InfoQ 时刻关注企业软件开发领域的变化与创新

架构师 Architect

试刊号

谷雪梅谈云计算

InfoQ中文站有幸与谷歌的资深工程师谷雪梅女士在一起探讨了云计算的相关话题。

厂商依赖是云计算采纳的障碍？

在21世纪的今天仍然步桌面套件和私有SQL的后尘、重演厂商依赖的悲剧，我们“脑残”到这个地步了吗？你知道，那等于把IT预算控制权交给平台厂商了？

探求真正的SOA

业界从来不缺造词机器，在如今各类缩写满天飞的时代，相信大家也有被搞晕的时候。

InfoQ中文站
www.infoq.com/cn

每月8日出版
InfoQ中文站制作发布

篇首语

由壹基金的专业性想到的

在西安出差时，恰逢凤凰卫视播放李连杰壹基金会为其评选出来的“典范工程”颁奖。观看整个晚会过程后，再联想到其一直以来所坚持的专业性，我不禁浮想联翩，想得最多的自然还是目前国内技术社区的发展状况。

记得在汶川大地震时，李连杰的壹基金团队也参与了救援，不过与其他类似的救援组织有所不同的是，与他们同去的还有德勤会计师事务所。据报道，其实从壹基金成立的那天起，他们就特别注重对基金的专业化管理，从而使得自己在民众的心中更显中立和公正性。相比于那些屡屡爆出救灾资金被私吞或者被滥用的其他基金会而言，壹基金的做法无疑高人一筹。而在这次的“典范工程”颁奖晚会上，壹基金再次彰显其公正的一面，为了选出最具有说服力的“典范”，他们邀请了多位口碑好、专业强的评委，如希望工程创始人、南都基金会秘书长徐永光等，面试过程也是非常严格。事实证明这些站在讲台上的组织确实都令人折服！

以他人之例，可以反思我们手中所做的事情。做网站难，做技术社区更难，我想这是很多社区创始人的同感。当然其中有很多客观的原因，比如相比于欧美国家，我们的 IT 产业大环境还不是很理想，专业人群的基数也不够大等等。但我想提醒的还是主观方面的原因，尤其是专业性方面。以从事技术社区的编辑记者为例，我们的很多“专业人士”对技术缺少最基本的了解，所以在一些专家见面会上，常见我们的记者大人们问出一些让人哭笑不得的问题。究其客观原因，是因为技术社区的特点决定了服务于这个社区的编辑记者需要具备技术和文学的复合能力；究其主观原因，是因为这样的复合人才确实稀少，为了缩减成本，某些社区只好退而求其次，请一些文采略好但不懂技术的人来担大梁。并不是说这样就不可以，只是专业性也许稍微差了点。

再说一下国内技术社区的特点，“大而全”是其中多数社区的一个特点，原因其实也很简单，商务所迫或者追逐利益所致。只要是客户感兴趣的，都马上设置相应的栏目。殊不知，

专业的内容如果没有专业的人才来打理，那结果只能是遗笑大方。那么有没有什么解决的办法？研究一下壹基金的架构，人家主动邀请德勤来做监督，我们是不是也应该给自己带个紧箍咒什么的，让自己将主要精力就放在对读者最有价值的事情上？这样一旦脑袋发热，紧箍咒就能提醒一下“要专注从事”。想起在做 InfoQ 中文站的过程中，我常想如果中文站也如国内其他的网站那么大该多好。就此事与 Floyd 聊起时，他常会嘿嘿一笑说，“我们目前只关注企业软件开发领域”。现在反思，也许这就是为什么 InfoQ 在不到两年半的时间里就能为全球中高端技术人员所熟知和喜欢的原因吧。

回到本期的《架构师》试刊第二期，相比于上期，我们在内容的选择安排和版式上都根据读者的意见重新做了修正，比如将“人物专访”放在最前位置，以使读者在最短的时间内阅读到技术专家的观点；将“新品推荐”的内容进行缩减，仅提供摘要性文字以提高信息量等；将段间距进行调整，以使得阅读过程更加轻松等。“细节决定成败”，我们希望基于 InfoQ 中文站的专业内容，《架构师》能够一步步成为中文社区架构师、项目经理、团队领导者和高级软件工程师等朋友喜欢的电子刊物！

霍泰稳

目　录

[人物专访]

谷雪梅谈云计算

摘要：InfoQ 中文站有幸与谷歌的资深工程师谷雪梅女士在一起探讨了云计算的相关话题，包括云计算的概念、它与网格计算和公用计算的异同、云计算的优势、虚拟化在其中扮演的角色、App Engine 等等。

谷雪梅，谷歌资深工程师，毕业于卡内基梅隆大学，四年前加入谷歌，负责谷歌中国基础架构方面的工作。

InfoQ 中文站：非常有幸能采访到谷歌的资深软件工程师谷雪梅女士，我们今天所谈论的话题是关于云计算方面的，现在大家都在热捧云计算，包括 IBM 他有蓝云，然后微软的也有 Live Mesh，Amazon 也有 EC2，雅虎有自己的云，那么我们想了解下 Google 的云是什么样子的，这些云之间，这些云概念之间有没有什么异同点？

谷雪梅：咱们还是从云计算的概念开始讲起——就是它的根源。最早云计算是二十年前，大概在美国就出现了，当时是戴克这个公司提出来的，他们的研究人员提出来说希望呢有那么一天我们的计算机也不会就是像现在一样，一台一台在每一个桌子上，而是变成一种服务，像电网一样，一插上电源，好，你的计算能力就来了。然后用户想做什么，就通过这个云的网，当时不叫云，当时他们认为这个像电网一样的计算机的网络帮你实现，所以这个是 20 年前的概念，然后这个概念一直没有被实现。主要原因有两点：一点是硬件，硬件很贵，所以我们当时也不太可能使用很多很多的硬件来给大家提供这种当时还看不到什么很大的应用的用户的需求；然后第二点是软件，后面当这 20 年来，硬件的价格不断的下降，有那么一天呢，大家终于可以用很便宜的价格买出很多很多硬件的时候，那么这个时候软件就进来了，它的这个角色是什么呢？好，我使用一大堆非常便宜的硬件，然后我给这个用户来构架一个非常大的平台，这个平台可以使用所有的硬件，有很大很大的计算能力，有很大很大的存储能力，那么这个时候，我觉得云计算才终于面世了。

它的概念很早，但是实现上面只有在硬件和软件都达到一定水准，才进入现实。所以我觉得不管是 IBM 还是微软还是 MSN 和雅虎，在概念上大家都没有区别，大家都想处理大规模的数据，然后通过处理大规模的数据给用户提供你原来可能想象不到的情况，但是在实现上面的话，我作为 Google 的工程师来讲一讲 Google 的特点，不太好跟别的这个比较了，Google 起步是 97 年 98 年，那个时候互联网是很小的，就是小到大概 Google 用几十台计算机，可能上百台计算机，也可以把整个互联网所有内容都搞下来了，所以那个时候的云是一个小小的云，因为互联网很小。那么在过去 10 年里边，Google 我认为他是一个很幸运的公司，他随着互联网的成长，他自己也在成长，互联网在爆炸性的成长的时候，Google 加很多很多的服务器，然后把云计算这个概念放到了每一个产品的开发之中。然后那个时候，当我发现互联网越来越大的时候，原来的那种计算模式可能已经不能满足它成长的速度，那么没有关系，好，我们就买了很多很多，像我刚才讲得便宜的硬件，然后在上面开发功能非常强大的软件，可以处理互联网，整个互联网级别的这些数据，然后给用户提供结果，所以 Google 有 10 年的，我认为是实践的经验，而且 Google 云计算实际上已经做出来了，这是一个可以摆在大家面前给看的东西，每天你在 Google 上面如果进行网页的搜索，全世界的网页不只是中国的，不只是美国的，全世界的网页，你在上面搜到结果。那么这个结果事实上就是后面很大很大的云，云然后回到现实就是很多很多的硬件，然后上面有非常非常大规模的软件来支撑它，这就是 Google 的实现。

InfoQ 中文站：刚才你所谈到的它可能像电网一样那么方便，那么简单，那么能不能再具体化一点，给我们介绍一下？

谷雪梅：具体化，其实你看现在很多工作并不需要你的台式计算机，有很多很多工作你不再需要了。那么你打开浏览器，你要进行的搜索没有问题，一行这个查询进去，结果就出来了，这是不是像电网一样方便？然后再有呢，比如说 Google 还有像网上的就是微软的 OFFICE 一样，Google 的 Docs，然后其他的 SpreadSheet，很多很多的网上的应用，那么这些应用，你本地不用再装任何的软件，Google 都替你做了，你所需要的事情，就是面对一个很简单的浏览器的界面，把你所需要东西放上去，那么你在其他任何地方，你想拿到，你可以随时可以拿回来了，我想这个是个非常简单的例子，可以帮助大家形象的理解。

InfoQ 中文站：很形象，根据我掌握的资料，云计算好像跟上世纪的两个概念，因为 80 年代末的时候有一个网格计算，然后 90 年代末的时候有个公用计算，好像看上去两个概念很类似，我理解看都是讲资源集中在网络服务器上，大家通过网络就可以应用或者计算就可以了，你的理解是什么样子的？

谷雪梅：对，其实这个是这样的，的确概念上我同意，是没有什么区别，还是咱们回到实现，我认为实现是特别重要的。这个科幻小说，你写一写就写出来了，这个不需要很多工程师做很多工作，但是你真的把那航天飞机发射上去，那就是聚留了几个大的系统工程了，云计算也是，这个概念可能几个聪明的人坐那想出来了，但是你要真的实现呢，你真的需要一大群非常非常有才华的工程师进行长期的、艰苦的、很有创造性的工作才能做出来，所以这是我认为他们最大的区别，就是八九十年代更多的时候是在概念方面，而现在是我们真正看到成果了，而这个成果不是一个小成果，他是针对整个互联网上几百亿网页，各种各样的这个内容，然后做出的结果。然后我再讲讲八九十年代，刚才我不是讲到硬件很贵的问题吗？当时还有一个很贵，就是网络资源很贵，当时可能你传输一个字节花的钱跟现在是完全不一样的概念，所以那个时候，你再讲云计算，你可能也是，我们叫 Local 的。你可能在这边，一个大楼里边有若干台服务器，没有问题，你直接传输下可以了，但是如果你想从这发出的资源，让世界另外一边的人很容易的得到他，那是不可能的，但是现在的世界就可能了，所以我认为，还是说实现上有最大的区别。

InfoQ 中文站：那么我们刚才也谈了一些云计算的概念，那么现在我们来看一下，云计算发展它的背后的主要助推因素是什么样子？或者说云计算现在对大家有什么好处？比如说推广云计算的厂商，包括 Google，包括微软，包括 IBM，还有使用云计算的这些终端用户，包括那些想在云计算平台上工作的架构师。

谷雪梅：因为我本人也是工程师，我觉得还是给大家举一个工程方面的例子，我不知道你做过没做过网站？

InfoQ 中文站：做过简单的。

谷雪梅：你当时做网站有没有感觉说，你可能需要什么软件，我得把这个网站搭起来，然后我打电话注册一个域名，然后再出去找这个服务器，然后还得找数据中心，然后可能自

己还得肩负点系统，叫什么 System Admin，网管员的工作，做个网站就得这样吗。再做复杂一点的，比如说你做一个电子商务的网站，支付了，然后所有的这些，比如像 Shopping Cart，什么 Transaction，很多很多东西都要做。那么其实在云计算的时候，我给大家举个很简单的例子，Google 有一个云上面的应用叫做 App Engine，App Engine 的概念是很简单，他就是说好吧，你有什么样的自己的想法，甚至你是个小公司，你把你自己的想法用 Google 给你提供的所有的工具，很容易的就形成你的网站，形成你的这个商业应用，你不用管，你不用再去操心这个网管，你不用再去操心你需要多少的 CPU，多少的内存，多少的这个 Storage，你甚至不用再操心你去哪里租数据中心，你这些东西所有都不用管，你需要做得事情就是把你的想法想出来，然后用一些简单的前台工具把它实现出来，所有这个底层的东西，全让 Google 给你实现。这个事实上我觉得是云计算一个非常典型的应用，把大家从这个很多很多技术上的这个很无聊的很乏味的很底层一些东西解放出来，让你把自己的精力主要集中在自己的这个想法的开发上面。

InfoQ 中文站：下面我们看几个目前比较流行的一些概念和云计算之间的一个比较，比如说大规模计算，分布式存储，还有什么 SaaS，就是软件及服务，包括虚拟化，那么你能简单帮我们解释一下这些概念之间的关系或区别吗？

谷雪梅：我觉得这些概念其实都是有联系的，比如说咱们先把云计算到一边，咱们先讲讲大规模，比如大规模计算各分布式存储。那么大规模计算，这个是这样，它强调的重点是在多个计算机之间的协作。这个计算机的协作，比如说其实你看一个非常大的数据库，它实际上是大规模计算的，它可能假如说要给你查一条东西的话，它要从很多很多地方来查，查完了最后把一个结果返回给用户。但是比如说大规模计算和分布式存储，它们两个就是一个很配合的关系，因为首先大规模的计算经常会需要很多很多的数据。那么这些数据，毫无疑问是不可能存在一台服务器上的，或者如果你存在一台服务器那会很贵的，你只能把它分布式存储，然后你通过分布式的存储，你必须要设计各种的算法，然后把 Data，把这个数据从不同地方拿过来进行什么样的计算，你要考虑到它的 IO，考虑到它在 Internet 上面的传输，考虑很多这种技术的问题，然后你前面实现了这个大规模计算。然后咱们再比如说，像软件及服务这种也不过是说，我们把很多的就是原来放在单独计算机上很独立的一些东西，然后放到网上去，或者放到，你不用关心它是在那里实现的，你只关心它的结果，所以这些你看，

这些东西呢，我认为都是关联的，它们有几个重点一定要有的，存储，大规模的存储，这几乎是没有疑问，是一定要的，另外一个是需要很大的，我们叫 CPU Power，就是很大的计算能力，因为你需要太多的数据，需要去把它算出来，那么算的结果，究竟是这个，是变成服务的软件呢？还是说你有个想新的想法，是一个互联网上的应用呢？还是什么东西都没有关系，这个商务的应用实际上都是类似的，只是说这个商务应用一定会使用存储计算，然后最后服务，就是前台的服务。那么云计算，我认为是说把这些概念都集中在一起，所以云计算，它更强调的就是说，我觉得它比较像软件即服务，这个服务在那里，你不用关心，你需要关心的只是它的一个结果，所以它更多的是一个应用层面的一个概念。

InfoQ 中文站：那虚拟化在里边办什么角色？

谷雪梅：虚拟化，我想是这样的，虚拟化当然是一定跟他前面这些概念是相关的。然后比如像云计算，它所谓的虚拟化是说，所有的后台这些计算，存储什么的，都和你现在的，你自己面前的这个计算机事实上是脱离的，你不用管他在哪，他可能是在美国，还可能是在英国，或者是在印度尼西亚，这些对你来讲已经没有关系了，你所需要关心的只是你面前的一个结果，所以我想后面对这些东西的话，他就比较虚拟，而不是你现实看到很多很多服务器在你面前。

