

架构师

1月 ARCHITECT



特别专题

DevOps @SomeWhere

DevOps，不是一个传说！

DevOps之于Prezi

Rafter的DevOps

细说 Spring

我为何停止使用Spring

用Spring实现非端到端验收测试

Spring框架依旧蓬勃发展

重构遗留程序的一次案例学习

JavaOne 2013综述

InfoQ
Venue



每月8号出版

卷首语

行业里的“聪明人”

做餐饮的为了降低成本用地沟油，卖赣南脐橙的嫌橙子熟的不够早要给青橙子打蜡上色，开医院的因为手术赚不到几个钱就要有事没事拉人做CT，做SP的天天琢磨怎么吸费，做黑产的发垃圾邮件还不满足非要去飞单……

不知道从什么时候开始，这种事情在天朝似乎成了一个司空见惯的套路：

step 1：一开始，有一个还算有点利润的行业

step 2：这个行业里有几个“聪明人”嫌赚钱不够快，找到了一些快速赚钱的法门，但这些法门实际上是骗钱的法门

step 3：这几个“聪明人”赚到了比同行多一些的钱

step 4：骗钱的法门很快在行业里传开，大家都学会了

step 5：这个行业赚不到那么多钱了，“聪明人”决定撤出这个行业，做其他事儿去了

step 6：骗钱的法门漏到了消费者/用户的耳朵里，引起舆论风波

step 7：消费者对该行业的信任度降低到负值，将整个行业视为骗子，能不用不吃的尽量不用不吃

step 8：这个行业大幅缩水，本来能赚到一点钱的从业者也赚不到钱了，有条件的玩家跑去做其他事儿去了

step 9：被迫留在这个行业里的玩家只好寻找新的骗钱法门以寻求生存之道

.....
无限恶化

最近跟一位做IaaS的创业者聊，他跟我吐槽说：

我认识很多企业，规模还不小，都是十几二十人到四五十人的企业，你不能说他是小公司。那些企业刚开始都是某某云的客户，但是他们都走了。走的原因不是某某云贵啊，他们天天打折。走了，是因为某某云让他们不放心，一个月坏两次。坏了之后打电话，打电话接线员是个小姑娘，完全不懂技术，只能给

你登记。登记了下来之后你要等，等着售后服务的工程师给你回。那个回来的电话还不一定是给你解决了，可能就告诉你说：对不起，张先生，你这个已经丢了。我记得是去年8月份，盛大云无锡机房发个通告说，I'm sorry，所有数据丢失，各位用户请以后自行做好备份，这是他们解决问题的方法。你告诉我，哪个企业老板还敢用云计算？这不是用哪个云的问题。比如你是一个养鸡场的老板，你不会去问什么叫阿猫云，什么叫阿狗云，你肯定一拍大腿说，云计算都是骗子。

所以我跟大家说，做公有云，第一是要有公德心。因为你伤害的是整个行业。

当时我写到这里，觉得公德心靠不住，一定要有一套制度，把那些骗子都罚的把骗到的钱连本带利都吐出来再让骗子蹲破牢房、永世不得翻身才能够解决这个信任缺失的问题。诚信缺失就是要重罚才能解决，但只有诚信社会才真正做到了严厉惩罚诚信缺失，所以与诚信社会接轨才是唯一的出路。这是当时的想法。

过了几天，忽然想起一件事。去年5月的淘宝十周年庆典，我也在黄龙体育场，听了马云的卸任演讲。他讲的大部分东西我都忘了，就记了一句话：

“因为信任，所以简单。”

大意是他做CEO很容易，因为大家信任他；陆兆禧做CEO还会比较困难，因为信任尚未建立。他希望阿里人能像信任他一样信任陆兆禧。

淘宝这么大一个奇葩，我怎么就把它给忘了呢？十年前他们做的事情不仅仅是做了一个电商网站，更重要的是在天朝的一个角落将信任重新建立，并将这信任又重新扩散至全国。我相信十年前的淘宝卖家和买家中也一定出现过“聪明人”，我们完全有理由猜测，当时的历史完全有可能走向另一条路，就是早期用户受骗，大呼上当，奔走相告“网购都是骗子”，于是网购从萌芽期没有发展到茁壮期，而是发展成了灰色产业，跟现在的电视购物称兄道弟去了。这样一个历史轨迹出现的几率，要比我们现在进入的这条历史轨迹的发生几率大多了。

这是要小心翼翼的绕开了多少坑！淘宝走通了这条路，这说明各个行业也完全可以自己建立一个限制“聪明人”不当获利、继而惩罚“聪明人”的规则。国内的IT基础设施一直提供不了QoS，可能一半是技术原因，一半是因为门槛太低总是打价格战的原因。但现在市场成熟，众口难调，总有用户不会只盯着便宜货看。这些客户稍微多一点，养起一个市场应该还是大有可为的。



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

目录

人物 | People

专访：UCloud季昕华谈网络存储的局限性、SDN在云上的七大应用场景、以及为什么OpenStack还不适合做公有云

观点 | View

API业务模型：如果通过API获取回报
软件正在吞噬组织机构中的“烟囱”

本期专题：DevOps @SomeWhere | Topic

DevOps，不是一个传说！
DevOps之于Prezi
Rafter的DevOps

推荐文章 | Article

重构遗留程序的一次案例学习
JavaOne 2013综述：Java 8是革命性的，Java回来了

特别专栏 | Column

我为何停止使用Spring
用Spring实现非端到端验收测试
虽然遭遇Oracle的挑战，Spring框架依旧蓬勃发展

避开那些坑 | Void

关于Cassandra的错误观点
Promise/A的误区以及实践

新品推荐 | Product

Ionic HTML5移动框架发布Alpha预览版
Reactive Extensions for C++简介
ASP.NET Identity 2.0预览版增加帐号确认、密码重置和安全令牌提供服务
Google眼镜开发工具箱允许开发者使用Xamarin.Android构建Google眼镜应用
Elastic Mesos服务实现EC2中集群自动化部署
推特开源CocoaSPDY

QCon

International
Software Development
Conference

全球软件开发大会

2014年4月25—27日 北京国际会议中心

北京站 2014

QCon北京2014 部分出品人团队



- 扩展性、可用性与高性能

杨卫华

新浪微博架构师



- 团队文化专题

段念

豆瓣网工程副总裁



- 尖端之上的Java

朱鸿（一粟）

阿里巴巴资深架构师



- 大数据应用与大数据处理技术

吴甘沙

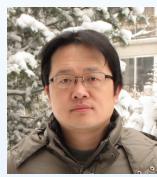
英特尔中国研究院首席工程师



- 移动应用案例分析

蒋炜航

网易技术总监



- 知名网站案例分析

池建强

瑞友科技IT应用研究院副院长



- 移动时代的前端

鄢学鹏

天猫前端团队负责人

大会门票持续热卖
2月19日前报名，享8折优惠

会务咨询 : qcon@cn.infoq.com

咨询热线 : 010-64738142 89880682

精彩内容策划中，讲师自荐/推荐，线索提供 : speakers@cn.infoq.com

更多经典专题 精彩内容 敬请登录 : www.qconbeijing.com

人物 | People

专访：UCloud季昕华谈网络存储的局限性、SDN在云上的七大应用场景、以及为什么OpenStack还不适合做公有云

作者 杨赛

2013年12月11日，UCloud正式宣布已经将盛科的硬件SDN交换机应用在自己的线上公有云环境中。UCloud CEO季昕华在发布会中介绍了SDN在其公有云上满足的七个用户需求，并表示这是世界上第一款SDN硬件交换机在云计算中的应用。

InfoQ中文站对季昕华进行了专访，谈论了对公有云营收的观点、网络存储应用在关键业务上的局限性、SDN的应用场景、在应用商店方面的尝试、以及为什么UCloud不担心OpenStack在公有云上的竞争。盛科软件总监张卫峰对SDN的技术实现方面进行了补充介绍。

InfoQ：上周你也去了金山云的发布会，那天[王育林有一个观点](#)是只有大公司才能做公有云，原因之一就是[IaaS](#)的高成本。你也看到了他们的规模，单是存储就要有上亿的成本。但是你之前在[4月份也提到过你的观点](#)，就是利用跟[IDC](#)厂商、服务器厂商合作的方式，解决了重资产、高投入的问题，同时利用私有云获取更多短期收入。现在到了[2013年底](#)，你感觉这条道路是否能走得通？是否做到收支平衡？成本开支最大的几块是哪些，它们的比重是多少？服务器和其他硬件设备自己采购的和厂商合作提供的比例是多少？

季昕华：做公有云有三大门槛：资金门槛，运营门槛，以及技术门槛。比如Amazon，是资金、技术、运营能力三者皆有，所以做起来了。微软的Azure，有资金、有技术，但是因为运营偏弱，所以一直动静不大。

在三个门槛中，技术和运营更为关键，因为技术和运营需要长期的积累，而且是大型云平台实战的经验积累，在目前的国内外环境都不缺资金，大批的风投手中拿着大量的资金在找好的云计算公司。相反有资金也不一定做得好，为什么这么说？因为当你资源充足的时候，你可能会想通过资金解决问题，而不是靠技术来解决，比如用比较昂贵的存储设备等等，而这其实不是做云的正确道路，这时候你那充足的资源反而成了制约你长期发展的瓶颈。

对于收支平衡这块，目前我们还不是很在意。我把云计算的发展分为三个阶段：

1. 替换服务器和IDC的阶段。这是一个用批发+虚拟化模式取代零售模式的阶段，利润相对不高。
2. 取代软件公司和软件渠道阶段。当云计算成为软件渠道中心的时候，云计算公司自研和代理的软件利润就比较高了。
3. 取代集成公司、咨询公司的阶段。在这个阶段，以前做集成、咨询的公司要么转而做云计算服务的集成和咨询，要么就被淘汰。

现在的美国云计算市场已经在第二个阶段，市场特征已经相当明显。比如AWS有了RDS服务之后，很多企业就不再会采购Oracle或者IBM的数据库了，因为RDS已经足够成熟。

中国的云计算市场现在还处在第一个阶段，当然我们UCloud也是，这个阶段主要靠规模化取胜，通过规模化来降低成本，提高利润率，更大的利润是在进入第二阶段之后。

目前的成本方面，人员研发投入的比重最大，大概在四成多；网络和服务器设备方面的投入其次，在三成多；剩下的三成则用于销售、市场、技术支持方面。其中，服务器和其他硬件设备目前主要仍然是通过合作提供的，包括分成模式和租赁模式。未来肯定会更多的购入自己的服务器以实现更高的利润。

InfoQ：我注意到你们给客户提供了非常好的服务支持，比如，每一个注册用户都会打电话沟通需求，这是你们的一个亮点，在业界也有非常好的口碑。你们支持的成本和投入怎么样？

季昕华：设立专门的售前客户服务，给每一个注册用户打电话沟通，主要出于两点考虑：第一点是为了更多的了解用户需求，避免他们进行不合理的配置，浪费了资源；第二点则是用于做客户的筛选，因为我们目前主要还是以服务企业客户，尤其是游戏行业的企业客户为主。我们希望给这些客户创造一个干净、专业的环境，所以我们会做相应的过滤。

InfoQ：你们的UDisk产品推出一年多了，还处于测试试用阶段。请问，由于UDisk的弹性和分布式的特性，比如提升高冗余，如果我有重要的业务希望我的VM上的数据全部放在UDisk上，是否推荐我这么做？IOPS和吞吐量是多少？SLA是多少？是如何保证的？

季昕华：重要的业务要看场景。如果是IO高的业务则不推荐放在UDisk上，这是网络存储的实现所决定的，包括AWS EBS、阿里云，IO普遍都是30~40M bps，对于此类业务普遍无法支持的很好。目前我们正在研发类似高IO的EBS

的细分产品，希望未来可以解决这个问题。

对于非高IO类型的重要业务，都是可以放的。

UDisk的可用性是3个9，因为网络有很多不确定性。可靠性方面则是11个9，我们对UDisk的冗余机制很有信心。

InfoQ：本次跟盛科合作推出的**SDN**，什么时候能用上？**SDN**的核心用了哪些开源组件，自研了哪些组件？盛科提供了哪些技术？**SDN**底层是否使用了万兆交换机？**SDN**用于哪些应用场景，用户如何受益？

季昕华：跟盛科合作是从一年多前开始，具体投产的话，硬件SDN交换机其实从今年7月就开始用了，经历了几个试点客户测试、反馈、改进的周期，完成了逐步迁移、灰度变更，目前已经在全国四个机房完成了全面部署。

具体技术上的合作，我们主要做控制端的开发，其他都是盛科做的，万兆交换机有部分用到，主要仍是千兆交换机。详细的情况卫峰可以做一下补充。

张卫峰：我们负责数据转发面的工作，主要从这几方面做了创新和优化。

1. 将OVS流表查找offload到TOR交换机，节省服务器CPU资源，提升查找性能
2. 将GRE tunnel的封装、解封装offload到TOR交换机，让网卡仍然可以对TCP报文进行分片加速，大大提升网络性能
3. 所有流表全部是proactive预先配置，有效减少未知报文广播和流表学习所带来的性能问题

后续还会继续做的，包括

1. 将L3 gateway从一个集中的网络节点offload到TOR交换机
2. 在TOR交换机上启用ARP代理，避免ARP广播

通过这些措施最终达到的目的是，将网络处理的部分工作从服务器移到TOR交换机，让服务器可以专注于计算，带宽损耗可以降低，CPU负载降低，一台物理服务器可以容纳更多虚机，帮云服务提供商降低成本，提升用户体验。

同时，由于GRE tunnel封装放在了TOR交换机上，TOR交换机可以直接看到用户数据，可以对用户数据进行统计，限速等策略应用，解决了纯软件方式带来的网络可视化问题，有助于云服务提供商对网络进行诊断和管理。

对于最终用户来说，他们看不到SDN，但是这并不代表他们没有享受到SDN带给他们的好处。公有云平台的用户希望是可以进行自助服务的（self-service），他们自己创建虚机，自己创建虚拟网络，自己创建虚拟路由器，自己创建防火墙等等，谁在底层支撑着这些动作的顺利执行？是SDN架构。没有SDN，就没有这一切。SDN的最高境界就是用户在享受着SDN带来的便利但却并没有意识到自己使用了SDN。

季昕华：SDN的应用场景和用户收益的方面，刚才在发布会中也讲到过。SDN更多是一种隐藏的特性，不是作为独立的服务提供的，但我们的VPN产品、安全产品、广播服务、混合云的实现，都是基于SDN来实现的；同时有了SDN之后，我们的自动化、隔离、物理机与虚拟机混合管理的方案也更加完善、更加智能、性能更高。对具体细节感兴趣的朋友可以查看[这篇新闻](#)。

InfoQ：目前，**Uhost**、**UDisk**、**UDB**、**ULB**、**UScan**这几个方面的研发，各自需要投入多少人？图片存储跟又拍云合作，**CDN**跟网宿合作，测试方面与云测合作，域名解析与**DNSpod**合作，广播服务跟极光推送合作，这几块服务是否也需要投入研发人员，还是只留下一些客服人员即可？

季昕华：目前全公司总共80多位员工，其中技术研发占据一半以上。

合作方面其实都是轻度整合，包括SSO的打通、界面整合之类的，并不会涉及到很多研发工作，而且客户反馈也是转给合作伙伴去做的。我们主要是做好运营，以及协助合作伙伴做好客服工作。

InfoQ：我看到**UCloud**上现在提供了网店系统这个应用级服务，这是跟**shopex**的试点合作吧，现在的用户量多少？

季昕华：这是我们的一个试点项目，主要是想尝试软件渠道中心的思路，从目前来看，这样的合作，对用户来说还是非常方便的，后续还会继续扩大，但目前我们的核心还是在游戏行业。

InfoQ：您对于**2014年UCloud**的技术投入方面有哪些规划？未来还计划推出哪些服务？对于云平台提供应用这事未来是如何规划的，是否会推出类**AWS marketplace**的产品？又是否考虑**PaaS**？

季昕华：我们的核心主要还是两块：一个是聚焦我们行业客户的需求开发功能

， 在我们来说主要是游戏行业的客户；第二个还是我们自身产品的稳定性。

应用超市是可能会做的，可能做一个用户自己提交应用的平台，然后可以像现在那个网店系统一样购买使用，但这个并没有具体的计划。

PaaS也是一样，如果有客户需求的话，我们会优先考虑找一个合作伙伴来一起做，而不会自己上。

InfoQ：在底层技术提升、将技术更好的产品化、为产品寻找更好的商品化道路、发展合作伙伴、发展客户这几项工作当中，您认为哪项是当前的您最为关注的？您和莫显峰、华琨现在是如何分工的？

季昕华：莫显峰是我们的CTO，负责技术这块；华琨主要负责销售和售后；我吗，就是做各种杂事儿的。

合作伙伴这块我沟通的比较多。关于产品反馈和合作伙伴沟通这一块，我们是这样一个流程：首先，华琨那边是负责跟用户沟通的，他那里收集到的反馈或需求会反馈给团队，看看这些需求中哪些是我们现在的产品无法满足的；我们很注重生态链的合作，会先去看这个需求有没有市场上成熟的产品可以满足的？如果已经有了，我们就不做了，跟这个成熟的产品供应方建立合作伙伴关系，将其引进到我们的平台；如果没有的话，就交给莫显峰去研发。

InfoQ：最后一个问题是，担心**OpenStack**在公有云领域的竞争吗？

季昕华：由于有一系列大公司的支持，加上OpenStack本身的架构设计的非常不错，OpenStack最近几年社区很活跃，发展也非常快速，为云计算的落地推广起到了很大的作用。不过从我们来看，我们不担心OpenStack在公有云领域的竞争。我认为OpenStack目前更适合做私有云，不适合做公有云。原因有三点：

1、目前全世界没有特别成功的OpenStack公有云案例。

我们看到AWS、Google、Windows Azure、阿里云和我们UCloud都是用自己研发的产品体系，而使用OpenStack进行大规模部署的我了解到的只有HP Cloud，但HP Cloud目前还在早期发展阶段。

有人可能会说RackSpace也是使用了OpenStack的，但实际上我们看到的信息是OpenStack中的存储部分是RackSpace提供的，但并不表示Rackspace是用OpenStack的。

2、OpenStack是一个软件，不是服务，而公有云是服务。

OpenStack一出来，好像大家忽然觉得是个人都可以做公有云了。但实际上，做公有云，只有一个软件是不行的，你还需要强大的运营来做支撑。

做过互联网运营的人都知道，看到的功能只是互联网产品里面的20%，还有80%是用户看不到的，但是运营很需要的功能，比如多帐号体系，计费体系，安全体系、后台的自动化管理、灰度发布、扩容等等。

3、与开源项目协调的问题。

假设你将OpenStack放到公有云上跑起来了，这时候用户跟你说有一个需求，这个需求是OpenStack现有的功能无法解决的，那你要不要去开发这样一个功能？开发了之后，你是要提交给社区，还是自己留着呢？

如果你提交了这个功能，社区又接受了，那么你就把你辛苦建立的竞争优势拱手让人了。

如果你提交了功能，但是社区不接受；或者你不提交这个功能，那么你将面临升级、维护难的问题。将你自己研发的功能，跟社区发布的新版OpenStack进行融合，会是非常困难的事情，而这个难度会随着时间推移而越来越严重。

而如果是自己研发，就可以做到灰度发布，完全在自己掌握当中。AWS、RackSpace、微软、阿里云、腾讯云、我们UCloud，都是这样运作的。

嘉宾简介

季昕华（[@benjerry](#)），Ucloud.cn创始人，CEO。前盛大云CEO，曾任盛大在线首席安全官、腾讯公司安全中心副总经理、华为研发经理等职位。

张卫峰（[@盛科张卫峰](#)），盛科网络软件总监，数据通信和芯片设计领域资深专家，有十几年的网络实践经验，对SDN、传统二三层交换机、数据传输设备（PTN和IPRAN），从管理面到协议控制面一直到芯片转发面，都有着深刻的理解。

原文链接：<http://www.infoq.com/cn/articles/ucloud-sdn>

相关内容

- [季昕华在UCloud发布会上分享SDN在公有云中的七大应用场景](#)
- [从OpenStack的角度看块存储的世界](#)
- [从“PayPal将弃用VMware转向OpenStack”这则新闻看OpenStack的发展](#)

观点 | View

API业务模型：如果通过API获取回报

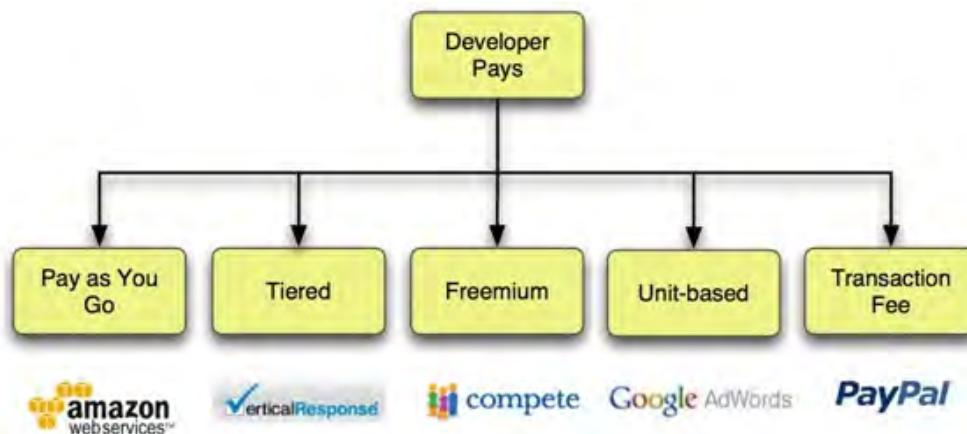
作者 [Saul Caganoff](#)，译者 李彬

John Musser深谙API之道，作为Programmable Web的创始人，他见证了数以千计的API和众多业务模型。最近，John在2013 API战略大会上的[主题演讲](#)中分享了他得经验。

John最常遇到的问题是“你如何用它赚钱？”其实，这个问题的答案，取决于有关“为什么”的反问——“为什么你想要拥有一套API？”人们有许多想要拥有API的理由，而这引出了John在演讲中披露的第一条API秘诀，“API战略并不是API业务模型”。API战略在于“为何”我们想要一套API；而业务模型则是在于我们“如何”用API来赚钱。

回首2005年，API方兴未艾，彼时恰逢Google Maps问世，API业务模型有四种核心类型：“免费”、“开发者付费”、“开发者得偿”和“非直接收入”。在今天，时间走到了2013年，API业务模型中仍旧存在四种核心类型；但每一种中，都已经拓展出许多种提供API并获得套现的方式。所以，John的第二条秘诀是：“大部分API都拥有不止一种类型的ROI。”

接下来，在主题演讲中，John深入分析了每一种核心业务模型，首先他从“免费”开始。与流行的观点相反，“免费”并不是主要的API业务模型。在分析中，John以Facebook为例子，同时他还以所有政府和公众领域API为补充——但这些免费API在全部API业务模型的世界中仅仅占据了“很小的一部分”。



John描绘的第二个分类是“开发者付费”。在这种模型下，开发者使用API提供的服务并支付费用。这一分类包含了许多子类别，例如“按需付费”——开发者仅需为他们实际使用的服务或资源付费。John在这里以Amazon Web Services (A

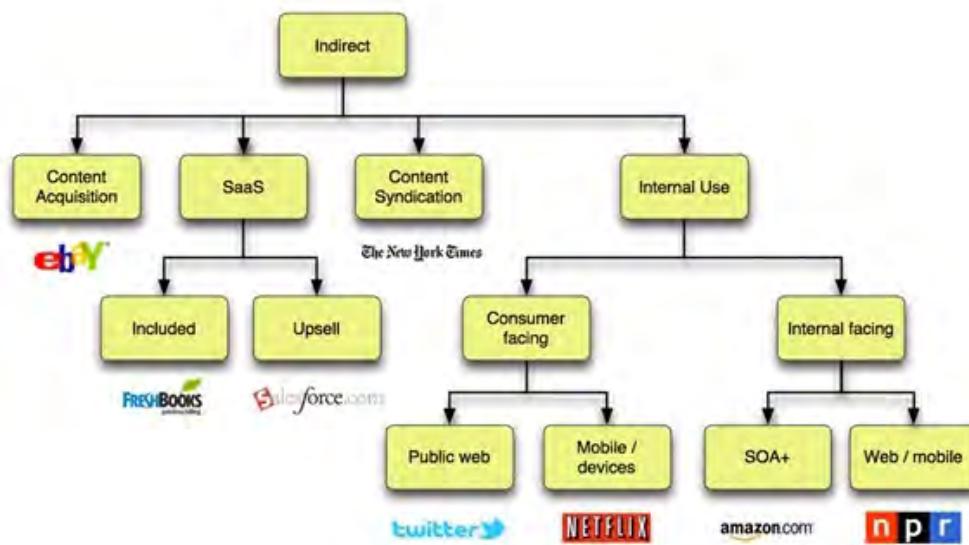
WS) 为例，在AWS价目表中这样写道：“用户只需要为了使用的部分付费。没有最低消费限制。”第二个子类别是“阶梯”模型，诸如Mailchimp等许多公司提供这样的模型。当用户使用的服务总量较高时，将享受到较低的单价。“免费增值 (Freemium)”是一个著名的模型，使用这一模型的API免费提供其基本特性，但如果用户希望叠加一些服务（例如额外的API、更高的SLA、服务专员等），就需要支付相应的费用。John指出，Compete和Google Maps正是免费增值模型的典型例子。而“按单元定价”则是另一个子类别，Sprint便提供这样的业务模型。在这里，API按不同特性收取不同的费用。最后一类“开发者付费”的例子是“事务费”，类似于API的付费用户，他们按处理事务的百分比付费。John列出了PayPal、Stripe和Chargify，作为这一模型的例子。



与“开发者付费”相对的模型是“开发者得偿”。这是John介绍的第三大类别，它同样包含许多子类别，例如，亚马逊合作营销计划 (Amazon Affiliate Program) 提供的“签约收入分成”模型，在这一模型下，开发者将从客户推荐（指推荐其他开发者签约成为亚马逊提供的API的客户）中获得报酬。另一个子类别是“消费分成”，开发者将从推荐的购买中获得提成。在这里John给出的例子是mysimon.com和howstuffworks.com，他们都使用比价网站shopping.com的API，

来支持产品对比并从引导的点击中获得收益。John指出，签约收入分成并不是一个“微不足道的业务”。Expedia从使用其API的联盟网络——价值高达20亿美元/年——中派生出的收益，占它总收益的90%。最后则是“经常性收入分成”模型，rdio.com等公司采用这一模型，已签约客户介绍新客户进行订阅购买后，将能够在该新增客户的订阅期限内一直获得收益分成。

John的第三条API业务模型是“令业务模型与API融为一体。”这是John希望听众能够带走并思考的最重要的秘诀。例如，亚马逊合作营销项目就是其零售业务模型的自然延伸。



作为第四种业务模型，John探讨了“非直接收入”这一类型。在他看来，该类型包含了一些最有趣的API变现方法。John介绍的第一个子类别是“内容获取”。eBay和Twitter等公司需要快速获得内容以实现增长，因此他们使用API来促进内容获取。eBay的API允许超级用户在eBay市场中创建大量列表。Twitter几乎只通过API和第三方应用派生和分发内容。“内容传播”是另一个子类别，纽约时报就采用了这种方式——纽约时报拥有很多内容，并通过API来聚合内容并提供给合作方。

SaaS追加销售是另一种“非直接收入”类别中的业务模型。Salesforce.com提供访问其平台的API，但仅限购买企业授权的公司使用。Salesforce.com认识到了API集成的重要性，并藉此来吸引客户升级到更贵的订购。John将API刻画为“任何SaaS的粘合剂。”API集成为SaaS应用增加了价值，并提供了粘性因素——能够显著减少客户流失。John的第四条API业务模型秘诀是“API业务模型并不是万金油。”

John介绍的最后一种业务模型是“内部使用”模型——企业使用API来支撑其自身业务。NPR开发了一套API，用于向网站、iPad和移动应用进行内容交付。Evernote（大陆地区提供的中文版服务名为：印象笔记）的API流量中，99%以上来

自iPad应用、移动应用和合作应用。而Netflix将它的API用来支持向超过800种不同类型的设备提供内容交付。因此，John的第五条，也是最后一条API秘诀是“内部使用或许是API最主要的用例。”

关于作者



Saul Caganoff是[Sixtree](#)（一家澳大利亚系统集成方面的顾问公司）的CTO。作为架构师和工程师，他在澳大利亚、美国和亚洲的重大集成和软件开发项目中积累了广泛的经验。Saul的专业领域涉及各个级别——包括企业级、解决方案和应用——的分布式系统架构、复合应用、云计算和云API。

查看英文原文：[API Business Models: 20 Models in 20 Minutes](#)

原文链接：<http://www.infoq.com/cn/articles/api-business-models>

相关内容

- [业务流程层的API](#)
- [MuleSoft副总裁James：API战争不可避免](#)
- [API设计中人的因素：专访Apiary的Jakub Nesetril](#)
- [Apigee现在支持Node.js 并开源了Volos](#)
- [MuleSoft开源用于设计RESTful APIs的工具RAML Tools](#)

观点 | View

软件正在吞噬组织机构中的“烟囱”

作者 [Luke Kanies](#)，译者 李彬

Marc Andreessen曾说过，“[软件正在吞噬这个世界](#)”，而在数据中心领域，我们也正在目睹这一潮流。软件正在接管那些过去一直由硬件实现的功能；而对于人们如何共事，这一变迁正在引发戏剧性的改变。

