促进软件开发领域知识与创新的传播

架构师

6月 ARCHITECT

特别专题

安全

负载增长时悄然袭来的42个怪兽问题

详细分析Java中断机制

下层可扩展性解决方案——聚合集合

林仕鼎谈数据中心计算

整体大于部分之和

存储资源分层

学术界与工业界，以及未来展望

不容小觑的Web安全

敏捷项目中的安全需求管理

移动设备攻击和防护

InfoQ Enterprise Software Development Community

每月8号出版

InfoQ

促进软件开发领域知识与创新的传播

中文 | 英文 | 日文 | 葡文 | ……

内容投稿 editors@cn.infoq.com | 商务合作 sales@cn.infoq.com | 加入我们 hr@cn.infoq.com

《架构师》6 月刊

每月 8 日发刊

本期主编

杨赛

总编辑

霍泰稳

美术/流程编辑

水羽哲

发行人

霍泰稳

读者反馈

editors@cn.infoq.com

商务合作

Sales@cn.infoq.com

15810407783

卷首语

数据与历史

司马迁写史，记录帝王、诸侯、高官、以及刺客等传奇人物的故事。各朝各代都有太史局，帝王的生平，大到出生、登基、颁布政策、发起战争，小到每天早朝谈论何事、起居饮食吃了什么，都有记录；大臣的言行，也有记录。史官写史，会参考这些档案，也会从一些口耳相传的渠道采集信息。如果一个人读遍来自各个渠道的历史记载，根据不同史官的记录进行对比、考证，就能够还原一个最贴近真实的历史轨迹。而读史读得通透之人，会获得一种预知的能力——因为人类在很多方面是重复性的动物，古人在历史上出于某种原因做了某些事情，今人往往也会延续同样的轨迹去产生行动。

社交网络的诞生，让每个人都有机会记录自己每天的零碎琐事。不仅如此，我们现在的档案规格要远远高于帝王们：我们不仅有文字，还有照片、视频、录音，像 Nike+这样的应用还会自动关注我们每天走了几步路，消耗了多少能量，体重减了几斤。如果现在存储的这些信息能够保留到一千年后，那么千年后的史官研究历史，就可以知道组成我们这个时代的所有普通人的各种数据。对一个历史事件，由于能够追述到各个方面的记录和评论，所以也能够更加真实的还原。

根据一些调研机构的预测，接下来几年，社会上产生的数据量每过一年都会增长 10 倍；在未来 5 年期间，全人类产生的总数据量将会超过人类在过去几千年所产生的数据量总和。在科研领域，这个事情已经成为现实；现在，数据采集的触角将深入全人类社会的各个角落，作为忠实的档案记录者与人类共存。

从选择性的数据记录到无差别的数据记录，这是量变引起质变。这变化背后的推动力，是大量廉价、可移动的设备，是越来越智能的软件，是开发者。

现在，再没有任何一个人类能够读完所有关于这个时代的档案，我们必须将阅读档案的工作交给机器，让机器成为我们这个时代的史官。机器能够理解人类的历史到怎样的程度？新时代的史官能够比我们更好的预知未来吗？

身为软件开发者，我们不一定想过要改变世界的事情。但是，人类的历史的确因我们而改变。

本期主编：杨赛

Powered by InfoQ

QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

中文 | 英文 | 日文 | 葡文 | .....

内容投稿 editors@cn.infoq.com | 商务合作 sales@cn.infoq.com | 加入我们 hr@cn.infoq.com

JavaOne

MAKE THE FUTURE JAVA

中国 • 上海

CHINA SHANGHAI

07/22~07/25

立即报名

更可享受QCon上海6折特惠

JavaOne 2013

2013年7月22-25日

上海世博中心

购票享

半价!

优惠只在6月14日前

精彩看点:

- JCP.next：复兴中的Java标准
- 性能实践：理解你的应用程序的真实性能
- NetBeans IDE 7.3中的新特性
- 透过Nashorn项目看服务器端JavaScript
- 用Java开发Web游戏中的经验教训
- 高效的Scala
- 基于MySQL 5.6版本的数据库开发运维
- 通往Lambda之路
- 为什么需要切换到Java SE 7？
- Taobao GCIH项目：直接分享Java对象，提醒垃圾收集性能
- 用HTML5解禁Java EE平台
- 用JavaFX构建一个平台独立的交互式电子书编译器
- 打开那扇隐藏的门：随处部署JavaFX
- 当Groovy、Clojure、Scala和JavaFX在一起
- 高效JavaScript单元测试
- 转向客户端：JavaFX和HTML5
- 迁移Spring应用到Java EE 6
- Java EE 7：生产效率高速提升和HTML5
- Java PaaS和Oracle Java云服务
- 淘宝的鹰眼分布式日志系统
- 便于移动支付的新型Java Card平台
- 用于嵌入式设备的简易中间件
- 基于Raspberry Pi的Java时间

目录

卷首语 3

人物专访 | INTERVIEW 8

Justin James 谈高效开发者的特质 8

热点新闻 | NEWS 11

程序员越老，就越博学吗? 11

Node v1.0 路线图 15

处理 JSON 的 Java API 标准发布 21

利用事件简化系统架构 25

Facebook Buck 和 xctool：针对 Android 和 iOS 的开源构建工具 27

推荐文章 | ARTICLES 32

负载增长时悄然袭来的 42 个怪兽问题 32

详细分析 Java 中断机制 41

下层可扩展性解决方案——聚合集合 50

C#性能优化实践 55

本期专栏 | TOPIC 65

阅读者（二十四）：不容小觑的 Web 安全 65

敏捷项目中的安全需求管理 69

移动设备攻击和防护 76

特别专栏 | COLUMN 81

林仕鼎谈数据中心计算（一）：整体大于部分之和 81

林仕鼎谈数据中心计算（二）：存储资源分层 87

林仕鼎谈数据中心计算（三）：学术界与工业界，以及未来展望 90

新品推荐 | PRODUCT 99

Blossom 准备将开发语言切换为 Dart 99

ScriptCS：将 C#转变为脚本语言 99

Go 1.1 正式发布，性能显著改善、语言细微变化 99

JSON.NET 5 中的架构变更 99

Clang 完整支持 C++11 标准 100

Microsoft 企业库 6.0 增加语义日志 100

Dart 编译器在浏览器中编译 Dart 代码 100

Visual Studio 2012 Update 3 预览版发布 100

Android Studio：Google 发布基于 IntelliJ IDEA 的新 IDE 101

Strongloop Node：第一个企业级的 Node.js 发布版本 101

AIDE 2.0 引入对原生 C/C++应用的支持 101

10gen 发布 MongoDB 增量备份服务 101

最新的 Dart VM 在 DeltaBlue 基准测试中击败 JVM 102

CloudMunch 发布了一体化 DevOps 平台 102

Appium 支持测试原生、混合及 Web Android 应用 102

推荐编辑 | 翻译团队编辑 李彬.....103

封面植物 105

百合.....105

人物专访 | Interview

Justin James 谈高效开发者的特质

作者 崔康

Justin James 是 Conigent 公司的首席架构师，他指出要想成为一名优秀的开发者，仅仅写出可用的代码是远远不够的，还需要具备一些综合的素质，包括学习技术和非技术两方面的新技能、务实但不教条主义、知道如何寻找答案等等。

高效的开发人员能够主动学习技术和非技术两方面的新知识。糟糕的程序员只知道了解对自己工作必要的东西。但是优秀的开发者会主动学习新的技术技能，不仅是技术方面，他们还会学习非技术技能，对各种知识来源都有一种开放的心态，而不会像有些人那样固步自封。具体而言，糟糕的程序员可能在参加了 WPF 有关的项目之后才会开始学习 XAML，但是出色的开发者也许在一年之前就已经了解了相关的知识，因为他们觉得这些新知识很有趣，而且他们还会阅读 WPF 应用的设计指南、可用性的相关内容，因此能够做出优秀的界面来。

优秀开发者必须遵循实用主义，而不是教条主义。严格的遵守编程规则对许多人来说非常可怕，难以坚持下来。James 指出，他经常能够碰到一些程序员，他们无法或者拒绝做某个任务只是因为完成这个任务的做法通常不符合最佳实践所。业务需求很少会受到实现需求所采用的技术的制约。没有人会说，“我们不应该把这个需求写到规格说明书里，因为要实现这个需求，程序员就不得不写一段很糟糕的代码。”程序员的任务是要创造一个可以工作的应用程序，而绝不是要求在技术方面达到十全十美。这不是在为垃圾代码做辩护。James 认为，总会在有些时候，你会写出一些不那么优雅的代码，但是这些代码永远不会作为范例向别人展示。如果只有一种写法，那么这种代码就不是糟糕的代码，但要保证你已考虑了其它所有可能的方案。

James 认为，“知道如何寻找答案”很重要。这项技能不仅仅是在搜索引擎中键入几个关键字那么简单， 也不是到 Stack Overflow 或者 MSDN 论坛上类似网站发求助帖。James 就碰到过在搜索引擎里根本搜不到答案的问题，然后在 Stack Overflow 和 MSDN 论坛发的所有问题贴也没有一个像样的答案，不过他最终还是解决了所碰到的问题。有些问题是与情境（上下文环境）相关的，所以如果依赖于搜索引擎和论坛就会被淹没在各种乱七八糟的答案中。要知道如何进行根本

原因分析，利用底层系统方面的知识找到其它的线索和解决方案，还要学习在深入分析问题之前要对问题本身有一个全面的高层次认识。

对工作有热情，如果不热爱本职工作，那么你无法成为顶级的开发者。现实世界的确存在一些非常优秀的、为了工作而工作的开发者，James 曾经也是这样的心态，但是如果这成为你的价值观，那么你不会乐意做促进事业成功的许多事情。这种观点会激怒许多开发者，因为他们认为这是一种对自己的侮辱："我是一名出色的开发者，但是我有其他优先级更高的事情要做，我不能把工作变成生活。"James 表示完全理解，他也有许多优先级高的事情，但是当他投身于工作当中时，他愿意暂时搁置其他优先的事情来完成工作。如果你不能乐意全身心的投入到工作中，那么你无法成为最优秀的开发者，这不是一种侮辱，而是事实。

程序员的热情不仅仅是编程，而且要热爱工作、使用的技术、你的老板、项目等等。James 发现有些出色的开发者在平时工作中表现一般，因为这些人讨厌手头的项目或者讨厌使用的技术。James 也有过类似的心态，他的同事也有这样的人。James 建议，如果你处于这样的情形下，那么需要立刻解决它，要么从工作中找到乐趣，要么换份工作。

许多开发人员都有很强的自尊心。你比其他人更聪明、更有知识或者更有经验并不意味着你比其他人更出色。你需要平等对待他人，认真聆听并考虑其他人的想法，不要小看他们。你应该更在意团队是否成功，而不是个人的成就。

高效开发者应该具备创业精神。最好的程序员不是机器人，他们对产品有自己的情感归属和创业精神。对他们来说，产品成功不仅仅意味着工资的回报。因为他们对工作有情感寄托，他们为了项目成功而工作，而且走得更远。

"该出手时就出手"，James 认为开发人员经常犯的一个错误是一头扎进代码堆里，但是又不知道应该做什么，更糟糕的是，这些开发者还将其行为称之为敏捷模式，好像这么说就会显得不错。对于出色的开发者而言，之所以扎进代码里，是因为产品的规格说明对他们来说非常熟悉和有经验。当高效开发者面临新挑战时，他们会认真思考、规划和研究。他们的优点在于，不会掉入"分析瘫痪"的泥沼里爬不出来，他们知道应该对某些要素保持关注。一旦分析太多，那么就是浪费时间。但是也有例外，比如原子弹项目、宇宙飞船计划等。

在某一点上停止计划开始编码，然后根据实际情况调整自己的计划。这也正是 James 喜欢敏捷模式的原因。高效的开发者如果发现项目不再适用于项目实际情况，那么就会做出调整甚至抛弃。

综上所述，James 的观点主要包括：

- 学习技术和非技术两方面的新技能
- 务实但不教条主义
- 知道如何寻找答案，而不是依赖于搜索引擎
- 热爱工作相关的一切，而不是编程本身
- 尊重同事，不要自负
- 对本职工作存在情感归属，而不仅仅是赚钱的途径
- 减少重复的分析工作，及时启动编码工作，随时调整计划

原文链接：http://www.infoq.com/cn/news/2013/05/effective-programmer

相关内容

- Justin James 谈高效开发者的特质
- 增进敏捷团队中测试和开发人员间的协作
- 架构师（4 月刊）
- 物理墙和虚拟墙之争
- 阅读者（二十三）——《精益和敏捷开发大型应用指南》
- 你的软件开发项目中使用敏捷方法了吗？

热点新闻 | News

程序员越老，就越博学吗?

作者 Abel Avram ，译者 廖煜嵘

最近一个基于 Stack Overflow 数据的研究尝试去解答以下问题：编程知识是否与年龄有关，程序员是否越老就越博学以及他们是否能够掌握新的知识技能等。

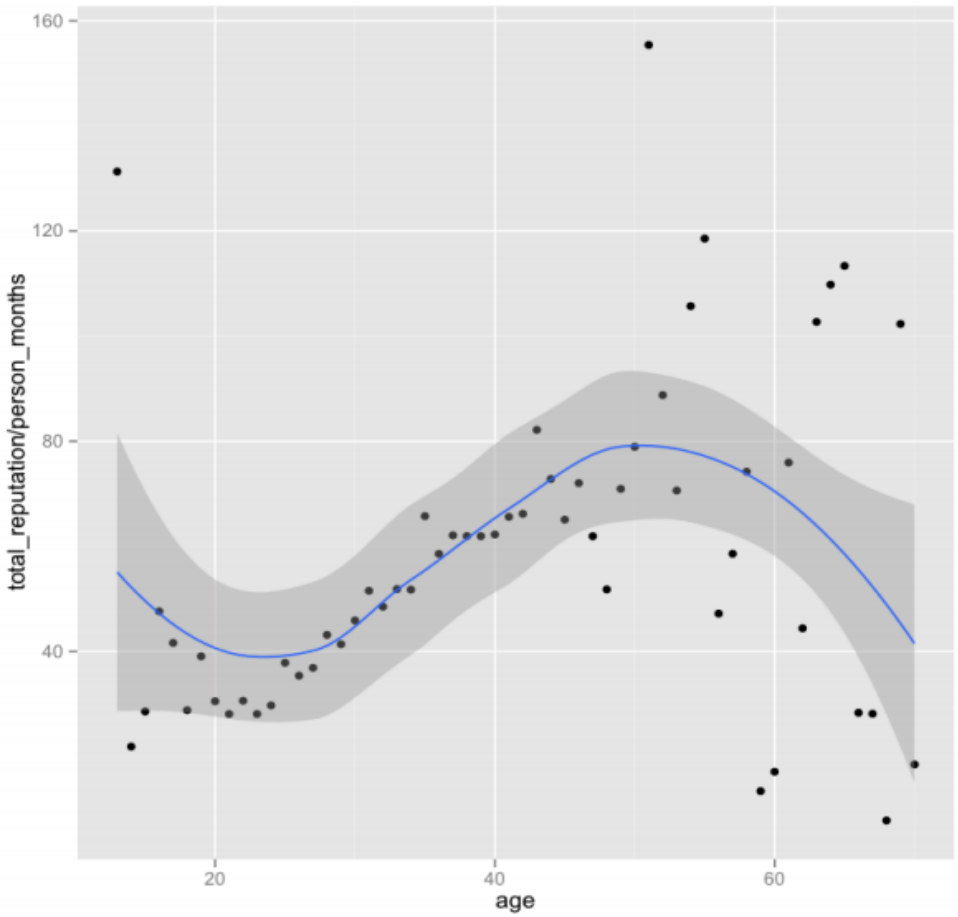
美国北卡罗来纳州立大学计算机科学系的博士生 Patrick Morrison 和助理教授 Emerson Murphy-Hill 最近发布了《编程知识是否和年龄有关？Stack Overflow 探秘》（PDF），研究了编程知识和年龄之间的关系。

这项研究的对象是 Stack Overflow 的用户数据库（2013 年 2 月）,包含 1 694 981 名程序员，平均年龄为 30.3 岁。在这些用户中，超过 300 000 名用户明确提供了具体年龄，这些数据均用于本项研究。在该用户群中，作者筛选出曾在 2012 年回答过问题并且其威望值在 1 到 1 000 000 之间的用户，总样本数为 84 284 个，平均年龄为 29.02 岁。作者认为，在该调查中回答问题比提问更具代表性，因为“很可能问一个好问题却无法回答，但不大可能有难以理解或错误的答案获得高分”，并且“回答编程问题的能力在程序员面试中更重要。”

作者在其调查中着重提出了三个问题，主要是关于编程知识方面，而不是编程能力方面：

年龄对编程知识起到积极作用吗？

在该调查中，编程知识是用 Stack Overflow(SO)的威望值来衡量的，结果显示年龄在 10 到 50 岁之间，威望值大致呈线性增长，意味着“年龄和 SO 威望值间正相关”，如下图所示：



程序员越老其技术和技能就越多样化？

作者通过选用 Stack Overflow 标签的方式，来衡量调查对象的知识领域，比如 Java、C#、Javascript、C 等。结果数据（如下图）显示“平均每个程序员所拥有的标签数量随着年龄增长都会有所下降，在 30 岁左右触底，然后在 40 到 50 岁之间反弹，而在 60 岁左右呈分散状。”这么来看，程序员越老似乎就拥有越多的技能集。

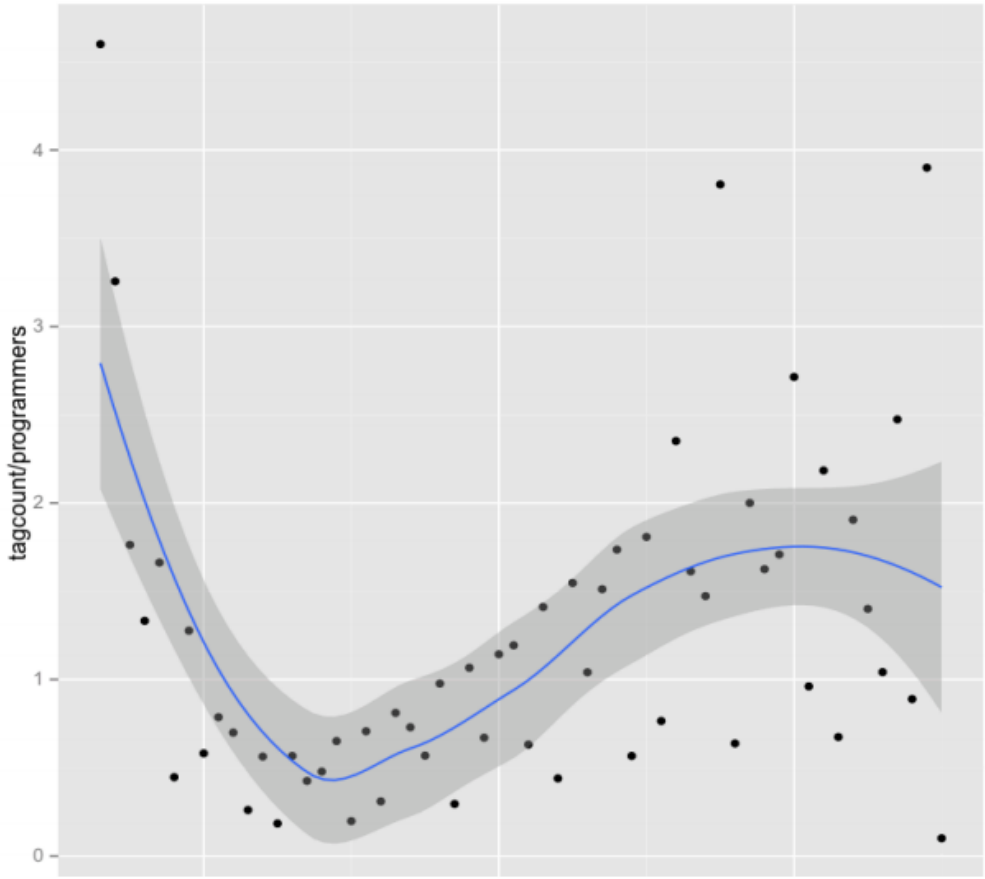


Figure 3: Unique tags by age

大龄程序员学习新的技术能达到什么程度？

为了衡量大龄程序员是否依然能掌握新的知识，作者将调查对象的标签限定在一些新的技术——clojure、 django、 git、 ios、 jquery、 linq、 mongodb、 ruby-on-rails、 silverlight 和 windows-phone-7-，并结合答案的分数：

我们推测，他们在新技术领域获取的知识，可以通过答案的分数来衡量。每个答案的分数基于支持和反对的数量。如果越老程序员学习越新的技术，我们期望跟他们在回答这些技术的问题时，能够获得和年轻程序员一样甚至更高的分数。

结果数据在下表中列出：

Table1: Ten 'new' technologies and younger/older statistical comparisons

tag	t-value	p-value	mean-younger	sd-younger	n-younger	mean-older	sd-older	n-older
clojure	-0.85	0.802	3.25	3.58	539	2.95	2.88	84
django	0.02	0.49	0.91	2.01	4763	0.91	1.72	470
git	0.32	0.62	1.82	3.44	2852	1.78	2.23	426
ios	4.3	8.87E-06	0.96	3.19	19627	1.25	2.9	2119
jquery	0.36	0.358	0.78	2.02	28546	0.79	1.96	3016
linq	0.71	0.24	1.26	2.28	2366	1.34	2.24	431
mongodb	-0.47	0.68	0.95	1.69	2093	0.91	1.38	252
ruby-on-rails	0.52	0.3	0.89	2.45	11089	0.93	2.22	1061
silverlight	0.55	0.29	0.63	1.23	1487	0.69	1.48	235
windows-phone-7	3.41	0.0004	0.68	1.48	2158	1.16	1.68	153