那么另外一个说法就是说，有人将云计算比喻成互联网为中心的软件，你是如何理解的？

我非常赞同，我觉得其实这对云计算是一个很好的，尤其对于那种不太了解云计算的人，是一个非常好的说法，能让他很明白的理解什么是云计算。然后你看这里边，其实我想强调一个重点是说，它是以互联网为中心的，这点为什么很重要呢？原来，假如说有些我们的的研究人员，他有想法，他有很好的想法，比如说翻译，那么翻译事实上，他那些算法几乎已经定型了，他所需要的是什么呢？需要大量的就是说我们叫做 Training Data，就是训练数据，这个数据量越大，结果就越准，现在如果这个研究人员，终于可以得到整个互联网上所有的内容的时候，那么你想想看，他的训练的数据就是整个互联网，那么其实这个对他的结果就非常有好处，所以我认为互联网时代，比如他的翻译就是革命性的，它是云计算一个非常好的应用。而且你可以从互联网上，看到有很多很多新鲜的想法出来，这个我们不用花时间再

讲了，但是说以互联网为中心的，我非常赞同，就是人类从来没有那么一个时候，可以让你在很简单的，你在一个小小的计算机前面，可以看到全世界所有的内容，从来没有过。

InfoQ 中文站：非常好，那么目前 Google，刚才你也提到了，他现在已经有了成熟的这么一个云计算的应用，我想问一下它的商务前景是什么样的？

谷雪梅：Google 这个公司挺有意思的，从来不收大家钱，而且不收终端用户钱，不收。如果你要是说几个产品，说我这个产品的商务模式是，我要收每个用户的钱，那这个公司，你是不会把它发布出去的，所以它的商务前景我想是在那里呢？首先有可能，我只说有可能，可以把广告放上去，你今天有一个自己的新鲜的商务想法，你可以用 Google 的云计算的技术，在网上有了自己的这么一个家，我们不收你钱，但是你可能需要很多很多的资源，那这个时候这个怎么办？我们也许可以把广告放在旁边，也不影响你整个商业运作，但是也许你会把一些广告给用户看，当然这是我个人的猜测，不代表公司。所以我认为这个商务前景，Google 有自己的一个理念，就是你把一个东西做得非常好，然后其他的就跟着来了，盈利就跟着来了，现在所以我想还是，Google 还是在摸索阶段，怎么把自己在公司内部非常成熟，然后做得很好的，做得很快，规模很大，然后也很便宜，这么好的一个云计算的系统推给大家，其实公司也在摸索，只是这个前景也很光明，比如 App Engine 在美国，他一发布出来，好多好多人想去用，然后在网上设一个网站，很快都用满了，Google 只给那么多资源，当然主要是想测试一下吗。然后现在大家有好多在等，然后 Google 就是往上加，但是想来用的人越来越多，所以大家只好都在等，所以那么现在怎么解决这个问题？也就是说它的商务的模式怎么样？其实还是在摸索。

InfoQ 中文站：他会向那些企业用户收钱吗？

谷雪梅：我想可以这样，这个我不能代表公司，只能说是我想企业用户，不过这是个双赢的结局，如果 Google 假如说收了钱，可以给用户带来更好的这个应用体验的话，我也许，只能说也许，但是 Google 一般来讲，它主要面对的这个客户还是个人客户多一点，然后所以也是广告模式也是比较成熟，所以往这个方面发展可能性更大一些。

那么如果说一个架构师要在 Google 的云计算平台，就是刚才提到这个 App Engine 上面做应用的话，他需要做那些准备？或者另外注意哪些事项？

我还是先讲讲做那些准备，注意事项，我想你在开发过程中可能总会遇到各种各样的问题，这个没有关系，可以跟其他用 App Engine 大家一起交流就好了。准备现在是这样子，App Engine 的 API 提供了，我个人认为还不算太完备，他现在只有 Python 开放的 API，他的 Java 的 API 还没有推出，但是很快会推出了，所以的话，像一般架构，就说如果咱们是写程序，程序员，工程师，可能更多的在语言方面，在 API 方面需要多注意一些，那么架构的话，我觉得更多的是一种它的概念的转换，就他原来可能是习惯是，他是软件开发的这套思路，你现在把它变过来，你要变成它是一个 Internet Service，它是一个互联网的服务，这个概念一定要转过来，如果这个概念转不过来的话，我觉得你会浪费 App Engine 资源。

InfoQ 中文站：有没有哪些比较特别注意的事项？

谷雪梅：特别注意的，想想看，比如说它对数据库什么的，这个是不太好，因为 Google 不是做数据库起家的，我们是做 File System 文件系统，都是拿这些东西起家的，所以的话，我觉得还是刚才我讲的，就说一定要从概念上转变过来，不要再拿传统的软件开发的模式再像 Google 的 App Engine，这样不行的。

OK，那么另外一个问题就是什么样的企业，或者组织是适合用云计算的，那么什么样的企业或组织就是还不方便用云计算，然后另外一个就是云计算的不足之处，你认为它在哪里？

就像我刚才讲得，我还是想回到刚才那个电网的例子，如果有那么一天云计算真的像电网一样方便了，你想没有企业不用电吧？当然云计算如果到了那一天的话，我想没有企业不适合用云计算，现在的话，这个云计算我个人认为，他基本上还是比较的，像先锋性质的这样一个实验性的东西，所以这个如果有些企业，他可以，比如说他想涵盖很大很大的规模，或者比如很多地方，他现在架构是那种比较分布式的，这个他可能跟云计算契合会比较方便一点，有些假如说传统的做财务软件的企业，然后你是开发的思路都是，像我刚才讲的是传统的思路，那么你转到互联网的服务上面来，可能会花一段段时间，但我觉得就是最终还是，大家可能都会往这个方向转，因为实在是，每个企业对于自己维护自己底下一大套的 IT 的东西是没有必要的，把这些都扔给云计算，都扔给那些服务器，然后减少自己的这些费用，然后把更多的精力投入到自己这种企业的商务开发上面会更好一些，我现在是这么认为。就

是互联网企业用云计算可能会更适合一些，跟互联网相关的 Video，网上的视频，或者网上很多很多的像这种，我们叫 Content Distribution，我觉得这个都比较适合。

InfoQ 中文站：那是不是说那些比较关键性的应用？包括一些购物还不太适合？

谷雪梅：我觉得你这个问题很有技巧，这个也不能说不适合了，像 App Engine 上面现在已经事实上有不少像这种电子商务的网站，已经在上面了，我只是想说这个，有人说 Google 还是做搜索引擎起家的，可能在这个比如说像金融的互联网安全，他是不是有特别的需要，在这个方面，可能我们关注的力度还不够，更多的关注尤其是安全方面，更多的是说搜索这个引擎所面临的问题，那么的确你说现在是不是有用户的数据都非常非常的那种敏感的，就可以使用云计算，这个的确是个问题。

InfoQ 中文站：那我们最后一个问题，来看一下云计算，就是你认为云计算在接下来一段时间，他会有什么样的发展和变化？

谷雪梅：我刚才讲说 Google 有 10 年的云计算经验了，都是在公司内部的，所以现在要推出这个概念，其实最重要的是说，让更多人去用它，让更多人用的话，首先让我们 Developer 要用，所以一定要有很好的 API，这点上是毫无疑问的，所以在云计算，Google 现在当然在这上面花很大的力气，各种各样的 API，地图的 API，然后 App Engine 用 API，然后我们叫 GData 就是网上的存储，各种各样的 API 越来越多，所以，那么随着 API 的发展，可能大家会越来越方便的使用这个云计算后面的这个技术架构，然后再比如说，我们还可能特别希望看到，有些什么那种叫杀手 Killer Application，杀手应用，这种最好出现一些，这样可以更好的让大家看到云计算它的威力在哪里，所以我想下面一段就是 API 和 Killer Application。

InfoQ 中文站：OK，非常感谢你接受我们的采访。

谷雪梅：没问题，谢谢。

观看完整视频：http://www.infoq.com/cn/interviews/guxuemei-cloudcomputing

相关内容：

- 演讲：认识云计算

- 基于 Microsoft 云计算平台的 Ruby SDK 发布
- 微软正式推出云服务平台——Windows Azure
- 厂商依赖是云计算采纳的障碍？
- 云是虚拟化的战略性选择吗？

[热点新闻]

讨论：衡量程序员的工作效率

作者 Sadek Drobi 译者 郑柯

与其他领域一样，在软件开发领域中，管理者需要评估程序员的绩效和项目的进度。然而，如何定义恰如其分的评价体系，很令人挠头。

计算代码行数（source lines of code, SLOC）是最常用的方式之一。不过，最近 Shahar Yair 和 Steve McConnell 指出了该方法的一系列重要缺陷。首先，使用代码行数之和无法有效评估一个项目的实际进度，因为它更注重行为而不是结果。最终产品在多大程度上依赖于代码的性能和质量，这也是代码行数无法说明的。因此，聚焦于此实际上是非常有限的工作效率测量方式。

SLOC 无法表明要解决的问题的复杂性，也不能以可维护性、灵活性、扩展性等等因素来说明最终产品的质量。说到质量，它反而可能起到负面作用。通过重构、使用设计模式会减少代码行数，同时提升代码质量。代码量大，可能意味着有更多不必要的代码、更高不必要的复杂性、更加僵化难懂。

更危险的是，要用这样一种不完整的视角来评价开发人员的绩效，会起到错误的激励作用。开发人员会因此更注重代码的数量，而不顾其对产品质量的损害，也不会从最终产品的角度考虑去优化他们的工作，他们甚至可能有意编写更多冗长无益的代码。“测量什么，就得到什么”，Steve McConnel 回忆。

他指出，有些问题可以通过测量度量功能点数解决掉。那么决定程序大小的因素就变成了输入、输出、查询和文件的数目。不过这种方式也有其缺陷。 McConnell 提出一些操作性上的问题，比如必须要有一个大家认可的功能点测量机制，而且要想把每个功能点映射到程序员身上也不容易。Daniel Yokomizo 是一位经过认证的功能点专家，他在评论中明确指出

了这种方式的其他问题：缺少测量功能点复杂度的工具；还需要考虑诸如代码共享、框架、程序库之类的事情。这些都会影响到完成一个功能的时间。

有很多人参与了对于测量方式的讨论，他们都同意这些做法有其局限，不过他们都觉得衡量开发人员的绩效还是有必要的。实际上，不少人认为 SLOC 可以作为基础，在其之上通过考虑多种不同因素来进行更复杂的分析。McConnell 提出了四条分析开发人员工作效率的必备指导原则，他们也都同意。这四条原则如下：

1. 不要指望单一维度的工作效率测量方式能告诉你每个人的真实情况。
2. 不要指望任何测量方式可以在很小的粒度上区分出每个人的工作效率差异。这些方式可以为你提出问题，却不会告诉你答案。
3. 牢记：趋势总是比单独一点的测量来得重要。
4. 问问你自己：为什么要测量个人的工作效率？

测量开发人员的工作效率在什么样的上下文中才是有意义的？有哪些条件？这些条件该如何组合？许多问题仍没有答案。如果你有过类似经验可以分享，请不要犹豫。

原文链接：http://www.infoq.com/cn/news/2008/10/measure-programmers-productivity

相关内容：

- 评价并改进架构能力
- 老资格、尊重、威信和敏捷团队
- 没有抱怨声的迭代
- 给敏捷团队发奖金就像在刀尖上跳舞
- 简析 Sun 在中国的 Java 认证培训策略

[热点新闻]

厂商依赖是云计算采纳的障碍？

作者 Jean-Jacques Dubray **译者** 徐涵

Tim Bray 上周写了一篇关于云计算采纳的文章。他觉得存在两大主要问题：

- 我们还没有找到最佳的云平台(cloud platforms)架构方案。是采取 Amazon 的 EC2/S3 “裸虚拟白盒（ Naked virtual whitebox)” 模型？还是采取像 Google App Engine 那样平台即服务（ Platform-as-a-service ）的风格？我们尚不知晓。
- 如果云平台要受到大家欢迎的话，那么它必须做到绝对不能出现厂商依赖的情况。

Tim 最后说道：

> 在 21 世纪的今天仍然步桌面套件和私有 SQL 的后尘、重演厂商依赖的悲剧，我们“脑残”到这个地步了吗？你知道，那等于把 IT 预算控制权交给平台厂商了？

Dare Obasanjo 不明白能够避免厂商依赖将意味着什么：

> 与 Web 开发平台的切换相比，从一个应用切换到另一个应用是一个类似且更容易的问题。

他认为，对于基于“云”的办公套件：

> 由于有 ODF 和 OOXML 这些标准，因此应该很容易对业务文档进行迁移。

但是他也敦促大家考虑以下问题：

> 有没有自动化批量执行导入导出的办法？是不是大家只能手工进行在线文档与标准格式间的导出/导入？若数据迁移导致指向公司数据的链接和引用失效，会有什么影响？你们公司迁移数据需要多大的代价（包括公司因切换服务而停工，以及 IT 部门迁移所有数据的实际开销）？

最后，Dare 建议道：

> 切换寄存应用提供商是件很好办的事...尽管技术上可行，然而大部分寄存应用提供商都做不到令用户可以简单方便地彻底迁入或迁出它们的服务。

至于云计算平台，Dare 解释道：

> 你面临着跟以上情形同样的问题，以及一些额外的问题。云计算平台关键的缺点是，这些服务背后缺乏标准化的 APIs 与平台技术...要避免厂商依赖，需要各个提供商实现同一组底层 APIs 才行。否则，云计算平台之间的迁移就会像把 Ruby on Rails+MySQL 切换为 Django+PostgreSQL 这么麻烦（等于要完全重写代码）。

Google 的 Dewitt Clinton 在评论 Tim Bray 的文章时解释道：

> 对 App Engine 来说，人们随时可以在其它提供商上运行开源的 userland stack（它暴露有 API），而且有很多开源的 bigtable 实现可供你选择。尽管如此，批量导出数据仍需手工完成，不过目前这是切实可行的，而且我们正在努力简化这个步骤。

Dewitt 还说，Dare 所说的已经在发生了：

> 但 Amazon 或 Google 除了采用正确的许可证以及尽可能编写更多的开源代码以外，如何促成它实现呢？显然，我们不会架设起多个备选实例。（尽管我们可以为之喝彩，就像当我们看到在 EC2 和 S3 上实现了 App Engine API 时所做的一样。）

Mark Pilgrim 觉得云计算还存在另一个问题：

> 真正令人害怕的问题不是厂商依赖，而是封禁。比方说，如果你的服务提供商突然认定你的帐户在进行可疑活动，然后禁用你的帐户，你去向谁求助？当人们得知自己在没有明显理由的情况下被禁止访问权限、而且无法知道真正原因、别无他法只有重头来过时，通常会感到万分震惊。但愿他们有做备份。

