时刻关注企业软件开发领域的变化与创新

架构师

8月 ARCHITECT

开发减速，
是为了赢利提速

“服务重用”是否
被过度使用？

云计算虚拟研讨会

面向服务的经济学

架构师修炼之道

InfoQ中文站
www.infoq.com/cn

每月8号出版

篇首语

技术的乐趣

在中国的技术圈子里，流行着这样一种说法：过了三十五岁，就一定得改行。在技术飞速发展的今天，只要稍不留神，就会掉下队来。因此，诸多技术工作者在仔细权衡利弊之后，终于还是决定跳离技术这个是非之地，将工作机会让给那些更青春更朝气的年轻一辈们。

当然，还是有相当一部分的技术从业人员对这种说法很不以为然。一方面，技术工作的人际关系相对简单，这对于不喜欢权术政治的人来说，无疑有很大的吸引力。有位朋友曾经跟我说："比起和人打交道，我更愿意和机器打交道"，相信抱有类似观点的朋友不在少数。而另一方面，技术给人带来的乐趣和满足感，往往是最吸引人的地方，让人沉迷其中，难以自拔。通过指尖流淌出的一行一行代码，拥有超乎想象的力量，不但可以承载数亿次的访问，还能够猜到你喜欢读什么书，喜欢听哪种音乐。这些代码，才是整个世界信息化建设的基石；而技术人员，才是 IT 行业真正的核心和价值所在。

有一个形容，说程序员就是魔法师，只要在键盘上输入正确的咒语，系统中相互啮合的齿轮便开始运转起来。技术很大程度上满足了人类原始的创造欲。还记得第一次成功编译一段代码，在屏幕上打印出"Hello World"的喜悦吗？那种愉悦有多纯粹多美妙，相信只有技术人员自己才能体会得到吧。通过架构改造，让一个系统的吞吐量翻了几倍；通过代码重构，让一个项目的代码量变成原来的几分之一；通过算法创新，让识别用户自然属性和预测用户行为成为可能。以上种种，皆是乐趣，妙趣横生。

喜爱，方能专注；专注，方能成功。与其做无谓之争，不如踏踏实实静下心里，品味技术的乐趣，寻找简单的快乐。用自己的双手改变世界，帮助他人。正是这种自豪感，支撑着我在技术上一路走来，相信还会伴随着我继续走下去。不知道作为读者的你，又是如何抉择的呢？

本期主编：李明

目　录

[新品推荐]

[架构师大家谈]

[封面植物]

[人物专访]

Simon探讨编程语言与研究工作

在伦敦 2008QCon的采访中，计算机科学家和研究员Simon Peyton Jones讨论了函数式编程语言的属性，特别是Haskell，它对主流语言的一些特性产生了启发。他还给出了自己对于语法和语言复杂性的观点，并谈到了一些关于数据并行和事务性内存的研究工作。

InfoQ：和我在一起的是 Simon Peyton Jones，我们将对编程语言作总体的探讨，特别是 Haskell。Simon，跟我们介绍下你自己，说说最近都在忙些什么吧？

Simon :我任职于微软剑桥研究院。我已在那里工作 10 年了。在此之前我是大学计算机教授。我转向微软以追求变化，因此感觉上我像是大学里面休假的人员，区别是一直都在休假。我得以能够做我自己喜欢的事，而我喜欢的事就是研习函数式编程语言，因为我乐衷于让计算机变得更易编程。此刻，我认为函数式语言至少提供了一个重要的方式使得计算机最终能得以更容易地编程。我所做的一切都跟一种叫 Haskell 的语言有关。Haskell 并不是我直接研究的对象，而是我其它研究的基石。我们拥有 Haskell，这一编程语言，与在 Marlow 的同事 - 我们为 Haskell 构建了名为 GHC 的编译器 - 以及许许多多使用它的人们一道。现在我可以研究一些新的编程语言特性，比如事务性内存。我们将其融入了 GHC 并加以精心打造，在下一版本发布中，许多人就可以开始使用它了；我们得到了很多良好的反馈。对于研究编程语言特性如何工作来说，这是一个非常好的反馈。

InfoQ：前一位微软研究员(指 QCon2008 上)试图找到一种多范型的语言，其本身实际上是函数式编程语言，同时也是带有某种改变的对象编程语言，比如 F#和 Scala；你能对此详细阐释一下吗？这能够解决我们目前主流环境中所面临的问题吗？

Simon：我认为两种不同的事物将继续：一是你可以从所熟悉的主流语言开始，其本质是面向对象编程，你可以尝试将来自函数式编程的优秀元素添加进去，通常是并发，这将把你带入一个良好的方向。这正是 F#所做的，而在一定程度上 C#本身也与之一样，逐渐发展着许多受函数式语言启发而来的特性。但不管怎样，你所做的始终都是一个折衷。从一开始，我们并不拿走什么东西，我们只是添加。

这使得整个企业更加复杂 - 这本就是一个够繁杂的开始了 - 而如果把事物隔离开来会更容易研究，因此我将那里所发生的看作是我们正处在编程世界的引力中心，也正是像我这样的人所做的。让我们采用一个更原则性的方式：特别的就 Haskell 而言这意味着对于纯粹性的严肃 - 没有副作用 - ，这并不是因为它能够解决目前所有的问题，而是因为它值得人们来研究，就像在一个实验室的条件下一样。我用刚才我所作的关于交互虚拟化的报告里的一个例子来说明我们如何采取这一方式：我从主流世界里获取了事务性内存，在 Haskell 的世界里研究它并从中领悟到新的东西，又可以移植回去。我不认为这一方式是相互竞争，谁比谁更好，而是将其看作相互补充并且最终都是朝着同一个目的地。举例，C#加入了一些函数式编程的结构 ;你认为语法重要吗？你用 C#语法进行函数式编程 ,不会觉得这会有什么影响吗？就好像是让程序员们以命令式编程的文化来编程，你看，他们由 C#获得了这一语法。

习惯性的来说语法在编程语言里并不特别重要，你一样能解决它，但我认为实际上语法是对一个编程语言接口的运用。因此它有着长期的并且普遍的作用，我认为。当然，向 C#这样本就有着非常完整的语法结构空间的语言添加新的特性 ,意味着你可能不能完全按照你选择的方式来做所有的事情。实际上，我认为这已经解决得很好 - 他们为 LinQ 和 lambda 表达式所选择的语法。除语法外，这个问题言外之意可能是“它究竟能如何方便的编写出显著的程序块。而不只是一点点函数式的小玩意儿而已？”。

在 C#里没有什么 lambda 表达式，而是与特定的 LinQ 相关，当你说：“按照这一预测，用这个布尔表达式过滤这一记录”，这就会非常有用。但这些 lambda 表达式非常小，通常都是一行，所以要将 C#发展为你可以仅用函数编写整个模块的语言，这并非是一个合适的载体。

这比语法意味着更多的东西，这是函数应用。如果你拿一个函数赋值的事物，与一个值，哪怕是做一个应用 - 不只是将它们每个挨个放在一起 - 你必须都得有一个应用操作符。因为 C#要同时完成很多事情，你也不能怪它，它做了很多像 Haskell 这样的语言没有做的事情，但如果我们试图同时做很多事情而不是任何特定的事情，这就显得不是那么方便了。是否是这样，反过来，人们对函数式编程的理解和感受可能会有偏见，比如他们会想“这也太笨拙了”，我希望不是的，我希望他们去想“这非常酷，我也许应该试一下纯粹的形式。”

InfoQ：使用像 C#或 Java 这样的语言我们创作对象，我们有着创作对象的模式。在函数式编程中我们创作函数。你对此有什么想法？同样的事情由两种范式来完成：在对象中你需要创建许多对象，在函数中你只需创作 monads 和其它的事物。你对此怎么看呢？

Simon：我并不是一个经验丰富的面向对象程序员，因此我不认为我对用面向对象语言创作大规模的对象有一个宏观的蓝图。我想你谈到的是设计模式。设计模式是对你如何组成对象的方式的一种非正规的描述。这种非正式性有其优势在于能给予它们更广泛的应用，但同时

也有其弱势，因为它不能作为一个库来验证，也不是对象；它更像是你脑海当中的一种勾划。

我经常思考设计模式在函数式语言中对应的是什么呢，因为不存在像在面向对象编程中那样的相应的函数式语言的设计模式书籍，这是为什么。因此我也没有一个绝对的回答。我想其中一个答案可能是很多组成函数的方式都可以被表达成高阶的函数，它们是将函数作为参数的函数并将它们结合在一起。它们只是更多的程序。这种意义下，我并不认为它们是什么不同的事物，它们只是更多的函数而已，但实际上它们的角色是将更多的函数粘合在一起。

让我们来看看这个例子如何：将一个把函数应用于列表的每个元素的函数映射到一个数组或集合，就是这样的例子，因为它将函数作为它的参数。但在大规模的编程中，你趋于获得更大规模的高阶函数。我不知道高阶函数是否真的囊括了面向对象程序员所有对于设计模式的看法。我仍在等待某天有位达人创作一本函数式编程的书名字叫“函数式设计模式”。

InfoQ：你能给我们详细的阐释一下你对 C#的列表内涵式(List Comprehension)实现有何看法吗？

Simon：它不是一个语言集成的查询，实际上是一个集成于语言的 monads，因为它将 monads 引入了 C#。

这是 LinQ 的聪明之处；他们本可以做的是，只需说：“获得 SQL 并将它丢进语言里并且只做这一件事”。但他们更聪明，他们找到了方法对任意的迭代器做 Sequel 一样的查询，因此能产生一个值流。而且他将其实现为可扩展的，因此你可以加入新的迭代器，可插入，并且能尽量发挥基本的 LinQ 框架，因为他们让你访问那些表达式树。说实话，Erik 是 LinQ 的专家，但如果说它是由“让我们来处理 Sequel 查询”这样的具体例子而概括出来的，是非常正确的。我相信在开发的过程中像 Erik 这样熟知 monad 的人一定会说到：“我们知道如何来概括这个”。

他可以从 Haskell 和其它相似语言中获取这一更加一般化的框架，将其运用于 C#的上下文并且在他们所能做的一般性的层面上产生重要影响。我想如果不是这一输入我们看到的可能会比现在出来的缺乏灵活。正在实现的和被设计为要实现的都是特殊地 C#化的。它们是精心设计的，如果没有认真的使用过它是无法理解它的所有分支的。但你能看到其直接的联系，而这也是他们今天所承认的。

LINQ 是列表内涵式的概念之一；除了列表外你认为还有其它的 monads 或其它的抽象可以从 Haskell 获得而被实现到主流语言当中吗?

部分而言主要的影响是简便性。在函数式编程当中构建数据结构并进行模式匹配是很方便的，而且你可以用一些与此同形的方法来达到，而在面向对象语言里，却没这么方便了。并

不是方言式的语法使人气馁；而是你必须写太多的东西了。这让你对特定的编程模式心灰意冷。我不知道慢慢的加入一些语法的支持是否会带来不同；或许是可能的。Scala 已经离此不远了，我认为。所以我想会是一个语法的模式匹配，贯穿函数式编程的早期事物(被加进到主流语言当中)。

今天早上我谈到关于作用和纯粹性的负担，我认为最终从一个纯函数世界能进入主流的最重要的事物将是约束子程序或者函数过程所带来的作用这一思想。或许只需说它是完全纯粹的，但你仍会想要限制它的作用 - 如果你牵扯到事务性内存的话 - 这样过程仅仅能读和写事务性的变量；它不能执行 IO，也不能读和写非事务性的变量。

当你有了这些保证，如果它被静态地保证，那么实现最佳方式就变得非常容易了。我猜想未来我们将会在主流语言中看到一些限制作用的方式；我不知道它看起来将会是什么样，但有太多的理由能说明它为何是如此重要了。

你认为 tree monads，或 continuation，或其它类型的 monads 能在 C#里得以实现吗？你是否认为这对业界是有益的？

这里你所指的是，在 Haskell 当中，monad 的想法通常一开始是以输入/输出的形式出现于人们的程序中。我们同样探讨过了事务性 monad，但实际上 Haskell 里并未内建任何 monads 相关的事物；编译器里不存在说“我知道这是一个 monad”，除了对"do"标记的一点点语法优惠。但这真正的意义是程序员可以自己定义新的 monads 以及你所提到的一系列的 continuations 传递；另外一个是封装状态。

封装状态意思是，有时候你要做像排序算法这样的计算，其外表上是一个纯函数。它获取 - 让我们假设 - 一个数组并产生排好序的数组。内部而言，它可能有很多的副作用，但它的外部接口是完全纯粹的因此你不能说它的实现是使用了副作用。那么你应当如何封闭才能让它的内部命令式对外部是不可见的并且得以静态地保证？其结果是，使用这一 monadic 构架你在像 Haskell 这样的语言里不需要对编译器作任何扩展。

我所说的是，比任何特定 monad 更重要的可能是，你可以定义新的 monads 或者你自己的 monad 变换器这一事实。这意味着 - 对于任何特定的应用程序 - 如果你想你的 monad 适合于你的程序，你需要裁剪它。这将会是在 C#这样的语言里非常乐于见到的一种功能，但我认为会有人相当怀疑并说：“你如何获得这些精致的抽象事物？” 为了拥有 monads 你需要拥有高阶的类型变量，比如像在普通的语言里，你可以说“list of T”；T 可以是指代 int 类型或 float 类型或者是 int 类型的列表的一个变量。所以 T 指代的是“类型”。我们同样希望 T 可以提供类型构造器，这样我们可以有 T 的列表以及 T 的数组，可能你还想要一些 M，它

可能会成为一个列表或者数组。

所以现在 M 成为了这样一种事物，它获得一个类型，并交付一个类型，它是一个类型到类型的函数，如果你高兴，它可以是一个对此抽象的变量。而这是你的.Net 所完全没有的。很重要的一点是 monads 在 Haskell 里是一般化的。我有一点还不是特别肯定的是，如何将它改装而不用给.NET 添加高阶的类型变量。也许迟早都是要加进去的，谁知道呢？但有些情况你无法仅仅把新东西塞进一种语言。这会有范型的冲突而你只会得到一团糟。

我不知道在 C#中我们是否已经达到了那个点。你问题的趋势是：我们能一直举出东西来并把它加到那里吗？但某种时候我猜这样做是不会满意的，这并不是对主流编程的批评，这只是因为它们是为不同的目的而设计的。

InfoQ：我们有 Lisp 和 Smalltalk 这样的语言，只有最少的语法；而我们也有 C++，C#等语言，有着很多的语法，那么你认为 Haskell 是处于中间什么位置，你就此有什么看法？语法是增加了语言的复杂性还是为开发者减少了复杂性呢？

Simon：这是使用接口问题的又一例子，不是吗？我认为语法有两种味道：有一些是表面的，在 Haskell 时我可以如此定义函数，*f* x=let y=x+1 in y * y ，但我同样可以这样定义同一个函数 *f* x= y * y where y=x+1。这只是语法本身而已，我只是用"where"替代了"let"；只有细小的差别："let"表达式在任何地方都会发生，而"where" 只在所附加的定义中产生，然而，这也只是表面上的差别。

当我们一开始设计 Haskell 的时候对于“所有的事情我们都应当只有一种方式来做”产生了很大的争论，实际上最终我们实现的语言提供了多种语法来达到同一件事情，因为当你写程序时，有时候"let"在某些情况表达了你想表达的东西，而有时候用"where"表达恰到好处。我认为这只是表面上的复杂性，因为对于程序员来说理解这两者从智力上来说都不复杂。你展现了，这就对了，你不必要求更多。

有些问题要复杂得多，比如列表内涵式。在 Haskell 当中你像这样就完全将列表表达出来了[y*y |y?yy, y > 3]。这种类型的内涵式已经在像 Python 和 Ruby 这样的语言中逐渐显现。因此对于其意义还有更多需要解释的。实际上，我最近将其概括成为更 SQL 化了，因此你同样可以以列表内涵式的方式来表达 group by 和 sort by 语句。但它们仍有些流于表面了，更深入的东西在此。一个更深层语法问题的例子可能是类型类。Haskell 的类型类跟面向对象的类极其不同，然而你仍可以说它们不过是语法的类声明，实例声明，但这实际上这只是你需要理解的一系列的概念集合当中最浅显的一部分罢了。因此我试图将表面的语法与深层次要素加以区别。

我对表面的语法比较随意，而对于深层次要素，Haskell 有着类型类这一强大的概念使得许多编程都变得容易了。但无庸质疑的是，如果你要用某种特定语言来编写程序，你必须要习惯于它，这对于面向对象的类而言也是一样的。

Smalltalk 和 Lisp 都有属性这一概念，但它们不仅语法更为得体，还有幸节省了许多概念，因为 Smalltalk 仅仅是对象和消息，只需 S 表达式和 lambda 来捕获自由的变量。对于深层次要素，它们更是尤为经济，而它们选择的要素使其非常强大。但这里也有一个交换条件，因为 Lisp 的 lambda 演算非常强大，但要表达任何特别的事物。你可能会需要很多的圆括号。这也被称作："很多烦人的过多的括号"。

所以你会说："那让我们来大大简化一下我们想做的事吧"，这就是表面复杂性的角色。我实际上比较支持一个中等的富语法集。Lisp 所带给你的，而当你拥有了富有的语法之后却会变得困难的，是 Lisp 中程序本身就是一个可以被 Lisp 程序操作的数据结构,这就使得元编程非常方便。但 Haskell 有着相对丰富的语法，用微软的话说：这一类的代码 "非常复杂并难于理解"，所以当人们提到 "丰富" 的时候，请总是保持怀疑，所以丰富意味着由许多构件，但我以为最大的不同无非是表面层次的。然而，如果你想要编写操作 Haskell 程序的程序，那你需要面对许多语法变种。

这就是交换条件-这会显得很为难，因为元编程对于编写特定类型的程序非常方便，但这意味着所有的 Lisp 开发者都不得不使用这一小语言。我想这就是所谓的交换条件吧。但以 C++ 为例，它极其复杂；它有很多深层次的复杂性。我不认为我全面的理解了 C++。Haskell 也有着一定的（复杂性），以示公平，它还在不停的增长。我建议 "专注于深层次复杂性而不是表面的语法"。

函数式语言的想法让我有点害怕。基本上，如果你思考对象，当你对一个给定的对象拥有它的方法和属性时，它可以映射到现实的世界中，如你可以说 "一个人.用手.挠痒（另一只手）"。当你对一个给定的对象拥有它的方法和属性时。而对于接受函数式编程和以函数的方式编程，看起来似乎是需要对范型的转换，以及对于整个思考过程的转换。你认为面向对象语言成为主流的原因之一是什么呢？有没有什么方法可以填补函数式和面向对象之间的间隙？

我想，毫无疑问，从整体上而言命令式编程语言，特别是面向对象的语言，对于操作的模型有着非常好的映射。你可以想像这些位置是如何变化以及程序是如何一步一步的执行的。这实际上是近距离的观察在底层真正的机器上发生了什么。而我们对于此有非常良好的可操作性理解。很多使用电子表格的人们并不认为自己是程序员；他们认为这是建模。而你如果不觉得这是一个步骤的序列的话，你也不并向他们解释了。我在之前的讲话试图表达这一观点。

试图将计算当作处理值的函数来对待，以函数式编程的方式来表达这一思想，其实是非常自然的。如果你一个熟练的面向对象程序员试图在半路出家学习像 Haskell 和 Lisp，其思维过程确实截然不同，很多你所熟悉的习惯可能不会很好的工作。我花了一个上午去学习单板滑雪的课程，而我本人是一个中级的双板滑雪者；我总是不断的跌倒，因为我的本能总是完全错误的。我做出的每个本能的动作都会让我跌倒。我放弃了，还是回到了双板滑雪。

这里就好像是这样的一种感觉，初看起来不光是笨拙并且还让人精疲力尽。你这样做又是为什么呢？你学习这门语言，而我们不知道如果你早一些学习它的话会发生什么。现在越来越多的毕业生走上社会时或多或少的接触过一点函数式编程，我认为这是一件很好的事情。但我要说的是：我希望看到这样一种经验的重复，尝试这一试验的人们会说："现在我将把我的自由理念放到一边，投入这个世界，好好尝试一下并坚持下去，得到一种惊喜"。而人们常常说的是："现在我要回到我的 Java 程序去了并尝试用一种不同的方式来编写它们。"

我换了一种方式来思考它们。所以我不得不让思想重新组织，但有一些重新的组织却让我受益终生。为什么呢？我想这是因为所能发生的是你可以在 Java 里做函数式编程了；你尝试更多的运用不可变的对象；给予你一个 String，你不是改变或胡搞一通，而是把它拷贝给另一个，这样你仍可以拥有原始的那个，而且你的程序交互更少了，你发现你越来越多的进行这种尝试。你会想："这是否已足够了？" 实际上，现代来说这可能意味的是：良好的垃圾回收器，与良好的存储分配器。

我作为一个编译器开发者，所做的一切目的都是让这些分配和事物更快的运转。这些函数式编程的习惯可以回到命令式的世界里，我在我的讲话里试图说明其理由。我认为那些对于计算内部而言是附带发生的，而对于其外部行为并不重要的副作用，如果能在编码时尽量避免它们，就像它们是偶然的，去掉那些你可以不必需要的副作用，这将改变你的整个编程模式。

但你仍可以在命令式语言中做到这些，这样你最终得到的程序就拥有了我试图描述的一些好处。底线是，这是一个思想重新组织的经历，并且无可逃避，这不像是从 Java 转换到 C#，或者从 Java 到 C++。

InfoQ：能告诉我们接下来的几个月会有些什么样的进展吗？有没有什么特别有趣的？从远处看去你们的 Haskell 测试仪上将会进行什么尝试？

Simon：实际上我知道的也不比你多。微软准备将 F#产品化，而我虽然为微软工作，我得到的微软的消息也大多来自媒体。至于我自己本人的工作，我所为之兴奋的，其中一个重要的东西就是并行，特别的是，事务性内存十分成功。你可以下载 GHC，你就在你的多核机器上拥有了工作得很好的事务性内存系统，但这还不够，因为事务性内存意味着你可以繁衍线程。

这就像是 fork 进程，它们通过共享内存来通信，因此在你的控制之内。这是有所限制的并行，你可以利用到 100 个进程。很难找到这么多进程来做事，我想你需要这么多进程忙碌工作的地方应该是数据并行。数据并行-一个非常古老的点子-在一个大的集合，比如数组中，许多元素在同一时间做同样的事情。

将你的数组分成块，每个进程一块，每一进程负责它的块并在它们完成的时候达成一致。这就是数据并行，在实践中非常成功：MPI，Google 的 Map Reduce，高性能计算 Fortran，这一类的东西。但这是平的，我的意思是一个数组的每一元素都做同样的事情，但你所完成的工作是顺序的。并不是每个算法都能够映射到这一范式。更加灵活的方式是：在数据并行中数组的每个元素都做同样的事情，但你所完成的工作本身应当是一个数据并行的算法。它跨越其它的向量以数据并行的方式工作，且你每个元素所做的工作其本身也可能是数据并行的。

现在你可以看到，这一并行树在顶点有一些分支，以及一些小型的分支，然后变得像灌木一样复杂的结构，对于程序员来说好多了但对于实现来说却糟透了。这很难实现，因为在这一茂密的树上的叶子可能是对于一个浮点数的小型的浮点指令。你不会想要为此繁衍一个线程，你也不会想为这些树叶繁衍上百万的线程。你将会尝试找到一些方式来恢复扁平数据并行的粒度，就算数组有上百万元素，如果你有 10 个进程，每个处理器一千万元素：顺序循环也是十分有效的机制。

Guy Blelloch，90 年代在卡耐基梅隆大学与他的同事 Sabah 一起展示了如何使用一个嵌套的数据并行算法，正如我之前给你描述的更加灵活的那种，来在编译时转换成一个新的算法，能够工作于一个扁平的数据数组。他们将这一嵌套的数据并行转换成扁平的数据并行，你知道如何有效的实现它。这非常棒因为你得到了一个更加灵活的编程机制，并且仍然有着高效的实现。Guy 的工作在 90 年代中间转化了成了一种叫做 Nastel 语言，这是一个原型实现；其本质上是一个解释器。

它实际上对于很少的数据类型是非强类型的，而且它除了该嵌套数据并行几乎不做其它的事情。而我此刻之所以兴奋，是拿来 Nastel 的思想并通过将其应用于 Haskell 而它们带入到 21 世纪。所以我与在澳大利亚悉尼的一些伙计一道将它加入了 GHC-一个 Haskell 编译器。主导思想是有一个单一的语言能够支持分支类的表达型并行事务性内存以及这一嵌套数据并行的东西，还有半隐式的繁衍型并行，都包括在一个语言的框架里。

也许你在你的应用中不同的部分运用不同类型的并行，而你的应用某些部分让你觉得厌恶，因此你也会需要连接。我认为这一嵌套的数据并行思想是利用那些可爱的多核最有前途的方式。有趣的是，这一转换是一个非常重大的转换，它转换了所有的数据：数据表示改变了，

代码改变了，所有的都改变了。这是对你程序里的源文本一种系统性的改变，而你能有机会这样的唯一可能就是因为程序是纯粹的。

如果它是命令式的话我们不会有机会。它就像是一个 SQL 查询优化器，对程序影响很大。而这种影响甚至可能让程序无法工作，如果还想要保持直接的副作用的话。如果我们真的能让嵌套数据并行工作起来，我想的话可能这就是杀手应用。这里有一定的风险，我不知道我能这样做，不过我们可以让它工作于弯曲的时钟，对于特定的有趣问题加速并且比 Fortran 还快，因为我们能编写的程序你无法用 Fortran 做到，而且可以工作于上百的处理器组上-这简直酷毙了！

InfoQ：最近我浏览了 Meijer 和其它人的文章。它谈到了编程，“香蕉”，“透镜”和“铁丝线”。我不知道你是否浏览了这篇文章，它似乎是对不同类型的同种递归的抽象，至少从我的理解来看。你认为这种类型的抽象，如果我们确实对递归进行抽象的话，是否会对优化和简化复杂的代码带来很多好处？

Simon：回到 F#，它吸取了很多函数式编程语言的概念，但仍是可变化的状态。你不认为小小的一点可变性会破坏整个系统吗，影响到整个编程语言的概念？

是的。.NET 到 F#是一种折衷。F#是.NET 的高阶温床，而就 F#的设计而言它是一门.NET 语言。在一门.NET 语言中你可以调用任意的.NET 过程，而任意的.NET 过程都可以做任何的事情。很难说这是一个纯粹的函数。相反 F#引诱你进入一个纯粹的世界，通过提供一个丰富的良好设计的语言用以编写纯粹的函数式程序。实际上这回到了语法的问题，因为 F#被翻译成 MSIL-中间语言-与 C#一样，所以你能用 F#做的你都可以用 C#做。你可以用 C#来写那些函数式程序，但却要麻烦得多。

所以 F#带来了这种便利；它使得编写函数的部分变得容易，命令式的部分也有这种可能，但你或许会正确的指出，这并不能得到保证。在有一些场合，比如并行的 LinQ 查询，必须要得以保证，而你能做的唯一办法就是通过约定。但也不至于这么差。我们对此已经习惯了-想想对可变数据的加锁保护。通常，锁和它所保护的数据之间的联系，大部分不是显式的存在于语言或者程序中，而是你的脑海中。

锁和数据的联系有时候是有一些私密的，但通常你访问什么数据之前需要获得什么样的锁在程序中是暗含的。这里并没有任何的保证，我们只是建立起约定并试图让它们工作。这并不像有一个岩石一般的保证那样让人满意。我们需要更强的保证，然而，我们也不能说：“因为你得不到保证，所以这是没用的”。这只是一个折衷，并且可能是一个很好的折衷。

