时刻关注企业软件开发领域的变化与创新

架构师

8月 ARCHITECT

特别专题

可伸缩性架构设计原则和实践

可伸缩性原则

可伸缩性最佳实践：来自eBay的经验

可伸缩性的最差实践

构建的可伸缩性和达到的性能

动态语言企业应用优缺点浅析

揭秘淘宝自主研发的文件系统——TFS

企业中的NoSQL

重构TekPub——从ASP.NET MVC框架迁移到Ruby on Rails

SOA十诫

Spring MVC与JAX-RS比较与分析

InfoQueue
Enterprise Software Development Community

每月8号出版

篇首语 | Editor's letter

七月骄阳红胜火

红红火火的七月过去了，随之而来的八月逐渐凉爽了。随着气温降低，头脑似乎也清醒了许多。

在计算机这个圈子里，从来不缺乏热点。从 J2EE 重型架构到 Spring 轻量级框架，从静态语言到动态语言，从 C++/Java 到 Ruby/Python，从大型机到 PC 到虚拟化，从集中计算到到云计算，从 Web Servcie 到 SOA，从胖客户端到瘦客户端，从关系数据库到 NoSql 数据库，从 Ajax 到 Flex.....计算机领域新技术的发展，从未止步。

有时候，面对如此迅速的技术进步，面对眼花缭乱的框架和名词，会有种乱花渐欲迷人眼的感觉，常常会有种时间不够用的紧迫感，有很多东西想去学，想去碰，实际上由于种种原因并没有真正的动手。现在回想起来，有很多东西如果当时坚持做下来，到今天会有很大的收获。值得欣慰的是，技术文章的翻译一直没有放弃，算是坚持下来了：)

除了翻译，是对技术的追求，对 Java 的喜爱，是一直坚持下来了。其实，技术本身没有优劣之分，做前端、做后台、做 web、做桌面、做互联网、做企业应用都有前途，都有钻研不完的地方。只要找准了方向，一直努力，就会成为牛人。

重要的是，只要选择了做技术这条道路，就要知道这不是一条坦途，不是一次短跑比赛，它需要你坚持不懈，沉淀下来，不断克服困难，才能取得进步，才能不落后。

最后，以《离骚》的名句与大家共勉：路曼曼其修远兮，吾将上下而求索。

本期主编：曹云飞

目　　录

[篇首语]

七月骄阳红胜火 1

[人物专访]

专访中国移动大云项目技术架构师钱岭博士 4

[热点新闻]

雅虎！来自 HADOOP 峰会的最新更新 7

面向对象编程——走错了路？ 10

走近淘宝开源平台 13

浏览器动态：CHROME 发布提速、SAFARI 支持插件扩展 16

微软回答“关于数据的 10 个问题” 18

移动 RUBY 近况 21

自顶向下还是自底向上—SOA 之争战火重燃 23

[特别专题]

可伸缩性原则 26

可伸缩性最佳实践：来自 EBAY 的经验 31

可伸缩性的最差实践 37

构建的可伸缩性和达到的性能：一个虚拟座谈会 43

[推荐文章]

动态语言企业应用优缺点浅析 50

揭秘淘宝自主研发的文件系统——TFS 55

企业中的 NOSQL 62

重构 TEKPUB——从 ASP.NET MVC 框架迁移到 RUBY ON RAILS 75

SOA 十诫 82

SPRING MVC 与 JAX-RS 比较与分析 106

[新品推荐]

NETBEANS 6.9 发布，支持 JAVAFX、JAVA EE 6 及 OSGI 119

ENTITY FRAMEWORK FEATURE CTP 4 更新 119

GRAILS 1.3.3 发布 119

APACHE 软件基金会发布 TOMCAT 7.0 120

MEMBASE，NOSQL 家族的一个新的重量级的成员 120

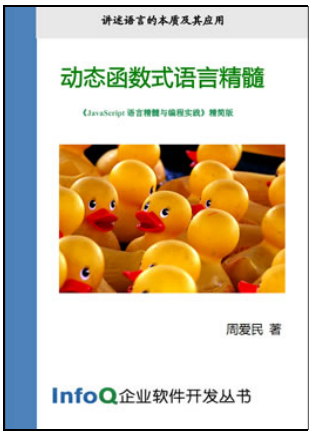
APACHE COUCHDB 1.0 破茧而出 120

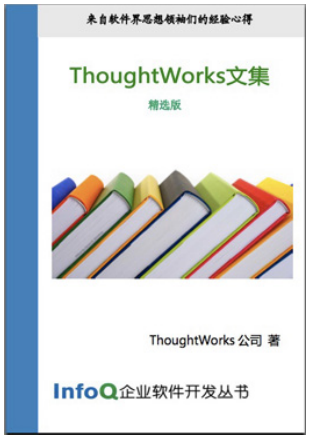
[封面植物]

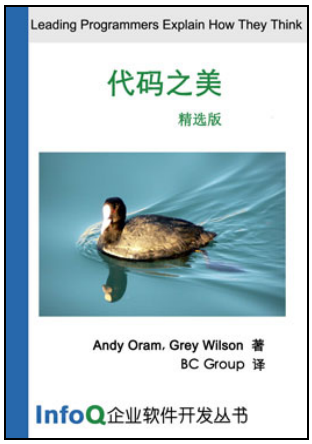
黄山木兰 122

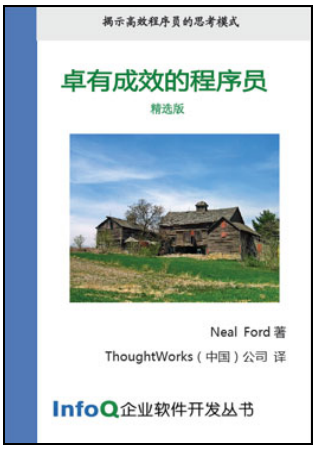
InfoQ企业软件开发丛书

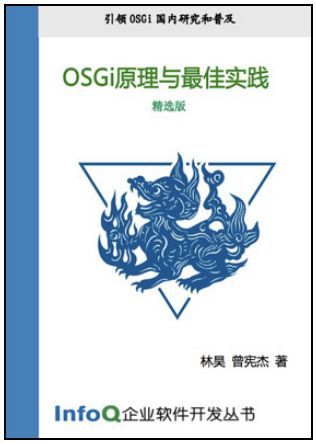
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com

"敏捷技术日"由敏捷领域最具有影响力的技术社区InfoQ中文站发起和组织，旨在进一步在国内技术社区推广敏捷概念和实践，将敏捷实施领域备受关注以及比较前沿的领域整合在一起，以培训和分享的形式深入探讨中国企业如何更好地实施敏捷，以及正确地使用敏捷。"敏捷技术日"是开放的社区活动，同时邀请国内外有着相同使命的企业和个人共同参与。

- **活动形式：** 1天=1个培训+2个演讲+3个嘉宾+开放空间
- **活动人数：** 100人
- **活动主题：** 所有与敏捷相关的主题
- **讲师嘉宾：** 全部来自社区推荐的一线敏捷实践者

第一期主题：Scrum实战

企业现在谈敏捷应用，十之八九离不开Scrum，其原因在于作为一个更偏向团队管理的"工具"，Scrum具备容易理解，上手快，见效快的特点。但是如同一门功夫虽然招式简单，但要达到最佳效果，需要反复训练一样，要应用好Scrum也不是件容易的事情。这也是为什么现在市面上有那么多Scrum培训、Scrum书籍的原因之一。我们希望能够邀请在Scrum应用方面有经验的专家，和与会者分享他们的实践历程，其中的成功、挫折经验均可以为后来者提供有益的参考。

8月21日，北京，京仪大酒店，立即报名 http://www.infoq.com/cn/zones/agileday/

讲师简介：

Scrum框架之Sprint实践

鲍央舟本人是具有认证的ScrumMaster（CSM），也是具有认证的Scrum Product Owner(CSPO)，还是认证的Scrum Practitioner（CSP）。

淘宝广告技术部开发流程和Scrum实践

讲师介绍：苏宁，2004年加入Yahoo!中国，参与了Yahoo! 中国第一代搜索竞价广告系统的开发。作为Yahoo! 中国ContentMatch广告系统的主要设计与开发人员，一直参与研发互联网广告相关系统。

汤森路透中国研发中心的敏捷实践

陈斌，1994~2004期间工作于中国电力科学研究院，负责软件开发及项目管理。2004~2008期间就职于 摩托罗拉中国软件中心，负责软件过程改进。2008至今工作于汤森路透中国软件中心，软件过程改进组经理。

第二期主题：测试驱动开发

这个话题技术性很强，也是非常有意思的一个话题。如果说Scrum是敏捷实施最好的入门之作，测试驱动开发则是实施敏捷很好的技术基础，这儿我们可以将持续集成、重构等已经为人熟知的实践纳入到该话题中。

第三期主题：精益软件开发

虽然在国外精益软件开发已经为众多的公司所应用，包括在InfoQ开发团队内部，但是在国内却依然是雷声小雨点也小，这可能和社区对这一技术的认知和推广有关系。虽然精益也是从Scrum中演变而来，但是目前显然它已经可以自成体系，用来指导软件企业更好地利用现有的资源快速开发软件或者基于Web的应用。我们希望通过这次Agile Days能够让更多的开发人员了解精益软件开发，以及如何尝试在工作中应用这些技术，并观察国外目前在这些领域的一些探索和案例。

活动信息

活动时间：
Aug 21th（from 9am to 6pm）
Sep 4th（from 9am to 6pm）
Sep 25th（from 9am to 6pm）

活动地点：
京仪大酒店，第一会议室

参会费用：
400元/场（含全天课程、讲义、午餐、茶歇）

是的，你没有看错！一次周末聚餐的价格，1个培训+2个演讲+与3位嘉宾学习+与100位与会者交流的收获！

相关培训

1. CSM Certified ScrumMaster培训课程
从这堂课中能够真正了解怎样能够成为好的一个ScrumMaster或者Scrum团队的成员。
时间：8月19日、20日，如名额满则改为9月25日前后
费用：6000 RMB/person

2. Certified Scrum Product Owner培训课程
两天的学习您能够学到作为一个产品负责人所应该了解的必要的技能。
时间：9月25日前后
费用：7000 RMB/person

课程说明：
以上课程由独立讲师提供，面向社区，旨在更广泛的推广敏捷，扩大敏捷的影响力。课程报价低于市场普遍标准，但不作为商业培训课程之参考。课程费用含50美金/人的认证费，5人以上团购、个人自筹资金、学生及教师报名，门槛更低。

培训讲师：
Michel Goldenberg
滕振宇（Danieleng）

联系人：宋芮莹（Ring）
电话：
010 - 8988 0682
010 - 6473 8142
邮箱：
agileday@cn.infoq.com

人物专访 | Interview

专访中国移动大云项目技术架构师钱岭博士

在前不久举行的第二届中国云计算大会上，中国移动发布了大云 1.0 版本，获得业界的关注。近日 InfoQ 中文站编辑有幸就中国移动研发的大云项目和钱岭进行了沟通，主要探讨了大云项目介绍、和其他云服务的区别，以及云计算中的安全问题等。

钱岭，博士，2001 年毕业于清华大学计算机课学与技术系，主修软件工程方向。毕业后加入贝尔实验室基础科学研究院，于 2007 年底加入中国移动通信研究院，先期从事移动广告平台研究，随后加入大云项目组，担任技术架构师，同时也负责子项目 HugeTable 的研发和应用，期间申请多项发明专利，并发表两篇学术论文。

InfoQ：很多读者对中国移动大云还不是很了解，请建议介绍一下该项目，以及其应用现状等。

钱岭：用一句话来概括，大云是一个云计算平台。从技术上来看，这个云计算平台主要分为两部分，一部分用来帮助解决后端数据的分析问题，另外一部分帮助解决前端的业务交易问题。两个部分的侧重点不一样，对于中国移动来说，因为用户量和业务量巨大，而且数据一般都要求存储很长的时间，这导致我们的后台服务器压力很大。我们慢慢发现传统的技术架构可扩展性不是很好，在用户数据达到一定程度之后，我们能够做得就是再购买性能更好的服务器，从而陷入到一个恶性循环。这个问题从前国外的运营商遇到过，我们也是一样，现在随着 3G 的普及，互联网业务的增多，传统的架构显然已经不能满足巨量数据存储和分析的要求。我们希望能够在这个平台上借助一些云计算工具来帮助我们优化服务器的效率。简单来说，大云项目是中国移动为了解决在业务支撑系统中存在的计算、存储能力不足等问题，以及为打造新型互联网业务平台的一次研发实践。目前大云已经在中国移动内部的一些支撑系统等应用领域进行试点和应用！

InfoQ：大云和 **Google App Engine** 以及亚马逊的 **EC2** 等平台相比，有什么区别？

钱岭：具体到 Google 的 App Engine、亚马逊的 EC2 的后端架构的设计和实现没有完全公开。大云目前是既具备后台分析能力，也具备前端交易能力，比如存储、交易、管理等，在使用上和 EC2 没有本质区别。另外我们也在为大云平台开发相应的接口，在未来一段时间内也会逐渐提供给用户使用。具体来说，大云的前端是一个弹性计算的架构，提供基础设施的服务，包括接口和具体操作；在后端提供一个 PaaS 平台，来做分析运算等。

InfoQ：在第二届云计算大会大云的项目发布会上，我们看到很多政府官员也到了，他们对云计算的态度如何？

钱岭：现在各国政府都希望引入和引领云计算，中国政府也不例外。目前中国很多城市都出现了云计算中心，预计今后政府会从行业、部门层面来引导云计算的进一步发展。大云项目对政府而言是一个很好的技术案例，可以告诉大家云计算是如何落地和实践的。

InfoQ：大云项目的硬件设施的规模如何，软件架构主要采用了什么技术，目前中国移动在大云上有多大的投入？

钱岭：目前整个大云项目的硬件设施达到 1024 个计算节点，包括一些安全设备，如防火墙；架构设计方面采用开源和自研相结合的模式，比如后端的数据分析存储平台就用了开源的 Hadoop 系列，当然我们并没有完全照搬它们，我们选用了像 MapReduce、Hive 等比较成熟的产品，并加以改进优化，在数据虚拟化和交易方面，我们使用了 REST 架构，为平台提供了一些自服务的能力，可以让其有很好的可扩展性。到目前为止，中移动研究院已经在大云平台投入了 3 年多的研发。

InfoQ：如何让用户信任中国移动的云计算平台？

钱岭：首先中国移动的品牌是毋庸置疑的，另外也在基于大云平台做一些实际应用项目和试点，具体情况我请大家拭目以待，大云项目从立项到现在已经进行了 3 年多的时间了。虽然在云计算方面，中国移动不是做得最早的，很多互联网公司都比我们做得早，但是他们的特点是多默默地自己用，很少做宣传。而中国移动有一个很大的愿景，就是通过大云来推动整个产业的发展，将整个产业链都纳入进来，使得相关的技术都能够得到改进和调整，从而让整体的 IT 运营成本下降。换句话说，这是一个长期的项目，我想这也是取得客户信任的一个很重要的因素。

InfoQ：目前大云 **1.0** 已经和很多企业进行了合作，邀请他们使用，大云主要帮助他们解决了哪些问题？

钱岭：比如有个合作单位将其内部的应用全部迁移到大云的弹性计算平台，来通过虚拟化技术提高其硬件利用率和管理能力。还有的企业将其在线应用通过视频和动画的方式来展现，

运算量很大，这种应用也比较适合使用弹性运算的平台。另外，将来中国移动的大云发展到一定规模的时候，也可能会支持我们自己的在线业务，比如社区、邮箱、说客等业务，这些业务和其他互联网应用有着同样的需求，比如可扩展、在交易的时候需要运算能力非常强等。或者是如果某个机器坏了，可以马上使用其它机器上的备份等。畅想一下，在未来，相信很多业务系统都会放在云计算平台上。

InfoQ：安全问题一直伴随着云计算的发展，不论是用户隐私数据还是企业关键数据等，大云如何让企业和用户安全使用其服务？

钱岭：坦白来说，安全问题目前还没有很好的解决方案，有一些交易式平台通过加密来保护隐私，但是对于数据分析系统则较为困难。现在一提到云计算，就提到服务器，于是安全问题也就转移到如何保护服务器的安全性上来了。要保证每个服务器的安全，这涉及到加密和授权的问题，所以说，对于云计算平台的安全问题很大程度上还是软硬件方面的管理问题。这也是目前我们在进行中的一个研究课题。

InfoQ：最后我们来谈一下大云在未来一段时间主要会对其自身做哪些优化。

钱岭：未来我们会在数据安全性、应用能力上做针对性的研发。比如现在我们的很多用户已经习惯了传统的解决方案，要让其迁移到云计算平台，单就易用性方面来说就不是件容易的事情。另外兼容性也是很大的问题，这个是历史遗留问题了，因为很多企业软硬件配置方面存在很大的差异。还有，就是性能优化，如果都是用 MapReduce 等技术的话，运行速度上还是存在问题，这样的性能我们是不能接受的。最后一点是和技术关系不大的，就是为了让整个产业链中的厂商都能够尽早转移到对云计算平台的支持上，我们还需要打消他们的疑虑，告诉他们转变的好处。这就好比人类发明了电灯泡，绝大部分人都很高兴，除了生产蜡烛的厂商。我们希望再有一年左右的时间，到大云 2.0 的时候，上面所提到的这些问题都能得到很大的改观。

原文链接：

http://www.infoq.com/cn/news/2010/07/chinamobile-big-cloud

相关内容：

- Joyent：别样的云计算平台
- Stu Charlton论云计算
- 云计算时代：LAMP何去何从？
- 云计算七问七答　Amazon EC2 推出基于Hadoop的MapReduce

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

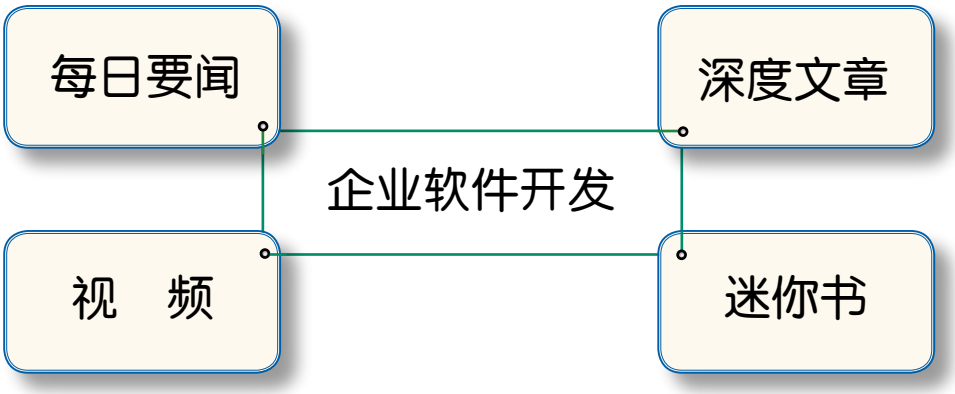
我们的**定位**：关注高级决策人员和大中型企业

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

热点新闻 | News

雅虎！来自 Hadoop 峰会的最新更新

作者 Ron Bodkin 译者 晁晓娟

雅虎首席产品官 Blake Irving 吹响了 2010 Hadoop 峰会的呜呜祖啦(2010 南非世界杯助威工具)雅虎发表的主题演讲阐述了他们的使用规模，贡献的技术方向，以及他们如何应用这项技术的架构模式。

显而易见，Hadoop 受到越来越多的关注：今年的会议有 1000 人参加并且门票在会前 10 天就告售罄，人数较前年的 300，去年的 650 有大幅的增长。Java 之父，James Gosling 也参加了该会议。这次会议标志着 Hadoop 5 周年（大约）。Irving 指出世界上只有 5%的数据是结构化的，而非结构化数据一直保持极大的增长，这些新产生的数据的特点是更多的瞬时性。他强调 Yahoo 使用 Hadoop 来分析每一个页面点击并优化内容的排名，每 7 分钟更新一次结果。他指出"我们相信 Hadoop 已经为主流企业的应用做好了准备"。

雅虎的云计算高级副总裁，Shelton Shugar 指出，雅虎每天为 1000 亿事件产生 120TB 数据输入，目前储存了 70PB,而其最高存储容量是 170PB。雅虎每天处理 3PB 数据，每个月在 38000 台服务器上运行超过百万个任务。由于雅虎的 Hadoop 的使用范围不断扩大，他们已经需要为主流应用程序员作为准备，建立支持提供和更好的 管理工具和数据安全。他指出，雅虎在生产环境中将 Hadoop 应用于各种产品：

- 数据分析
- 内容优化
- 雅虎公司反垃圾邮件
- 广告产品
- 广告优化
- 广告选择

- 大数据处理和 ETL

雅虎还在其应用科研中大量使用 Hadoop，比如：

- 用户兴趣预测
- 广告库存预测
- 搜索排名
- 广告定位
- 垃圾邮件过滤

Eric Baldeschwiele，雅虎 Hadoop 软件开发副总裁指出，在去年雅虎已经：

- 把他们的集群从每个 2000 节点增加到 4000 个节点
- 受惠于摩尔定律带来的 CPU 计算能力增加，每个节点的任务数翻了一倍
- 现在有超过 80%的磁盘利用率，通常 50-60%的 CPU 使用率，并且数据使用的增长速度高于处理使用
- 贡献了超过 70%的 Hadoop 补丁

他们在去年的重点是改善 Hadoop 的 map-reduce,这包括：

- 一个新的容量调度程序
- 任务跟踪的稳定性和支持混合工作负载的健壮性
- 增加资源的使用的限制：安全围栏（safety rails）

现在他们的重点是开发 Hadoop 的分布式文件系统，HDFS：

- 在他们的每一个集群节点的存储现在是 12TB。他们现正兴建一个 48PB 的集群 - 由于 Name 节点可伸缩性的限制，“这对 Hadoop 来说是颠覆性的“
- 提高内存，连接和缓冲区的使用，并提供度量的体系。
- 把存储拆分成一组文件卷集（使用多个 HDFS 集群）
- 在 Hadoop 的下一个主要版本将会发布跨 HDFS 实例的联合存储

Baldeschwieler 解释雅虎如何个性化他们的主页：

- 实时服务系统使用 Apache 从数据库中读取从 user 到 interest 的映射
- 每隔 5 分钟，他们使用生产环境中的 Hadoop 集群基于最新数据重新排列内容，并每 7 分钟更新结果
- 每个星期，他们在 Hadoop 科研集群上重新计算他们关于类别的机器学习模式

雅虎 Mail 以类似的方式使用 Hadoop：

- 在生产集群上频繁根据垃圾邮件模式为邮件计分
- 每隔几个小时在科研集群上训练反垃圾邮件模型。
- 该系统每天推动 50 亿次的邮件投递，覆盖了 4.5 亿个邮箱

因为 HDFS 有一个单点故障（Name 节点），这对高可用性生产系统来说是个风险。为了减轻该风险，雅虎将数据复制到多个群集，因此分布式文件系统的中断可以使用备份文件系统来弥补和解决。在雅虎的演讲中，除了自己的 Pig 项目，他们表示正在使用 Hadoop 的 Hive 项目。

Baldeschwieler 宣布，雅虎已经发布了 Hadoop Security 的 Beta 测试版，它使用 Kerberos 进行身份验证，并允许在同一集群托管商业敏感数据。他们还发布了 Oozie，一个 Hadoop 的工作流引擎，这已在雅虎成为事实上的 ETL 标准。它集成了 MapReduce，HDFS，Pig 和 Hadoop Security。

总体而言，雅虎展示了其在 Hadoop 技术的持续领导地位，与此同时他们感到高兴的是，领先的互联网公司和独立技术供应商纷纷加入到了这一生态系统当中来。

原文链接：http://www.infoq.com/cn/news/2010/07/yahoo-hadoop-summit

相关内容：

- 微软发布了分布式计算技术Dryad和DryadLINQ的学术版
- Qizmt：MySpace的开源MapReduce框架
- 雅虎将Traffic Server捐献给Apache基金会
- 分布式计算开源框架Hadoop介绍

热点新闻 | News

面向对象编程——走错了路？

作者 Dave West 译者 池建强

“如果我们现在回头看一下面向对象这个思想是从哪来的，如果以基于消息传递机制的 Smalltalk-80 的特性来衡量现在的状态继承和面向对象的使用方式，我们不禁要问，我们是不是已经走错路了？”这是在 2010 伦敦 QCon 大会采访中提给 Joe Armstrong 和 Ralph Johnson 的一个公开问题。Joe Armstrong 是 Erlang 语言的早期开发人员，而 Ralph Johnson 则长期从事 Smalltalk、面向对象编程和模式等领域的研究。受访者都表明目前我们都已经在这条“错路”上了，但这是由于实现上的缺陷 导致的，并不是面向对象的思想本身有问题。Ralph Johnson 的原话是这样的：

> 事实上这样的事情总在发生，你有了一个好的想法，并把它公布了出去，但对于大多数人来说，它总是显得很激进。很多人不愿意全盘接受这套理论，他们总是采用其中的一部分，这样，你就会看到很多相似的东西出来。

甚至是 Smalltalk，被很多人奉为面向对象语言的典范，也只是在向面向对象这个理念无限逼近。Johnson 博士为 Smalltalk 提了两个具体的问题：

> … 我想 Smalltalk 犯了一个根本性的错误。我觉得让那些不用 Smalltalk 编程的人去理解 Smalltalk 是非常困难的，但是如果你在使用 Samlltalk 进行编程和调试，你会发现你是在调试整个系统。

而且

> 在 Smalltalk 中你可以从映像中得到任何你想要的对象。但你没法对新旧对象的版本进行跟踪 … 还有，复杂度总是一个问题。你构建了一个系统，只有少数人能够维护它，而且 Smalltalk 并不是总能工作的很好。

事实上，把所有的东西(你的应用类，开发和调试工具和库)都放在“映像”中一直是 Smalltalk 的问题，即使是对这门语言的崇拜者来讲也是一样。但是，这并不妨碍人们去构建大型的关键任务的系统，例如 Cargill Lynx 项目。Lynx 是全球的粮食贸易系统，该系统支持的用户超过

1500，在全美有 150 多个站点，而且已经运行了十多年了。在 Lynx 项目的生命周 期内，有超过一百名程序员对其进行开发维护，全版本控制，健壮性测试和调试。类似 Lynx 这样的巨型项目，并不能消除 Ralph Johnson 的质疑，当然他们也没有否认 Smalltalk 是一个有缺陷的面向对象的实现。他们只是从不同视角提出了自己的观点。

什么样的特征能够让一门语言具备面向对象的特性，这个问题在 90 年代进行了广泛和激烈的辩论。在伦敦 QCon 大会的采访中，Joe Armstrong's 的论文导师的话被引述之后，同样引发了争论：

> 我开始想知道面向对象编程到底是什么，而且我认为 Erlang 不是面向对象的，它是函数式编程语言。那我的论文导师就说了："你是错的，Erlang 是完 全面向对象的"。他说面向对象的语言其实不是去面向对象。我在想，虽然我不能确信是不是能相信这句话，但 Erlang 可能是唯一的面向对象语言，这时因为 Erlang 具备面向对象编程的三原则：基于消息传递机制，对象分离和多态。

Armstrong 博士表示，他并没有被他的导师的论点说服，但似乎认为 Erlang"可能是唯一的面向对象语言。"除了之前提到的三个特征，单继承和动态类型也被引用为面向对象语言的"绝对需求"。

在这次采访中，Johnson 博士和 Armstrong 博士都表示，面向对象的思想在过去和现在都是非常重要的。Armstrong 博士花时间分析了 Erlang 是如何试图实现面向对象的思想和语言的特性。Johnson 博士则花了更多的时间批判以前的试图实现面向对象思想的实践，特别是 Smalltalk。这有点奇怪，因为长期以来他一直是 Smalltalk 和面向对象思想（**Johnson** 博士合著的设计模式的书的标题包含了面向对象）的倡导者，但是 Ralph Johnson 的论点并不孤立。

Dave Thomas 是另一个对面向对象和 Smalltalk 有相近观点的人。他的团队为 Digitalk's Method（第一个 PC 上的 Smalltalk 应用）创建了第一个"Goodies Packs"，他是该公司的创始人和 CEO，他们的成果最终成为了 IBM 的 VisualAge Smalltalk，而且他的团队开发了非常流行的工具 Eclipse（本来是为 Smalltalk 写的）。Thomas 博士被引用的话包括"面向对象是个错误"和"我是国家罪人。" 这些描述当然是为了增加戏剧效果，但同样也指出了 Smalltalk 在实现对象思想时的"错误"——例如，只关注状态，在类和基于映像的语言里缺乏良好的并发模型和消息机制。

Alan Kay 最先提出了"面向对象"词条，他和 Dan Ingalls 以及其他在 Xerox PARC 共事的人发明了重要的 Smalltalk 语言。在最近的澳大利亚计算机世界的采访中，他谈到：

> 我确实创造了这个词条（而且现在看来这是个糟糕的选择，因为它并没有强调消息传递的重要性）。现在其中一部分思想仍然存在（在几个系统中）。考虑到所有的有效虚拟机的通讯都能通过消息进行交互，我们其实可以建立一个更为全面的理论基础。这样的基础可以提供更好的伸缩性，是我的研究社区的虚拟版本，ARPA-IPTO（美国国防部研究机构的信息处理技术办公室）已经开始构建大规模网络，同时具备强大的“代数”属性（类似多态性）…… 然而，我并不是 Smalltalk 的粉丝，虽然至今 Smalltalk 仍然比其他大多数编程系统更好用（我不喜欢其中任何一种，而且至今我也不认为它们适 合解决真正的编程问题，无论是对系统还是对终端用户）。

最近对 Smalltalk 的批评使我想起了在旧杯垫歌曲中重复的那句歌词，**Charlie Brown** 唱到：“（为什么每个人都针对 我）。” 当然，Johnson、Armstrong、Kay 和 Thomas 都没有“针对我（Smalltalk）” 。在伦敦 QCon 大会采访中，其他相对低调一点的批评者提出，正如 Johnson 博士所言，好的思想总是“对大多数人显得很激进”，那么什么程度的思想才 能够被任何程序语言实现呢？

这很可能是因为编程语言无法直接通过定义和结构去描述面向对象的思想。在澳大利亚计算机世界的采访中，Kay 博士指出：

> 对我来说，采用真实对象语义的好处之一就是它们是“real computers all the way down （RCATWD)”——这意味着真实对象保留了全部的能力，可以描述任何事物。用旧的方式虽然可以快速获取数据和过程，但它们不是电脑，延迟优化和分析某些行为是否合理的能力会突然丢失。换句话说，总有真实对象保留着能力去模拟你想要的东西，并把它发送到周围环境中…… 而且 RCATWD 还提供了两个方向上的完美保护。我们能够从网络（可能是工作中唯一真正面向对象的系统）的硬件模型中了解这种思想。你只需要简单的接受消 息格式的约定，就可以免费获得语言的扩展性。我在 70 年代的想法是，我们一起在互联网上工作并进行个人计算，这是一个非常好的可扩展的设计，而且我们还可以通过虚拟机构建虚拟网络，通过硬件机器进行缓存等。非常糟糕的是这些想法并没有得以实现。

如果‘真实对象’是 RCATWD，那么就可以使用编程语言以最恰当的方式实现每个对象的内在特性，这将为‘多语言编程’赋予新的含义。

原文链接：Object Oriented Programming: The Wrong Path?

相关内容：

- 将Javascript作为编译目标：Clamato、GWT Smalltalk、Python、Scheme

热点新闻 | News

走近淘宝开源平台

作者 崔康

淘宝开源平台自 6 月底上线以来，引起了国内社区的广泛关注。目前，平台已经发布了若干开源项目，其中不乏来自于淘宝之外的项目在此落户。为了使国内开发社区的朋友对淘宝开源平台有进一步的了解，InfoQ 中文站对平台的负责人残剑（全佳营）进行了专访。

InfoQ 中文站：淘宝提供开源平台的初衷是什么?

> 对于开源平台，我们最初的设想是通过类似于实验室的形式，形成一套完整的软件研发过程体系，从初期的创意开始，我们提供完善的平台，供用户发布创意，汇集创意并通过多人的交流能不断完善创意，这是第一阶段。
>
> 当创意发布者通过创意平台，拉到一部分志同道合的用户时，即可申请立项，从而开始项目阶段，同时，我们的平台提供出完整的项目支持体系，包括项目管理、文档管理、测试沙箱环境等，这是第二阶段。
>
> 在项目发布之后，又可以提供下载、演示环境。开源平台解决用户很多资源缺乏的问题，无论是硬件或是开发人力等，也起到宣传创意项目的作用。
>
> 最后，我们更希望聚集起开源爱好者，为国内技术领域提供一个良好的沟通交流平台，而淘宝的技术也将先开源，带头做好开源意识。

InfoQ 中文站：淘宝开源平台的架构是怎样的？采用了哪些技术？

> 目前，平台的技术采用 python+django 开发，后端使用 mysql 数据库，目前整合了 trac 的功能，后续会自行开发相关功能，毕竟 trac 的功能有些还是不适合国内用户的习惯的，而整合度也不够好。
>
> taobaocode(淘宝开源平台)本身也是作为第一个开源的项目，放上平台，目前刚起步，希望有更多人来参与，平台不只是属于淘宝的，而是属于更多技术爱好者。

InfoQ 中文站：淘宝开源平台审批项目的原则是什么？

> 对于项目审批原则，可能外界觉得淘宝审批太慢，目前我们希望是有高质量的项目能给用户看到。
>
> 其实，未审批的项目也是可以上传代码的，我们审批会根据用户对项目的描述和适当查看项目本身代码质量去决定。

InfoQ 中文站：开源平台未来的发展规划怎样?

