时刻关注企业软件开发领域的变化与创新

架构师

10月 ARCHITECT

Stu谈云计算

一个100%的Ruby云方案

企业架构适用于小企业吗？

SOA与云计算有多大关联？

演进架构中的领域驱动设计

我来选架构

InfoQ中文站
www.infoq.com/cn

每月8号出版

篇首语 | Editor's letter

保持一颗好学之心

初见这个题目，许多人可能会对自己相当满意："我还是很好学的"。真的是这样吗？个人之见，有不少人其实并不像他们想象的那么好学，尤其是那些有了一定经验，在某些方面有些过人之处的"聪明"之人，随着技术水平和自信心的积累，往往丢失了好学之心，而他们自己却浑然不知。对此，去年 Infoq 刊登的一片新闻《初心，聆听之术》谈到了如何保持学习心态的问题。或许有人要说，好学与否，真的就那么重要吗？工作如期、按质完成不就得了。好学不好学，无非就是其自身修炼的心态发生了变化，难以进步而已。确实，对于一般的程序员来说，仅此而已。但是如果此人担负着重要的角色，比如架构师，或领导者，那就得另当别论了。他的固执和自负，不但会使其丢掉好学之心，而且难以接受别人的建议，甚至造成与其他人沟通不畅，这样，整个项目和团队都将受到影响。另外，如果没有充分利用好一些新的、成熟的成果或共享资源，凡事都自行研究，这无异于闭门造车，代价高昂而难以保证质量，在竞争中将自己摆在不利的位置之上。毕竟，IT 行业的发展已经不是前些年那种"闭塞"的局面了，互联网极大的推动了各种信息及制品的流通。在互联网上，信息的传递与共享速度是以前所无法比拟的，共享资源也是异常丰富。如果我们能够充分利用这些信息以及共享资源，可以节约大量宝贵时间，从而更快更好的达成目标。这就更要求我们始终保持一颗好学之心，充分摄取各种有用资源及信息，将其应用于自身的工作任务当中，以便又快又好的完成任务。同时通过运用成熟先进的框架和工具，也能大大提高项目的质量。

当然，对于新技术、新方法、新框架的运用也要根据实际需求而定，不能走另一种极端，而应充分了解其特性，仔细筛选，逐步采用。这也要求我们有非常强的学习能力，能够快速准确掌握这些这些技术、方法、工具、框架的特点、适用范围及场合。因此，我们要时刻保持一种初学者的好学之心："那是一种不包含预测、评判、和偏见的纯朴心态。是探索、观察并发现事情本质的一种表现。"而不应有一颗"专家之心"："认为自己知道一切，因而关闭了学习之门"。

本期主编：宋玮

目　录

[新品推荐]

[架构师大家谈]

[封面植物]

人物专访 | Interview

Stu Charlton论云计算

在2008年旧金山QCon的这次采访中，Stu Charlton谈论的问题包括云计算，传统托管与云托管的区别，平台云与基础设施云，云计算如何改变软件开发/部署，厂商锁定，如何来推进到云计算，云计算带给中小企业的好处，以及云计算的工具支持。

InfoQ：大家好，我是Ryan Slobojan，和我在一起的是Stu Charlton。Stu，让我们来谈谈云计算。传统的托管供应商与云托管有什么不同呢？

John：传统的托管供应商通常不是向你提供...多租户的机器，你可以在他们预先安装和设置好的机器上运行软件，并且支持你运行各种web应用或者数据库等等，就是允许你在托管环境里专门分配一台机器，通常按月支付月租费，从而获得相应的SLA。

而另一方面，较之传统托管，云计算在实际预估需求所用的耗时上有很大的差别。首先，由于实际上使用自服务的接口或API来提供机器，启动和回收一台机器的引导时间缩短到了以分秒计的程度。第二，如果你的需求有着太多的变量，你可以使用你的云来灵活分配。比如说，由你在云服务商那里获得的限额而定，你需要的时候，可能在一周内将服务器从30台增加到3000台，而在传统的难以拓展的托管模型中，你不得不给他们一再地通知："嘿，我可能会需要3000台服务器"，而他们会回答你："请给我们两个月"。

一个主要的不同之处在于云服务供应商更愿意做他们自己的容量分析与需求趋势，当然是在一定的限额范围内。这很重要——运转机器与停止机器的能力，以及完全基于使用的收费。我认为这里面还有更多的价值。话说回来，实际上你会发现，如果将在传统的托管方式下运行一台专用机一个月与在云中部署纯粹从成本进行比较，许多的云服务——至少现在——的确显得有昂贵一些。现在有了"外包"云服务，不是指仅仅关于外包的云，实际上是关于你是否在自己的数据中心里拥有它以及进行费用退回。而是你真的能够通过它来缩短时间，比如，我想部署什么，不管是在自己的数据中心或者是其它的第三方共享托管或者是Amazon EC2之类的。我形容...它是如此之快——这就是云服务不同与其它所有事物的地方。

InfoQ：你对于不同的云计算产品有什么样的看法？

John：对于云计算是什么有着广泛而多样的定义，这已成为不变的主题了...所有这些技术热

潮都有着不同的趋向...当市场与它们联系起来的时候你发现那里充斥着多种多样的定义。我认为云计算有两种不同的极端。一种是平台云计算，有人为你创建好了一个完整的系统供你使用——这是一个黑盒子，我常这样开玩笑。这就像是，你插入你的代码，它非常面向开发者和终端用户，就像你看到的 SalesForce 或是 Google App Engine 那样。

对于一些特定的应用点或者独立的应用，这非常合适，你可以做 CRM，做帐，故障通知等等之类的工作，也许你需要在后台做一些集成，但这都是一些自包含的事情，这样一来，支持服务于下载的应用交换并且提供一个可工作的开发平台就显得很有意义。但这将高度依赖于所暴露出来的这些能力，而 API 及其暴露方式也将让你受到限制。

但它们大多有自己私有的 SQL 版本，事实上还不能算是 SQL，因为这些版本只拥有标准 SQL 的一小部分特征。这种云的特点是“你所有的基础都是我的”，它对于很多这类的应用程序来说将十分有用，但是，我不确定它对于那些大公司而言用处有多大，特别是那些有很多已有应用，且并不打算重写它们的大公司。

这就将我们带到了云计算的另一个极端，那就是面向基础设施的云计算，或者说，基础设施即服务(infrastructure-as-a-service)。我不确定你怎么读它，IAAS 什么的，没关系...这包括了...其典型代表是 Amazon 的 Elastic Compute Cloud，它背后的意思非常接近系统管理员或者说是数据中心的运营官，在这种情况下你所做的是自动的分配存储网络容量与计算能力，通常伴随着一些事先定义好的软件包。它的一个好处就是能够有效的支持你现在所使用的东西，你可以将你到现在所做的大部分工作，包括现有架构，软件等等，都迁移上去。

这可以类比到一个供应链的生态系统——软件行业仍然存在，并非所有的东西都是...软件并非仅仅是服务，仍然存在真正意义上的软件，你可以分发它们并且把它们安装在云上。像“架构师们不断寻找处理问题的最佳方案”这样的生态系统仍然存在。话说回来，当你工作于基础设施云之上的时候，你不会得到像平台云服务那样的开箱即用的伸缩性，因为你需要设计自己的系统以让它可以达到你需要的伸缩性。

有人说：“云计算就意味着无限的伸缩性”，这可能是一种谬误，因只有你将系统设计成这样，它才会有这种能力。你实际上会看到很多分析师出于这一目的声称平台云服务最终会赢得市场，因为这种黑箱的方式让你不必有太多考虑，而它自己会像变魔法般的实现伸缩。对于这一点我认为：“当然很好，只不过我还有巨额的按照特定方式设计的遗留系统的投资”，因此这里就说到了企业的真正好处在于它能够真正的降低交付周期以及降低收费，因为就算是与许多本地场所的数据中心相比，这种伸缩的经济效益比如 Amazon 也是非常有优势的，他们是这一低端企业的主导者，并正在积极扩大空间。

InfoQ：你觉得云计算将怎样改变软件的开发与部署？

John :云计算最重要的一点是...有句老话说得好，"当你解决了问题 1 ,问题 2 就被提上日程"。现在实际分配后台的数据中心资源已变得十分容易了，我们将拥有更多的这些资源。以前可能是 100 台服务器的，现在可以是 1000 台服务器了。有这样一种说法，我们将会降低利用率，所有这些都将被合并起来，然而...我也同意，虚拟化以及其它的一些技术可以帮助云计算做到这一点，但如果你给了某人实现某种东西的能力，他们可能会利用它或滥用它，因此我们需要管理的东西将比以前多得多。

"IT 服务管理"有着许多传统的挑战，正如这一术语所表达的，可能是配置管理，而主要是处理需求管理；这又是另一个不同的世界，它来自于运维行业——他们来自于英国的商贸部办公室，创建了 ITIL（IT 基础设施库）。这虽然看似官僚，但却是考虑缜密的管理 IT 的方式。

当你进入到云计算这样的方式的时候，这些问题被提到了一个十分重要的层面，因为我们有这么多需要管理的东西，尽管以前它们是以一种随意的方式，或是非正式的方式，或者是取决于你的组织的某种非常官僚的方式而存在。我认为正是因为挑战的存在才使得大家对它如此关心，因此，我们过去在软件上所做的许多工作，有许多...特定点解决方案(point solution)来管理复杂性。如果你是一个开发者，在构建系统时你通常需要组合一些依赖管理器，比如 Maven 或者 Ruby 社区的 Capistrano 又或者是 Ant 等等——这些社区都是很不错的，但它们不能处理像"好的，我已经把软件打包好了，它们各有各的设置，有一些可能在配置文件里，有一些可能在 SNMP，有一些可能在 JMX 里"这样的总体情况。关于如何设置这些有着一堆的标准，而小型的特定点解决方案也有很多。我的组织里正在致力于的一件事就是尝试创造一种统一的形式以超媒体的方式来描述它们 ,这样的话我们就有能力让你的软件以这样的方式来描述，可以处理所有这些麻烦的问题，并且独立于你想要分配的数据中心或者云，不管是你自己的或者是场外的公共云。

我认为这里有一个隐秘的酱汁理论（sauce theory）——Google 的隐秘酱汁理论——云计算所做的重要的事情之一就是消除 IT，并且减少对 IT 的依赖，因为一些神奇的架构会使 IT 变得更可管理。实际上我认为将要发生的事情完全相反——我们得把我们以前对于 IT 所有的想法完全打包起来并且抛到脑后，这将会是一个大事件，并且软件将会因此而改变。

InfoQ：我们如何既能使用云计算与此同时又能避免与厂商的绑定呢？

John：这是一个很好的问题，现在这个答案很不好说，因为就目前而言，在某种程度上你不能有效地达到这个目标。每一个云服务都有它自己的私有元素。就像我之前提到的那样，平

台云计算有着各种不同程度的私有性，尽管其中有一部分云服务将在一定程度上基于标准的基础设施作为了它的一部分。如果你看看 Google 的话，他们提供给了你一个 Python 引擎，你可以在你的 Web 应用里面使用一些标准的 Python 框架比如 Django 等等，这非常棒，但他们的底层数据库层，是对他们的 BigTable 技术的一种封装，是以他们自己的 SQL 语言来做的，这就局限了你能做的事情，举例来讲，如果你想要把你的应用移植到本地场所的服务之上的话，实际上你无法做到。这种模型有一定的局限，尽管我相信随着时间的发展它们会有所改进。而如果是基础设施的云服务呢，其真正的基础设施是可以良好移植的，就像 linux 机器或者 OpenSolaris 或者 Windows，或者任何你使用的机器。

从这种意义上讲所谓绑定就少得多了，但目前而言启动机器的 API 仍然可能是私有的，因为在这一领域还没有标准，但这是好事情——我的意思是，这还是一个新兴的领域，如果你愿意的话，这些小公司需要时间来发展不同的管理云服务的方式。我认为，目前而言真正要保证的是要分离构建软件的方法与提供资源的方法，这样至少在所谓的锁定方面你可以保持一定的模块化：资源提供模块，管理模块，还是软件本身呢？我经常向我们的客户或潜在客户提出的另一问题是，“就算是在这些基础设施的云服务上，到底要我如何改变应用？” 我这样问的意思是，应用的改变不会太多，但是他们往往具有局限性，因为他们所做的事情较之传统的数据中心而言是基于一系列不同的假设或前提上。

这里有一个关于灾备的极好的例子——如果让我举例的话...如果我有一台容错的，或者说是高可用，或者是集群的机器，通常失效是发生在应用层上，而这层通常有一个负载平衡器或者是智能代理等等这样的工具，它将会处理这样的问题：“如果这台机器宕掉了我就会找到另一台机器” 的情况，但接着你就会遇到灾备的问题，“好了，现在这台机器再次恢复工作了，我要将它重新加入集群中”。如果是传统的数据中心，通常的做法是替换物理机器。因此你要到数据中心所在的房间，将东西拔出来，换一个新的东西上去（比如一个新的硬盘或者其他），它将会使用它一直使用的 IP 地址和路由信息。在云计算中，因为我们有软件来处理这些，新的机器有可能在数据中心的另外一个地方。

如果我要给它分配 IP 的话——就像 Amazon 的例子一样，他们有这种弹性 IP——这将会需要稍微长一点的时间，因为新 IP 必须要重新路由网络才能处理，“这是我的 IP”，而且要实际指向数据中心的某个区域，原因是内部 IP 是不一样的，而它属于不同的段，不同的交换器，要统一处理。

有没有其他的方法也可以做到呢？我们在处理多租户的环境时有许多细微的差别，因为在这样的环境里你不能做出实际的预测...而好处仍然是，灾难恢复的时间要快很多。所以在某种意义上，我的环境里灾难恢复的平均时间也比更换机器的手动恢复要快得多，但没有热备份

的私有刀片阵列那么快，而是介于两者之间。但这是云计算会带给你锁定的一个方面，因为这是一种不同的思维方式。

另一个方面是他们可能限制你使用某些网络中你已经习惯使用的某些特性——一个经典的例子是，目前在 Amazon 上，你无法使用 UDP 广播或者 IP 多播。而这有时就会带来一些问题，例如，使用会话复制的应用服务器。如果你将应用服务器设置成点对点的——TCP，它可以很好的工作并支持会话复制，但如果你依赖于 IP 多播来做的话，就会遇到问题，因为它不支持这样做。我觉得大多数的限制所带来的架构改变不像所意识到的限制那么多，而且还提供了替代方案，因此它是可解决的，它只是...在本质上它不是一种改写。

InfoQ：我们如何从现有的系统，部分或完全地转向云计算呢？

John：事实上我认为往往会发生相反的事情，接下来我会做出解释。首先，我认为在现实中有这样几个云计算的用例。有这样一种业界的鼓吹："所有的事物都会移到外包的云服务上"，我和我们的 CEO 经常和客户开玩笑说 CIO 的枕头下总会藏着几张 DVD 的软件拷贝，这样他才会安心。没有必要将所有的东西都移到外包于某个地方的云服务之上，但我再次重申云计算并不仅仅是外包而已。

我们来谈谈外包，对比一下"我在本地运行的"与在"因特网的云服务上的"。如果你思考一下企业是如何运作的，他们在某些情况会表现出来那种保守。而他们愿意的例子是...我认为在云中进行测试开发是比较有意义的，因为你处于一种迭代过程之中，不想被别的东西所妨碍，也不想因为基础设施的原因而减缓进展，而且这个阶段也没有质量或 SLA 方面的考虑，因此我同意在这里云服务是一个绝妙的使用用例，特别是当你在做压力测试或者负载测试的时候。

它的妙处在于能让你真正的用 100 台服务器来做有代表性的压力测试，只需数百美元，与之相对的是花费成千上万的美元去租用服务器。所以我认为这样的事情将会普遍发生。特别是售前，我们对企业软件的概念验证所做的许多工作都花费了很长的时间——很多个星期——并且存在很多问题...问题不在于 POC 实际实施的过程，而在于搭建环境的后勤工作。云计算对于助力售前人员开展和运行 POC 的工作上起着很大的作用。

这里有一个术语叫做"云爆"(cloudburst)，意思是"需要将数据中心的能力扩展到一个公共云上以处理高峰时刻或者短期蜂拥的请求"，如零售商在圣诞节可能遇到的情况一样。我非常赞同，我想这就是弹性云真正能够发挥作用的地方，取决于需求的弹性，我们可以增长或者缩减——我想这会表现得非常好。但你回到我开始的关于开发测试的例子，比如压力测试，最终会发生的是，你以云计算开始，因为它很容易也很灵活，但当真正生产的时候我实

际上认为你会出于多种原因将你的应用移到内部，比如你自己的数据中心或者私有云之上，私有云可能运行于某种虚拟基础设施之上，比如 VMWare，Xen 或者 Hyper-V。

现在绑定问题真正变得重要了，比如“行了，我可以肯定我能在某处的云服务之上运行我的应用，但我同时也需要在内部运行它”，也许是出于隐私条款的考虑，萨班斯法案的考虑，或者仅仅让我我感觉更舒服，因为自己管理和运维私有云的成本和运行在别人提供的云上相比，不会多太多。随着时间的推移，现在这些可能发生改变，可能会出现如果不把应用运行于某个外包的云服务之上就会显得非常被动的情况。但我想至少在未来五年，云计算对于企业来讲是非常实际的，但将会是以一种“我由云计算开始，再移回内部”的方式，与向外移出的方式相反，除非在某种情况下你要处理 “云爆”(cloudburst)的情况。

InfoQ：在现在的经济情况下，你认为云计算如何帮助中小型企业与大型企业展开竞争？

John：这是一个很好的问题。我认为云计算所带来的一个巨大变化是运营开支。之前是资本开支的项目现在变成了运营开支，而且预投资的量也少很多。它的负面影响就是许多大型的企业更愿意做资本投入，因为随着时间的流逝它可以分期的收回成本，而运营开支就完全是开支了。

另一方面，小公司，特别是没有太多资本的公司，肯定愿意利用这种快速启动和即用即付的运营开支模型的能力，因为这有助于他们的运营风格。我想比较有意思的是静观经济条件如何影响到云服务的定价。另一值得观看的是云服务提供者是如何合并的，在我看来最后剩下的将是几家大的竞争者—— Amazon，Google，微软，还有一些其它的竞争者会冒出来——IBM 有他们的蓝云，他们在这方面做了很多。我想他们会生存下来的。

还有很多中等水平的竞争者，并且我也非常尊敬，但我并不认为他们能在这样的条件下存活下来，因为这是一类底层业务…就像是零售业，我认为只有大的零售商店能够成功，而他们都有各自的市场。因此我想在云计算行业最终会有某种清算，最后会发生的是，就像因特网服务提供商的情况一样，如果你想一下 90 年代有许多作坊式的因特网服务提供商——现在某种程度上仍然存在一些，那些比较成功的——但最后它们最终都被收购了，而我们现在的因特网服务提供都来自于大型的电讯企业或者有线媒体公司。

我想在云计算领域会发生同样的事情。我不认为会有一家或两家，我觉得可能会是五家以上，或者比这个少一点。同时会有一些市场划分，我想小一些的云服务供应商想要生存的关键就是要找准市场位置，或者是关注于垂直应用或者是平台即服务（PaaS）型的云计算，而不是以基础设施为中心的云计算，因为这已经是底层的业务了，这样他们或许能获得很大的成功。如果成功，它不仅将影响他们的客户，而且也将会影响到新兴的行业的其他提供者。

一个好处是，软件供应商也开始创建基础设施了，云计算的中间层，以使得云能够更快的分配和管理。我想这就是真正会生存的东西，因为它有助于破坏锁定，它有助于帮助在各个云之间创建一致性，当然形成标准需要花费一些时间，因为这需要由市场来挑选它最倾向于的解决方案。

InfoQ：你认为工具在帮助提升云计算产品的采用方面起到了怎样的支持作用？

John：我想就工具来说话是一个不断发展的问题，但对于各种各样的战略而言却有很多可以谈的。一方面我们可以看到 Google 很好的利用了 Python 工具以及框架还有整个社区，这非常不错。也有一些初创公司 ,比如 EngineYard ,这是 Amazon 投资的公司 ,同时还有我们 Elastra。EngineYard 利用 Rails 做了一些非常有趣的工作，并且他们在创建一个可移植的云计算环境，如果我没记错的话，是将 Ruby 和 Erlang 组合起来创建一种新的体验能够让 Ruby 开发者真正的开始一种可伸缩的可恢复的云服务。再一次，这是定位于平台的层面，关注于开发者，这非常好。

我想很多先行者可能将会是开发者。另一种更有意思的方式是我最近看到的微软在 Azure 上面的一些工作。当然喽，微软有着很多的资源，所以当他们看见 Google 与 Salesforce.com 以及其它的云计算供应商所做的事情之后，他们会说："我们有这样一个平台，它专注于开发者，而微软是一家一直以来都非常关注开发者的公司。"

而你又会发现 Amazon 有着一个非常受欢迎的基础设施服务，于是微软就会说："好的，让我们两手都抓吧"，于是他们开始覆盖整个市场并开始和每个人竞争，但他们所带给开发者的角度是.NET，意思是对.NET 开发者来说，云计算真正所做的是使得部署和分配.NET 应用变得非常的容易了。

还有一件事情要说...长期以来对微软的基础设施比如 BizTalk 或者 SQL Server 以及这一类各种各样的服务器的指责就是他们比较难以安装和操作 ,因为有太多的安装组合并且还要按一定的顺序，还有补丁等等。当你再看他们用 Azure 和.NET 服务开始尝试所做的，就是在云计算上提供更丰富的开发者体验，这不仅能保留他们的.NET 开发者用户群，同时还使得微软的云平台成为了你启动开发的顺理成章的选择，不论最终会部署的环境是什么。

所以我认为他们为大家带来了一个非常有趣的视角，也非常适合他们的世界观。当然，Amazon 允许你运行几乎所有的基础设施，不管是 Java，Ruby，Python 还在现在的.NET，因为你可以在 EC2 上运行 Windows，因而他们更关注于这些机器的管理和存储方面，而不是真正的关注在开发工具上 ,但这从另一方面创建了你在传统的软件行业里面所拥有的一种生态环境——所有在传统的软件行业里面有的，在它们的环境里都有，所以这是一种更为开

放的方式。我想还是会存在所谓的陪审团来评判哪一种方式对于开发者而言最好，而且不只是开发者，还有更多企业的决策者，影响者或者企业架构师也会在这些方案中表现出自己青睐的一面，管理员/运维员可能会更倾向于平台模型，因为这减轻了他们的负担，同时会避开基础设施模型，“哇，我有那么多要处理的东西”，或者可能出现相反的情况：他们可能会想：“我想要控制，我想能够处理，这是我的虚拟实例和存储”而不相信那种神秘的黑盒子。我想所谓的审判团还会存在而且这将是一个非常有趣的过程。

观看完整视频：

http://www.infoq.com/cn/interviews/charlton-cloud-computing-cn

相关内容：

- 认识云计算
- OpenSocial的分析与实现
- SpringSource Cloud Foundry发布了
- SOA与云计算有多大关联
- PHP与微软云计算

热点新闻 | News

JRuby综述：1.4 的新特性、JRubyConf议程及MLVM

作者 Werner Schuster 译者 张龙

JRuby 1.4 RC1 即将发布，我们来看看新版本都有哪些新特性。

JRuby团队成员Nick Sieger为我们概览了JRuby 1.4 的新特性。除了新的YAML支持以及对 1.9 支持的持续改进外，对 1.8.7 的支持工作还在继续。

JRuby 1.4 默认使用的是Ruby 1.8.7。虽然大多数1.8.7的支持工作已经结束了，但Charles Nutter还是解释了目前的外部迭代（增加到了 1.9.x及 1.8.7 中）如此缓慢以及优化如此困难的原因所在

> Ruby 1.8.7 增加了遍历 Enumerator 的能力。咋一看很不错，它仅仅是个外部枚举。然而问题在于这种枚举的复杂性防不胜防。
>
> Ruby 1.8.7 与 1.9 是通过连续（划界连续，比如 Fibers 或 coroutines）来实现外部迭代的，这使得集合遍历的速度相当的慢。由于 JRuby 中具有一个进程中的#each，因此我们不得不在遍历每个元素后暂停一下，而 Enumerator#next 不得不使用*new native thread*来解决这个问题。接下来每个#next 调用通过线程来得到新的结果。

我们希望能看到一个快速的解决方案出来。

JRuby 1.4 在Java集成上也进行了不少改进，这样我们就可以轻松从Ruby代码中访问Java类中的方法。还有其他一些可能的变化，如类型强制的增强，新的方法 java_send（GitHub提交了该方法），它会接受被调用方法的方法签名。

1.4 中具体的特性集还在不断变化，至于会添加哪些 Java 集成还尚不明朗。

JVM的未来版本将会提升动态语言的执行速度。首个绑定了JRuby与MLVM且具有动态特性的构建版也已经面世了。

最后，在宣布首届JRubyConf即将召开不久之后门票就宣布告罄。现在EngineYard 公布了

JRubyConf的最终议程，同时还**增加了不少席位**

> 令人兴奋的是，随着赞助商的不断增加以及 Embassy Suites 酒店的大力支持，我们已经将大会的举办地转移了，新会场的容纳量将是现在的两倍之多。

原文链接：http://www.infoq.com/cn/news/2009/09/jruby-mlvm-14

相关内容：

- JRuby综述：Ruby 1.8.7 支持、Android支持及Bcrypt-ruby
- JRuby综述：JRuby团队转投EngineYard，YAML支持的更新，OSGi的支持，Installer的讨论
- JRuby综述：JRuby发布、ruby2java、JSR 292 进展
- JRuby综述：GitHub:FI、借助于TorqueBox支持的JRuby on JBoss及支持JRuby的EngineYard
- JRuby综述：JRuby 1.3 RC1、Timeout及Nailgun