InfoQ：在 LinQ 当中我们传递任何东西都是用的引用，我们有闭包，以及引用对象内的闭包。

这违背了使用 lambda 演算和列表内涵式的模型吗？

Simon：这是可以的。共享也没什么问题；引用也挺好的。如果你和我都指向同一个 String，事实上这个 String 表示“Hello”，我们没必要保有两份 “Hello”，我们可以共享同一份拷贝，只要我们不改变它。共享并不是问题，它带来的影响才是。如果你共享的引用是指向不可变的值，那非常好。如果你共享的引用是指向可变的值而你更改了它们，那所有的事情就可能变糟。

InfoQ：这些共享不是会对性能和优化带来好处吗？

Simon：当然，纯粹性有着一些良好的性能效果，因为你可以有力的共享更多的东西-我们在终端比较过 2 个 NSD C++集合实现，当做一个并集时不得不拷贝输入集合，以防它们因此被更改，而 F#的实现就不会。但它同样也会带来代价。假设我拥有两个 String，并且想返回“Hello”和“World”给你，我想返回连接的 String “Hello world”。我不能改变输入 String 的任中一个，因为它们是不可变的，所以我必须创建一个新的 String-“Hello world” String-通过拷贝另外两个，或者至少拷贝其中一个并共享尾部。

总是会有一些拷贝参与进来，有些时候你会获得更好的共享性能，而有时候会变得差一些。我假设这最终会成为你的思维的一部分，你的思维重组的一部分。你会尝试区分你将会更改的数据和你不会去更改的数据，这样就不必严格的拷贝事物。你不能说这就是绝对的好或是绝对的坏，从性能的角度来看；它是变化的。

观看完整视频：

http://www.infoq.com/cn/interviews/simon-peyton-jones-about-mainstream-programming-languages-cn

相关内容：

- 新的Scala Actor类型系统——谁说竞争安全与性能不可兼得
- 语言约束和责任感，我们应该信赖谁？
- 使用LESS或Sass重构CSS代码
- VB和C#的自动实现属性
- F#中不同类型的NULL值

[热点新闻]

开发减速，是为了赢利提速

作者 Vikas Hazrati 译者 郑柯

> 人们一般都认为：如果团队中每个人都能发挥最大能力，那么这个团队一定是最有产出的团队。与之相反，Steve Bockman 认为这个假定不一定永远成立。在有些情况下，可能有必要 放慢团队速度，少干点儿活，而不是全速前进，最终目的是为了迅速提高产出和利润。

Steve 提到他在两个团队中做过的练习，步骤如下：

> 操作一：从库房里取出一些原材料（一叠纸）。
>
> 操作二：把纸向内折叠，然后再展开。
>
> 操作三：把纸张的上面两个角向内折叠。
>
> 操作四：把纸张两边向内折叠，只折一半，折成类似于翅膀的形状，再放下来。

一个团队用尽全力工作，而另一个团队被要求放慢速度，以配合最慢的操作。

两个团队都在 5 分钟内产生出同样数量的纸飞机，然而查看由于在制品库存造成的损失，全力工作的团队的损失要远高于配合最慢操作的团队造成的损失。第二个团队没有产生堆积如山的在制品，因此减少了消 耗的物料和劳力。

Steve 进一步引用了商业小说《目标》， 其中指出可采取下列步骤改进产出：

> 步骤 1：识别系统的瓶颈。
>
> 步骤 2：想办法如何让瓶颈发挥最大潜力（比如：不让瓶颈处于空置状态）.
>
> 步骤 3：协调系统其他部分，以配合步骤 2 中的决策（比如减缓瓶颈上游步骤的操作速度）。
>
> 步骤 4：提升系统的瓶颈（比如让缓慢的操作提速）。

> 步骤 5：如果在之前的步骤中，一个瓶颈被打破（比如某个瓶颈不再是瓶颈），那就回到步骤 1。

因此，推荐做法是：在试图提升瓶颈速度之前，先放慢所有非瓶颈操作的速度。

将上面的推荐做法与自己在试验中得到的体会梳理过一遍之后，Steve 将软件开发过程做了一个类比，这个类比可以总结为知识的创建与分享。他建议采用下列咒语来提升利润：

> 知识是软件开发的库存。
>
> 人们会以自己的速率来消耗知识。
>
> 如果知识的创建速度快于消耗速度，那就会产生过量库存。
>
> 过量库存会降低利润。

根据上面的讨论，我们可以得出结论：如果系统中存在瓶颈，而且以相对较低的速度水平运作，要是此时仍拼尽全力，反而对系统产出利润不利。关键在于保持与瓶颈相同的处理能力，并先让瓶颈提速，让整个 系统向最高处理能力接近。这样就可以降低在制品数量，从而保证系统的健康，还能产出利润。

原文链接：http://www.infoq.com/cn/news/2009/07/slow-down-increase-profits

相关内容：

- 我们需要创建信息系统分级
- 契约的版本管理、兼容性和可组合性
- 为什么我们要放弃Subversion
- 豆瓣网技术架构变迁
- Ruby on Rails项目的救赎

[热点新闻]

Sun股东同意Oracle的收购

作者 Charles Humble 译者 张龙

Sun 的一则新闻表明持有 Sun 62%普通股的股东已经投票通过了 Oracle 对公司的收购，价格为每股$9.50。最新的财务数字表明这将使 Sun 的负债净额和留存现金达到 56 亿美金的金额。

此次交易还需要美国与欧洲监管局的批准，上月美国司法部扩大了反垄断审查的范围，特别是 Java。Oracle 说司法部的调查“走进了死胡同”，主要是 Sun Java 技术的许可方式。Oracle 的律师 Dan Wall 说“我真的希望这个调查尽早结束，不要延误今夏的这笔交易”。

今年四月 Oracle 宣布将要收购 Sun，其每股$9.50 的出价击败了竞争对手 IBM。泄漏出来的 3 月份的谈判情况表明，这个出价几乎是 Sun 此前股价的 2 倍了。与此同时，Sun 在财政上还在不断挣扎，本周初宣布与去年相比，Sun 第二季度的销售额同比下降 1/3 还要多。

原文链接：http://www.infoq.com/cn/news/2009/07/sunoracle

相关内容：

- Atlassian收购GreenHopper，在JIRA中加入敏捷项目管理特性
- Oracle如何处理Sun的开源资产
- SpringSource收购Hyperic
- SpringSource宣布收购G2One，正式介入Groovy和Grails领域
- 王雷谈开源以及新兴市场计划

[热点新闻]

综述：Scala是Java未来的后继者

作者 Dionysios G. Synodinos 译者 杨晨

作为 Java 未来的后继者之一，Scala 最近受到了大量关注。Groovy 的创始人 James Strachan 和 James Gosling、Charles Nutter 一样，是 Scala 的拥趸，后两人分别是 Java 的创造者和 JRuby 的核心开发者。

James 首先解释了他不喜欢的 Java 特性：

> Java是一个令人惊讶的复杂语言（规范有600页，但是有人的确对Java的特性心领神会了吗？），表现在它的自动装箱 （在这里隐藏了可爱的NPE），原生类型，讨厌的数组（它们不是collection，而且由于缺少多态性，对于通用数据结构和bean特性需要很冗繁的 语法，并且仍然没有闭包（即使在JDK7中），导致了大量令人烦躁的try/catch/finally的语句，除非你使用包含了全新自定义API的架构，但是这样会导致语言更加复杂。Java甚至有类型引用，不过我们还是不要使用它来存储任何typing/reading。
>
> 尤其是没有Java7（即使在Snorcle之后它显得更加有意义 - 我想知道javac是否会被jdkc取代？我猜javac已经达到其巅峰；而且闭包看起来不会带来任何的简化或者进步），这个问题表现得更加严重。

他看起来已经被 Scala 深深影响了，尤其是当他说到如果那个时候有可用的 Scala，那么他不会一开始就去发明 Groovy：

> 老实说，如果在2003年就有人给我介绍了Martin Odersky Lex Spoon和Bill Venners的Programming in Scala，那么我很可能不会创造Groovy。

当然，也有一些 Scala 的特性他不是那么热衷：

> 对于任何一门语言来说，都有你喜欢和不喜欢的东西。Scala给我的早期印象的确看起来它在尝试使用一点更多的符号，但是你不需要全部使用。如果你喜欢，你可以仍然使用Java风格的OO。但是我想未来为“特殊物体”使用符号来避免和标识符冲突。
>
> 我不是嵌入import语句的狂热粉丝，使用_root_.java.util.List来区分从

> 相对import 中得到的“全局”import。我更喜欢子前缀，例如，如果你从com.acme.cheese.model.Foo导入，那么导入 model.impl.FooImpl的时候，我喜欢使用一个相对前缀，也就是说，导入_.impl.FooImpl将会使事情简化，而且和Scala在 简化和删除冗余代码（导入java.util._是多种类型的）保持一致。

任何时候和 Java 相比，James 都认为 Scala 好太多了：

> Java的不足可以比作大量的毛疣，那么同样在Scala中，这些地方正是表现了Scala的美、简化和强大。

Adam Bien 在他的博客中指出，即使是 Java 之父 James Gosling，看起来也是对 Scala 喜爱有加：

> 在一个社区（java.net booth）举办的和James Gosling对话会议上，一个与会者问了一个非常有意思的问题：“除了Java，现在你会把哪种语言运行于JVM之上？”。答案是惊人地快速简洁：Scala。

Charles Nutter ,JRuby 核心开发者 ,他也认为和 Groovy 和 JRuby 相比 ,Scala 更可能替代 Java：

> 我必须说Scala看起来是是现在Java王座的继承人。其他在JVM的语言看起来不可能有Scala那样的能力来取代 Java，Scala背后的推动力是无可置疑的。Scala还不是一个动态语言，但是它有许多流行动态语言的特性，例如它的灵活富类型系统，稀疏和简洁的 语法，函数式语言和面向对象范式的完美结合。Scala的缺点：“太复杂”或者“太丰富”，但这些可以通过编码规范很好避免，从而构建更健壮的编辑器和工 具，以及指导多语言开发者明白如何更好地使用Scala。Scala是JVM上静态语言的重生，它也像JRuby那样延伸平台的性能，这些都是Java做 不到的。

Scala 现在已经是今年 JavaOne 的一个主题，有一些相关的议程，而且在大会的最后一天甚至会有一个开放的讨论。

你怎么想呢？Scala 是不是在将来最合适取代 Java 的语言？或者，Java 是最后一门巨型语言（LBL）？

原文链接：http://www.infoq.com/cn/news/2009/07/scala-replace-java

相关内容：

- 剖析短迭代
- 用GPL工具实现COBOL到Java自动移植
- 敏捷的文档

[热点新闻]

“服务重用”是否被过度使用？

作者 Mark Little 译者 马国耀

服务重用经常作为 SOA 的一个重要特征被提及。很多人甚至用它来衡量 SOA 是否成功。如 Eric Roch 曾说：

> 毫无疑问，在衡量SOA成功时，最明显的尺度当然是服务重用。开发团队间为实现重用的最大化而展开的友好竞争是宣传和鼓励服务开发和重用的最好方法。

或者 IBM 的立场：

> 重用在SOA中占很大比重。它是SOA简洁性的一部分，也是将服务串接起来解决端到端的业务问题或流程的一部分。

正如上面所说，在衡量 SOA 成功与否时，服务重用的总数经常被作为重要的衡量指标。

> 服务重用即是SOA的特征之一，也是SOA带来的好处之一。

然而，事情并非那么简单，早在 SOA 刚刚兴起时就有人认为服务的重用无足轻重，或者，至少不能被做为 SOA 背后的主要驱动力。如 Dave Chappel 在 2006 年说道：

> 为重用而创建服务，就必须要预见未来……服务创建者怎么知道未来的应用需要什么？“守株待兔（`if-you-build-it-they-will-come`）”式的方法很难实现真正的重用。

现在 Burton 的 Richard Watson 也加入了讨论，他认为“人们对于重用的预期一直过高”；开发人员、用户和决策者不应该把眼光定格在服务重用上。他说：

> 服务有可能永远都不会被重用，但它仍然以其他的方式在创造价值：通过适配的方式、维护成本低 、减少冗余、通过政策的坚实执行来提高安全性和合规性等，这里仅列举少数几个其他方面的价值。过分强调重用让我们忽视了服务的其他价值。

他提议将重用的价值问题分解成一个方程，随着时间的推移来计算重用的数量及节省的成本，当然也要将部署以及应用程序的具体要求等因素考虑在内。依 Richard 看来，我们真正需要重视的是服务的价值，而重用仅仅是其 中一小部分而已。他还说：

> [……]服务的价值偶尔可以体现出来，比如当汇报制度的改变要求使用一组不同的规则并且需要修改的是某个孤立的点，而不是全盘的修改。它将我们带回到服务“使用”的价值，而不是服务“重用”的价值。

对象重用经常被曲解成是面向对象的主要好处，但实际往往相悖于理论。最终，人们逐渐不再用它，而关注 OO 带来的其他能够摸得到的好处。服务重用是否会重蹈覆辙呢？

原文链接：http://www.infoq.com/cn/news/2009/07/service-reuse

相关内容：

- 使用扩展方法对调用进行验证
- InfoQ中文站翻译之作《实现模式》中文版面市
- 采访Clone Detective项目创始人Immo Landwerth
- 是否该重新衡量SOA产品了？
- 测试驱动的代码重用

[热点新闻]

使用LESS或Sass重构CSS代码

作者 Werner Schuster 译者 杨晨

在 Web 开发中，CSS 的使用是非常普遍的，但滥用的情况也是屡见不鲜。LESS 和 Sass 都是用 Ruby 实现的工具 ,可以帮助开发者写出复用性更优的 CSS 文件。它们的方法基本类似：将类似 CSS 但是更为强大的输入语言，最终转换为 CSS 代码。

两种语言给 CSS 添加的特性都是相似的，具体参见 LESS 和 Sass 的文档。下面是一个简略的概述：

- **变量**： LESS 中的@name 和 Sass 中的!name 都是变量。我们可以给变量赋值，然后在文件中使用它们。
- **内嵌**：这个功能将另外一个急需的特性加入 CSS：将选择器嵌入到其他等级，而不是不得不取消在一些高级选择器定义中嵌套。LESS 和 Sass 翻译器将这个简洁的特性扩展到了 CSS。
- **混合类型**：允许开发者抽象出性质的共同点，然后命名并且加入到选择器中。熟悉 Ruby 混合类型的开发者会了解混合类型在 CSS 中的应用。Sass 也允许将混合类型作为参数，使得混合类型的应用更加灵活。
- **操作**：LESS 和 Sass 都支持简单的算术操作，例如加法。将这个特性和变量结合起来，会使得 CSS 变得更加灵活。这两个工具需要保证操作的正确性（例如字体大小）。

Sass 是由 Haml 的团队开发的。它采用了 Haml 的思想，使用缩进而不是括号这样的分隔符来定义代码块或者内嵌级别。

Sass 的解析器和翻译器将 Sass 语言翻译成 CSS，并且用变量值替换文件中变量的引用以及混合类型等等。

LESS 是受 Sass 启发而开发的工具，它列出了如下开发的理由：

> 为什么要开发一个Sass的替代品呢？原因很简单：首先是语法。Sass的一个关键特性是缩进式的语法，这种语法可以产生柱式外观的代码。但是你需要花费时间学习一门新的语法以及重新构建你现在的样式表。
>
> LESS给CSS带来了很多特性，使得LESS能够和CSS无缝地紧密结合在一起。因此，你可以平滑地由CSS迁移到LESS，如果你只是对使用变量或者操作感兴趣的话，你不需要学习一整门全新的语言。

LESS 的解析器是使用 TreeTop 编写的 ,TreeTop 是一个 Ruby 编写的 PEG 解析器的生成器(LESS TreeTop 语法)。

LESS 和 Sass 工具（编译器和 API）能够作为 Gems 安装，使用命令行工具进行编译，但是也可以在 Ruby 代码中使用。

Sass 看起来在提供的特性上占有优势，但是 LESS 能够让开发者平滑地从现存 CSS 文件过渡到 LESS，而不需要像 Sass 那样需要将 CSS 文件转换成 Sass 格式。Sass 的维护者 Nathan Weizenbaum 在一篇对比 LESS 和 Sass 的博文中提到，未来 Sass 将会提供括号，而不是像 CSS 或者 LESS 那样的缩进。

原文链接：http://www.infoq.com/cn/news/2009/07/dry-css-less-yass

相关内容：

- MountainWest RubyConf 2009 视频
- JetBrains元编程系统支持面向语言编程和DSL
- 外部DSL：成功与失败的因素
- 微软的模型驱动开发战略
- 创建内部DSLs——Groovy比Java更好吗？

[热点新闻]

Ruby on Rails项目的救赎

作者 Robert Bazinet 译者 杨晨

Ruby on Rails 发布已有五年，在此期间，开发者编写了大量的应用。其中很多应用程序都是开发者们在学习 Ruby 和 Rails 的过程中写就的 ,代码质量堪虞 ,却已经运行于生产环境之中。

这些 Web 应用程序的规模数年来不断猛增，由于代码行数不断增加，它们已经变得臃肿，难以维护。这是很多开发者们不得不面对的情况。而且，他们几乎不知道从何下手。

一本新近出版的书能够帮助开发者解决这类问题，这本书叫做 Rails 拯救手册，由 Mike Gunderloy 编写。这本书也同时以 PDF 的形式发布，购买的用户可以终身收到最新版。InfoQ 获得了一个和 Mike 直接对话的机会，试图了解到这本书的核心思想。

Mike 介绍他的新书时说：

> Rails拯救手册是我心血的结晶，当你面对一个乱糟糟的Rails应用程序，并且希望把它进行重构（或是仅仅使它能够工作），它能够一步一步指导你如何 去做。这本书介绍了一些简单Rails应用程序的共同问题点，并且在战略和战术级别上对如何重构，以及回到高效开发上给出了建议。

这本书面向的读者不是 Ruby on Rails 的新手。Mike 解释了本书的读者群：

> 本书面向的是正在和陈旧的Rails应用程序搏斗，并且需要帮助的开发者。在大部分情况下；一个项目会有第二个开发 者，他会在第一个开发者不能发布可运行代码的时候被加入到这个项目中。有些时候，甚至不会有第二个开发者：一旦你意识到你可能会有问题（或者，很多问 题），没有人能够阻止你重构你的项目。

修正代码问题的关键一环便是知道问题在哪，有些非常明显，但是有些就不是。关于这个问题：

> 表面上说，Rails项目中发现问题的数量和客户需求无关。我看到过这些问题是以两种形式。开发者不能够按时发布能 够工作的代码，或者发布的新特性破坏了程序的其他部分。当然，后者是最常见的问题之一：缺乏足够的高质量测试覆盖。当你获得源代码的时候，经常可以在这些 地方发现问题：由于缺乏对数据库层的考

> 虑，不可思议的臃肿的控制器以及充满了SQL语句的视图等等导致了低下的性能。

一旦这些问题被找出，那么需要一个结构化的方法来修正它们。Mike 给出了一个开放的结构化的方法：

> 首先，你需要了解客户的需求，明白哪些修改是必须的。客户需要了解到当开发者试图使程序稳定的时期内，应用程序可能会很少甚至没有新特性。这个过程的一个 关键点在于设置良好的沟通渠道，及时通知客户—经常你需要处理一些不可思议的遗留问题，但是不会满足客户的需求。透明是这个过程中的口号。
>
> 在你确认你已经获得了能够继续进行的足够多的支持之后，下一步就是对现存代码做一些巨大的改动（假设这些都不是你写的）。检查Rails的版本，将一份拷 贝放到你的桌面（如果可以的话），检查一下路由器以及模型、控制器和视图。仔细查找问题点。通常客户将会知道性能和功能问题出在哪儿；仔细听他们说。
>
> 当你已经知道代码中的问题之后，那么着手重构它们。通常重构之后需要进行测试。你需要确认问题已经走上正轨，而且开发过程处于掌控之中。

我们然后讨论了通过选择一些容易完成的目标来达到快速制胜：

> 书中有一些秘诀，能够轻松为客户带来益处（也能够帮助他们成为反馈式的开发者）。例如，花费在理解数据模型和寻找模式的时间能够帮助你寻找到如何得到本质的性能提升。

也有一些地方需要注意，他介绍了一些入手点：

> 此外如果需要关注数据库层（尤其是在开发者的第一个或者第二个Rails项目），通过运行Google Page Speed类似的工具能够知道那些地方存在性能瓶颈（性能问题是客户最关心的问题之一）。在一个公共站点，确保有一些异常通知来告诉你哪些东西失败了，是 另外一个能够知道代码的哪些部分需要特别关注的方法。

当被问道他在 Ruby 社区的背景以及他如何意识到这些常见问题的类型的时候，Mike 指出他多年的咨询经验以及学习帮助了他：

> 在过去两年中，我已经帮助了大量的项目转到正轨来，但是为了客户的信心，我不能透露他们的名字。我也曾经在做Action Rails的 时候，做过一些代码检查。这些项目有些只是一些简单站点，基于10个左右模型，但是这些模型缺少了一些基本的Rails功能，例如模型关联（因此编写了太 多自定义代码），有些却是大型应用程序，已经被部署用作主要的功能实现，但是却被Bug缠身。我花费了很多时间在客户的项目以及转包合同，而不是在产品或 者临时工作中，这些并没有给我太多机会来审视它们为什么失败。

更多信息可以在 Rails 拯救手册的官方站点上找到。

Mike Gunderloy 在多种语言和平台上有着二十年的开发经验。从 2006 年开始，他全职转向 Rails 工作，对于和各种规模的分布式敏捷团队的协作，以及 和各类项目参与人员（从开发

者到项目经理）的合作有着丰富的经验。他对 core Rails 也是贡献良多，还是 Rails 文档团队的成员。此外，在他在 Lark 集团的本职咨询工作中，是一位在高端 Rails 咨询领域的合伙人，同时也是 RailsBridge 的创始人之一。

原文链接：http://www.infoq.com/cn/news/2009/07/rails-rescue-handbook

相关内容：

- Kent Beck建议超短项目跳过测试
- 通过测试驱动开发和结对编程提高生产水平
- J.B. Rainsberger：“集成测试是个阴谋”
- 质量意味着什么？
- 扔掉bug跟踪系统？

[热点新闻]

各方未就HTML 5 Video Codec达成一致

作者 Abel Avram 译者 张龙

近日 HTML 5 规范的编辑 Ian Hickson 分别从 video 与 audio 标签的草案标准中移除了 codec，这是因为对于那些在网上发布视频和音频的大多数公司来说很难就这个议题达成和解。

目前主要使用两个标准：H.264 与 Ogg Theora。H.264 或 MPEG-4 是私有的视频压缩标准，如果用于商用则需要购买 license，它特别适合于大容量的视频；而 Ogg Theora 则是个开源免费的标准，但其质量却不敢恭维，同时支持它的大厂商也少的可怜，Hickson 说到：

> Apple拒绝在QuickTime中（Safari使用的也是QuickTime）实现Ogg Theora，因为其缺少硬件支持及不确定的专利问题。

Opera 与 Mozilla 对 H.264 提出了反对意见：

> Opera拒绝实现H.264，因为其相关的专利协议花费太大。
>
> Mozilla也拒绝实现H.264，因为其无法获得可以涵盖下游分发者的协议。

Google 却双管齐下：

> Google已经在Chrome中实现了H.264与Ogg Theora，但却无法向Chromium的第三方分发者提供H.264 codec license，同时也指出Ogg Theora的quality-per-bit并不适合YouTube所处理的视频容量大小。

微软甚至未就 HTML 5 规范的<video>标签表态。

Opera Software 的开发者 Philip Jägenstedt 表明了他们的立场：

> 我们认为专利协议导致H.264与开放的Web平台格格不入。就目前情况来说，我们暂时会支持Ogg Vorbis/Theora，从专利角度来看这是最好的选择了，而且其quality-per-bit也在不断改进（尤其得力于最近编码器的改进）。我们希望它能成为HTML5的基线，然而我们还是衷心希望Web社区能够再使一把劲以使其成为事实上的标准。

在不远的将来还看不出合理的解决方案。Hickson 有两个想法：

> Ogg Theora编码器在不断改进。现在已经出现了可用的硬件Ogg Theora解码器芯片了。长久以来Google一直在支持codec而没有获得起诉，这导致Apple也逐渐打消了对专利的顾虑 。这么做会让Theora成为Web上codec事实上的标准。
>
> H.264 专利（那些把持着专利的公司还不想免费开放它们）也在不断消亡，这样对H.264 的支持无需支付任何协议费用。这么做会让H.264 成为Web上codec事实上的标准。

Hickson 认为最后的赢家要满足如下条件：

- 无需费用就可以实现，而且可由任何人分发
- 拥有可用的解码器芯片
- 使用广泛以弥补额外的专利费用
- 拥有足够高的`quality-per-bit`以处理大容量的视频站点

综上所述，不同公司又一次在公共标准问题上不欢而散，每个人都想按自己的方式行事，最终的胜者又是谁呢？我们期待着梦想照进现实的那一天。

原文链接：http://www.infoq.com/cn/news/2009/07/HTML-5-Video-Codec

相关内容：

- Google发布新版本的Protocol Buffers
- 语义面向服务架构参考本体论
- 在Scrum中识别非功能性需求
- Web IDL：W3C DOM规范语言绑定有了新名称
- 关于是否需要开放Web基金的争论

[热点新闻]

微软向Linux Kernel贡献两万行代码

作者 Abel Avram 译者 张龙

微软在 GPLv2 协议下向 Linux Kernel 2.6.32 贡献了 3 个 Linux 设备驱动，两万行代码。

微软开源技术中心主管 Tom Hanrahan 解释了微软作出这个非同寻常举动背后的动因：

> 我们有必要理解虚拟化的一个关键点。如果操作系统作为虚拟机运行，那么它得清楚这件事，这样就不会将调用直接发给各种外围设备了。在微软的术语中，我们称其为启迪（enlightenment）。Windows Server 2008就被设计成这样，因此它清楚何时作为虚拟机运行，何时在物理硬件上运行。
>
> 为了让Linux能在Hyper-V上拥有同样的体验，我们必须将这种启迪赋予给它。要想实现这一点则需要运行Linux设备驱动。

以前这些设备驱动是可以下载并用在 Hyper-V 的第一版上的，然而 Linux 社区使微软相信（通过 Greg Kroah-Hartman，在这个议题上与微软进行交涉的第一人）增加 Linux 设备驱动的办法就是将其贡献给社区，这样任何商业或非商业发布者都能够随意使用、修改并分发它们了。

微软开源技术中心经理 Hank Janssen（他领导的团队为这些驱动编写代码）承诺他们不会半途而废：

> 我们将继续更新驱动代码以不断增强其协同性，同时也希望社区中的开发者觉得这些代码对他们是有用的，值得为其付出。

451 Group 的分析师 Jay Lyman 就微软的这个举动发表了自己的一些看法。他认为微软会保留这些代码的知识产权：

> 这些代码的版权属于微软，贡献者的荣誉归功于该工程的领导者Hank Janssen—微软开源技术中心的程序经理。

然而他的理解却是微软并不会声明任何专利权，因此也不会向使用代码的人索取任何费用：

> 我们不妨做最坏的打算，假设微软这么做是个阴谋：他现在为Linux贡献代码，然后去申请专利。但理论上是否存在这个可能完全取决于我们对GPLv2的理解。
>
> ...最终这是一个法律上的问题，或取决于律师的口才（显然这么说带有讽刺意

> 味）。与此同时，我们认为微软通过GPLv2来贡献代码包含了一个承诺：不会对代码的使用收取费用，也不会对代码申请任何专利。

Lyman 还解释了微软之所以这么做的原因所在：

> Red Hat与Novell发布的Linux已经支持enlighten模式了，这要归功于与微软的合作开发。微软向Kernel贡献代码的一个好处是减少了重复开发工作，同时也降低了支持多种不同Linux实现的代价。一旦Kernel接受了代码，微软将以这些代码为基础进行未来的虚拟化集成开发。
>
> 这也意味着Linux的社区发布版也可以使用这些代码，这为微软在主机市场上开创了更多的机会，因为Linux的社区发布版如Ubuntu、Debian与CentOS都是非常重要的。这也加强了这些社区操作系统挑战Red Hat与Novell的能力，而后者对于Windows来说是更加直接的挑战者。
>
> 别搞错了，微软之所以这么做是由其利益驱使的。他必须要满足使用多种操作系统和混合环境的企业用户的各种需求，微软已经从差异化其Hyper-V技术与虚拟化领导者VMware中受益无穷了。我们有理由相信相对于VMware来说，微软对Windows的虚拟化会对Linux提供更加友好的体验。

