参考：<https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html>

## [演化架构和紧急设计: 使用****DSL****](http://www.ibm.com/developerworks/cn/java/j-eaed13/index.html)

惯用模式可以是 技术也可以是 领域。技术模式为常用的技术软件问题指出解决方案，例如在应用程序（或应用程序套件）中怎样处理验证、安全和事务数据。前几期主要关注获取技术惯用模式所用的技术，例如元程序设计。域模式关注的是如何抽象常见业务问题。而技术模式几乎出现在所有的软件中，域模式之间的差异与业务之间的差异一样大。然而获取它们有一套丰富的技术，这就是本期以及后续几期将要谈论的话题。

##### 关于本系列

本 [系列](http://www.ibm.com/developerworks/cn/java/j-eaed/)旨在从全新的视角来介绍经常讨论但是又难以理解的软件架构和设计概念。通过具体示例，Neal Ford 将帮助您在 演化架构和 紧急设计的灵活实践中打下坚实的基础。通过将重要的架构和设计决定推迟到最后责任时刻，您可以防止不必要的复杂度降低软件项目的质量。

本文为使用 DSL 技术作为一种抽象样式获取域模式提供动力，DSL 提供多种选择，包括自己命名的模式。Martin Fowler 最近的一本书对 DSL 技术有较为深入的研究（见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#artrelatedtopics)）。在后续几期中，我会使用他的许多模式名，也会将他的示例用于我的例子中，逐步讲述具体技术。

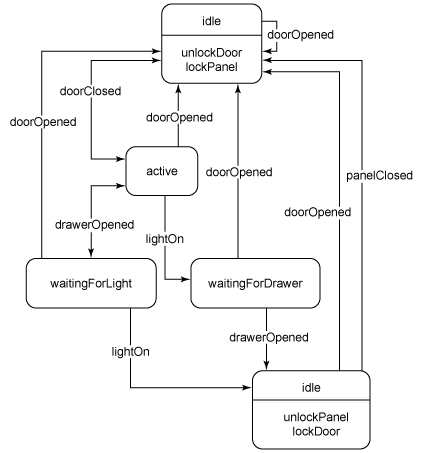
## DSL 的动机

为什么我们要费那么多周折创建一个 DSL ？仅仅为了获取一个惯用模式？正如我在 “[利用可重用代码，第 2 部分](http://www.ibm.com/developerworks/cn/java/j-eaed12/)” 中指出的，区分惯用模式的最好方法就是让它看起来与其他代码不一样。这种看得见的不同是最直接的线索，您不需要再看常规 API。同样，使用 DSL 的目的之一是写代码，使这些代码看起来不像源代码而更像您正在尝试解决的问题。如果能够达到这个目标（或者接近这个目标），那您将填补软件项目中的这一空白，为开发人员和业务涉众的沟通架起了桥梁。允许用户阅读您的代码很有必要，因为这消除了将代码转换为语言的需求，这是项很容易出错的工作。让您的代码对于非技术人员是易读的，因为他们了解软件的预期设想，这样你们之间就会有更多的交流。

为激励开发人员使用这种技术，我将借用 Fowler DSL 书中的例子（见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#artrelatedtopics)）。假设我正为一家制作软件控制的暗格（secret compartments，想想 James Bond）的公司工作。公司的一个客户，H. 夫人，想要在她的卧室里装一个暗格。然而我们公司使用 .com 泡沫破碎后留下的 Java™驱动的 toasters 来运行软件。尽管 toasters 比较便宜，但更新其中的软件却很昂贵，因此我需要创建基础暗格代码，然后将其永久地设置在 toasters 上，然后找到一种方法根据每位用户的需求进行配置。您也知道，在现代软件世界中，这是一个很常见的问题：普遍行为不常改变，而配置需要根据个人情况进行改变。

H. 夫人想要一个暗格，打开这个暗格的方法是：首先关闭卧室门，接着打开梳妆台的第二个抽屉，最后打开床头灯。这些活动必须依次进行，如果打乱次序，必须重头开始。您可以将控制暗格的软件想象为一个状态机，如图 1 所示：

##### 图 1. H. 夫人的暗格是状态机



基本状态机 API 比较简单。创建一个抽象事件类，可以同时处理状态机内的事件和命令，如清单 1 所示：

##### 清单 1. 状态机的抽象事件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public 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;} |