作者总结道：

> “iOS”和“windows-phone-7”这两个标签，显著偏离了统计学中的零假设。这可能表明即使技术的年代久远，但其知识依然有用武之地，比如“iOS”以 20 年历史的 Objective-C 为基础，就给大龄程序员带来了优势。年龄和所选新技术之间的关联性相对较弱，我们还没有强有力的证据证明大龄程序员无法掌握新技术。这样看来，大龄程序员的确能掌握新的技术。

作者承认，Stack Overflow 可能不是最好的样本去衡量编程知识和年龄的关系，因为“美国对程序员就业的调查数据显示，与 SO 的用户分布相比，专业程序员的年龄分布倾向大龄化”，并且“也许是年轻的程序员把加入【SO】看作理所当然的事，而大龄程序员如果知道自己在某方面特别在行也会加入 SO。”他们也“不确信 RQ3【第三个研究问题】所采取的调查手段是公平的，但是我们到目前为止也没有找到更好的方法。”

原文链接：http://www.infoq.com/cn/news/2013/05/programming-age-study

相关内容

- 程序员越老，就越博学吗?
- 利用精益创业从失败中学习
- Sea.js 作者玉伯：如何向开源社区提问题
- 百度技术沙龙第 34 期回顾：机器学习之多媒体方向的思考（含资料下载）
- .NET 工具和实践调查结果
- 社群驱动的调研！InfoQ 的新服务

热点新闻 | News

Node v1.0 路线图

作者 田永强

在上周（5.9）于 San Francisco, CA，由 Node.js++俱乐部举办的分享会议上，Isaac Schlueter（Node 现今的 Gatekeeper）介绍了 Node v1.0 的详细路线图，并于 Youtube 上传了分享视频。Isaac Schlueter 在本次分享中介绍了 Node1.0 发布前 Node 还将经历的增强和改动。这个经历 4 年左右发展的技术，一出世就吸引了太多太多的目光，惊人的社区活力，相信它的大版本会得到更多人的关注。

尽管 Isaac Schlueter 在 Node v0.10 发布时就对未来的计划通过博客进行过说明，但没有本次分享详细和正式，那让我们就内容先睹为快吧。

Node v0.10

Node v0.10 的发布，带来了以下这些改进，并随着 v0.10.x 的发布，将持续在这几个方面带来增强：

- 主要引入了新的 Stream API（Stream2）
- API 的改进。包括 domains, nextTick, IdleGC 等
- 持续集成。目前 Node 通过 jenkins 搭建了持续继承的环境，每天为新的改动执行单元测试，以检测对多个平台的影响
- 一些企业场景的支持
- 依然成指数增长的社区模块

这里 Stream2 主要方向是内部重构和向下兼容方面的工作，不再重点详述。

npm-www 的计划

npm-www 是 Node 官方的第三方公共模块平台，目前上面有 3 万多个第三方模块，在数量和开发速度上都呈现出社区的活跃度。但是这里隐藏的问题是第三方模块存在良莠不齐的质量，需要更科学的管理，和更良好的社区生态环境。Isaac Schlueter 提到将会通过以下三个方面入手：

- 集成 Github。集成 Github 上的项目仓库，和集成 Github 的登录。npm 上的模块大多托管在 Github 上，集成 Github 将两个社区联系得更紧密
- 推荐引擎。目前寻找目标模块的方式还比较原始，主要是通过 Node 的模块列表 Wiki、通过 npm-www 搜索、通过 Github 等。对于社区的发展还需要更好的推荐方式，如何让优秀的模块让需要它的人更快的找到，是故，推荐引擎不可少。
- 模块排行。Isaac Schlueter 计划引入 CPAN 社区中的“Kwalitee”风格来让模块进行排序。

“Kwalitee”是一个拟声词，发音与“quality”相同。CPAN 社区对它原始的解释如下：

> "Kwalitee" is something that looks like quality, sounds like quality, but is not quite quality.

总体意思就是模块的质量不是那么容易确认，只能从一些层面对它进行考察，但即使考察都通过，也并不能意味着它就是高质量的模块，所以存在“Kwalitee”这么一个新发明的词。这个方法能排除大部分不合格的模块，虽不精确，但是有效。总体而言，符合“Kwalitee”的模块应该是满足如下的条件：

- 具有良好测试
- 具有良好文档（README、API）
- 具备良好的测试覆盖率
- 具备良好的编码规范
- etc.

对 API 采用 Deprecate 标记， 而不是暴力修改 API

没有人料到 Node 目前已经这么成熟，随着 Node 的成长，刻意破坏 API 兼容性的日子结束了。假如发现 Node 中有些 API 不合理，也不会直接暴力删除了，而是采用温和的 deprecate 标记。Node 团队保证只要你的应用现在能跑起来，不用任何修改过了一年以后仍然能跑起来。

Node v0.12

对于 v0.12，可以将它看成是 1.0 的候选版本。这个版本之后，就是我们翘首企盼了 4 年的 1.0 版本。这个版本主要的工作几种在 4 个方面：

- 重构 TLS 模块。主要是在性能方面
- 清理 HTTP 模块
- 提升 Cluster 模块的负载均衡能力
- 处理掉 Buffers 中性能慢的点

Buffers

Buffers 是这个版本要重点攻克的地方，因为 Buffers 在 Node 中有着广泛的应用，任何 Buffers 带来的性能提升都会让 Node 受益。现在的难点在于 Persistent 和 MakeWeak 的 GC 方式太慢。

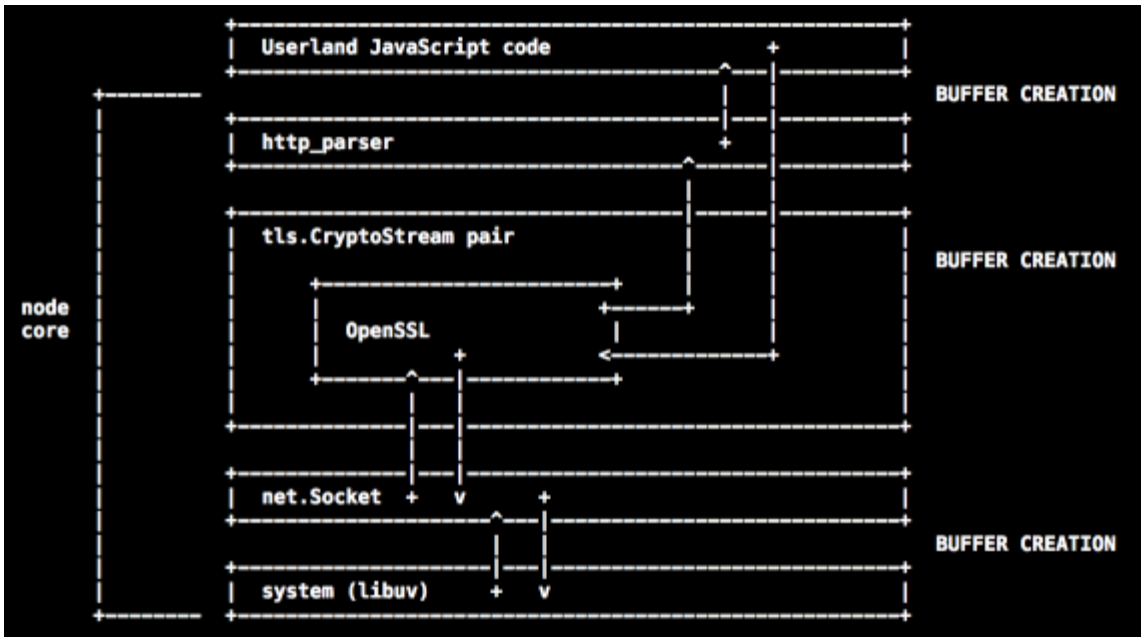
目前调研了两种方法来解决这个问题：

- ThinBuffers。一种用户分配/销毁的的 Buffers 对象
- 让 Buffer 成为 V8 的原生对象，视图从 V8 的层面来改进

不久之前，Trevor Norris 加入了 Node 核心开发团队。Trevor Norris 来自 Mozilla 公司，之前为 Node 贡献的开发工作主要在 node.cc 中，围绕 process.nextTick, node::MakeCallback 和 Domains 部分。他的加入主要是致力让 Buffers 更快。让我们翘首以待。

TLS

TLS 模块是 Node 典型的弱点。由于涉及安全，在系统中存在更多的层次。完成一次安全传输，TSL 是先要经过 libuv（C），进入 TCP 层（JavaScript），处理的结果立刻扔给 OpenSSL（C），OpenSSL 干完活后又扔出 Buffer，给后面的 http_parser，处理之后才扔出 Buffer 给用户层（JavaScript）。如图所示：



我们可以看到这这过程中有大量的 JavaScript 与 C 之间的对象调用，中间创建了大量无意义的 Buffer。所以将会对 TLS 中的 tls.CryptoStraeam 进行重构。重构后这一层，以下的代码全部用 C 完成，这以上的代码全用 JavaScript 完成，这样只要创建一次 Buffer 就可以了。

将来 tls.CryptoStream 会为了向下兼容而存在，不会被默认启用。

http

http 模块目前一个文件中既包含服务端也包含客户端代码，主要的目的是试图共用一些代码，但这并不是一个好主意。目前已经着手分离它，以使代码更容易管理。同时 http 模块与 TLS 有相同的问题，就是创建了太多无意义的 Buffers 对象，这也是待改进的点。

对于 http 客户端的连接池，Node 默认设置的连接数量是 5，这个数字可能是当时拍脑袋决定的，该版本将会改进它。

另外 http 的客户端实现只在队列中有请求时，才让底层 TCP 保持连接，如果队列中没有请求，就会关掉它，即是你立即又发起请求。这个版本的改进将使它真正能保持连接。

cluster

cluster 模块使得 Node 能够实现单机的进程集群，但它目前在负载均衡上的表现欠佳，总有一个或两个 worker 进程干了太多活。这使得你需要集群好好工作的

地方，却在某些情况下引起问题。接下来的解决方案是将会采用 round-robin 调度来使集群负载均衡良好。

其他地方

- 子进程同步执行 execSync
- 流对象写入的 writev 支持（将多个缓冲区中的数据一次写出，以提升性能）
- V8 API 改变带来的某些改变
- 更好的 IPv6 支持
- 特殊的 DNS 支持
- Bug 修复等

Node v1.0

Node v1.0 将是 v0.12 之后的下一个稳定版，目前的 API 在 1.0 中都会有效，但是内部会有小的改动。v0.12 版本之后不会新的功能特性开发计划，从 0.12 到 1.0 之间不会有大体上的区别。

总结

总体来说，这个路线图主要反映如下这几个方向： - 持续性的稳定系数提升 - 速度提升，这可能得益于 V8 的升级，或者新的 ECMAScript 特性等 - Node 现在已经相当稳定，可以放心使用。 - 不会有奇怪的事情发生 - Node 绝不会出现 Perl6/Python3（不会出现语法变动，不向下兼容的版本）

另外 Node v1.0 之后会带来稳定和持续性的发展势能，这主要体现在： - Node 国度中，对核心代码的改动影响将会减缓 - 稳定的基础是最好的培养社区创新的最好方式 - 仍然可能发生变化，虽然是稳定的立场相悖，但会做出权衡 - 疯狂而高产，快速成长的社区 - 涌现更多用 Node 的产品将势不可挡

参考

http://j.mp/2013-road-to-node-10

http://cpants.cpanauthors.org/kwalitee

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/news/2013/05/node_v1_0

相关内容

- Strongloop Node：第一个企业级的 Node.js 发布版本
- Node v1.0 路线图
- 使用 HTML5 开发体感游戏——VeloMaze 的开发简介
- 超低数据传输量，服务器端动态扩展——Node.js 开源游戏框架升级至 0.3 版
- Node.js 包含新的流 API
- 沃尔玛实验室开源项目一览

热点新闻 | News

处理 JSON 的 Java API 标准发布

作者 Bienvenido David III ，译者 孙镜涛

作为 JSON 处理（JSON-P）的 Java API，JSR-353 已经到达了最终的批准投票阶段。JSON-P（类似于 JAXP）由一个流 API（类似于 StAX）和一个对象模型 API（类似于 DOM）组成。它的参考实现是 jsonp，目前正处在测试阶段，它基于 CDDL v1.1 和 GPL v2 协议开放源码。JSON-P 将作为即将发布的 Java EE 7 的一部分，同时 JAX-RS 会使用它作为默认的 JSON 实现。请注意，该 API 并没有包含 JSON 数据绑定（类似于 JAXB），同时它和更流行的 JSONP（JSON with padding）并没有关系。

JSON（JavaScript 对象标记）是一个轻量级数据交换格式，被广泛用于 Web 应用程序、REST 服务和 NoSQL 数据库。随着 JSON 的流行以及 Java 平台上诸多 JSON 类库（org.json、Jackson 及 google-gson 等）的出现，现在已经到了需要对开发者创建并使用 JSON 的方式进行标准化的阶段。JSON-P 就是即将到来的标准，它能够被独立地使用，或者作为 Java EE 7 容器的一部分。

JSON-P 分为两个 API，流 API（javax.json.stream）和对象模型 API（javax.json）。流 API 是一种底层且高效的解析和生成 JSON 的方式。它由两个主要的抽象构成——JsonParser 和 JsonGenerator。JsonParser 是一个拉模式解析器，允许以向前、只读的方式访问 JSON 输入源。JsonGenerator 则提供了向流中写入 JSON 的方法，并且允许使用方法链。Generator 输出 JSON 对象中的名/值对以及 JSON 数组中的值。

下面是 JsonParser 和 JsonGenerator 的代码示例。首先是我们将会读取或者创建的 JSON 数据。

```
[
 {
  "type" : "home",
  "number" : "(800) 111-1111"
 },
 {
  "type" : "cell",
  "number" : "(800) 222-2222"
 }
]
```

下面是 JsonParser 示例，示例的底部是它的输出。

```
JsonParserFactory factory = Json.createParserFactory(null);
JsonParser parser = factory.createParser(new StringReader(json));

while (parser.hasNext()) {
  Event event = parser.next();

  switch (event) {
    case KEY_NAME: {
      System.out.print(parser.getString() + "="); break;
    }
    case VALUE_STRING: {
      System.out.println(parser.getString()); break;
    }
  }
}

type=home
number=(800) 111-1111
type=cell
number=(800) 222-2222
```

下面是 JsonGenerator 的示例，它向 System.out 中打印 JSON。

```
JsonGeneratorFactory factory = Json.createGeneratorFactory(null);
JsonGenerator generator = factory.createGenerator(System.out);

generator.writeStartArray().
  writeStartObject().
    write("type", "home").
    write("number", "(800) 111-1111").writeEnd().
  writeStartObject().
    write("type", "cell").
    write("number", "(800) 222-2222").writeEnd().
  writeEnd().close();
```

第二部分是对象模型 API，一个简单易用的高层 API，它基于流 API 实现。它创建了一个类似于树的结构代表内存中的 JSON 数据，能够非常容易地进行操纵

和查询。对象模型 API 中最主要的抽象是 JsonObject 和 JsonArray，这两部分都是不可变的。JsonObject 提供了一个 Map 视图，用于访问模型中未被排序的名/值对集合。JsonArray 提供了一个列表视图访问排序的值序列。为了创建这些对象模型，你可以使用构建者模式（JsonObjectBuilder 和 JsonArrayBuilder）或者使用 JsonReader 从一个输入源（InputStream 或者 Reader）中读取它们。然后你可以使用 JsonWriter 将这些对象模型写入一个输出源（OutputStream 或者 Writer）。

下面是使用 JsonArrayBuilder 创建 JsonArray 的示例。需要再次提醒的是，我们使用的依然是上面的 JSON 数据。

```
JsonBuilderFactory factory = Json.createBuilderFactory(null);
JsonArray jsonArray = factory.createArrayBuilder()
    .add(factory.createObjectBuilder().
        add("type", "home").
        add("number", "(800) 111-1111"))
    .add(factory.createObjectBuilder().
        add("type", "cell").
        add("number", "(800) 222-2222")).build();
```

下面是 JSONReader 示例。

```
try (JsonReader jsonReader = Json.createReader(new StringReader(json))) {
  JsonArray array = jsonReader.readArray();
  System.out.println(array);
}
```

下面是 JsonWriter 示例。

```
try (JsonWriter jsonWriter = Json.createWriter(System.out)) {
  jsonWriter.writeArray(jsonArray);
}
```

如果你想尝试这个测试版本，那么可以下载 jsonp，或者使用 javax.json:javax.json-api:1.0-b06 和 org.glassfish:javax.json:1.0-b06 Maven 产品。你需要使用 Java SE 6 或者更高版本。如果想要获取更多信息，可以访问

官方网站提供的 JSON 处理的 Java API ，或者阅读 JSON 处理的 Javadocs 。你还可以通过 YouTube 观看 JavaOne 提供的 JSON 处理的 Java API 呈现。

原文链接：http://www.infoq.com/cn/news/2013/05/standard-java-api-for-json

相关内容

- 处理 JSON 的 Java API 标准发布
- 详解 Java 7 中新的文件 API
- Oracle java EE 7 规划丢弃对云的支持
- 深入了解 Java 社区进程
- JSR-107（JCache）正在制定当中，将成为 Java EE 7 的一部分
- Java 7 Hotspot 循环 Bug 详解

热点新闻 | News

利用事件简化系统架构

作者 Jan Stenberg ，译者 臧秀涛

在小型业务组件之间使用事件进行交互可以简化系统架构，上周 Russ Miles 在探讨“通过事件简化架构（Architectural Simplicity through Events）”时谈到了这一点。

简洁真的非常重要，Barbara Liskov 在 QCon London 2013 的主题演讲中这样说。资深顾问 Russ Miles 对此深表赞同。他的经验是，团队的开发速度经常因所编写代码体积和复杂性的增长而放缓。他认为简化架构是帮助团队保持开发势头并加快开发过程的最佳方式。

对 Russ 而言，组件结构在简化架构方面非常重要。将组件放置在传统的分层模型中往往会把事情弄得一团糟。因此他创建了 Life-Preserver 模型（形成了一个环），所有的基础设施软件都在环上处理集成，而核心业务组件在环内加入业务价值。他指出该模式与 Alistair Cockburn 创建的六边形架构（Hexagonal Architecture）风格非常相似。

事件

下一步是定义组件如何彼此合作。因为我们很可能并不是一开始就知道所有交互，因此可以通过发送事件保持系统的灵活性。组件要遵守的唯一契约是事件的接收与广播，这为我们带来了一个标准的事件驱动架构，不过该架构是组件级的。

管道与过滤器

再进一步，Russ 几乎将组件缩减成了函数，从强类型事件变为简单的数据，比如，以文档或 XML 的形式，使核心通信方式与集成组件和外部事件之间的通信方式相同。这一步使架构变为了在小型组件之间分享不可变数据，而每个组件就是负责对数据做某种处理并返回数据，这就是管道与过滤器模式，不过是在系统架构内部而已。

原文链接：
http://www.infoq.com/cn/news/2013/05/architecture-simplicity-events

相关内容

- 利用事件简化系统架构
- 设计模式必须通过面向对象来实现吗？
- Shuttle（飞梭）服务总线入门
- Twitter 开源了基于事件的组件框架 Flight
- 降低代码重构的风险
- Dagger: 一种 Android 平台的依赖注入框架

热点新闻 | News

Facebook Buck 和 xctool 针对 Android 和 iOS 的开源构建工具

作者 Abel Avram ，译者 侯伯薇

本文包含了最新发布的 Buck 和 xctool 的细节，这是两种 Facebook 内部用于本地 Android 和 iOS 应用程序的构建工具。

Buck

Facebook 最近开源了他们用于构建本地 Android 应用程序的工具 Buck。Buck 受到了 Google Blaze 的启发，创建它是为了处理与多个 Android 库有复杂关联的应用程序，从而减少构建时间。引入 Buck 之后，Facebook 开发的四种本地 Android 应用程序中使用了单一的代码树和构建工具，这让开发更简单、更流畅，错误更少。最初的 38 个库在四种应用程序之间共享了 500 个模块。使用 Buck 替换了最初基于 Ant 的系统之后，第一次针对代码树运行时，构建时间就从 3 分 40 秒降到 1 分 30 秒。

Buck 是用 Java 编写的，而构建规则最初是用 Python 定义的，后来为了更快的规则评估而采用了 Jython。工具现在被用于构建 Android Java 代码，但是团队正在努力使其支持构建纯粹的 Java 代码，而与 Android 无关，它可能会扩展到构建 Python 项目，可能还会支持其他类型的项目。

Facebook 的 Android 代码是以树的形式组织的，其中独立的子树构成了有向无环图，支持与各种不同子树相关的构建规则的并行执行，从而减少了构建时间。包含规则的构建文件分布在代码树中，每个文件针对一个已存在的 Android 模块。只有模块中的文件有变化时，构建文件才会执行，而这些构建可以并行执行。据 Buck 的创造者 Michael Bolin 所说，把并行能力设置为八个线程，会把单元测试的时间从 20 分钟降低到 4 分钟。他在 DevCon New York 2013 的话题中详细描述了构建系统。

Buck 知道如何处理文件的多个版本，正如 Bolin 在 G+的一个话题中说明的：

> 在构建 Android 应用时，另一个困难的问题是生成 R.java 文件。如果不探究太多细节，你可能会构建多个 Android 应用，它们的 Java 文件依赖于同一