需要说明的是，Linux 无需借助于上面提到的设备驱动就能运行在 Hyper-V 上，但性能却很差。这是微软首次向 Linux Kernel 贡献代码，也是首次在 GPLv2 下发布代码。

原文链接：http://www.infoq.com/cn/news/2009/07/Microsoft-Contributes-to-Linux

相关内容：

- 23 个.NET开源项目
- 平台多样化：Gavin Grover的Groovy之路
- RIA平台：除了Flex、Silverlight，还有Laszlo
- Google凭借Chrome 2.0 和Wave推动Web平台
- Gizmox发布Visual WebGui 6.4 预览版

[热点新闻]

Google开发全新操作系统Google Chrome OS，直接挑战微软核心业务

作者 Abel Avram 译者 韩锴

Google 宣布正在开发一款名为 Google Chrome OS 的全新的操作系统。该系统基于 Linux 内核，拥有一个新的窗口系统，目前的目标设备是 Netbook。

随着在搜索领域凌驾于 Microsoft 之上，并且利用 Android 打入移动设备市场之后，Google 开始涉足 Microsoft 的业务核心：操作系统。Google 一直在向自己的远景靠拢：由云服务来提供所有数据和应用，用户只需要浏览器即可访问它们，这是一个彻底的在线世界。Google 已经有了 Chrome 浏览器，但是他们还需要自己的操作系统，因为他们担心 Microsoft 最终会利用占据统治地位的 OS 来牵制自己。

不过 Google 并没有从头开发一套成熟的 OS，而是选择了 Linux 内核，并在之上创建了一个新的最简单的窗口系统。它运行的程序主要（可能唯一）是 Chrome 浏览器。其他的应用程序全部在线，要使用浏览器才能访问。Google 这种做法的背后因素在于希望为 Chrome 提供一个不受到他人控制的支持环境，尤其不要重蹈上世纪九十年代浏览器之战的覆辙。当时 Netscape 希望利用自己的浏览器占领桌面，从而排斥 Windows。

Google 将在今年晚些时候开放 Chrome OS 的源代码，届时将邀请社区加入开发的行列。他们也在与一些制造商沟通，这些制造商会在 2010 年下半年生产基于 Chrome OS 的 Netbook。当然，Netbook 并不是终点。Google 还希望把 Chrome OS 带入笔记本和台式机中。Google 称虽然 Android 也可用于 Netbook，但是它与 Chrome OS 是互补的，Android 的首要目标仍然是移动设备。Ghrome OS 将会运行在 x86 和 ARM 处理器上。

开发者仍然可以使用他们选择的 Web 平台，像以往一样工作。他们的目标不是桌面，而是要创建在线应用，如同他们之前所做的。

Google 希望将新的 OS 打造为：

> 效率、简单和安全...我们正在打造快速、轻便的OS，让你从开机到接入网络只需几秒钟。用户界面对你的操作习惯影响很小，大部分用户体验都发生在Web上。如同我们在Google Chrome浏览器上所做的一样，我们会审视OS的基础部分，并且完全重新设计底层的安全架构。这样，用户就不必再受到病毒、恶意软件和无休止的安全升级的困扰。尽管享受它就是了。

Google 这次明确地把目标指向了 Microsoft：

> 人们希望可以更便捷地查看email，不愿意等待机器和浏览器启动，白白浪费时间。他们还希望自己的计算机永远像刚刚买回来时那样快。他们希望无论身处何处都能访问到数据，不必担心计算机丢失或者忘记备份数据带来的损失。更重要的是，他们不希望花好几个小时去应付硬件的变更，以及摆脱无休止的软件更新。

历史即将打开新的一页。未来是什么样子的我们无从预测。Google 无疑会引领一部分市场，但是 Microsoft 也不可能袖手旁观。我们应该静观他们之间的斗法。他们的第一次交锋应该是由一款名为 Gazelle 的“更加安全的浏览器”引发的。Google 的成功之路并不会一帆风顺。在线应用的成熟度和效率还远远不及桌面应用。同时还有很多网络问题亟待解决。桌面应用仍然拥有庞大的用户群，要想改变他们的习惯，不是一蹴而就的事情。但是不要忘记，我们身处的这个世界无时不在变化！

原文链接：http://www.infoq.com/cn/news/2009/07/Google-Chrome-OS

相关内容：

- 与苏哲谈开源以及基于Linux的软件开发
- 谷雪梅谈云计算
- Google图表及gchartrb初探
- 和Google互补的搜索引擎Wolfram|Alpha
- Google Wave加速HTML5 发展？

[热点新闻]

微软的浏览器操作系统：Gazelle

作者 Jonathan Allen 译者 王瑜珩

Google 并不是唯一试图创建基于浏览器的安全操作系统的厂商。今年二月，来自微软的研究人员就透漏了 Gazelle 的细节。Gazelle 被称为“使用多用户操作系统技术构造的安全浏览器。Gazelle 的内核可以隔离不同来源的网页代码，并管理所有系统资源”。

与 Chrome OS 构建在 Linux 上一样，Gazelle 并不是一个真正的操作系统。它是构建在操作系统级别的一个服务层，以便扩展浏览器的安全模型。Principal 由协议、域名以及端口三部分组成，在进程级别不同的组合之间互相隔离，它是整个系统的关键部分。

为了防止跨网站脚本攻击等问题，页面只能直接渲染同一来源（Principal）的内容，其他来源的内容将根据来源被放到单独的进程中进行渲染，不同来源 的内容通过“浏览器内核”进行通信。浏览器内核实际上是一个受限的操作系统进程，用 C#代码写成，它负责协调进程间的通信。浏览器内核与进程通过在命名管 道(named pipes)中发送 XML 消息进行通信。每个进程的渲染结果，将被浏览器内核组合在一起，成为最终的显示结果。

Gazelle 拥有针对不同浏览器问题的特性，如插件、混合 http/https，以及递归整合攻击（recursive mashup attack）。

> 插件：Gazelle在使用插件的内容上强制使用同源策略（same-origin policy），也就是说插件中的内容以它自己的来源运行（而不是放置它的页面的来源）。由于插件是浏览器漏洞增加的主要原因，因此在浏览器内核中对插件 强制执行策略非常重要。现有的插件需要改写（移植或重写）以调用浏览器内核来实现功能。Alexa前100的网站中有34个网站只使用了Flash，而没 有使用任何其它插件。这显示仅修改Flash就可以解决很大一部分插件的兼容性问题。
>
> 包含HTTP脚本和CSS的HTTPS：在Gazelle中，使用HTTPS来源的页面不能包含任何使用HTTP的脚本和CSS。其它类型的内容如图片和 插件，则会运行于页面来源的进程中。这在Gazelle中可以改变，但是为了防止HTTPS包含HTTP脚本和CSS，Gazelle强制执行更严格的安 全策略以对抗网络攻击。由于Alexa前100网站并不使用SSL，我们选择了另一些使用SSL的网站：amazon.com、

> mail.google.com、mail.microsoft.com、blogger.com以及一些其它常见的银行网站。这些网站都没有违反我们的 策略。
>
> 限制框架导航：父框架（frame）只能访问它的子框架，而不能访问子框架的子框架或其它来源（principal）的框架。在Gazalle中那些访问 非子框架的代码将无法运行。通过限制导航范围，Gazelle可以防止在正常网站中访问到恶意网页。Barth等人分析了这一类攻击并称之为“递归整合攻击”。我们无法用我们的测试框架来自动测试是否有网站违反了这一策略。

这种隔离的代价颇为昂贵，对于 mytime.com 这种包含跨来源框架的复杂网页来说，渲染时间可能会翻倍。不过这些影响可以通过改进浏览器内核的组合显示内容部分来降到最低。

你可以在微软研究院的网站上查看论文全文，比较一下与现有的浏览器有什么不同，比如 Google Chrome。

原文链接：http://www.infoq.com/cn/news/2009/07/Gazelle

相关内容：

- MINIX 3 承诺比Windows或Linux更安全
- 微软互联系统部门负责人详解Azure服务平台
- Atom发布协议是一个失败吗？
- Silverlight提供了脱离浏览器的体验
- Flash遥居首位，Silverlight紧追不舍

[热点新闻]

4 个Office应用将会推出在线版：

Word、 Excel、 PowerPoint和OneNote

作者 Abel Avram 译者 王瑜珩

微软计划提供在线版本的 Office 2010 ,让用户能够在浏览器中使用一些轻量级的 Office 应用。

就在 Google 宣布以 Google Chrome OS 进军操作系统领域后不久 ,微软宣布将 会在云中运行 Office 应用程序，用户可以通过浏览器使用 Word、Excel、PowerPoint 和 OneNote。这些应用程序将用 Javascript 和 Ajax 技术构建，而不会使用 Silverlight。支持的平台包括 PC 上的 IE、Firefox 和 Safari，以及像 iPhone 这样的移动设备。Office 2010 目前还只是技术预览版，微软全球合作伙伴大会的出席者将收到微软的邀请，来评估这个版本。Office 2010 计划于 2010 年上半年上市。Web 版并不在目前的技术预览版中，但是会包含在今年晚些的版本中。

为了与 Google Docs 和 Zoho 这些对手竞争，Office Web 版将会完全免费，并可以支持 5 亿用户通过 Windows Live 进行访问。至于其中是否会含有广告，目前还不得而知。

Office Web 版并不会取代 Office 的桌面版，它将会缺失一些重要的特性：

> Capossela（微软商业部高级副总裁）认为Offlice Web和桌面版拥有不同的用户。Capossela说“我们并没有将Web版作为桌面版的克隆，我们希望两个版本能够适合用户使用的设备”。在最近的一个季度中，微软的Office桌面版销售下降了30%。

其中缺失的一个特性就是协作编辑功能。在 Office 2007 中，这项功能被作为补丁和插件添加进来，而在即将到来的 Office 2010 中会得到进一步的改进。目前还不清楚协作编辑是否会在以后添加到 Office Web 版中。据 Office 2010 的项目经理 Chris Bryant 称，Excel Web 会有这项功能，而 Word 和 PowerPoint 则不会有，原因是用户对此不感兴趣。

原文链接：http://www.infoq.com/cn/news/2009/07/Microsoft-Office-Web

相关内容：

- RIA特别专题和Flash开发平台工具下载资源发布
- 8 月 8 日QClub北京RIA技术深度探析活动，欢迎报名
- GraniteDS不断发展
- ZK发布 3.6.2 版本：性能提升，include模式
- 使用Adobe Flex进行模型驱动开发

[热点新闻]

中国人寿构建国内首个Silverlight企业级应用

作者 霍泰稳

周微软中国在北京发布了 Microsoft Silverlight 3 和 Expression 3。新品 Silverlight 3 支持更多视音频编码标准，支持浏览器外运行 Silverlight 应用以及对大幅图形的性能改进等。中国人寿目前已经基于 Silverlight 3 研发了国内首个 Silverlight 企业级应用——养老金精算咨询系统。

关于 Silverlight 3 的详细特性，请阅读 InfoQ 中文站以前的报道“微软发布 Silverlight 3 正式版及相关信息”。值得一提的是为了吸引使用 Adobe Photoshop 和 Adobe Illustrator 的设计人员，Expression Blend 3 提供了针对这些文件的导入功能。设计者可自由查看并且逐层导入 Photoshop 文件，这些文件层可轻松进行重新组合，所有元素也保留其原有格式——比如层、层的位置、可编辑文本和矢量等，并在 Expression Blend 3 内保持可编辑状态。

发布会上，来自中国人寿养老保险信息技术部的副总经理王勇和高级主管吴磊演示了国内首个全部基于 Silverlight 的企业级应用。作为中国最大的商业保险集团、《财富》全球 500 强企业，中国人寿的这一案例具有一定的代表性。据王勇介绍：

> Silverlight 3在企业级应用中能够提供更加丰富的用户体验，增强了公司应用系统的友好性、易用性和先进行，实现了客户友好性与专业化技术的结合。

在采访吴磊的过程中，InfoQ 中文站编辑了解到中国人寿选择 Silverlight 的原因和历程：

> 和很多人理解的不一样，这次我们使用Silverlight不是微软推销的结果，而是我们先找到微软。在对决定使用何种技术时，我们事先进行了比较，包括Adobe的Flash平台。调查后发现Flash没有很好的后端语言支持，而Silverlight有.NET平台。另外企业级应用很关注前后端技术的联动，以及需要所使用的技术是连续性的，也就是不断代，防止像Borland和BEA这样中途被收购的情况出现。综合这些原因，我们选择了Silverlight。

吴磊笑称，希望通过 Silverlight 的控件、主题等为用户提供比较炫的应用，为国企的呆板形象正名。通过其现场的演示，编辑发现 Silverlight 3 在养老金精算咨询系统中对数据的筛选和对应、搜索、分页等方面都有很好的支持。

据微软中国开发平台合作部大中华区总经理谢恩伟介绍，Silverlight 自发布以来在全球范围内已经有超过 1/3 的电脑下载并安装了 Silverlight，这次发布的 Silverlight 3 不仅满足了市场需求，也使开发团队更加高效地合作——从概念设计到原型开发，从交互实现到测试部署等。除了中国人寿的企业级应用，发布会现场新浪财经和搜狐娱乐也分别演示了自己基于 Silverlight 的应用：新浪 A 股行情银光版和搜狐高清影视剧等。据编辑观察，目前搜狐高清影视剧平台使用的依然是 Adobe Flash 应用。

欢迎访问 InfoQ 中文站的 RIA 技术专题，了解 RIA 领域最新技术进展，比如 Adobe 发布 Flash Builder 4，RIA 和 Ajax 技术的现状与展望，以及用 Microsoft Visual Studio 开发 Flex——Amethyst IDE 等。

原文链接：http://www.infoq.com/cn/news/2009/07/microsoft-silverlight3-solutions

相关内容：

- RIA特别专题和Flash开发平台工具下载资源发布
- 微软发布Silverlight 3 正式版及相关信息
- 微软发表.NET RIA Services的路线图
- Silverlight和Java的互操作
- 虚拟座谈会：RIA和Ajax技术的现状与展望

InfoQ中文站

www.infoq.com/cn

我们的**使命**：成为关注软件开发领域变化和创新的专业网站

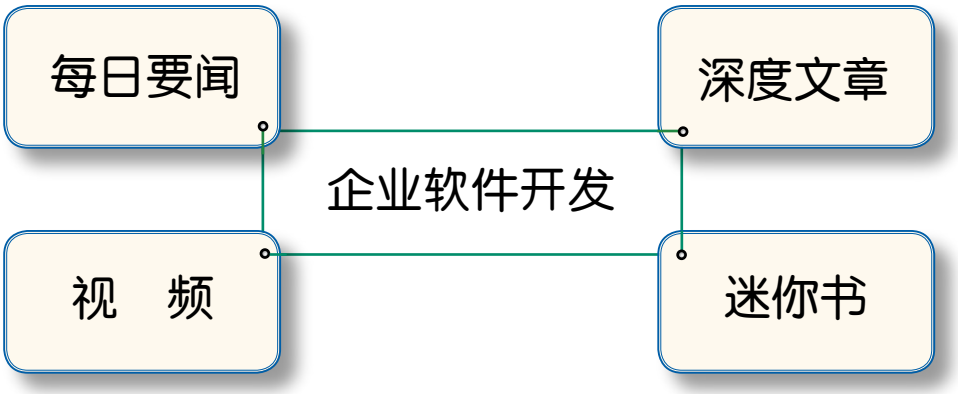
我们的**定位**：关注高级决策人员和大中型企业

我们的**社区**：Java、.NET、Ruby、SOA、Agile、Architecture

我们的**特质**：个性化RSS定制、国际化内容同步更新

我们的**团队**：超过30位领域专家担当社区编辑

……



讨 论 组：groups.google.com/group/infoqchina
编辑电邮：editors@cn.infoq.com
广告合作：sales@cn.infoq.com

[推荐文章]

J2EE应用下基于AOP的抓取策略实现

作者 Manjunath R Naganna 译者 张龙

使用了 O/R Mapping 工具的典型 J2EE 应用都会面临这样一个问题：如何通过最精简的 SQL 查询获取所需的数据。很多时候这可不是轻而易举的事情。默认情况下，O/R Mapping 工具会按需加载数据，除非你改变了其默认设置。延迟加载行为保证了依赖的数据只有在真正请求时才会被加载进来，这样就可以避免创建无谓的对象。有时我们的业务并不会使用到依赖的那些组件，这时延迟加载就派上用场了，同时也无需加载那些用不上的组件了。

典型情况下，我们的业务很清楚需要哪些数据。但由于使用了延迟加载，在执行大量 Select 查询时数据库的性能会降低，因为业务所需的数据并不是一下子获得的。这样，对于那些需要支持大量请求的应用来说可能会产生瓶颈（可伸缩性问题）。

来看个例子吧，假设某个业务流程想要得到一个 Person 及其 Address 信息。我们将 Address 组件配置成延迟加载，这样要想得到所需的数据就需要更多的 SQL 查询，也就是说首先查询 Person，然后再查询 Address。这增加了数据库与应用之间的通信成本。解决办法就是在一个单独的查询中将 Person 和 Address 都得到，因为我们知道这两个组件都是业务流程所需的。

如果在 DAO/Repository 及底层 Service 开发特定于业务的 Fetching-API，对于那些拥有不同数据集的相同领域对象来说，我们就得编写不同的 API 进行抓取并组装了。这么做会使 Repository 及底层 Service 过于膨胀，最终变成维护的梦魇。

延迟抓取的另一个问题就是在获取到请求的数据前要一直打开数据库连接，否则应用就会抛出一个延迟加载异常。

说明：如果在查询中使用预先抓取来获取二级缓存中的数据时，我们将无法解决上面提出的问题。对于 Hibernate 来说，如果我们使用预先抓取来获取二级缓存中的数据，那么它将从数据库而不是缓存中去获取数据，哪怕是二级缓存中已经存在该数据。这就说明 Hibernate 也没有解决这个问题，从而表明我们不应该在查询中通过预先抓取来获得二级缓存中的对

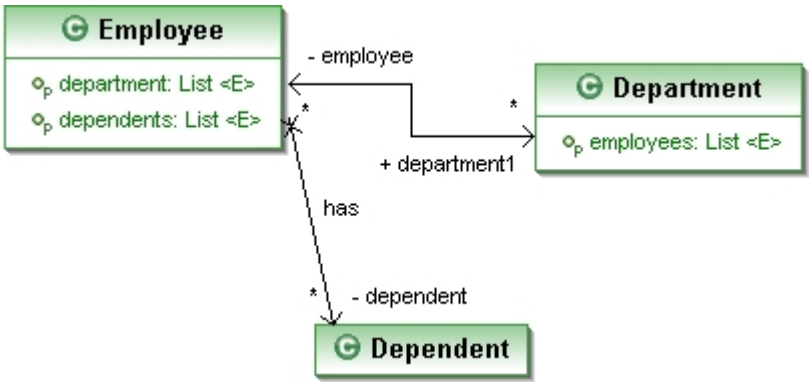
象。

对于那些可以让我们调节查询以获取缓存对象的 O/R Mapping 工具来说，如果缓存中有对象就会从缓存中获取，否则采取预先抓取的方式。这就解决了上面提到的事务/DB 连接问题，因为在查询的执行过程中会同时获取缓存中的数据而不是按需读取（也就是延迟加载）。

通过下面的示例代码来了解一下延迟加载所面对的问题及解决办法。考虑如下场景：某领域中有 3 个实体，分别是 Employee、 Department 及 Dependent。

这三个实体之间的关系如下：

- Employee 有 0 或多个 dependents。
- Department 有 0 或多个 employees。
- Employee 属于 0 或 1 个 department。



我们要执行三个操作：

1. 获取 employee 的详细信息。
2. 获取 employee 及其 dependent 的详细信息。
3. 获取 employee 及其 department 的详细信息。

以上三个操作需要获取并呈现不同的数据。使用延迟加载有如下弊端：

- 如果对实体 employee 所关联的 dependent 和 department 这两个实体使用延迟加载，那么在操作 2 和 3 中就会生成更多的 SQL 查询语句。
- 在多个查询语句的执行过程中需要保持数据库连接，否则会抛出一个延迟加载异常，这将导致数据出现问题。

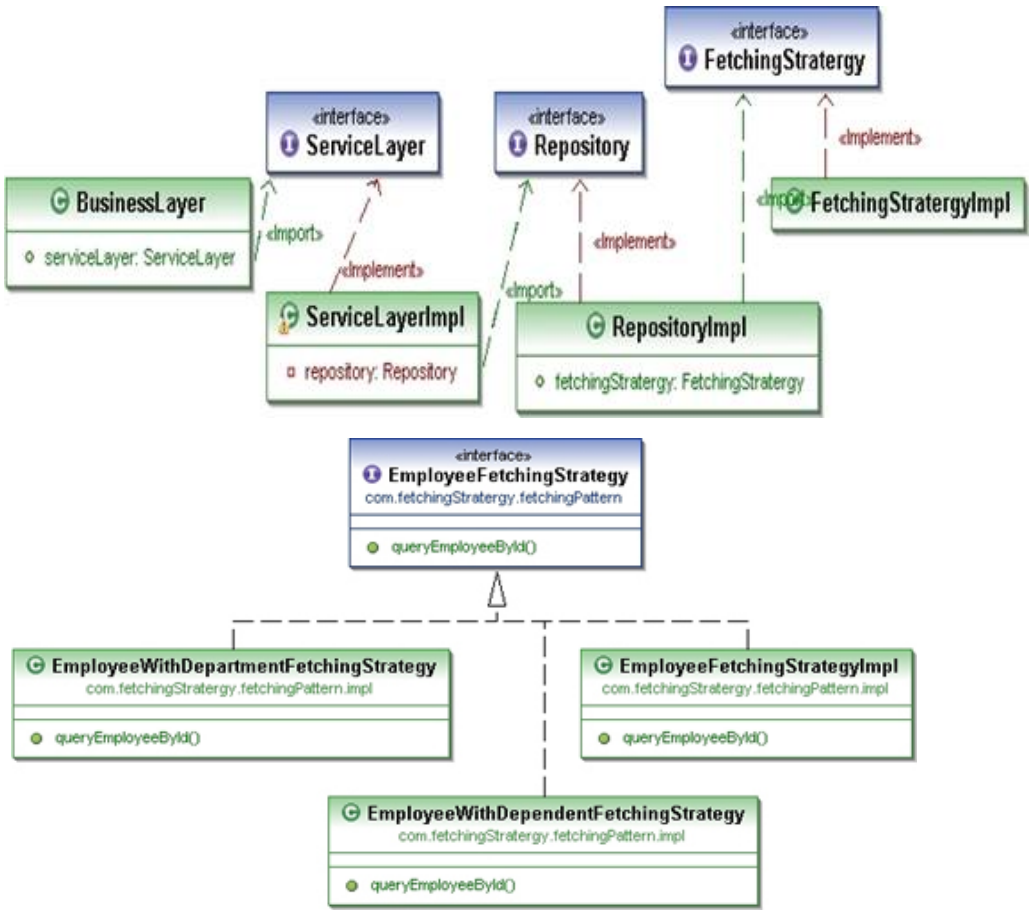
但另一方面，使用预先抓取也存在如下弊端：

- 对 employee 所对应的 dependents 和 department 采取预先抓取会产生不必要的数据。
- 无法在特定的场景下对查询进行调优。

在 Repository/DAO 或底层服务中使用特定于操作的 API 可以解决上述问题，但却会导致如下问题：

- 代码膨胀——不管是 Service 还是 Repository/DAO 类都无法幸免。
- 维护的梦魇——不管是 Service 还是 Repository/DAO 层，只要有新的操作都需要增加新的 API。
- 代码重复——有时底层服务需要在获取的实体上增加某些业务逻辑，与之类似，还要在数据返回前检查 DAO/Repository 层的查询响应以验证数据可用性。

为了解决上面这些问题，Repository/DAO 层需要根据不同的业务情况执行不同的查询来获取实体。就像 Aspect 类所定义的那样，我们可以根据特定的操作使用不同的抓取机制来覆盖 Repository/DAO 类所定义的默认抓取模式。所有的抓取模式类都实现了相同的接口。



```
public Employee findEmployeeById(int employeeId) {
   List employee =
hibernateTemplate.find(fetchingStrategy.queryEmployeeById()
,
      new   Integer(employeeId));
  if(employee.size() == 0)
  return null;
  return (Employee)employee.get(0);
}
```

Repository 类使用了上述的抓取模式来执行查询，如下代码所示：

Repository 类中的 employee 的抓取策略需要根据实际情况进行调整。我们决定将 Repository 层的抓取策略调整到 Repository 和 Service 层外，放在一个 Aspect 类中，这样当需要增加新的业务逻辑时只需修改 Aspect 类并增加一个针对于 Repository 的抓取策略实现即可。这里我们使用了面向方面的编程（Aspect Oriented Programming）以根据业务的不同使用不同的抓取策略。

什么是面向方面的编程？

面向方面的编程（AOP）可以通过模块化的形式实现实际应用中的横切关注点，如日志、追踪、动态分析、错误处理、服务水平协议、策略增强、池化、缓存、并发控制、安全、事务管理以及业务规则等等。对这些关注点的传统实现方式需要我们将这些实现融合到模块的核心关注点中。凭借 AOP，我们可以在一个叫做方面（aspect）的独立模块中实现这些关注点。模块化的结果就是设计简化、易于理解、质量提升、开发时间降低以及对系统需求变更的快速响应。

接下来读者朋友们可以参考 Ramnivas Laddad 所著的《AspectJ in Action》一书以详细了解 AspectJ 的概念以及编程方式，还可以了解一下 AspectJ 的开发工具。

Aspect 在抓取策略实现上扮演着重要角色。抓取策略是个业务层的横切关注点，它会随着业务的变化而变化。Aspect 对于特定的业务逻辑下使用何种抓取策略起到了至关重要的作用。这里我们将对抓取策略的管理放在了底层服务和 Respository 层之外。任何新的业务都可能需要不同的抓取策略，这样我们就无需修改底层服务或是 Respository 层的 API 就能应用新的抓取策略了。

FetchingStrategyAspect.aj

```
/**
    Identify the getEmployeeWithDepartmentDetails flow where
you need to change the fetching
    strategy at repository level
*/
pointcut empWithDepartmentDetail(): call(*
EmployeeRepository.findEmployeeById(int))
&& cflow(execution(*
EmployeeDetailsService.getEmployeeWithDepartmentDetails(int
)));

/**
   When you are at the specified poincut before continuing
further update the fetchingStrategy in
   EmployeeRepositoryImpl to
EmployeeWithDepartmentFetchingStrategy
*/
before(EmployeeRepositoryImpl r): empWithDepartmentDetail()
&& target(r) {
r.fetchingStrategy = new
EmployeeWithDepartmentFetchingStrategy();
}

/**
  Identify the getEmployeeWithDependentDetails flow where you
need to change the fetching
  staratergy at repository level
*/
pointcut empWithDependentDetail(): call(*
EmployeeRepository.findEmployeeById(int))
&& cflow(execution(*
EmployeeDetailsService.getEmployeeWithDependentDetails(int)
));

/**
  When you are at the specified poincut before continuing
further update the fetchingStrategy in
   EmployeeRepositoryImpl to
EmployeeWithDependentFetchingStrategy
*/
before(EmployeeRepositoryImpl r): empWithDependentDetail() &&
target(r) {
r.fetchingStrategy = new
EmployeeWithDependentFetchingStrategy();
}
```

