

# DSL 的误区

DSL 时不时地会成为一个话题，所以今天想专门说一下。

DSL 也就是 Domain Specific Language 的简称，是指为特定领域（domain）设计的专用语言。举个例子，Linux 系统下有很多配置文件，每个配置文件格式都不大一样，它们可以被看成是多种 DSL。IP Tables 的规则是一种 DSL，FWWM 窗口管理器的配置文件是一种 DSL，VIM 和 Emacs 的配置文件，当然也是 DSL。Makefile 是 DSL。CSS 是 DSL。JSON 是 DSL。SQL 也可以被看成是数据库领域的 DSL。也有很多人在自己的工作中创造 DSL，用它们来解决一些实际问题。

由于自己的原则，我个人从来没有设计过 DSL，但我用过别人设计的 DSL，并且对此深有感受。我觉得人们对于 DSL 有挺多的误解，所以我今天想分享一下自己对 DSL 的看法和亲身经历。

## DSL 与库代码 (Library)

开门见山说说对 DSL 的看法吧。我觉得大部分 DSL 都是不应该存在的，我们应该尽量避免创造 DSL。这一论点不但适用于只有少量用户的产品内部 DSL，也适用于像 SQL 这样具有大量从业者的 DSL。

DSL 这名字本身就是一种误导，它让人误以为不同“领域”（domain）的人就该用不同的语言，而其实不是那样的。这不过是在制造领域壁垒，为引入不必要的 DSL 找借口。绝大部分所谓不同“领域”，它们对语言的基本需求都是一样的。很多时候人们误以为需要新的 DSL，是因为他们没有分清“库代码”（library）和“新语言”（language）的差别。

不同领域需要的，绝大部分时候只是针对该领域写出的“库代码”，而不是完全不同的“新语言”。分析大部分所谓 DSL，你会发现它们不过提取了通用程序语言里的一部分，比如结构定义，算术表达式，逻辑表达式，条件语句，等等。极少有 DSL 是不能用通用的程序语言构造表示的。绝大部分时候你都可以用一种通用的语言，写出满足领域需求的库代码，然后领域里的人就可以调用库函数来完成他们的任务。

绝大部分 DSL 的存在，都是因为设计它的人没有理解问题的本质，没有意识到这问题并不需要通过设计新的语言来解决。很多人设计 DSL，是因为看到同类产品里面有 DSL，所以就抄袭照搬。或者因为听说 DSL 很酷，设计出 DSL 会显得自己很厉害，很有价值。同时，设计 DSL 还可以让同事和公司对自己产生依赖性。因为有人用我的 DSL，所以公司需要我，离不开我，那么 job security 就有所保证；)

然而如果你仔细分析手头的问题，就会发现它们绝大部分都可以用库代码，利用已有的语言来解决。就算类似的产品里面实现了 DSL，你会发现它们绝大部分也可以用库代码来代替。在自己的工作中，我一般都首先考虑写库代码来解决问题，实在解决不了才会考虑创造 DSL。

因为遵循这一原则，加上对问题透彻的理解，我发现自己几乎每次都能用库代码解决问题，所以我从来没有在自己的职业生涯中创造过 DSL。

## “新语言问题” (The New Language Problem)

现在我来讲一下，盲目创造 DSL 带来的问题。很多人不明白 DSL 跟库代码的区别，拿到一个问题也不想清楚，就一意孤行开始设计 DSL，后来却发现 DSL 带来了严重的问题。由于 DSL 是一种新的语言，而不只是用已有语言写出来新函数，所以 DSL 必须经过一个学习和理解的过程，才能被其他人使用。

举个例子。如果你看到 `foo(x, y + z)` 这样的库代码，很显然这是一个函数调用，所以你知道它会先计算 `y+z`，得到结果之后，把它传递给 `foo` 函数作为参数，最后得到 `foo` 函数算出来的结果。注意到了吗，你并不需要学习新的语言。虽然你不知道 `foo` 函数的定义，然而你很清楚函数调用会做什么：把参数算好放进去，返回一个结果。也就是说，你对函数调用已经有一个“心理模型”。

可是一个 DSL 就很不一样，对于一个新的 DSL 构造，你也许没有任何心理模型存在。同样看到 `foo(x, y + z)`，它的含义也许根本不是一个函数调用。也许 `foo` 在这个 DSL 里就表示 `foreach` 循环语句，那么 `foo(x, y + z)` 表示类似 Java 的 `foreach (x : y + z)`，其中 `y` 和 `z` 都是链表，`+` 号表示连接两个链表。