> 个名为 R 的类。对于普通的 Java 类，你可能会编译一次，然后任意数量的其他 Java 类都可以依赖它。然而，在 Android 中，你需要编译的 R 的版本依赖于 Android 应用，其中编译的 Java 代码最终会被包含在内。为你构建的每种 Android 应用都编译 Java 代码的不同版本，效率会很低，所以 Buck 使用了一些聪明的技巧（为编译时而不是运行时创建不同的 R.java 文件），从而减少重复编译。

在不久的将来，团队准备增加一个后台程序，它会监控文件的编辑情况，一旦发现就会立即开始构建。

Buck 运行在 Mac OS 和 Linux 平台上，团队正在考虑把它也移植到 Windows 上。

Gerrit 的创建者 Shawn Pearce 建议使用 Buck 替换 Maven，他提到了速度以及其他几种原因：

> buck: clean: 0.452s, full 1m21.083s [*], no-op: 7.145s,
> maven: clean: 4.596s, full 2m53.776s, no-op: 59.108s,
>
> [*] 完整构建包括下载所有依赖，时间可能会因为远程服务器的性能差别而不同。

Pearce 还指出 Buck 的一些缺点：

-不支持 Windows
-没有本地的 Maven Central 支持（通过宏）
- 没有本地的 GWT、Prolog 和 WAR 支持（通过宏）
- Buck 的引导程序需要 Ant

xctool

xctool 是 Facebook 最近开源的另一种构建工具，它用于构建 iOS 应用程序。xctool 替换了 xcodebuild，具有以下特性：

- 能够作为 Xcode.app 运行相同的测试
- 构建输出和测试结果都是 JSON 格式的，使得我们不需要解析输出
- xctool 只有在发现错误的时候才打印消息，而 xcodebuild 对每个源文件都会打印。

我们想要知道为什么 Facebook 基于 xcodebuild 构建了另一种工具，所以采访了 xctool 的提交者 Fred Potter，询问他为什么这个工具更好一些：

> xctool 的最大好处在于，它可以从命令行构建和运行单元测试，这和 Xcode.app 从图形化界面上达到一样的效果。如果你为 iOS 设置了持续集成系统，那么这就非常重要了。你想要能够自动化运行测试，那些测试与你的开发人员在本地计算机上运行的完全相同，而 xcodebuild 不会用和 Xcode.app 相同的方式来构建和运行测试。在 Xcode 4 中，苹果把单元测试集成到了 Xcode 中——与“构建”和“运行”一起，有一个新的“测试”动作；使用 Xcode scheme，你可以选择启用或者禁用哪些单元测试；如果你依赖于 iOS 模拟器（也就是应用程序测试）来编写测试，那么 Xcode 会自动载入模拟器并运行测试。这些都是很大的改进，但看起来苹果并没有把这些改进融入到 xcodebuild 中，那使得自动化构建和测试非常困难。
>
> 另一个重大的问题是构建和测试失败的报告。使用 xcodebuild，你会得到大量文本输出，其中包含编译命令、编译错误和警告以及 OCUnit 的测试输出。如果你想要自动确定哪个组件编译失败，或者哪个单元测试失败，那么你就需要编写自己的正则表达式解析器，那也是我们和其他 iOS 社区中的人一直在做的工作。那会有效果，但实在很麻烦。有了 xctool，我们会让 xcodebuild 和 OCUnit 测试运行器把构建输出和测试结果作为 JSON 对象的结构化流输出。 这让我们可以很容易地以需要的形式来显示构建和测试结果。例如，我们创建了一个报表，以吸引人的、带有颜色的输出形式来显示结果（https://fpotter_public.s3.amazonaws.com/xctool-uicatalog.gif）。 还有人使用这来把测试结果输出为 JUnit XML，那在流行的 Jenkins 构建系统中会显示得很好。

所以，我们最初创建 xctool 只是为了持续集成系统，但很多开发者最后都在本地计算机上使用它。如果你想要为运行测试拥有命令行的工作流，它会非常方便。

原文链接：http://www.infoq.com/cn/news/2013/05/Facebook-buck-xctool-build

相关内容

- Facebook Buck 和 xctool：针对 Android 和 iOS 的开源构建工具
- Icenium：Doug Seven 谈针对 iOS 与 Android 构建混合移动应用
- Orubase：为 Windows Phone、Android 和 iOS 平台开发混合本地手机应用程序

- 前 Sun 开发人员为 Android ,iOS 等其他移动平台提供 JAVA 的 WORA 支持
- 移动应用平台首选 iOS 还是 Android？
- 社区讨论：Android 应用和 iOS 应用的开发成本比较

创意 Web 训练营，第三期
HTML5 与 创意实现

点击报名

2013年6月27日（星期四）：13:00-17:30

北京 清华科技园 创新大厦二层 会议厅

日程安排

时间	主题	演讲嘉宾
1:30pm - 2:00pm	注册签到	
2:00pm - 2:30pm	Behance：实现你的创意	朱建华
2:30pm - 3:15pm	交互式Web设计	马鉴
3:15pm - 4:00pm	Edge Animate 轻松制作HTML5动画	董龙飞
4:00pm - 4:10pm	中场休息10分钟	
4:10pm - 5:10pm	Type is Beautiful	石岩
5:10pm - 5:30pm	自由交流讨论	

推荐文章 | Articles

负载增长时悄然袭来的 42 个怪兽问题

作者 Todd Hoff ，译者 赵震一

解决方案可参看：抵御负载怪兽攻击，确保可伸缩性的 7 条秘诀

随着负载的增长，你精心设计的程序可能会遭遇很多突如其来的问题：系统原有的平稳将被打破，我们将对这些问题逐一考察。当然，你可以进行横向或纵向的扩展，也可以选择编写出更好的程序，让你的系统可以处理更高的负载。这样做可以节省更多的硬件开支，并让你的整个应用更加可靠，具有更快的响应时间。对于程序员来说，这必将获得非常大的满足感。

大量的对象

当系统中的对象数量增长到一定程度，我们通常将面临规模问题。随着对象数目的增长，所有类型的资源开销显然会带来很大的压力。

持续的故障会产生一个无穷事件流

在大型网络的故障场景中，我们没有时间进行系统恢复。因为我们始终处于一个持续的压力状态下。

大量的高优先级任务

举个例子，重选路由（rerouting）是个高优先级的活动。如果存在大量不能分发（shed）或取消（squelched）的重选路由任务，资源将连续不断的被消耗以支持高优先级任务。

更大的数据流

当媒体资源大小增长得更大时，系统的负载将会增长。系统负载会随着请求来源数目的增长而增长。

功能变更

在系统原有设计之外添加更多的功能后，系统中的漏洞也随之暴露。

客户端的增长

更多的客户端意味着更多的资源消耗。需要设置线程为客户端推送事件；客户端队列会消耗内存；通信会消耗带宽；需要为每个客户端维持一份独立数据。

不够好的设计决策

以下是一些会加剧规模问题的设计问题。

- 设计未考虑处理大量对象。
- 缺乏端到端应用级别的流控。
- 应用级别的重试将导致资源再分配和消息发送。
- 内存中代码库的开销。
- 并不真正可靠的发布。
- 特定数据结构过度占用内存。
- 消息协议没有处理所有故障场景。
- 使用磁盘作为存储。
- 磁盘同步没有使用块复制。
- 认为应用层协议会更有好。
- 依靠大功耗的 CPU 来提升功能负载。
- 操作系统不支持进程架构。（译者注：该点讨论见原文链接处的评论）
- 缺乏对于某些敏感操作（甚至是单条消息的删除）的硬件支持。
- 特别的网络问题，比如 ARP 数据包，当网络负载增长时就会出现丢失。

无效的假设

你所做的大部分假设都是无效的，比如你需要使用多少内存，某些任务需要运行多少时间，设置多少超时时间才算合理，一次会消费多少资源，可能发生什么样的故障，系统在不同点的延迟，你的队列需要多大，等等。

内存不足

内存使用基线的增长和内存使用峰值的增长，都会导致内存不足。

CPU 饥饿

随着对象操作的增加，将需要占用更长的时间，这是因为这些操作需要操作更多的对象。可用 CPU 将变得更少从而导致其他的操作将面临 CPU 饥饿。系统一处出现了饥饿，就会传播到其他各处。这样一来就没有足够的 CPU 来处理那些需要去完成的必要任务。导致这样的原因可能是任务的基数过高或某些情况下很多高优先级的任务需要完成。

原始资源使用率增长

更多的对象将占用更多的内存。如果某人想要支持 1000 个并发对象，很可能是做不到的，因为你可能根本就没有足够的内存。

隐式资源使用率增长

大多数功能中，针对“原始”资源使用开销中的每项资源，都将会需要大量的额外资源。如果你将一个对象存储在两个不同的列表里，你的对象数目和内存开销将是原来的两倍；队列大小可能也需要向上调整；磁盘的数目需要增加；将数据复制到从机的时间在增长；将数据加载到应用中的时间在增长；为了处理这些工作 CPU 的使用率在增长；启动时间也在增长。

事件的叠加

很多系统面临的一个新的现实是无穷工作流。Web 服务器和应用服务器服务着非常大的用户群，这是一个真实的可预计的关于新工作的无穷流。而工作将永不结束。一周 7 天 24 小时都会有请求进来。以 100%的 CPU 使用率进行工作可以很容易使服务器达到饱和。

习惯上我们将 100%的 CPU 使用率视为一个不良的信号。作为补偿，我们将创建复杂的基础设施来对工作进行负载均衡，复制状态并做好主机集群。

CPU 永不疲倦，所以你可能会认为我们应该尝试使用尽可能多的 CPU。

在其他领域我们试图通过最大限度的利用资源来提升生产率。

而在服务器的世界里，我们尝试通过人为的方式强制地降低 CPU 使用率从而保证一定的响应水平。该理由是因为如果我们没有更高的 CPU 可用性，我们将无法在合理的延迟时间内响应新的工作或完成现有的工作。

CPU 使用率达到 100%真的存在问题吗？我们在对系统做架构设计时，宁可使用 CPU 可用性和任务优先级作为架构设计认知上简略的一种解释依据，也不愿意先去理解下我们系统的底层工作流，从而使用这些信息再来做出明确的规划决策。难道这不正是真正的问题所在吗？

我们基于负载均衡服务器做出了拙劣的架构决策，通过主观臆想猜测了使用的线程数以及这些线程的优先级。除了以上这些，我们根本没有使用工具做过任何其他事。

扩展一个系统需要仔细地关注架构。在当前框架的应用程序中，却很少有对所辖应用是如何运行的进行说明。

延迟的增长

你所经历的延迟增长与规模增长可能是完全不同的。CPU 饥饿是该问题的主要原因。

任务优先级被证明是错误的

任务优先级方案可以有效地工作在较小负载下，但是在高负载下就会产生问题。举个典型的例子，有一套简陋的流控装置，一个高优先级任务向一个较低优先级的任务传递工作将会导致工作丢失和内存使用峰值，因为低优先级的任务的运行机会将会很少。

队列容量不足够大

大量的对象意味着可以进行更多的并发操作，这意味着队列容量将很可能需要扩充。

启动时间更久

更多的对象需要更久的时间才能将它们从磁盘加载到应用中。

同步时间更久

需要更多的时间才能将更多的对象在应用之间进行同步。

在大型配置中没有进行足够的测试

因为测试装置成本很高，所以我们实际花费在大型设置上的测试时间非常少。你在开发期间不需要接触大型系统，所以很可能你的设计一开始就不能支持大规模的场景。

操作耗时更久

如果一个操作作用于每个对象，当更多的对象被添加进来后，该操作的耗时也将会更久。当数据表变得更大时，过去针对一定数据量足够快速的查询，如今的耗时也会大幅度的增长。

更多的随机故障

在正常操作中你可能不会看到某些故障。但在一定规模下，响应将丢失，ARP 请求将丢失，文件系统可能会出现某些错误，消息可能丢失，回复也可能丢失，等等。

更大的故障窗口

规模化导致每个环节都会耗时更久，这意味着出现故障的概率就会更大。一个数据交换协议在处理少量数据集的时候会很快，这意味着它只有很小的机会遭遇重启或超时。但是更大的规模中，故障窗口将扩大从而第一之间就会遭遇到新问题。

没有提高超时设置

任何超时在较小的数据集中可以有效工作，但是当数据集逐渐增长后将不再适用。就 CPU 饥饿问题而言，你所编写的代码可能还没来得及跑，超时时间却早已达到。

没有增加重试次数

没有办法在确定故障之前为某应用指定重试次数，因为他们没有这方面足够的信息来支持决策。每秒 4 次重试是否合理？为什么不是 20 次呢？

优先级继承

更久的持有大范围的锁将有更好的机会遭遇优先级继承问题。

消费模式的打破

在一种规模下你可以从生产者获取所有数据，但是在另一规模下你将会耗尽队列的空间或内存。举个例子：某个轮询程序在将数据传递到下一个队列之前，会一直向远程队列轮询所有的数据源。当队列中只有很少量数据项的时候该程序可以有效的运行。但是很可能因为某个功能的变更扩大了该远程队列中的数据项数量，这样一来该轮询程序将会导致某个节点内存不足。

监控器超时

100%CPU 的情况将导致监控器超时。这在小规模系统中很少发生，但是在设计不良的较大型规模系统中就会发生。

慢速的内存泄露变成快速泄露

较小规模系统中不大引人注意的一个内存泄露问题在较大规模的系统中就变得影响重大。

原本未注意到的锁问题变得引人注目

应该在适当的地方使用锁。但是如果使用不当，该问题在较小规模的系统中也许会被人忽略。因为持有锁的线程会在另一段产生问题的指令运行前释放掉它长期占用的 CPU 使用权。但是在大规模系统中将会有更多的 CPU 抢占，这意味着将会有更多的机会看到不同线程对同一数据的并发访问。

死锁的机会变大

不同的调度模式将以不同的路径运行代码，所以遭遇死锁的机会也就更大。举个例子，当 CPU 使用率很高时文件系统没有机会得到运行，而当以某种方式打破这一情形时，文件系统随即占用了 100%的 CPU 使用权却再也没有运行。

时间同步变糟

时间同步任务的优先级并不高，所以当可用的 CPU 和网络资源变得更少时，不同节点的时钟将出现偏差。

日志数据丢失

由于日志队列容量过小从而无法应对增长的负载或是因为 CPU 太忙而没有时间片给予日志记录器分发日志数据，都将可能导致日志记录器开始丢失数据。根据队列的容量和类型不同，或将导致内存不足。

定时器没有在准确的时间触发

一个繁忙的系统无法在期望的时间触发定时器，这将导致系统的其余部分出现一连串的延迟。

ARP 数据包丢失

在高负载的 CPU 或网络环境中，在主机间传送的 ARP 数据包可能会丢失。这是因为数据包被发送到了错误的网卡，一旦更新完路由表，将不会再发生这种情况。

文件描述符限制

在一个硬件上通常都会有一个固定的文件描述符数量上限。系统设计必须将所需的最大文件描述符数量限制在该上限以内。如果取用的套接字描述符超过了文件描述符的可用池，那么涉及到大量连接（ftp，com，启动，客户端等等）的设计将会产生问题。规模化将可能造成描述符需求数量的峰值。当规模增长时，描述符泄露将会耗尽可用池。

套接字缓冲限制

系统都会为每个套接字分配一定量的缓冲空间，大量的套接字可能会减少系统整体的可用内存。随着规模增长，消息丢失也开始增长。这是因为接收消息的缓冲空间数量不足从而跟不上负载的压力。这同样也和优先级相关，因为一个任务没有足够的优先级从套接字中读取数据。较低优先级的任务可能会被发送者一方某个高优先级任务的消息所淹没。

启动镜像服务限制

一个节点的启动卡同一时间可服务的限制为 X。FTP 服务器基础设施必须限制启动卡服务的数量，否则将造成该节点发生 CPU 饥饿。

消息次序混乱

你的消息系统在高负载压力下传递消息的次数可能会发生混乱，这对非幂等的操作来说将产生问题。

协议的弱点

除非小心谨慎的创建应用层协议，否则规模的增长将带来大量的问题。

连接限制

一个某种类型的中央服务器在应付十个客户端的情况下也许绰绰有余。但是当有一千客户端的时候，它将无法满足到响应时间的需求。在这种情况下，平均响应时间将根据客户端数量成线性增长，我们称该复杂度为 O(N)(“order N”)，但是若是其他更差的复杂度将会产生问题。举个例子，我们希望一个网络中的 N 个节点可以互相通信，我们可以让每个节点链接到一台中央交换服务器，这将需要 O(N)条连接线。或者我们在每两个节点之间直接建立连接，这将需要 O(N^2)条连接线（确切的数字或公式通常不重要，这只跟涉及到的 N 的最高次有关）。

分层架构

这是一个很好的总结，所以我在此处引用了它：基于分层的架构从来就不是用来构建低延迟，高吞吐量应用的。对于多层架构，究其本质是被创建用于解决昨天

的历史问题的。从客户端-服务器时代过渡到互联网时代，它是可伸缩性方面最完美的解决方案。

该问题域是关于如何扩展应用的规模以支持成百上千的用户。然而今天的我们都知道对于该问题的解决方案就是 n 层架构。在可伸缩性的维度上我们选择了通过负载均衡的表现层，事实上这的确解决了问题。然而，当今时代，问题发生了演变。这些日子里，很多行业的问题已不再是仅仅关于提升用户体验了，数据量也成为了一个问题。

多处理器性能问题

当处理器被要求在大量无关工作间切换时，原本强大的硬件缓存加速常常会失效。

原文链接：http://www.infoq.com/cn/articles/42-monster-problems

相关内容：

- 负载增长时悄然袭来的 42 个怪兽问题
- 绕开 Android 应用开发的那些“坑”
- 访谈与书摘：精通需求过程
- 2013 华东数据库技术大会亮点回顾：数据库扩展性，高效 SQL，MariaDB
- Mike Barlow 谈实时大数据分析
- 电商推荐系统迷思

推荐文章 | Articles

详细分析 Java 中断机制

作者 丁一

1. 引言

当我们点击某个杀毒软件的取消按钮来停止查杀病毒时，当我们在控制台敲入 quit 命令以结束某个后台服务时……都需要通过一个线程去取消另一个线程正在执行的任务。Java 没有提供一种安全直接的方法来停止某个线程，但是 Java 提供了中断机制。

如果对 Java 中断没有一个全面的了解，可能会误以为被中断的线程将立马退出运行，但事实并非如此。中断机制是如何工作的？捕获或检测到中断后，是抛出 InterruptedException 还是重设中断状态以及在方法中吞掉中断状态会有什么后果？Thread.stop 与中断相比又有哪些异同？什么情况下需要使用中断？本文将从以上几个方面进行描述。

2. 中断的原理

Java 中断机制是一种协作机制，也就是说通过中断并不能直接终止另一个线程，而需要被中断的线程自己处理中断。这好比是家里的父母叮嘱在外的子女要注意身体，但子女是否注意身体，怎么注意身体则完全取决于自己。

Java 中断模型也是这么简单，每个线程对象里都有一个 boolean 类型的标识(不一定就要是 Thread 类的字段，实际上也的确不是，这几个方法最终都是通过 native 方法来完成的），代表着是否有中断请求（该请求可以来自所有线程，包括被中断的线程本身）。例如，当线程 t1 想中断线程 t2，只需要在线程 t1 中将线程 t2 对象的中断标识置为 true，然后线程 2 可以选择在合适的时候处理该中断请求，甚至可以不理会该请求，就像这个线程没有被中断一样。

java.lang.Thread 类提供了几个方法来操作这个中断状态，这些方法包括：

public static booleaninterrupted	测试当前线程是否已经中断。线程的中断状态由该方法清除。换句话说，

	如果连续两次调用该方法，则第二次调用将返回 false（在第一次调用已清除了其中断状态之后，且第二次调用检验完中断状态前，当前线程再次中断的情况除外）。
public booleanisInterrupted()	测试线程是否已经中断。线程的中断状态不受该方法的影响。
public void interrupt()	中断线程。

其中，interrupt 方法是唯一能将中断状态设置为 true 的方法。静态方法 interrupted 会将当前线程的中断状态清除，但这个方法的命名极不直观，很容易造成误解，需要特别注意。

上面的例子中，线程 t1 通过调用 interrupt 方法将线程 t2 的中断状态置为 true，t2 可以在合适的时候调用 interrupted 或 isInterrupted 来检测状态并做相应的处理。

此外，类库中的有些类的方法也可能会调用中断，如 FutureTask 中的 cancel 方法，如果传入的参数为 true，它将会在正在运行异步任务的线程上调用 interrupt 方法，如果正在执行的异步任务中的代码没有对中断做出响应，那么 cancel 方法中的参数将不会起到什么效果；又如 ThreadPoolExecutor 中的 shutdownNow 方法会遍历线程池中的工作线程并调用线程的 interrupt 方法来中断线程，所以如果工作线程中正在执行的任务没有对中断做出响应，任务将一直执行直到正常结束。

3. 中断的处理

既然 Java 中断机制只是设置被中断线程的中断状态，那么被中断线程该做些什么？

处理时机

显然，作为一种协作机制，不会强求被中断线程一定要在某个点进行处理。实际上，被中断线程只需在合适的时候处理即可，如果没有合适的时间点，甚至可以不处理，这时候在任务处理层面，就跟没有调用中断方法一样。“合适的时候”与