可以使用另一个简单类 States对状态机中的状态建模，如清单 2 所示 :

##### 清单 2. 状态机类的开始部分

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public class States {   private State content;   private List<TransitionBuilder> transitions = new ArrayList<TransitionBuilder>();   private List<Commands> commands = new ArrayList<Commands>();     public States(String name, StateMachineBuilder builder) {     super(name, builder);     content = new State(name);   }     State getState() {     return content;   }     public States actions(Commands... identifiers) {     builder.definingState(this);     commands.addAll(Arrays.asList(identifiers));     return this;   }       public TransitionBuilder transition(Events identifier) {     builder.definingState(this);     return new TransitionBuilder(this, identifier);   }     void addTransition(TransitionBuilder arg) {     transitions.add(arg);   }       void produce() {     for (Commands c : commands)       content.addAction(c.getCommand());     for (TransitionBuilder t : transitions)       t.produce();   }  } |

[清单 1](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#listing1)和 [清单 2](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#listing2)仅做参考。需要解决的问题是如何表示状态机的配置。这种表示是安装暗格的一种惯用模式。清单 3 展示了状态机基于 Java 的配置：

##### 清单 3. 一个配置选择：Java 代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | 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); |

[清单 3](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#listing3)显示了使用 Java 进行状态机配置的几个问题。首先，阅读这些 Java 代码并不能明确知道这就是状态机配置，和多数 Java API 一样，这只是一堆没有差别的代码。第二，冗长且重复。为状态机的每部分设置更多的状态和转换时，变量名重复使用，所有这些重复使代码难于阅读。第三，代码不能满足最初目标 —— 无需重新编译就可配置暗格。

事实上，在 Java 世界几乎看不到这种代码了，现在流行使用 XML 编写配置代码。用 XML 编写配置很简单，如清单 4 所示：

##### 清单 4. 用 XML 编写的状态机配置

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | <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> |

[清单 4](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#listing4)中的代码相比 Java 版本有几个优势。

1. 延迟绑定，这意味着可以修改代码并将其放进 toaster，可以使用 XML 解析器阅读配置。

第二，对于这个特定问题，这段代码是更富于表现力，因为 XML 包含容器（containership）概念：States 将它们的配置包含为子元素。这有助于删除 Java 版本中令人讨厌的冗余。

第三，代码本质上是声明式的。通常，如果您只是进行声明而不需要 if和 while语法，声明式代码更易于阅读。

暂时退后一步，先理解其含义。外化配置在现代 Java 世界中是一种很常见的模式，我们不再认为它是独特实体。实际上这也是每个 Java 框架的特征。配置是一个惯用模式，我们需要捕获方式，使其区别于周围框架的一般行为，并将其分离出来。使用 XML 进行配置，我是使用外部 DSL 编写代码的（句法 [syntax] 是 XML，语法 [grammar] 是由 XML 相关模式定义的），因此不需要重新编译框架代码对其进行转换。

我们没有必要因为 XML 的优势，总是使用 XML。可以考虑以下配置代码，

如清单 5 所示：

##### 清单 5. 定制语法（custom-grammar）的状态机配置

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | 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 |

XML 版本有的优势，它也有：是声明式的，有容器概念，并且是简明的。同时它也超越了 XML 和 Java 版本，因为它很少有 噪音字符（例如 <和 >），尽管这对技术实现是必需的，但是影响可读性。

此版配置代码是一个用 ANTLR 编写的定制外部 DSL，也是一个开源工具，它使得用自定义语言编写变得很容易（见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#artrelatedtopics)）。曾经在大学时候不喜欢编译器（包括诸如 Lex 和 YACC 之类的经典工具）课程的人，将很高兴知道这些工具已经变得好多了。这个例子来自 Fowler 的书中，他说构建 XML 版本和构建定制语言版本所用时间相同。

清单 6 中的是用 Ruby 写的另一种可选版本 :

##### 清单 6. JRuby 中的状态机配置

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | 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 |

这是一个很好的 内部DSL 例子：DSL 使用基础语言的语法，这意味这个 DSL 必须是符合语法的 Ruby 代码。（因为它是用 Ruby 编写的，可以使用 JRuby 运行，就是说，您的 toaster 所需的全是 JRuby JAR 文件。）

