有提供良好的错误诊断，但是错误诊断也是一件需要折衷的事情。多花一分时间在错误处理上，就意味要少花一分时间在功能添加上。许多来自现实世界 DSL 的证据都表明，人们的确可以忍受表现不佳的错误诊断。毕竟，DSL 脚本很小，相比于通用语言，粗糙的错误处理也是可以接受的。

这么说并不是劝你不要在错误诊断上花时间。对一个使用频率很高的程序库而言，良好的错误诊断可以帮助我们节省很多时间。每次折衷都有所不同，需要根据自己的环境决定。确实，这种说法会让我不再为“没有在本书为这个主题开辟一节”感到太多内疚。

虽然不能如我所愿般对这个话题做深入探讨，但如果你决定提供更好的错误诊断支持，我希望我所说的这些可以让你开始对其有更多的思考。

（我应该谈一下最暴力的一种错误查找技术：注释。如果要用外部 DSL，请确保它可以支持注释。并不只是为了那个显而易见的原因，它还可以帮助人们找到问题。以换行符结束的注释是最方便的。针对不同的受众，我会使用“#”（脚本风格）或者“//”（C 风格），这些都可以通过一条简单的词法分析规则实现。）

如果你遵循我的一般建议，采用“语义模型”（第 11 章），那么有两个地方可以放置错误处理：模型或者解析器。对于语法错误，处理它最明显的地方就是解析器。有些语法错误

有人会替我们处理，比如内部 DSL 中宿主语言的语法错误，或是使用“解析器生成器”（第 23 章）时外部 DSL 中的文法错误。

在处理语义错误时，要从解析器和模型中进行选择，二者各有其优。如果要检查语义规则是否结构良好，模型是一个正确的地方。我们拥有所有的信息，这些信息以思考所需的方式组织，所以，在这里可以写出最清晰的错误检查代码。此外，如果要在多个地方组装模型，比如，有多个 DSL 或者用到命令 – 查询接口，模型也是最合适进行检查的地方。

纯粹把错误处理放在语义模型中，确实有个严重的问题：无法链接回 DSL 脚本中出问题的源码，甚至无法得到一个大概的行号。这会使定位错误越发困难，但这也可能不是一个棘手的问题。一些经验表明，在多数情况下，纯粹基于模型的错误消息已足以帮我们找到问题。

如果确实需要，我们还是有办法拿到 DSL 脚本的上下文。最显而易见的一种做法就是把错误检测规则放在解析器中。不过，这种做法的问题在于，它会让编写规则变得更加困难，因为在这种情况下，我们是在语法树的层次上进行工作，而不是在语义模型的层次上。另外，我们还要承担规则重复带来的极大风险，正如代码重复所带来的问题一样。

另一种可选方案是，把语法信息放入语义模型。比如，在语义转换对象中添加一个记录行号的字段，这样，如果语义模型在转换中检测到错误，就可以输出脚本中的行号。问题是，由于要跟踪信息，这种做法会让语义模型变得更加复杂。此外，脚本与模型之间的映射可能并不那么清晰，所以，错误消息非但无益，反而会让人困惑。

第三种，也是我认为最好的一种方案是：使用语义模型检测错误，但是，错误检测的触发是在解析器中。具体来说，就是解析器解析一大块 DSL 脚本，组装出语义模型，然后让模型去找错误（如果组装模型时并不直接这么做的话）。如果模型发现任何错误，解析器都可以拿到这些错误，提供其所知的上下文。这样，就分离了对语法知识（在解析器中）和语义知识（在模型中）的关注。

还有一种有用的方法是，把错误处理分为初始、检测和报告三个阶段。最后一种方案是，把初始化放在解析器中，把检测放在模型中，报告则二者皆有，模型负责提供错误的语义，解析器负责添加语法上下文。

3.8　DSL 迁移

DSL 拥趸们应该警惕的一个风险是“先编写，后使用”的想法。同其他软件一样，成功的 DSL 需要不断演化。也就是说，以早期版本 DSL 编写的脚本可能无法在新版上运行。

DSL 的诸多属性（无论好坏）同程序库完全一样，这一点也不例外。如果从别人那里得到一个程序库，基于它编写一些代码，他们升级了程序库，我们可能最终就卡在那里。DSL 并不会真正改变这一点；本质上 DSL 定义的就是已发布接口（published interface），我们不