> 淘宝开源平台目前刚起步，未来我们会推出更多的功能，我们希望平台不仅是代码开源平台，更是开源思想的平台，有交流，有热情，体验开源的过程，同时享受开源的结果，有产出。所以，在未来的规划上，我们希望平台在技术支持上提供更多开发、测试工具，在沟通上，有更好的交流方式，能沉淀下很多很好的知识体系，同时，我们也会积极与国内的开源组织开展各种线下活动，拉近用户距离。

InfoQ 中文站：能否简要介绍几个已经发布到平台上的淘宝开源项目？

就此报告，Forrester 的 Randy Heffner 评论道，最有意思的结果是：

> 目前，在平台上的淘宝开源项目，主要是核心系统部的 tair、taobaocode 和 nginx_concat_module 项目。
>
> - tair 是淘宝内部的分布式高性能的存储系统，目前支撑了淘宝大部分业务系统，为淘宝每天的上亿的交易量提供了保障。
> - taobaocode 则是我们的开源平台。
> - nginx_concat_module 项目是 Apache mod_concat 的 Nginx 增强版本。主要功能是组合 JS 和 CSS，使得多个请求合并成一个，进而减少用户的响应时间，提高用户体验。
>
> 淘宝的其他项目也在准备着开源，期待有更多更好的项目共享给业界。
>
> 另外其他开源项目则是由用户提交上来的，可以关注下，还有很多很好的创意，我想更多的需要通过交流沉淀下来一些东西。

InfoQ 中文站：您想对国内开源社区的朋友们说些什？

> 我们希望国内的开源能发展的更好些，有更多的人参与，有更好的平台支持，淘宝本身也会大力支持开源项目，如果有好的项目上来，通过淘宝技术委员会的审核，淘宝可以资助其团队开发，我们相信国内的开源也会发展的如火如荼。

InfoQ 中文站将会继续关注国内开源社区的发展，也欢迎读者朋友发表自己的看法。

专家介绍：全佳营，花名残剑，目前在淘宝运营支撑部门，主要负责研究流程引擎这块的工作，通过流程编排现有的服务，后续正计划开发适合淘宝业务的执行引擎，更好的整合流程和规则引擎等。

原文链接：http://www.infoq.com/cn/news/2010/07/tabao-opensource

相关内容：

- 开源网络分析工具TCP Traffic Analyzer
- AppengineJS：使用JavaScript访问Google App Engine Python SDK
- 开源技术选型手册
- 豆瓣数据存储实践
- 浅谈可扩展的MySQL数据库设计

热点新闻 | News

浏览器动态：Chrome 发布提速、Safari 支持插件扩展

作者 崔康

随着 RIA 技术的不断发展，浏览器市场竞争越来越激烈，本文简要总结了近期几款浏览器的最新进展。

Internet Explorer

根据有关报道，微软首席运营官 Tevin Turner 最近在微软年度财务分析会上表示，下一代浏览器 IE9 将在 9 月份进行公开测试。目前，IE 9 平台预览第三版供用户下载试用。不过，微软之前发布的 IE9 平台预览版并非完整浏览器，而 Beta 版则会首次提供完整的 IE 功能，包括用户界面等。

Firefox

Firefox 4.0 beta 2 开始提供 23 种语言版本供用户下载试用。对于 Firefox 4.0，开发社区最为关注的是全新的 JavaScript JIT 引擎——JaegerMonkey， 从 V8 基准测试来看，目前的 JaegerMonkey 性能还不是很乐观，但是 Mozilla 称，JaegerMonkey 是重新编写的，过去 8 周的时间 JaegerMonkey 已经有很大的改进。在下面 6 周的时间，Mozilla 将完善 JaegerMonkey 引擎，为 9 月 1 日顺利发布做好充分地准备。

Chrome

虽然 Google 是浏览器市场的新军，但是其表现一直不俗：不仅性能出色，而且发布速度也很快。最近，Google 宣布 Chrome 的发布再次提速，大约每两周交付一个稳定版本，Google Chrome 项目经理 Anthony Laforge 指出其目的是：

1. 加快发布周期，同时仍然为热衷升级的用户提供更丰富的特性，免去“漫长的”等待。
2. 让发布进程更好预测、更容易达成，也方便项目管理，能够确定在特定时间内所需要完

成的工作量。

3. 减轻工程师们发布一个新版本的压力。

相信不久之后，大家会看到 Chrome 7.0、8.0、9.0....

Chrome 6 的一个重要特性是添加了对页面加载计时（Web Timing）的支持， 据悉，运行 Chrome 6.0.472.11 的开发者可以在 window.webkitPerformance 下获取新的网页性能测试结果。Google 公司软件工程师 Tony Gentilcore 透露："Web Timing 技术可以让开发者测试网页加载时间，包括响应时间、生成时间和接收 HTML 文档的时间.....用户可以清楚地了解到浏览器把时间都用到了哪里。"

Safari

苹果发布了 Safari 浏览器的 5.0.1 版本升级，正式引入了 Extensions 插件扩展功能。

Safari 浏览器扩展由第三方开发者使用 HTML5、CSS3、JavaScript 等开放网页标准写成，跨平台兼容 Mac OS X 和 Windows 版浏览器，类型涵盖工具栏、网页过滤包等常见浏览器插件种类。安全性和简易性也令人瞩目：

> 扩展插件运行在独立沙盒中，不会导致用户系统或个人信息泄露，苹果还会通过数字证书保证其安全性。
>
> 目前已经推出 Safari 扩展的知名企业包括微软 Bing 搜索，纽约时报、Twitter、亚马逊、eBay 等。苹果在官方网站上专门开辟了一个"Safari 扩展库"页面，介绍了数十款优秀插件。在该网页点击安装即可直接从开发者网站下载安装，无需重启浏览器即可使用。用户可以在 Safari 浏览器的帮助菜单找到"Safari 扩展库"，偏好设置选项中则可以管理已安装扩展，并配置自动更新。

最后，针对读者朋友做一个浏览器关注度的小调查，请到原文中进行投票。

原文链接：http://www.infoq.com/cn/news/2010/08/browser-trends

相关内容：

- ANGLE为在Windows下进行WebGL渲染提供了便利
- Internet Explorer 9 Preview 3 提供了更好的HTML 5 支持
- 代号Gazelle，微软研发浏览器上的操作系统
- HTML 5 通过sandbox属性提升iFrame的安全性

热点新闻 | News

微软回答“关于数据的 10 个问题”

作者 Abel Avram 译者 王瑜珩

微软回答了他们称之为“关于数据的 10 个问题”，解释了 Oslo、ADO.NET Data Services、WCF、LINQ to SQL、T-SQL 以及其它技术的发展情况和未来的计划。

Oslo

微软宣称“会继续开发这些技术”，但不会继续使用“Oslo”这个代号，而是改名为 **SQL Server Modeling CTP**。由于与 SQL 相关技术的紧密联系，特别是 Quadrant（译注：用来查看和修改数据库中数据的工具）和 Repository（现在叫做 SQL Server Modeling Services），这些技术将来会被集成到 SQL Server 中。

微软还解释了 SQL Server Modeling 和.NET 之间的联系：它使得创建模型驱动的应用程序更加容易。

ADO.NET Data Services 和.NET RIA Services ADO.NET Data Services 变成了 WCF Data Services，而.NET RIA Services 则变成了 WCF RIA Services，目的是使 WCF 成为创建服务和多层应用的一站式框架，ADO.NET Data Services 和.NET RIA Services 在此方面完善了 WCF。

LINQ to SQL

LINQ to SQL 在.NET 4.0 中将会继续得到增强：

> 没错，我们会继续开发 LINQ to SQL。但是我们会更偏重于 Entity Framework，因为此框架是围绕 Entity Data Model（EDM)构件的。而 EDM 代表了微软很多产品的主要战略方向，包括 SQL Server、.NET 和 Visual Studio。

这条消息明确表明：LINQ to SQL 将不会被包含在更长远的计划中。

“M”语言

微软对于这个语言的定位很模糊。首先，他们说 M 将会和 Oslo 一样，被集成到 SQL Server

中：

> 微软正在整合“M”语言和 Entity Data Model，以便 Entity Framework 和 Data Services 可以使用。这将导致同一种数据模型拥有两种不同的表现方式：CSDL，目前 Entity Data Model 所使用的基于 XML 的语言，更适合互操作；或者“M”语言，更适合开发人员。

目前仍不清楚 M 语言将会走向何方。可能将会被包含在 SQL Server 中，但以后将会整合到 Entity Data Model 中。

“M” vs. T-SQL

由于 M 语言还没有发布，微软建议目前还是继续使用 T-SQL，但以后则会不同：

> 一旦“M”语言发布，就可以使用“M”语言在更高的抽象层次描述问题域。然后可以将“M”语言转换为成 T-SQL、C#或者其他格式，以便在本地运行环境中做进一步的优化。

WCF Data Services

微软强调了.NET 4.0 和 VS 2010 中包含的 WCF Data Services 新特性：

- 双向数据绑定
- 计算数据的总行数
- Feed 定制化
- 服务器端分页
- 增强的 BLOB 支持
- 为 Custom Provider Writers 提供的新的“Data Service Provider”接口
- Projections

ADO.NET Entity Framework

ADO.NET Entity Framework 最重要的特性是：

- 外键
- 支持 POCO
- 新模型默认使用延迟加载
- 为 Query Extender 和 POCO 提供的 EntityDataSource

- 支持二进制类型的主键和外键
- ObjectMaterialized 事件
- 改进的 Object Service API，支持 N 层架构和 Self Tracking Entities
- 改进的 SQL 生成机制
- Navigation Property 管理
- 改进数据库生成机制
- 新的扩展 API
- 根据存储过程的返回值生成复杂类型
- 改进的管理工具
- 改进的 LINQ to Entities

微软还解释了什么是 Quadrant - “用于显示、查询和编辑 SQL 数据库中数据的工具，具有多种不同的视图，如树形、列表、表格以及 master/detail”，以及什么是 Open Data Protocol（OData)，请参考 InfoQ 以前的新闻。

原文链接：http://www.infoq.com/cn/news/2010/07/Top-10-Questions-on-Data

相关内容：

- 微型ORM——用VB和C#编写的动态类型ORM，只有 160 行
- LLBLGen Pro 3.0 现已支持多种持久层框架
- 即将到来的.NET访问控制服务
- 表达式即编译器
- 使用Brahma在GPU上执行LINQ

热点新闻 | News

移动 Ruby 近况：Rhodes 2.0 现遵循 MIT 协议，通过 Ruboto 在 Android 上使用 JRuby

作者 Werner Schuster 译者 丁雪丰

Rhomobile 的 Rhodes 是少数几款针对移动应用程序的跨平台框架之一。除了在应用程序的 GUI 部分使用 HTML/Javascript/CSS 以外，Rhodes 应用程序的逻辑是由 Ruby 来编写的。

Rhodes 2.0 的最终版现已提供下载，该框架目前遵循 MIT 协议。除了 Rhodes 2.0 Beta 版中的特性外，框架还允许使用跨所有系统的平台映射组件。

在苹果修改 iOS SDK 协议后，很多针对 iPhone 的第三方编程工具的未来都生死未卜，尤其是像 Rhodes 这样自带解释器的。Rhodes 2.0 的发布说明中表示 Rhodes 应用程序目前仍被 AppStore 接受：

> 由于大家比较关心，故特此更新——Rhodes app 仍被 App Store 接受（我们之前就给出过暗示，他们会接受的），即便在 iPhone 4.0 服务条款生效后也是如此。最近被接受的一些 app 包括 TrackR 2.0（一个针对 PivotalTracker 的免费 app）、Metalingual（Sarah Allen 的一个语言翻译 app）和 WorshipPlanner。

此外，正如几个月前声明的那样，在未来的版本中将会移植 Rhomobile 的同步库，至少会有 Objective-C 的版本。

欲了解更多 Rhodes 的内容，请移步 QCon 伦敦时录制的 InfoQ 对 Rhomobile 的 Adam Blum 的采访。

在 Rhodes 使用 **Ruby** 本地版本带来跨平台的移动开发（除了黑莓）的同时，另一些人正致力于让 **JRuby** 运行于 Android 之上。Ruboto 项目将 JRuby 带到 Android 上，Ruboto-IRB 允许开发者直接在 Android 设备上编写 JRuby 脚本。

Ruboto 的创始人是 JRuby 的 Charles Nutter，现在它是 Daniel Jackoway 在 Ruby Summer of Code

的一个项目，其目的是在今年晚些时候带来一款改良过的使用 JRuby 的 Android app 开发工具。

原文链接： http://www.infoq.com/cn/news/2010/07/rhodes20-ruboto

相关内容：

- JRuby近况：1.5、AOT、Java 7
- 运用Ruby纤程进行异步I/O：NeverBlock和Revactor
- 标准化对Ruby意味着什么
- 利用Ruby简化你的Java测试
- Rhodes：将Ruby应用带给iPhone、Windows Mobile和BlackBerry

热点新闻 | News

自顶向下还是自底向上—SOA 之争战火重燃

作者 William Martinez Pomares 译者 吴宇

开源 ESB 厂商 MuleSoft 在宣布其管理控制台发布时，声称支持使用自底而上的方法来实现 SOA 管理理念，在这之后，SOA 社区中一个一直以来争论不休的话题：使用自顶向下还是自底向上的 SOA 方法，又引起了大家新一轮的争论。

在 SearchSOA 上，Rob Barry 收集了一些关于自底向上和自顶向下两种方法的观点：

> 当增建 SOA 时，一个自底向上的治理方法会关注于将围绕一些可以快速组装的 ESB 个体的服务集成起来。这种方法由于需要过度的更新和随后的返工而遭遇批评。与此同时，与之相反的“自顶向下”的治理方法包括了周密的计划和严格的政策规范。但该方法也存在缺陷，因为它需要花太多时间才能产生结果。

文章汇集的观点认为：基本上，如果主要的目标在于集成，自底向上方法是一个不错的出发点。他们也认同自顶向下的方法需要更多业务的介入。至于到底是使用哪种策略，他们的结论是这取决于业务和 IT 的关系。

Barry 在 ebizQ 上的一篇博文点燃了导火索，引起了不多但却很有意思的回复。在其中一个回复中，Avi Rosenthal 基于你正要构建的东西对两种方法进行了区分：

> SOA 是一种架构风格。构建架构是自顶向下而不是自底向上的。Web Service，有时错误的被定义为 SOA，是技术性的。Web Services 是自底向上构建的。自底向上地构建 SOA 是错误的方法，有时也被称为 ABOS(一堆服务)。如果你自底向上的构建 SOA，那么很有可能你 最终会得到很多冗余的东西，而毫无架构可言。尽管如此，如果仅仅通过自顶向下构建 SOA，其结果往往是得到不能构建在任何运行时产出物之上的感性架构，因此应该将 SOA 的一些工作放在自底向上的事情上。总而言之：最开始，SOA 是自顶向下的方法，但实际使用中需要同时结合自顶向下和自底向上两种方法。

在对该问题的另一则回复中，Michael Poulin 认为 SOA 以用户为中心的本质迫使了自顶向下

的方法的使用：

> 如果从现在你已有的东西去构建服务——使用自底向上方法的话——最终很可能以你有什么而告终，而不是你的用户需要的。SOA 是以用户为中心、业务为导向的架构。以用户的需求为出发点将使你不得不去使用自顶向下的方法。这始终都可以作为出发点。但是，接下来，你最好评估你自身的能力，比如，挖掘你的最底层的资源来审视用户的需求。

这样的争论已经不是什么新鲜事儿了。早在 2005 年，John Crupi 就撰文称 SOA 是业务驱动的架构风格，正因如此只有自顶向下的方法才能获得成功：

> 自顶向下，意味着从问题，到架构，再到解决方案。它并不是指，从我们现有的东西着手，然后采用一些新技术对其进行包装，仅仅就因为我们可以使用这些技术。虽然自底向上的方法看起来是那么自然和简单，好像是治疗 SOA 失败的绝妙处方。

在那以前，还有其他一些诸如 Bill de hÓra 的博文那样反对“自顶向下或失败原则”的声音：

> 仅仅使用自顶向下方法的困难之处在于并不存在所谓的“顶”，现实中的 SOA 系统往往是分散式的——不存在架构杠杆支点或治理点，没有哪一个人有能力声称，“他可以十分钟之内做出决定或是下一个决定是有效力的，立刻就能付诸执行”。

这样的争论已经持续很多年了。到目前为止，看起来一些工具厂商已经选择了自顶向下的策略。

原文链接： http://www.infoq.com/cn/news/2010/07/top-down-bottom-up-soa

相关内容：

- 白话SOA
- MuleSource CTO给出是否选用ESB的检查清单
- mySOA：敏捷的、治理的并且可持续的
- 建立服务治理组织
- Spring Integration 入门

Adobe Flash Builder 4 简体中文正式版

隆重发布

出众的创新，合适的技术

Adobe Flash 平台是一套全方位的技术，外围是一个由支持程序、业务合作伙伴和热情的用户社区构成的可靠的生态系统。它们共同为您提供了为最广的观众群创建和交付最引人注目的应用程序、内容和视频所需的一切。

现在下载使用

即可体验更多最新特性，开发更炫RIA应用

更可优先免费获得产品序列号

数量有限，先下载先获得！

InfoQ中文站提供以下Adobe Flash Flatform相关产品和工具的高速下载：

- Flash Platform一览表（PDF，231k）下载
- Flash Catalyst Beta 2 292M 下载
- Flash Builder 4 285M 本地下载 Adobe 官方网站下载
- Adobe Flex Builder 3 本地下载 Adobe 官方网站下载
- Flash Player 下载
- Flex 4 SDK 下载
- Adobe AIR 15M 下载

立即下载

特别专题

策划：**InfoQ** 中文站； 执行：霍泰稳

可伸缩性是互联网企业，特别是大访问量网站所一直关注的话题。相同的硬件，相同的网站规模，不同的架构设计可以让网站的性能有天壤之别。换句话说，架构的可伸缩性与否，和架构师的水平有着莫大的关系。这也是为什么有关可伸缩性架构的讨论在现在的各个技术大会，都那么受欢迎的原因之一。本专题希望从可伸缩性的原则出发，结合相关的最差和最佳实践等，给读者提供架构设计过程中的一些可参考建议。

可伸缩性原则 - 可伸缩性最佳实践：来自 eBay 的经验

可伸缩性的最差实践 - 构建的可伸缩性和达到的性能：一个虚拟座谈会

特别专题

可伸缩性原则

作者 Simon Brown 译者 王丽娟

从最简单的水平来看，可伸缩性就是做更多的事情。更多的事情可以是响应更多的用户请求，执行更多的工作，或处理更多的数据。设计软件这件事本身是复杂的，而让软件做更多的工作也有其特有的问题。这篇文章针对构建可伸缩软件系统提出了一些原则和方针。

1. 减少处理时间

增加应用所做工作数量的一个方法就是减少完成单项工作所花费的时间。举例来说，减少处理一个用户请求所需的时间意味着你能在同样长的时间内处理更多的用户请求。这里有一些本原则适用的例子和一些可能的实现策略。

- 并置（Collocation）：通过并置数据和代码，减少因获取所需数据而产生的必要开销。
- 缓存：如果数据和代码不能并置，就缓存数据，以减少反复取数据的开销。
- 池化：通过池化昂贵的资源，减少与其使用相关的开销。
- 并行化：通过分解问题、并行化独立的步骤，减少完成一个工作单元所需的时间。
- 分区处理：通过分割处理代码、并置相关的分区，尽可能将相关的处理过程集中在一起。
- 远程处理：减少访问远程服务所花费的时间，比如可以通过更粗粒度地划分接口。远程还是本地是明确的设计决策，不能随意来回更动，这一点应当牢记。还要考虑分布式计算的第一准则——不要分布你的对象。

软件开发人员总爱在不需要的地方引入抽象和层。是的，这些概念对软件组件之间的解耦来说是很好的工具，但它们可能会增加复杂性、影响性能，尤其是在每层的数据表示之间都需要转换的情况下。因此，减少处理时间还要注意保证抽象不要过于抽象化，并且没有过多的分层。另外，对于我们视为理所当然的运行时服务，有必要理解其成本，因为除非它们提供了特定的服务水平协议，否则很有可能最终会成为应用中的瓶颈。

2. 分区

减少单个工作单元的处理时间能达到不错的效果，但当你达到单进程方案的极限，最终还是需要对系统作水平伸缩。在典型的 Web 应用中，水平伸缩可能很简单，只要加入更多的 Web 服务器来处理用户请求，再给它们加上负载均衡就行了。但是，你可能会发现总体架构的某些部分会成为资源争用的焦点，因为一切东西都会 在同一时间变得忙碌起来。一个很好的例子就是所有 Web 服务器后端的单一数据库服务器。当这个单一的数据库服务器变成瓶颈时，你必须改变方法，其中一种方式就是采用分区策略。简而言之，这涉及到将架构的单个部分分解成更小、更容易管理的部分。将单个的元素分割成更小的部分能实现水平伸缩，这恰恰也是 eBay 这样的大型网站采用、以此来确保它们的架构可伸缩的技术。分区是一个很好的解决办法，尽管你可能会发现牺牲了一致性。

至于如何分割你的系统，那要看情形而定。真正无状态的组件能简单地作水平伸缩，将工作负载分散到所有实例上，让组件的所有实例都能有效地运行。另一方面，如果需要维护某状态，你需要找到一种工作量分割策略能允许有状态组件的多重实例，让每个实例负责工作和/或数据的一个独特的子集。

3. 可伸缩性在于并发

可伸缩性天生就和并发联系在一起；毕竟，它就是要在同样的时间内做更多的工作。像 EJB 早期版本这样的技术试图提供一种简化的编程模型，鼓励我们编写单线程的组件。遗憾的是，组件往往要依赖于其它组件，还是导致了并发问题。如果没有考虑并发，系统中的数据会很容易被损坏。另一方面，围绕并发做了太多的保护会导致系统实质上变成串行的，限制了伸缩的能力。并发编程不是很难做到，在构建可伸缩系统的时候，有一些简单的原则会有所帮助。

- 如果你确实需要持有锁（比如本地对象、数据库对象等），试着尽可能短地持有它们。
- 设法减少对共享资源的争用，并尽可能是争用避开关键处理路径（比如通过异步调度工作）。
- 任何针对并发的设计都需要预先完成，以便能被充分地理解哪些资源可以被安全共享、哪里可能会是潜在的可伸缩性瓶颈。

4. 必须知道需求

为了构建一个成功的软件系统，你需要知道你的目标是什么、你针对什么去做。尽管功能性需求往往是明确的，但常常会缺少非功能性需求（或系统质量需求）。如果你真是需要构建一套高可伸缩的软件，那你首先需要调查清楚关键组件/工作流的以下特质：

- 目标平均性能和峰值性能（即响应时间、延迟等）。
- 目标平均负荷和峰值负荷（即并发用户、信息量等）。
- 性能和可伸缩性可接受的极限。

性能也许不是最紧要的方面，但你必须尽早知道这个信息，因为处理可伸缩性的方法会由性能需求决定。

5. 持续测试

理解了需求就可以开始设计和构建解决方案。我们提出的设计、编写的代码实际上都是静态的，所以你在执行之前不能完全断定它会怎样运转。此外，所有关于性能和可伸缩性的决策应该由证据支持的原因也在于此，而且应当从项目一开始就收集和审核这些证据，此后也要一直继续。换句话说，就是设立贯穿系统的可度量目标，证实并度量实际的性能，并在项目的各个阶段考虑性能。

最常犯的错误之一是，我们对系统性能和可伸缩性的见解会被我们自己的经验或道听途说所混淆。你可能要审核对工程做出的其它决策，这样做的原因之一是要满足系统的非功能性特性。比如说，非功能性需求可能会影响你选择不使用标准，改用非主流/流行的一些东西。非功能性需求可能会打破僵化的教条，证据胜过教条。

6. 架构先行

或许对构建可伸缩的系统来说最重要的原则是，如果你需要使系统具备这样的性质，就必须预先设计出这样的性质。很多人（包括我自己）陷入的陷阱，就是以为可以构建一个应用，它会自动地垂直伸缩（scale up）或水平伸缩（scale out），尤其是在 J2EE 刚出现的时候。设计为可水平伸缩的应用几乎总能垂直伸缩，但是设计为垂直伸缩的应用几乎不可能水平伸缩。大多数应用能通过在更加强大的硬件上运行来垂直伸缩，但水平伸缩却是一个更为复杂的问题。比如说，你怎么确保数据在应用实例之间保持一致性？你如何使你的单例和同步代

码块跨线程工作？

当然，预先思考这件事情不一定等同于做一个瀑布式的、预先的大设计。迭代和敏捷过程都是助力，它们能提供一个框架帮助我们可以做出刚好够用的设计来解决问题。要务实。哦，不管我们自认为是多么擅长于设计可伸缩的应用，不要相信自己、尽早地编写/测试代码才是最好的举动。

7. 着眼于全局

最后，记着要着眼于全局——看到树木之前先看看森林。对我们来说，在细粒度代码级别调整组件确实很容易，但最终需要优化的却是作为一个整体的系统。关注每一个环节的性能和可伸缩性，必要时牺牲局部的优化。如果你需要使用性能分析工具确定瓶颈，不妨去做，但在对全局的性能有所认识之前先不要急于动手。由于性能与整个系统所有等待时间的集合成反比，任何等待时间增加得比负载还快的操作都会成为问题。尽管说了这么多，但我还想指出，如果你发现满足性能和可伸缩性目标很困难，那就有必要怀疑一下是否选择了正确的架构。还是那句话，着眼于全局，确保有人在承担架构师的责任。

总结

这篇文章针对构建可伸缩应用提出了一些原则和方针，覆盖了软件开发过程中许多不同的方面。无论谁要构建可伸缩的系统，我能给他的最好建议就是你需要明确地考虑并设计你的系统。可伸缩性不是魔术，它也不会无偿获得。最后一点，更快的硬件也许能救你于一时，但还是不要依赖它为妙！

关于本文

2005 年年底，英国伦敦举办了针对架构师的非官方峰会，本文中的大多数原则就来源于其中一场可伸缩性讨论的一些笔记。该峰会由Alexis Richardson、Floyd Marinescu、Rod Johnson、John Davies、Steve Ross-Talbot组织。题为“JP Rangaswami谈企业中的开源与信息前景”的视频也来源于此峰会。

关于作者

Simon是一名注重实践的软件架构师，他在 Detica'的全球金融市场集 团工作。Simon专心参

与的项目有桌面客户端和Web应用，也有高度可伸缩的分布式系统和面向服务的体系架构（SOA）。他的专攻技术是Java，作为 一名注重实践的权威，他被要求建议并设计解决方案；定义、交付并保证所选择的架构适合于目的，能满足非功能性需求。Simon已经编写或与他人合著过很多 Java EE Web技术相关的书籍，在数次会议上发表过演讲，并创建了 Coding the Architecture——一个介绍关于软件架构实用和务实观点的网站。

特别专题

可伸缩性最佳实践：来自 eBay 的经验

作者 Randy Shoup 译者 郭晓刚

在 eBay，可伸缩性是我们每天奋力抵抗的一大架构压力。我们所做的每一项架构及设计决策，身前身后都能看到它的踪影。当我们面对的是全世界数以亿计的用户，每天的页面浏览量超过 10 亿，系统中的数据量要用皮字节（10^{15} 或 2^{50}）来计算——可伸缩性是生死交关的问题。

在一个可伸缩的架构中，资源的消耗应该随负载线性（或更佳）上升，负载可由用户流量、数据量等测量。如果说性能衡量的是每一工作单元所需的资源消耗，可伸缩性则是衡量当工作单元的数量或尺寸增加时，资源消耗的变化情况。换句话说，可伸缩性是整个价格-性能曲线的形状，而不是曲线上某一点的取值。

可伸缩性有很多侧面——事务的方面、运营的方面、还有开发的方面。我们在改善一个 Web 系统的事务吞吐量的过程中学到了很多经验，本文总结了其中若 干关键的最佳实践。可能很多最佳实践你会觉得似曾相识，也可能有素未谋面的。这些都是开发和运营 eBay 网站的众人的集体经验结晶。

最佳实践 #1：按功能分割

相关的功能部分应该合在一起，不相关的功能部分应该分割开来——不管你把它叫做 SOA、功能分解还是工程秘诀。而且，不相关的功能之间耦合程度越松散，就越能灵活地独立伸缩其中的一部分。

在编码层次，我们无时不刻都在运用这条原则。JAR 文件、包、Bundle 等等，都是用来隔离和抽象功能的机制。

在应用层次，eBay 将不同的功能划分成几个应用程序池。销售功能由一组应用服务器运行，投标功能由另一组负责，搜索又是另外一组服务器。我们把总共约 16,000 台应用服务器分成 220 个池。这样就可以根据某项功能的资源消耗，单独地伸缩其中一个池。我们也因此得以进一步隔离及合理化资源依赖关系 ——比如销售池只需要访问后台资源的一个相对较

小的子集。

在数据库层次，我们也采取同样的做法。eBay 没有无所不包的单一数据库，相反我们有一组数据库主机存放用户数据、一组存放商品数据、一组存放购买数据……总共 1000 个逻辑数据库分布在 400 台物理主机上。同样，这种做法让我们得以单独为某一类数据伸缩其数据库设施。

最佳实践 #2：水平切分

按功能分割对我们的帮助很大，但单凭它还不足以得到完全可伸缩的架构。即使将功能一一解耦，单项功能的资源需求随着时间增长，仍然有可能超出单一系统的能力。我们常常提醒自己，“没有分割就没有伸缩”。在单项功能内部，我们需要能把工作负载分解成许多我们有能力驾驭的小单元，让每个单元都能维持良好 的性能价格比。这就是水平分割出场的时候了。

在应用层次，由于 eBay 将各种交互都设计成无状态的，所以水平分割是轻而易举之事。用标准的负载均衡服务器来路由进入的流量。所有应用服务器都是均等的，而且任何服务器都不会维持事务性的状态，因此负载均衡可以任意选择应用服务器。如果需要更多处理能力，只需要简单地增加新的应用服务器。

数据库层次的问题比较有挑战性，原因是数据天生就是有状态的。我们会按照主要的访问路径对数据作水平分割（或称为“sharding”）。例如用户 数据目前被分割到 20 台主机上，每台主机存放 1/20 的用户。随着用户数量的增长，以及每个用户的数据量增长，我们会增加更多的主机，将用户分散到更多的机器上去。商品数据、购买数据、帐户数据等等也都用同样的方式处理。用例不同，我们分割数据的方案也不同：有些是对主键简单取模（ID 尾数为 1 的放到第一台主机，尾数为二的放到下一台，以此类推），有些是按照 ID 的区间分割（1-1M、1-2M 等等），有些用一个查找表，还有些是综合以上的策略。不过具体 的分割方案如何，总的思想是支持数据分割及重分割的基础设施在可伸缩性上远比不支持的优越。

最佳实践 #3：避免分布式事务

看到这里，你可能在疑惑按功能划分数据和水平划分数据的实践如何满足事务要求。毕竟，几乎任何有意义的操作都要更新一个以上的实体——立即就可以举出用户和商品的例子。正统的广为人知的答案是：建立跨资源的分布式事务，用两段式提交来保证要么所有资源全都更新，要么全都不更新。很不幸，这种悲观方案 的成本很可观。伸缩、性能和响应延迟

都受到协调成本的反面影响，随着依赖的资源数量和客户数量的上升，这些指标都会以几何级数恶化。可用性亦受到限制，因为所有依赖的资源都必须就位。实用主义的答案是，对于不相关的系统，放宽对它们的跨系统事务的保证。

左右逢源是办不到的。保证跨多个系统或分区之间的即时的一致性，通常既无必要，也不现实。Inktomi 的 Eric Brewer 十年前提出的 CAP 公理是这样说的：分布式系统的三项重要指标——一致性（Consistency）、可用性（Availability）和分区耐受性（Partition-tolerance）——在任意时刻，只有两项能同时成立。对于高流量的网站来说，我们必须选择分区耐受性，因为它是实现可伸缩的根本。对于 24x7 运行的网站，选择可用性也是理所当然的。于是只好放弃即时一致性（immediate consistency）。

在 eBay，我们绝对不允许任何形式的客户端或者分布式事务——因此绝不需要两段式提交。在某些经过仔细定义的情形下，我们会将作用于同一个数据库 的若干语句捆绑成单个事务性的操作。而对于绝大部分操作，单条语句是自动提交的。虽然我们故意放宽正统的 ACID 属性，以致不能在所有地方保证即时一致性，但现实的结果是大部分系统在绝大部分时间都是可用的。当然我们也采用了一些技术来帮助系统达到最终的一致性(eventual consistency)：周密调整数据库操作的次序、异步恢复事件，以及数据核对（reconciliation）或者集中决算（settlement batches）。具体选择哪种技术要根据特定用例对一致性的需求来决定。

对于架构师和系统的设计者来说，关键是要明白一致性并非“有”和“没有”的单选题。现实中大多数的用例都不要求即时一致性。正如我们经常根据成本和其他压力因素来权衡可用性的高低，一致性也同样可以量体裁衣，根据特定操作的需要而保证适当程度的一致性。

最佳实践 #4：用异步策略解耦程序

提高可伸缩性的另一项关键措施是积极地采取异步策略。如果组件 A 同步调用组件 B，那么 A 和 B 就是紧密耦合的，而紧耦合的系统其可伸缩性特征是各部分必须共同进退——要伸缩 A 必须同时伸缩 B。同步调用的组件在可用性方面也面临着同样的问题。我们回到最基本的逻辑：如果 A 推出 B，那么非 B 推出非 A。也就 是说，若 B 不可用，则 A 也不可用。如果反过来 A 和 B 的联系是异步的，不管是通过队列、多播消息、批处理还是什么其他手段，它们就可以分别地伸缩。而且，此时 A 和 B 的可用性特征是相互独立的——即使 B 受困或者死掉，A 仍然能够继续前进。

整个基础设施从上到下都应该贯彻这项原则。即使在单个组件内部也可通过 SEDA（分阶段的事件驱动架构，Staged Event-Driven Architecture）等技术实现异步性，同时保持一个易于