热点新闻 | News

CloudCrowd——一个 100%的Ruby云方案

作者 Sebastien Auvray 译者 霍泰稳

一年前，纽约时报和ProPublica联合发起了耐特新闻挑战2009 竞赛。DocumentCloud赢得715,500 美元的奖金，任务是构建一个基于文档的应用，使得组织和检查文档变得更加容易。因为考虑到同时要处理好几个资源消耗比较大的任务，DocumentCloud决定完全用Ruby实现自己的云方案：CloudCrowd。

DocumentCloud 主要是使用 CloudCrowd 处理 PDF 文档，但是也可以用来处理下面一些资源消耗大的任务：

- 创建或者伸缩图片；
- 在 PDF 上进行文字抽取，或者 OCR；
- 视频解码；
- 迁移大文件集合或者数据库；
- Web 抓取

在CloudCrowd架构文档中有如下描述：

> CloudCrowd 不是为大量小事务所设计，而是为那些大型、消耗资源比较多的事务所准备的。

CloudCrowd的灵感来自于MapReduce框架。它的架构基于一个使用工作者守护进程进行实际处理的中央服务器。它提供了一个REST-JSON API，以及一个用于检测的Web控制台。CloudCrowd使用Amazon S3用于文件存储，但如果需要的话，也可以使用其他工具进行存储。

InfoQ据此采访了CloudCrowd的作者Jeremy Ashkens。Jeremy是DocumentCloud的新人，也是

Ruby-Processing项目的作者。

InfoQ：和 RightScale Gems 或者 Nanite 相比，CloudCrowd 有什么特点？为什么你们要构建自己的方案？

Jeremy Ashkenas：嗯，其实 CloudCrowd 使用了 RightScale AWS gem。它使用 S3 进行所有结果数据（包括中间和最终数据）的分布式存储。这个变化很大，也是一个很有意思的研究方向，它要通过 RightScale gem 装载更多的 EC2 实例，以支持 CloudCrowd 群集的自动伸缩。中央服务区拥有用于决策自动伸缩需要的所有信息——它了解工作者的数目，它们的地点，它们的状态，以及工作序列的大小等。这只是一个关于为实例装载选择算法，并确保新的实例已有所有需要安装的依赖文件的问题。

和 Nanite 相比较，CloudCrowd 的方向是成为 Ruby 高手容易理解和定制的简单易用工具。它使用的是绝大多数 Ruby 开发者熟悉的技术，比如 ActiveRecord 和标准的 ActiveRecord 数据库，以及用于通讯和伸缩的 HTTP 和 S3 等。所有的这些都很容易调试和研究。最小的 CloudCrowd action 是一个简单的 Ruby 类，它被用来定义一个“进程”方法，执行计算中的并行部分，并保存到 S3。你可以将 CloudCrowd 理解为像 Hadoop 和 Nanite 等企业级系统的替代品。不需要 Erlang，AMQP 或者 RabbitMQ 等。另外，它没说一定可以用来处理极大容量的数据，但大多数情况下应该都可以应用，这主要取决于你处理问题的效率如何。

InfoQ：为什么使用 Ruby？有没有考虑到用其他语言，比如 Erlang？

Jeremy Ashkenas：我之所以使用 Ruby，是因为我们想将 DocumentCloud 其他的部分都粘合在一起。注意，许多示例 action 中的事务不是在 Ruby 里完成的，而是将他们交付给适当的工具。作为一个粘合语言，Ruby 很棒，类如图片处理、PDF 转换、视频编码等都可以轻松地交给 GraphicsMagick、Tesseract 和 FFmpeg 等工具去处理。

原文链接：http://www.infoq.com/cn/news/2009/09/ruby-cloudcrowd

相关内容：

- Joyent：别样的云计算平台
- 认识云计算
- Gregg Pollack和他的Scaling Rails教学视频
- RGen：Ruby建模和代码生成的框架
- 书摘及访谈：Aptana RadRails，一个Rails的集成开发环境

热点新闻 | News

企业架构适用于小型企业吗？

作者 Abel Avram 译者 王丽娟

对那些能负担得起企业架构（EA）所需人力和财力的大企业来说，EA 往往被认为是必定会使用的工具。但也有人对 EA 同样适用于中小企业的断言提出了质疑。

Mike Kavis是一家小型新兴公司（只有 20 人）的企业架构师和咨询师，他坚称他们正在使用EA方法，因为EA对他们来说很有价值：

> 我们首先根据 CEO/创始人最初的商业构想进行了两天的头脑风暴。然后创建了一个用幻灯片演示的业务架构。我们用这个幻灯片向技术人员和非技术人员描述了业务。通过描述行业的整个生态系统，我们迅速分析出了自己的商业模式并进行了完善。我们可以确定在生态系统的哪部分该参与竞争，哪部分该寻求合作。创建出这个可视化的模型之前，我们可是打算在那些不该参与竞争的领域去抢一杯羹。

Kavis 提到，考虑到他们的目标和现有资源，他们没有使用完整的堆栈，只是利用了有意义的规划和业务元素：

> 公司里全职员工、顾问、咨询师总共不到二十人（规模算是小的）。但我们的组织由经验丰富的老兵组成。我们不主张严格的过程。由于是天使投资者【译注】进行的投资，所以预算有限。我们拥有极有才华的工程师和具备多年领域知识积累的业务人员。我们不会在接下来的两三年里退出，也不希望二十年后还是这样（那个时候是持久的中期）。我们所期望的结果是能敏捷而又容易地与合作伙伴、客户合并，成长为一流的组织被世人尊重。所以我分析了这一标准，提出了 EA 的以下组成部分，我认为这些部分有助于我们实现自己的目标：
>
> - 业务架构
> - 三至五年的企业发展路线图

- 组合管理（优先考虑什么时候做哪些事情）
- 各种技术的可视化（基础设施和信息等）

在 Kavais 帖子的评论中，Colin Wheeler 自问自答了三个问题：

什么是企业？

根据《牛津英语大辞典》中的定义，企业是一个人或者是具有共同目标的多个人。

我们何时需要架构？

依据模块或组件了解业务的企业和架构师都需要企业架构。

中小企业应该进行企业架构吗？

这个问题的答案非常简单……如果中小企业做架构有商业价值，那就应该去做。我们必须记住，企业架构可以用非常轻量级的方法来完成，而这些方法能给所有企业带来巨大的价值。

在另一条评论中，Peter Evans-Greenwood 给中小企业建议了一个完成 EA 的轻量级方法：

> 目前 TOGAF 架构框架等都在实践 EA，而中小企业中还没有进行企业架构。EA 只是太昂贵了。一般的 EA 团队仅在工资上就需要一百多万美元，大部分中小企业都筹集不到资金。
>
> 也就是说，中小企业对快速、敏捷的 IT 战略施行方法的需求越来越多。一旦我们搞清楚如何满足这个需求，这个方法也就不像传统的企业架构了。

Joe McKendrick是个作家和独立分析师，他支持Kavis的思想——EA不仅仅适用于大块头：跟大企业相比，EA对中小型企业可能更加重要。但大家一直误认为只有大企业才需要EA。

McKendrick 引用了 IFEAD 的定义，事实上该定义并没有提及大型组织，而是认为 EA 适用于任何企业——EA 被看作是达成目标的一种尝试：

> 企业架构是企业的全面表现。企业架构是个总体计划，它让目标、愿景、战略和治理原则等业务规划的各方面进行协作；让业务术语、组织结构、流程和数据等商业运作的各方面能够协作；让信息系统和数据库等自动化的各方面进行协作。企业架构还是企业的技术基础设施能力，比如计算机、操作系统和网络。

企业咨询师 Brenda Michelson 在发送给 Kavis 的 Twitter 消息中一语中的：问题是，很多人将 EA 的 w/jumbo 框架视同于刚性的管理，而非针对交付能力的价值和实践。

如果 EA 确实和 TOGAF 这样的大型架构框架密切相关，那中小企业就不用处理 EA 了，因为他们根本负担不起 EA。但如果 EA 只是一组实践和价值，就应该建议中小企业利用有意义的内容进行企业架构，即便是那些非常小的企业。

原文链接：http://www.infoq.com/cn/news/2009/09/EA-for-Small-Businesses

相关内容：

- Zapthink：敏捷和企业架构并不矛盾
- JOSH：企业软件组合的新提议
- Enterprise 2.0，一个时髦的新词儿
- 企业架构的价值
- Felix Bachmann谈软件架构评估

热点新闻 | News

构建可伸缩的Web服务

作者 Dilip Krishnan 译者 黄璜

Tom Killalea ,Amazon负责基础设施与分布式系统的技术副总裁在近期的ACM queue上发表了一篇关于构建可伸缩性Web服务的文章。 他概述了构建可伸缩性Web服务的指导原则并举了许多现实世界的实际案例，其核心主题是“只构建你所需要的”。

警惕：过早优化

花费在优化可伸缩性上面的时间和资源不如花费在改进用户体验和吸引流量上。

采纳：他人的成果

他解释到，学习他人在框架与基础设施方面的工作可以减短上市时间，帮助将重点转移到提供客户价值上。

三个重要的进展从不同的方面对降低门槛作出了贡献：迈向 SOA 的趋势(面向服务的架构)，云计算基础设施服务的涌现，以及 ASP.NET，Django，Rails 和 Spring 等等 Web 应用框架的可用性。

警惕：过度优化

他引用了Nicholas Nassim Taleb在高度非概然性不可测事件所产生的重大影响方面所做的工作，并建议使用冗余作为提高可用性的策略；使用冗余作为负载平衡而不仅仅是故障恢复机制这一想法比起对于低概率的可能性事件进行过度优化来说，显然更加有成本效率。

采纳：云

Tom 给出了 Animoto 的例子，这一通过 Amazon.com 的 EC2 基础设施托管的社交 Web 应用是如何随需应变的快速平面伸缩(scale out)的，甚至扩展到 3500 个实例。同样的情况在非云的基础设施里，为了保证尖峰时刻的流量将会花费巨大的成本。

警惕：目标驱动的优化

对于期望的流量进行建模然后构建精确的伸缩性计划以满足这一目标是极具风险的。好的模型难于构建，并且会因为简化或者是降低变因的乐观估计而受到影响。[...]如果你的 Web 服务是成功的，你最终会遇到比目标模型更大的需求——也许不是这个黑色的星期一或者超级碗周末，但有可能是很快以后，在你所没想到的时间范围内。

采纳：扯下翅膀

“除了分析哪部分会第一个出问题以及其原因以外”，Tom 谈到“我们会查看给定的应用或者服务在没有出问题或缺少这部分的情况下会有怎样的表现，并且重新进行测试，以找下一个出问题的部分”。

Tom 这样总结了他的文章“构建一个可伸缩的 Web 服务所面临的最困难的挑战就是在出现故障以及高度的并发访问械的情况下,如何去处理持续性，可靠性，性能以及成本效率之间的折衷。”。

除了Tom的这篇文章，2008 年 10 号还有其它的关于构建可伸缩性Web服务的精彩文章。

原文链接：http://www.infoq.com/cn/news/2009/09/building-scalable-web-services

相关内容：

- REST-*组织
- 即将到来的.NET访问控制服务
- 使用Google PubSubHubbub协议实现即时通知
- 异步REST操作的处理
- 用Web服务代替软件版权保护

热点新闻 | News

斯坦福大学研究显示 :重度采取多任务工作方式的人，其绩效会受到负面影响

作者 Deborah Hartmann Preuss 译者 郑柯

斯坦福大学上个月在Proceedings of the National Academy of Sciences学报上发布了一个研究结果："媒体行业中多工人员（multitasker）的认知控制"，强调指出一个显而易见的事实：从效率的角度考虑同时从事多任务，绝对会影响工作效率。该研究审视了IT领域中一个广为人知、却常常为人忽略的现象：不断出现、正在发生的多任务工作方式。敏捷实施者们这样写：这下可算有理由让团队只开发一个产品了，而且只能有一个产品负责人——将时间花在多个任务之上绝对是效率低下的工作方式。

Wired杂志指出：虽然其他研究重点关注多任务工作方式的眼前效果（比如：办公室里的工作人员经常检查邮件，这种情况下的工作效率），该研究提出了一个不同寻常的问题："要是人们总在使用多任务工作方式会怎么样？" Stanford的研究者Clifford Nass、Anthony Wagner和Eyal Ophir调查了 262 名学生的媒体消费习惯。19 名使用多任务方式最多的学生和 22 名多任务方式最少的学生此后参加了两个电脑测试，集中精力完成手上的测试。

他们使用了一些标准的心理测试指标 ,研究结果显示出 :经常在多个信息流之间转换的学生，他们会在 e-mail、网页、视频、聊天和电话之间来回切换，他们取得的进展远低于不怎么采取多任务方式的学生。更令研究人员惊讶的是：在任务切换能力的测试上，"重度媒体多任务人士" 表现更差，"似乎他们过滤不相干任务的干扰的能力更差。"

该研究再次强调了认知科学家反复提到的事情 :同时处理多个信息流的输入被认为是人类认知能力的问题。

> 对于造成差异的原因——被定位使用多任务方式的人是不是先存在心智上的不健全，还是说多任务方式造成了这种情况——"这是一个需要投入上百万美金才能回答的问题，可是我们没有一百万美金去取得答案。" Ness 这么说。

Wagner 接下来打算用脑部造影方法来研究多任务方式的神经学解释，而 Ness 将会研究儿童

人群在多任务习惯上的发展。

原文链接：http://www.infoq.com/cn/news/2009/09/study-multitasking-performance

相关内容：

- 引领精益&敏捷——一切只关乎人
- 王速瑜谈腾讯敏捷研发方法与平台
- Sprint规划：故事点数 vs. 小时数”
- 回顾之回顾
- 想知道如何解决“切换上下文”问题么？进入“等待打扰”状态吧

热点新闻 | News

别删除数据

作者Abel Avram 译者 郭晓刚

Oren Eini（又名Ayende Rahien）建议开发者尽量避免数据库的软删除操作，读者可能因此认为硬删除是合理的选择。作为对Ayende文章的回应，Udi Dahan强烈建议完全避免数据删除。

所谓软删除主张在表中增加一个 IsDeleted 列以保持数据完整。如果某一行设置了 IsDeleted 标志列，那么这一行就被认为是已删除的。Ayende 觉得这种方法“简单、容易理解、容易实现、容易沟通”，但“往往是错的”。问题在于：

> 删除一行或一个实体几乎总不是简单的事件。它不仅影响模型中的数据，还会影响模型的外观。所以我们才要有外键去确保不会出现“订单行”没有对应的父“订单”的情况。而这个例子只能算是最简单的情况。……
>
> 当采用软删除的时候，不管我们是否情愿，都很容易出现数据受损，比如谁都不在意的一个小调整，就可能使“客户”的“最新订单”指向一条已经软删除的订单。

如果开发者接到的要求就是从数据库中删除数据，要是不建议用软删除，那就只能硬删除了。为了保证数据一致性，开发者除了删除直接有关的数据行，还应该级联地删除相关数据。可 Udi Dahan 提醒读者注意，真实的世界并不是级联的：

> 假设市场部决定从商品目录中删除一样商品，那是不是说所有包含了该商品的旧订单都要一并消失？再级联下去，这些订单对应的所有发票是不是也该删除？这么一步步删下去，我们公司的损益报表是不是应该重做了？
>
> 没天理了。

问题似乎出在对“删除”这词的解读上。Dahan 给出了这样的例子：

> 我说的“删除”其实是指这产品“停售”了。我们以后不再卖这种产品，清掉库存以后不再进货。以后顾客搜索商品或者翻阅目录的时候不会再看见这种商品，但管仓库

> 的人暂时还得继续管理它们。“删除”是个贪方便的说法。

他接着举了一些站在用户角度的正确解读：

> 订单不是被删除的，是被“取消”的。订单取消得太晚，还会产生花费。
>
> 员工不是被删除的，是被“解雇”的（也可能是退休了）。还有相应的补偿金要处理。
>
> 职位不是被删除的，是被“填补”的（或者招聘申请被撤回）。
>
> 在上面这些例子中，我们的着眼点应该放在用户希望完成的任务上，而非发生在某个实体身上的技术动作。几乎在所有的情况下，需要考虑的实体总不止一个。

为了代替 IsDeleted 标志，Dahan 建议用一个代表相关数据状态的字段：有效、停用、取消、弃置等等。用户可以借助这样一个状态字段回顾过去的数据，作为决策的依据。

删除数据除了破坏数据一致性，还有其它负面的后果。Dahan 建议把所有数据都留在数据库里：“别删除。就是别删除。”

原文链接：http://www.infoq.com/cn/news/2009/09/Do-Not-Delete-Data

相关内容：

- J2EE应用下基于AOP的抓取策略实现
- Qizmt：MySpace的开源MapReduce框架
- SQLAzureMW：用于将SQL数据库迁移到SQL Azure上的向导工具
- 数据服务简介
- 如何对企业数据进行架构设计

热点新闻 | News

Project Coin发布语言变化最终列表

作者Charles Humble 译者 张龙

近日 Joseph Darcy 发布了 Project Coin 的最终列表，宣布了即将发布的 JDK 7 中对 Java 语言所做的改进。这些改进是：

1. 自动化的资源管理。提供一种处理资源回收的机制：类似于C# using声明的ARM（Automatic Resource Management）块，但形式上却是基于try声明。这样，using声明只能处理单一资源，而ARM却能在块的范围内处理多种资源。

2. 更好的整型字面值。为数字增加二进制字面值以及下划线分隔符支持以增加可读性，例如：

   ```
   long creditCardNumber = 1234_5678_9012_3456L
   ```

   如果能及时找到解决方案还会提供对无符号字面值处理方式的改进。

3. 集合字面值。通过类似于数组初始化器的语法为不变的list、set以及map字面值提供支持，同时还会为list与map的索引访问提供支持。

4. 改进的用于泛型实例创建的类型推断。使用有限的类型推断进行类实例创建需要为构造方法显式声明参数化类型，然而这些类型却可以从上下文中推断出来，然后它们就会被一个空的类型参数集合替换掉。这样，相对于：

   ```
   Map<String, List<String>> anagrams = new HashMap<String,
   List<String>>();
   ```

   我们可以这样写：

   ```
   Map<String, List<String>> anagrams = new HashMap<>();
   ```

5. 对JSR 292 的语言支持。包括invokeDynamic指令、方法句柄调用、某些不严格的约定以及外来的标识符。

6. 简化的可变参数方法调用。当方法将可变参数与非具化的数组类型组合在一起时就会产

生警告，现在将该警告由调用处转移到了方法声明处。

7. 可以在switch语句中使用String。

以上大部分提案都将于今年 10 月底反映到 JDK 7 的 Mercurial 仓库中。

还有三个提案尚未最终发布，它们是：改进的Java异常处理、Elvis与其他Null-Safe操作符以及大数组。

Joseph Darcy说到：

> 对于 Java 语言来说，改进的异常处理很值得我们期待，然而它对于类型系统来说是个风险，我们尚未评估是否有足够的资源以在 JDK 7 中实现该特性。我倒是期望能在未来重新考虑该特性以促进语言的不断发展。虽然 Elvis 与其他相关的操作符在 Groovy 中很有用，但由于 Groovy 与 Java 的差别，比如原生类型的存在以及与装箱/拆箱的交互使得这些操作符对 Java 意义不大了。JDK 7 将提供其他方式来简化空操作（null-handling）的烦恼，如 JSR 308 的空检查。毫无疑问，天生支持 32 位以上条目的集合一直是大家所梦寐以求的。对集合的语言支持会开发一个程序库来实现这一点，这样平台就可以直接处理大的数据结构了。

原文链接：http://www.infoq.com/cn/news/2009/09/coin_final

相关内容：

- 淘宝开放平台架构组件体系初探
- 问卷结果：JavaEE容器重部署和重启时间
- Lucene 2.9：数字字段支持、新分析器及性能优化
- JDK 7：java.util.Objects中应该包含哪些常用方法呢
- Java应用开发中代码生成工具的作用

热点新闻 | News

SOA与云计算有多大关联？

作者Boris Lublinsky 译者马国耀

Joe McKendrick已发布一篇题为"经验总结：SOA和云面临的挑战和机会"的会议纪要，参会人员由Phil Wainewright，David Bressler，Ed Horst和Joe McKendrick等人组成。

讨论从 Phil Wainewright 的问题开始：**你们每个人是如果定义云的，如果它和 SOA 存在区别的话，那么最关键的区别又是什么？**

Joe McKendrick 认为:

> ……过去的一年太令人惊叹了，这些概念一齐汇聚到大家面前，这里我只谈 SOA 和云。SOA 在 90 年代初就已经来到人们周围了，而且很过公司正进行着 SOA……现在，我们更多地看到人们在强调向云的转型……我认为这二者的主要区别是：SOA 是一种架构，是底层架构，是人们创建、管理、编排服务的方式。而云是一种技术，从一方和另一方之间交付这些服务是通过它实现的。但是我认为，目前正处于这样一个阶段，你不能只有其中之一，而应该二者兼备。

Phil Wainewright 继续问道：

> 可否这么简单地说：云就是 SOA，只不是过它是一种以面向 Web 方式实现的 SOA？……我这么说的意思是，因为它是一个开放的环境，而且你不知道你正在和谁进行交互，因此不得不要做所有的服务层的工作，如定义合约（服务的合约）……还有安全需求，你必须做所有的那些在受控的企业环境中不一定非要做的事情，因为在受控的环境里，你了解正在发生的事情……我认为我们在 SOA 和云之间已经建立了很多的共性。云正在做的很多事情是当初 SOA 建好后要做的事情，因此，也许我们可以继续做出这样的前提假设：云可以学习 SOA 的经验。那么，SOA 教给我们什么 ？有哪些经验教训可以让我们在实施云的过程中不再犯相同的错误。

为了回答这个问题，Ed Horst 提到了 SOA 的三个主要经验教训：

> ……（1）从一个具体的项目开始，这个项目要有合理的边界，并将在完成是能够对日常业务有所影响……你得需要能够经常使用的东西。（2）另一个是……避免煮沸整个大海的做法，比如在还没做任何云的工作之前，就要将所有的东西变成云……但这也不是说在做第一个项目的时候就完全不考虑最终的方向。因此，我所看到的一种更为成功的策略是混合的做法，在考虑整体架构走向的同时采取更广阔的举措，最后我们可能会花2年、3年、4年甚至5年的时间才能完成，但是在启动项目之后要开展一些实际可行的工作。（3）然后……管理系统，要尽早、尽可能经常地对系统进行监管。做得早的话你一般不会后悔，相反你可能经常会因为没有这么做而后悔。

Phil Wainewright 的观点是，一个包罗万象的面向 Web 的架构可以让我们统一 SOA 和云计算。他继续说到：

> ……面向 Web 的架构的一个特征是 REST 接口，它更加简单，因为他不需要做 SOA 相关的其他事情。这个特征也是云计算的特征，一个必要的特征，云计算追求更加“lowest common denominator”的接口（译注：在数学中，Lowest common denominator 被称为最小公分母；在计算机中，最初的意思是计算机平台的指令选择技术，当对一个程序进行编译生成可执行程序并使之可移植到其他平台上时，由于每个处理器都有自己的补充指令，所以只有使用不同处理器所共有的那些指令编译出来的可执行程序才具有最好的移植性。在这里，指得是接口定义要更通用，更简单）。

Ed Horst 认为 REST 和 SOAP 都有特定的应用场合，这取决于不同的交互类型：

> 如果交互本身就更加事务性，更谨慎，更业务敏感的功能，那么通常使用 SOAP 接口来进行交互。但如果是一个类似于查询和更新这种轻量级的操作，而且对业务的影响也较小，那么用户一般使用 REST 接口……这也不是说 REST 就不能用于具有事务性的场合，但是当你为了实现事务、安全、良结构的消息或者其他类似的要求而向 REST 增加一些元素进去的话，这时的 REST 看起来就很像 SOAP 了。

在今天的 IT 界“云”是一个热词，而 SOA 似乎渐渐失宠，最少在分析家的眼里是这样的。相应地，当前还有一个非常流行的假设，无论 SOA 曾有多大的缺点和困难，云计算都将改进之并能够解决这些困难。事实上，如 Joe 所说的 SOA 关心的是合理的系统架构，而云关心的是基础设施。总所周知，再好的基础设施也不能挽救糟糕的架构。所以我们应该停止对灵丹妙药的祈祷，而着手去关心最基本的工作：合理的服务架构。

原文链接：http://www.infoq.com/cn/news/2009/09/SOACloud

相关内容：

- SOA的SPEC基准性能测试
- 使用Reservation模式实现SOA事务
- Open Group发布新SOA治理框架与服务集成成熟度模型
- 复杂事件处理的Forrester报告
- BPM和SOA的最佳实践和最差实践