这样，Repository 到底要执行何种查询就不是由 Service 和 Repository 层所决定了，而是由外面的 Aspect 决定，纵使增加了新的业务也无需修改底层服务和 Repository 层。决定执行何种查询的逻辑就成为一个横切关注点了，它被放在 Aspect 中。Aspect 会根据业务规则的不同在 Service 层调用 Repository 层的 API 之前将抓取策略注入到 Repository 中。这样我们就可

```
pointcut empWithDependentAndDepartmentDetail(): call(*
EmployeeRepository.findEmployeeById(int))
&& cflow(execution(*
EmployeeDetailsService.getEmployeeWithDepartmentAndDependen
tsDetails(int)));

before(EmployeeRepositoryImpl r):
empWithDependentAndDepartmentDetail() && target(r) {
    r.fetchingStrategy = new
EmployeeWithDepartmentAndDependentFetchingStaratergy();
}
```

以使用相同的 Service 和 Repository 层 API 来满足各种不同的业务规则了。

来看个具体示例吧，该示例会同时抓取一个 employee 的 Department 和 Dependent 的详细信息。我们需要对业务层进行一些变更，增加一个方法：getEmployeeWithDepartmentAndDependentsDetails(int employeeId)。实现新的抓取策略类 EmployeeWithDepartmentAndDependentFetchingStaratergy，后者又实现了 EmployeeFetchingStrategy 并重写了 queryEmployeeById 方法，该方法会返回优化后的查询，可以在一个 SQL 语句中获取所需数据。

由 Aspect 将上述的抓取策略注入到相关的业务中，如下所示：

如你所见，我们并没有修改底层业务与 Repository 层而是使用 Aspect 和一个新的 FetchingStrategy 实现就完成了上述新增的业务。

现在我们来谈谈关于二级缓存的查询优化问题。在上面的示例代码中，我们对 department 实体进行一些修改并配置在二级缓存中。如果对 department 实体采取预先抓取，那么对于同样的 department 实例，纵使它位于二级缓存中，每次也都需要查询数据库。如果不在查询中获取 department 实体，那么业务层就需要参与到事务当中，因为我们并没有将 department 实体缓存起来而是通过延迟加载的方式得到它。

这样，事务声明就从底层移到了业务层，虽然我们知道该业务需要哪些数据，但 O/R Mapping

工具却没有提供相应的机制来解决上面遇到的问题，即预先抓取缓存中的数据。

对于那些没有缓存的数据来说这种方式没什么问题，但对于缓存数据来说，这就依赖于 O/R Mapping 工具了，因为只有它才能解决缓存数据问题。

该示例附带的源代码详细解释了抓取策略。该 zip 文件含有一个工程示例，阐述了上面谈到的所有场景。你可以使用任何 IDE 或是使用 aspectj 编译器从命令行执行代码。在执行前请确保 jdbc.properties 文件与你机器上的信息一致并创建示例应用所需的表。

你可以使用 Eclipse IDE 以及 AJDT 插件运行代码，请按照下面的步骤进行：

解压缩下载好的代码并将工程导入到 Eclipse 中。

配置 Resources/dbscript 目录下的 jdbc.properties 文件中的数据库信息。

完成上面的步骤后请执行 resources\dbscript\tables.sql 脚本 ,这将创建该示例应用所需的表。

以 AspectJ/Java 应用的方式运行 Main.java 文件来创建默认数据并测试上面的抓取策略实现。

结论

本文介绍了如何通过不同的抓取策略从后端系统中获取数据 ,这是以模块化的方式根据业务需求实现的，同时又不会导致底层服务或 Repository 层过度膨胀。

关于作者：Manjunath R Naganna 目前供职于阿尔卡特朗讯公司，担任高级软件工程师一职，尤其专注基于 Java/J2EE 的企业应用设计与实现。他感兴趣的领域包括 Spring Framework、领域驱动设计、事件驱动架构以及面向方面的编程。对于 Hemraj Rao Surlu 对本文所做的编辑与排版工作，Manjunath 表达了自己的谢意，同时他也很感谢自己的领导 Ramana 与 Saurabh Sharma 在百忙之中抽出时间对本文进行审阅并提出很多重要的反馈信息。

原文链接：http://www.infoq.com/cn/articles/fetching-aop

相关内容：

- 领域模型管理与AOP
- 使用AOP实现应用程序失败转移
- Aspects：一个处理注解的简单工具？
- Ramnivas Laddad谈AOP选型
- Embarcadero公布Delphi Prism平台开发路线图

[推荐文章]

RGen：Ruby建模和代码生成的框架

作者 Martin Thiede 译者 杨晨

简介

本文介绍了支持以“Ruby 方式”进行建模和代码生成的 RGen 框架[1]。从 MDA 和 MDD[2]（但是除开那些不严格遵守这些方法的行为）的意义上说，我使用了“建模”这个名词：模型是元模型的实例。元模型即是（或者是很大程度上接近于）领域特定语言（DSL）。模型转换被用来将模型转换成不同元模型的实例，代码生成是一种将模型转换成文本输出的特殊转换方法。

RGen 受到了 openArchitectureWare（oAW）[3]这个有着相似应用范围的 Java 框架的影响。RGen 的核心思想不仅仅是使用 Ruby 在框架内作为应用程序的实现逻辑，而且还用来定义元模型，模型转换和代码生成。RGen 通过对每个切面提供内在的 DSLs，简化了这个过程。其他的项目也证明了 Ruby 十分适合在在这种情况下使用。一个著名的案例即是 Ruby on Rails[4]，它包含了一些内置的 Ruby DSLs。但是，oAW 使用了一些外部的 DSLs 来定义模型转换和代码生成。

经验告诉我们 RGen 方法是非常的轻量级，而且极其灵活，使得开发更加高效，部署更加简易。我发现在那些启发式的项目中，即发现缺少支持的工具，却没有预先制定工具开发计划的项目中特别有用。使用 Ruby 和 RGen 我们能够以最小的努力来开发需要的工具，而且人们将会从这个工具中受益非凡。

使用诸如 RGen 和 oAW 这样框架的典型应用是代码生成器（例如在嵌入式设备中）和构建以及操纵模型的工具，通常以 XML 或者一种自由的文本或者图形语言表示。在本文中，我将会使用“C++代码生成器的 UML 状态图”作为例子。在现实世界中，我们仍然在上述的项目中使用 RGen 来为汽车的嵌入式电子控制单元（ECU）进行建模和生成代码的工作。

模型和元模型

建模框架最重要的基本方面是表示模型和元模型的能力。元模型描述了特殊目的的模型看起来会是什么样，即定义了领域特定语言的抽象语法。典型地说，一个建模框架与元模型的应用和这两者之间的相互转换相关。

RGen 在 Ruby 中采用了一种前向的模型和元模型表示方法，就像面向对象的语言一样：对象用来表示模型元素，类用来表示元模型元素。模型和元模型之间的关系以及它们的 Ruby 表示法如图 1 所示。

为了支持领域特定语言，一个建模框架必须支持自定义的元模型。RGen 通过提供元模型定义语言，简化了这个过程，这个语言看起来和领域特定语言很相似。像其他的 DSL 一样，元模型定义语言的抽象语法也是由其元模型来定义，也就是所谓的元-元模型。

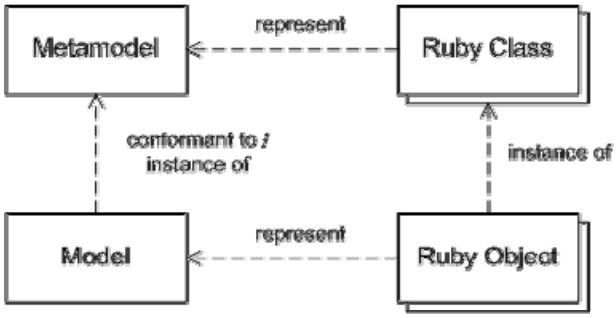


图 1：模型，元模型以及其 Ruby 表示

不像元模型，元-元模型在 RGen 里面已经得到了修复。此架构使用了 ECore -- Eclipse 建模框架（EMF）[5]的元-元模型。图 2 是 ECore 元模型的一个简单视图：在 ECore 中，元模型基本由一些以层级的包组织起来的类组成，这些类的属性和引用相互指向。从多个超类中可以抽象出一个类来。引用是无向的，但是可能会和一个相反的引用连接在一起，因此成为有向的引用。引用的目标便是其类型，必须是一个类；目标的角色便是引用的名字。属性是（非类的）数据类型的实力，可能是原生类型或者是枚举类型。

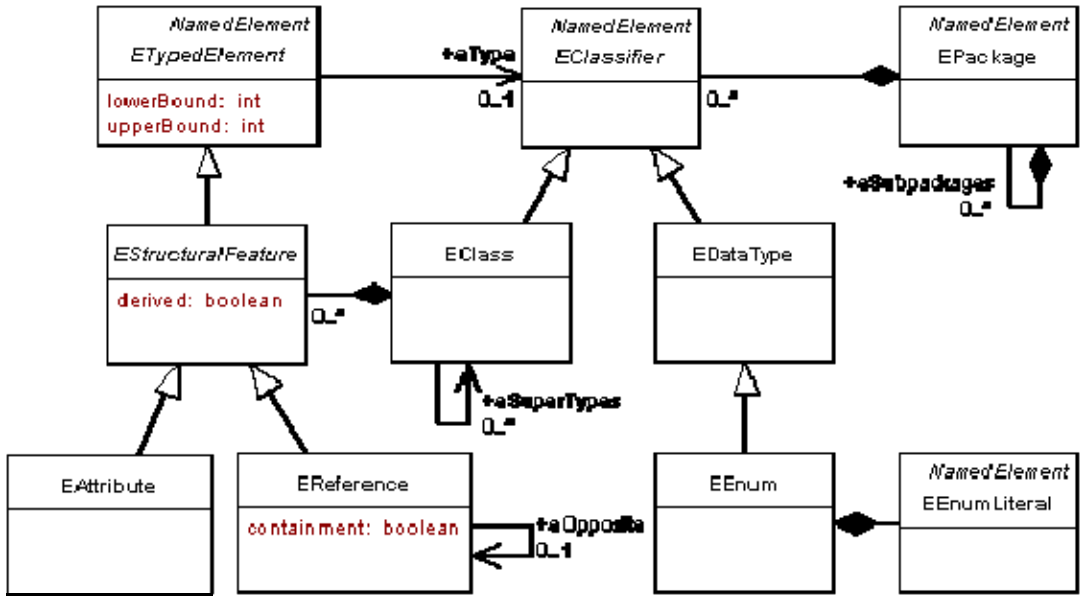


图 2：ECore 元模型的简化视图

和其他例如 oAW 的框架相反，RGen 的元模型定义语言的具体语法是 Ruby 形式，一种内部的 DSL。列表 1 描述了简单的状态机元模型：代码使用普通的 Ruby 关键词来定义一个模块 1 和一些类分别表示一个元模型包和元模型类。为了从普通的 Ruby 类和模块中分辨出这些元素，需要一些额外的代码：模块加上了一些特殊的 RGen 模块扩展（1），而且这些类都是继承于 RGen 元模型的基类 MMBase（2）。

元模型类的超类关系由 Ruby 类继承来表示（3）。注意到 Ruby 原始不支持多重继承，但是得益于其灵活性，一个特殊的 RGen 命令可以支持这个特性。这样，这个类必须是 MMMultiple（ , , ……）的返回值，这个方法是一个全局方法，用于在全局中构建一个中间超类。

```
# 表1：元模型的状态机样例
module StatemachineMetamodel
  extend RGen::MetamodelBuilder::ModuleExtension          #
(1)
  class ModelElement < RGen::MetamodelBuilder::MMBase     #
(2)
    has_attr 'name', String
  end
  class Statemachine < ModelElement; end                  #
(3)
  class State < ModelElement; end
  class SimpleState < State; end
```

```
  class CompositeState < State; end

  class Transition < ModelElement

    has_attr 'trigger', String                              #
(4)

    has_attr 'action', String

  end

  Statemachine.contains_one_uni 'topState', State           #
(5)

  Statemachine.contains_many_uni 'transitions', Transition

  CompositeState.contains_many 'subStates', State,
'container'

  CompositeState.has_one 'initState', State

  State.one_to_many 'outgoingTransitions', Transition,
'sourceState'

  State.one_to_many 'incomingTransitions', Transition,
'targetState'

end
```

元模型的属性和引用由 MMBase 提供的特殊的类方法指定。因为 Ruby 类的定义是在相关代码执行的时候才被解释。在一个类的定义域内，当前对象是这个类的一个对象，所以这个类对象的方法能够被直接调用 2.

has_attr 方法用来定义属性。它的参数是属性的名字以及一个原生的 Ruby 数据类型 3（4）。RGen 在内部将 Ruby 类型映射成 ECore 的原生类型，这个例子中是转换成 EString。对于引用的每个规范，都有一些方法可用。 contains_one_uni 和 contains_many_uni 定义了单向的引用，而 one_to_one，one_to_many 和 many_to_many 则是双向的。这些方法在源类的对象中调用，第一个参数是目标的角色，然后是目标类和在双向引用（5）中源的角色。注意，目标类需要在能够被引用之前定义好。因为这个原因，大多数 RGen 元模型中的引用都是在类的定义外定义的。

当模型创建之后，元模型 Ruby 类会被初始化。属性值和引用目标都保存在相关对象的实例变量中。由于 Ruby 不允许对实例变量的直接访问，于是需要使用专用的存取方法。在 Ruby 的传统表示法中，getter 方法的名字和相关变量保持一致，setter 方法也是一样。setter 方法能够用在表达式的左边。

上述的类方法通过元编程方式来构建需要的存取方法。除开对实例变量的存取，这些方法还需要检查参数的类型，在特定时候需要抛出异常。运行时类型校验是构建于 Ruby 的顶端，

此处是不需要原生的类型校验 4。在一对多的引用中，getter 方法会返回一个数组，setter 方法也用数组来添加或者删除引用对象。在双向引用中，存取方法自动地加上反引用。

表 2 举了这样一个例子：常规的 Ruby 类初始化机制（1）能够被用来创建对象以及一个特殊的构造器，这个构造器能够方便地设置属性和引用（4）。注意包的名字需要标示出类的名字。将包中的模块加入到当前的命名空间之后，就能够避免重复（3）。状态 s1 的名字属性可以被设置，getter 方法的结果也需要检查（2）。一个传出转换被添加到 s1（to-many）中，然后自动创建的后向引用（to-one）会被检查（5）。转换的目标状态被设置为 s2，通过显式地复制（to-one），结果队列包含了一个元素 t1（to-many）。第二个目标状态创建之后使用另外一种转换连接到源状态。最后，所有传出转换的目标状态被设置成 s2 和 s3（7）。方法 targetState 是在 outgoingTransitions 的结果（数组）上再被调用的。这种简化的记法是可以被接受的，因为 RGen 扩展了 Ruby 数组，通过中转调用一个未知方法，存取其中的元素，然后将输出放入一个单独的集合中。

```
# 表2：状态机元模型的初始化样例。
s1 = StatemachineMetamodel::State.new                          #
(1)
s1.name = 'SourceState'                                        #
(2)
assert_equal 'SourceState', s1.name
include StatemachineMetamodel                                  #
(3)
s2 = State.new(:name => 'TargetState1')                        #
(4)
t1 = Transition.new
s1.addOutgoingTransitions(t1)                                  #
(5)
assert_equal s1, t1.sourceState
t1.targetState = s2                                            #
(6)
assert_equal [t1], s2.incomingTransitions
s3 = State.new(:name => 'TargetState2')
t2 = Transition.new(:sourceState => s1, :targetState => s3) #
(7)
assert_equal [s2,s3], s1.outgoingTransitions.targetState
```

如上所示，RGen 元模型定义语言创建了需要表示 Ruby 元模型的模块、类和方法。不仅如此，

元模型本身还可以作为一个普通的 RGen 模型。因为 RGen 包含了 ECore 的元模型，ECore 的元模型用其自身的元模型定义语言来表达。元模型的 RGen 模型能够通过表示元模型元素的 Ruby 类或者模块中的 Ecore 方法来存取。

列表 3 的例子是：在 StatemachineMetamodel 模块上调用 ecore 方法得到了 EPackage 的一个实例（1），在 State 类上调用会得到一个 EClass 的实力（2）。而且这两个都是属于同一个模型，事实上名字叫做“State”的 EClass 是叫做 “StatemachineMetamodel”的 EPackage 中的一个分类器（3）。元模型 Rgen 模型能够像其他的任何 RGen 模型那样被操作。样例代码说明了“State”类的超类有一个叫做“name”的属性（4）。

```
# 表3：存取ECore元模型
smPackage = StatemachineMetamodel.ecore
assert smPackage.is_a?(ECore::EPackage)                    # (1)
assert_equal 'StatemachineMetamodel', smPackage.name
stateClass = StatemachineMetamodel::State.ecore
assert stateClass.is_a?(ECore::EClass)                     # (2)
assert_equal 'State', stateClass.name
assert smPackage.eClassifiers.include?(stateClass)         # (3)
assert
stateClass.eSuperClasses.first.eAttributes.name.include?('name') # (4)
```

作为一个普通的模型，元模型模型能够被任何可用的序列器和初始器序列化和初始化。RGen 包含了一个 XMI 序列器以及 XMI 初始器，使得元模型能够和 EMF 进行交换。同样地，元模型模型能够作为 RGen 模型转换的源或者目标，例如从或者到一个 UML 类模型。模型转换将会在下节中讲到。最后，使用 RGen 元模型生成器，元模型模型能够返回到 RGen 元模型 DSL 表示。图 3 总结了不同的元模型表示方法以及它们之间的关系。

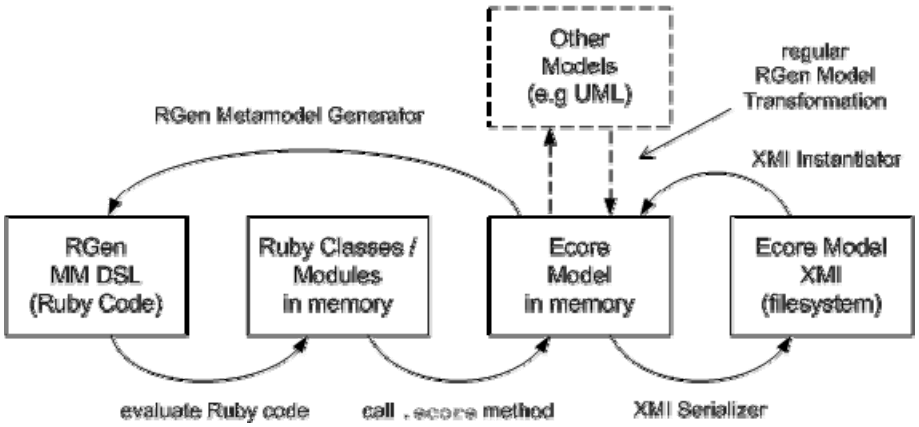


图 3：RGen 元模型表示总结

RGen 元模型模型提供了类似于 EMF 的在元模型上的反射机制。反射机制对于程序员来说是非常有用的，例如，当实现一个自定义的模型序列器或者初始器的时候。事实上元-元模型是 ECore 用来确保大量的建模架构的可交换性：使用 RGen 元模型生成器，任何 ECore 元模型能够直接在 RGen 中使用。

表 4 表示了元模型模型到 XML（1）的序列化过程，和使用元模型生成器来重新生产 RGen DSL 表示法（2）一样。注意在这两个例子中，元模型是被 StatemachineMetamodel 的 ecore 方法返回到根 EPackage 元素所引用。生成的元模型的 DSL 表示法可以从文件中读取和解释（3）。为了避免在初始的类、模块和重载的类、模块之间的名字冲突，解释是在名字空间中进行。如果不考虑在重载版本中的“instanceClassName”属性值，那么这两个模型是一样的。

```
# 列表4：序列化元模型
File.open("StatemachineMetamodel.ecore","w") do |f|
  ser = RGen::Serializer::XMI20Serializer.new
  ser.serialize(StatemachineMetamodel.ecore)             #
(1)
  f.write(ser.result)
end
include MMGen::MetamodelGenerator
outfile = "StatemachineModel_regenerated.rb"
generateMetamodel(StatemachineMetamodel.ecore, outfile)  #
(2)
module Regenerated
  Inside = binding
end
```

```
File.open(outfile) do |f|
  eval(f.read, Regenerated::Inside)                          #
(3)
end
include RGen::ModelComparator
assert modelEqual?( StatemachineMetamodel.ecore,           #
(4)
  Regenerated::StatemachineMetamodel.ecore,
["instanceClassName"])
```

现在 RGen 提供了对运行时元模型动态改变的有限支持。尤其是，ECore 元模型模型的改变不会影响内存中的 Ruby 元模型类和模块。但是，现在我们仍然致力于实现 Ruby 元模型表示的动态版本。动态元模型包含了动态类和模块，紧密地和 EClass 还有 EPackage ECore 元素联系在一起。当 ECore 元素被修改的时候，动态类和模块将会立刻改变它们的行为，即使实例已经存在。这个特性能够支持下节将要讲到的模型转换的高级技术。

RGen 最大的一个优势便是使用内部 DSLs 以及和 Ruby 紧密关联所获得的好处。这样使得程序员能够程式化地创建元模型类和模块，然后调用类的方法来创建属性和引用。利用这个优势的一个应用程序便是一种 XML 初始器，它在运行时根据遇到的 XML 标签和属性，以及一系列的映射规则，动态地创建目标元模型。RGen 分发版包含了这个初始器的原型版。

另外一个有意思的是使用内部 DSL，可以将元模型嵌入到常规代码中。这个特性是非常用帮助的，因为代码有的时候必须要处理复杂的，内部有联系的数据结构。开发者也许会考虑(元模型) 类，属性和引用的结构，然后在定义域内使用它们。使用这种元模型方法，开发者能够决定在运行时 5 自动地检查属性和引用。

模型转换

许多现实世界的建模应用能够从一些元模型的使用中获益。举例来说，一个应用程序可能会有内部元模型和一些特定的输入输出元模型。模型转换经常被用来将一个元模型的实例转换成另外一个元模型的实例，如图 4。

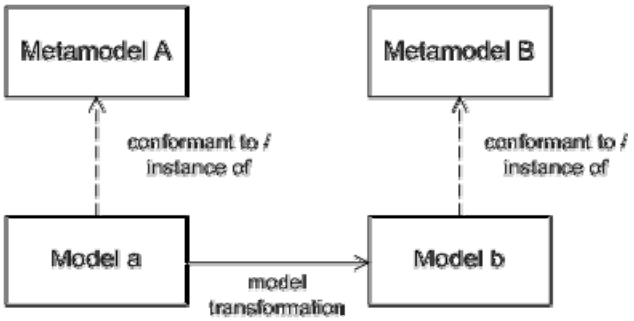


图 4：模型转换

考虑上面介绍的Statemachine例子，UML1.3 的Statechart模型的输入可以通过模型转换来添加。RGen包含了UML1.3 版的元模型，以RGen的元模型DSL来表示。RGen同样也包含了一个XMI初始器，允许直接通过一个UML工具存储的XML文件创建一个UML元模型的实例。图 5 给出了一个来自于[6]的输入状态图的例子。

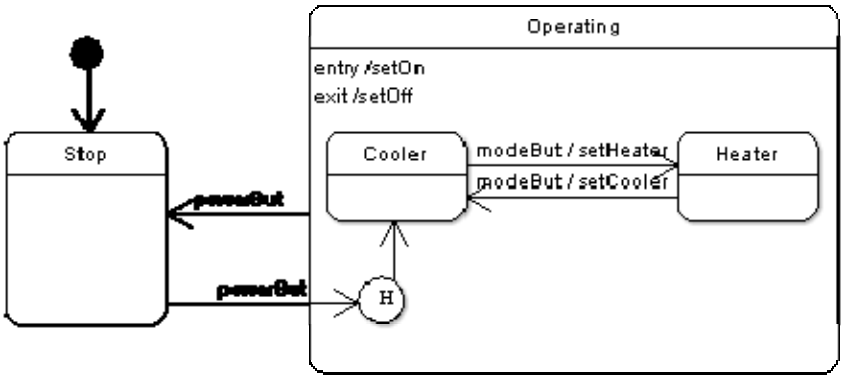


图 5：UML 状态图样例

除开创建一个新的目标模型 ,模型转换也可以用来修改源模型。这种情况叫做在位模型转换，它要求在转换的过程中，元模型的元素能够被改变。这种特性现在还不被 RGen 支持，但是如之上所述，我们现在在做这方面的工作。

举例来说，在位模型转换能够用在需要向后兼容地读取老版输入模型的场合下。新版工具中的输入元模型的每次更改能够使用一个内建的在位模型转换来提供向后支持。每一次这种转换只是服务于元模型和模型的一些少少改变，但是他也许需要用于大规模数据场合。在同样的源模型上使用一系列的在位转换，输入模型的迁移能够非常高效[6]。

像之前解释过的元模型定义 DSL 一样，RGen 提供了一个内部的 DSL 来定义模型转换，RGen 模型转换规范包括了源模型的单个元模型类的转换规则。规则定义了目标元模型类以及目标属性和引用的赋值，后者包括了应用转换规则所得到的结果。

图 6 以一个例子给出了转换规则的定义和应用[7]：源元模型类A的规则指定了目标元模型A'，定义了多引用b'以及单引用c'的赋值。b'的目标值是转换源引用b的元素的结果。这意味着使用了元模型类 B（见下文）的相关规则（b':=trans(b)）。在RGen中，数组的转换结果也是一个数组，每个元素都是各自转换过。同样地，c'的值被指定为每一个引用c的元素的转换结果（c':=trans（c））。元模型类C的转换规则反过来指定了引用b1'和b2'的值，b1'和b2'是引用了源引用b1 和b2 的元素的转换结果（b1':=trans(b1)，，b2':=trans(b2)）。对于元模型类B的转换，在这个例子中不需要更多的赋值。

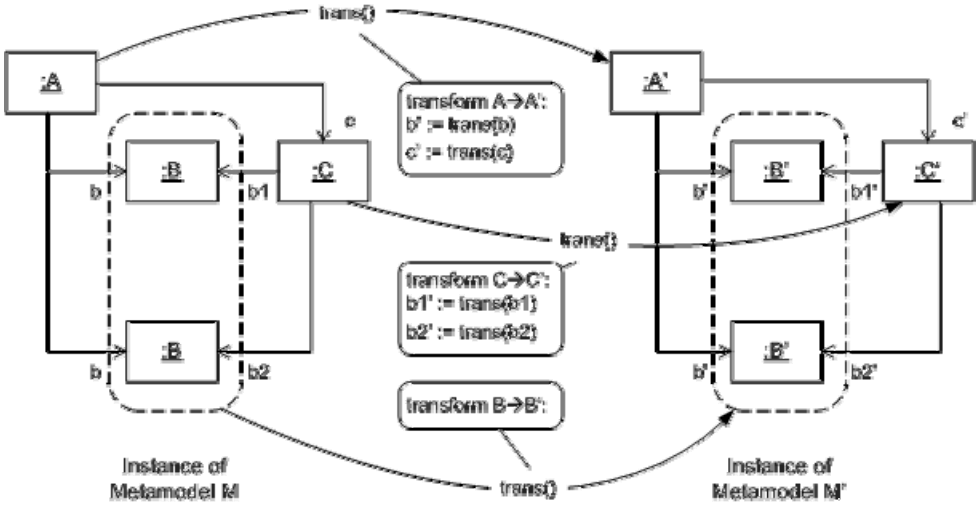


图 6：转换规则的定义和应用

作为一个内部 DSL，模型转换语言采用了 Ruby 明文作为具体的语法。每一个模型转换都是由一个 Ruby 类来定义，这个 Ruby 类是通过一些特定的类方法，继承于 RGen 的 Transformer 类。最重要的是，转换类方法定义了转换规则，将源和目标元模型作为参数。属性和引用的赋值由一个 Ruby Hash 对象来指定，它将属性和引用的名字映射到实际的目标对象。Hash 对象是由一个和转换方法调用有关的代码块创建 ,这个转换方法调用是在元模型元素的上下文中使用。