线程正在处理的业务逻辑紧密相关，例如，每次迭代的时候，进入一个可能阻塞且无法中断的方法之前等，但多半不会出现在某个临界区更新另一个对象状态的时候，因为这可能会导致对象处于不一致状态。

处理时机决定着程序的效率与中断响应的灵敏性。频繁的检查中断状态可能会使程序执行效率下降，相反，检查的较少可能使中断请求得不到及时响应。如果发出中断请求之后，被中断的线程继续执行一段时间不会给系统带来灾难，那么就可以将中断处理放到方便检查中断，同时又能从一定程度上保证响应灵敏度的地方。当程序的性能指标比较关键时，可能需要建立一个测试模型来分析最佳的中断检测点，以平衡性能和响应灵敏性。

处理方式

1、 中断状态的管理

一般说来，当可能阻塞的方法声明中有抛出 InterruptedException 则暗示该方法是可中断的，如 BlockingQueue#put、BlockingQueue#take、Object#wait、Thread#sleep 等，如果程序捕获到这些可中断的阻塞方法抛出的 InterruptedException 或检测到中断后，这些中断信息该如何处理？一般有以下两个通用原则：

- 如果遇到的是可中断的阻塞方法抛出 InterruptedException，可以继续向方法调用栈的上层抛出该异常，如果是检测到中断，则可清除中断状态并抛出 InterruptedException，使当前方法也成为一个可中断的方法。
- 若有时候不太方便在方法上抛出 InterruptedException，比如要实现的某个接口中的方法签名上没有 throws InterruptedException，这时就可以捕获可中断方法的 InterruptedException 并通过 Thread.currentThread.interrupt()来重新设置中断状态。如果是检测并清除了中断状态，亦是如此。

一般的代码中，尤其是作为一个基础类库时，绝不应当吞掉中断，即捕获到 InterruptedException 后在 catch 里什么也不做，清除中断状态后又不重设中断状态也不抛出 InterruptedException 等。因为吞掉中断状态会导致方法调用栈的上层得不到这些信息。

当然，凡事总有例外的时候，当你完全清楚自己的方法会被谁调用，而调用者也不会因为中断被吞掉了而遇到麻烦，就可以这么做。

总得来说，就是要让方法调用栈的上层获知中断的发生。假设你写了一个类库，类库里有个方法 amethod，在 amethod 中检测并清除了中断状态，而没有抛出 InterruptedException，作为 amethod 的用户来说，他并不知道里面的细节，如果用户在调用 amethod 后也要使用中断来做些事情，那么在调用 amethod 之后他将永远也检测不到中断了，因为中断信息已经被 amethod 清除掉了。如果作为用户，遇到这样有问题的类库，又不能修改代码，那该怎么处理？只好在自己的类里设置一个自己的中断状态，在调用 interrupt 方法的时候，同时设置该状态，这实在是无路可走时才使用的方法。

2、 中断的响应

程序里发现中断后该怎么响应？这就得视实际情况而定了。有些程序可能一检测到中断就立马将线程终止，有些可能是退出当前执行的任务，继续执行下一个任务……作为一种协作机制，这要与中断方协商好，当调用 interrupt 会发生些什么都是事先知道的，如做一些事务回滚操作，一些清理工作，一些补偿操作等。若不确定调用某个线程的 interrupt 后该线程会做出什么样的响应，那就不应当中断该线程。

4. Thread.interrupt VS Thread.stop

Thread.stop 方法已经不推荐使用了。而在某些方面 Thread.stop 与中断机制有着相似之处。如当线程在等待内置锁或 IO 时，stop 跟 interrupt 一样，不会中止这些操作；当 catch 住 stop 导致的异常时，程序也可以继续执行，虽然 stop 本意是要停止线程，这么做会让程序行为变得更加混乱。

那么它们的区别在哪里？最重要的就是中断需要程序自己去检测然后做相应的处理，而 Thread.stop 会直接在代码执行过程中抛出 ThreadDeath 错误，这是一个 java.lang.Error 的子类。

在继续之前，先来看个小例子：

```
package com.ticmy.interrupt;
import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class TestStop {
 private static final int[] array = new int[80000];
 private static final Thread t = new Thread() {
  public void run() {
   try {
    System.out.println(sort(array));
   } catch (Error err) {
    err.printStackTrace();
   }
   System.out.println("in thread t");
  }
 };

 static {
  Random random = new Random();
  for(int i = 0; i < array.length; i++) {
   array[i] = random.nextInt(i + 1);
  }
 }

 private static int sort(int[] array) {
  for (int i = 0; i < array.length-1; i++){
   for(int j = 0 ;j < array.length - i - 1; j++){
    if(array[j] < array[j + 1]){
     int temp = array[j];
     array[j] = array[j + 1];
     array[j + 1] = temp;
    }
   }
  }
  return array[0];
 }

 public static void main(String[] args) throws Exception {
  t.start();
  TimeUnit.SECONDS.sleep(1);
  System.out.println("go to stop thread t");
  t.stop();
  System.out.println("finish main");
 }
}
```

这个例子很简单，线程 t 里面做了一个非常耗时的排序操作，排序方法中，只有简单的加、减、赋值、比较等操作，一个可能的执行结果如下：

```
go to stop thread t
java.lang.ThreadDeath
 at java.lang.Thread.stop(Thread.java:758)
 at com.ticmy.interrupt.TestStop.main(TestStop.java:44)
finish main
in thread t
```

这里 sort 方法是个非常耗时的操作，也就是说主线程休眠一秒钟后调用 stop 的时候，线程 t 还在执行 sort 方法。就是这样一个简单的方法，也会抛出错误！换一句话说，调用 stop 后，大部分 Java 字节码都有可能抛出错误，哪怕是简单的加法！

如果线程当前正持有锁，stop 之后则会释放该锁。由于此错误可能出现在很多地方，那么这就让编程人员防不胜防，极易造成对象状态的不一致。例如，对象 obj 中存放着一个范围值：最小值 low，最大值 high，且 low 不得大于 high，这种关系由锁 lock 保护，以避免并发时产生竞态条件而导致该关系失效。假设当前 low 值是 5，high 值是 10，当线程 t 获取 lock 后，将 low 值更新为了 15，此时被 stop 了，真是糟糕，如果没有捕获住 stop 导致的 Error，low 的值就为 15，high 还是 10，这导致它们之间的小于关系得不到保证，也就是对象状态被破坏了！如果在给 low 赋值的时候 catch 住 stop 导致的 Error 则可能使后面 high 变量的赋值继续，但是谁也不知道 Error 会在哪条语句抛出，如果对象状态之间的关系更复杂呢？这种方式几乎是无法维护的，太复杂了！如果是中断操作，它决计不会在执行 low 赋值的时候抛出错误，这样程序对于对象状态一致性就是可控的。

正是因为可能导致对象状态不一致，stop 才被禁用。

5. 中断的使用

通常，中断的使用场景有以下几个：

- 点击某个桌面应用中的取消按钮时；
- 某个操作超过了一定的执行时间限制需要中止时；
- 多个线程做相同的事情，只要一个线程成功其它线程都可以取消时；
- 一组线程中的一个或多个出现错误导致整组都无法继续时；
- 当一个应用或服务需要停止时。

下面来看一个具体的例子。这个例子里，本打算采用 GUI 形式，但考虑到 GUI 代码会使程序复杂化，就使用控制台来模拟下核心的逻辑。这里新建了一个磁盘文件扫描的任务，扫描某个目录下的所有文件并将文件路径打印到控制台，扫描的过程可能会很长。若需要中止该任务，只需在控制台键入 quit 并回车即可。

```
package com.ticmy.interrupt;
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStreamReader;

public class FileScanner {
 private static void listFile(File f) throws InterruptedException {
  if(f == null) {
   throw new IllegalArgumentException();
  }
  if(f.isFile()) {
   System.out.println(f);
   return;
  }
  File[] allFiles = f.listFiles();
  if(Thread.interrupted()) {
   throw new InterruptedException("文件扫描任务被中断");
  }
  for(File file : allFiles) {
   //还可以将中断检测放到这里
   listFile(file);
  }
 }

 public static String readFromConsole() {
  BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
  try {
   return reader.readLine();
  } catch (Exception e) {
   e.printStackTrace();
   return "";
  }
 }

 public static void main(String[] args) throws Exception {
  final Thread fileIteratorThread = new Thread() {
   public void run() {
    try {
     listFile(new File("c:\\"));
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
  };
  new Thread() {
   public void run() {
    while(true) {
     if("quit".equalsIgnoreCase(readFromConsole())) {
      if(fileIteratorThread.isAlive()) {
       fileIteratorThread.interrupt();
       return;
      }
     } else {
      System.out.println("输入quit退出文件扫描");
     }
    }
   }
  }.start();
  fileIteratorThread.start();
 }
}
```

在扫描文件的过程中，对于中断的检测这里采用的策略是，如果碰到的是文件就

不检测中断，是目录才检测中断，因为文件可能是非常多的，每次遇到文件都检测一次会降低程序执行效率。此外，在 fileIteratorThread 线程中，仅是捕获了 InterruptedException，没有重设中断状态也没有继续抛出异常，因为我非常清楚它的使用环境，run 方法的调用栈上层已经没有可能需要检测中断状态的方法了。

在这个程序中，输入 quit 完全可以执行 System.exit(0)操作来退出程序，但正如前面提到的，这是个 GUI 程序核心逻辑的模拟，在 GUI 中，执行 System.exit(0)会使得整个程序退出。

6. 参考资料

《Java Concurrency in Practice》

《Concurrent Programming in Java Design principles and patterns》

http://docs.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html

作者介绍

丁一(ticmy)，高级 Java 开发工程师，关注并发编程，目前在信雅达系统工程股份有限公司从事工作流引擎的研发。个人博客：http://www.ticmy.com/ 微博：http://weibo.com/freish，邮箱:ticmy@foxmail.com，欢迎通过以上方式进行技术交流。

原文链接：http://www.infoq.com/cn/articles/java-interrupt-mechanism

相关内容：

- 详细分析 Java 中断机制
- 聊聊并发（三）——JAVA 线程池的分析和使用
- Oracle 调整 Java SE 版本编号方案
- 最新修订之后，Java 脆弱依旧
- 深入理解 java 内存模型
- 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

白皮书

embarcadero

管理多数据库平台以提高性能和可用性的最佳实践

优化您的数据库与时间，实现更高性能

最大限度提升虚拟化环境下Oracle数据库性能

企业数据管理中您不能犯的7个错误

选择数据库工具的10大理由

开发多平台、多设备的原生应用

开发面向Windows、Mac、iOS的多平台本地应用

Delphi“移动”版进化史

快速开发跨平台应用实践之路

推荐文章 | Articles

下层可扩展性解决方案——聚合集合

作者 Todd Hoff ，译者 丁雪丰

没有解决方案的问题有什么好的？在《负载增长时悄然袭来的 42 个怪兽问题》一文中，我们讨论了一些问题；在这篇文章（早些时候有过一篇文章，我会重新组织一下内容）里，我们将讨论我所谓的聚合策略（aggregation strategies）。

请牢记，这些都是下层架构方面的一些建议，比如如何组织代码组件的结构，它们该如何交互。本文不会讨论大规模集群，但是会讨论你的应用程序在内部——在服务接口之下的深层——可能是什么样的。这个世界上除了事件架构之外，还有很多东西。

聚合，简单说来就是不用愚蠢的队列，我们的队列会很聪明。我们打心底里把队列视为工作内容的容器，该容器最终会呈现出整个系统是如何工作的。作为工作内容的容器，我们熟知队列里会有什么请求和数据，我们可以把这种情报作为我们巨大的优势。

合理排序工作内容

此处的关键理念是一个基本上必须要牢记于心的设计方法，有程序员将其视为一等概念——在工作创建时的方方面面里什么该做，为什么要做它，该什么时候做这三个问题的优先级。

避免级联故障

为什么合理排序工作内容如此重要？我们要避免的最不靠谱的情况就是级联故障。天真的系统没有合理排序的理念，在故障发生时，会让无用的控制层或数据层流量挤占必要的控制流量。

如果你需要向交换机发送一个请求，对路由进行重新编程以对流量做故障转移，如果此时低优先级的工作项造成了线端拥塞（head-of-line blocking），那么在不关闭整个系统的情况下，你将永远都无法获得系统的控制权。那些不了解情况

的啰嗦的程序会不断发出低优先级的控制和数据流量，让系统一直处于繁忙状态，而做的事情却一点用都没有。这就好像代码中的垃圾食品。

一个了解优先级的系统会试着在确保高优先级的工作按时完成的同时避免无用的工作。用于控制的网络和数据网络是各自独立的，因此控制消息能自由通行。你会有智能重试策略，无用工作不会占据队列，作废的消息会从队列中剔除。在网络各处都要考虑缓存，确保不会看到老的版本。当最新的消息入主之时，可以用控制消息来暂停当前的工作，这样才能处理更高优先级的工作，它们的消息会立即发送出去，而非进入等待。为了让系统更强健，有很多非常酷的事情可做。

处理无限的工作负荷

排定优先级是处理无限工作负荷的关键思想，在《负载增长时悄然袭来的 42 个怪兽问题》里就讨论过这一点。传统观念里我们认为 100% CPU 使用率是一个不好的信号。作为补偿，我们制造了复杂的基础设施对工作进行负载均衡、状态复制、构建服务器集群。CPU 是不会疲劳的，所以我们可以将其榨干。当我们讨论排定优先级时，我们也在讨论系统在全负荷下如何做出漂亮的、可预见的反应。如果我们进行了合理的条件设置，这些都不成问题。只有对于那些架构天真的软件而言，这才会成为问题。

有意识的控制

合理排序的工作内容中说到开发者应该对以下内容进行有意识的控制：

- 什么工作该做，什么工作该扔
- 处理工作的顺序
- 处理工作的任务优先级
- 给予工作的资源量：
  - CPU 时间
  - 内存
  - 队列空间
  - 磁盘
  - 网络
  - 锁

你可能会注意到典型的编程环境里，你对这些内容的控制力都不强。必须改变这一点。

找出优先级

合理排序工作内容也是很多其他领域的可扩展性解决方案里的常见做法。排定优先级需要对系统正在发生什么，以及你希望系统里正在发生什么有一个深刻的理解。优先级取决于：

- 客户
- 请求类型
- 尝试提供公平的服务
- 请求数
- 已使用的资源量
- 需要的资源量
- 合约中的截止日期

SLA

除了避免级联故障之外，将拥抱优先级作为基本概念的另一个关键原因是为了保证 SLA。

你有一个高优先级的客户，他为了获得优于普通客户的服务而付费。为了实现他们的 SLA，系统中的所有组件都必须习惯于理解优先级，同时还不能让其他客户处于饥饿状态，为他们提供周到的服务。

这和 OS 任务调度以及网络流量调度有很多共同之处。其中的思想是一样的，但开发者需要在系统里运用这些思想。

合并聚合

合并聚合就是把单独的数据和/或命令合并到一起。其理念就是按固定限定成比例地使用资源。

例如，假设一个对象有如下命令序列：

- 新建

- 更新
- 更新

这三个独立的请求能被合并到一个请求里，而非三个。要是这组命令重复 100 次，那么队列里仍然始终只有一个请求。

另一个例子是属性变更事件。独立的多个事件可以合并为一个变更事件。

想想这有多强大啊。无论发生了多少事件，队列都不会超过对象个数。

你的通信子系统需要同应用程序合作才能实现这一切。

删除聚合

删除聚合就是在任何可能的时候删除工作内容。比方说，以下述操作为例：

- 新建
- 删除

其结果是新建和删除操作能在消息发出前被删除。在发生大量新建和删除操作的重定路由或抖动输入（flapping inputs）风暴中，这能省下相当数量的资源。

批量聚合

通过定时分析（timing analysis），将许多数据打包到一起能极大地提升性能。一个总比一堆好。一个挨着一个发送操作会比把所有操作放在一个批次里发送要慢很多。

理想情况下，应用程序不应该手动来处理分批操作。打个比方，应该让框架来替你处理这些事。

变更聚合

在这个策略中，很多变更被收缩成了一个变更。这和合并聚合有所不同，在变更聚合中，某些东西改变了，我们会保持对象状态，无论它变了多少次，我们都只会当它改了一次。

我们不会记下每次变更，但是我们会把值发给客户端，告诉它有东西变了。

在一个大系统中，我们没办法保存系统中发生变更的所有东西的历史。

集成聚合

在这个策略里，只有当事件在取消前经过了一段时间后才能存在。

最常见的例子是报警，只有当它经历了一段集成期后它才真正成立。否则，我们会在硬件故障或者一阵阵的条件触发时制造出报警风暴。

在告知系统其他部分有事发生前，先让事件“等”一会儿，这能避免很多无用功。

信号集成

我们后续会再进一步讨论这个话题，但这里的思想是要考虑系统有自己的情报网，监控系统正在监控整个网络内发生的所有事情。所有部署在系统里的设施会将它们收集到的情报汇报给总部，这样就可以采取协同行动来应对各种情况发展。

这个情报系统会产生一个恒定的信号流，可以用它来控制系统其他部分的行为。聚合策略对系统中正在发生的事非常敏感。为此，你需要配置自己的情报操作，管理自己的操作，甚至设计自己的计数机关。所有的传感器都会说谎。

原文链接：http://www.infoq.com/cn/articles/aggregate-collection

相关内容：

- 下层可扩展性解决方案——聚合集合
- Node.js 获得企业开发者青睐
- 2013 华东数据库技术大会亮点回顾：数据库扩展性，高效 SQL，MariaDB
- Go 1.1 正式发布，性能显著改善、语言细微变化
- 可扩展性的那些事儿：最真实的城市模拟
- 从小型网站到超大规模网站的 MySQL 参考架构

推荐文章 | Articles

C#性能优化实践

作者 胡森

性能主要指两个方面：内存消耗和执行速度。性能优化简而言之，就是在不影响系统运行正确性的前提下，使之运行地更快，完成特定功能所需的时间更短。

本文以.NET 平台下的控件产品 MultiRow 为例，描述 C#性能优化的实践。

性能优化原则

· 理解需求

MultiRow 的一个性能需求是："百万行数据绑定下平滑滚动。"整个 MultiRow 项目的开发过程一直在考虑这个目标。

· 理解瓶颈

99%的性能消耗是由于 1%的代码造成的。大部分性能优化都是针对这 1%的瓶颈代码进行的。具体实施也就分为两步："发现瓶颈"和"消除瓶颈"。

· 切忌过度

性能优化本身是有成本的。这个成本不单单体现在做性能优化所付出的工作量，还包括为性能优化而写出复杂的代码导致额外的维护成本，比如引入新的 Bug，额外的内存开销等。性能优化常常需要在收益和成本之间做出权衡。

如何发现性能瓶颈

性能优化的第一步是发现性能瓶颈，下面是一些定位性能瓶颈的实践。

· 如何获取内存消耗

以下代码可以获取某个操作的内存消耗。

```
long start = GC.GetTotalMemory(true);
// 在这里写需要被测试内存消耗的代码，例如，创建一个GcMultiRow
var gcMulitRow1 = new GcMultiRow();
GC.Collect();
// 确保所有内存都被GC回收
GC.WaitForFullGCComplete();
long end = GC.GetTotalMemory(true);
long useMemory = end - start;
```

· 如何获取时间消耗

以下代码可以获取某个操作时间消耗。

```
System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch();
watch.Start();
for (int i = 0; i < 1000; i++)
{
 gcMultiRow1.Sort();
}
watch.Stop();
var useTime = (double)watch.ElapsedMilliseconds / 1000;
```

为了获得更加稳定的时间消耗，这里把一个操作循环执行了 1000 次，取时间消耗的平均值以排除不稳定数据。

· ANTS Performance Profiler

ANTS Performance Profiler 是款功能强大的性能检测软件。熟练使用这个工具，我们可以快速准确的定位到有性能问题的代码。这是一款收费软件，会在 IL 中加入一些钩子用来记录时间，所以在分析时，软件的执行速度会比实际运行慢一些，获得的数据也因此并不是百分之百的准确，还要结合其他技巧来分析程序的性能。

· CodeReview

CodeReview 是发现性能问题的最后手段。CodeReview 应该对产品的性能瓶颈尽可能多的关注，确保该部分逻辑执行的尽可能的快。

性能优化的方法和技巧

定位了性能问题后，解决的办法有很多。下面是一些性能优化的技巧和实践。

· 优化程序结构

在设计时就应该考虑产品结构是否可以达到性能需求。如果后期发现了性能问题，调整结构会带来非常大的开销。

例如：

GcMultiRow 要支持 100 万行数据。假设每行有 10 列的话，就需要有 1000 万个单元格，每个单元格上又有很多的属性。如果不做任何优化，大数据量时，一个 GcMultiRow 软件的内存开销会相当的大。GcMultiRow 采用的方案是使用哈希表来存储行数据：只有用户改过的行放到哈希表里，大部分没有改过的行都直接使用模板代替。这就达到了节省内存的目的。

WPF 平台和 Silverlight 平台的画法和 Winform 平台不同，是通过组合 Visual 元素的方法实现的。SpreadGrid for WPF 产品同样支持百万级的数据量，但是又不能给每个单元格都分配一个 View。所以 SpreadGrid 使用了 VirtualizingPanel 来实现画法。思路是每一个 Visual 是一个 Cell 的展示模块，可以和 Cell 的数据模块分离，这样就只需要为显示出来的 Cell 创建 Visual。当发生滚动时会有一部分 Cell 滚出屏幕，有一部分 Cell 滚入屏幕。这时，让滚出屏幕的 Cell 和 Visual 分离，然后再复用这部分 Visual 给新进入屏幕的 Cell。如此循环，就只需要几百个 Visual 就可以支持很多的 Cell。