这样一来，为了理解 `foo(x, y + z)` 是什么意思，你不能直接通过已有的，关于函数的心理模型，而必须阅读 DSL 设计者给你的文档，重新学习。如果 DSL 设计者是有素养的语言专家，那也许还好说。然而我发现绝大部分 DSL 设计者，都没有受到过专业的训练，所以他们设计出来的语言，从一开始就存在各种让人头痛的问题。

有些 DSL 表达力太弱，所以很多时候用户发现没法表达自己的意思。每当需要用这 DSL 写代码，他们就得去请教这个语言的设计者。很多时候你必须往这个 DSL 添加新的特性，才能解决自己的问题。到后来，你就发现有人设计了个 DSL，到头来他自己是唯一会用这 DSL 的人。每当有人需要用一个语言，就得去麻烦它的作者，那么这个语言的存在还有什么意义？

当然，很多 DSL 还会犯下程序语言设计的一些常见问题。很多人把设计语言想得太容易，喜欢耍新花样，到后来就因此出现各种麻烦事。容易出错，产生歧义，语法丑陋繁琐，难学难用，缺乏编辑器 IDE 支持，出错信息难以理解，无

法用 debugger 调试，等等。最后你发现还不如不要设计新的语言，使用已有的语言来解决问题就可以了。

## 最强大的 DSL 实现语言

有些人很崇拜 Haskell 或者 Scala，说这两个语言有着非常强大的“DSL 实现能力”，也就是说你可以用它们来实现自己想要的 DSL。这是一种误解。虽然我已经指出创造 DSL 并不是什么好事，我觉得还是应该把这个问题说清楚。如果你跟我一样看透了各种语言，就会发现世界上最强大的 DSL 实现语言，并不是 Haskell 或者 Scala，而是 Scheme。

2012 年的时候，我参加了 POPL 会议（Principles of Programming Languages），这是程序语言界的顶级会议。虽然名字里面含有 principle（原理）这个词，明眼人都看得出来，这个会议已经不是那么重视根本性的“原理”，它已经带有随波逐流的商业气息。那时候 Scala 正如日中天，所以在那次会议上，Scala 的 paper 简直是铺天盖地，“Scala 帮”的人趾高气昂。当然，各种 JavaScript 的东西也是如火如荼。

很多 Scala 人宣讲的主题，都是在鼓吹它的 DSL 实现能力。听了几个这样的报告之后，我发现 Scala 的 DSL 机制跟 Haskell 的挺像，它们不过是实现了类似 C++ 的“操作符重载”，利用特殊的操作符来表达对一些特殊对象的操作，然后把这些操作符美其名曰为“DSL”。

如果你还没看明白 Haskell 的把戏，我就提醒你一下。Haskell 的所谓 type class，其实跟 Java 或者 C++ 的函数重载（overloading）本质上是一回事。只不过因为 Haskell 采用了 Hindley-Milner 类型系统，这个重载问题被复杂化，模糊化了，所以一般人看不出来。等你看透了就会发现，Haskell 实现 DSL 的方式，不过是通过 type class 重载一些特殊的操作符而已。这跟 C++ 的 operator+(...) 并没有什么本质区别。

操作符重载定义出来的 DSL，是非常有局限性的。实际上，通过重载操作符定义出来的语言，并不能叫做 DSL，而只能叫做“库代码”。为什么呢？因为一个语言之所以成为“语言”，它必须有自己独特的语义，而不只是定义了新的函数。重载操作符本质上只是定义了新的函数，而没有扩展语言的能力。就像你在 C++ 里重载了 + 操作符，你仍然是在使用 C++，而不是扩展了 C++ 的语义。

我用过 Haskell 实现的一个用于 GPU 计算的“DSL”，名叫 Accelerate。这个“语言”用起来相当的蹩脚，它要求用户在代码的特定位置写上一些特殊符号，因为只有这样操作符重载才能起作用。可是写上这些莫名其妙的符号之后，你就发现代码的可读性变得很差。但由于操作符重载的局限性，你必须这样做。你必须记住在什么时候必须写这些符号，在什么时候不能写它们。这种要求对于程序员的头脑，是一个严重的负担，没有人愿意去记住这些不知所云的东西。