理解的编程模型。组件之间也遵守同样的原则——尽可能避免同步带来的耦合。在多数情况下， 两个组件在任何事件中都不会有直接的业务联系。在所有的层次，把过程分解为阶段（stages or phases），然后将它们异步地连接起来，这是伸缩的关键。

最佳实践 #5：将过程转变为异步的流

用异步的原则解耦程序，尽可能将过程变为异步的。对于要求快速响应的系统，这样做可以从根本上减少请求者所经历的响应延迟。对于网站或者交易系统，牺牲数据或执行的延迟时间（完成全部工作的实践）来换取用户的延迟时间（用户得到响应的时间）是值得的。活动跟踪、单据开付、决算和报表等处理过程显然都 应该属于后台活动。主要用例过程中常常有很多步骤可以进一部分解成异步运行。任何可以晚点再做的事情都应该晚点再做。

还有一个同等重要的方面认识到的人不多：异步性可以从根本上降低基础设施的成本。同步地执行操作迫使你必须按照负载的峰值来配备基础设施——即使在任务最重的那一天里任务最重的那一秒，设施也必须有能力立即完成处理。而将昂贵的处理过程转变为异步的流，基础设施就不需要按照峰值来配备，只需要满足平均负载。而且也不需要立即处理所有的请求，异步队列可以将处理任务分摊到较长的时间里，因而起到削峰的作用。系统的负载变化越大，曲线越多尖峰，就越能从异步处理中得益。

最佳实践 #6：虚拟化所有层次

虚拟化和抽象化无所不在，计算机科学里有一句老话：所有问题都可以通过增加一个间接层次来解决。操作系统是对硬件的抽象，而许多现代语言所用的虚拟机又是对操作系统的抽象。对象-关系映射层抽象了数据库。负载均衡器和虚拟 IP 抽象了网络终端。当我们通过分割数据和程序来提高基础设施的可伸缩性，为各种分割增加额外的虚拟层次就成为重中之重。

在 eBay，我们虚拟化了数据库。应用与逻辑数据库交互，逻辑数据库再按照配置映射到某个特定的物理机器和数据库实例。应用也抽象于执行数据分割的路由逻辑，路由逻辑会把特定的记录（如用户 XYZ）分配到指定的分区。这两类抽象都是在我们自己开发的 O/R 层上实现的。这样虚拟化之后，我们的运营团队可以按需要在物理主机群上重新分配逻辑主机——分离、合并、移动——而完全不需要接触应用程序代码。

搜索引擎同样是虚拟化的。为了得到搜索结果，一个聚合器组件会在多个分区上执行并行的查询，但这个高度分割的搜索网格在客户看来只是单一的逻辑索引。

以上种种措施并不只是为了程序员的方便，运营上的灵活性也是一大动机。硬件和软件系统

都会故障，请求需要重新路由。组件、机器、分区都会不时增减、移动。明智地运用虚拟化，可使高层的设施对以上变化难得糊涂，你也就有了腾挪的余地。虚拟化使基础设施的伸缩成为可能，因为它使伸缩变成可管理的。

最佳实践 #7：适当地使用缓存

最后要适当地使用缓存。这里给出的建议不一定普遍适用，因为缓存是否高效极大地依赖于用例的细节。说到底，要在存储约束、对可用性的需求、对陈旧数据的容忍程度等条件下最大化缓存的命中率，这才是一个高效的缓存系统的最终目标。经验证明，要平衡众多因素是极其困难的，即使暂时达到目标，情况也极可能随着时间而改变。

最适合缓存的是很少改变、以读为主的数据——比如元数据、配置信息和静态数据。在 eBay，我们积极地缓存这种类型的数据，并且结合使用“推”和“拉”两种方法保持系统在一定程度上的更新同步。减少对相同数据的重复请求能达到非常显著的效果。频繁变更、读写兼有的数据很难有效地缓存。在 eBay，我们大多有意识地回避这样的难题。我们一直不对请求间短暂存在的会话数据作任何缓存。也不在应用层缓存共享的业务对象，比如商品和用户数据。我们有意地牺牲 缓存这些数据的潜在利益，换取可用性和正确性。在此必须指出，其他网站采取了不同的途径，作了不同的取舍，也同样取得了成功。

好东西也会过犹不及。为缓存分配的内存越多，能用来服务单个请求的内存就越少。应用层常常有内存不足的压力，因此这是非常现实的权衡。更重要的一点，当你开始依赖于缓存，那么主要系统就只需要满足缓存未命中时的处理要求，自然而然你就会想到可以削减主要系统。但当你这样做之后，系统就完全离不开缓存了。现在主要系统没办法直接应付全部流量，也就是说网站的可用性取决于缓存能否 100%正常运行——潜在的危局。哪怕是例行的操作，比如重新配置缓存资源、把缓存移动到别的机器、冷启动缓存服务器，都有可能引发严重的问题。

做得好，缓存系统能让可伸缩性的曲线向下弯曲，也就是比线性增长还要好——后续请求从缓存中取数据比从主存储取数据成本低廉。反过来，缓存做得不好 会引入相当多额外的经常耗费，也会妨碍到可用性。我还没见过哪个系统没机会让缓存大展拳脚的，关键是要根据具体情况找到适当缓存策略。

总结

可伸缩性有时候被叫做“非功能性需求”，言下之意是它与功能无关，也就比较不重要。这

么说简直错到了极点。我的观点是，可伸缩性是功能的先决条件——优先级为 0 的需求，比一切需求的优先级都高。

希望以上最佳实践能对你有用，希望能帮助你从新的角度审视你的系统，无论其规模如何。

参考

- eBay's Architectural Principles (video)
- Werner Vogels on scalability
- Dan Pritchett on You Scaled Your What?
- The Coming of the Shard
- Trading Consistency for Availability in Distributed Architectures
- Eric Brewer on the CAP Theorem
- SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

关于作者

Randy Shoup 是 eBay 的杰出架构师。从 2004 年起担任 eBay 搜索基础设施的主要架构师。在加入 eBay 之前，他是 Tumbleweed Communications 公司的总架构师，也曾在 Oracle 和 Informatica 担任多个软件开发和架构的职位。

他经常在业界的会议上讲授可伸缩性和架构模式。

特别专题

可伸缩性的最差实践

作者 徐磊

引言

在扩展大量大型的分布式系统期间，我有机会观察（并实践）了一些最差实践。这些最差实践中的大部分在开始时都没有危害，但如果疏忽大意，它们就会对系统的发展和可伸缩性构成危害。很多文章都聚焦于最佳实践，以确保拥有一个易于维护和可伸缩的系统，但在本文中，我主要强调的则是一些应该规避的最差实践。

技术

没有任何一种技术或架构能实现所有的需求。了解何时该反思现有的方法、如何拓宽视野以超越局部范围、或如何进行依赖的有效控制，这些都是可伸缩性的关键特性。让我们进一步分别研究一下。

金锤子

金锤子起源于一条古老的谚语：如果你只有一把锤子，那么任何东西在你眼里都是一枚钉子。很多开发人员都局限在仅使用一种技术的观念中——其代价是不得不使用选定的技术来构建和维护基础设施，即便已经存在另一种技术更适用于特定问题域的功能和抽象。强行把一种技术用在它所不擅长的方面，有时会适得其反。

举例来说，持久化键—值对问题的常见解决方案是使用数据库。之所以常常这样选择是因为组织或开发者有坚实的数据库实践，针对许多问题自然而然就会沿用同样的解决途径。当数据库的特性（关系完整性、锁、连接和方案）成为瓶颈或阻碍了其伸缩扩展时，问题也就出现了。这是因为应用基于数据库的解决方案要发展，其成本通常要比使用其它可用技术更为昂贵。随着键—值存储访问率的增加，数据库并发模式的性能就开始降低，而数据库具备的高级特性却被闲置。许多传统关系数据库的替代方案都是针对这些缺点的，比如

CouchDB、SimpleDB 或 BigTable。

另一个常见的“锤子”就是总利用线程来进行并发编程。尽管线程确实是针对并发的，但它们也带来了成本，这些成本包括代码复杂性的增加、以及由于目前线程的的锁定和访问模型造成的组件编排（composability ）方面的固有不足。由于如今最流行的编程语言都使用线程处理并发，因此数千行代码都含有竞态条件、潜在的死锁和不一致的数据访问管理。有些正在成长的社区提出了另一些并发方案，这些方案不存在线程的可伸缩性问题，也就是由 Erlang 或 Stackless Python 提倡的并发模型。即便不在实际生产中选择那些语言，研究一下它们的概念（比如消息传递或异步 I/O）仍然是一种不错的实践。

资源滥用

小范围的问题开发者们一般都能处理得得心应手：使用分析工具、了解算法的空间和时间复杂度、或者了解哪种场合应该用哪种列表实现。但并非每个人都善于认识到大型系统的约束条件，比如识别共享资源的性能要求、了解服务的各种客户、或发掘数据库的访问模式。

应用程序实现伸缩性的普遍方法是不断横向部署冗余的、无状态的、彼此不共享内容的服务，以此作为最理想的体系架构。但以我的经验看来，这种扩展往往会忽视新增服务对共享资源的影响。

比如说，如果一个特定的服务使用数据库作为持久存储，它通常通过一个线程池来管理数据库连接。使用池是不错的方法，有助于避免进行过多的数据库连接处理。然而数据库仍然是共享资源，除了单个池配置，还必须对所有池从总体上进行管理。下面两个实践就会导致失败：

- 持续增加服务数，但并不减小池的最大数。
- 增大单个池的大小，而不减小服务数量。

以上两种情况中，除了按性能要求配置应用之外，连接的总数也必须加以管理。此外，还要持续监控数据库的容量，以保持连接均衡。

处理共享资源的可用性至关重要，准确的说，这是因为它们一旦失效，由于其“共享”的本质，失效会对系统造成全面的影响，而非孤立存在。

大泥球

依赖是很多系统讨厌却又必不可少的东西，不积极地处理好依赖及其版本会损害灵活性和可

伸缩性。

代码的依赖管理有多种不同的模式：

- 同时编译整个代码集
- 基于已知版本选取构件和服务
- 发布的模型和服务所有变更都向后兼容

让我们看看这些情形。首先，在大泥球模式下，整个系统作为一个单元编译和部署。这种模式拥有明显的优势，也就是将依赖管理交给编译器处理，并能提前捕获一些问题，但它会因每次都部署整个系统（包括测试、交付和大范围变化引起的风险）而引发可伸缩性的问题。在这种模式下，会更难隔离系统的变化。

在第二种模式中，依赖都是按需挑选的，但是变化经过依赖传递之后依旧出现第一种模式一样的难题。

第三种模式中，服务负责依赖的版本化，并向客户端提供向后兼容的接口。这明显减轻了客户端的负担，从而允许逐步升级到新的模型和服务接口。此外，当数据需要转换的时候，它是依靠服务而不是客户端完成的——这进一步稳固了隔离性。向后兼容的变更意味着打补丁、升级和回滚都不能干扰客户端操作。

采用变更能向后兼容的服务体系架构在最大程度上避免了依赖问题。它同时方便了在受控环境下进行独立测试，隔离了客户端和版本化数据的变化。这三个优点对隔离变化来说都很重要。最近发布的 Google Protocol Buffers 项目也在倡导向后兼容的服务模型和接口。

全部打包还是部分打包

处理依赖时要考虑的另一件事情是如何对应用内容打包。

在一些场景中，比如 Amazon Machine Images 或 Google AppEngine 应用，它们的整个应用和所有的依赖都一起打包发布。这种囊括一切的打包方法保持了应用的自包含，但它增加了包的总大小，而且应用中任何地方的一个小小改变，都会迫使系统重新部署整个应用包（甚至对同一台物理机器上许多应用使用的共享库也是如此）。

替代方案是将应用的依赖移出主机系统，令应用包只包含依赖图的若干部分。这控制了包的大小，但由于应用在能提供服务之前需要将特定的组件传递到每台机器上，所以增加了部署配置。依赖项目没有立即准备好、机器没有经常测试、抑或是依赖错误，由于以上种种，不将整个包部署为自包含的方式会制约将应用部署到异构的、非标准化的机器上。

后一种方案——分成不同范围（全局的、机器的、应用的）去处理依赖——必然会增加疏漏和复杂性。它减少了配置和依赖隔离，增加了操作的复杂性。一般而言，隔离能增加可伸缩性，所以尽可能选用囊括一切的方法，除非有例外情况。

无论在代码还是在依赖处理中，最差实践就是不清楚模块间的关系，没有规划好模块以便于对其进行管理。未能增强控制是可伸缩性的一大绊脚石。

忘记检查时间

在分布式系统中，通常的目标是尽可能地将开发者和负责分布式调用的复杂方法隔离开来。这使主要的开发工作集中于核心的业务逻辑上，而不用担心失效恢复、超时以及其它分布式系统必需的需求。但是，让远程调用看起来像本地调用一样就意味着开发者要像本地调用一样编码。

我常发现很多代码都期望所有的远程请求能及时完成，但这样的期望是不合理的。比如说，Java 在 JDK1.5 中仅为 HTTPURLConnection 类引入了读超时，而让开发者要么创建线程去杀死进程，要么天真地等待响应。

Java 中，另一个潜在的时间处理不合理的例子是 DNS 查找。在一个长时间运行的典型系统中，执行完最初的 DNS 查找之后，如果不进行明确的配置，结果会缓存在 JVM 的生命期内。如果外部系统更改了主机的 IP 地址，将不能正确处理该条目，而且在很多情况下，因为编程时没有设置连接超时时间，连接就会被挂起。

为了对系统进行合适的伸缩扩展，为请求处理分配好时间是极其重要的。有很多方法可以实现，有一些是语言内置的（像 Erlang），其它的则作为库的形式提供，比如 libevent 或 Java 的 NIO。抛开实现语言或架构不谈，正确地管理操作等待时间是非常必要的。

运行时

建立一个符合成本效益的可扩展方案、处理好依赖、预先考虑到失效都是创建优秀架构的各方面要求。而在生产环境中，系统易于部署和运维的能力也同等重要。这里同样有很多不利于系统可伸缩性的最差实践。

英雄模式

运维问题普遍的解决方案是有一个“英雄”（关键性人物），他能处理、并经常处理大部分的

操作需求。在小规模环境中，当某个人有天赋和能力熟悉整个系统（包括保持系统正常运行的许多细节之处），英雄模式可以正常运行。尽管这是最常见的实施方案之一，但对拥有许多组件的大型系统而言，这种方法就不能进行伸缩扩展了。

在没有形式说明的情况下，“英雄”往往要理解服务依赖，牢记如何开、关特性，或了解其他人已经遗忘了的系统。“英雄”虽然至关重要，但他不应该是一个个体。

我认为英雄模式最好的解决方案是自动化。如果组织的情况允许，让个人在团队之间轮换也有帮助。在银行里，休假有时是强制性的，好让“你这里不行，要到我的机器上做”之类的问题及时暴露出来。

非自动化

系统过度依赖于人工干预往往是存在“英雄”的后果，这面临着可重复生产能力的问题和“英雄”出现意外情况带来的问题。能重现特定的构建、部署和环境很重要，而明确定义的元数据控制下的自动化是实现可重复能力的成功关键。

在一些开源项目中，工件的发布过程依赖于个体开发者在自己工作站上构建工件，没有任何措施保证产生出来的工件版本能实际对应到源码控制系统中的某个分支。在这些情况下，完全有可能发布软件，其代码从未被提交到源码控制系统。

综上所述，“英雄”的活动应该由自动化取代，从而确保个人（或许多人）可以相对容易地替换其他人。自动化的替代方案是增加流程——Clay Shirky 为流程给出了一个有趣的定义：

> 流程是对先前蠢行的内在反应。

先前的蠢行在所难免——自动化应该吸取教训。

监控

当时间紧迫时，监控（比如测试）往往是第一个牺牲的环节。有时，在我问及有关组件的运行时表现方面的细节问题时，总没有答案。缺乏对运行系统内部的深入了解和迅速切入问题的能力，不利于对从哪里入手和着手做什么做出正确攸关的决策。

Orbitz 很幸运地拥有久经考验的监控软件，它们既能提供服务调用的细粒度详细信息，也能

精确显现出问题域的数据。来自监控基础设施的可用度量数据有利于快速有效地解决问题。

总结

在不久前 Amazon 的 S3 出现服务中断之后，Jeff Bezos 说道：

> 遇到问题的时候，我们知道直接原因，我们从那里入手分析并找到了根本原因，然后从根本上进行了修复，又向前迈进了一步。

软件和系统的开发是一个迭代的过程，在这个过程中，失败和成功的机会并存。简单但较难伸缩的解决方案有其一席之地，特别是计划或应用尚处于不成熟的阶段。“好”和“完美”不是对立的。但随着系统的日臻完善，应该除去其中的那些最差实践，这样，成功也就是理所当然的了。

非常感谢 Monika Szymanski 对本文初稿提出的建议。

关于作者

Brian Zimmer 是旅游业新创企业 Yapta 的架构师，是一位受人尊敬的开源社区成员，也是 Python 软件基金会的成员。他之前作为高级架构师服务于 Orbitz。他的博客在 http://bzimmer.ziclix.com。

特别专题

构建的可伸缩性和达到的性能：一个虚拟座谈会

作者 James Cox 译者 宋玮

实现伸缩性和性能调优的经验所保有的价值容易被低估。两者都是“晚些时候”或“我们真正流行起来时”所面临的难题。早期创业公司确实不应该立即花这份钱， 而大公司通常不能做出足够快速的反应以实现所需的改变。再加上这需要一个多学科的团队，它立刻就变成了一个需要解决的政治上和工程上的难题。

但是它从来不会从我们的视线中消失——在过去的少数几个 Qcon 会议上，“Architectures You've Always Wondered About（你始终感到惊讶的架构）”专题被淹没了，正因为如此，我们渴望学习“大人物”们如何去做这些事情的技巧和诀窍。

在这次 InfoQ.com 的虚拟座谈会上，我们从几个最大的公司和项目中邀请了几个在实现可伸缩性和性能方面的专家，让我们进入他们取得了成果而我们却依然梦想的神秘世界吧。

座谈会的参与者包括有 Blaine Cook，Twitter 的前任架构师，他领导开发者进行了 Starling 消息队列 ruby gem 的开发。Blaine 在“如何让 Ruby 和 Rails 伸缩自如”的会谈中发出了积极的声音，在这一领域有非常多的经验。

他是被 Randy Shoup 加入的，Randy Shoup 是 eBay 市场架构小组的一个分布式架构师。从 2004 年起，他一直是 eBay 搜索基础架构的主要架构师。而 Matt Youill，则是 Betfair 的首席技术专家。在超过 6 年的时间里，他引领了许多 Betfair 的技术架构。最近他又与 Betfair 的高级技术小组转向一个更长期的高级项目——包括了 Flywheel 技术及其他技术的策略、研究和特殊项目。

为了让与会者来源更多样，我们请专业开发 Rails 性能工具的 FiveRuns 也参加进来。他们派了整个工程团队用了一个下午来商讨其答案——简直可以另外开一个座谈会了。

问题 **1**：许多人把性能和技术混为一谈。你们怎么回应这种误解呢？

Randy：我同意独立于任何特定语言或框架来讨论伸缩性问题确实很有价值——不管实现策略如何，模式都是一样的。可是，首先让我们确保自己已经把性能和可伸缩性区分开了。性能是服务于一个单独请求时的资源使用问题。可伸缩性是当要服务更多（或更大）请求时资源消费量如 何随之增长的问题。

它们是相关的，但是不是同一件事情:-)。幸运的是，许多改善其中一个问题的方法通常也会改善另一个问题。但是，速度非常快的系统可能并不是最具伸缩性的系统，反之亦然。

FiveRuns：我们关心性能、可伸缩性、可用性的对比。我们把性能定义为一个操作（或一些操作）如何快速地完成，比如，响应时间、每秒事件处理的数量等等；而可伸缩性——就是应用怎样很好地按比例扩大规模以应对更大的使用需求（比如，用户数、请求速度、数据容量）。

问题 **2**：当你碰到性能瓶颈时，你如何开始调查是什么导致了这一问题？

Matt：每个问题都是不同的，但是我想这通常是一个搜集观测资料（即，与许多人进行交谈）并缩小问题范围的过程——通过检查什么仍在继续执行，你可以分离出不再执行的东西。

有两个重要的事情要谨记在心。确保它确实是问题的瓶颈所在。例如，忧虑处理器使用效率低下的问题与一个受 IO 束缚的应用可能没什么关系。从产品环境去进行观测非常重要，不要在阶段环境或开发环境亦或是其他环境上进行观测。试图在“假”的环境里再现问题并收集信息将只能产生“ 假”的结果。

Blaine：直觉（快速但不可靠，而且容易产生错误）和监测（需要预先的工作，但是这能让你随心所欲地分析问题）。人们经常谈论测试驱动开发，但是可伸缩性需要测量驱动（metric-driven）开发。

像 JMeter 和 Tsung 这样的工具，可以让你模拟负载，但是接受任何事情都要有所保留。如果你有很长的时间(到底多长时间取决于你的流程和构建测试的人)去构建非常周到的测试，你可以获得更好的一手资料。但是，不可能做到和真正的流量一样逼真。

Randy：通常我们会从注意到问题的地方开始，然后向前追溯。在应用栈的所有层次所进行的监测/勘测得越多，就越容易这样做。它还可以帮助有各种学科专长团队里的人们来检查这个问题。

问题 **3**：当处理 **Web** 应用的问题时，你发现什么工具最有用？

FiveRuns：在浏览器层，firebug 是我们所选的工具，因为它提供了对用户执行单页面的感觉的一些深入观察。我们还喜欢 ab & httperf，它给我们提供了一些自动化的负载测试，因此

对比于我们从 firebug 中获得页面级的计时，使用这个工具可以获得会话级的计时。

Matt：我们混合使用成品工具和定制工具。成品工具是非常好的，但是成品意味着它们是通用的。它们没有你应用程序中 非常重要的特定标准的知识。例如，在 Betfair 的 Web 应用上下文里，这可能与请求频度和含有相应业务事件（就像足球比赛中的射门、进球）的特殊页面 类型相关。通用工具没有"足球比赛"选项。

Blaine：Ganglia 是非常优秀的。同时 Nagios 或 Zabbix（举个例子）将告诉你何时资料遭到破坏，使用少量加工你就能够让 ganglia 给你提供任何东西。对于 MySQL，Innotop + slow query log 帮了大忙。使用 mysql 微妙级日志补丁来察看所有查询中哪些查询的时间在 1 毫秒到 1 秒之间。

Randy：我非常同意使用日志——我们可以支配的最有用的工具是我们的监测框架，我们的应用服务器使用它来记录我们所做的各个操作的调用/事务发生时间——整个 URL 的执行情况、嵌套的数据库访问、嵌套的服务调用等等。这就让我们能够远程诊断大多数产品问题，而不需要 在任何地方都配备调试器。我们还保留了这些日志的可查询历史，我们可以用之比较以前和当前的性能，以及随时间推移的趋势。

问题 **4**：当你有一个堆栈（或核心）问题需要诊断时，什么工具可以提供帮助？

Randy：GDB 和 DTrace 是用于 C++的基础架构。core 或 pstack 是个颇有价值的工具。对于 Java，我们使用了 tracing 和各种 Java 调试器，它们组成了 eBay 基础架构的主要部分。可是，调试器不应当是你用的第一个工具；它应该是你用的最后一个工具。产品代码需要用足够多的仪器来远程诊断问题。

Matt：我们有许多魔法！我们使用各种工具来重现问题并调试它们（包括栈的问题）——Visual Studio、 Eclipse、WinDbg、cdb、Purify、Fortify、dtrace 以及许多定制的东西，为我们的架构所构建的东西。

问题 **5**：你认为像 **Yahoo!**性能前十名等等这样的文章有用吗？领域或应用的可伸缩是个问题吗？

Blaine：这两者的答案都是肯定。从某点上讲，伸缩性已经从领域问题（即，如果你不使用内存缓存或者一个等价的分布式哈希表和基于内存的缓存）转移了，而你仍然处于"领域"范围。一些应用充分地理解了这一点。当今静态内容的可伸缩性已不那么重要了，那只是花钱的问题并需要公司有好的社会组织的问题。

FiveRuns：我们认为像 Yahoo!性能文章这样的资源是有用的。我们认为把应用架构与问题领

域进行匹配是非常关键的——实际上，我们对问题领域的所有假设都定义了软件系统质量方面的约束。

换句话说，我们都期望问题领域可以定义针对性能、伸缩性、可用性等等的预期。当然，需要在这些（或其它）质量之间进行权衡，比如，功能性和上市时间，它们将能显著地改变问题领域。

Matt：当然，所有有用的素材都在这些文章中了。虽说这样，完成一个高质量（性能）系统是相对的——我从未看到关于如何维护品质一个好的建议——这是治理部分的内容。看上去 IT 系统似乎注定一开始很鲜亮、然后衰退、最终消亡，接着再全部重新构建一遍。连确保它们至少偶尔有一个漂亮的外观都很难。

问题 **6**：实现可伸缩性和性能调优通常被看作是一种“救火”措施；目标是是立刻修正问题。你们在一个成熟的代码库上是如何跟踪性能衰退的？

Randy：在 eBay，所有项目都经过了严格的负载和性能测试阶段。在将项目发布到网站之前，我们使用这一阶段来检测并解决潜在的性能问题。

Matt：我们经过的是一个不同的过程——我认为这些都是在应用程序运行的时候才能检测到的。要确保一个应用被有效地划分并部署到真实环境的服务器上。如果一切正常，再把它部署到另一个服务器上，然后再是另一个，等等。确保在早期初次公开展示过程中，你投入了有效的监测和管理基础架构以捕获性能问题。

不要试图在部署之前就捕获性能问题。你不可能重建真实环境中的条件，因此你不可能得到真实可靠的测量结果。

Blaine：是的，监测。非常仔细的监测。我提到的是监测吗？监视迟缓现象。当你看到一个衰退时，回去检查一下你的改变记录。应该很容易定位，因为你通常所做的是增量部署，不是吗？

FiveRuns：我们得补充一下，适当地建立一个基线是很重要的：一个已知的好的性能表现。并且试图梳理出任何交织于最终用户功能瑕疵的性能问题——有时这些瑕疵看起来只和功能有关，但实际上也包含了性能问题。

问题 **7**：那么捕获问题的真正重要的衡量标准是什么？

Randy：衡量标准就是那些影响你业务（例如，页面负担、用户响应时间）的衡量标准，以及使你受限（例如，内存、CPU、网络带宽等）的衡量标准。正如你可能想象的，eBay 基础架构的一些部分是受 CPU 限制的，另一些是受内存限制的，还有一些则是受 IO 限制的。当

然，墨菲法则（一种幽默的规则，它认为任何可能出错的事终将出错）确保了你没有严密跟踪的衡量标准就是那个对你不利的标准！

Blaine：取决于你的应用。任何事都有响应时间。找到你认为花了最长时间的地方，对它们进行计时，如果结果随你所测量的比特位数的多少而非常不同，找到它们。页面装载时间在高于 250ms 时都是很重要的问题。在这个响应时间之内，用户（通常）都看不出差别，而任何优化（例如，针对 API）都是性能和成本的优化。

了解集群的当前状态；如果是超出了集群的处理能力，那就去增加处理能力，句号。没什么好说的，去买更多的机器就是了。不要再阅读本文了，去买更多的机器吧。如果处理能力是一个影响因素，而且你的应用并没有大的问题（无明显性能问题，你已经调优了 mysql 等等），去买机器吧。你可以让它运行得更快并节约了后面的花费，但同时，购买机器意味着你将减轻很多压力并给自己一个喘息机会以对应用正确实现可伸缩性。

Matt：确保它与你的业务相关，确保你所采取的测量是有意义的。除非你知道当时正在执行什么业务功能，否则一个 CPU 测量是无意义的。

FiveRuns：我们全体都同意最终的衡量标准是捕获了客户期望响应的那一个。通过响应，我们真正把整个系统中的其它元素汇集到了一起。例如，gmail 比其它邮件服务递送邮件更加迅速；因为它的“响应”（例如，从邮件被发送到在 gmail 中看到邮件之间所花的时间）满足了用户的预期，其它技术上的标准是无关紧要的。

问题 **8**：性能对应用程序的总体“健康”能起什么作用？

Matt：性能是一个重要的部分，但仅仅是部分。虽然旧了一点，但 ISO 9126 依然是建立有关测量系统的衡量标准的很好入手点。

“健康”是个很有意思的概念。在医学里，身体里每个细胞的性能不能够被测量，但是大量细胞汇集在一起就有了“生命信号”，如脉搏、温度、血压和呼吸都是可以测量的。大系统应该以类似的方法进行测量——单个服务器的内存使用与整个系统在线用户数相比起来没什么用处。每个大系统需要一些关键的生命信号。

Randy：性能绝对是基础。差的性能直接反映了底层基础架构的花销，以及用户的体验。这两者都是底线。一个性能低于预期的应用不是“完成”了的应用，一个缺乏必须功能的应用同样是没有完成的。

FiveRuns：通常，我们都同意把性能不佳看做是一个大 bug，而当作一个大 bug，我们还得关注存在大 bug 的地方，这可能是另外一个大 bug。因此，我们认为健康是依赖于用户/所

有者体验的一种东西。

可是，也有反例。例如，针对很小一部分受众的一个意在提供一些新颖功能的 Web 应用，其可伸缩性问题不被计入到应用的“健康”范畴。如果是一个工业级应用，比如 gmail，它一向有性能（或伸缩性或可用性）问题，那么其可伸缩性就给应用“健康”的带来了消极影响。

Blaine：有时一个应用就是慢，因为它要处理很复杂的事情，加快速度是不可能的。这变成了一个交互和负载容量问题。有时应用慢是因为代码太烂，当负载增加时所有事情都慢下来了（尤其与触发锁的代码有关，例如，sql 锁或 nfs 所或任何种类的分布式锁）。

问题 **9**：真正的性能和感知的性能之间的存在概念差异。对修复来说哪个更重要？

FiveRuns：感知的性能是最终用户感觉得到的，因此最应该修正。可是我认为感知的性能是很难测量的——即，它是依赖于上下文的，作为观察者，我们不能总是访问同一个上下文环境，而且我们不可能控制组成上下文的所有东西。

Blaine：掩饰。感知的性能更重要。使用后台队列卸载工作，显示用户所期望的东西并且断定是怎么泄露到应用的其它方面的。使用 Javascript 轮询廉价的请求而不是集中到一个单一的昂贵请求。在请求开始的时候将开销大的事情拆成并行的，在请求结束的时候再把结果收集起来。

Matt：我不同意——真实更重要。与其假装成某种情况，不如真的变成那种情况。

Randy：这两者都重要:-)。感知的性能影响了用户体验，而所谓的真实性能会影响成本。这最终就是一个业务决策：此刻最大的痛苦是什么，因此要看哪个相对更优先。

问题 **10**：几乎每个星期，总有些人发布一个新的 **Web** 或应用服务器或数据库**/ORM** 层软件。怎么看待这种情况？这是好事情——还是让 **Job** 调优更加困难了？

Blaine：我使用 apache。它有其缺点，但是工作良好。我使用 MySQL，它也有其缺点，但是工作良好。没有应用可以解决你所有的问题；它们有些是让事情更容易理解或修正，有些的确更好，但是有些使得调试和修正更加困难。

至于数据库，拥有 DHT 并不能使你获得索引（index）。你需要一个分布式索引，只是你还不知道。BigTable 和相关开源项目都不是分布式的索引。MapReduce 不是分布式的索引。查明你的分布式索引是怎样的，自己构建它吧。

Matt：这很好，但是可以更好。大多数同类产品都是大同小异的。能看到持续创新挺好，但

是如果能看到每一类工具的数量少一些而完全不同类型工具的数量和差异多一些，那就更好了。

FiveRuns：作为这种产品的制造者，我们对这个问题特别感兴趣！我们认为它是好事——它鼓励创新并给我们带来新的思路。但是我们专注于使用最好的工具组合——新的不见得就是好的。

Randy：有选择是好事情。这取决于你怎么去选择。追赶潮流此刻不再比顽固坚持不能工作的东西更有生产效率。我要说的是在一个像 eBay 这样 24x7 都要运转的站点，我们在使用经证明是好的技术时需要非常小心。这是我们对公司的责任。在我们考虑引入任何新东西的时候，在我们考虑将其应用到站点之前都要经过非常广泛的评估。

问题 **11**：那么在这一领域还有哪些内容经常被曲解？

Matt：嗯，我觉得让人有点失望的是，当一个好思想被商业化或者更通用化时，这个思想的质量就变差了。比如数据存储——很难建立一个既快速又可存储大量数据的软件。很明显，为了商业化（即，有大量潜在客户）两者都可实现当然最好了，但是这就意味着最终是一个妥协的产物，既不快也不大——不管厂商怎么宣传！

不是说这就是普遍现象或者总会是这样。

Blaine：你只能通过使用软件实现伸缩性。“语言是不能伸缩的。框架是不能伸缩的。而架构是可以伸缩的。”

伸缩性是一个社会问题。协调你的业务、操作和工程师们是极其关键的，如果你在这方面不成功，你实现伸缩性的努力也会失败。如果你的公司不理解需要多方协调以培育成功，那么你也会失败。YouTube 正是通过正确的处理这一社会问题实现了伸缩性。他们购买服务器、有一个好的投资计划、有足够的带宽。他们选择一件事情，把它做好，以组织的方式决策伸缩性，在天才工程师们的努力下，伸缩性如魔术般被实现了。

你可能会失败。快速吸取教训，然后成功。没问题。不要构建开箱即可伸缩的服务，因为你或许会发现你的交互设计是错误的，你必须回过头重新来一遍。在这期间，那些没有构建伸缩性的人可能正确设计了交互，因而足足领先了你 6 个月的时间。