注意，转换规则能够递归地使用其他规则。RGen 转换机制关心的是整个使用过程，它缓存了每个转换的结果。一个规则的代码块的执行结束标志是任何递归使用的规则的代码块执行结束。当自定义代码被加到这个代码块的时候，确定性行为尤其重要。表 5 是一个模型转换例子 ,它从 UML 1.3 元模型转换成之前介绍过得状态图元模型 :一个新类 UmlToStatemachine 是继承于 RGen Transformer 类，为了是的目标类名字尽可能短（1），目标元模型模块被包含在当前命名空间中。一个常规的 Ruby 实例方法（在这个例子中叫做 transform）作为转换的入口点。它调用 trans 转换方法，当所有的状态机元素在输入模型 8（2）的时候触发转换。trans 方法寻找已定义的转换规则，使用转换类的方法从源对象的类开始，如果没有规则的

话，那么沿着其继承的层级向上寻找。 UML13：：StateMachine 的转换规则指定了将要被转换成 StateMachine（3）的实例的元素。注意源和目标元模型类都是普通的 Ruby 类对象，而且必须使用 Ruby 命名空间机制。相关的代码块创建了一个 Hash 对象，给属性“name”和引用“transitions”和 “topState”赋值。当在源模型元素上调用继承的方法的时候，这些值都会在代码块的上下文中计算出来。对于引用目标值，trans 方法会递归地调用。

```
#表5：状态机模型的转换样例
class UmlToStatemachine < RGen::Transformer                    #
(1)
  include StatemachineMetamodel
  def transform
    trans(:class => UML13::StateMachine)                       #
(2)
  end
  transform UML13::StateMachine, :to => Statemachine do
    { :name => name, :transitions => trans(transitions), # (3)
      :topState => trans(top) }
  end
  transform UML13::Transition, :to => Transition do
    { :sourceState => trans(source), :targetState =>
trans(target),
      :trigger => trigger && trigger.name,
      :action => effect && effect.script.body }
  end
  transform UML13::CompositeState, :to => CompositeState do
    { :name => name,
      :subStates => trans(subvertex),
      :initState => trans(subvertex.find { |s|
        s.incoming.any?{ |t|
t.source.is_a?(UML13::Pseudostate) && # (4)
        t.source.kind == :initial }})}
  end
  transform UML13::StateVertex, :to => :stateClass, :if
=> :transState do # (5)
```

```
    { :name => name, :outgoingTransitions => trans(outgoing),
      :incomingTransitions => trans(incoming) }
  end
  method :stateClass do
    (@current_object.is_a?(UML13::Pseudostate) &&        #
(6)
      kind == :shallowHistory)?  HistoryState : SimpleState
  end
  method :transState do
    !(@current_object.is_a?(UML13::Pseudostate) && kind
== :initial)
  end
end
```

引用几乎任何 Ruby 代码都能够在创建 Hash 对象的代码块中使用，所以一些高级的赋值成为了可能：在例子中 CompositeState 的目标模型有一个对初始状态的显式引用，尽管在源元模型中初始状态被标记为从一个“初始”伪状态引入转换而来。使用 Ruby 内建的 Array 方法，这个转换能够实现，首先寻找一个有这样引入转换的子状态，然后使用 trans 方法（4）转换。

转换方法能够选择性地使用一个方法，计算目标类对象。在上面的例子中，UML13::StateVertex 需要被转换成 SimpleState 或者是 HistoryState，取决于 stateClass 方法（5）的结果。当有更多而可选的方法参数的时候，规则可以有条件的制定出来。在例子中，规则不会在伪状态中使用，结果将会是 nil，因为没有规则应用。除开常规的 Ruby 方法，Transformer 类提供了方法类的方法，允许定义这样一类方法，这类方法的方法体是在当前转换的源对象（6）的上下文中被求值。为了防止二义性，当前转换源对象能够使用@current_object 实例变量存取。

由于调用转换类方法是使用的常规代码，它也可以以更加复杂精密的形式调用，允许对转换定义“脚本化”。实现 Transformer 类的拷贝类方法的一个不错的例子如表 6：方法使用一个源，然后选择性地使用一个目标元模型类，假设它们相同或者有相同的属性和引用。然后它在一个代码块中调用 transform 方法，自动地为每一个给定的元模型创建一个右赋值 hash 对象，通过元模型反射 9 从中寻找它的属性和引用。

```
#表6：Transformer的拷贝命令实现
def self.copy(from, to=nil)
  transform(from, :to => to || from) do
```

```
Hash[*@current_object.class.ecore.eAllStructuralFeatures.inject([]) {|l,a|
      l + [a.name.to_sym, trans(@current_object.send(a.name))]
    }]
  end
end
```

拷贝类方法也可以在任何元模型类中使用。如果需要对一个元模型做一个实例的深层次拷贝（clone），这里是一个创建给定元模型的副本的一般方法。表 7 的例子是 UML 1.3 元模型的拷贝转换。

```
#表7：UML1.3元模型的拷贝转换样例
class UML13CopyTransformer < RGen::Transformer
  include UML13
  def transform
    trans(:class => UML13::Package)
  end
  UML13.ecore.eClassifiers.each do |c|
    copy c.instanceClass
  end
end
```

RGen 框架中，另外一个转换机制的有意思的应用是元模型反射的实现。当调用 ecore 方法的时候，接受的类或者模块被注入到内建的 ECore 转换器中，然后被应用属性和引用的转换规则。因为机制是如此的灵活，不仅仅需要元模型类，而且需要将 Ruby 类作为“输入元模型”，所以这类实现是可能的。

代码转换

在 RGen 框架中，除开对模型的转换盒修改，代码生成是另外一种非常重要的应用。代码生成可以被认为是一种特殊的转换，将模型转换成文本输出。

RGen 框架包含一个基于生成器机制的模板，这个模板和 oAW 中的很相似。其他一些基于模板、模板文件和输出文件关系的模板在 RGen 和 oAW 中都是不同的：一个模板文件可能包含多个模板，一个模板可能创建多个输出文件，每个输出文件的内容可能是多个模板生成的。

图 7 的是这样一个例子：在文件“fileA.tpl”中定义了两个模板“tplA1”和“tplA2”，在文件“fileC.tpl”中定义了模板 “tplC1”（关键词定义）。模板“tplA1”创建了一个输出文件“out.txt”（关键词文件），在文件中写入了一行文字。然后扩展模板 “tplA2”和“tplC1”的内容，输出到相同的输出文件中（关键词扩展）。因为模板“tplC2”在一个不同的文件中，所以它的名字必须带上模板文件的相对路径。

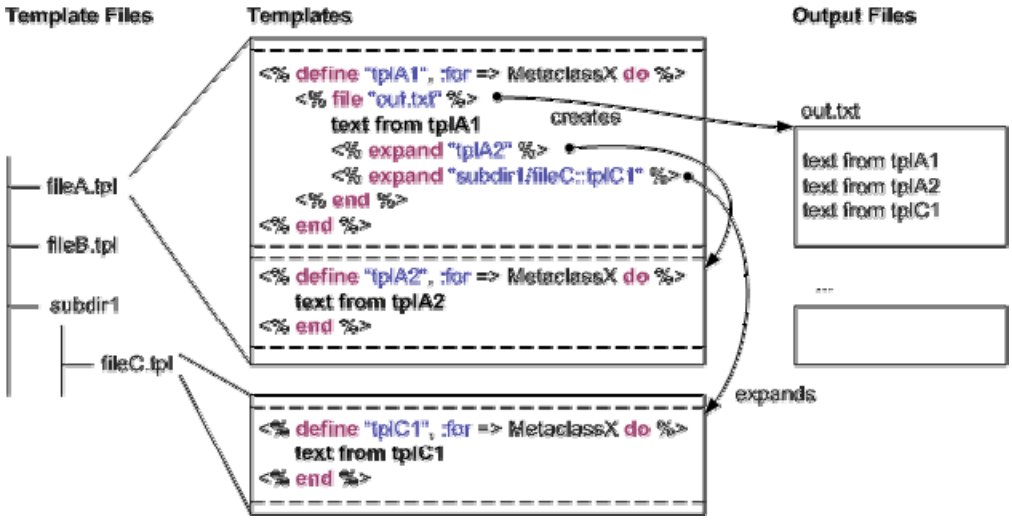


图 7：RGen 生成器模板

当 RGen 模板被扩展的时候，它们的内容会在模型元素的上下文中被求值。每一个模板都和一个元模型类相关联，这些元模型类都是使用：来定义属性和扩展元素。默认情况下，扩展命令会在当前上下文对模板进行扩展，但是不同的上下文元素可以使用：或者：foreach 属性来指定。在后面的例子中，一个数组中的每个元素都会按照其模板展开。模板同样也可以被重载，根据不同上下文类型和扩展的规则，自动地选择合适的模板。

RGen 模板机制是构建在 ERB（嵌入式 Ruby）之上的，它为 Ruby 提供了最基本的模板支持。ERB 是标准 Ruby 的一部分，允许使用标 签<%，<%=和%>将 Ruby 代码嵌入到任意的文本中去。Ruby 模板语言包括了 ERB 语法以及一些额外的关键词，可以像常规的 Ruby 方法一样实现。这样的话，模板语言成为了 RGen 的另外一个内部 DSL。构建在标准的 ERB 机制上，实现是非常轻量级的。

一个代码生成的主要内容便是格式化输出：因为模板本身应该是人类可读的，额外的空格会对输出的可读性造成影响。一些方法能够很好的处理这种情况。但是这些方法可能会需要一些时间，而且需要额外的工具来实现，并且，有的时候对于某种特定的输出不可用。

RGen 模板语言提供了一个简单的输出格式化器，不需要任何额外的工具：默认来说，行首和行尾的空格会被删除掉。开发者然后 可以通过显示的 RGen 命令来控制缩进和空行的创

建：iinc 和 idenc 用来设置当前的缩进级别，nl 用来插入一个空行。经验表明，添加格式化命令所付 出的努力是值得的。尤其是当特殊的输出需要格式化的时候，这种方法就非常有用。

例 8 给出了一个从状态图例子中得到的完整的模板。它用来产生一个为每个复合状态创建的 C++抽象类的头文件。跟随着状态模式和[6]，能够从这个类得到每个子状态的状态类。

```
#表8：状态机生成器模板例子
<% define 'Header', :for => CompositeState do %>          # (1)
  <% file abstractSubstateClassName+".h" do %>
    <% expand '/Util::IfdefHeader',
abstractSubstateClassName %> # (2)
    class <%= stateClassName %>;
    <%nl%>
    class <%= abstractSubstateClassName %>                 #
(3)
    {
    public:<%iinc%>                                        #
(4)
      <%=abstractSubstateClassName%>(<%=stateClassName%>
&cont, char* name);
      virtual ~<%= abstractSubstateClassName %>() {};
      <%nl%>
      <%= stateClassName %> &getContext() {<%iinc%>
        return fContext;<%idec%>
      }
      <%nl%>
      char *getName() { return fName; };
      <%nl%>
      virtual void entryAction() {};
      virtual void exitAction() {};
      <%nl%>
      <% for t in (outgoingTransitions +
allSubstateTransitions).trigger %> # (5)
        virtual void <%= t %>() {};
```

```
      <% end %>
      <%nl%><%idec%>
    private:<%iinc%>
      char* fName;
      <%= stateClassName %> &fContext;<%idec%>
    };
    <% expand '/Util::IfdefFooter',
abstractSubstateClassName %>
  <% end %>
<% end %>
```

模板最开始是定义其名字和上下文元模型类，然后打开一个输出文件（1）。所有的 C/C++头文件必须有防止重复包含的宏定义，这个宏定义通常就是文件名的大写。模板“IfDefHeader”就是生成这样的宏定义（2）。在下一行开始进行类的定义（3）以及在“public”关键字之后增加缩进级别（4）。除了这些基本的方法之外，需要为每一个对象的传出转换声明一个虚方法。这个方法会被一个简单遍历所有相关的转换的 Ruby for 循环实现（5）。在这个 for 循环体中创建了方法的生命。为了将触发器的属性值写到输出中，这里使用了<%=而不是<%。在模板的最后，需要在页脚添加防止重复包含的宏定义。

一般来说，所有的不需要逐字拷贝的模板输出是从模型表示的信息中创建的。上下文模型元素的属性存取方法能够被直接读取，其他的模型元素能够通过引用的父辈方法得到。并且，能够在模型元素的数组上调用方法是非常有用的。

但是在很多情况下，模型中的信息需要在用来输出之前进行处理。如果计算过于复杂或者需要在不同的场合下应用，那么将其实现为一个单独的方法是一个不错的注意。在上述例子中，stateClassName，abstractSubstateClassName 和 allSubstateTransitions 是元模型类 CompositeState 的派生属性/派生引用，但是是以方法的形式表示。

这类派生属性或者派生引用可以作为常规的 Ruby 元模型类方法来实现。但是，因为 RGen 支持元模型类的多重继承，因此需要特别注意。用户定义的方法必须只能添加到一个特殊的“类模块”中，这个事每一个 RGen 元模型的组成部分，并且可以通过常量 constant ClassModule 存取。

```
#表9：状态机元模型扩展例子
require 'rgen/name_helper'
module StatemachineMetamodel
```

```
  include RGen::NameHelper                                    # (1)
  module CompositeState::ClassModule                          # (2)
    def stateClassName
      container ? firstToUpper(name)+"State" : firstToUpper(name) # (3)
    end
    def abstractSubstateClassName
      "Abstract"+firstToUpper(name)+"Substate"
    end
    def realSubStates
      subStates.reject{|s| s.is_a?(HistoryState)}
    end
    def allSubstateTransitions
      realSubStates.outgoingTransitions +                     # (4)
        realSubStates.allSubstateTransitions
    end
  end
end
```

表 9 的例子是派生属性和派生引用的实现：首先 StatemachineMetamodel 包模块被打开，然后加入一个 helper 模块（1）。在这个包模块中，CompositeState 类的类模块被打开（2）。在类中的方法实现利用了来自于混合模块的常规父辈方法、其他的派生属性、引用以及可能的方法（3）。allSubstateTransitions 的方法利用了 RGen 允许在数组上调用元素方法的特性，递归地实现。

注意在表 9 中的方法没有像原始元模型类那样在同一个文件中定义。Ruby 的“open classes”特性即可在不同的文件中定义方法。虽然可以再同一个文件中定义方法，但是建议使用一个或多个元模型扩展文件。这样的话，使用了 helper 方法的元模型就不会“乱成一团”了，尽管这种方法经常只是在特定的场合下使用。

在这个例子中，扩展文件被命名为“statemachine_metamodel_ext.rb”，扩展方法被用来生成代码。在实际应用中，根据项目的规模，对于一般扩展使用“statemachine_metamodel_ext.rb”，

对于生成特殊扩展使用 “statemachine_metamodel_gen_ext.rb”是非常有用的。因为保存在另外一个文件中，因此生成器逻辑非常清晰。

代码生成需要加载模板文件和扩展根模板。表 10 的例子便是入耳在状态图中完成这个：首先创建 DirectoryTemplateContainer 的实例（1）。容器需要知道输出目录，例如：输出文件（关键词文件）创建的目录。而且也需要知道作为引用的命名空间的元模型。然后指定模板目录，加载模板文件（2）。在这个例子中，用来作为代码生成的模型元素通过模型转换放到当前的 RGen 环境中。根上下文元素（例如元模型 Statemachine 的实例）从环境中得到（3）。代码生成便是从扩展根模板开始的（4）。

#表 10：启动生成器

```
outdir = File.dirname(__FILE__)+"/../src"

templatedir = File.dirname(__FILE__)+"/templates"

tc =
RGen::TemplateLanguage::DirectoryTemplateContainer.new( #
(1)

   StatemachineMetamodel, outdir)

tc.load(templatedir)                                      #
(2)

stateMachine = envSM.find(:class =>
StatemachineMetamodel::Statemachine) # (3)

tc.expand('Root::Root', :foreach => stateMachine)         #
(4)
```

表 11 的是最终生成器输出的一部分：文件“AbstractOperatingSubstate.h”是由表 8 的模板生成。注意现在不需要任何的后续处理。

```
// Listing 11: Example C++ output file
"AbstractOperatingSubstate.h"

#ifndef ABSTRACTOPERATINGSUBSTATE_H_

#define ABSTRACTOPERATINGSUBSTATE_H_

class OperatingState;

class AbstractOperatingSubstate

{

public:

   AbstractOperatingSubstate(OperatingState &context, char*
name);
```

```
   virtual ~AbstractOperatingSubstate() {};
   OperatingState &getContext() {
      return fContext;
   }
   char *getName() { return fName; };
   virtual void entryAction() {};
   virtual void exitAction() {};
   virtual void powerBut() {};
   virtual void modeBut() {};
private:
   char* fName;
   OperatingState &fContext;
};
#endif /* ABSTRACTOPERATINGSUBSTATE_H_ */
```

应用说明

Ruby 是一种能够让程序员以一种简洁的方式表达思想的语言，它能够高效地开发易于维护的软件。RGen 将建模和代码生成加入到 Ruby 中，使得开发者能够以一种简单而且熟悉的形式处理建模和代码生成。

根据简单的原则，RGen 框架在规模上是轻量级的，开发者无需关心依赖和规则。因此在日常的脚本中使用框架是非常灵活的。

作为一个动态解释语言，Ruby 在开发中引入了一些不同。一个最优秀的特性是类型检查上无需编译器支持。另外一个是编辑器支持自动完成。这些在 RGen 中也同样支持，因为 RGen 完全依赖于 Ruby 语言的特性。在特定情况下，类似于 oAW 的框架使用外部 DSLs 能够提供更好的编辑器支持。

Ruby 开发者和其他的动态类型语言例如 Python 和 Smalltalk 的开发者一样，也对使用的语言缺点如数家珍：缺失编译器检查需要更多的密集（单元）测试，不过这个是个好办法。缺失编辑器支持不能很好地利用动态语言特性，但是动态语言的优秀的内建编辑器支持是非常困难的。

尤其在大型项目中，就更加需要这些特性的力量了。当程序员需要添加（运行时）检查的时候，这些特性的好处就很明显了。但是，语言本身支持这些特性是因为它允许定义特定的项

目检查 DSLs。

RGen 元模型定义语言也可以看做是一个 DSL：它可以用来定义属性和引用，在运行时进行检查。也就是说一个 RG 元模型能够作为一个大型 Ruby 应用程序的股价。元模型同样支持开发者常识理解，尤其是使用 ECore 活 UML，甚至是图形可视化工具的时候。RGen 添加的类型检查也是保证程序和模型的核心数据的是处于一致性状态。

RGen 几年前启动的时候，只是作为一个将建模领域和 Ruby 绑定到一起的试验。正如前面说提及的，它已经在汽车工业的咨询项目中成功地作为一个原型工具。现在这个源性工具已经成熟，被用来在汽车电子控制单元（ECU）中作为常规的开发工具。

在这个工具中，一些元模型包含了大概 600 个元模型类。例如，我们的一个模型包含了大概 7000 个元素，最后会在大约一分钟内生成 90000 行代码。实现表明在特殊领域，基于 RGen 的方法能够比基于 Java 或者 C#的更容易完成，考虑到运行时和内存耗费。而且，这个工具能够单独地部署成一个大概 2.5MB 的可执行文件，包含了 Ruby 解释器，而且在服务器系统上运行不需要安装 10。

虽然 RGen 本身是基于内部 Ruby DSLs 的，但是它还不支持程序员创建新的内部 Ruby DSLs 的具体语法。它同样还没有提供外部 DSLs 例如生成解析器和与发起的具体语法。现在，自定义元模型（抽象语法）的实例需要被以文中提及的程式化方法或者使用已经存在的初始器来创建。这个话题当然属于未来的改进方案。

本文中所使用的完整的代码可以在 RGen 项目主页上下载[1]。

总结

基于 Ruby 的 RGen 框架提供了对处理模型和元模型的支持，能够定义模型转换和文本输出生成。由于使用了内部 DSLs，它和 Ruby 语言紧密地结合在一起。遵循 Ruby 设计原则，它是轻量而且灵活的，支持高效地开发，提供了编写简介可维护的代码方法。

RGen 在用于汽车工业作为建模和代码生成工具的时候是非常成功的。实现表明这种方法的高效，尽管还有着很多缺点，例如缺失编译器检查和编辑器支持。它仍然在运行时和内存使用上表现出了杰出的性能，你都不敢相信 Ruby 是一门解释语言。

除了 RGen 的工业应用，这个框架仍然在实验中使用。一个正在开发的扩展是对动态元模型的支持，这种动态元模型会在实例已经存在的情况下，运行的时候会发生改变。

原文链接：http://www.infoq.com/cn/articles/thiede-ruby-modelling

相关内容：

- SOA编程模型
- 将架构作为语言：一个故事
- 用Qi4j进行面向组合编程
- Michael Poulin炮轰SoaML
- 使用Adobe Flex进行模型驱动开发

时刻关注企业软件开发领域的变化与创新

InfoQ中文站
www.infoq.com/cn

Java — .NET — Ruby — SOA — Agile — Architecture

Java社区：企业Java社区的变化与创新

.NET社区：.NET和微软的其它企业软件开发解决方案

Ruby社区：面向Web和企业开发的Ruby，主要关注Ruby on Rails

SOA社区：关于大中型企业内面向服务架构的一切

Agile社区：敏捷软件开发和项目经理

Architecture社区：设计、技术趋势及架构师所感兴趣的话题

讨 论 组：groups.google.com/group/infoqchina
编辑电邮：editors@cn.infoq.com
广告合作：sales@cn.infoq.com

[推荐文章]

面向服务的经济学

作者 Enrique Castro-Leon&Jackson He&Mark Chang&Parviz Peiravi 译者 胡键

摘要

面向服务的目标是通过重用和标准化来实现 IT 服务交付过程中的结构性成本降低。然而，把传统系统转变成基于服务的系统需要将整个应用拆分成标准的服务，而且经常使用 Web 前端来推动服务在内部和外部的重用。记住一点很重要，然而，如何这么做，许多服务会被外包出去，进而导致更低的运营成本同时牺牲了员工利益。同时，向 SOA 业务模型的转变并非仅限于大型企业，它同样也适合中小型企业。

虚拟化和网格计算通过促进物理资源的重用和共享，从而使那些特殊资源（不论是计算机还是网络）的使用更为高效，使成本得到了降低。在这一节中，我们将探讨由面向服务导致的结构性经济变更。当前的大多数 IT 组织都处于使其预算可控的极大压力之下。他们的成本在不断增长，而预算却象图 1 中显示的那样日益平坦。服务概念和在服务级别的重用所带来的结构调整承诺长效地减轻这种成本折磨。

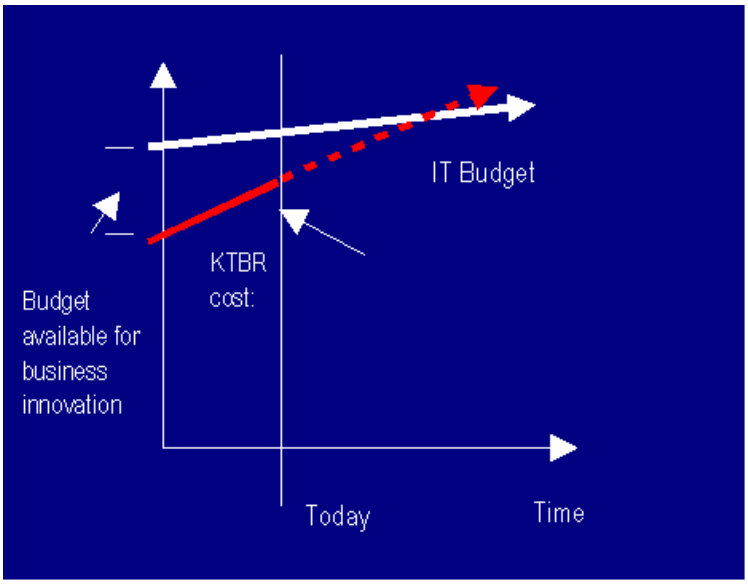


图 1：不可抗拒的遗留成本

从某种程度上讲，面向服务其实没有带来什么新的东西。但它却具有产生深远变革的潜力，这要归因于其中涉及的经济学。面向服务极大地降低了引入特定能力的成本。我们通过这 4 张图阐述了这种成本变化的某些方面。从概念上讲，有一部分 IT 预算是被用来维护现有项目的。这部分预算之所以重要，其原因在于，它是“使业务保持运转（keeps the business running，KTBR）”的一个组成部分。在大多数企业中，这个 KTBR 部分在预算中是不可获缺的。KTBR 的缺点是它只看到过去，而不是将来，因而它是用于拓展业务的残留部分。它的另一个问题是：不受控的 KTBR 部分的增长速度往往超过了整个 IT 预算，这种情形显然是不可持续的。

为了控制 KTBR 的增长，一系列策略已经在 IT 组织中被采用。让我们以图 2 中显示的外包为例进行说明。在引入外包（而且可能是海外的）之时，成本实际会因发生组织调整和洽谈合同而上升。一旦外包计划得以实施，成本可能会下降，但仍然有可持续性问题。另一个问题是，成本在提供外包服务的国家可能会大幅上升，几年后这些国家中发生的成本将会和以前的方式持平。

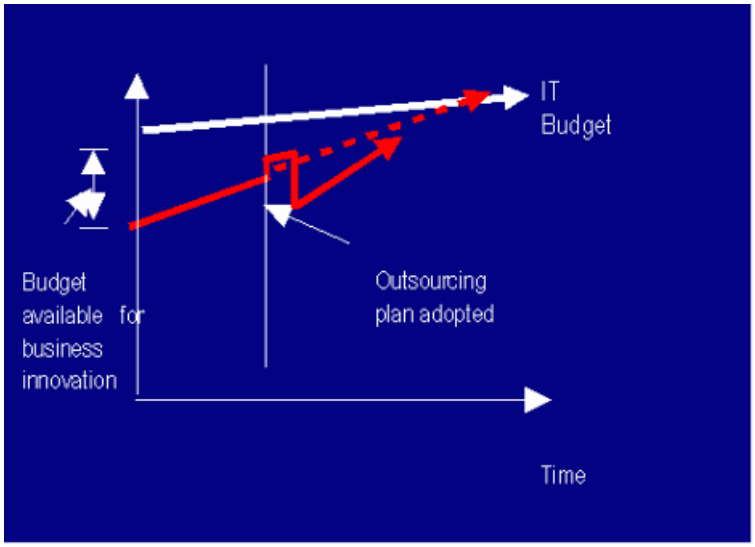


图 2：外包的效应

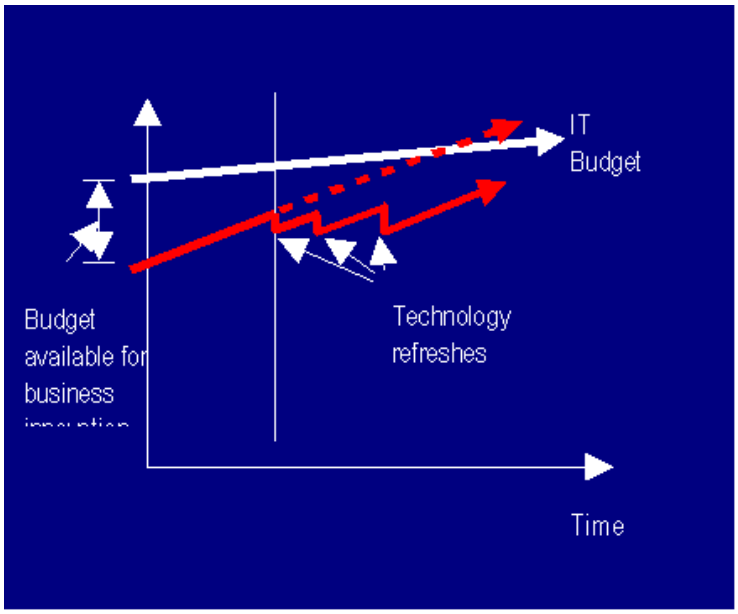


图 3：引进新技术的效应

第三种选择如图 3 所示：引进新技术，如 Intel® vPro™处理器技术，降低了处理业务的成本，可以看作成本挤压。成本可以通过积极实行技术应用得到管理，但这并不会改变其整体的上升趋势，而且没有太多企业愿意或有能力支撑这种技术革新计划。