热点新闻 | News

MonoDevelop正式迈入跨平台时代

作者 朱永光

昨天Miguel de Icaza在其博客上宣布了MonoDevelop的最新版本——2.2 beta 1。这是Novell官方正式支持的第一个跨平台版本，除了支持原本的Linux，现在还支持Windows和Mac OS X。

Miguel de Icaza 说道：

> 一直以来，人们都渴望得到一款跨平台的.NET IDE。 .....直到今天，2009 年 9 月 9 日，世人可以来尝试这样的工具了。

这个版本在带来Windows和Mac OS X安装包的同时，还和这两个平台也进行了紧密的集成，并支持在每个平台上的原生调试。除此之外，MonoDevelop的一大特色是具有丰富的插件，这次带来的新插件有：ASP.NET MVC开发插件、Silverlight开发插件和iPhone开发插件（利用了MonoTouch）。

MonoDevelop 2.2 beta 1 包含的完整特性如下：

- Windows 支持：官方支持，并提供安装包
- Mac 支持：官方支持，并提供安装包
- 项目管理：
  - 多目标运行时
  - 针对每种文件类型提供不同编辑和格式化策略
  - 自定义的执行模式
  - 全局程序集文件夹
  - 删除项目的自定义对话框

- 可以从 Mac 的 Nautilus 或 Windows Explorer 中拖文件到解决方案树上
- 加强了项目重载
- 开始支持.NET 4.0

- 文本编辑器：
  - 自动保存
  - 代码模板
  - 代码块选取
  - 提升了编辑大文件的性能
  - 提供了新的代码格式化功能
  - 即时代码格式化
  - 代码自动完成支持首字母匹配
  - XML 文档支持代码自动完成
  - 对 vi 模式进行了加强
  - 可自动生成某些代码片段
- 重构
  - 可解析命名空间
  - 具备预览功能的重命名
  - 抽取方法
  - 声明局部变量
  - 综合临时变量
  - 创建常量
  - 为类型创建单独的文件
  - 删除无用的 Usings
  - 对 Usings 进行排序

- 为属性创建对应的字段，或删除字段
- 支持多种键盘命令
- 内联（inline）重命名

- 调试器
  - 立即窗口
  - 在 Windows 上利用 Win32 调试器
  - 在 Linux 上开始支持 ASP.NET 调试
- ASP.NET MVC 插件
- IPhone 插件
- Moonlight 插件
- 极大加强了 Python 语言的支持
- 版本控制
  - 提供显示注解（Show Annotations）的命令
  - 加强了审阅更改视图（Review Changes View）的功能
  - 新增了创建补丁（Create Patch）的命令
- 其他
  - 在搜索结果中提供语法高亮
  - 加强了数据库插件
  - “Go to File”对话框现在支持多选
  - 可生成 Makefile
  - Vala 语言支持的加强
  - C/C++插件的代码自动完成现在更加稳定
  - C#代码自动完成的加强

随着 Mono 的逐步成熟，让.NET 应用程序跨平台地运行已经成为现实，而随着 MonoDevelop

支持跨平台并逐步成熟，相信跨平台地开发.NET 应用程序也将成为可能。

原文链接：http://www.infoq.com/cn/news/2009/09/monodevelop22b1

相关内容：

- 讨论：所有的成员都应该是virtual的吗？
- 微软为AJAX和jQuery类库提供CDN服务
- PowerShell 2 简介
- ASP.NET vs. PHP，哪个更快？
- .NETZ——.NET函数库的压缩和打包工具

推荐文章 | Articles

应用jBPM4 解决中国特色的流程需求

作者 辛鹏

1. jBPM4 的特点

jBPM 是 JBoss 众多开源项目中的一个工作流开源项目，也是目前应用最广泛的工作流项目。在今年的 7 月 10 号，JBoss jBPM 团队正式发布了 jBPM4 的正式版。jBPM4 完全基于流程虚拟机（PVM）的机制，对核心引擎进行了重新设计，而 PVM 的引入也使得 jBPM4 可以支持多流程语言了。除此之外还有很多其它的特点：

- 流程定义对象的变化

  在流程定义的对象上，节点类型划分更清晰，详细的对象解析，可参见我曾经写过的文章：《jBPM3 与jBPM4 实现对比》。

- 基于观察者模式的 Event-Listener 机制

  在 jBPM4 中活动节点对象 ActivityImpl、转移对象 TransitionImpl、流程定义对象 ProcessDefinitionImpl，全部继承了 ObservableElementImpl 对象，因此组成流程定义的三个主要元素都可作为观察者模式中的被观察者来观察，而这些对象本身就直接支持注册各种 Event，然后由 Listener 来监听这些 Event。

- 基于 ExecutionImpl、Command 模式和 AtomicOperation 实现的引擎调度

  在 jBPM4 中用 ExcutionImpl 替掉了 jBPM3 中 Token 机制，流程的流转，依赖于 ExcutionImpl 调用各个 AtomicOperation 原子操作（ExecuteActivity、MoveToParentActivity、TransitionTake、TransitionStartActivity、ExecuteEventListener、TransitionEndActivity）进行推进，而 ExecutionImpl 在各个实例对象之间进行转移，详细情况同样参见《jBPM3 与 jBPM4 实现对比》中的”流程启动序列图”和”流程推进序列图”。(注：当时的那篇文

章是基于 jBPM4 alpha 版本写的，在正式版中有一些变化。）

- 历史库的加入

  作为任何一个正式运行的大数据量的软件系统来讲，没有历史库的切分肯定只能当作玩具，而 jBPM3 没有任何的处理，必须由开发人员自己来解决。在 jBPM4 中终于加入了这个功能，不过我认为目前 Jbpm4 的历史库设计还是存在一些问题的，例如，它是在活动（或任务）结束时，直接将数据归入历史库，实际上我认为这是没有必要的，我认为在整个流程实例结束时去做这个事情反而更好一些。其次，在将运行期的实例归入历史库时，并不是将所有数据都进行了 copy，这就造成了在历史库中丢掉了一些运行期的数据属性。还有任务历史库中没有针对 task candidates 参与模式的任务拾取时间的记录导致无法做绩效统计等。

2. 国内人工任务密集型流程的典型特点

中国目前在工作流领域主要的应用是人工工作流，也就是以人工任务密集型的工作流为主。当然随着中国企业和公共组织的信息化发展越来越快，IT 系统的积累和建设经验也越来越丰富，因此以自动任务密集型为主的应用正在逐渐增多。但本文主要的还是讲述人工任务密集型工作流的特色、需求、场景及实现。这种类型的流程应用，主要集中在以下领域如：电子政务，行政审批，企业协同办公，电信、电力之工单，企业采购、合同、销售等。

从功能需求上来说，这些人工任务密集型流程的典型特点主要有以下几个方面：

1. 用户友好的流程定义工具，就是说由业务用户自己对流程进行定义。其实这是一个伪命题，因为此处讲的流程是可以 run 的，与业务系统有紧密的关系，完全地让最终用户来设计一个这样的流程是不现实的（即使是不可以 run 的 BPMN 同样也很困难，需要进行大量的培训）。但是业务用户是可以对已经建立好的流程进行改进的（呵呵，此处就是 BPI 的一个体现），因此提供一个基于 WEB 的，简单易用的、用户友好的流程设计器是非常有必要的。

2. 表单自定义，就是说能有一个可高度定制的表单设计器，用户可以随时根据业务需求进行调整，或者新建表单，这个需求对于有“语义层”的 form engine 来讲是可以实现的。那么除了表单自定义以外，当然还要能实现表单与流程任务的轻松绑定及表单权限的设定，使得不同环节的处理人能够对表单的不同域有不同的读写权限。

3. 灵活的临时动态性需求，例如：任意回退、会签（包括加、减签，补签）、撤销（又叫回退）、自由流（又叫动态路由）。此处之所以叫做灵活的临时动态性需求，就是因为这

些需求，存在着很强的人为性因素（呵呵，此处才是真正的中国特色）。

3. 应用jBPM4 解决国内的典型流程需求

3.1 用户友好的流程定义工具

关于流程自定义，jBPM4 本身提供了基于eclipse的plugin，可以让开发人员来进行流程的建模，但是我们不可能让一个业务流程管理员去下载使用Eclipse，因此我发起了一个基于jBPM4的开源工作流项目jBPM-side，上边我已经提到了国内流程的典型需求，而这些需求直接用jBPM4 来实现，需要很高的成本，因此将jBPM4 进行本土化的改造封装以满足国内流程应用的需求，就是发起jBPM-side项目的目的。而上边提到的所有典型需求，就是jBPM-side项目的特点。

目前jBPM-side正在全力开发基于flex的流程设计器，本项目的代码在google托管地址如下：http://code.google.com/p/jbpmside/，此设计器是真正意义上的一个用户友好的流程定义工具。我们初步的界面原型设计如下：

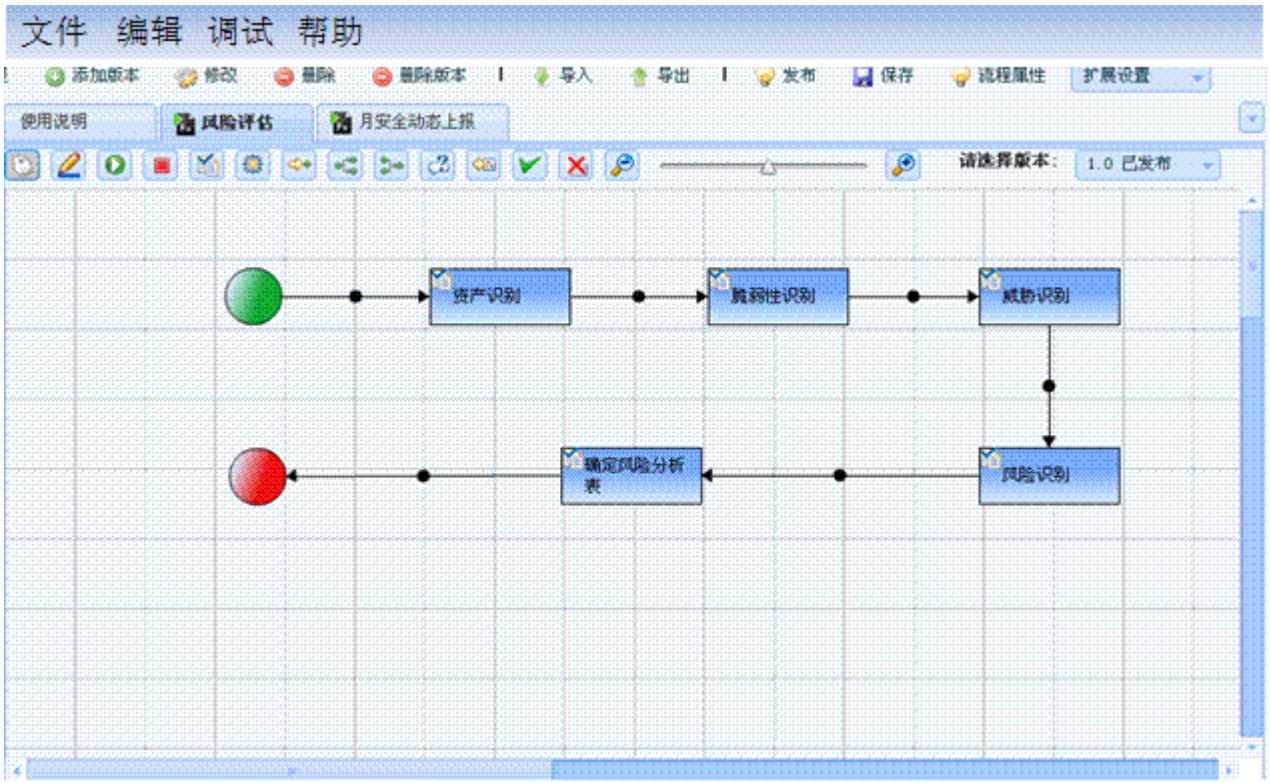


图 0 jBPM-side designer 界面原型示意图

另：jBPM4 本身也已经开始 BPMN 的相关代码开发了，建议也密切关注。

3.2 表单自定义

表单自定义的需求，应该来说在国内、国外都存在这种需求，我记得在jBPM的官方论坛里，Tom baeyens和其它人讨论过电子表单的问题，但是现在在论坛中已经找不到了，在wiki上还可以看到Tom写的文章：jBPM4andXForms，在这个文档中，Tom baeyens认为可以用xform来作为Task form的一个实现，并给出了Orbeon和Chiba的链接，但同时也提到了他们会继续调查freemarker和velocity这两个模板引擎，并且说这件事情仍需要继续讨论，所以目前jBPM的团队并没有在表单自定义方面的计划。jBPM4 的子项目GWT-console目前默认的表单实现采用了freemarker方案，但是freemarker仅仅是一个模板引擎框架，与真正的电子表单产品（例如infopath，一个完整的电子表单产品，包括表单设计器、表单引擎、数据存储、事件引擎等）还相差甚远。jBPM-side项目可能会基于Orbeon或Chiba进行表单引擎及设计器的本土化封装，同时也提供商业电子表单产品的调用接口。

3.3 灵活的临时动态性需求

3.3.1 回退

需求描述：

回退作为审批流来讲是最常见的需求，对于审批流来说，每一个审批环节都有可能会有审批通过、不通过 2 种情况，而审批不通过时，一般是回退到上一个环节，但是在某些情况下，有可能跨环节回退，例如，在第 5 个审批环节，审批不通过时，直接回退到第 2 或第 1 个环节。而到底回退到哪个环节是可以让用户根据业务需求进行自定义的，并且在回退环节工作完成之后，其下一步的方向也可以让用户自定义。如，要是由第 5 个环节回退到第 2 个环节，那么当第 2 个环节重新修改业务数据并办理完毕后，流程引擎可以设定是重新按照 2-3-4-5 的顺序重新执行一遍，也可以设定由第 2 个节点返回给第 5 个节点，由第 5 个节点重新审批。

场景及 jBPM4 实现思路：

图 1 串行流程示意图

场景 1 串行流程： 在这个场景中，由于流程是串行的，没有分支的情况，因此处理起来最简单（如图 1 所示）

场景 1 实现思路：

对于这个场景，在 jBPM4 中,最直接的方式，就是在需要回退的各个节点之间建立回退线(如图 5 所示，如果需要从节点 4 回退到节点 2，则直接在节点 4 之后加一个分支决策节点，连接两个分支，一个是 pass to 节点 5，一个是 reject to 节点 2)。

对于节点 2 来讲，在 jBPM4 中，其参与模式又分为 4 种 :task-assignee(assignee user、assignee expression)、 task candidates（ candidate-groups、 candidate-users)、 task assignment handler、task swimlanes。

1. task-assignee 任务的办理人的值可以直接指向一个特指的用户的 id 或者一个关联到业务中某个特指用户的表达式（ 例如 order.owner)。此时，如果 assignee 赋予了一个特指用户的 id，回退时不用做任何处理。如果 assignee 赋予了一个业务表达式，在回退时，需要保证业务表达式的运算逻辑能够正确执行。

2. task candidates 任务的办理人为几个特定的组的集合，或者用户的集合，在这个场景下，实际上就是我们俗称的“竞办”。这样的任务被称为 groupTask，必须被某一个组或者用户拾取才能进行办理，因此如果回退到这样的节点时，需要把任务回退给当时的拾取人。而拾取人需要到 HistoryTask 和 HistoryDetailImpl 两个实体对应的历史库中取得。

3. task assignment handler 任务的办理人通过用户自己编写的代码来实现，此时回退到这样的节点时，也不需要做特殊处理，只要保证自己的代码（ AssignmentHandler ）在执行回退逻辑时同样能够正确执行即可。

4. task swimlanes 任务的办理人为“泳道”，回退到这样的节点时，任务的办理人可以自动取得，同样不需要做任何处理。

以上 4 种情况，在回退之后的处理，又分为 2 种情况，一种是，按照原来的路径重新执行，即节点 2—节点 3—节点 4，另一种是，节点 2 办理完毕后，直接返回给节点 4。对于第 1 种情况，不需要做处理。而对于第 2 种情况，由于在节点 2 和节点 4 之间并没有一个转移存在，因此 jBPM4 也没有提供原生的支持。那么针对这种情况，笔者的解决思路是：通过动态创建跳转来实现(具体实现方法，详见 3.3.4)，实际上回退也可以用动态创建跳转来实现，而就不用画回退线了。

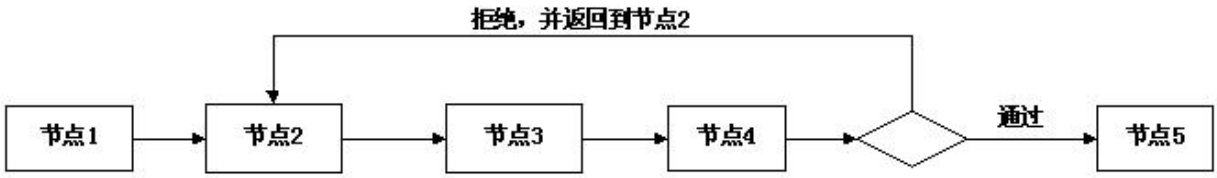


图 2 串行流程的回退实现

场景 2 M 选 N 分支流程： 对于这个场景，N 的取值范围为 M>N>=1，假设回退前流程的执行路径为节点 1-节点 2-节点 4-节点 5（见图 3），此时回退有两种情况：

1. 由节点 5 回退到节点 1，而节点 1 在进行业务数据的修改后，直接返回给节点 5 的处理人进行办理或审批；
2. 由节点 5 回退到节点 1，而节点 1 在进行业务数据的修改后，重新按照节点 1-节点 2-节点 4-节点 5 的路径再执行一遍；

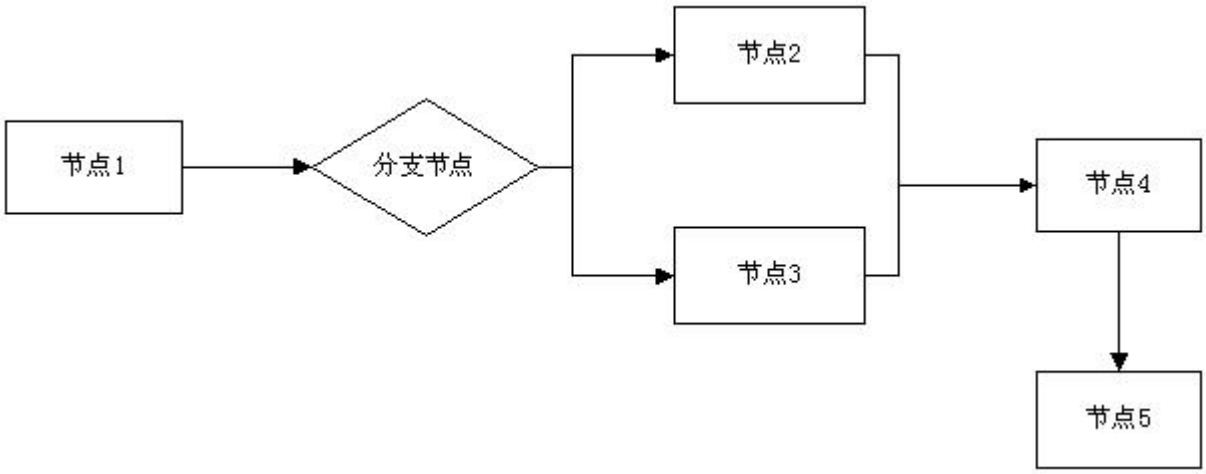


图 3 分支流程示意图

场景 2 实现思路：此场景与场景 1 不同的是，在回退之后继续审批时，需要考虑分支节点的决策条件，在自动决策时要保证决策逻辑正确执行，在人工决策时，需要记录原来的执行路径（保证重走 1-2-4-5 ）。

场景 3 同步分裂、与汇聚流程（如图 4 所示），回退前执行的路径为节点 1--节点 2--（节点 3、节点 4）--节点 5--节点 6，此时回退有 4 种不同的情况要考虑：

1. 由节点 6（或节点 5）回退到节点 2（或节点 1），节点 2 重新办理完毕后，直接返回给节点 6 的处理人进行办理或审批；

2. 由节点 6（或节点 5）回退到节点 2（或节点 1），节点 2 重新办理完毕后，重新按照节点 2--（节点 3、节点 4）--节点 5--节点 6 的路径再次执行一遍；
3. 由节点 6（或节点 5）回退到节点 3（或节点 4），节点 3（或节点 4）办理完毕后，直接返回给节点 6；
4. 由节点 6（或节点 5）回退到节点 3（或节点 4），节点 3（或节点 4）办理完毕后，按照节点 3—节点 5—节点 6 的执行路径，再次执行。

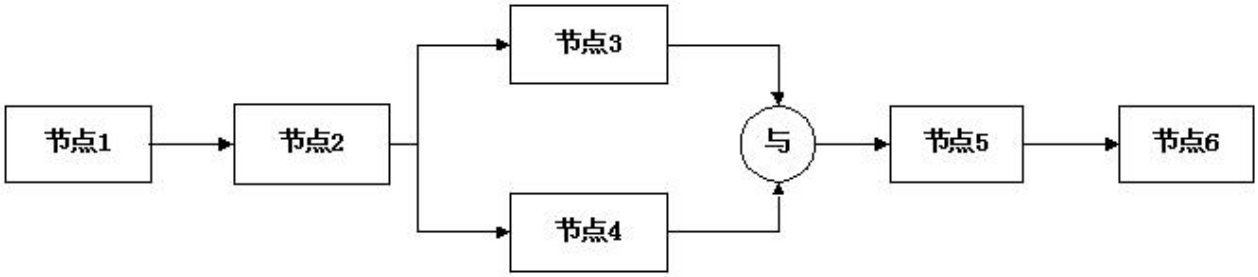


图 4 同步分裂，与汇聚流程示意图

场景 3 实现思路：此场景当流程由节点 6 回退到节点 3，节点 3 办理完毕后，重走路径节点 3-节点 5-节点 6 时，在“与节点”的与运算逻辑就会无法执行，因为另一个与分支，节点 4 并没有新的实例产生，流程就会僵死此处。在 jBPM4 中，可以通过修改 JoinActivity.java 这个类中的 protected boolean isComplete(List<executionimpl> joinedExecutions, Activity activity) 这个方法，在方法中加入对此场景的处理代码即可。

场景 4 同步分裂、或汇聚流程（如图 5 所示），回退前执行的路径为节点 1--节点 2--（节点 3、节点 4）--节点 5--节点 6，此时回退有 4 种不同的情况要考虑：

1. 由节点 6（或节点 5）回退到节点 2（或节点 1），节点 2 重新办理完毕后，直接返回给节点 6 的处理人进行办理或审批；
2. 由节点 6（或节点 5）回退到节点 2（或节点 1），节点 2 重新办理完毕后，重新按照节点 2--（节点 3、节点 4）--节点 5--节点 6 的路径再次执行一遍；
3. 由节点 6（或节点 5）回退到节点 3（或节点 4），节点 3（或节点 4）办理完毕后，直接返回给节点 6；
4. 由节点 6（或节点 5）回退到节点 3（或节点 4），节点 3（或节点 4）办理完毕后，按照节点 3—节点 5—节点 6 的执行路径，再次执行。

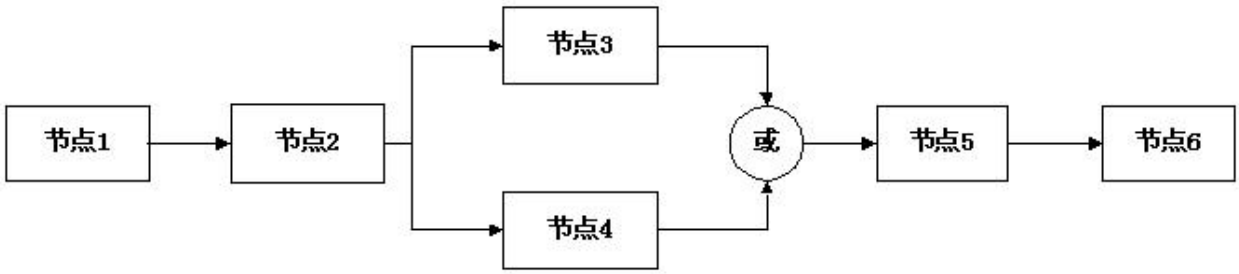


图 5 同步分裂，或汇聚流程示意图

场景 4 实现思路 :此场景与场景 3 相比 ,因为是或汇聚 ,因此在实现上不需要做任何处理了。

场景 5 子流程： 就是流程嵌套的场景，在父流程中通过子流程节点启动了子流程实例，此时回退时，就更复杂了，因为涉及到了不同的流程实例、还有在父子流程之间的数据传递。

场景 5 实现思路 : 子流程的回退，原则上是不支持的，因为涉及到不同的流程实例之间的回退，这个场景在 jBPM4 中实现起来异常的复杂，而且实际的业务场景也极少，因此在此不建议做这个实现。