FiveRuns：性能改进/伸缩性提升是一件“容易”的事情——修正性能或伸缩性问题不总是像外界看得那么简单——很容易因此受到指责且很难真正解决“所有”给定的约束。以我们的经验，修正性能和伸缩性问题需要对技术和问题领域两方面都有深入了解和经验。

Randy：经常被曲解的是性能和可伸缩性这两者间的区别 ;-)

Microsoft Visual Studio

Microsoft

/*CODING完美世界*/

Code，创造了万事万物。几乎任何地方，均有它的存在，正如作为开发人员的你，眼中所见的无限可能。利用Visual Studio 2010的新工具，你设计、开发或是部署的工作方式，都将发生意想不到的变化；你心中的伟大构想，终将以前所未有的方式，付诸实现。

/*CODE，无处不在*/
用Visual Studio 2010打造你的奇迹

立即登录VisualStudio.com
或拨打咨询电话: 800-820-3800 / 021-96081318

请用手机拍摄

拍摄此标签，获取Visual Studio 2010最新消息
如手机未安装二维码应用程序，请至http://gettag.mobi免费获取 *信息及数据资费按相关标准收取

©2010 Microsoft Corporation保留所有权利。Microsoft, Visual Studio及Visual Studio徽标是Microsoft Corporation在美国和/或其他国家（或地区）的注册商标或商标。

推荐文章 | Articles

动态语言企业应用优缺点浅析

作者 李明（nasi）

动态语言的兴起已经有些年头了。现在，人们早已不再去争论动态语言是否能够取代静态语言，因为这种争论毫无意义。越来越多的开发者开始在动态语言更为擅长的领域应用它们。比如，Django 和 Ruby on Rails 等开发框架的盛行使得像 Python 和 Ruby 这样的动态语言可以在 Web 开发领域大放异彩，PHP 和 JavaScript 也早已在 Web 开发领域占有一席之地。

不过目前动态语言在企业开发中的应用还不够广泛，很多企业只是用它来做一些粘合系统的工作，并没有承担起主力开发语言的重任。尤其是在底层系统开发方面， 动态语言远没有在 Web 开发方面那么风光。在运行时效率和虚拟机稳定性方面的不足，使得动态语言注定无法与编译型语言竞争，并取代它们在高性能领域的地位。然而，动态语言也有自己的优势所在。如何克服自己的劣势，将优势发扬光大，便是每一位动态语言开发者所面临的机遇和挑战。

我所在的团队用了近两年的时间，将一个电信领域的公司绝大部分的生产系统用动态语言（主要是 Python）重写。包括短/彩信消息网关、业务订阅服务、座 席查询系统、销售支撑系统，乃至搜索引擎等多个核心系统，都在重写之列。重写的理由很多，一方面原有系统无论是从性能上，还是从应对需求变化的能力上，都已经不能满足业务发展的需要；另外一方面，动态语言的诸多优势，也是我们重写的动力。这里仅以开发这些系统时获得的经验，来谈谈动态语言在应用时的优缺点。

动态语言的优势

动态语言的优势有很多，归纳起来主要有以下几个方面：

1、 生产力。动态语言在开发效率方面有着无与伦比的优势，这也与动态语言“优化人的时间而不是机器的时间”这个理念相吻合。利用传统的静态语言要开发几周的功能和特性，使用动态语言也许几天甚至几个小时就可以实现。不仅如此，动态语言在开发原型系统

和常用工具方面的开发效率也非常高，尤其 值得一提的是原型系统。更快地让原型系统运转起来，不仅可以尽早验证一些假设，也能够更好地与迭代开发相结合，更及时地与需求方进行沟通，帮助需求方挖掘和了解自己真正的需求。开发效率可以说是动态语言最为吸引人的地方，这也被认为是将来开发语言的前进方向。这些年随着敏捷开发的盛行，越来越多的开发者意识到，原来动态语言的特性和敏捷开发的价值观也相当契合：缩短反馈时间，对变化的响应能力更强。所以许多动态语言团队选择了敏捷开发的实践来组织团队。我们在实际开发的过程中，以两周为一个迭代周期，基本上 1-2 个迭代就可以完成一个系统的开发和上线，而原型系统的开发通常能在 1-3 天内完成。

2、 代码量。曾有报道说，用 Ruby on Rails 写同样的项目，代码量大概只有 Java 的 1/10。且先不说这个说法是否有夸张的成分，但就实际来看，动态语言的确从代码量上来说，要比 Java/C/C++等传统静态编译型语言要少的多（当然语言的表达能力与动态静态关系并不大，静态函数式语言的表达能力也很强），可能几千行的项目就算得上是个大项目。例如，我们开发的系统中较为复杂的消息网关，生产代码行数大概在 7000 行左右；而订阅服务系统的生产代码行数不到 3000 行；借助于 Xapian 的 Python-bindings，我们的搜索引擎系统代码行数只有 800 行左右。代码量少的好处非常多：首先，这意味着将来需要花在维护上的代价更低；其次，因为代码少了，犯错误的机会也就少了，因此代码量少还意味着 BUG 的减少；再次，代码量的减少也有助于优化系统，有句话说的好，“There is no code faster than no code”。开源项目 Nebula Device 有一个设计哲学，就是“每一次 Release 要比前一次代码量更少”，窃以为高论也。

3、 测试。因为动态语言很容易实现反射等动态特性（JUnit 也是等到 Java 支持了反射以后才出现的），因此测试也更为容易实现。Python 和 Ruby 的标准库中都带有 unittest 的框架，这几乎可以让你无成本地使用单元测试来加固代码。因为动态语言本身不具有编译过程，因此 犯下某些低级错误的几率大大增加，也为重构带来了重重困难。没有单元测试的重构如同梦魇一般，动态语言尤甚。因此，在开发语言以动态语言为主的开源项目中，单元测试总是占有相当大的比重。还有建议称测试代码与生产代码的比率（Unit Test To Code Ratio）要达到 2:1 以上。另外，动态语言的测试环境更容易搭建，实现 Mock 也更为简单。

4、 原生数据结构。现在主流的动态语言多为脚本语言发展而来，而在这些语言中，集合、列表和词典这样的数据结构都是原生的，而静态语言的数据结构往往是通过程序类库来实现的。比如 Python 就提供了 set、tuple、list 和 dict 等原生数据结构，同时还提供了大量操作（如数组分片等），让这些数据结构使用起来非常方便。原生数据结构使得对数据的操控融入到了语言的语法当中，让程序更为易读，这也让基于代码的沟通更为顺畅。

5、 简单易学。动态语言的语法相对简单，学习成本看似也比较低。有人举例说，Python 和 Ruby 写个 Hello World 只需要一行即可，这是很多静态语言所达不到的（把多行代码写成一行的不算）。当然你可以认为这只不过是句玩笑话，不过单就语法而言，动态语言的学习门槛要比很多静态语言要低的多。可是，开发不仅仅只是语法而已。很多动态语言的初学者，能够用动态语言写一些简单的小程序小工具，却很难构建起庞大复杂的商业系统，究其原因，主要是还是因为系统设计和面向对象的功底欠缺所导致的。如何设计，如何抽象，如何重构，这些能力与语言无关，而是个人的修为。正如陆游所言，“功夫在诗外”，这些能力也不是一朝一夕、通过学学语言就能够轻易练就的。当然，动态语言的各种特性（如 Duck Typing）也使得在静态语言中不得不使用的设计模式可以很自然地表达，这些差异也增加了动态语言学习的隐性成本。

不足之处

任何事物都具有两面性，动态语言也不例外，虽然优势显而易见，动态语言的不足之处也有很多。这里列举一些我们在开发过程中所遇到的问题，以及一些初步的解决方案，来供大家参考。

1. 运行效率。运行效率低下使得动态语言饱受诟病。“跑得太慢”这顶帽子已经在动态语言的头上扣了许多年。甚至有 Benchmark 表明，在某些应用场景下，动态语言的运行效率和 C/C++、Java 等成熟的静态语言相比，相差数十倍甚至上百倍，这也为动态语言的普及埋下阴影。不少开发 者因为运行效率的问题，纷纷表示 “对动态语言很失望”。其实我倒是觉得大可不必纠结在这个问题上，原因有两点。第一，很多动态语言的应用场景使得运行效率的重要程度大大降低。就拿 Ruby on Rails 来说，在 Web 开发这个应用场景里，数据库的响应时间无疑是最大时延，与之相比代码运行时间就微不足道了。而且通过 Cache 和优化，基本上可以消除代码运行效率低对项目的影响。又如我们的消息网关系统，最耗时的部分就是网络通信和文件 I/O，而这两部分动态语言和静态语言相比并无明显劣势，运行效率的问题可以完全忽略。第二，如果遇到很耗 CPU 或者很耗内存的运算，完全可以通过 C/C++实现的扩展来解决。无论是 Python 还是 Ruby，都支持采用 C/C++编写扩展。通过这些扩展，可以极大地提高运行效率，从而弥补动态语言在运行效率上的不足。

2. **BUG** 难于发现。动态语言由于没有构建的过程，因此很多错误只有等到运行时才会发现。而这些错误很可能是些低级错误，比如拼写错误、没有 import 相关的类库，或者括号不匹配等等。如果每次修复这样的 BUG 都要通过去测试环境中部署来验证的话，则会浪费了大量时间。因此动态语言往往需要 充分的自动化测试套件，才能

够确保代码基本可用。另外，使用动态语言的时候，一个良好的代码静态检查工具也是很有必要的。它不但可以纠正一些低级错误，而且还可以帮助你发现代码中的 Bad Smells，大大提高开发效率。对于 Python 来说，Pyflakes 或 Pylint 都是不错的选择；而 Ruby 也有众多工具可供使用。测试充分的代码也更容易重构，在重构动态语言项目时要万分小心，因为动态语言极容易犯错，稍不留意就会引入新的 BUG。保持小步前进的步伐，每次修改后都执行测试，最好再通过持续集成环境来帮助发现测试失败的情况，这样重构起来才能得心应手。

3. 专业人员少。不少使用动态语言的公司都会遭遇一个问题，那就是使用动态语言的资深开发人员很少，不但很难招聘到靠谱的员工，核心人员的离队也会对公司造成很大的损失。这是因为完全使用动态语言进行开发的公司少的可怜，只有极少数的开发者能够参与其中并获得相关的开发经验。绝大多数的动态语言 使用者还处在爱好者阶段，跟着 Tutorials 写写 Demo，或者随手写个 Utils 等等。因为高水平的动态语言开发者的确是可遇不可求，因此寻找有经 验的开发者也许要花上不少的时间和成本。当团队有了较为有经验的开发者以后，就需要通过内部培训、结对编程等手段，帮助公司里没有经验的开发者迅速积累经验，逐渐成为动态语言方面的靠谱人才。其实，对于动态语言的圈子，还有一个有趣的说法：因为学习动态语言的人往往都是在其他领域有了很深的积累后，在有余力的情况下才接触动态语言的，因此往往相对都比较靠谱，动态语言的圈子反而能够帮助雇主们甄选出一批高素质的开发者。

4. 不够成熟。动态语言的发展历史虽然不比静态语言差到哪里（比如 Ruby 和 Java 就同为 1995 年始创），然而由于其较为小众，因此无论是虚拟机的实现上，语言本身的机制上，还是相关的配套工具上都算不得十分成熟。例如，Ruby 虽然以其优美灵活的语法为人所称道，但也因为其虚拟机效率低下 和内存泄露问题所为人诟病，使用 Ruby on Rails 的网站往往需要加配监控程序，一旦发现某个 VM 内存超标立刻重启；Python 的虚拟机虽然还算稳定，但长久以来一直受 GIL（Global Interpreter Lock）问题所困扰，完全无法发挥多核的优势，这在家用 PC 都早已多核的今天的确是个不小的问题（事实上 Ruby 也存在 GIL 问题）。不过，虽然官方实现不够成熟，现在已经有很多逐渐成熟的其他选择可供使用。比如 JRuby 就充分利用了 Java 成熟的虚拟机和 Ruby 优良的语法特性，还可以允许开发者使用 Java 背后庞大的类库。通过 multiprocessing 或 Stackless Python，甚至手工将任务切成多份，分发给多个进程运行，都可以规避掉 GIL 的问题，更充分地利用系统性能。当然，随着时间的推移，动态语言的实现将会越来越成熟，不但 MRI 逐渐完善，MagLev 和 Rubinius 等一系列优秀的 Ruby 虚拟机也开始登上舞台；Python 3000 甚至打破了向后兼容性，试图将 Python 以前的设计错

误全面改写。回头去看 Java 等一批成熟开发语言的发展路线，有谁没有经历过不成熟的青春期呢？

小结

通过实践我们发现，动态语言既不是什么洪水猛兽，也不是什么奇巧玩物，它们已经逐渐成长为称手的兵器，帮助开发者们快速完成项目，进而达成商业目标。使用动态语言，已经让我们切切实实感受到了它的开发效率为我们所带来的好处。在商业机会瞬息万变的今天，谁能以最快的速度实现自己的想法，谁能尽快应对市场带 来的变化，谁就能离成功更进一步。

诚然，动态语言目前还存在很多问题。但瑕不掩瑜，如果在使用时可以意识到这些问题，并善加处理的话，动态语言也可以成为复杂商业系统的主角，在企业开发中占据自己的地位。而且随着开源社区的努力，很多问题正逐一被解决。我们有理由相信，在不远的未来，动态语言一定会有一片更为广阔的天空。

感谢田乐（Tin）和赖翥翔（Jason Lai）对本文提出了大量的反馈意见，感谢霍泰稳为本文找到如此贴切的标题。

原文链接：http://www.infoq.com/cn/articles/pros-and-cons-of-dynamic-languages

相关内容：

- 什么是IronRuby？开发者如何在Rails中使用它？
- FlightCaster秘笈——Clojure和Rails
- Python Web框架UliWeb开发进展
- 专访Guido van Rossum：迎接Python 3000
- TIOBE编程语言排行榜：别了，Smalltalk！

推荐文章 | Articles

揭秘淘宝自主研发的文件系统——TFS

作者崔康

目前，国内自主研发的文件系统可谓凤毛麟角。淘宝在这一领域做了有效的探索和实践，Taobao File System（TFS）作为淘宝内部使用的分布式文件系统，针对海量小文件的随机读写访问性能做了特殊优化，承载着淘宝主站所有图片、商品描述等数据存储。

最近，淘宝核心系统团队工程师楚材（李震）在其官方博客上撰文（《TFS 简介》，以下简称文章）简要介绍了 TFS 系统的基本情况，引起了社区的关注。

文章首先概括了 TFS 的特点：

> - 全扁平化的数据组织结构，抛弃了传统文件系统的目录结构。
> - 块设备基础上建立自有的文件系统，减少 EXT3 等文件系统数据碎片带来的性能损耗。
> - 单进程管理单块磁盘的方式，摒除 RAID5 机制。
> - 带有 HA 机制的中央控制节点，在安全稳定和性能复杂度之间取得平衡。
> - 尽量缩减元数据大小，将元数据全部加载入内存，提升访问速度。
> - 跨机架和 IDC 的负载均衡和冗余安全策略。
> - 完全平滑扩容。

当前，TFS 在淘宝的应用规模达到“数百台 PCServer，PB 级数据量，百亿数据级别”，对于其性能参数，楚材透漏：

> TFS 在淘宝的部署环境中前端有两层缓冲，到达 TFS 系统的请求非常离散，所以 TFS 内部是没有任何数据的内存缓冲的，包括传统文件系统的内存缓冲也不存在……基本上我们可以达到单块磁盘随机 IOPS（即 I/O per second）理论最大值的 60%左右，整机的输出随盘数增加而线性增加。

TFS 的逻辑架构图 1 如下所示：

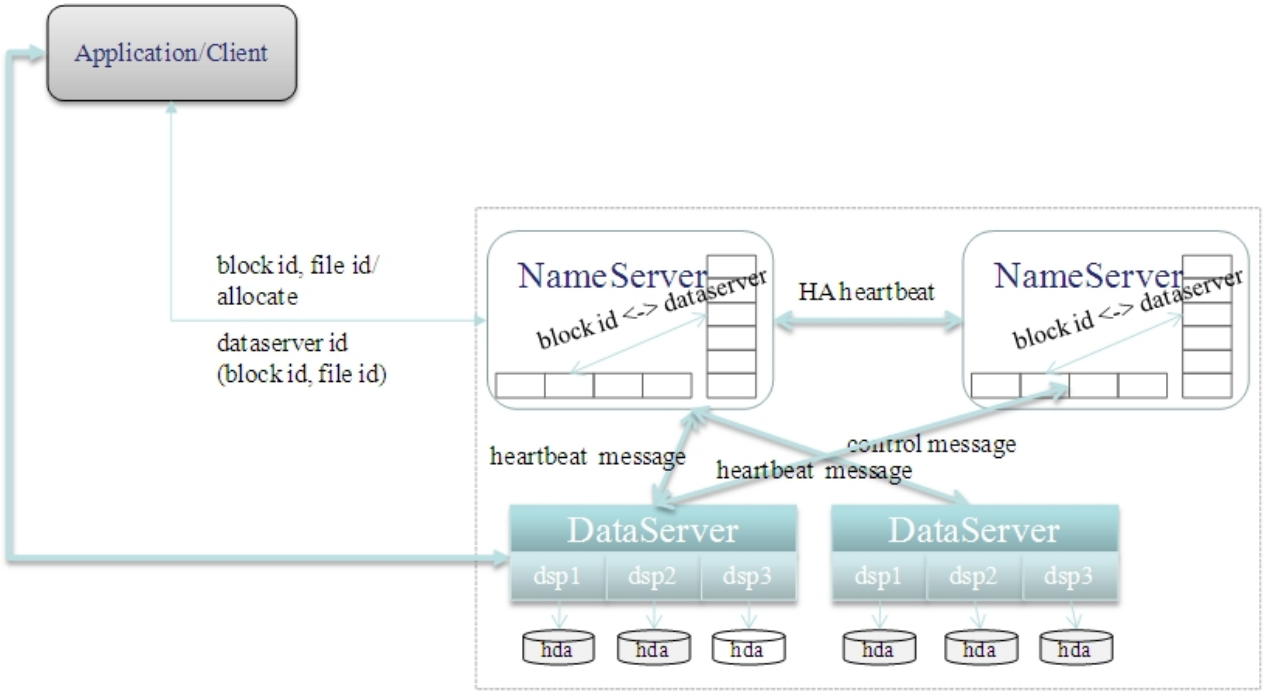


图 **1.** TFS 逻辑架构图（来源：淘宝核心系统团队博客）

楚材结合架构图做了进一步说明：

1、 TFS 尚未对最终用户提供传统文件系统 API ,需要通过 TFSClient 进行接口访问 ,现有 JAVA、JNI、C、PHP 的客户端

2、 TFS 的 NameServer 作为中心控制节点，监控所有数据节点的运行状况，负责读写调度的负载均衡，同时管理一级元数据用来帮助客户端定位需要访问的数据节点

3、 TFS 的 DataServer 作为数据节点，负责数据实际发生的负载均衡和数据冗余，同时管理二级元数据帮助客户端获取真实的业务数据。

文章发表以后，读者反响热烈，在评论中提出了各种问题与作者楚材进行技术交流，由此可见国内社区对自主研发文件系统的关注程度。

为了让读者更深入地了解 TFS 的奥秘，InfoQ 中文站针对 TFS 的来由、运行环境、扩展性、架构、发展规划及开源事宜等问题对楚材（李震）进行了专访。

InfoQ：淘宝为何会选择自主研发文件系统？

“TFS(Taobao File System)，是淘宝自主研发的一套分布式文件系统，用于存储淘宝网主站的数据，例如商品图片、商品描述、交易快照、社区图片等等等等。

这些数据的一个突出特点是单个文件尺寸较小，通常不大于 1MB，但是数量巨大，传统的文件系统或者网络存储设备很难解决类似的问题。

淘宝网最早采用 NetApp 提供的网络存储设备，但是在 2006 ~ 2007 年期间，由于数据量的急剧膨胀，我们面临着系统更新换代的压力，而由于应用连接数量和文件数量的限制，实际上简单的升级已经变得非常昂贵且得不偿失了。

数据是淘宝应用的核心，在这方面我们需要提供更加安全、高效、廉价的解决方案，需要发展自己的技术，更加适应淘宝网自身的应用特性。经过评估，已有的一些开源分布式文件系统都无法完全满足我们需求。淘宝网也许是当时国内互联网最先遇到类似问题的公司，当然也可能是其他公司遇到但未进行相关的宣传，总之，当时并不像现在一样有一些相对可靠的开源系统可以选择。

InfoQ：TFS 系统的运行环境如何？

“在做 TFS 选型的时候，我们依照当时淘宝网使用的主流操作系统使用了 REHL，并构建在 LINUX 的 EXT3 文件系统上。选择 REHL 的理由是开源操作系统，我们有更多的自由度，同时成本也更加节省。

关于硬件平台，随着硬件的发展和我们认识的变化，经过了一个相对比较长的变迁。分布式文件系统对 CPU 的要求并不高，同时由于我们采用了一些特殊的手段， 也使得系统元数据减少，那么对内存大小的要求也并不高了，所有的性能和成本瓶颈都集中在了磁盘 IO 上。我们所有硬件架构的演变也都体现在磁盘的使用方式上。

最早我们使用 SAS146G×6 的磁盘进行存储，后来换成了 SAS300G×6 的磁盘，使用了 RAID5 机制，这也是当时硬件平台的主流。当我们积累了一定的经验并逐步对系统稳定性有了信心之后，这种方式就逐步落后了。

RAID5 机制的备份和我们已经在应用层实现的数据冗余功能重复，磁盘空间被大量浪费，RAID 卡的工作原理也无法优化 TFS 更加注重随机 IO 的特性，最终我们去除了 RAID 卡，并使用多进程，每个进程管理一块磁盘的方式，充分发挥磁盘的随机读写性能。

SAS300G×6 的磁盘我们使用了一段时间，相对比较稳定，这让我们有信心使用更大容量的节点而不会过于担心服务器失效带来的数据迁移对系统性能的影响，当前 TFS 主要的机型使用 SAS300G×12 的磁盘配置。

随着淘宝网前端的 CDN 越来越高效、稳定，到达 TFS 的请求流量也更加平缓，这让我们有可能对单个节点的服务能力做出更加可靠的规划，也可以使用更加廉价的存储设备，未来我们的主要存储节点将使用 1TSATA×12 甚至更大的磁盘设备，进一步降低成本。

InfoQ：TFS 的应用规模达到数百台 PC server 和 PB 级数据量，其扩展性如何？架构上是如何保证的？

“在 TFS 中我们将大量的小文件合并成为一个大文件，类似 GFS 中 Chunk 的概念，而 Chunk 的定位信息我们称之为一级索引，而 chunk 内部具体的文件定位信息我们称之为二级索引，同时在 TFS 文件名称中包含这些索引信息，在用户写入一个文件之前，他必须向 TFS 系统申请一个文件名称。这种方式虽然在某些情况下显得不像传统文件系统那样灵活，但也给了我们系统更大的可扩展性。我们保证可以中心控制节点的内存可以支撑 PB 级别的一级索引，而二级索引仅需 要针对单台数据量。这样，我们就避免了数据量膨胀带来的扩容难度。

当存储容量出现不足，我们需要进行系统扩容的时候，可以根据数据增长情况进行规划，任意数量的加入提供相应存储的服务器。而这些新的存储服务器会向中心控 制节点进行报告。而中心控制节点在有数据写入时，将根据已存储容量的百分比、系统当前负载等参数动态地分配写入的服务器。同时，在系统空闲时间段，中心控 制节点也会根据当前的数据分布情况制定数据迁移计划，并逐步完成数据平衡。与此类似，当发生服务器崩溃时，中心控制节点也会进行数据迁移以保证足够的备 份，同时也会进行数据均衡操作。这些操作都是自动进行的，不需要人工干预。

这种方式在 1000 台以内的集群基本上能够工作良好，如果集群规模更大，中心控制节点可能会出现一些性能瓶颈，届时我们可以使用一个分布式集群来解决，相应已经比较成熟的技术方案现在已经比较多了。

InfoQ:您在介绍 TFS 的特点时，多次提到“性能”这个关键词，请问对于一个文件系统来说，性能一般应该考虑哪些因素？目前，提高文件系统性能的通用方法有哪些？

“现在我们可以讨论一下 TFS 的性能考量的维度了。其实当前每个流行的分布式文件系统都有自己的侧重点，分别针对自己不同的应用场景。

对于离线型的数据分析需求，由于数据总量巨大，底层分布式文件系统更加注重系统的整体吞吐率，为了适应当前流行的 map_reduce 模型，这一类的文件系统会将文件切成多份，分而治之。同时和上层的逻辑配合，采用大块顺序写入、读出的方式来

提升性能，典型的代表就是 Hadoop 使用的 HDFS。

实际提供线上服务的分布式文件系统中，也根据服务类型的不同而采用差异化的存储方式，例如针对音、视频等相对比较大的文件，可能采用和离线应用相同的方式将文件切片并发读写访问，从而达到更高的传输速度。

由于相同容量下文件数量比较少，甚至有可能完全实现类似传统文件系统的目录、权限等功能，而不会受到 inode 的限制。如果面临淘宝相同的海量小文件存储，这种方式就完全无法提供性能的支持了，inode 数量的膨胀会很快吃掉大量的昂贵内存，如果要平衡成本将部分 inode 放入磁盘，面对基本上完全随机、没有热点的访问又无法保证寻址的效率，我们只能通过减少元数据的方式来解决这个问题。

为什么说我们的 TFS 面临着完全随机、基本上没有热点的数据访问？在淘宝的数据部署中，TFS 前端有两层更加靠近用户的缓存系统来保证用户展示页面的速度，最终到达 TFS 的请求大概只有总请求数量的 2%左右，随着前端缓冲的效率不断提升，这个比例还会继续下降，我们可以想象一下是否仍然有热点数据存在。与此同时，淘宝网的业务特点决定了相对于读取请求，写入请求量不在一个数量级上，而修改操作量就更少，这就决定了我们使用进行随机修改、随机写入的方式来 避免顺序写入不进行修改给随机读取带来的成本。

不同的应用场景，不同的存储架构，不同的性能考量，有什么通用的性能优化手段吗？从我们的实践经验来看，似乎没有，很难有一种方法解决所有的问题，或者说解决了这些问题之后，大家会发现相对专用的文件系统，通用系统总是不能达到你预期的性能，同时也带来了大量开发和调优方面的复杂性。这种情况下，也许我们建立不同的集群解决不同的问题更加高效，这一切都取决于你支撑的业务需求是怎么样的，Google 级别集群规模的架构和简单搭建起来的 NFS 都有其存在的价值。

当然，还是有一些通用的规则的。如果你需要支持的是一个线上服务，那你就要尽量减少一次请求引发的 IO 数量，IO 包括网络及物理磁盘，这些操作引发的性能开销是由物理结构决定，无法用技术手段去优化的。也就是说，这些操作的数量实质上形成了你这个系统的性能天花板，当你设计一个文件系统的架构时，你可以根据这个天花板预估到在不同情况下这套系统的性能表现是怎么样的。

其次，你需要在并发和同步开销之间做出平衡，根据业务的特点选择更适合于你自己的方案，例如从一块磁盘上连续读 1MB 数据和通过网络并发从 10 块磁盘上分别读取 100KB 数据返回给用户，你会选择哪种？那么从一块磁盘上连续读取 100MB 数据和

100 并发分别读取读取 1MB 呢，系统是否有能力支撑这样的并发？绝大部分时候，我们是在做类似的选择。

第三，你需要在成本和性能之间做出平衡，我们可以很方便的使用内存型缓冲来极大地提升系统性能，如果你的系统热点数据集中的话。随着 SSD 技术的成熟，它可以提供的随机读性能相对传统机械磁盘让人兴奋，随机写性能也有了极大的改善，虽然不像读取提升的那样夸张。如果你的应用读写比例比较高，又可以承受成本，尽量使用 SSD 吧，虽然当前的成本还是相对昂贵的。

InfoQ：从 **TFS** 的逻辑架构图来看，**NameServer** 作为中心控制节点，**DataServer** 作为数据节点，这样的架构基于哪些因素或者需求的考虑？

> TFS 有中心控制节点和数据节点的区分，分别管理两级索引解决数据访问的寻址和传输。相对当前一些开源分布式系统的去中心化趋势，似乎不是非常优秀。但如同对性能的论述一样，每种系统架构都有其存在的价值，中心节点的存在可以保证事务性，我们在某些情况下必须保证写入数据的强一致性，同时在当前的应用规模中也可以非常安全、高效的解决问题。如果有更高的可用性、扩展性需求，我们会在 TFS 部分中部分体现去中心化的思想。

InfoQ：未来 **TFS** 的发展计划如何？

> TFS 未来的开发首先仍然会立足于淘宝网自身的业务需求，同时会照顾开发社区中的需求。我们会逐步支持大文件的存储，也会支持目录和用户权限，同时计划实现按照访问特性分布的分级存储，尽量在性能和成本之间达到一个平衡，还有一些更加精细化的管理功能，例如数据量配额的管理等等。从上面的一些讨论来看，我们不大会做一个通用的分布式文件系统，始终会专注解决当前尚无很好解决方案的问题。

InfoQ：您在博客中提到 **9** 月份建立 **TFS** 开源社区，这对国内社区是个绝佳的学习机会，请问目能否再描述一点开源的细节？

> TFS 计划在 9 月底开源，而今年 6 月底，淘宝网将推出自己的开源社区——code.taobao.org，TFS 将完全基于这个开源社区进行开源，大家马上就可以看到。同时已经基本确定使用 GPL V2 开源版权协议。

InfoQ：对于有志于参与到 **TFS** 或者目前从事其他类似系统级核心应用的开发人员来说，请问有什么好的建议？

> 对于有志于参加 TFS 开发，或者自身已经在从事基础平台、核心系统研发的同行，我相信大家都有相同的感受，在当今计算机体系结构内，我们很难有革命性的技术进展。当前已知的大规模的分布式文件系统都构建在 Unix 类操作系统之上，或者说绝大部分都构建在 Linux 之上，这也是从成本方面进行的考量。而各种分布式文件系统架构也大同小异，决定是否成功的关键在于细节，这些细节包括操作系统级别的特化、文件系统级别的特化、实现方面是否足够优秀、足够稳定等等。因此你需要对系统内核有所了解，对文件系统有所了解，比如你知道 EXT3 的组织方式才可能尽量避免读取一段数据却引发多次磁盘磁头移动的情况，这样你才能最大化的利用好系统资源。而实现的特化可能体现在一个优秀算法的编写、一个高效的通信机制等等，这就要求你有扎实的代码编写能力，对算法和数据结构有颇深的造诣。大家都知道，细节决定成败！

InfoQ 中文站将继续关注 TFS 等国内自主研发文件系统的发展，也欢迎大家发表自己的看法。

专家介绍：

李震，花名楚材，工作于淘宝技术研发部核心系统研发，负责存储组的开发，关注领域主要包括大集群分布式系统的研发、海量数据处理、海量数据检索等等。

原文链接：http://www.infoq.com/cn/articles/tao-tfs

相关内容：

- Zynga CTO谈如何设计高可扩展性社交游戏
- 百度大规模数据处理
- Google编程之夏和Ruby编程之夏中的动态语言项目
- Nokia基于EPL协议向Symbian基金会捐献手机版Java运行时系统
- AMQP和RabbitMQ入门

推荐文章 | Articles

企业中的 NoSQL

作者 Sourav Mazumder 译者 黄璜

引子

作为企业架构师，我的职业习惯之一，就是不断的探求各种新的有前景的概念和思想，看其是否有潜力为我所服务的来自各行各业的企业客户带来价值。同样出于对这种理念的追求，我对 NoSQL 领域的关注了也有一段时间了，甚至从这个术语产生(或者错误的产生?)之前就开始了。Google 首先在这方面点了一把火，发布了论文 Big Table 架构，对关系数据库是银弹这种普遍的信念提出了质疑，而 Amazon 关于 Dynamo 的论文则紧随其后。过去的一年中我们见证了 NoSQL 强劲的势头，在这一领域有多达 25 种产品/解决方案发布，并且 NoSQL 的触角已经伸向了业界的各个角落。在此前提下，我最近考虑深入这一领域，评估一下我的客户究竟如何才能从这种 NoSQL 运动中获益。不仅如此，我还想探究对于企业来说，是否是到了该认真考虑采纳 NoSQL 的合适时机了。

什么是 NoSQL——快速回顾

像许多关注这一领域的人一样，我不喜欢从本质上将 SQL 与 NoSQL 这一术语对立起来。同时我对该术语现有的解释"Not Only SQL"也不甚满意。对我来说，我们这里所讨论的并非是是否使用 SQL。(相反的是，我们仍然可以选择类似 **SQL** 这样的查询接口(缺少对 **join** 等的支持)来与这些数据库交互，使用现有的资源和技术来管理开发伸缩性和可维护性。) 这一运动是要找到存储和检索数据的其他高效的途径，而不是盲目地在任何情况下都把关系数据库当作万金油。因此，我认为'Non Relational Database'(非关系型数据库)能够更好的表达这一思想。

无论采用哪个名字，“非关系型数据库”这一范围所传达出来的“囊括所有”类型的意味，使得这一概念比较模糊(并且它还是否定型的)。这又使得人们(特别是企业中的决策者)对于哪些是属于这个范围，哪些不是，更重要的是，对他们来说这到底意味着什么，感到非常迷惑。