最后，图 4 说明了面向服务是如何最终导致了一种结构性且可持续的成本降低，这都要归因于重用的协同效应。和外包一样，一开始成本也是向上增加的，这是因为需要进行先期投资。注意，这种转变可能需要花好几年才能完成，而且要求艰难的企业文化和组织结构的调整。

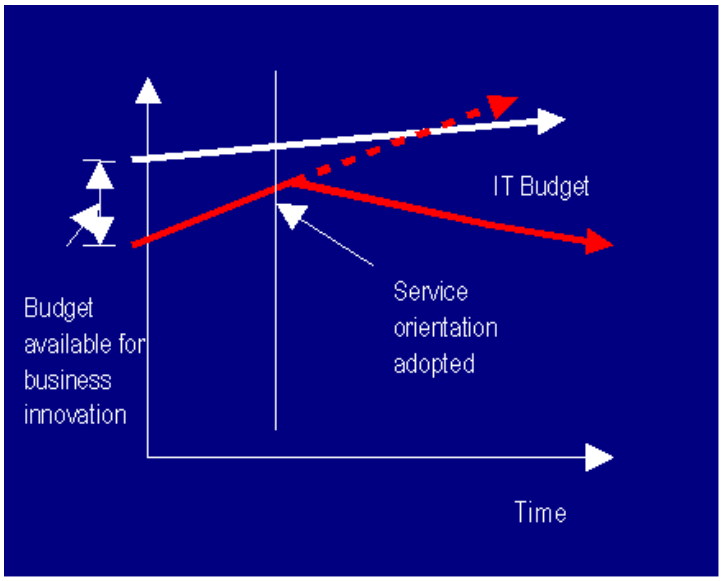


图 4：通过采用面向服务获得结构性成本降低

面向服务带来了模块化这一早已在软件工程社区广为人知超过 30 年的戒律，尽管如此，其

却极少应用公司内的项目中。前文已经说过，面向服务的目标是通过重用和标准化来实现 IT 服务交付过程中的结构性成本降低。和考虑互操作性和安全性一样，架构和规划同样需要更多的先期投入。但是，正如我们今天在软件工程中所看到的，随着越来越多具有同样作用的服务被开发出来得到重用，模块化和互操作性的好处将会显现。

总而言之，要想在合理可测的成本之下实现 IT 和业务的对齐是项具有挑战性的问题。最佳解决方案可以通过包含着业务流程改进（以面向服务的形式）、技术以及恰当业务规划的综合战略来实现。

面向服务架构的可扩展性

我们将在本节探讨可扩展性这一主题和 SOA 的采用模式。本节的讨论具有前瞻性，内容是关于尚未发生或发生中的事件和流程，因而其内容应该更多地应被视为一种思维激发练习，而不应该被视为对事实的陈述。

行业内 SOA 应用的普及有望极大降低 IT 项目规划、部署和运营的成本。那些从千年之交发生的网络公司集体崩溃死亡线上逃离，业务与 IT 对齐压力持续加大的 IT 组织对于这种趋势部分地起到了推波助澜的作用，他们唯恐自己成为下次预算消减下的牺牲品。这段时间内增加的法规合规特性为探索减轻法规压力的方法带来了更多激励。

把遗留应用转变成 SOA 可以通过一种创新的分解处理来实现，籍此，这些应用被分解成服务，他们具有基于 Web 服务的标准接口。然后，这些服务得到重用以支持其他应用。反过来说，也可以通过组装这些服务的标准接口来构建新应用。

由于 SOA 承诺通过重用来极大降低 IT 项目的规划、部署和运营成本，企业计算环境中 SOA 的采用呈增长趋势。然而，把传统企业应用转变成 SOA 应用的组织多数是大型企业。小型企业很少没有出现在这种 SOA 转变过程中。我们将看到，SOA 方法同样适合中小型企业（SMB）。

我们描述了一种新的 SOA 采用和服务交付模式，其内涵超越了在大型企业转变过程中所熟知的 SOA 的角色。

让面向业务的服务由公司防火墙外的独立服务提供商开发和交付，小型企业就可以挑选他们觉得有价值的服务。他们可以把这些服务“混搭（MashUp）”成最能满足他们业务需要的服务，其中遵循的 SOA 服务集成概念和大型企业采用的完全一样。因此，对小型企业而言，SOA 不再是一个抽象概念，也不再是一个大型企业才能玩得起的游戏。这就是由外向内（outside-in）的 SOA 采用模式，和传统的人们熟知的发生在大型企业中由内向外（inside-out）模式完全相反。

总的说来，在传统的由内向外方法下，服务是通过组合企业内部更简单的服务得到的，而由外向内的方法则假定小型组织能够从生态系统中可以找到服务和服务集成商提供的服务来构建服务。服务集成商可以依次选择其他厂商的服务组件来构建他们要提供的服务。由此下去，完全有可能发展成一个丰富多彩的生态系统。

中小型企业的影响不容忽视，因为其潜在数量相当的大：根据美国小企业管理局的统计，在美国，小企业代表了 99.7%的公司雇主，其雇员占私营企业员工总数的半数以上（ http://www.sba.gov/advo/stats/sbfaq.txt ）。

面向业务的这种由外向内的视角可以成为普及 SOA 采用和转变小型企业使用计算机技术的路标。

我们同时也意识到，这种由外向内的 SOA 模型需要花些时间才能成熟。一些重大技术障碍需要业界厂商的实质努力才能克服。与改变小企业经营过程中的业务交易所涉及的业务流程和人们的行为相比，这些技术挑战算不上什么。

但是，正如我们在消费者市场中互联网和 Web 2.0 的采用情况中所看到的，只要我们能够交付具有吸引力的收益并降低进入门槛，这种采用情景就能而且一定会发生。

了解这种变化将帮助参与者、服务消费者和供应商们发现其中的增值机会。

最初的筒仓状态

根据前面已经描述的常见进化模式，公司应用已被部署成图 5 中显示出来的烟囱结构，一台服务器一个应用，或服务器层驻留了一个完整的解决方案堆栈。具有讽刺意味的是，这种趋势正是由 15 年前出现的基于 Intel 的低成本服务器所推动的，它鼓励以物理服务器为部署单元。

在得到这些系统物理服务器之前，大约要花 2 周到半年的时间不等。一旦这些服务器到达，它们就会被装上操作系统、数据库软件、中间件和应用。为了支持一个业务的运行，实际需要多种这样的管道。某些 IT 组织在升级到 SAP 的企业资源计划（ ERP ）应用时是以大约 15 个烟囱为阶段进行的。

迈向 SOA 转变的第一步就是把部分筒仓打碎，转化成更小的逻辑组件，并增加 Web 服务前端以使这些组件可为当前和未来的应用所用。在这一阶段，需要识别冗余组件，并使之退役。

因而，在 SOA 中，整体应用被分解成标准化的服务。安装 Web 服务前端代表了一种额外的开发成本，其回报并不能立即产生。在打着 SOA 旗号所进行的破旧立新活动之下，实施团队在一开始就可能会发现这种为了支持将来重用而需进行的额外工作。要想确保这项附加工

作能够给组织带来更大好处，还需解决企业文化和行为方面的问题，即便它与项目的当前目标并不一致。这需要大量的传播和培训活动，而且即使如此，文化的转变也是缓慢且费劲的。

最后会达到一个平衡点，在该位置，某项目的这种额外实施成本通过重用过去项目的整体储备而得以结余。最终目标是得到一个正投资收益表。这还不是停止宣传的时候，因为储备可能还没有在各个组织中显现，而只是对那些可以观察多个项目的组织可见。

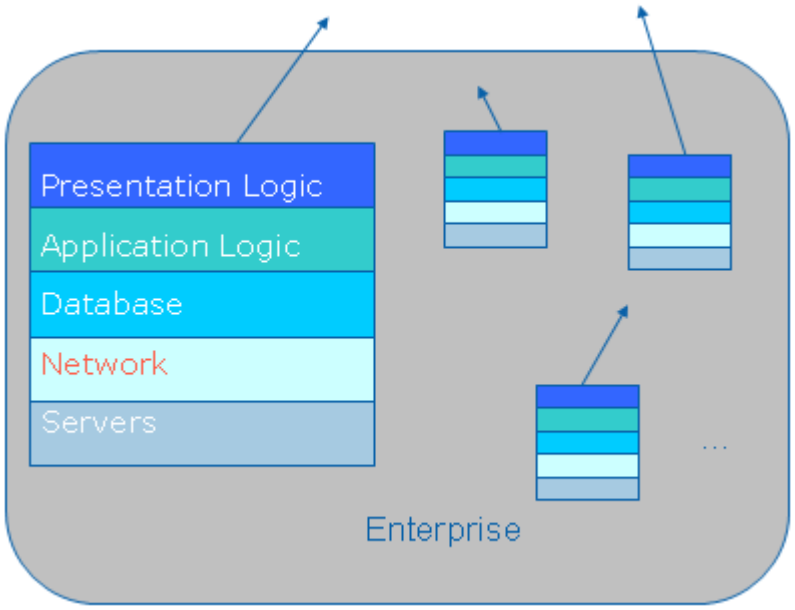


图 5：传统的应用烟囱

传统的由内向外的 SOA

图 6 显示了大型组织的 SOA 转变。堆栈层已经被服务组件替换掉了，由于有这么多的重用存在，使得这些堆栈几乎全部消失。

企业架构师可能发现某些功能是通用的，可以通过第三方厂商提供的产品替换掉。但是要注意，“通用性”的考虑取决于技术和生态系统的现状。对某些组织而言，它可能是指 HR 应用；而另一些可能是邮箱服务或者甚至是一整套 ERP 的实现。

即使从技术角度看转变的执行毫无破绽，然而它还是有可能带来破坏和痛苦：原来的内部服务组合变成越来越小的内核。如果内部服务被外包服务以低成本替换掉，整体成本就能降下来。这个小小的内核完全可能导致裁员和技能的重新调整。内部应用开发更少，并且需要具有业务和技术技能的人来管理与服务提供商的关系。

这种 SOA 转变过程可能由局部开始，其中那些非任务关键型服务是首要的替换对象，而 IT 内部开发团队则关注核心、复杂、任务关键型服务。

这些核心服务可能代表了企业核心的知识产权（IP）。某些公司可能在这一领域没有什么太多的限制，最后这个核心变得太小了，与外包服务相比显得无足轻重。达到这种状态就完成了由内向外的转变。

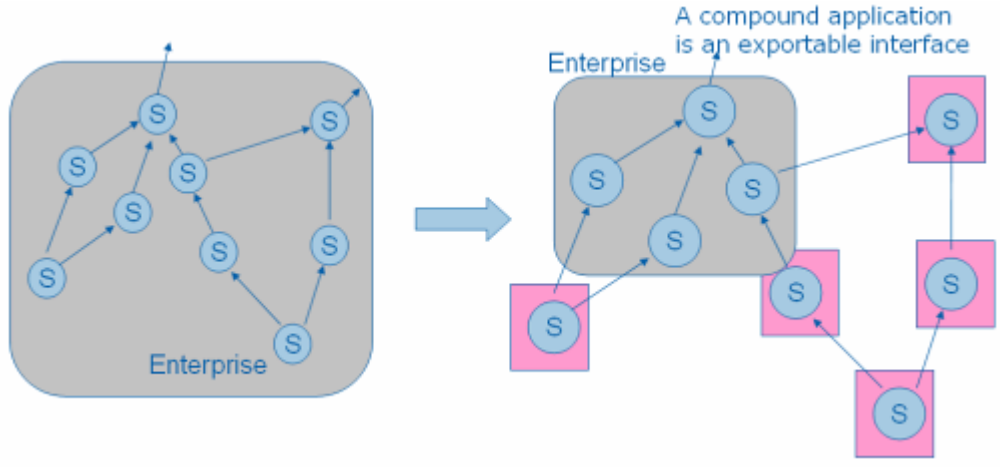


图 6：大型组织中的 SOA

由外向内的 SOA：中小型企业的 SOA

在前一节中，我们目睹了大公司中发生的从由内向外到由外向内的转变。在这一节中，我们可以看到相同的进化过程可以很自然的扩展至中小型企业（SMB）身上。其区别在于，该过程是发生在整个生态系统而非某个单独的大公司身上。

就其本身性质而言，SMB 通常不会有奢侈的大规模 IT 预算或大的 IT 部门。大部分这种公司只有一些懂 IT 的雇员兼职充当 IT 支持。他们也没钱购买那种“只面向内部”的 SOA 实施模型。因此，SMB 并没有迫切的要求去建立针对于由内向外过程的内部服务组合。

但是，如果我们假设大型企业成了 SOA 的第一批使用者，并且在这一过程中服务市场已经创建起来，那么 SMB 就可以完全跳过由内向外的过程。SMB 可以从外部购买服务。事实上，一旦生态系统成熟，由内向外的过程就会变得过时，因为今天它完全是从头开始构建内部应用。

在一个成熟的 SOA 环境，SOA 组合应用（Compound application）将主要使用外包服务构建。图 7 说明了这个概念。一个定义良好的业务流程（如采购订单的创建和处理或是一次银行的交易）代表了一组 SOA 服务，它们由不同的用户实例化并被集成到一个用户解决方案中以支持业务需要，而它们本身又是由业务驱动的。本质上，SMB 拥有者为自己企业挑选合适服务的过程非常类似当今 Web 2.0 世界的混搭。

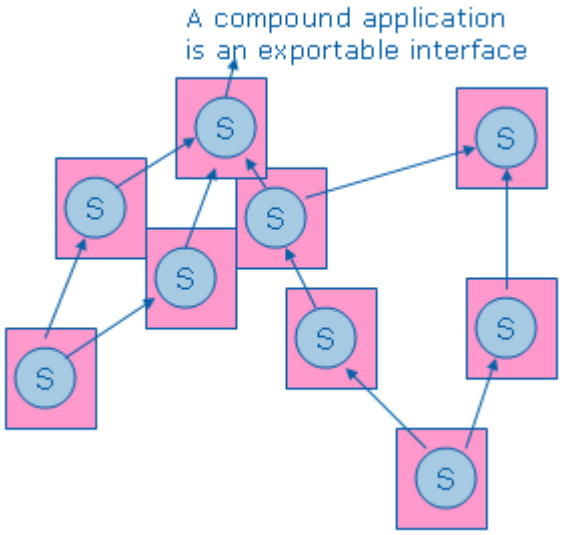


图 7：对 SMB 环境而言，其企业核心已经小到不可再分的程度了。

这种采用 SOA 的方法可能会导致令人惊异的结果：根据大型组织中为创建 SOA 而制订战略的经验来看，在 SOA 市场的某个成熟阶段，采用 SOA 的前提条件将不再是大型组织。SOA 同样也能在小型组织环境中遍地开花。这一结论同样也与 Web 服务的开放性和标准化概念一致。这些过程具有架构合理性和技术可行性，即便有大量的技术和社会行为障碍要克服。

SOA 在中小型企业中的采用过程是一种完全不同的形式 :重用并没有发生在大型企业的内部或组织之间——在考虑临界物质（critical mass）的必要条件时，我们将看到相同的临界物质——而是发生在整个经济学生态系统中。

换句话说，我们所建议的模型取决于多个生态系统厂商，他们提供的组合应用被用来构建更复杂的 SOA 风格的组合应用。在这样的环境下，我们可以预计某种程度的规范化，其中各服务由拥有合适专家的更小厂商提供。这种情况在图 8 中得到了反映。为了将这种方法和传统的“只面向内部”或“由内向外”的特定于大型企业的 SOA 方法区分开来，我们将其称为“由外向内”的 SOA。这种由外向内的方法并不仅限于中小型企业，它同样也可以发生在前一节所讲的大型企业中。

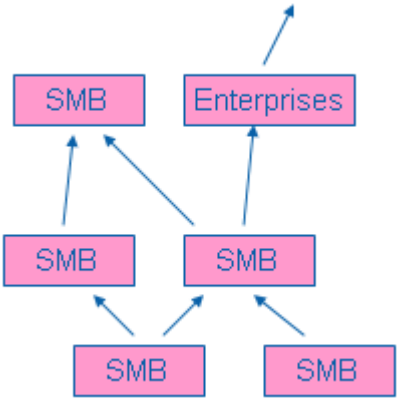


图 8：最终的状态是一个丰富多彩、技术和业务需求完美结合在一起的经济学生态系统

谁会有兴趣让这种由外向内的 SOA 转变成为现实？在经济学生态系统中的一条真理就是，某些参与者的成本方式被转换成另一参与者相同的收入报酬。当收获的价值远大于购买者所花成本，且卖家意识到由于这个价值还需要其他产品和服务时，整个生态系统的好处就会显现。

服务消费者一定会获利，因为通过利用组合服务的组合应用降低了整体购买应用功能的成本。软件工具提供商将通过提供针对于由外向内的 SOA 环境的产品获利，技术构建单元的提供商也可以利用相同方法盈利。这些构建单元必须具有支持自动供给和虚拟化设施概念的 SOI 能力。

服务提供者之间的关系将以相当复杂的方式发展。服务提供商将成为服务的提供者和消费者。甚至像 Amazon.com 这样的非传统服务提供者也开始开发剩余能力。这些公司将开始销售这些服务，创造新的收入来源。事实上，Amazon.com 并非一个好例子，因为它不是一家小公司。这个模型所适用的规模也可以是拥有 10–100 雇员的增值二手商（VAR）公司。

要使这套模式发挥作用，那些正规的服务组件，即那些从底层设计出来的服务，并不是必需的。可以通过给中间件进行填充使其行为与可组合服务一样来翻新图 5 中的传统烟囱应用，其方式类似于用那些屏幕抓取程序来延续遗留大型机应用的生命和用途一样。由外向内模型的障碍要比乍一看时要低，因为行业并不需要一直等待某个可重用服务的大型组合可用。

随着技术的成熟，以及更多参与者进入市场，我们可以预期会出现大量的服务作坊，绝大多数可以想象出来的应用都可以使用它们提供的服务来完成。这个市场随着地理区域的不同呈现高度的多样化，主要是受当地需求和法规的影响。在这种环境中，在构建特殊的功能时，通过在市场中把组成组件立约外包出去，其成本要比完全自己实现相同功能要便宜得多。

汽车或飞机行业的这种模式，具有巨大的供应网络，也将应用到软件应用。

这种由外向内的方法不同于传统的外包协议：洽谈把薪水计算功能外包出去可能得花上几个月，且还要面对面地坐在桌子旁边。相反，由外向内的交易将是高度自动化和动态的，使用开放标准以及自动化的注册中心和发现服务即可。象这样的一种交易可能花费不到一秒，至多花费几秒。

关于面向服务的经济学的更多信息，请参见 Enrique Castro-leon、Jackson He、Mark Chang 和 Parviz 所著的《虚拟面向服务网格的商业价值》（The Business Value of Virtual Service-Oriented Grids）。

原文链接：http://www.infoq.com/cn/articles/intel-services-economics

相关内容：

- 软件营销，App Store为什么会取得成功？
- 供敏捷软件开发使用的合同
- 将企业视为一个事件网络
- 基于SaaS模式的全程电子商务
- BPM不是软件工程

[推荐文章]

云计算的虚拟研讨会

作者 Abel Avram 译者 马国耀

云计算承诺提供几乎无限制的随需资源访问 ,人们一直追寻的业务延展性以及通过按需购买的方式降低成本。在这个虚拟研讨会中，InfoQ 希望从这些顶尖的专家们那里了解到云计算带来的好处以及使用时有何限制，私有云和公共云哪一个更好，是否需要云互操作，提供基础设施和提供平台的区别，客户如何加强规范遵守等等。

回答我们问题的专家们包括：

Jerry Cuomo，IBM 副总裁，WebSphere 产品线 CTO

David Linthicum，Blue Mountain Labs 的创始人

Geva Perry，GigaSpaces Technologies 云计算总经理

Jamin Spitzer，微软 Developer & Partner Evangelism Group 部总裁

云计算给企业带来的好处是什么？

David：基于旧分时模型，云计算是比传统方式更加追求共享、经济和高效的一种新方法。通过规模经济，企业将可以使用原先他们根本负担不起的资源，包括分析服务，企业应用以及在按时按需购买的基础设施服务等。此外，网络方面的好处是和使用 Web 绑定的资源的能力，比如社区网络，它比在本地集成资源的方式要容易得多。它是一种旧的计算方式在新技术市场上的新表现，并且，它使得企业能够更经济地重建计算基础设施以及企业架构。

Geva：云计算从两个方面推动企业进步。从战术上，提高效率，节省成本。从战略上，助力开发人员和企业用户快速和高效创新。

1）提高效率和节省成本：云计算利用规模经济的优势，通过让很多用户共享相同的基础设施，云的提供者可以获得更高的资源利用率，而原来只能在特殊的专属环境下才能达到。另

外，随着越来越多的公司将他们的 IT 提供成“云源”，云提供者可以更专业，更高效地运行大规模数据中心，而他们客户们只需要关注自己的核心业务。

2）快速创新：云计算加速了个人和企业的创新能力，缩短了产品和服务进入市场的时间周期。之所以云可以做到这点，是因为它节省了大量的 IT 预投资，通过流水线或自动流程（比如提供服务器，从测试环境到生产环境的移植等服务），向开发人员和企业用户提供直接的，自助的 IT 资源访问，从而提高了他们的生产力。

Jamin：下一代计算潮流——被称之为“便宜革命”的集中式计算资源和大量高性能设备带来的边界能力的结合——让用户和企业便利地选择远程计算资源和个性化设备体验。云为用户，企业和开发者带来这样的体验机会：可以利用更高效、更广阔的技术基础设施平台来创建和访问应用，从而增加现有投资和传统的部署选项（比如，企业数据中心，个人电脑等）。我们应该把云看成是现有计算平台的扩展。最终，一组良好定义的准则将会出现，帮助企业理解在什么时间，如何对长期持续的投资作出重新调整。

Jerry：我们实实在在地看到云计算模型让企业更智慧地工作。在当前金融危机的影响下，我们仍然看到完美的风暴式发生的事件，通过使用先进的技术（例如虚拟化）对齐业务模型（如 pay per sip）和标准/架构（如 Web 和 SOA），从而获得相当可观的计算输出。因此，它对任何人都有吸引力。企业不需要于先付费，并且/或者，将一部分 IT 运维外包出去，如此他们可以将更多宝贵的时间集中的他们的核心业务上。IT 可以将精力集中于基于应用程序的需求来扩展基础设施（过去系统使用率总体低下的日子将离我们远去）。软件开发人员不再等待 IT 提供系统，他们可以（在云中）自助获取合适的软件。

当公司决定采用云计算架构时，需要牢记哪些实施上的限制？

Jerry：当你思考云架构时，我们建议使用面向服务的方法。考虑到一个成功的面向服务的架构是从一些业务目标开始的，最好你也从业务目标开始。降低人力和能耗，加速产品到价值的转变是一些关键的业务指标。设定一个很低的门槛，让一些企业（或企业的部门）有使用的机会采用云，而在传统的模型中，这些企业可能根本没有机会。最近我正在写一篇关于我们的云模型的博文，在里面我还写到了如何将云分解成一组服务层，每一层都为企业带来特定的价值。服务的层级包括基础设施层，平台层和应用服务层（当然还有其他层，通常我们谈得最多的是这三层）。为合适的任务选择正确的云服务，也不要担心创建自己的云。并非所有云都一样——这就出现了关于私有云和公共云（或超云）的差别的问题。我们很多客户关心安全以及应用程序和数据的隔离——因此我们的多数客户是在防火墙之后开始（私有）云的。

David：性能和安全首当其冲。在我们将国家机密放入云中之前，云提供者还有一段路要走，但是基于高速发展，我认为“足够好”的安全系统会出现的。性能是另一个问题，主要由于网络和系统的延时，但这个问题因云提供者的不同可能有很大差别。还有一个限制是互操作性，后面我们会谈到。企业应该要有这样的认识：不是所有的应用程序都适合云计算。

Geva：有好几个问题需要仔细考虑，比如安全和可移植性，但是在这里我要强调的是可扩展性。在一个专属的静态的环境下的一些假设和最佳实践在云环境里可能不会想当然成立了。或者，即使能用，他们也不会让你能享受到如 Amazon EC2 带来的云的优势。

比如，我们说在云的世界里人们可以按需扩展或收缩资源，并且只为使用的资源买单。这固然很好，但是很多应用程序的架构本身并不允许方便地实现在需要时资源时扩展到多个服务器，而不需要时收缩——并且要在几分钟内完成。好在越来越多最佳实践和产品在解决这方面的问题，我一直在和几个公司合作并评估解决这个问题的几种可选方案。

再比如，HighScalability.com 网站报道了我的朋友们在 Rocketier 如何在 10 台商业服务器上搭建一个系统，这个系统可以每天处理 10 亿条事件。这个系统还可以根据需要随时扩展和收缩，因此可以利用到按使用付费系统（如 EC2）的优势。这个系统通过使用一个分块的内存数据网格作为记录系统，这种做法与我们大多熟悉的集中式数据库系统不同。

Jamin：在应用程序层，若采用云，针对每一个应用需要考虑一组问题：在给定负载下部署和运行的费用？如何实现针对应用程序和数据管理安全和私有需求？你需要哪种 SLA？可定制化和可配置化程度的需求?