· 缓存

缓存（Cache）是性能优化中最常用的手段，针对需要频繁的获取一些数据，同时每次获取数据需要的时间比较长的场景。如果使用了缓存的优化方法，需要特别注意缓存数据的同步：如果真实的数据发生了变化，应该及时的清除缓存数据，确保不会因为缓存而使用了错误的数据。

使用缓存的情况比较多。最简单的情况就是缓存到一个 Field 或临时变量里。

```
for (int i = 0; i < gcMultiRow.RowCount; i++)
{
// Do something;
}
```

以上代码一般情况下是没有问题的，但是，如果 GcMultiRow 的行数比较大。而 RowCount 属性的取值又比较慢的时候，就需要使用缓存来做性能优化。

```
int rowCount = gcMultiRow.RowCount;
for (int i = 0; i < rowCount; i++)
{
// Do something;
}
```

使用对象池也是一个常见的缓存方案，比使用 Field 或临时变量稍微复杂一点。例如，在 MultiRow 中，画边线，画背景，需要用到大量的 Brush 和 Pen。这些 GDI 对象每次用之前要创建，用完后要销毁。创建和销毁的过程是比较慢的。GcMultiRow 使用的方案是创建一个 GDIPool。本质上是一些 Dictionary，使用颜色做 Key。所以只有第一次取的时候需要创建，以后就直接使用以前创建好的。

以下是 GDIPool 的代码：

```
public static class GDIPool
{
 Dictionary<Color, Brush > _cacheBrush = new Dictionary<Color, Brush>();
 Dictionary<Color, Pen> _cachePen = new Dictionary<Color, Pen>();
 public static Pen GetPen(Color color)
 {
  Pen pen;
  if_cachePen.TryGetValue(color, out pen))
  {
   return pen;
  }
  pen = new Pen(color);
  _cachePen.Add(color, pen);
  return pen;
 }
}
```

· 懒构造

大多时候，对于创建需要花费较长时间的对象，往往并不是所有的场景下都需要使用。这时，使用懒构造的方法可以有效提高程序启动性能。

举例来说，对象 A 需要内部创建对象 B。对象 B 的构造时间比较长。 一般做法：

```
public class A
{
 public B _b = new B();
}
```

一般做法下，由于构造对象 A 的同时要构造对象 B，导致 A 的构造速度也变慢了。

优化做法：

```
public class A
{
 private B _b;
 public B BProperty
 {
  get
  {
   if(_b == null)
   {
    _b = new B();
   }
   return _b;
  }
 }
}
```

优化后，构造 A 的时候就不需要创建 B 对象，有效的提高了 A 的构造性能。

· 优化算法

优化算法可以有效的提高特定操作的性能。使用一种算法时应该了解算法的适用情况、最好情况和最坏情况。 以 GcMultiRow 为例，最初 MultiRow 的排序算法使用了经典的快速排序算法。这看起来是没有问题的。但是，对于表格软件，用户经常的操作是对有序表进行排序，如顺序和倒序之间切换。而经典的快速排序算法的最差情况就是基本有序的情况。所以经典快速排序算法不适合 MultiRow。

改进的快速排序算法使用了 3 个中点来代替经典快排的一个中点的算法，每次交换都是从 3 个中点中选择中间值。这样，乱序和基本有序的情况都不是这个算法的最坏情况，从而优化了性能。

· 正确的使用既有数据结构

我们现在工作的.NET framework 平台有很多现成的数据结构。我们应该了解这些数据结构，提升我们程序的性能。

例如：

1. String 的加运算符和 StringBuilder： 字符串的操作是我们经常遇到的基本操作之一。 我们经常会写这样的代码 string str = str1 + str2。当操作的字符串很少的时候，这样的操作没有问题。但是如果大量操作的时候（例如文本文件的 Save/Load， Asp.net 的 Render），这样做就会带来严重的性能问题。这时，我们就应该用 StringBuilder 来代替 string 的加操作。

2. Dictionary 和 List: Dictionary 和 List 是最常用的两种集合类。选择正确的集合类可以很大的提升程序的性能。为了做出正确的选择，我们应该对 Dictionary 和 List 的各种操作的性能比较了解。 下表中粗略的列出了两种数据结构的性能比较。

操作	List	Dictionary
索引	快	慢
Find（Contains）	慢	快
Add	快	慢
Insert	慢	快
Remove	慢	快

3. TryGetValue: 对于 Dictionary 的取值，比较直接的方法是如下代码：

```
if(_dic.ContainKey("Key")
{
 return _dic["Key"];
}
```

当需要大量取值的时候，这样的取法会带来性能问题。优化方法如下：

```
object value;
if(_dic.TryGetValue("Key", out value))
{
return value;
}
```

后一种用法要比前一种用法取值性能提高一倍。

4. 为 Dictionary 选择合适的 Key: Dictionary 的取值性能很大情况下取决于做 Key 的对象的 Equals 和 GetHashCode 两个方法的性能。如果可以的话，使用 Int 做 Key 性能最好。如果是一个自定义的 Class 做 Key 的话，最好保证以下两点：1. 不同对象的 GetHashCode 重复率低。2. GetHashCode 和 Equals 方法简单，效率高。

5. List 的 Sort 和 BinarySearch 性能很好，如果能满足功能需求，推荐直接使用。

```
List<int> list = new List<int>{3, 10, 15};

list.BinarySearch(10); // 对于存在的值，结果是1

list.BinarySearch(8); // 对于不存在的值，会使用负数表示位置，

// 如查找8时，结果是-2， 查找0结果是-1，查找100结果是-4.
```

· 通过异步提升响应时间

1. 多线程

有些操作确实需要花费比较长的时间。在处理的过程中，如果用户进行操作时失去响应，这个用户体验是很差的。使用多线程技术可以解决这个问题。例如，有一个类似 Excel 的计算引擎，在构造的时候要初始化所有的函数定义。由于函数比较多，初始化时间会比较长。这是如果用到了多线程，在工作线程中做函数定义进行的初始化，就不会影响到 UI 线程快速响应用户的其他操作了。

代码如下:

```
public CalcParser()
{
 if (_functions == null)
  {
   lock (_obtainFunctionLocker)
   {
    if (_functions == null)
    {
      System.Threading.ThreadPool.QueueUserWorkItem((s) =>
      {
       if (_functions == null)
       {
         lock (_obtainFunctionLocker)
         {
           if (_functions == null)
           {
             _functions = EnsureFunctions();
           }
           }
         }
        });
       }
      }
     }
    }
```

这里比较慢的操作就是 EnsureFunctions 函数，是在另一个线程里执行的，不会影响主线程的响应。当然，使用多线程是一个比较有难度的方案，需要充分考虑跨线程访问和死锁的问题。

2. 加延迟时间

在 GcMultiRow 实现 AutoFilter 功能的时候使用了一个类似于延迟执行的方案来提升响应速度。AutoFilter 的功能是用户在输入的过程中根据用户的输入更新筛选的结果。数据量大的时候一次筛选需要较长时间，会导致用户输入不流畅，体验不好。使用多线程虽然是个好方案，但是会增加程序的复杂度。MultiRow 的解决方案是当接收到用户的键盘输入消息的时候，并不立即出发 Filter，而是等待 0.3 秒。如果用户连续输入，会在这 0.3 秒内再次收到键盘消息，放弃上一个任务，再等 0.3 秒，直到连续 0.3 秒内没有新的键盘消息时再触发 Filter。这样就实现了比较流畅的用户体验。

3. Application.Idle 事件

在 GcMultiRow 的 Designer 里，经常要根据当前的状态刷新 ToolBar 上按钮的 Disable/Enable 状态，一次刷新需要较长的时间。这个又一次影响了用户输入的流畅性。GcMultiRow 的优化方案是通过系统的 Application.Idle 事件，仅当系统

空闲的时候处理刷新逻辑。接到这个事件时，一般都是用户已经完成了连续的输入，这时就可以从容的刷新按钮的状态了。

4. Refresh, BeginInvoke

平台本身也提供了一些异步方案。例如在 WinForm 下触发一块区域重画的时候，调用 Refresh 方法不会导致立即重画，而是设置 Invalidate 标记，触发异步的刷新。在控件开发中，这个技巧可以有效的提高产品的性能，同时简化实现复杂度。

Control.BeginInvoke 方法可以被用来触发异步的自定义行为。

· 进度条，提升用户体验

有时候，以上提到的方案都没有办法快速响应用户操作。进度条、一直转圈圈的图片、提示性文字（如"你的操作可能需要较长时间，请耐心等待"）等，都可以有效的提升用户体验，可以作为最后方案来考虑。

作者简介：

胡森，葡萄城公司软件工程师，PowerTools 产品系列项目经理。长期专注于 .NET UI 控件产品的设计和开发。致力于为广大.NET 开发者提供强大，高效，易用的控件产品。

原文链接：http://www.infoq.com/cn/articles/C-sharp-performance-optimization

相关内容：

- C#性能优化实践
- 简化 C#的不可变对象图
- C#中的异步陷阱
- ScriptCS：将 C#转变为脚本语言
- 面向 Android 上 Dalvik 运行时的 C# 编译器 dot42 简介
- C#中安全并行机制的唯一性和引用不可变性研究

QCon 全球软件开发大会

International Software Development Conference

[上海站]2013

大会时间：
2013年11月1-3日

大会地点：
上海光大会展中心

7月3日前报名即可享 7折优惠

5人团购 享更多优惠

精彩内容策划中...

票务咨询，讲师自荐/推荐，
线索提供：qcon@cn.infoq.com

★ 部分特色专题 ★

1. 移动应用案例分析
2. JavaScript生态圈
3. 游戏服务器实践
4. 金融系统设计与架构
5. 系统安全设计面面观

更多经典专题 精彩内容
敬请登录 qconshanghai.com

更多信息欢迎访问

www.qconshanghai.com

本期专栏 | Topic

阅读者（二十四）：不容小觑的 Web 安全

作者 丁雪丰

近日来互联网安全事件不断，前有 Evernote 遭黑客攻击用户数据存在泄露风险，后有 Cloudflare 被超大流量 DDoS，纽约时报称其为"欧洲电脑狂人引爆网络核战争"，再往前看段时间，CSDN 的 600 万用户明文密码事件依然历历在目……照理来说，Web 安全向来都是不容忽视的，但在问题发生之前，又有多少人真的把安全问题放在心上？

记得还在读书时，一位"少年黑客"出版了一部《黑客攻击防范秘技》，红极一时，我当时出于好奇还买来读过，要知道影视剧作品中的黑客无所不能，入侵系统拯救世界，也许很多懵懂少年会因为这个形象去学习黑客攻击技术。"黑客"一词在诞生之初其实并非贬义，我相信很多人心中也有黑客精神，就像"中国黑客传说"系列中的黑哥和 V.

去年，"道哥的黑板报"作者、前阿里巴巴安全专家吴翰清（@aullik5，人称"刺"或者"道哥"）结合自己的经验出版了一部安全领域的"实战宝典"——《白帽子讲 Web 安全》。该书前言中是这样来描述"白帽子"的：

> 黑帽子是指那些造成破坏的黑客，而白帽子则是研究安全，但不造成破坏的黑客。白帽子均以建设更安全的互联网为己任。

好一句"以建设更安全的互联网为己任"，国内著名的安全网站乌云上就有不少白帽子在发现漏洞，向厂商积极反馈。道哥除了用自己的技术为无数网名保驾护航，还用文字来传播安全知识与技术。《白帽子讲 Web 安全》一书的内容安排非常清晰：

- 世界观安全（先统一认识，站在同一起跑线上再去谈具体的内容）
- 客户端脚本安全
- 服务器端应用安全
- 安全运营（光有技术、有想法，整个公司没有安全运营的意识，产品的规划和设计不当，技术上的投入都可能打水漂）

在介绍具体的安全技术时，基本分成了两部分——攻击方法说明和防御手段介绍，所谓知己知彼方能百战不殆，只有先了解攻击的原理，攻击是如何发生的，才能有更好的防御思路。以 XSS 和 CSRF 为例，这两个词耳熟能详，但说实话，我之前对它们的了解并不深刻，还是服务端的 SQL 注入比较熟悉，书中的介绍不止由浅入深，还有点手把手的感觉，更重要的是除了网上常见的做法之外，还能有很多进阶技术。在介绍 DDoS 时，除了常见的大流量暴力攻击外，还有诸如 Slowloris 这样的攻击方式，甚至还有正则引发的血案。随之而来的则是相对应的防御技术，特定的配置与参数，各种内容娓娓道来。

书中有一个观点，是我个人非常赞同的，即将部分安全相关的功能默认集成到开发框架之中。由于开发人员的水平参差不齐，而且有时迫于项目进度的压力，可能在安全方面考虑不周，很容易造成安全漏洞。而像 XSS、CSRF 等威胁，在框架层面上提供一些辅助方法后，只要使用得当即可起到很好的保护作用。对于限制频繁访问，可以在框架中加入限流模块，通过验证码或限流页面等方式保护系统，而那些非 Web 的提供接口服务的系统，同样也能融入类似的限流功能自我保护。

说到框架，其实众多流行的开发框架都有不同程度的安全漏洞，发布安全补丁也是常有的事情，书中就介绍了 Struts2、Spring MVC、Django、jQuery 等前后端框架的多个漏洞，其实不止这些，Rails 也曝出过多个严重漏洞，GitHub 一定深有感触。而且，近日 Apache 官方宣布 Struts 1.x 正式退出历史舞台，最关键的是宣布 EOL 后，发现严重安全漏洞也不会再进行维护了，这时使用 Struts 1.x 的用户只能自己解决了。可见，选择一个有大量用户使用，有团队稳定维护的框架是很重要的。

在本书的最后部分谈到了微软的 SDL，对于互联网公司来说，可能经常需要上线新功能，修正线上问题等等，这时完整运用 SDL 也许不太合适，但是不妨参考其中一些思想和做法。此外，事先制定一些突发安全事件的应急预案也是必须的，不光是技术的，还有业务和公关的等等……

前几天，从“道哥的黑板报”上得知《白帽子讲 Web 安全》销量很好，还被评为 2012 年 10 大最佳原创技术类书籍，看来还是有很多同学对安全方面的知识非常感兴趣的，希望本书能帮助大家。

原文链接：http://www.infoq.com/cn/articles/web-security

相关内容：

- 阅读者（二十四）：不容小觑的 Web 安全
- 阅读者(二十一）:美丽的架构
- 《阅读者》第 20 期：架构模式的圣经
- 阅读者（十九）：我读《管理 3.0：培养和提升敏捷领导力》
- 阅读者（十六）：项目百态
- 阅读者(十五):透过书本了解 HTML5

esri China BEIJING

分 享 地 理 价 值

2013年Esri中国技术公开课

2013年的技术公开课也已经拉开了帷幕。全年的时间安排为2013年4月–12月，每月两场（分别是第二周和第四周的周二下午）。今年的课程采用视频录播的形式，您可以关注当日发布在gisall的最新课程，也可以随时回顾以往的课程。今年的课程在内容方面增加了行业应用、传统产品解析、IT 新技术方面的选题。通过网络投票的形式让观众来选择最想听的课程，入选课程将优先投放。

2013Esri技术公开课新举措：

1、课程内容选材更丰富
2、录制哪些课程由观众做主
3、课程网络观看更流畅
4、建立QQ群（178608911）让讲师与大家充分交流
5、向积极参与者颁发各种奖励

观众选中的课程将在
gisall网站视频频道投放
欢迎参加！

视频频道在线观看
www.gisall.com

Esri中国信息技术有限公司
电 话：010–57632000　　传 真：010–57632299

本期专栏 | Topic

敏捷项目中的安全需求管理

作者 Rohit Sethi ，译者 李竞

在软件开发初期处理安全需求是防止安全问题最经济的方式。大多数安全需求都属于非功能性需求（Non-Functional Requirements ，NFRs）。很多从业者发现，在敏捷项目中处理安全和其他 NFR 非常具有挑战性。原因有二：

- 匹配 NFR 和特性驱动的用户故事需要付出很大努力；
- 安全控制常因缺少可见度而被忽视。敏捷过程容易让团队不自觉地侧重于那些可以直观改善客户体验 的新功能开发或缺陷修复。

在本文中，我们会探讨以上两个问题。

在用户故事中处理 NFR

敏捷专家们提出过一些方法 ,用以定义用户故事驱动的开发过程中的 NFR。Mike Cohn 在他的文章中讨论了评论者们提出的一些有趣的建议。Scott Ambler 就该主题写了一系列文章 ,旨在证明并非所有的 NFR 都可以独立存在于用户故事中。而 Dean Leffingwell 可以说在这方面做了最深入的研究，他在自己的著作《敏捷软件需求》中花了整整一章探讨该主题。这些专家中大部分人都认为，NFR 大致可以分成两类：

1. 非功能性用户故事：以用户故事的形式展现的可测试功能块。这些用户故事中的参与者（Actor）可能是内部 IT 人员。例如：“作为安全分析师，我希望系统能抑制不成功的身份认证尝试，这样应用程序在面对暴力破解时不至于太过脆弱”。
2. 约束：这是个跨领域的问题，常常涉及多个用户故事。我们可以将它看作是对所有相关开发工作收取“课税”。例如：开发 Web 应用时，要求所有开发者对 Http 表单中的字段进行数据校验即为一种约束。

处理第一类 NFR 很简单：创建一系列的 NFR 用户故事，将它们加入某种工作序列，例如 Product Backlog。

NFR 约束的挑战：

然而处理第二类 NFR 则困难很多。敏捷专家们对此提出过一些解决办法，包括：

- 在特定的用户故事中添加恰当的约束作为验收标准
- 在某个中央知识库——例如，贴在墙上或者发表在 wiki 上的一份文件——中维护包含所有约束的列表。

但无论哪个方法都无法解决的问题是，如何针对特定的用户故事选择适用的约束。而且，使用 SD Elements 进行安全需求匹配的经验表明，要将这些技术限定在一定范围之内很困难。下面这个列表是标准 Web 应用需要面对的安全约束中的一部分，我们以此为例：

- 在 SQL 语句中以变量绑定的方式阻止 SQL 注入
- 对来自客户端的只读数据进行完整性验证，以防止参数篡改
- 避开存在于 HTML、HTML 属性、CSS 以及 JavaScript 中的不可信数据，从而防止跨站点脚本攻击)（XSS）
- 避免客户端 JavaScript 中基于 DOM 的 XSS
- 使用安全的算法以避免整型溢出
- 不接受外部重定向以防止自由重定向攻击
- 授权给受保护的页面，以防止权限提升攻击)
- 使用防止跨站请求伪造（CSRF）的令牌
- 验证输入
- 尽量使用正则表达式，因为它对于拒绝服务攻击有较强的抵抗力。
- 为高价值的事务实现事务身份认证
- 不要硬编码密码

该列表列举的可能只是程序员在实现用户故事时需要考虑的安全约束中的一小部分。若将需要遵循的法规标准——诸如支付卡行业数据安全标准(PCI DSS)之类——考虑在内，约束还要增加不少。如果你开始添加其他 NFR 约束，例如可访问性，约束列表会快速增长，转眼间就会超过程序员的承受范围。一旦列表变得臃肿，我们的经验是，程序员往往会将它全部忽略，仅根据自己的记忆来应用 NFR 约束。而在很多不断专业化的领域——例如应用安全——NFR 的数量不断增长，这给程序员的记忆力造成了沉重的认知负担。因此，如何使用静态分析技

术侦测代码中违背 NFR 约束之处就成了关注重点。研究表明，静态分析可以侦测出的可预防缺陷最多不超过总数的 40%，也就是说仍有大量的约束漏网，包括特定领域的安全控制，更别说在缺乏定制的情况下辨别静态分析产生的“假阳性”结果也不是那么简单。

应对 NFR 约束的挑战

为了解决 NFR 约束过多的问题，我们需要做到以下四点：

- 按优先级排序：与用户故事和缺陷一样，NFR 约束也应拥有不同的优先级，例如：相较于避开 HTML 中的不可信数据以防止持续的跨站脚本攻击，对 HTTP 响应头中的不可信数据进行编码或校验以防止 HTTP 响应拆分攻击就不那么重要。因此我们假定程序员通常没有足够的时间处理所有约束，然后我们提供一种机制为如何定义约束的相对优先级提供建议。之所以说“建议”，是因为优先级有可能随着具体情况的变化而变化，程序员需要自己作出判断——对于他们的特定代码，哪种相对优先级是适用的。你可以选用简单的优先级设置，如“高、中、低”，也可以使用数字范围，如 1 - 10。
- 过滤：使用简单的标准常常就可以为特定的用户故事移除大量的 NFR 约束。举例来说，假如你正在实现的用户故事与表现层无关，那么就可以安全地忽略大量表现层相关的约束。使用标签系统或者简单的 Excel 过滤器，就可以为特定的用户故事削减约束数量。以下是一些 Web 应用相关的安全性约束过滤器的范例：
  - 用户故事是否涉及用户输入；
  - 用户故事是否涉及保密数据，如密码、信用卡信息以及非公开的财务数据；
  - 用户故事是否返回一个全新或修正过的用户输出，例如一个网页；
  - 用户故事是否与数据库进行交互；
  - 用户故事是否会暴露或使用来自 RESTful 或 SOAP API 调用的数据；
  - 用户故事是否会访问受保护的数据（例如，不是所有用户都能查看或修改的数据）。
- 情境（Context）：程序员在写代码或定义任务单（tickets）/ 用户故事的时候更容易记起并应用约束。理想情况下，如果把约束列表内嵌