厂商依赖（或封禁）是采纳基于“云”的方案（SaaS、PaaS、IaaS）的主要症结吗？或者，你觉得一个厂商在提供更好的产品的同时，会不会像现实世界里的各种服务一样周到、为你解决迁移问题？

原文链接：http://www.infoq.com/cn/news/2008/10/cloud-architecture

相关内容：

- 亚马逊 EC2 服务：从 beta 版转换到生产环境
- Ruby 和 Rails 软件栈概览
- 数据库即服务是个坏主意吗？
- David Chappell：云计算简介
- CloudCamp 的 Reuven Cohen 谈虚拟化和云计算

[热点新闻]

移动 SOA 的门柱

作者 Boris Lublinsky **译者** 胡键

John Evdemon 在其最新的一篇帖子展示了针对 SOA，业界进行定义、重定义和自相矛盾的尝试。这些定义五花八门，无章可循。

首先是关于松耦合的大体定义：

> 随着 Web 服务和 SOA 的来临，我们正试图创建耦合更松的架构和系统。松耦合系统提供了许多好处，包括：支持运行时迟绑定或动态绑定到其它组件，可以化解组件结构中的差异，安全模型、协议和语义，从而对易变性进行了抽象。

接着，重用占据了舞台中央：

> 软件重用的核心思想并不是什么新东西，即从最初已经创建出的组件中获取更多价值。在 SOA 之前，由于组织内业务单位之间缺乏集中控制和沟通，导致相同的解决方案一再重复发明。为了确保组织内各业务单位不创建重复服务，SOA 干系人，如业务分析师和架构师，应该能够以一种广为人知和系统的方法来寻找现有服务并评估它们是否能够重用。

业界最后发现真正的重用实在是太复杂了并且并不是总能实现：

> 创建可重用服务要求对未来进行预测……服务创建者怎么可能准确地猜到未来应用的需求？“车到山前必有路”的想法非常难以实现真正的重用。加上，鼓励所创建组件可被其他小组重用的组织，即使有，也不多。

接着，它谈到了业务流程的松耦合：

> 硬连接流程应该是一种罪恶，我们要坚决地抵制它……松耦合将迫使我们反思业务活动的方方面面……它将对我们管理业务操作的方式产生深远影响。厚重的流

> 程手册留下了对我们硬连接业务操作方式的遗嘱。预先指定全部活动的详细细节，然后监测所有活动来清除任何不符合标准的变化。尽可能地加固操作。松耦合业务流程的运转方式完全不同。

当 SOA 无能为力之时，事件驱动架构（EDA）成了（一个更流行的）SOA 替补。：

> EDA 交付了 SOA 只是承诺却没有做到的松耦合。它并非同步的“命令并控制”类型的模式，而是其反面：异步的“发布并订阅”类型的模式。发布者和订阅者完全不知道对方的存在；在某种程度上，组件是松耦合的：它们只共享了消息的语义。

接下来，它谈到了业务/IT 的对齐：

> 自从发迹于面向对象设计和基于组件的软件开发方法论，SOA 已经进入了理想的崇高领域。随着故事的发展，SOA 不单单是为重振 IT 而设计的了；它还成了能改变 IT 所服务于的业务的魔弹。

并且，最终它发现 SOA 跟技术没有任何瓜葛：

> 太令人震惊了；SOA 似乎是第一个对技术没要求的技术。大厂商们告诫说，只要你坚持走有 SOA 特色的架构基本路线不动摇，在技术上不管你做什么都没关系。

这些定义明显的不一致似乎是来自于以下几个因素：

和任何架构一样，SOA 相当复杂且涉及多个关注点。随着 SOA 实现经验的增加，架构师和开发者都发现，他们在实现中需要开始考虑更多的事情。这并非真的移动了门柱，而是意识到“操场”的真正尺寸。

- 相比起正确定义他们确实要做的东西，人们往往更关心变得“时髦”。考虑当前的 SOA 流行程度，许多东西被人为的拉入 SOA 领域并非是因为它们真的就属于那儿，而是因为术语引起了 CEO/CIO 的注意。这并非真的移动了门柱，而是使用流行词汇来引起管理层的更多注意。
- 人们往往把 SOA 架构和 SOA 平台/实现混为一谈。尽管 SOA 架构没有改变，但是平台/实现的门柱却总是在往更好地支持客户需求的方向移动。

原文链接：http://www.infoq.com/cn/news/2008/11/SOAGoalPost

相关内容：

- 视频采访：与阿里软件首席架构师赵进探讨了 SaaS
- 观点：REST 在 SOA 中适合哪个位置？
- SOA 中的数据联邦技术解密
- 文章：支付宝首席架构师程立谈架构、敏捷和 SOA 实践
- 文章：SOA 治理基础

[热点新闻]

如何实现真正的 REST 风格？

作者 Mark Little **译者** 徐涵

SocialSite 的 REST API 最近因 Roy Fielding 称其不符合 REST 风格而受到批评。Roy 说，它是众多自称符合 REST 风格而实则不然的系统之一。

> OpenSocial 的 REST API 是 RPC 式的，而且是公然宣誓其 RPC 本性。它在如此多的方面存在耦合，所以理应将它评为“差”。

从 OpenSocial 网页上提供的信息来看，你不难同意 Roy 的观点。例如：

- 在服务端为 OpenSocial 风格的 REST 和 JSON-RPC 提供支持
- 在客户端为 JSON-RPC 批量请求提供支持
- 服从 OpenSocial 对扩展的需求

另外，我们都知道 REST 跟 RPC 是紧密相关的。 鉴于已经见过很多自称符合 REST 风格而实则不然的网站，Roy 接着就如何构建真正符合 REST 风格的网站（及 API）给予了指导。现部分摘录如下：

- REST API 不应依赖于某一个通信协议。如果一个协议元素（protocol element）要将 URI 用于标识的目的，那么它必须允许采用任意 URI 方案（scheme）。[做不到这一点，就意味着标识与交互没有分离。]
- 媒体类型（media types）是用于表示资源和推进应用状态（application state）的，REST API 应将绝大部分描述精力用于定义媒体类型，或者是为现有的标准媒体类型定义扩展的关系名称（relation names）和/或基于超文本的标记（markup）。如果要就“对所谈及的 URIs 采用什么方法”进行定义的话，那么应完全把它放在媒体类

型的处理规则范围内进行定义（不过在大多数情况下，现有媒体类型都已经定义好了）。[做不到这一点，就意味着交互不是由超文本、而是由外部信息（out-of-band information）推进的。]

- REST API 一定不能定义固定的资源名称或层次。服务器必须可以自由控制它自己的名称空间（namespace）。应该像 HTML 表单（HTML forms）和 URI 模版（URI templates）那样，通过媒体类型和链接关系（link relations）给出指示，使得服务器可以指导客户端如何构造正确的 URIs。[做不到这一点，就意味着客户端在根据外部信息（比如跟领域相关的标准）假定资源结构——相当于面向数据方法里的 RPC 功能耦合。]
- 要使用 REST API，应该只需知道初始 URI（书签）和一套适合于目标用户（即可被任何使用该 API 的客户端所理解）的标准媒体类型。这样的话，所有的应用状态迁移，都必须以“客户端在服务器提供的选项里挑选”这样的方式进行；服务器提供的选项，或者直接出现在用户收到的表示（representations）里，或者在用户对那些表示进行处理后得到。客户端可以根据自己所掌握的关于媒体类型与资源通信机制的知识来决定（或限制）状态转移，客户端可以即时增加对媒体类型与资源通信机制的支持（比如通过代码请求）。[做不到这一点，就意味着交互不是由超文本、而是由外部信息推进的。]

Roy 的这篇文章收到了很多反馈，有的是直接回复评论，有的是另发文章，其中有人提出了一些关于超文本/超媒体使用的问题，对此 Roy 回答道：

> 我所说的超文本（hypertext）指的是信息与控件的同时呈现，这样一来，信息便具有自解释性（affordance）了，从而用户（或程序）可以通过它获取选项、并作出选择。超媒体（hypermedia）只是对文本的含义加以延伸、在媒体流里增 加了时间锚（temporal anchors）；大部分研究者已经不对它们加以区分了。超文本不一定就是浏览器里的 HTML。机器只要理解数据格式和关系类型，它就可以跟随链接。

当被问及为什么他觉得很多人未能正确实现 REST 风格时，Roy 说：

> 某种程度上，人们未能正确实现 REST 风格，是因为我在博士论文里没有就媒

> 体类型设计（media type design）作充分详细的论述。那并不是因为我觉得媒体类型设计不如 REST 的其他方面重要，而是因为我当时时间不够。还有，我想很多人做得不对，可能是因为他们仅仅阅读了根据非权威资料撰写而成的 Wikipedia 相关条目。不过，我觉得很多人存在一个错误的认识，他们认为：设计简单的东西，应该是轻而易举的。而在现实中，设计某样东西需要花费的精力，与结果的简单程度是成反比的。与其他架构风格相比，REST 是相当简单的。REST 是用于长远考虑的软件设计：它的每一个细节都是为了提升软件寿命和独立演化。有许多约束是直接与短期功效对立的。不幸的是，人们较擅长于短期设计，而对待长期设计就很糟糕了。大部分人认为他们不需要为以后的版本作考虑。有不少软件方法都把长远 考虑说成是执迷不悟的、象牙塔的设计（若不是有实际需求的话，那么可能是的）。

实际上，如果你对 REST 感兴趣的话，对该文的所有回复都值得一读。Dare Obasanjo 在一篇单独的文章中进行了概括总结：

> 最要铭记的是，REST 所构建的是在万维网（World Wide Web）上使用、对 Web 生态系统有利的软件。理想情况下，一个 REST 风格的 API 既可为众多网站所用、又可被运行在各种平台上的应用所用，且客户端应用与 Web 服务之间是零耦合的。RSS/Atom 提要（feed）就是一个很好的例子，它也是世界上最成功的 REST 式 API。

他专门考察了 Roy 提到了一种错误做法：实现 API 的服务需具有一种特定的 URI 结构。

> 这种做法的问题是，它假定每一个实现者都对他们的 URI 空间拥有完全控制权，而且客户端应该把 URL 结构写进代码里去。Joe Gregorio 在文章《No Fishing - or - Why 'robots.txt and 'favicon.ico' are bad ideas and shouldn't be emulated》里很好地解释了这一做法不好的原因，他在文章中列出了写死 URL 不好的几点理由，比如：缺乏扩展性；不支持那些采用寄存环境、从而无法控制 URI 空间的用户。

网上有大量其他 REST 资源（双关）。它们大部分由权威人士编写，并记录下了有关实现。不过显而易见的是，并不是所有使用 Web 的应用都是 REST 式应用，也并不是所有自称符合 REST 风格的应用都符合。

原文链接：http://www.infoq.com/cn/news/2008/10/rest-api

相关内容：

- 文章：AtomServer：数据分发的发布动力
- Netflix 发布 REST API
- 文章：REST 反模式
- 视频采访：Dan Diephouse 谈 Atom、AtomPub、REST 和 Web 服务
- SOAP 协议栈是令人尴尬的失败？

[热点新闻]

Memcached 的 JGroups 实现支持失败转移和 JMX

作者 Srini Penchikala **译者** 宋玮

Memcached 是一个分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。Memcached 基于的是一个存储键/值对的 hashmap。其守护进程（daemon ）是用 C 写的，但是客户端可以用任何语言来编写，并通过 memcached 协议与守护进程通讯。但是它并不提供冗余（例如，复制其 hashmap 条目）；当某个服务器 S 停止运行和崩溃了，所有存放在 S 上的键/值对都将丢失。

Bela Ban，JBoss 的 JGroups 和 Clustering 团队的领导，最近写了一个基于 JGroups 的 memcached 实现，它允许 Java 客户端直接访问 memcached。该实现完全是用 Java 编写的，而且拥有少量优于 memcached 框架的特性 ：

- Java 客户端和 PartitionedHashMap（org.jgroups.blocks.PartitionedHashMap）可以在同一地址空间运行，因此不需要使用 memcached 协议进行通信。这使得 servlet 可以直接访问缓存，而不需要在其上进行序列化。
- 所有 PartitionedHashMap 进程彼此知道对方，当一个集群成员发生改变时，它们能够决定做什么。例如，一个要停止服务的服务器可以把它管理的所有键都迁移到下一个服务器。使用 memcached，存 放在 S 服务器上的条目在 S 关机的时候就丢失了。
- 当一个集群成员发生改变时（例如，一个新服务器 S 启动了），那么所有服务器都检查自己所保存的一个条目事实上是否应该存放在 S 上。它们会把所有条目都转给 S。这样的好处是不需要再次从 DB 中 重新读取这些条目并插入到缓存中（memcached 正是这么做的），但是缓存要自动地使自己重新达到平衡 。

- PartitionedHashMap 拥有一个一级缓存（level 1 cache——L1 cache）。这就可以使缓存的数据离真正需要它的地方很近。例如，如果我们拥有 A、B、C、D 和 E 几个服务器，且一个客户端给 C 增加了一个（要被高度访问的）报纸文章，那么 memcached 总是把所有对该文章的单一请求都转给 C。这样，一个正在访问 D 的客户端总是会触发一个从 D 到 C 的 GET 请求，并返回一篇文章。JGroups 在第一次访问时把这篇文章缓存在 D 的 L1 缓存中，这样所有从 D 访问这篇文章的其它客户端将获得这个被缓存的文章，因而我们可以避免又一轮对 C 的访问。注意，每个条目都有其失效时间，它将导致该条目在失效时被从 L1 缓存中删除，那么下一个访问将不得不重新从 C 获取该文章并再次把它放在 D 的 L1 缓存中。这个失效时间是由该文章的提交者定义的。

- 因为 GET、SET 和 REMOVE 的 RPC 都使用 JGroups 作为传输，传输的类型和服务的质量可以通过定义传输的底层 XML 文件来控制和定制。例如，我们可以压缩或者加密所有 RPC 信息。它还让我们可以选用 UDP （IP 多点传送和/或 UDP 数据报）或 TCP。

- 连接器（org.jgroups.blocks.MemcachedConnector）负责分析 memcached 协议并调用 PartitionedHashMap 上的请求（PartitionedHashMap 代表了 memcached 的实现），服务器（org.jgroups.demos.MemcachedServer）和 L1 及 L2 缓存（org.jgroups.blocks.Cache）可以被随意装配或替代。因此定制 JGroups memcached 实现很简单；比如使用一个不同的 MemcachedConnector 来处理二进制协议（当然需要与客户端代码匹配）。

- 所有管理信息和操作经由 JMX 被暴露。

启动 JGroups memcached 实现的主类是 org.jgroups.demos.MemcachedServer。它创建了一个 L1 缓存（如果配置了）、一个 L2 缓存（存储所有条目的默认 hashmap）、以及一个 MemcachedConnector。API 非常简单且包含如下缓存方法：