由于操作符重载的局限性，Haskell 和 Scala 实现的 DSL，虽然吹得很厉害，发了不少 paper，却很少有人拿来实用。

世界上最强大的 DSL 实现语言，其实非 Scheme 莫属。Scheme 的宏系统（hygienic macro）超越了早期 Lisp 语言的宏系统，它本来就是被设计来改变和扩展 Scheme 的语义的。Scheme 的宏实质上是在对“语法树”进行任意变换，扩展编译器的功能，所以你可以利用宏把 Scheme 转变成几乎任何你想要的语言。这种宏系统不但可以实现 Haskell 和 Scala 的“重载型 DSL”，还能实现那些不能用重载实现的语言特性（比如能绑定变量的语句）。

miniKanren 就是一个用 Scheme 宏实现的语言，它是一个类似 Prolog 的逻辑式语言。如果你用 Haskell 或者 Scala 来实现 miniKanren，就会发现异常的困难。就算实现出来了，你的 DSL 语法也会很难看难用，不可能跟 miniKanren 一样优雅。

我并不是在这里鼓吹 Scheme，搞宣传。正好相反，对 Scheme 的宏系统有了深入理解之后，我发现了它带来的严重问题。内行人把这个问题称为“新语言问题”（The New Language Problem）。

因为在 Scheme 里实现一个新语言如此的容易，几行代码就可以写出新的语言构造，改变语言本来的语义，所以这带来了严重的问题。这个问题就是，一旦你改变了语言的语义，或者设计出新的语言构造，人们之间的交流就增加了一道障碍。使用你改造后的 Scheme 的人，必须学习一种新的语言，才能看懂你的代码，才能跟你交流。

由于这个原因，你很难看懂另一个人的 Scheme 代码，因为很多 Scheme 程序员觉得宏是个好东西，所以很喜欢用它。他们设计出稀奇古怪的宏，扩展语言的能力，然后使用扩展后的，你完全不理解的语言来写他的代码。本来语言是用来方便人与人交流的，结果由于每个人都可以改变这语言，导致他们鸡同鸭讲，没法交流！

再次声明，我不是在这里称赞或者宣扬 Scheme，我真的认为宏系统的存在是 Scheme 的一个严重的缺点。我那热爱 Scheme 的教授们知道了，一定会反对我这种说法，甚至鄙视我。但我确实就是这么想的，这么多年过去了，仍然没有改变过这一看法。

Scheme 宏系统的这个问题，引发了我对 DSL 的思考。后来我发现所谓 DSL 跟 Scheme 宏系统，存在几乎一模一样的问题。这个问题有一个名字，叫做“新语言问题”（The New Language Problem）。下面我详细解释一下这个问题。

## NaCl 的故事

现在我来讲一个有趣的故事，是我自己跟 DSL 有关的经历。

在我曾经工作过的某公司，有两个很喜欢捣鼓 PL，却没有受过正规 PL 教育的人。说得不好听一点，他们就是“PL 民科”。然而正是这种民科，最喜欢显示自己牛逼，喜欢显示自己有能力实现新的语言，以至于真正的专家只好在旁边静静地看着他们装逼 :P

他们其中一个人知道我是研究 PL 的，开头觉得我是同类，所以总喜欢走到桌前对我说：“咱们一起设计一个通用程序语言吧！然后用它来解决我们公司现在遇到的难题！”每当他这样说，我都安静的摇摇头：“我们公司真的需要一个新的语言吗？你有多少时间来设计和实现这个语言？”

当时这两个人在公司里，总是喜欢试用各种新语言，Go 语言，Scala，Rust，..... 他们都试过了。每当拿到一个新的项目，他们总是想方设法要用某种新语言来做。于是乎，这样的历史就在我眼前反复的上演：

1. 为一种新语言兴奋，开始用它来做新项目
2. 两个月之后，开始骂这语言，各种不爽
3. 最后项目不了了之，代码全部丢进垃圾堆
4. Goto 1

这两个家伙每天就为这些事情忙得不亦乐乎，真正留下来的产出却很少。之前他们还设计了一种 DSL，专门用于对 HTML 进行匹配和转换。这个 DSL 被他们起了一个很有科学味的名字，叫做 NaCl（氯化钠，食盐的化学分子式）。