小结：

综合来讲，回退本身在理论上来讲有各种各样的情况存在，再加上业务的回退（或者说业务逻辑补偿？如果需要你就必须在流程的每个环节进行业务快照的备份 ,在回退时调用这个快照进行补偿），此时回退可以说是整个流程体系中最复杂的情况了。但是在真实的业务场景中，有些情况可能是根本不会出现或者说很少出现的（毕竟理论不等于现实嘛）。因此技术人员一定要摒弃一个习惯，就是不要完全从理论和技术的角度考虑问题，一定要看用户是否有这样的需求，即便有了特定的需求，也首先要考虑的是能不能通过变通的方式处理，或者说服用户放弃这个不合理的需求，最后实在没办法了，再去考虑技术的实现。

3.3.2 会签

需求描述：

会签在政府或企业来讲都是必有的功能，尤其是审批流中。简单来说，会签是可以分为单步会签（只有一个审批环节），及多步会签（每一个子审批流有多个审批环节）的。

单步会签：很简单，就是在流程的某个环节需要由多个办理人（多个不同部门的领导）共同办理，或者签署意见。这个场景就不用说了，在企业或政府的内部都很常见。

多步会签（也叫并联审批）：其实就是一个单步的审批环节就变为了在部门内部一个比较复杂的审批流程，这个审批流程有多个审批环节。

加签：在流程定义期已经定义好会签范围（例如某个岗位或部门），但是在运行期，会签发起人发现对于某个个例需要新增会签人或会签单位，而且新增的会签对象不在原来设定好的范围内。此时由会签发起人直接进行加签操作。

减签：同上，只是相反的操作而已。

补签：会签发起人已经将会签任务发送给张、李、王三个人，而此时，张发现这个任务还需要孙来会签，那么此时，可以由张直接发起一个给孙的补签任务，而不必回退到会签发起人那里。

会签百分比：会签发起人将任务发送给 5 个人办理，而结果是只要有 80%的会签百分比即可算审批通过（也就是说只要有 4 个人审批通过就 OK 了）。

场景：

单步会签：对于单步会签的场景很简单不在此描述了。

单步会签的实现思路：可以对 TaskService 进行扩展开发，实现动态任务实例的创建，可参照 TaskActivity 这个类中的方法进行扩展，扩展之后再调用 addTaskParticipatingUser()或 addTaskParticipatingGroup()方法实现动态增加任务办理人，此时即实现了单步会签功能。

多步会签场景一：审批环节相同。在企业内部的各个部门之间（例如，办公室、采购部、财务部）进行并联审批，每个部门中都需要多个审批环节，而这些部门的审批环节完全相同，只是每个审批环节的办理人不同而已（例如在财务部，需要财务专员、财务经理、财务总监等审批；在办公室需要办公室科员、办公室副主任、办公室主任审批），因此可以公用一个子流程定义。

场景一的实现思路：最常见的方案是通过启动一个子流程定义的多个子流程实例来实现多步会签。这时，即便是对于会签的部门数是未知的，需要动态决定，也可以轻松实现，只需要在运行期根据会签部门数动态地创建同等数量的子流程实例就可以了。 但是不要高兴的太早 J，由于在 jBPM4 中，流程的推进是依赖于 ExecutionImpl 来执行的，而对于每一个流程实例 ProcessInstance 持有的 ExecutionImpl 实例也只有一个与之关联的 subProcessInstance，因此对于一个子流程节点 SubProcessActivity 来说也就只能有一个子流程实例与之关联了，此时要想通过启动一个子流程定义的多个子流程实例来实现多步会签，实现方法与在 jBPM3 中实现多子流程实例类似（可以在网上搜到很多 demo），就是结合 Event-Listener 机制和对

Variable 的使用，去创建多个子流程实例（注意的是在 jBPM4 中没有 ActionHandler 了，取而代之的是基于观察者模式的 Event-Listener 机制）。

多步会签场景二： 审批环节不同。实际上与场景一相比，就是会签部门的审批环节不同了，也就是说在企业内部的每个部门都要有自己的审批流程，其它与场景一是完全一致的。

场景二的实现思路：此场景，可以和场景一的实现相同，唯一不同的是由一个子流程定义的多个实例，变为了不同子流程定义的不同实例。

多步会签场景三： 分布式审批。在政府部门，例如我们需要去政府的行政大厅去办理新公司注册，那么在行政大厅启动一个新公司注册的流程，在申请人提交完所有资料后，流程继续向下执行，这时可能就需要工商局、公安局、地税、国税等多个委办局进行内部的并联审批，每个委办局都需要在内部走一个复杂的审批流程，每个委办局的流程审批完毕后，流程回到行政大厅的那个父流程中。此场景与场景二相比，其实就是企业内部的各个部门变为了不同的委办局或子公司，此时的流程是分布式部署在各个委办局或子公司的。

场景三的实现思路：由父流程执行到某个会签节点时，通过 jms 消息向各个会签部门（注意这个会签部门一般都是分布的，例如公安局、工商局、税务局等）发送业务数据，而父流程在此等待会签结果，而各个会签部门都有自己的监听器，在监听到会签请求时，在内部发起自己的审批流，内部审批完毕再发送业务数据给父流程，父流程接收到审批结果的业务数据后，流程继续向下执行。

在 jBPM4 中实现起来就很简单了，因为 jBPM4 提供了 jms 的消息、Event-Listener 机制，而 jBPM4 本身也完全是基于观察者模式进行设计的，此时通过在会签节点上绑定特定的 Listener，在 Listener 中向特定的目标发送 jms 消息（可参见 MailListener 的实现）。

小结：

对于单步会签的应用场景较多，在 jBPM4 中也提供了直接的支持。

对于多步会签场景一，实际上这个场景在真实的企业中并不多见，因为大多数的需要会签的业务都是只需要部门中的一个关键领导审批就可以了，也就是单步会签的场景。当然如果在某个特定的企业中，这种情景很多，为了流程管理员使用方便，那么对 jBPM4 的代码做一定的修改也是可以的。

对于多步会签场景三，实际上，由于各个部门（公安局、工商局、税务局）都是分布的，采用的工作流产品也是不同的，即便是同一个公司的产品，也是分布式部署的，因此在这个场景中，是不需要用 subprocess 或 subProcessActivity 这些概念的。因为此时的并联审批本质上

是两个相同等级的流程之间的通信而已。

3.3.3 撤销

需求描述：

任务在发送给下一个办理人之后，发现任务发送错误了，此时在下一个办理人还没有办理之前，可以撤回当前任务，而重新选择其它人进行办理。

场景：

撤销的场景与回退的场景很类似，虽然有很多的场景，但是各个场景的处理情况是一样的，因此在此只给出最简单的一个场景，如下图 6 所示：

图 6 串行流程示意图

节点 2 的处理人（假设是张三），办理完毕之后，将任务提交，此时任务到达了节点 3（假设李四办理），这时李四就会收到一个待办任务，在李四还没有办理之前，张三突然发现，有一个业务数据填写错误，或者粘贴的附件错误，这时张三需要将发送给李四的任务撤销（也叫收回），重新更正数据后或修改粘贴的附件后再发送给李四审批。还有一种情况是，假设节点 3 的办理人有 2 个人（李四和王五），那么张三需要在运行期根据业务特性手动地选择任务是提交给李四还是王五，但是由于李四的误操作，把本来应该发给王五的办理任务错发送给了李四，此时，在李四办理之前，张三也可以将发送给李四的任务撤销（或收回），然后重新发送给王五。

jBPM4 实现：

在 jBPM4 中实现撤销，虽然场景有很多，但是各个场景的处理是一样的，也就是在撤销时，首先删除需要撤销的任务实例及其与此任务实例相关的所有工作流实例。在 jBPM4 提供了级联删除任务实例的相关方法，如下：

```
TaskServiceImpl.java
public void deleteTaskCascade(String taskId) {
   commandService.execute(new DeleteTaskCmd(taskId, true));
 }
```

其次修改当前任务实例的状态，即将张三的已经办理完毕的节点 2 对应的 TaskInstance 的状态更改为待办状态：

```
（Task.STATE_OPEN）
                        task.setState(ask.STATE_OPEN);
                        taskService.saveTask(task);
```

小结：

撤销的需求在审批流中也是最常见的业务需求，毕竟人都有犯错的时候嘛，而且一般的软件都有 Undo 功能。但是对于 jBPM4 中的 fork-join，sub-process 都需要把撤销任务的相关实例都删除。

3.3.4 自由流（动态路由）

需求描述：

针对于特定的业务实例，在原本没有转移关系的环节之间进行特定的跳转，例如在一个串行的流程中（1-2-3-4-5），节点 2 与节点 5 之间是没有任何转移存在的，但是针对于某个运行期的特定业务实例，要求，审批环节直接从节点 2 跳转到节点 5(而略过了节点 3 和节点 4)。

场景：

图 7 串行流程示意图

jBPM4 实现：

图 8 动态路由实现示意图

如图 8 所示，在节点 2 和节点 4 之间由程序动态地创建一个转移（transition），并设定为优先级最高，那么在执行 takeTransition()方法时，按照优先级优先执行动态的转移，然后对外

暴露一个 jumpTransition（String destinationActivityName）方法给客户端。在 jBPM4 中，可按照如下步骤实现：

1. 参照 CompleteTaskCmd.java 扩展开发用于跳转的 cmd：JumpTaskCmd（jBPM4 中的动作都是基于 Command 模式的）；
2. 参照 TransitionStartActivity.java 的原子操作，定制开发用于跳转的原子操作 JumpTransitionStartActivity;

小结：

jBPM4 中的执行动作都是基于 Command 模式来实现的，因此我们在扩展开发自己的跳转动作时，就可以做到对 jBPM4 本身的代码不做侵入修改而实现。

4. 总结

jBPM4 做为目前应用最广泛的开源工作流产品，有着很好的架构及扩展性，但是由于国外的流程应用与国内的应用存在着一些不同，因此要想让 jBPM4 更好地满足国内的流程应用的需求，就需要做一定的定制开发。而其最大的问题就是没有一个可供业务流程管理员用来进行流程改进的设计器，因此从这一点上，也大大阻碍了其在国内的应用。而本文针对国内的流程应用的一些典型特点进行较详细的分析，并给出基于 jBPM4 的大概的解决思路和办法，但是并没有给出很详细的完全可以 run 的 code，但是笔者认为这些不是最重要的，最重要的是解决问题的思路。而具体的解决方案的 code，请读者关注 jBPM-side，因为是一个开源项目，因此在此项目中，将会看到完全可以 run 的 code。

原文链接：http://www.infoq.com/cn/articles/jbpm4-process-requirement

相关内容：

- 开放流程用户组（OPUG）发起人辛鹏专访
- 使用JBoss ESB和JBPM实现垂直市场解决方案（VMS）
- BPMN 2.0 虚拟圆桌访谈
- OSWorkflow中文手册
- 过程组件模型：下一代工作流

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

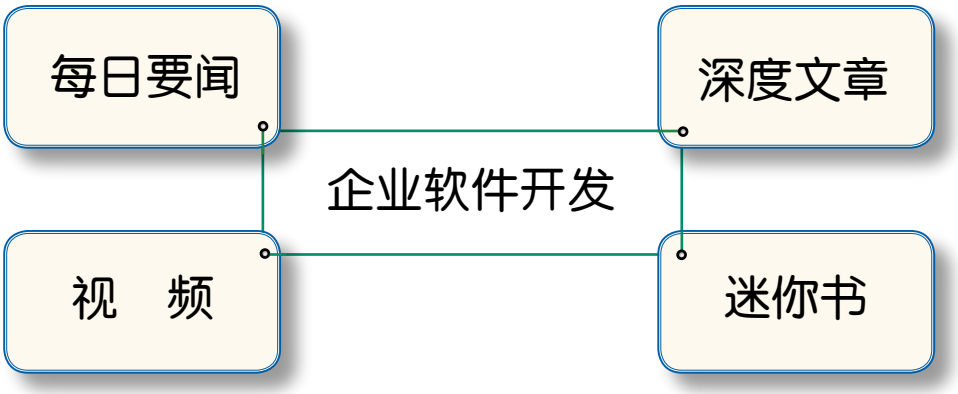
我们的**定位**：关注高级决策人员和大中型企业

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 | Articles

表达式即编译器

作者 Marc Gravell 译者赵劼

介绍：问题所在

无论你是否喜欢反射，很多情况下你不可避免地会需要在运行时（而不是编译时）访问一个类型的成员。可能你在尝试着编写一些验证、序列化或是 ORM 代码，也可能必要的属性或方法是在运行时从配制文件或数据库中获得的。无论是什么原因，你在某些时候一定写过 GetType() ——就像这样：

```
Type type = obj.GetType();
foreach (var property in type.GetProperties())
{
    Console.WriteLine("{0} = {1}",
       property.Name,
       property.GetValue(obj, null));
}
```

虽说这个普通的示例可以正常工作，但它不够理想——这里有一些关键性的问题：

- 它比较慢——你偶尔使用的话还好，但是在密集的循环中就会有明显的差距了。
- 没有比较方便的做法把那些准备工作“打包”成可复用的代码。

而且，在动态代码的需求变复杂时，情况则会更加糟糕。

当然，一个明显的应对方式是“不要使用反射”——例如，手动（或使用代码生成工具）为每个类型写一个方法来做对应的事情。表面看来这种简单的做法没有问题，但是这会导致大量的重复代码，而且也无助于我们使用编译期间还不可知的类型。

我们需要的是某种形式的元编程（meta-programming）API。幸运的是，在.NET 2.0 中提供 Reflection.Emit 来动态编写 IL（Java“二进制码（bytecode）”在.NET 中的对应物），不过这也要求开发人员直接处理所有细微的装箱、类型转换、调用堆栈细节或操作符“提升”等操作——简而言之，你需要了解大量本不需要知道的 CLI 内容。另一个选择是 CodeDom，不过这也是一种显式的代码生成方式，我们想要的做的事情往往会淹没在大量构造 CodeDom 所需要的繁琐事务中。

我们需要的是一种折衷的方案，一种足够高的抽象让我们不必关心实际的 IL 如何，但也不能过于高级：我们的代码想要尽可能的简单并富于表现力。

背景：走近System.Linq.Expressions.Expression

微软在.NET 3.5 中引入了 LINQ。LINQ 的关键部分之一（尤其是在访问数据库等外部资源的时候）是将代码表现为表达式树的概念。这种方法的可用领域非常广泛，例如我们可以这样筛选数据：

```
var query = from cust in customers
            where cust.Region == "North"
            select cust;
```

虽然从代码上看不太出彩，但是它和下面使用 Lambda 表达式的代码是完全一致的：

```
var query = customers.Where(cust => cust.Region == "North");
```

LINQ 以及 Where 方法细节的关键之处，便是 Lambda 表达式。在 LINQ to Object 中，Where 方法接受一个 Func<T, bool>类型的参数——它是一个根据某个对象（T）返回 true（表示包含该对象）或 false（表示排除该对象）的委托。然而，对于数据库这样 的数据源来说，Where 方法接受的是 Expression<Func<T, bool>>参数。它是一个表示测试规则的**表达式树**，而不是一个委托。

这里的**关键点**，在于我们可以构造自己的表达式树来应对各种不同场景的需求——表达式树还带有编译为一个强类型委托的功能。这让我们可以在运行时轻松编写 IL。

那么，什么是一个表达式树？

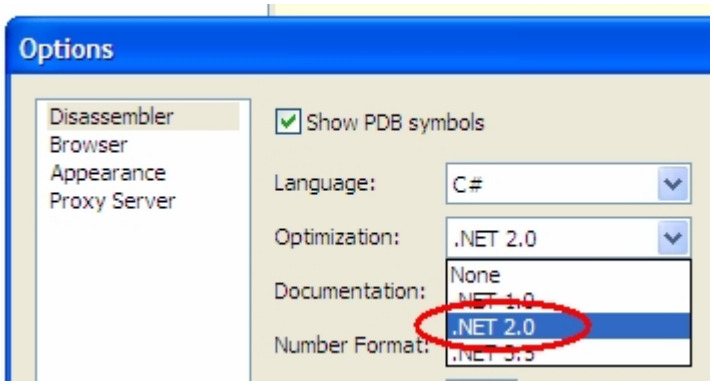
与一般情况下 C#编译得到的 IL 不同，一个 Lambda 表达式会被编译为一个**表现代码逻辑**的对象模型。于是，例如数据库提供者（provider）便可以观察这个对象模型，理解我们编写

代码的**目的**，在必要时便可以将这种目的转化为其他形式（如 T-SQL）。

我们不妨独立观察先前的 Lambda 表达式：

```
Expression<Func<Customer, bool>> filter =
    cust => cust.Region == "North";
```

如果要观察编译器做的事情，我们需要如平时般编译示例代码（Lambda表达式），然后在强大（且免费）的Reflector中查看——不过在此之前，我们要将.NET 3.5 (C# 3.0)优化选项关闭。



然后观察反编译器的输出内容，就会发现一些原本应该由我们自己编写的 C#代码：

```
Expression<Func<Customer, bool>> filter = Expression.Lambda<Func<Customer, bool>>(Expre
```

请注意，虽然编译器直接访问了 MemberInfo 对象，并且使用了非法的变量名——所以你无法直接让这些输出编译通过，它只是用来参考的。有趣的是，事 实上 C#语言规范中并没有指明编译器是如何将代码转化为表达式树的，因此，使用编译结果作为参考实现，是为数不多的可用于研究表达式树的方法之一。

为了研究表达式树的手动编写方法，我们需要把相等性判断（==）和成员访问（.）视为接受运算对象的普通方法：

```
Expression<Func<Customer, bool>> filter =
    cust => Equal(Property(cust,"Region"),"North");
```

我们现在可以构建一个相同的，接受 Customer 类型作为参数，判断其 Region 属性是否为字符串“North”，并返回一个布尔值的表达式树。

```
// declare a parameter of type Customer named cust
ParameterExpression cust = Expression.Parameter(
```

```
    typeof(Customer), "cust");
// compare (equality) the Region property of the
// parameter against the string constant "North"
BinaryExpression body = Expression.Equal(
    Expression.Property(cust, "Region"),
    Expression.Constant("North", typeof(string)));
// formalise this as a lambda
Expression<Func<Customer, bool>> filter =
    Expression.Lambda<Func<Customer, bool>>(body, cust);
```

最后的“Lambda”语句是将表达式树标准化为一个完全独立的单元，并包含一系列（这个例子中只有一个）用于表达式树内部的参数。如果我们传入一个 Lambda 内不存在的参数，则会抛出一个异常。

一个表达式树，其实只是一个表现我们意图的不可变的对象模型：

- filter (lambda)
  - parameter: Customer: “cust”
  - body (binary)
    - method: equals
    - left (member)
      - member: “Region”
      - expression: “cust” parameter
    - right (constant)
      - string: “North”

在获得一个完整的表达式树之后，我们可以将其交由 LINQ 提供者处理（此时它就会被真正当作一颗“树”来处理了），或者把它编译为一个委托——即动态生成所需的 IL：

```
Func<Customer, bool> filterFunc = filter.Compile();
```

现在我们向委托对象中传入一个 Customer 实例并返回一个布尔值。

这看上去有些麻烦，但已经远比使用 Reflection.Emit 的方式来的简单了，尤其在一些较为复杂的情况下（如装箱）。

重要：将一个表达式树编译为委托对象涉及到动态代码的生成。如果想要获得最佳的性能，

你应该仅编译一次——将其储存起来（如放在一个字段中）并复用多次。

Expression的妙用

到目前为止，我们只是观察了一个用 C#就能完成的简单示例——这自然没什么大不了的。所以我们现在来看一些 C#无法做到的东西……我经常被人问及，有什么 方法可以编写一个通用的“加法”操作，可以对任意类型进行计算。简而言之：没有这样的语法。不过我们可以使用 Expression 来做到这一点。

简单起见，我使用 var 关键字来代替显式变量类型，我们现在将要观察如何简单地缓存可重用的委托对象：

```
public static class Operator<T>
{
    private static readonly Func<T, T, T> add;
    public static T Add(T x, T y)
    {
        return add(x, y);
    }
    static Operator()
    {
        var x = Expression.Parameter(typeof(T), "x");
        var y = Expression.Parameter(typeof(T), "y");
        var body = Expression.Add(x, y);
        add = Expression.Lambda<Func<T, T, T>>(
            body, x, y).Compile();
    }
}
```

因为我们在静态构造函数中编译委托对象，这样的操作只会为每个不同的类型 T 执行一次，接着便可重用了——所以对于任何代码我们现在都可以直接使用 Operator<T>。有趣的是，这个简单的“Add”方法隐藏了许多复杂性：值类型或引用类型、基础操作（直接有 IL 指令与

之对应）、隐藏操作符（方法为类型定义的一部分）以及提升操作（用于自动处理 Nullable<T>）等等。

这个做法甚至支持用户自定义类型——只要你为它定义了+运算符。如果没有合适的运算符，这段代码便会抛出异常。在实际使用过程中这不太会是个问题，如果你不放心的话，可以在 Add 外加上 try 语句进行保护。

MiscUtil类库提供了完整的通用操作符实现。

Expression支持哪些东西？

在.NET 3.5 中，Expression 支持完整的用于查询或创建对象的操作符。

- 算术运算：Add、AddChecked、Divide、Modulo、Multiply、MultiplyChecked、Negate、NegateChecked、Power、Subtract、SubtractChecked、UnaryPlus
- 对象创建：Bind、ElementInit、ListBind、ListInit、MemberBind、MemberInit、New、NewArrayBounds、NewArrayInit
- 二进制运算：And、ExclusiveOr、LeftShift (<<)、Not、Or、RightShift (>>)
- 逻辑运算：AndAlso (&&)、Condition (? :)、Equal、GreaterThan、GreaterThanOrEqual、LessThan、LessThanOrEqual、NotEqual、OrElse (||)、TypeIs
- 成员访问：ArrayIndex、ArrayLength、Call、Field、Property、PropertyOrField
- 其他：Convert、ConvertChecked、Coalesce (??)、Constant、Invoke、Lambda、Parameter、TypeAs、Quote

例如，我们可以用设置公开属性的方式实现浅克隆——最简单的做法可以这样写：

```
person => new Person
{
    Name = person.Name,
    DateOfBirth = person.DateOfBirth, ...
};
```

对于这种情况，我们需要使用 MemberInit 方法：

```
private static readonly Func<T, T> shallowClone;

static Operator()

{

    var orig = Expression.Parameter(typeof(T), "orig");

    // for each read/write property on T, create a
    // new binding (for the object initializer) that
    // copies the original's value into the new object

    var setProps = from prop in typeof(T).GetProperties (

                    BindingFlags.Public | BindingFlags.Instance)

                where prop.CanRead && prop.CanWrite

                select (MemberBinding) Expression.Bind(

                    prop, Expression.Property(orig, prop));

    var body = Expression.MemberInit( // object initializer

       Expression.New(typeof(T)), // ctor
       setProps // property assignments

    );

    shallowClone = Expression.Lambda<Func<T, T>>(

       body, orig).Compile();

}
```

这种做法比标准的反射方式的性能要高的多，而且不需要为每种类型维护一段代码，也不会遗漏一些新增的属性。

类似的方法还可以用于其他一些方面，如在 DTO 对象之间映射数据，在类型与 DataTable 对象之间建立关系——或比较两个对象是否相等：

```
(x,y) => x.Name == y.Name &&

  x.DateOfBirth == y.DateOfBirth && ...;
```

在这里我们需要使用 AndAlso 来结合每个操作：

```
private static readonly Func<T, T, bool> propertiesEqual;

static Operator()
```

```
{
   var x = Expression.Parameter(typeof(T), "x");
   var y = Expression.Parameter(typeof(T), "y");
   var readableProps =
       from prop in typeof(T).GetProperties(
          BindingFlags.Public | BindingFlags.Instance)
      where prop.CanRead
      select prop;
    Expression combination = null;
    foreach (var prop in readableProps)
    {
      var thisPropEqual = Expression.Equal(
          Expression.Property(x, prop),
          Expression.Property(y, prop));
      if (combination == null)
      { // first
         combination = thisPropEqual;
      }
      else
      { // combine via &&
          combination = Expression.AndAlso(
                combination, thisPropEqual);
      }
   }
   if (combination == null)
   { // nothing to test; return true
      propertiesEqual = delegate { return true; };
   }
```

```
    else
    {
        propertiesEqual = Expression.Lambda<Func<T, T, bool>>(
              combination, x, y).Compile();
    }
}
```

Expression的限制——及展望

到目前为止，Expression 的表现几近完美——然而，还有 LINQ 表达式的构造方式还有许多明显的限制：

- 没有内置的机制可以改变对象的属性/字段。
- 没有方法可以执行一系列的操作

在上面的例子中，我们可以使用 AndAlso 将多个步骤连接成一个逻辑语句。然而，在.NET 3.5 中无法使用一个表达式树来表现如下的语句：

```
person.DateOfBirth = newDob;
person.Name = newName;
person.UpdateFriends();
person.Save();
```

我们之前看到的 Bind 操作只能在创建新对象时使用。我们可以获取 setter 方法，但是我们无法将多个方法调用串联起来（就像“流畅”API 那样，可惜目前没有）。

幸运的是，.NET 4.0 扩展了 Expression API，增加了新的类型和方法。这么做的目的是支持动态语言运行时（DLR，Dynamic Language Runtime）。这大大扩展了：