- public void put(K key, V val)：按照默认的缓存时间把键/值对存入缓存

- public void put(K key, V val, long caching_time)：与上面方法相同，但是可以定义缓存失效时间。0 表示永远缓存，-1 表示不缓存，任何正值代表该条目要缓存的毫秒数
- public V get(K key)：获得键 K 所对应的值
- public void remove(K key)：从缓存（L2 及 L1，如果启用的话）中删除一个键/值对

InfoQ 就 memcached 的 JGroups 实现背后的动机采访了 Bela Ban。他说 memcached 的 JGroups 实现使得他们可以试验分布式缓存并看看不同的缓存策略适应 JBoss 集群的程度如何。他还阐明了这个新的 memcached 实现与 JBossCache 缓存框架的比较：

> 我们把缓存看作是一个连续统一体 :从分布式缓存(数据跨越集群中多个节点，但是没有 冗余)到完全复制数据缓存(每个数据条目整体复制到每个集群节点上)。在分布式和整体复制之间， 我们还有 buddy replication，它只把数据复制到一些选定的后备节点上。这可以被比作 RAID，RAID 0 没有冗余（分 布式），RAID 0+1 是全冗余，而 RAID 5 是部分冗余。
>
> 当前，JGroups 的 PartitionedHashMap 提供了分布式缓存，JBossCache 提供了全复制和部分复制（使用 Buddy Replication）缓存。其想法是让用户定义他们要放在集群中的 K（每个数据项——per data item ）值，K=0 表示分布式，但是如果一个节点保存有一个或多个条目，节点崩溃则数据就会丢失；K=X（这 里 X
>
> memcached 的 JGroups 实现是尝试 K=0 的第一步，它是纯数据分布式缓存，没有冗余。其最终会被纳入到 JBossCache 中。

memcached 实现适合放在 JBoss 应用服务器的哪个模块？

> 它将成为 Clustering 子系统的一部分，由 JBossCache 提供。注意我们的实现是真正给“Java”客户端写的，因此不必使用那些非常低效的 memcached 协议，而是在上层使用了编组（marshalling）/解读（unmarshalling）/复制（copying）。

谈到使用 memcached 的 JGroups 实现的典型使用场景，Bela 说道：

> 运行在 JBoss 或 Tomcat 集群上的服务器端代码（例如 servlets），其访问一个 DB 并需要缓存以提高速度并避免 DB 瓶颈。其它使用场景也类似，只是访问的不是

DB 而是文件系统。例如，一个 HTML 页面缓存服务器（ Squid 立刻浮现在脑海里 ）。

有无计划将来把 memcached 引入到 JBoss 应用服务器中。

当然有。数据分区（ Data Partitioning ）特性将使得用户可以按照自己的需要来配置缓存。这样使得分布式缓存看上去不像是一个新特性，而是 JBossCache 的配置而已。更酷的是这是动态的，因此开发者可以决定他们所放进 JBossCache 的每个数据项（per data item）要使用哪种冗余特性（none=distribution，full=total replication 或 partial ）。

至于该项目新特性的未来方向，Bela 罗列了要做的一些事情：

- 基于缓存中的字节数而不是元素数提供一种逐出策略。
- 把从远端服务器接受到元素存储为 byte[] buffer 而不是对象。在第一次访问时，把 byte buffer 解读成对象。这在 JBoss 的 HTTP 会话复制代码中被使用且一直表现良好：因为不需要解读过程因此不会影响到性能。
- 实现全部 memcached 协议：现在我只提供 GET、GET-MULTI、SET 和 DELETE。尽管其它的（ APPEND 、PREPEND、CAS ）很容易实现，但是我还没有做，因为 Java 客户端的主要使用场景位于我们 memcached 实现的同一 JVM 中，因此不需要 memcached 协议。
- 提供一个更好的一致散列的实现。

memcached 的 JGroups 实现和其依赖类库可以从其 sourceforge 站点上下载。下面是运行该程序的命令：

```
java -jar memcached-jgroups.jar
```

Bela 正在期待着社区的反馈。他说这是一个试验特性，但是将成为 JBossCache 支持的一个特性，社区意见将会极大影响这一特性的方向。

原文链接：http://www.infoq.com/cn/news/2008/10/jgroups-memcached

相关内容：

- 文章：Hadoop 基本流程与应用开发
- 使用 EhCache Server 部署 1TB 缓存
- Grails 获得 Morph AppSpace 云计算托管服务的支持
- JBoss Cache 分布式缓存：Manik Surtani 访谈
- 观点与争锋：多重处理器计算的挑战远在技术层面之上

[热点新闻]

社区对 SpringSource 的改变反应强烈

CEO Rod Johnson 忙出来澄清

作者 Scott Delap **译者** 张龙

本周末，使用广泛的开源框架 Spring 背后的公司 SpringSource 发布了一个新的维护方案：

> 使用 SpringSource Enterprise 的客户，如果其订阅有效，将会收到持续三年的维护版本，从主要的新版本的 GA 版开始。这些客户会收到进行中的、快速的修复，同时还会收到通常的维护发布，用以修复 bugs、改进安全及可用性，这会使 SpringSource Enterprise 成为最好的产品系统。
>
> 在 Spring 新的主版本发布后，我们就会在 3 个月内连续发布社区维护更新以解决开始时的稳定问题。随后的维护发布将会面向 SpringSource Enterprise 的客户。Bug 修复将被放在开源开发的主干上，并且在下一个主要的社区版发布时可用...

以上表述引发了很多争论。Daniel Gredler 推测到既然修复还会被打到公共的源上，那么开发者就可以从源代码中构建其自己的发布。SpringSource 的 Mark Brewer 随后证实了该推测。基于这一点，InfoQ 将该新策略总结为以下几点：

- Spring 的源代码树依然会保持开放
- SpringSource 将根据需要继续为主版本和关键版本创建官方发布。
- Enterprise 的客户可以访问所有这些构建。
- 开源用户可以在主版本发布后的前三个月内访问这些构建。
- 社区可以进行构建，但他们得不到 SpringSource 的官方支持

接下来 InfoQ 向 SpringSource 的 Rod Johnson 咨询他对维护策略和我们的总结的看法：

> 现在随着越来越多的 Spring 版本应用于产品中，SpringSource（或者其他任何人）不可能再为所有这些发布提供免费、高质量的维护了。
>
> 我们提出的这个策略既满足了开源社区的需要（他们想访问最新的源代码），也满足了保守的企业用户的需要（他们需要对旧版本的 Spring 的支持，因为他们不能或者不想遵循典型的开源实践——升级到最新版本）。
>
> 该策略不会对深信开源的技术人员造成影响。他们依旧可以访问源代码，而这些源代码是 SpringSource 花钱开发的。它只会影响到那些不想与开源走的太近或者不想升级到最新版的人。对于那些不太能承担风险的组织，3 年的维护期加上对源代码 24x7 的支持是促使其购买 SpringSource Enterprise 的重要因素。
>
> 在 SpringSource 内部，我们为自己对开源的贡献而感到骄傲和自豪。这超出了 Spring 的范畴：我们还是 Tomcat、Apache HTTPD web 服务器等很多其它项目的主要贡献者。我们拥有大量在企业级 Java 方向有才能的人，同时我们使其可以向开源作出贡献，而这一切在以前是不可想象的。

对最近 Spring 的声明感兴趣的读者可能还会对 Peter Mularien 关于 Spring 核心开发和提交者的分析感兴趣。

原文链接：http://www.infoq.com/cn/news/2008/10/springsource-maintenance_zn

相关内容：

- Polyforms——减少 DAO 代码重复
- 文章：对话 Spring.NET
- 视频采访：Joe Walker 谈 DWR
- 文章：Spring MVC 中的新特性
- Spring MVC 的安全隐患及建议

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

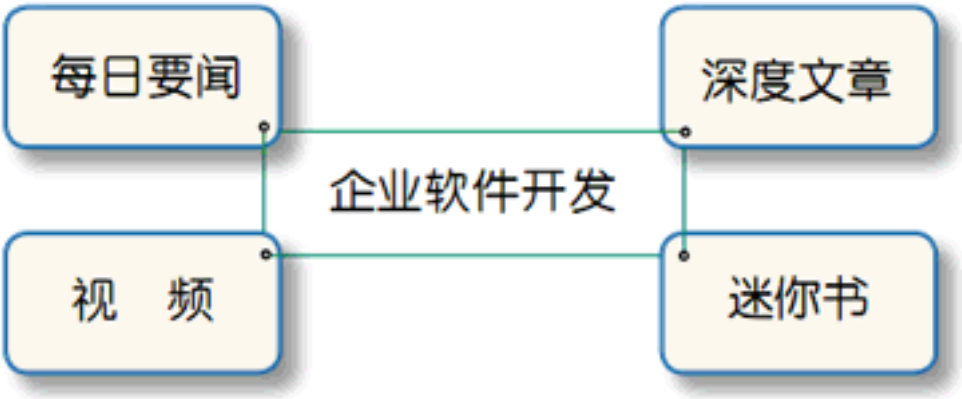
我们的**定位**：关注高级决策人员和大中型企业

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]

Java 6 中的线程优化真的有效么？

作者 Jeroen Borgers 译者 韩锴

推荐理由：在最新的JVM 中，偏向锁、锁粗化、逸出分析与锁省略、自适应自旋锁这些技术真的能在高并发的程序中发挥作用么？Jeroen Borgers 在本文中尝试进行了探索。本篇原分为上下两部，在这里我们把它们合二为一了。

介绍 — Java 6 中的线程优化

Sun、IBM、BEA 和其他公司在各自实现的 Java 6 虚拟机上都花费了大量的精力优化锁的管理和同步。诸如偏向锁（biased locking）、锁粗化（lock coarsening）、由逸出（escape）分析产生的锁省略、自适应自旋锁（adaptive spinning）这些特性，都是通过在应用程序线程之间更高效地共享数据，从而提高并发效率。尽管这些特性都是成熟且有趣的，但是问题在于：它们的承诺真的能实现么？在这篇由两部分组成的文章里，我将逐一探究这些特性，并尝试在单一线程基准的协助下，回答关于性能的问题。

悲观锁模型

Java 支持的锁模型绝对是悲观锁（其实，大多数线程库都是如此）。如果有两个或者更多线程使用数据时会彼此干扰，这种极小的风险也会强迫我们采用非常严厉的手段防止这种情况的发生——使用锁。然而研究表明，锁很少被占用。也就是说，一个访问锁的线程很少必须等待来获取它。但是请求锁的动作将会触发一系列的动作，这可能导致严重的系统开销，这是不可避免的。

我们的确还有其他的选择。举例来说，考虑一下线程安全的 StringBuffer 的用

法。问问你自己：是否你曾经明知道它只能被一个线程安全地访问，还是坚持使用 StringBuffer，为什么不用 StringBuilder 代替呢？

知道大多数的锁都不存在竞争，或者很少存在竞争的事实对我们作用并不大，因为即使是两个线程访问相同数据的概率非常低，也会强迫我们使用锁，通过同步来保护被访问的数据。"我们真的需要锁么？"这个问题只有在我们将锁放在运行时环境的上下文中观察之后，才能最终给出答案。为了找到问题的答案，JVM 的开发者已经开始在 HotSpot 和 JIT 上进行了很多的实验性的工作。现在，我们已经从这些工作中获得了自适应自旋锁、偏向锁和以及两种方式的锁消除（lock elimination）——锁粗化和锁省略（lock elision）。在我们开始进行基准测试以前，先来花些时间回顾一下这些特性，这样有助于理解它们是如何工作的。

逸出分析 — 简析锁省略（Escape analysis - lock elision explained）

逸出分析是对运行中的应用程序中的全部引用的范围所做的分析。逸出分析是 HotSpot 分析工作的一个组成部分。如果 HotSpot（通过逸出分析）能够判断出指向某个对象的多个引用被限制在局部空间内，并且所有这些引用都不能"逸出"到这个空间以外的地方，那么 HotSpot 会要求 JIT 进行一系列的运行时优化。其中一种优化就是锁省略（lock elision）。如果锁的引用限制在局部空间中，说明只有创建这个锁的线程才会访问该锁。在这种条件下，同步块中的值永远不会存在竞争。这意味这我们永远不可能真的需要这把锁，它可以被安全地忽略掉。考虑下面的方法：

```
publicString concatBuffer(String s1, String s2, String s3) {,
StringBuffer sb = new StringBuffer();
sb.append(s1);
sb.append(s2);
sb.append(s3);
return sb.toString();
 }
```

图 1. 使用局部的 StringBuffer 连接字符串

如果我们观察变量 sb，很快就会发现它仅仅被限制在 concatBuffer 方法内部了。进一步说，到 sb 的所有引用永远不会"逸出"到 concatBuffer 方法之外，即声明它的那个方法。因此其他线程无法访问当前线程的 sb 副本。根据我们刚介绍的知识，我们知道用于保护 sb

的锁可以忽略掉。

从表面上看，锁省略似乎可以允许我们不必忍受同步带来的负担，就可以编写线程安全的代码了，前提是在同步的确是多余的情况下。锁省略是否真的能发挥作用呢？这是我们在后面的基准测试中将要回答的问题。

简析偏向锁（Biased locking explained）

大多数锁，在它们的生命周期中，从来不会被多于一个线程所访问。即使在极少数情况下，多个线程真的共享数据了，锁也不会发生竞争。为了理解偏向锁的优势，我们首先需要回顾一下如何获取锁（监视器）。

获取锁的过程分为两部分。首先，你需要获得一份契约.一旦你获得了这份契约，就可以自由地拿到锁了。为了获得这份契约，线程必须执行一个代价昂贵的原子指令。释放锁同时就要释放契约。根据我们的观察，我们似乎需要对一些锁的访问进行优化，比如线程执行的同步块代码在一个循环体中。优化的方法之一就是将锁粗化，以包含整个循环。这样，线程只访问一次锁，而不必每次进入循环时都进行访问了。但是，这并非一个很好的解决方案，因为它可能会妨碍其他线程合法的访问。还有一个更合理的方案，即将锁偏向给执行循环的线程。

将锁偏向于一个线程，意味着该线程不需要释放锁的契约。因此，随后获取锁的时候可以不那么昂贵。如果另一个线程在尝试获取锁，那么循环线程只需要释放契约就可以了。Java 6 的 HotSpot/JIT 默认情况下实现了偏向锁的优化。

简析锁粗化（Lock coarsening explained）

另一种线程优化方式是锁粗化（或合并，merging）。当多个彼此靠近的同步块可以合并到一起，形成一个同步块的时候，就会进行锁粗化。该方法还有一种变体，可以把多个同步方法合并为一个方法。如果所有方法都用一个锁对象，就可以尝试这种方法。考虑图 2 中的实例。

```
public static String concatToBuffer(StringBuffer sb, String s1, String s2, String s3) {
sb.append(s1);
sb.append(s2);
```

```
        sb.append(s3);
        return
        }
```