基本上，整个云计算的发展都与用软件来取代或抽象硬件有关。它始于虚拟化，已经走过了数十年的时光，但只是在最近10来年间才在商品化硬件方面普及开来。

将硬件与其上运行的服务解耦，将让大量服务器得以合并，从而加速乐硬件商品化，并进一步降低制造商的底线——他们已经处于勉强维持利润率的境地了。尽管对硬件制造商来说这并不是最佳产出，但对于VMware这样的创新型软件制造商来说则是非常好的消息。同样，技术买方也乐得如此。

很快，人们就注意到了，虚拟化并不只是用来节约资金的——它还提供了新的可能。物理基础设施的变动率受到两方面的制约，分别是供应链，以及诸如服务器上电或固件升级等工作所需要的时间。但一般来说，虚拟基础架构则是通过API工作，因此会明显快很多。过去在物理基础设施上进行的需要数周甚至数月的事情，在虚拟服务器和网络上操作的时候，可能往往只要几分钟或几小时。

在2009年，我们终于看到了虚拟服务器的数量超越物理机。这意味着现在大部分从事基础设施工作的人，已经不再需要在硬件上操作了。讽刺地是，这可能会创建一个新的前所未有的烟囱（注：Silo，在本文中指企业内，处于互相独立状态、缺乏横向联系的垂直管理体系，如开发、IT、运营等）：专注于维护从硬件到虚拟机层面的团队。同时，这个星球上的其他人则在纯软件的环境中执行操作。

好吧，至少一定程度上是这样。事实上大部分组织机构都已经虚拟化了他们的计算资源，并且或许已经迁移到商品化硬件上，来实现防火墙或负载均衡器（基本上，使用的是运行Linux或BSD的x86盒子，并叠加了某些特殊的软件）。但是他们的存储和网络资源依旧侧重于硬件，即使其上的商品化IP设备，其行为也是尽可能地按照硬件设备的方式，而不是类似软件的表现。

下一个重大的发展，不仅会把计算抽象到软件层面，还会抽象存储和网络。这是一种向着完全由软件定义的基础设施的转变，将为采用它的组织机构开启无数的

可能性。

DevOps和软件定义基础设施

对于云计算来说，DevOps既非因也非果，不过DevOps倒的确是一项与云计算关系紧密的运动。基本上它算是一种文化变革，其目标是打破开发者与IT系统管理员之间泾渭分明的状态，从而以更高的频率，更好地部署软件，并更好地匹配业务需求。

开发与IT之间的这种分隔之所以会存在，不仅仅是因为开发者和运营人员的兴趣不同，而且实际上他们的典型工作场景甚至会发生冲突。系统管理员被要求保持IT运行平滑可靠——没有故障，没有宕机，而且能够根据业务要求按需扩展。开发部门则处于另一个方向，他们被鼓励频繁发布新代码，从而在竞争中保持领先。在IT运营部门看来，开发者们发布充满bug的代码，而留给IT接手时则会让IT难以开展自己的工作。而在开发部门看来，IT运营部门对流程和程序的坚持，则阻碍了开发工作的开展。

不过，如果我们将基础设施看作软件，那么就可以将它当作一个正在运行的应用，并围绕运营构建软件开发工作流——比如应用升级、软件补丁等，从而让系统管理员和开发者结成一线，和谐地工作在一起。如果我们认为基础设施只不过是服务器、交换器和路由器，那么刚刚说的这些就不可能了。但一旦我们将基础设施视作一系列服务，那么很明显我们就可以把它当作软件来对待和操作。

DevOps应归于文化，云计算应归于技术——然而在现实中，云计算必须与二者同时相关。尽管技术无法解决“政治”问题，但会有助于缓解。云计算技术能够让应用开发者围绕着无法提供软件世界观的IT部门工作。但对于能够获得完整的软件定义基础设施的人来说，传统的系统、网络和存储的烟囱正在被打破。单独一支团队就能够构建并部署新应用，支持更高的敏捷度、更快的速度，同时从软件就绪到真正将价值交付给目标用户所需要的时间也会大幅减少。

新型IT与新型系统管理员

我们正在见证一类新型管理员的兴起，他们用基础设施能力和服务的方式进行思考——而不是沿用设备的思维方式。对于运行在类似亚马逊AWS等环境里的应用，我们已经在其管理员身上，看到了这场思潮。在大部分情况下，这些管理员无需考虑网络或存储，他们只需要围绕亚马逊提供的软件，构建自己的基础设施。

这种将基础设施作为软件的重新定义，正在融入企业——一般通过基于OpenStack等技术的私有云。即使现代私有云并不能完全取代传统上基于设备的企业基础设施，但是他们依旧正在推动组织机构，非常迅捷地朝着将基础设施作为软件的

方向前进。哪怕能够在一小时内启动1000个虚拟机——启动、运行并执行真正的工作——但要想建立起新的虚拟局域网或配置好存储，依旧需要10倍的时间，因为做这些事情的人处于不同的烟囱，组织机构很快就注意到自己需要发生改变。

随着云计算对企业的“侵蚀”，我们会看到从纵向团队向横向团队的转化。我们的云计算团队会把自己的工作看作维持API和SLA——它们能够理解全部基础设施的需求。我们的应用团队将依据这些API和SLA来谈论基础设施，而不是讨论操作系统或硬件需求。从硬件到软件的抽象，将从技术界面转移到团队界面——而我们的业务将看到这样的好处。我们会更频繁地发布新产品，更快速地获取客户反馈，而且能够更迅捷地响应市场变化。把基础设施当作软件来治理，并将其作为一种核心能力，将为我们提供与竞争对手相比显著的优势。

毫无疑问，最终我们将发现，我们所有的应用都运行在服务层和按需的基础设施。在商品化硬件之上运行的软件，自身也会商品化。我们的绝大多数应用，将不再拥有独一无二的特殊基础设施要求——相反，它们将能够运行在通用软件平台上，这些平台完全消除了应用层面操心基础实施的需求。

在这个世界上，存储、网络和系统团队（更不必说VMware、防火墙和负载均衡团队）等垂直的烟囱，带来了如此之多的不便——并延缓了业务的发展——将不再有任何意义。在这些领域，我们将拥有负责业务服务可靠交付的团队。毕竟，这才是IT：一项业务服务。

你的团队是在努力迎接这一变革，还是在拼命抗拒这场变革？

关于作者



Luke Kanies，在2005年创建了[Puppet及Puppet实验室](#)——出于恐惧和绝望，带着产出更好的运营工具并改变我们如何管理系统的目标。自1997年以来，他就通过文章及演讲来介绍自己系统管理领域中的工作，而自2001年起他开始专注于开发。他开发并发布了多个独立系统管理工具，对Cfengine等产品的建立做出了共享，并出品了Puppet及其他工具，包括OSCON、LISA、[Linux.Conf.au](#)和FOSS.in。他在Puppet方面的工作成为DevOps中的重要部分，并且正在实践关于云计算的承诺。此外，他还是俄勒冈州软件协会的董事会成员。

查看英文原文：[Software Is Eating Your Organizational Silos](#)

原文链接：<http://www.infoq.com/cn/articles/sw-eating-silos>



独立云分发

全国布点
全网加速

移动互联网
优 化

UPYUN[又拍云]重磅推出独立云分发服务，
国内唯一一家纯静态网络加速服务。

全网屏蔽了一切的动态访问，
只专注于加速静态资源，
以追求最快访问速度，
和极致用户体验。

按需收费

自 定 义
图片处理

高级防盗链

强效防攻击

实 时 监 控
分 析

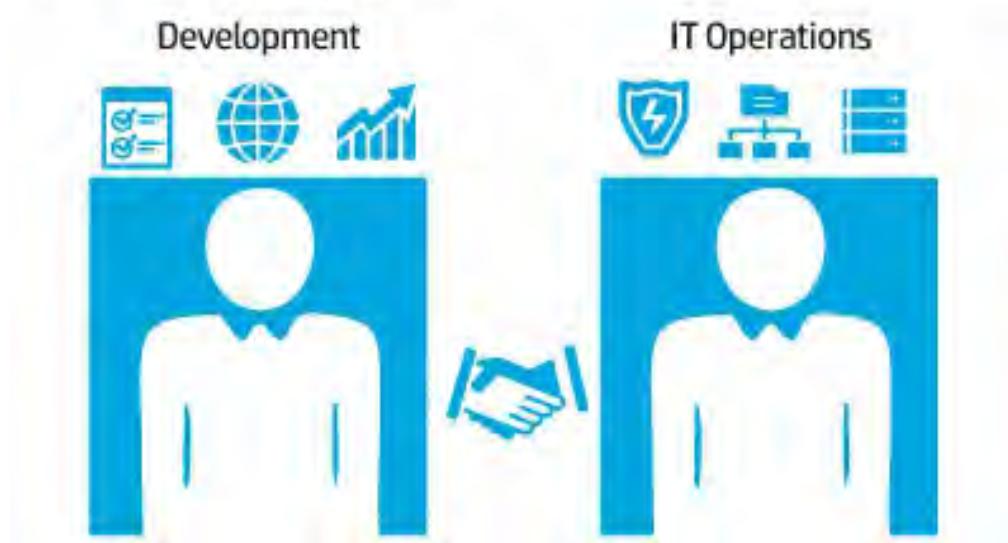
www.upyun.com

专题推荐语

DevOps @SomeWhere

当然，DevOps不乏反对者。反对意见不一而足，有人认为DevOps是个误导（DevOps只是系统管理的一个新名字而已，新瓶装老酒），有人对DevOps不屑一顾（DevOps只是一些疯狂开发者的疯狂想法，他们想摆脱运维人员，或者，DevOps只是一些疯狂运维人员的疯狂想法，他们想像开发者一样工作），甚至有人公开抨击（可惜的很，他们的言论往往毫无逻辑）。通过这一期的内容回顾DevOps在Prezi、Rafter的实际使用案例，您将不会再觉得DevOps是个神话。

One team, one goal



DevOps focuses both the Apps team's drive for agility and responsiveness and the NOC's concern with quality and stability on the ultimate goal of providing business value.

Source: HP

本期专题：DevOps @SomeWhere | Topic

DevOps，不是一个传说！

作者 [乔梁](#)

DevOps最近成了热词，望文生义，你也能猜个八九不离十，它就是在说“研发团队”与“运维团队”之间的那点事儿。那么，到底什么是“DevOps”呢？

WikiPedia上说：“DevOps是软件开发、运维和质量保证三个部门之间的沟通、协作和集成所采用的流程、方法和体系的一个集合。它是人们为了及时生产软件产品或服务，以满足某个业务目标，对开发与运维之间相互依存关系的一种新的理解。”这恰好体现了精益管理中的客户价值原则，即：以客户的观点来确定企业从设计到生产交付的全部过程，实现客户需求的最大满足。我们也可以把DevOps看作是一种能力，在缺乏这种能力的组织中，开发与运维之间存在着信息“鸿沟”。

如何获得这种能力呢？关键有两点：一是全局观：要从软件交付的全局出发，加强各角色之前的合作；二是自动化：人机交互就意味着手工操作，应选择那些支持脚本化、无需人机交互界面的强大管理工具，比如各种受版本控制的script，以及类似于Nagios这样的基础设施监控工具，类似于Puppet、Chef这样的基础设施配置管理工具。

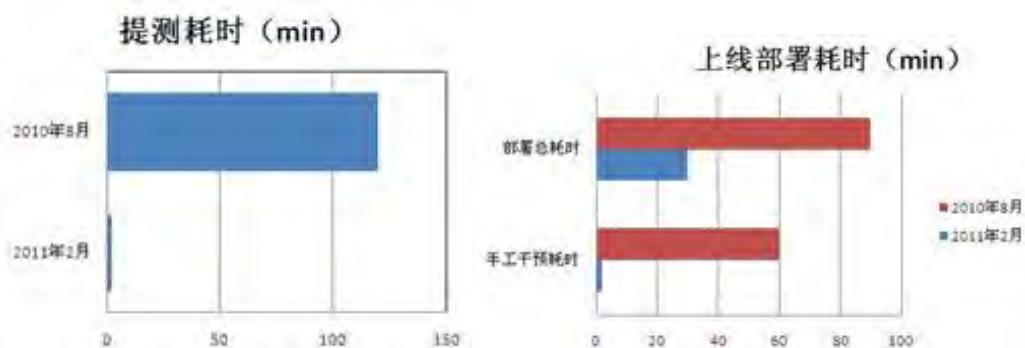


有人评论说：“针对目前国内情况，DevOps还是很遥远。也许只有行业顶尖的公司，或者新成立的公司会有这样的尝试。大多数的企业还未开始进行敏捷的推进，传统的重重阻碍会使敏捷的推进进程遥遥无期。” DevOps真的离我们有那么远吗？DevOps应该从哪里开始呢？

一、让数据说话

让我们看一看百度某产品线在半年内的变化吧。首先要说明两个百度术语。“提测”是指某个项目开发完成后，在正式上线前，将其提交给测试组进行测试的活动。对于客户来说，“提测”这个动作本身并不增加什么价值，但也需要花费一定的时间。“上线”是指某个项目验证合格后，将其部署到服务器的过程，其中包括“上线申请”和“实际部署”两个活动。也许在各公司中对这两个活动叫法不同，但在软件生命周期中，“提测”、“上线”这两件事无论花长时间，大家可能都不会感

到奇怪。下面两张图是该产品线进行改进之后的对比数据。

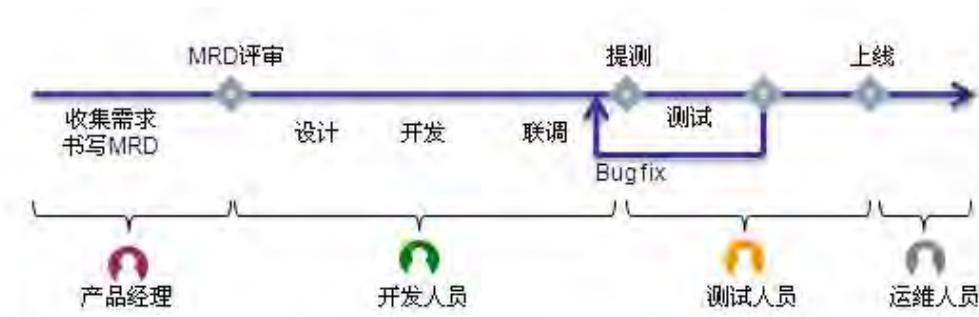


从图中不难看出，提测和上线部署的效率已大大提高。象百度这样的互联网企业，产品线多得数不清，几乎每个产品线每周都有新功能部署。仅从这两个数据来看，其收益可想而知。那么，半年前的状况是什么样的呢？

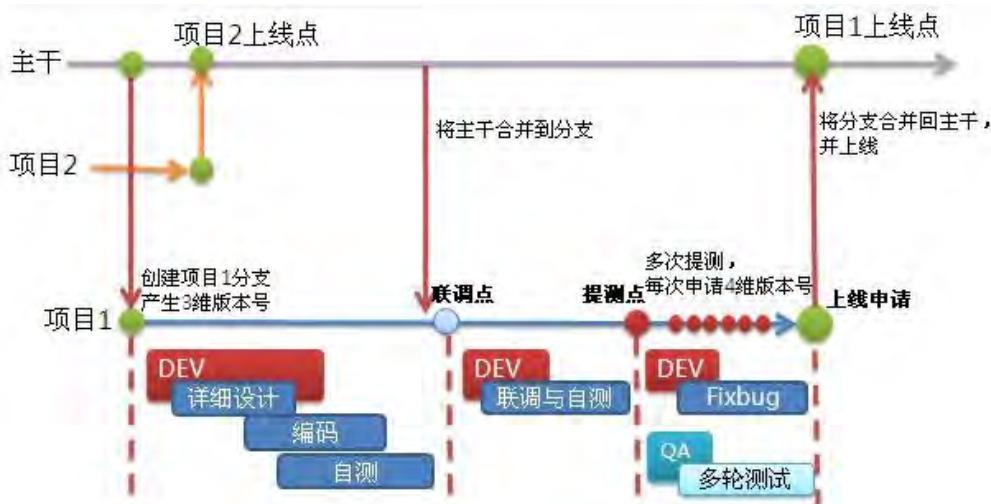
二、流程建模

既然DevOps关注于价值交付的全过程，那就让我们看看该产品线常见的交付过程吧。

对于单个项目来说，它大体上是一个典型的瀑布开发过程。首先是需求收集与整理，撰写MRD(Marketing Requirement Document)或总体设计后，进行评审。如果涉及到多模块，每个模块的开发人员会对各自负责的模块进行详细设计，给出大致的开发计划，并商定联调时间点。之后，开发人员会从主干上拉出项目分支，并在该分支上进行开发。当到最后联调点时，几个开发人员才会在将代码合在一起，进行联调。当调通之后，开发人员再申请提测。测试人员接到提测申请单后，进行测试，记录Bug，通知开发人员修复，直至质量达到标准。之后，开发人员会填写上线申请单，经运维人员确认后，运维人员操作进行上线部署工作。如图所示。



开发的复杂性还在于：该产品线有很多并行项目，为了避免互相干扰可能带来的冲突，每个项目启动后都会重新在主干上拉出分支，在上线前才进行合并。如下图所示。



另外，并行项目太多，导致每个开发人员会同时参与多个处于不同阶段的项目。那些周期较长的项目虽然会被分解成多个迭代，但每个迭代内都是同样的开发流程，只是最后仅有一次上线而已。

总而言之，突出的问题表现在：

1. 同一角色多个人员的合作开发；
2. 各角色部门之间的协作以各自的产品物为目标，如MRD、产品代码、测试用例、上线操作单；
3. 基于人机交互方式的内部流程管理平台。

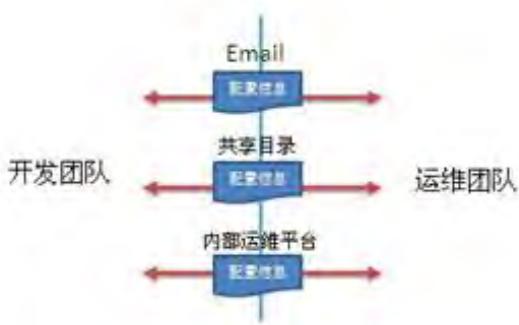
三、发现浪费

从精益思想出发，为了尽早交付价值，必须首先找出整个流程中的浪费，并将其消除，从而提高流程效率，让“一个想法从提出到实现”可在最短时间里完成。那么，浪费到底表现在哪里呢？

- 一些不必要的多分支开发，合并后发生问题的风险高。多个项目中可能都要修改同一个模块的代码，每次在最后合并代码时都会出现一些问题，非常痛苦，尤其是修改比较大的时候，合并及修复时间较长。
- 推迟问题被发现的时间。每个开发人员会将需求分解成多个技术任务后开发。所以，所有任务完成之前，应用程序一直处于不可用状态。当最后在一起联调时，常常会发现一些意想不到的问题。
- 基于流程平台的沟通。在提测环节中，沟通完全基于内部项目管理平台和即时消息工具或Email。比如开发人员在提测前，需要在项目管理平台上申请该项目的4位版本。拿到4位版本后，才能提交平台统一编译。如果编译失败，那么问题解决后还要再次申请4位版本。如果成功，则在项目管理平台上填写表单，回答一系列的问题（比如，是否做过单测？测了哪些功能点？部署步骤是什么？），发起提测工作流，管理平台会自动发送电子邮件给相关测试人员，通知他们进行测试。测试人员收到该提测工作流后，

必须在平台上进行相关确认操作，通知开发人员已收到该版本。如果测试人员对部署和测试内容有疑问的话，还会通过即时通讯工具或邮件与开发人员进行确认。

- 常规的例行工作很难自动化。上线部署也需要通过内部平台来完成。开发人员拿到已测试通过的4位版本后，先要登录到内部平台，再提交上线申请单，填写上线步骤。当运维人员收到上线步骤后，再将其“翻译”成平台可以识别的“半自动上线步骤”，再让平台来执行。如果运维人员不理解上线步骤，就要和开发人员通过电子邮件或即时通讯工作等进行反复确认。部署配置信息分散在各处。如下图所示：



另外，该产品的一个重要特征是需要不断地尝试调整程序算法策略，以得到最佳的流量效果，而这种调整的频率较高（至少每周一次）。当需要调整策略时，开发人员修改代码后重新进行编译打包，由于产品代码发生变化，所以测试人员仍需要进行大量的回归测试，而运维人员在部署时也需要将对二进制文件包进行整体部署，整个周期比较长。

从上面这些内容中，我们不难发现，流程中更倾向于将问题推迟到后面解决（比如最后集成联调），将工具（平台、邮件、即时通讯）作为协作的基础，而角色间的沟通几乎完全依赖于前一个环节的产物（比如MRD、产品代码、上线步骤）。那么我们使用哪些对策进行优化，达到消除浪费的目的呢？

四、应对措施

1. 无人工干预方式的脚本自动化

- 自动化提测——由于已做到了每日集成，所以每天都有可测试的版本，开发人员不再需要为提测进行专门的准备工作，只要从成功构建的列表中选择一个给测试人员就可以了。使用Hudson平台后，通过插件即可调用自动化脚本，完成提测版本的标识。
- 统一配置信息源——将所有的配置项全部放在Subversion库中进行版本控制；并根据应用环境的不同，分别保存在Dev, Test和Online三个目录中。



- 常规流程脚本化——经过各角色的共同讨论和可行性分析，最后配置上线部署的实施方案是：由开发人员将产品二进制包与配置项进行剥离，这样仅做策略调整时，测试人员只要对已修改的配置项进行相关测试即可。运维人员用一系列的脚本代替了内部运维平台的手工上线操作，再通过Hudson平台的插件，以 "Click Button"的方式达到了一键式部署。

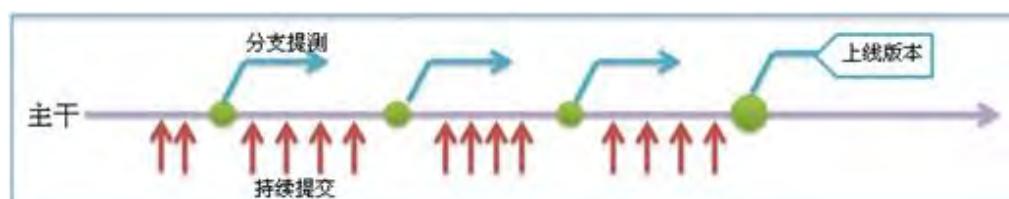
2. 尽早发现问题，解决问题

- "需求细分，及时开发，及时验证"——将需求拆分成端到端可测试的需求（即"用户故事"），这些需求一般可在3天内完成。在实现每个需求之前，开发人员与测试人员进行充分沟通，对需求与验收条件达成共识。每开发完成一个用户故事，就进行测试，并用自动化测试进行覆盖。
- "主干开发，分支提测"——将原来的多个分支进行合并，统一在主干上开发，每周结束时拉出一个分支，进行提测，一旦发现问题，就在主干上修复。
- "持续集成"——为了确保每次提交质量，对主干开发建立持续集成环境，开发人员和自动化测试人员都严格遵守持续集成纪律"Check-in Dance"。

新的开发流程如下图所示。



分支开发策略变更为Single Branch模式。



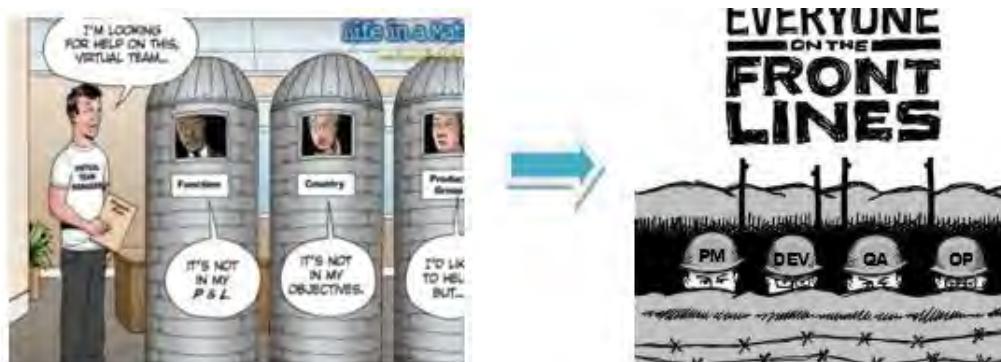
五、小结

通过以上改进措施，让团队的合作方式发生了重大变化，从"碉堡防御"走向了"

战线统一"。

原来，各角色仅关注于自己本身的工作，虽然大家都同处于一个项目中，但各自划分了"领地"，产品经理就应该将MRD写得清清楚楚，如果开发人员认为不清楚，那就回去再改。开发人员只管按照MRD上的内容进行开发，很少考虑可测性和易测性问题。测试人员只管按照MRD中内容来测试，有问题通过内部工作流平台提交问题单。运维人员只管根据开发人员提交的上线操作单进行操作。似乎各角色之间的沟通介质只有各自的"交付物"。

现在，各角色都能够共同合作，以项目的最终交付为目标，积极讨论需求，优化实现。因为角色之间的这种紧密合作让所有人对不同角色都有了深入的了解。开发人员耐心为产品经理解释技术实现，说明计划安排，测试人员与开发人员共同讨论验收条件，避免遗漏需求。开发人员让运维人员了解架构设计，细心听取运维人员的建议，进行技术改造，使部署工作更快捷有效。



通过这些活动，大家都认识到原有内部管理平台仅是个公文流转的支撑平台，要想提高工作效率，就要将这种"办公自动化工具"进一步提升为"全面自动化工具"，使所有人更关注于端到端的价值，而非各角色之间的分界点。

六、结束语

百度刚刚开始敏捷之旅，还没人谈及"DevOps"运动，虽然还没有什么强大的工具支撑，但基于"提高效率"的朴素思想进行的过程改进也带来了"DevOps"效果。可见，DevOps听上去很神秘，但其实并不难。只要本着精益思想，聚焦于快速交付价值，不断发现并消除浪费，你也一定会有很大收获。

感谢**熊节**对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家加入到[InfoQ中文站用户讨论组](#)中与我们的编辑和其他读者朋友交流。

原文链接：<http://www.infoq.com/cn/articles/devops-not-legend>

本期专题：DevOps @SomeWhere | Topic

DevOps之于Prezi

作者 [Peter Neumark](#)，译者 李彬

本文是“[月度DevOps战报](#)”系列文章的第二篇。每个月我们聆听DevOps为不同的组织带来了什么，我们学习哪些有效或无效，并描绘出采用DevOps过程中所面对的挑战。

从内部学习DevOps

引言

Prezi中[像“微创业”一样运作](#)每个产品团队的哲学是尽量去中心化，意味着每个团队都需要自给自足。要实践这种哲学，唯有使用经典DevOps方法将软件开发人员和有大规模系统运营经验的工程师们进行混编。Prezi的管理团队注意到了这一点，因而开始在全公司范围推广DevOps。随着这一行动的展开，员工如何发展实践DevOps所需的技能组的问题变得特别重要。

长期以来，Prezi拥有由设计师、QA工程师、开发人员、用户体验研究员和产品经理组成的跨职能团队。尽管如此，直到最近Prezi也只不过拥有一支[“DevOps团队”](#)。目前这支团队依旧负责大部分关键任务服务的可用性，例如核心数据库和HTTP负载均衡器等。与以前不同的地方是，各个产品团队假定拥有运行其产品或特性的基础架构的所有权。其中许多团队都没有任何有DevOps经验的成员，因此学习DevOps的基本知识成为它们的首要任务。

传统上，学习一系列技术栈（注：指与某个主题相关的一系列技术组合）需要参加一些一般由供应商组织的课程，并且往往还会获得一份认证。尽管这条经过实践检验的途径拥有许多便利，它依旧是一个笨重的方法，需要公司和员工个人投入大量的金钱和时间。那些熟悉入门课程中大部分内容却又没有准备好进入下一阶段的人，会感到枯燥或迷茫。最后，很不幸的是，供应商课程往往忽略了“大局”并聚焦于其产品线所解决的问题，而不是针对那些学员所面对的问题。考虑到这些缺点，如果可能的话，工程师们最好从他们的同事那里学习。

幸运的是，在每个团队都转向DevOps前，Prezi早就已经开始尝试不同的方法在内部分享信息。所以当我们真正需要在公司内部分享我们拥有的知识的时候，我们已经有了有效的方法。这篇文章的目的正是分享这些“技术”。

训练营

如果你熟悉团队伙伴，那么你很容易向他们学习，但是新员工该怎么办？Prezi努力让新加入的工程师跟上代码库和公司价值的发展。解决方案是缩短Facebook令人印象深刻的为期6周的马拉松式训练，创建自己独有的两周训练营，在其中新雇员花一天时间与构成公司的各支团队共处。设计这样一个“迎新周”的目的是传递若干不同类型的知识。

首先，花一些时间与Prezi的每支团队共处，是新人了解其新同事们的好方法。与一般“旋风之旅”中的数千次握手相比，新人更容易通过围绕着有趣主题简短对话或速成课来记住团队成员的名字。除了能够将名字与面孔联系起来，训练营的毕业生们还将知道他们可以找谁来解决某个特定的问题。

其次，训练营为新的DevOps工程师提供了鸟瞰视角，来了解Prezi的基础架构和代码。一般在简短的演讲后，训练营学员会被分派一个小却相当有挑战性的任务。例如编写一个用于配置apache实例的chef“配方”，并针对为了Prezi Desktop使用hadoop数据而导出的prezis的数量创建仪表盘，这些精心挑选的微型项目不仅是在给定时间内可完成的，而且还对学员自身有帮助，而不仅仅是“附加作业”（注：指没有实际价值、强制安排的任务）。学员一般还会要求与两到三位有经验的工程师沟通，以获取一些解决手头的问题的建议。