- 修改操作：AddAssign、AddAssignChecked、AndAssign、Assign、DivideAssign、ExclusiveOrAssign、LeftShiftAssign、ModuloAssign、MultiplyAssign、MultiplyAssignChecked、OrAssign、PostDecrementAssign、PostIncrementAssign、PowerAssign、PreDecrementAssign、PreIncrementAssign、RightShiftAssign、SubtractAssign、SubtractAssignChecked
- 算术操作：Decrement、Default、Increment、OnesComplement

- 成员访问：ArrayAccess、Dynamic
- 逻辑运算：ReferenceEqual、ReferenceNotEqual、TypeEqual
- 逻辑流：Block、Break、Continue、Empty、Goto、IfThen、IfThenElse、IfFalse、IfTrue、Label、Loop、Return、Switch、SwitchCase、Unbox、Variable
- 异常操作：Catch、Rethrow、Throw
- 调试：ClearDebugInfo、DebugInfo

这对编写动态代码来说可谓是个天大的好消息（它甚至包含了将代码编译为 MethodBuilder 的能力，可生成动态类型）。例如，我们可以用它来编写一个简单的 for 循环来打印数字 0 到 9：

```
            var exitFor = Expression.Label("exitFor"); // jump label
            var x = Expression.Variable(typeof(int), "x");

            var body = Expression.Block(new[] { x }, // declare scope
variables
                Expression.Assign(x,
                    Expression.Constant(0, typeof(int))), // init
                Expression.Loop(
                    Expression.IfThenElse(
                        Expression.GreaterThanOrEqual( // test for
exit
                            x,
                            Expression.Constant(10, typeof(int))
                        ),
                        Expression.Break(exitFor), // perform exit
                        Expression.Block( // perform code
                            Expression.Call(
                                typeof(Console), "WriteLine", null,
x),
                            Expression.PostIncrementAssign(x)
                        )
                    ), exitFor));
```

```
            var runtimeLoop =
Expression.Lambda<Action>(body).Compile();
```

虽然看上去似乎不是那么一鸣惊人，但如果您要在运行时编写编译好的代码，它比其他方式要方便和灵活得多。

之前我们通过属性复制来克隆一个对象——不过现在我们已经有能力把一个对象的数据复制给另一个对象了，这点对于 ORM 工具来说非常有用，例如跟踪一个对象的状态，执行外部更新，再提交这些改变。我们可以这么做：

```
            var source = Expression.Parameter(typeof(T), "source");
            var dest = Expression.Parameter(typeof(T), "dest");
            // for each read/write property, copy the source's value
            // into the destination object
            var body = Expression.Block(
                from prop in typeof(T).GetProperties(
                    BindingFlags.Public | BindingFlags.Instance)
                where prop.CanRead && prop.CanWrite
                select Expression.Assign(
                    Expression.Property(dest, prop),
                    Expression.Property(source, prop)));
            var copyMethod = Expression.Lambda<Action<T,T>>(
                body, source, dest).Compile();
```

使用表达式在运行时创建类型

.NET 4.0 中有个有趣的功能：在此之前我们可以把一个 Expression 编译为一个独立的委托，而如今你可以在动态创建新类型时，使用 CompileToMethod 方法将一个表达式树作为方法体提供给一个 MethodBuilder 对象。这意味着，在未来我们可以方便地编写功能丰富的 类型（包括多态和接口实现），而不用直接接触 IL。

语言中的高级表达式

尽管运行时已经支持非常成熟的表达式树了，但是在 C# 4.0 中并没有额外的语言支持——所

以我们只能手动创建表达式。这一般不会成为问题，因为几乎没有 LINQ 提供者会支持这些复杂表达方式。如果你在编译期 知道这些类型，不如直接编写普通的方法或匿名方法。不过这也为 4.0 以后的版本留下了想象空间。例如（只是推测），想象一下如果语言可以使用 Expression.Assign 和 Expression.Block 为数据库构造一个表达式树，就好比：

```
// imaginary code; this doesn't work
myDatabase.Customers
    .Where(c => c.Region == "North")
    .Update(c => {
       c.Manager = "Fred";
       c.Priority = c.Priority + 10;
    });
```

再次强调这只是"假如"——这需要在 LINQ 提供者上耗费大量的工作，但是我们从中可以看出：理论上说，一个 Expression 对象是有能力来封装我们的意图的（更新实体的多个属性）。可惜，无论是语言还是框架都还有很长的路要走，只有时间可以说明一切。

原文链接：http://www.infoq.com/cn/articles/expression-compiler

相关内容：

- 使用LINQ to SharePoint检索SharePoint中的数据
- LINQ to XSD的源代码发布在CodePlex上
- 通过索引器简化C#类型信息访问
- Aaron Erickson谈论LINQ和i4o
- Jimmy Nilsson谈LINQ to SQL

推荐文章 | Articles

演进架构中的领域驱动设计

作者 Mat Wall and Nik Silver 译者 王丽娟

领域驱动设计能非常容易地应用于稳定领域 ,其中的关键活动适合开发人员对用户脑海中的内容进行记录和建模。但在领域本身不断变化和发展的情况下，领域驱动 设计变得更具有挑战性。这在敏捷项目中很普遍 ,在业务本身试图演进的时候也会发生。本文分析了在反思、重建 guardian.co.uk 这一为期两年的计 划背景下我们是如何利用 DDD 的。我们展示了如何确保在软件架构中反映最终用户演变的认知 ,以及怎样实现该架构来保证以后的变化。我们提供了模型中重要项 目过程、具体演进步骤的细节。

顶层标题：

1. 计划背景

Guardian.co.uk 有相当长的新闻、评论、特性历史记录，目前已拥有超过 1800 万的独立用户和每月 1.8 亿的页面访问量。在此期间的大部分时间，网站都运行在原始的 Java 技术之上，但在 2006 年 2 月开始了一项重要的工作计划——将网站移植到一个更现代的平台上去，计划的最初阶段于 2006 年 11 月推出，当时推出了具有新外观的旅游网站，接着在 2007 年 5

月推出新的主页，之后相继推出了更多内容。尽管在 2006 年 2 月仅有少数几个人在做这项工作，但后来团队最多曾达到了 104 人。

然而，计划的动机远不止是要一个新外观。多年的经验告诉我们有更好的办法来组织我们的内容，有更好的方式将我们的内容商业化，以及在此背后，还有更先进的开发方法——这才是关键之所在。

其实，我们考虑工作的方式已超越了我们软件可以处理的内容。这就是为什么 DDD 对我们来说如此有价值。

遗留软件中阻碍我们的概念不匹配有两个方面，我们先简单看一下这两方面问题，首先是我们的内部使用者，其次是我们的开发人员。这些问题都需要借助 DDD 予以解决。

1.1. 内部使用者的问题

新闻业是一个古老的行业，有既定的培训、资格和职制，但对新闻方面训练有素的新编辑来说，加入我们的行列并使用 Web 工具有效地工作是不可能的，尤其在刚来的几个月里。要成为一个高效的使用者，了解我们 CMS 和网站的关键概念还远远不够，还需要了解它们是如何实现的。

比如说，将缓存内容的概念（这应该完全是系统内部的技术优化）暴露给编辑人员；编辑们要将内容放置在缓存中以确保内容已准备好，还需要理解缓存工作流以用 CMS 工具诊断和解决问题。这显然是对编辑人员不合理的要求。

1.2. 开发人员的问题

概念上的不匹配也体现在技术方面。举例来说，CMS 有一个概念是“制品”，这完全是所有开发人员每天工作的核心。以前团队中的一个人坦言，在足足九个月之后他才认识到这些“制品”其实就是网页。围绕“制品”形成的含义模糊的语言和软件越来越多，这些东西让他工作的真正性质变得晦涩难懂。

再举一个例子，生成我们内容的 RSS 订阅特别耗时。尽管各个版块的主页都包含一个清晰的列表，里面有主要内容和附加内容，但底层软件并不能对两者予以区分。因此，从页面抽取 RSS 订阅的逻辑是混乱的，比如“在页面上获取每个条目，如果它的布局宽大约一半儿、长度比平均长度长一些，那它可能是主要内容，这样我们就能抽取链接、将其作为一个订阅”。

很显然，对我们来说，人们对他们工作（开始、网页和 RSS 订阅）及其如何实现（缓存工作

流、“制品”、混乱逻辑）的认识之间的分歧给我们的效益造成了明显而惨重的影响。

2. 从DDD开始

本部分阐述了我们使用 DDD 的场景：为什么选择它，它在系统架构中所处的位置，还有最初的领域模型。在后面的章节中，我们会看一下如何把最初的领域知识传播给扩充的团队，如何演进模型，以及如何以此为中心来演进我们的编码技术。

2.1. 选择DDD

DDD 所倡导的首要方面就是统一一致的语言，以及在代码中直接体现用户自己的概念。这能有效地解决前面提及的概念上的不匹配问题。单独看来，这是一个有价值的见解，但其本身价值或许并不比“正确使用面向对象技术”多很多。

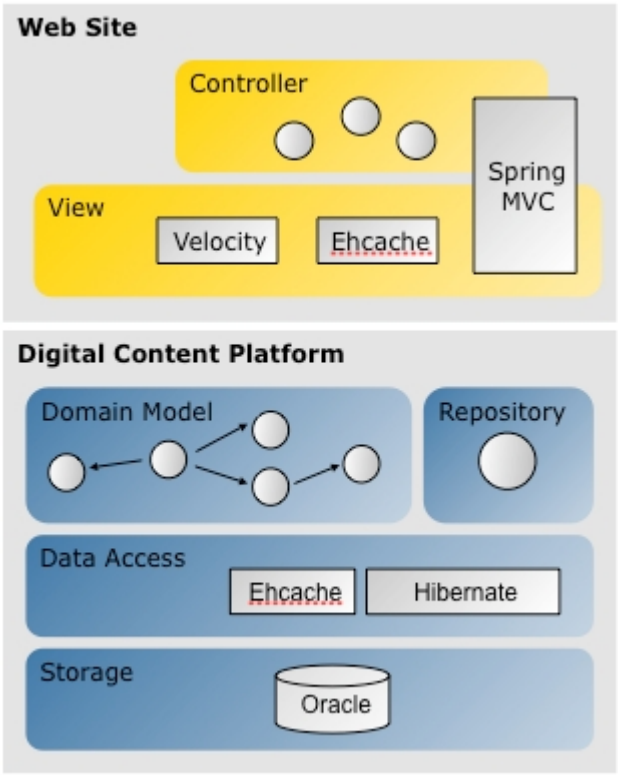
使其深入的是 DDD 引入的技术语言和概念：实体、值对象、服务、资源库等。这确保了在处理非常大的项目时，我们的大型开发团队有可能一致地进行开发——长远来看，这对维护质量是必不可少的。甚至在废弃我们更底层的代码时（我们稍后会进行演示），统一的技术语言能让我们恢复到过去、改进代码质量。

2.2. 在系统中嵌入领域模型

本节显示了 DDD 在整个系统架构中的地位。

我们的系统逐渐建立了三个主要的组件：渲染应用的用户界面网站；面向编辑、用于创建和管理内容的应用程序；跟系统交互数据的供稿系统。这些应用都是基于 Spring 和 Hibernate 构建的 Java Web 应用，并使用 Velocity 作为我们的模板语言。

我们可以看下这些应用的布局：



Hibernate 层提供数据访问，使用 EHCache 作为 Hibernate 的二级缓存。模型层包含领域对象和资源库，服务则处于其上一层。在此之上，我们有 Velocity 模板层，它提供页面渲染逻辑。最顶层则包含控制器，是应用的入口点。

看一下这个应用的分层架构，仅仅将模型当做是应用的一个自包含的层是很有意思的。这个想法大致正确，但模型层和其它层之间还是有一些细微的差别：由于我们使用领域驱动设计，所以我们需要统一语言，不仅要在我们谈论领域时使用，还要在应用的任何地方使用。模型层的存在不仅是为了从渲染逻辑中分离业务逻辑，还要为使用的其它层提供词汇表。

另外，模型层可作为代码的独立单元进行构建，而且可以作为 JAR 包导入到依赖于它的许多应用中。其它任何层则不是这样。对构建和发布应用来说，这意味着：在我们基础设施的模型层改变代码一定是跨所有应用的全局变化。我们可以在前端的网站中改变 Velocity 模板，只用部署前端应用就可以，管理系统或供稿系统则不用。如果我们在领域模型中改变了一个对象的逻辑，我们必须更新所有依赖于模型的应用，因为我们只有（而且期望只有）一个领域视图。

这有一种危害，就是领域建模的这种方法可能会导致单一模型，如果业务领域非常庞大，改变起来的代价会非常昂贵。我们认识到了这一点，因此随着领域不断增长，我们必须确保该层没有变得太过笨重。目前，虽然领域层也是相当庞大和复杂的，但是这还没有带来什么问题。在敏捷环境中工作，我们希望无论如何每两周都要推出所有系统的最新变化。但我们持

续关注着该层代码改变的成本。如果成本上升到不能接受的程度，我们可能会考虑将单一模型细分成多个更小的模型，并在每个子模型之间给出适配层。但是我们在项目开始时没有这样做，我们更偏向于单一模型的简单性，而不是用多个模型时必须解决的更为复杂的依赖管理问题。

2.3. 早期的领域建模

在项目初期，大家在键盘上动手开始编写代码之前，我们就决定让开发人员、QA、BA、业务人员在同一个房间里一起工作，以便项目能够持续。在这个阶段我们有一个由业务人员和技术人员组成的小型团队，而且我们只要求有一个稳妥的初次发布。这确保了我们的模型和过程都相当简单。

我们的首要目标是让编辑（我们业务代表的关键组成部分）就项目最初迭代的期望有一个清楚的认识。我们跟编辑们坐在一起，就像一个整体团队一样，用英语与他们交流想法，允许各个功能的代表对这些想法提出质疑和澄清，直到他们认为我们正确理解了编辑需要的内容。

我们的编辑认为，项目初期优先级最高的功能是系统能生成网页，这些网页能显示文章和文章分类系统。

他们最初的需求可归纳为：

- 我们应该能将一篇文章和任何给定的 URL 关联起来。
- 我们要能改变选定的不同模板渲染结果页面的方式。
- 为了管理，我们要将内容纳入宽泛的版面，也就是新闻、体育、旅游。
- 系统必须能显示一个页面，该页面包含指向特定分类中所有文章的链接。

我们的编辑需要一种非常灵活的方式来对文章进行分类。他们采用了基于关键字的方法。每个关键字定义了一个与内容相关的主题。每篇文章可以和很多关键字关联，因为一篇文章可以有很多主题。

我们网站有很多编辑，每个人负责不同版块的内容。每个版块都要求有自己导航和特有关键字的所有权。

从编辑使用的语言来看，我们似乎在领域中引入了一些关键实体：

- **页面**：URL 的拥有者。负责选择模板来渲染内容。

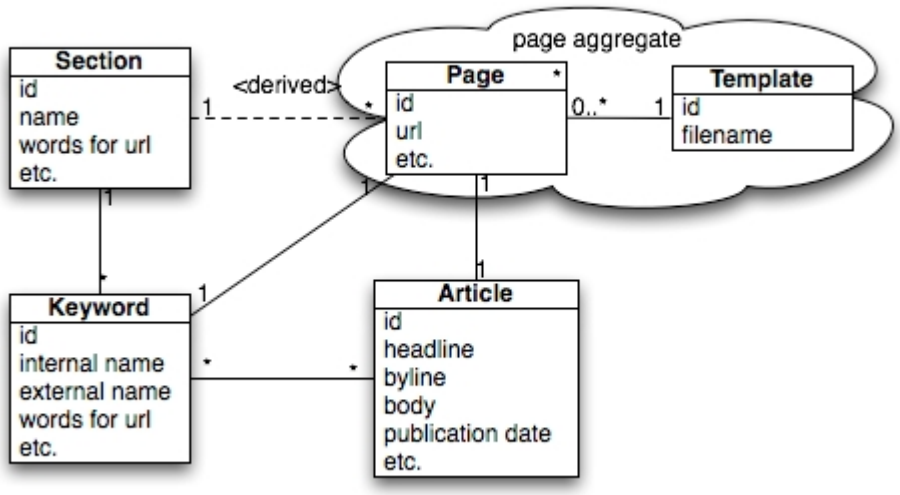
- **模板**：页面布局，任何时候都有可能改变。技术人员将每个模板实现为磁盘上的一个 Velocity 文件。
- **版块**：页面更宽泛的分类。每个版块有一个编辑，还有对其中页面共同的感官。新闻、旅游和商业都是版块的例子。
- **关键字**：描述存在于版块中主题的方式。关键字过去用于文章分类，现在则驱动自动导航。这样它们将与某个页面关联，关于给定主题所有文章的自动页面也能生成。
- **文章**：我们能发布给用户的一段文本内容。

提取这些信息后，我们开始对领域建模。项目早期做出的一个决定：编辑拥有领域模型并负责设计，技术团队则提供协助。对过去不习惯这种技术设计的编辑来说，这是相当大的转变。我们发现，通过召开由编辑、开发人员、技术架构师组成的研习会，我们能用简单、技术含量较低的方法勾画出领域模型，并对它持续演进。讨论模型的范围，使用钢笔、档案卡和 Blu-Tak 绘制备选解决方案。每个候选模型都会进行讨论，技术团队把设计中的每个细节含义告诉给编辑。

尽管过程在最初相当缓慢，但很有趣。编辑发现这非常容易上手；他们能信手拈来、提出对象，然后及时从开发人员那里获得反馈，以知道生成的模型是否满足了他们的需求。对于编辑们能在过程中迅速掌握技术的要领，技术人员都感到很惊喜，而且所有人都对生成的系统是否能满足客户需求很有信心。

观察领域语言的演变也很有趣。有时文章对象会被说成是“故事”。显然对于同一实体的多个名称，我们并没有一种统一语言，这是一个问题。是我们的编辑发现他们描述事物时没有使用统一的语言，也是他们决心称该对象为文章。后来，任何时间有人说“故事”，就会有人说：“你的意思不是文章吗？”在设计统一语言时，持续、公共的改进过程是种很强大的力量。

我们的编辑最初设计生成的模型是这样的：



由于并非所有的团队成员都参与了该过程的所有阶段，所以我们需要向他们介绍工作进展，并将这些全部展现在了墙上。然后开发人员开始了敏捷开发之旅，而且由于编辑和开发人员、BA、QA 都在一起工作，任何有关模型及其意图的问题都能在开发过程的任何时候获得第一手的可靠信息。

经过几次迭代之后系统初具形态，我们还建立工具来创建和管理关键字、文章和页面。随着这些内容的创建，编辑们很快掌握了它们，并且给出了一些修改建议。大家普遍认为这一简单的关键模型能正常运行，而且可以继续下去、形成网站初次发布的基础。

3. 增量计划中的DDD过程

首次发布之后，我们的项目团队伴随着技术人员和业务代表们的成长，一起取得了进步，打算演进领域模型。很显然，我们需要一种有组织的方式来为领域模型引入新的内容、进行系统的演进。

3.1. 新员工入门

DDD 是入门过程中的核心部分。非技术人员在整个项目生命周期内都会加入项目，因为工作计划是横跨各个编辑领域的，反过来，在恰当的时机我们也会引入版面编辑。技术人员很容易就能加入项目，因为我们持续不断地雇用新员工。

我们的入门过程包括针对这两类人的 DDD 讲习，尽管细节不同，但高层次的议题涵盖两个相同的部分：DDD 是什么，它为什么重要；领域模型本身的特定范围。

描述 DDD 时我们强调的最重要的内容有：

- 领域模型归业务代表所有。这就要从业务代表的头脑里抽象概念，并将这些概念嵌入到软件中，而不能从软件的角度思考，并试图影响业务代表。
- 技术团队是关键的利益相关者。我们将围绕具体细节据理力争。

覆盖领域模型的特定范围本身就很重要，因为它予以就任者处理项目中特定问题的真正工具。我们的领域模型中有几十个对象，所以我们只关注于高级别和更为明显的少数几个，在我们的情况中就是本文所讨论的各种内容和关键字概念。在这里我们做了三件事：

- 我们在白板上画出了概念及其关系，因此我们能给出系统如何工作的一个有形的表示。
- 我们确保每位编辑当场解释大量的领域模型，以强调领域模型不属于技术团队所有这一事实。
- 我们解释一些为了达到这一点而做出的历史变迁，所以就任者可以理解(a)这不是一成不变的，而是多变的，(b)为了进一步开发模型，他们可以在即将进行的对话中扮演什么样的角色。

3.2. 规划中的DDD

入门是必不可少的，不过在开始计划每次迭代的时候，知识才真正得以实践。

3.2.1 使用统一语言

DDD 强制使用的统一语言使得业务人员、技术人员和设计师能围坐在一起规划并确定具体任务的优先次序。这意味着有很多会议与业务人员有关，他们更接近技术人员，也更加了解技术过程。有一位同事，她先担任项目的编辑助理，然后成长为关键的决策者；她解释说，在迭代启动会议上她亲自去看技术人员怎样判断和（激烈地）评估任务，开始更多地意识到功能和努力之间的平衡。如果她不和技术团队共用一种语言，她就不会一直出席会议，也不会收获那些认识。

在规划阶段利用 DDD 时我们使用的两个重要原则是：

- 领域模型归属于业务；
- 领域模型需要一个权威的业务源。

领域模型的业务所有权在入门中就进行了解释，但在这里才发挥作用。这意味着技术团队的

关键角色是聆听并理解，而不是解释什么可能、什么不可能。需求抽象要求将概念性的领域模型映射到具体的功能需求上，并在存在不匹配的地方对业务代表提出异议或进行询问。接着，存在不匹配的地方要么改变领域模型，要么在更高层次上解决功能需求（“你想用此功能达成什么效果？”）。

对领域模型来说，权威的业务源是我们组织的性质所明确需要的。我们正在构建一个独立的软件平台，它需要满足很多编辑团队的需求，编辑团队不一定以同样的方式来看世界。Guardian 不实行许多公司实行的“命令和控制”结构；编辑台则有很多自由，也能以他们认为合适的方式去开发自己的网站版面，并设定预期的读者。因此，不同的编辑对领域模型会有略微不同的理解和观点，这有可能会破坏单一的统一语言。我们的解决办法是确定并加入业务代表，他们在整个编辑台都有责任。对我们来说，这是生产团队，是那些处理日常构建版面、指定布局等技术细节的人。他们是文字编辑依赖的超级用户，作为专家工具建议，因此技术团队认为他们是领域模型的持有者，而且他们保证软件中大部分的一致性。他们当然不是唯一的业务代表，但他们是与技术人员保持一致的人员。

3.2.2 与 DDD 一起计划的问题

不幸的是，我们发现了在计划过程中应用 DDD 特有的挑战，尤其是在持续计划的敏捷环境中。这些问题是：

- 本质上讲，我们正在将软件写入新的、不确定商业模式中；
- 绑定到一个旧模型；
- 业务人员“入乡随俗”。

我们反过来讨论下这些问题……

Eric Evans 撰写关于创建领域模型的文章时，观点是业务代表的脑海中存在着一个模型，该模型需要提取出来；即便他们的模型不明确，他们也明白核心概念，而且这些概念基本上能解释给技术人员。然而在我们的情况中，我们正在改变我们的模型——事实上是在改变我们的业务——但并不知道我们目标的确切细节。（马上我们就会看到这一点的具体例子。）某些想法显而易见，也很早就建立起来了（比如我们会有文章和关键字），但很多并不是这样（引入页面的想法还有一些阻力；关键字如何关联到其它内容则完全是各有各的想法）。我们的教科书并没有提供解决这些问题的指南。不过，敏捷开发原则则可以：

- 构建最简单的东西。尽管我们无法在早期解决所有的细节，但通常能对构建下一个有用的功能有足够的理解。

- 频繁发布。通过发布此功能，我们能看到功能如何实际地运转。进一步的调整和进化步骤因此变得最为明显（不可避免，它们往往不是我们所预期的）。
- 降低变化的成本。利用这些不可避免的调整和进化步骤，减少变化的成本很有必要。对我们来说这包括自动化构建过程和自动化测试等。
- 经常重构。经过几个演进步骤，我们会看到技术债务累积，这需要予以解决。

与此相关的是第二个问题：与旧模型有太多的精神联系。比如说，我们的遗留系统要求编辑和制作人员单独安排页面布局，而新系统的愿景则是基于关键字自动生成页面。在新系统里，Guantánamo Bay 页面无需任何人工干预，许多内容会给出 Guantánamo Bay 关键字，仅简单地凭借这一事实就能显示出来。但结果却是，这只是由技术团队持有的过度机械化的愿景，技术团队希望减少体力劳动和所有页面的持续管理。相比之下，编辑人员高度重视人的洞察力，他们带入过程的不仅有记述新闻，还有展现新闻；对他们来说，为了突出最重要的故事（而不仅仅是最新的），为了用不同的方法和灵敏性（比如 9·11 和 Web 2.0 报导）区别对待不同的主题，个人的布局是很有必要的。

对这类问题没有放之四海而皆准的解决办法，但我们发现了两个成功的关键：专注于业务问题，而不是技术问题；铭记“创造性冲突”这句话。在这里的例子里，见解有分歧，但双方表达了他们在商业上的动机后，我们就开始在同一个环境里面工作了。该解决方案是创造性的，源于对每个人动机的理解，也因此解决了大家了疑虑。在这种情况下，我们构建了大量的模板，每个都有不同的感觉和影响等，编辑可以从中选择并切换。此外，每个模板的关键区域允许手动选择显示的故事、页面的其余部分自动生成内容（小心不要重复内容），对于该手动区域，如果管理变得繁重，就可以随时关闭，从而使网页完全自动化。