图 2. 使用非局部的 StringBuffer 连接字符串

在这个例子中，StringBuffer 的作用域是非局部的，可以被多个线程访问。所以逸出分析会判断出 StringBuffer 的锁不能安全地被忽略。如果锁刚好只被一个线程访问，则可以使用偏向锁。有趣的是，是否进行锁粗化，与竞争锁的线程数量是无关的。在上面的例子中，锁的实例会被请求四次：前三次是执行 append 方法，最后一次是执行 toString 方法，紧接着前一个。首先要做的是将这种方法进行内联。然后我们只需执行一次获取锁的操作（为整个方法），而不必像以前一样获取四次锁了。

这种做法带来的真正效果是我们获得了一个更长的临界区，它可能导致其他线程受到拖延从而降低吞吐量。正因为这些原因，一个处于循环内部的锁是不会被粗化到包含整个循环体的。

线程挂起 vs. 自旋（Thread suspending versus spinning）

在一个线程等待另外一个线程释放某个锁的时候，它通常会被操作系统挂起。操作在挂起一个线程的时候需要将它换出 CPU，而通常此时线程的时间片还没有使用完。当拥有锁的线程离开临界区的时候，挂起的线程需要被重新唤醒，然后重新被调用，并交换上下文，回到 CPU 调度中。所有这些动作都会给 JVM、 OS 和硬件带来更大的压力。

在这个例子中，如果注意到下面的事实会很有帮助：锁通常只会被占有很短的一段时间。这就是说，如果能够等上一会儿，我们可以避免挂起线程的开销。为了让线程等待，我们只需将线程执行一个忙循环（自旋）。这项技术就是所谓的自旋锁。

当锁被占有的时间很短时，自旋锁的效果非常好。另一方面，如果锁被占有很长时间，那么自旋的线程只会消耗 CPU 而不做任何有用的工作，因此带来浪费。自从 JDK 1.4.2 中引入自旋锁以来，自旋锁被分为两个阶段，自旋十个循环（默认值），然后挂起线程。

自适应自旋锁（Adaptive spinning）

JDK 1.6 中引入了自适应自旋锁。自适应意味着自旋的时间不再固定了，而是取决于一

个基于前一次在同一个锁上的自旋时间以及锁的拥有者的状态。如果在同一个锁对象上，自旋刚刚成功过，并且持有锁的线程正在运行中，那么自旋很有可能再次成功。进而它将被应用于相对更长的时间，比如 100 个循环。另一方面，如果自旋很少发生过，它将被遗弃，避免浪费任何 CPU 周期。

StringBuffer vs. StringBuilder 的基准测试

但是要想设计出一种方法来判断这些巧妙的优化方法到底多有效，这条路并不平坦。首要的问题就是如何设计基准测试。为了找到问题的答案，我决定去看看人们通常在代码中运用了哪些常见的技巧。我首先想到的是一个非常古老的问题：使用 StringBuffer 代替 String 可以减少多少开销？

一个类似的建议是，如果你希望字符串是可变的，就应该使用 StringBuffer。这个建议的缘由是非常明确的。String 是不可变的，但如果我们的工作需要字符串有很多变化，StringBuffer 将是一个开销较低的选择。有趣的是，在遇到 JDK 1.5 中的 StringBuilder（它是 StringBuffer 的非同步版本）后，这条建议就不灵了。由于 StringBuilder 与 StringBuffer 之间唯一的不同在于同步性，这似乎说明，测量两者之间性能差异的基准测试必须关注在同步的开销上。我们的探索从第一个问题开始，非竞争锁的开销如何？

这个基准测试的关键（如清单 1 所示）在于将大量的字符串拼接在一起。底层缓冲的初始容量足够大，可以包含三个待连接的字符串。这样我们可以将临界区内的工作最小化，进而重点测量同步的开销。

基准测试的结果

下图是测试结果，包括 EliminateLocks、UseBiasedLocking 和 DoEscapeAnalysis 的不同组合。

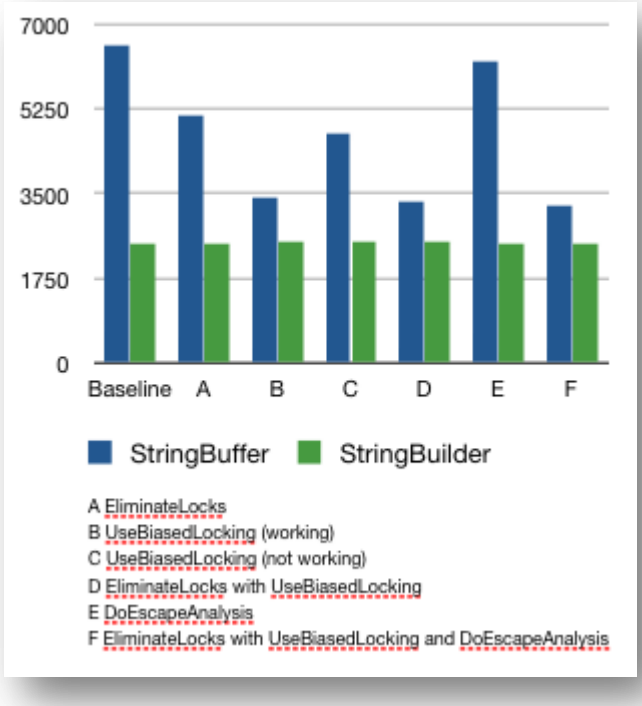


图 3. 基准测试的结果

关于结果的讨论

之所以使用非同步的 StringBuilder，是为了提供一个测量性能的基线。我也想了解一下各种优化是否真的能够影响 StringBuilder 的性能。正如我们所看到的，StringBuilder 的性能可以保持在一个不变的吞吐量水平上。因为这些技术的目标在于锁的优化，因此这个结果符合预期。在性能测试的另一栏中我们也可以看到，使用没有任何优化的同步的 StringBuffer，其运行效率比 StringBuilder 大概要慢三倍。

仔细观察图 3 的结果，我们可以注意到从左到右性能有一定的提高，这可以归功于 EliminateLocks。不过，这些性能的提升比起偏向锁来说又显得有些苍白。事实上，除了 C 列以外，每次运行时如果开启偏向锁最终都会提供大致相同的性能提升。但是，C 列是怎么回事呢？

在处理最初的数据的过程中，我注意到有一项测试在六个测试中要花费格外长的时间。由于结果的异常相当明显，因此基准测试似乎在报告两个完全不同的优化行为。经过一番考虑，我决定同时展示出高值和低值（B 列和 C 列）。由于没有更深入的研究，我只能猜测这

里应用了一种以上的优化（很可能是两种），并且存在一些竞争条件，偏向锁大多时候会取胜，但不非总能取胜。如果另一种优化占优了，那么偏向锁的效果要么被抑制，要么就被延迟了。

这种奇怪的现象是逸出分析导致的。明确了这个基准测试的单线程化的本质后，我期待着逸出分析会消除锁，从而将 StringBuffer 的性能提到了与 StringBuilder 相同的水平。但是很明显，这并没有发生。还有另外一个问题；在我的机器上，每一次运行的时间片分配都不尽相同。更为复杂的是，我的几位同事在他们的机器上运行这些测试，得到的结果更混乱了。在有些时候，这些优化并没有将程序提速那么多。

前期的结论

尽管图 3 列出的结果比我所期望的要少，但确实可以从中看出各种优化能够除去锁产生的大部分开销。但是，我的同事在运行这些测试时产生了不同的结果，这似乎对测试结果的真实性提出了挑战。这个基准测试真的测量锁的开销了么？我们的结论成熟么？或者还有没有其他的情况？在本文的第二部分里，我们将会深入研究这个基准测试，力争回答这些问题。在这个过程中，我们会发现获取结果并不困难，困难的是判断出这些结果是否可以回答前面提出的问题。

```
public class    LockTest {
private static final int MAX = 20000000; // 20 million
public static void main(String[] args) throws InterruptedException {
// warm up the method cache
for (int i = 0; i < MAX; i++) {
        concatBuffer("Josh", "James", "Duke");
        concatBuilder("Josh", "James", "Duke");
}
System.gc();
Thread.sleep(1000);
System.out.println("Starting test");
long start = System.currentTimeMillis();
for (int i = 0; i < MAX; i++) {
        concatBuffer("Josh", "James", "Duke");
}
long bufferCost = System.currentTimeMillis() - start;
```

```
        System.out.println("StringBuffer: " + bufferCost + " ms.");
        System.gc();
        Thread.sleep(1000);
        start = System.currentTimeMillis();
        for (int i = 0; i < MAX; i++) {
                concatBuilder("Josh", "James", "Duke");
        }
        long builderCost = System.currentTimeMillis() - start;
        System.out.println("StringBuilder: " + builderCost + " ms.");
        System.out.println("Thread safety overhead of StringBuffer: "
        + ((bufferCost * 10000 / (builderCost * 100)) - 100) + "%\n");
        }
        public static String concatBuffer(String s1, String s2, String s3) {
                StringBuffer sb = new StringBuffer();
                sb.append(s1);
                sb.append(s2);
                sb.append(s3);
                return sb.toString();
        }
        public static String concatBuilder(String s1, String s2, String s3) {
                StringBuilder sb = new StringBuilder();
                sb.append(s1);
                sb.append(s2);
                sb.append(s3);
                return sb.toString();
        }
        }
```

运行基准测试

我运行这个测试的环境是：32 位的 Windows Vista 笔记本电脑，配有 Intel Core 2 Duo，使用 Java 1.6.0_04。请注意，所有的优化都是在 Server VM 上实现的。但这在我的平台上不是默认的 VM ,它甚至不能在 JRE 中使用 ,只能在 JDK 中使用。为了确保我使用的是 Server VM，我需要在命令行上打开-server 选项。其他的选项包括：

- -XX:+DoEscapeAnalysis, off by default
- -XX:+UseBiasedLocking, on by default
- -XX:+EliminateLocks, on by default

编译源代码，运行下面的命令，可以启动测试：

```
java-server -XX:+DoEscapeAnalysis LockTest
```

在本文的前半部分中，我们通过一个单一线程的基准，比较了同步的 StringBuffer 和非同步的 StringBuilder 之间的性能。从最初的基准测试结果来看，偏向锁提供了最佳的性能，比其他的优化方式更有效。测试的结果似乎表明获取锁是一项昂贵的操作。但是在得出最终的结论之前，我决定先对结果进行检验：我请我的同事们在他们的机器上运行了这个测试。尽管大多数结果都证实了我的测试结果，但是有一些结果却完全不同。在本文的第二部分中，我们将更深入地看一看用于检验测试结果的技术。最后我们将回答现实中的问题：为什么在不同的处理器上的锁开销差异如此巨大？

基准测试中的陷阱

通过一个基准测试，尤其是一个“小规模基准测试”（microbenchmark），来回答这个问题是非常困难的。多半情况下，基准测试会出现一些与你期望测量的完全不同的情景。即使当你要测量影响这个问题的因素时，结果也会被其他的因素所影响。有一点在这个实验开始之初就已经很明确了，即这个基准测试需要由其他人全面地进行审查，这样我才能避免落入报告无效基准测试数据的陷阱中。除了其他人的检查以外，我还使用了一些工具和技术来校验结果，这些我会在下面的几节中谈到。

结果的统计处理

大多数计算机所执行的操作都会在某一固定的时间内完成。就我的经验而言，我发现即使是那些不确定性的操作，在大多数条件下基本上也能在固定的时间内完成。正是根据计算的这种特性，我们可以使用一种工具，它通过测量让我们了解事情何时开始变得不正常了。这样的工具是基于统计的，其测量结果会有些出入。这就是说，即使看到了一些超过正常水平的报告值，我也不会做过多过的解释的。原因是这样的，如果我提供了指令数固定的 CPU，而它并没有在相对固定的时间内完成的话，就说明我的测量受到了一些外部因素的影响。如果测试结果出现了很大的异常，则意味着我必须找到这个外部的影响进而解决它。

尽管这些异常效果会在小规模基准测试中被放大，但它不至于会影响大规模的基准测试。对于大规模的基准测试来说，被测量的目标应用程序的各个方面会彼此产生干扰，这会带来

一些异常。但是异常仍然能够提供一些很有益的信息，可以帮助我们对干扰级别作出判断。在稳定的负荷下，我并不会对个别异常情况感到意外；当然，异常情况不能过多。对于那些比通常结果大一些或小一些的结果，我会观察测试的运行情况，并将它视为一种信号：我的基准测试尚未恰当地隔离或者设置好。这样对相同的测试进行不同的处理，恰恰说明了全面的基准测试与小规模基准测试之间的不同。

最后一点，到此为止仍然不能说明你所测试的就是你所想的。这至多只能说明，对于最终的问题，这个测试是最有可能是正确的。

预热方法的缓存

JIT 会编译你的代码，这也是众多影响基准测试的行为之一。Hotspot 会频繁地检查你的程序，寻找可以应用某些优化的机会。当找到机会后，它会要求 JIT 编译器重新编译问题中的某段代码。此时它会应用一项技术，即当前栈替换（On Stack Replacement，OSR），从而切换到新代码的执行上。执行 OSR 时会对测试产生各种连锁影响，包括要暂停线程的执行。当然，所有这样的活动都会干扰到我们的基准测试。这类干扰会使测试出现偏差。我们手头上有两款工具，可以帮助我们标明代码何时受到 JIT 的影响了。第一个当然是测试中出现的差异，第二个是-XX:-PrintCompilation 标记。幸运的是，如果不是所有的代码在测试的早期就进行 JIT 化处理，那么我们可以将它视为另外一种启动时的异常现象。我们需要做的就是在开始测量前，先不断地运行基准测试，直到所有代码都已经完成了 JIT 化。这个预热的阶段通常被称为“预热方法的缓存”。

大多数 JVM 会同时运行在解释的与本机的模式中。这就是所谓的混合模式执行。随着时间的流逝，Hotspot 和 JIT 会根据收集的信息将解释型代码转化为本机代码。Hotspot 为了决定应该使用哪种优化方案，它会抽样一些调用和分支。一旦某个方法达到了特定的阈值后，它会通知 JIT 生成本机代码。这个阈值可以通过 -XX:CompileThreshold 标记来设定。例如，设定 -XX:CompileThreshold=10000，Hotspot 会在代码被执行 10,000 次后将它编译为本机代码。

堆管理

下一个需要考虑的是垃圾收集，或者更广为人知的名字—堆管理。在任何应用程序执

行的过程中，都会定期地发生很多种内存管理活动。它们包括：重新划分栈空间大小、回收不再被使用的内存、将数据从一处移到另一处等等。所有这些行为都导致 JVM 影响你的应用程序。我们面对的问题是：基准测试中是否需要将内存维护或者垃圾回收的时间包括进来？问题的答案取决于你要解决的问题的种类。在本例中，我只对获取锁的开销感兴趣，也就是说，我必须确保测试中不能包含垃圾回收的时间。这一次，我们又能够通过异常的现象来发现影响测试的因素，一旦出现这种问题，垃圾回收都是一个可能的怀疑对象。明确问题的最佳方式是使用 -verbose:gc 标志，开启 GC 的日志功能。