为了解答这些疑问，我尝试通过以下几点特征的描述，来刻画“非关系型数据库”的内在本质。

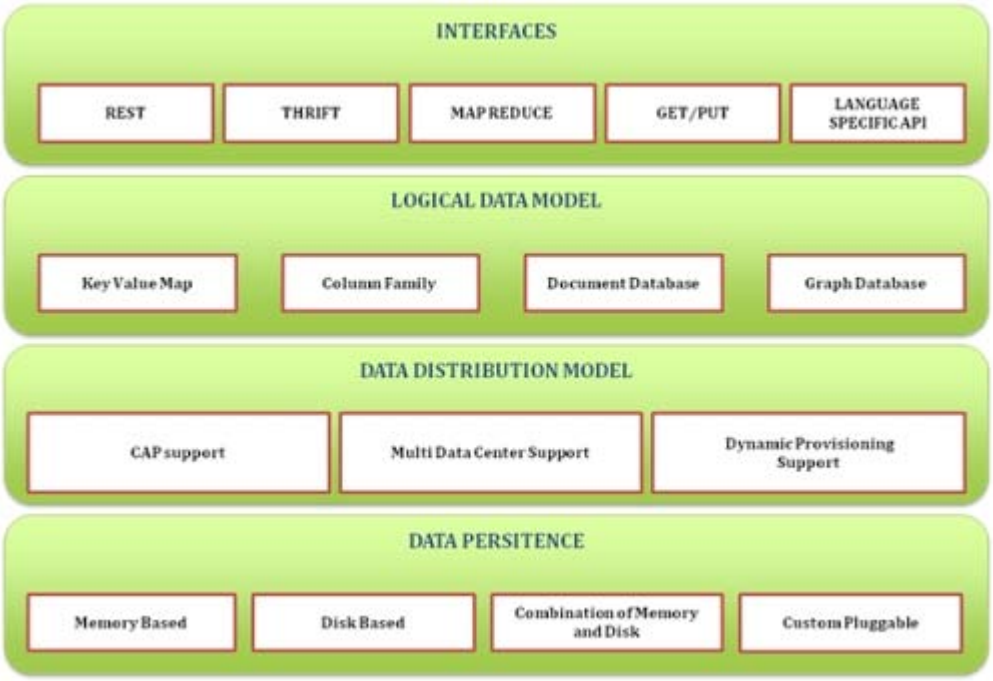
所谓“非关系型数据库”指的是

1、 使用松耦合类型、可扩展的数据模式来对数据进行逻辑建模(Map，列，文档，图表等)，而不是使用固定的关系模式元组来构建数据模型。

2、 以遵循于 CAP 定理（能保证在一致性，可用性和分区容忍性三者中中达到任意两个）的跨多节点数据分布模型而设计，支持水平伸缩。这意味着对于多数据中心和动态供应（在生产集群中透明地加入/删除节点）的必要支持，也即弹性(Elasticity)。

3、 拥有在磁盘或内存中，或者在这两者中都有的，对数据持久化的能力，有时候还可以使用可热插拔的定制存储。

4、 支持多种的'Non-SQL'接口(通常多于一种)来进行数据访问。



围绕着图中四个特征的（数据持久性、逻辑数据模型、数据分布模型和接口）“非关系型数据库”的各种变形，在最近的一些文章中有详尽的描述，并且在因特网上有着广泛的传播。所以我就不做过多繁复的描述，而是通过一些例子对关键的方向进行总结，供快速参考。

接口——REST (HBase ,CouchDB ,Riak 等) ,MapReduce (HBase ,CouchDB ,MongoDB ,Hypertable 等)，Get/Put (Voldemort，Scalaris 等)，Thrift (HBase，Hypertable，Cassandra 等)，语言特定的 API(MongoDB)。

逻辑数据模型——面向键值对的(Voldemort，Dynomite 等)，面向 Column Family 的(BigTable，

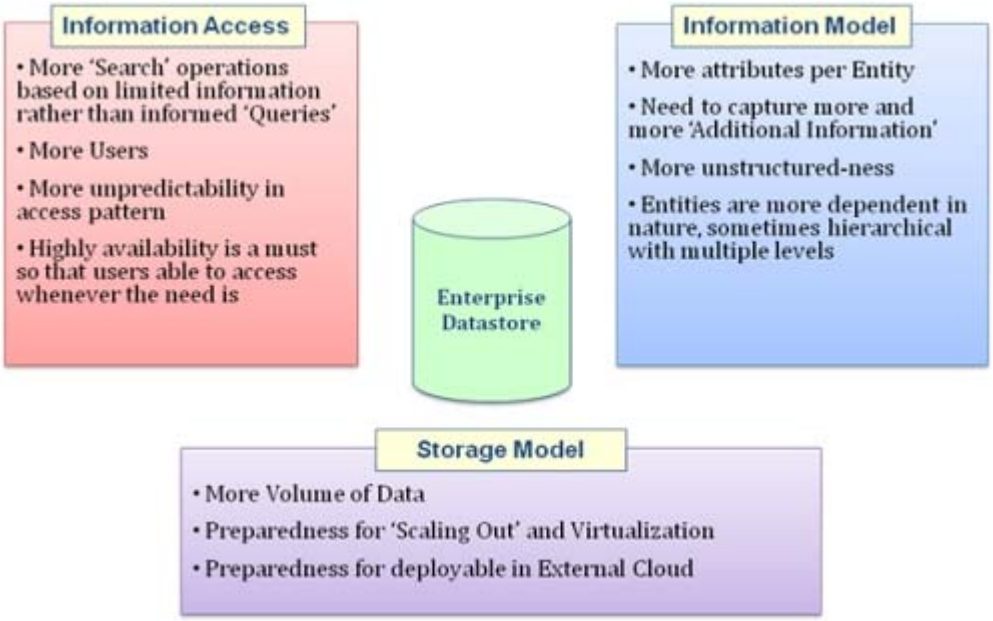
HBase，Hypertable 等)，面向文档的(Couch DB，MongoDB 等)，面向图的(Neo4j， Infogrid 等)

数据分布模型——一致性和可用性(HBase，Hypertable， MongoDB 等)，可用性和可分区性(Cassandra 等)。一致性和可分区性的组合会导致一些非额定的节点产生可用性的损失。有趣的是目前还没有一个“非关系型数据库”支持这一组合。

数据持久性——基于内存的(如 Redis，Scalaris， Terrastore)，基于磁盘的(如 MongoDB，Riak 等)，或内存及磁盘二者的结合(如 HBase，Hypertable，Cassandra)。存储的类型有助于我们辨别该解决方案适用于哪种类型。然而，在大多数情况下人们发现基于组合方案的解决方案是最佳的选择。既能通过内存数据存储支持高性能，又能在写入足够多的数据后存储到磁盘来保证持续性。

如何将其与企业 IT 融合

如今的企业中，并非所有用例都直观地倾向于使用关系型数据库，或者都需要严格的 ACID 属性(特别是一致性和隔离性)。在 80 年代及 90 年代，绝大部分存储在企业数据库里的数据都是结构化的业务事务的“记录”，必须用受控的方式来生成或访问，而如今它已一去不复返了。无可争辩的是，仍有这一类型的数据在那 里，并将继续也应该通过关系型数据库来建模，存储和访问。但对于过去 15 年以来，随着 Web 的发展，电子商务和社交计算的兴起所引起的企业里不受控的非结构化并且面向信息的数据大爆炸，该如何应对呢？企业确实不需要关系型数据库来管理这些数据，因为关系型数据库的特点决定了它不适用于这些数据的性质和使用方式。



上图总结了现今以 web 为中心的企业中信息管理的新兴模式。而“非关系型数据库” 是处理这些趋势的最佳选择(较之关系型数据库来说)，提供了对非结构化数据的支持，拥有支持分区的水平伸缩性，支持高可用性等等。

以下是支持这一观点的一些实际应用场景：

日志挖掘——集群里的多个节点都会产生服务器日志、应用程序日志和用户活动日志等。对于解决生产环境中的问题，日志挖掘工具非常有用，它能访问跨服务器的日志记录，将它们关联起来并进行分析。使用“非关系型数据库”来定制这样的解决方案将会非常容易。

分析社交计算——许多企业如今都为用户(内部用户、客户和合作伙伴)提供通过消息论坛，博客等方式来进行社交计算的能力。挖掘这些非结构化的数据对于获得用户的喜好偏向以及进一步提升服务有着至关重要的作用。使用“非关系型数据库” 可以很好的解决这一需求。

外部数据 **feed** 聚合——许多情况下企业需要消费来自合作伙伴的数据。显然，就算经过了多轮的讨论和协商，企业对于来自合作伙伴的数据的格式仍然没有发言权。同时，许多情况下，基于合作伙伴业务的变更，这些数据格式也频繁的发生变化。通过“非关系型数据库”来开发或定制一个 ETL 解决方案能够非常成功的解决这一问题。

高容量的 **EAI** 系统——许多企业的 EAI 系统都有高容量传输流(不管是基于产品的还是定制开发的)。出于可靠性和审计的目的，这些通过 EAI 系统的消息流通常都需要持久化。对于这一场景，“非关系型数据库” 再次体现出它十分适用于底层的数据存储，只要能给定环境中源系统和目标系统的数据结构更改和所需的容量。

前端订单处理系统——随着电子商务的膨胀，通过不同渠道流经零售商、银行和保险供应商、娱乐服务供应商、物流供应商等等的订单、应用、服务请求的容量十分巨大。同时，由于不同渠道的所关联的行为模式的限制，每种情况下系统所使用的信息结构都有所差异，需要加上不同的规则类型。在此之上，绝大部分数据不需要即时的处理和后端对帐。所需要的是，当终端用户想要从任何地方推送这些数据时，这些请求都能够被捕获并且不会被打断。随后，通常会有一个对帐系统将其更新到真正的后端源系统并更新终端用户的订单状态。这又是一个可以应用“非关系型数据库”的场景，可用于初期存储终端用户的输入。这一场景是体现“非关系型数据库”的应用的极佳例子，它具有高容量，异构的输入数据类型和对帐的"最终一致性"等等特点。

企业内容管理服务——出于各种各样的目的，内容管理在企业内部得到了广泛的应用，横跨多个不同的功能部门比如销售、市场、零售和人力资源等。企业大多数时间所面临的挑战是用一个公共的内容管理服务平台，将不同部门的需求整合到一起，而它们的元数据是各不相

同的。这又是“非关系型数据库”发挥作用的地方。

合并和收购——企业在合并与收购中面临巨大的挑战，因为他们需要将适应于相同功能的系统整合起来。“非关系型数据库” 可解决这一问题，不管是快速地组成一个临时的公共数据存储，或者是架构一个未来的数据存储来调和合并的公司之间现有公共应用程序的结构。

但我们如何才能准确的描述，相对于传统的关系型数据库，企业使用“非关系型数据库”带来的好处呢？下面是可通过非关系型数据库的核心特点(正如上一节所讨论的)而获得的一些主要的好处，即企业的任何 IT 决策都会参考的核心参数——成本削减，更好的周转时间和更优良的质量。

业务灵活性——更短的周转时间

“非关系型数据库”能够以两种基本的方式带来业务灵活性。

- 模式自由的逻辑数据模型有助于在为任何业务进行调整时带来更快的周转时间，把对现有应用和功能造成影响减到最少。在大多数情况下因任意的变更而给你带来的迁移工作几乎为零。
- 水平伸缩性能够在当越来越多的用户负载造成负载周期性变化，或者应用突然变更的使

用模式时，提供坚固的保障。面向水平伸缩性的架构也是迈向基于 SLA 构建(例如云)的第一步，这样才能保证在不断变化的使用情形下业务的延续性。

更佳的终端用户体验——更优越的质量

在现今企业 IT 中，应用的质量主要由终端用户的满意度来决定。“非关系型数据库”通过解决如下终端用户的考虑因素，能够达到同样的效果，而这些因素也是最容易发生和最难以处理的。

- “非关系型数据库”为提升应用的性能带来了极大的机会。分布式数据的核心概念是保证磁盘 I/O(寻道速率)绝不能成为应用性能的瓶颈。尽管性能更多的是由传输速率来决定。在此之上，绝大部分解决方案支持各种不同的新一代的高速计算的范式，比如 MapReduce，排序列，Bloom Filter，仅可追加的 B 树，Memtable 等。
- 现今用户满意度的另一个重要的方面就是可靠性。终端用户希望在想要访问应用时就能访问到，并且至少是在当他们分配到时间的时候能随时执行他们的任务。所以不可用的应用需要不惜代价的避免。许多现代的“非关系型数据库”都能适应并支持这一类有着严格和最终一致性的可用性的需求。

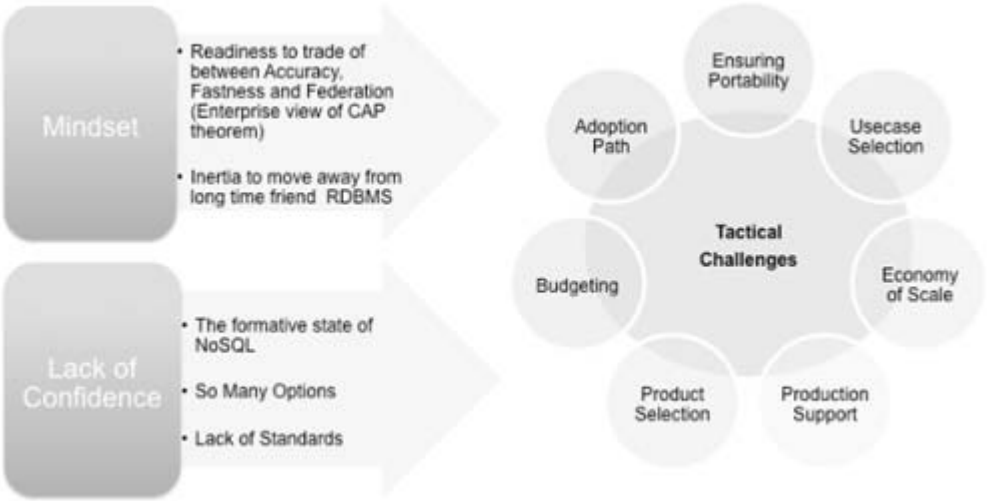
更低的所有者总成本

在如今的竞争市场中，企业 IT 支出随时都要仔细审查，以合理的成本获取合理的质量才值得赞许。在这一领域中“非关系型数据库”在一定程度上胜于传统的数据库，特别是当存储和处理的数据容量很大时。

- 水平伸缩性的基本前提保证了它们可以运行于廉价机器之上。这不仅削减了硬件资本的成本，同时还削减了诸如电力，维护等运维成本。同时这还进一步的为利用诸如云、虚拟数据中心等下一代低成本的基础设施打下了基础。
- 从长期来看，更少的维护能带来更多的运维成本优势。对于关系型数据库，这绝对是一个需要存储大容量数据的场景。为大容量的数据调优数据库需要高超的技艺，也就意味着更高的成本。相较之下，“非关系型数据库”始终提供快速和响应的特点，就算是在数据大幅上升的情况下。索引和缓存也以同样的方式工作。开发者不必过多担心硬件、磁盘、重新索引及文件布局等，而是把更多的精力投入了应用程序的开发上。

企业采用中所面临的挑战

抛开所有这些长远的好处，在企业拥抱“非关系型数据库”之前，当然还需要经历各种各样的挑战。



不考虑因现有思想的转换和缺乏信心而产生的来自高层的阻力，目前我认为的最主要的战术性挑战是：

为“非关系型数据库”认定正确的应用/使用场景

尽管从理论上容易论证并非所有的企业数据都需要基于关系和 ACID 的系统，然而由于关系型数据库与企业数据间多年的绑定关系，要作出所有的数据可以通过非关系的解决方案而解耦的决定仍然有很多困难。许多时候 IT 经理(以及其它对于应用程序负有核心的底线责任的各级人员)不明白他们将会失去什么，这样的担忧对于从关系型数据库转变出来比较不利。企业 IT 最有价值的资产就是数据。因此，要作出决定使用一种不太明确或者未被广泛采用的解决方案来管理同样的数据，这种能力不仅需要转换思维方式，同时还需要来自高层的强大的支持(和推动)。

我们如何选择最适合我们的产品/解决方案

另一个重大的挑战是找出合适的产品/工具来提供“非关系型数据库”。正如前面所提到的那样，现今业界里面有多于 25 种不同的产品和解决方案，它们在四个方面有着不同的特点。正因为每个产品在这四个方面特点各异，所以要选择一个产品来应对所有的需求显得尤为困

难。有的时候，可能在企业的不同部门使用到多种类型的非关系型数据库，最后人们可能会完全出于对标准的需要而转向关系型数据库。

如何获得规模经济

这一想法本质上是从前一个问题分支出来的。如果一个组织需要使用多个非关系型数据库解决方案(由于单个方案的适用问题)，那么保证在技术(开发者，管理者，支持人员)，基础设施(硬件成本，软件许可成本，支持成本，咨询成本)，以及工件(公共组件和服务)方面的规模经济就是一个大问题。这一方面与传统的关系型数据库解决方案比较起来确实更为严峻，因为大部分时间组织的数据存储都是以共享服务的模式在运行的。

我们如何保证解决方案的可移植性

从“非关系型数据库”的发展来看，我们可以很直观地推测在未来的几年中这一领域会有许多变化，比如供应商的合并，功能的进步以及标准化。所以对于企业来说一个更好的策略是不要把宝押在某个特定的产品/解决方案上，以后才可以更灵活的转换到一个更好的经过考验的产品。由于现在的非关系型产品/解决方案大部分是私有的，因此 IT 决策者在考虑尝试“非关系型数据库”之前，不得不认真考虑可移植性这一重要的问题。这纯粹是出于保护现有投资的需要。

我们如何获得合适的产品支持类型

现在的“非关系型数据库”能通过外部组织而提供支持方案的少之又少。就算有，也无法与 Oracle，IBM 或者微软等相比。特别是在数据恢复，备份和特定的数据恢复方面，由于许多“非关系型数据库”在这些方面未能提供一个健壮而易于使用的机制，对于企业决策者来说，仍存在很大的问题。

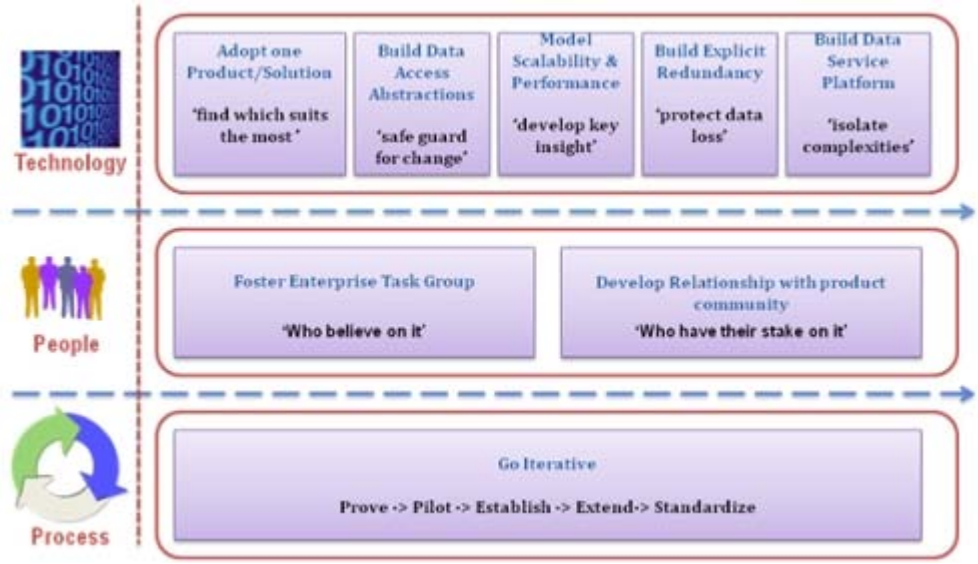
我们如何预算整体成本

与重量级的关系型数据库相比，“非关系型数据库”通常在性能和伸缩性特征方面能提供的数据更少。我也没有发现有 TPC 基准程序方面和类似的其它方面的数据。这将企业决策者置于了一个“没有方向”的情况下，因为他们不知道需要在硬件、软件许可、基础设施管理和支持等方面支出多大的费用。要得出一个预算估计，缺乏判断的数据就成了一个主要的障碍。因此在项目启动阶段，大部分情况下决策者还是会选择基于熟悉的关系型数据库的解决方案。

有时候，就算可以得到这些数字，但也不足以用来形成 TCO 模型并与传统的基于关系型数据库的数据存储和非关系型数据存储进行整体的成本分析（Capex+Opex)比较。通常情况下水平伸缩性所要求的大量的硬件机器(以及软件许可成本，支持成本)，如果与垂直伸缩性乍一比较，会让人觉得战战兢兢，除非由此带来的好处经过基于 TCO 模型的全方位比较仍然被证明是可以持续的。

关于如何采用 NoSQL 的两点思考

这是否意味着目前来看企业应该对 NoSQL 运动持观望的态度呢？并非如此。诚然，“非关系型数据库”对于广泛的采用来说还未到完全成熟的阶段。但“非关系型数据库”作为未来企业骨架的潜力仍不能忽视。特别是不远的将来企业将更多地处理大容量的半结构化/非结构化以及最终一致性的数据，而不是相对而言小容量的，严格结构化的遵循 ACID 的数据。所以现在而言至关重要的是做企业的关键决策人的思想工作，让他们明白企业的数据处理需要使用“非关系型数据库”。在这一过程中，要采取一些渐进的步骤把“非关系型数据库”应用到企业 IT 的一些关键的方面(技术，人员和流程)，并产生一定的价值。这样，就可以用一种缓慢而稳健的方式从整体上来解决我们之前所总结出来的一系列问题。



采用一个产品/解决方案

如今市场上的选择非常多样化，可根据“非关系型数据库”侧重的面不同而进行差异化的处理。与此同时，企业应用场景可能需要不同类型的特点。然而以不同的解决方案来处理不同的应用/使用场景从规模经济的角度出发对于企业是不适宜的。因此最好是根据目标应用的需要最终落实到某一个具体的产品/解决方案上。需要注意的是大多数的解决方案在特性上

都会有一些折中，有些特性可能在其它的产品中可以获得，有些可能只是在发展路线图当中暂时设定了一个位置。因为大部分的产品会在不久的将来不断趋于成熟，因此可以通过不同配置来提供不同的解决方案。所以只要现有的解决方案能适合目前大部分的需要，不妨作为一个起点将其采纳。

选择产品/解决方案的经验法则

- 支持所需要的逻辑数据模型应当被给予更高的权重。这将从实质上决定该解决方案在当前或未来能否灵活地适应不同的业务需求。
- 调查该产品所支持的物理数据模型的合适与否，据此对这一解决方案所需要的水平伸缩性、可靠性、一致性和分区性作出合理的评估。这同样能表明备份和恢复机制的可能性。
- 接口支持需要与企业标准的运行环境对齐。由于这些产品支持多样的接口，所以这一点可以得到很好的处理。
- 只要产品支持水平伸缩性，对于持久化模型的选择就不再重要了。

这里有一份一系列“非关系型数据库”的对照表。对于现在正认真考虑采用的企业来说，这是一个不错的起点。为了更贴近企业本身的情况，从 25+的集合中挑选出的子集所用到的的关键选择标准是：

1、 最重要的一点首先是企业应用程序必须支持有一定复杂程度的数据结构。否则的话，应用程序管理复杂性的责任将变得非常大。我认为比较合理的应当是介于纯粹的键值对与关系型模式中间的一种方案。出于这方面的考虑像 Vlodemort，Tokyo Cabinet 等产品就排除在了我的列表之外。

2、 第二点是以低成本的分片/分区为大容量数据提供水平支持。缺乏这样的支持就使得解决方案与任何关系型数据库无异了。因此像 Neo4J(尽管他有丰富的基于图的模型) ,Redis，CouchDB 等此时此刻就被过滤出我的列表之外了。

3、 最后一条评判标准，在企业级推广之前我会考虑一定程度的商业支持。否则的话，一旦出现生产环境的问题，我该去找谁呢？出于这一点，我不得不将现在的一些明星产品排除在外 ,比如 Cassandra(尽管有很大的可能不久的将来 Rackspace 或者 Cloudera 就会对其提供支持，因为它已经被用于一些生产环境里边了，比如 Twitter，Digg，Facebook)。

有了这些过滤标准，我可以精简这一列表，符合目前企业可用的产品有 MongoDB (下一版本就会提供 shards 支持)， Riak，Hypertable 和 HBase。下面这个表格中总结了这四个产品的主要特性。一个企业可以基于自己具体的实际情况从中作出选择，找到最适合自己需要的

特性。

特性	MongoDB	Riak	HyperTable	HBase
逻辑数据模型	富文档，并提供对内嵌文档的支持	富文档	列家族(Column Family)	列家族(Column Family)
CAP 支持	CA	AP	CA	CA
动态添加删除节点	支持(很快在下一发布中就会加入)	支持	支持	支持
多 DC 支持	支持	不支持	支持	支持
接口	多种特定语言 API(Java，Python，Perl，C#等)	HTTP 之上的 JSON	REST，Thrift，Java	C++，Thrift
持久化模型	磁盘	磁盘	内存加磁盘(可调的)	内存加磁盘(可调的)
相对性能	更优(C++编写)	最优(Erlang 编写)	更优(C++编写)	优(Java 编写)
商业支持	10gen.com	Basho Technologies	Hypertable Inc	Cloudera

数据访问抽象

为数据访问创建一个单独的抽象层对于“非关系型数据库”来说是必须的。它可以带来多方面的好处。首先，应用开发者可以与底层解决方案的细节完全隔离开来。这对于技术方面的伸缩性带来了好处。同时未来如果需要更改底层的解决方案也很方便。这也以一个标准的方式满足了多个应用的要求(即去掉了 Join，Group by 等复杂特性的 SQL)。

为性能和伸缩性创建模型

不管选择怎样的解决方案，使用标准技术(比如排队网络模型，分层排队网络等)来对性能和伸缩性进行建模都是高度推荐的。它能够为基本的服务器规划、拓扑以及整体的软件许可证成本，管理运行等提供必要的数据。这将实质上成为所有预算计划的主要参考数据，并对作出决策提供帮助。

构建显式的冗余

要防止数据丢失，除了将数据复制到备份服务器上，没有其它的办法了。尽管许多非关系型

数据库提供自动复制功能，但仍然存在主节点单点失效的风险。因此最好是使用次节点备份，并准备好用于数据恢复和自动数据修复的脚本。出于这样的目的，应当充分的了解目标解决方案的物理数据模型，找出可能的恢复机制备选方案，基于企业的整体需求和实践来对这些选项作出评估。

构建公共数据服务平台

就像公共共享服务的关系型数据库一样，也可以构建非关系型数据库的公共数据服务来促进规模经济效应，满足基础设施和支持的需要。这对于未来进一步演化和更改也有帮助。这可以作为愿望列表上的最终目标，通过中期或长期的努力来达到这一成熟水平。然而，初始阶段就设立这样的远景有助于在整个过程中作出正确的决策。

壮大企业的技术力量

每个组织都有一部分人对于学习新生的和非传统的事物充满热忱。成立这样的小组，并挑选人员(全职的或兼职的)，密切关注这方面的动向，了解问题和挑战，进行前瞻性的思考，能够为使用这些技术的项目提供方向和帮助。同时，这个小组还可以为决策者澄清炒作的疑云，提供来自真实数据的观点。

建立与产品社区的关系

选择了产品之后，与产品社区建立起良好的关系对于双方的成功都有极大的好处。许多非关系型数据库目前都有十分活跃的社区，非常愿意相互帮助。企业与社区之间的良好合作能给大家带来一个双赢的局面。如能提前对问题和解决方案有了解，那么企业在对某些特性或版本作出决策时就能成竹在胸。反过来，企业又能对产品特性的路线图产生影响作用，这对他们自身和社区都是有利的。另一方面，社区也能从实际层次的问题中得到反馈，从而丰富和完善产品。来自大型企业的成功案例同样能让他们处于领先。

迭代前进

考虑到非关系型数据库相对的成熟度，风险最小的采用策略就是遵循迭代开发的方法论。构建公共数据服务平台和标准化数据访问抽象不可能是一蹴而就的。相反，通过迭代和面向重构的方式能更好的达到目标。运用不太成熟的技术进行转型的过程，中途改变解决方案也不会太意外的。与此同时，采用敏捷的方式来看待事物，能够帮助建立起一个能从管理和实现

两方面不断吸引改进的开放态度。

然而，在这一问题上实现迭代，非常重要的一点是定义一个决策条件矩阵。比如操作指南(和例子)，来判断一个应用的对象模型是否适合关系型或非关系的范围，对基础设施规划作出指导，列出必需的测试用例等等。

结束语

企业的非关系型数据库采用过程中最大的挑战就是转变决策者的思想观念——让他们相信并非所有的数据/对象都适合关系型数据库。最能证明这一点就是选择合适的用例去尝试非关系型数据库，进而证实在合适的背景下，非关系型数据库是比关系型数据库更有效的解决方案。找到一些“非关键业务”(但能立竿见影的)适合于非关系型数据库的项目。这些项目的成功(甚至失败)都能有助于观念的改变。这也能有助于不断学习如何才能以一种不同的方式来更好的采用非关系型数据库。这些少儿学步般的尝试所作出的努力与投入都是值得的，如果企业想要在将来使用“非关系型数据库”来重塑其信息管理体系的话。

关于作者

Sourav Mazumder 目前是 InfoSys Technologies 的首席技术架构师。他在信息技术领域有 14 年以上的经验。作为 Infosys 技术顾问团的主要成员，Sourav 为 Infosys 在美国、欧洲、澳洲和日本的主要客户，提供保险、电信、银行、零售、安全、交通以及建筑、工程、施工等多个行业的服务。他曾参与 Web 项目的技术架构和路线图定义，SOA 战略实施，国际战略定义，UI 组件化，性能建模，伸缩性分析，非结构化数据管理等等。Sourav 参考的 Infosys 自身的核心银行产品 Finacle，也为他提供了丰富的产品开发经验。Sourav 还曾参与开发 Infosys 的 J2EE 可重用框架，和定义 Infosys 在架构方面和开发定制应用方面的软件工程方法。Sourav 的经历还包括在保证架构合规和开发项目的治理方面的工作。

Sourav 是 iCMG 认证的软件架构师，同时也是 TOGAF 8 认证的执行者。Sourav 最近在 LISA 伯克利全球化会议上发表了演讲。Sourav 关于 SOA 的最新白皮书在社区里十分流行。

Sourav 目前关注 NoSQL，Web 2.0，治理，性能建构和全球化。

原文链接：http://www.infoq.com/cn/articles/nosql-in-the-enterprise

相关内容：

- Google Apps的应用程序市场和即时错误转移服务　Google向开发者提供云存储

推荐文章 | Articles

重构 TekPub——从 ASP.NET MVC 框架迁移到 Ruby on Rails

作者 Robert Bazinet 译者 池建强

TekPub 是一个面向开发人员的站点，致力于为开发人员提供一系列主题的在线培训，主题范围非常广泛，从微软的 O/R Mapping 框架 Microsoft Entity Framework，到如何使用 Ruby on Rails 技术编写自己的日志引擎等内容都有涉及。该网站是由前微软员工 Rob Conery 与 Lounge 的老板 James Avery 创立的。

TekPub 是个很有趣的学习案例，公司开始时使用 ASP.NET MVC 框架，之后很快迁移到了 Ruby on Rails 上。InfoQ 与 Rob 和 James 探讨了这次迁移之旅。

InfoQ：和我们谈谈 **TekPub** 吧，对于哪些不熟悉你们的产品的读者，**TekPub** 意味着什么？

James Avery （简称 **JA**）：TekPub 为程序员提供了高质量的技术视频演示。我们的目标是帮助一些人在几个小时内学习一项新技术，主要方式就是观看一些牛人的演讲视频。他们对演讲的技术非常了解，演讲内容不仅仅覆盖基础知识，还深入到了这些具体技术在真实项目中的应用。与其等几个月才拿到一本很可能已经过时的书，还不如订阅我们的产品立刻获得新技术的提升。我们已经完成 ASP.NET MVC 2 的系列视频，但目前还没有与这个主题相关的书籍。

InfoQ: James Avery 和 **Rob Conery** 有什么来头？

JA：除了 TekPub 之外，我还搞了几个科技创业公司。我运营了 Lounge 和 Ruby Row 广告网络，分别关注.NET 和 Ruby 开发人员。我还帮助运营 DotNetKicks 公司，一个.NET 开发人员的社区网站。我最新的关注点是 Adzerk，这是我自己构建的 Web 服务器，用来有效的运行 Lounge 和 Ruby Row，让其他人来使用。从 90 年代中期我就开始使用 Web，.NET 发布后，我开始转移到微软的技术上。最近我一直再用.NET、Ruby on Rails 和 MongoDB 等，还有任何让我感兴趣的技术。

Rob Conery （简称 **RC**）：我自 1991 年以来一直从事软件行业，做一些数据库，CGI 和 HTML 之类的开发。从 1997 年开始使用 ASP，从那之后一直坚持使用微软技术。2006 年我开始为微软工作，主要职责是帮助人们学习和使用新的 ASP.NET MVC 框架。2009 年我离开了微软开始做些不同的事情（更多的关注开源平台），之后和 James Avery 一起创办了 Tekpub。

InfoQ：能介绍一下你们刚创业时 **TekPub** 的架构吗？

JA：TekPub 的第一个版本是基于 Ruby on Rails 构建的，当时 Rob 花费了整整一个周末的时间。之后我们仔细的进行了讨论，最终决定放弃这个版本。由于我们都很了解 ASP.NET MVC 技术，所以决定用 ASP.NET MVC 来实现 TekPub。我相信，当开始一项新业务时是没有时间尝试和学习新技术的。Rob 和我过去都写过 Rails 应用，但真正使用时，我们都觉得 ASP.NET MVC 比 Rails 好得多。于是我们放弃了 Rails 转而使用 ASP.NET MVC 技术开发网站。后来我们取消该站点并重新构建它，因为它已经变得太复杂。在我们开始进行技术迁移时，TekPub 的版本是 3，主要技术是 ASP.NET MVC，C#和 MS SQL Server。