我们发现的第三个挑战是随着业务人员“入乡随俗”，也就是说他们已深深融入技术且牵涉到了要点，以至于他们会忘记对新使用系统的内部用户来说，系统该是什么样。当业务代表发现跟他们的同事很难沟通事情如何运转，或者很难指出价值有限的功能时，就有危险的信号了。Kent Beck 在《解析极限编程》第一版中说，现场客户与技术团队直接交互绝不会占用他们很多的时间，通过强调这一点，就可以保证在现场的客户。但我们在与有几十个开发人员、多名 BA、多名 QA 的团队一起工作时，我们发现即使有三个全职的业务代表，有时也是不够的。由于业务人员花费了太多的时间与技术人员在一起，以至他们与同事失去联系成为真正的问题。这些都是人力解决方案带来的人力问题。解决方案是要提供个人备份和支持，让新的业务人员轮流加入团队（可能从助理开始着手进行，逐步成长为关键决策角色），允许代表有时间回归他们自己的核心工作，比如一天、一周等。事实上，这还有一个附加的好处，就是能让更多的业务代表接触到软件开发，还可以传播技巧和经验。

4. 进化的领域模型

在本章中，我们看看模型在方案的后期是如何演进的。

4.1. 演进第一步：超越文章

首次发布后不久，编辑要求系统能处理更多的内容类型，而非只有文章一类。尽管这对我们来说毫不稀奇，但在我们构建模型的第一个版本时，我们还是明确决定对此不予考虑太多。

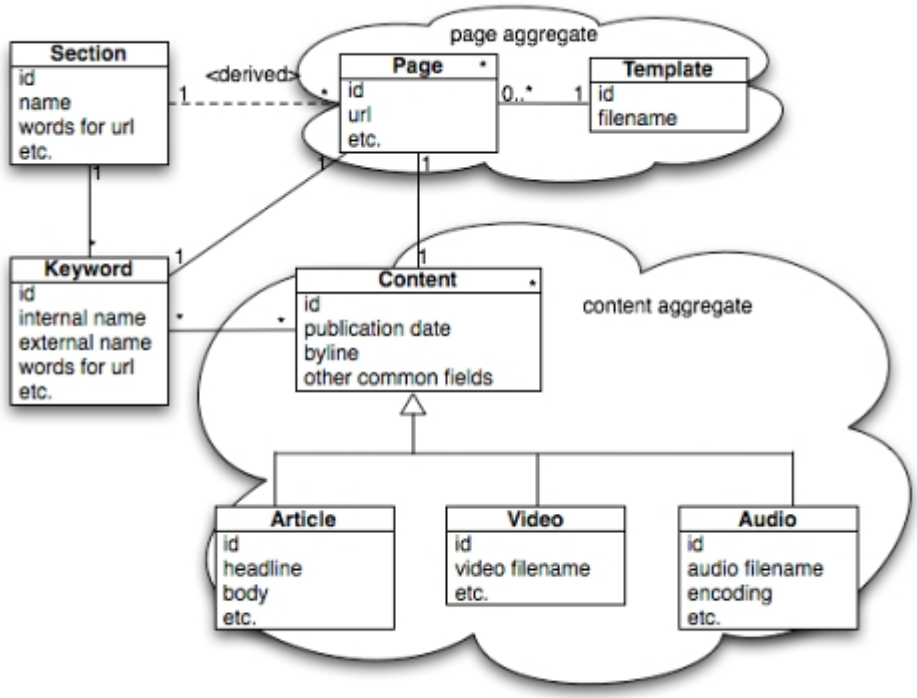
这是个关键点：我们关注整个团队能很好地理解小规模、可管理块中的模型和建模过程，而不是试图预先对整个系统进行架构设计。随着理解的深入或变化，稍后再改变模型并不是个错误。这种做法符合 YAGNI 编码原则（你不会需要它），因为该做法防止开发人员引入额外的复杂度，因而也能阻止 Bug 的引入。它还能让整个团队安排时间去对系统中很小的一块达成共识。我们认为，今天产出一个可工作的无 Bug 系统要比明天产出一个完美的、包括所有模型的系统更为重要。

我们的编辑在下一个迭代中要求的内容类型有音频和视频。我们的技术团队和编辑再次坐在一起讨论了领域建模过程。编辑先跟技术团队谈道，音频和视频很显然与文章相似：应该可以将视频或音频放在一个页面上。每个页面只允许有一种内容。视频和音频可以通过关键字分类。关键字可以属于版面。编辑还指明，在以后的迭代中他们会添加更多类型的内容，他们认为现在该是时候去理解我们应该如何随着时间的推移去演进内容模型。

对我们的开发人员来说，很显然编辑想在语言中明确引入两个新条目：音频和视频。音频、视频和文章有一些共同点：它们都是内容类型，这一点也很明确。我们的编辑并不熟悉继承的概念，所以技术团队可以给编辑讲解继承，以便技术团队能正确表述编辑所看到的模型。

这里有一个明显的经验：通过利用敏捷开发技术将软件开发过程细分为小的块，我们还能使业务人员的学习曲线变得平滑。久而久之他们能加深对领域建模过程的理解，而不用预先花费大量的时间去学习面向对象设计所有的组件。

这是我们的编辑根据添加的新内容类型设计的模型。



这个单一的模型演变是大量更细微的通用语言演进的结果。现在我们有三个外加的词 :音频、视频和内容；我们的编辑已经了解了继承，并能在以后的模型迭代中加以利用；对添加新的内容类型，我们也有了以后的扩展策略，并使其对我们的编辑来说是简单的。如果编辑需要一个新的内容类型，而这一新的内容类型与我们已有的内容类型相同、在页面和关键字之间有大致相同的关系，那编辑就能要求开发团队产出一个新的内容类型。作为一个团队，我们正通过逐步生成模型来提高效率 ,因为我们的编辑不会再详细检查漫长的领域建模过程去添加新的内容类型。

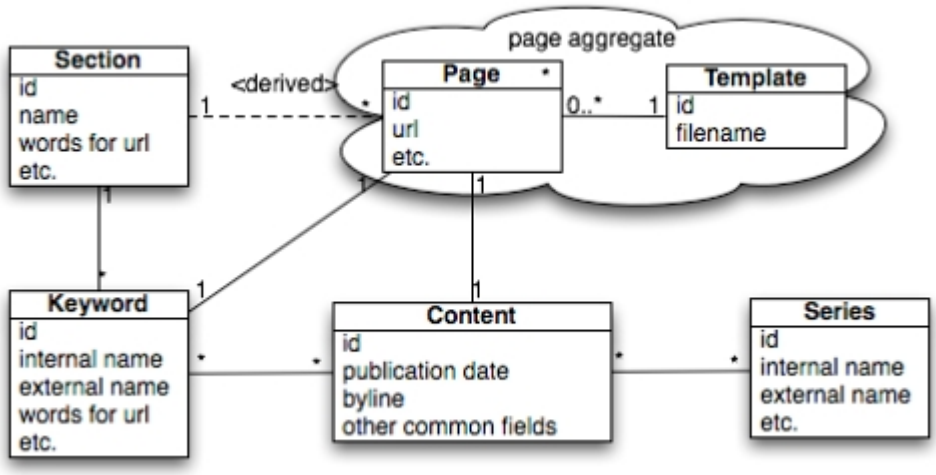
4.2. 演进第二步

由于我们的模型要扩展到包括更多的内容类型，它需要更灵活地去分类。我们开始在领域模型中添加额外的元数据，但编辑的最终意图是什么还不是非常清楚。然而这并不让我们太过担忧 ,因为我们对元数据进行建模的方法与处理内容的方法一样 ,将需求细分为可管理的块，将每个添加到我们的领域中。

我们的编辑想添加的第一个元数据类型是系列这一概念。系列是一组相关的内容，内容则有一个基于时间的隐含顺序。在报纸中有很多系列的例子，也需要将这一概念解释为适用于 Web 的说法。

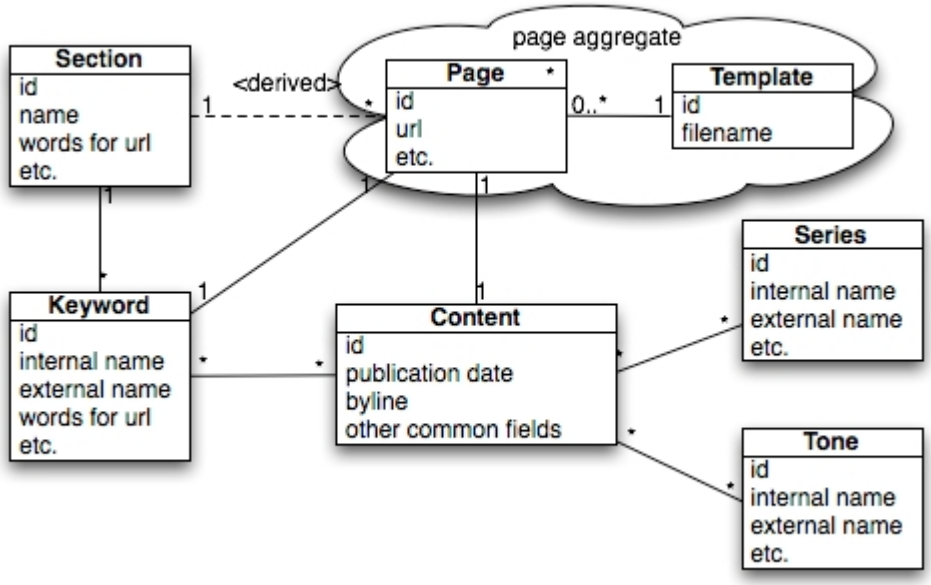
我们对此的初步想法非常简单。我们将系列添加为一个领域对象，它要关联到内容和页面。这个对象将用来聚集与系列关联的内容。如果读者访问了一种内容，该内容属于某个系列，我们就能从页面链接到同一系列中的前一条和后一条内容。我们还能链接到并生成系列索引页面，该页面可以显示系列中的所有内容。

这里是编辑所设计的系列模型：



与此同时，我们的编辑正在考虑更多的元数据，他们想让这些元数据与内容关联。目前关键字描述了内容是关于什么的。编辑还要求系统能根据内容的基调对内容进行不同的处理。不同基调的例子有评论、讣告、读者供稿、来信。通过引入基调，我们就可以将其显示给读者，让他们找到类似的内容（其它讣告、评论等）。这像是除关键字或系列外另一种类型的关系。跟系列一样，基调可以附加到一条内容上，也能和页面有关系。

这里是编辑针对基调设计的模型：



完成开发后，我们有了一个能根据关键字、系列或基调对内容进行分类的系统。但编辑对达到这一点所需的技术工作量还有一些关注点。他们在我们下次演进模型时向技术团队提出了这些关注点，并能提出解决方案。

4.3. 演进第三步：重构元数据

模型的下一步演进是我们的编辑想接着添加类似于系列和基调的内容。我们的编辑想添加带有贡献者的内容这一概念。贡献者是创建内容的人，可能是文章的作者，或者是视频的制作人。跟系列一样，贡献者在系统中有一个页面，该页面会自动聚集贡献者制作的所有内容。

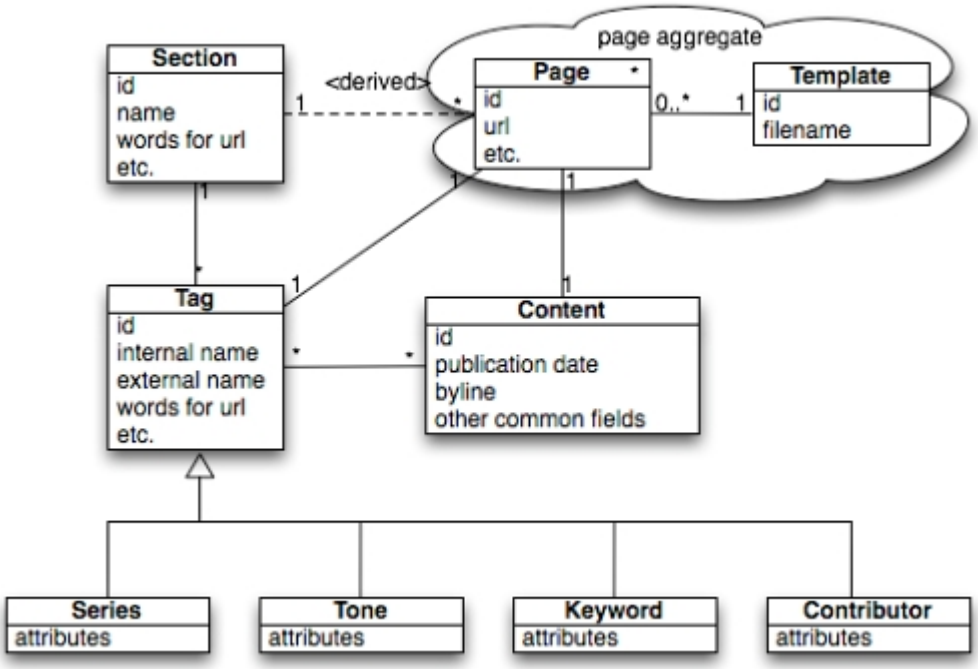
编辑还看到了另一个问题。他们认为随着系列和基调的引入，他们已经向开发人员指明了大量非常相似的细节。他们要求构建一个工具去创建系列，构建另一个工具去创建基调。他们不得不指明这些对象如何关联到内容和页面上。每次他们都发现，他们在为这两种类型的领域对象指定非常相似的开发任务；这很浪费时间，还是重复的。编辑更加关注于贡献者，还有更多的元数据类型会加入进来。这看起来又要让编辑再次指明、处理大量昂贵的开发工作，所有这些都非常相似。

这显然成为一个问题。我们的编辑似乎已经发现了模型的一些错误，而开发人员还没有。为什么添加新的元数据对象会如此昂贵呢？为什么他们不得不一遍又一遍地去指定相同的工作呢？我们的编辑问了一个问题，该问题是“这仅仅是‘软件开发如何工作’，还是模型有问题？”技术团队认为编辑熟悉一些事情，因为很显然，他们理解模型的方式与编辑不同。我们与编辑一起召开了另一个领域建模会议，试图找出问题所在。

在会议上我们的编辑建议，所有已有的元数据类型实际上源于相同的基本思想。所有的元数据对象（关键字、系列、基调和贡献者）可以和内容有多对多的关系，而且它们都需要它们自己的页面。（在先前的模型版本中，我们不得不知道对象和页面之间的关系）。我们重构了模型，引入了一个新的超类——Tag（标签），并作为其它元数据的超类。编辑们很喜欢使用“超类”这一技术术语，将整个重构称为“Super-Tag”，尽管最终也回到了现实。

由于标签的引入，添加贡献者和其它预期的新元数据类型变得很简单，因为我们能够利用已有的工具功能和框架。

我们修订后的模型现在看起来是这样的：



我们的业务代表在以这种方式考虑开发过程和领域模型，发现这一点非常好，还发现领域驱动设计有能力促进在两个方向都起作用的共同理解：我们发现技术团队对我们正努力解决的业务问题有良好且持续的理解，而且出乎意料，业务代表能“洞察”开发过程，还能改变这一过程以更好地满足他们的需求。编辑们现在不仅能将他们的需求翻译为领域模型，还能设计、检查领域模型的重构，以确保重构能与我们目前对业务问题的理解保持同步。

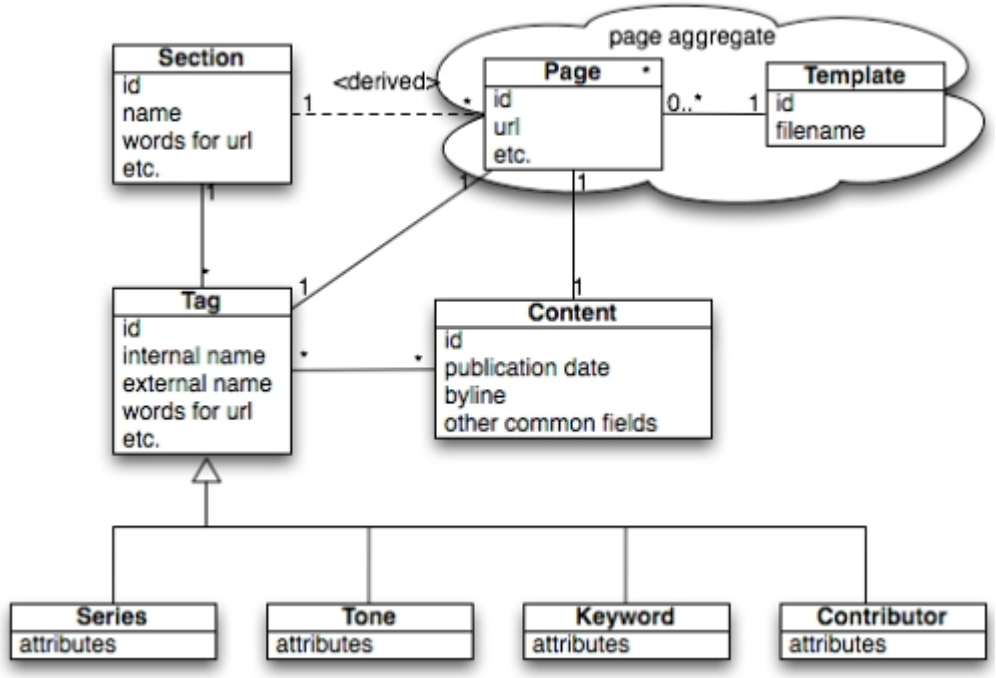
编辑规划领域模型重构并成功执行它们的能力是我们领域驱动设计 guardian.co.uk 成功的一个关键点。

5. 代码级别的演进

前面我们看了领域模型方面的进化。但 DDD 在代码级别也有影响，不断变化的业务需求也意味着代码要有变化。现在我们来看看这些变化。

5.1. 构建模型

在构建领域模型时，要确认的第一件事就是领域中出现的聚集。聚集可认为是相关对象的集合，这些对象彼此相互引用。这些对象不应该直接引用其它聚集中的其它对象；不同聚集之间的引用应该由根聚集来完成。



看一下我们在上面定义的模型示例，我们开始看到对象成形。我们有 Page 和 Template 对象，它们结合起来能给 Web 页面提供 URL 和观感。由于 Page 是系统的入口点，所以在这里 Page 就是根聚集。

我们还有一个聚集 Content，它也是根聚集。我们看到 Content 有 Article、Video、Audio 等子类型，我们认为这些都是内容的子聚集，核心的 Content 对象则是根聚集。

我们还看到形成了另一个聚集。它是元数据对象的集合：Tag、Series、Tone 等。这些对象组成了标签聚集，Tag 是根聚集。

Java 编程语言提供了理想的方式来对这些聚集进行建模。我们可以使用 Java 包来对每个聚集进行建模，使用标准的 POJO 对每个领域对象进行建模。那些不是根聚集、且只在聚集中使用的领域对象可以有包范围内使用的构造函数，以防它们在聚集外被构造。

上述模型的包结构如下所示（“r2”是我们应用套件的名称）：

```
com.gu.r2.model.page
com.gu.r2.model.tag
com.gu.r2.model.content
com.gu.r2.model.content.article
com.gu.r2.model.content.video
com.gu.r2.model.content.audio
```

我们将内容聚集细分为多个子包，因为内容对象往往有很多聚集特定的支持类（这里的简化图中没有显示）。所有以标签为基础的对象往往要更为简单，所以我们将它们放在了一个包里，而没有引入额外的复杂性。

不过不久之后，我们认识到上述包结构会给我们带来问题，我们打算修改它。看看我们前端应用的包结构示例，了解一下我们如何组织控制器，就能阐述清楚这一问题：

```
com.gu.r2.frontend.controller.page
com.gu.r2.frontend.controller.articl
```

这里看到我们的代码集要开始细分为片段。我们提取了所有的聚集，将其放入包中，但我们没有单独的包去包含与聚集相关的所有对象。这意味着，如果以后领域变得太大而不能作为一个单独的单元来管理，我们希望将应用分解，处理依赖就会有困难。目前这还没有真正带来什么问题，但我们要重构应用，以便不会有太多的跨包依赖。经过改进的结构如下：

```
com.gu.r2.page.model  (domain objects in the page aggregate)
com.gu.r2.page.controller (controllers providing access to
aggregate)
com.gu.r2.content.article.model
com.gu.r2.content.article.controller
...
etc
```

除了约定，我们在代码集中没有其它任何的领域驱动设计实施原则。创建注解或标记接口来标记聚集根是有可能的，实际上是争取在模型包锁定开发，减少开发人员建模时出错的几率。但实际上并不是用这些机械的强制来保证在整个代码集中都遵循标准约定，而是我们更多地依赖了人力技术，比如结对编程和测试驱动开发。如果我们确实发现已创建的一些内容违反了我们的设计原则（这相当少见），那我们会告诉开发人员并让他完善设计。我们还是喜欢这个轻量级的方法，因为它很少在代码集中引入混乱，反而提升了代码的简单性和可读性。这也意味着我们的开发人员更好地理解了为什么一些内容是按这种方式组织，而不是被迫去简单地做这些事情。

5.2. 核心DDD概念的演进

根据领域驱动设计原则创建的应用会具有四种明显的对象类型：实体、值对象、资源库和服务。在本节中，我们将看看应用中的这些例子。

5.2.1 实体

实体是那些存在于聚集中并具有标识的对象。并不是所有的实体都是聚集根，但只有实体才能成为聚集根。

开发人员，尤其是那些使用关系型数据库的开发人员，都很熟悉实体的概念。不过，我们发现这个看似很好理解的概念却有可能引起一些误解。

这一误解似乎跟使用 Hibernate 持久化实体有点儿关系。由于我们使用 Hibernate，我们一般将实体建模为简单的 POJO。每个实体具有属性，这些属性可以利用 setter 和 getter 方法进行存取。每个属性都映射到一个 XML 文件中，定义该属性如何持久化到数据库中。为了创建一个新的持久化实体，开发人员需要创建用于存储的数据库表，创建适当的 Hibernate 映射文件，还要创建有相关属性的领域对象。由于开发人员要花费一些时间处理持久化机制，他们有时似乎认为实体对象的目的仅仅只是数据的持久化，而不是业务逻辑的执行。等他们后来开始实现业务逻辑时，他们往往在服务对象中实现，而不是在实体对象本身中。

在下面（简化）的代码片段中可以看出此类错误。我们用一个简单的实体对象来表示一场足球赛：

```
public class FootballMatch extends IdBasedDomainObject
{
    private final FootballTeam homeTeam;
```

```
    private final FootballTeam awayTeam;
    private int homeTeamGoalsScored;
    private int awayTeamGoalsScored;

    FootballMatch(FootballTeam homeTeam, FootballTeam awayTeam) {
        this.homeTeam = homeTeam;
        this.awayTeam = awayTeam;
    }

    public FootballTeam getHomeTeam() {
        return homeTeam;
    }

    public FootballTeam getAwayTeam() {
        return awayTeam;
    }
    public int getHomeTeamScore() {
        return homeTeamScore;
    }

    public void setHomeTeamScore(int score) {
        this.homeTeamScore = score;
    }

    public void setAwayTeamScore(int score) {
        this.awayTeamScore = score;
    }
}
```

该实体对象使用 FootballTeam 实体去对球队进行建模，看起来很像使用 Hibernate 的开发人员所熟悉的对象类型。该实体的每个属性都持久化到数据库中，尽管从领域驱动设计的角度来说这个细节并不真的重要，我们的开发人员还是将持久化的属性提升到一个高于它们应该在的水平上去。在我们试图从 FootballTeam 对象计算出谁赢得了比赛的时候这一点就可以显露出来。我们的开发人员要做的事情就是造出另一种所谓的领域对象，就像下面所示：

```
public class FootballMatchSummary {


    public FootballTeam getWinningTeam(FootballMatch footballMatch)
{
        if(footballMatch.getHomeTeamScore() >
footballMatch.getAwayTeamScore()) {
            return footballMatch.getHomeTeam();
        }
        return footballMatch.getAwayTeam();
    }
 }
```

片刻的思考应该表明已经出现了错误。我们已经创建了一个 FootballMatchSummary 类，该类存在于领域模型中，但对业务来说它并不意味着什么。它看起来是充当了 FootballMatch 对象的服务对象，提供了实际上应该存在于 FootballMatch 领域对象中的功能。引起这一误解的原因好像是开发人员将 FootballMatch 实体对象简单地看成了是反映数据库中持久化信息，而不是解决所有的业务问题。我们的开发人员将实体考虑为了传统 ORM 意义上的实体，而不是业务所有和业务定义的领域对象。

不愿意在领域对象中加入业务逻辑会导致贫血的领域模型，如果不加以制止还会使混乱的服务对象激增——就像我们等会儿看到的一样。作为一个团队，现在我们来检视一下创建的服务对象，看看它们实际上是否包含业务逻辑。我们还有一个严格的规则，就是开发人员不能在模型中创建新的对象类型，这对业务来说并不意味着什么。