在这个基准测试中，我做了大量的 String、StringBuffer 和 StringBuilder 操作。在每次运行的过程中大概会创建 4 千万个对象。对于这样一种数量级的对象群来说，垃圾回收毫无疑问会成为一个问题。我使用两项技术来避免。第一，提高堆空间的大小，防止在一个迭代中出现垃圾回收。为此，我利用了如下的命令行：

```
>java -server -XX:+EliminateLocks -XX:+UseBiasedLocking -verbose:gc
-XX:NewSize=1500m    -XX:SurvivorRatio=200000 LockTest
```

然后，加入清单 1 的代码，它为下一次迭代准备好堆空间。

```
System.gc();
Thread.sleep(1000);
```

清单 1. 运行 GC，然后进行短暂的休眠。

休眠的目的在于给垃圾回收器充分的时间，在释放其他线程之后完成工作。有一点需要注意：如果没有 CPU 任何活动，某些处理器会降低时钟频率。因此，尽管 CPU 时钟会自旋等待，但引入睡眠的同时也会引入延迟。如果你的处理器支持这种特性，你可能必须要深入到硬件并且关闭掉“节能”功能才行。

前面使用的标签并不能阻止 GC 的运行。它只表示在每一次测试用例中只运行一次 GC。这一次的暂停非常小，它产生的开销对最终结果的影响微乎其微。对于我们这个测试来说，这已经足够好了。

偏向锁延迟

还有另外一种因素会对测试结果产生重要的影响。尽管大多数优化都会在测试的早期发

生，但是由于某些未知的原因，偏向锁只发生在测试开始后的三到四秒之后。我们又要重述一遍，异常行为再一次成为判断是否存在问题的重要标准了。-XX:+TraceBiasedLocking 标志可以帮助我们追踪这个问题。还可以延长预热时间来克服偏向锁导致的延迟。

Hotspot 提供的其他优化

Hotspot 不会在完成一次优化后就停止对代码的改动。相反，它会不断地寻找更多的机会，提供进一步的优化。对于锁来说，由于很多优化行为违反了 Java 存储模型中描述的规范，所以它们是被禁止的。然而，如果锁已经被 JIT 化了，那么这些限制很快就会消失。在这个单线程化的基准测试中，Hotspot 可以非常安全地将锁省略掉。这样就会为其他的优化行为打开大门；比如方法内联、提取循环不变式以及死代码的清除。

如果仔细思考下面的代码，可以发现 A 和 B 都是不变的，我们应该把它抽取出来放到循环外面，并引入第三个变量，这样可以避免重复的计算，正如清单 3 中所示的那样。通常，这都是程序员的事情。但是 Hotspot 可以识别出循环不变式并把它们抽取到循环体外面。因此，我们可以把代码写得像清单 2 那样，但是它执行时其实更类似于清单 3 的样子。

```
int A = 1;
int B = 2;
int sum = 0;
for (int i =    0; i < someThing; i++) sum += A + B;
```

清单 2 循环中包含不变式

```
int A = 1;
int B = 2;
int sum = 0;
int invariant = A + B;
for (int i =    0; i < someThing; i++) sum += invariant;
```

清单 3 不变式已抽取到循环之外

这些优化真的应该允许么？还是我们应该做一些事情防止它的发生？这个有待商榷。但至少，我们应该知道是否应用了这些优化。我们绝对要避免“死代码消除”这种优化的出现，否则它会彻底扰乱我们的测试！Hotspot 能够识别出我们没有使用 concatBuffer 和 concatBuilder 操作的结果。或者可以说，这些操作没有边界效应。因此没有任何理由执行这

些代码。一旦代码被标识为已“死亡”，JIT 就会除去它。好在我的基准测试迷惑了 Hotspot，因此它并没有识别出这种优化，至少目前还没有。

如果由于锁的存在而抑制了内联，反之没有锁就可能出现内联，那么我们要确保在测试结果中没有包含额外的方法调用。现在可以用到的一种技术是引入一个接口（清单 4）来迷惑 Hotspot。

```
public interfaceConcat {
    String concatBuffer(String s1, String s2, String s3);
    String concatBuilder(String s1, String s2, String s3);
public class LockTest implements Concat {
...}
```

清单 4 使用接口防止方法内联

防止内联的另一种方法是使用命令行选项-XX:-Inline。我已经验证，方法内联并没有给基准测试的报告带来任何不同。

执行栈输出

最后，请看下面的输出结果，它使用了下面的命令行标识。

```
>java -server -XX:+DoEscapeAnalysis -XX:+PrintCompilation -XX:+EliminateLocks
-XX:+UseBiasedLocking -XX:+TraceBiasedLocking LockTest
```

图 1 基准测试的执行栈输出

JVM 默认会启动 12 个线程，包括：主线程、对象引用处理器、Finalize、Attach 监听器等等。上图中第一个灰色段显示的是这些线程的对齐，它们可以使用偏向锁（注意所有地址都以 00 结尾）。你尽管忽略可以忽略它们。接下来的黄色段包含了已编译方法的信息。我们看一下第 5 行和 12 行，能够发现它们都标记了一个额外的“s”。表 1 的信息告诉我们这些方法都是同步的。包含了“%”的各行已经使用了 OSR。红色的行是偏向锁被激活的地方。最底下的蓝绿色框是基准测试开始计时的地方。从记录基准测试开始时间的输出中可以看到，所有编译都已经发生了。这说明前期的预热阶段足够长了。如果你想了解日志输出规范的更多细节，可以参考这个页面和这篇文章。

b	Blocking compiler (true only for client compiler)
*	Generating a native wrapper
%	On stack replacement
!	Method has exception handlers
s	Synchronized method

表 1 编译示例码

单核系统下的结果

尽管我的多数同事都在使用 Intel Core 2 Duo 处理器 ,但还是有一小部分人使用陈旧的单核机器。在这些陈旧的机器上，StringBuffer 基准测试的结果和 StringBuilder 实现的结果几乎相同。由于产生这种不同可能是多种因素使然，因此我需要另外一个测试，尝试忽略尽可能多的可能性。最好的选择是，在 BIOS 中关闭 Core 2 Duo 中的一个核，然后重新运行基准测试。运行的结果如图 2 所示。

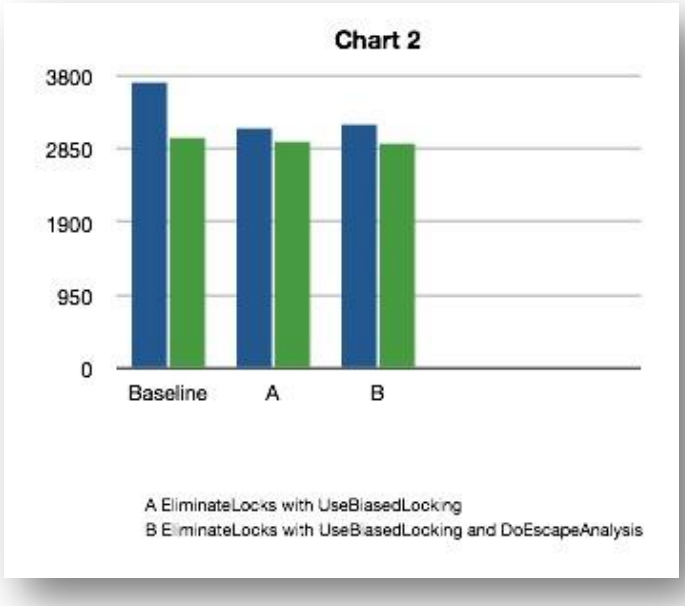


图 2 单核系统的性能

在多核环境下运行的时候，关闭了三种优化行为后获得了一个基准值。这次，StringBuilder 又保持了平稳的吞吐量。更有趣的是，尽管 StringBuffer 比 StringBuilder 要稍慢，但是在多核平台下，StringBuffer 的性能更接近于 StringBuilder。从这个测试开始我们将一步步勾勒出基准测试的真实面目。

在多核的世界中，线程间共享数据的现实呈现出一种全新的面貌。所有现代的 CPU 必须使用本地存储的缓存，将获取指令和数据的延迟降到最低。当我们使用锁的时候，会导致一次存储关卡（Barrier）被插入到执行路径中。存储关卡像一个信号，它通知 CPU 此时必须和其他所有的 CPU 进行协调，以此获得最新的数值。为了完成这个任务，CPU 之间将要彼此通讯，从而导致每个处理器暂定当前正在运行的应用程序线程。这个过程要花多少时间已经成了 CPU 存储模型的指标之一。越是保守的存储模型，越是线程安全的，但是它们在协调各个处理器核的时候也要花费更多的时间。在 Core 2 Duo 上，第二个核将固定的运行基准从 3731ms 提高到了 6574ms，或者说增加了 176%。很明显，Hotspot 所提供的任何帮助都能明显改进我们的应用程序的总体性能。

逸出分析真的起作用了么？

现在，还有一种优化很明显会起作用，但是我们还没有考虑，它就是锁省略。锁省略是最近才实现的技术，而且它依赖于逸出分析，后者是一种 Profiling 技术，其自身也是刚刚才实现的。为了稳妥一些，各公司和组织都宣称这些技术只有在有限的几种情况下才起作用。比如，在一个简单的循环里，对一个局部变量执行递增，且该操作被包含在一个同步块内，由一个局部的锁保护着。这种情况下逸出分析是起作用的[http://blog.nirav.name/2007_02_01_archive.html]。同时它在 Mont Carlo 的 Scimark2 基准测试中可以工作（参见[http://math.nist.gov/scimark2/index.html]）。

将逸出分析包含在测试中

那么，为什么逸出分析可以用于上述的情况中，却不能用于我们的基准测试中？我曾经尝试过将 StringBuffer 和 StringBuilder 的部分方法进行内联。我也修改过代码，希望可以强制逸出分析运行。我想看到锁最终被忽略，而性能可以获得大幅提升。老实说，处理这个基准测试的过程既困惑，又让人倍感挫折。我必须无数次地在编辑器中使用 ctrl-z，以便恢复

到前面一个我认为逸出分析应该起作用的版本，但是却不知由于什么原因，逸出分析却突然不起作用了。有时，锁省略却又会莫名其妙地出现。

最后，我认识到激活锁省略似乎和被锁对象的数据大小有关系。你运行清单 2 的代码就会看到这一点。正如你所看到的，无论运行多少次，结果都毫无区别，这说明 DoEscapeAnalysi 没有产生影响。

```
>java -server -XX:-DoEscapeAnalysis EATest
thread unsafe: 941 ms.
thread safe: 1960 ms.
Thread safety overhead: 208%
>java -server -XX:+DoEscapeAnalysis EATest
thread unsafe: 941 ms.
thread safe: 1966 ms.
Thread safety overhead: 208%
```

在下面的两次运行中，我移除了 ThreadSafeObject 类中一个没有被用过的域。如你所见，当开启了逸出分析，所有性能有了很大的提高。

```
>java -server -XX:-DoEscapeAnalysis EATest
thread unsafe: 934 ms.
thread safe: 1962 ms.
Thread safety overhead: 210%
>java -server -XX:+DoEscapeAnalysis EATest
thread unsafe: 933 ms.
thread safe: 1119 ms.
Thread safety overhead: 119%
```

逸出分析的数目在 Windows 和 Linux 上都能看到。然而在 Mac OS X 上，即使有额外未被使用的变量也不会有任何影响，任何版本的基准测试的结果都是 120%。这让我不由地相信在 Mac OS X 上有效性的范围比其他系统更广泛。我猜测这是由于它的实现比较保守，根据不同条件（比如锁对象数据大小和其他 OS 特定的特性）及早地关掉了它。

结论

当我刚开始这个实验，解释应用各种锁优化的 Hotspot 的有效性的时候，我估计它将花费我几个小时的时间，最终这会丰富我的 blog 的内容。但是就像其他的基准测试一样，对

结果进行验证和解释的过程最终耗费了几周的时间。同样，我也与很多专家进行合作，他们分别花费了大量时间检查结果，并发表他们的见解。即使在这些工作完成以后，仍然很难说哪些优化起作用了，而哪些没有起作用。尽管这篇文章引述了一组测试结果，但它们是特定我的硬件和系统的。大家可以考虑是否能在自己的系统上看到相同类型的测试结果。另外，我最初认为这不过是个小规模基准测试，但是后来它逐渐既要满足我，也要满足所有审核代码的人，而且去掉了 Hotspot 不必要的优化。总之，这个实验的复杂度远远地超出了我的预期。

如果你需要在多核机器上运行多线程的应用程序，并且关心性能，那么很明显，你需要不断地更新所使用的 JDK 到最新版本。很多（但不是全部）前面的版本的优化都可以在最新的版本中获得兼容。你必须保证所有的线程优化都是激活的。在 JDK 6.0 中，它们默认是激活的。但是在 JDK 5.0 中，你需要在命令行中显式地设置它们。如果你在多核机器上运行单线程的应用程序，就要禁用除第一个核以外所有核的优化，这样会使应用程序运行得更快。

在更低级的层面上，单核系统上锁的开销远远低于双核处理器。不同核之间的协调，比如存储关卡语义，通过关掉一个核运行的测试结果看，很明显会带来系统开销。我们的确需要线程优化，以此降低这一开销。幸运的是，锁粗化和（尤其是）偏向锁对于基准测试的性能确实有明显的影响。我也希望逸出分析与锁省略一起更能够做到更好，产生更多的影响。这项技术会起作用，可只是在很少的情况下。客观地说，逸出分析仍然还处于它的初级阶段，还需要大量的时间才能变得成熟。

最后的结论是，最权威的基准测试是让你的应用程序运行在自己的系统上。当你的多线程应用的性能没有符合你的期望的时候，这篇文章能够为你提供了一些思考问题的启示。而这就是此文最大的价值所在。

关于 Jeroen Borgers

Jeroen Borger 是 Xebia 的资深咨询师。Xebia 是一家国际 IT 咨询与项目组织公司，专注于企业级 Java 和敏捷开发。Jeroen 帮助他的客户攻克企业级 Java 系统的性能问题，他同时还是 Java 性能调试课程的讲师。他在从 1996 年开始就可以在不同的 Java 项目中工作，担任过开发者、架构师、团队 lead、质量负责人、顾问、审核员、性能测试和调试员。他从 2005

年开始专注于性能问题。

鸣谢

没有其他人的鼎力相助，是不会有这篇文章的。特别感谢下面的朋友：

Dr. Cliff Click ,原 Sun 公司的 Server VM 主要架构师 ,现工作在 Azul System ;他帮我分析，并提供了很多宝贵的资源。

Kirk Pepperdine，性能问题的权威，帮助我编辑文章。

David Dagastine，Sun JVM 性能组的 lead，他为我解释了很多问题，并把我引领到正确的方向。

我的很多 Xebia 的同事帮我进行了基准测试。

原文链接：http://www.infoq.com/cn/articles/java-threading-optimizations-p1

http://www.infoq.com/cn/articles/java-threading-optimizations-p2

相关内容：

- Date 和 Time API：第三轮投票
- 提供给每个人的行为驱动开发
- IcedTea：首个 100%兼容、开源的 Java
- JSR 308：Java 语言复杂度在恣意增长？
- 讨论 5 种跟踪 Java 执行的方法