InfoQ：从网站使用者的角度你遇到了什么样的挑战？从网站管理者的角度呢？

RC：最初，或者说网站运营的第一天，我们就遇到了网络带宽的问题。我们在 Twitter 上宣布网站开张，这直接导致我们的 ISP 由于带宽的需求被淘汰。他们毫不夸张的让工程师"坐在开关上"来保证网站的正常运行，但最终网站还是停了。之后两小时内我把我们所有免费的内容都放到了亚马逊的 S3 上，这对我们帮助非常大。

还有，一小部分人不喜欢 Silverlight，不愿意安装它。我们在 Reddit 上投放了广告，人们却简单的认为我们的网站是微软资助的一个什么东西——这是和我们的目标背离的。对我们来说这是个大问题。

JA：ASP.NET MVC 运行的很好，Windows 架构也不错。事实上我们在这个领域没碰到什么麻烦。主要的挑战来自于我们决定使用 Silverlight 播放流媒体。很多用户不愿意安装 Silverlight，这给我们带来很大的困扰。与 Rob 确认后，我们不得不迁移到 Flash 技术。迁移到 Flash 之后还没有一个人 抱怨过。我们希望 HTML5 很快面世。

InfoQ：架构如何应对用户需求？

RC：我们从来没有遇到底层框架的问题——它处理的很好。这并不是一个负载非常大的网站（在功能方面），所以我们从没真正遇到那方面的问题。

InfoQ：既然平台运行的非常好，为什么要做架构的改变呢？

RC：成本。我们加入了微软的 BizSpark 计划， 而且它的确给了我们一个很棒的起点，但是

随着项目的发展，我们发现，3 年以后，从我们使用的数据库到开发环境，每一件事情都需要付费。而且我们很可能需要 为 Silverlight（使用流媒体）提供一个分离的服务器，用来播放视频，这样就需要再支付一个使用许可的费用——此外，为了使视频流平滑输出，我们还需要买媒体编码器。

对于大公司来说，这的确不算什么，但当我们坐下来看帐单时——噢，这可是个 5 位数啊。最终我们发现，根据我们现有的业务规模，是无法承担这笔费用的。

不仅仅如此，James 和我都清楚 Rails 非常好用。我们意识到可以把所有事情都推到云端，只需支付很低的价格就可以获得很好的流媒体和吞吐量。

JA：正如 Rob 提到的，成本压力是其中一个因素，BizSpark 是很好，但却像一个定时炸弹。我觉得比成本更重要的推动因素是每天我们希望使用什么技术。ASP.NET MVC 和.NET 技术在某些领域是有缺陷的，而这些领域的工作对我们非常重要。在.NET 上做测试就并不让人满意，你不得不大费周章才能以正确的方式设计你的应用，处理测试，而且编写测试本身也不像其他语言那么清晰和有用。另一个问题是部署，虽然有很多种方式可以处理，但确实不如 Capistrano 好用。

InfoQ：那么现在 **TekPub** 平台是什么样子的？

RC：我们迁移到 Rails 2.3.5，使用了针对 MongoDB 的 MongoMapper 技术。我们做了一个报告设置，用 MySQL 来跟踪反馈使用 DataMapper 的情况。我们还植入了 New Relic RPM，来跟踪我们的站点和健康情况——所有的这些，平均成本仅仅是我们使用 BizSpark 的 1%。

JA：这解决了我们所有与使用许可有关的问题，我们每月需要为 Amazon EC2 平台上的一个 Ubuntu 实例支付$80，这是保留实例后的费用。Rob 和我都很喜欢这种技术，我们使用 Cucumber 做了大量测试，用 Capistrano 部署变得非常简单容易。

InfoQ: 能给我们讲讲你们的技术架构细节吗？

RC：使用 Rails 2.3.5 及其对应的 MongoMapper/MongoDB，使用 MySQL 5.2 和 O/R Mapping 工具 DataMapper。我们还用了 New Relic RPM 工具，非常棒。

JA：我们选择使用 Rails 2.3.5 和 Ruby 1.8.7，通过 Passenger 运 行 Rails 应用。在数据库方面，我们通过使用优秀的 MongoMapper gem 访问 MongoDB 1.3.4。我们还通过 DataMapper 实现 MySQL 5.1 的数据访问。我们并不使用很多其他的 Gem，只是用 Pony 发邮件，rpx_now 访问 RPX 做身份验证。同时我俩还都是 HAML 的爱好者，所有的界面都是基于 HAML 实现的。

InfoQ：你们的 **EC2** 怎么配置的？

JA：所有的内容都是托管在云端，相当于一个比较小的服务器，也就是一个单独的 EC2 实例（虚拟双核，2 个计算单元，7G 内存）。在这个实例上运行着 Ubuntu 9.1（karmic）和 Apache。我希望未来有更多的资源，你知道，运行在 EC2 上，想增加资源是非常容易的。

InfoQ：为什么选择 **Ruby on Rails**？

RC：James 和我都知道，插件的世界是非常吸引人的。例如 New Relic，简直是上帝送来的礼物。我不担心服务器挂掉，而且我能看到所有代码的"瓶颈点"。他们还能跟踪最高发生频率的问题，甚至会提出改进意见。

MongoMapper 是一个令人难以置信的数据工具，而 MongoDB 本身的速度比闪电还快。Rails 的平台已经成熟到了这样的地步，几乎难以说服自己*不*使用它。

JA：我们遇到的所有问题（使用许可、测试和部署）都得到了很好的解决。我们还能使用一些变通的办法，例如写自己的部署框架等。随之而来的是我们都很享受与 Rails 工作，我们喜欢 Rails 社区，还有很多工具和类库可用。经过迁移之后最棒的地方就是，这一切让我们变的非常高兴。

InfoQ：相对很多传统数据库例如 **MySQL** 或者 **PostgreSQL**，你们为什么选择 **MongoDB**？、

RC：速度和扩展性。使用 MongoDB 非常简单——不用担心迁移，非常灵活。而且它不可思议的快——这一点对用户带来的感受是巨大的。

JA：事实上我们都用了，我们用 MongoDB 做那些它最擅长的（灵活存储产品、用户和订单等信息），同样我们也这么使用 MySQL，我们用 MySQL 提供正在发生的持久事务日志。

InfoQ：完成了新的设计和重写应用之后，最直接的好处是什么？

RC: 最直接的——第一件事就是人们发现网站变快了。他们还注意到一个"更严格"的设计，诚实的说，确实有些繁琐，但它让我们能更多的关注服务器端。

不仅仅如此——当出现问题时，我们可以在几秒钟之内消除它。这是由于基于 Capistrano 的部署是如此简单。问题的修复、推出——仅需几秒。

我们的测试套件使用的非常好——使用 Cucumber 做一些事情简直是一种享受，但用 ASP.NET MVC 就会很困难。例如——使用 ASP 测试 PayPal 的支付接口时就会很麻烦。

基于 Rails/Cucumber/Webrat，在这样的框架下要做的就是填空并提交——然后确认所有事情都是按照计划执行的。我想我们的用户会看到很多功能在按照预期的方式运行——这确实很棒。

最直接的好处是我们想做一些变化时，可以快速完成，并且这种响应速度直接反应到了产品上。在原来基于 ASP.NET MVC 技术的网站上，我经常会写些 SQL 脚本来处理一些复杂的支持需求，比如改变某人的 OpenID，或合并帐户什么的。但现在我只需花 30-60 分钟的时间写了几行的 Ruby 脚本就能完成。

在重写网站过程中另一个重大的改变之一是如何处理用户的文件。过去我们从自己的服务器或运行在 EC2 上的 Wowza 媒体服务器上以流的方式处理文件。这两种方案都可以，但是太贵了，而且在其他国家就运行的没那么好了。这次我们开始重新设计亚马逊服务，提供了从云端和云端流来处理私有内容的能力。现在我们的全部内容都是通过云端处理，并使用了安全签名的 URL，这就意味着世界上任何地方的用户都可以进行高速下载。这也节省了我们的成本，因为我们只需支付使用带宽的费用，不需要额外的设备或许可费用。

InfoQ: 在你们当前的实现中，**TDD** 测试方法包含哪些内容，引入模拟对象了吗？

JA：我们用 Cucumber 和 Pickle 实现测试驱动开发。这方面 Rob 应该比我更清楚。

RC：没有进行模拟测试，都是 BDD（行为驱动开发）。我痛恨那种拖拉的感觉，就像我需要做“代码覆盖率”一样——事实上我们有个商业应用在运行，我更需要确认用户获得良好的体验，这对我来说就是 BDD。Cucumber 扮演了一个真正的重要角色——就像 Pickle 和 Factory Girl 做的那样——它与 MongoMapper 配合的好极了，对比我在微软框架下做的和我现在能做的事情，这让我高兴的笑了好几次。我几乎完全在考虑业务——甚至没有注意到测试方法这回事。

InfoQ: 你们现在的测试方法和以前在微软使用的方法有什么不同？

RC：100%不同。测试微软的东西是极其痛苦的，当时你很穷，还有三个月钱就花完了，在这种情况下，测试就是变成了一个很难判断的的东西。不可否认——我为我们的第一次修改几乎写了一吨的垃圾。如果你没有客户，那么你的应用系统好不好用就无关紧要，我真正需要确定的是不出现那些不可预期的错误。所以我进行了一些测试，但并没有得到想要的结果。我们对 PayPal 的东西进行了大量地测试，这是你必须要做的。当开始使用他们的 API 时，我在 他们的沙箱环境中折腾了整整三天，还是会不断的出问题。

另一边呢——感觉是黑夜和白天的对比。现在我们覆盖了所有的行为类型，我想 James 甚至抽时间写了一些。

JA：对微软产品的测试总是像打一场战争，主要是因为 C#和静态语言对测试的支持不是很好。在.NET 上测试时你总是需要在各种循环中跳来跳去。话虽这么说，.NET 的 SpecFlow 框架看上去还是很乐观，但我对它能解决.NET 的测试方面的所有问题持怀疑态度。

InfoQ: 在微软框架上做测试会有多少困难？为什么会这样？

RC：语言和工具造成的。假如微软对这一点想的多一些，测试可能会更容易一些。使用 RSpec 进行测试，没有其他语言比 Ruby 做得更好了——它让测试变得非常容易。微软可以利用 DLR 实现这些功能……但是他们没有这么做。因为这不符合".NET Story"，那么好吧，这是他们的商业决策，我们这次的迁移也是一样。

InfoQ: 在这次全面的重新设计中，你们有什么经验教训吗？

RC：没什么——我想 James 和我都知道，我们到了需要调整和重写的那个点。这是我的第三次创业，对 James 来说应该是第五次。开始时我们经过很长的时间讨论采用什么技术。我们都同意我们要"跟随你所知道的，并且去实现它"——于是我们就做了。

我们很幸运把握住了这个机会——然后我们进入了这样一个状态："好吧，让我们开始为之后的 3-5 年规划，进入可控的增长模式"——这就是我们现在的状态。我们不想做得很大，我们不想引入 IT 人员或一组开发人员。我们想自己做而且保持小的规模——重点放在了我们的创意上。

对于我们这样的两个家伙来说，Rails 超级简单，易于维护。我们有世界上最可靠的支撑（可自动扩展的亚马逊 EC2），我们对正在做的事情非常满意，获得了很多乐趣。

JA：我想大部分人看到这个故事，会认为我们应该从开始就使用 Rails，事实上我很高兴我们没有那么做。开始创业时我想我们要相信自己的直觉，哪些是当时我们认为最好的，我们应该关心所有的业务问题，而不是使用什么工具。现在业务已经正常运转，在这样一个坚实的基础之上，我们就有了很多美妙的时光，释放内心的渴望做一些好玩的事情。平台移植让我们享受了很多乐趣。

InfoQ: 你们现在的架构有什么缺点吗？如果今天开始重新做一遍的话，有什么改变吗？

RC：从我的观点来看没什么——我爱现在的一切。对我来说这很不可思议——我在很长一段时间内曾是一名微软员工，并被扣上了这样一顶帽子……其实这对我们来说并不意味着什么。我非常喜欢硬件成本的可扩展性——我想在很长时间之内我们不再需要其他的硬件方案了。我们的服务器非常灵活，完全根据需求动态扩展。使用微软技术你就没法做到这些（据我所知），如果你想搞一台独立的机器，那你就需要支付一大笔款项。

JA：针对现有架构，我唯一想做的改变就是把 MongoDB 从我们的单一服务器上分离出来，同时运行两个 MongoDB 对我们来说是有意义的，但现在这部分功能被废弃了，对我们来说更有意义的事就是等 MongoDB 开发新的配对策略。但是考虑到额外的两个虚拟机的成本，

以及我们的业务在现有的单一服务器上运行良好，我们暂时还不准备这么做。

InfoQ: 感谢 **Rob** 和 **James** 抽时间接受我们的访谈。

关于 TekPub 的更多信息，请参考公司网站。

原文链接：http://www.infoq.com/cn/articles/architecting-tekpub

相关内容：

- 采访.NET版Cucumber的创始人Aslak Hellesøy
- 采访与书摘：ASP.NET MVC in Action融合思想：深入探索S#arp架构
- ASP.NET MVC 2特性汇总
- 用Rails创建高质量Web应用
- Rubinius 1.0 驾临

推荐文章 | Articles

SOA 十诫

作者 Hans Wierenga 译者 吴宇

面向服务的架构（SOA）是一种组织信息处理的方法。各系统为协同工作在各方面达成了协议，SOA 通过减少这些协议的数量，能够降低信息系统互操作性的成本。如果 SOA 能得到大范围的应用，系统将呈现与现在截然不同的前景，这就好比当今货运行业有别于集装箱出现前的货运业时代一般。然而，目前的应用方式却导致了额外的开支却并未体现出这些互操作性的优势。将适用于数据库时代的范式应用于 SOA 中，会招致反效果，往往是愚蠢的，有时甚至是十分危险的设计。这些模式必须由新的思想和行为方式所替代，以确保 SOA 朝着接口更简单、IT 方案更优化以及项目更可控的方向发展。这一点可以通过遵守以下十大戒条来实现。

引言

SOA 的潜在影响

面向服务的架构（SOA）是一种组织信息处理的方法。这种方法以服务的形式描述所有交互活动，服务请求者请求代理完成某些处理，代理确保处理得以完成并将处理结果反馈给服务请求者。这种思维方式可以应用于业务级别，以描述各组织机构之间的交互；应用于功能级别，以描述组成业务流程的活动的交互方式；应用到信息系统级别，以描述系统及系统各部分的交互方式。每个级别的准则都是相同的：代理完成所需工作的方式与请求者无关，乃至与是否完全自动、完全人工亦或两者兼具都无关系。哪怕代理将部分或者甚至全部工作外包给其他代理完成也与请求者无关。所有请求者所需关注的是与代理就以下方面达成一致：请求及响应应该如何制定，以及服务的效果如何。

SOA 被大肆宣扬为一种具有巨大潜力的范式，能够降低系统发展、测试及维护的成本。特别需要指出的是，SOA 承诺可以通过大幅度减少达成协议的因素的数量，从而降低信息系统各模块协同工作的成本。采用 SOA，诸如像计算平台和数据格式之间的差别造成的系统间通信屏障会较采用早期的范式要少得多。这使得更大范围上的协作变得可能，因为它减少了障碍，

使系统设计师们不必被强行要求相互达成一致，就此而言，也使得系统配置员之间不必被强行要求达成一致。如果这种承诺可以实现，其结果将会是革命性的。就像汽车改变了城市区域形态，集装箱运输革新了货运行业，以及交易费用的降低发展了现代自由市场经济，SOA 将开启新的合作模式。当 SOA 主导我们应用 IT 的方式，系统前景将与今日迥然不同，好比围绕汽车来设计规划的城市和围绕火车来设计规划的城市截然不同一般。对我们之中那些思维受限于目前技术的人来说，SOA 可以产生多大的不同是难以想象的。然而 SOA 所提供的灵活性优势就好比汽车胜过火车一样：即便火车可以被制造跑得和汽车一样快，火车还是绝不可能像汽车那样提供门到门的运输服务。把火车站安置在每个车道的尽头，亦或甚至把铁轨铺设在每条道路上都是根本不现实的。

为何此影响尚未实现

为获取新范式带来的好处，我们必须好好利用其所能提供的各种新的可能性。遗憾的是，目前围绕 SOA 的言过其实的宣传对这些可能性的提及是言之甚少。大部分讨论似乎都关注于如何利用 SOA 帮助单独信息系统更快速地开发。然而，这并非 SOA 所能创造的最大价值之处。事实上，SOA 是否真正能够改进以往的方法， 即各个功能点通过某一共同的数据池（通常是以数据库的方式实现）来实现交互，还存在争议。使用 SOA 来构建单独、孤立的信息系统就像使用集装箱在加工厂附近搬运货物一样：当然，它限定了内部物流的顺序和组织，但是集装箱更多的是挡了道路而非提供帮助。SOA 使信息系统间达到更好的互操作性，就像集装箱促使了运输商之间的互操作性一样。那是一种重要优势，因为从需求确定到信息系统可操作之间的时间周期通常很大程度上是由互操作性决定的。要使某一信息系统能够与其操作环境中的其他系统一起工作，那将会花费比重新构建这样一个系统更多的时间和精力。

关注于 SOA 在信息系统内部而非各系统之间的使用是情况更加恶化的征兆：因为看起来 SOA 是一种全新的处理我们一直以来所做的事情的方式，我们无法直接获取它所带来的收益。SOA 概念和技术正为目前的系统开发范式所利用。这些范式还是数据库时代的开发产物，同时也带有数据库技术的一些限制。在这些限制下应用 SOA 将会导致额外的开支，而不能获得额外的收益。然而，这些“数据库化”的范式是如此普遍和有害以至于我们常常忽略了它们对我们的思维影响有多大。它们是如此根深蒂固，以至于我们会不自觉将其视作常理。遗憾的是，这样通常招致相反效果，常常是愚蠢的，有时甚至是相当危险的。它们导致了一种不好的方案， 这种方案集合了数据库时代的缺点以及 SOA 不好的一面，而又不能体现 SOA 必定提供的优点。

需要改变什么

SOA 范式有其自身的常识守则，较之数据库范式的守则截然不同。基本戒律有十项。前五项关于如何简化事物，使其比数据库化的范式要求更加简化——从坚持要点意义上更加简化。如果我们以此种方式简化事物，同一问题的不同解决方案相互间协作的可能性将大大提升。接下来的四项关于使 IT 解决方案优于同等数据库解决方案，这是通过阻止那些戴着有色眼镜、惯于数据库思维方式的人开发出无效解决方案来实现的。最后一项关于如何使 IT 更可控，尤其是组织系统开发以降低项目复杂度和风险。SOA 使这些成为了可能——同样也是必须的——因为它让更多的功能交付成为基础架构。

简化之

理论

在高空杂技表演中，高效的合作的基础是每个空中飞人演员完全默契地配合，对方会及时地在每个时点出现。一些空中飞人演员非常自信，他们经常蒙着眼进行表演。他们能够接到彼此正因为他们确定在某个特定时刻对方只可能出现在一个可能位置。

成功应用 SOA 以达到最大化的协作性与高空杂技表演非常相似。互操作性要求关于如何进行交流的解决方案从数以百万计减少到只有一个，交互双方都可以依赖该方案。这并不意味着其他方案有问题：这好比我们既可以靠左行驶也可以靠右行驶，但重要的是我们必须都靠同一边行驶。

当只有一方执行服务，一方接受服务时，只要双方协议好，具体使用哪种方案其实并没有太大区别。双方中任一方的特异性可以决定最终方案，无需做更多的沟通努力。毕竟，无论使用哪种方案，这些特异性总要被处理。但是如果是多方请求服务或者多方执行服务，那将呈现不一样的情景。此时使通信方案精简非常重要，如此，各方必须处理各自的特异性，无需面对另一方。

将信息通信与外科移植手术对比很能说明问题。成功的移植手术，要将一个人身上的器官移植到另一个人身上，要求该移植器官必须在多方面与接收者匹配，而其中大部分匹配因素与该器官的生物功能无关。换句话说，被移植的器官必须和接受者本身有相同的特征。因此，我们的器官不能像拼搭乐高积木一样随便被移植。目前的信息通信恰恰如此。当一个信息系统为另一个系统提供服务时，它们必须在很多方面达成一致。它们必须使用相同的词汇（元数据）、相同的由一方调用而另一方执行的功能集、对于每个功能请求应答数据内容的相同

期望、相同的编码系统、相同的技术通讯协议、相同的用于信息传递的寻址模式、兼容的可预期的响应速率、兼容的确保消息不被丢失的技术、兼容的认证机制以确保双方安全通信而不是与冒名者通信、兼容的加密技术以及密钥管理以确保消息不被窃听或者篡改等等。为了促进互操作性，必须确保参与各方从彼此独立制定各自标准变为形成兼容的标准规范。只有当一些非常严谨的规则得到遵守时这才有实现的可能，接口才能减至精要。如此一来特异性将无容身之地。

严谨的规则都关注于如何使得服务接口更为简单。我们规定的越少，争议的余地就越小。

1.不了解你不需要了解的

你不需要去了解的东西不会伤害到你

SOA 范式的本质在于使得合作各方或系统之间达成最少限度的协议却可以实现最大程度的合作。这是一种巨大的优势，因为任何你不需要了解的东西既不需要被测试也不需要被维护。你不需要去了解的东西不会伤害到你。假设 40%的系统开发成本用于测试上，而高达 80%的信息系统生命周期的成本被花费到了系统维护阶 段，任何 SOA 范式让你无需了解的东西都代表了你能节省的金钱。

元数据

你最不需要了解的就是结构、含义以及容许值——这些元数据——不会被系统中筛选、排序或执行计算的逻辑使用的数据。你并不需要去了解这些，因为 SOA 技术使得数据和元数据同时出现。你的系统可以实时解读元数据，所以如果你要做的仅仅是获取、呈现或传送相应的数据，你完全不需要为系统构建元数据知识。在有相当精密的表示（presentation）功能的帮助下，它甚至可以为用户实现各种各样特定的筛选及计算，且只使用与已有数据同时提供的元数据，而不是内部构建元数据。

通过解读数据相应的元数据，而不是把元数据构建到系统中，你的系统不需要随元数据的改变而改变。需要改变的仅仅是源系统。想想如果遵守该守则，你能在开发、测试和维护上节省多少金钱！记住，在两个系统上做更改，平均来说，其复杂度是在单个系统做更改的四倍，因为其中包含了所有各方的协作。

对于很多面对客户的系统来说，表示以及特定筛选功能基本是其主要的功能。这些系统只针对最基本的客户数据要求内部构建元数据。这并不包括当前或过去的订单、客户通讯录、照

片、信函以及任何可用于展示的其他数据，所有这些数据都可以用一种不需要这些数据本质相关知识的方式进行表示，内建于系统中。

技术

许多你不需要了解的事情是与技术相关的。有了 SOA，你不需要了解你正在接口的系统是否采用“软件即服务”（Software-as- aservice），不需要了解实施该系统的计算机安放在何处，是哪种类型的计算机或者运行于何种操作系统，防火墙是如何配置，使用的是哪种数据库管理系 统，亦或可以使用哪种交易管理系统。其他你不需要了解的事情是与你所通信的系统内部相关的。尤其是，你不需要去了解任何用于内部数据存储的元数据，因为任何其他系统需要同 XSD 一致的转换都是其自身的问题，而不是你的。

即便如此，使用 SOA 进行通信的各方必须达成一致的技术相关的标准还有很多选择。特别是有很多与 Web 服务相关的那些标准，SOA 从业者将其统称为 WS-*标准（*指可以使用很多可能的标签替换）。在一定程度上，这些标准提出得很恰当，因为 SOA 社区并没有满足于不去了解它不需要了解的东西；本文这个白皮书给出了一些指导以期降低由这些标准引起的问题的影响。遵守这些指导，SOA 需要的先期协议将比其他方法要少得多。

设计稳定的接口

如果想获取 SOA 提供的种种好处，不去了解你不需要了解的东西会成为你的习惯。请铭记这点！比如说，当设计一个订货服务时，请记住服务请求者只需要知道，当他需要货物的时候该货物是否会有货，而不需要去了解当前的库存量。如果你的程序调用某一安全服务以判断请求活动是否被授权，不要在系统内构建任何超过其所需服务工作的知识。例如，如果安全服务需要使用输入到程序的安全证书，唯一必须做的就是传递该证书！对你来说，它们只是被封装在输入消息中的单个数据项。该证书是否是格式完整的 XML 也不要去验证。如果，由于某些只有那些负责安全的小鬼知道的原因，他们选择了违背标准的 SOA 操作或对证书进行了加密，那么这是他们的问题，不是你的。如果他们改变了任何与证书相关的东西，你的程序不应该为此做任何改变或调整。任何你不需要了解的东西不会伤害到你。当然了，除非你硬要去了解它，在这种情况下如果你们不想在 SOA 上浪费时间的话，其他人可能最好离远点儿。

不去了解那些你不需要了解的东西可能比你想象的要难。如果你开发专门用于与你通信的信息系统的信息检索服务，你的思路已经不对了，因为你已经把其他系统的知识归并到系统中了。需求中的任何更改将会迫使双方系统都作出更改。通常来讲，比较好的方式是采用数量

有限的检索服务暴露系统数据，当检索服务结合在一起使用时，它们涵盖了所有相关服务的信息检索需求。例如，某个产品数据库可能通过好几个服务分别暴露出去：一个简单的仅包含编码、描述、部门以及产品定价的服务、一个暴露出所有该产品财务方面信息的服务，以及一个暴露出所有该产品物流方面信息的服务。许多系统仅需简单服务即可得到满足，大部分只需要部分数据而非全部，或财务或物流的服务，而有一些两者都需要，但此外没有任何一个需要特别接口的系统。这种工作方式被称为麦当劳方式：客户从标准产品中搭配出自己需要的产品。支持这种方式并不困难，因为不管怎样你都需要这些服务去支持面向客户的程序。你甚至可以用这种方式来支持非常特别的信息需求，因为那些不需要的数据可以在消费前就过滤掉。如果不想在巨无霸汉堡中放小黄瓜，扔掉它就可以了！这种方式的基本思路是提供过多的信息要比提供过少的信息遇到的问题少，因为接收方系统可以很容易通过程序过滤掉不需要的信息，但是如果缺少信息那就麻烦了。

不去了解你不需要了解的东西也会使得为支持业务流程所需的信息交互大大简化。在 SOA 的范式中，当你请求另一个代理为你做一些事，那就是你所需要做的全部。你不需要为代理提供可能有助于完成任务的或者是其必需的额外信息。在点菜时，确保有用于这道菜的原料是厨师的职责。你说出想要的东西，然后就可以静候佳音了。反过来，代理会使用信息检索服务来向你咨询所有信息，但是检索什么、何时检索以及从何检索，这些问题都应该由他来决定，你无须去了解，更不用将该知识归并至你的系统中。这样，在他那一端的更改几乎不需要你这边作出更改。比如说，如果他决定停止维护对你数据的拷贝，你什么更改都不需要做。

当然，不去了解你不需要了解的事情确实会导致效率低下。原本只需要一次交换即可实现的操作现在将需要多个步骤。麦当劳方式常常会导致原本一个服务即可满足却提供了多个服务的情况，另一边却还在检索信息，而这些信息又常常是冗余而非十分必要。总会出现一些情形，可以通过好的商业意识来优化这些通信模式。也会有很多场合你会想要优化用户接口，那也只是因为当前的表示设备并不擅长提供给用户吸引人的界面。但是在你优化之前，请考虑你会失去什么样的灵活性。另外绝不要想去优化那些尚未稳定的功能需求。

2.不要了解你还不能了解的事情

为时过早的规范冻结

数据库范式中，一个真正的难题在于：它要求在你还未足够了解并有能力去确定数据的具体结构前，就去做这件事。因为它们忽视了一个生活中简单的事实：只有当用户看到他们不想

看到的东西时，他们才知道其真正想要的是什么。

其工作原理是这样：一旦完成了数据结构的设计，任何后续修改都会引起杂乱的数据库转换，除此之外每个访问该数据库的系统也会改变。所有这些改变必须都协调好，当所有对数据库的操作都限制在单个系统时候，这种协调是很困难的，但如果有多个系统都在操作，那就更难了，尤其是：如果其中有些系统被不受你控制的参与方管理的时候。事实上，在系统开发阶段做这些更改就已经很成问题了。其后果是，数据库设计早在系统开发阶段就被冻结，然后数据分析师们再去竭力修正这些设计。

现在的问题是数据分析师们面临着不可能完成的工作。他们必须在用户理解这个设计（且不说赞赏这个设计实际应用如何）前就确定该设计。只有在过了很久之后——系统已经构建好之后——用户才能对该系统有所体会并对其是否满足自己的需求作出评估。如果此时发现数据结构上有任何大问题，要想修复就太晚了。

SOA 原型法

你可能会问："SOA 是怎么个不同寻常呢？"说到底，难道 SOA 不像数据库范式那样一样需要结构化数据吗？这个问题的简单答案：不管请求数据时，被人工代理处理还是被自动化系统处理，SOA 都是管用的，并且就算数据没有被最优结构化，人们还是可以解读它。比如说，用户可以判断信件是否正在被发送至另一个国家，无论信件的地址是用行一、行二、行三和行四来表示的，而信息系统需要至少"国家"来作为数据结构可识别的一部分。

仔细回答这个问题就包括了对 SOA 原型法的讨论，这种讨论包括以下几方面：

1. 识别将被构建到系统中的元数据。把它放在主命名空间中，用传统方式根据其结构把数据存储到数据库管理系统（DBMS）。例如，用户信息，这个元数据可能由用户 ID 和用户名构成。因为这个元数据被构建到系统中，所以为了使用这些字段而把逻辑也构建到系统中是完全有可能的，比如通过用户 ID 检索记录并基于用户名排序和筛选记录。

2. 对每一条数据库记录，把不包含在主命名空间的所有数据放到一个单独的 XML 字符串中，该字符串作为一个单独的字段添加到数据库记录中。对每个 XML 字符串，构建一个二级命名空间，开发 XSD，同时添加一个单独数据项到主命名空间。对用户记录的初步实现来说，这个字符串可能包括地址行一、行二、行三和行四的元数据。用户记录本身的 XSD 将会包括三个字段的元数据：用户 ID、用户名和以 XML 字符串包括的附加的用户相关数据。附加用户数据的 XSD 包括每条地址行的元数据。

3. 使用基于 XSD 的逻辑来为数据库记录的各相关层次获取及展示所有数据，这可以通过给

每个 XML 字符串（包含对主记录的字符串）增加一个标准验证服务来实现。如有可能，利用某种工具来产生使用 XSD 的接口而不要自己去编程。该工具必须使用二级 XML 字符串所包含的元数据来将其解包，并且必须同时使用主、次两级 XSD 的逻辑来获取新记录。其结果就是一个可运行的原型，用户可用以评估预期使用的系统。

4. 要尽量适应和修改基于 XSD 的逻辑以及验证服务直到用户对系统满意为止。就拿用户记录来说，一个新的针对字符串的 XSD 可包括以下元数据：街道地址、邮编、县市、以及国家。使用旧 XSD 存储的数据仍会正确显示，所以无需立即将其转换。而使用新 XSD 将会获取新的记录。

5. 一旦用户和你确信元数据已经稳定了，那么请考虑把数据迁移到传统的数据结构和主命名空间。

当然，SOA 原型法并不总是必要的。当需求受限于如此多的不确定性时，请使用原型法，这样先做原型然后再将其稳定比较经济。但是请记住这个肯定要比数据库化原型法要简单的多。在数据库化的时代，原型法只是可有可无，但是在 SOA 的时代，它便是家常便饭。不要将设计细节固定，除非你确定它们是正确的；就是这 么简单。

3.除了要求的不要多做

产品驱动的服务分解

SOA 主要的优势就是它是一个能被转化成技术的业务概念，不像数据库世界里那样，技术概念总想试图跟上业务的步伐。在 SOA 中，每个服务必须明确地增加价值，不只是在抽象意义层面上，而更具体地要针对那些调用方而言。发生的一切之所以这样发生了，是因为服务请求者要求其如此。通过不实现任何服务请求者没有明确要求的东西，服务可以被限定在它们的核心功能上。如果 SOA 中所有的参与者都积极这样去做，那么将会使互操作性大大增加。

SOA 中最高层次的服务是业务交易：某个客户下了一个货物或服务的订单——除非订单被供应商拒绝——否则这就开始了一个订单交付。在 SOA 范式中，任何从接受订单到订单交付之间所发生的都可以也应该以服务的形式实现。为实现订单交付而要求的每个中间产品或状态，供应商会要求某些人或某些组织机构—— 不一定是供应商内部的组织——来执行交付。然后这些中间服务自身又可以分解成多个服务，一直到增加价值的基本组织层面。

层次服务分解是 SOA 范式中非常重要的一部分，因为它使服务交付通过服务水平协议的方

式管理。它可以确保每个服务都有专人负责，并且服务消费者们知道他们会得到什么。

通用的中间产品和通用的服务

服务分解都是关于产品的。只有当组成服务的那些子服务不能立即执行时流程才会出现。例如，如果决定是否接受客户订单的服务要求：先执行判断订单价值的子服务，再执行确定客户信用状态的子服务，那么实现该服务就包含了一个流程。每个这样的流程只与一个服务相关。如果你发现有个流程不仅限于单个这样的服务，那么你很有可能是忘了把客户的初始需求建模为服务了。

有可能出现这样的情况，相同的中间产品——当然还有相同的服务——可能会被不同的高级别产品需要。比如，那些要求明显区分交付过程的商品，在中间产品的环节，其中的差别一般来说几乎微乎其微，中间产品其实也是由客户来支付——为了从客户那里赚取利润，我们需要金额、日期以及让我们可以冠冕堂皇地向客户要求支付的条款，实际是什么则并不用去理会。在这种情况下，正常的基准是如果存在通用的中间产品，那么它应该由单个服务来实现，而这个服务能够被多个高级别的服务调用。