最后，训练营向新员工展示了在创业环境中工作是什么样的。Prezi按团队而不是部门组织，我们没有太多管理层级，但我们有一个用户体验研究团队。所有这些独特之处可能会对新人造成一些迷惘。对新加入的工程师来说，Prezi的迭代开发风格大概是最难整合到他们的工作习惯中的新概念，而这仅仅是因为这种现象的普遍存在。这里的迭代开发是指遵循[构建-评估-学习](#)这样的循环。在Prezi中，我们尝试采用这种“精益创业”的方式，不止用于重要产品决策，还用在15分钟的小任务以及二者之间任何规模的工作上。每位工程师都有责任通过构建足够好的解决方案来避免浪费，这些解决方案是可测试的，并且如果需要的话，销毁它们也不必消耗大量的时间和精力。Prezi对启用工作系统的迫切期望，与第一时间寻找完美解决方案的任务恰恰相对立，这在学院和大型组织中是很普遍的现象。一位新加入的团队成员谈到，在训练营中听到每个组都在演讲中谈论重写代码或重新考虑流程，这着实能帮助人理解迭代精神。从训练营毕业后，他对Prezi的工作方式了解了更多，而且他还知道他带来的知识能够帮助谁——学习是双向的。

聚会

结识陌生人是寻找新理念的一种可靠途径，而聚会对于寻找其他组织中面对相似挑战的人而言也是个好办法。除了交流“血汗史”外，与公司之外的工程师们沟通还能够作为一种现实问题的检查。例如，如果我们使用个特定的数据库时遇到了

很多麻烦，那么我们就很有必要弄清楚，到底是我们配置的问题，抑或我们采用的这个版本也让其他人吃到了苦头。

Prezi承办若干技术聚会，聚会一般向与会者免费提供披萨饼和啤酒。这是一个经典的双赢局面：对聚会组织者而言，得到一个免费提供的合适场所是一种很大的帮助。对Prezi而言这样的安排则得到以下若干好处。

首先，这便于员工参加与他们日常工作相关的聚会。当我的一位团队成员在[布达佩斯DevOps聚会上](#)做[Apache Kafka](#)方面的演讲时，我们许多同事都到现场观看，因为他们只需要在下班后爬上楼即可。第二天，每个参加的人都在谈论：基于昨天晚上的演讲，我们应该如何重新考虑为服务分配的硬件总量。

其次，这是个培养员工对技术的兴趣的好办法。如果我痴迷于计算机科学或编程语言方面的某个研究领域，我唯一需要做的就是寻找足够多的人来分享我的兴趣。由于我不需要操心场地，组织这样一场聚会的门槛是非常低的。即使聚会主题目前不能在Prezi中运用，也许某一天它会成为解决我们公司面对的某个问题的必要知识。

最后，承办聚会将为Prezi带来工程师人才。对任何穿过Prezi大门的人来说，显然它并不是一家普通的软件公司。当这些人打算跳槽的时候，很有可能会向Prezi投一份简历。

将聚会中提出的一个有趣想法在团队中传播，是引发一场技术讨论的极好途径，这样的讨论可能对所有参与者都有帮助（即使最初的想法被证明无法用于目前的问题）。

便当演讲（Brown bag talks）

讲座也许是古老的分享知识的形式，而它在Prezi中依旧盛行。45分钟到一个小时的“便当演讲”（取名于听众们用来包装三明治的棕色纸袋）是可选的午餐时间讲座。演讲的主题可以是任何内容。

我们定期邀请讲座嘉宾举办这样的演讲。一些令人难忘的演讲包括由匈牙利前驻美大使带来的欧美贸易关系速成课程，以及Spotify的数据服务团队如何应对技术和组织体制挑战的演讲。

任何员工都可以自由地组织这样的演讲。最好的演讲之一是关于paxos算法及其备选方案的。演讲者在加入Prezi之前已经成为该主题的专家。演讲结束后，听众们对什么是可行的折衷有了新的理解。在我们团队里，当学习了数据一致性后，那些有运营背景的工程师疯狂地努力回忆他们是否曾意外地为MongoDB配置了危险的设置。

便当演讲与普通内部演讲的不同之处，在于它非正式的本质。如果某位观众发现这个演讲并不适合自己，他们可以自由离开。因此，只有很少的人会在演讲过程中沉浸于笔记本电脑或智能手机里。这种非正式氛围也有助于演讲者更轻松地传递他的想法，因为在组织演讲内容的时候，他不必考虑是否与整个公司的方向一致。

与即兴谈话相比，这样的演讲活动需要更多的准备，这使其成为与同事分享自己专业知识的有效途径。

编程马拉松(Hackathons)

最可靠的学习方式是“在工作中学习”。不过这并不意味着你必须独自阅读帮助页面并解决全部问题。对团队的成员来说，互相学习共同解决不常见的问题的一种良好方式。

在布达佩斯的工程总部，Prezi定期组织编程马拉松活动并欢迎任何有兴趣的人参加。活动由一场关于有待实践的奇思妙想的头脑风暴开始。员工和访客们围绕着热门理念组成了混合的团队。这些团队在接下来的一天半中将通过“hack”（注：这里指编程）来证明他们理念的可行性。

其中一些团队着眼于与prezi.com相关的项目，但这并非强制要求。一个明显的反例是某个团队编写了一个能够列出当前谁在办公室的Web应用。它通过列出连接到公司WiFi网络的无线设备实现这一功能。这个应用的编写很好地搭配了低层UNIX编程、Python Web应用代码，以及在客户端侧的一些巧妙的JavaScript代码。

编程马拉松是模拟DevOps团队的生活中实际发生的紧急事件的好方法——它们在许多方面都有相似之处。两种情况下，混杂的团队（注：指dev-ops的团队组合）都在处理非常模棱两可的问题。此外，时间是一项制约因素，因此团队需要快速解决方案。最后，每个团队成员能够发挥其任何力量以更好地提供贡献，团队就越有可能成功。在技术因素之外，两种环境中团队成员被设定的角色也是相似的。自然而然地，一些成员会承担协调员的角色，其他人则会选择“多干少说”的角色。无须赘述，一个成功的团队同时需要这两种角色。找出每个团队成员的角色，对于准备应对诸如MySQL主节点宕机导致停工和数据丢失这样的真正的DevOps灾难至关重要。

内部课程

以上介绍的“技术”（注：此处“技术”指公司内部知识分享的方法，而非具体开发技术）都属于知识分享而不是供应商课程。然而，当超过一定技术深度后，这些非正式方法往往就失去了作用，特别是当听众的技术背景过于多样化的时候。

通过可预知的课程安排和适中的速度，学员们有机会完全浸入到课程的主题中。例如，Prezi组织过一场简要介绍Haskell编程的课程，在连续四个周五的早上举行。公司里的每个团队都参加了课程，设计师和QA工程师们，与拥有多年函数式编程经验的开发者坐在一起。课程从基本内容开始，以稳定的节奏讲授了相当多的内容。课程鼓励参与者们互相之间以及向导师（他也是一位同事）提问。因此，关于课程内容如何在日常工作中应用的问题和建议上花了许多时间，即使这导致我们偏离了原来的课程计划。虽然我们只在很少的产品系统中使用Haskell，但这样的灵活性和开放的氛围还是使其成为一次成功的课程。

公司还用类似的方式组织了一个专注于经典计算机科学文章的阅读组。每周都有一位组员介绍阅读列表中的一篇文章。接下来的讨论里，每个组员都可以评论或挑战演讲者对文章的解读。

供应商课程一般由经验丰富的演讲者使用专业教学材料来讲授。这是一种由专家向听众进行单向信息传播的有效方式，其中的一些案例使得参加课程对Prezi的工程师是有意义的。Prezi的内部课程与厂商提供的内容并不是直接竞争关系。相反，组织内部课程的原因根植于这样的现实：单向模型并不是交流的唯一途径。实际上，甚至它也许并不是最重要的一种。毕竟一个DevOps团队解决问题的方式，往往是更非结构化的流程，经常需要某人为自己应对某个问题的解决方案进行辩护，或是进行正确的询问以快速寻找导致问题的根本原因。在提供有用的技术信息的同时，这些内部课程和阅读组也绝对有助于提升那些对顺畅运作一个团队所更需要的软技能。

Prezi正在成为一个演讲公司，这样的氛围特别有利于让员工不仅获得技术方面的成长，还增强分享理念的能力。通过分享能力的提升，工程师将会在技术方面更好的互相帮助。

总结

自从在Prezi开始作为DevOps工程师的工作以来，我从同事们那里学到了很多很多。帮助其他人解决问题并最终成为更好的工程师的感觉很棒。大部分人希望在类似Prezi这样的环境中工作，这里鼓励每个人分享自己的理念，而任何理念都可能受到挑战。Prezi采用的这些替代供应商课程的知识分享技术有助于创造这样的环境。在DevOps团队内促进信息的无障碍流通非常关键，因为团队成员各自背景不同，每个人都可能拥有一些能够帮助其他人的知识。建立团队间的对话也会带来类似的价值。不管公司是否让雇员们参加专业课程，尝试Prezi的这些选择都有机会让公司获得收益。

关于作者



Peter Neumark是Prezi公司的DevOps倡导者。他与妻子Anna和两个孩子住在匈牙利首都布达佩斯。在查找Python代码错误或换尿布之外的时间里他喜欢骑自行车。

原文链接：<http://www.infoq.com/cn/articles/monthly-devops-02-prezi>

本期专题：DevOps @SomeWhere | Topic

Rafter的DevOps

引导阶段

作者 [Chris Williams](#)，译者 陈菲

在过去的6年里，我有着独一无二的机会观察我们公司是如何由几个只想着出租教科书的应届毕业生发展成一个大而成熟的公司。当我回头看时，我会将我们的成长分成两个不同阶段：A轮融资之前和A轮融资之后，与你听到或读到的大部分创业公司不同，我们经历了相当长一段时间的A轮融资（大概有3年）。

在这一发展阶段，我们并没有花费大量的资源或人力在DevOps上，相反我们主要关注产品的构建。在刚开始几年的大部分时间里，我把自己定位为一软件工程师，每月只需几天时间花费在系统管理上。

我们经常开玩笑说，我是那个抽到下签，被“困于”服务器管理的那个人。但是，实际上，我自己特别享受。尽管如此，我也从来没想过我对软件开发的热情居然会用在服务器管理上。

尽量简单（在你能控制的范围）

我们初期之所以能不在DevOps上花费力气是有几个原因的。首先，很大程度上是因为我们是小公司，变化也比较少。我们选择了一个简单的架构，应用程序数量也比较少。所以尽管我们的产品不断发展，但是需要对底层的基础架构进行修改的地方却比较少。

过度投资

第二，早期我们在服务器硬件上投资过度了。我们可能只需要两台服务器就可以运行我们整个网站，而我们却买了10台。但这允许我们之后花很少的时间去担心那些性能或基础架构发展等问题，而我们也是在几年之后才耗尽初期购买的服务器的性能。当然，配置这些服务器是有前期成本的，但是一旦设置好了，就不需要大的改动。

选择好的工具

最后，我们采用Ruby on Rails作为应用程序框架，及其关联的工具，比如：Git、Capistrano和Teamcity，允许我们在早期就采用那些良好的发布和部署实践。与此同时，我们也构建于那些行之有效且稳定的开源解决方案之上，比如：ng

inx、MySQL和memcached。

也正因为对这些工具和框架的采用，从而让我们避免了采用不必要的复杂且专有的解决方案。很多时候我觉得这些方案会随着公司的成长，降低其开发速度。

成长

随着我们公司进入第二个成长阶段，我们的工程师和产品团队也稳步增长。只有两个开发人员的日子已经一去不复返了，一眨眼间，我们就有10个，甚至20个人为现有的以及新产品工作。可想而知，我们需要引进的变化数量也会稳步增加。我们的硬件上需要承载的应用绝对数量也直线上升，先是以前的两倍，然后三倍。同时开发人员也想使用新的应用框架、编程语言、数据库、排队系统及缓存服务器等。

随着这些复杂性的增加，错误和停机的成本也增加了。很快我们发现旧的手动配置基础架构已不再适用，而我们基础架构层对成熟性和灵活性的缺乏也将给我们带来以下问题：减缓新产品和功能的发布，危害我们的稳定性。认识到这一点，我停止了在产品上的工作，并将重点全部放在研究开发自动化、监控和发布流程上。

引进Chef

幸运的是，我们认识到我们需要实行DevOps实践。而当时，我们遇到了OpsCode Chef，以及它所倡导的“基础架构即代码”原则。最初，我们花费了几个月时间为现有基础框架的每一部分编写自动化脚本。一旦完成，我们就可以利用这些自动化脚本重构所有服务器，这将大大减轻我们肩上的负担。现在我们所有服务器都已设置一致，而我们最终也有一个地方能让团队所有人都可以看到每块基础架构到底是如何建立和设置的。同样重要的是，它允许我们快速地将额外资源添加进来。

DevOps团队的成立

DevOps开始在公司内担负起独特的责任，为我们的产品线提供关键支持，并保证我们的基础架构可以持续拓展。除了这些重点之外，我们也在不断开发工具和产品，用于管理这些需要持续支持和改进的基础架构。尤其在前期，通常来说如果往基础架构中引进新的应用，就要求对底层自动化脚本进行修改。由于越来越多人开始依赖我们的工作，内部用户（开发人员和运维人员）也随之提出了更多与DevOps相关的需求。

由于我们的开发团队已经分为不同的产品团队，且每个团队都关注不同的业务，我们决定设定一个单独的DevOps团队。这样我们就有一个人专门的团队全职解决

基础架构问题。另外，应用平台的可用性和稳定性对我们的业务来说也是及其重要的，宕机或别的问题对我们的服务都会有很大的影响。我们需要随时保证有专门的、训练有素的工程师来协助和调查问题，尤其当问题的归属可能跨越多个团队或无法当时就明了时。

自动化果实

按需配置

在采用Chef和自动化我们产品的基础架构之后，我们立马着手做的事情就是改善我们的测试和staging系统，给它们配上同等程度的自动化。我们经常听到的抱怨之一就是开发人员无法简单快速地给业务人员展示他们当前所做的工作。为此，我们开发了一个100%自助服务的门户，这样公司里的任何人都可以启动一个预配置服务器在EC2上运行完整的程序。

New Server My Servers All Servers Install CA Credentials

Create a New Staging Server

Availability Zone: us-west-1b (EC2) Domain: staging Name: my-server

Instance Type: m1.large RAM: 7.5GB CPU: 2x2GHz IO Performance: High Hourly Cost \$0.38

Database Location: Local Remote Database Engine: MySQL Clustrix Database Type: Development Production Snapshot: Fri May 17 13:58:40 -0700 2013 Default Branch: master This branch will be used for any apps that don't have a branch specified.

Automatically Redeploy:

Rails Applications:

<input checked="" type="checkbox"/> Mercury	<input checked="" type="checkbox"/> Venus	<input type="checkbox"/> Earth
my_feature_branch	optional branch	Mars
		Jupiter
<input type="checkbox"/> Saturn	<input type="checkbox"/> Uranus	<input type="checkbox"/> Neptune
<input type="checkbox"/> Ceres	<input type="checkbox"/> Pluto	<input type="checkbox"/> Haumea
<input type="checkbox"/> Eris	<input type="checkbox"/> Makemake	<input type="checkbox"/> Phobos
<input type="checkbox"/> Deimos	<input type="checkbox"/> Eros	<input type="checkbox"/> Gaspra
<input type="checkbox"/> Ida		

Clojure Applications:

<input type="checkbox"/> Elara	<input checked="" type="checkbox"/> Pasiphae	<input type="checkbox"/> Sinope
	optional branch	

Backend Applications:

<input type="checkbox"/> Common Libs	<input type="checkbox"/> Inventory	<input type="checkbox"/> Order Processing
<input type="checkbox"/> Ananke	<input type="checkbox"/> Adrastea	<input type="checkbox"/> Aitne
<input type="checkbox"/> Carme	<input type="checkbox"/> Autonoe	<input type="checkbox"/> Euanthe
<input type="checkbox"/> Leda	<input type="checkbox"/> Callirhoe	<input type="checkbox"/> Euporie
<input checked="" type="checkbox"/> Lysithea	<input type="checkbox"/> Chaidene	<input type="checkbox"/> Eurydome
	optional branch	<input type="checkbox"/> Hegemone
<input checked="" type="checkbox"/> Thebe	<input type="checkbox"/> Harpalyke	<input type="checkbox"/> Hermippe
	optional branch	<input type="checkbox"/> Kale
	<input type="checkbox"/> Iocaste	<input type="checkbox"/> Orthosie
	<input type="checkbox"/> Isonoe	<input type="checkbox"/> Pasithee
	<input type="checkbox"/> Kalyke	<input type="checkbox"/> Sponde
	<input type="checkbox"/> Megalite	
	<input type="checkbox"/> Metis	
	<input type="checkbox"/> Praxidike	
	<input type="checkbox"/> Taygete	
	<input type="checkbox"/> Themisto	
	<input type="checkbox"/> Thyone	

Additional Options

Send all mail to: email@address.com Multiple emails can be separated with commas. If left blank, the server will not deliver any mail

VPN Access:

Termination Reminders: Email me if my server has been running for more than 2 weeks without any new activity

Sleep Off Hours: Put my server to sleep at 8 pm and wakeup at 8 am (GMT-08:00) Pacific Time (US & Canada) and do NOT wakeup on weekends

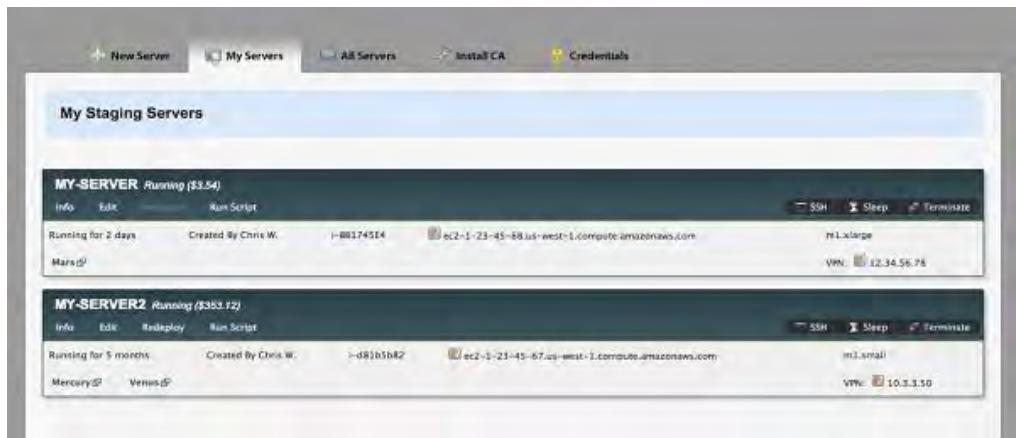
Terminate after: 2013-05-15 Automatically terminate my server after this time.

[Advanced Settings](#)

[Create](#)

用户可以自己选择要在服务器上安装哪个版本的哪些应用程序，他们也可以选择使用测试数据库或使用产品数据库某一时间的快照。该系统的最大优点就是可以重用我们构建产品服务器时的Chef脚本。这允许我们在发布到产品服务器之前，就把很多潜在问题给过滤掉了。我们的开发和QA团队也会更自信，因为他们测试新特性的环境与产品环境上的设置及运行的版本是一致的。

现在在我们公司，此Staging系统尤其受欢迎。很多员工都说到他们之前所在的公司如果想要搭建一个模拟服务器，需要数周的时间及大量的文本工作。在我们这里，所有人都可以在15分钟内构建出一个Staging系统。当然创建这样的系统肯定离不开健壮的自动化基础。



数据中心故障恢复只需几分钟

在Rafter，数据中心故障恢复是DevOps自动化帮助我们解决的另外一个难题。大概在两年前，我们决定我们需要有能力在任意时候都能切换到辅助数据中心，以减少停机所带来的影响和损失。由于我们面向的是教育行业，我们的业务有季节性，如果在返校季遭受长时间的停机，其代价是昂贵的。同时，我们的运维人员也从中获利极大，他们可以在数据中心无实时数据时，执行那些具有风险的维护。

正因为有了自动化的基础架构，我们可以相对比较容易地完成这个难题。要换在最初，如果有人告诉我有天我们能在几分钟内对整个数据中心进行故障恢复，我肯定会非常震惊。但是现在我们完全可以依靠自动化，设计出相应的基础架构来支持这一目标。

共享部署

另外一块需要我们DevOps团队重点工作的地方是改进部署流程。从一开始，我们就使用Capistrano这一优秀工具管理部署，并对其非常满意。我们需要做出的改进就是让Campfire bot（又名为Reginald）通过Capistrano部署。现在我们的产品开发人员只要通过Reginald来发布应用就可以了。

The screenshot shows a Campfire 'Ship It' chat room. On the left, a log of messages is displayed:

- 1:40 AM: Cameron B. has entered the room
- 1:40 AM: Cameron B. has left the room
- Chris W. reg deploy apps:awesome-app
- Reginald N. Deploying awesome-app: id: 2746. Say 'reg deploy:status' to see status of deploys or go to <https://deployments/2746>
- Deploy Command 2746: awesome-app DEPLOYED SUCCESSFULLY
- Chris W. reg compliment Reginald
- 1:45 AM: Reginald N. Reginald, you are a true marvel.

On the right, the 'Ship It' interface shows:

- Who's here?**: Chris Williams, Reginald Netsky
- Guest access**: off (Turn it on)
- Conference call**: Start a new call
- Upload a file**: (all)

其实这里最大的胜利就是让部署变成一个共享的经验。Campfire聊天室的任何人都可以查看部署是否正在进行，如果有问题，马上就可以有人参与解决。所有的错误和部署日志都存储于我们的数据库中，并且可以从Reginald给开发人员指定的web应用上查看。这样一来，就更容易分享那些潜在的问题。在以前，当开发人员在服务器上部署时更多的是私人操作，而现在我们采用公开部署，让团队中每个人都能看到具体情况，这样部署起来就更简单。

自助工具

对于我们DevOps团队构建并使用的产品，我们都秉持着尽可能使其可自助化这一宗旨。自动化的成功在于移除人为的障碍，尤其是来自我们自己的。为相应人员提供工具和平台用于管理他们在乎的那部分基础架构，会让整个组织更有效地执行起来。尤其是在那些别人可以为你工作的领域（有可能他们会比你做得更好），你的工具应该允许他们去这么做。在我们采用的工具和实践中，我们也保持这个原则。

我们的团队构成

我发现，[行业里很大范围内趋向于将DevOps结构定义为：开发vs运维](#)。我们的DevOps团队出自于产品开发团队，所以一直由传统软件开发人员组成。我们所有现有团队成员在成为DevOps的一员前，都为我们主要产品做过开发。我们非常幸运拥有这样一个出色的运维团队，照顾和改善我们的硬件和数据中心，允许我们全心全意地关注于软件开发。

关注开发

我相信对“开发”这一层面的极大关注对我们来说非常行之有效。随着我们需要构建和支持更加复杂的应用基础架构，我们对传统软件开发的需要将不断增加。这样说来，我们发现招聘工作越来越难，因为很多软件工程师对运维/基础架构相关领域并没有太多兴趣，而很多运维工程人员也对传统软件开发的各个角色没有太多经验。可以肯定的是，我们需要特殊人员来填补该空缺。在Rafter，我注意

到DevOps似乎引起那些对某一领域有很深造诣的人员，而非那些通才。现在我们的DevOps团队包含有三个工程师，同时我们也一直在寻找更多的优秀人才。

使我们DevOps团队独特的一点是我们经常会接受那些可能需要正常产品团队或网站稳定性工程师来完成的任务。比如：我们领导完成过将我们的应用升级到能在Ruby及Ruby on Rails各主要新版本上运行的项目。如果你曾参与过大型应用针对重要的Rails更新的话，你就会明白这并非易事，它需要对Rails及底层代码库有很深的了解。同时，我们也经常帮助产品团队调试和解决他们产品上的性能及可拓展性问题。正因为这些工作，我们团队不断地接触那些我们需要支持的软件平台的各部分。

日常工作

支持

我们DevOps团队成员典型的一天是由以下几部分组成：解决支持请求、调查告警和从事长期项目。通常，我们会花费大概50%的时间在支持和告警上，50%的时间在项目上，但是支持请求和告警会占更多，因为一般情况下它们都具有时间紧迫性。其中支持请求往往涉及各种各样的问题，但主要还是围绕着应用支持相关领域。举个例子，工程师希望往平台上添加一个新的应用，又或想给现有应用拓展更多服务器。有时，这些请求可以是大范围的，需要很多修改和测试的，比如：支持一种新的编程语言或新的数据库。

监控和告警

另外一组常见请求是调查应用告警（有自动的，也有工程师报告出来的）。通常，DevOps扮演警报协调员的角色，进行着初步调查，并寻找出解决问题的最佳团队。这样说来，我们并不会亲自处理所有的应用告警。我们会调查那些影响基础架构的告警（如涉及对网络、IO、CPU和内存过度使用的情况），或那些由各团队共享的职责和使用的领域（如共享的库、数据库和遗留应用程序）。

项目工作

很多情况下，我们在工程支持请求和调查告警上所做的工作，会帮助我们发现那些我们需要进一步改进的地方。具体说来，这些领域里，我们往往没有为工程师提供足够的工具或信息去解决问题。我们现在就面临着这样一个问题：我们还没有能力让工程师自己部署自己的cronjob改动。因此，当需要为应用修改cronjob时，他们就需要开一个支持请求。当每周有10个这样请求的时候，很显然我们需要为工程师们开发出一工具以允许他们部署自己的cronjob。因此，我们的项目工作很大一部分来自处理支持和告警请求时发现的缺陷。

我们还花费大量时间在系统更新，及为提升对更新的信心所做的测试上。我需要花上好几页才能列全我们平台上所支持的服务器、数据库、应用程序、库及操作系统。保证所有东西都以其最新最安全的版本运行着，其本身就够多位工程师忙碌上一整年。

我们另外一个项目工作源自直接的外部请求。例如：管理层要求我们能够故障恢复到另外一个数据中心，或审计上要求一个特定安全功能，又或是个由工程师发出的可能影响多个应用的基础架构修改请求。

DevOps即平台

随着公司的不断壮大，DevOps也在我们的产品工程团队和运营团队间找到了自己的位置。我把这个结构设想成三层蛋糕：底层是我们的运维团队，负责获取和建立物理硬件；然后中间是我们的DevOps团队，负责提供一个平台，以便大家使用这些硬件资源；最上面一层是我们的产品工程团队，他们使用DevOps平台部署、监控和管理他们的应用。

当讨论DevOps时，我认为“平台”是个关键概念。当我第一次开始DevOps时候，我曾认为我们的工具是些独立的项目。但是，一旦你开始构建这些独立部分，你会意识到各个组件都能构成更大平台的一部分，并且它们都要接触到共用的信息集。所有的工具都需要集成起来，否则其复杂度和重复性就会慢慢增加。

构建DevOps平台

将我们的DevOps工具转变成服务平台是当前也是未来需要不断进行的一个工程。这听起来，可能会很奇怪。但是你想想，我们会有不同的客户想要了解我们的基础架构信息。比如：本文提到的某些应用可能需要通过以下方式访问DevOps平台：

- 我们的Staging服务器门户需要知道哪些应用可以安装在它上面。
- 我们的部署框架需要知道哪些应用可以部署，及部署在哪里。
- 为了切换数据中心，我们需要知道如何在另外的数据中心启动这些应用。
- Chef脚本需要知道如何根据我们要安装的应用要求，正确配置服务器。

从这些例子中可以看出，通过服务发布那些通用的信息集可以用来回答关于基础架构的基本问题。

我们可以设想得更远，开发出更多影响基础架构中变化的服务。比如：让服务器或应用进入维护模式的服务；往平台中添加新应用服务等等。有的DevOps组织可能会通过特定脚本来执行这些功能。但是它的缺点在于一般情况下只能有一个DevOps工程师执行这类脚本。服务的好处在于其它工具和应用可以通过共享方式与其交互。为了更大的灵活性，我们也可以在这些服务上面开发出相应的应用

。这也支持了“任何时候都可以构建能自我服务的工具”这一想法。

Chef即数据库

我们看待Chef在整个系统中的角色也发生了转变。它不仅仅是个自动化服务器的工具。我们把它当作DevOps平台中的数据层。通过引用Chef平台和APIs的优点，我们可以保存和查询所有关于基础架构里的服务器和应用的信息。在Chef提供数据存储层的同时，我们也在建立自己的服务集以访问这些数据和提供灵活的交互方式。

我们的工作永远不会结束

所有的这些都是为了确保我们依然可以持续拓展，且基础架构不会成为新产品发布的障碍。就算有大量的自动化就绪，我们仍然需要在现有自动化基础上进行持续迭代。

有个例子很快就呈现在我脑海里。我们的团队曾花费很多精力构建一个如何在每个服务器上设置应用的框架。以前我们只需要编写一个小的JSON文件，描述下应用及依赖的基本信息。然后就可以拍拍手去解决下一个问题了。

但是我们开始意识到我们需要添加很多新应用，并要花费很大一块时间编写这些配置，因为通常需要与产品工程师多次来回沟通，其效率比较低。很快成为自动化的一个瓶颈。因此自动化工作永远不会结束，就算你今天有足够的自动化，很可能明天它就会阻碍你的发展。

关于作者



Chris Williams是[BookRenter.com](#)，首家在线教科书租赁服务提供商，是于2012年成立的Rafter Inc联合创始人之一。现在他在管理Rafter的DevOps团队，并负责其基础架构自动化、部署和发布流程、及平台可用性。

原文链接：<http://www.infoq.com/cn/articles/monthly-devops-04-r-after>

相关内容

- [书评：DevOps for Developers](#)
- [全面实现自动化！Windows Azure添加对DevOps工具Puppet的支持](#)
- [天生一对：云与DevOps](#)
- [定义DevOps 2.0：统一工具 + 环境整合？](#)
- [诺基亚娱乐部门如何用DevOps补敏捷之不足](#)

QClub

我们影响有影响力的人

非盈利、非商业、纯技术交流的技术社区活动

We are QClub

QClub作为InfoQ线下技术沙龙品牌，定期在全国主要城市举办免费的技术沙龙，邀请国内外知名公司技术总监，项目经理，高级研发工程师等走入各地技术社区，分享他们的经验与对行业趋势的预测与讨论，为中国技术人员搭建交流、分享的平台，尽自己微薄之力架起中高端技术人员之间的桥梁，为中国技术社区的发展与价值的传播贡献自己的力量。

非盈利 非商业 纯技术交流

QClub的话题

QClub的话题可能包括任何当地技术人员感兴趣的话题，比如：编程语言、架构设计、企业级开发、运维、基础架构、过程与实践、云计算、大数据、互联网、移动开发、安全、敏捷、性能、业务流程、SOA、.....