入 IDE 或任务单系统，那么程序员就可以在编码时获取相关的约束。既然现在很多 IDE 都支持内嵌的 Web 浏览器，那么使用轻量级的网站或者 SharePoint 站点就可以实现这个目标。

- 框架：框架可以大幅减少约束产生的“税收”。例如，像 Django 这样的框架自带了 CSRF 防护，这意味着程序员可以安全地忽略该约束，除非他们主动绕过这个内建的功能。根据我们的经验，多数大型的应用软件都会采用某种形式的定制框架，这些框架可以无缝处理最高优先级的约束。虽然这样的框架定制可能需要较高的先期投入，但如果你将约束看成一种基于用户故事的“税收”，那么减少约束数量就相当于永久性的降低了“税率”。

验证

虽然主动处理 NFR 很重要，但仍有必要进行验证。如果你有手动代码审查（Code Review）的经验，就可以检查用户故事中作为验收标准出现的 NFR。然而，如果仅使用手动代码审查，很多约束的验证将是非常棘手或者繁重的。静态分析工具则可以自动检查编码标准的遵守情况，其中一些产品特别关注安全方面。将你的静态分析工具与持续集成过程整合在一起，可以帮助你始终遵守编码标准。但也务必注意，不要一味依赖于代码审查。防范特定领域缺陷——如 HTTP 参数篡改——的 NFR 要求对背后的业务领域有深刻的理解。将手动验证与针对特定攻击的自动测试结合起来，可以帮助我们建立一套对于特定领域缺陷拥有更强抵抗力的系统。

可见度

难以将安全需求集成进敏捷项目的另一方面原因是安全需求通常缺乏可见度。面向客户的会议，如 Scrum 的 Sprint 评审会议），容易使开发人员产生一种固有的倾向，即实现用户故事或修复缺陷时倾向于那些能直接改善用户体验的故事或缺陷。一部分非功能属性，例如易用性和性能，对于系统的使用体验有具体的影响，因此也比较容易向客户展示。而另外一些——如安全性——却很难在 Sprint 评审会议上给客户留下很好的印象。实际上，某些安全特性，如账号封锁（Account Lockout），可能给客户体验带来相当负面的影响。大多数安全特性对系统的影响，普通用户很难察觉。因此，安全性 NFR 是不可见的，它们必须和可见度更

好的特性竞争，以夺得宝贵的开发周期。在软件开发初期，准确的说在程序员应该构建安全特性的时候，处理这些“隐形”需求的机会成本会特别高。

让安全特性可见

对于敏捷团队来说，没有“银弹”能让安全需求在开发伊始就拥有最高优先级。但有一些方法可以让安全 NFR 对客户和/或 PO 可见。

- 图形化安全用户故事：如果你在应用程序安全方面足够专业，就可以针对已知的安全弱点创建一套全面的安全控制。然后将它们与 NFR 用户故事匹配起来，并绘制简单的图形，用以说明系统中已实现的和未实现的安全用户故事数量。这个图可以放在大型可视图表中。

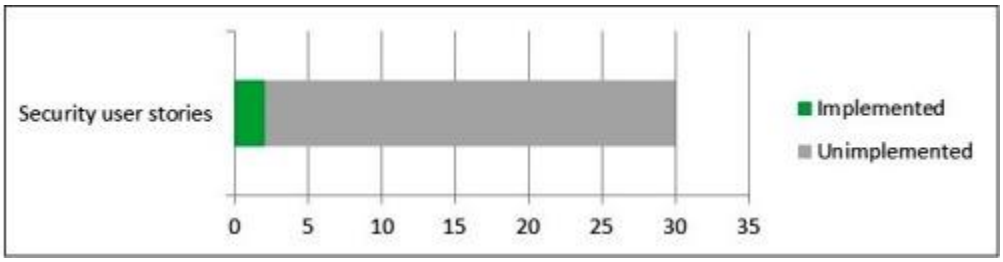


图 1：安全用户故事图

当然，这个图没必要局限于安全用户故事，你可以将其他重要的 NFR 囊括进来。实际上，你可以使用故事点（Story Point）取代用户故事数量，从而更准确的反映工作量。另外，你有可能希望图中只记录那些针对高风险安全问题的、拥有最高优先级的用户故事或控制。每完成一个故事，“已完成”数量都会增加，客户也会籍由此图直观地了解项目的进度。

- 展示可利用性（exploitability）：没有什么比可利用性展示更能让人们理解一个安全漏洞的各个方面了。这是安全性渗透测试和漏洞评估的主要区别。你可以展示在上一版本的应用中如何成功地利用了漏洞，而在新的版本中却没办法这么做了。然后你可以从对业务和 / 或客户影响的角度解释漏洞的方方面面。这非常有利于解释为什么在安全控制上花的时间是合理的。

结论

主动处理 NFR，特别是安全特性，在敏捷环境中非常重要。不少专家对于如何处理此类需求——特别是当它们与创建用户故事相关时——给出了建议。全面地处理约束是个相当大的挑战。记住本文讨论的提议，你可以极大地增加维护一系列约束的价值。即使不能完全做到以上所提的四点内容以改善对约束的管理，但从长远来看，任何一点改进都能带来很高的回报。进行验证能保证团队始终遵守 NFR。最后，提高安全特性的可见度，可以让你证明用在 NFR 上的时间是合理的，就算它们对系统的改善不能反映到用户体验上。

关于作者

Rohit Sethi（ @rksethi on Twitter）是一位安全领域的专家，他致力于将安全控制构建到整个软件开发生命周期中（SDLC）。他曾帮助一些对安全相当敏感的企业改善软件安全性，涉及的行业包括金融服务、软件、电子商务、保健以及电信等。Rohit 创建并讲授针对 Secure J2EE 开发的 SANS 课程。他曾在以下机构讲学 :FS-ISAC、RSA、OWASP、安全开发会议(Secure Development Conference)、Shmoocon、CSI National、SecTor、Infosecurity、CFI-CIRT 等等。他曾在 InfoQ、Dr. Dobb's Journal、TechTarget、Security Focus 以及 Web Application Security Consortium (WASC)上发表过文章，上过 Fox News Live，还曾被 ITWorldCanada 和 Computer World 以应用安全专家的身份引用过。另外，他还创建了 OWASP Design Patterns Security Analysis 项目。

原文链接：
http://www.infoq.com/cn/articles/managing-security-requirements-in-agile-projects

相关内容：

- 敏捷项目中的安全需求管理
- QSecurity 月度安全评论（2013 年 5 月）：谷歌眼镜隐患、IE8 漏洞、Apache 存漏洞、土豆被黑
- 阅读者（二十四）：不容小觑的 Web 安全
- QSecurity 月度安全评论（2013 年 4 月）：史上最大 DDoS 攻击、思科存漏洞、TP-Link 路由器后门、Apple ID 安全问题
- Android 4.2.2 中对安全性的改进
- RightScale 安全总监 Phil Cox 分析 IaaS 安全监控的三大问题

时刻关注软件开发领域的变化与创新

架构师

www.infoq.com/cn/architect

每月8号出版

InfoQ
Enterprise Software Development Community

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

本期专栏 | Topic

移动设备攻击和防护

作者 Charlie Miller ，译者 蔡坚安

这篇文章最初发布在 IEEE 的安全与隐私杂志上，后来 InfoQ 和 IEEE 计算机协会向您推荐了此文。

在我的青少年时期，手机只是普通用途的手机。你可以用手机呼叫别人，然后和人通话；别人也能呼叫你。当时的手机就只是这样。随着智能手机日益流行，手机的用途发生了很大变化。今天的智能手机比起以前我们所使用的，更像是计算机了。事实上，智能手机就是附加了额外硬件的计算机。具体来说，就是附加了 GMS（移动通信全球系统）无线电设备，以及控制此设备的基带处理器。这些特性非常重要，但它们在提供强大功能的同时也带来了威胁。和 PC 机一直以来的情况一样，目前智能手机正成为攻击者的目标。在这里，我要关注两款智能手机操作系统的安全模型：苹果的 iOS 以及谷歌的安卓。这两款手机在我心目中有特殊的地位，因为我是公开挖掘它们漏洞的先行者。

设备安全涉及多个方面。为了简洁明了，我会抛开加密、锁定和隐私这些主题，而把焦点放在攻击者真正想要做的事情：在你的设备上运行他们的代码。和在 PC 机世界一样，攻击者可以通过两种方式让远程代码在移动设备上运行。第一种方式是诱导用户下载、安装和运行他们的软件——这是恶意软件方式，另一种方式是利用软件漏洞来攻击设备——这是强制下载方式。我下面会探究 iOS 和安卓如何预防这两种攻击方式。

移动设备恶意软件

iOS 和安卓手机操作系统各自提供了公共软件市场——App Store 和 Android Market——但他们采取了不同的方式来限制恶意软件安装到设备上。

iOS

就像苹果的一贯风格，App Store 也是被苹果从上到下严格控制的。应用程序只有经过苹果的批准才能在 App Store 里出售。苹果采用代码签名技术在设备上施行这种措施。除非经过苹果的私人密钥签名，否则 iPhone 不会加载程序库和运行应用程序。除了苹果，没有人知道 iOS 应用程序审查有多么严密。作为一名

专业代码审查员，我晓得市场上应用程序数量巨大，苹果不可能对每个应用程序都仔细审查；但任何形式的审查都可以消灭掉最明显的恶意软件。如果某款恶意软件侥幸逃过了审查到达 App Store ,而之后被发现了 ,苹果会将它从 App Store 移除，同时将它从已安装的设备上卸载。

你也许会抱怨 App Store 对开发者不利，但它是恶意软件的有效防护墙。当然，这防护墙偶尔也有失效的时候。

在移动设备上，应用程序运行于限制它们活动范围的沙箱中。例如，一个程序不能访问另一个程序的数据。没有程序可以读取设备上存储的 SMS(短消息服务) 消息，等等。所有应用程序遵守同样的沙箱规则，它们被赋予应用程序本身该有的任何活动权限。例如，所有应用程序都能够自由地访问因特网和手机通讯录。[1]

安卓

开发者能够直接将他们的应用程序投放到 Android Market,在投放上去之前没有任何审查。安卓手机要求应用程序具有签名，但允许应用程序自我签名。谷歌只是用签名做记录，而不是想用签名控制什么样的代码才能够运行。因此，安卓手机用户能够从任何地方下载应用程序，而不只是从 Android Market。

不同于苹果采用自上而下的恶意软件防御方式 ,安卓依赖用户群体解决防御问题：让用户对应用程序评分和评论，让用户查看有多少用户下载了某款软件，让用户向谷歌举报恶意软件。如果有足够多的用户批评某款软件，谷歌会将通过将它从市场上移除，还会远程将它从移动设备上清除。对安卓用户很有帮助的一条建议是，永远不要下载没有上千下载量以及大多数正面评论的应用。另一条建议是只到 Android Market 下载应用。目前已经出现较多恶意安卓应用程序，不过大多只出现在非 Android Market 市场上。Android Market 的开放性方便了开发者的使用，但同时也方便了恶意软件开发者的入侵。

对于运行在设备上的应用，安卓同样使用了沙箱模型。然而，安卓的沙箱是针对特定应用的。在安装过程中，应用程序会通知用户它需要什么权限。用户可以同意授予权限或拒绝授予。如果用户拒绝，应用程序就不会安装。这种方式的好处是能够根据每个应用定制沙箱，而不是像苹果那样使用满足各种应用、但规格统一的沙箱。例如，俄罗斯方块游戏不需要访问因特网，所以它就不需要具有访问权限的沙箱。这种模式的弊端是让用户自己做安全决定，而历史已表明这中方式

并不好。更进一步分析，首先是用户确实想要安装应用，所以他们才会主动安装，然后他们倾向于随着屏幕一直点击下一步，授予应用要求的权限。

漏洞挖掘

当然，攻击者可以试图绕过设备安装和审查过程，然后直接操控设备。移动设备的攻击面（attack surface）和 PC 机的非常相似；最容易受到攻击的目标是像网页浏览器和邮件客户端这样的应用程序。通常，移动设备的攻击面更小些，因为可以攻击的代码相对要少。例如，你在手机浏览器中找不到 Java 和 Flash，但在桌面电脑浏览器中他们却很常见（同时也是常见的漏洞挖掘目标）。

然而，智能手机中存在两条 PC 机中没有的攻击途径。其中一条途径利用是 SMS 消息处理。Collin Mulliner 和我曾展示过怎样利用 iPhone 手机的 SMS 消息解析器漏洞来控制手机。[2] Intrepidus 组织的研究员拿 Palm Pre 手机做过类似的研究。[3] 另一条攻击途径是利用 GSM 无线电设备，它最近才被挖掘出来。Ralf - Philipp Weinmann 曾展示过怎样利用 GSM 软件的漏洞控制手机的基带处理器。[4]

iOS

知道软件存在漏洞是一回事，能够写代码利用它的漏洞是另外一回事。就拿 iOS 来说，它使用了分层安全技术以防止被恶意攻击。iOS 采用了数据执行防护（DEP）技术和地址空间布局随机化（ASLR）技术。DEP 区分了数据和代码，使得漏洞挖掘很困难。这样，攻击者不能在向进程输送数据的同时又跳到该进程去执行数据。绕过 DEP 的典型方式是使用面向返回编程（ROP）技术。不过，在 ASLR 存在的情况下，ROP 技术是无效的，因为攻击者找不到要重用的代码。所以，找到代码执行漏洞并开发出一个利用漏洞的功能是很困难的。

上面说的只是第一层防护。就算攻击者利用漏洞让恶意代码在进程中执行，iOS 还有许多机制以限制攻击者的破坏范围。例如，代码只能在沙箱中执行。攻击者不能做诸如发送和接收 SMS 消息之类的事情。另外，代码只以低级权限用户“Mobile”的身份运行，而不是根级别的身份。最终的结果是，攻击者不能在设备上安装和运行任何恶意软件或工具程序。攻击者通常想上载键盘嗅探器或其他攻击工具，但代码签名的要求使这些方式不可行。通过这层防护，再加上 iOS 没有命令行和其他公用功能，意味着攻击者只能在被攻击进程中完成所有活动，而不能在设备上持久攻击。

当然，不存在完美的防护。分层的 iOS 模型也曾几次被攻破。首先就是我前面提到的 SMS 攻击。这种攻击的后果是使负责接收 SMS 消息的进程在根级别上运行，而不是在沙箱中。另外一次攻击请见 jailbreakme.com 网站，它将两个漏洞利用程序联合起来。第一个是 Safari 手机浏览器的代码执行漏洞利用程序，这个漏洞利用程序的 payload 包含了第二个漏洞利用程序。而第二个漏洞利用程序使用根级别的权限停用代码签名，然后下载安装真实的 payload，该 payload 能够越狱（破解）移动设备（请见侧栏，了解更多关于越狱的信息）。

安卓

安卓将所有相关应用程序（比如网页浏览器）都放进沙箱中运行，以限制攻击者的破坏。这样，攻击者须要挖掘两个漏洞发起攻击，和前面探讨 iOS 时强调的一样：一是让代码运行，二是突破沙箱环境。能保护安卓免受攻击的另一个原因是许多应用程序是用 Java 写的，这样可以避免内存溢出问题。Collin 和我在安卓中发现了一个 SMS bug，它和导致 iOS 安全模型被破坏的 bug 相似。但由于安卓中的这个 bug 处于 Java 应用程序中，从而无法利用。

然而，安卓的一点很大不足就是没有采用 ASLR 和 DEP 技术。这导致在安卓中挖掘漏洞要比在 iOS 和 Windows Phone 7 上容易得多。后两者都采用了 ASLR 和 DEP 技术。除了安卓，Palm Pre 和黑莓两款手机也没采用 ASLR 和 DEP。iOS 在发布的头年也没有采用 DEP，到了今年才加了 ASLR。希望安卓很快会填补这一不足。

在日常生活中，智能手机已经越来越重要，不使用手机的人越来越少。从安全的角度考量，移动设备没有 PC 机那么开放和暴露，并且还采用了 PC 机所没有的安全策略，比如沙箱和代码签名技术。然而，移动设备存储着私人信息，它们因此成为狂热的攻击目标。不过时至今天，你若丢失私人信息，更可能会是因为把手机忘在了出租车或酒吧中，而不是因为恶意软件或强制下载。

越狱

越狱（破解）技术能冻结代码签名，使 iPhone 能运行非来自 App Store 的应用。但这样会破坏几乎所有 iOS 提供的保护。首要一点是冻结代码签名会导致平台裸露给了恶意软件；另外一个问题是让许多没有签名的应用以根级别权限运行于非沙箱环境中。越狱补丁还会允许内存可写和可执行（这在 iOS 中是不正常的），

一定程度上使数据执行防护技术失效。所以，越狱技术在带来自由的同时也导致了潜在的安全问题。

关于作者

Charlie Miller 是 Accuvant 实验室的计算机安全研究员。他的联系邮箱是 charlie.miller@accuvant.com。

IEEE 安全与隐私协会的首要目标是跟踪和促进安全、隐私及可靠性技术的发展，并将这些发展以有用的方式提供给各领域的专业社区——包括学术研究者和行业从业者。

[1] N. Seriot, “iPhone Privacy,” presentation at Black Hat DC 2010

[2] C. Mulliner and C. Miller, “Fuzzing the Phone in Your Phone,” presentation at Black Hat 2009

[3] “WebOSL: Examples of SMS Delivered Injection Flaws,” Insight, 16 Apr. 2010

[4] R.P. Weinmann, “All Your Baseband Are Belong to Us”

[5] B. Alberts and M. Oldani, “Beating Up on Android”

[6] “CVE 2009-2999” Mitre, 2011

原文链接：http://www.infoq.com/cn/articles/mobile-attacks-and-defense

相关内容：

- 移动设备攻击和防护
- TopGeek：移动互联网时代，选择 Android 还是 iOS？
- 使用.NET/Mono 在移动设备上进行单元测试
- Tabris 1.0：使用 JAVA 进行跨平台移动开发
- 前哈佛教授、Google 工程师 Matt Welsh 谈移动开发趋势
- 专家谈如何从传统开发向移动领域转型

特别专栏 | Column

林仕鼎谈数据中心计算（一）：整体大于部分之和

作者 杨赛

《失控》中提到一个很普遍的现象，就是整体大于部分之和。比如，把 5 只蜜蜂加起来，仍然是 5 只蜜蜂；把 10000 只蜜蜂加起来，可就不仅仅是 10000 只蜜蜂了：它是一个蜂群，具备很多只有蜂群才有（而蜜蜂没有）的特性。

那么，把一万台服务器加起来，我们能得到什么？

在 IDC 时代，这一万台服务器就是一万台服务器，各自做各自的事情，使用各自的计算能力和存储，互相之间的交集无非是抢夺一下网络带宽。

现在，情况已经在转变：我们尝试让这些服务器把资源共享出来，服从统一的管理。一万台服务器现在变成了一个集群，同时它还有另一个名字：云。

> 我现在就是把数据中心当做一台机器来看。

林仕鼎如是说。

从百度基础架构部主任架构师到基础体系首席架构师，林仕鼎对数据中心各层架构的理解和构思，逐渐都投射为百度云的实现。从学术界到工业界的跨界背景，在操作系统内核、存储系统、分布式系统等领域的多重经验，决定了他独特的眼界和架构理念。

2013 年年初，林仕鼎公开介绍了百度在南京的数据中心，并提到了百度定制的 SSD 和将在今年取代 BDDB 的新型存储系统 CCDB。在微博中，林仕鼎说道：

> 下一个我要隆重介绍的神器是百度 SSD，这是我一直鼓吹的 Datacenter Computing 领域中 application-driven, software-defined 设计理念的典型实践。

根据林仕鼎的介绍，百度 SSD 的性能达到 PCIe Flash 的两倍，SATA SSD 的六倍。这是如何做到的？这个定制 SSD 搭配 CCDB 的设计理念跟以前的存储思路有何不同？这会对其他底层软件的设计思路产生怎样的影响？

带着这些问题，InfoQ 编辑与林仕鼎先生进行了一次深度交流。在这次将近两个小时的对话当中，仕鼎谈到了基于 PC 的设备和基于数据中心的设备的根本性不

同，设计针对数据中心的底层软件都需要考虑到哪些方面，百度在软件定义硬件的其他方向的一些工作，以及关于 12306 的一些看法。在未来的一段时间内，InfoQ 会将这次对话的内容分批整理、发布。

今天的主题：为什么要针对数据中心设计一套与以前不同的存储系统？

	桌面/嵌入式/移动系统	数据中心
使用场景	单用户多任务	多用户单任务
物理并行度	少	极多
容错需求	本机处理容错	多机处理容错

正如同很多蜜蜂构成蜂群一样，由很多服务器构成的集群，拥有很多不同于单机的特性。但是，在数据中心发展的过程中，单机一直被当做最小单元来处理，正如同蜂群的最小单元是蜜蜂一样——如果把单只蜜蜂再往下分解，会造成无法重组回蜂群的情况。

就自然界而言，这样的处理方式最为灵活，很能满足蜜蜂这个族群的发展需求。但是，自然界的选择有一个最大的问题，那就是效率低下。对于数据中心而言，其实并没有必要把单机作为最小的单元来处理。这就是林仕鼎提出的数据中心计算的核心思想：让蜂群的意志直接操作最基本的组件。