作为团队，我们在项目开始时还被实体对象给弄糊涂了，而且这种困惑也与持久化有关。在我们的应用中，大部分实体与内容有关，而且大部分都被持久化了。但当实体不被持久化，而是在运行时由工厂或资源库创建的话，有时候还是会混淆。

一个很好的此类例子就是“标签合成的页面”。我们在数据库中持久化了编辑创建的所有页面

的外观，但我们可以自动生成从标记组合（比如 USA+Economics 或 Technology+China）聚集内容的页面。由于所有可能的标记组合总数是个天文数字，我们不可能持久化所有的这些页面，但系统还必须能生成页面。在渲染标记组合页面时，我们必须在运行时实例化尚未持久化的新 Page 实例。项目初期我们倾向于认为这些非持久化对象与“真正的”持久化领域对象不同，也不像在对它们建模时那么认真。从业务观点看，这些自动生成的实体与持久化实体实际上并没有什么不同，因此从领域驱动设计观点来看也是如此。不论它们是否被持久化，对业务来说它们都有同样的定义，都不过是领域对象；没有“真正的”和“不是真正的”领域对象概念。

5.2.2 值对象

值对象是实体的属性，它们没有特性标识去指明领域中的内容，但表达了领域中有含义的概念。这些对象很重要，因为它们阐明了统一语言。

值对象阐述能力的一个例子可以从我们的 Page 类更详细地看出。系统中任何 Page 都有两个可能的 URLs。一个 URL 是读者键入 Web 浏览器以访问内容的公开 URL。另一个则是从应用服务器直接提供服务时内容依存的内部 URL。我们的 Web 服务器处理从用户那里传入的 URL，并将它转换为适合后端 CMS 服务器的内部 URL。

一种简化的观点是，在 Page 类中两个可能的 URL 都被建模为字符串对象：

```
public String getUrl();
public String getCmsUrl();
```

不过，这并没有什么特别的表现力。除了这些方法返回字符串这一事实之外，只看这些方法的签名很难确切地知道它们会返回什么。另外想象一下这种情况，我们想基于页面的 URL 从一个数据访问对象中加载页面。我们可能会有如下的方法签名：

```
public Page loadPage(String url);
```

这里需要的 URL 是哪一个呢？是公开的那个还是 CMS URL？不检查该方法的代码是不可能识别出来的。这也很难与业务人员谈论页面的 URL。我们指的到底是哪一个呢？在我们的模型中没有表示每种类型 URL 的对象，因此在我们的词汇里面也就没有相关条目。

这里还含有更多的问题。我们对内部 URL 和外部 URL 可能有不同的验证规则，也希望对它们执行不同的操作。如果我们没有地方放置这个逻辑，那我们怎么正确地封装该逻辑呢？控制 URLs 的逻辑一定不属于 Page，我们也不希望引入更多不必要的服务对象。

领域驱动设计建议的演进方案是明确地对这些值对象进行建模。我们应该创建表示值对象的

简单包装类，以对它们进行分类。如果我们这样做，Page 里面的签名就如下所示：

```
public Url getUrl();

public CmsPath getCmsPath();
```

现在我们可以在应用中安全地传递 CmsPath 或 Url 对象，也能用业务代表理解的语言与他们谈论代码。

5.2.3 资源库

资源库是存在于聚集中的对象，在抽象任何持久化机制时提供对聚集根对象实体的访问。这些对象由业务问题请求，与领域对象一起响应。将资源库看成是类似于有关数据库持久化功能的数据访问对象，而非存在于领域中的业务对象，这一点很不错。但资源库是领域对象：他们响应业务请求。资源库始终与聚集关联，并返回聚集根的实例。如果我们请求一个页面对象，我们会去页面资源库。如果我们请求一个页面对象列表来响应特定的业务问题，我们也会去页面资源库。我们发现一个很好的思考资源库的方式，就是把它们看成是数据访问对象集合之上的外观。然后它们就成为业务问题和数据传输对象的结合点，业务问题需要访问特定的聚集，而数据传输对象提供底层功能。

这里举一小段页面资源库的代码例子，我们来实际看下这个问题：

```
private final PageDAO<Page> pageDAO;
private final PagesRelatedBySectionDAO pagesRelatedBySectionDAO;

public PageRepository(PageDAO<Page> pageDAO,
        EditorialPagesInThisSectionDAO pagesInThisSectionDAO,
        PagesRelatedBySectionDAO pagesRelatedBySectionDAO) {
    this.pageDAO = pageDAO;
    this.pagesRelatedBySectionDAO = pagesRelatedBySectionDAO;
}

public List<Page>
getAudioPagesForPodcastSeriesOrderedByPublicationDate(Series
series, int maxNumberOfPages) {
```

```
    return
pageDAO.getAudioPagesForPodcastSeriesOrderedByPublicationDate(ser
ies, maxNumberOfPages);
 }


 public List<Page> getLatestPagesForSection(Section section, int
maxResults) {
    return
pagesRelatedBySectionDAO.getLatestPagesForSection(section,
maxResults);
 }
```

我们的资源库有业务请求：获取 PublicationDate 请求的特定播客系列的页面。获取特定版面的最新页面。我们可以看看这里使用的业务领域语言。它不仅仅是一个数据访问对象，它本身就是一个领域对象，跟页面或文章是领域对象一样。我们花了一段时间才明白，把资源库看成是领域对象有助于我们克服实现领域模型的技术问题。我们可以在模型中看到，标签和内容是一种双向的多对多关系。我们使用 Hibernate 作为 ORM 工具，所以我们对其进行了映射，Tag 有如下方法：

```
public List<Content> getContent();
```

Content 有如下方法：

```
public List<Tag>  getTags();
```

尽管这一实现跟我们的编辑看到的一样，是模型的正确表达，但我们有了自己的问题。对开发人员来说，代码可能会编写成下面这样：

```
 if(someTag.getContent().size() == 0){
    ... do some stuff
 }
```

这里的问题是，如果标签关联有大量的内容（比如“新闻”），我们最终可能会往内存中加载几十万的内容条目，而只是为了看看标记是否包含内容。这显然会引起巨大的网站性能和稳定性问题。

随着我们演进模型、理解了领域驱动设计，我们意识到有时候我们必须要注重实效：模型的某些遍历可能是危险的，应该予以避免。在这种情况下，我们使用资源库来用安全的方式解

决问题，会为系统的性能和稳定性牺牲模型个别的清晰性和纯洁性。

5.2.4. 服务

服务是通过编排领域对象交互来处理业务问题执行的对象。我们所理解的服务是随着我们过程演进而演进最多的东西。首要问题是，对开发人员来说创建不应该存在的服务相当容易；他们要么在服务中包含了本应存在于领域对象中的领域逻辑，要么扮演了缺失的领域对象角色，而这些领域对象并没有作为模型的一部分去创建。项目初期我们开始发现服务开始突然涌现，带着类似于 ArticleService 的名字。这是什么呀？我们有一个领域对象叫 Article；那文章服务的目的是什么？检查代码时，我们发现该类似乎步了前面讨论的 FootballMatchSummary 的后尘，有类似的模式，包含了本该属于核心领域对象的领域逻辑。为了对付这一行为，我们对应用中的所有服务进行了代码评审，并进行重构，将逻辑移到适当的领域对象中。我们还制定了一个新的规则：任何服务对象在其名称中必须包含一个动词。这一简单的规则阻止了开发人员去创建类似于 ArticleService 的类。取而代之，我们创建 ArticlePublishingService 和 ArticleDeletionService 这样的类。推动这一简单的命名规范的确帮助我们将领域逻辑移到了正确的地方，但我们仍要求对服务进行定期的代码评审，以确保我们在正轨上，以及对领域的建模接近于实际的业务观点。

6. 演进架构中DDD的一些教训

尽管面临挑战，但我们发现了在不断演进和敏捷的环境中利用 DDD 的显著优势，此外我们还总结了一些经验教训：

- 你不必理解整个领域来增加商业价值。你甚至不需要全面的领域驱动设计知识。团队的所有成员差不多都能在他们需要的任何时间内对模型达成一个共同的理解。
- 随着时间的推移，演进模型和过程是可能的，随着共同理解的提高，纠正以前的错误也是可能的。

我们系统的完整领域模型要比这里描述的简化版本大很多，而且随着我们业务的扩展在不断变化。在一个大型网站的动态世界里，创新永远发生着；我们始终要保持领先地位并有新的突破，对我们来说，有时很难在第一次就得到正确的模型。事实上，我们的业务代表往往想尝试新的想法和方法。有些人会取得成果，其他人则会失败。是逐步扩展现有领域模型——甚至在现有领域模型不再满足需求时进行重构——的业务能力为 guardian.co.uk 开发过程中遇到的大部分创新提供了基础。

7. 附录：具体示例

为了了解我们的领域模型如何生成真实的结果，这里给出了一个例子，先看单独的内容……

- 有关页面本身的一些音频内容
- 它有几个标签 :贡献者是 Matt Wells ;关键字包括“Digital Media”和“Radio” ;它属于“Media Talk”系列。这些标签都链接在页面上。
- Matt Wells有他自己的页面，而且有特定的模板
- “Digital Media”关键字也有其自己的页面，使用不用的模板
- “Media Talk”系列也有自己的页面
- 音频内容有一个页面，页面列出了所有的音频
- 标签组合页面可以完全实时生成。

8. 关于作者

Nik Silver是Guardian News & Media软件开发总监。他于 2003 年在公司引入敏捷软件开发，负责软件开发、前端开发和质量保证。Nik偶尔会在blogs.guardian.co.uk/inside上写Guardian技术工作相关的内容，并在他自己的站点niksilver.com上写更宽泛的软件问题。Matthew Wall是Guardian News & Media的软件架构师，深入研究敏捷环境下大型Web应用的开发。他目前最关心的是为guardian.co.uk开发下一代的Web平台。他在JAOO、 ServerSide、 QCon、 XTech和OpenTech上做过关于此及相关主题的各种演讲。

原文链接：http://www.infoq.com/cn/articles/ddd-evolving-architecture

相关内容：

- ODBMS.ORG新增持久化模式资源
- DCI：James O. Coplien和Trygve Reenskau提出的新架构方法
- “优秀的设计”意味着...?
- 并发与不可变性
- 使用单实例类来处理对象元信息

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 | Articles

设计者-开发者工作流中的迭代模式

作者 Doug Winnie 译者 罗小平

设计者-开发者工作流（designer-developer workflow）这个词已经流行了好几年。它描述了设计人员、开发人员在为 Web 或桌面应用创造交互体验过程中的关系，而没有表达出设计者、开发者之间的交互和协作。工作流这个术语让我们觉得这种关系是线性的，但实际上它不是。

在项目的整个生命周期中，我们会不停地为项目增加内容。项目本身从开始到结束可能是线性的，但项目参与者之间的协作不是。需要协作的项目，不会变成一个装配流水线；在项目结束之间，我们每个参与者都可能要往项目中不停地添加各种组件、功能片、代码和设计方案。这个过程是有机的，而且——更重要是迭代式的。作为 Adobe 的产品部门经理，我经常要和构建各种交互应用、内容的设计和开发人员一起工作。在此过程中我常听到的一点，就是“团队成员间的工作流程，是项目成功的关键；有效减少团队可能遇到的困难的方法是保证项目中每个人之间的清晰、高效沟通”。

在本文中，我将讨论一些可在开发和设计工作中应用的迭代模式，并说明如何利用这些迭代模式实现团队内的高效沟通。

迭代

所谓迭代开发，可定义为全程功能构建。比如在一个项目中，逐步增加各种新特性、交互体验、特性提升和新功能——每次只增加一项。我们以开发一个简单的游戏为例。游戏的一个关键特性是记录玩家的分数。而这个特性的最初版本，可能就只是通过调用计分系统的功能为用户增加分数。计分系统除了按给定点数未用户加分，没有其他任何功能，目的非常简单。在项目全过程中，我们可以逐步演进和完善计分系统，每次增加一个小的特性或功能，即每次就是一个迭代。对于这里的计分系统而言，我们可以通过迭代逐步增加的各项功能和特性大致有：

- 通过对计分系统的调用，实现每次按一个点数为某玩家计分
- 通过对计分系统的调用，实现每次按个数不定的一组点数为某玩家计分
- 分数达到预先固定的满分时，让计分系统通知游戏
- 在计分系统创建时，可自定义满分分数
- 为多个玩家创建多个计分系统实例

作为开发人员，我可以此作为开发某个组件的“特性路标图”，也可用于在完成独立的每个步骤后，标示下一步骤的工作。这样，即使项目周期很长，无论何时我们都能知道自己身在何处，从而可专注于每个独立特性的构建。以这种方式设计整个开发过程，可保证我们不会在未来某个时候遗漏任何工作。对于大型项目，我可能没有足够时间去定义一份完整的应用规格文档。但我知道大体上可分为哪些部分，并从中选择简单的入手，比如一套 Adobe Flex 组件，我可以先将这些最基础的组件开发出来。在接下来的每个独立的开发步骤中，我可以扩展已完成功能片的功能，并按我最终希望的形式和作用的要求将它们整合在一起。在此之后，我可以如法炮制对付这个大应用中的其他类似或差别不大的组件。如此这般，最终我可以构建出所有功能模块，然后将它们集成，从而形成整个应用。这些模块的组织集成也可以用迭代方法完成，通过事件、监听器，逐步在这些模块间构建出通讯桥梁。

设计

迭代如何应用于设计呢？最简单的答案是有两种应用途径——你或许不能立刻想到。

首先，在做系统的整体设计，我从基本构建块（building block）入手：在哪里实现导航？主要内容放在哪里？应用中的这个或那个功能安排什么地方？所有基本构建块一起组成了套件的整体框架（有关终端用户如何使用套件、应用或内容的结构、方法的总体设计）。

当你将系统的主内容块（如导航、部件 A/B/C）定下来后，设计过程就开始了。在这个过程中，每完成一个基本构建块的设计，就是一次迭代。在整个过程中，你可以逐步演进和完善自己的设计意图。第二个方法应用在我已经完成了应用的整体结构，准备在此基础上进行可视化设计的时候。此时用户界面已经确定，结构中每个离散的元素都可以取出来独立设计。在设计了整体结构后再设计其组成元素是非常重要的，因为你需要知道每个组成元素在整个大套件或应用中的运行环境。每个独立的功能模块与其他模块如何交互，将决定该套件或应用的用户界面设计是相对固定还是相对可变化。

比如前面举到的导航例子，我将迭代设计此组件，最开始使用其缺省状态，然后再扩展引入

其他元素。假设在我的游戏中需要一个菜单栏。在设计它时，我可以迭代式设计各种特性，具体可包括：

1. 设计初始导航状态，不支持鼠标交互。
2. 支持鼠标悬浮移动。
3. 支持鼠标悬浮移动和提示框。
4. 支持鼠标点击一级导航元素。
5. 支持二级导航。
6. 在二级导航中支持鼠标悬浮移动。
7. 实现完整导航结构。

在每一步骤中，组件（本例中即菜单）的设计，都需要考虑与导航结构协作；同时，它需工作在整个套件或应用的环境中，因此必须保证其设计和交互流程与该环境适配。

导航结构完成后，你就可以想办法对付下一个组件元素了。

回退

迭代式开发和设计的一大好处是当我走远或走歪的时候，可以清晰回溯到还原点纠正错误，继续前进。比如，我们现在需要游戏中的计分系统支持减分，就可以倒回去增加这项功能。对于更复杂的项目来说，即使某些基础性的东西需要变化，我们也能将项目回退可实现此变化的状态，然后修改、重新将它引入原模块，并修复任何可能出现的集成问题。以前面的导航结构为例，如果发现需要增加三级导航，我可以回到构建二级导航的地方，并添加三级导航。如果发现三级导航不能和该套件或应用的整体环境较好的协同工作，我可以回溯到更远点，将二级导航返工，然后再引入三级导航，并使其生效。

对于多数设计和开发人员而言，这些听起来有点老生常谈，但的确都是指导我们如何工作、如何向项目持续增加内容的基本方法。即使在软件开发领域之外的设计准则中，它往往也是适用的。例如视频编辑就是一个典型的迭代式设计实践。编辑人员每段时间只处理一个视频片段，然后是场景，再然后才是整个视频。从单独元素开始，逐步让其成长，容纳更多元素，这是我们在大型项目中采用的最基本的工作方法。

迭代离不开沟通

在迭代模式中，无论是设计人员还是开发人员，都会面临一个难题：每个成员如何与正在构建套件或应用中的小组中其他专业的成员实现有效沟通？

答案之一是在每步迭代中向全体成员广播信息——但无疑只有每次迭代在粒度上得到了充分划分，这种沟通才会产生作用。此外还要求它与项目存在相关性。当然在项目的当前时候，它不可能总是相关的；但在开发过程中的未来某个时候，它必定又是有价值的。这样，团队成员在整个过程中都可以获得他们需要的信息。灵活性，是迭代开发和设计的一大要点。团队所有成员都向迭代过程贡献自己的成果，因此他们的工作必须是灵活的、开放的，只有这样，过程产出的各种组件、设计方案和代码最终才能有效集成。要想迭代设计和开发最后取得成功，那么所有参与者就必须在项目中取得共识。很多时候，应用的设计和开发人员最开始都会认真制定详细规格书，并以此为基础开始工作。但不久，他们就会按照每个人自己的想法"私奔"了。多数情况下，规格书并不能容纳应用中全部设计和功能用例，因此在执行过程中，他们有为满足需求而自行改编的倾向。但问题是设计和开发彼此分离，互不依赖，那么最后碰头时，必然发现互不兼容，必须予以修改才能解决二者之间的差异。规定统一的沟通语言，是项目第一步。在项目开始的时候，设计和开发团队必须就项目的主要组件达成一致。详细规定这些组件的功能和设计或许并非必需，但在它们的主要方面达成一致至关重要。例如某应用程序包含一个菜单、某种聊天功能和一些对象。团队必须就应用中的命名规则、基本组件的作用、以及每个主要组件的主要目标达成一致。

完成了这些要素的定义后，团队成员就可以开始迭代工作了。开发和设计人员可以将这些定义作为他们不断开展迭代的基础，在每步工作的同时，规划出下一步骤。如果功能发生变化，可以在未来的迭代步骤做出调整以适应变化了的需求。如在一个迭代步骤中，不同部件间出现了冲突，参与者可以通过沟通解决这些问题并继续前进，无需将整个组件、套件或应用返工。

组织与管理

有多种技术和系统可帮助设计和开发人员组织和管理迭代。对开发人员而言，必不可少的工具就是代码版本控制系统，它负责将每个迭代过程形成的代码予以保存，供未来使用，开发人员可以根据要求返回到先前的任意迭代阶段。

对于设计人员而言，像 Adobe Version Cue 这样的设计档案管理系统，能为迭代开发提供重要的版本管理能力。此外，在团队范围内采用某种统一的文件命名规范也会大有好处。像

Subversion 这样的代码存储系统，也可用于设计迭代过程。对于团队来说，建立 Wiki、内部开发博客或其他类似的沟通工具，帮助团队成员自动实现信息的收集和分发，对整个团队的沟通效果也有极大帮助。不过，最关键的一点，还是无论你们选择什么工具，整个团队都必须要使用这些工具。否则，工具将不存在任何意义。如果一个团队成员只顾干自己的，不利用这些工具跟踪其他成员的迭代成果，他们最后开发出来的模块将被弃用，或在集成时出现问题。

总结

研究并和你的团队成员讨论文中提到的技术，并确定哪项技术最适用于你的团队，这是要做的第一步工作。要找到这项技术，需要打开心扉，仔细思考各种可能和以前可能从未入过你眼的工具。再次强调，这些技术的最终目标是让你的团队良好工作，就像我前面提到过的那样，团队成功沟通和协作，是项目走向成功的秘诀。若需了解设计人员-开发人员工作流的更多信息，请参考Fireworks Developer Center中design/development、iterative prototyping等相关文章，以及Adobe Labs的Flash Catalyst、 Flash Builder。

作者简介

Doug Winnie：Adobe Systems公司工作流产品部门经理，致力于Adobe产品、平台和技术间的流程协作。在加入Adobe之前，Doug负责过设计人员和用户体验开发组织的领导工作。他热衷于Flash、 Flex和Dreamweaver等Web应用和平台方面的开发工作。他的博客是：http://www.adobe.dougwinnie.com/。

原文链接：

http://www.infoq.com/cn/articles/designer-developer-workflow

相关内容：

- 敏捷应对“团队的五重机能障碍”
- 敏捷背后
- Steven "Doc" List 谈开放空间会议
- 借助信息化工作空间实现高效的团队自我管理
- 远程工作人员的Team Foundation Server

推荐文章 | Articles

收获面向服务

作者 Wil Leeuwis 译者 胡键

要想获得理想的结果，就值得常常进行回顾和自我反省："我们学到了什么？"我们正处在红皇后（译注：红皇后是小说《Through the Looking-Glass, and What Alice Found There》中的一个人物，请参考维基百科、中文资料 1、中文资料 2。该小说是《爱丽丝漫游奇境》的续集。）所讲的年代，有效地进行学习，并且因此常常要花些时间来从过去的教训中吸取经验，这样做比以前按时获得优质结果显得更有必要 ,这是一种似是而非的隽语。今天的难题由来已久，而今天的答案是：服务。因此我会把矛头指向服务。

模型和系统

模型是日常现实的简化表述。系统是一类特殊的模型，现实世界或其一部分在其中被建模成元素和元素之间的关系。在系统中，我们可以定义子系统（共享某 些特性的元素子集）和方面系统（aspect system）（共享某些特性的关系子集）。此外，所有这些都是递归的：一个系统集合完全也可以是一个系统。创立于十九世纪 40 年代的系统论提供了大量的 理论基础，随时可帮助 21 世纪的问题分析师和解决者在调查研究中去发现事情的本质[1]，[2]，[3]。

我们创建模型和系统的原因是为了更好地理解我们周围或者我们内部世界的某些方面或者局部。这种理解的获得是以领悟并理解我们所舍弃和抽取的部分和方 面为代价的。模型范围的选择要从我们的目标以及需要学习和交流的内容出发：制作模型是项有针对性的活动。数学模型有别于其他模型的重要一点就是：数学通过 定义公理来创造属于自己的世界。数学模型内的事物都是绝对清晰的，甚至在处理模糊逻辑时也是如此。在自然数的集合里，每个数字不是奇数就是偶数，不存在一 个偶数或多或少比另一个偶数更像偶数这样的情形。这和我们所熟知的世界差别太大了！

任何事物都是模糊的

在现实世界中，任何事物都在一定程度上是模糊的，而且如果不设法使其精确，你就不会意识到这一点。并且，事物的精确表述与我们平时想到的相距甚远，以至于在我们说出自己的想法时，你无法马上理解我们的真实意思[4]。

由于已经习惯于使用我们熟悉的数学抽象进行工作，我们常常会忘记这一点。并且，当我们忘记这一点的时候，我们会错误地把模型当成现实世界。数学太伟 大了。正是由于它，使得大量的现实世界问题得以解决。但是每一位热衷于使用数学对现实世界建模的工程师都知道，模型只是模型。由于现实事物都是模糊的，它 与模型之间存在偏差并不奇怪。正是由于模型的两面性让我的一位老师说出了这样的话“不要相信模型，但也不要忽视模型”。不要把模型误以为是现实世界，要意 识到那些模型忽略掉的事物：不要以后大惊小怪。总而言之：尽可能使用模型。它将帮助你来理解这个对凡人来说过于复杂的世界。

模型增加了复杂性？

模型一旦被描述出来后，它就成了这个世界的一部分。但是有点违反直觉和令人不悦的是：虽然制作模型的出发点是为了控制复杂度，但最终却得到了一个甚 至更复杂的世界。对于这个问题，有 3 点要说明。首先：它完全取决于你的观点。从全局观点来看，复杂性确实增加了。但是从局部看来，一个形式完好的模型肯定 能有助于控制复杂度。这正是“天下没有免费的午餐”规律的一个例子：局部获利是以全局损失为代价的，很多时候情况恰恰是这样。第二：为世界建模带来了二义 性。以汽车类实例化而得到的汽车对象来说，它当然不同于 GM 生产线上生产出的汽车。所以当现实世界和模型都在我们讨论范围内的时候，我们需要慎重地选择措 辞。第三：员工要为他们工作的公司负责，并且也要对整个社会负责：如果没有充分的理由，他们不应该增加复杂性。

服务

服务、面向服务、面向服务的架构。有旧、有新、有借、有蓝（译注：婚礼习俗。参见《有旧有新有借有蓝——百年婚礼习俗》）。 新瓶装旧酒还是真正的创新？在我看来这两种说法都对。在服务领域，只要我们能远离非此即彼的谬论，我们就会有很多收获。不要再试图去争辩它是CORBA再 世，或者说你早就拥有它，思路开阔些。因为有很多旧的、易理解的、可用于实践的理论能够帮助我们从服务世界的创新中汲取营养。仅仅出于文章的需要，让我们 构建一个模型来更加紧扣“服务”的概念。我们将在文章结尾推翻这个模型，这样对整体