参会人群：

开发人员、技术或
团队负责人、技术
爱好者、学生等

活动城市：

北京、长沙、成都、大连、福
州、广州、上海、太原、天津、
温州、西安、郑州、.....

举办周期：

每个城市
每1-2个月一次活动

活动形式多样：

- 讲师演讲 ◦ 线上交流
- Open Space ◦ 粉丝 QQ 群
- 户外活动 ◦ 等等.....

QClub专栏

QClub&Global Code Retreat全球编程日见闻与有感

作者 fatbigbright

上周六，QClub同时组织了天津、上海、太原三地的Global Code Retreat 活动，感谢三地朋友的热情参与。一下是天津参与者@fatbigbright 的参会感受，展出和大家一起感受一下当天的活动。qclubcrtj

上周六参与了QClub Global Code Retreat全球编程日的天津站活动，这里整理一下当天的一些经过和自己的一些感受。

先给提供场地的[@天津纳吧创业咖啡](#)免费打个广告，环境清悠，设备齐全，赞一下。

本次的活动主要是通过反复结对编程实现[生命游戏](#)（链接内有详细描述），来进行敏捷开发中测试驱动等方面的练习。

有意思的是，这个看似简单的问题，仅仅是对它的理解，所有的参会者们直到第二轮结对结束的午饭时间之前，才消除了彼此的重大分歧。可以说上午的两次迭代几乎只是脑力的一次热身。大家到下午才开始进入状态。

经历了前两次迭代熟悉问题之后，下午的迭代开始正式加入了测试驱动，对于如何写出可测试的程序，教练[@申导](#)做的演示给我了很大的启发。之前的迭代中，我一直试图建立一个初始状态随机的细胞类的矩阵，而这个矩阵是一个操纵它的工作类的内部成员，这个思路当然没有问题，但随机的初始状态由于作为工作类的内部状态，其不可预测性使得单元测试成为不可能。教练的Demo则是将一个随机产生初始状态的功能与工作类剥离，使初始状态成为工作类的外部输入，这样有了可预测试的输入，才能进行可预测的单元测试。

本次活动中我使用了C与C#这两种开发语言开发控制台程序进行问题的模拟，但没有图像的直观感受始终不尽如意，于是活动之后，我又使用html5+js实现了一个[在线可视化版本](#)，并且添加了图形化的单元测试。

可视化版本的开发中，因为js二维数组深拷贝的问题，一度没能得到正确的结果，后来在借助万能的互联网才发现slice()函数的局限性。还有对this对象引用的作用域，也有了一些新的理解。

这个可视化初步可测试的版本直到今天早上才开发结束，也不由感叹自己驾驭代

码的能力仍需提高啊。

有趣的小细节：

- 生命游戏是对初始状态极为敏感的，一个细胞生死状态的扰动，可能使得全局成为一个循环平衡态或是一个衡定态，又或者最后全部死亡。
- 假如死细胞的复活条件从周围有正好3个活细胞，改成周围有3个及以上的活细胞，就会发现细胞会像一个菌落一样越发壮大，最后充满全局，躁动不已，像是老式模拟信号的电视的白噪点。感谢本文作者@fatbigbright 的翔实记录，原文链接：<http://fatbigbright.github.io/2013/12/16/code-retreat-game-of-life/>

推荐文章 | Article

重构遗留程序的一次案例学习

作者 [Chen Ping](#)，译者 [邵思华](#)

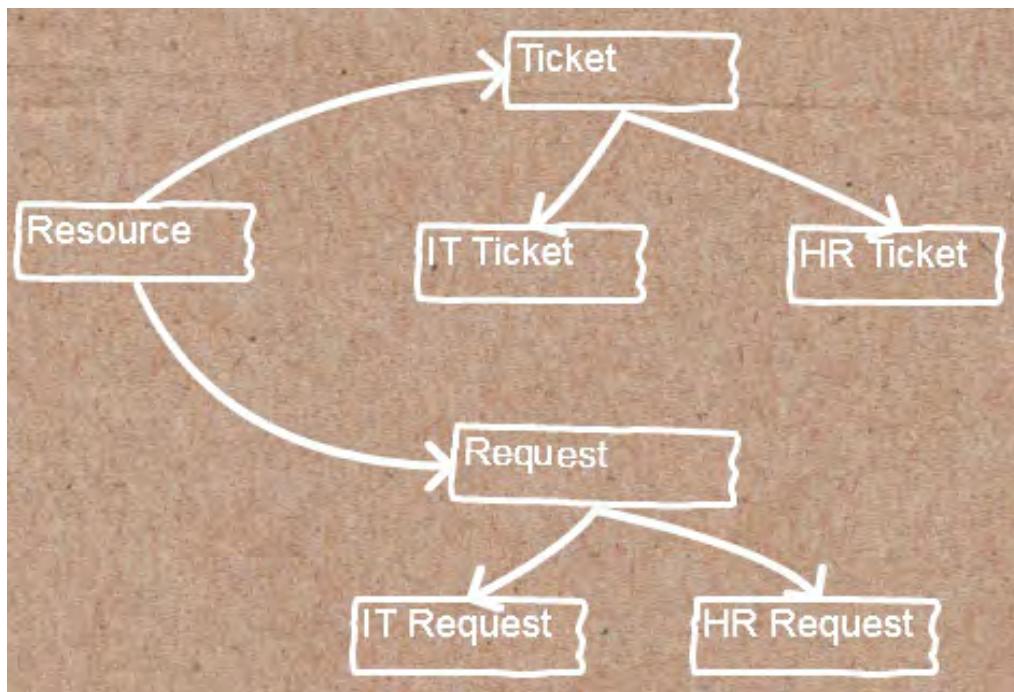
遗留代码经常是腐臭的，每个优秀的开发者都想把它重构。而进行重构的一个理想的先决条件是，它应该包含一组单元测试用例，以避免产生回归缺陷。但是为遗留代码编写单元测试可不是件容易的事，因为它经常是一团糟。要想为遗留代码编写有效的单元测试，你大概得先把它重构一下。但要重构它，你又需要单元测试来确保你没有破坏任何功能。这种状况相当于要回答是先有鸡还是先有蛋。这篇文章通过分享一个我曾参与过的真實案例，描述了一种可以安全地重构遗留代码的方法。

问题描述

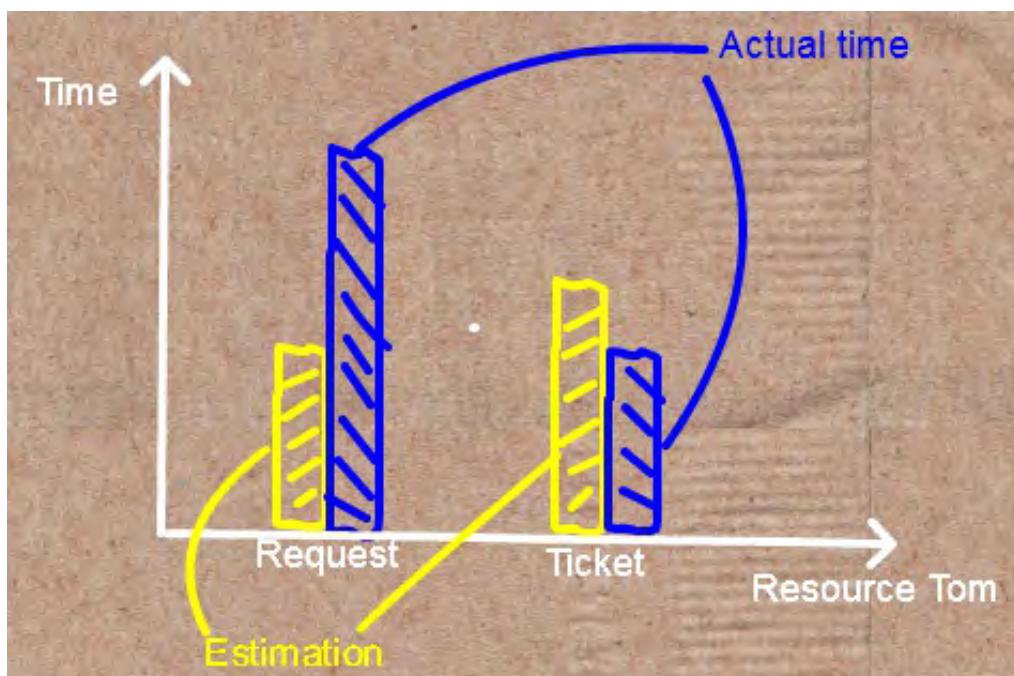
在这篇文章中，我将用一个真实案例来描述测试与重构遗留系统的有效实践。这个例子的代码由Java编写，不过这个实践对其他语言也是适用的。我将原始场景稍做了些改动以免误伤无辜，并稍做简化以便让它更容易被理解。这篇文章所介绍的实践帮助我重构了近期我所参与的一个遗留系统。

这篇文章并不打算介绍单元测试与重构的基本技巧。你可以通过阅读相关书籍以学习该主题的更多内容，如[Martin Fowler](#)的[《重构：改善既有代码的设计》](#)及[Joshua Kerievsky](#)的[《重构与模式》](#)。相对而言，这篇文章的内容将描述一些真实际景中的复杂性，我也希望它能够为解决这些复杂性提供一些有用的实践。

在这个案例中我将描述一个虚构的资源管理系统，其中资源指的是可指派给其任务的某个人。可以为某个资源指派一个HR票据（ticket）或者IT票据，也可以为某个资源指派一个HR请求或IT请求。资源经理可以记录某个资源处理某项任务的预计时间，而资源本身可以记录他们在某个票据或请求上工作的实际时间。



可以用饼图的方式表示资源的使用情况，图中同时显示了预计时间与实际花费的时间。



好像不太复杂嘛？不过，真实的系统能够为资源分配多种类型的任务，当然从技术上讲这也不是多么复杂的设计。但当我初次看到系统的代码时，我感觉自己似乎看到了一件老古董，从中看得出代码是如何从开始逐步进化的（或者不如说是退化的）。在一开始，这一系统仅能用来处理请求，之后才加入了处理票据以及其它类型任务的功能。某位工程师开始编写代码以处理请求：首先从数据库中获取数据，随后按照饼图的方式显示数据。他甚至没有考虑过要将信息组织为合适的对象：

```

class ResourceBreakdownService {
    public Map search (Session context) throws SearchException{

        //omitted twenty or so lines of code to pull search criteria ou
t of context
        and verify them, such as the below:
        if(resourceIds==null || resourceIds.size ()==0){
            throw new SearchException("Resource list is not provided");
        }
        if(resourceId!=null || resourceIds.size()>0){
            resourceObjs=resourceDAO.getResourceByIds(resourceIds)
        ;
        }

        //get workload for all requests

        Map requestBreakDown=getResourceRequestsLoadBreakdown (resource
Objs,startDate,
finishDate);
        return requestBreakDown;
    }
}

```

我相信你肯定被这段代码里的坏味道吓到了吧？比方说，你大概很快就会发现**search**并不是一个有意义的名称，还有应该使用Apache Commons类库中的**CollectionUtils.isEmpty()**方法来检测一个集合，此外你大概也会疑惑该方法返回的Map对象到底包含了些什么？

别着急，坏味道陆续有来。接下来的一位工程师继承了先人的衣钵，按照相同的方式对票据进行了处理，以下就是修改后的代码：

```

// get workload for all tickets

Map ticketBreakdown =getResourceRequestsLoadBreakdown(resourceObjs,star
tDate,
finishDate,ticketSeverity);
Map result=new HashMap();
for(Iterator i = resourceObjs.iterator(); i.hasNext();) {
    Resource resource=(Resource)i.next();
    Map requestBreakdown2=(Map)requestBreakdown.get(resource);
    List ticketBreakdown2=(List)ticketBreakdown.get(resource);
    Map resourceWorkloadBreakdown=combineRequestAndTicket(requestBreakd
own2,
ticketBreakdown2);
    result.put(resource,resourceWorkloadBreakdown)
}
return result;

```

先不管那糟糕的命名、失衡的代码结构以及其它任何代码美观度上的问题了。这段代码中最坏的味道就是它返回的**Map**对象了，这个**Map**对象完全是个黑洞，里面塞满了各种数据，但又不会提示你里面究竟包含的是什么。我只能编写了一些调试代码，将**Map**中的内容循环打印出来后，才看懂了它的数据结构。

在这个示例中，`{}` 代表一个**Map**，`=>` 代表键值映射，而`[]` 代表一个集合：

```

{resource with id 30000=> [
    SummaryOfActualWorkloadForRequestType,
    SummaryOfEstimatedWorkloadForRequestType,
    {30240=>[
        ActualWorkloadForRequestWithId_30240,
        EstimatedWorkloadForRequestWithId_30240],}

    30241=>[
        ActualWorkloadForRequestWithId_30241,
        EstimatedWorkloadForRequestWithId_30241]
    }
    SummaryOfActualWorkloadForTicketType,
    SummaryOfEstimatedWorkloadForTicketType,
    {20000=>[
        ActualWorkloadForTicketWithId_2000,
        EstimatedWorkloadForTicketWithId_2000],
    }
]
}

```

这个糟糕的数据结构使得数据的编码与解码逻辑在直观上非常冗长乏味，可读性很差。

集成测试

希望我已经让你认识到这段代码确实是非常复杂的。如果在开始重构之前让我首先解开这团乱麻，然后理解每一行代码的意图，那我非疯了不可。为了保持我的心智健全，我决定采用一种自顶向下的方式来理解代码逻辑。也就是说，我决定首先尝试一下这个系统的功能，进行一些调试，以了解系统的整体情况，而不是一上来就直接阅读代码，并试图从中推断出代码的逻辑。

我所使用的方法与编写测试代码完全相同，传统的方法是编写小段的测试代码以验证每一段代码路径，如果每一段测试都通过，那么当所有的代码路径组织在一起之后，方法能够按照预期方式工作的机会就很高了。但这种传统方式在这里行不通，**ResourceBreakdownService**简直就是一个[“上帝类”](#)，如果我仅凭着对系统整体情况的一些了解就对这个类进行分解，很可能会造成很多问题 - 在遗留系统的每个角落里都有可能隐藏着众多不为人知的秘密。

我编写了以下这个简单的测试，它反映了我对整个系统的理解：

```
public void testResourceBreakdown(){
    Resource resource=createResource();
    List requests=createRequests();
    assignRequestToResource(resource, requests);
    List tickets=createTickets();
    assignTicketToResource(resource, tickets);
    Map result=new ResourceBreakdownService().search(resource);
    verifyResult(result,resource,requests,tickets);
}
```

注意一下**verifyResult()**这个方法，我首先循环式地将result的内容打印出来，以找出其中的结构，随后**verifyResult()**方法根据这个结构对结果进行验证，确保其中包含了正确的数据：

```
private void verifyResult(Map result, Resource rsc, List<Request> requests,
List<Ticket> tickets){
    assertTrue(result.containsKey(rsc.getId()));

    // in this simple test case, actual workload is empty
    UtilizationBean emptyActualLoad=createDummyWorkload();
    List resourceWorkLoad=result.get(rsc.getId());

    UtilizationBean scheduleWorkload=calculateWorkload(rsc, requests);
    assertEquals(emptyActualLoad, resourceWorkLoad.get(0));
    assertEquals(scheduleWorkload, resourceWorkLoad.get(1));

    Map requestDetailWorkload = (Map)resourceWorkLoad.get(3);
    for (Request request : requests) {
        assertTrue(requestDetailWorkload.containsKey(request.getId()));
        UtilizationBean scheduleWorkload0=calculateWorkload(rsc, request);
        assertEquals(emptyActualLoad, requestDetailWorkload.get(request.getId()).get(0));
        assertEquals(scheduleWorkload0, requestDetailWorkload.get(request.getId()).get(1));
    }

    // omit code to check tickets
    ...
}
```

用临时方案绕过障碍

以上测试用例看起来简单，但实际却隐含了许多复杂性。首先，**ResourceBreakdownService().search**方法与运行时紧密相关，它需要访问数据库、其它服务，或许还有些不为人知的依赖项。而且和许多遗留系统一样，这个系统也没有建立任何单元测试的架构。为了访问运行时服务，唯一的选择就是启动整个系统，这不仅造成巨大的开销，而且也带来了很大的不便。

ServerMain类启动了整个系统的服务端功能，这个类也是个老古董了，你完全可以从中观察到它的进化过程。这个系统的编写时间已经超过10年了，当时还没有**Spring**、**Hibernate**这些东西，**JBoss**和**Tomcat**也才刚刚冒头。因此那些勇敢的先驱们不得不手工打造了许多工具，他们创建了一个自制的集群、一个缓存服务、一个连接池以及其它许多东西。之后他们在某种程度上引入了JBoss和Tomcat（但不幸的是他们仍然保留了那些手工艺品，导致了现在的代码中存在着两种事务管理机制以及三种连接池）。

我决定将**ServerMain**复制到**TestServerMain**类中，但运行**TestServerMain.main()**方法产生了以下失败信息：

```
org.springframework.beans.factory.BeanInitializationException: Could not load properties; nested exception is
java.io.FileNotFoundException: class path resource [database.properties] cannot
be opened because it does not exist
at
org.springframework.beans.factory.config.PropertyResourceConfigurer.
postProcessBeanFactory(PropertyResourceConfigurer.java:78)
```

好吧，它还挺灵活！我随意拿了个**database.properties**文件，把它放到测试类的文件夹中并再次运行测试。但这一次程序又抛出了下面的异常：

```
java.io.FileNotFoundException: .\server.conf (The system cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:106)
at java.io.FileInputStream.<init>(FileInputStream.java:66)
at java.io.FileReader.<init>(FileReader.java:41)
at com.foo.bar.config.ServerConfigAgent.parseFile(ServerConfigAgent.java:1593)
at com.foo.bar.config.ServerConfigAgent.parseConfigFile(ServerConfigAgent.java:1720)
at com.foo.bar.config.ServerConfigAgent.parseConfigFile(ServerConfigAgent.java:1712)
at com.foo.bar.config.ServerConfigAgent.readServerConf(ServerConfigAgent.java:1581)
```

```

    at com.foo.bar.ServerConfigFactory.initServerConfig(ServerConfigFac
tory.java:38)
    at com.foo.bar.util.HibernateUtil.setupDatabaseProperties(Hibernate
Util.java:207)
    at com.foo.bar.util.HibernateUtil.doStart(HibernateUtil.java:135)
    at com.foo.bar.util.HibernateUtil.<clinit>(HibernateUtil.java:125)

```

看起来只要找到server.conf文件就可以了，但这种方式让我有些不爽。仅仅才编写了一个简单的测试用例，就暴露出了代码中的一个问题。正如**HibernateUtil**的名字所建议的，它仅仅关心数据库信息，而这些信息应该都由**database.properties**文件提供，为什么还需要访问用以配置服务端启动信息的**server.conf**文件呢？这里似乎暗示代码散发出坏味道了：当你感觉到自己如同在读一本侦探小说，不断地问“为什么”的时候，这就意味着代码是糟糕的。如果我再多花一些时间完整地看一下**ServerConfigFactory**、**HibernateUtil**和**ServerConfigAgent**这些类，大概能够找到让**HibernateUtil**直接使用**database.properties**的方法吧。但那个时候的我已经心烦意乱了，始终没法让程序启动。顺便说一句，这里有一种处理它的临时方案，这把武器就是[AspectJ](#)：

```

void around():
    call(public static void com.foo.bar.ServerConfigFactory.initServerC
onfig()){
    System.out.println("bypassing com.foo.bar.ServerConfigFactory.inits
erverConfig");
}

```

让我用直白一点的方式为不了解AspectJ的读者介绍一下以上代码的含义吧：当运行时准备调用**ServerConfigFactory.initServerConfig()**方法，让它打印出一条信息，然后跳过该方法的执行并直接返回。听起来好像是某种hack，但它大大降低了开销。遗留系统中充斥着问题与谜团，与其打交道的每一时刻都得采取些策略。眼下，从客户满意度这点看来，对我来说最有意义的事情就是修复这个资源管理系统中的缺陷，并改善它的性能。其它方面的代码整理并不是我的当前目标，但我已记住了这个问题，我决定之后再回来处理**ServerMain**中的问题。

接下来，在每一处**HibernateUtil**需要读取**server.conf**文件的地方，我都强制让它从**database.properties**中进行读取。

```

String around():call(public String com.foo.bar.config.ServerConfig.getJ
DBCUrl()){
    // code omitted, reading from database.properties
}

```

```

String around():call(public String com.foo.bar.config.ServerConfig.getDBUser()){
    // code omitted, reading from database.properties
}

```

接下来的工作你大概能猜到了，如果使用临时方案会比较方便又显得自然，那就使用它。而如果有现成的mock对象可以使用，那么就重用它们。举例来说，**TestServerMain.main()**方法在某一时刻会产生如下错误：

```

- Factory name: java:comp/env/hibernate/SessionFactory
- JNDI InitialContext properties:{}
- Could not bind factory to JNDI
javax.naming.NoInitialContextException: Need to specify class name in environment
or system property, or as an applet
parameter, or in an application resource file: java.naming.factory.initial
    at javax.naming.spi.NamingManager.getInitialContext(NamingManager.java:645)
    at javax.naming.InitialContext.getDefaultInitCtx(InitialContext.java:288)

```

这是由于JBoss命名服务未启动造成的，虽然我也可以使用相同的hack技术作为临时方案，但**InitialContext**是一个庞大的Java接口，它包含了数量众多的方法，我可不想把每个方法都用hack方式给补完，那实在太冗长了。我很快发现Spring里已经包含了一个mock的**SimpleNamingContext**类了，那么就把它放到测试里去试试：

```

SimpleNamingContextBuilder builder = new SimpleNamingContextBuilder();
builder.bind("java:comp/env/hibernate/SessionFactory", sessionFactory);
builder.activate();

```

经过几次反复的修改后，我终于能够成功地运行**TestServerMain.main()**方法了。它比起**ServerMain**来说要简单许多，不仅mock了许多JBoss的服务，而且完全避免了集群管理的麻烦。

创建构造块

TestServerMain连接了某个真实的数据库，而遗留系统往往会在存储过程、甚至是触发器中隐藏了各种出人意料的逻辑。基于对系统整体情况的考虑，我认为在当前状况下试图理解数据库中的所有奥秘、并以此创建一个伪造的数据库是一个不明智的选择，因此我决定仍然在测试用例中访问真实的数据库。

这些测试用例必须保证它们能够重复运行，以确保我对产品代码所做的任何小改动都能够通过测试。每一次运行，测试用例都会在数据库中创建资源与请求。与单元测试的习惯作法不同的是，有时你并不希望在每次运行之后对测试用例所创建的各种数据进行清理。我们目前为止所做的测试与重构练习更像是一次实地考查的探索——通过对遗留系统进行测试的方式来学习它的功能。为了确保一切功能都像预期一样工作，你也许需要在数据库中检查由测试用例所创建的数据，或者需要在运行时使用这些数据。这就意味着测试用例每一次运行时都要在数据库中创建新的唯一实体，以避免与其它测试用例相冲突。最好能编写一些实用工具类以方便地创建这些实体。

以下是一个创建资源的简单构造块：

```
public static ResourceBuilder newResource (String userName) {  
    ResourceBuilder rb = new ResourceBuilder();  
    rb.userName = userName + UnitTestThreadContext.getUniqueSuffix();  
    return rb; }  
  
public ResourceBuilder assignRole(String roleName) {  
    this.roleName = roleName + UnitTestThreadContext.getUniqueSuffix();  
    return this;  
}  
public Resource create() {  
    ResourceDAO resourceDAO = new ResourceDAO(UnitTestThreadContext.get  
Session());  
    Resource rs;  
    if (StringUtils.isNotBlank(userName)) {  
        rs = resourceDAO.createResource(this.userName);  
    } else {  
        throw new RuntimeException("must have a user name to create a  
resource");  
    }  
  
    if (StringUtils.isNotBlank(roleName)) {  
        Role role = RoleBuilder.newRole(roleName).create();  
        rs.addRole(role);  
    }  
    return rs;  
}  
  
public static void delete(Resource rs, boolean cascadeToRole) {  
    Session session = UnitTestThreadContext.getSession();  
    ResourceDAO resourceDAO = new ResourceDAO(session);  
    resourceDAO.delete(rs);  
  
    if (cascadeToRole) {  
        RoleDAO roleDAO = new RoleDAO(session);  
        List roles = rs.getRoles();  
    }  
}
```

```

        for (Object role : roles) {
            roleDAO.delete((Role)role);
        }
    }
}

```

ResourceBuilder是创建者模式与工厂模式的一个实现，你可以以方法链接的形式使用它：

```
ResourceBuilder.newResource("Tom").assignRole("Developer").create();
```

其中包含了一个打扫战场的方法**delete()**，在这次重构练习的早期，我并没有非常频繁地调用**delete()**方法，因为我经常启动整个系统并添加一些测试数据以检查饼图是否正确显示。

UnitTestThreadContext类非常有用，它保存了某个特定于线程的Hibernate Session对象，并且为你打算创建的实体提供了唯一字符串作为名称前缀，以此保证实体的唯一性。

```

public class UnitTestThreadContext {
    private static ThreadLocal<Session> threadSession=new ThreadLocal<
Session>();
    private static ThreadLocal<String> threadUniqueId=new ThreadLocal<
String>();
    private final static SimpleDateFormat dateFormat = new SimpleDateFormat(
"yyyy/MM/
dd HH_mm_ss_S");

    public static Session getSession(){>
        Session session = threadSession.get();
        if (session==null) {
            throw new RuntimeException("Hibernate Session not set!");
        }
        return session;
    }

    public static void setSession(Session session) {
        threadSession.set(session);
    }

    public static String getUniqueSuffix() {
        String uniqueId = threadUniqueId.get();
        if (uniqueId==null){
            uniqueId = " - "+dateFormat.format(new Date());
            threadUniqueId.set(uniqueId);
        }
    }
}

```

```

        return uniqueId;
    }

    ...
}

```

完成重构

现在我终于可以启动一个最小化的可运行架构，并执行这个简单的测试用例了：

```

protected void setUp() throws Exception {
    TestServerMain.run(); //setup a minimum running infrastructure
}

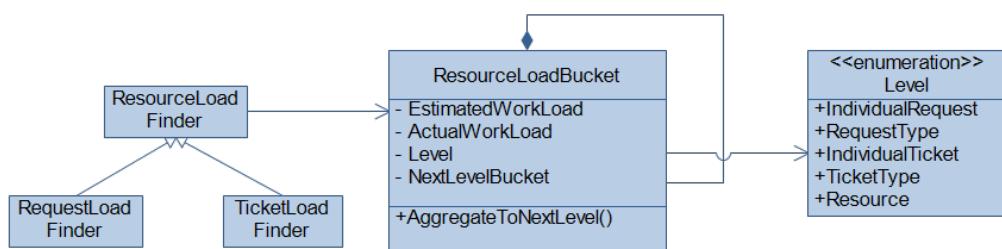
public void testResourceBreakdown(){
    Resource resource=createResource(); //use ResourceBuilder to build
    unique resources
    List requests=createRequests(); //use RequestBuilder to build unique
    requests
    assignRequestToResource(resource, requests);
    List tickets=createTickets(); //use TicketBuilder to build unique
    tickets
    assignTicketToResource(resource, tickets);
    Map result=new ResourceBreakdownService().search(resource);
    verifyResult(result);
}

protected void tearDown() throws Exception {
    // use TicketBuilder.delete() to delete tickets
    // use RequestBuilder.delete() to delete requests
    // use ResourceBuilder.delete() to delete resources
}

```

接下来我又编写了多个更复杂的测试用例，并且一边重构产品代码一边编写测试代码。

有了这些测试用例，我就可以将**ResourceBreakdownService**那个上帝类一点点进行分解。具体的细节就不必多啰嗦了，市面上已经有许多优秀书籍指导你如何安全地进行重构。为了本文的完整性，以下是重构后的结构图：



那个恐怖的“数组套Map再套数组再套Map.....”数据结构现在已经组织为新的**ResourceLoadBucket**类了，它的实现用到了[组合模式](#)。它首先包含了某个特别级别的预计完成时间和实际完成时间，下一个级别的完成时间将通过**aggregate()**方法聚合之后得到。最终的代码干净了许多，而且性能也更好。它也暴露了隐藏在原始代码的复杂性中的一些缺陷。当然，我也同时改进了我的测试用例。