企业关心的重点不仅仅是本公司和其员工的需求，他们越来越多地关心来自他们的消费者以及合作伙伴的技术需求。通过提供跨企业和网络的集成功能，并且提供多设备来访问能力，企业可以获得更高生产力，更忠诚的客户以及更高效的供应链。因为云带来的部署可选性，企业可以灵活的选择将应用部署在本地还是在云中（或者二者都有），从而可以重新思考如何通过灵活性、易用性、安全性以及丰富性提高商业利益。这种方法把 IT 人员解放出来，他们将首要考虑功能性价值，其次是如何部署这些功能的相关技术。企业应该考虑那些既能推进现有资产的投资又能创建一组全新业务的应用场景。

您如何看待提供商锁定以及云平台之间的互操作性？

Jamin：云之间的互操作性只是话题的一部分。拥有多年 IT 投资的公司需要在现有环境和云环境之间架起一座坚实而灵活的桥梁，从而使得本地部署的应用程序和部署在云中的新应用程序能够顺利交互，既推动业务发展，又不需要新增数据集成方面的投资。很多情况下，多数公司关于云互操作性的讨论是对过去本地的互操作性和可移植性的扩展。

在云中，互操作性非常重要。云平台中规定互操作性术语的标准很快会出现，它们使得云-云，云-企业数据中心之间的互操作成为可能。

Microsoft 的 Azure 服务平台是一个通过 Web 寻址、SOAP、XML 以及 REST 等标准定义开放的、灵活的平台；这种做法的目地就是要确保可扩展的编程模型，使单个的服务可以被运行在微软的和非微软的产品族的应用程序和基础设施所用。详细信息参见这里。

Jerry：我祝福任何尝试把锁定策略拿到台面上的提供商。在当前时代，客户要开放的软件和系统——以及由它们带来的可选性和互操作性。现在是一些新生标准日趋成熟的“成型期”，然而，（提供商们）共同的担心已经汇聚。以云基础设置为例，IBM 曾经和工业界联合致力于为客户带来一个基于 Java 的中间件世界，现在我们正努力为我们的客户带来相同的利益。“写一次，到处执行的”仍然是个非常诱人的想法。和 Java 一样，我们正努力让我们的客户将基础设施虚拟化一次，即可在任何地方使用（云和/或 Hyper-Visor）。目前有好几个标准正在为客户带来这个级别的灵活性而努力着，Open Virtual Format （OVF）就是重要的标准之一。实际上，在 IMPACT 2009 大会上，我们发布了一个 WebSphere Application Server 二进制的可选购买方案，包括预安装的，预配置的（包括操作系统），虚拟镜像（使用 OVF 标准）等。一旦客户购买（或升级）到某方案，它就不需要重新安装 WebSphere，他只需要将介质拷贝到 OVF 感知的 hyper-visor，然后服务自动开启并运行。

David：这是一个大问题，因为云提供者已经创建好他们的平台，这些平台大多基于私有架构、API、以及语言等。因此，一旦你的应用程序是基于某个云平台上创建的，你就很难将其移植到本地或者另一个云平台。不过一些进行中的项目致力于形成支持可移植性的标准，但是我们离云计算空间的一组准业界标准的距离还有点远。

Geva：我写了一篇博文，名为谨慎对待尚未成熟的（云标准的）阐述，在这篇博文中我讨论了这个话题。如我所写，最终获得云操作的互操作性，甚至正式标准，是广泛采用云计算的关键所在。然而目前还为时尚早，我们需要小心谨慎，而不是随大流跟风。现在我们甚至对标准所需要解决的问题都没有足够的认识。

不过，一些有意思的发展是可以跟踪的，比如 GoGrid 已经开放 API，并且正在努力让其他供应商采纳这些 API。有一个名为 EUCALYPTUS 开源项目已经实现了 Amazon EC2 和 S3 的一些 API。其他如 Enomaly 等供应商也提供开源的云软件，并开放了云计算互操作性论坛。随着时间的推移，市场将会做出投票，准业界标准将会出现，正式标准也终将形成。

为什么有些人选择私有云而不是公共云？

Geva：一些大的公司或企业已经为 IT 基础设施投入了大量资金，其中很多公司在为特定业

务或企业的需求建设高效数据总线方面已经拥有相当深入的技术和经验。为什么 Amazon 决定进入 IaaS（基础设施即服务）的业务呢？因为他们已经意识到自己在运行极其高效的 Web 基础设施方面做得很好。当然，也有很多公司拥有这样的技术（并且是与他们特定的业务和工业相关的技术）。他们对规范遵守可能还存在一些疑虑（如医疗行业的 HIPPA 约束），这种疑虑阻碍了他们使用公共云，或者是因为他们的 SLA 太苛刻以至于其他云提供者目前无法支持。

也许，更加有意义的问题是私有云与普通的数据中心之间的区别。首先要记住，“私有云”并非只能在公司本地数据中心上运行。在公共环境下创建虚拟私有云的技术已存在，如 CohesiveFT 的 VPN-Cubed 就是一款应用该技术的好产品。

对于某些大公司，即使运行一个私有的，内部的（本地的）云也是有意义的。这些企业可以采用云的基本原理，如多租户架构、虚拟化、自动化和自助服务等，从而可以获得三个方面的好处：1）高效的硬件利用率 2）高效的 IT 运维 3）快速创新周期。

Jerry：在 IBM，很多客户都对私有云的前景感到振奋。事实上，我们的云策略始于“人工降雨”的思想。当我向客户传递“在企业内部‘播种’云”这种思想的时候，我经常使用这个词，这个思想公共云服务的场合也是有意义的。我们的客户正在构建私有云，而我们的工作重点是辅助他们创建、使其自动化，优化以及管理那些云。基于两个方面的考虑，很多客户选择私有云的路线，安全（应用程序和数据的安全）是一方面，另一方面是他们已经在基础设施建设上投入了很多资金和人力，他们希望能够将这部分投资利用起来。我们看到客户创建私有云并在云上运行很多饶有趣味的任务。用于测试和开发的云越来越流行。事实上，我们现在正使用一个用于测试的私有云（利用我们最近引入的技术 WebSphere Cloudburst）来测试我们的 WebSphere 应用服务器（WAS）。这种做法让我们共享资源，简单来说是（基于安全和可复用的模式）创建了一个测试环境，减少了日常的安装和卸载环境的运维工作。

Jamin：如果私有还是公共的问题指得是本地用户管理的还是提供商托管的的问题的话，那么上述关于实施限制的讨论已经覆盖了这方面的内容。这个问题主要是关于控制的，企业在实施或客户化他们的基础设施、平台和应用程序时，仍然需要权衡他们在过去考虑的那些控制因素。

David：安全、性能和控制是关键问题。所以，你可以在防火墙后面架设可共享的基础设施，享受云计算带来的高效的资源共享，按需获取资源等好处。这将是云计算最大的成长空间，因为，至少在最初，很多公司并不希望放弃对他们的核心 IT 基础设施的控制权。

有的云提供商提供基础设施（Amazon），有的提供平台（Google），应用架构师通常如何选

择？

David：依赖于业级架构和应用程序的需要。Amazon 允许你通过资源的类别（存储，数据库等）来消费，而 Google 则提供全套的应用程序开发和部署平台。因此，首先你要理解你的需求是什么，定义需要解决的业务问题，然后再选择合适的解决方案——本地部署还是由云提供，或者取二者兼有。

Geva：在大多数情况下，平台即服务(PaaS)的提供者会很大程度上限制使用场景和技术栈。这样的限制带来的好处是，开发和运行时管理变得非常简单。如果使用场景和技术栈的限制正好满足你的应用程序需求，那么就是一个好的匹配。需要指明的是这些平台区别也很大。因为 Google 指定 Python 作为编程语言，以及特定的数据模型以及多线程技术等，所以 Google App Engine 上应用程序可能不需很大改动就可以轻松地移植到其他平台。Force.com 就是一个完全私有化的 PaaS，它甚至包含私有的编程语言 Apax。你的应用程序如果不重写，那么几乎不可能在别处运行。还有一种如 Heroku 这样的 PaaS，尽管它需要你按照特定的方式为应用程序写一些元素，才能使之运行在 Heroku 的平台上，但这些做法都是“最佳实践”，而且你的程序不会绑死在 Heroku 上。

Amazon Web 服务和 GoGrid 这样的 IaaS（基设即服务）提供供原始 IT 资源，如计算能力、内存和存储。这种方式更灵活，但需要更多的管理工作；它不提供高层服务如开箱即用可扩张性（包括自动扩展）和容错机制。然而，随着时间的推移，IaaS 提供者开始提供附加的上层服务，基础设施和平台服务之间的界限模糊了。此外，一些第三方提供者如 RightScale 和 GigaSpaces 也致力于缩减 IaaS 和 PaaS 之间的差别。

Jerry：基础设施服务的强项是允许用户在云中运行他们现有的应用程序和中间件。大多数基础设施提供商提供通用的基础设施支持，包括操作系统以及基本中间件栈。用户提供其他的——应用程序和应用运行的“技术细节”（扩展，安全，执行）。比如，在 WebSphere 环境里，基于 10 多年在 WebSphere 部署方面支持客户的经验，我们的基础设施服务已吸收这些“技术细节”。我们引入 Patterns 的概念，模式是一种虚拟的部署方案，它将安全、高可用性以及性能等方面的最佳实践都融入在一起。此外，我们已经在公共云中提供我们的软件镜像，像 Amazon 一样，为我们的用户提供一个非常低的门槛，让他们使用 IBM 的中间件产品（包括开发，测试和其他）。

通过平台服务——巧妙划分的基础设施——你需要专注的是应用程序以及服务。很多云平台都具备一些特定的编程模型（这又回到了绑定的问题上去了:-)），这些编程模型让应用程序更具可预见性，因此，平台可以更加自动地进行扩展，安全以及执行。对于新建的应用程序来说，这是很好的。然而在移植现有的应用程序时，通常需要开发人员重写应用。这些平台

经常以公共云的方式提供——因此，伴随着应用程序的大改——使得它（平台）对于某些公司来很具吸引力，而对另外一些公司却不然。显然，对于 Java 提供商来说是一个明显的机会，通过为云创建基于 Java 的支持，给用户带来云平台之间的可移植性（包括公共云和私有云）。

我认为还有一些混合模型也是相当有趣的。比如，在 2009 IMPACT 大会上，我们介绍了 BPM BlueWorks，这是一个为业务领导提供的托管服务。BlueWorks 应用程序为业务专家提供一个门户，让他们学习，分享和相互协作来创建业务战略和流程。一旦业务资产被创建出来，它可以导出到本地的云基础设施（可能以一种基于标准的 BPMN 2.0 文档的形式）。这样就产生了一种“在公共云中开发资产，在私有基础设施服务中运行”的混合模型。对于这些混合模型来说，关键是要在公共云和私有云之间搭建一个安全的通道。稍后对此作更多介绍。

Jamin：对于那些要移植现有应用的架构师来说，考虑到要对现有应用程序做必要的修改，他们可能认为向云移植的好处有限。在某些情况下，优势立刻就能体现，用户很快就可以感受到将云实例外包出去带来的好处。然而，还有些用户已经抱怨：与原有的私有数据中心相比，在虚拟化的云中维护遗留系统的开销更大。

对于一些架构师，他们可能正在设计全新的应用场景，或者已经预见到可以利用云平台的优势来建立新应用。在一个平面型的云平台如微软的 Windos Azure 上写应用或移植应用比在传统的基于实例的基础设上做要更高效。然而，没有一个通用公式来计算最佳的效果，架构师们要分析各种备选方案，基于需求和现有应用的架构，现有的开发技能以及其他因素综合作出正确的决定。

客户如何加强规范遵守？

Jamin：为加强规范遵守，客户必须和云提供者一起合作，共同来确保当工作被外包时，所有的数据和私有规则（包括法律的和私有协议）都可以得到保护和加强。与云提供者之间的协议不仅要包括可靠性 SLA，还要包括遵守性 SLA，这样才能确保客户的数据在任何地方都合法地使用和管理，从而最大限度满足客户需求。

David：这里有两个层面：技术和程序。在技术上，要确保创建合适数量的系统管理规则，包括对核心资源的访问、审计路径以及管理所需的其他约束。选择一个外部机构来为你审计，确保所有事务的合法性。在程序上，应保留所有的法律文档以及将整个过程文档化。

Geva：在某些情况下，在云环境中维护规范遵守和在传统上托管，收集和管理本地数据中心并无区别。从某些方面来看，客户完全依赖于云提供者，必须预先明确提供商采取哪些步骤来确保规范遵守。（例如，Amazon 发布了它的安全实践的白皮书）。一些人已经成为这方

面的专家，如 PCI 规范的设计者。

Jerry：云计算的一个非常有意思的话题是将云虚拟化成基础设施，平台和应用服务等三个层面，从而可插入控制点。IBM 主要在云架构的各个层面为用户提供管理云的使用的能力。客户利用这个能力获取他们的系统被如何使用的密切信息，用户还可以利用云作为控制点，来加强策略和规范的遵守。例如，我们的 WebSphere Cloudburst 产品可以产生详细的报表，该报表包括谁在使用云，如何使用等信息。管理员可以使用这些数据生成用于管理和控制的个性化报表。另外一个例子，我们看到一些用户使用我们的 WebSphere DataPower SOA 设备和 Service Registry 来发现服务和（细粒度地）控制对这些服务的访问，在公用云和私有云中都有看到。DataPower 可以为私有云创建公共通道，也可以为公共云提供。一个安全的网关可以帮你挡住云的各种威胁，同时提供一个审计双向应用服务通信的控制点。对客户来说，不论是否有实际的规范遵守的需求，在云周围架设一个双向的（web）应用程序防火墙也是一种更加聪明（和安全）的设计方法。

专家组成员介绍

Jerry Cuomo：IBM 院士，副总裁和 WebSphere 产品线 CTO，他是 WebSphere 的创始人之一，在 IBM 研究院和软件部共工作了 20 年之久。Jerry 是高性能事务系统，中间件设备，和企业云计算以及 Web2.0 领域的开拓者。

David Linthicum：（Dave）是国际知名的云计算和面向服务的架构（SOA）专家。在他的职业生涯中，他形成和发展了很多现代分布式计算（包括 EAI，B2B 应用集成和 SOA 等现在广为使用的技术和方法）背后的理念。最近 10 年，Dave 专注云计算方面的技术和战略，曾与多位云计算的创始人合作。Dave 的业界经验包括多家成功软件公司的 CTO 和 CEO，多家财富 500 强公司的高层管理职位。

Dave 专注云计算的最佳实践和实际业务价值，包括技术在企业需求环境中的实际价值。他的专长是使用熟悉的工具和通俗的词语描述业务在哪里可以和云计算结合。

Geva Perry ：Geva 最近 5 年都在 GigaSpace Technologies 工作，在那里，他曾担任过多个管理职务。他目前是云计算总经理，在这个职位上，Geva 负责 GigaSpaces 全球所有云计算相关的市场化活动，包括战略和定位，产品市场化以及战略联盟等。在加入 GigaSpaces 之前，他是 SeeRun 的 COO，负责实时业务活动监控软件的开发。Geva 在 Hebrew 大学拿到学士学位，获得 Columbia 新闻系研究生院的硕士学位和 Columbia 商学院的 MBA。

Jamin Spitzer：是微软 Developer &Partner Evangelism (DPE)在 Redmond 的平台战略执行官，加入微软已有 5 年光景。在加入微软之前，他在 J.D Edwards 的业务应用站和 PeopleSoft 工

作多年。

原文链接：http://www.infoq.com/cn/articles/Virtual-Panel-Cloud-Computing

相关内容：

- 认识云计算
- OpenSocial的分析与实现
- 云计算七问七答
- Joyent：别样的云计算平台
- REST在IT/Cloud管理中的角色——API的对比

[推荐文章]

backlog是一种关键的产物和实践，还是一种浪费？

作者 Amr Elssamadisy 译者 金毅

Dave West 为 InfoQ 撰文，提议到，如果将软件开发看做理论构建（theory building）的话，backlog 是非常重要的。这篇文章已经有一些忠实的拥趸，他们对此完全赞同。随后，在 leanagile Yahoo!小组上又掀起了一场讨论。不论这篇文章观点正确与否，关键在于，我们能否从进一步的研习中获益。

最终，InfoQ 联络了几个业界大师：Mary Poppendieck、Ron Jeffries、Jeff Patton、David West、Steve Freeman 和 Jason Yip，来进一步探究这个议题，请他们就以下几个要点做了阐述：

1. 你认为 backlog 的作用是什么？
2. 你会怎样定义一个 backlog？
3. Backlog 什么时候/情况下会被认为没用？
4. Backlog 什么时候/情况下是必不可少的？
5. 有没有一些我该提及却没有提及的问题呢？如果有，请提出并做答。

这里是他们的回答：（注：Mary Poppendieck 的答案在一篇名为《重新定义 Backlog》的随笔中，并被附在本文末尾。）

1) 你认为 backlog 的作用是什么？

Steve Freeman：
为了让团队认识到他们当前的状态和未来可能的走向。而这些应该得到团队的足够重视。

Ron Jeffries：

可能有很多作用。我想说两个，因为它们价值不同：

1．对将来有一天可能要做的事做备忘。

2．跟相关人员交流他们提出的请求的状态。

Jeff Patton：

对我来说，backlog有两个作用：首先它能帮助我理解我正在做的产品的“形状”，即它是什么，它将能做什么，它为谁服务。我从宏观角度来理解这个“形状”，或者说是看看故事的全貌。这使我与Dave的描述产生了共鸣，他引用了Alistair的书中由Nauer提出的“理论构建”一说。没错，确实应该是这样，我们就该这么做。我把backlog看作“故事地图”，以此来展开探讨理论：

http：//www.agileproductdesign.com/blog/the_new_backlog.html

但是，我们迟早要决定做什么，理想情况下，一旦需要马上就决定。随后stories就变成了backlog，包含已计划好的工作，但仅包含我已经明确选定的工作。

至少，story和backlog是两回事。当用backlog一词来做些理论探讨的时候，这一名字就不那么传神达意了。正是由于这个糟糕的名字，而且需要一些技巧和理解才能用它来表述理论，我认为大多数人做不到。

David West：

Backlog的用途起源于user story的用途。User story曾被设计为一种用户（或客户）授权机制。它曾经用很草根的形式写过，用对话形式写过，用来从系统中捕获过一些用户想要的东西，即用非正规的方法 描述用户想要的行为或职责。story也可以用来分解并且关注工作，因此story是可估计的、在一个迭代中可完成的，如果定义得太大就要重构等等。

一个story卡片可以提醒用户和开发人员，他们之间的对话已经开始并将继续，直到完全实现了这个详细的story。在此过程中也会产生其它一些或短或长的提醒方式：像白板上的模型、测试、编写的代码。这一切，包括story卡片，没有任 何其它目的，它们切实地提醒着：用户想要什么，用户怎样理解她想要的东西，开发人员怎样解释了她想要的东西。对用户和开发人员间难以共识和沟通的内容，它们可以作为一种有形的、可追溯的证据。

一组User story卡片是一个独立对话（meta-conversation）的有形证据，这时，它们大部分是用户专有的。每个独立story(meta- story)关系到优先级（这个story在业务中有多重要？）、共用性（我是不是唯一想要这个的人？）、情节和特征演化（虽从不同的视角出发，你的X目 标和我的Y目标却完全相同）、可选择性、变动性等。每个独立对话会根据每个story完成所得到的反馈信息做改动，并且在软件项目的整个过程中持续进行。

综上所述，对正在进行的会话来说，一组story卡片相当于有形的备忘录：用户想要什 么？它能怎样帮助他们和他们的组织？如果你把一组story卡片看成一个backlog，你将置身于危险的境地——你在语义上搞错了，即把那组卡片强制看 成一些别的东西。但如果你坚持要那样认为，那么：对一个多人参与，并发进行的软

件系统来说，backlog也成了有形的备忘录。

Jason Yip：

假设Sprint Backlog可以为当前Sprint提供一致认可的、详尽清晰的工作目标，再假设产品Backlog可以指示产品的潜在功能。那么，它可以用来表达整个产品的愿景，但典型的简单backlog在这里完全不起作用。使用故事图、停车场图、流程图等工具来放置backlog项，相比之下会更好。

2) 你会怎样定义一个 backlog？

Steve Freeman：

1) 一个可以帮助团队理解他们和客户之间关系的思考工具。

2) 一个我们所能理解的需要提供哪些特性的粗略列表。

Ron Jeffries：

backlog 是一个已计划的功能项的清单，针对“backlog 所有者”当前的优先级做排序。

Jeff Patton：

重申一下，我痛恨 backlog 这个名字。

它们根本就是两回事。

组织级 story 列表是我们对产品理论高层次认识的详细描述。

优先级 story 列表是已计划工作的 backlog，但不包括那些未被正式提交的 story。

就是说，你必须知道组织 backlog 和为=backlog 区分优先级是不同的。

David West：

“产品 backlog”是一组 story，用于表示目前我们对用户想从软件系统里面获得什么的最好的理解，当然这会变化的。“迭代 backlog”是我们（开发组）按照迭代计划的，在一定时间内打算完成的一组会话。迭代 backlog 确实 承担了待完成清单（to-do）的作用，是一种工作清单。每个 story 会话都会产生一些额外提醒，来促使我们继续前进，并且保证我们获得反馈以便持续追 踪状态。除了它原来的使命（即作为正在进行的动态会话的有形标记），一个迭代 backlog 承担着更多的职责（即支持迭代管理和跟踪）。

backlog **不是**一组需求。

backlog 也**不是**未完成工作的详细目录。

Jason Yip：

一个潜在要做的工作的队列。如果 backlog 被排好序，结合了任务、工艺流程、总体产品愿景等上下文，而且随着项目进行逐步细化，那么 backlog 会更有效。

3) Backlog 什么时候/情况下会被认为没用？

Steve Freeman：

再声明一下，当维护它所花的成本大于它带来的价值的时候。

比如说，在做维护/支持相关工作的时候，快速做出决定比去挖掘一大堆可能的需求来得更有效。如果需求经常有的大变化，更是这样。

Ron Jeffries：

我认为 backlog 不会完全是浪费。它就是一个清单。由于它是按优先级排序的，也不需要频繁的关注很多项，就看看前面几项就可以了。

也就是说，处理于一个较长的 backlog 无疑要比较短的 backlog 花更多的时间，可见，放弃那些优先级不是很高的是合情合理的。我很期待在 backlog 方面有种智能曲线，可以提示我 80%“自然发生”的项可能不怎么重要，就算舍弃掉影响也很小。

要是 backlog 没提供什么价值，它们就是种浪费，有投入无产出的活动。要是公布出来的 backlog 在误导大家，让项目干系人幻想他们的想法正被实施中，而实际上他们等啊等，可能永远等下去，那真的是太不厚道了。

我为什么要说他们可能永远等下去呢？因为 backlog 上没有被选来做的项比已选的价值低。也就是说，新出现的主意非常有可能也比待选项更有价值，它们会被插在待选项之前……就这样周而复始。

其实在 backlog 靠前的位置确实有根曲线的，如果你在这根线之后，你的主意将永远不会被实施了。在这种情况下，更好的做法是告诉大家可能不够走运，得完全放弃那些排在后面的。这也给提出主意的人相应地留出了更多决策空间，例如用其它的方式提出主意。

Jeff Patton：

当一份工作真的很有价值去做，你又非常深入地理解它，而且已经很详尽地安排了未来要做的一切的时候，你就不需要 backlog 了。

David West：

在两种状况下，（项目或迭代）backlog 变得没用。

一，每当它变成“文物”。我的意思是说，当 story 已经沦落到跟我们交流讨论时做的 记号一样。一组 story 只能反应项目初期我们对系统的理解。这时的 story 变成了一个“石碑”——一个无法修改的、纯语法的、没上下文的陈述，所以没 什么语义。于是那些 story 成了历史文档证明我们早先的无知程度，此外什么都不是。

二，当一组 story（backlog）被强制用于有关“管理”或“产品跟踪”的目的 时。敏捷确实是一个新范式（老套的说法）。敏捷是种与众不同的文化。习惯的力量、懒惰的力量以及来自我们身边的形式主义文化的压力，一直力求将敏捷和敏捷 实践打回原点，回到由结构性的/工程的/理性的/科学的开发和“泰勒主义”管理的文化中。

Jason Yip：

backlog 总是没用的。

4) Backlog 什么时候/情况下是必不可少的？

Steve Freeman：

当你需要了解还有多少剩下没做的时候，比如当你不得不跟诸如市场部或者实施人员需要长期交流的时候。

我认为我们值得去区分一下不同层次的 backlog，就像 Scrum 里面区分产品 backlog 和 Sprint backlog 那样。可能你要求产品人员只关注前面 5 项(因为前 5 项包含更广)有点不合情理——但这些可能就是开发团队需要做的。如果我们真的想要比较汽 车生产，就好比比较丰田产品开发体系和生产体系的不同。前者有着更多的浪费，因为它依赖于设计的按时完工。

Ron Jeffries：

必不可少的？这世上可能几乎没有必不可少的东西。

只有当 Backlog 表达真正要去做什么时才有价值。一旦远离真实，它的价值也会降低。建议比较短的 backlog 更好。

Backlog 有助于考虑下一步可能做什么、怎么选择以及精化所要实现的东西。

Backlog 能够作为列表提醒人们不要忘记一些事情。很多人在列表上可能有很多事情，但实际需要去做的并没有那么多，但这多少能给大家带来方便。

Jeff Patton：

对 Backlog 的理解是不可少的。如果你把 backlog 作为一种辅助理解工具，那么它总是必不可少的。如果你不知道怎么使用它，那么就不是必须的。

David West：

如果你正在致力于敏捷开发，当你需要来表述你对所从事领域的深入理解的时候，当你想让你的软件能帮助支持真正的客户，在真实世界里，真正地工作的时候。

Jason Yip：

我们需要知道下一步做什么，不管它能不能给出更大的产品目标。对产品可能的样子有个总体的广泛认识是很有帮助的。如果你们不选择 Scrum 这种形式的 backlog，就得选点别的。

5) 有没有一些我该提及却没有提及的问题呢？如果有，请提出并做答。

Ron Jeffries：

做**backlog**，好的和坏的方式各是什么？

将 backlog 存放在电脑里听上去很吸引人，但却是危险的。电脑能保存数量庞大的信息，而结果几乎不可避免的是：太多东西被太详细地写了下来。如果不管不顾，这些就是垃圾。如果关注了，它们大多浪费时间。

用索引卡（index card）记录 backlog 有很多优点：

你不能带着很多索引卡（index card）到处走，因此backlog的大小有先天限制。

个人或小组就能给一堆索引卡（index card）按优先级排序，只要把它们放在一张桌子上摆来摆去就行了。过程简单，有效，参与度又高。

你不能在索引卡（index card）上写很多，所以那些不需要的细节就被去掉了。

撕掉一张卡，用新的替代很容易。这使得更新非常容易。

如果需要，我还能继续说出不少优点。每个人都“知道”为什么这些东西“应该”被保存在电脑里。但更多的时候，用简单的方法几乎各个方面都更好。

Jeff Patton：

你怎样用**backlog**来理解产品的“理论”？有别的什么能够帮助理解产品理论么？

（Jeff 没做答。）

David West：

为什么每个人都不对？（这假设，当然我是对的。）或者，换句话说，“为什么我们大家对**story**、**backlog**还有敏捷的理解如此大相径庭？”

简言之：我们是在计算机科学与软件工程专业接受了教育的技术人员，而那里除了“正规的 ”东西什么都不尊重，那里的工作文化根本就看不起非正规的东西，以及那些所谓“神秘”的东西。大体上说来我们都已经不再理会 Brooks 关于“概念构建” 这一关键特性的慷慨陈词以及他对数学、理学、方法论和模型缺陷的评论。我们都忽视了 Naur 提出的理论构建的思想、理论的不可言喻性、方法和文档的缺陷 性。我们大家都喜欢抽象数据类型而忽略了对象思想。我们大家都遗忘了 Alexander 提出的建筑的永恒之道（Timeless Way of Building）以及无名的质（QWAN），而喜欢把那些因为不能理解设计艺术而引出的问题的解决方案写成文档。我们大家都在用旧的思维模式来生搬硬套 敏捷文化、哲学和价值观。

Mary Poppendieck 在题为“重新定义 backlog”的随笔中，通过重申这个问题，给出了他的答案。

我想由“backlog”这个名词来谈开，据我所知，它是通过Scrum被引入到软件开发界的。我试图避免使用Scrum术语，特别是一些类似“backlog”已经有多种不同用途的单词。我相信是时候再来谈谈精确定义术语的问题了。例如，标准精益概念中，像队列(queues)、知识创造 (knowledge creation)和基于集合的开发(set-based development)。一个“backlog”就能涵盖这三个东西了，而且通常混在一起。

队列(Queues)在这样的环境中最适合拿来讨论：有人希望改进自己的工作，因此不断有一些小的需求给软件公司来做。那么你做的越快，客户就 越快认识到他们要寻找的价值。这种情况下，控制你接受的请求的数量，以你现有的资源在短时期内完成它们，几乎总是最优方案。这原因来自于队列理论 （queuing theory），特别是Little’s 法则：完成需求的平均时间与队列的大小是直接成比例的。

这样，当相对比较小的需求犹如涓涓细流，从客户那里涌来时，你的响应时间将和

你队列的长度也将成正比。如果队列短，你就能很快响应，比较及时地 解决客户提出的需求。如果你总是能够在较短时间内给出响应，并且结果可靠，那么客户会逐渐依赖你的响应时间。如果你的响应时间值得信赖，那么客户们就会在 要结果之前尽可能搞清他们的问题，因为他们知道，即使他们的问题发生变化，提出的请求也不必排等待很久。限制了队列长度，你就要在客户提出需求时诚实地给 他们快速反馈：你会搞定他们的问题，或说不能做。如果你说要做，客户就大概知道这个问题什么时候可以解决掉了。如果你说不能做，客户会认为你很诚实，而为 解决这个问题给些其它条件。他们也能很快明白，需求实现是有限度的。