我进公司的时候，NaCl 已经存在了挺长一段时间，然而很少有人真正理解它的用法，大部分人对它的态度都是“能不碰就不碰”。终于有一天，我遇到了需要修改 NaCl 代码的时候。也就一行代码，看了半天 NaCl 的“官方文档”，却不知道如何才能用它提供的语法，来表达我所需要的改动。其实我需要的不过是一个很容易的匹配替换，完全可以用正则表达式来完成，可是已有的代码是用 NaCl 写的，再加上好几层的框架，让你绕都绕不过，所以我不知道怎么办了。

问了挺多人，包括公司里最顶级的“NaCl 专家”，都没能得到结果。最后，我不得不硬着头皮去打扰两位日理万机的“NaCl 之父”。叽里呱啦跟我解释说教了一通之后，眨眼之间噼里啪啦帮我改了代码，搞定了！其实我根本没听明白他在说什么，为什么那样改，也不知道背后的原理。总之，我一个字都没打，目的就达到了，所以我就回去做自己的事情了。

后来跟其他同事聊，发现我的直觉是很准的。他们告诉我，公司里所有 NaCl 代码可以表达的东西，都可以很容易的用正则表达式替换来解决，甚至可以用硬邦邦的，不带 regexp 的字符串替换来解决。同事们都很不理解，为什么非得设计个 DSL 来做这么简单的事情，本来调用 Java 的 String.replace 就可以完成。

后来“NaCl 专家”告诉我，虽然他很了解 NaCl，却根本不喜欢它。在那两个家伙提出要设计 NaCl 的时候，他就已经表示了强烈的反对，他觉得不应该创造 DSL 来解决这样的问题。当时他就给大家解释了什么是“新语言问题”，警告大家新语言会带来的麻烦。可是领导显然跟这两个家伙有某种政治上的联盟关系，所以根本没听他在说什么。

在领导的放任和支持下，这两个家伙一意孤行创造了 NaCl，然后强行在全公司推广。到后来，每次需要用 NaCl 写点什么，就发现需要给它增加新的功能，就得去求那两个家伙帮忙。所以我能用上今天的 NaCl，基本能表达我想要的东西，还多亏了这位“NaCl 专家”以前栽的跟头，他把各种坑基本给我填起来了；)

我有一句格言：如果一个语言，每当用户需要用它表达任何东西，都得去麻烦它的设计者，甚至需要给这个语言增加新的功能，那这个语言就不应该存在。NaCl 这个 DSL 正好符合了我的断言。

当然 NaCl 只是一个例子，我知道很多其它 DSL 的背后都有类似的故事。几个月之后，这两个民科又开始创造另一个 DSL，名叫 Dex，于是历史又开始重演.....

## 动态逻辑加载

Dex 的故事跟 NaCl 有所不同，但最后的结果差不多。NaCl 是一个完全不应该存在的语言，而 Dex 的情况有点不一样。我们确实需要某种“嵌入式语言”，只不过它不应该是 Dex 那个样子，不应该是一个 DSL。由于 Dex 要解决的需求有一定的代表性，很多人在遇到这类需求的时候，就开始盲目的创造 DSL，所以这是一个很大的坑！我想把这个故事详细讲一下，免得越来越多的人掉进去。

原来的需求是这样：产品需要一种配置方式，配置文件里面可以包含一定的“逻辑”。通过在不更换代码的情况下动态加载配置文件，它可以动态的改变系统的逻辑和行为。这东西有点像“防火墙”的规则，比如：

1. 如果尺寸大于 1000，那么不通过，否则通过。
2. 如果标题含有“猪头”这个词，不通过，否则通过.....

这些规则从本质上讲，就是一些逻辑表达式“size > 1000”，加上一些分支语句“if ... then ...”。在 Dex 出现之前，有人用 XML 定义这样的规则，后来发现 XML 非常不好理解，像是这个样子：

```
<rule>
  <condition>
    <operator>gt</operator>
    <first>size</first>
```

```
<second>1000</second>
</condition>
<action>block</action>
</rule>
```

看明白了吗？这个看得人眼睛发涨的 XML，表达的不过是普通语言里面的 `if (size > 1000) block()`。为了理解这一点，你可以把这个 XML 所表示的“数据结构”，想象成编译器里面的“抽象语法树”（AST）。所以写这个 XML，其实是在用手写 AST，那当然是相当痛苦的。