得不自行处理这个后果。

在重构 [Fowler Refactoring] 一书里，我开始使用已发布接口这个术语。已发布接口和更一般的“公共”接口的不同之处在于，前者由其他团队编写的代码使用。所以，修改已发布接口是很困难的，因为他们无法重写调用代码。修改已发布 DSL 对于内部 DSL 和外部 DSL 都是个问题。如果 DSL 尚未发布，而且相关语言有自动化重构工具，修改内部 DSL 还容易一点。

解决 DSL 修改问题的一种方式是，提供工具，自动把 DSL 从一个版本迁移到另一个版本。这些工具可以在升级时运行，也可以在尝试运行旧版脚本时自动运行。

有两种方式实现迁移。一种是增量迁移的策略，这种方式本质上同人们处理数据库设计的演化所采用的想法是一致的。我们会创建一个程序，对 DSL 定义的每个变化，都可以自动将旧版脚本迁移到新版本上。采用这种方式，当发布新版 DSL 时，还要提供一个脚本，迁移使用 DSL 的代码库。

对于增量迁移，有一点很重要，就是要让每一个修改尽可能小。想象一下，我们要从版本 1 升级到版本 2，DSL 定义有 10 处修改。在这种情况下，不要只创建一个“从版本 1 到版本 2”的迁移脚本；相反，可以创建至少 10 个脚本。一次修改 DSL 定义的一个特性，为每个修改编写一个迁移脚本。将修改拆分为更多步骤，某些特性甚至要多于一步（因此要多于一个迁移脚本），你会发现这种做法非常有用。听起来，这种做法比单一脚本要多做一些工作，但重点在于，如果迁移都很小，脚本写起来会容易得多，而且把多个脚本串起来也很容易。所以，写 10 个脚本反而比写一个更快。

另一种方式是基于模型的迁移。这种策略可以与“语义模型”（第 11 章）配合使用。有了基于模型的迁移，我们的语言就可以支持多个解析器，每个发布版对应一个（当然，我们只会为版本 1、2 这样的大版本这样做，而不会理会中间步骤）。每个解析器都会组装语义模型。当采用语义模型时，解析器的行为相当简单。所以，有多个解析器也不是什么麻烦事。之后，针对所处理脚本的版本，可以运行相应的解析器。这种方式可以处理多个版本，但并不迁移脚本。为了处理迁移，可以根据语义模型编写一个生成器，生成 DSL 脚本。采用这种方式，可以对版本 1 的脚本运行解析器，组装语义模型，然后用生成器生成版本 2 的脚本。

基于模型的方式有一个问题，它很容易丢掉一些与语义无关但脚本编写者希望保留的东西。最明显的一个例子就是注释。如果解析器太过聪明，这个问题就会越发严重。以这种方式进行迁移的需求，会促使解析器写得更加简单，这是好事。

如果 DSL 的修改够大，很可能无法从版本 1 的脚本直接转换为版本 2 的语义模型。在这种情况下，可能需要保留版本 1 的模型（或者中间模型），给予它生成版本 2 脚本的能力。

对这两种方式，我没有很强烈的倾向。

迁移脚本可以由脚本程序员在需要的时候自己运行，或者让 DSL 系统自动运行。如果要自动运行的话，在脚本中记录它是哪个版本将会非常有用，这样，解析器可以很容易地检

测到，并触发相应的迁移脚本。事实上，有一些 DSL 作者甚至强调说，DSL 脚本上的版本信息是必需的，这样就可以更容易地检测出过时的 DSL，以便进行脚本的迁移。虽然版本信息可能会给脚本添加一些噪音，但它基本上是不太会更改的东西。

当然，还有一个迁移选项，就是不迁移。保留版本 1 的解析器，让它组装版本 2 的模型。我们应该帮助人们进行迁移，如果他们需要更多特性，他们就会这么做。可以的话，直接支持旧脚本也未尝不可，这样，人们就可以按照自己的节奏去迁移。

虽然这样的技术很吸引人，但在实际中，是否值得这么做依然存疑。如我之前所说，这个问题同广为使用的程序库完全一样，自动迁移的策略没有多少人用。