这种服务只有结果是重要的——不是开始——因为收集对交付服务有用的信息是服务本身的一部分，而不是请求服务源头的一部分。需要注意的是：是否需要单个服务是由业务决定的，和信息技术没有任何关系。如果不管是现在还是相关的未来，交付相同的中间产品都是切实有用的，那么就应该有单个服务。除非在现在或相关的未来有商业力量发挥作用使中间产品发生结构性分化，那我们最好不要为每组需求分别提供相应服务。然而事实偏偏相反。在 SOA，事物是由相同走向不同，而在数据库化的方法中，事物则是由不同走向相同。

带有多种行为的通用服务

你可能会问，如果需要两种完全不同的行为而你只为其提供一种服务，这样做的意义何在呢？难道我们还在被同样困扰数据库世界的“一刀切”做法限制？比如，如果我们出售两种类型的产品，其中一种使用固定价格而另一种则根据某些复杂的公式计算出变动价格，为什么不用两种结算服务，为每种特定的产品各订制一种呢？

这些问题很好，但问错了地方。它们是好问题，那是因为任何不能充分应对发生在问题域内变化的设计方法注定会失败。但是它们问在了错误的地方，这是因为根据 问题域中的变化来调整方案的灵活性应该构建到服务中，而不是围绕着服务来构建。就拿结算服务来说，每次调用一种方法时，它应该决定这两种算法应该使用哪一 种。那样的话，如果引入第三种

结算算法，那么只有结算服务需要去做调整。请注意这并不意味着结算服务必须被设计成某种“万能”机器；它同样可以很好地为每 个算法分别调用相应的子服务。这种选择——介于多功能解决方案和包含不同组件的框架之间——是一种可以根据服务来决定的选择，因为它不需要被服务请求者知道。在 SOA 中，这种选择更常见，但目前的做法则常常导致产生框架解决方案，因为需要多功能方案的感觉在很大程度上是由于不能从那些需要执行的信息里识别 出服务请求造成的。这种需要高素质专家来实现的带有如此多参数的以一应十的多功能方案的日子不会长久了。

坚持要点

流程驱动方法也可以使我们分清我们是代表服务请求者做事还是为自己做事。以服务请求者身份做的事应该作为服务的一部分来执行，而其余的就不应该了。通过这种区分，我们可以让服务尽可能简单，这样可以在不需要改变各式各样其他东西的情况下替换掉该服务。举例来说，我们会生产产品来满足外部服务请求，但是维护簿记系统是为了满足我们自己的需求，而不是请求者的。如果要开发一项服务将客户订单转变成制造活动及账目簿记，那么只要我们实现一个新的簿记系统，我们就要去修改一次这个服务。这听起来似乎还不是太糟糕，但试想一下所有事情都是为我们自己而做，那就太糟糕了。包含最新数据的数据仓库的簿记、日志、储存、员工绩效数据的维护：所有这些及其他事情一般都由系统完成，这些系统随组织机构和时间的不同而不同，因此在业务服务中包含这些知识会降低互操作性，增加了用其他系统来实现服务替换的难度。我们可以通过产生通知的方式来避免此类问题，即：如果对这些功能很重要的事情发生了，就发出通知，然后它们可以使用通用服务检索处理事件所需的信息。

另一类不应作为服务一部分执行的业务活动包括那些一旦服务请求被撤销就无法逆转的活动。通常来说，这类活动包括诸如因客户订货导致库存量低于补货水平从而需要订购补给、注册新用户以及更新现有用户信息。这些活动是整个流程中的各个步骤，应使用单独的服务——执行。

当然，这种思维方式可以与数据库范式紧密结合。但并不是其所特有的，因为它与 SOA 有关。结果是，许多 SOA 的实现与数据库化的思维方式背道而驰，而正是这种思维方式激发了他们以自下而上的方式识别服务，而非 SOA 的自上而下方式。在自下而上方式中，原本一开始为某个问题开发的服务也可以为其他问题复用，这多亏了设计它的人对于如何更广泛地应用做了认真的思考。SOA 中，在多个上下文中使用某个服务是缜密设计的结果，而不是靠直觉，并且从设计之初就将所有那些上下文都考虑了进来。说到复用某一服务来交付某一

通用的中间产品就好比说重复使用前门进入房子，不管那扇门是通向客厅、厨房，还是卫生间。你能想象厨房设计师说："真妙！那家伙设计的室外通向客厅的入口正好可以被我用来作为通向厨房的室外入口！"吗？任何谈论"复用"（如复用支付服务）的人都还没有实现它。请注意，这并不是错误的，只是有点奇怪和没有什么启发性罢了。

服务遵从业务

采用 SOA 方式思考问题的一个结果就是 SOA 会使得服务依据其业务意义而非机械的实现来表述。例如，名为增加用户记录的服务是数据库化的，而名为注册新用 户的服务就是 SOA 的，即便这两个服务做的事情完全一样。我们对服务的命名十分重要，因为它告诉我们是谁在请求该服务，以及为什么他要请求这项服务。在这 个特定的例子中，SOA 自上而下的方法会得到一个结论，那就是业务流程需要一个有效的用户注册服务，该服务通过修改现存的注册服务（如果有的话）来完成， 而不是重新自动创建一个新的。在 SOA 中，这是用户注册服务的责任，而数据库化的方法却会把这个责任推到服务请求者身上。类似地，SOA 注册用户服务自身会决定用户 ID 是什么，而数据库化的服务可能就会干脆让服务请求者做这个决定。

汇聚到单个方案

总的看来，SOA 由上至下的思维方式使得很多设计决策只存在一个选项，而数据库化的思考者会把该选项仅仅看作众多可选方案之一。这是 SOA 很重要的一个优势，考虑到 SOA 是以互操作性为导向的，而互操作性要求我们行车时都在同一边行驶而不用去和我们碰到的每辆车去交涉。那些忽视这点的人--比如通过主张 Web 服务只是众多实现跨系统边界 SOA 的一种方式--能够一直成功的机会和那些只考虑下一步的棋手差不多。诚然，总是有比 Web 服务更简单的方法去连接两个系统，但是为了让呼叫中心或输出管理设施能使用相同数据你会怎么做？为了确保数据仓库能够在你把事件从一个系统转换到另一个系统时得到通知，或者事件一发生便能及时通知你的客户，你又会怎么做呢？在 SOA 的世界里，数据和事件必须在多个系统中可用，而 Web 服务是能够确保在低投资和低维护成本的前提下达到这一效果的最有效的方法。

存在这一明显差异的一个领域就是电子数据交换（EDI）。传统的电子数据交换（EDI）技术旨在确定组织之间通信可能需要的信息，并为该信息定义词汇。那和定义某一特定的信息交互不是一回事。比如，你可以使用相同的 EDI 报文来下订单、查询其进展以及修改它。两个组织想通过这些规范进行通信必须坐下来一起就这一词汇的使用方式达成一致。SOA 分离了

这些关注点：词汇由命名空间来处理，而这些命名空间可能会被多个服务使用，每个服务有各自特定的目的。

“SOA 由上至下方式导致更具体的结论”的另一领域是这样一个问题：是什么筑起了一个编排过的流程边界。通常，这个问题会引起无休止的争论。比如，采用输出管理服务给客户发送确认信息是否应被编排为用户流程的一部分，如果是的话，它应该采用即发即弃（fire-and-forget）的方式处理还是应该由输出管理服务来报告动作成功完成？按 SOA 的话讲，所有这些东西都是用户意图的一部分，因此都应该被编排的。其他一些行为，比如考虑到新用户订单的更新数据仓库或者更新总分类，很显然都不是用户意图的一部分，不应该包含在流程编排中，哪怕它们是同一方执行的。

4.不要自己做琐事

通用功能

业务服务应该只包含特定于该服务的那些功能逻辑。它应该把其他功能都委托出去。那样的话，服务自身就可以尽可能简单。这使得设计、测试以及替换服务，如有 必要的话，更容易。这是个基本的数学知识：新软件模块需要匹配的因素越多，那么同价位下的成品软件(COTS)的共性越少，而且若恰好有个方案可用时它也会更贵。如果可以将这些非特定功能委托给标准服务，那么就可以降低需要匹配的因素个数。

服务可以代劳的首要任务是琐事：都是些“家务事”而非真正业务相关的功能。这些琐事天生就是普遍的，换句话说完成这些琐事的方式并不是为支持业务上下文的服务量身定制的。

通用用户接口

当你得知信息系统最不应该做的一大琐事就是管理用户体验时你可能会觉得惊讶。这是一个通用的功能，应该尽可能的采用标准工具来处理。用户体验包括用户可以选择要执行工作项的工作列表，工作项执行的工具——比如通过启动一个用户界面（如果有的话）来关闭已完成的工作项。它包含了用户有可能执行的交易甄选，输入信息屏幕的表示——一般是从 XSD 生成——以及使用和交易相对应的标准验证服务进行验证。它包括了保持当前用户上下文环境，这样他就无需再次输入当前的客户、产品、项目、流程实例或者其他任何东西，而是可以使用这些默认值或者在必要时候重写它们。它包括了和当前用户上下文环境相关的所有文档的介绍。它包含了用户可能需要作出响应的提示对话框。

所有这些东西都可以使用无需包含任何业务知识的工具实现。总的来说，给用户提供一个统一、包含所有东西的环境远比为某个特定行为而优化的用户界面要好。如果有业务案例违背了这一原则，请至少记住这点：在稳定用户界之前，不要去优化它。

典型通用功能

其他的琐事还包括，但并不仅限于以下几个方面：

- 安全：建立服务请求者身份和访问权限。
- 通知：确认某一业务事件应通知哪些人。这包括了维护基于此的事件订阅。
- 输出管理：在线下进行信息通信，而不是作为一种服务响应。典型例子就是当客户请求必须被正式确认时，比如使用电子邮件来确认你刚刚通过浏览器所做的网上采购。对那些主动提供的消息也是需要的，比如每月的账单。输出管理必须决定通过哪条渠道去发送信息，以及使用哪个地址来发送。它应该把消息转换成接收方能够接收的格式，发送消息，并把消息添加到归档文档。输出管理包括维护那些被用来将数据转换成用户可理解消息的模板，以及潜在接收者的地址和渠道偏好。
- 数据转换：把数据从一种格式转换为另一个，把独立的各个服务打包为一个服务——用麦当劳的说法，开心乐园餐——以及分解拆包，将服务请求拆分成适用于不同人群的各个独立请求，汇集各个回应，排队及出列，或协议转换。
- 流程编排：编排某一流程，以确保按适当顺序且仅相关时来执行那些组成流程的服务，确保快要逾期时发送告警信息，以及确保因辅助信息或者逾期打断从而引起的新流程分支被启用。
- 归档管理：维护及访问相关的归档信息。这些可能是虚拟的档案，从某种意义上是展现给用户的信息，当他需要某个档案时可以使用查询来检索。对那些从数据库中抽取的内容，这被认为是正常的，但是没有特殊理由不对文档使用相同的方法。在某些情况下应使用设备来特别增加指派文档至档案中。
- 记录管理：维护那些不允许被更改的信息。

至下而上叙述

这些实现琐事的服务不能形成业务服务层次的部分。不能使用自上而下的方式去设计它们，因为这些服务都没有所谓的“上”。对这样的服务，使用旨在实现最大程度重用的自下而上

的方法会更适合。这使得从这个阶段起就能够实现服务的优胜劣汰。

采用数据库化的方法，很少能够实际把次等通用功能用更好的替换，因为这要求修改所有使用该方法的应用。使用 SOA 则不同，这种替换很简单，前提是已经应用了“不去了解你不需要了解的事情”这条规则，包括其推论：服务请求不应该包含超过指定该请求必要信息之外的其他信息，而且服务本身应该在需要时主动要求更多信息。如授权服务，该服务由某应用调用，旨在决定是否允许某特定用户在某客户数据上执行某项功能——比如说：“我们的雇员 Donald Jones 是否被授权可以访问 Acme Widgets company 公司相关的财务数据？”。服务的简单版本可能具备处理某些特定情况的能力，在此特定情形下可以通过使用雇员功能对应表来回答这些问题。稍微复杂一点的版本可能会识别出 Donald Jones 属于某个或多个组的成员，除了个人权限外还拥有该组的权限。再更近一步，授权服务可能会使用用户证书去区别雇员和客户，并允许客户只能够访问他们自己的数据。一个完善的版本可能会使用标准化的服务去调用业务流程管理系统或者项目管理系统，询问 Donald Jones 是否已经被赋予了任何我们和 Acme Widgets 交易相关的职责。更好的做法是，授权服务会记录请求和应答的日志，而简单的版本不会这样做。组织可以从一个服务的版本切换到另一个而无需对调用服务的应用做任何改变。

也有可能设计出根据操作必需的条件自动配置自己的通用服务。例如，授权服务可以检查是否有服务可以告知它员工对某些特定客户有职责，并在该服务不可用的情况下决定只使用个人还是群组的访问权限。用这种方式，服务可以在很多组织之间复用。

5.不要在测试上自寻烦恼

为什么 **SOA** 更容易测试

对 SOA 缺点的一种看法是测试困难。这种看法完全不恰当，原因有很多。

首先，在 SOA 中使用元数据可以避免错误被植入到系统中。可以在元数据层次就对系统进行验证，例如，保证所有需要处理的数据在使用前就被汇总并校验。在整个业务流程范围内都可以实现这点。当你可以验证设计的时候就不要测试整个系统。

其次，几乎所有测试，包括所有系统集成测试，一旦测试基准被建立后都可以自动完成。但是，需要一些前提条件。表现层和业务执行层必须被严格的区分。好在这是使用 SOA 的一种很自然的方式。对所有输入，都应该存在相应的 XSD。使用该 XSD，可以生成测试记录。同理，也可以生成带有预期输出结果的测试记录。在测试过程中，不能产生可以证明系统运

行正常的任何输出地方，必须在测试脚本中添加专门为此生成的附加输出的查询语句。当测试开始运行时，测试记录被一条条输入系统，然后输出的结果自动和期望的结果进行对比。这会产生一个异常列表，其中每项都应仔细考虑。测试可以按需进行。自然，测试的结果可能取决于存积在数据库中的数据，所以这点需要进行弥补。而且，系统不可表现出时间相关的行为。系统必须有能力响应每隔一段时间（它对自动化测试序列更适合）就产生的事件，而不是花上一周时间去等待某个基于时间的触发器被触发。用户界面的测试应该单独进行，而且永远不在集成测试中使用。

第三，SOA 的设计趋向于产生更加健壮的系统：系统出错的机会更少。SOA 减少了信息系统为了协同工作而需要达成协议的因素数量，这样一来，导致在某关键因素上产生分歧的设计错误的概率也减少了。就算真的出错，也能够在造成损害之前检测到。使用 SOA，消息在被处理前会被验证，这样可以判断消息是否格式正确、是否符合相应的 XSD。

可行性测试

最后，作为数据库时代特有的产物——测试环境和生产环境必须严格区分，从此不再需要了，而且有时候这也是不适合的。这是很有可能的，这是因为我们不再实际 进行系统测试了，而是对测试通路和信息处理的方式进行测试。SOA 提供了三重安全的、有效区分测试消息和生产消息的方法。除了被封装好的消息，其他每个消 息自身和相应的命名空间都包含版本号。而且每个消息都包含一个标签用以指示它是用于测试还是生产。所需的只是一个 SOA 网关，它存在于防火墙内部，对每条 进入消息进行如下处理：

1. 校验消息以确定其是否与一个已知 XSD 的版本相符（被封装好的消息除外）。
2. 使用我们对相应 XSD 的副本对消息进行校验，以确定其是否有效。
3. 如果消息用于生产的，就验证消息版本号是否被允许用于生产。只有这样，消息才能够被传递到生产系统。其他所谓的“生产”消息都会被拒绝。
4. 如果消息用于测试，消息可能会被传递到指定的测试版系统。在特殊情况下，消息如果只是用来做数据检索，那也有可能被传递到生产系统。
5. 只有在消息被完全测试过后，生产版本的注册库和 XSD 才能得以更新。

这样的处理方法不仅仅只是三重安全的，而且使得消息的路由能以一种合乎实际的方式得到测试。这也大大降低了从测试系统切换到生产系统时重新进行配置的需求。因为这种重新配置天生就是不可测试的，常常成为错误的根源。发布经理只能通过在半夜或者周末发布新的

版本软件来弥补这类错误；这样，就算新版本出现了 任何错误，也可以在有人发现错误之前恢复到老版本。但如果这样的变化影响到了其他组织那就没有办法这样操作了。SOA 发布管理要简单得多！

我们对于 SOA 测试的一般认识是时候该改变了。SOA 是能够把测试需求和设置测试的工作减少到最低的一种方案。它能使重要测试更自动化的完成，结果也更好。

完善之

理论

SOA 使得信息系统的开发和部署能够比使用数据库化的方法支持更为丰富的用户体验。这样的系统能够涵盖更多的信息格式、更广的行为集合，其行为上也可以达到更高层次的统一和一致以及更加可靠，不管是从客户还是从组织内关注合规的人员的观点来衡量。然而，要想获得这些好处需要我们跟已有的数据库方法实践说再见。

6.始终信守你的诺言

为什么数据库不能一直信守诺言

接收一个服务请求的动作是通过定义一个承诺，即向请求者承诺服务请求会被执行，来确定的。这种执行定义了一个流程，其至少包含一个步骤，但通常是多个步骤。

数据库化的思考和流程不能融洽相处。从它们各自的本质来看，数据库就像是一个个孤岛。而孤岛会促使偏狭地思考问题：任何孤岛之外的东西都不重要。可以通过数据库中的事务概念来形象地解释这个问题：某个工作单元把数据库从一种一致性状态转移到另一个。在一些特殊的情况下，该概念可能会被扩展到多个数据库，虽然可以通过两阶段提交技术来做，但这也有局限性。逻辑一致性可能需要贯穿整个业务流程得以维护，而不只是恰好在某个时刻；需要在信息改变波及的所有地方去维护，其中不仅包括数据库还有流程管理系统、信息以及发送和接受信息的人工代理，而这一切从数据库世界的观点看来是完全陌生的。

SOA 交易概念

对数据库世界陌生的东西对与 SOA 来说却是再自然不过了。业务交易——实现它的一般是一

个流程——就是服务。为了理解 SOA 何以能很好支持逻辑一致性需求，理解业务交易的需要很重要。业务交易由下面元素组成：

- 厂商给客户提供信息，客户据此可作出采购决定。一般来说，这种信息包括出售商品和服务的属性、得到该商品的条件——包括，当然，价格——以及可用性。从法律的角度，厂商所描述的这一信息是真实的。
- 用户基于厂商描述的信息下采购订单。
- 厂商核实该采购决定依据的信息是否仍然适用，如果是，则确认该订单。如果有变化——可能由于产品或服务已终止，或产品已经涨价，也可能是货物或服务的规格已经变更——那么需要一些处理来决定到底应该如何做：是不管三七二十一继续提交订单，还是通过协商修改订单，或者干脆取消订单。
- 厂商和客户双方履行采购协议中各项条约。

SOA 怎样维持一致性

SOA 通过多种方法维护业务交易的逻辑一致性。第一，所有暗含对前一个状况改变的通信都要使用能够保证消息安全交付的协议来完成。只要厂商或用户做出某个承诺，为实现该承诺所要做的动作就应该包含这样的改变。这样的话，客户和厂商就不可能对当前业务状态持有不同的看法了。

第二，数据库的逻辑一致性和流程管理系统中处理过程的记录可以通过两阶段提交协议来维护。跨多个数据库的逻辑一致性通过一连串这样的两阶段提交维护。首先让数据库 A 和流程管理系统同步，然后流程管理系统再和数据库 B 同步，以此类推。

第三，从厂商给客户展示产品到订单下达期间，厂商面对的任何改变都可以使用乐观锁并发控制机制来处理。这种处理方式对 SOA 来说很自然：判断是否存在相应改动的过程可以完全自动化。因为 SOA 使得数据在被用来作决定的同时也可以进行访问，找不到乐观锁而需要人工介入的情况几乎鲜有发生。

最后，一笔交易的中止——比如说用户撤销了订单，原因可能是用户没有能力支付，或者用户去世了——使用 SOA 可以相对容易地处理。因为在交易上下文中使用或产生的记录可以被清晰地识别，所以可以确定需要哪些补偿信息用以纠正用户和厂商的不同认知。因为哪个数据库更新和该交易有直接关系是很清楚的，在数据库中的哪些变更需要使用补偿交易服务做回滚也很清楚。假设，交易中止的发生相对不那么频繁，使这些活动完全自动化通常是

高投入低产出的，但是你的设计必须要考虑到这些。

请注意业务交易的范围限制在那些为了处理某个服务请求而直接完成的操作。在 SOA 中，编排你自己的流程并提供事件通知给他人，是一条守则。如果服务中止了，那么有必要发通知用户这一个变化。至于他们要怎么处理，则完全是他们自己的事。

也请注意，创建用户服务请求的内容，严格说起来，应该在流程处理之前进行，而不应该作为流程处理的一部分。这真的不是你该做的事：原则上，提供消息的 XSD 和消息验证服务给用户就够了，让他决定是从键盘录入数据还是从他自己的信息系统以某种方式直接生成数据。对消费者来说，这并不是——尚未成为—— 一个可行的方法，但采集数据和处理数据是两件独立的事情，并使用不同工具在各自的环境中执行这些事情，对于这一点仍然是有效的。同样是数据采集，有很多实现方式，这取决于用户的期望以及他们和你沟通的渠道，但不管怎样都应该只有一个服务去处理这事儿。

7.不要将自己禁锢于模型

数据库只是模型，**SOA** 代表更多

领域模型是领域本身的体现，操作领域模型要比直接操作领域更简单。绝大多数的数据库都是模型。它们以这样一种方式代表一些管理的、物理的或者社会的领域： 相关领域的问题都可以通过查询数据库来得到答案，而存在于领域中所需的行为可以从数据库内部得到指示。比如说，通过访问数据库查询用户的地址信息要比跟随 用户到他家然后抄下他所进房子的门牌号要方便的多。而且，在数据库中对记录进行计数也要比清点一个城市有多少人或仓库里有多少产品要容易。原则上，提供对这些功能的支持足以确定数据库的数据结构。尽管这样，因为一般不太可能事先就能明确所有这些功能，所以我们使用数据建模技术对领域中感兴趣的对象进行分析，包括对象之间的关系，以及信息项（它们会告诉我们关于其自身的一些我们可能想知道的东西）。例如，若数据库和用户有关，那么每个用户通常都应该存在对 应的数据库记录，包括一些数据库使用者感兴趣的与用户相关的信息。因为数据模型更多是基于功能上的考虑而不是某些特定数据库的使用，所以将数据结构建立在 这种模型之上会使数据库能够更易适应各种未预见的需求。数据库在语义上被结构化了，换句话来讲，数据库是根据其表达出的意思来构造的。

对没有实现数据模型的数据结构，数据库技术就没那么方便了。它实在不能在描述、文本、图像等方面做很多贡献。相反，SOA 却能够很好地处理文档、记录，就像可以很好处理那些根据数据模型构建的数据一样。SOA 允许单个设计能够处理语义结构化信息和文档。尽管如

此，由于数据库系统支配了我们的思维模式，所以当应用 SOA 时免不了很自然的会坚持其局限性。但是在 SOA 的世界里不存在特殊的理由要去应用这种限制，任何理由都不能够这样做。

超越模型的益处

将语义上结构化的数据和一些像超链接、文本、图片以及音频片段的东西结合起来的一个主要原因是创建更为丰富的用户体验。对消费者来说，这是必须的，而不是可选的。对你雇佣的知识型工作人员而言，这会使他们做事更有效率。只有对从事日常管理的员工来说，它才会成为一种障碍。然而你应用 SOA 越多，你对这些人员的需要就越少。他们从表单键入数据的工作可以外包给任何人做：你需要做的仅仅是扫描每个表单然后把图像发送给代理，代理本质上使用和用户一样的 Web 表单去键入数据。至少真正需要雇员日常处理的工作都可以用自动化完成。

不要让自己禁锢于数据库数据的第二大原因是，把数据库化和基于文本的数据结合起来是目前维持营运合规所需记录信息和审计线索最简单的方法。数据库因其特有的性质并不适合这样的最终目的。因为通常某个数据库是为担当某种管理现状模型而构建的，所以当该管理现状改变的时候，它应该很容易去做相应改变。数据库管理系统就是为了方便这种改变而设计的。当它们被用来维护不允许被更改的记录时——比如，簿记条目——设计者不得不把各种安全防御构建到系统中，从而防止对记录的恶意操作。即便如此，神智清醒的外行人也不会去信任数据库的。为营运合规目的需要的记录信息和审计线索必须被委托给记录管理程序，然后通过 SOA 来与数据库系统进行关联。

使用 SOA，把语义结构化的数据和其他形式的信息表示连接起来很容易。这使得它是内容管理和内容表示的理想之选。数据库可以将超链接保存到其他信息中。比如，我们可以用交易的数据库记录来存储产生该交易的输入文档的超链接。当显示交易记录的同时，链接也跟着展现，而且可以让使用者通过单独的服务去激活它。开发一个使其能够从归档文件去访问文档的服务不会比开发一个从数据库访问数据的服务更困难。当然了，除非你的归档文件不具有 SOA 能力，此时第一件要做的事情就是替换掉它。

但是，反过来也一样。不仅语义格式化的数据通过链接来丰富记录，而且记录也可以通过链接来丰富语义格式化数据。当你将记录存储于归档文件中的时候，与记录相关的语义结构化数据通常会包含所有为其索引而需要的信息。几乎不可能不需要手工索引记录。可喜的是，每当交易在数据库中进行一次，索引就可以自动对其更新从而逐步丰富起来。

集成数据库化和基于文档的数据有另一大优点，就是很容易支持数据的多个版本。在数据库世界这是说不通的，因为数据库作为一种模型，且作为一种有用的模型，它必须在对管理现状模型的任何提问作出至少一个最佳的猜测以应答。一个文档，实际是某个特定组织在特定时间以及特定环境所做的一个声明，而且这也很有可能——经常被用来了解——相冲突声明的存在。

文化冲击

请注意，数据库化的思考者不是唯一在统一文档和数据库世界遭遇困难的人。那些思维方式受当代文档和记录管理系统影响的人会遇到更大的困难。一时之间，归档文件成为业务流程的一部分，而不是流程完成后形成。而索引不是人工完成，也不是一个文档一个索引，而是自动并持续更新的。索引也不再是归档的一部分：归档 仅仅只包含文件，而索引分开维护。因此不需要把自己局限于一个归档，或者只局限于你自己的归档。这些改变是如此深远，很难想象那些适应了旧世界的人也能对 新世界适应。

8.不要使用快照模型

具体而正式的历史信息

我职业生涯中最大的错误就是设计了这样的数据库，其中所有服务请求都被给予一个时间戳，以便数据添加进去后，服务可以根据指定时刻数据库存放的数据内容来响应，而不是根据当前内容。这个数据库极其昂贵，把执行速度拖累得就像龟行，而且没能起到很大作用。当用户想看看过去某个特定的结果是怎么被计算出来的时候，他们不免只能够查询归档文件，因为这些归档文件总是会包含之前与用户沟通的结果。而这也正是他们想要知道的东西。

请注意数据库的用途不是去记录过去那些成为问题的情况。数据库是现实世界某些部分的模型，以及在过去某一段时间里现实世界那部分的状态——它的具体历史 ——可能和模型的目的非常相关。但是，如果它想要再现自己在过去某个点的状态，换句话说去记录它的正式历史信息，数据库就太把自己当回事儿了。

SOA 如何支持正式历史记录

SOA 方法提供了一种再现正式历史信息的高效方案，不只针对单个数据库，而是针对组织在其管理流程中使用或者产生的所有信息。SOA 对组织信息更新所需的数据收集和实际执行更

新的分离让数据可以在记录管理程序中得以归档。一旦应用了 SOA，这就可以自动完成，而且合并新的数据集几乎不消做多余的努力。这样一来，你可以获得由自己雇员所做的所有输入消息以及更新。

对即将离开的线下信息情况来说，可能更简单。SOA 厌恶将琐事与对客户的附加值混合起来，这种混合导致客户很自然地会去使用输出管理服务以确保离线信息通过适当的渠道发送至目标接收者，发送时要确保地址是接收者会使用的，并且格式是接收者确实能收到的。输出管理服务能够自动记录输出信息。

尽管如此，还是有个问题。组织使用 SOA 传播信息的难易程度有可能导致对外界的服务响应数量大大增加。通常这些服务响应都隐含了组织对他们声明内容的承诺。如果它们不正确的话，组织可能会陷入法律难题。SOA 处理这种问题的标准方法包括以下指导方针：

- 将放弃承诺包含在对全部信息进行检索的服务的服务水平协议中。
- 在需要使用时才访问数据。这样可以使数据尽可能的最新。
- 不要撒谎。如果一直讲真话，你将很少遇到关于信息不正确的投诉。
- 如果使用放弃承诺还使得你不得不对声称信息是不正确的声明作出反应，那对每个服务请求和回应进行日志，采用标准的组织范围的服务。

9.不要相信系统

SOA 安全

如果你的身体使用与大多数 Web 网站相同的防御规则来抵制外物，只消一天你就会死亡。我们生活在一个怀有敌意的世界，这里唯一能够避免信息系统遭受攻击的办法就是让它们不可访问。在这样的世界，SOA 是一种绝妙的安全概念。原则上说，唯一必须让其通过防火墙的消息是牢牢封装好的 XML 消息。它们可以以这样的方式通过防火墙：在进一步处理之前它们会接受格式良好、XSD 一致性以及授权有效性方面的检查。这样，造成危害的可能性非常小。缓冲区溢出只能够在安全防御区内爆发，而 SQL 注入不过是我们对其充耳不闻的一种语言上的祸害。像这样畸形的消息在其作出任何伤害之前就能被识别出来。

认识到这点很重要：即这种方法的优势是基于消息的认证和校验而非系统的。这是必要的，因为封装好的 XML 常常通过那些对其内容不负任何责任的系统传递，以至于检查消息是否来自认证过的系统并不能保证什么。它更健壮，因为某信息系统或电脑一次成功侵入并不会

为其他侵入打开大门。

SOA 的安全和数据库时代基于系统的安全形成了鲜明对比。有了基于系统的安全保障，对每个消息类型和数据项来说，每个系统的任务就是避免缓冲区溢出。有上千个点可能成为出错点。因此，就少不了薄弱点存在的可能性。而另一方面，某个 SOA 关守（gatekeeper），可以在单一的点上完成所有这些。基于系统的访问安全倾向于把授权控制构建到系统中，这不仅使得跟踪记录变得几乎不可能，而且引入结构性改变（比如把外部组织的客户添加进来）也是很昂贵的花销。

使用 SOA 关守的另一大优势在于可以保证所有输出消息被适当加密和签名，这样就没人能够窃取消息或在传输中篡改消息。原则上，大部分 WS-*标准都可以完全由 SOA 关守处理。

超越基于系统的授权

基于系统的安全也会产生了一些心态，这些心态更多的是由基于系统的思维决定的，而不是如我们所愿意承认的那样。因为系统是围绕着功能点和数据结构构建的，授权就是这样表达的。例如，雇员要么有权限改变产品价格，要么没有，这是基于他在组织中的角色而定。对后端雇员不会遇到这种问题，但对管理层雇员以及用户而言，这就不适合了。对他们来说，允许他们查看和完成事情的决定因素是横跨所有系统和数据库的手头客户的信息切片。一个简单的信息安全文献扫描就足以构建 那些信息安全咨询公司尚未企及的一点：对每一个基于切片的访问安全引用有数以千记的基于角色的访问安全。从积极的角度看，涌现出大量的概念，它们对充分发挥 SOA 提供的各项技术（基于声明的访问控制、联邦身份及网络边界去除技术）的优点来实现访问安全有长期正面的影响。

可管理之

理论

想要取得 SOA 最大化的利益，仅仅采用新的行为准则是不够的。组织机构考虑自身的方式及 IT 方式也需要改变。否则当前的组织将还是退回到老样子，并不是因为它想要这样，而是因为它把这看作是控制自身日常行为的唯一方法。为了克服这点，你必须从项目的组织去着手。

10.不到最后不要扔掉基础设施

为什么要担心基础设施？

SOA 就是要使基础设施从业务中分离开来，不只是在功能上，而且还在项目管理上：项目经理应该能够把思想集中在他负责开发的服务的商业附加值上，而不是担心基础设施。业务项目经理应该去组织开发一个作为项目一部分的新授权服务，而不是去构建一个新电话局，这不会荒谬到哪里去。让项目经理承担类似这些事情会导致他要协调的事物数目增加，从而进一步导致其工作复杂性不成比例的增加。就算让他等待直到其他项目经理把基础设施交付了，这也是应该尽可能去避免的。

上述评论适用于硬件基础设施也同样适用于基础设施服务。计算能力和网络容量应该总要超过需求，这样的话实施新服务的项目经理就不用担心基础设施是否能够为比原先预想更多的每天上千个服务请求提供支持。鉴于软件开发一般都比 IT 基础设施要贵出一个数量级，我们最不想要的就是：系统开发项目由于那些需要确保 IT 基础设施可用性的官僚性流程而被纠缠停滞不前。

基础设施：**SOA** 有何不同

好消息是用 SOA 的方式去提出你的要求然后静观其变会得到简单、稳定的基础设施服务接口。这也就是为什么在你没有启动任何业务项目前，没有必要去实施世界上最先进的基础设施服务。这些服务的复杂性不在于用来调用它们的接口，而在于它们从其他来源——或者从自我维护中——要求的信息，以便生产一个优化的结果。因此，完全可行的办法是有了初步的基础设施服务就启动项目，然后在业务项目中花足够多的时间去使用这些服务来做测试和生产。