从单机到数据中心，使用场景有何不同？

以前的 SSD 是为桌面系统和嵌入式系统准备的，可以说是单用户多任务的系统，并没有考虑到很多并行性的问题。

林仕鼎举了一个桌面端应用的例子：

> 作为通用操作系统，桌面操作系统可以运行很多种任务，这样的系统对底下 IO 的优化并没有做特别多，因为没有这样的需求。比如 Email 客户端吧，现在没有一个 Email 客户端能够处理几万封邮件的；无论是我现在在 Mac（SSD）上用的 Mail，还是用 Outlook，一旦我邮件超过一万封以上，它就非常慢了。

之所以有这样的设计，有很大的历史原因：

在 PC 时代，软硬件是一个开放的体系，厂商们共生于其中，形成了一套标准，没有哪个厂家能独立构造整套体系。对于设备制造商而言，其设计首先不是考虑性能，而是考虑如何兼容这套标准，怎样更方便的让开发者在上面编程。存储设备面对的是标准的接口，比如文件系统或者嵌入式的 SQLite DB。这层软件不变，下层硬件就只能屈就于它。所以，为了能适配这些软件，SSD 要对外提供单一设备接口，并隐藏细节，比如要先擦除才能写入等等。于是，在优化之后，SSD 拿出了 20~30%的空间来做写缓冲，又使用内存的映射表达到擦除平衡延长寿命。

然而，数据中心对存储的需求是不同的：

在数据中心里，上层的存储系统可以面对很高的并行度，并没有单一设备接口这样的要求；而且类似读写平衡这样的处理，也已经在软件层面上完成了。

数据中心的应用也是不同的：

我们针对数据中心做的应用，通常都是一个专门的应用，这个应用是单任务多用户的，而且用户非常非常多，IOPS 至少都是几万、几十万这个数量级。这就对底层的要求很高。可以说，需求变了。

在数据中心里，存储系统本身已经做了很多的事情：

比如从存储系统过来的写操作，本来就不会对某一个单元写得特别多，它一定是均匀来写的。也就是说，我在上面已经做了写缓冲，也做了读和写的均衡，所以下层就不用做了，我们就可以把这些逻辑移出来。这样一来，SSD 就变得很简单：你只要提供简单直接的存储功能就够了。

这样的存储系统，不再需要有“自我意识”的底层硬件设备：

在存储系统看来，经过一系列的数据分解和组织的机制之后，对于存储设备的要求就是一堆的块设备。它同样也可以不用依赖文件系统。虽然我们可能还在使用文件，但其实都是定长的数据块。然后对于这些块来说，我们其实还可以把块空间直接映射到 SSD 里面去。

所以，我们把 SSD 上的写缓冲、映射表这些设计，这些控制逻辑，全都去掉。这样就把延迟和浪费的因素都去掉了，这就是为什么我们的性价比能够高。硬件其实是一样的，我们只是把它的架构变了。

针对单机和数据中心在并行度方面的不同，又做了什么？

林仕鼎简单介绍了一下以前的文件系统的设计场景：

> 以前，文件系统是针对单个硬盘设计的。这样比较简单，但缺点是没法将多个硬盘的并行度充分发挥出来。以前的数据库系统设计也是一样，希望面对单个硬盘，容量很大而且不会出故障。这就是 RAID 卡的作用，把一台服务器上的多个硬盘整合起来，加上冗余算法，形成一块大硬盘。这么做的好处是上层软件不需要有太复杂的控制逻辑，但坏处是降低了并行度也增大了延迟。

这跟 CPU 的情况其实类似：

> 再好比以前操作系统设计的时候，它面对的就是一块 CPU，后来多 CPU 出现时才改造成 SMP 的架构。但后来到了几十个核的时候，SMP 也不合适了——它的上限基本上在 16 个核左右，多了性能有问题。不过，对于 PC、手机来说，单个设备本身的并行度并没有特别大的改变，所以架构也并不需要特别去调整。

现在到了数据中心里面，量变已经引发了质变：

> 以前这么做是合理的，因为大家协同工作，所以兼容性和标准接口变得重要。在 PC 体系中，需求确定后，软件系统也就确定了，于是硬件只能在标准接口下发展。但数据中心给了我们一个机会，去重新考虑整套软硬件体系的构建，只要能满足最终的应用系统需求，兼容性和既有标准都不再重要。
>
> 此时，我们希望中间的层次越少越好。因为存储系统已经做了大量的数据管理工作，本身也具有操作多个硬件设备的能力，而且也不再依赖文件系统。正是因为存储系统能够直接访问硬件设备，我希望底下的并行度能够更充分的发挥出来，暴露物理细节而不是隐藏起来变成一个设备。
>
> 我们在设计存储系统的时候，已经充分考虑到了这个并行度。比如，一个服务器上可能挂了十几块硬盘，我们能同时调度这十几块硬盘，使它们能很好的并行工作。

针对高并行度的设计，其实林仕鼎在五年前就已经在 BDDB 上进行过实现，并专门为此找到硬件合作伙伴定制了设备：

> 在 2008 年前后，RAID 卡还是服务器的标配，多数存储系统的设计还是集中在数据组织上，对系统架构与硬件资源调度的考虑较少。在我设计 BDDB 时，采用了全异步的架构，可以对多硬盘、多网络和多 CPU 进行调度，RAID

的存在反而成了障碍。但要把 RAID 卡拆掉却非易事，我们花了很长时间才说服厂家给我们设计了一个 12 块硬盘直连的服务器型号。这个新型号现在已经是存储型服务器的标准了，但在刚开始使用时故障率很高，我们的存储系统一开始也不成熟，二者都用了一年多的时间才逐渐稳定下来。另外，这种服务器型号是先在中国开始应用然后再向美国推广的，这在过去是很少有的。

当然，现在的实现可以获得比当时更高的并行度：

我们 SSD 的做法就是充分暴露物理细节，尽可能让每个单元都成为可以被独立调度的设备。极端情况下，甚至每一个颗粒都可以成为一个独立单元——如果总线够宽、通路控制器够便宜的话。当然，具体能并行多少还要看成本和收益的折中。最终选择某个粒度形成最小的单元，这个单元可以独立控制，然后有多少个这样的单元，也就有了多大的并行度。

针对容错的设计又有怎样的不同？

单机的设计理念，一定是要越可靠越好，因为你只有一台机器，坏掉就没了。所以要在里面加很多冗余信息和校验逻辑，这样在出现错误之后还可以恢复回来。

分布式集群的概念出现之前，针对更多存储的冗余设计仍然是一样的思路：

以前的服务器在设计上的很多逻辑是从桌面上过来的，像是 RAID 卡，也是一个单机的思路，在内部做冗余，因为我能够控制的只有这台机器。

到了数据中心时代，还是那句话，量变引起质变。

在数据中心里面，我们已经不是从一个单机的视角去考虑这些事情了。

我们的存储系统，天然是分布式系统。分布式系统假设所有的设备最终都会发生故障，所以它可以容忍任意一台设备出现问题。因为没有什么设备可以做到 100%可靠，我这个数据一定要分布到不同的机器上去——现在的工业标准是三个副本。

当我们把数据的副本放在不同的机器上时，我们以前所有的限制都可以去掉了。而且以前单机上的数据验证逻辑，也可以不要。RAID 卡的容错功能也可以去掉，它带来的修复和管理成本很高，可靠性却不如三个副本。这样，我们可以提升性能，也能再省一些空间出来。

总结

以上，林仕鼎总结了数据中心计算中，在系统设计上跟单机系统的三大区别：使用场景、硬件并行度、以及对容错的处理思路。

从系统的角度来看，整个数据中心就是一台机器。

接下来的对话中，林仕鼎会介绍设计一个数据中心级别的系统时的一个关键概念：分层。

原文链接：http://www.infoq.com/cn/articles/linshiding-on-sddc-design-01

相关内容：

- 林仕鼎谈数据中心计算（三）：学术界与工业界，以及未来展望
- 林仕鼎谈数据中心计算（一）：整体大于部分之和
- 英特尔 IDF 2013 热点回顾：Scorpio，HTML5，开放混合云
- 软件定义的数据中心已经来临
- VMware 的 CloudFoundry 服务获得对 PostgreSQL 支持

开发
推广
运营
变现
开发者的最佳选择
加入百度开发者 获得全流程支持

特别专栏 | Column

林仕鼎谈数据中心计算（二）：存储资源分层

作者 杨赛

在之前的对话中，百度基础体系首席架构师林仕鼎先生提到要将数据中心当做逻辑上的一台机器来设计架构。但是从物理的层面上，一台机器内部的几个 CPU、几个硬盘，跟一个数据中心里面上万个 CPU、上万块硬盘相比，完全是不一样的概念。因此，在为数据中心这台“机器”设计软件——比如存储系统时，对这种差异性的处理至关重要。

> 之所以会从存储系统先开始，是因为数据存储是最根本的需求。不管你做什么，你都需要存储。所以，通常都是先做存储系统，然后才是其他系统。我们做一个大规模存储系统，它就已经在推动整个系统的架构发生了变化。

对于系统架构的变化，前一篇文章中提到最根本的原因就是使用场景的变化：

> 从单用户多任务到多用户单任务，带来了系统架构的变化，然后带来了软件设计理念的变化。那么，数据中心里面使用的这些系统，无论是分布式的文件系统也好，数据库也好——实际上我们也不需要区分它是文件系统还是数据库，它们都是分布式的存储系统——它们整体设计理念是发生了变化的。

在林仕鼎看来，这种软件设计理念的变化包含很多方面，但最基本的一点就是：分层。

> 这个“机器”里面是一个有层次的架构。有些东西是比较紧密的，比如说我这些机器正好在同一个盒子，即同一台物理机器里面，这时候我们的交换是通过内存，而不是通过网络，我们就可以做共享内存，而不是消息。也可能我们在同一个机架里面，那么我们的交换基本可以做到 1：1。也有可能我们分散在不同的机架上，这时候我就不能毫无限制的把信息扔给另一台机器了，因为这时候会有个收敛比。综合考虑这些信息，我就可以根据整个物理的情况来选择一个平衡：成本和软件复杂度的平衡。

“所以，”林仕鼎补充到，“因为不同层下的限制不同，你的软件本身必须要能够去适应。”

CCDB 是林仕鼎为百度设计的第二套存储系统。当前在生产环境下使用的 BDDB 是 2008 年的设计，从 2009 年开始用于百度网页的存储。BDDB 实际上已经是按照分布式的思想去设计的，但它还是把一台机器当做一台机器来看。

> 一个存储系统有三个层：block 层，stream 层，以及 table 层。CCDB 的设计是，我把这三个层放在一块儿，放在一台机器上面去。然后，机器和机器之间连起来，就把每个层直接扩展到多机了。这样，你由软件上的层次，映射到物理机器的架构上的时候，就可以尝试多种不同的设计方案出来。就像我刚才说的，我们有同一台机器里面通过内存交换的架构，有同一个交换机下 1：1 交换的架构，有跨交换机的、有一定收敛比的架构。我们需要选择合适的单元映射到合适的硬件上。不同的映射方式会带来很多种不同的变化。

除了物理结构上的层次和存储软件本身的层次之外，这个分层在单机的存储设备上也有所体现：

> 我们看到存储，有硬盘，有 SSD，有内存。所以还有一个需要考虑的因素就是，我们需要把什么东西分别放在这不同的介质上面？

林仕鼎介绍了现在希捷的一个做法：把 SSD 和机械硬盘打包，做成混合硬盘：

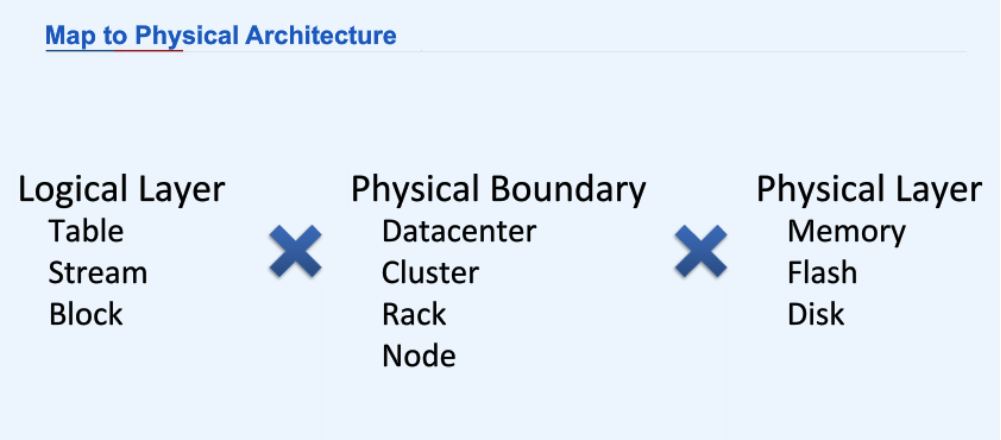
> 它在一个盘里同时有内存这样的 cache，有 SSD 的 cache，而它把这些内存、SSD、硬盘都给隐藏了起来，做成一个设备。

对百度的应用场景而言，这肯定又是不可取的，因为单机做了多余的事情：

> 数据中心里面，我们本身就把这两个分开来考虑。你把它并到一块，反而会限制它的范围，导致你这边数据的 cache 只能放到这块盘里面。而如果我把它分来的话，我就可以跨多机建一个逻辑上的 SSD 层。在物理上，我在这块盘的 SSD 上 cache 的东西，可以弄到另一块盘上去。

所以，CCDB 的做法是这样的：把 block 层存在硬盘上，把 stream 层放在 SSD 上（这层是可选的），而内存则用于做写缓冲。

于是，就变成这样：



（来自林仕鼎在深圳的分享资料）

将多机资源合并成逻辑上的一个层，一个最直接的好处就是可以实现另一种意义上的高可靠性：

> 因为我们把多机的内存放在一块儿，这就意味着同一份数据我可以写到不同的内存上面去。这种实现的可靠性级别和我从内存写入硬盘的可靠性级别是不同的，但很多情况下，我有这种多机内存级的可靠性级别就够了，不一定要到硬盘上去，那样性能成本太高了。
>
> 我们一旦把整个设计的理念从软件这个层面来考虑的话，那么在设计上存在的可能性，以及架构层面的考虑，是完全不一样的，因为你什么都可以去做，而且可以很灵活的去调整。

原文链接：http://www.infoq.com/cn/articles/linshiding-on-sddc-design-02

相关内容：

- 林仕鼎谈数据中心计算（三）：学术界与工业界，以及未来展望
- 林仕鼎谈数据中心计算（一）：整体大于部分之和
- 英特尔 IDF 2013 热点回顾：Scorpio，HTML5，开放混合云
- 软件定义的数据中心已经来临
- VMware 的 CloudFoundry 服务获得对 PostgreSQL 支持

特别专栏 | Column

林仕鼎谈数据中心计算（三）：学术界与工业界，以及未来展望

作者 杨赛

在《林仕鼎谈数据中心计算》的第一部分，百度基础体系首席架构师林仕鼎先生提到要将数据中心当做逻辑上的一台机器来设计架构。在之后的对话中，林仕鼎具体介绍了对数据中心的存储资源做逻辑分层、物理映射的思路。

在接下来的对话中，林仕鼎会介绍他对于“为什么以前的系统做的并不是很好”的理解，以及“为什么会想要去做这个系统”的原因。此外，对话中也会涉及到他对12306 系统的部分解读，以及一些相关问题的看法。

InfoQ：像您提到的将数据中心看做一台 PC、逻辑分层这样的设计思路，为什么很少看到其他人来这样做呢？

林仕鼎：

设计一个好的系统架构，不仅仅是为了把系统做出来，更重要的是去考虑这个东西本质上应该是什么样的，怎样才会更优雅、更具有普遍性的推广意义。而且，数据中心里的系统涉及到应用、软件系统以及硬件架构等，需要多个团队配合，也需要产业链上合作伙伴的配合。我们可以提出思路并给出设计方案，但这些方案的实施需要业界的参与。这是一次产业的变革，我们很希望能在技术上起一些引领作用。

这同样也是现代 IT 技术演进的特点。创新，包括基础技术的创新，更多由工业界，特别是大型互联网公司来引导。因为，只有在大型互联网公司里才有需求有条件去做这些创新，真实的海量的数据和用户请求、实际运行的大规模平台，这些都是高校和研究机构所不具备的。

当然，也不是有需求有条件就必然带来创新，跟环境氛围、个人意愿和能力也有很大的关系。就我个人来说，我首先是一个工程师，一向 hands-on，一行一行代码写起，做过很多套系统；其次，我也有多年的系统研究背景，对这个领域的方法论、发展历史和现状比较熟悉。也就是说，我了解问题所在，有条件有能力

去解决，并且也有意愿去定义一套完整的技术体系和理论框架。这个情况在业界是不太多的。

互联网公司里，多数工程师们在使用开源技术，这对于快速完成项目有很大帮助。在长期的实践过程中，大家也积累了很多经验，但这只是应用级开发经验，因为很多时候他们对这些技术的了解仅止于应用，缺乏对设计思路和实现细节的深究。在没有合适开源技术或者需要从头构建一个新的系统时，这些应用级开发能力就不足以胜任了。当然，深入了解这些开源技术也只是开始而已，说实话，大多数开源项目的设计和实现的水平并不太高。

学术界也有其固有的问题，从逻辑上的设计思路到物理上的系统架构和代码，还有很大的鸿沟存在。如我起先说的，一个复杂的系统在逻辑上有很多个层次，具体映射到模块、介质、机器、机群等物理结构上时，有诸多因素需要考虑，而这些因素有时候要到具体实施时才能被确定。另一方面，系统的实现难度也是很高的，怎么把硬件的并行度发挥出来，怎么让系统去异步地调度资源，这些都很复杂，需要大量的开发经验。

另外，系统研究产生的通常是原型系统，这跟实际的生产系统有两点根本性的不同。一个是规模，研究系统一般跑在几十台机器上，这已经很大了。把几十台机器当作一台机器来看，和把几千台机器当作一台机器来看，完全是不一样的。我经常开玩笑说，这会影响一个工程师的世界观，如果没有实际做过这样的大系统，根本体会不到其中的区别。比如刚才说的层次结构，在几十台机器的时候，根本就不明显。对于故障的处理方法，对于自动化工具和机制的重视程度也会完全不同。第二个不同在于，生产系统面临的压力是不一样的，用户请求的类型也完全不同。刚来百度的时候，我做了第一个存储系统，实现原型只用了半年，但是花了半年的时间才让它稳定运行，再花半年的时间才结束试运行，然后上线成为一个真正的生产系统。也就是说，从设计到实现的过程只是 1/3，把它变成真正可用的系统还要再花两倍的时间。

InfoQ：从原型到真正可用的系统，最大的难点在哪里？

林仕鼎：

我之前也说过一些，在讨论 12306 的时候。一个系统最复杂的地方不在于取得高性能，而是实现大压力下的稳定性。比如说，一个系统在运行时，有人有 scan 的需求，有人要写，有人要读。Scan、写和读，这些操作的资源消耗量的差异很大。而且这些操作是并发产生的，读写需要并行，随机读写和 scan 需要并行，

scan 和 scan 也需要并行。在最糟糕的时候，比如几十个大的 scan 请求同时进来，你不能让资源被它消耗光，需要留一些出来去响应写和读。

同样是 scan，也有其他的问题要考虑。为了使 scan 更快，需要为每个 scan 准备更多的缓冲。在有很多 scan 请求的情况下，系统资源不足，只能大家共享这些缓冲块。这时候有两个问题，第一，如何在大量请求的情况下共享资源。很多系统的处理方案是限制请求数量，比如只允许 10 个并发请求，其他请求在队列里等待着。为了保险起见，要根据每个请求可能的最大资源消耗量来限制并发请求数量，通常这会导致资源的浪费。第二，每个 scan 接收方的能力或者带宽都不同，如果多数 scan 接收方很慢，它们占着系统资源，这很可能导致快的接收方也被拖慢。

比如某系统每秒只能发 100MB 的数据，有两个请求进来，请求 A 每秒只能接收 1MB，而 B 每秒能接收 200MB。有时候可能会出现这种情况，系统把 100MB 全给了 B，而 A 这边一点都拿不到。还有另一种情况是，A 收 1MB，B 也只能收 1MB，也就是说，最慢的那个出口拖慢了整个系统。这些输出情况已经算是不错的了，有很多系统在这种负载下甚至都不能正常工作。

好的处理方式应该是这样的，在有两个请求的时候，系统资源先分成一半一半进行输出。A 这边其实资源有富余，他只能收 1MB，50MB 的输出他是接收不了的，我们可以把多出来的 49MB 分给 B。也就是说，先平均分配资源，然后对消耗不了的多余资源进行再次分配，这是一个兼顾公平和效率的分配策略——所有的请求都能得到响应，如果资源有富余，可以分配给有更强能力的接收方。当然，要实现这一点，需要有很多机制的支持。比如需要反馈机制，这样才能知道 A 的接收能力不足。也需要有细粒度的资源分配方式，这样才能把多余的预分配资源回收。

刚才说的这个情况比较简单，请求量大了之后要实现这个规则就复杂了。而且，不同类型的请求对资源的消耗量是完全不一样的。比如 scan 消耗的资源可能是一次随机查询的几万倍，持续的时间也更长。假如有 100 个随机请求进来，然后又有 100 个 scan 请求进来，这时候我要怎样分配资源，让大家都觉得可以接受？这就是类似 12306 的系统遇到的最根本的问题。有些操作的资源消耗是其他操作的几十倍或者几百倍，而并发操作的总的资源消耗量是超过极限的。这种情况下，需要合理的资源调度策略来保证系统的稳定输出，而简单的排队和并发数限制策略的效果并不够好，因为一个请求的资源消耗量并不是事先能够确定，并发数少了导致资源浪费，而多了又将带来资源消耗过多导致系统崩溃的风险。