[推荐文章]

探求真正的 SOA

作者 Alex Maclinovsky **译者** 徐涵

推荐理由：业界从来不缺造词机器，在如今各类缩写满天飞的时代，相信大家也有被搞晕的时候。以微软为例，从最早的ActiveX、COM、COM+，直到如今的.Net，谁又能真正说明这些词汇真正的内涵是什么？当下，SOA 真热，关于 SOA 的各类资料层出不穷。而这篇文章的特点是，作者以其自身经历说明他对 SOA 的理解过程，进而总结出自己对于 SOA 乃至 SOA 治理的认识。或许，从其行文之间，读者也会找到自己的影子。

从第一次读到罗斯尼·安（J.H. Rosny Aine）的《求火（Quest for Fire）》开始，我就一直着迷于这种“探求”风格的作品。在我看来，一个探求类故事的精髓，就是一组形形色色的人，他们开始一段漫长而艰险的历程，试图寻找难以发现的目标，最终却找到了别的东西，而这个意外的收获通常比原来要找的东西更有价值、更重要。无论是《绿野仙踪》还是《指环王》，我总无法抗拒被这类故事深深感染。

我认为把我关于 SOA 的看法的演变过程比喻为“探求”是相当贴切的，而且“探求”也解释了我对治理的愿景与业界流行看法不同的原因。在 2004 年秋，我在一家主要的 EAI/SOA 平台厂商工作。那时 SOA 很热，于是依据市场需求，我们的产品被定位为 SOA。不用说，我们的竞争对手们也都声称自己的产品是 SOA。然而，随着我们完成越来越多的企业级实现，我们发现，很明显市场讯息不完全对。如果你采用我们的产品、采取我们的方法、按照我们的指导方针与最佳实践（即便采用我们的专业服务）去实施的话，最终得到的将不是 SOA，至少不是原本预料的 SOA。没错，它能完成一定的功能并满足最初的特定目标，但至于履行已展望多年的 SOA 最初承诺，它就不行了。SOA 承诺将提供多用途的、粗粒度的、

松耦合的、可动态发现的、易于组合的、易于重用的服务，正是这种承诺使得 SOA 的概念如此受到业务与 IT 部门的青睐，从而令 SOA 成为 CIO 们的战略计划、RFI/RFP 及产品特性列表中不可或缺的一项。我不是在批评我们的产品——当时别的厂商能给自己的产品贴上 SOA 的标签，我们当然也能。问题出在 SOA 定义自身。当这个概念流行起来时，大家都在竞相“实现”它。于是，厂商和 IT 机构们开始按照它们自己所能支持的方案，悄悄地改动 SOA 的定义。结果，人人都实现了 SOA，但几乎没人能从中受益。

基于这一背景，我们团队提出了一个经验性的问题：如何才能构建一个大家都认为符合 SOA 特征的平台呢？我们写了一份长达 40 页的白皮书来回答这一问题，并阐述了我们对 SOA 的看法，我们认为企业级 SOA 服务应当具备以下核心特征与支撑设施：

1. 服务的自包含性与模块性：包括模块可分解性（modular decomposability）、模块可组合性（modular composability）、模块可理解性（modular understandability）、模块化连续性（modular continuity）以及模块保护性（modular protection）；
2. 服务、消费者、基础设施、工具及遗留应用间的互操作性；
3. 消费者与提供者间的松耦合性；
4. 支持编配（Orchestration）及服务的可组合性；
5. 服务的可发现性，以及消费者与提供者间的动态绑定；
6. 服务、消费者及基础设施组件的位置透明性；
7. 无处存在的安全性：通过解决认证、授权、完整性、保密性、可问责性、身份识别及策略等问题中的一些或全部，在设计上确保 SOA 环境（包括服务、消费者、代理、合成应用、基础设施等）中的信任；
8. 支持可续航性（Sustainability）和自恢复性（Self Healing），包括监测、预测和缓解不期而至的意外行为和状况的能力，以及按需获取与释放资源的能力；
9. 服务版本化：通过允许同时存在不同版本的服务实现和消费者，支持 SOA 环境的演化；

10. 服务租用：显式定义并维护服务消费者与提供者之间的关系，并消除可能存在不利的相互假设（mutual assumptions）；
11. 网络可寻址的服务接口：具有灵活的调用机制，支持多种传输、协议及同步选项，以营造一个有活力的、多用途的 SOA 环境；
12. 粗粒度的服务接口：可以利用它们轻松地支持许多种服务消费环境；
13. 服务使用计量（实时的和历史的）；
14. 服务及其实现的监控、管理与控制；
15. 支持相关性与根源分析的异常与警报处理；
16. 有效的服务虚拟化，确保服务接口与服务实现的彻底分离；
17. 公开的、可核实的和可实施的服务质量（QOS），包括性能、可靠性、可用率、制度性、可访问性、完整性及安全性等等；

以上列出的这些，为我们评估 SOA 产品及实现的成熟度与完整性提供了一个很好的尺度。当我们用它来评价我们的旗舰产品时，得分仅为 40±5%（具体得分取决于你想给予自己的代码多少宽容）。于是，不用说，我们开始为争取“剩下那 60%”而努力。直到 2005 年秋 1.0 版发布之前，我们都是如此称呼我们所做的工作的。

正当我们要开始努力时，我们发现上述特征列表里漏了几项。不过，“SOA 的 19 项核心特征”听起来不如“17 项核心特征”好听，于是我们称之为“17+项核心特征”——我们正是这样偶然得到基于方面的 SOA（Aspect-Based SOA）基础设施这个想法的。此想法是基于这样一个前提，即我们无法列举出 SOA 服务——作为适当的企业计算平台——所应具备的全部特征，是有其根本原因。这个原因就是：这些特征是基于实际的业务、制度等需求的，而这些需求是因行业、地理、时间及各个客户的具体情况而变的。市场所需要的，是一个可以实现和支持任意项服务特征的平台，而且要允许用户根据当前的业务需要（或甚至自由地）对这些服务特征进行增加、修改和删除。

这个相当荒唐的想法居然得到了我们第一个客户的首肯。这个来自部队的客户，即使按

照军方标准衡量也算相当偏执的了，他们拿出了一份很长的关于安全性需求的列表，其中大部分我们都能做到（要么直接支持，要么可以推给他们自己的安全基础设施）。但是有一项需求让我相当吃惊：该客户想防止一个写得很差的服务意外地把机密信息泄露给合法的、通过正确认证且经过适当授权的、但还尚未得到充分批准的用户。我的原始反应是：雇用优秀的开发人员，开展严格的质量保证（QA）活动，那么就没问题了！然后我意识到，在这个客户看来，解决意外泄露，跟受到别的行业重视的其他服务特征同样重要。于是，我在一小时之内设计好了一个审查方面（Censorship Aspect），它检查所有发给客户的服务响应，判断消息或各个片断的分类级别，将之与客户概要（在来访认证的过程中创建起来）中的许可级别进行对比，然后进行必要的纠正活动：或者隐藏部分信息，或者完全拦截。

其间随着 1.0 版的发布，市场部需要一个名称来称呼它（由于某些莫名其妙的原因，他们不再热衷于围绕“剩下那 60%”而努力了）。他们先是想到了 SOAi 和 SOAf（分别代表基础设施与基础），之后又有人提出了治理（Governance）（也许是因为当时这个词正逐渐时髦起来）。当时我还不熟悉这个词，所以在 Google 里搜索了一下，但怎么也无法理解它跟我们所开发的技术有什么关系。那时我的主要忧虑是：明确的定义没几个，含糊的定义却一大堆。

大部分厂商只是按照自己的能力范围来定义 SOA 治理。Systinet 公司当时推行这样的观点，即 SOA 治理是一个用于 WSDL 的记录系统，它们提供了支持分类层次、发布工作流及订阅/通知机制的制品（artifacts）。AmberPoint 将 SOA 治理看成一种自动发现依赖关系及“流氓服务”的机制，以及一种轻量级安全、监控和端到端错误分析的机制。设备厂商们（appliance vendors）将治理看成 Web 服务安全以及他们所拥有的媒介功能。所谓的专家们不断地提出朦胧而模糊的定义，比如下面这个令 Pythia 引以为豪的 OASIS SOA 参考模型里的定义：

> SOA 治理：通过结构化的关系、适合本机构的步骤与策略、以及分布式能力（可能处于不同控制域之下）的运用，来确保结果符合可测量的前提与预期的技术与原则。

经过一番思索之后，我觉得我们对 SOA 的理解就像是盲人摸象，都不够全面，于是我开始试图探求一个全面的定义。在本文剩余部分，我将给出 SOA 的定义，然后将 SOA 治理

的愿景描述为该 SOA 的标志特征，另外我们也会通过一个例子来举例说明。

既然整个 SOA 问题域都充斥着模糊的术语，那么我们就先提出一些关键定义吧。首先，我们要看要治理的是什么。答案是显而易见的：当然是服务！然而，这一术语已被用于描述太多不同事物了，其中许多都跟 SOA 关系不大，所以我觉得有必要澄清这个概念——我通常的做法是给它增添一些限定。

> 一个企业服务就是一个自包含的组件，它提供业务功能，并具有一组可扩展的非功能性的、基于策略的特性（比如安全性、行业/客户定义的服务策略、管理、监控，生命周期管理等），它通过一个良好定义的、标准的、公开的接口来响应请求。

另一种描述企业服务的方式是：一个名称为 CEO 或业务总管所熟悉、而且其具体功能受到后者关注的服务。按照这一定义， invoiceCustomer、dischargePatient 和 launchICBM 等都是企业服务，而 invokeBAPI、 saveLogRecord 和 generateToken 等则不是。参照该定义，SOA 本身可被描述为：

> 一种提供并促使企业 IT 朝着企业服务环境的方向演化的架构风格。

最后，SOA 治理就成为：

> 通过流程、实践与工具相结合，推动企业服务的生命周期，并提供方法来创建、传达、贯彻和管理目前对公司很重要的、关于非功能性服务特征的公司策略。

按照以上定义，SOA 治理就是通过实现跨 SOA 参与者控制域边界的合理重用、把企业服务（Enterprise Services）从数字制品转变为现实业务制品的活动。为便于理解，我们举一个例子。

设想有一家名为热狗有限公司的跨国公司，它们有面包研究部和香肠开发部两个业务部门。这两个业务部门都有独立的管理团队、盈亏责任、IT 部门及预算，只是在同一公司实体下运作。

某个时候，面包研究部的领导层决定实现一个新的应用来执行他们的核心业务功能。他们为实现这一任务花费了一百万美元。为了整个公司的利益，他们另外花了五万美元，以可

重用服务的形式将新开发的功能包装并发布出来。

次年，香肠开发部的管理团队觉得他们需要构建一个新的应用来执行他们的核心业务，而面包研发部六个月前开发、包装、部署并发布的功能刚好可以满足他们的需要。[跟书本里的 SOA 业务案例颇为相似，是吧？]香肠开发部的 IT 部门现在有两个选择，要么重用面包研发部的服务，要么自己重新花一百万美元开发一套。我敢保证，如果下层技术采用的是 CORBA 或无治理的（按照前面的定义）SOA，那么对香肠开发部来说，唯一合理的做法是花钱实现处于自己控制域下的功能。这样做的理由是：原来的服务是由面包研究部实现并发布的，所以尽管那些服务在功能上 100%符合需求，但对于以下这些决定“那些服务能否为香肠开发部业务环境所用”至关重要的因素，香肠开发部无法得知：

- 预期的可用率、可靠性、平均故障间隔时间（MTBF）及修复时间怎样？
- 现有的安全性和可问责性是什么样的？
- 那些服务实现已经通过审核并经证实符合相关条例（如美国的 SOX，欧洲关于隐私的规定以及健康方面的 HIPAA 等）了吗？
- 支持什么样的性能水平？能够接受什么样的负载水平？
- 那些服务是永久性的吗？会不会随日期、星期或季节等有所变化？
- 对服务使用是如何进行计量和记账的？

即便香肠开发部的管理团队从面包研究部的同事那里得到了所有这些信息，仍存在一个至关重要的问题：我们怎么知道这些信息会不会在下周（也许是下个月或下一年）发生改变呢？

当我们的思考过程结束，并最终形成以上对 SOA 治理的看法时，有几件事变得明了了（至少对我们是这样）：

1. 与爱因斯坦相对论扩展牛顿经典力学、并形成后者所不适用的范围相似，这个对 SOA 治理的统一观点并非否定前面提到的、以及我在《SOA governance – the perspectives》这篇文章里详细描述过的那些零碎的定义与方法，相反，它而是构筑

在后者基础之上的。

2. 若正确实现的话，这种 SOA 治理观点将为你带来一个能够符合前面提到的全部“17+项企业 SOA 核心特征”的方案。

3. 这样一种方案，能够把有缺陷的 SOA 平台与实现改造为真正的面向服务的架构（SOA），从而实现 SOA 的最初设想，并兑现所有承诺的优点。

最后，这一治理平台的构建其实比我们想象的要简单。而且，正如“探求”风格的特点，我们在这一过程中发现了许多预料外的、有价值的东西，比如委托治理（Delegated Governance）、分析时治理（Analysis-time Governance）、改建项目友好的治理（Brownfield-friendly Governance）和统一治理信息模型（unified Governance Information Model）等，这些为我们后续的文章提供了不错的主题。

原文链接：http://www.infoq.com/cn/articles/quest-for-true-soa

相关内容：

- SOA 治理的业务流程
- SOA 案例研究竞赛结果揭秘 SOA 成功的主要因素
- 文章：用消费者驱动的契约进行面向服务开发
- 文章：SOA 治理——真的需要还是在浪费时间？
- 克服 SOA 实施过程中的障碍

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]

运用 Ruby 纤程进行异步 I/O：NeverBlock 和 Revactor

作者 Werner Schuster **译者** 李明（nasi）

推荐理由：Ruby 的高并发和并行处理一向是软肋。在 Ruby 1.9 中，开发者们迎来了纤程（Fiber）模型，可以更有效地解决并行处理的问题。而异步 I/O 则可以极大地提高系统的性能，这从 NeverBlock 在 ActiveRecord 上所带来的改进就可见一斑。如今，开发者们可以借用这些利器，来创建真正高性能、高并发的 Ruby 应用。

纤程(Fibers)慢慢地进入了 Ruby 开发者们的视线中 ,将作为新的并发原语而广为使用。

通过对纤程和非阻塞（或者说异步）I/O 这两种方法的组合，可以解决用户空间线程问题或者 Ruby 1.9 的巨型解释器锁（Giant Interpreter Lock，简称 GIL）的问题，这个问题使得 Ruby 语言线程每次只能有一个是激活的。