在整个练习中，我始终坚持从系统的整体方向进行思考的原则，我从这个方向开始了重构之路，并始终保持正确的方向。那些相对于手头上的任务来说不太重要的问题就用临时方案绕过它。此外，我建立了一个具有最小可行性的测试架构，让我的团队也可以使用它继续重构其它一些领域。在测试构架中依然保留了一些**hack**的部分，因为从业务的角度来说没有太大的必要去清理它们。我所获的不仅是重构了一块非常复杂的功能区域，并且加深了对遗留系统的理解。将遗留系统当作一件易碎的瓷器并不会使你感觉更安全，只有大胆地深入它的内在并进行重构，才能使你的遗留系统在未来也能够继续它的使命。

关于作者



Chen Ping 住在中国上海，她于2005年计算机硕士毕业后，曾分别就职于朗讯与摩根士丹利，目前在HP担任开发经理一职。工作之余，她还喜欢学习一些中医知识。

查看英文原文：[Refactoring Legacy Applications: A Case Study](#)

原文链接：<http://www.infoq.com/cn/articles/refactoring-legacy-applications>

相关内容

- [降低代码重构的风险](#)
- [2012.3.6 微博热报：测试的价值&重构之惑](#)
- [用Mikado方法重构遗留软件](#)
- [重构时应避免◆◆度思考](#)
- [前搜狗搜索技术负责人郭昂指出：大多数重构可以避免](#)

推荐文章 | Article

JavaOne 2013综述：Java 8是革命性的，Java回来了

作者 [Matt Raible](#)，译者 [马德奎](#)

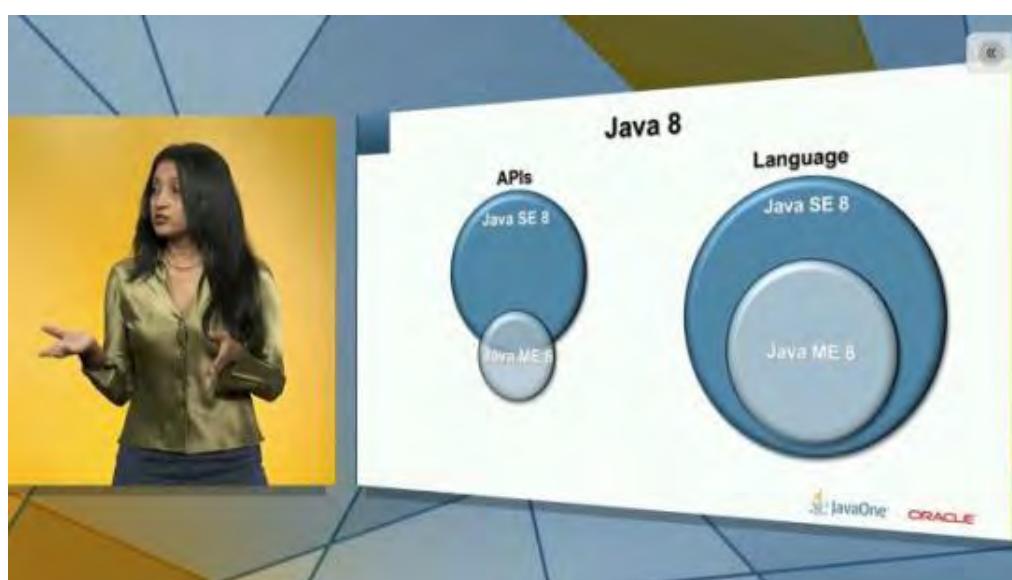
JavaOne 2013已于近日在旧金山举行。9月22日，来自Oracle员工[Peter Utzschneider](#)、[Nandini Ramani](#)和[Cameron Purdy](#)的[战略主题演讲](#)拉开了此次庆典的序幕，活动持续到9月26日。

这是第十八次JavaOne大会，Java社区并没有显出放缓的迹象。Utzschneider告诉观众，Java仍然是世界第一的开发平台，并且Java用户组的数量以每年10%的速度增长。

Java的未来

Ramani探讨了Java的现状以及Java如何有若干不同的SDK，这里仅举几例，如Java SE 7、CDC 1.1（基于SE 1.4.2）、CLDC（基于SE 1.3）和Java ME。过去，这些实现能很好地服务于特定的垂直市场，但多年来，每种实现都各自演变而变得越来越孤立。在Java 8中，Compact Profile将取代CDC。

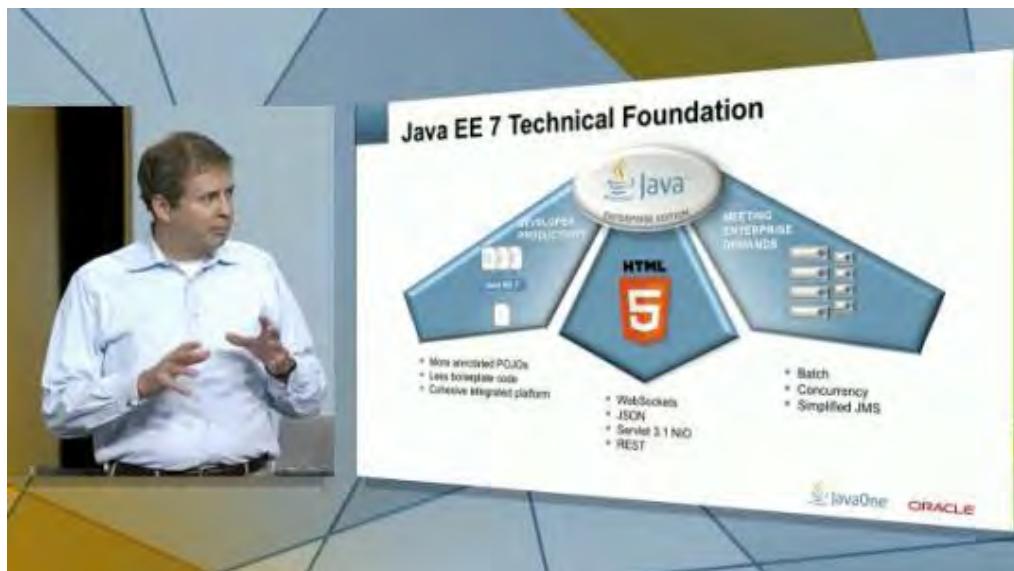
Java ME和Java SE之间的API会很类似，Java语言会支持两者的所有特性。Java 8会带来统一的平台：代码可移植、通用的API和工具——从SE嵌入式开发到服务器端Java EE开发。Java开发人员的类型将来会只有一种。



Java平台战略的其它要素还包括：同步发布（Java 8预览版现在已经可以下载）以及与合作伙伴（ARM、Freescale和Qualcomm）一起使Java成为芯片上的一

等公民。为了能够简单地移植和扩展Java Embedded，Oracle在8月份启动了Java平台集成器项目。

Java EE 7在刚刚过去的夏天发布，上两届JavaOne大会都针对它进行了讨论，可见这是一个重要的里程碑。Purdy提到，Java EE 7重点关注三个方面：开发人员的生产力、满足企业需求和HTML5。



两年前，在Java EE 7宣布的时候，主题是云。现在，Java EE 7有许多用于云部署的简单易用的特性，包括安全增强、默认资源、数据库结构生成、RESTful服务客户端API以及用于多租户应用程序的JSF皮肤。最后，Cameron宣布[Avatar项目](#)从现在起开源。Avatar跟Node.js类似，但运行在JVM上。

Java 8是革命性的，Java回来了

Java 8是此次大会一个很重要的演讲主题，这点从[Mark Reinhold的技术主题演讲](#)中可见一斑。Java 8包含了许多新特性，包括新的Date和Time API ([JSR 310](#))、Nashorn JavaScript引擎、类型注解 ([JSR 308](#))、Compact Profile和Lambda项目 ([JSR 335](#))。

Lambda是编程模型最大的单一升级，比以往任何升级都要大，甚至比泛型还大。我们精心协调，同时对虚拟机、语言和库进行了改良，自从有Java以来，这是第一次。但结果感觉仍然像Java。——Mark Reinhold

Oracle Java语言架构师[Brian Goetz](#)继续展示Lambda表达式如何去掉大量只用于表达简单思想的样板文件。在Lambda表达式出现之前，开发人员经常使用蹩脚的“牛肉面包比 (beef to bun ratio)”来表达思想，通常是用内部类。Goetz展示了下面的例子：

```

Collection<Person> people = ...;

Iterator<Person> ip = people.iterator();
while (ip.hasNext()) {
    Person p = ip.next();
    if (p.getAge() > 18) {
        ip.remove();
    }
}

```

为了抽象上述思想，开发人员可以用Predicate重写上述测试代码，写法如下：

```

Collections.removeAll(people,
    new Predicate<Person>() {
        public boolean test(Person p) {
            return p.getAge() > 18;
        }
    });

```

使用Lambda表达式，写法要简单许多：

```
Collections.removeAll(people, p -> p.getAge() > 18);
```

Lambda表达式不仅仅是一种更好的语法，它还使用invokedynamic生成一种更简洁高效的字节码。作为Java语言及其API已经变得更好的证明，Goetz谈论了新的流API以及如何用它在集合上进行批量操作。例如：

```

int highestWeight = people.stream()
    .filter(p -> p.getGender() == MALE)
    .mapToInt(p -> p.getWeight())
    .max();

```

这提供了语法、性能和抽象，而开发人员还获得了并行。Java 7新增了用于任务分解的Fork/Join框架，但其API使它很难使用。在Java 8中，开发人员只需修改一行代码，将stream（）改成parallelStream（）即可：

```

int highestWeight = people.parallelStream()
    .filter(p -> p.getGender() == MALE)
    .mapToInt(p -> p.getWeight())
    .max();

```

要了解更多关于Lambda项目的信息，查看[这里](#)或者[下载Java 8](#)。

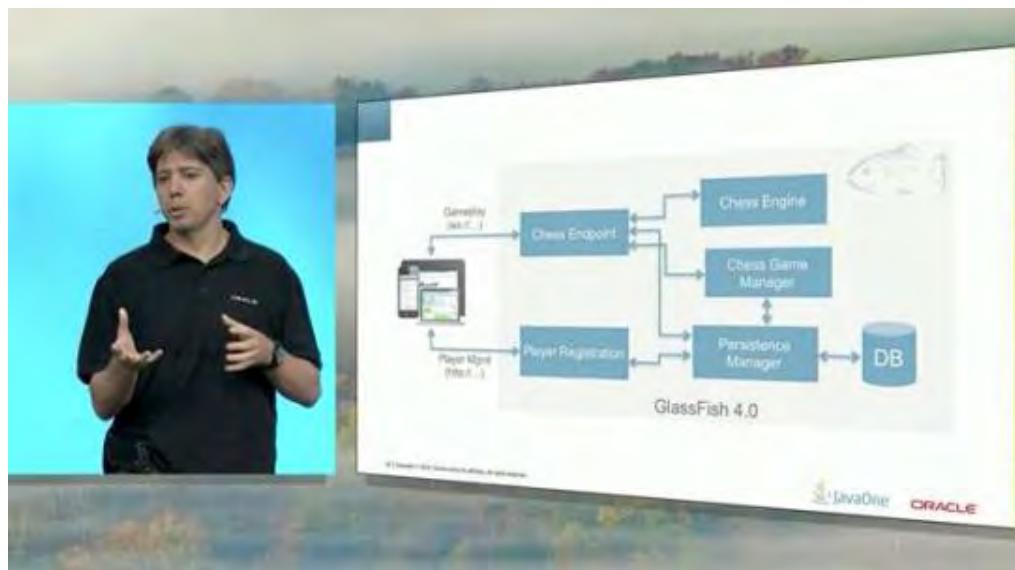
NetBeans 7.4

NetBeans工程总监[John Ceccarelli](#)用象棋游戏演示了HTML5应用程序（用Knockout.js编写）的编辑。他示范了如何在NetBeans中调整属性（与在Firebug或者Chrome开发人员工具中的操作类似）并使调整结果即时反映在浏览器中，而且无需重新加载。这看起来非常像IntelliJ IDEA的LiveEdit插件。

去年，NetBeans引入了Easel项目，其目的是在NetBeans中增加高级HTML5支持。Ceccarelli提到，社区对此的反应是“嘿，那太棒了，不过我们希望在EE项目里完成这一工作。”好消息是，就在JavaOne大会前夕，NetBeans 7.4 RC 1发布了，支持HTML5、Java EE、Java Web和Maven Web项目。

除了HTML和CSS的实时编辑功能外，NetBeans 7.4还支持Angular、jQuery和Knockout.js等JavaScript框架。这意味着代码编辑器可以识别JavaScript中所有的DOM id以及Model名称。NetBeans 7.4全是关于移动Web应用程序和移动混合应用程序的开发（通过支持Cordova 3.0）。有趣的是，如何在桌面浏览器之外的移动设备上使用实时编辑功能。最新的候选版本可以从[netbeans.org](#)上下载。

上文提到的演示程序，其象棋服务器用Java EE 7编写，并部署在GlassFish 4服务器上。该应用程序有五个不同的模块：象棋端点、玩家注册、象棋引擎、象棋游戏管理器和持久性管理器。



象棋服务器使用了许多Java EE新技术，包括：WebSockets、Batch、EJB、JPA和JAX-RS 2.0。客户端与服务器的所有通信都是通过JSON完成。GlassFish团队的一名成员[Santiago Pericas-Geertsen](#)展示了一些代码，用于说明在Java EE 7中建立WebSocket端点非常容易：

```

@ServerEndpoint(value = "/chessserver",
                encoders = MessageEncoder.class,
                decoders = MessageDecoder.class);
public class ChessServerEndpoint {

    @Inject private GameCatalog catalog;

    @OnMessage
    public Message onMessage(String message, Session session) {
        return message.processMe(this);
    }
    ...
}

```

跟该端点交互的客户端API与此非常类似，而且看上去很容易实现。

Oracle技术主题演讲中展示的最后一项技术创新是[DukePad](#)。这是一款可以在家DIY的平板电脑，基于Raspberry Pi和JavaSE 8 Embedded。他们发现CPU性能欠佳：Raspberry Pi CPU的速度几乎和Pentium II一样，比Samsung Galaxy S4慢14倍，比Intel Core i7处理器慢94到100倍。不过，它的GPU非常好，比Pentium II在1996年的速度快400倍。

[OpenJFX](#)开源了大部分组件，包括iOS和Android原型。演讲者提到，OpenJFX论坛很健康，他们已经从用户那里收到了大量的Bug报告。他们也收到了相当数量的社区贡献。JavaFX包含在JavaSE 8中。

对于Java 9及其未来，Oracle有若干方案，主要包括Java On GPUs、Reification（处理泛型的类型擦除问题）、JNI 2.0、Memory-Efficient数据结构以及用Jigsaw构建模块化平台。

Java社区

Oracle产品经理高级主管[Donald Smith](#)拉开了[Java社区主题演讲](#)的序幕。他带来了许多不同的人，在台上谈论Java技术令终端用户欢欣鼓舞的案例。

[Tori Wieldt](#)谈了“[Raspberry Pi挑战](#)”活动，25名开发人员参与其中，完成了六个项目。“心脏眼镜（Heart of Glass）”（用谷歌眼镜实时监控心率）和MTAAS（怪兽卡车服务）是此次活动中出现的两个成功的项目。Donald Smith还宣布，Oracle已经与Raspberry Pi基金会签订了一项OEM协议。后者将开始在他们的一部分镜像中包含Java SE，因此Java会以开箱即用的方式包含其中。

在社区主题演讲中，还有其它值得注意的公告，包括[Square](#)成为OpenJDK的一

员以及[Devoxx4Kids](#)正在寻找JUG负责人和家长，以便在他们所在的城市主办讲习班。当然，Aditya Gupta如何成为Minecraft编程高手的演示，也是亮点之一。同时，这也是主题演讲中的第一个Eclipse演示程序。他让猪飞起来，并使爆炸创造更多的爆炸。他是从他的爸爸[Arun Gupta](#)那里和[Minecraft Forge](#)上学到了其中大部分知识。

Alison Derbenwick Miller提到了[Oracle学院](#)。后者为从幼儿园到12年级的学生以及大学生提供课程。该学院去年培训了250万名学生，并提供了学生讲习班、教师发展和认证折扣。

在社区主题演讲的教育部分之后，进行了许多机器人演示。James Gosling甚至还作为嘉宾谈了他目前的工作。

Java回来了的证据

为什么说Java回来了？战略和技术主题演讲都帮助解释了这个问题。Java 8希望再次使Java编程变得有趣（通过减少样板代码），而Java EE 7中大量的API将会使构建现代应用程序变得简单。在最近的一些文章中，还有进一步的证据：

- Wired的“[Java迎来第二个春天：遗物归来，统治Web](#)”
- Dr. Dobb的“[Java濒临死亡？它看起来确实非常健康](#)”

如果读者没有机会参加今年的JavaOne大会，可以在日历上记下明年的会议日期。如果不是为了技术内容，那么可以来参与交流。Oracle答谢晚会上有免费的食物、啤酒以及Mroon 5和Black Keys的音乐表演。之后还有派对，展厅里到处都是人。

最重要的是，开发人员社区的热情似乎一如既往的强烈。

关于作者



Matt Raible成年以后大部分时间都在构建Web应用程序。他甚至在Netscape 1.0发布之前就开始摆弄Web。在超过15年的时间里，Matt帮助企业采用开源技术（Spring、Hibernate、Apache、Struts、Grails、Bootstrap和jQuery），并有效地使用这些技术。Matt著有*Spring Live*和*Pro JSP*。他还是AppFuse的创建者，这是一个可以使开发人员快速上手Java框架的项目。同时，他还是Apache Roller和Apache Struts项目的提交者。他了解并喜爱：HTML5、CSS、JavaScript、CoffeeScript、jQuery、AngularJS、Java、Spring、Scala、Play!Framework、Groovy、Rails、Tomcat、Jetty以及PhoneGap。

p

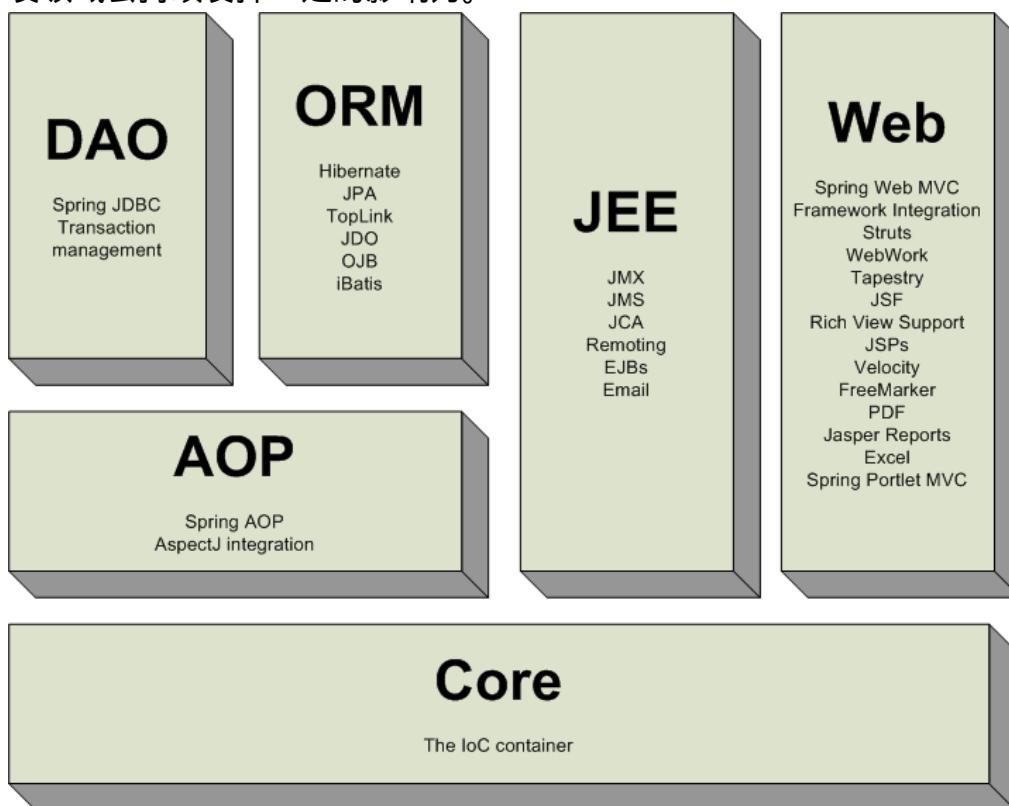
相关内容

- [JavaOne上海2013进展：50+场讲座，Java EE 7发布，Duke Choice大奖](#)
- [共创Java未来：JavaOne重回上海，现报名启动，关注Java EE 7、Java SE 8与Java Card](#)
- [“Java不会灭亡”后续报道](#)
- [打造未来的Java](#)
- [对话《X幻想》贾可：与客户端媲美的Java网页游戏引擎JGnet](#)

推荐语

细说Spring

近几年来Java特性有着快速的发展和演进，Oracle也在鼓励开发人员将应用直接迁移
到Java EE之上以抗衡Spring，在本期的专栏中，我们将会带领您一起回顾一些Sprin
g的细节，有赞同也有反对的案例，不过相信在未来的一段时间内，Spring在Java开
发领域会持续发挥一定的影响力。



特别专栏 | Column

我为何停止使用Spring

作者 张龙

[Johannes Brodwall](#)是一位程序员、解决方案架构师、用户组与会议组织者、会议演讲者与布道师。Johannes一直在不遗余力地将敏捷原则应用到大型软件项目中，不过他真正感兴趣的是与全世界的程序员分享更多关于编程的有趣经验。目前，Johannes就职于Exilesoft，担任首席科学家一职。近日，Johannes撰写了一篇关于他[为何停止使用Spring的文章](#)，在程序员群体中引起了很大的反响。

我之前发表的一篇题为[谦卑的架构师](#)的文章引起了很多争论，特别是Spring与依赖注入框架这个话题，这次我打算来谈谈为什么我要停止Spring。我是挪威最早采用Spring框架的一批人。我们在开发一个大型系统时最后不得不考虑诸如如何重用XML配置文件的各种不同机制等问题。最后，这演变成了@Autowire与component-scan，这种方式解决了大量配置文件的难题，不过却降低了人们了解全部源代码的能力，这直接导致开发者被限制在应用中非常小的部分代码中。随着应用变得越来越复杂，文化、工具、文档等东西导致开发者们不得不在一个个不必要的层次上构建另外一些不必要的层次。

不久之后，我尝试在不使用依赖注入框架的情况下构建应用，不过这引发了另外一些问题，那就是何时该使用“new”呢、何时使用setter或是构造器参数呢，哪种类型适合作为依赖呢、哪些东西导致了对基础设施的耦合呢。

根据直觉我发现DI框架确实能够帮助我改善设计，不过与此同时，我还发现在离开容器时，解决方案变得更小（这很棒！）、理解起来更简单，并且易于测试。这让我感到进退维谷，我发现使用容器的代价是非常高的，它会不断增加复杂性与规模，同时会降低一致性。不过话又说回来，它教会了我一些更棒的设计技巧。

总的来说，我个人认为一致、小型的系统要比那些为了解耦而解耦的系统更有价值。一致性与解耦是对立的，我站在一致性这一边。同时，我发现依赖注入文化对重用性有更强的偏爱，不过重用会引入耦合。如果模块A重用了模块B，那么我们就说模块A与模块B是耦合的。模块B中的变化可能会对模块A产生更好（修复Bug）或是更糟（引入新的Bug）的影响结果。如果重用所带来的好处更多，那就值得使用；否则就不值得。因此，重用与解耦是对立的两个方面，我自己更偏向于解耦。如果出现冲突，我个人认为优先级应该是一致性大于解耦，解耦大于重用性，而Spring的基因似乎与此相反。

Johannes的文章发布后，旋即引起了程序员社区的激烈讨论，有些讨论也很有意思，下面摘录几篇：

Marc Stock说到：

如果使用Spring的依赖注入增加了系统的复杂性，那么问题的症结在于你自己。我使用Spring有好几年的经历，它总是让事情变得更棒和更整洁。我不敢说所有的Spring项目都是这样，不过对于依赖注入来说绝对没错。也就是说，我发现有很多使用Spring的方式值得商榷，事实上他们做的很多事情都是不必要的（不过他们却并不这么认为）。如果你能举出Spring依赖注入会引起混乱的例子，我愿意拭目以待。

Johannes Brodwall说到：

文章的质疑很不错。我来举个简单的例子，假如一个系统有很多层次，所有东西都被加上了@Autowire与component-scan。我尝试实例化其中的某些服务，不过却缺少依赖。最后只能将所有的依赖加进来来实例化测试中所需的一个简单服务，因为查找存在哪些依赖、应该使用哪些依赖来作为模拟花费了我大量的时间。

Hendy Irawan说到：

文中提到“同时，我发现依赖注入文化对重用性有更强的偏爱，不过重用会引入耦合。如果模块A重用了模块B，那么我们就说模块A与模块B是耦合的”。这里需要对“模块”做一些澄清。接口会帮助我们更好地理解。在重用时，使用的是Spring、CDI还是你自己的什么东西并不重要。你提到“Spring文化”，是真的么？举个例子，使用(c3p0)数据源与PlatformTransactionManager(Jpa TransactionManager)来配置一个JPA EntityManager(FactoryBean)。这里会有大量的重用，不过解耦性却很不错。你可以将JPA切换到Hibernate，也可以将c3p0换到其他数据源，还可以将TransactionManager切换到Hibernate或是JTA的。

你讨厌XML配置，也讨厌@Autowired，不过配置总归是要有的。如果喜欢set或是new的方式，那么你可以使用注解，文中并没有提及这一点。如果使用的是CDI，那么讨厌@Autowired/@Inject有情可原，不过这是Spring，你有很多选择。根据我的经验，对于后端服务绑定使用注解配置，对于UI组件使用@Inject会更好一些。我们也不使用XML，你可以尝试一下这种方式。

Manuel Rascioni说到：

这几天我一直在思考是否该使用Spring，现在我觉得使用是值得的。看看整个(Web)应用，你需要这些东西：一个依赖注入管理系统、一个持久化框架、

在各个层之间转换对象的东西、一个安全框架与一个AOP系统（管理事务、安全等东西）。你有多种选择，也可以自己创建。如果使用现有的，那需要注意的就是不同框架的集成；如果自己创建，那就需要自己编写大量代码。我的经历告诉我使用Spring可以很好地解决这些问题。它不仅仅是个依赖注入框架，还是一套完整的框架生态系统，可以实现组件之间的解耦，同时又很好地实现了这些框架之间的集成。对于测试来说，我使用mock框架进行单元测试，而没有使用Spring（对于单元测试来说它太慢了）。对于集成测试来说，我们需要使用Spring配置文件。因此，根据我的经验来看，使用Spring是值得的。关于文中提到的耦合，如果模块C需要模块A的一些东西，你是怎么解决的呢？难道是编写一个新的模块，然后复制模块A么？这完全违背了DRY原则吧。

各位InfoQ读者，你是如何看待文中的观点以及各个评论的看法的呢？Spring从诞生到现在已经有很多年了，从最早的依赖注入与面向方面编程到现在的一站式框架体系，Spring本身也变得越来越庞大了，但同时功能也是越来越强大。特别是前不久刚刚发布的Spring 4.0更是增加了不少令人激动的新特性，那么在你的项目与系统中是否使用到了Spring呢？你觉得Spring带给你的好处与它本身的缺陷相比如何呢？换句话说，Spring框架的性价比高么？欢迎发表评论与大家一同分享你的观点与看法。

原文链接：<http://www.infoq.com/cn/news/2013/12/why-i-stop-using-spring>

相关内容

- [Spring团队的年度总结](#)
- [虽然遭遇Oracle的挑战，Spring框架依旧蓬勃发展](#)
- [Spring 4 增强了对Java 8、Java EE 7、REST 和HTML5的支持](#)
- [用Spring实现非端到端验收测试](#)
- [Spring框架4.0 GA版本发布](#)

特别专栏 | Column

用Spring实现非端到端验收测试

作者 [周宇刚](#)

验收测试让交付团队超越了基本的持续集成，即验证应用程序是否为用户提供了有价值的功能。不过对于刚开始尝试部署流水线的团队来说，想要自动化验收测试，需要跨过三大门槛。

一是实现和维护验收测试的技术门槛。理想情况下，验收测试最好可以模拟用户与应用程序的真实交互，因此如果有图形界面的话，验收测试理应通过这个界面和系统打交道。然而，直接通过GUI进行测试会遇到几个问题：界面变化速度很快、场景的准备相对复杂、拿到测试结果较难等。比如一个典型的WEB应用程序，如果通过GUI测试，那么一般需要解析HTML标签来填写参数，提交表单，最后再次通过解析来获取系统的返回值。如果测试代码中充斥着操作HTML的细节，测试的可读性就会大大下降，验收测试本身也更脆弱，在需求变更时反而会拖慢进度。

二是交付团队工作方式的变化。在传统团队中，需求分析、开发和测试是独立而又顺序的过程。就算能形成详细的需求文档，三方对同一段文字可能都有自己的理解。结果经常出现偏差，需求分析人员抱怨开发人员没有正确理解需求文档，开发人员抱怨需求文档不清晰、抱怨测试人员故意挑刺。敏捷实践和验收测试的出现缓解了这一问题，通过预先定义验收规格，减少文字上的误解，明确了开发工作的完成标准。不过这种思维方式的转变很难一蹴而就，需要交付团队及其利益关系人共同持续努力才能成功。