[清单 6](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#listing6)同定制语言有许多相同的优点。注意，大量使用 Ruby 块充当容器，这能给您同 XML 和定制语言版本一样的容器语义。它比定制语言使用更少的噪音字符（noise characters）。例如，在 Ruby 中 ：前缀表明一个符号，在本例中基本上是用作标识符的不变字符串。

使用 Ruby 实现这类 DSL 相当简单，如清单 7 所示：

##### 清单 7. JRuby DSL 的部分类定义

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class StateMachineBuilder   attr\_reader :machine, :events, :states, :commands     def initialize     @events = {}     @states = {}     @state\_blocks = {}     @commands = {}   end     def event name, code     @events[name] = Event.new(name.to\_s, code)   end     def state name, &block     @states[name] = State.new(name.to\_s)     @state\_blocks[name] = block     @start\_state ||= @states[name]   end     def command name, code     @commands[name] = Command.new(name.to\_s, code)   end |

Ruby 语法比较灵活，这使它适用于此类 DSL。例如，声明一个事件时，不会强制包含一个圆括弧作为方法调用的一部分。在这个版本中，不需要编写自己的语言或者用尖括弧妨碍自己。这更能说明为什么这个方法在 Ruby 世界是如此流行。

## DSL 特征

DSL 为捕获惯用模式提供了很好的可供选择的语法。正如 Martin Fowler 所定义的，DSL 有 5 个主要特征。

### 计算机编程语言

要成为一个 DSL，这个语言必须是一个计算机编程语言。如果没有这一限制，容易引起 “滑坡”，您遇到的所有事物都有可能是一个 DSL。如果您定义 DSL 术语太广泛，所有的上下文会话都可能是 DSL。例如，我有些同事是板球迷， 当我同他们在一起时，他们总是不停的谈论板球，尽管他们是用英语，我也不明白他们在说什么。我缺乏适当的上下文，以至于我不能明白他们所用的单词。然而，我们可以使用 DSL 术语谈论板球和其他运动。但是如果没有范围定义，很难将其缩小到可用约束范围内 —因此 Fowler 坚持将其限制在计算机编程语言范围之内。

### 语言天性

Fowler 关于 DSL 的第二条准则是，它应该有 “语言天性”，这意味您的 DSL 对于非程序员至少是隐约可读的。语言天性包含多种格式，在后续几期中，我将向您展示其中的一些，我将继续探索 DSL —— 作为一种捕获惯用模式的方法 —— 的引用。

### 领域焦点

要成为一个合适的 DSL，该语言必须只关注一个特定的问题领域，尝试创建 DSL 的风险之一是使其太宽泛。DSL 是一个抽象机制，创建太宽泛的抽象会降低它的优势。

### 有限的表现力

限制表现力也是 DSL 的一个特点。很少能找到一个 DSL 含有诸如循环和判定的控制结构。DSL 应该特别关注它正在尝试描述的领域，而且也只能关注该领域。因此，相当多的 DSL 是声明式的，而不是指令式的。

### 非图灵完整的（Turing complete）

前面两个标准暗示了这一特征，但是在这里，我将正式确认它。您的 DSL 应当不是图灵完整的（见 [参考资料](https://www.ibm.com/developerworks/cn/java/j-eaed13/index.html#artrelatedtopics)）。事实上，人们认为在 DSL 中一个反模式将意外地变成图灵完整的。例如，经典的 UNIX®sendmail配置文件就是意外图灵完整的。您可以在 sendmail配置文件中写一个操作系统，如果您愿意，而且又有很多时间的话。

意外地变成图灵完整的是惊人的简单。一些熟悉的基础设施工具可以意外地进行这种转变 —例如，XSLT。确定一种语言是否是 DSL，有时候取决于其上下文。使用 XSLT 来将一种版本的本文转换成另一个版本的文本时，您就是将它作为 DSL 使用的。如果您使用 XSTL 解决汉诺塔问题，您是将它作为一种图灵完整语言使用的（并且您可能会找到一种新的爱好）。

## 结束语

这一期为使用 DSL 作为一种获取惯用模式的提取机制奠定了基础。DSL 在这方面做得很不错，因为它们很容易与常规 API 区分开，更倾向于声明式的，并改善了项目中开发人员与非开发人员之间的信息交流和反馈。下一期，我将探索多种构建 DSL 的技术。在后续几期中，我将逐一介绍几种可用于寻求发现和设计代码的 DSL 技术。