坏消息是 SOA 会导致更多的功能被分类为基础设施。尤其是，每个琐事引起一个通用的服务，而这一服务必须作为基础设施的一部分而不是成为需要它的每个业务功能的一部分来实现。虽然就本身来说不算很坏，但这的确导致了一个新问题：对于基础设施的每个部分来说，如何去组织安排各项事宜才能使既拥有权限又具有资源的人去确保它能按时可供使用而且具备了适当的功能和容量。组织若是不能妥善处理这一问题的话，就不能很好地实施 SOA——事情就这么简单。

敢问路在何方？

虽然组成 SOA 的要素已存在多时，但 SOA 本身是全新的。就像集装箱一样，它不只是一种全新的处理我们一直以来所做事情的方法，因为它使得我们能以新的规模合作。而且 SOA 根本不是一项新技术，而是一种新的思维方式。SOA 与我们现在考虑问题的方式是如此不同，所以要是你通读本文几遍才开始理解其中的意思，你并不用感到难为情。你对此的自然倾向是，你会觉得这些建议不切实际或没有必要或者两者皆是，而使你根本无法对它作出评估，在这之前你还需要适应一段时间。

公正地说，也会出现 SOA 方案对你不直接起作用的情况。比如，你可能会从一些厂商那边以打包的形式获得你所有的系统，而那些厂商对采用本文中提到的方式来应用 SOA 一点特别的兴趣都没有。那样的话，把本文中所给的建议作为指南：使用它们来判定是否你做的选择会把你带向正确的方向。

SOA 是如此之新，所以并不是所有为了充分利用 SOA 而被我们需要的概念、工具及标准都已经可用了。我们不能期待现行的关守（gatekeeper）像一个 SOA 关守（SOA gatekeeper）一样为我们代劳一切，现行的用户接口设备对于封装的 XML 有困难。更糟的是，正如本文表明的那样，当前 SOA 世界里存在很多扑朔迷离、多余或者错误的东西。目前，SOA 概念和技术的应用无论在哪里都没有达到 SOA 互操作性能力所能够达到的层次。我们遇到的互操作性问题往往都是由于其他一些人选择了实现稍有不同的众多 WS-*标准中的子集。那些数以百万计的多方通信的可能方案已经减少到很少的一把了，但依然还是实在太多了。

不管怎样，我们开了个好头。接下来我们可以沿着这条路继续走。SOA 也同样如此。我们不需要为了进步而要去了解我们还不能了解的东西。向前看，我们有极大的潜力去使得事情更简单、更可靠，更具可预测性以及功能更丰富。是时候出发了。

原文链接：http://www.infoq.com/cn/articles/10-soa-commandments

相关内容：

- 创建RESTful服务，有GET和POST足矣？
- 有必要定义SOA的非原则吗？
- SOA语法——服务(Services) 是动词还是名词？
- 拉近SOA和BPM的距离
- “我能以后再调用你吗？”使用SCA开发异步服务

Spring MVC 与 JAX-RS 比较与分析

作者 Rossen Stoyanchev 译者 张龙

过去几年，REST 逐渐成为影响 Web 框架、Web 协议与 Web 应用设计的重要概念。如果你还不了解 REST，那这个简短的介绍将有助你快速掌握 REST，此外还可以点击这里了解关于 REST 的更多信息。

现在有越来越多的公司希望能以简单而又贴合 Web 架构本身的方式公开 Web API ,因此 REST 变得越来越重要也就不足为奇了。使用 Ajax 进行通信的富浏览器端也在朝这个目标不断迈进。这个架构原则提升了万维网的可伸缩性，无论何种应用都能从该原则中受益无穷。

JAX-RS（JSR 311）指的是 Java API for RESTful Web Services，Roy Fielding 也参与了 JAX-RS 的制订，他在自己的博士论文中定义了 REST。对于那些想要构建 RESTful Web Services 的开发者来说，JAX-RS 给出了不同于 JAX-WS（JSR-224）的另一种解决方案。目前共有 4 种 JAX-RS 实现，所有这些实现都支持 Spring，Jersey 则是 JAX-RS 的参考实现，也是本文所用的实现。

如果你使用 Spring 进行开发，那可能想知道（或者有人曾问过你）Spring MVC 与 JAX-RS 有何异同点？更进一步，如果你手头有一个 Spring MVC 应用，使用了控制类继承（SimpleFormController 等），你可能还意识不到现在的 Spring MVC 对 REST 广泛的支持。

本文将介绍 Spring 3 中的 REST 特性并与 JAX-RS 进行对比，希望能帮助你理顺这两种编程模型之间的异同点。

开始前 ,有必要指出 JAX-RS 的目标是 Web Services 开发(这与 HTML Web 应用不同)而 Spring MVC 的目标则是 Web 应用开发。Spring 3 为 Web 应用与 Web Services 增加了广泛的 REST 支持，但本文则关注于与 Web Services 开发相关的特性。我觉得这种方式更有助于在 JAX-RS 的上下文中讨论 Spring MVC。

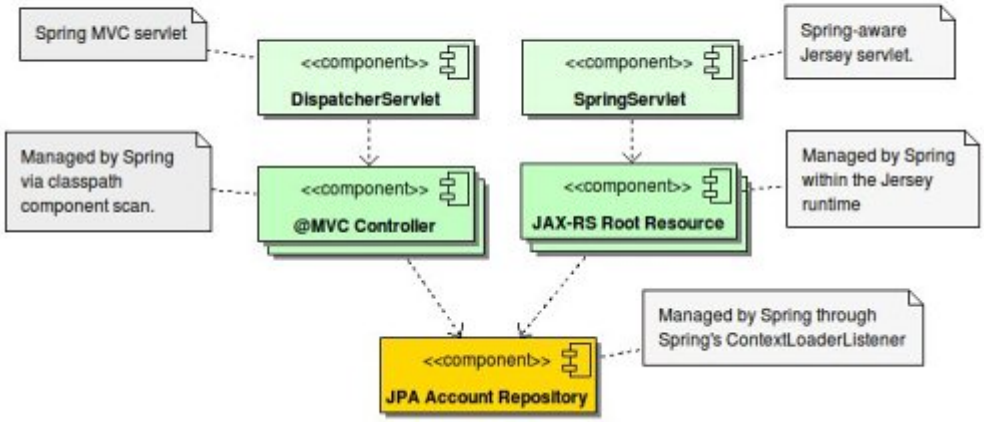
要说明的第二点是我们将要讨论的 REST 特性是 Spring Framework 的一部分，也是现有的 Spring MVC 编程模型的延续，因此，并没有所谓的“Spring REST framework”这种概念，有的只是 Spring 和 Spring MVC。这意味着如果你有一个 Spring 应用的话，你既可以使用 Spring MVC 创建 HTML Web 层，也可以创建 RESTful Web Services 层。

关于文中的代码片段

文中的代码片段假想了一个简单的领域模型：两个 JPA 注解实体，分别是 Account 和 Portfolio，其中一个 Account 对应多个 Portfolio。持久层使用 Spring 配置，包含了一个 JPA 仓储实现，用于获取和持久化实体实例。Jersey 和 Spring MVC 用于构建 Web Services 层，通过调用底层的 Spring 托管应用来服务客户端请求。

引导程序与 Web 层包装

我们会在 Spring MVC 和 JAX-RS 中都使用 Spring 实现依赖注入。Spring MVC DispatcherServlet 和 Jersey SpringServlet 会把请求代理给 Spring 管理的 REST 层组件（控制器或资源），后者会由业务或持久层组件包装起来，如下图所示：



Jersey 和 Spring MVC 都使用 Spring 的 ContextLoaderListener 加载业务与持久层组件，比如 JpaAccountRepository：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:META-INF/spring/module-config.xml
    </param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener 可用于任何 Web 或 REST 框架环境中。

在 Jersey 中创建 Spring 管理的 JAX-RS 资源

Jersey 支持在 REST 层中使用 Spring，两个简单的步骤就能搞定（事实上有 3 步，还需要将构建依赖加到 maven artifact com.sun.jersey.contribs:jersey-spring 中）。

步骤一：将如下配置片段加到 web.xml 中以保证 Spring 能够创建 JAX-RS 根资源：

```
<servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>
        com.sun.jersey.spi.spring.container.servlet.SpringServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

步骤二：使用 Spring 和 JAX-RS 注解声明根 JAX-RS 资源类：

```
@Path("/accounts/")
@Component
@Scope("prototype")
public class AccountResource {

    @Context
    UriInfo uriInfo;

    @Autowired
    private AccountRepository accountRepository;

}
```

如下是对这些注解的说明：

@Component 将 AccountResource 声明为 Spring bean。

@Scope 声明了一个 prototype Spring bean，这样每次使用时都会实例化（比如每次请求时）。

@Autowired 指定了一个 AccountRepository 引用，Spring 会提供该引用。

@Path 是个 JAX-RS 注解，它将 AccountResource 声明为“根”JAX-RS 资源。

@Context 也是一个 JAX-RS 注解，要求注入特定于请求的 UriInfo 对象。

JAX-RS 有“根”资源（标记为@Path）和子资源的概念。在上面的示例中，AccountResource 就是个根资源，它会处理以 “/accounts/”开头的路径。AccountResource 中的方法如 getAccount()只需声明针对类型级别的相对路径即可。

```
@Path("/accounts/")
```

```
@Component
@Scope("prototype")
public class AccountResource {

    @GET
    @Path("{username}")
    public Account getAccount(@PathParam("username") String username) {

    }

}
```

访问路径“/accounts/{username}”（其中的 username 是路径参数，可以是某个账户的用户名）的请求将由 getAccount()方法处理。

根资源由 JAX-RS 运行时（在本示例中是 Spring）实例化，子资源则由应用本身实例化。比如说，对于“/accounts/{username}/portfolios/{portfolioName}”这样的请求，AccountResource（由路径的第一部分“/accounts”标识） 会创建一个子资源实例，请求会被代理给该实例：

```
@Path("/accounts/")
@Component
@Scope("prototype")
public class AccountResource {

    @Path("{username}/portfolios/")
    public   PortfolioResource   getPortfolioResource(@PathParam("username")
String username) {
        return new PortfolioResource(accountRepository, username, uriInfo);
    }

}
```

PortfolioResource 本身的声明并没有使用注解，因此其所有的依赖都是由父资源传递过来的：

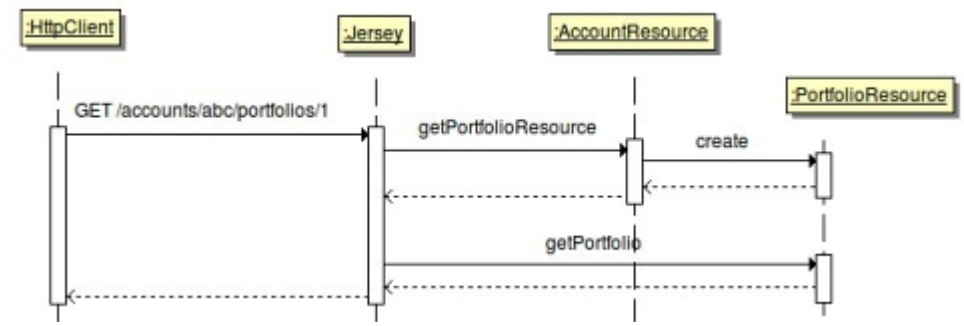
```
public class PortfolioResource {

    private AccountRepository accountRepository;
    private String username;
    private UriInfo uriInfo;

    public   PortfolioResource(AccountRepository   accountRepository,   String
username, UriInfo uriInfo) {
        this.accountRepository = accountRepository;
        this.username = username;
        this.uriInfo = uriInfo;
    }

}
```

JAX-RS 中的根与子资源创建了一个处理链，它会调用多个资源：



请记住，资源类是 Web Services 层组件，应当关注于 Web Services 相关的处理，比如输入转换、准备响应、设定响应代码等等。此外，将 Web Services 逻辑与业务逻辑分隔开来的实践需要将业务逻辑包装到单独的方法中以作为事务边界。

创建 **Spring MVC @Controller** 类

对于 Spring MVC 来说，我们需要创建 DispatcherServlet，同时将 contextConfigLocation 参数指定为 Spring MVC 配置：

```
<servlet>
   <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-c
lass>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/*.xml
      </param-value>
   </init-param>
</servlet>
```

要想在 Spring MVC（@MVC）中使用基于注解的编程模型还需要少量的配置。下面的 component-scan 元素会告诉 Spring 去哪里寻找@Controller 注解类。

```
<context:component-scan base-package="org.springframework.samples.stocks" />
```

接下来，我们声明了 AccountController，如下代码所示：

```
@Controller
@RequestMapping("/accounts")
public class AccountController {

   @Autowired
   private AccountRepository accountRepository;

}
```

@RequestMapping 注解会将该控制器映射到所有以“/accounts”开头的请求上。AccountController 中的方法如 getAccount()只需声明针对“/accounts”的相对地址即可。

```
@RequestMapping(value = "/{username}", method = GET)
public Account getAccount(@PathVariable String username) {

}
```

Spring MVC 则没有根资源与子资源的概念，这样每个控制器都是由 Spring 而非应用来管理的：

```
@Controller
@RequestMapping("/accounts/{username}/portfolios")
public class PortfolioController {

    @Autowired
    private AccountRepository accountRepository;

}
```

对“/accounts/{username}/portfolios”的请求会被直接代理给 PortfolioController，AccountController 则完全不会参与其中。需要注意的是，该请求也可以直接由 AccountController 处理，这样就不需要 PortfolioController 了。

Web 层组件范围

在 JAX-RS 中，AccountResource 是通过前请求（per-request）语义声明的，这也是 JAX-RS 默认的推荐设置。这么做可以将特定于请求的数据注入并存储到资源类本身当中，这适用于由 JAX-RS 所管理的根级别资源。子资源由应用实例化，并不会直接从这种方法中获益。

在 Spring MVC 中，控制器永远都是单例的，他们将特定于请求的数据作为方法参数。JAX-RS 也可以这么做，以单例的方式创建资源。

将请求映射到方法上

接下来，我们看看 Spring MVC 和 JAX-RS 如何将请求映射到方法上。@Path 和@RequestMapping 都可以从 URL 中抽取出路径变量：

```
@Path("/accounts/{username}")
@RequestMapping("/accounts/{username}")
```

这两个框架也都可以使用正则表达式抽取路径变量：

```
@Path("/accounts/{username:.*}")
@RequestMapping("/accounts/{username:.*}"
```

Spring MVC 的@RequestMapping 可以根据查询参数的有无来匹配请求：

```
@RequestMapping(parameters="foo")
@RequestMapping(parameters="!foo")
```

或是根据查询参数值进行匹配：

```
@RequestMapping(parameters="foo=123")
```

@RequestMapping 还可以根据头信息的有无来匹配请求：

```
@RequestMapping(headers="Foo-Header")
@RequestMapping(headers="!Foo-Header")
```

或是根据头信息的值进行匹配：

```
@RequestMapping(headers="content-type=text/*")
```

处理请求数据

HTTP 请求中包含着应用需要提取和处理的数据，如 HTTP 头、cookie、查询字符串参数、表单参数以及请求体（XML、JSON 等）中所包含的大量数据。在 RESTful 应用中，URL 本身也可以带有重要的信息，如通过路径参数指定需要访问哪个资源、通过文件扩展名（.html, .pdf）指定需要何种内容类型等。HttpServletRequest 提供了处理这一切的所有底层访问机制，但直接使用 HttpServletRequest 实在是太乏味了。

请求参数、**Cookies** 和 **HTTP** 头

Spring MVC 和 JAX-RS 拥有能够抽取这种 HTTP 请求值的注解：

```
@GET @Path
public void foo(@QueryParam("q") String q, @FormParam("f") String f,
@CookieParam("c") String c,
    @HeaderParam("h") String h, @MatrixParam("m") m) {
    // JAX-RS
}

@RequestMapping(method=GET)
public void foo(@RequestParam("q") String q, @CookieValue("c") String c,
@RequestHeader("h") String h) {
    // Spring MVC
}
```

上面的注解非常像，区别在于 JAX-RS 支持矩阵参数（matrix parameters）的抽取，拥有单独的注解来处理查询字符串和表单参数。矩阵参数并不常见，他们类似于查询字符串参数，但却使用了特殊的路径片段（比如 GET /images;name=foo;type=gif）。稍后将介绍表单参数。

假如使用了前请求范围声明资源，那么 JAX-RS 可以在属性和 setters 方法上使用上述注解。

Spring MVC 有个特性能让我们少敲几个字符，如果注解名与 Java 参数名相同，那么就可以省略掉上面的注解名了。比如说，名为“q”的请求参数要求方法参数也得为“q”：

```
public void foo(@RequestParam String q, @CookieValue c, @RequestHeader h) {

}
```

这对于那些在参数中使用了注解而导致方法签名变长的情况来说实在是太方便了。请记住，这个特性要求代码使用调试符号进行编译。

类型转换与 **HTTP** 请求值的格式化

HTTP 请求值（头、cookies 和参数）是不变的字符串并且需要解析。

JAX-RS 通过寻找 valueOf()方法或是在客户化的目标类型中接收字符串的构造方法来解析请求数据。JAX-RS 支持如下类型的注解方法参数，包括路径变量、请求参数、HTTP 头值和 cookies：

- 原生类型。
- 拥有接收单个字符串参数的构造方法的类型。
- 拥有一个接收单个字符串参数的名为 valueOf 的静态方法的类型。
- List<T>、Set<T>或是 SortedSet<T>，其中的 T 满足上面 2 个或 3 个要求。

Spring 3 支持上面所有要求。除此之外，Spring 3 提供了一种全新的类型转换与格式化机制，并且可以使用注解实现。

表单数据

如前所述，JAX-RS 处理查询字符串参数和表单参数的方式是不同的。虽然 Spring MVC 只有一个@RequestParam ,但它还提供了一种 Spring MVC 用户很熟悉的数据绑定机制来处理表单输入。

比如说，如果一个表单提交了 3 个数据，那么一种可能的处理方式就是声明一个带有 3 个参数的方法：

```
@RequestMapping(method=POST)
public void foo(@RequestParam String name, @RequestParam creditCardNumber,
@RequestParam expirationDate) {
    Credit card = new CreditCard();
    card.setName(name);
    card.setCreditCardNumber(creditCardNumber);
    card.setExpirationDate(expirationDate);

}
```

然而，随着表单数据量的增加，这种处理方式就会变得不切实际。借助于数据绑定，Spring

MVC 可以创建、组装并传递包含有嵌套数据（账单地址、邮件地址等）、任意结构的表单对象。

```
@RequestMapping(method=POST)
public void foo(CreditCard creditCard) {
    // POST /creditcard/1
    //      name=Bond
    //      creditCardNumber=1234123412341234
    //      expiration=12-12-2012
}
```

要想与 Web 浏览器协同工作，表单处理是个重要环节。另一方面，Web Services 客户端一般会在请求体中提交 XML 或 JSON 格式的数据。

处理请求体中的数据

无论是 Spring MVC 还是 JAX-RS 都能够自动处理请求体中的数据：

```
@POST
public Response createAccount(Account account) {
    // JAX_RS
}

@RequestMapping(method=POST)
public void createAccount(@RequestBody Account account) {
    // Spring MVC
}
```

JAX-RS 中的请求体数据

在 JAX-RS 中，类型 MessageBodyReader 的实体供应者负责转换请求体数据。JAX-RS 的实现需要拥有一个 JAXB MessageBodyReader，这可以使用具有注解@Provider 的客户化 MessageBodyReader 实现。

Spring MVC 中的请求体数据

在 Spring MVC 中，如果想通过请求体数据初始化方法参数，那可以将@RequestBody 注解加到该方法参数前，这与之前介绍的表单参数初始化正好相反。

在 Spring MVC 中，HttpMessageConverter 类负责转换请求体数据，Spring MVC 提供了一个开箱即用的 Spring OXM HttpMessageConverter。它支持 JAXB、Castor、JiBX、XMLBeans 和 XStream，此外还有一个用于处理 JSON 的 Jackson HttpMessageConverter。

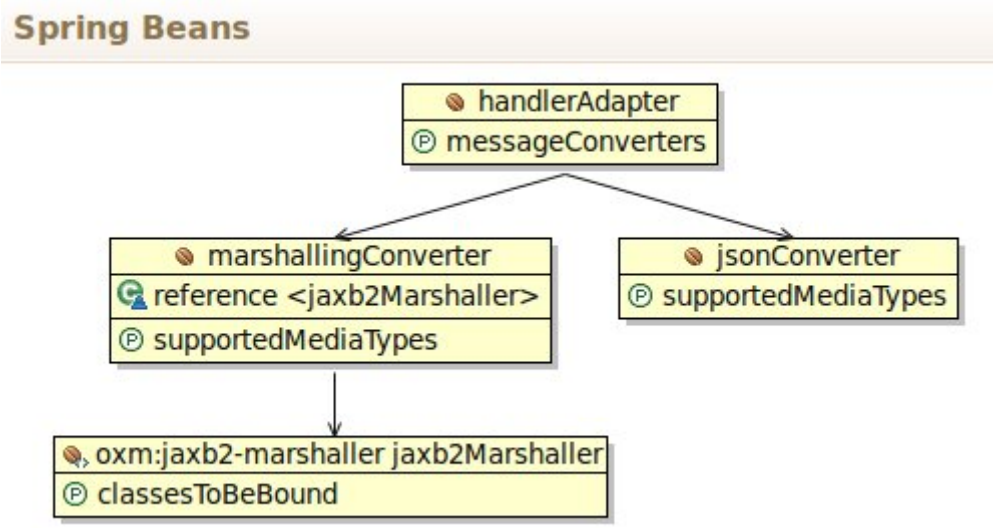
HttpMessageConverter 会注册到 AnnotationMethodHandlerAdapter 上，后者会将到来的请求映射到 Spring MVC @Controllers 上。下面是其配置：

```
<bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandl
erAdapter" >
    <property name="messageConverters" ref="marshallingConverter"/>
</bean>

<bean                                                id="marshallingConverter"
class="org.springframework.http.converter.xml.MarshallingHttpMessageConvert
er">
    <constructor-arg ref="jaxb2Marshaller"/>
    <property name="supportedMediaTypes" value="application/vnd.stocks+xml"/>
</bean>

<oxm:jaxb2-marshaller id="jaxb2Marshaller"/>
```

下图阐述了该配置：



Spring 3 新增的 mvc 客户化命名空间将上述配置自动化了，只需增加如下配置片段即可：

```
<mvc:annotation-driven />
```

如果 JAXB 位于类路径上，它会注册一个用于读写 XML 的转换器；如果 Jackson 位于类路径上，它会注册一个用于读写 JSON 的转换器。

准备响应

典型的响应需要准备响应代码、设定 HTTP 响应头、将数据放到响应体当中，还需要处理异常。

使用 **JAX-RS** 设定响应体数据

在 JAX-RS 中，要想将数据加到响应体中，只需要从资源方法中返回对象即可：

```
@GET
@Path("{username}")
public Account getAccount(@PathParam("username") String username) {
    return accountRepository.findAccountByUsername(username);
}
```

JAX-RS 会寻找类型 MessageBodyWriter 的实体供应者，它能将对象转换为所需的内容类型。JAX-RS 实现需要具备一个 JAXB MessageBodyWriter，这可以使用具有注解@Provider 的客户化 MessageBodyWriter 实现。

使用 **Spring MVC** 设定响应体数据

在 Spring MVC 中，响应是通过一个视图解析过程来实现的，这样就可以从一系列视图技术中选择了。但在与 Web Services 客户端交互时，更加合理的方式则是舍弃视图解析过程，转而使用方法所返回的对象：

```
@RequestMapping(value="/{username}", method=GET)
public @ResponseBody Account getAccount(@PathVariable String username) {
    return accountRepository.findAccountByUsername(username);
}
```

如果对控制器方法或其返回类型应用注解@ResponseBody，那么就会使用 HttpMessageConverter 处理返回值，然后用该返回值设定响应体。用于请求体参数的 HttpMessageConverter 集合也用于响应体，因此无需再做任何配置。

状态代码与响应头

JAX-RS 使用一个链式 API 来构建响应：

```
@PUT @Path("{username}")
public Response updateAccount(Account account) {
    // ...
    return Response.noContent().build(); // 204 (No Content)
}
```

这可以与 UriBuilder 联合使用来为 Location 响应头创建实体链接：

```
@POST
public Response createAccount(Account account) {
    // ...
    URI                          accountLocation                          =
uriInfo.getAbsolutePathBuilder().path(account.getUsername()).build();
    return Response.created(accountLocation).build();
}
```

上面代码中所用的 uriInfo 要么被注入到根资源（使用了@Context）中，要么是从父资源传递给子资源。它可以附加到当前请求的路径之后。

Spring MVC 提供了一个注解来设定响应代码：

```
@RequestMapping(method=PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void updateAccount(@RequestBody Account account) {
    // ...
}
```

可以直接使用 HttpServletResponse 对象设定 Location 头：

```
@RequestMapping(method=POST)
@ResponseStatus(CREATED)
public void createAccount(@RequestBody Account account, HttpServletRequest
request,
              HttpServletResponse response) {
    // ...
    String requestUrl = request.getRequestURL().toString();
    URI uri = new UriTemplate("{requestUrl}/{username}").expand(requestUrl,
account.getUsername());
    response.setHeader("Location", uri.toASCIIString());
}
```

异常处理

JAX-RS 允许资源方法抛出 WebApplicationException 类型的异常，该异常会包含一个响应。下面的示例代码将一个 JPA NoResultException 转换为特定于 Jersey 的 NotFoundException ,这会导致一个 404 的错误：

```
@GET
@Path("{username}")
public Account getAccount(@PathParam("username") String username) {
    try {
        return accountRepository.findAccountByUsername(username);
    } catch (NoResultException e) {
        throw new NotFoundException();
    }
}
```

WebApplicationException 实例会封装必要的逻辑来生成特定的响应，但每个独立的资源类方法中都需要捕获异常。

Spring MVC 支持定义控制器级别的方法来处理异常：

```
@Controller
@RequestMapping("/accounts")
public class AccountController {

    @ResponseStatus(NOT_FOUND)
    @ExceptionHandler({NoResultException.class})
    public void handle() {
        // ...
    }
```

```
}
```

如果任何控制器方法抛出了 JPA 的 NoResultException 异常，上面的处理器方法就会得到调用并处理该异常，然后返回一个 404 错误。这样，每个控制器就都能处理异常了，好象来自同一个地方一样。

总结

希望本文有助于你理解 Spring MVC 构建 RESTful Web Services 的方式及其与 JAX-RS 编程模型之间的异同点。

如果你是个 Spring MVC 用户，那么你可能用它开发过 HTML Web 应用了。REST 概念适用于 Web Services 和 Web 应用，尤其是富客户端交互上更是如此。除了本文介绍的特性之外，Spring 3 还增加了对 RESTful Web 应用的支持。这是部分新特性的列表：用于从 URL 模板构建 URL 的新的 JSP 客户化标签、基于 HTTP PUT 和 DELETE 模拟表单提交的 Servlet 过滤器、根据内容类型自动选择视图的 ContentTypeNegotiatingViewResolver、新的视图实现等等。此外，Spring 文档也改进颇多。

关于作者

Rossen Stoyanchev 是 SpringSource 的高级咨询师。在其职业生涯中，他做过贸易应用、记账系统和电子商务等 Web 应用。在 SpringSource 中，Rossen 专注于 Web 技术，包括咨询、培训和“Rich-Web Development With Spring”课程的内容开发，该课程旨在帮助受训者成为认证的 Spring Web 应用开发者。

原文链接：http://www.infoq.com/cn/articles/springmvc_jsx-rs

相关内容：

- 用Restlet创建面向资源的服务
- 专访Restlet框架首席开发者Jérome Louvel
- Java EE 6 依赖注入提供了统一的EJB与JSF编程模型
- 解耦应用与依赖注入框架
- 云需要服务器虚拟化吗？

时刻关注企业软件开发领域的变化与创新

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

新品推荐 | New Products

NetBeans 6.9 发布，支持 JavaFX、Java EE 6 及 OSGi

作者 Josh Long 译者 张龙

近日，Oracle 发布了流行的开源 Java IDE NetBeans 6.9，这是自收购 Sun Microsystems 以来 Oracle 发布的首个 NetBeans 版本。

原文链接：http://www.infoq.com/cn/news/2010/07/netbeans_69_released

Entity Framework Feature CTP 4 更新了 Code First 编程模型并提升了生产率

作者 Abel Avram 译者 张龙

近日，Entity Framework Feature CTP 4 发布了，更新了 Code First 编程模型，增加了新的编码约定与数据注解，同时还新增了两个类型：DbContext 与 DbSet，可以使用 Code First 简化实体模型的创建。

原文链接：http://www.infoq.com/cn/news/2010/07/EF-CTP-4

Grails 1.3.3 发布

作者 张龙

昨日，SpringSource 发布了 Grails 1.3.3，该版本包含了多处 bug 修复以及少量的改进。

原文链接：http://www.infoq.com/cn/news/2010/07/grails1.3.3-release

Apache 软件基金会发布 Tomcat 7.0，支持 Servlet 3.0、JSP 2.2 及 EL 2.2

作者 张龙

上月 29 日，Apache 软件基金会（ASF）发布了屡获大奖的开源 Java Web 服务器 Tomcat 7.0。

原文链接：http://www.infoq.com/cn/news/2010/07/tomcat7-release

MemBase，NoSQL 家族的一个新的重量级的成员

作者 Michael Hunger 译者 池建强

North Scale 宣布了开源 NoSQL 数据库解决方案 Membase 已经可用了，Membase 是 North Scale 的 memcached 核心团队成员开发完成，其中还包括来自 Zynga 和 NHN 的贡献者。

原文链接：http://www.infoq.com/cn/news/2010/06/membase

Apache CouchDB 1.0 破茧而出

作者 张龙

North Scale 宣布了开源 NoSQL 数据库解决方案 Membase 已经可用了，Membase 是 North Scale 的 memcached 核心团队成员开发完成，其中还包括来自 Zynga 和 NHN 的贡献者。

原文链接：http://www.infoq.com/cn/news/2010/06/membase

推荐编辑 | Agile 社区编辑 李剑

同学们好，我叫李剑，ThoughtWorks 咨询师，擅长面向对象设计、重构及 Java 图形界面开发；致力于推动敏捷思想在国内技术社区的传播和应用。我 的译作包括：《硝烟中的 Scrum 和 XP》、《实现模式》、《ThoughtWorks 文集》、《精益和敏捷开发大 型应用指南》、《Scala 程序设计》。怎么样，作品不少吧？其实这些作品，大部分是在我加入 InfoQ 之后完成的。InfoQ 像一个大家庭，其中有很多志同道合的朋友。我在翻译的过程中，在和大家交流的过程中，成长了很多。

在推广敏捷思想的过程中，经常听到有人说这样的话，敏捷虽然好，但是不一定适合我们的情况，还是要具体问题具体分析。

“具体问题具体分析”几乎已经成了万能的防具，无论在什么场合都可以不假思索的说出这句话来搪塞过去。

我们说，要做到迭代式交付，就必须要有测试自动化。就有人出来说了，写自动化测试的成本很高啊，不是什么时候都能做到测试自动化的，这个要具体问题具体分析。

我们说，“把密切合作的一群人组成一个部门”优于“把相似技能的人组成一个部门”，有人就说公司规模大了就不适用，这个地方还要具体问题具体分析。

熊常说，你能具体问题抽象分析一下么？

所谓的“具体问题具体分析”，实际上就等于没有分析。我几乎从来没有听到过任何一个说这句话的人，能够举出几个具体的场景来具体分析一下。这句话所代表的意义就是，你彻底放弃了思考的权利。

封面植物

黄山木兰

黄山木兰，渐危种。落叶乔木，高 8 至 10 米，胸径达 30 厘米。星散分布于安徽、浙江、江西及福建部分地区海拔 600 至 1700 米山地疏林中。适生于雨量充沛、温凉、多雾气候，幼树稍耐阴，根系发达，萌蘖性强。天然更新困难。3 月下旬开花，9 月果熟。

生态习性：黄山木兰，中国特产，主产于华东地区。安徽、福建、江西、湖北等省有分布。本省以浙南和浙西北较多。自然生长在海拔 600-1700 米之间，龙泉县的凤阳山、庆元百祖山、遂昌九龙山以 900-1500 米处生长较集中。黄山木兰对气候适应性强，能耐-20℃的低温，能受 40℃的高温。自然分布在年降雨量 1200-1800 毫米地区。但凉爽、相对湿度大、土层深厚肥沃、排水良好、酸碱度 4.5-6.0 的沙壤土中生长最好。

保护措施：已在黄山、清凉峰、天日山、武夷山、井冈山等地建立风景保护区和自然保护区。建议对星散分布的古老植株加强保护管理，保存和发展天然生长的居群，对幼苗、幼树加强抚育管理，促进天然更新。并做好采种繁殖工作。

开花动态：黄山木兰，3 月上旬始花，下旬进入末期，有时可延至 4 月初；冬芽于 3 月中下旬萌动，4 月初展叶，到 7 月中旬开始形成越冬花蕾。月下旬至 5 月开花，9 月果实成熟。

1kg.org 多背一公斤

爱自然 | 更爱孩子

架构师 8 月刊

每月 8 日出版

本期主编：曹云飞

总编辑：霍泰稳

编辑：李明 胡键 宋玮 郑柯 朱永光 池建强

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：曹云飞，**InfoQ** 中文站 **Java** 社区编辑

曹云飞，InfoQ 中文站 Java 社区编辑，西安交通大学计算机软件硕士。从事软件开发多年，包括 Web 应用，桌面应用，前端，后端开发，热衷于计算机理论与应用技术的钻研，软件架构与敏捷开发。

InfoQ Enterprise Software Development Community

《架构师》月刊由 InfoQ 中文站出品。

所有内容版权均属 C4Media Inc.所有，未经许可不得转载。

时刻关注企业软件开发领域的变化与创新

架构师

www.infoq.com/cn/architect

每月8号出版

InfoQ neue
Enterprise Software Development Community

商务合作：sales@cn.infoq.com
读者反馈/投稿：editors@cn.infoq.com