三是对组织的环境、配置管理及部署流程的挑战。当引入自动化验收测试后，对整个部署流水线的自动化程度会有更高要求。比如部署流水线应该能够自动将应用程序部署到待测试的环境中。如果应用程序依赖数据库，那么还应该能够部署数据库schema。另外一些运行时配置也需要通过脚本完成设置。这当中除了脚本准备之外，组织的环境管理也是要能跟上的。一般情况下，稍微大一些的组织都是有专门的运维团队（而非交付团队）来管理硬件设备和其配置的。因此，这个问题一般也涉及多个团队来协作解决。

面对这三座大山和进度压力，新手团队可能会感慨“信息量略大”而止步不前。这时不妨考虑各个击破，三个问题中的工作方式转变涉及的利益干系人最多，难度也最大；环境管理问题虽然涉及不同团队，但一般还是技术部门内的问题，关起门来好商量；验收测试的实现/维护主要是技术问题，相对最简单。如果时间和资源确实有限，不妨考虑牺牲一部分验收测试的有效性，采用简单的非端到端验

收测试，在自动化部署流程方面也可以做一些折中，集中力量转变工作方式。当整个工作已经进入节奏，再去改进某个具体环节时就顺利很多了。团队只要愿意迈出一小步，也能获得很大的价值。

过渡方案：相对简单的非端到端验收测试

如果团队的技术积累还不足，又没有足够的资源，不妨考虑简单一些的验收测试策略作为过渡方案。非端到端的验收测试是指直接调用应用程序内部的逻辑结构来驱动测试。由于测试代码和产品代码都使用同一种语言编写，可以省去比较繁琐的数据格式解析。而在准备测试数据和场景时，直接调用内部逻辑块一般也更方便。以典型的使用SpringFramework的Java WEB应用程序为例，团队可以采用和集成测试类似的基础架构来编写非端到端的验收测试。

这里所说的集成测试的目的是验证应用程序与外部服务的连接能否正常工作。这与应用程序实现的具体功能关系不大，因此一般只加载必需的ApplicationContext。

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:flow-ota-mail.xml",
    "classpath:context-infrastructure-mail.xml",
    "classpath:context-infrastructure-config.xml" })
public class MailsenderIntegrationTests implements IntegrationTests {

    @Resource(name = "oag.mail.inboundchannel")
    private MessageChannel inboundChannel;
    private GreenMail mailserver;                      只加载必需的上下文

    @Before
    public void startMailserver() {
        mailserver = new GreenMail(ServerSetupTest.SMTP);
        mailserver.start();
    }

    @After
    public void shutdownMailserver() {
        mailserver.stop();
    }

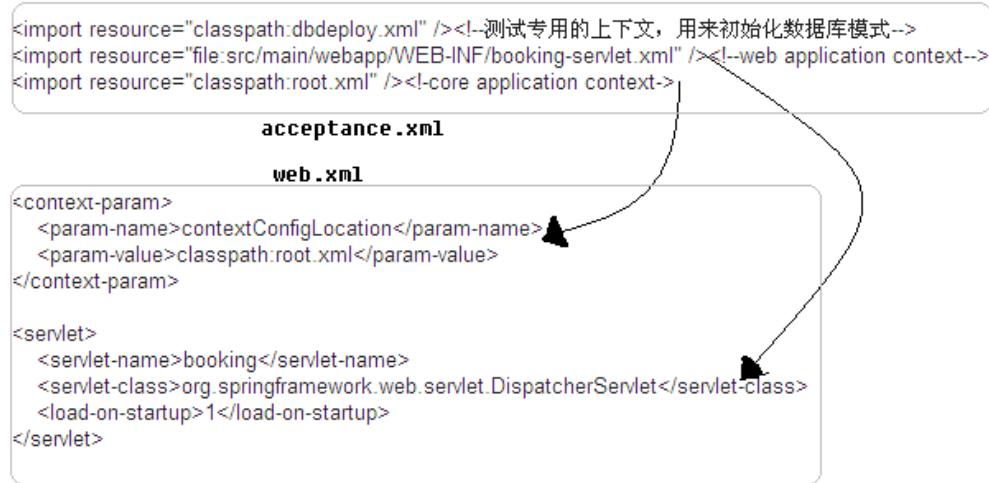
    @Test
    public void mailsendingIsFine() throws Exception {
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setSubject("test subject");
        mailMessage.setText("test content");           只验证是否能够集成，与功能无关
        message<SimpleMailMessage> message = messageBuilder.withPayload(
            mailMessage).build();
        inboundChannel.send(message);

        assertTrue(mailserver.waitForIncomingEmail(2000, 1));
        assertEquals("test content");
        assertEquals(mailserver.getReceivedMessages()[0].getSubject(),
            "[oag develop env]test subject");
        // omitted asserts using GreenMailUtil
    }
}

```

图表 1 集成测试

非端到端的验收测试可以采用和集成测试一样的测试基础架构，这样你就可以使用熟悉的测试库了，不同的是需要加载整个ApplicationContext以尽可能模拟应用程序被部署后的情况。



图表 2 加载整个上下文

由于可以访问整个ApplicationContext中的任一对象，我们可以通过访问应用程序的内部组件来执行测试，比如应用层的某个Service。但需要注意的是，选取的组件离UI层越远，其模拟真实用户交互的有效性就越差，而且受内部实现变更的影响越大。如果应用程序使用spring-webmvc的3.2以上版本，推荐使用[它的mvc测试库](#)。spring-test-mvc提供了类似http请求的DSL，此时虽然测试还是基于ApplicationContext，但并不直接访问内部组件了。这个方案对于新手团队比较友善，但请注意，这仅仅是个过渡方案，因为：

- 非端到端测试无法提供全面的回归测试，尤其是UI操作。在好几个项目中，我们发现仅采用非端到端测试覆盖的功能，团队不得不保留手工回归测试。如果UI上包含了大量复杂的控制逻辑甚至有业务逻辑泄漏到UI组件中，这会稀释验收测试带来的收益。
- 由于测试加载的ApplicationContext和Web容器加载的ApplicationContext存在差异，非端到端测试可能会漏掉一些问题。比如在非端到端测试中一次性加载了booking-servlet.xml和root.xml，使他们成为了一个整体的上下文，而实际上在Web容器中并不完全是这样，root.xml中的bean并不能访问和控制booking-servlet.xml中的bean。一个常见问题是如果在booking-servlet.xml中需要使用占位符，而恰巧我们已经在root.xml中有一个现成的<context:placeholder/>，看起来水到渠成，而且在测试中也没有问题，但实际部署到web容器时，就会加载失败。
- 非端到端的验收测试不能作为任务完成的最终标准。因为还有UI部分还没有完成。当这类验收测试通过时，我把这个任务称作“可以进入UI调试的”。

因此，如果团队有足够的技能和资源时还是应该直接使用端到端的验收测试，尤其当应用程序提供API（比如WebService）或是采用更易于解析的数据格式与客户端交互时。比如如果应用程序提供了基于JSON的API，完全可以使用[http-client](#)来驱动测试。

实现非端到端的验收测试

来看看第一个[验收测试](#)，这个案例来自于著名的[ddsample](#)，为了让验收测试能够看上去高端大气上档次，我们将使用Cucumber来组织验收测试。验收场景描述的是业务员如何登记航运货件并解释了登记完成后货件的各项状态。

Feature: Cargo Registration^{v1}

In order to get the cargo shipped^{v1}

As an operator^{v1}

I want to register a new cargo^{v1}

^{v1}

Scenario: Operator registers a new cargo^{v1}

When I submit the form with origin, destination and arrival deadline^{v1}

Then a new cargo is registered^{v1}

And the cargo is not routed^{v1}

And the transport status of the cargo is NOT_RECEIVED^{v1}

图表 3 第一个用户故事及其验收场景

Cucumber提供了一系列的Annotation来帮助我们验收场景文本与测试代码粘连在一起。

```
@WebAppConfiguration
@ContextConfiguration("classpath:acceptance.xml")//加载整个 ApplicationContext
public class CargoAdminSteps {
    @Autowired private WebApplicationContext wac;
    /* test data starts */
    private String origin = "CNSHA";
    private String destination = "CNPEK";
    private Date arrivalDeadline = aWeekLater();
    /* test data ends */
    private String trackingId;//由于有时断言会分布在若干个步骤中，
    private ResultActions cargo;//因此需要一个实例变量来传递被断言的对象
}

@When("^I submit the form with origin, destination and arrival deadline$")
public void I_fill_the_form_with_origin_destination_and_arrival_deadline() {
    this.trackingId = mockMvc().perform(put("/cargo").content(
        json(new RegisterCargoRequest(origin,destination,arrivalDeadline)))
        .contentType(MediaType.APPLICATION_JSON))
        .andDo(print()).andExpect(status().isOk());
    andReturn().getResponse().getContentAsString();
    //使用 spring-test-mvc 的 fluent api 发起请求并解析返回值
    this.cargo = mockMvc().perform(get("/cargo/" + trackingId))
        .andDo(print()).andExpect(status().isOk());
}
}
```

图表 4 实现验收测试-1

```

@Then("^a new cargo is registered$")
public void a_new_cargo_is_registered() throws Throwable {
    assertThat(trackingId, not(nullValue()));
}

@Then("^the cargo is not routed$")
public void the_cargo_is_not_routed() throws Throwable {
    cargo.andExpect(jsonPath("routingStatus").value(NOT_ROUTED.getCode()));
}

@Then("^the transport status of the cargo is NOT RECEIVED$")
public void the_transport_status_of_the_cargo_is_NOT_RECEIVED() throws Throwable {
    cargo.andExpect(jsonPath("transportStatus").value(NOT_RECEIVED.getCode()));
}

private MockMvc mockMvc() { return webAppContextSetup(this.wac).build(); }
private String json(Object object) throws Exception {
    return new String(new ObjectMapper().writeValueAsBytes(object), "UTF-8");
}
}

```

图表 5 实现验收测试-2

接下来，当运行测试时，你就可以得到一份漂亮的html报告

```

@RunWith(Cucumber.class)
@Cucumber.Options(features = { "classpath:" }, format =
{"html:target/acceptance-cucumber-html"})
public class AcceptanceTests {}

```

图表 6 测试运行入口

维护非端到端的验收测试

当团队开始编写验收测试之后，一般没过多久就会发现验收测试的开发进度越来越慢，而且有时遇到测试失败，但其实应用程序并没有缺陷的情况。验收测试对代码质量的要求也很高，相比单元测试，为了要达到测试所需的起始状态，验收测试的准备工作要更复杂。而且由于需要解析应用程序返回的数据，验收测试的断言也会更加琐碎。因此，团队最好尽早开始重构验收测试，下面的建议或许有用处：

- **建立最小测试数据集并且尽可能隔离测试的数据。**有时团队会发现两组测试由于依赖同一批数据而产生冲突，单独执行任一组测试都能通过，但一起执行就会失败。比如在示例代码中，对于不同的货件处理事件登记场景，验收测试都会注册一个新的货件。

- 隐藏断言细节。这样可以减少重复代码，并提升测试的可读性。把琐碎的解析逻辑隐藏在领域语言编写的方法中。

例如：如果多个测试用例都会对货件的运输状态进行断言，可以把解析细节提取出来，这样可以去除重复代码，并且减少语法噪声。

```

import static com.github.hippoom.dddsample.cargocqrs.acceptance.CargoAssertions.*;
public class CargoAdminSteps {
    @Then("^the transport status of the cargo is marked as ONBOARD_CARRIER$")
    public void the_transport_status_of_the_cargo_is_marked_as_ONBOARD_CARRIER() throws Throwable {
        theTransportStatusOf(cargo, shouldBe(ONBOARD_CARRIER));
    }
    @Then("^the transport status of the cargo is marked as IN_PORT$")
    public void the_transport_status_of_the_cargo_is_marked_as_IN_PORT() throws Throwable {
        theTransportStatusOf(cargo, shouldBe(IN_PORT));
    }
}
public class CargoAssertions {//通过静态导入使断言更贴近自然语言
    public static void theTransportStatusOf(ResultActions cargo, String transportStatus) throws Exception {
        cargo.andExpect(jsonPath("transportStatus").value(transportStatus));
    }
    public static String shouldBe(TransportStatus transportStatus) {
        return transportStatus.getCode();
    }
}

```

图表 7 抽取断言

- 尽可能使用已实现的功能来实现测试场景的准备。有一些步骤可能是多个测试用例都需要来准备数据的，可以把此类步骤抽取出来。这样也可以减少重复的代码，当应用程序随着需求变化时，验收测试会有更强的适应性，而且抽取出来的方法由于隐藏了技术细节，使用起来更简练。直接使用数据脚本的方案看起来很诱人，但一旦内部结构改变，数据脚本也得跟着改。

```

public class CargoAdminSteps {^
    @Given("^a cargo arrives at the final destination$")^
    public void a_cargo_arrives_at_the_destination() throws Throwable {^
        this.trackingId = aNewCargoIsRegistered();^
        theCargoIsRouted();^
    }^
    @Given("^a cargo is unloaded at the destination$")^
    public void a_cargo_has_been_unloaded_at_the_destination() throws Throwable {^
        this.trackingId = aNewCargoIsRegistered();^
        theCargoIsRouted();^
    }^
    private String aNewCargoIsRegistered() throws Exception {^
        return new CargoFixture(wac).^
            aNewCargoIsRegisteredWith(origin,destination,arrivalDeadline);^
    }^
    private void theCargoIsRouted() throws Exception {^
        new CargoFixture(wac).^
            assignCargoToRoute(trackingId,firstLeg,lastLeg).andExpect(status().isOk());^
    }^
}^

public class CargoFixture {^
    private WebApplicationContext wac;^
    public CargoFixture(WebApplicationContext wac) {^
        this.wac = wac;^
    }^
    public String aNewCargoIsRegisteredWith(String origin, String destination, Date arrivalDeadline) {^
        return mockMvc().perform(. ....);^
    }^
    public ResultActions assignCargoToRoute(String trackingId, LegDto... legs) throws Exception {^
        return mockMvc().perform(. ....);^
    }^
}^

```

图表 8 抽取公共步骤

验收测试对实践部署流水线的团队有着重要意义，也是很大的挑战。希望大家都能找到合适自己的方法。最后介绍几个有用的测试库。

- [Moco](#)，当有外部系统集成需求时，集成测试和验收测试的一大利器。在[示例代码](#)中你可以找到一处例子。
- [GreenMail](#)，如果应用程序需要发送邮件的话，它可以提供一臂之力。不过在部署流水线上的端到端验收测试中，由于一般应用程序和测试并不运行在同一台机器上，很难对邮件进行直接的断言。这时一种方案是修改应用程序的架构，把发送邮件的实现分离到一个专用的应用中去并使用消息队列。

列集成。那么在验收测试中，我们就可以通过监听对应的消息队列来断言了。

- [Awaitility](#)，在需要对异步处理进行断言时有所帮助。

作者简介

周宇刚是一位乐于磨练技艺的开发者。他的研究和兴趣包括IT架构、领域驱动设计和敏捷实践。

感谢[崔康](#)对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

原文链接：<http://www.infoq.com/cn/articles/spring-to-realize-non-end-acceptance-testing>

相关内容

- [Spring团队的年度总结](#)
- [虽然遭遇Oracle的挑战，Spring框架依旧蓬勃发展](#)
- [Spring 4 增强了对Java 8、Java EE 7、REST 和HTML5的支持](#)
- [我为何停止使用Spring](#)
- [Spring框架4.0 GA版本发布](#)

特别专栏 | Column

虽然遭遇Oracle的挑战，Spring框架依旧蓬勃发展

作者 张龙

众所周知，[Spring框架在几周前发布了4.0版](#)，这也是从2009年以来Spring发布的首个主要升级版，该版本支持数量广泛的特性，如HTML5/WebSocket、REST、Java 8、微服务框架等。虽然遭遇了来自于Oracle的各种挑战，不过Spring这个坚实的开源Java开发平台依旧在蓬勃发展，不断形成着自己的生态圈。

Spring 4.0对很多特性都进行了改进，这包括对Java 8标准版的支持（如Lambda）、一个安全的REST栈、HTML5/WebSocket集成、自定义注解及Java 7企业版支持（如JMS 2.0等）。时至今日，Spring 4.0的下载量已经有数百万之多，它提供的依赖注入等功能已经领先于Java企业版，并且不会受到流程缓慢的Java社区进程的阻碍。现在，从技术上来说，Spring已经完全可以替代掉由Oracle（之前的Sun）所制订的官方Java企业版标准。

不过从另一方面来看，Oracle也在极力说服Spring开发者[迁移到Java EE上](#)，并说其实Spring也没有什么特别的优势：

Oracle建议开发者从流行的Spring框架迁移到Java EE上，不过Spring创始人却认为这些技术可以搭配使用，和谐共处，说Oracle的这一举动完全是从财务角度着眼的。

在过去的几个月中，Oracle一直在各种Web会议上动员广大开发者从Spring迁移到Java EE上。有报道采访了来自于Luminis Technologies的Paul Bakker与Bert Ertman，他们建议大家迁移到Java EE 6上，并认为现今的Spring已经与过去不同了，相比于Java EE来说已经没有任何优势可言。此外，Bert Ertman也曾在去年的上海JavaOne大会上接受了InfoQ的[专访](#)，谈到了如何将Spring及遗留应用迁移到Java EE平台上。

Luminis资深软件工程师Bakker说到“很多开发者几年前根据Spring框架创始人Rod Johnson编写的图书认为企业级Java开发有很多不足之处，不过时至今日，我们有Java EE 5与Java EE 6，我们有经过完全修订的编程模型，他们非常轻量级，并且基于POJO。现在是时候让开发者们知道Java EE表示的并不是Java Evil Edition了，我们完全可以使用它来构建非常棒的企业应用”。

Spring中用来链接关联对象的依赖注入现在已经出现在了Java EE 5中。轻量级

、面向方面编程也在Java中得以实现。不过Johnson在回复问题的一封邮件中消除了Java EE与Spring之间的冲突，他认为“这都是人们自己搞出来的问题，Spring与Java EE 6完全可以和谐共处”。

Johnson说到“从本质上来说，Java EE 6想要干掉Spring的论调完全是由商业推动的，Spring减少了人们对于[Oracle WebLogic](#)等传统应用服务器的需求，用户可以选择更加轻量级的基础设施。虽然Java EE 6对之前的版本做了一些改进，不过Spring依然提供了非常重要的附加值”。

Spring的应用场景要比Java EE 6多不少，这样Spring用户就会有更多的选择权。他们可能并不想要使用Java EE应用服务器，即便使用Java EE亦是如此，他们可能不想使用Java EE 6，他们可能处于云环境下，这时Java EE并不适用，他们可能使用任意一台应用服务器，他们可能想要部署在各种设备上。这时，Spring的可移植性就是非常有价值的了。

Spring的生态系统所解决的问题要比Java EE多很多，比如说集成、批处理和非关系型数据等。细粒度的安全也得到了很完善的支持，使用Spring的组件模型可以提供很多其他的好处。

在Java EE领域中，根据Oracle的Arun Gupta所述，“Oracle正在寻求通过Java EE 7来扩展EJB的事务能力以及事务语义，我们在Java EE 7中所做的就是抽象出语义，使之具有更加广泛的应用场景。比如说对于Managed Bean或是CDI Beans等。借助于CDI Beans，Managed Bean可以通过Java类来实现”。

根据Java EE 6指南所述，如果一个顶层类是根据任何Java EE技术规范定义的或是满足某些条件，比如说是非静态的内部类，那么这个类就是个Managed Bean。Java EE 7的主要特性就在于支持GlassFish Server 4应用服务器。

不过，Pivotal的市场经理Pieter Humphrey却认为大家不必为此担心。他认为这个消息仅仅是一面之言而已，并相信Spring的流行还将持续下去。现今的Spring技术已经涵盖了移动应用开发、NoSQL、大数据以及云计算等领域。

Spring 4.0中值得关注的一个特性就是Spring Boot，这是个类似于Ruby on Rails的快速应用开发框架。Spring Boot能够极大减少样板代码的数量，开发者可以根据最少量的样板或是配置相关的代码开始项目的开发。

上个月，Spring网站有成千上万的访问者，达到了历史上的访问高峰。此外，通过Maven构建管理平台下载的Spring数量也在持续增长。

有很多读者也对Spring的未来及与Java EE标准之间的关系发表了自己的评论，分别从项目所采用的技术标准、Spring的特性及Java EE标准的不断演化等方面

谈起。

Jim Smith说到：

Spring Boot确实太酷了。我实在是搞不懂这个世界上怎么还有人使用臃肿的Java EE服务器，只是为了部署一个Restful Web服务。

Anil chalil说到：

我认为对于JEE来说，最好的东西就是CDI了，它直接能干掉EJB模型，Apache DeltaSpike就是围绕着CDI生态圈的一个项目。

Frans Thamura说到：

Spring不仅是个技术了，而且是个生态圈，他们的模块使之能够形成一个生态圈，JavaEE能做到这一点么？我觉得够呛。不过对于移植来说，没错，你是可以做到的，但Spring并不是私有技术，这是个问题。这就好比是为什么要将程序从JavaEE迁移到Oracle ADF上一样。

Nicolas说到：

没错，我知道Oracle的这个事情，当时很多人都在说JEE已经不再吸引人了。谁在乎呢？JEE的目标依旧是围绕着传统的应用服务器制订的，演进得非常缓慢，封闭，而且还抄袭其他的创新，至少要比别人晚5年时间。下一步应该是在云中兜售JEE了，作为获得厂商封闭策略的另一种途径。没错，我知道从理论上来说你可以不再依赖于某个JEE厂商，转而使用其他厂商的服务，如果他们二者真的是兼容的。

各位InfoQ读者，Spring现在基本已经成为了构建Java应用事实上的标准，而且也从最初的依赖注入和面向方面编程的框架发展成为现如今的一站式应用平台，Spring现已形成了自己的一个完善的生态圈，提供了对Web开发、移动开发、大数据、云计算、集成、批处理、NoSQL等系列的支持。相比于Spring，传统的JavaEE标准的关注度似乎没有以前那么高了，而且发展速度比较缓慢，这其中有很多的因素，毕竟标准的诞生还是需要经过方方面面的考量，一定的滞后性也是必然的。那么根据你的经验，采用Spring与采用标准的JavaEE各有什么样的利弊呢？从长远来看，哪一种策略才是最优的呢？欢迎写下你的看法与见解，我们一起讨论。

原文链接：<http://www.infoq.com/cn/news/2014/01/spring-springs-facing-oracle>

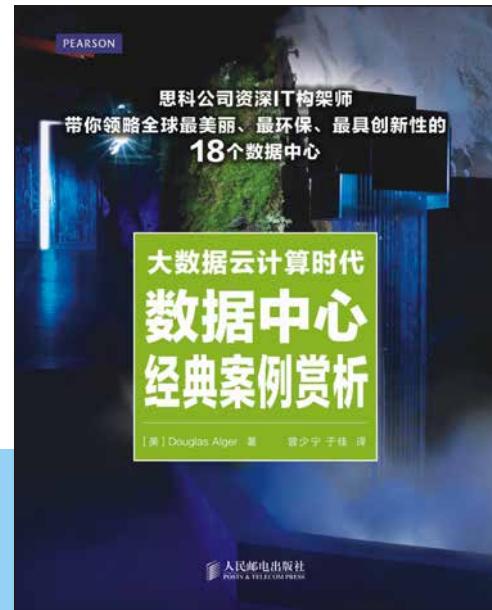
- * 思科公司资深IT构架师最新力作
- * 全面介绍世界上18个极具特色的数据中心
- * 采访问答方式，直接与项目单位就技术关键点展开探讨
- * 丰富精美的图片，直观感受数据中心之美

思科资深技术专家通过与数据中心设计师的访谈，揭示了数据中心决策要点，并说明了如何建设数据中心以及如何面对其他挑战，其中包含了思科，易趣，脸谱，雅虎等著名企业在不同环境下的的数据中心实践案例。思科资深技术专家通过与数据中心设计师的访谈，揭示了数据中心决策要点，并说明了如何建设数据中心以及如何面对其他挑战，其中包含了思

大数据云计算时代 数据中心经典案例赏析

作者：[美] Douglas Alger

译者：曾少宁 于佳



- * 最畅销的Oracle RAC原创图书之一
- * 立足于云端时代的全新视界
- * 安装、平台、私有云、高可用性全是干货

全书分4个部分，第一部分“安装”，从安装入手，分析安装过程出现中的新元素，第二部分“平台”，着重介绍Grid，包括Grid的内部组成、ASM、ADVM、ACFS、SCAN和SIHA等。

大话Oracle Grid：云时代的RAC

作者：张晓明





图灵社区 iTuring.cn



图灵教育微信



图灵访谈微信

图灵教育推荐

在线出版

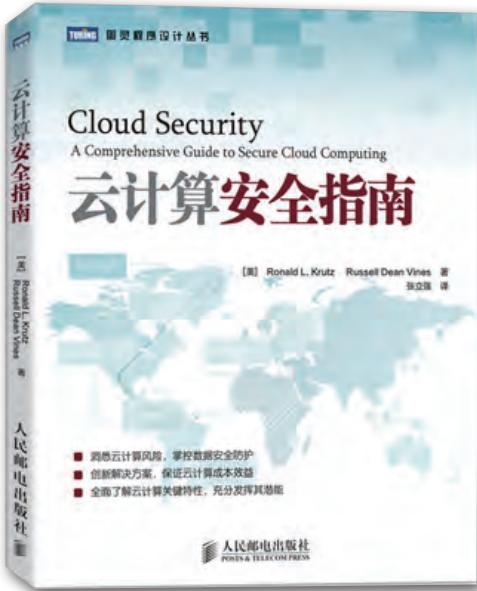
电子书

《码农》杂志

图灵访谈

线下活动

技术交流



- 洞悉云计算风险，掌控数据安全防护
- 创新解决方案，保证云计算成本效益
- 全面了解云计算关键特性，充分发挥其潜能

《云计算安全指南》全面探讨云的基础知识、架构、风险、安全原则，让我们不仅了解其灵活性、高效性、经济实用性，而且充分认识其脆弱性并学会补救方法。两位知名的安全专家 Ronald L. Krutz 和 Russell Dean Vines 将带你攻克数据所有权、隐私保护、数据机动性、服务质量与服务级别、带宽成本、数据防护、支持等难题，帮助你保证信息安全并通过云计算获得最大的投资回报。

书名：云计算安全指南

作者：[美] Ronald L.Krutz,
Russell Dean Vines

译者：张立强

书号：978-7-115-32150-3

详情请点击：<http://www.ituring.com.cn/book/991>

- 国内第一本云计算网络书
- 云计算与大数据时代，网络技术人员必看
- “弯曲评论”网站“拨云见日”系列热文加量
- 10倍的强烈之书首次完整呈现

通过阅读本书，读者将清楚地了解到如何在云计算与大数据时代构建安全、可靠、高速与灵活的网络。本书主要内容包括：云计算对基础架构的驱动、云计算网络的组成、如何构建安全可靠灵活的网络通道、虚拟化数据中心的扩张、外部和内部网络的实现、大数据网络设计要点，以及厂商解决方案等等。

本书语言通俗易懂，内容深入浅出，可作为云计算网络技术入门和提高阶段的自学、参考书籍。适合国内云计算网络、新一代网络建设、网络管理、系统集成行业的开发人员、技术工程师、售前与售后技术支持人员学习。

详情请点击：<http://www.ituring.com.cn/book/966>

书名：腾云：云计算和大数据时代网络

技术揭秘

作者：徐立冰

书号：978-7-115-31150-4

我们在微博：@图灵教育 @图灵新知 @图灵社区

我们在微信：图灵教育: turingbooks 图灵访谈: ituring_interview

读者俱乐部：218139230 (QQ群)

避开那些坑 | Void

关于Cassandra的错误观点

作者 [Jonathan Ellis](#)，译者 邵思华

正如[Apache Cassandra](#)的名称是来自于著名的物洛伊女巫一样，在它身上确实存在着各种误解。和大多数误解一样，至少在一开始时它们确实是有那么一点道理的，但随着Cassandra不断地深化与改善，这些误解的内容已经不复存在了。在本文中，我将针对五个常见的疑惑作出解释，澄清人们的困惑。

误解：Cassandra就是一个嵌套的map

随着使用Cassandra的应用程序变得越来越复杂，以下观点正在逐渐变得清晰起来：与“任何东西都是一个数组缓冲”或者“任何东西都是一个字符串”这种设计方式相比，schema与数据类型会使大型应用的开发与维护更加简单，

现如今，理解Cassandra的数据模型的最好方式是将其想像为表与行的组合，并且与关系型数据相似的是，Cassandra的列也是强类型的，并且可以进行索引。

你也许还听到过其它这些说法：

- “Cassandra是一种列数据库。”[列数据库](#)会将某个列的全部数据一起保存在磁盘上，这种方式对于数据仓库的检索方式是比较适合的，但对于那些需要对特定的行进行快速访问的应用程序来说就不太适合了。
- “Cassandra是一种宽行数据库。”这种说法有一定的道理，因为Cassandra的存储引擎是由Bigtable所启发而设计的，而后者可以说是宽行数据库的祖先了。但宽行数据库的数据模型与存储引擎结合得太过紧密，虽然实现起来比较容易，但针对它进行开发就增加了困难，而且它还使[许多优化方式](#)变得不可行了。

我们之所以在开始的部分选择避开“表与行”这种方法，原因之一是因为Cassandra的表与你所熟的关系型数据库的表的确存在着某些微妙的差别。首先，主键的首个元素是分区键，在同一个分区中的所有行都会存储在同一台服务器上，而分区是[分布在整个集群](#)中的。

其次，Cassandra不支持关联查询与子查询，这是因为在分布式系统中跨越硬件进行关联查询的性能很差。Cassandra的做法是鼓励你采用去正规化（denormalization）的方式，从一个单独的表中获取你所需的数据，同时提供[集合](#)等工具以简化操作。

举例来说，考虑一下以下代码所表示的users表：