那我们为什么不把 `if (size > 1000) block()` 这条语句直接写到系统的 Java 代码里面呢？因为 Java 代码是编译之后放进系统里面的，一旦放进去就不能随时换掉了。然而我们需要可以随时的，“动态”的替换掉这块逻辑，而不更新系统代码。所以你不能把这条 Java 语句“写死”到系统代码里面，而必须作为“配置”。

想清楚了这一点，你就自然找到了解决方案：把 `if (x > 1000) block()` 这样的 Java 代码片段写到一个“配置文件”里，然后使用 JVM 读取，组合，并且编译这个文件，动态加载生成的 class，这样系统的行为就可以改变了。实际上这也就是公司里另外一个团队做过的事情，让用户编辑一个基于 Java 的“规则文件”，然后加载它。

我觉得这还不失为一个可行的解决方案。为了实现动态逻辑加载，你完全可以用像 Java 或者 JavaScript 那样已有的语言，利用已有的编译器来达到这个目的，而不需要设计新的语言。然而当 PL 民科们遇到这样的问题，他们首先想到的是：设计一个新的 DSL！于是 Dex 就诞生了。

Dex 要表达的东西，本质上就是这些逻辑表达式和条件语句，然而 Dex 被设计为一个完全独立的语言。它的语法被设计得其它语言很不一样，结合了 Haskell，Go 语言，Python 等各种语言语法的缺陷。后来团队里又进来一个研究过 Prolog 逻辑式语言的人，所以他试图在里面加入“逻辑式编程”的元素。总之他们有很宏伟的目标：让这个 DSL “可靠”，“可验证”，成为“描述式语言”……

他们向团队宣布这个雄心勃勃的计划之后，一个有趣的插曲发生了。听说又要创造一个 DSL，“NaCl 专家”再次怒发冲冠，开始反对这个计划。这一次他拿出了实际行动，自己动手拿 Java 内嵌的 [JavaScript 解释器](#)，做了一个基于 JavaScript 的动态配置系统，只开发了一个星期就可以用了。

我觉得用 JavaScript 也不失为一个解决方案，毕竟浏览器的 PAC 文件就是用 JavaScript 定义代理规则的，而这种代理规则跟我们的应用非常类似。我虽然没有特别喜欢 JavaScript，但它其中的一些简单构造用在这种场景，是没什么大问题的。

其实在此之前我也看不下去了，所以自己悄悄做了一个类似的配置系统，拿已有的 JavaScript parser，提取 JavaScript 的相关构造，做了一个解释器，嵌入到系统里，只花了一天时间。但我心里很清楚，一切技术上的努力在政治斗争的面前，都是无足轻重的。我早已经伤不起了，在好心人的帮助下，我离开了这个团队，但暗地里我仍然从精神上支持着“NaCl 专家”继续他的抗争。

争吵的最后结果，当然是由于领导偏心庇护，否决了“外人”的作法，让两个民科和一个 Prolog 狂人继续开发 Dex。几个月之后，公司的第二个奇葩 DSL 诞生了。它用混淆难读的方式，表达了普通语言里的条件语句和逻辑表达式。他们为它写了 parser，写了解释器，写了文档，开始在公司强行推广。“可靠”，“可验证”，“描述式”的目标，早已被抛到九霄云外。用的人都苦不堪言，好多东西没法表达或者不知道如何表达，出错了也没有足够的反馈信息，每次要写东西就得去找“Dex 之父”们。

嗯，历史就这样重演了……

## 结论

所以，我对于 DSL 的结论是什么呢？

1. 尽一切可能避免创造 DSL，因为它会带来严重的理解，交流和学习曲线问题，可能会严重的降低团队的工作效率。如果这个 DSL 是给用户使用，会严重影响用户体验，降低产品的可用性。
2. 大部分时候写库代码，把需要的功能做成函数，其实就可以解决问题。
3. 如果真的到了必须创造 DSL 的时候，非 DSL 不能解决问题，才可以动手设计 DSL。但 DSL 必须由程序语言专家来完成，否则它还是可能给产品和团队带来严重的后果。
4. 大部分 DSL 要解决的问题，不过是“动态逻辑加载”。为了这个目的，你完全可以利用已有的语言（比如 JavaScript），或者取其中一部分构造，通过动态调用它的解释器（编译器）来达到这个目的，而不需要创造新的 DSL。

（本文建议零售价 ¥30，如果它让你的团队或者公司幸免落坑，请付款 ¥1000000 :)）