最大的问题要数和使用那些阻塞的系统调用了，尤其是 I/O 方面的。阻塞的系统调用，就拿读取来说，直到数据准备好的时候才会返回。在用户态的线程系统中，这意味着这个进程中的所有线程也都会阻塞。解决方法是将 I/O 系统与阻塞 I/O 的方式进行解耦。有一种方法是在引发一个阻塞的系统调用以前捕捉它，以一种非阻塞的方式来处理这个 I/O 请求，然后挂起这个纤程，给另外的纤程运行的机会。一旦系统收到这个 I/O 请求的回应的话，纤程会再度被调度。

纤程提供了一个好用的并发工具来实现这个概念 ,而不会让使用者去忍受些什么不愉快。目前，两个程序库都着重于使用纤程实现解决方案。比较新的解决方案是 NeverBlock 程序

库（Github 代码库包括 NeverBlock、NeverBlock PG、支持 ActiveRecord 的 NeverBlock PG 适配器，以及支持异步操作的 MySQLPlus MySQL 适配器）。更加通用的解决方案是建立在诸如 Actor 和进程通信等 Erlang 的思想之上的，来自 Tone Arcieri 的 Revactor 程序库。我们对来自 NeverBlock 项目的 Mohammad A. Ali 和来自 Revactor 项目的 Tone Arcieri 进行了访谈。

NeverBlock

InfoQ: 准确点儿说，NeverBlock 做了些什么？依我看，它提供了一个连接池把传入代码包装在纤程当中，等到连接可用的时候再恢复纤程。NeverBlock 的主要贡献是什么？

Mohammad A. Ali 实现池机制对于 NeverBlock 其他部分的正常工作是必须的。最初的想法是，我们需要为应用程序提供即时的 IO 并发，例如 Rails、Merb 或者 Ramaze 等不支持完全线程安全的应用。

想要达到这个目标的话，我们需要一个 web 服务器来包装来自纤程的请求和 IO 程序库，纤程来自于 NeverBlock 的纤程池，而 IO 程序库（不仅仅是 DB 而是全部的 IO 操作）则是那些与纤程有关并利用其进行暂停和恢复请求的程序库。所有的一切都要使用一个诸如 EventMachine 或者 Rev 这样的事件循环来进行搭配。

InfoQ: NeverBlock::PG 是基于 NeverBlock 构建的，而且 PostgreSQL 驱动已经支持了非阻塞 I/O 且立等可用。如果这样的话，再引入 NeverblockPG 做什么呢？如果想添加另外一个事件驱动的话，比如说 Mysql，还没有非阻塞 I/O 的支持，那么 NeverBlock 会提供什么帮助呢？

Mohammad A. Ali NeverBlock::PG 驱动可以让非阻塞操作透明化，你只需要使从纤程池中 spawn 一个纤程出来即可。这样的话，你就可以这么写程序：

```
pool.spawn do
  res1 = db.exec(query1)
  res2 = db.exec(query2_that_depends_on_query1)
end
```

一个普通的非阻塞实现会比这更加复杂。多亏了纤程，我们可以写出这种看似明明是阻塞的代码，却能够以非阻塞的行为运行。

对于 NeverBlock::PG 驱动，我们再更深入一点。我们刚刚发布了一个全新的

activerecord-neverblock- postgresql 适配器，将非阻塞 IO 带到了 ActiveRecord 中。我觉得很容易猜到 NeverBlock 的下一个目标是什么了。

最有意思的是，当你将 NeverBlock 应用于全栈应用的时候，它能以一种几乎透明的方式来实现。

NeverBlock 不能让一个阻塞的驱动变得非阻塞。它能让一个非阻塞的驱动的操作看起来更像是阻塞的方式，而不会牺牲非阻塞的特性。这既是说，我们未来的努力方向是类似于 Asymy 的（事件化 mysql 驱动）。

InfoQ: 你的程序库需要纤程（Fibers）的支持，而纤程只有 Ruby 1.9 才支持。你认为这对用户来说是一个问题吗（考虑到 Ruby 1.9 依然变化频繁而且并没有太多人去尝试）？你的程序库的特性会不会对用户去尝试 Ruby 1.9 起到一个刺激作作用呢？

Mohammad A. Ali 我相信纤程是 Ruby 1.9 中最棒的事情之一。同样的功能可以通过 Ruby 1.8 的 continuation 实现，但是性能上要差得多。另外稳定的 1.9 目前离发布也为期不远了，我觉得我们真的应该前行了。我相信 NeverBlock、Revactor 或者类似的东西提供的优点会帮助大家心甘情愿地转换过去的。

InfoQ: 看了看 NeverBlock 开放的源代码：你对 Fiber 使用了开放类并加入了一些方法。这么做的目的是什么？

Mohammad A. Ali 纤程对于存储纤程本地变量方面的功能是缺失的，然而线程可以做到。我需要这些功能来替代事务的当前 ActiveRecord 实现并使得纤程可查。我还需要它们来做到非阻塞操作的可选性，还有线程的上下文等等。

InfoQ: 你听说过 Revactor 吗？

Mohammad A. Ali 听过，我还搞过一点 Revactor。实际上我用 Rev 做为 NeverBlock 除了 EventMachine 以外的第二后端（仅仅是实验性质的）。但是 NeverBlock 和 Revactor 的目的不同。Revactor 为 Ruby 引入了一个新的并发模型，而 NeverBlock 目的是以最小的变化把并发引入当前的 Ruby 程序中。

编辑注：在本次采访完成之时，第一个异步 MySQL 驱动的尝试——MySQLPlus 已经可用了。

Revactor

InfoQ: Revactor 的当前状态是什么？

Tony Arcieri: 在无数的项目当中有些被忽视，然而在很多商业应用上已经获得了成功。

InfoQ: Revactor 新版本的计划是什么？

Tony Arcieri: 最近 Aman Gupta 发布了一个“poor man's Fibers”的 Ruby 1.8 的实现。这样的话将 Revactor 移植到 1.8 也是可以的了，然而性能上可能会很差。

目前，Revactor 还只是支持将全部的 Actors 放在同一个 Ruby 线程中。而更好的做法是可以在不同的线程中运行的 Actors 之间互相传递消息，但是目前还做不到。我已经和一些有兴趣实现它的人们聊过，希望很快就可以将这种方式引入 Revactor。

InfoQ: 用户需要直接对 Revactor 的 actors 进行编程吗？还是可以使用 RevActor 为其他的程序库实现后端，这样可以（对开发者）透明化？

Tony Arcieri: Revactor 和 Rubinius 中的 Actor 实现基本上兼容，但是目前 Rubinius 中的 Actors 还没有一个简单的方法可以像 Erlang 的 gen_tcp 那样进行网络编程。这也就是说，开发者们想使用 Actors 编写网络应用的话，可以先从 Revactor 开始，将来再移植到 Rubinius 上。

InfoQ: 你是如何调度阻塞 I/O 请求的？你使用内核线程来运行 I/O 请求吗？

Tony Arcieri: 当所有已发现的 Actor 消息都处理了、而且没有其他 Actors 是运行态的时候，Revactor 使用一个我编写的、叫做 Rev（和 EventMachine 类似）的事件程序库来监控 I/O 事件。Rev 使用 Ruby 1.9 中的 rb_thread_blocking_region()函数来做到阻塞监控 I/O 读取的系统调用，因此不需要再使用独立的内核线程。 Revactor 动态地扩展了 Ruby 的 TCPSocket 类（Revactor::TCP），这样能做到看起来像阻塞的调用方式，但是实际上则是传回给 Actor 调度器。对现存的库进行 Moneky Patch（动态打补丁）来做到 Revactor::TCP 替换 Ruby Sockets

是很容易做到的。比如说，Revactor 就发布了一个小补丁，来实现在 Mongrel 中使用 Actors 而不是线程来并发。

InfoQ: 你如何处理 I/O 请求序列？是批量请求处理吗？

Tony Arcieri: 从表面上看，调用方式是“阻塞”的。当一个 I/O 请求处理结束后，当前的 Actor 就会休眠，并在下一次 I/O 竞争发生的时候被重新调度。这使得那些依赖于阻塞方式接口的程序库可以很方便的构建在 Revactor 之上。比如说 ActiveRecord 或者 DataMapper 之类的（目前还没有运行在 Revactor 之上）。

InfoQ: Rubinius 上 Actors 的状态是什么?

Tony Arcieri: 它做了所有 Revactor 能做的事，并且宣布将支持 TCP 套接字“active 模式”。这意味着在从 TCP 套接字中读取输入的时候，和以前不同的是，传入的数据是通过标准内部 Actor 消息异步送达到指定的 Actor 的。这使得 Actor 可以并行处理 I/O 和内部 Actor 消息。Rubinius 虚拟机目前正在使用 C++重写，在重写之后希望可以包含实现“active 模式”消息传递的全部特性。当它一旦可用的时候，我就会去尝试一下。

InfoQ: 有计划让 Revactor 支持 Rubinius 吗？

Tony Arcieri: 没有，Revactor 大量的利用到了 YARV 的特性。Rubinius 有一个很棒的并发模型和以 Task 和 Channel 的形式的 I/O 支持，而且 Rubinius 现有的 Actor 实现让这两者都非常的有效率。Revactor 和 Rubinius Actors 在 duck type 方面很大程度上是相似的，所以编写两者都兼容的程序并不是一件很头疼的事情。

InfoQ: 在 Rubinius 上运行 Actors 或者 Revactor 有什么优势呢（超过 Ruby 1.9）？

Tony Arcieri: 现在的话还是 Ruby 1.9 平台更好一些，对于现有的库有着更好的兼容性。Rubinius 正在开发当中，而且目前在重写。在将来，Rubinius 上会有很多优势能超越 Ruby 1.9，比如用于并发和 I/O 的 Task/Channel 抽象就会比 Ruby 1.9 中现存的更加清晰。在 1.8 和 1.9 上用于 I/O 的解决方案一直都是运行一个诸如 EventMachine 或者 Rev 的事件框架，与 Ruby 内建的 I/O 并行工作，而 Rubinius 在 I/O 方面一开始就已经占优势了。

InfoQ：你知道使用 Revactor 的项目吗？

Tony Arcieri：我已经听说了很多人在一些内部项目中使用它了，主要是用于并发 HTTP 客户端。我还不知道任何已经发布的项目用到了它。

InfoQ：依赖 Ruby 1.9 的话，会不会对 Revactor 或者使用纤程的库的推广造成阻碍？

Tony Arcieri：我当然可以想象的到这种情况。我也被 Ruby 1.9 的 bug 折磨过，所以能想象绝大部分的试用者都会小心翼翼的使用的。我已经了解到最近很多利用纤程来实现事件框架的项目冒了出来，比如 Ry Dahl 的 Flow web server。

原文链接：http://www.infoq.com/cn/articles/fibers-neverblock-revactor

相关内容：

- 文章：探索 RailsKits——停止编写重复冗余的代码
- RubyEncoder：Ruby 混淆器和代码保护
- Ruby in Steel：包含 Visual Studio 的免费 IDE
- Exceptional 和 Hoptoad 扩展 Rails 异常监控
- Kenai: 构建于 JRuby on Rails 上的项目宿主服务

[新品推荐]

Ruby VM 近况：Ruby 1.9.1 第一预览版发布，Rubinius 向 C++ VM 迁移

作者 Mark Levison 译者 郑柯

Ruby 1.9.1 第一预览版已经发布，标记着语言特性和其他属性的冻结，而 1.9.1 预定的最终发布时间是 2009 年 1 月底。按照预定计划，Ruby 1.9.1 是 1.9.x 的第一个稳定发布版。另外：Rubinius 的 C++分支已经升级为主干分支。

原文链接：http://www.infoq.com/cn/news/2008/10/ruby-191preview1-rubinius-cpp

解决云计算安全问题的虚拟专用网络——VPN-Cubed

作者 Jean-Jacques Dubray **译者** 霍泰稳

CohesiveFT 的 CTOPatrick Kerpan 说安全是阻碍企业云计算实施的门槛因素。他的公司最近发布了首款针对云的 VPN（虚拟专用网络），使得企业用户可以保护三种拓扑架构：云、云 -云（Cloud-to-Cloud）和企业-云（Enterprise-to-Cloud）。

原文链接：http://www.infoq.com/cn/news/2008/10/cloud-vpn

微软正式推出云服务平台——Windows Azure

作者 Jonathan Allen **译者** 霍泰稳

在上个月的 PDC 大会上，微软正式对外推出一款新产品：Windows Azure。Azure 是一

开“云服务操作系统”。具体来说，它是一个提供完整的基于云的开发、宿主和管理服务的独立平台。Azure 的正式推出表示微软彻底转身，开始拥抱云计算。

原文链接：http://www.infoq.com/cn/news/2008/10/azure

支持 GlassFish、Spring 2.5 和分布式垃圾收集——Terracotta 2.7 正式发布

作者 Srini Penchikala **译者** 崔康

Terracotta 是一款开源 Java 集群框架，它的最新版提供了对 GlassFish、Spring 2.5 的支持及其它一些新功能，比如自动高可用模式、改进的分布式垃圾回收性能和可视化、集群范围的运行时数据统计。Terracotta 开发团队在上周发布了该最新版本——Terracotta 2.7。

原文链接：http://www.infoq.com/cn/news/2008/10/terracotta-2.7-release

Merb 1.0 即将发布，RC1 现已可用

作者 Mirko Stocker **译者** 杨晨

Merb 1.0 的 RC1 版本现在已经提供给开发者使用了。如其网站所述，这个工具提供一个小巧但是功能强大的内核，当要扩展这个内核的时候，开发者可以开发各种插件，而不是试图在内核中生成一个能够做所有事情的巨大的库。

原文链接：http://www.infoq.com/cn/news/2008/10/merb-1-0

ThoughtWorks 发布功能测试自动化平台——Twist

作者 Chris Sims **译者** 沙晓兰

ThoughtWorks 公司开发了一个针对 Java 应用的功能测试集成开发平台——Twist。这个工具将用户故事建档、抓取可执行的请求、开发、维护、做功能测试以及发送相关测试报告集中到同一个平台上。目前，您可以下载到 Twist 的免费版。

原文链接：http://www.infoq.com/cn/news/2008/10/twist-announced

Mono 2.0 正式发布

作者 Jonathan Allen **译者** 张善友

Mono 2.0 已经发布。虽然仍然在某些方面落后于微软的.NET，在其它方面已经走到前面。例如在运行于 64 位的系统上时，Mono 支持数组的 64 位索引，这是微软还没有实现的一个 ECMA 规范。

原文链接：http://www.infoq.com/cn/news/2008/10/Mono-2


1kg.org
多背一公斤
爱自然 | 更爱孩子

架构师 试刊号

每月 11 日出版

总编辑：霍泰稳

总编助理：刘申

编辑 :宋玮 朱永光 李剑 胡键 郭晓刚 李明

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
13810038718 010-84725788

《架构师》月刊由 InfoQ 中文站出品（ www.infoq.com/cn ），Innobook 制作并发布。

更多精彩电子书，请访问 Innobook。