```
CREATE TABLE users (
    user_id uuid PRIMARY KEY,
    name text,
    state text,
    birth_year int
);
```

目前多数主流服务都会考虑到一个用户可以拥有多个email地址的情况。在关系型数据库中，我们必须建立一个多对一的关系，随后使用关联查询将地址与用户关联起来，如以下所示：

```
CREATE TABLE users_addresses (
    user_id uuid REFERENCES users,
    email text
);

SELECT *
FROM users NATURAL JOIN users_addresses;
```

而在Cassandra中，我们会以去正规化的方式将所有email地址直接加入用户表中，使用一个set集合就可以完美地实现这一点：

```
ALTER TABLE users ADD email_addresses set<text>;
```

随后我们可以以如下方式为用户添加多个地址：

```
UPDATE users
SET email_addresses = {'jbe@gmail.com', 'jbe@datastax.com'}
WHERE user_id = '73844cd1-c16e-11e2-8bbd-7cd1c3f676e3'
```

关于Cassandra数据模型的更多内容，包括自届满数据 (self-expiring data) 以及分布式计数器，请参考[在线文档](#)。

误解： Cassandra的读取速度较慢

Cassandra采用的日志结构存储引擎意味着它不会在硬盘中寻找更新，也不会造成固态硬盘的写入放大，而同时它的读取速度也很快。

以下图示是关于随机访问读取、随机访问及顺序扫描，以及混合读写情况下的吞

吐数据，它们来自于多伦多大学的NoSQL性能指标分析结果：

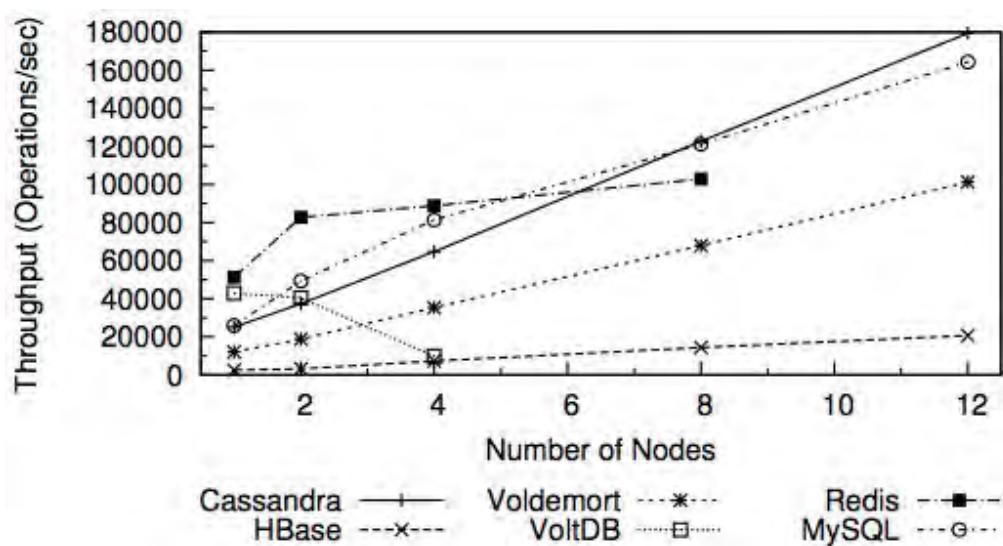


Figure 3: Throughput for Workload R

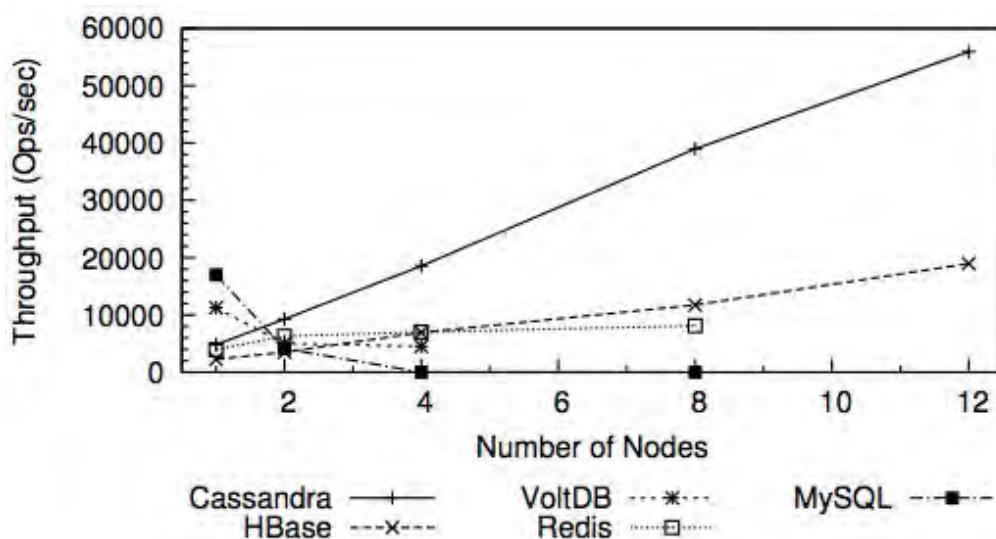


Figure 12: Throughput for Workload RS

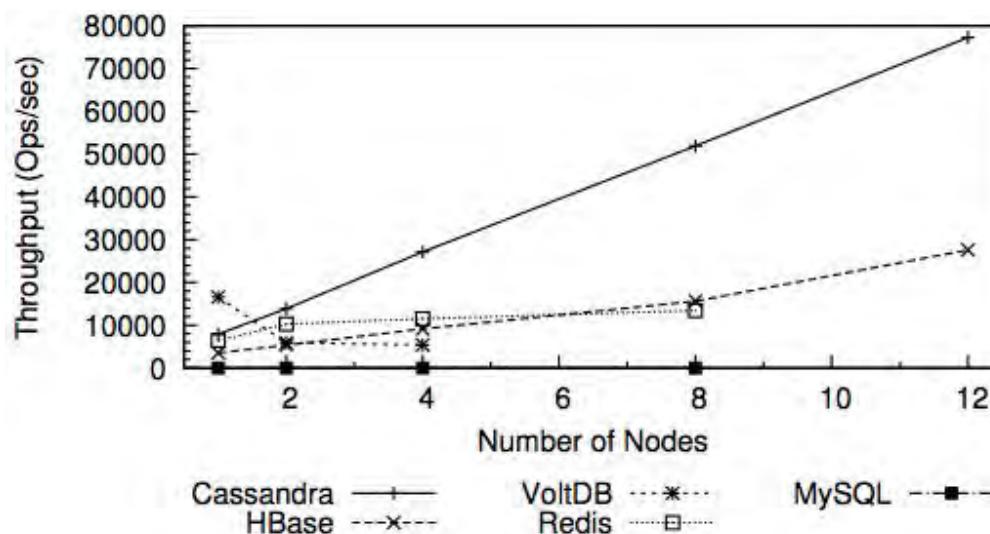
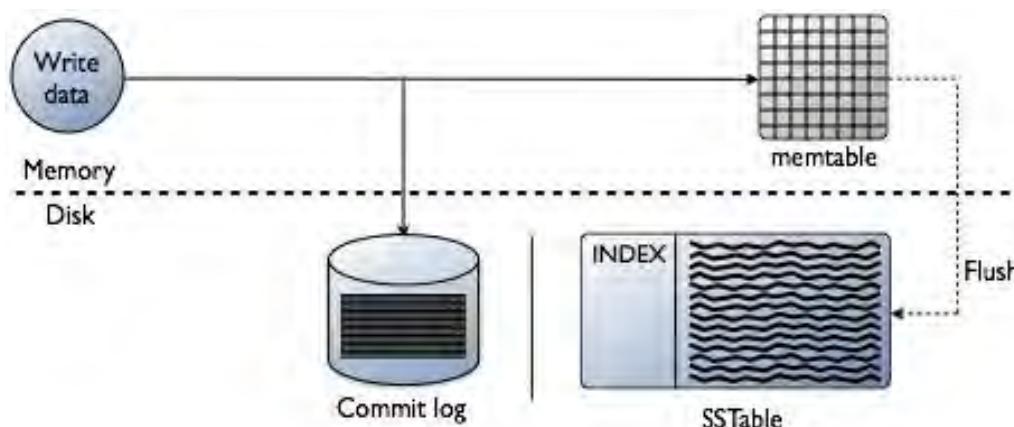


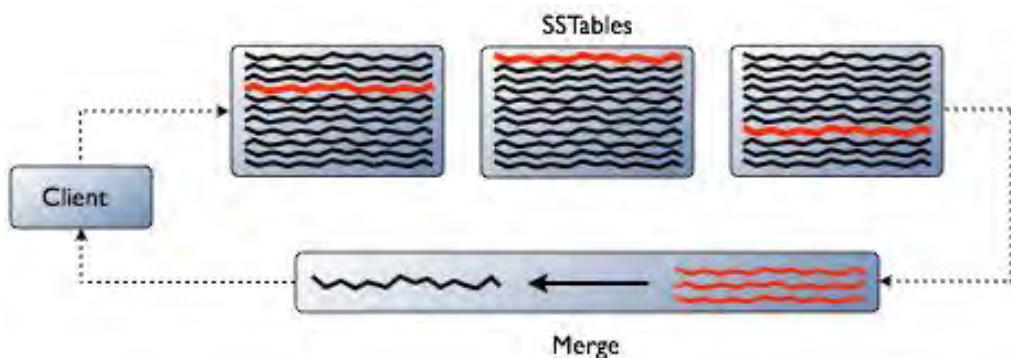
Figure 14: Throughput for Workload RSW

来自Endpoint公司的性能指标检测对Cassandra、HBase与MongoDB进行了比较，也证实了以上结论的正确性。

Cassandra是怎样实现的呢？从一个较高的层次来看，Cassandra的存储引擎看起来与Bigtable很像，它们都使用了一些相同的术语。更新内容会添加到某个commitlog中，随后收集到某个“memtable”里，该表会最终将数据写入磁盘并进行索引，类似于一个“sstable”：



原生的日志结构存储系统确实会倾向于在读取时稍慢，而由于同样的原因，它们在写入时会比较快：因为新的数据不会替换每一行中的原始数据，而是在后台压缩后再进行合并。因此在最坏的情况下，为了获取某个“碎片化”的行中的每一列的值，你将不得不检查多个sstable。



为了达到更好的读取性能，Cassandra对此基本设计方式进行了一些改善：

- 压缩策略是以插件形式提供的。例如LeveledCompactionStrategy[会通过更为激进的方式组合重叠的sstable，以实现对读取的优化。](#)
- Cassandra以时间倒序对sstable进行检查，如果你要求Cassandra执行SELECT x, y FROM foo WHERE key = 42语句，当Cassandra找到x和y对应的某个最新写入的数据时，[它就不会再去检查时间更早的sstable](#)。同样的原则也可以应用在对某个范围内的扫描上，虽然[稍有些麻烦](#)，但并非不可能实现。
- 在必须要从多个sstable中进行读取的情况下，我们[将会在读取时将去碎片化的结果重新写入](#)，这样之后的读取操作就只需要访问一个单独的表了。
- 当某个分区被访问时，它的[索引就会被缓存起来](#)，因此只需（每个sstable）一次查找就可以访问分区中的所有行了。
- 存储引擎的元数据中[会从堆中被剔除出去](#)，这样就避免了垃圾回收带来的影响问题。

误解：Cassandra的运行很麻烦

比起在一台独立的机器上运行数据库，在一个分布式系统上运行会在以下三个方面遇到更多的困难：

1. 初始化时的部署与配置
2. 日常维护工作，例如升级、添加新节点、或者替换故障节点
3. 故障检测

Cassandra是一个完整的分布式系统：因为Cassandra集群中的每一台机器都具有相同的角色，不存在专门的元数据服务器以调整内存中的各种信息，也不存在专门的配置服务器以进行分发，同样也不存在主服务器或者是故障转移服务器。这种特性使运行Cassandra从各方面而言都要比其它的一些替代产品来得更简单。这也意味着可以很方便地搭建一个单节点的集群以进行开发与测试任务，而它的功能表现与在一个包含大量节点的完整集群中的表现完全一样。

从某种意义上说，初始化时的部署工作其实是一项最不重要的任务，因为如果其它方面的表现相同，那么即使是初始化时的安装稍为复杂一些，随着系统生命周期

期的推移，这一点麻烦也不是很大的问题，并且自动化的安装工具能够为你隐藏大多数头疼的细节问题。但是！如果你因为对某个系统的了解太小而选择放弃手动安装，那么当你需要对某个问题进行故障诊断时就会遇到麻烦，因为解决问题需要你完全掌握系统中的各个部分是怎样在一起动作的。

因此我的建议是，如果你打算利用某些工具来进行安装，例如[Windows MSI安装文件](#)、Oracle的[Ops Center Provisioning](#)、或是[自配置的AMI](#)，请确保你已经深刻理解了安装过程中的细节。你可以研究一下这个[搭建Cassandra集群的两分钟示例](#)。

Cassandra的日常维护工作很简单。任一时刻都可以在某台节点上进行[升级工作](#)，而当某个节点停机时，其它节点会保留本应应用在该节点上的升级内容，并在[该节点恢复后将升级内容重新发送](#)给它。此外，添加新节点的操作可以在[整个集群中并行进行](#)，在操作完成后也无需重新进行平衡。

即使是对那些时间较长的、计划之外的停机状态进行处理也非常方便。Cassandra可在[运行时进行修复](#)，如同其它数据库中的rsync一样，它只需传输丢失的数据即可，这就将网络数据传输降至最低。如果你没有特别留意的话，也许根本不会意识到[它的发生](#)。

Cassandra在对[多数据中心的支持](#)方面在整个业界都处于领先地位，即使是[整个AWS区域挂掉](#)，甚至是[整个数据中心在飓风中被摧毁](#)这些极端情况下，也可以顺利地进行恢复。

最后，[DataStax OpsCenter](#)能够让你随时看到集群的各种重要系统指标，这样就可以方便地将历史活动数据与造成服务性能下降的事故相关联起来，以达到简化故障检测的目的。[Cassandra的DataStax社区版本](#)自带了一个“轻量级”版本的OpsCenter，可以在生产环境中免费使用。而[DataStax企业版](#)则包括了备份与恢复的调度，可配置的系统警告以及其它各种特性。

误解：在Cassandra上进行开发非常困难

早先的Cassandra Thrift API的目标是尽量减少用户开发一个跨平台的应用所付出的精力，而它也达到了这一目标，但现在业界已公认这套API是[难以使用的](#)。随后Cassandra推出了一套自己的SQL语言：CQL。它提供了一套更易于使用的接口，学习曲线更为平滑，同时还推出了[一套异步协议](#)，因此取代了Thrift API的使用。

CQL的早期使用者在两年前就可以使用0.8版本了，而今年1月份发布的1.2版本终于使CQL成为一个可用于生产环境的产品了。新版本包含了多种[驱动程序](#)，[性能也比Thrift更好](#)。DataStax也为[最流行的各种CQL驱动程序](#)提供了官方支持，

从此就可以不必再依赖来自社区的Thrift驱动程序的支持了，有时这种支持真的很差。

除了[在线文档中所介绍的CQL基础知识外](#)，Patrick McFadin的演讲“Next Top Data Model”（[第1部分](#)、[第2部分](#)）也是一个很好的CQL介绍。

误解：Cassandra依然是一种无人问津的边缘产品

从开源的角度来说，Apache Cassandra已有5年的历史，并且已经发布了多个版本，最新的版本2.0还是在今年七月刚刚发布的。而从企业的角度来说，DataStax提供了[DataStax企业版](#)，其中包含了一个经过认证的Cassandra版本，该版本经过了特定的测试、性能指标衡量、并且得到认可在生产环境中进行使用。

各个商业机构都看到了Cassandra为他们的组织所带来的价值，财富榜上的百强内有20个机构都依赖于Cassandra为他们的关键应用程序提供服务，这些机构来自几乎每个行业，包括金融、医疗、零售、娱乐、在线广告与市场。

将应用迁移至Cassandra平台上的最常见原因之一，是现有技术的伸缩性已经不足以满足现代化大数据应用程序的需求了。[全球最大的云应用Netflix已经将其95%的数据从Oracle迁移至Cassandra](#)，而Barracuda Networks也[用Cassandra取代了MySQL](#)，因为MySQL已经不能够应对巨量的垃圾请求了。而Ooyala[每天都要进行20亿次数据处理](#)，它所使用的Cassandra已有超过两个PB的数据量了。

对于那些管理和维护成本过高的陈旧的关系型数据库，Cassandra也在逐步取而代之。Constant Contact的首个基于Cassandra的项目[开发了三个月，成本为25万美元](#)，而他们之前基于关系型数据库的方案则开发了九个月，花费了250万美元。如今，他们已经搭建了[6个集群](#)，共有超过100TB的数据存放于Cassandra中。

在DataStax的[案例学习页面](#)，以及Planet Cassandra的[用户访问](#)页面上还可以找到许多其它案例。

这一条并非误解：关于在旧金山举办的2013 Cassandra Summit大会

我们刚刚结束了本次会议，这可以说是学习更多Cassandra知识的最好机会了。[本次会议有超过1100名与会者和65场演讲](#)，主讲者分别来自Accenture、Barracuda Networks、Blue Mountain Capital、Comcast、Constant Contact、eBay、Fusion-io、Intuit、Netflix、Sony、Splunk、Spotify、Walmart和其他一些公司。[演讲的幻灯片已上传](#)，而演讲视频也即将开放下载，具体时间

请密切关注[Planet Cassandra](#)的公告。

关于作者



Jonathan Ellis是DataStax公司的CTO兼联合创始人。在创办DataStax之前，他在受雇于Rackspace公司时在工作中大量使用了Apache Cassandra。而在Rackspace之前，他基于Reed-Solomon编码技术，为内容备份提供商Mozy编写了一个可容纳多个PB、伸缩性良好的存储系统。

查看英文原文：[Cassandra Mythology](#)

原文链接：<http://www.infoq.com/cn/articles/cassandra-mythology>

相关内容

- [NoSQL与RDBMS：何时使用，何时不使用](#)
- [12款免费与开源的NoSQL数据库介绍](#)
- [探讨NuoDB数据库的架构—2](#)
- [Jepsen：测试PostgreSQL、Redis、MongoDB以及Riak的分区容忍性](#)
- [SQL借助于NewSQL开始回归](#)

避开那些坑 | Void

Promise/A的误区以及实践

什么是Promise

作者 [李光毅](#)

Promise是一种让异步代码书写起来更优雅的模式，能够让异步操作代码像同步代码那样书写并且阅读，比如下面这个异步请求的例子：

```
$ .get("/js/script.js", function () {
    // callback to do
})
```

就可以改写为Promise模式：

```
var promise = $ .get("/js/script");
```

返回值promise即代表操作的最终结果。返回值promise也可以作为“第一类对象” ([First-class object](#)) 被当做参数传递。这个模式最大的优势就是避免了传统异步操作代码中，回调函数嵌套回调函数的糟糕情况。

如果你之前对Promise模式有所了解的话（可以参考InfoQ之前的这篇[文章](#)），谈到Promise，最先想到的一定是它的then函数，的确它非常重要，在Promise模式的定义中([CommonJS Promises/A](#))中，then函数是这么被定义的：

(原文) A promise is defined as an object that has a function as the value for the property then: then(fulfilledHandler, errorHandler, progressHandler)

(译) 一个promise被定义为一个拥有then属性的对象，并且此then属性的值为一个函数: then(fulfilledHandler, errorHandler, progressHandler)

也就是说每一个promise结果一定会自带一个then函数，通过这个then函数，我们可以添加promise转变到不同状态(定义中promise只有三种状态，unfulfilled, fulfilled, failed.这里说的状态转变即从unfulfilled至fulfilled，或者从unfilled至failed)时的回调，还可以监听progress事件，拿上面的代码为例：

```

var fulfilledHandler = function () {}
var errorHandler = function () {}
var progressHandler = function () {}

$.get("/js/script").then(fulfilledHandler, errorHandler, progressHandler)
  
```

这有一些类似于

```

$.ajax({
  error: errorHandler,
  success: fulfilledHandler,
  progress: progressHandler
})
  
```

这个时候你会感到疑惑了，上面两种方式看上去不是几乎一模一样吗？——但promise的重点并非在上述各种回调函数的聚合，而是在于提供了一种同步函数与异步函数联系和通信的方式。之所以感到相似这也是大部分人对Promise的理解存在的误区，只停留在then的聚合(aggregating)功能。甚至在一些著名的类库中也犯了同样的错误(下面即以jQuery举例)。下面通过列举两个常见的误区，来让人们对Promise有一个完整的认识。

Promise/A模式与同步模式有什么联系？

抛开Promise，让我们看看同步操作函数最重要的两个特征

- 能够返回值
- 能够抛出异常

这其实和高等数学中的[复合函数](#)(function composition)很像：你可以将一个函数的返回值作为参数传递给另一个函数，并且将另一个函数的返回值作为参数再传递给下一个函数……像一条“链”一样无限的这么做下去。更重要的是，如果当中的某一环节出现了异常，这个异常能够被抛出，传递出去直到被catch捕获。

而在传统的异步操作中不再会有返回值，也不再会抛出异常——或者你可以抛出，但是没有人可以及时捕获。这样的结果导致必须在异步操作的回调中再嵌套一系列的回调，以防止意外情况的发生。

而Promise模式恰好就是为这两个缺憾准备的，它能够实现函数的复合与异常的抛出(冒泡直到被捕获)。符合Promise模式的函数必须返回一个promise，无论它是fulfilled状态也好，还是failed(rejected)状态也好，我们都可以说把它当做

同步操作函数中的一个返回值：

```
$.get("/user/784533") // promise return
.then(function parseHandler(info) {
    var userInfo = parseData(JSON.parse(info));
    return resolve(userInfo); // promise return
})
.then(getCreditInfo) // promise return
.then(function successHandler(result) {
    console.log("User credit info: ", result);
}, function errorHandler(error) {
    console.error("Error:", error);
})
```

``` 上面的例子中，`$.get`与`getCreditInfo`都为异步操作，但在Promise模式下，（形式上）转化为了链式的顺序操作

`$.get`返回的promise由`parseHandler`进行解析，返回值“传入”`getCreditInfo`中，而`getCreditInfo`的返回值同时“传入”`successHandler`中。

之所以要在传入二字上注上引号，因为并非真正把promise当做值传递进入函数中，但我们完全可以把它理解为传入，并且改写为同步函数的形式，这样以来函数复合便一目了然：

```
try {
 var info = $.get("/user/784533"); //Blocking
 var userInfo = parseData(JSON.parse(info));

 var resolveResult = parseData(userInfo);
 var creditInfo = getCreditInfo(resolveResult); //Blocking

 console.log("User credit info: ", result);
} catch(e) {
 console.error("Error:", error);
}
```

但是在jQuery1.8.0版本之前，比如jQuery1.5.0（jQuery在1.5.0版本中引入Promise，在1.8.0开始得到修正），存在无法捕获异常的问题：

```
var step1 = function() {
 console.log("-----step1-----");
 var d = $.Deferred();
 d.resolve('Some data');
 return d.promise();
```

```

},
step2 = function(str) {
 console.log("-----step2-----");
 console.log("step2 recevied: ", str);

 var d = $.Deferred();
 // 故意在fulfilled hanlder中抛出异常
 d.reject(new Error("This is failing!!!"));
 return d.promise();
},
step3 = function(str) {
 console.log("-----step3-----");
 console.log("step3 recevied: ", str);

 var d = $.Deferred();
 d.resolve(str + ' to display');
 return d.promise();
},
completeIt = function(str) {
 console.log("-----complete-----");
 console.log("[complete]----->", str);
},
handleErr = function(err) {
 console.log("-----error-----");
 console.log("[error]----->", err);
};

step1().
then(step2).
then(step3).
then(completeIt, handleErr);

```

上述代码在[jQuery-1.5.0](#)中运行的结果：

```

-----step1-----
-----step2-----
step2 recevied: Some data
-----step3-----
step3 recevied: Some data
-----complete-----
[complete]-----> Some data

```

在step2中，在解析step1中传递的值后故意抛出了一个异常，但是我们在最后定义的errorHandler却没有捕获到这个错误。

忽略捕获异常的错误，上面的结果还反映出另一个问题，最后一步completeHandler中处理的值应该是由step3中决定的，也就是step3中的

```
d.resolve(str + ' to display');
```

最后应打印出的结果为

```
some data to display
```

而在[jQuery-1.9.0](#)中异常是可以捕获的，运行结果为：

```
-----step1-----
-----step2-----
step2 recevied: Some data
-----error-----
[error]-----> Error {}
```

但是打印出的结果仍然有问题

注意到step3没有执行，因为step3中只定义了fulfilled的回调，异常只有在最后errorhandler才被捕获。

其实我们可以试试，在step3中添加处理异常的回调函数：

```
step1().
then(step2).
then(step3, function (str) {
 console.log("-----[error] step3-----");
 console.log("step3 revecied: ", str);

 var d = $.Deferred();
 d.resolve(str + ' to display');
 return d.promise();
}).
then(completeIt, handleErr);
```

运行结果如下：

```
-----step1-----
-----step2-----
step2 recevied: Some data
```

```
-----[error] step3-----
step3 reveced: Error {}
-----complete-----
[complete]-----> Error: This is failing!!! to display
```

虽然错误在step3被捕获了，但是由于我们将错误信息传递了下去，最后一步打印出的仍然是error消息

## 细节：返回Promise

让我们继续看看Promise/A定义的第二段：

(原文) This function should return a new promise that is fulfilled when the given fulfilledHandler or errorHandler callback is finished. This allows promise operations to be chained together. The value returned from the callback handler is the fulfillment value for the returned promise. If the callback throws an error, the returned promise will be moved to failed state.

(译文) 这样的函数应该返回一个新的promise，该promise是被指定回调函数(成功执行或者捕获异常)解析之后的结果。如此一来promise之间的操作便能链式的串联起来。回调函数返回的值是解析返回的promise的结果。如果回调函数抛出了异常，返回的promise便会转化为异常状态

这段定义告诉我们两点：

- 无论返回值是fulfilled也好，还是被rejected也好，必须返回一个新的promise；
- then关键字并非只是各个回调的填充器，在输入的同时它同时也输出新的promise，以便形成链式；

同样以jQuery的代码为例：

```
var step1 = function() {
 console.log("-----step1-----");
 var d = $.Deferred();
 d.resolve('Some data');
 return d.promise();
};

var step2 = function (result) {
 console.log("-----step2-----");
 return result;
};
```

```

 console.log("step2 received: " + result);
 var d = $.Deferred();
 d.resolve("step2 resolve: " + result);
 return d.promise();
 }

 var step3 = function (result) {
 console.log("-----step3-----");
 console.log("step3 received: " + result);
 var d = $.Deferred();
 d.reject(new Error("This is failing!!!"));
 return d.promise();
 }

 var promise = step1();

 var promise1 = promise.then(step2);
 var promise2 = promise.then(step3);

```

var promise1 = promise.then(step2); var promise2 = promise.then(step3); `` step1返回的promise是fulfilled状态，但不同的是step2 fulfilled之后，返回一个仍然可被解析的promise(1)，而step3则抛出一个异常(promise2)。

按照定义所说，promise1与promise2是相互不同的promise，无论是被正确解析还是抛出异常，返回的都应该是一个独立的promise。

为了验证产生的是否为独立的promise，只需看他们的执行结果如何，接着给promise1和promise2定义fulfilled和failed回调函数：

```

promise1.then(function (result) {
 console.log("Success promise1: ", result);
}, function () {
 console.log("Failed promise1: ", result);
})

promise2.then(function (result) {
 console.log("Success promise2: ", result);
}, function (result) {
 console.log("Failed promise2: ", result);
})

```

让我们看看在jQuery-1.5.0中执行的结果：

```

Success promise1: Some data
Success promise2: Some data

```

虽然是一个被抛出的异常，但仍然可以被正确解析，并且解析使用的参数是上一个promise的返回值

在jQuery-1.9.0中：

```
Success promise1: step2 resolve: Some
Failed promise2: Error {}
```

能被正常解析。

## 实践

完整认识了promise之后，我们可以用简单的代码实现一个Promise模式。

参照jQuery的Deferred，我们可以了解Promise的大致结构：

```
var Promise = function () {}

Promise.prototype.when = function () {
 // to do
}

Promise.prototype.resolve = function () {
 // to do
}

Promise.prototype.rejected = function () {
 // to do
}
```

Promise.prototype.rejected = function () { // to do } ``` 并且我们用最简单一个异步操作setTimeout来验证我们的Promise是否奏效：

```
var delay = function (throwError) {
 var promise = new Promise();

 if (throwError) {
 promise.reject(new Error("ERROR"));
 return promise;
 }

 setTimeout(function () {
 promise.resolve("some data");
 }, 1000);
}
```

```

 }, 1000);

 return promise;
}

delay().then(function (result) {
 console.log(result);
}).then(function () {
 console.log("This is the second successHandler");
}).then(function () {
 console.log("This is the third successHandler");
})

```

首先我们要为每一个Promise准备一个队列来存储自己的回调函数

```

function Promise() {
 this.callbacks = [];
}

```

我们可以暂且把then()理解为往队列中填入回调的函数，并且为了能以链式的形式添加处理函数，最后必须返回当前promise：

```

Promise.prototype.then = function (successHandler, failedHandler) {
 this.callbacks.push({
 resolve: successHandler,
 reject: failedHandler
 });

 return this;
}

```

} ``` 其实resolve和reject虽然名称不同，但是都是执行各自对应的回调函数，于是可以抽象出一个公共的complete方法：

```

Promise.prototype = {
 resolve: function (result) {
 this.complete("resolve", result);
 },

 reject: function (result) {
 this.complete("reject", result);
 },

 complete: function (type, result) {
 // to do
 }
}

```

```

 }
}
```

complete的工作非常显而易见，根据type不同执行回调函数出队，以result为参数，执行相应type的的函数：

```

complete: function (type, result) {
 this.callbacks.shift()[type](result);
}
```

但是这样只能执行队首回调函数，在链式的情况下，可能在callbacks中添加了多个回调函数，为了实现链式的执行，需要把callbacks中的回调全部出队，complete可以改进为：

```

complete: function (type, result) {
 while(this.callbacks.length) {
 this.callbacks.shift()[type](result);
 }
}
```

完整版如下：

```

function Promise() {
 this.callbacks = [];
}

Promise.prototype = {
 resolve: function (result) {
 this.complete("resolve", result);
 },
 reject: function (result) {
 this.complete("reject", result);
 },
 complete: function (type, result) {
 while(this.callbacks[0]) {
 this.callbacks.shift()[type](result);
 }
 },
 then: function (successHandler, failedHandler) {
 this.callbacks.push({
 resolve: successHandler,
 reject: failedHandler
 });
 }
}
```

```

 });

 return this;
 }
}

```

第一个版本即完成，可以看到测试上面开始例子的结果，能够顺利打印出信息。

接下来我们来完成处理异常部分

首先我们写一个能够故意抛出异常的测试用例

```

delay(true).then(function (result) {
 console.log(result);
}, function firstErrorHandler(error) {
 console.error("First failedHandler catch: ", error);

 var promise = new Promise();
 promise.resolve("some data");
 return promise;

}).then(function secondSucHandler(result) {
 console.log("Second successHandler received: ", result);
}, function (error) {
 console.error("Second failedHandler catch: ", error);
})

```

我们在delay中抛出异常，希望在firstErrorHandler捕获异常后，返回一个能fulfilled的promise，并且用secondSucHandler顺利解析、

如果直接用上面版本执行，会发现没有任何结果，为什么？

因为在执行**delay()**时，第一个**reject**也同时被执行，但此时**then**函数还没执行，也就是处理**reject**的**handler**还没有被定义。当然也就不会有任何结果了。反过来也能想通，也就能说通**resolve**能被执行。

那么我们只要在**reject**函数中加上一定的延时即可：

```

...
reject: function (result) {
 var _this = this;
 setTimeout(function () {
 _this.complete("reject", result);
 });
},
...