复杂性就不会带来任何损失。

我提议的这个模型包含四个组件：内存、处理器、二者间的连接以及执行器。内存代表了世界上所有内部和机器可读的内存，同样的规则也被适用于处理器和 连接，因此后者还包含诸如互联网之类的事物。为了简单起见，执行器代表了世界上所有能够直接跟内存交互（即他们可以直接读写内存）的机器和人。鉴于处理 器、连接器和内存的组合就是一台机器，因此这个模型能够用于描述递归流程。

探索模型

我把详细设计该模型的任务留给了读者。例如，你可能会在内存或者处理器上安置可执行程序，并附以用于描述执行过程的特定结果。但即使在这个描述的不 严格的模型中，有一点也很清楚：19 世纪 50 年代汇编语言时代的子程序与久经考验的 Cobol 和 2008 年 Web 2.0+ 时代的 Web 服务，它们被调用的方式完全一样。

什么正在发生？这个流程是什么？先是有选择地收集一些数据。接着会有一个发往该世界一个不同参与者的请求，被收集到的数据往往作为请求参数一并提 供。接下来是代码执行，可能是直接执行，也可能过一段时间之后执行，其执行结果又作为另一请求的一部分被发往这个世界的另一参与者，以此类推。在执行完成 后，可能会发送给请求者一条工作已完成的消息，并伴随有一些数据。

怎样？我的第一印象就是，你根本无法从中把服务指出来。从我们简单模型中的所有当前流程中，很难指出那个服务到底是什么。在很大程度上，我们所知道 的就是它的名字。余下的流程步骤是：调用、执行代码、改变存储在特定内存位置中的值、当特定内存位置中的值等于或者超过某个值时候采取行动，等等。我们是 否能推断：服务是客观实在的，通过给它命名，我们就让像“存在”这样的事物被抽象了出来？或者服务具有突发性，它是偶然发生的，因所有的活动都发生了而产 生？我认为玩味措辞没有太大的价值，因此毫无疑问，我想我们应该会对能够给一个服务命名而感到高兴。

自从分支到子程序时代以来，现在有什么已经被改变了？通过达尔文我们知道了现实世界并不存在本质，但是 IT 系统和它们的数学模型联系得实在太紧密 了，以至于在某些限制下有必要问一句：当跟它们的遗留对应物进行比较时，当代应用中有什么本质区别使得服务概念有价值？ 这个问题的答案很简单：复杂度。在汇编程序时代，交叉引用表是关于所有你所需服务的概览，记录了什么地点什么人物使用它们。在 2008 年该对象的 Web 服 务版本中，你确实需要模型和系统来指导服务的开发和使用。在所有层面，服务都很复杂。如果你需要

一些隐藏细节的有趣例子，那么可以去看看面向服务在实践中 使用的情况。让业务人员可以使用服务并使用这些服务来创建业务流程并不简单 ,这需要服务目录用业务语言表达服务的目标。在技术层面存在大量的复杂性，因为 要使所有不同平台和网络技术必须在一个服务环境下无缝的协作。在设计和构建层面，这些面向业务和面向技术的元素被编织在一起。一旦完成部署，还需满足服务 水平。并且在所有领域，同一世界中的所有这些都喜欢每天进行一些少量的改变。因此一成不变并不好，服务必须随变化而构建。我甚至都没敢提信息安全！

如果服务是少数人的兴趣，那本文所说的就完全不切题了。事实上，正好相反。它们承诺带来机动性，故此赢得了业务人员的喜欢！

因此，为了行动迅速，我们需要站在巨人的肩膀上。耦合和内聚的概念、结构化分析和设计、封装、模式的使用，全都在那儿供我们差遣[5]、[6]、[7]。

关键词

和面向服务关联使用的词有：架构、成熟度模型、路线图、治理。在这一领域是否已经完成了一些降低复杂度的工作？我认为是有的。

成熟度模型讲的是最佳实践。这就意味着它们的主要内容是关于在特定条件下产生高质量结果的真实存在的流程。通过指引企业去关注那些重要性已被证明了 的流程领域，成熟度模型可以用于引导企业的执行水平更上一层楼。以这种方式使用的成熟度模型起到了路线图的作用：在旅途中，它引导你由此及彼。但是还有另 一种广为人知的路线图，它的目的地是未知的 ,从某种意义上来说 ,我们无法明确指出真的存在这样一个所谓的目的地。与之相反，目的地是一种可感知、值得期待 的位置。在这样的位置上，下一步要做的就是计划如何通过一系列中间步骤由当前位置通向天堂。这种迁移并没有任何错误，实际情况正相反。但是你不应该把最终 模型称为成熟度模型。要是这样做了，你就把成熟度模型的概念扩展到了一个使它无意义的程度。这种做法仅仅增加了全局复杂性，而没有消除局部复杂性。

面向服务是否是完全不同的事物？从某些角度看，确实如此。仅仅由于它的不同，就真的需要它自己的成熟度模型、治理等等吗？视情况而定。开发一个全面 的成熟度模型，涵盖整个面向服务领域，潜在范围从业务流程到 XML 和语义接口，不是件简单的事情。如有可能，我们应使用易于获得的元素去及时得到这样一个 打算使用的成熟度模型。像 Photoshop 和 Eclipse 这样差别巨大的应用程序也都采用了同样的机制：插件。尤其是在面向服务环境下，出于复用的目 的对现有模型进行评估是第一步。当采用插件视角时，许多像 CMMI、ITIL

和 COBIT 这样的模型都能够为面向服务的实现速度和业务成功做出贡献。

因而我要再次强调“保持简单”这句格言。不要增加模型，除非不这样做会导致恶果，坚持爱因斯坦的观点，尽量简单地表达每个事物，但又不是过于简单。 把握词汇的既定含义：成熟度模型讲的是最佳实践，路线图被用于以一种 Rand McNally（译注：美国制图厂商，有百年历史，其所制之图极其详尽。）风格来了解目标，迁移计划是为了通往天堂。并且最后但并非不重要的是：复用，复用，复用。

最后一个关键词：ABC

有一个被称为 ABC 的臭名昭著三人组：学究（Academics）、企业（Business）和顾问（Consultants）。学究铸造新学说。 顾问用自己的语言把它们翻译成企业钟意的部件。而企业给顾问付费，并让学究有机会在组织内部四处闲逛收集新学说的基础素材。很明显，要在这个领域挣到钱， 你就不应该使用具有清晰含义的措辞。相反，你要采用“新瓶装旧酒”的策略。用营销的话来讲：你是唯一的。但效果却是相同的：大量的文字游戏，却没有完成多少实事。像这样一个过程能否解释为什么我们看到有这么多的新模型，却很少主动让模型适应变化——或乃至适合使用——并让它们与时俱进?

结束语

记住，模型摒弃了你和你的目标并不关注的方面和部分。为理解顾问们的故事而创建的模型未必能解决你的问题。正如 Mark Anthony Luhrmann 所说“谨犯那些劝你买东西的人，但是对于给你忠告的人应有耐心。忠告是某种形式的怀旧，给出忠告就像从过去的废墟中寻宝一样，擦去表面的 污垢，美化其丑陋的部分，重新利用它让其物超所值。”

如果这个文章是一个服务，那请求可能就是：“我如何从面向服务中有所收获”，而返回的数据就是：“设法弄清楚哪些是新的。尽量掌握新的内容。不要忘记之前有很多像你这样或那样的人都或多或少的已经学习了这些。复用它。新部件要复杂些。复用并不简单。让这种力量与你同在。”

[1] Kenneth E. Boulding, 1956. General Systems Theory, The Skeleton of Science. In: Management Science, 2, 3 (Apr. 1956) pp.197-208; reprinted in General Systems, Yearbook of the Society for General Systems Research, vol. 1, 1956.

[2] W. Ross Ashby, 1956. An Introduction to Cybernetics, London, Chapman & Hall.

[3] Ludwig von Bertalanffy, 1968. General System Theory: Foundations, Development, Applications, New York: George Braziller, revised edition 1976: ISBN-13: 978-0807604533.

[4] Bertrand Russell, 1918. Lecture: The Philosophy of Logical Atomism. Bertrand Russell, David Peers (ed.), 1985. The Philosophy of Logical Atomism, Open Court Classics. ISBN-13: 978-0875484433. Also available via the Amazon Online Reader.

[5] Edward Yourdon and Larry L. Constantine, 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall. ISBN-13: 978-0138544713 facsimile edition 1986.

[6] Glenford Myers, 1979. Reliable Software Through Composite Design. Van Nostrand Reinhold, ISBN-13: 978-0442256203.

[7] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, a.k.a The Gang of Four, 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN-13: 978-0201633610.

服务概念通常对企业和社会都很重要。它意味着迈向无约束信息流（Open Group 的愿景）和 Brewster Kahle 的互联网档案馆所设想的随时访问所有人类知识的一大步。多么令人激动的两个愿景！

原文链接：http://www.infoq.com/cn/articles/leeuwis-harvesting-soa

相关内容：

- SOA在互联网系统中的应用
- Mule的消息路由
- SOA治理成熟度——一名架构师的观点
- 面向服务的经济学
- SOA契约成熟度模型

新品推荐 | New Products

FindBugs 1.3.9 发布了

作者Charles Humble **译者** 张龙

Bill Pugh 于近日发布了 FindBugs 1.3.9——流行的 Java 静态代码分析工具。该版本增加了 12 个新的 bug 检测器，同时依然致力于不断改进效率以胜任大规模的代码，今年底将要发布的 2.0 版也将效率问题看作是重中之重。

原文链接：http://www.infoq.com/cn/news/2009/09/findbugs

微软发布新版Pex框架，对测试提供了更好的支持

作者 赵劼

不久前微软发布了新版本的 Pex 框架，其中的 Moles 组件可用于模拟框架中包括静态成员在内的几乎所有成员，大幅提高了对测试的支持程度。

原文链接：http://www.infoq.com/cn/news/2009/09/pex-stubs-moles

SproutCore：一个HTML 5 应用框架

作者 Craig Wickesser **译者** 张龙

此前 InfoQ 曾对 SproutCore 有过多次报道 ,最近获悉其 1.0 版行将发布的消息。SproutCore 的目标是在浏览器中提供极佳的桌面效果应用而无需任何浏览器插件。

原文链接：http://www.infoq.com/cn/news/2009/09/sproutcore-1-0

Mono的第一个商业版本：MonoTouch

作者 Jonathan Allen **译者** 朱永光

MonoTouch 是 Mono 运行时的一个移植版本，附带了一个适配器层，以便.NET 开发人员能够使用原生的 iPhone GUI 工具包。MonoTouch 在 Mono 的世界中是独一无二的，因为它是 Novell 发布的第一个商业 Mono 产品。如大家预料的一样，引起了社区的强烈反应。

原文链接：http://www.infoq.com/cn/news/2009/09/MonoTouch

无模式数据库MongoDB 1.0 版发布

作者 赵劼

Mongo 是一个高性能，开源，无模式的文档型数据库，它在许多场景下可用于替代传统的关系型数据库或键/值存储方式。

原文链接：http://www.infoq.com/cn/news/2009/09/mongodb

JBoss HornetQ项目发布了

作者 Dionysios G. Synodinos **译者** 张龙

近日 JBoss 发布了 HornetQ 项目，这是一个开源、多协议、嵌入式、高性能、集群、异步的消息系统。过去几年，JBoss Messaging 2.0 一直使用 HornetQ 的代码基。

原文链接：http://www.infoq.com/cn/news/2009/09/hornetq

SpringSource Cloud Foundry发布了

作者Scott Delap **译者** 张龙

SpringSource 一直在不遗余力地推动着 Java 与云技术的整合，近日又发布了 SpringSource Cloud Foundry，我们可以凭借它轻松将 Java Web 应用部署到云平台上，如 Amazon EC2。

原文链接：http://www.infoq.com/cn/news/2009/09/cloudfoundry

Lucid Imagination发布了Apache Lucene性能监测工具

作者 Charles Humble **译者** 宋玮

采用 Apache 搜索引擎 Lucene 从事相关开发工作的开发者通常都会依靠代码剖析器（code profiler）、调试源代码、或者给 Lucene 代码手工增加跟踪代码等方法来捕捉其 Lucene 实现的性能变化。而 Lucid Imagination 出品的一个免费产品 LucidGaze 则提供了另一种方法。Infoq 注意到了该产品并对 Lucid Imagination 进行了采访。

原文链接：http://www.infoq.com/cn/news/2009/09/lucidgaze

架构师大家谈

我来选架构

架构很重要，这是毋庸置疑的。但是面对现如今各种新的技术、协议或框架，让人看了不免头晕眼花，到底如何选择呢？有哪些原则、需要注意些什么问题呢？让我们来看看以下这几位具有实际经验的同行们的观点。当然，也欢迎您提出自己的独特见解。

- 郭晓刚：InfoQ 中文站 Architecture 社区首席编辑。
- 麦天志：InfoQ 中文站 Agile 社区编辑。
- 诸鸿君：Sybase 西安研发中心，CEP（Complex Event Procesing) QA 团队负责人。
- 王军：长期致力于软件开发工作，目前在中兴通讯任职开发经理。

如何选择架构实现技术、协议或框架？需要注意哪些问题或遵循哪些原则呢？

郭晓刚：

选择架构时先要明确选择的标准和约束条件。判据可以有项目的需求、框架本身的发展情况、开发者的观感等等。我有一个从 Matt Raible（http://raibledesigns.com/）那里学来的打分的办法。办法是这样的：

1. 挑几个候选。
2. 在给定的短时间内用候选框架分别搭建一个 Demo。Demo 涵盖未来应用需要的一些重要的特性。这样可以实际地体会各框架的差异。框架的学习难度（或开发者对它的熟悉程度）也得到检验。
3. 记下 Demo 开发中的发现。把重要的判据列个表给各框架打分。总结并作出结论。如何打分请参考例子（http://raibledesigns.com/rd/entry/ajax_framework_analysis_results ）。

要不要做这个过程，以及要不要放在项目开始之前做，我的判断依据是它算不算基础设施，

也就是说改变的代价大不大。记录下判据和分析是重要的，能防止将来不经意的破坏，也能减轻团队中无意义的 flame war。未来出现新选择的时候，重新下结论也有个谱。我发现明确区分开直接关系到（有潜在竞争关系的）不同利益方的判据有帮助，分清哪些选择标准是出于客户的需要，哪些是出于开发者的需要。这样在打分的时候，开发者表达自己好恶不会不好意思，替别人做决定的时候也会谨慎一些。在此过程中，首先不要忘了问一个问题：是否真的需要引入这么一个东西？

总是自动套用一套既定的设施，这是 Big Up Front 的设计，坏处不用多说。我也犯过反面的错误，为了保留随时重新选择的自由，把系统搞得复杂无比。两种极端都是极其昂贵的。只有不确定性高的部分才有必要保留灵活性，付出复杂度提高的代价。
如果总是跳过第 2 步，很容易出现把一切过程都变成填表格的倾向，丢掉了探索意义之后，操弄数据可以得到任何你想要的结果。

麦天志：

选择架构传统以来都是让人头痛的问题，之所以头痛是跟架构的本质有直接关系，很多前辈都曾经为架构提出定义，Neal Ford 在“Evolutionary architecture and emergent design”系列中提出的最直接了当：”以后很难改动的东西”（”Stuff that is hard to change later”），是因为难以作出改动，所以作出了错误的决定风险也很高，改过来的成本也很高。同时这样也提出了一些原则，第一点就是到最后必须关头才作架构决定，更理想的是减少不能挽回的决定，这样我们不用过早，甚至欠缺充份资讯下作出架构上的决定，作决定时不妨问问自己：”我必需现在作决定嘛？”，”有没有什么可以让我延迟作出这决定呢？”。

第二个原则就是以最简单的方式和架构去实现需求，过度架构是危险陷阱，作架构决定时很易受“弹性”，”伸延性”等吸引而希望做出最灵活的架构，但其实这同时很可能过早在项目上限制设计演化的可能性，增加不需要的灵活度住住使以后加入代码时更复杂。简单的架构设计另一个好处在于能及早付运，其实要知道架构好不好，最佳方法还是实现出来，好的架构能替客户及早增值，通过实现架构能知道自己所处的情况，然后有能充份的资讯去作出更好的决定。

第三个原则是让真正写代码的团队去作架构决定，而且是自我组织团队，这可能对部份架构师来说是很难接受的事情，不过使用自己设计的架构最能明白架构上的问题，所以好的架构师其实也是开发团队中的领导编程员，以集体方式作出架构决定，好的架构师更应该能意识到改变所需要的代价和后果，而且能与各方项目相关人士沟通相关事宜，在思考架构决定时还要照顾跟开发人员，测试人员等的沟通，让架构在团队中清析可见，而不仅仅是一份没有

人想看的文档。

另一方面要注意的是质量问题，在作架构决定要考虑的是如何去保証质素良好，一方面要让作出架构决定的开发团队去作测试，这是把保证内建质素的最好方法，另一方面测试同时也是替日后改变的安全网，让团队知道作出的改变有没有破坏原有的功能。
最后，架构是持续的活动，而架构改变的价值跟不作出改变的成本直接成正比，能适应业务需求改变的架构才能为客户带来价值。

诸鸿君：

简单的说，系统架构的选择取决于用户的需求。具体说包括以下几个方面：

1. 这个应用是事务型的还是分析型的，即偏向于 OLTP 还是 OLAP
2. 系统的实时性：不是指对用户请求的实时响应，而是指系统对现实世界的实时反映
3. 数据交换方面的特征：每次交互涉及的数据量；数据的来源及流向
4. 用户体验方面的特征：总用户数；在线用户数；并发用户数；对用户请求响应时间的要求

需要注意的是对用户需求变更的可能性进行估计是非常重要的，这对系统架构的灵活度起决定性的影响。

王军：

要实现一个可重用、可扩展、简明、高效、安全的架构，我们需要遵循一些原则：

1. 功能划分，就是把功能分解为不同的模块，每个模块都有自己的单独功能。
2. 够用就好，只要能够满足需求就好。
3. 灵活设计不同的耦合度，要根据需求的稳定性来确定耦合程度。对于稳定性高的需求，不会再变化，那么紧耦合也未尝不可，对于需求变化快的部分，才充分考虑降低耦合度的问题。
4. 应用已有模式，其实就是拷贝他人的思想，学习他人的经验。

在长期的软件开发中，个人一向强调满足需求就好，认为架构一切都是服务于需求的，如果没有需求，也就无需谈什么架构了。正所谓过犹不及，适合需求的架构才是最完美的。

推荐编辑 | Java 社区编辑 张凯峰

各位读者好，很高兴在这里跟大家见面。我是 InfoQ 中文站 Java 社区编辑张凯峰，目前在 IBM 中国软件开发中心从事 Lotus 产品的开发工作。我比较喜欢的技术领域有 Web 开发、Java 等方向，有志同道合的朋友欢迎骚扰 zhangkf [at] cn.infoq.com。

我加入 InfoQ 中文站，还得从认识泰稳（InfoQ 中文站总编）说起。在泰稳还在 CSDN《程序员》杂志社任职时，我从他担任主持的 dev2dev 技术社区活动中结识了他。因为他的实诚和友善，我很快和他结识，并保持朋友关系到现在，甚至一度因缘际会可能加入 CSDN《程序员》与他并肩战斗。他的个人魅力感染着我，他对国内技术社区发展的执着追求也深深让我折服。为了自己的梦想，泰稳离开了 CSDN，开始承担起 InfoQ 这一高端技术媒体在中国的发展使命，我有幸自始关注事件的发生。还记得那天，泰稳、我还有至顶网的李宁在五道口东坡酒楼，开始商量筹划 InfoQ 中文站的事宜，大事件总会从琐碎开始，甚至从购买电脑设备、摄像器械开始。就这样，InfoQ 中文站算是白手起家，一篇篇新闻，一篇篇文章，充实了整个中文网站，一步步走到今天，最离不开帮助和支持 InfoQ 中文站的朋友们和广大读者，这其中有曾经贡献过自己力量并一直关注网站的人们，更包括无私献出自己宝贵业余时间为广大国内读者提供高品质的内容的编辑群体。

而我在这样的一个以无私贡献为己任的群体里面，感到的不仅仅是执着的使命感和温暖的气氛，更有个人知识结构能力水平的提升，还有效地拓宽了自己的人脉圈。因为编辑工作的需要，我要对英文新闻及时地浏览，以准确的翻译成中文供读者们飨用，而如果是要原创新闻，则更要博采线索，在广泛的技术社区中摘取最有价值的部分提炼给读者们。在这样的过程中，自己的认知能力得到极大的提升，每次看到读者们积极的回复评论，甚至是挑错，我都由衷感到高兴，高兴的是我、InfoQ 中文站和读者们一道，在慢慢的成长，但茁壮的。

我有幸一直在繁忙的本职工作之外，还能坚持InfoQ的兼职工作至今，我从中得到的远比我失去的只能用来无所事事的业余时间要多得多。InfoQ兼职编辑的模式在国内社区算是独树一帜，质量高而品质好，每当有同行朋友听说是InfoQ中文站的编辑时，我的自豪心会随着他们的赞叹一道升腾。在InfoQ期间，我有幸参与了《Ajax实战》、《开源技术选型手册》、《Google API大全 编程开发实例》、《我是一只IT小小鸟》的译作，让我体会到其中的艰辛和收获的喜悦，我还以InfoQ中文站编辑的身份参加了今年四月份在北京举行的QCon大会 ,和九月份刚刚结束的敏捷中国大会 ,而这都是InfoQ中文站为编辑们免费提供的技术盛宴。

我很高兴也很愿意继续为 InfoQ 中文贡献我的一己之力 ,更欢迎相信自己实力有志加入 InfoQ 中文站的朋友，为国内的技术社区发展和分享添砖加瓦，同时最大限度地提升自身能力。

封面植物

半日花

现状：稀有种。半日花为古地中海植物区系的残遗植物。现为亚洲中部荒漠的特有种。在我国可形成小面积的荒漠群落。由于过度放牧及乱挖烧柴，受到较严重的破坏，数量已逐渐减少。

形态特征：落叶小灌木，高 10-15 厘米，多分枝，形成较紧密的灌丛；老枝褐色或淡褐色，无毛，小枝淡灰褐色，被短柔毛，先端常尖锐而呈刺状。单叶对生，革质，长圆状狼形至长圆状披针形，长 5-12 毫米，宽 2-4 毫迷，边缘全缘，常向下反卷，两面被白色绵毛；无柄或具短柄；托叶小，钻形。花两性，单生枝顶，鲜黄色，径约 1.4 厘米；花梗长 6-10 毫米；萼片 5，不等大，外面 2 片线形，长约 2 毫米，内面 3 片卵形或宽放形，长 5-7 毫米，背面具 3-5 条纵肋；花瓣 5，倒卵形；雄蕊多数，长为花瓣的一半；子房上位，密生柔毛，花柱线形。蒴果卵圆形，长约 5 毫米，被短柔毛；种子卵圆形，长约 3 毫米。

地理分布：分布于新疆伊宁、巩留、特克斯，甘肃民乐及内蒙古伊克昭盟的桌子山等地。生于海拔 1000-1300 米的低山石质残丘坡地上。苏联哈工萨克斯坦的东部也有分布。

生态学和生物学特性：分布区为强大陆性气候，冬季寒冷、夏季炎热，最低气温可达-35℃，最高气温可达 39℃；干旱少雨，年降水量约 150 毫米，蒸发量远远超过降水量。土壤为漠钙土，地表具大量碎石块，其覆盖率可达 70%以上，有的地方有积沙覆盖。半日花为超早生的小灌木，多在山麓石岳残丘形成半日花荒漠群落，在山前洪积平原少见。在典型的半日花荒漠中，亚优势种为刺旋花 Convolvulus tragacanthoides Turcz.、主要伴生种为灌木青兰 Dracocephahum fruticulosum subsp.psammohpilum(C. Y. Wu et W. T. Wang)H. C. Fu et Sh. Chen.等。在一些有覆沙的地段上，可出现四合木 Tetraena mongolica Maxim.、霸王 Zygophylhum xanthoxylum(Bunge)Maxim.等灌木。开花期为 5 月下旬至 7 月上旬，有时 8-9 月能第二次开花。

保护价值：半日花是亚洲中部荒漠的特有种，对研究亚洲中部，特别是研究我国荒漠植物区系的起源以及与地中海植物区系的联系有重要的科学价值。


1kg.org
多背一公斤
爱自然 | 更爱孩子

架构师 10月刊

每月 8 日出版

本期主编：宋玮

总编辑：霍泰稳

总编助理：刘申

编辑：胡键 朱永光 郑柯 李明 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：宋玮，InfoQ 中文站 Java 社区首席编辑

毕业于西安交通大学计算机科学与工程系，曾在西安某研究所工作并获得硕士学位。有多年软件开发经验，并先后在多个开发或实施项目中担任主要开发者或项目经理。由于亲身感受到开放、开源对软件发展所起的重要推动作用，因此是开源技术及开放理论的坚定支持者，在项目实践中尽可能推动或尝试将一些成熟技术和理论运用于实际当中。又因经常参与需求及设计的相关工作，架构也是常挂在嘴边的词汇，于是本着实践结合理论的思想，参与了《SOA实践指南——应用整体架构》一书的翻译。回顾以往，虽无建树却也有所感悟，欢迎各位同行与我联系：songwei12 [at] gmail.com。

InfoQ中文站
www.infoq.com/cn

《架构师》月刊由InfoQ中文站出品。