顺便提一下高性能系统，这个一直没有过合适的定义，我的看法是一个能够按硬件所能提供的最大能力持续稳定输出的系统才是高性能系统。

InfoQ：那么，即使百度、淘宝来做 12306，也会遇到这个问题么？

林仕鼎：

12306 所遇到的问题，说实话，要完美高效地解决还是不容易的。淘宝交易系统的压力其实并不高，系统的压力主要在搜索和图片上。12306 也是如此，交易量不大，但查询量很大，而它的服务器规模也不可能有百度淘宝这么大。另外，它有一个更大的问题，就是后台要跟铁路内部的 ERP 类系统连接，那个系统的部件是很慢的。如果给这个很慢的部件太大压力，很可能拖慢整个系统并使其崩溃。

一个系统里面通常由很多部件组成，每一个部件都在输入输出，也具有不同的服务能力。有些部件每秒可以支持 100 个并发，而有些部件每秒可能只能支持 1 个并发，你给他 10 个并发，它就崩溃并拖垮全系统。要达到高性能，需要让每个部件都有足够的并发量；而要稳定输出，则需要让每个部件都工作在它的能力范围之内。所以，在每对上下游的部件之间，都需要有队列来控制压力。但这个控制机制不能基于固定的数量，要根据现有的空闲资源量进行动态反馈调节。同时，对于资源，预分配可以提高效率，但会带来 live-lock，需要有一套细粒度的分配机制，在效率和可回收之间取得平衡。

InfoQ：不过话说回来，2013 年的 12306 事实上可用性已经比之前一年提高很多了，虽然买到票仍然很难，但至少不会总出现页面刷不出来的问题。

林仕鼎：

我猜测今年他们做了一个事情，把 transaction 和 query 分开了。transaction 的压力其实不大，毕竟票量也就那么多。Query 不需要保证严格的一致性，再多用一些 cache，性能就提升了。

（InfoQ 编辑注：林仕鼎在自己的博客上写过三篇文章，专门评论火车票系统架构的设计。具体请见：简单讨论火车票系统后面的架构设计，再谈谈火车票系统，三谈火车票系统）

InfoQ：我注意到，您一直在用存储系统的例子在讨论 12306，其实这跟火车票本身的业务逻辑没有关系。我感觉这是一个通用的系统架构的设计问题，对于工程师来说，要怎么学习才能对这些系统架构设计有了解呢？

林仕鼎：

是这样的，不同系统有业务逻辑带来的固有差异，但却具有更多的作为系统的共同点。还是以存储系统为例，在我看来，做一个存储系统有三部分工作：第一是存储系统本身对数据的组织，第二它是个分布式系统，第三它是个操作系统。

第一部分就是存储系统自身的业务逻辑，数据的组织就是怎么去做文件系统，怎么去做对象系统，怎么去做数据库。这些涉及到的问题是，数据在介质上怎么排列，索引该怎么建，修改的逻辑应该是什么样子——比如是原地修改，还是在后面追加然后合并等等。

分布式系统要解决的就是数据怎么分布，如何容错的问题，简而言之就是 partition 和 replication。而操作系统要解决的问题实际上就是如何在面临压力的时候合理的调度资源，如何去实现硬件所能提供的最大资源输出，同时保持稳定。现在很多存储系统的设计或实现，其实那只是在第一个层面的工作，在其他两个层面并没有深入。真正做一个完整的存储系统是非常难的，这也是很多项目仿照存储系统论文的设计，依葫芦画瓢，却没有取得很好效果的原因。

火车票系统也是如此，卖票的业务逻辑只是其中的一部分，它同样也有需要分布式系统和操作系统的支持。

说到学习，我个人的看法是要不断设计并搭建实际系统，同时注意方法论和设计理念的学习和提炼。这好比一个武林高手，内力、招式和实战经验缺一不可。编程能力就是内力，需要勤加修炼，不断对代码进行重构。实战经验通过战斗获得，也就是需要实际搭建系统。而招式呢，则通常是前人积累下来的，来自于拳谱或剑法，也就是所谓的武功秘笈。我总结过一篇学习材料，可作为入门功法。

内力、招式和实战经验，这三者通常不是单独成长的，只有在一次次的迭代过程中才能得到提升，即在实战中锻炼内力并加深对招式的理解，而后两者可以带来更多实战的机会和成功概率。

InfoQ：百度用 ARM 做存储服务器，您对冷数据和热数据的存储有什么建议？比如说网盘（共享居多）服务，需要定制什么样的存储服务器好？在线备份（写多读少）服务怎样定制比较好？又比如对 IOPS 要求高的数据库存储服务怎样定制比较好？

林仕鼎：

我们希望硬件比较统一，而不是有很多种的硬件。对于不同的需求，尽可能通过软件来处理这种不同。我做 CCDB 的出发点也是希望能用一套统一的存储技术体系来适配不同的存储需求。之前提到了存储系统逻辑上的三个层次，可以按不同情况映射到不同的硬件介质上。比如说你的场景是写多读少，那就不引入 Flash 这层，直接使用硬盘。具体实现是在部署的时候，你会有个配置，配置上你定义它没有 Flash 只有硬盘，或者调节 Flash 和硬盘的比例。

于是，在合理地定义软件系统架构之后，我们把需求的不同转化成了配置问题。在硬件模型上，趋向于归一化到一个标准架构，通过调节硬件参数，配合软件系统满足应用的不同需求。

（InfoQ 编辑注：目前阿里、腾讯、新浪等，也都在进行硬件机型标准化的工作，需求类似的服务器硬件机型很多都在进行合并。虽然可能会造成闲置资源的浪费，但由于批量采购统一的机型更加便宜，整体成本还是有所降低的。）

InfoQ：百度目前在 SDN 这方面采用的什么方案，定制的交换机么？有哪些进一步的打算？

林仕鼎：

在很多材料中我都提到，我们对数据中心系统的设计思路是，application-driven 和 software-defined。网络是数据中心的一部分，我们同样希望它越简单越好，通过应用和软件系统的合理设计，可以使其变得更简单。

以前有人问我，我们的一个数据中心有多大。我的理念是，一个紧耦合的逻辑数据中心应该尽量限制在万台以内，这里的收敛比最好是 1:1，而数据中心之间的互联带宽和延迟可以低一个等级。上层应用如果需要超过万台的规模，则自己按照数据中心去切分，把无需紧密相连的模块部署到不同的数据中心上。在这个前提假设下，我们数据中心的内部互联和相互连接都可以做得比较简单。

我个人的看法是，SDN 更适合于拓扑结构需要经常变化的复杂路由情况，比如广域网。我们现在也在做一些 SDN 的研究，比如通过公网搭建 overlay 降低数据中心互联的成本并提高可靠性。

InfoQ：之前听说百度的负载均衡系统也是自主研发的，基于 64 核处理器，可为业务提供最大 320G 的负载均衡。另外之前听杜熙介绍过，BAE 这块的路由算法也是带智能的随机路由，会跳过繁忙或故障的后端节点。能简单介绍一下百度负载均衡系统的设计思路吗？

林仕鼎：

百度外网接入的负载均衡系统是我们自主研发的。BAE 的路由算法相对来说简单一些，更多考虑故障处理的问题。负载均衡算法其实很简单，把请求轮流分配到某几台机子上面去，在出问题时跳过故障机器。只要能知道目标机器的状态，这个实现并不难。

InfoQ：通过软件定义硬件后，将会出现很多一体机，是否会出现各家做各家的，结果出现标准不统一，对软硬件发展不利呢？或者会出现另外新标准？百度是否会出新标准呢，像以前提的 ARM 存储服务器一样。

林仕鼎：

我倒不觉得会在大型数据中心里出现很多一体机，一体机可能更多按终端的方式销售到企业里。在软件定义数据中心之后，整个数据中心成为一台机器，原先的节点成为一个部件。软件需根据整个数据中心的架构来开发，如果有一体机，那也是整个数据中心的系统。

我们对硬件的要求很简单，硬件的设计原则最好是简单、傻（dummy），同时对外暴露所有可编程的接口（programmable），复杂的控制逻辑由软件实现，而且在全系统规模进行考虑。

新标准我们也在考虑，不仅仅是 ARM 服务器，还有机架。我们说可编程，不只是服务器可编程，服务器周边的东西也需要可编程，这包括机架位置，包括主板温度、风扇转速等等，这些都需要可编程，也需要可以收集数据。

InfoQ：去年，Frank Frankovsky 在 Open Compute 的网站上提到他们的 Open Rack 和百度、腾讯设计的 Project Scorpio 这两个规范计划在 2013 年合并。这个合并的计划进展是否顺利？

林仕鼎：

合并的计划我不是很清楚。

如起先的介绍，其实我们内部做的事情比起 Open Rack 会更多一些。未来我们也希望能够将这些思路和实现方案分享出去，多做一些开源方案，包括软硬件架构，让大家了解我们在做些什么，也帮助业界共同进步。

原文链接：http://www.infoq.com/cn/articles/linshiding-on-sddc-design-03

相关内容：

- 林仕鼎谈数据中心计算（一）：整体大于部分之和
- 英特尔 IDF 2013 热点回顾：Scorpio，HTML5，开放混合云
- 软件定义的数据中心已经来临
- VMware 的 CloudFoundry 服务获得对 PostgreSQL 支持

InfoQ
Enterprise Software Development Community

新品推荐 | Product

Blossom 准备将开发语言切换为 Dart

作者 Zef Hemel ，译者 臧秀涛

Dart 是 Google 设计的新的 Web 编程语言和平台，可用于替代 JavaScript。Thomas Schranz 在一篇博客中宣布，他们公司准备将其产品 Blossom（面向产品团队的 Web 看板系统）迁移到 Dart。

原文链接：http://www.infoq.com/cn/news/2013/05/blossom-dart-switch

ScriptCS：将 C#转变为脚本语言

作者 Abel Avram ，译者 李彬

ScriptCS 支持开发者使用简单的文本编辑器编写 C#应用，并通过 Roslyn 执行编译，使用 NuGet 进行包管理。

原文链接：http://www.infoq.com/cn/news/2013/05/ScriptCS

Go 1.1 正式发布，性能显著改善、语言细微变化

作者 崔康

最近，Go 1.1 正式发布，相比 1.0，Go 1.1 包含了许多改进。最重要的改进是性能。优化的地方包括编译器、链接器、垃圾回收器、goroutine 调度、map 实现和部分标准库等等。

原文链接：http://www.infoq.com/cn/news/2013/05/go-lang-release

JSON.NET 5 中的架构变更

作者 Jonathan Allen ，译者 李彬

Newtonsoft 对 JSON.NET 进行了两项架构升级，以使其能够更容易地在涉及第三方库的项目中使用。这两项特性是默认设置和扩展数据。

原文链接：http://www.infoq.com/cn/news/2013/05/json-net-5r5

Clang 完整支持 C++11 标准

作者 Jeff Martin ，译者 臧秀涛

希望使用 C++11 标准所有特性的开发者可以关注一下 Clang 编译器。该项目已经完成了对 C++11 标准的完整支持。

原文链接：http://www.infoq.com/cn/news/2013/05/clang_CPP11

Microsoft 企业库 6.0 增加语义日志

作者 Abel Avram ，译者 孙镜涛

Microsoft发布了企业库6.0 ,该版本新增了一个语义日志应用程序块 ,支持Unity 3.0 和 Windows 商店应用 ,同时还简化了使用方式 ,有更好的文档和整体体验。

原文链接：
http://www.infoq.com/cn/news/2013/05/Microsoft-Enterprise-Library-6

Dart 编译器在浏览器中编译 Dart 代码

作者 Zef Hemel ，译者 孙镜涛

Google 重新启动了 try.dartlang.org 在线实验室，用户不需要下载并安装 Dart SDK 就能够尝试 Dart 。和前一版不同的是，新版本会在浏览器中将 Dart 代码编译成 JavaScript，这意味着它可以离线使用。此外，它还支持 dart:html——一个用于 HTML DOM 操作的 Dart 库。

原文链接：http://www.infoq.com/cn/news/2013/05/try-dart

Visual Studio 2012 Update 3 预览版发布

作者 Jeff Martin ，译者 廖煜嵘

Visual Studio Update 3 的第一个预览版本已经发布。本次更新主要集中在 Team Foundation Server 2012 的 Bug 修复和功能改进方面。

原文链接：http://www.infoq.com/cn/news/2013/05/vs2012_3

Android Studio：Google 发布基于 IntelliJ IDEA 的新 IDE

作者 Abel Avram ，译者 臧秀涛

Google 基于 IntelliJ IDEA 社区版发起了另一个 Android IDE，它与基于 Eclipse 的 IDE 有很多相同之处。

原文链接：http://www.infoq.com/cn/news/2013/05/Android-Studio

Strongloop Node：第一个企业级的 Node.js 发布版本

作者 Zef Hemel ，译者 臧秀涛

雇佣了一些 Node.js 核心贡献者的 Strongloop 公司最近发布了 Strongloop Node 1.0 GA，这是该公司面向企业级应用并提供商业支持的第一个 Node.js 版本。该版本基于 Node.js v0.10.7，还包含了 Express、Connect 和 Passport 等流行的 npm 模块。

原文链接：http://www.infoq.com/cn/news/2013/05/strongloop-node

AIDE 2.0 引入对原生 C/C++应用的支持

作者 Abel Avram ，译者 臧秀涛

AIDE 是一款运行在 Android 上的 Android IDE。AIDE 2.0 添加了对 C/C++的支持，加入了新的设计视图和更好的 UI，并改进了与 Git 的集成。

原文链接：http://www.infoq.com/cn/news/2013/05/AIDE-2-C

10gen 发布 MongoDB 增量备份服务

作者 Roopesh Shenoy ，译者 李彬

10gen 最近宣布限量发行其 MongoDB 备份服务，该服务提供了增量备份和指定时间点的恢复功能。

原文链接：http://www.infoq.com/cn/news/2013/05/mongodb-backup-service

最新的 Dart VM 在 DeltaBlue 基准测试中击败 JVM

作者 Abel Avram ，译者 李彬

前不久 InfoQ 曾经报道过，在 DeltaBlue 基准测试中，Dart2js 生成的 JavaScript 代码已经优于手写的 JavaScript 代码，最近，还是基于该基准测试，最新的 Dart VM 又击败了 JVM7 的 64 位服务器版本及其他版本。

原文链接：http://www.infoq.com/cn/news/2013/05/Dart-Java-DeltaBlue

CloudMunch 发布了一体化 DevOps 平台

作者 Grischa Ekart ，译者 李彬

CloudMunch 发布了 DevOps 平台——该平台预先无缝集成了版本控制、构建管理、验证、自动化测试、部署和云连接器等工具，并在此之上提供了控制面板。该公司宣称该平台极大地简化了应用和基础架构的部署。

原文链接：http://www.infoq.com/cn/news/2013/05/CloudMunch-launch

Appium 支持测试原生、混合及 Web Android 应用

作者 Abel Avram ，译者 臧秀涛

Sauce Labs 扩展了 Appium，支持对 Android 应用和 Firefox OS 应用进行自动化测试。今年年初又引入了对 iOS 的支持。

原文链接：http://www.infoq.com/cn/news/2013/05/appium-android

推荐编辑 | 翻译团队编辑 李彬

大家好，我叫李彬，虽然在工作上已经转型为产品经理的角色，但内心依旧保留着对技术宅生活的眷恋。我也非常荣幸能够受邀成为本期《架构师》的推荐编辑。

很早以前，我就习惯于访问 InfoQ，了解业界各项动态，但走入 InfoQ 的社区却是件非常偶然的事情，说起来这也要得益于社交媒体的发展——某次在微博上看到 InfoQ 招聘，附带对 InfoQ 介绍的长文却是发布在我常玩的 Lofter 上，恰好我也好奇 InfoQ 一篇篇文字背后的面孔和环境到底是怎样的，于是打开链接看到了霍泰稳老大的一篇日志，并对那篇文章提了一些建议，不想老大回复问我是否也有兴趣参与到社区的活动中，于是接下来的事情自然便是我藉此更深一步的走入了 InfoQ 的世界。

在这个年代，随着产业的发展，产品经理不再停留在通过沟通技能确保推动产品开发进展的定位上，而是对设计、UI/UE、技术原理和部分技术实现等多方面都存在要求的角色。而参与 InfoQ 的技术新闻/文章翻译，不仅仅给了我一个机会能够充分利用业余时间去紧跟业界的动态，也对自己提出了更高的要求——很多时候我们阅读一篇文字，往往知其大意、抓住兴趣点即可；而提笔撰写或是翻译，则要尽可能准确的理解其中所述的各个细节，以尽量避免纰漏、“咬文嚼字”。对于已经很大程度上脱离开发环境的我来说，这无疑又会促使自己去重温一些技术方面的知识和技能，同时也要摆脱惰性更努力地接触新鲜事物。

“促进软件开发领域知识和创新的传播”是 InfoQ 的宗旨，我想 InfoQ 团队的同仁们也是在这一口号的感召下发挥着自己的热情，同样也是参与各个开源社区并为自由软件发展、知识传播和技术分享做贡献的人们的动力吧。

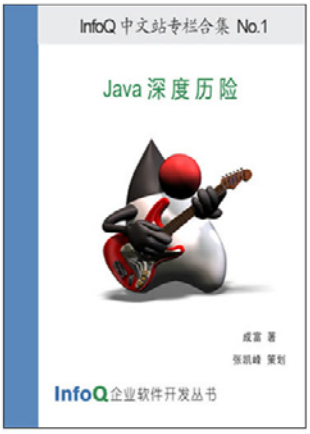
最后，还想告诉大家一个重要消息：加入 InfoQ 团队，能够免费参加每年的 QCon，在 QCon 北京 2013 那段时间里，这可是最令我兴奋的福利。QCon 是一场技术的盛会，也是结识业界同仁、增进相互交流的机会。

如果你有意成为 InfoQ 团队的一员，请邮件至：

editors@cn.infoq.com；而如果大家希望跟我进行更多的交流，可以通过新浪微博@neverlive。

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

封面植物

百合

百合是百合科百合属多年生草本球根植物，主要分布在亚洲东部、欧洲、北美洲等北半球温带地区。全球已发现有百多个品种，中国是其最主要的起源地，原产五十多种，是百合属植物自然分布中心。近年有不少经过人工杂交而产生的新品种，如亚洲百合、麝香百合、香水百合等。百合的主要应用价值在于观赏，有些品种可作为蔬菜食用和药用。百合的主要应用价值在于观赏，其球茎含丰富淀粉质，部分品种可作为蔬菜食用。以食用价值著称于世的中国兰州百合，最早记载在甘肃省平凉县志中，迄今已有 450 多年。兰州七里河等地区广泛栽种食用百合，在国内外享有很高声誉。兰州百合个大、味甜，既可作点心，又可作菜肴，宜兴的卷丹制成百合汤是夏日消暑佳品。百合还可制作成百合干、百合粉，在国际市场上价格很高。到目前为止，百合仍然是中药中的常用药材。中医认为百合性微寒平，具有清火、润肺、安神的功效，其花、鳞状茎均可入药，是一种药食兼用的花卉。百合花素有“云裳仙子”之称。由于其外表高雅纯洁，天主教以百合花为玛利亚的象征；而梵蒂冈以百合花象征民族独立、经济繁荣，并把它做为国花。百合的鳞茎由鳞片抱合而成，有“百年好合”、“百事合意”之意，中国人自古视为婚礼必不可少的吉祥花卉。百合在插花造型中可做焦点花，骨架花。它属于特殊型花材。产地及分布：中国、日本、北美和欧洲等温带地区。百合花花姿雅致，叶片青翠娟秀，茎干亭亭玉立，是名贵的切花新秀。**生长习性：**性喜湿润、光照、要求肥沃、富含腐殖质、土层深厚、排水性极为良好的砂质土壤，最忌硬粘土；多数品种宜在微酸性至中性土壤中生长，土壤 pH 值为 5.5～6.5。百合喜凉爽潮湿环境，日光充足的地方、略荫蔽的环境对百合更为适合。忌干旱、忌酷暑，它的耐寒性稍差些。百合生长、开花温度为 16～24℃，低于 5℃或高于 30℃生长几乎停止，10℃以上植株才正常生长，超过 25℃时生长又停滞，如果冬季夜间温度低于 5℃持续 5～7 天，花芽分化、花蕾发育会受到严重影响，推迟开花甚至盲花、花裂。

JavaOne™
2013 年 7 月 22 - 25 日 | 上海
立即注册
7月21日前注册
节省500元
注册网址: oracle.com/cn/javaone2013
ORACLE®
版权所有 © 2013，甲骨文和/或其关联公司。保留所有权利。Oracle 和 Java 是甲骨文和/或其关联公司的注册商标。

架构师 6 月刊

每月 8 日出版

本期主编：杨赛

美术/流程编辑：水羽哲

发行人：霍泰稳

读者反馈：editors@cn.infoq.com

投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博 :http://weibo.com/infoqchina

商务合作：sales@cn.infoq.com　15810407783

本期主编：杨赛，InfoQ 中文站主编

对软件开发、系统运维、互联网产品、移动技术、项目管理等方向感兴趣。写过一点 Flash 和前端，现在只是个伪码农。在 51CTO 做过三年多，创办了《Linux 运维趋势》电子杂志，也做过一些失败的互联网项目。最近关注如何做好的采访，以及中文界技术作者生态圈的发展。曾在新加坡、英国求学，在大学读的物理，在天幻网做过一些专题和游戏汉化，练过点长拳，玩过足球、篮球、羽毛球等各类运动和若干乐器。KK 的粉丝。

《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。