```

执行测试代码结果如下：

```
First failedHandler catch: Error
Second failedHandler catch: Error
```

虽然错误被捕获了，但错误被一直传递一下去了，这也就是我们之前说的jQuery无法返回新的promise，接下来要解决这个问题。

我们来写一个更复杂的测试用例，来验证下面的解决方案：

```
delay()
// -----Level 1-----
.then(function FirstSucHandler(result) {
 console.log("First successHandler received: ", result);

 var p = new Promise();
 p.reject(new Error("This is a test"));
 return p;

}, function FirstErrorHandler(error) {
 console.error("Second failedHandler catch: ", error);
})
// -----Level 2-----
.then(function SecondSucHandler(result) {
 console.log("Second successHandler received: ", result);

})
// -----Level 3-----
.then(function ThirdSucHandler(result) {
 console.log("Third successHandler received: ", result);
}, function ThirdErrorHandler(error) {
 console.error("Third failedHandler catch: ", error);
})
// -----Level 4-----
.then(function FourSucHandler(result) {
 console.log("Fourth successHandler received: ", result);
})
```

正确的执行顺序应该是FirstSucHandlerfulfilled之后抛出异常，略过Second SucHandler，异常被ThirdErrorHandler捕获，并且返回一个新的promise，由FourSucHandler解析。

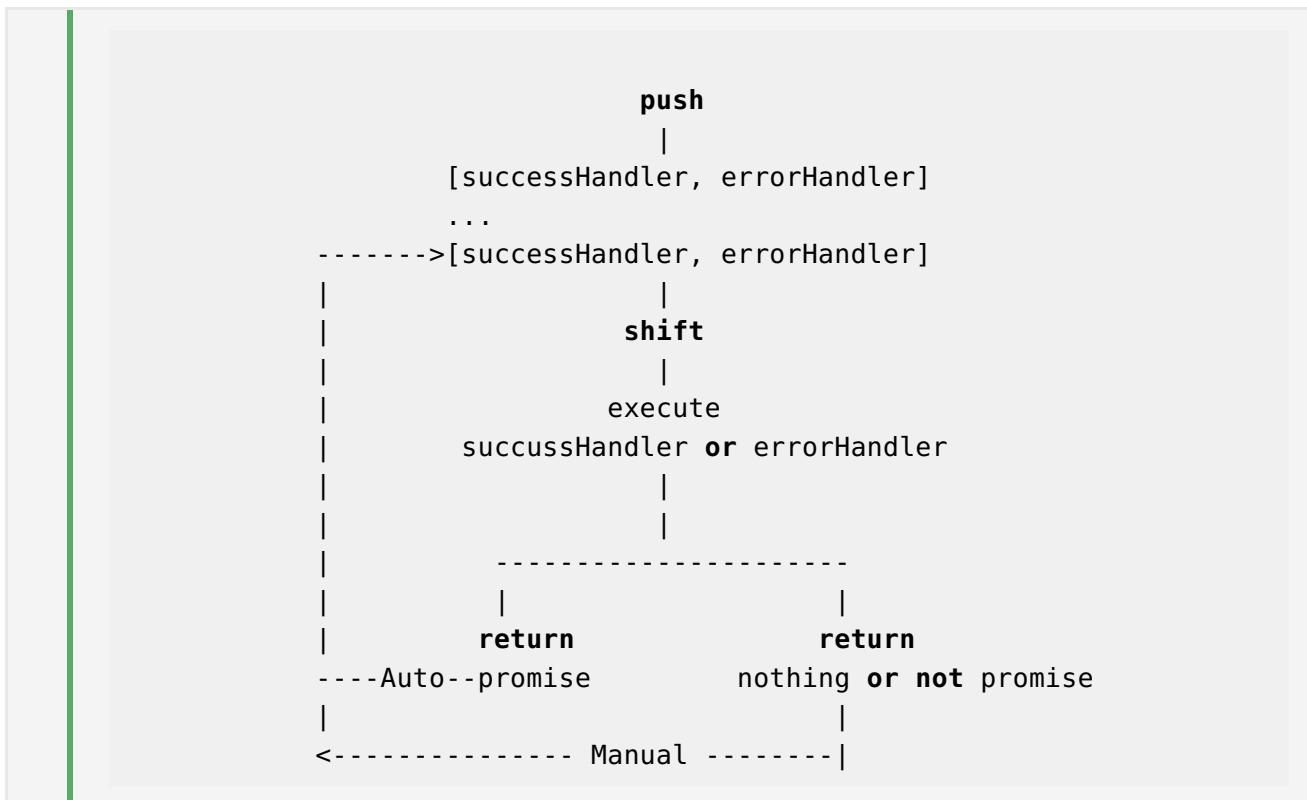
接下来还要修复并且考虑这些问题 1. 当异常被捕获之后将阻止异常往下传递 2. 定义中描述在fulfilled之后必须返回一个新的promise，但如果返回新的pro

mise，或者是返回其他的值，应该作何处理？

对于第二点，我们暂且处理规则是：

1. 如果没有返回值，那么下一个回调函数将继续解析上一promise
2. 如果返回值存在，但返回值不为promise，默认调用resolve handler，并且返回值作为回调函数的参数传入

大致流程图如下所示



``` 从上图中可以看出，promise模式是一个周而复始执行resolve或者reject过程，每一轮必须执行二者之一，必然导致一组回调函数出队。如果抛出异常，但这组回调函数中没有异常处理函数errorHandler，那么这组回调函数便作废，直到找到下一个能捕获异常的回调函数。直到队列中回调函数全部出队。看来我们有必要写一个“直到找到我们需要的函数”的函数：

```

getCallbackByType: function (type) {
  if (callbacks.length) {

    var callback = callbacks.shift()[type];

    while (!callback) {
      callback = callbacks.shift()[type];
    }
  }
}
  
```

```

    return callback;
}

```

从上图中可以看出所有的promise可以共用一个callbacks队列，并且考虑到需要判断返回值是否为promise类型，我们最好还需要一个标志位，做如下修改：

```

var callbacks = [];

function Promise() {
    this.isPromise = true;
}

...

```

根据以上描述，这类似于一个递归的过程

```

promise --> resolve/reject ---> promise ---> resolve/reject

```

注意在上面Level 1的FirstSucHandler中，新返回的promise执行了reject，这会自动使队列一组回调函数出队并执行。但面对一些没有返回值的情况应该怎么办，那么就应该遵循我上面说的标准，要么执行上一个promise，要么默认执行下一个resolve：

```

...
executeInLoop: function (promise, result) {
    // 1. 如果回调队列还没有被清空
    // 2. 或者没有返回值,
    // 3. 或者有返回值但不是promise
    if ((promise && !promise.isPromise || !promise) && callbacks.length)
    {
        // 默认执行resolve
        var callback = this.getCallbackByType("resolve");

        if (callback) {
            var promise = callback(promise? promise: result);
            this.executeInLoop(promise, promise? promise: result);
        }
    }
},
...

```

最后Complete函数也要做相应修改：

```

...
complete: function (type, result) {

    var callback = this.getCallbackByType(type);

    if (callback) {
        var promise = callback(result);
        this.executeInLoop(promise, promise? promise: result);
    }
},
...

```

最后贴上完整版代码：

```

var callbacks = [];

function Promise() {
    this.isPromise = true;
}
Promise.prototype = {
    resolve: function (result) {
        this.complete("resolve", result);
    },
    reject: function (result) {
        var _this = this;
        setTimeout(function () {
            _this.complete("reject", result);
        });
    },
    executeInLoop: function (promise, result) {
        // 如果队列里还有函数 并且( 要么 没有返回一个值 或者 ( 有返回值但不是promise类型 )
        )
        if ((promise && !promise.isPromise || !promise) && callbacks.length) {
            var callback = this.getCallbackByType("resolve");
            if (callback) {
                var promise = callback(promise? promise: result);
                this.executeInLoop(promise, promise? promise: result);
            }
        }
    },
    getCallbackByType: function (type) {
        if (callbacks.length) {
            var callback = callbacks.shift()[type];
            while (!callback) {
                callback = callbacks.shift()[type];
            }
        }
        return callback;
    },
    complete: function (type, result) {

```

```

var callback = this.getCallbackByType(type);
if (callback) {
  var promise = callback(result);
  /*
  1. 有返回值, promise类型
  2. 有返回值, 其他类型
  3. 无返回值
  */
  this.executeInLoop(promise, promise? promise: result);
}
},
then: function (successHandler, failedHandler) {
  callbacks.push({
    resolve: successHandler,
    reject: failedHandler
  });
  return this;
}
}

```

并附上执行结果：

```

First successHandler recevied: some data
Third failedHandler catch: Error {}
Fourth successHandler recevied: Error {}

```

参考文献

- [Promises/A+ - understanding the spec through implementation](#)
- [Promise patterns](#)
- [You're Missing the Point of Promises](#)
- [Creating Responsive Applications Using jQuery Deferred and Promises](#)
- [Asynchronous Programming in JavaScript with “Promises”](#)
- [Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics.](#)
- [tiny Promise.js](#)

作者简介

李光毅，新晋前端工程师，现就职于爱奇艺，热于前端技术分享。联系邮箱：juststayinvegas@gmail.com

原文链接：<http://www.infoq.com/cn/articles/promise-a-misunderstanding-and-practical>

新品推荐 | Product

Ionic HTML5移动框架发布Alpha预览版

作者 [Burke Holland](#) , 译者 [孙镜涛](#)

Ionic是一个新的、可以使用HTML5构建混合移动应用的用户界面框架，它自称是“本地与HTML5的结合”。该框架提供了很多基本的移动用户界面范例，例如像列表（lists）、标签页栏（tab bars）和触发开关（toggle switches）这样的简单条目。它还提供了更加复杂的可视化布局示例，例如在下面显示内容的滑出式菜单。

原文链接：<http://www.infoq.com/cn/news/2013/12/ionic-HTML5-mobile-alpha-preview>

Reactive Extensions for C++简介

作者 [Jonathan Allen](#) , 译者 [姚琪琳](#)

Reactive Extensions for C++（也叫Rx.cpp），已经在WinRT（C++/CX）和OS X（clang）中使用了。尽管还很年轻，但很多工作已经在上一个预览版中完成了。

原文链接：<http://www.infoq.com/cn/news/2013/12/rx-cpp>

ASP.NET Identity 2.0预览版增加帐号确认、密码重置和安全令牌提供服务

作者 [Anand Narayanaswamy](#) , 译者 [孙镜涛](#)

Microsoft最近发布了ASP.NET Identity 2.0预览版，新版本支持帐号确认、密码重置和安全令牌提供服务，能够对UsersStore和RolesStore执行IQueryable，另外还修复了一些bug。

原文链接：<http://www.infoq.com/cn/news/2013/12/asp-net-identity-preview-2>

Google眼镜开发工具箱允许开发者使用Xamarin.Android构建Google眼镜应用

作者 [Anand Narayanaswamy](#) , 译者 [孙镜涛](#)

Google最近在开发者事件上发布了Google眼镜开发工具箱（Glass Developer Kit，简称GDK），借助于该工具箱开发者能够使用C#和Xamarin.Android构建运行在Google眼镜上的本地应用程序。根据官方所提供的信息，开发者将能够很容易地找到该工具箱中所包含的工具和功能，因为很多API已经在现在的Android应用中使用了。

原文链接：<http://www.infoq.com/cn/news/2013/12/glass-developer-kit>

Elastic Mesos服务实现EC2中集群自动化部署

作者 [Charles Menguy](#)，译者 [孙镜涛](#)

EC2用户现在能够自动化部署Apache Mesos了，后者是一个能够在多个数据处理框架之间共享集群资源的开源工具，可以通过大数据创业公司Mesosphere所提供的一个称为Elastic Mesos的新Web服务实现规模化。

原文链接：<http://www.infoq.com/cn/news/2013/12/elastic-mesos>

推特开源CocoaSPDY

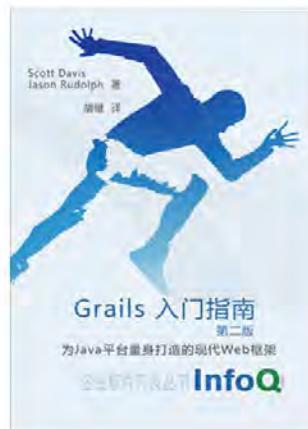
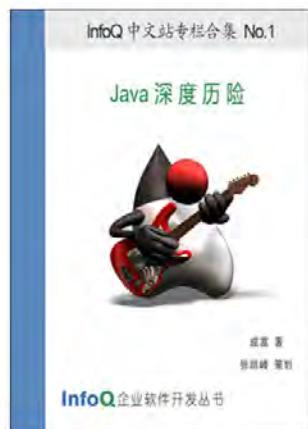
作者 [Abel Avram](#)，译者 [马德奎](#)

推特开源了其开发的CocoaSPDY，这是一个面向OS X（Cocoa）和iOS（Cocoa Touch）的SPDY框架，基于他们先前对Netty的贡献，同时，他们更新了其iOS应用程序，使用SPDY代替了纯HTTP。推特已经注意到，通信延迟降低了多达30%，当“用户的网络状况变得更糟”时，改善效果更明显。

原文链接：<http://www.infoq.com/cn/news/2013/12/cocoa-spdy>

InfoQ 软件开发丛书

欢迎免费下载



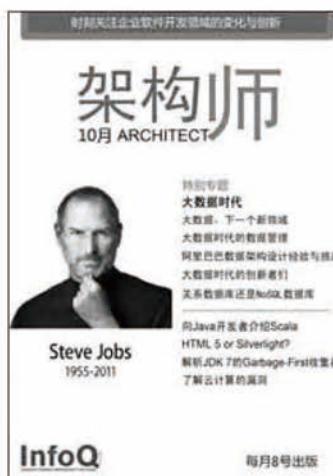
商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com

架构师

www.infoq.com/cn/architect

每月8号出版



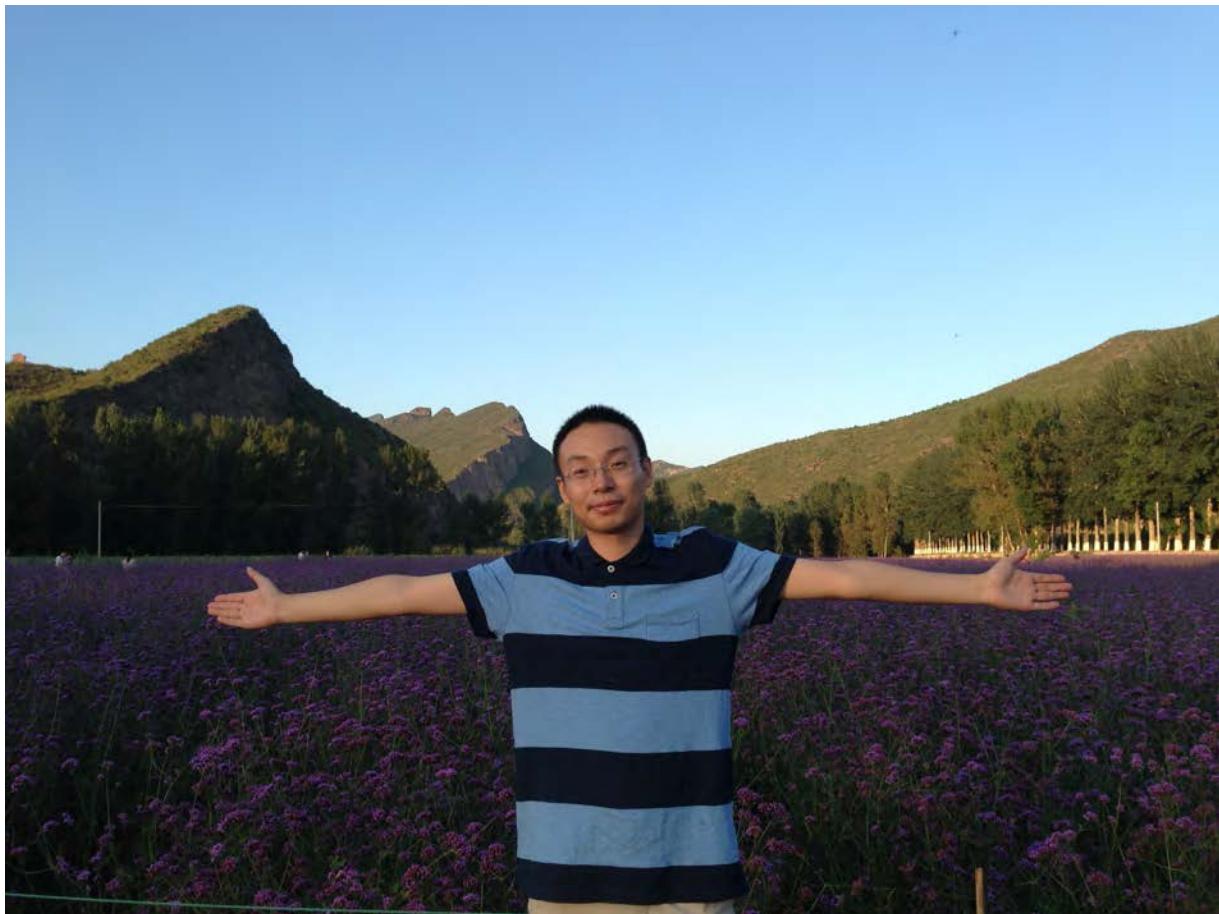
封面植物

橡树



橡树，壳斗科；印度橡皮树，桑科，大型常绿乔木。原产于北印度、马来西亚及印尼一带，现在世界各地均有种植。印度橡皮树生长可高至20米，在热带森林，有些更可以高达30米。由於印度橡皮树适宜种植於良好、潮湿的泥土及有蔽护的地方，所以经常被种植於公园及花圃中作为填补空隙之用。橡树是世上最大的开花植物；生命周期比较长，它有高寿达400岁的。果实是坚果，一端毛茸茸的，另一头光溜溜的，好看，也好吃，是松鼠等好玩动物的上等食品。 橡树形优美，树冠塔形，高可达24米，生长中速。冠幅10米，叶型独特，新叶亮红色，成熟叶片深绿色，有光泽，9月变成橙红色，落叶期晚。 印度橡树生长可高至20米，在热带森林，有些更可以高达30米。橡树顶生的叶芽被尖锐、粉红色的托叶所遮盖，当叶扩展时便会脱落。全株植物含有白色黏性的乳汁。托叶和乳汁两者都是所有榕属树木，包括印度橡树的辨别特征。此外，印度橡树有长而扩展的枝条，和由树干及大枝条垂下的气根，这些都是它的辨别特征。 橡树是世上最大的开花植物；生命周期比较长，它有高寿达400岁的。他们都具有共同特点，叶子比手掌还大，也像手儿那样，伸出几根粗壮的手指；果实是坚果，一端毛茸茸的，另一头光溜溜的，好看，也好吃，是松鼠等好玩动物的上等食品。

推荐编辑 | 唐巧



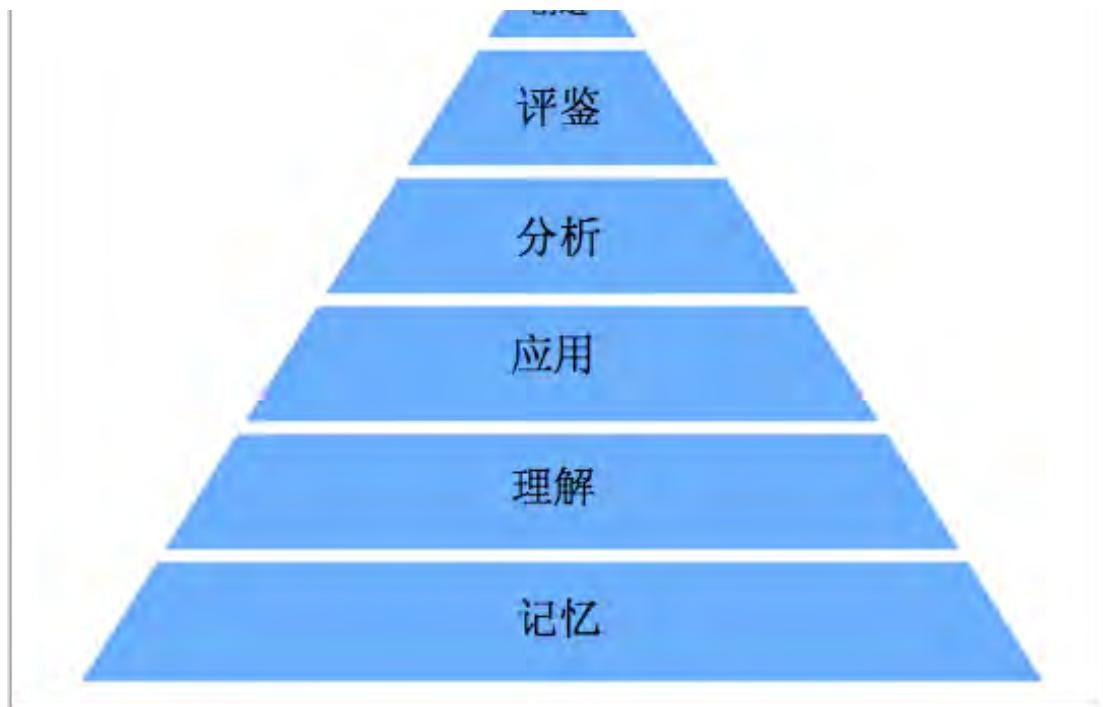
作为码农，我们为什么要写作？

在程序员这个行业，坚持做技术写作的人一直比较少。我和身边的朋友沟通后，发现他们除了借口没有时间外，大多没有意识到写作带来的收益。在他们看来，将自己学到的知识简单记录下来就足够自己需要的时候回顾了。而技术写作通常需要花更多时间，因为需要将技术的细节以及来龙去脉讲清楚。不得不承认，这的确是一个事实，通常情况下，把一个知识讲清楚比理解它更难。那我们为什么要花时间写作呢？我想写作至少有以下好处。

提高自己对知识的掌握层次

美国教育心理学家[Bloom](#)将知识认知分为了两个维度，其中认知历程维度又分为6个层次，分别为：记忆、理解、应用、分析、评鉴、创造。如下图所示，层次越高，表示对知识的掌握程度越深。





对于写作者来说，在写作过程中，因为需要对知识进行精确地表述，常常要对知识的细节再次的探索。在这个过程中，写作者可能会发现自己的观点不清晰的地方，通过二次学习，使自己的理解更加完善。写作者也可能会发现自己观点中的错误，从而改正自己的曲解。在经历过这段过程后，通常对于自己所写的知识的掌握程度，都上升了一个层次。我自己的每次技术写作都经历了这样的提高过程。所以，我更多时候是把写作当成学习的一种方式。这种学习方式比普通的学习方式更加深入，效果更好。当然，花费的时间也更多。

提高表达和沟通的能力

作为一个程序员，日常工作大部分时间都是面对电脑。许多人周末也喜欢当一个技术宅，待在家里上网、看电影或者玩游戏来消遣。长时间的面对机器，使得我们的语言表达能力极度衰退。而写作是一个很好的机会，让我们练习自己的表达能力。长时间写作之后，你会更加注意平时沟通的语言。你的用词更加精准，表达更加生动。在表达能力提高的同时，你的沟通效率也得到提高。

接受读者的沟通和反馈

当你的文章通过博客或者InfoQ网站发表出来后，你就会接着获得写作的第三个好处：来自读者的沟通和反馈。一篇好的文章通常会吸引一些读者回复，通过和读者的交流，你可以收获以下好处：

1. 错误内容反馈：尽管文章在写作时经历过二次学习，但是人难免会犯错。写作将你的思想完全暴露出来，有水平的读者可以指出你文章中的错误，从而使你对知识的理解更加准确。我的很多博客文章都有一些细微错误，通过读者的找反馈，我很快就将错误内容改正过来了，自己的水平也得到

了提高。

2. 认识朋友：一个乐于分享的人总会比沉默寡言的人更招人喜欢。所以通过写作，你可以结交很多和你一样，乐于分享的朋友。
3. 了解更多相关信息：一些读者会回复说：“某某框架也用了这个技术方案”，或者是：“你的这个实现方案没有另一个某某开源方案好”。这些信息，作为你当前文章知识点的补充，使你能够了解更多相关的资料，再一次完善自己所学的知识。

影响力

当你持续的写作，坚持一年以上，你就会慢慢收获影响力。这个时候，你也会收到技术大会的分享邀请，出版社的约稿邀请，著名互联网公司的工作邀请，甚至是创业项目的合伙人邀请。你相比那些不分享的人，获得了更多的机会。当然你的技术观点也会被更多人接受，你也会收获到传递知识的乐趣。

结束语

在写作过程中，你将收获提高自己对知识的掌握层次和提高表达和沟通的能力的好处。在写作结束后，你将收获错误内容反馈、认识朋友和了解更多相关信息的好处。在坚持写作一段时间，你将收获影响力和传递知识的乐趣。看了写作的这么多好处，你是否心动？那赶快创建一个博客，开始你的技术写作之旅吧！

● 想成为技术**牛人**? ● 想获得技术**干货**? ● 想结识技术**圈内朋友**?

百度技术沙龙

与技术大咖一起，讨论时下技术热点



牛人



大咖



圈内朋友



热点



干货

百度技术沙龙第46期

读图时代的识图技术

01月18日，北京 车库咖啡



salon.baidu-tech.com

畅想 • 交流 • 争鸣 • 聚会



百度技术沙龙是由百度主办，InfoQ负责策划、组织、实施的线下技术交流活动，每月一期，每期一个主题，由2场演讲以及Open Space开放讨论环节组成。旨在为中高端技术人员提供一个自由的技术交流和分享的平台。每期沙龙会邀请1名百度讲师分享百度在特定技术领域的成果及实践经验，同时还会邀请1名优秀的互联网公司或企业技术负责人对同一话题进行分享。活动主要面向开发者、技术负责人、项目经理、架构师等IT技术人员。

新浪微博：
[infoqchina](#)

Bai
度

InfoQ

1kg 多背一公斤
.org

爱自然 | 更爱孩子





架构师 1 月刊

每月8日出刊

本期主编：杨赛

美术/流程编辑：水羽哲

总编辑：霍泰稳

发行人：霍泰稳

读者反馈：editors@cn.infoq.com

投稿：editors@cn.infoq.com

商务合作：sales@cn.infoq.com

InfoQ 中文站：[新浪微博](#)



本期主编：杨赛

杨赛 (@lazycai)，InfoQ高级策划编辑。写过一点Flash和前端，现在只是个伪码农。在51CTO创办了《Linux运维趋势》电子杂志，偶尔也自己折腾系统。曾混迹于英联邦国家，学过物理，做过一些游戏汉化，练过点长拳，玩过足球、篮球、羽毛球等各类运动和若干乐器。喜欢读《失控》。



《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。