用这种方式，对客户真诚以待，客户也会信赖团队。团队用坦诚和守时向客户表示尊敬。团队与客户紧密地联系在了一起，并致力于不断给客户带来价值 ——正如精益理论中，客户利益至上。知识创造是精益思想中另一个有巨大价值的东西。因此，知道你将迎来什么，不断跟踪一下你已经考虑过的东西，了解对未来 的种种可能，是非常棒的。每个客户的问题都是一个学习机会，你和你的客户搭档，站在他们的立场上来理解他们所考虑的未来。当客户和开发团队成为最佳拍档 时，他们也会一起花时间来经常看看各自的问题，从做过的各种尝试中提取些经验，对未来市场和技术的趋势保持警觉。

知识创造还有另一个好时机，就是当你在做一个很大的项目时，想要找到路标，即项目前进的大致方向。这是一个很不错的思想。我敢肯定，如果你不多 花点时间在方向操控上，而是一头扎进成堆的小事中，随之而来的只有返工。事实上，只要能从实践中学到足够的东西，也算只是创造了。知识总是好东西。有很多 手段可以追踪知识。精益（Lean）理论中所推荐的一个工具是A3文档（A3是一种纸的规格，大约11×17英寸，不过美国不用这种规格标准）。做法是将 所有关于某个特定话题的想法都汇总到一张A3文档上。用一个A6纸做索引，因为在软件开发中，大家发现这个大小的纸张用来捕获客户价值信息最适合。其实是 A3还是A6根本无关紧要，重点是你要让潜台词明朗化，用有效的方式捕获它们，这样你就不必再反复研究就能做出更有理有据的决定了。这是精益（Lean） 理论的精髓所在。

基于集合的开发(set-based development)是一种以不同意见作为基础来进行开发的方式。当遇到一旦做出觉得很难挽回这种棘手情况，它可以被用来获取更多的知识。意思就是说 你提出很多不同的选择，随后根据知识创造所获得的信息，来决定哪个最佳。你可以提出2到3套理论解释到底怎么工作的，随后进行些实验，来比较结果。如果结 果跟理论不一致，你需要去改进你的理论。如果结果跟理论完全吻合，那就太完美了。

通过实验来改进认知的想法对于工程改进也十分有效。比如，理解理想中队列长度的最好方法是建立一个理论，解释队列长度将怎么影响一些重要数据点的，随后去尝试不用的队列长度，看看到底怎么影响那些数据点的。

读者可以看出，我们对 backlog 还远没有达成一致。但是我们还是发现了一些重要的共同点：

1) 对你将要做的事情有个整体认识非常重要。

2) 明白你现在状态也是很重要。

3) 从（1）和（2）中可知，关键是要维护和并让每个人都知道 backlog。

4) 如果一个列表太长或者内容陈旧，它的浪费远大于价值本身。

原文链接：http://www.infoq.com/cn/articles/backlog-panel

相关内容：

- ScrumMaster项目面谈诀窍
- 讨论：Scrum回顾会议怎么开最有效？
- 评论：Scrum联盟会改变它的本性吗？
- 案例分析：荷兰铁路公司的分布式Scrum开发
- Rachel Davies谈Generic Agile

[推荐文章]

反对if行动 / 反对for行动

作者 Amr Elssamadisy 译者 金毅

意大利XP倡导者Francesco Cirillo为他著名的“反对if行动”创建了一个网站，一时吸引了不少支持者。Francesco认为：

if 所带来的问题主要在于建立了模块（方法、对象、组件等）之间的依赖，也增加了代码路径的分支（这会降低代码的可读性）。

Francesco 举了一个示例，认为如下的代码：

```
// Bond class
double calculateValue() {
       if(_type == BTP)  {
        return calculateBTPValue();
       } else if(_type == BOT) {
                  return calculateBOTValue();
              } else {
                  return calculateEUBValue();
              }
}
```

应该使用多态将显式的 if 或 switch 语句消除：

```
// Bond class
double calculateValue() {
    _bondProfile.calculate();
}
// AbstractBondProfile class
abstract double calculate();

// classe BTPBondProfile >> AbstractBondProfile
double calculate() {
    ...
}
```

```
// classe BOTBondProfile >> AbstractBondProfile
double calculate() {
    ...
}
// classe EUBondProfile >> AbstractBondProfile
double calculate() {
    ...
}
```

Francesco 认为这样做的好处在于

当我们需要增加新的 bond 类型时，只需创建一个新的类型来保存这部分独立逻辑即可。

创建抽象类或接口并非是改进的唯一方式，我们的目的是将程序变得更灵活、更易于交流、更容易测试、并且随时拥抱变化。

对于“反对if行动”，Matteo Vaccari**补充了几点做法**：

	修改前	**修改后**

直接使用布尔运算结果	`if (foo) {`
	`  if (bar) {`
	`  return true;`
	`  }`
	`}`
	`if (baz) {`
	`  return true;`
	`} else {`
	`  return false;`
	`}`
		`return (foo && bar || baz);`

使用辅助函数	`if (x > y)`
	`  return x;`
	`return y;`
		`return max(x, y);`

灵活使用 0 的作用	`int arraySum(int[] array) {`
	`  if (array.length == 0) {`
	`  return 0;`
		`int arraySum(int[] array) {`
		`  int sum = 0;`
		`  for (int i=0; i <`

	}
	int sum = array[0];
	for (int i=1; i < array.length; i++) {
	sum += array[i];
	}
	return sum;
	}
		array.length; i++) {
		sum += array[i];
		}
		return sum;
		}

不过社区中对此也有不同看法，有人讽刺道：

哦，这里我发现了一个问题。尊敬的客户，您前一个顾问使用了“if”语句，让我把这个函数替换为 80 个新类，这样显得更加敏捷一些。

Don't Repeat Yourself 看上去更敏捷一些。更好的做法是：等到出现第 3 遍重复代码的时候才去重构（不过也别等太久了）。这样可以避免很多无所谓的代码，也可以让程序员有更灵活的选择余地。

也有人为“反对 if 行动”进行了补充：

我想这个行动并不是说“删除所有的 if 代码”，而是将类型判断逻辑使用多态进行替换。如果把它称为“反对类型字段行动”会更合适一些。

“反对if行动”也在征集“if-free”的代码示例，您也可以在那里提交代码片断。事实上，国内社区也提出了不少消除繁琐if语句的案例，如：

Jeffrey Zhao编写了一个简单的类库，使.NET中的枚举类型可以携带一些信息，以便统一地获取数据或进行运算。

木野狐使用了State模式，简化了不同状态下，Web界面中相应组件的样式切换方式。

Tristan G设计了一组流畅的API，使开发人员能够方便地编写实体验证规则。

您在这方面是否也有独特的经验呢？不妨也一起分享出来吧。

Francesco Cirillo于不久前发起了“反对if行动”，受此影响，Matthew Podwysocki也用这种方式提出了自己的声明，即“反对for行动”。

Matthew Poswysocki 生活在华盛顿特区，作为微软的高级咨询师，维护或参与了诸多社区活

动（如 DC ALT.NET 讨论组），并致力于推广各种敏捷实践。这次他提出，在代码中应该尽量使用和构建可以进行组合的函数，而不是显式的循环语句（包括 for、foreach 和 while）。

Matthew 认为，通过循环来实现的功能往往可以分为以下三种情况：

查询（映射、过滤等等）

聚合（求和、计数等等）

进行一些有副作用（Side Effect）的操作（读取文件、发送消息等等）

Matthew 看来，使用 for 循环来处理“查询”和“聚合”时，最大的问题在于将关注点放在了如何做（How）而不是做什么（What）。他举了一个例子，“找出 100 以内所有质数”，并给出了一个实现：

```
var numbers = Enumerable.Range(1, 100);
var output = new List<int>();

foreach(var number in numbers)
    if(IsPrime(number)) output.Add(number);
```

Matthew 认为：

这里的问题在于我们很难将现有逻辑与另一个操作进行组合，因为这里的实现涉及了“怎么做”。我们应该使用.NET 2.0 以上版本中的泛型及延迟加载的特性进行函数式的构造。这样可以带来“声明式（declarative）”的感觉，而关注点就只有“做什么”了：

```
var primes = Enumerable.Range(1, 100)
    .Where(x => IsPrime(x));
```

Matthew提出，应该尽量避免在一个循环中进行多种操作，这样会为代码的可读性和维护性带来负面影响。而在Martin Fowler的**Refactoring站点**中，拆分循环内部逻辑的重构方式被命名为“**Split Loop**”。Matthew认为较好的方式是将逻辑进行组合，例如求出“100 以内质数的数量”便可以这样实现：

```
var primesCount = Enumerable.Range(1, 100)
    .Where(x => IsPrime(x))
    .Count();
```

对于产生副作用的情况，Matthew引用了Eric Lippert**对于“为什么IEnumerable<T>没有ForEach扩展方法”的回应**。Eric认为，引入IEnumerable<T>的ForEach扩展方法事实上带来了副作用，而违背了IEnumerable<T>的设计初衷。此外，ForEach还会形成闭包，可能会造成一些难以发现的引用问题。

Matthew 并没有赞同这种说法，不过它对这个看法表示理解。他认为，如果是使用 C#进行编程，使用 foreach 来遍历一个 IEnumerable 没有太大问题。不过在 F#中，最好还是使用 iter 或 iteri 方法进行遍历。关于这点，他使用 F#交互命令行进行了演示：

```
> let flip f y x = f x y
- [1..10]
-   |> List.map((*) 2)
-   |> List.filter(flip (%) 3 >> (=) 0)
-   |> List.iter(printfn "%d");;
6
12
18
> [1..10]
-   |> List.map((*) 2)
-   |> List.filter(flip (%) 3 >> (=) 0)
-   |> List.iteri(printfn "%d\t%d");;
0       6
1       12
2       18
```

社区中有人**认为**，这个行动的目的是希望在面向对象编程环境中融入部分函数式编程的理念。C# 3.0 引入了Lambda表达式，将**高阶函数在.NET中的应用**切实地推广开来。同时，其他平台也在进行着类似的改变。例如**最近颇受好评的Scala语言**也引入了函数式编程特性。

你对 for 的使用有何看法，并且对函数式编程的看法如何？

原文链接：

http://www.infoq.com/cn/news/2009/07/anti-if-campaign

http://www.infoq.com/cn/news/2009/07/anti-for-campaign

相关内容：

- 任何人都可合法实现C#与CLI规范
- VB.NET路在何方？
- VB和C#的自动实现属性
- 平台多样化：Gavin Grover的Groovy之路
- 深入探察相等操作符

[新品推荐]

JUnit 4.7 的新特性：Rule

作者 Geoffrey Wiseman 译者 张龙

JUnit 4.7 RC 版已经发布了，该版本具有一个重要的新特性：Rule。本质上，Rule 是 JUnit 的另一种扩展机制，可在每次测试中为 JUnit 增加新功能。规则可以替换掉大多数使用旧版本 JUnit 所编写的客户化运行器，同时添加新的功能。

原文链接：http://www.infoq.com/cn/news/2009/07/junit-4.7-rules

Rails 2.3.3 发布、Rails 3.0 与Merb现状

作者 Werner Schuster 译者 张龙

近日 Rails 2.3.3 发布了。除了修复一些 bug 外，该版本还增加了一些新特性，如 ActiveRecord touch 功能，同时修改了一些 JSON 相关的 API。除此以外，看看 Rails 3 和 Merb 1.1 中都有什么新玩意吧。

原文链接：http://www.infoq.com/cn/news/2009/07/rails233-state-of-rails3-merb

FlexMonkey 1.0 发布了

作者 Jon Rose 译者 张龙

Gorilla Logic 公司于近日发布了 FlexMonkey 1.0。FlexMonkey 是个面向 Flex 和 AIR 应用的开源测试工具，它可以对 Flex 用户界面功能进行捕获、回放及验证。

原文链接：http://www.infoq.com/cn/news/2009/07/flex-monkey-1.0-released

Oracle和BEA完成产品集成，融合中间件 11g发布

作者 霍泰稳

近日 Oracle 对外发布了融合中间件 11g，这是 Oracle 去年完成 收购 BEA 之后发布的第一款中间件集成产品，也被业界认为是两家公司完成产品集成的标志。融合中间件 11g 具有全面、集成、可热插拔等特性，包括 SOA 套 件、WebLogic 套件、WebCenter 套件和身份管理等子产品。

原文链接：http://www.infoq.com/cn/news/2009/07/oracle-fusion-middleware-11g

Android开始支持脚本语言Python、Lua及Beanshell，未来还将支持Ruby

作者 Werner Schuster 译者 张龙

Android Scripting Environment（ASE）项目为 Android 增加了脚本支持。像 Lua 及 Python 等语言的本地版本可以使用脚本来调用通过 JSON-RPC 公开的 Android API。未来还将支持 Ruby 及基于 JVM 的语言。

原文链接：http://www.infoq.com/cn/news/2009/07/android-scripting

Oracle Coherence 3.5 带来增强的WebLogic支持和万亿级数据网格

作者 Scott Delap 译者 崔康

Oracle 发布了 Coherence 3.5，其支持万亿级数据网格和改善集群健康和稳定的服务维护。

原文链接：http://www.infoq.com/cn/news/2009/07/coherence-35

微软发布了分布式计算技术Dryad和DryadLINQ的学术版

作者 赵劼

Dryad 和 DryadLINQ 是微软的分布式计算技术，前者提供了分布式计算的基础架构，而后者提供了普通程序员也可以轻易使用的高级语言接口。不久前微软发布了它们的学术版供下载试用。

原文链接：http://www.infoq.com/cn/news/2009/07/Dryad-Academic-Release-CTP

PowerShell 2.0 RTM即将发布

作者 赵劼

PowerShell 2.0 RTM 将随 Windows 7 和 Windows Server 2008 R2 一同发布，可惜 PowerShell 团队在博客中公开到，可以在 Windows XP SP3、Server 2003 SP2、Vista SP1 及 Server 2008 中使用的 PowerShell 2.0 需要在“今后几个月”才能发布。

原文链接：http://www.infoq.com/cn/news/2009/07/Get-Ready-to-PowerShell-2-RTM

[架构师大家谈]

架构师修炼之道

架构师是一个神秘而又神圣的名词，作为软件开发领域的设计师，架构师承载着太多的责任和挑战。对于一个程序员或者工程师来说，架构师就像是一个目标，一条道路，抑或是一座山峰。如何能够成为一名合格的架构师？架构师应该具备何种素质？而架构师又是如何做到持续不断的成长和提高的呢？带着这些问题，我们请到了五位 InfoQ 中文站的编辑，同时也是各领域出色的架构师或者咨询师，来谈谈他们心中的“架构师修炼之道”。他们是：

- 宋玮：InfoQ 中文站 Java 社区首席编辑
- 王瑜珩：InfoQ 中文站.NET 社区编辑，ThoughtWorks 咨询师
- 赵劼：InfoQ 中文站.NET 社区编辑，微软最有价值专家，现任某创业团队架构师
- 张龙：InfoQ 中文站 Java 社区编辑
- 李明：InfoQ 中文站 Ruby 社区首席编辑，现任某通信公司架构师

1) 在你的心目中，架构师意味着什么？

张龙：架构师是一个项目组的灵魂人物，他决定着整个系统的技术选型、整体架构以及模块划分，同时还可能担当与领导层的沟通角色，从某种意义上来说，架构师在很大程度上决定着项目的成败与否，正所谓火车跑得快，全靠车头带。

王瑜珩：对我来说，架构师一直是一个很迷惑人的词，似乎每个人的理解都多少有些不一样。我认为架构师更像是一个投资家，需要权衡各方面的利益和风险，反复思量，最后给出一个现实可行的方案，争取用最小的风险获得最大的利益。

李明：我觉得，架构师不仅仅是一个头衔，更是一份责任。所谓“在其位，谋其政”，我倒是觉得架构师更像是父母，而系统和项目则如同子女一般，需要架构师耐心的呵护和培养。完成一个项目，绝不是架构师工作的全部。通过代码重构和架构改造，让这个项目如同有了生命一般逐渐成长起来，这才是架构师最终的目标。

2) 架构师应该具备何种技能或者素质？

宋玮：架构师应该具备一定的业务知识和业务分析能力，能够准确地把握需求。要有较强的学习能力，对于新出现的技术、框架和工具，能够快速掌握。扎实的基本功，能够把握住技术方向。良好的沟通能力，能够清楚地表达自己的意图和想法。

李明：代码能力绝对是很必要的。我见过太多只懂得画图的架构师了，“识大体不拘小节”这个说法，在架构师身上并不适用。作为一名架构师，在系统的性能和可扩展性上，要有足够的敏感性，既要充分利用现有资源，又要为长远做好打算。另外，对业务的理解是很多技术架构师所忽视的地方，只要彻底了解业务需求，技术才能派得上用场。

赵劼：在我看来，一个合格的架构师需要具备开放的眼光，各种平台、系统、项目随手拈来皆可组合，唯一的目标则是针对合适的环境选择合适的做法，这显然需要在成本和质量之间进行权衡。作为一个架构师，应该具有很好的“弹性”，在真正的环境中，很少会遇见与过去一模一样的情况，因此也需要架构师能够大胆尝试，灵活应对，使用踏实而严谨的做法来进行推测。一个架构师也必须有着足够的沟通和交流能力，把自己的想法使用合适的方式告诉别人，并且根据别人的反馈进行不断调整自己的观点。没有东西是永远正确的，但是一个人往往会倾向于自己的结论，而作为一个合格的架构师，需要有能力认识到自己存在的缺陷，使用各种方式进行弥补。

王瑜珩：架构师需要高瞻远瞩着眼未来，从外部功能与内部架构两方面来考虑可能面临的变化。诚如周爱民所说，架构师要在开发 1.0 版的时候就想到 2.0、3.0，甚至更远。然而在考虑未来的同时，也不能脱离现在，不能由于对未来的设想而大幅提高现在的开发成本，万一未来并没有到来，所有对未来的投资都将毫无意义。因此架构师需要平衡投资与风险之间的关系，以适当的风险来获得最大的利益。架构师需要有良好的沟通能力，才能将自己的想法展示给开发团队中的每个人，确保整个团队对系统架构的理解是一致的。架构师不能脱离实际，设计一个无法实现或成本很高的架构。同时对于一个实际的团队来说，也需要了解团队中成员的能力，知道何种架构可为，何种架构不可为。

张龙：很多优秀的架构师都是从一个优秀的开发人员转变过来的，但优秀的开发人员未见得都能成为合格的架构师。与架构师相比，开发人员所需担当的任务相对狭隘的多，其最大的目标就是编写出精良的代码、做好充分的测试以及撰写高质量的文档等；而架构师所要面对的则相对宽泛得多，除了过硬的技术之外，还需要有良好的表达能力，同时还要有宏观的驾驭整个系统的能力。

3) 架构师需要不断修炼和提高的是什么？

宋玮：扩充知识面，学习了解众多技术及框架的特点和适用范围。了解非功能特性的相关技术和方法，包括可用性、容错性、可扩展性、可伸缩性等等；了解系统安全性方面的技术和框架以及系统性能和状态监测方面的知识及工具。除了技术方面，还架构师还应扩展自身的业务知识，不断提高业务分析能力。想要做到持续不断的学习，保持对各种技术、框架、产品的浓厚兴趣是必不可少的，另外还要掌握他们各自的优缺点及相应的适用场景。学习途径和方式则是多种多样的，但是有一点是可以肯定的，架构师们相互间经常交流对成长是非常有益的。InfoQ 的《架构师》就提供了一个很好的交流平台，通过大家的广泛参与，相信《架构师》能够在分享经验、促进交流方面起到积极的作用。

李明：我觉得这个问题可以从两个方面去谈。一方面，架构师要紧跟技术潮流，了解技术的发展和趋势，利用新技术、新方法来提升团队的生产力，将技术转化为收益。这就要求了架构师平时要多关注所在领域或社区最新的新闻和报道，最简单有效的方式莫过于每天都看看 InfoQ 中文站了。而另一方面，架构师要培养自己的专业领域。虽然从技术的层面上说，很多解决方案放之四海而皆准。但是，从业务的角度来说，很多行业的解决方案放到另外一个行业中，未必行得通。这就要求了架构师必须对所属行业的业务十分了解才行，这也是一个平日里需要修炼的地方。

赵劼：架构师的学习过程是痛苦也是美好的，一个合格的架构师应该可以从学习过程中找到，至少是追求“架构之美”，把架构当作一种“艺术”来对待，并且可以把这种美给传播出去，带领技术团队把这种美变成产品，让更多人体会得到。

张龙：架构师之路是崎岖的，充满了荆棘与挑战，但这却是无数开发者的梦想。架构师是多项技能与素质的综合体，每一位以此为目标的开发者都需要在平日的工作中不断提升自己，在这里我衷心的祝愿架构师这个梦想能照进每一位有心人的现实。

对于合格的架构师应该具备的素质和技能方面，张龙还给出的详细的列表：

有人曾说过，20 几岁的编程天才好找，但 30 多岁的优秀架构师难寻。架构师何其难？除了敏锐的洞察力之外，我认为一个好的架构师必须具备如下几方面的素质：

- 过硬的技术能力：有人说架构师就不需要编写代码，只需设计整体架构就行了。但我认为这是很片面的，试想一个人如果长时间不写代码，他还能具备持续的技术敏感度么？当然了，这里所说的写代码并非一般开发人员的行为，而是让自己保持住对代码的感觉。还有人说架构师不一定是技术高手，这一点我很同意，但他一定是个优秀的开发者。
- 良好的沟通能力：这一点尤为重要，因为架构师需要与项目组的开发人员以及领导

层不断交换意见，向对方传递自己的设计意图与思想，没有良好的表达与沟通能力是很容易出现问题的。这一点在沟通方式并非母语的企业中尤为明显。

- 良好的软件工程素质：虽说架构师不是项目经理，但我认为他需要对软件开发过程有清晰明确的认识，这里的开发过程是个泛指，也许是 RUP，也许是 XP，是什么无所谓，但这种工程素质是每个优秀架构师必备的品格之一。
- 宽广的知识领域：架构师的眼界一定要开阔，绝对不能局限于眼前的小范围事务，否则极易出现“鼠目寸光”的后果。这就需要架构师不断学习，这里的学习既包括技术上的，同时也包括业务上的以及沟通上的。
- 领域知识：架构师务必对自己所从事的业务领域有深刻的认识，他未必要成为业务专家，但他一定要对业务知识有深刻的理解。很难想象经常从事金融领域项目的架构师能轻松设计好电信领域的项目架构。知识需要积累，业务也是这样的。
- 处理系统非功能性需求的本领：架构师尤其需要对系统的性能、容错、并发等非功能性需求方面有独到的认识与解决办法。一个项目到了后期，往往都是这些问题成为整个项目的瓶颈，这时架构师就要发挥其优势了。

通过上面诸位的发言，我们可以看出，架构师得是一个“全才”，不但在技术上和业务上要做到“两手抓，两手都要硬”，更是得需要持续不断的修炼和学习，才能成为一名合格的架构师。虽然这是一条充满挑战的道路，但也同样充满了乐趣与收获，正所谓“无限风光在险峰”，读者朋友们，你们做好准备了吗？

[推荐编辑]

Ruby社区编辑 杨晨

很高兴我能够在这期《架构师》上留下只言片语。我是来自于 Ruby 社区的杨晨，负责 Ruby 相关新闻的本地化工作。

去年秋天，长期混迹于 IT 界兼山寨专业吉他手的 Nasi 学长，问我是不是愿意加入一个叫 InfoQ 的网站，并且派了两篇文章给我试译一下，这就是我和 InfoQ 的第一次亲密接触。再后来，我就成了一名 InfoQ 的编辑。

刚刚加入 InfoQ 大家庭的时候，InfoQ 的前沿气息宛如一股清风，给我带来了业界最新的资讯。在我看来，InfoQ 紧密地将技术社区与开发者联系起来，并且通过挖掘报道有深度有意思的事件，帮助开发者从中了解到最新的业界动态，把握业界发展的脉搏。最令我感到满足的是，在 InfoQ 的讲座和文章中，开发者还能学习到工业界的先进经验技术。当然，如此优秀的网站怎么能够不推动开发者们线下的交流呢？看看 InfoQ 每年一度的 QCon 吧。在这个会议中，汇集了世界的顶级开发人员，和中国最优秀的开发者们一起坐而论道。在 InfoQ 的时光虽然尚显短暂，但是我却在其中收获良多。不仅仅是自身翻译水平的提高，更重要的是对自己的视野也裨益甚多。每一天，我都能了解到业界的最新趋势，从一点一滴的讯息中，发现自身的不足和缺陷，并且及时进行弥补。InfoQ 有很多留给我深刻印象的东西，例如 QCon 的成功举办，全球英豪聚首北京，节节攀升的人气。而最令我感到兴奋的，就是这本名为《架构师》杂志的诞生。

现在开发者网站遍地皆是，但是，InfoQ 正是凭借其独到的优势：高质量的原创文章，本地化新闻和资源类型（不仅有新闻、讲座、视频，还有最近推出的《架构师》这本优秀电子杂志）成为这类网站中的佼佼者。而新近推出的这本电子杂志，就是 InfoQ 编辑部集 InfoQ 资源中大成，博观约取，厚积薄发，浓缩于数十页之中，力求将最快，最优秀的信息分享给大家，在技术和业务上给每一位开发者带来帮助，一道成长，一同为美好的明天而奋斗。

[封面植物]

金花茶

金花茶是一种古老的植物，极为罕见，分布极其狭窄，全世界90%的野生金花茶仅分布于我国广西防城港市十万大山的兰山支脉一带，生长于海拔 700 米以下， 以海拔 200-500 米之间的范围较常见，垂直分布的下限为海拔 20 米左右。与银杉、桫椤、珙桐等珍贵“植物活化石”齐名，是我国八种国家一 级保护植物之一，属《濒危野生动植物种国际贸易公约》附录Ⅱ中的植物种，国外称之为神奇的东方魔茶，被誉为“植物界大熊猫”、“茶族皇后”。

金花茶喜温暖湿润气候，喜欢排水良好的酸性土壤，苗期喜荫蔽，进入花期后，颇喜透射阳光。对土壤要求不严，微酸性至中性均土壤中可生长。耐瘠薄，也喜肥。耐涝力强。

形态

常绿灌木或小乔木，高 2-- 6m，树皮灰白色，平滑。叶互生，宽披针形至长椭圆形。花单生叶腋或近顶生，花金黄色，开放时呈杯状、壶状或碗状，径 3— 3．5cm；花瓣 9--11 枚，阔卵形至倒卵形或矩圆形，肉质，具蜡质光泽；花期 11 月至翌年 3 月。蒴果三角状扁球形，黄绿色或紫褐色；果期 10-12 月。

应用

金花茶花色金黄，多数种具蜡质光泽，晶莹可爱，花型有杯状、壶状、碗状和盘状等，形态多样，秀丽雅致，在山茶 类群中，被誉为“茶族皇后”。亚热带地区可植于常绿阔叶树群下或植荫棚中，供以观赏。


1kg.org
多背一公斤
爱自然 | 更爱孩子

架构师 8月号

每月8日出版

本期主编：李明

总编辑：霍泰稳

总编助理：刘申

编辑：宋玮 朱永光 郑柯 胡键 郭晓刚

读者反馈：editors@cn.infoq.com

投稿：editors@cn.infoq.com

交流群组：
http://groups.google.com/group/infoqchina

商务合作：sales@cn.infoq.com 13911020445

本期主编：李明（Nasi），InfoQ 中文站 Ruby 社区首席编辑。

毕业于东北大学，曾供职于百度网页搜索部，从事分布式网络爬虫及其国际化的研发工作。目前在某通信公司任系统架构师，进行高性能大规模分布式系统的设计与开发。擅长搜索引擎技术，关注开源社区发展，注重敏捷开发实践。参与翻译《Website Optimization》和《Algorithms in a Nutshell》等多部技术图书，《Google API 大全》的合著者之一。同时他还是一位吉他手，兴趣广泛，乐于分享。可以在 http://twitter.com/nasiless 找到他。

InfoQ中文站
www.infoq.com/cn

innobook
创造·共享·传播

所有内容版权均属C4Media Inc.所有，未经许可不得转载。